🧩 Basic Level (Conceptual + Theory)

📘 Fundamentals

1. What is a Data Structure?

A **data structure** is a way of organizing, storing, and managing data in a computer so that it can be used efficiently.

In Simple Terms:

Think of a data structure like a container that holds data in a specific format, making it easier to access and perform operations like searching, inserting, deleting, and updating.

Common Types of Data Structures:

1. **Linear Data Structures** (elements arranged in a sequence)
   - **Array** – Fixed-size collection of elements of the same type.
   - **Linked List** – Elements connected using pointers.
   - **Stack** – Last In, First Out (LIFO).
   - **Queue** – First In, First Out (FIFO).
2. **Non-Linear Data Structures**
   - **Tree** – Hierarchical structure (e.g., binary trees, binary search trees).
   - **Graph** – A set of nodes connected by edges.
3. **Hash-based Structures**
   - **Hash Table / Hash Map** – Key-value pairs with fast access via hashing.

Why Are Data Structures Important?

- **Efficiency:** Choosing the right data structure can drastically improve performance.
- **Memory Management:** Helps optimize memory usage.
- **Problem Solving:** Certain algorithms require specific data structures to work effectively.

Example:

To store a list of student names:

- An **array** might be a good choice if the number of students is fixed.
- A **linked list** could work better if students are frequently added or removed.

---

2. What are the types of Data Structures?

Data structures can be broadly categorized into several types based on how they organize data and how they operate. Here's a clear breakdown:

🔹 1. Primitive Data Structures

These are the basic building blocks provided by programming languages.

- **Integer**
- **Float**
- **Character**
- **Boolean**
- **Pointer**

🔹 2. Non-Primitive Data Structures

These are more complex and are built using primitive data types.

They are further divided into:

✅ A. Linear Data Structures

Data elements are arranged in a sequence or a linear order.

Data Structure	Description	Example Use Case
Array	Fixed-size collection of elements of the same type.	Storing list of student IDs
Linked List	Elements (nodes) are connected using pointers.	Dynamic memory use, frequent insertion/deletion
Stack	LIFO (Last In First Out) structure.	Undo functionality in editors

Data Structure	Description	Example Use Case
Queue	FIFO (First In First Out) structure.	Print queue, task scheduling
Deque	Double-ended queue (insert/delete from both ends).	Browser history navigation

---

✅ B. Non-Linear Data Structures

Data elements are not arranged in a sequence, but in a hierarchical or interconnected way.

Data Structure	Description	Example Use Case
Tree	Hierarchical structure with parent-child relationships.	File systems, organizational charts
Binary Tree	Each node has at most two children.	Searching, sorting
Binary Search Tree (BST)	A tree where left < root < right	Fast lookup, insertion
Heap	Special tree for priority-based access.	Priority queues, scheduling
Trie	Tree for storing strings efficiently.	Autocomplete systems
Graph	Set of nodes connected by edges.	Social networks, maps, web pages

---

✅ C. Hash-based Data Structures

These use a **hash function** to map keys to values for fast access.

Data Structure	Description	Example Use Case
Hash Table / Hash Map	Key-value pair storage with fast access.	Database indexing
Hash Set	Stores unique values only.	Duplicate detection

Summary Table:

Category	Examples
Primitive	int, char, float, boolean
Linear	Array, Linked List, Stack, Queue
Non-Linear	Tree, Graph
Hash-based	Hash Table, Hash Map, Hash Set

---

3. What is the difference between linear and non-linear data structures?

The **difference between linear and non-linear data structures** lies mainly in how the data elements are organized and accessed. Here's a clear comparison:

🔸 Linear Data Structures

✅ Definition:

In linear data structures, elements are **arranged in a sequential or linear order**. Each element is connected to its previous and next element.

📌 Characteristics:

- Elements are stored **one after another**.
- Traversal is done in a **single level**.
- Typically **one element per level**.
- Simple implementation and memory usage.

📋 Examples:

- **Array**
- **Linked List**
- **Stack**
- **Queue**

🧠 Analogy:

Think of it like a **line of people** in a queue—each person has one person before and after them.

🔸 Non-Linear Data Structures

✅ Definition:

In non-linear data structures, elements are **not arranged sequentially**. They are organized **hierarchically or in complex relationships** (e.g., parent-child or graph connections).

📌 Characteristics:

- Elements can be connected in **multiple ways**.
- Traversal is done across **multiple levels**.
- Complex structure and memory use.
- More powerful for representing **real-world relationships**.

📋 Examples:

- **Tree**
- **Binary Search Tree**
- **Heap**
- **Graph**
- **Trie**

🧠 Analogy:

Think of it like a **family tree** or a **web** where one person can have multiple children or connections.

🔄 Key Differences Table:

Feature	Linear Data Structure	Non-Linear Data Structure
Data Arrangement	Sequential (line or list)	Hierarchical or networked
Traversal	Single level (one by one)	Multiple paths or levels
Memory Usage	Generally efficient and simple	Can be complex and dynamic
Examples	Array, Stack, Queue, Linked List	Tree, Graph, Heap, Trie
Real-world Use Cases	Lists, task scheduling, memory buffers	File systems, routing, relationships

---

4. What is an Abstract Data Type (`ADT`)?

✅ What is an Abstract Data Type (`ADT`)?

An **Abstract Data Type (`ADT`)** is a **logical description of how data is organized and what operations can be performed on it**, without specifying how it's implemented.

🔍 Key Idea:

> **`ADT` focuses on *what* operations are to be performed, not *how* they are implemented.**

It hides the internal details (data storage and algorithms) and only shows the functionality.

📌 Characteristics of `ADT`:

- **Encapsulation**: Implementation details are hidden.
- **Interface-based**: Defined by the operations it supports (like `insert()`, `delete()`, `search()`).
- **Implementation Independent**: Can be implemented using different data structures.

📋 Common Examples of `ADTs`:

ADT	Common Operations	Can Be Implemented Using
List	Insert, Delete, Traverse	Array, Linked List
Stack	Push, Pop, Peek	Array, Linked List
Queue	Enqueue, Dequeue, Peek	Array, Linked List
Deque	Insert/Delete at both ends	Array, Doubly Linked List
Map	Insert, Delete, Search by Key	Hash Table, Binary Search Tree
Set	Add, Remove, Contains	Hash Table, Tree

🧠 Real-Life Analogy:

Think of a **vending machine**:

- You interact with buttons (the interface/ADT).
- You don’t know how it works internally (mechanism/data structure).
- You only care that pressing a button gives you a snack (operation/output).

✅ Why Use ADTs?

- **Improves modularity** – easy to swap implementations.
- **Enhances code readability** – focuses on operations, not internals.
- **Supports abstraction and** `reusability`.

---

5. What are static vs dynamic data structures?

✅ Static vs Dynamic Data Structures

The difference between **static** and **dynamic** data structures lies in how memory is allocated and managed during the program's execution.

🔸 Static Data Structures

✅ Definition:

A **static data structure** has a **fixed size**, defined at compile-time. Memory is allocated once and does not change during execution.

📌 Characteristics:

- Fixed memory size
- Easy to implement
- Less flexible but more memory-efficient if size is known in advance
- Faster access due to predictable memory layout

📋 Examples:

- **Array**
- **Static Stack (implemented with an array)**
- **Static Queue**

🧠 Analogy:

Think of it like a **train with a fixed number of seats** — you can't add or remove seats once it's built.

🔸 Dynamic Data Structures

✅ Definition:

A **dynamic data structure** can **grow or shrink in size** during program execution. Memory is allocated and deallocated at runtime.

📌 Characteristics:

- Memory is managed during runtime
- More flexible and scalable
- Slightly more complex to implement
- May use more memory due to overhead (pointers, etc.)

📋 Examples:

- **Linked List**
- **Dynamic Stack (linked list-based)**
- **Dynamic Queue**
- **Trees**
- **Graphs**

🧠 Analogy:

Like a **building made of LEGO bricks** — you can add or remove parts as needed.

🔄 Key Differences Table:

Feature	Static Data Structure	Dynamic Data Structure
Memory Allocation	Compile-time	Runtime
Size	Fixed	Flexible
Performance	Faster access, less overhead	Slight overhead for flexibility
Examples	Array, Static Stack, Static Queue	Linked List, Dynamic Stack, Tree

Feature	Static Data Structure	Dynamic Data Structure
Flexibility	Low	High

✅ Summary:

- Use **static** structures when the size is known and fixed.
- Use **dynamic** structures when the size may change during execution.

📦 Arrays

6. What is an array?

✅ What is an Array?

An **array** is a **linear data structure** that stores a **collection of elements** (typically of the **same data type**) in **contiguous memory locations**.

🔍 Key Characteristics:

Feature	Description
Fixed size	Once defined, the size of the array cannot change (in most languages).
Same data type	All elements must be of the same type (e.g., all integers, all floats).
Index-based access	Elements can be accessed directly using an index (starting at 0).
Efficient traversal	Arrays support fast access and iteration.

📌 Syntax Example

Python:

```
arr = [10, 20, 30, 40]
```

```
print(arr[2])  # Output: 30
```

C++:

```
int arr[4] = {10, 20, 30, 40};
cout << arr[2];  // Output: 30
```

🧠 Visual Representation:

```
Index:     0    1    2    3
Element:  10   20   30   40
```

Access `arr[2]` gives `30`.

✅ Advantages of Arrays:

- **Fast access** using index (`O(1)` time).
- **Easy to implement** and use.
- **Memory efficiency** (compact storage).

❌ Disadvantages:

- **Fixed size** (can't grow or shrink dynamically).
- **Costly insertions/deletions** (especially in the middle).
- **Wasted memory** if the array is not fully used.

📋 Common Operations:

Operation	Description	Time Complexity
Access	Get element by index	O(1)
Insertion	Add element (middle or front)	O(n)
Deletion	Remove element (middle or front)	O(n)
Search	Find element by value	O(n)
Traverse	Go through all elements	O(n)

Here's a quick breakdown of all three topics you mentioned, with clear explanations and comparisons. You can dive deeper into any of them afterward.

🔷 1. Multi-Dimensional Arrays

✅ What is it?

A **multi-dimensional array** is an array of arrays—used to represent data in matrix or tabular form (`2D` , `3D` , etc.).

📌 Example (`2D` Array - like a matrix):

Python:

```python
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix[1][2])  # Output: 6
```

C++:

```cpp
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
cout << matrix[1][2]; // Output: 6
```

✅ Real-world use:

- Game boards (chess, tic-tac-toe)
- Matrices in mathematics
- Grids in images or maps

🔷 2. Dynamic Arrays

✅ What is it?

