Chapter # 04
Computational Structures

Q.1. What is a list in Python? Explain its properties with examples.

Ans. In our daily life, we often need to organize and manage different types of information. For example, we save contacts on our phone, write a to-do list to manage tasks, or stack plates one by one in the kitchen.

Similarly, in the world of computers, we need to store and manage data in an organized way. This is done with the help of **computational structures**.

One of the most commonly used computational structures in Python is the **list**. A list allows us to store **multiple values in a specific order**, manage them easily, and perform various operations on the data.

List in Python:

A **list** is a data structure in Python that is used to store **multiple types of data** in a **specific sequence**. Each piece of data in a list is called an **element**, and it is placed at a specific **index** to allow easy access and management.

List Creation:

Lists in Python are created using **square brackets [ ]**. Each item in the list is separated by a **comma**.

Pythod Code:

```
# Create a list of items
items_list = ["Cup", "Cola", "Cold drinks", "Glass", "Balloons"]
# Print the list
print(items_list)
```

In the above code:

- **Items_list** is the name of the list.
- The list contains elements: "Cup", "Cola", "Cold drinks", "Glass", and "Balloons".
- Each text item is written in **quotation marks** and separated by a **comma**.
- The **print()** function is used to display the list.

Lists allow us to **add**, **remove**, or **change** items. They are useful for managing multiple values in one variable.

Properties of Lists:

1. **Dynamic Size:**

   A list in Python can change its size. We can add or remove items in a list, and the size of the list will automatically adjust.

   Example:

```
In [2]: items = ["Decorations", "Snacks"]
        items.append("Cold drinks")  # Adds a new item
        items.remove("Snacks")       # Removes an item
        print(items)  # Output: ['Decorations', 'Cold drinks']
```

```
['Decorations', 'Cold drinks']
```

   - This dynamic resizing is a core feature of Python lists.

2. **Index-Based Access**

   Every item in a list has a position called an **index**. Indexing starts at 0. The first item is at index 0, the second at index 1, and so on.

   You can access or modify any item directly using its index.

Example:

```
In [3]: items = ["Decorations", "Snacks", "Cold drinks"]
        print(items[1])  # Output: Snacks
```

```
Snacks
```

- This allows for fast and direct retrieval or assignment of values.

3. **Maintains Order:** A list keeps the **same order** in which the items are added. The sequence of items is preserved **unless we change them manually** (e.g., using sort() or reverse()).

Example:

```
In [4]: items = ["A", "B", "C"]
        print(items)  # Output: ['A', 'B', 'C']
```

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

4. **Store Different Types of Data (Mixed Data Types or Heterogeneous Elements)**

A list can store **different types of values** at the same time. For example, it can have **numbers, text (strings), decimal values, Booleans**, or even **another list** inside it.

Example:

```
In [5]: mixed = ["Apple", 42, 3.14, True, [1, 2, 3]]
        print(mixed)
```

```
['Apple', 42, 3.14, True, [1, 2, 3]]
```

This makes lists highly versatile

5. **Changeable (Mutable)**

Lists are **mutable**, which means they can be **changed after creation**. We can **add new items, remove existing ones**, or **update values** anytime without making a new list.

Example:

```
In [6]: items = ["A", "B", "C"]
        items[1] = "Z"
        print(items)  # Output: ['A', 'Z', 'C']
```

```
['A', 'Z', 'C']
```

This property distinguishes lists from immutable types like tuples.

6. **Allows Repeated Values (Supports Duplicates)**

A list can contain the **same value more than once**. This means **duplicate items** (repeating values) are allowed in a list.

Example:

```
In [7]: items = ["A", "B", "A", "C"]
        print(items)  # Output: ['A', 'B', 'A', 'C']
```

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

Duplicates are allowed and retained in the order added.

Q.2. Write a short program to show how to create and modify a list.

Ans. Example Program:

File Edit Format Run Options Window Help

```
# Example List
items = ["Books", "Notebook", "Pencil", "Eraser", "Sharpener"]
# Nature of the Data Structure
print("List of items:", items)
# Mutability (Changeable)
items.append("Ruler")          # Add new item
items.remove("Notebook")       # Remove an item
print("After modifications:", items)
# Homogeneous vs Heterogeneous
print("Homogeneous List:", [10, 20, 30, 40])
print("Heterogeneous List:", ["Orange", 99, 4.5, False])
# Properties of Lists
print("Number of items:", len(items))                    # Length
print("Item at index 2:", items[2])                      # Indexing
print("Order maintained:", items)                        # Order
print("List with duplicates:", ["Books", "Pencil", "Books", "Marker"])
```

Explanation:

- **List Creation:** A list named **items** is created with school stationery items.
- **Nature of the Data Structure:** The full list is displayed using the **print()** function.
- **Mutability:** The **append()** function adds "Ruler", and **remove()** deletes "Notebook", showing that lists can be changed after creation.
- **Homogeneous vs Heterogeneous:** One list contains only numbers (homogeneous), and the other contains mixed data types (heterogeneous).
- **Length: len()** shows the total number of items in the list.
- **Indexing:** Using index 2, the third item in the list is accessed.
- **Ordering:** The order in which items are added is maintained.
- **Duplicates:** A list with repeated values ("Books") shows that duplicates are allowed.

Output :

```
List of items: ['Books', 'Notebook', 'Pencil', 'Eraser', 'Sharpener']
After modifications: ['Books', 'Pencil', 'Eraser', 'Sharpener', 'Ruler']
Homogeneous List: [10, 20, 30, 40]
Heterogeneous List: ['Orange', 99, 4.5, False]
Number of items: 5
Item at index 2: Eraser
Order maintained: ['Books', 'Pencil', 'Eraser', 'Sharpener', 'Ruler']
List with duplicates: ['Books', 'Pencil', 'Books', 'Marker']
>>>
```

Q.3. Discuss the dynamic size property of lists in Python. How does this property make lists more flexible?

Ans. In Python, **lists have a dynamic size**, which means the number of elements in a list can be **increased or decreased at any time**. We do not need to declare the size of the list in advance. This feature makes lists very flexible and easy to use.

Explanation:

When we create a list in Python, we can start with a few items. Later, we can **add more items using functions like append() or insert()**, and we can **remove items using remove() or pop()**. The list will automatically adjust its size as per the changes.

Why Is It Flexible?

- We do not need to know how many items will be stored before creating the list.

- We can easily manage data as it grows or shrinks.
- This is very useful in programs where the amount of data is not fixed.

