GRAPH ALGORITHMS

GRAPH THEORY

Graphs

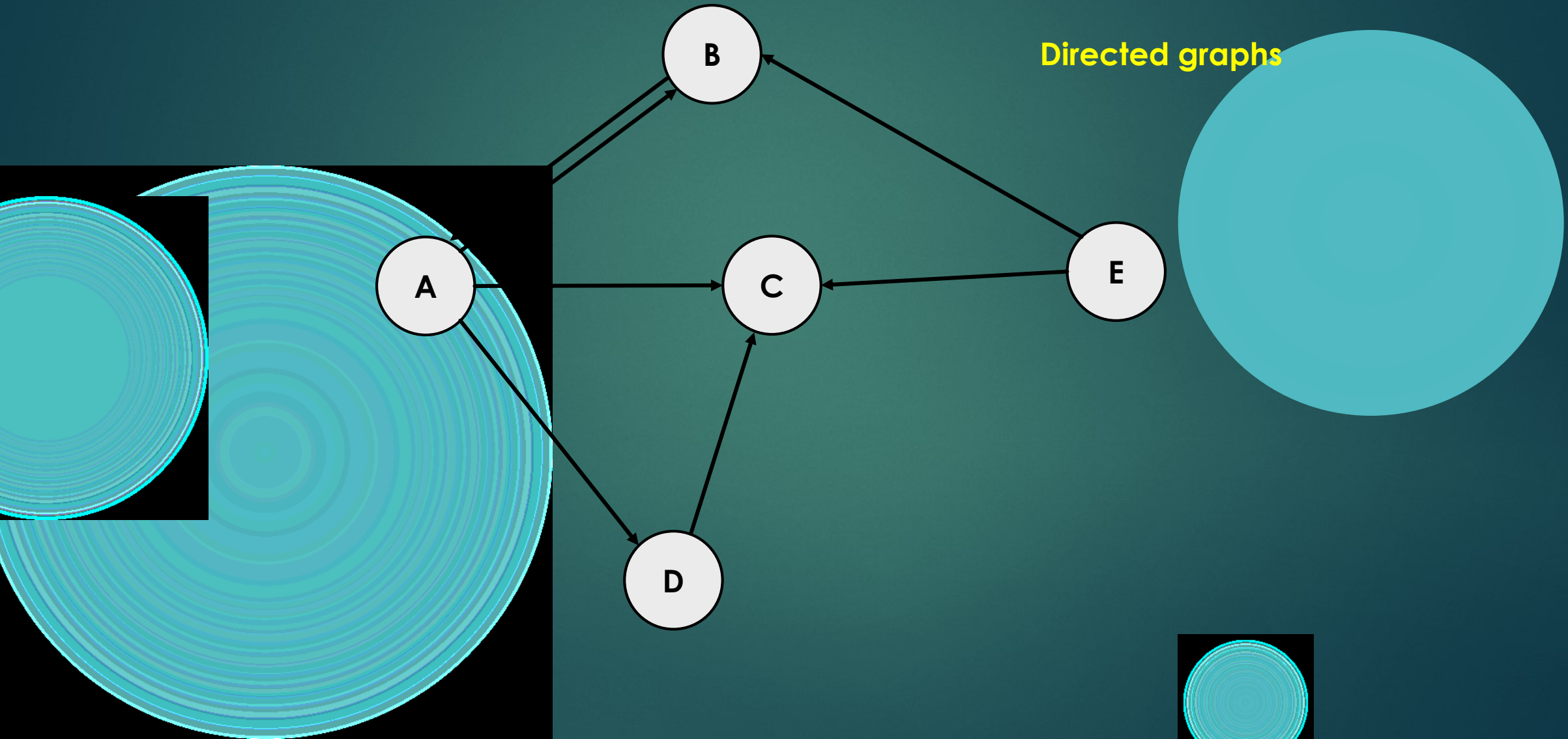
- Graphs **G(V,E)** are mathematical structures to model pairwise relations between given objects.
- A graph is made up of vertices/nodes and edges.
- There are two types of graphs: directed and undirected graphs.

Graphs Applications

- Shortest path algorithm (GPS, high frequency trading)
- Graph traversing: web crawlers for Google
- Spanning trees
- Maximum flow problem: lots of problems can be reduced to maximum flow!!!
- Because there are two representations for graphs, we can handle these problems with metrices as well
  - That's why most Google's algorithms have something to do with matrix related operations although they are graph algorithms.

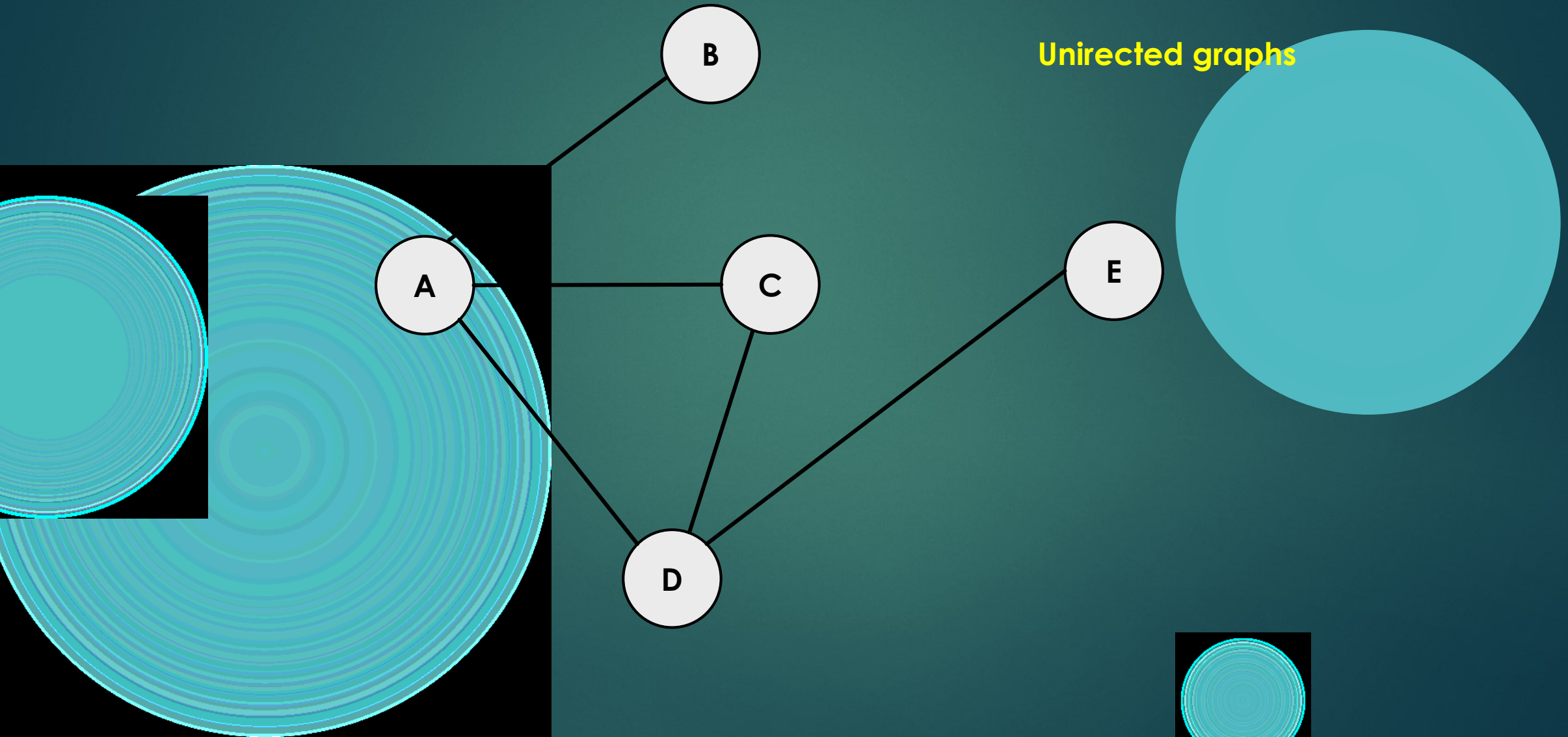
Graphs

Directed graphs

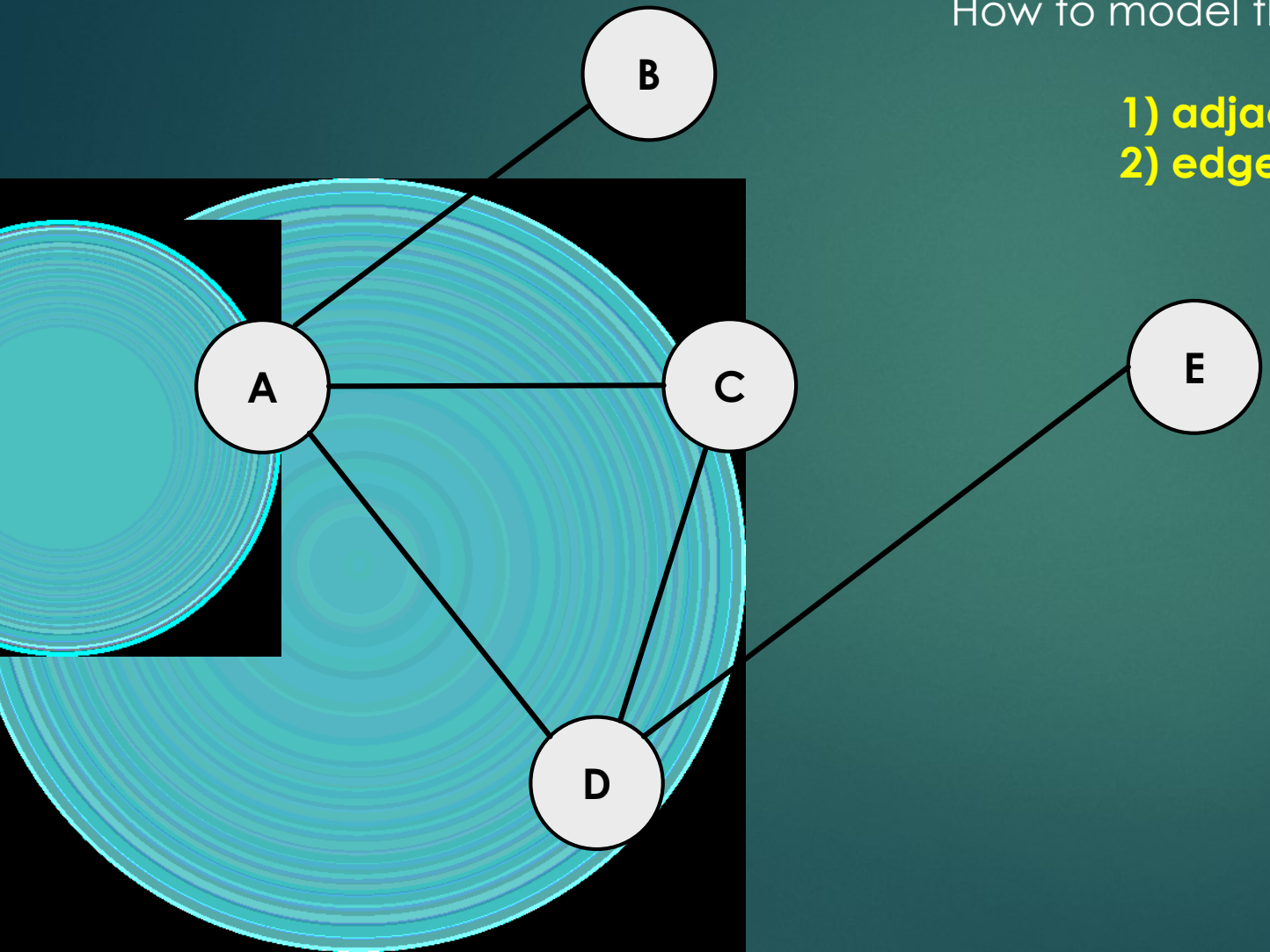


Graphs

Unirected graphs



Graphs



How to model them in programming languages?

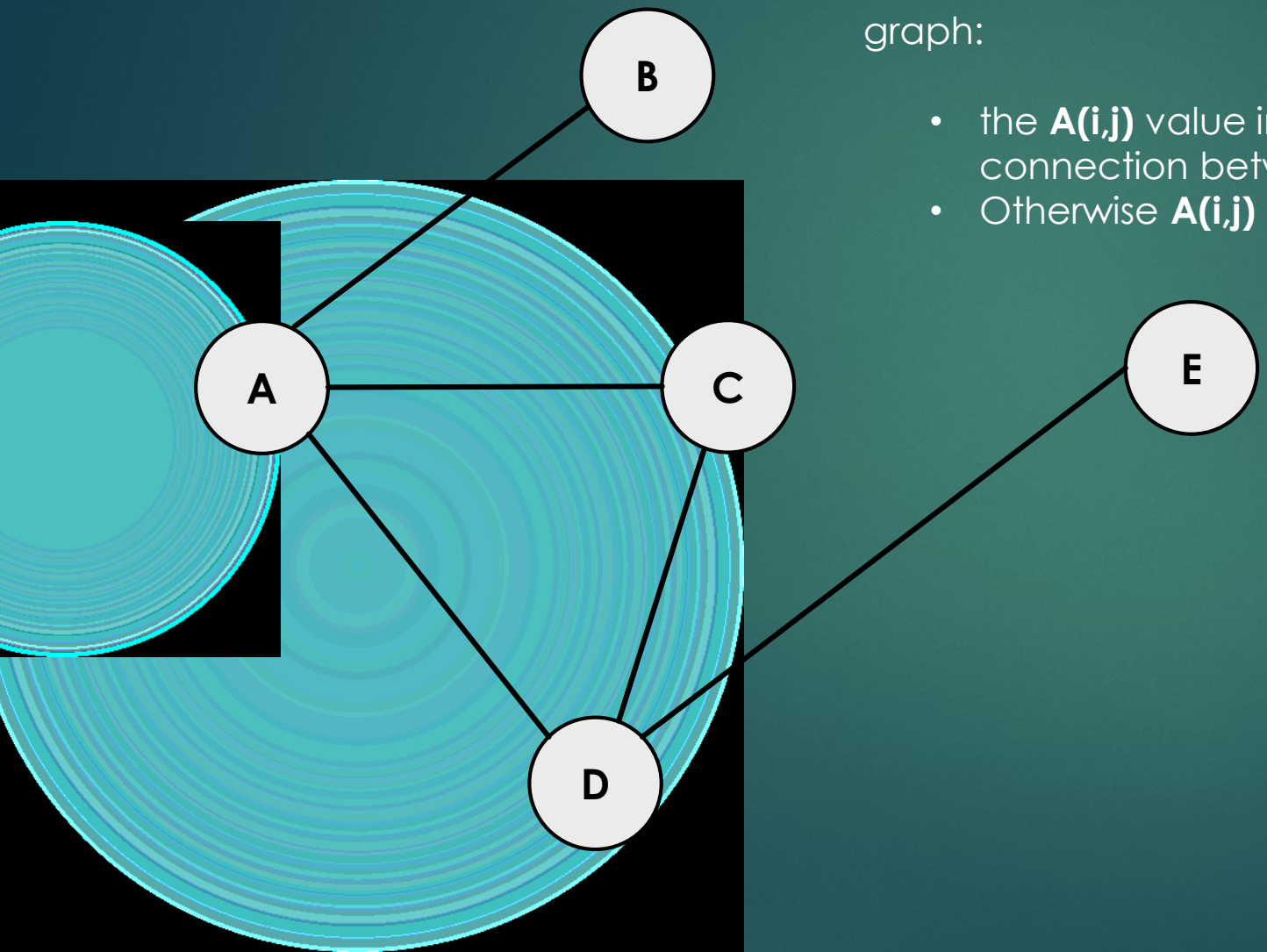
1) adjacency matrices
2) edge list representation

Graphs

Adjacency matrices

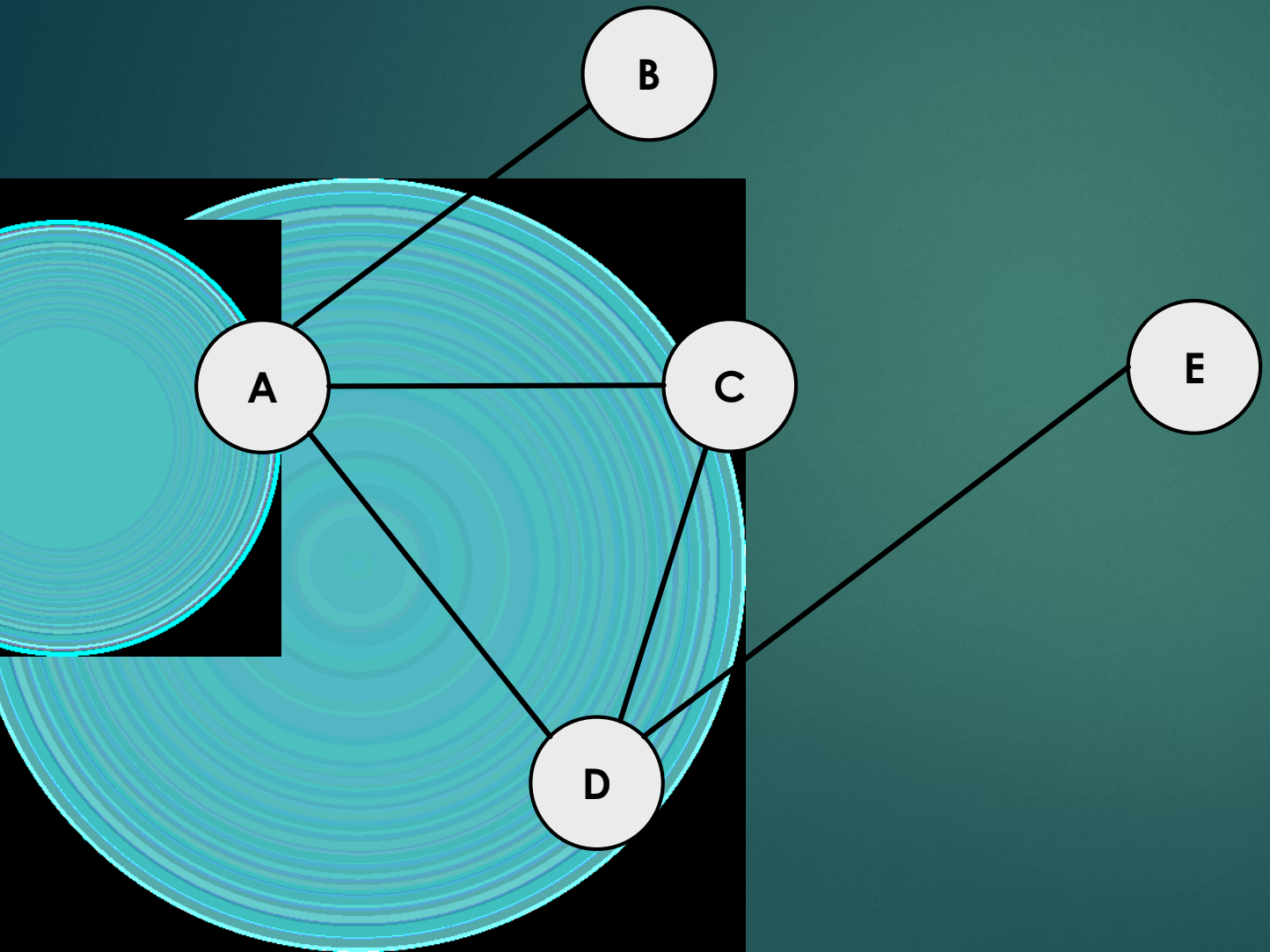
We have an A matrix constructed out of the vertices of the graph:

- the **A(i,j)** value in the matrix is **1** if there is a connection between node I and node j
- Otherwise **A(i,j)** is **0**



Graphs

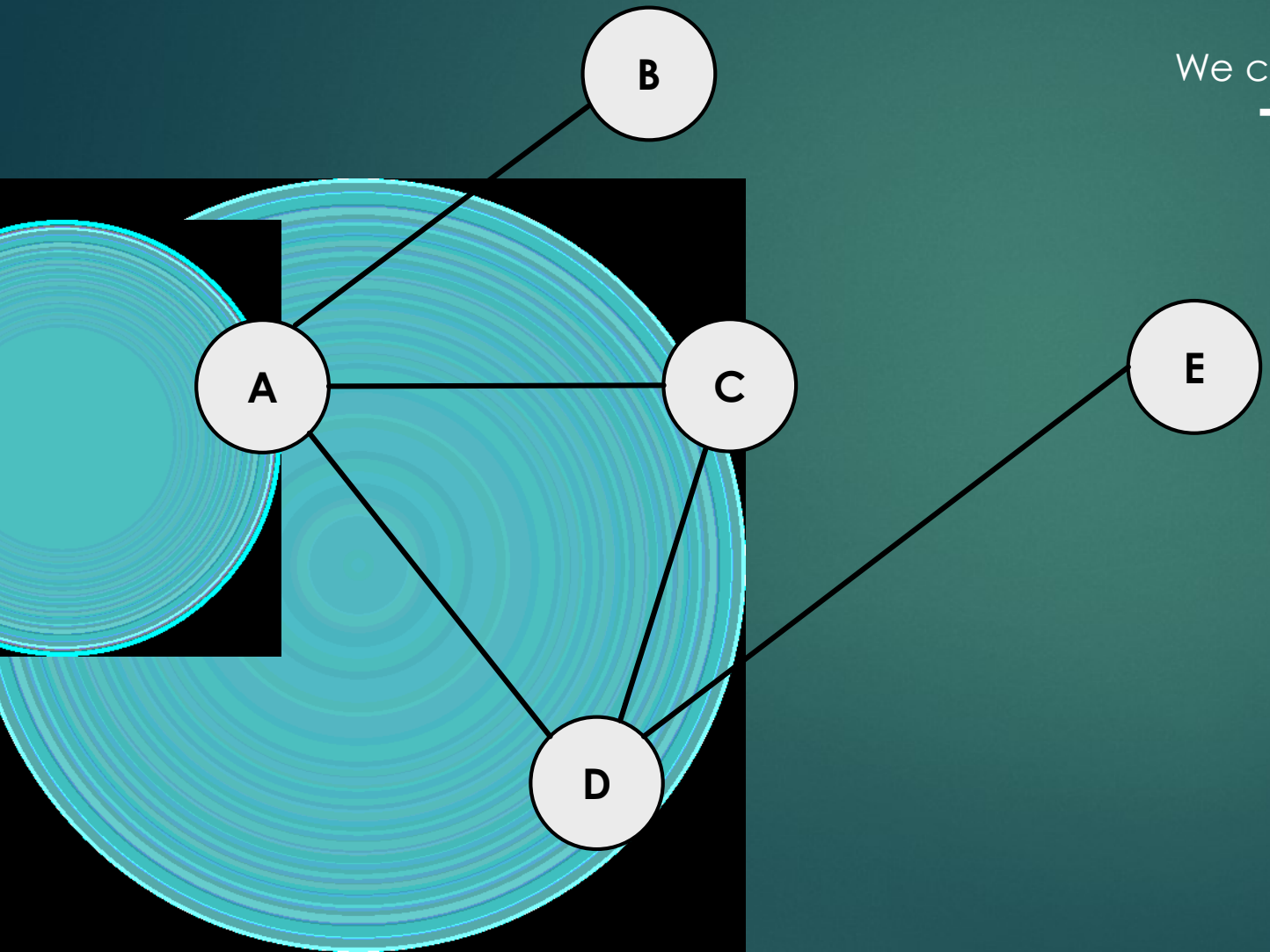
Adjacency matrices



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	0
C	1	0	0	1	0
D	1	0	1	0	1
E	0	0	0	1	0

Graphs

Edge list representation



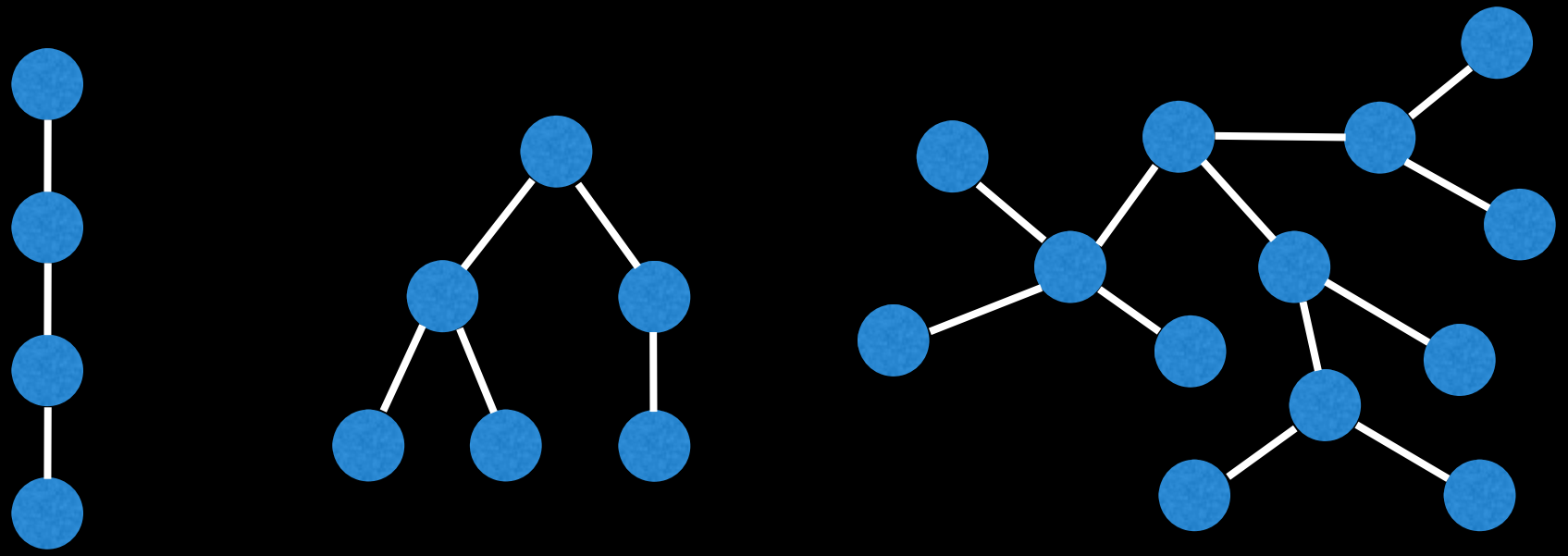
We create a **Vertex** class
➔ It stores the neighbors accordingly

```
class Vertex

    vetexName;
    visited;
    Vertex[]  neighbors;
```

Special Graphs

Trees!

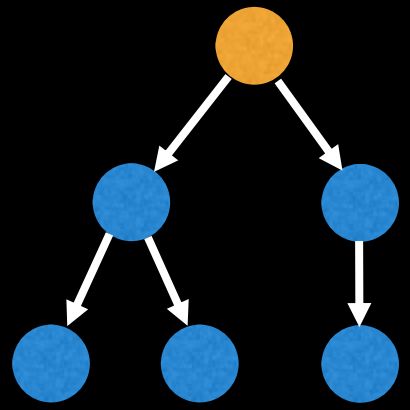
A **tree** is an **undirected graph with no cycles**.
Equivalently, it is a connected graph with N nodes and N-1 edges.



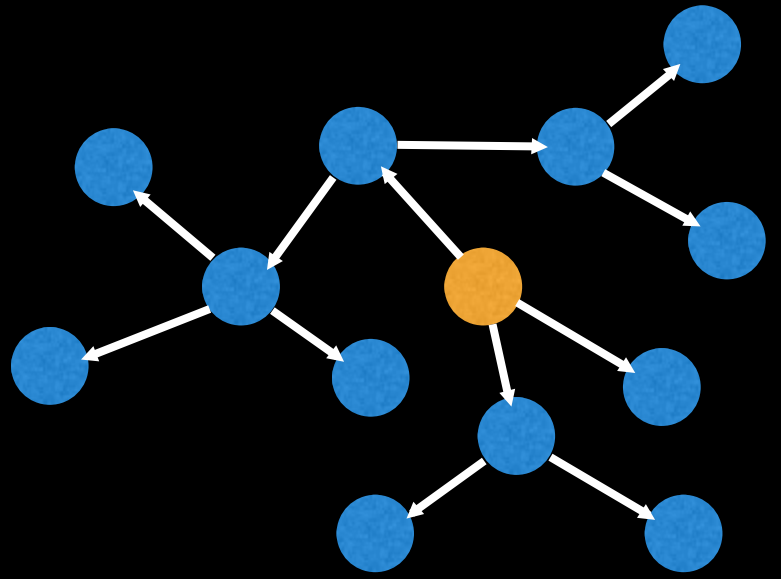
Rooted Trees!

A **rooted tree** is a tree with **a designated root node** where every edge either points away from or towards the root node.
When edges point away from the root the graph is called an **arborescence (out-tree)** and anti-arborescence (in-tree) otherwise.

In-tree



Out-tree

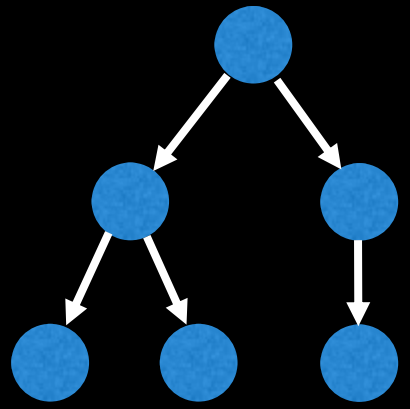


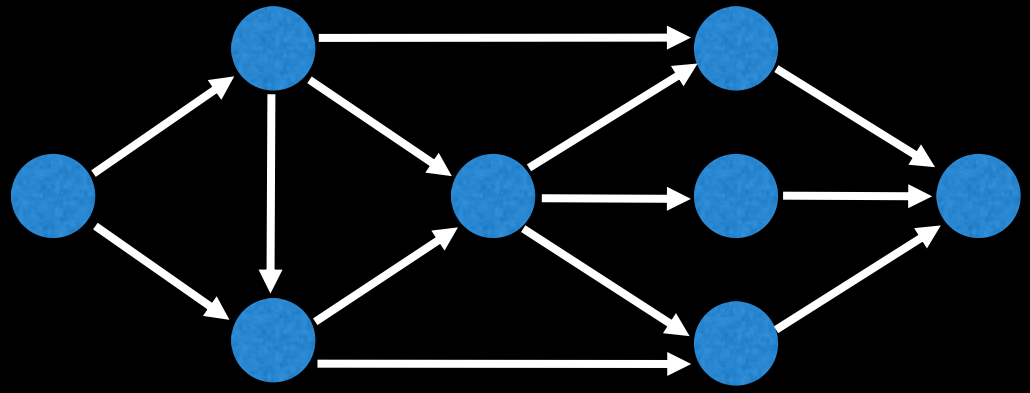
Out-tree

Directed Acyclic Graphs (DAGs)

DAGs are **directed graphs with no cycles**. These graphs play an important role in representing structures with dependencies. Several efficient algorithms exist to operates on DAGs.

Cool fact: **All out-trees are DAGs** but **not all DAGs are out-trees**.

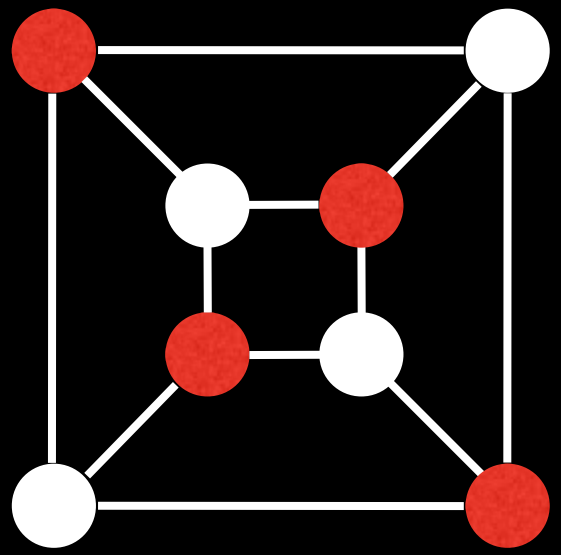


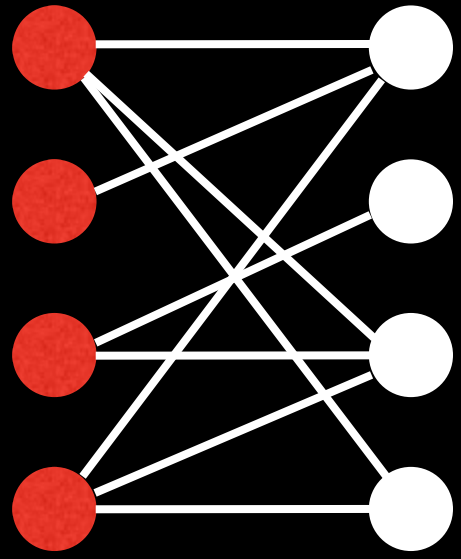


Bipartite Graph

A **bipartite graph** is one whose *vertices* can be split into two independent groups U, V such that every edge connects between U and V.

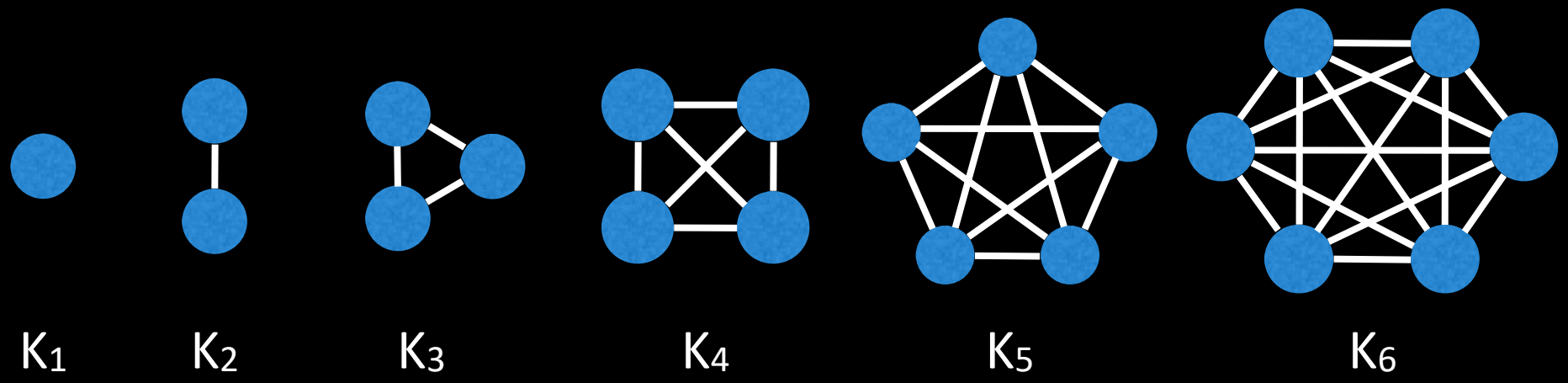
Other definitions exist such as: The graph is two colourable or there is no odd length cycle.





