BREADTH FIRST SEARCH

bfs

Breadth-first search

- What is it good for?
- We have a graph and we want to visit every node → we can do it with BFS
- We visit every vertex exactly once
- We visit the neighbours then the neighbours of these new vertices and so on
- Running time complexity: **O(V+E)**
- Memory complexity is not good: we have to store lots of references
- Thats why DFS is usually preferred
- BUT it constructs a shortest path: Dijkstra algorithm does a BFS if all the edge weights are equal to 1

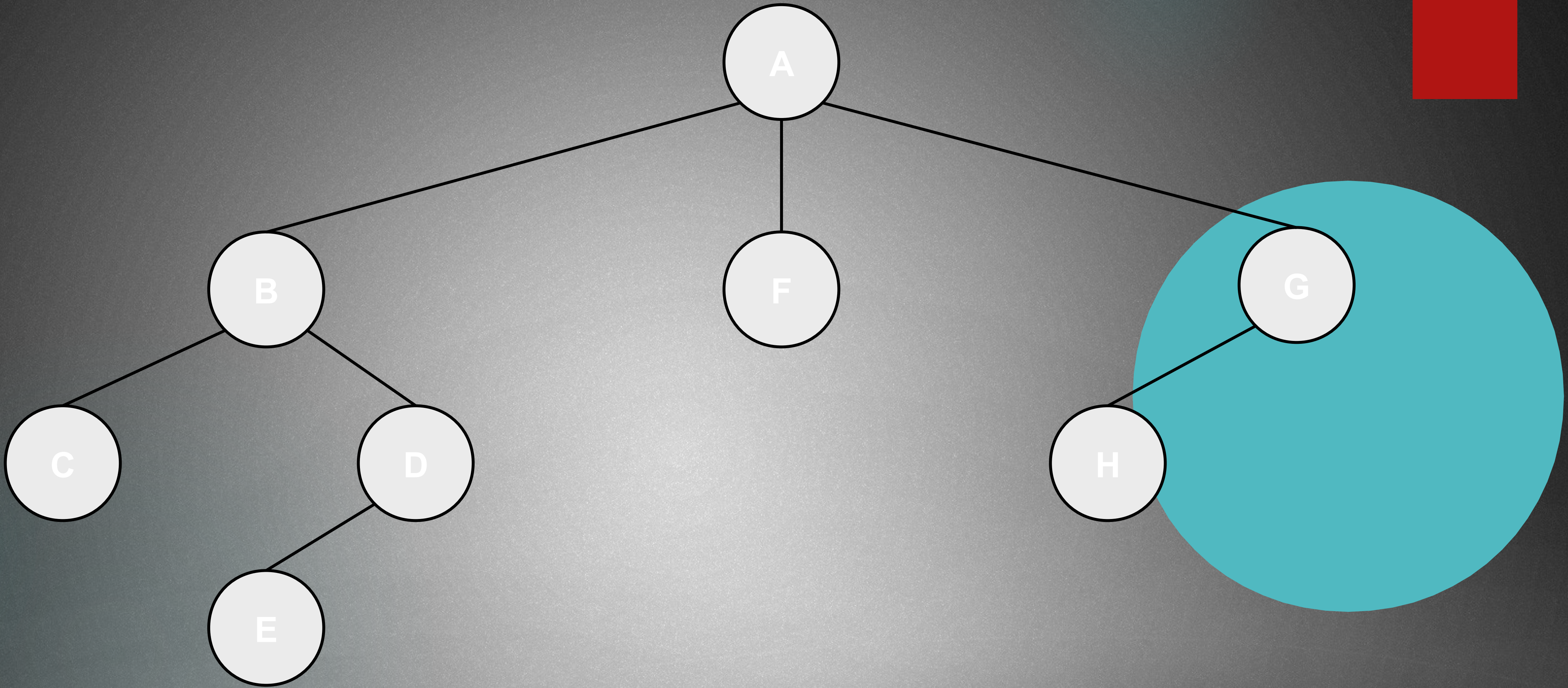
Breadth-first search

```
bfs(vertex)

    Queue queue
    vertex set visited true
     queue.enqueue(vertex)

    while queue not empty
        actual = queue.dequeue()

        for v in actual neighbours
            if v is not visited
                v set visited true
                queue.enqueue(v)
```

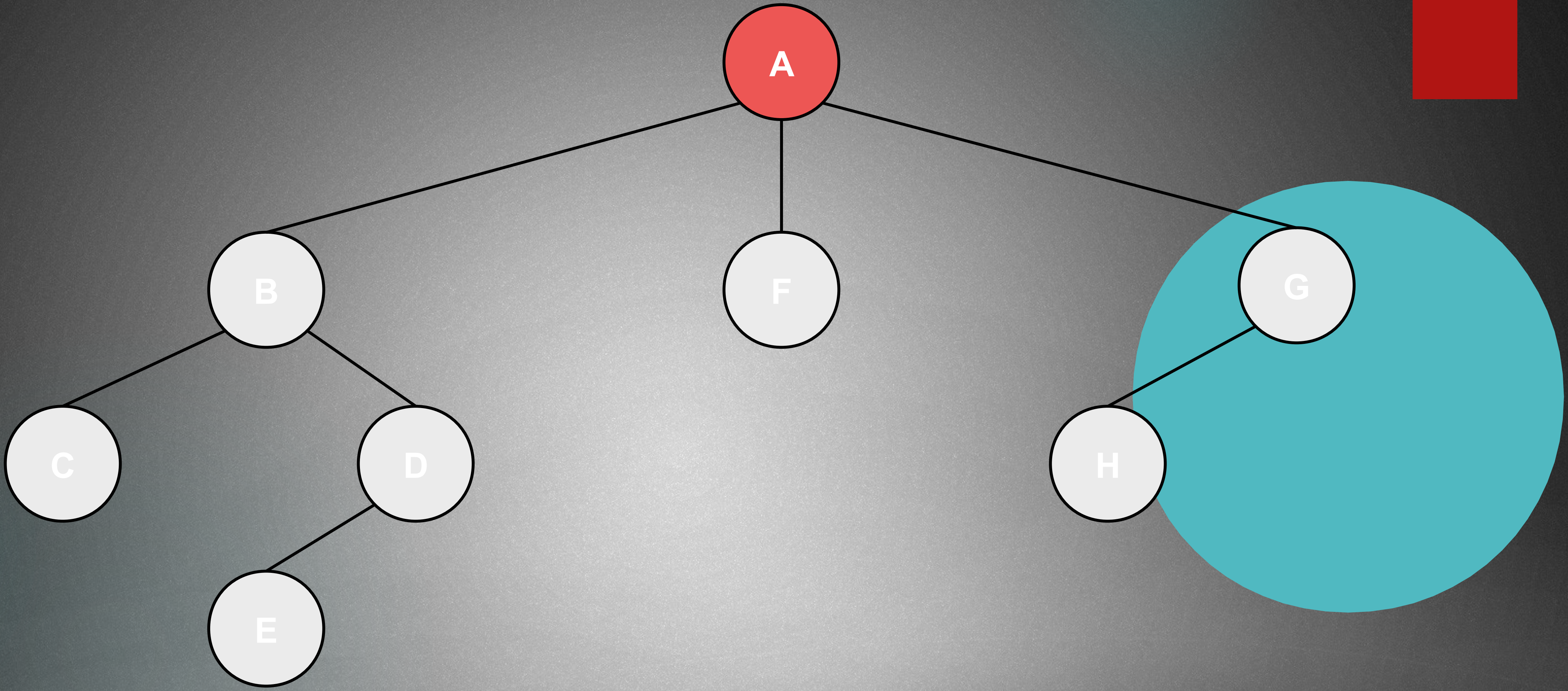
ITERATION

We have an empty queue at the beginning and we keep checking whether we have visited the given node or not

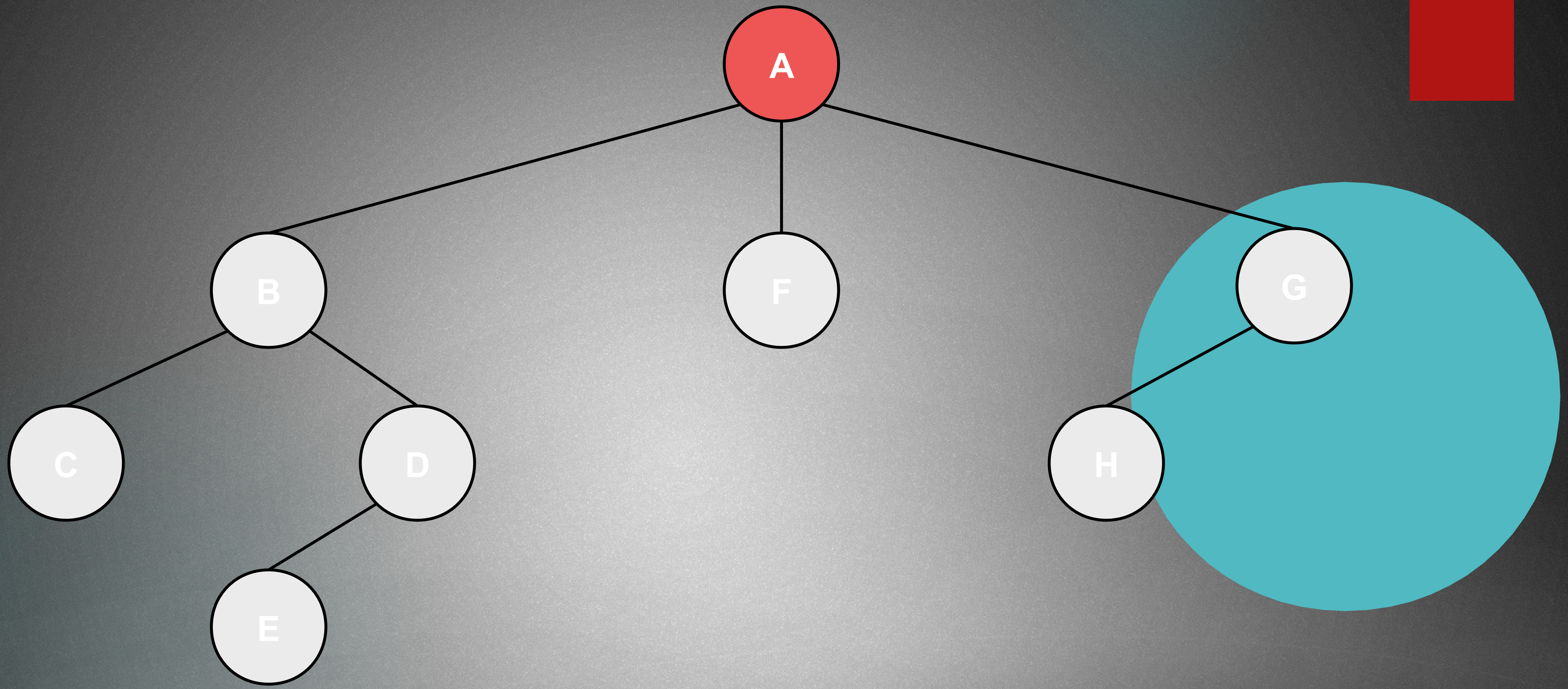
~ keep iterating until queue is not empty



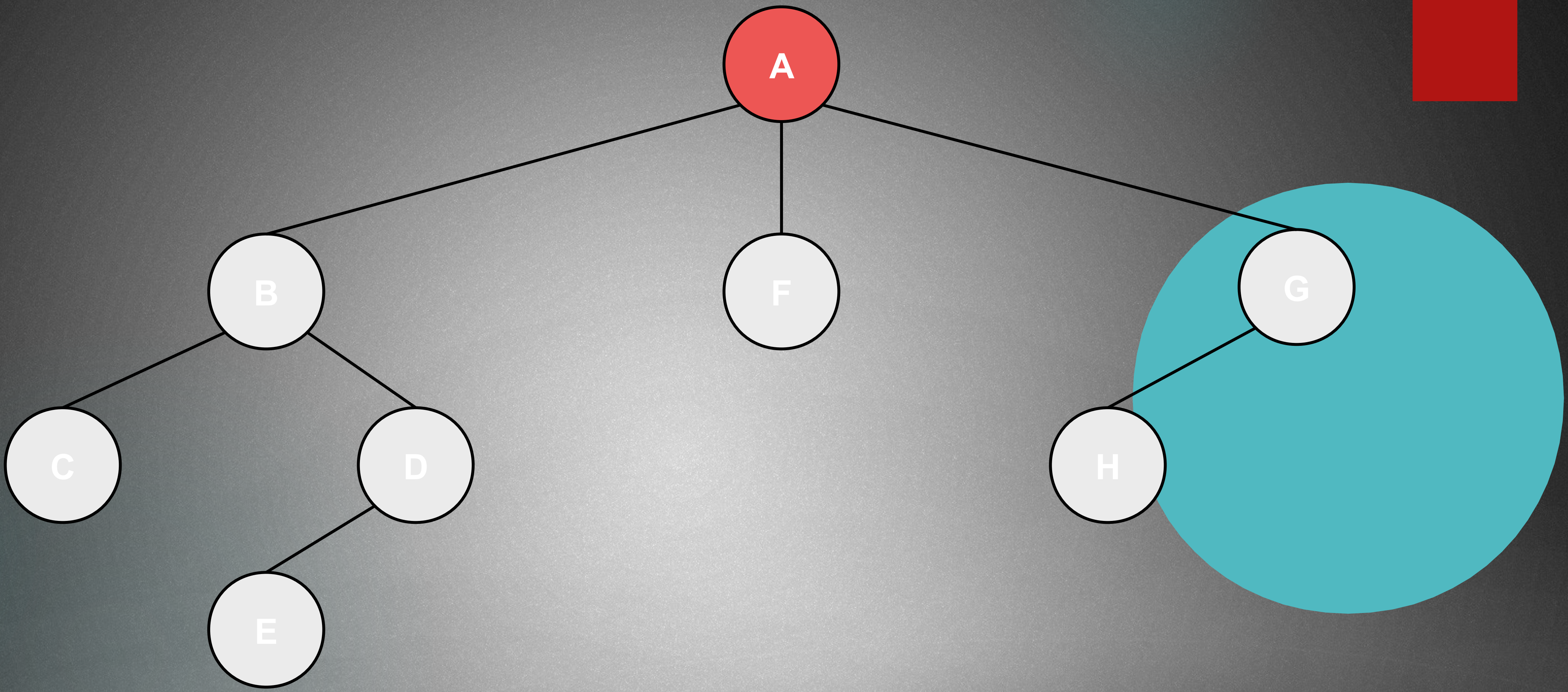
Queue: { }



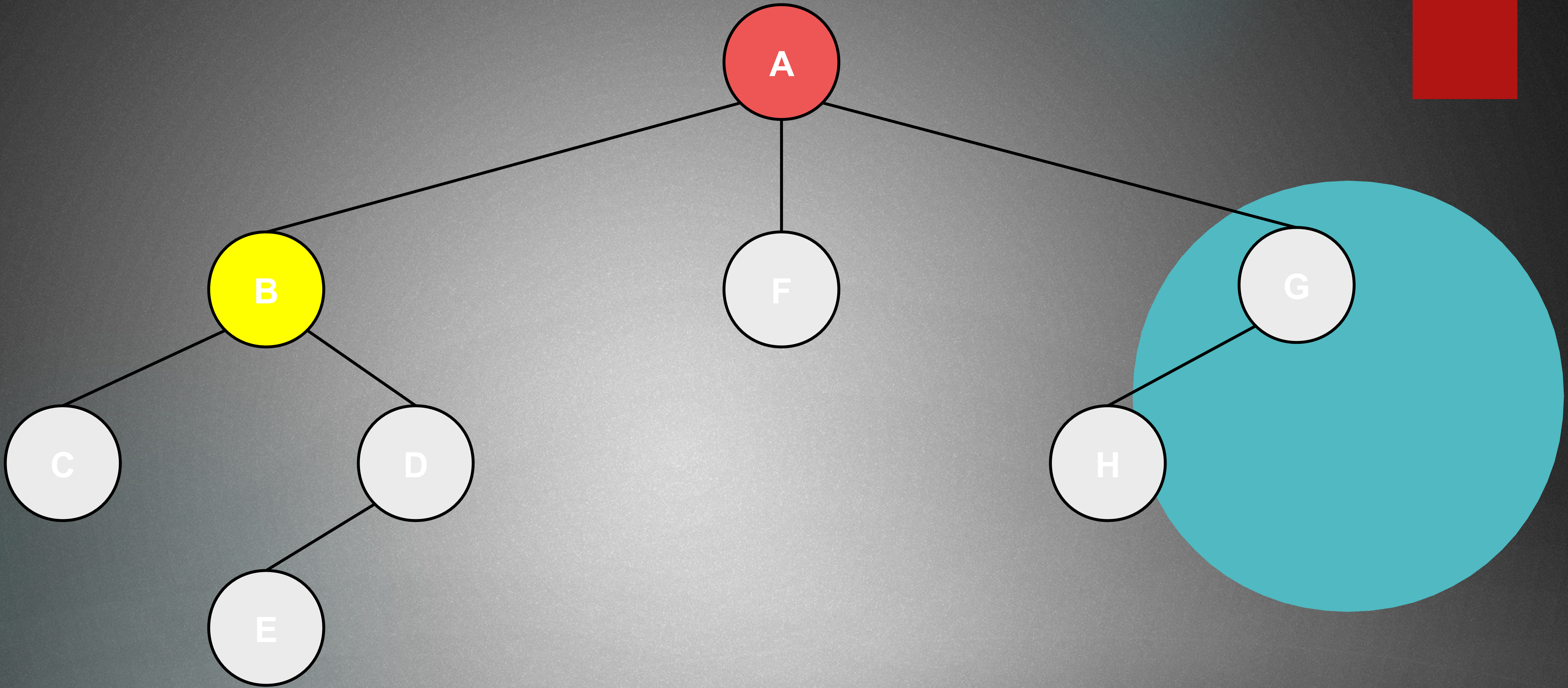
Queue: { A }



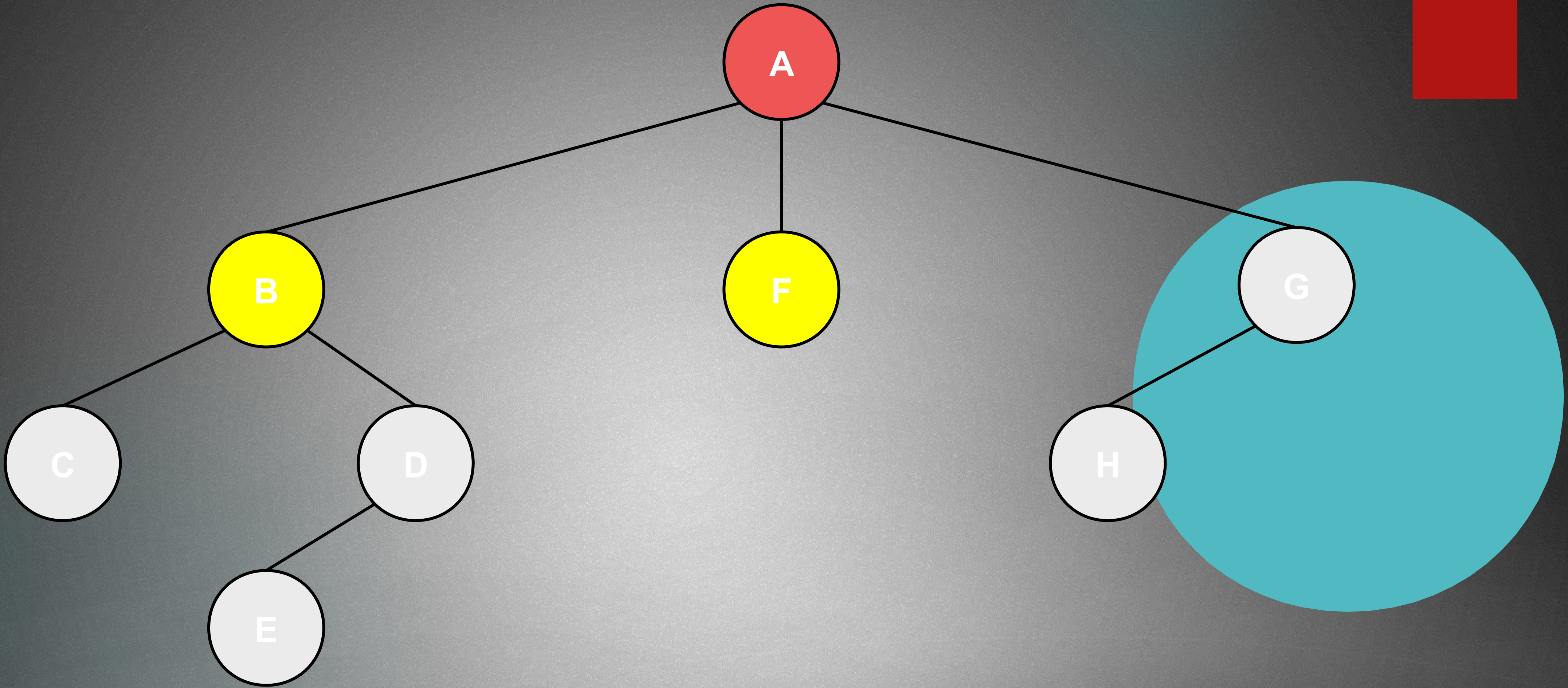
Queue: { **A** }   dequeue node A to be able to process its



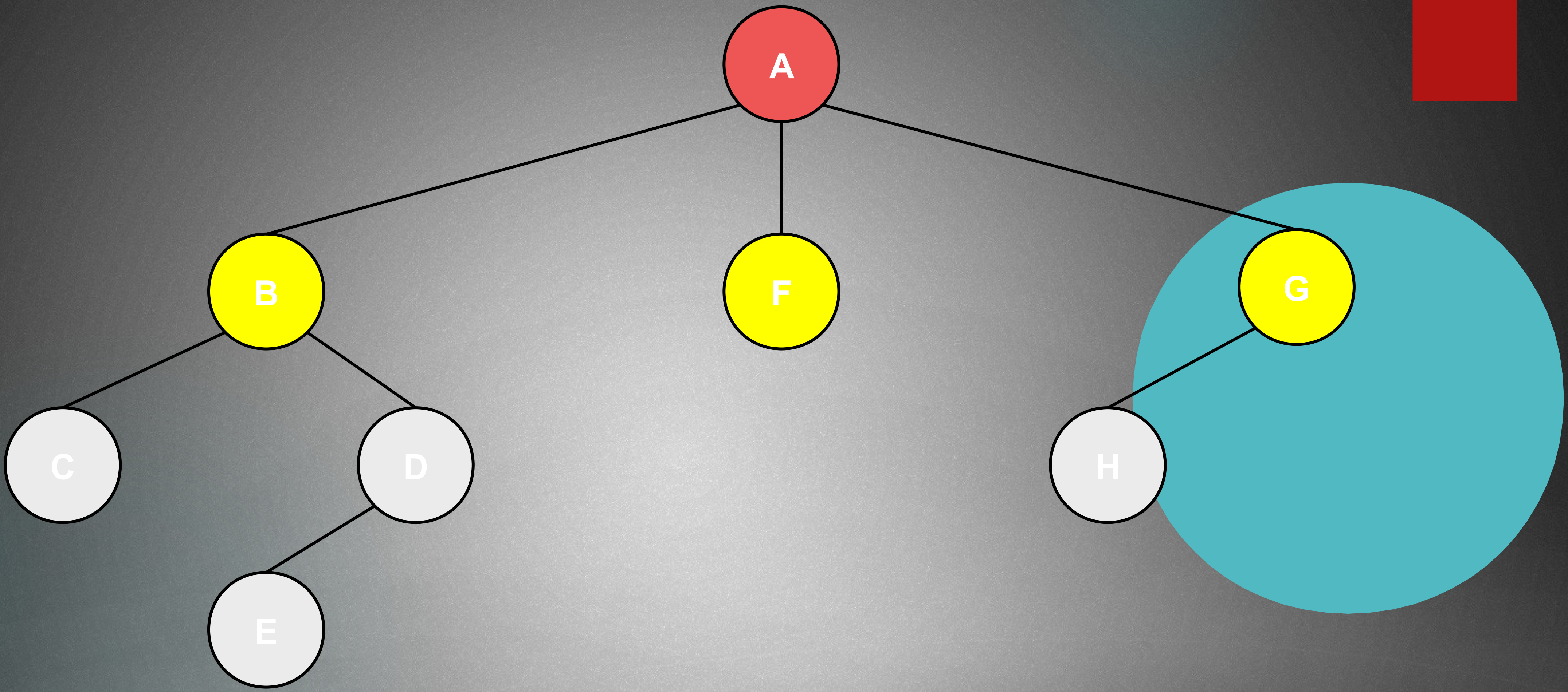
Queue: { } dequeue node A to be able to process its neighbours → visit all the children, if a node is unvisited then enqueue it !!!



