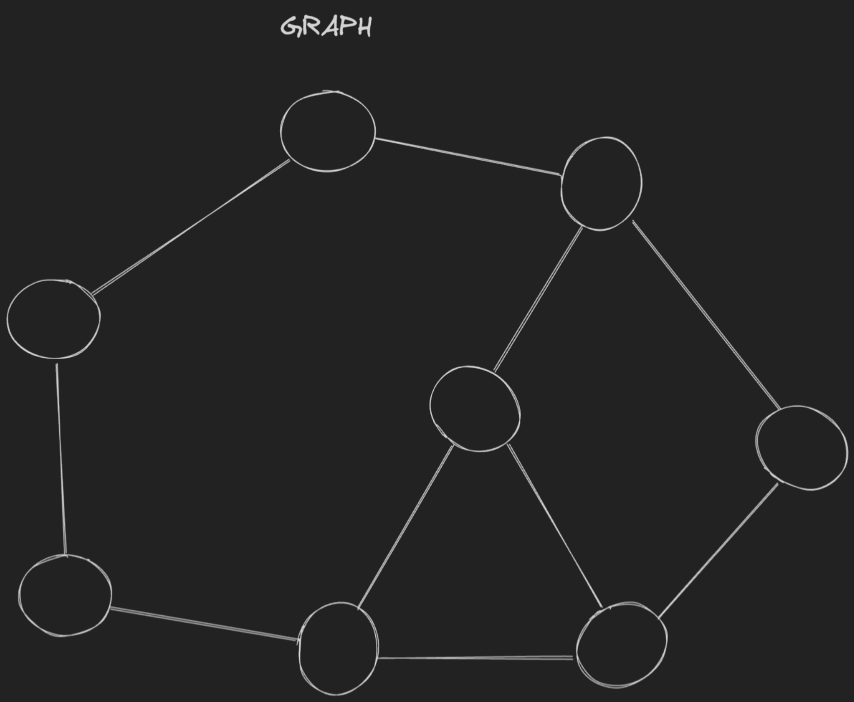
Operation	Singly Linked List	Doubly Linked List
Search	O(n)	O(n)
Insertion (at head)	O(1)	O(1)
Insertion (at tail)	O(1)	O(1)
Remove (at head)	O(1)	O(1)
Remove (at tail)	O(n)	O(1)
Remove (at middle)	O(n)	O(n)

> V — the number of vertices
>
> E — the number of edges

Definition

A non-linear data structure consisting of **vertices** (also called nodes) and **edges**. It is a collection of interconnected nodes.