Complete Graphs

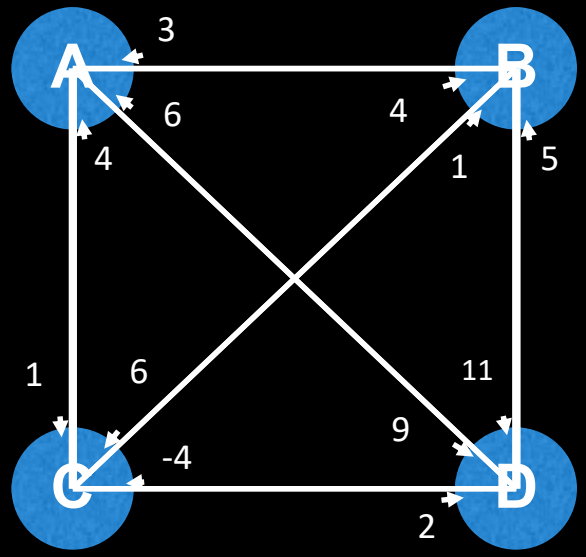
A **complete graph** is one where there is a unique edge between every pair of nodes. A complete graph with n vertices is denoted as the graph K_n.



Representing Graphs

Adjacency Matrix

A **adjacency matrix** *m* is a very simple way to represent a graph. The idea is that the cell m[i][j] represents the edge weight of going from node i to node j.



	A	B	C	D
A	0	4	1	9
B	3	0	6	11
C	4	1	0	2
D	6	5	-4	0

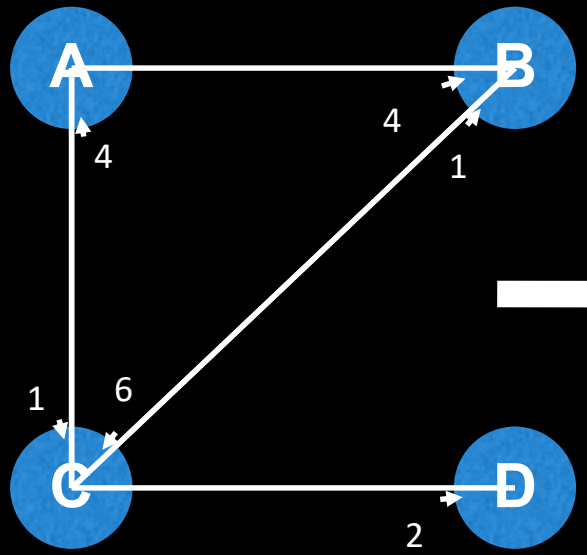
NOTE: It is often assumed that the edge of going from a node to itself has a cost of zero.

Adjacency Matrix

Pros	Cons
Space efficient for representing dense graphs	Requires $O(V^2)$ space
Edge weight lookup is $O(1)$	Iterating over all edges takes $O(V^2)$ time
Simplest graph representation	

Adjacency List

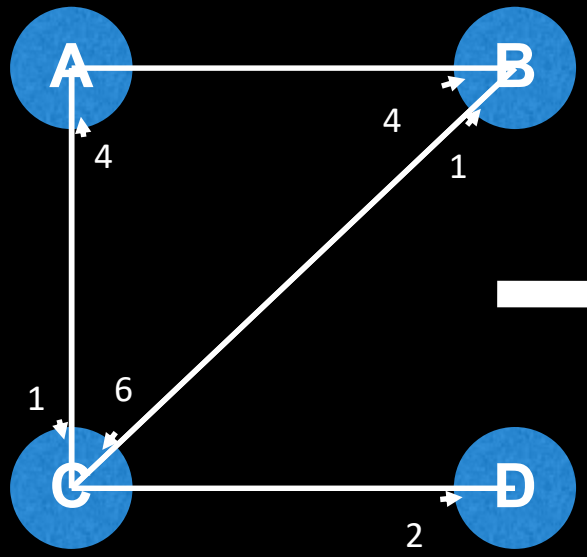
An **adjacency list** is a way to represent a graph as a **map** from nodes to lists of edges.



A -> [(B,4),(C,1)]
B -> [(C,6)]
C -> [(A,4),(B,1),(D,2)]
D -> []

Adjacency List

An **adjacency list** is a way to represent a graph as a map from nodes to lists of edges.



A -> [(B,4),(C,1)]
B -> [(C,6)]
C -> [(A,4),(B,1),(D,2)]
D -> []

Node C can reach

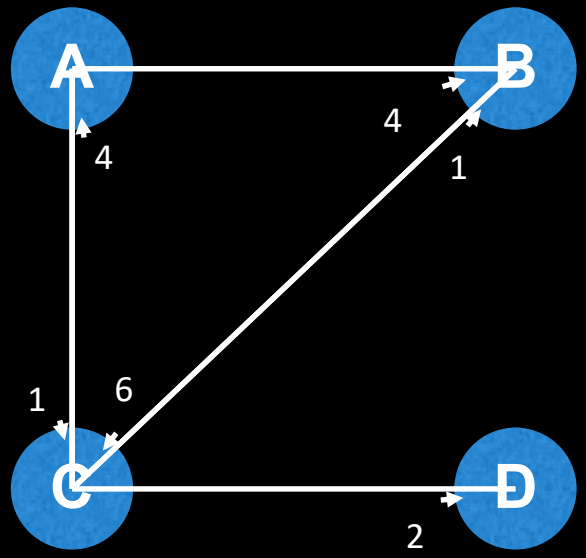
Node A with cost 4
Node B with cost 1
Node D with cost 2

Adjacency List

Pros	Cons
Space efficient for representing sparse graphs	Less space efficient for denser graphs.
Iterating over all edges is efficient	Edge weight lookup is **O(E)**
	Slightly more complex graph representation

Edge List

An **edge list** is a way to represent a graph simply as an unordered list of edges. Assume the notation for any triplet (u,v,w) means: “the cost from node u to node v is w”



[(C,A,4), (A,C,1),
(B,C,6), (A,B,4),
(C,B,1), (C,D,2)]

This representation is seldom used because of its lack of structure. However, it is conceptually simple and practical in a handful of algorithms.

Edge List

Pros	Cons
Space efficient for representing sparse graphs	Less space efficient for denser graphs.
Iterating over all edges is efficient	Edge weight lookup is **O(E)**
Very simple structure	

Common Graph Theory Problems

William Fiset

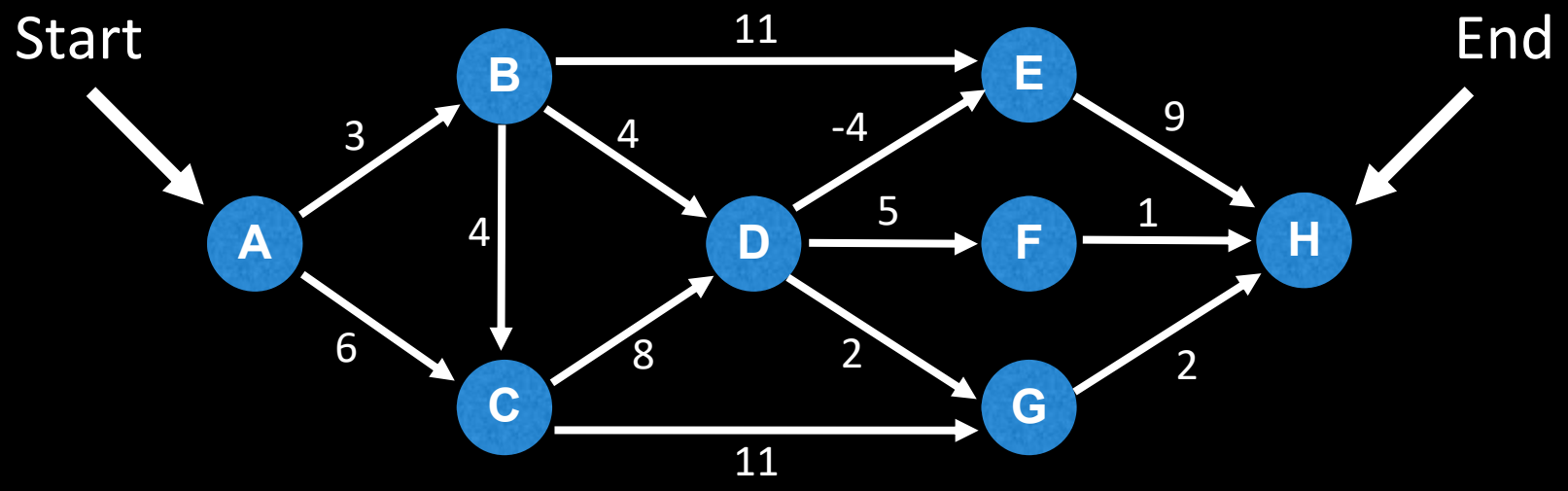
Common Graph Theory Problems

For the upcoming problems ask yourself:

- Is the graph directed or undirected?
- Are the edges of the graph weighted?
- Is the graph I will encounter likely to be sparse or dense with edges?
- Should I use an adjacency matrix, adjacency list, an edge list or other structure to represent the graph efficiently?

Shortest Path Problem

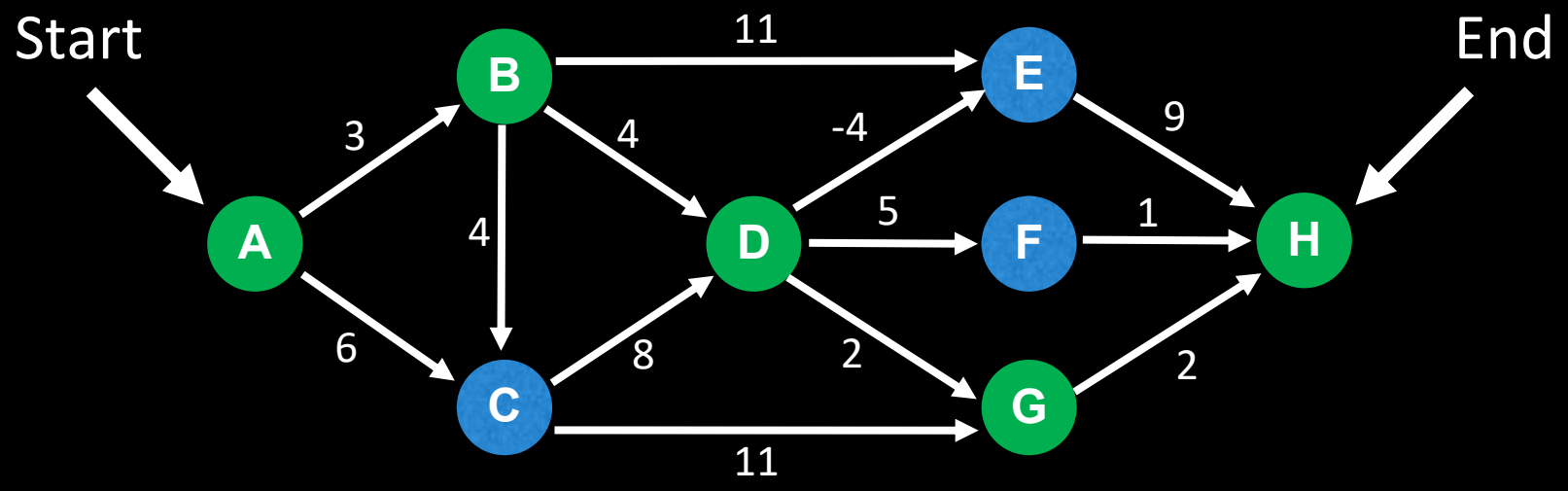
Given a weighted graph, find the shortest path of edges from node A to node B.



Algorithms: BFS (unweighted graph), Dijkstra's, Bellman-Ford, Floyd-Warshall, A*, and many more.

Shortest Path Problem

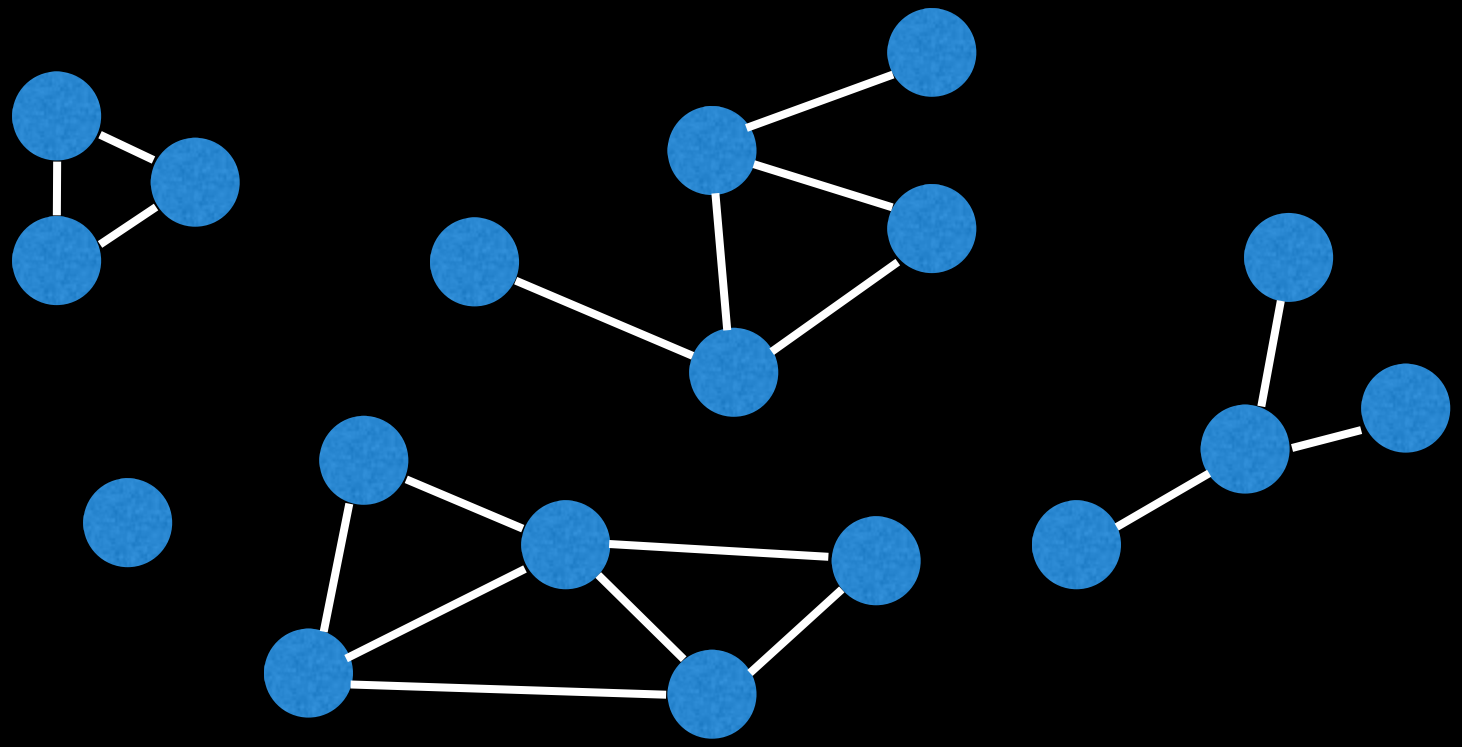
Given a weighted graph, find the shortest path of edges from node A to node B.



Algorithms: BFS (unweighted graph), Dijkstra's, Bellman-Ford, Floyd-Warshall, A*, and many more.

Connectivity

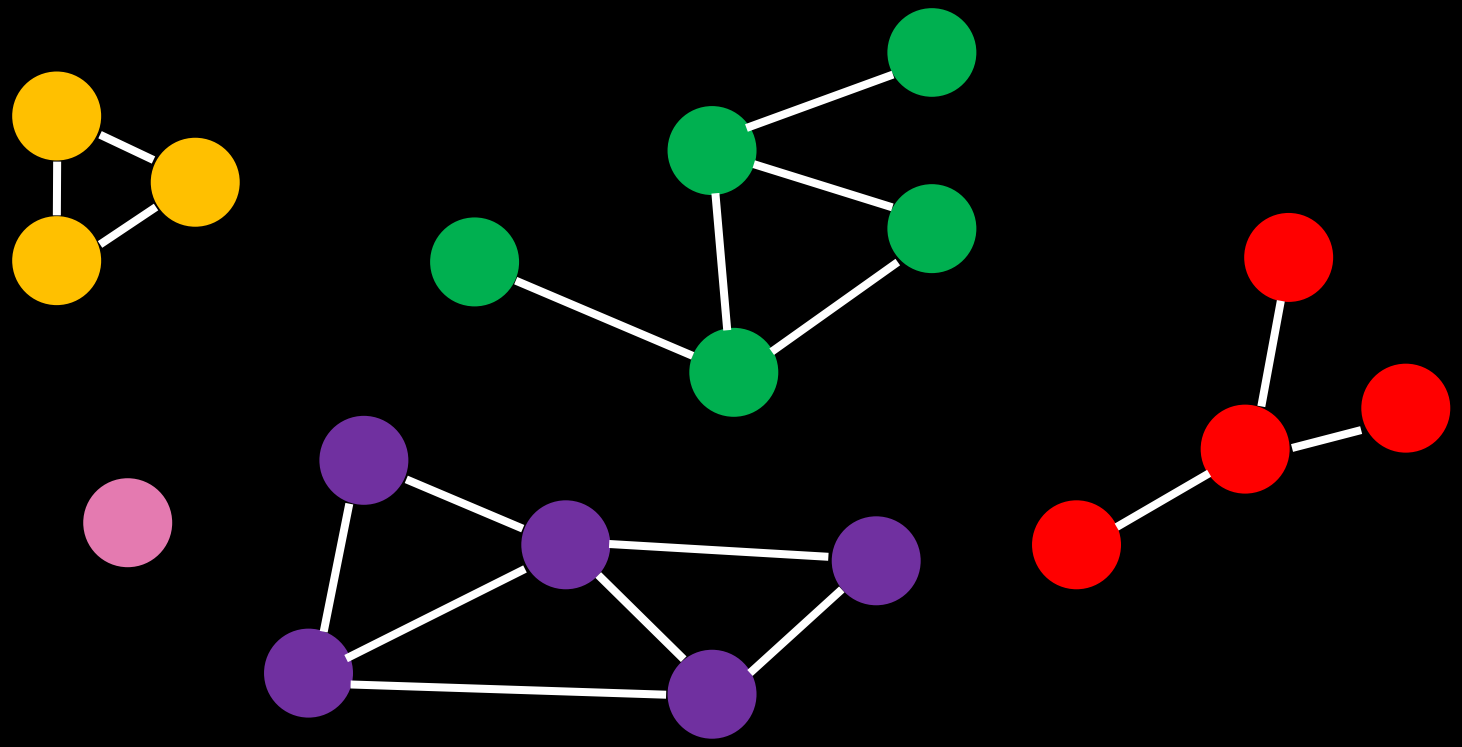
Does there exist a path between node A and node B?



Typical solution: Use union find data structure or any search algorithm (e.g DFS).

Connectivity

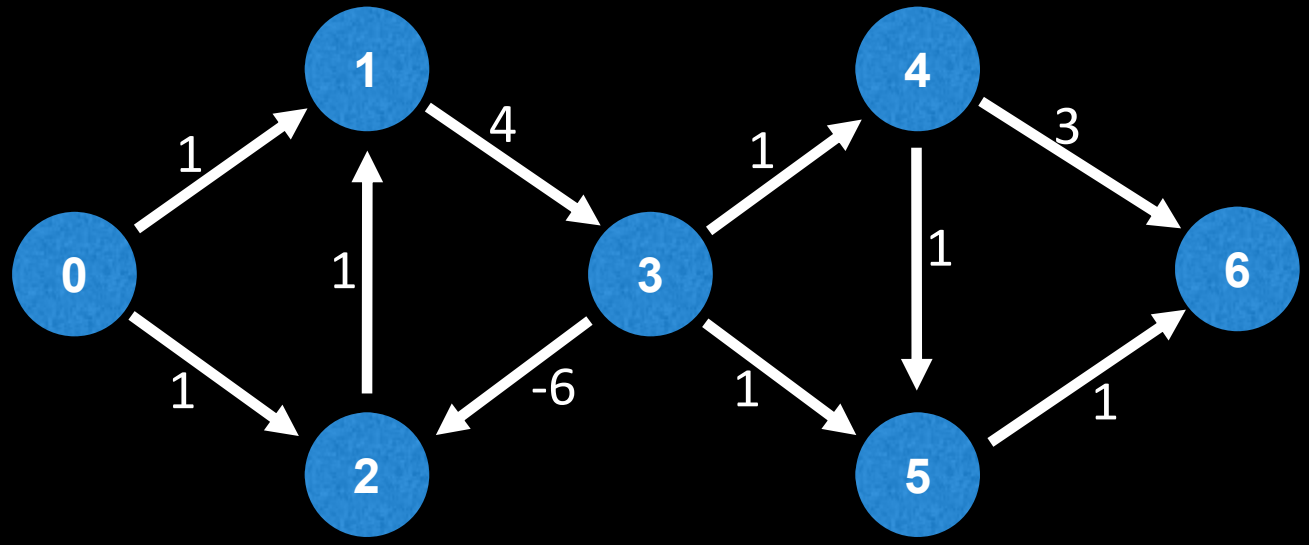
Does there exist a path between node A and node B?



Typical solution: Use union find data structure or any search algorithm (e.g DFS).

Negative cycles

Does my weighted digraph have any negative cycles? If so, where?



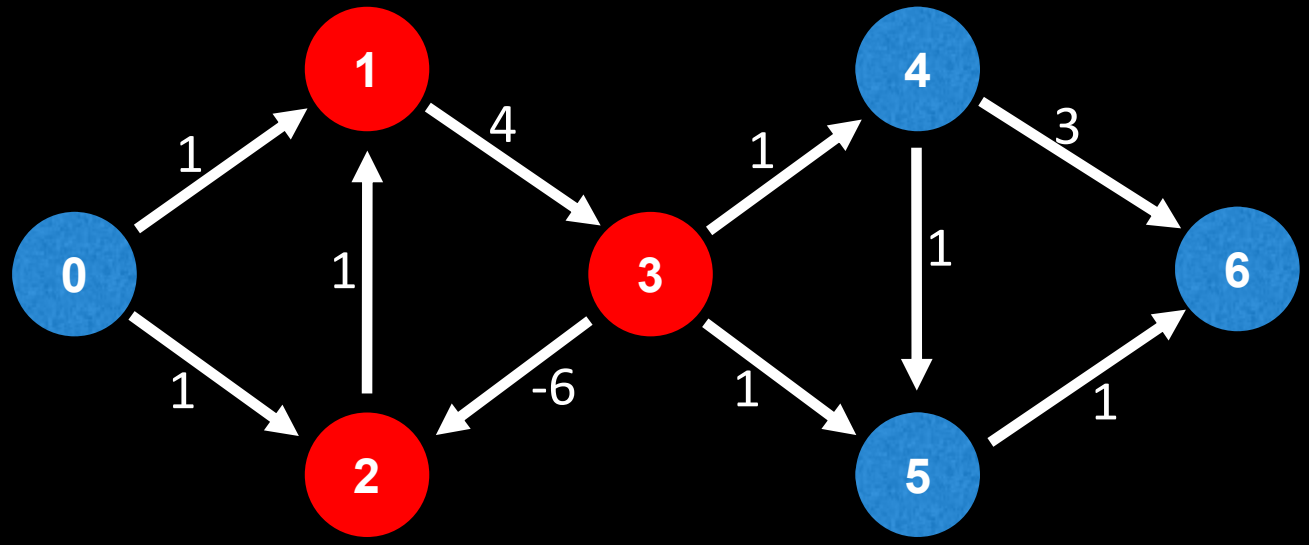
Directly in negative cycle

Unaffected node

Algorithms: Bellman-Ford and Floyd-Warshall

Negative cycles

Does my weighted digraph have any negative cycles? If so, where?



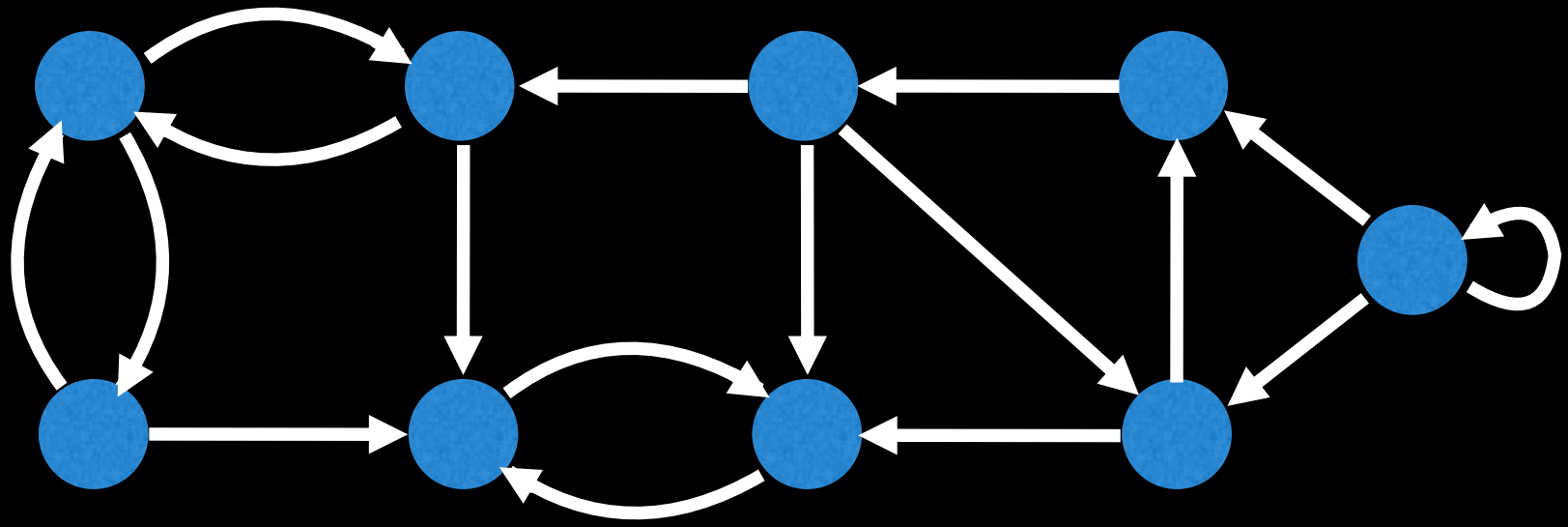
Directly in negative cycle

Unaffected node

Algorithms: Bellman-Ford and Floyd-Warshall

Strongly Connected Components

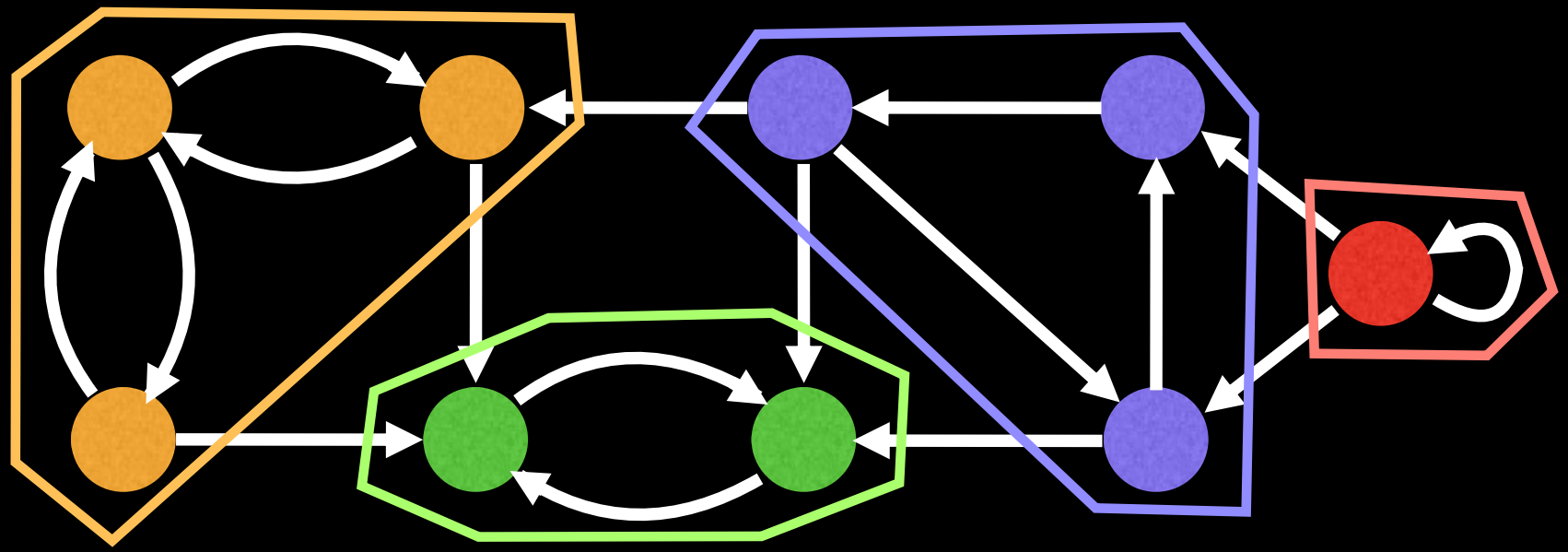
Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.



Algorithms: Tarjan’s and Kosaraju's algorithm

Strongly Connected Components

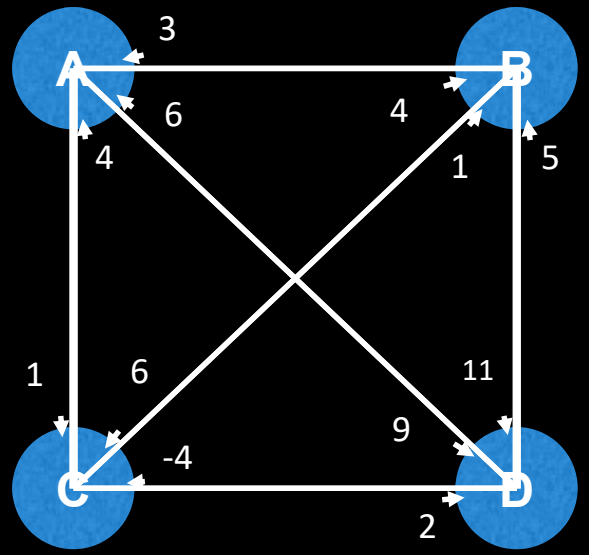
Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.



Algorithms: Tarjan’s and Kosaraju's algorithm

Traveling Salesman Problem

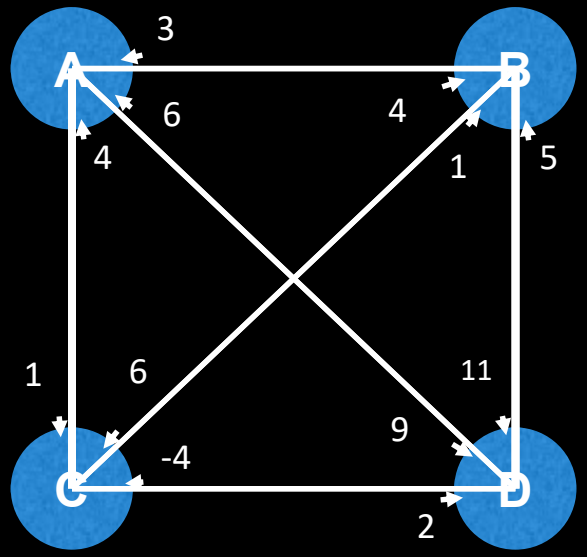
"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" - Wiki

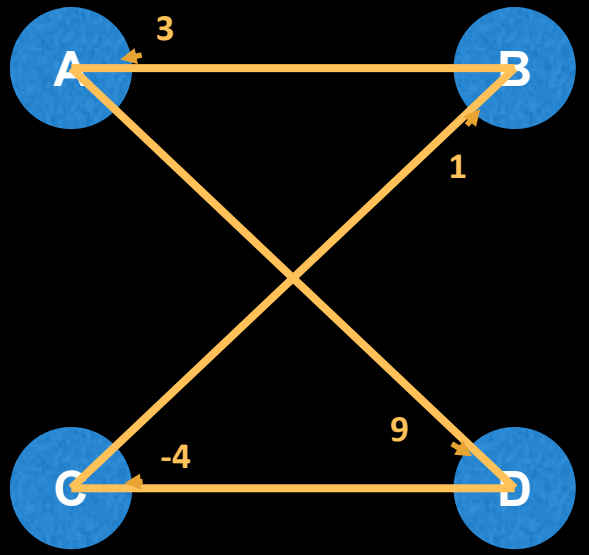


Algorithms: Held-Karp, branch and bound and many approximation algorithms

Traveling Salesman Problem

The TSP problem is NP-Hard meaning it's a very computationally challenging problem. This is unfortunate because the TSP has several very important applications.

