

Final Exam Presentation

Shortest Path Algorithm

Analysis

Member Info

Department | CSE

陳伯豪 | s1131525

黃博彥 | s1131406

黃以慈 | s1133313

Report Content

What is Shortest Path

Theory of BFS

Theory of DFS

Dijkstra

Bellman-Ford

Floyd-Warshall

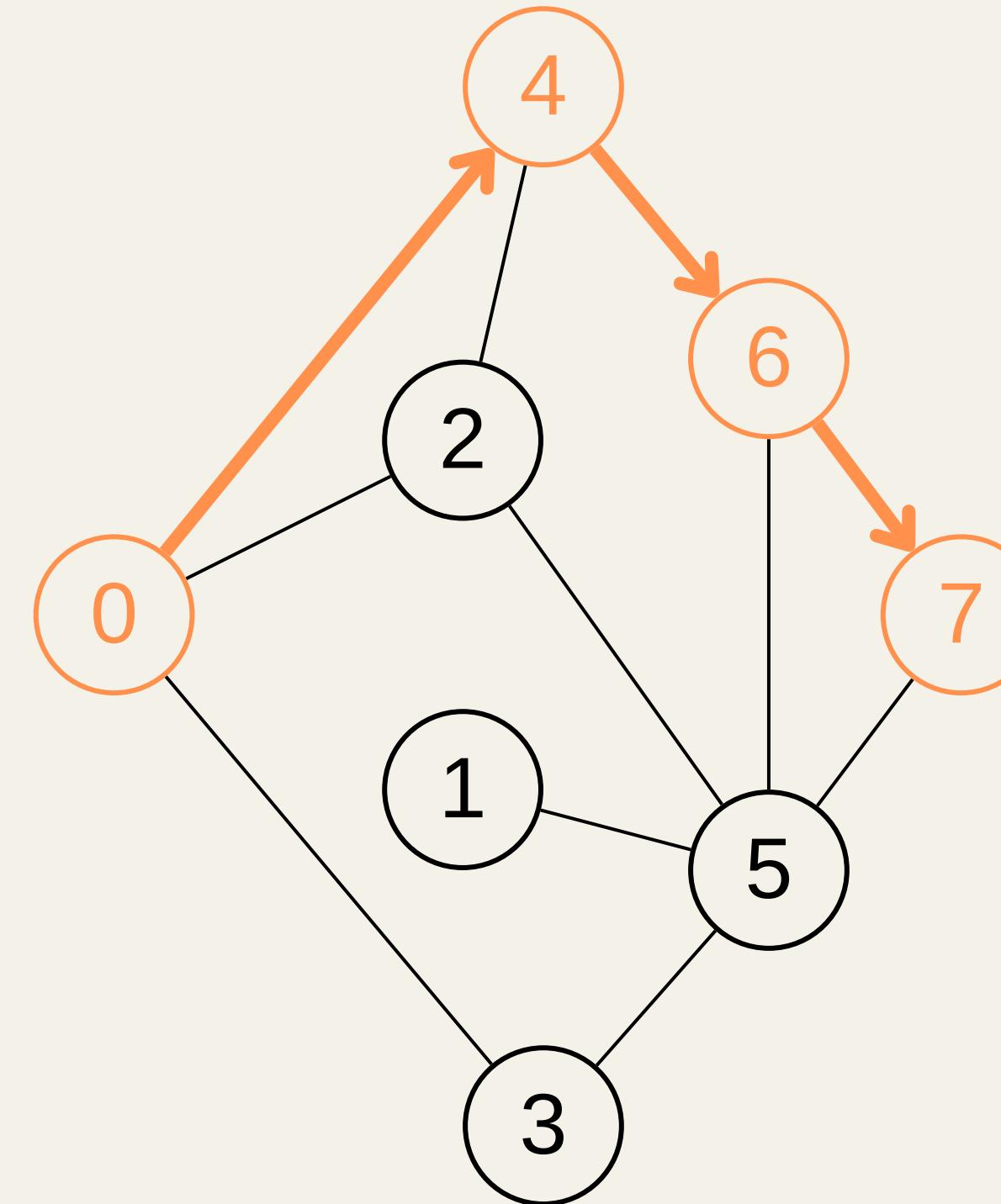
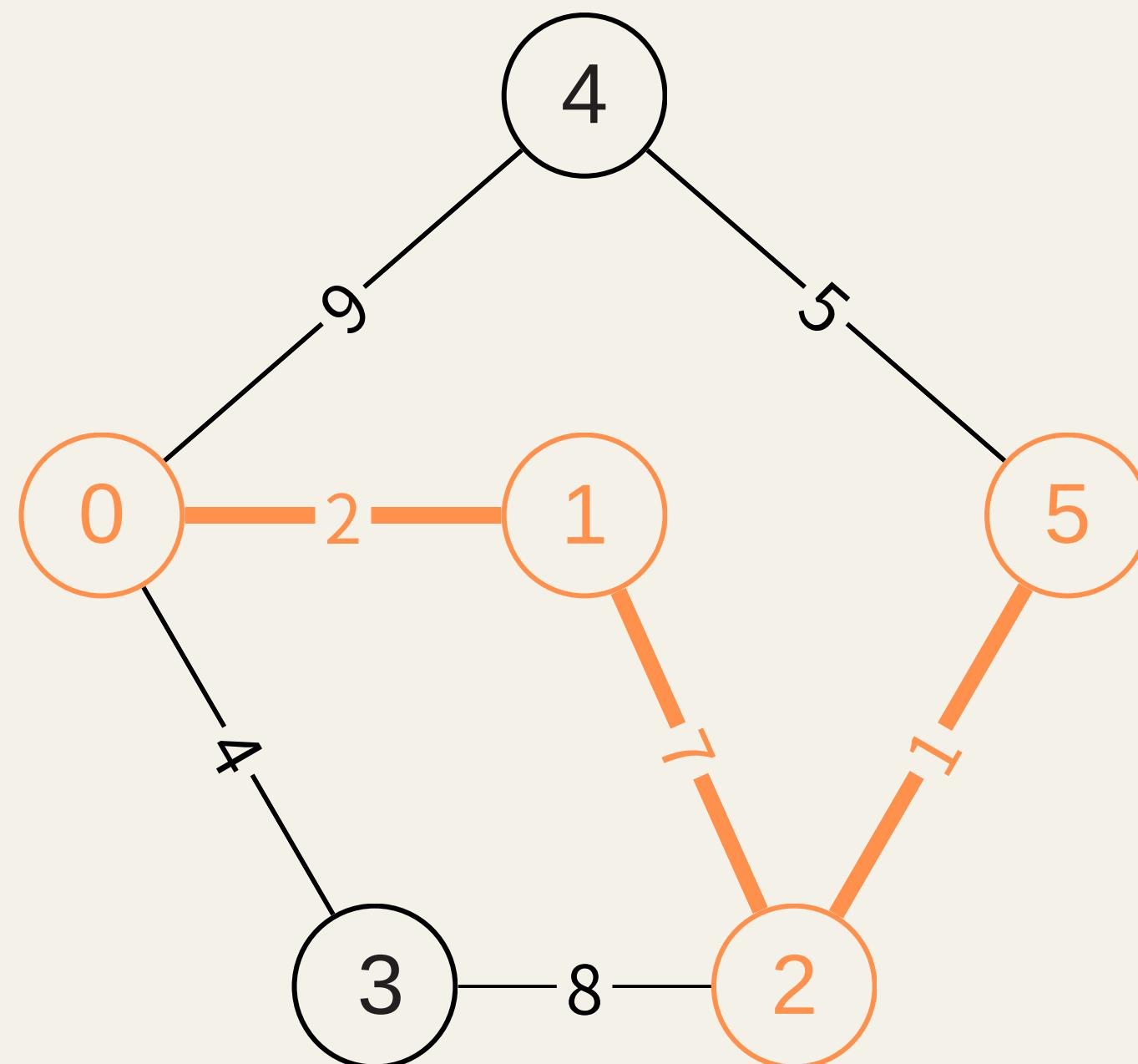
Comparison

Implementation

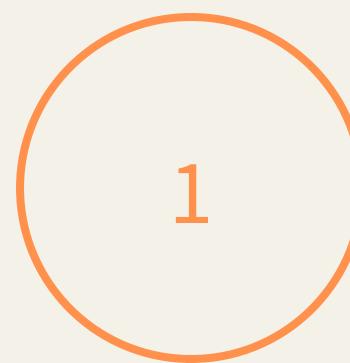
Theory

Implementation

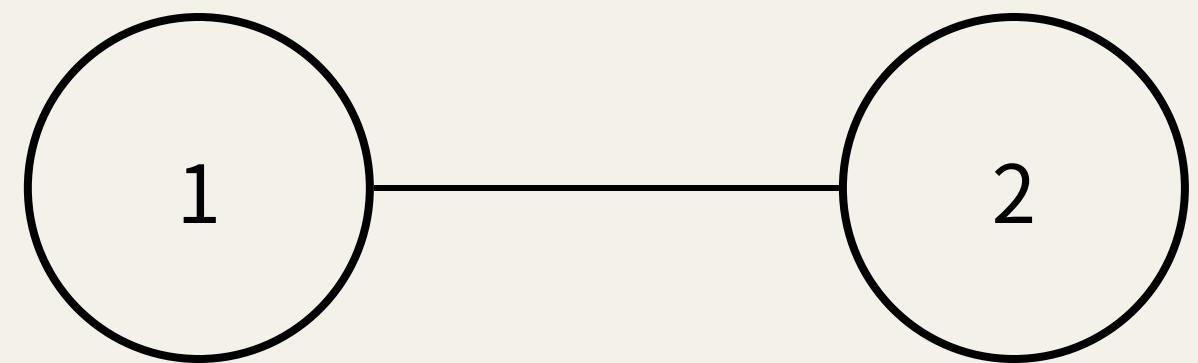
Shortest Path



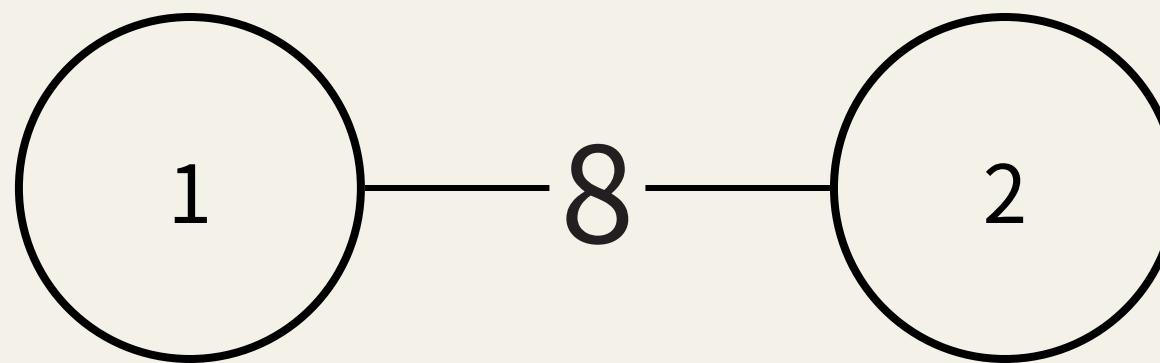
Graph Elements



Vertex

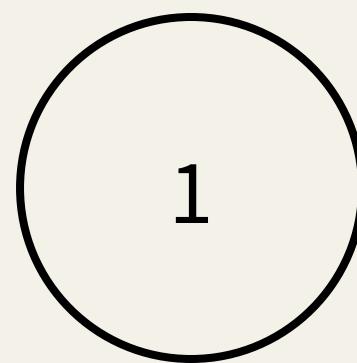


Edge



Weight

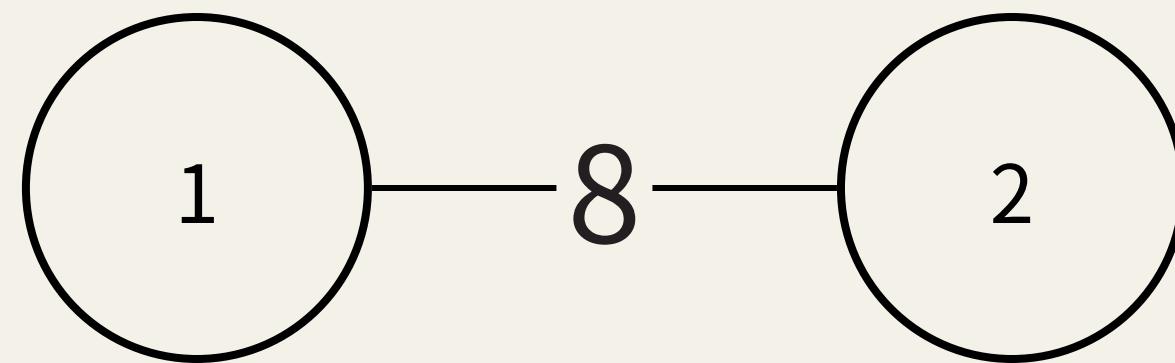
Graph Elements



Vertex

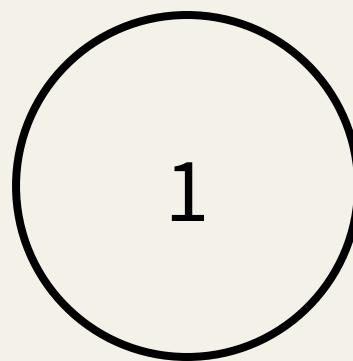


Edge

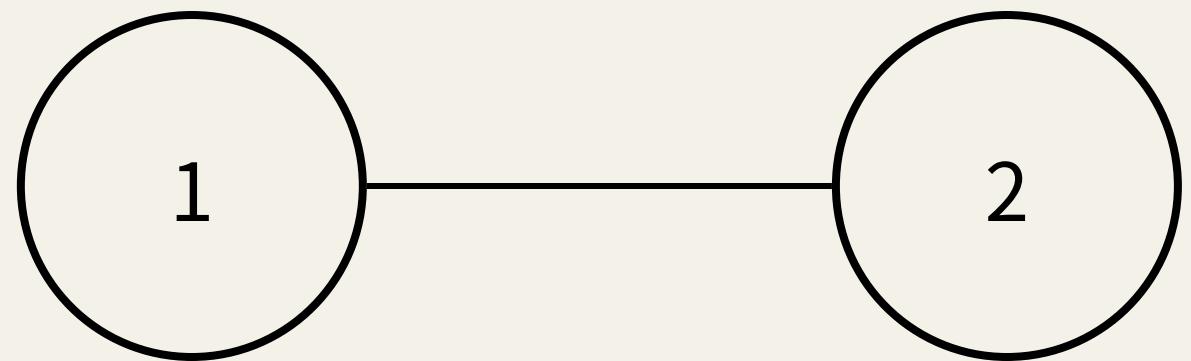


Weight

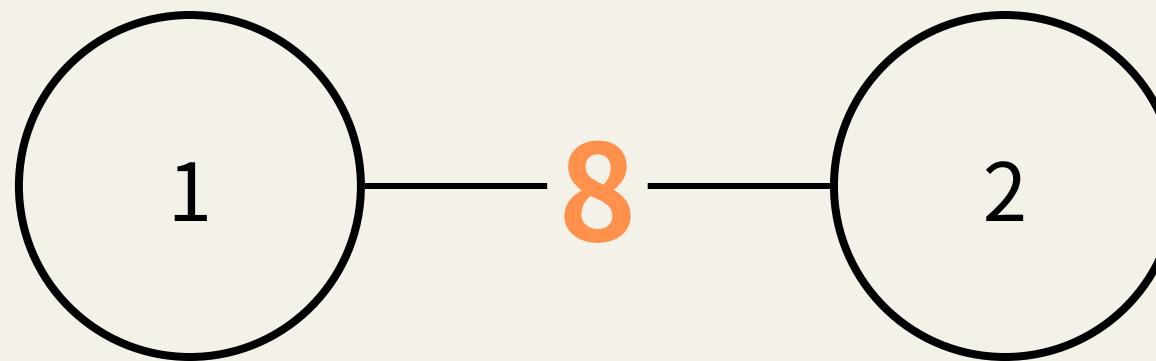
Graph Elements



Vertex



Edge



Weight



Breadth First Search

廣度優先搜尋演算法

Initialize

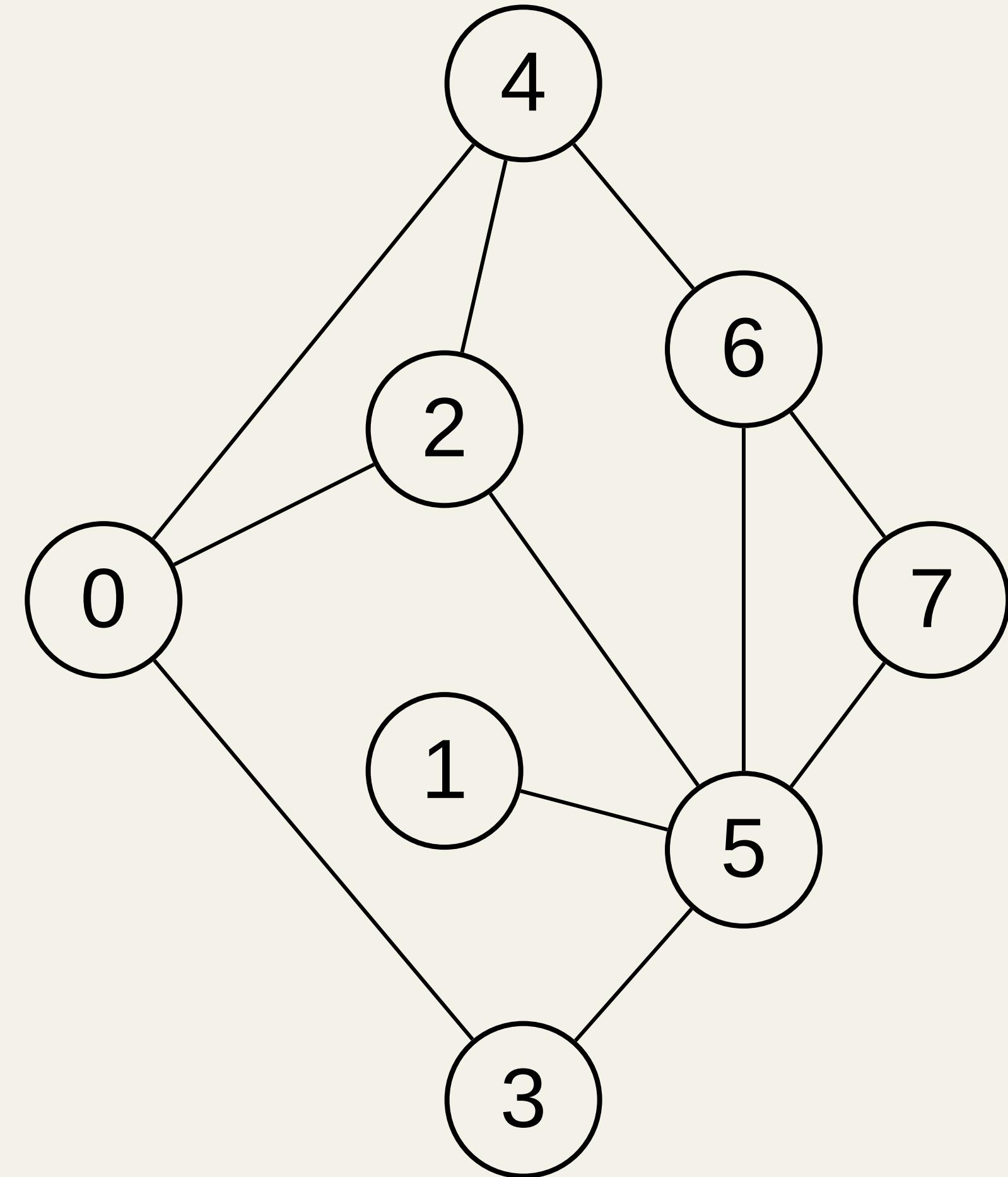
index	0	1	2	3	4	5	6	7
dis[]	-1	-1	-1	-1	-1	-1	-1	-1

- ① Set all of the value in distance array to default value which equivalent to the distance you will never reach → -1

queue<pair<int, int>> q

q.empty() = True

- ② Make sure the queue we are going to use is empty → declare a new queue

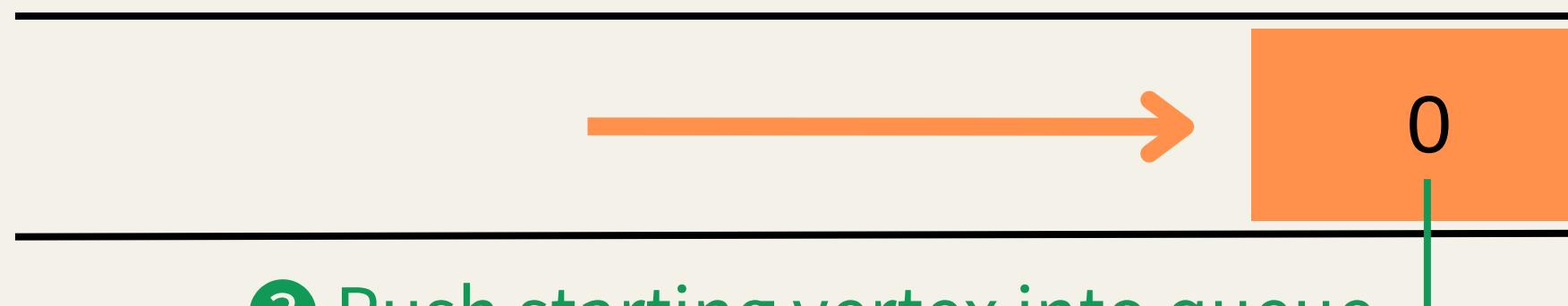


Initial Operation

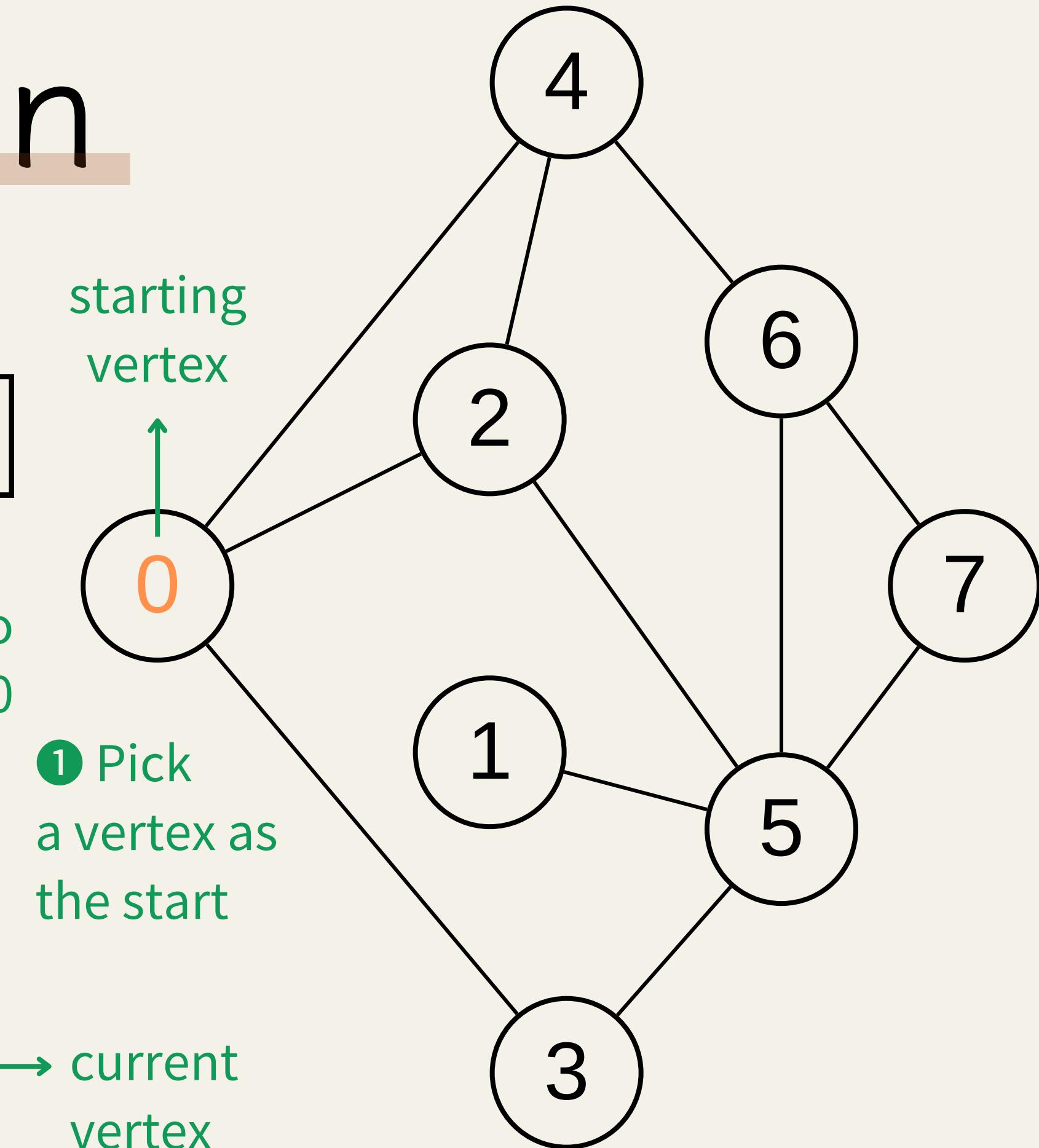
index	0	1	2	3	4	5	6	7
dis[]	0	-1	-1	-1	-1	-1	-1	-1

- ② Set the distance of starting vertex to 0
→ The distance from starting vertex to starting vertex will definitely be 0

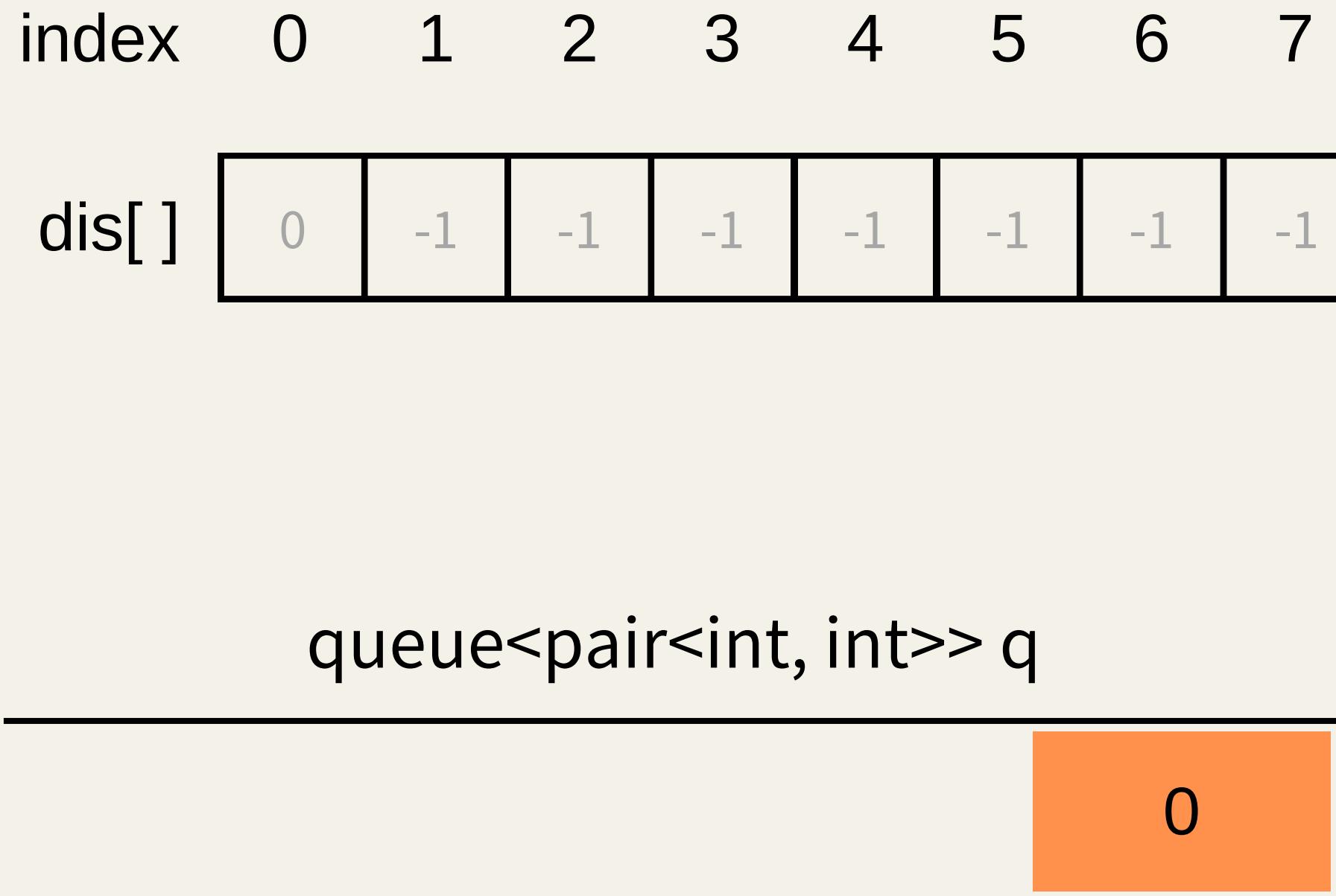
queue<pair<int, int>> q



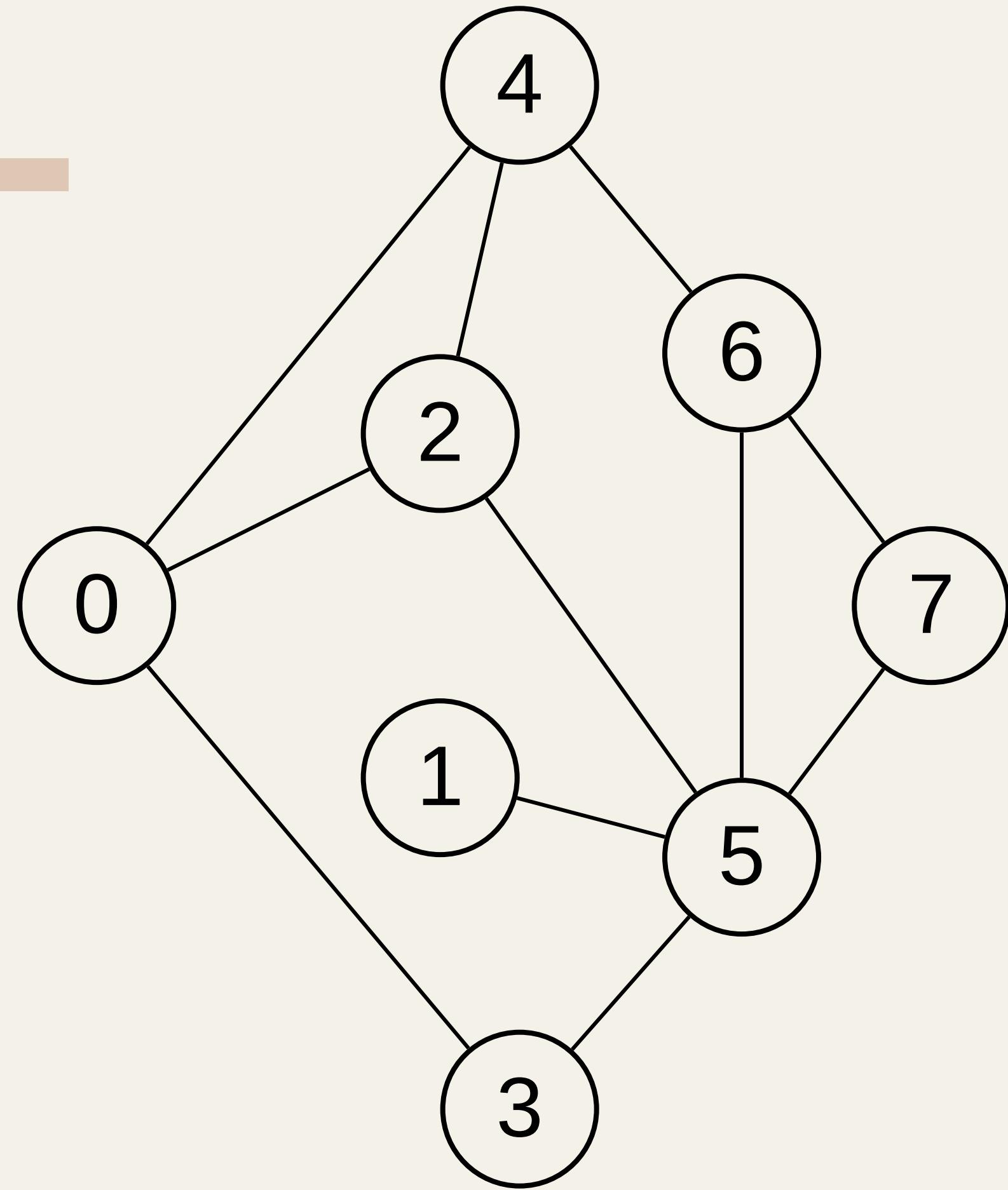
- ③ Push starting vertex into queue
→ This means we are ready to deal with this vertex



Operating BFS



Once the queue is not empty
→ Keep going with the remaining vertices



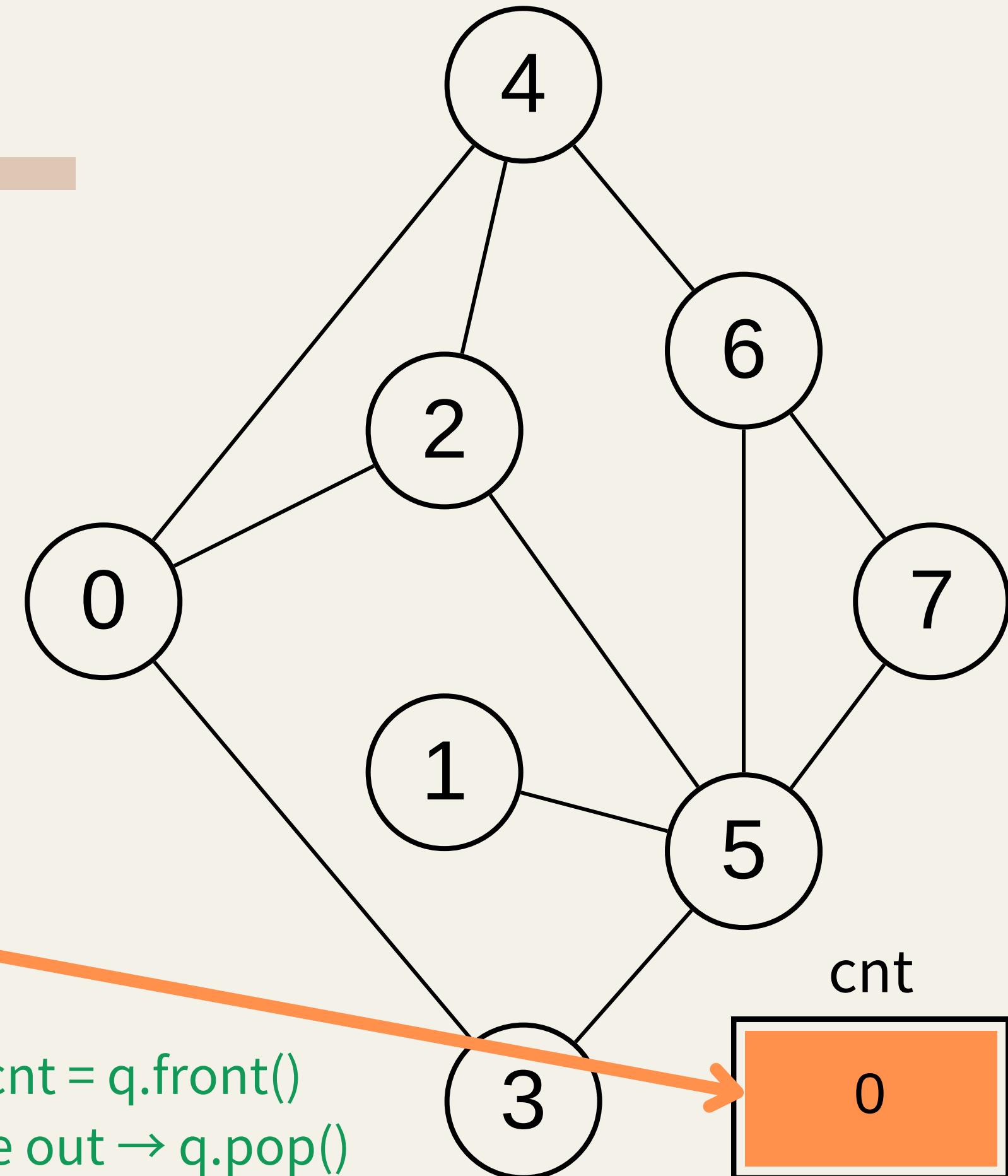
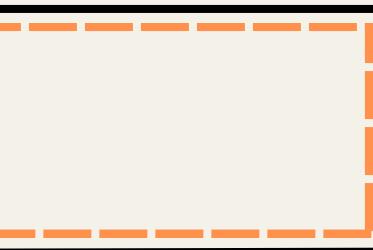
Operating BFS

index 0 1 2 3 4 5 6 7

dis[]

0	-1	-1	-1	-1	-1	-1	-1
---	----	----	----	----	----	----	----

queue<pair<int, int>> q



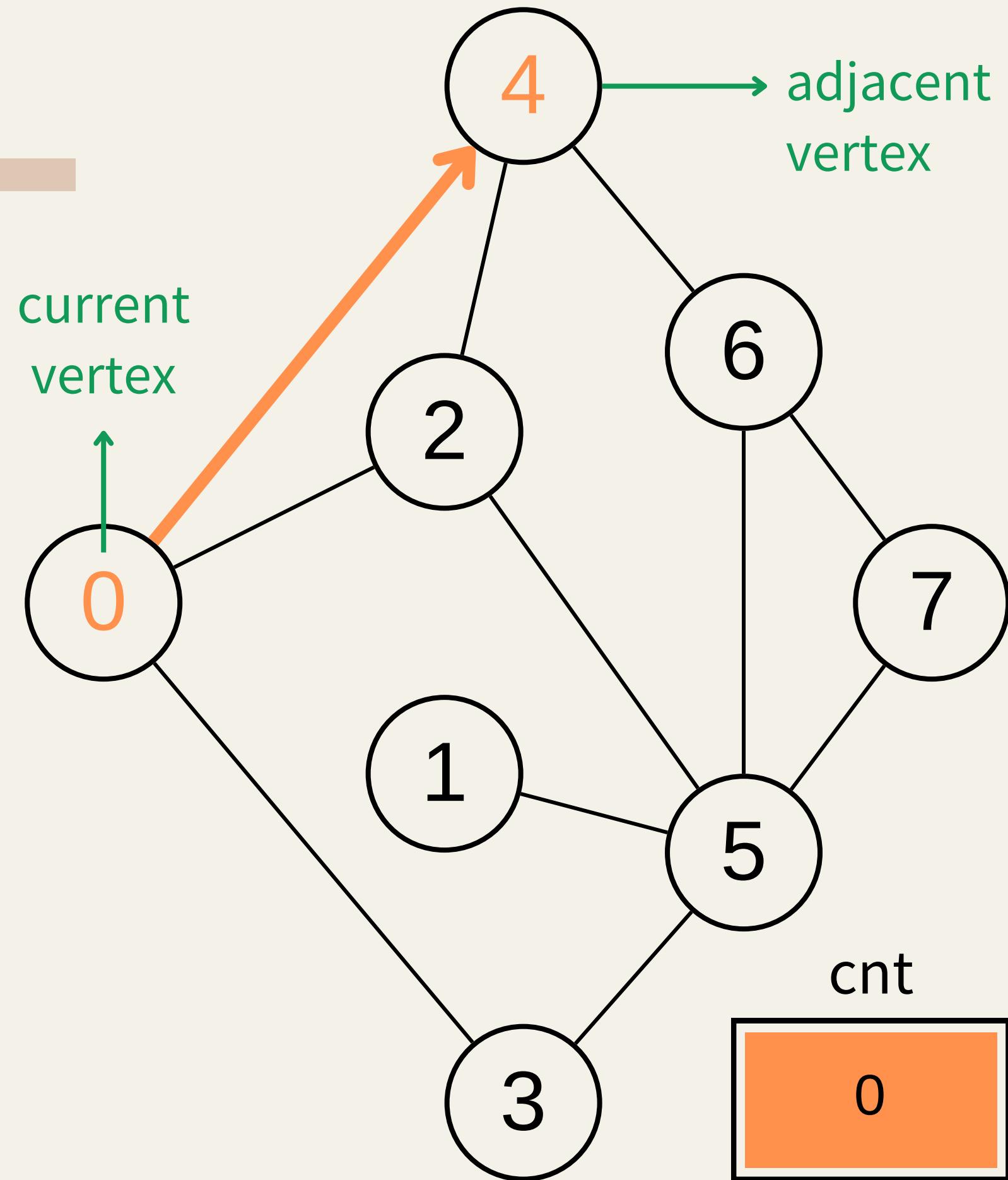
- ① Put the front most element of queue into cnt \rightarrow `cnt = q.front()`
- ② Pop out the old element which already been take out \rightarrow `q.pop()`

Operating BFS

index	0	1	2	3	4	5	6	7
dis[]	0	-1	-1	-1	-1	-1	-1	-1

Check whether adjacent vertex is available or not
→ $\text{dis}[\text{adjacent}]$ equals to default value

```
queue<pair<int, int>> q
```

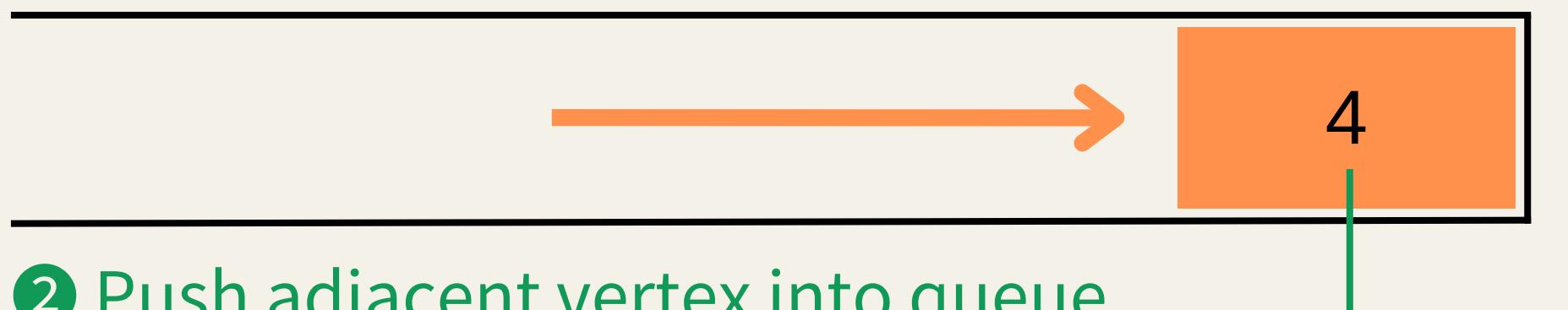


Operating BFS

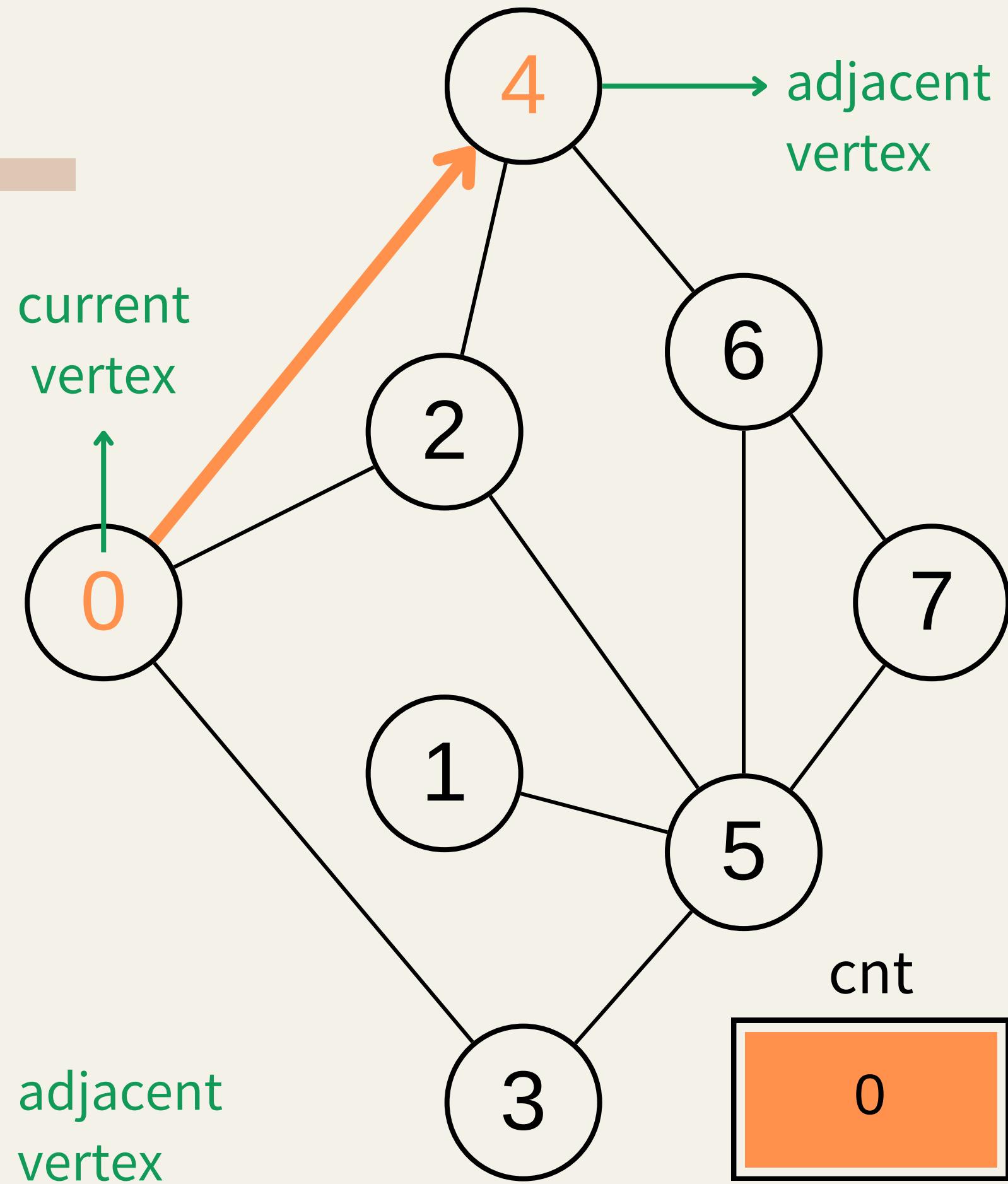
index	0	1	2	3	4	5	6	7
dis[]	0	-1	-1	-1	1	-1	-1	-1

- ① If adjacent vertex is available
→ Update $\text{dis}[\text{adjacent}]$ to $\text{dis}[\text{cnt}] + 1$

queue<pair<int, int>> q



- ② Push adjacent vertex into queue

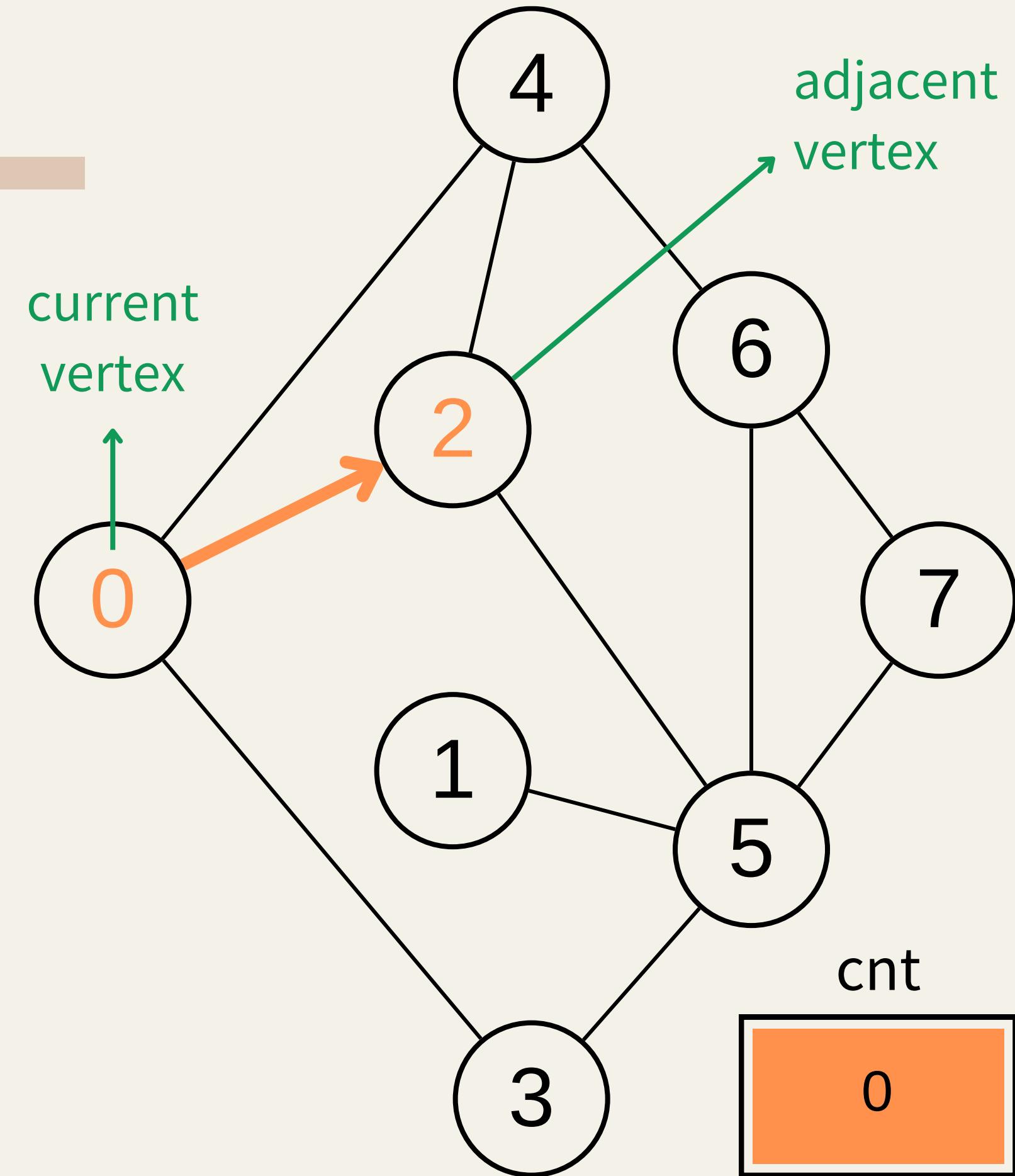


Operating BFS

index	0	1	2	3	4	5	6	7
dis[]	0	-1	-1	-1	1	-1	-1	-1

Check whether adjacent vertex is available or not
→ $\text{dis}[\text{adjacent}]$ equals to default value

```
queue<pair<int, int>> q
```

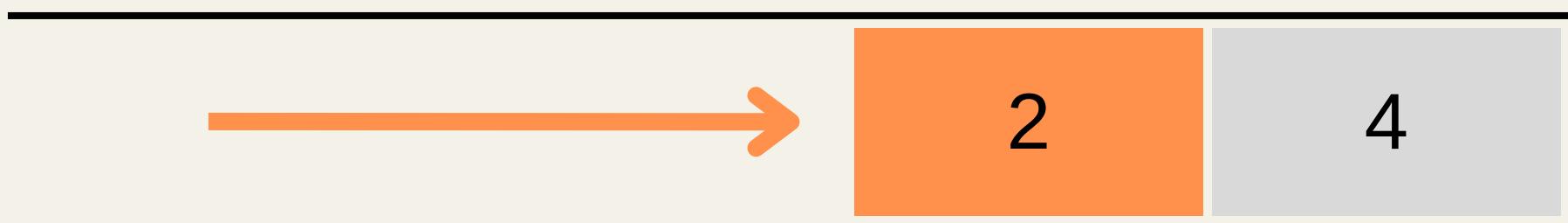


Operating BFS

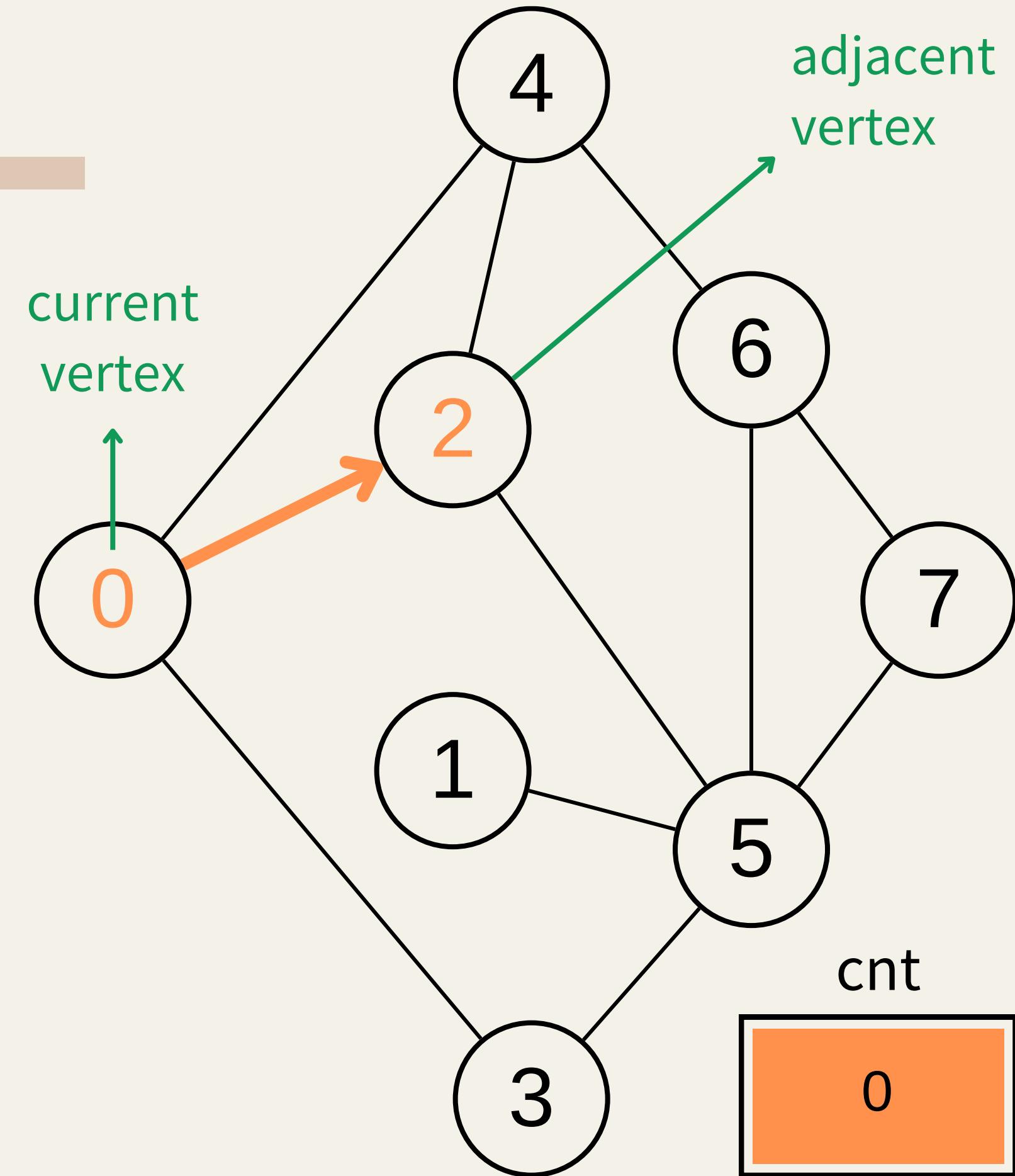
index	0	1	2	3	4	5	6	7
dis[]	0	-1	1	-1	1	-1	-1	-1