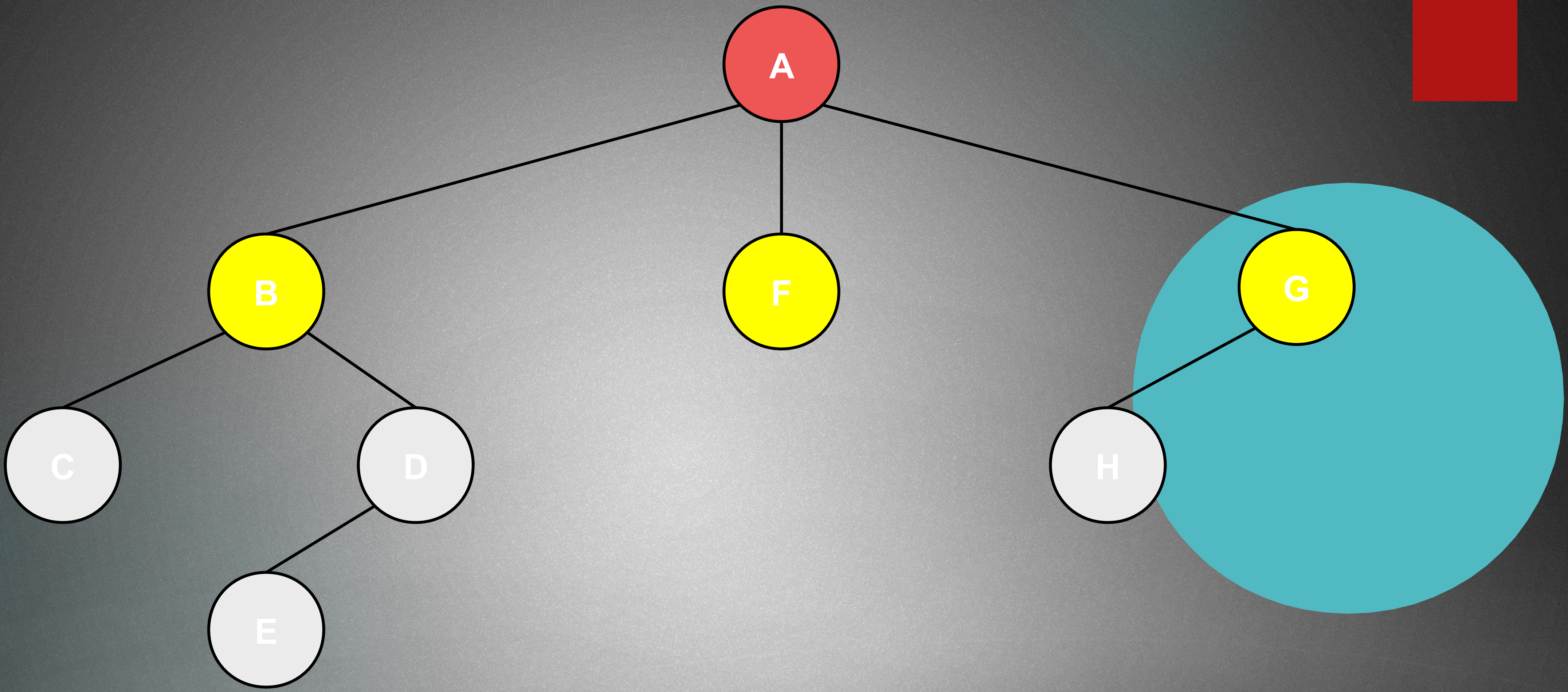
Queue: { B }   node B is not visited → put it to the queue



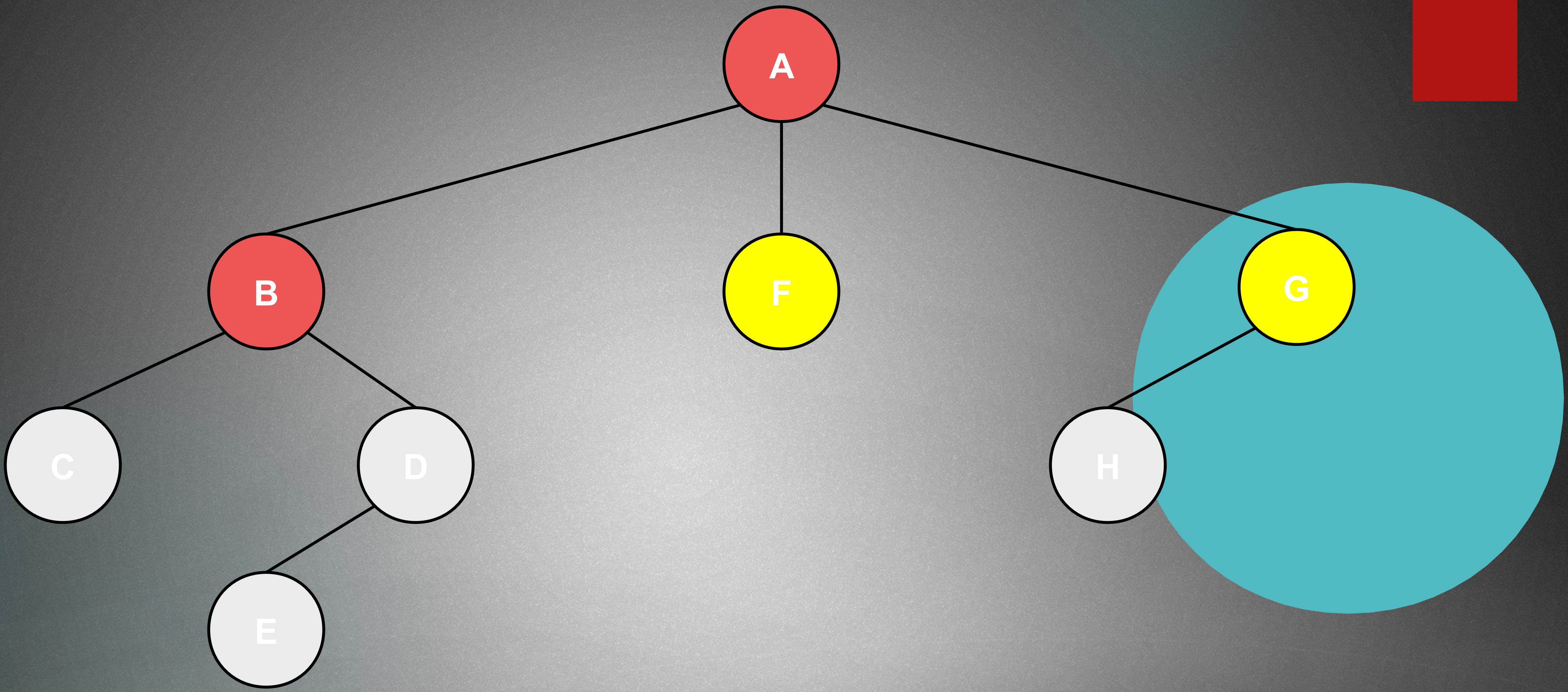
Queue: { F B }   node F is not visited → put it to the queue



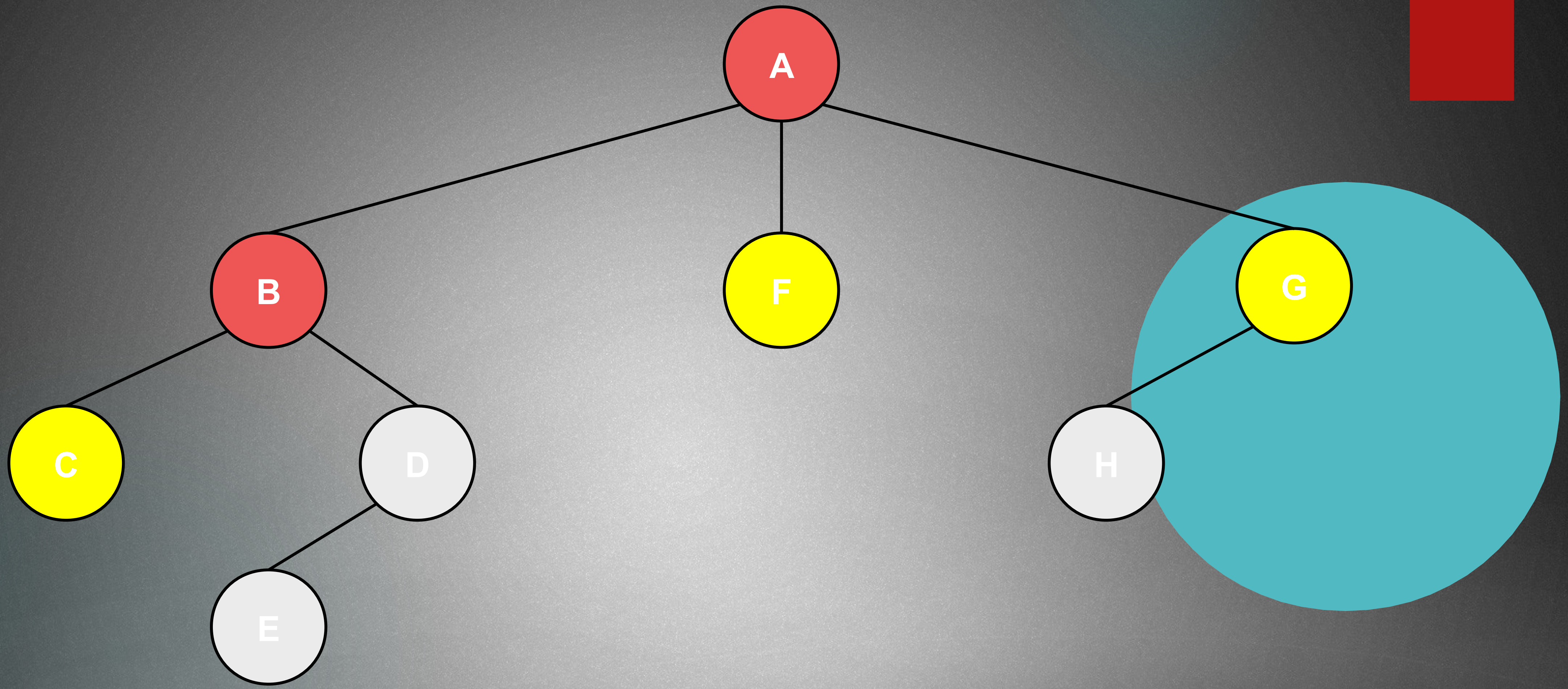
Queue: { G F B }   node G is not visited → put it to the queue



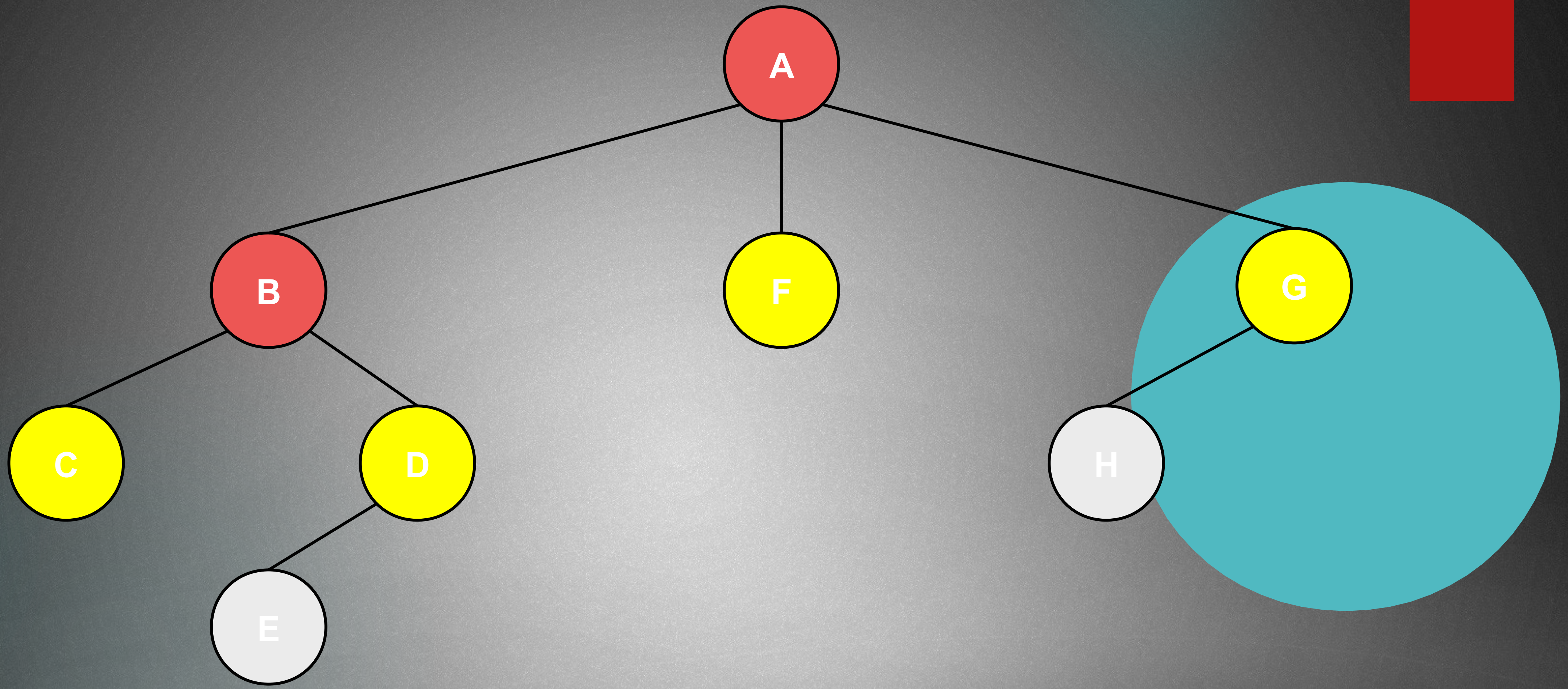
Queue: { G F **B** } dequeue the next node → it is B and visit its children + put them into the queue if necessary



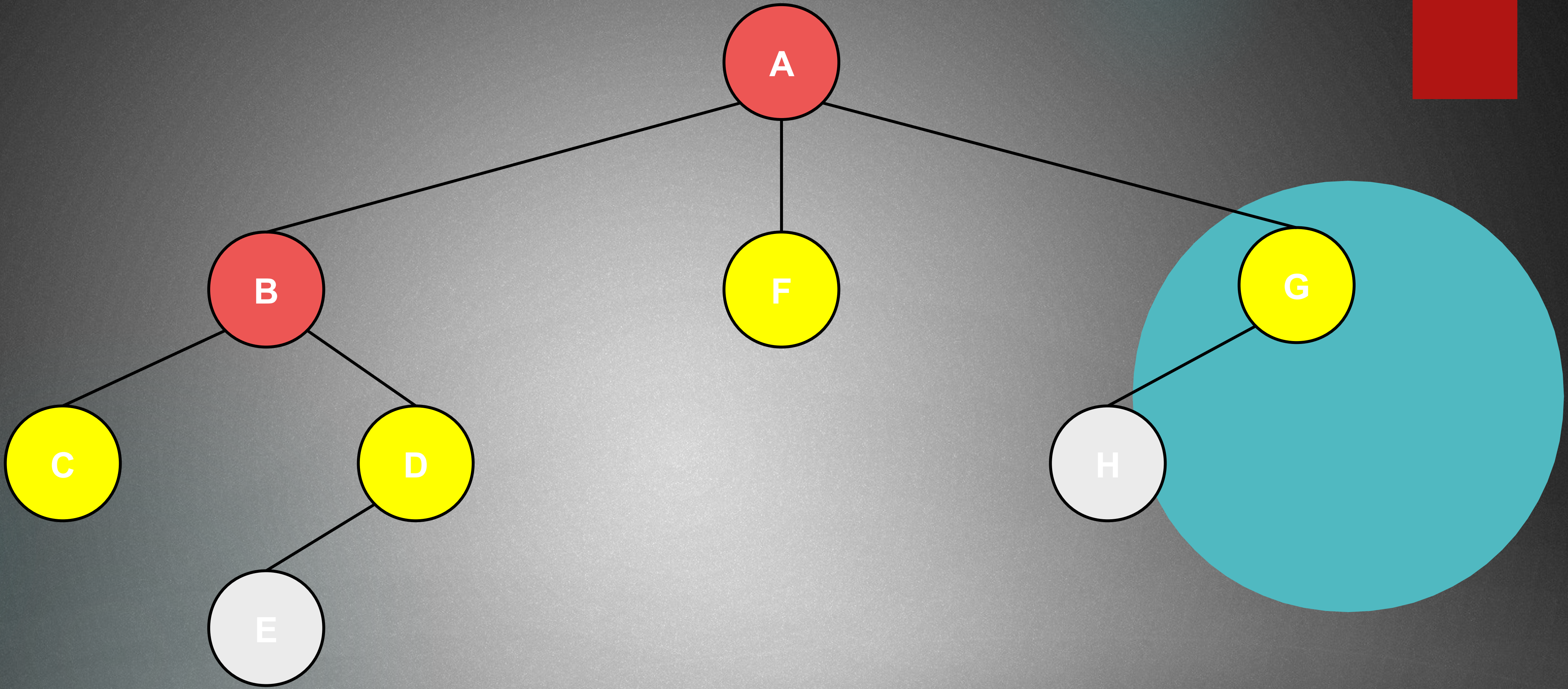
Queue: { G F }	dequeue the next node → it is G and visit its children + put them into the queue if necessary



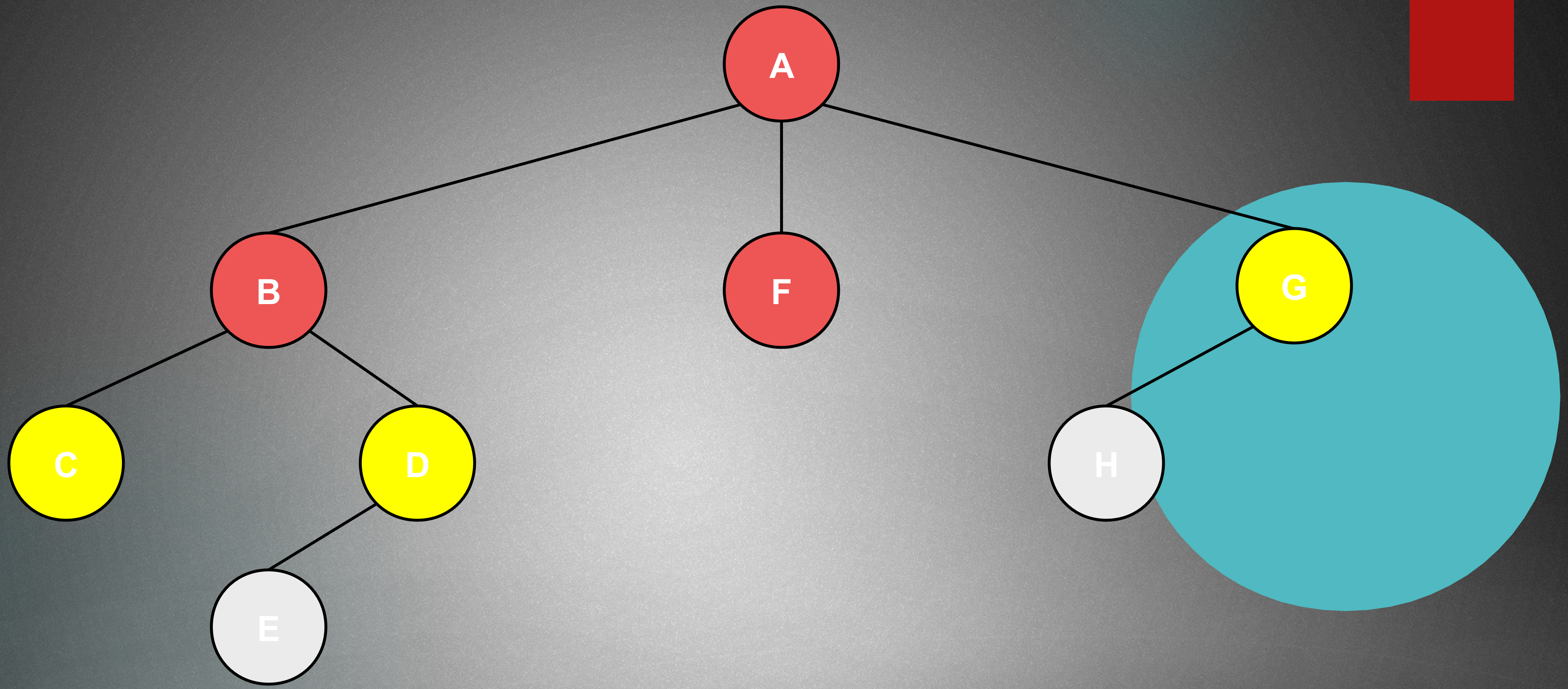
Queue: { C G F } dequeue the next node → it is G and visit its children + put them into the queue if necessary



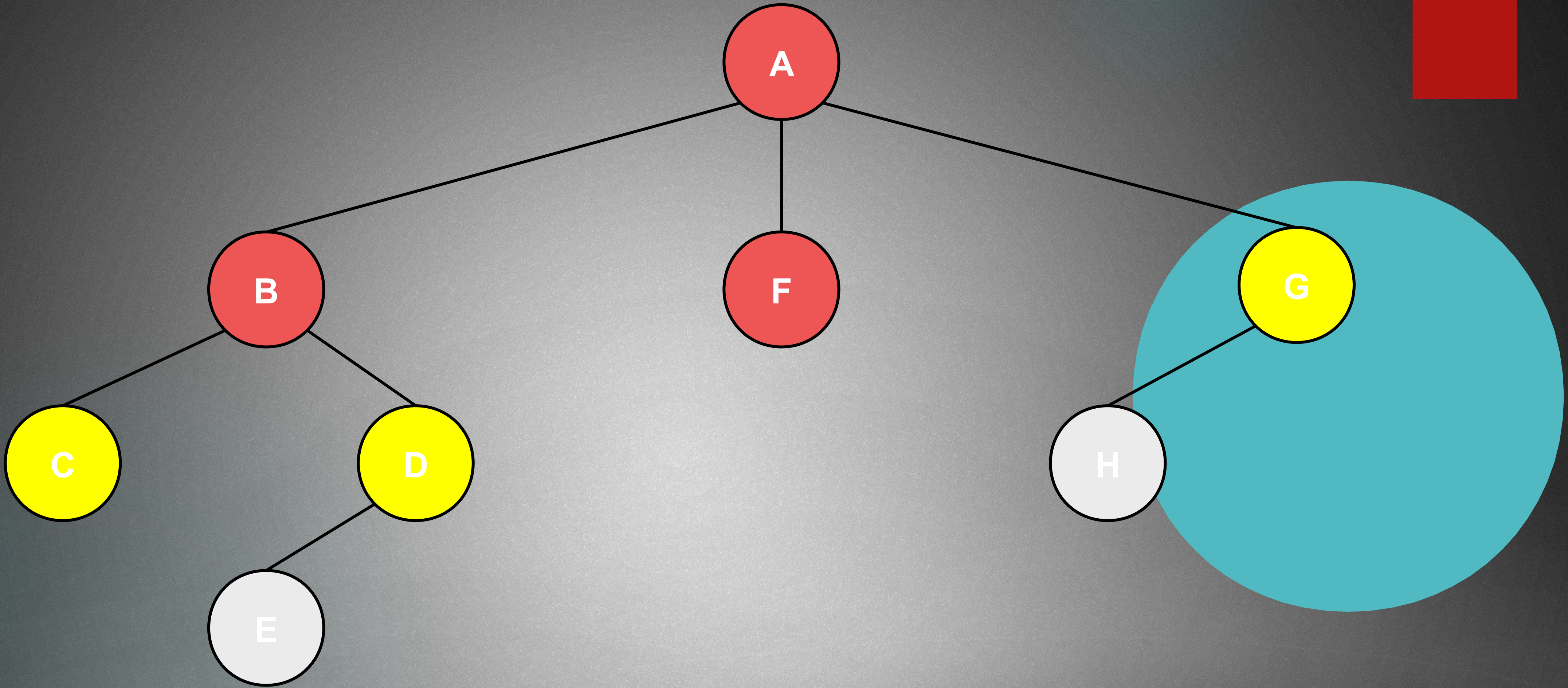
Queue: { D C G F }   dequeue the next node → it is G and visit its children + put them into the queue if necessary



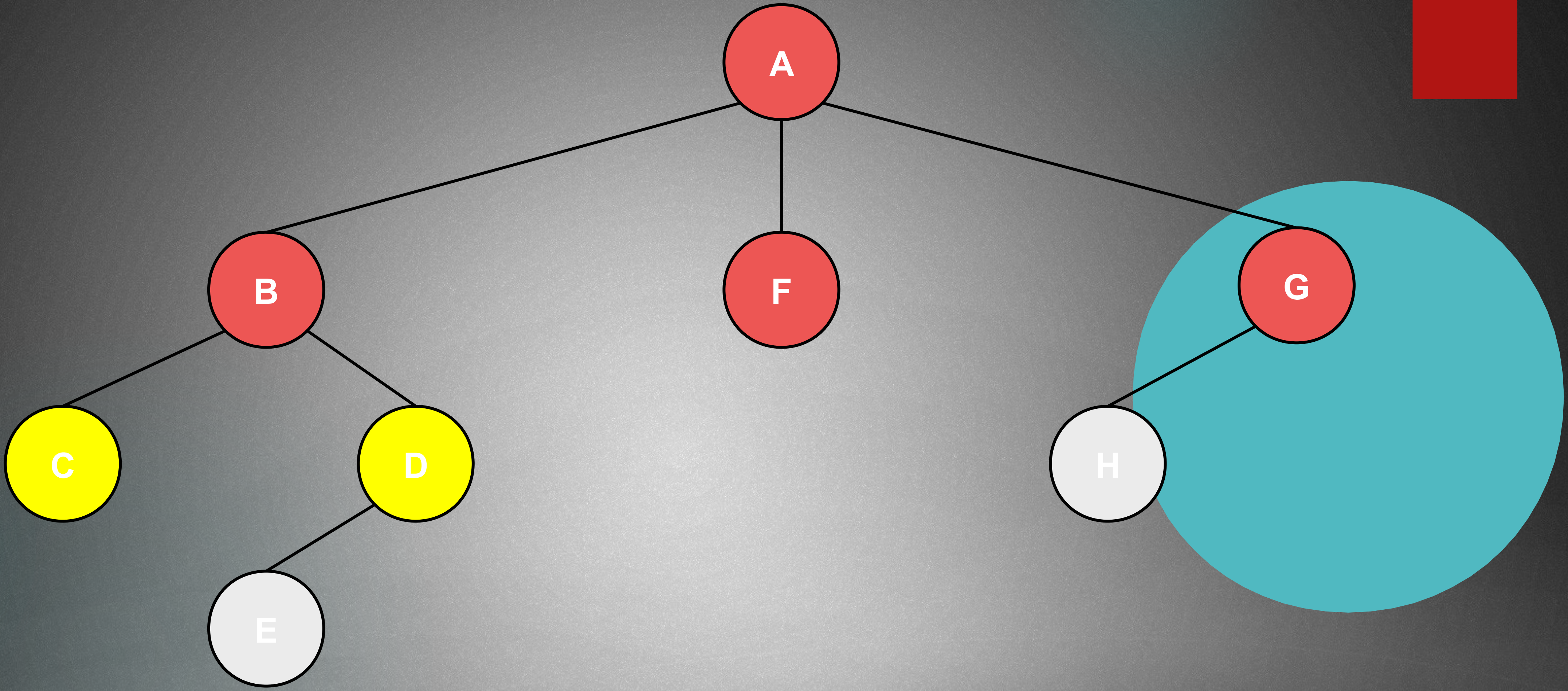
Queue: { D C G **F** }



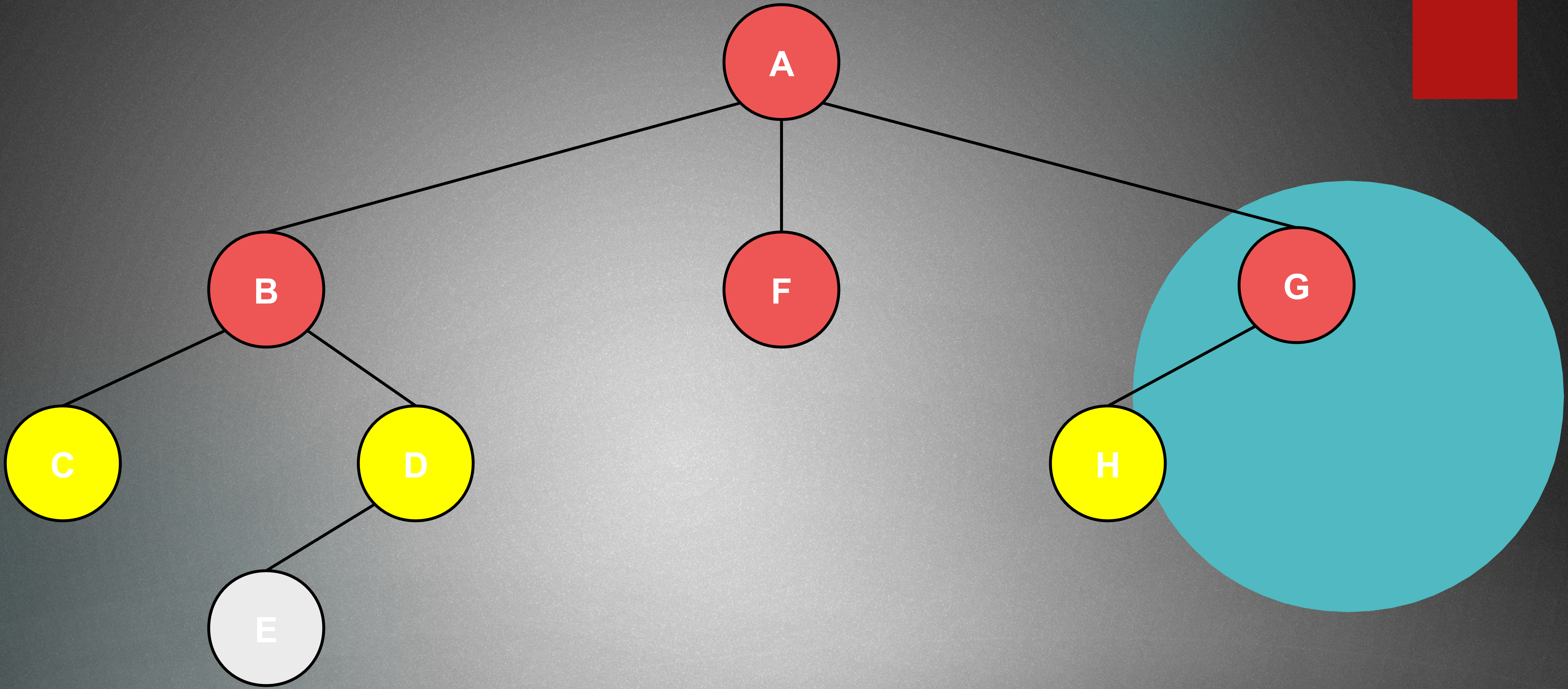
Queue: { D C G }



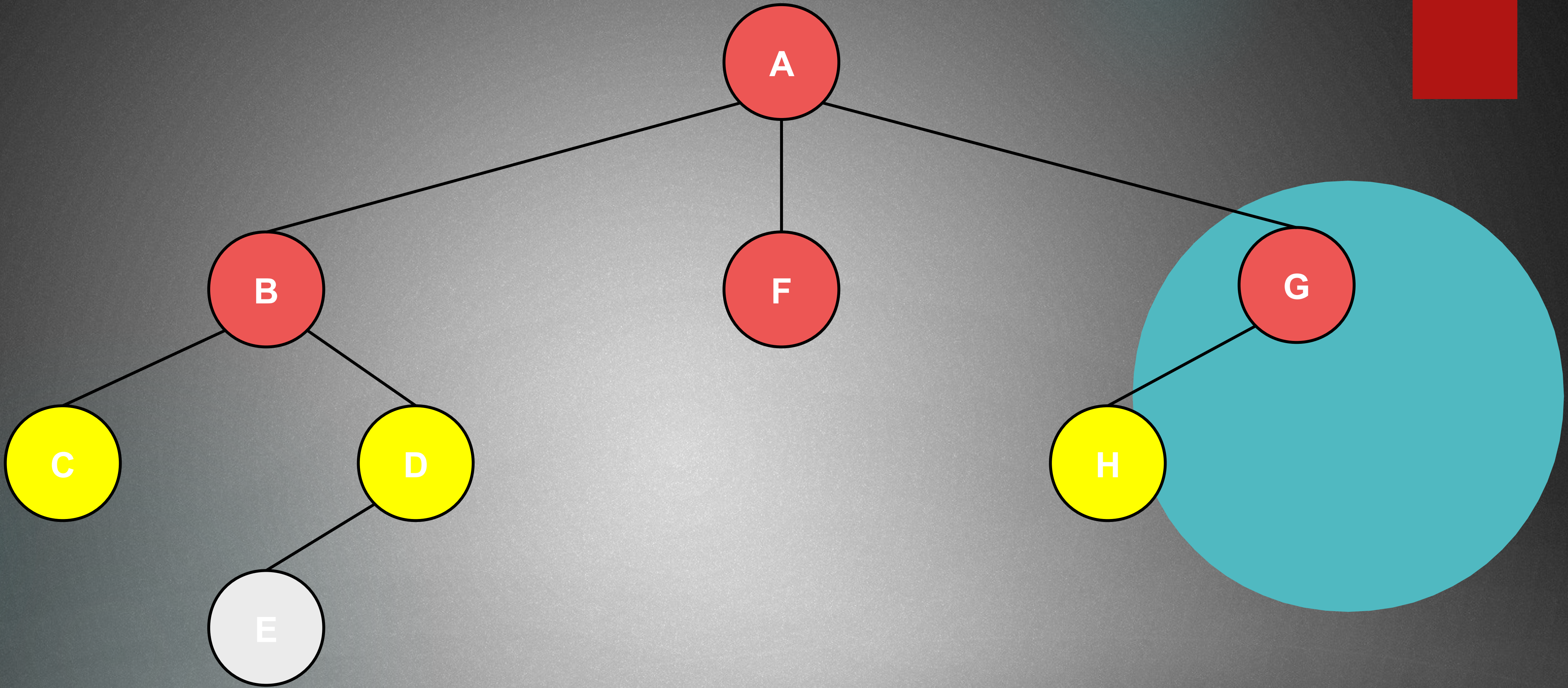
Queue: { D C **G** }



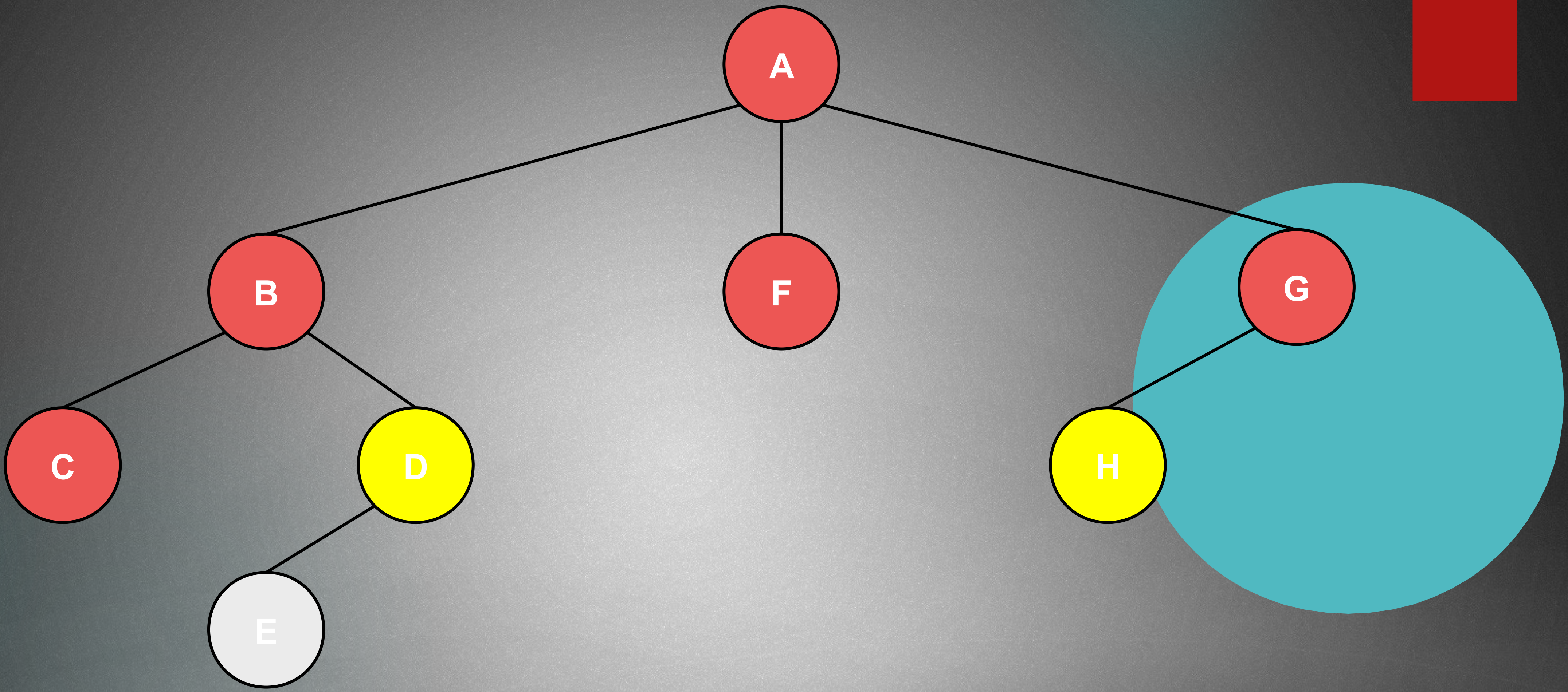
Queue: { D C }



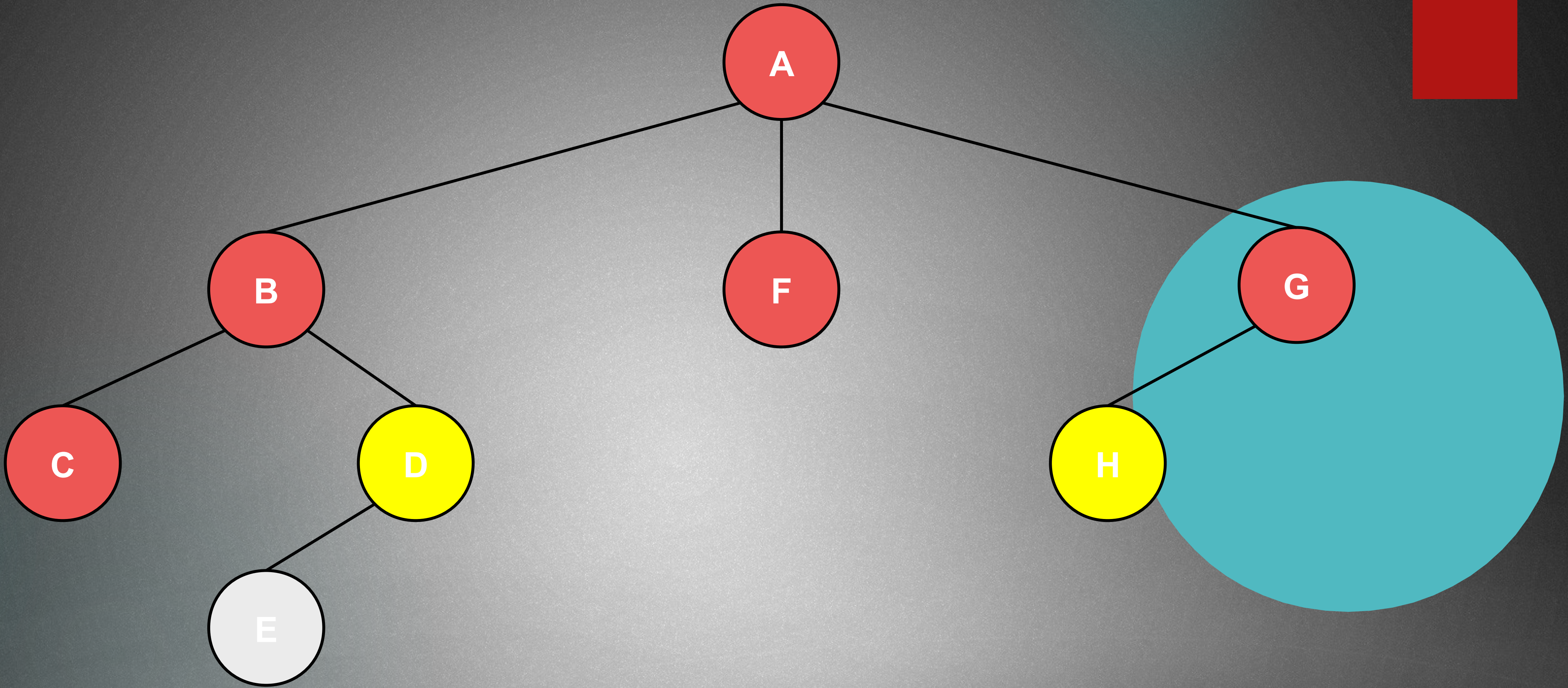
Queue: { H D C }



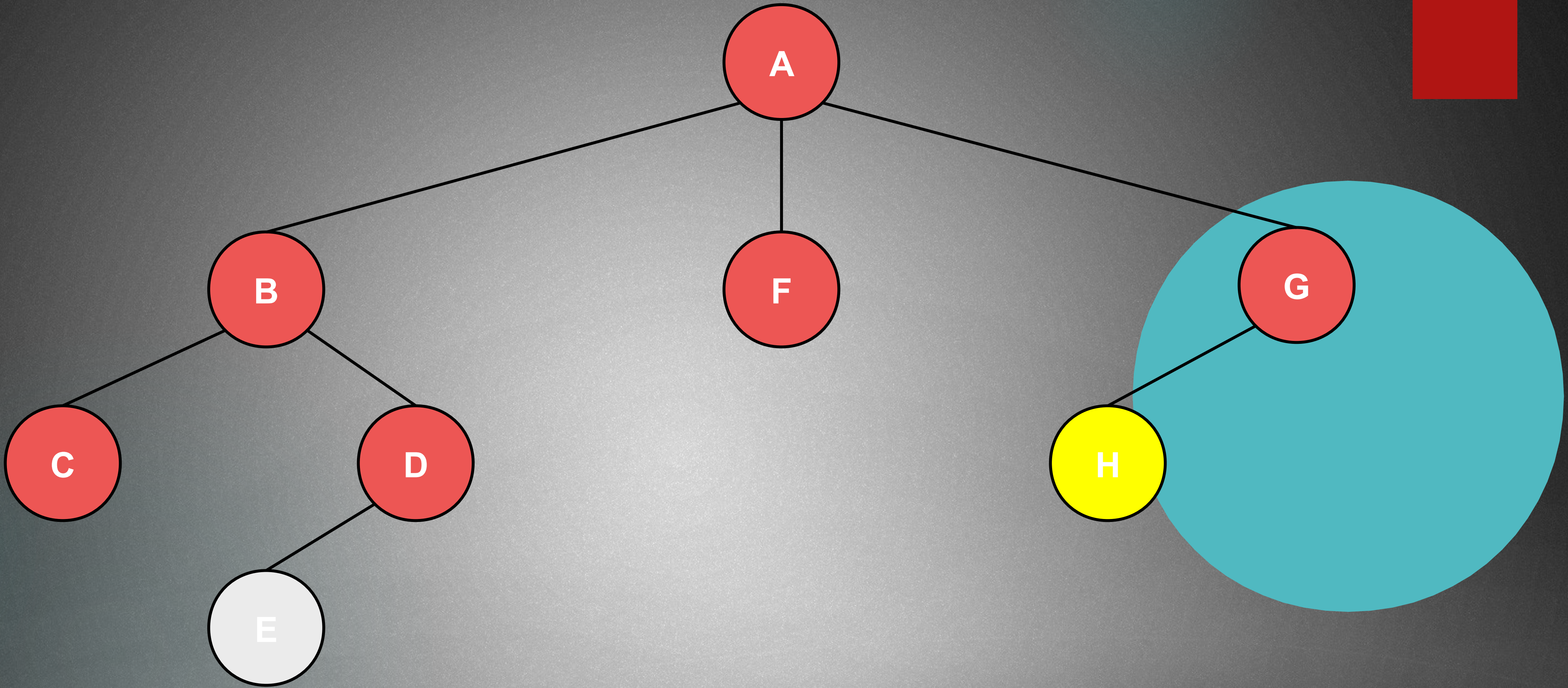
Queue: { H D **C** }



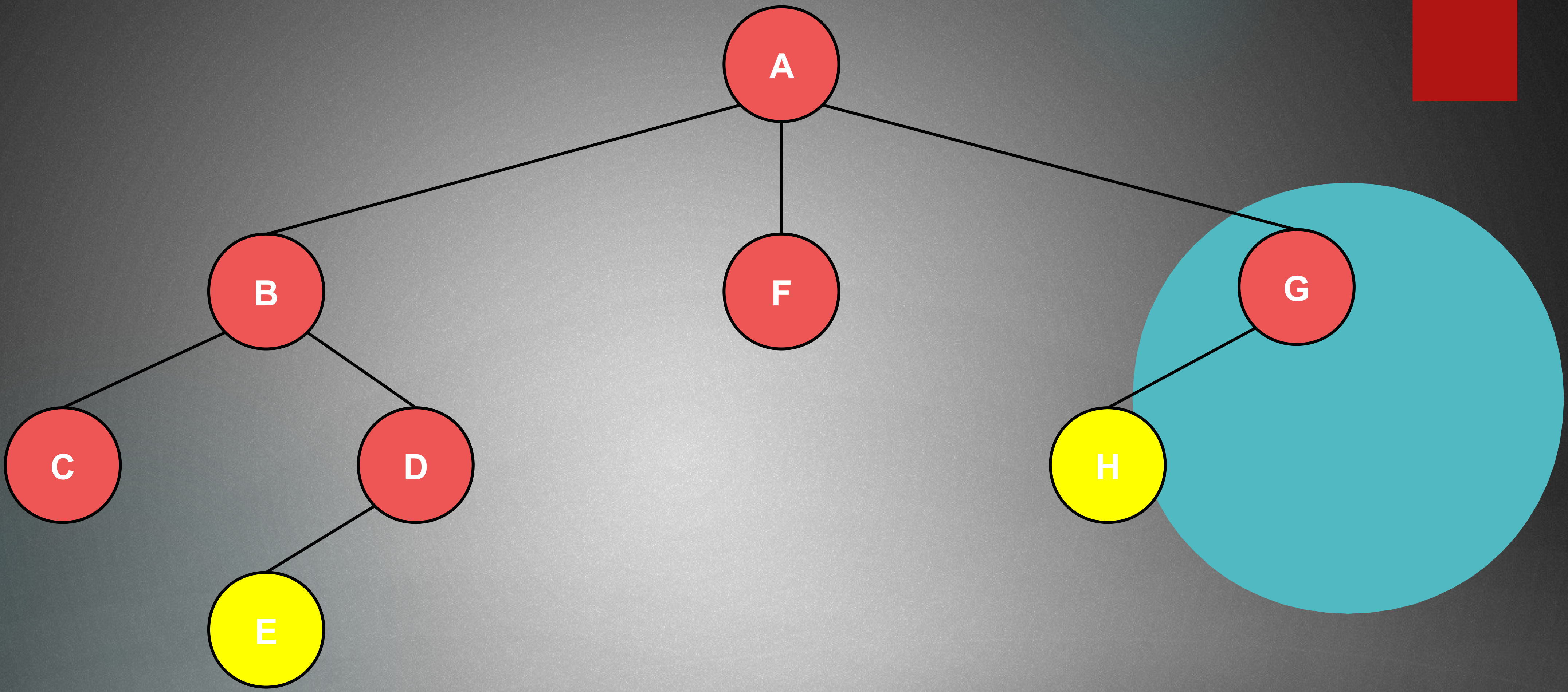
Queue: { H D }



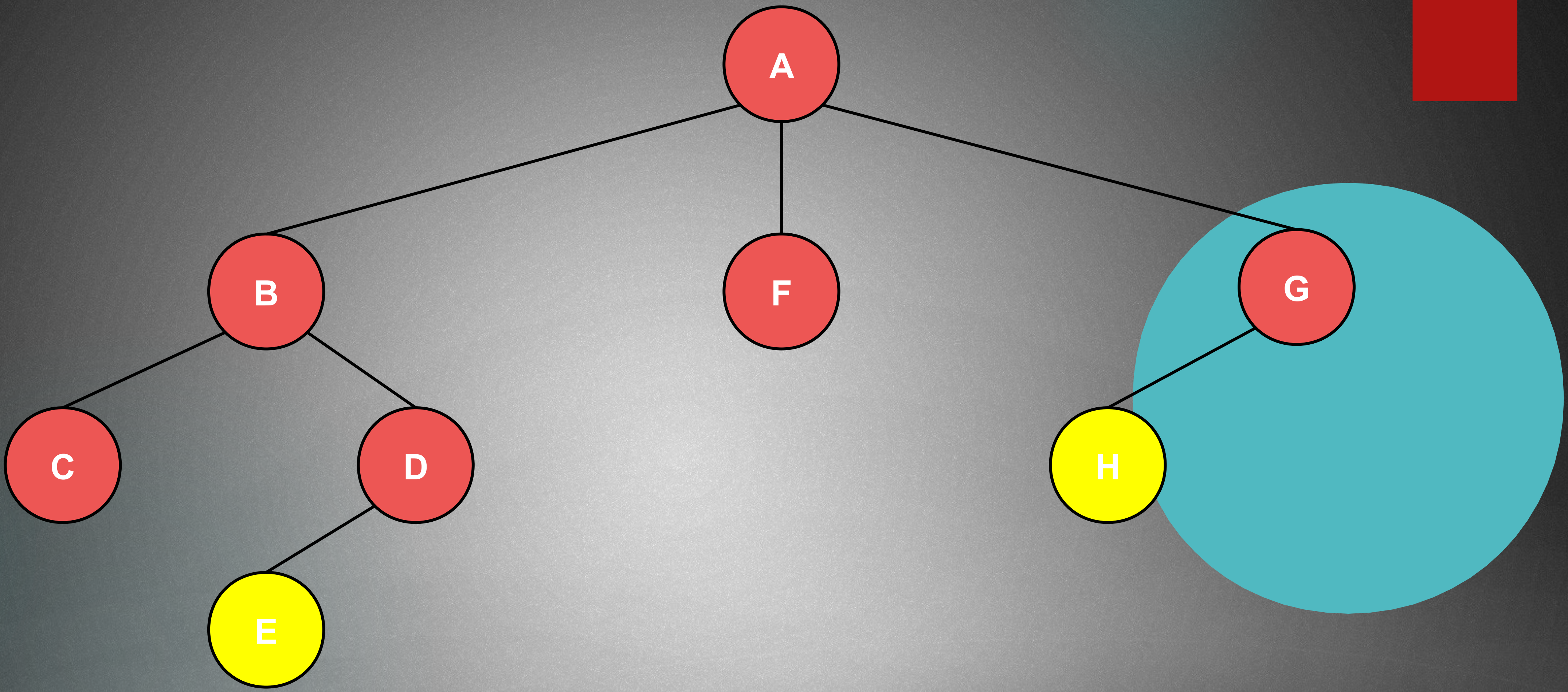
Queue: { H **D** }



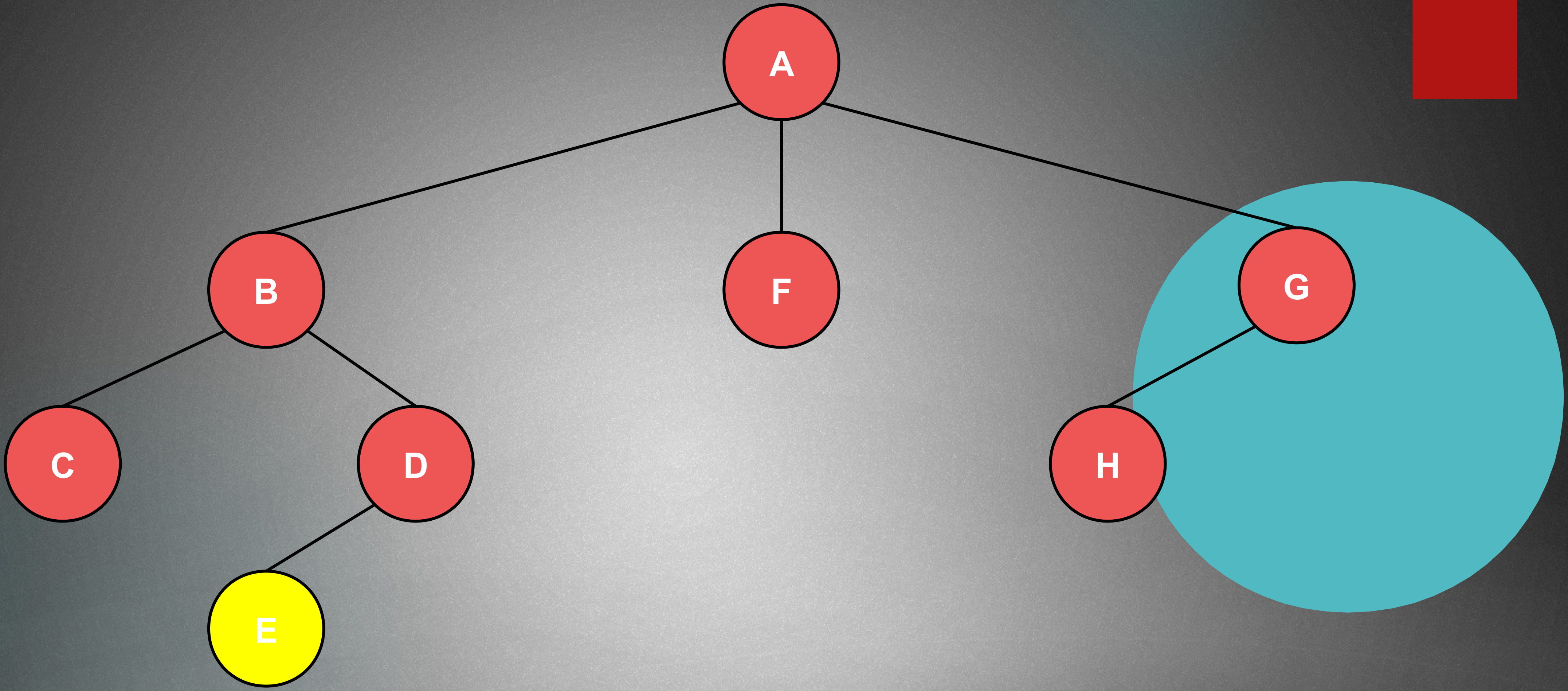
Queue: { H }



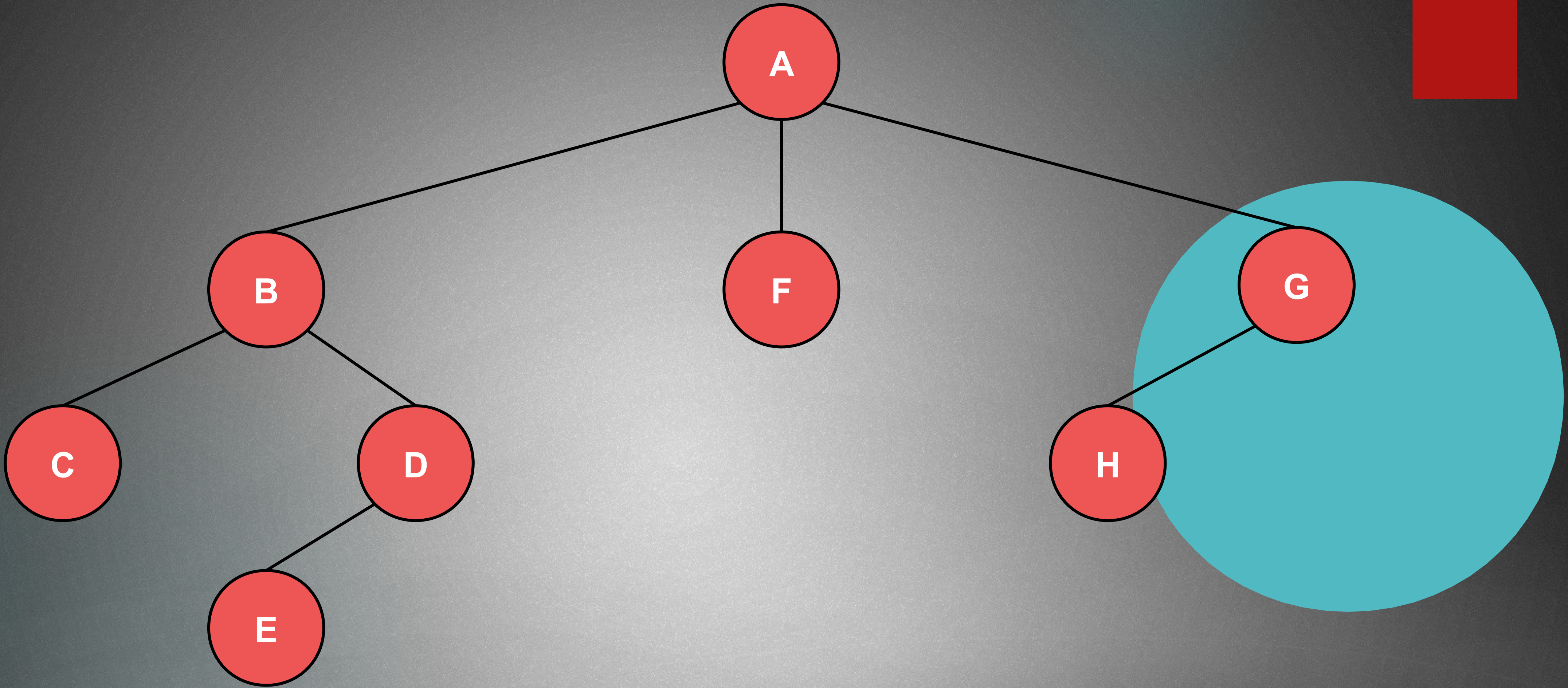
Queue: { E H }



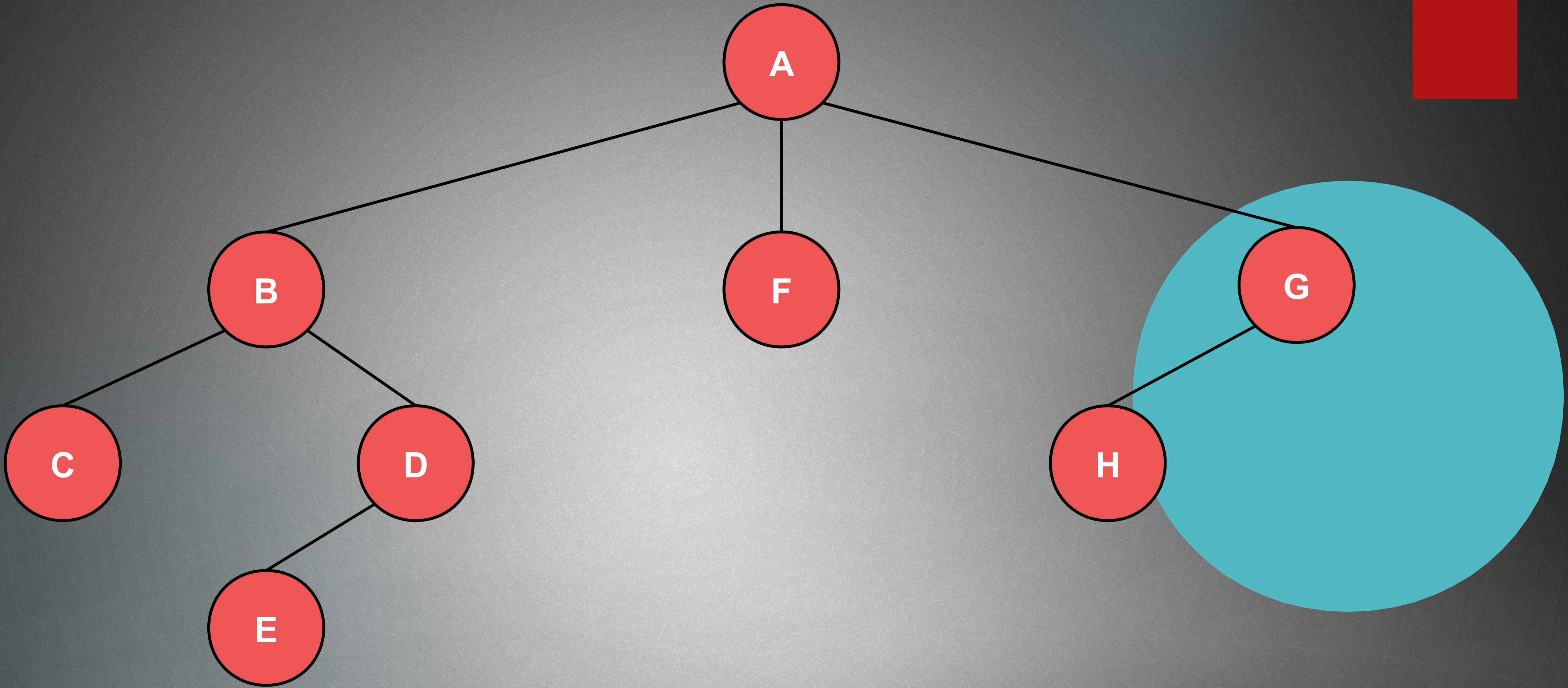
Queue: { E **H** }



Queue: { E }

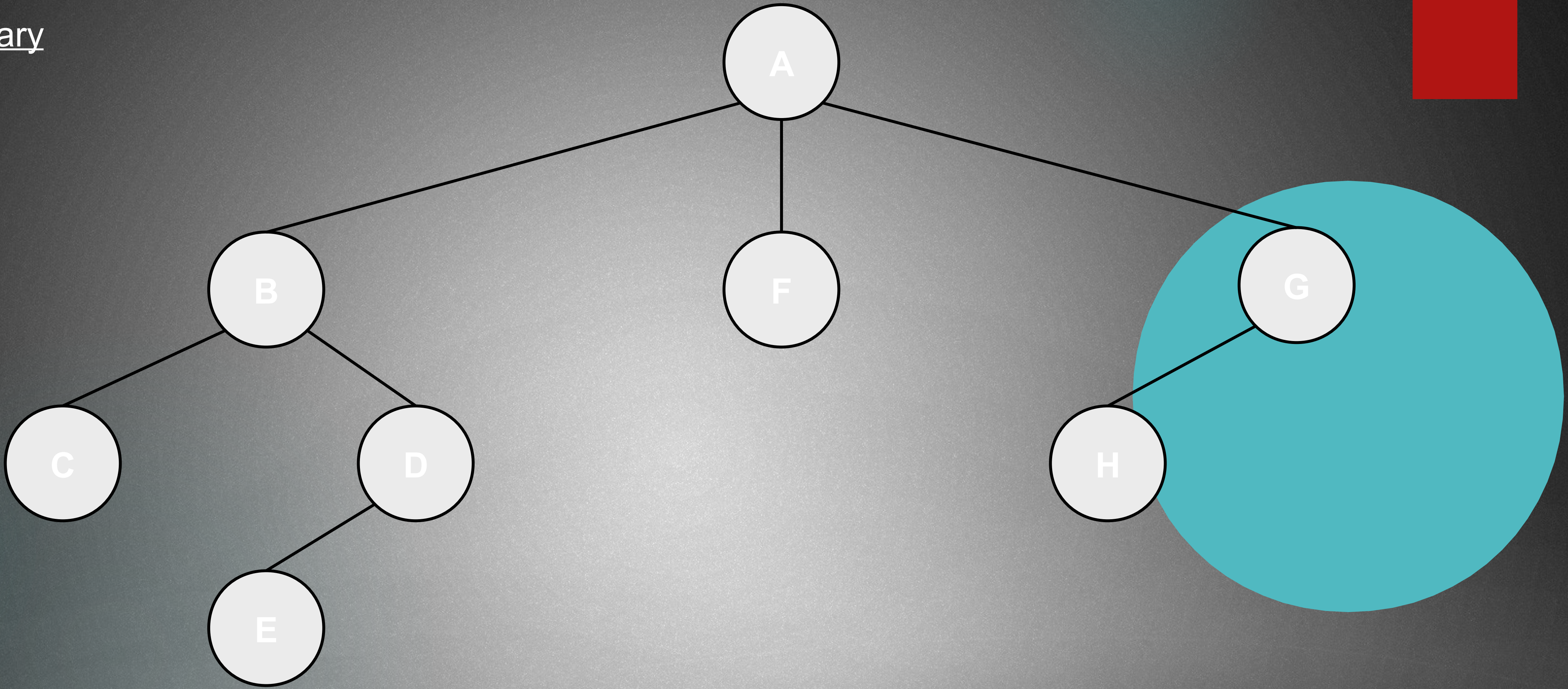


Queue: { **E** }

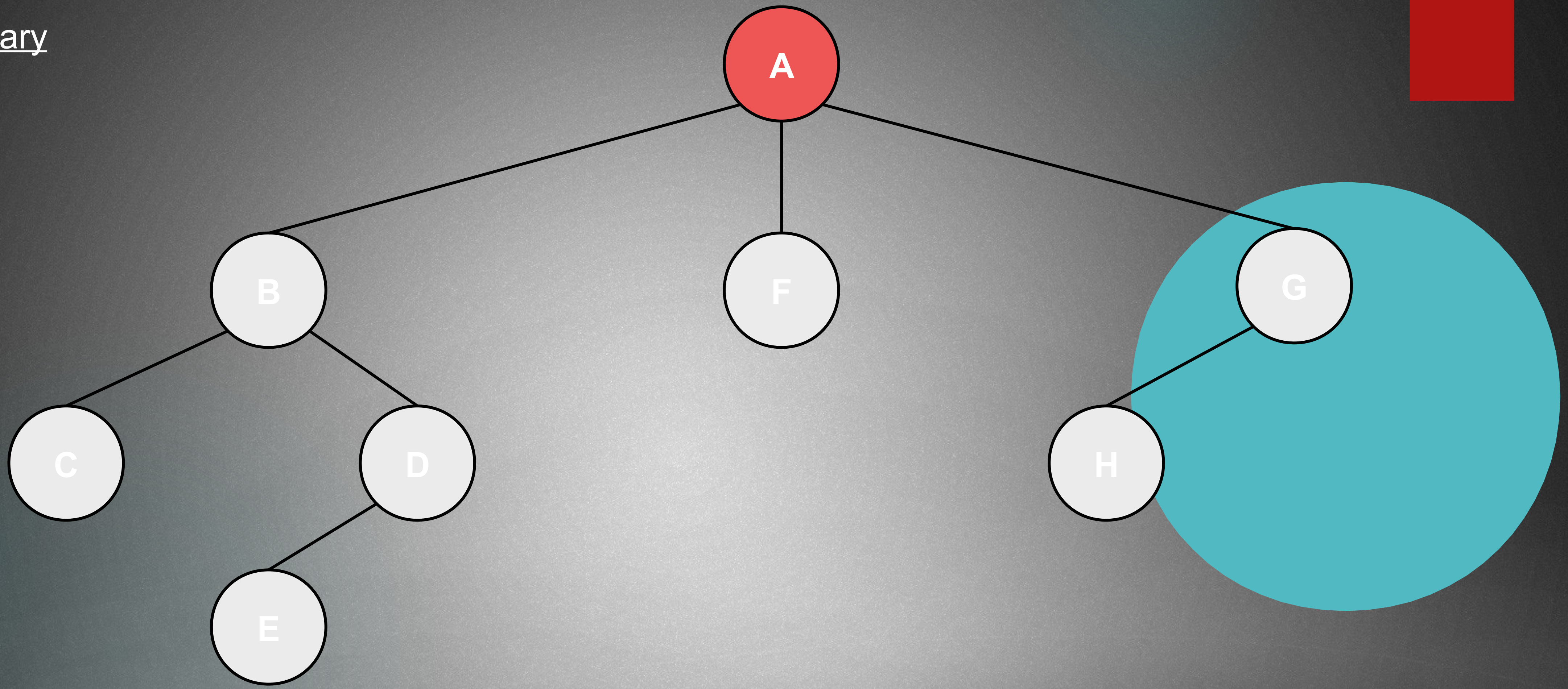


Queue: { } the queue is empty → **FINISHED** !!!

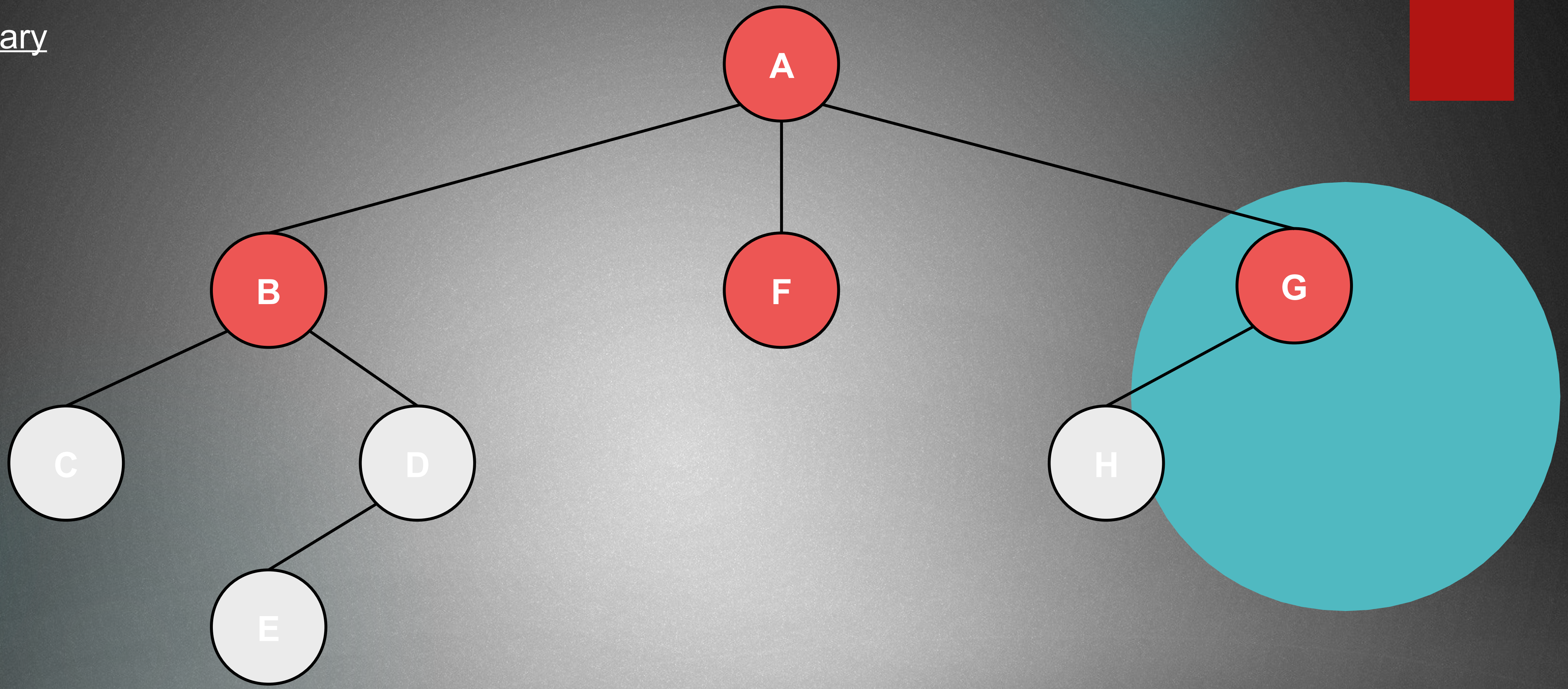
Summary



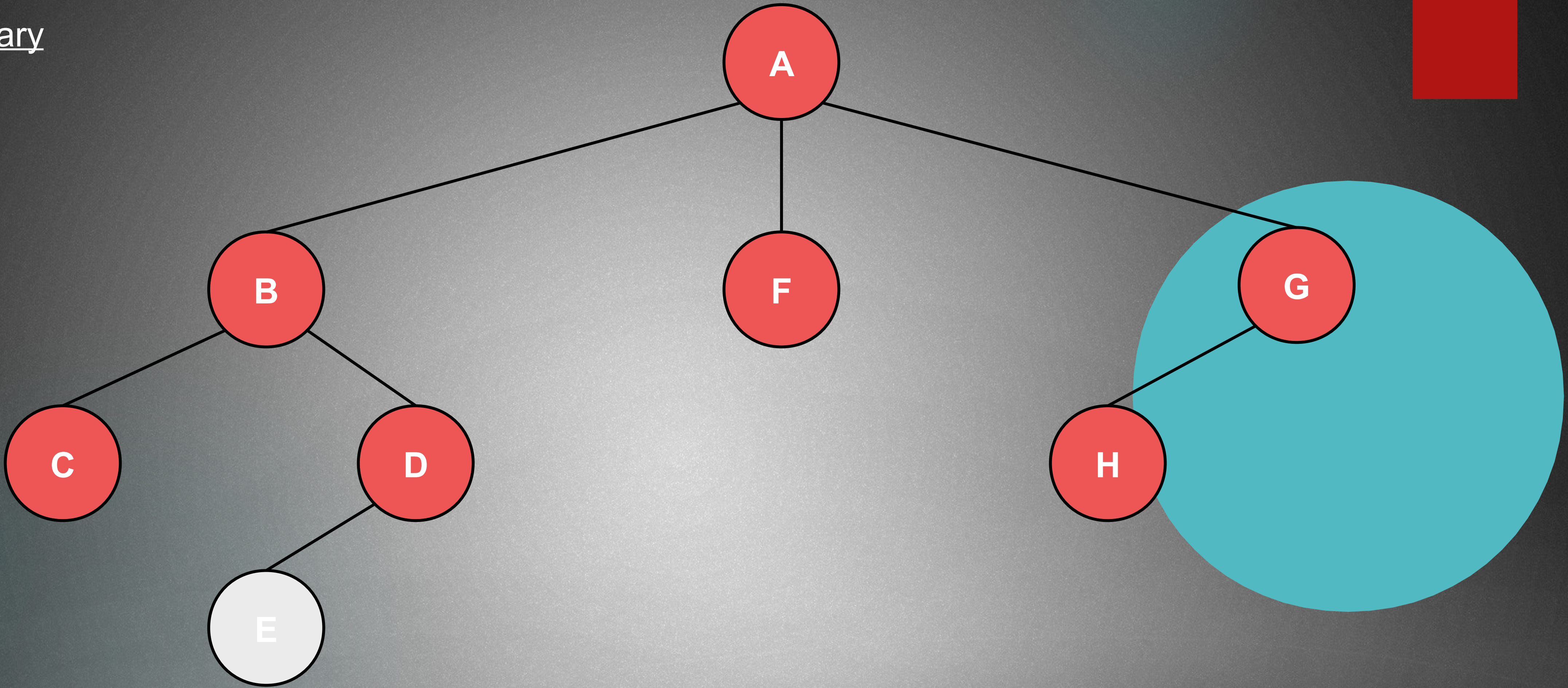
Summary



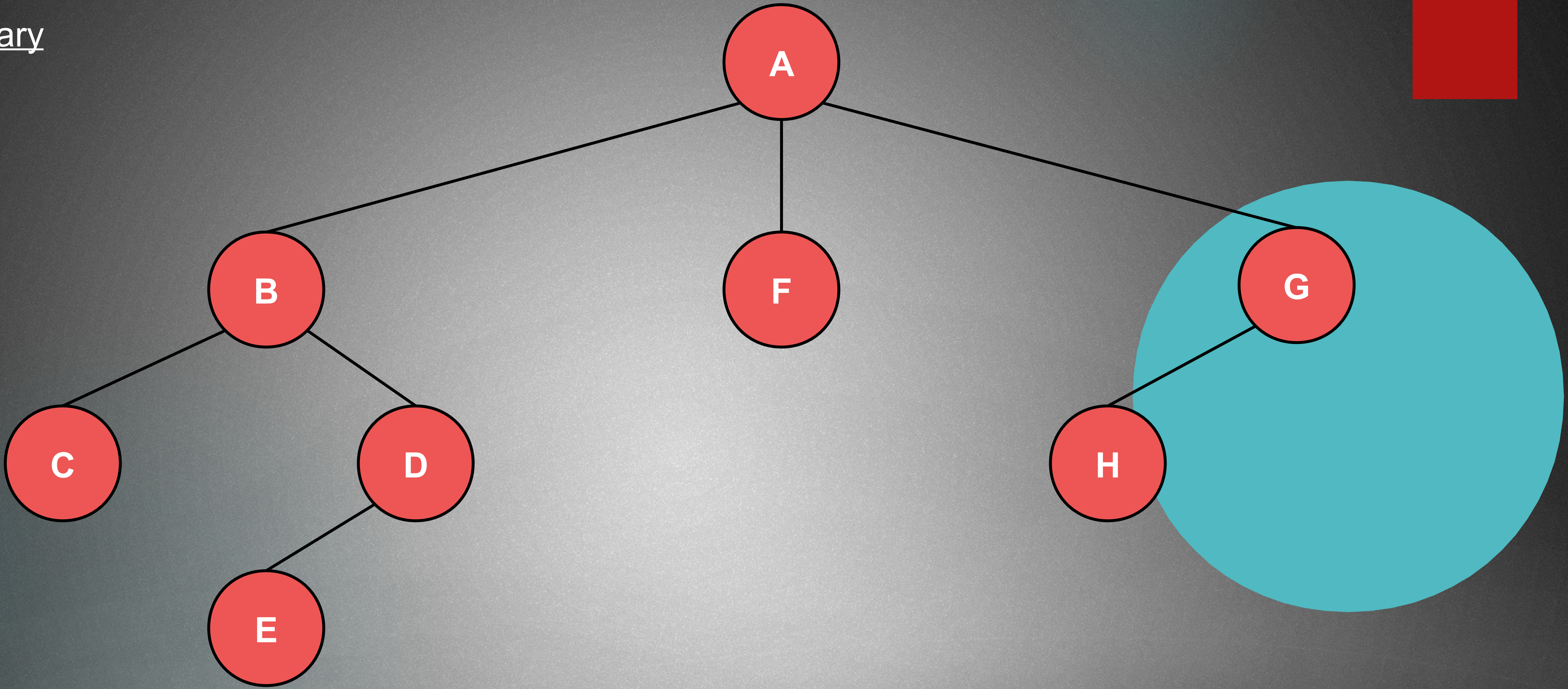
Summary



Summary



Summary



Applications

- In artificial intelligence / machine learning it can prove to be very important: robots can discover the surrounding more easily with BFS than DFS
- It is also very important in maximum flow: Edmonds-Karp algorithm uses BFS for finding augmenting paths
- Cheyen's algorithm in garbage collection → it help to maintain active references on the heap memory
- It uses BFS to detect all the references on the heap
- Serialization / deserialization of a tree like structure (for example when order does matter) → it allows the tree to be reconstructed in an efficient manner !!!

DEPTH FIRST SEARCH

dfs

Depth-first search

- Depth-first search is a widely used graph traversal algorithm besides breadth-first search
- It was investigated as strategy for solving mazes by Trémaux in the 19th century
- It explores as far as possible along each branch before backtracking // BFS was a layer-by-layer algorithm
- Time complexity of traversing a graph with DFS: **O(V+E)**
- Memory complexity: a bit better than that of BFS !!!

Depth-first search

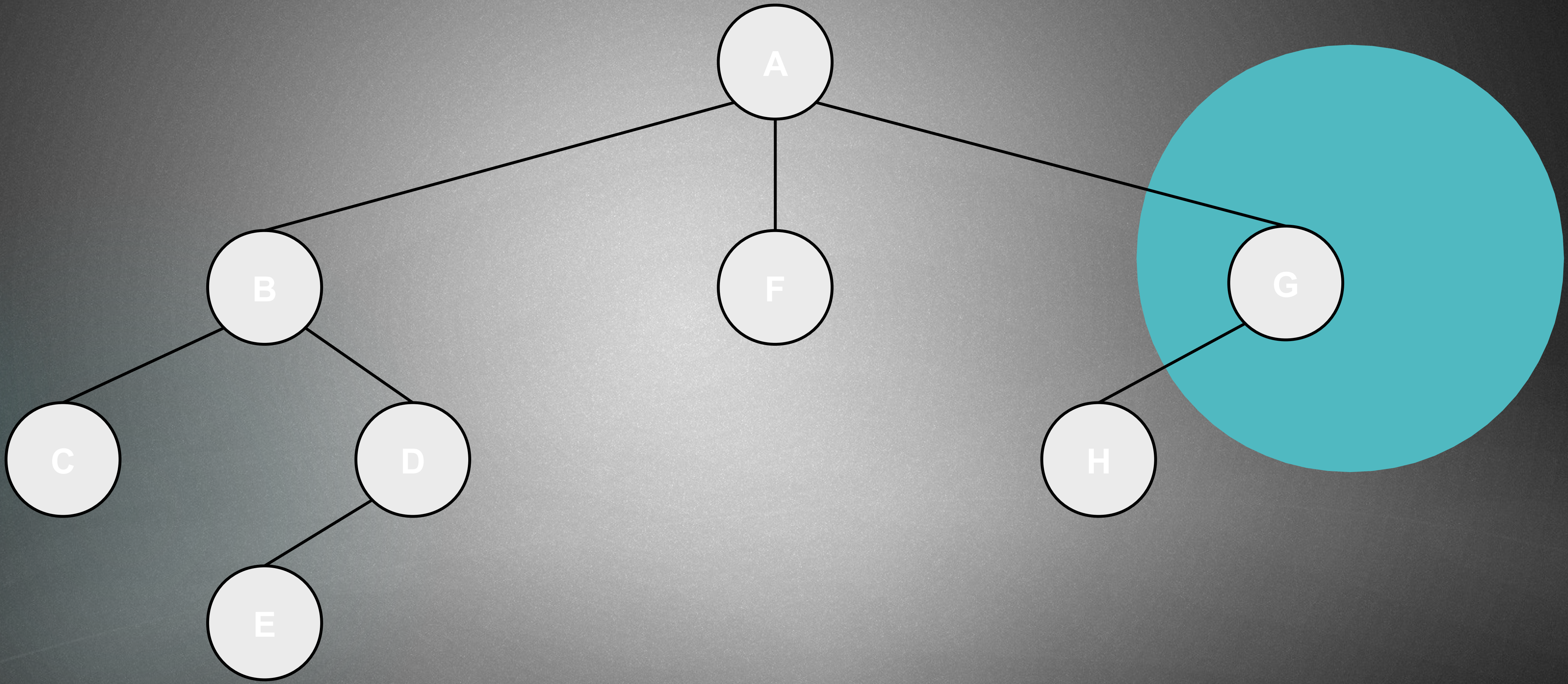
```
dfs(vertex)

    vertex set visited true
    print vertex

    for v in vertex neighbours
        if v is not visited
            dfs(v)
```

RECURSION

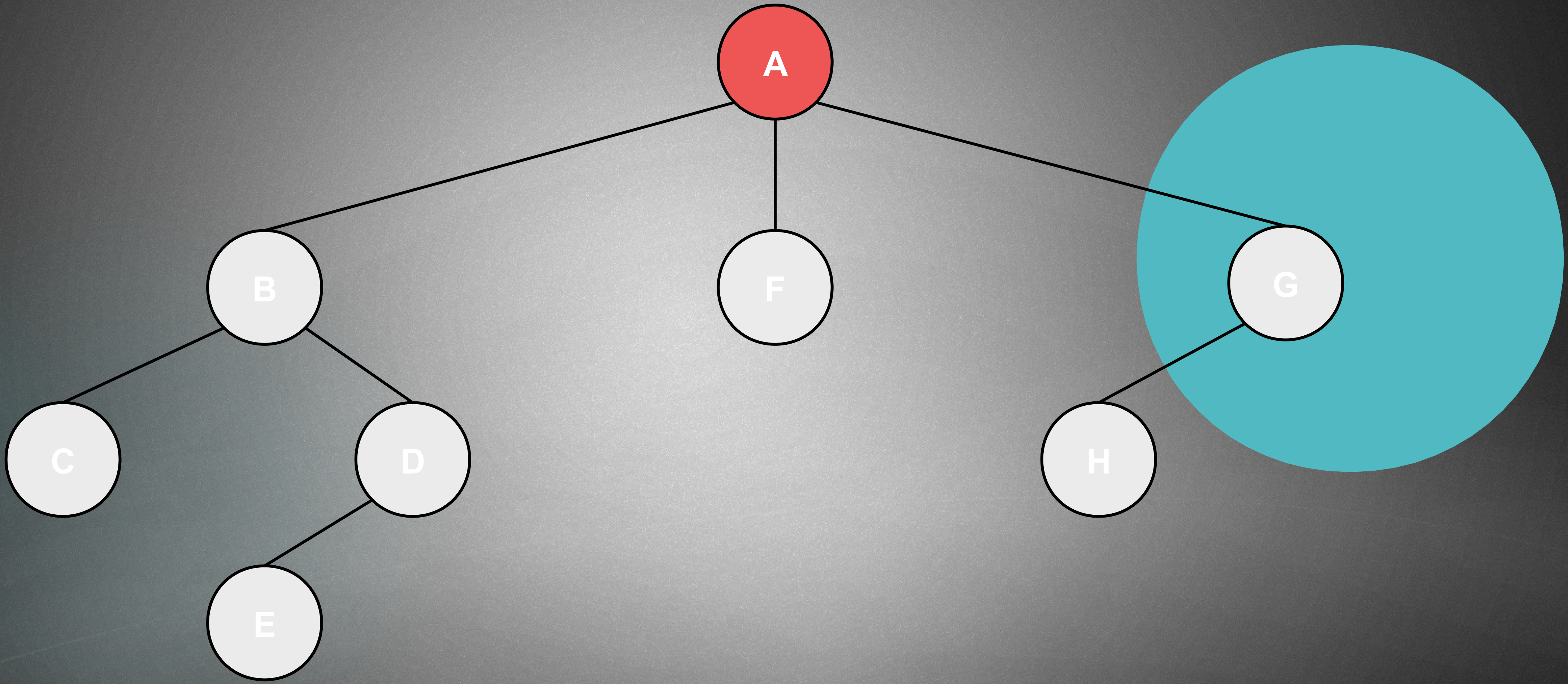
```
dfs(vertex)

    Stack stack
    vertex set visited true
    stack.push(vertex)

    while stack not empty
        actual = stack.pop()

        for v in actual neighbours
            if v is not visited
                v set visited true
                stack.push(v)
```

ITERATION

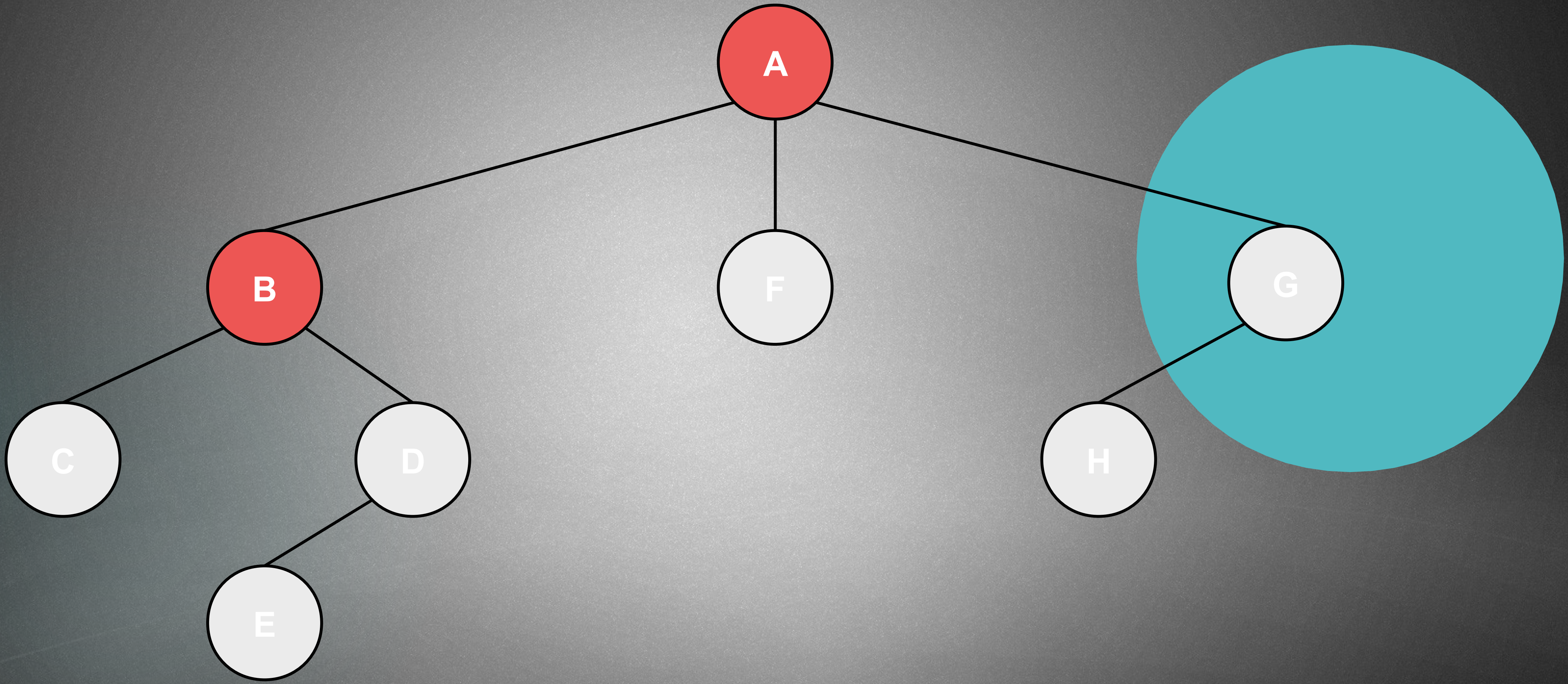
Recursively check every unmarked vertices!



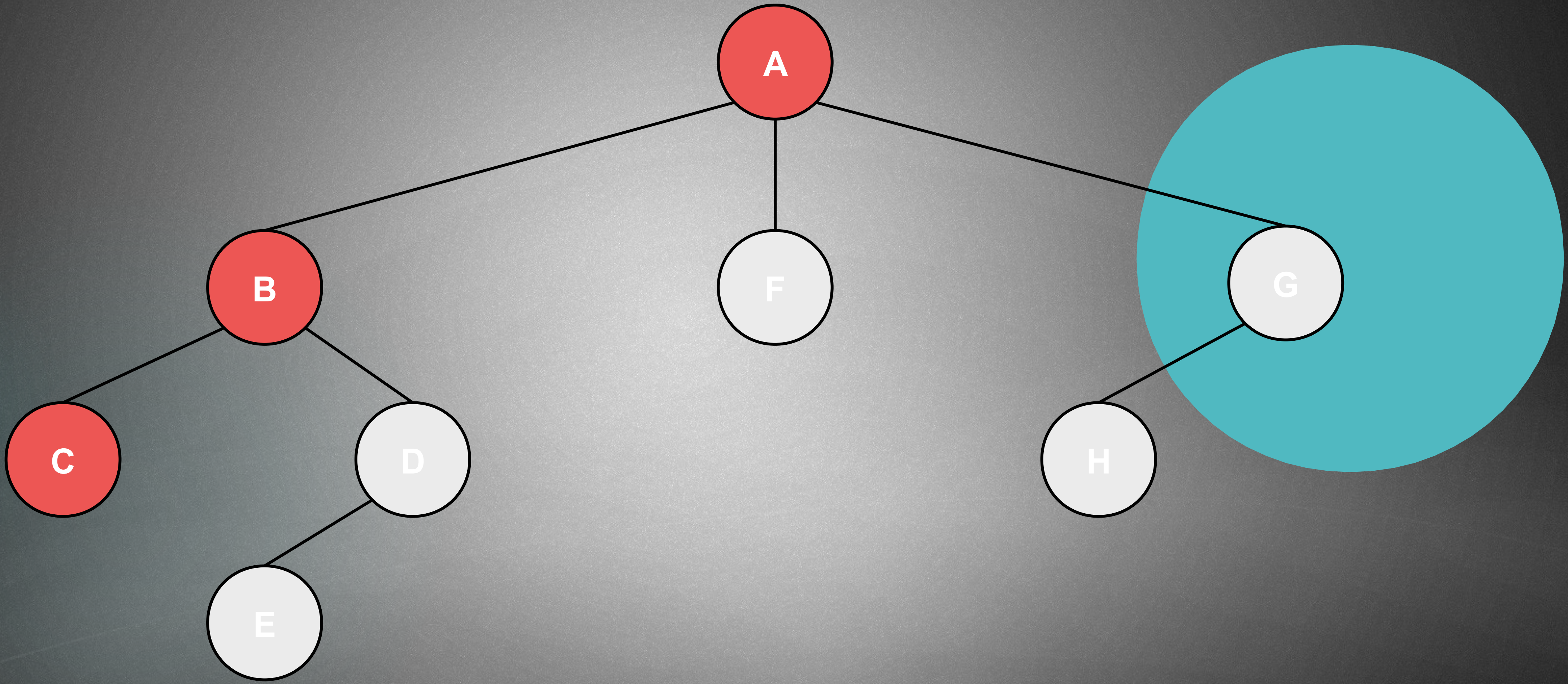
Recursively check every unmarked vertices!



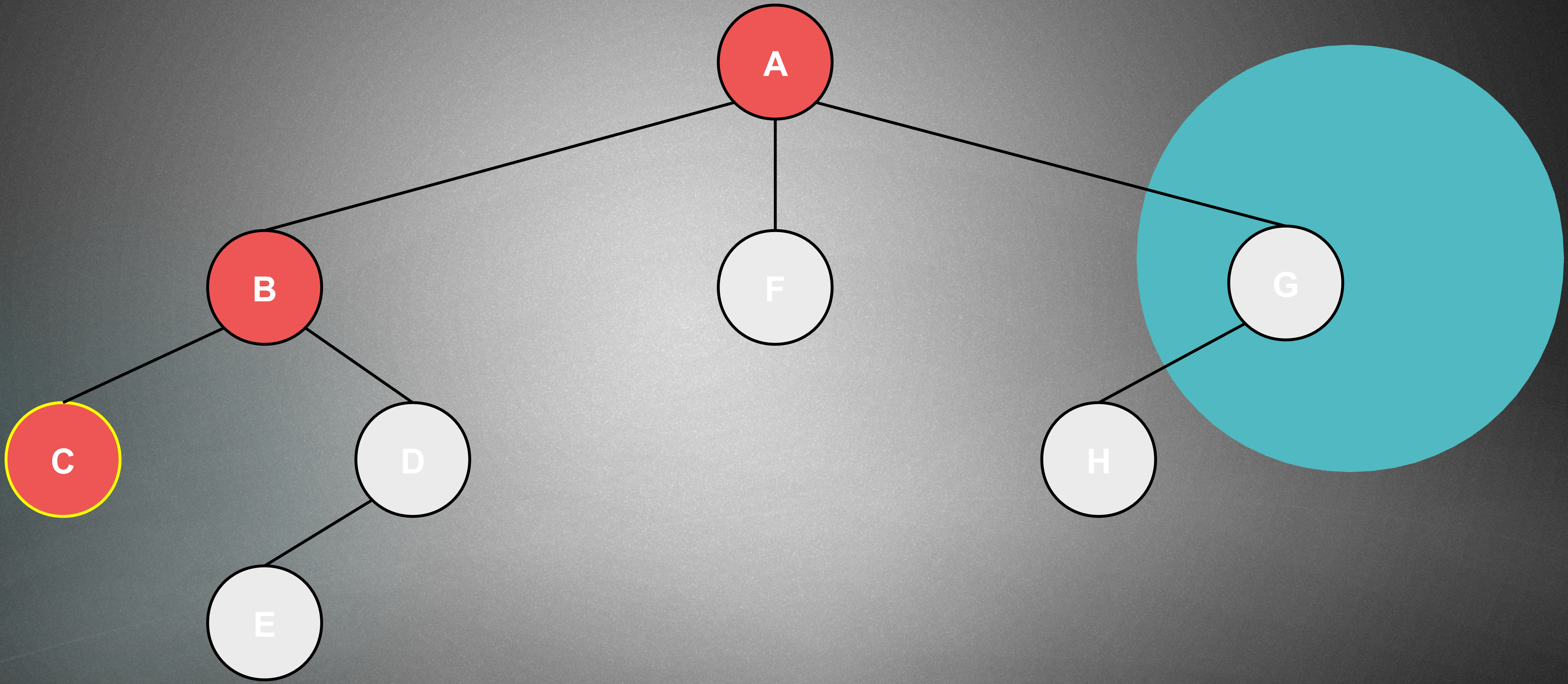
Recursively check every unmarked vertices!



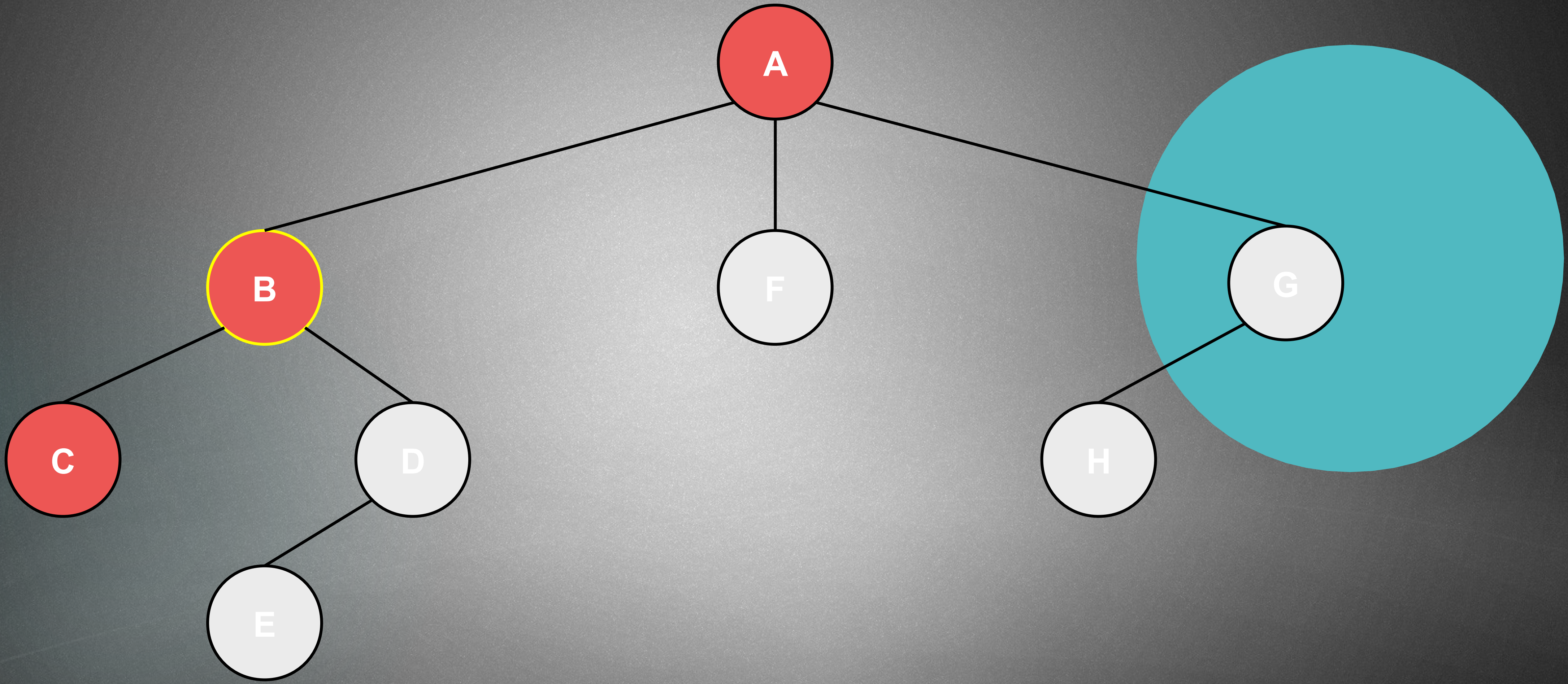
Recursively check every unmarked vertices!



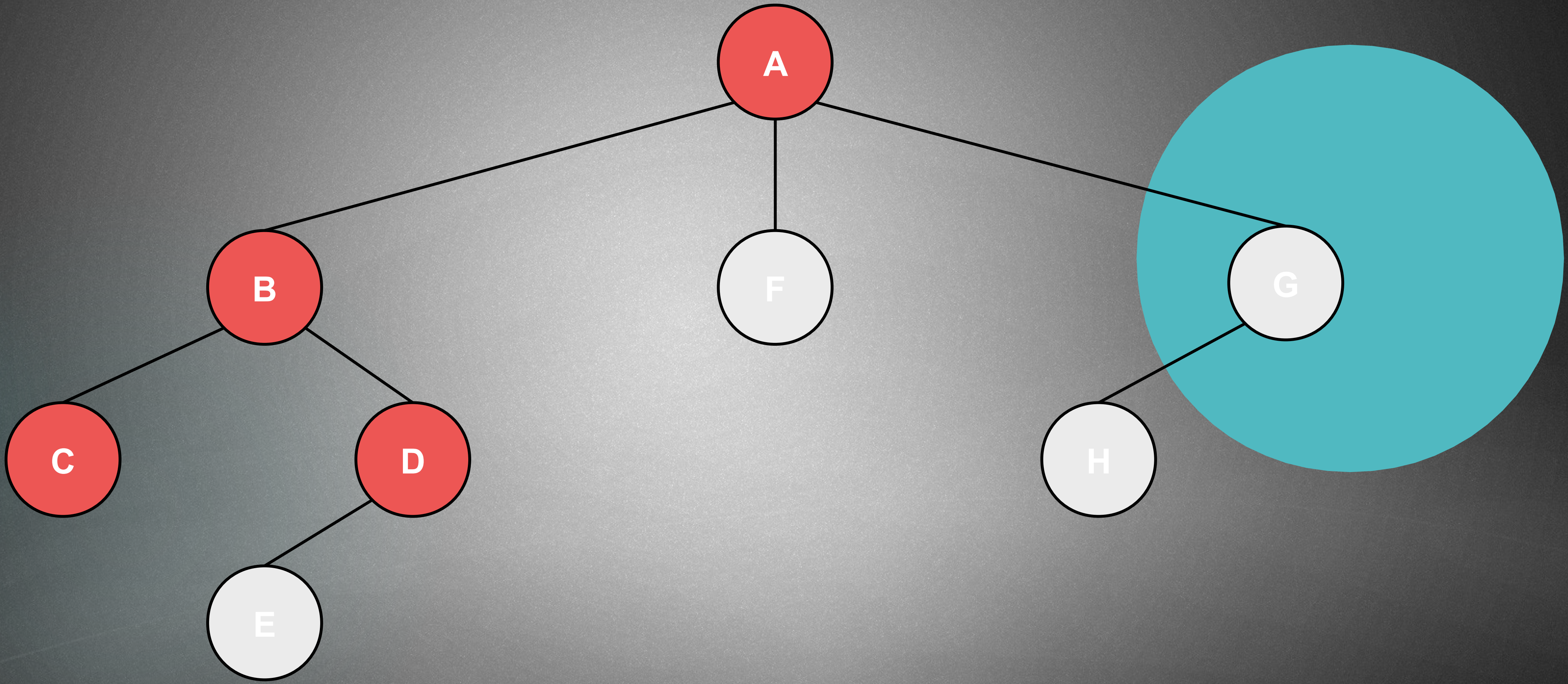
Recursively check every unmarked vertices!



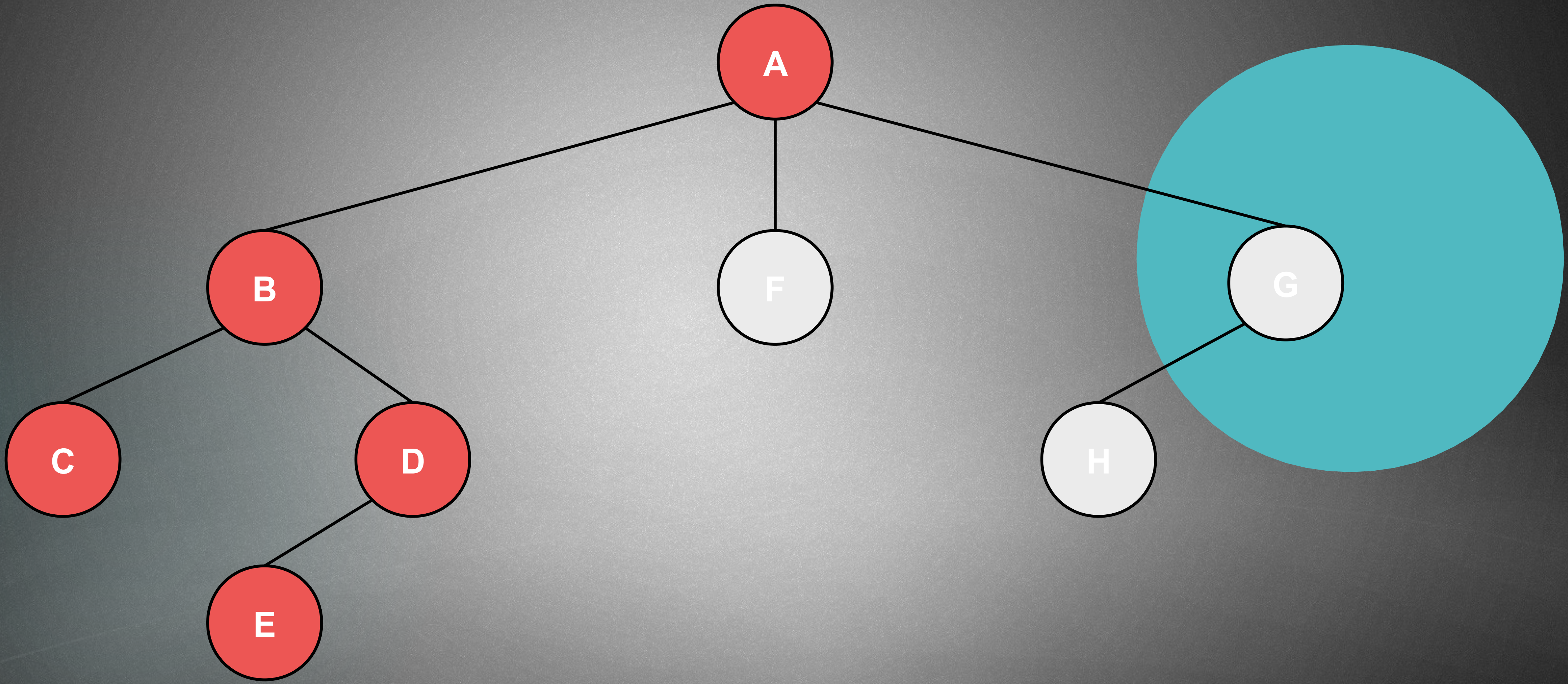
Recursively check every unmarked vertices!



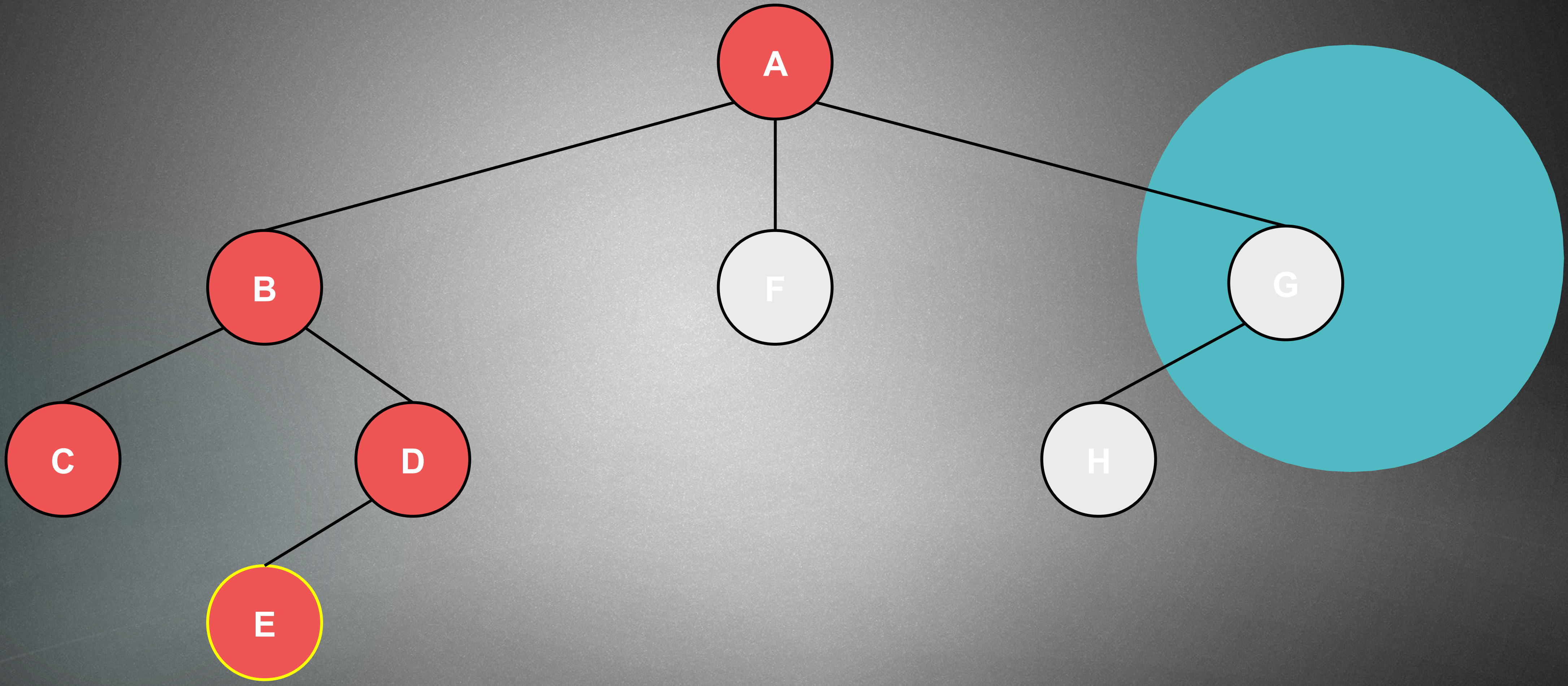
Recursively check every unmarked vertices!



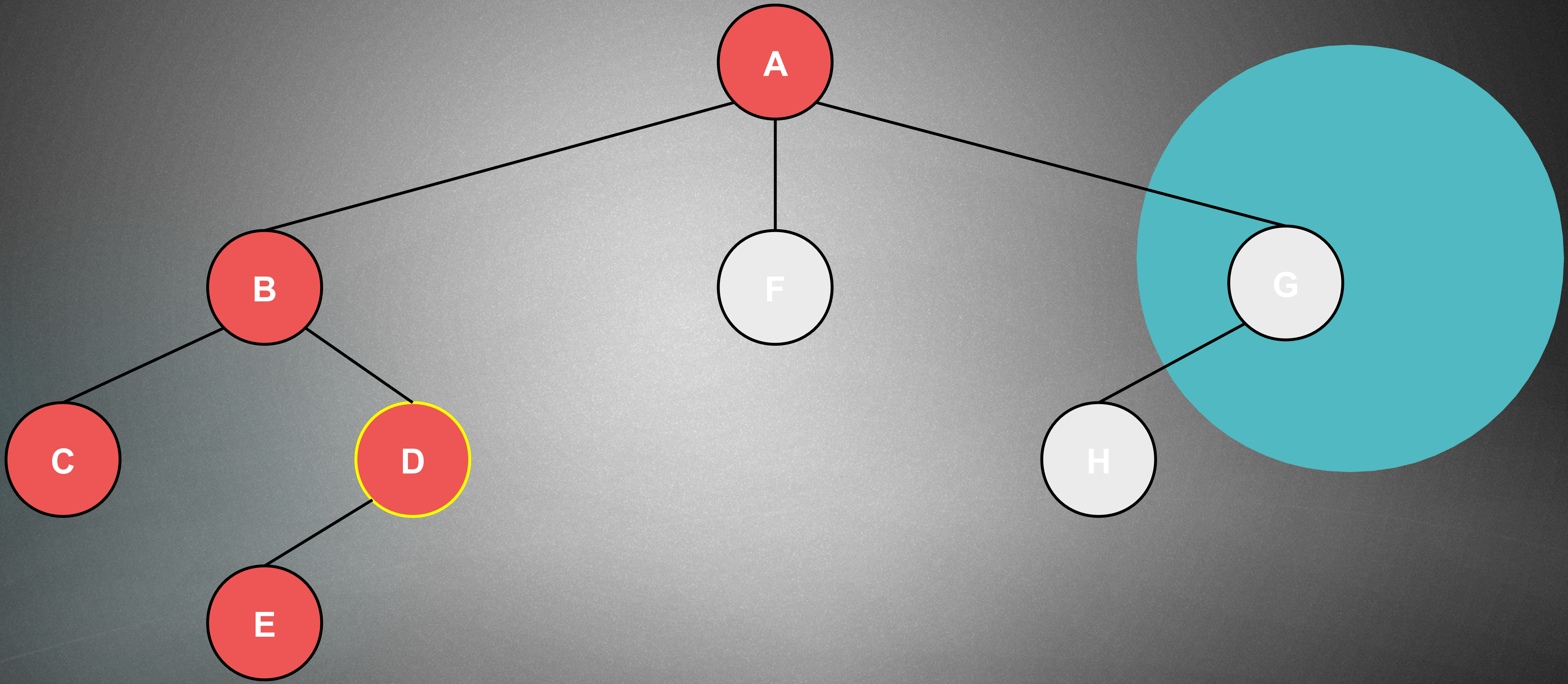
Recursively check every unmarked vertices!