A **dynamic array** is an array that can **grow or shrink** at runtime. It overcomes the size limitation of static arrays.

🧠 Example:

- In **Python**, lists are dynamic arrays.
- In **C++**, you can use `std::vector`.

Python:

```python
arr = [1, 2, 3]
arr.append(4)
print(arr)  # Output: [1, 2, 3, 4]
```

C++:

```cpp
#include <vector>
vector<int> arr = {1, 2, 3};
arr.push_back(4);
```

✅ Internally:

Dynamic arrays **resize automatically** when full (usually by doubling the size), but this can have performance costs occasionally.

🔷 3. Array vs Linked List

Feature	Array	Linked List
Size	Fixed (unless dynamic)	Dynamic (grows/shrinks easily)
Memory	Contiguous memory	Non-contiguous (uses pointers)
Access time	Fast (O(1) index access)	Slow (O(n), must traverse)
Insertion/Deletion	Slow (O(n) in middle)	Fast (O(1) if pointer is known)
Overhead	Low	Higher (extra memory for pointers)

🧠 Analogy:

- **Array** = Books on a shelf: Easy to find by position, but hard to insert in between.
- **Linked List** = Chain of paper clips: Easy to add/remove clips, but hard to jump to the 50th one.

✅ Summary:

- Use **multi-dimensional arrays** when modeling tables, grids, or matrices.
- Use **dynamic arrays** when the size of the array isn't known in advance.
- Use **linked lists** when you need frequent insertions/deletions, especially in the middle of data.

Would you like to:

- Dive deeper into **dynamic arrays vs linked lists**?
- Learn about **3D arrays**?
- See implementation examples in a specific language like Python or C++?

---

7. What are the advantages and disadvantages of arrays?

✅ Advantages and Disadvantages of Arrays

Arrays are one of the most fundamental data structures in programming. While they are powerful, they come with both strengths and limitations.

✅ Advantages of Arrays

Advantage	Description
🔹 **Fast Access (O(1))**	You can access any element instantly using its index. Example: `arr[3]`
🔹 **Memory Efficiency**	Elements are stored in **contiguous memory**, which allows efficient memory usage.
🔹 **Ease of Iteration**	Easy to loop through using loops like `for` or `while`.
🔹 **Simple Implementation**	Arrays are supported directly in most programming languages with simple syntax.
🔹 **Efficient for Static Data**	Ideal when the number of elements is fixed or predictable.

❌ Disadvantages of Arrays

Disadvantage	Description
🔸 **Fixed Size**	Once declared, the size cannot be changed (in static arrays). This can lead to **wasted memory** or **overflow**.
🔸 **Costly Insertion/Deletion**	Adding or removing elements (especially in the middle) requires **shifting** elements → **O(n)** time.
🔸 **No Dynamic Memory Management (in static arrays)**	Can't grow or shrink at runtime unless using a dynamic version (like Python lists or C++ vectors).
🔸 **Wasted Space (if over-allocated)**	If you overestimate the size needed, unused space sits idle.
🔸 **Same Data Type**	All elements must be of the **same type** in most languages (e.g., all integers).

🧠 Example:

Imagine an array of size 5:

```
[10, 20, 30, 40, 50]
```

- **Access** `arr[2]` → Fast (O(1)) → gives `30`
- **Insert** `25` **at index 2** → Need to shift `30, 40, 50` → Slower (O(n))
- **Delete** `20` **at index 1** → Need to shift `30, 40, 50` left

✅ When to Use Arrays:

- When you **know the number of elements** ahead of time.
- When **fast indexing** is important.
- When **memory layout matters** (e.g., in systems programming or performance-critical code).

---

8. How do you insert and delete elements in an array?

✅ 1. Inserting an Element into an Array

📌 Case A: Inserting at a Specific Index (e.g., middle or beginning)

Since arrays are stored in **contiguous memory**, inserting at a position requires **shifting elements** to make room.

➕ Steps:

1. Check if the array has space (for static arrays).
2. Shift elements to the **right** from the insertion point.
3. Insert the new element at the target index.

🧠 Example:

Insert `25` at index `2` in:

```
[10, 20, 30, 40]
```

👉 Result after shifting and inserting:

```
[10, 20, 25, 30, 40]
```

🧮 Time Complexity:

- **Best case (insert at end)**: O(1)
- **Worst case (insert at start or middle)**: O(n)

✅ 2. Deleting an Element from an Array

📌 Case A: Deleting at a Specific Index

You must **shift elements to the left** to fill the gap.

➖ Steps:

1. Identify the index to delete.
2. Shift all elements to the left, starting from the next index.
3. Optionally, reduce the size (if dynamic).

🧠 Example:

Delete element at index `1` in:

```
[10, 20, 30, 40]
```

👉 Result after deletion:

```
[10, 30, 40]
```

🧮 Time Complexity:

- **Best case (delete last element)**: O(1)
- **Worst case (delete first or middle)**: O(n)

🔍 Code Examples

🐍 Python (using list)

Insert:

```
arr = [10, 20, 30, 40]
arr.insert(2, 25)  # Inserts 25 at index 2
print(arr)         # Output: [10, 20, 25, 30, 40]
```

Delete:

```
arr.pop(2)         # Removes element at index 2 (25)
print(arr)         # Output: [10, 20, 30, 40]
```

💡 Summary Table

Operation	Description	Time Complexity
Insert at end	Add to last position	O(1)
Insert at start/middle	Shift elements, insert new one	O(n)
Delete from end	Remove last element	O(1)
Delete from start/middle	Shift elements to fill gap	O(n)

---

9. What is the time complexity for insertion, deletion, and access in arrays?

⏱️ Time Complexity in Arrays

Operation	Time Complexity	Explanation
Access	**O(1)**	Direct access via index, e.g., `arr[5]`.
Insertion		
- At end	**O(1)** (if space available)	Just place element at the next free index.
- At beginning or middle	**O(n)**	Need to shift elements to the right to make space.
Deletion		
- At end	**O(1)**	Remove last element, no shifting needed.
- At beginning or middle	**O(n)**	Need to shift elements to the left to fill the gap.

Why?

- **Access is O(1)** because arrays store elements in contiguous memory, so the address of any element can be calculated directly.
- **Insertion/deletion at the end is O(1)** for dynamic arrays that have reserved space.
- **Insertion/deletion elsewhere is O(n)** due to shifting elements to maintain the order.

9. Difference between 1D, 2D, and jagged arrays.

🔹 1D Array (One-Dimensional Array)

- **Structure:** A simple linear sequence of elements.
- **Memory:** Contiguous block of memory.
- **Access:** Single index used to access elements.
- **Example Use Case:** List of student scores, names, temperatures.

Example:

```
arr = [10, 20, 30, 40]
print(arr[2])  # Output: 30
```

🔹 2D Array (Two-Dimensional Array)

- **Structure:** Array of arrays (a matrix or grid).
- **Memory:** Contiguous block or row-major/column-major order (depending on language).
- **Access:** Two indices—one for row, one for column.
- **Example Use Case:** Chess board, spreadsheet data, image pixels.

Example:

```
matrix = [
  [1, 2, 3],
  [4, 5, 6]
]
print(matrix[1][2])  # Output: 6
```

🔹 Jagged Array (Array of Arrays with Varying Lengths)

- **Structure:** An array whose elements are arrays of **different lengths**.
- **Memory:** Each sub-array can be stored separately; not necessarily contiguous.
- **Access:** Two indices, but sub-arrays may vary in length.
- **Example Use Case:** Storing different numbers of test scores for students.

Example (Python-like):

```
jagged = [
  [1, 2],
  [3, 4, 5],
  [6]
]
print(jagged[1][2])  # Output: 5
```

🔄 Summary Table

Feature	1D Array	2D Array	Jagged Array
Dimensions	One	Two	Two (rows of varying lengths)
Element Access	`arr[i]`	`arr[i][j]`	`arr[i][j]`

Feature	1D Array	2D Array	Jagged Array
Length of Rows	N/A	All rows have equal length	Rows can have different lengths
Memory Layout	Contiguous block	Typically contiguous	Sub-arrays can be non-contiguous
Use Case	Simple lists	Grids, matrices	Variable-length rows, irregular data

---

🔗 Linked List

11. What is a linked list?

✅ What is a Linked List?

A **linked list** is a **linear data structure** where elements, called **nodes**, are stored in **non-contiguous memory locations**. Each node contains:

- **Data** (the value)
- **Pointer (or reference)** to the **next node** in the sequence

---

🔍 Key Features:

- **Dynamic size:** Can grow or shrink at runtime by allocating/ `deallocating` nodes.
- **Non-contiguous storage:** Nodes can be anywhere in memory, connected via pointers.
- **Efficient insertions/deletions:** Especially at the beginning or middle (no shifting required).

🧠 Visual Representation:

```
[Data | Next] -> [Data | Next] -> [Data | Next] -> NULL
```

For example, a linked list with elements `10 -> 20 -> 30` looks like:

```
[10 | *] -> [20 | *] -> [30 | NULL]
```

Types of Linked Lists:

- **Singly Linked List:** Each node points only to the next node.
- **Doubly Linked List:** Nodes have pointers to both next and previous nodes.
- **Circular Linked List:** Last node points back to the first node.

Advantages:

- Dynamic size
- Easy insertion/deletion without shifting elements
- Flexible memory use

Disadvantages:

- No random access (must traverse from head to find an element) — O(n) access time
- Extra memory needed for pointers

Simple Example in Python (Singly Linked List Node):

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

implementation

full implementation of a **Singly Linked List** in Python with basic operations: **insertion**, **deletion**, and **traversal**.

