Graph II

Jason

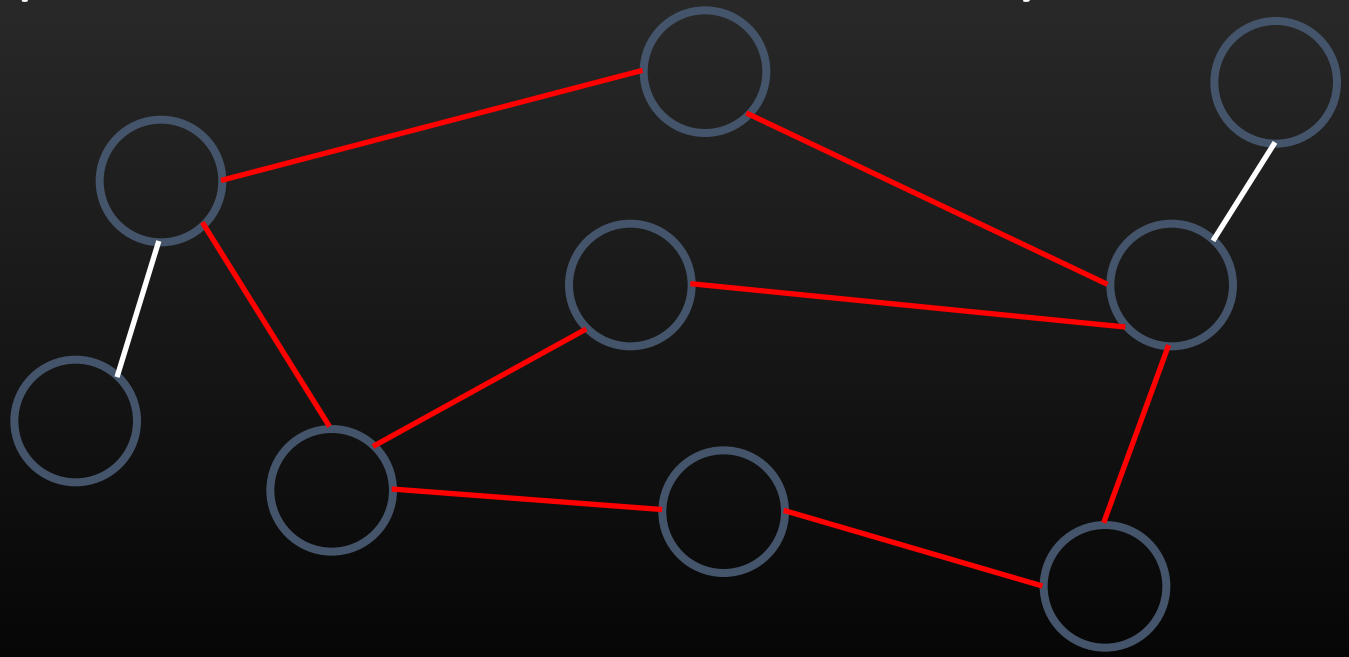
Prerequisite

- Graph I
  - Definition of graphs
  - Graph representations
  - DFS and BFS

Table of Contents

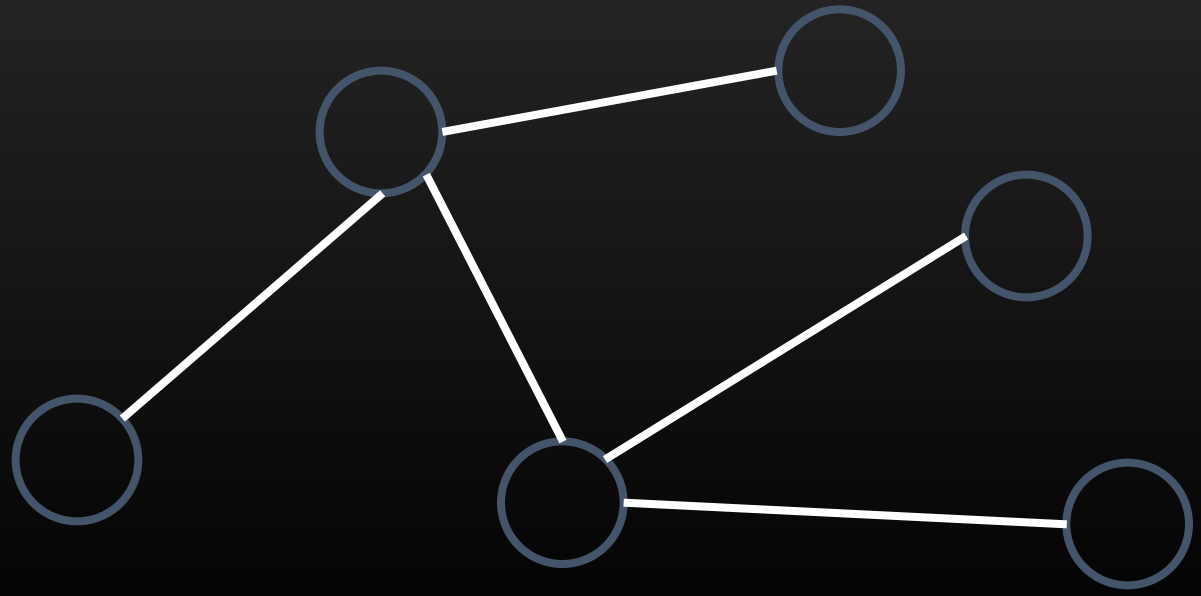
Cycles

- Cause trouble in graphs
- Make problems difficult
- If there are no cycles, problems can be solved efficiently



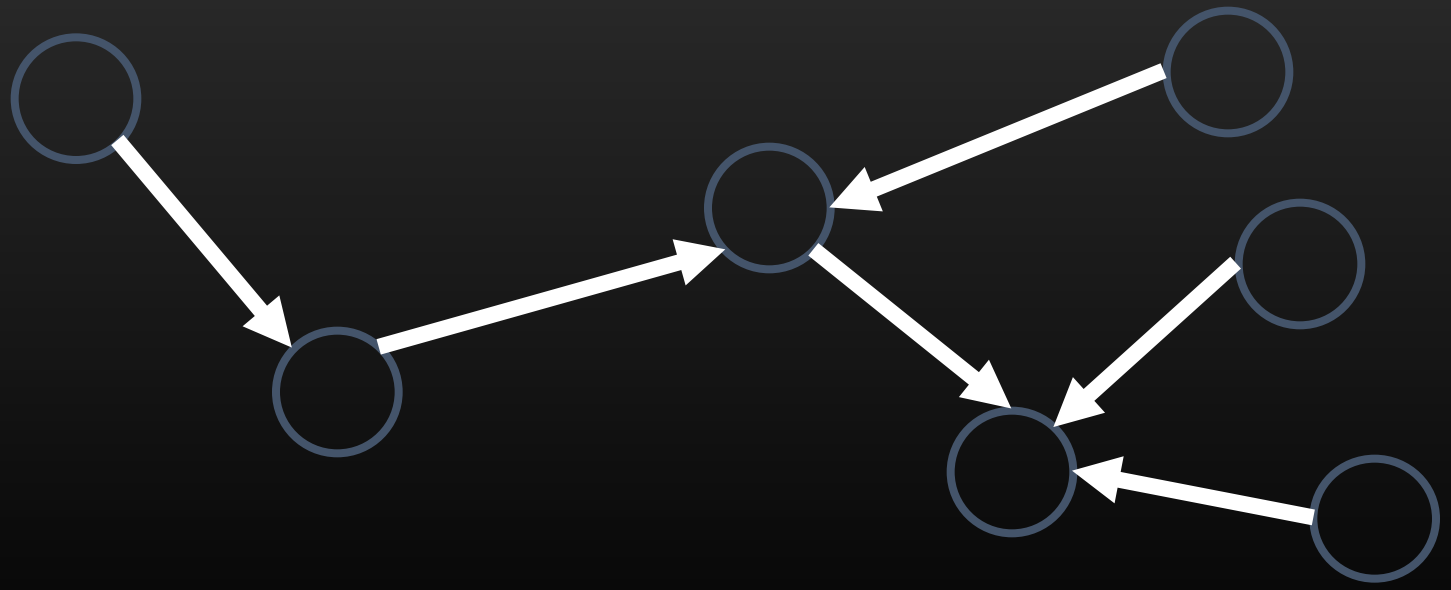
Trees

- Either one of the followings is the definition
  - A connected graph with |V|-1 edges
  - A connected graph without cycles
  - A graph with exactly one path between every pair of vertices



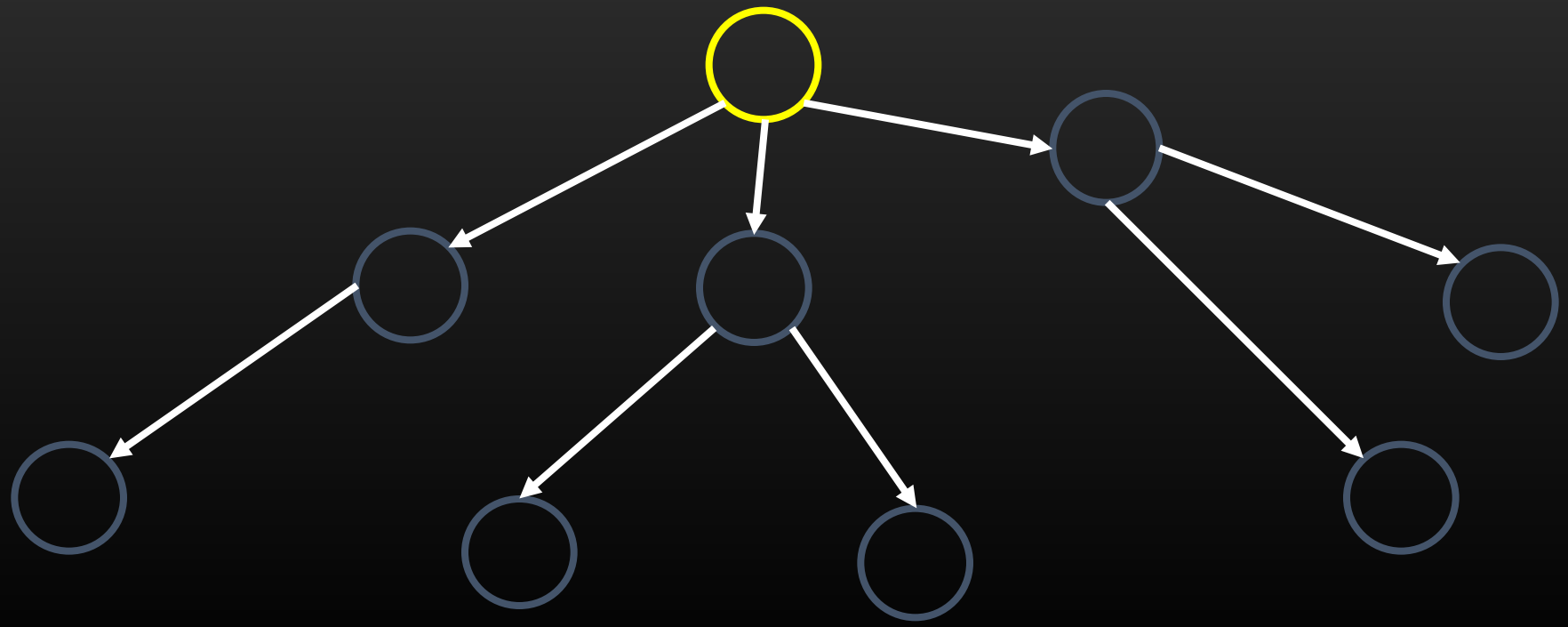
Directed Tree

- Sometimes the edges can be directional

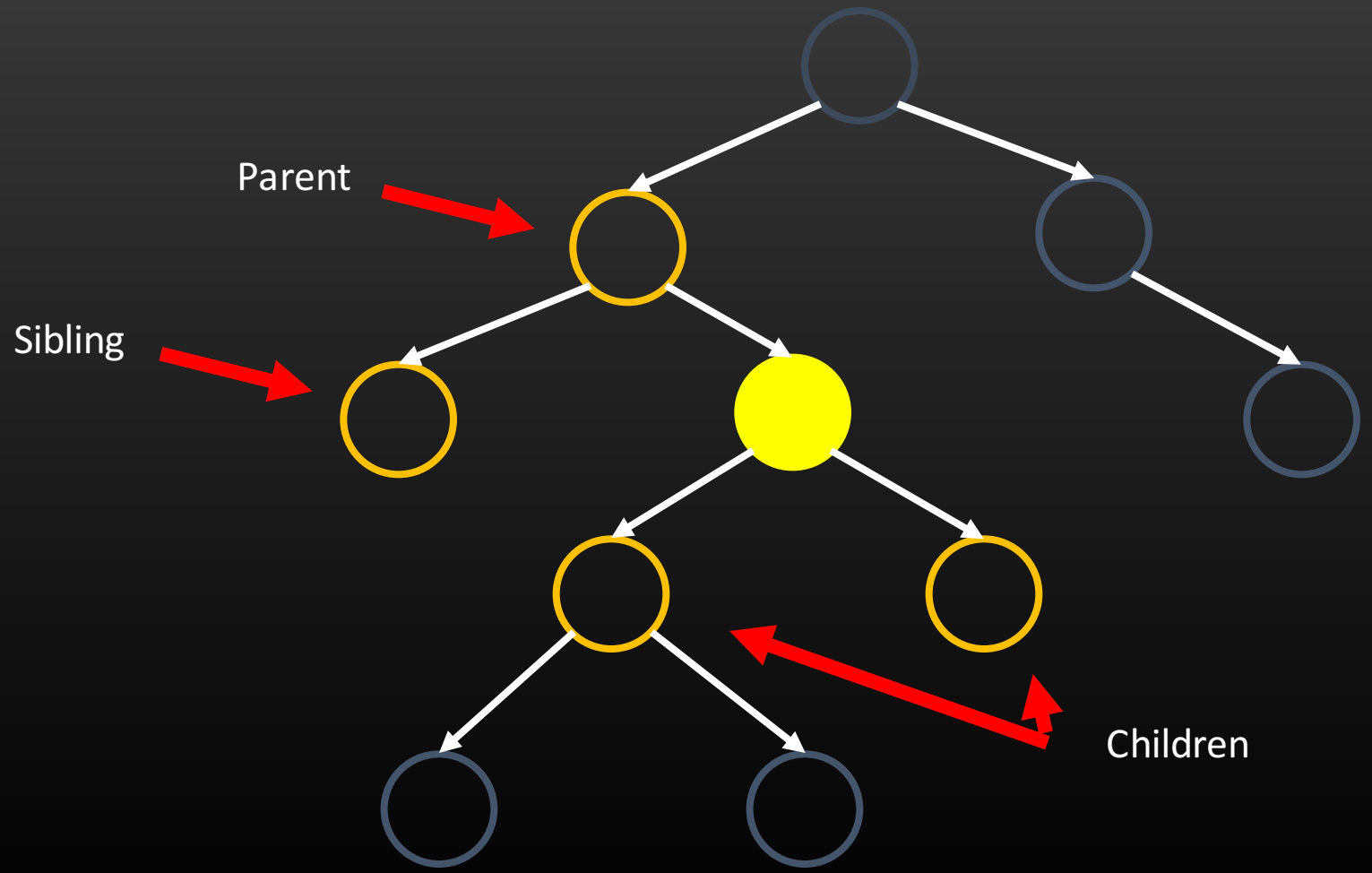


Rooted Tree

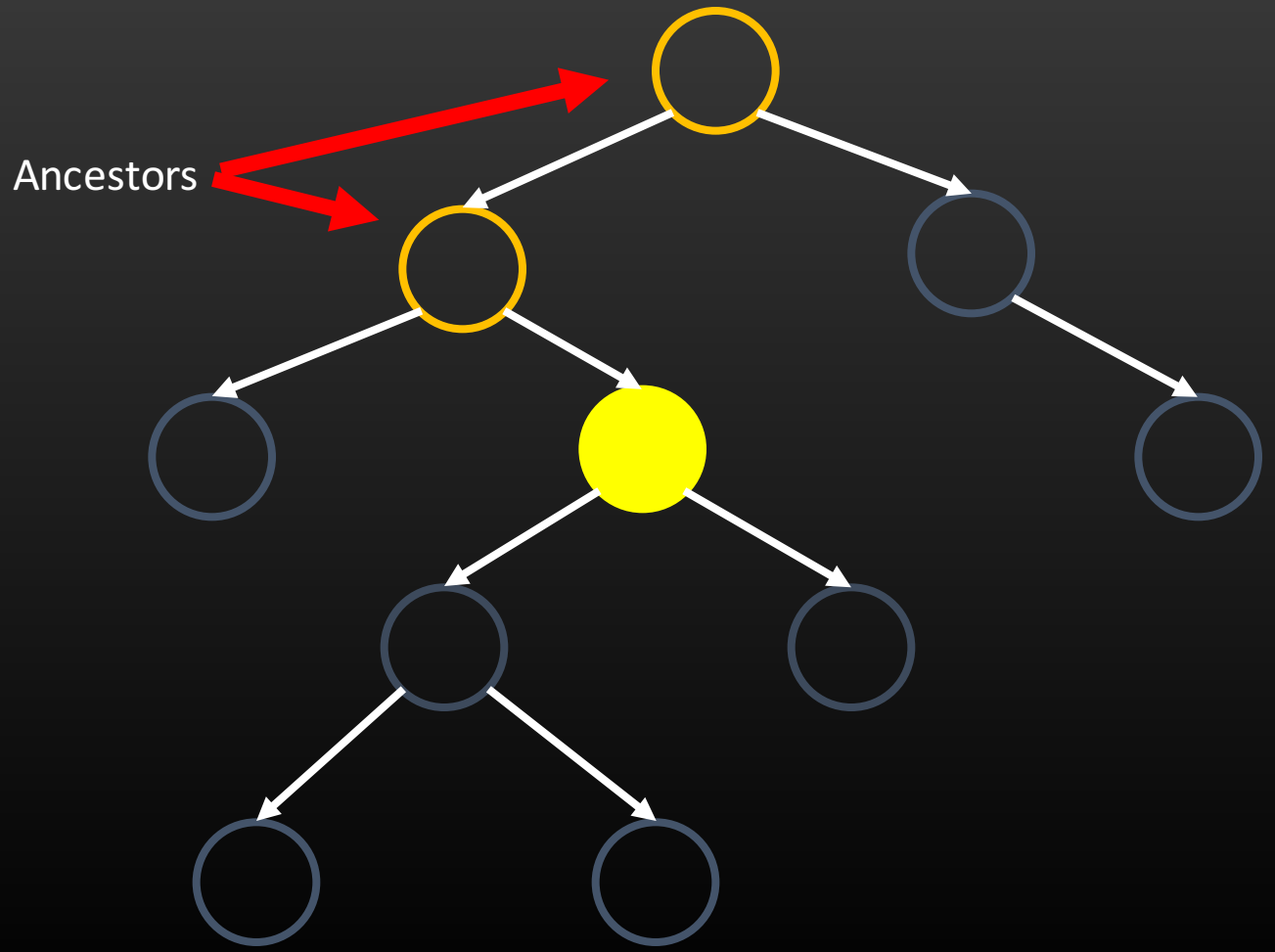
- One vertex has been designated the root
- The edges are directional and all away from the root



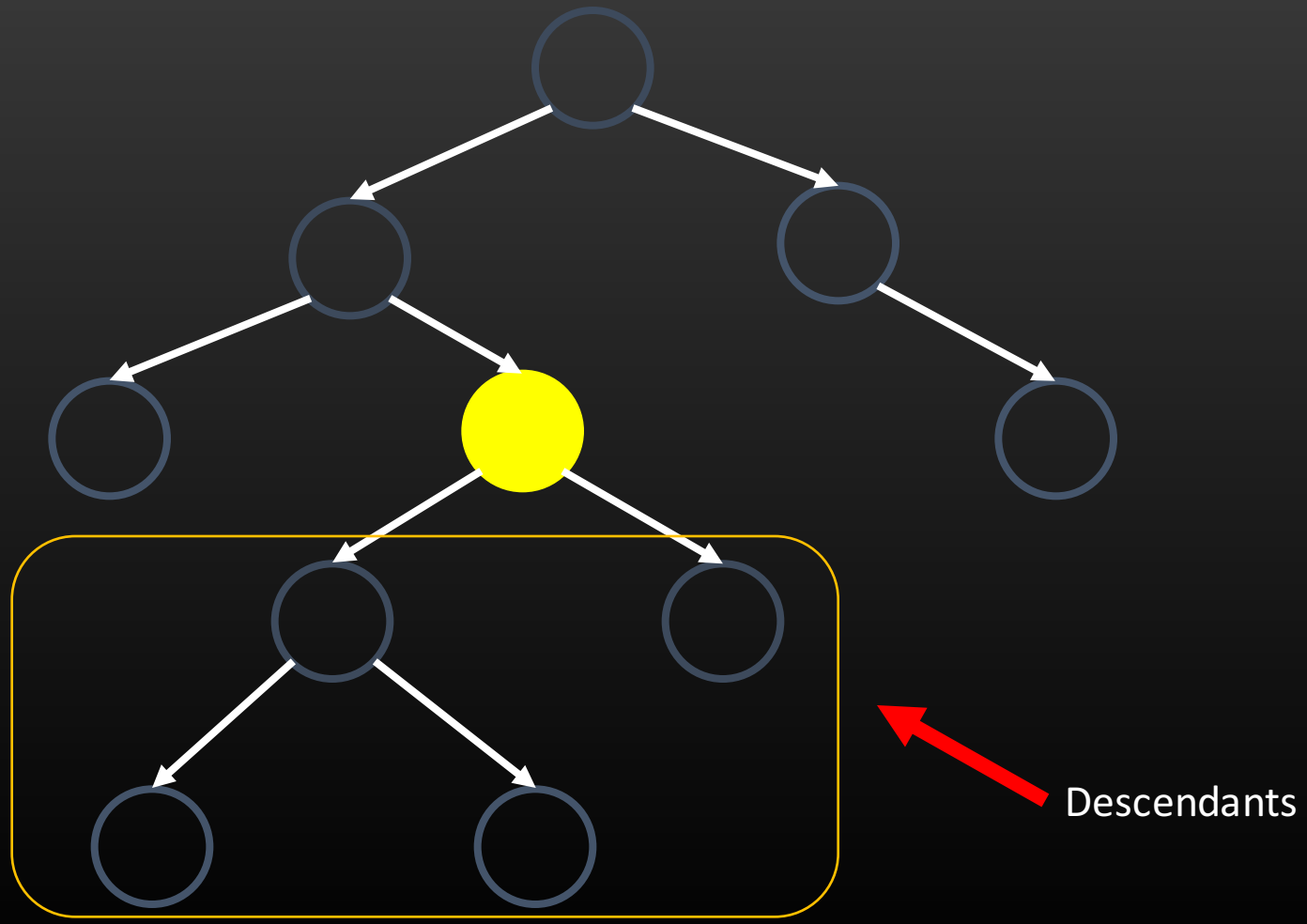
Terms on directed tree



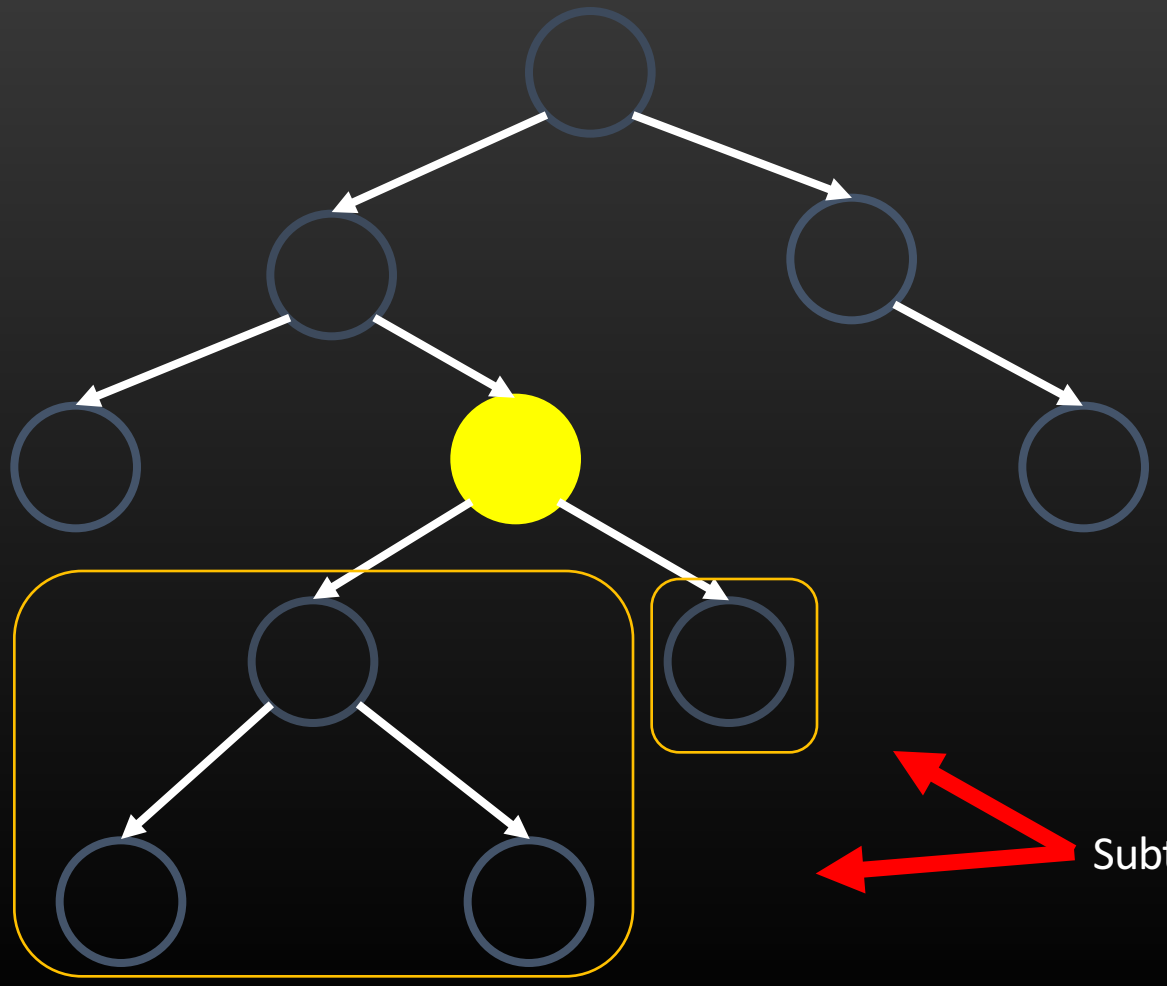
Terms on directed tree



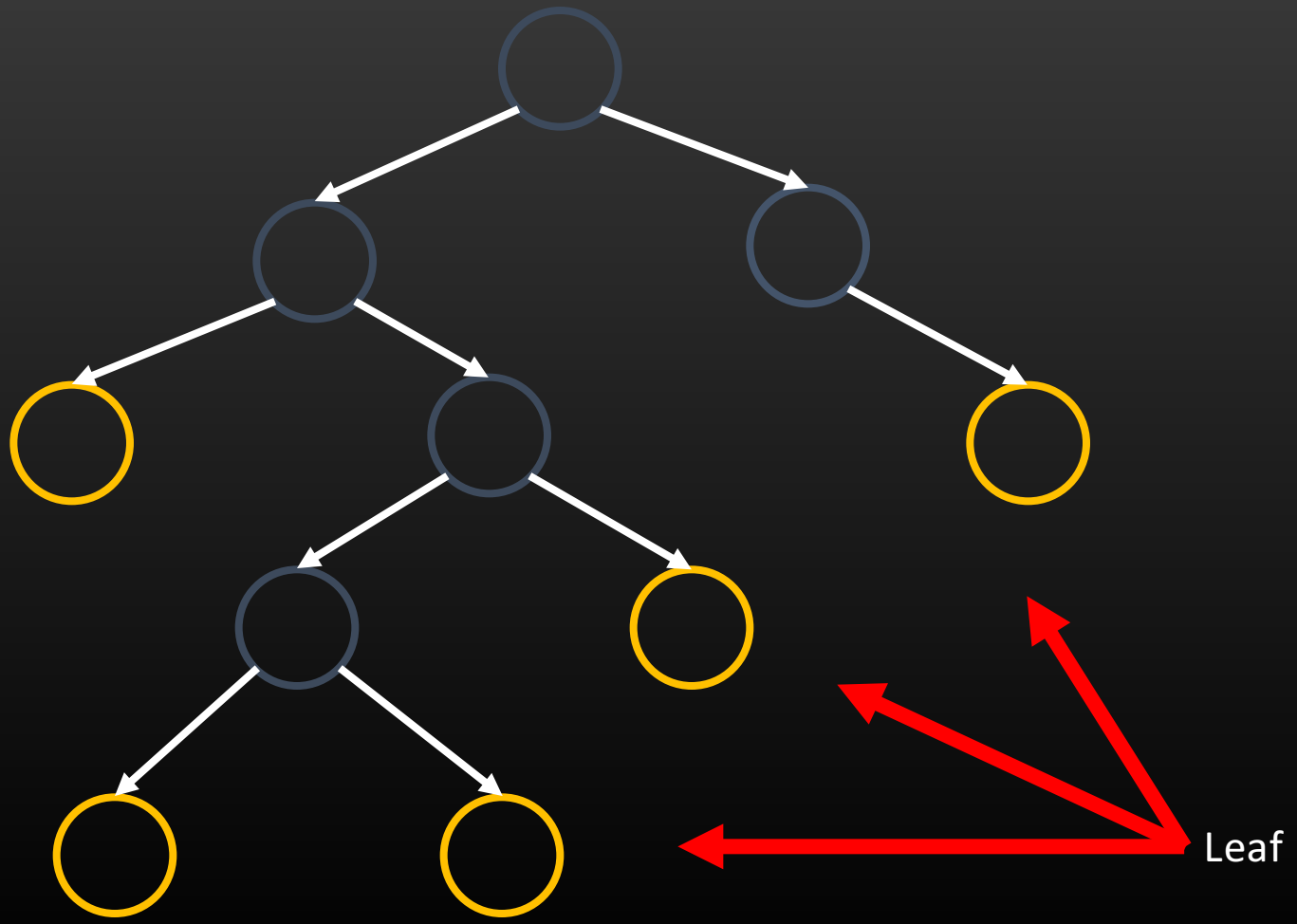
Terms on directed tree



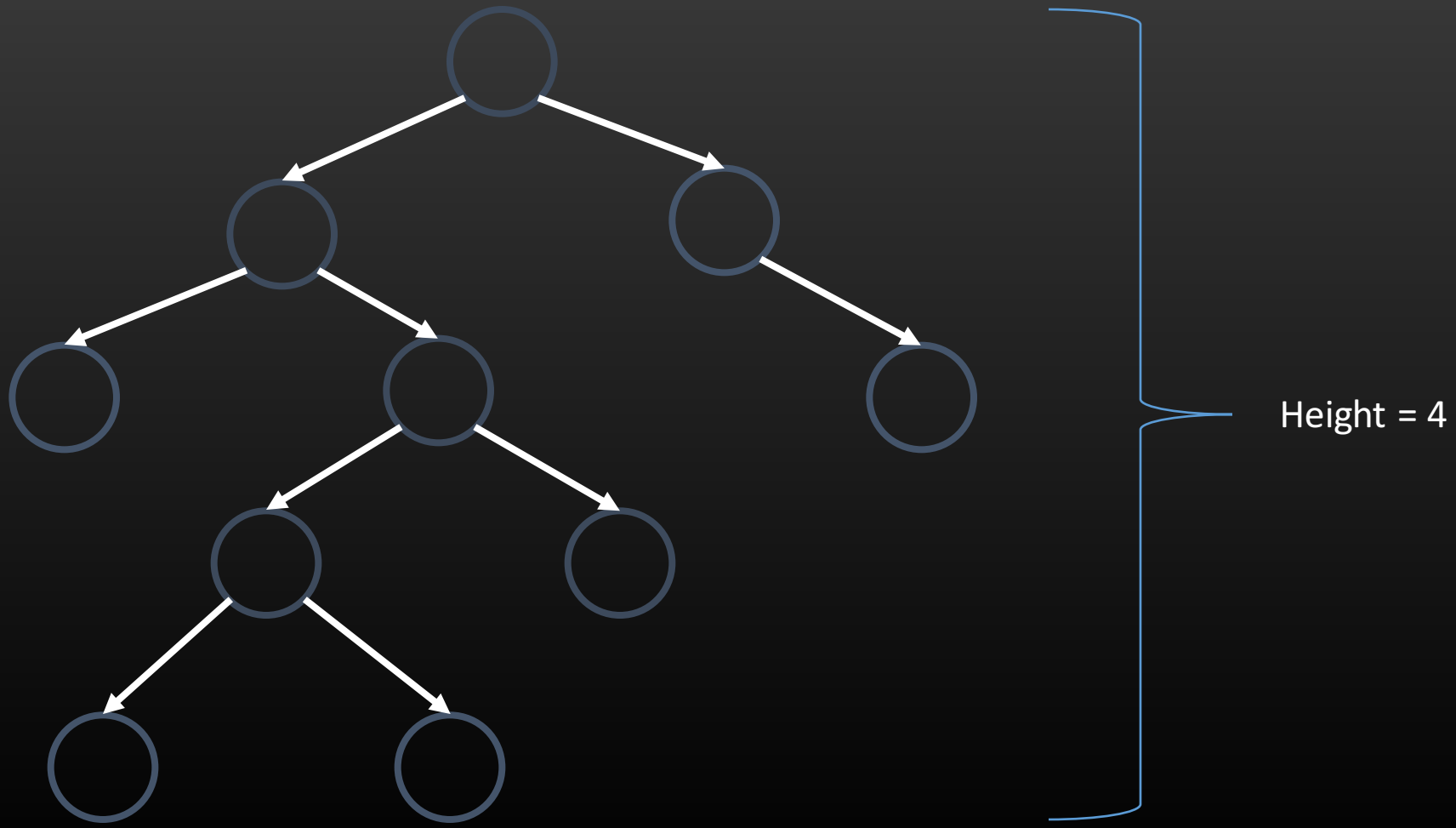
Terms on directed tree



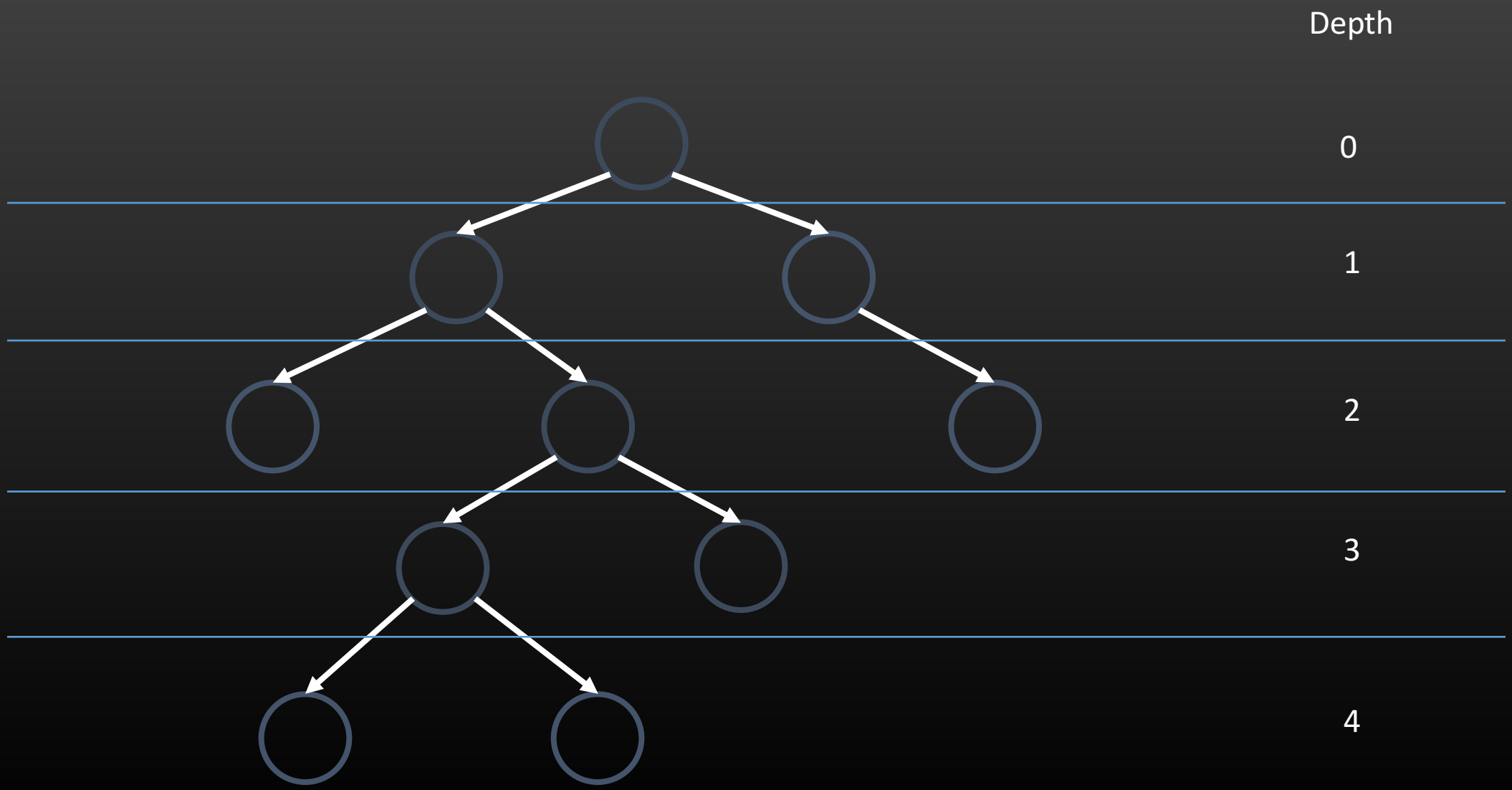
Terms on directed tree



Terms on directed tree

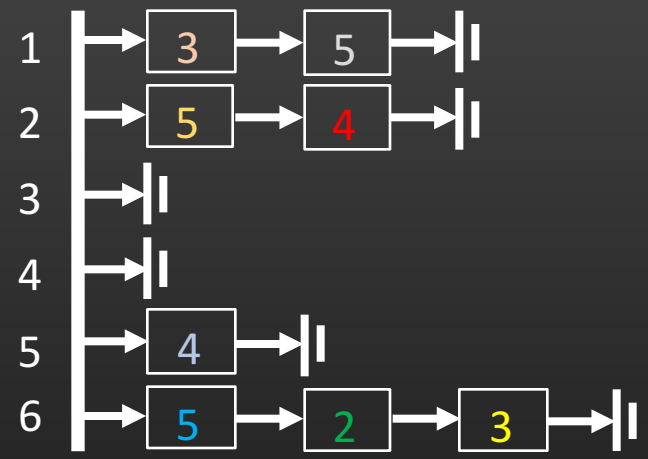


Terms on directed tree



Tree Implementation

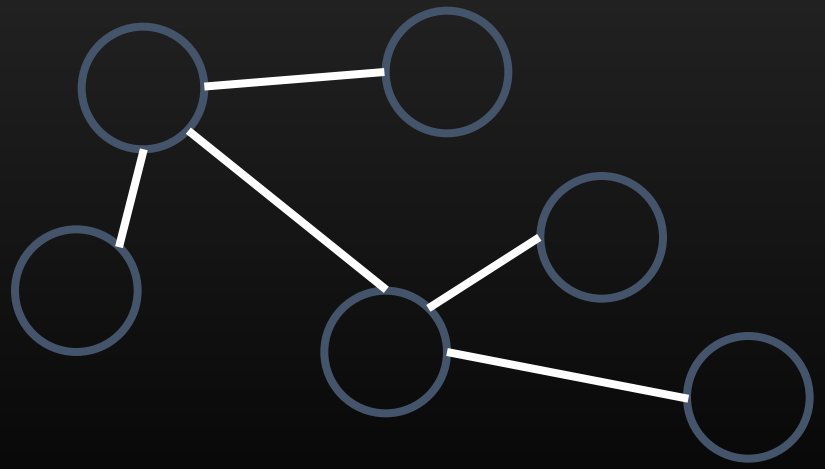
- Trees are also graphs
- Use graph representations
  - Adjacency matrix
  - Adjacency lists
  - Edge list



