Graph III

Jason

Prerequisite

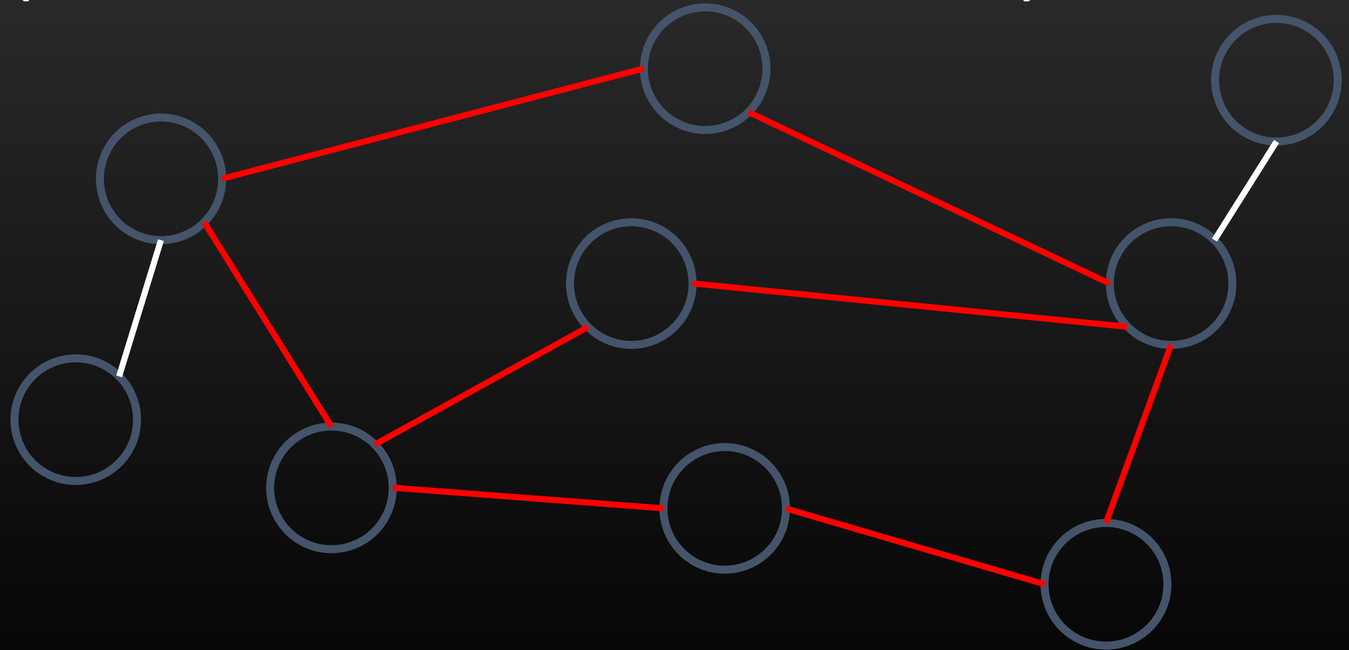
- Graph I
  - Definition of graphs
  - Graph representations
  - DFS and BFS

Table of Contents

- Trees
- Tree Traversals
- Directed Acyclic Graph
- Topological Sort
- Algorithms on Tree
- Lowest Common Ancestor

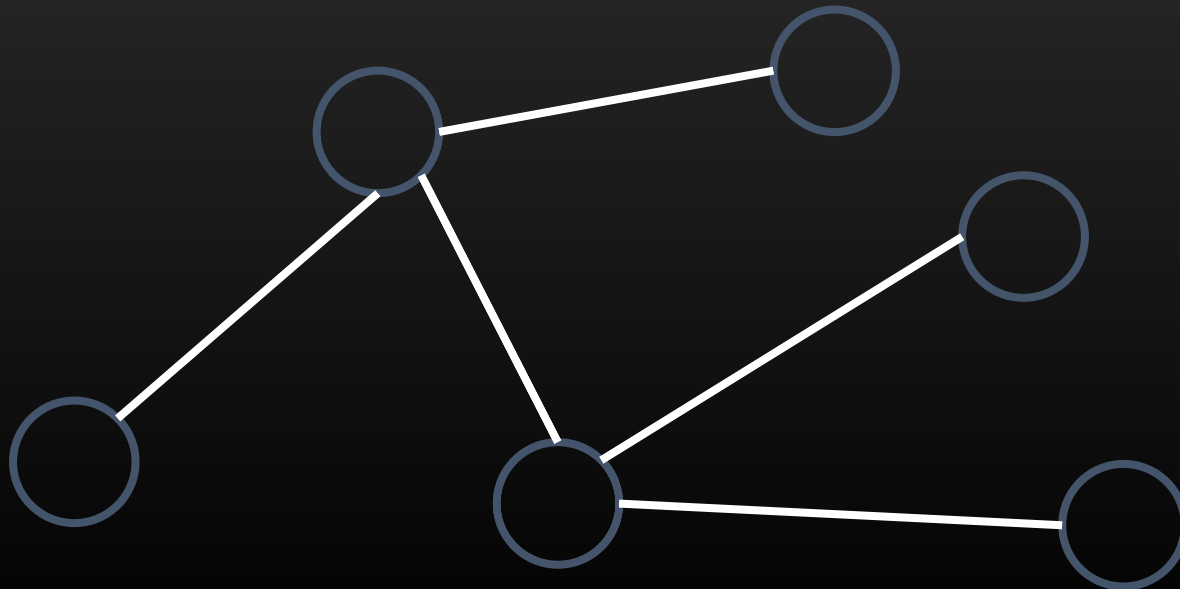
Cycles

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



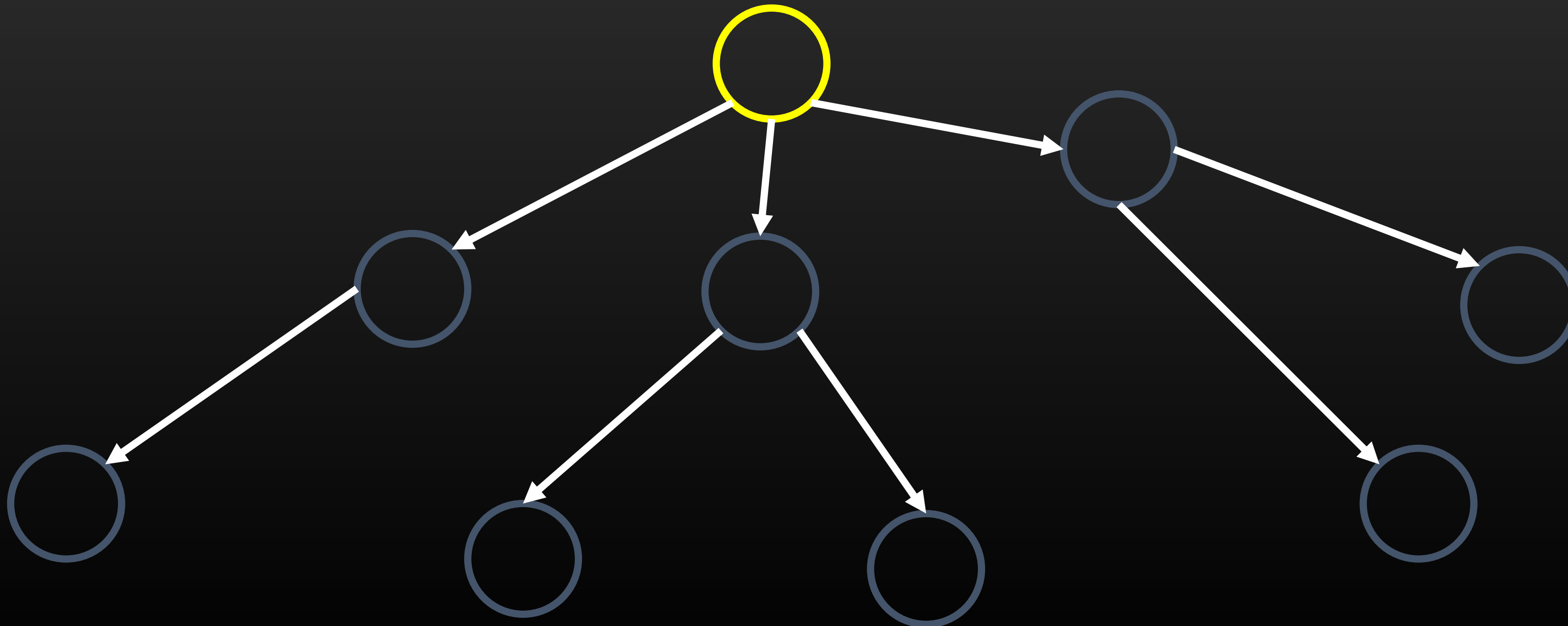
Trees

- 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

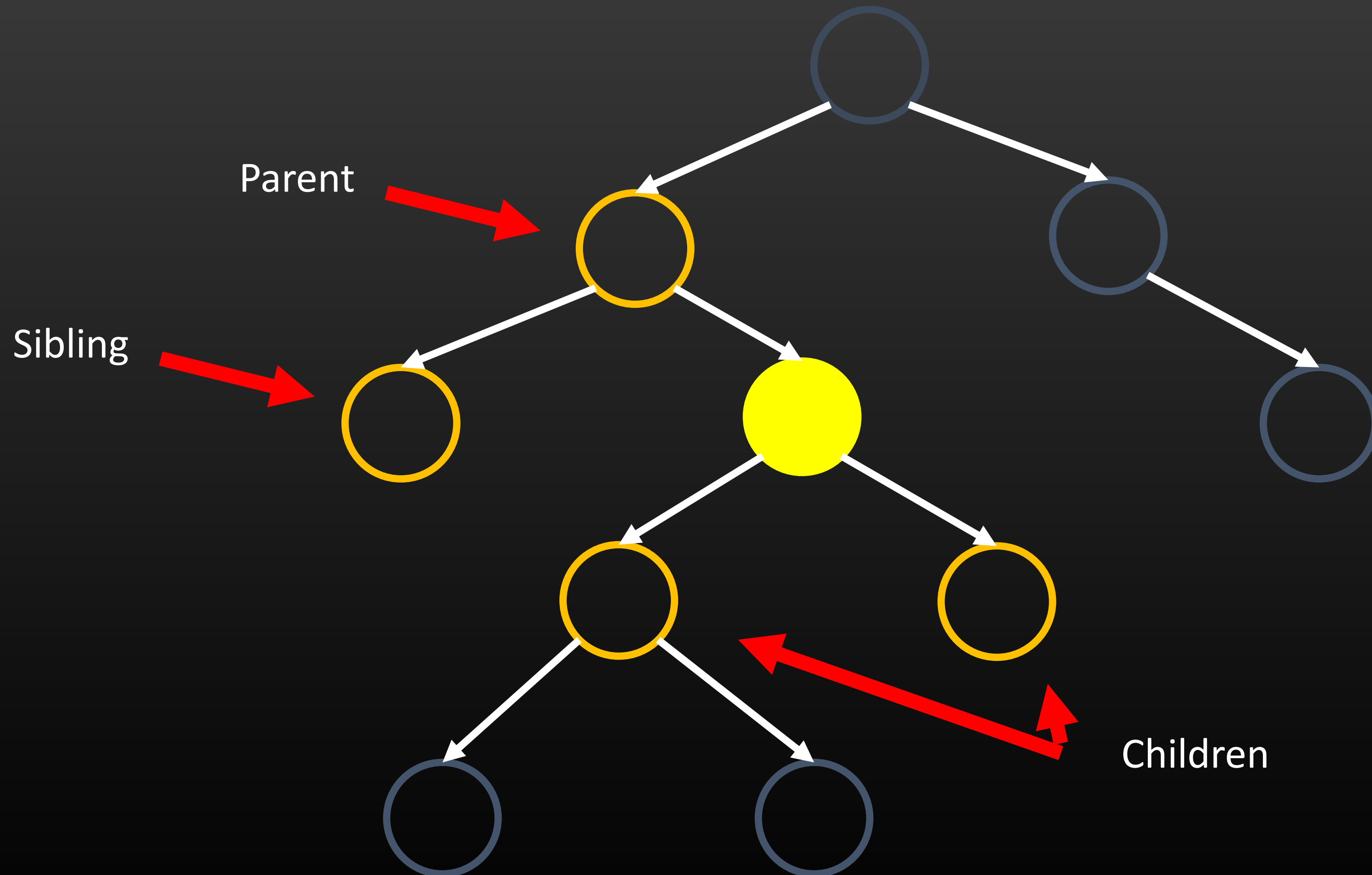


Rooted Tree

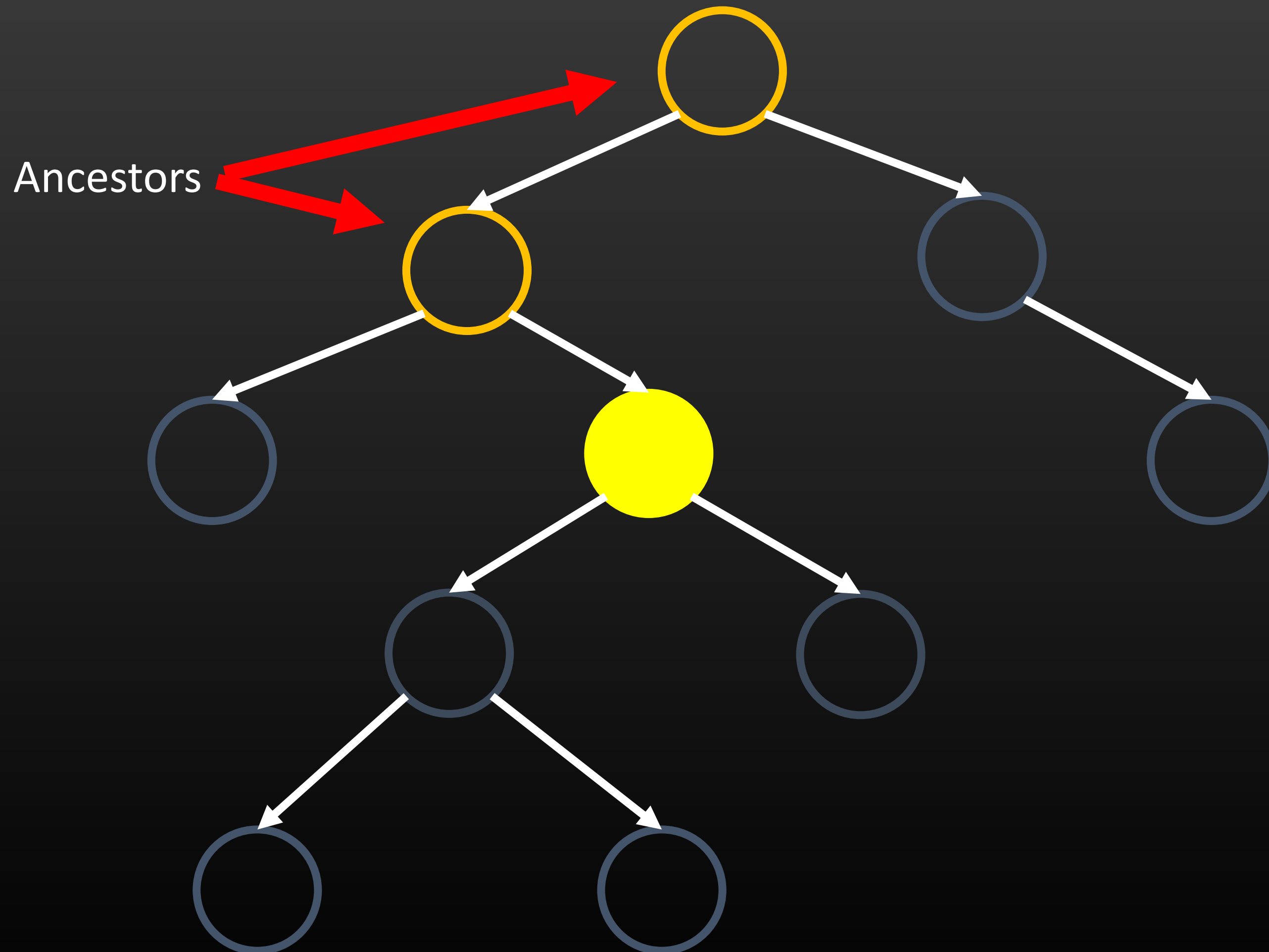
- Sometimes the edges can be directional
- One vertex has been designated the root
- The edges are directional and all away from the root



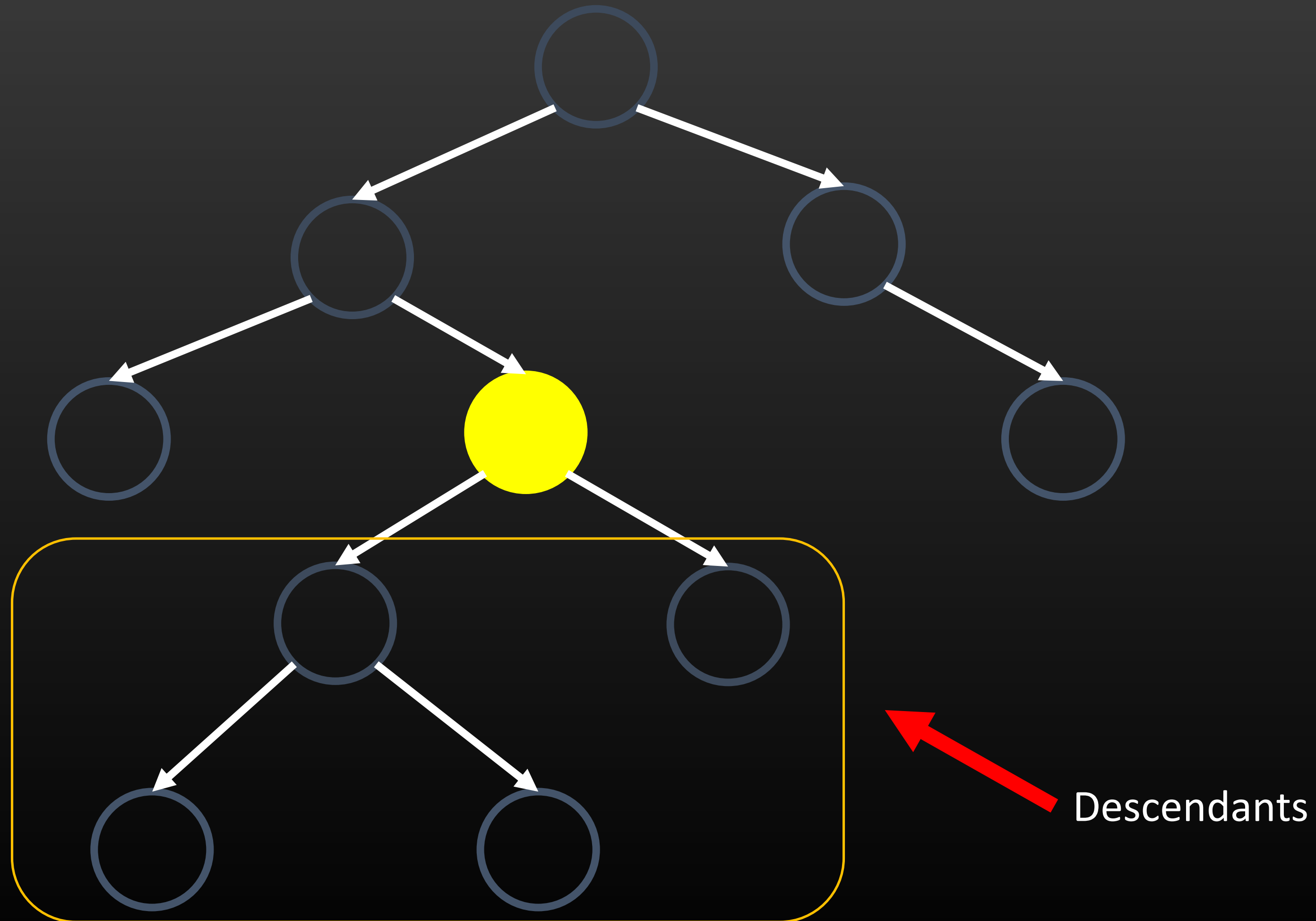
Terms on directed tree



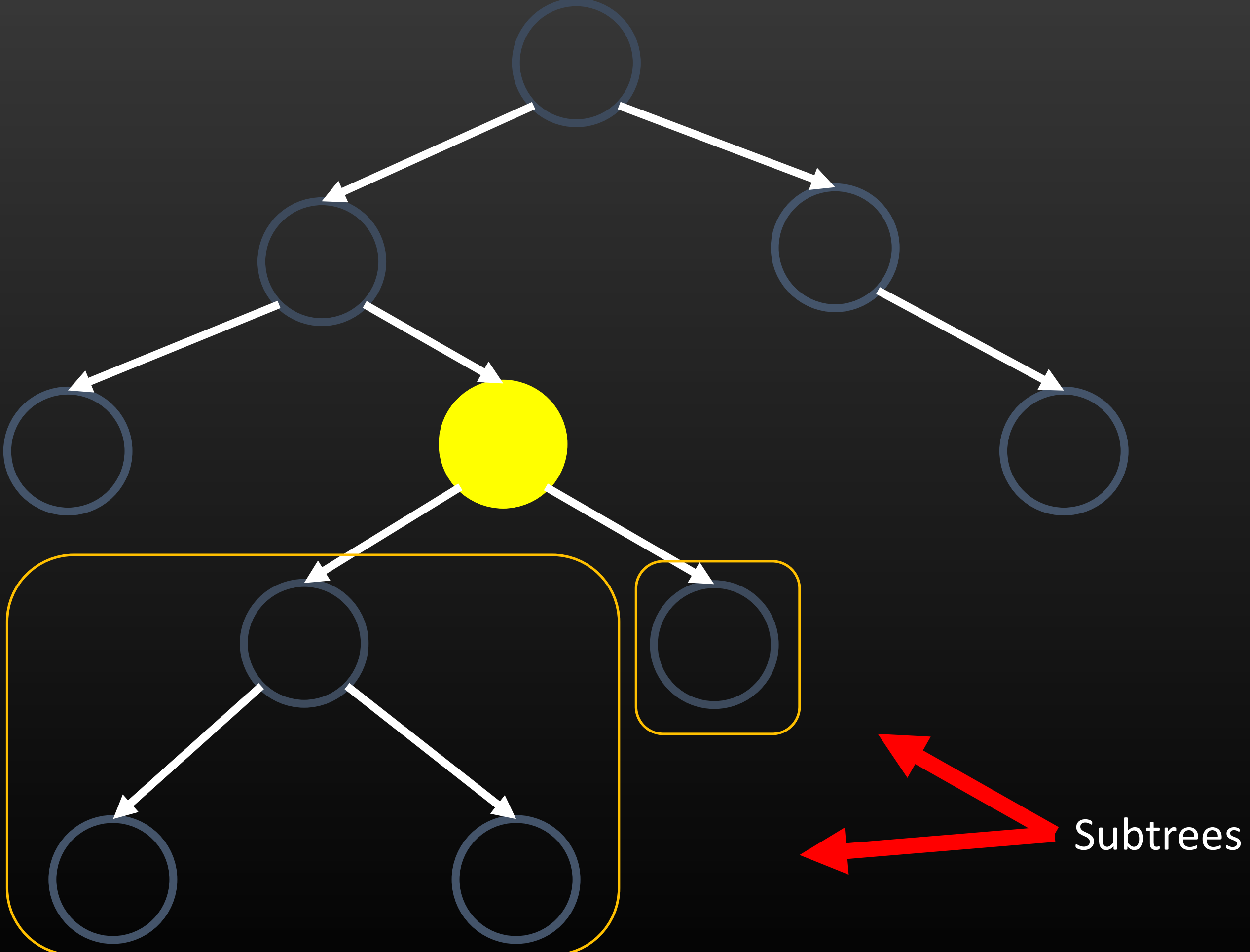
Terms on directed tree



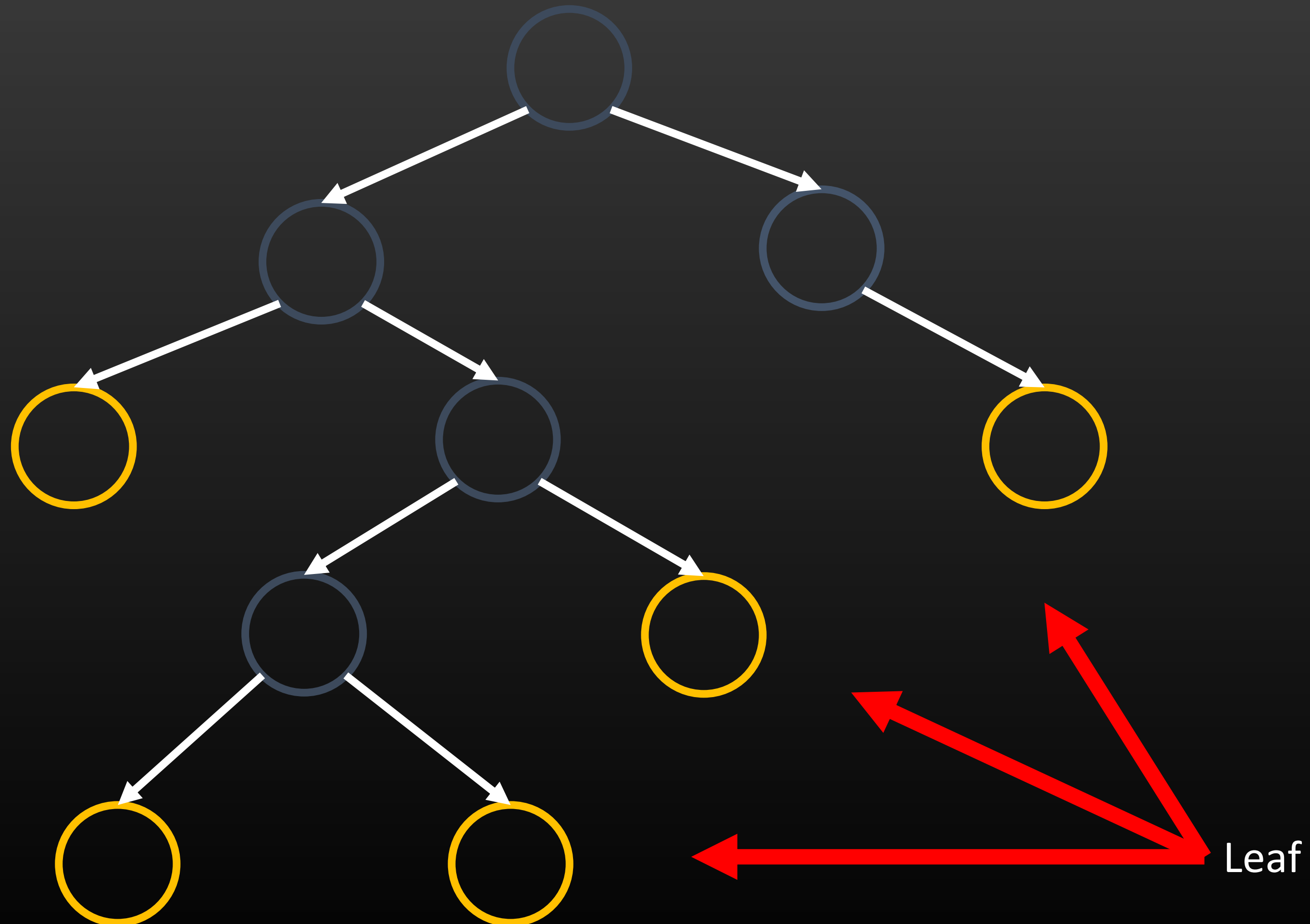
Terms on directed tree



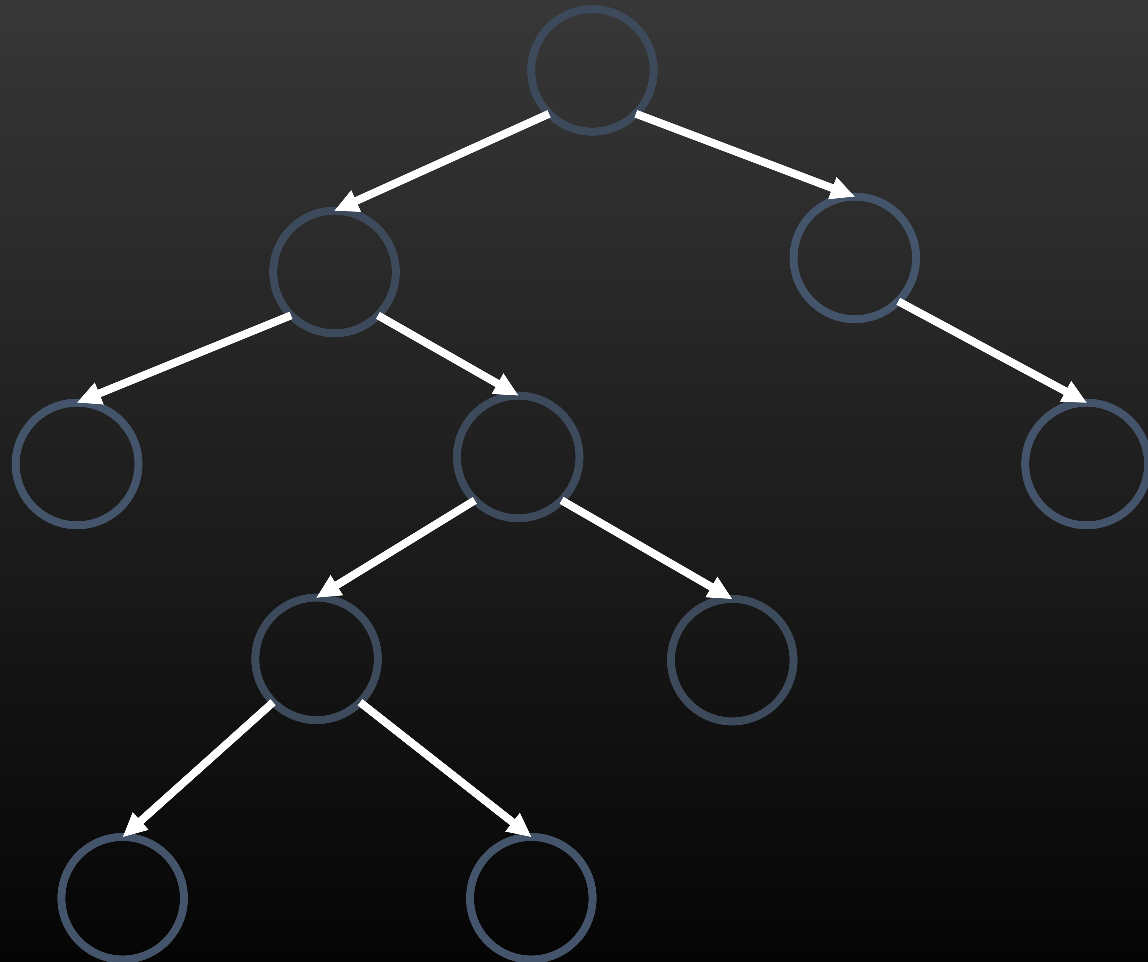
Terms on directed tree



Terms on directed tree

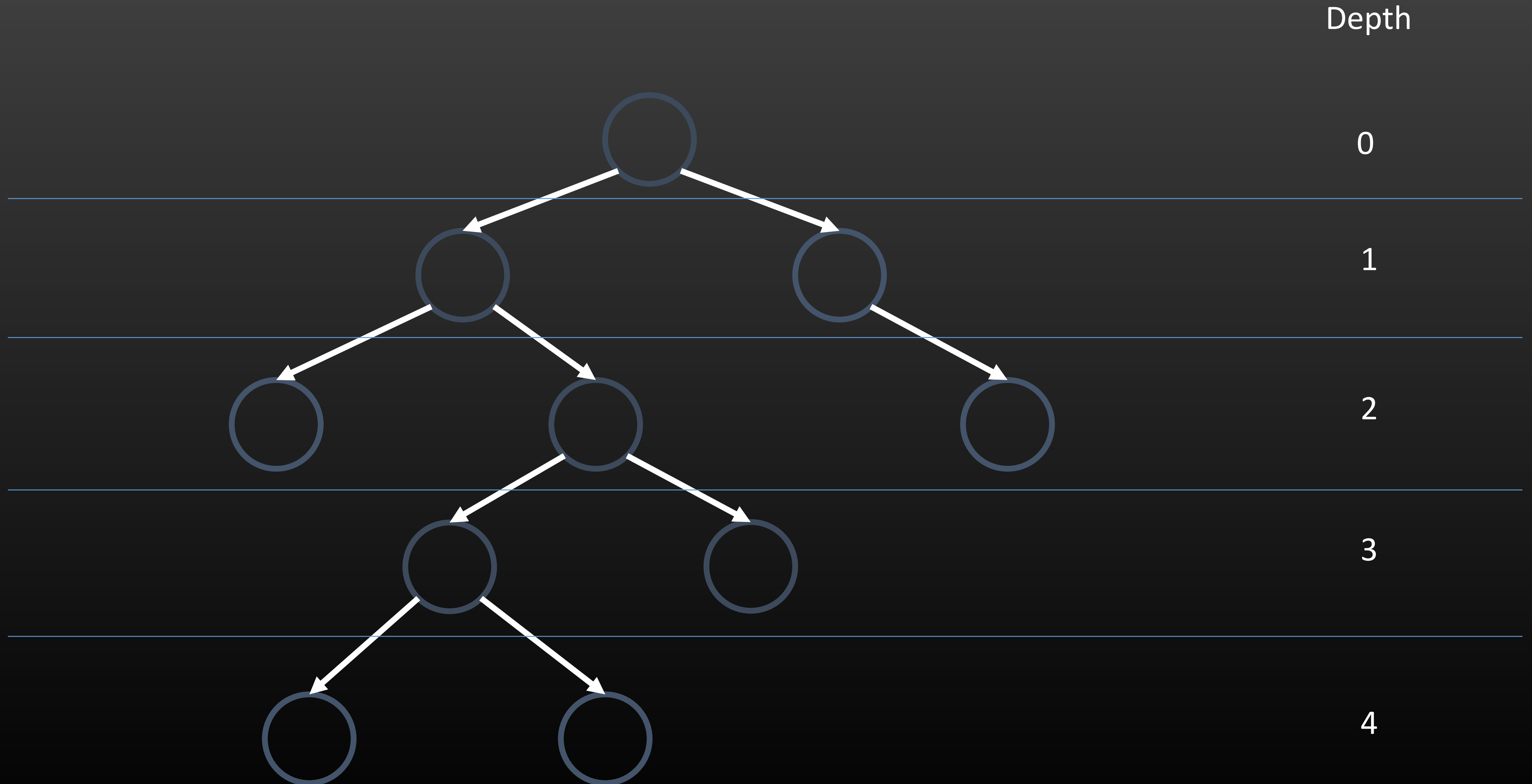


Terms on directed tree



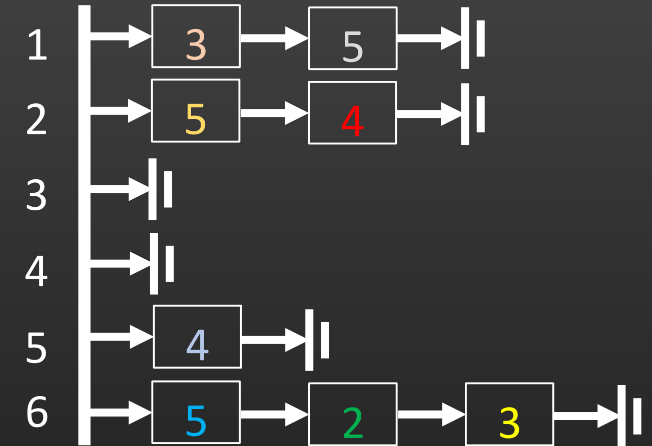
Height = 4

Terms on directed tree



Tree Implementation

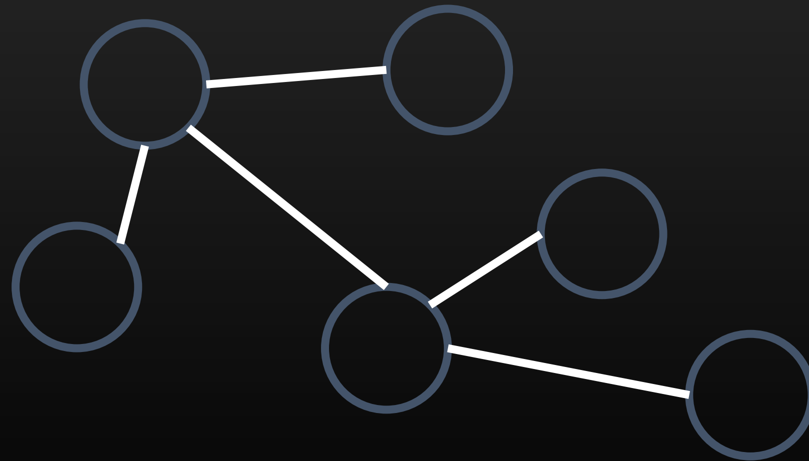
- Trees are also graphs
- Use graph representations
  - Adjacency matrix
  - Adjacency lists
  - Edge list