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

class LinkedList:
    def __init__(self):
        self.head = None
```

```
    # Insert at the end
    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    # Insert at the beginning
    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    # Delete first node with given data
    def delete(self, key):
        curr = self.head
        prev = None

        while curr and curr.data != key:
            prev = curr
            curr = curr.next

        if not curr:  # Key not found
            return

        if prev is None:  # Deleting head
            self.head = curr.next
        else:
            prev.next = curr.next

    # Traverse and print list
    def print_list(self):
        curr = self.head
        while curr:
            print(curr.data, end=" -> ")
            curr = curr.next
        print("None")

# Example usage:
ll = LinkedList()
ll.append(10)
ll.append(20)
ll.prepend(5)
ll.print_list()     # Output: 5 -> 10 -> 20 -> None
```

```
ll.delete(10)
ll.print_list()     # Output: 5 -> 20 -> None
```

What this does:

- **append(data)**: Adds a node at the end.
- **prepend(data)**: Adds a node at the start.
- **delete(key)**: Removes the first node with the specified value.
- **print_list()**: Prints the whole linked list.

---

12. Compare arrays and linked lists.

Feature	Arrays	Linked Lists
Memory Allocation	Contiguous block of memory	Non-contiguous nodes linked via pointers
Size	Fixed (static arrays) or resizable (dynamic arrays)	Dynamic, can grow/shrink easily
Access Time	O(1) — direct indexing by position	O(n) — must traverse nodes sequentially
Insertion Time	O(n) — need to shift elements	O(1) — insertions at head or known position
Deletion Time	O(n) — need to shift elements	O(1) — deletion at head or known node
Memory Overhead	Low — only stores data	Higher — stores data + pointer(s)
Cache Locality	High — good cache performance	Low — scattered memory locations
Ease of Use	Simple syntax and access	More complex (requires pointer handling)
Use Cases	When fast random access needed, size known	When frequent insertions/deletions, unknown size
Examples	Storing fixed-size data, lookup tables	Implementing stacks, queues, graphs

---

Summary:

- **Arrays** are great for fast **random access** and **memory efficiency** but poor at insertions/deletions in the middle.
- **Linked lists** excel at **dynamic size management** and **efficient insertions/deletions**, but have slower access times and higher memory overhead.

---

13. What are types of linked lists (singly, doubly, circular)?

---

1. Singly Linked List

- **Structure:** Each node contains data and a pointer to the **next node** only.
- **Traversal:** One-way — from head to the end (`NULL`).
- **Use case:** Simple linear lists where only forward traversal is needed.

Visualization:

```
[Data | Next] -> [Data | Next] -> [Data | NULL]
```

---

2. Doubly Linked List

- **Structure:** Each node contains data, a pointer to the **next node**, and a pointer to the **previous node**.
- **Traversal:** Two-way — you can go forwards and backwards.
- **Use case:** When you need efficient bidirectional traversal or deletion of nodes when only the node pointer is known.

Visualization:

```
NULL <- [Prev | Data | Next] <-> [Prev | Data | Next] <-> [Prev | Data |
Next] -> NULL
```

---

3. Circular Linked List

- **Structure:** The last node points back to the first node, forming a loop.

- Can be **singly circular** (nodes have only next pointer) or **doubly circular** (nodes have next and prev pointers).
- **Traversal:** Can keep looping through the list indefinitely.
- **Use case:** Useful for buffering, round-robin scheduling, or when you want continuous cycling through data.

Visualization (Singly Circular):

```
[Data | Next] -> [Data | Next] -> [Data | Next] -+
       ^                                         |
       +-----------------------------------------+
```

Summary Table:

Type	Next Pointer	Prev Pointer	Circular?	Traversal	Use Case
Singly Linked List	Yes	No	No	Forward only	Simple lists
Doubly Linked List	Yes	Yes	No	Forward & Backward	Browsers history, undo-redo
Circular Linked List	Yes	Optional	Yes	Continuous loop	Buffers, scheduling algorithms

---

13. How do you detect a cycle in a linked list?

Detecting a cycle in a linked list is a classic problem! The most common and efficient method is called **Floyd's Cycle-Finding Algorithm** (also known as the **Tortoise and Hare algorithm**).

How Floyd's Cycle-Finding Algorithm Works:

- Use **two pointers**:
  - **Slow pointer** (`slow`) moves **one step** at a time.
  - **Fast pointer** (`fast`) moves **two steps** at a time.

- If there is **no cycle**, the `fast` pointer will reach the end (`NULL`).
- If there **is a cycle**, the `fast` pointer will eventually **meet** the `slow` pointer inside the cycle.

Steps:

1. Initialize `slow` and `fast` pointers to the head of the list.
2. Move `slow` by one node and `fast` by two nodes in each iteration.
3. If `fast` becomes `NULL` or `fast.next` is `NULL`, the list has no cycle.
4. If `slow` equals `fast` at any point, a cycle exists.

Python Code Example:

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

def has_cycle(head):
    slow = head
    fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True  # Cycle detected

    return False  # No cycle

# Example Usage:
# Creating a cycle linked list for testing
head = Node(1)
head.next = Node(2)
head.next.next = Node(3)
head.next.next.next = head.next  # Cycle here

print(has_cycle(head))  # Output: True
```

Time and Space Complexity:

- **Time:** O(n) — Both pointers traverse the list at most once.
- **Space:** O(1) — Uses only two pointers, constant space.

---

13. How do you reverse a linked list?

Reversing a linked list is a common operation! The idea is to **reverse the direction of the `next` pointers** so that the last node becomes the head, and the head becomes the last node.

How to Reverse a Singly Linked List (Iterative Method):

Concept:

- Use **three pointers**: `prev`, `current`, and `next_node`.
- Traverse the list once.
- For each node, point its `next` to the previous node.
- Move forward until all nodes are reversed.

Step-by-step:

1. Initialize:
   - `prev = None`
   - `current = head`
2. Iterate while `current` is not `None`:
   - Store `next_node = current.next` (save next node)
   - Reverse the pointer: `current.next = prev`
   - Move `prev` to current node: `prev = current`
   - Move `current` to next node: `current = next_node`
3. At the end, `prev` will be the new head of the reversed list.

Python Code:

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

def reverse_linked_list(head):
    prev = None
    current = head

    while current:
        next_node = current.next   # Save next node
        current.next = prev        # Reverse pointer
        prev = current             # Move prev forward
        current = next_node        # Move current forward

    return prev  # New head of reversed list

# Example Usage:
# Creating a linked list 1 -> 2 -> 3 -> None
head = Node(1)
head.next = Node(2)
head.next.next = Node(3)

# Reverse it
new_head = reverse_linked_list(head)

# Print reversed list
current = new_head
while current:
    print(current.data, end=" -> ")
    current = current.next
print("None")
# Output: 3 -> 2 -> 1 -> None
```

Recursive Method (optional):

You can also reverse a linked list recursively, but the iterative approach is more common due to simplicity and better space efficiency.

🧱 Stack & Queue

16. What is a stack? What are its applications?

What is a Stack?

A **stack** is a **linear data structure** that follows the **LIFO** (Last In, First Out) principle. This means the **last element added** to the stack is the **first one to be removed**.

Key Operations:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove the top element from the stack.
- **Peek/Top:** View the top element without removing it.
- **`IsEmpty`:** Check if the stack is empty.

Visualization:

```
Top
 ↑
| 5 |  <-- Last pushed, first to pop
| 4 |
| 3 |
| 2 |
| 1 |
```

Applications of Stack

1. **Function Call Management:**
   The system uses a call stack to keep track of active functions and return addresses.
2. **Expression Evaluation and Syntax Parsing:**
   - Converting infix expressions to `postfix/prefix`.
   - Evaluating `postfix` expressions.
3. **Backtracking Algorithms:**
   - Maze solving, puzzles (like Sudoku), undo mechanisms.
4. **Browser History Navigation:**
   Keeps track of visited pages to allow going back.
5. **Recursive Algorithm Implementation:**
   Can replace recursion with an explicit stack.
6. **Balanced Parentheses Checking:**
   Verify if every opening bracket has a corresponding closing bracket.

Implementation

Here's a simple **Stack implementation in Python** using a list:

```python
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty stack")
        return self.items[-1]

    def size(self):
        return len(self.items)

# Example Usage:
stack = Stack()

stack.push(10)
stack.push(20)
stack.push(30)

print(stack.peek())  # Output: 30

print(stack.pop())   # Output: 30
print(stack.pop())   # Output: 20

print(stack.is_empty())  # Output: False
print(stack.pop())       # Output: 10
print(stack.is_empty())  # Output: True
```

Explanation:

- **push:** Adds an element to the top.

- **pop:** Removes and returns the top element.
- **peek:** Returns the top element without removing it.
- **is_empty:** Checks if the stack has no elements.
- **size:** Returns number of elements in the stack.

---

implement using single link list

how to implement a **Stack using a Singly Linked List** in Python:

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

class Stack:
    def __init__(self):
        self.top = None  # Points to the top node of the stack

    def is_empty(self):
        return self.top is None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top  # New node points to current top
        self.top = new_node       # Update top to new node

    def pop(self):
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        popped_node = self.top
        self.top = self.top.next  # Move top to next node
        return popped_node.data

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty stack")
        return self.top.data

    def size(self):
        count = 0
        current = self.top
        while current:
            count += 1
            current = current.next
```

```
        return count

# Example Usage:
stack = Stack()

stack.push(10)
stack.push(20)
stack.push(30)

print(stack.peek())  # Output: 30

print(stack.pop())   # Output: 30
print(stack.pop())   # Output: 20

print(stack.is_empty())  # Output: False
print(stack.pop())       # Output: 10
print(stack.is_empty())  # Output: True
```

How it works:

- The **top** pointer always points to the latest added node.
- `push` adds a new node at the front.
- `pop` removes the node at the front.
- `peek` returns the data of the top node without removing it.

---

16. What is a queue?

What is a Queue?

A **queue** is a **linear data structure** that follows the **FIFO** (First In, First Out) principle. This means the **first element added** to the queue is the **first one to be removed**.

Key Operations:

- **Enqueue:** Add an element to the **rear** (end) of the queue.
- **Dequeue:** Remove an element from the **front** of the queue.
- **Peek/Front:** View the front element without removing it.
- **`IsEmpty` :** Check if the queue is empty.

Visualization:

```
Front                                  Rear
  ↓                                     ↓
[ 10 ] -> [ 20 ] -> [ 30 ] -> [ 40 ]
```

- The element `10` was added first and will be removed first.
- New elements are added at the rear.

Use Cases of Queue

1. **Scheduling:** CPU task scheduling, printer job scheduling.
2. **Breadth-First Search (`BFS`):** Used in graph and tree traversals.
3. **Handling requests:** In web servers, managing requests in order.
4. **Buffering:** Data buffers, IO Buffers, asynchronous data transfer.
5. **Real-world queues:** Like ticket queues, call center systems.

---

implement using python list

simple **Queue implementation in Python using a list**:

```python
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)  # Add to the end of the list

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)  # Remove from the front (index 0)

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.items[0]  # Front element
```

```
    def size(self):
        return len(self.items)

# Example Usage:
q = Queue()

q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

print(q.peek())    # Output: 10
print(q.dequeue()) # Output: 10
print(q.dequeue()) # Output: 20
print(q.is_empty())# Output: False
print(q.dequeue()) # Output: 30
print(q.is_empty())# Output: True
```

Note:

- Using a list's `pop(0)` for dequeue **is not very efficient** (O(n) time) because it shifts all elements left.
- For better performance, use **`collections.deque`** which provides O(1) time for append and pop from both ends.

