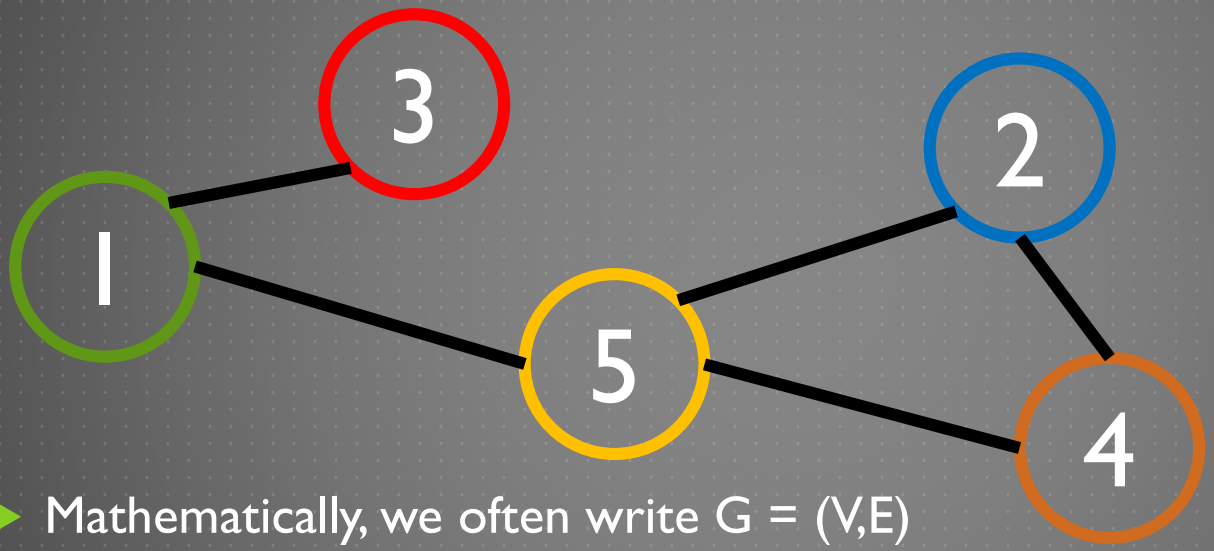
GRAPH

Yik Wai Pan

GRAPH

- A set of vertices (or nodes) linked by edges



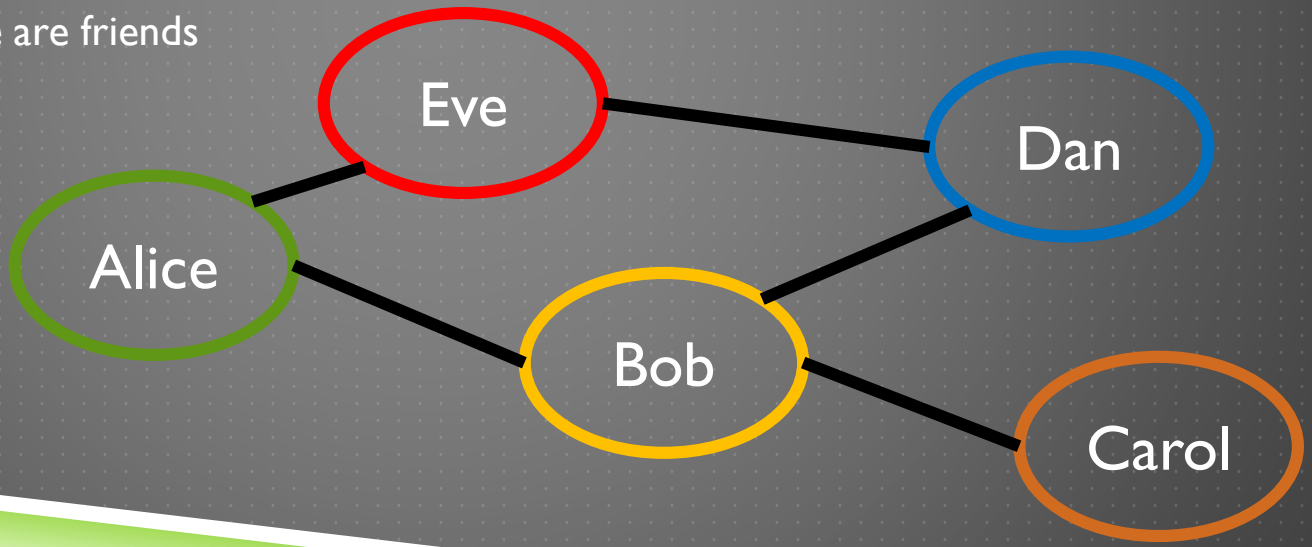
- Mathematically, we often write G = (V,E)
  - V: set of vertices, so |V| = number of vertices
  - E: set of edges, so |E| = number of edges

USAGE

- To present the relationships between different objects/elements in a mathematical way
- Examples:
  - Social Networks
  - Maps
  - Grids
  - States

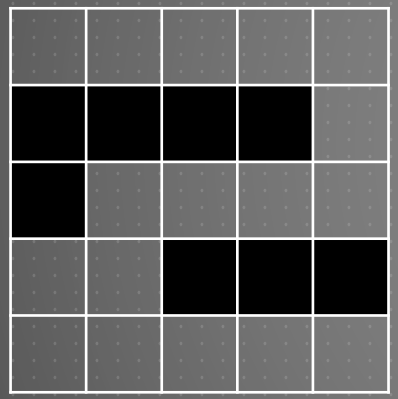
USAGE - EXAMPLE

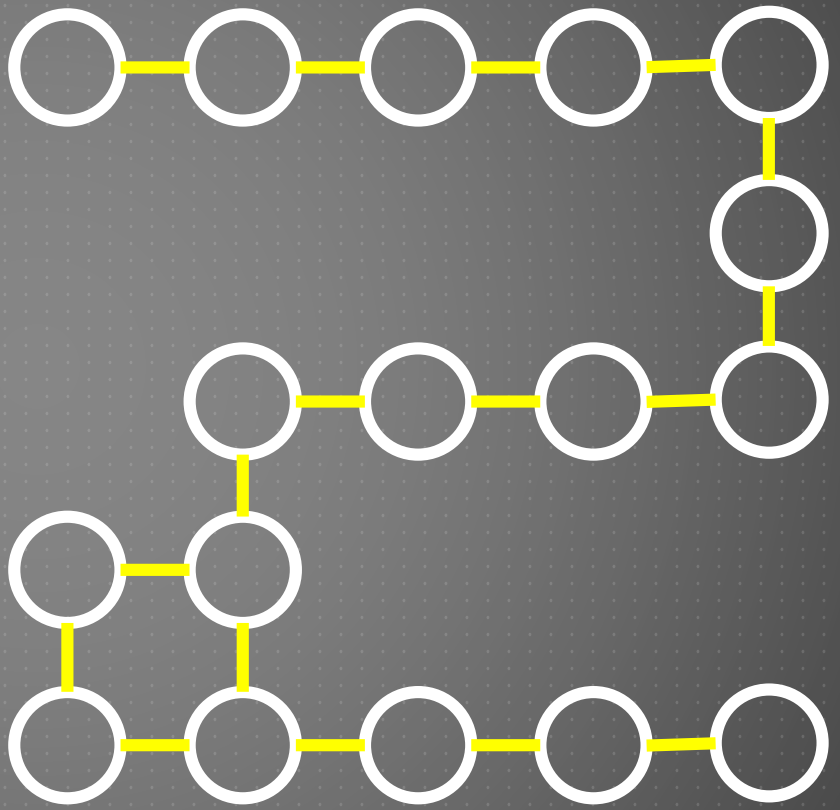
- Social Networks
  - Alice and Bob are friends
  - Bob and Carol are friends
  - Bob and Dan are friends
  - Dan and Eve are friends
  - Alice and Eve are friends



USAGE - EXAMPLE

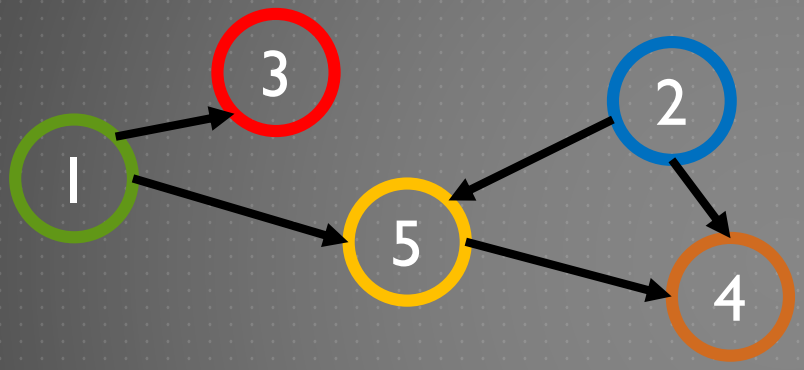
- Maze



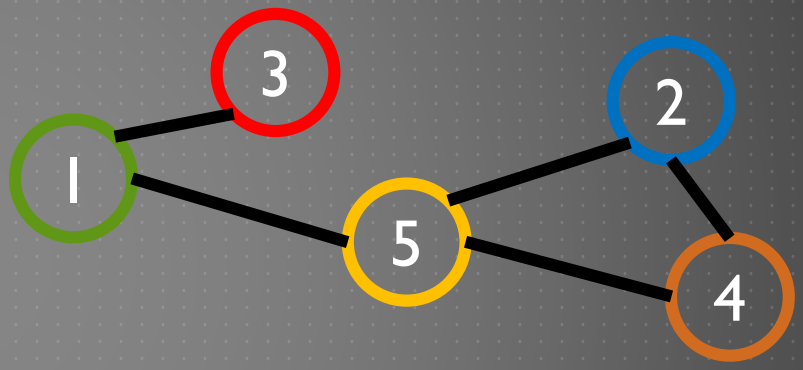


GRAPH

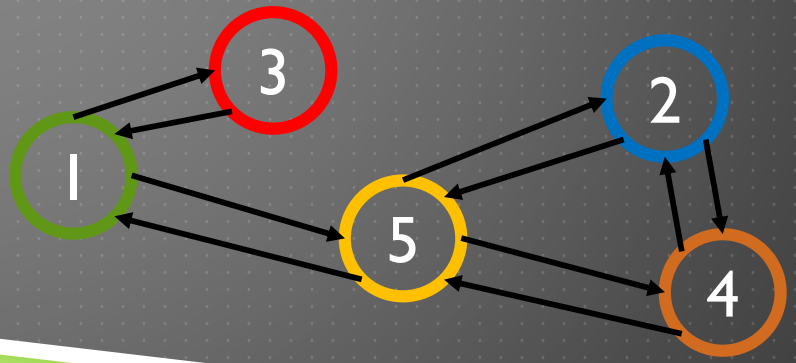
Directed graph:



Undirected graph:

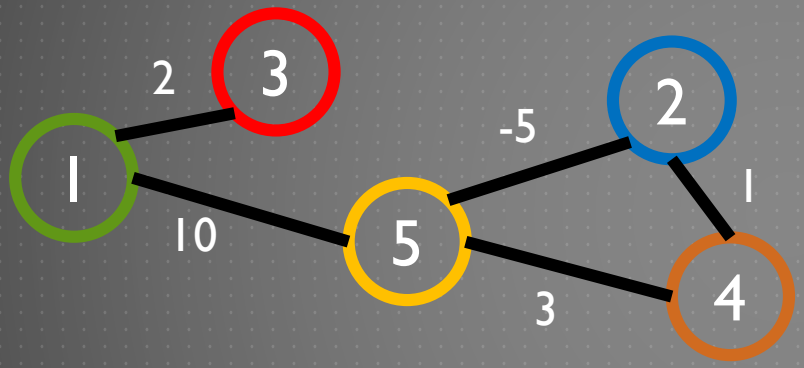


- You may use two directed edges to represent an undirected edge

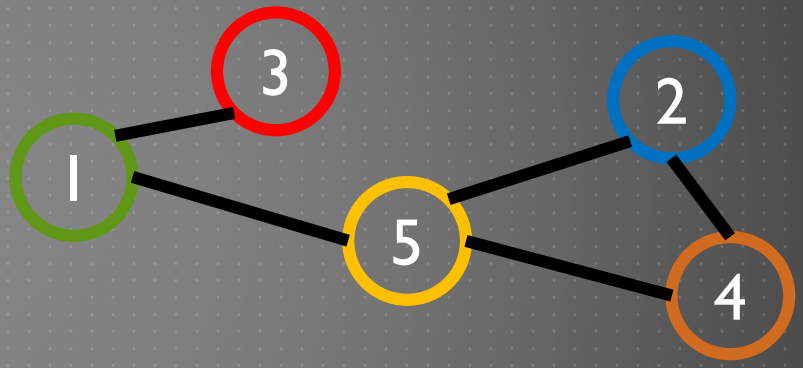


GRAPH

Weighted graph:



Unweighted graph:



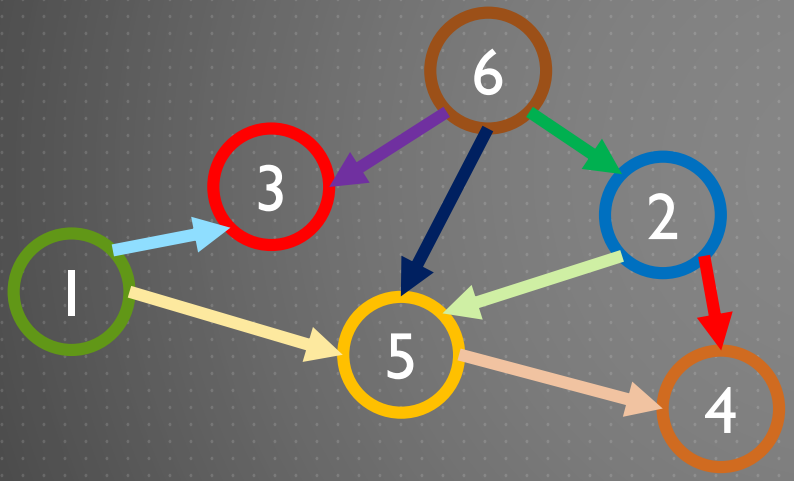
- You may treat unweighted edges to be weighted edges of equal weights

CONTENT

- Graph Adjacency Representation
- Special graphs
- Graph Traversal
  - Depth-First Search
  - Breadth-First Search
- Topological Sort

ADJACENCY MATRIX

- Use a 2D array to store the edges
- A[i][j] = 0 if there are no edges from vertex i to vertex j
- A[i][j] = 1 if there are edges from vertex i to vertex j

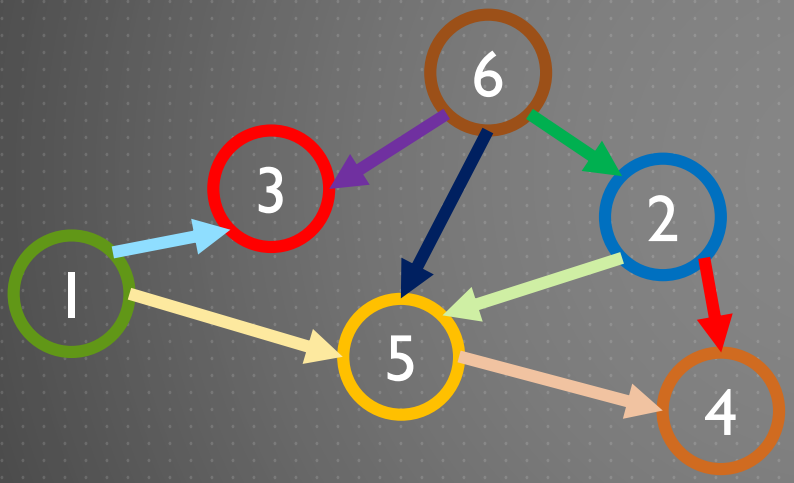


A	1	2	3	4	5	6
1	0	0	1	0	1	0
2	0	0	0	1	1	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	1	0	0
6	0	1	1	0	1	0

- Memory Complexity : $O(|V|^2)$

EDGE LIST

- A list of edges

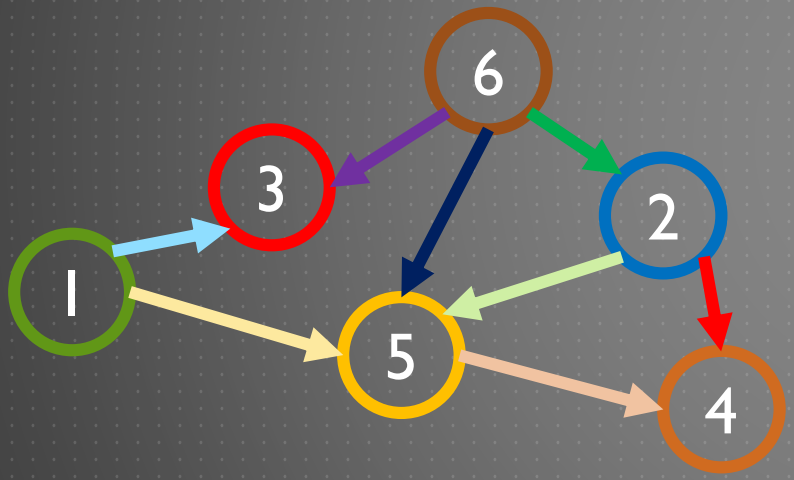


- Memory Complexity : O(|E|)

id	x	y
0	1	3
1	6	3
2	1	5
3	2	5
4	2	4
5	6	2
6	5	4
7	6	5

EDGE LIST

- You may sort the edges in ascending order of x so that all adjacency nodes of a given node can be easily found



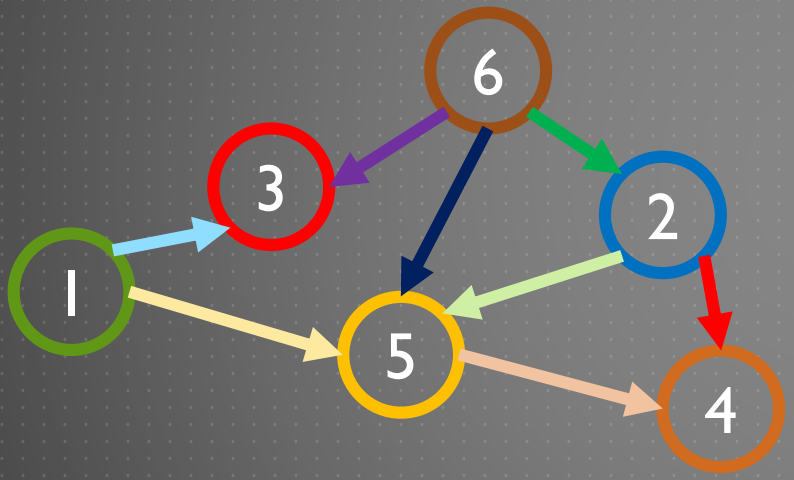
node	first	last
1	0	1
2	2	3
3	4	3
4	4	3
5	4	4
6	5	7

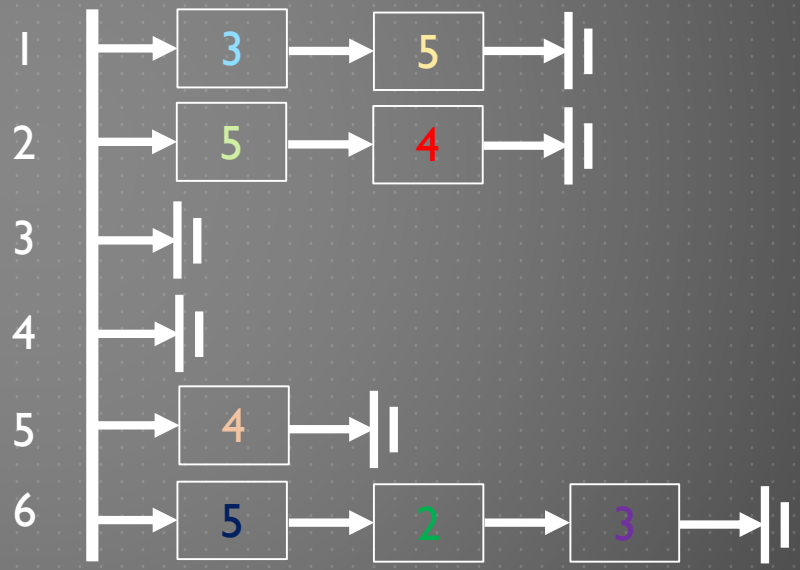
id	x	y
0	1	3
1	1	5
2	2	5
3	2	4
4	5	4
5	6	3
6	6	5
7	6	2

- Time Complexity : O(|E| log |E|)

ADJACENCY LISTS

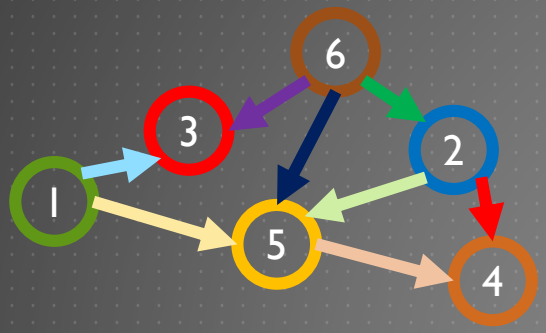
- Use |V| linked list
- The i-th linked list stores the vertices connecting to vertex i

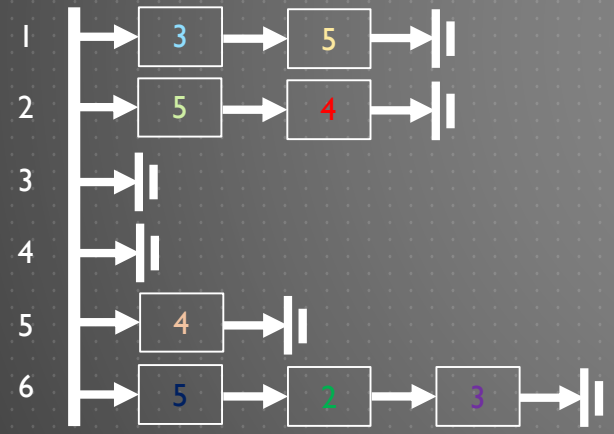




- Memory Complexity : O(|E|)

ADJACENCY LISTS-IMPLEMENTATION



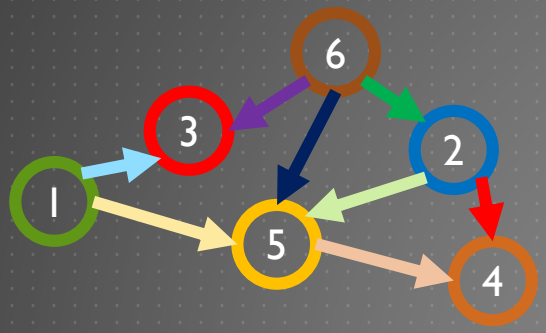


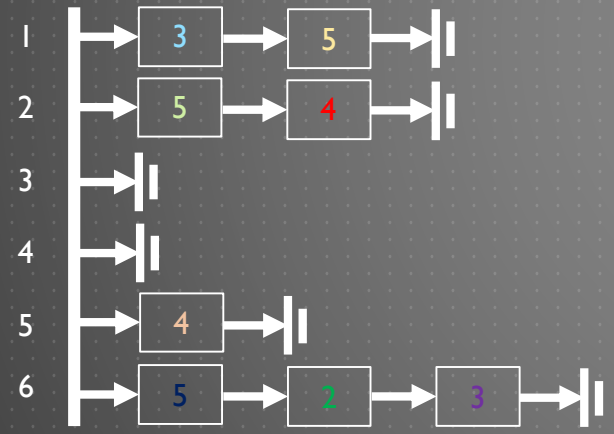
node	first
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1

id	y	next

```
void initialize(){
    for (int i=1;i<=n;i++) first[i]=-1;
}
```

ADJACENCY LISTS-IMPLEMENTATION





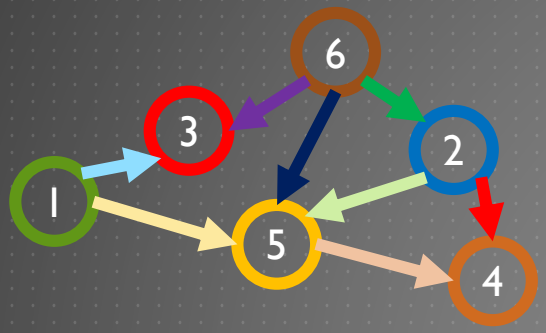
- 1->3

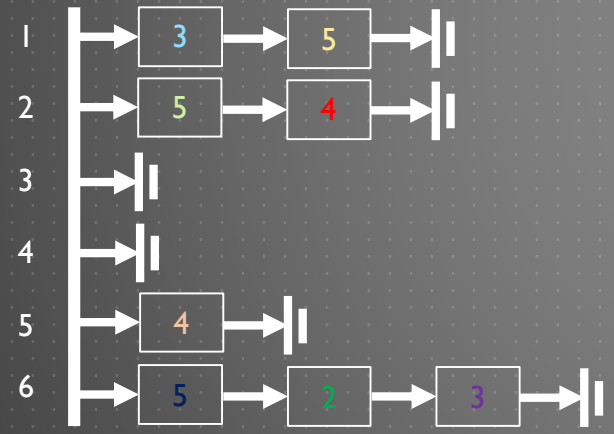
node	first
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1

id	y	next
0	3	

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





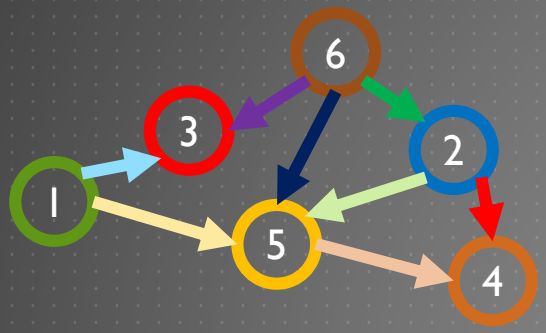
- 6->3

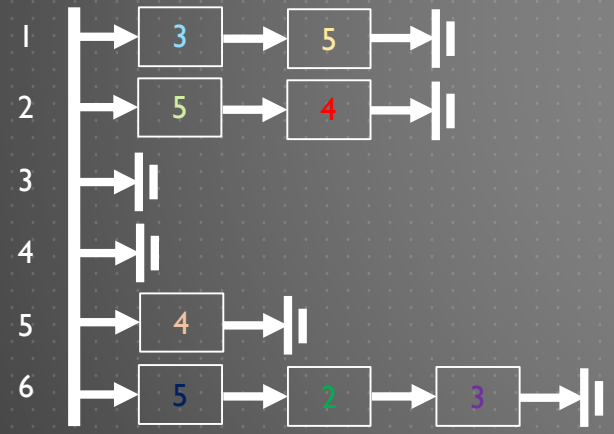
node	first
1	0
2	-1
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





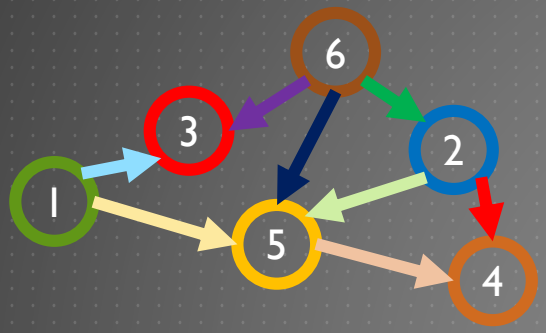
- 1->5

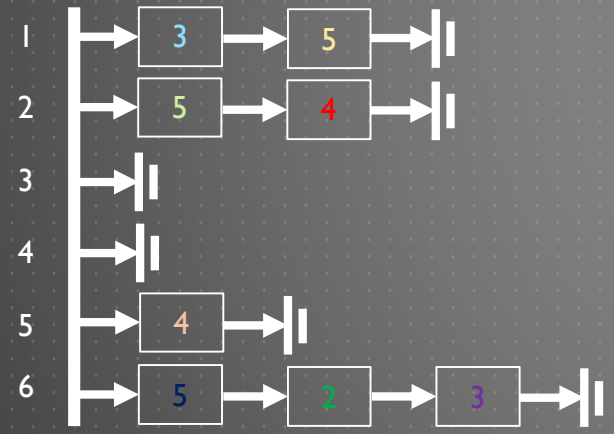
node	first
1	0
2	-1
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