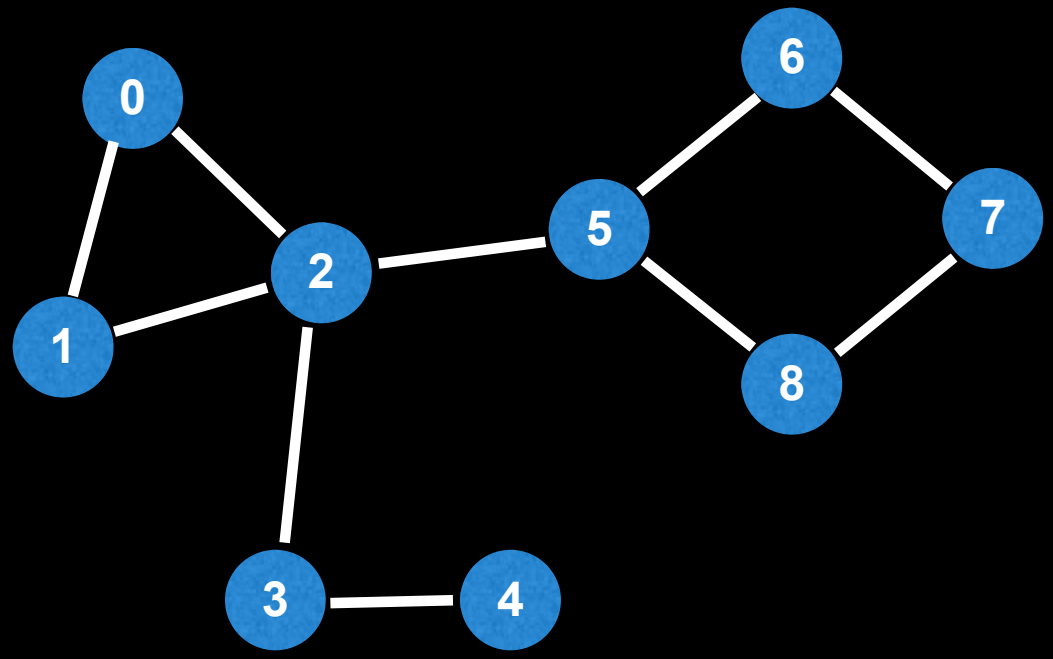




Algorithms: Held-Karp (DP), branch and bound and many approximation algorithms

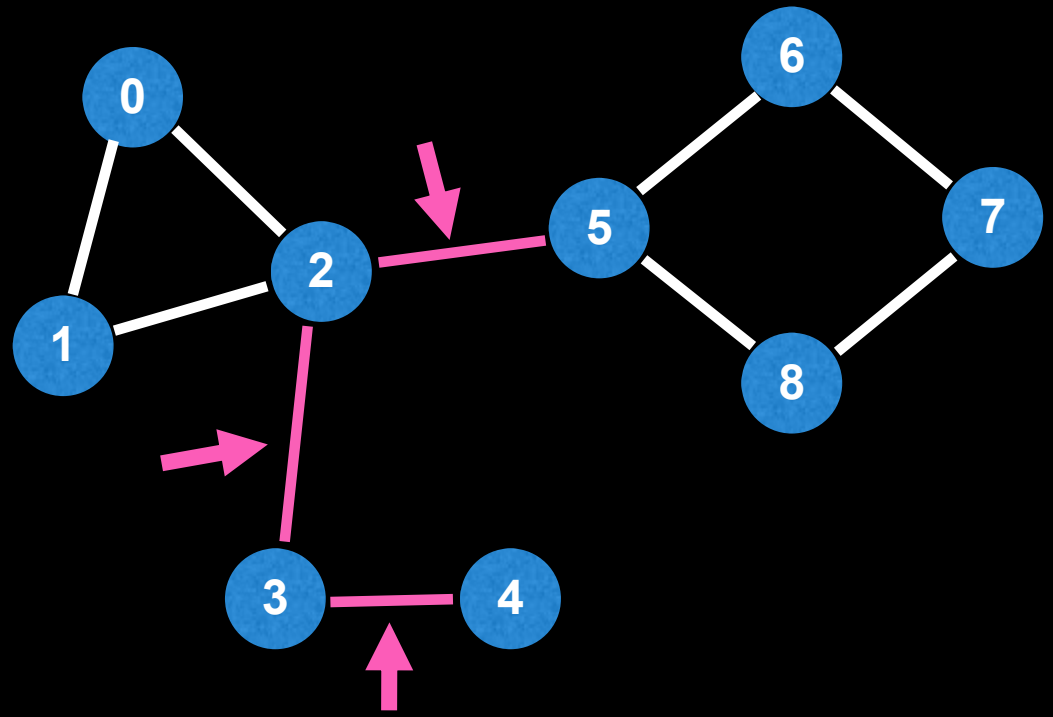
Bridges

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.



Bridges

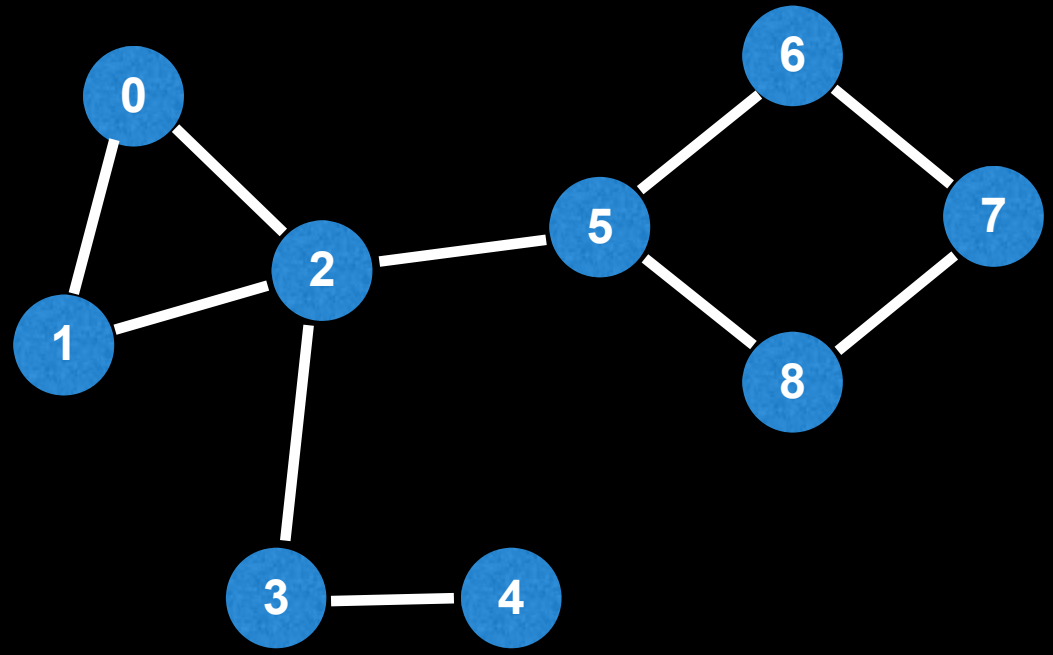
A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.



Bridges are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph.

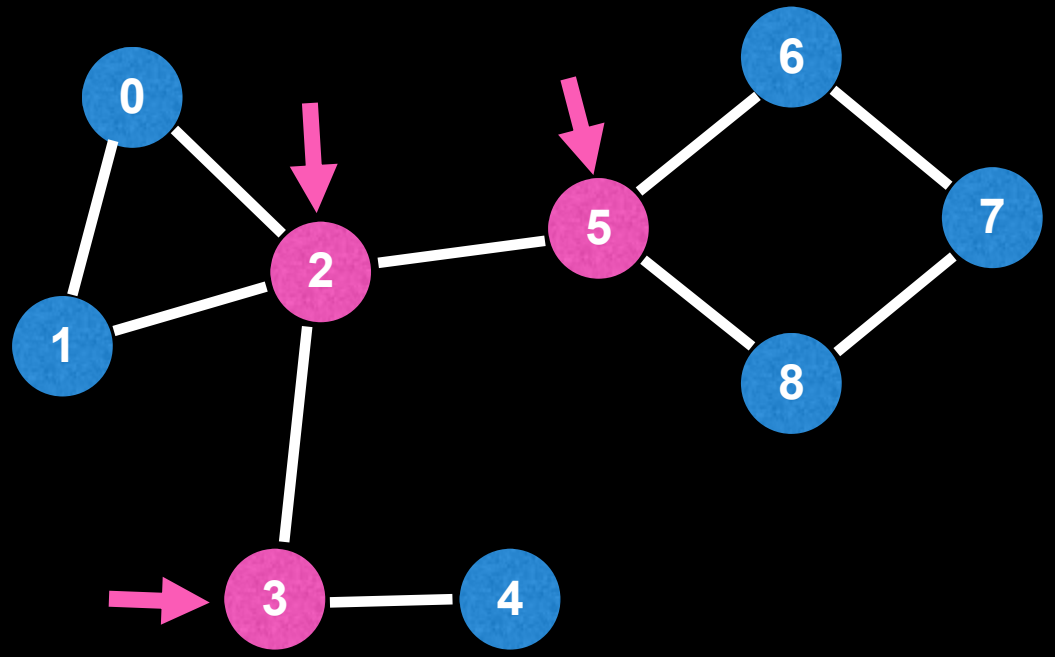
Articulation points

An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.



Articulation points

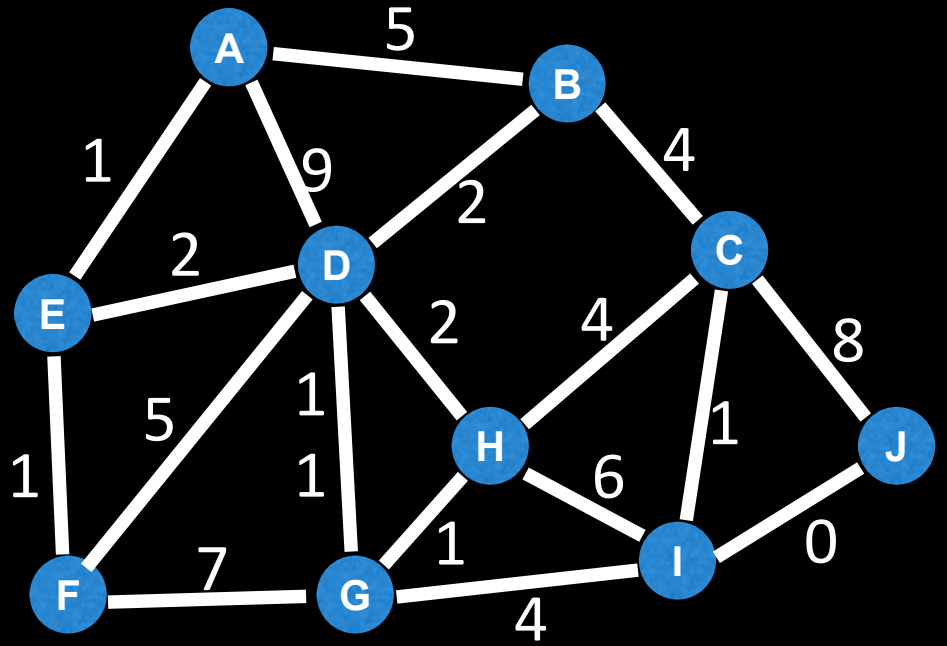
An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.



Articulation points are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph.

Minimum Spanning Tree (MST)

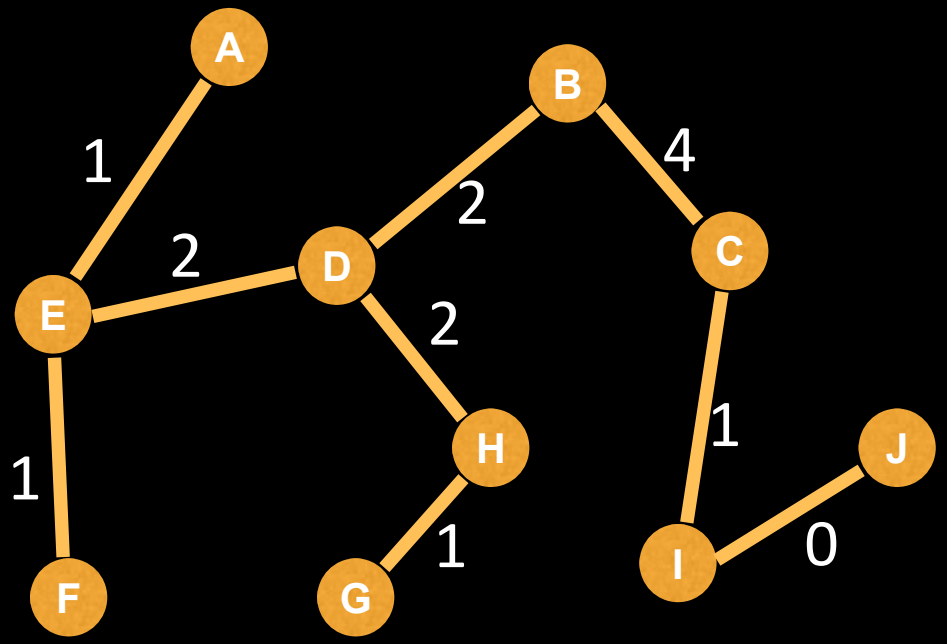
A **minimum spanning tree (MST)** is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. - Wiki



Algorithms: Kruskal’s, Prim’s & Borůvka's algorithm

Minimum Spanning Tree (MST)

A **minimum spanning tree (MST)** is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. - Wiki

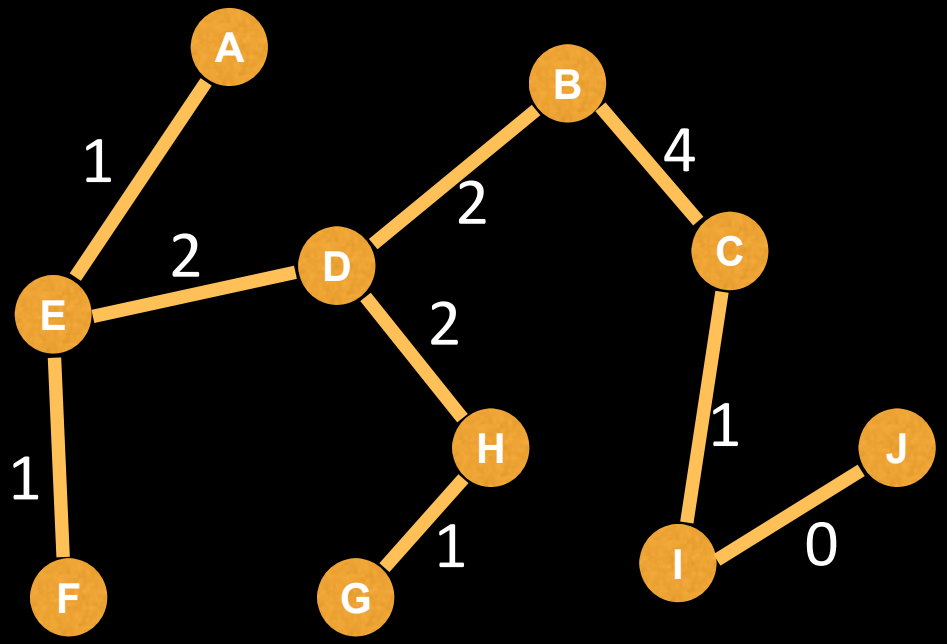


This MST has a total weight of 14. Note that MSTs on a graph are not always unique.

Algorithms: Kruskal’s, Prim’s & Borůvka's algorithm

Minimum Spanning Tree (MST)

MSTs are seen in many applications including:
Designing a least cost network, circuit design, transportation networks, and etc…

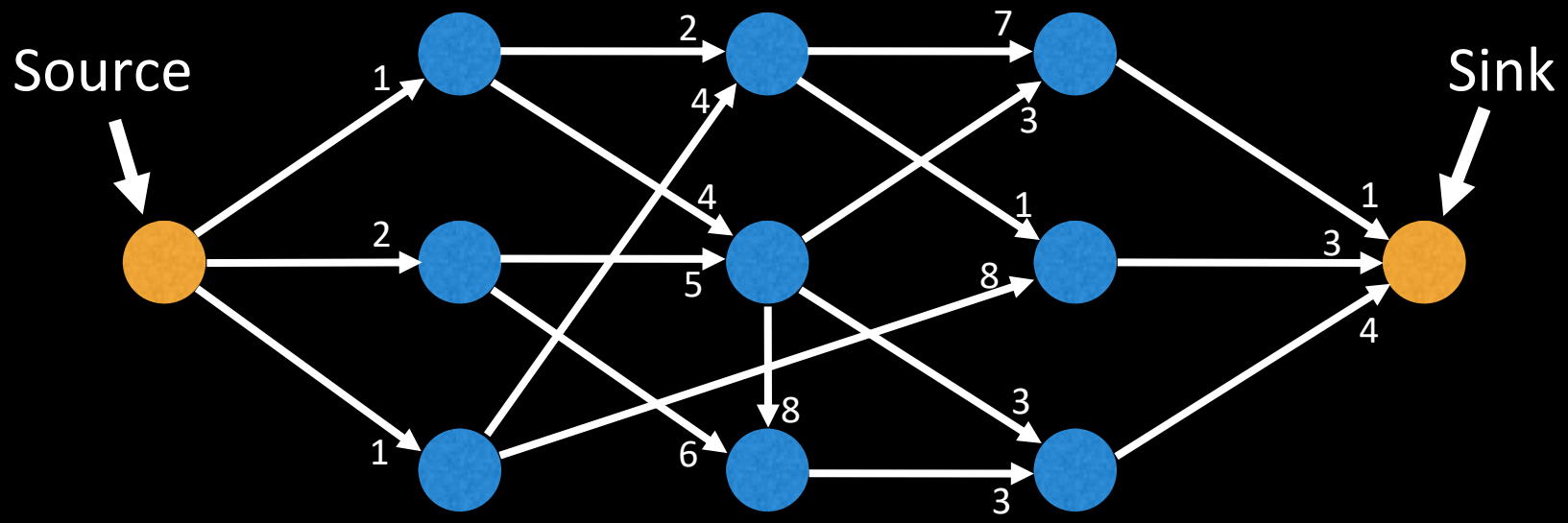


This MST has a total weight of 14. Note that MSTs on a graph are not always unique.

Algorithms: Kruskal’s, Prim’s & Borůvka's algorithm

Network flow: max flow

Q: With an infinite input source how much "flow" can we push through the network?



Suppose the edges are roads with cars, pipes with water or hallways with packed with people. Flow represents the volume of water allowed to flow through the pipes, the number of cars the roads can sustain in traffic and the maximum amount of people that can navigate through the hallways.

Algorithms: Ford-Fulkerson, Edmonds-Karp & Dinic's algorithm

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

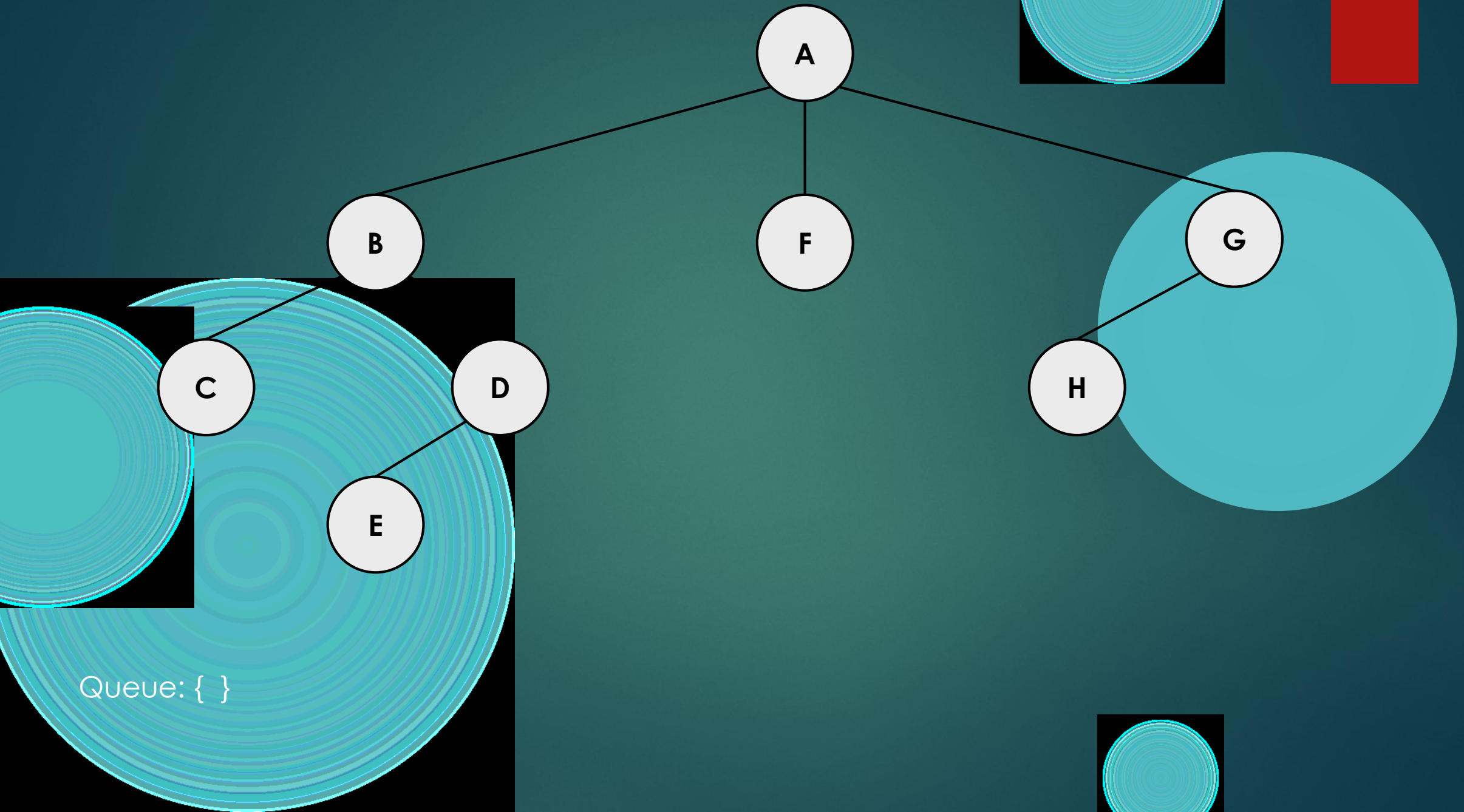
Breadth-first search

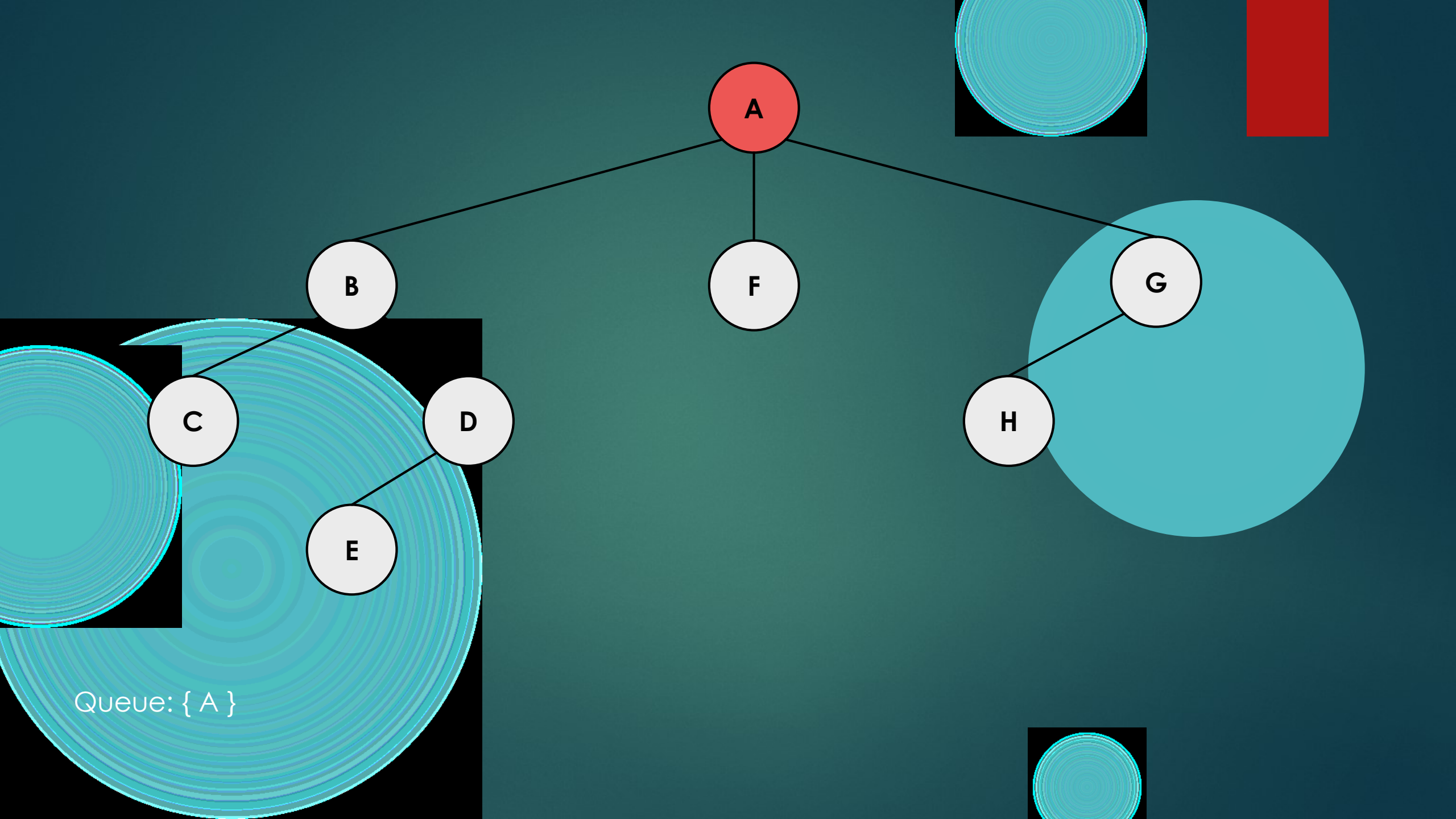
```
bfs(vertex)

    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)
```

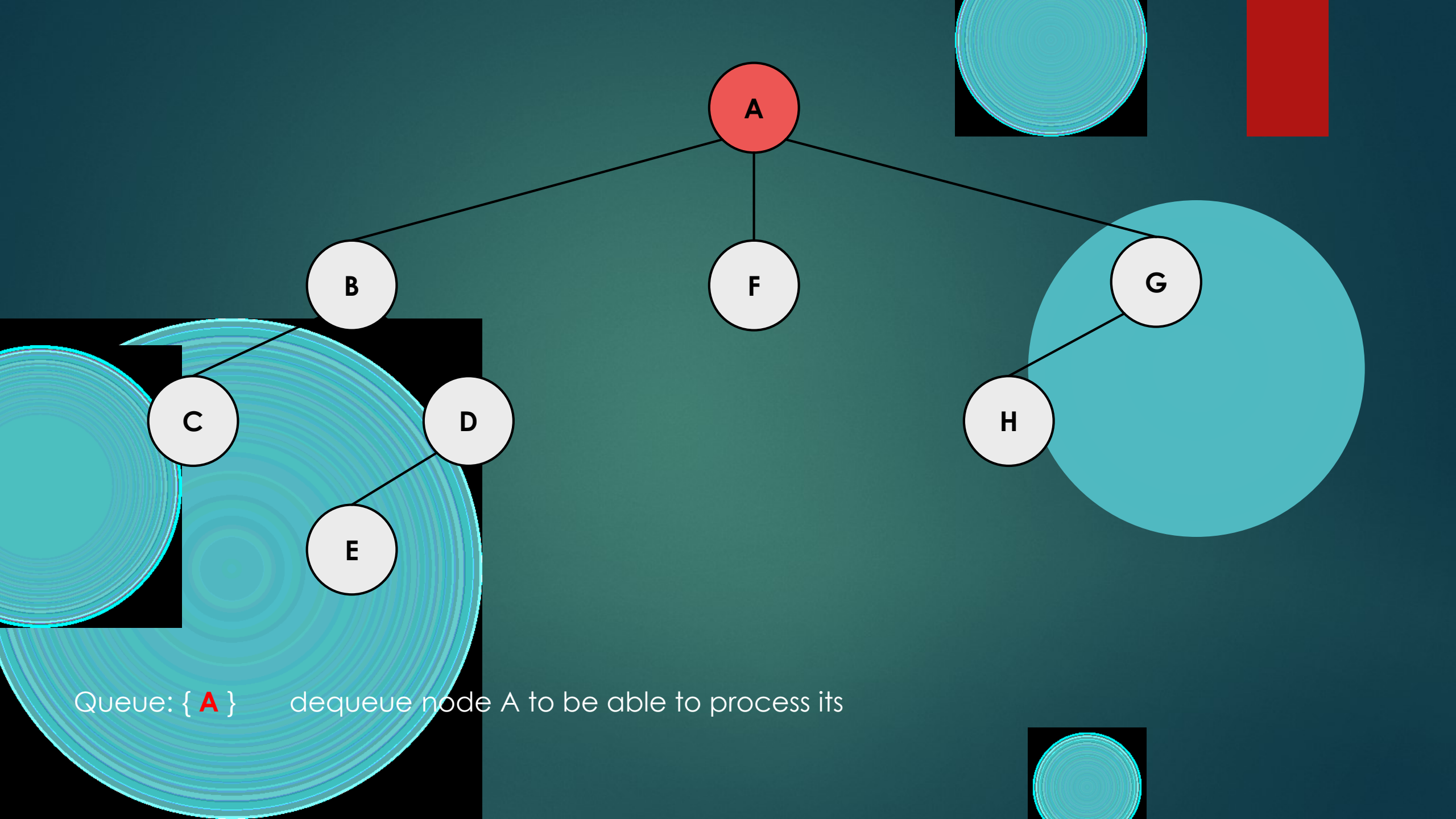
We have an empty queue at the beginning and we keep checking whether we have visited the given node or not

~ keep iterating until queue is not empty

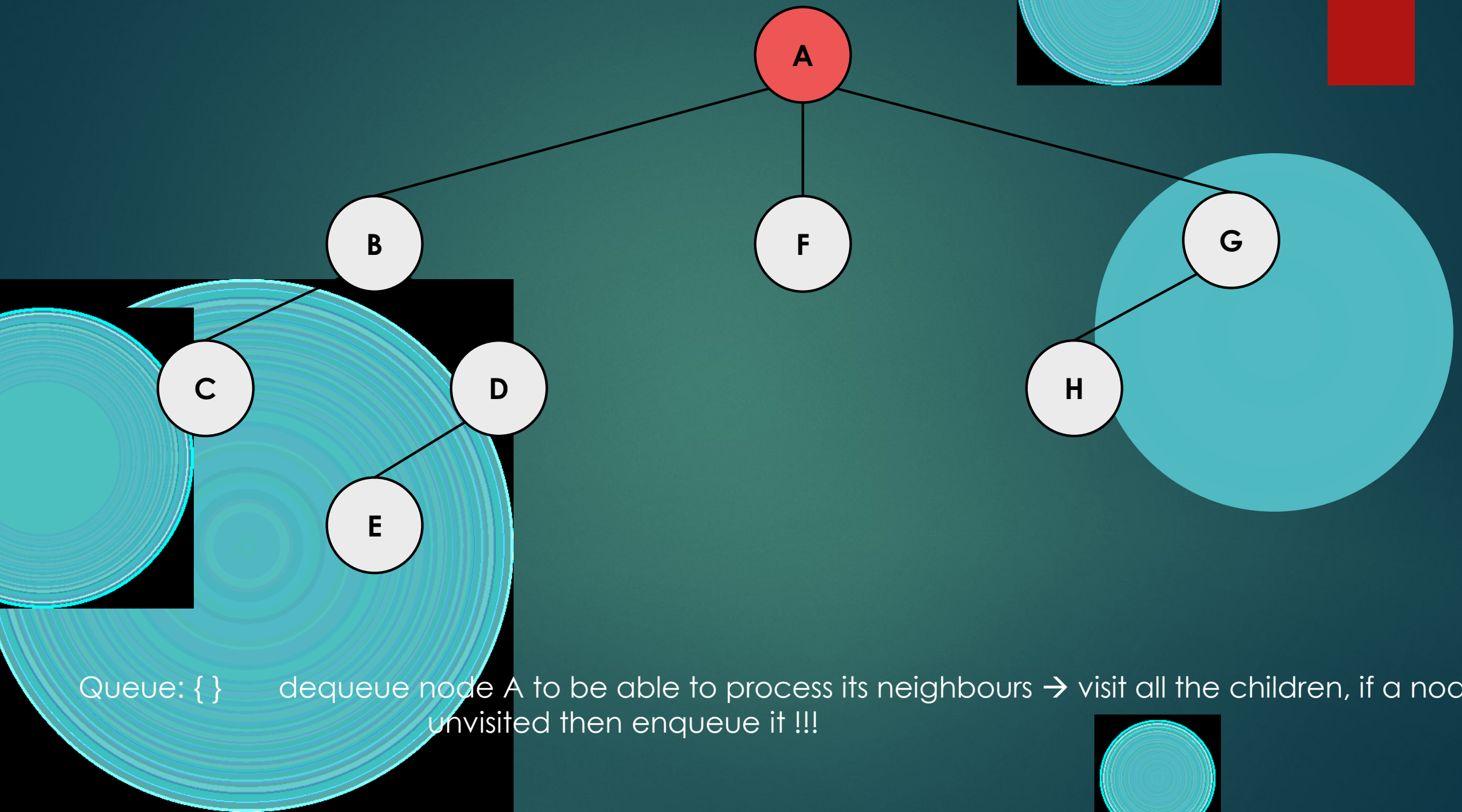
ITERATION


A
B
F
G
C
D
H
E
Queue: { }

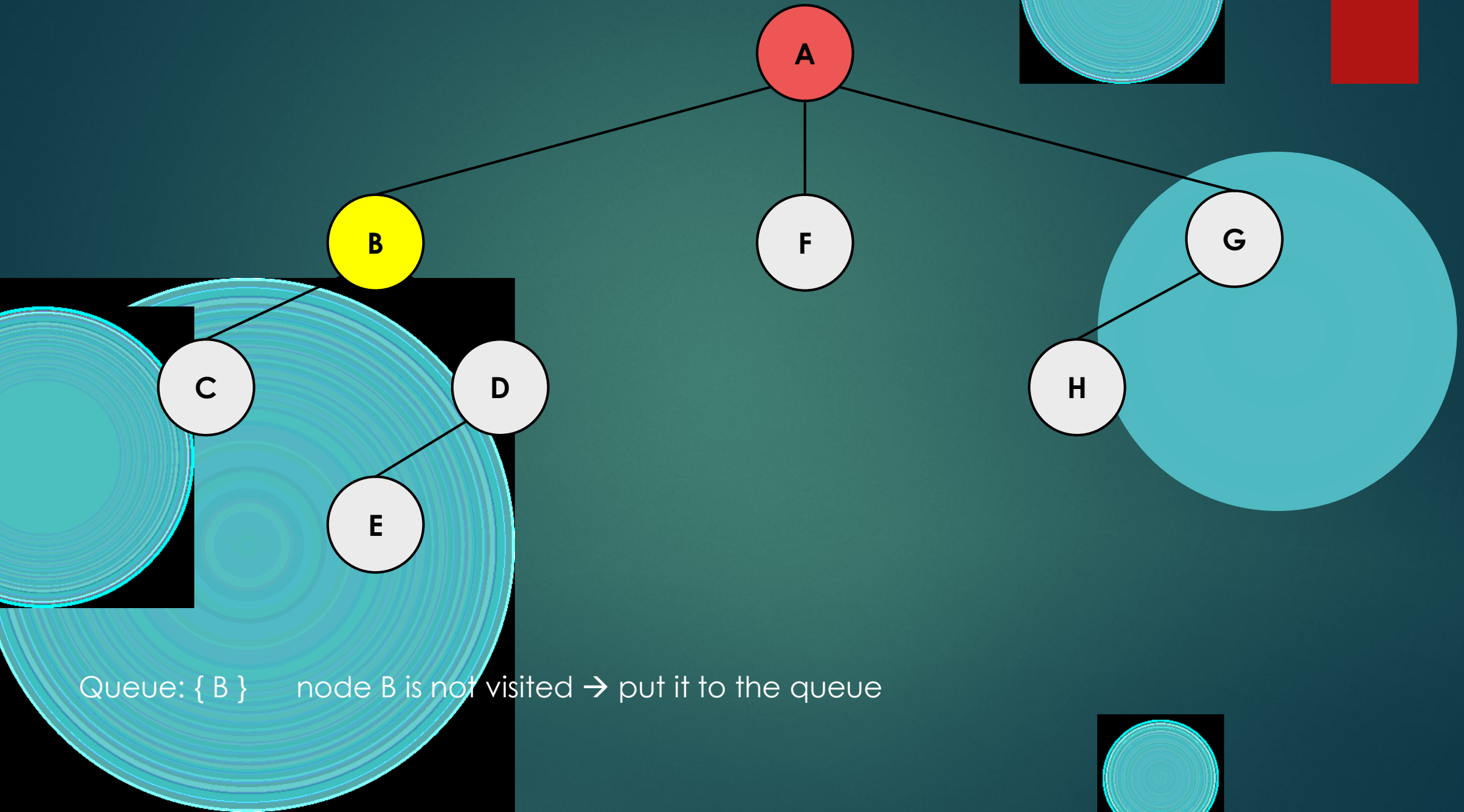

A
B
F
G
C
D
H
E
Queue: { A }



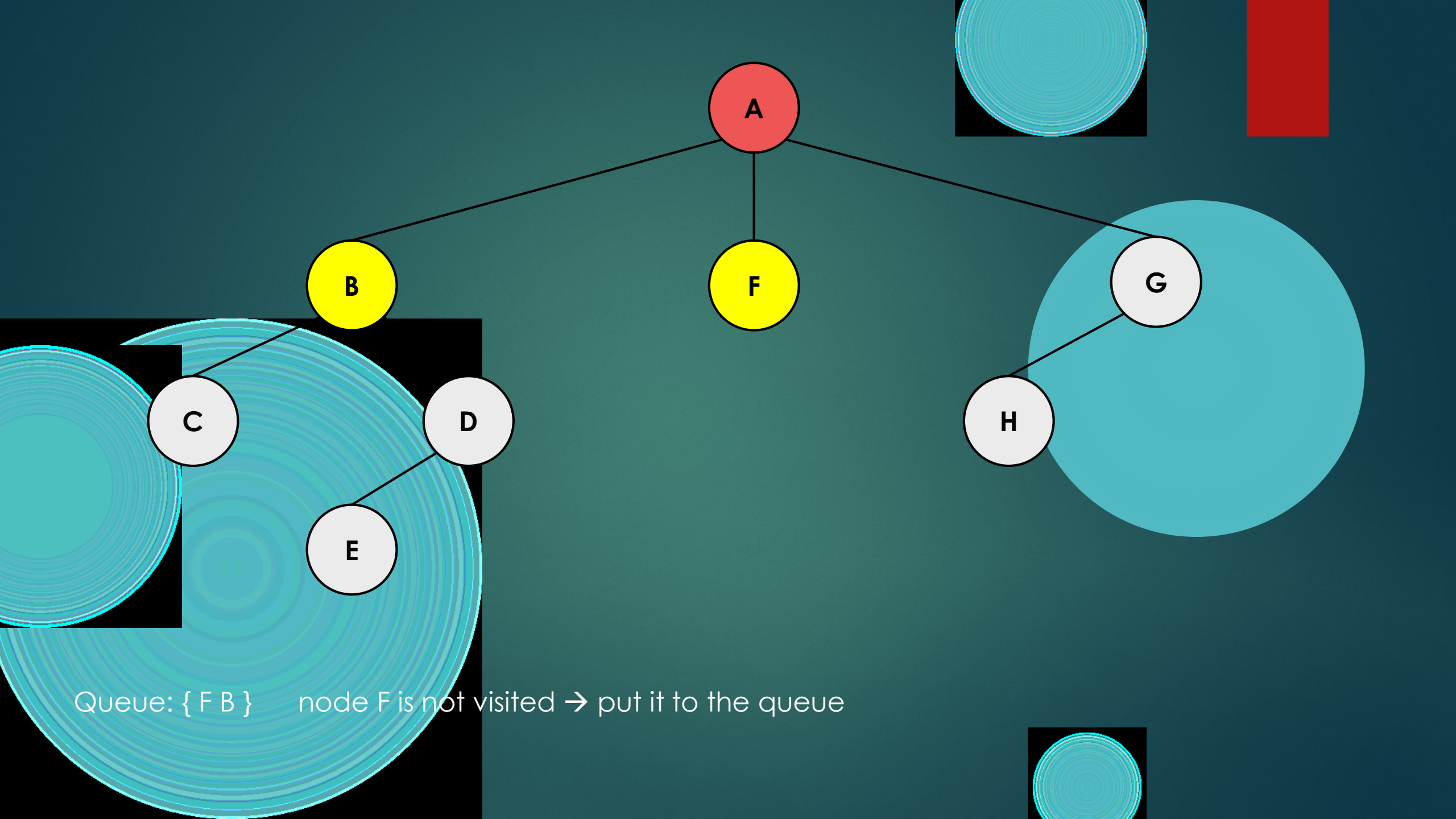
Queue: { **A** } dequeue node A to be able to process its



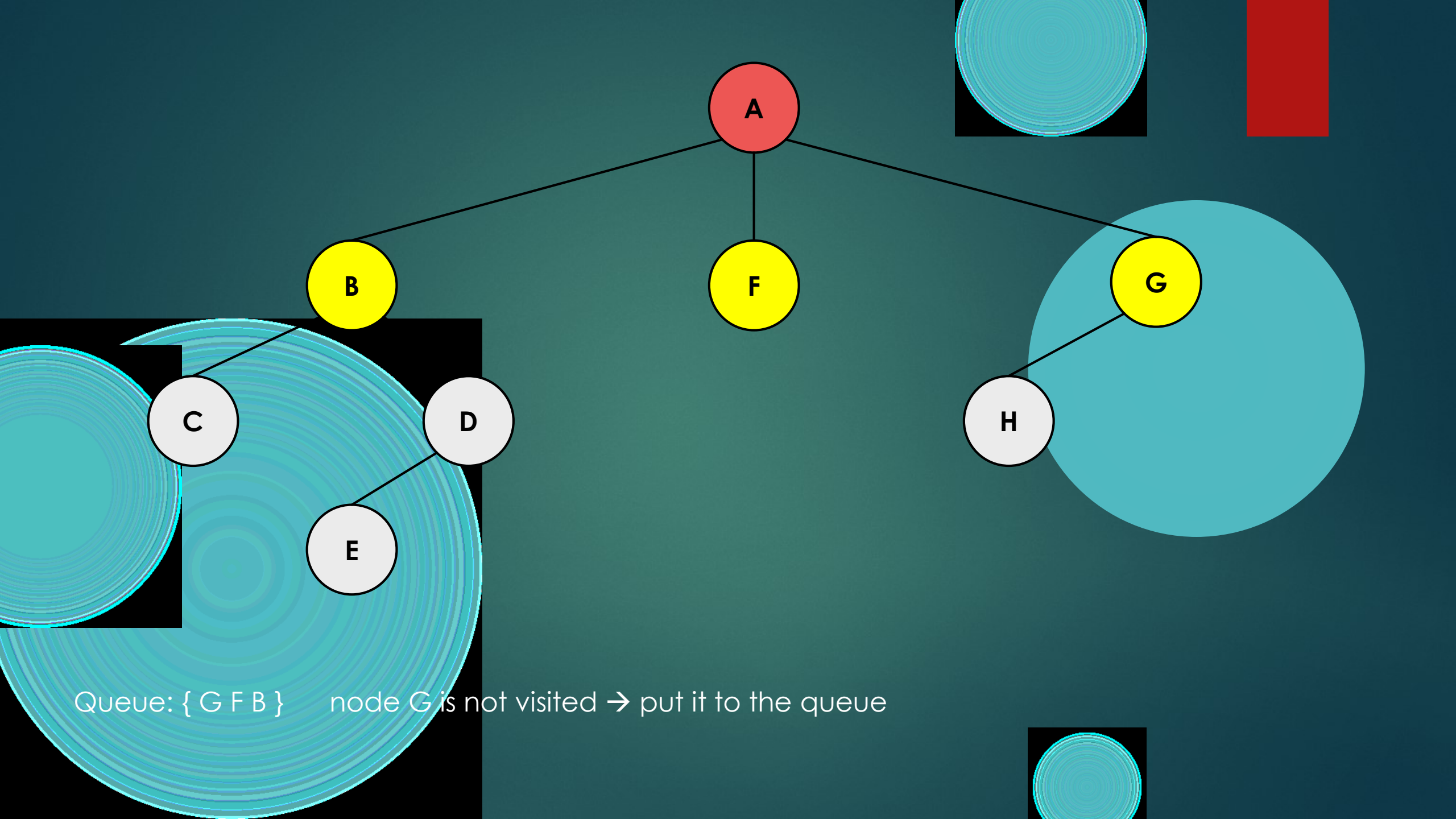
Queue: { } dequeue node A to be able to process its neighbours → visit all the children, if a node is unvisited then enqueue it !!!



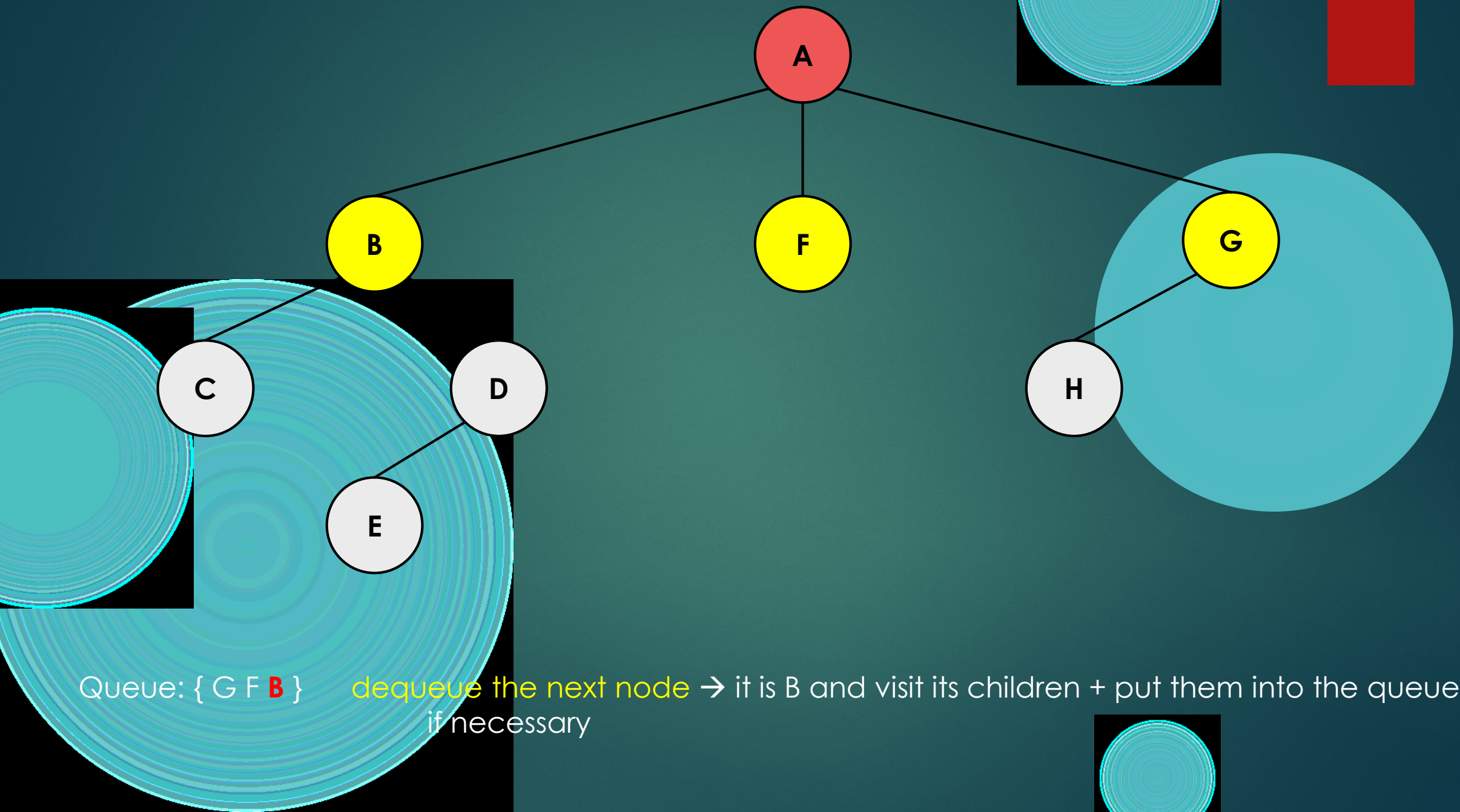
Queue: { B } node B is not visited → put it to the queue



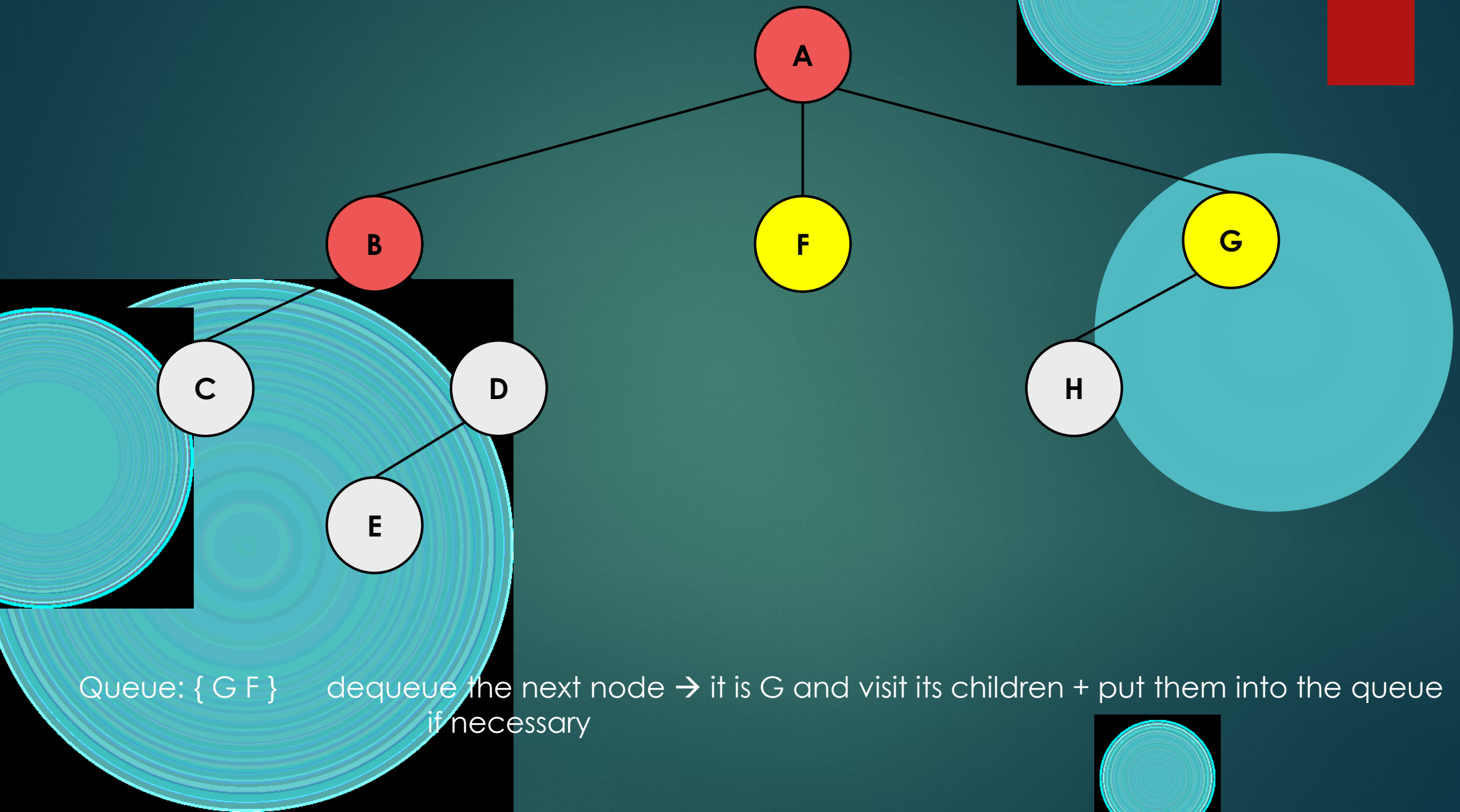
Queue: { F B } node F is not visited → put it to the queue



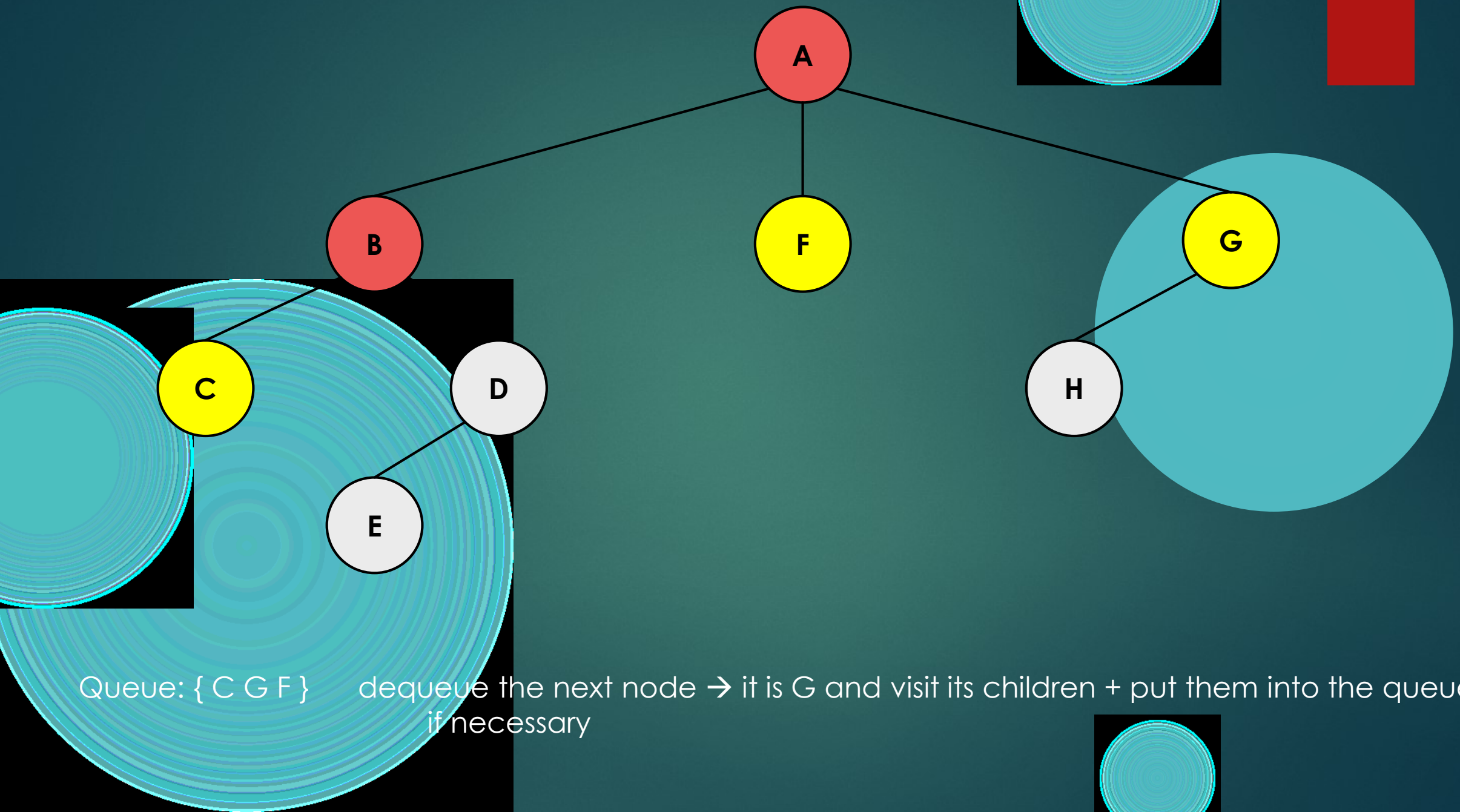
Queue: { G F B } node G is not visited → put it to the queue



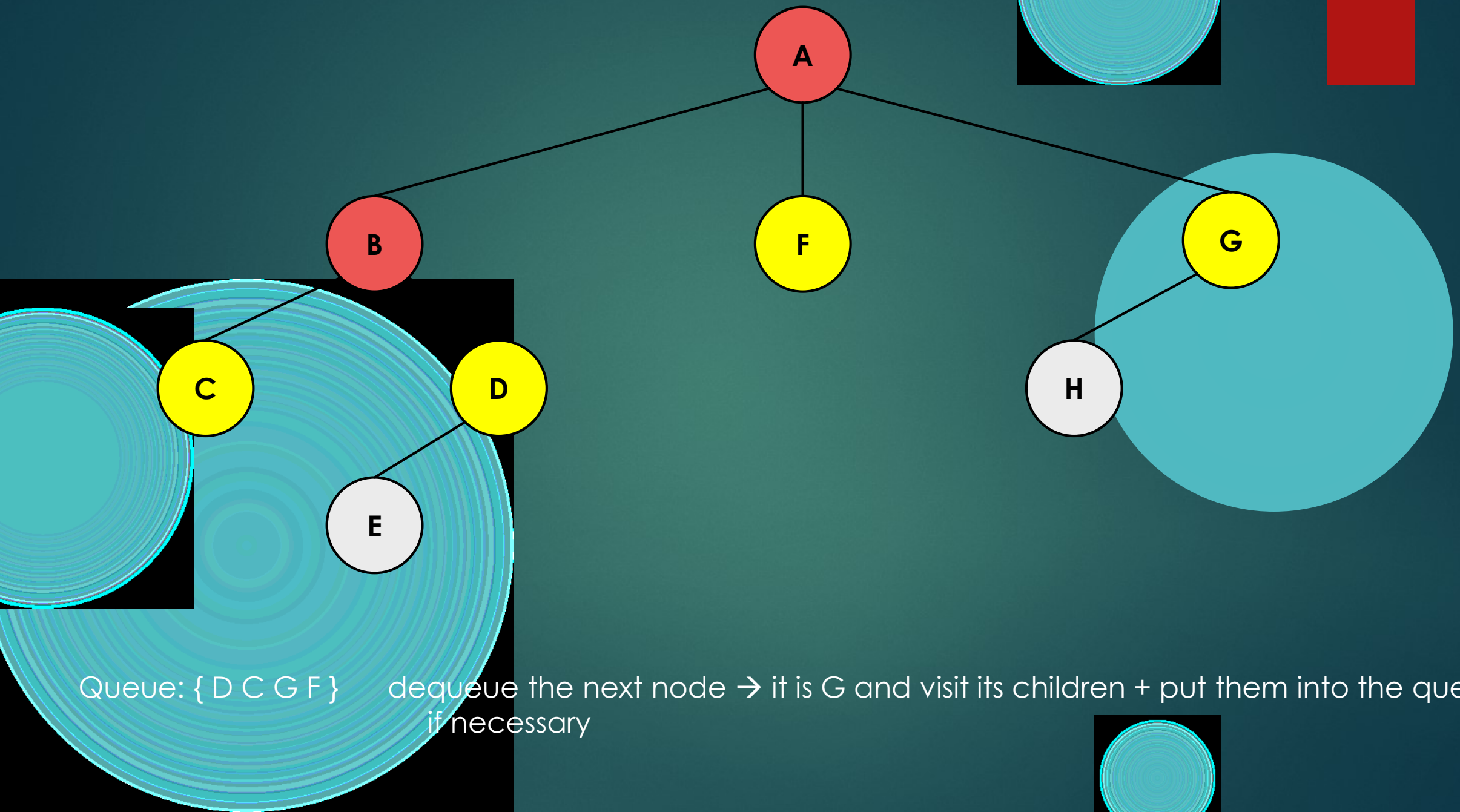
Queue: { G F **B** } dequeue the next node → it is B and visit its children + put them into the queue if necessary



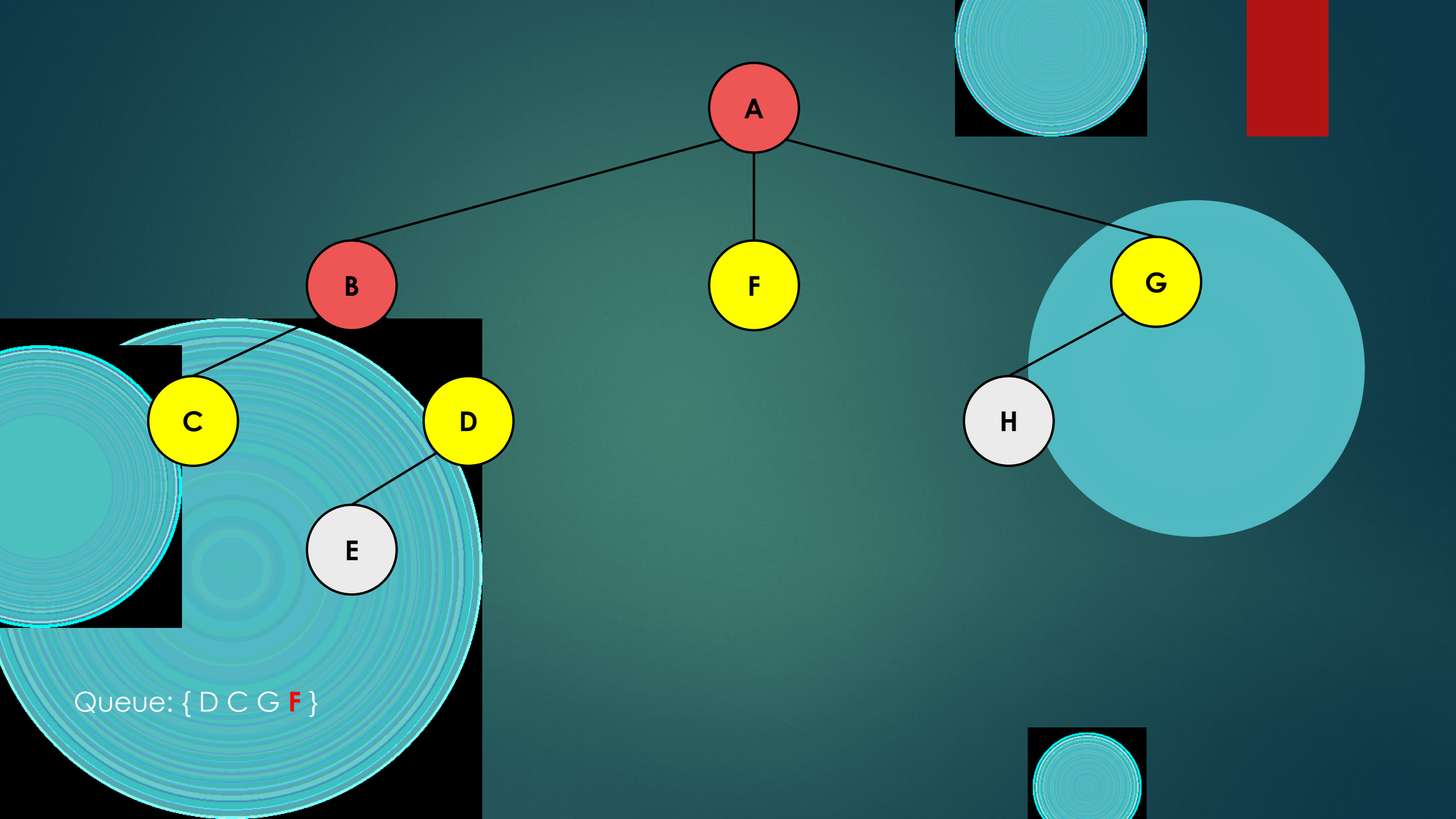
Queue: { G F } dequeue the next node → it is G and visit its children + put them into the queue if necessary

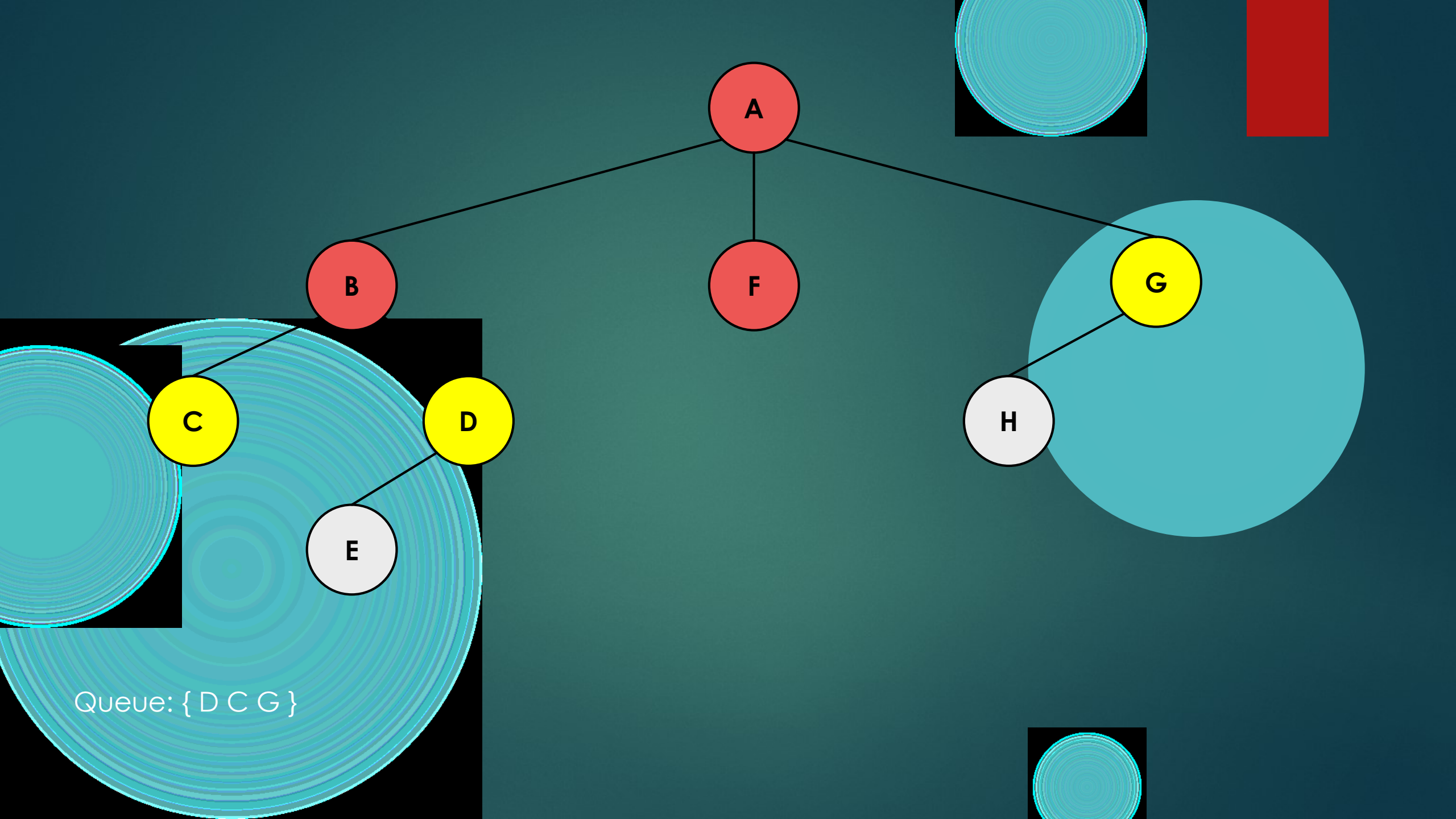


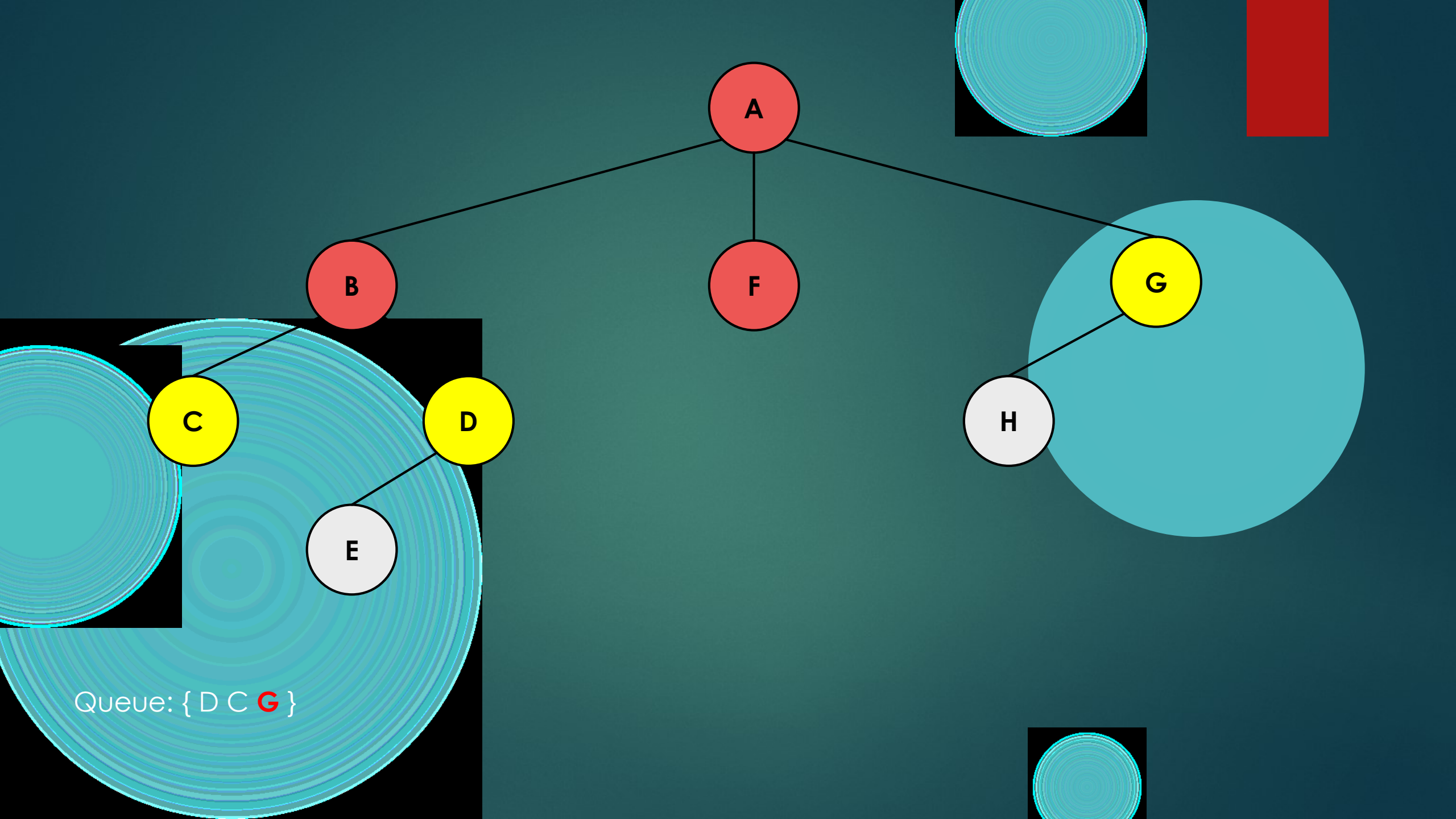
Queue: { C G F } dequeue the next node → it is G and visit its children + put them into the queue if necessary

