Module 3: Data Structures

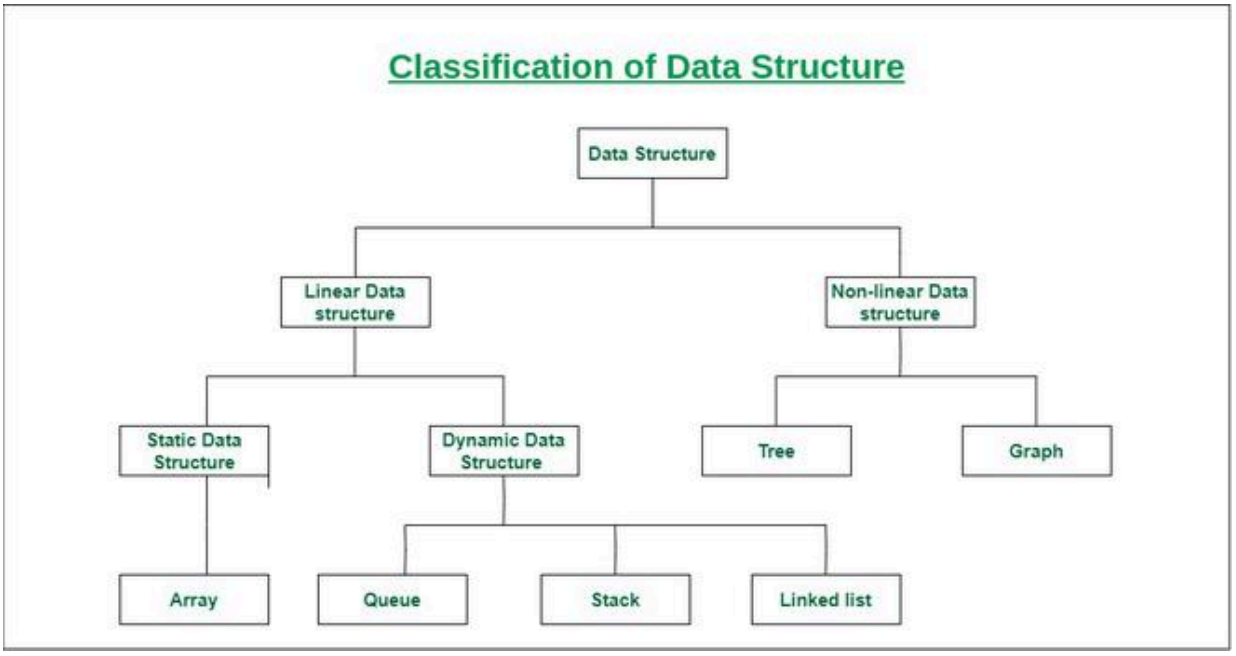
Introduction to Data Structures

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge of data structures.

Data structures are an integral part of computers used for the arrangement of data in memory. They are essential and responsible for organizing, processing, accessing, and storing data efficiently.

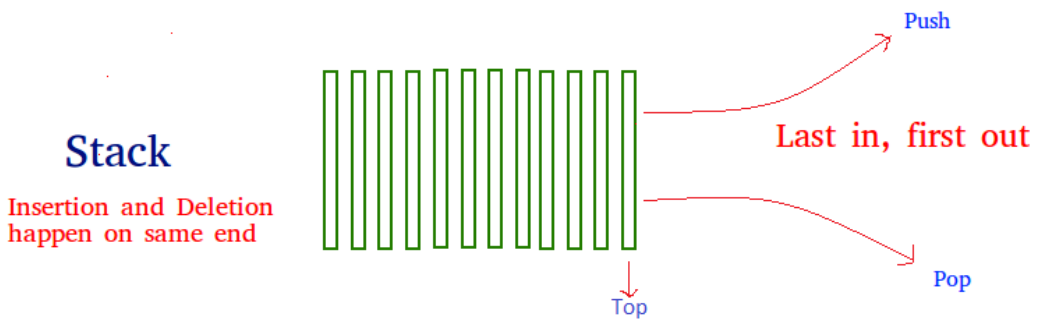
Types of Data Structures:



- Linear data structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
  Examples of linear data structures are array, stack, queue, linked list, etc.
    - Static data structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
      An example of this data structure is an array.
    - Dynamic data structure: In the dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
      Examples of this data structure are queue, stack, etc.
- Non-linear data structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we cannot traverse all the elements in a single run only.
  Examples of non-linear data structures are trees and graphs.