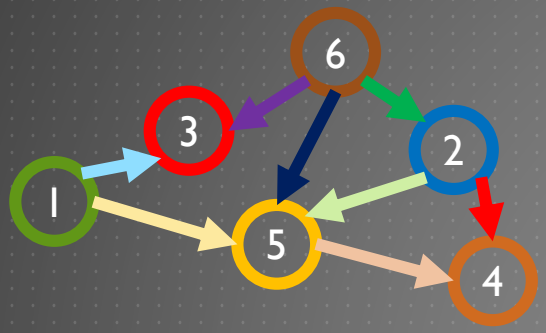
- 2->5

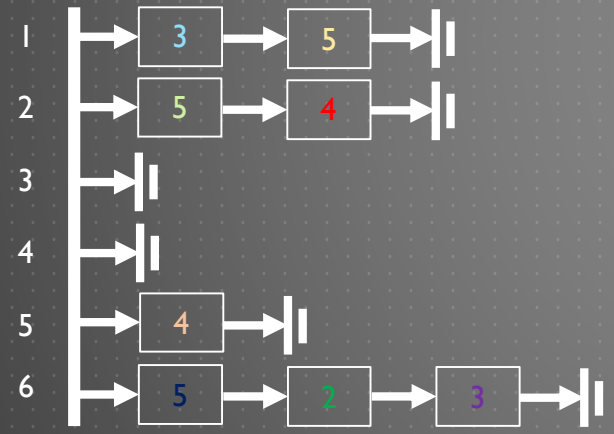
node	first
1	2
2	3
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





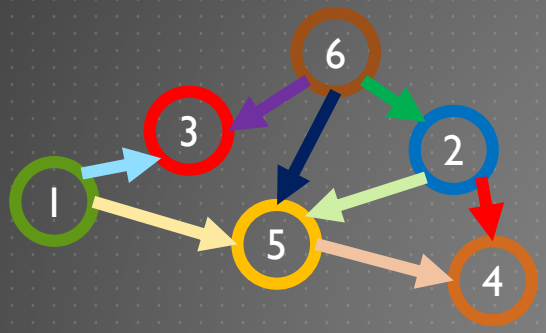
- 2->4

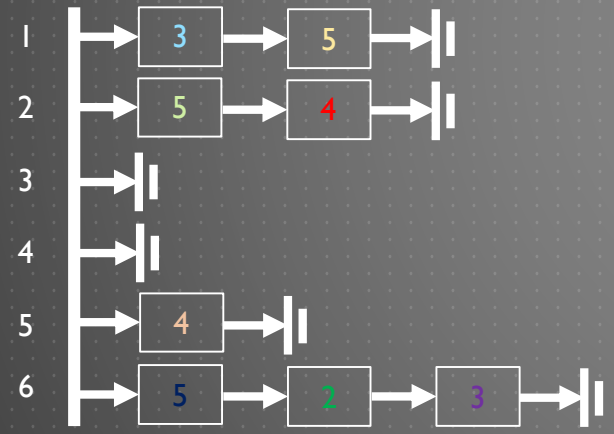
node	first
1	2
2	4
3	-1
4	-1
5	-1
6	1

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





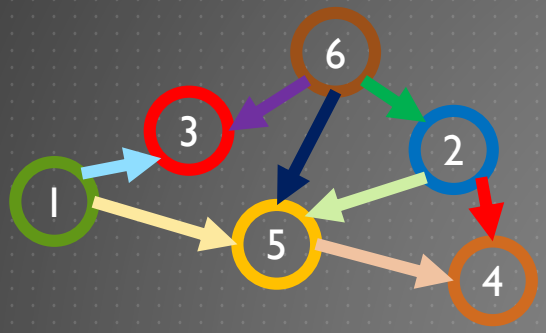
- 6->2

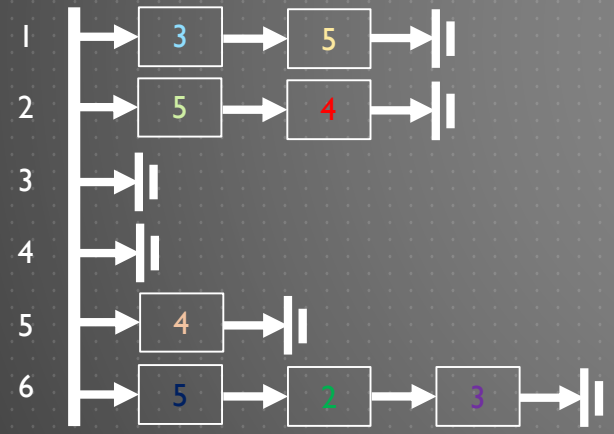
node	first
1	2
2	4
3	-1
4	-1
5	-1
6	5

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





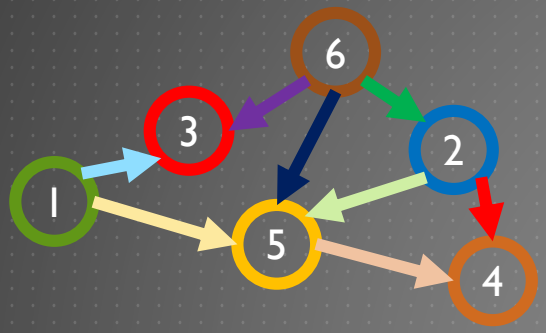
- 5->4

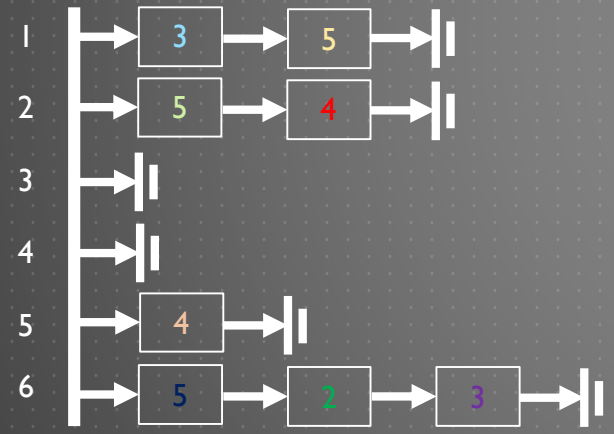
node	first
1	2
2	4
3	-1
4	-1
5	-1
6	5

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1
6	4	-1

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





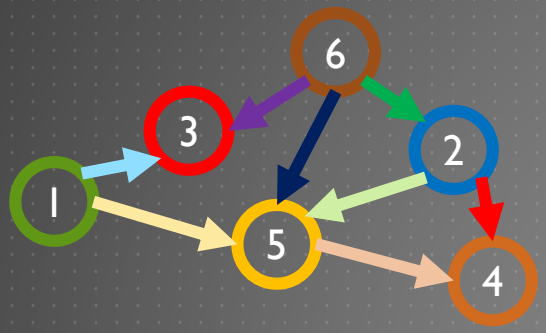
- 6->5

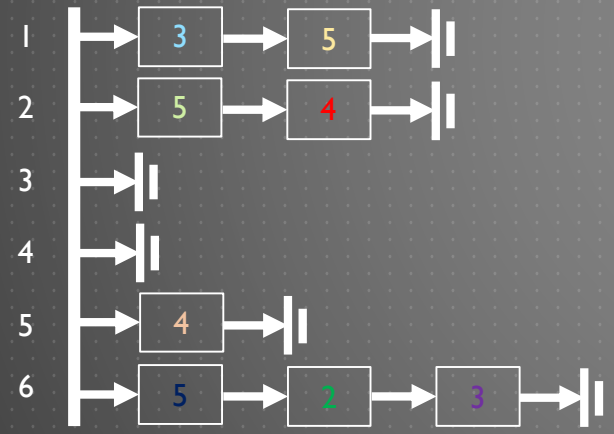
node	first
1	2
2	4
3	-1
4	-1
5	6
6	7

id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1
6	4	-1
7	5	5

```
void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```

ADJACENCY LISTS-IMPLEMENTATION





- Query 6's neighbors

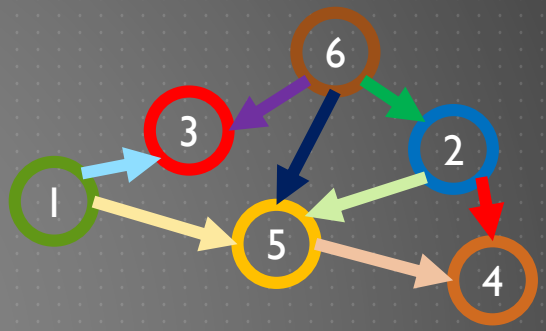
node	first
1	2
2	4
3	-1
4	-1
5	6
6	7

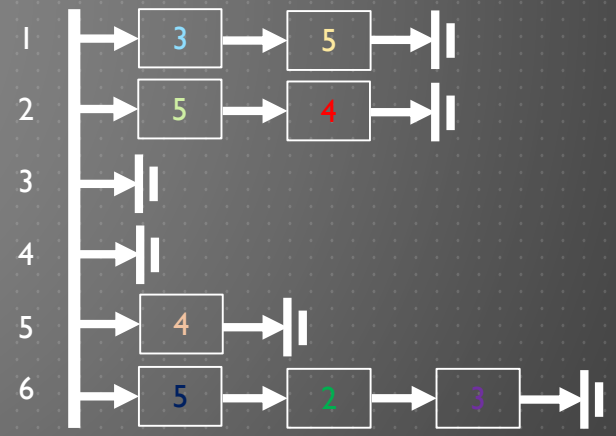
id	y	next
0	3	-1
1	3	-1
2	5	0
3	5	-1
4	4	3
5	2	1
6	4	-1
7	5	5

```
void query(int x){
    //Querying all adjacent nodes of a given node x
    for (int i=first[x];i!=-1;i=next[i])
        printf("%d\n",v[i]);
}
```

ADJACENCY LISTS-IMPLEMENTATION

```
int v[10000],w[10000],next[10000];
int first[1000];
int n,m=0;

void initialize(){
    for (int i=1;i<=n;i++) first[i]=-1;
}

void new_edge(int from,int to,int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}

void query(int x){
    //Querying all adjacent nodes of a given node x
    for (int i=first[x];i!=-1;i=next[i])
        printf("%d\n",v[i]);
}
```

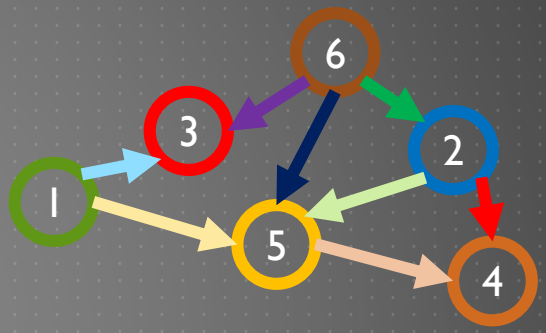


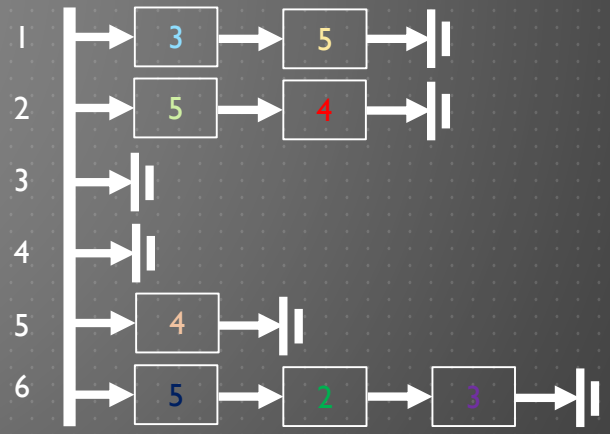


ADJACENCY LISTS-IMPLEMENTATION

- We can use vector in C++ to implement the lists

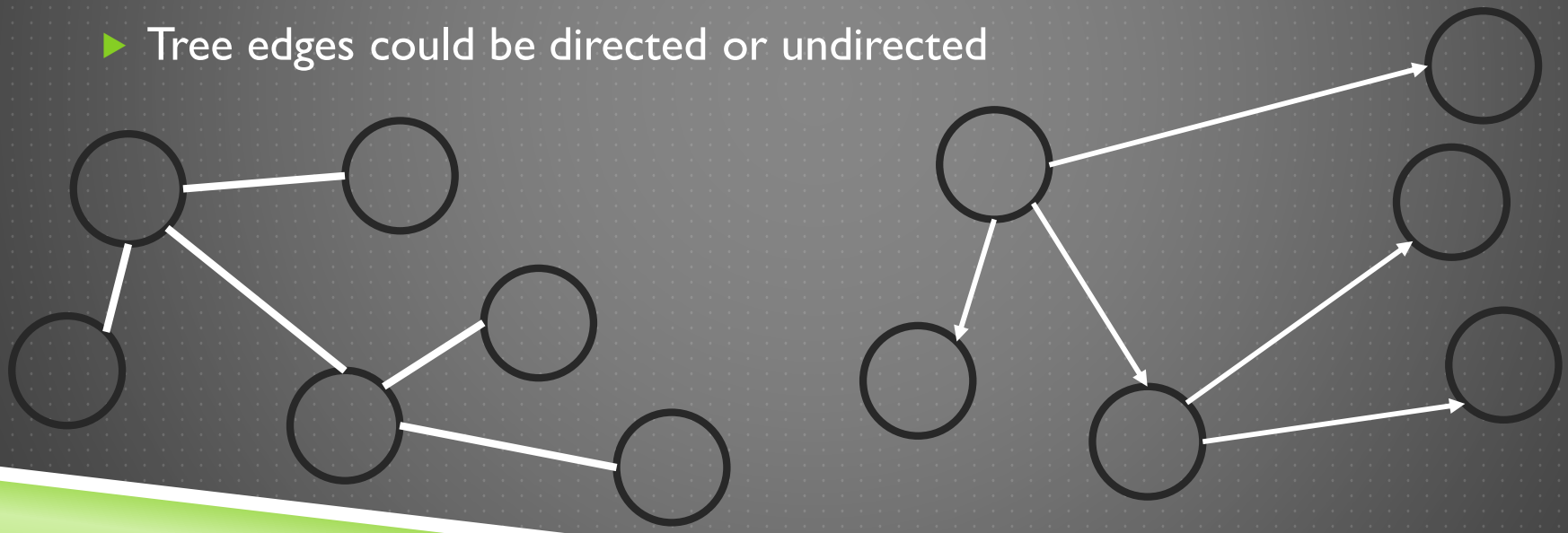
```
#include<vector>
using namespace std;

int n,m;
vector<int> v[10000],w[10000];
void new_edge(int from,int to,int weight){
    v[from].push_back(to);
    w[from].push_back(weight);
}