Example:

Python Code:

```
File  Edit  Format  Run  Options  Window  Help
items = ["Pen", "Pencil"]
items.append("Eraser")        # Adding new item
items.remove("Pen")           # Removing an item
print(items)
```

Output:

```
>>> ['Pencil', 'Eraser']
```

In the example above, the size of the list changes when an item is added or removed. Python handles this automatically.

The **dynamic size property** of lists in Python allows programmers to **store, update, and manage data easily without worrying about size limitations.** This makes lists a powerful and flexible data structure in Python programming.

Q.4. Explain common list operations in Python with examples. Also discuss any two important applications of lists.

Ans. In Python, **lists are powerful and flexible data structures** that allow us to **store, manage, and organize multiple values**. To work with lists effectively, Python provides several operations that help in adding, removing, and searching for items. These are known as **list operations.**

Common List Operations:

1. **Insertion**

   This means **adding a new item to the list**. You can add an item at any position using the insert() function.

   Example:

   Python Code:

   ```
   subjects = ["Physics", "Math", "Computer"]
   subjects.insert(1, "English")
   print(subjects)
   ```

   Output:

   ```
   ['Physics', 'English', 'Math', 'Computer']
   ```

   Explanation:

   - A list named **subjects** is created with **items**: "Physics", "Math", and "Computer".
   - The **insert(1, "English")** function adds "English" at **index 1**.
   - The **updated list** becomes: ['Physics', 'English', 'Math', 'Computer'].

2. **Deletion**

   This means **removing items** from a list. There are two common methods:

a. **Removing by Value** – using **remove()**

   Python Code:

   ```
   fruits = ["Apple", "Banana", "Mango", "Orange"]
   fruits.remove("Mango")
   print(fruits)
   ```

   Output:

   ```
   ['Apple', 'Banana', 'Orange']
   ```

b. **Removing by Index – using pop()**

Python Code:

```
fruits = ["Apple", "Banana", "Orange"]
fruits.pop(1)
print(fruits)
```

Output:

```
['Apple', 'Orange']
```

Explanation:

- A list named fruits is created with three items: "Apple", "Banana", and "Orange".
- The pop(1) function removes the item at index 1, which is "Banana".
- The updated list is printed as: ['Apple', 'Orange'].

3. **Searching**

To **check whether an item exists** in a list, we use the **in** keyword.

Example:

Python Code:

```
city_list = ["Lahore", "Karachi", "Islamabad"]
if "Karachi" in city_list:
  print("Karachi is in the list.")
else:
  print("Karachi is not in the list.")
```

Output:

```
Karachi is in the list.
```

Explanation:

- A list named **city_list** is created with three cities: "Lahore", "Karachi", and "Islamabad".
- The **condition** checks if "Karachi" is **present** in the list using the **in** keyword.
- Since "Karachi" exists in the list, the output will be: **"Karachi is in the list."**

Applications of Lists:

1. **Using Lists to Store and Manage Data**

Lists are commonly used to **organize collections of information** such as student names, item prices, or scores. They make it easy to **add, remove, or access** items, which helps in managing and manipulating data efficiently. Lists are a very useful tool in everyday programming tasks because they allow for quick changes and retrieval of information.

2. **Stack and Queue Implementations**

Lists can be used to create **stacks** (Last In, First Out – LIFO) and **queues** (First In, First Out – FIFO). These structures are important in **computer programs** for tasks like managing **browsing history, customer service systems**, or organizing **task management**. By using lists, we can organize data in a specific order to make processing easier and more efficient.

3. **Sorting and Searching Algorithms**

Lists are commonly used as the underlying structure for sorting (e.g., bubble sort, selection sort) and searching (e.g., linear search, binary search) algorithms. These operations are crucial for organizing and retrieving data efficiently.

4. **Function Arguments**

Lists enable passing multiple values as arguments to functions, allowing for flexible and efficient input handling. This is especially useful when the number of arguments is variable or unknown in advance.

5. **Matrix Operations**

Lists of lists can represent matrices, which are widely used in numerical and scientific computing. Each inner list can represent a row or column, enabling operations such as addition, multiplication, or transposition.

Q.5. Define stack & explain the operations on stack with real life example and python code.

Ans. A **stack** is a simple and important data structure that works on the **Last-In, First-Out (LIFO)** principle. This means the item that is added last will be the first one to be removed.

In a stack, we can only insert or remove elements from **one end**, which is called the **top** of the stack.

Real-Life Example:

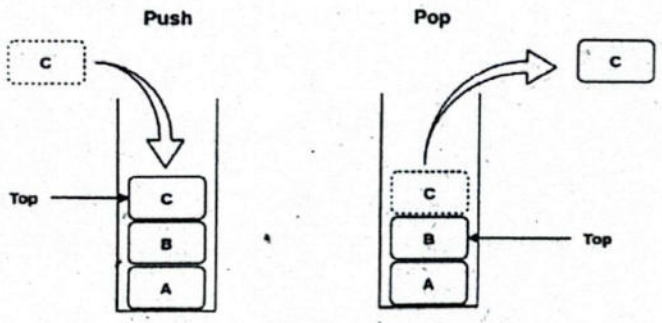
A common example of a stack is a **pile of notebooks** on a table.

- When we add a new notebook, we place it on the **top** of the pile.
- When we remove a notebook, we also take it from the **top**.

This means the **last notebook placed is the first one removed**, just like in a stack.

It also has the following properties:

- **Single Access Point :** Only the top of the stack can be accessed for adding or removing items.
- **Order Preservation :** The order in which items are added is preserved, with the last item added being the first to leave.
- **Size Flexibility :** A stack can either have a fixed size or grow dynamically as needed.



Stack Data Structure

Stack Operations:

There are two main operations in a stack:

1. **Push Operation**
   - It means **adding an item to the top** of the stack. In Python, this is implemented using the append() method of a list.
   - Example: Placing a notebook on the top of the pile.
2. **Pop Operation**
   - It means **removing the item from the top** of the stack. In Python, this is implemented using the pop() method of a list.
   - Example: Taking the top notebook off the pile.

Python Code Example:

Below is a simple Python program to show how a stack works:

Python Code:

```
# Create an empty stack
notebook_stack = [ ]
print("Initial stack:", notebook_stack)

# Push operation - Adding notebooks to the stack
print("\nAdding notebooks to the stack (push operation):")
notebook_stack.append('Notebook 1')
print("Stack after pushing 'Notebook 1':", notebook_stack)
notebook_stack.append('Notebook 2')
print("Stack after pushing 'Notebook 2':", notebook_stack)

# Pop operation - Removing the top notebook
print("\nRemoving top notebook (pop operation):")
top_notebook = notebook_stack.pop()
print("Removed notebook:", top_notebook)
print("Stack after popping the top notebook:", notebook_stack)
```

Output:

```
Initial stack: []

Adding notebooks to the stack (push operation):
Stack after pushing 'Notebook 1': ['Notebook 1']
Stack after pushing 'Notebook 2': ['Notebook 1', 'Notebook 2']

Removing top notebook (pop operation):
Removed notebook: Notebook 2
Stack after popping the top notebook: ['Notebook 1']
```

Explanation of Code:

- We first create an **empty stack** named notebook_stack.
- We use append() to **push** 'Notebook 1' and 'Notebook 2' onto the stack.
- We use pop() to **remove** the top item, which is 'Notebook 2', showing the **LIFO** behavior.

Q.6. What is a queue in data structures? How do Enqueue and Dequeue operations work? Explain with an example.

Ans. A **queue** in data structures is a collection where the first item added is the first one to be removed. This method of working is called **FIFO** – *First In, First Out*. It is just like people standing in a line at a bank or a ticket counter. The person who comes first is served first.

There are two main operations in a queue:

1. **Enqueue (Add an Item)**

   This means adding a new item at the **end (back)** of the queue.

   Example: If three people named A, B, and C join the queue one after another, A is added first, then B, and then C.

   Each new person is added at the **back (or end)** of the queue.

   So, the order of the queue becomes: **A → B → C**, where A is at the front and C is at the back.

2. **Dequeue (Remove an Item)**

   This means removing the item from the **front** of the queue.

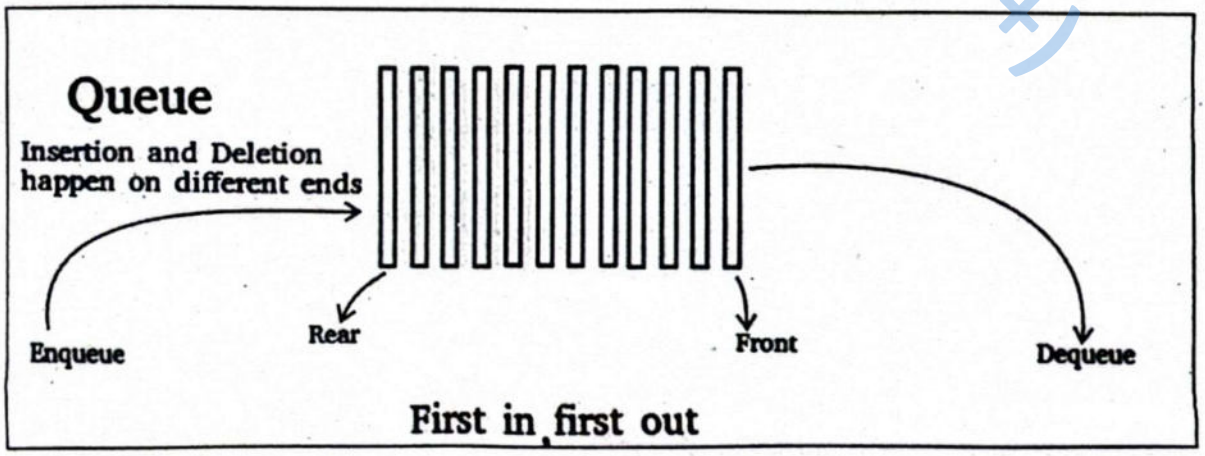
   Example: If the queue has three people in the order **A → B → C**, then when we perform a **Dequeue** operation, person **A** will be removed first because A is at the **front** of the queue.

   This shows that in a queue, the item added first is also the first to be removed.

Additional operations include:

- **Check if Empty:** Checking if the queue is empty.
- **Peek:** Looking at the front item without removing it.
- **Size:** Checking how many items are in the queue.

In short, a queue is a simple data structure that works just like a real-life line where the first person who enters is the first to leave.



Q.7. Write, a simple program to implement a queue (insertion and deletion).

Ans. In Python, we can use the **queue module** to implement a queue. A queue follows the **FIFO** (First-In, First-Out) principle. Below is a simple program that performs **insertion (enqueue)** and **deletion (dequeue)** operations:

Python Code:

```
# Import the built-in module to use queue
from queue import Queue
# Create a new queue
my_queue = Queue()
# Add elements to the queue (Enqueue)
my_queue.put("Ali")      # Adds 'Ali' to the end of the queue
my_queue.put("Zara")     # Adds 'Zara' to the end of the queue

# View the queue
v_queue = list(my_queue.queue)
print("Current queue:", v_queue)
# View the person at the front of the queue (Peek)
first_person = my_queue.queue[0]
print("Person at the front:", first_person)
# Remove an element from the front of the queue (Dequeue)
removed_person = my_queue.get()
print("Removed from front:", removed_person)
# View the updated queue
current_queue = list(my_queue.queue)
print("Updated queue:", current_queue)

# Add another person to the queue (Enqueue)
print("Add another element")
my_queue.put("Hassan")   # Adds 'Hassan' to the end of the queue

# View the updated queue
current_queue = list(my_queue.queue)
print("Current Updated queue:", current_queue)
```

Output:

```
Current queue: ['Ali', 'Zara']
Person at the front: Ali
Removed from front: Ali
Updated queue: ['Zara']
Add another element
Current Updated queue: ['Zara', 'Hassan']
```

Explanation:

1. **Import Queue Module:**
   from queue import Queue imports the built-in Queue class.
2. **Create Queue:**
   my_queue = Queue() creates an empty queue.
3. **Enqueue Elements:**
   put() adds elements to the **end** of the queue.
   "Ali" and "Zara" are added.

4. **View Queue:**
   list(my_queue.queue) shows the current items in the queue.
5. **Peek Front Element:**
   my_queue.queue[0] shows the **first item** without removing it.
6. **Dequeue Element:**
   get() removes the **first item** (FIFO – Ali is removed).
7. **View Updated Queue:**
   Shows queue after removing one element.
8. **Add Another Element:**
   "Hassan" is added at the end using put().
9. **Final Queue Display:**
   Prints the updated queue again.

Q.8. What is a Tree data structure? Explain its properties and applications with examples. / Define Tree and explain its properties.

Ans. A **tree** is a type of data structure used in computer science to organize data in a **hierarchical** way. It starts with a **root node**. The root node connects to other **nodes** through **branches**. This forms a structure that looks like a tree.

Unlike a **list**, where elements are arranged in a straight line, a tree allows each node to connect to multiple **child nodes**. This makes it very useful for storing and working with data that has **parent-child relationships**.

Why Use Trees?

- Trees efficiently represent data with a natural hierarchy, such as family trees, file systems, and organizational charts.
- They allow for fast searching, insertion, and deletion, especially in balanced trees.

