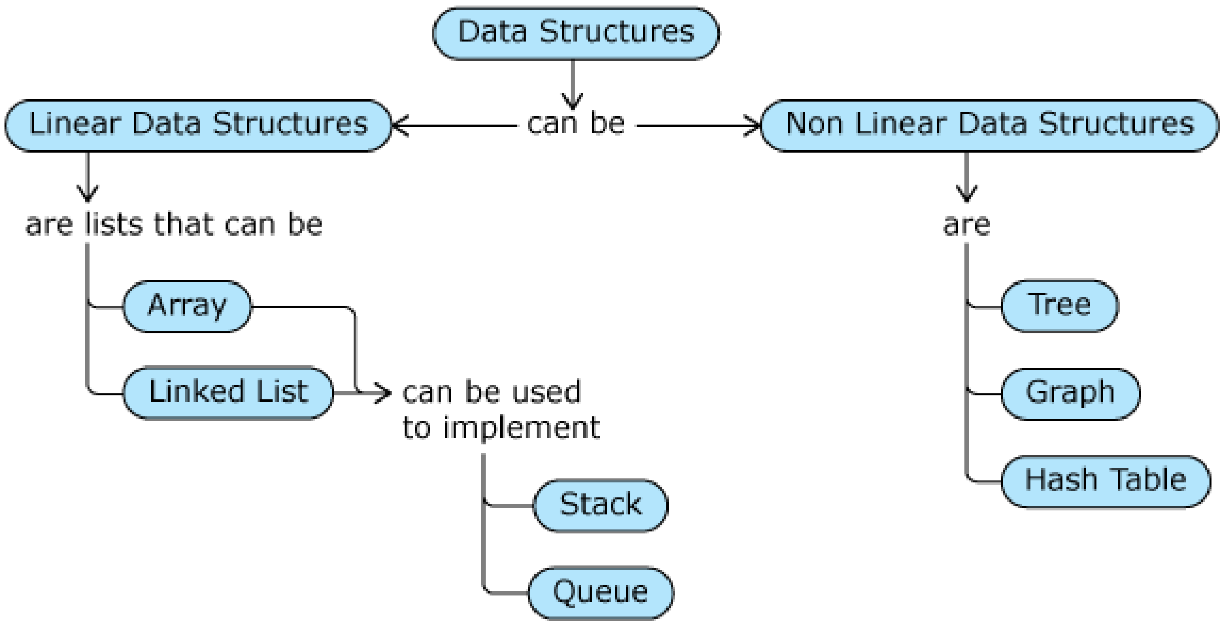
#InfyTQ Data Structures Using Python PART- 1 Notes: -

Let’s Start with the Example that why we use Data Structures

Data Structures: - **Data structures** represent the way of arranging data in computer so that it can be accessed and used efficiently. It is all about grouping and storing collections of data in memory, operations that can be performed on these collections and algorithms that can be used for these operations.

There are different types of data structures like list, stack, queue etc. Each of them is suitable for specific type of tasks.

ONLINE LEARNING



ONLINE LEARNING

# Basic Terminology of Data Structures:-

1. Data
2. Group Items
3. Record
4. File

#Need of Data Structures: -

1. Processor Speed
2. Data Search
3. Multiple Request

#Advantages: -

1. Efficiency
2. Reusability

#Operations:-

1. Insert
2. Delete
3. Searching
4. Traversing
5. Sorting
6. Merge

# List PART-1: -

ONLINE LEARNING

:- List is a group of different types of elements.

Following are the features of List:-

1. Linear Data Structures.
2. Used to store sequence of values.
3. Grow and Shrink according to need.

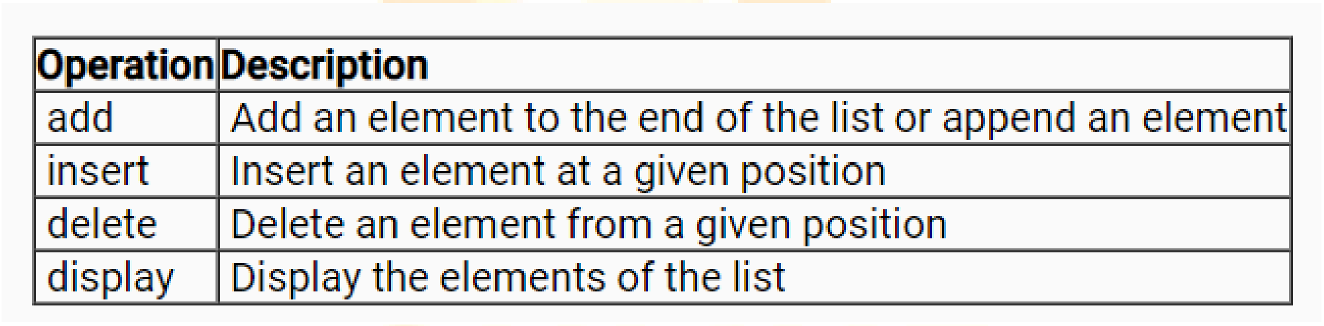
Ex: - Preparing daily routine

Array is a data type which is fixed capacity and can store a collection of elements. However, we can use it to implement list data structure.

