

CSE 421: Introduction to Algorithms

BFS

Shayan Oveis Gharan

Sparse Graphs

A graph is called **sparse** if $m \ll n^2$ and it is called **dense** otherwise.

Sparse graphs are very common in practice

- Friendships in social network
- Planar graphs
- Web graph

Q: Which is a better running time $O(n + m)$ vs $O(n^2)$?

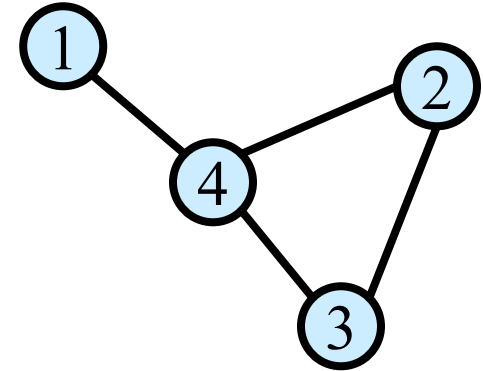
A: $O(n + m) \leq O(n^2)$, but $O(n + m)$ is usually much better.

Storing Graphs (Internally in ALG)

Vertex set $V = \{v_1, \dots, v_n\}$.

Adjacency Matrix: A

- For all, $i, j, A[i, j] = 1$ iff $(v_i, v_j) \in E$
- Storage: n^2 bits



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

Advantage:

- $O(1)$ test for presence or absence of edges

Disadvantage:

- Inefficient for sparse graphs both in storage and edge-access

Storing Graphs (Internally in ALG)

Adjacency List:

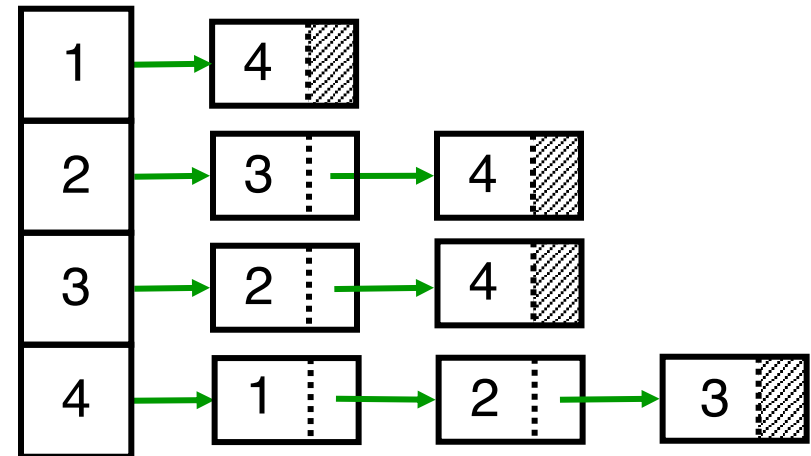
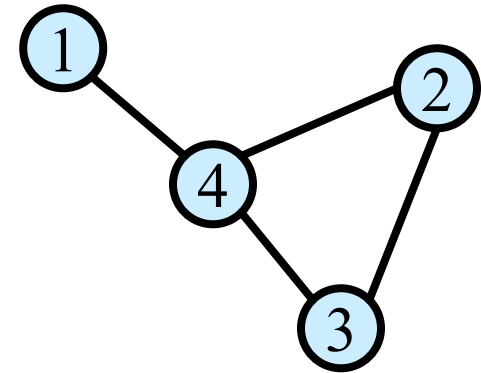
$O(n+m)$ words

Advantage

- Compact for sparse
- Easily see all edges

Disadvantage

- No $O(1)$ edge test
- More complex data structure



Storing Graphs (Internally in ALG)

Adjacency List:

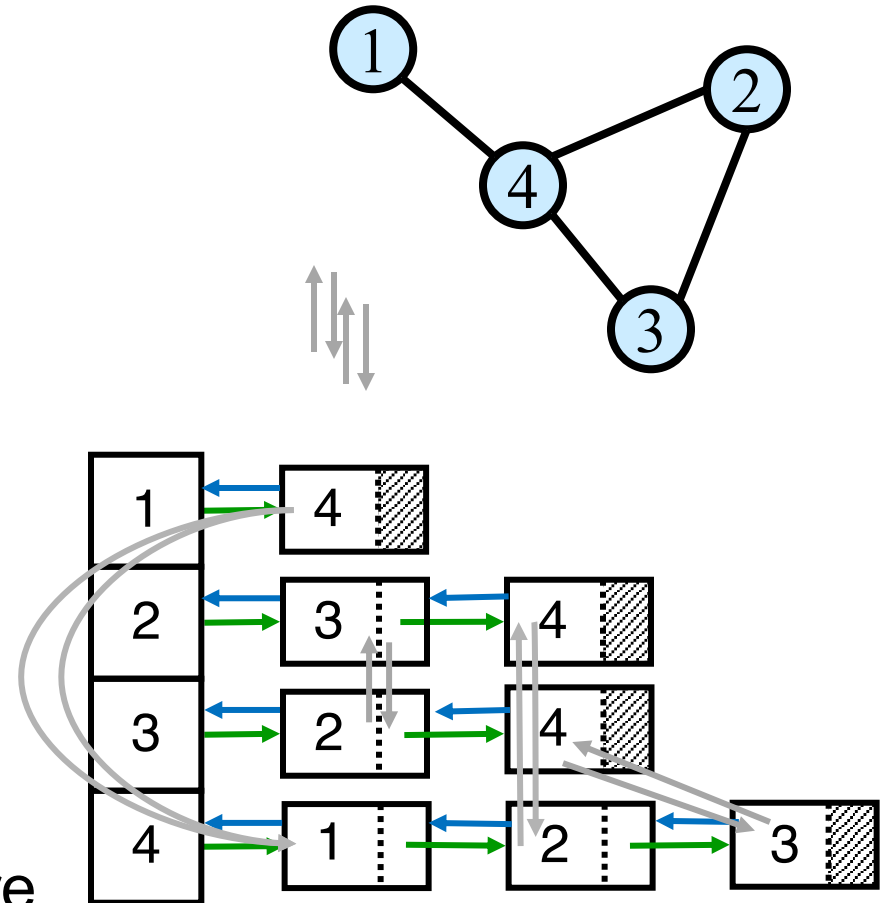
$O(n+m)$ words

Advantage

- Compact for sparse
- Easily see all edges

Disadvantage

- No $O(1)$ edge test
- More complex data structure



Graph Traversal

Walk (via edges) from a fixed starting vertex s to all vertices reachable from s .

- Breadth First Search (BFS): Order nodes in successive layers based on distance from s
- Depth First Search (DFS): More natural approach for exploring a maze; many efficient algs build on it.

Applications:

- Finding Connected components of a graph
- Testing Bipartiteness
- Finding Articulation points

Breadth First Search (BFS)

Completely **explore** the vertices in order of their distance from s .

Three states of vertices:

- Undiscovered
- **Discovered**
- **Fully-explored**

Naturally implemented using a queue

The queue will always have the list of Discovered vertices

BFS implementation

Global initialization: mark all vertices "undiscovered"

BFS(s)

mark s "discovered"

queue = { s }

while queue not empty

 u = remove_first(queue)

 for each edge {u,x}

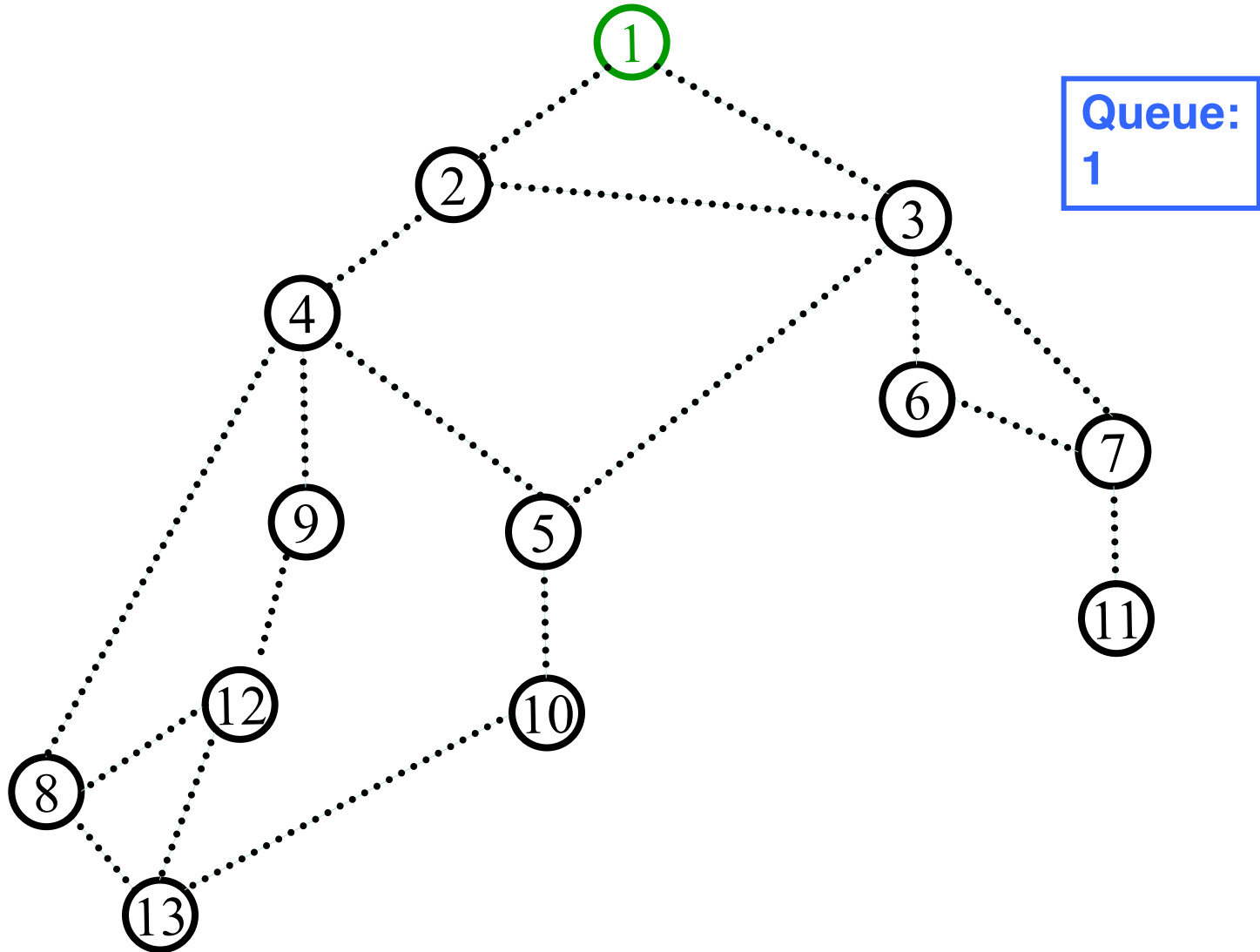
 if (x is undiscovered)