- ① If adjacent vertex is available
→ Update $\text{dis}[\text{adjacent}]$ to $\text{dis}[\text{cnt}] + 1$

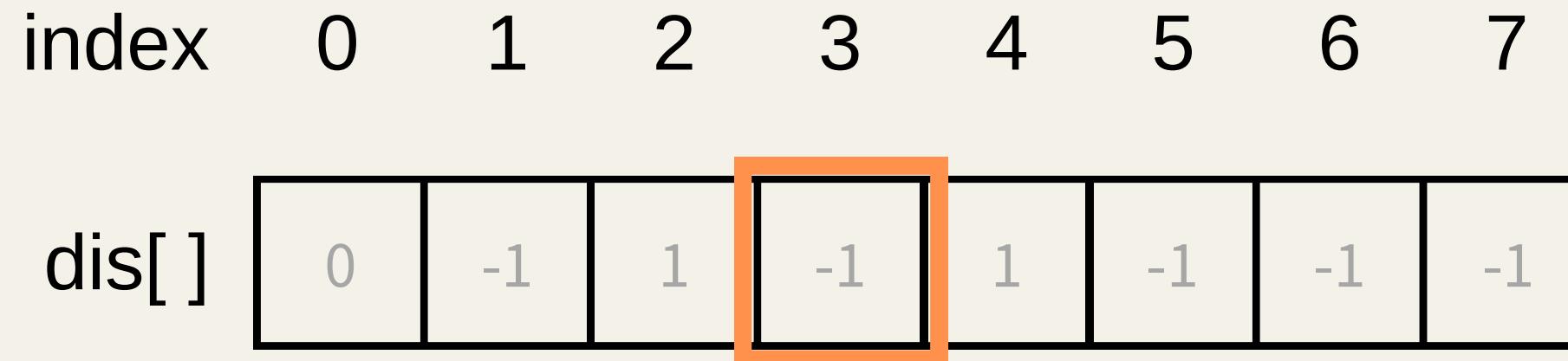
queue<pair<int, int>> q



- ② Push adjacent vertex into queue



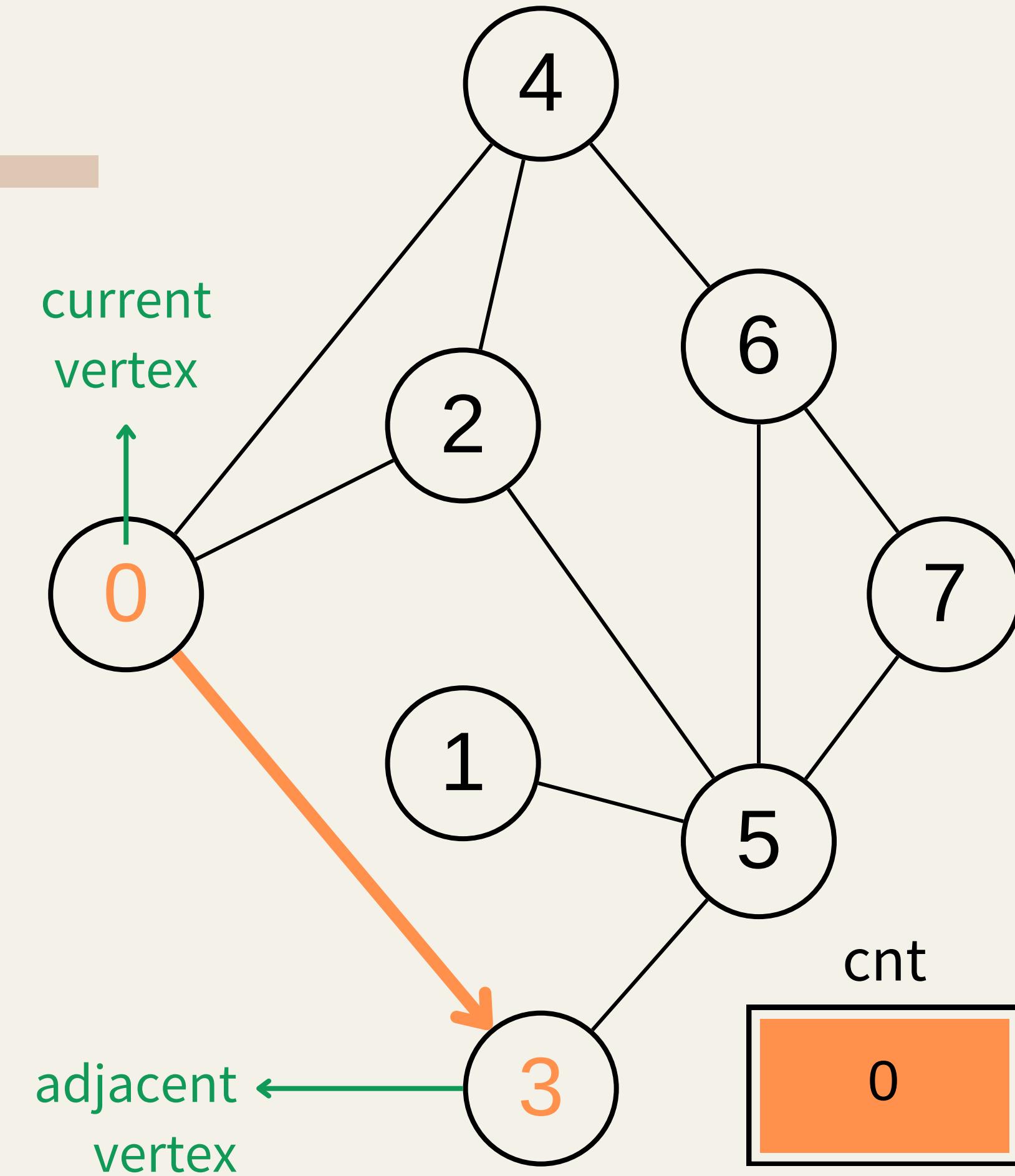
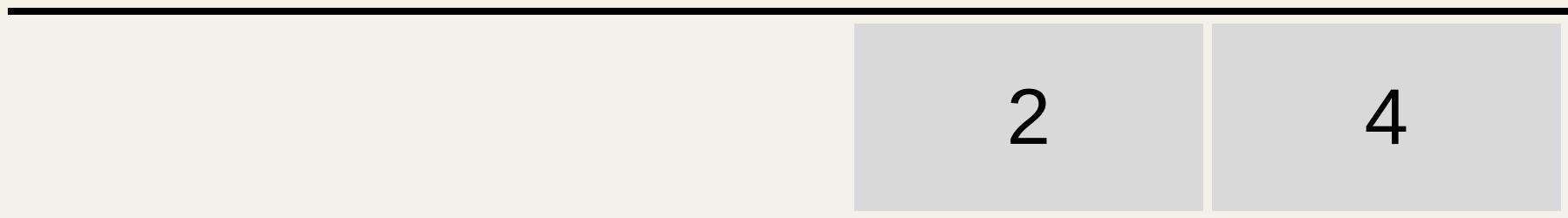
Operating BFS



Available ✓

Check whether adjacent vertex is available or not
→ dis[adjacent] equals to default value

queue<pair<int, int>> q

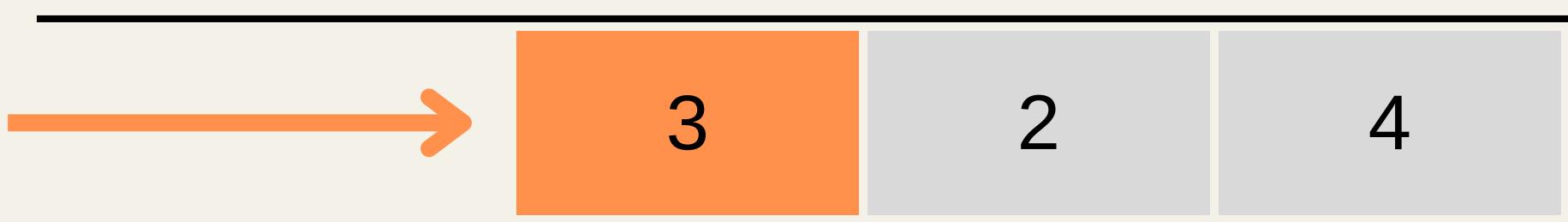


Operating BFS

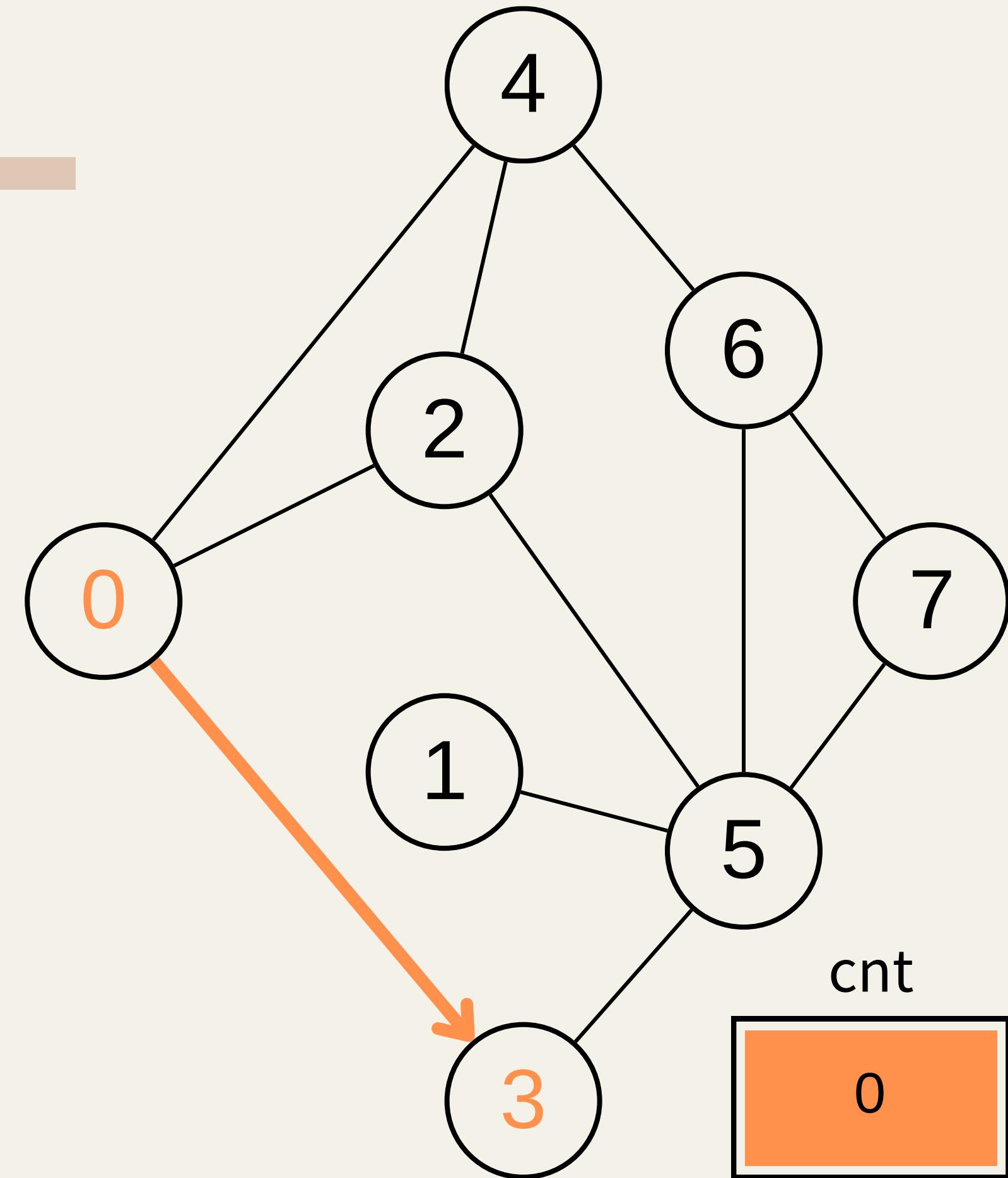
index	0	1	2	3	4	5	6	7
dis[]	0	-1	1	1	1	-1	-1	-1

- ① If adjacent vertex is available
→ Update $\text{dis}[\text{adjacent}]$ to $\text{dis}[\text{cnt}] + 1$