ONLINE LEARNING

For Implementation we will be using list data type in python which is internally dynamic array which can grow and shrink based on the elements added or removed from it.

# Common operations on list are:

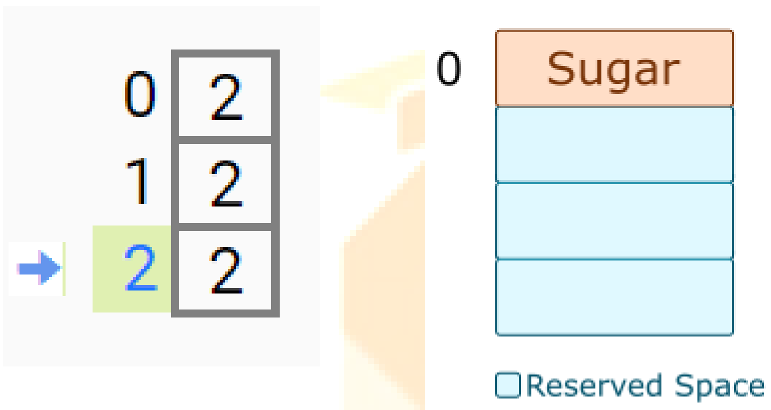


Operation	Description
add	Add an element to the end of the list or append an element
insert	Insert an element at a given position
delete	Delete an element from a given position
display	Display the elements of the list

ONLINE LEARNING

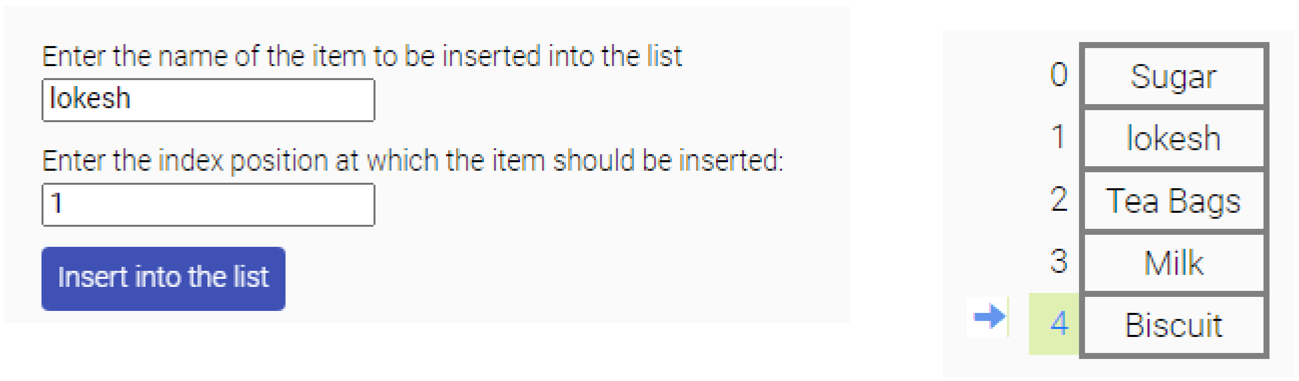
1. List using Array (Add Operation): -

When an element is added to an empty list in Python, a block of memory is allocated and element is added at index position 0. The remaining memory is considered to be reserved space which will be used later for addition or insertion of elements.



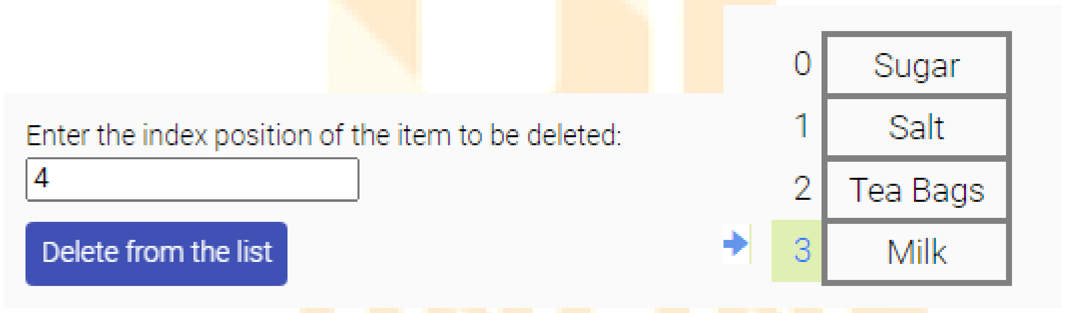
ONLINE LEARNING

2. List using Array (Insert Operation): -


Enter the name of the item to be inserted into the list
lokesh
Enter the index position at which the item should be inserted:
1