Stack

Stack is a linear data structure that follows a particular order in which the operations are performed. The order is LIFO(Last in first out). Entering and retrieving data is possible from only one end. The entering and retrieving of data is also called push and pop operation in a stack. There are different operations possible in a stack like reversing a stack using recursion, Sorting, Deleting the middle element of a stack, etc.



Methods of Stack

Python provides the following methods that are commonly used with the stack.

- empty() - It returns true, it the stack is empty. The time complexity is O(1).
- size() - It returns the length of the stack. The time complexity is O(1).
- top() - This method returns an address of the last element of the stack. The time complexity is O(1).
- push(g) - This method adds the element 'g' at the end of the stack - The time complexity is O(1).
- pop() - This method removes the topmost element of the stack. The time complexity is O(1).

Real-Life Applications of Stack:

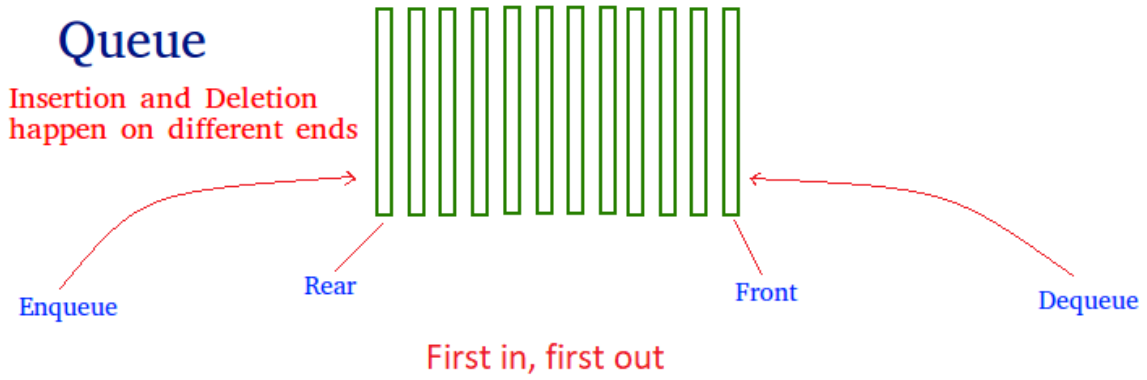
Real life example of a stack is the layer of eating plates arranged one above the other. When you remove a plate from the pile, you can take the plate to the top of the pile. But this is exactly the plate that was added most recently to the pile. If you want the plate at the bottom of the pile, you must remove all the plates on top of it to reach it.

Browsers use stack data structures to keep track of previously visited sites.

Call log in mobile also uses stack data structure.

Queue

Queue is a linear data structure that follows a particular order in which the operations are performed. The order is First In First Out (FIFO) i.e. the data item stored first will be accessed first. In this, entering and retrieving data is not done from only one end. An example of a queue is any queue of consumers for a resource where the consumer that came first is served first. Different operations are performed on a Queue like Reversing a Queue (with or without using recursion), Reversing the first K elements of a Queue, etc. A few basic operations performed In Queue are enqueue, dequeue, front, rear, etc.



Methods Available in Queue

Python provides the following methods, which are commonly used to perform the operation in Queue.

- put(item) - This function is used to insert element to the queue.
- get() - This function is used to extract the element from the queue.
- empty() - This function is used to check whether a queue is empty or not. It returns true if queue is empty.
- qsize - This function returns the length of the queue.
- full() - If the queue is full returns true; otherwise false.

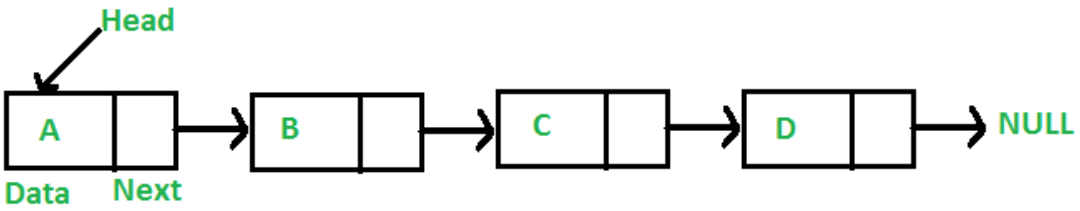
Real-Life Applications of Queue:

- A real-world example of a queue is a single-lane one-way road, where the vehicle that enters first will exit first.