 mark x discovered

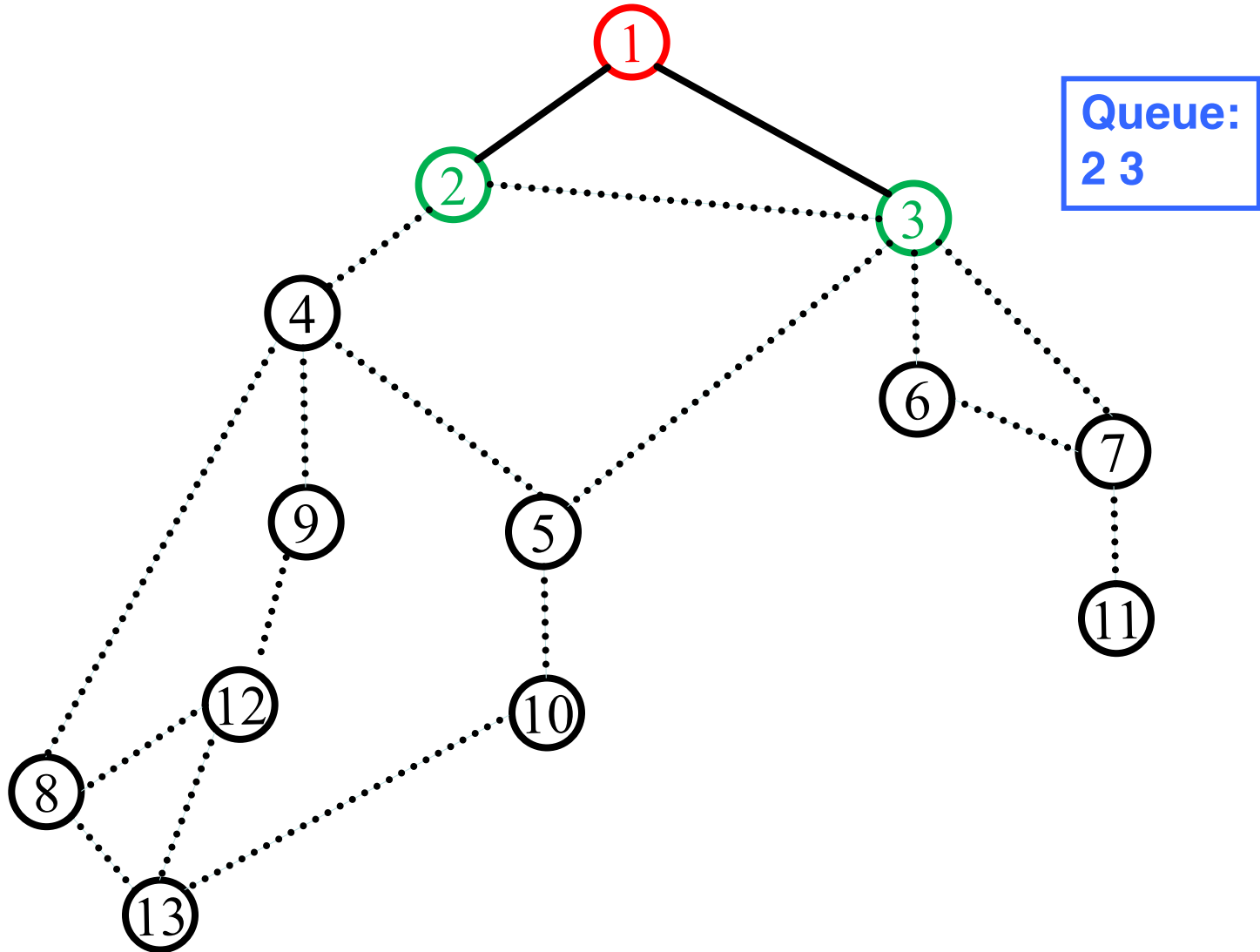
 append x on queue

 mark u fully-explored

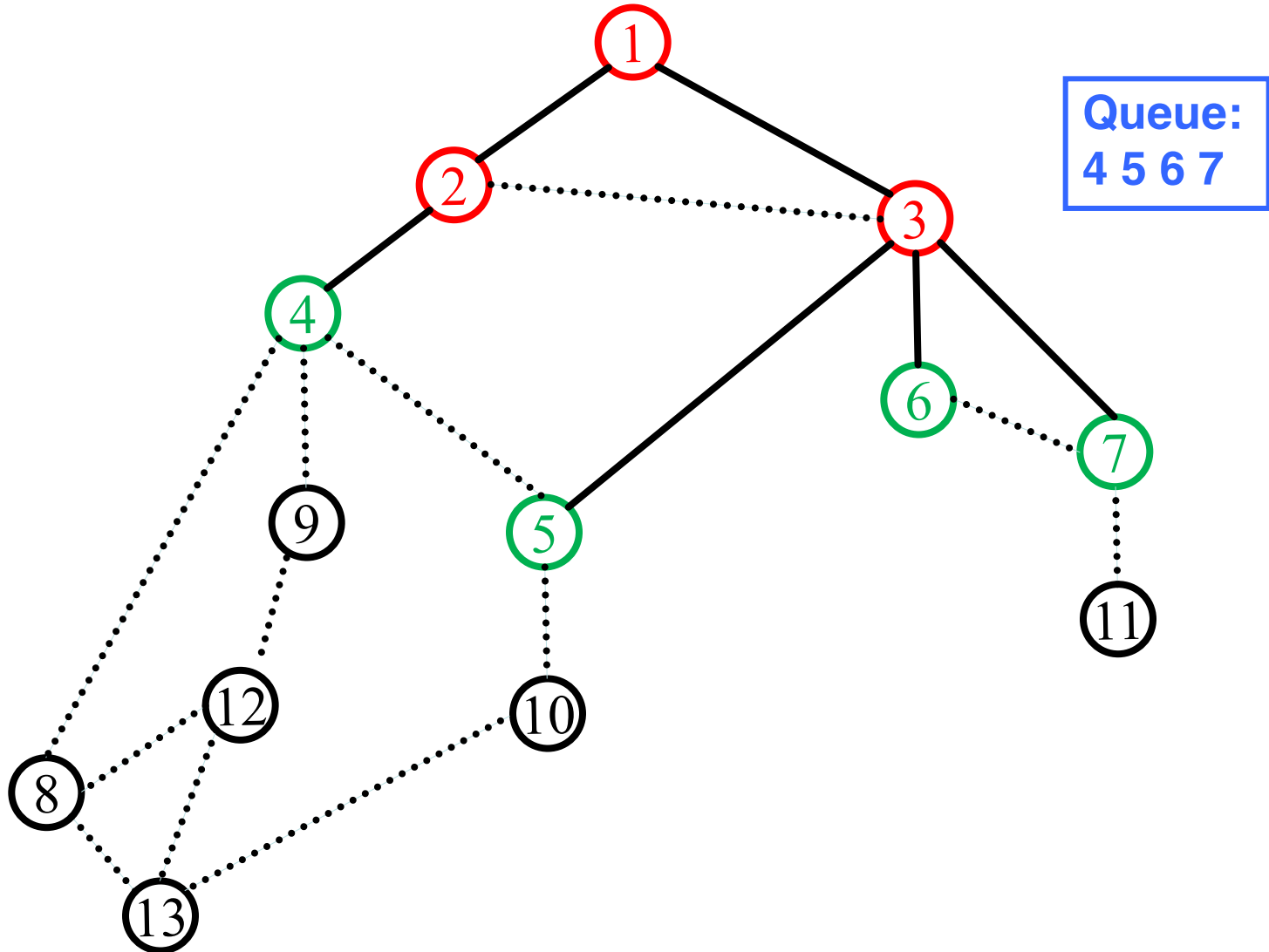
BFS(1)