0	Sugar
1	lokesh
2	Tea Bags
3	Milk
4	Biscuit

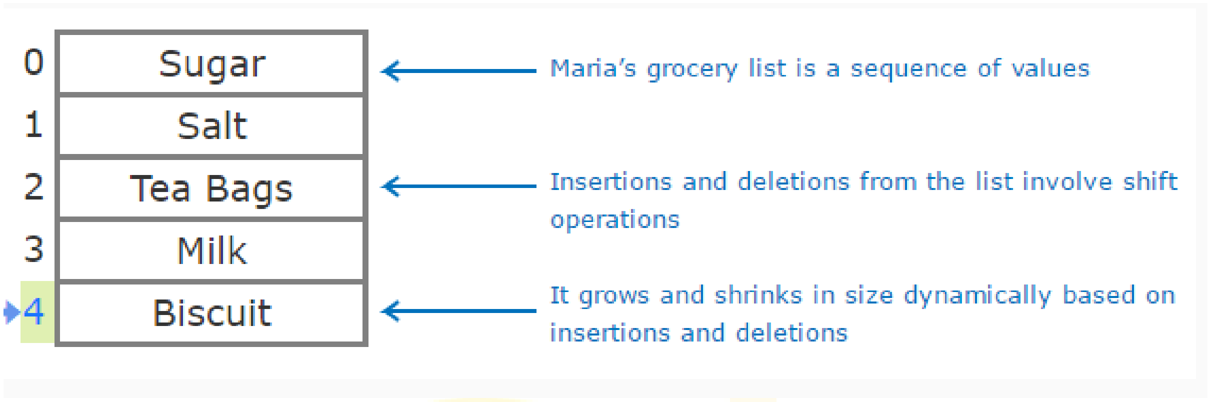
3. List using Array (Delete Operation): -


Enter the index position of the item to be deleted:
4

0	Sugar
1	Salt
2	Tea Bags
3	Milk

ONLINE LEARNING

#Summary: -


0
Sugar
1
Salt
2
Tea Bags
3
Milk
4
Biscuit
Maria's grocery list is a sequence of values
Insertions and deletions from the list involve shift operations
It grows and shrinks in size dynamically based on insertions and deletions

Not Good Choice to Implement with Array.

#List Using Linked List: -

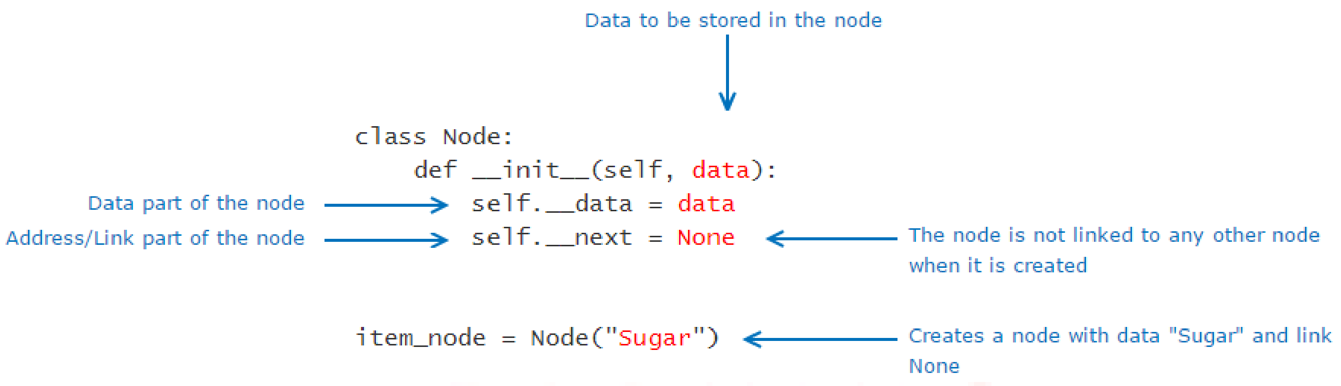
There is one more implementation for list which is using Linked List

A linked list consists of a group of nodes which together represent a sequence or a list. Each node will have a **data part** which holds the actual data and an address part which holds the link to the next node. The first node in the list is known as **head node** and the last node is known as **tail node**. Unlike array, in linked list, the nodes need not be stored in contiguous memory locations.