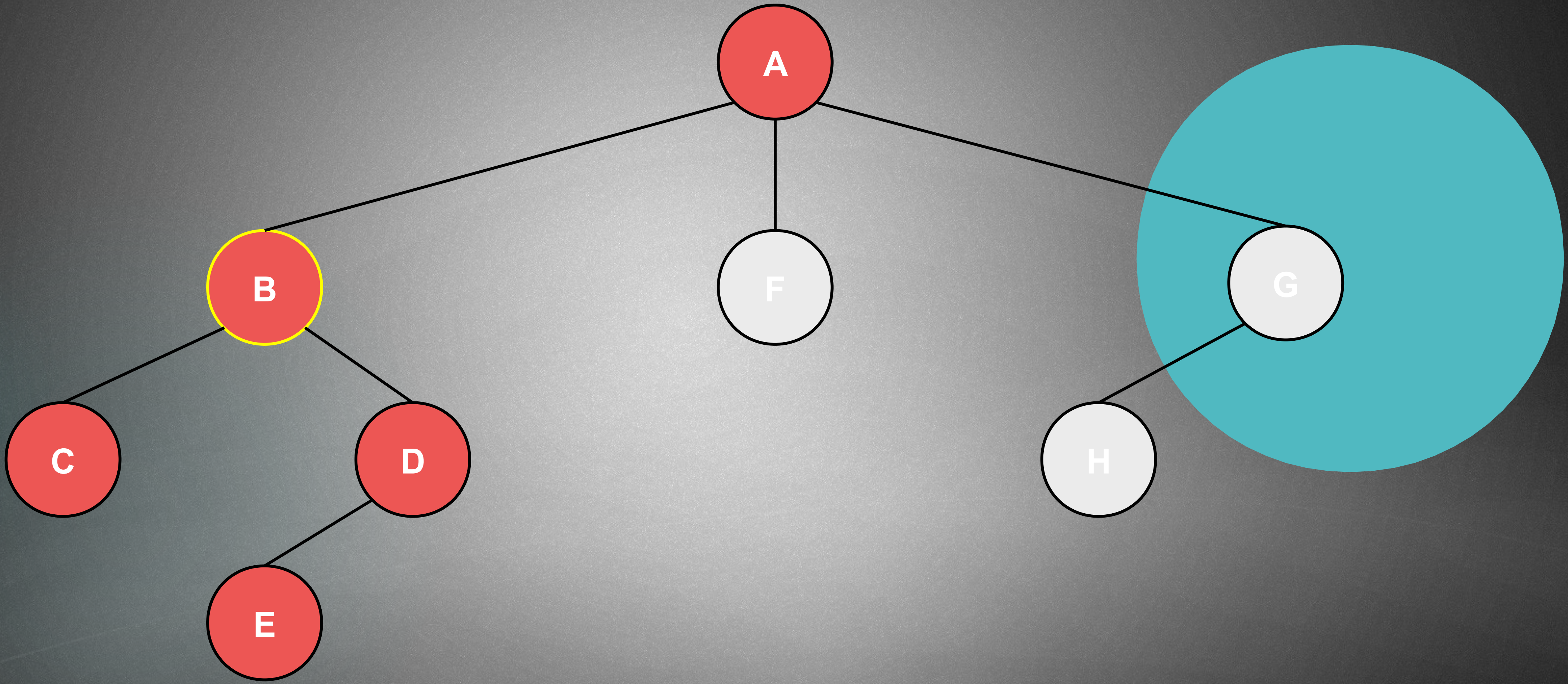
Recursively check every unmarked vertices!



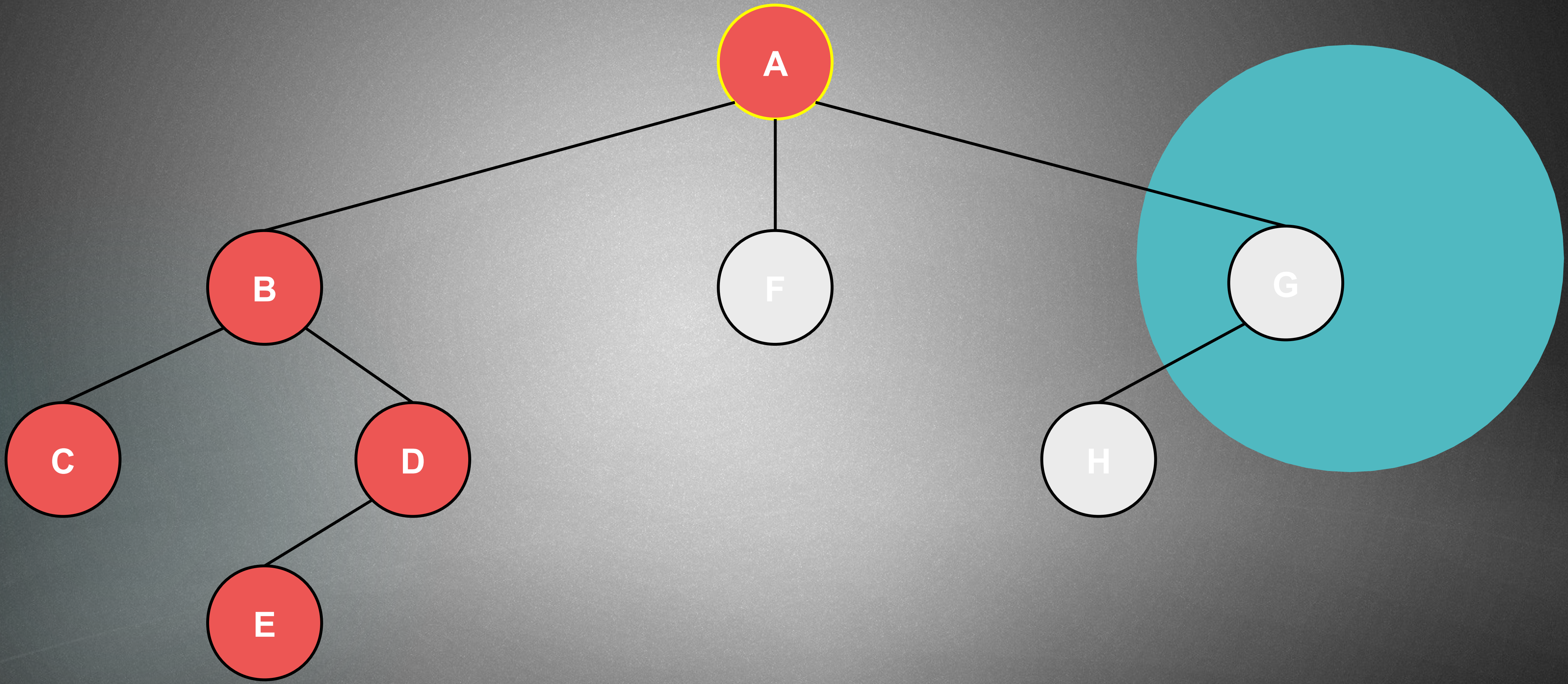
Recursively check every unmarked vertices!



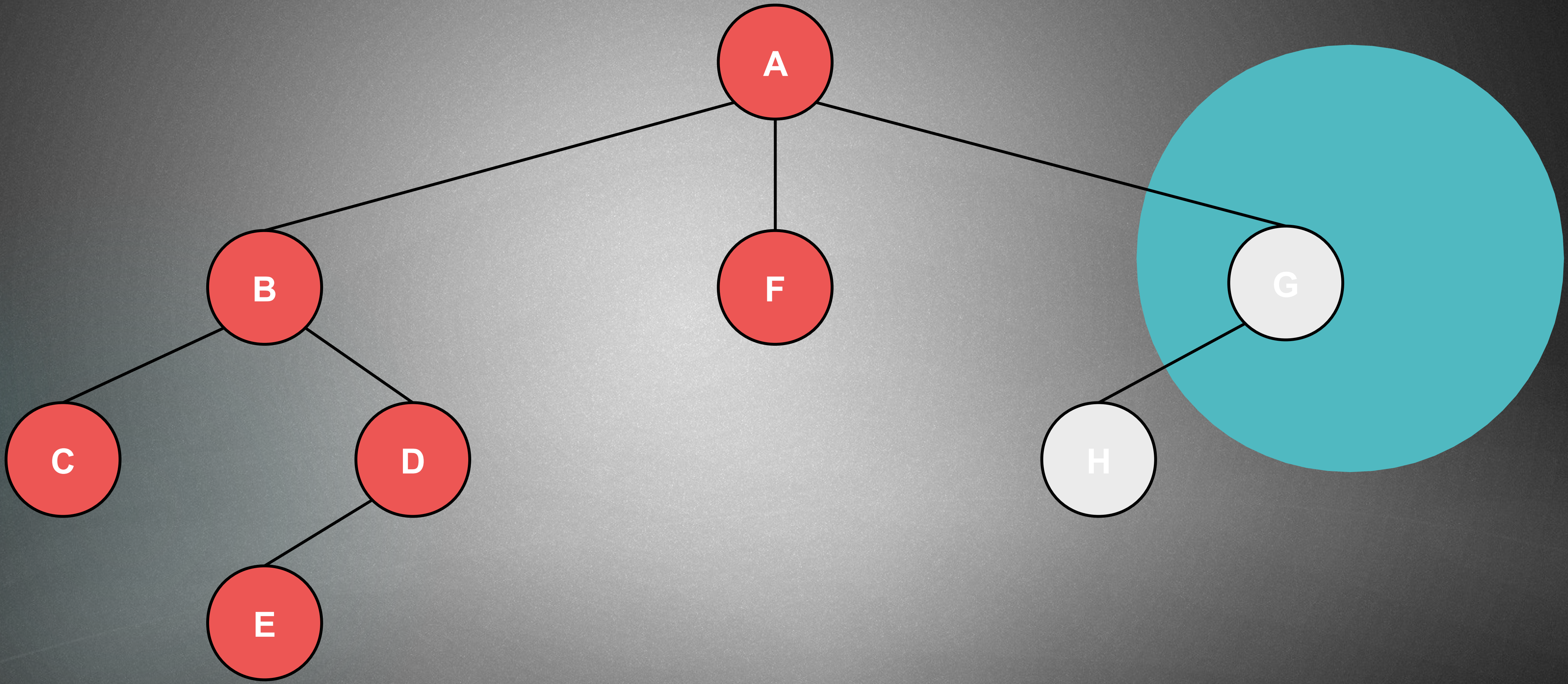
Recursively check every unmarked vertices!



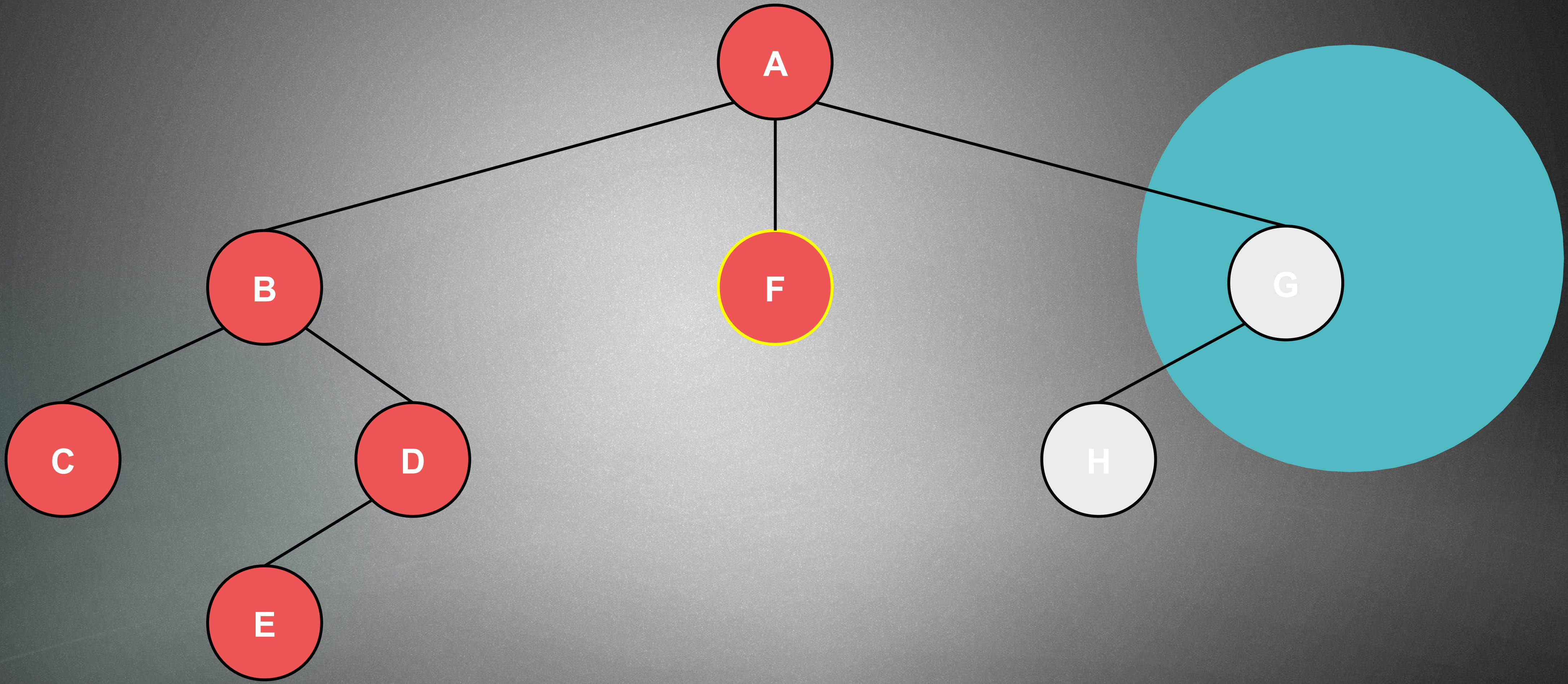
Recursively check every unmarked vertices!



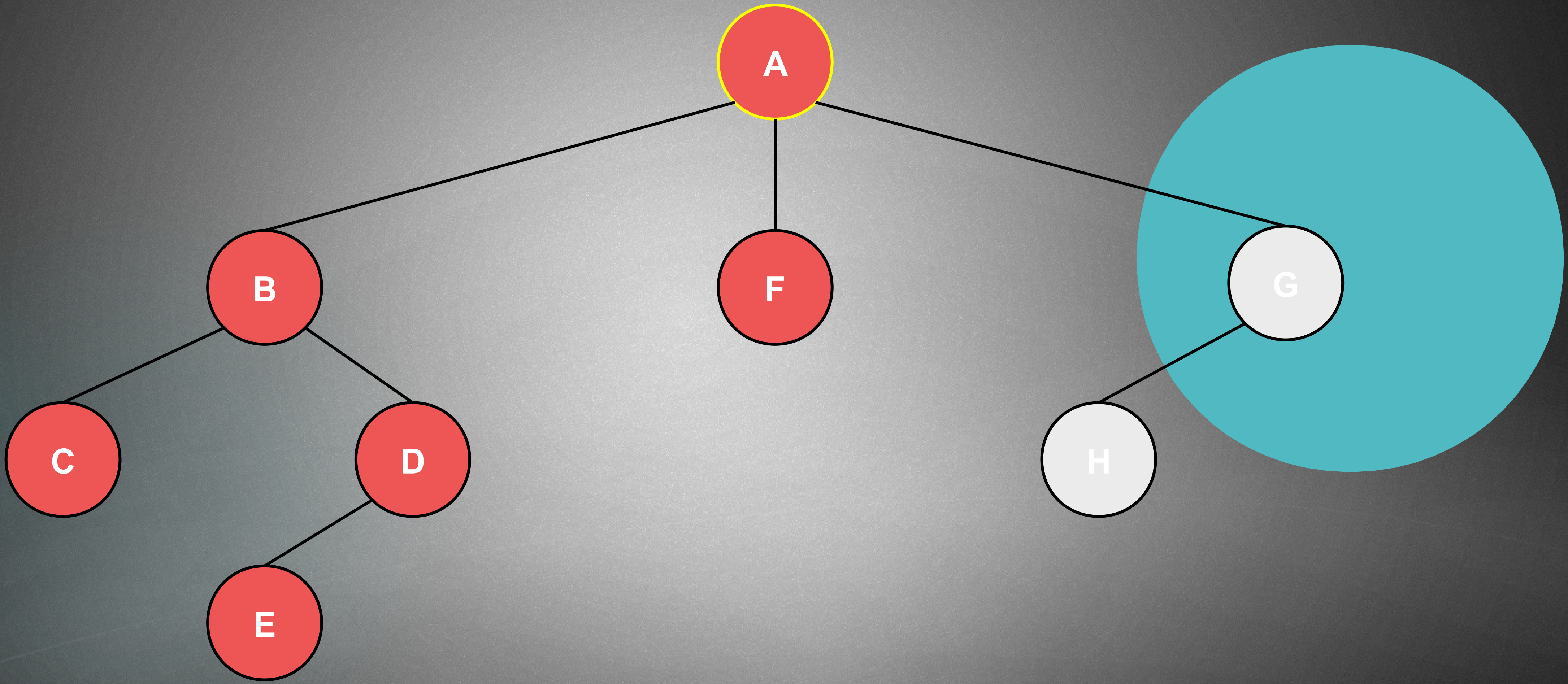
Recursively check every unmarked vertices!



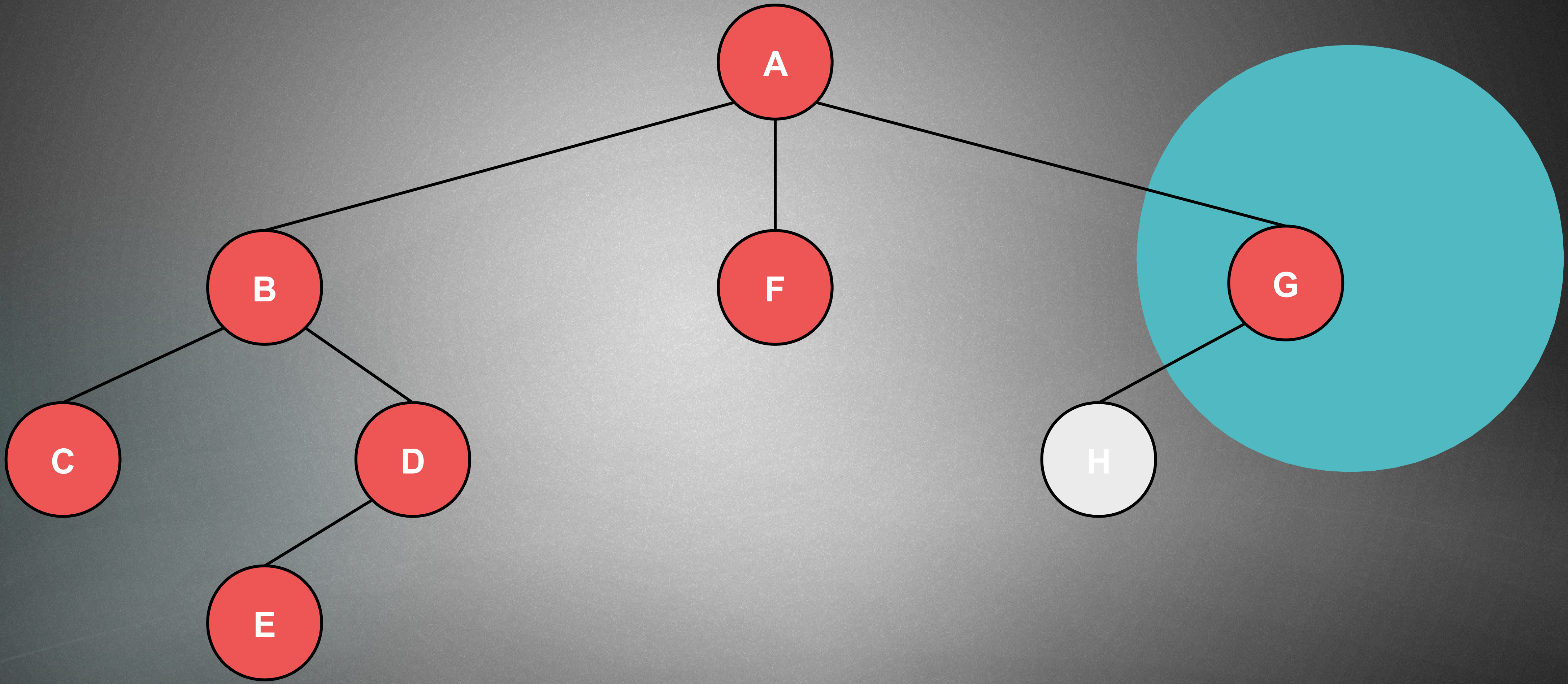
Recursively check every unmarked vertices!



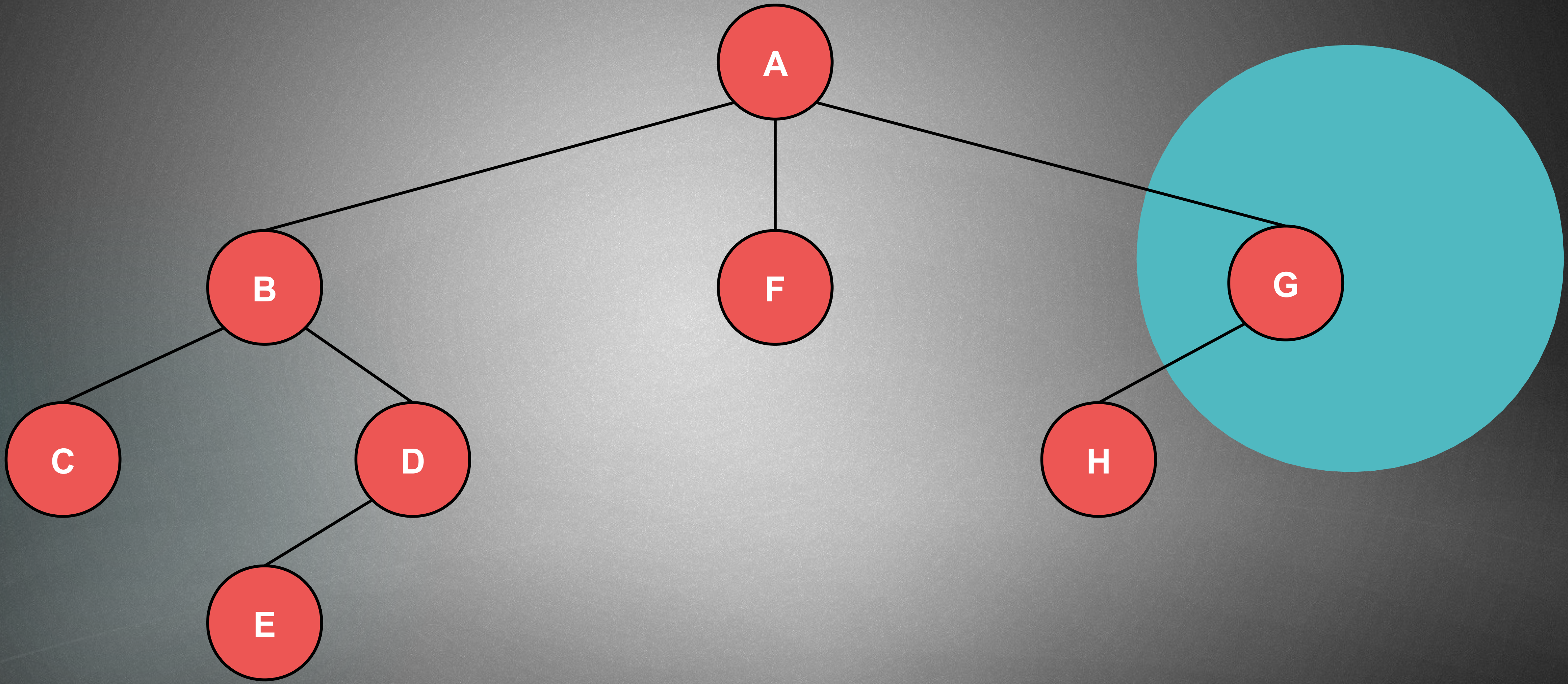
Recursively check every unmarked vertices!



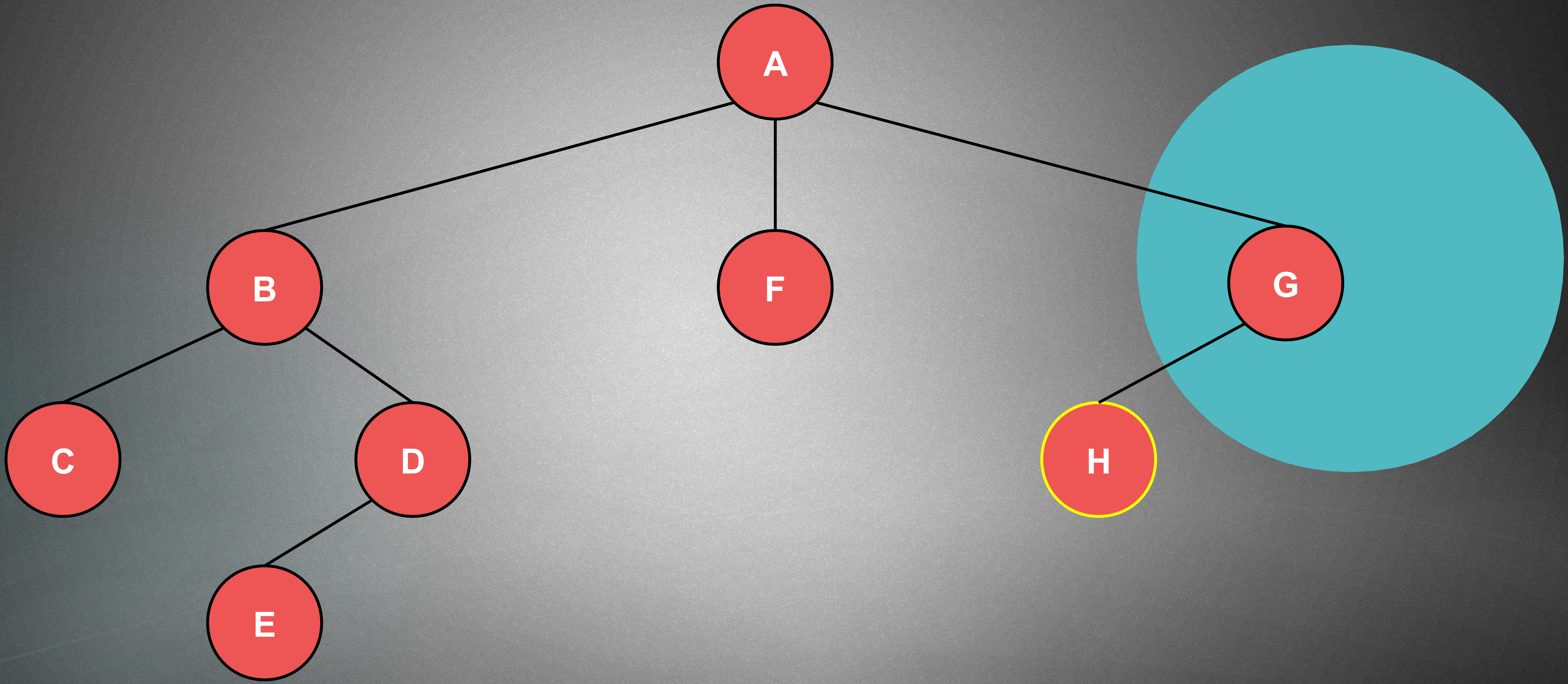
Recursively check every unmarked vertices!