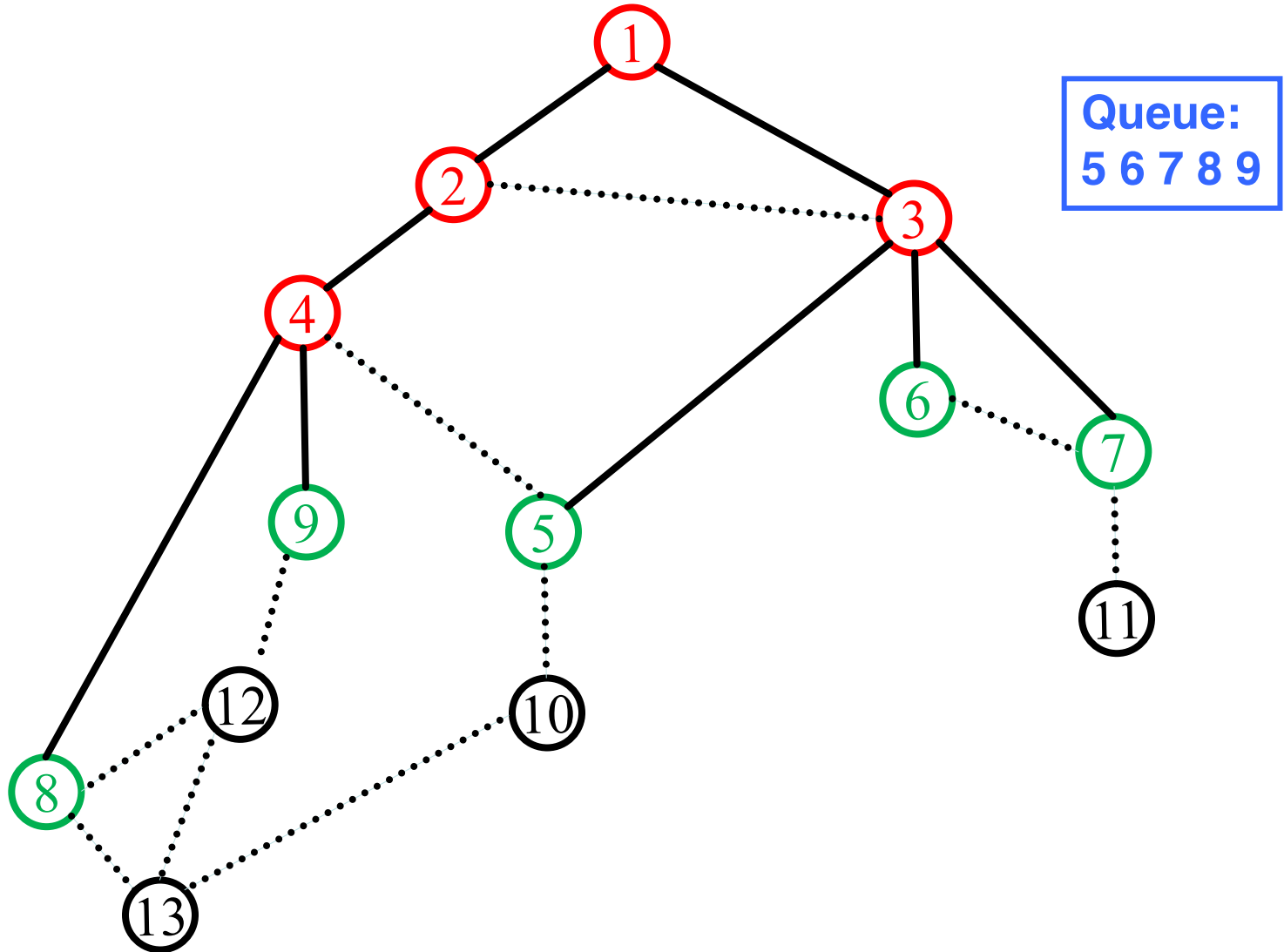
BFS(1)



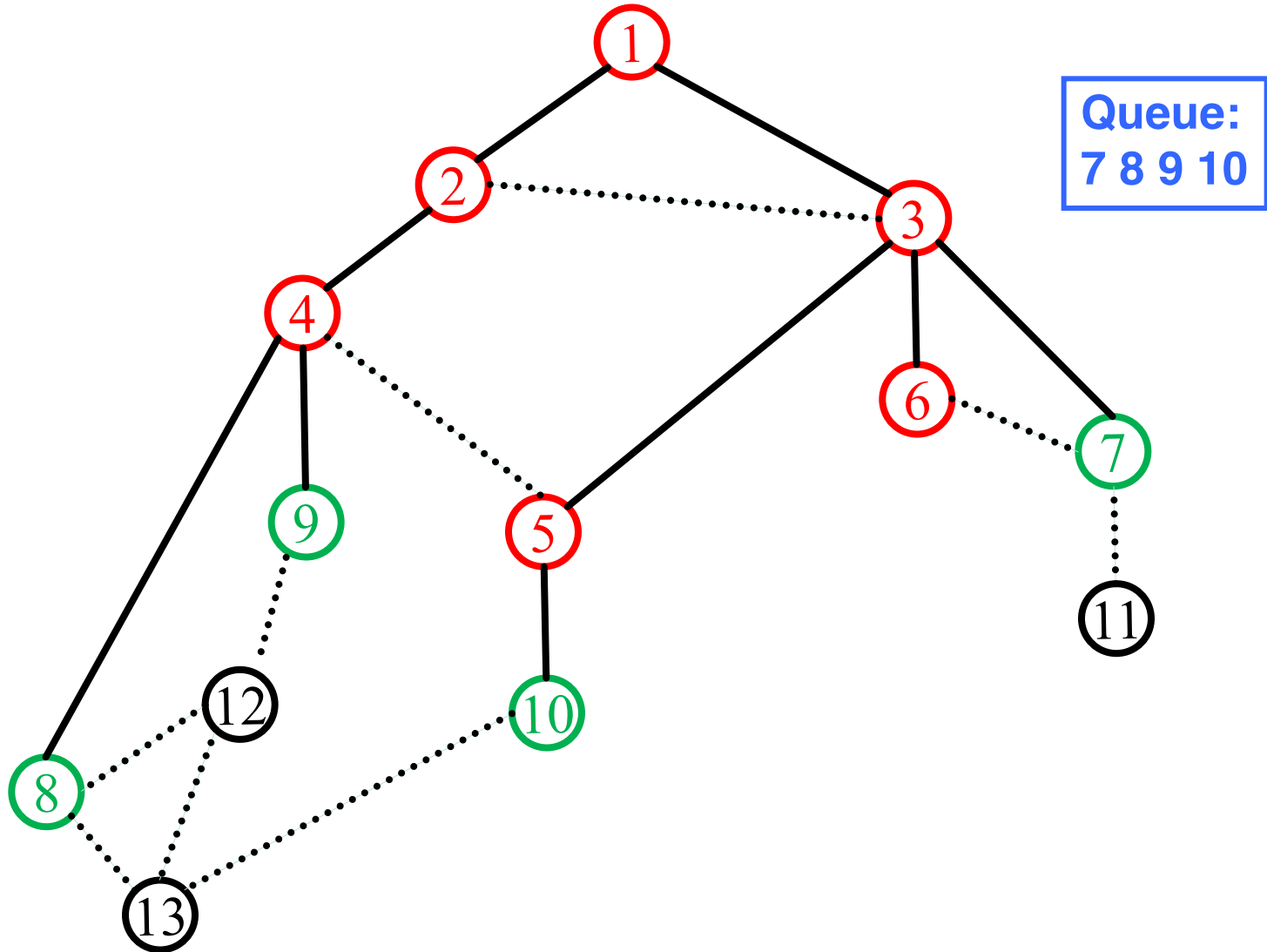
BFS(1)



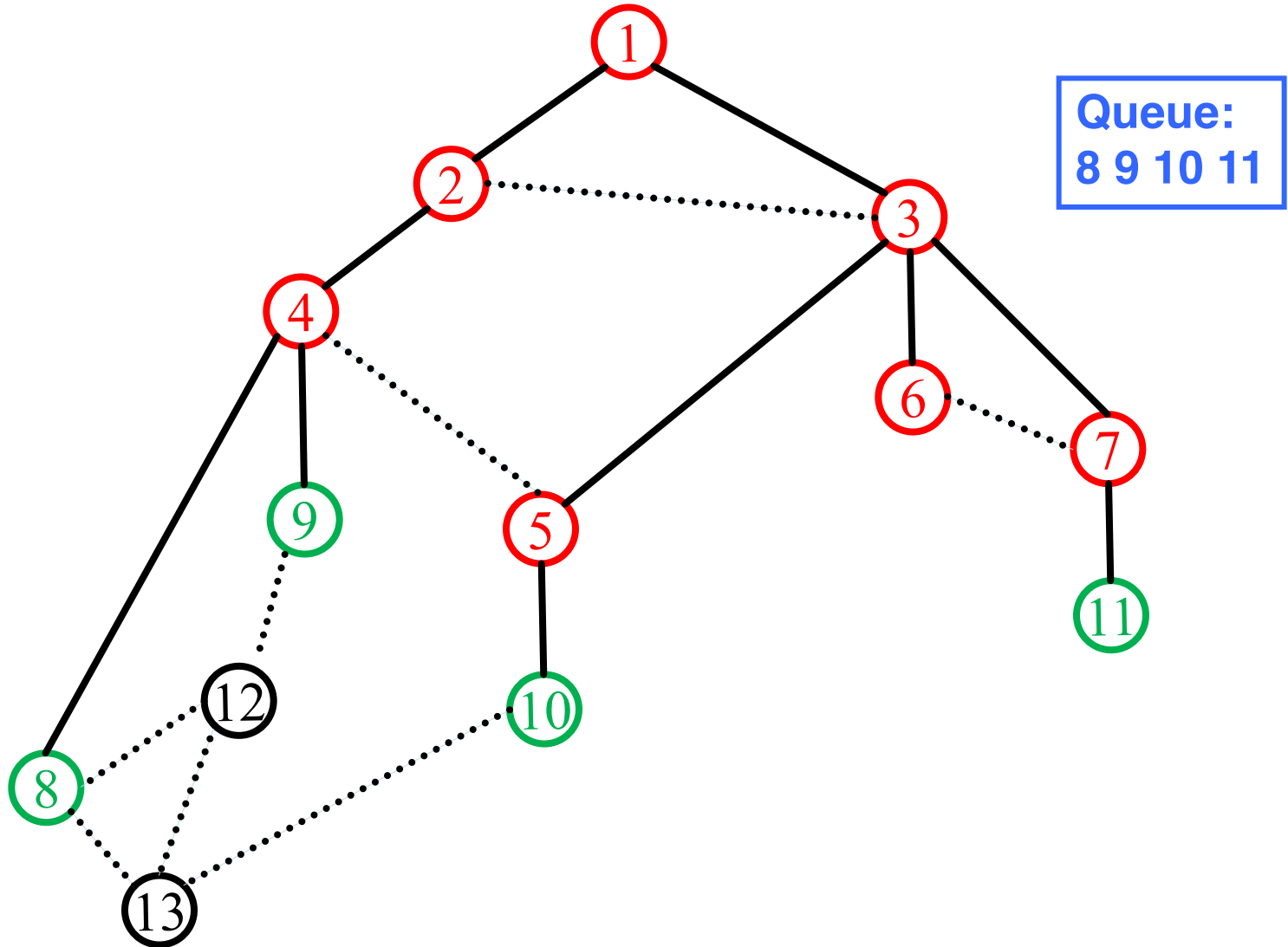
BFS(1)