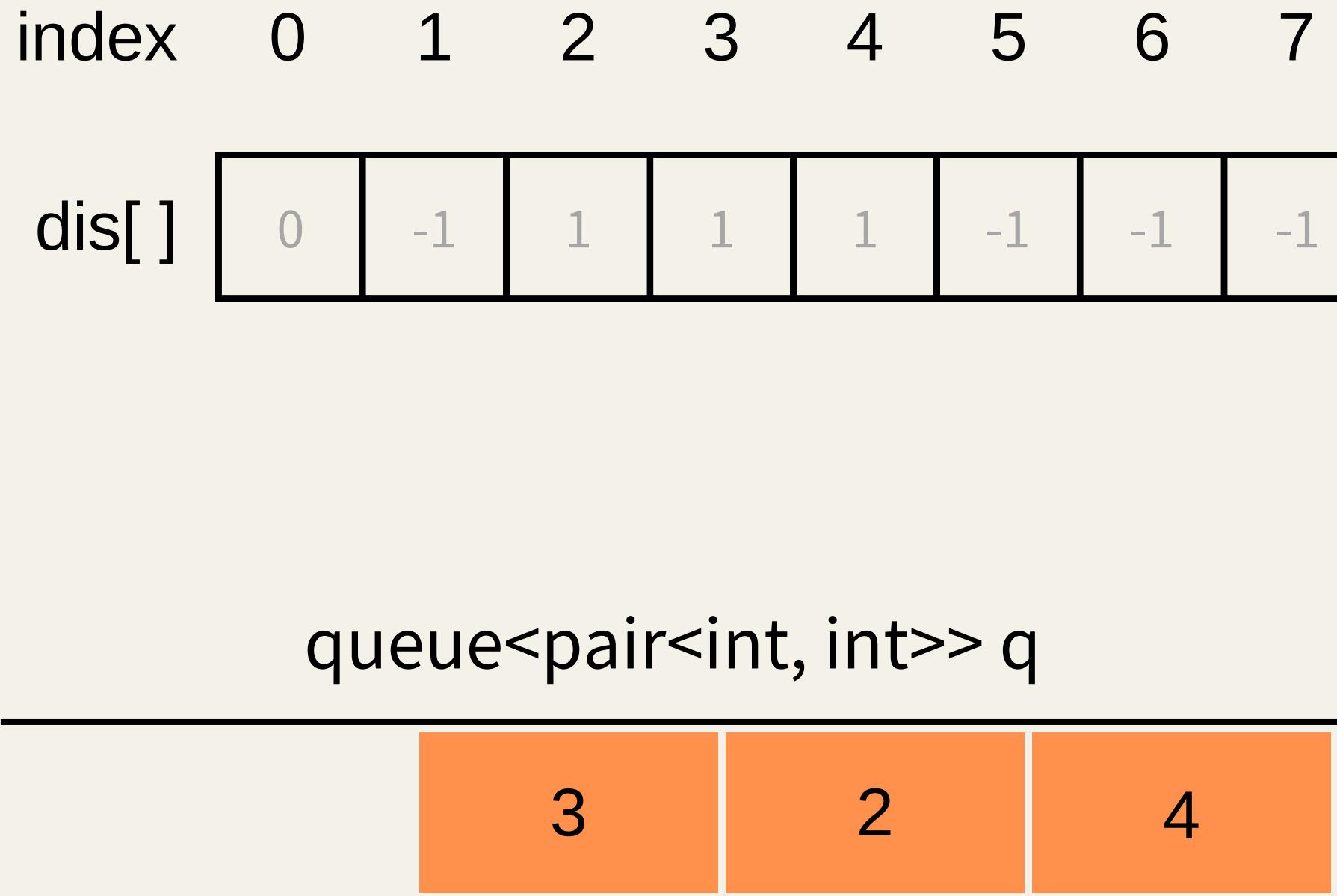
queue<pair<int, int>> q



- ② Push adjacent vertex into queue



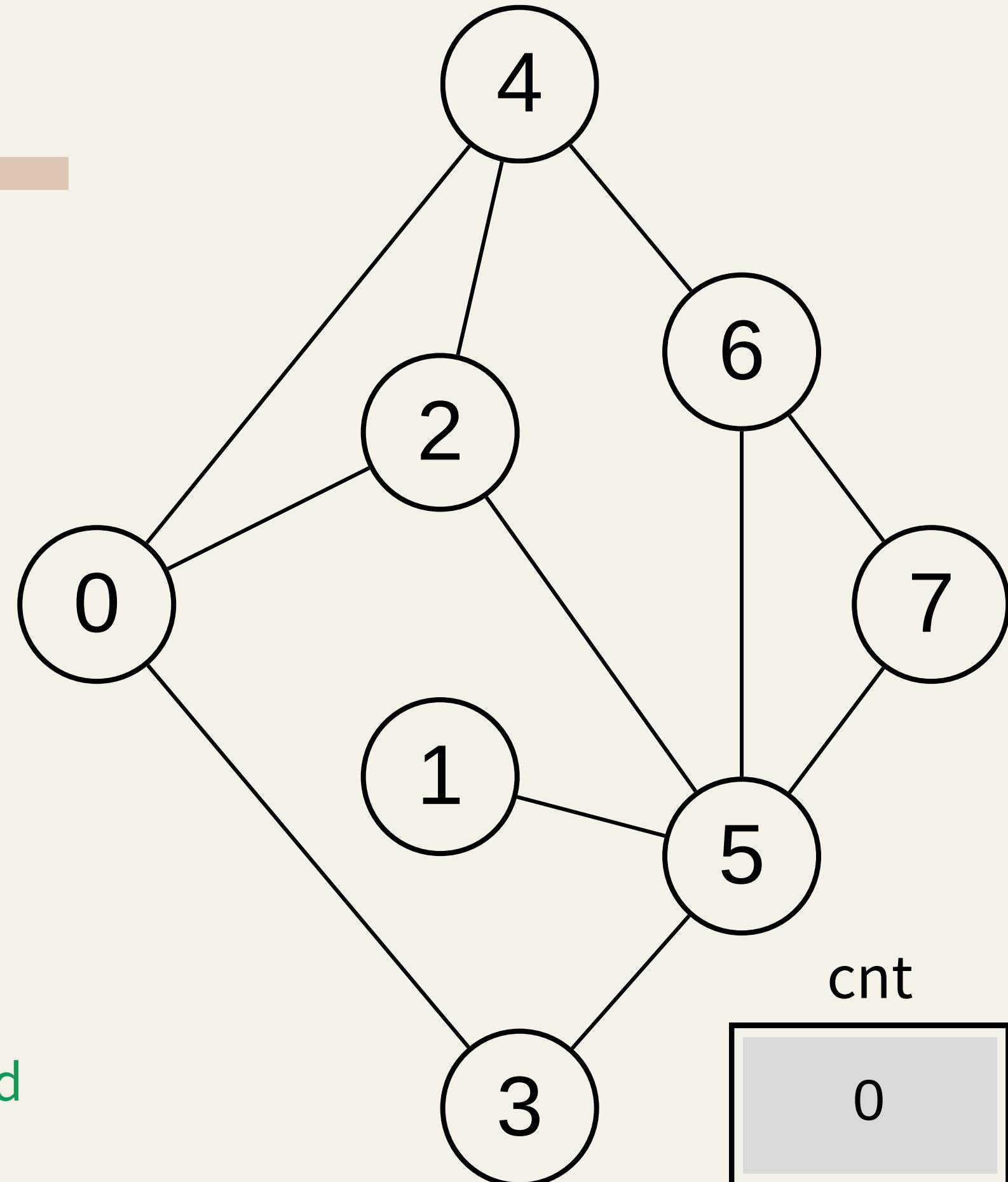
Operating BFS



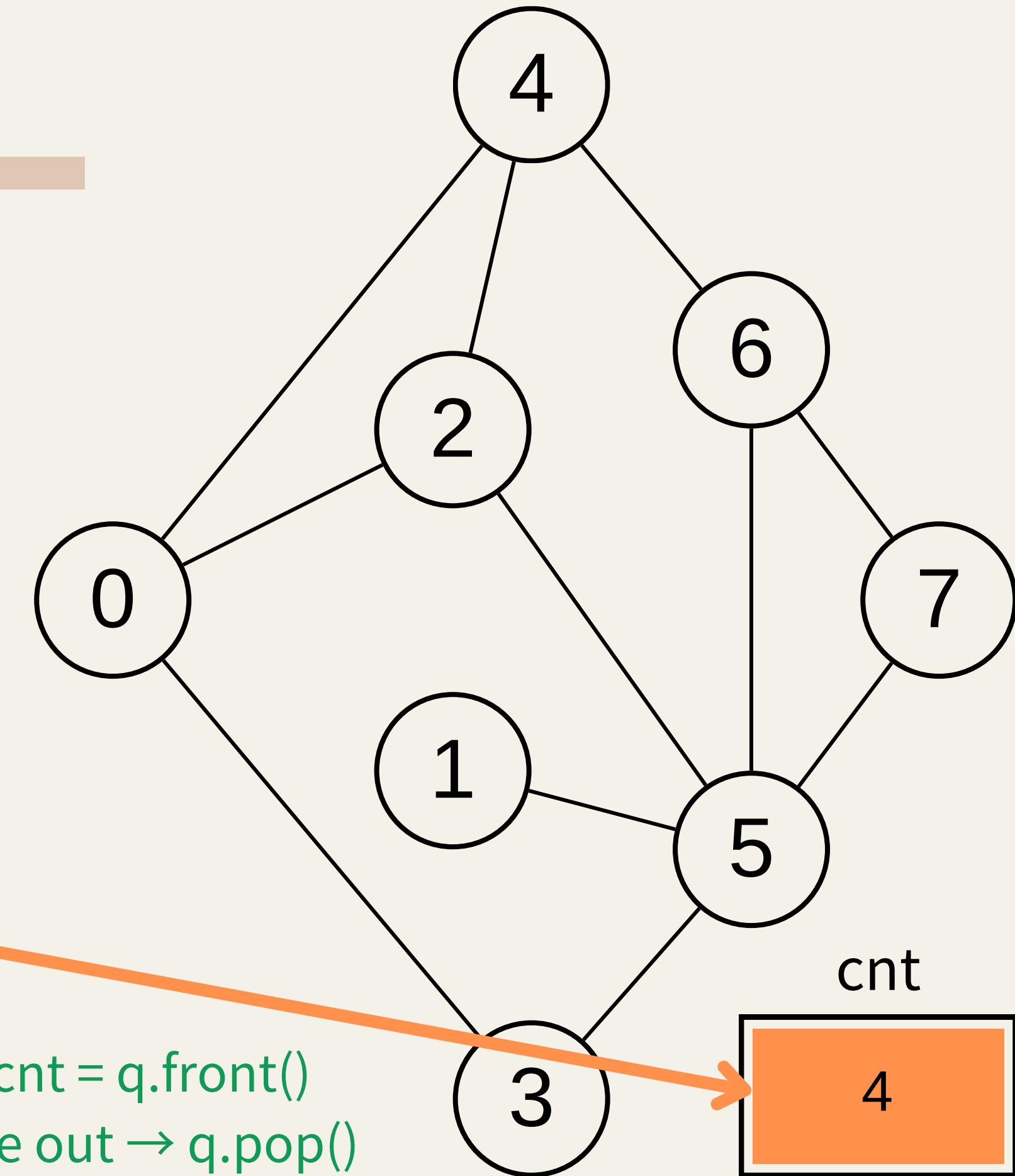
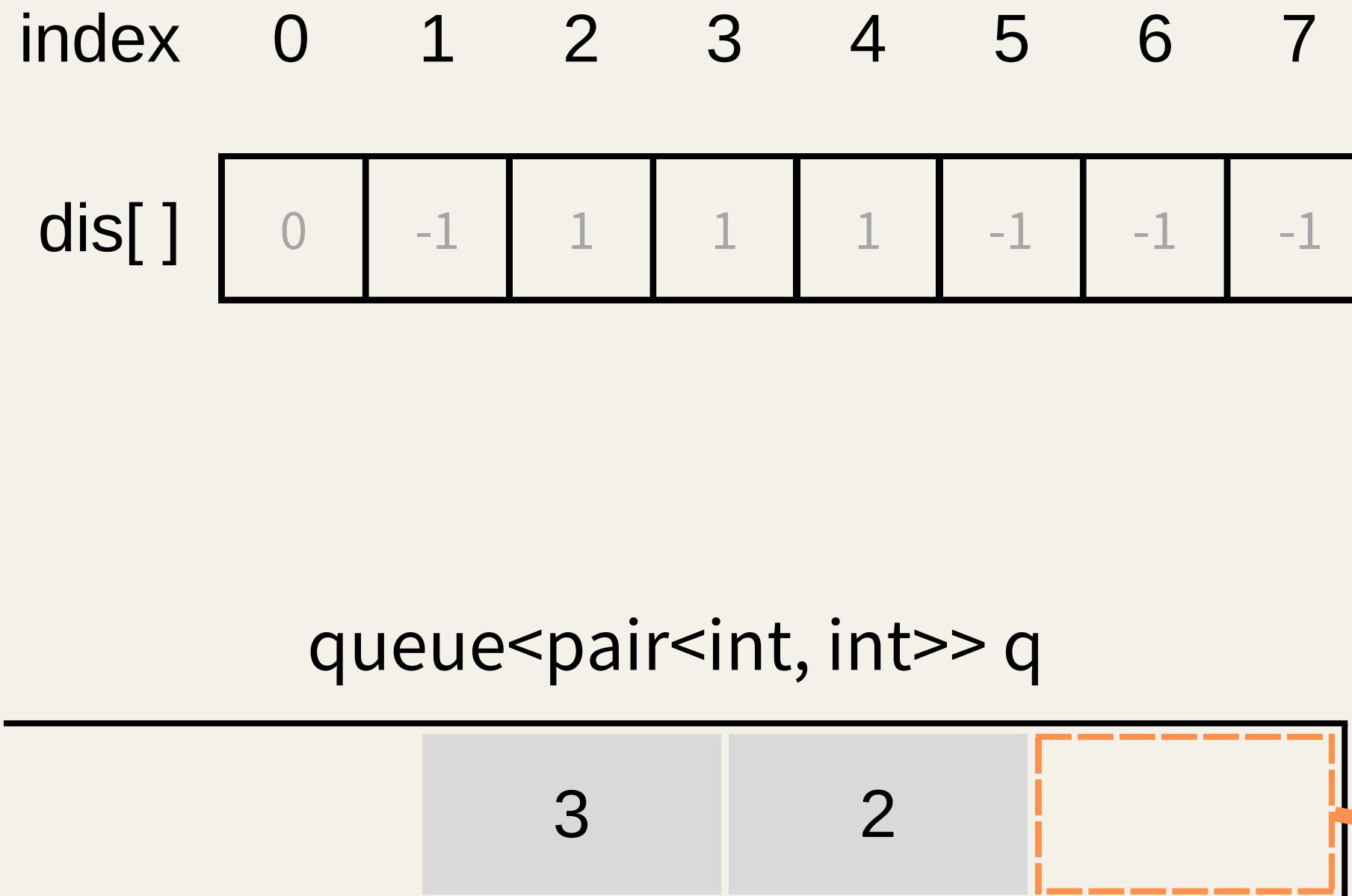
When all vertex adjacent to cnt have been processed

Then, Check the whether queue is empty or not

→ If true, keep going with the remaining vertices



Operating BFS



- ① Put the front most element of queue into cnt \rightarrow `cnt = q.front()`
- ② Pop out the old element which already been take out \rightarrow `q.pop()`

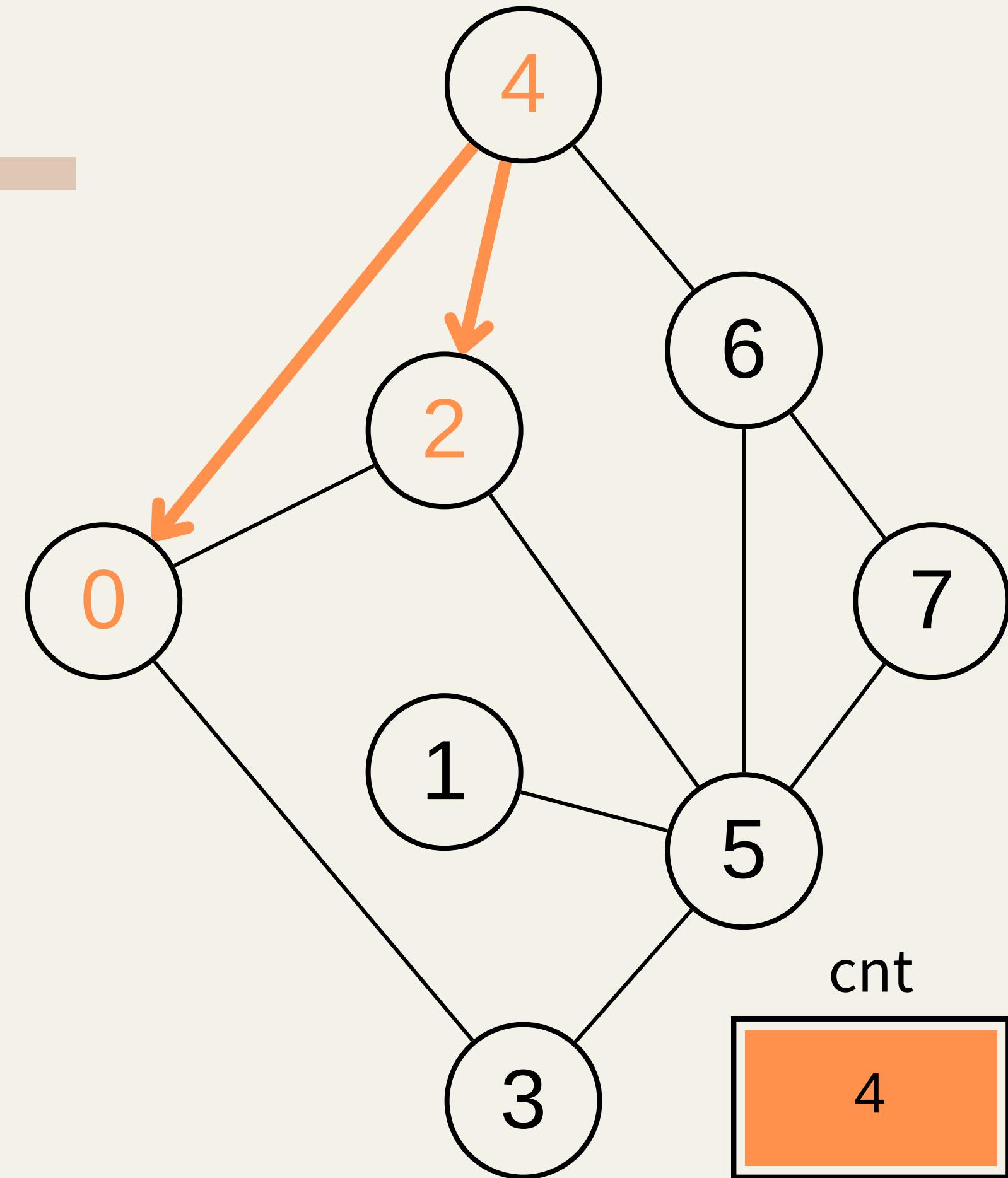
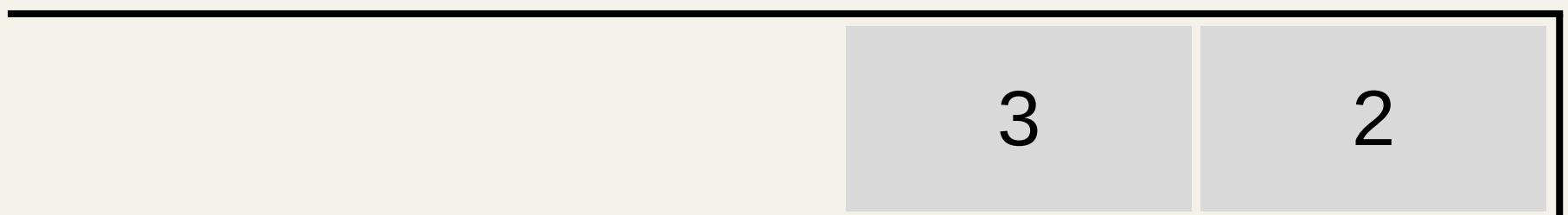
Operating BFS

index	0	1	2	3	4	5	6	7
dis[]	0	-1	1	1	1	-1	-1	-1

Not Available \times

Check whether adjacent vertex is available or not
→ $\text{dis}[\text{adjacent}]$ equals to default value

`queue<pair<int, int>> q`



Operating BFS

index 0 1 2 3 4 5 6 7

dis[]

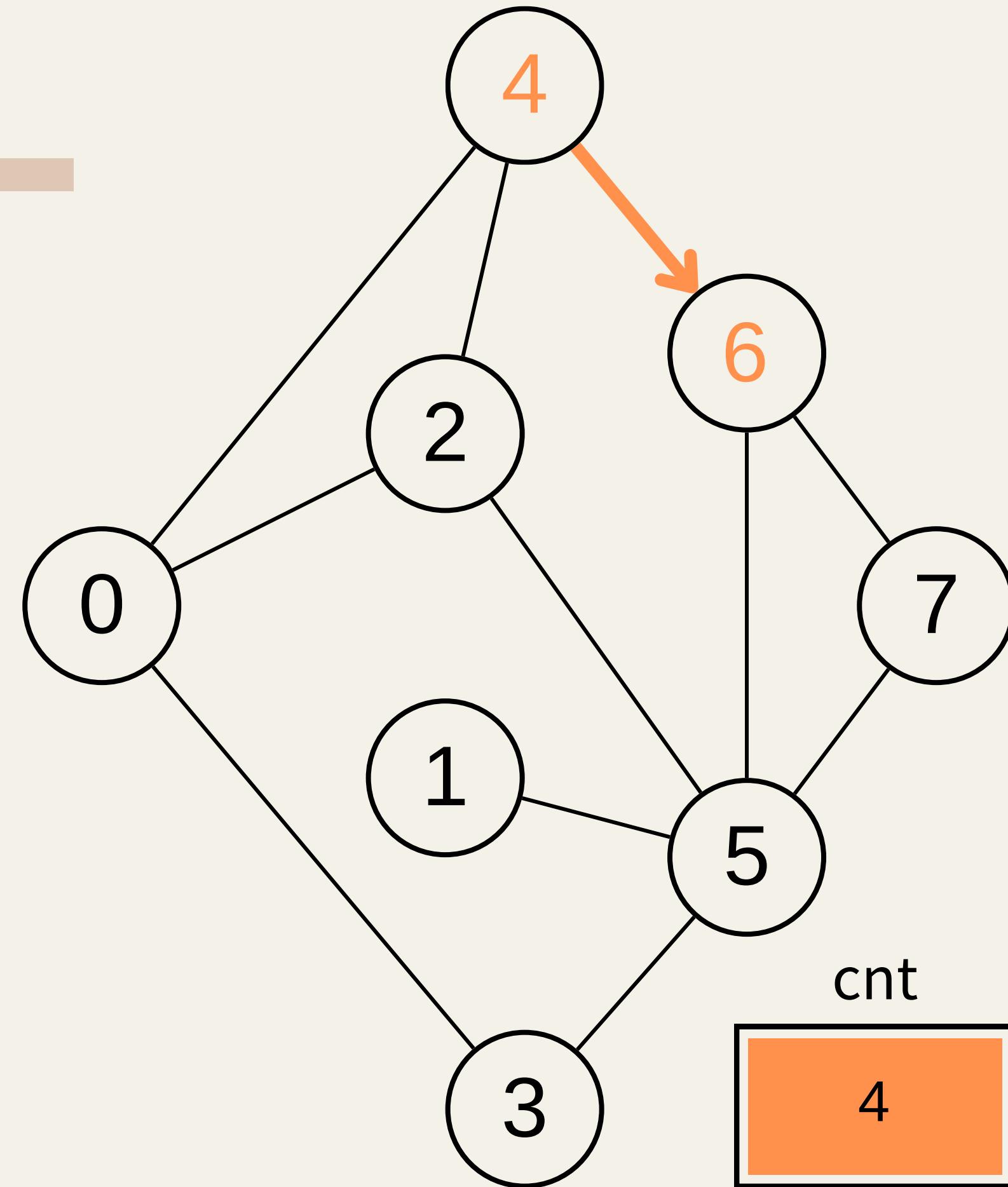
0	-1	1	1	1	-1	-1	-1
---	----	---	---	---	----	----	----

Available ✓

Check whether adjacent vertex is available or not
→ dis[adjacent] equals to default value

queue<pair<int, int>> q

3	2
---	---

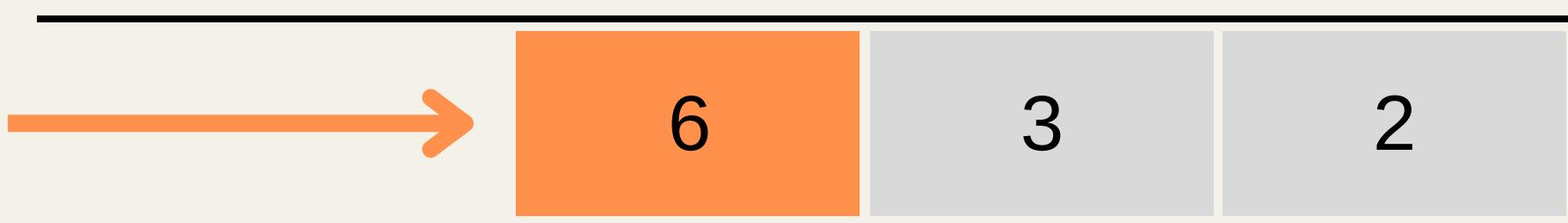


Operating BFS

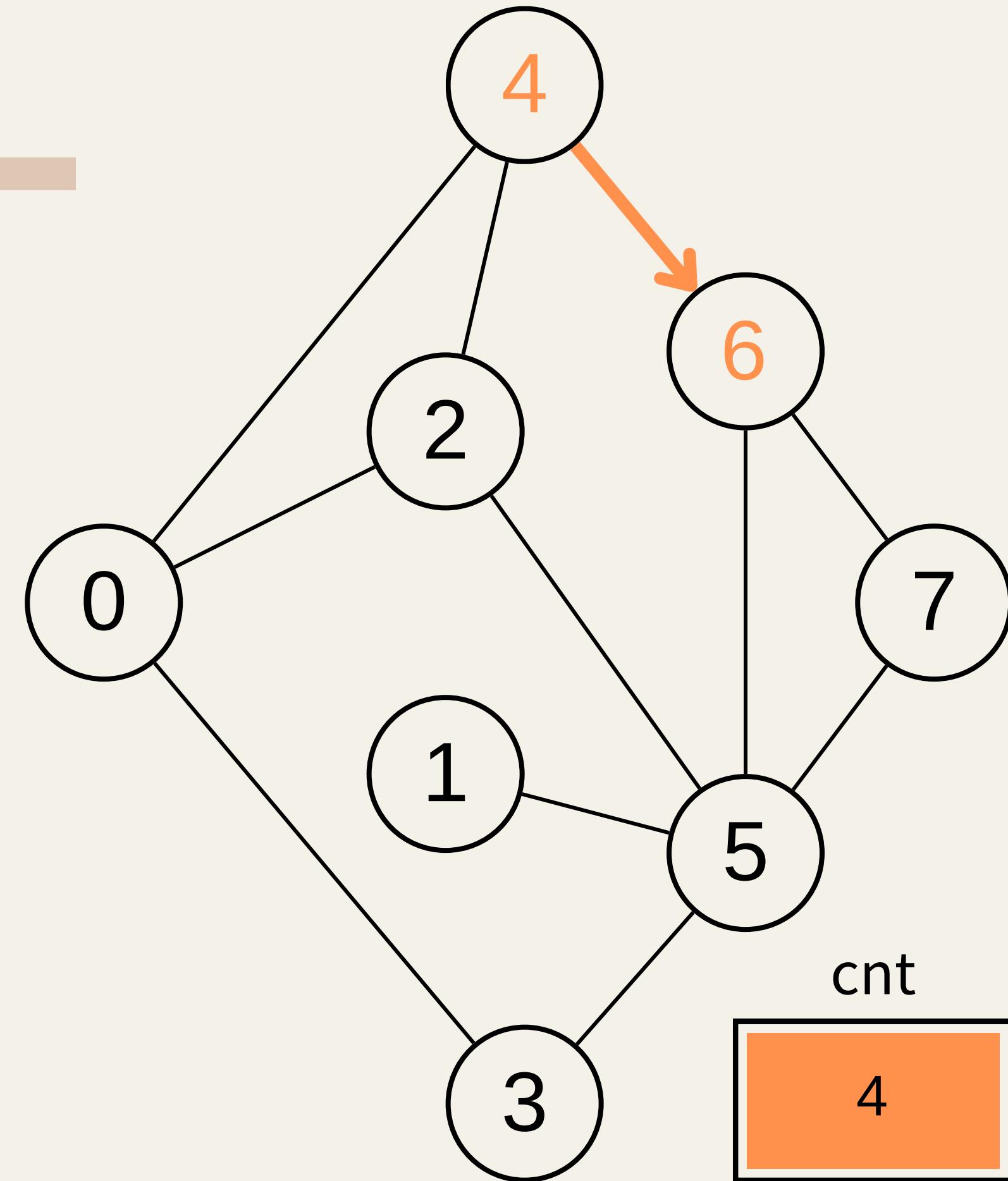
index	0	1	2	3	4	5	6	7
dis[]	0	-1	1	1	1	-1	2	-1

- ① If adjacent vertex is available
→ Update $\text{dis}[\text{adjacent}]$ to $\text{dis}[\text{cnt}] + 1$

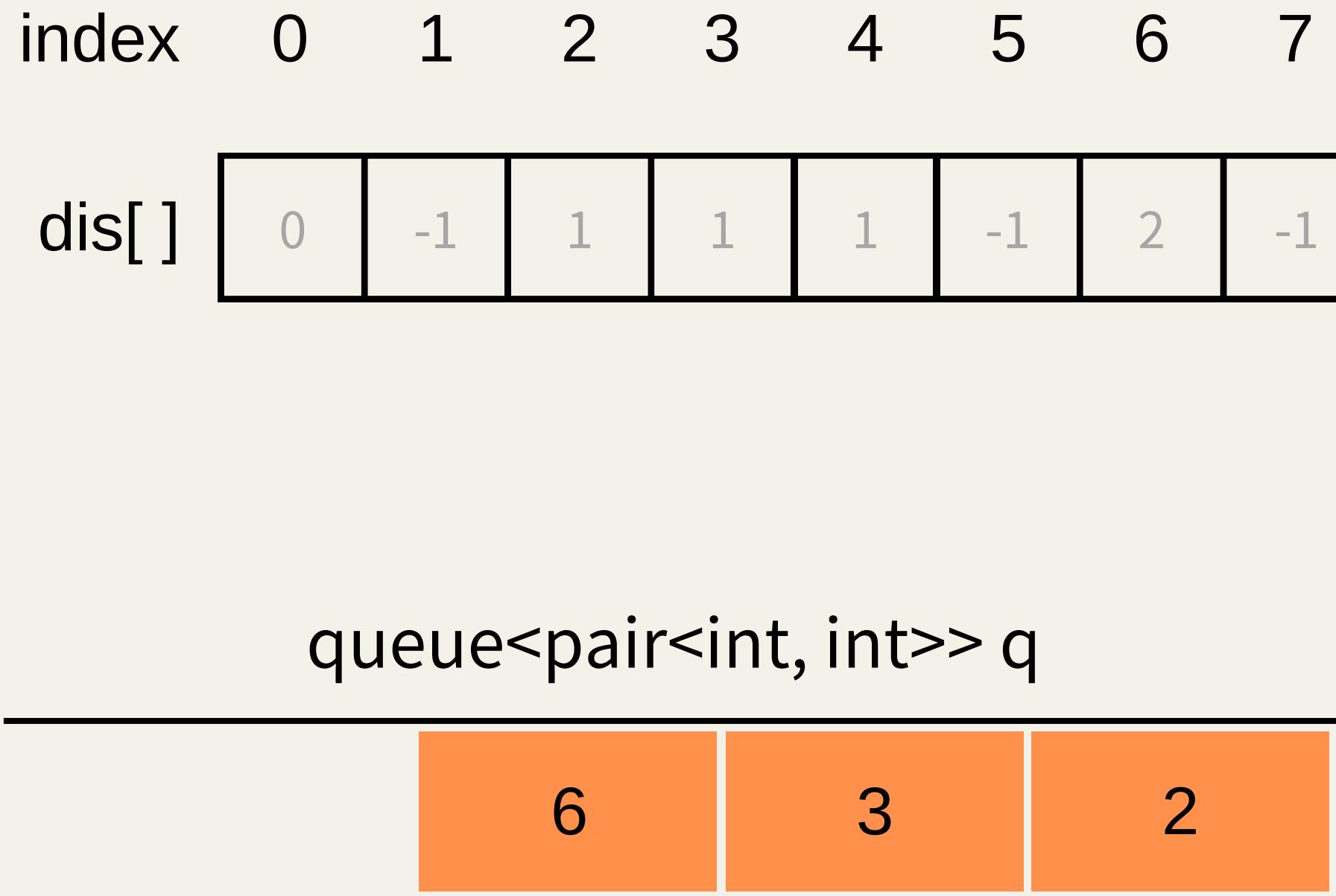
queue<pair<int, int>> q



- ② Push adjacent vertex into queue



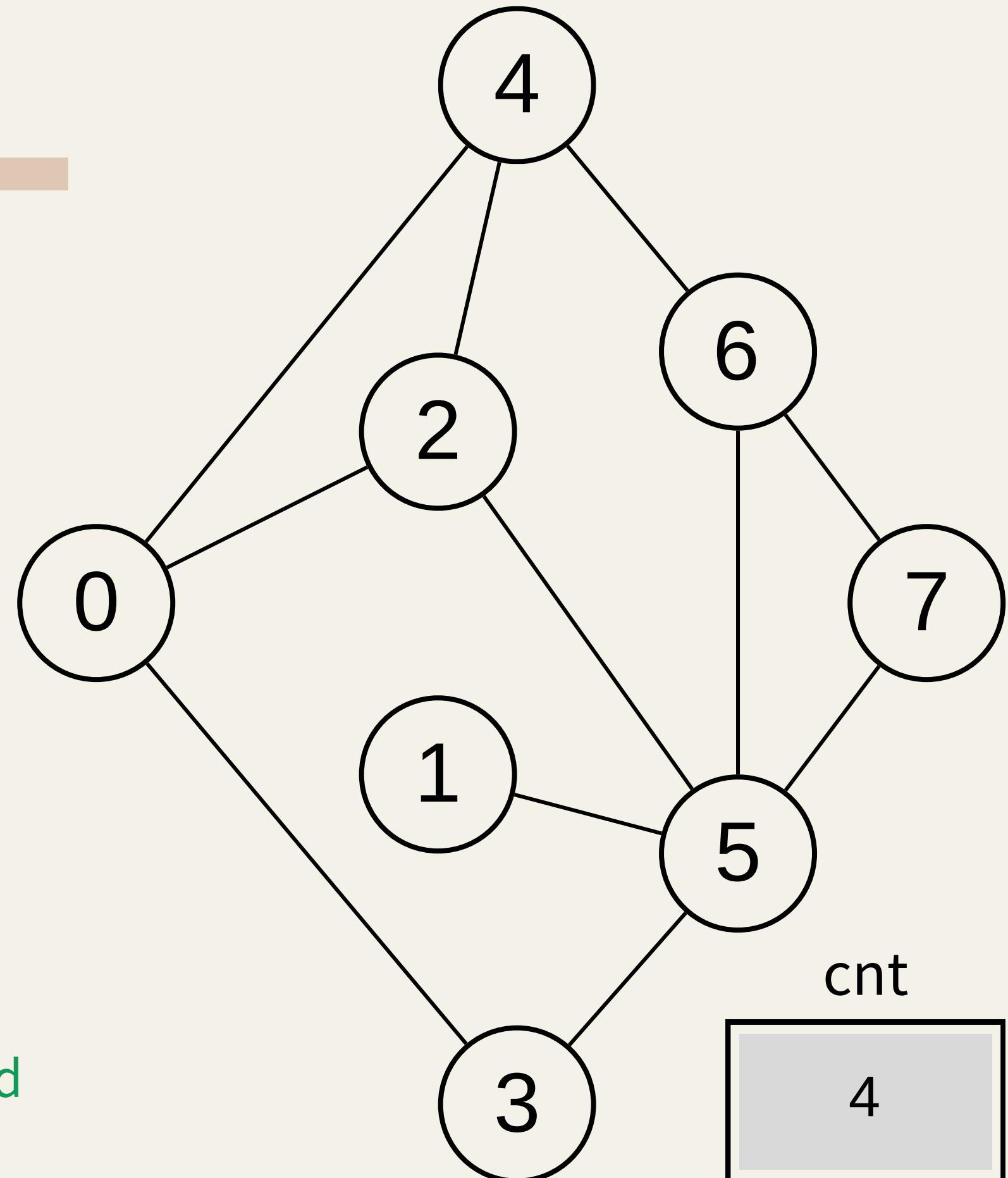
Operating BFS



When all vertex adjacent to cnt have been processed

Then, Check the whether queue is empty or not

→ If true, keep going with the remaining vertices

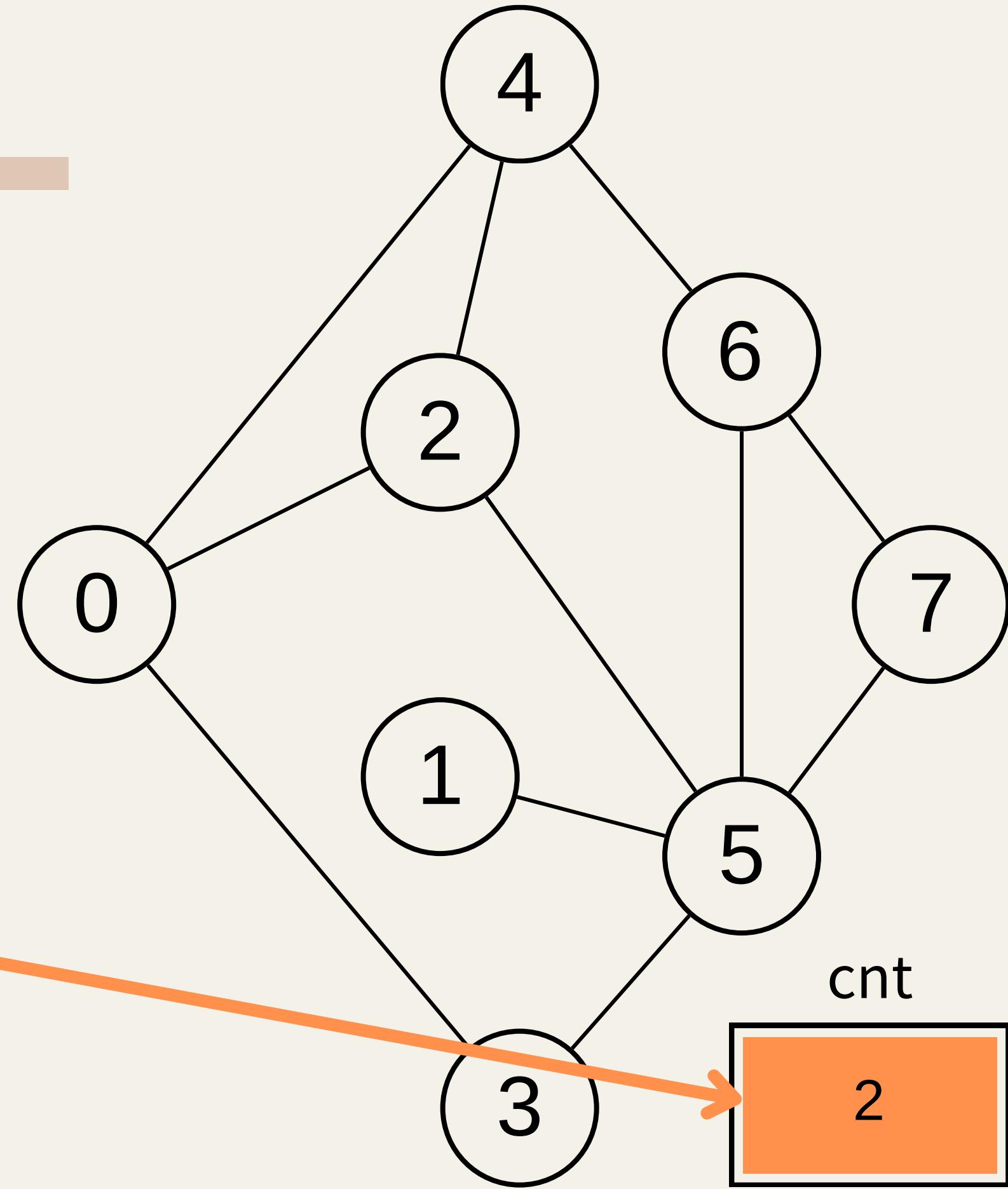
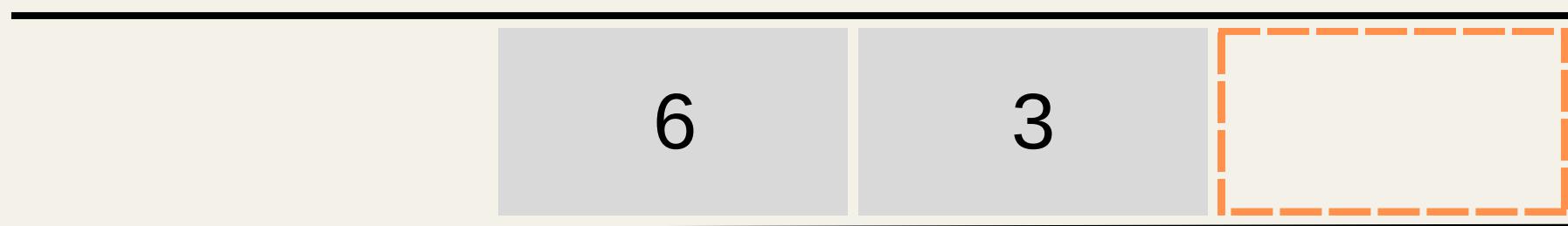


Operating BFS

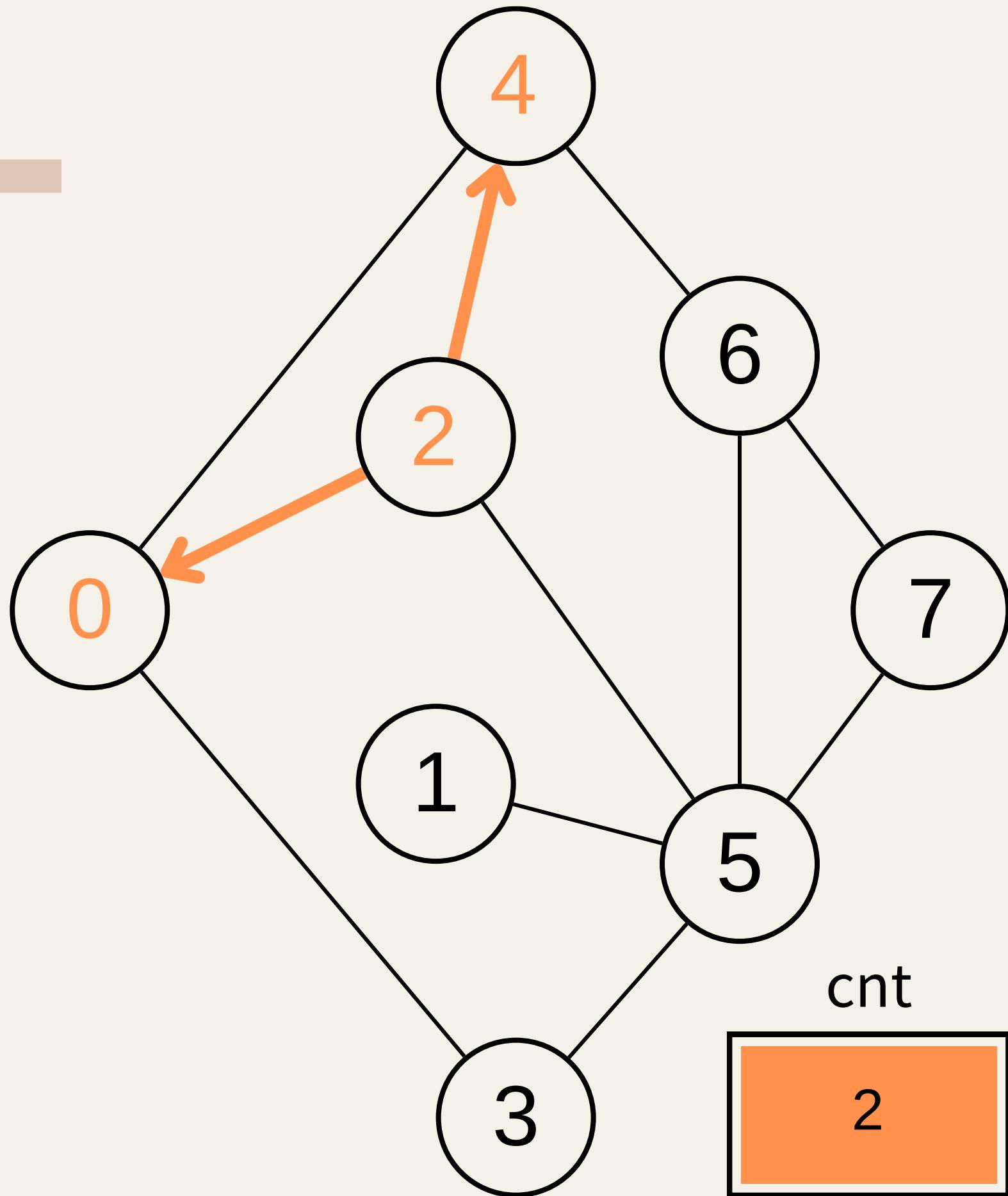
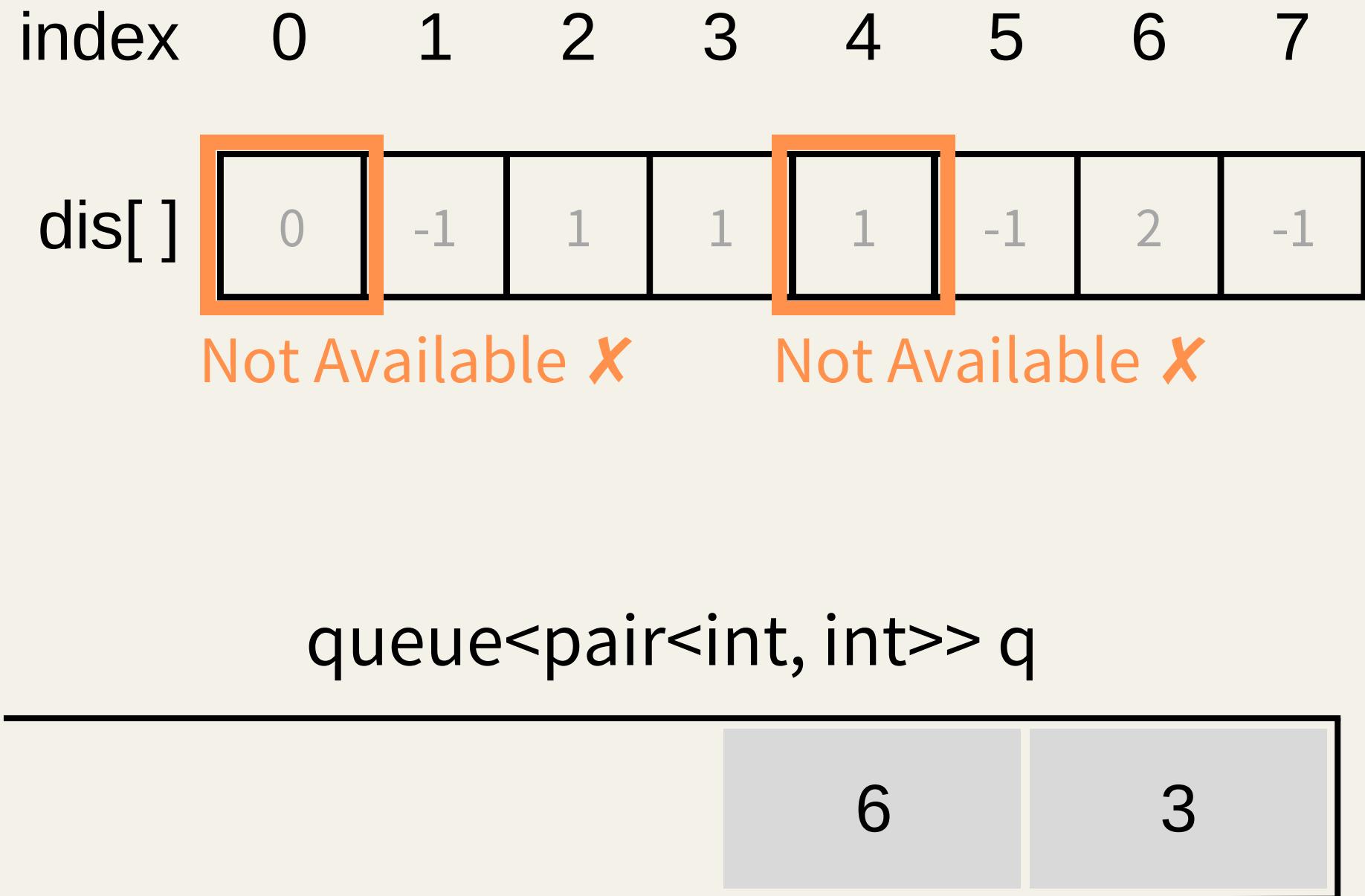
index 0 1 2 3 4 5 6 7

dis[] [0 | -1 | 1 | 1 | 1 | -1 | 2 | -1]

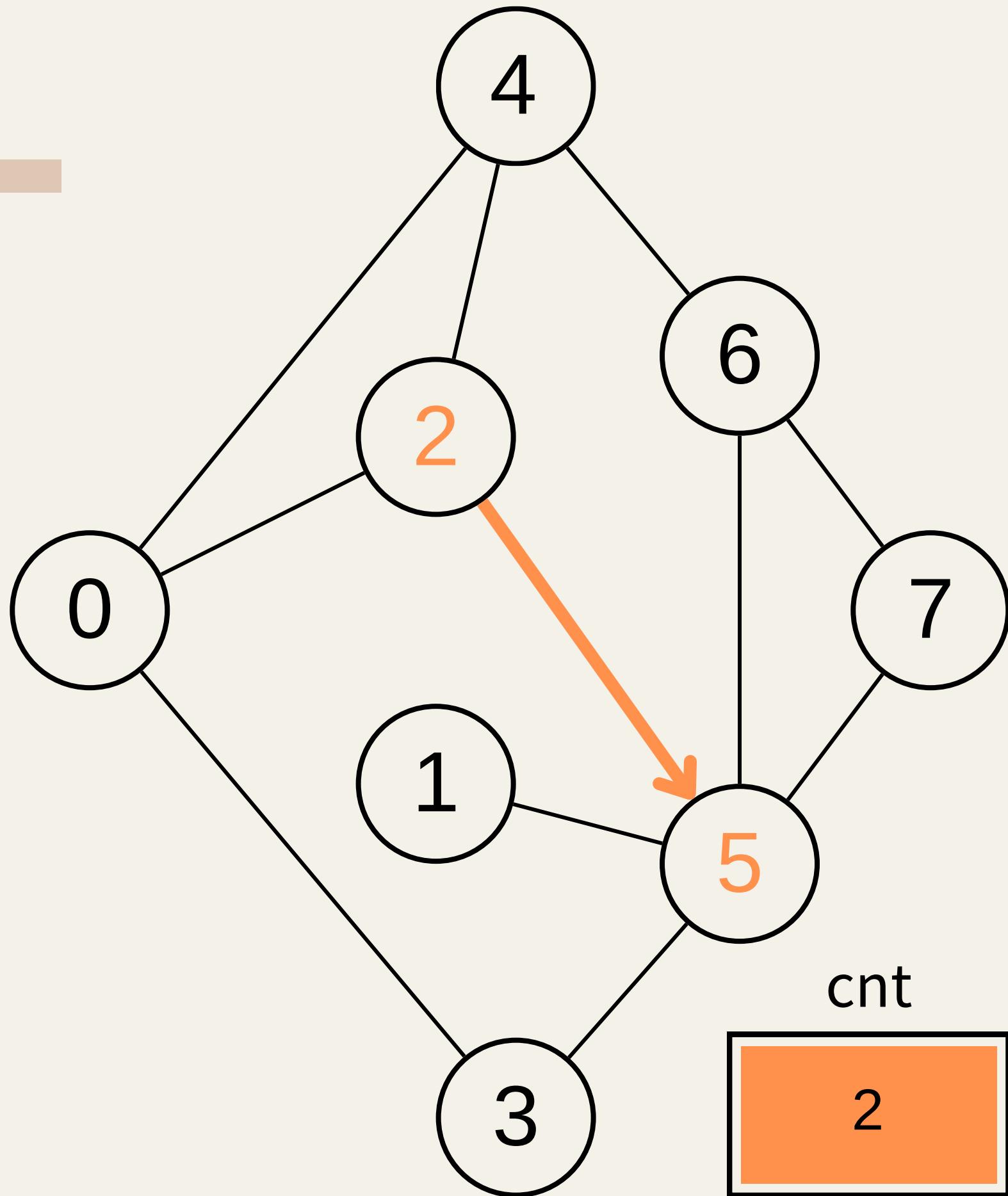
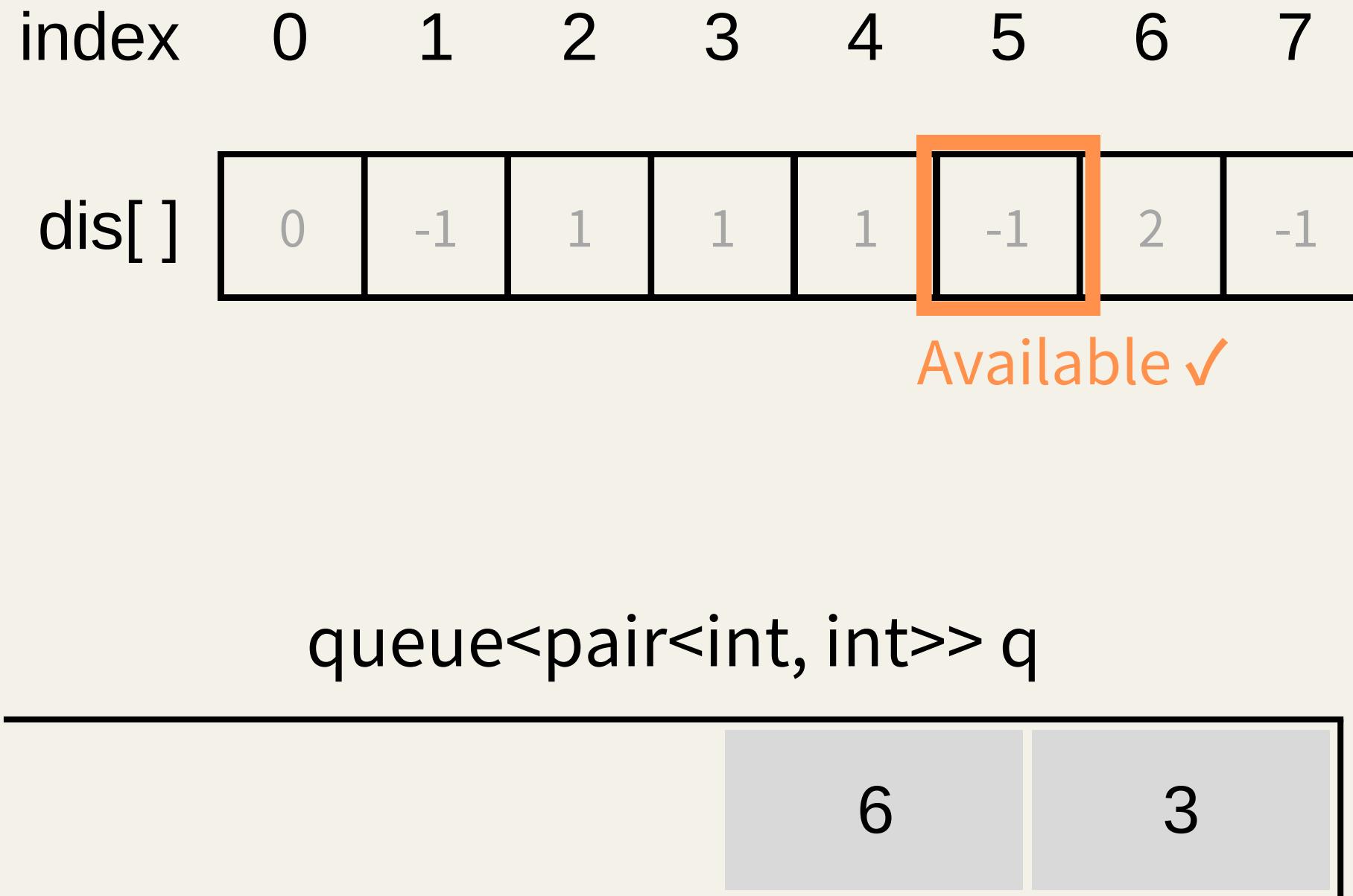
queue<pair<int, int>> q



