In [2]:

```python
# Graphs
# Breadth First Search --> Analogous to Level orders traversal
def bfs(graph,start):
    visited=set()
    queue=[start]
    visited.add(start)
    while queue:
        vertex=queue.pop(0)
        print(vertex,end=" ")
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Depth First Search --> Analogous to inorder,preorder and postorder traver
def dfs(graph,start):
    visited=set()
    stack=[start]
    while stack:
        vertex=stack.pop()
        if vertex not in visited:
            print(vertex,end=" ")
            visited.add(vertex)
            stack.extend(reversed(graph[vertex]))

# Example usage
graph={"A":["B","C"],"B":["A","D","E"],"C":["A","F"],
       "D":["B"],"E":["B","F"],"F":["C","E"]}
start_vertex="A"
print("Breadth First Traversal: ",end="")
bfs(graph,start_vertex)
print()
print("Depth First Traversal: ",end="")
dfs(graph,start_vertex)
```

```
Breadth First Traversal: A B C D E F
Depth First Traversal: A B D E F C
```

In [4]:

```python
# Best First Search
from queue import PriorityQueue

# Graph represented as an adjacency list
graph = {
    0: [(1, 1), (2, 2), (3, 3)],
    1: [(4, 4)],
    2: [(5, 5)],
    3: [(6, 6)],
    4: [(7, 3)],
    5: [(7, 2)],
    6: [(7, 1)],
    7: []
}

def best_first_search(source, target):
    visited = set()
    pq = PriorityQueue()  # Priority queue to explore nodes by lowest cost 
    pq.put((0, source))   # Start with the source node (priority, node)
    while not pq.empty():
        cost, node = pq.get()  # Get node with the lowest cost
        if node in visited:
            continue
        print(node, end=" ")   # Print the current node
        visited.add(node)
        if node == target:     # Stop if the target is reached
            break
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                pq.put((weight, neighbor))  # Add neighbors to the queue wi

# Run Best First Search
source = 0
target = 7
best_first_search(source, target)
```

```
0 1 2 3 4 7
```

In [21]:

```python
from queue import PriorityQueue

# A* Search Algorithm
def a_star(graph, heuristics, start, goal):
    pq = PriorityQueue()  # Priority queue for A* (min-heap based on f-cost
    pq.put((0, start))  # Start node with f-cost 0
    came_from = {start: None}  # Track the path (parent nodes)
    g_cost = {start: 0}  # Cost from start to the current node (g-cost)

    while not pq.empty():
        current_f_cost, current_node = pq.get()

        if current_node == goal:  # Goal reached
            path = []
            while current_node:
                path.append(current_node)
                current_node = came_from[current_node]
            return path[::-1]  # Return reversed path from start to goal

        # Explore neighbors
        for neighbor, cost in graph[current_node]:
            new_g_cost = g_cost[current_node] + cost
            if neighbor not in g_cost or new_g_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_g_cost
                f_cost = new_g_cost + heuristics[neighbor]  # f(n) = g(n) +
                pq.put((f_cost, neighbor))
                came_from[neighbor] = current_node

    return None  # No path found

# Graph (Adjacency List)
graph = {
   'A':[('B',2),('E',3)],
   'B':[('C',1),('G',9)],
   'C':None,
   'E':[('D',6)],
   'D':[('G',1)]
}

# Heuristic (h-cost) for each node (estimated cost to goal)
heuristics = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }

# Run A* search
start = 'A'
goal = 'G'
path = a_star(graph, heuristics, start, goal)
print("Path found:", path)
```

```
Path found: ['A', 'E', 'D', 'G']
```