Queue implementation in Python using a linked list for efficient enqueue and dequeue operations:

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

class Queue:
    def __init__(self):
        self.front = None  # Points to the front node
        self.rear = None   # Points to the rear node

    def is_empty(self):
        return self.front is None

    def enqueue(self, data):
        new_node = Node(data)
        if self.rear is None:  # Queue is empty
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
```

```
            self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        data = self.front.data
        self.front = self.front.next
        if self.front is None:  # Queue became empty
            self.rear = None
        return data

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.front.data

    def size(self):
        count = 0
        current = self.front
        while current:
            count += 1
            current = current.next
        return count

# Example Usage:
q = Queue()

q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

print(q.peek())    # Output: 10
print(q.dequeue()) # Output: 10
print(q.dequeue()) # Output: 20
print(q.is_empty())# Output: False
print(q.dequeue()) # Output: 30
print(q.is_empty())# Output: True
```

How it works:

- **enqueue:** Adds an element at the rear.
- **dequeue:** Removes an element from the front.
- **peek:** Returns the front element without removing it.
- **is_empty:** Checks if the queue is empty.
- **size:** Counts the number of elements in the queue.

16. Difference between stack and queue.

Feature	Stack	Queue
Order	Last In, First Out (LIFO)	First In, First Out (FIFO)
Insertion	Push at the **top**	Enqueue at the **rear**
Deletion	Pop from the **top**	Dequeue from the **front**
Access	Only top element is accessible	Only front element is accessible
Use Cases	Function call stack, undo operations, expression evaluation	Task scheduling, buffering, BFS traversal
Example Analogy	Stack of plates (last placed is first removed)	Queue in a line (first person in line served first)
Common Operations	push(), pop(), peek()	enqueue(), dequeue(), peek()

Summary:

- **Stack** is about **reversing order** (LIFO).
- **Queue** is about **preserving order** (FIFO).

17. What is a circular queue?

What is a Circular Queue?

A **circular queue** (also called a **ring buffer**) is a type of queue in which the **last position is connected back to the first position**, forming a circle. This means when the queue reaches the end of the allocated space, it wraps around to the beginning if there is space available.

Why use a Circular Queue?

- In a **normal (linear) queue** implemented with an array, after several enqueues and dequeues, even if there’s free space at the front, you can’t use it because the rear pointer has reached the array's end.
- A **circular queue** efficiently uses the allocated space by reusing the freed slots, avoiding the need to shift elements.

Characteristics:

- Fixed-size buffer.
- Two pointers:
  - **front**: points to the front element.
  - **rear**: points to the last element.
- When `rear` reaches the end of the array, it wraps around to 0 if there’s free space.
- Empty condition: `front == -1` or `front == rear + 1` (depending on implementation).
- Full condition: `(rear + 1) % size == front`.

Visualization:

```
Index:    0   1   2   3   4
Queue:   [10, 20, -, -, -]

front -> 0
rear  -> 1

Enqueue 30:
Queue:   [10, 20, 30, -, -]
front -> 0
rear  -> 2

Enqueue 40 and 50:
Queue:   [10, 20, 30, 40, 50]
front -> 0
rear  -> 4

Dequeue 10 and 20:
Queue:   [-, -, 30, 40, 50]
front -> 2
rear  -> 4

Enqueue 60 (wrap-around):
Queue:   [60, -, 30, 40, 50]
```

```
front -> 2
rear  -> 0
```

Use Cases:

- Buffering data streams.
- CPU scheduling.
- Handling real-time streaming data.
- Network data packets buffering.

Here's a Python implementation of a **circular queue** using a fixed-size list (array):

```python
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def is_empty(self):
        return self.front == -1

    def is_full(self):
        return (self.rear + 1) % self.size == self.front

    def enqueue(self, data):
        if self.is_full():
            raise OverflowError("Queue is full")
        if self.is_empty():
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = data

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        data = self.queue[self.front]
        if self.front == self.rear:  # Queue has only one element
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.size
        return data

    def peek(self):
```

```
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.queue[self.front]

    def display(self):
        if self.is_empty():
            print("Queue is empty")
            return
        print("Queue elements:", end=" ")
        i = self.front
        while True:
            print(self.queue[i], end=" ")
            if i == self.rear:
                break
            i = (i + 1) % self.size
        print()

# Example usage:
cq = CircularQueue(5)

cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.enqueue(40)
cq.enqueue(50)

cq.display()  # Output: Queue elements: 10 20 30 40 50

print(cq.dequeue())  # Output: 10
print(cq.dequeue())  # Output: 20

cq.enqueue(60)
cq.enqueue(70)

cq.display()  # Output: Queue elements: 30 40 50 60 70
```

Explanation:

- `front` and `rear` track the start and end positions.
- Enqueue and dequeue operations wrap around using modulo `%`.
- `is_full()` checks if the queue is full to prevent overflow.
- `is_empty()` checks if the queue is empty.

---

17. What is a priority queue?

What is a Priority Queue?

A **priority queue** is a special type of queue where each element is associated with a **priority**. Elements are served based on their priority, **not just their order of arrival**.

- The element with the **highest priority** is dequeued **before** elements with lower priority.
- If two elements have the same priority, they are served according to their order in the queue (depending on implementation).

Key Characteristics:

- **Insertion (enqueue):** Add elements with a priority.
- **Deletion (dequeue):** Remove the element with the highest priority.
- Can be implemented using:
  - **Arrays or linked lists** with sorting.
  - **Heap data structures** (most efficient).

Use Cases:

- CPU scheduling (processes with different priorities).
- Dijkstra's shortest path algorithm.
- Huffman coding tree.
- Managing tasks in real-time systems.

Example:

Element	Priority
Task A	2
Task B	1
Task C	3

- When `dequeuing`, Task C (priority 3) will be served first, then Task A (2), then Task B (1).

---

simple Python implementation of a priority queue using the built-in `heapq` module, which provides an efficient min-heap:

```python
import heapq

class PriorityQueue:
    def __init__(self):
        self.heap = []

    def is_empty(self):
        return len(self.heap) == 0

    def enqueue(self, item, priority):
        # heapq is a min-heap, so use priority as the first element
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty priority queue")
        priority, item = heapq.heappop(self.heap)
        return item

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty priority queue")
        priority, item = self.heap[0]
        return item

# Example Usage:
pq = PriorityQueue()

pq.enqueue("task low priority", 5)
pq.enqueue("task high priority", 1)
pq.enqueue("task medium priority", 3)

print(pq.dequeue())  # Output: task high priority
print(pq.dequeue())  # Output: task medium priority
print(pq.dequeue())  # Output: task low priority
```

Explanation:

- The `heapq` module uses a **min-heap** by default, so the smallest priority number is treated as highest priority.
- You insert items as tuples `(priority, item)`.

- Dequeue removes the item with the smallest priority number.

⚙️ Intermediate Level

🌲 Trees

22. What is a binary tree?

A **binary tree** is a type of data structure in computer science where each node has **at most two children**. These children are usually referred to as the **left child** and the **right child**.

Key characteristics of a binary tree:

- Each node contains a value or data.
- Each node can have zero, one, or two children.
- The top node is called the **root**.
- Nodes with no children are called **leaves**.

Visual example:

```
      A
     / \
    B   C
   /   / \
  D   E   F
```

- Here, `A` is the root.
- `B` and `C` are children of `A`.
- `D` is the left child of `B`.
- `E` and `F` are children of `C`.
- `D`, `E`, and `F` are leaves because they have no children.

Common types of binary trees:

- **Full binary tree**: Every node has either 0 or 2 children.
- **Complete binary tree**: All levels are completely filled except possibly the last, which is filled from left to right.
- **Binary search tree (`BST`)**: A binary tree where the left child's value is less than the parent, and the right child's value is greater.

---

23. Difference between binary tree and binary search tree (`BST`).

Binary Tree

- **Structure:** Each node can have at most two children (left and right).
- **Ordering:** No specific order or property about the values of nodes.
- **Purpose:** General-purpose tree structure for hierarchical data.
- **Example:**

```
      10
     /  \
    5    20
   / \
  7   3
```

  Here, the values don't necessarily follow any order.

Binary Search Tree (BST)

- **Structure:** Also a binary tree where each node has at most two children.
- **Ordering property:**
  - The left child node's value is **less than** its parent node's value.
  - The right child node's value is **greater than** its parent node's value.
- **Purpose:** Enables efficient searching, insertion, and deletion (usually O(log n) time complexity for balanced BSTs).
- **Example:**

```
      10
     /  \
    5    20
   / \
  3   7
```

  Here, the values follow the BST property:
  - Left subtree of 10 contains values less than 10.
  - Right subtree contains values greater than 10.
  - This property holds for every node in the tree.

Summary Table:

Feature	Binary Tree	Binary Search Tree (BST)
Max children per node	2	2
Node ordering	No specific order	Left < Parent < Right
Purpose	General hierarchical structure	Efficient searching & sorting
Search performance	O(n) (linear search)	O(log n) average, O(n) worst-case

---

implementation

simple Python implementation of a **Binary Search Tree (BST)** with basic operations: insertion, search, and an in-order traversal to print the values in sorted order.

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

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert(node.left, key)
        elif key > node.key:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert(node.right, key)
        # If key == node.key, do nothing or handle duplicates here
```

```
    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node is None:
            return False
        if key == node.key:
            return True
        elif key < node.key:
            return self._search(node.left, key)
        else:
            return self._search(node.right, key)

    def inorder_traversal(self):
        elements = []
        self._inorder(self.root, elements)
        return elements

    def _inorder(self, node, elements):
        if node:
            self._inorder(node.left, elements)
            elements.append(node.key)
            self._inorder(node.right, elements)

# Example usage:
bst = BST()
bst.insert(10)
bst.insert(5)
bst.insert(20)
bst.insert(3)
bst.insert(7)

print("In-order traversal:", bst.inorder_traversal())  # Output: [3, 5, 7,
10, 20]
print("Search 7:", bst.search(7))  # Output: True
print("Search 15:", bst.search(15))  # Output: False
```

Explanation:

- **`Node` class:** Represents each node in the tree.
- **`BST` class:** Manages the BST operations.
- **`insert`:** Adds a new key in the right position to keep the BST property.
- **`search`:** Checks whether a key exists in the BST.
- **`inorder_traversal`:** Returns all keys sorted (in-order traversal).

