MF20006: Introduction to Computer Science

Lecture 5: Data Structures

Hui Xu

xuh@fudan.edu.cn

Outline

1. Basic Concept

How to Describe Companies with a Program?

HOME > NVDA • NASDAQ

NVIDIA Corp

$116.00 ↓10.33% -13.37 1M

Sep 23, 12:31:01 AM UTC-4 · USD · NASDAQ · Disclaimer

1D 5D 1M 6M YTD 1Y 5Y MAX

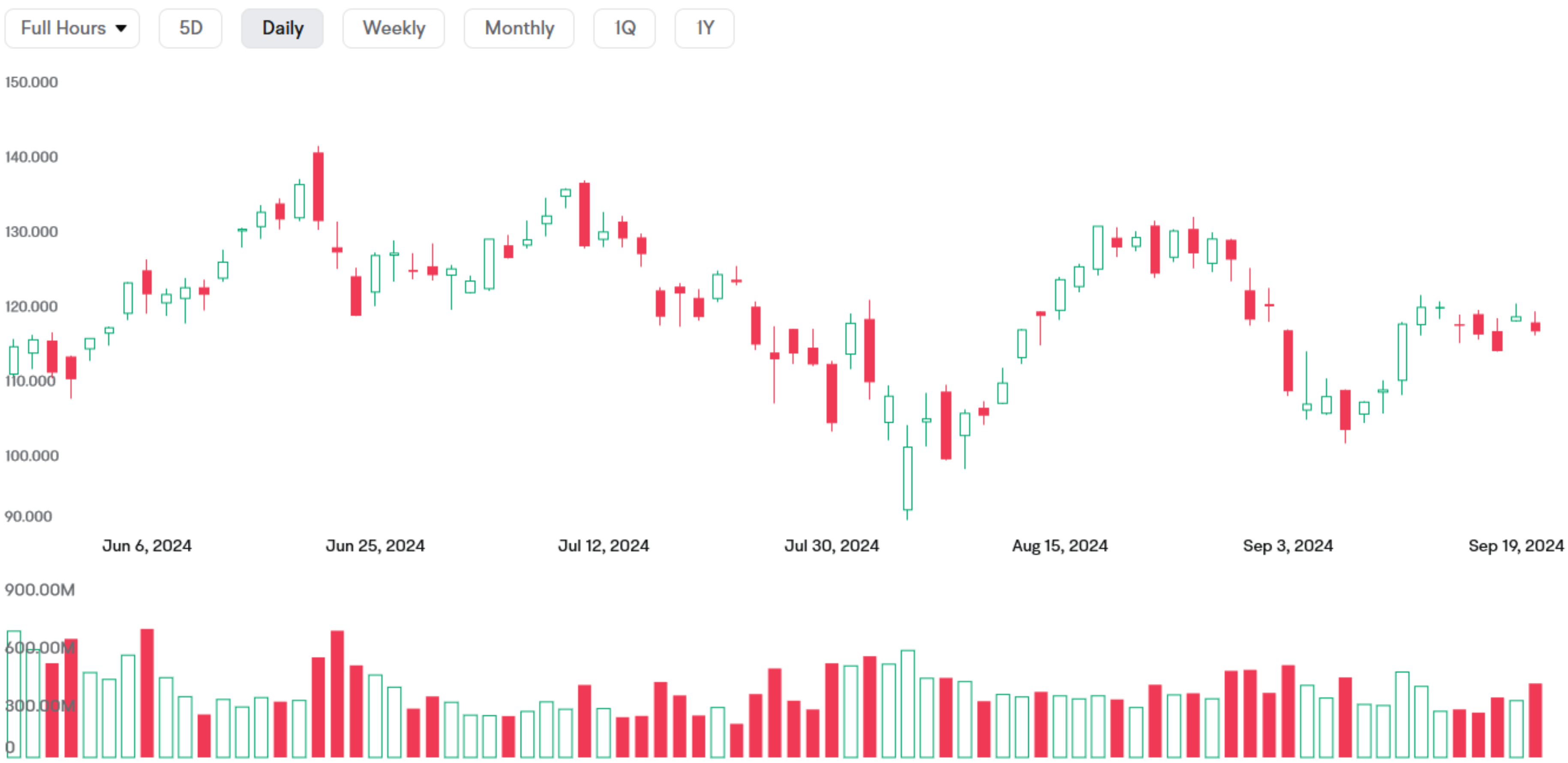
Compare to

Tesla Inc	Apple Inc	Microsoft Corp	Amazon.com Inc
$238.25	$228.20	$435.27	$191.60
TSLA ↑8.14%	AAPL ↑0.60%	MSFT ↑4.43%	AMZN ↑8.22%

PREVIOUS CLOSE	$117.87
DAY RANGE	$115.39 - $118.62
YEAR RANGE	$39.23 - $140.76
MARKET CAP	2.85T USD
AVG VOLUME	332.34M
P/E RATIO	54.48
DIVIDEND YIELD	0.03%
PRIMARY EXCHANGE	NASDAQ

How to Manage Historical Prices?

- ❑We want to collect the historical price, *e.g.*, as a candlestick chart



Data Structure

- **A specialized format for organizing and storing data, with defined operations for accessing and modifying it efficiently.**
  - How data is arranged in memory.
  - How we insert, delete, search, and update elements.

Implement Your Own Data Structure with Dataclass

- A dataclass is a special kind of Python class that automatically gives you the usual methods you'd write for a data-holding class.
- It generates an __init__() method automatically.
  - The field type should be specified.

```python
from dataclasses import dataclass

@dataclass
class Point():
    x: int
    y: int

    def other_methods():
        ...
```

```python
class Point():

    def  __init__(self, x, y):
        self.x = x
        self.y = y

    def other_methods():
        ...
```

If the field type is not specified...