A	1	2	3	4	5	6
1	0	0	1	0	1	0
2	0	0	0	1	1	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	1	0	0
6	0	1	1	0	1	0

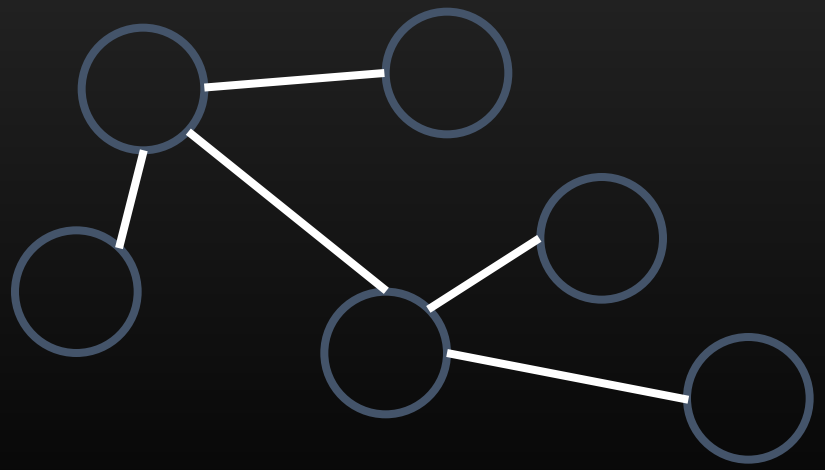
Trees - Usage

- Some problems are reduced or become easier
- Special algorithms can be used



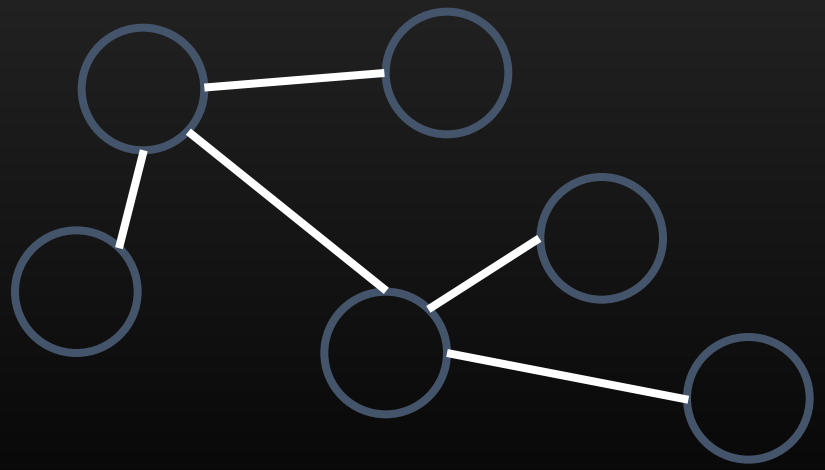
Trees - Usage

- Shortest Path?
  - “A graph with exactly one path between every pair of vertices”
- Minimum Spanning Tree?????
  - It is a tree



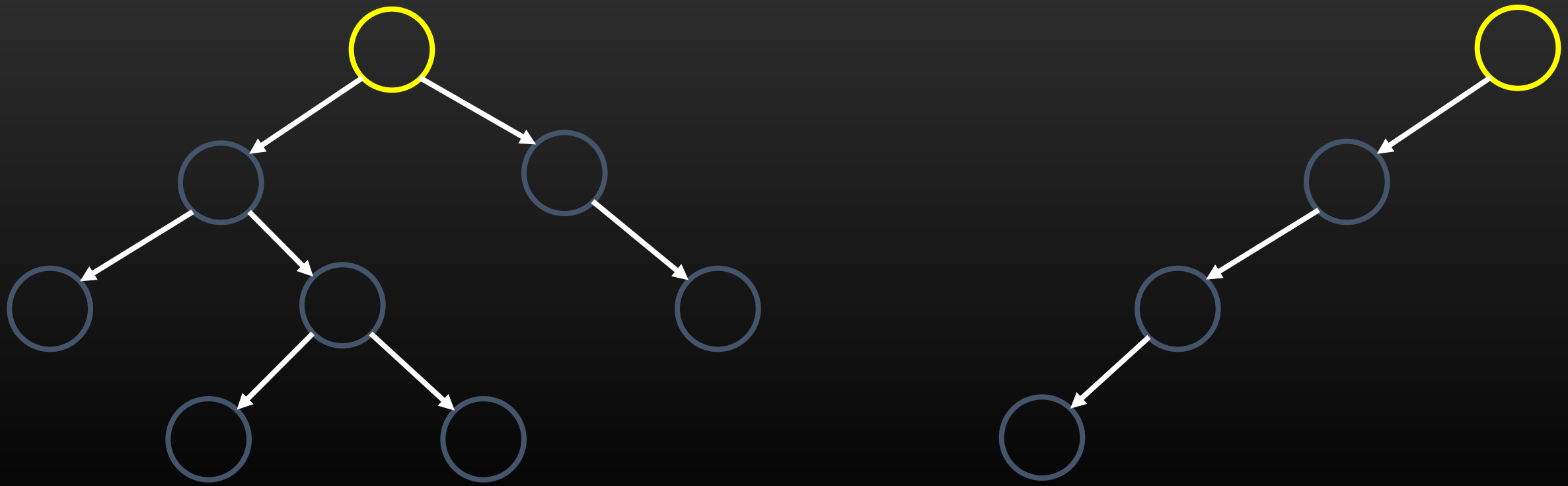
Trees - Usage

- Can also be used as data structures by storing data in a specific way
  - Binary Search Trees
  - Heaps
  - Tries
  - Segment Trees
  - Suffix Trees
  - …



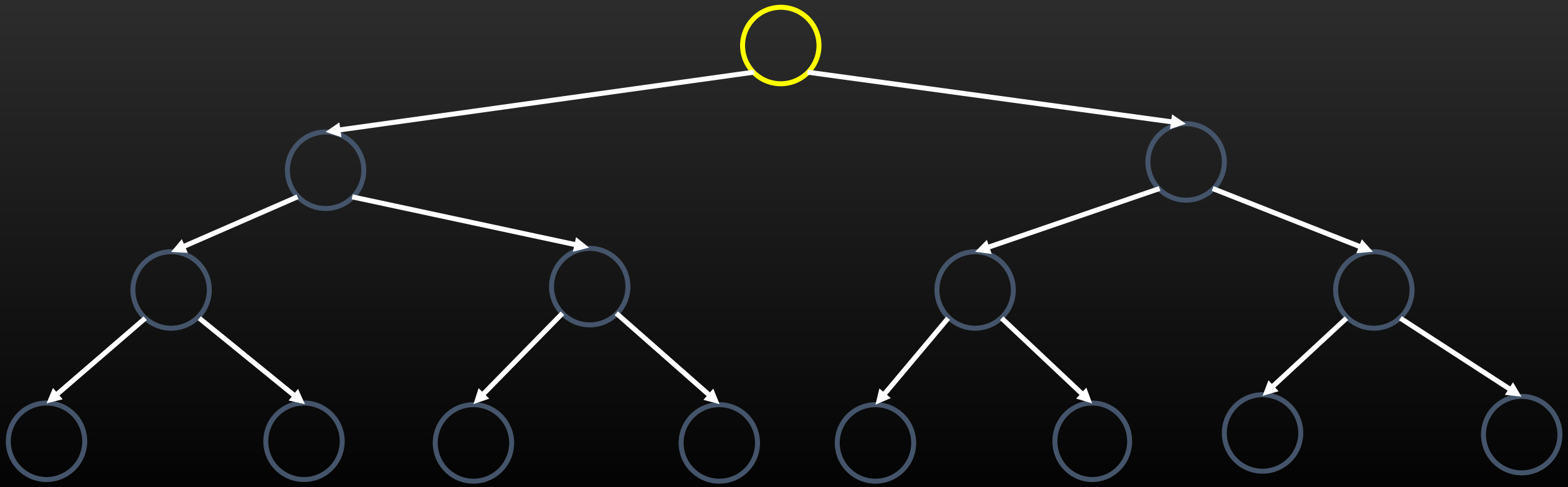
Binary Tree

- A rooted tree which all vertices (nodes) have at most 2 children



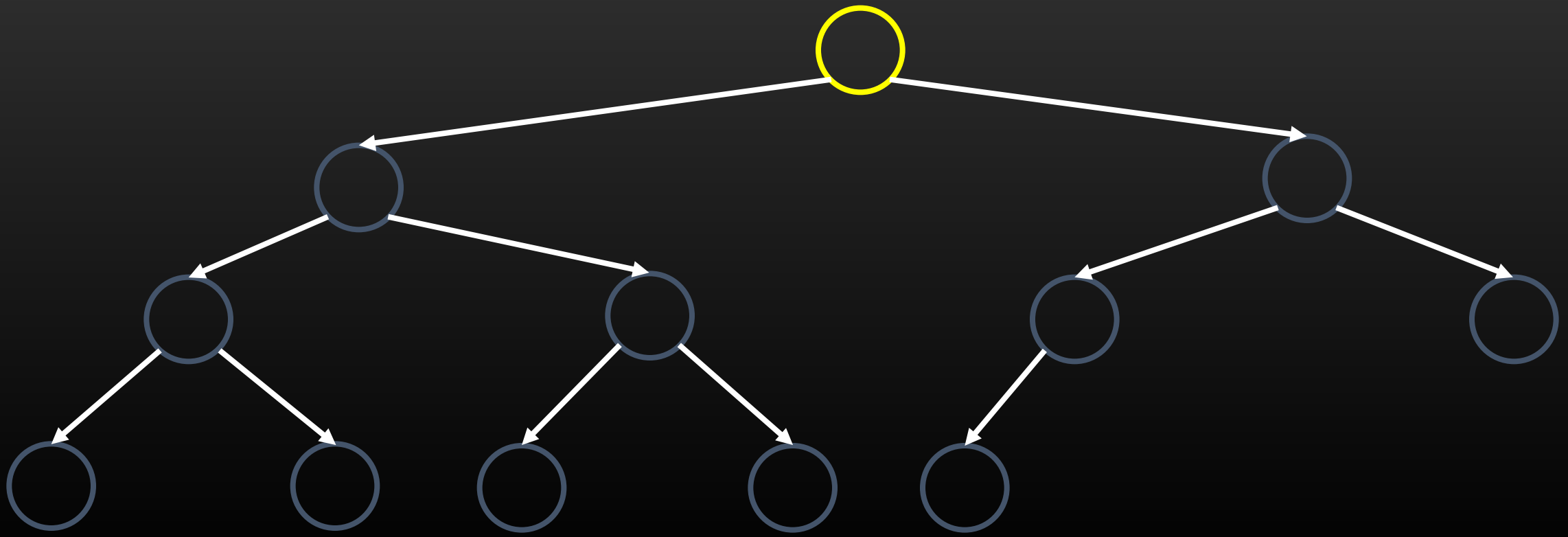
Perfect Binary Tree

- A binary tree in which all non-leaf nodes have two children and all leaves have the same depth



Complete Binary Tree

- A perfect binary tree with some or all rightmost leaf nodes removed



Complete Binary Tree

- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
- The height of a complete binary tree with n nodes is $\lfloor \log n \rfloor$
- Instead of using pointers or lists, it can be implemented using arrays

Binary Tree Implementation

- Use pointers
- Each pointer stores one of the node's children

```
struct node{
    int value;
    node* left, right;
}
```

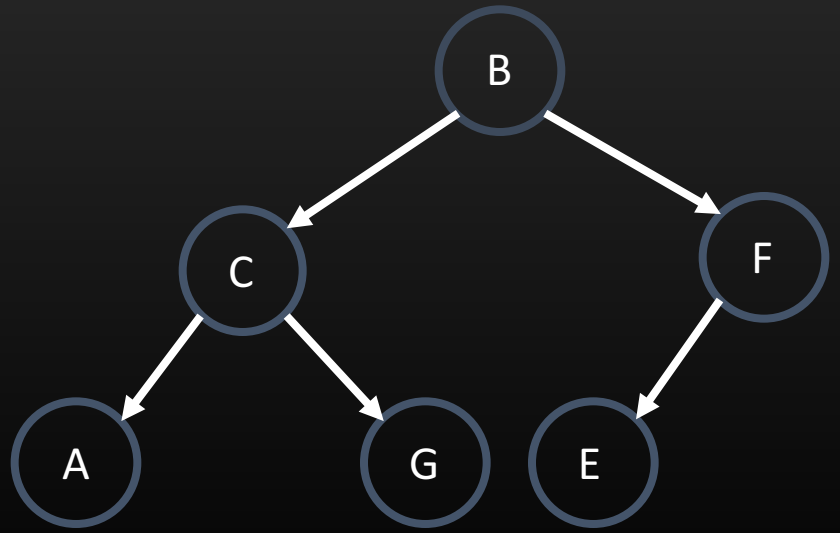
- May be difficult to trace
- Useful when implementing balanced BST (e.g. AVL Tree)

Complete Binary Tree Implementation

- The root node is stored in array position 1
- For any element in array position i:
  - The left child is in position 2 * i
  - The right child is in the cell after the left child (2 * i + 1)
  - The parent is in position $\lfloor i / 2 \rfloor$

Complete Binary Tree Implementation

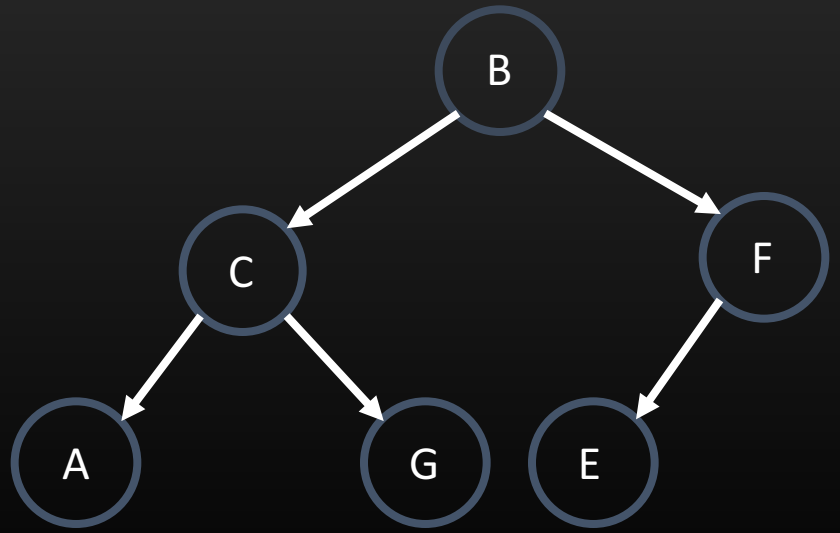
Index	0	1	2	3	4	5	6
A[]		B	C	F	A	G	E



Complete Binary Tree Implementation

Index	0	1	2	3	4	5	6
A[]		B	C	F	A	G	E

Depth = 0 Depth = 1 Depth = 2



Complete Binary Tree Implementation

- Nodes with depth h are stored in position 2^h to $2^{h+1}-1$
- Space Complexity : O(N)

- Useful in implementing tree data structures
  - Binary Heap
  - Segment tree

Tree Traversal

- Like graphs, nodes in a tree can also be visited using Depth First Search (DFS) and Breath First Search (BFS)

Depth First Search

- The search tree is deepened as much as possible on each child before going to the next sibling

```
procedure DFS(vertex v){
    label v as visited
    for all w in children of v do
            DFS(w)
    }
```

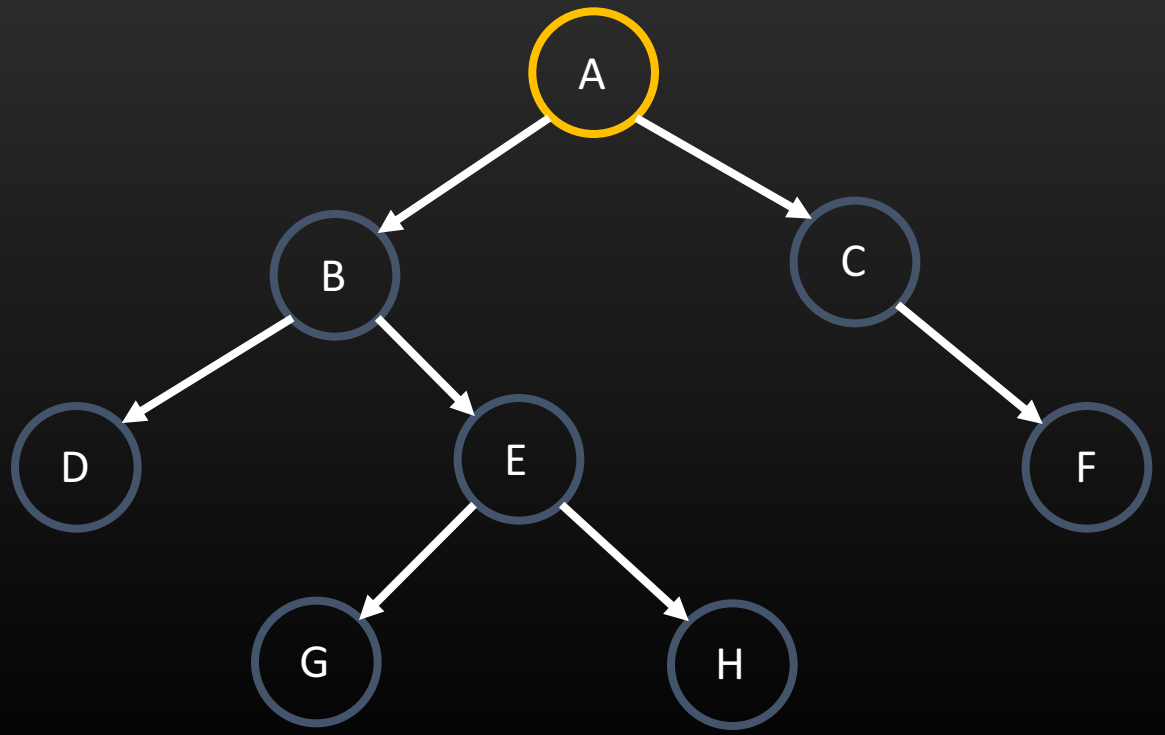
Depth First Search

- Usually we only want to process every node exactly once
- For a binary tree, we have three orders for tree traversal:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

Depth First Search – Pre-order

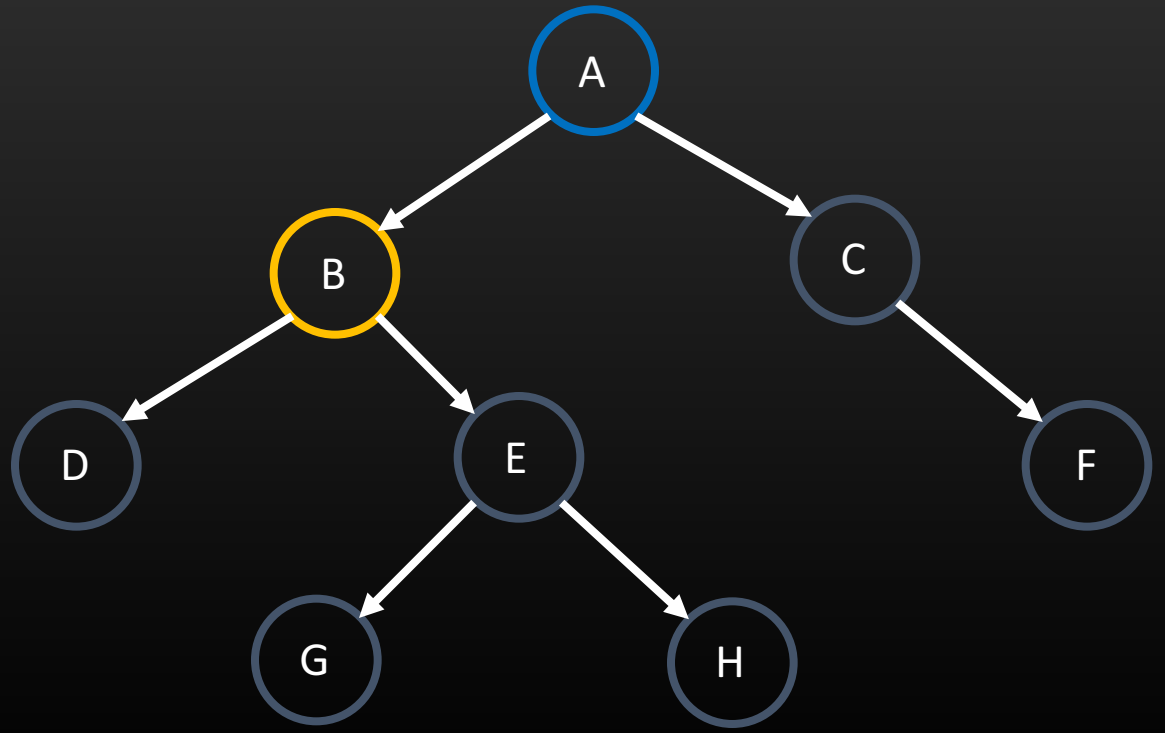
- Starting from the root
- For the traversal at node N:
  - Process the node N
  - Recursively traverse its left subtree
  - Recursively traverse its right subtree

Depth First Search – Pre-order



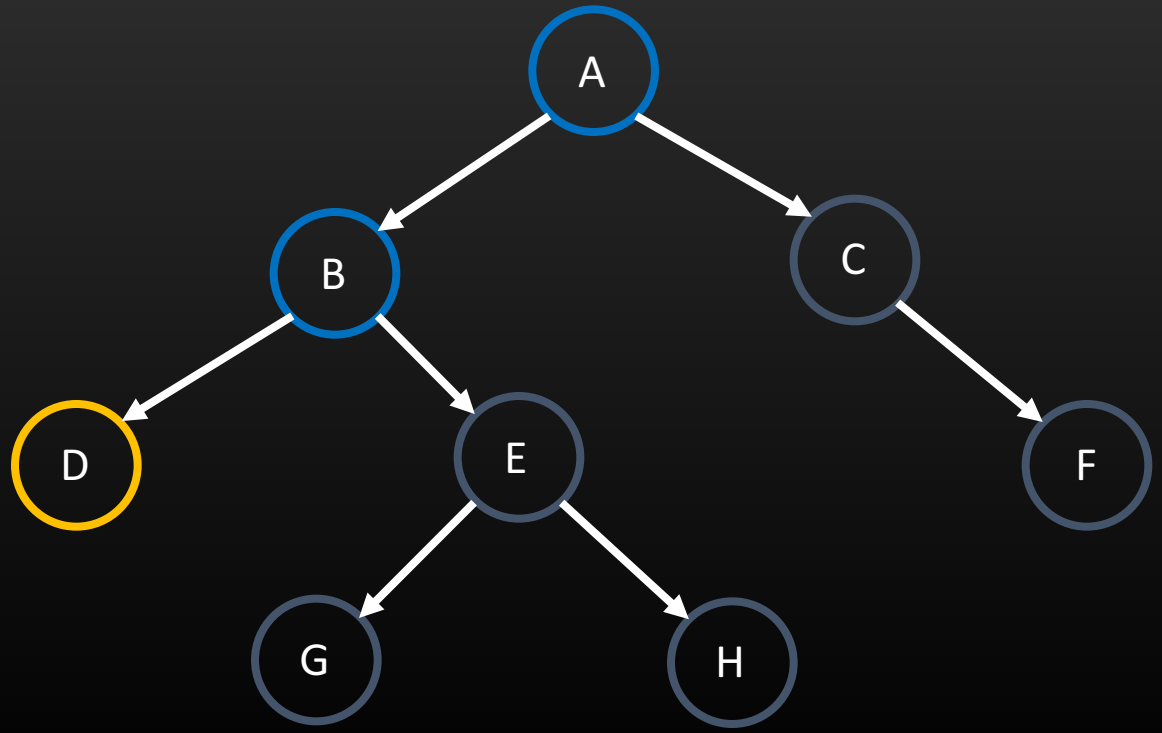
Depth First Search – Pre-order

A	B							



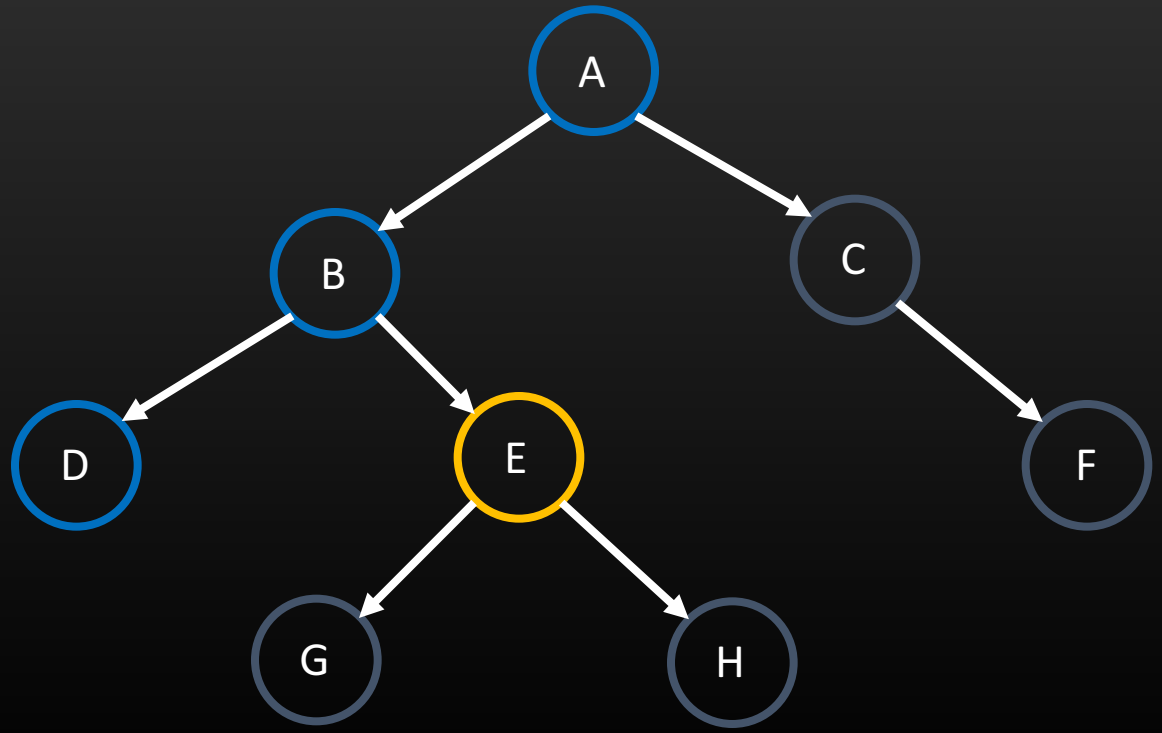
Depth First Search – Pre-order

A	B	D						



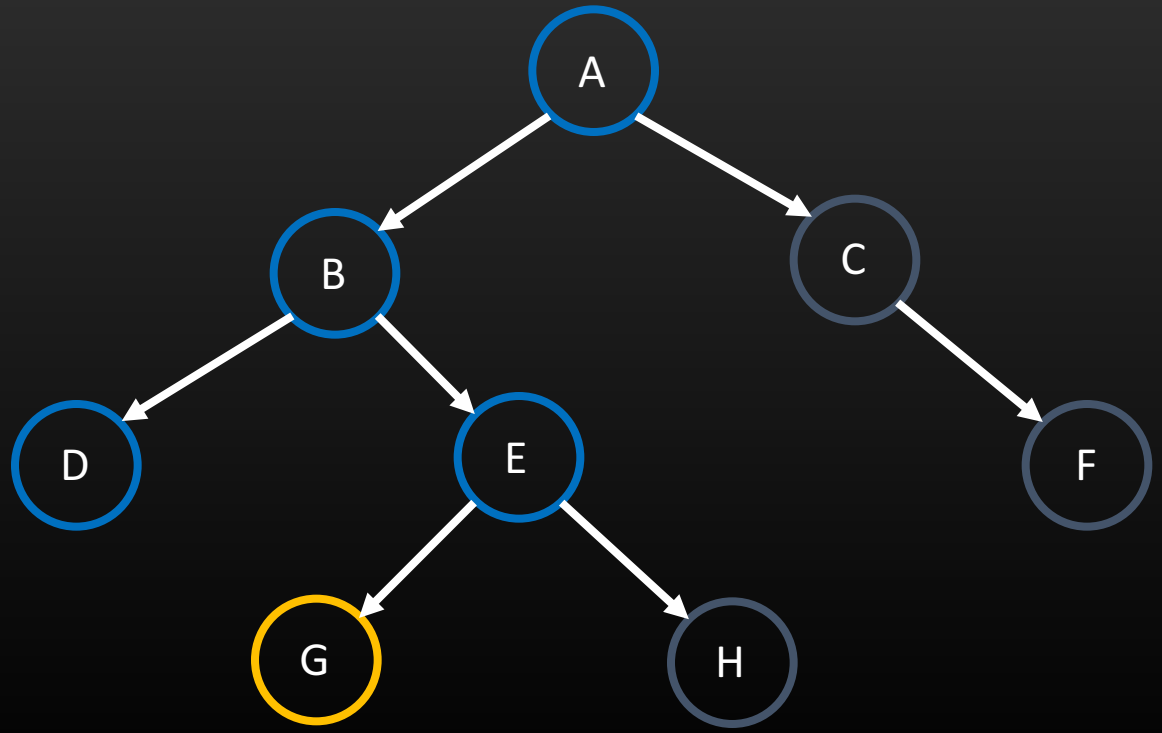
Depth First Search – Pre-order

A	B	D	E					



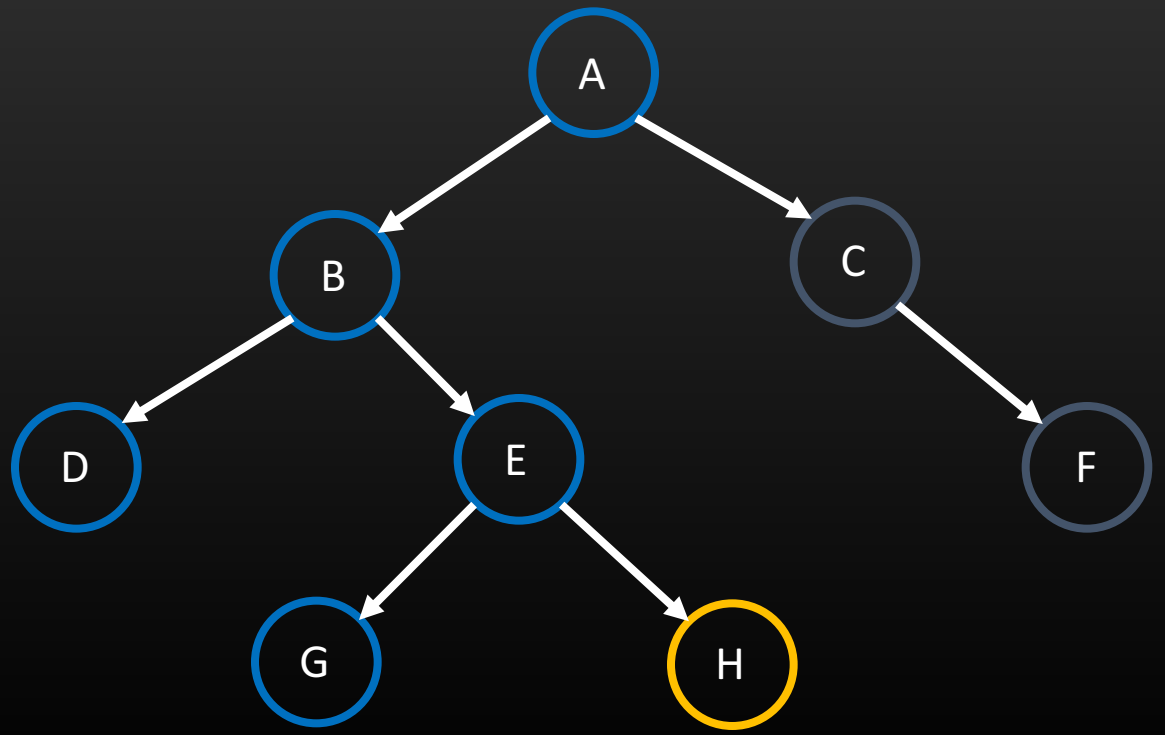
Depth First Search – Pre-order

A	B	D	E	G				



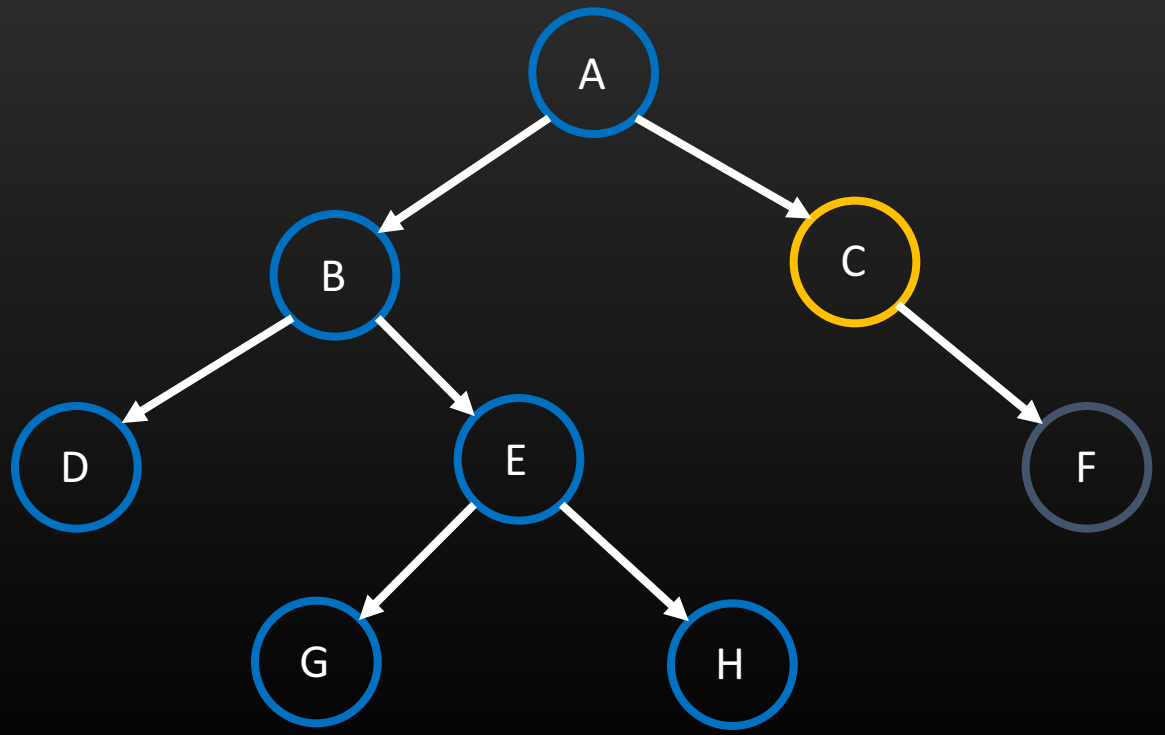
Depth First Search – Pre-order

A	B	D	E	G	H			



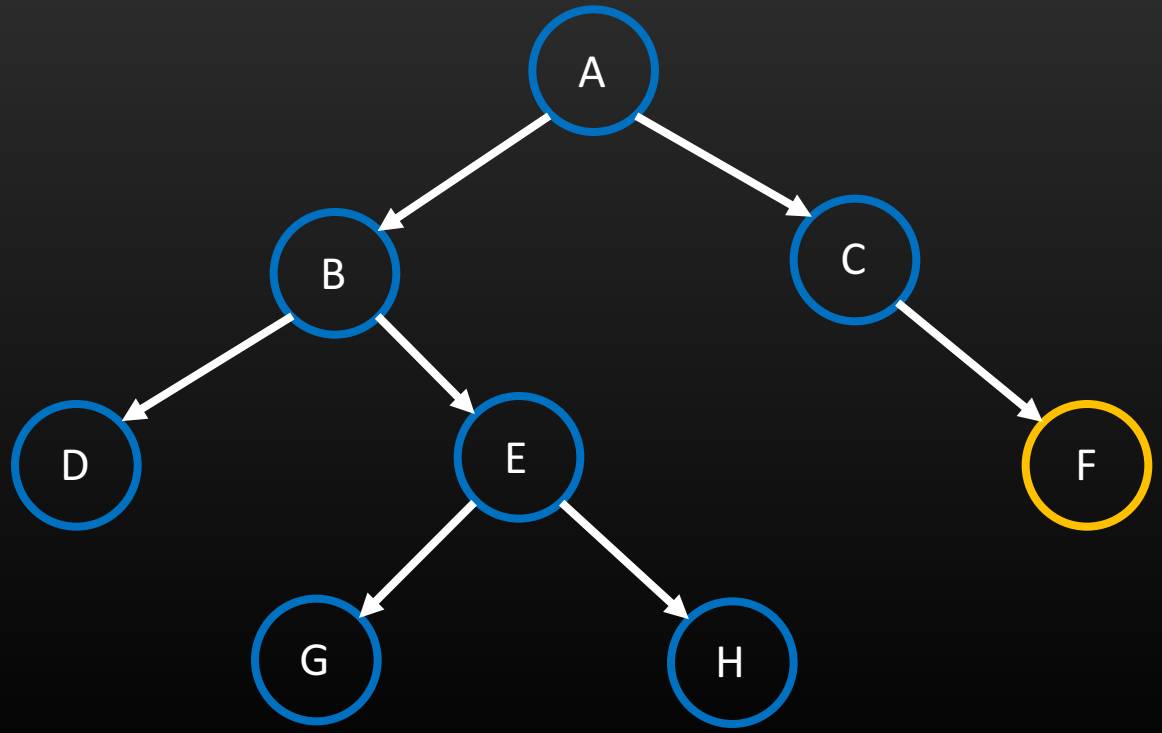
Depth First Search – Pre-order

A	B	D	E	G	H	C		



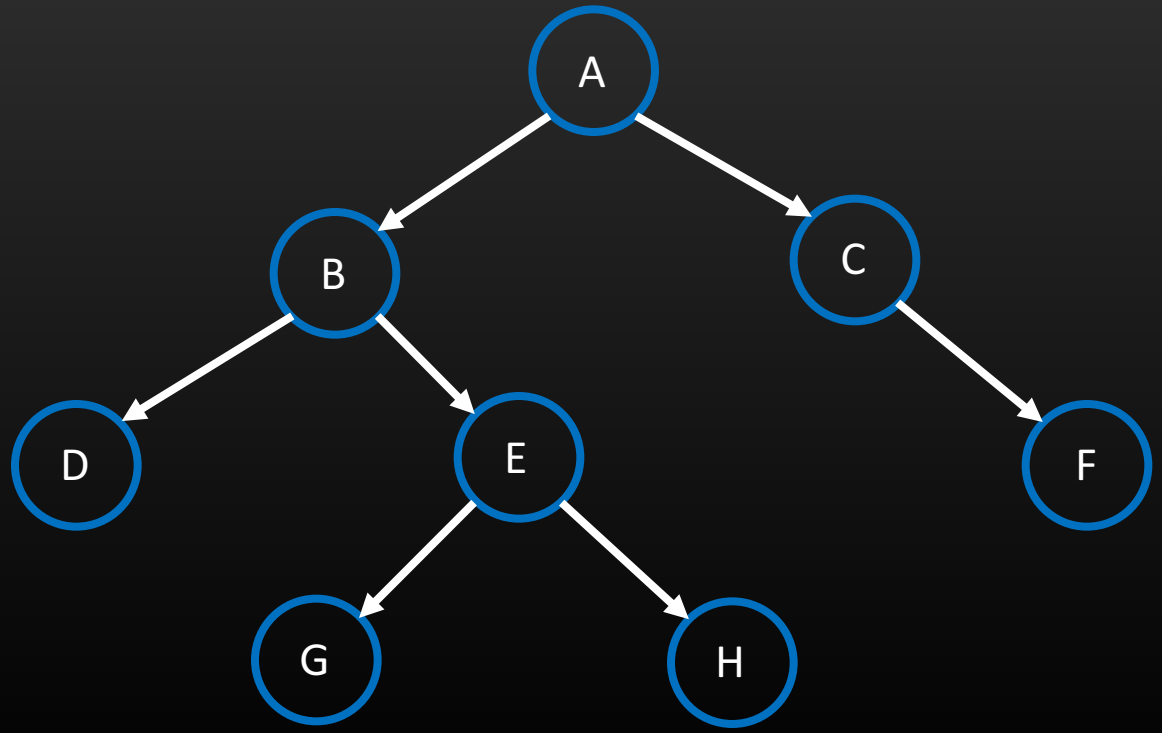
Depth First Search – Pre-order

A	B	D	E	G	H	C	F	



Depth First Search – Pre-order

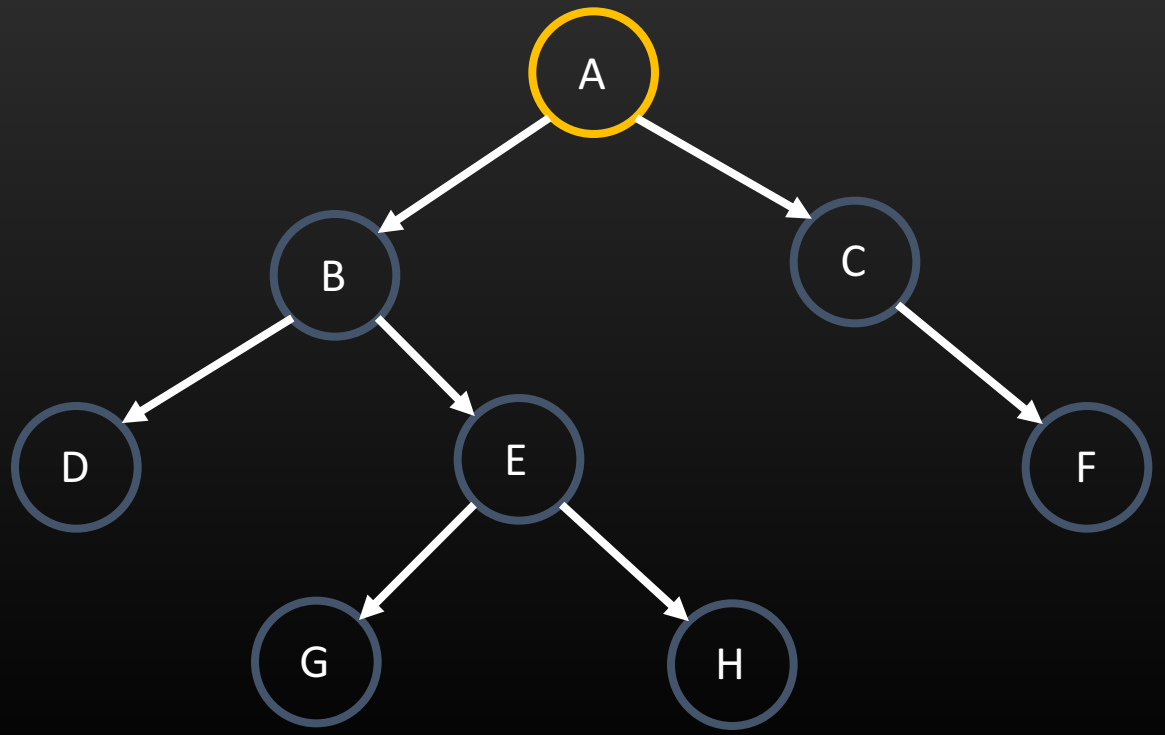
A	B	D	E	G	H	C	F	



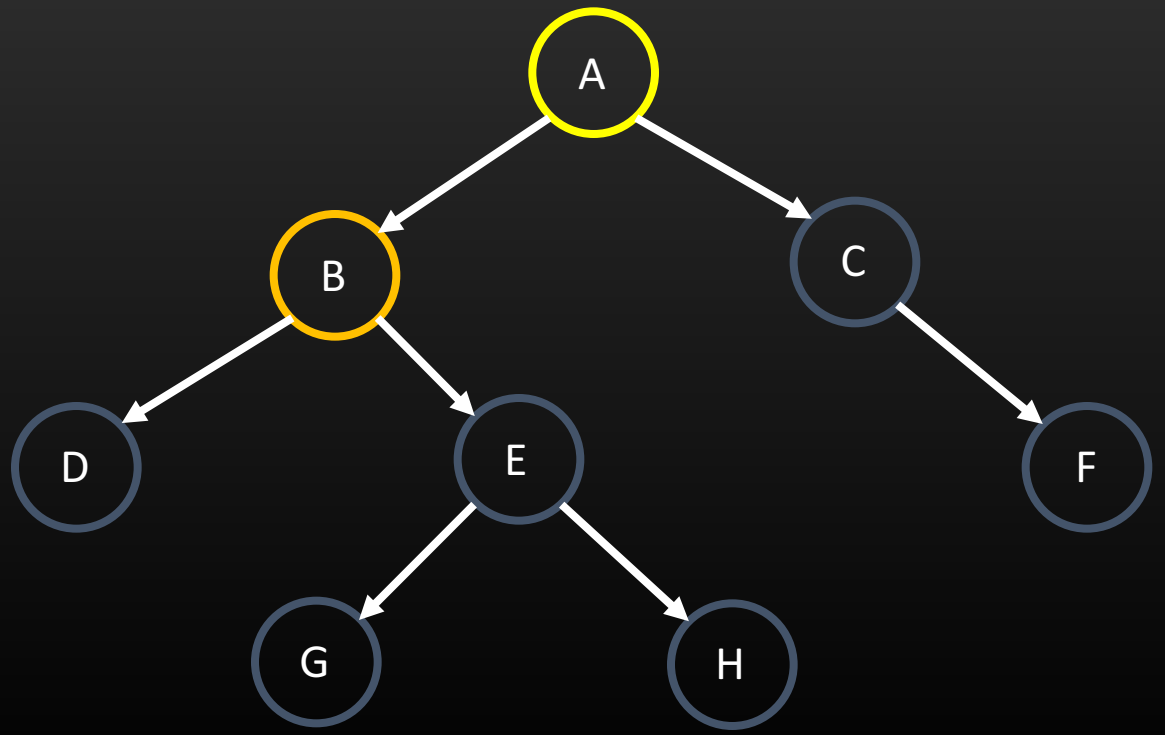
Depth First Search – In-order

- Starting from the root
- For the traversal at node N:
  - Recursively traverse its left subtree
  - Process the node N
  - Recursively traverse its right subtree

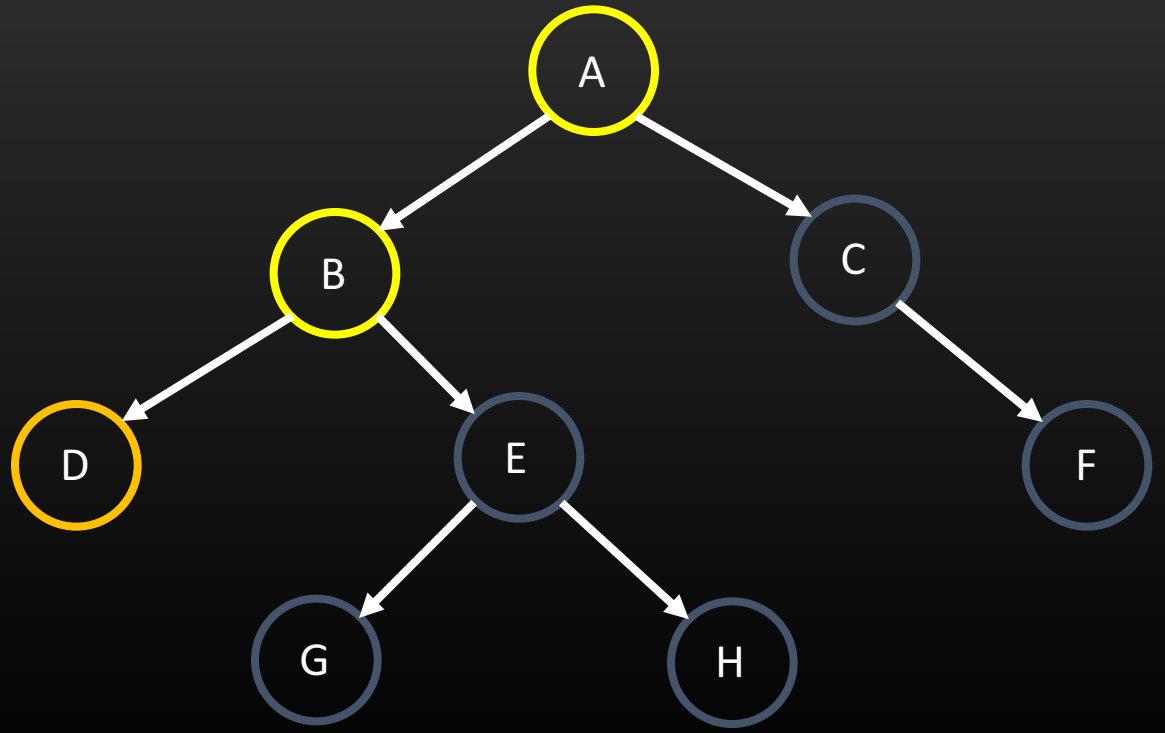
Depth First Search – In-order



Depth First Search – In-order

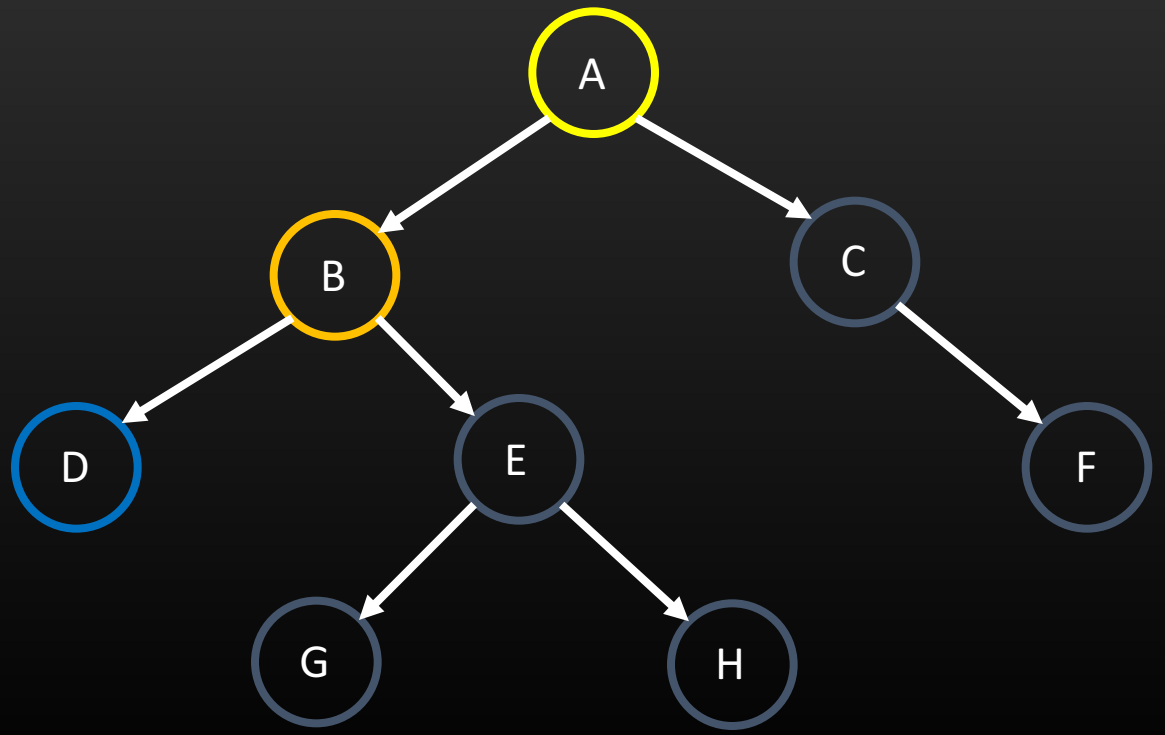


Depth First Search – In-order



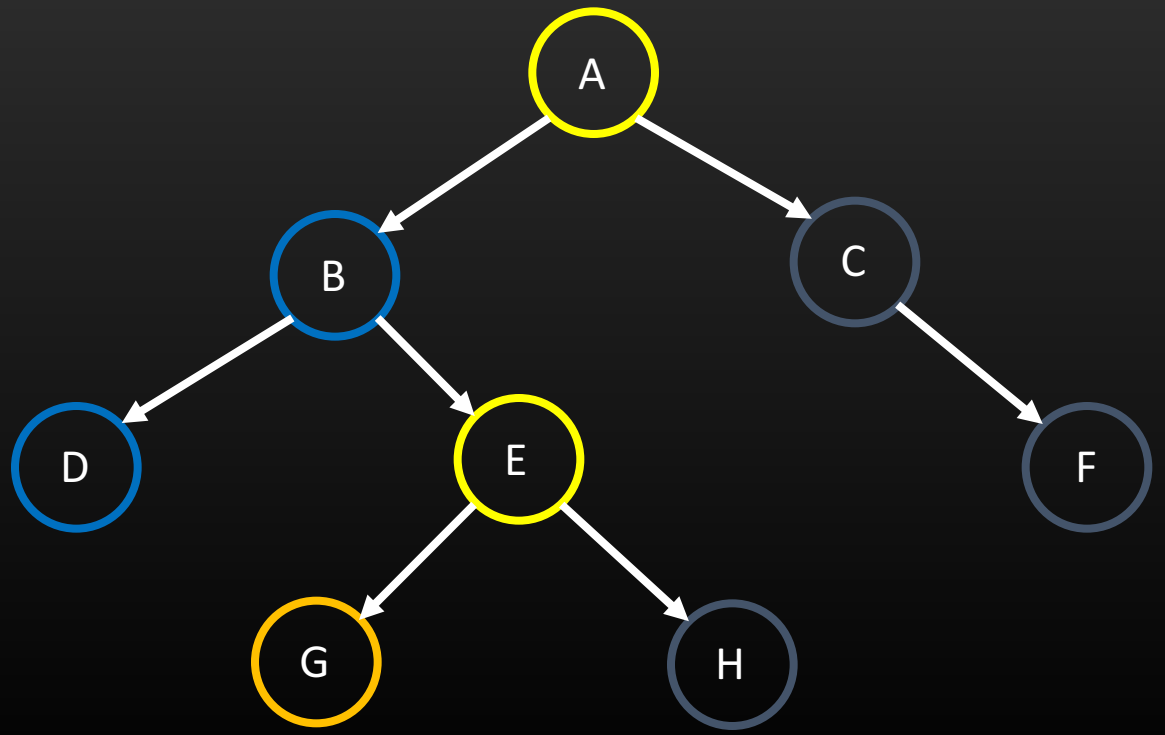
Depth First Search – In-order

D	B							



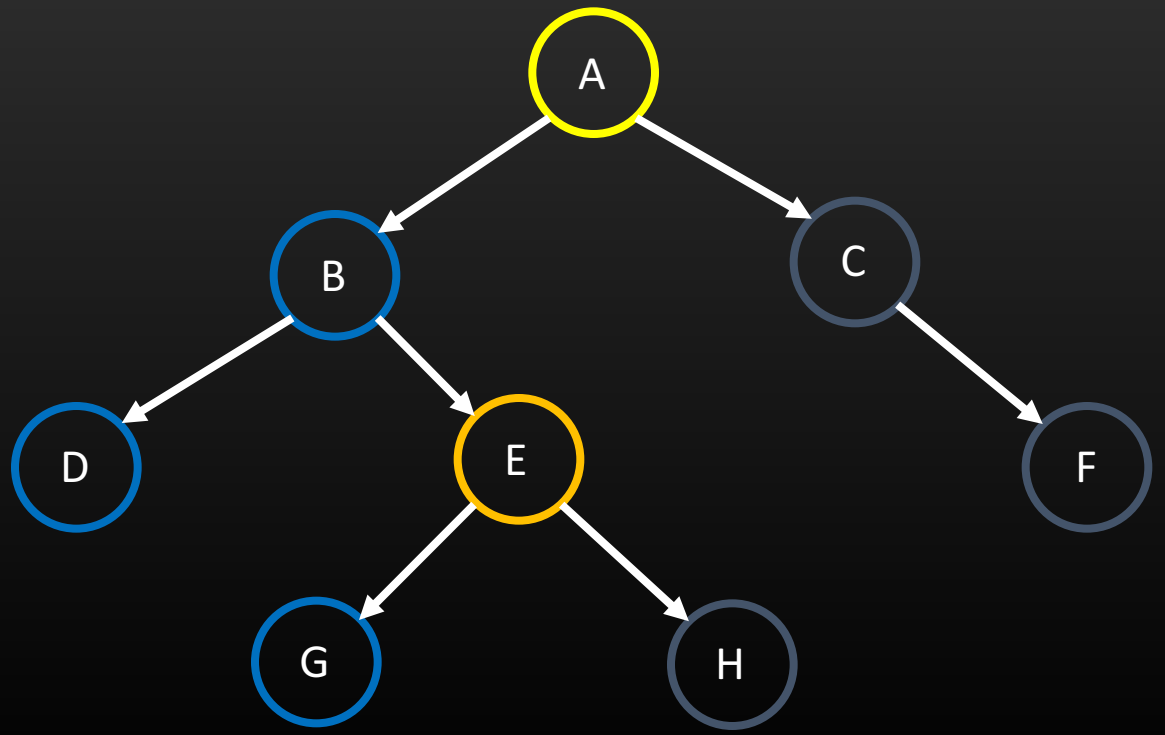
Depth First Search – In-order

D	B	G						



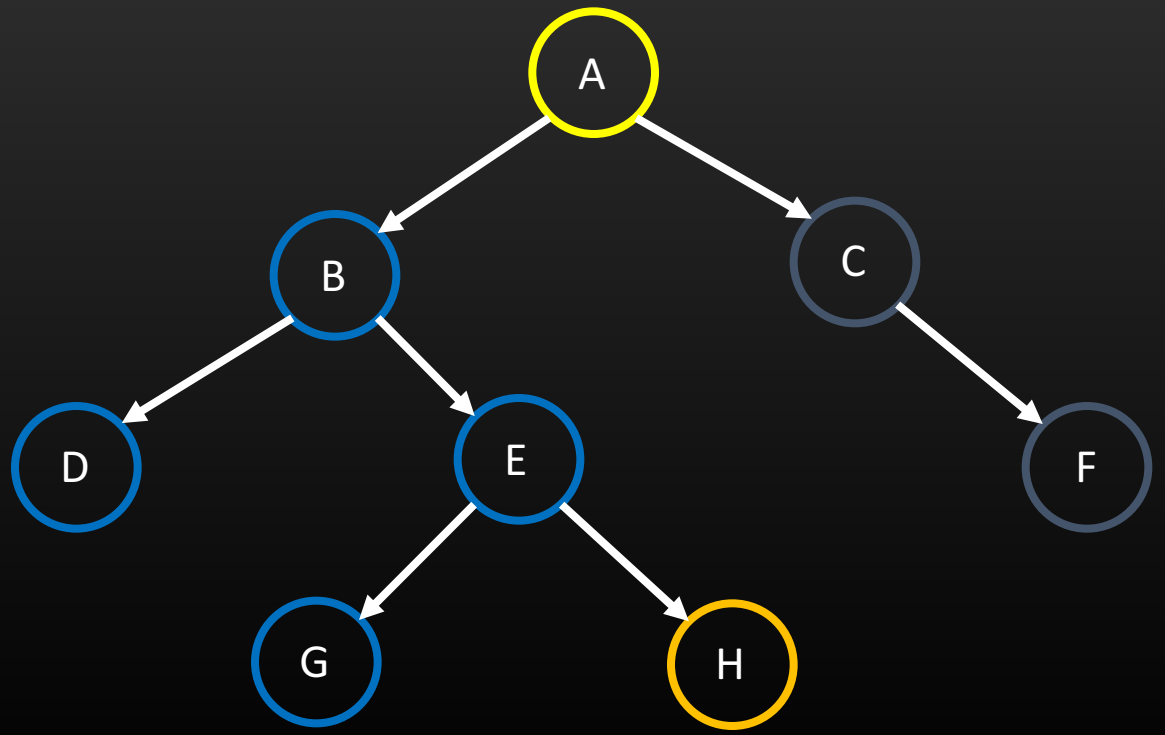
Depth First Search – In-order

D	B	G	E					



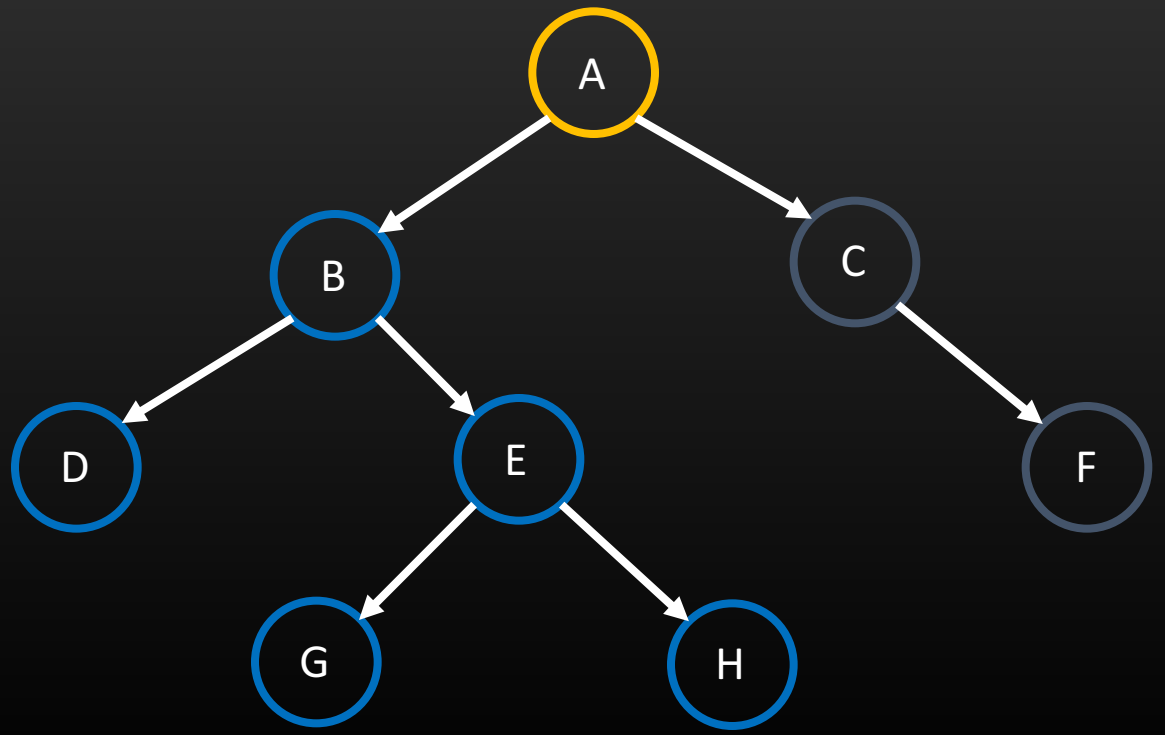
Depth First Search – In-order

D	B	G	E	H				



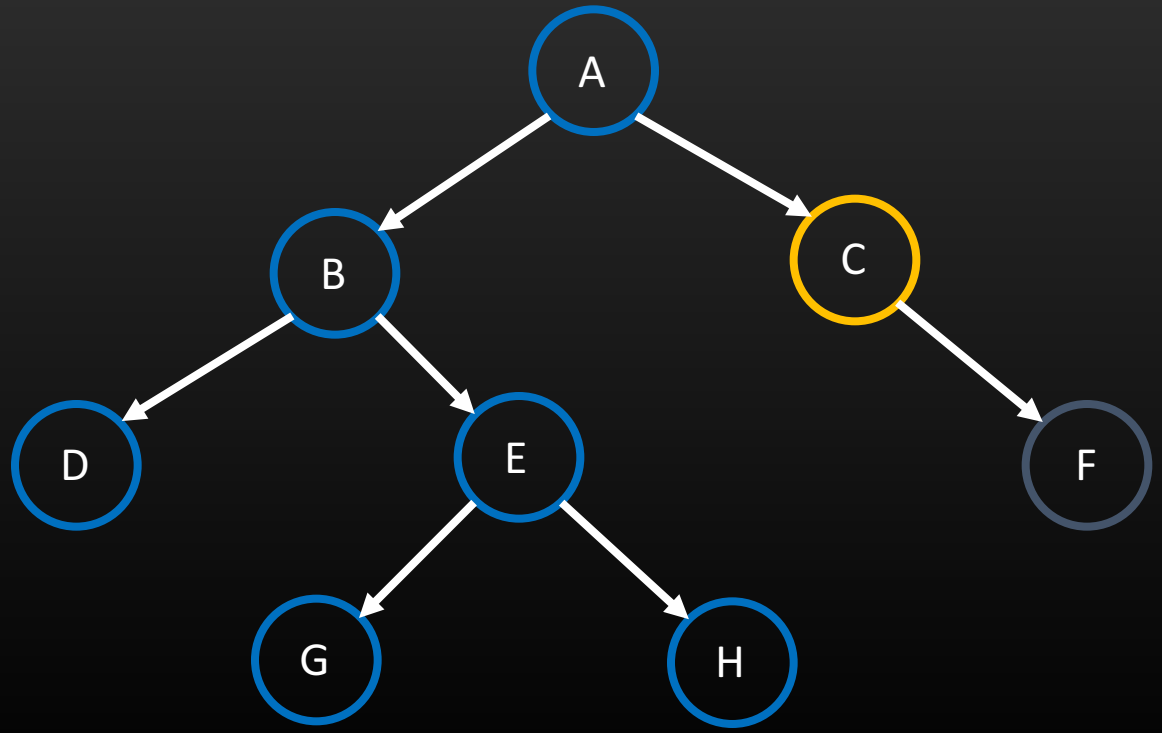
Depth First Search – In-order

D	B	G	E	H	A			



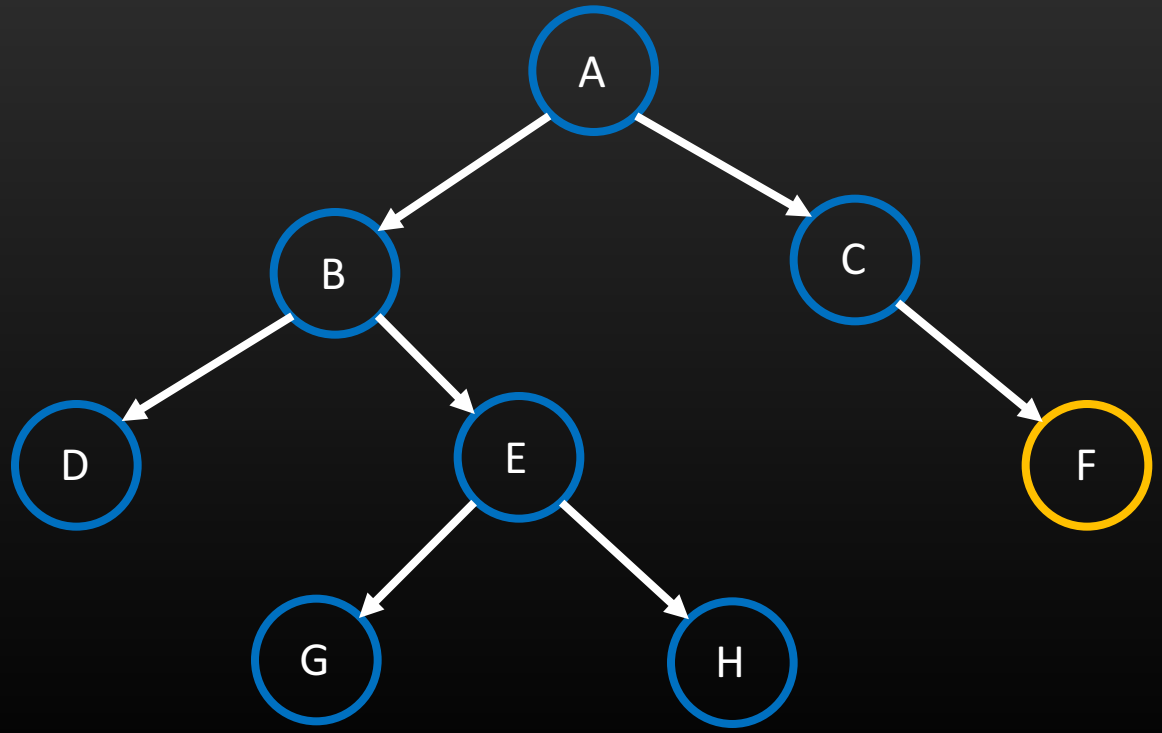
Depth First Search – In-order

D	B	G	E	H	A	C		



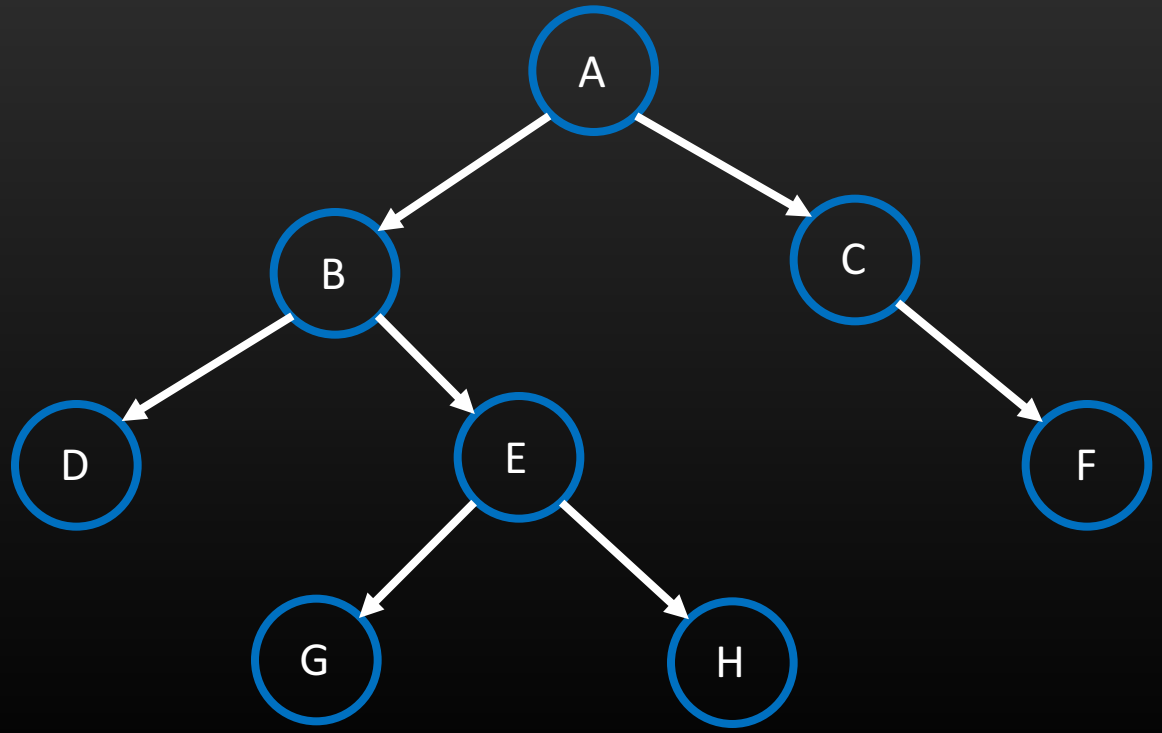
Depth First Search – In-order

D	B	G	E	H	A	C	F	



Depth First Search – In-order

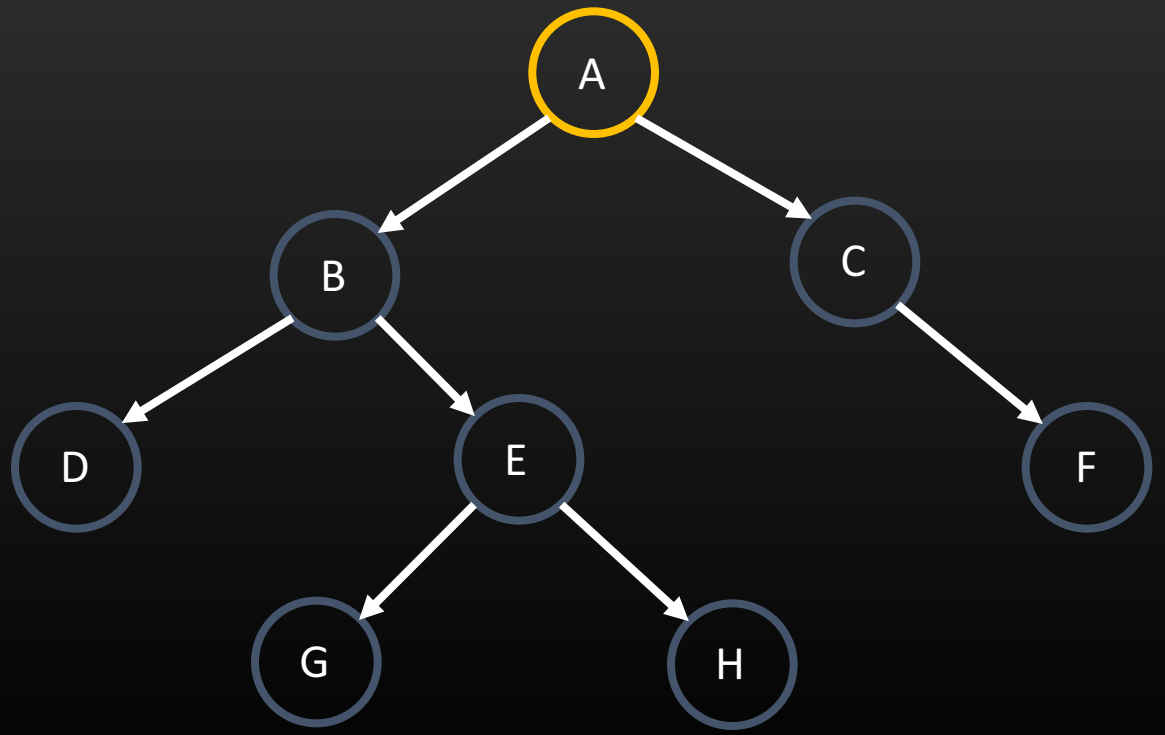
D	B	G	E	H	A	C	F	



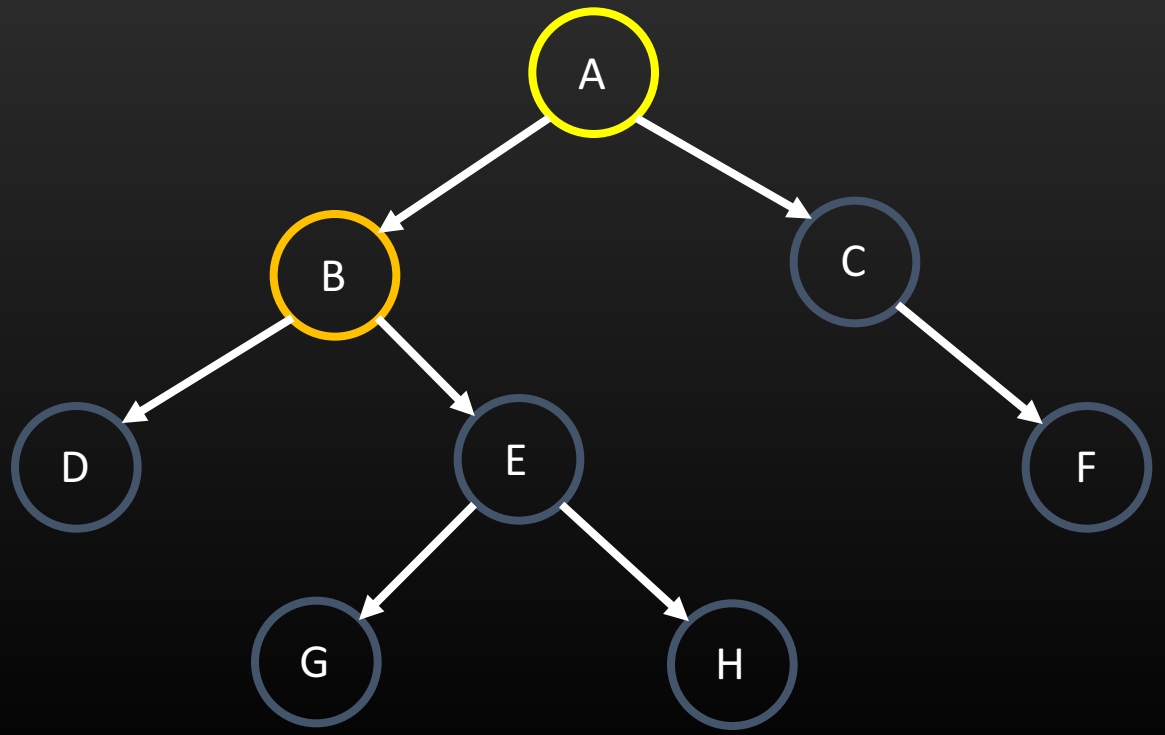
Depth First Search – Post-order

- Starting from the root
- For the traversal at node N:
  - Recursively traverse its left subtree
  - Recursively traverse its right subtree
  - Process the node N

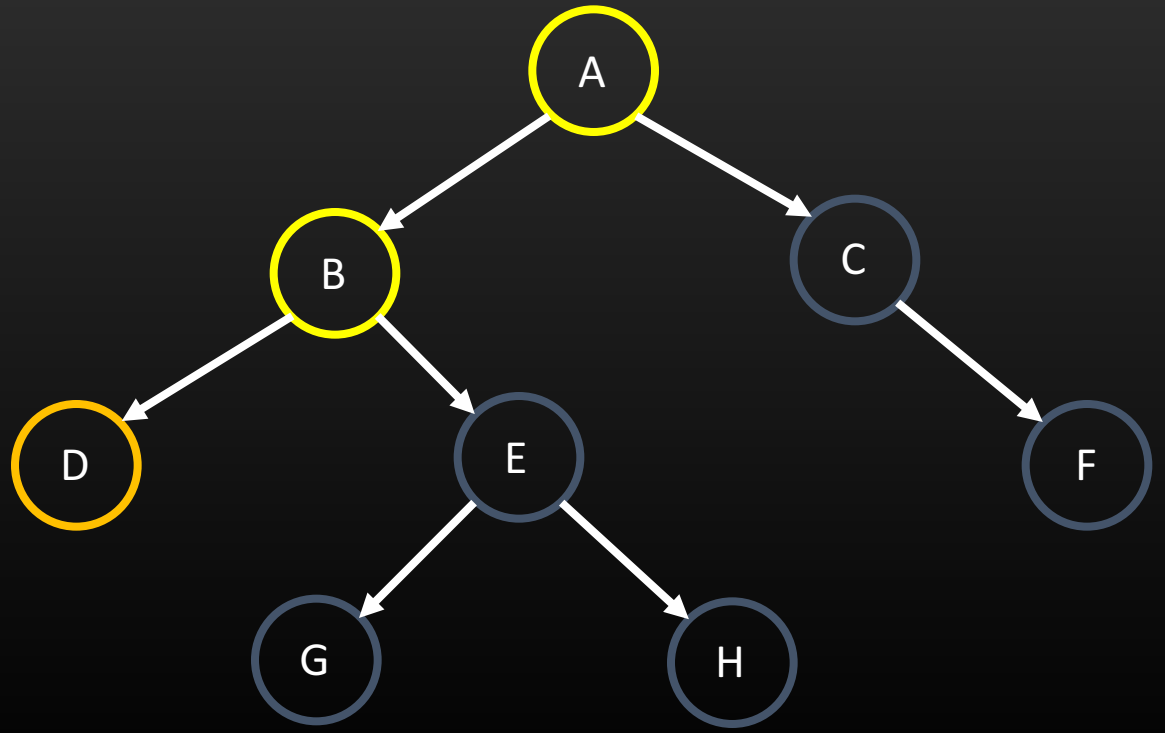
Depth First Search – Post-order



Depth First Search – Post-order

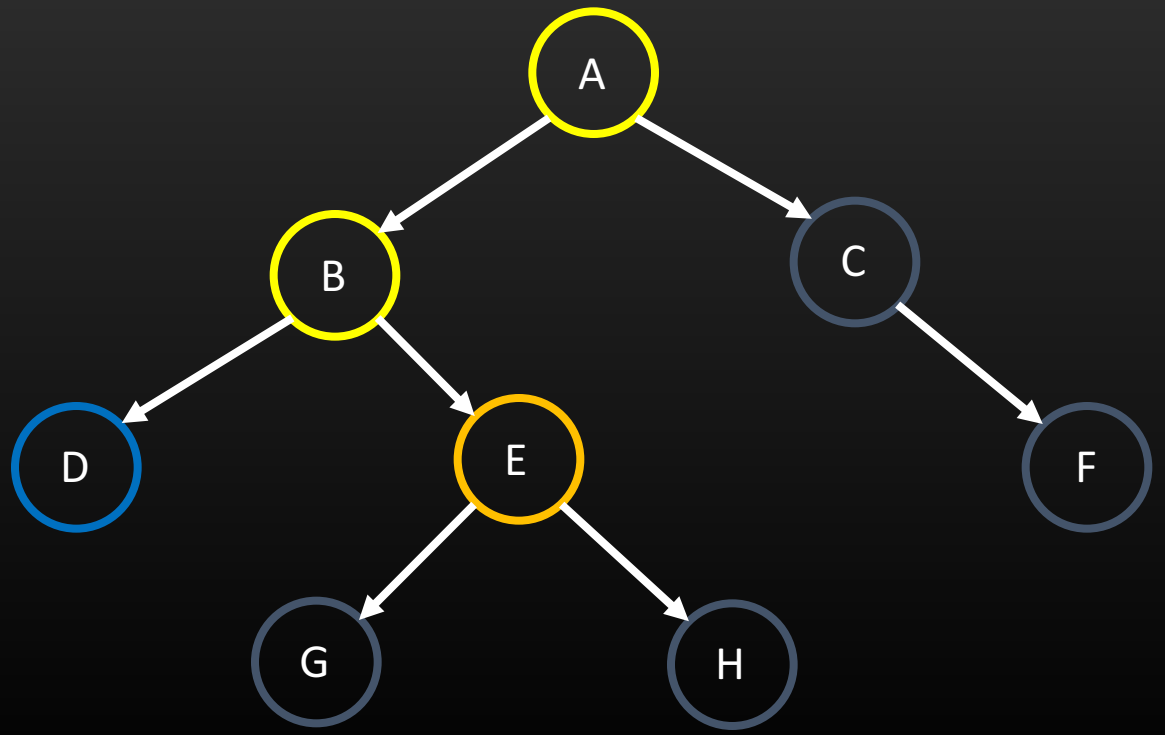


Depth First Search – Post-order



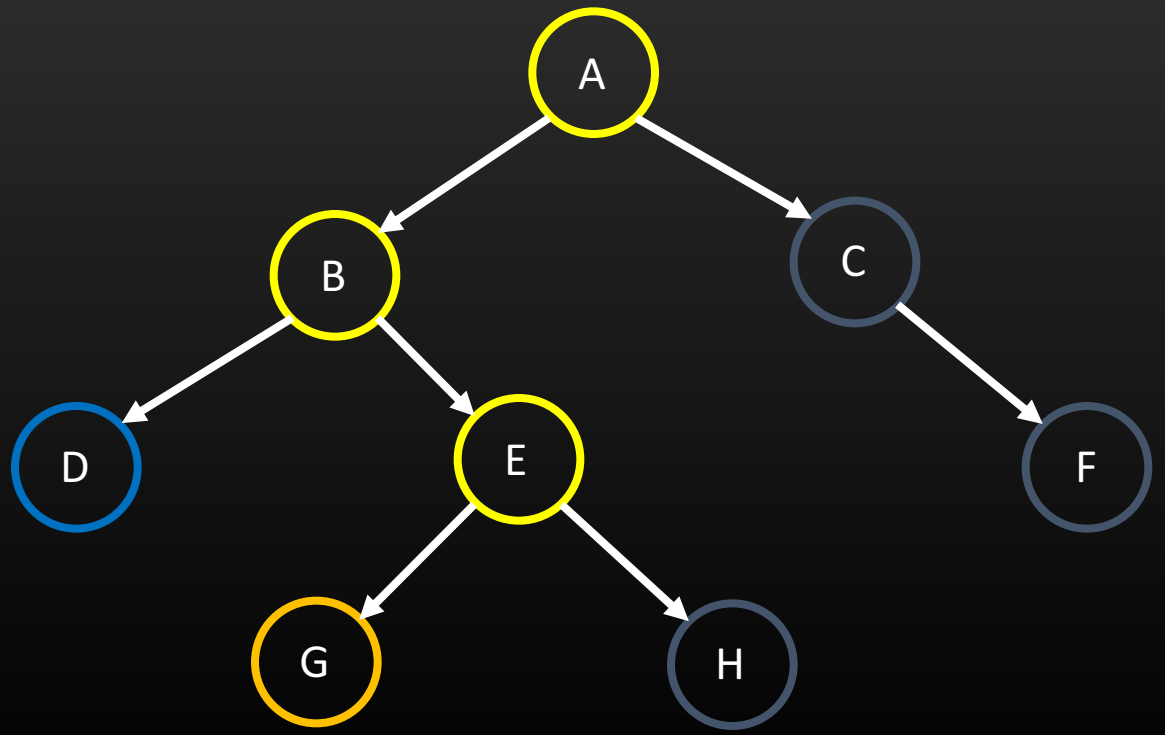
Depth First Search – Post-order

D								



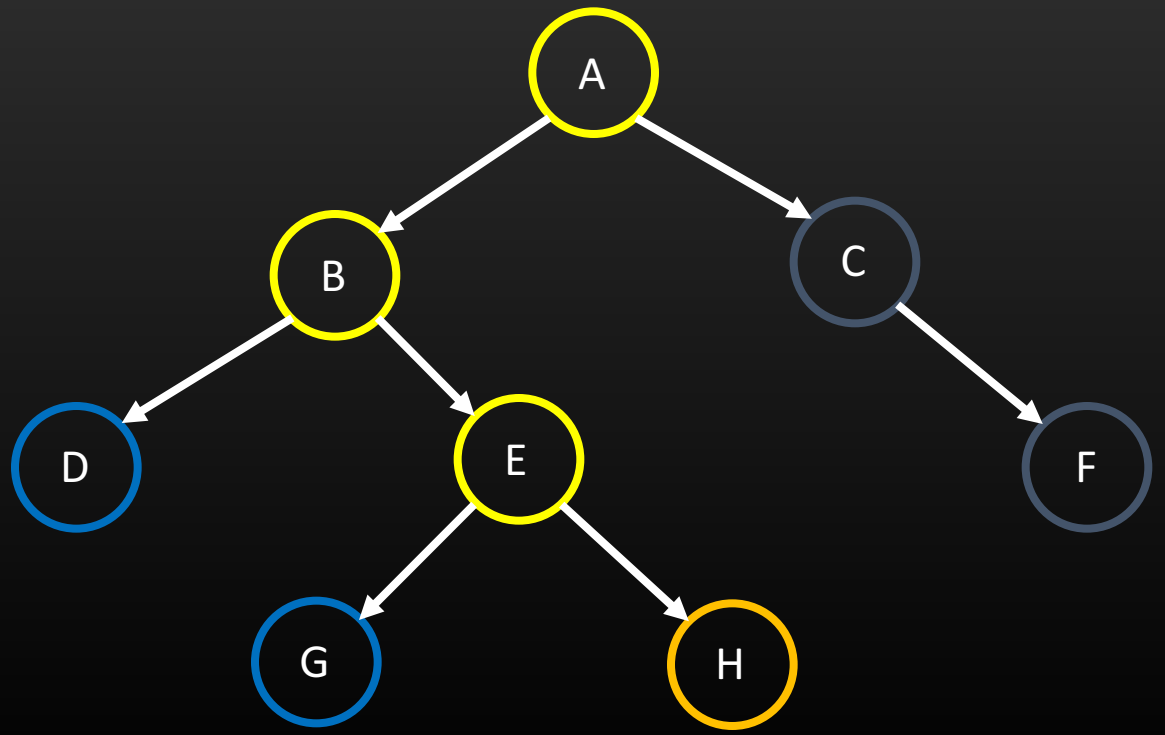
Depth First Search – Post-order

D	G							



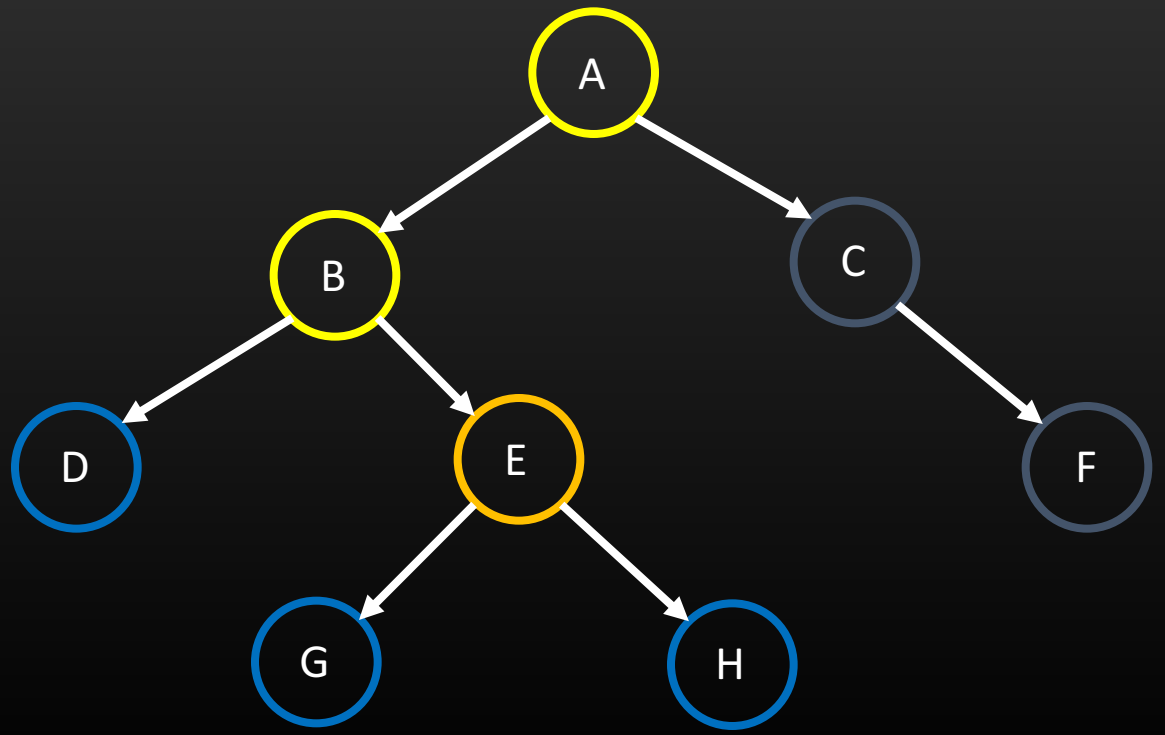
Depth First Search – Post-order

D	G	H						



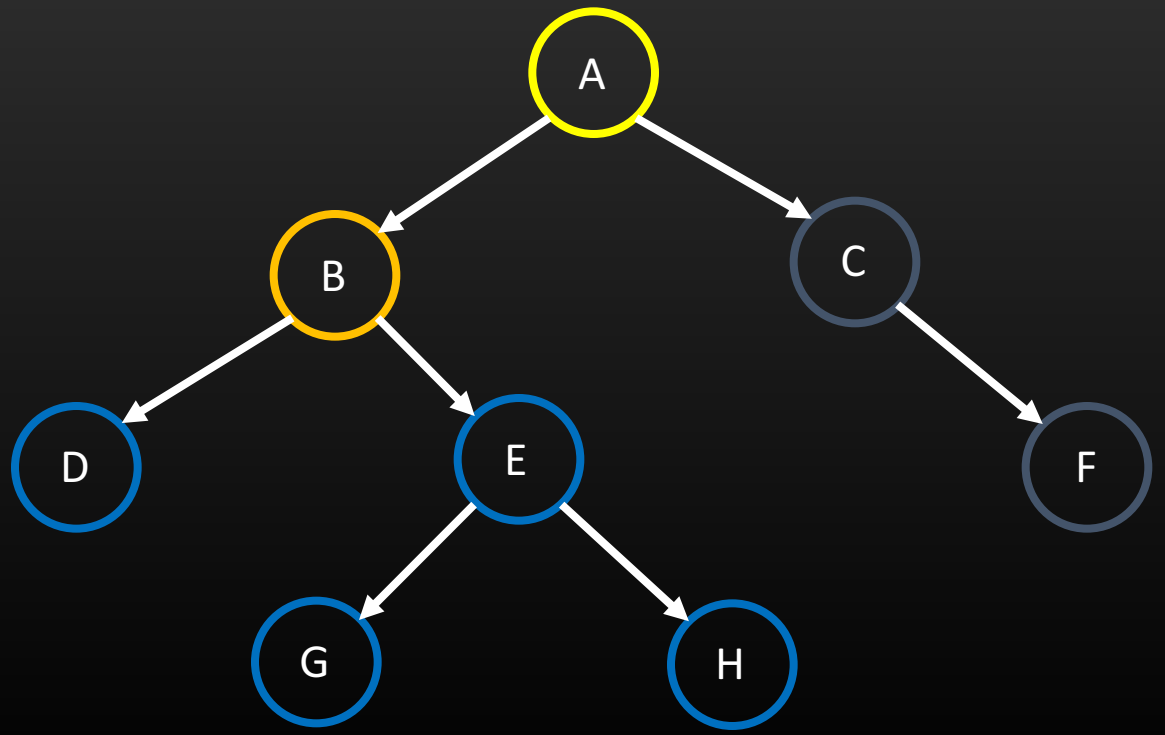
Depth First Search – Post-order

D	G	H	E					



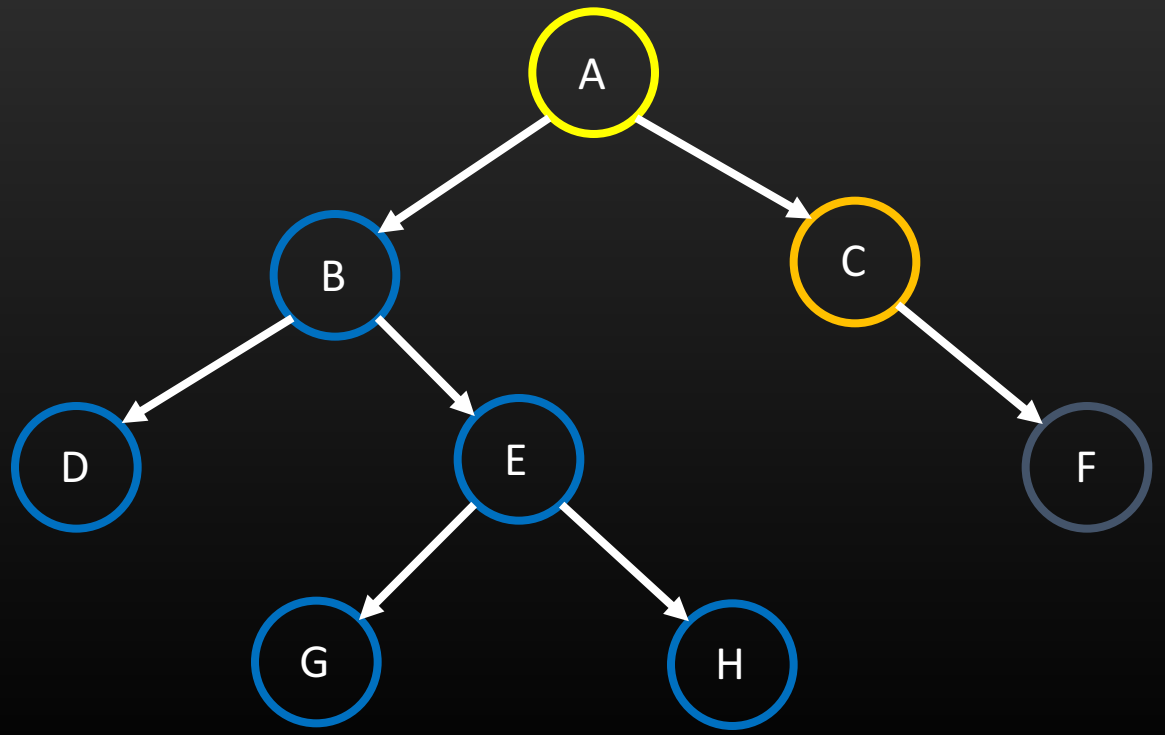
Depth First Search – Post-order

D	G	H	E	B				



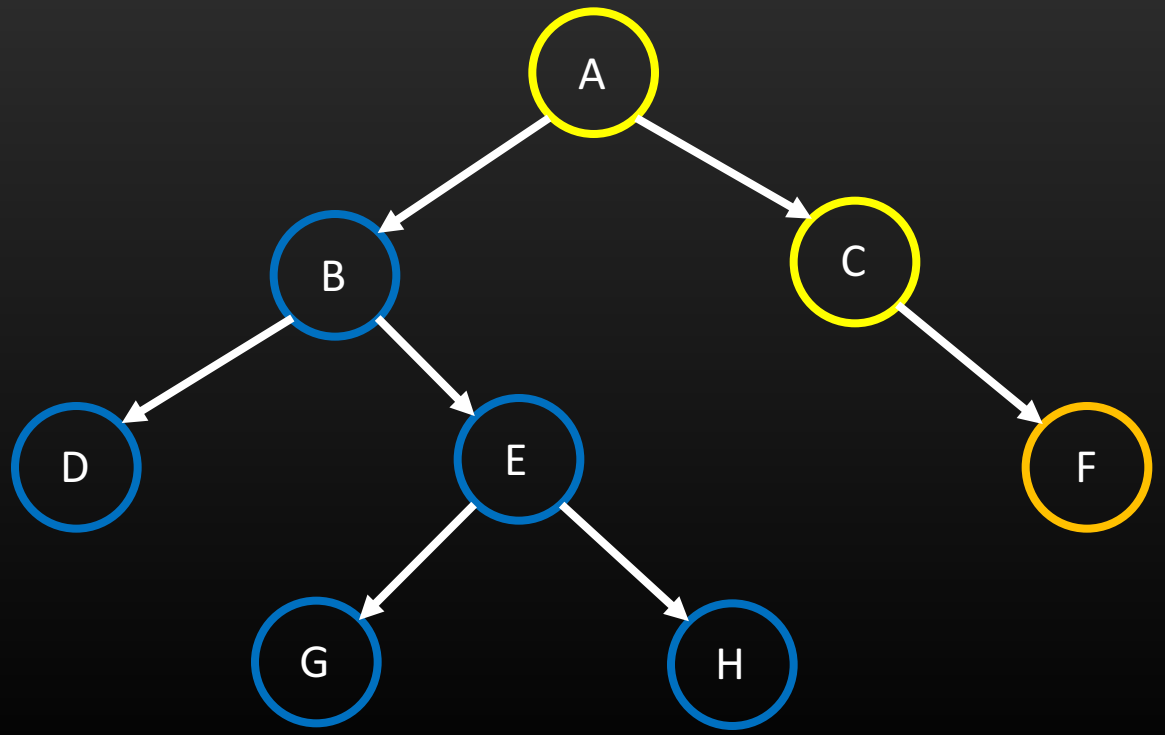
Depth First Search – Post-order

D	G	H	E	B				



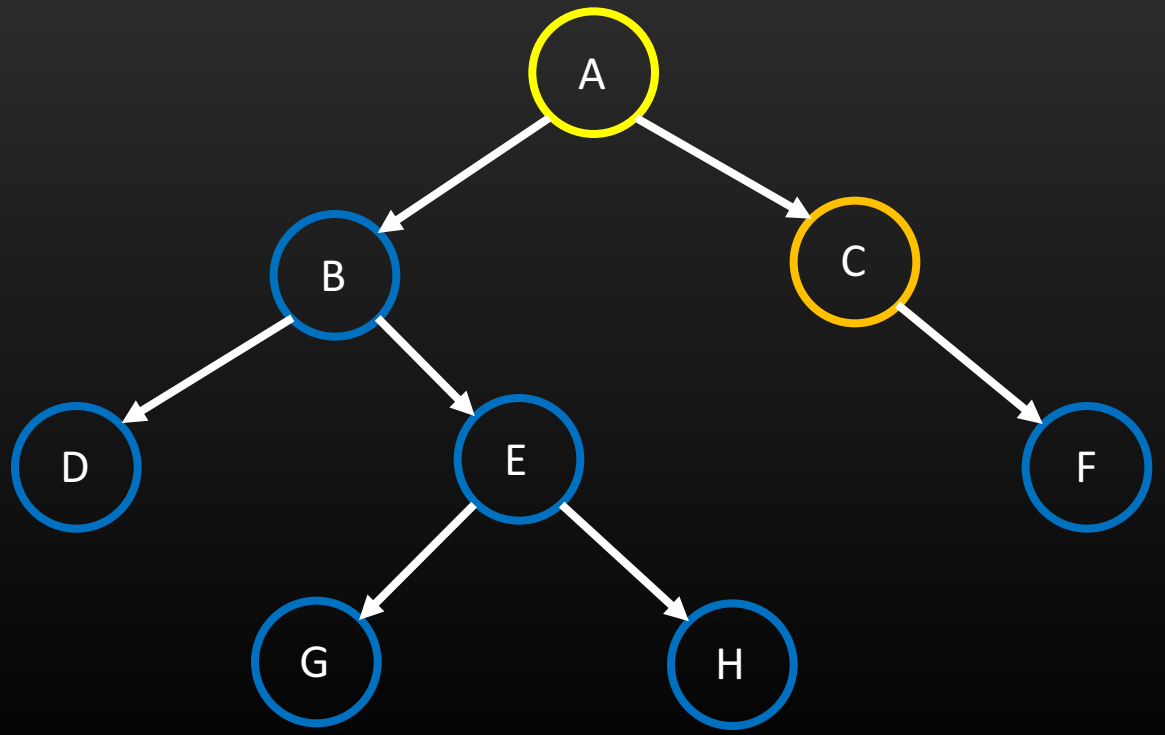
Depth First Search – Post-order

D	G	H	E	B	F			



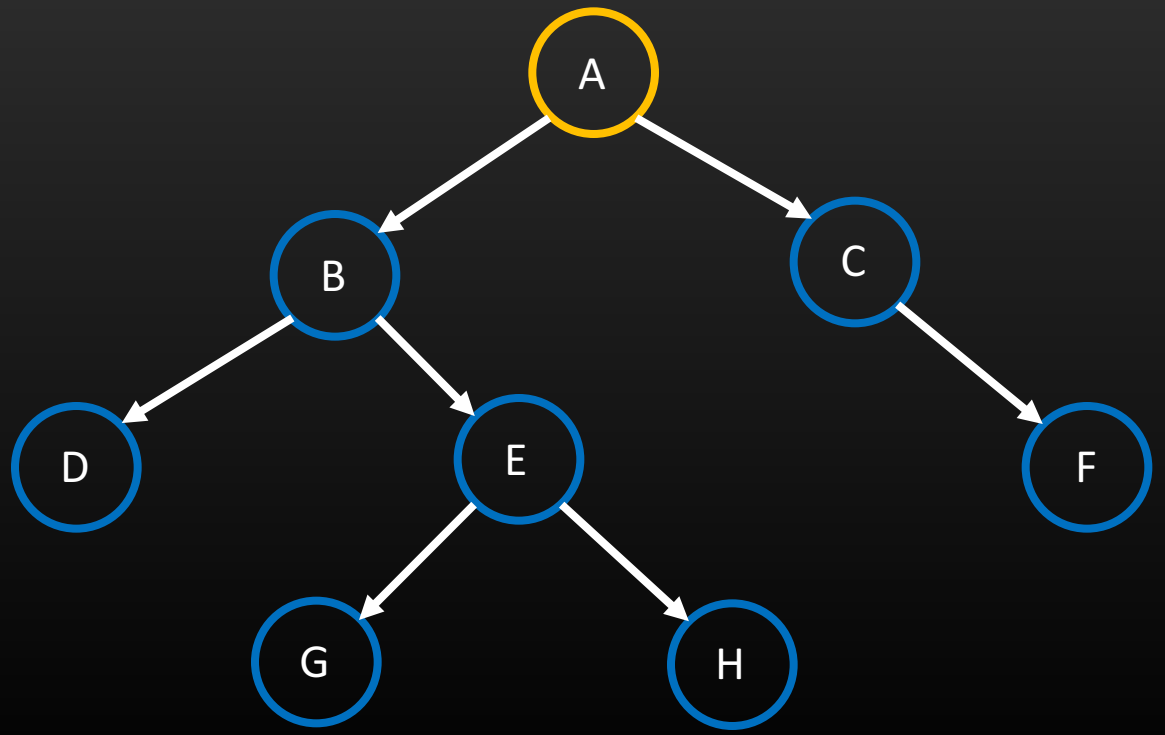
Depth First Search – Post-order

D	G	H	E	B	F	C		



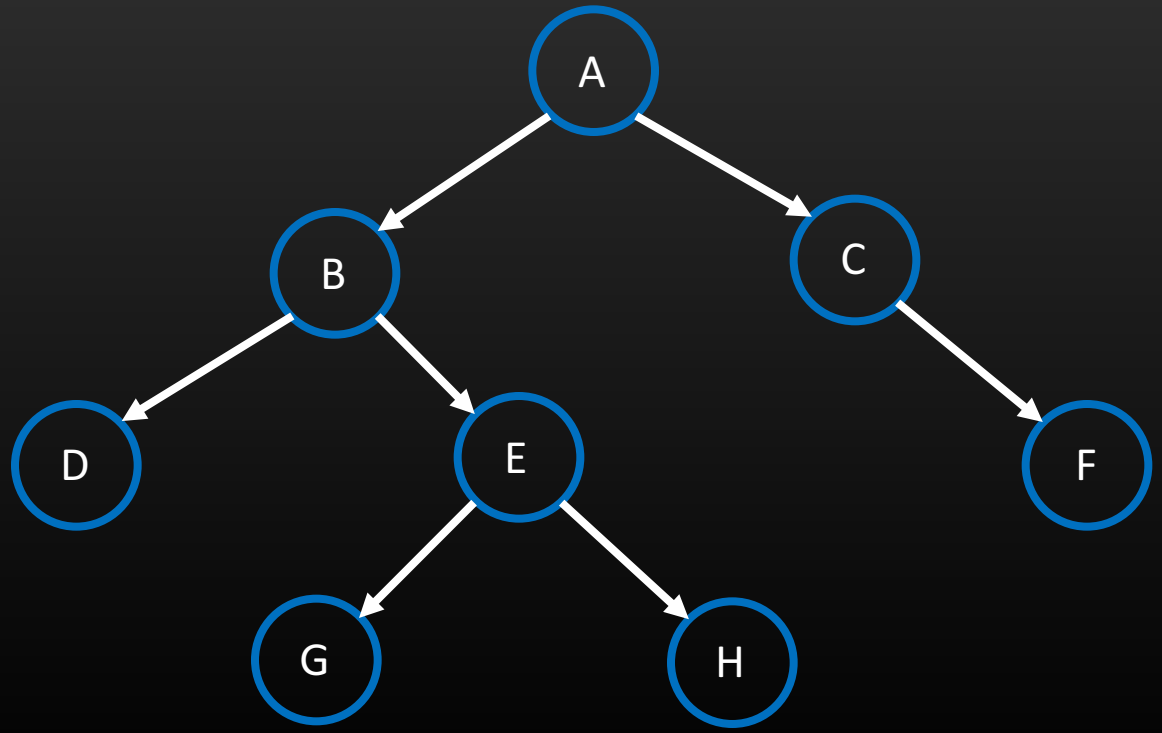
Depth First Search – Post-order

D	G	H	E	B	F	C	A	



Depth First Search – Post-order

D	G	H	E	B	F	C	A	



Depth First Search

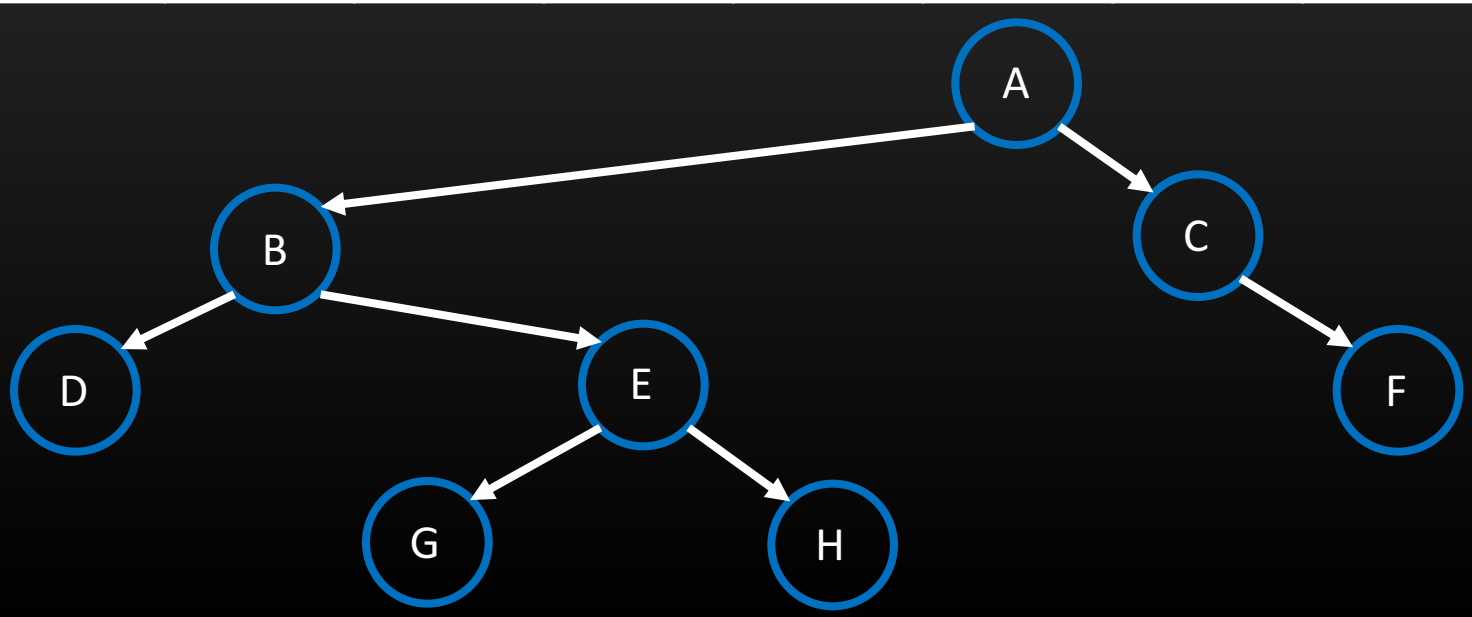
- Given either pre-order or post-order paired with in-order, the structure of the binary tree can be known

Pre-order	A	B	D	E	G	H	C	F
In-order	D	B	G	E	H	A	C	F

Depth First Search

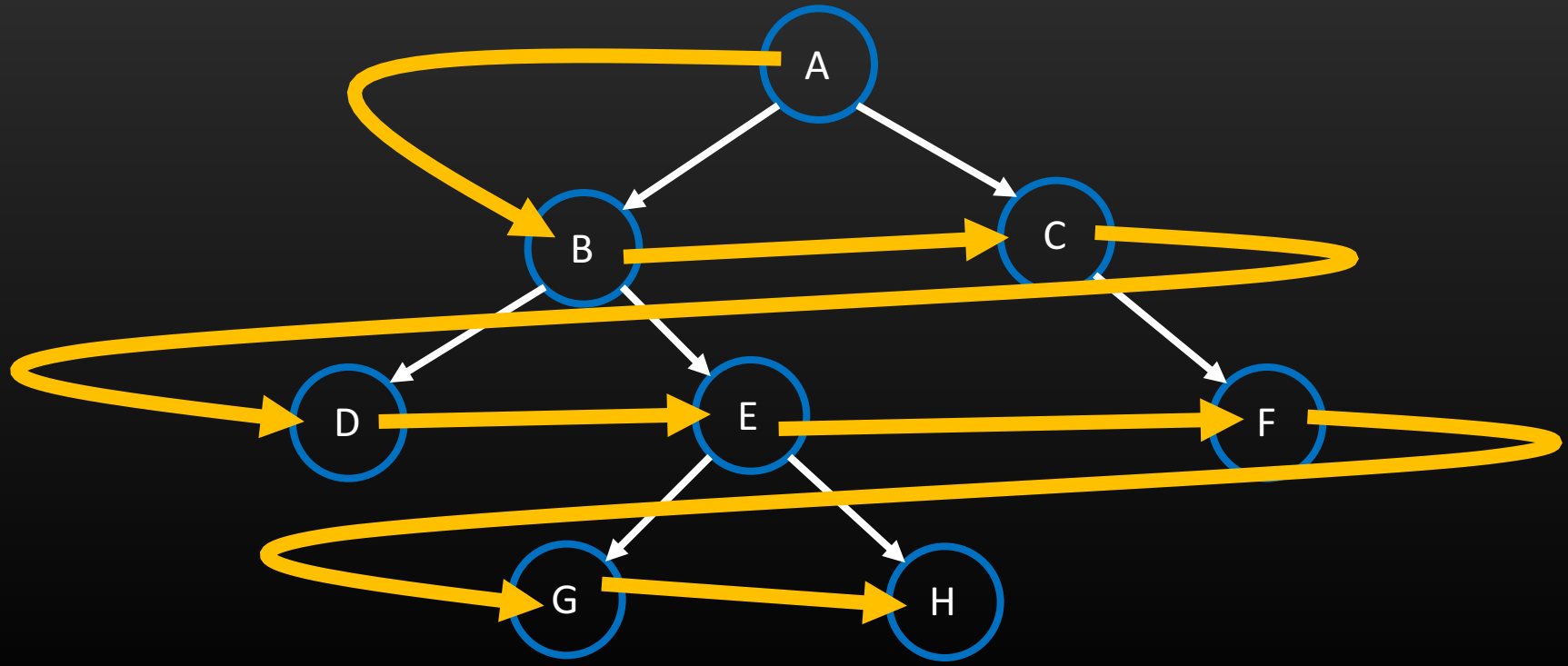
- Given either pre-order or post-order paired with in-order, the structure of the binary tree can be known

Pre-order	A	B	D	E	G	H	C	F
In-order	D	B	G	E	H	A	C	F



Breadth First Search

- Starts at the root
- Explores all children first before moving to the next level

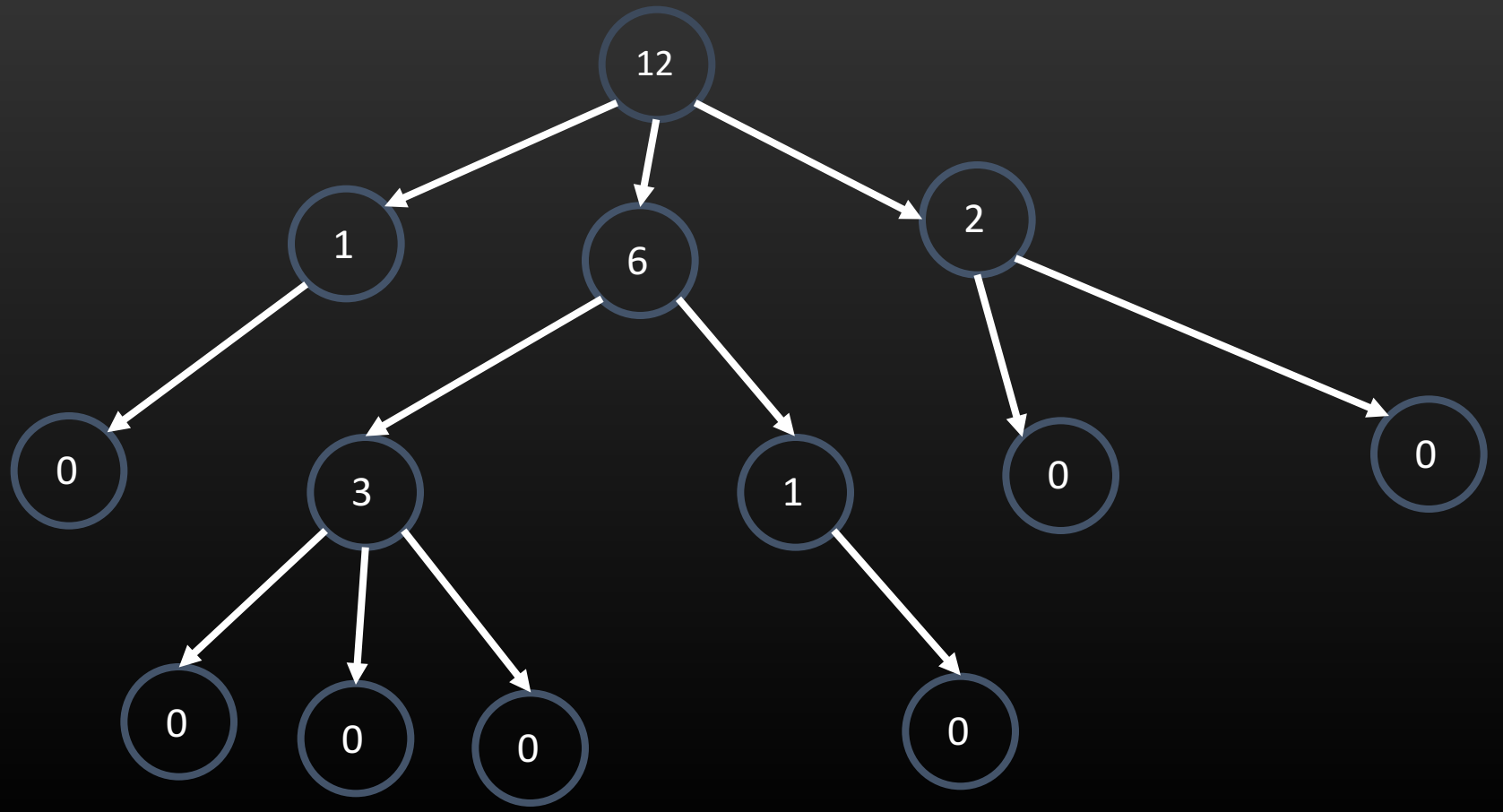


Breadth First Search

```
Procedure bfs(){
    push the root into the queue
    while the queue is not empty do
        t = front of the queue
        process node t
        for all w in children of t do
            push w into the queue
        pop t from the queue
}
```

Example 1

- Given a rooted tree, find the number of descendants of each node

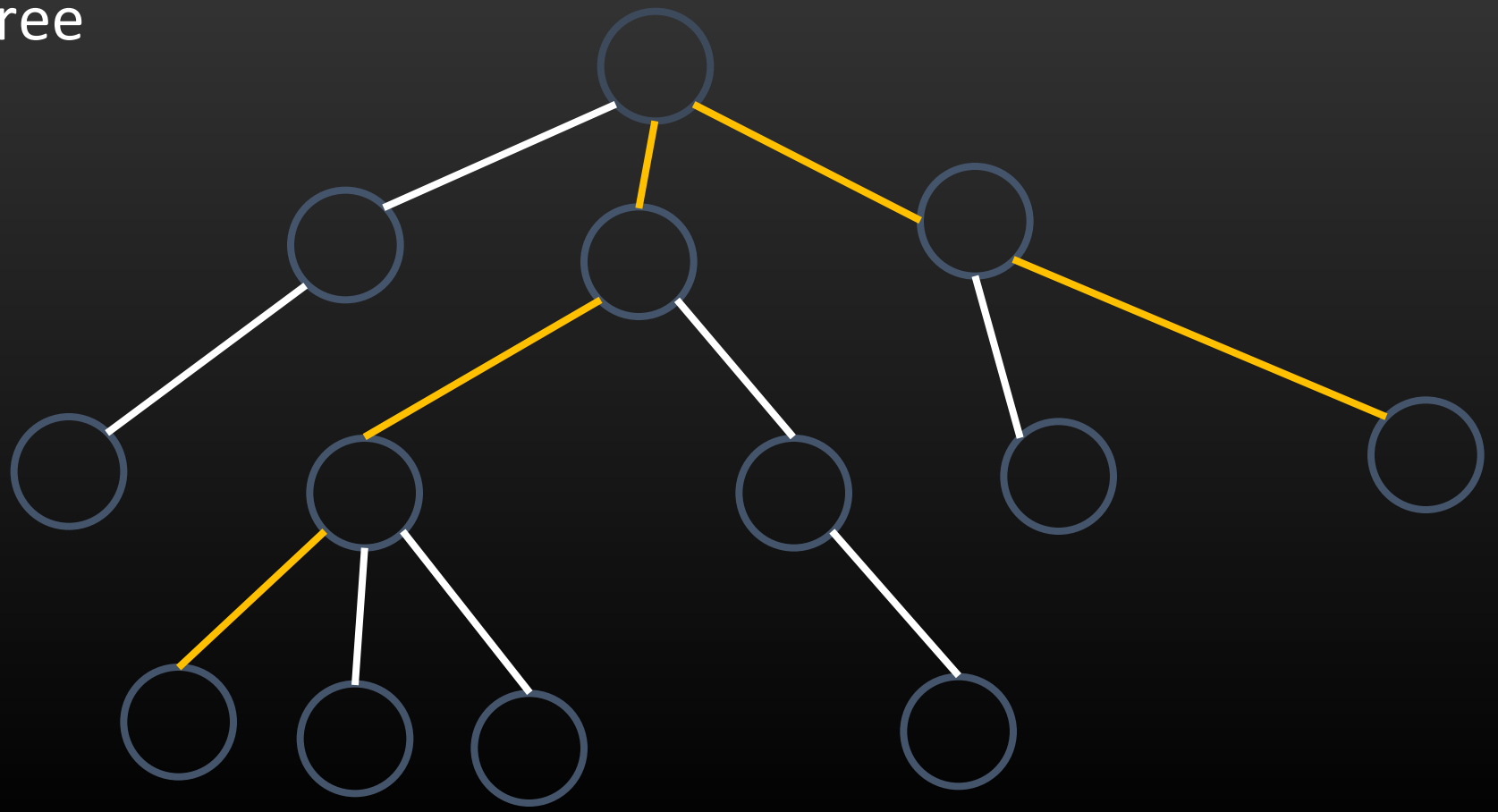


Example 1

- To find the number of descendants of node N, find the number of descendants of all of its children first
- The answer will be the sum of the answer of its children + number of its children
- Depth First Search using Post-order

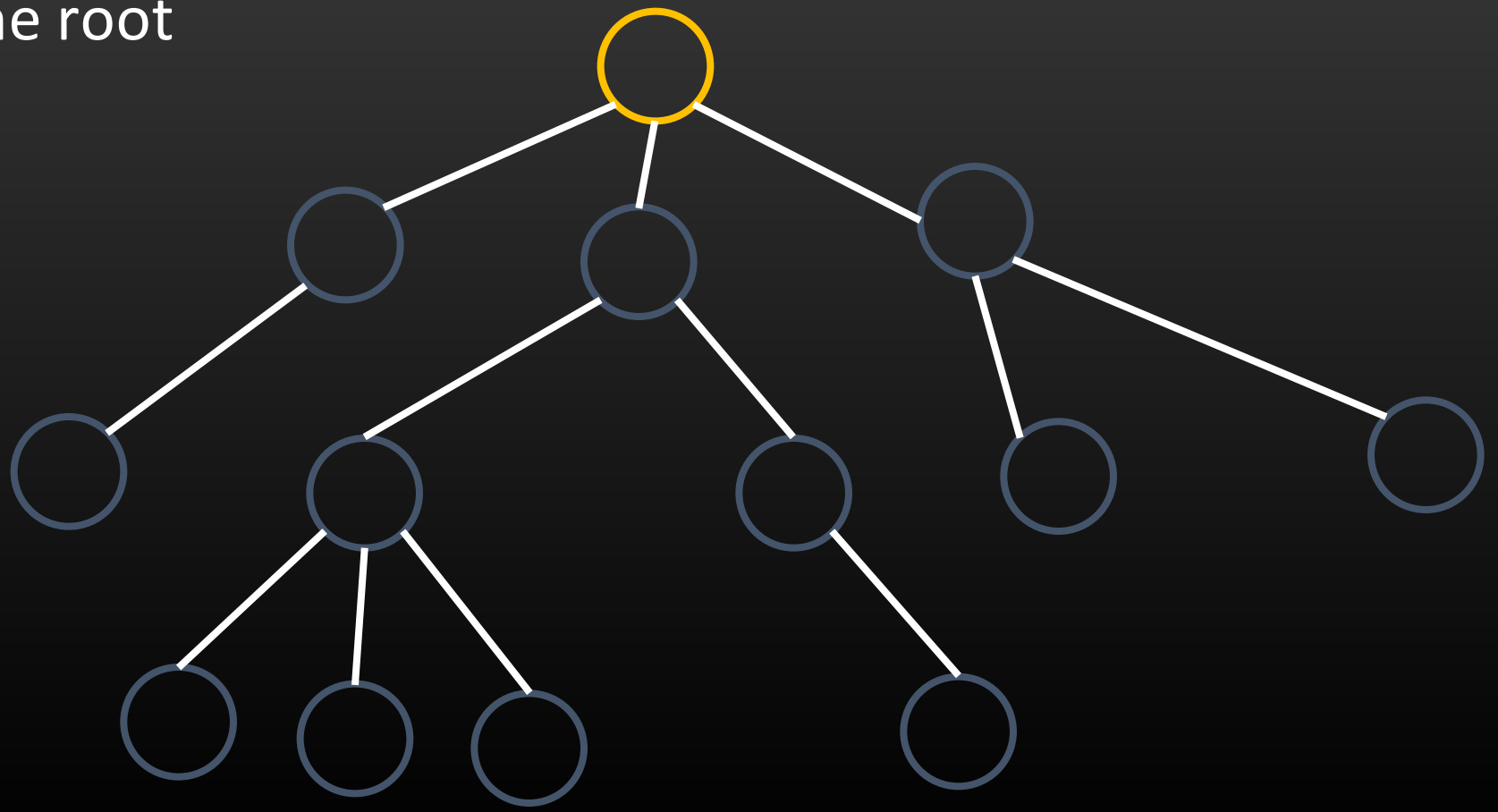
Example 2

- Given a tree, find the diameter of the tree, i.e. the longest distance in the tree



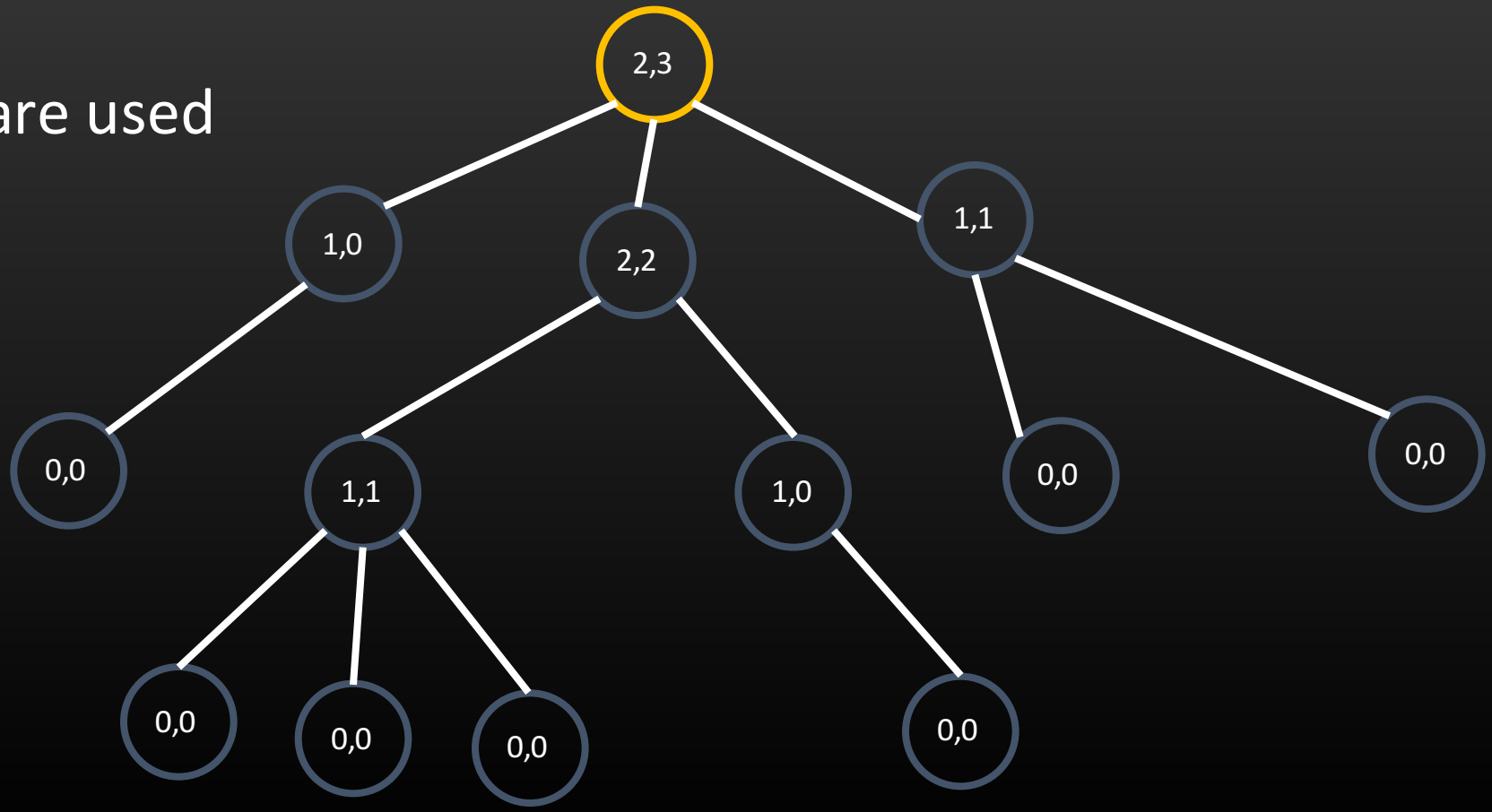
Example 2

- Although the root is not specified, we can randomly choose a node to be the root



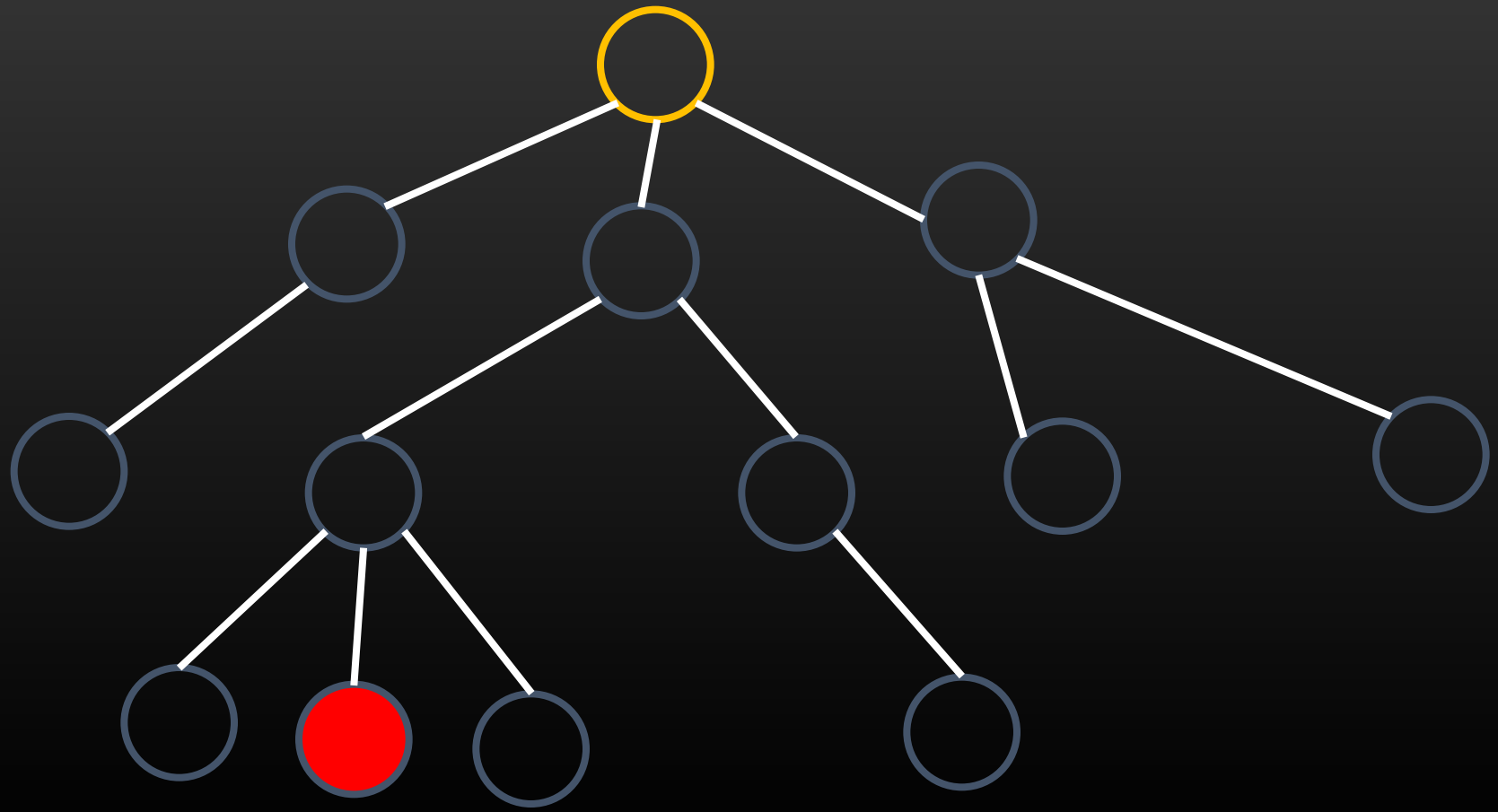
Example 2

- For each vertex, find the longest and second-longest distance from its descendants
- Children’s answers are used
- Depth First Search



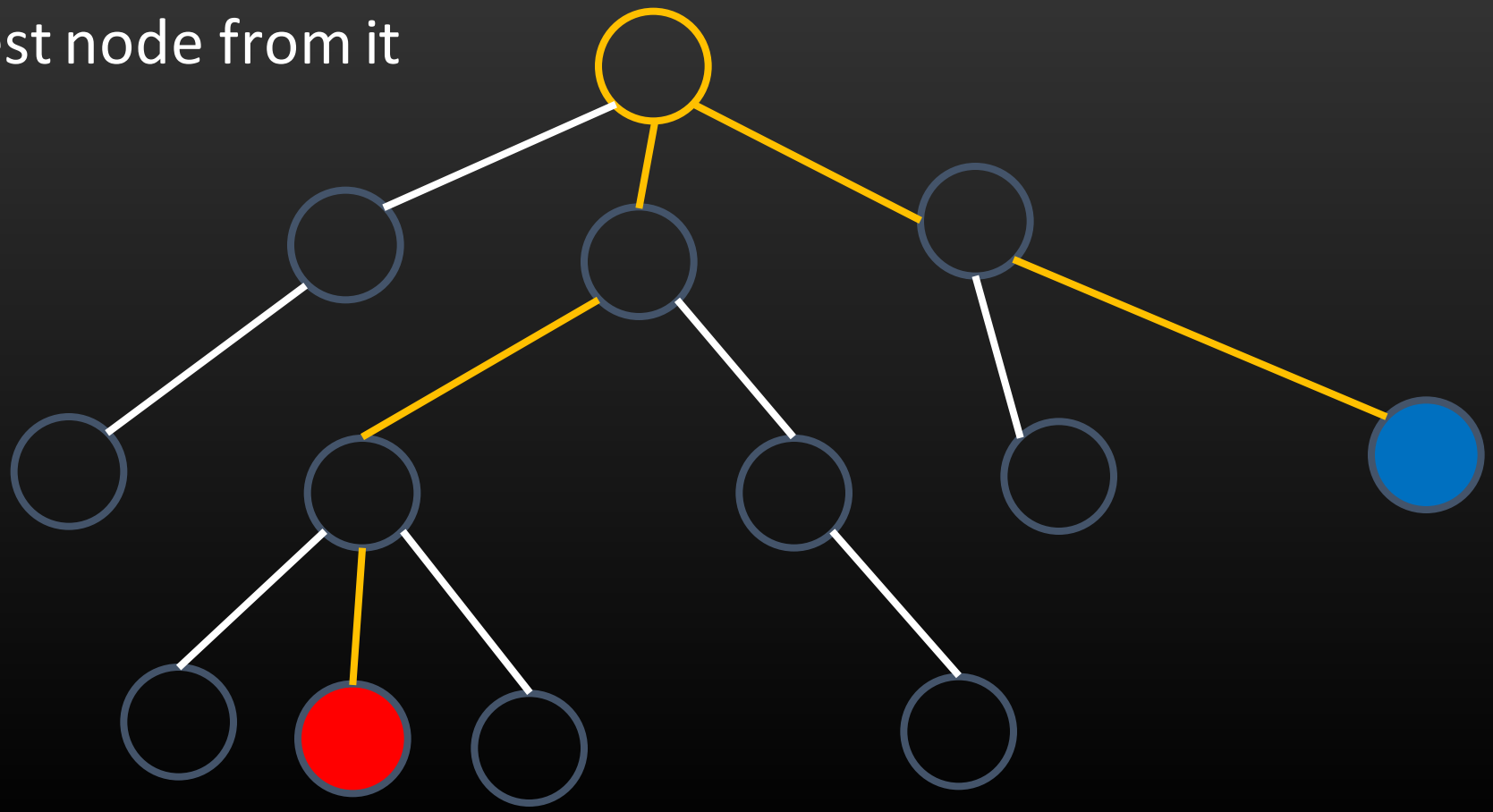
Example 2

- Or we can find the deepest node from the root first (using DFS)



Example 2

- Or we can find the deepest node from the root first (using DFS)
- Then find the deepest node from it

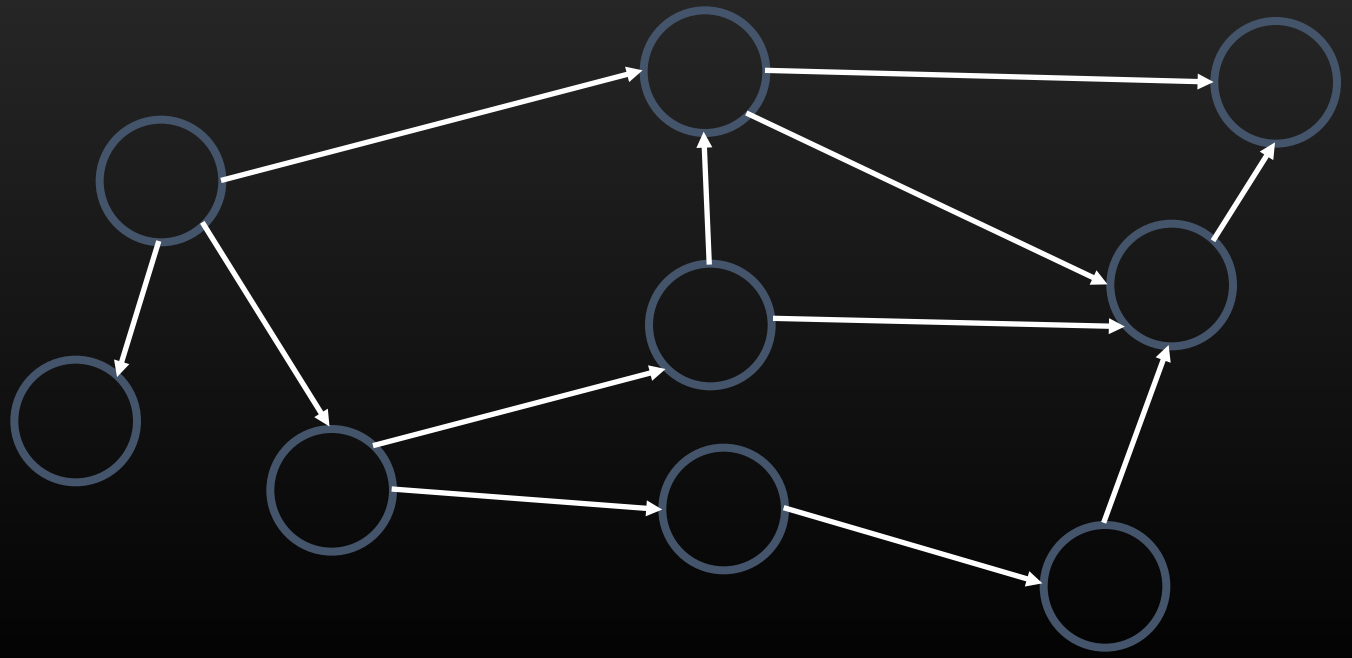


More usage

- See **Dynamic Programming II**

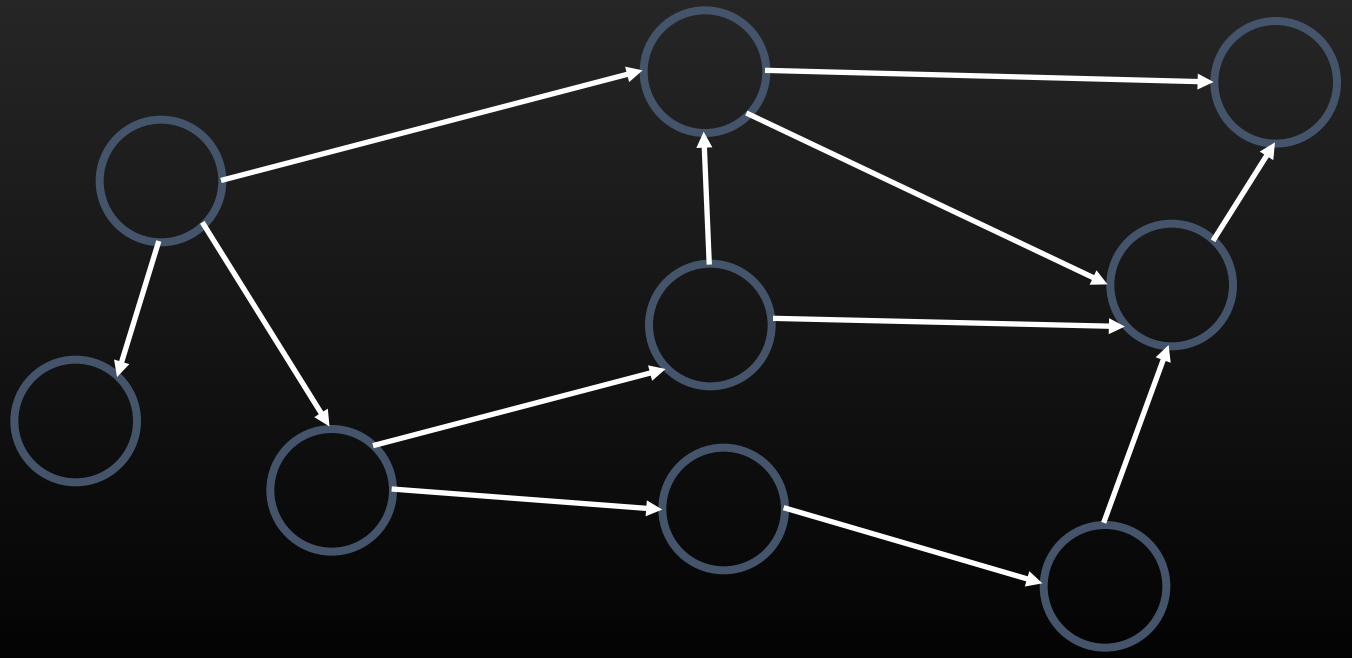
Directed Acyclic Graph

- Other than trees, some directed graph may have no cycles as well
- We call them Directed Acyclic Graphs (DAG)



Directed Acyclic Graph

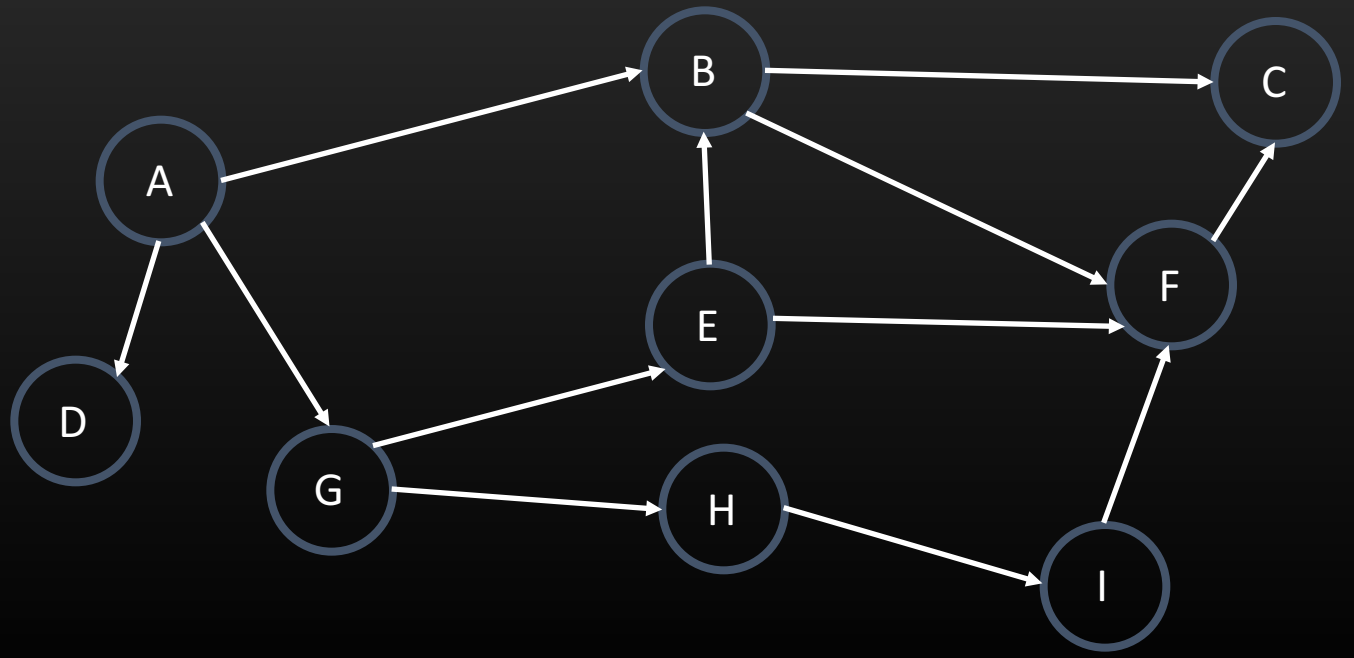
- We may want to process the nodes layer by layer
- Nodes having edges to others should be manipulated first



Topological Order

- An ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.
- For example, one of the topological orders of this graph is

  A D G E H B I F C



Topological Sort

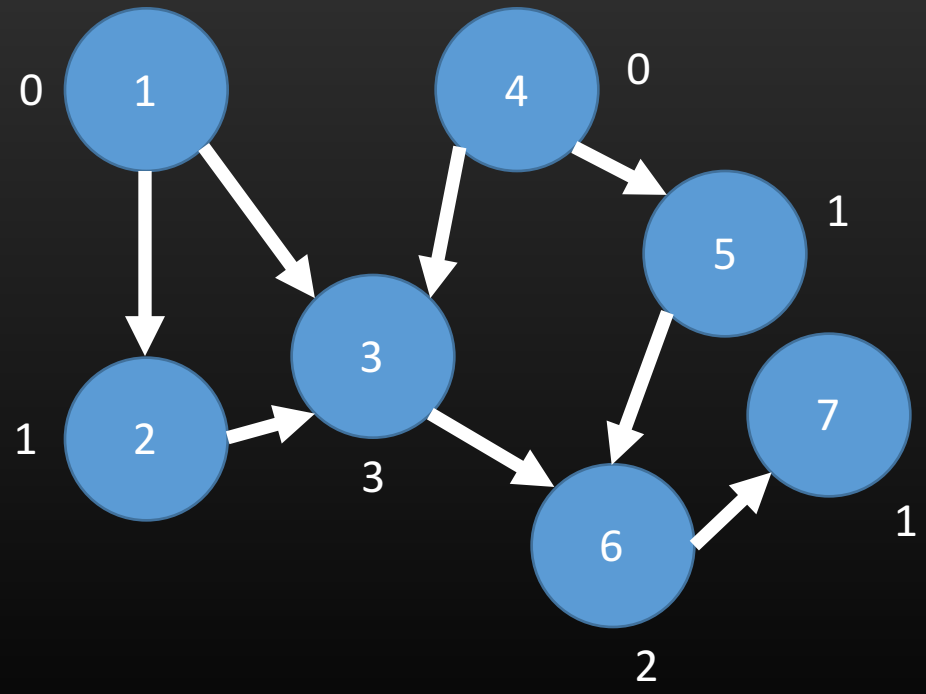
- Find a vertex with no incoming edges. Number it and remove all outgoing edges from it
- Repeat the above process
- Can be implemented by DFS or BFS

Topological Sort

- Define the indegree of a vertex v as the number of edges connecting to v
- Instead of removing the vertices and edges, we can manipulate the indegree of all vertices
- When a vertex is removed, lower the indegree of the nodes connected by it by 1

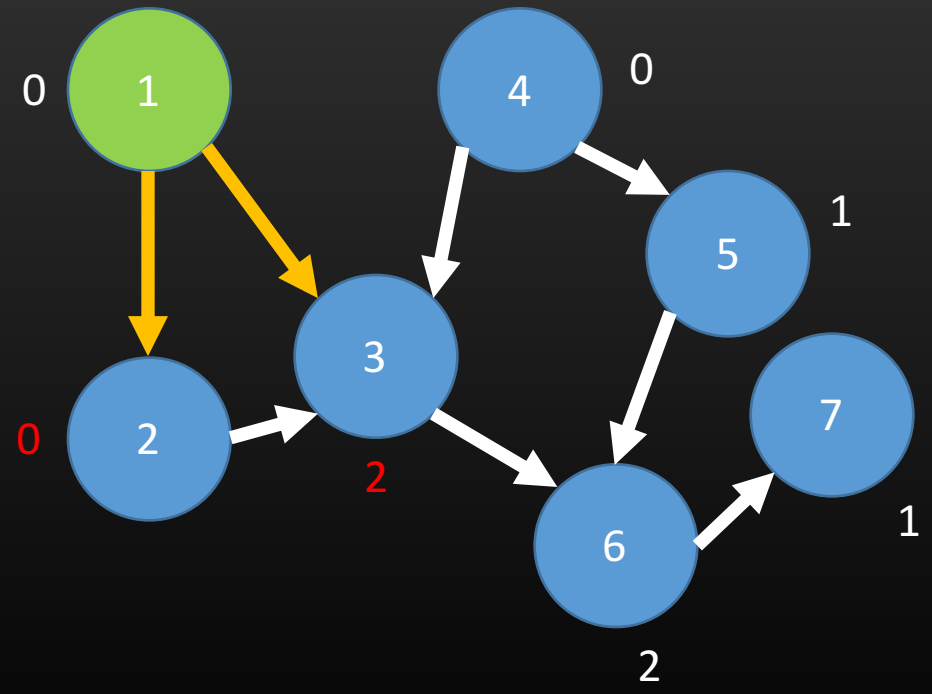
Topological Sort

- Topological order:



Topological Sort

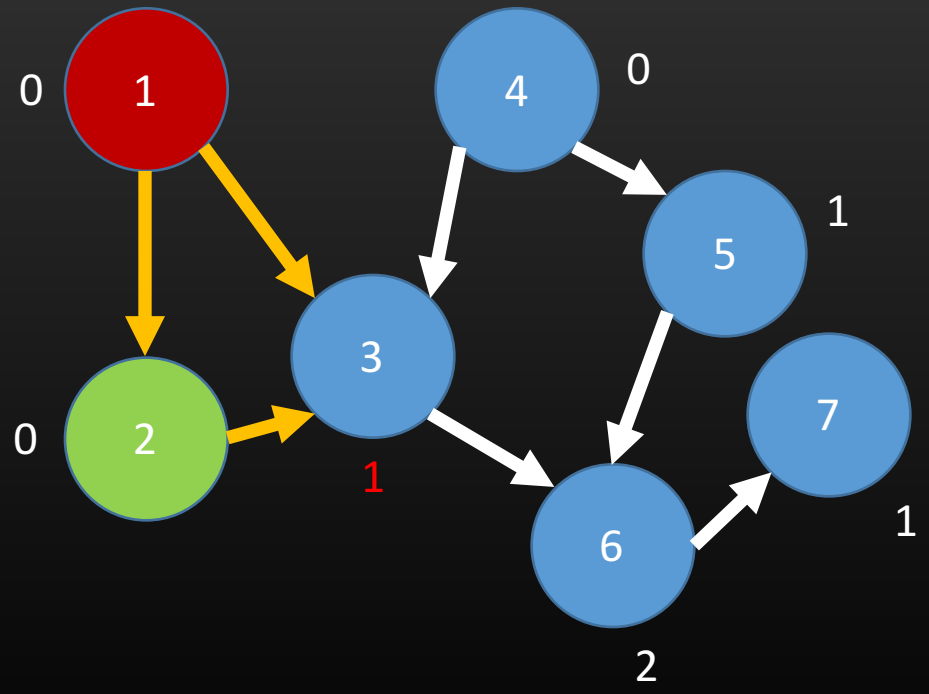
- Topological order:



Topological Sort

- Topological order:

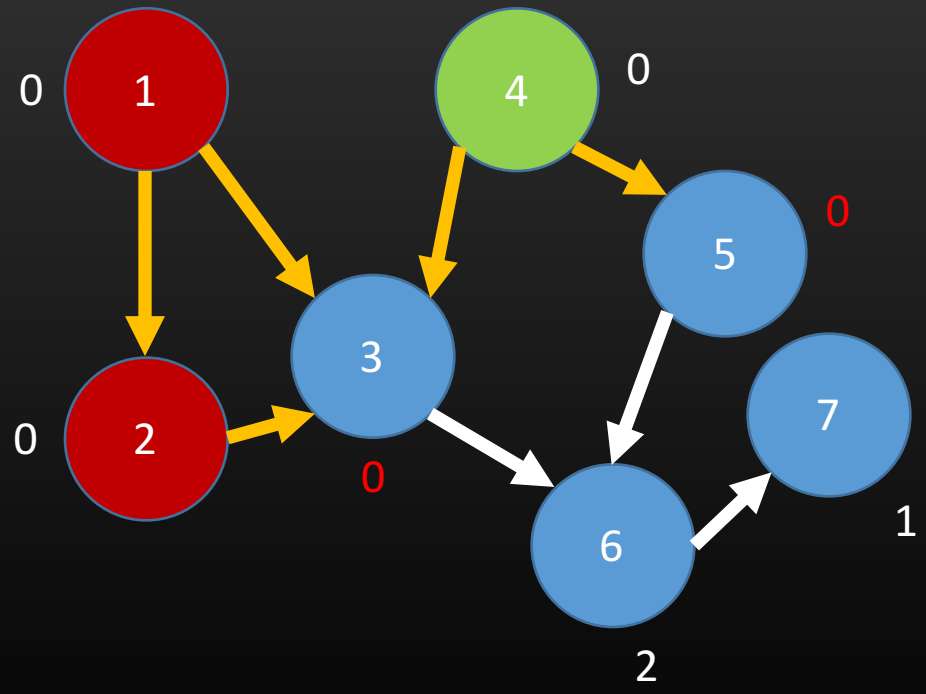
1	2					



Topological Sort

- Topological order:

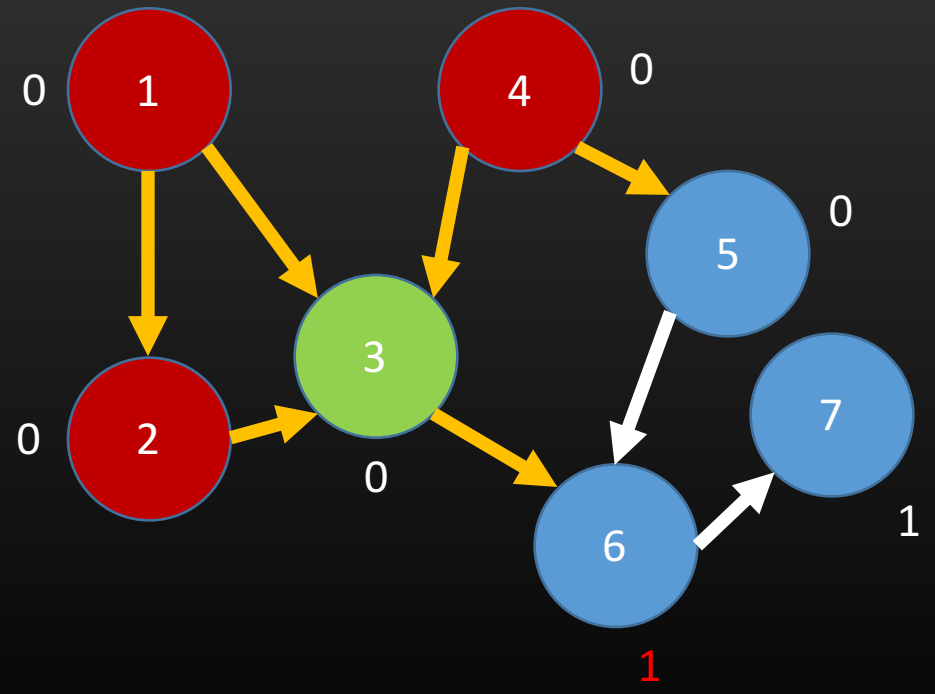
1	2	4				



Topological Sort

- Topological order:

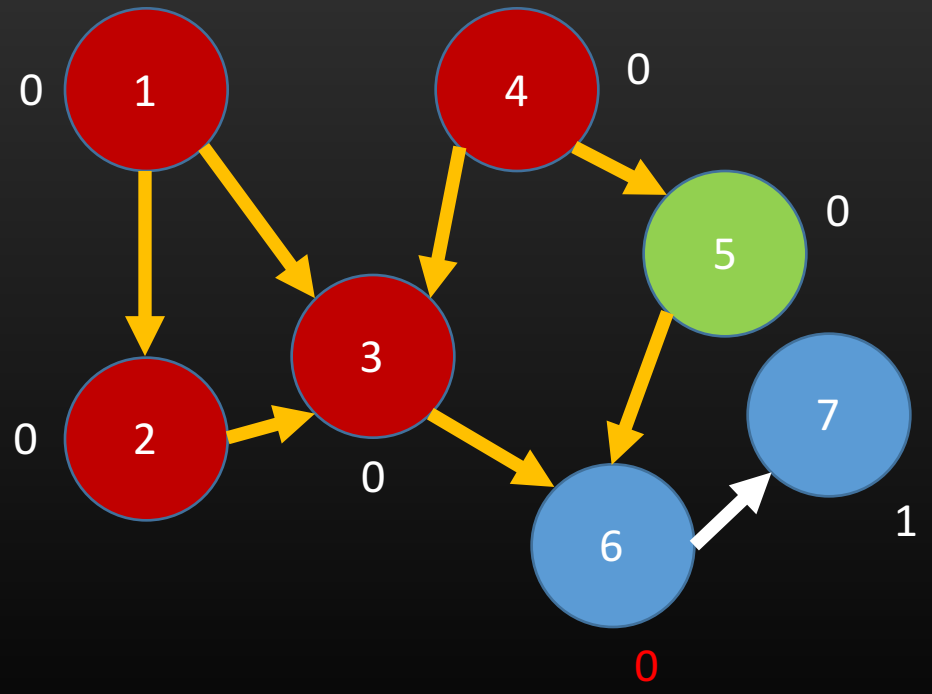
1	2	4	3			



Topological Sort

- Topological order:

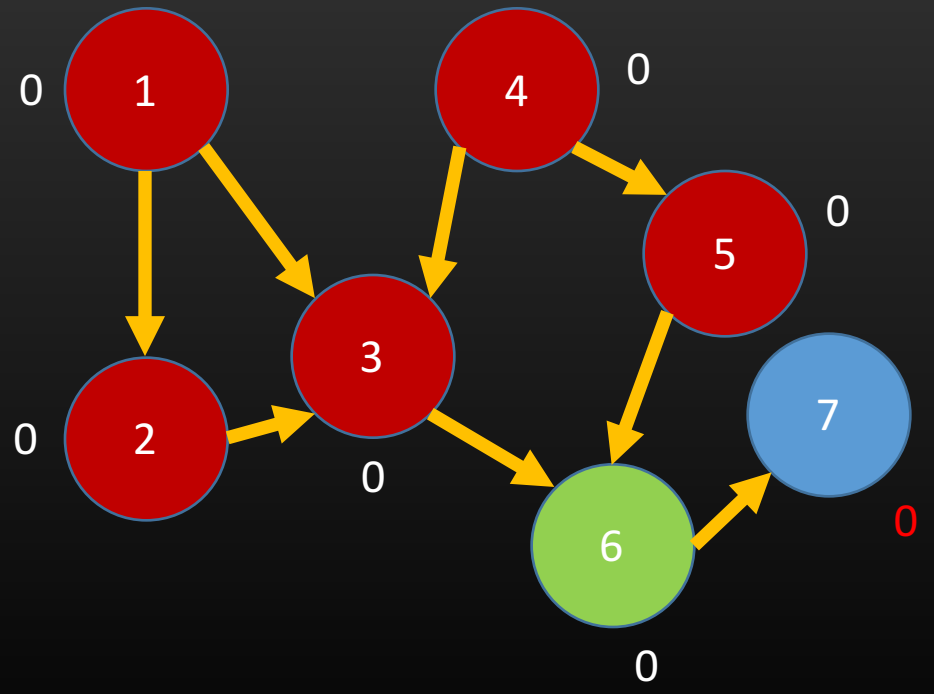
1	2	4	3	5		



Topological Sort

- Topological order:

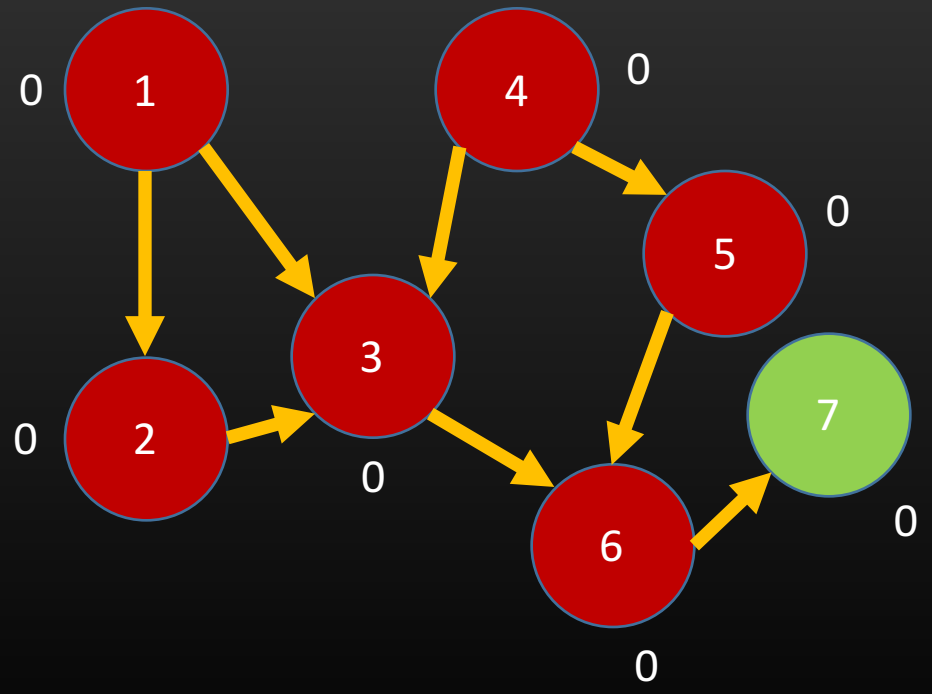
1	2	4	3	5	6	



Topological Sort

- Topological order:

1	2	4	3	5	6	7



Topological Sort

```
Procedure Tsort(vertex v){
        put v into the order
        for all edge from v to u do
                indegree[u] = indegree[u] – 1
                if indegree[u] = 0 then
                        Tsort(u)
}
```

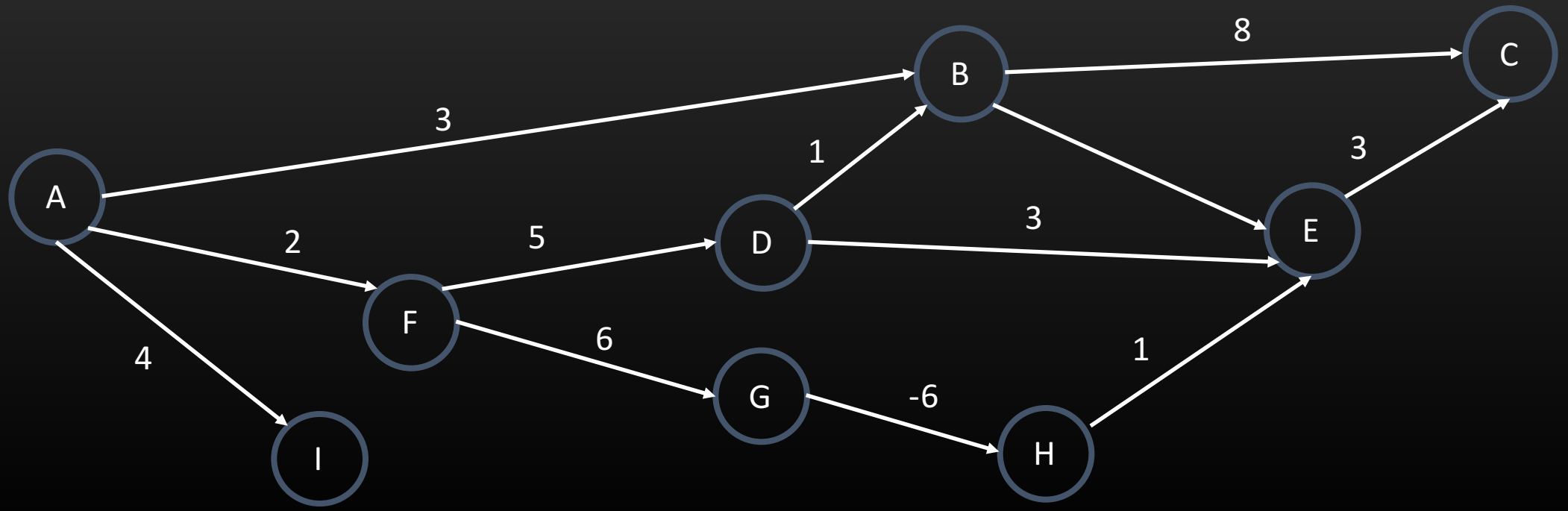
Topological Sort

- Time Complexity: $O(|V|^2)$ or $O(|V|+|E|)$ (same as DFS and BFS)

- Using the topological order, we can process the nodes correctly
- All information from previous layers can be retrieved by next layer

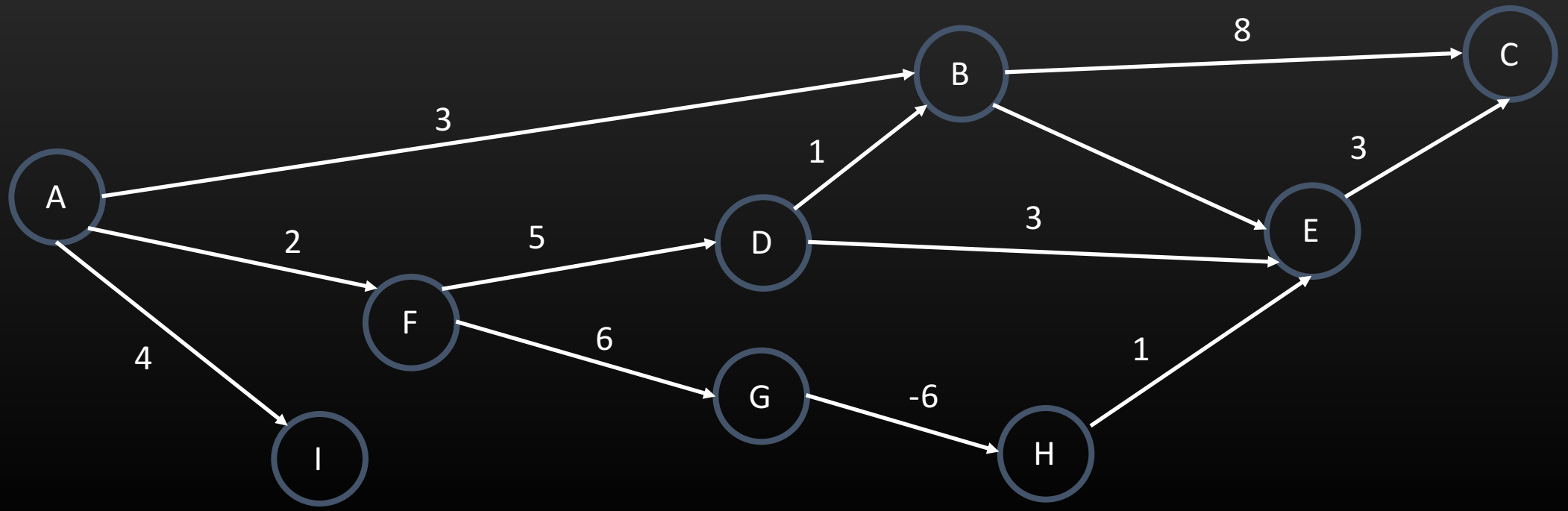
Example

- Given a DAG, find the shortest path from one node to the other



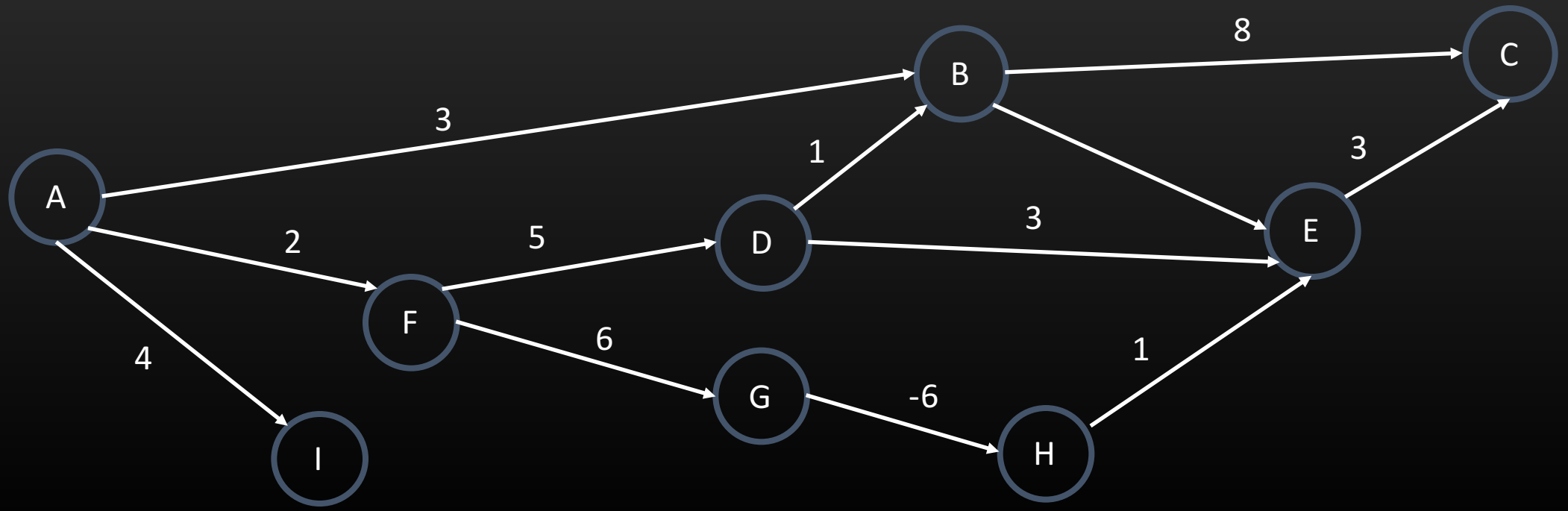
Example

- Find the topological order first: AFIDGBHEC



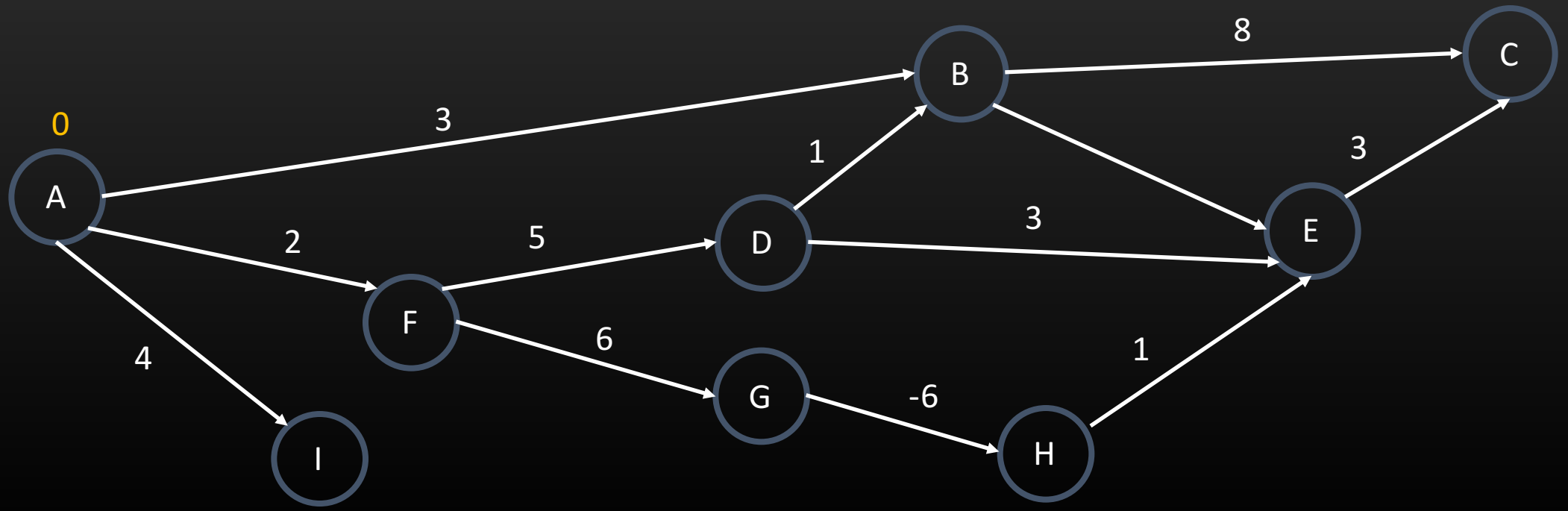
Example

- Find the topological order first: AFIDGBHEC
- Calculate the shortest paths based on the topological order



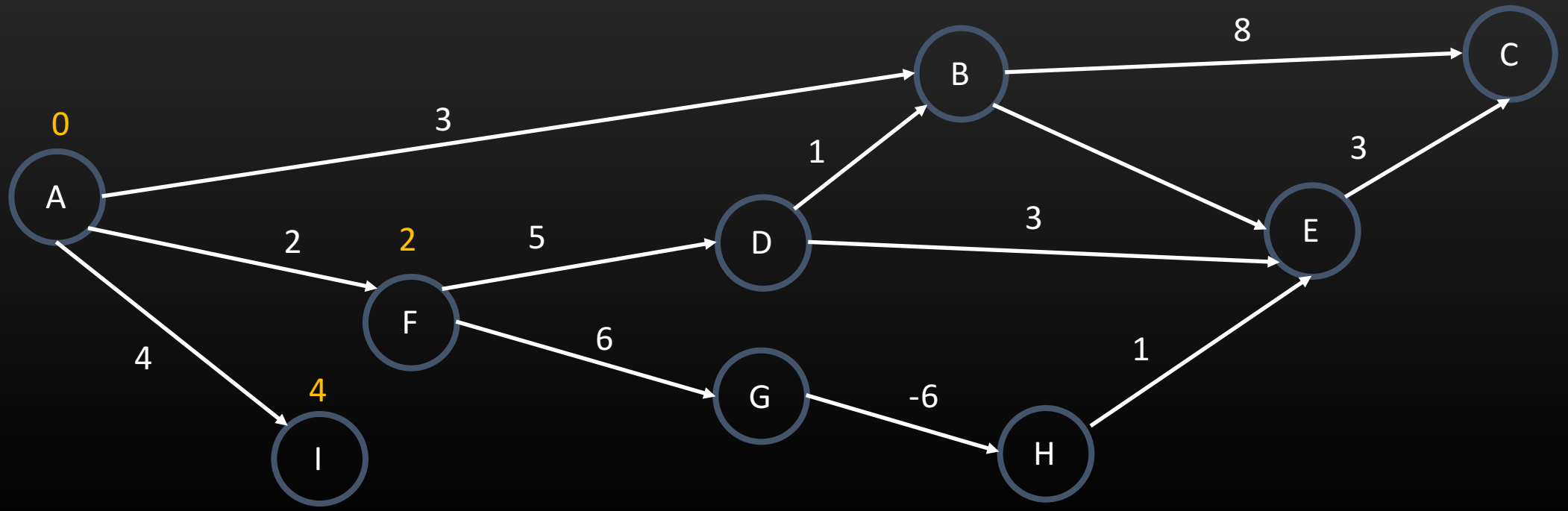
Example

- Find the topological order first: AFIDGBHEC
- Calculate the shortest paths based on the topological order



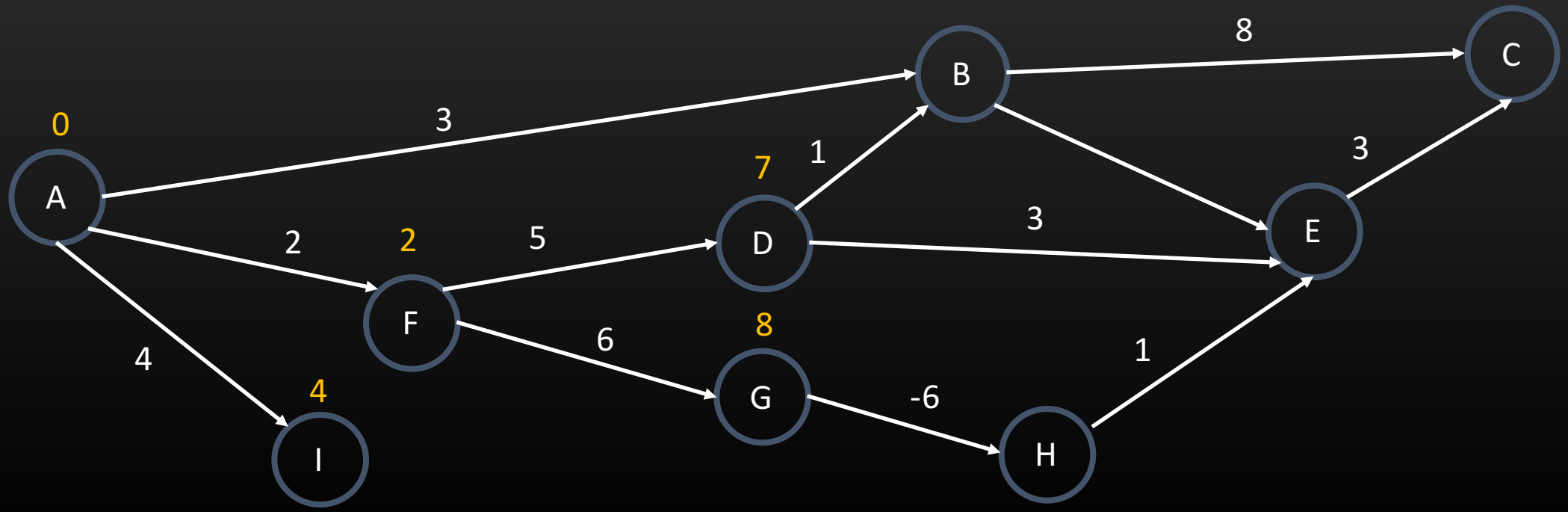
Example

- Find the topological order first: AFIDGBHEC
- Calculate the shortest paths based on the topological order



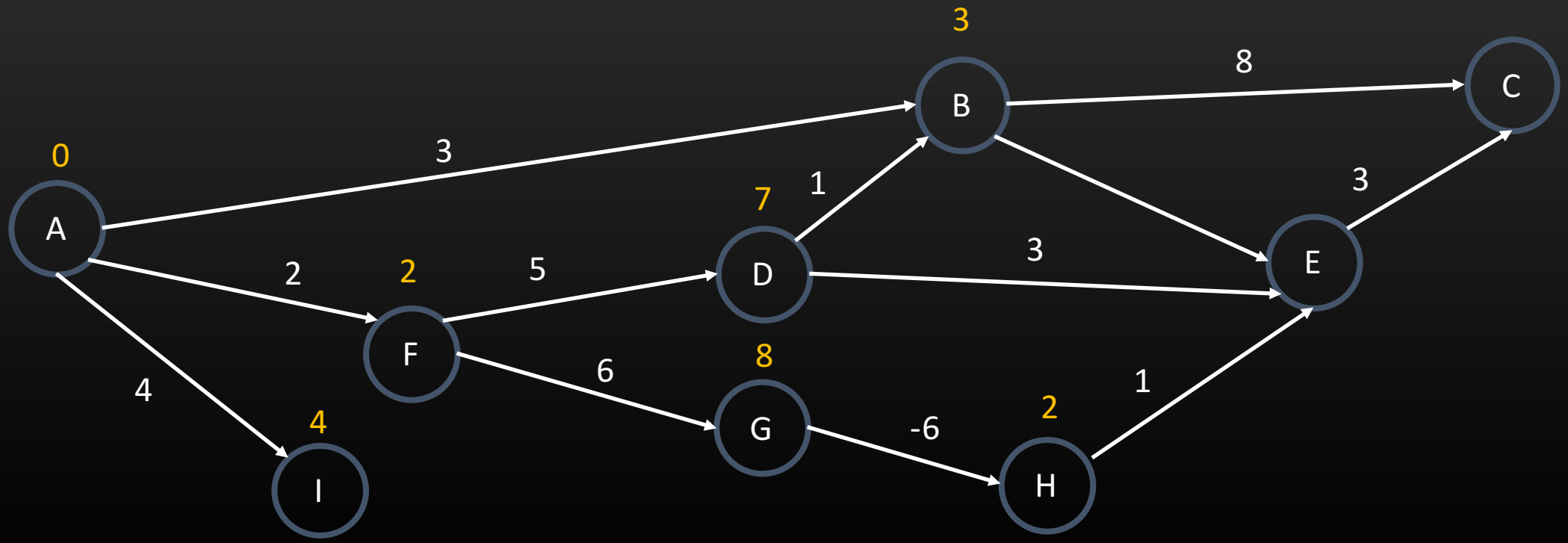
Example

- Find the topological order first: AFIDGBHEC
- Calculate the shortest paths based on the topological order



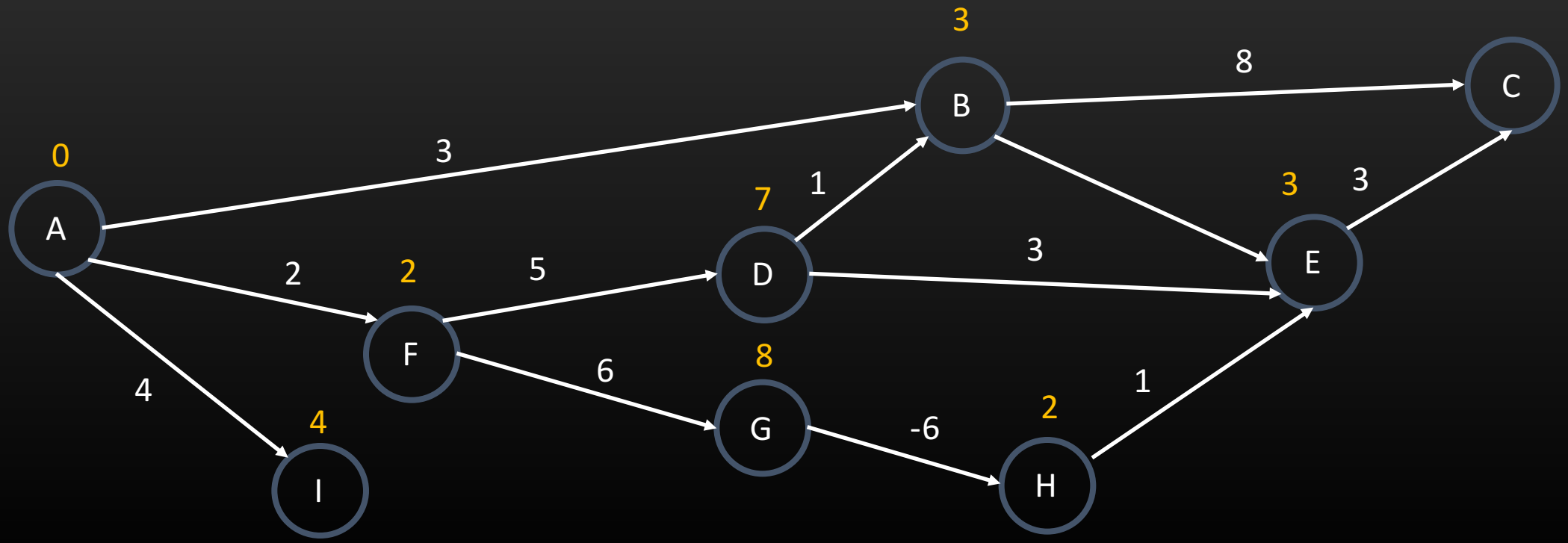
Example

- Find the topological order first: AFIDGBHEC
- Calculate the shortest paths based on the topological order



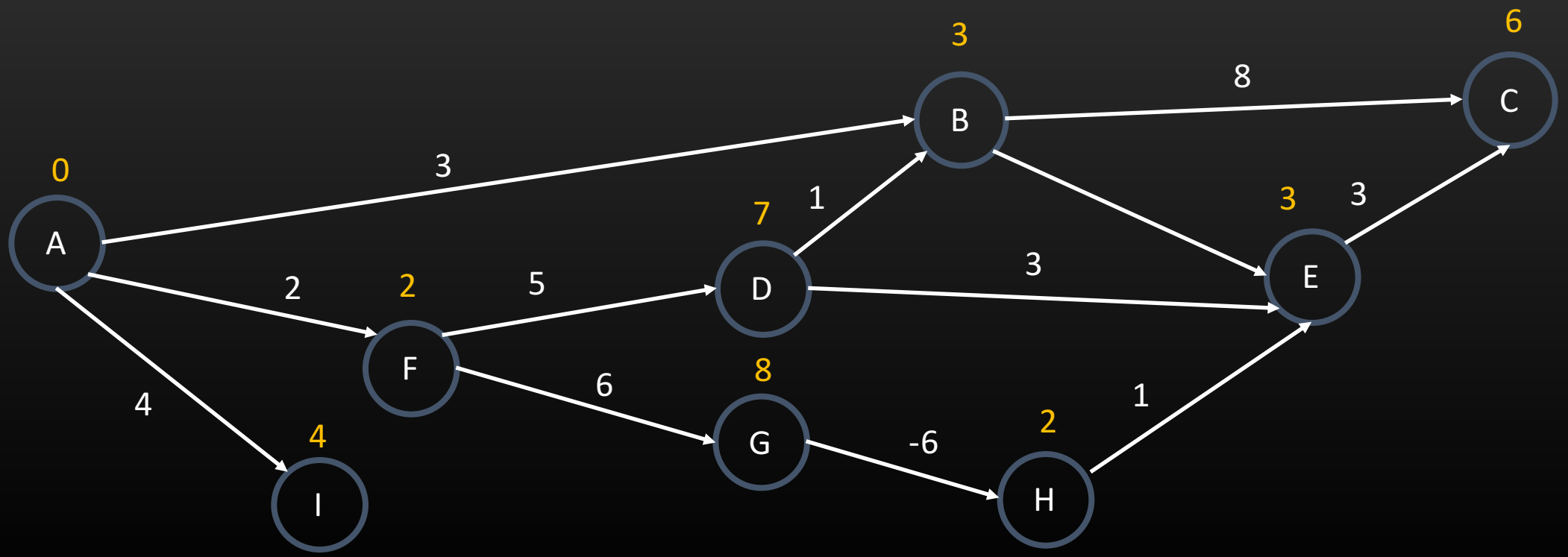
Example

- Find the topological order first: AFIDGBHEC
- Calculate the shortest paths based on the topological order