---

23. What is tree traversal? Explain `inorder, preorder, and postorder traversal`.

What is Tree Traversal?

Tree traversal means visiting all the nodes in a tree **in a specific order**. Since trees are hierarchical, traversal defines the order in which nodes are processed.

There are several traversal methods, but the most common for binary trees are:

- **Inorder**
- **Preorder**
- **Postorder**

---

1. Inorder Traversal (Left, Root, Right)

- Visit the **left subtree** first.
- Visit the **root node**.
- Visit the **right subtree**.

Use case: In a binary search tree (BST), inorder traversal visits nodes in **ascending sorted order**.

Example:

For the tree:

```
      10
     /  \
    5    20
   / \
  3   7
```

Inorder output: `3, 5, 7, 10, 20`

---

2. Preorder Traversal (Root, Left, Right)

- Visit the **root node** first.
- Visit the **left subtree**.
- Visit the **right subtree**.

Use case: Preorder traversal is useful for **copying** the tree or getting a prefix expression of an expression tree.

Example:
For the same tree,

Preorder output: `10, 5, 3, 7, 20`

---

3. Postorder Traversal (Left, Right, Root)

- Visit the **left subtree**.
- Visit the **right subtree**.
- Visit the **root node** last.

Use case: Postorder is useful for **deleting** the tree or getting a postfix expression of an expression tree.

Example:
For the same tree,

Postorder output: `3, 7, 5, 20, 10`

Summary Table:

Traversal	Order	Example Output (on above tree)
Inorder	Left → Root → Right	3, 5, 7, 10, 20
Preorder	Root → Left → Right	10, 5, 3, 7, 20
Postorder	Left → Right → Root	3, 7, 5, 20, 10

---

how you can implement **inorder, **preorder**, and **postorder** traversals in Python for a binary tree:**

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

def inorder(node):
```

```
    if node:
        inorder(node.left)          # Left
        print(node.key, end=' ')    # Root
        inorder(node.right)         # Right

def preorder(node):
    if node:
        print(node.key, end=' ')    # Root
        preorder(node.left)         # Left
        preorder(node.right)        # Right

def postorder(node):
    if node:
        postorder(node.left)        # Left
        postorder(node.right)       # Right
        print(node.key, end=' ')    # Root

# Example tree construction:
root = Node(10)
root.left = Node(5)
root.right = Node(20)
root.left.left = Node(3)
root.left.right = Node(7)

print("Inorder traversal:")
inorder(root)  # Output: 3 5 7 10 20
print("\nPreorder traversal:")
preorder(root)  # Output: 10 5 3 7 20
print("\nPostorder traversal:")
postorder(root)  # Output: 3 7 5 20 10
```

---

24. What is a balanced binary tree?

A **balanced binary tree** is a type of binary tree where the tree is structured to minimize its height, ensuring that the depths of the left and right `subtrees` of **every node** differ by no more than a certain amount (usually 1).

Why does balancing matter?

- A balanced tree keeps operations like **search, insertion, and deletion efficient**—usually **O(log n)** time.

- An unbalanced tree can degenerate into a linked list, making these operations O(n).

Common Definition:

For **each node** in a balanced binary tree:

$$
| \text{height(left subtree)} - \text{height(right subtree)} | \leq 1
$$

If this condition holds for all nodes, the tree is balanced.

Examples:

- **Balanced tree:**

```
      10
     /  \
    5    20
   / \
  3   7
```

- **Unbalanced tree:**

```
    10
      \
       20
         \
          30
```

In the unbalanced tree, the right subtree is much deeper than the left, making operations less efficient.

Types of Balanced Binary Trees:

- **AVL Trees:** Strictly balanced by maintaining the height difference ≤ 1 for every node.
- **Red-Black Trees:** Looser balancing but guarantees O(log n) operations.
- **B-Trees and others:** Balanced trees optimized for databases and file systems.

---

24. How do you check if a tree is a `BST`?

Checking if a binary tree is a **valid Binary Search Tree (`BST`)** means verifying whether it satisfies the `BST` property:

> For every node, all values in its left `subtree` are **less than** the node's value, and all values in its right `subtree` are **greater than** the node's value.

How to check if a tree is a `BST`?

A common and efficient way is to use **`inorder` traversal** and verify if the resulting sequence is sorted strictly increasing, **or** to recursively check the nodes with valid value ranges.

Method 1: Using `Inorder` Traversal

- Perform `inorder` traversal.
- Check if the values are in strictly increasing order.

```
def is_bst_inorder(root):
    inorder_vals = []

    def inorder(node):
        if node:
            inorder(node.left)
            inorder_vals.append(node.key)
            inorder(node.right)

    inorder(root)
    # Check if sorted strictly increasing
    for i in range(1, len(inorder_vals)):
        if inorder_vals[i] <= inorder_vals[i-1]:
            return False
    return True
```

Method 2: Using Value Range Limits (Preferred)

- Recursively check each node.

- Pass down allowable min and max values.
- For each node, its value must be strictly between min and max.

```
def is_bst_util(node, min_val, max_val):
    if node is None:
        return True
    if node.key <= min_val or node.key >= max_val:
        return False
    return (is_bst_util(node.left, min_val, node.key) and
            is_bst_util(node.right, node.key, max_val))

def is_bst(root):
    return is_bst_util(root, float('-inf'), float('inf'))
```

Explanation:

- For the root, allowed range is (-∞, +∞).
- For left child, max allowed value is the parent node's key.
- For right child, min allowed value is the parent node's key.

---

24. What is the height or depth of a tree?

The terms **height** and **depth** are related but distinct concepts in tree data structures.

Height of a Tree

- The **height** of a tree is the length of the **longest path** from the **root node** down to the **farthest leaf node**.
- It's usually measured in **edges** or **nodes**, but commonly in edges.
- If the tree has only one node (root), the height is 0 (no edges).

Example:

```
      10        <-- Level 0 (root)
     /  \
    5    20     <-- Level 1
```

```
    / \
   3   7        <-- Level 2
```

- Height of this tree = 2 (from root `10` down to leaf nodes `3` or `7` through two edges).

Depth of a Node

- The **depth** of a node is the number of edges from the **root node** to that node.
- The root node's depth is 0.
- For example, in the above tree:
  - Depth of node `10` = 0
  - Depth of node `5` = 1
  - Depth of node `3` = 2

Summary

Term	Definition	Example in above tree
Height	Longest path from root to a leaf (max depth)	2
Depth	Distance from root to a specific node	Varies (e.g., node 3 has depth 2)

---

25. What is an `AVL` tree?

An **`AVL` tree** is a type of **self-balancing binary search tree**. It was the first such data structure invented and is named after its inventors **`Adelson-Velsky and Landis`**.

Key characteristics of an `AVL` Tree:

- It maintains a strict **balance condition**:
  For every node in the tree, the **height difference** (called the **balance factor**) between its left and right `subtrees` is at most **1**. [
  \text{balance factor} = \text{height(`left subtree`)} - \text{height(`right subtree`)} \in {-1, 0, 1}
  ]

- If inserting or deleting a node causes the balance factor of any node to go outside this range (less than -1 or greater than 1), the tree **performs rotations** to restore balance.

Why use an `AVL` tree?

- Ensures that the tree height remains **O(log n)**, making operations like search, insert, and delete efficient.
- Prevents degeneration into a skewed tree (which behaves like a linked list).

How does balancing work?

When the balance factor becomes too high or too low after an insertion or deletion, the `AVL` tree applies one of four types of **rotations**:

1. **Right Rotation (LL case)**
2. **Left Rotation (RR case)**
3. **Left-Right Rotation (LR case)**
4. **Right-Left Rotation (`RL` case)**

These rotations restructure the tree to restore the balance factor condition.

Summary:

Feature	Description
Type	Self-balancing binary search tree
Balance condition	Balance factor ∈ {-1, 0, 1} for all nodes
Balancing method	Rotations (single and double)
Time complexity	O(log n) for search, insert, delete

---

25. What is a heap?

A **heap** is a special type of **binary tree** that satisfies the **heap property**, and it's most commonly implemented as a **binary heap** using an array.

✅ Key Features of a Heap:

- **Complete binary tree**:
  - All levels are fully filled **except possibly the last**, which is filled **from left to right**.
- **Heap property** (determines the type of heap):
  - **Max-Heap**: Parent nodes are **greater than or equal to** their children.
  - **Min-Heap**: Parent nodes are **less than or equal to** their children.

🔁 Types of Heaps:

Type	Heap Property	Root Contains
Min-Heap	Every parent ≤ its children	**Minimum** value
Max-Heap	Every parent ≥ its children	**Maximum** value

📦 Example: Min-Heap

```
      2
     / \
    4   5
   / \
  10  15
```

Here:

- 2 ≤ 4 and 5
- 4 ≤ 10 and 15

So, this is a valid **min-heap**.

🔢 Heap as an Array

Heaps are usually implemented using arrays (not pointers like trees), where:

- For any element at index `i`:
  - **Left child** = at index `2i + 1`
  - **Right child** = at index `2i + 2`

- **Parent** = at index `(i - 1) // 2`

This array-based structure makes heaps efficient for priority queues.

🕑 Time Complexities (Binary Heap):

Operation	Time Complexity
Insertion	O(log n)
Deletion (min/max)	O(log n)
Access min/max	O(1)

🧠 Common Use Cases:

- **Priority queues**
- **Heap sort**
- **Scheduling systems**
- **Graph algorithms** (e.g., Dijkstra's shortest path)

---

25. Difference between min-heap and max-heap.

✅ Key Difference Between Min-Heap and Max-Heap:

Feature	Min-Heap	Max-Heap
Heap Property	Parent ≤ children	Parent ≥ children
Root Node	Contains the **smallest** element	Contains the **largest** element
Use Case	When you need quick access to the **min**	When you need quick access to the **max**
Structure Type	Complete binary tree	Complete binary tree
Insert/Delete	O(log n)	O(log n)
Access Root	O(1) (fast access to smallest)	O(1) (fast access to largest)

🧠 Visual Example:

Min-Heap:

```
      1
     / \
    3   5
   / \
  7   9
```

- Every parent node is **less than or equal** to its children.
- 1 is the smallest element, at the root.

Max-Heap:

```
      9
     / \
    5   6
   / \
  3   2
```

- Every parent node is **greater than or equal** to its children.
- 9 is the largest element, at the root.

🧰 Real-World Use Cases:

Scenario	Use Heap Type
Dijkstra’s algorithm (shortest path)	**Min-Heap**
Job scheduling with highest priority	**Max-Heap**
Maintaining median (via 2 heaps)	Both
Heap sort (descending order)	**Max-Heap**
Heap sort (ascending order)	**Min-Heap**

---

implementation

Python implementation of both a **Min-Heap** and a **Max-Heap** using lists and manual operations.

✅ Min-Heap and Max-Heap (Custom Python Implementation)

We'll use a common class and change the comparison logic for min vs. max behavior.

🔧 Base Class

```
class BinaryHeap:
    def __init__(self, is_min_heap=True):
        self.heap = []
        self.is_min_heap = is_min_heap

    def _compare(self, a, b):
        # For Min-Heap: a < b → True
        # For Max-Heap: a > b → True
        return a < b if self.is_min_heap else a > b

    def insert(self, value):
        self.heap.append(value)
        self._heapify_up(len(self.heap) - 1)

    def extract(self):
        if not self.heap:
            return None
        root = self.heap[0]
        last = self.heap.pop()
        if self.heap:
            self.heap[0] = last
            self._heapify_down(0)
        return root

    def peek(self):
        return self.heap[0] if self.heap else None

    def _heapify_up(self, index):
        parent = (index - 1) // 2
        if index > 0 and self._compare(self.heap[index],
self.heap[parent]):
            self.heap[index], self.heap[parent] = self.heap[parent],
self.heap[index]
            self._heapify_up(parent)

    def _heapify_down(self, index):
        smallest_or_largest = index
        left = 2 * index + 1
```

```
        right = 2 * index + 2
        size = len(self.heap)

        if left < size and self._compare(self.heap[left],
self.heap[smallest_or_largest]):
            smallest_or_largest = left
        if right < size and self._compare(self.heap[right],
self.heap[smallest_or_largest]):
            smallest_or_largest = right

        if smallest_or_largest != index:
            self.heap[index], self.heap[smallest_or_largest] =
self.heap[smallest_or_largest], self.heap[index]
            self._heapify_down(smallest_or_largest)

    def display(self):
        print(self.heap)
```

✅ Using the Class for Min-Heap

```
print("Min-Heap Example:")
min_heap = BinaryHeap(is_min_heap=True)
min_heap.insert(10)
min_heap.insert(4)
min_heap.insert(15)
min_heap.insert(1)
min_heap.display()       # Output: [1, 4, 15, 10]
print(min_heap.extract())  # Output: 1
min_heap.display()       # Output: [4, 10, 15]
```

✅ Using the Class for Max-Heap

```
print("\nMax-Heap Example:")
max_heap = BinaryHeap(is_min_heap=False)
max_heap.insert(10)
max_heap.insert(4)
max_heap.insert(15)
max_heap.insert(1)
max_heap.display()       # Output: [15, 10, 4, 1]
print(max_heap.extract())  # Output: 15
max_heap.display()       # Output: [10, 1, 4]
```

🧠 Summary

- This class allows you to switch between Min-Heap and Max-Heap by passing `is_min_heap=True` or `False`.
- Basic operations: `insert`, `extract` (remove min/max), and `peek`.

🔁 Graphs

31. What is a graph?

A **graph** is a mathematical structure used to model relationships between objects. It consists of:

- **Vertices** (also called **nodes**): These are the objects or entities.
- **Edges** (also called **links** or **arcs**): These represent the relationships or connections between the vertices.

Types of Graphs:

1. **Undirected Graph**:
   - Edges have no direction.
   - If there is an edge between node A and node B, it means A is connected to B *and* B is connected to A.
2. **Directed Graph (Digraph)**:
   - Edges have a direction.
   - An edge from A to B (written A → B) means the connection goes one way, from A to B.
3. **Weighted Graph**:
   - Edges have weights (or costs), often used to represent things like distance, time, or cost.
4. **Unweighted Graph**:
   - All edges are considered equal; no weights are assigned.

Real-world Examples:

- **Social Networks**: People are nodes; friendships or connections are edges.
- **Maps**: Locations are nodes; roads are edges (can be weighted by distance or time).
- **Internet**: `Webpages` are nodes; hyperlinks are edges.

Visualization:

A graph can be drawn as a diagram with dots (vertices) connected by lines or arrows (edges), making it easier to understand complex relationships.

31. Difference between directed and `undirected` graphs.

The key difference between **directed** and **undirected** graphs lies in whether the **edges** (connections between nodes) have direction or not.

🔁 Undirected Graph

- **Definition**: Edges have **no direction**.
- **Connection**: If node A is connected to node B, then B is also connected to A.
- **Edge Representation**: As unordered pairs: (A, B)
- **Use Case Examples**:
  - Friendships in social networks (if A is friends with B, B is friends with A)
  - Road maps where roads go both ways

Example:

```
A ----- B
```

This means A is connected to B and B is connected to A.

➡️ Directed Graph (Digraph)

- **Definition**: Edges have a **direction**.
- **Connection**: If node A is connected to node B, it doesn't necessarily mean B is connected to A.
- **Edge Representation**: As ordered pairs: (A → B)
- **Use Case Examples**:
  - Following on Twitter (A can follow B, but B may not follow A)
  - One-way streets
  - Data flow or task scheduling

Example:

```
A → B
```

This means A is connected to B, but B is **not** necessarily connected back to A.

Summary Table:

Feature	Undirected Graph	Directed Graph
Edge Direction	No	Yes
Edge Representation	(A, B)	(A → B)
Mutual Connection	A connected to B ⇔ B to A	A → B ≠ B → A
Typical Use Cases	Mutual relationships	One-way relationships

---

31. What are adjacency matrix and adjacency list representations?

Adjacency matrix and **adjacency list** are two common ways to represent **graphs** in computer programs.

📊 1. Adjacency Matrix

✅ Definition:

An **adjacency matrix** is a `2D` **array (or matrix)** used to represent which vertices (nodes) are connected to which other vertices.

- The rows and columns represent nodes.
- The entry at position **(i, j)**:
  - Is **1** (or the edge **weight**) if there's an edge from node `i` to node `j`.
  - Is **0** if there is **no edge**.

Example (Undirected, Unweighted Graph):

Graph:

```
A --- B
|     |
C ----
```

Nodes: A, B, C → [0, 1, 2]

Adjacency Matrix:

```
    A  B  C
A [ 0, 1, 1 ]
B [ 1, 0, 1 ]
C [ 1, 1, 0 ]
```

> Note: For **undirected** graphs, the matrix is **symmetric**.

✅ Pros:

- Very fast to check if two nodes are connected → O(1) time.
- Easy to implement.

❌ Cons:

- Uses **O(V²)** space even if the graph is sparse (few edges).
- Not efficient for graphs with many nodes but few connections.

📋 2. Adjacency List

✅ Definition:

An **adjacency list** stores each node and a **list of its neighbors** (connected nodes).

Example (Same Graph as Above):

```
Graph:
A --- B
|     |
C ----
```

Adjacency List:

```
A: [B, C]
B: [A, C]
C: [A, B]
```

Or as a dictionary in code:

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C'],
```

```
    'C': ['A', 'B']
}
```

✅ Pros:

- **Space-efficient**: Uses **O(V + E)** space (better for sparse graphs).
- Easier to iterate over neighbors of a node.

❌ Cons:

- Slower to check if an edge exists → O(degree of node).
- Less suitable for dense graphs where edge lookups are frequent.

🔍 Summary Table:

Feature	Adjacency Matrix	Adjacency List
Space Complexity	$O(V^2)$	O(V + E)
Check Edge Existence	O(1)	O(k), where k = degree
Iterating Neighbors	O(V)	O(k)
Best For	Dense graphs	Sparse graphs

---

implementation

Basic implementation of **adjacency matrix** and **adjacency list** in **Python** for an **undirected and unweighted graph**.

✅ Graph:

Let's take this simple graph as an example:

```
A --- B
|     |
C     D
```

Nodes: A, B, C, D
Edges: A-B, A-C, B-D

🔢 1. Adjacency Matrix Implementation

```
# Map node names to indices
nodes = ['A', 'B', 'C', 'D']
index_map = {node: i for i, node in enumerate(nodes)}

# Initialize 4x4 matrix with 0s
n = len(nodes)
adj_matrix = [[0 for _ in range(n)] for _ in range(n)]

# Add edges: (A-B), (A-C), (B-D)
edges = [('A', 'B'), ('A', 'C'), ('B', 'D')]

for u, v in edges:
    i, j = index_map[u], index_map[v]
    adj_matrix[i][j] = 1
    adj_matrix[j][i] = 1  # because it's undirected

# Print adjacency matrix
print("Adjacency Matrix:")
for row in adj_matrix:
    print(row)
```

🔍 Output:

```
Adjacency Matrix:
[0, 1, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 0]
[0, 1, 0, 0]
```

📋 2. Adjacency List Implementation

```
# Initialize empty adjacency list
adj_list = {node: [] for node in nodes}

# Add edges
for u, v in edges:
    adj_list[u].append(v)
    adj_list[v].append(u)  # because it's undirected

# Print adjacency list
print("\nAdjacency List:")
```

```
for node in adj_list:
    print(f"{node}: {adj_list[node]}")
```

🔍 Output:

```
Adjacency List:
A: ['B', 'C']
B: ['A', 'D']
C: ['A']
D: ['B']
```

---

implement both the Adjacency Matrix and Adjacency List for a Directed Graph in Python.