- A more real-world example can be seen in the queue at the ticket windows.
- A cashier line in a store is also an example of a queue.
- People on an escalator

Linked List

A linked list is a linear data structure in which elements are not stored at contiguous memory locations.



Types of Linked Lists:

- Simple Linked List – In this type of linked list, one can move or traverse the linked list in only one direction. where the next pointer of each node points to other nodes but the next pointer of the last node points to NULL. It is also called "Singly Linked List".
- Doubly Linked List – In this type of linked list, one can move or traverse the linked list in both directions (Forward and Backward)
- Circular Linked List – In this type of linked list, the last node of the linked list contains the link of the first/head node of the linked list in its next pointer.
- Doubly Circular Linked List – A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node.
- Header Linked List – A header linked list is a special type of linked list that contains a header node at the beginning of the list.

Basic operations on Linked Lists:

- Deletion
- Insertion
- Search
- Display

Real-Life Applications of a Linked list:

A linked list is used in Round-Robin scheduling to keep track of the turn in multiplayer games.

It is used in image viewer. The previous and next images are linked, and hence can be accessed by the previous and next buttons.

In a music playlist, songs are linked to the previous and next songs.

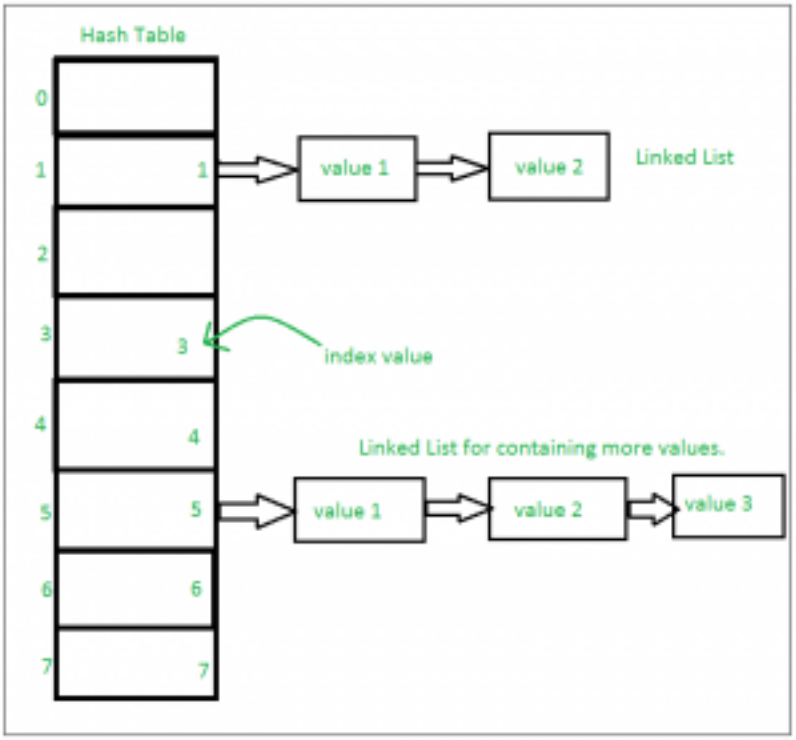
Sets

A Set is an unordered collection data type that is iterable, mutable, and has no duplicate elements.

The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table. Since sets are unordered, we cannot access items using indexes as we do in lists.

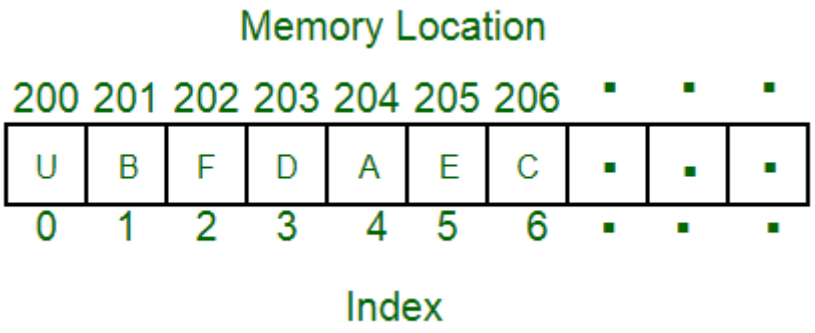
Internal working of Set

This is based on a data structure known as a hash table.
If Multiple values are present at the same index position, then the value is appended to that index position, to form a Linked List. In, Python Sets are implemented using a dictionary with dummy variables, where key beings the members set with greater optimizations to the time complexity.