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

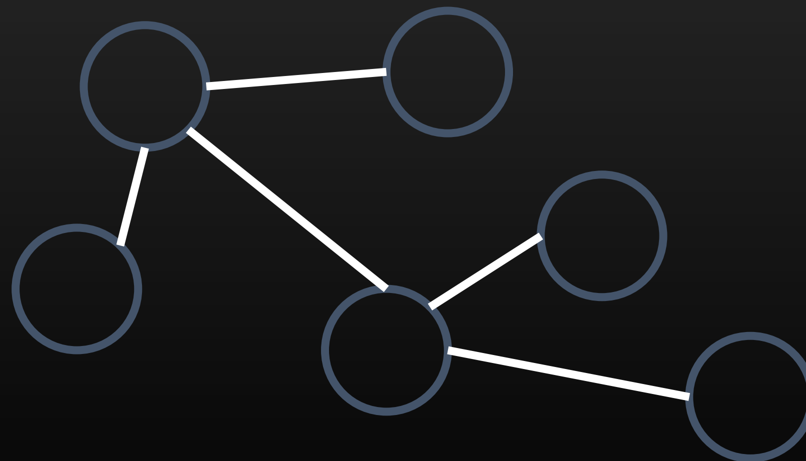
Trees - Usage

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



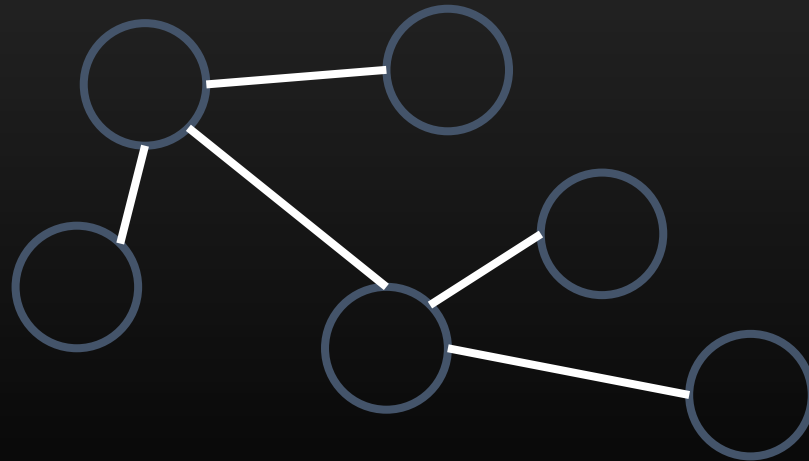
Trees - Usage

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



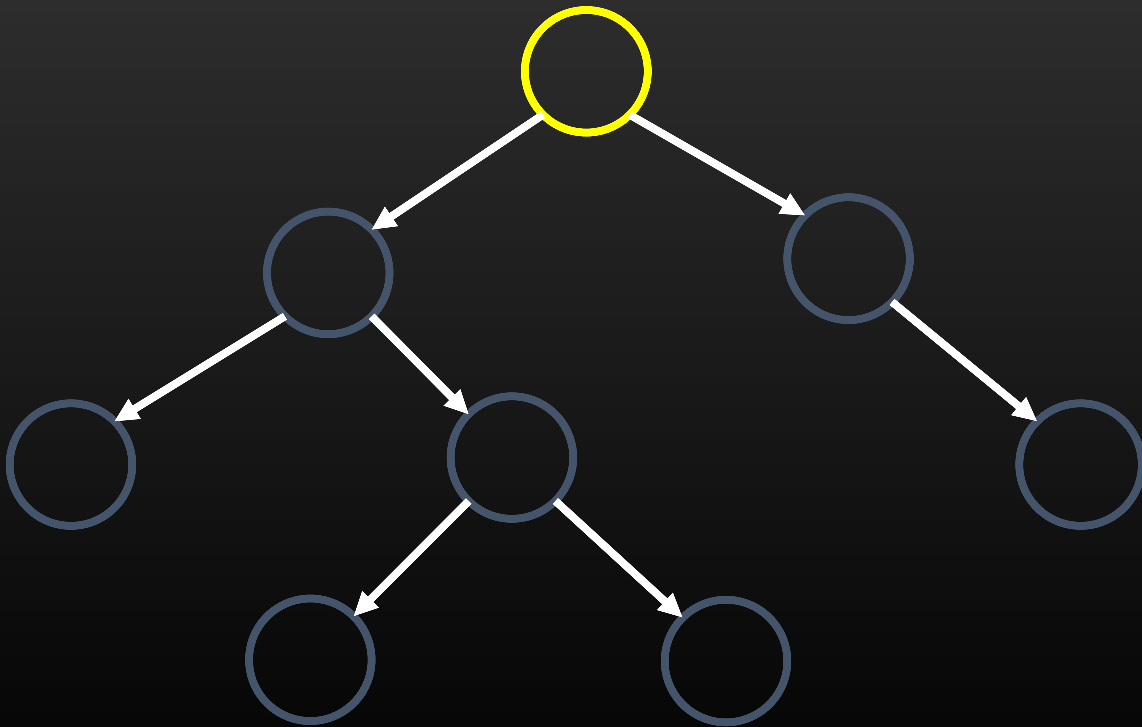
Trees - Usage

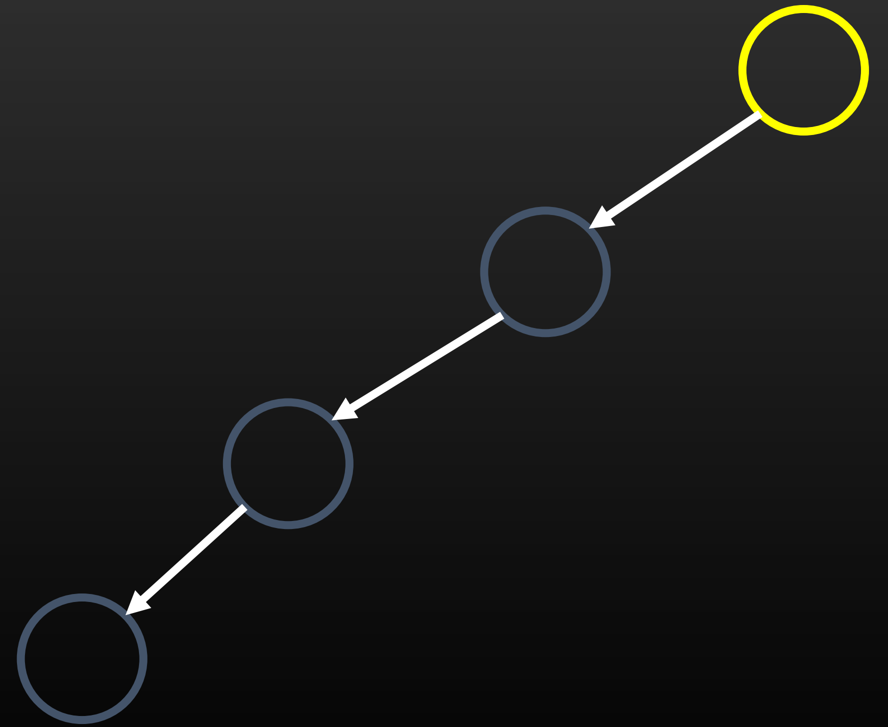
- Can also be used as data structures by storing data in a specific way
  - Binary Search Trees
  - Heaps
  - Tries
  - Segment Trees
  - Suffix Trees
  - …



Binary Tree

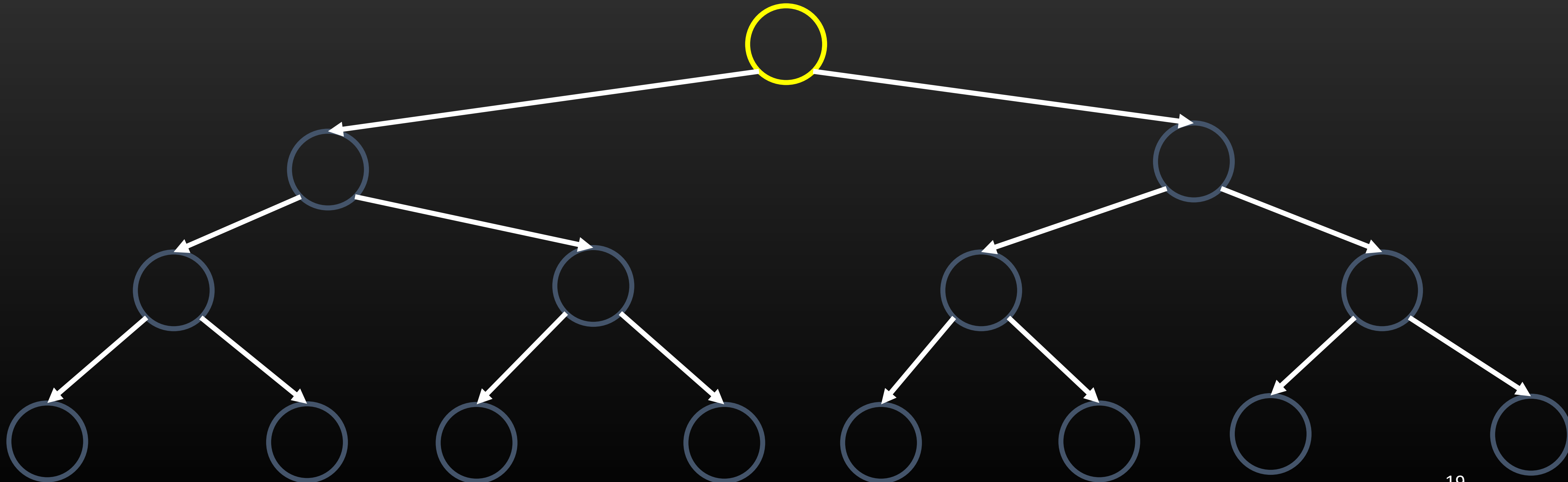
- A rooted tree which all vertices (nodes) have at most 2 children





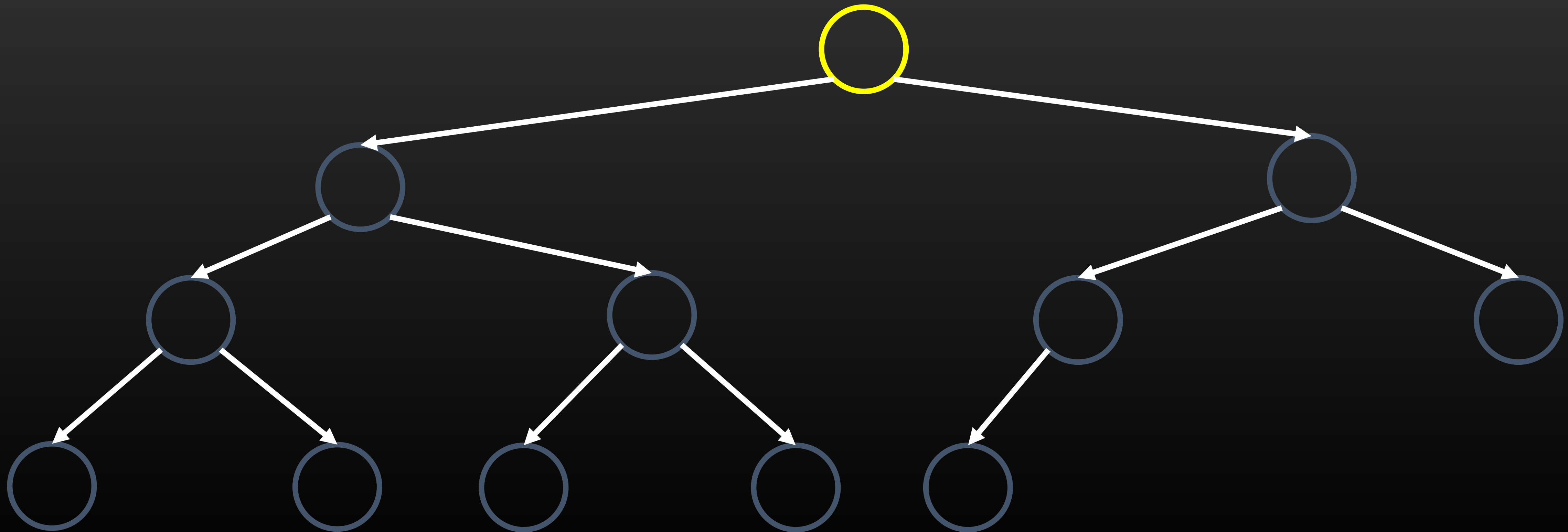
Perfect Binary Tree

- 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;
}
```

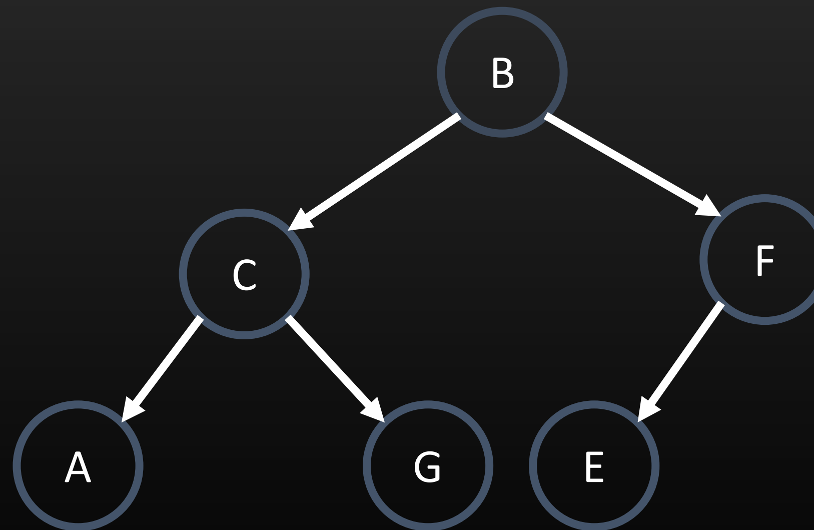
- May be difficult to trace
- Useful when implementing balanced BST (e.g. AVL Tree)

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

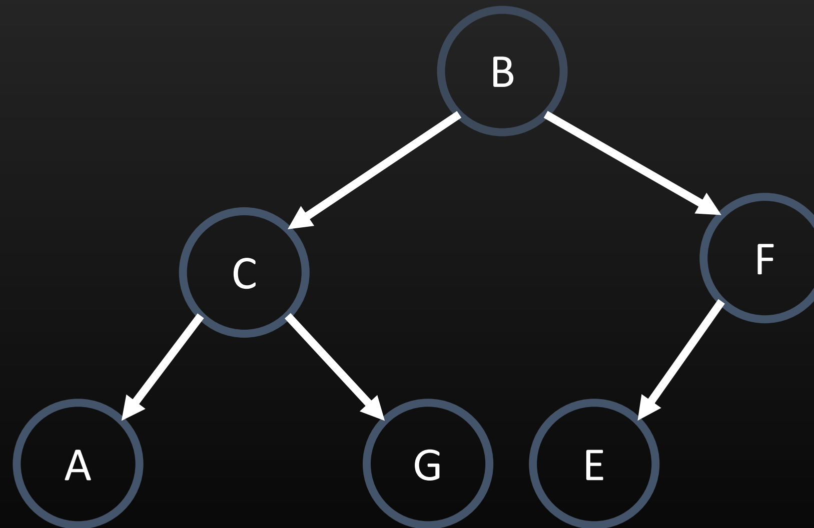
Index	0	1	2	3	4	5	6
A[]		B	C	F	A	G	E



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)
    }
```

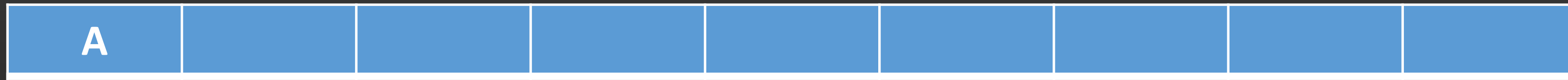
Depth First Search

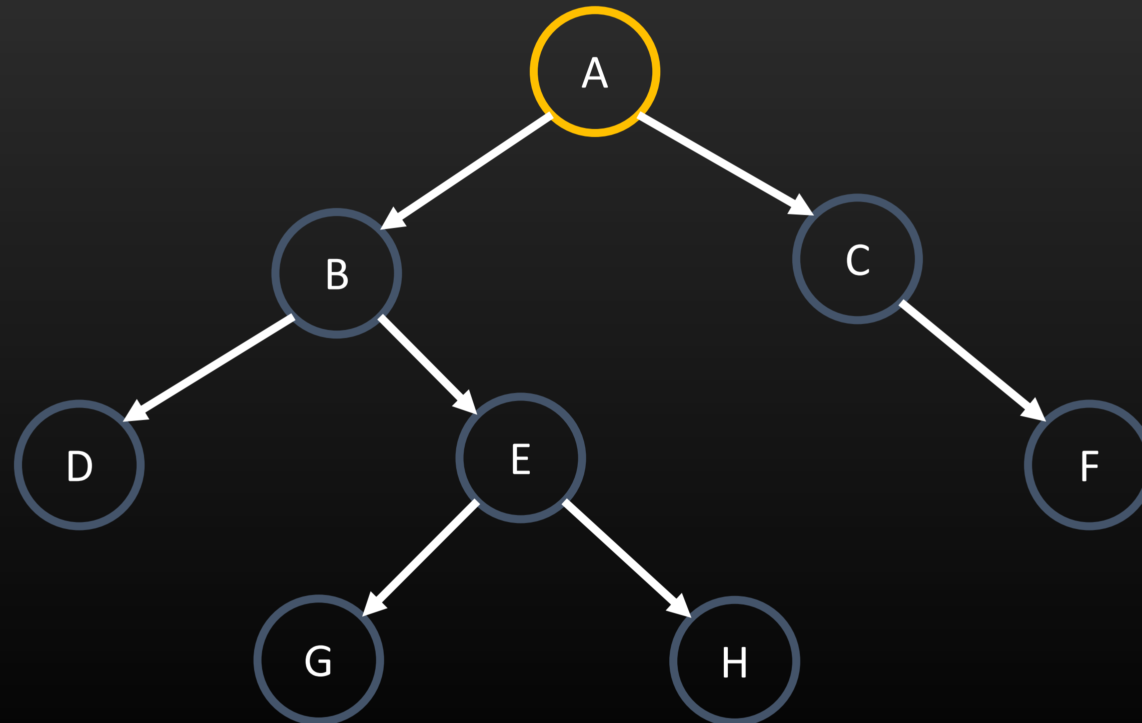
- Usually we only want to process every node exactly once
- For a binary tree, we have three orders for tree traversal:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

Depth First Search – Pre-order

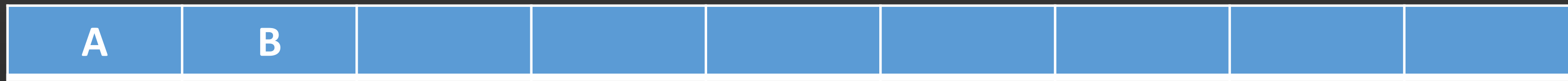
- Starting from the root
- For the traversal at node N:
  - Process the node N
  - Recursively traverse its left subtree
  - Recursively traverse its right subtree