Example:

A **family tree** is a common example of a tree structure. The oldest ancestor is the **root**, and their children, grandchildren, and so on are the **child nodes**, forming different levels in the hierarchy.

Properties of Trees:

1. **Root Node**
   The **root** is the **topmost node** in a tree. It is the starting point from where all other nodes branch out.
   Example: In a computer, the main folder (C:) is like the root node that contains all other files and folders.
2. **Nodes and Edges**
   - A **node** is an individual element in a tree.
   - An **edge** is the line or connection between two nodes.
   - A node that has no children is called a **leaf node**.

   Example: A file that does not contain any other files is a leaf node.
3. **Height of a Tree**
   The height of a tree is the length of the **longest path** from the root node to the **last leaf node**. It shows how **deep or tall** the tree is.
4. **Balanced Tree**
   A tree is called **balanced** if both left and right sides have almost equal height. Balanced trees help in faster searching and better performance.

Applications of Trees:

1. **File System Backup (Using Pre-order Traversal)**
   In computers, folders and files are stored in a tree structure.
   Pre-order traversal means visiting the **main folder first**, then all its subfolders and files.
   This is useful when creating a **backup**, so the folder is saved before its contents.

2. **File Deletion (Using Post-order Traversal)**

When we want to delete a folder and everything inside it, we use post-order traversal.

It deletes **all the files and subfolders first**, then deletes the main folder.

This way, nothing is left behind.

3. **Hierarchical Data Representation**

Trees are best for showing data that has levels or rankings.

Examples include **family trees, company staff charts,** or **school systems.**

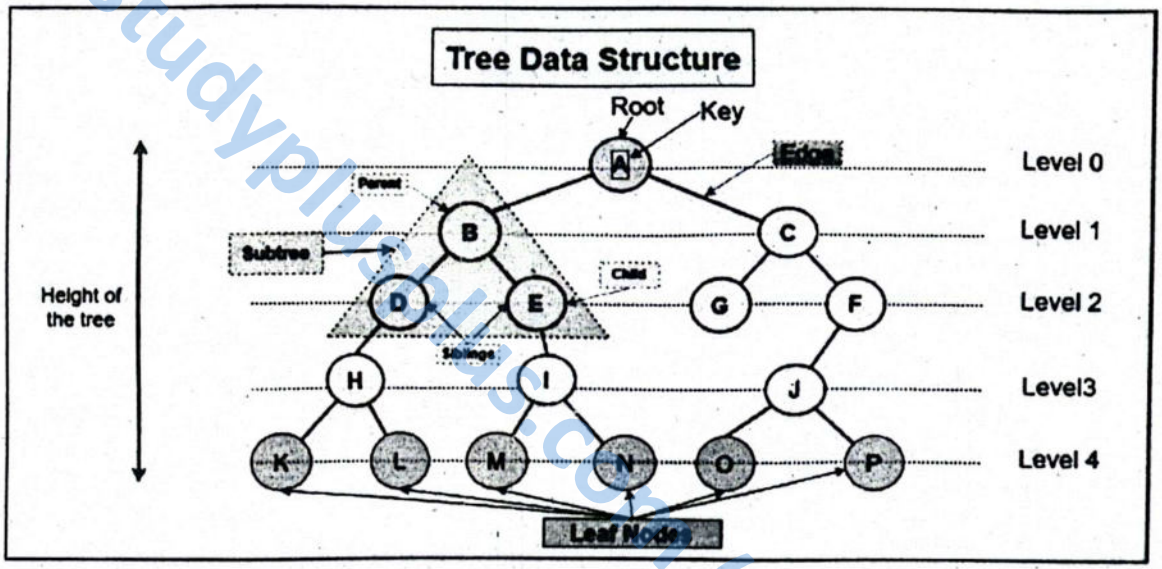
Each level shows who comes under whom, like parents and children.

4. **Decision Making (Decision Trees)**

In programming and artificial intelligence, **decision trees** are used to make choices.

Based on **questions and answers,** the computer can decide what to do next.

This is used in games, apps, and many smart systems.



Q9. What is a Graph in data structures? Explain its characteristics, properties, and difference from a Tree with examples.

Ans. Definition of Graph:

A **Graph** is a type of data structure that is used to show **relationships or connections** between different items. It consists of two main parts:

- **Vertices (also called Nodes):** These represent the individual objects.
- **Edges**: These are the connections or links between the vertices.

Graphs are used to model **networks**, such as cities connected by roads, people connected in social media, or websites connected through hyperlinks.

Example of a Graph:

Imagine a map of Pakistan. Each **city** is a **vertex**, and the **roads** between them are **edges**. This is a real-life example of a graph. We can travel from one city to another using these connecting roads (edges).

Importance:

Graphs are powerful data structures that help represent **complex relationships** in a clear and flexible way. They are widely used in **networking, social media, maps, and computer science**. Unlike trees, graphs allow **multiple connections and paths**, which makes them ideal for real-world systems that don't follow a strict hierarchy.

Difference Between Graph and Tree

Tree	Graph
Has a **single root node** and follows a **hierarchy**.	No root node and no fixed hierarchy.
Only **one path** between two nodes (no cycles).	Can have **multiple paths** and **cycles**.
Best for **structured data** like family trees or file systems.	Best for **networked data** like maps or social media.

Characteristics of Graphs

1. **Vertices (Nodes):**
   These are the **main elements or points** in the **graph**.
   Example: *In a **social network**, each **person** is a **vertex**.*
2. **Edges (Links):**
   These are the **connections** between the **vertices**.
   Example: *In a **transport system**, the **road** between **two cities** is an **edge**.*

Properties of Graphs:

- **Degree:**
  The **degree** of a vertex is the **number of connections (edges)** it has with other vertices.
  Example: If **Lahore** is connected to **3 cities**, then the degree of Lahore is **3**.
- **Weight:**
  Some edges have a **value**, like **distance, cost, or time**. This value is called the **weight** of the edge.
  Example: If a road between two cities is **100 km**, then the edge has a **weight of 100**.
- **Direction:**
  Direction in a graph tells us **whether the connection (edge) between two points goes one way or both ways.** Some graphs show **one-way connections**, and some show **two-way connections.**
  There are **two types** of edges based on direction:
  - **Directed Graph (One-way connection):**
    In this type, edges go **in only one direction**, just like a **one-way road.**
    Example: If a bus goes from City A to City B, but not back, it is a **directed edge.**
  - **Undirected Graph (Two-way connection):**
    In this type, edges go **both ways**, just like a **two-way road.**
    Example: If people can travel freely between City X and City Y in both directions, it is an **undirected edge.**

Q.10 What is graph? Explain differences between Directed and Undirected Graphs. / Explain types of graph.