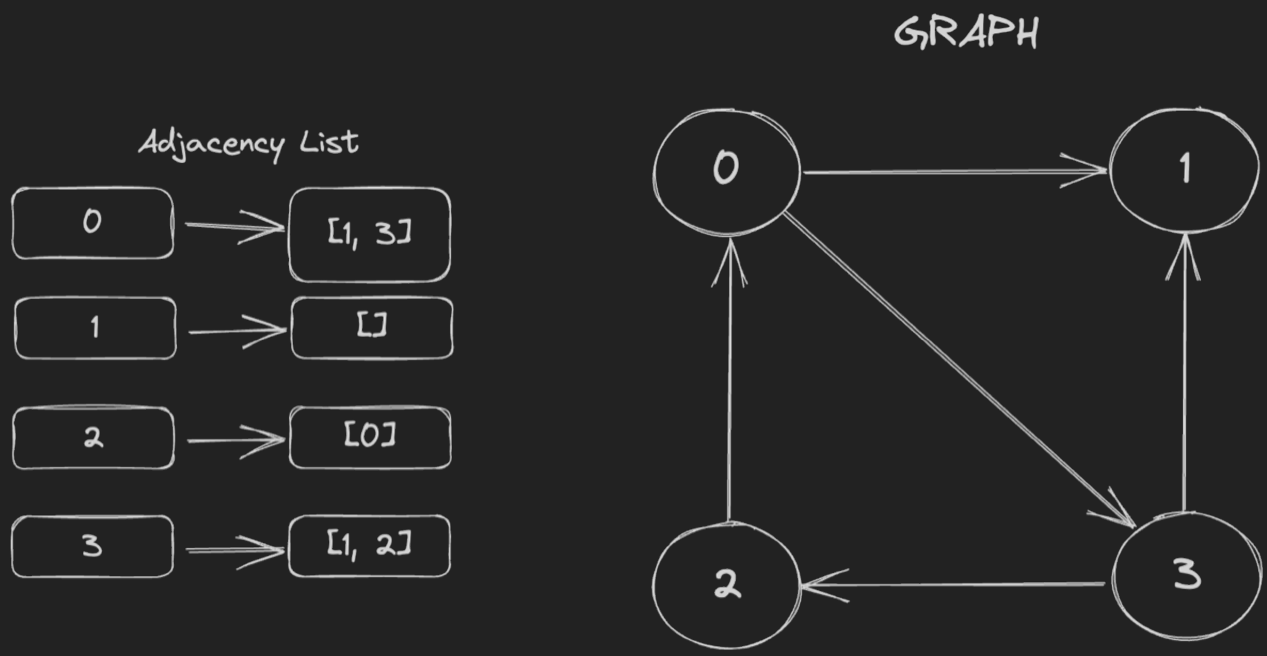


Terminology

Node (Vertex)	A node represents an individual element or object. Nodes are the fundamental units that make up a graph, and they can be connected to other nodes by edges.
Edge	The connections between the vertices. Edges can be directed or undirected, and they can also have weights associated with them
Connected	A connected graph is a graph in which there is a path between every pair of vertices. This means that you can reach any vertex from any other vertex in the graph. This can also be applied to just two verticies of a graph.
Connected Components	A subset of verticies $V_i \subseteq V$ that is connected.
Neighbors	Two nodes are neighbors if verticies u, v are connected by some edge (u, v)
Degree	Number of edges connected to vertex v
Path	Sequences of verticies connected by edges
Path Length	Number of edges in a path
Cycle	A cycle is a path that starts and ends at the same vertex, allowing you to traverse multiple edges and vertices. It forms a closed loop within the graph.
Acyclic	An acyclic graph is a graph that does not contain any cycles. It means there are no paths that start and end at the same vertex, resulting in a tree-like structure.
Directed Graph	A graph in which edges have a specific direction, meaning they can be traversed in one direction only. Edge (u, v) does not imply (v, u). If not considered a DAG, can be assumed there is cycles.
Undirected Graph	A graph where there is no direction, they can be traversed in both directions. Edge (u, v) implies (v, u).
Weighted Graph	A graph in which edges have a weight or cost associated with them. This weight represents some value or distance between the vertices
Directed Acyclic Graph (DAG)	A directed acyclic graph is a directed graph that does not contain any cycles. It means that there are no directed paths that start and end at the same vertex.
Tree	Trees are 1. Connected and acyclic, 2. Removing edge disconnects graph, 3. adding edge creates a cycle.