void query(int x){
    //Querying all adjacent nodes of a given node (x)
    for (int i=0;i<v[x].size();i++)
        printf("%d\n",v[x][i]);
}
```





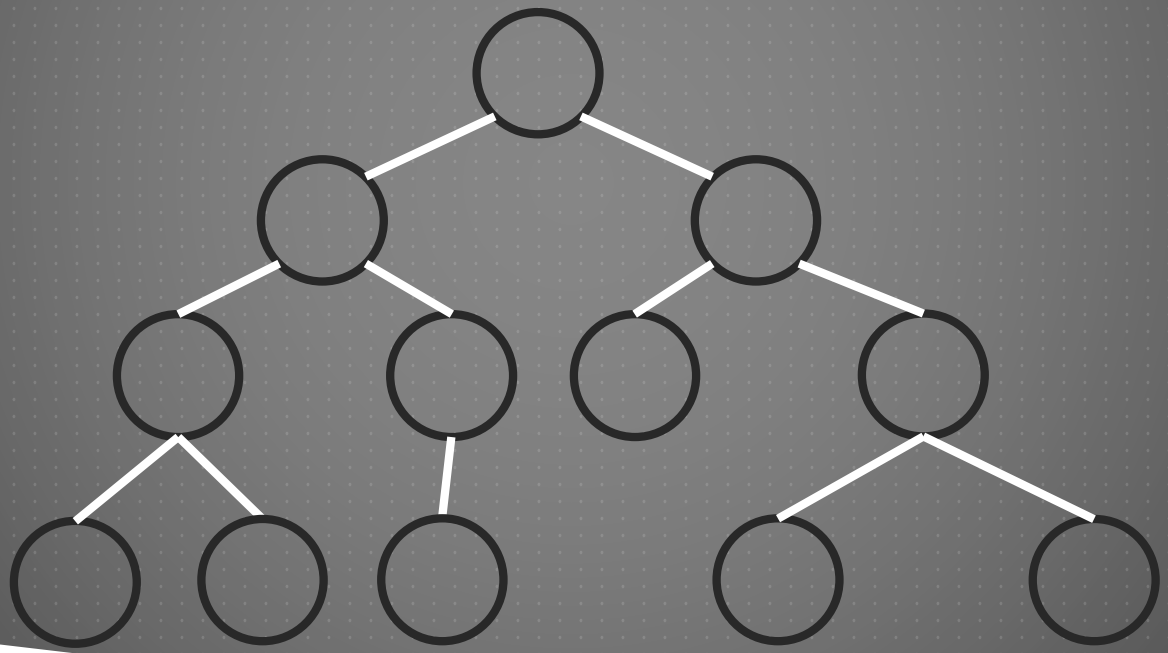
SPECIAL GRAPHS - TREES

- 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
- Tree edges could be directed or undirected



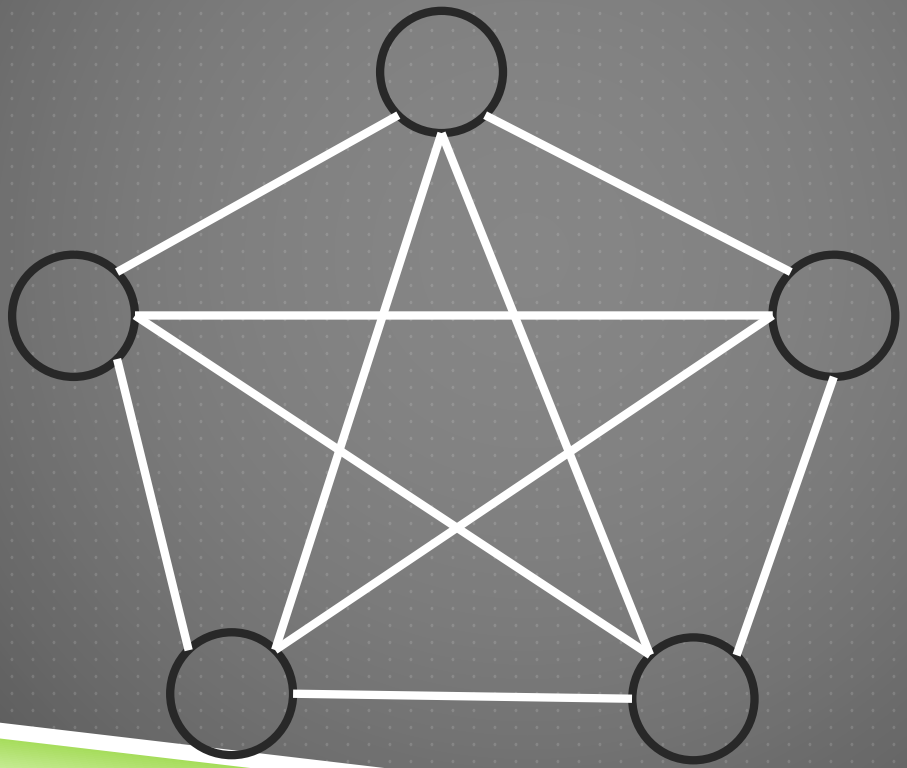
SPECIAL GRAPHS – BINARY TREES

- A tree in which each node has at most two children
- Commonly used as other data structures



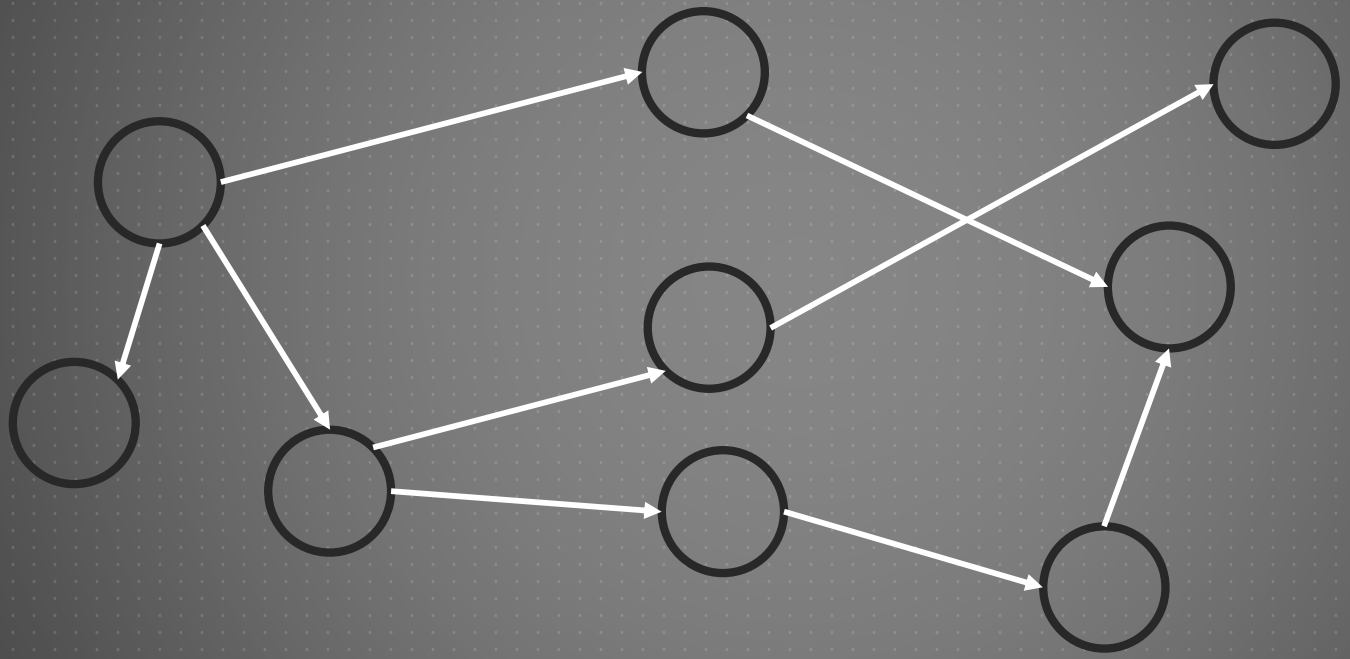
SPECIAL GRAPHS - COMPLETE GRAPH

- Every pair of distinct vertices is connected by a unique edge



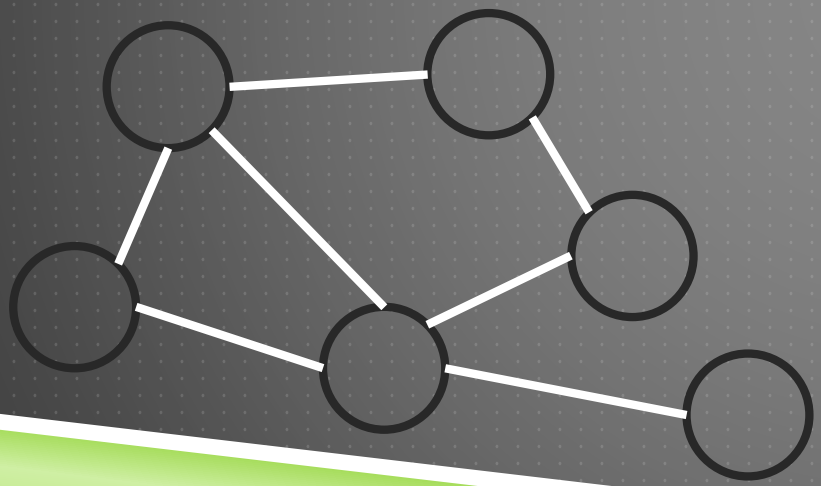
SPECIAL GRAPHS – DIRECTED ACYCLIC GRAPHS (DAG)

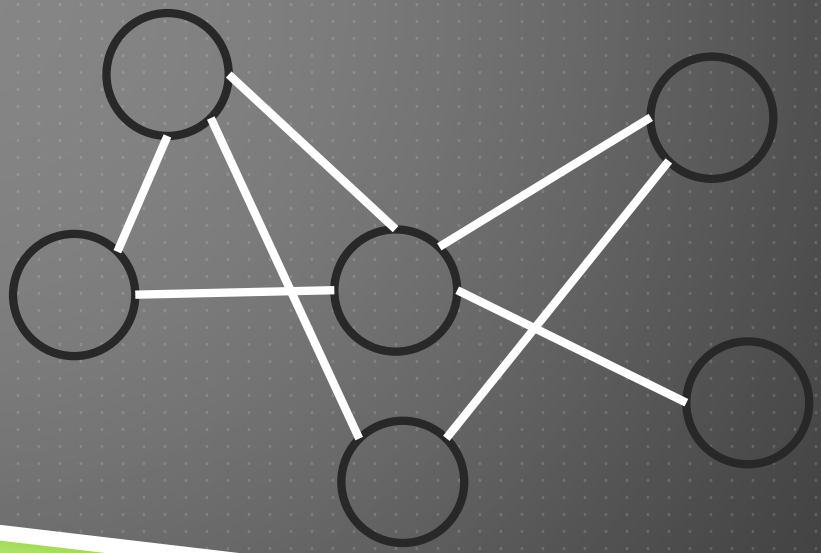
- A directed graph with no directed cycles



SPECIAL GRAPHS – PLANAR GRAPHS

- A graph whose vertices and edges can be drawn in a plane such that no two of the edges intersect
- More efficient algorithms can be applied on planar graphs





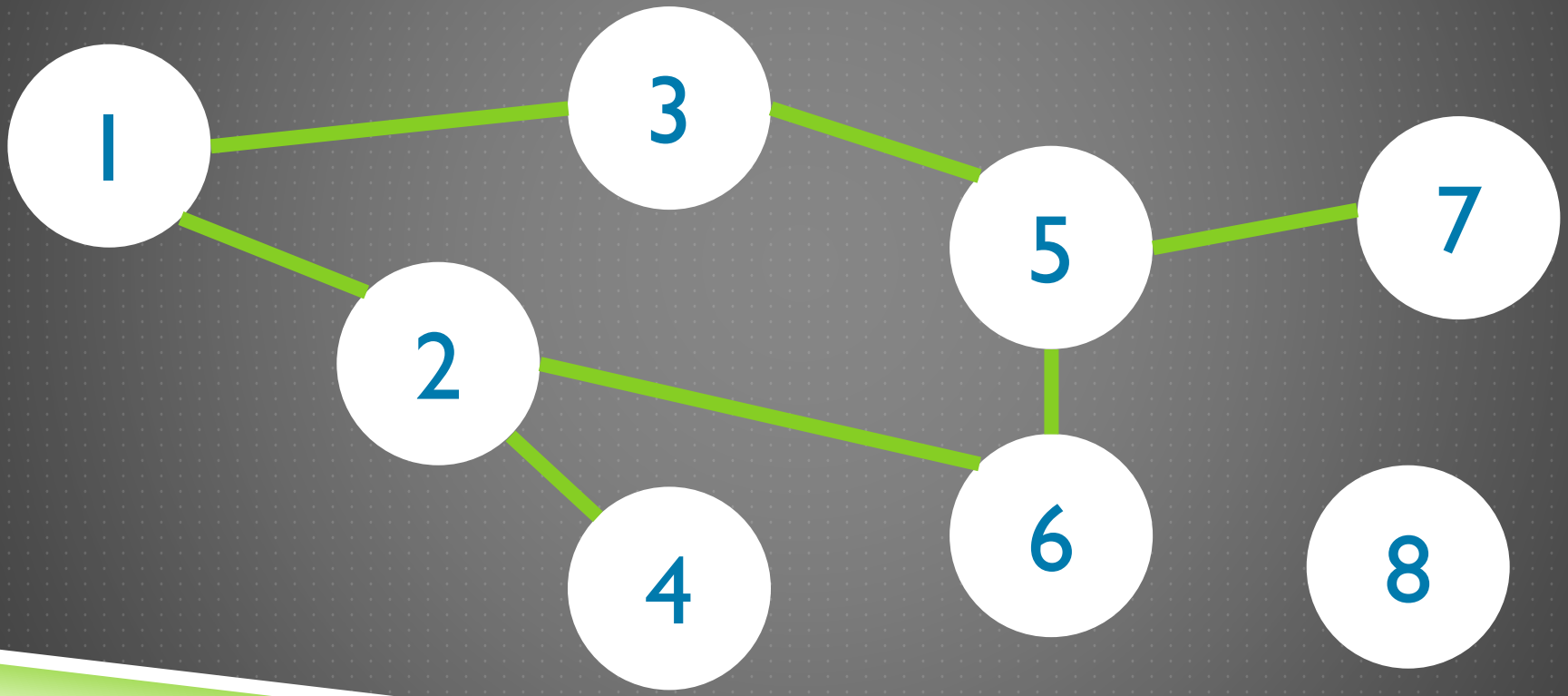
GRAPH TRAVERSAL

- The process of visiting (checking and/or updating) each vertex in a graph.
- Classified by the order in which the vertices are visited
  - Depth-first search
  - Breadth-first search

DEPTH-FIRST SEARCH

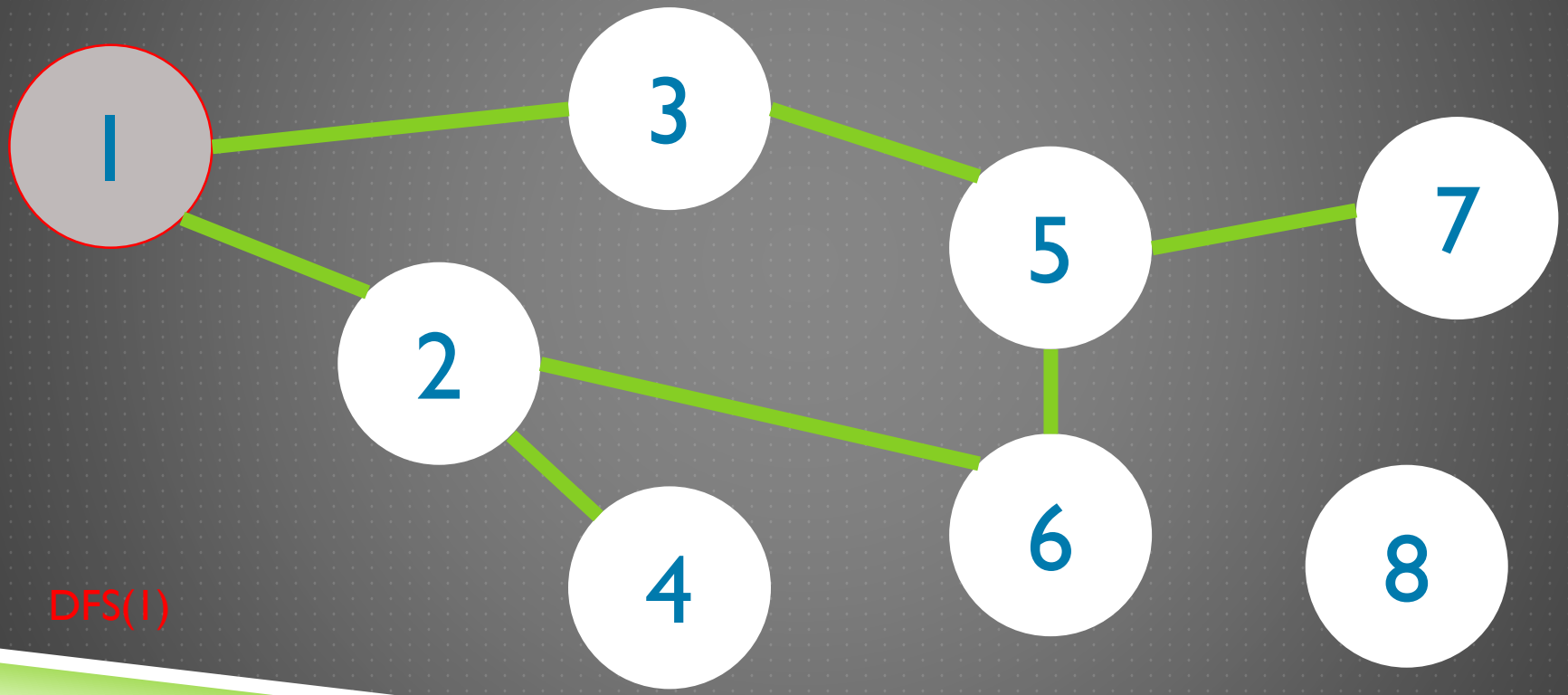
- Starts at a node
- Explores as far as possible along each branch before backtracking
- Recursion is often used

DEPTH-FIRST SEARCH - EXAMPLE



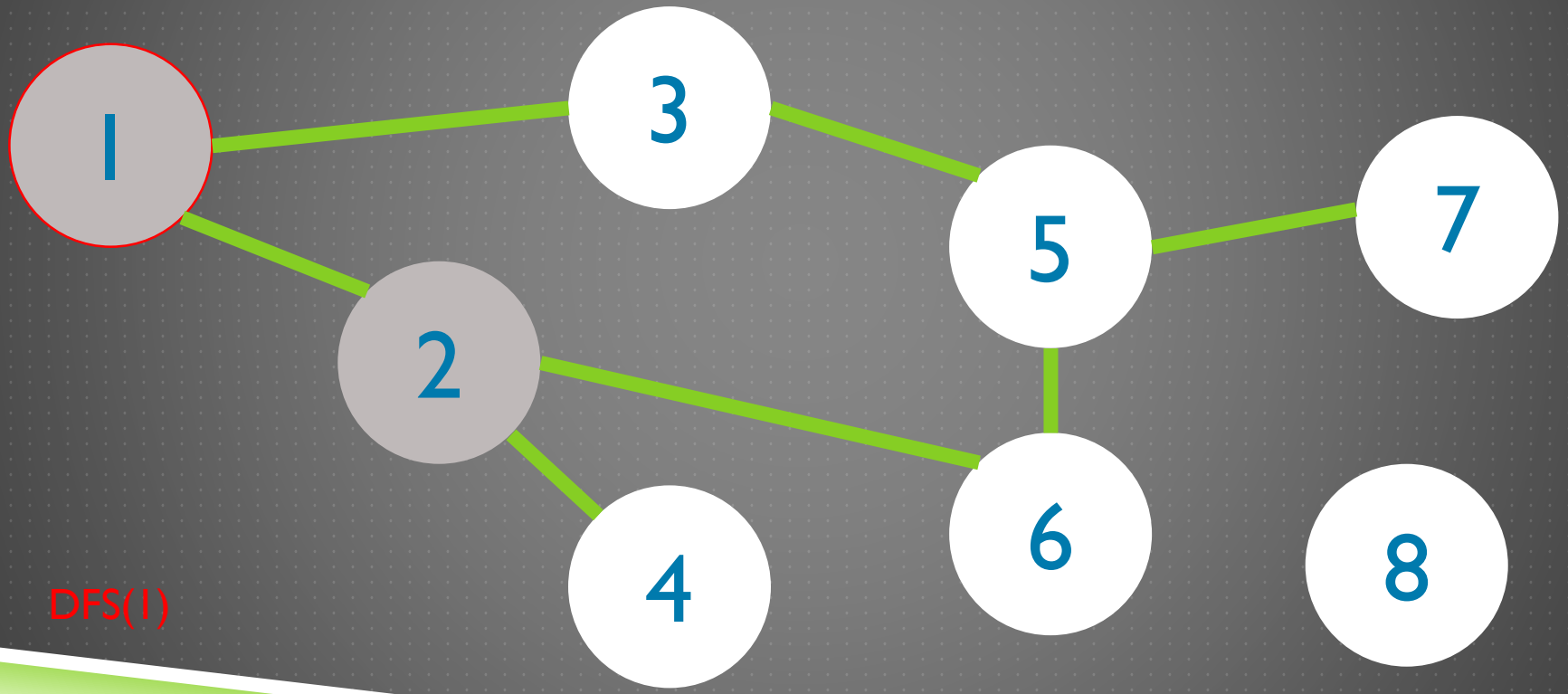
DEPTH-FIRST SEARCH - EXAMPLE

- Perform DFS on node 1



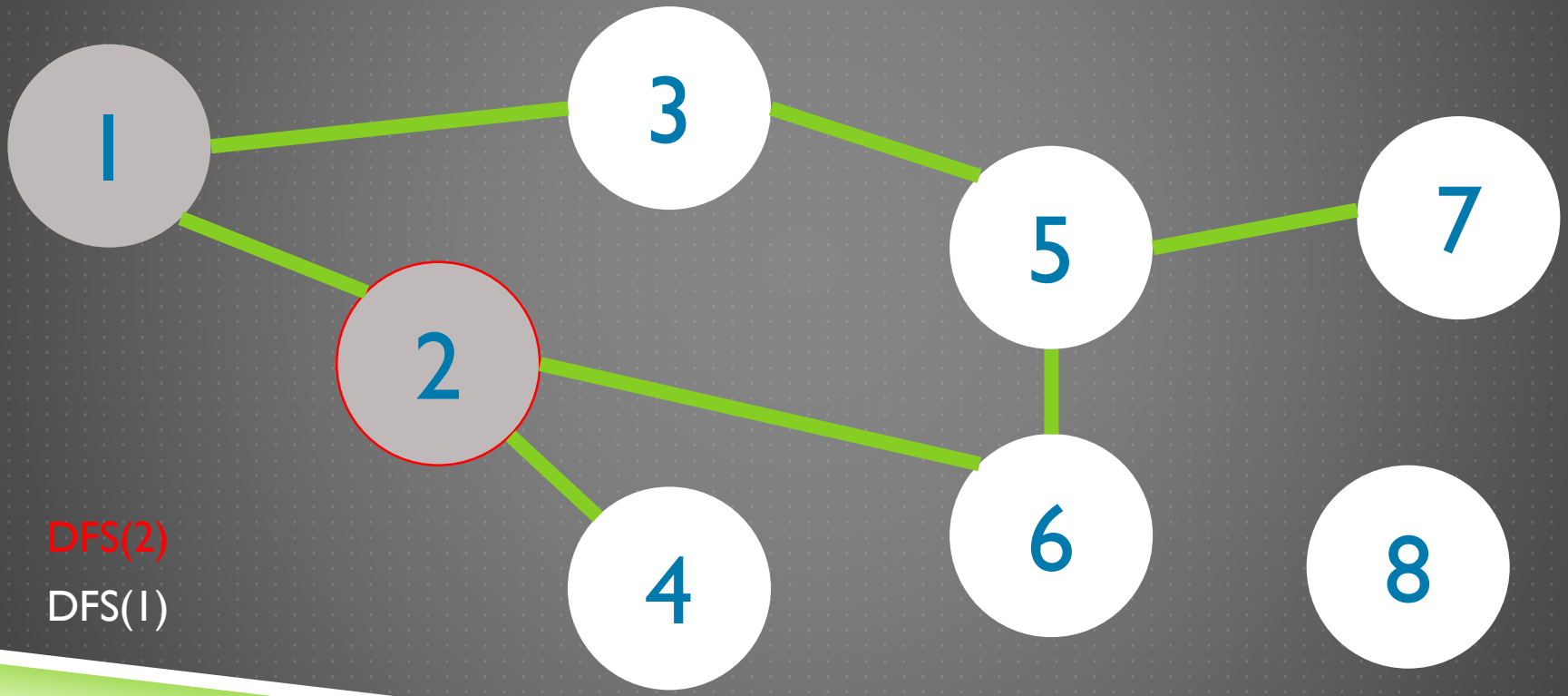
DEPTH-FIRST SEARCH - EXAMPLE

- Node 2 is not visited, perform DFS on node 2



DEPTH-FIRST SEARCH - EXAMPLE

- Node 4 is not visited, perform DFS on node 4

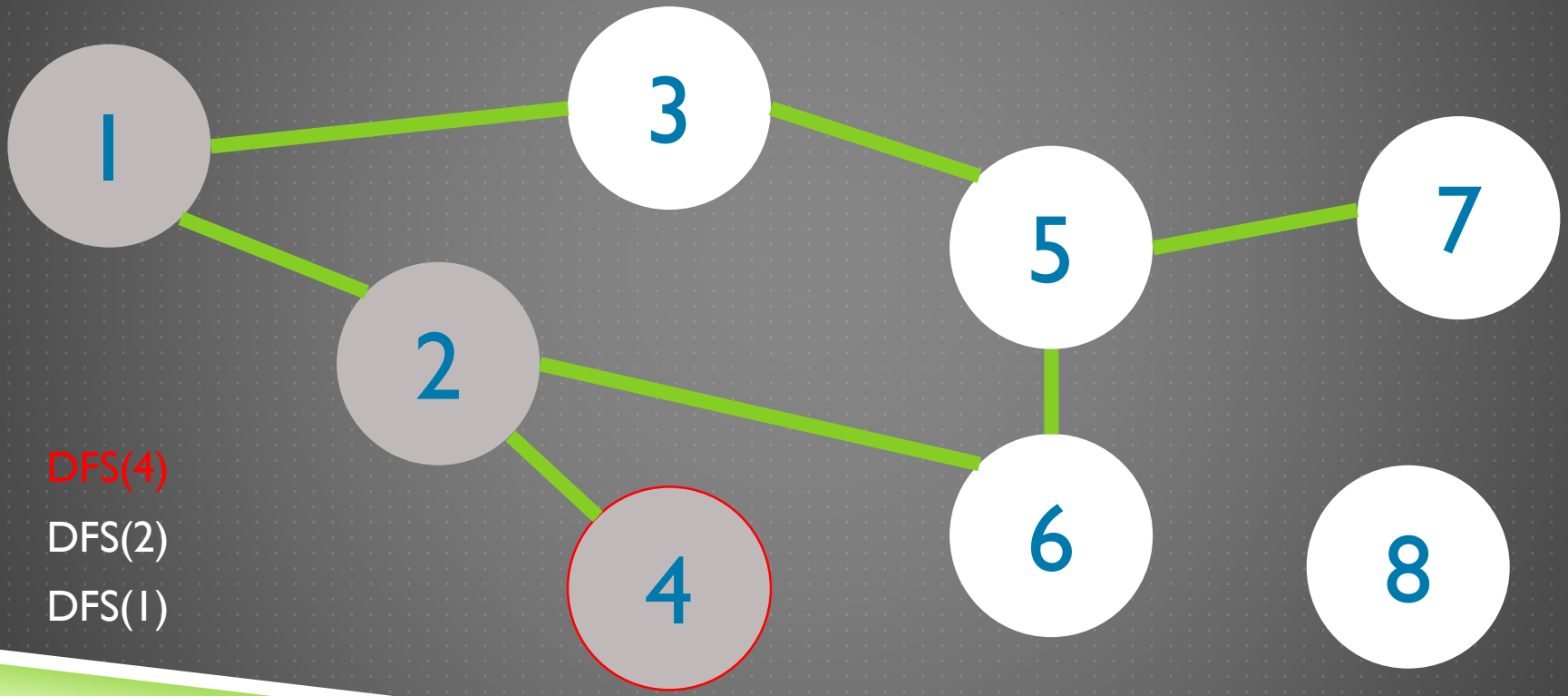


DFS(2)

DFS(1)

DEPTH-FIRST SEARCH - EXAMPLE

- Node 2 is visited, no need to perform DFS on node 2



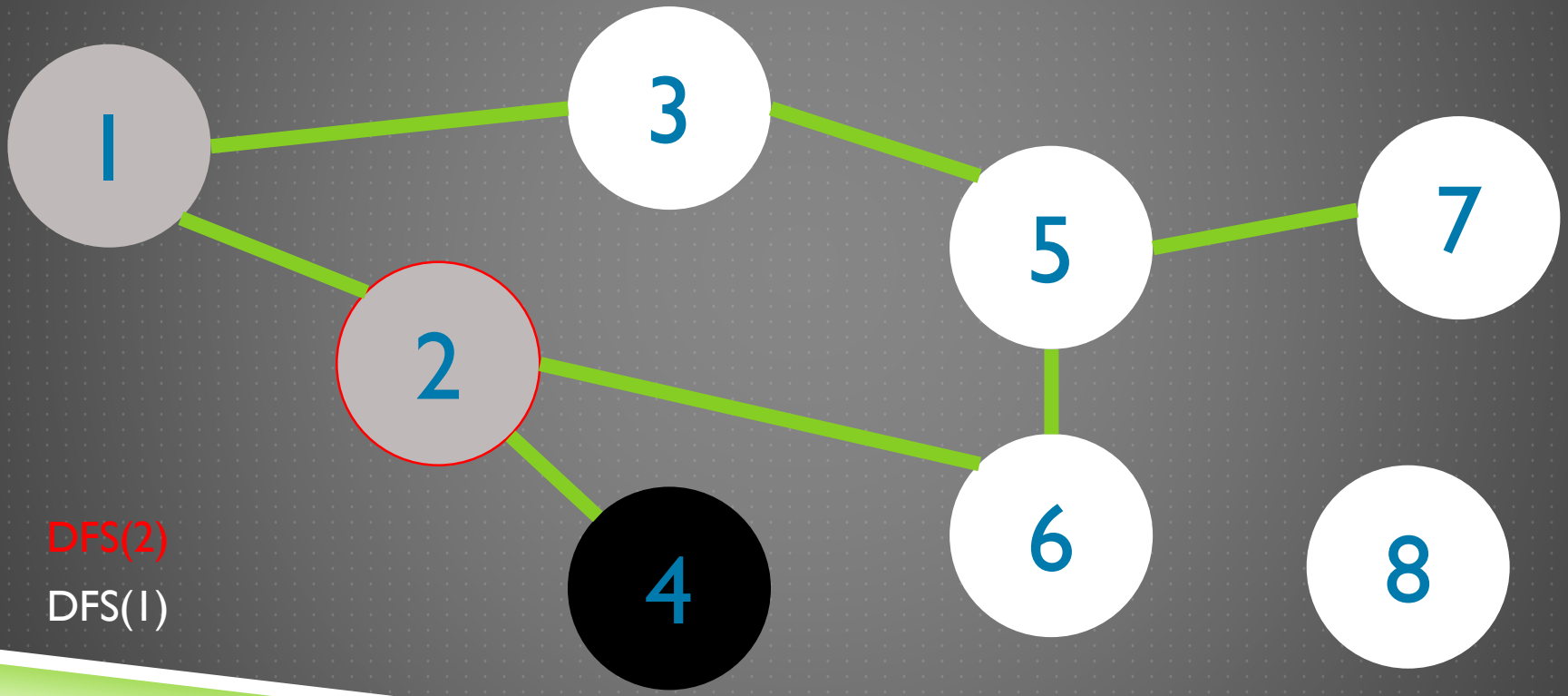
DFS(4)

DFS(2)

DFS(1)

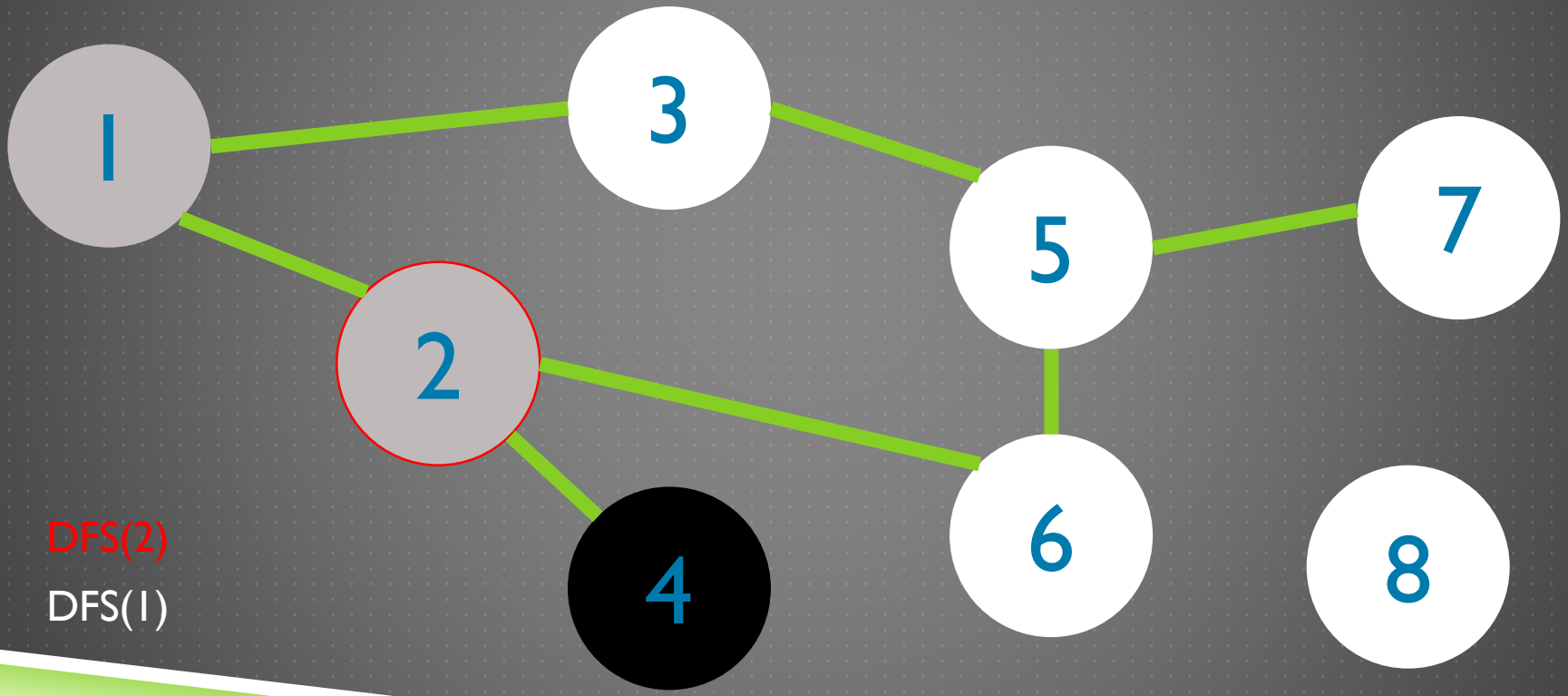
DEPTH-FIRST SEARCH - EXAMPLE

- DFS(4) is finished, go back to DFS(2)



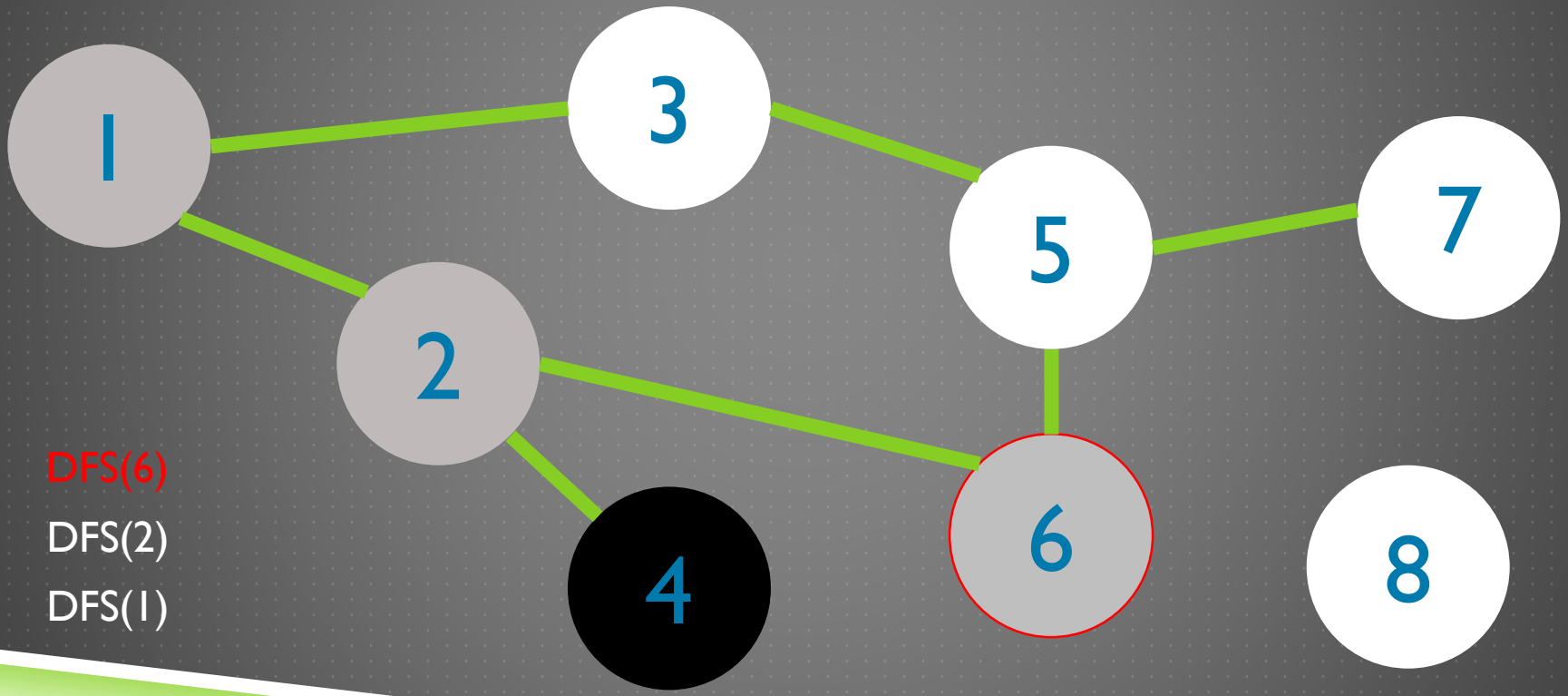
DEPTH-FIRST SEARCH - EXAMPLE

- Node 6 is not visited, perform DFS on node 6



DEPTH-FIRST SEARCH - EXAMPLE

