Graphs** are fundamental for representing relationships, with applications in navigation, networking, and optimization.

Let me know if you'd like further explanation or practical applications!

Sets:

Definition:

A set is a collection of unique and unordered elements. Sets are widely used in programming to handle distinct elements efficiently, as they automatically prevent duplicate values.

Properties of Sets:

1. **Unordered:** Elements in a set are not stored in a specific order.
2. **Unique Elements:** No duplicates are allowed.
3. **Mutable (in Python):** Elements can be added or removed after creation.
4. **Set Operations:** Union, Intersection, Difference, and Symmetric Difference are commonly used.

Set Implementation in Python:

Python provides a built-in `set` data type.

Example:

```python
# Creating a set
fruits = {"apple", "banana", "cherry"}

# Adding elements
fruits.add("orange")

# Removing elements
fruits.discard("banana")

# Checking membership
print("apple" in fruits)  # True

# Set operations
A = {1, 2, 3}
B = {3, 4, 5}

print("Union:", A | B)          # {1, 2, 3, 4, 5}
print("Intersection:", A & B)  # {3}
print("Difference:", A - B)    # {1, 2}
print("Symmetric Difference:", A ^ B)  # {1, 2, 4, 5}
```

Disjoint Set Union (Union-Find):

Definition:

Disjoint Set Union (DSU) is a data structure used to manage a partition of disjoint (non-overlapping) sets and efficiently perform union and find operations.

Key Operations:

1. **Find:** Determine which set an element belongs to. Implements path compression to speed up future queries.
2. **Union:** Merge two sets into a single set. Implements union by rank to keep the tree shallow.

Applications:

- Kruskal's algorithm for Minimum Spanning Tree.
- Connected components in graphs.
- Network connectivity.

Implementation in Python:

```python
class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
```

```
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            # Union by rank
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1

# Example Usage
dsu = DisjointSetUnion(5)  # Create 5 elements
dsu.union(0, 1)
dsu.union(1, 2)
print(dsu.find(2))  # Output: 0 (root of set)
print(dsu.find(3))  # Output: 3 (different set)
```

Graphs:

Definition:

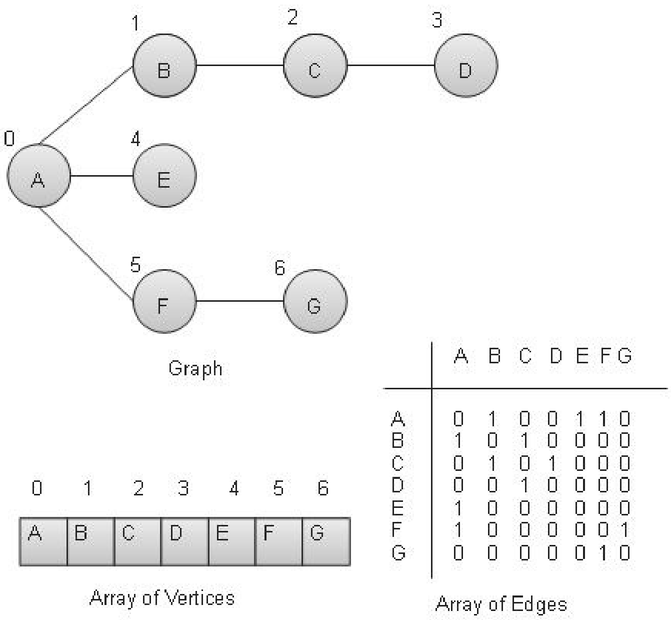
A graph is a data structure consisting of nodes (vertices) connected by edges. It is used to represent relationships and networks.

Types of Graphs:

1. **Directed vs Undirected:** Edges have direction or not.
2. **Weighted vs Unweighted:** Edges have weights or not.
3. **Cyclic vs Acyclic:** Graph contains cycles or not.

Graph Representation:

1. **Adjacency Matrix:** A 2D array where the element at `[i][j]` is `1` (or weight) if there is an edge from vertex `i` to `j`, else `0`.
2. **Adjacency List:** A list of lists where each list represents the neighboring vertices of a vertex.



Graph Implementation in Python:

Using Adjacency List:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)
```

```python
    def display(self):
        for node, neighbors in self.graph.items():
            print(f"{node} -> {neighbors}")

# Example Usage
g = Graph()
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.display()
```

Using Adjacency Matrix:

```python
class GraphMatrix:
    def __init__(self, size):
        self.size = size
        self.matrix = [[0] * size for _ in range(size)]

    def add_edge(self, u, v):
        self.matrix[u][v] = 1

    def display(self):
        for row in self.matrix:
            print(row)

# Example Usage
gm = GraphMatrix(3)
gm.add_edge(0, 1)
gm.add_edge(1, 2)
gm.display()
```

Graph Traversal:

1. **Depth-First Search (DFS):**

   ```python
   def dfs(graph, node, visited):
       if node not in visited:
           print(node, end=" ")
           visited.add(node)
           for neighbor in graph[node]:
               dfs(graph, neighbor, visited)

   # Example
   graph = {0: [1, 2], 1: [2], 2: [0, 3], 3: []}
   visited = set()
   dfs(graph, 0, visited)  # Output: 0 1 2 3
   ```
2. **Breadth-First Search (BFS):**

   ```python
   from collections import deque

   def bfs(graph, start):
       visited = set()
       queue = deque([start])
       visited.add(start)

       while queue:
           node = queue.popleft()
           print(node, end=" ")

           for neighbor in graph[node]:
               if neighbor not in visited:
                   visited.add(neighbor)
                   queue.append(neighbor)

   # Example
   graph = {0: [1, 2], 1: [2], 2: [0, 3], 3: []}
   bfs(graph, 0)  # Output: 0 1 2 3
   ```

Summary:

1. **Sets** are efficient for unique elements and operations like union and intersection.
2. **Disjoint Set Union (Union-Find)** is powerful for partitioning and dynamic connectivity.
3. **Graphs** are fundamental for representing relationships, with applications in navigation, networking, and optimization.

Let me know if you'd like further explanation or practical applications!

In [ ]: