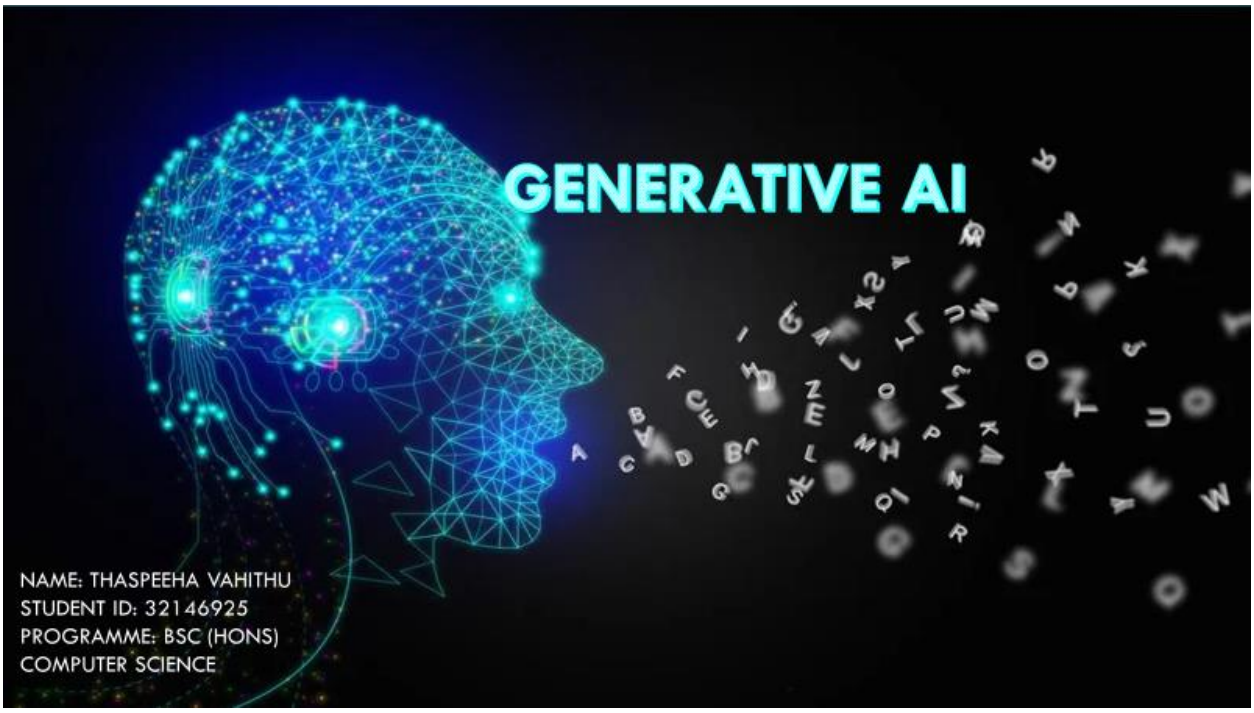
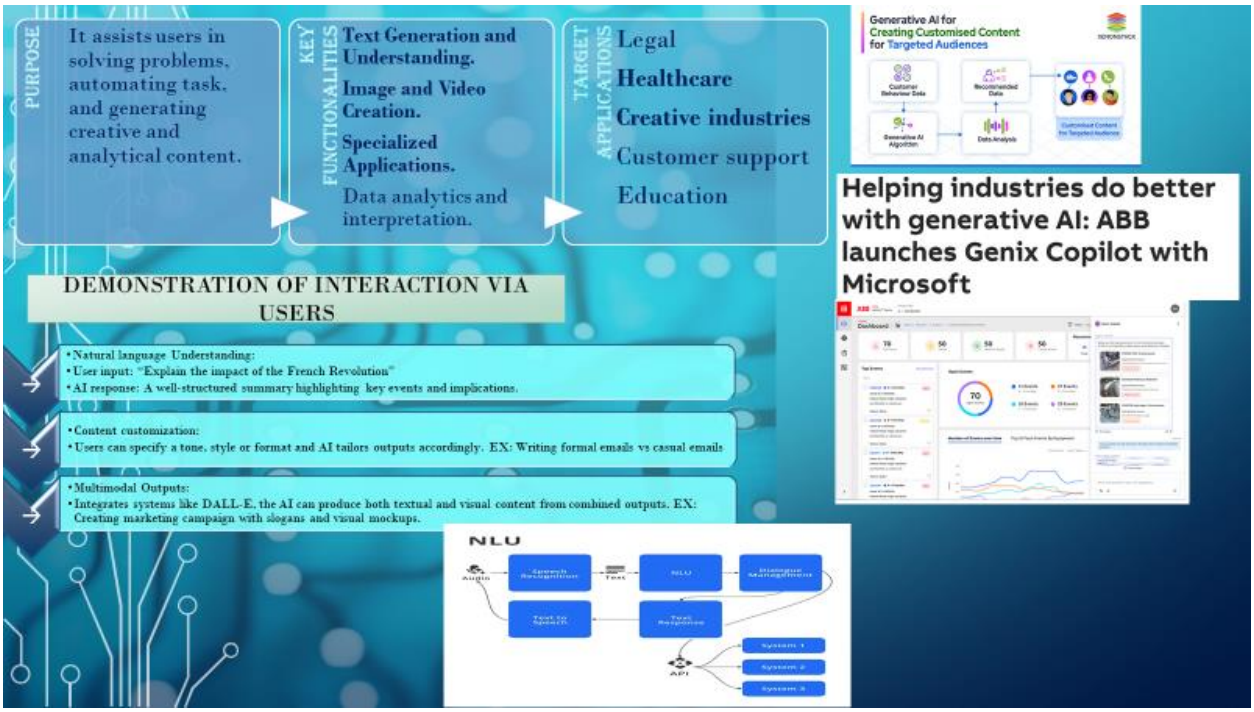
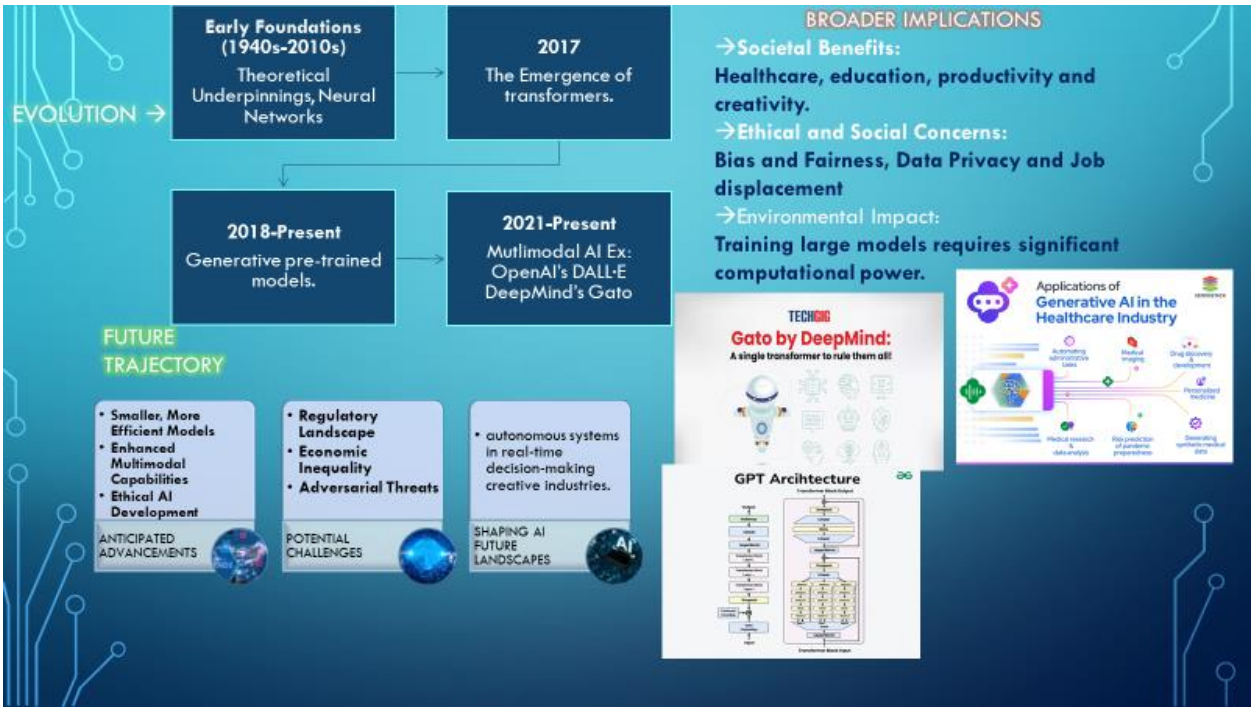
ARTIFICIAL ASSIGNMENT

Q1.







Video Presentation:

https://youtu.be/jUT7mxVGF6M

References:

CHATGPT

https://chatgpt.com/c/675b0ed8-24d8-8013-91ce-723b3e68035b

XENOSTACK

https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.xenonstack.com%2Fblog%2Fmarketing-campaigns-with-generative-ai&psig=AOvVaw0z1UnJrliXUGxCyTQUMeOZ&ust=1734108575275000&source=images&cd=vfe&opi=89978449&ved=0CAMQjB1qFwoTCOjaj4DYoooDFQAAAAAdAAAAABAE

FORBES

https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.forbes.com%2Fsites%2Fshashankagarwal%2F2024%2F05%2F25%2Fgenerative-ai-the-new-lifeline-to-overwhelmed-healthcare-systems%2F&psig=AOvVaw3p-gr2hG8n3nsGyIG-

1iKu&ust=1734108783535000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCNCC9-XYoooDFQAAAAAdAAAAABAZ

TechGig

https://www.google.com/url?sa=i&url=https%3A%2F%2Fcontent.techgig.com%2Ftechnology%2Fgato-by-deepmind-a-single-transformer-to-rule-them-all%2Farticleshow%2F91785212.cms&psig=AOvVaw3mf2g154OnSE1HvUOBCUQ1&ust=1734110755297000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCLi9iJngoooDFQAAAAAdAAAAABAE

ABB

https://new.abb.com/news/detail/122021/helping-industries-do-better-with-generative-ai-abb-launches-genix-copilot-with-microsoft

GeeksForGeeks

https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.geeksforgeeks.org%2Fintroduction-to-generative-pre-trained-transformer-gpt%2F&psig=AOvVaw2JW2sNFgDxKfvSh5WtrIAb&ust=1734119773921000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCLCVtNyBo4oDFQAAAAAdAAAAABAE

BotPenguin

https://botpenguin.com/glossary/natural-language-understanding

The Official Microsoft Blog

https://botpenguin.com/glossary/natural-language-understanding

MENLO Ventures

https://menlovc.com/2024-the-state-of-generative-ai-in-the-enterprise/

IBM

https://www.ibm.com/think/insights/artificial-intelligence-trends

2(a)PEAS Framework Analysis for a Smart Home Assistant

Performance:

A smart home assistant achieves success when its goals are met. To achieve that achievement, quantifiable performance criteria are used, such as:

1. **Task Completion Rate:** The proportion of user commands that are completed in a predetermined amount of time. For example, if a user asks, "Turn off the living room lights," the assistant should answer precisely and rapidly (preferably within two or three seconds).

2. **User satisfaction:** It is assessed via surveys or feedback ratings. Users consistently rank the helper 4.5/5 or above for ease of use, accuracy, and helpfulness, it signals success.

Environment:

Key aspects of the home setting include:

1. **Noise Levels**: Homes can range from quiet to noisy environments (e.g., kids playing, TV on). The assistant should differentiate commands from background sounds.
2. **Varied Room Environments**: Homes consist of rooms with specific functions, such as a living room(requires to assist turning on TV, or air conditioners) and a bedroom (managing lights, and alarms).
3. **User Diversity**: Accommodating multiple users with different preferences, voices, and routines. For example, one user might prefer lights dimmed to 50%, while another might prefer them at 75%.
4. **Connectivity**: Homes may have inconsistent internet or device connections. The assistant should work offline for basic commands when the internet is unavailable.

Real-world example: A user in a bedroom with a balcony asks to manage lights or alarms. The assistant must accurately hear the request despite any noises coming from outside.

Actuators:

Specific actions the assistant performs include:

1. **Smart Device Control**: Turning lights on/off, adjusting thermostats, and operating smart appliances like smart refrigerators for example its sensors can detect changes in temperature, humidity, and even the amount of

food inside. You can then access this information remotely via a smartphone app to monitor your fridge's contents when you're away from home.

2. **Communication**: Responding to user queries (e.g., weather forecasts) or sending notifications. For example, the assistant might announce, "Dinner will be ready in 10 minutes," or “You have an important business meeting in 2 hours”, across all connected devices.
3. **Multimedia Playback**: Playing music, podcasts, or videos on demand.
4. **Physical Actuators**: If connected to robotic components, such as a robotic vacuum or an object-picking robot, it could issue start or stop commands and, pick and drop respectively.

Sensors:

Types of input required:

1. **Audio Sensors (Microphones)**: To capture user commands in varying conditions. Challenges include filtering background noise and recognizing accents with different languages.
2. **Visual Sensors (Cameras, if privacy settings allow)**: For identifying users, monitoring home security, or recognizing gestures (e.g., waving or clapping to turn off a light or air conditioner). Privacy concerns might limit the use of cameras.
3. **Environmental Sensors**: They include temperature, light, and motion sensors. For example, a motion sensor might detect when someone enters a room and turn on the lights automatically.
4. **Connectivity Sensors**: Monitoring network and device status to ensure seamless operation.

Challenges include:

- Differentiating between overlapping voices in a multi-user household.
- Privacy concerns with always-on microphones or cameras.
- Sensor degradation due to dust, wear, or environmental factors.

Conclusion:

Each PEAS component contributes to the smart home assistant's functionality by ensuring it operates reliably, adapts to diverse environments, and responds effectively to user needs. The integration of these elements creates a system that enhances daily life while balancing usability and privacy concerns.

2(b)Evaluation of the Smart Assistant's Environment

1. Fully versus Partially Observable:

The environment is **partially observable** because the assistant does not have complete knowledge or does not give full access to the "complete" state of the home environment at all times.

- **Example:** The assistant may not know if a door is open unless it is equipped with door sensors. Similarly, it may not "see" scattered toys in the bedroom without a camera or environmental sensors.
- **Impact:** This lack of information could lead to confused actions. For instance, if a user says, "Turn off the TV," the assistant might not know which room they are referring to if it does not have location awareness.
- The assistant must rely on user input, historical or added data, or additional sensors to fill gaps in its knowledge.

---

2. Deterministic versus Stochastic:

The environment is **stochastic** because it involves uncertainty and unpredictable events.

- **Example:** The assistant may attempt to turn on a smart light, but it fails due to a power outage or a disconnected Wi-Fi network. Additionally, unpredictable user behavior, like giving conflicting commands ("AC on" immediately followed by "AC off"), adds to the uncertainty.
- **Impact:** The assistant must be robust enough to handle such scenarios, either by providing meaningful feedback (e.g., "The light is not responding") or offering alternative solutions (e.g., "Would you like me to turn on the AC after an hour?").

---

3. Episodic versus Sequential:

The environment is **sequential** because the assistant's actions and interactions depend on past events.

- **Example:** If a user sets a reminder for "go grocery shopping at 7 am," the assistant must remember this event to notify the user at the right time. Similarly, if the user says, "Turn on the lights as usual," the assistant must recall the user's preferred light settings from previous interactions.
- **Impact:** Sequential environments require the assistant to maintain a history of interactions, preferences, and context to make informed decisions. This increases complexity but enhances the user experience.

---

4. Static versus Dynamic:
The environment is **dynamic**, changing over time independently of the assistant's actions.

- **Example:** People move between rooms, devices connect or disconnect, and external factors (e.g., weather changes) impact user needs. For instance, the assistant might need to adjust the thermostat as outdoor temperatures drop.
- **Impact:** A dynamic environment requires the assistant to adapt in real time. It must continuously monitor changes (e.g., detecting motion to adjust lighting) and update its state to remain effective.

---

5. Discrete versus Continuous:
The environment involves both **discrete** and **continuous** elements, but the interaction leans towards **continuous**.