- Node 5 is not visited, perform DFS on node 5



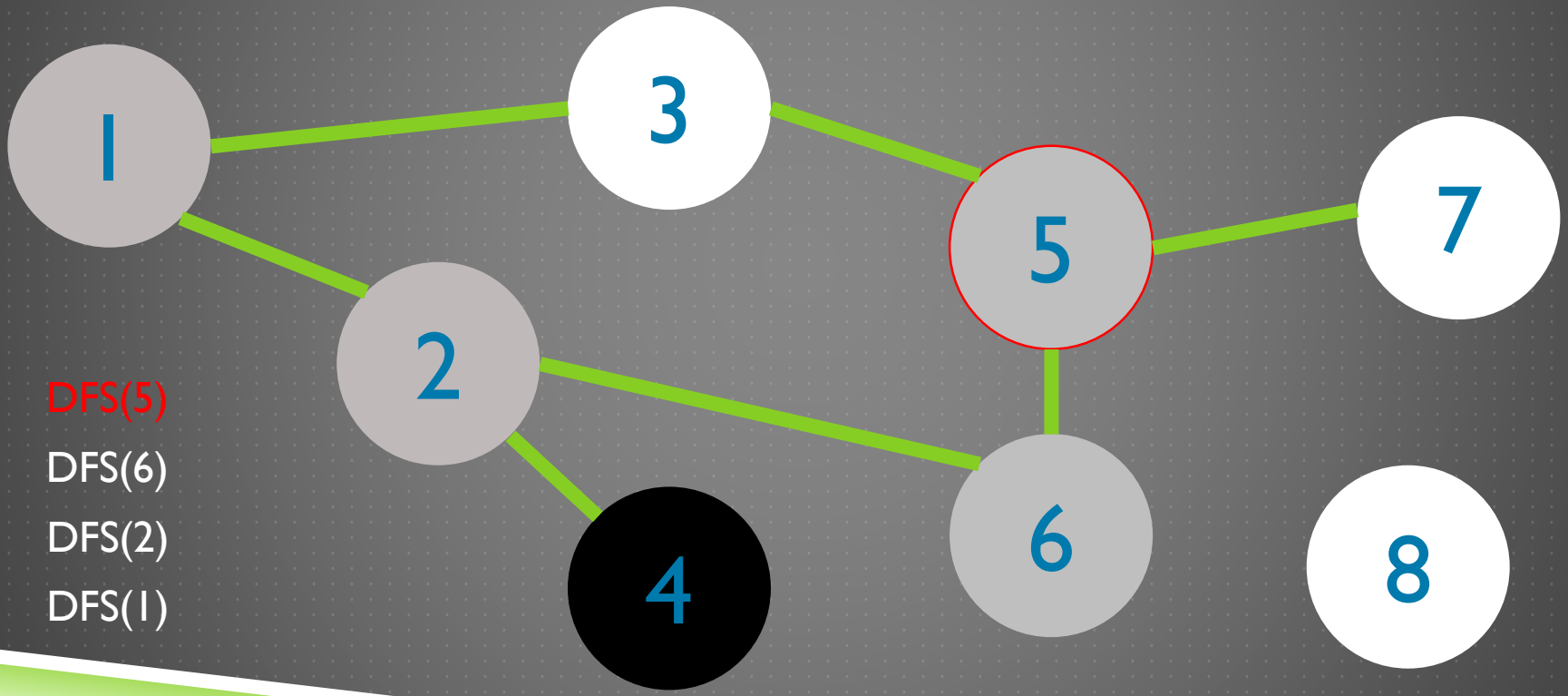
DFS(6)

DFS(2)

DFS(1)

DEPTH-FIRST SEARCH - EXAMPLE

- Node 3 is not visited, perform DFS on node 3



DFS(5)

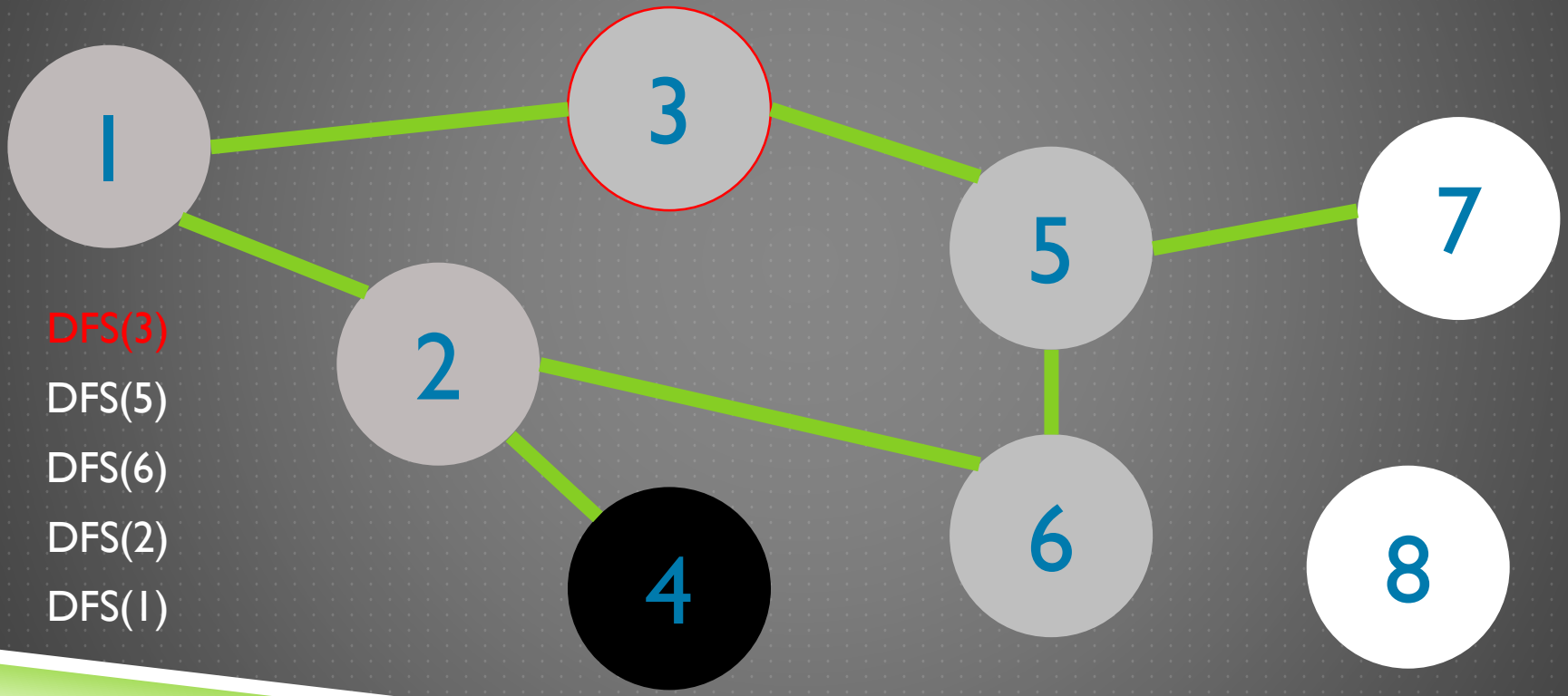
DFS(6)

DFS(2)

DFS(1)

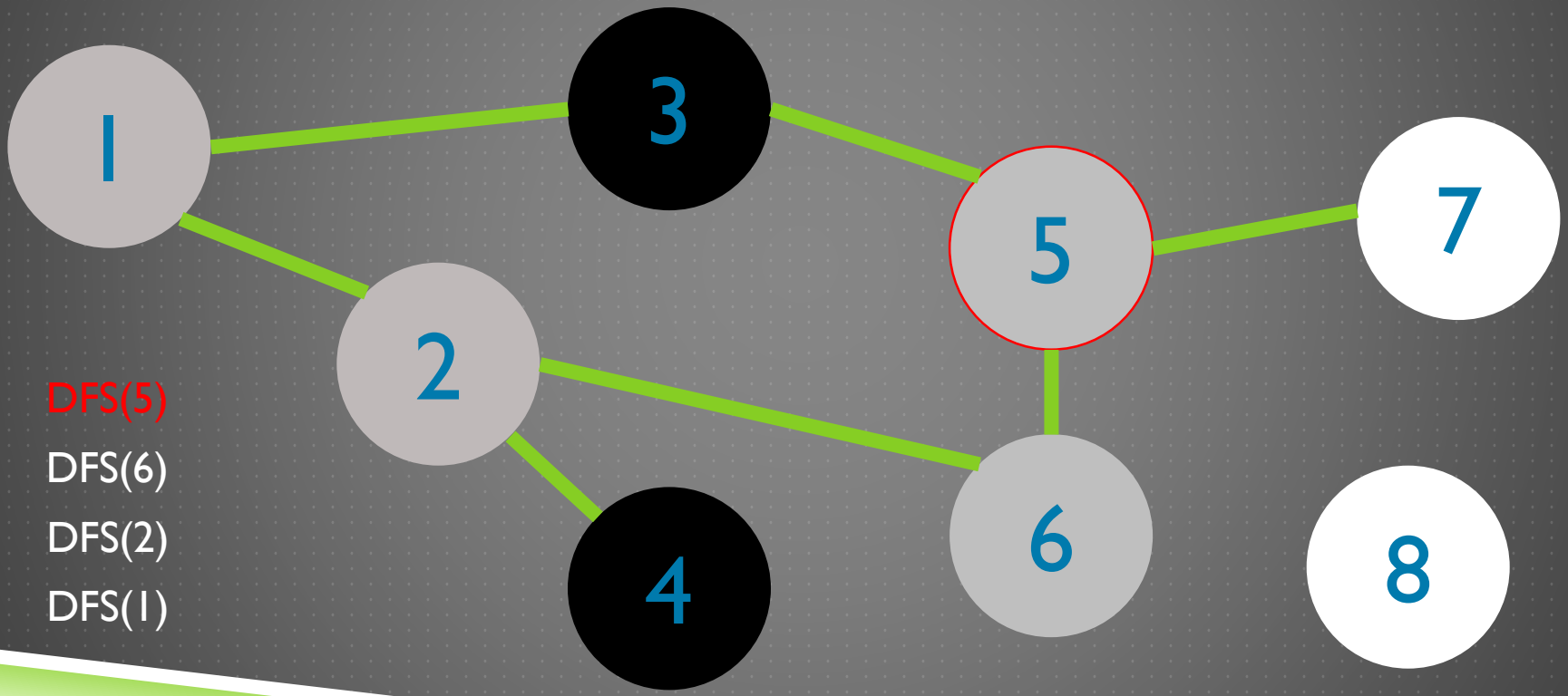
DEPTH-FIRST SEARCH - EXAMPLE

- No neighbors of node 3 is not visited, go back to DFS(5)



DEPTH-FIRST SEARCH - EXAMPLE

- … and so on



DFS(5)

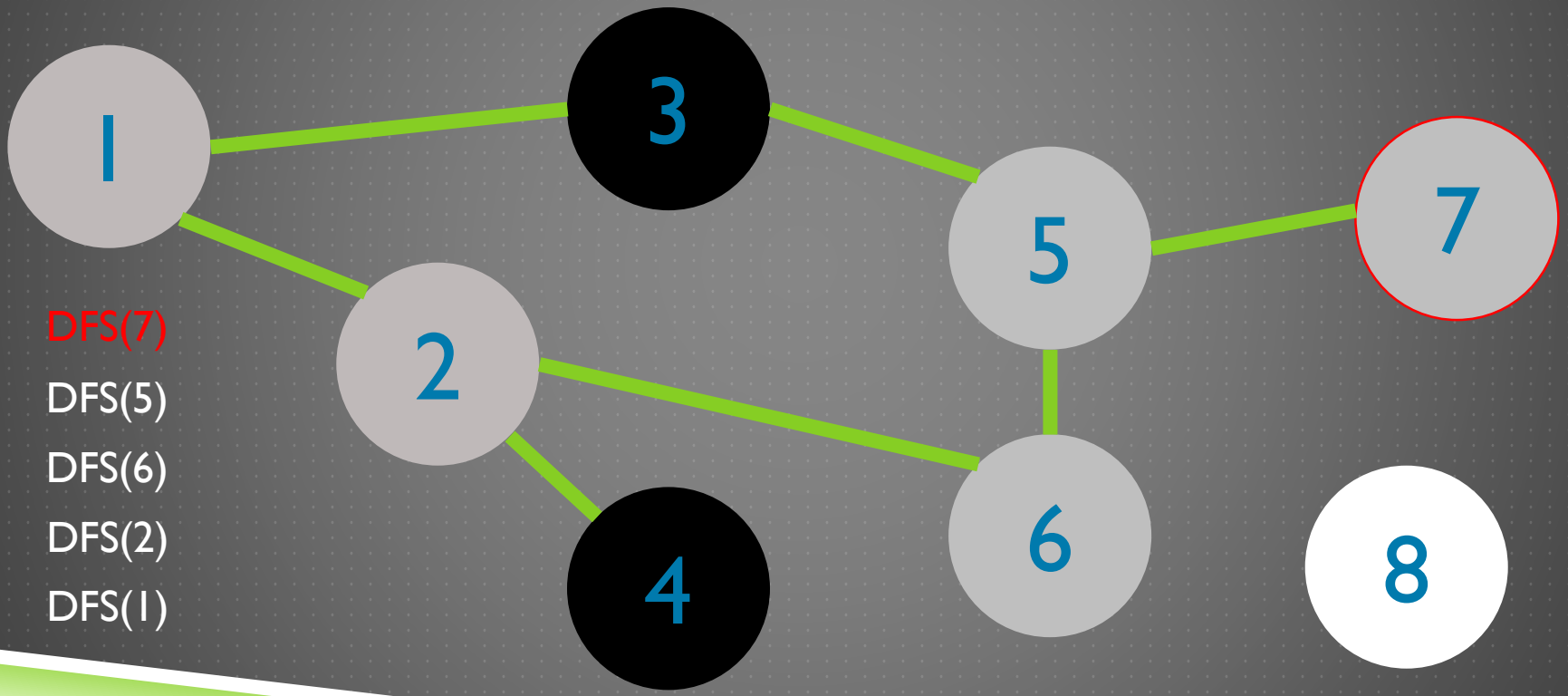
DFS(6)

DFS(2)

DFS(1)

DEPTH-FIRST SEARCH - EXAMPLE

- … and so on



DFS(7)

DFS(5)

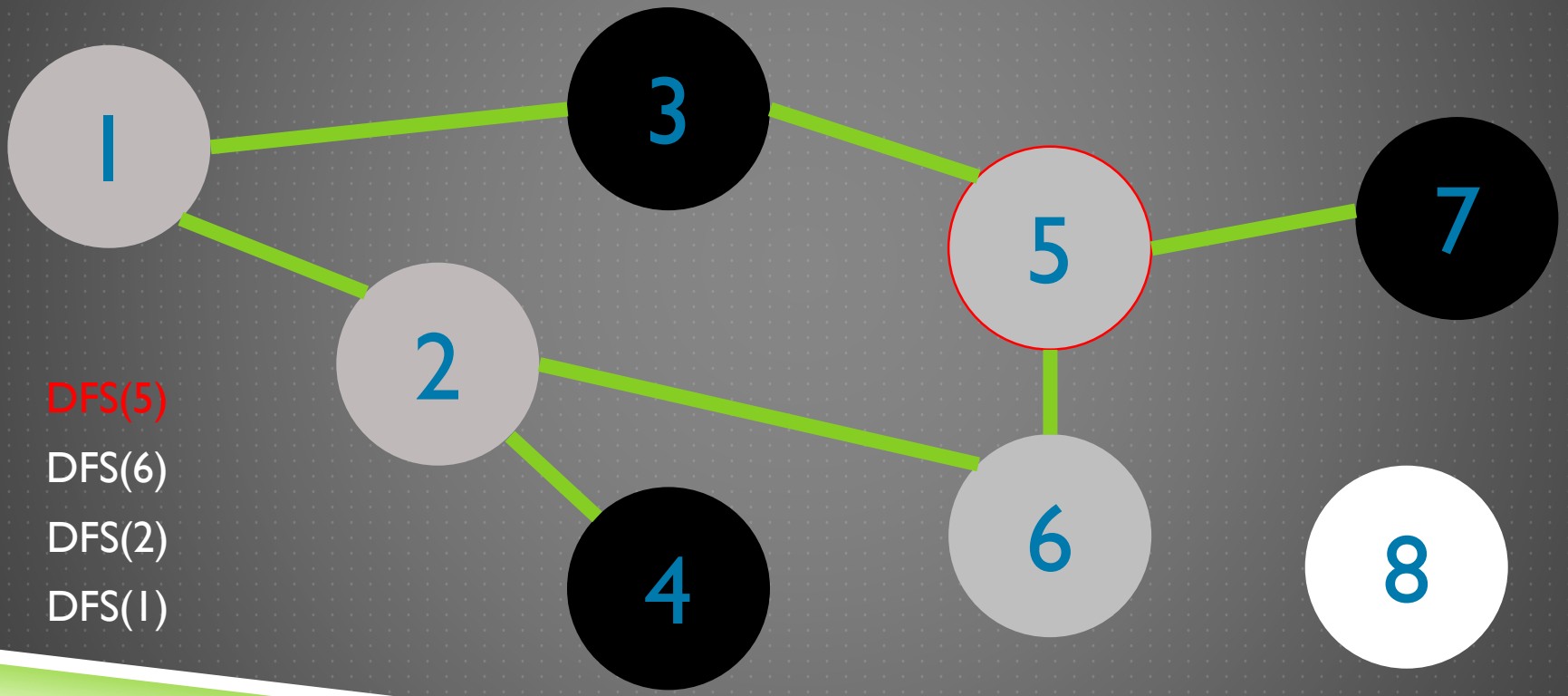
DFS(6)

DFS(2)

DFS(1)

DEPTH-FIRST SEARCH - EXAMPLE

- … and so on



DFS(5)

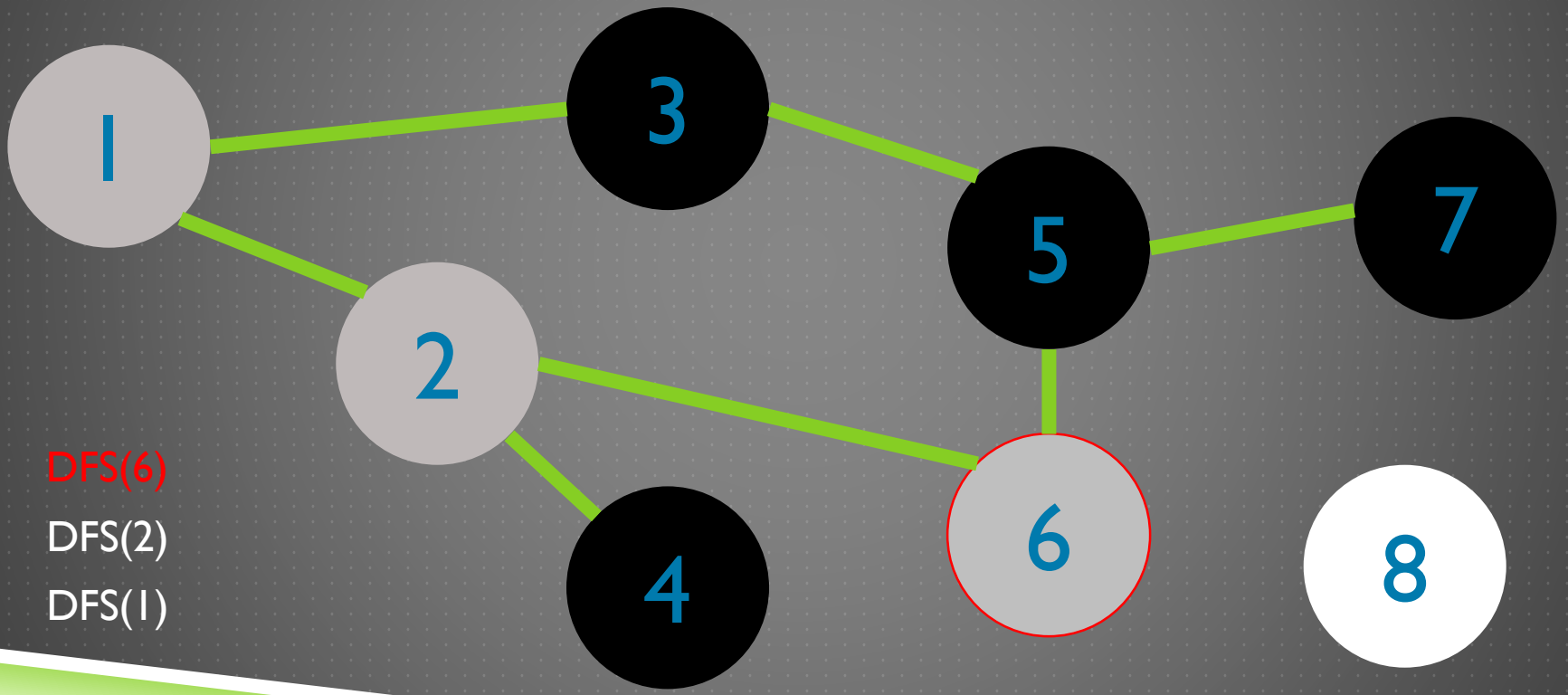
DFS(6)

DFS(2)

DFS(1)

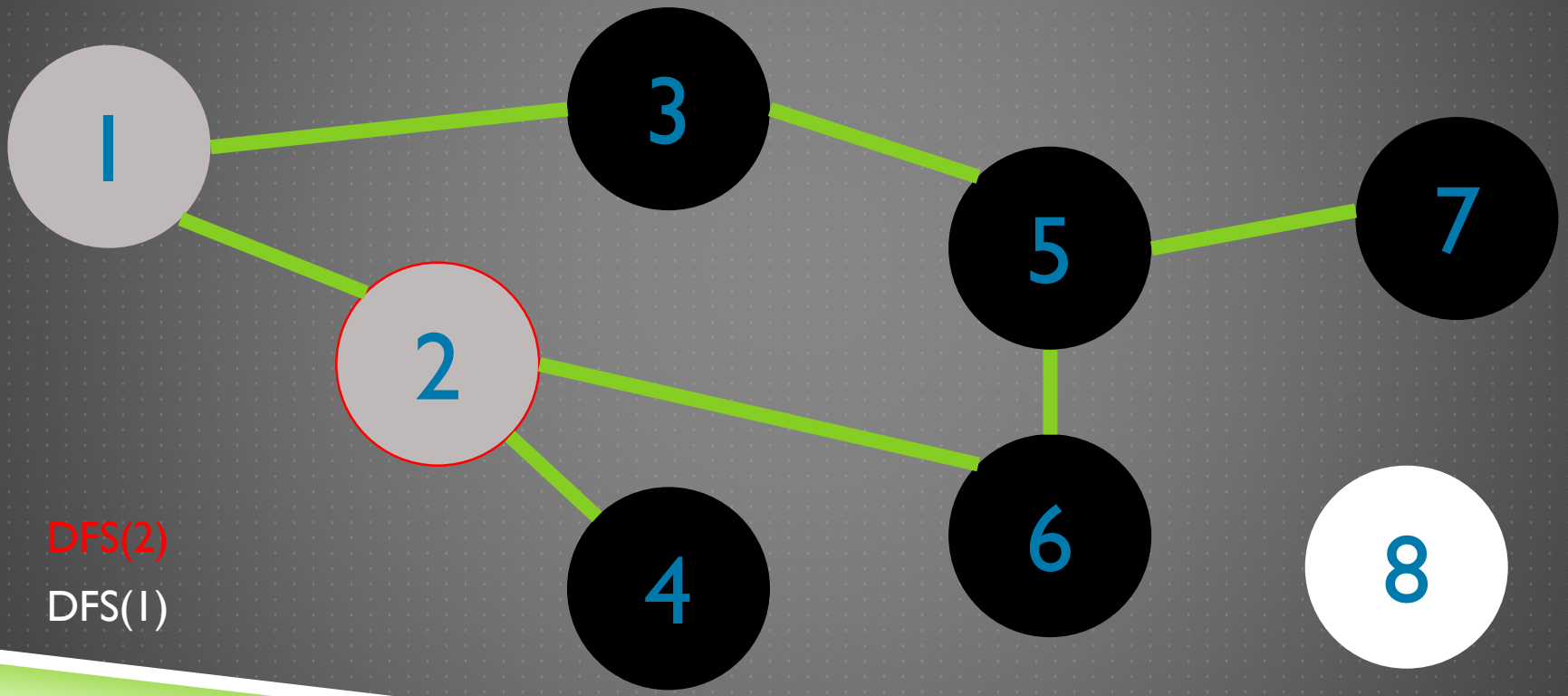
DEPTH-FIRST SEARCH - EXAMPLE

- … and so on



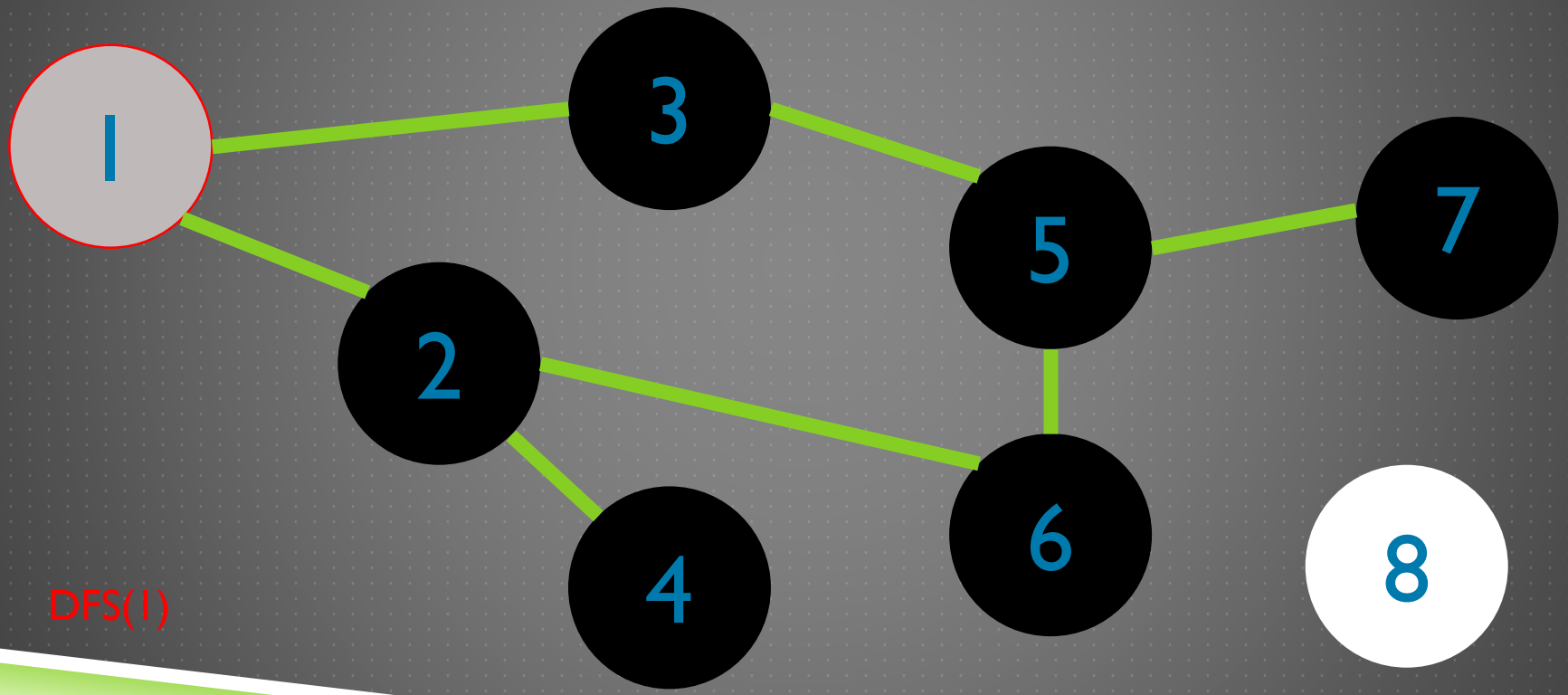
DEPTH-FIRST SEARCH - EXAMPLE

- … and so on



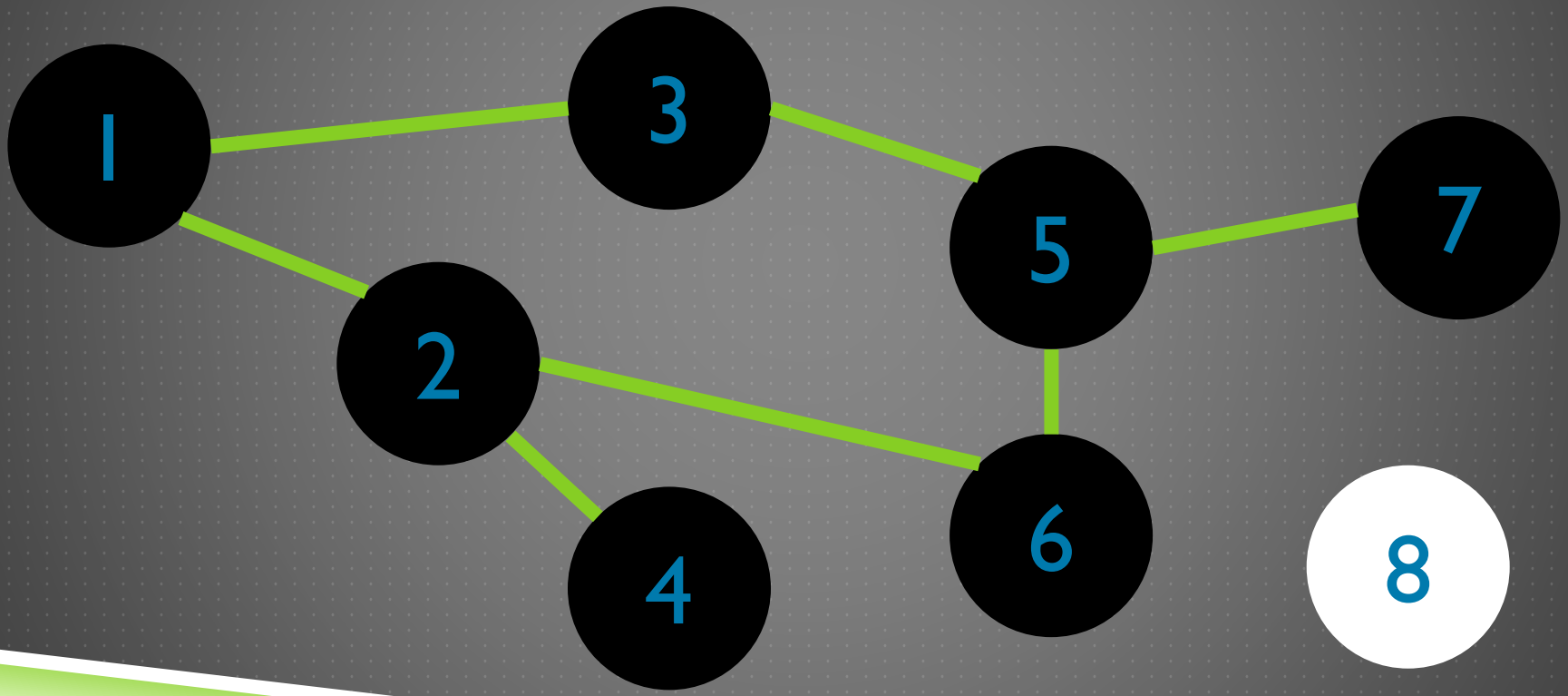
DEPTH-FIRST SEARCH - EXAMPLE

- … and so on



DEPTH-FIRST SEARCH - EXAMPLE

- Done



DEPTH-FIRST SEARCH - CODE

```
procedure DFS(vertex v){
    label v as discovered
    for all w adjacent to v do
        if w is not labeled as discovered then
            DFS(w)
    }
```

DEPTH-FIRST SEARCH

- Time Complexity:
  - $O(|V|^2)$ when using adjacency matrix
  - $O(|V|+|E|)$ when using adjacency list / edge list

DEPTH-FIRST SEARCH - APPLICATION

- Solving mazes
- Finding connected components
- Bi-coloring a graph

PROBLEM

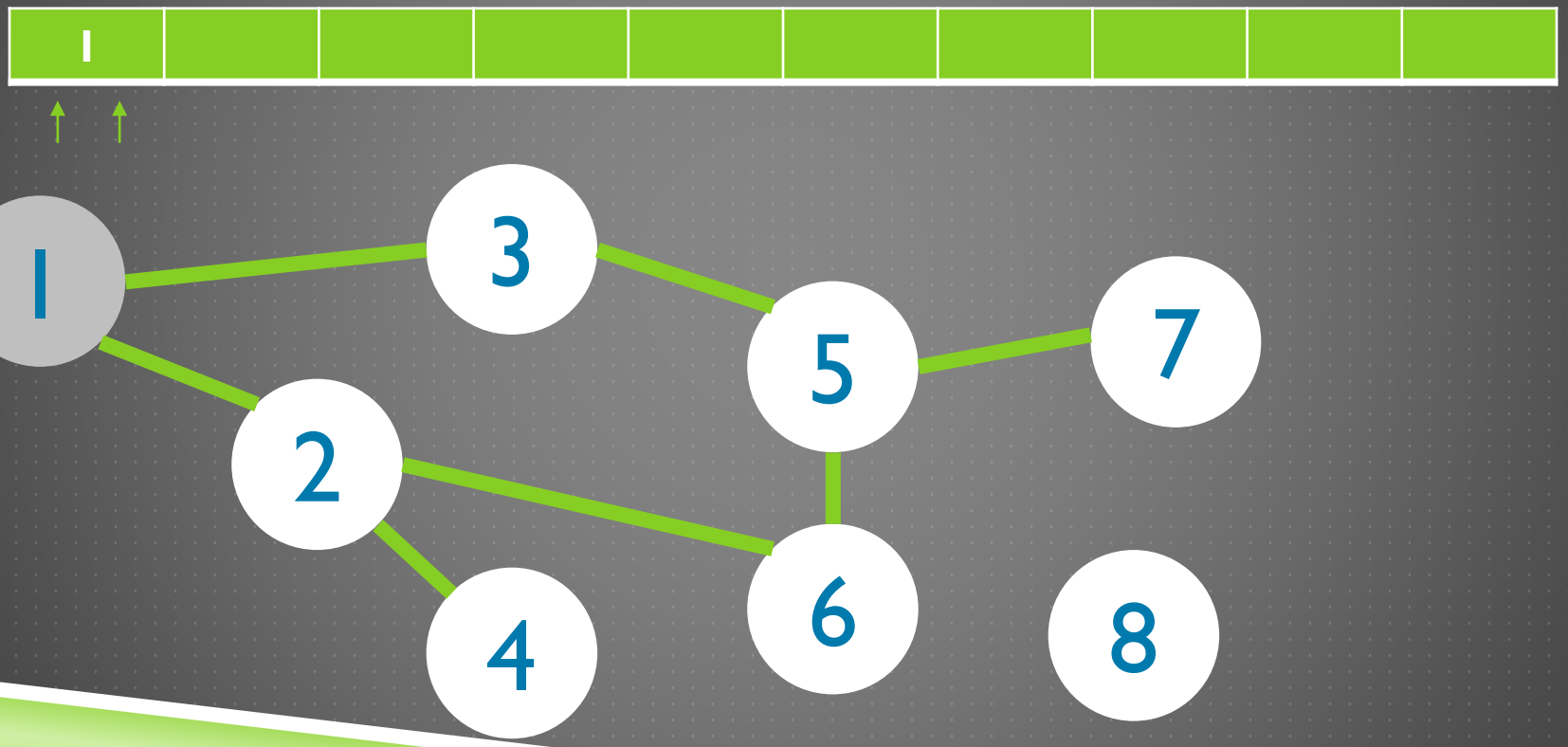
- HKOJ 01035 Patrol Area

BREADTH-FIRST SEARCH

- Starts at a node
- Explores the neighbor nodes first before moving to the next level neighbors
- Use a queue to implement

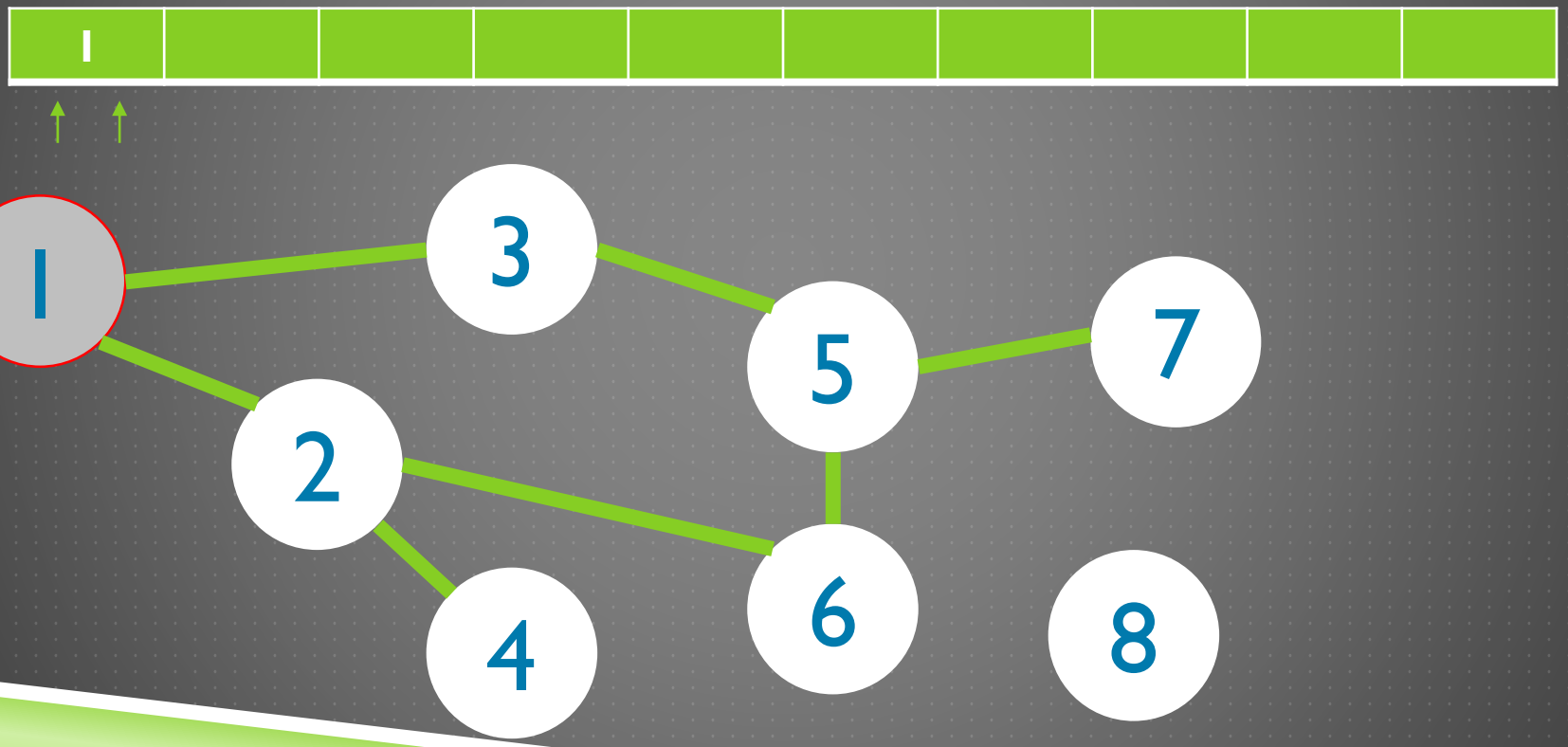
BREADTH-FIRST SEARCH - EXAMPLE

- Push 1 into the queue and mark node 1 as visited first



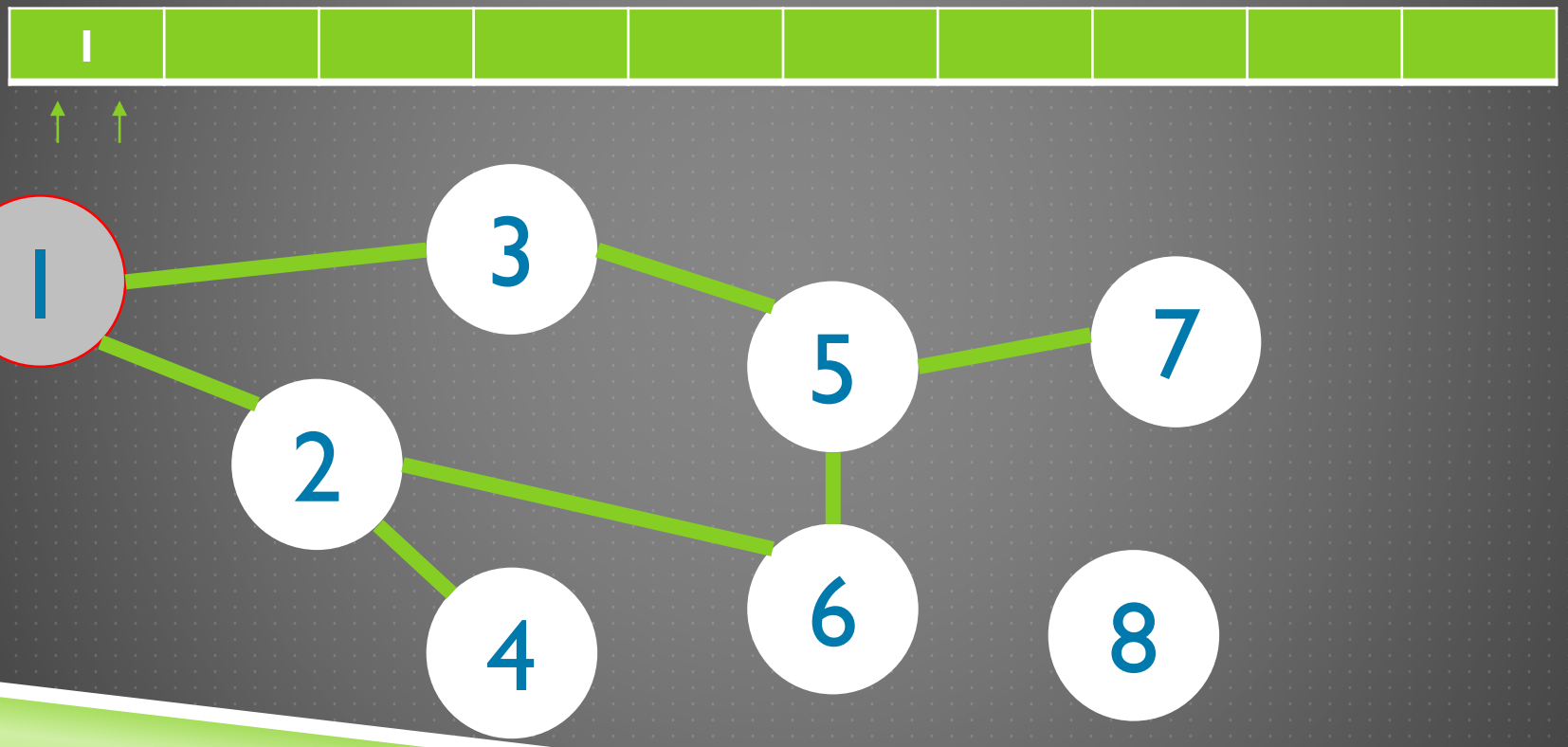
BREADTH-FIRST SEARCH - EXAMPLE

- Perform BFS on node 1



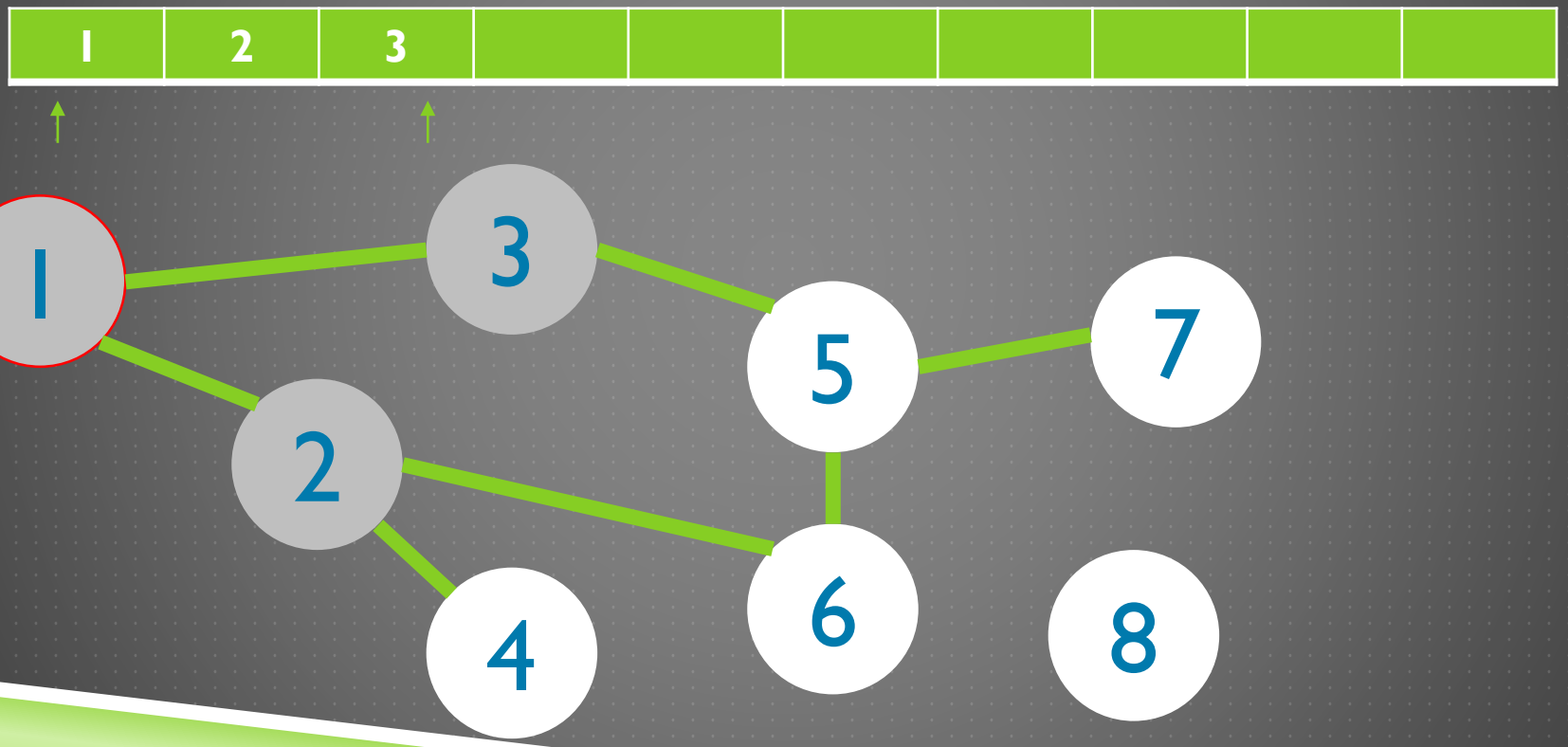
BREADTH-FIRST SEARCH - EXAMPLE

- Node 2 and 3 are adjacent to node 1 and they are not visited



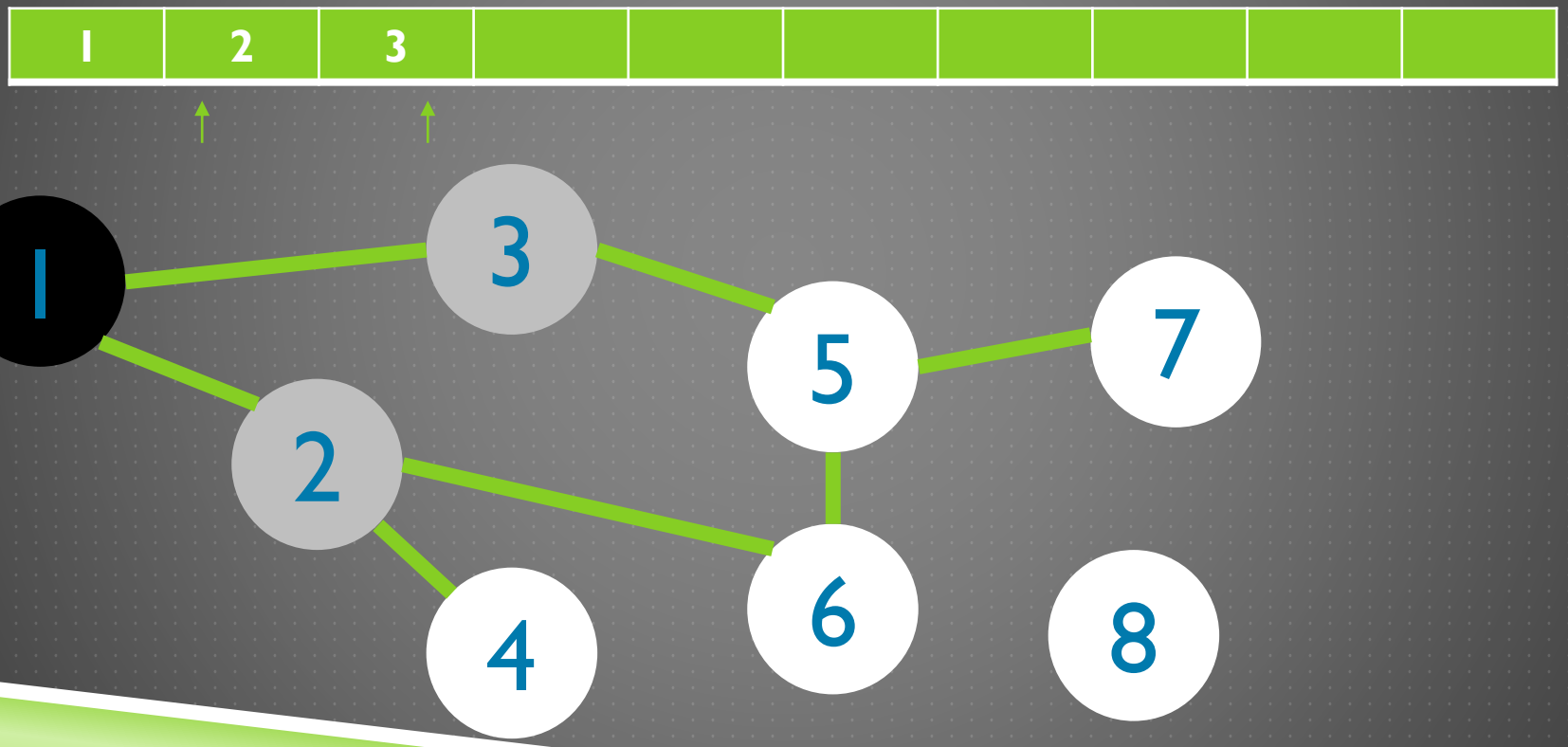
BREADTH-FIRST SEARCH - EXAMPLE

- Mark node 2 and 3 as visited and push them into the queue



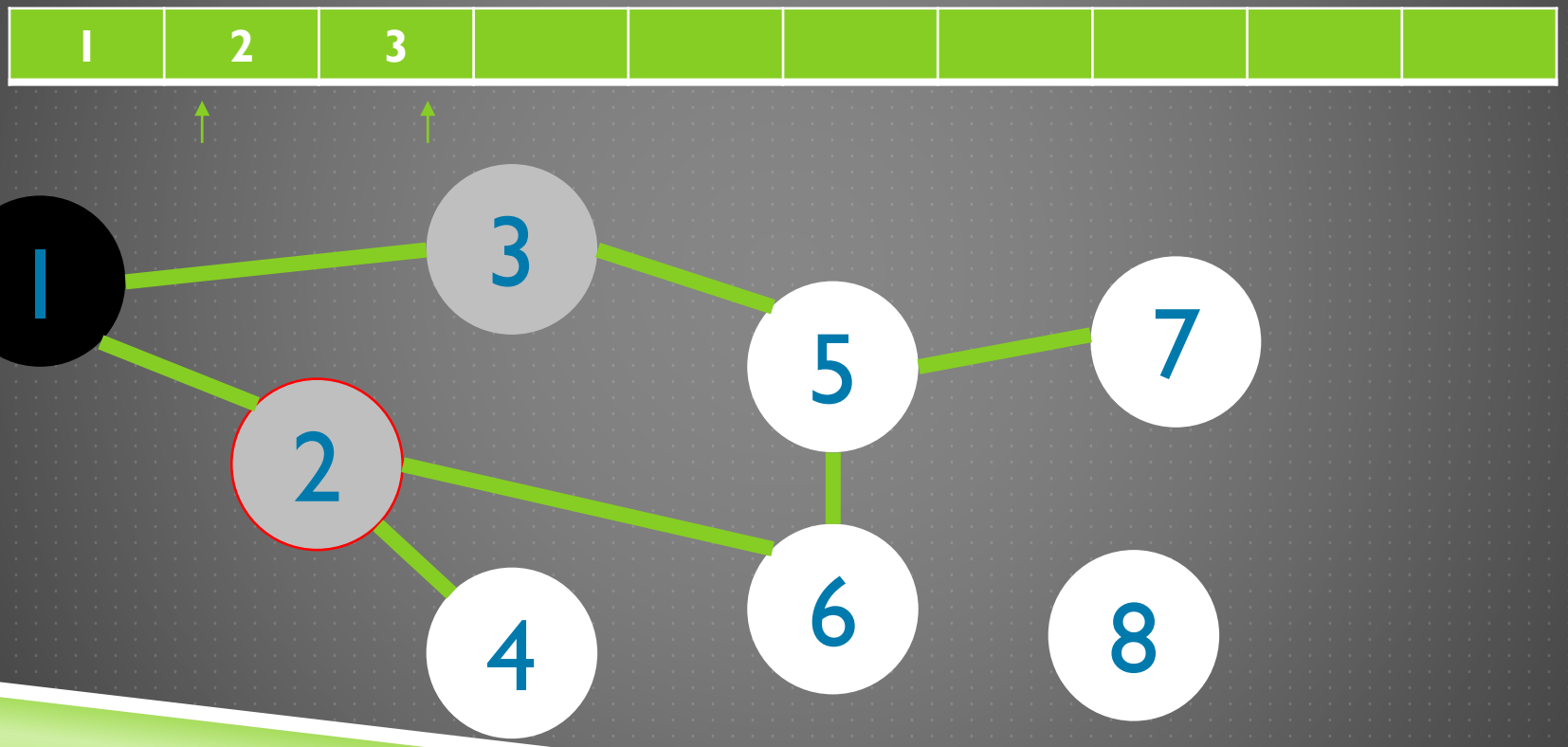
BREADTH-FIRST SEARCH - EXAMPLE

- BFS on node 1 is finished, pop 1 from the queue



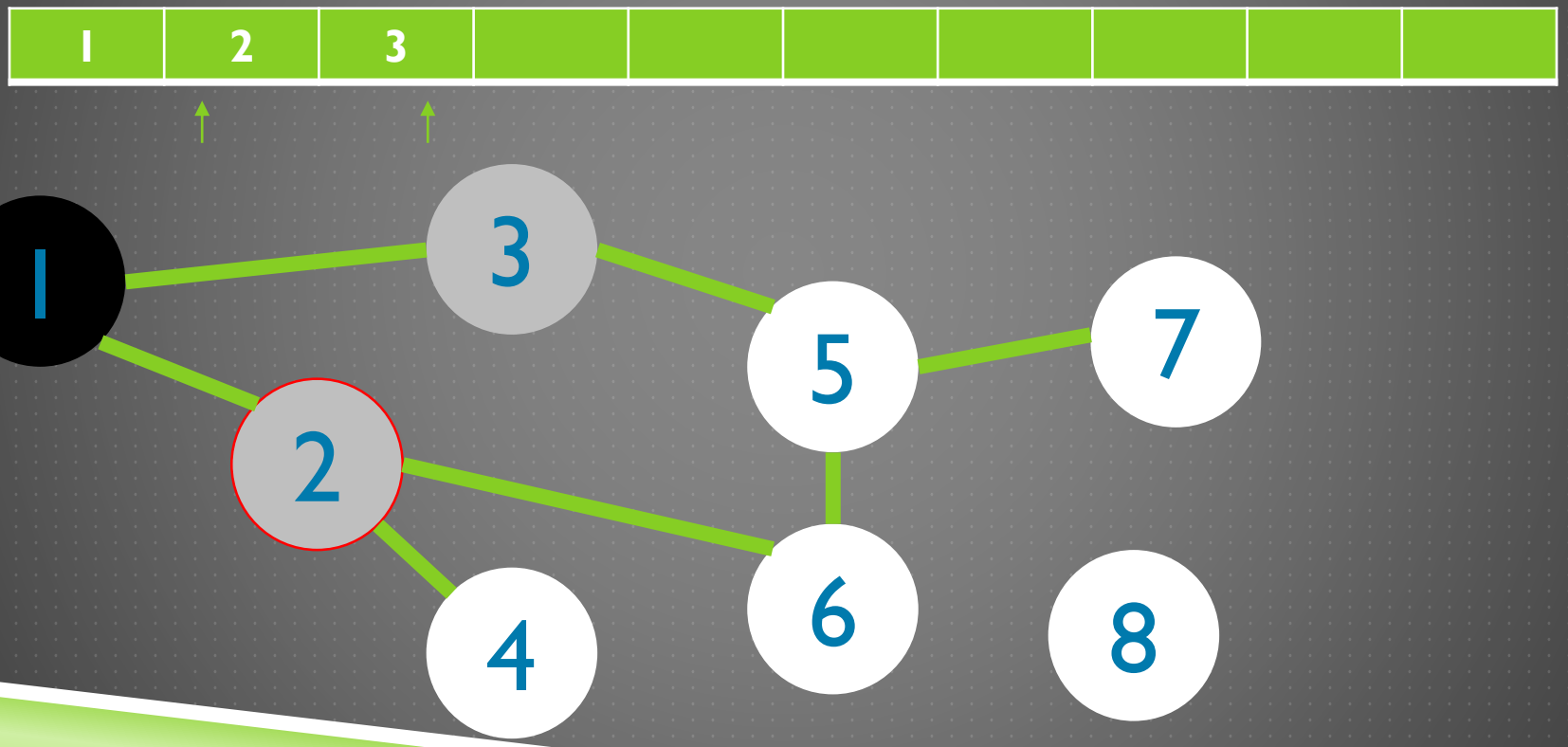
BREADTH-FIRST SEARCH - EXAMPLE

- Perform BFS on node 2



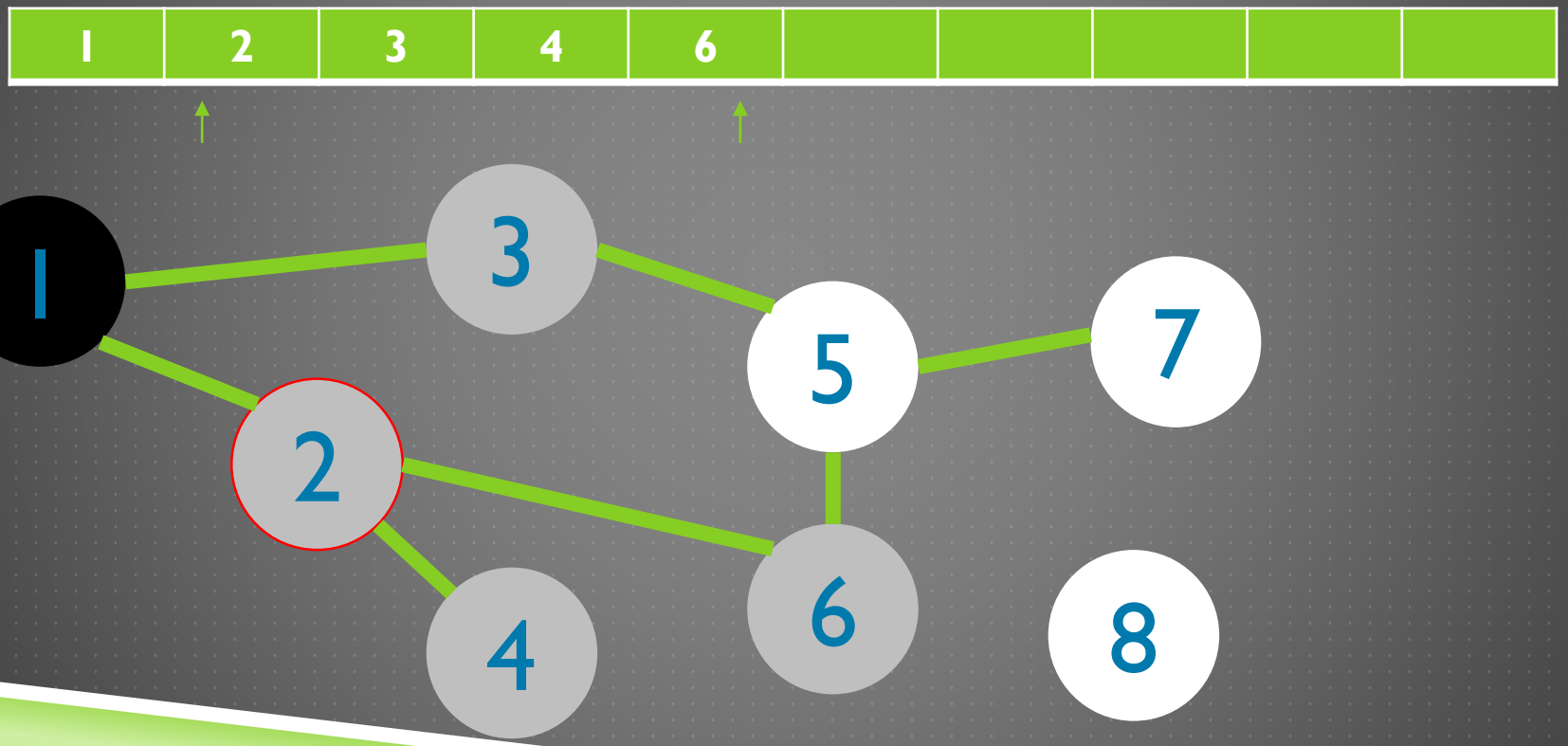
BREADTH-FIRST SEARCH - EXAMPLE