ONLINE LEARNING

Check Diagram from Handwritten Notes Pdf

The node in the linked list can be represented as follows:



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

item_node = Node("Sugar")
```

Data to be stored in the node
Data part of the node
Address/Link part of the node
The node is not linked to any other node when it is created
Creates a node with data "Sugar" and link None

LEARNING

1. List Using Linked List (Creation): -

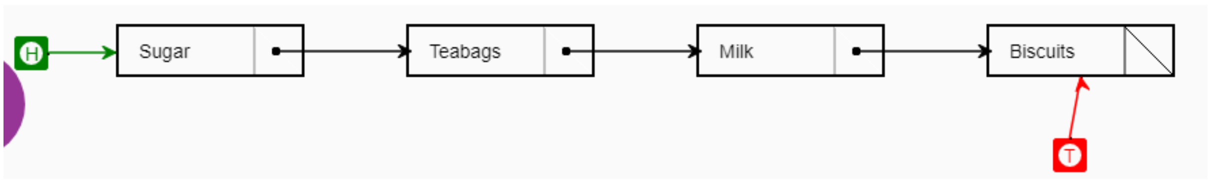
To link the nodes and create a linked list, let’s create a new class, **LinkedList** with two attributes, head and tail both initialized to None as shown below.

```
class LinkedList:
    def __init__(self):
        self.__head=None
        self.__tail=None
    def get_head(self):
        return self.__head
    def get_tail(self):
        return self.__tail
```

2. List Using Linked List (Display): -

Assume that Maria's list is maintained as a linked list and she wants to traverse through the list and display the items in the list starting from the first item.

```
display()
1. Call the head node as temp
2. While temp is not None,
   a. Display temp’s data
   b. Make the next node as temp
```



3. List Using Linked List (Add): -

Let's start creating Maria's list from the beginning. She wants to add "Sugar" as the first item and after that add "Teabags" to the end of the linked list.

```
add(data)
1. Create a new node with the data
2. If the linked list is empty (head node is not referring to any other node),
   make the head node and the tail node refer to the new node
3. Otherwise,
   a. Make the tail node's link refer to new node
   b. Call the new node as tail node
```

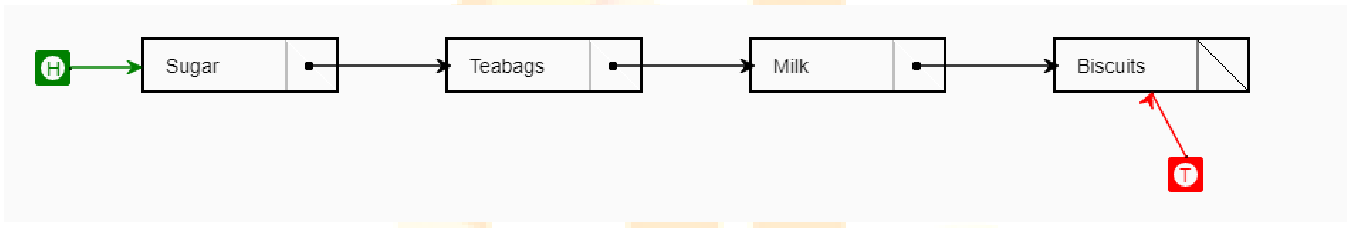
ONLINE LEARNING

#List- PART-2

1. List Using Linked List (Search):

Maria wants to find out whether her list has the following items:

Milk, Salt, Biscuits, Apple Juice, Pomegranate, Watermelon



2. List Using Linked List (Insert):

Maria wants to insert an item after an existing item in the list.

ONLINE LEARNING

```
insert(data,data_before)
1. Create a new node with the given data
2. If the data_before is None,
    a. Make the new node's link refer to head node
    b. Call the new node as head node
    c. If the new node's link is None, make it the tail node
3. Else
```

```
    a. Find the node with data_before, once found consider it as node_before
    b. Make the new node’s link refer to node_before’s link.
    c. Make the node_before’s link refer to new node
    d. If new node’s link is None, make it the tail node
4. If node with data_before is not found, display appropriate error message
```

3. List Using Linked List (Delete):