In [25]:
```python
def calculate_cost(H, condition, weight=1):
    total_cost = 0
    # Calculate AND conditions cost
    if 'AND' in condition:
        total_cost += sum(H[node] + weight for node in condition['AND'])

    # Calculate OR conditions cost (minimum of all OR nodes)
    if 'OR' in condition:
        or_cost = min(H[node] + weight for node in condition['OR'])
        total_cost += or_cost

    return total_cost

def find_shortest_path(start, H, conditions, weight=1):
    path = start
    if start in conditions:
        condition = conditions[start]
        # Calculate the cost directly while finding the path
        cost = calculate_cost(H, condition, weight)
        H[start] = cost  # Update heuristic for the node

        # Process OR paths
        if 'OR' in condition:
            next_node = condition['OR'][0]  # Take the first OR node
            path += f' <-- {find_shortest_path(next_node,H,conditions,weigh

        # Process AND paths
        if 'AND' in condition:
            and_nodes = condition['AND']
            path += f' <-- (AND: {", ".join(and_nodes)})'
            for and_node in and_nodes:
                path += f' + {find_shortest_path(and_node,H,conditions,weig

    return path.strip()

# Heuristic values
H = {'A': -1, 'B': 4, 'C': 2, 'D': 3, 'E': 6,
     'F': 8, 'G': 2, 'H': 0, 'I': 0, 'J': 0}
# Conditions representing the graph structure (AND/OR)
conditions = {
    'A': {'OR': ['B'], 'AND': ['C', 'D']},
    'B': {'OR': ['E', 'F']},
    'C': {'OR': ['G'], 'AND': ['H', 'I']},
    'D': {'OR': ['J']}
}

# Weight for cost calculation
weight = 1

# Shortest Path Calculation
print('Shortest Path:')
print(find_shortest_path('A', H, conditions, weight))
```

```
Shortest Path:
A <-- B <-- E <-- (AND: C, D) + C <-- G <-- (AND: H, I) + H + I + D <-- J
```

In [14]:
```python
# AO* algorithm
class Graph:
    def __init__(self, graph, heuristic):
        self.graph = graph         # The graph structure (adjacency list)
        self.heuristic = heuristic  # Heuristic values
        self.solution = {}          # To track the solution of sub-problems

    def ao_star(self, node, backtrack=False):
        # If it's already solved, return the solution
        if node in self.solution:
            return self.solution[node]

        print(f"Processing Node: {node}")

        # If the node has no further children, it's a goal node
        if not self.graph.get(node):
            self.solution[node] = ([], 0)  # No children, zero cost
            return self.solution[node]

        min_cost = float('inf')
        best_child = None

        # Explore all sub-graphs (AND conditions)
        for children in self.graph[node]:
            total_cost = 0
            sub_solution = []
            for child in children:
                cost = self.heuristic[child]
                sub_solution.append(child)
                total_cost += cost

            # Check if we found a better (less costly) solution
            if total_cost < min_cost:
                min_cost = total_cost
                best_child = sub_solution

        # Store the best solution for this node
        self.solution[node] = (best_child, min_cost)

        # If backtracking is allowed, solve the child nodes recursively
        if backtrack:
            for child in best_child:
                self.ao_star(child, backtrack=True)

        return self.solution[node]

# Graph representation: Each node has a list of AND conditions (group of ch
graph = {
    'A': [['B'], ['C','D']],
    'B': [['E'], ['F']],
    'C': [['G'],['H','I']],
    'D': [['J']]
}

# Heuristic values (assumed cost to reach goal from each node)
heuristic = {'A': -1, 'B': 4, 'C': 2, 'D': 3, 'E': 6,
             'F': 8, 'G': 2, 'H': 0, 'I': 0, 'J': 0}

# Initialize the graph
g = Graph(graph, heuristic)
```

```python
# Perform AO* search starting from node 'A'
solution = g.ao_star('A', backtrack=True)
print("\nFinal Solution:", solution)
```

```
Processing Node: A
Processing Node: B
Processing Node: E

Final Solution: (['B'], 4)
```