Ans. What is a Graph?

A **graph** is a data structure used in computer science to show how things are connected. It is made up of:

- **Vertices (Nodes):** These are the individual points or items in a graph.
  Example: In a social network, each person is a vertex.
- **Edges (Links):** These are the lines that connect one vertex to another.
  Example: In a map, a road between two cities is an edge.

Graphs are commonly used to represent **networks**, such as city maps, transport systems, and online connections.

Types of Graph:

Graphs come in different types based on how they are connected and how they work. The main types are:

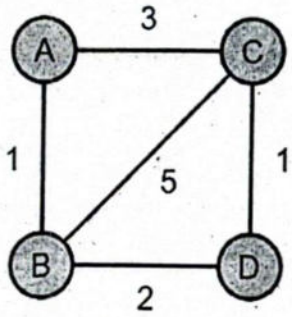
1. **Directed Graph (One-way)**
   - In a **directed graph**, the edges go in one direction only.
   - This means the connection moves from one node to another in a specific direction.
   - You cannot move back unless there is a separate edge for the return.
     Example: If there is a one-way road from **City A to City B**, you can go from A to B, but not back to A.

2. **Undirected Graph (Two-way)**
   - In an **undirected graph**, the edges **have no direction**.
   - It shows a **two-way connection** between the vertices.
   - You can **move in both directions** freely.

   Example: If **Person A is friends with Person B**, then **Person B is also friends with Person A.**
3. **Weighted Graphs**

   A weighted graph has numbers (called weights) on the edges. These weights show things like distance, cost, or time between two points.

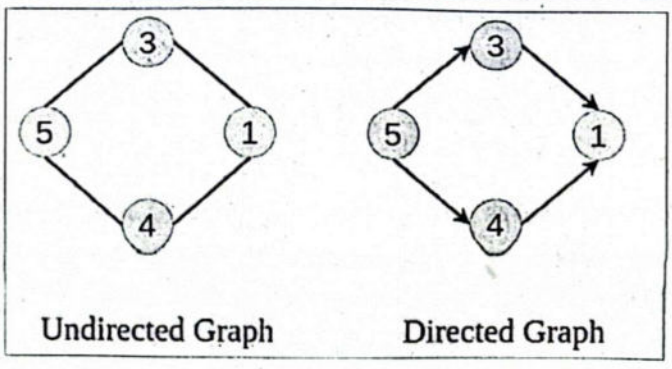
   Example: Think of a city map where roads between places have distances or travel times written on them. This helps you choose the shortest or fastest route to your destination.



Weighted graph

Difference Between Directed and Undirected Graph

Directed Graph	Undirected Graph
Edges have a **fixed direction** (one-way).	Edges have **no direction** (two-way).
Movement is possible **only from one node to another.**	Movement is possible **in both directions.**
Represent **one-way relationships.**	Represent **mutual relationships.**
Commonly used in **road maps with one-way streets,** workflows, or instructions.	Commonly used in **social networks,** friendship maps, or networks with mutual connections.
Each edge goes from **source to destination** (e.g., A → B).	Each edge connects **both nodes equally** (e.g., A — B).
Different directions between two nodes are possible.	**Only one connection** is enough for both directions.
Requires **two edges** for two-way connection (A → B and B → A).	**One edge** is enough for two-way connection (A — B).
More complex in structure and analysis.	**Simpler** to represent and understand.
Example: One-way Road from City A to City B allows travel **only from A to B.**	**Example**: If Ali is friends with Sara, then **Sara is also friends with Ali.**



Undirected Graph Directed Graph

Exercise
Multiple Choice Questions

1. **The function used to add an item at the end of a list in Python:**
   a) insert( ) b) append() c) remove() d) pop()
2. **The purpose of the in keyword used with a Python list:**
   a) Adds an item to the list b) Removes an item from the list
   c) Checks if an item exists in the list d) Returns the length of the list
3. **An operation that removes an item from the top of the stack:**
   a) Push b) Pop c) Peek d) Add
4. **The operation used to add an item to a queue:**
   a) Dequeue b) Peek c) Enqueue d) Remove
5. **True statement about the height of a tree:**
   a) Number of edges from the root to the deepest node
   b) Number of nodes from the root to the deepest node
   c) Number of children of the root node
   d) Always equal to the number of nodes in the tree
6. **A scenario where a graph data structure is most suitable:**
   a) Managing a to-do list b) Modeling a line of customers in a store
   c) Representing connections in a social network d) All of the above

Correct Answers

Q. No.	Correct Option
1.	b) **append()** - Adds an item at the end of a list in Python
2.	c) **Checks if an item exists in the list** - The purpose of the in keyword
3.	b) **Pop** - Removes an item from the top of the stack
4.	c) **Enqueue** - The operation used to add an item to a queue
5.	a) **Number of edges from the root to the deepest node** - True statement about the height of a tree
6.	c) **Representing connections in a social network** - A scenario where a graph data structure is most suitable

SHORT ANSWER QUESTIONS

Q.1. Explain how the 'insert()' function works in python lists. Provide an example.

Ans. The insert() function in Python is used to **add an item at a specific position** in a list.
It takes two arguments: the **index number** where the item should go, and the **item** itself.
Other items in the list are **shifted to the right** to make space for the new item.
Example:

```
subjects = ["Math", "Computer", "Physics"]
subjects.insert(1, "English")
print(subjects)
```

Output:

```
= RESTART: C:/Users/hp/OneDrive/Documents/pyth
['Math', 'English', 'Computer', 'Physics']
>>>
```

In this example, "**English**" is added at index 1, pushing "**Computer**" and "**Physics**" to the right.

Q.2. **Explain the potential issues which could arise when two variables reference the same list in a program? Provide an example.**

Ans. When two variables reference the same list, **changing the list using one variable will also affect the other,** because both are pointing to the same memory location.

This can cause **unexpected changes** in your data, especially if you are not aware that both variables are linked.

Example:

Python Code:

```
list1 = ["Math", "English"]
list2 = list1        # list2 points to the same list as list1
list2.append("Physics")
print(list1)
```

Output:

```
['Math', 'English', 'Physics']
```

Even though we added "Physics" using list2, it also changed list1.

This is because both variables refer to the same list.

Q.3. **Define a stack and explain the Last-In, First-Out (LIFO) principle.**

Ans. A **stack** is a linear data structure that allows data to be added and removed **only from one end,** which is called the **top** of the stack. It works on the **Last-In, First-Out (LIFO)** principle.