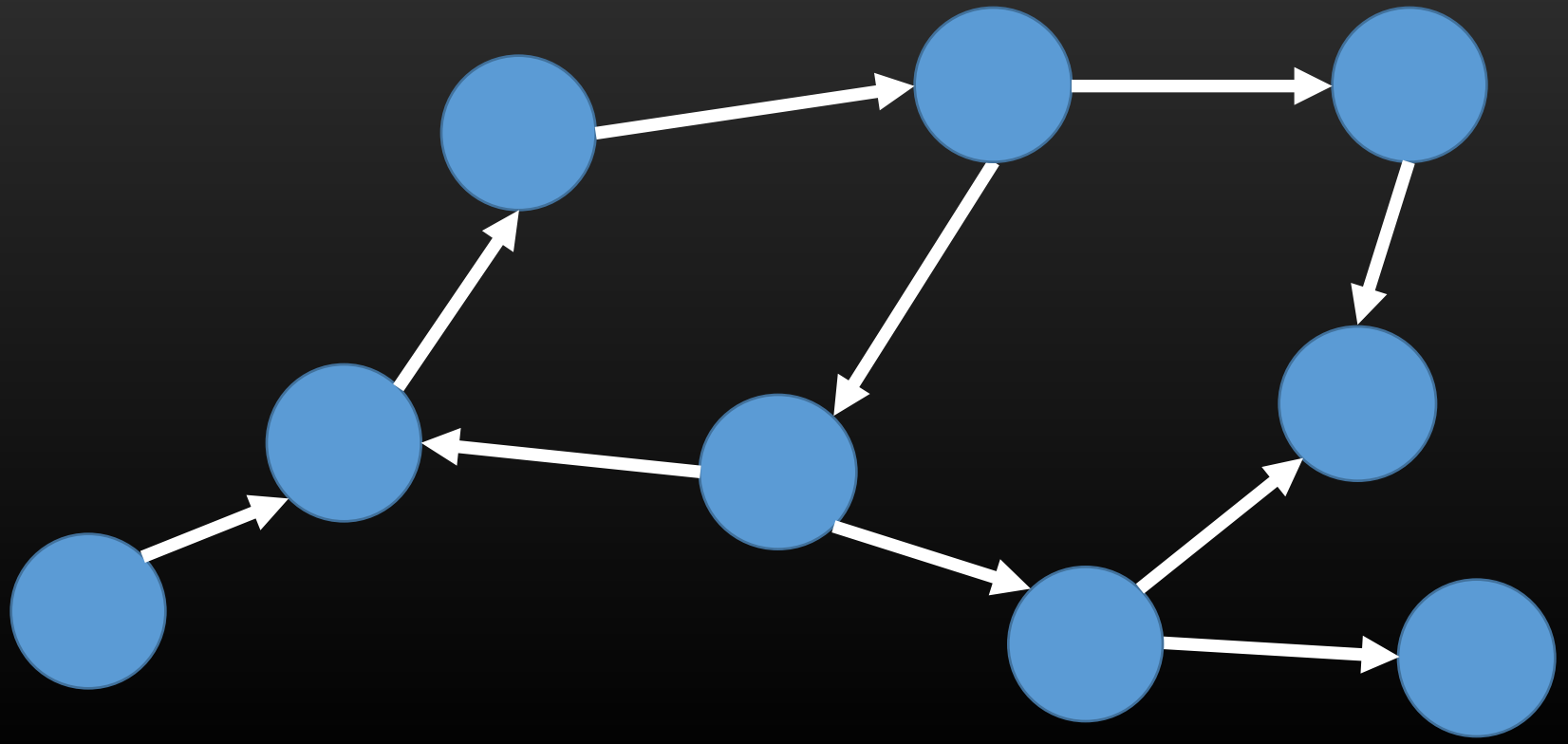
Example

- Find the topological order first: AFIDGBHEC
- Calculate the shortest paths based on the topological order

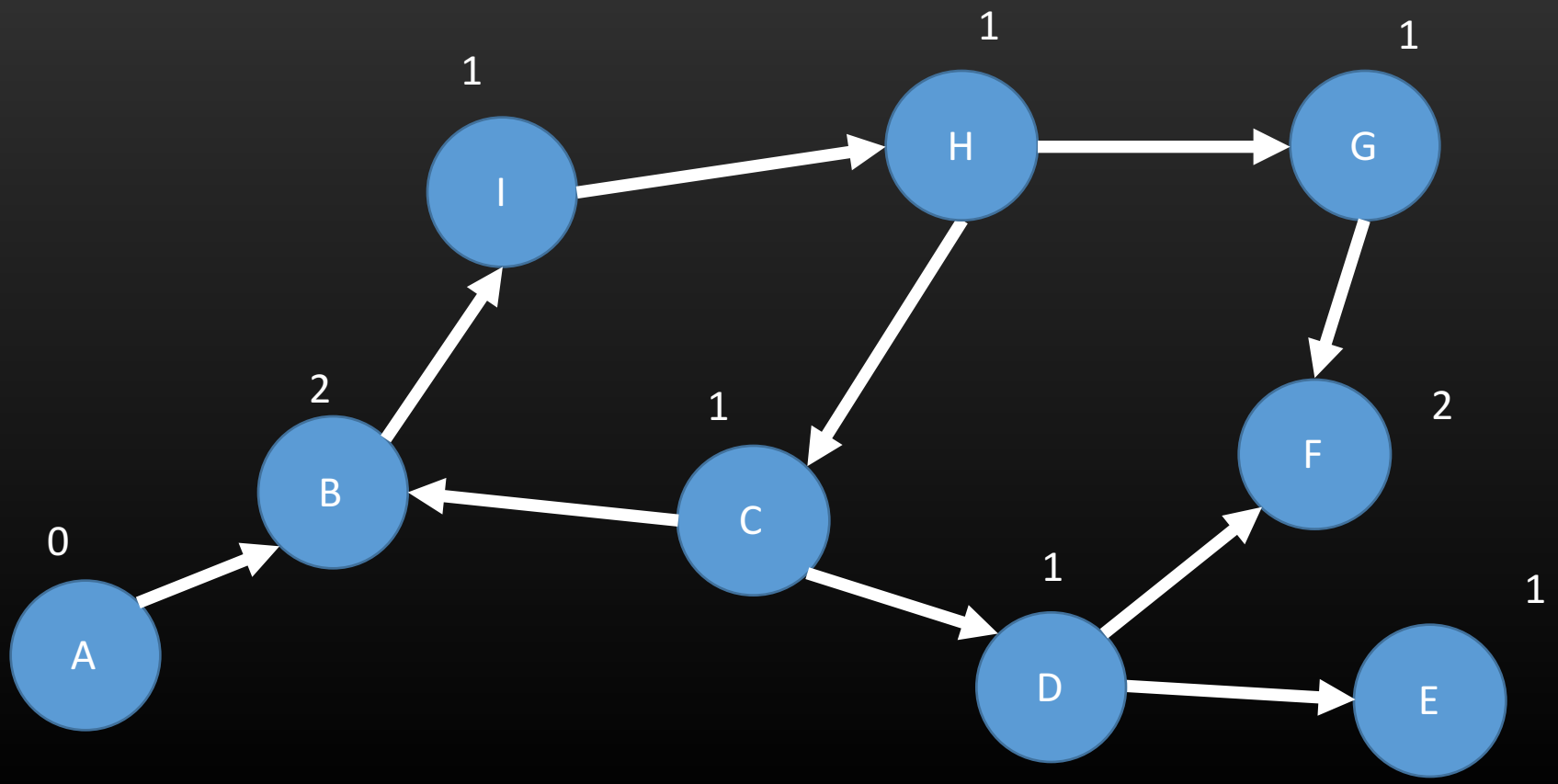


Detecting cycles

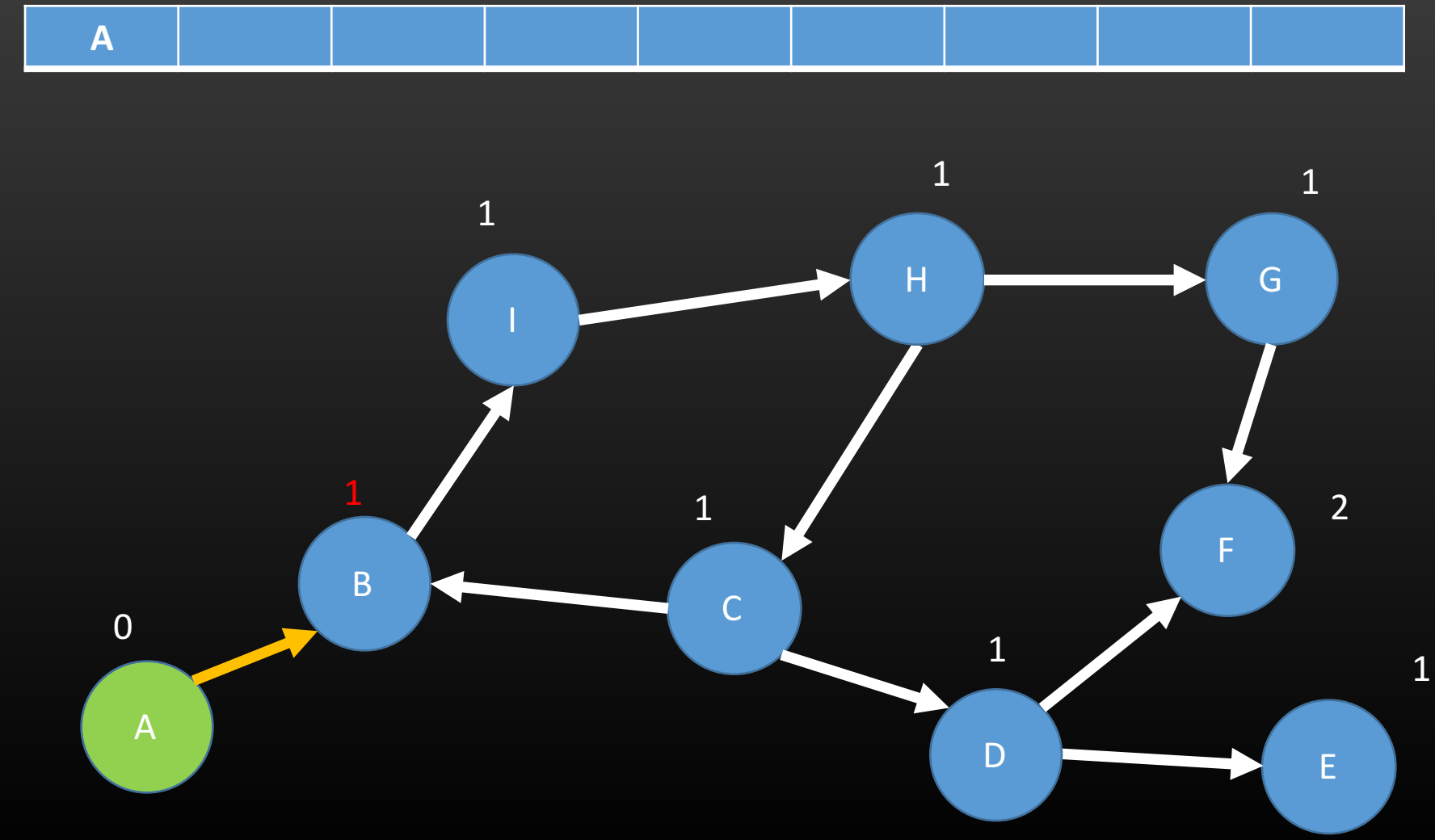
- Topological sort can find an order in a DAG
- What will happen if the graph is not acyclic?



Detecting cycles

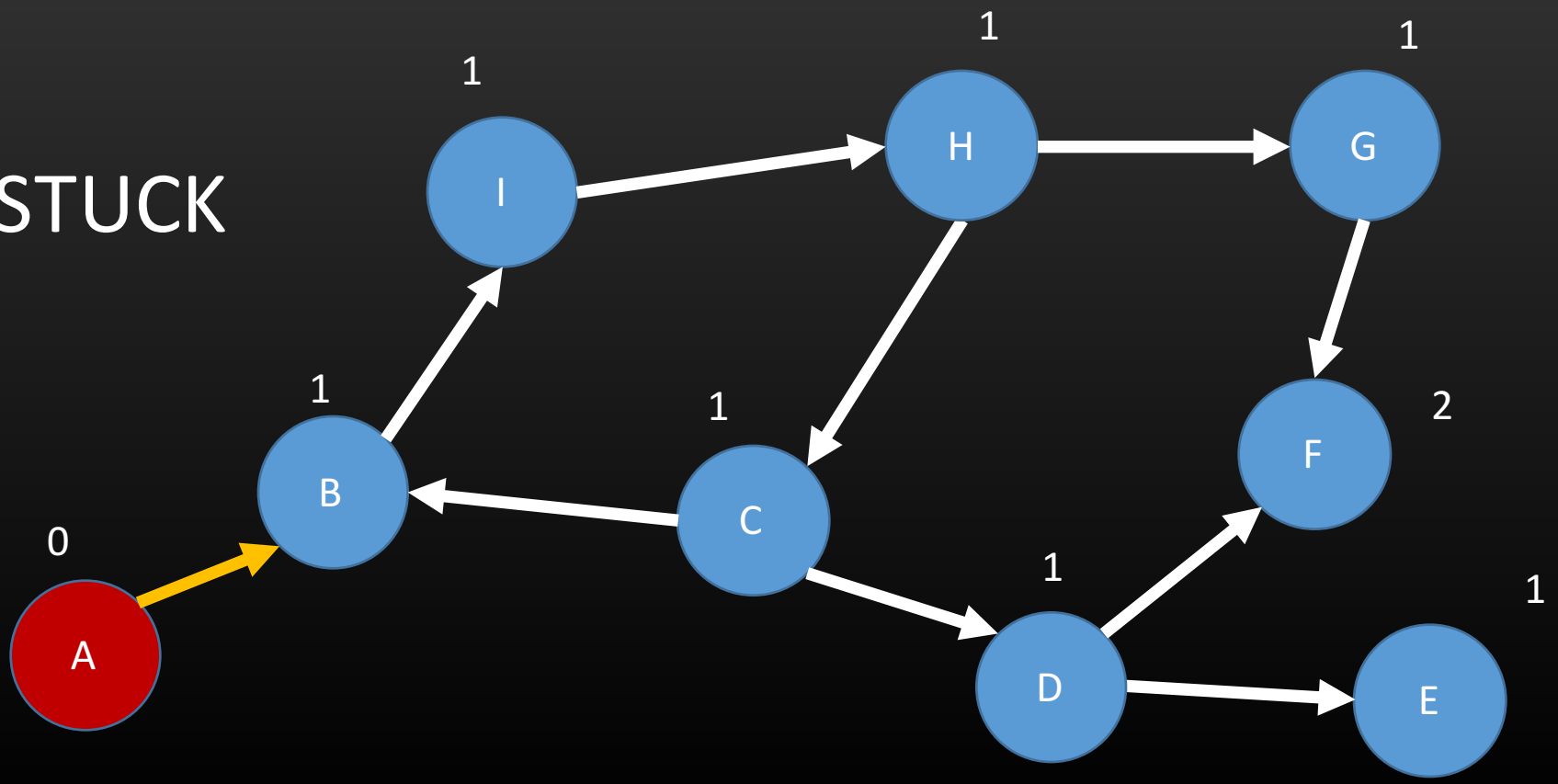


Detecting cycles



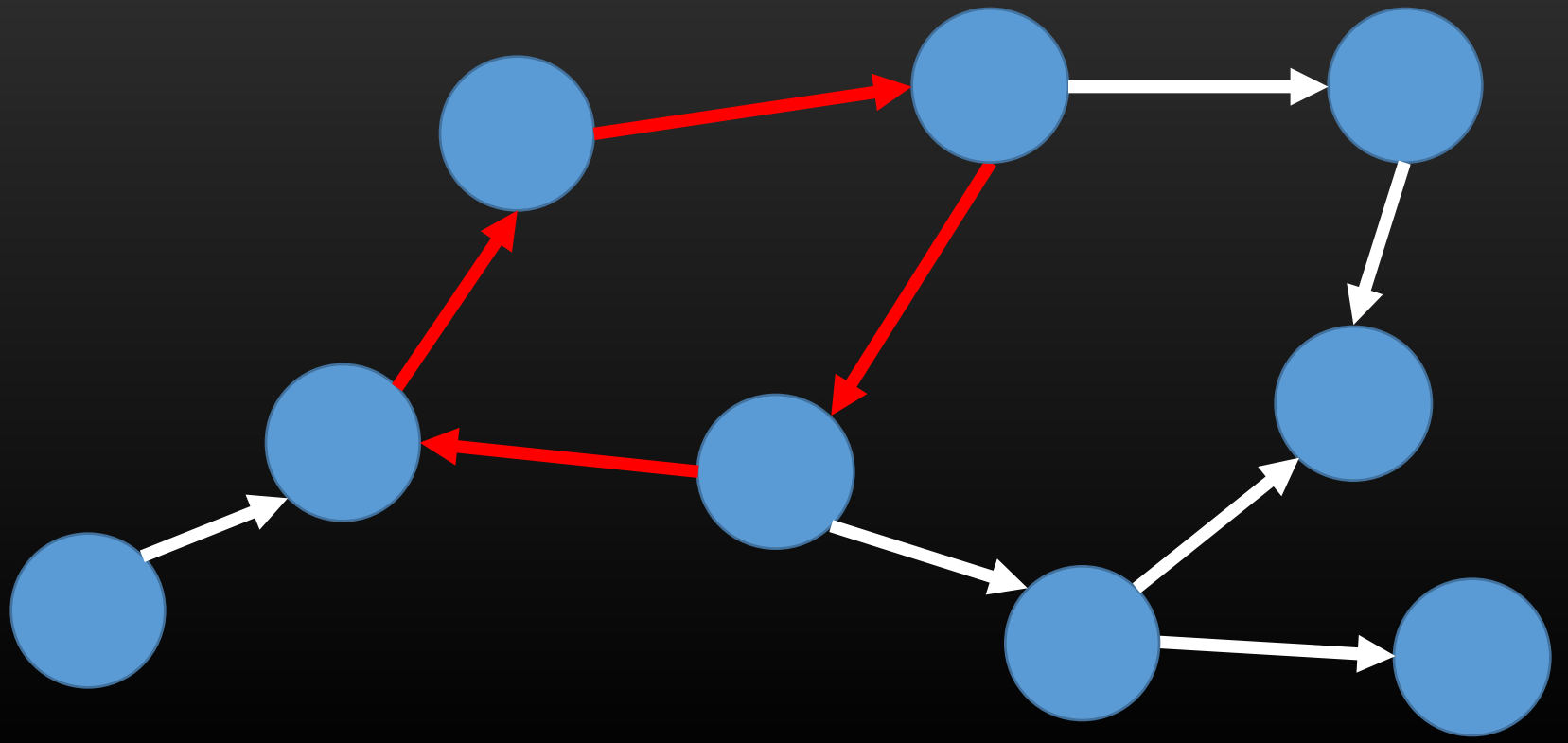
Detecting cycles

STUCK



Detecting cycles

- If the topological sort can not be finished, there are cycles in the graph

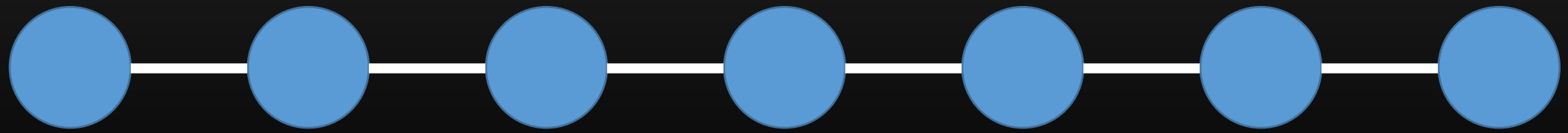


Other Special Graphs

- Beside of trees and DAG, there are other kinds of graphs that can make problems easier

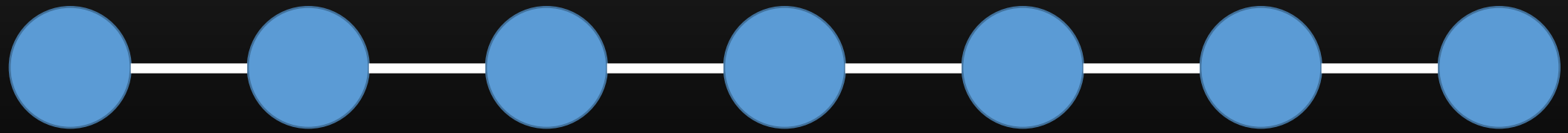
Chain

- All vertices have 2 neighbors, except two of them only have 1 neighbor



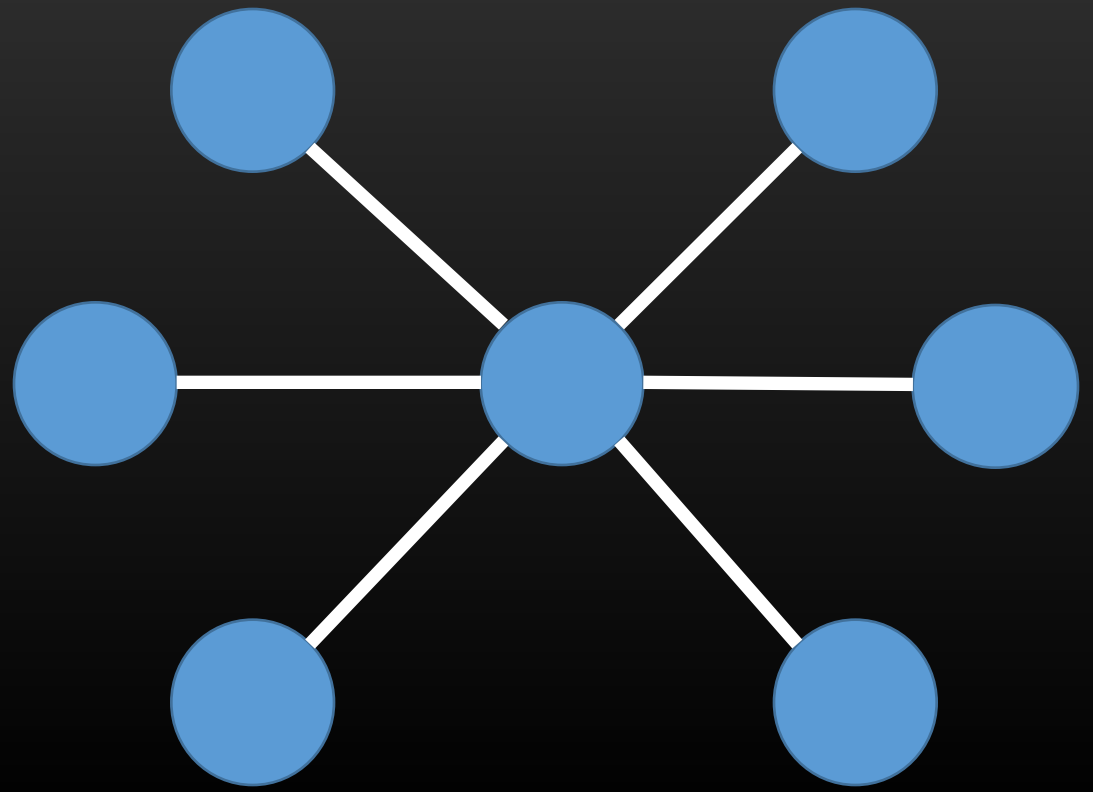
Chain

- No cycles
- No branches
- Easy calculations
- Process the nodes from head to tail



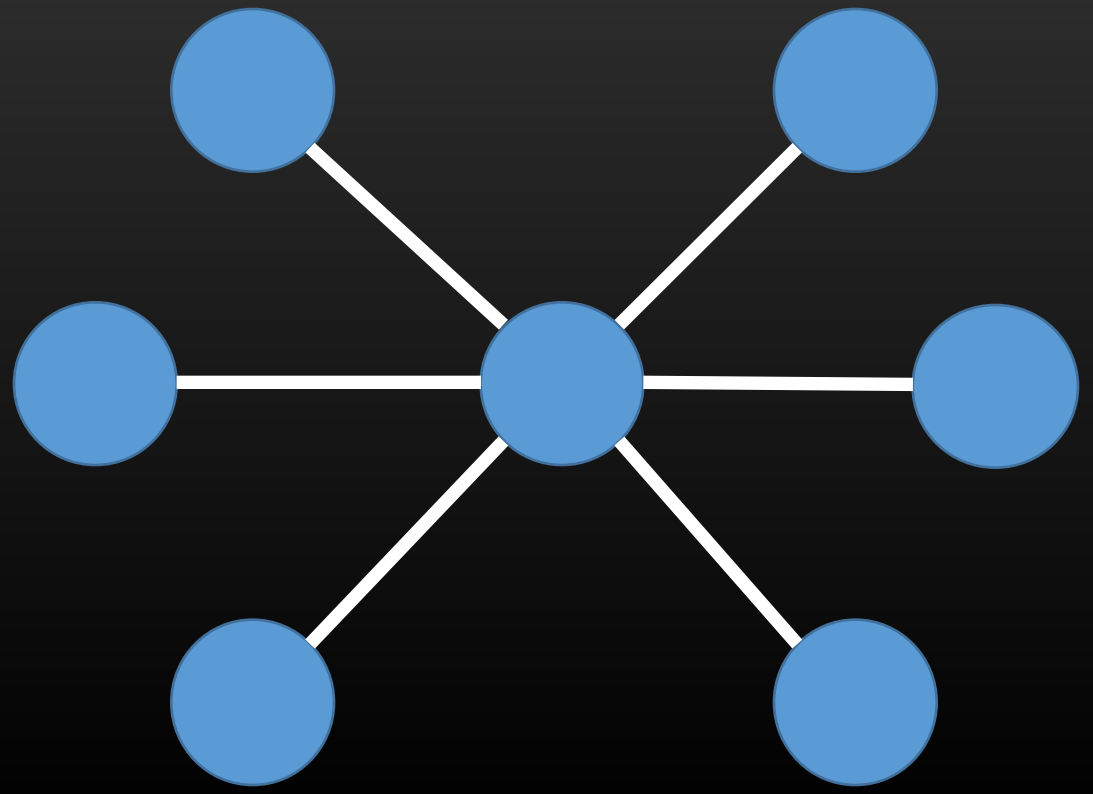
Star

- A tree with one internal node and k leaves



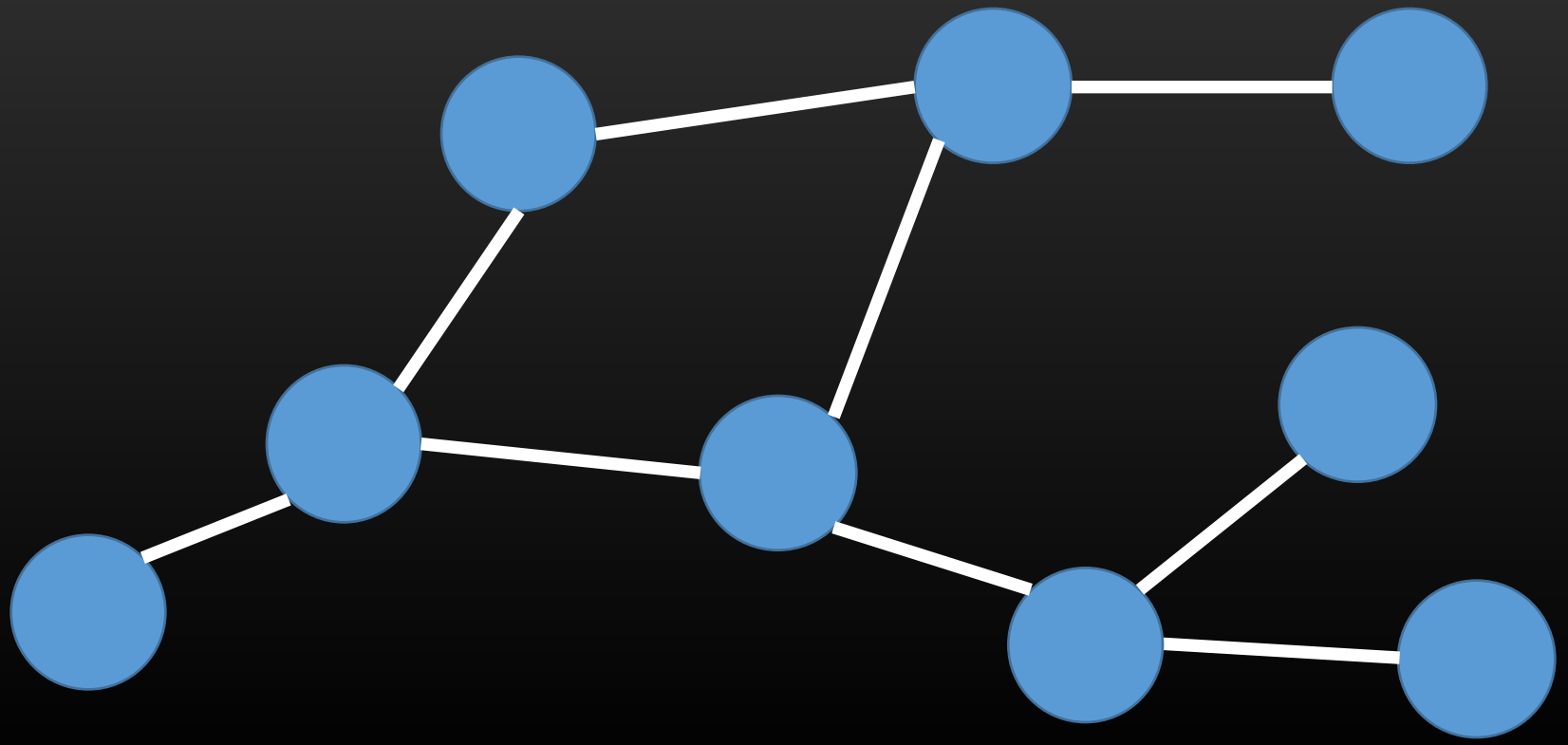
Star

- No cycles
- Maximum distance = 2
- Easy calculations
- Special handle the internal node



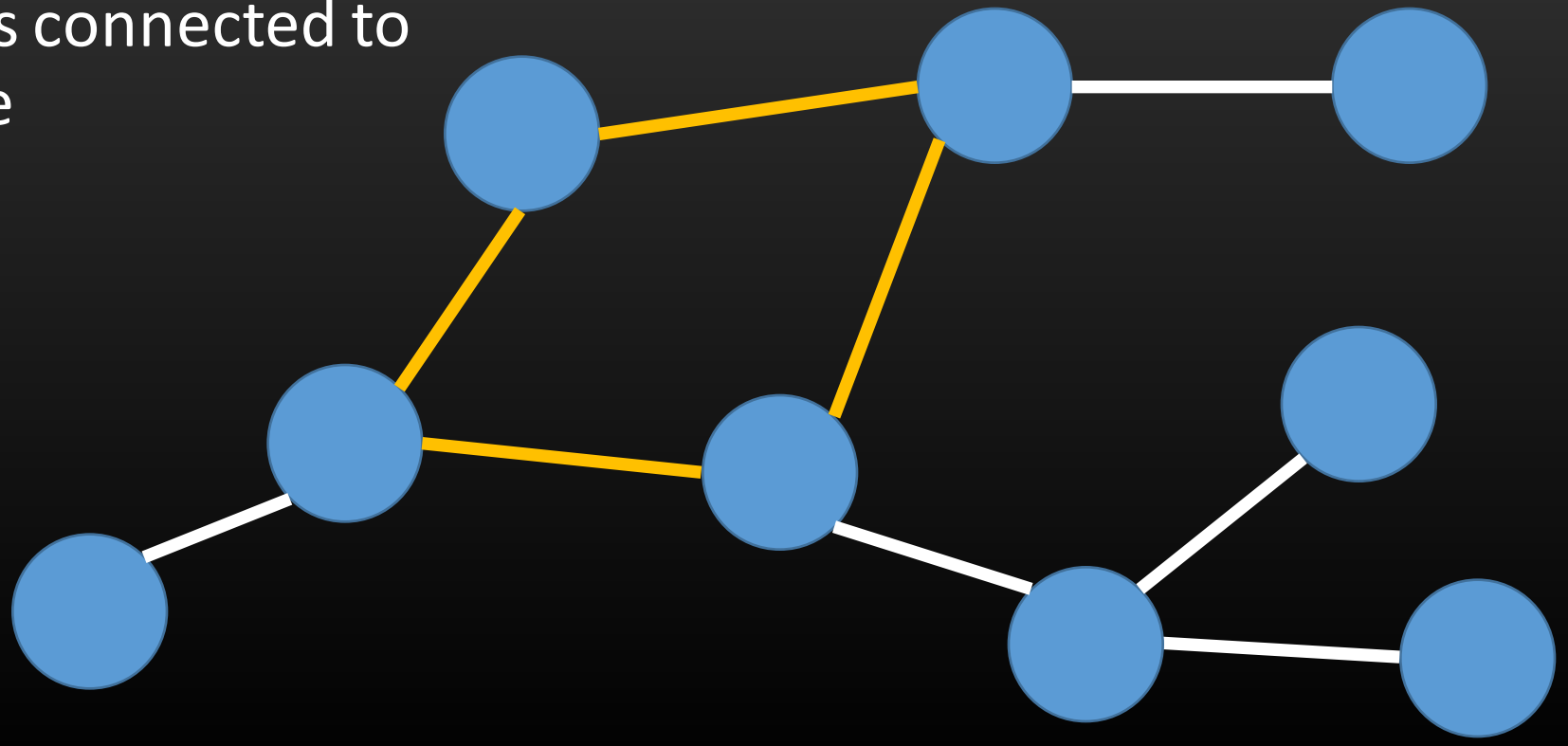
Unicyclic Graph

- A connected graph with exactly one cycle



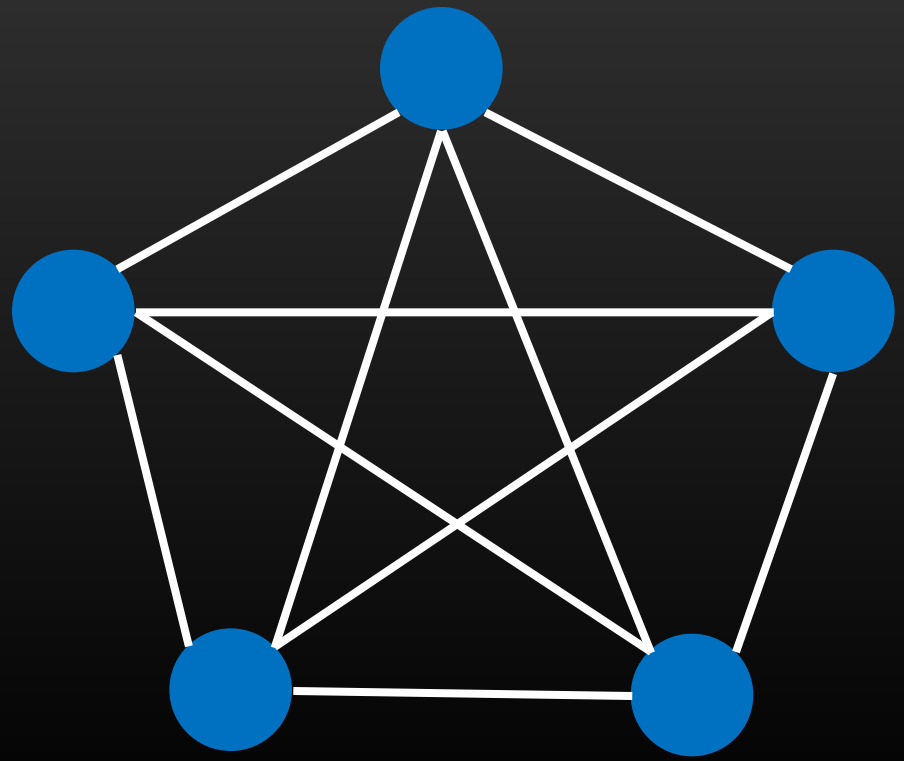
Unicyclic Graph

- |V| = |E|
- Work on the cycle first
- Process the subtrees connected to the cycle one by one



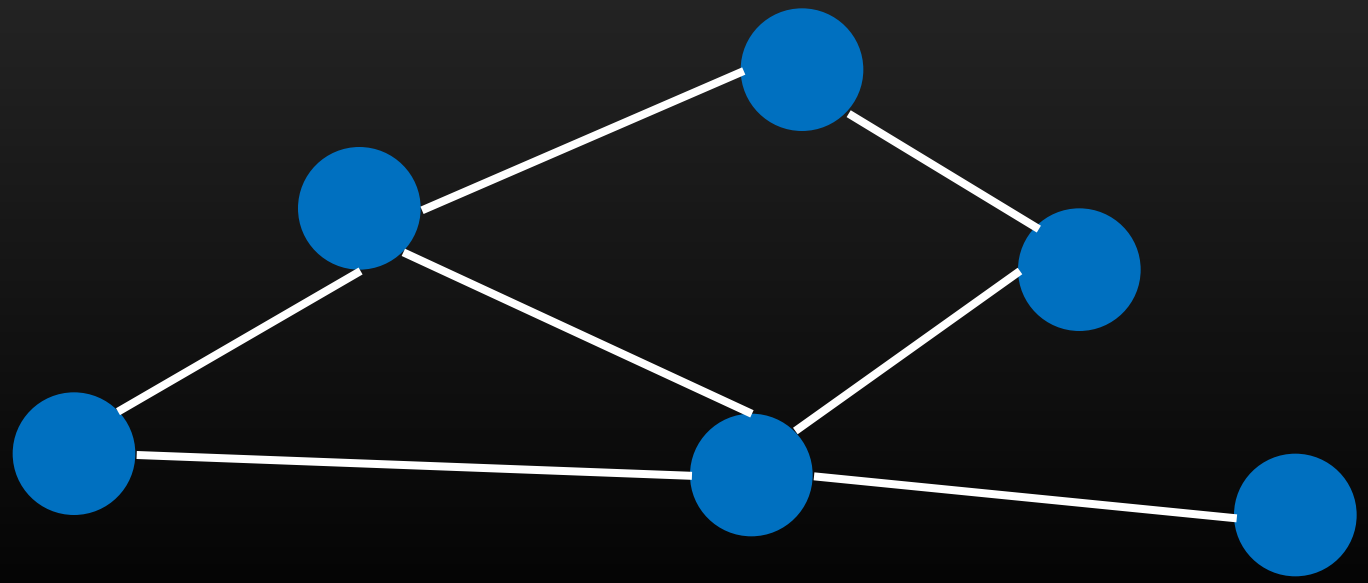
Complete Graph

- Every pair of distinct vertices is connected by a unique edge
- |E| = |V| * (|V| - 1) / 2