- **Example:** Discrete: Commands like "Turn off the kitchen light" are distinct events. Continuous: Background noise or temperature changes provide ongoing data that the assistant might monitor to make decisions (e.g., maintaining room comfort levels).
- **Impact:** Continuous data processing challenges the assistant to filter and prioritize information while managing discrete tasks. For instance, it might have to process continuous temperature readings to adjust the thermostat but still respond promptly to a user command.

---

6. Single versus Multi-Agent:
The environment is **multi-agent**, involving interactions with multiple agents (human and non-human).

- **Example:** In a family home, multiple users give commands, sometimes simultaneously. Additionally, the assistant interacts with other AI-powered systems like a robotic vacuum or a smart doorbell. For instance, the assistant might coordinate with the vacuum to avoid scheduling cleaning during family dinner time.
- **Impact:** The assistant must manage conflicting or overlapping inputs, prioritize tasks, and maintain harmony among various agents. For example, if two users give contradictory commands ("Set the thermostat to 60°F" and "Set it to 65°F"), the assistant might need to clarify or choose based on user preferences.

Conclusion:

The smart home assistant operates in all of the above environments. These characteristics highlight the complexity of its operation and the need for advanced sensors, adaptability, memory, and conflict resolution capabilities to ensure optimal performance in a real-world household setting.

2(c)Proposed Agent Design: Learning Agent

Justification:

A **learning agent** is the most suitable design for a smart home assistant because it can learn from past experiences, or have learning capabilities. They start to act with basic knowledge and adapt to dynamic, multi-agent, and partially observable environments, as well as evolve over time to meet user-specific needs. Here's why:

---

1. Adaptation or Learning from Experience:

- **Reason:** Each household has unique habits, routines, and preferences. A learning agent can observe and adapt its behavior based on user interactions.
- **Example:** If a user frequently requests the thermostat to be set at 72°F during the evenings, the assistant can learn this pattern and automate the adjustment without explicit commands.

---

2. Partially Observable Environment:

- **Reason:** A learning agent can infer missing information based on historical data and probabilistic reasoning.
- **Example:** If the assistant notices the lights are frequently turned off after 10 PM in the living room, it can suggest automating this action.

---

3. Dynamic and Stochastic Environments:

- **Reason:** Dynamic environments require the ability to adapt to changing conditions. A learning agent can refine its understanding of the environment and improve its decision-making over time.
- **Example:** If the internet goes down, the assistant could learn to execute offline commands more efficiently while notifying the user of the limitation.

4.Improvement with User Feedback:

- **Reason:** A learning agent can evaluate its actions based on explicit feedback or implicit signals (e.g., task completion rates).
- **Example:** If the assistant's initial response to "Play relaxing music" results in a user changing the music, it can learn what "relaxing music" means for that user and refine future recommendations.

---

5. Other Agent Designs:
A learning agent encompasses aspects of other agent types, enhancing its flexibility:

- **Reflex actions:** Quick responses for simple tasks (e.g., "Turn on the lights").
- **Model-based reasoning:** Using an internal model of the environment to track states (e.g., knowing which devices are on).
- **Goal-based:** Prioritizing tasks to meet explicit user goals (e.g., setting reminders for events).
- **Utility-based decision-making:** Optimizing for user satisfaction and efficiency (e.g., adjusting temperature settings to balance comfort and energy savings).

Example Use Case:

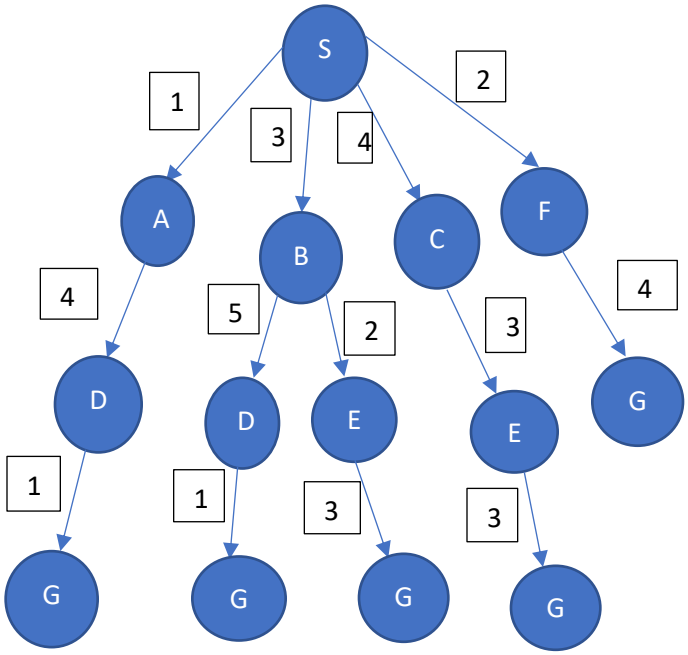
A family installs the assistant, and over time, it learns:

1. **Morning Routines:** By observing repeated actions (turning on the coffee maker at 7 AM and playing news), it automates these tasks.
2. **Personal Preferences:** It identifies that one user prefers jazz music while another prefers pop, customizing playback based on who's in the room.
3. **Proactive Behavior:** Noticing the family often forgets to lock the front door, it suggests setting up an automatic lock schedule.

Conclusion:

A **learning agent** design offers the flexibility, adaptability, and personalization required to manage the complexities of a smart home environment effectively. This design aligns with user expectations for a proactive and intuitive smart assistant that improves over time.

Q3.Binary Search Tree



3(a) Depth-First Search (DFS) explores a graph by expanding the deepest unvisited node first, backtracking only when no further progress can be made.

STACK	PUSH	CHILD	POP	PATH
S	S		S	S
A B C F	A B C F	ABCF	A	S,A
D B C F	D	D	D	S,A,D
G B C F	G	G	G	S,A,D,G

Steps:

1. **Order of Expansion**: DFS explores first the initial state which is stacked and popped and the children of the initial state are stacked using the logic Last In First Out (LIFO). DFS follows a stack-based approach:
    - Start from S.
    - Explore A (first child of S), then move to D (child of A), and finally reach G (child of D).

    Order of states expanded: S → A → D → G.

2. **Path Returned**: S → A → D → G.

3(b) Breadth-First Search (BFS) explores all neighbors at the current depth level before moving to the next depth level. It is guaranteed to find the shortest path (in terms of the number of edges) to the goal state in an unweighted graph.

QUEUE	ENQUEUE	CHILD	DEQUEUE	PATH
S	S		S	S
A B C F	A B C F	ABCF	A	S,A
B C F D	D	D	B	S,A,B
C F D D E	D E	DE	C	S,A,B,C
F D D E E	E	E	F	S,A,B,C,F
D D E E G	G	G	D	S,A,B,C,F,D
D E E G G	G	G	D	S,A,B,C,F,D,D
E E G G G	G	G	E	S,A,B,C,F,D,D,E
E G G G G	G	G	E	S,A,B,C,D,D,E,E
G G G G G	G	G	G	S,A,B,C,D,D,E,E,G

1. Order of Expansion:

BFS explores from the initial state S which is enqueued and dequeued and the initial state's children are enqueued using the logic FIFO. BFS follows a **queue-based approach**:

- Start at S.
- Expand all immediate neighbors of S (A, B, C).

- Expand the neighbors of A, B, and C (D, E, F).
- Expand the neighbors of D, E, and F (G).

Order of states expanded:
S → A → B → C → F → D → E → G.

---

2. Path Returned:

- From S to G via A and D.

Path returned: S→A→B→C→D→D→E→E→G

Shortest Path: S → A → D → G.

3(c) Iterative Deepening Search (IDS) is a combination of **depth-first search (DFS)** and **breadth-first search (BFS)**. It repeatedly performs DFS with increasing depth limits until the goal state is found. At each depth limit, it explores all nodes up to that depth.