- The class constructor does accept the field parameter.

```
@dataclass
class Point():
    x: int
    y: int

let p = Point(10, 20)
```

==

```
@dataclass
class Point():
    x = 0
    y = 0

let p = Point()
p.x = 10
p.y = 20
```

Data Structure of a Company

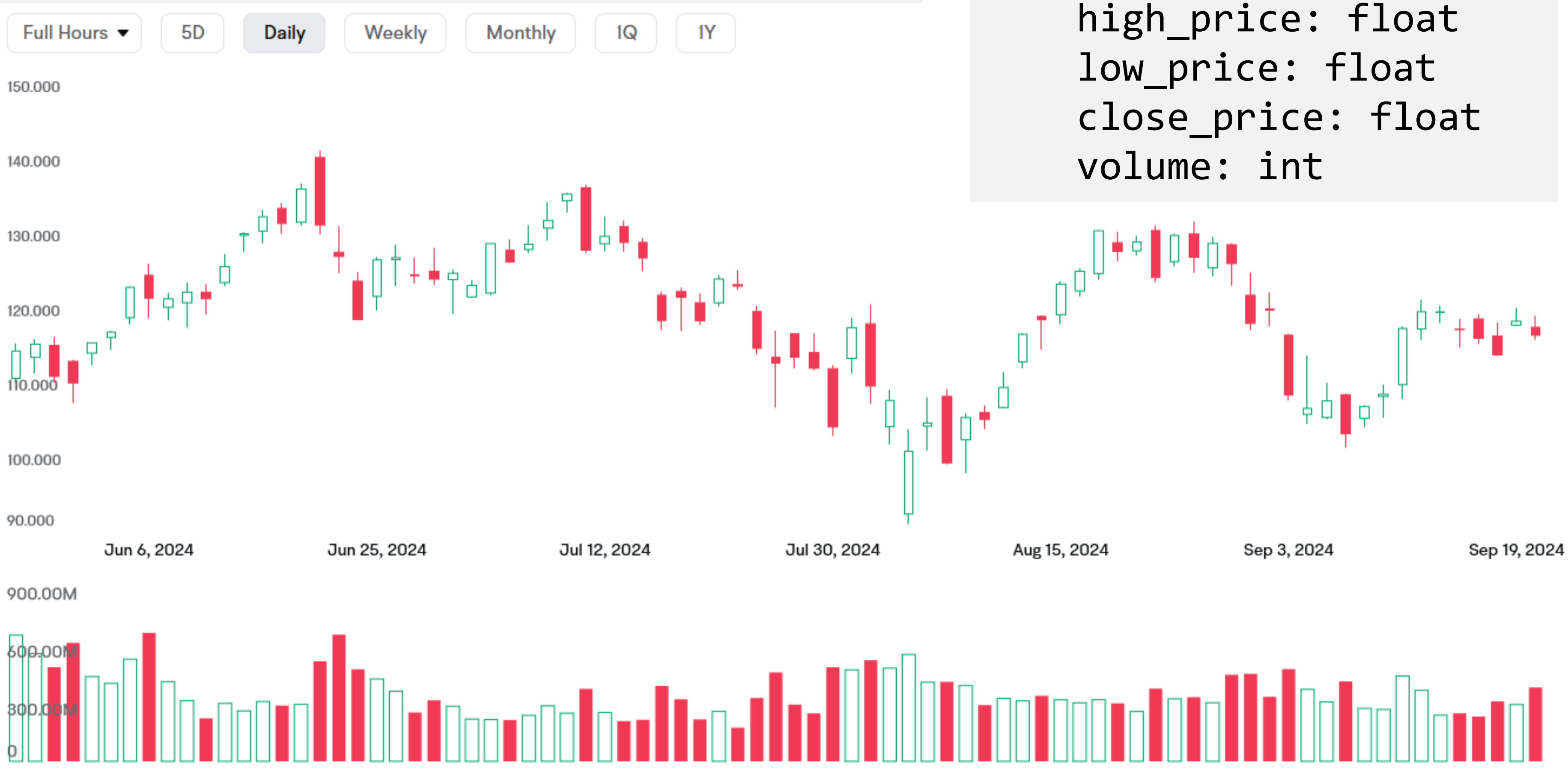
```python
@dataclass
class Company:
    name: str                             # "NVIDIA Corporation"
    ticker: str                           # "NVDA"
    exchange: str                         # "NASDAQ"

    # Trading info
    current_price: float                  # $100.1
    open_price: float                     # $99.12
    close_price: float
    high_price: float
    low_price: float                      # $98.79
    volume: int                           # 1000

    # Financial info
    pe: int                               # P/E ratio
    pb: float                             # P/B ratio
    market_cap: float                     # e.g., $1000 billion
```

Data Structure of Historical Prices

- We want to collect the historical price, *e.g.,* as a candlestick chart

```
@dataclass
class Company:
    ...
    historical_prices: List[TradingInfo]
```

```
@dataclass
class TradingInfo:
    date: int
    open_price: float
    high_price: float
    low_price: float
    close_price: float
    volume: int
```



2. Linear Data Structures

Linear Data Structures

❑Consider the distribution of data:

- Array: a collection of elements stored in contiguous memory locations.
- Linked List: data are not contiguously stored.

❑Consider the behaviors:

- Queue: first-in-first-out.
- Stack: first-in-last-out.

Array

- A collection of elements stored in contiguous memory locations.
- All elements are of the same data type.
- Length: number of elements within the array.
- Size: means the memory space it occupied.

Memory Address	0x200	0x201	0x202	0x203	0x204	0x205	0x206	0x207	0x208	0x209
Data	**F**	**I**	**S**	**F**	**1**	**3**	**0**	**0**	**2**	**0**
Index	0	1	2	3	4	5	6	7	8	9

Array

- Access any elements via base address + offset.
- Supposing the size of each data unit is 4 types, *e.g.*, 32bit integer, we can retrieve the *i*th data from the memory address via a[i].

```
a[i] = a[0] + 4*i
```

Memory Address	0x200	0x204	0x208	0x20c	0x210	0x214	0x218	0x21c	0x220	0x224
Data	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**21**	**34**	**55**
Index	0	1	2	3	4	5	6	7	8	9

Array length: 10

Array size: 40 bytes

Array Operations

- **Read/write any elements via base address + offset.**
  - Constant time, denoted as O(1).
- **Searching an element from an array of length n.**
  - Average: n/2 elements until find it, denoted as O(n).
- **Insertion or deletion an element requires shifting the rest elements.**
  - Average: moving n/2 elements, denoted as O(n).