- Node 4 and 6 are adjacent to node 2 and they are not visited



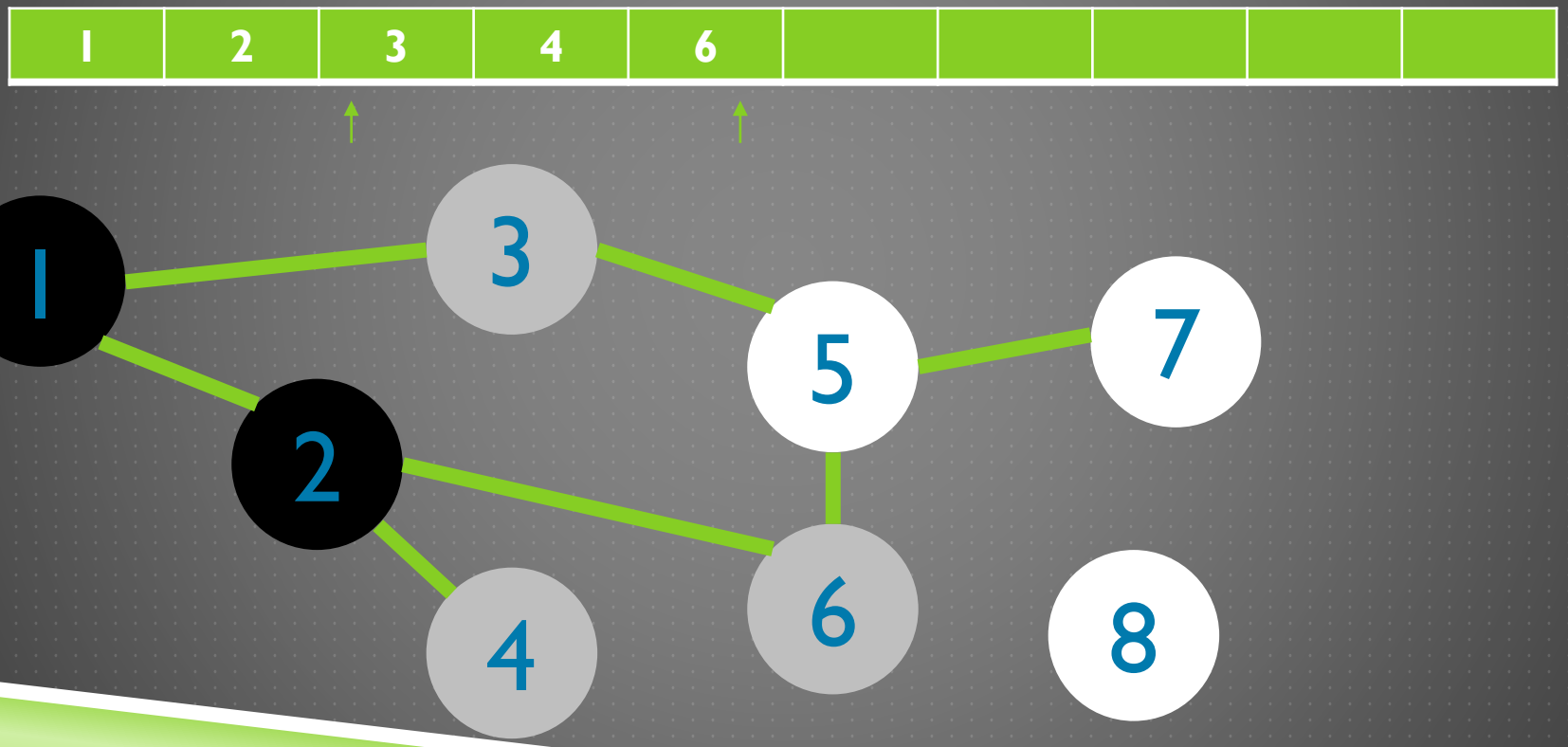
BREADTH-FIRST SEARCH - EXAMPLE

- Mark node 4 and 6 as visited and push them into the queue



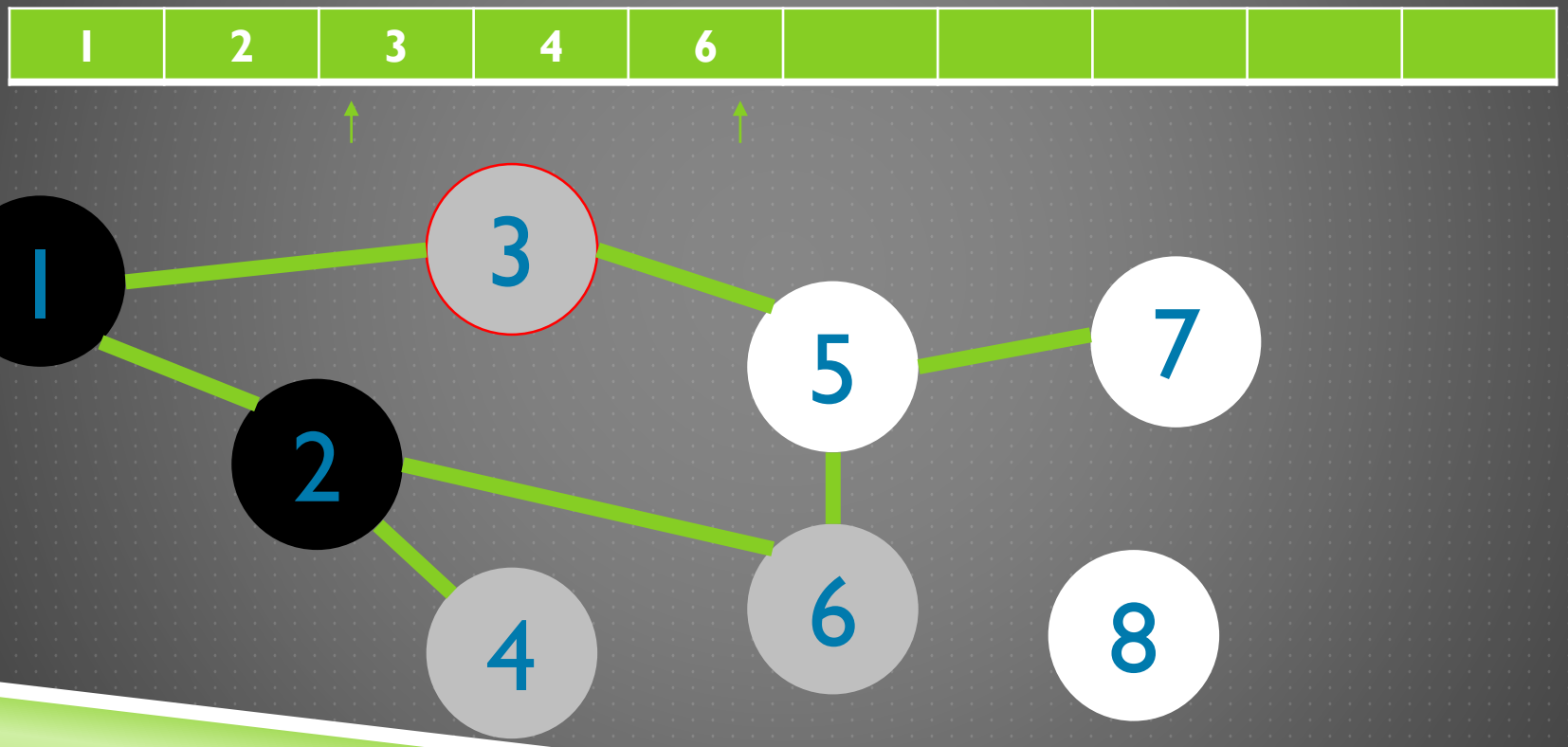
BREADTH-FIRST SEARCH - EXAMPLE

- BFS on node 2 is finished, pop 2 from the queue



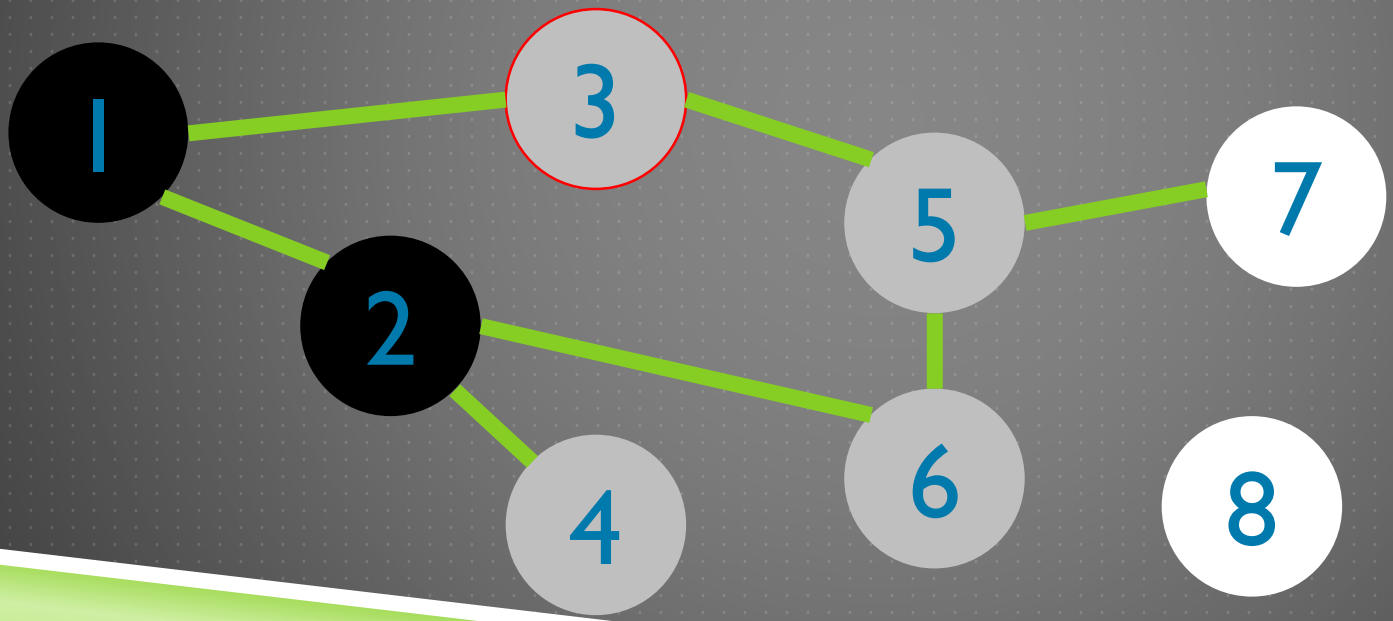
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



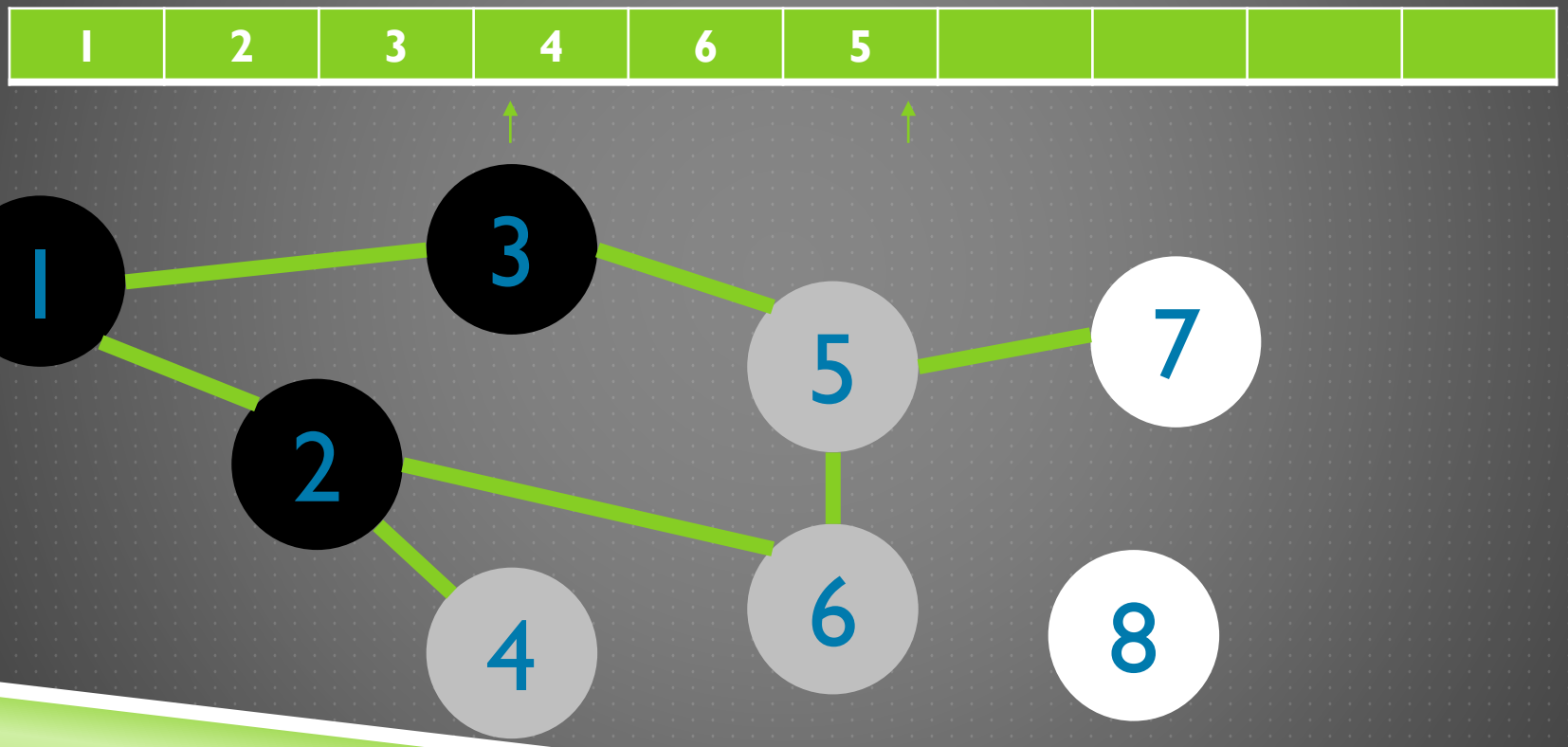
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



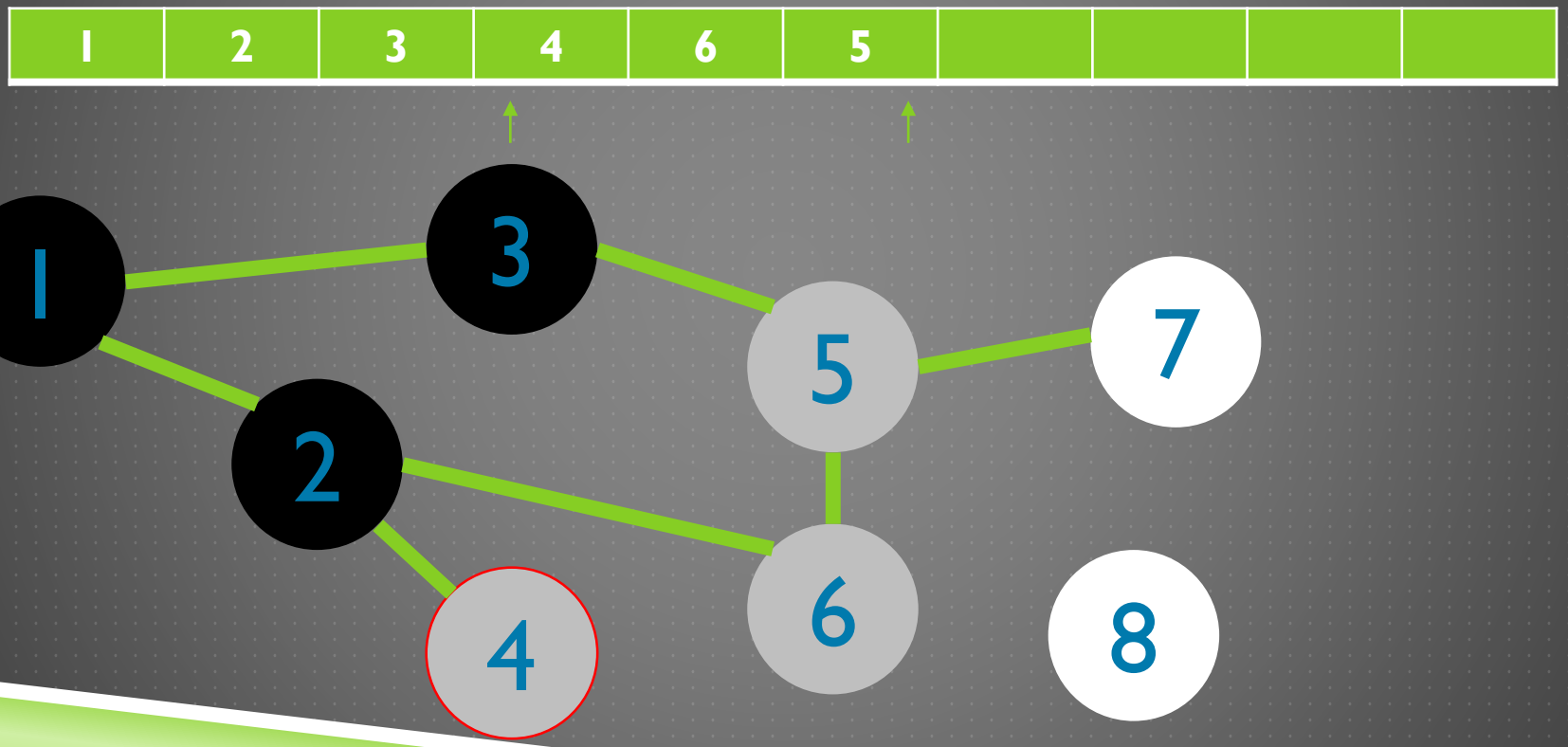
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



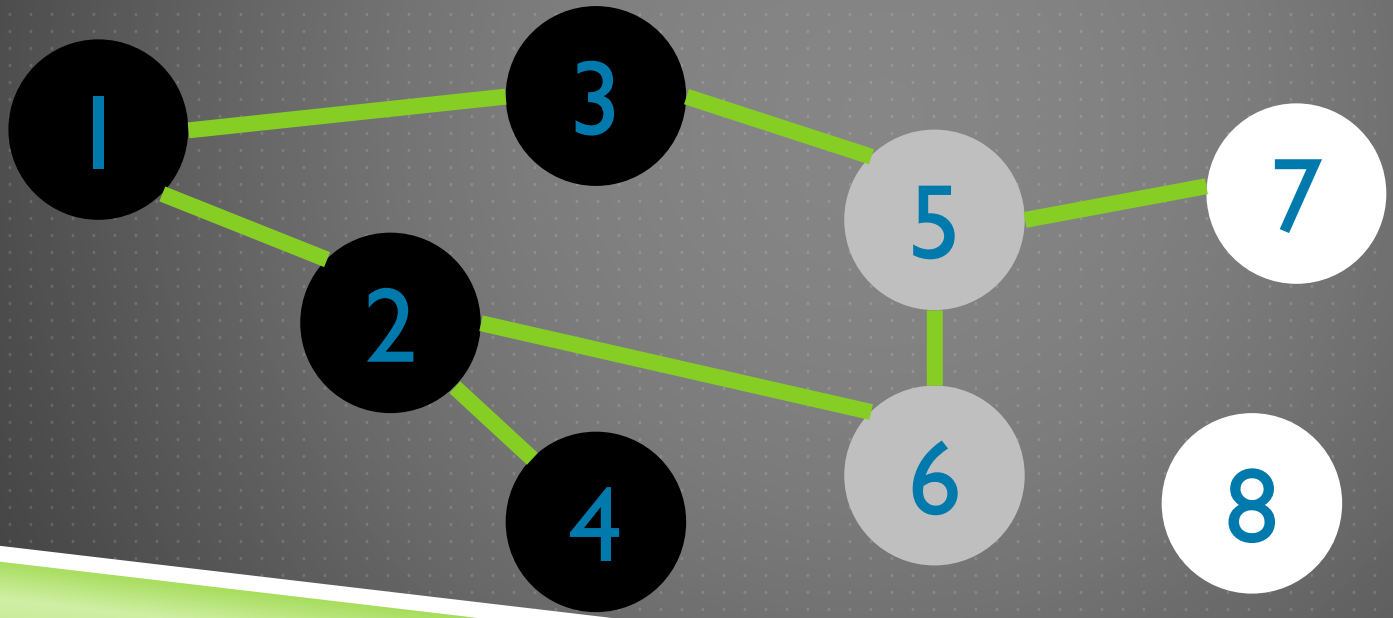
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



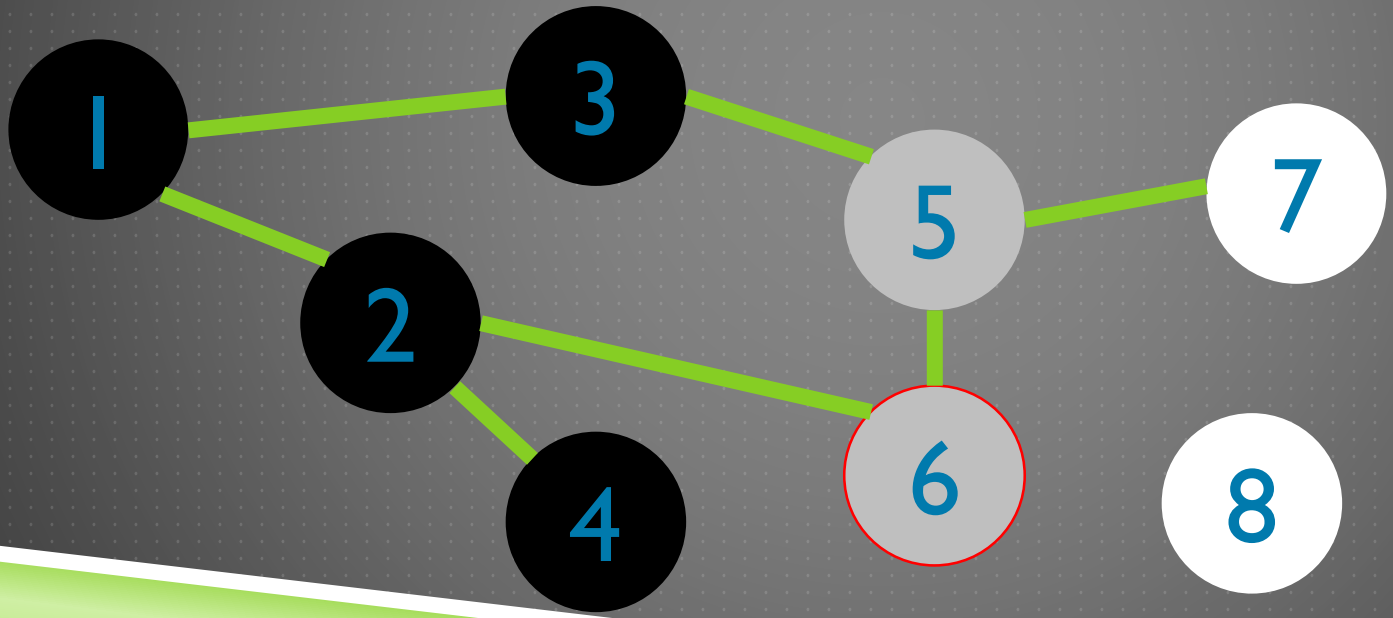
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



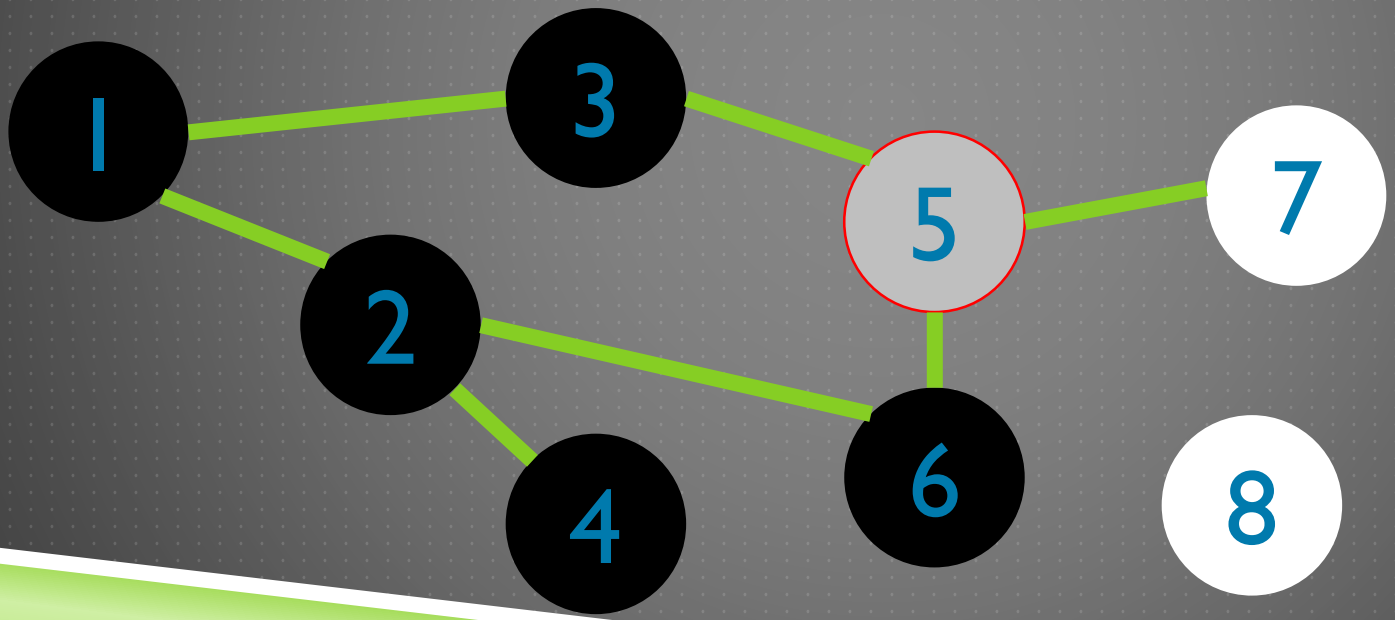
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



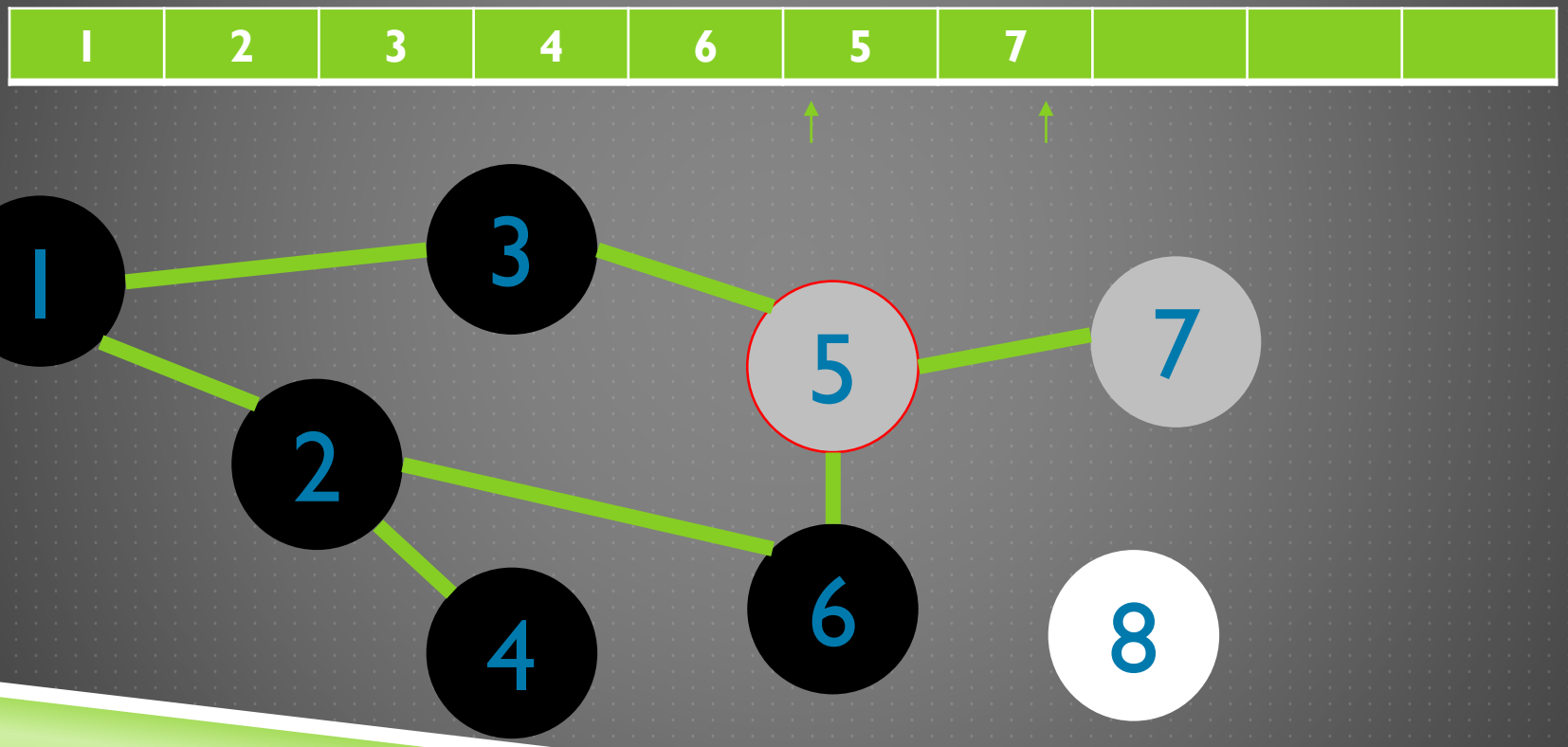
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



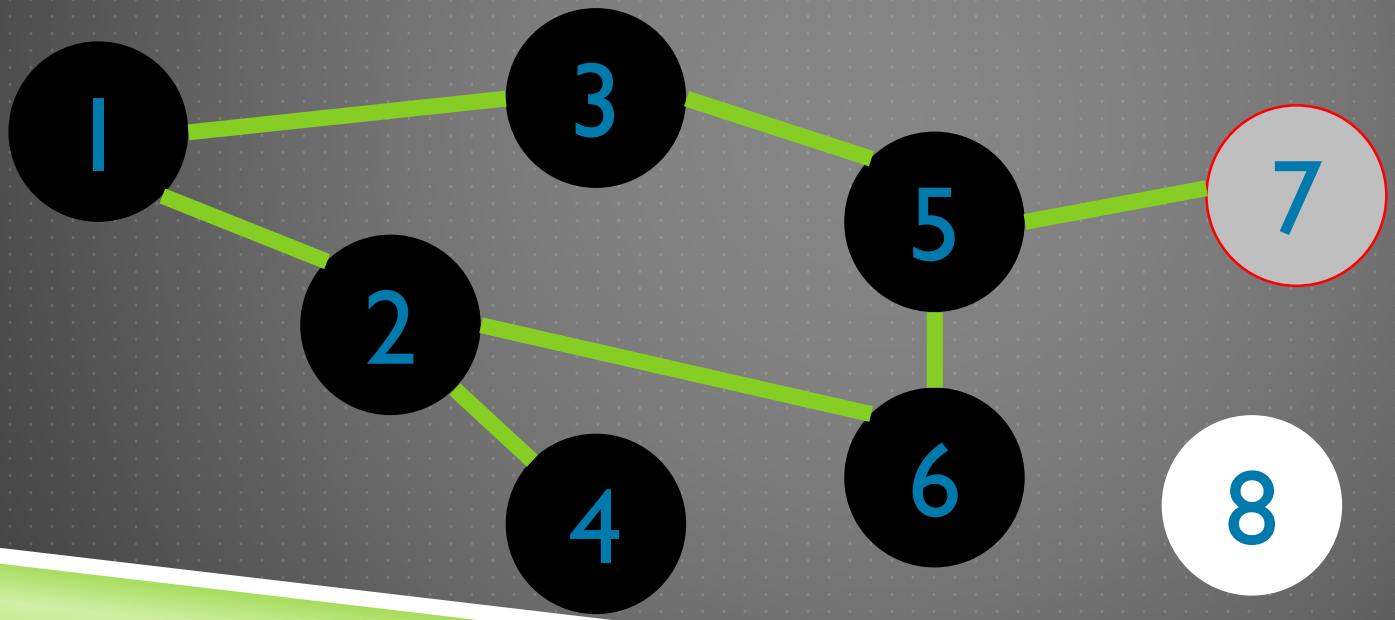
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



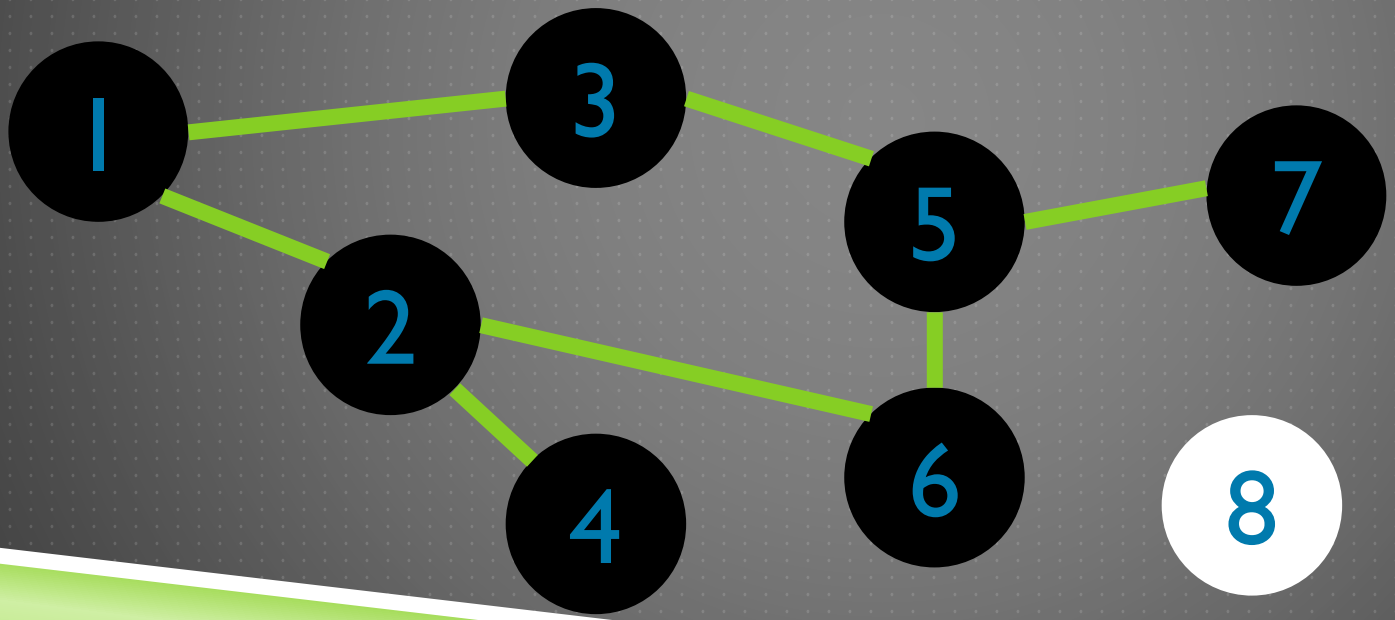
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



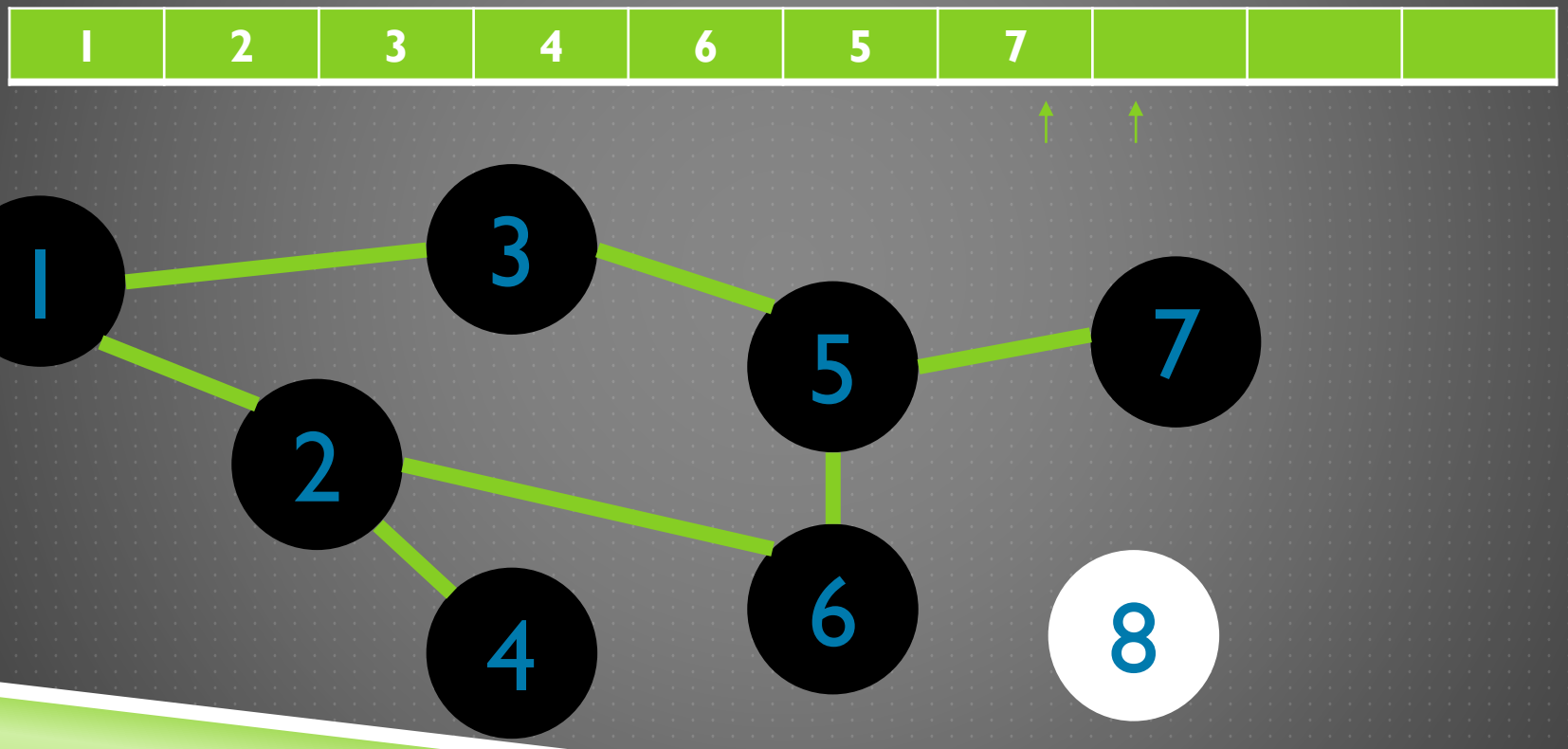
BREADTH-FIRST SEARCH - EXAMPLE

- … and so on



BREADTH-FIRST SEARCH - EXAMPLE

- Since the queue is empty now, the search can be stopped



BREADTH-FIRST SEARCH - CODE