The **LIFO** principle means that the **last item inserted** into the stack is the **first one to be removed.**

A real-life example is a **stack of plates,** where the last plate placed on top is the first one to be taken off.

Q.4. **Differentiate between the Enqueue and Dequeue operations of queue. / What is difference between enqueue () and dequeue.**

Ans.

Enqueue	Dequeue
Enqueue means **adding** an element to the **end** (rear) of the queue.	Dequeue means **removing** an element from the **front** of the queue.
It is used to insert new data into the queue.	It is used to remove the oldest data from the queue.
Example: If we write queue.put("Ali"), it adds "Ali" to the queue.	**Example:** If we write queue.get(), it removes the person at the front (like "Ali").

Q.5. **Name two basic operations performed on stack**

Ans. Two basic operations performed on a stack are:

1. **Push Operation:**
   Adds an element to the top of the stack.
   Example: If the stack is [A, B] and we push C, the stack becomes [A, B, C].
2. **Pop Operation:**
   Removes the top element from the stack.
   Example: If the stack is [A, B, C] and we pop an element, the stack becomes [A, B].

Long Question Answers

Q.1. **Discuss the dynamic size property of lists in Python. How does this property make lists more flexible?**

Ans. See Answer of chapter question # 3

Q.2. **Explain the operations on stack with real life example and python code.**

Ans. See answer of chapter question # 5

Q.3. **Write, a simple program to implement a queue (insertion and deletion).**

Ans. See answer of chapter question # 7

2

Q.4. **Define Tree and explain its properties**

Ans. See answer of chapter question # 8

Q.5. **What is graph? Explain differences between directed and undirected graphs.**

Ans. See answer of chapter question # 10

SLO Based Questions

Q.1. **What is a list in Python and how does it differ from other data structures?**

Ans. A **list** in Python is a collection of elements stored in a specific order, and the elements can be of different data types. Lists are created using square brackets [ ] and are **mutable**, meaning elements can be added, removed, or changed after creation.

Difference from Other Data Structures:

Tuple: A tuple is similar to a list but is **immutable**, meaning its elements cannot be changed once created.

Q.2. **What are the main properties of Python lists?**

Ans. The main properties of lists are:

- **Dynamic Size:** Lists can grow or shrink as needed.
- **Ordered:** Elements maintain the order of insertion.
- **Mutable:** Lists can be modified by adding, removing, or changing elements.
- **Heterogeneous:** Lists can store elements of different data types.
- **Supports Duplicates:** Lists can contain duplicate elements.

Q.3. **How does the append() method work in Python lists?**

Ans. The **append()** method adds a single item to the end of specified list.

Example:

Python code:

```
fruits = ["apple"]
fruits.append("banana")
print(fruits)
# Output:
['apple', 'banana']
```

Q.4. **What is the difference between the pop() and remove() methods in lists?**

Ans.

pop()	remove()
Removes and returns an item from a specified index. If no index is given, it removes the last item.	Removes the first occurrence of a specified item in the list.
Returns the item that was removed from the list.	Does not return any value.
If the index is out of range, it raises an **IndexError**.	If the item is not found, it raises a **ValueError**.
Useful when you want to remove an item at a specific position or the last item in the list.	Useful when you want to remove a specific item by value, not position.

Example:
`fruits = ["apple", "banana", "cherry"]`
`fruits.pop(1)`
`print(fruits)`
Output: ["apple", "cherry"]
	Example:
	`fruits = ["apple", "banana", "cherry"]`
	`fruits.remove("banana")`
	`print(fruits)`
	Output: ["apple", "cherry"]

Q.5. How do you check if an item exists in a Python list?

Ans. You can check if an item exists in a list using the **in** keyword. It returns True if the item is found and False otherwise.

Example:

Python Code

```
fruits = ["apple", "banana", "cherry"]
if "banana" in fruits:
  print("Banana is in the list.")
# Output :
Banana is in the list.
```

Q.6. Explain how the insert() method works in Python lists.

Ans. The **insert()** method adds an item at a specific index in the list, shifting the other elements to the right. It takes two arguments: the index where the item should be inserted and the item itself.

Example:

Python Code:

```
fruits = ["apple", "banana"]
fruits.insert(1, "orange")
print(fruits)
# Output :
['apple', 'orange', 'banana']
```

Q.7. Discuss how lists are useful in real-world applications like stacks and queues.

Ans. Lists in Python are very useful in real-world programming because they can be used to create **stacks** and queues.

- A **stack** follows the **Last In First Out (LIFO)** method, like a pile of books; the last book placed is the first to be removed.
- A **queue** follows the **First In First Out (FIFO)** method, like a line at a ticket counter; the first person in line is served first.

Python lists allow us to easily add and remove items from both the front and end, which makes them perfect for these types of tasks.

Q.8. What is meant by the dynamic size property of Python lists?

Ans. The **dynamic size** property means that Python lists can automatically resize when items are added or removed. There's no need to pre-define the size of a list.

Example:

Python Code:

```
fruits = ["apple"]
fruits.append("banana")  # List grows automatically
print(fruits)
# Output :
['apple', 'banana']
```

Q.9. How do lists help with data storage and manipulation?

Ans. Lists are useful for **data storage and manipulation** as they allow for easy insertion, deletion, and access to elements. They are ideal for storing collections of items like names, prices, or scores and can be modified as required.

Q.10. What is the purpose of using the remove() method in Python lists?

Ans. The **remove()** method is used to remove the first occurrence of a specific item in a list. If the item is not found, it raises a **ValueError**.

Example:

Python Code:

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)
```

```
# Output :
['apple', 'cherry']
```

Q.11. How can lists be used to implement a queue in Python?

Ans. A queue in Python can be implemented using a list by adding elements to the end using the **append()** method and removing elements from the front using **pop(0)**, which follows the FIFO (First In First Out) principle.

Example:

```
queue = [ ]
queue.append("Task 1")
queue.pop(0)  # Removes 'Task 1'
```

Q.12. Describe the advantages of using lists in Python.

Ans. Lists in Python offer:

- **Dynamic Size**: Lists can automatically grow or shrink in size when you add or remove items.
- **Easy to Modify**: You can easily add, remove, or change items in a list.
- **Ordered**: The items in a list stay in the same order as you add them.
- **Can Hold Different Types**: A list can store different types of data like numbers, text, or even other lists.

Q.13. What is the purpose of the pop() method in Python lists?

Ans. The **pop()** method removes and returns an item from the list at a specified index. If no index is provided, it removes and returns the last item from the list.

Example:

Python Code