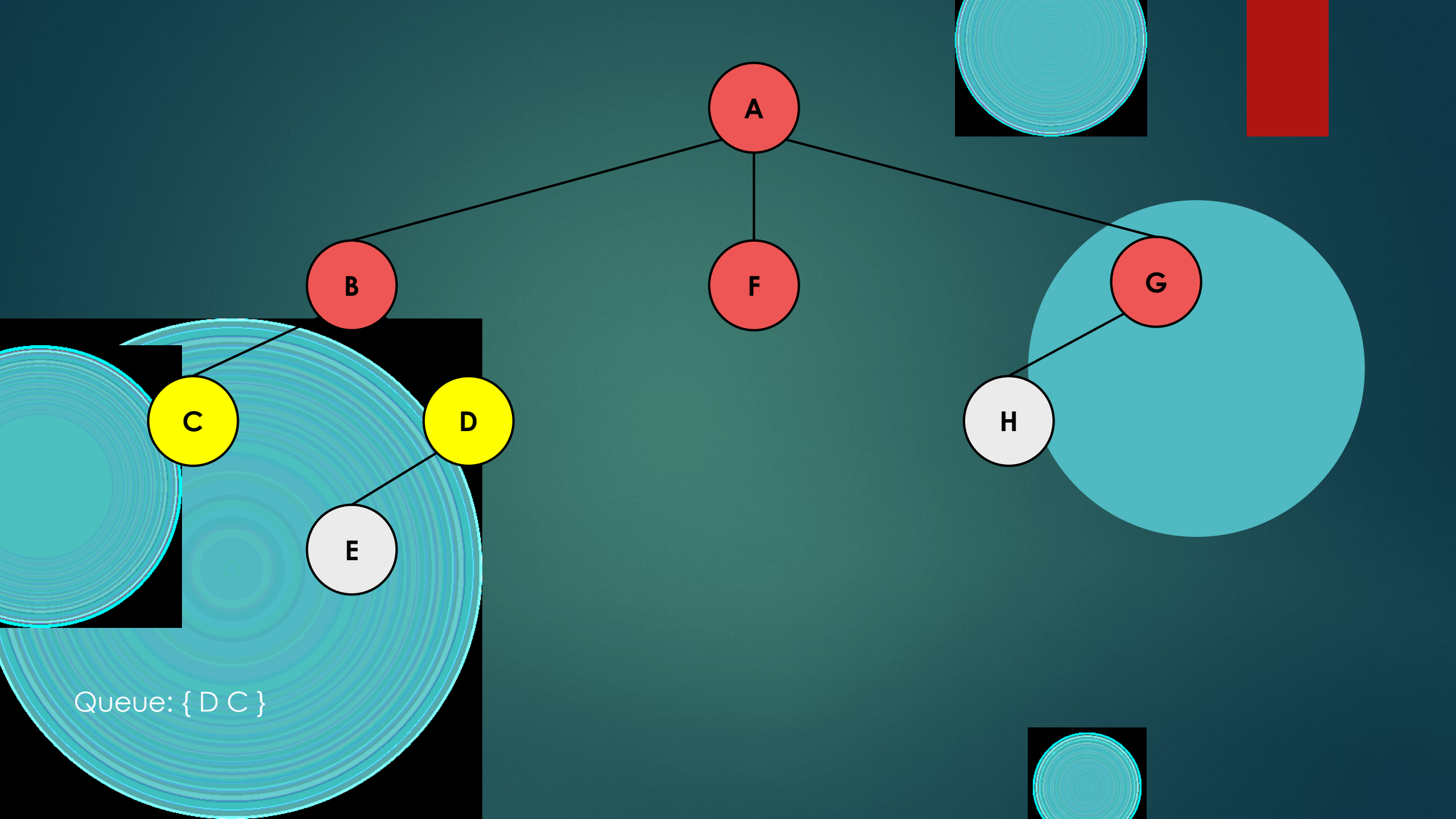


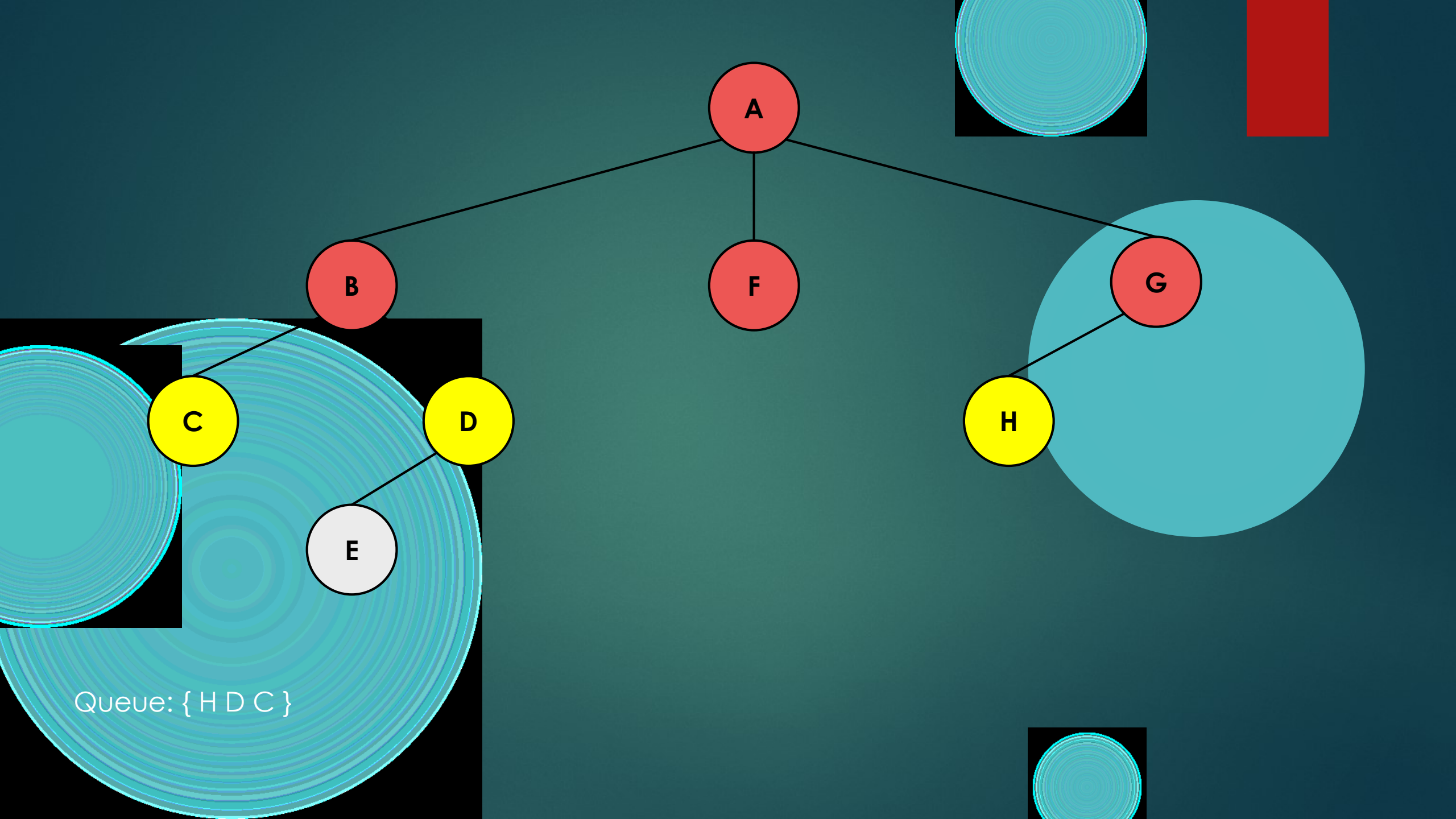
Queue: { D C G F } dequeue the next node → it is G and visit its children + put them into the queue if necessary