Array

An array is a linear data structure and it is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together in one place. It allows the processing of a large amount of data in a relatively short period. The first element of the array is indexed by a subscript of 0. There are different operations possible in an array, like Searching, Sorting, Inserting, Traversing, Reversing, and Deleting.



Array Representation

An array can be declared in various ways and different languages. The important points that should be considered are as follows:

- Index starts with 0.
- We can access each element via its index.
- The length of the array defines the capacity to store the elements.

Array operations

Some of the basic operations supported by an array are as follows:

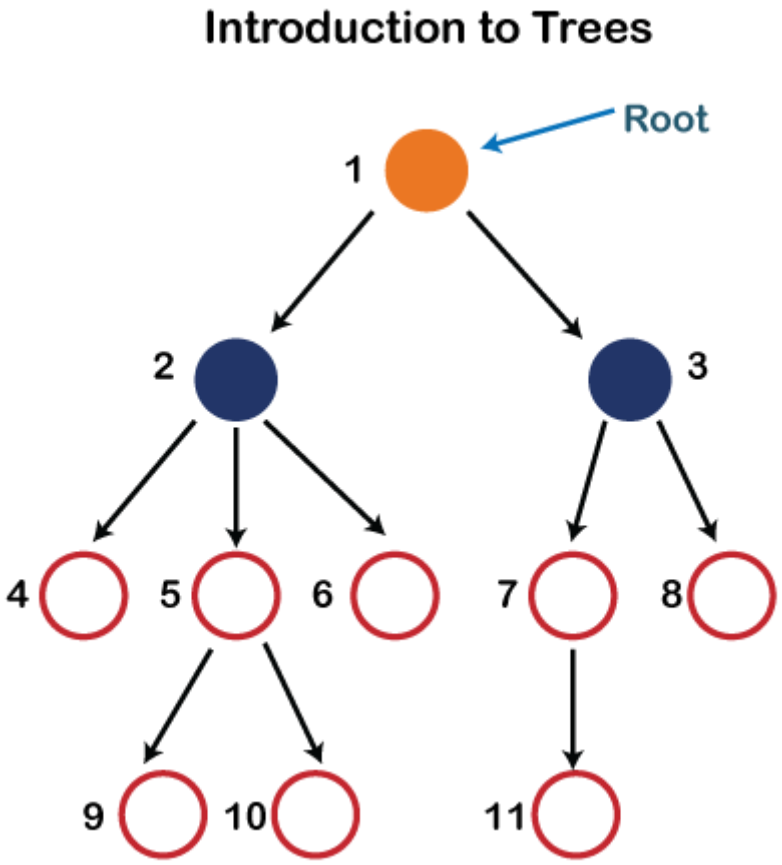
- Traverse - It prints all the elements one by one.
- Insertion - It adds an element at the given index.
- Deletion - It deletes an element at the given index.
- Search - It searches an element using the given index or by the value.
- Update - It updates an element at the given index.

Real-Life Applications of Array:

- An array is frequently used to store data for mathematical computations.
- It is used in image processing.
- It is also used in record management.
- Book pages are also real-life examples of an array.
- It is used in ordering boxes as well.

Trees

A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy. A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels. In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string. Each node contains some data and the link or reference of other nodes that can be called children.



In the above structure, each node is labelled with some number. Each arrow shown in the above figure is known as a link between the two nodes.

Root: The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that does not have any parent. In the above structure, node numbered 1 is the root node of the tree. If a node is directly linked to some other node, it would be called a parent-child relationship.

Child node: If the node is a descendant of any node, then the node is known as a child node.

Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.

Sibling: The nodes that have the same parent are known as siblings.

Leaf Node: - The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

Internal nodes: A node has atleast one child node known as an internal

Ancestor node: - An ancestor of a node is any predecessor node on a path from the root to that node. The root node does not have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

Descendant: The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Programs :

Pre-order

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
def printPreorder(root):

    if root:

        # First print the data of node
        print(root.val),

        # Then recur on left child
```

```
        printPreorder(root.left)

        # Finally recur on right child
        printPreorder(root.right)



# Driver code
if __name__ == "__main__":
  root = Node(1)
  root.left = Node(2)
  root.right = Node(3)
  root.left.left = Node(4)
  root.left.right = Node(5)

  # Function call
  print "Preorder traversal of binary tree is"
  printPreorder(root)
```

Post-Order:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# A function to do postorder tree traversal
def printPostorder(root):

    if root:
```

```
        # First recur on left child
        printPostorder(root.left)

        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val),



# Driver code
if __name__ == "__main__":
  root = Node(1)
  root.left = Node(2)
  root.right = Node(3)
  root.left.left = Node(4)
  root.left.right = Node(5)

  # Function call
  print "\nPostorder traversal of binary tree is"
  printPostorder(root)
```