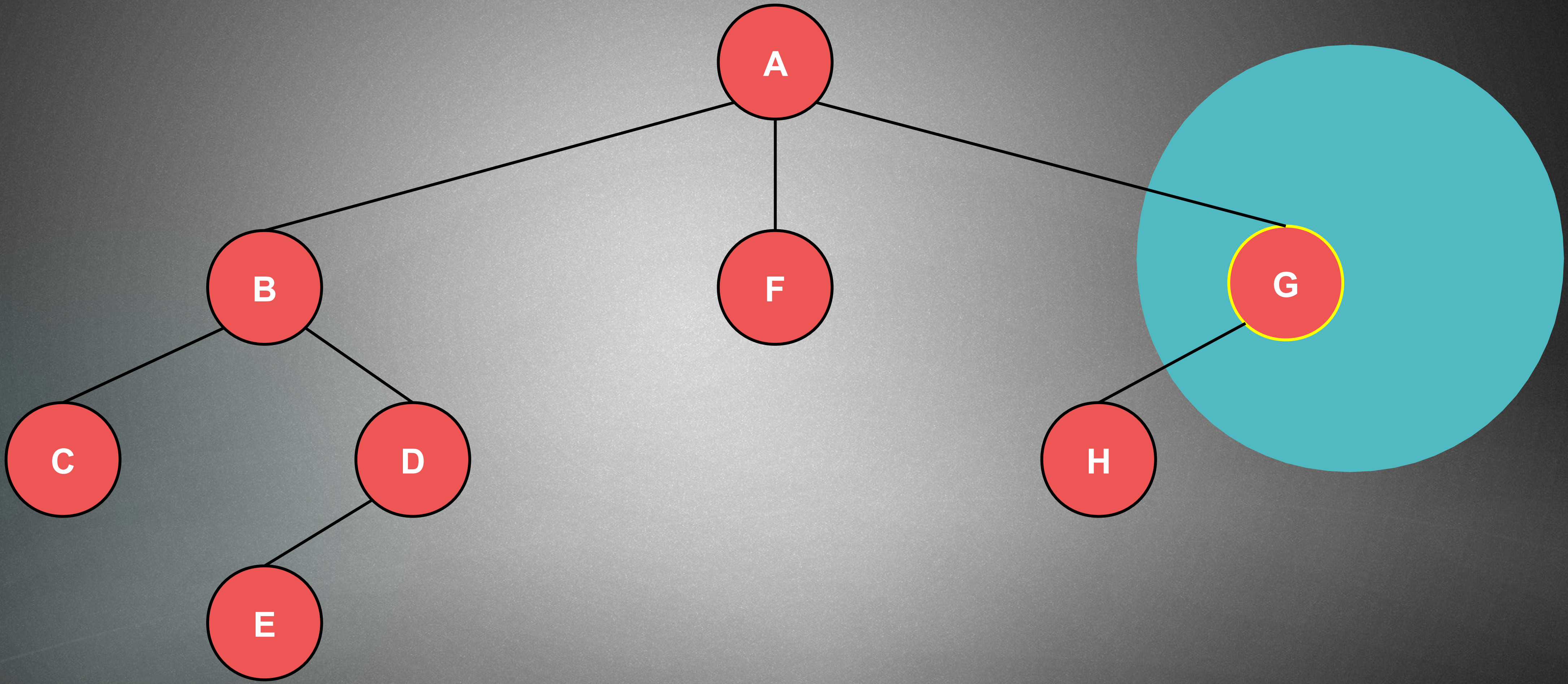
Recursively check every unmarked vertices!



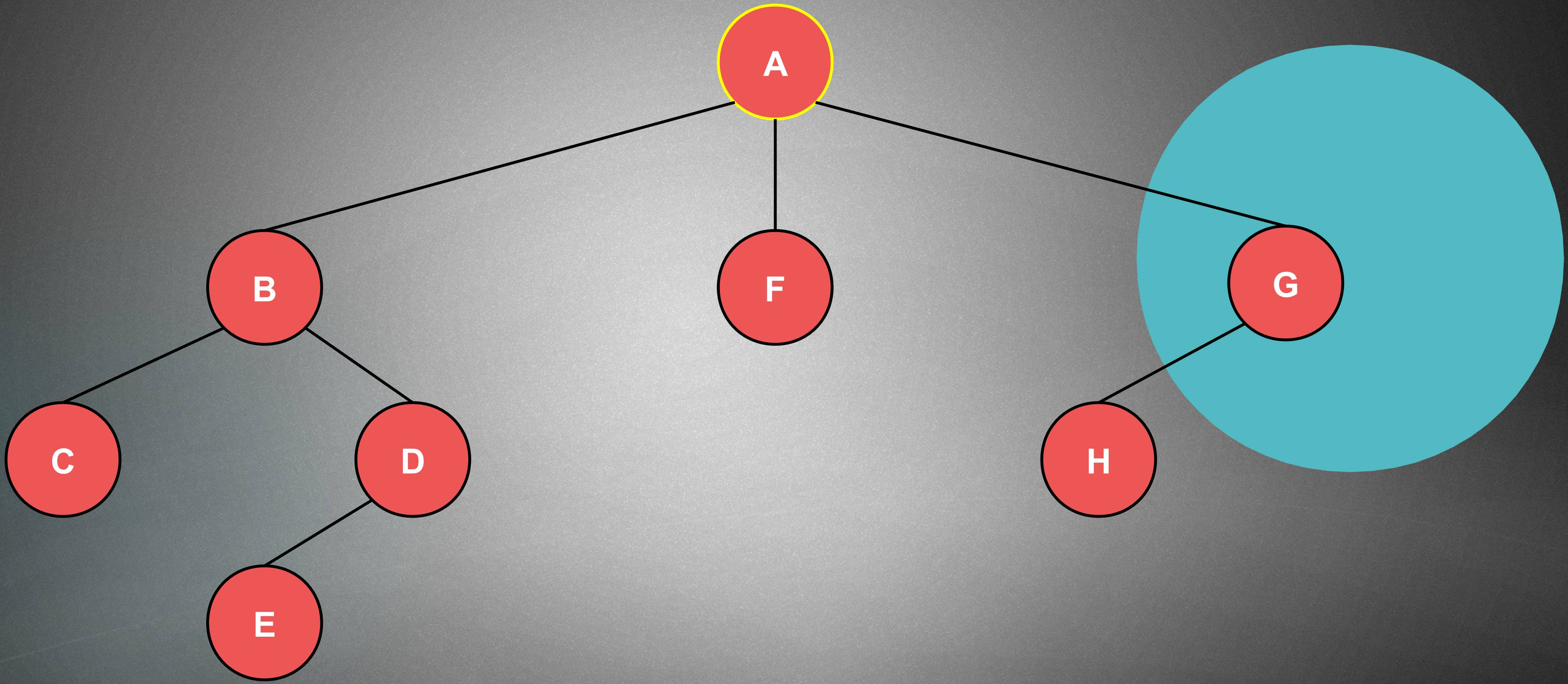
Recursively check every unmarked vertices!



Recursively check every unmarked vertices!

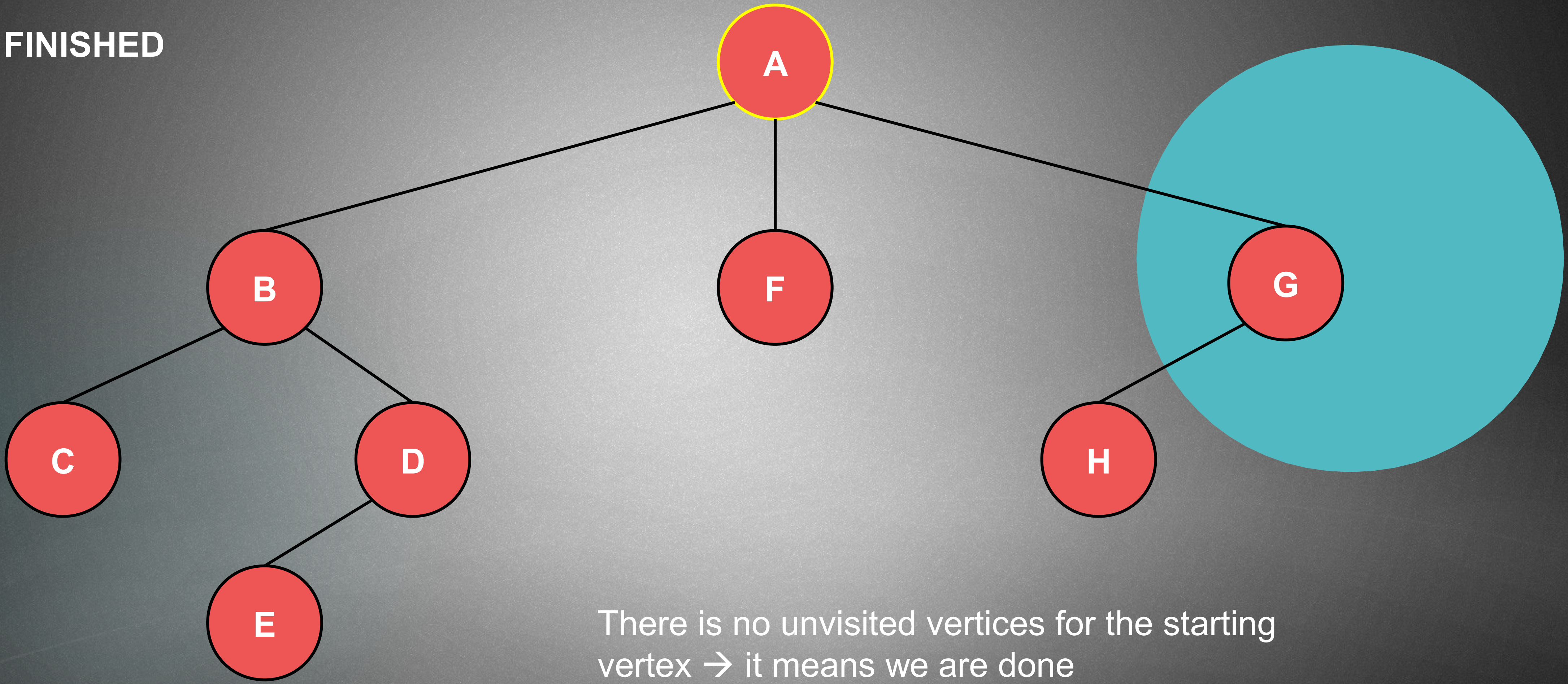


Recursively check every unmarked vertices!



Recursively check every
unmarked vertices!

FINISHED

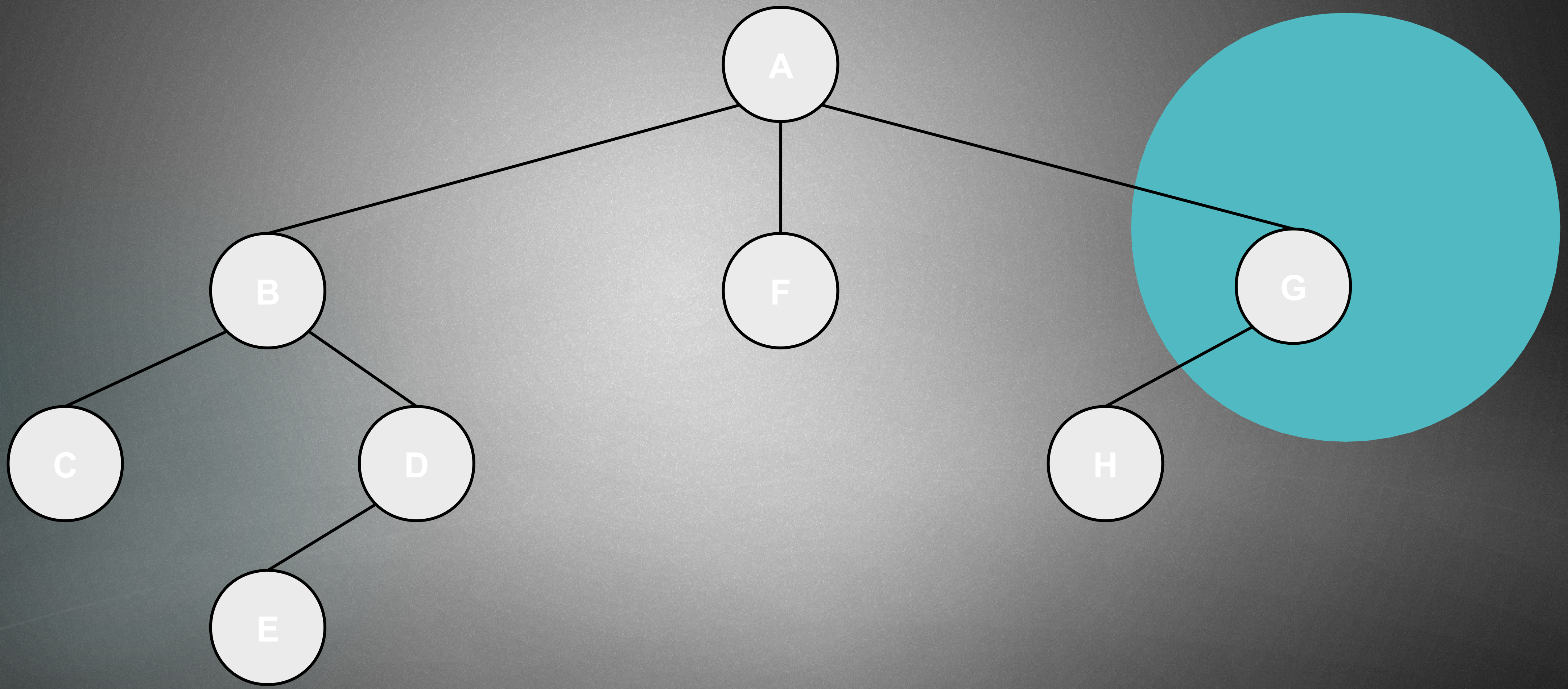


There is no unvisited vertices for the starting
vertex → it means we are done

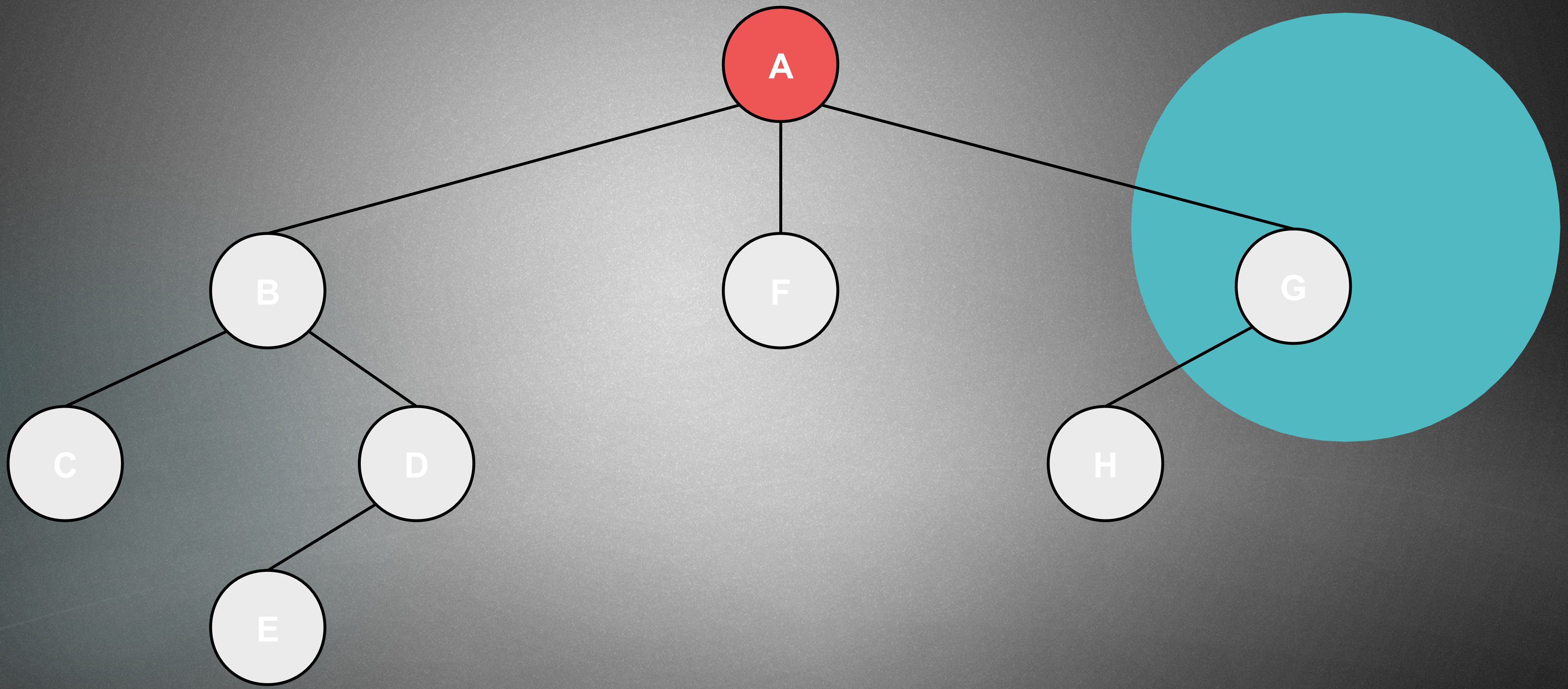
Applications

- Topological ordering
- Kosaraju algorithm for finding strongly connected components in a graph which can be proved to be very important in recommendation systems (youtube)
- Detecting cycles (checking whether a graph is a DAG or not)
- Generating mazes OR finding way out of a maze