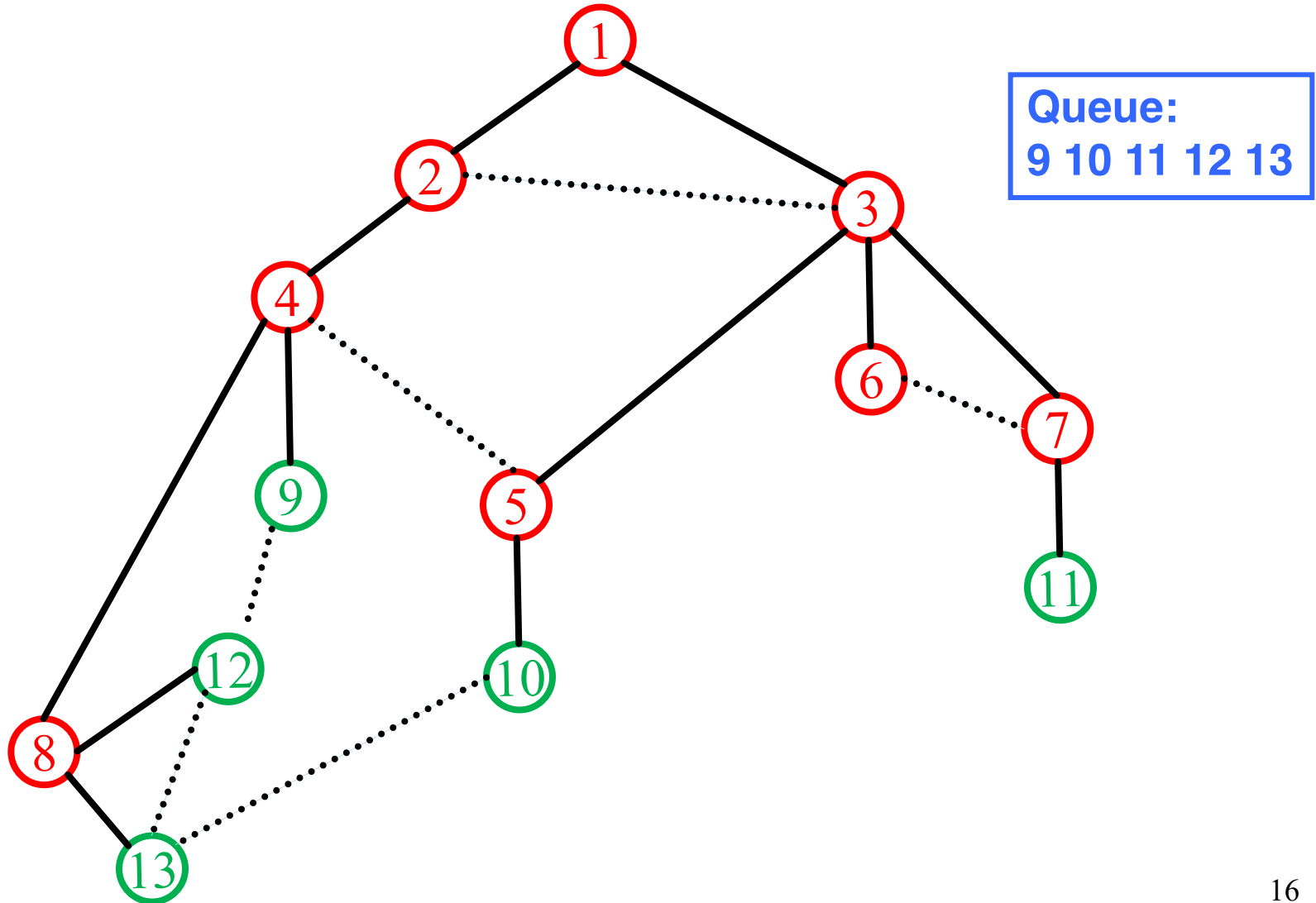
BFS(1)



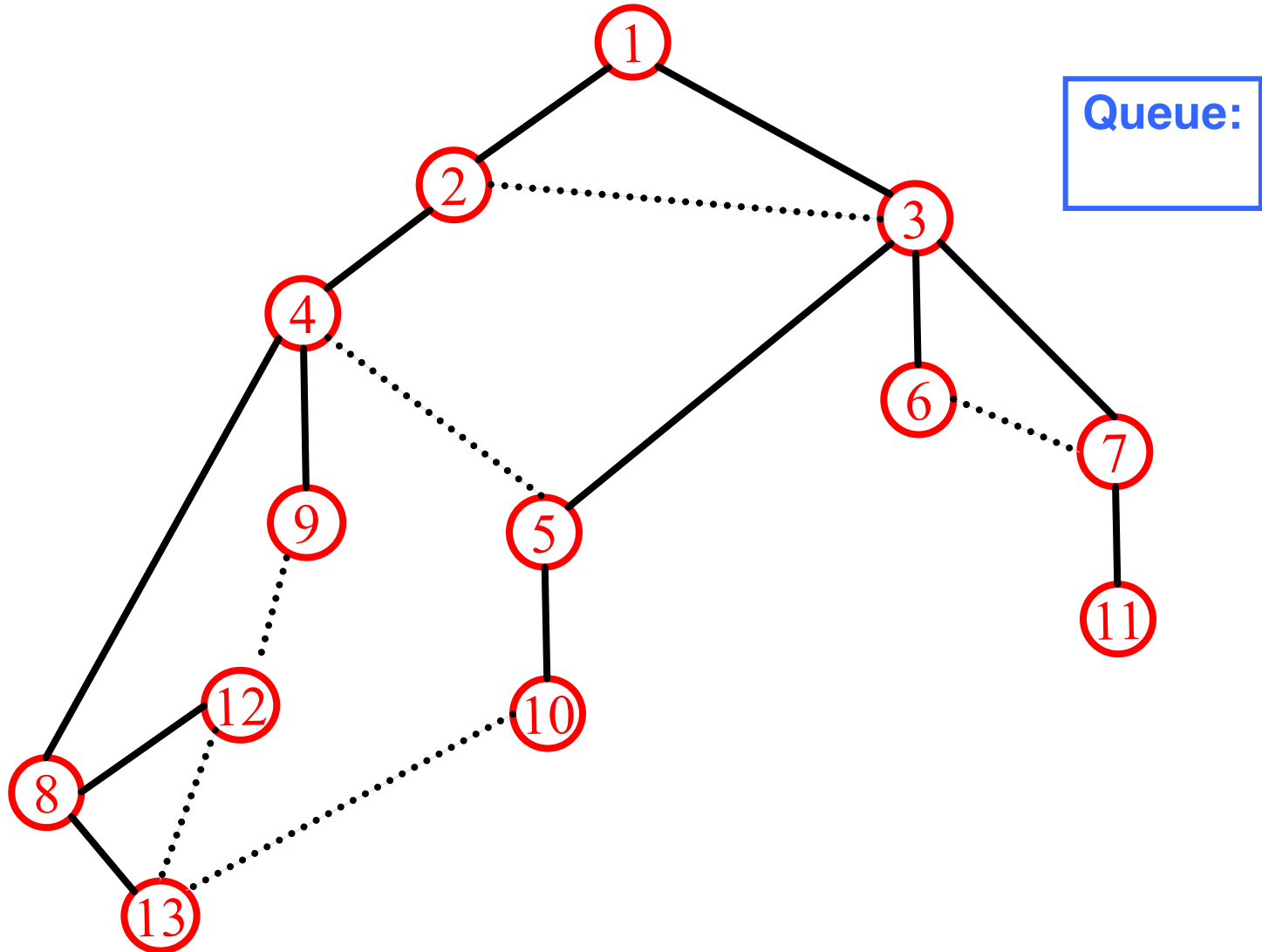
BFS(1)