```
fruits = ["apple", "banana", "cherry"]
removed_item = fruits.pop(1)
print(fruits)
# Output :
['apple', 'cherry']
print(removed_item)
# Output :
'banana'
```

Q.14. Explain how you can add an item to a list in Python.

Ans. Items can be added to a list using:

append(): Adds an item to the end of the list.

insert(): Adds an item at a specified index.

Example:

Python Code:

```
fruits = ["apple"]
fruits.append("banana")
fruits.insert(2, "Mango")
print(fruits)
# Output :
['apple', 'banana', 'Mango']
```

Q.15. Differentiate between insert() and append() function.

Ans.

append()	insert()
Adds an item at the **end** of the list.	Adds an item at a **specific index** in the list.
Syntax: list.append(item)	**Syntax:** list.insert(index, item)
Easy to use when position does not matter.	Useful when you need control over where the item goes.

Does not shift other items.	Shifts other items to the right to make space for the new one.
Commonly used for adding new data at the end of a list.	Commonly used when inserting in between existing items.
Example: *fruits* = ["apple"] fruits.append("banana") print(fruits) **# Output :** ['apple', 'banana']	**Example:** fruits = ["apple"] fruits.insert(1, "Banana") print(fruits) **# Output :** ['apple', 'banana']

Q.16. What is a stack? Explain with an example.

Ans. A **stack** is a linear data structure where elements are inserted and removed from only one end, called the **top**. It follows the **Last-In, First-Out (LIFO)** rule.

Example: A **stack of books** is a real-life example. The last book placed on the top is the first one to be taken off.

Q.17. Explain the LIFO principle with a real-life example.

Ans. **LIFO** stands for **Last-In, First-Out**, which means the last item added to the stack is the first to be removed.

Example: In a **stack of plates**, the plate placed last on the top is the first one to be removed when needed.

Q.18. What are the two basic operations of a stack?

Ans. The two basic operations of a stack are:

Push: Adds an item to the top of the stack.

Pop: Removes the item from the top of the stack.

Both operations follow the **LIFO** principle.

Q.19. Write any two uses of stack in computer science.

Ans. Two common uses of stack in computer science are:

Undo operation in text editors.

Expression evaluation in programming languages.

Stacks help in managing data in reverse order as needed.

Q.20. Write a short Python code to demonstrate push and pop operations on a stack.

Ans. **Python Code:**

```
stack = [ ]
stack.append('Item 1')  # Push
stack.append('Item 2')  # Push
stack.pop()             # Pop
```

This code shows how to **add** items using append() and **remove** the top item using pop().

Q.21. What is the difference between push and pop operations in a stack?

Ans.

Push	Pop
Adds an element to the top of the stack	Removes the top element from the stack
Increases the number of items in the stack	Decreases the number of items in the stack
Example: stack = [ ] stack.append('Book A') Stack: ['Book A']	**Example:** stack.pop() → Removes 'Book A' → Stack: [ ]

Q.22. What is a queue? Explain its working with an example.

Ans. A queue is a linear data structure that follows the **FIFO (First-In, First-Out)** rule. The first element added is the first to be removed.

Example: Like a line at a bank — the person who joins the line first is served first.
In programming, items are **added at the rear (enqueue)** and **removed from the front (dequeue)**.

Q.23. Explain Enqueue and Dequeue operations with examples.

Ans. **Enqueue**: Adding an item at the **end** of the queue.
Example: If "Ali" and "Zara" are added, Zara goes behind Ali.
Dequeue: Removing an item from the **front** of the queue.
Example: If Ali is at the front, he will be removed first

Q.24. Write a simple Python program to implement a queue using built-in module.

Ans. **Python Code:**