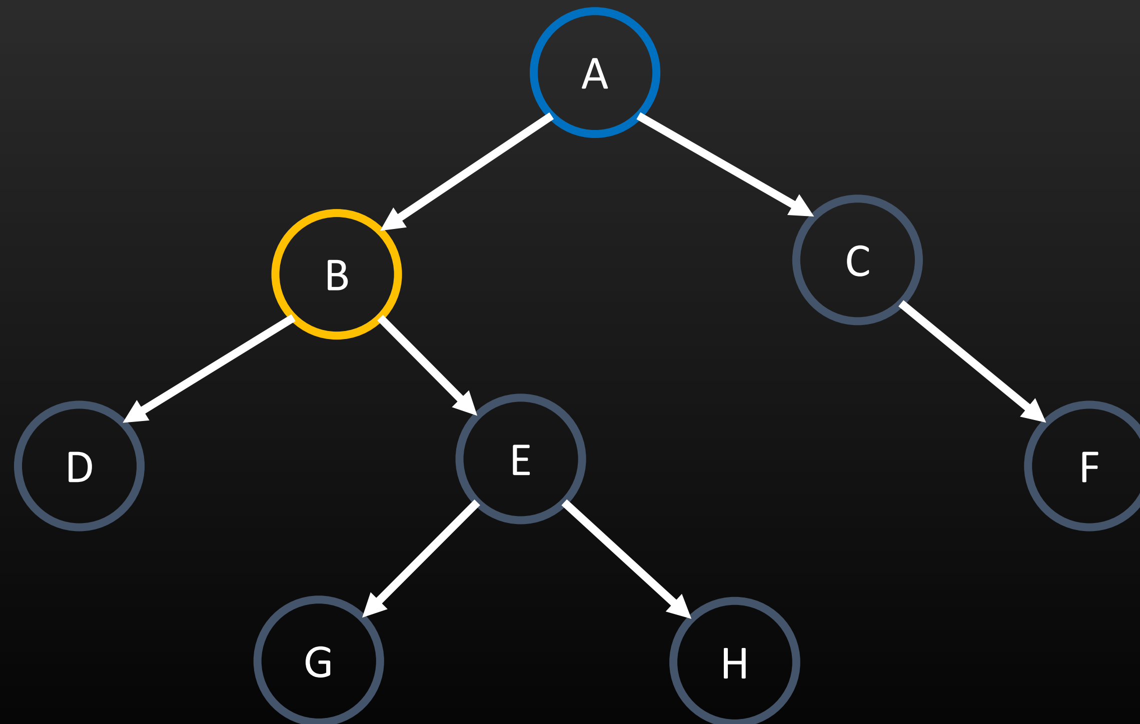
Depth First Search – Pre-order



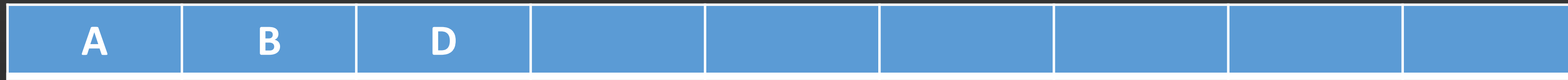


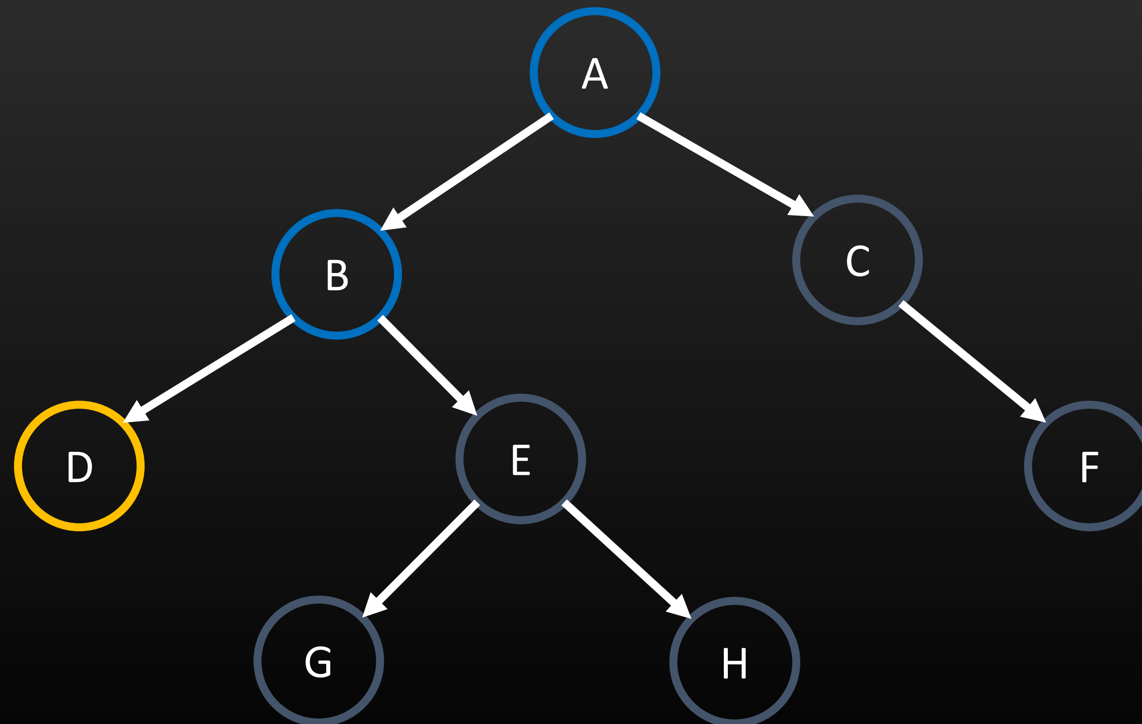
Depth First Search – Pre-order



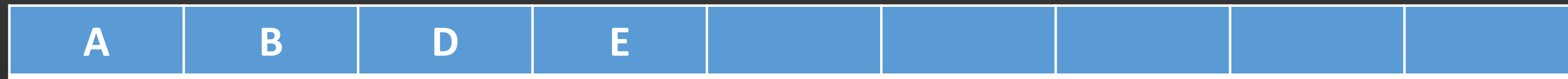


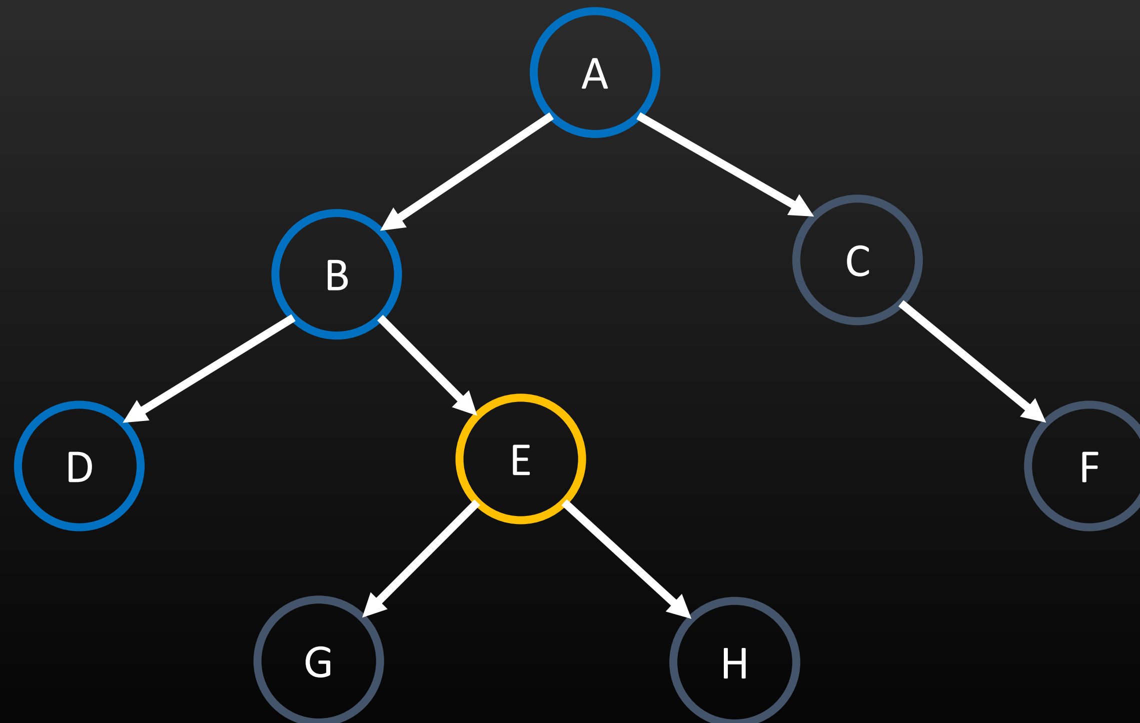
Depth First Search – Pre-order



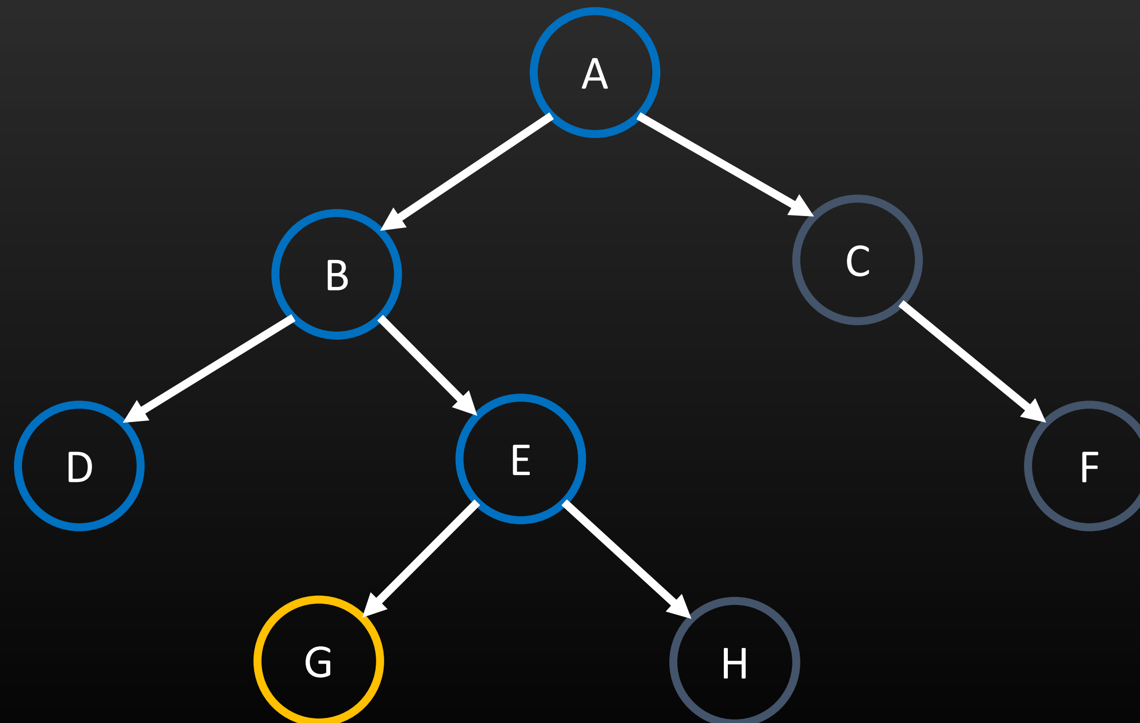


Depth First Search – Pre-order

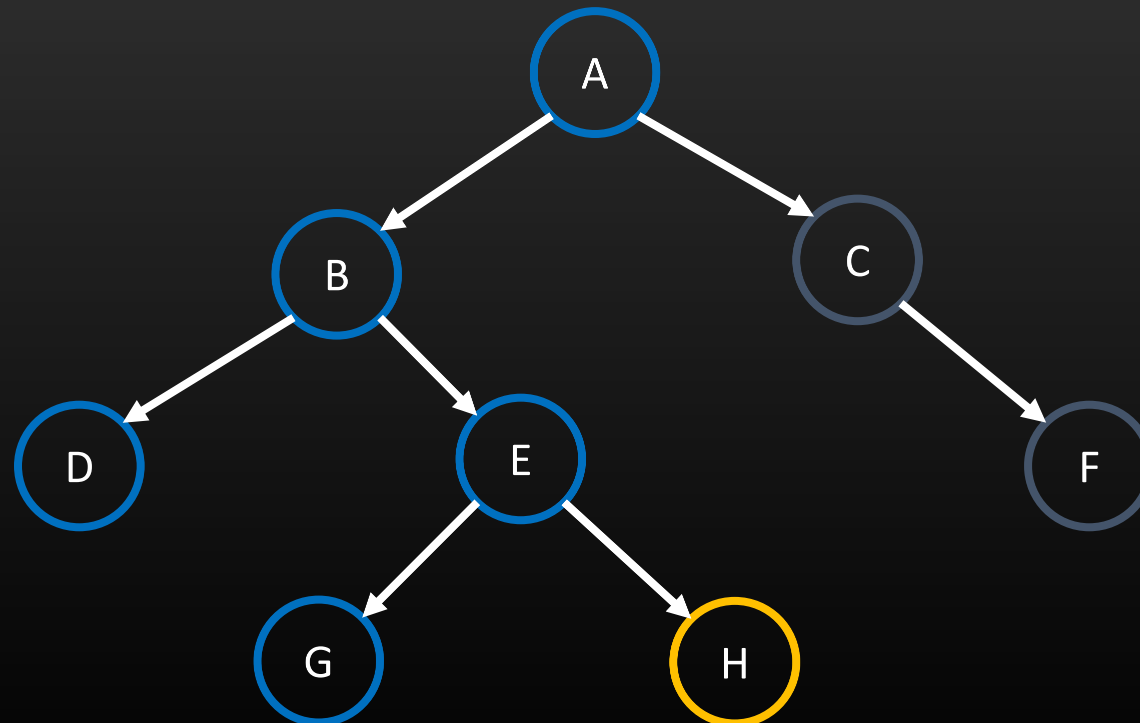




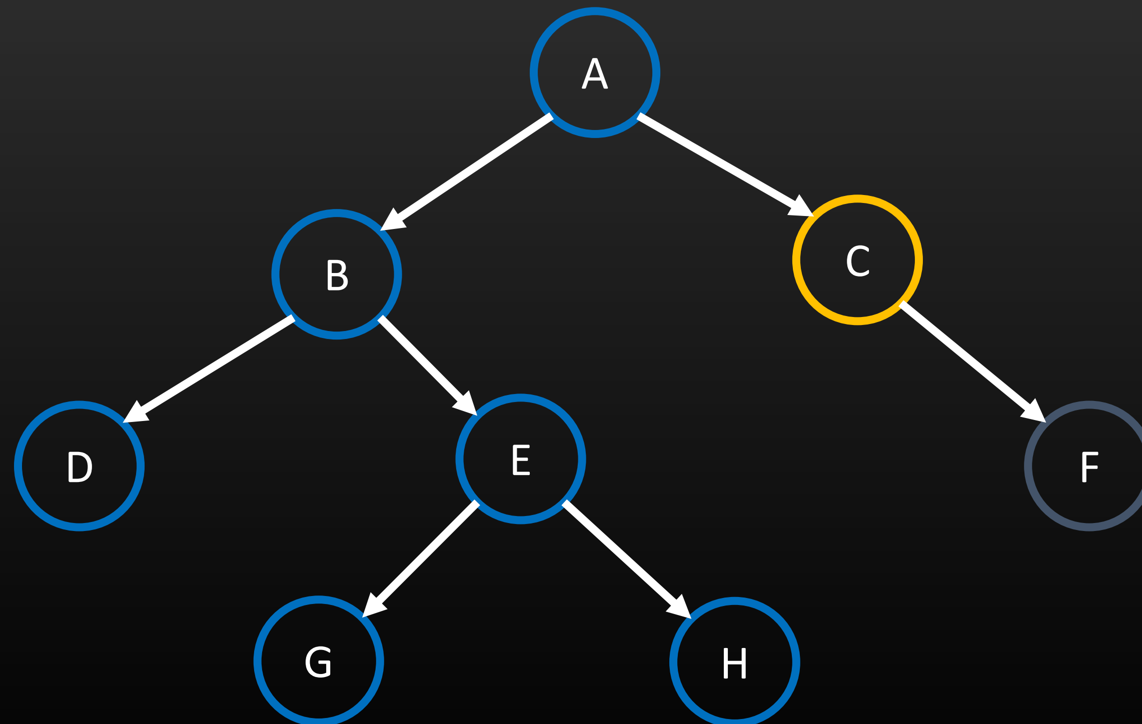
Depth First Search – Pre-order



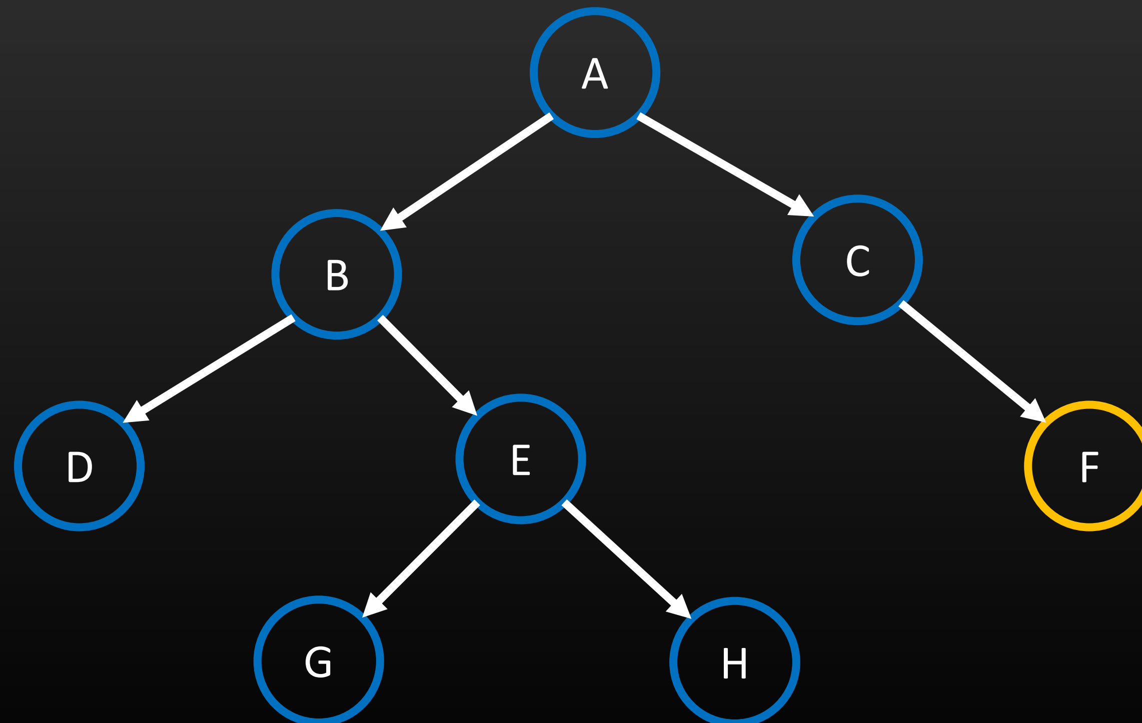
Depth First Search – Pre-order



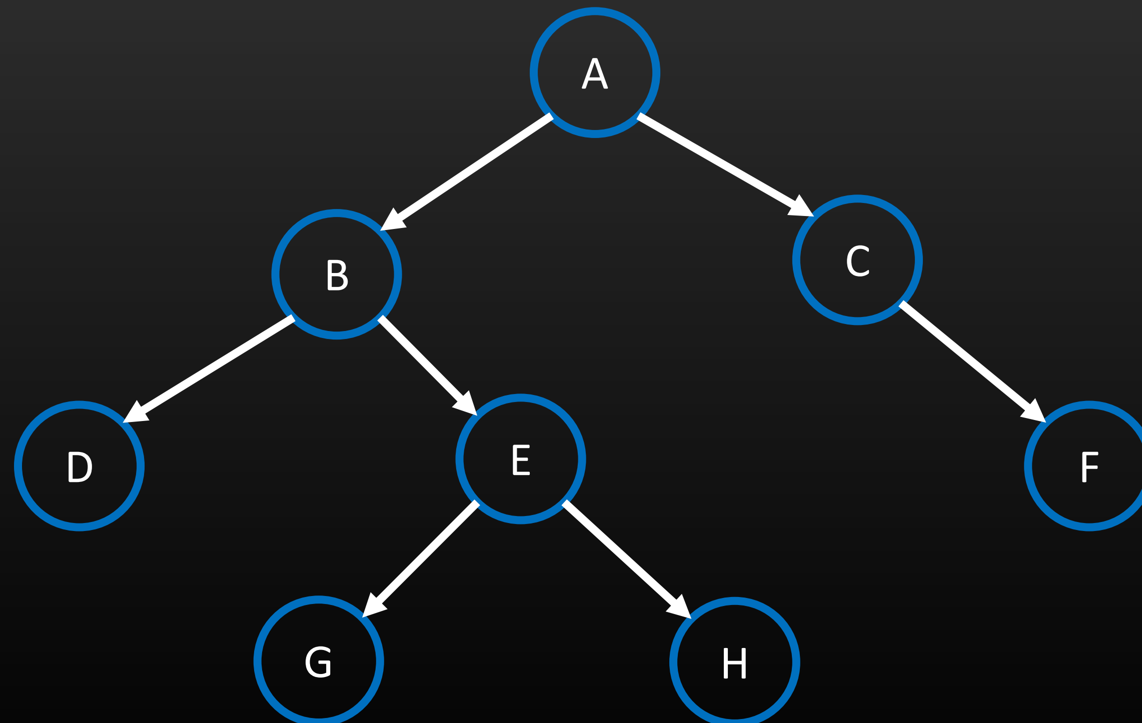
Depth First Search – Pre-order



Depth First Search – Pre-order



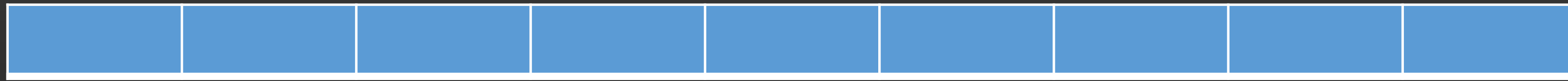
Depth First Search – Pre-order

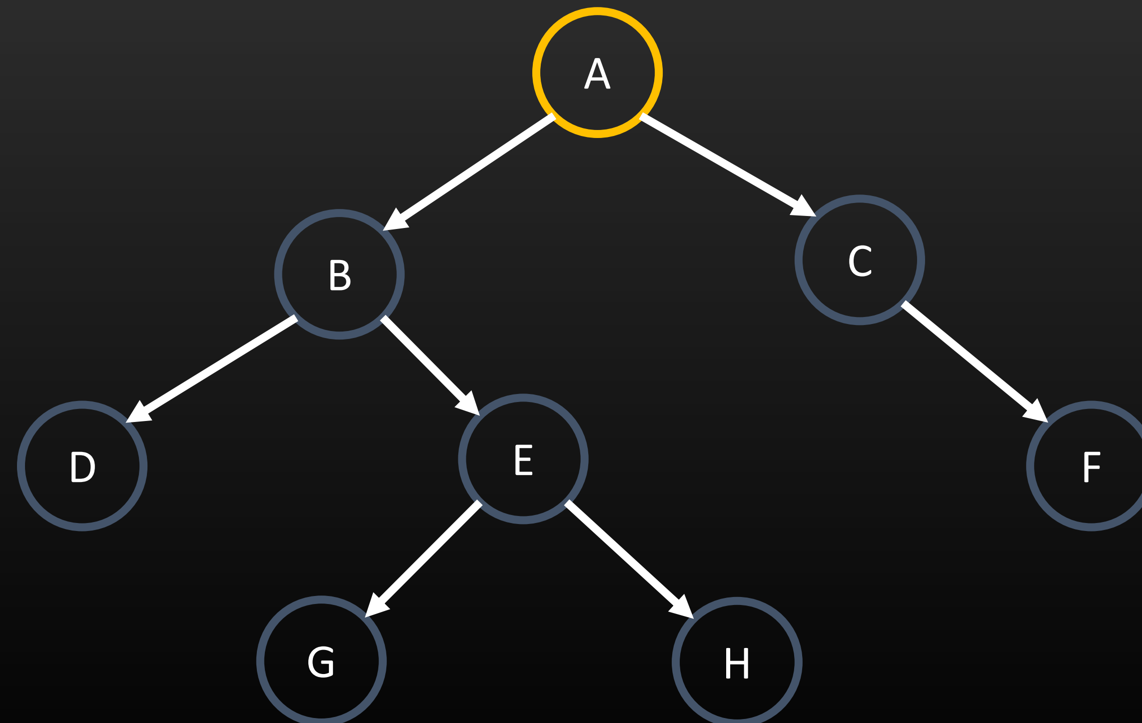


Depth First Search – In-order

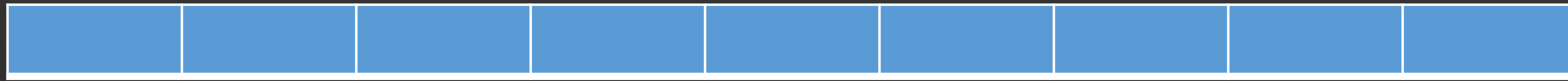
- Starting from the root
- For the traversal at node N:
  - Recursively traverse its left subtree
  - Process the node N
  - Recursively traverse its right subtree

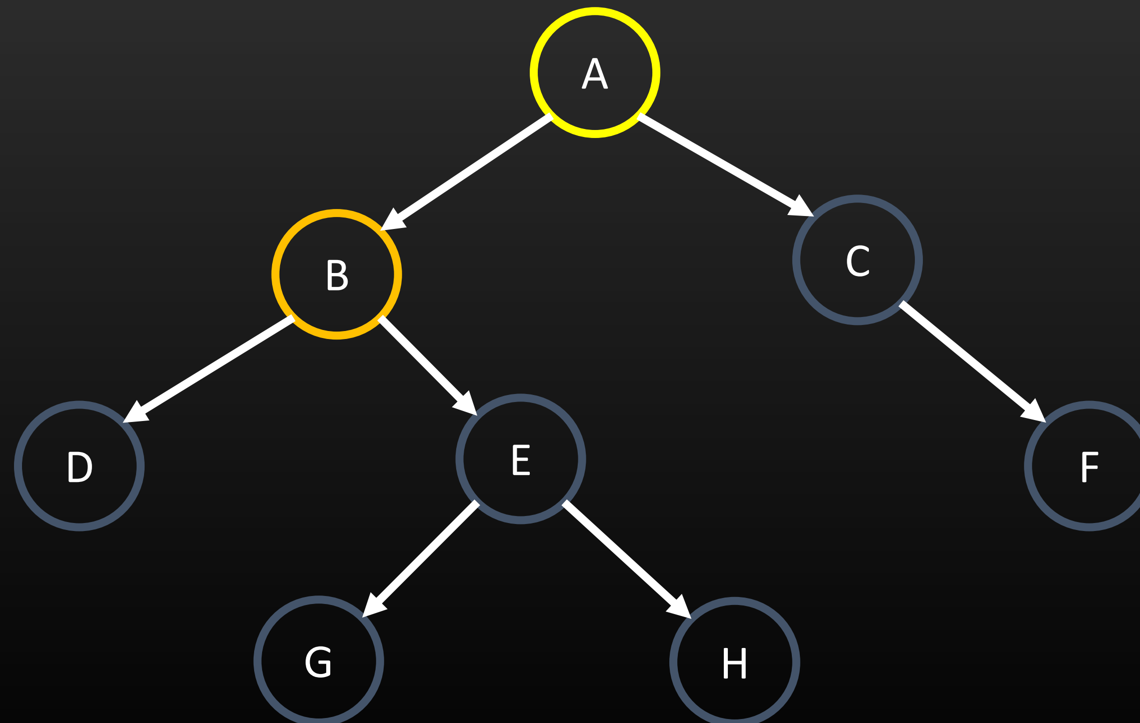
Depth First Search – In-order



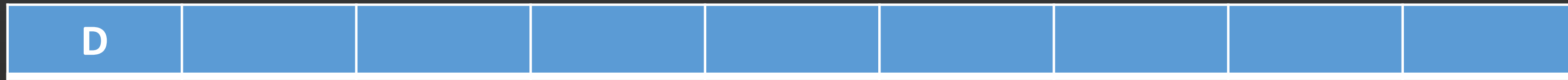


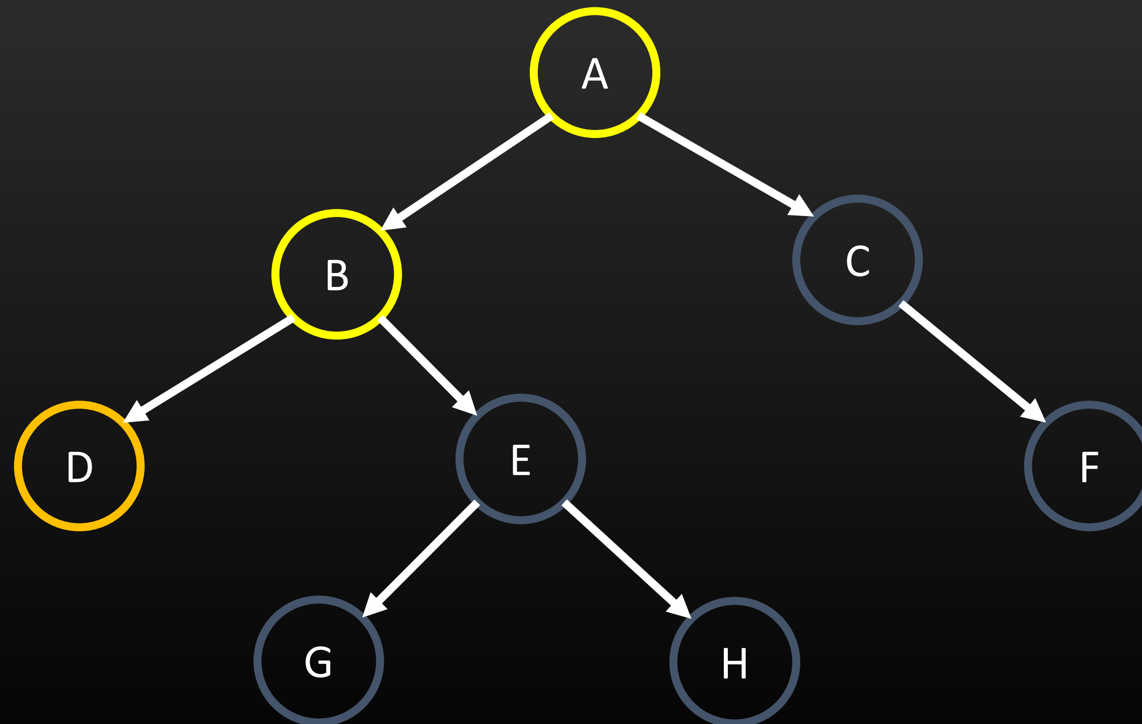
Depth First Search – In-order



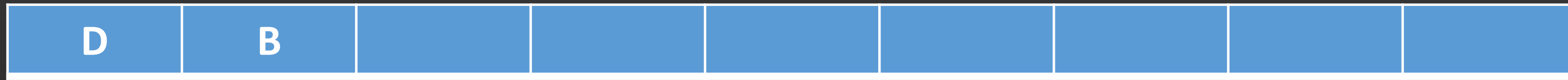


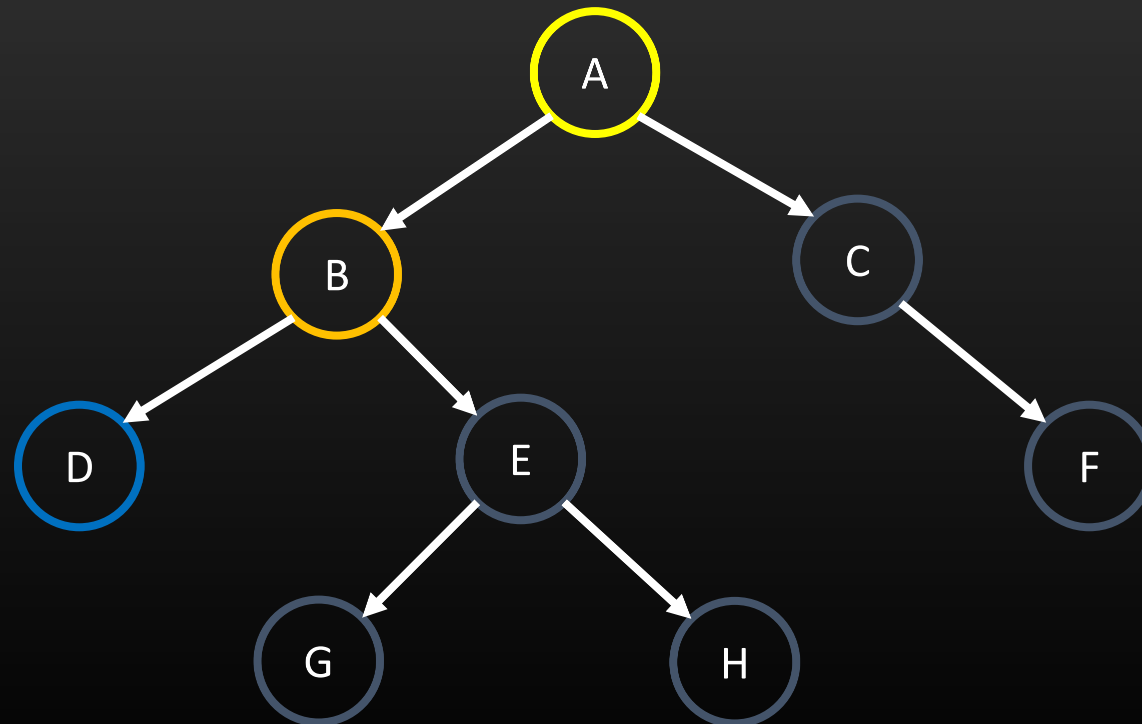
Depth First Search – In-order



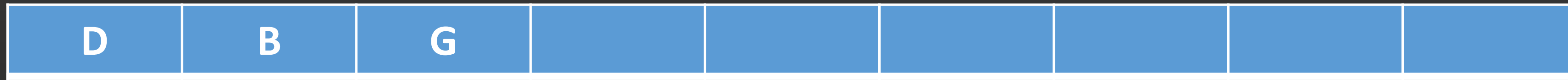


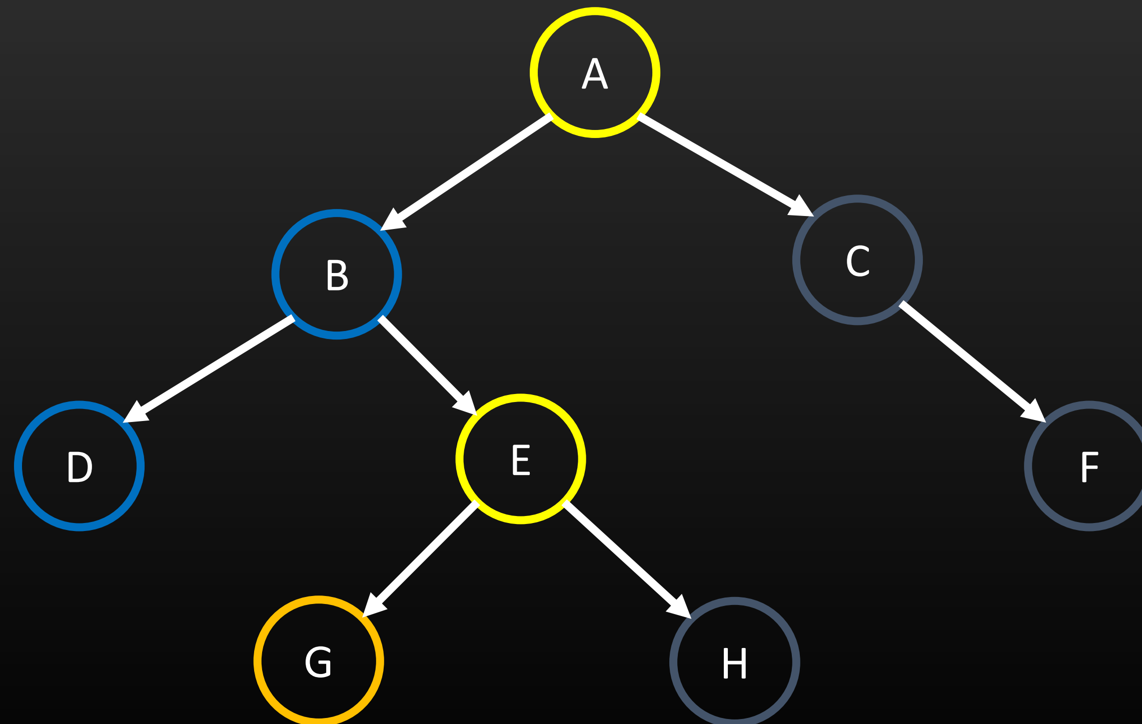
Depth First Search – In-order



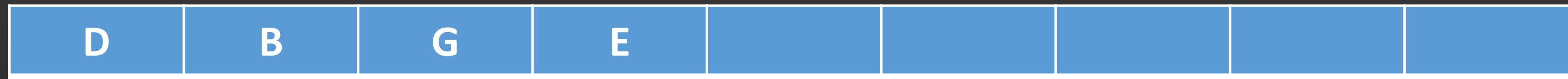


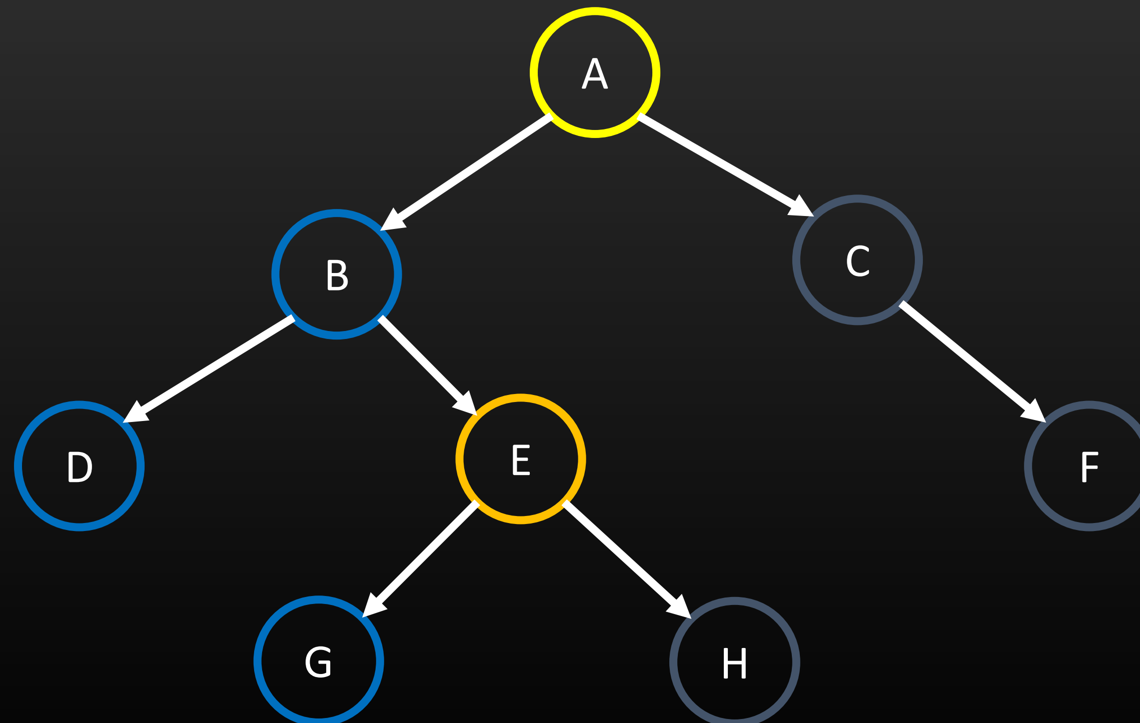
Depth First Search – In-order



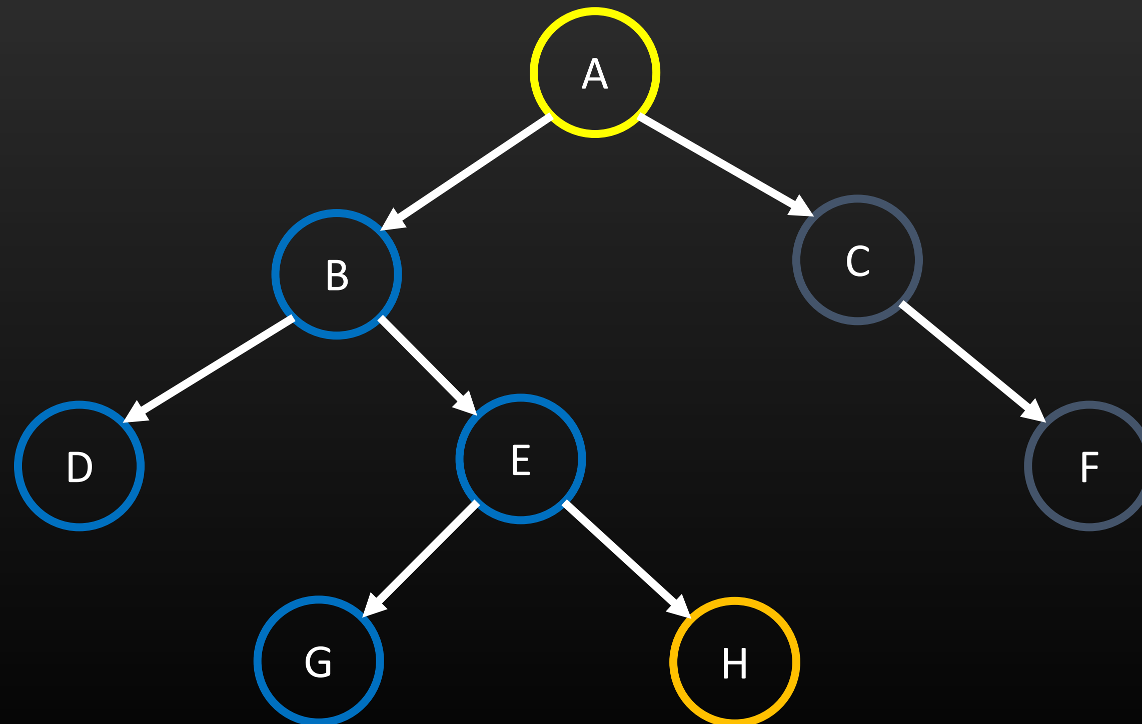


Depth First Search – In-order

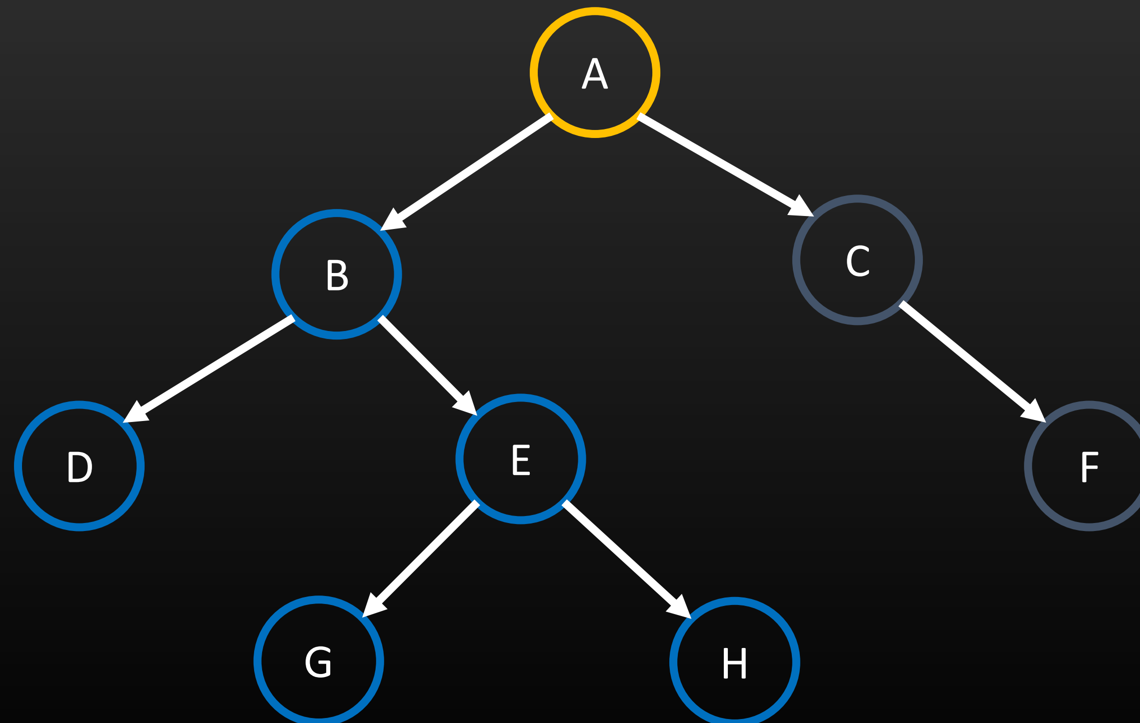




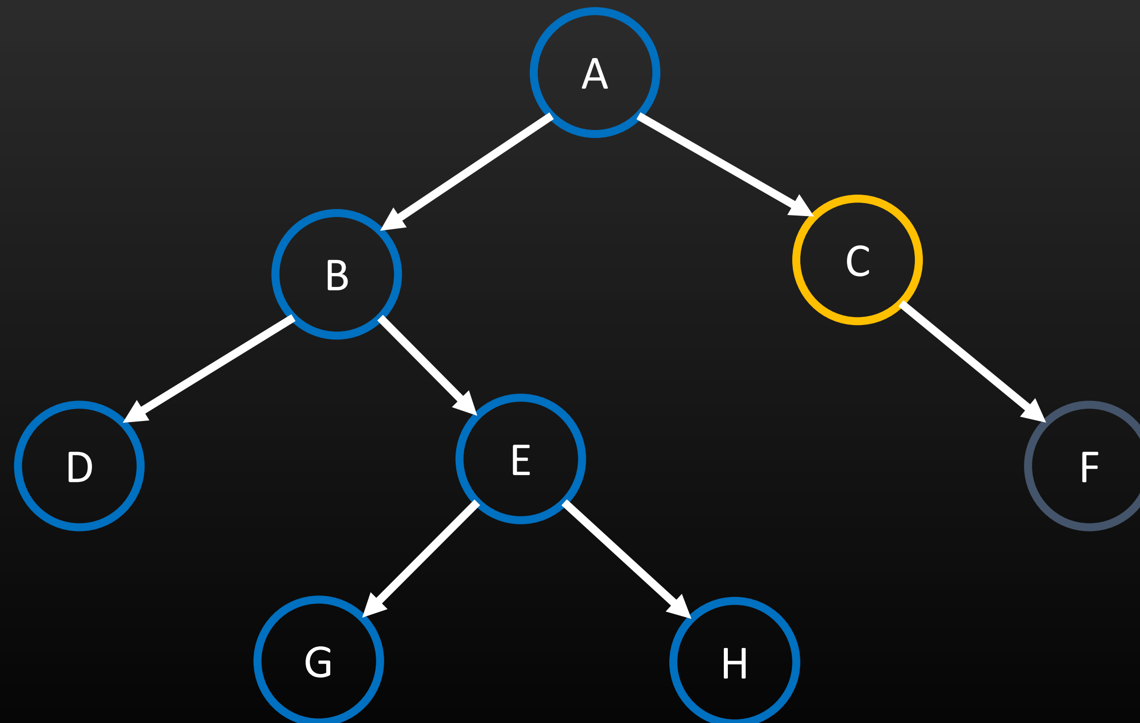
Depth First Search – In-order



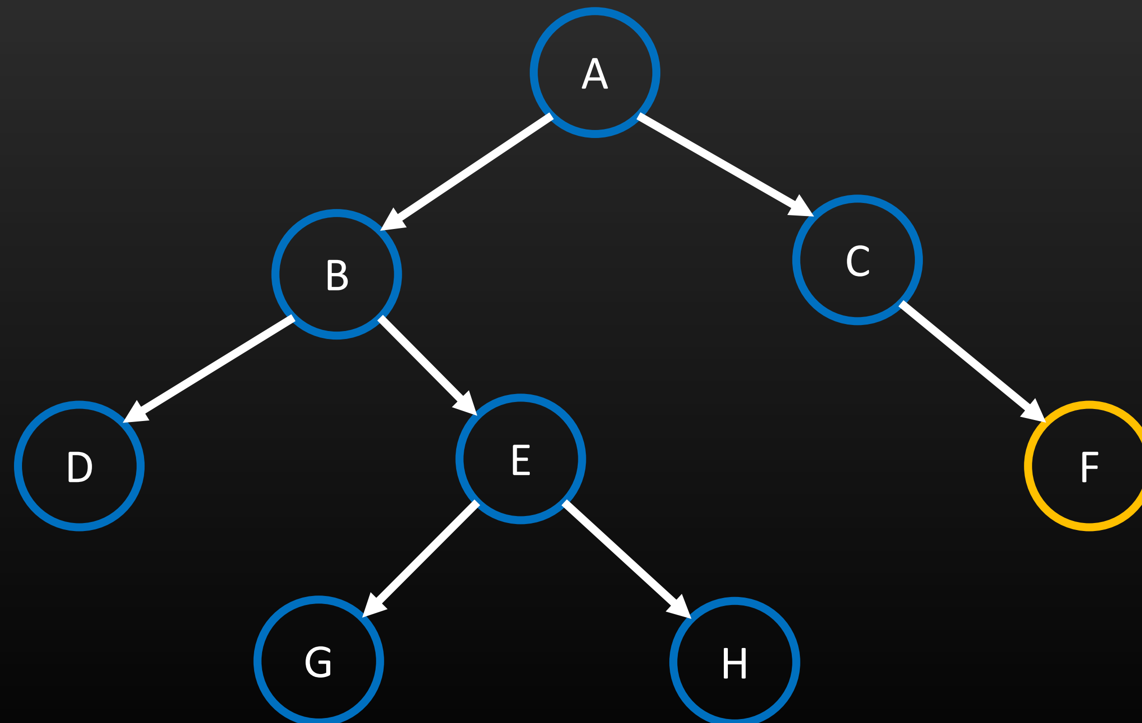
Depth First Search – In-order



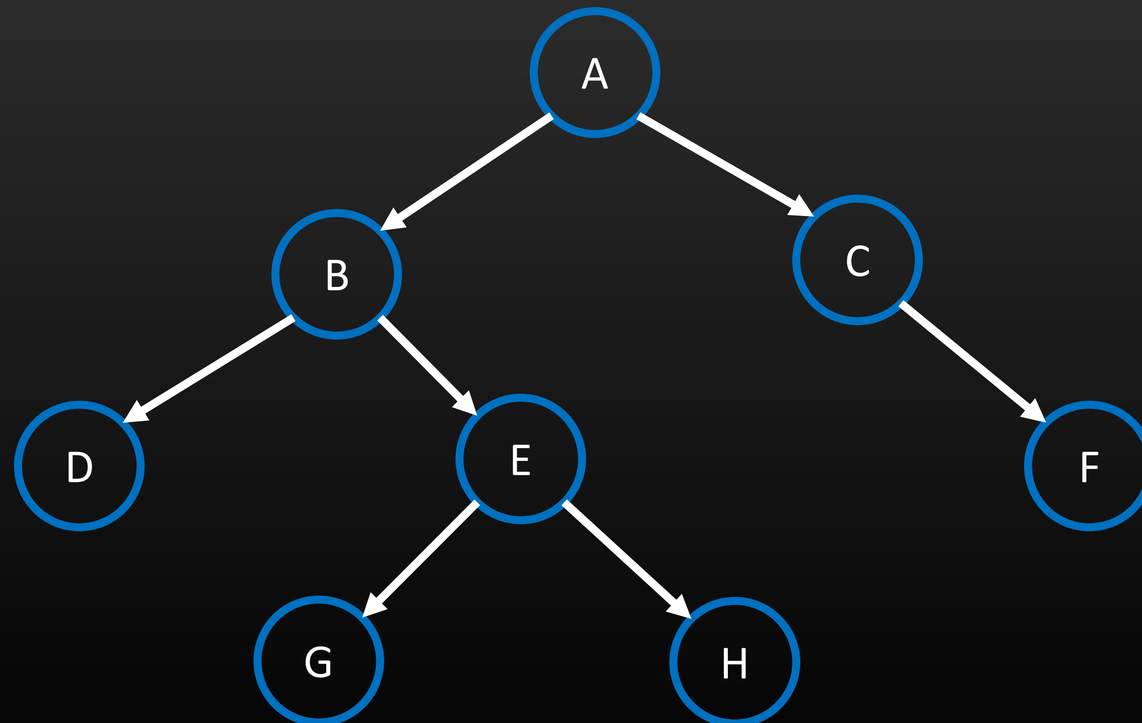
Depth First Search – In-order



Depth First Search – In-order



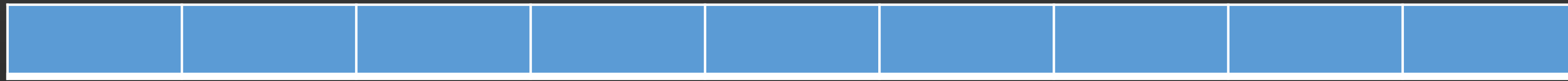
Depth First Search – In-order

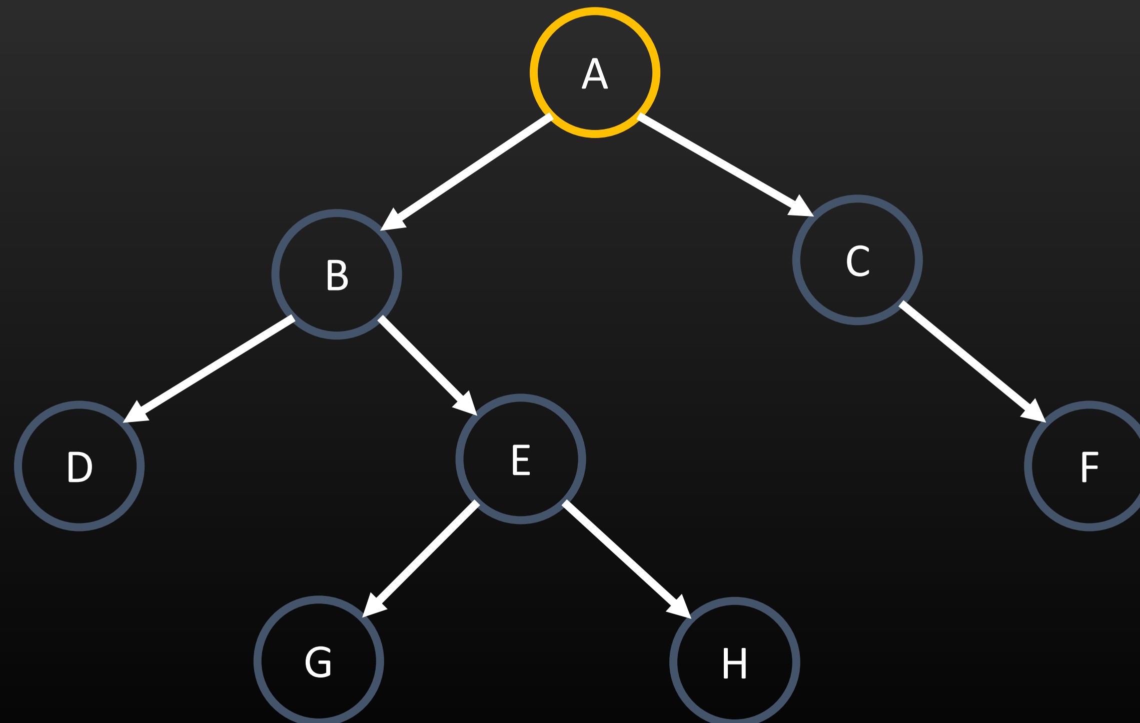


Depth First Search – Post-order

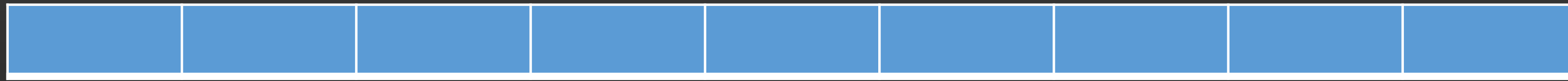
- Starting from the root
- For the traversal at node N:
  - Recursively traverse its left subtree
  - Recursively traverse its right subtree
  - Process the node N

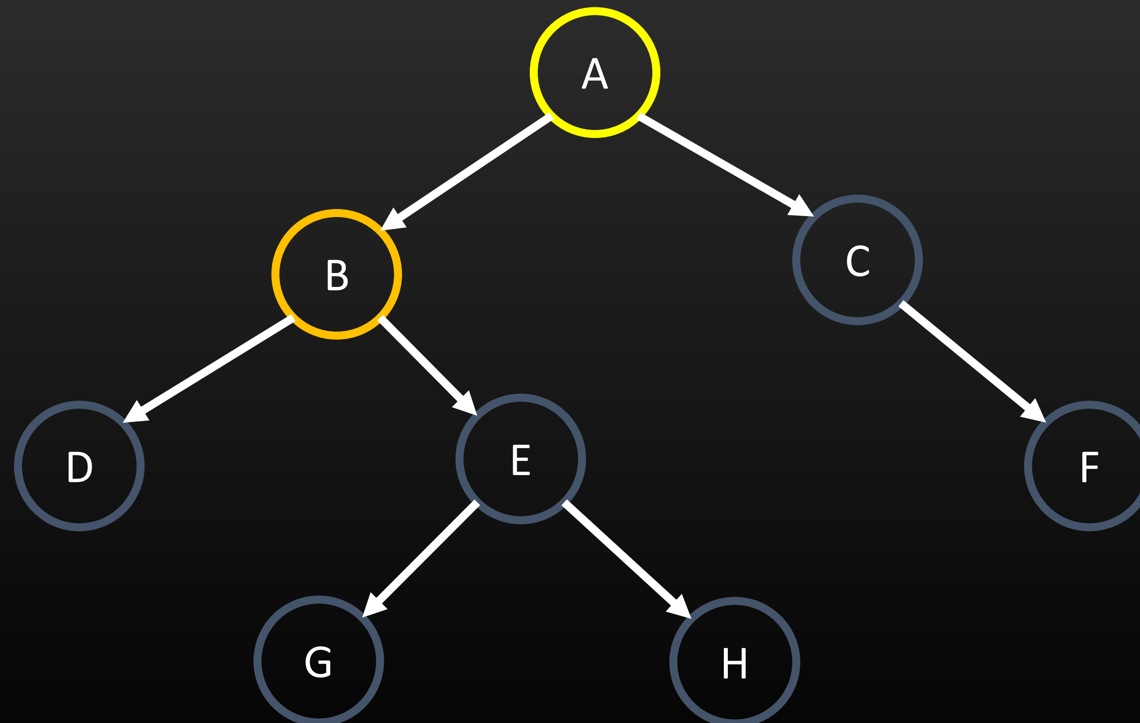
Depth First Search – Post-order



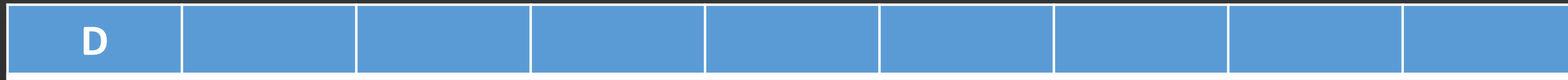


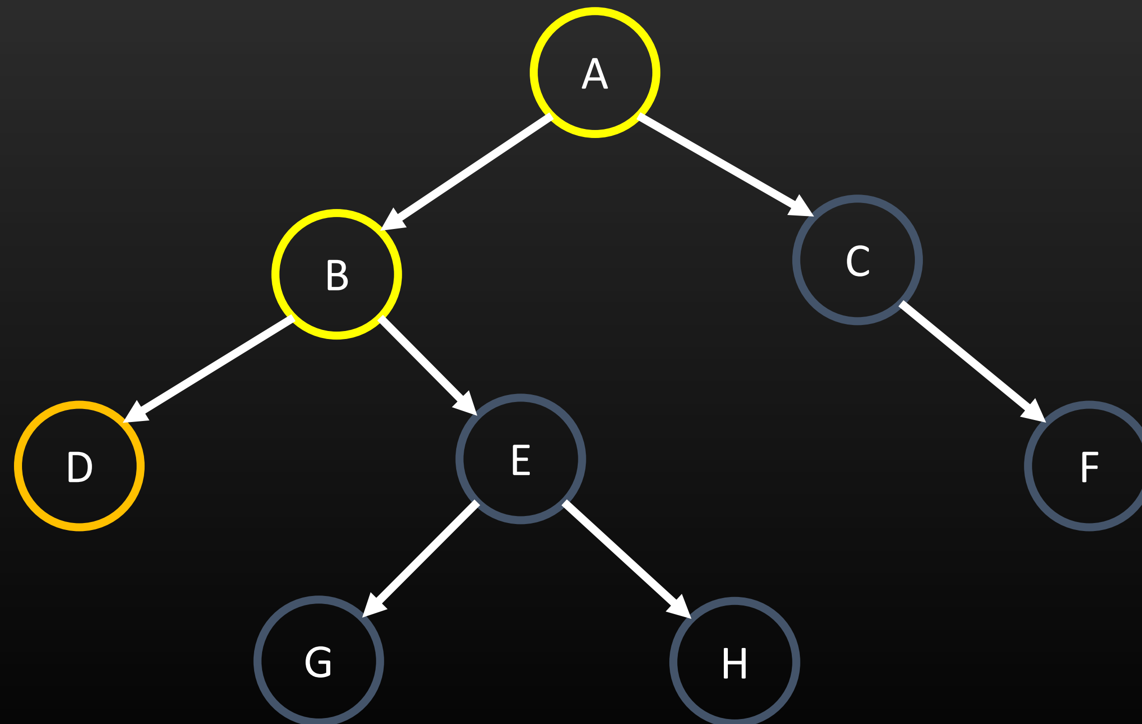
Depth First Search – Post-order



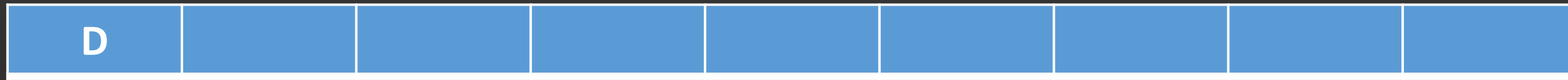


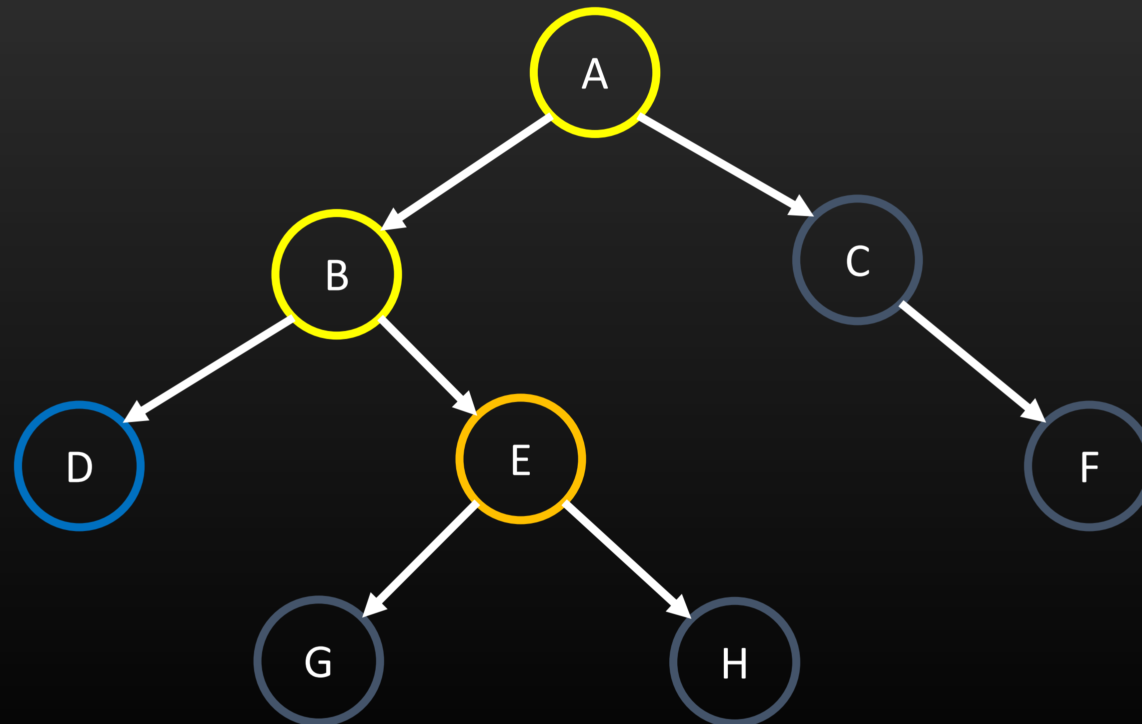
Depth First Search – Post-order



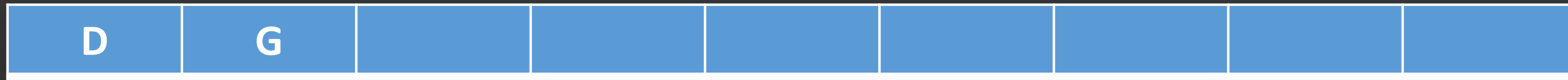


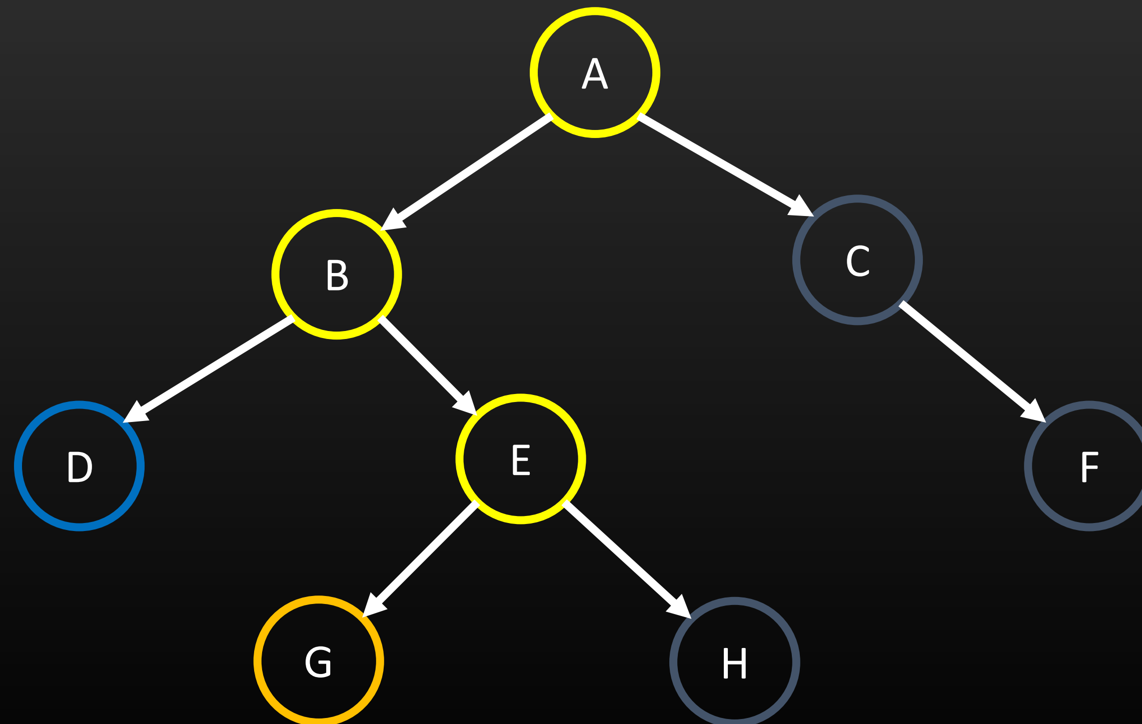
Depth First Search – Post-order



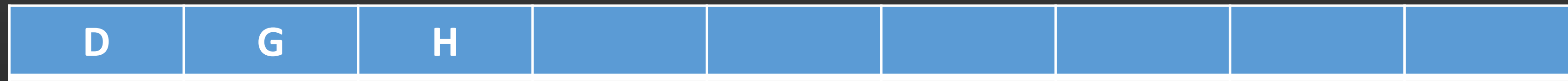


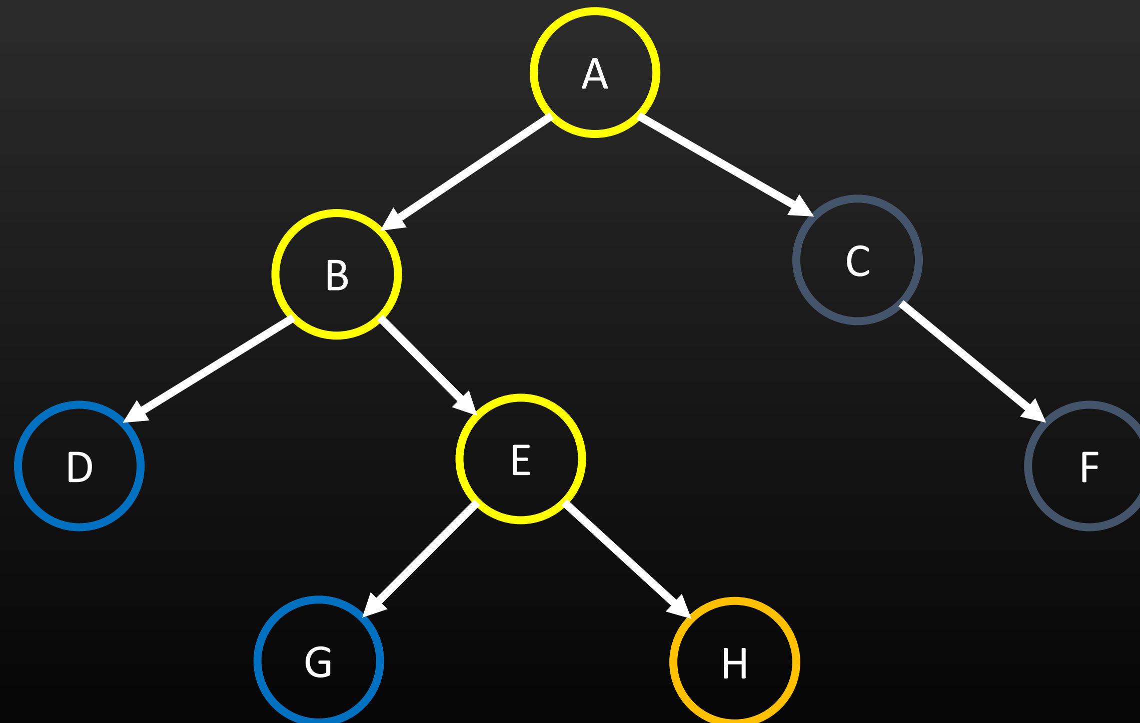
Depth First Search – Post-order



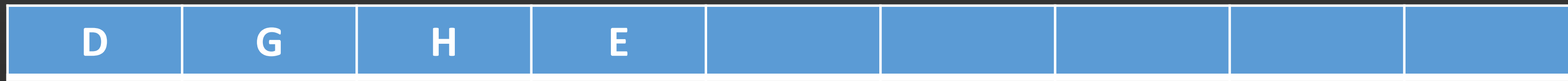


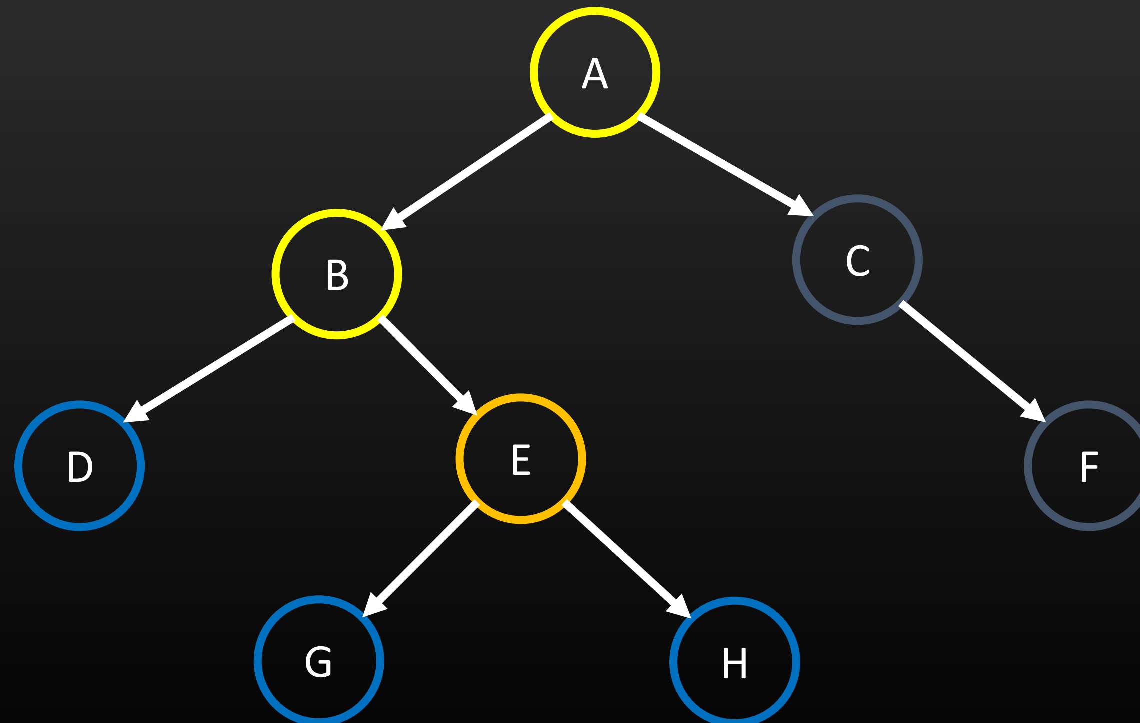
Depth First Search – Post-order



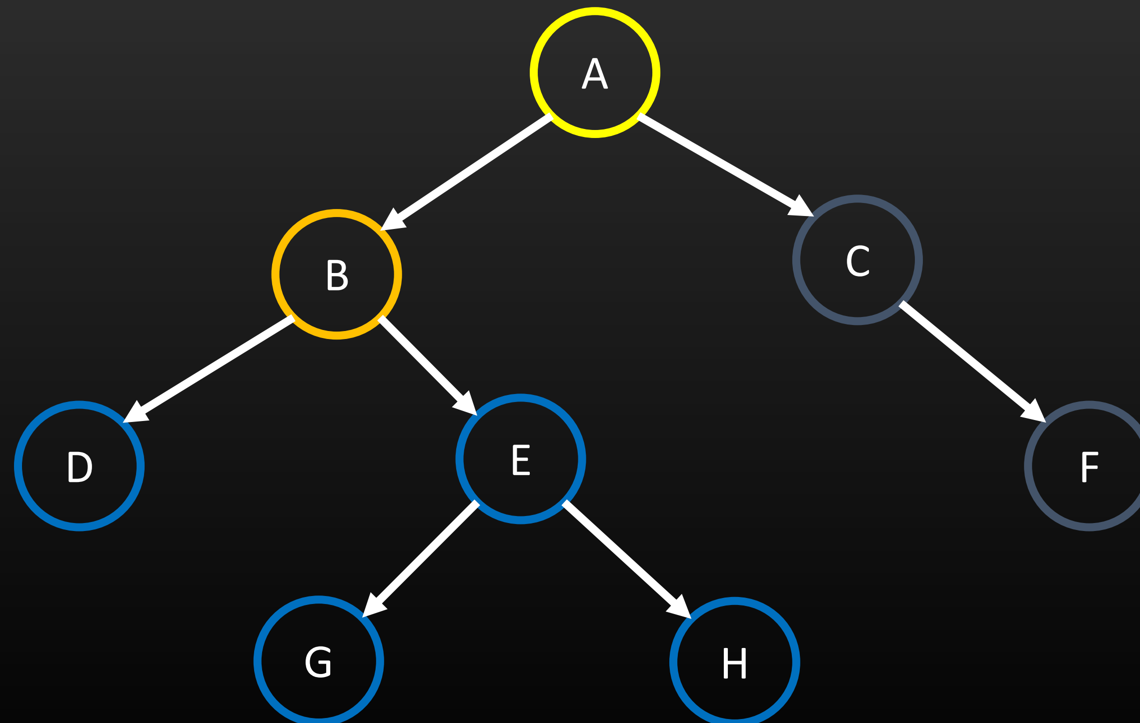


Depth First Search – Post-order



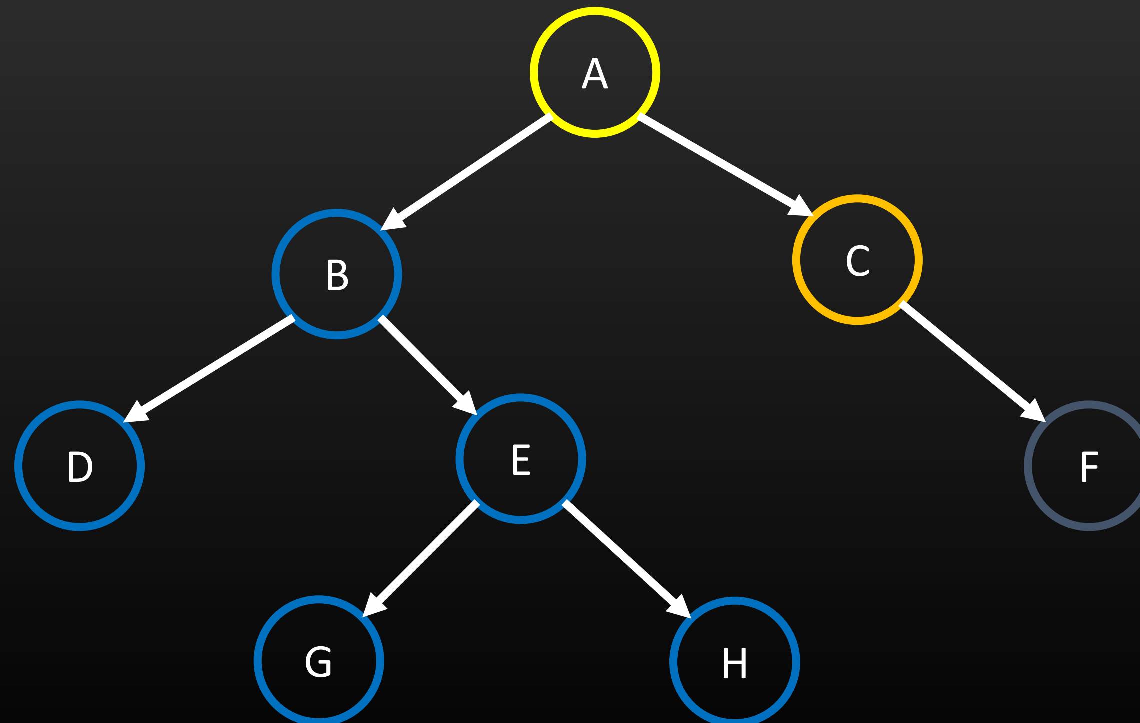


Depth First Search – Post-order

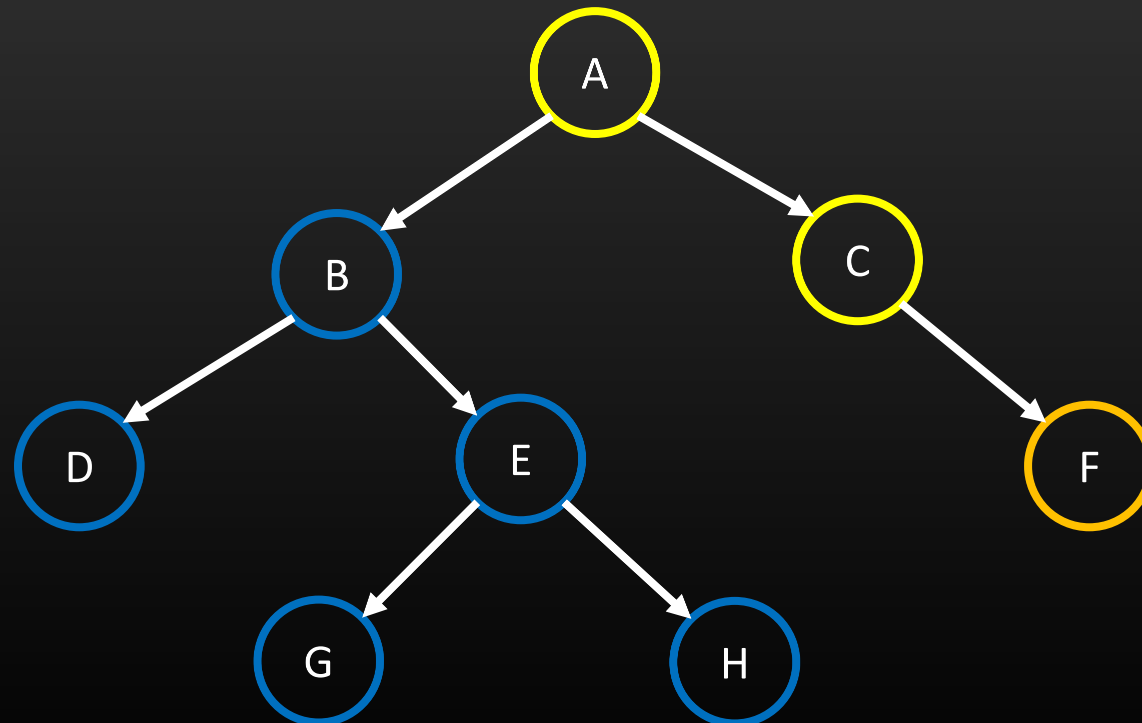


Depth First Search – Post-order

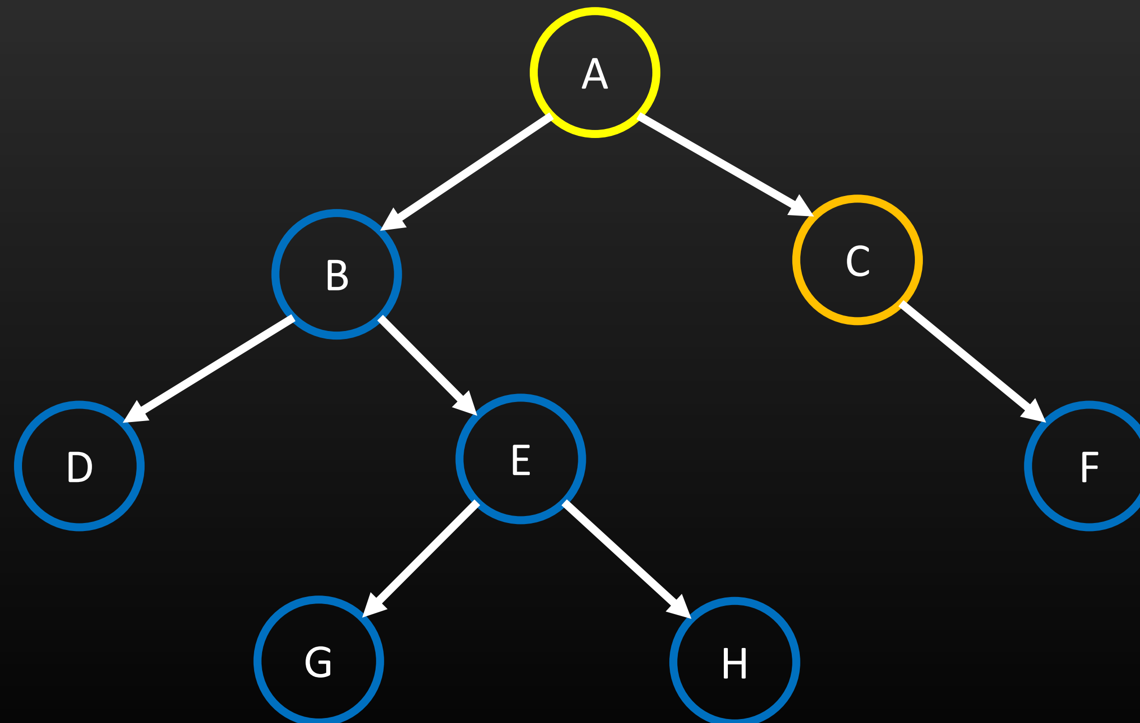
D	G	H	E	B				



Depth First Search – Post-order

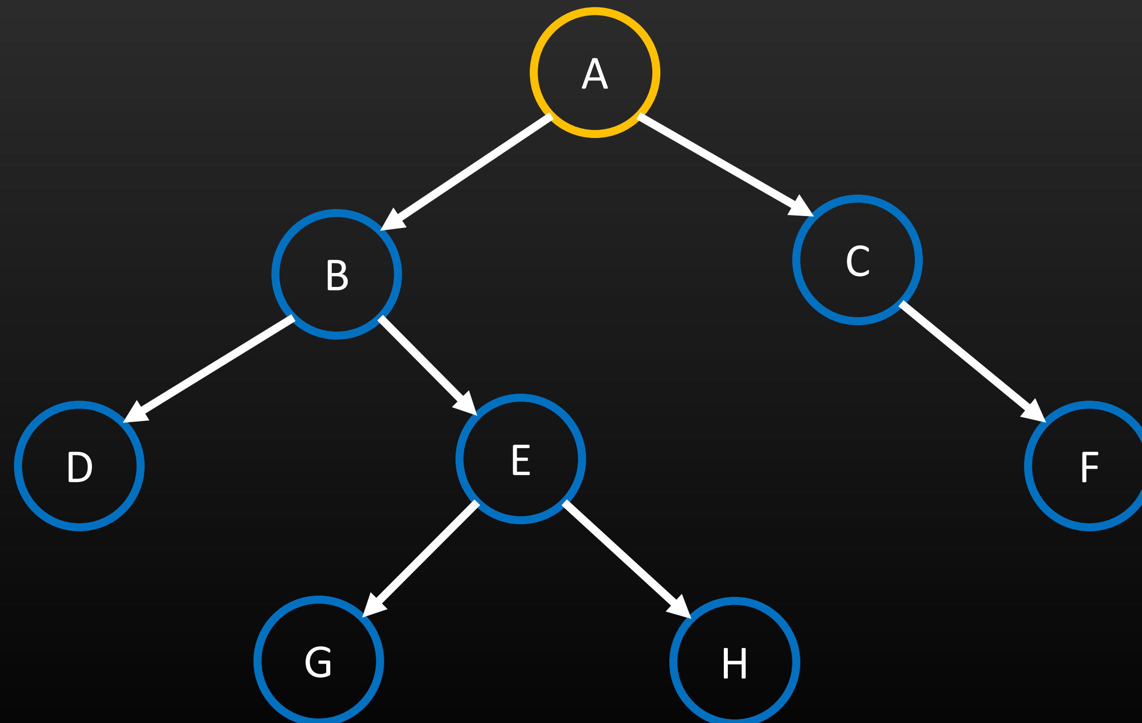


Depth First Search – Post-order

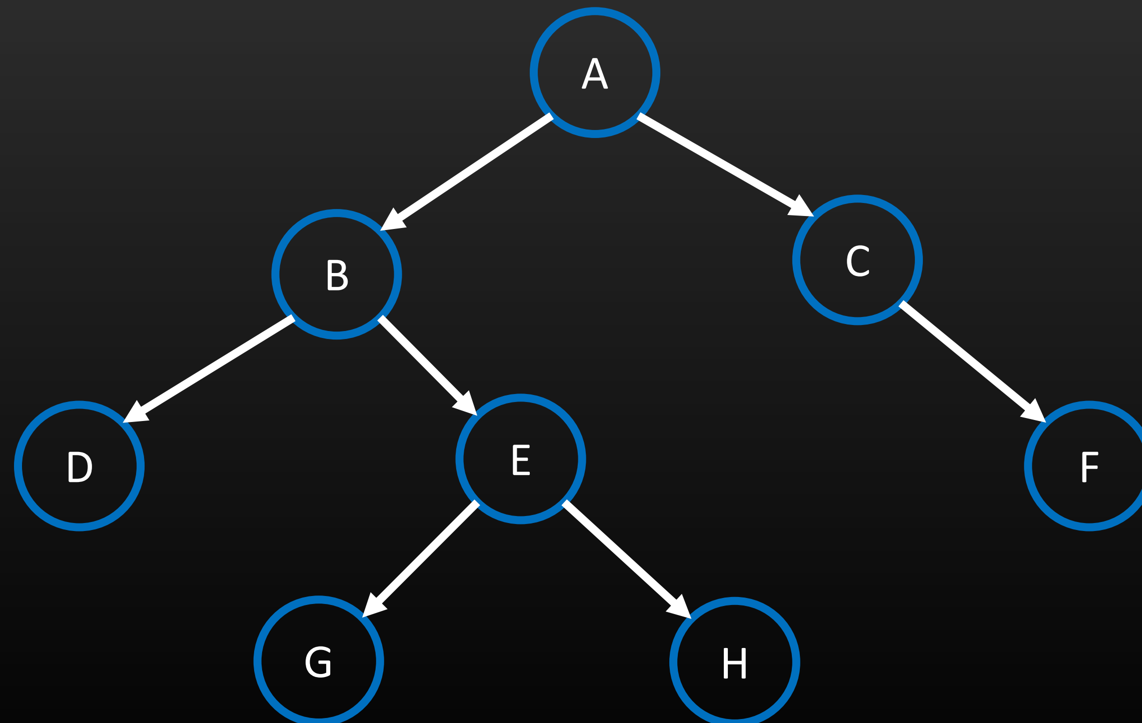


Depth First Search – Post-order

D	G	H	E	B	F	C	A	



Depth First Search – Post-order



Depth First Search

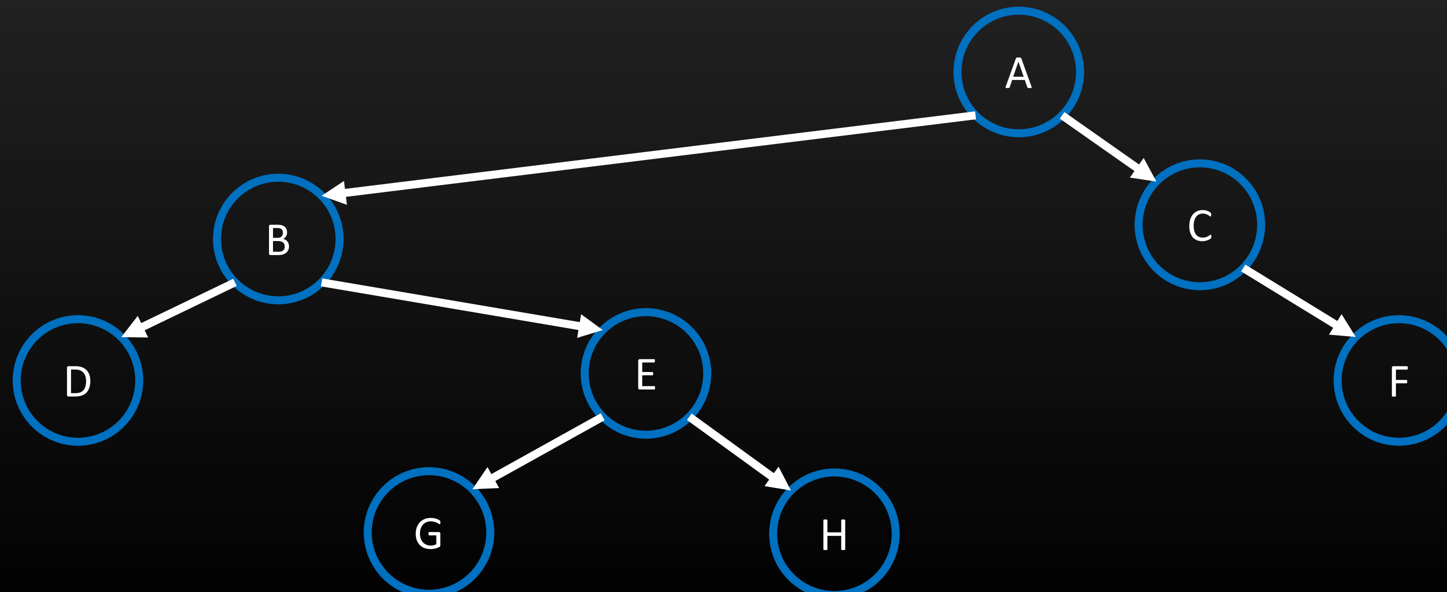
- Given either pre-order or post-order paired with in-order, the structure of the binary tree can be known

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

Depth First Search

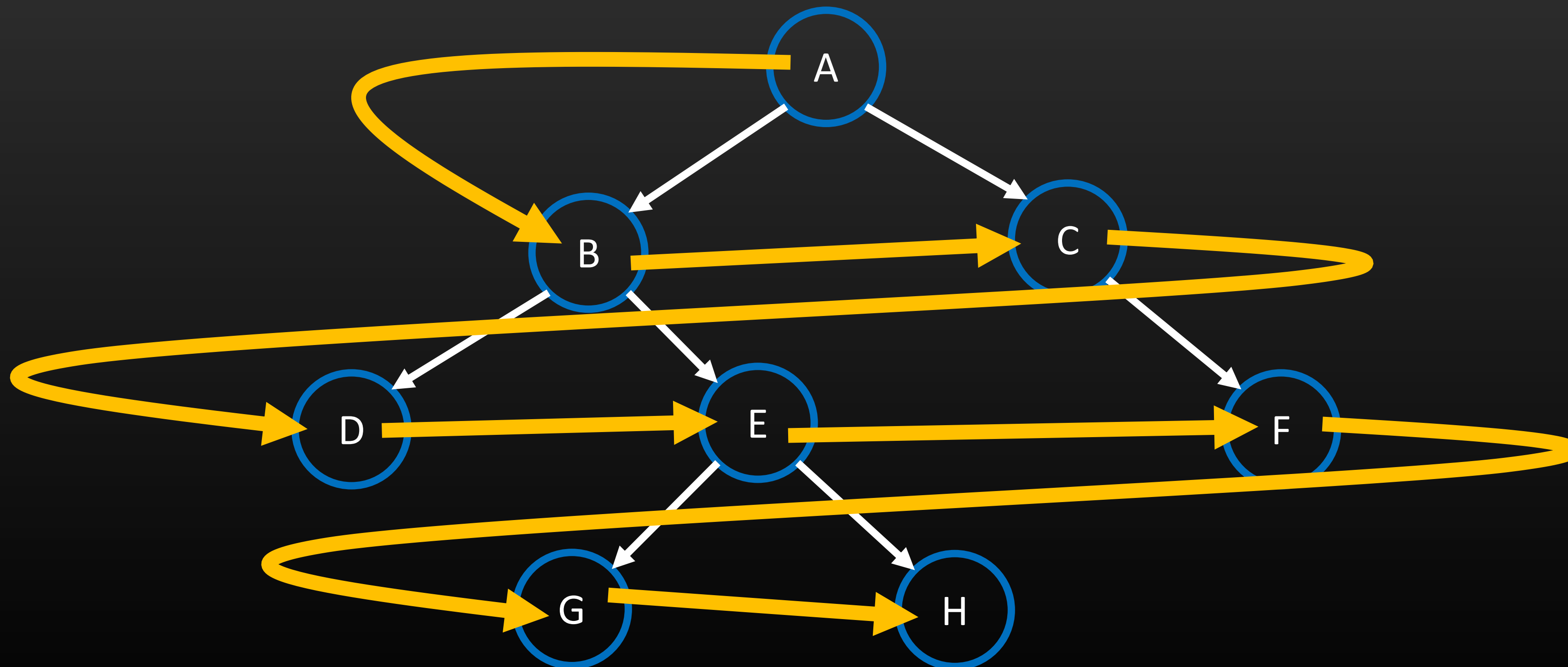
- Given either pre-order or post-order paired with in-order, the structure of the binary tree can be known

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

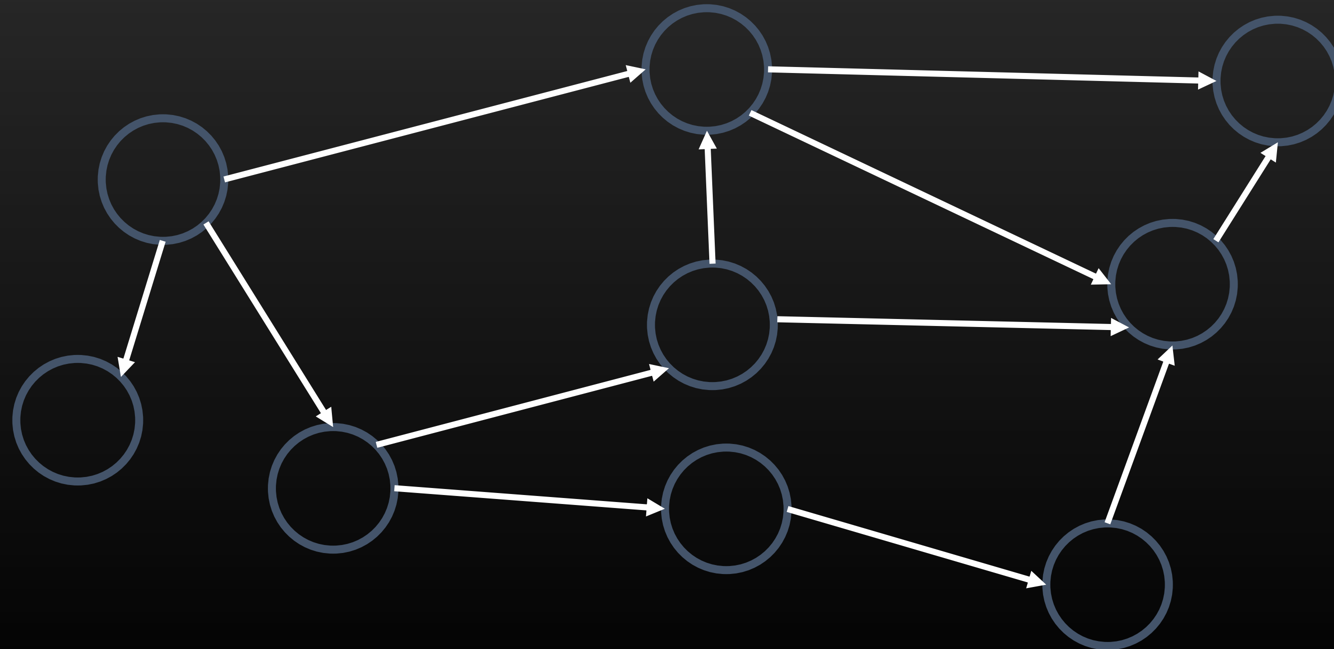


Breadth First Search

```
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
}
```

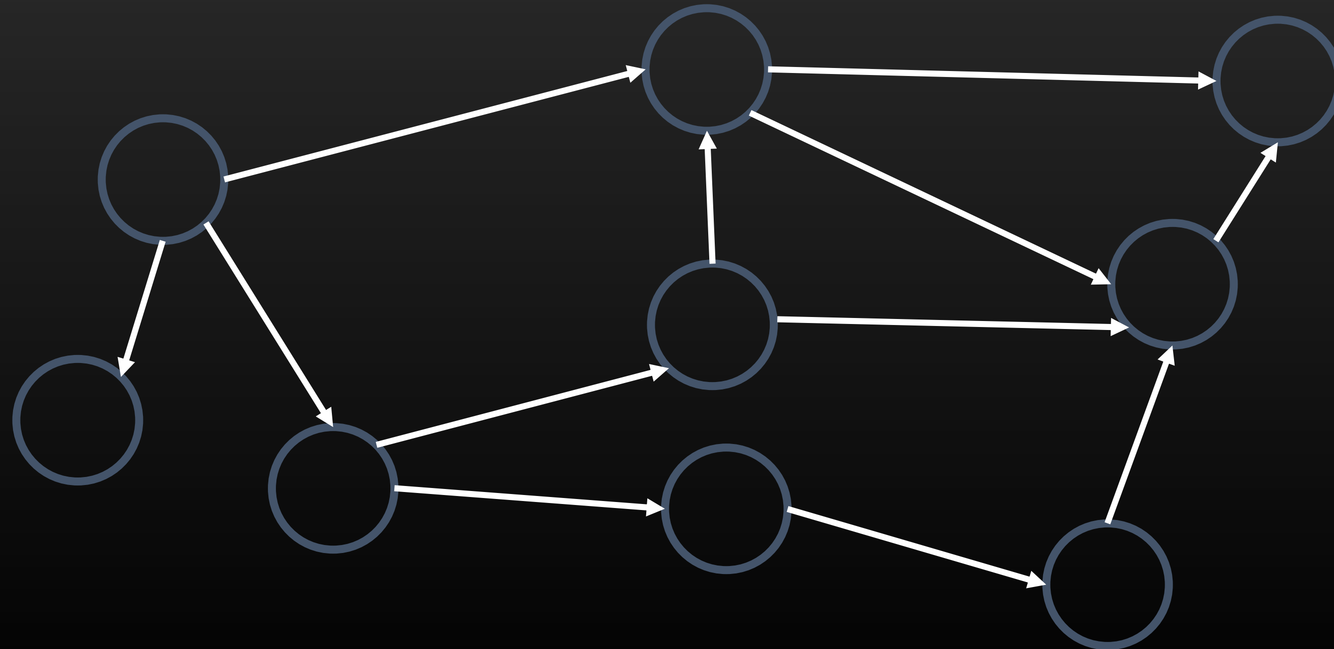
Directed Acyclic Graph

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



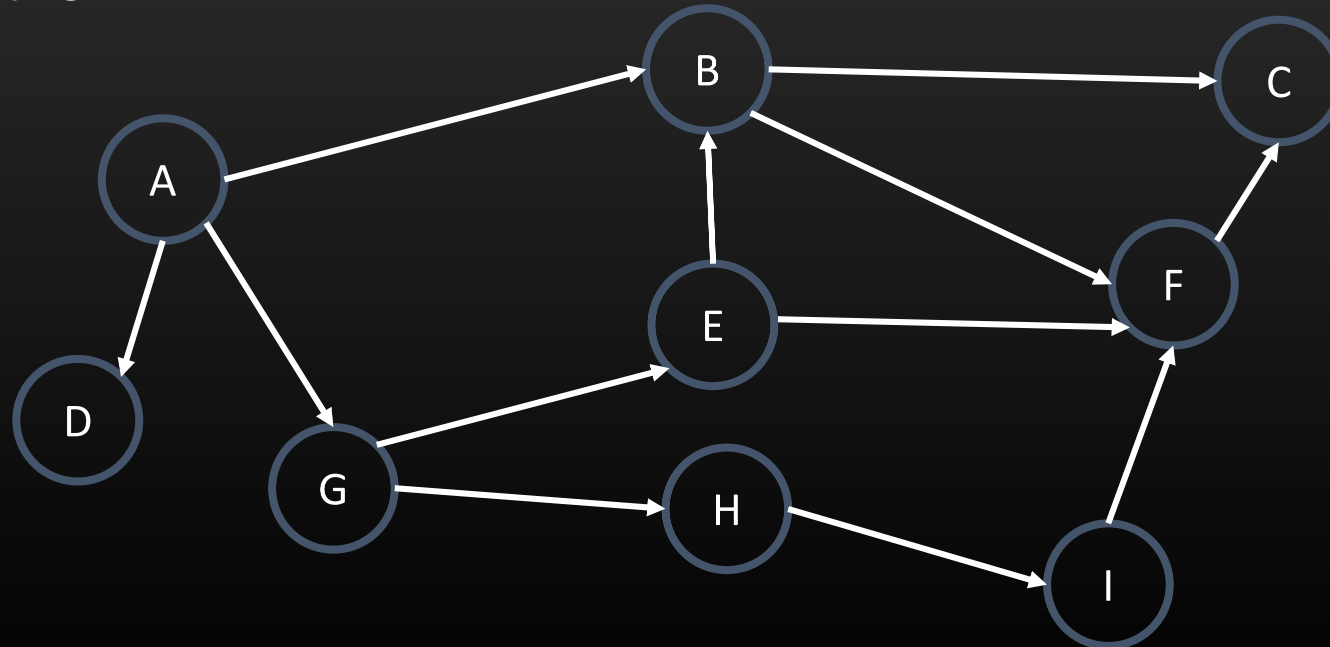
Directed Acyclic Graph

- 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

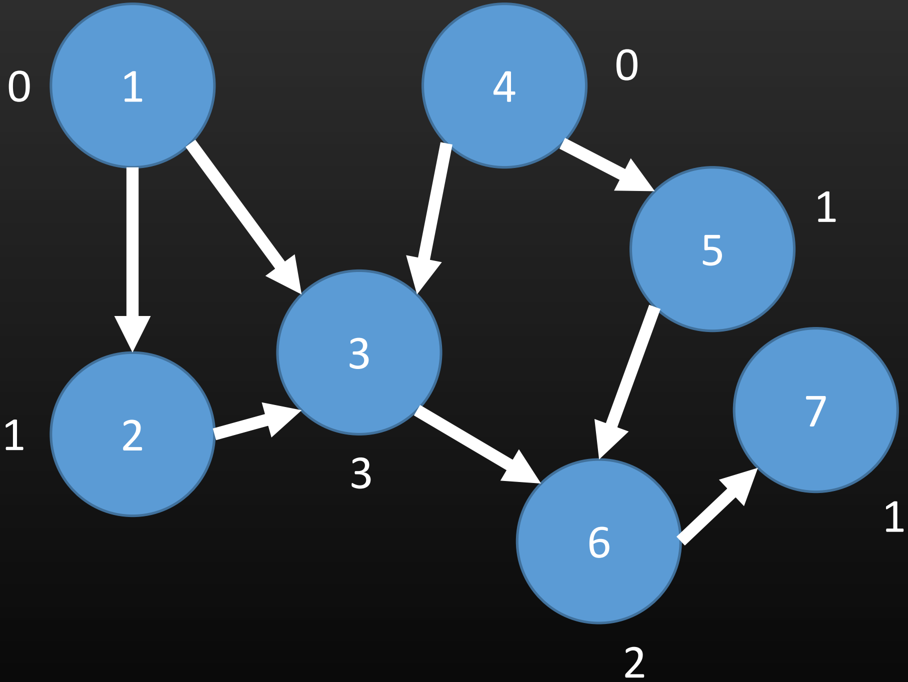
- Find a vertex with no incoming edges. Number it and remove all outgoing edges from it
- Repeat the above process
- Can be implemented by DFS or BFS

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

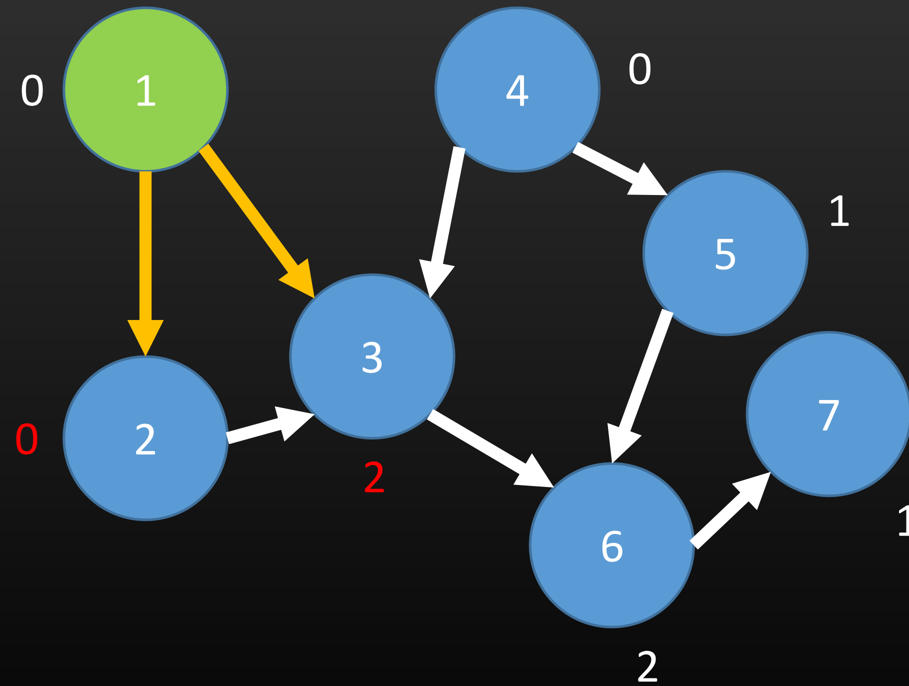
Topological Sort

- Topological order:



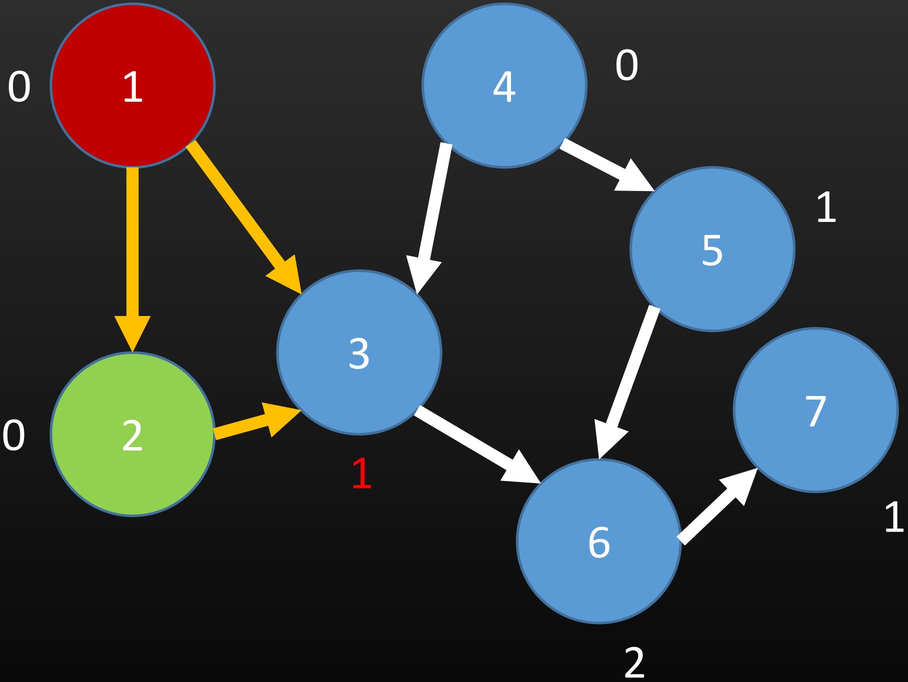
Topological Sort

- Topological order:



Topological Sort

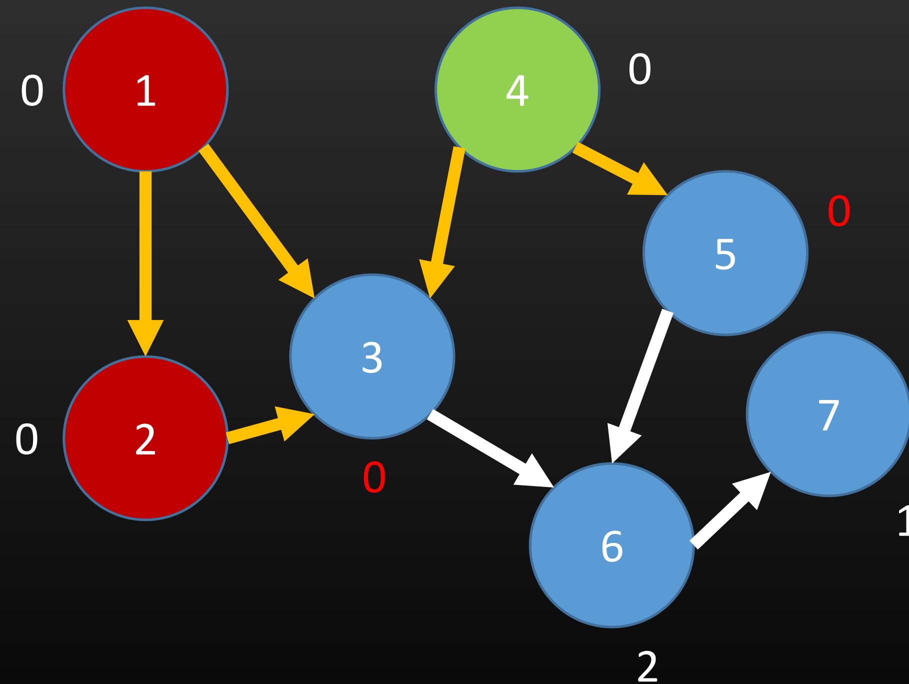
- Topological order:



Topological Sort

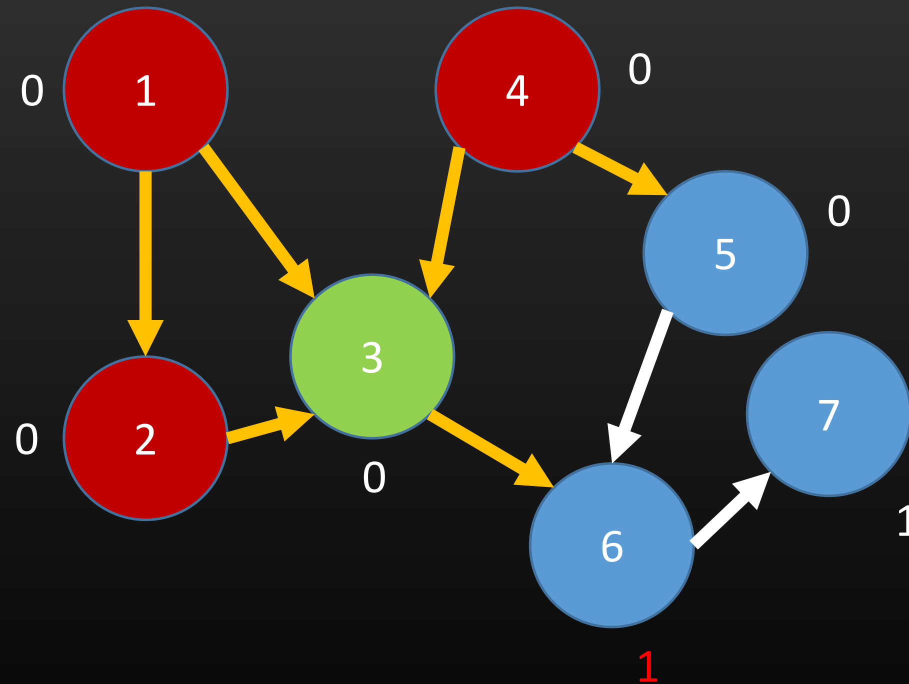
- Topological order:

1	2	4				



Topological Sort

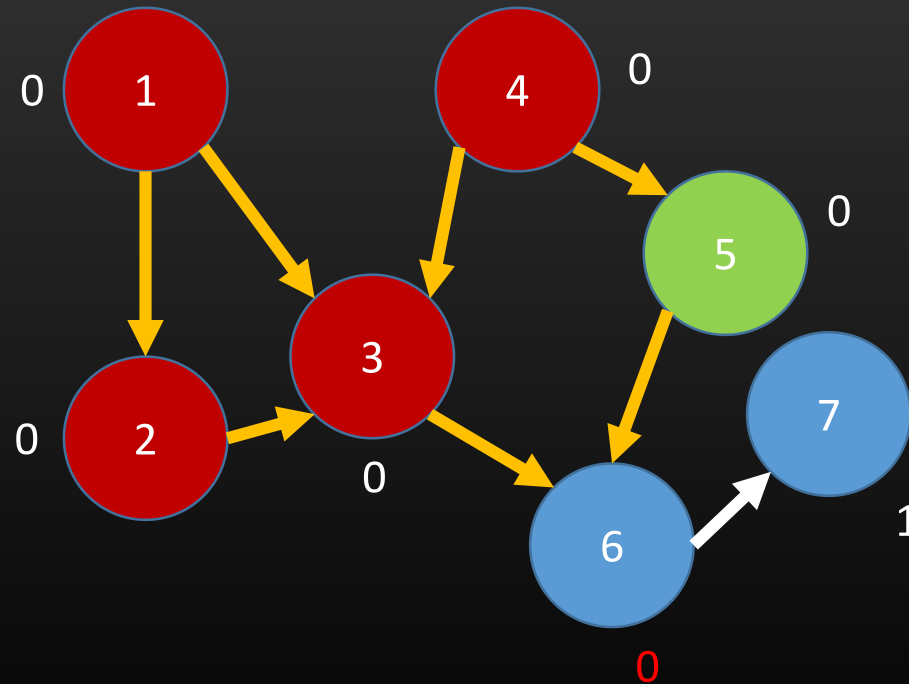
- Topological order:



Topological Sort

- Topological order:

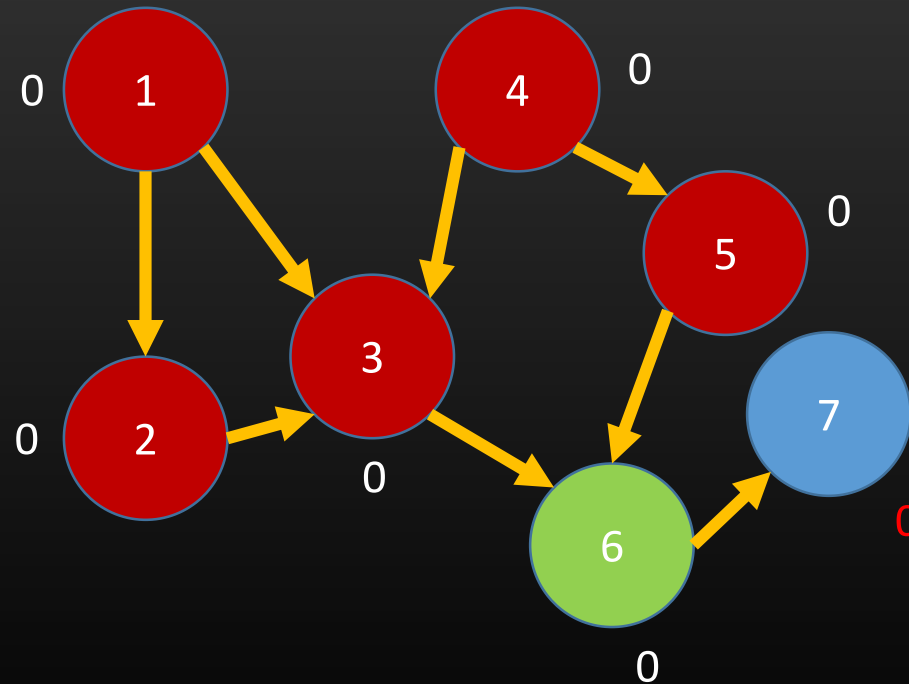
1	2	4	3	5		



Topological Sort

- Topological order:

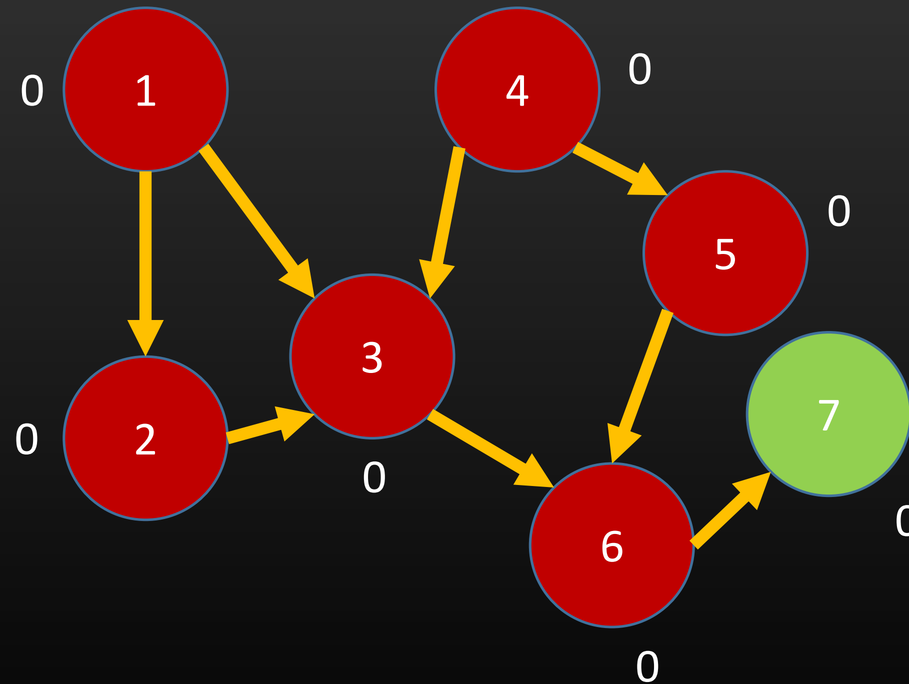
1	2	4	3	5	6	



Topological Sort

- Topological order:

1	2	4	3	5	6	7



Topological Sort

```
Procedure Tsort(){
    for all node i do
        if indegree[i] = 0 then
            push the node i into the queue
    while the queue is not empty do
        t = front of the queue
        append node t to the order
        for all w in children of t do
            indegree[w] = indegree[w] – 1
            if indegree[w] = 0 then
                push node w into the queue
        pop t from the queue
}
```

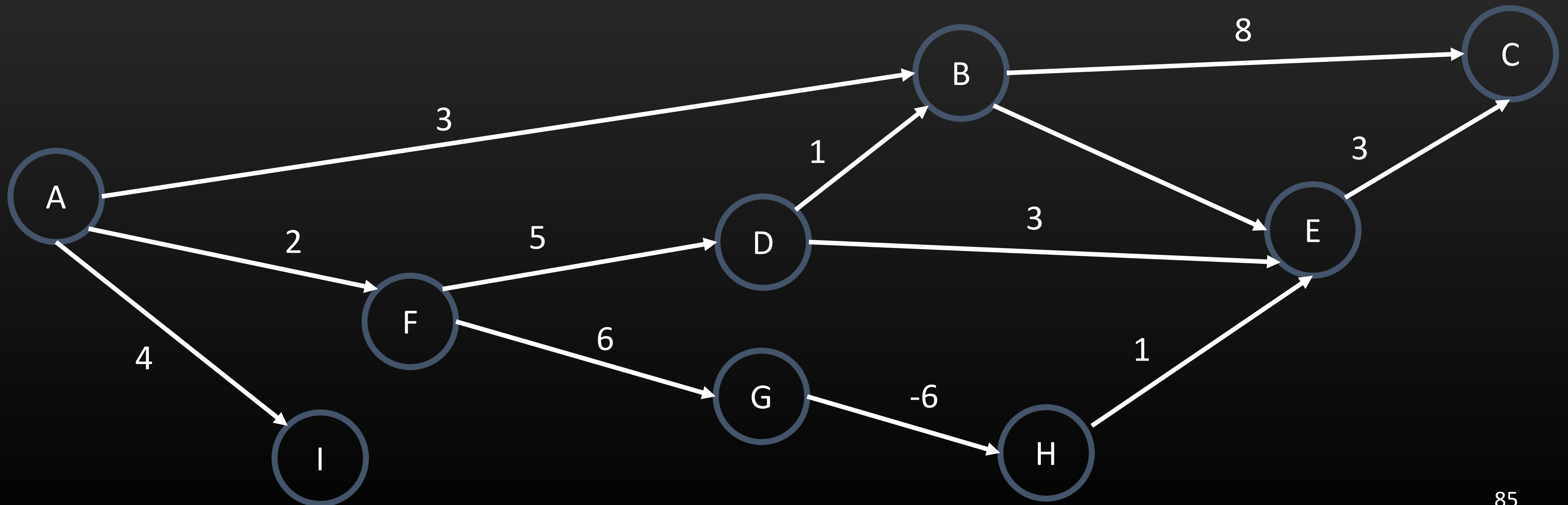
Topological Sort

- 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

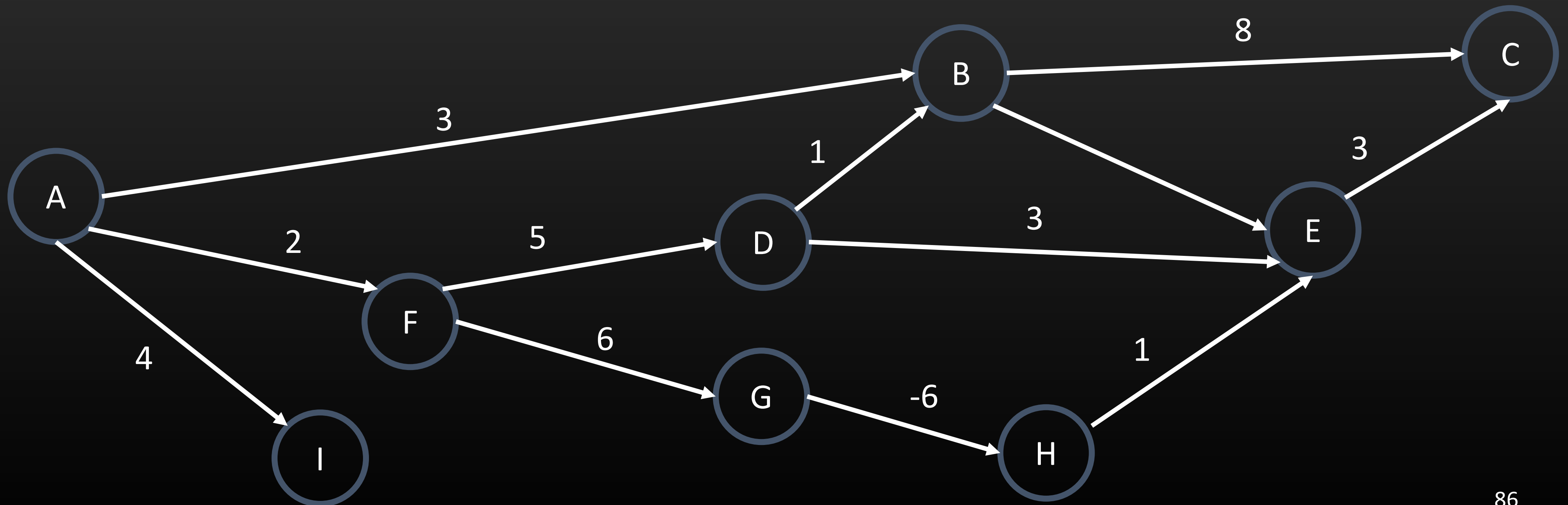
Example

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



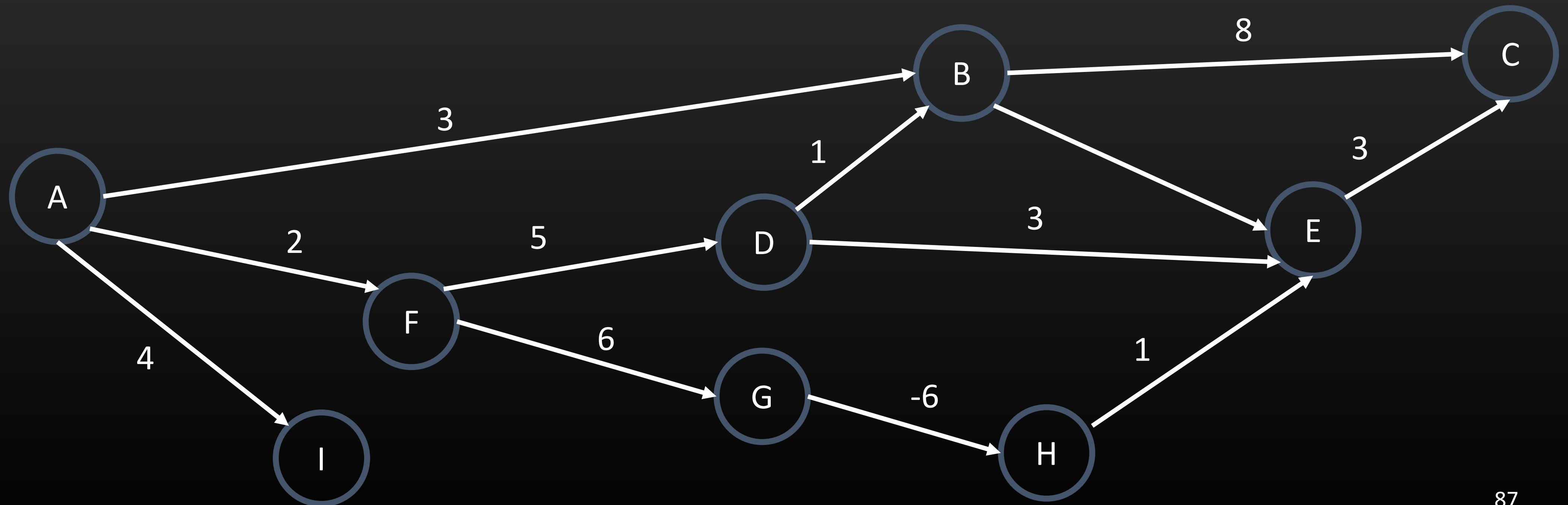
Example

- Find the topological order first: AFIDGBHEC



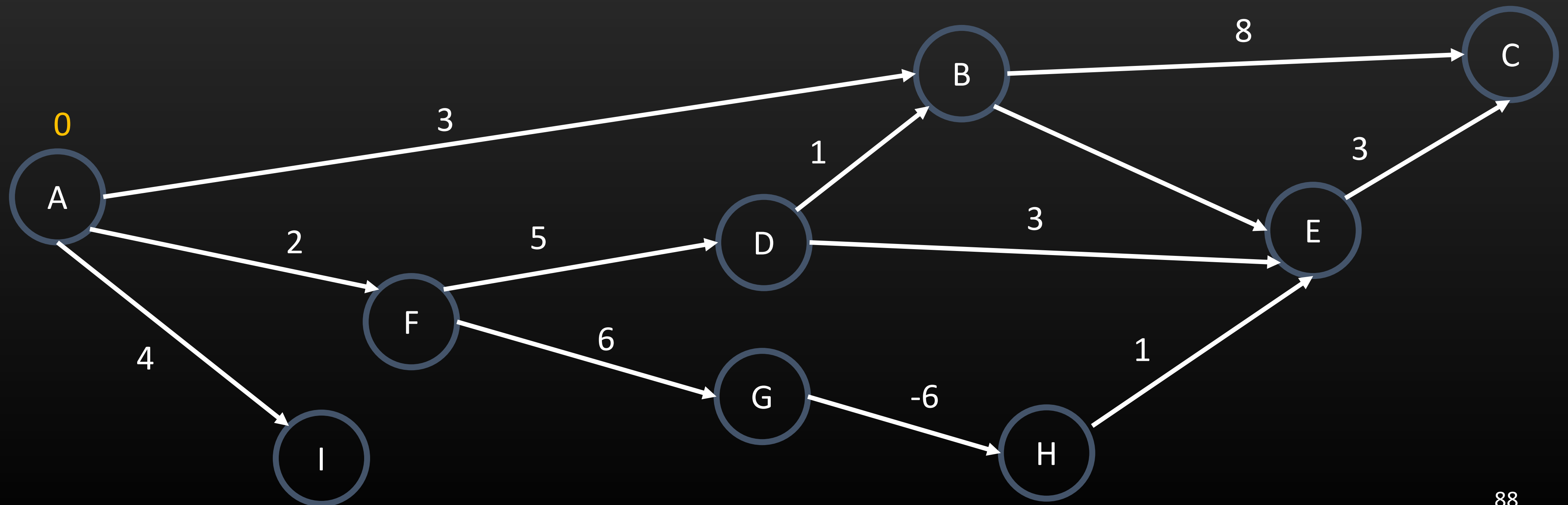
Example

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



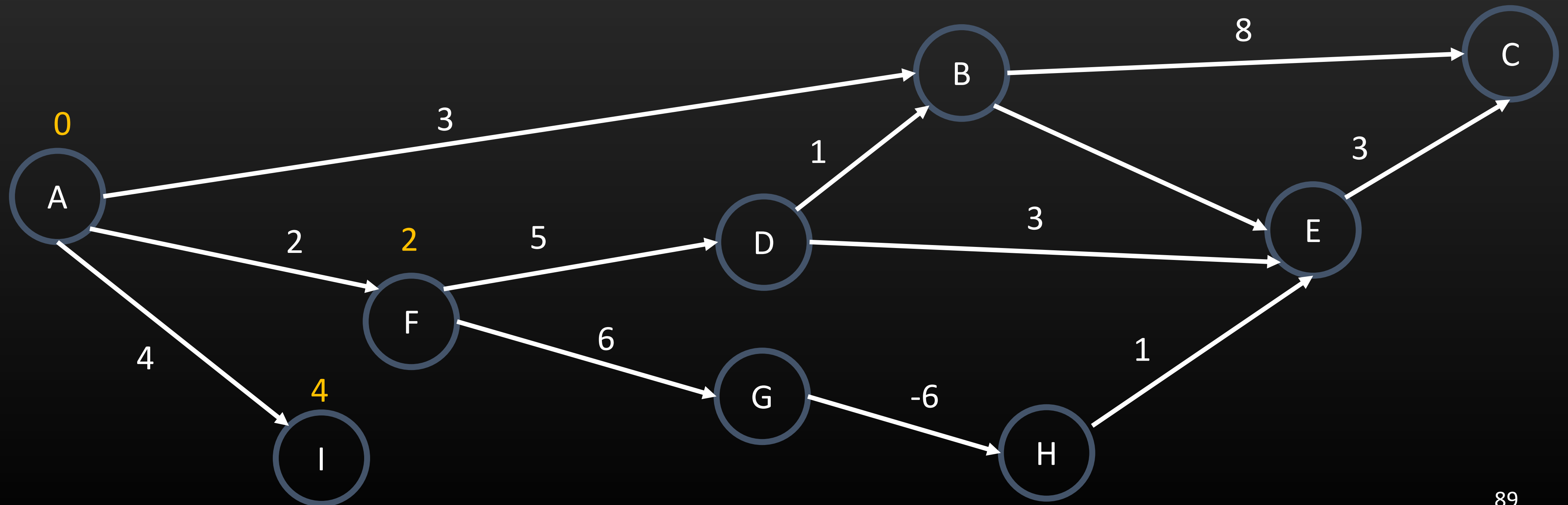
Example

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



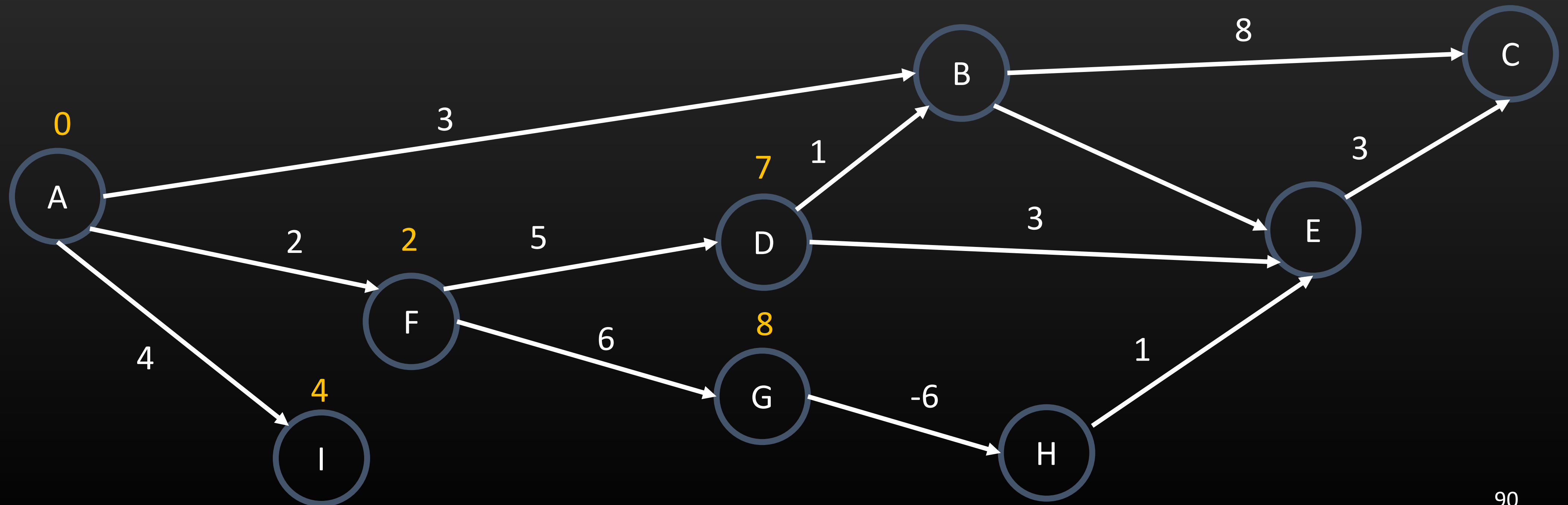
Example

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



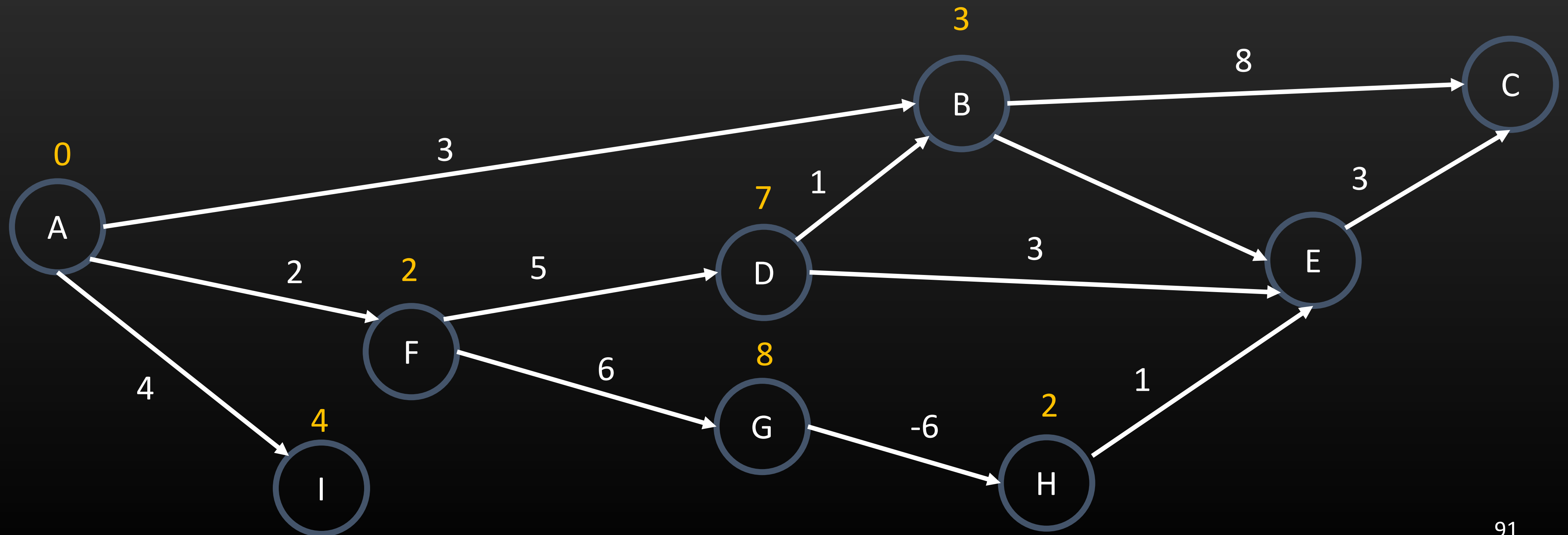
Example

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



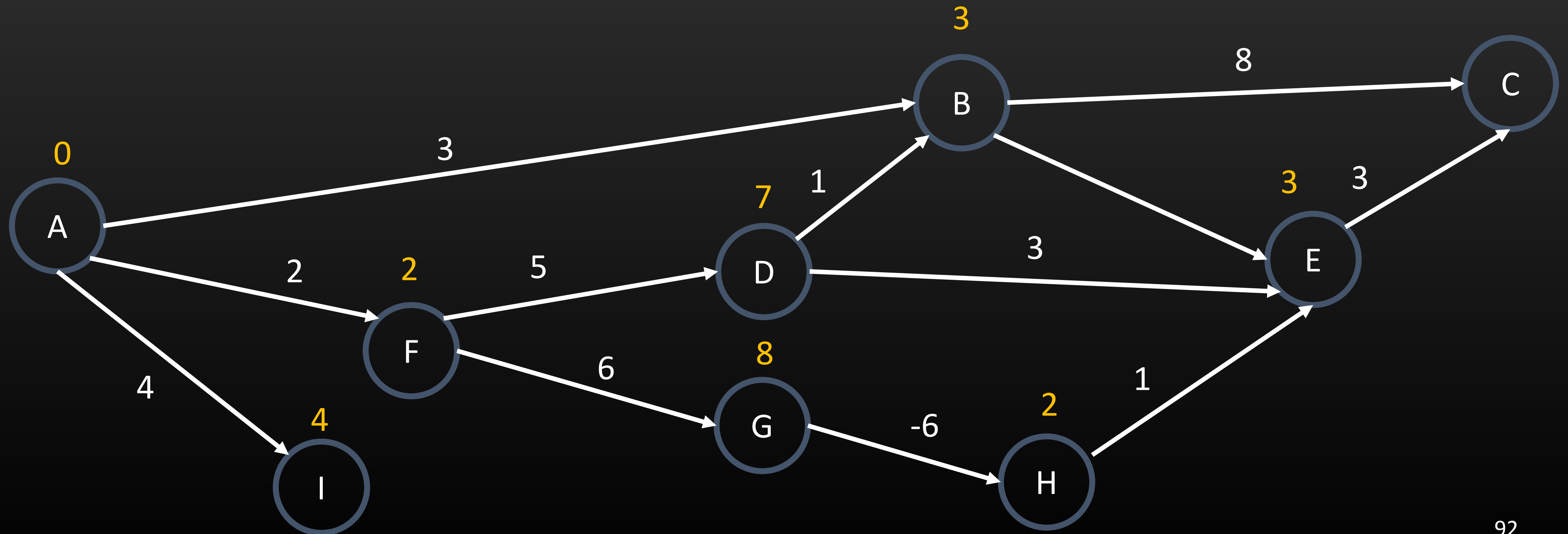
Example

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



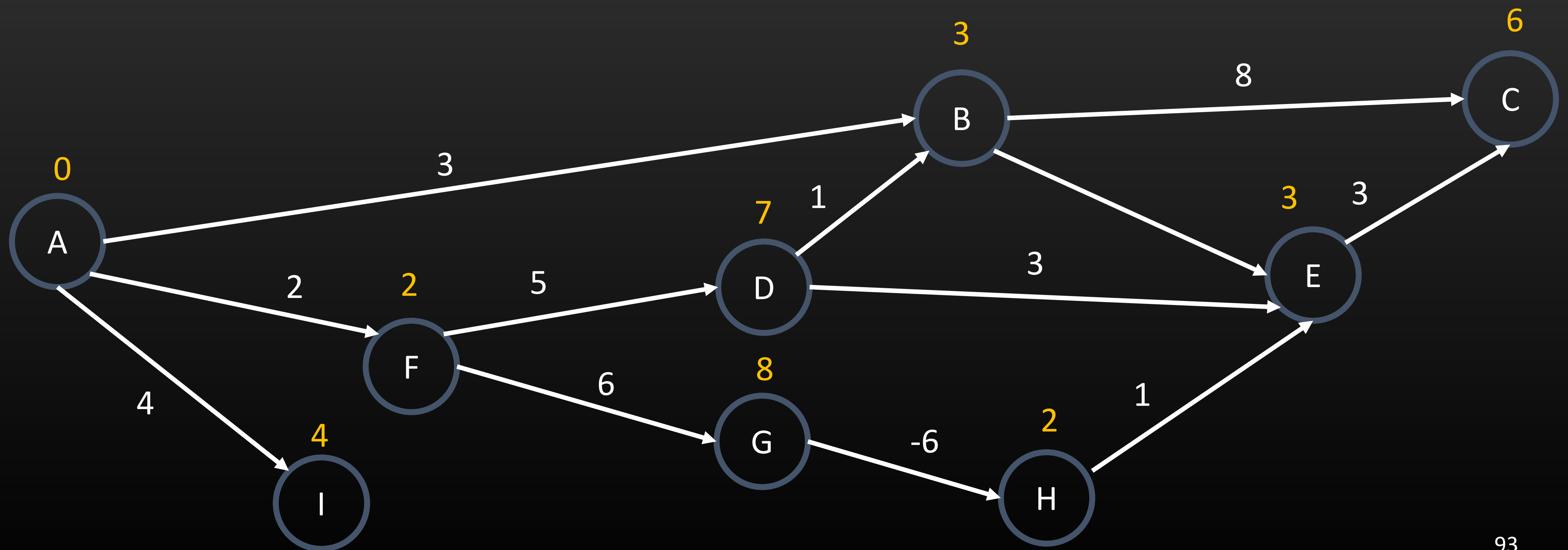
Example

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



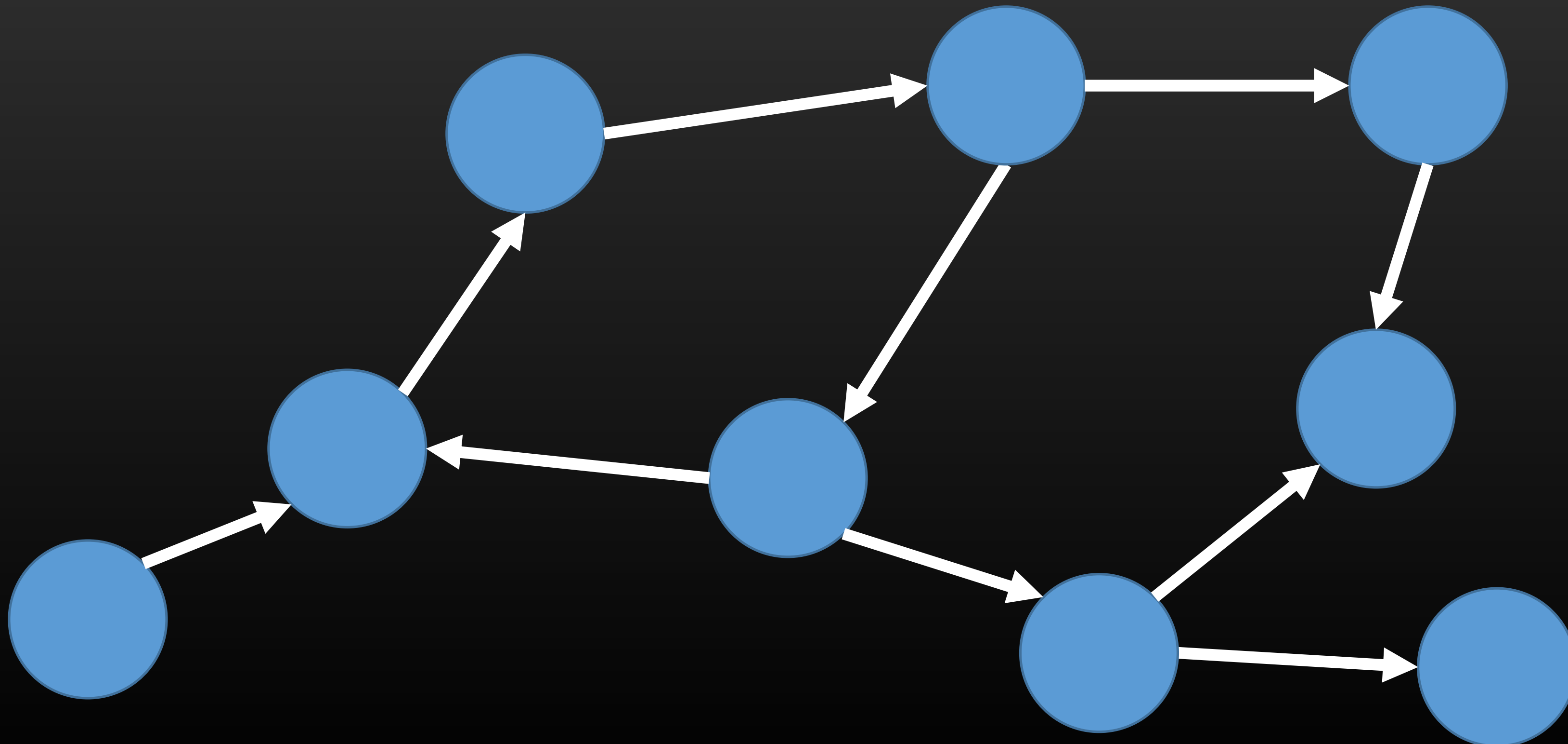
Example

- 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

1 1 1
I H G
2 1 2
B C F
0 1 1
A D E

Detecting cycles

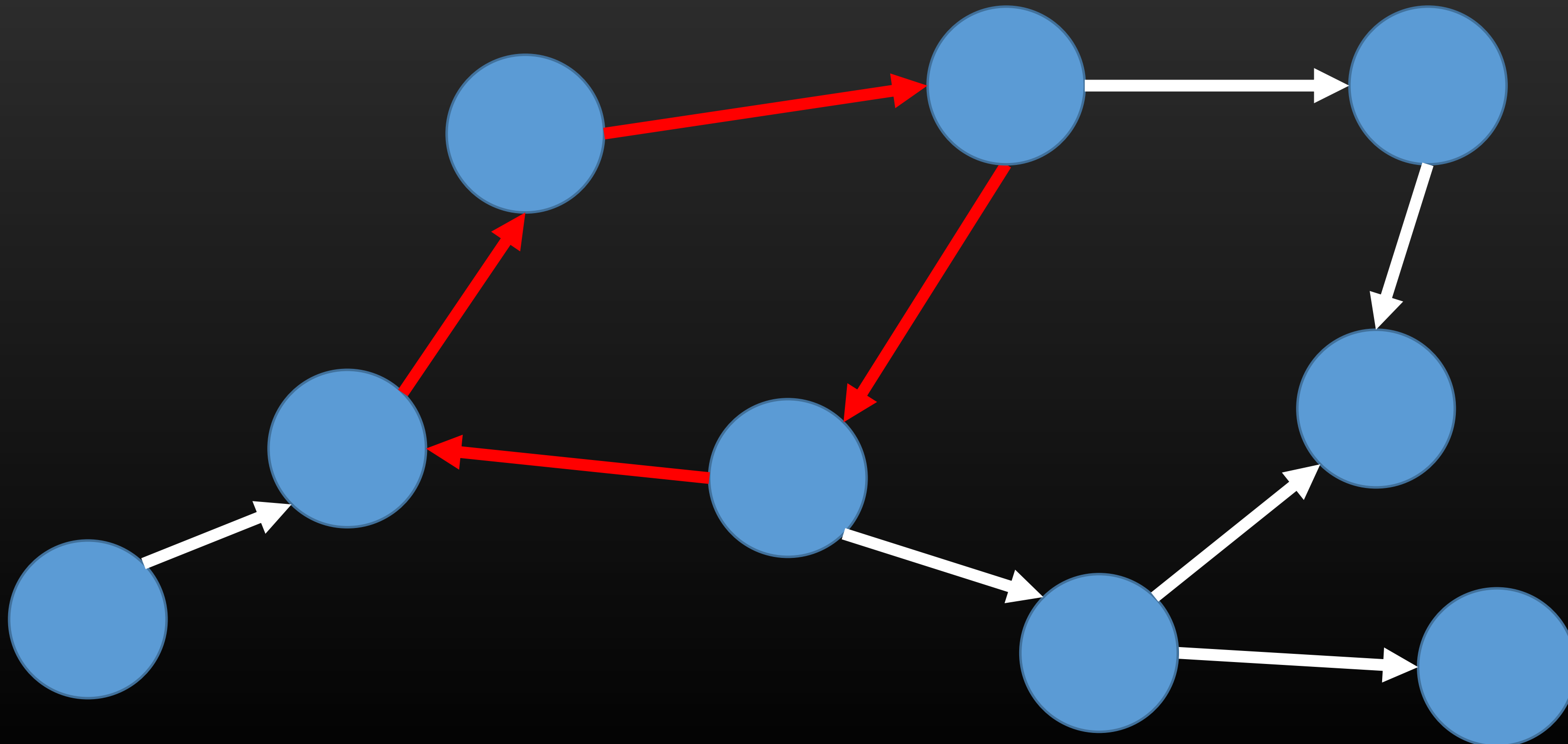
0	1	1	1	1	1	1	2	1
A	B	I	H	G	C	D	F	E

Detecting cycles

STUCK

Detecting cycles

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

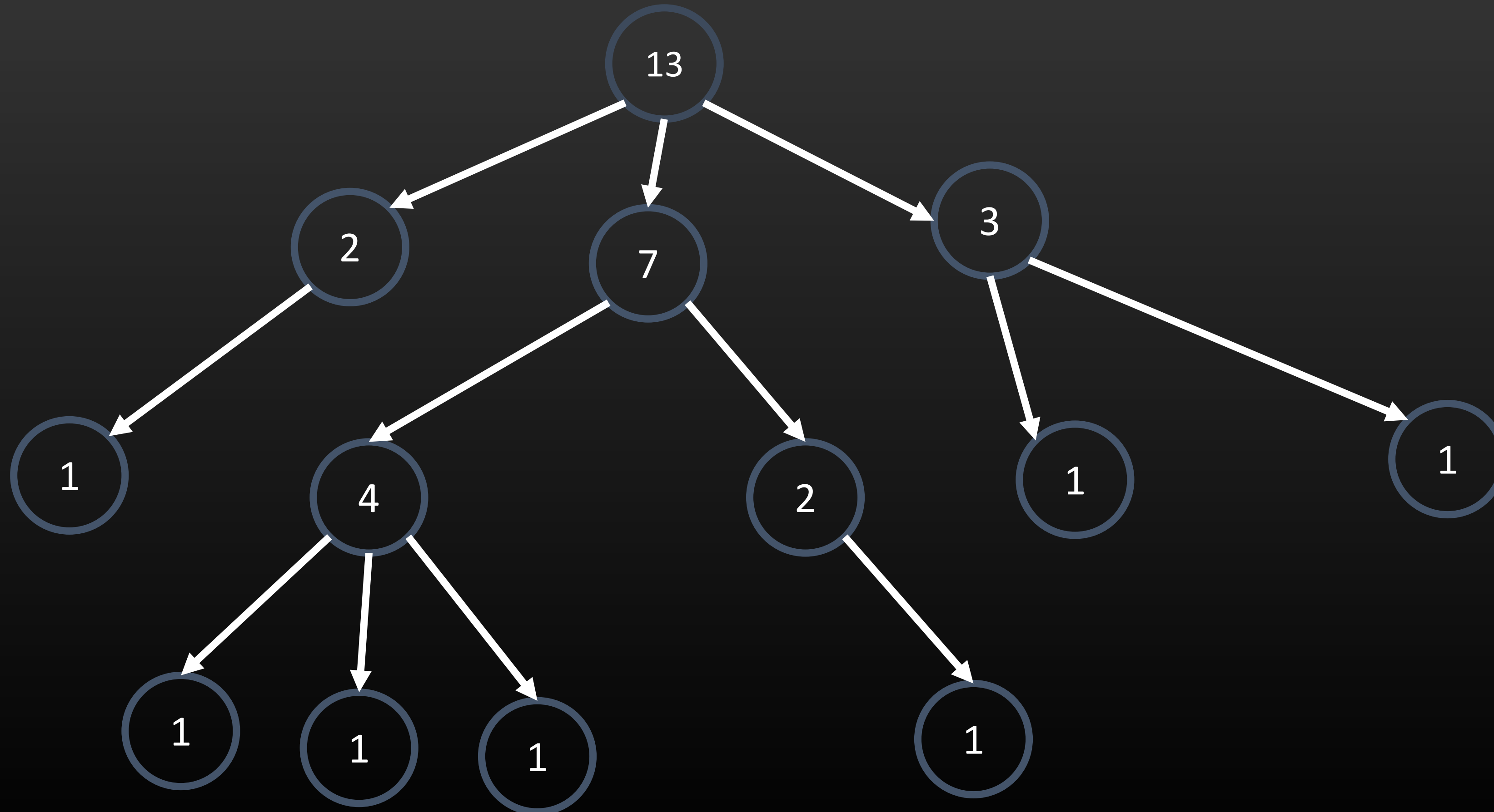


Algorithms on Tree

- There are no cycles on tree
- Many problems can be solved efficiently

Example 1 – Size of Subtrees

- Given a rooted tree, find the size of subtrees rooted on each node



Example 1 – Size of Subtrees

- The number of descendants of each node relates to the number of descendants of all of its children
- The answer will be the sum of the answer of its children + 1

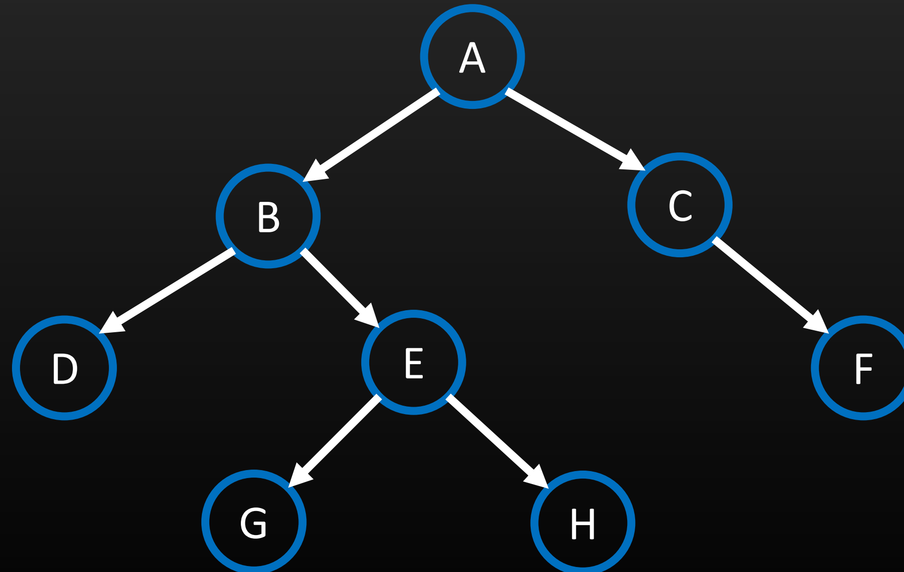
- Need to compute the answers of the children first
- Depth First Search using Post-order

Example 1 – Size of Subtrees

```
void dfs(int x, int parent){
	size[x] = 1;
	for (auto i : e[x])
		if (i != parent){
			dfs(i, x);
			size[x] += size[i];
		}
}
```

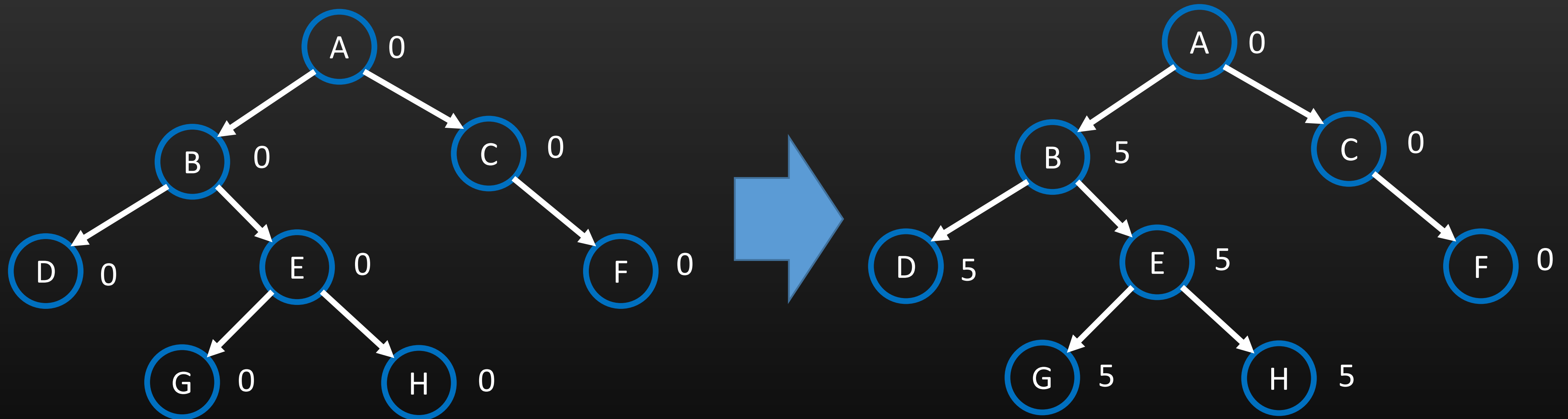
Example 2 – Subtree Updates

- A tree is given and each node has its own value (initially zero)
- Two kinds of queries will be performed:
  1. Update(x, v): Increase the values of all nodes in subtree x by v
  2. Answer(x): Return the current value of the node x



Example 2 – Subtree Updates

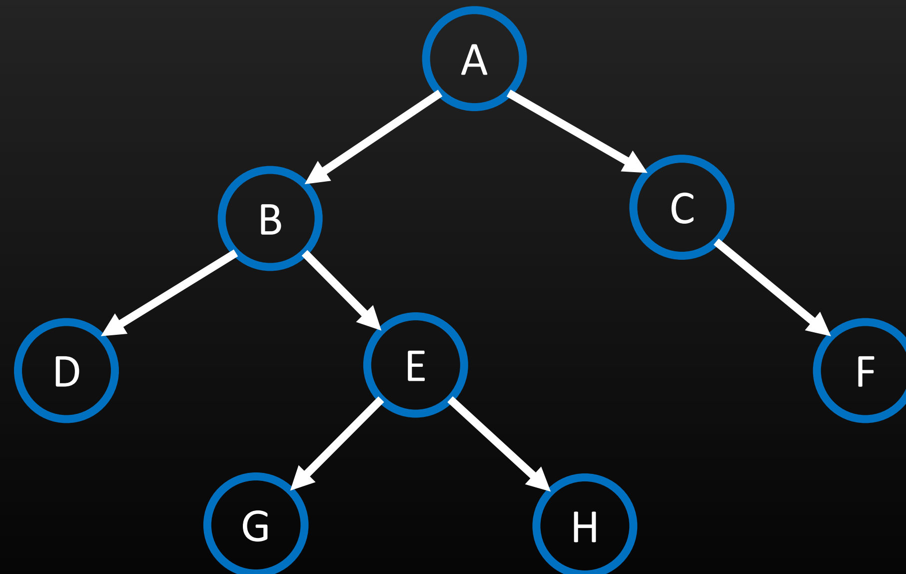
- Update(B, 5):



Example 2 – Subtree Updates

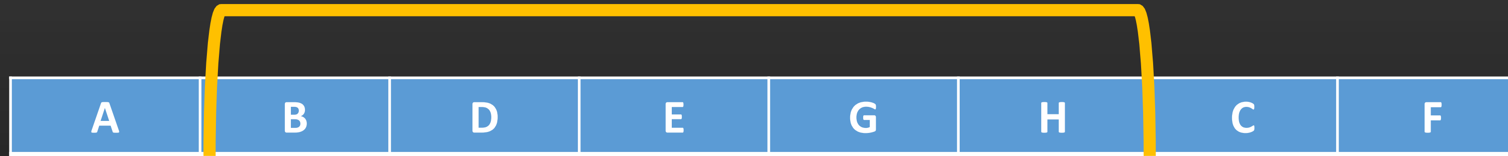
- Consider the pre-order of the tree

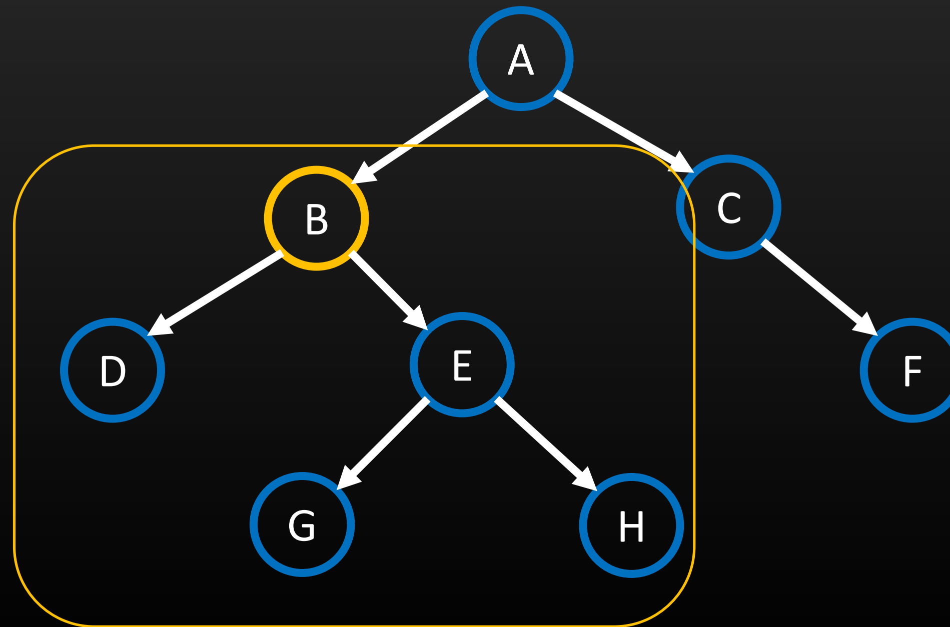
A	B	D	E	G	H	C	F



Example 2 – Subtree Updates

- The nodes of the subtree *x* appear consecutively after *x*

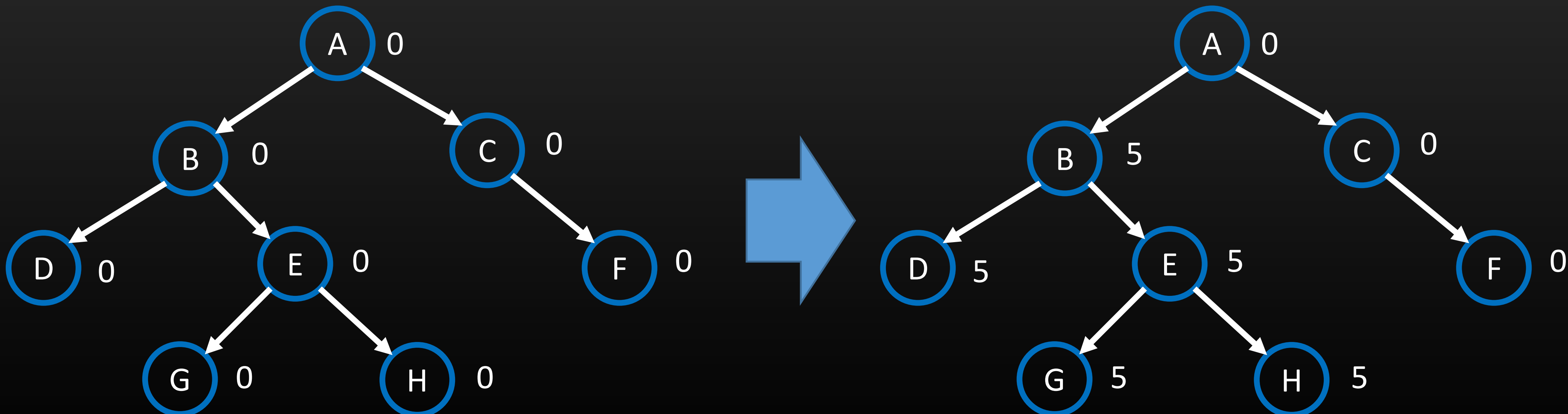




Example 2 – Subtree Updates

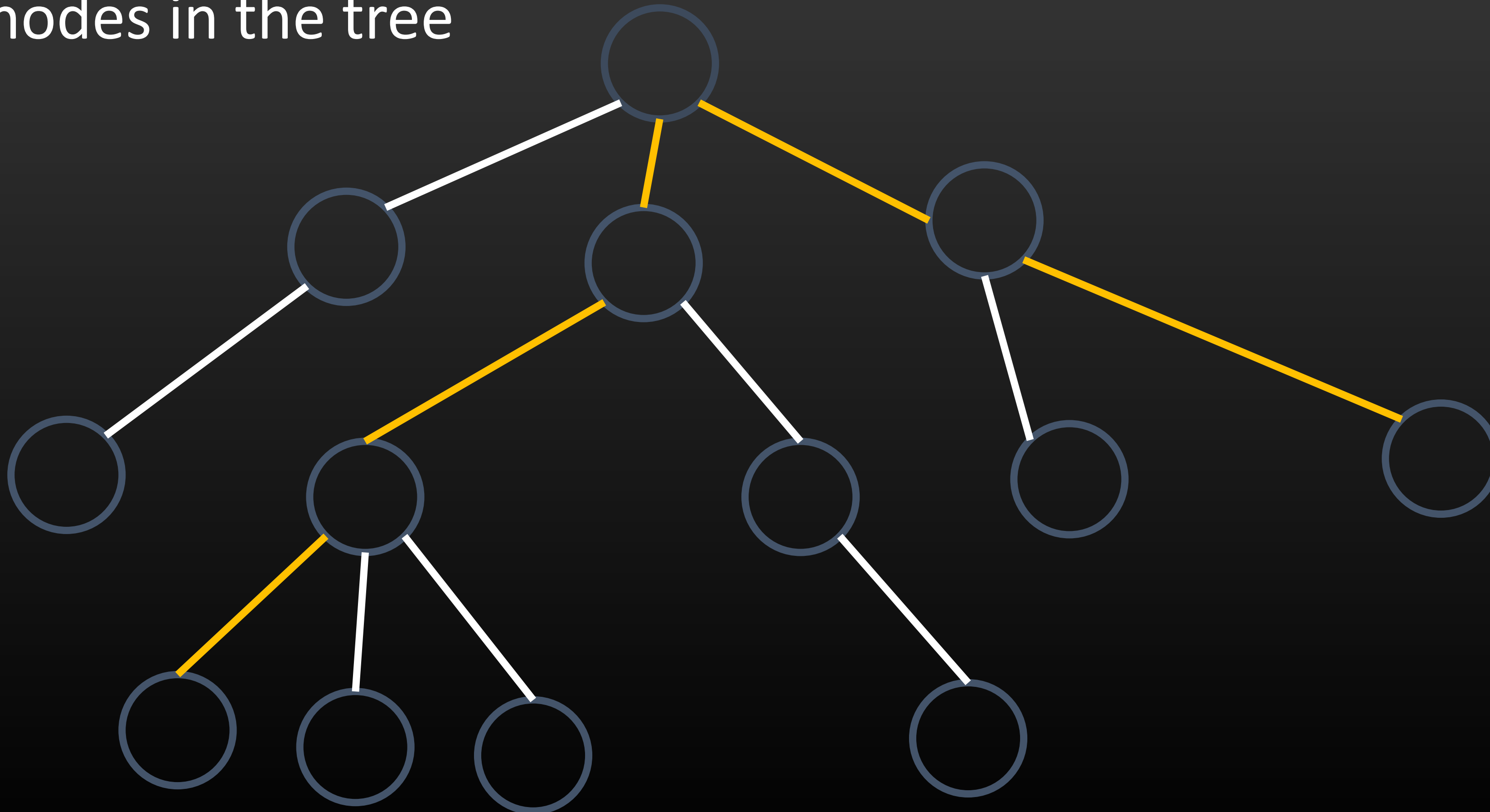
- The problem becomes updating the values of a subarray, which can be done using segment tree

A	B	D	E	G	H	C	F
0	5	5	5	5	5	0	0



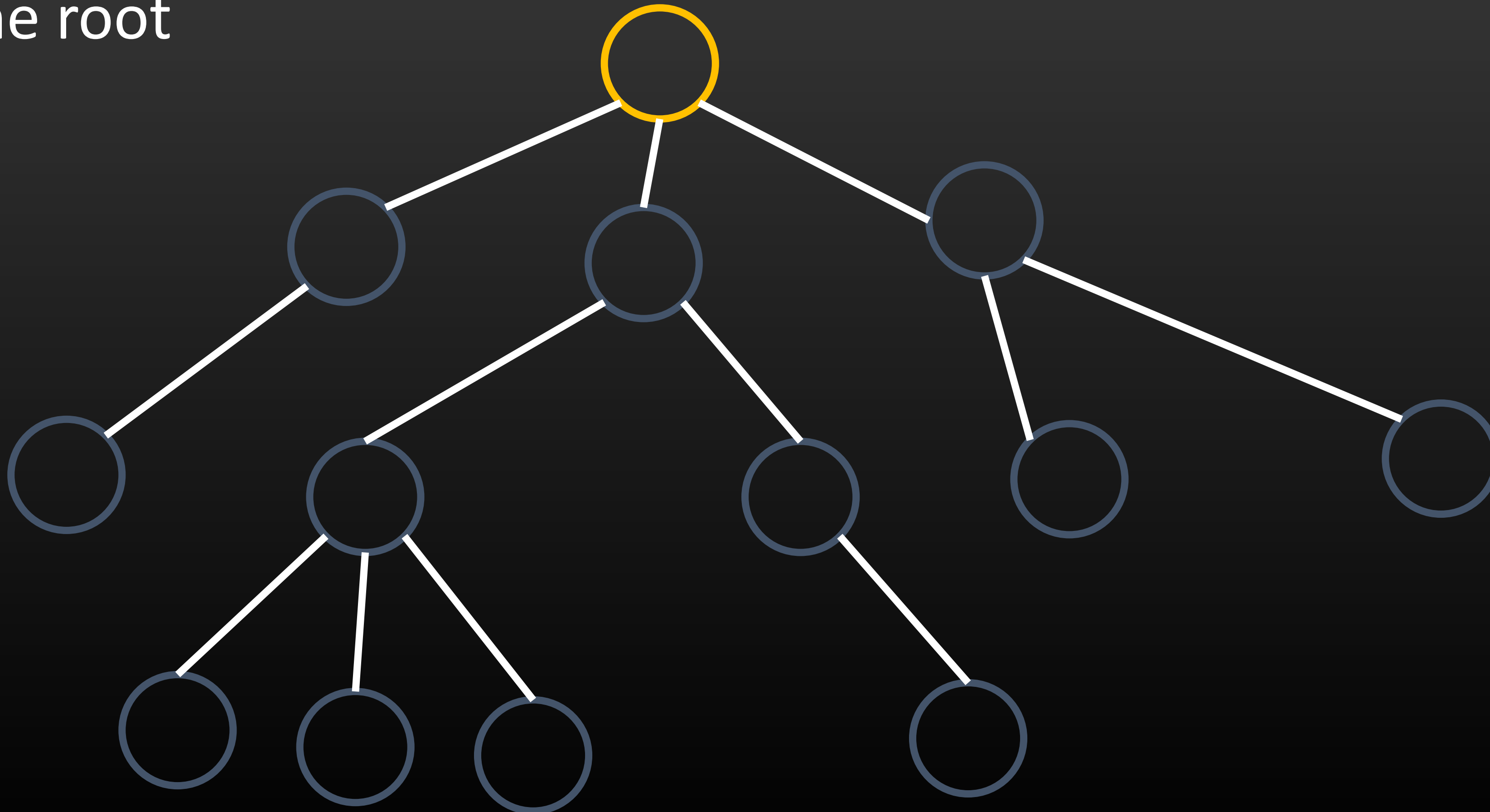
Example 3 – Tree Diameter

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



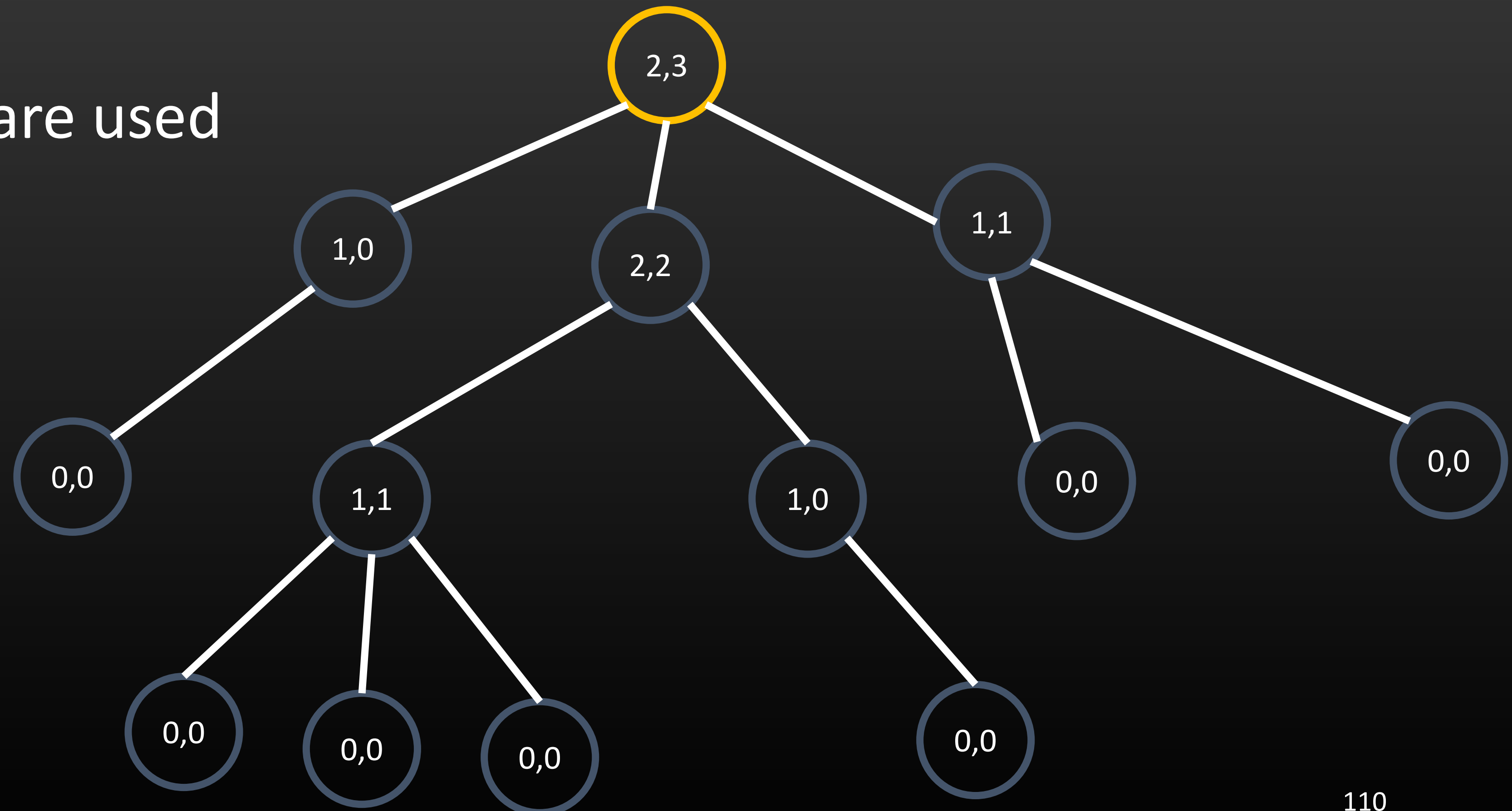
Example 3 – Tree Diameter

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



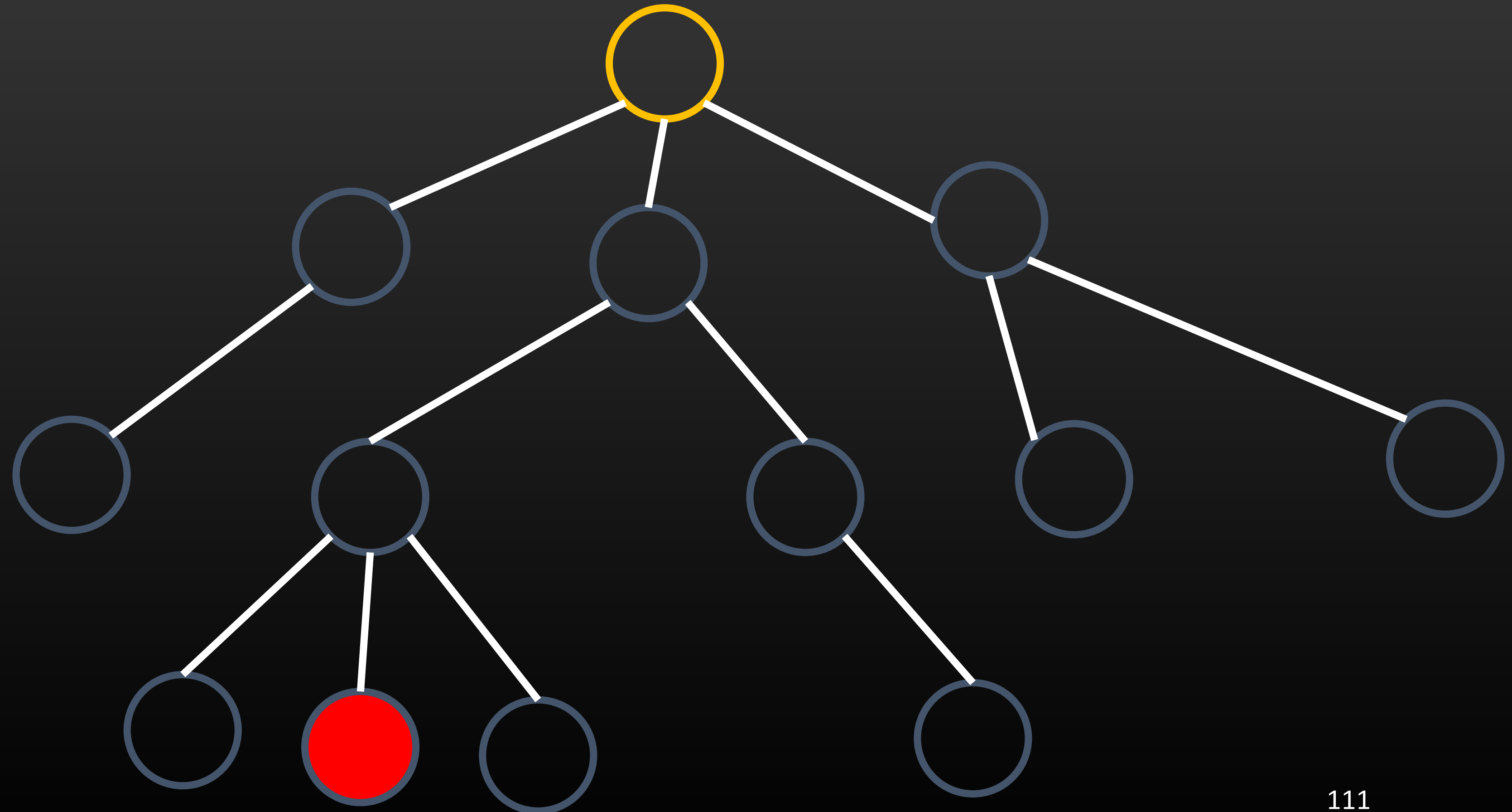
Example 3 – Tree Diameter

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



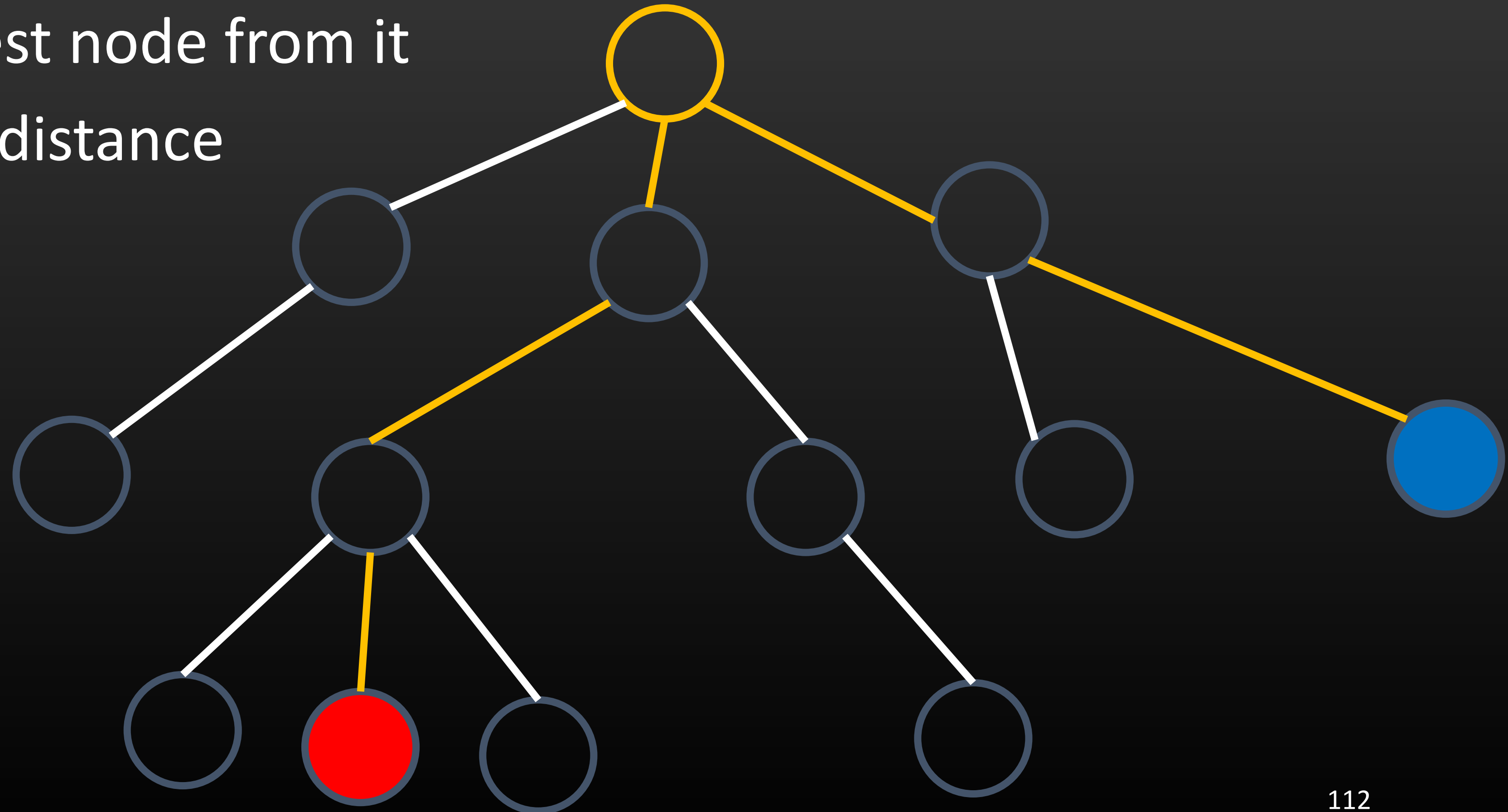
Example 3 – Tree Diameter

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



Example 3 – Tree Diameter

- Or we can find the deepest node from the root first (using DFS/BFS)
- Then find the deepest node from it
- The diameter is the distance between them

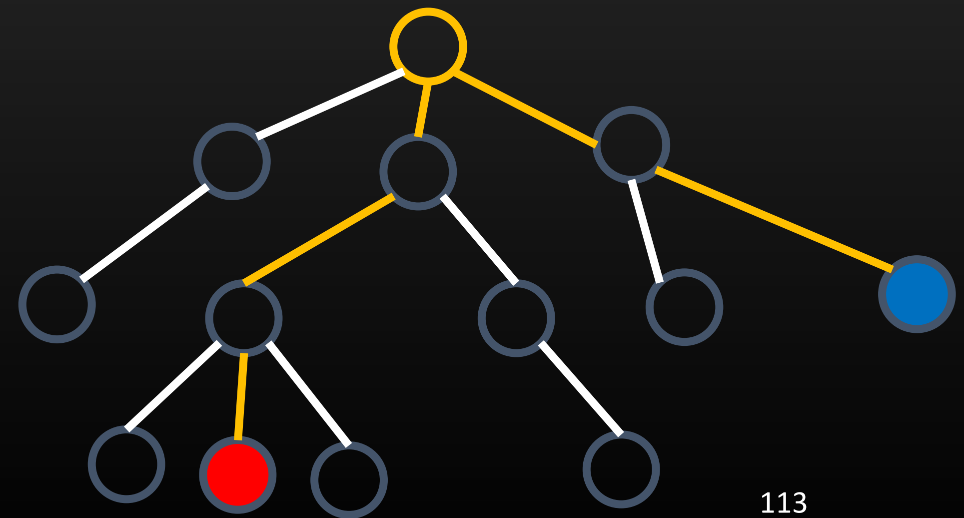


- Why does it work?

Example 3 – Tree Diameter

- Proof by contradiction

- Assume the longest path is not from the deepest node from the root
- You can always extend the path by changing one side to the deepest node

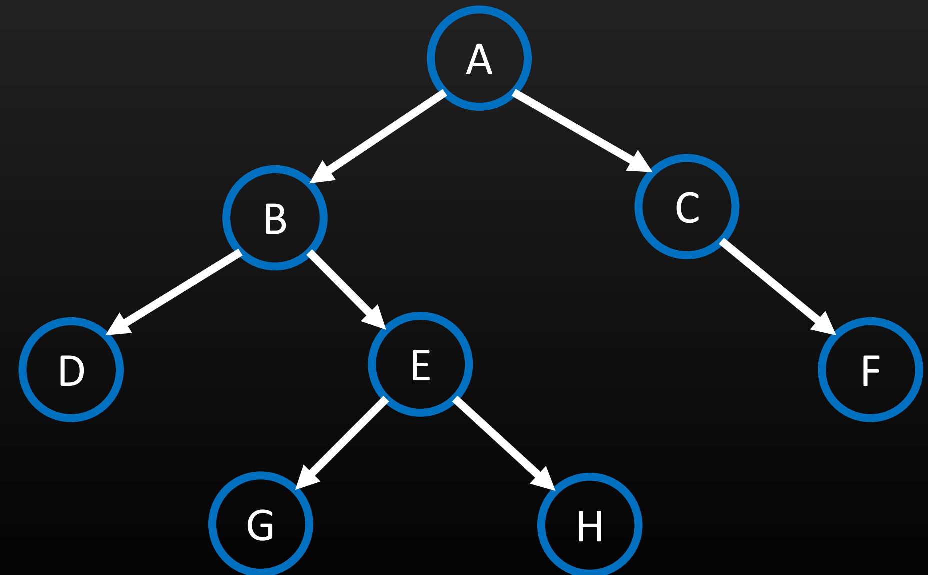


Example 3 – Tree Diameter

```
void dfs(int x, int parent, int dist){
	if (dist >= max_dist){
		deepest_node = x;
		max_dist = dist;
	}
	for (auto i : e[x])
		if (i != parent) dfs(i, x, dist + 1);
}
int tree_diameter(){
	max_dist = 0;
	dfs(1, -1, 0);
	max_dist = 0;
	dfs(deepest_node, -1, 0);
	return max_dist;
}
```

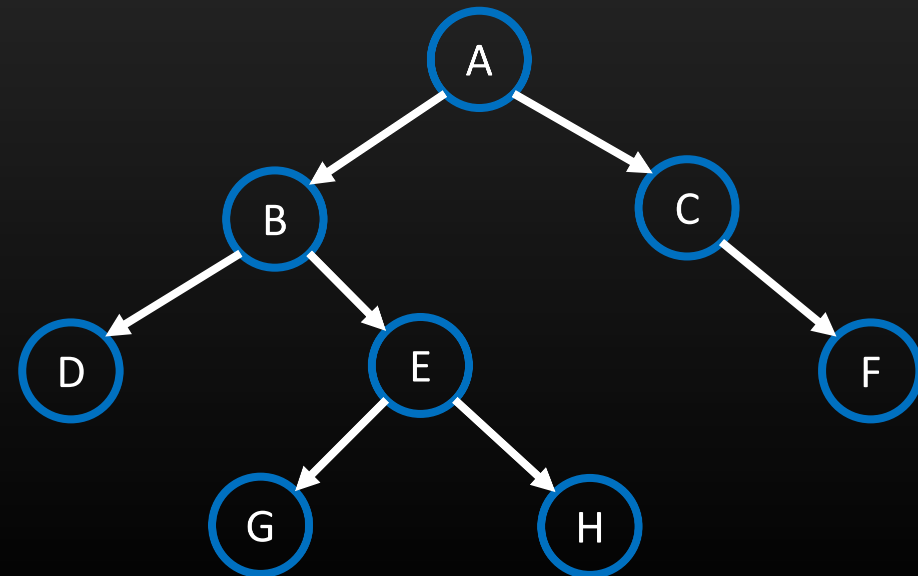
Lowest Common Ancestor (LCA)

- Given a rooted tree, the lowest common ancestor of two nodes *u* and *v* is the node that is the ancestor of both *u* and *v* and has the highest depth
- If one of them is the ancestor of another, it is the LCA
- E.g. LCA(D, H) is B, LCA(A, F) is A



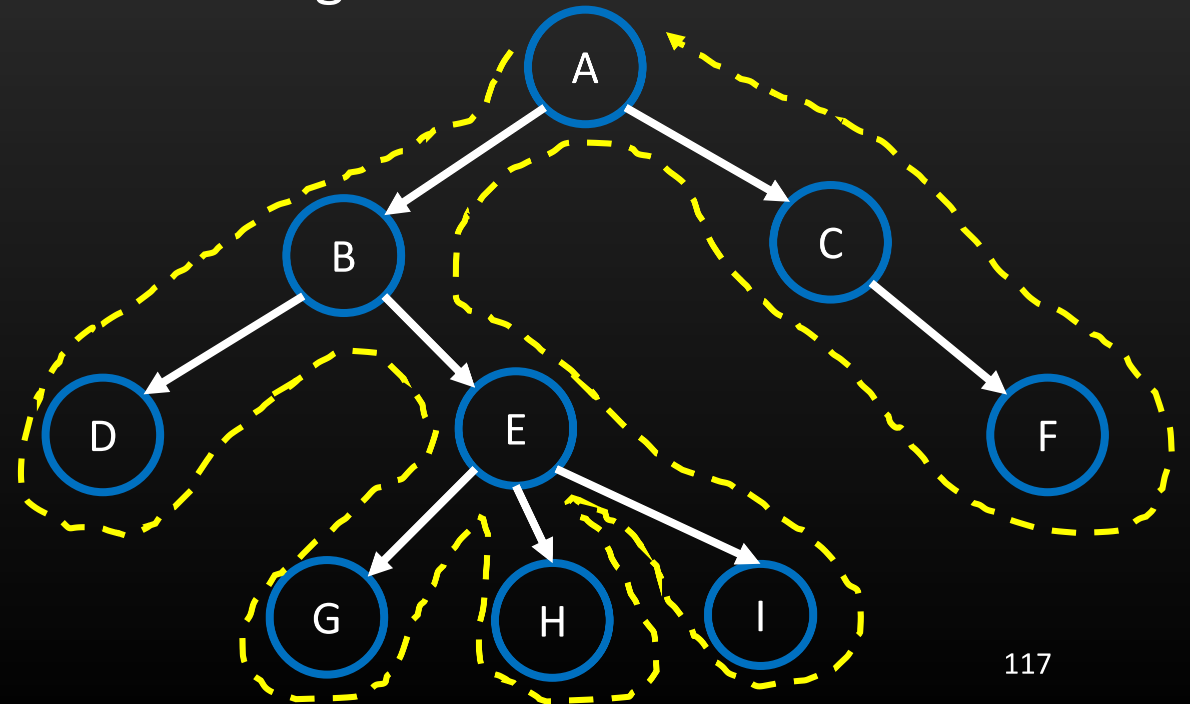
Lowest Common Ancestor (LCA)

- Naïve solutions:
  1. Check the ancestors of the given nodes and return the lowest common one
     - Time complexity for each query: O(N)
  2. Precompute the answers of all pairs by performing DFS starting on each node
     - Time complexity for each query: O(1)
     - Time complexity for precomputation: $O(N^2)$



Lowest Common Ancestor (LCA) – Solution 1

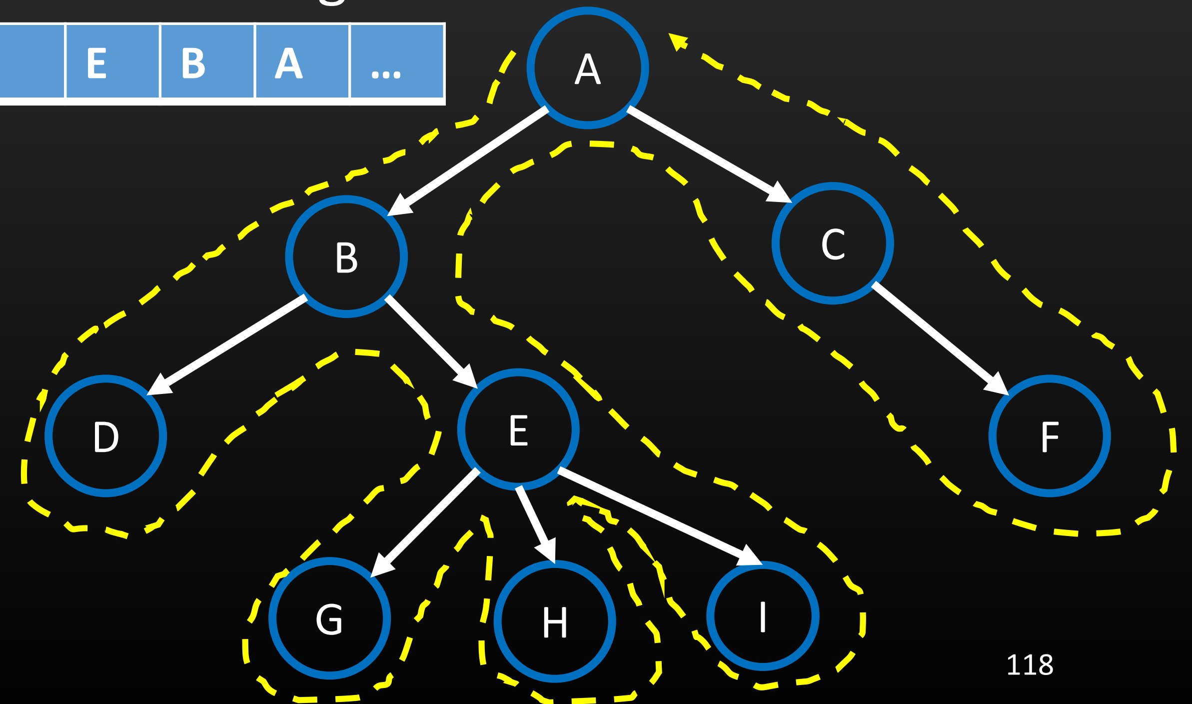
- Perform DFS once and generate the "Euler tour" of the tree
  - Insert the node once when the node starts being visited
  - Insert the node every time when one of its child has been visited
  - Insert the node once when the node finishes being visited



Lowest Common Ancestor (LCA) – Solution 1

- Perform DFS once and generate the "Euler tour" of the tree
  - Insert the node once when the node starts being visited
  - Insert the node every time when one of its child has been visited
  - Insert the node once when the node finishes being visited