IDS follows stack-based approach:

RECURSION 1	PUSH	CHILD	POP	PATH
STACK				
S	S		S	S
RECURSION 2				
S	S		S	S
A B C F	A B C F	ABCF	A	S,A
B C F	—	No Child Backtrack	B	S,A,B
C F	—	No Child Backtrack	C	S,A,B,C
F	—	No Child Backtrack	F	S,A,B,C,F
RECURSION 3				
S	S		S	S

A B C F	A B C F	ABCF	A	S,A
D B C F	D	D	D	S,A,D
B C F	—	No Child Backtrack	B	S,A,D,B
D E C F	D E	DE	D	S,A,D,B,D
E C F	—	No Child Backtrack	E	S,A,D,B,D,E
C F	—	No Child Backtrack	C	S,A,D,B,D,E,C
E F	E	E	E	S,A,D,B,D,E,C,E
F	—	No Child Backtrack	F	S,A,D,B,D,E,C,E,F

G	G	G	G	S,A,D,B,D,E,C,E,F,G

1. Depth 0: (RECURSION 1)

Here in this case the limit is the initial state is S itself, and it stacked and popped which is the first recursion.

Nodes Expansion: S

Depth 1: (RECURSION 2)

Here in this case the limit is till the children A, B, C, and F of the initial state is stacked and use the LIFO logic to a second recursion.

Nodes Expansion: S→A→B→C→F

Depth 2: (RECURSION 3)

Here in this case the limit is till the children D,(D, E), E, G of states A, B, C, F respectively.

Nodes Expansion: S→A→D→B→E→C→F→G

Final Order of expansion: S→A→D→B→E→C→F→G

2.Path Returned: S→A→D→B→D→E→C→E→F→G

3(d)**Uniform cost search (UCS)**: is a method that uses the lowest cumulative cost to find a path from the source to the destination. **It**'s a search algorithm that expands nodes based on the **cumulative cost** from the initial state (S). It uses a **priority queue** where nodes with the smallest path cost are dequeued first. This ensures that UCS always explores the least costly path to reach the goal.

EXPANDED NOTES	CHILD	PRIORITY QUEUE
—	—	$\{S_0\}$
S_0	A_1, F_2, B_3, C_4	$\{A_1, F_2, B_3, C_4\}$
A_1	D_5	$\{F_2, B_3, C_4, D_5\}$
F_2	G_6	$\{B_3, C_4, D_5, G_6\}$
B_3	E_6, D_8	$\{C_4, D_5, E_5, G_6, D_8\}$
C_4	E_7	$\{D_5, E_5, G_6, E_7, D_8\}$
D_5	G_7	$\{E_5, G_6, G_7, D_8\}$
E_5	G_8	$\{G_6, G_7, D_8, G_8\}$
G_6	NO CHILD	$\{G_7, D_8, G_8\}$
G_7	NO CHILD	$\{D_8, G_8\}$
D_8	G_9	$\{G_8, G_9\}$
G_8	NO CHILD	$\{G_9\}$

1. Order in Which States Are Expanded

Step 1 (Start at S):

- The priority queue initially contains only the start node: **Queue: [(S, 0)]** (cost is 0).
- Expand S, enqueue its neighbors with their respective costs:
  - (A, 1), (F,2),(B, 3), (C, 4)

Order of Expansion: S
Queue after Step 1: [(A, 1),(F,2), (B, 3), (C, 4)]

Step 2 (Expand A):

- Dequeue (A, 1) (smallest cost).
- Enqueue its neighbor with an updated cumulative cost:

- (D, 5) (cost from S → A → D).

Order of Expansion: S → A
Queue after Step 2: [(F, 2), (B, 3),(C,4), (D, 5)]

Step 3 (Expand F):

- Dequeue (F, 2) (smallest cost).
- Enqueue its neighbor with updated cumulative cost:
  - (G, 6) (cost from S → F → G).

Order of Expansion: S → A → B
Queue after Step 3: [(B, 3), (C, 4), (D, 5),(G,6)]

Step 4 (Expand B):

- Dequeue (B, 3) (smallest cost).
- Enqueue its neighbor with an updated cumulative cost:
  - (D,8),(E,5) (cost from S → B → D, S->B->E respectively)

Order of Expansion: S → A → F → B
Queue after Step 4: [(C, 4), (D, 5), (E, 6), (G,6), (D,8)]

Step 5 (Expand C):

- Dequeue (C, 4) (smallest cost).
- Enqueue its neighbor with an updated cumulative cost:
  - (E, 7) (cost from S → C → E).