```
Procedure bfs(vertex v){
    mark v is visited
    push v into the queue
    while the queue is not empty do
        t = front of the queue
        for all vertex w adjacent to t do
            if w is not visited
                mark w as visited
                push w into the queue
        pop t from the queue
}
```

BREADTH-FIRST SEARCH

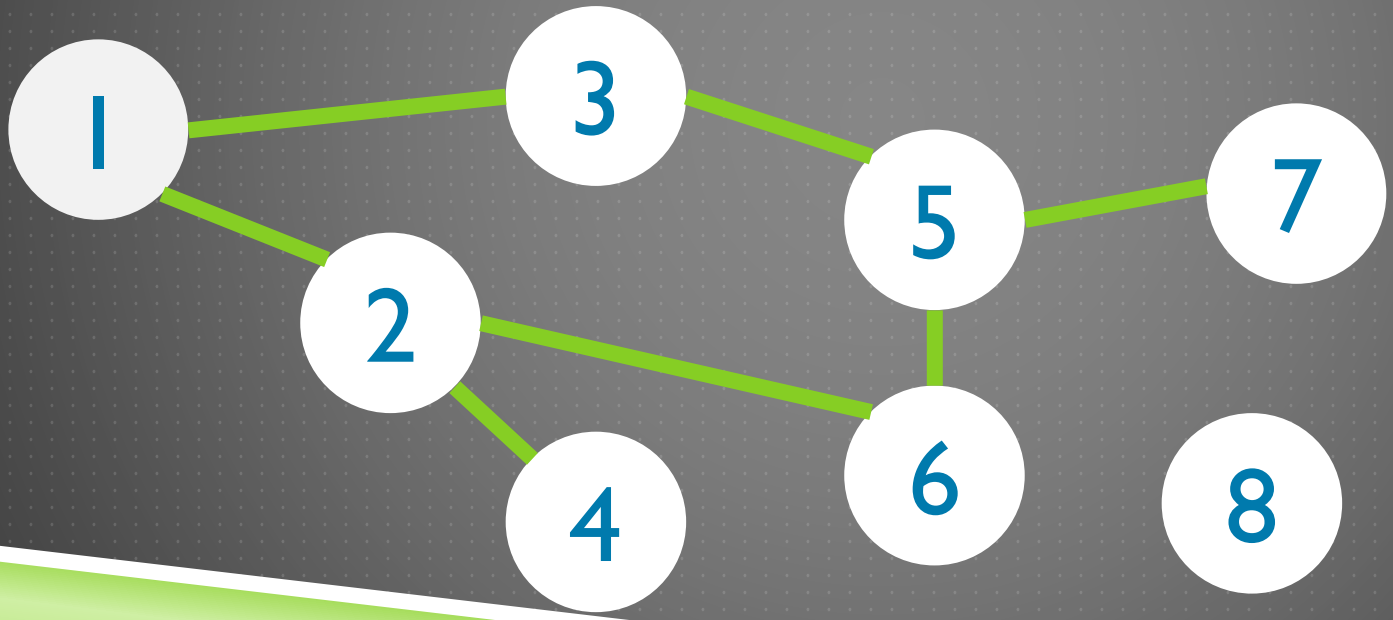
- Time Complexity:
  - $O(|V|^2)$ when using adjacency matrix
  - $O(|V|+|E|)$ when using adjacency list / edge list

BREADTH-FIRST SEARCH - APPLICATION

- Find the shortest path of all nodes from one node, with path length measured by number of edges

BREADTH-FIRST SEARCH - EXAMPLE

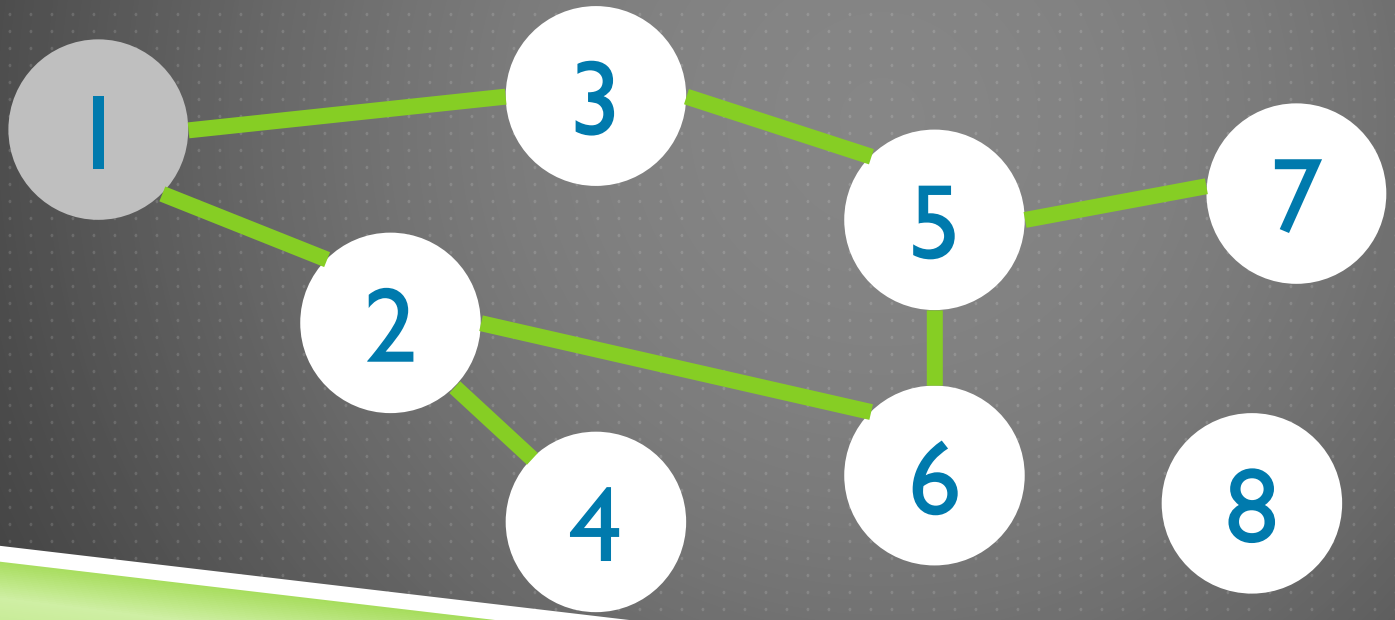
- Find the shortest paths from node 1



BREADTH-FIRST SEARCH - EXAMPLE

1								

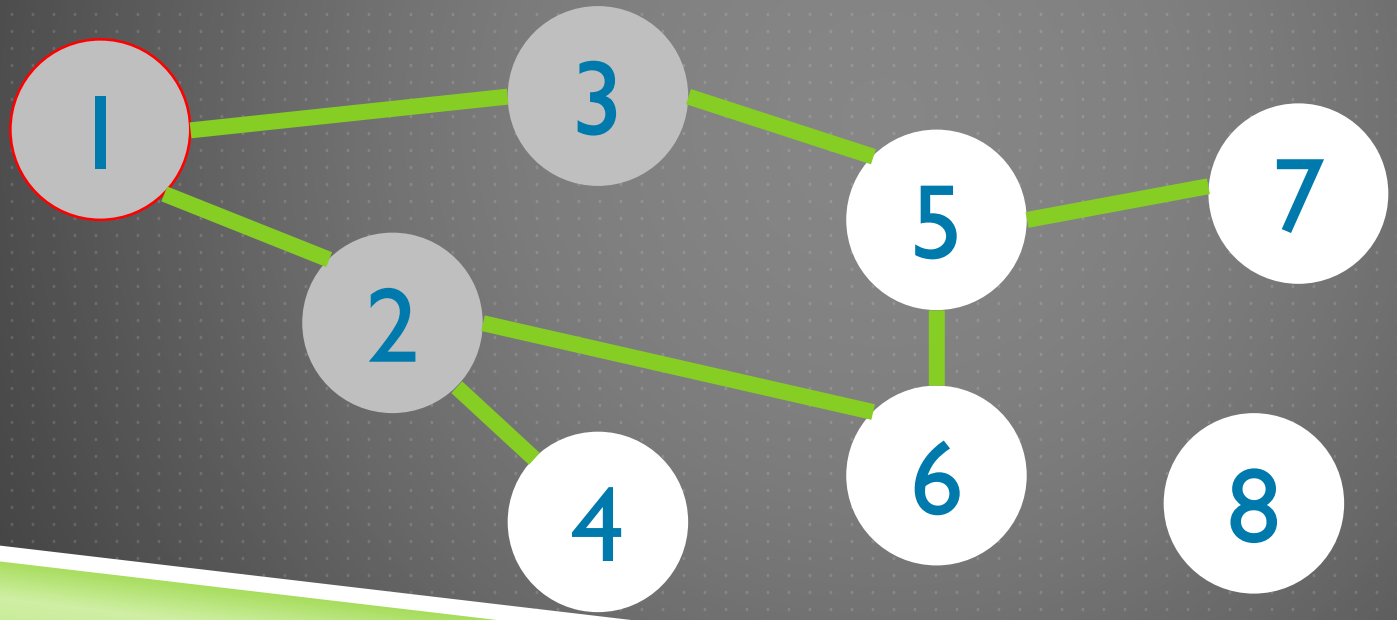
Node	1	2	3	4	5	6	7	8
Distance	0	∞	∞	∞	∞	∞	∞	∞



BREADTH-FIRST SEARCH - EXAMPLE

1	2	3							

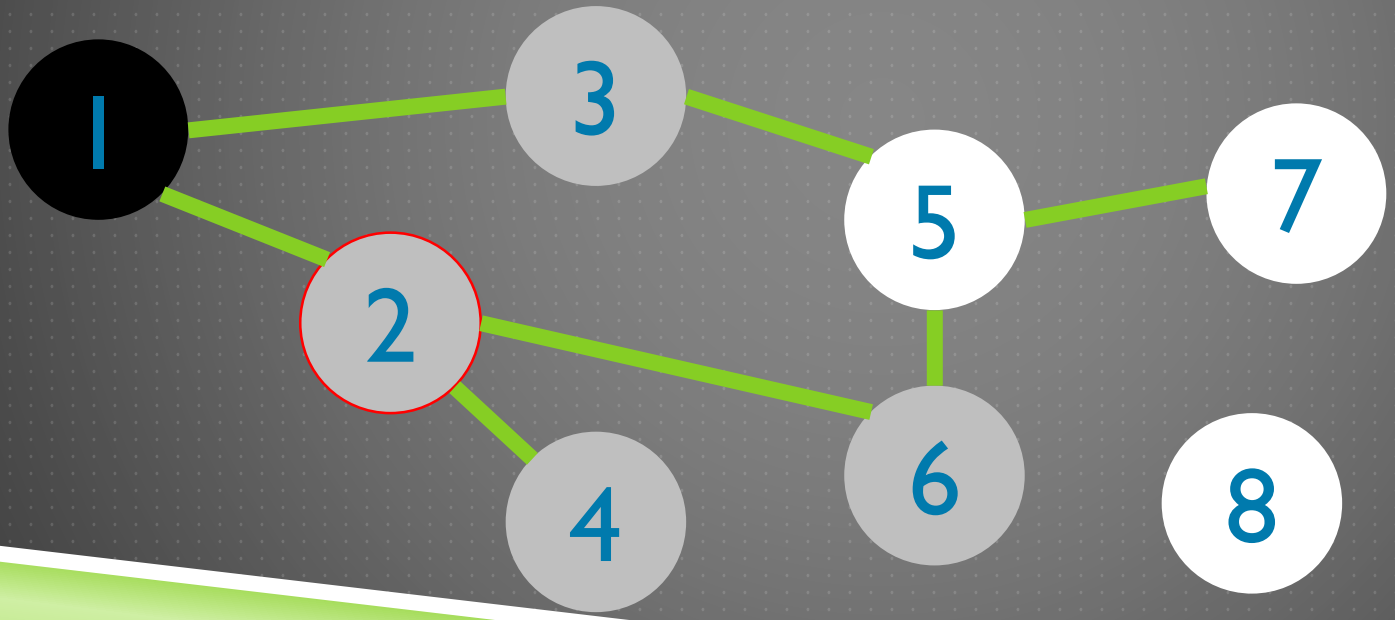
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	∞	∞	∞	∞	∞



BREADTH-FIRST SEARCH - EXAMPLE

1	2	3	4	6					

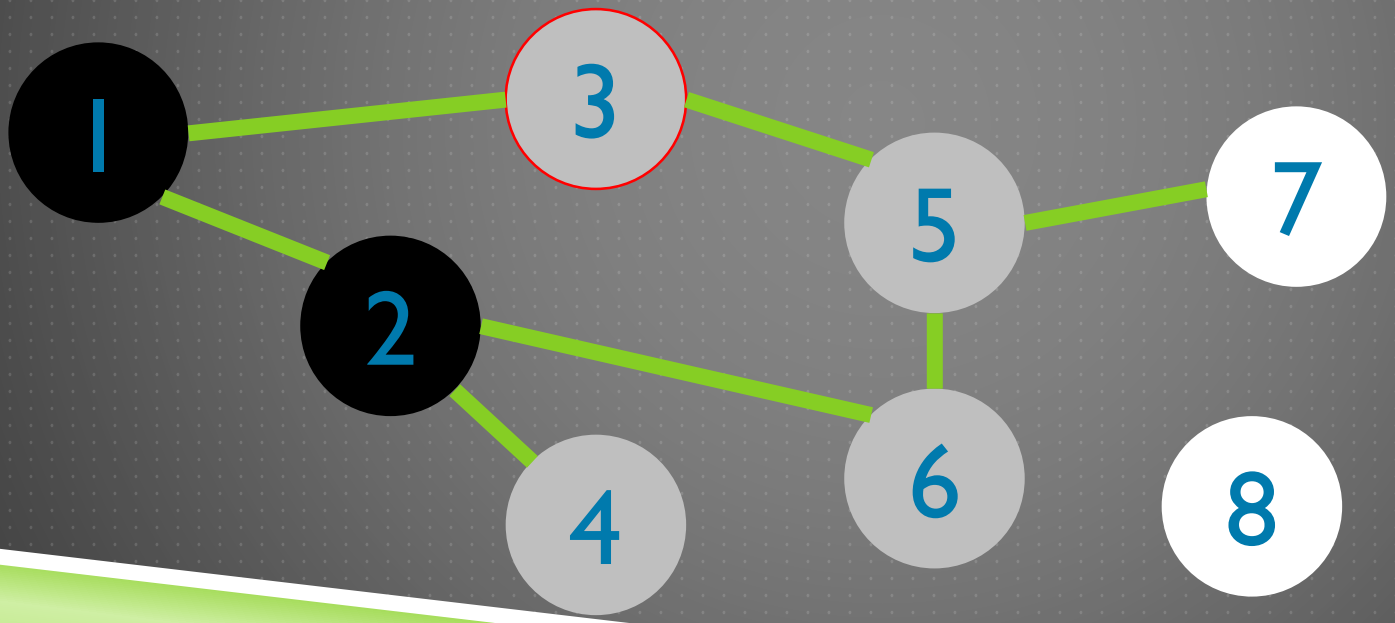
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	∞	2	∞	∞



BREADTH-FIRST SEARCH - EXAMPLE

1	2	3	4	6	5				

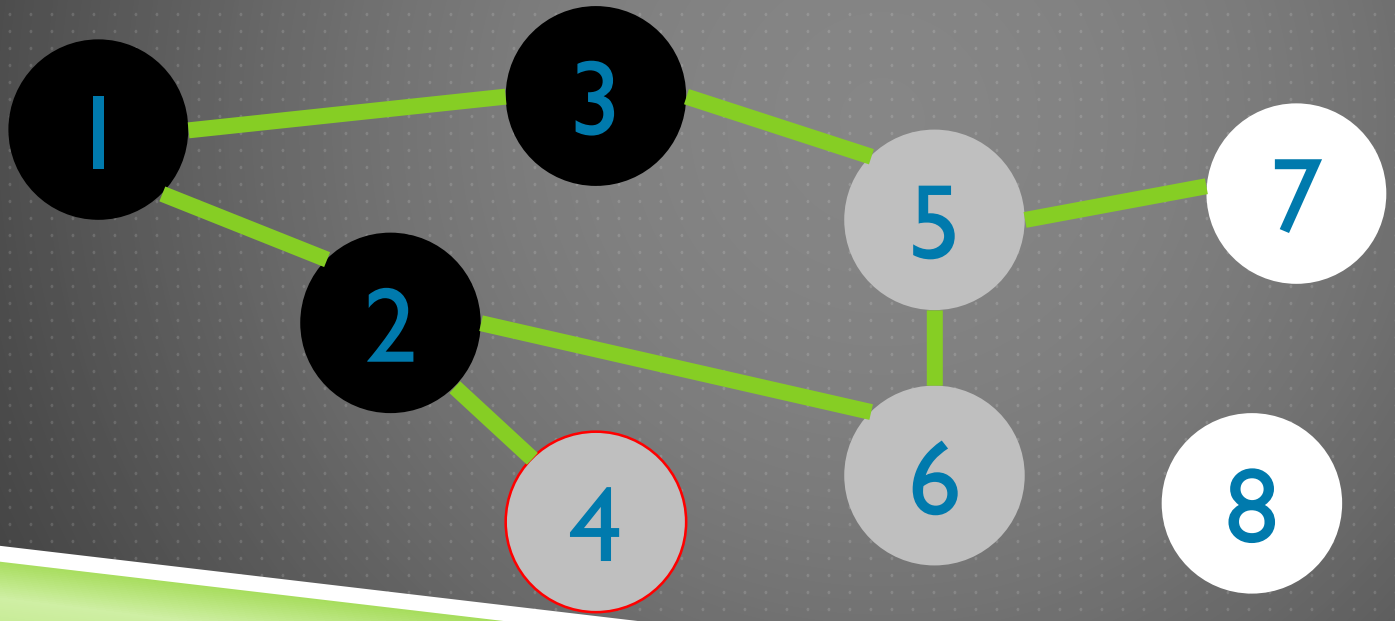
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	∞	∞



BREADTH-FIRST SEARCH - EXAMPLE

1	2	3	4	6	5				

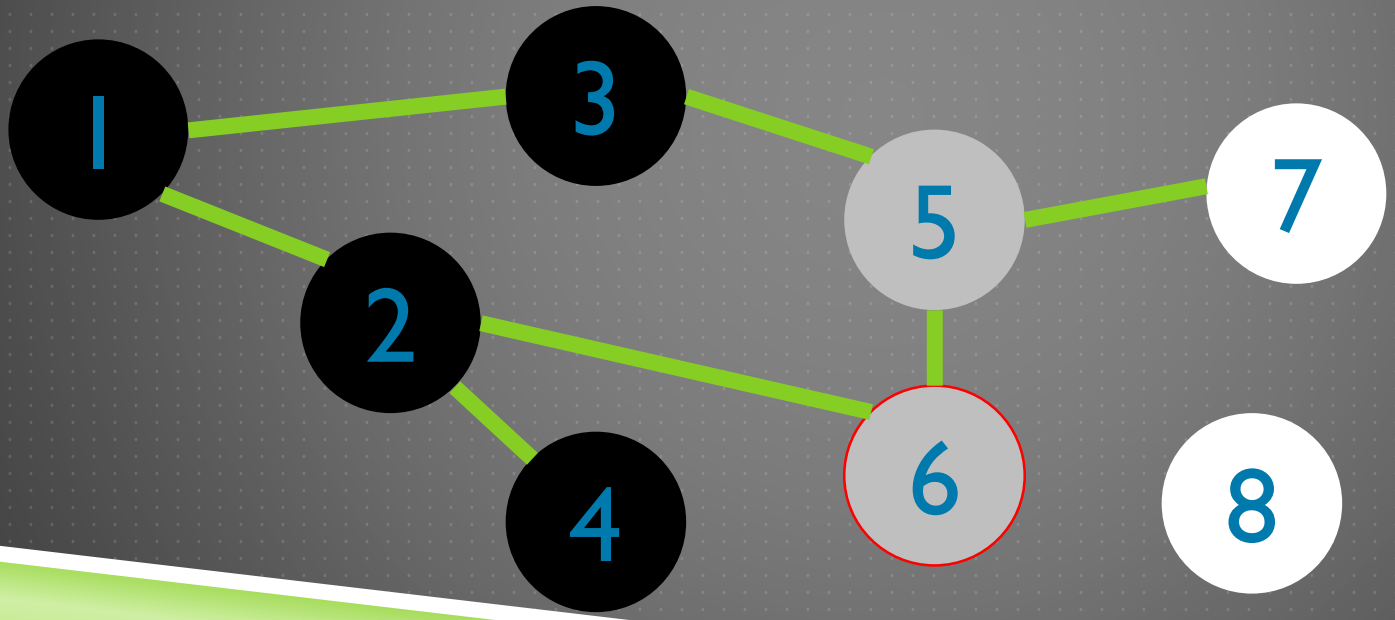
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	∞	∞



BREADTH-FIRST SEARCH - EXAMPLE

1	2	3	4	6	5				

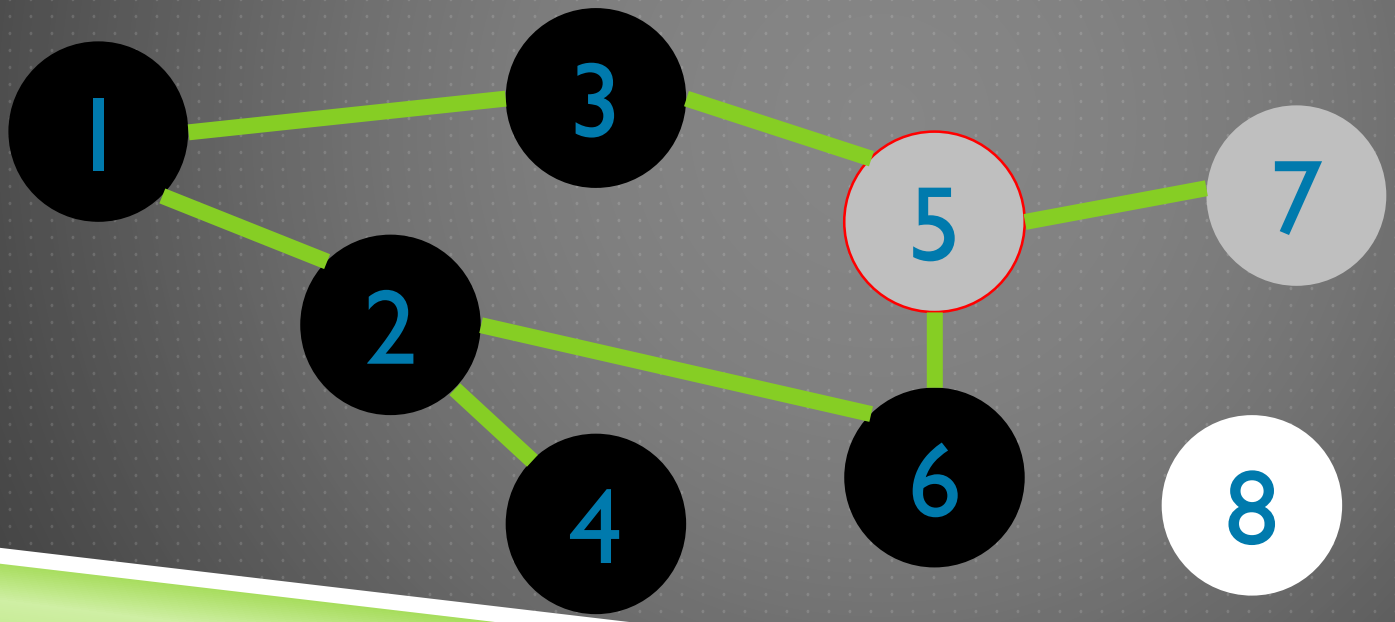
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	∞	∞



BREADTH-FIRST SEARCH - EXAMPLE

1	2	3	4	6	5	7			

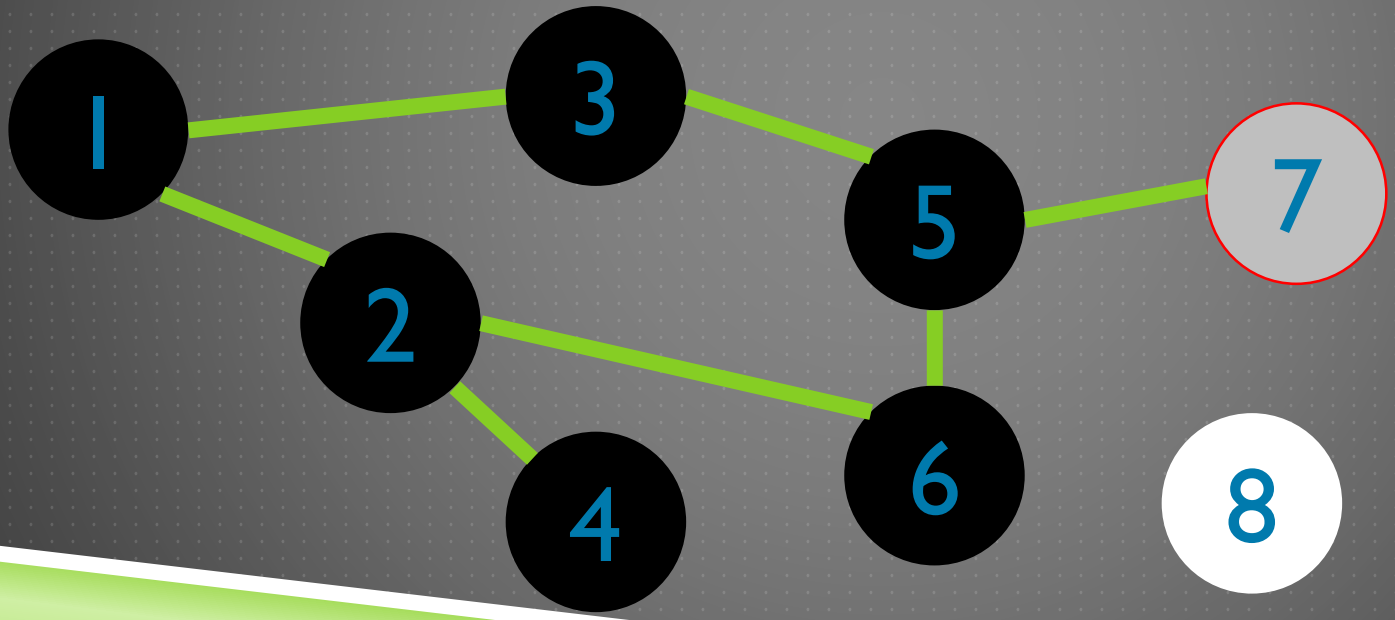
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	3	∞



BREADTH-FIRST SEARCH - EXAMPLE

1	2	3	4	6	5	7			

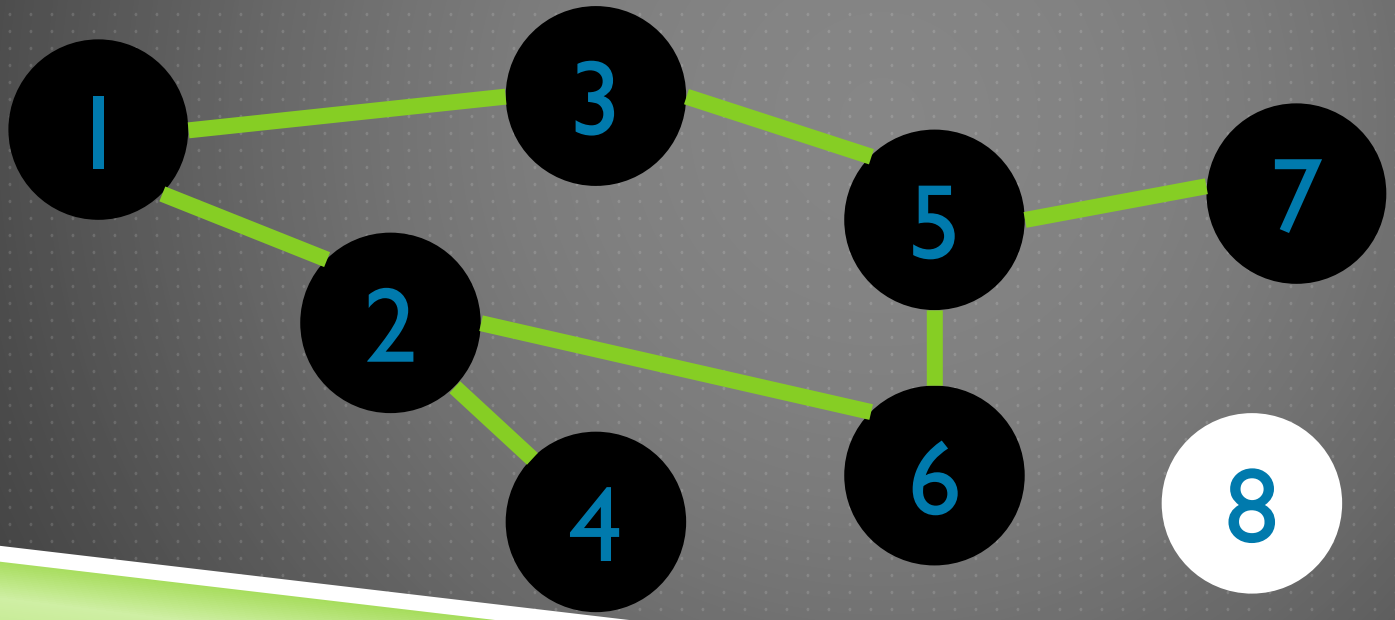
Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	3	∞



BREADTH-FIRST SEARCH - EXAMPLE

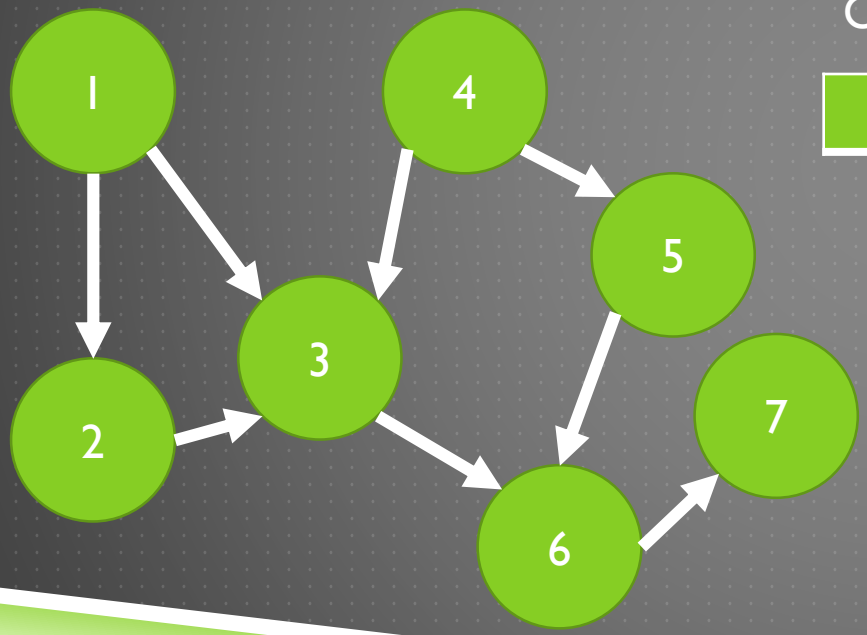
1	2	3	4	6	5	7			

Node	1	2	3	4	5	6	7	8
Distance	0	1	1	2	2	2	3	∞



TOPOLOGICAL SORT

- Find a linear ordering of the vertices such that for every directed edge from vertex u to vertex v, u comes before v in the ordering



One of the topological orders:

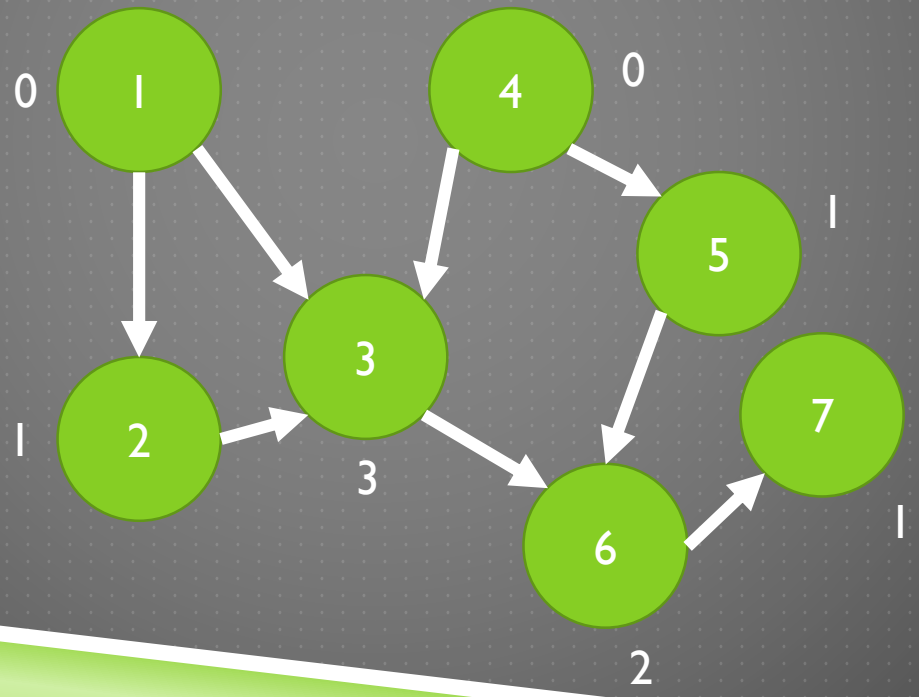
1	4	2	3	5	6	7

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

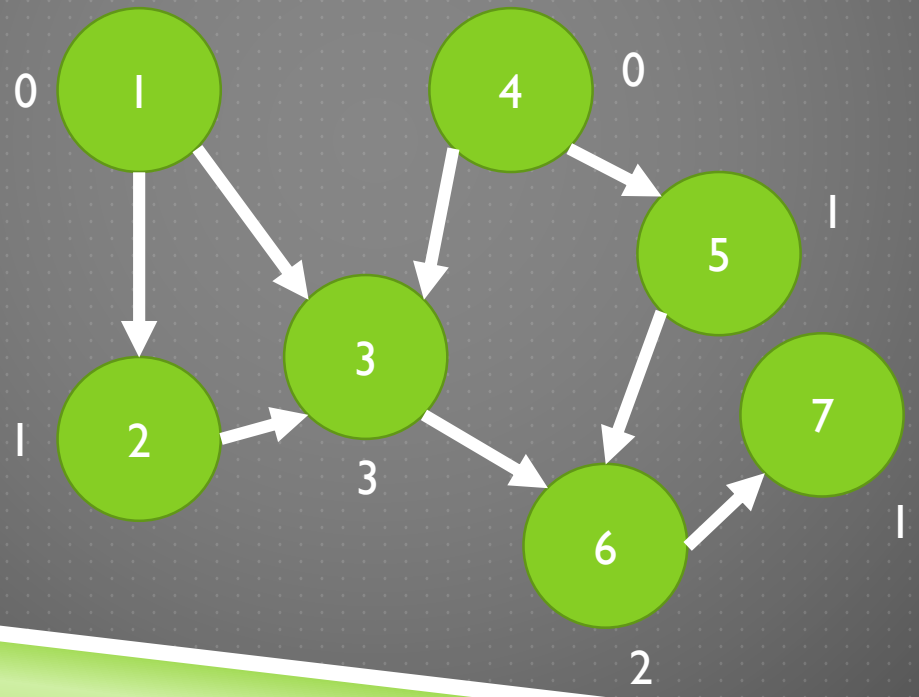
TOPOLOGICAL SORT - EXAMPLE

- Topological order:



TOPOLOGICAL SORT - EXAMPLE

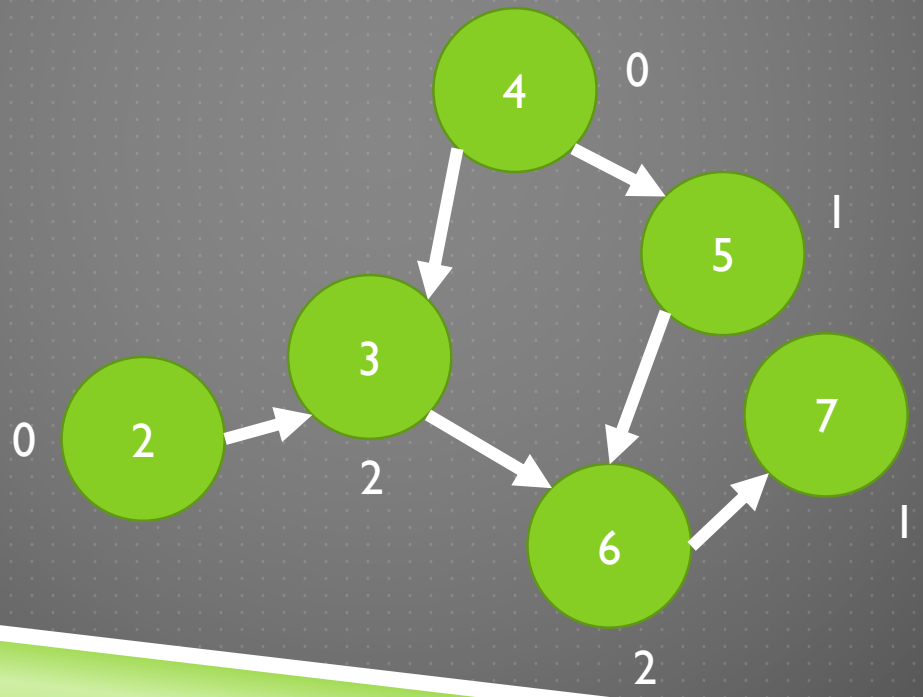
- Topological order:



TOPOLOGICAL SORT - EXAMPLE

- Topological order:

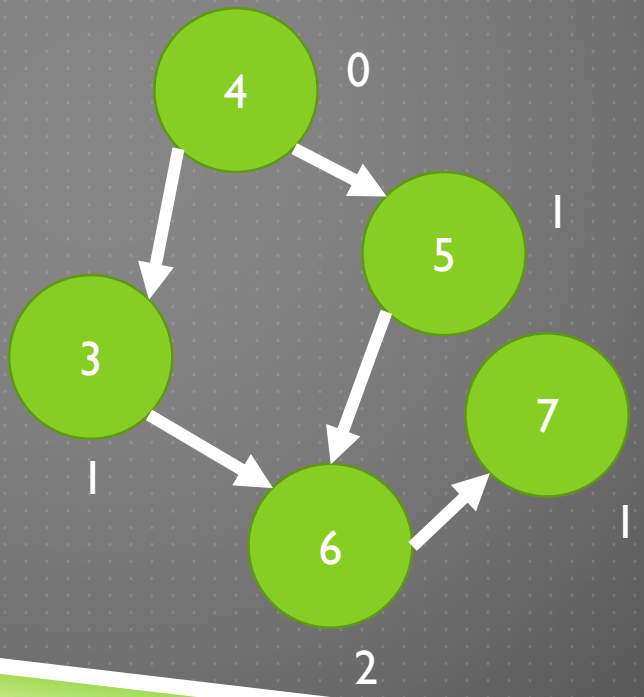
1	2					



TOPOLOGICAL SORT - EXAMPLE

- Topological order:

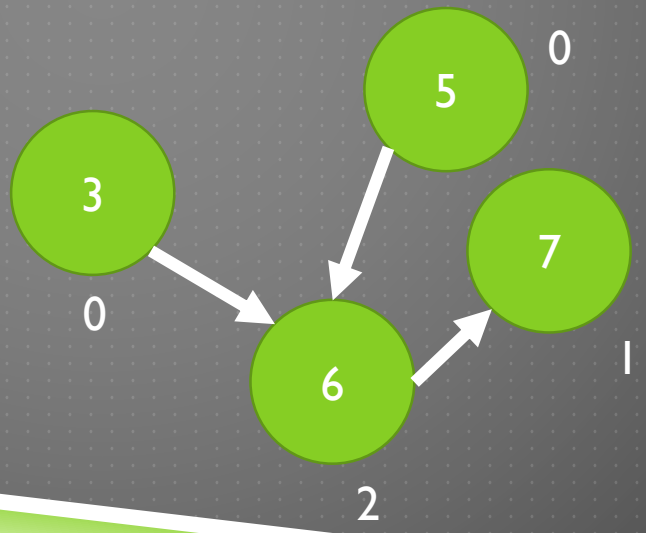
1	2	4				



TOPOLOGICAL SORT - EXAMPLE

- Topological order:

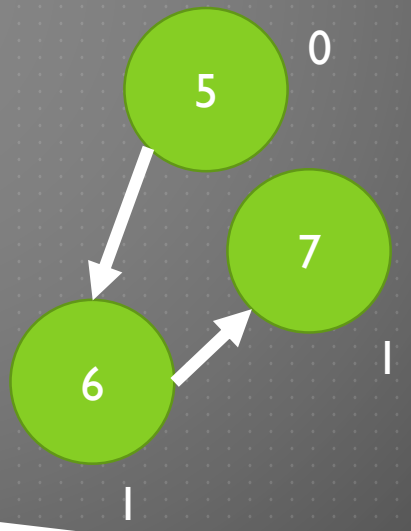
1	2	4	3			



TOPOLOGICAL SORT - EXAMPLE

- Topological order:

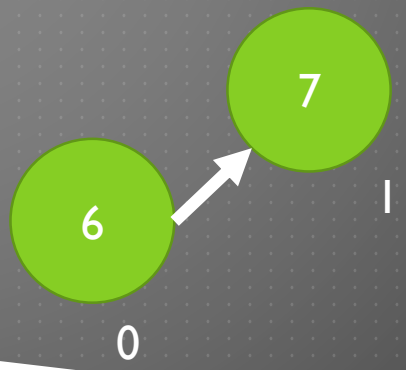
1	2	4	3	5		



TOPOLOGICAL SORT - EXAMPLE

- Topological order:

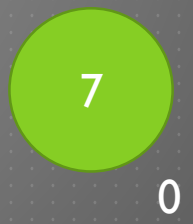
1	2	4	3	5	6	



TOPOLOGICAL SORT - EXAMPLE

- Topological order:

1	2	4	3	5	6	7



TOPOLOGICAL SORT

- Time complexity : O(|V|+|E|)
- Application
  - Determine whether a graph is a directed acyclic graph (DAG)

PROBLEMS

- 01067 Maze
- S042 Teacher's Problem
- T022 Bomber Man
- 01075 Sokoban