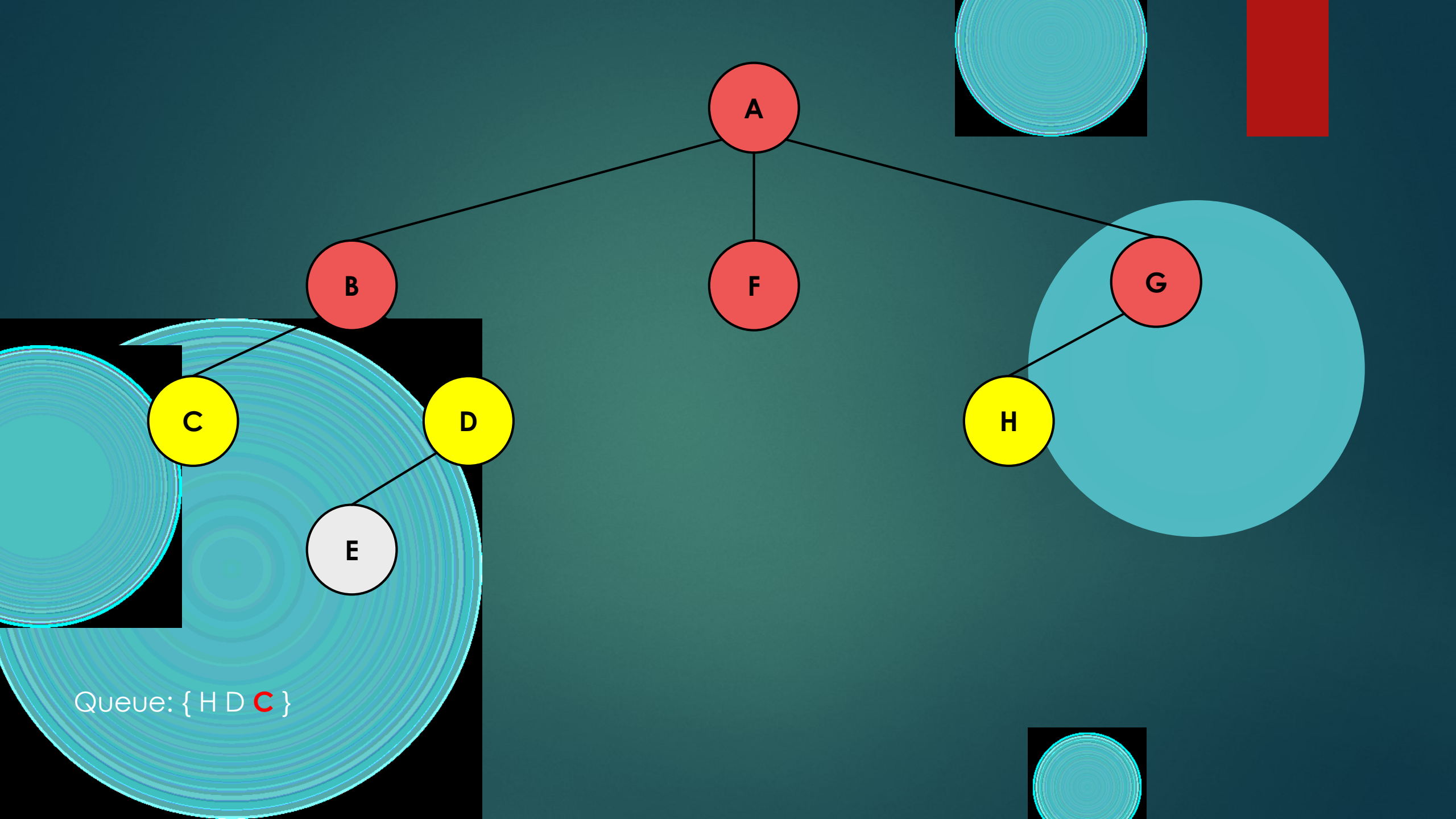

A
B
F
G
C
D
H
E
Queue: { D C G F }

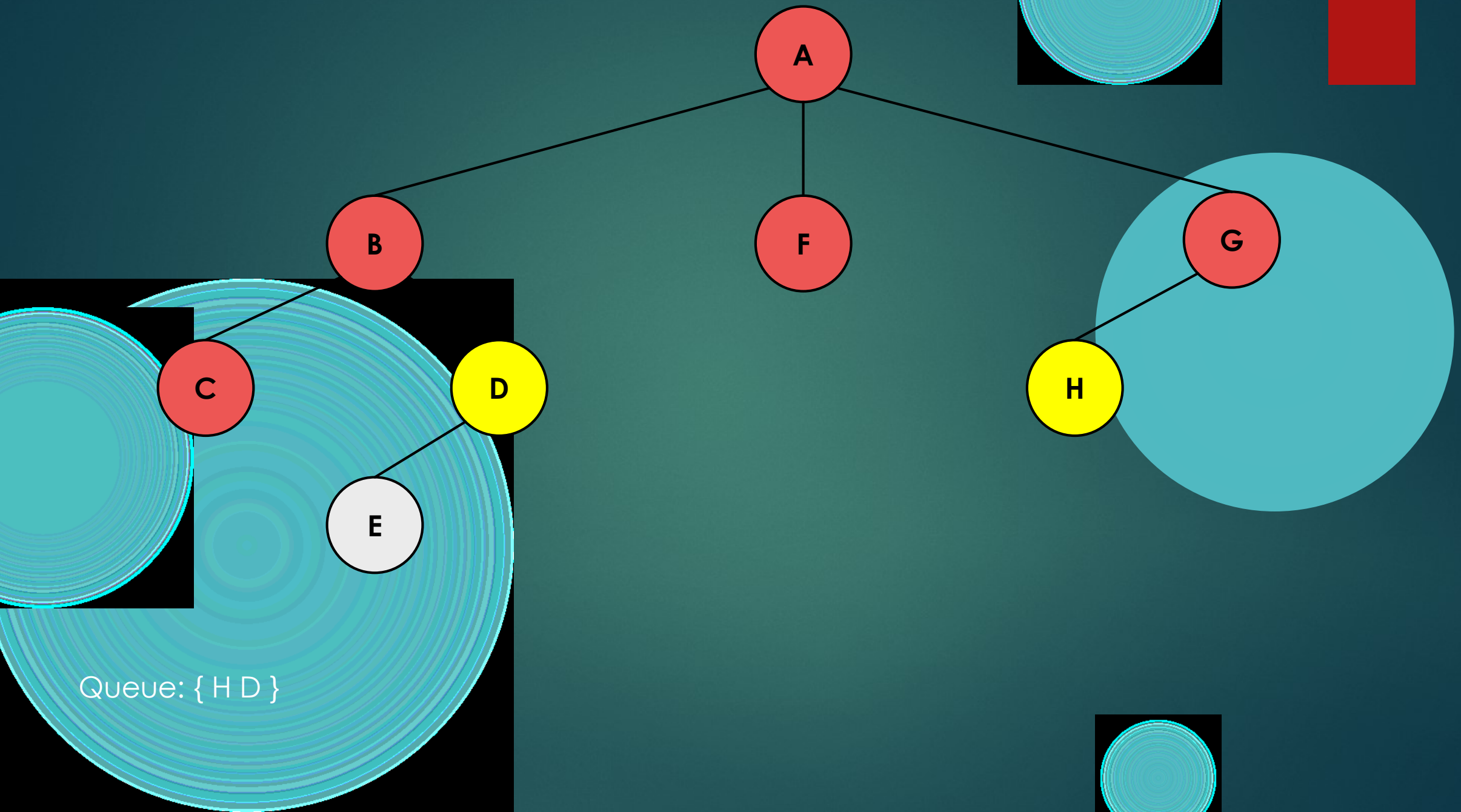

A
B
F
G
C
D
H
E
Queue: { D C G }

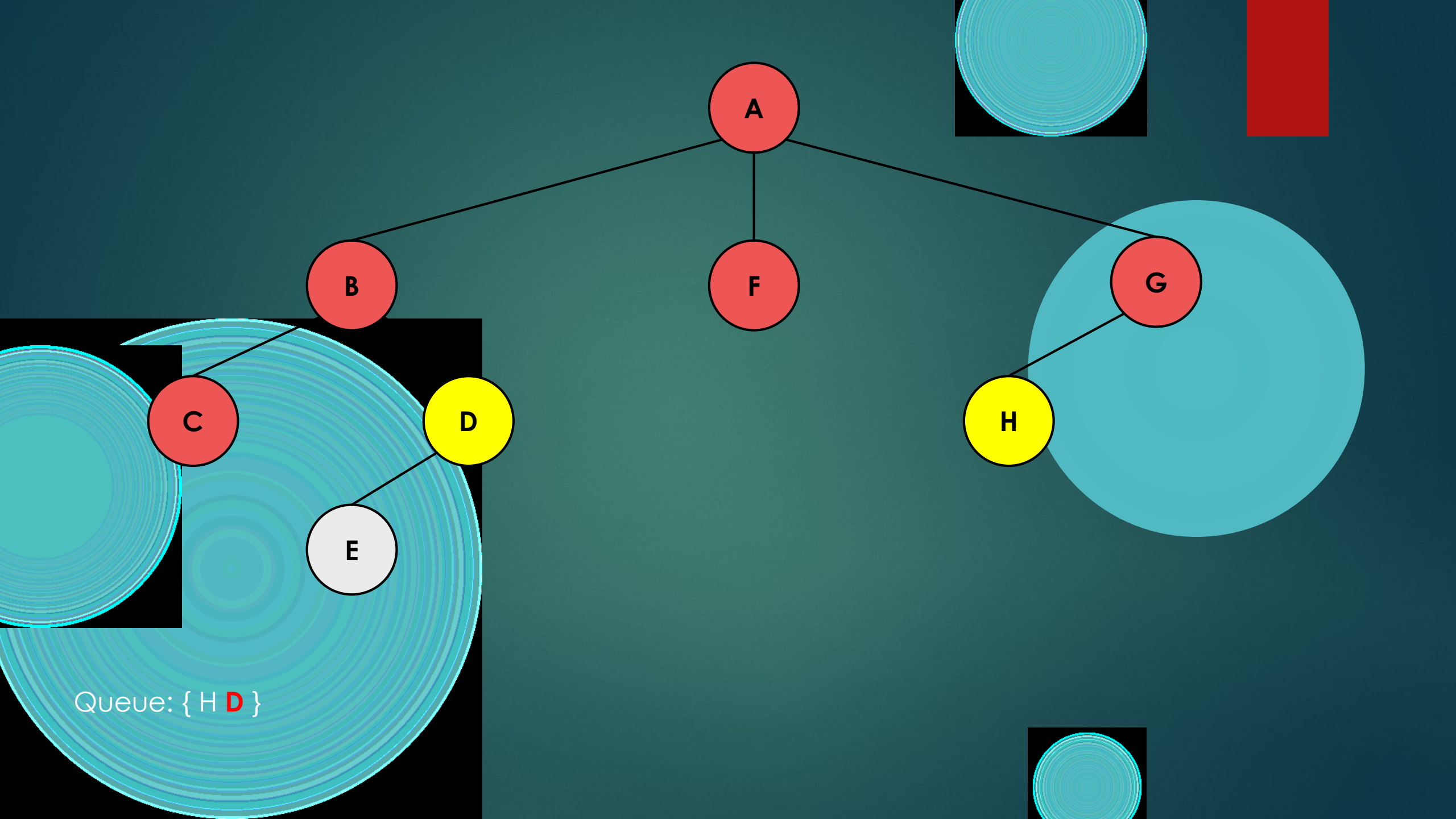

A
B
F
G
C
D
H
E
Queue: { D C G }

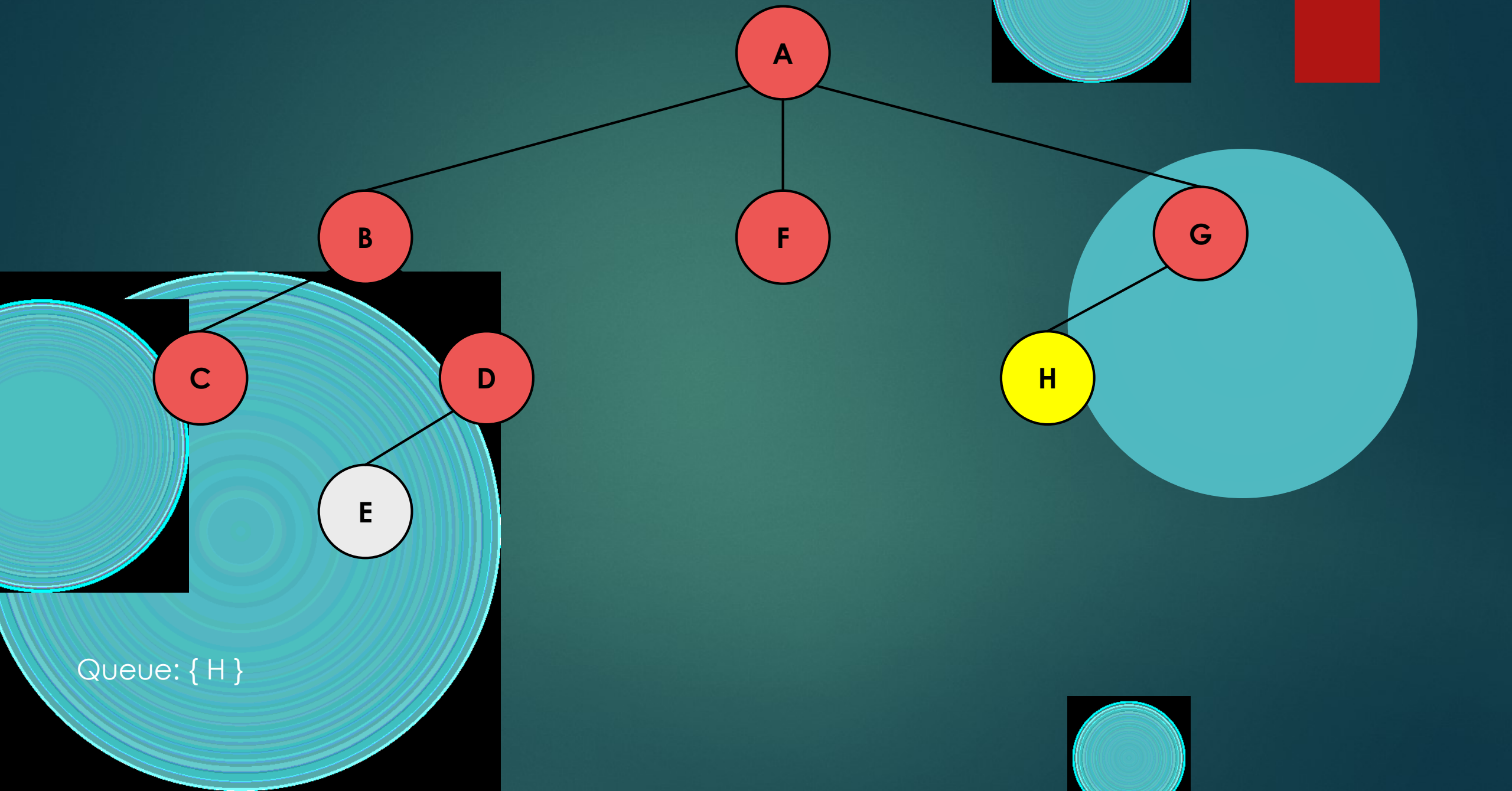

A
B
F
G
C
D
H
E
Queue: { D C }

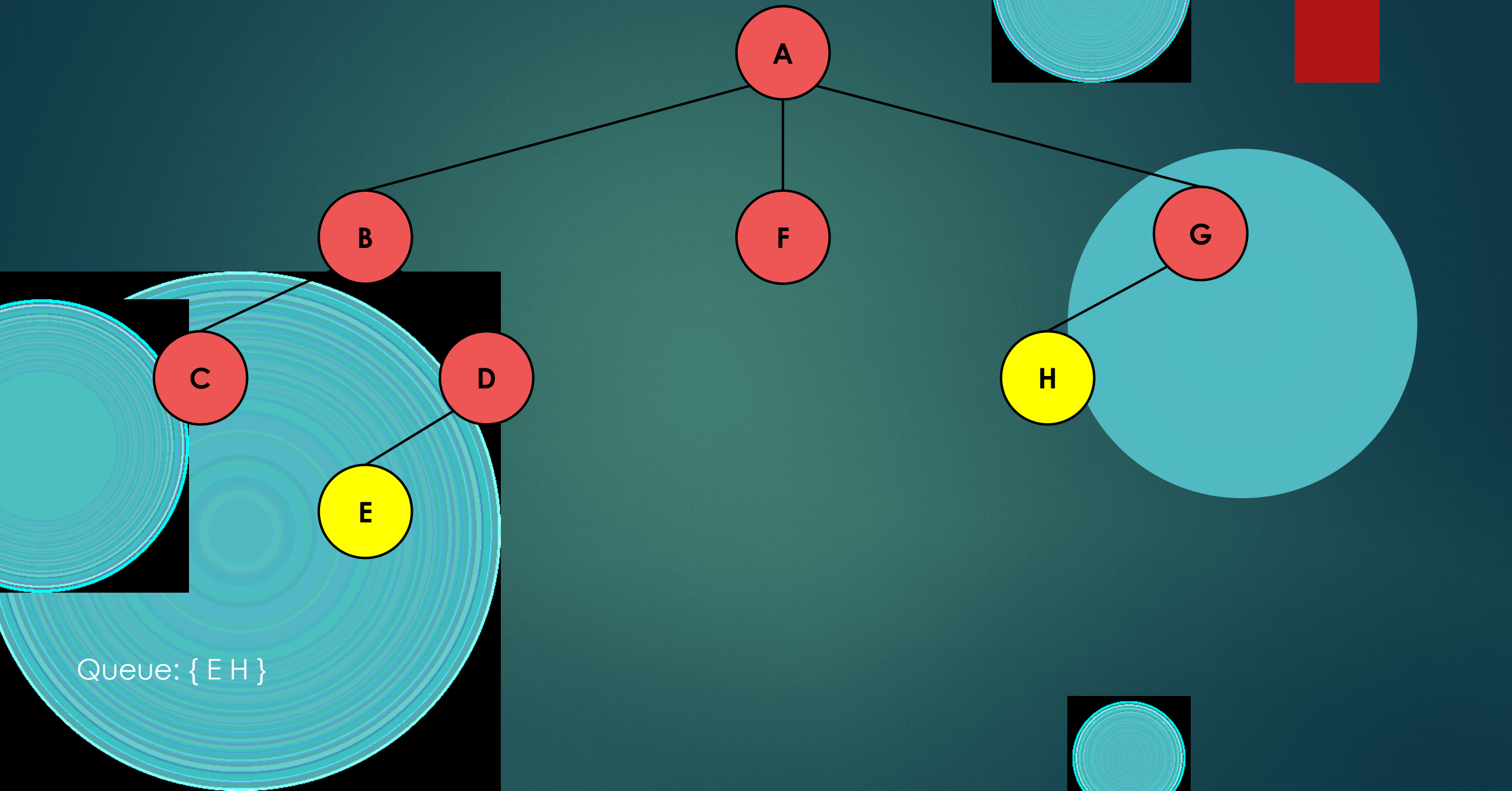

A
B
F
G
C
D
H
E
Queue: { H D C }

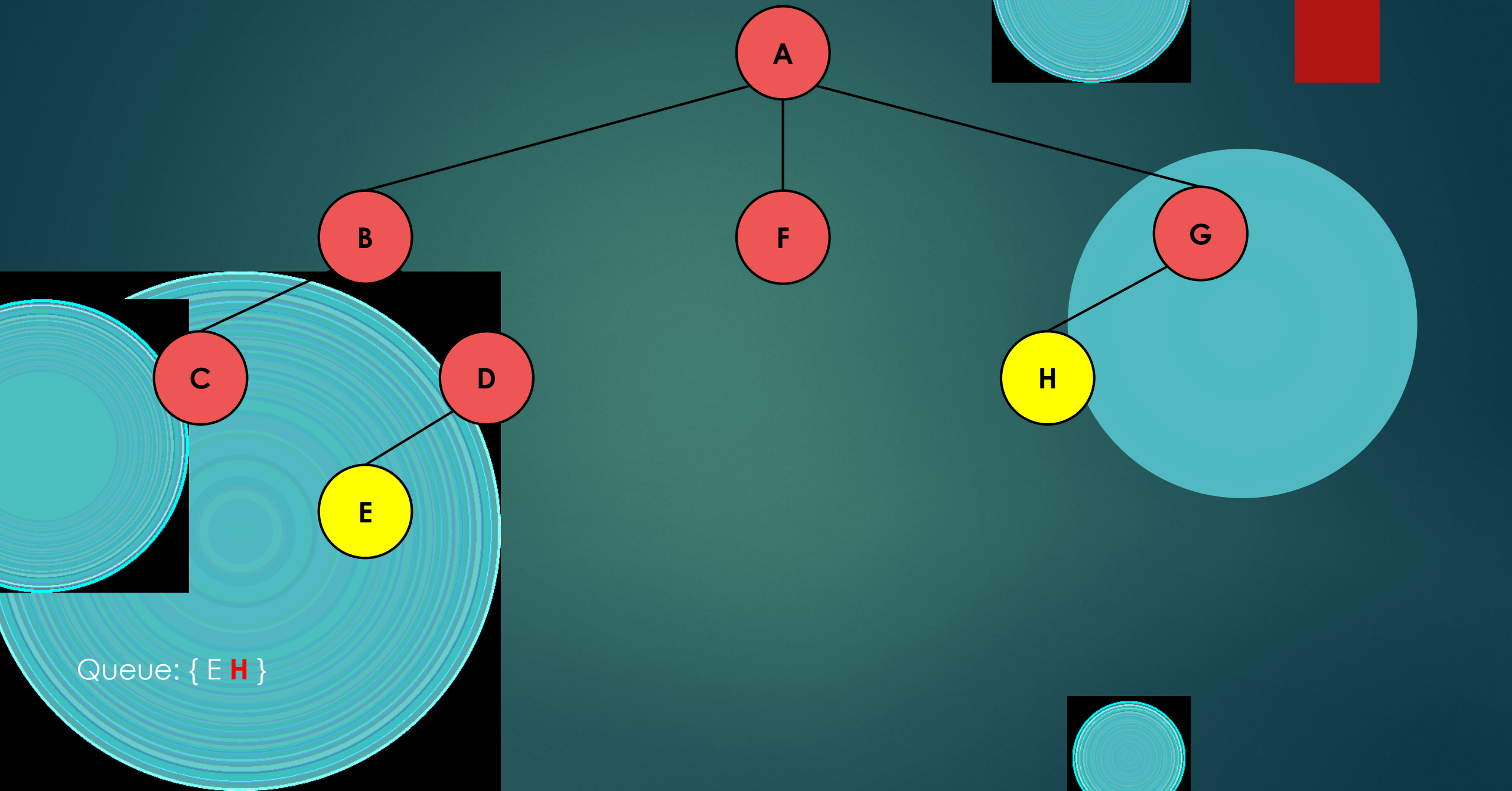

A
B
F
G
C
D
H
E
Queue: { H D C }

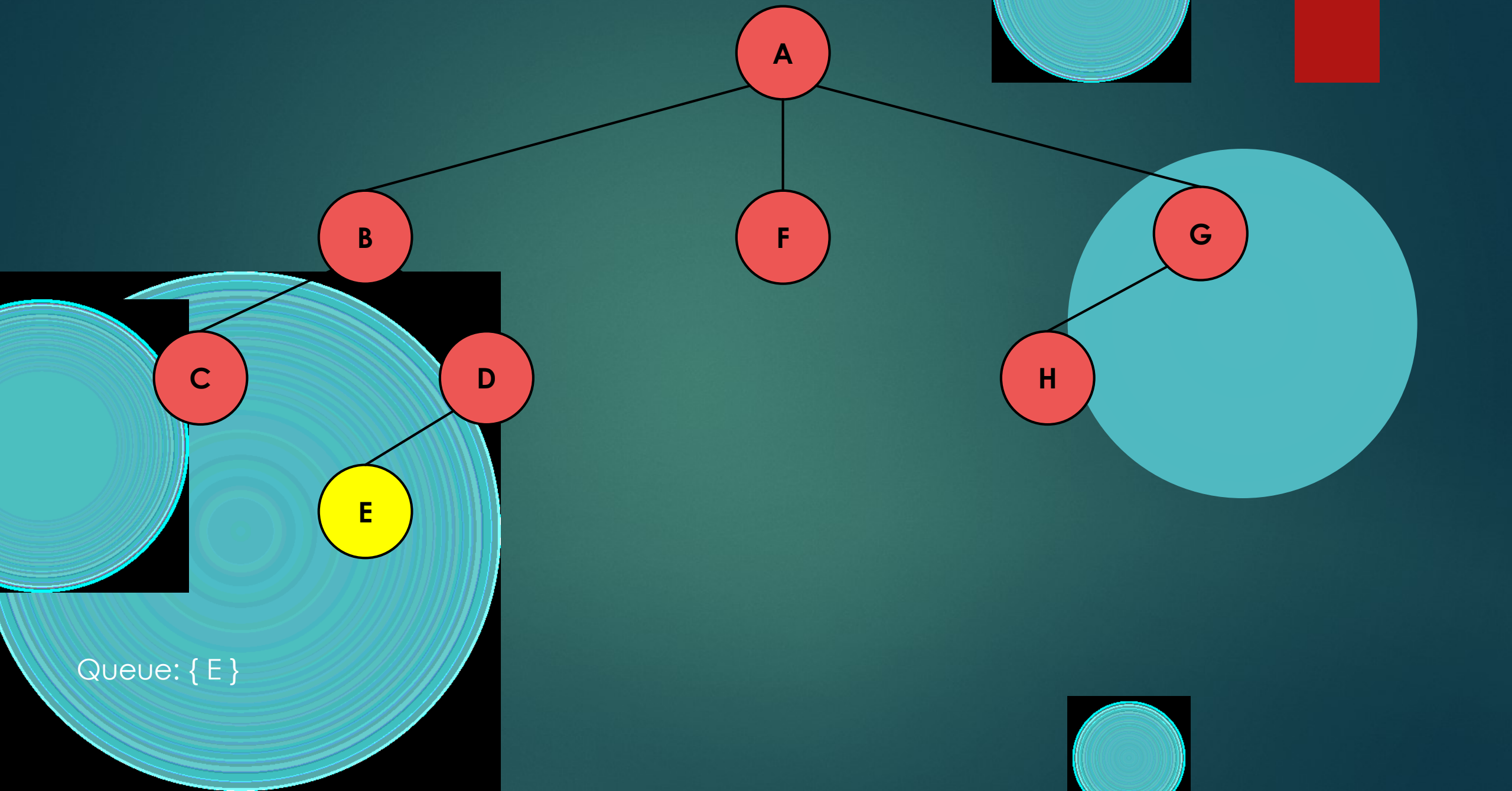

A
B
F
G
C
D
H
E
Queue: { H D }

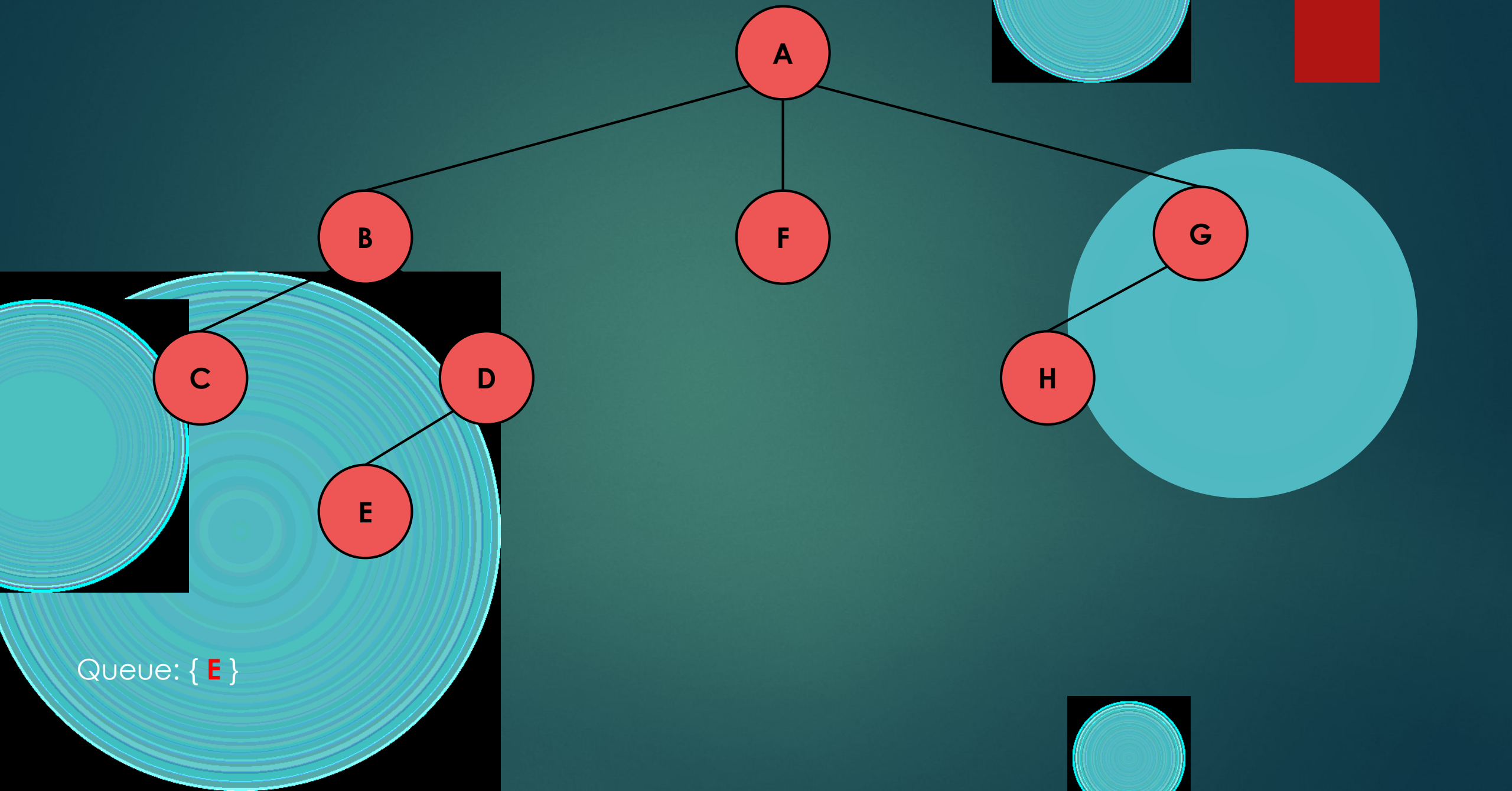

A
B
F
G
C
D
H
E
Queue: { H D }

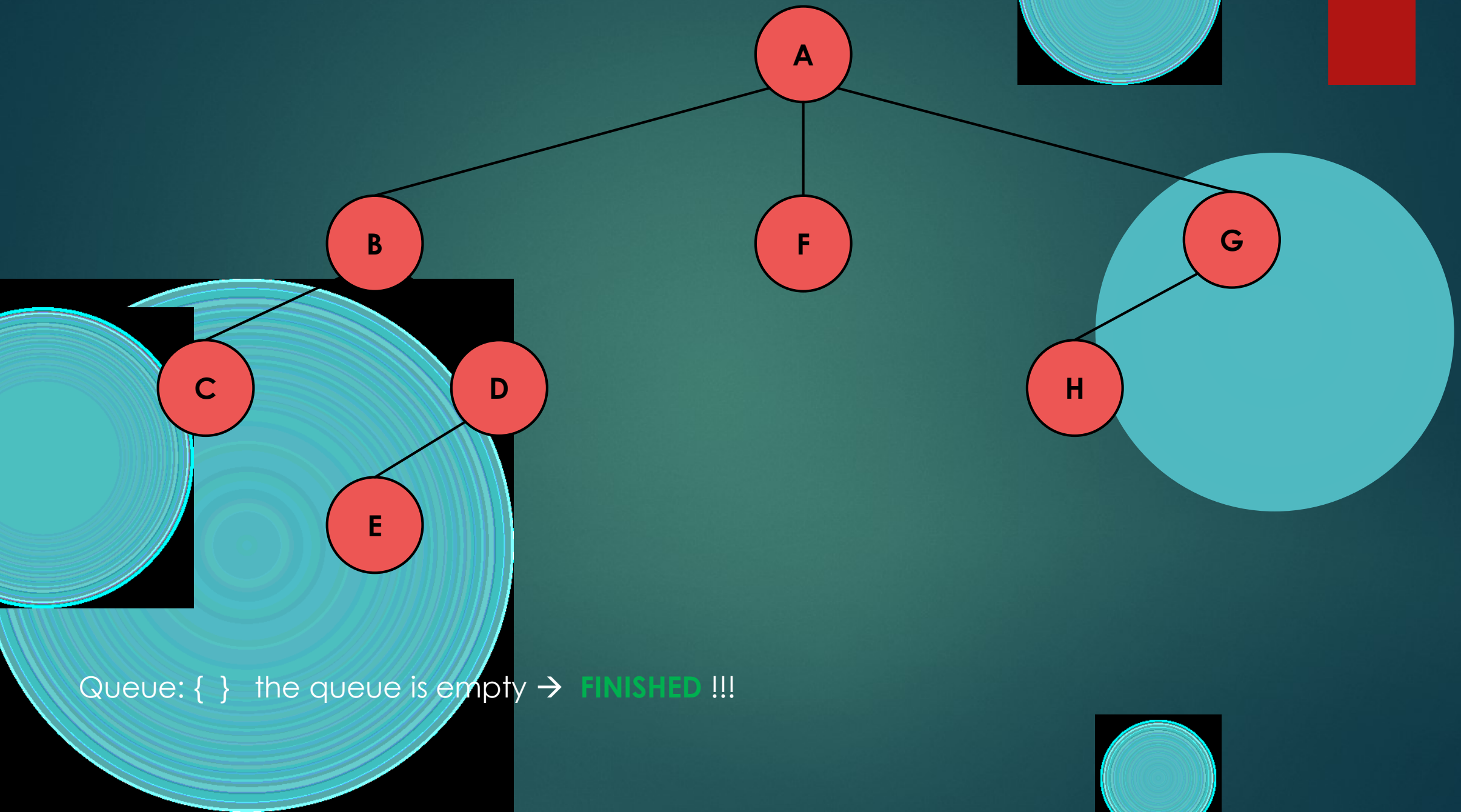

A
B
F
G
C
D
H
E
Queue: { H }


A
B
F
G
C
D
H
E
Queue: { E H }


A
B
F
G
C
D
H
E
Queue: { E H }

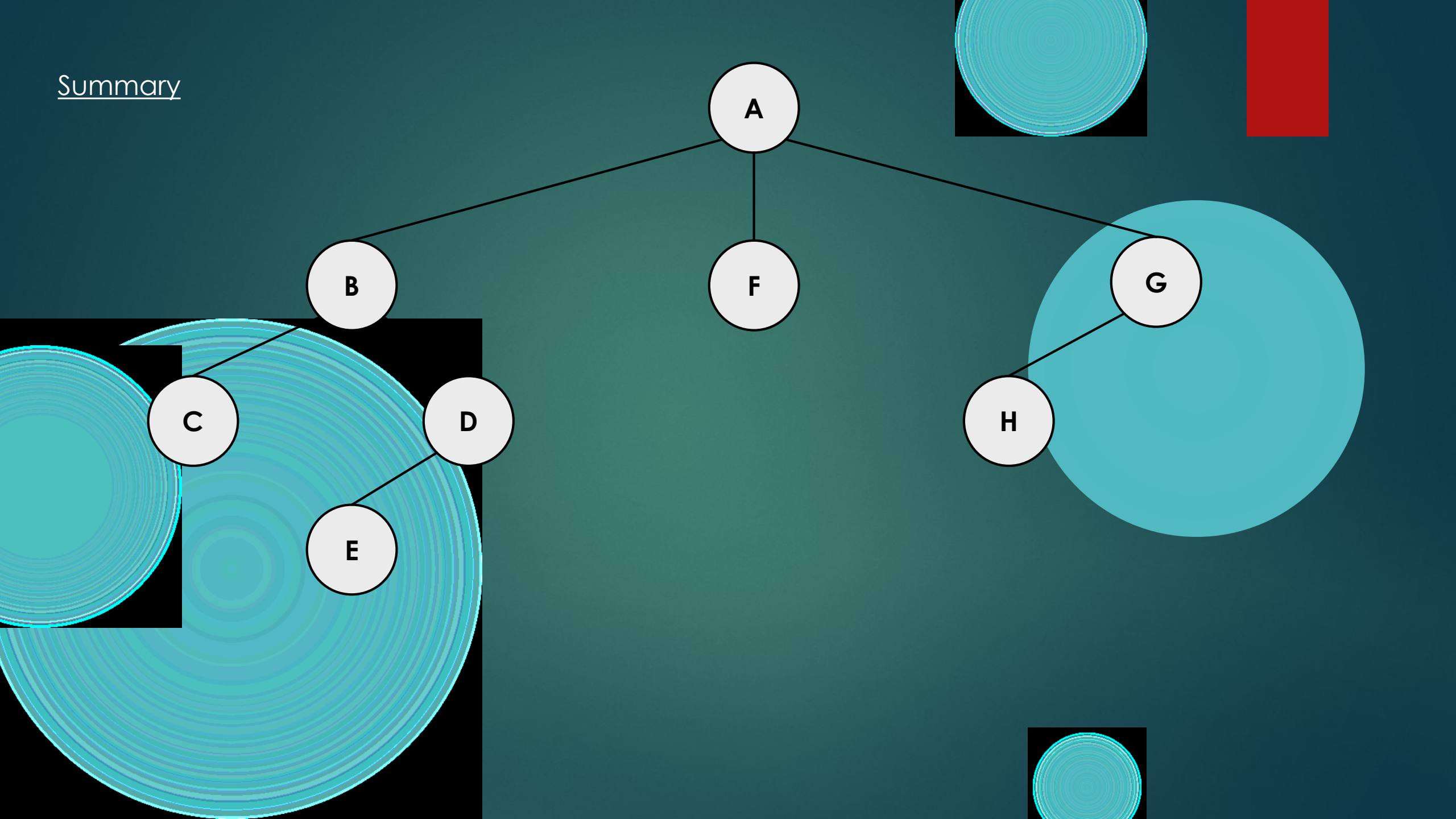

A
B
F
G
C
D
H
E
Queue: { E }


A
B
F
G
C
D
H
E
Queue: { E }

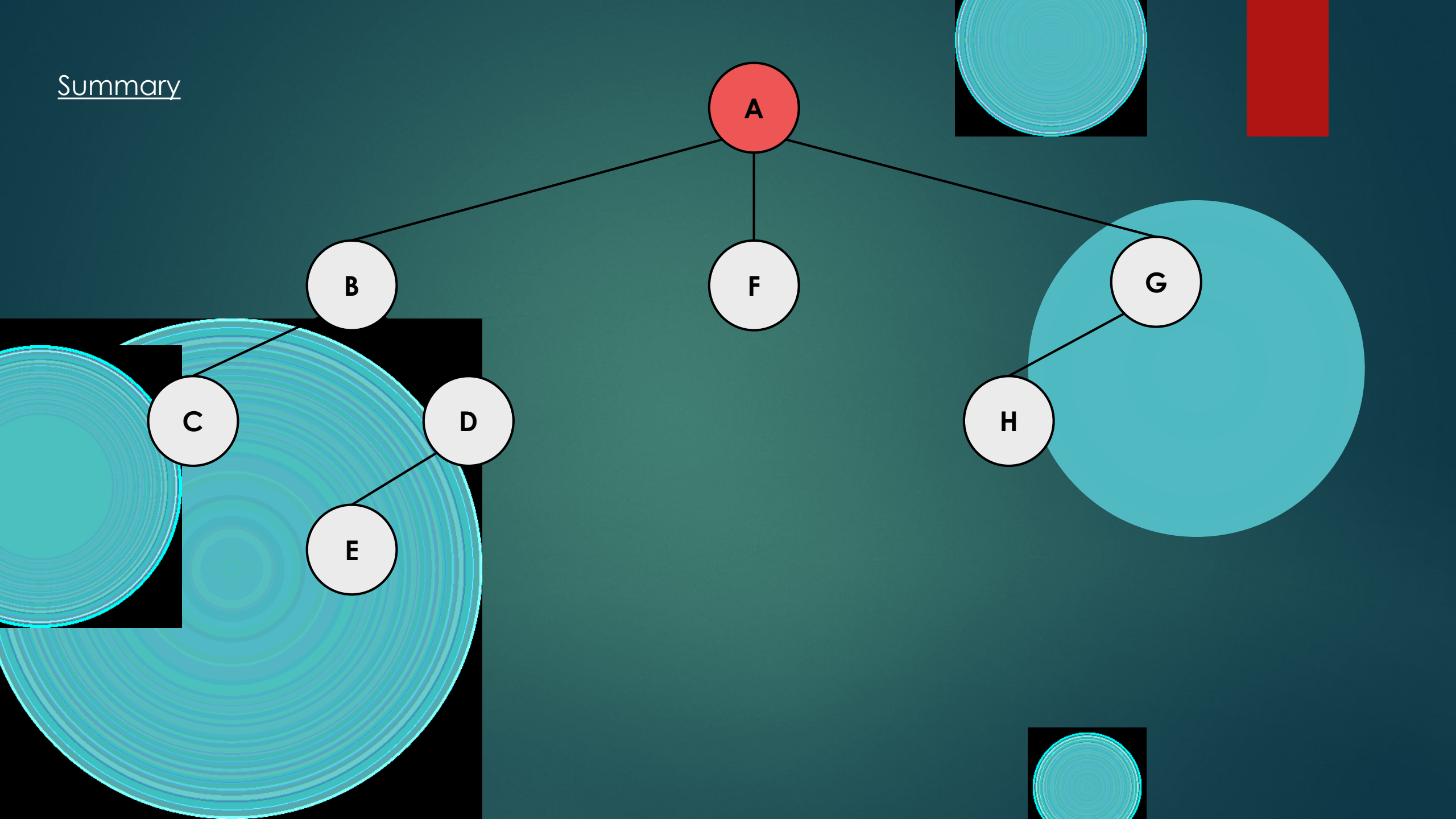


Queue: { } the queue is empty → **FINISHED** !!!

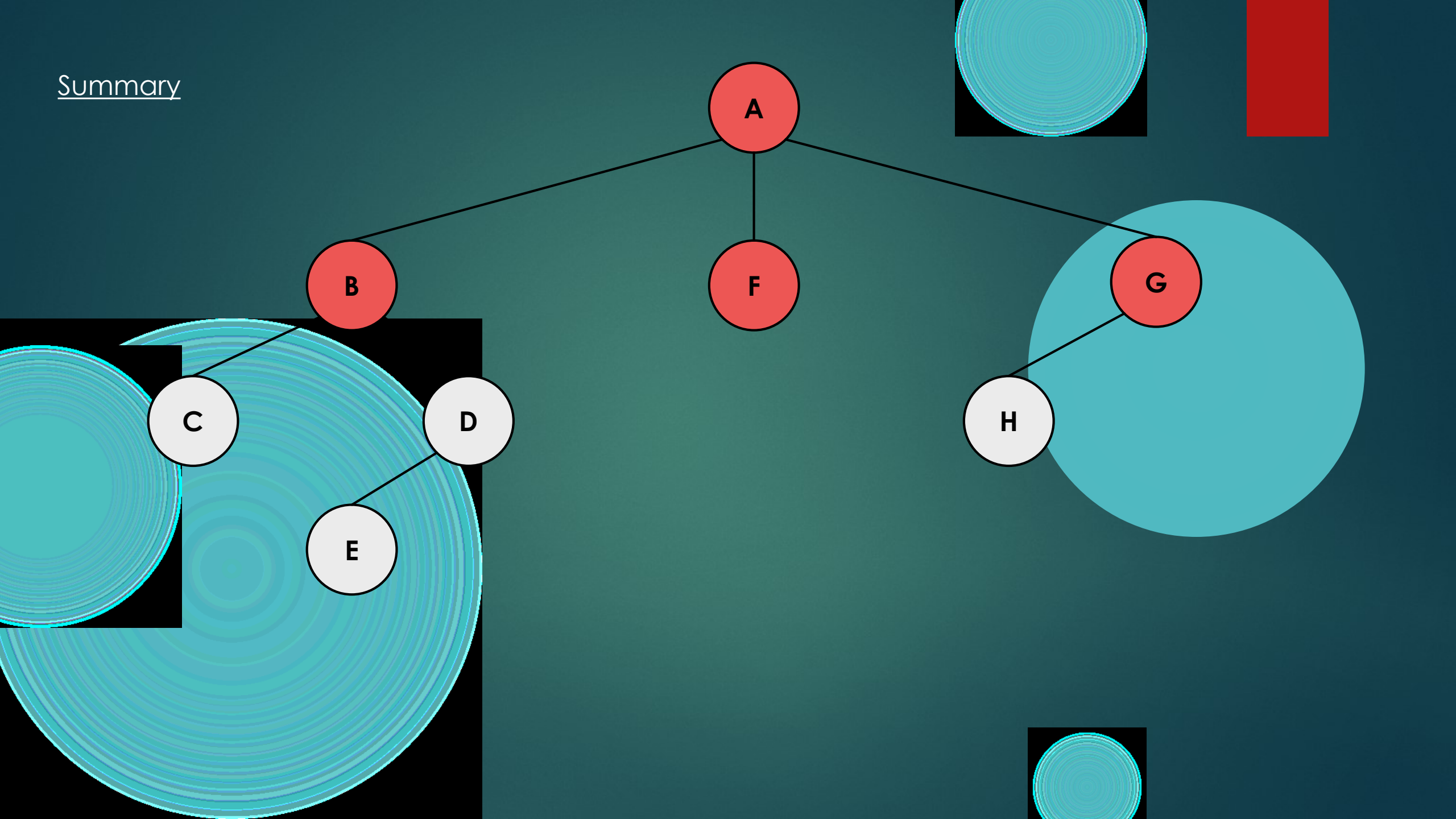
Summary



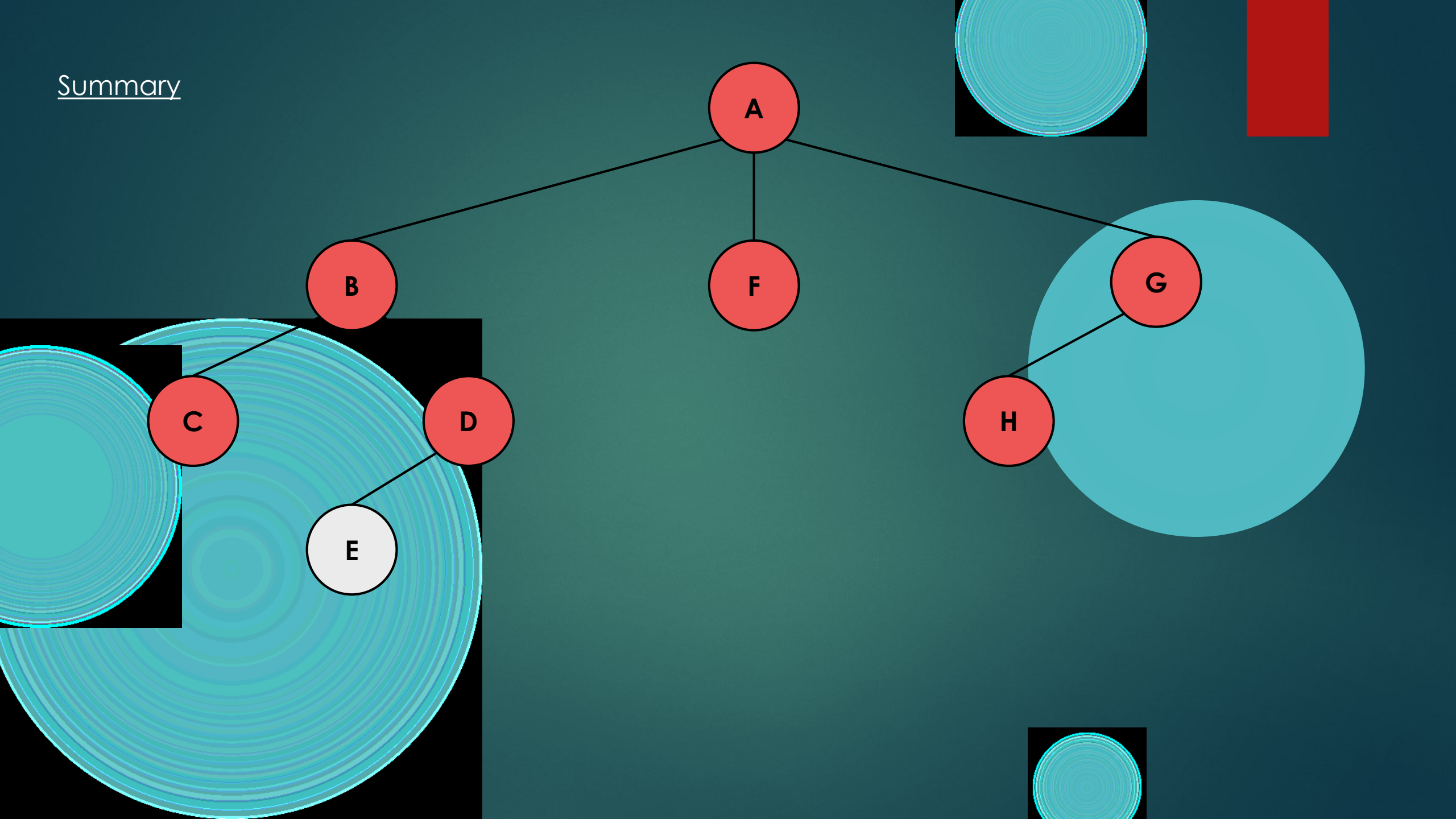
Summary



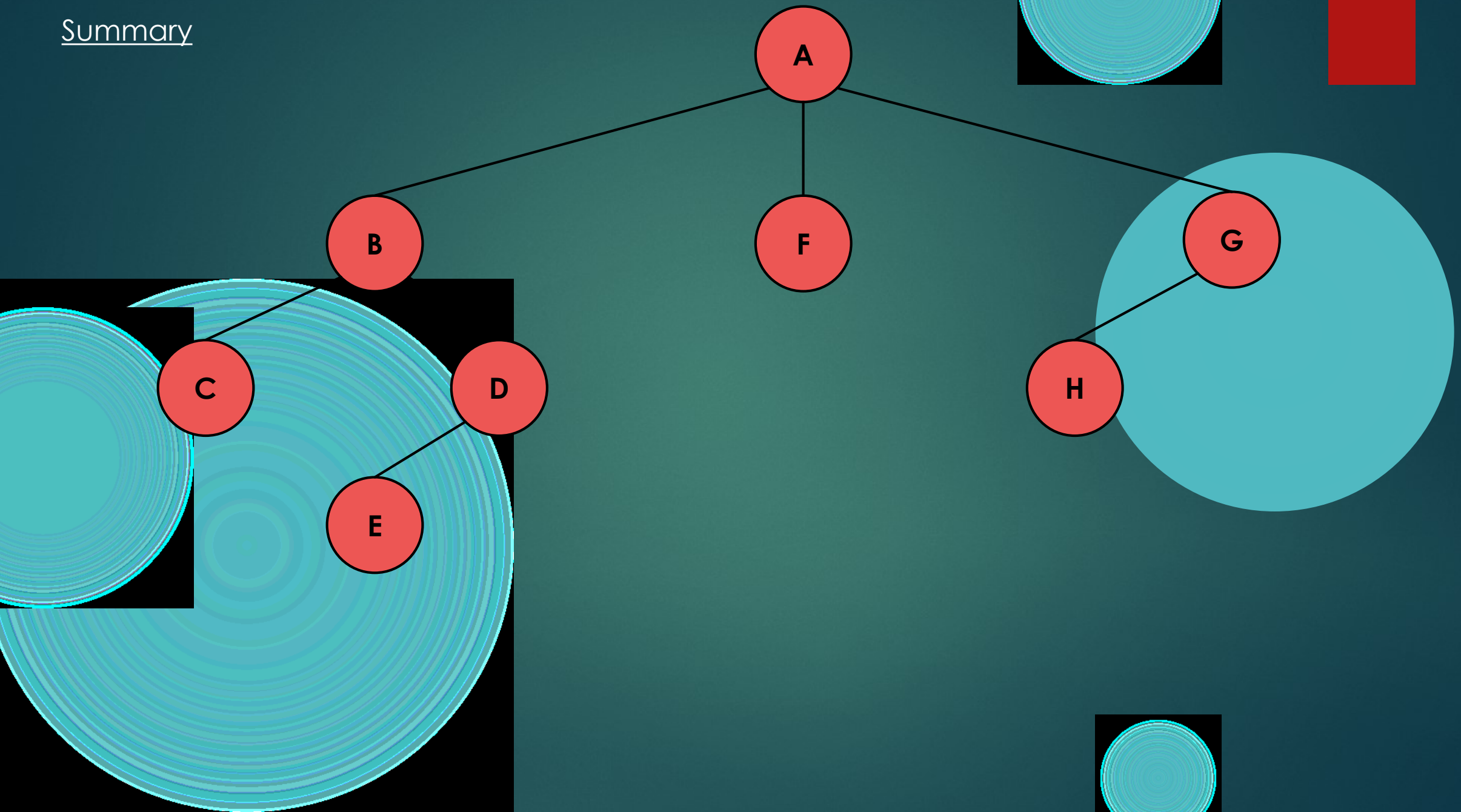
Summary



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 !!!

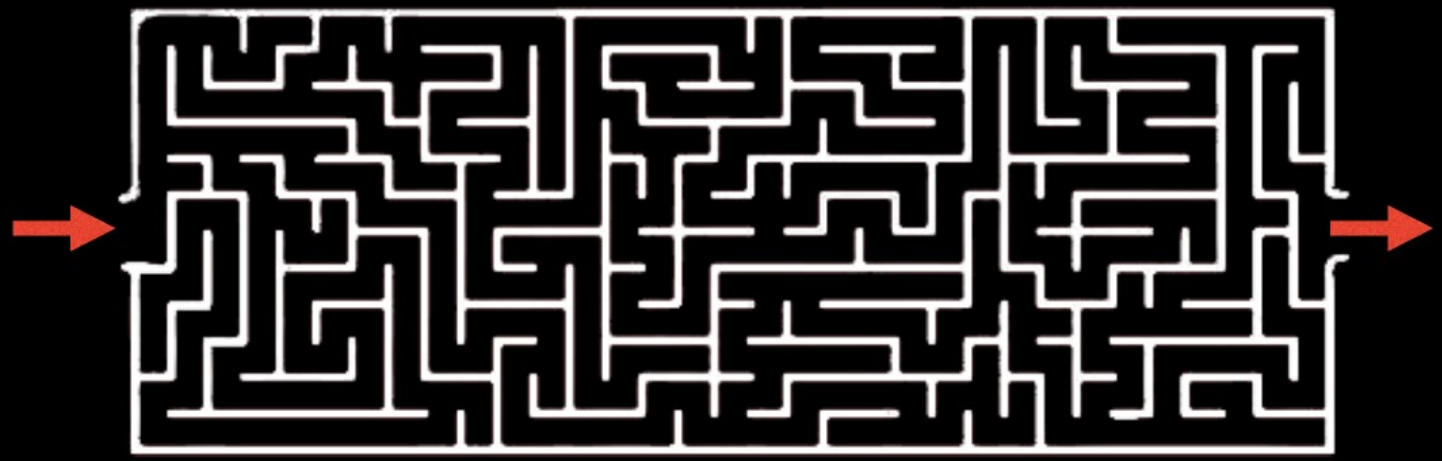
BFS IMPLEMENTATION

BFS Shortest Path on a Grid

Motivation

- Many problems in graph theory can be represented using a grid.
- Grids are a form of implicit graph because we can determine a node's neighbors based on our location within the grid.

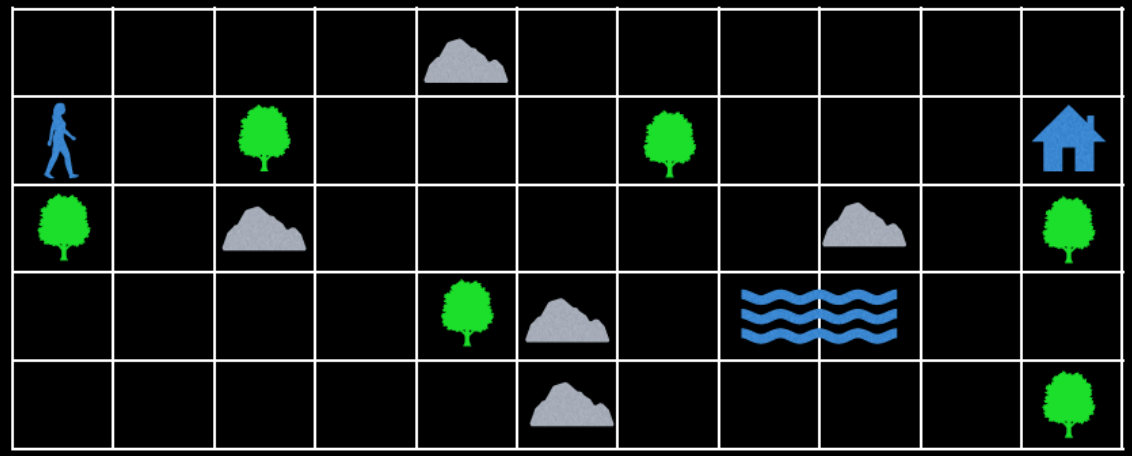
A type of problem that involves finding a path through a grid is solving a maze:



Motivation

- Many problems in graph theory can be represented using a grid.
- Grids are a form of implicit graph because we can determine a node's neighbors based on our location within the grid.