Memory Address	0x200	0x204	0x208	0x20c	0x210	0x214	0x218	0x21c	0x220	0x224
Data	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**21**	**34**	**55**
Index	0	1	2	3	4	5	6	7	8	9

Two-dimensional Array: Matrix

- Consist of multiple one-dimensional array; each has the same length.
- Supposing an i32 array has m rows, and each row has length n, we can retrieve data of the *i*th row and *j*th column via a[i][j].

```
a[i][j] = a[0][0] + 4*i*n + j
```

Row Index					
0	**1**	**1**	**2**	**3**	**5**
1	**8**	**13**	**21**	**34**	**55**
	0	1	2	3	4

Column Index

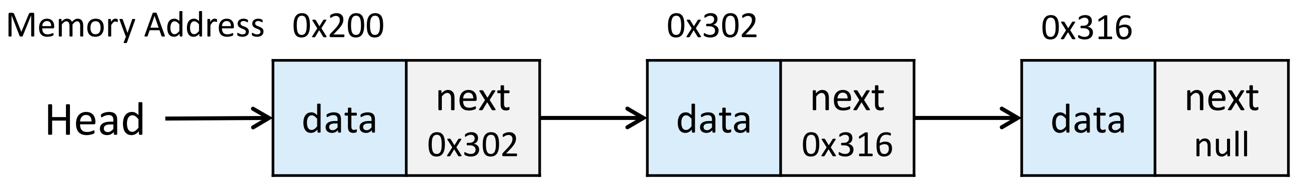
Array View

Memory Address	0x200	0x204	0x208	0x20c	0x210	0x214	0x218	0x21c	0x220	0x224
Data	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**21**	**34**	**55**
Index	0, 0	0, 1	0, 2	0, 3	0, 4	1, 1	1, 2	1, 3	1, 4	1, 5

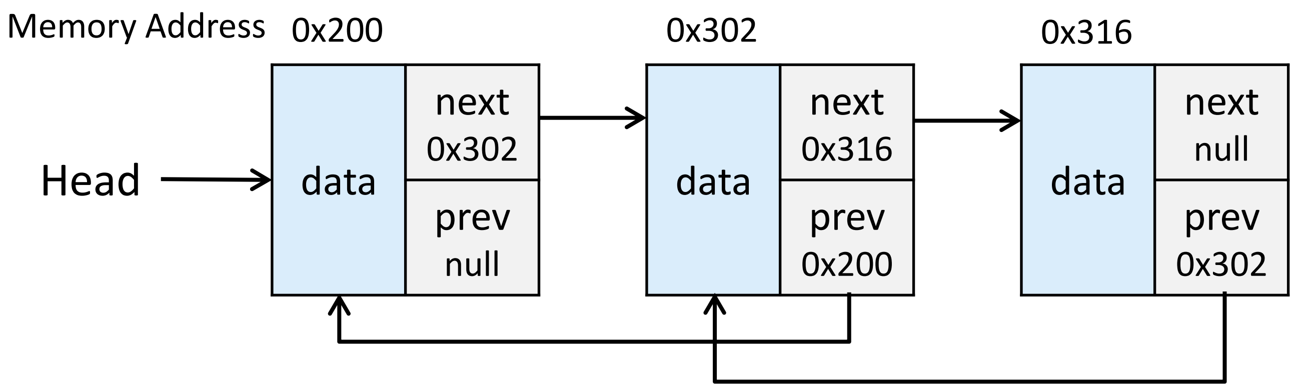
Memory View

Linked List

- ❑Similar as array, but the data are not contiguously stored.
- ❑Each list node has a data field and an address field to the next or the previous node.



Single-linked List



Double-linked List

Pros and Cons of List Operations

- **The cost of accessing elements of different positions varies a lot.**
  - Related to the length of the list: denoted as O(n).
- **Insertion/deletion an element at any position costs constant time.**
  - Unrelated to the length of the list: O(1).

Does Python have Array or Linked List?

- **Python's built-in list is neither linked list nor array.**
  - The built-in list is a dynamic array.
  - Each array element is an object pointer.
  - Elements can be different types.
- **Alternative feature: numpy.ndarray.**

Example: Matrix Addition with Python

```python
def add(a: list[list[int]], b: list[list[int]]) -> list[list[int]]:
    rows = len(a)
    cols = len(a[0])
    result = [[0 for _ in range(cols)] for _ in range(rows)]
    for i in range(rows):
        for j in range(cols):
            result[i][j] = a[i][j] + b[i][j]
    return result
```

Matrix Addition via NumPy Native Code

- Python executes each statement through the interpreter.
- In comparison, NumPy native code bypasses the interpreter.
  - Compiled directly into machine instructions.

```
# Create two 2x2 arrays
a = np.array([[1, 2],
              [3, 4]])
b = np.array([[5, 6],
              [7, 8]])

c = a + b
```

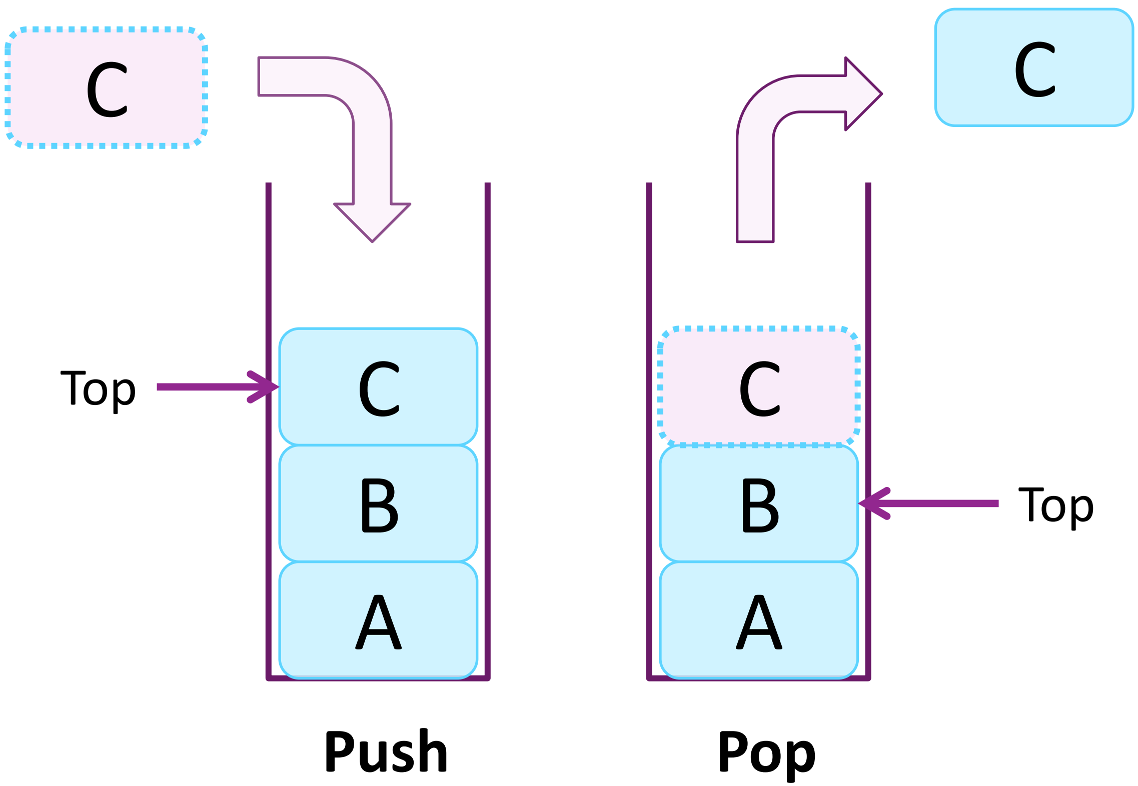
Benchmark the Performance

```python
size = 1000  # 1000x1000 matrix
a_list = [[random.random() for _ in range(size)] for _ in range(size)]
b_list = [[random.random() for _ in range(size)] for _ in range(size)]
a_np = np.array(a_list)
b_np = np.array(b_list)

start = time.perf_counter()
add(a_list, b_list)
end = time.perf_counter()
print(f"Pure Python (list) time:  {end - start:.4f} seconds")

start = time.perf_counter()
a_np + b_np
end = time.perf_counter()
print(f"NumPy (ndarray) time:     {end - start:.4f} seconds")
```

```
#: python3 mat_add.py
Benchmarking matrix addition for 1000x1000 matrices...
Pure Python (list) time:  0.1569 seconds
NumPy (ndarray) time:     0.0026 seconds
```