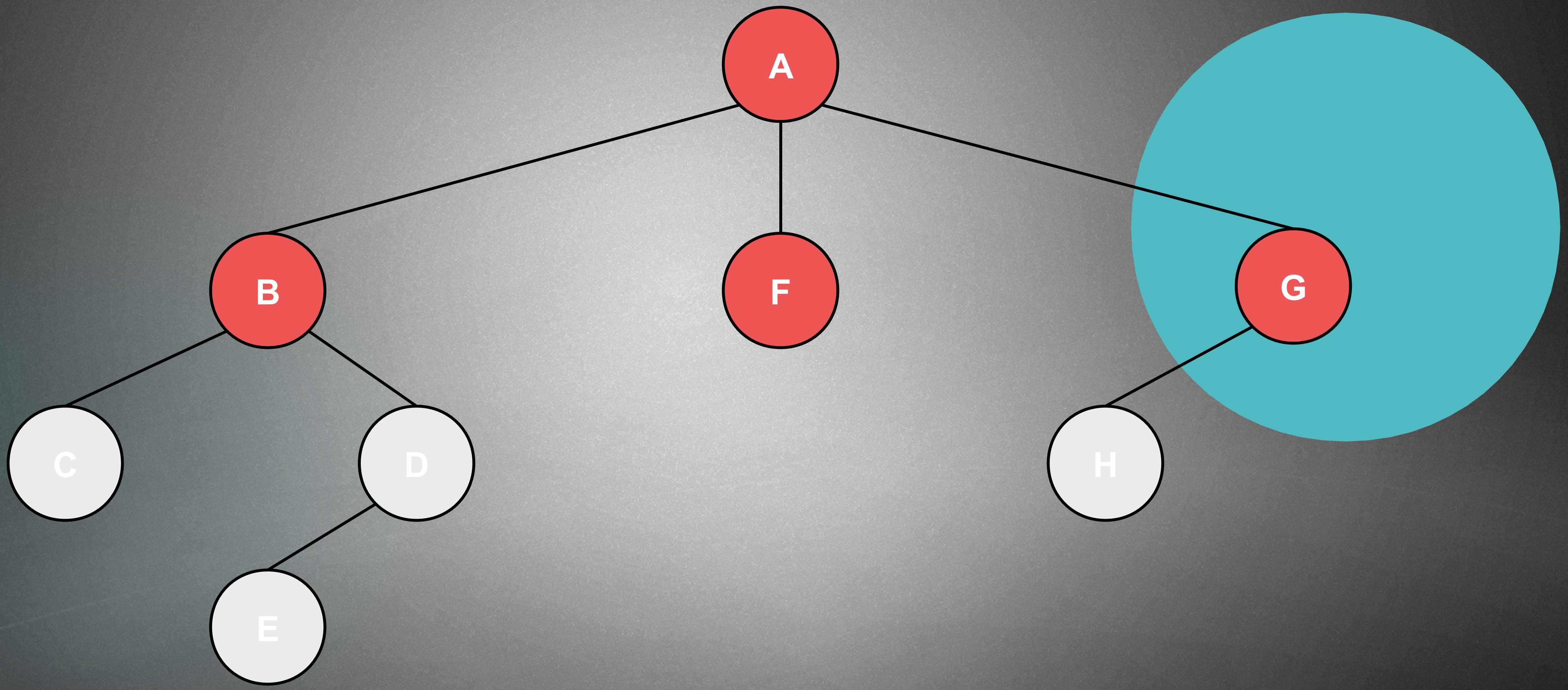
Revisiting breadth-first search



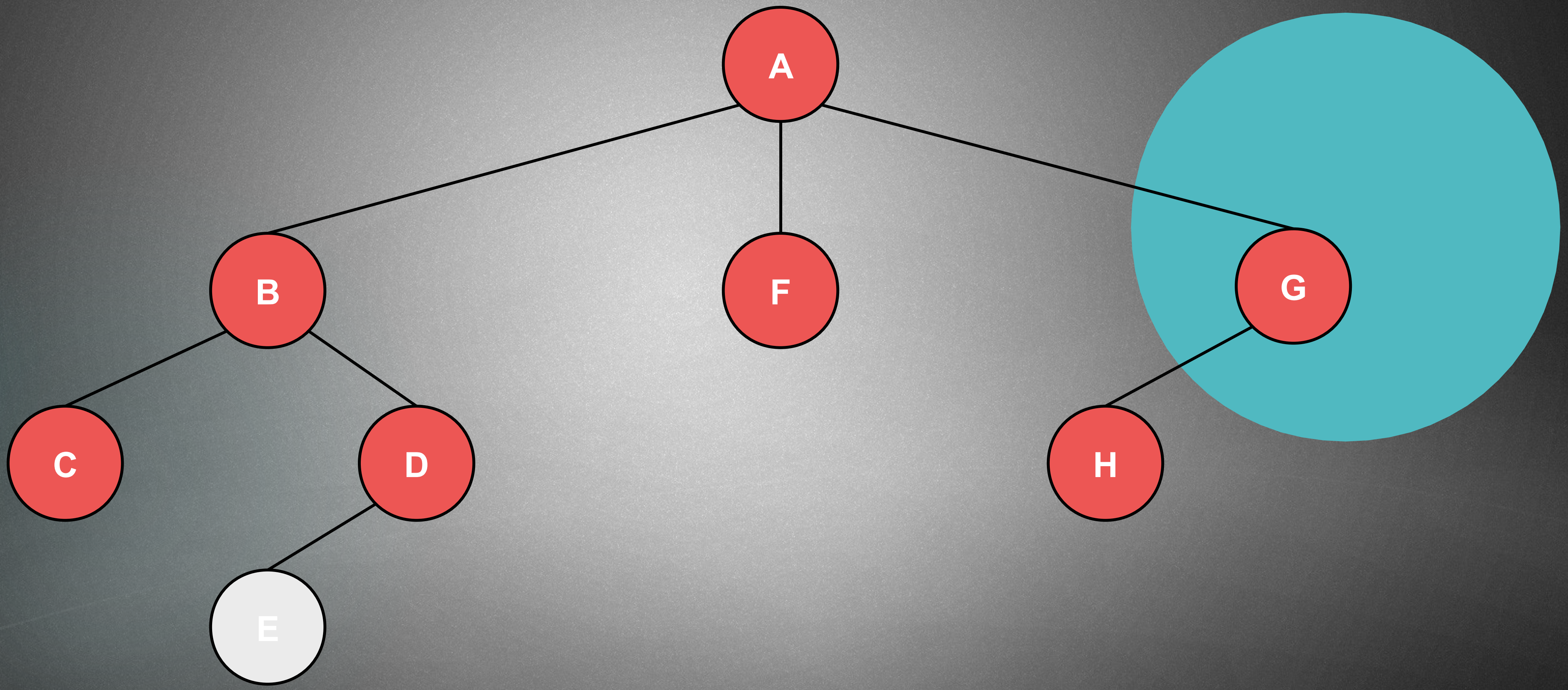
Revisiting breadth-first search



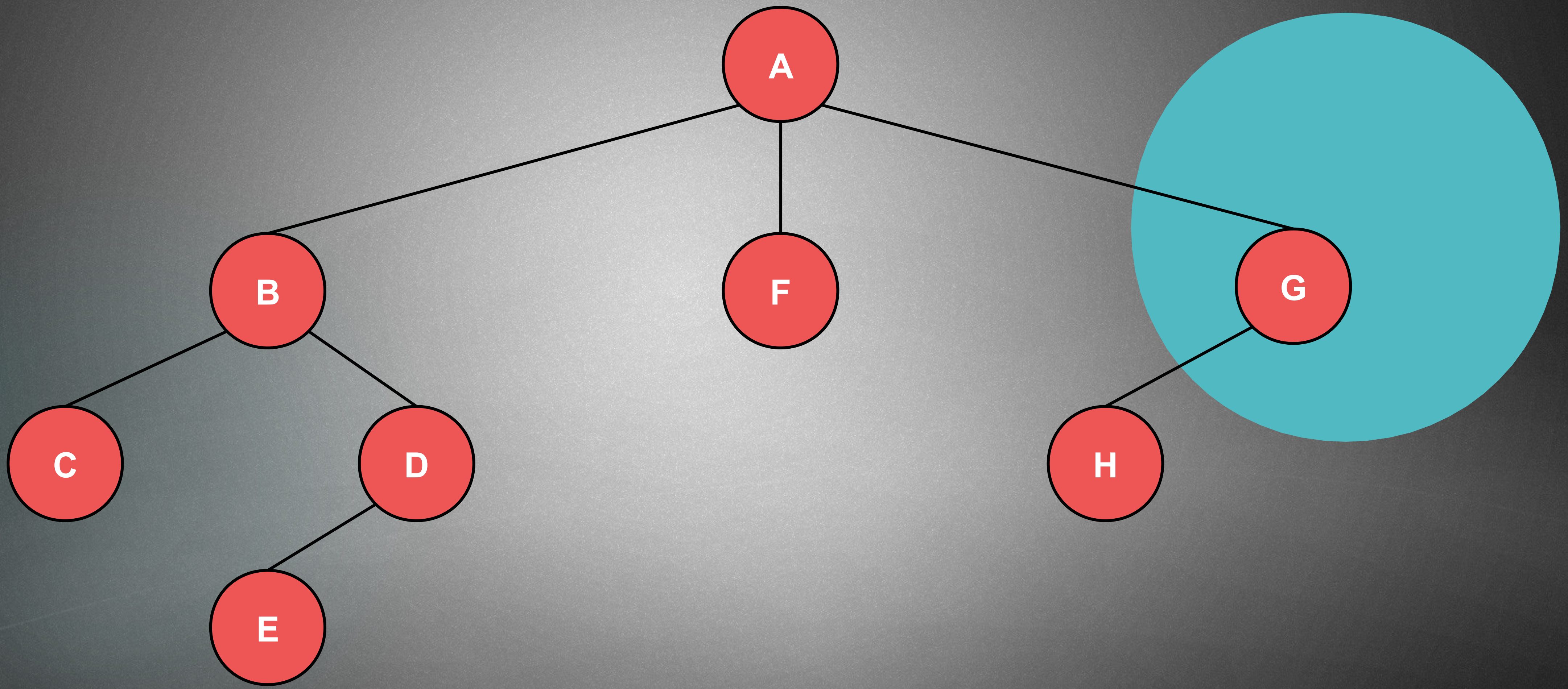
Revisiting breadth-first search



Revisiting breadth-first search

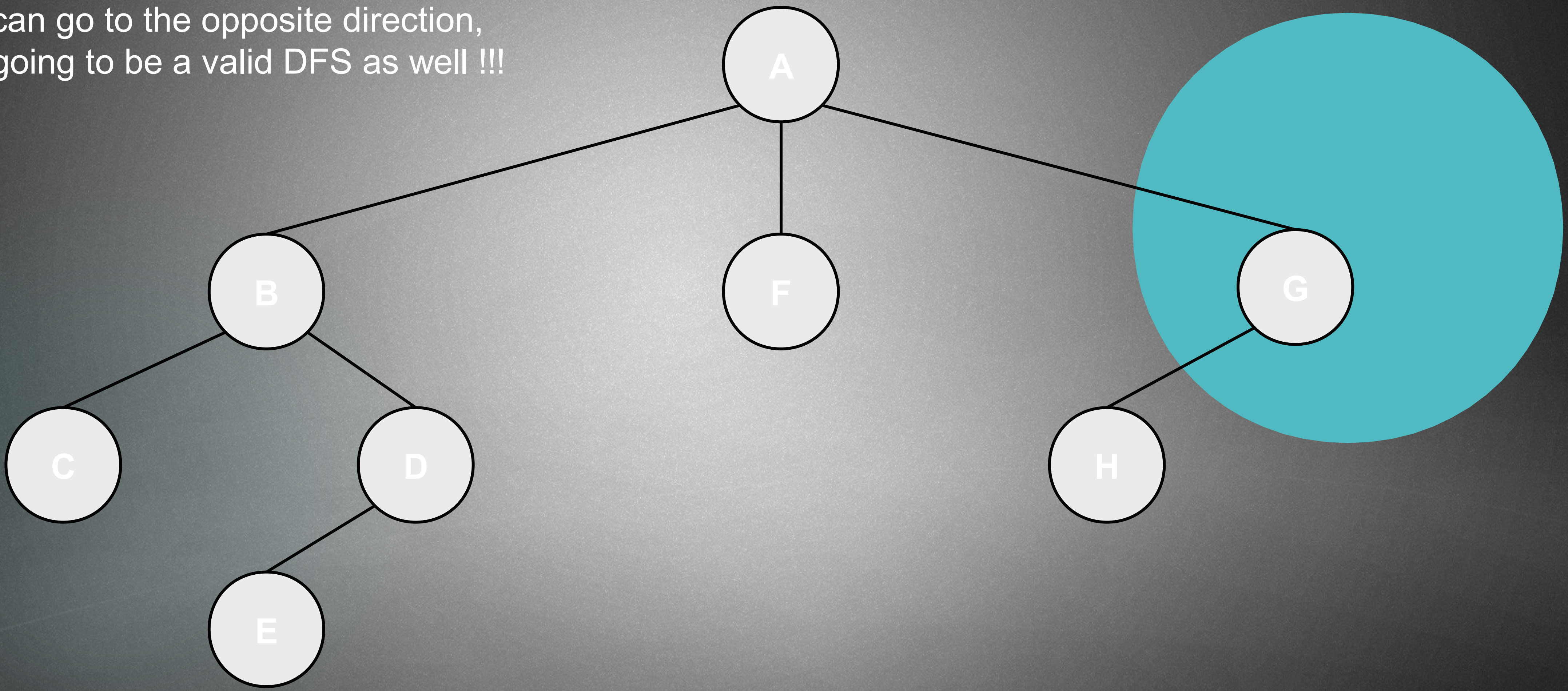


Revisiting breadth-first search



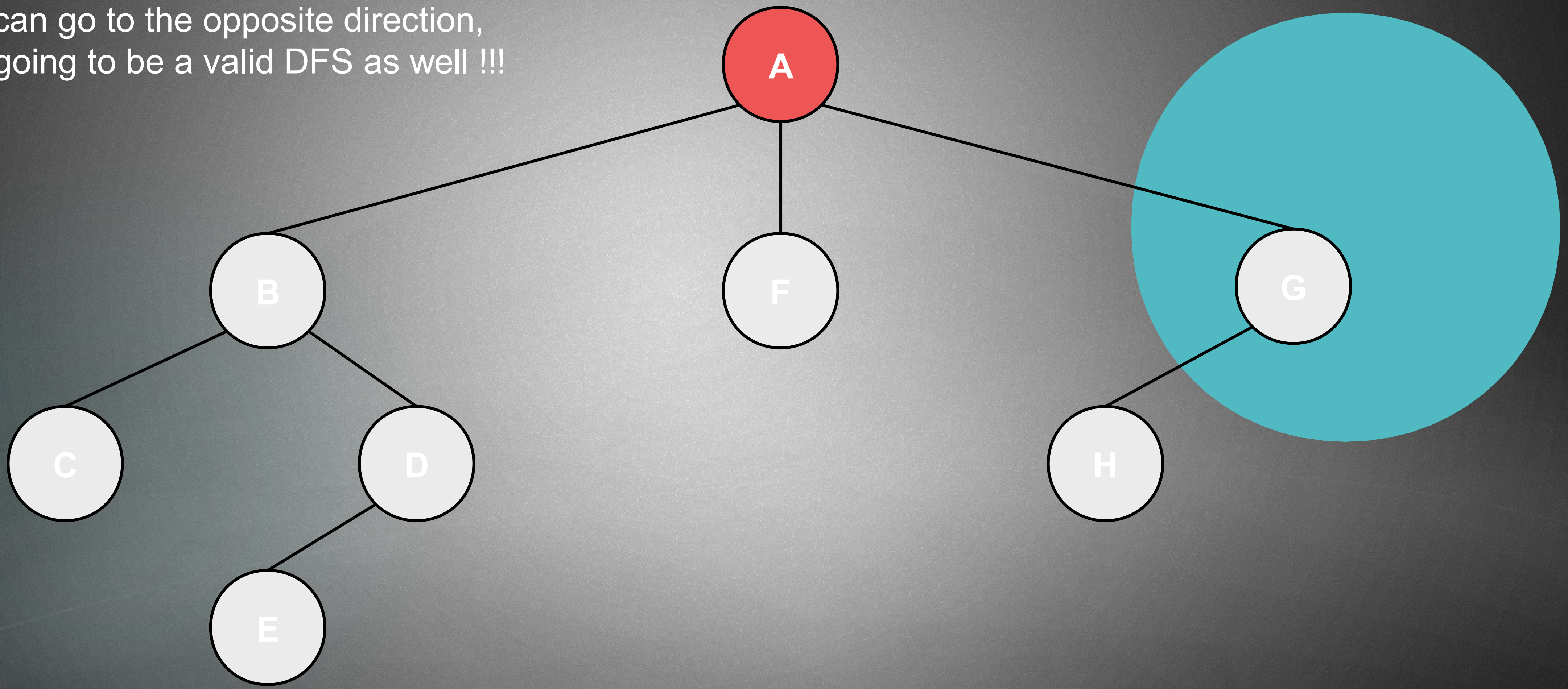
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



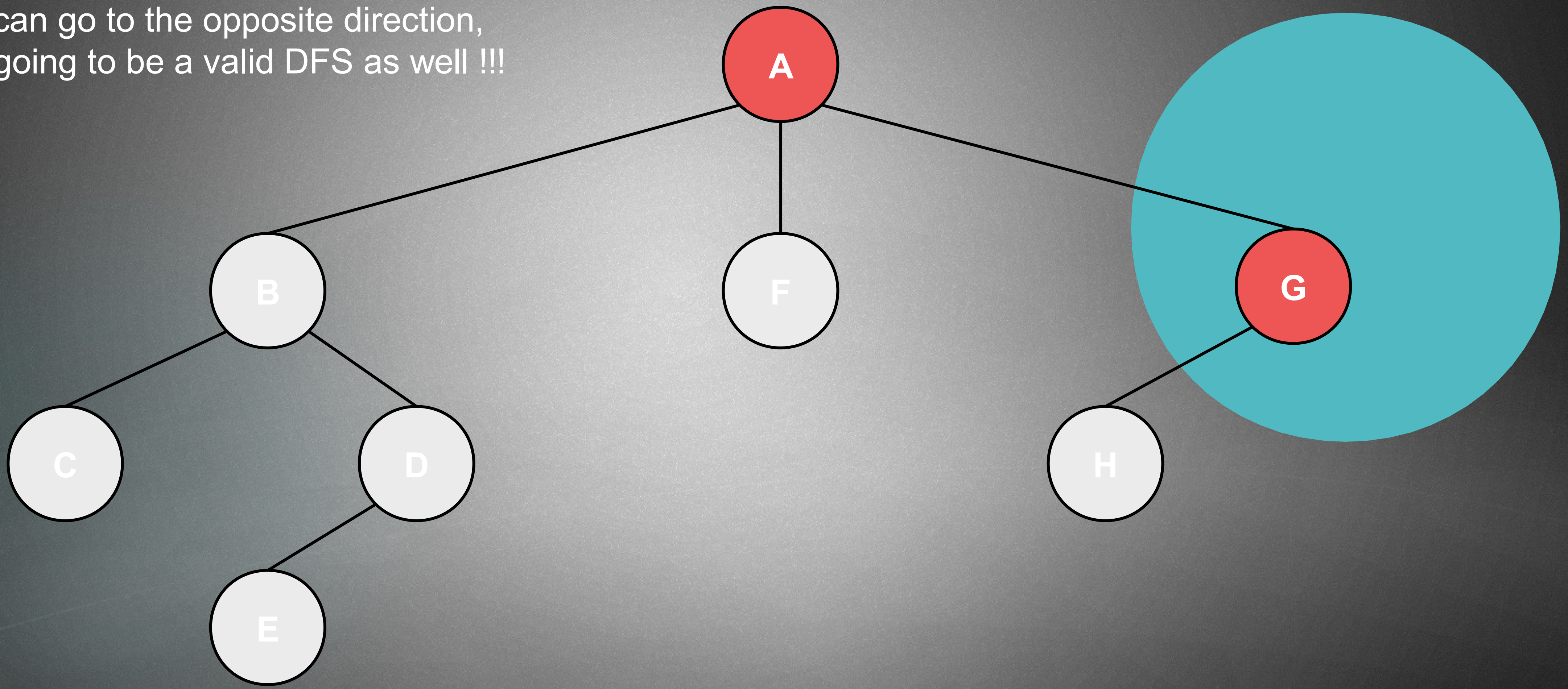
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



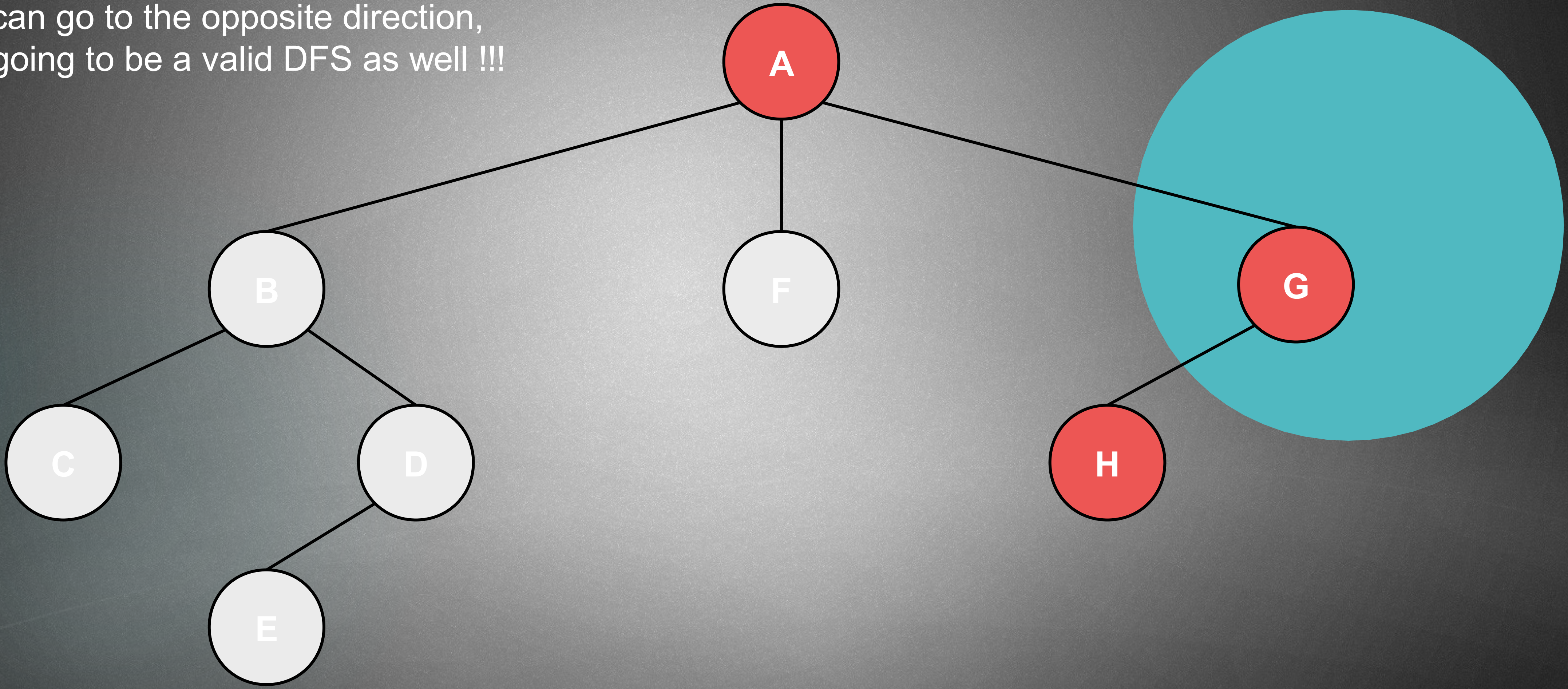
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



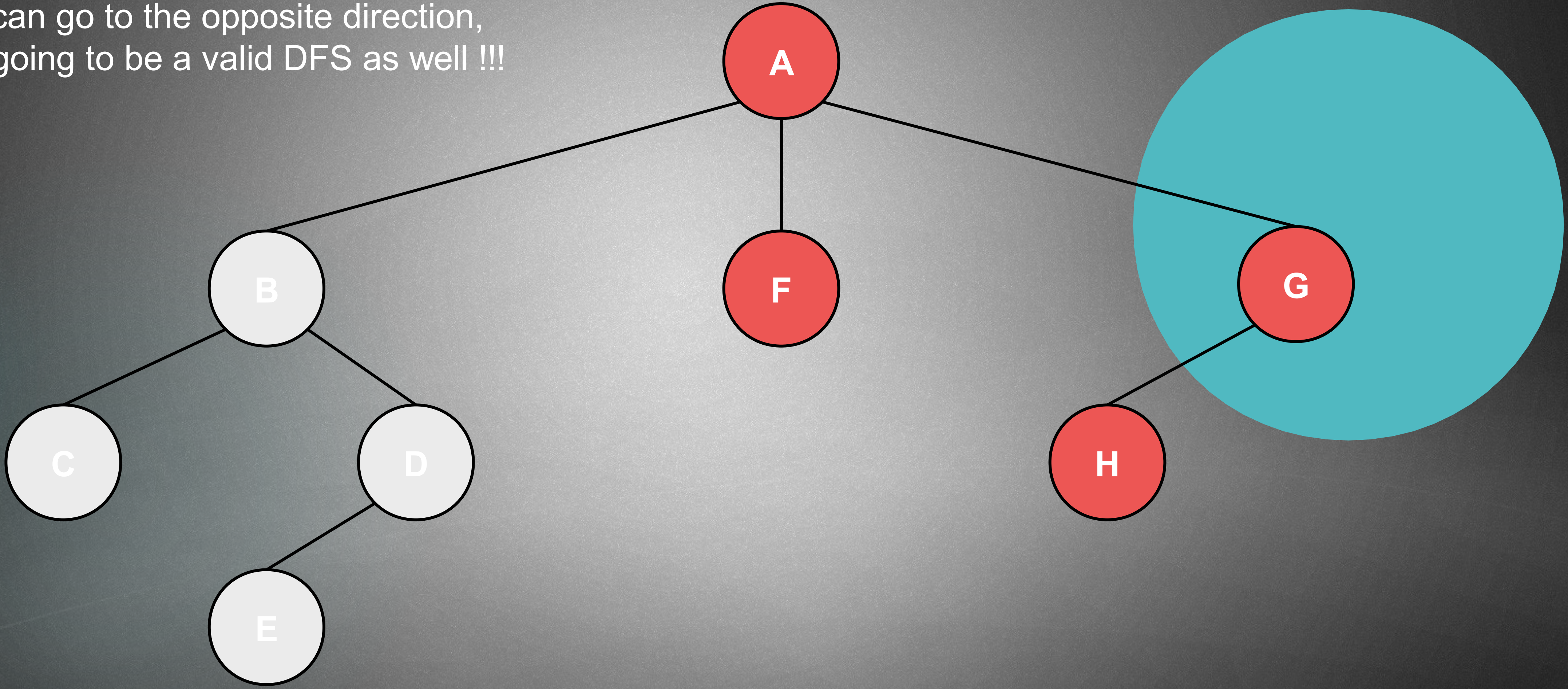
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



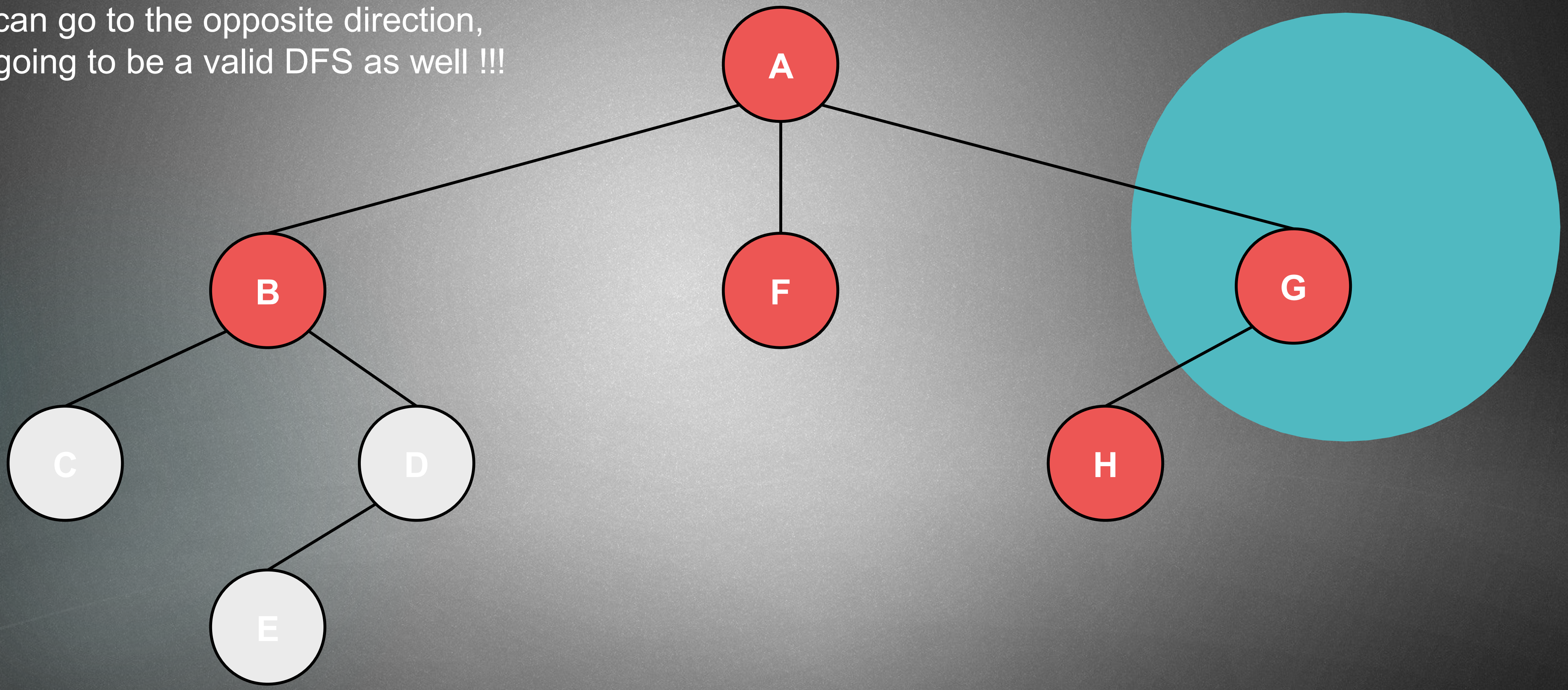
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



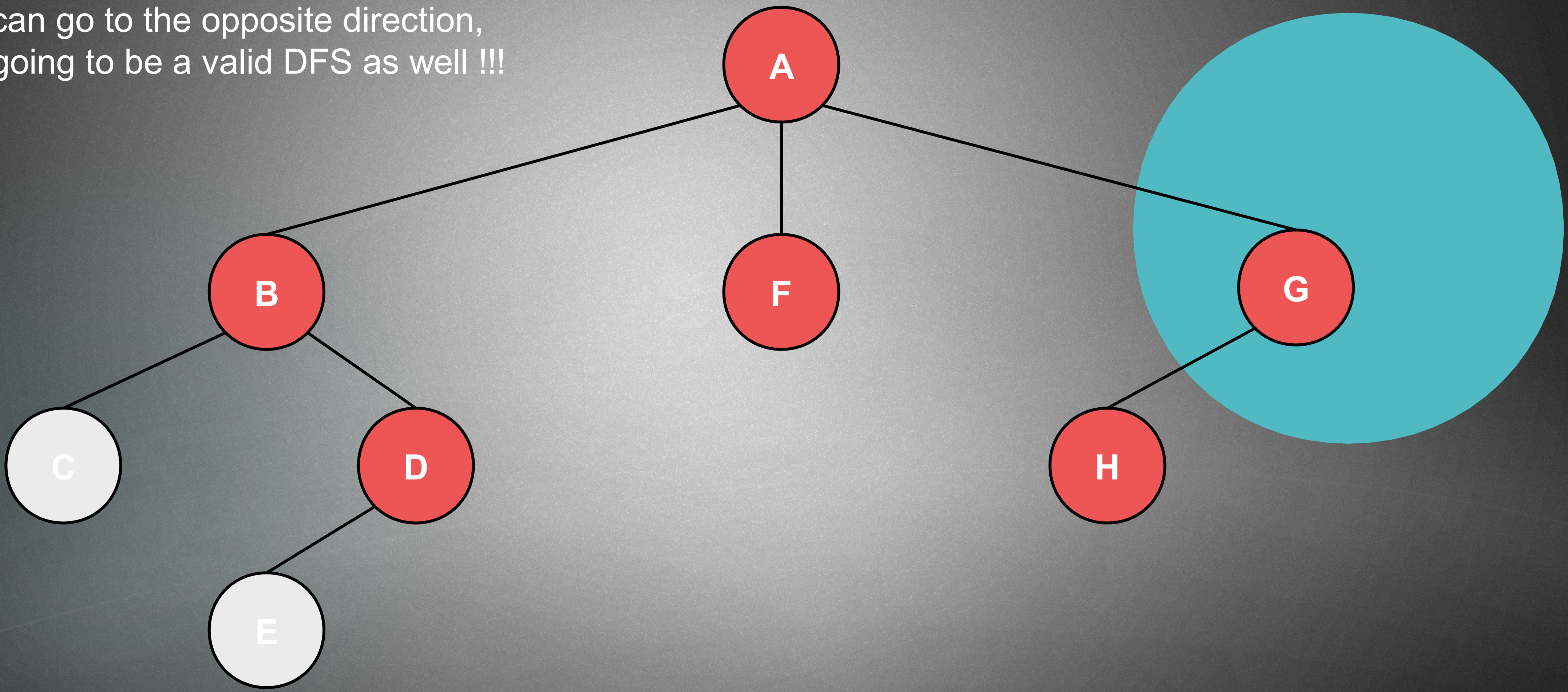
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



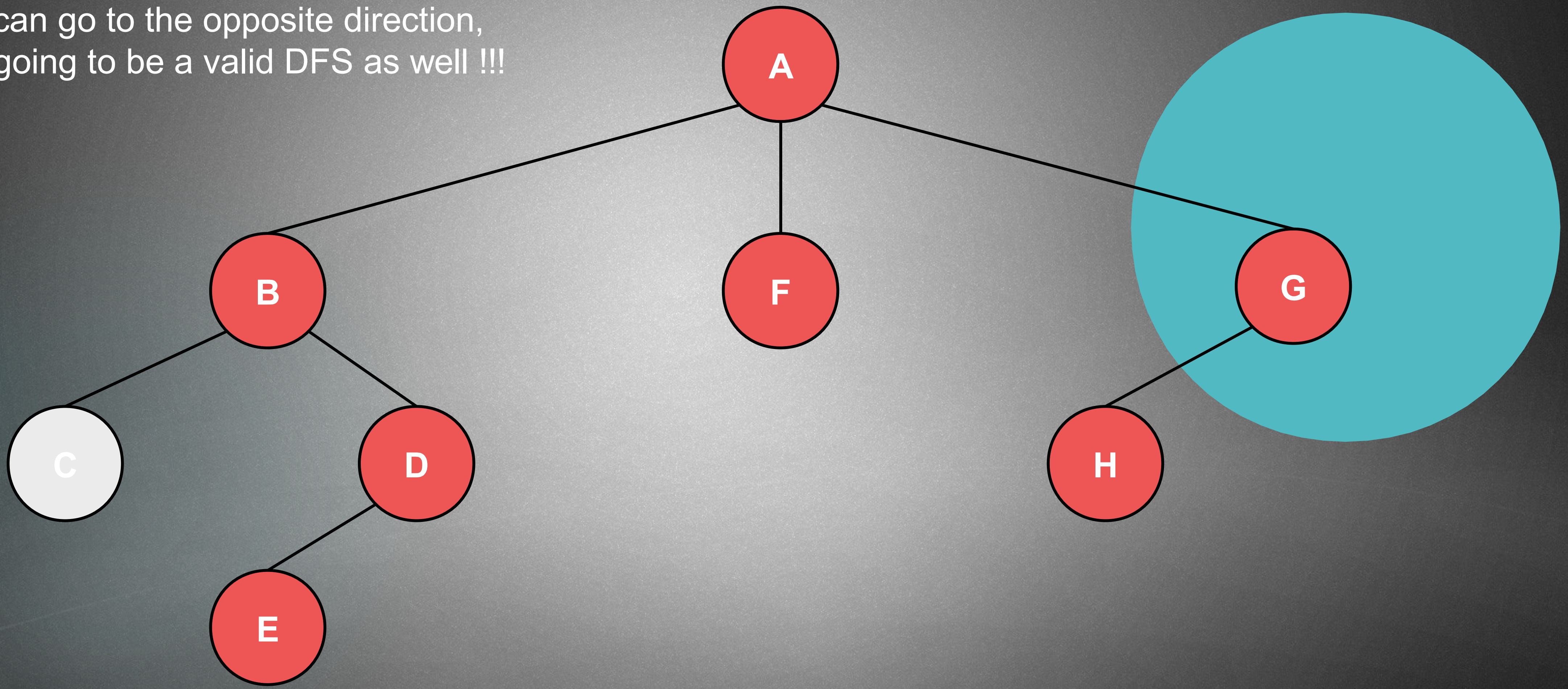
Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!



Symmetry in DFS

We can go to the opposite direction,
it is going to be a valid DFS as well !!!