Planar Graph

- A graph whose vertices and edges can be drawn in a plane such that no two of the edges intersect
- Using its properties, more efficient algorithms can be applied

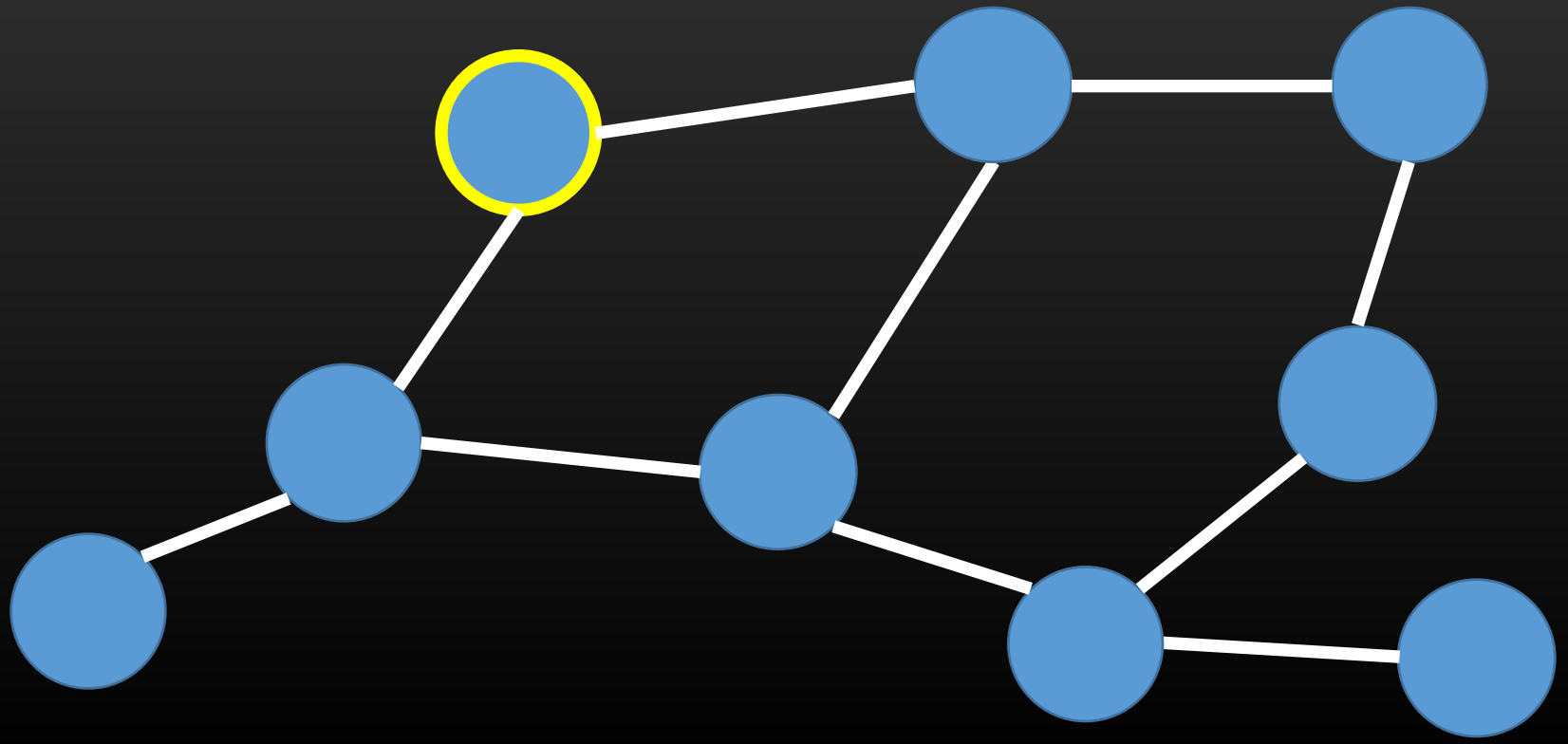


More on DFS and BFS

- As mentioned in Graph I, DFS cannot find shortest paths in a normal unweighted graph
- However, we can modify it

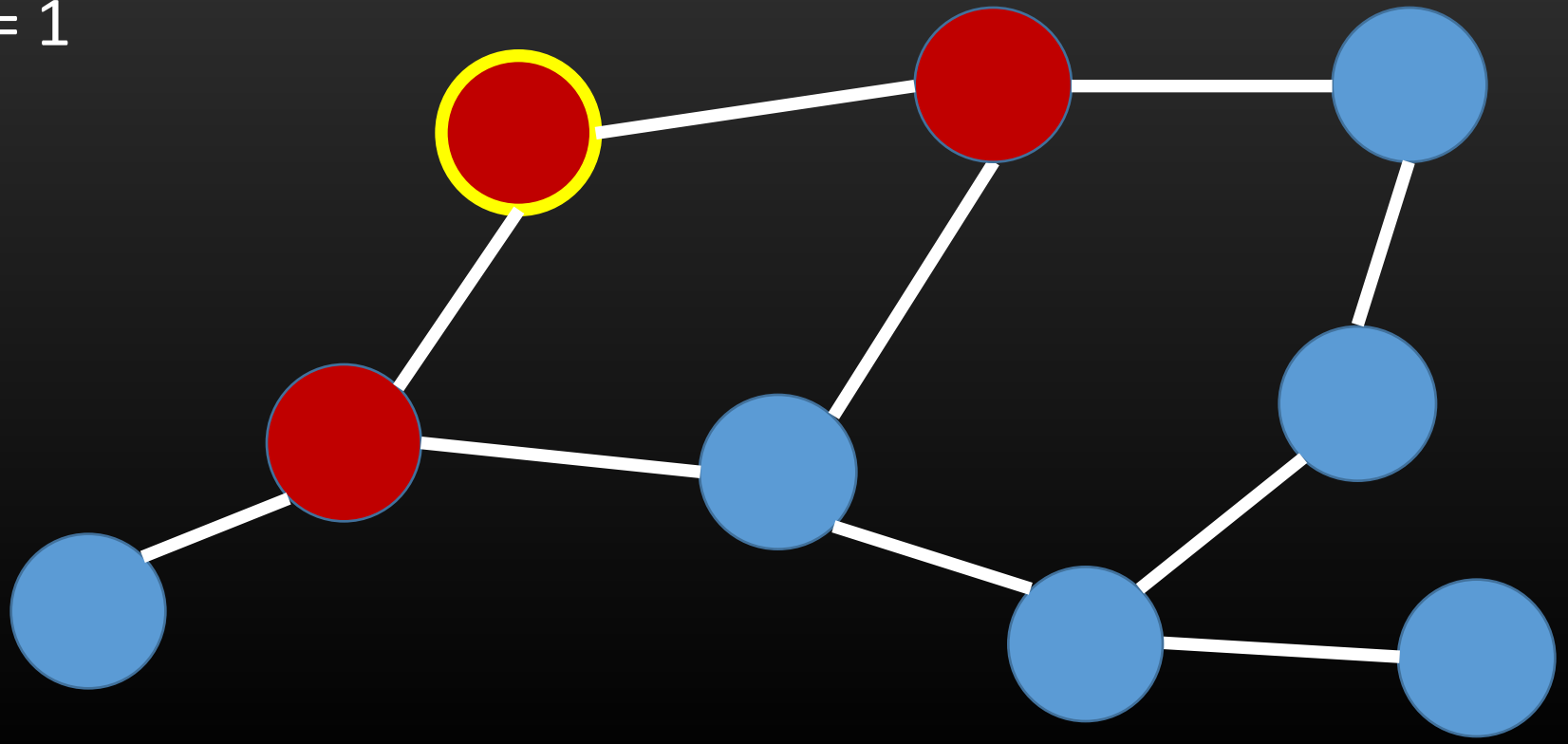
Iterative deepening depth-first search

- A depth-limited version of depth-first search
- Run repeatedly with increasing depth limits until the goal is found



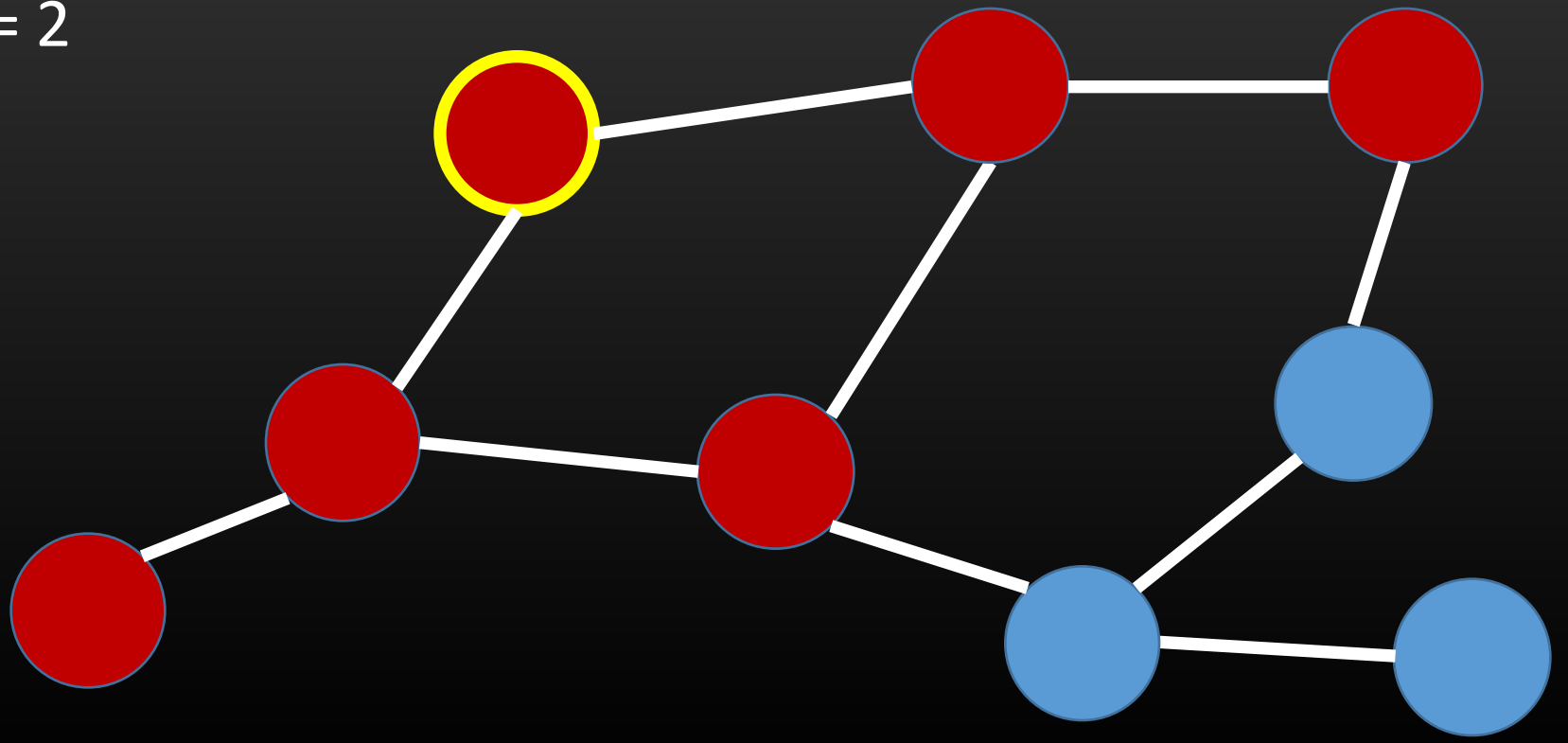
Iterative deepening depth-first search

- A depth-limited version of depth-first search
- Run repeatedly with increasing depth limits until the goal is found
- Limit = 1



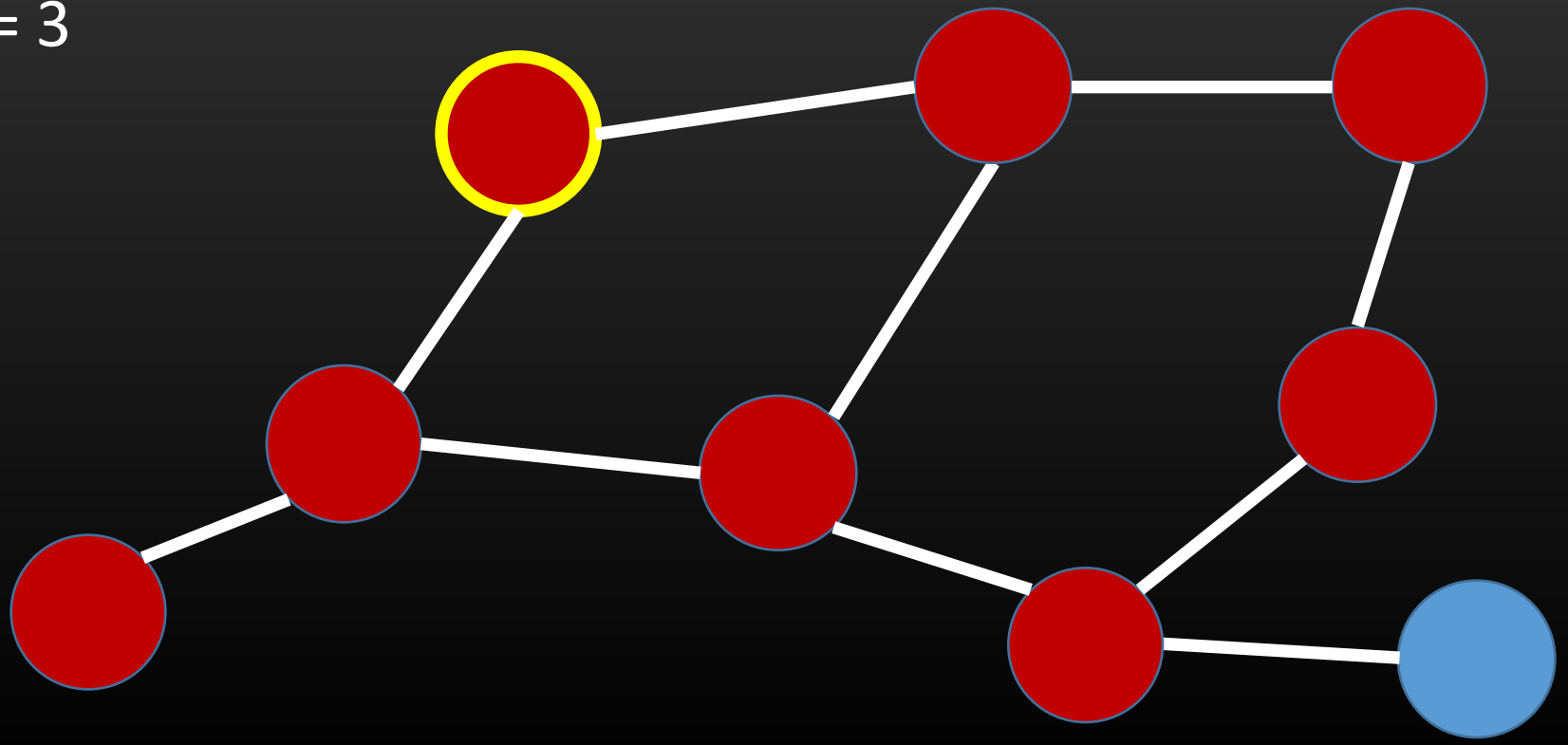
Iterative deepening depth-first search

- A depth-limited version of depth-first search
- Run repeatedly with increasing depth limits until the goal is found
- Limit = 2



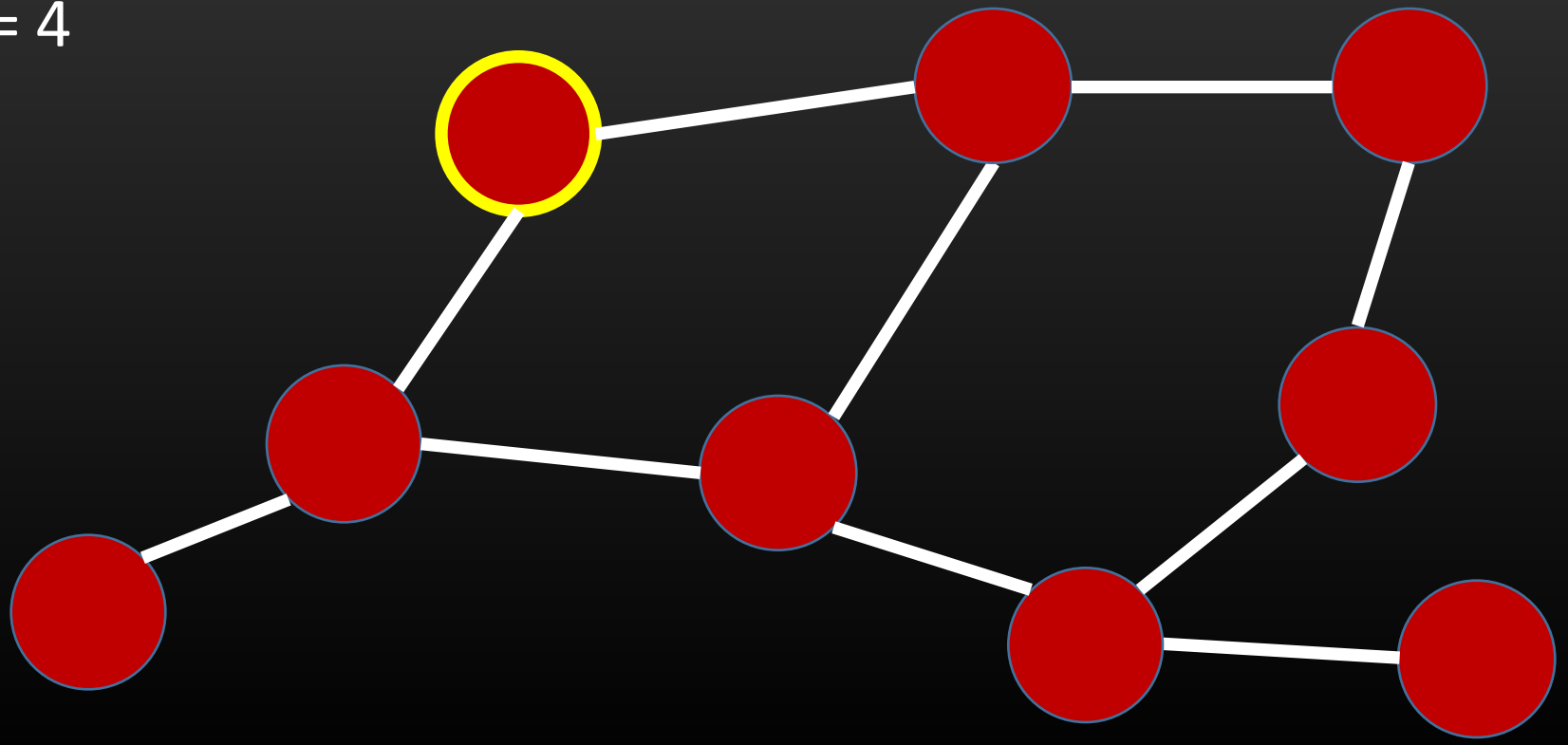
Iterative deepening depth-first search

- A depth-limited version of depth-first search
- Run repeatedly with increasing depth limits until the goal is found
- Limit = 3



Iterative deepening depth-first search

- A depth-limited version of depth-first search
- Run repeatedly with increasing depth limits until the goal is found
- Limit = 4



Iterative deepening depth-first search

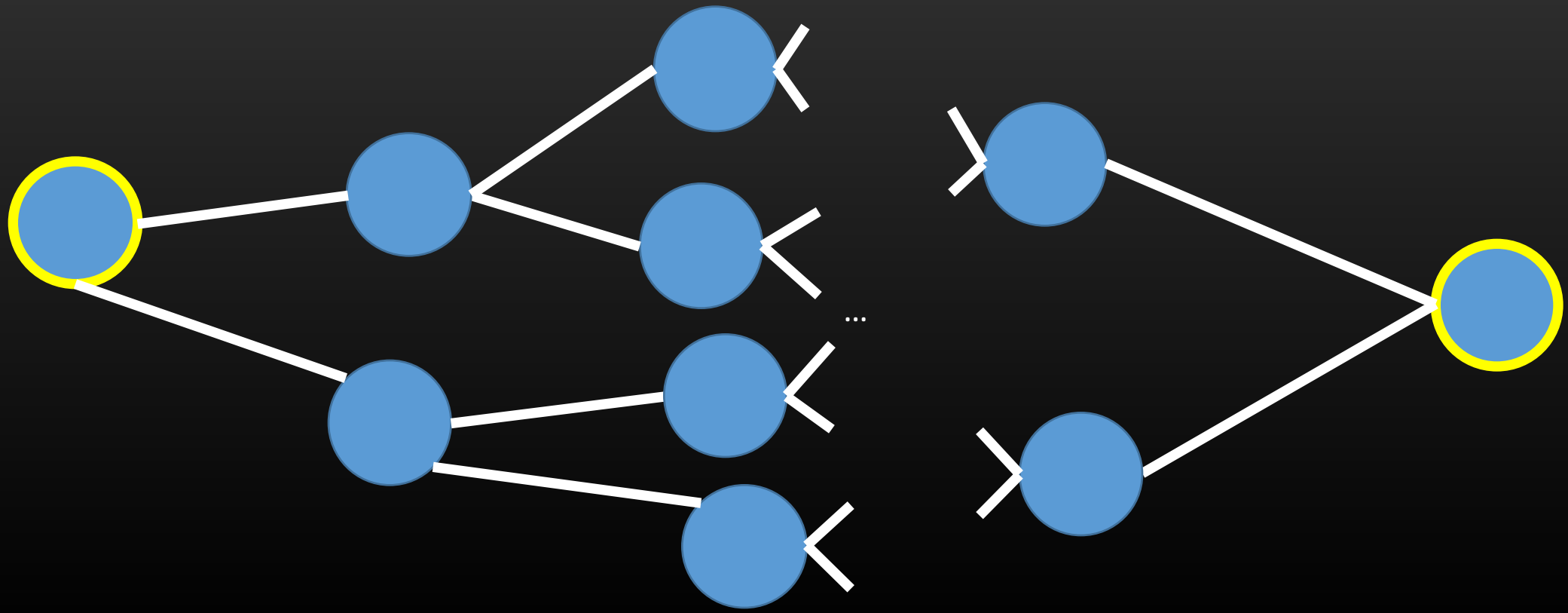
```
Procedure ids(){
        limit = 0
        while goal is not found
                limit = limit + 1
                dfs(root,limit)
}
Proceudre dfs(node, depth){
        if depth<0 return
        process the node
        for all neighbor w of node
                if w is not visited
                        dfs(w, depth – 1)
}
```

Iterative deepening depth-first search

- Same as Breadth First Search
- Much less memory is used

Bidirectional search

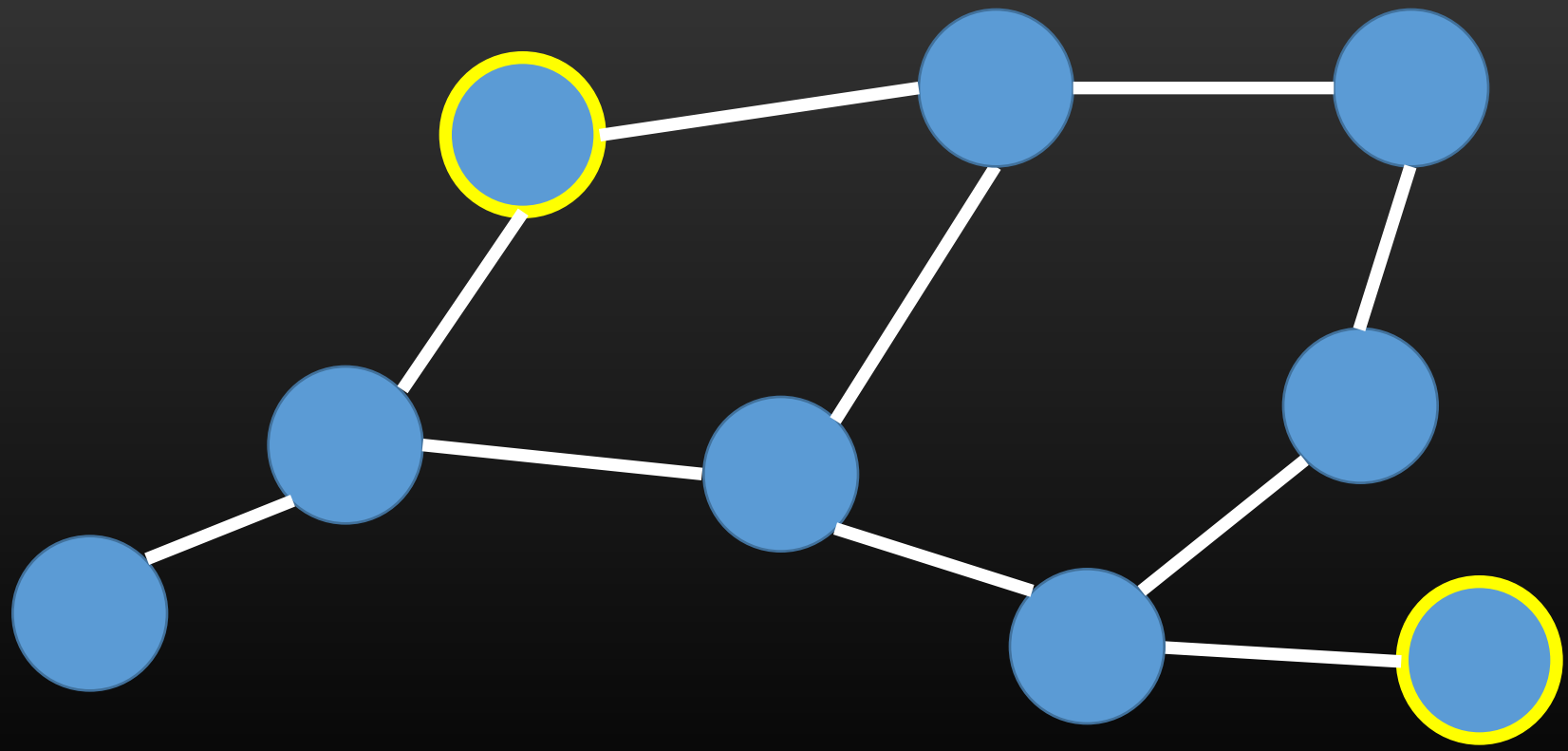
- Sometimes BFS is not fast enough
- Too many useless nodes/states are visited



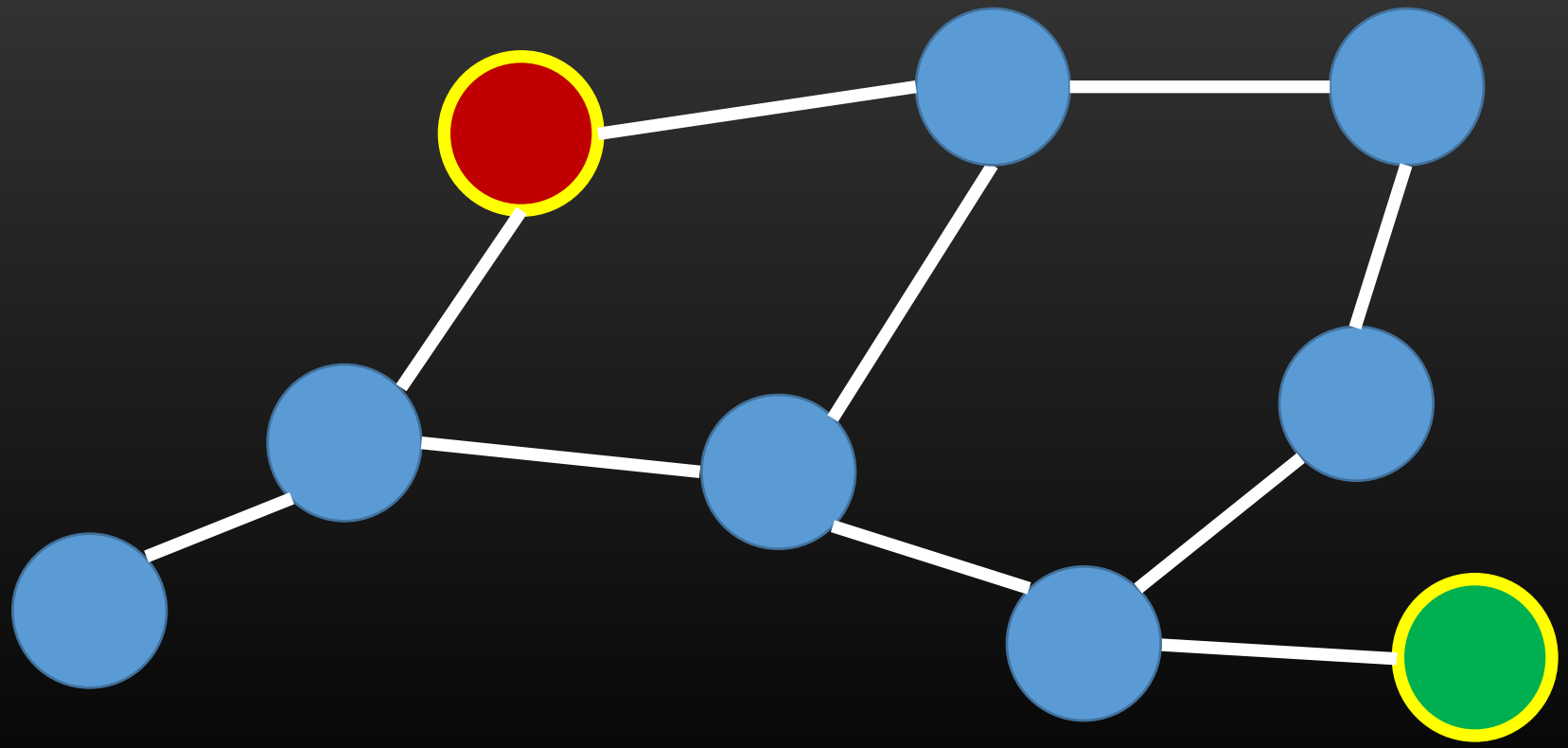
Bidirectional search

- Runs two searches simultaneously
- One forward from the initial state
- One backward from the goal
- Stop when the two meet in the middle

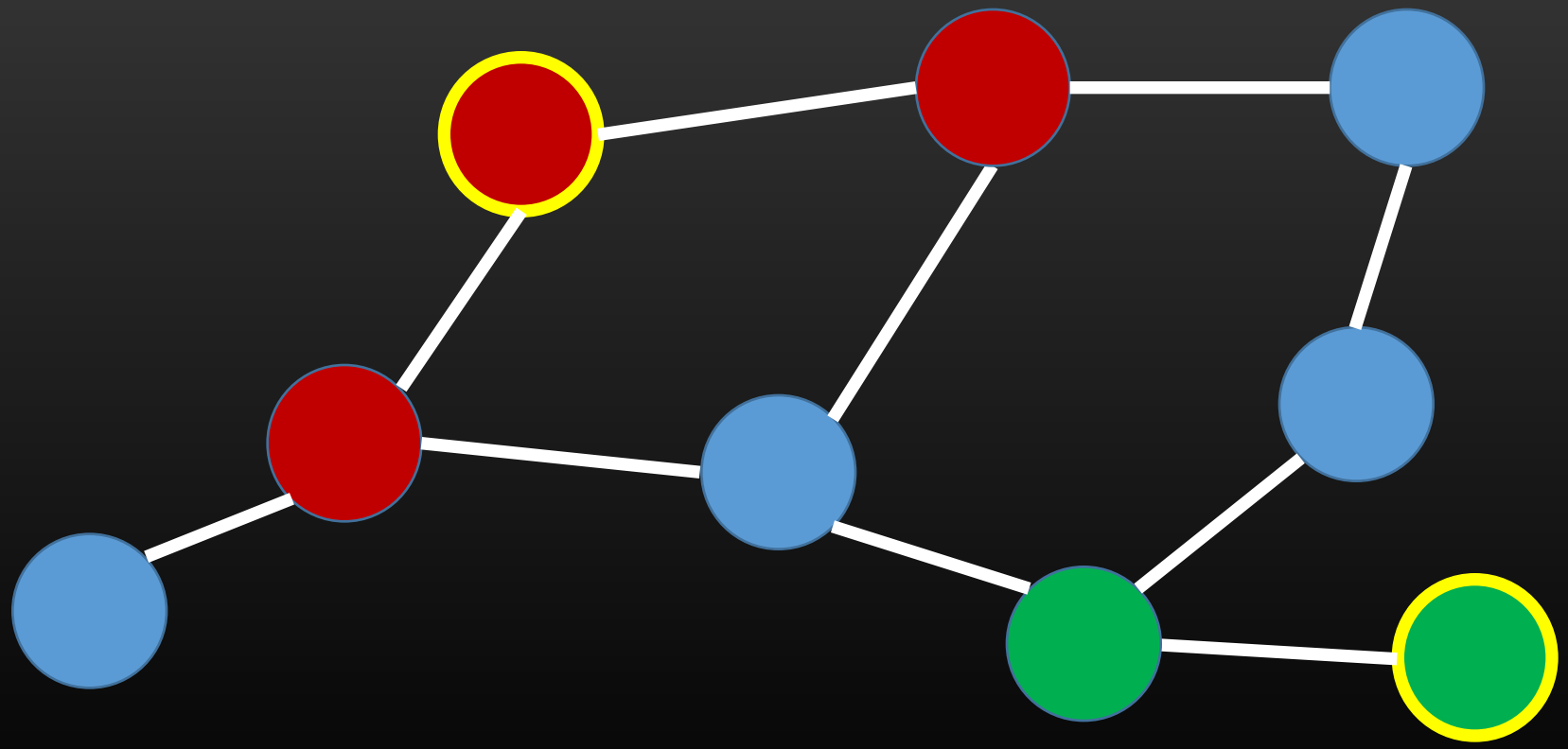
Bidirectional search



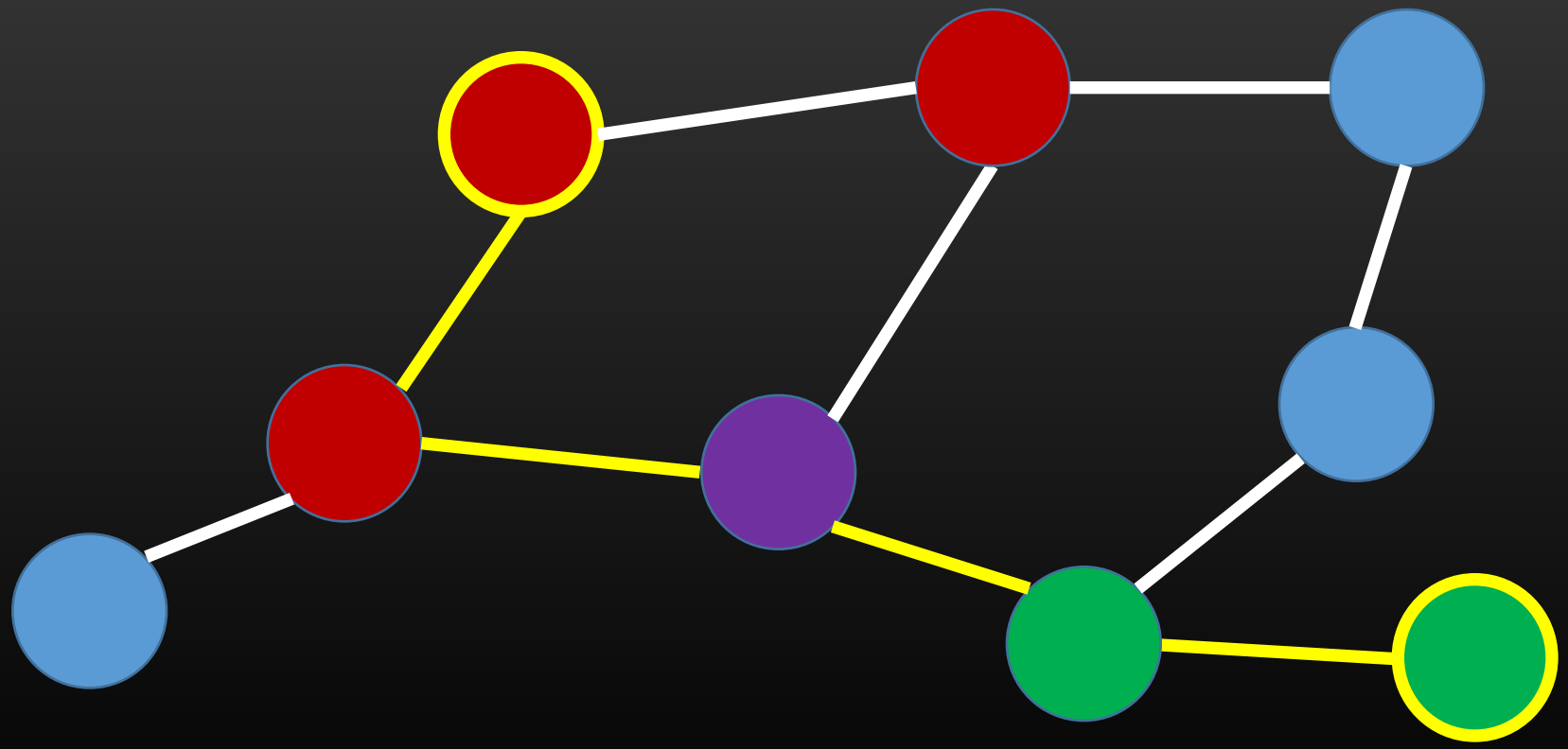
Bidirectional search



Bidirectional search



Bidirectional search



Bidirectional search

```
Procedure bds{
        push starting point and ending point into the queue
        while the queue is not empty do
            t = front of the queue
            for all vertex w adjacent to t do
                if w is not visited
                    calculate the distance to w from the original direction
                    push w into the queue
                else if w was visited from the other direction
                    calculate the distance to w
                    return the distance from starting point to w + the distance from w to ending point
        pop t from the queue
}
```

Bidirectional search

- The running time is better than BFS
- Less irrelevant searching is done
- Useful in state searching