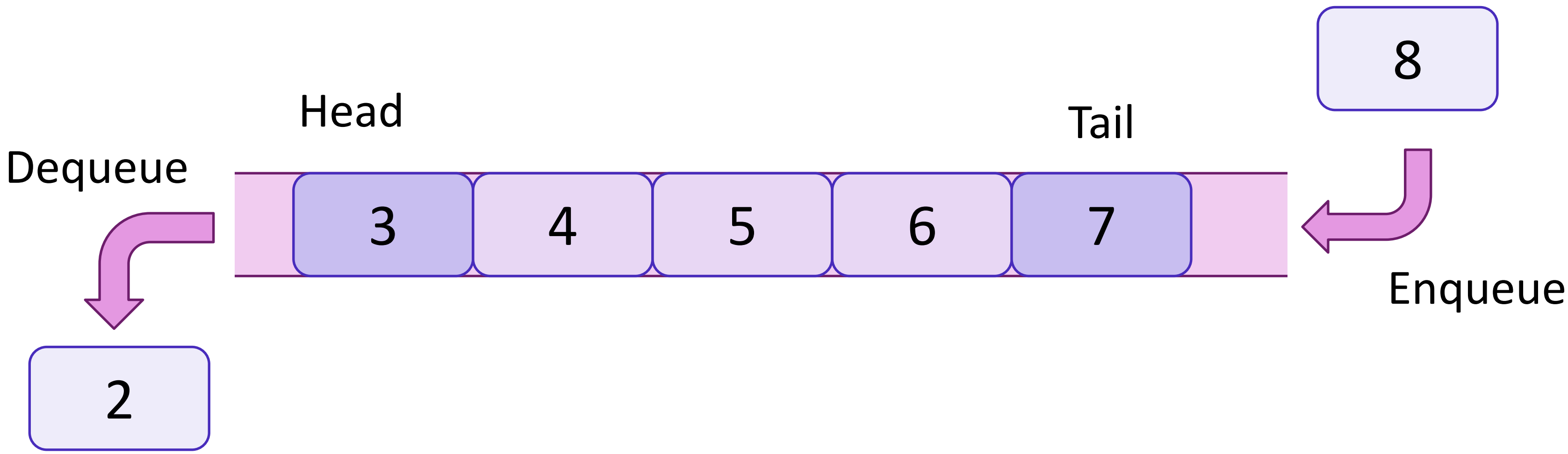
Stack

- ❑A collection of elements with Last-In-First-Out (LIFO) order.
- ❑Push: Add an element to the top of the stack.
- ❑Pop: Remove the top element from the stack.



Queue

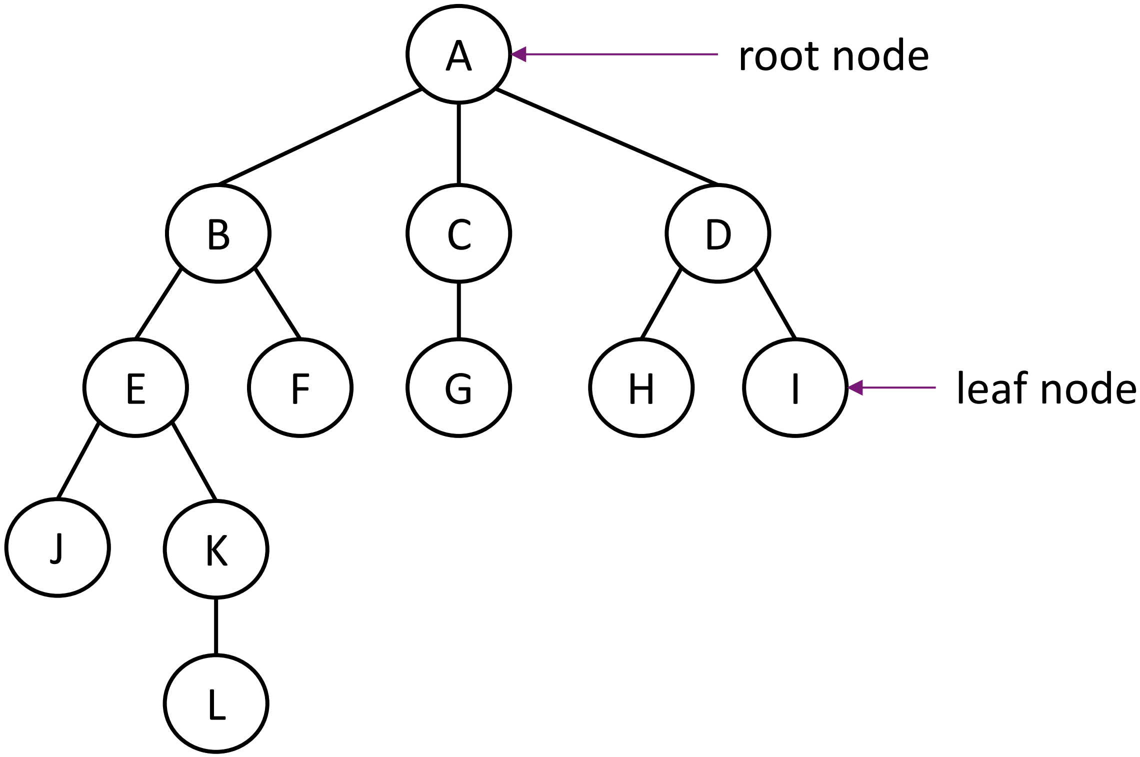
- Similar as stack but with First-In-First-Out (FIFO) order.
- Enqueue: Add an element to the end of the queue.
- Dequeue: Remove the head element from the queue.