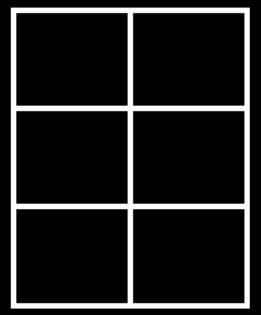
Another example could be routing through obstacles (trees, rivers, rocks, etc) to get to a location:



Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.

Empty Grid

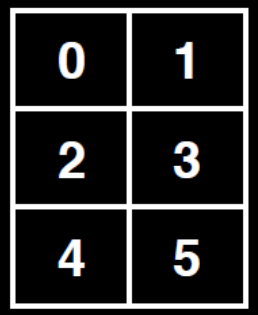


IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.

Empty Grid

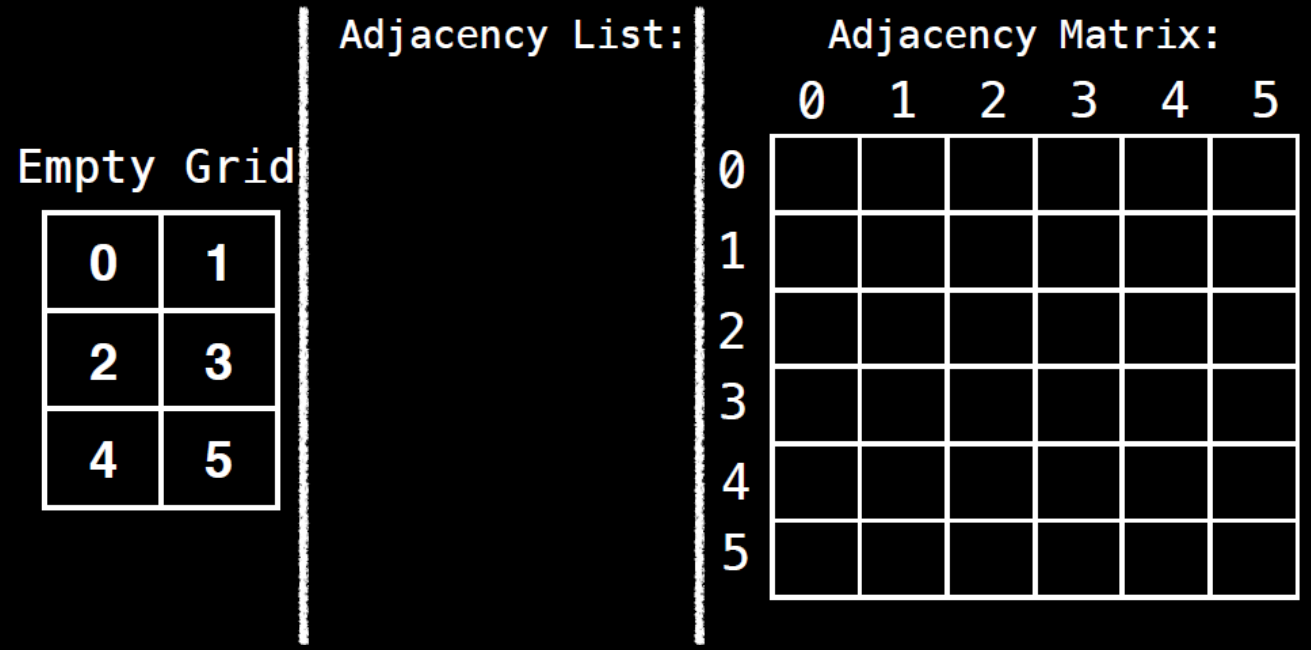


First label the cells in the grid with numbers [0, n) where n = #rows x #columns

IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

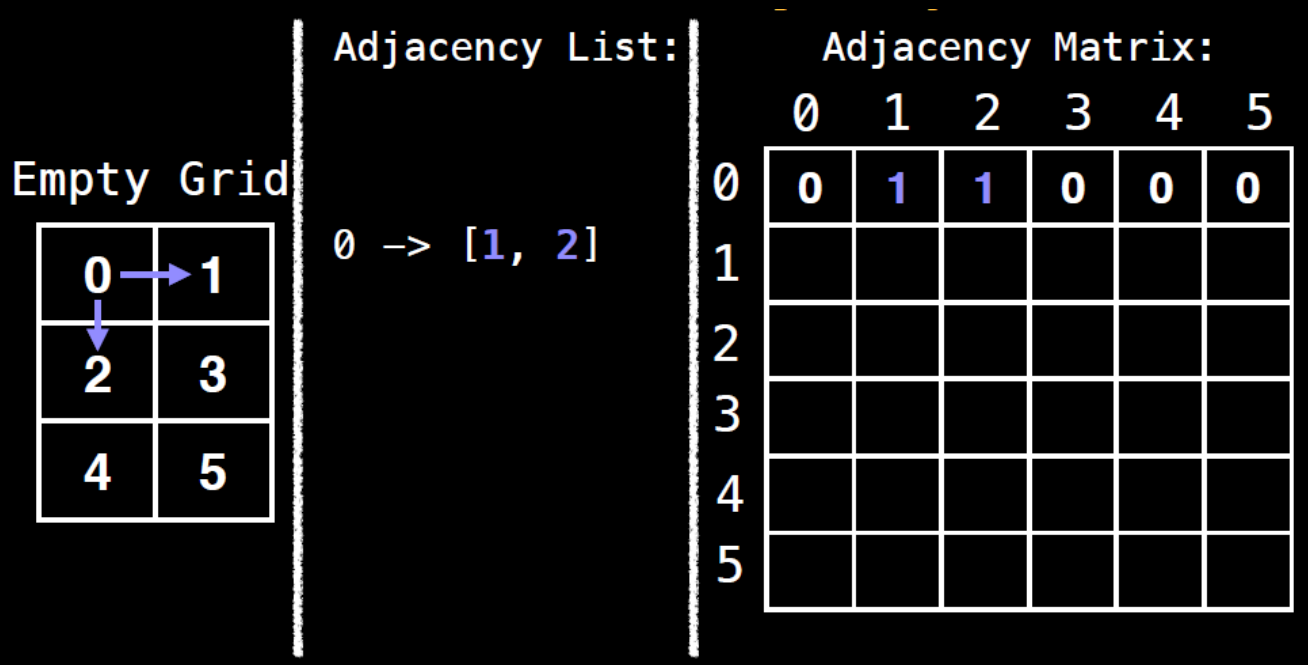
A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.



IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

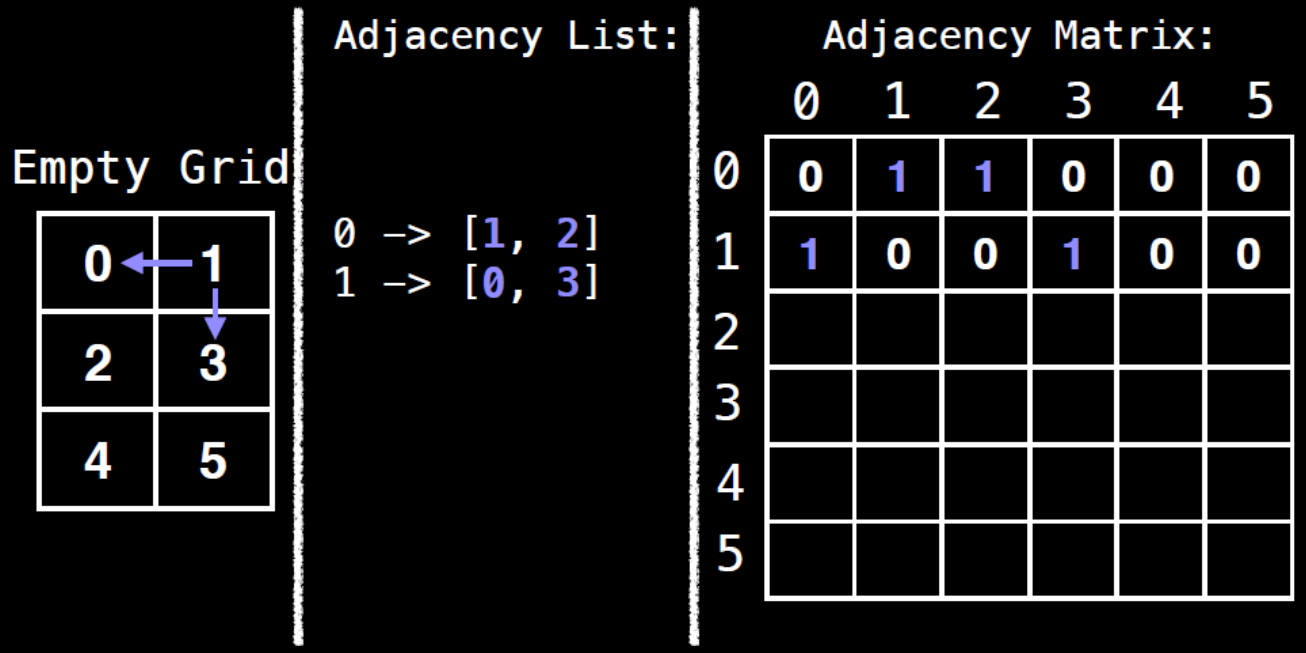
A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.



IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

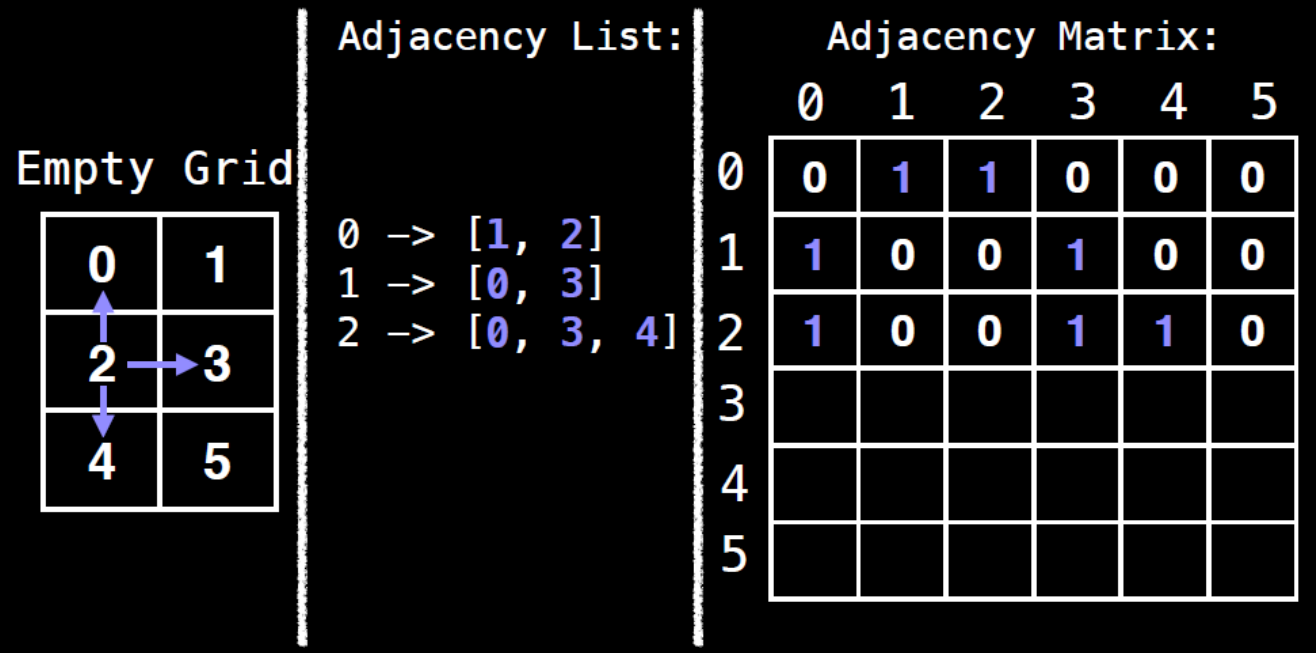
A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.



IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

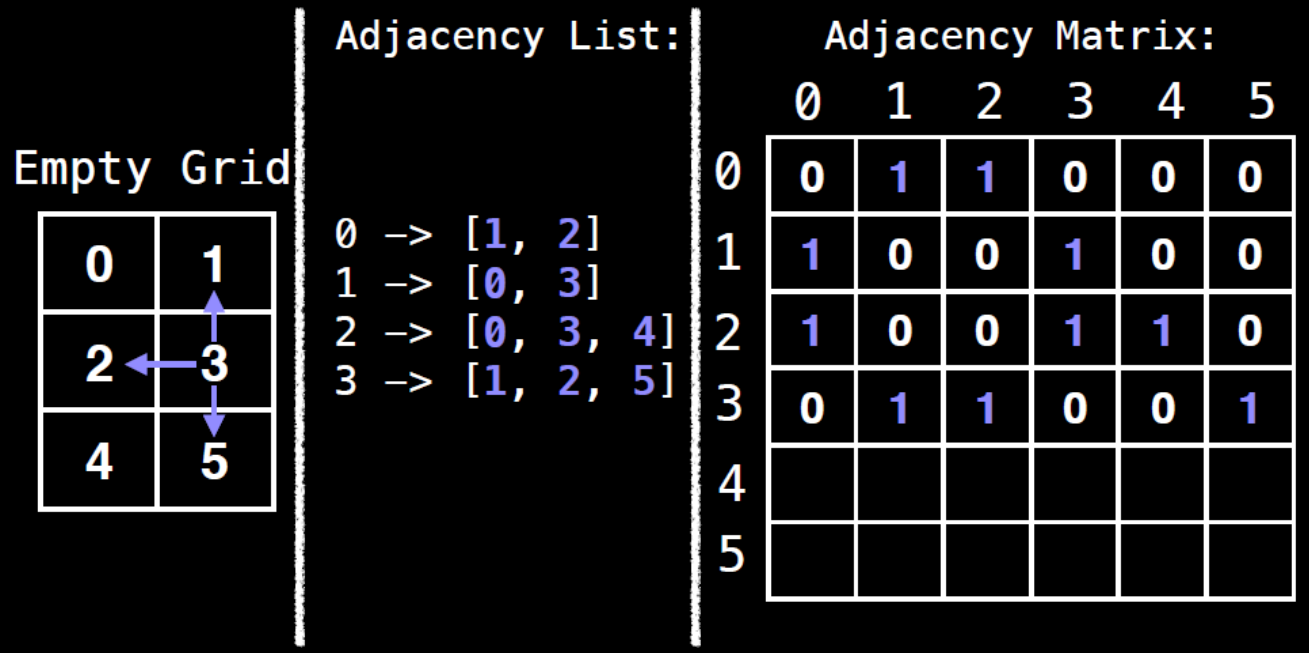
A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.



IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

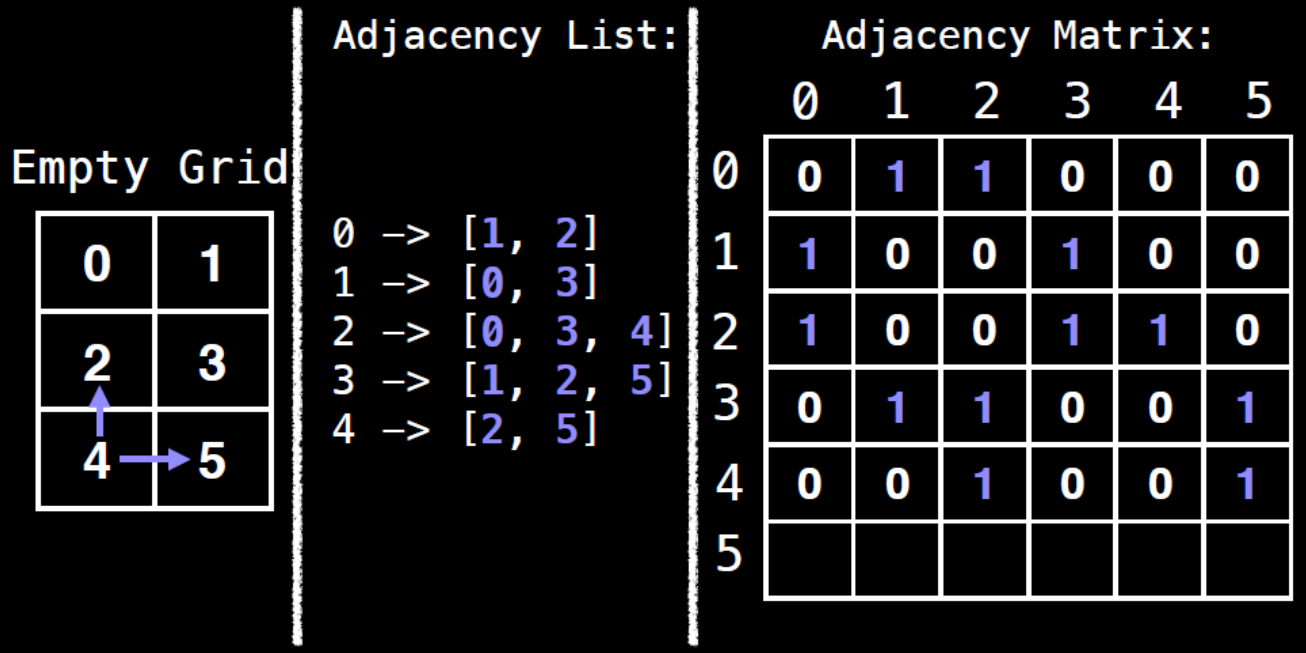
A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.



IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

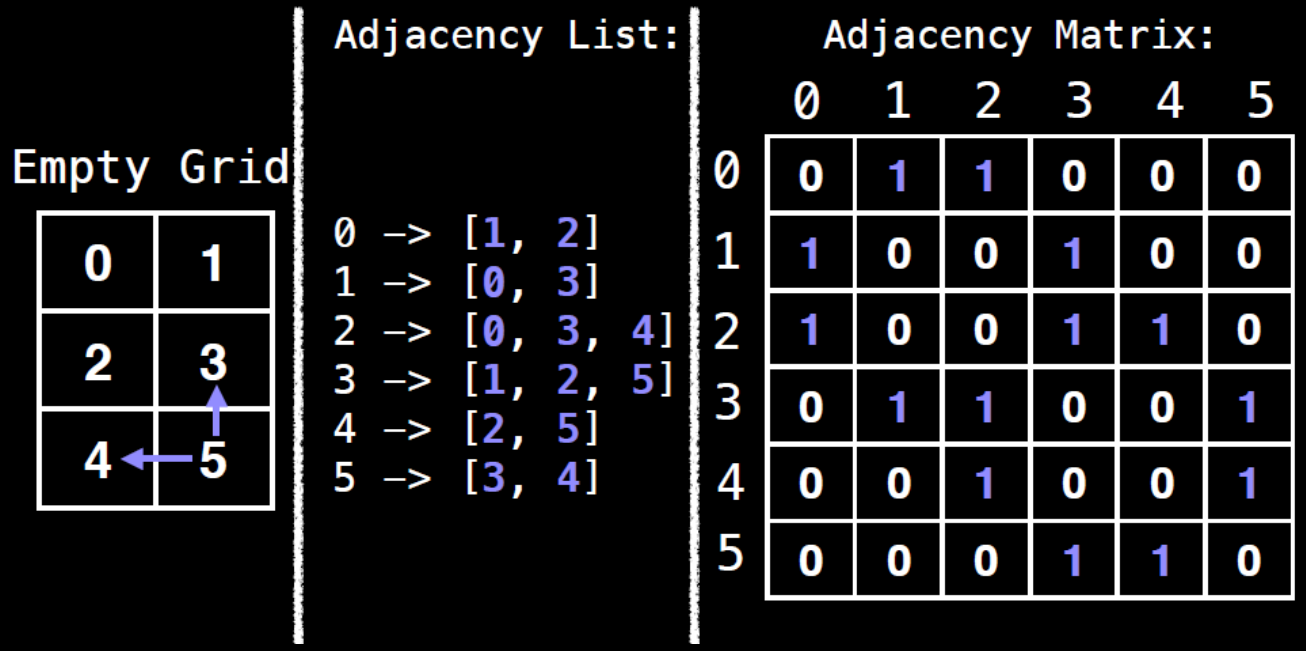
A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.



IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

Graph Theory on Grids

A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format such as an adjacency list/matrix.



IMPORTANT: Assume the grid is unweighted and cells connect left, right, up and down.

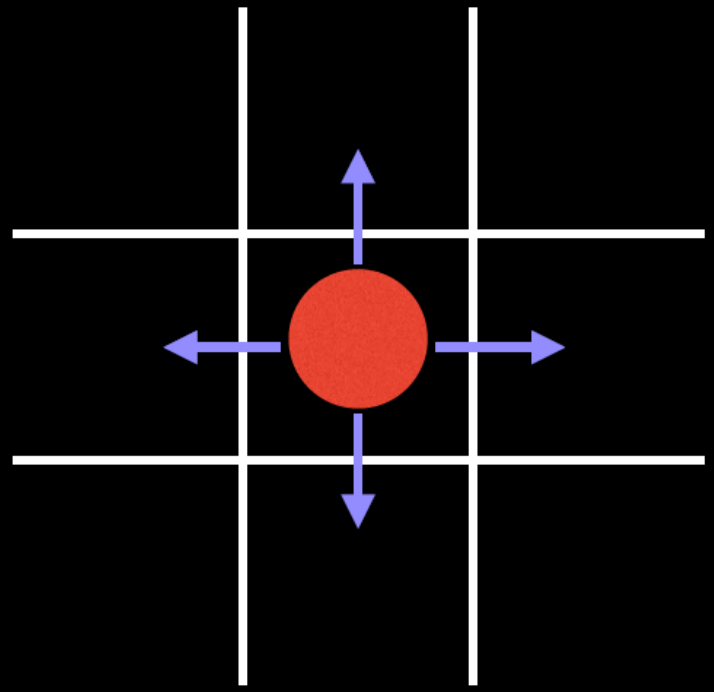
Graph Theory on Grids

Once we have an adjacency list/matrix we can run whatever specialized graph algorithm to solve our problem such as: shortest path, connected components, etc…

However, transformations between graph representations can usually be avoided due to the structure of a grid.

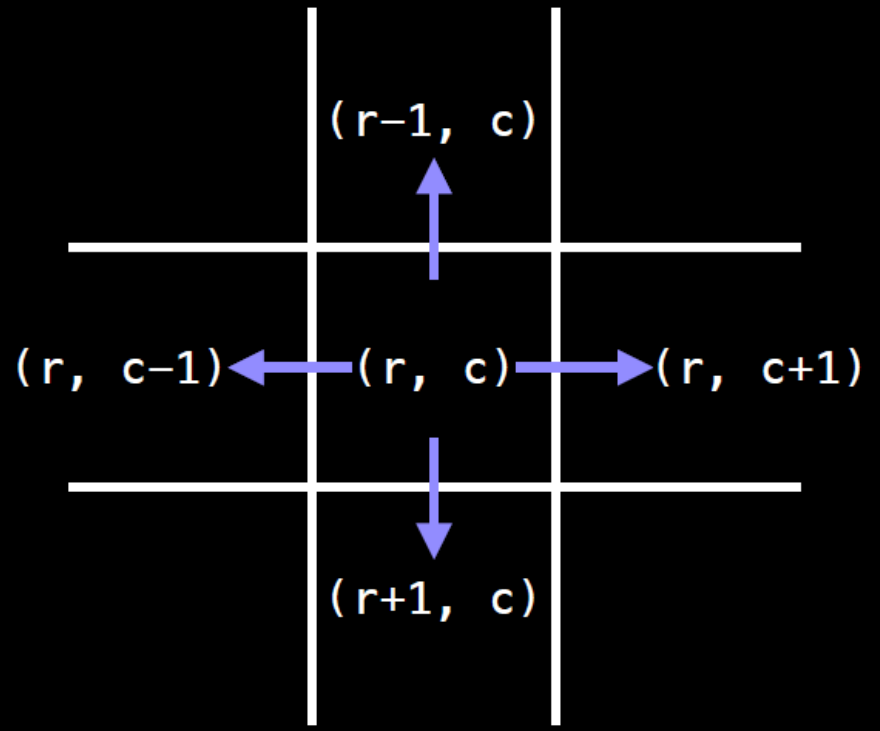
Direction Vectors

Due to the structure of a grid, if we are at the red ball in the middle we know we can move left, right, up and down to reach adjacent cells:



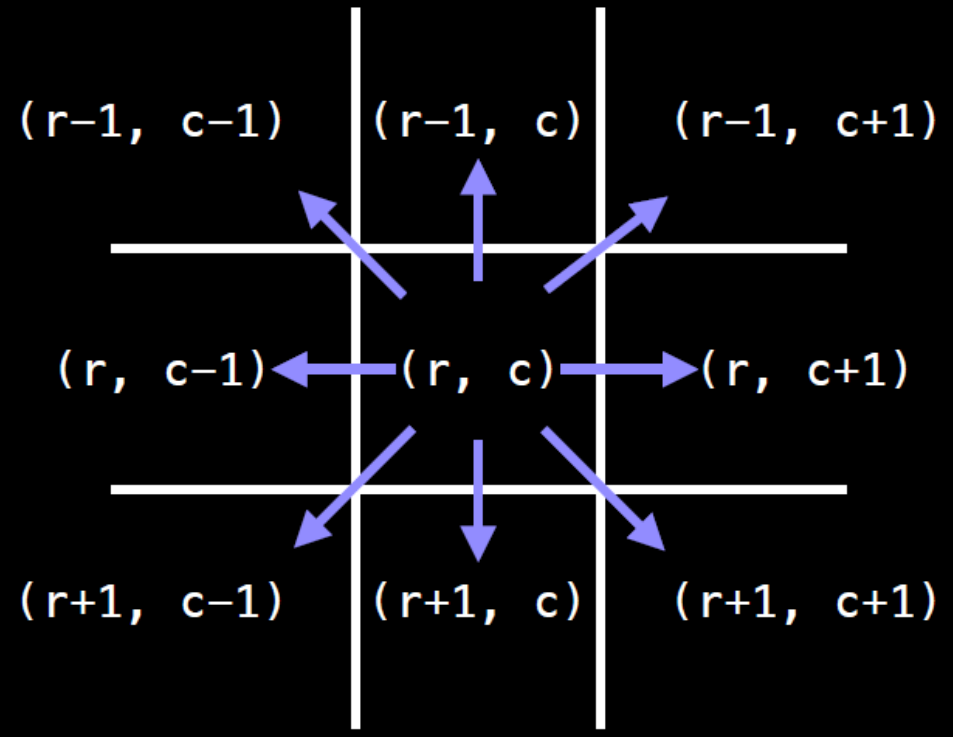
Direction Vectors

Mathematically, if the red ball is at the row-column coordinate (r, c) we can add the row vectors [-1, 0], [1, 0], [0, 1], and [0, -1] to reach adjacent cells.



Direction Vectors

If the problem you are trying to solve allows moving diagonally then you can also include the row vectors: [-1, -1], [-1, 1], [1, 1], [1, -1]



Direction Vectors

This makes it very easy to access neighbouring cells from the current row-column position:

```
# Define the direction vectors for
# north, south, east and west.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]

for(i = 0; i < 4; i++):
  rr = r + dr[i]
  cc = c + dc[i]
  # Skip invalid cells. Assume R and
  # C for the number of rows and columns
  if rr < 0 or cc < 0: continue
  if rr >= R or cc >= C: continue
  #(rr, cc) is a neighbouring cell of (r, c)
```

Dungeon Problem Statement

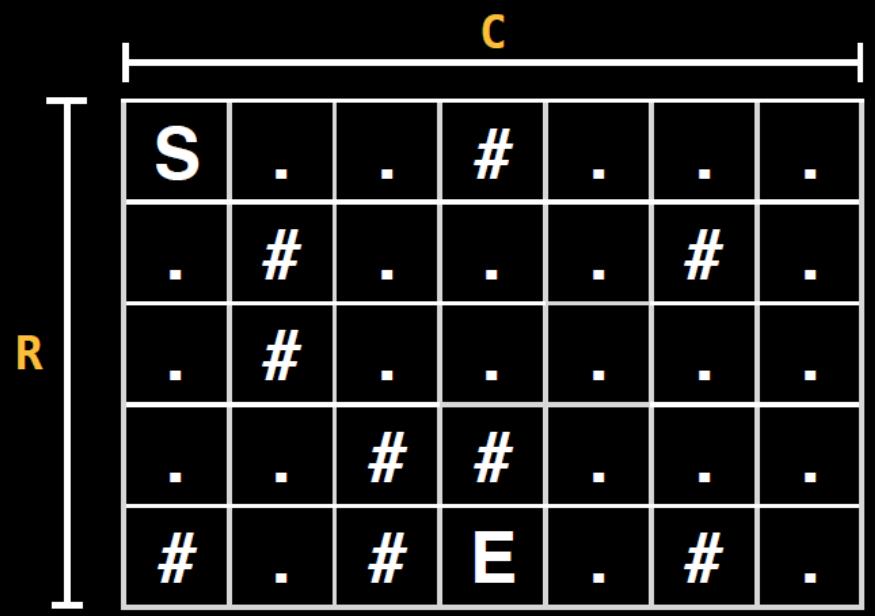
You are trapped in a 2D dungeon and need to find the quickest way out!
The dungeon is composed of unit cubes which may or may not be filled with rock.
It takes one minute to move one unit north, south, east, west. You cannot move diagonally and the maze is surrounded by solid rock on all sides.

Is an escape possible?
If yes, how long will it take?

This is an easier version of the “Dungeon Master” problem on Kattis:
open.kattis.com/problems/dungeon.

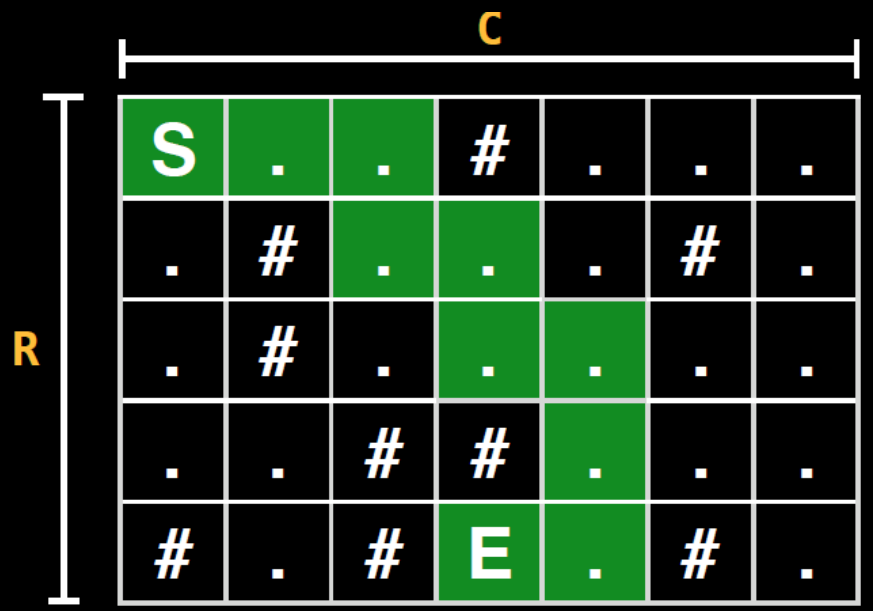
Dungeon Problem Statement

The dungeon has a size of R x C and you start at cell ’S’ and there’s an exit at cell ‘E’. A cell full of rock is indicated by a ’#’ and empty cells are represented by a ’.’

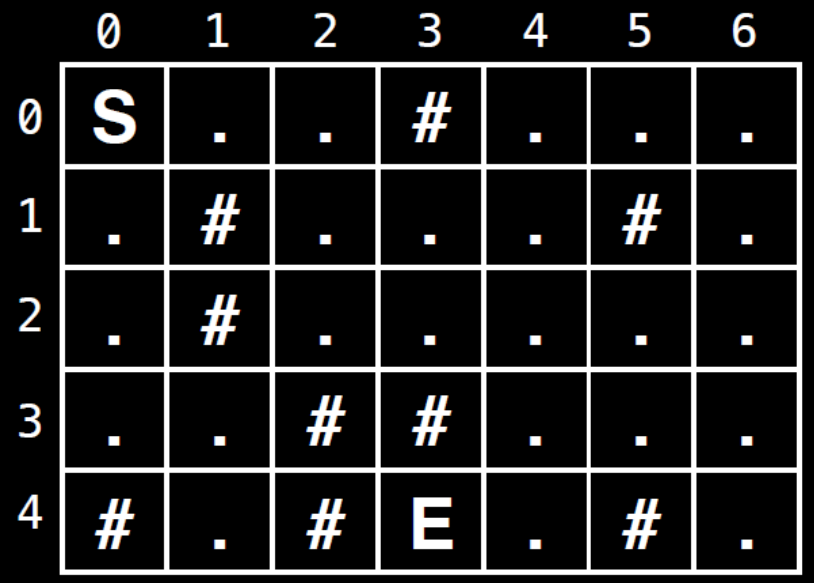


Dungeon Problem Statement

The dungeon has a size of R x C and you start at cell ’S’ and there’s an exit at cell ‘E’. A cell full of rock is indicated by a ’#’ and empty cells are represented by a ’.’