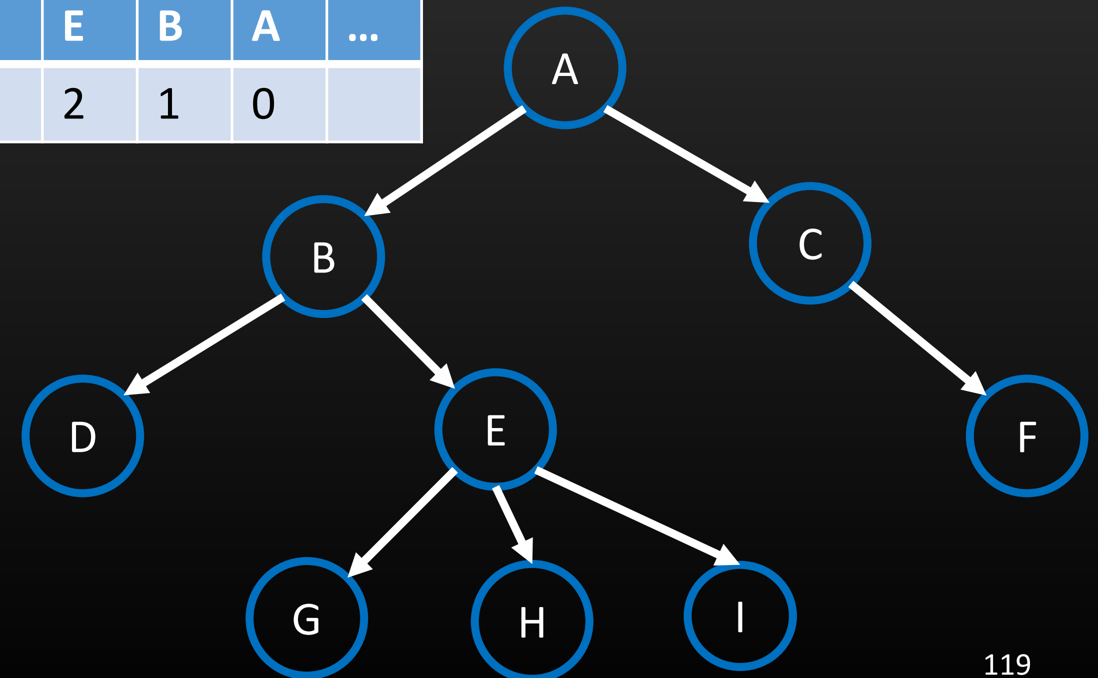
A	B	D	D	B	E	G	G	E	H	H	E	I	I	E	B	A	…



Lowest Common Ancestor (LCA) – Solution 1

- Consider the depths of each node in the order

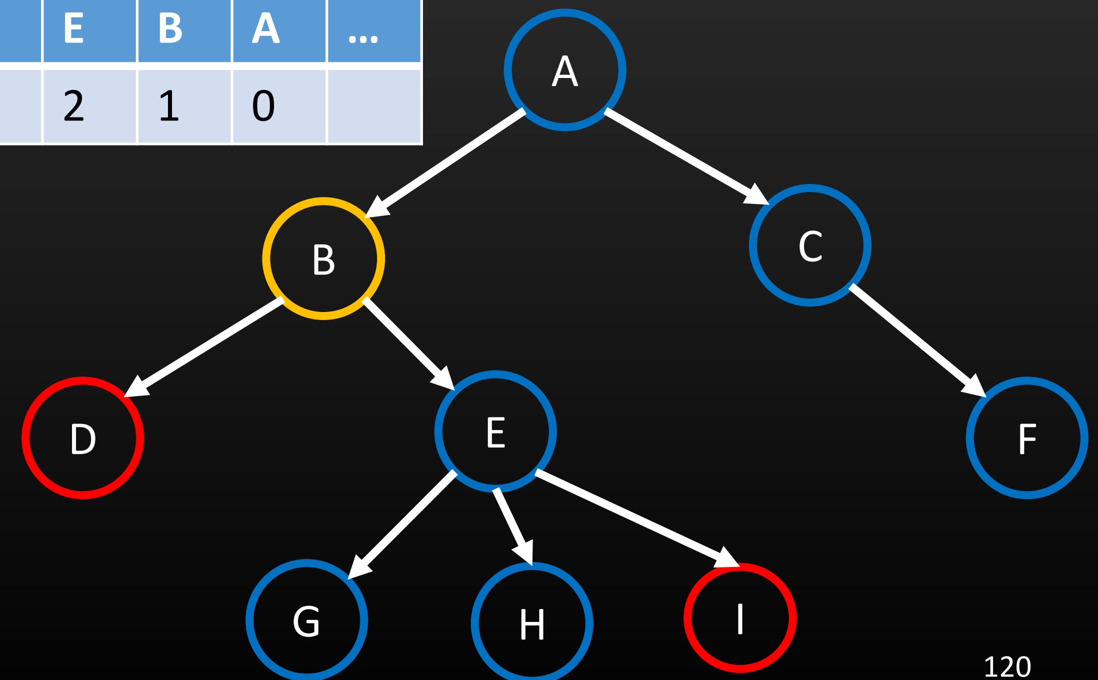
A	B	D	D	B	E	G	G	E	H	H	E	I	I	E	B	A	…
0	1	2	2	1	2	3	3	2	3	3	2	3	3	2	1	0	



Lowest Common Ancestor (LCA) – Solution 1

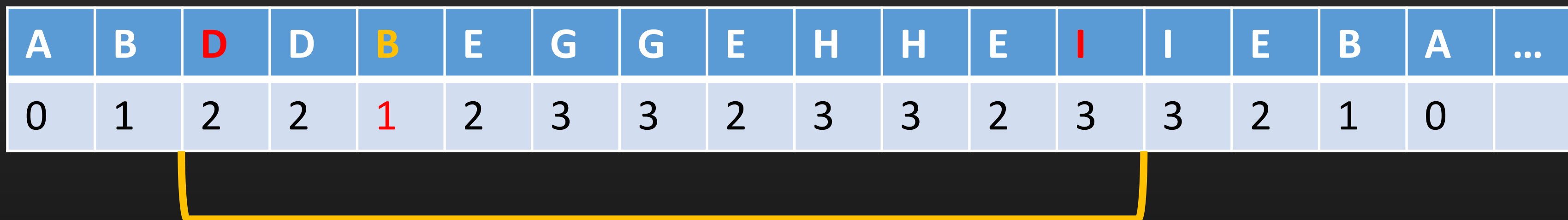
- Consider the depths of each node in the order
- The lowest common ancestor of two nodes always appear between them
- The one with the smallest depth inside the subarray

A	B	D	D	B	E	G	G	E	H	H	E	I	I	E	B	A	…
0	1	2	2	1	2	3	3	2	3	3	2	3	3	2	1	0	



Lowest Common Ancestor (LCA) – Solution 1

- Become a Range Minimum Query problem
- Can be solved using Segment Tree or Sparse Table



A	B	D	D	B	E	G	G	E	H	H	E	I	I	E	B	A	…
0	1	2	2	1	2	3	3	2	3	3	2	3	3	2	1	0	

- Time complexity of precomputation: O(N log N)
- Time complexity of each query: O(log N) for segment tree, O(1) for sparse table
- Space complexity: O(N) for segment tree, O(N log N) for sparse table

Lowest Common Ancestor (LCA) – Solution 1

- Precomputation:
  1. Generate the Euler Tour with depths of each node using DFS
  2. Store the first occurrence of each node in the Tour
  3. Construct the Segment Tree or Sparse Table from the Tour

- Query:
  - Return the node with the smallest depth within the sub-array between the first occurrence of the two given nodes

Lowest Common Ancestor (LCA) – Solution 2

- One of the naïve solution is to check the ancestors of the given nodes and return the lowest common one
- Slow because it may take O(N) time to visit the ancestors one by one

- How to speed up the process?

Lowest Common Ancestor (LCA) – Solution 2

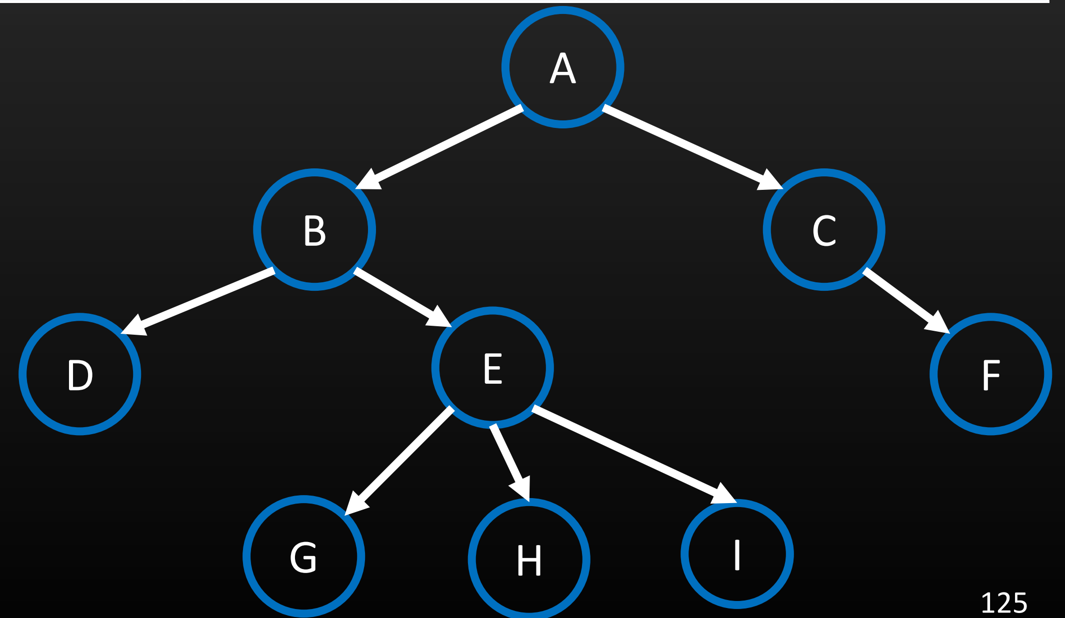
- Precompute the 1st, 2nd, 4th, 8th, …, 2^{x} th ancestor of each node
- This can be computed quickly, as the 2^{x} th ancestor of the node *u* is the 2^{x-1} th ancestor of the 2^{x-1} th ancestor of *u*

- Compute the 1st ancestor of each node, then compute the 2nd ancestor of each node, then the 4th ancestor, and so on

Lowest Common Ancestor (LCA) – Solution 2

Node	A	B	C	D	E	F	G	H	I
1st Ancestor	A	A	A	B	B	C	E	E	E
2nd Ancestor									
4th Ancestor									
8th Ancestor									

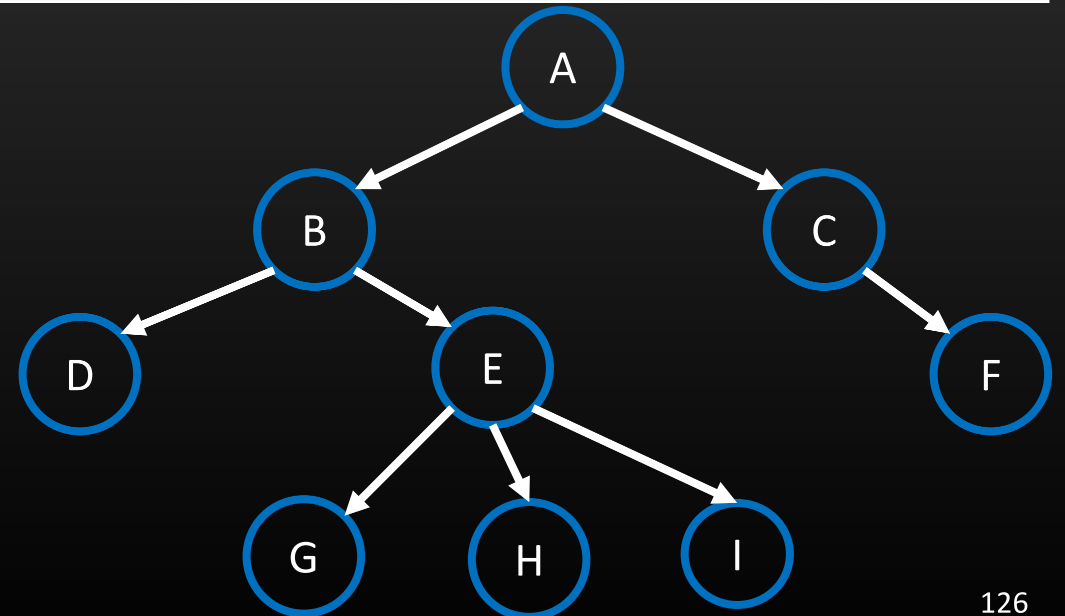
For simplicity, the ancestor = the root (A) when it does not have that much ancestor



Lowest Common Ancestor (LCA) – Solution 2

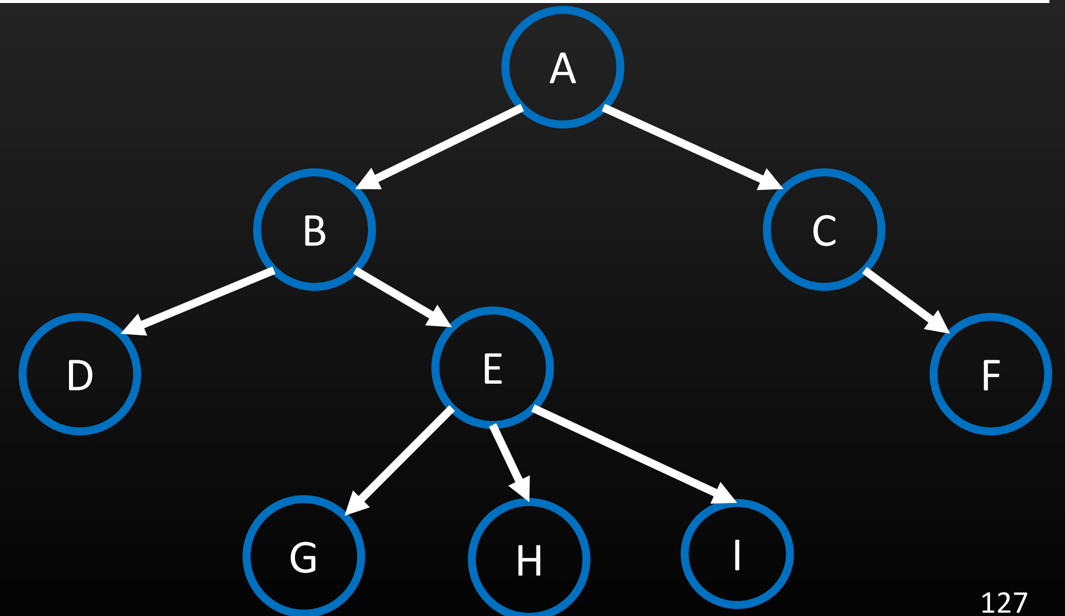
Node	A	B	C	D	E	F	G	H	I
1st Ancestor	A	A	A	B	B	C	E	E	E
2nd Ancestor	A	A	A	A	A	A	B	B	B
4th Ancestor									
8th Ancestor									

e.g. 2nd Ancestor of G
= 1st Ancestor of (1st Ancestor of G)
= 1st Ancestor of E
= B



Lowest Common Ancestor (LCA) – Solution 2

Node	A	B	C	D	E	F	G	H	I
1st Ancestor	A	A	A	B	B	C	E	E	E
2nd Ancestor	A	A	A	A	A	A	B	B	B
4th Ancestor	A	A	A	A	A	A	A	A	A
8th Ancestor	A	A	A	A	A	A	A	A	A

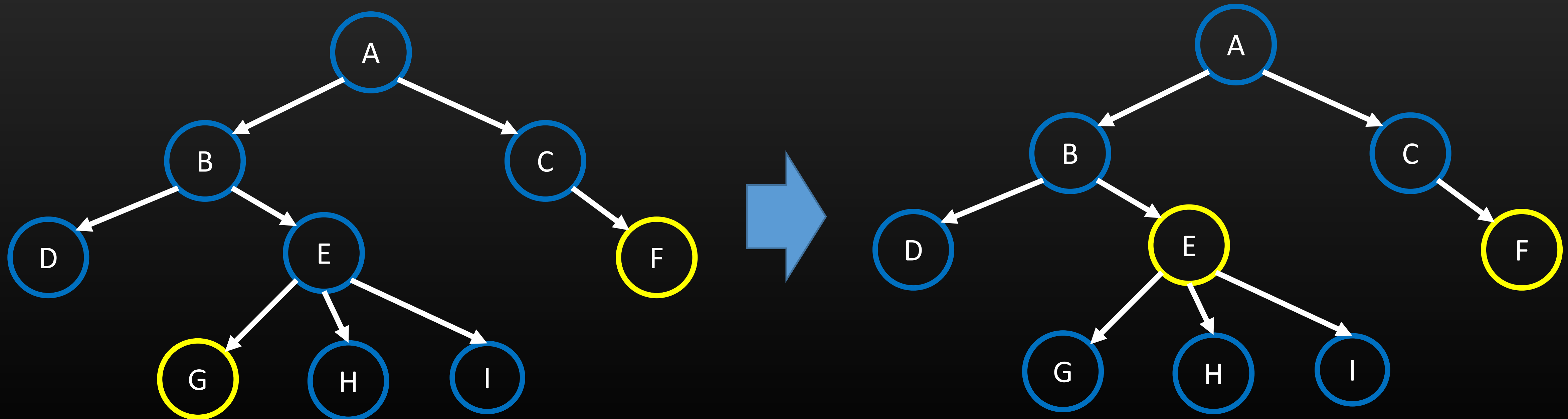


Lowest Common Ancestor (LCA) – Solution 2

- Using these 2^x th ancestor, one can find the yth ancestor in O(log N) time for any y
- E.g. the 11th ancestor of a node = the 8th ancestor of the 2nd ancestor of the 1st ancestor of the node
- It is called Binary Lifting

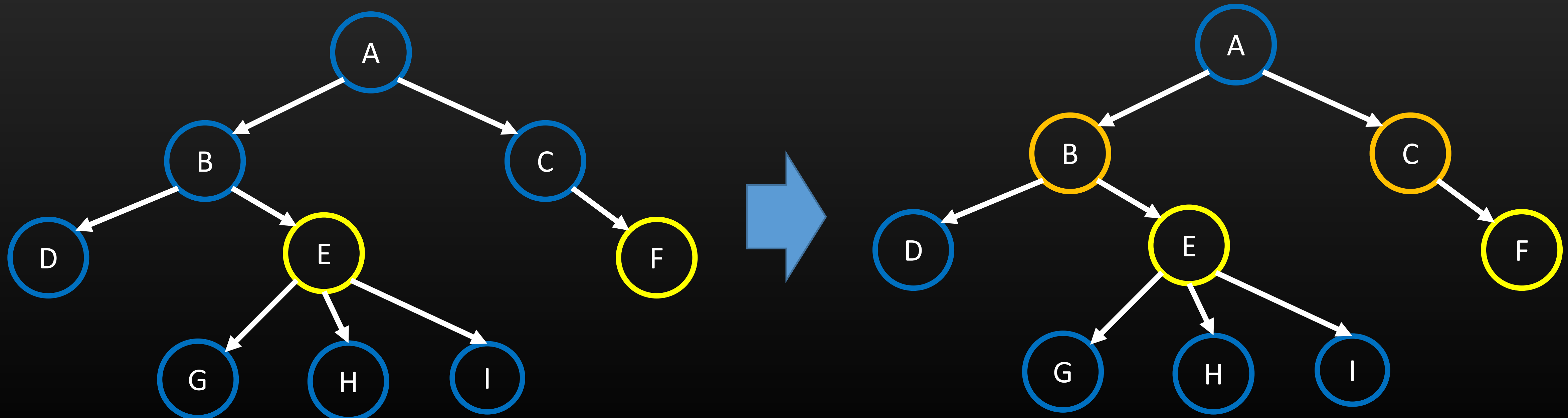
Lowest Common Ancestor (LCA) – Solution 2

- Compute the LCA of node *u* and *v* (let depth[u] >= depth[v])
- First we find the ancestor of *u* that has the same depth with *v* using binary lifting



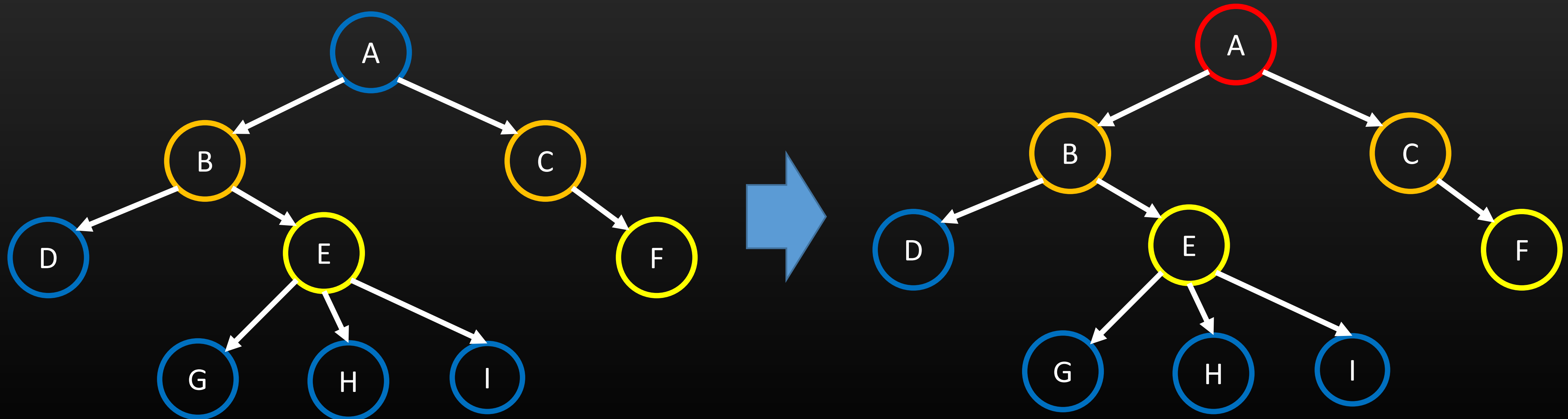
Lowest Common Ancestor (LCA) – Solution 2

- Then we find the highest ancestors of each node that are not common similar to binary search
  - If the 2^x th ancestor of *u* != the 2^x th ancestor of *v* then move *u* and *v* up as their 2^x th ancestors, with x changing decreasingly until 0



Lowest Common Ancestor (LCA) – Solution 2

- The LCA will be the parent of the two nodes



Lowest Common Ancestor (LCA) – Solution 2

- Precomputation:
  1. Compute the depths of each node using DFS
  2. Compute the 1, 2, 4, …, 2^{x} th ancestors of each node

- Query:
  1. Lift the lower node until both nodes have the same depth
  2. Lift the nodes to the highest ancestors that are not common
  3. Return the parent of the nodes

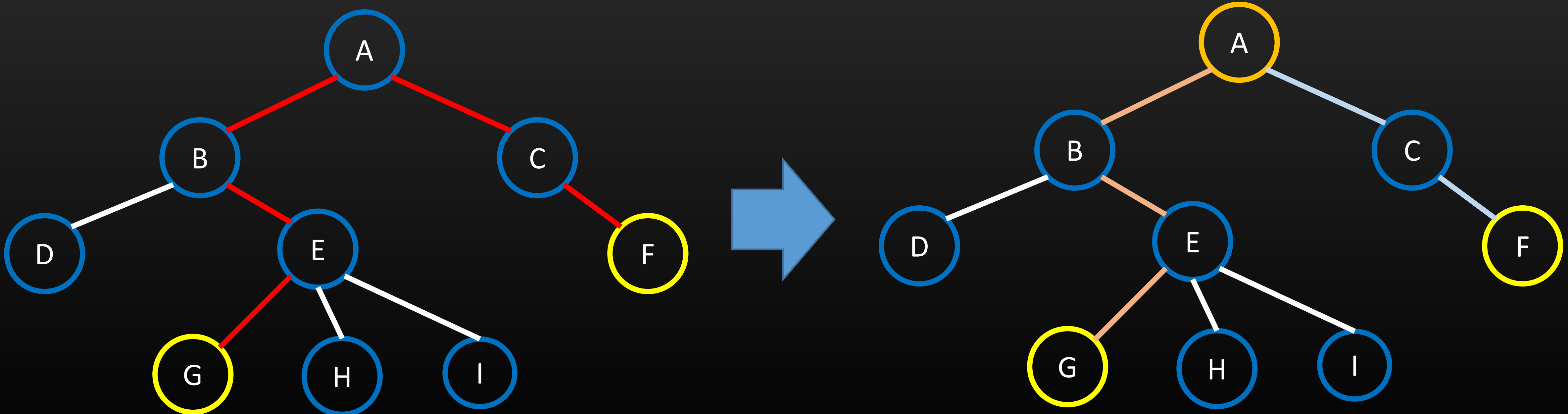
Time complexity: O(N log N) for precomputation, O(log N) for each query

Lowest Common Ancestor (LCA) – Solution 2

```
int lca(int u, int v){
	if (depth[u] < depth[v]) swap(u, v);
	for (int i = m; i >= 0; i--)
		if (depth[u] – (1 << i) >= depth[v])
			u = ancestors[u][i];
	if (u == v) return v;
	for (int i = m; i >= 0; i--)
		if (ancestors[u][i] != ancestors[v][i]){
			u = ancestors[u][i];
			v = ancestors[v][i];
		}
	return ancestors[u][0];
}
```

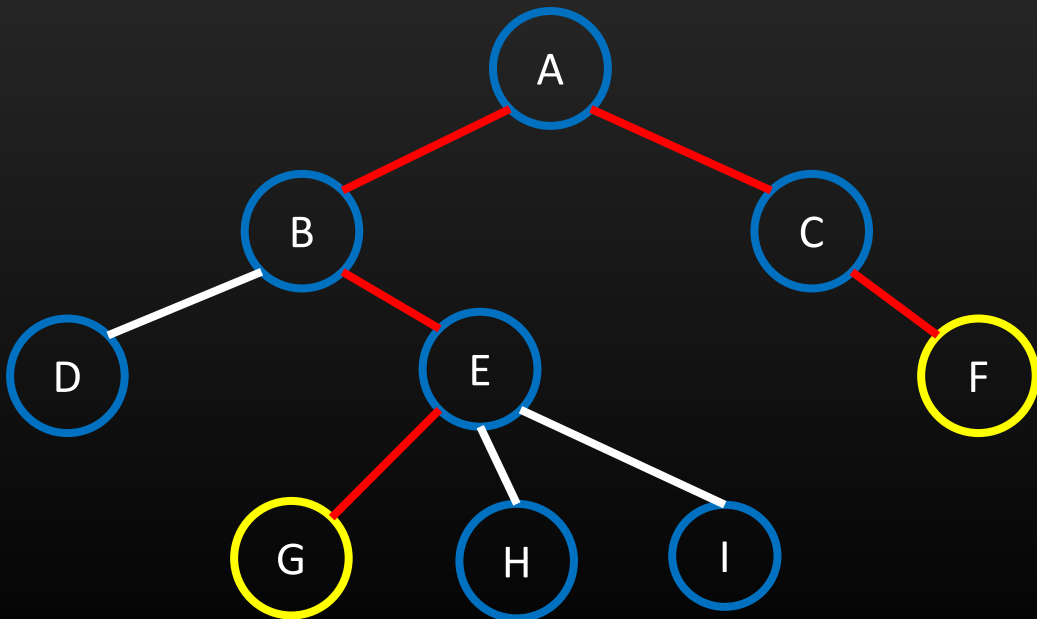
Lowest Common Ancestor (LCA)

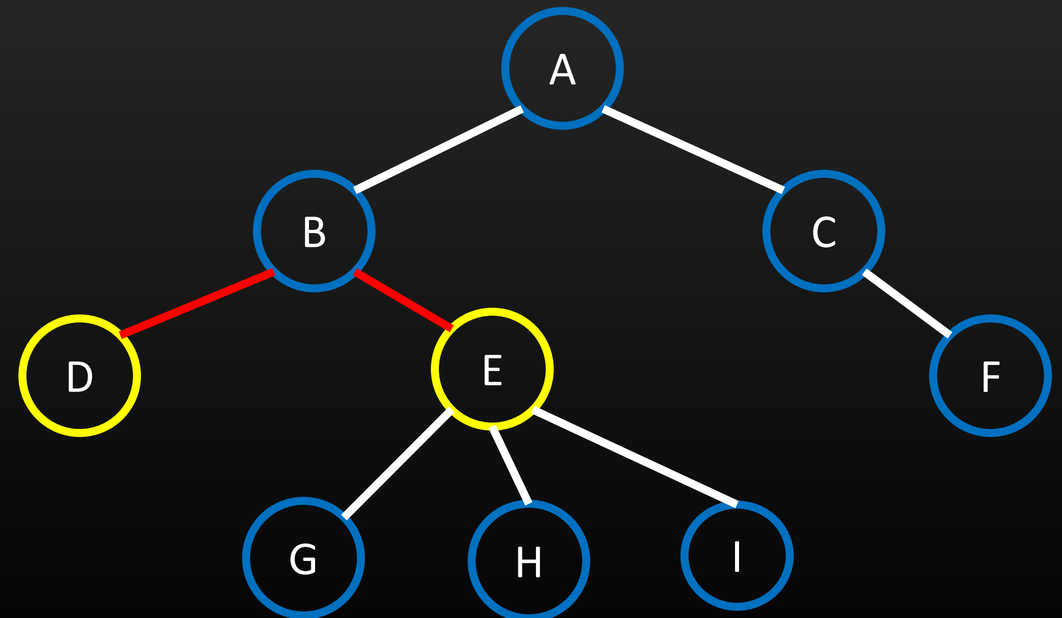
- LCA is useful to solve problems related to paths in tree
- The paths *u*-*v* can be split into two paths *u*-LCA(*u, v*) and LCA(*u, v)*-*v*, which can usually be handled easier
  - Easier queries, easier update, easier precomputation, …



Lowest Common Ancestor (LCA) - Example

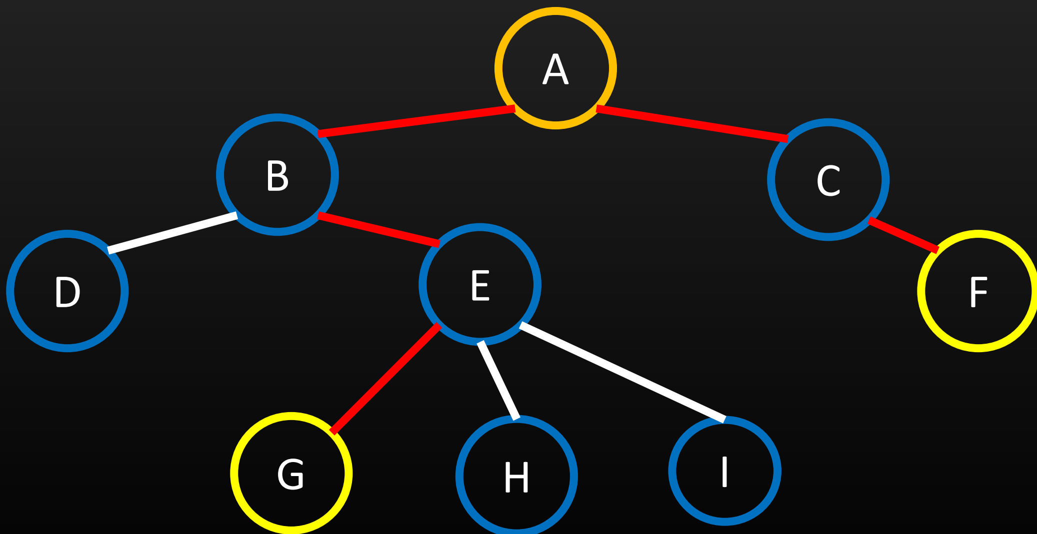
- Given a tree, answer the queries that ask the distance of two given nodes
- E.g. Query(G, F) = 5, Query(D, E) = 2





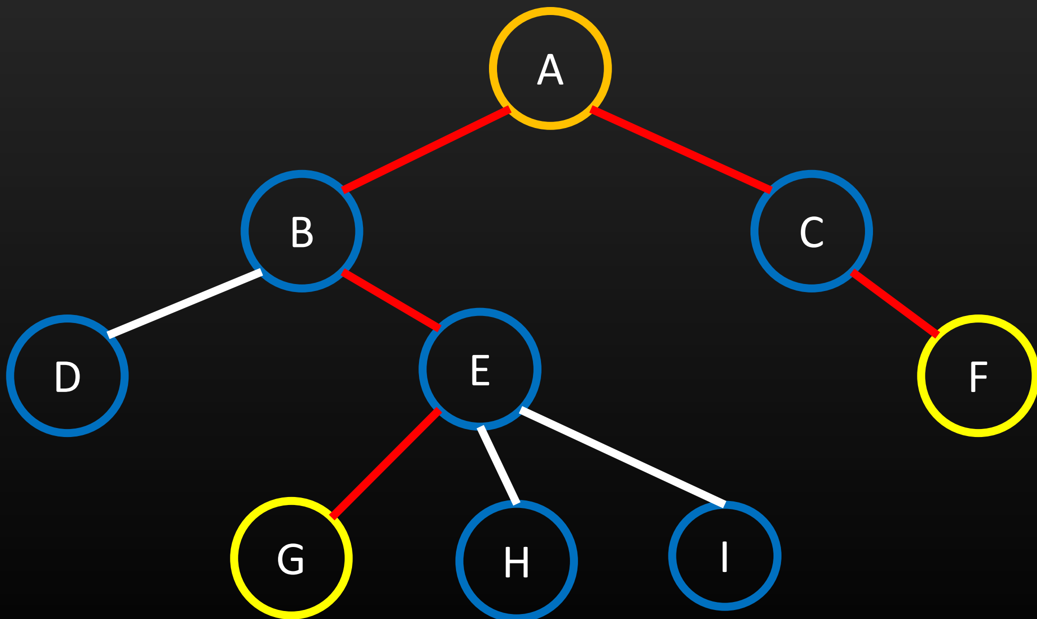
Lowest Common Ancestor (LCA) - Example

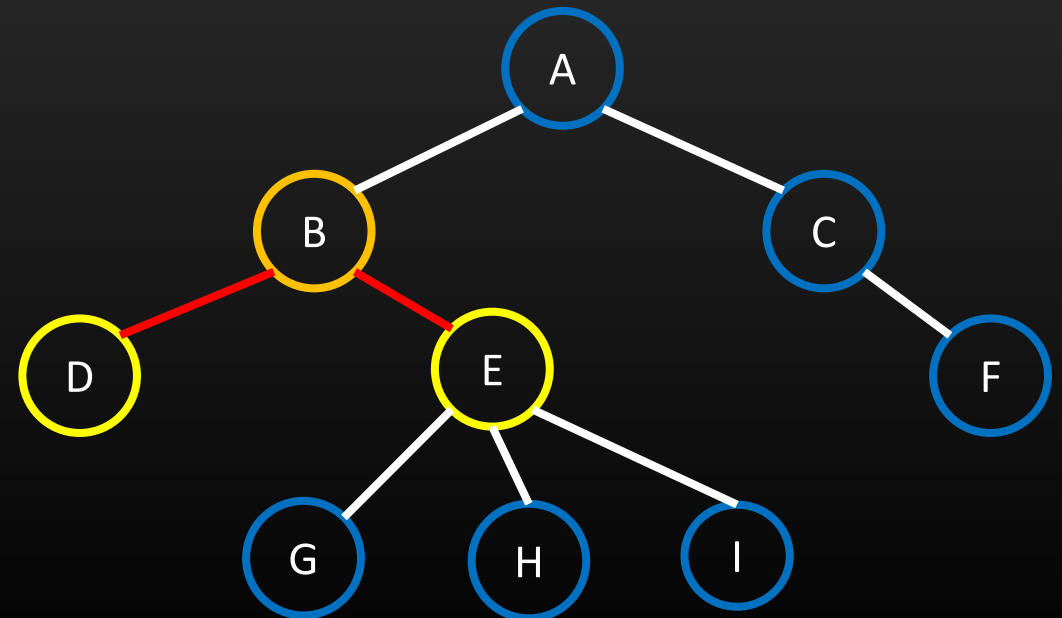
- Randomly choose a node as the root
- Compute the LCA of the given nodes
- The path can be split into two paths starting from the LCA, with the distance equal to the difference of the depth of the node and the LCA



Lowest Common Ancestor (LCA) - Example

- Query(G, F) = (Distance from G to A) + (Distance from A to F) = (Depth of G – Depth of A) + (Depth of F – Depth of A) = (3 – 0) + (2 – 0) = 5
- Query(D, E) = (Distance from D to B) + (Distance from B to E) = (Depth of D – Depth of B) + (Depth of E – Depth of B) = (2 – 1) + (2 – 1) = 2





Practice Problems

- HKOJ 01038
- HKOJ 01040
- HKOJ S042

- HKOJ M0642
- Codeforces 191C
- Codeforces 208E