3. Trees and Graphs

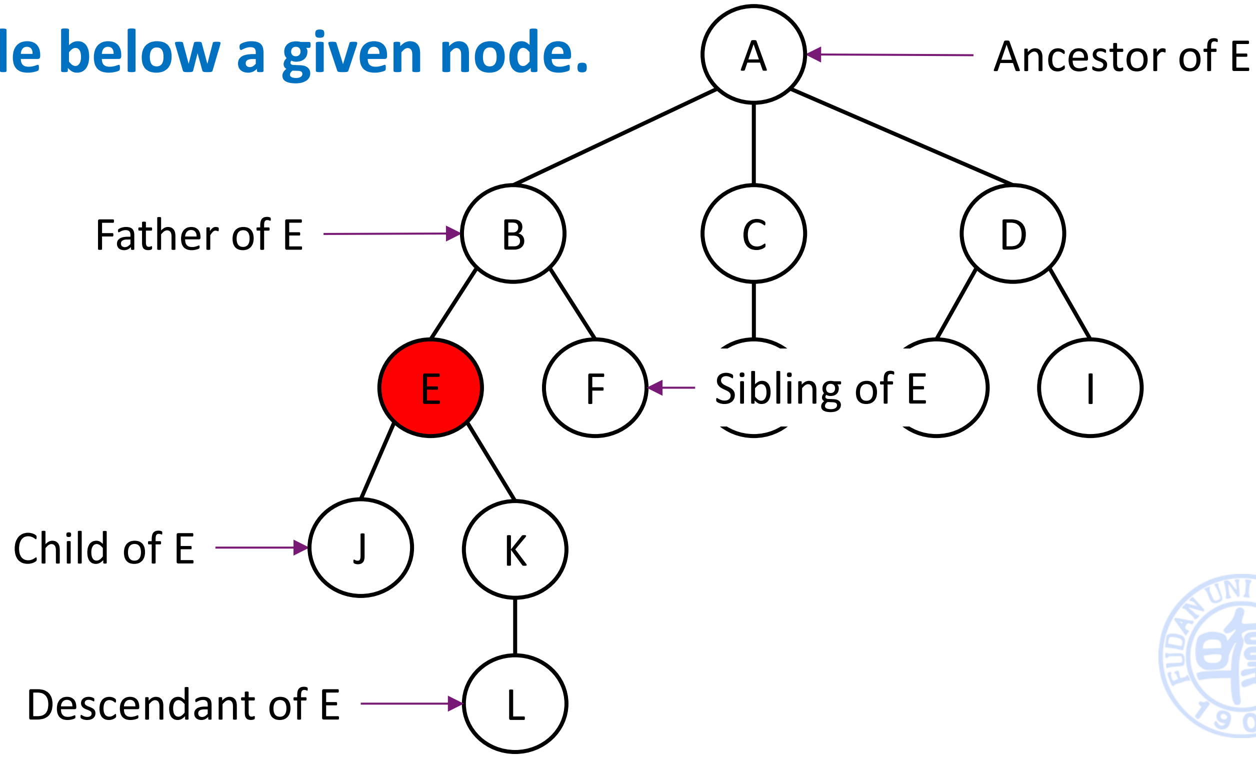
Trees

- Represent a hierarchical relationship among data units.
- There is only one root node.
- Each node may have one or more children except the leaf nodes.



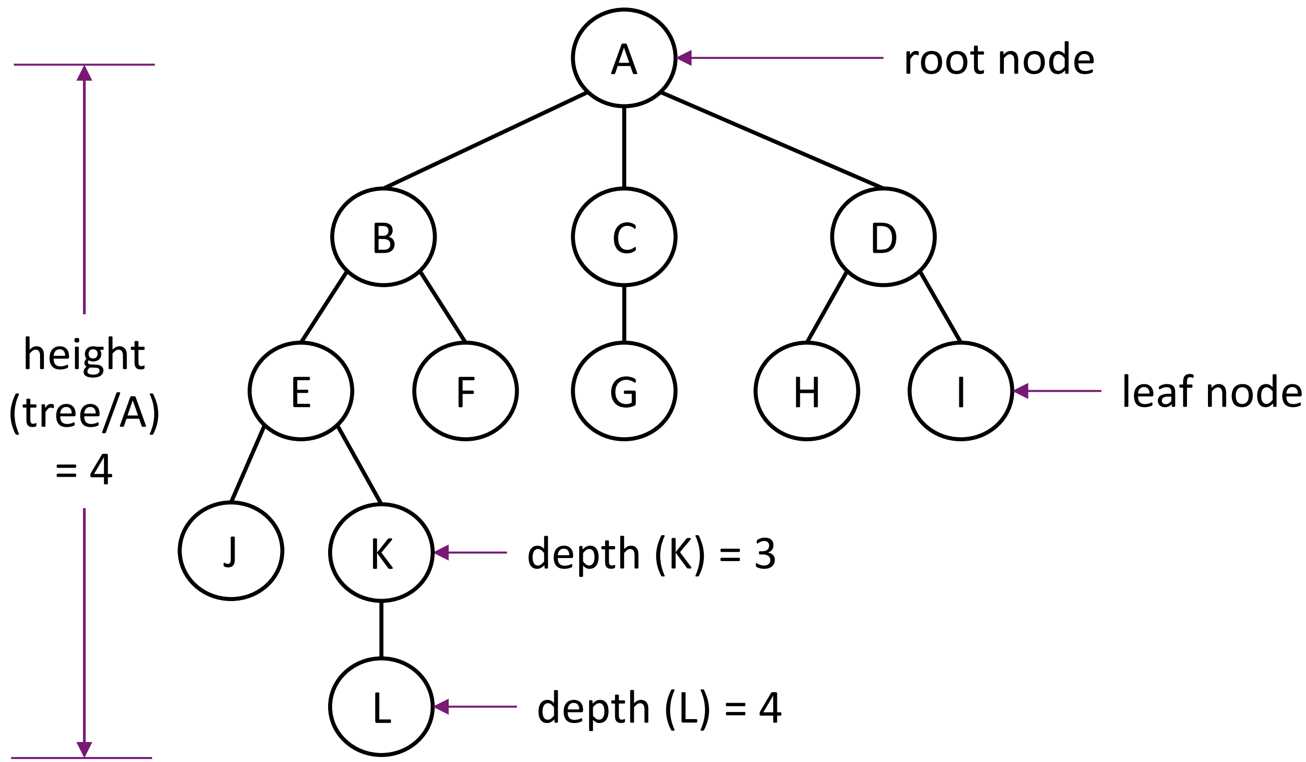
Relationships Among Nodes

- **Father node: The direct parent of a node.**
- **Sibling: Nodes that share the same parent.**
- **Child: A node directly below a given node.**
- **Ancestor: A node above a given node.**
- **Descendant: A node below a given node.**



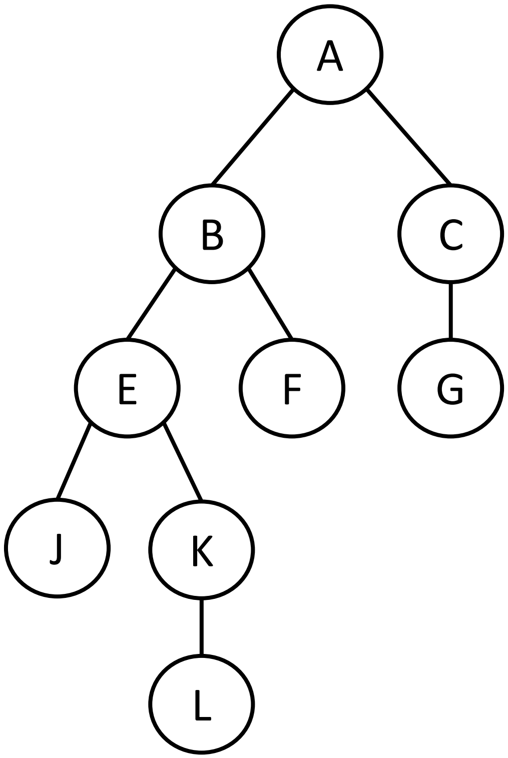
Terminologies

- Height of a node: The longest path (no. of edges) from it to a leaf.
- Height of a tree: Height of the root node.
- Depth of a node: Number of edges from the root to the node.
- Degree of a node: Number of children of the node.

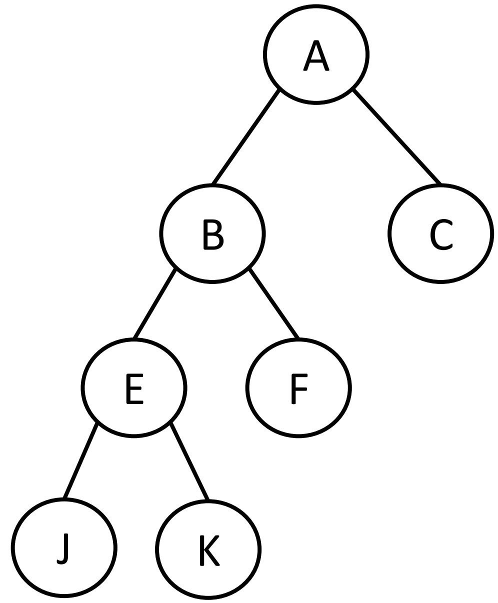


Binary Tree

- Binary tree: The degree of each node on a tree is at most two.
- Full binary tree: The degree of each node on a tree is two or zero.



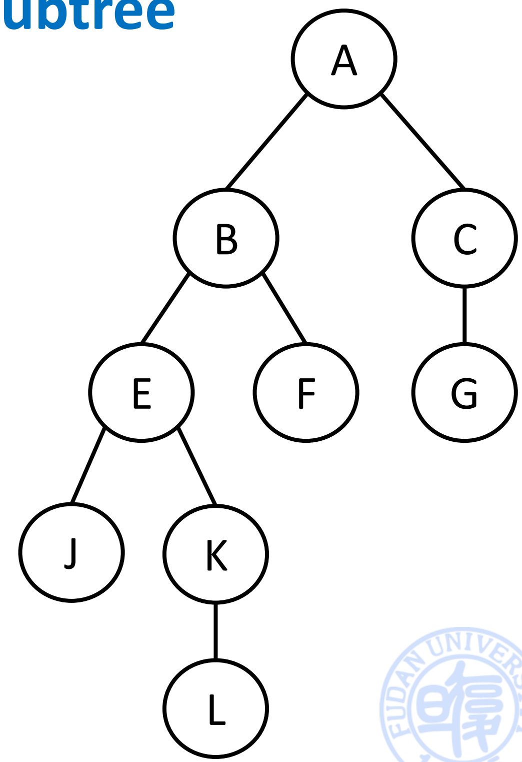
Binary Tree



Full Binary Tree

Tree Traversal

- **Pre-order Traversal: root => left subtree => right subtree**
  - A=>B=>E=>J=>K=>L=>F =>C=>G
- **Post-order Traversal: left subtree => right subtree => root**
  - J=>L=>K=>E=>F=>B=>G=>C=>A
- **In-order Traversal: left subtree => root => right subtree**
  - J=>E=>K=>L=>B=>F=>A=>C=>G



Implement Binary Tree with Python

- **A binary tree has multiple nodes.**
  - One root node.
  - Root and intermediate nodes have at most two children.
  - Leaf nodes have no children.
- **Using Optional[X] type: The value can be either of type X, or None.**

```python
@dataclass
class Node():
    value: int
    left: Optional[Node[int]] = None
    right: Optional[Node[int]] = None

@dataclass
class BinaryTree():
    root: Optional[Node[int]] = None
```

Implement Binary Tree with Python

- **Generic type: Write one class or function that works for many types.**

```python
T = TypeVar("T")

@dataclass
class Node(Generic[T]):
    value: T
    left: Optional[Node[T]] = None
    right: Optional[Node[T]] = None

@dataclass
class BinaryTree(Generic[T]):
    root: Optional[Node[T]] = None
```

Question

- **How to define the data structure of a tree?**
  - The number of child nodes are unlimited.