```
delete(data):
1. Find the node with the given data. If found,
    a. If the node to be deleted is head node, make the next node as head node
        1. If it is also the tail node, make the tail node as None
    b. Otherwise,
        1. Traverse till the node before the node to be deleted, call it temp
        2. Make temp’s link refer to node’s link.
        3. If the node to be deleted is the tail node, call the temp as tail node
        4. Make the node's link as None
2. If the node to be deleted is not found, display appropriate error message
```

ONLINE LEARNING

# Stack: -

Ex:-

Maria is arranging the shirts of Peter one on top of the other in the cupboard. She is very particular that Peter should always wear the shirt at the top.

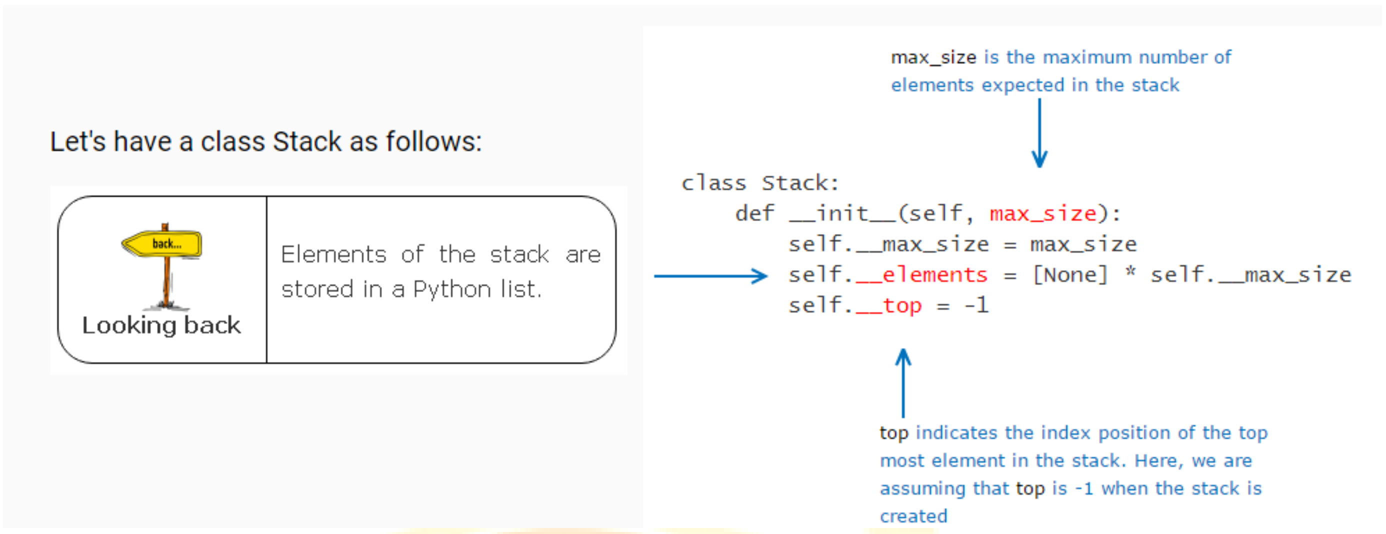
1. Arranging shirts in the Cupboard?
2. Peter take which shirt

Maria needs your help to arrange the shirts one on top of the other in the cupboard and she wants Peter to always take the shirt on top of the pile.

This pile of shirts arranged one on top of the other which follows **Last-In-First-Out (LIFO)** principle is known as **Stack**.

Operations possible on the stack are:

1. Push or insert an element to the top of the stack
2. Pop or remove an element from top of the stack



Let's have a class Stack as follows:

Elements of the stack are stored in a Python list.

max_size is the maximum number of elements expected in the stack

```
class Stack:
    def __init__(self, max_size):
        self.__max_size = max_size
        self.__elements = [None] * self.__max_size
        self.__top = -1
```

top indicates the index position of the top most element in the stack. Here, we are assuming that top is -1 when the stack is created

1. Push Algorithm: -

```
push(data):
    1. Check whether the stack is full. If full, display appropriate message
    2. If not,
        a. increment top by one
        b. Add the element at top position in the elements array
```

ONLINE LEARNING

2. Pop Algorithm: -