In [16]:

```
# AO* Algorithm
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode):
        self.graph = graph
        self.H = heuristicNodeList
        self.start = startNode
        self.parent = {}
        self.status = {}
        self.solutionGraph = {}

    def applyAOStar(self):
        self.aoStar(self.start, False)

    def getNeighbors(self, v):
        return self.graph.get(v, [])

    def getStatus(self, v):
        return self.status.get(v, 0)

    def setStatus(self, v, val):
        self.status[v] = val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n, 0)

    def setHeuristicNodeValue(self, n, value):
        self.H[n] = value

    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:"
        print("------------------------------------------------------------
        print(self.solutionGraph)
        print("------------------------------------------------------------

    def computeMinimumCostChildNodes(self, v):
        minimumCost = float('inf')
        costToChildNodeListDict = {}
        for nodeInfoTuplelist in self.getNeighbors(v):
            cost = 0
            nodeList = []
            for c, weight in nodeInfoTuplelist:
                cost += self.getHeuristicNodeValue(c) + weight
                nodeList.append(c)
            if cost < minimumCost:
                minimumCost = cost
                costToChildNodeListDict[minimumCost] = nodeList
        if minimumCost == float('inf'):
            minimumCost = 0
            costToChildNodeListDict[minimumCost] = []
        return minimumCost, costToChildNodeListDict[minimumCost]

    def aoStar(self, v, backTracking):
        print("HEURISTIC VALUES :", self.H)
        print("SOLUTION GRAPH :", self.solutionGraph)
        print("PROCESSING NODE :", v)
        print("------------------------------------------------------------

        if self.getStatus(v) >= 0:
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(
            print(minimumCost, childNodeList)
            self.setHeuristicNodeValue(v, minimumCost)
```

```
            self.setStatus(v, len(childNodeList))

            solved = True
            for childNode in childNodeList:
                self.parent[childNode] = v
                if self.getStatus(childNode) != -1:
                    solved = False

            if solved:
                self.setStatus(v, -1)
                self.solutionGraph[v] = childNodeList

            if v != self.start:
                self.aoStar(self.parent[v], True)

            if not backTracking:
                for childNode in childNodeList:
                    self.setStatus(childNode, 0)
                    self.aoStar(childNode, False)

print("Graph - 1")
h1 = {'A': -1, 'B': 4, 'C': 2, 'D': 3, 'E': 6,
      'F': 8, 'G': 2, 'H': 0, 'I': 0, 'J': 0}
graph1={
    'A': [[('B',1)], [('C',1),('D',1)]],
    'B': [[('E',1)], [('F',1)]],
    'C': [[('G',1)],[('H',1),('I',1)]],
    'D': [[('J',1)]]
}
G1 = Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()
```

```
Graph - 1
HEURISTIC VALUES : {'A': -1, 'B': 4, 'C': 2, 'D': 3, 'E': 6, 'F': 8, 'G':
2, 'H': 0, 'I': 0, 'J': 0}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------
5 ['B']
HEURISTIC VALUES : {'A': 5, 'B': 4, 'C': 2, 'D': 3, 'E': 6, 'F': 8, 'G':
2, 'H': 0, 'I': 0, 'J': 0}
SOLUTION GRAPH : {}
PROCESSING NODE : B
-----------------------------------------------------------
7 ['E']
HEURISTIC VALUES : {'A': 5, 'B': 7, 'C': 2, 'D': 3, 'E': 6, 'F': 8, 'G':
2, 'H': 0, 'I': 0, 'J': 0}
SOLUTION GRAPH : {}
PROCESSING NODE : A
-----------------------------------------------------------
7 ['C', 'D']
HEURISTIC VALUES : {'A': 7, 'B': 7, 'C': 2, 'D': 3, 'E': 6, 'F': 8, 'G':
2, 'H': 0, 'I': 0, 'J': 0}
SOLUTION GRAPH : {}
PROCESSING NODE : E
-----------------------------------------------------------
0 []
HEURISTIC VALUES : {'A': 7, 'B': 7, 'C': 2, 'D': 3, 'E': 0, 'F': 8, 'G':
2, 'H': 0, 'I': 0, 'J': 0}
SOLUTION GRAPH : {'E': []}
PROCESSING NODE : B
-----------------------------------------------------------
1 ['E']
HEURISTIC VALUES : {'A': 7, 'B': 1, 'C': 2, 'D': 3, 'E': 0, 'F': 8, 'G':
2, 'H': 0, 'I': 0, 'J': 0}
SOLUTION GRAPH : {'E': [], 'B': ['E']}
PROCESSING NODE : A
-----------------------------------------------------------
2 ['B']
FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A
--------------------------------------------------------------------
{'E': [], 'B': ['E'], 'A': ['B']}
--------------------------------------------------------------------
```