Implement Tree with Python

```
from __future__ import annotations
from dataclasses import dataclass, field

@dataclass
class TreeNode(Generic[T]):
    value: T
    # Way 1: Not allowed (assigning a mutable object: list).
    # children: List['TreeNode[T]'] = []
    # Way 2: List should be crated before use.
    # children: Optional[List['TreeNode[T]']] = None
    # Way 3: crate a list for each class instance via the factory.
    children: List[TreeNode[T]] = field(default_factory=list)

@dataclass
class Tree(Generic[T]):
    root: Optional[TreeNode[T]] = None
```

Practice (assisted with LLM)

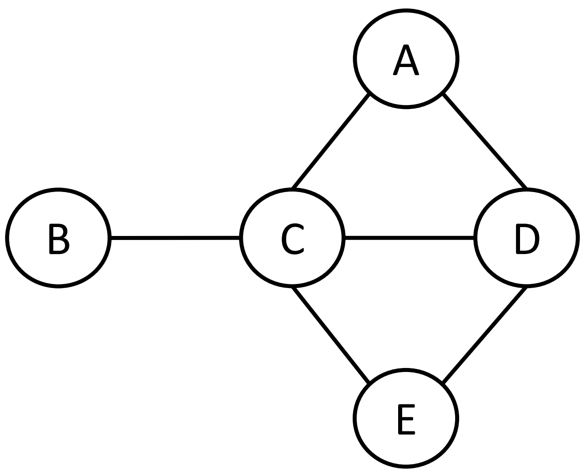
- **Implement a (binary) tree with the following features:**
  - Add: insert a node into the tree
  - Remove: delete a node from the tree
  - Traversal: visit all nodes using DFS or BFS
  - Search: check for the existence of a node

Graphs

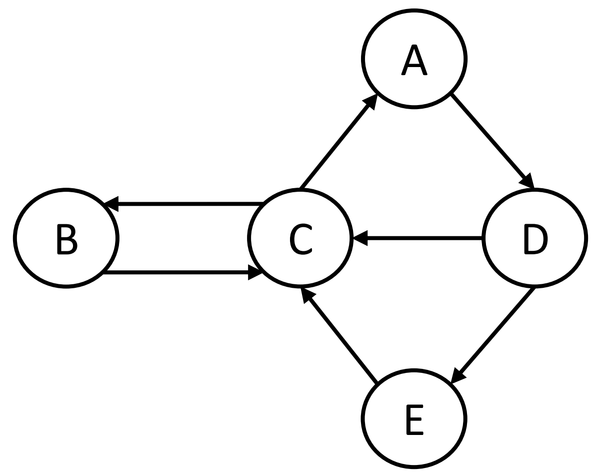
❑Similar to trees except that:

- ➢Graph may have loops, while trees cannot.
- ➢There is no partial order (*e.g.*, parent vs child nodes) among nodes.

❑A graph can be directed or undirected.



undirected graph



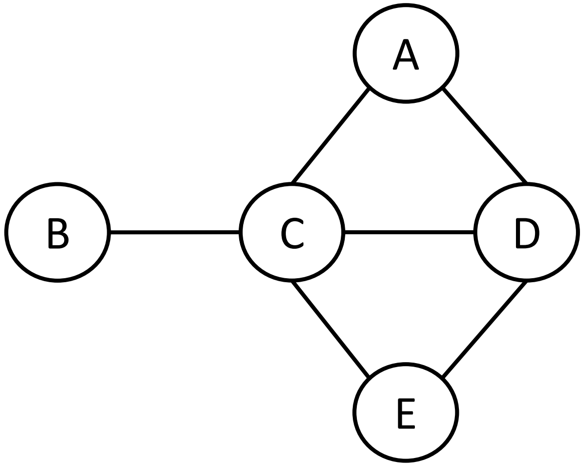
directed graph

Representing Undirected Graphs: Adjacent Matrix

- The adjacency matrix for undirected graph is defined as:

$$adj(i,j) = \begin{cases} 1, \text{if there is an edge between vertex } i \text{ and } j \\ 0, \text{otherwise} \end{cases}$$

- The adjacency matrix for undirected graphs is symmetric.



undirected graph

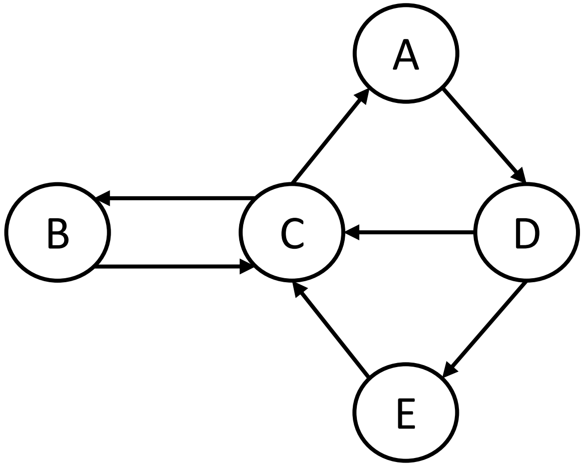
	A	**B**	**C**	**D**	**E**
A	0	0	1	1	0
B	0	0	1	0	0
C	1	1	0	1	1
D	1	0	1	0	1
E	0	0	1	1	0

adjacency matrix

Representing Directed Graphs: Adjacent Matrix

❑The adjacency matrix for directed graph is defined as:

$$adj(i,j) = \begin{cases} 1, \text{if there is an edge from } i \text{ to } j \\ 0, \text{otherwise} \end{cases}$$



directed graph

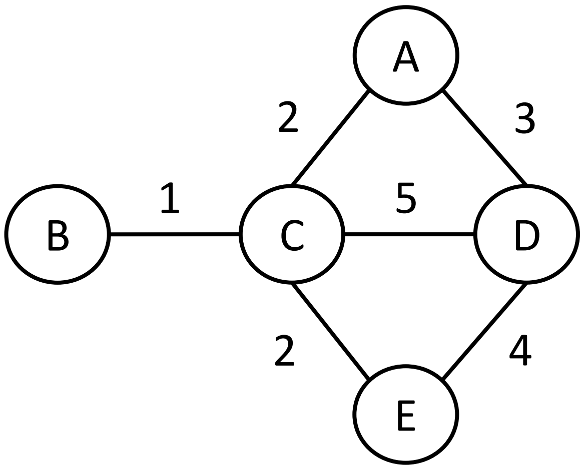
	A	B	C	D	E
A	0	0	0	1	0
B	0	0	1	0	0
C	1	1	0	0	0
D	0	0	1	0	1
E	0	0	1	0	0

adjacency matrix

Weighted Graphs

❑The adjacency matrix for undirected weighted graph is defined as:

$$adj(i,j) = \begin{cases} w, \text{weight between node } i \text{ and } j \\ \infty, \text{no edge between } i \; and \; j \end{cases}$$