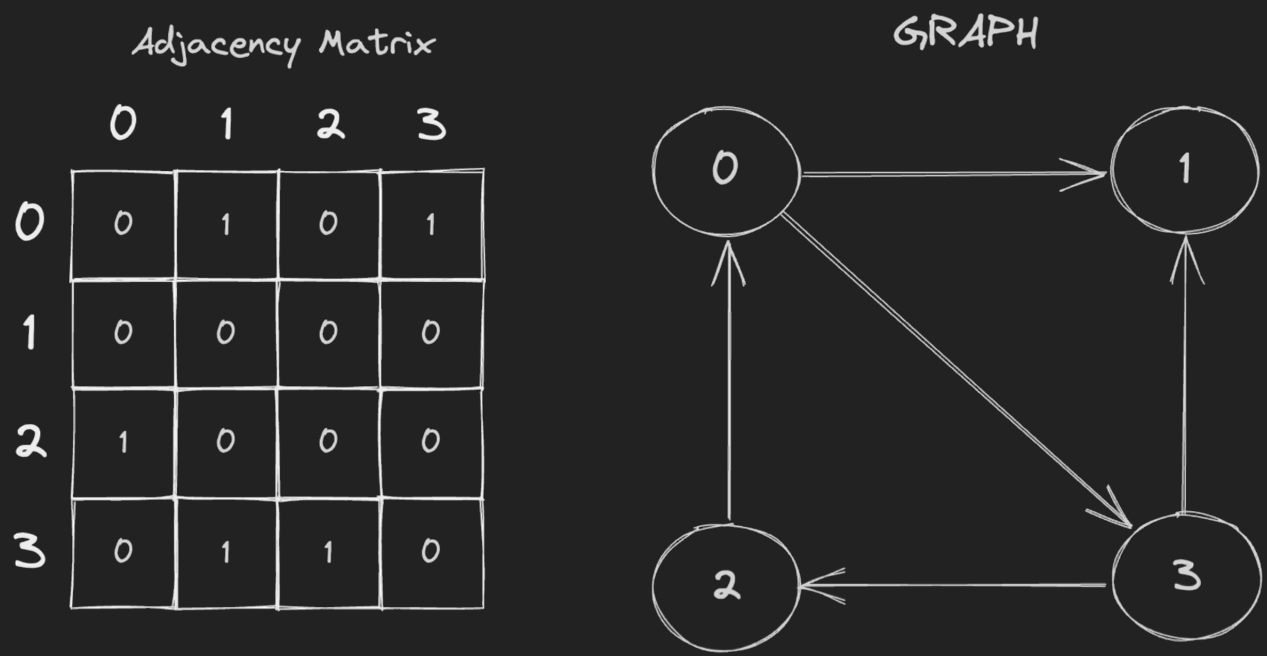
Adjacency List

A collection of linked lists or arrays where each vertex maintains a list of its adjacent vertices. If the edges have weights associated you can have a tuple of values `(1, 10)`



Adjacency Matrix

A square matrix representing the graph, where the rows and columns correspond to the vertices, and the matrix elements indicate the presence or absence of an edge between the vertices. In an undirected graph, both vertices would be mapped to each other. If the edges have weights, the value can indicate the weight.



BFS / DFS

Since all trees are graph we can implement BFS and DFS in a similar fashion as we did for trees

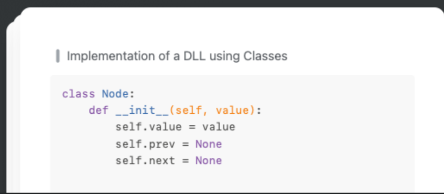
BFS on Adjacency Matrix

Use Cases:

- Representing relationships between objects/entities
- Modeling networks, social connections, and dependencies
- Pathfinding algorithms, such as Dijkstra's algorithm or A* search algorithm
- Representing geographical maps or transportation networks
- Recommendation systems and collaborative filtering

Doubly Linked List Implementation

Implementation of a DLL using Classes



↑ Graph

BFS on Adjacency Matrix

```python
from typing import List, Union

def bfs(graph: List[List[int]], source: int, needle: int) -> Union[List[int], None]:
    seen: List[bool] = [False] * len(graph)
    prev: List[int] = [-1] * len(graph)

    seen[source] = True
    queue: List[int] = [source]

    while queue:
        curr: int = queue.pop(0)

        if curr == needle:
            break

        adjs: List[int] = graph[curr]
        for i in range(len(graph)):
            if adjs[i] == 0:
                continue

            if seen[i]:
                continue

            seen[i] = True
            prev[i] = curr
            queue.append(i)

        seen[curr] = True

    # Build path
    curr = needle
    out: List[int] = []

    while prev[curr] != -1:
        out.append(curr)
        curr = prev[curr]

    if out:
        return [source] + out[::-1]

    return N
```

↑ Graph

Doubly Linked List Implementation

> Implementation of a DLL using Classes

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.prev = None
        self.next = None


class DoublyLinkedList:
    def __init__(self):
        self.length = 0
        self.head = None
        self.tail = None

    def prepend(self, item):
        node = Node(item)

        self.length += 1
        if not self.head:
            self.head = self.tail = node
            return

        node.next = self.head
        self.head.prev = node
        self.head = node

    def insert_at(self, item, idx):
        if idx > self.length:
            raise IndexError('Linked List is not long enough')
        elif idx == self.length:
            self.append(item)
            return
        elif idx == 0:
            self.prepend(item)
            return

        self.length += 1
        curr = self.get_at(idx)
        node = Node(item)

        node.next = curr
        node.prev = curr.prev
        curr.prev = node

        if node.prev:
            node.prev.next = curr

    def append(self, item):
        node = Node(item)

        self.length += 1
        if not self.tail:
            self.head = self.tail = node
            return

        node.prev = self.tail
        self.tail.next = node
        self.tail = node

    def remove(self, item):
        curr = self.head
```