```
pop:
    1. Check whether the stack is empty. If empty, display appropriate message
    2. If not,
        a. Retrieve data at the top of the stack
        b. Decrement top by 1
        c. Return the retrieved data
```

ONLINE LEARNING

#Queue: -

Peter and Maria went for a movie one day. In the multiplex, the tickets were issued on first-come-first-serve basis and people were standing behind each other waiting for their turn. So, they went to the back and stood behind the last person waiting for the ticket.

Peter and Maria went for a movie one day. In the multiplex, the tickets were issued on first-come-first-serve basis and people were standing behind each other waiting for their turn. So they went to the back and stood behind the last person waiting for the ticket.

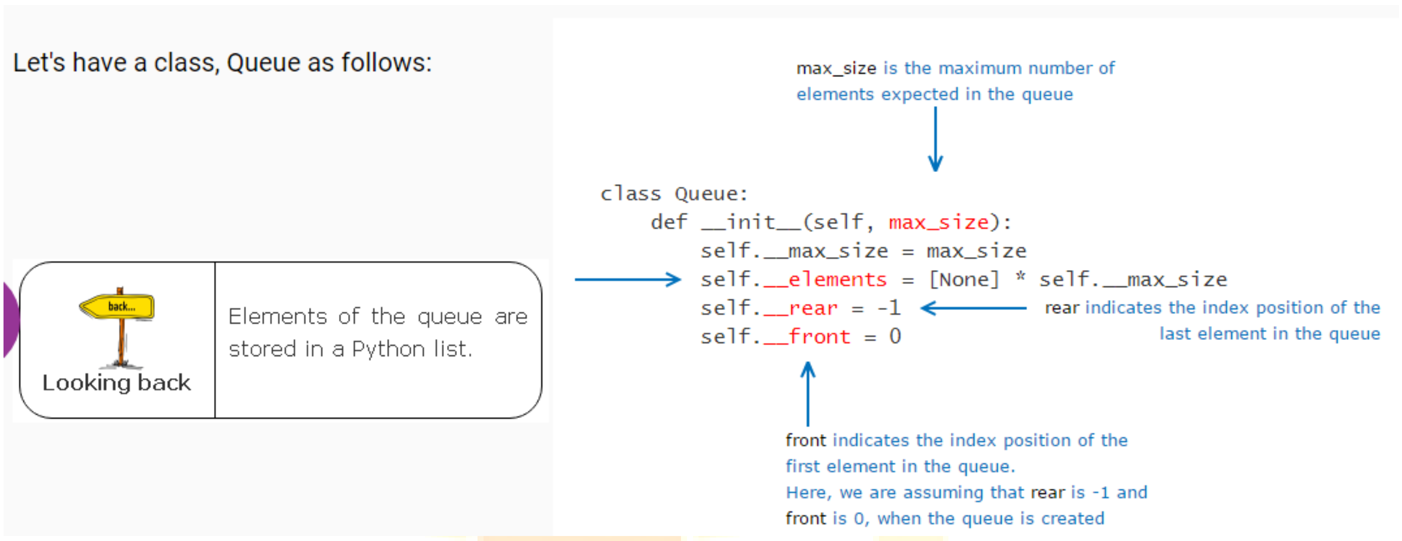
Maria is waiting for her turn to buy the movie tickets in the multiplex. She is trying to understand how this First-in-First-Out system is working. Let's see whether we can help her understand it.

Here, the people are standing one behind the other and they are serviced based on **First-In-First-Out (FIFO)** mechanism. Such an arrangement is known as **Queue**.

Operations possible on the queue are:

1. En-queue or add an element to the end of the queue
2. De-queue or remove an element from the front of the queue

ONLINE LEARNING


Let's have a class, Queue as follows:

```
class Queue:
    def __init__(self, max_size):
        self.__max_size = max_size
        self.__elements = [None] * self.__max_size
        self.__rear = -1
        self.__front = 0
```

max_size is the maximum number of elements expected in the queue

rear indicates the index position of the last element in the queue

front indicates the index position of the first element in the queue. Here, we are assuming that rear is -1 and front is 0, when the queue is created

Elements of the queue are stored in a Python list.

ONLINE LEARNING

1. Queue(enqueue Operation):-

enqueue (data):

1. Check whether queue is full. If full, display appropriate message

2. If not,

a. increment rear by one

b. Add the element at rear position in the elements array

2. Queue(dequeue Operation):-

