A white sheet with black text

Description automatically generated

a.

A diagram of a network

Description automatically generated

b.

Descendent Vertices Of Vertex 6 :2/9

Ancestor Vertices Of Vertex 6 :5

c.

CC is 1

d.

11/16; 12/13; 14/15

Note:

Vertex= 1 pre= 1 post = 32 ; CC = 1 prev = -1

Vertex= 2 pre= 16 post = 27 ; CC = 1 prev = 6

Vertex= 3 pre= 9 post = 14 ; CC = 1 prev = 5

Vertex= 4 pre= 2 post = 7 ; CC = 1 prev = 1

Vertex= 5 pre= 8 post = 31 ; CC = 1 prev = 1

Vertex= 6 pre= 15 post = 30 ; CC = 1 prev = 5

Vertex= 7 pre= 17 post = 20 ; CC = 1 prev = 2

Vertex= 8 pre= 21 post = 26 ; CC = 1 prev = 2

Vertex= 9 pre= 28 post = 29 ; CC = 1 prev = 6

Vertex= 10 pre= 18 post = 19 ; CC = 1 prev = 7

Vertex= 11 pre= 22 post = 23 ; CC = 1 prev = 8

Vertex= 12 pre= 10 post = 11 ; CC = 1 prev = 3

Vertex= 13 pre= 12 post = 13 ; CC = 1 prev = 3

Vertex= 14 pre= 3 post = 4 ; CC = 1 prev = 4

Vertex= 15 pre= 5 post = 6 ; CC = 1 prev = 4

Vertex= 16 pre= 24 post = 25 ; CC = 1 prev = 8

A diagram of a graph

Description automatically generated

a.

Pre-order: {'A': 1, 'C': 2, 'D': 3, 'F': 4, 'G': 5, 'H': 7, 'E': 11, 'B': 15}

Post-order: {'G': 6, 'H': 8, 'F': 9, 'D': 10, 'E': 12, 'C': 13, 'A': 14, 'B': 16}

b.

Sources Of The Graph :A/B

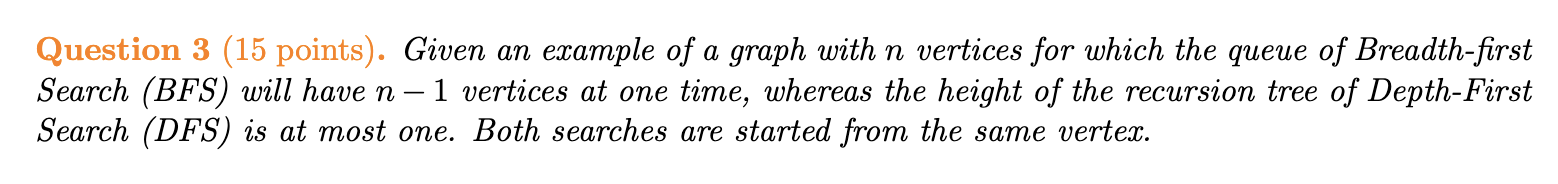
Sinks Of The Graph: G/H

c.

BACEDFHG

d.

There is 1 topological ordering in this graph.



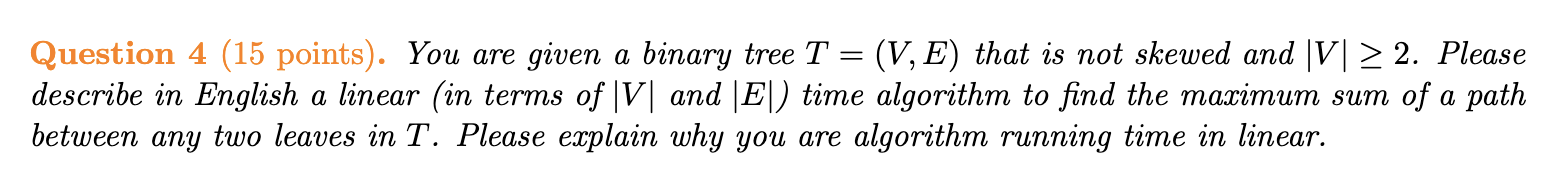
BFS explores neighbors level by level while DFS follows a path as deep as possible before backtracking.

BFS start from node A, then the queue contains: {C, D, E, F, G, H} (size = n-1).

DFS also from node A, then explore the depth that is only 1. For example: DFS will explore C after completing A.

A diagram of a network

Description automatically generated



1.

The binary tree includes root, left tree and right tree. Due to the tree that is not skewed and   
|V | ≥ 2, computing the potential max path sum through this node:

maxSum = leftSum + rightSum + node.value

2.The reason for running time in linear:

Each node is visited only once → DFS traversal takes O(n).

Each edge is processed once → Every edge is used to compute the path sum, contributing to O(n).

the total runtime is O(|V|+|E|) that is O(n), which is optimal for tree traversal.

A text on a white background

Description automatically generated

a.

**Description:**

1.Computing the degree of Vertex

* Traversal the adjacency-list. So store vertex and degrees of u’s neighbor into array that is named by “degree”.

2.Using two loop to traversal adjacency-list and sum the degrees of u’s neighbor in the “degree”.

**Time Complexity:**

Computing degrees traversals the vertex and edge: O(O(|V| + |E|)

Calculating twodegree --Each edge is visited twice → O(|E|) and traversal every vertex: O(O(|V| + |E|)

**Overall complexity: O(|V| + |E|) (linear time)**

**b.**

def c\_two\_degree(graph):  
 degree = {}  
 two\_degree = {}  
 for v, neighbors in graph.items():  
 degree[v] = len(neighbors)  
  
 for v, neighbors in graph.items():  
 sum = 0  
 for u in neighbors:  
 sum += degree[u]  
 two\_degree[v] = sum  
  
 return two\_degree

**c.**

Step 1:

degree = {u: len(neighbors) for u, neighbors in graph.items()}

Computes the degree of each vertex in O(|V| + |E|) time.

Step 2:

For each vertex u, it sums the degrees of its neighbors: O(|V|).

Since every edge is counted twice in an undirected graph, it takes O(|E|).

Total Time Complexity:

O(|V| + |E|) — linear in the size of the graph.