BFS(1)



BFS(1)



BFS Analysis

Global initialization: mark all vertices "undiscovered"

BFS(s)

mark s discovered

queue = { s }

$O(n)$ times: Once from every vertex if G is connected

while queue not empty

u = remove_first(queue)

$\deg(u) \leq O(n)$ times

for each edge {u,x}

if (x is undiscovered)

mark x discovered

append x on queue

mark u fully-explored

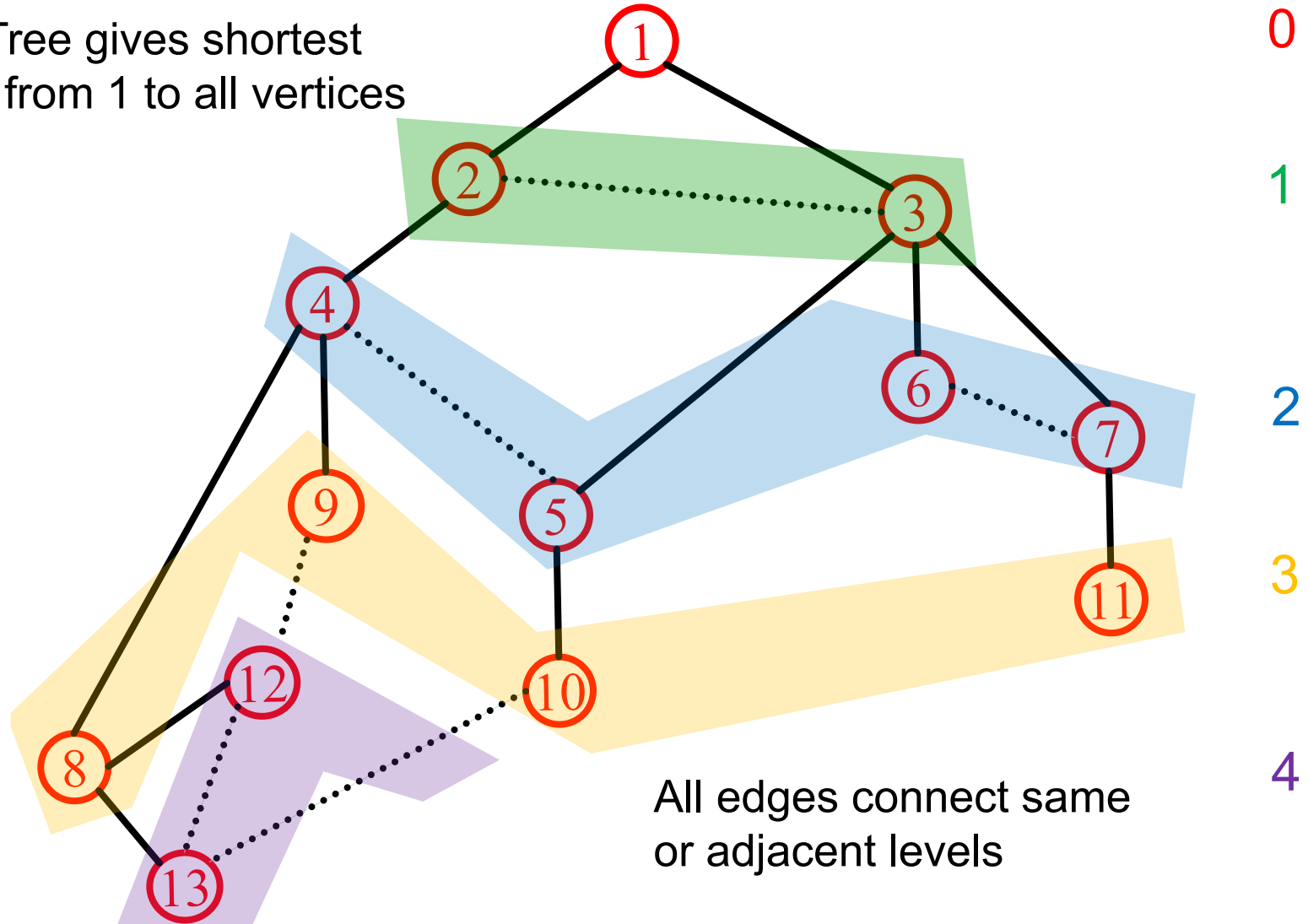
If we use adjacency list: $O(n) + O(\sum_v \deg(v)) = O(m + n)$

Properties of BFS

- **BFS(s)** visits a vertex v if and only if there is a path from s to v
- Edges into then-undiscovered vertices define a tree – the “Breadth First spanning tree” of G
- Level i in the tree are exactly all vertices v s.t., the shortest path (in G) from the root s to v is of length i
- **All edges** join vertices on the same or adjacent levels of the tree