```
dequeue()
1. Check whether the queue is empty. If it is empty, display appropriate message
2. If not,
   a. Retrieve data at the front of the queue
   b. Increment front by 1
   c. Return the retrieved data
```

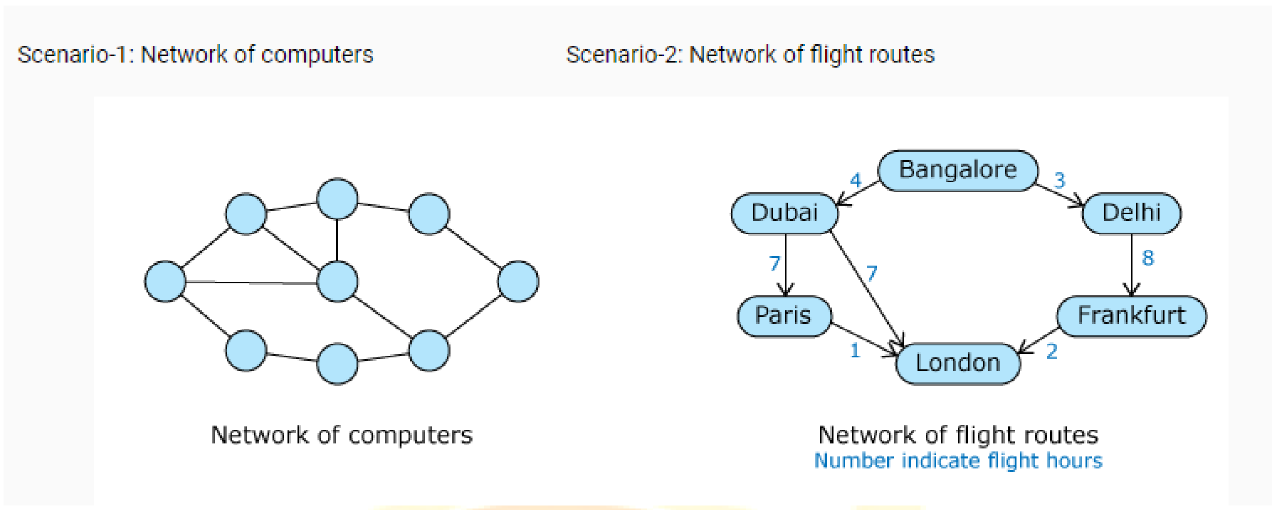
#Non- Linear Data Structures: -

Scenario-1:

The network engineers of a company are trying to connect all the computers (numbered 1 to 9) in the company network. They also need to provide a path to traverse from one computer to the other.

Scenario-2:

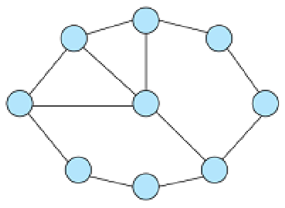
Maria is planning a vacation trip to Europe and is trying to choose the best air route from the available options based on the travel time in hrs.
Option-1: Bangalore(4hrs) -> Dubai(7hrs)-> Paris(1hr)-> London
Option-2: Bangalore(3hrs)->Delhi(8hrs)->Frankfurt(2hrs)->London
Option-3: Bangalore(4hrs)->Dubai(7hrs)->London


Scenario-1: Network of computers
Scenario-2: Network of flight routes
Bangalore
Dubai
Delhi
Paris
Frankfurt
London
4
3
7
7
8
1
2
Network of computers
Network of flight routes
Number indicate flight hours

#GRAPH

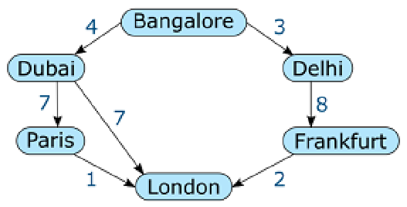
In these scenarios, we understand that we cannot use any of the linear data structures like array, linked list, stack or queue to represent it. Here, we need an arrangement which allows to have a set of vertices and edges between them. Such a data structure is known as **graph**.

Graph is a non-linear data structure having a set of **vertices**(or **nodes**) and edges between vertices. It can have any number of edges and nodes and any node can be connected to any other node by an edge. It can be implemented using arrays or linked lists.



Undirected Graph

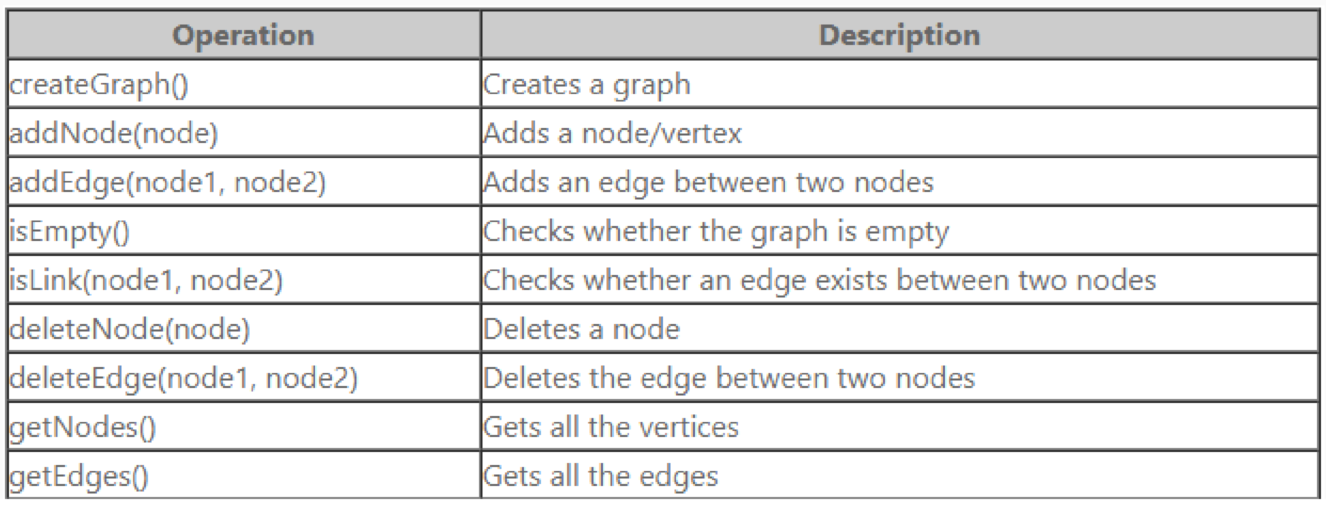