```
from queue import Queue
q = Queue()
q.put("Ali")    # Enqueue
q.put("Zara")
print("Front:", q.queue[0])    # Peek
print("Removed:", q.get())    # Dequeue
q.put("Hassan")          # Enqueue
print("Queue:", list(q.queue))  # Updated Queue
```

This program uses Python's queue module to add, view, and remove elements from a queue.

Q.25. What is a tree? How is it different from a linear data structure?

Ans. A tree is a **non-linear data structure** where data is organized in a **hierarchical form**. It starts with a **root node**, which branches into child nodes.
Unlike lists or arrays, which are linear, trees can represent complex relationships like **family trees** or **organizational charts**.

Q.26. Describe any three properties of a tree with examples.

Ans. **Root Node**: Starting point (like the main folder in a computer).
Leaf Node: Node with no children (like a file with no subfolders).
Height: Longest path from root to any leaf node (shows tree depth).

Q.27. Write any four applications of trees in computer science.

Ans. **File System Backup** – Uses Pre-order traversal.
File Deletion – Uses Post-order traversal.
Hierarchical Representation – For family trees and company charts.
Decision Making – Used in AI and machine learning.

Q.28. What is a graph? How is it useful in computer science?

Ans. A graph is a **collection of vertices (nodes)** connected by **edges (links)**.
It represents complex relationships like **city roads, social networks**, or **computer networks**.
Graphs help model systems where multiple connections exist.

Q.29. Define any three properties of a graph with examples.

Ans. **Degree**: Number of edges connected to a vertex.
Example: A city connected to 3 cities has degree 3.
Weight: Value on an edge (like distance or cost).
Direction: One-way or two-way connection (like one-way or two-way roads).

Q.30. What is the difference between Directed and Undirected graphs? Give examples.

Ans.

Directed Graph	Undirected Graph
Edges have a **specific direction** (like one-way streets).	Edges have **no direction** (like two-way roads).
You can **only move** from one node to another in the **given direction**.	You can move **in both directions** between connected nodes.

Arrow (→) is used to show direction between nodes.	A simple line (—) connects nodes without arrows.
Example: City A → City B (you can go from A to B but not B to A directly).	**Example:** Person A — Person B (both are friends; connection goes both ways).
Used in traffic systems, web page linking, or task scheduling.	Used in social networks, road maps, and electricity grids.

Q.31. What is a Weighted Graph? Explain with example.

Ans. A weighted graph has values (called **weights**) on edges that represent **distance**, **cost**, or **time**.
Example: A road between two cities is 50 km — the edge has weight 50.
It helps find **shortest or cheapest paths**.

Q.32. Differentiate between Stack and Queue.

Ans.

Stack	Queue
A **stack** is a linear data structure that follows **LIFO** (Last In, First Out) order. The last element added is the first one to be removed.	A **queue** is a linear data structure that follows **FIFO** (First In, First Out) order. The first element added is the first one to be removed.
Elements are added and removed from the **top** of the stack.	Elements are added at the **back** and removed from the **front.**
Operations are **push** (adding an element) and **pop** (removing an element).	Operations are **enqueue** (adding an element) and **dequeue** (removing an element).
Example: Plates stacked one on top of another (last plate added is the first to be taken out).	**Example:** People standing in a line at a ticket counter (first person to enter is the first to be served).
Used in situations like **function calls, undo operations**, and **balancing symbols.**	Used in scenarios like **waiting lines, printing tasks**, and **queue management systems.**

Important MCQs

Lists

1. What is a list in Python?
A) A data structure that stores multiple values in a specific order
B) A function to print data
C) A mathematical operation
D) A built-in module

2. Which symbol is used to create a list in Python?
A) () B) {}
C) [] D) <>

3. What is the index of the first element in a list?
A) 1 B) -1
C) 0 D) Any

4. Which function is used to insert an item at a specific position in a list?
A) insert() B) append()
C) pop() D) remove()

5. What does the append() method do in a list?
A) Removes an item
B) Adds an item at a specific index
C) Adds an item at the end
D) Sorts the list

6. Which method is used to remove an item by value from a list?
A) pop() B) delete()
C) remove() D) erase()

7. How do you remove the item at index 2 in a list?
A) list.remove(2)
B) list.pop(2)
C) delete(2)
D) list.erase(2)

8. How do you check if an item exists in a list?
A) if exists in list B) find(item)
C) item in list D) check(item)

9. What type of collection is a list?
A) Unordered B) Static
C) Ordered D) Immutable

10. What is the output of len([1,2,3])?
A) 0 B) 2
C) 3 D) Error

Stacks

11. What is the key principle of a stack?
A) FIFO B) LILO
C) LIFO D) FILO

12. **Which method adds an item to the stack?**
A) push() B) append()
C) add() D) insert()

13. **Which method removes the top item from the stack?**
A) delete() B) remove()
C) pop() D) clear()

14. **Which end is used for insertion and deletion in a stack?**
A) Front B) End
C) Top D) Bottom

15. **What type of list is used to create a stack in Python?**
A) Dictionary B) String
C) List D) Set

Queues

16. **What does FIFO stand for?**
A) First In First Off B) First In First Out
C) Fast In Fast Out D) First In Final Out

17. **Which Python module is used to implement queues?**
A) list B) queue
C) stack D) collections

18. **What does enqueue mean?**
A) Remove an item B) Add item at front
C) Add item at end D) Delete queue

19. **What is dequeue operation in queues?**
A) Add item at start B) Add item at end
C) Remove from front D) Remove from end

20. **Which method is used to add an item to a queue?**
A) append() B) insert()
C) put() D) push()

Trees

21. **What is the top node in a tree called?**
A) Leaf B) Branch
C) Root D) Parent

22. **What are the connections between nodes called in a tree?**
A) Nodes B) Links
C) Edges D) Paths

23. **What is a node with no children called?**
A) Root B) Branch
C) Leaf D) Parent

24. **What does the height of a tree represent?**
A) Number of leaves B) Depth of tree
C) Path to root D) Number of nodes

25. **In which tree is the height difference of subtrees minimal?**
A) Binary Tree B) Unbalanced Tree
C) Balanced Tree D) Search Tree

Graphs

26. **What is a graph made up of?**
A) Loops and lines B) Edges and trees
C) Vertices and edges D) Points and shapes

27. **What are vertices also called?**
A) Links B) Lines
C) Nodes D) Edges

28. **What are directed graphs also called?**
A) Bidirectional graphs B) Flow graphs
C) Digraphs D) Cycles

29. **In a directed graph, what do edges have?**
A) Shape B) Color
C) Direction D) Name

30. **Which graph allows two-way travel between vertices?**
A) Directed B) Undirected
C) Weighted D) Labeled

31. **What does a weighted graph have on each edge?**
A) Weight B) Direction
C) Node D) Arrow

32. **What is a graph used for in navigation systems?**
A) Style B) Coloring
C) Route finding D) Tree conversion

33. **What does the weight on an edge represent?**
A) Time or cost B) Vertex ID
C) Edge number D) Edge color

34. **What graph shows relationships where order doesn't matter?**
A) Directed B) Weighted
C) Undirected D) Complex

35. **In an undirected graph, edges are:**
A) One way B) Two way
C) Weighted D) Directed

36. **The function used to add an item at the end of a list in Python:**
A) insert( ) B) append( )
C) remove( ) D) pop( )

37. **The purpose of the in keyword used with a Python list:**
A) Adds an item to the list
B) Removes an item from the list
C) Checks if an item exists in the list
D) Returns the length of the list

38. **An operation that removes an item from the top of the stack:**
A) Push B) Pop
C) Peek D) Add

39. **The use of a stack in converting infix to postfix expressions:**
A) Order of operands B) Priority of operators
C) Parentheses D) Both b and c

40. **The operation used to add an item to a queue:**
A) Dequeue B) Peek
C) Enqueue D) Remove

41. **Queue operation used in Breadth-First Search (BFS) for visiting nodes level by level:**
A) Adding nodes to the end of a stack
B) Removing nodes from the end of a list
C) Enqueueing nodes to a queue
D) Dequeueing nodes from a stack

42. **Tree traversal that visits the root node before its children:**
A) In-order B) Pre-order
C) Post-order D) Level-order

43. **True statement about the height of a tree:**
A) Number of edges from the root to the deepest node
B) Number of nodes from the root to the deepest node
C) Number of children of the root node
D) Always equal to the number of nodes in the tree

44. **Graph traversal that explores all immediate neighbors before moving deeper:**
A) Depth-First Search (DFS)
B) Breadth-First Search (BFS)
C) Depth-Limited Search (DLS)
D) Iterative Deepening Search (IDS)

45. **A scenario where a graph data structure is most suitable:**
A) Managing a to-do list
B) Modeling a line of customers in a store
C) Representing connections in a social network
D) All of the above

ANSWER KEY

1.	A	**2.**	C	**3.**	C	**4.**	A	**5.**	C	**6.**	C	**7.**	B	**8.**	C	**9.**	C	**10.**	C
11.	C	**12.**	B	**13.**	C	**14.**	C	**15.**	C	**16.**	B	**17.**	B	**18.**	C	**19.**	C	**20.**	C
21.	C	**22.**	C	**23.**	C	**24.**	B	**25.**	C	**26.**	C	**27.**	C	**28.**	C	**29.**	C	**30.**	B
31.	A	**32.**	C	**33.**	A	**34.**	C	**35.**	B	**36.**	B	**37.**	C	**38.**	B	**39.**	D	**40.**	B
41.	C	**42.**	B	**43.**	A	**44.**	B	**45.**	C										