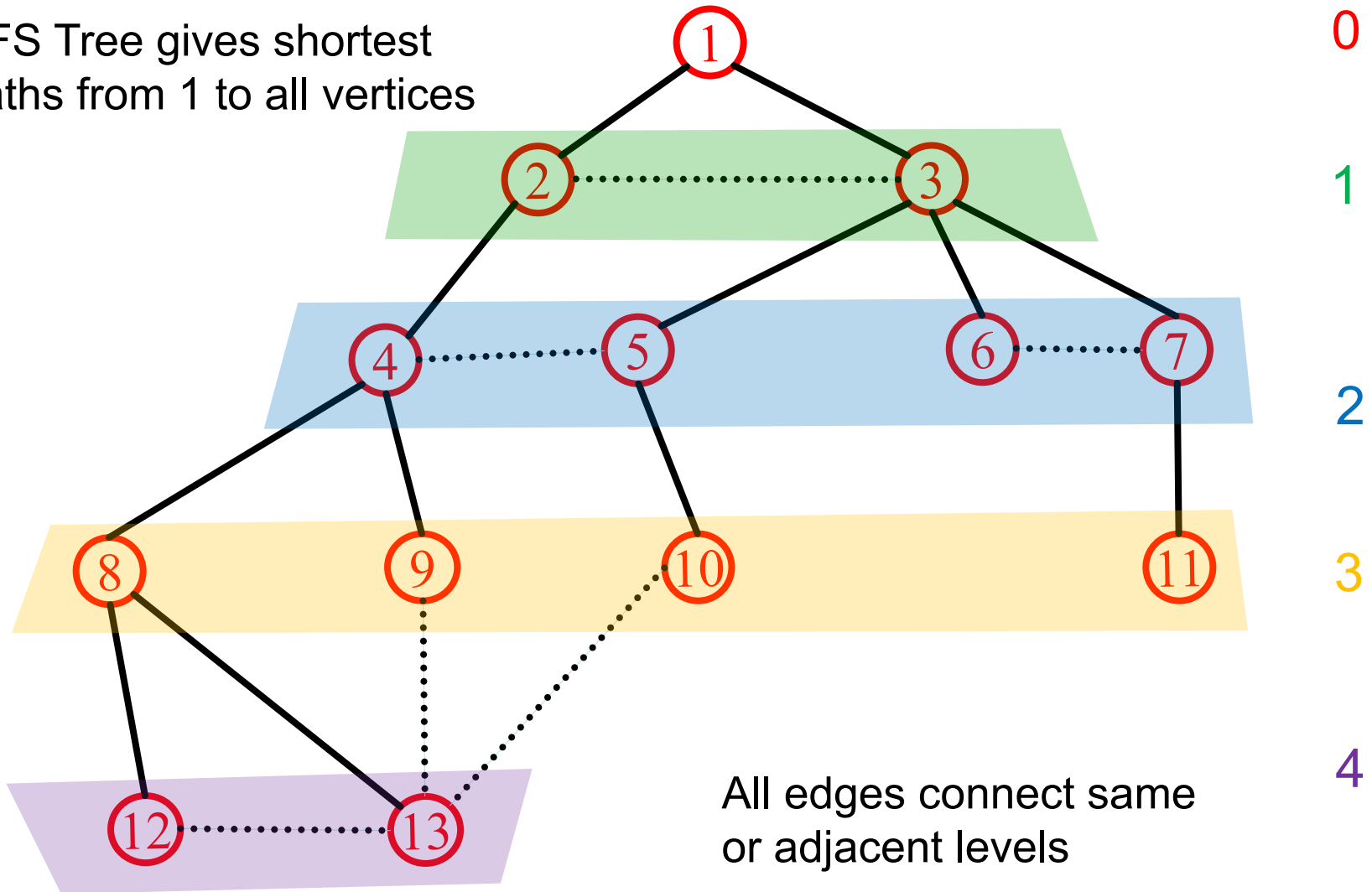
BFS Application: Shortest Paths

BFS Tree gives shortest paths from 1 to all vertices



BFS Application: Shortest Paths

BFS Tree gives shortest paths from 1 to all vertices



Simple Observations

Obs 1: Every vertex in the queue is marked discovered

Obs 2: When we process u at top of queue of level i , any undiscovered neighbor of u will be at level $i + 1$.

Obs 3: At any time, the vertices in the queue are either on a level i or $i + 1$ where vertices of level i are ahead of vertices of level $i + 1$.

Properties of BFS

Claim: All edges join vertices on the same or adjacent levels of the tree

Pf: Consider an edge $\{x,y\}$

Say x is first discovered and it is added to level i .

We show y will be at level i or $i + 1$

This is because when vertices incident to x are considered in the loop, if y is still undiscovered, it will be discovered and added to level $i + 1$.

Properties of BFS

Lemma: All vertices at level i of BFS(s) have shortest path distance i to s .

Claim: If $L(v) = i$ then shortest path $\leq i$

Pf: Because there is a path of length i from s to v in the BFS tree

Claim: If shortest path = i then $L(v) \leq i$

Pf: If shortest path = i , then say $s = v_0, v_1, \dots, v_i = v$ is the shortest path to v .

By previous claim,

$$\begin{aligned}L(v_1) &\leq L(v_0) + 1 \\L(v_2) &\leq L(v_1) + 1\end{aligned}$$

$$L(v_i) \leq \overset{\dots}{L(v_{i-1})} + 1$$

So, $L(v_i) \leq i$.

This proves the lemma.

Why Trees?

Trees are simpler than graphs

Many statements can be proved on trees by induction

So, computational problems on trees are simpler than general graphs

This is often a good way to approach a graph problem:

- Find a "nice" tree in the graph, i.e., one such that non-tree edges have some simplifying structure
- Solve the problem on the tree
- Use the solution on the tree to find a "good" solution on the graph

Graph Search App: Connected Comp

We want to answer the following type questions (**fast**):

Given vertices u, v is there a path from u to v in G ?

Idea: Create an array A such that

For all u , $A[u]$ is the label of the connected component that contains u

Therefore, question reduces to

If $A[u] = A[v]$?

Connected Components Implementation

Initial State: All vertices undiscovered, $c \leftarrow 0$
for $v = 1$ to n do
 If $\text{state}(v) \neq \text{fully-explored}$ then
 BFS(v): setting $A[u] \leftarrow c$ for each u found
 (and marking u discovered/fully-explored)
 $c \leftarrow c + 1$

Note: We no longer initialize to undiscovered in the BFS subroutine

Total Cost: $O(m+n)$

In every connected component with n_i vertices and m_i edges BFS takes time $O(m_i + n_i)$.

Connected Components

Lesson: We can execute any algorithm on disconnected graphs by running it on each connected component.

We can use the previous algorithm to detect connected components.

There is no overhead, because the algorithm runs in time $O(m+n)$.

So, from now on, we can (almost) always assume the input graph is **connected**.