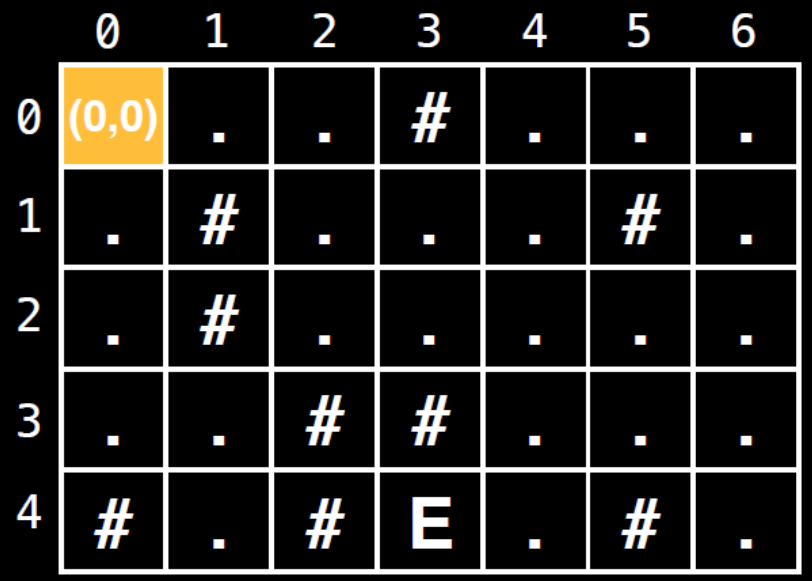


Dungeon Problem Statement



Dungeon Problem Statement

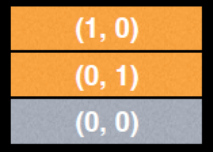
Start at the start node coordinate by adding (sr, sc) to the queue.



(0, 0)

Dungeon Problem Statement

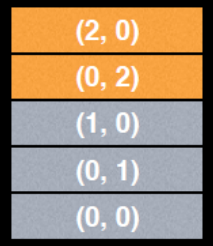
Start at the start node coordinate by adding (sr, sc) to the queue.



	0	1	2	3	4	5	6
0	(0,0)	(0,1)	.	#	.	.	.
1	(1,0)	#	.	.	.	#	.
2	.	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

Dungeon Problem Statement

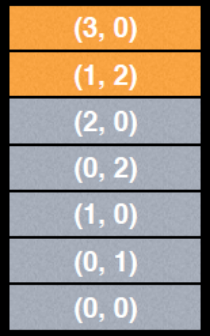
Start at the start node coordinate by adding (sr, sc) to the queue.



	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	.	.	.	#	.
2	(2,0)	#	.	.	.	.	.
3	.	.	#	#	.	.	.
4	#	.	#	E	.	#	.

Dungeon Problem Statement

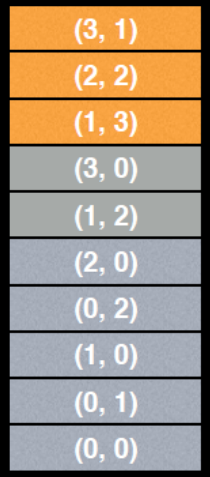
Start at the start node coordinate by adding (sr, sc) to the queue.



	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	.	.	#	.
2	(2,0)	#	.	.	.	.	.
3	(3,0)	.	#	#	.	.	.
4	#	.	#	E	.	#	.

Dungeon Problem Statement

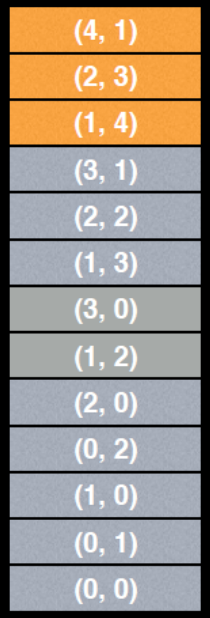
Start at the start node coordinate by adding (sr, sc) to the queue.

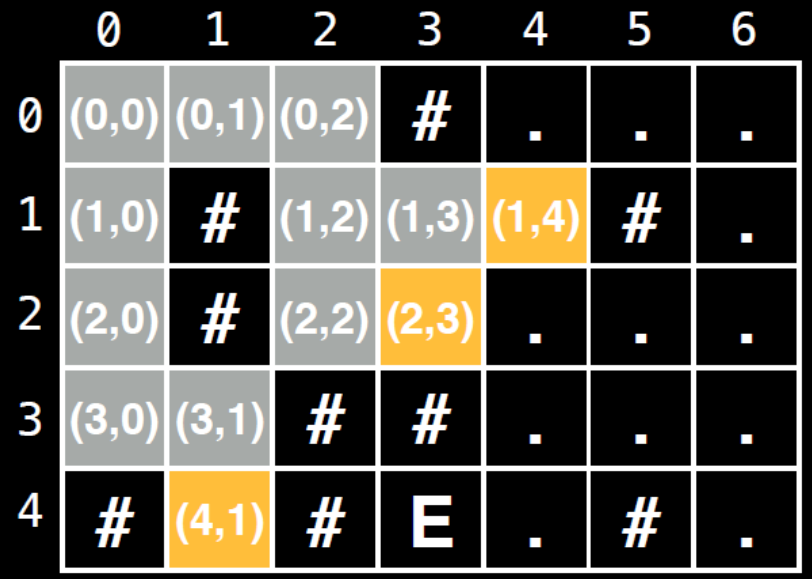


	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	.	.	.
1	(1,0)	#	(1,2)	(1,3)	.	#	.
2	(2,0)	#	(2,2)	.	.	.	.
3	(3,0)	(3,1)	#	#	.	.	.
4	#	.	#	E	.	#	.

Dungeon Problem Statement

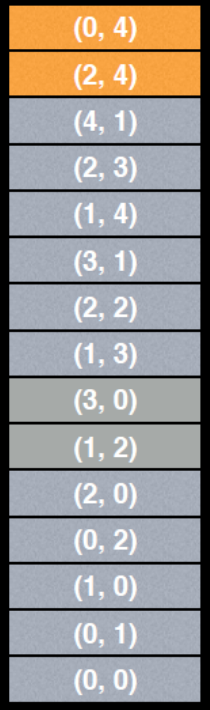
Start at the start node coordinate by adding (sr, sc) to the queue.

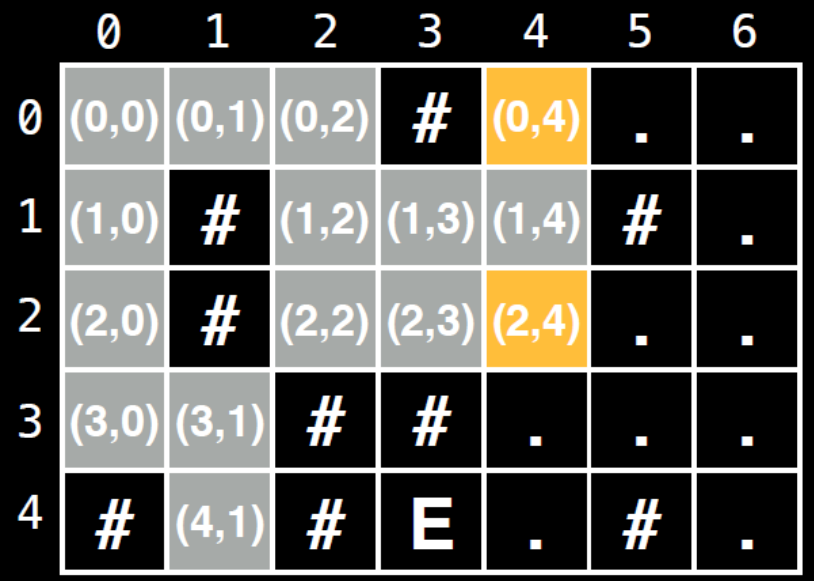




Dungeon Problem Statement

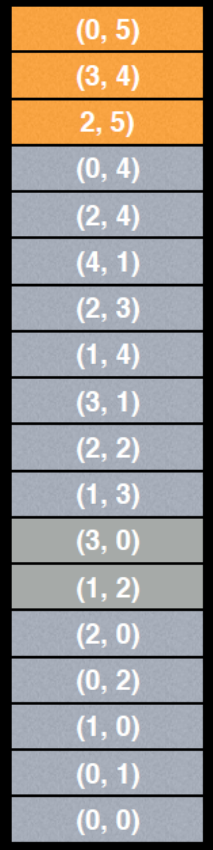
Start at the start node coordinate by adding (sr, sc) to the queue.





Dungeon Problem Statement

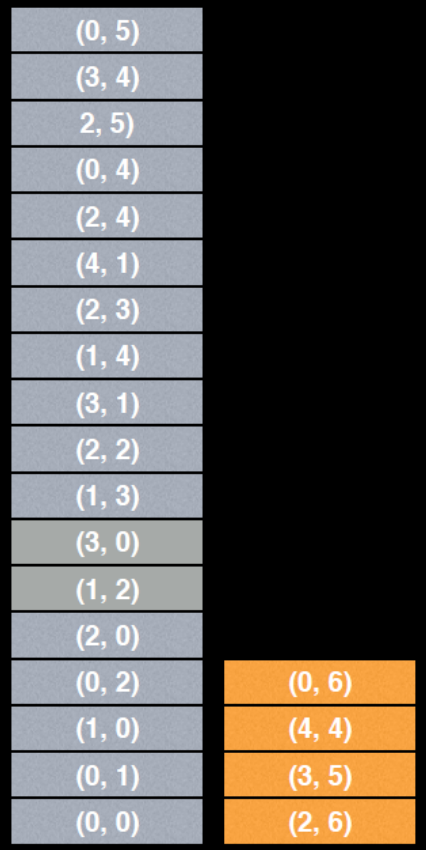
Start at the start node coordinate by adding (sr, sc) to the queue.



	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	.
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	.
3	(3,0)	(3,1)	#	#	(3,4)	.	.
4	#	(4,1)	#	E	.	#	.

Dungeon Problem Statement

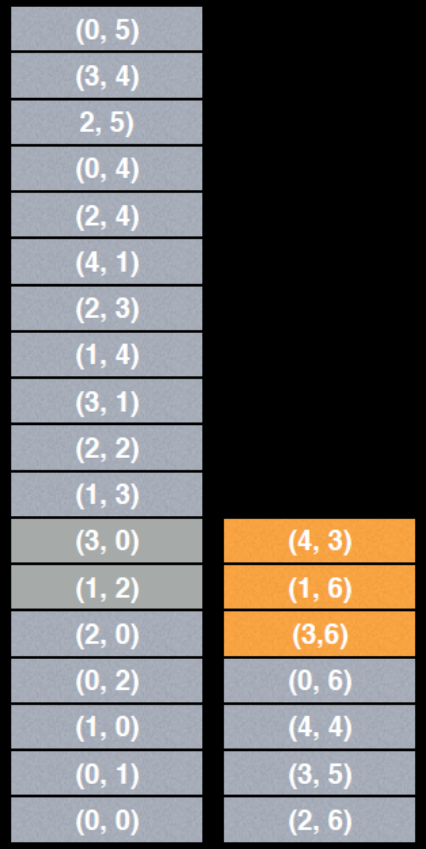
Start at the start node coordinate by adding (sr, sc) to the queue.



	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	.
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	.
4	#	(4,1)	#	E	(4,4)	#	.

Dungeon Problem Statement

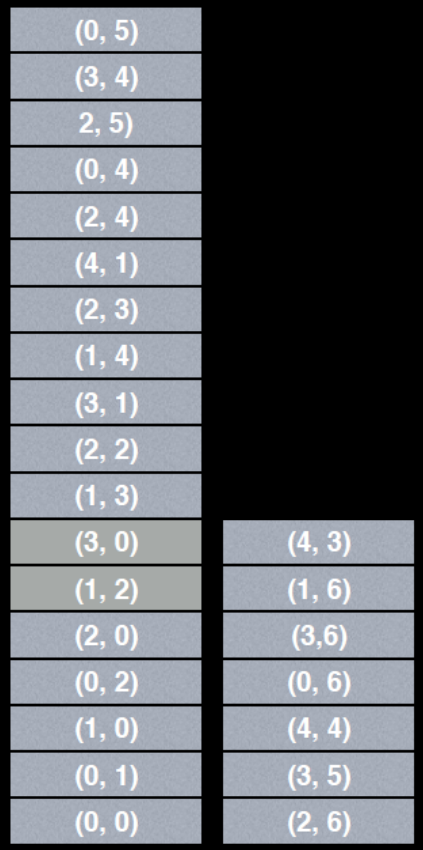
Start at the start node coordinate by adding (sr, sc) to the queue.

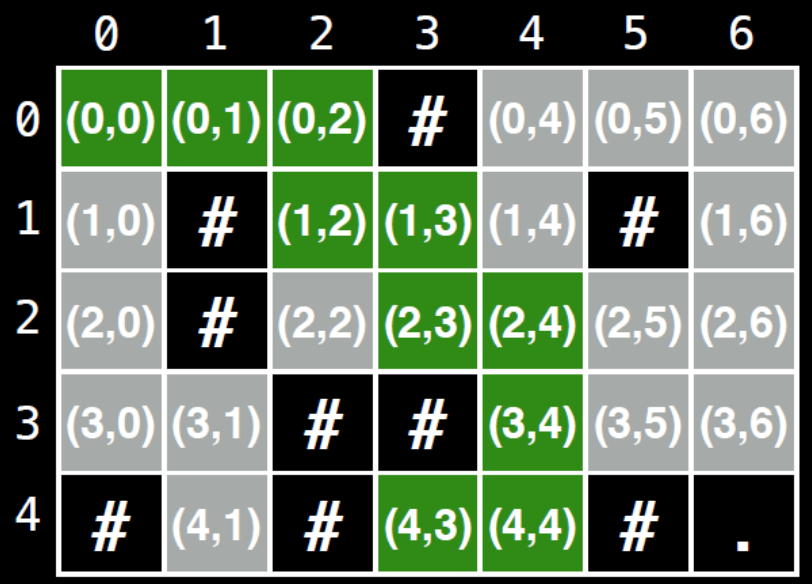


	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	#	(0,4)	(0,5)	(0,6)
1	(1,0)	#	(1,2)	(1,3)	(1,4)	#	(1,6)
2	(2,0)	#	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	#	#	(3,4)	(3,5)	(3,6)
4	#	(4,1)	#	(4,3)	(4,4)	#	.

Dungeon Problem Statement

We have reached the end, and if we had a 2D prev matrix we could regenerate the path by retracing our steps.





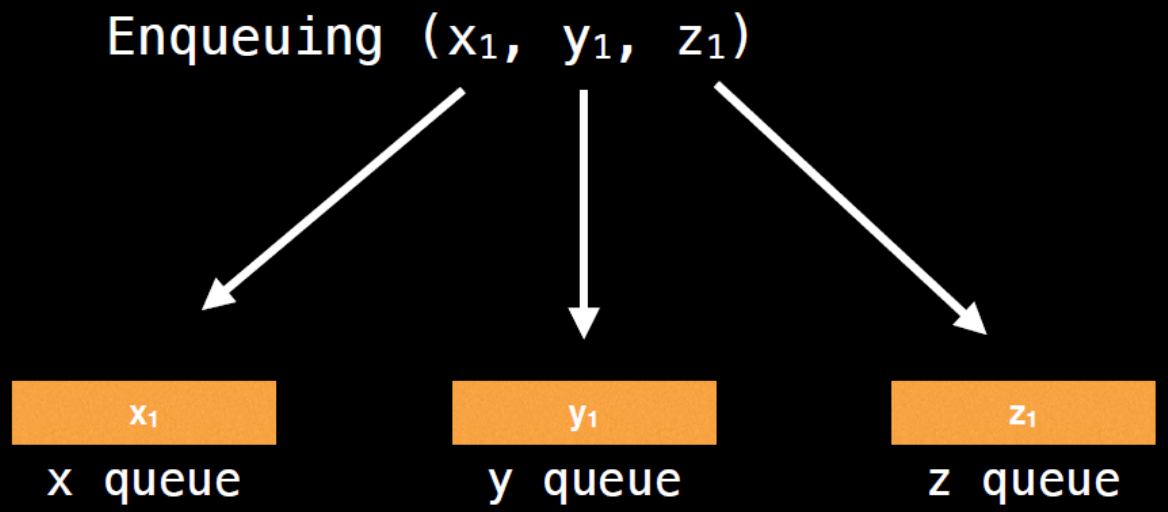
Alternative State representation

So far we have been storing the next x-y position in the queue as an (x, y) pair. This works well but requires either an array or an object wrapper to store the coordinate values. In practice, this requires a lot of packing and unpacking of values to and from the queue.

Let’s take a look at an alternative approach which also scales well in higher dimensions and (IMHO) requires less setup effort…

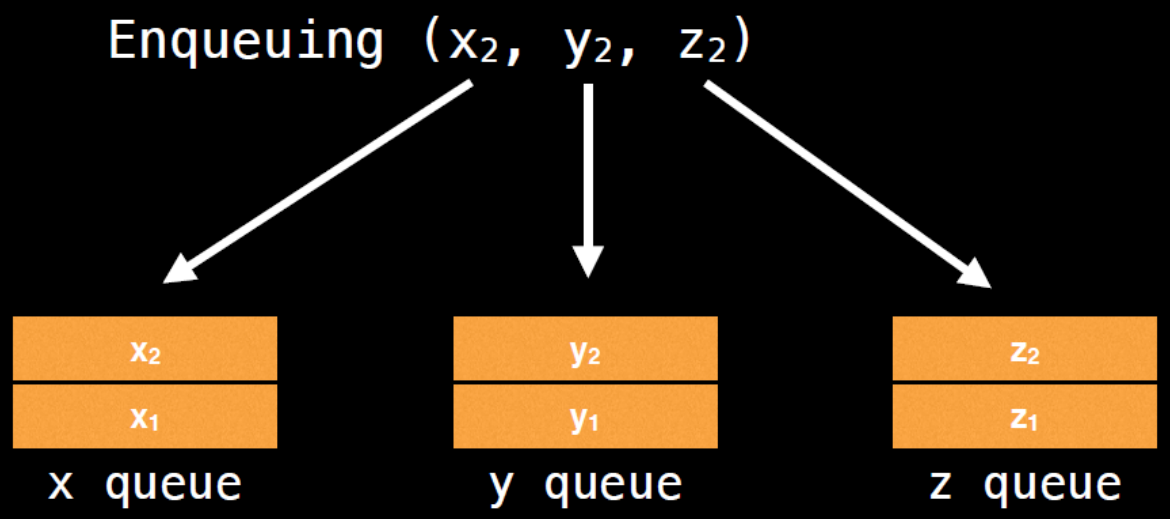
Alternative State representation

An alternative approach is to use one queue for each dimension, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.



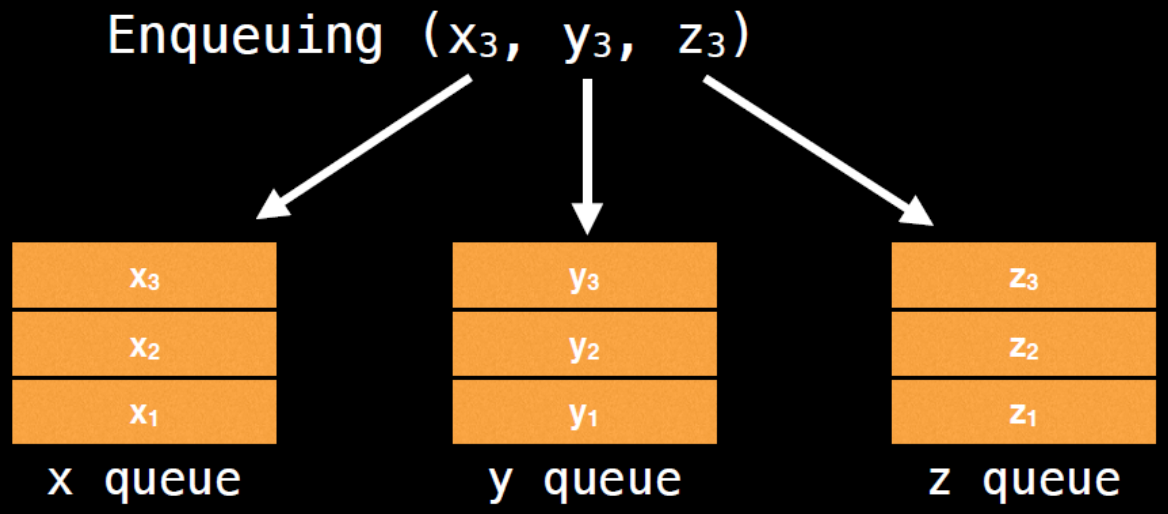
Alternative State representation

An alternative approach is to use one queue for each dimension, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.



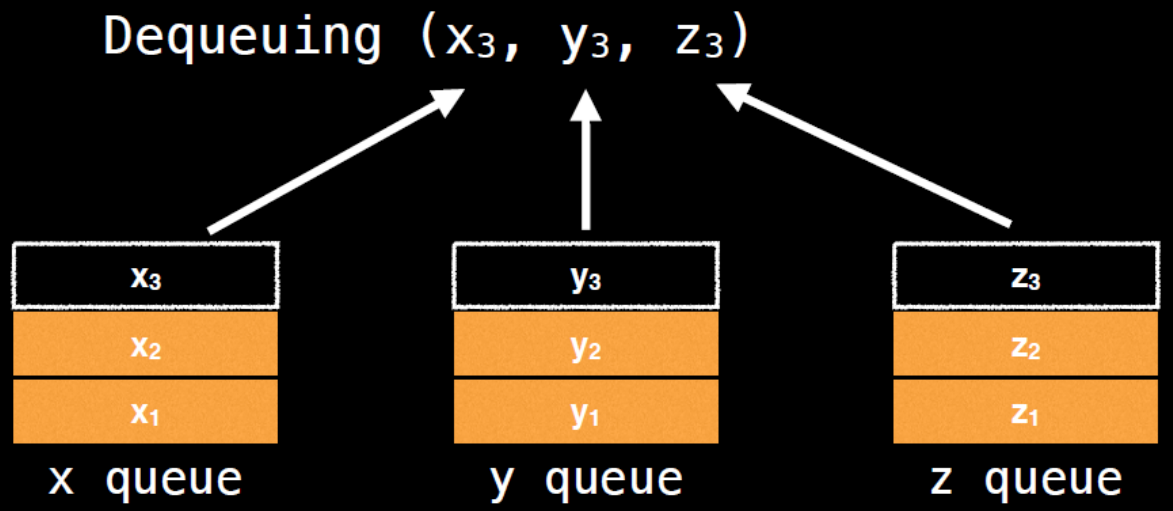
Alternative State representation

An alternative approach is to use one queue for each dimension, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.



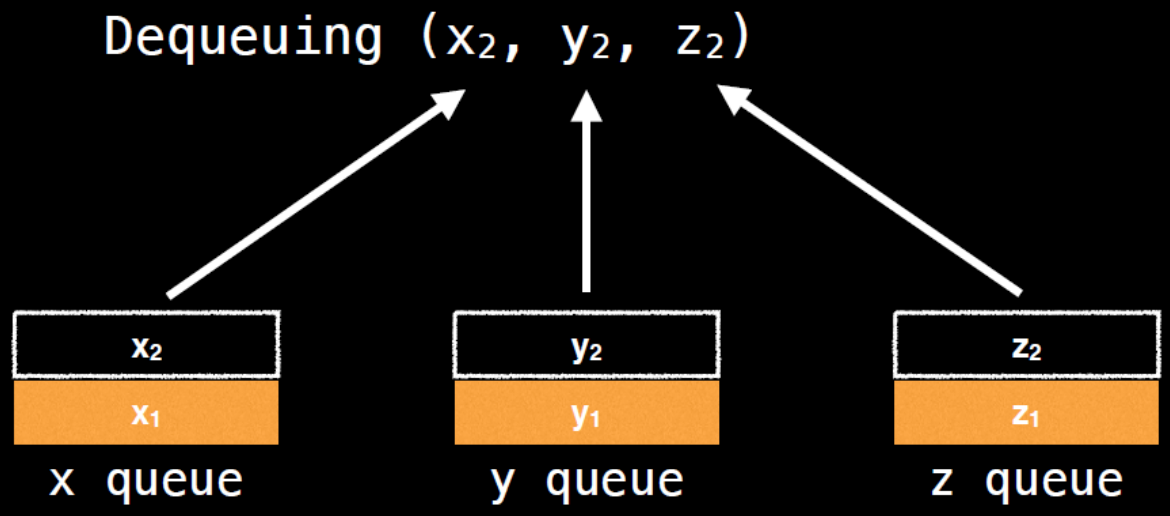
Alternative State representation

An alternative approach is to use one queue for each dimension, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.



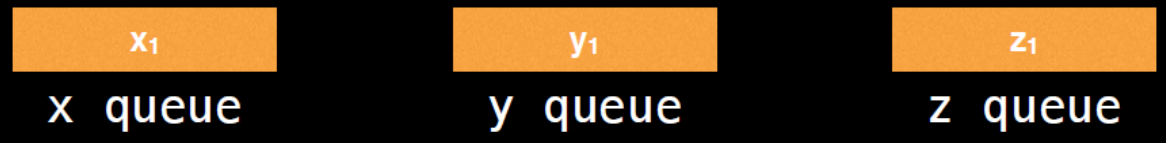
Alternative State representation

An alternative approach is to use one queue for each dimension, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.



Alternative State representation

An alternative approach is to use one queue for each dimension, so in a 3D grid you would have one queue for each of the x, y, and z dimensions.



Alternative State representation

```
# Global/class scope variables
R, C = … # R = number of rows, C = number of columns
m = …    # Input character matrix of size R x C
sr, sc = … # 'S' symbol row and column values
rq, cq = … # Empty Row Queue (RQ) and Column Queue (CQ)

# Variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0

# Variable used to track whether the 'E' character
# ever gets reached during the BFS.
reached_end = false

# R x C matrix of false values used to track whether
# the node at position (i, j) has been visited.
visited = …

# North, south, east, west direction vectors.
dr = [-1, +1,  0,  0]
dc = [ 0,  0, +1, -1]
```

Alternative State representation

```
function solve():
  rq.enqueue(sr)
  cq.enqueue(sc)
  visited[sr][sc] = true
  while rq.size() > 0: # or cq.size() > 0
    r = rq.dequeue()
    c = cq.dequeue()
    if m[r][c] == 'E':
      reached_end = true
      break
    explore_neighbours(r, c)
    nodes_left_in_layer--
    if nodes_left_in_layer == 0:
      nodes_left_in_layer = nodes_in_next_layer
      nodes_in_next_layer = 0
      move_count++
  if reached_end:
    return move_count
  return -1
```

Alternative State representation

```
function explore_neighbours(r, c):
  for(i = 0; i < 4; i++):
    rr = r + dr[i]
    cc = c + dc[i]

    # Skip out of bounds locations
    if rr < 0 or cc < 0: continue
    if rr >= R or cc >= C: continue

    # Skip visited locations or blocked cells
    if visited[rr][cc]: continue
    if m[rr][cc] == '#': continue

    rq.enqueue(rr)
    cq.enqueue(cc)
    visited[rr][cc] = true
    nodes_in_next_layer++
```

Summary

Representing a grid as an adjacency list and adjacency matrix.

Empty Grid

0	1
2	3
4	5

Adjacency List:

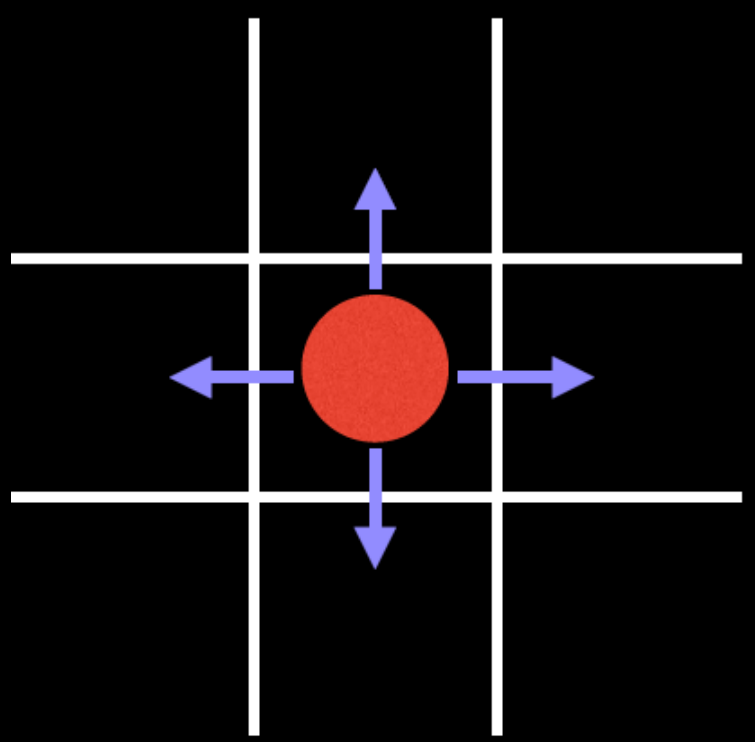
```
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 3, 4]
3 -> [1, 2, 5]
4 -> [2, 5]
5 -> [3, 4]
```

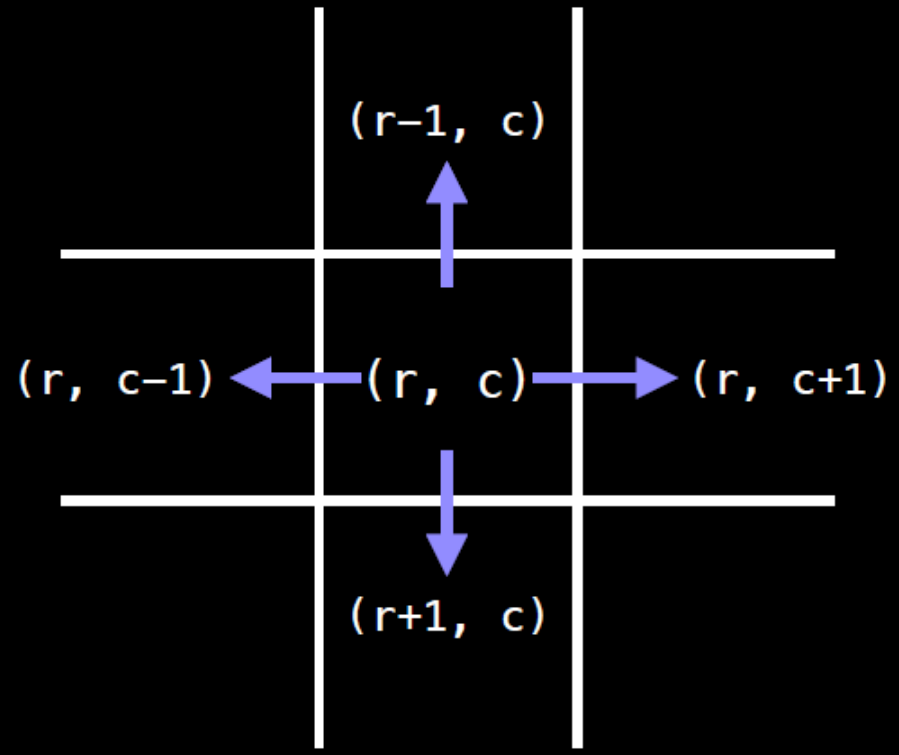
Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

Summary

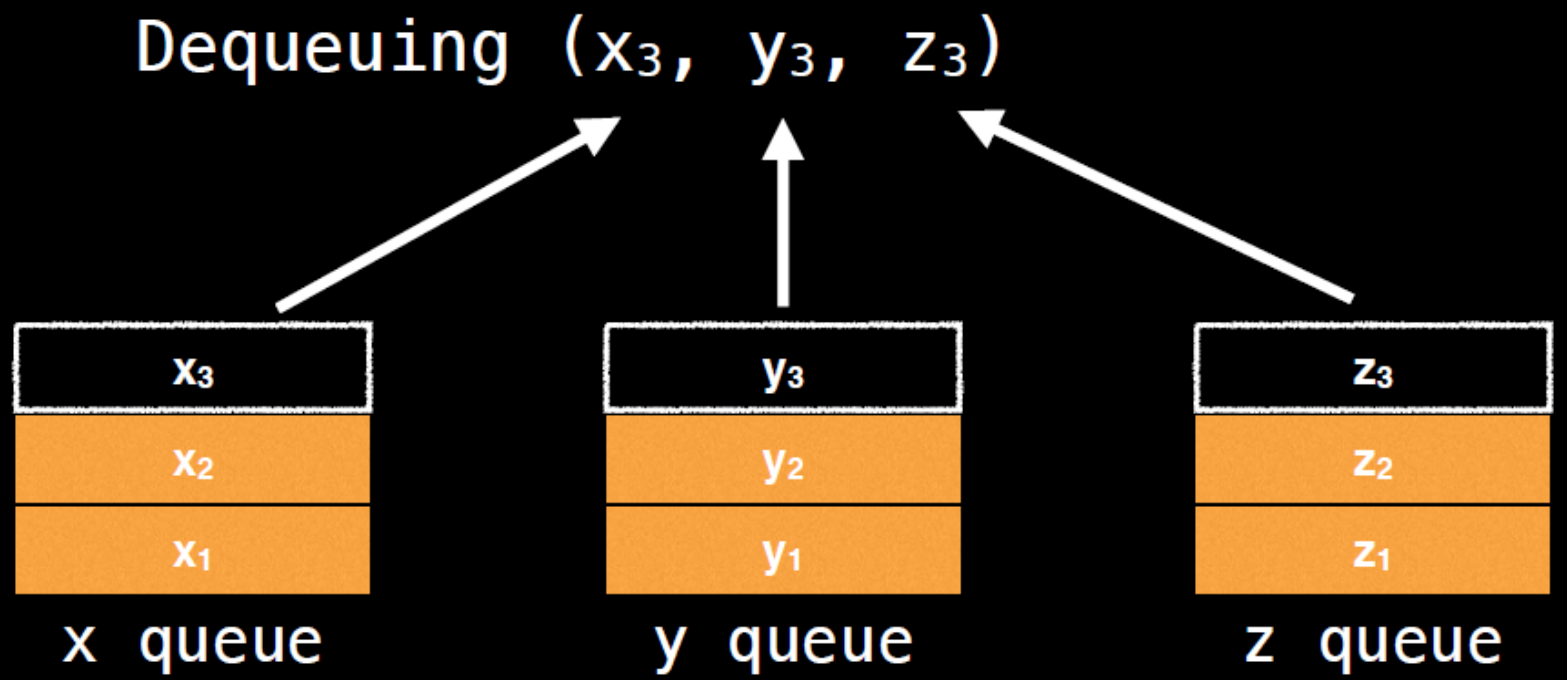
Using direction vectors to visit neighbouring cells.





Summary

Explored an alternative way to represent multi-dimensional coordinates using multiple queues.



Summary

How to use BFS on a grid to find the shortest path between two cells.