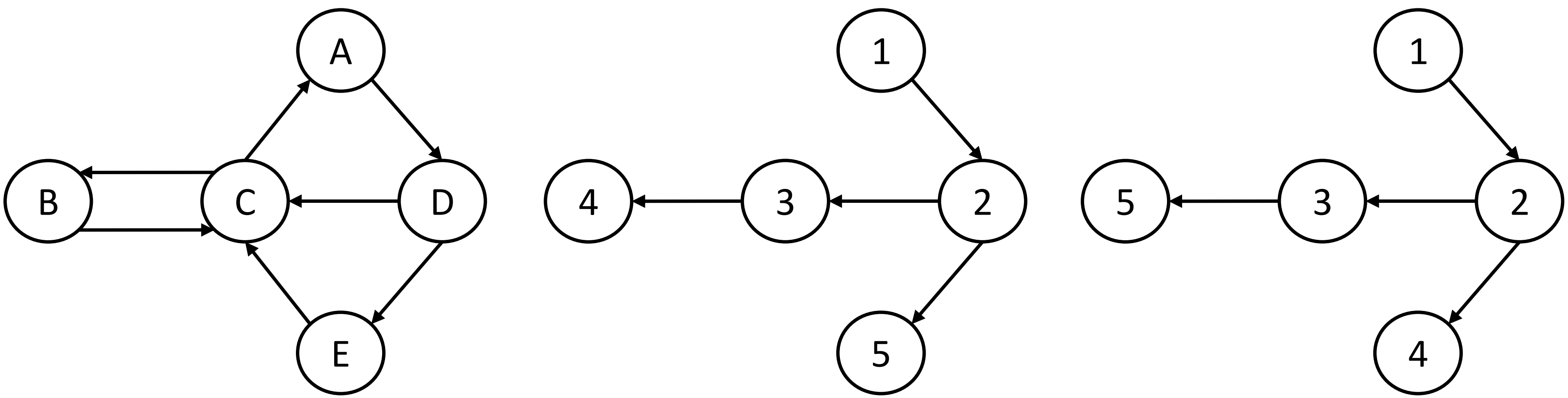
weighted graph

	A	B	C	D	E
A	∞	∞	2	3	∞
B	∞	∞	1	∞	∞
C	2	1	∞	5	2
D	3	∞	5	∞	4
E	∞	∞	2	4	∞

adjacency matrix

Graph Traversal: Two Orders

- **Depth-first Traversal: Explore as far as possible along each branch before backtracking.**
  - A=>D=>C=>B=>C(visited) => back to B=>back to C=>back to D=>E=>C(visited)=>back to D=>back to A
- **Breadth-first Traversal: Explore all the nodes at the present depth level before moving on to nodes at the next depth level.**



directed graph **depth-first** **breadth-first**

Question

❑How to define the data structure of a graph?

1) Unweighted vs Weighted graph.
2) Directed vs Undirected.

Implement Unweighted Graph via Adjacent Matrix

```
@dataclass
class Graph():
    adj: Dict[int, Set[int]] = field(default_factory=dict)
```

❑Using data class with attributes (eq=True, frozen=True):

- ➢eq: Two class instances can be compared by value.
  - ▪ This is required by dict.
- ➢ frozen=True: The dataclass is read only.

```
@dataclass(eq=True, frozen=True)
class Node():
    id: int

@dataclass
class Graph():
    adj: Dict[Node, Set[Node]] = field(default_factory=dict)
```

Implement Weighted Graph via Adjacent Matrix

```
@dataclass
class Node():
    id: int

@dataclass(eq=True, frozen=True)
class Graph():
    adj: Dict[Node, Dict[Node, int]] = field(default_factory=dict)
```

Implement Graph via List

```python
@dataclass
class Node():
    id: int
    # unweighted version
    neighbors: List["Node"] = field(default_factory=list)
    # weighted version:
    List[Tuple["Node", int]] = field(default_factory=list)


@dataclass
class Graph():
    nodes: List[Node] = field(default_factory=list)
```

Another More Readable but Less Efficient Way

```
@dataclass
class Node():
    id: int

@dataclass
class Edge():
    u: Node
    v: Node
    weight: int

@dataclass
class Graph():
    nodes: List[Node] = field(default_factory=list)
    edges: List[Edge] = field(default_factory=list))
```

Practice (assisted with LLM)

- **Implement an undirected/directed graph with the following features:**
  - Add: insert nodes and edges into the graph.
  - Remove: delete nodes and edges from the graph.
  - Traversal: visit all nodes using depth-first or breadth-first order.
  - Search: check for the existence of nodes, edges, or paths.

4. Hash Set and Hash Map

Hash Set

- Python built-in type Set is a hash set.
- Map each value to an index using a hash function.
- Allow fast insertion, deletion, and lookup.
- The search time is constant.
- Tradeoff between space and time.

Index	Value
…	
231	JPM
…	
286	AAPL
…	
295	META
…	
…	

Index Calculation via Hash Function

- A hash function takes some input and outputs a fixed-size integer.

Key
AAPL
MSFT
AMZN
META
TSLA
JPM

Hash

Index
286
314
310
295
308
231

Toy function: sum of ASCII Code

Example: AAPL

'A' = 65

'A' = 65

'P' = 80

'L' = 76

Sum: 65+65+80+76 = 286

Collision and Handling

- **Collision: When two different keys produce the same hash value.**
  - They are mapped to the same slot in the underlying array.
- **Possible handling strategy:**
  - Separate chaining:
    - Each slot in the table stores a linked list.
    - When a collision occurs, simply append the new element to the list.
  - Open addressing (Probing):
    - Search for the next available slot using a probing sequence, *e.g.,* plus an offset.

Hash Table Provided in Python: Set

❑For unique elements collection.

```python
s = set()
s.add("apple")
s.add("banana")
s.add("orange")

s.remove("banana")  # Remove elements
print("apple" in s)    # True

for fruit in s:
    print(fruit)
```

Hash Map

- Python built-in type Dict is a hash map.
- Hash map is a data structure that stores key-value pairs.

Index	Key	Value
...		
231	JPM	
...		
286	AAPL	
...		
295	META	
...		
...		

Hash Map Provided in Python: Dict

- With both key and value.

```
students = {
    "Alice": 20,
    "Bob": 21,
    "Charlie": 19
}
students["Dylan"] = 22
del students["Alice"]
print(students["Bob"])  # Output: 21
```

Practice (assisted with LLM)

- **Design and implement a Hash Map:**
  - Try different solutions for collision handling.
  - Benchmark the performance.