✅ Example Directed Graph:

```
A → B
↓
C     D
     ↑
     B
```

Nodes: A, B, C, D

Edges:

- A → B
- A → C
- B → D

🔢 1. Adjacency Matrix for Directed Graph

```
# Map node names to indices
nodes = ['A', 'B', 'C', 'D']
index_map = {node: i for i, node in enumerate(nodes)}

# Initialize 4x4 matrix with 0s
n = len(nodes)
adj_matrix = [[0 for _ in range(n)] for _ in range(n)]
```

```python
# Directed edges
edges = [('A', 'B'), ('A', 'C'), ('B', 'D')]

# Fill the matrix
for u, v in edges:
    i, j = index_map[u], index_map[v]
    adj_matrix[i][j] = 1  # Only one direction (u → v)

# Print adjacency matrix
print("Directed Graph - Adjacency Matrix:")
for row in adj_matrix:
    print(row)
```

🔍 Output:

```
Directed Graph - Adjacency Matrix:
[0, 1, 1, 0]
[0, 0, 0, 1]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

📋 2. Adjacency List for Directed Graph

```python
# Initialize adjacency list
adj_list = {node: [] for node in nodes}

# Fill the list
for u, v in edges:
    adj_list[u].append(v)  # Only add v to u's list

# Print adjacency list
print("\nDirected Graph - Adjacency List:")
for node in adj_list:
    print(f"{node}: {adj_list[node]}")
```

🔍 Output:

```
Directed Graph - Adjacency List:
A: ['B', 'C']
B: ['D']
C: []
D: []
```

✅ Summary of Key Differences for Directed Graphs:

Feature	Adjacency Matrix	Adjacency List
A → B only	`matrix[A][B] = 1`	`B` is added to `A`'s list
B → A ≠ A → B	Not implied	Not implied
No need to mirror edges	No `matrix[B][A] = 1`	No `A` in `B`'s list

---

32. What is DFS (Depth First Search)?

DFS (Depth-First Search) is a fundamental graph traversal algorithm used to explore nodes and edges of a graph in depthward motion — meaning it explores **as far as possible** along a branch before **backtracking**.

🔍 How DFS Works:

1. Start at a **source node**.
2. Visit the node and **mark it as visited**.
3. Recursively (or using a stack) visit **unvisited neighbors**.
4. Backtrack when you reach a node with **no unvisited neighbors**.

DFS can be implemented using:

- **Recursion** (easy and elegant)
- **Stack** (explicit, for iterative approach)

🧠 Key Concepts:

Feature	DFS
Data Structure	Stack (explicit or via recursion)
Strategy	Go deep before going wide
Use Case Examples	Pathfinding, cycle detection, topological sort, maze solving
Works On	Both directed and undirected graphs

✅ Example Graph (Undirected):

```
    A
   / \
  B   C
     / \
    D   E
```

Adjacency List:

```
graph = {
    'A': ['B', 'C'],
    'B': ['A'],
    'C': ['A', 'D', 'E'],
    'D': ['C'],
    'E': ['C']
}
```

👩‍💻 DFS Using Recursion (Python)

```
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()

    print(node, end=' ')
    visited.add(node)

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Run DFS starting from node 'A'
dfs(graph, 'A')
```

🔍 Output (one possible traversal):

```
A B C D E
```

👩‍💻 DFS Using Stack (Iterative Version)

```python
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            # Add neighbors in reverse to visit leftmost first
            stack.extend(reversed(graph[node]))

# Run DFS
dfs_iterative(graph, 'A')
```

📌 DFS Use Cases:

- Detecting **cycles** in graphs
- Finding **connected components**
- **Pathfinding** (e.g., maze-solving)
- **Topological sorting** (for DAGs)
- **Web crawlers**
- AI decision trees

---

32. What is `BFS` (Breadth First Search)?

`BFS` **(Breadth-First Search)** is a fundamental graph traversal algorithm that explores a graph **level by level**, meaning it visits all the **neighbors** of a node **before** going deeper into the graph.

🔍 How `BFS` Works:

1. Start at a **source node**.
2. Visit the node and **enqueue** it.
3. Dequeue a node, then visit and enqueue all of its **unvisited neighbors**.
4. Repeat until the queue is empty.

It uses a **queue** to keep track of the next nodes to visit.

🧠 Key Concepts:

Feature	BFS
Data Structure	**Queue**
Strategy	**Level-order traversal**
Works On	**Directed** and **Undirected** graphs
Finds	**Shortest path** in **unweighted** graphs
Type	**Iterative** (not recursive)

✅ Example Graph (Undirected):

```
    A
   / \
  B   C
     / \
    D   E
```

Adjacency List:

```
graph = {
    'A': ['B', 'C'],
    'B': ['A'],
    'C': ['A', 'D', 'E'],
    'D': ['C'],
    'E': ['C']
}
```

👨‍💻 BFS in Python:

```
from collections import deque

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

```
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

# Run BFS starting from node 'A'
bfs(graph, 'A')
```

🔍 Output (one possible BFS traversal):

```
A B C D E
```

📌 BFS Use Cases:

- 🔍 **Shortest path** in unweighted graphs
- 🌐 **Crawling** websites or networks
- 🧩 **Solving puzzles** (like Rubik's cube, mazes)
- 🧠 **AI pathfinding** (e.g., in games)
- 🔄 Finding **connected components** in graphs
- 🌳 **Level-order traversal** of trees

🆚 BFS vs DFS Summary:

Feature	BFS	DFS
Data Structure	Queue	Stack (or recursion)
Strategy	Explore **wide first**	Explore **deep first**
Finds Shortest Path	✅ (in unweighted graphs)	❌ (not guaranteed)
Implementation	Iterative	Recursive or iterative
Memory Usage	Higher for wide graphs	Higher for deep graphs

---

33. How do you detect a cycle in a graph?

Detecting cycles in a graph depends on whether the graph is **directed** or **undirected** — the approach differs slightly.

1️⃣ Cycle Detection in Undirected Graph

Idea:

- Use **DFS**.
- If you visit a vertex that's **already visited** and is **not the parent** of the current vertex, a cycle exists.

Why check parent?

- Because in undirected graphs, edges go both ways, so the immediate parent will always be visited.

Python Implementation (using DFS):

```
def has_cycle_undirected(graph):
    visited = set()

    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                # Visited and not parent → cycle found
                return True
        return False

    for node in graph:
        if node not in visited:
            if dfs(node, None):
                return True
    return False

# Example usage
graph_undirected = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
}
```

```
print(has_cycle_undirected(graph_undirected))  # Output: True (cycle
exists)
```

2️⃣ Cycle Detection in Directed Graph

Idea:

- Use **DFS** with a recursion stack.
- Keep track of nodes currently in the recursion call stack.
- If you revisit a node that's in the recursion stack → cycle detected.

Explanation:

- If a back edge (to an ancestor) is found during DFS, the graph has a cycle.

Python Implementation:

```
def has_cycle_directed(graph):
    visited = set()
    rec_stack = set()

    def dfs(node):
        visited.add(node)
        rec_stack.add(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
            elif neighbor in rec_stack:
                # Back edge found → cycle exists
                return True

        rec_stack.remove(node)
        return False

    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False

# Example usage
graph_directed = {
    'A': ['B'],
    'B': ['C'],
```

```
    'C': ['A']  # Cycle A->B->C->A
}

print(has_cycle_directed(graph_directed))  # Output: True (cycle exists)
```

Summary:

Graph Type	Approach	Key Idea
Undirected	DFS + Parent check	Visited node that's not parent
Directed	DFS + Recursion stack	Node revisited in current stack

---

34. What is Dijkstra's algorithm?

Dijkstra's algorithm is a classic algorithm used to find the **shortest path** from a single source node to all other nodes in a **weighted graph** with **non-negative edge weights**.

🔍 What Does It Do?

- Finds the minimum total **distance (or cost)** to reach every node from the starting node.
- Useful for things like GPS navigation, network routing, and many optimization problems.

🧠 How Does It Work?

1. Initialize distances to all nodes as **infinity**, except the source node which is 0.
2. Maintain a **priority queue (min-heap)** to pick the next node with the smallest tentative distance.
3. For the current node, **update distances** to its neighbors if a shorter path is found.
4. Mark the current node as **visited** and repeat until all nodes are processed or the queue is empty.

🔑 Key Points:

- Works only if **edge weights are non-negative**.
- Uses a **greedy** strategy by always expanding the closest unvisited node.
- Can be implemented with a priority queue (efficiently) or a simple list (less efficient).

📋 Example Graph:

```
   (1)
A-------B
|\      |
| \     |(4)
(3) \   |
|    \  |
C-------D
   (2)
```

Edges & weights:

- A → B: 1
- A → C: 3
- B → D: 4
- C → D: 2

👨‍💻 Python Implementation

```python
import heapq

def dijkstra(graph, start):
    # Distances dictionary
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Min-heap priority queue (distance, node)
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # If this distance is not up to date, skip
        if current_distance > distances[current_node]:
            continue

        # Check neighbors
```

```
        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight

            # If shorter path found
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Graph represented as adjacency list with weights
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 4)],
    'C': [('D', 2)],
    'D': []
}

# Find shortest distances from A
print(dijkstra(graph, 'A'))
```

🔍 Output:

```
{'A': 0, 'B': 1, 'C': 3, 'D': 5}
```

⏩ Summary:

Feature	Description
Input	Weighted graph, source node
Output	Shortest path distances
Edge weights	Must be **non-negative**
Time complexity	O((V + E) log V) with priority queue
Uses	GPS, network routing, game AI

---

34. What is a topological sort?

Topological Sort is an algorithm that **orders the vertices of a Directed Acyclic Graph (DAG)** in a linear sequence such that for every directed edge `u → v`, vertex `u` comes

before vertex `v` in the ordering.

🔍 What is it used for?

- Scheduling tasks with dependencies (e.g., build systems, course prerequisites)
- Resolving symbol dependencies in linkers
- Ordering jobs in workflows

⚠️ Important:

- Works **only on Directed Acyclic Graphs (DAGs)** — graphs with **no cycles**.
- If there's a cycle, a topological sort is **not possible**.

🧠 How it works (high-level):

1. Find nodes with **no incoming edges** (no dependencies).
2. Add them to the sorted list and **remove them** from the graph.
3. Repeat with the remaining graph until all nodes are processed.

Common Approaches:

1. Kahn's Algorithm (BFS-based)

- Maintain a queue of nodes with zero incoming edges.
- Repeatedly remove a node from the queue, add it to the result, and reduce the in-degree of its neighbors.
- If at the end some nodes remain, the graph has a cycle.

2. DFS-based Topological Sort

- Perform DFS.
- When a node finishes processing (all descendants visited), **push it to a stack**.
- Reverse the stack for the topological order.

👨‍💻 Example Using Kahn's Algorithm:

```python
from collections import deque

def topological_sort(graph):
    in_degree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    # Queue of all nodes with zero in-degree
    queue = deque([u for u in graph if in_degree[u] == 0])
    topo_order = []

    while queue:
        u = queue.popleft()
        topo_order.append(u)

        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    if len(topo_order) == len(graph):
        return topo_order
    else:
        # Graph has at least one cycle
        return None

# Example graph
graph = {
    'A': ['C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['F'],
    'E': ['H', 'F'],
    'F': ['G'],
    'G': [],
    'H': []
}

print(topological_sort(graph))
```

🔍 Possible Output:

```
['A', 'B', 'D', 'C', 'E', 'H', 'F', 'G']
```

Summary:

Aspect	Description
Input	Directed Acyclic Graph (DAG)
Output	Linear ordering of nodes
Use case	Task scheduling, dependency resolution
Detects cycles?	Yes (if cycle exists, returns None)
Algorithms	Kahn's Algorithm, DFS-based

🧮 Hashing

39. What is hashing?

Hashing is a technique used to **convert data of arbitrary size into a fixed-size value** — called a **hash value** or **hash code** — usually for fast data retrieval, comparison, or indexing.