The graph in which edges are not directed from one vertex to another is known as undirected graph.



Weighted Directed Graph

The graph in which edges are directed from one vertex to another is known as directed graph. The graph in which weights or values are associated with each edge is known as weighted graph.

Common operations possible on graph are listed below:



Operation	Description
createGraph()	Creates a graph
addNode(node)	Adds a node/vertex
addEdge(node1, node2)	Adds an edge between two nodes
isEmpty()	Checks whether the graph is empty
isLink(node1, node2)	Checks whether an edge exists between two nodes
deleteNode(node)	Deletes a node
deleteEdge(node1, node2)	Deletes the edge between two nodes
getNodes()	Gets all the vertices
getEdges()	Gets all the edges

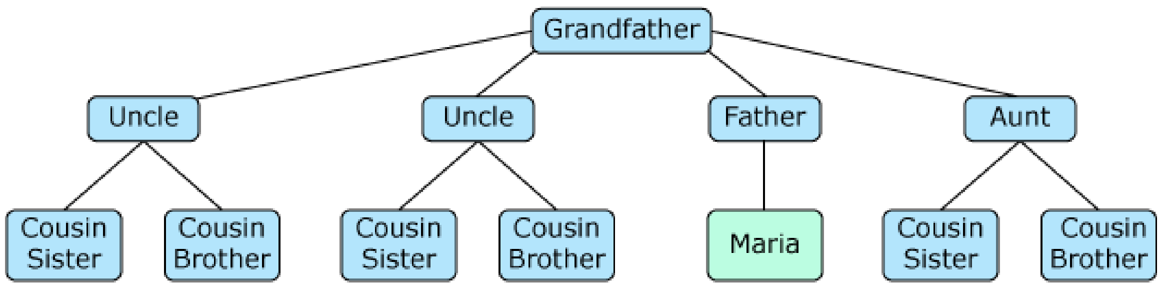
#TREE: -

Scenario-1:

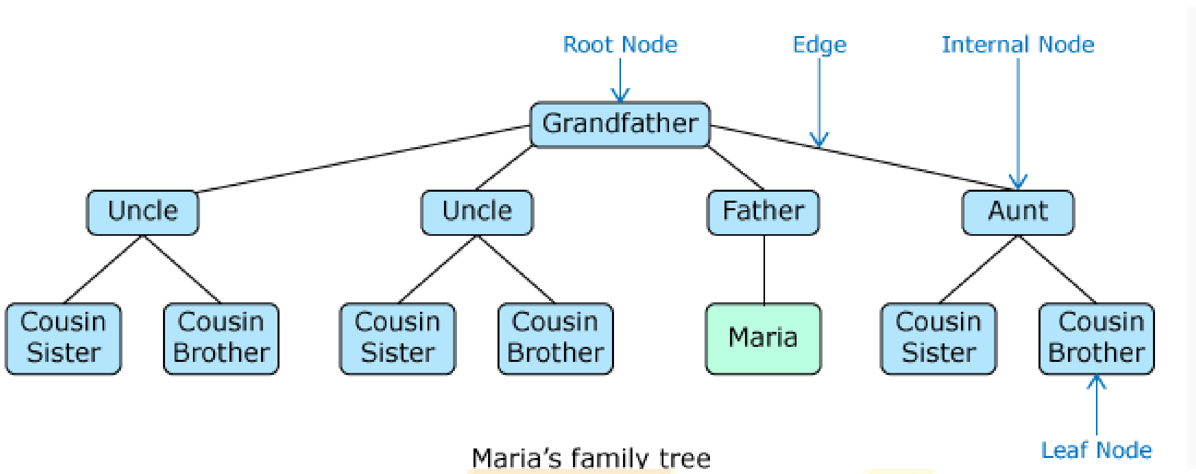
Maria is preparing her family tree. She has her grandfather as the head of the family. Her grandfather has three sons and one daughter. Her aunt and uncles have one son and one daughter each. Maria is the only child of her father.

Scenario-2:

Peter who has become the head of his team is preparing the organization structure. He has two senior managers reporting into him. To each of the senior managers, there are two managers reporting and to each of the managers, there are again two software engineers reporting.



This type of non-linear arrangement where a node is attached to one or more nodes directly beneath it, is a special type of graph known as **tree**. In this data structure, the top most node is called the root and the connections between nodes are called **edges**. Nodes that have no children are called **leaf nodes** and non-root and non-leaf nodes are called **internal nodes**.



Maria's family tree

ONLINE LEARNING