In-order:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

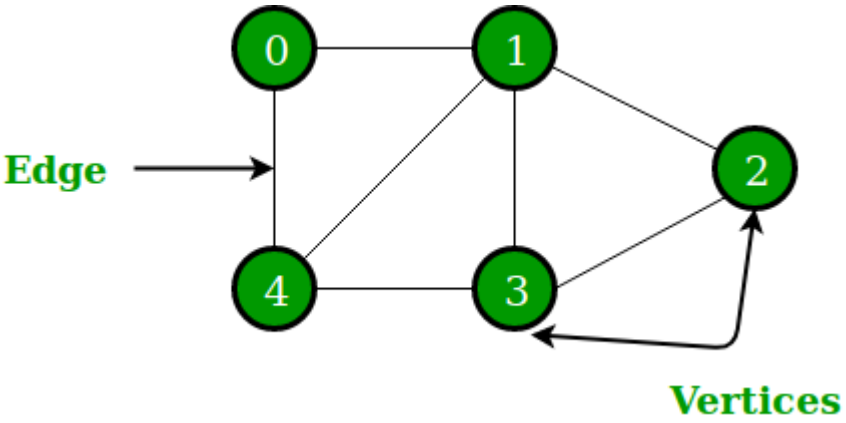
```
# A function to do inorder tree traversal
def printInorder(root):

    if root:

        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val),

        # now recur on right child
        printInorder(root.right)


# Driver code
if __name__ == "__main__":
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)

    # Function call
    print "\nInorder traversal of binary tree is"
    printInorder(root)
```

```
class Node:
    def _init_(self, data):
        self.data = data
        self.left = None
        self.right = None
class BinaryTree:
    def _init_(self):
        self.root = None
    def add_node(self, data):
        new_node = Node(data)
        # If root is None, assign the new node to the root
        if self.root is None:
            self.root = new_node
        else:
            focus_node = self.root
            parent = None
            while True:
                parent = focus_node
                # If data is less than focus_node, assign focus_node to the left child
                if data < focus_node.data:
                    focus_node = focus_node.left
                    # If there's no left child, assign the new node to the left child
                    if focus_node is None:
                        parent.left = new_node
                        return
                else:
                    focus_node = focus_node.right
                    # If there's no right child, assign the new node to the right child
```

```
                if focus_node is None:
                    parent.right = new_node
                    return
    def pre_order_traversal(self, focus_node):
        if focus_node is not None:
            print(focus_node.data, end=" ")
            self.pre_order_traversal(focus_node.left)
            self.pre_order_traversal(focus_node.right)
# Example usage
tree = BinaryTree()
tree.add_node(50)
tree.add_node(25)
tree.add_node(75)
tree.add_node(12)
tree.add_node(37)
tree.add_node(43)
tree.add_node(30)
tree.pre_order_traversal(tree.root)
```

Graphs

A graph is a nonlinear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as a Graph consisting of a finite set of vertices (or nodes) and a set of edges that connect a pair of nodes.



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

The following two are the most used representations of a graph.

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. The adjacency matrix for an undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Adjacency List

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.

Program:

```
import networkx
G = networkx.Graph()
G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(7)
G.add_node(9)

G.add_edge(1,2)
G.add_edge(3,1)
G.add_edge(2,4)
G.add_edge(4,1)
G.add_edge(9,1)
G.add_edge(1,7)
G.add_edge(2,9)

node_list = G.nodes()
print(node_list)

edge_list = G.edges()
print(edge_list)

G.remove_node(3)
node_list = G.nodes()
```

```
print(node_list)

G.remove_edge(1,2)
edge_list = G.edges()
print(edge_list)

n = G.number_of_nodes()
print(n)

m = G.number_of_edges()
print(m)

d = G.degree(2)
print(d)

neighbor_list = G.neighbors(2)
print(neighbor_list)

G.clear()
```

```
# importing networkx
import networkx as nx
# importing matplotlib.pyplot
import matplotlib.pyplot as plt

g = nx.Graph()

g.add_edge(1, 2)
g.add_edge(2, 3)
```

```
g.add_edge(3, 4)
g.add_edge(1, 4)
g.add_edge(1, 5)

nx.draw(g, with_labels = True)
plt.savefig("filename.png")
```