🔍 What is hashing?

- It uses a **hash function** to map input data (like strings, numbers, objects) to a number (the hash).
- This hash is typically used as an **index** into a data structure called a **hash table** (or hash map).
- Hashing helps achieve **fast lookup, insertion, and deletion** — often close to **O(1)** time on average.

🧠 How does it work?

1. You provide input data (called the **key**).
2. The **hash function** processes the key and produces an integer (the **hash code**).
3. This integer is used to place or find the key/value in the **hash table**.

Example:

Suppose you want to store student records by their ID.

- Key: Student ID (e.g., 12345)
- Hash function might compute: `hash = 12345 % 10 = 5`
- The record is stored in bucket 5.

Important Concepts:

Term	Meaning
Hash Function	Function converting input to hash code
Hash Table	Data structure storing key-value pairs using hashes
Collision	When two keys produce the same hash
Collision Resolution	Techniques like chaining or open addressing
Load Factor	Ratio of number of elements to hash table size

Why use hashing?

- **Fast access** to data without searching the whole dataset.
- Widely used in databases, caching, password storage (with cryptographic hashing), data indexing.

Simple Python Example:

```python
# Simple hash function (not good for real use!)
def simple_hash(key, size):
    return sum(ord(c) for c in key) % size

size = 10
key = "apple"
print(f"Hash for '{key}':", simple_hash(key, size))
```

Collision Handling (brief):

- **Chaining:** Store collided elements in a list at the bucket.
- **Open Addressing:** Find another empty bucket (linear probing, quadratic probing).

If you want, I can explain:

- How real-world hash functions work (like MD5, SHA)
- Implement a hash table with collision resolution
- Cryptographic hashing vs data hashing
- Applications of hashing in security or data structures

---

40. What are hash functions?

Hash functions are special functions that take input data (of any size) and produce a **fixed-size string or number** — called a **hash value** or **hash code** — that uniquely (ideally) represents the input.

🔍 What are hash functions used for?

- To quickly **map data** to a hash code (index) for efficient storage and retrieval (like in hash tables).
- To **verify data integrity** (e.g., checksums).
- In **cryptography** for secure hashing of passwords, digital signatures, etc.
- To **fingerprint data** uniquely.

🧠 Properties of a Good Hash Function:

Property	Description
Deterministic	Same input always produces the same hash output
Fast to compute	Should generate hash quickly
Uniform distribution	Should spread outputs evenly to avoid clustering (reduce collisions)
Minimize collisions	Different inputs should rarely produce the same hash (collision)
Pre-image resistance (cryptographic)	Hard to find input from hash output (important in security)

Example:

If you use a simple hash function like:

```
def simple_hash(key, size):
    return sum(ord(c) for c in key) % size
```

- Input: `"apple"`
- Hash value: sum of ASCII codes of letters modulo table size

---

Types of Hash Functions:

Type	Use Case
Non-cryptographic	Hash tables, checksums, data indexing
Cryptographic	Security: SHA-256, MD5, SHA-1

Why hash functions matter:

- They enable **fast data lookup**.
- Protect data with **secure hashing** (passwords, digital signatures).
- Verify files or messages haven't been tampered with.

Want me to:

- Show examples of cryptographic hash functions?
- Explain collision handling in hashing?
- Discuss how hashing is used in password storage?
- Build a simple hash table using a hash function?

---

41. What are collisions, and how do you handle them?

🔍 What is a collision?

A **collision** happens when **two different keys** produce the **same hash value** (or hash code).

Since a hash function maps a large set of possible inputs to a limited range of outputs (like an array index), collisions are **inevitable**.

Why do collisions matter?

- If two keys map to the same position in a hash table, they compete for the same "slot".
- Without handling collisions, data can be **overwritten or lost**.
- Proper collision handling ensures the **integrity and efficiency** of hash-based data structures.

👩‍🔧 How to handle collisions?

There are two main strategies:

1. Chaining

- Each slot in the hash table stores a **linked list (or other data structure)** of entries.
- When multiple keys hash to the same index, they are stored in the list at that index.
- Lookup involves scanning the list to find the right key.

Example:

```
hash_table = [[] for _ in range(size)]

def insert(key, value):
    index = hash_function(key) % size
    hash_table[index].append((key, value))
```

---

2. Open Addressing

- If a collision occurs, find another empty slot **in the table** using a probing sequence.
- Common probing methods:
  - **Linear Probing:** Check the next slot sequentially (index+1, index+2, ...)
  - **Quadratic Probing:** Check slots using quadratic function (index + 1^2, index + 2^2, ...)
  - **Double Hashing:** Use a second hash function to compute the step size.

Example (Linear Probing):

```
def insert(key, value):
    index = hash_function(key) % size
    while hash_table[index] is not None:
        index = (index + 1) % size  # Move to next slot
    hash_table[index] = (key, value)
```

Summary Table:

Collision Handling	Description	Pros	Cons
Chaining	Store collided items in linked lists	Simple, flexible	Extra memory for lists
Open Addressing	Probe for next empty slot	Uses less memory	Clustering can slow down search

---

Why collisions happen despite good hash functions?

- Finite table size vs infinite possible keys.
- Some hash functions aren't perfect at spreading keys uniformly.
- Load factor (table fullness) affects collision frequency.

If you want, I can:

- Show you a full hash table implementation with collision handling.
- Explain pros and cons of each collision strategy more deeply.
- Show how collision handling impacts performance.
- Discuss **rehashing** and resizing hash tables.

---

42. What is linear probing, quadratic probing, and chaining?

1️⃣ Linear Probing

- When a collision occurs at index `i`, try the next slot `i+1`.
- If `i+1` is occupied, try `i+2`, and so on, **linearly scanning** the table until an empty slot is found.
- Wrap around to the beginning if the end is reached.

Pros:

- Simple to implement.
- Good cache performance (due to locality).

Cons:

- Can cause **primary clustering** — long runs of occupied slots, slowing down insertion and search.

Example:

If collision at index 5, try 6, then 7, etc.

2️⃣ Quadratic Probing

- Instead of moving linearly, the probe sequence moves according to a quadratic function:
  [
  i, (i + 1^2), (i + 2^2), (i + 3^2), \ldots
  ]
- For collision at `i`, next checks are at `i+1`, `i+4`, `i+9`, etc.

Pros:

- Reduces clustering compared to linear probing.
- Avoids primary clustering but can still have **secondary clustering**.

Cons:

- More complex probing sequence.
- May not always find an empty slot if the table isn't sized properly.

3️⃣ Chaining

- Each slot in the hash table stores a **linked list** (or another data structure) of all keys that hash to the same index.
- On collision, just **append the new key** to the list at that slot.

- Lookup involves searching the list for the key.

Pros:

- Simple and effective.
- Table size can remain smaller.
- Can handle high load factors gracefully.

Cons:

- Uses extra memory for pointers.
- Lookup time depends on the length of the chain (can degrade to O(n) if many collisions).

Quick Comparison:

Method	Memory Usage	Speed (Average)	Clustering Issue	Implementation Complexity
Linear Probing	No extra memory	Fast	Primary clustering	Simple
Quadratic Probing	No extra memory	Fast	Secondary clustering	Moderate
Chaining	Extra memory for lists	Depends on chain length	None	Simple

---

Simple Example of Linear Probing (Python):

```python
def linear_probing_insert(hash_table, key, size):
    index = hash(key) % size
    original_index = index
    while hash_table[index] is not None:
        index = (index + 1) % size
        if index == original_index:
            raise Exception("Hash table is full")
    hash_table[index] = key
```

---

42. Difference between a hash map and a hash set.

🔑 Key Differences between Hash Map and Hash Set

Aspect	Hash Map	Hash Set
Purpose	Stores **key-value pairs**	Stores **unique keys** only
Data Stored	Each entry: (key, value)	Each entry: key
Lookup	Find value by key	Check presence of a key
Duplicates	Keys must be unique; values can be duplicated	No duplicates allowed (keys unique)
Typical Use	When you want to associate values with keys (like a dictionary)	When you just need to track existence or uniqueness of items
Operations	Insert, delete, update, get by key	Insert, delete, check membership

---

Examples:

- **Hash Map:**
  Store student grades: `{ "Alice": 85, "Bob": 92 }`
  You can get Bob's grade by key `"Bob"`.
- **Hash Set:**
  Store list of unique users who visited a website: `{ "Alice", "Bob", "Charlie" }`
  You can quickly check if `"Alice"` has visited.

Summary in Python terms:

Data structure	Python equivalent	Usage
Hash Map	`dict`	Key-value pairs
Hash Set	`set`	Unique collection of keys

🚀 Advanced Level (Problem Solving)

💡 Common Coding Questions

44. Reverse a linked list.

45. Find the middle element of a linked list.
46. Detect and remove loop in a linked list.
47. Implement two stacks using one array.
48. Implement a queue using two stacks.
49. Find the next greater element for each element in an array.
50. Implement a min-heap.
51. Merge two sorted linked lists.
52. Find the kth smallest element in an array.
53. Implement BFS and DFS for a graph.
54. Find the shortest path in an unweighted graph.
55. Check if two binary trees are identical.
56. Find the lowest common ancestor (LCA) in a binary tree.
57. Serialize and deserialize a binary tree.
58. Implement LRU Cache.
59. Find all connected components in a graph.
60. Detect cycle in directed and undirected graphs.

⚡ Bonus: Time Complexity Quick Reference

Operation	Array	Linked List	Stack	Queue	BST (Avg)	Hash Table
Access	O(1)	O(n)	O(n)	O(n)	O(log n)	O(1)
Search	O(n)	O(n)	O(n)	O(n)	O(log n)	O(1)
Insert	O(n)	O(1)	O(1)	O(1)	O(log n)	O(1)
Delete	O(n)	O(1)	O(1)	O(1)	O(log n)	O(1)

🧠 Bonus HR/Conceptual Questions

61. Which data structure would you use for expression evaluation?
62. Which data structure is best for implementing recursion?
63. What is the difference between BFS and DFS in terms of space and time complexity?
64. Why is a hash table faster than a binary search tree in average cases?
65. When would you prefer a linked list over an array?