Order of Expansion: S → A → F → B → C
Queue after Step 5: [(D, 5), (E, 5), (G, 6), (E,7), (D,8)]

Step 6 (Expand D):

- Dequeue (D, 5) (smallest cost).
- Enqueue its neighbor with an updated cumulative cost:
  - (G, 7) (cost from S → A → D → G).

Order of Expansion: S → A → F → B → C → D
Queue after Step 6: [(E, 5), (G, 6),(G,7), (D, 8)]

Step 7 (Expand E):

- Dequeue (E, 5) (smallest cost).
- Enqueue its neighbor with an updated cumulative cost:
- (G,8) (cost from S→B->E→G)

Order of Expansion: S → A → B → C → D → E → F
Queue after Step 7: [(G, 6), (G, 7),(D,8),(G,8)]

Step 8 (Expand G):

- Dequeue (G, 6) (smallest cost). Goal state reached. UCS terminates.

Order of Expansion: S → A → B → C → D → E → F → G

Queue after step 8: $[(G, 7), (D, 8), (G, 8)]$

---

2. Path Returned

To reconstruct the path, trace back the cumulative costs:

- Path: **S → A → D → G**
- Cumulative Cost: **6**

Shortest Path:

1. **Order of States Expanded**: S → A → B → C → D → E → F → G
2. **Path Returned**: **S → A → D → G** with a total cost of **6**.

Q5. Algorithm

```
#Imports deque class from Python's collections
from collections import deque

#Defines a function

def bfs_shortest_track(graph, start, goal):

    # Queue for BFS, initialized with the starting node as a path

    queuelist = deque([[start]])

    # Set to track visited nodes

    visited = set()


    while queue:

        # Get the first path in the queue

        track = queue.popleft()

        # Get the last node in the current path

        node = track[-1]


        # If the node has not been visited

        if node not in visited:

            # Mark the node as visited

            visited.add(node)


            # Check if this node is the goal

            if node == goal:

                return track
```

```
        # Add neighbors of the current node to the queue
        for neighbor in graph.get(node, []):
            new_path = list(path)  # Copy the current path
            new_path.append(neighbor)  # Add the neighbor to the path
            queuelist.append(new_path)

    # Return None if no path is found
    return None
```

Q5.Implementation

```
#Name:Thaspeeha Vahithu, Student ID: 32146925
#The BFS program uses a queue to explore nodes level by level, starting from the
start node.
#It tracks visited nodes to avoid re-exploration and appends neighbors to the
queue while maintaining paths; when the goal node is reached.
#The program returns the shortest path.

#Imports deque class from Python's collections
from collections import deque

#Define a function
def bfs_shortest_track(graph, start, goal):
    # Queue for BFS
    queuelist = deque([[start]])
    # Set to keep track of visited nodes
    called = set()

    # Order of explored nodes
    explored_order = []

    while queuelist:
        # Get the first path in the queue
        track = queuelist.popleft()
        # Get the last node in the path
        node = track[-1]
```

```
        # If the node has not been visited yet
        if node not in called:
            # Add it to the visited set
            called.add(node)
            explored_order.append(node)

            # Check if we reached the goal
            if node == goal:
                return explored_order, track

            # Add neighbors of the current node to the queue
            for neighbor in graph.get(node, []):
                new_path = list(track)
                new_path.append(neighbor)
                queuelist.append(new_path)

    # If the goal is not reachable
    return explored_order, None

# Graph representation as an adjacency list means a dictionary
graph = {
    'S': ['A', 'B', 'C'],
    'A': ['D'],
    'B': ['E'],
    'C': ['F', 'S'],  # S included as a loop
    'D': ['G', 'H'],
    'E': ['I', 'J'],
    'F': [],
    'G': [],
    'H': [],
    'I': [],
    'J': []
}

# Start and goal nodes
start_node = 'S'
goal_node = 'J'

# Run BFS
explored_order, shortest_path = bfs_shortest_track(graph, start_node, goal_node)

# Print the results
print("Order of nodes explored:", explored_order)
if shortest_path:
```

```
    print("Shortest path from {} to {}: {}".format(start_node, goal_node,
shortest_path))
else:
    print("No path found from {} to {}".format(start_node, goal_node))
```

Q5.OUTPUT

Order of nodes explored: ['S', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']

Shortest path from S to J: ['S', 'B', 'E', 'J']