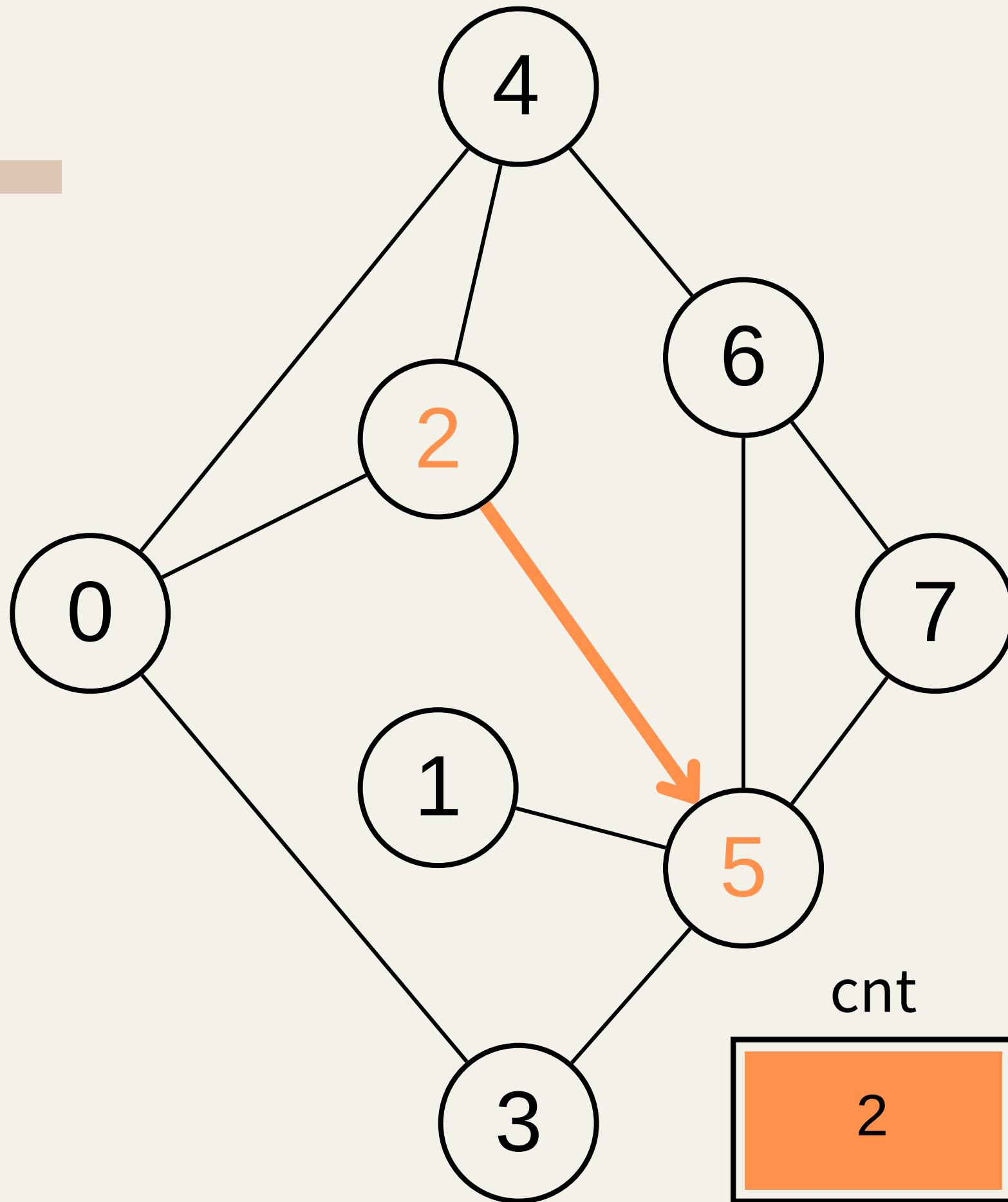
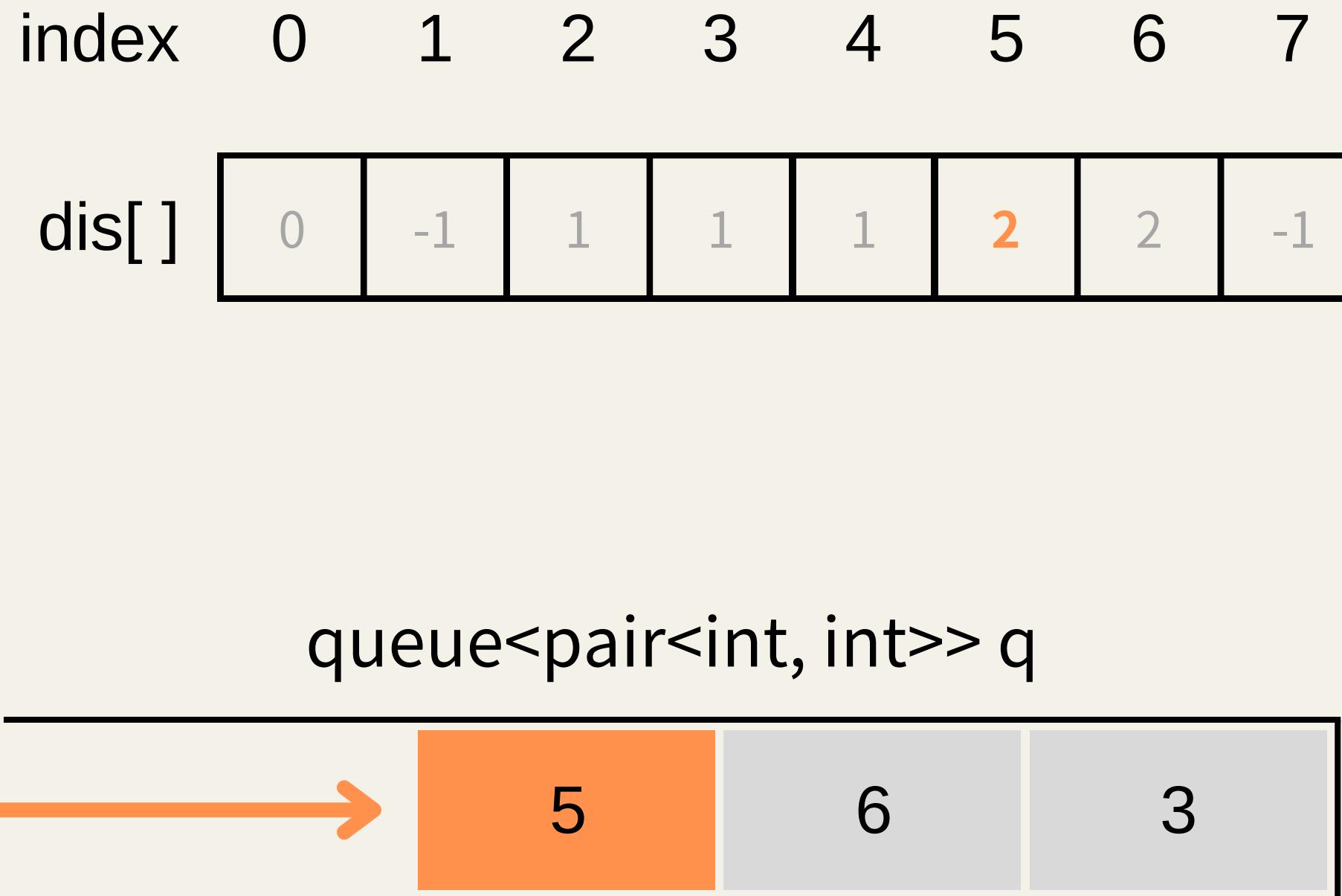
Operating BFS



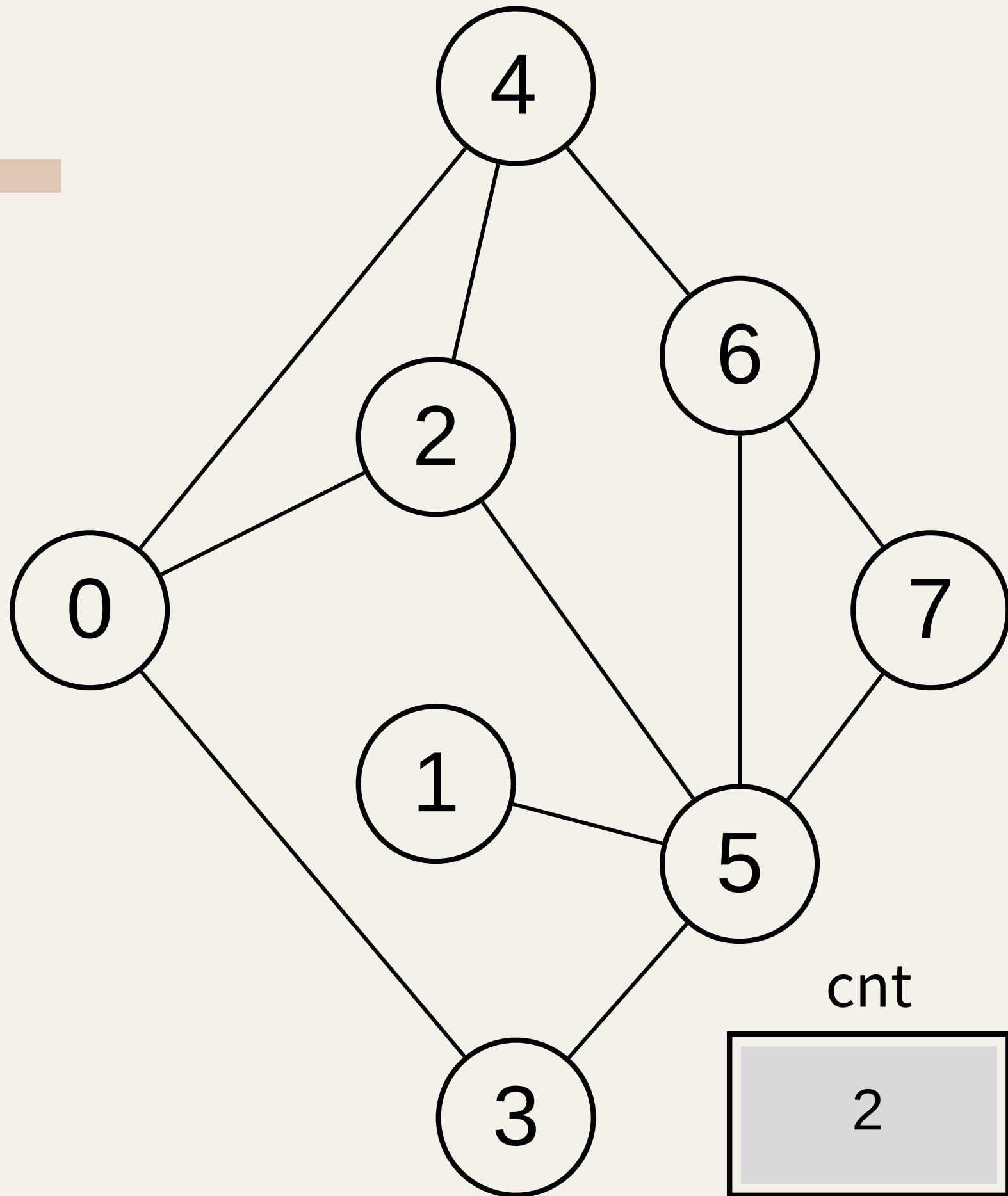
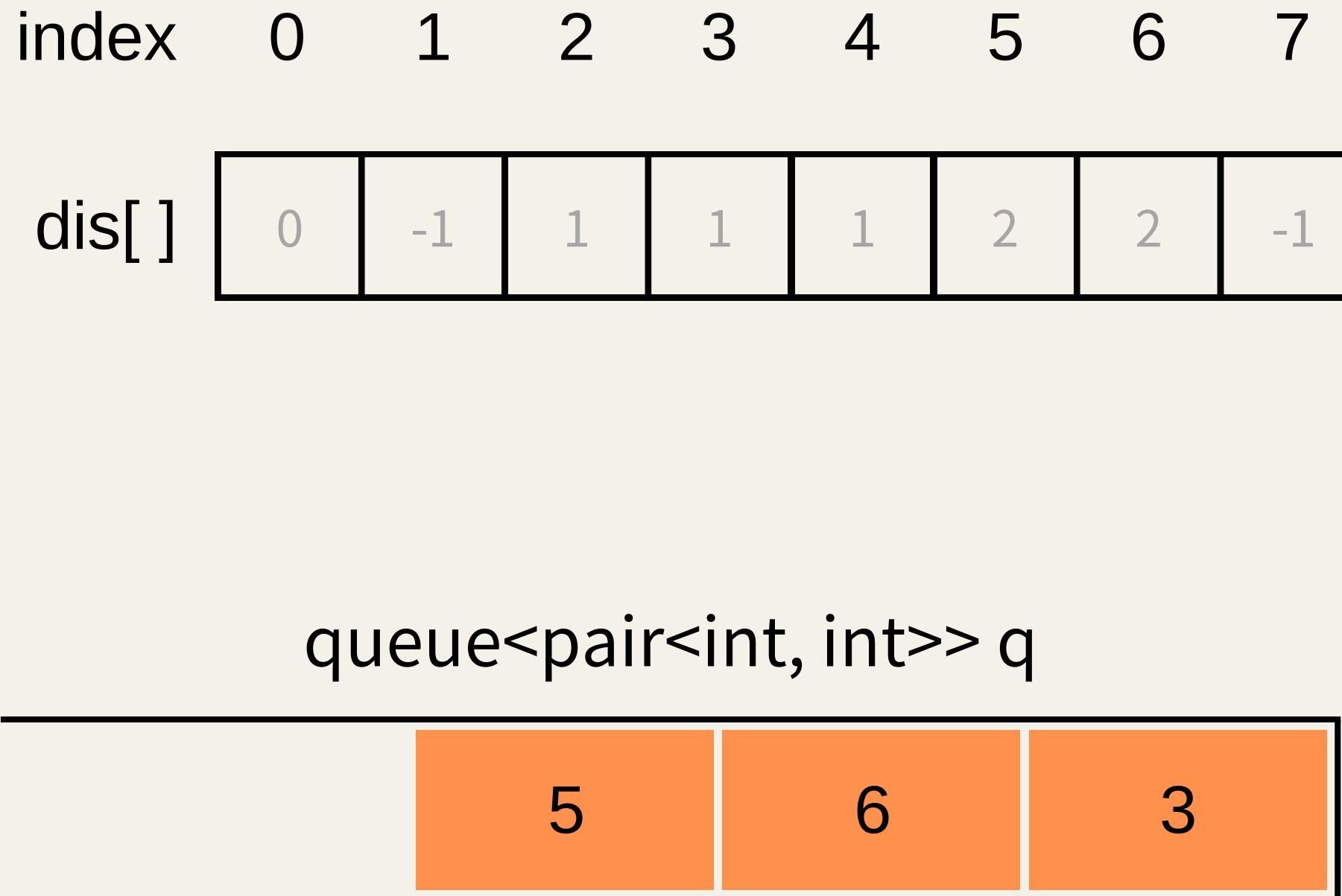
Operating BFS



Operating BFS



Operating BFS

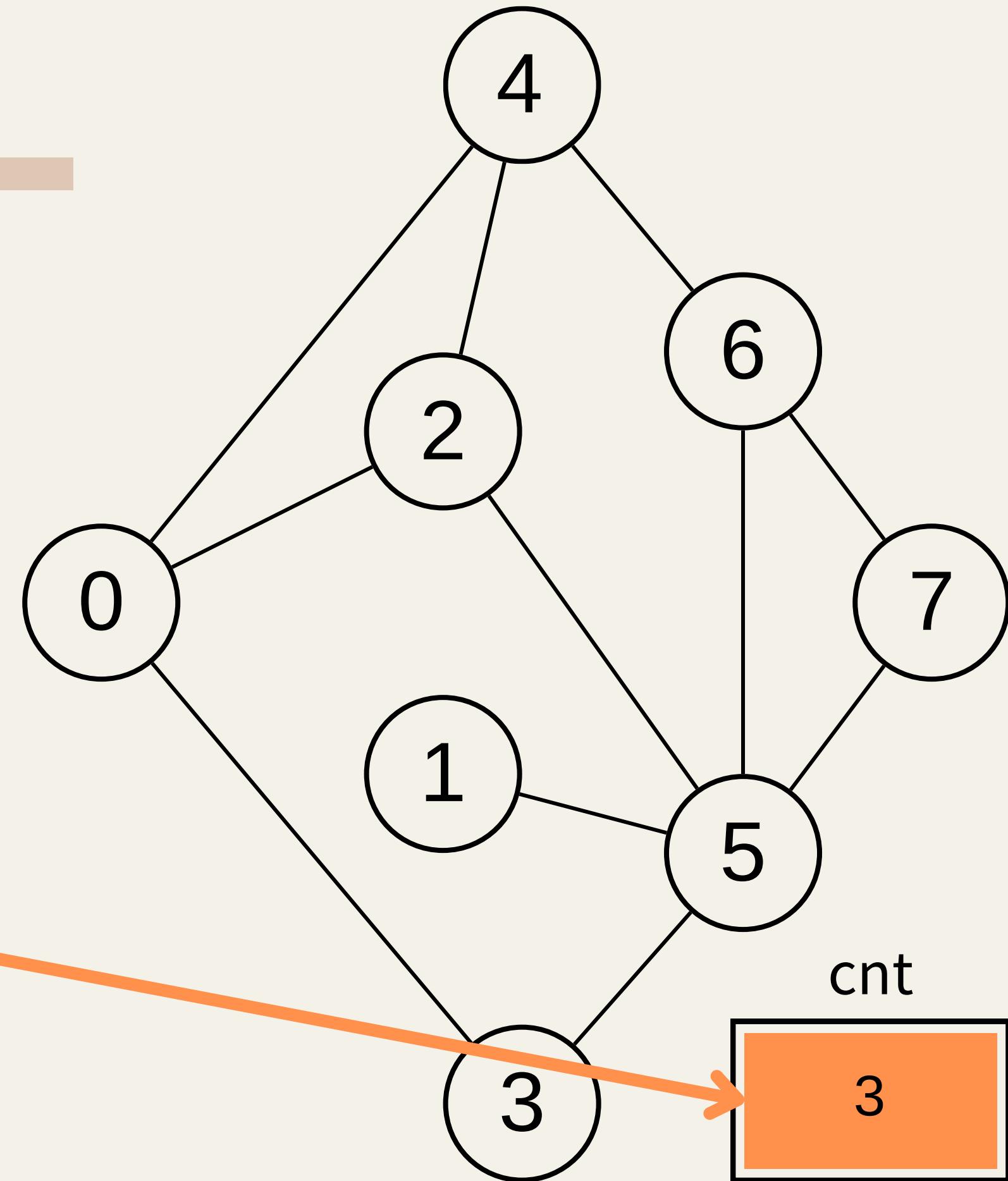
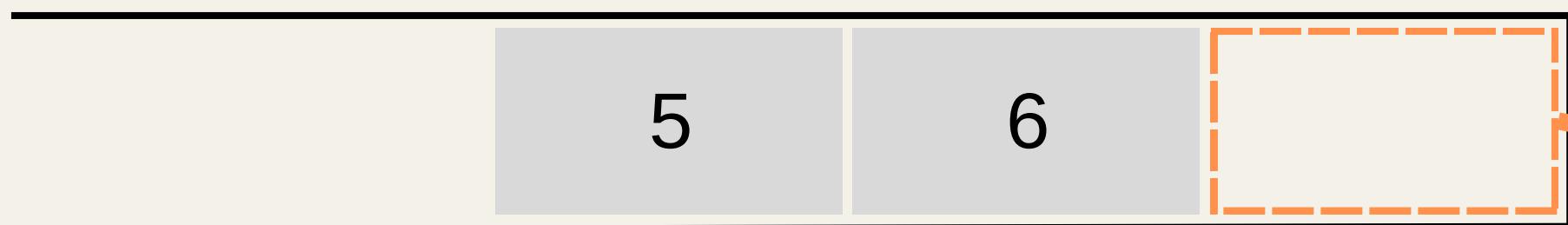


Operating BFS

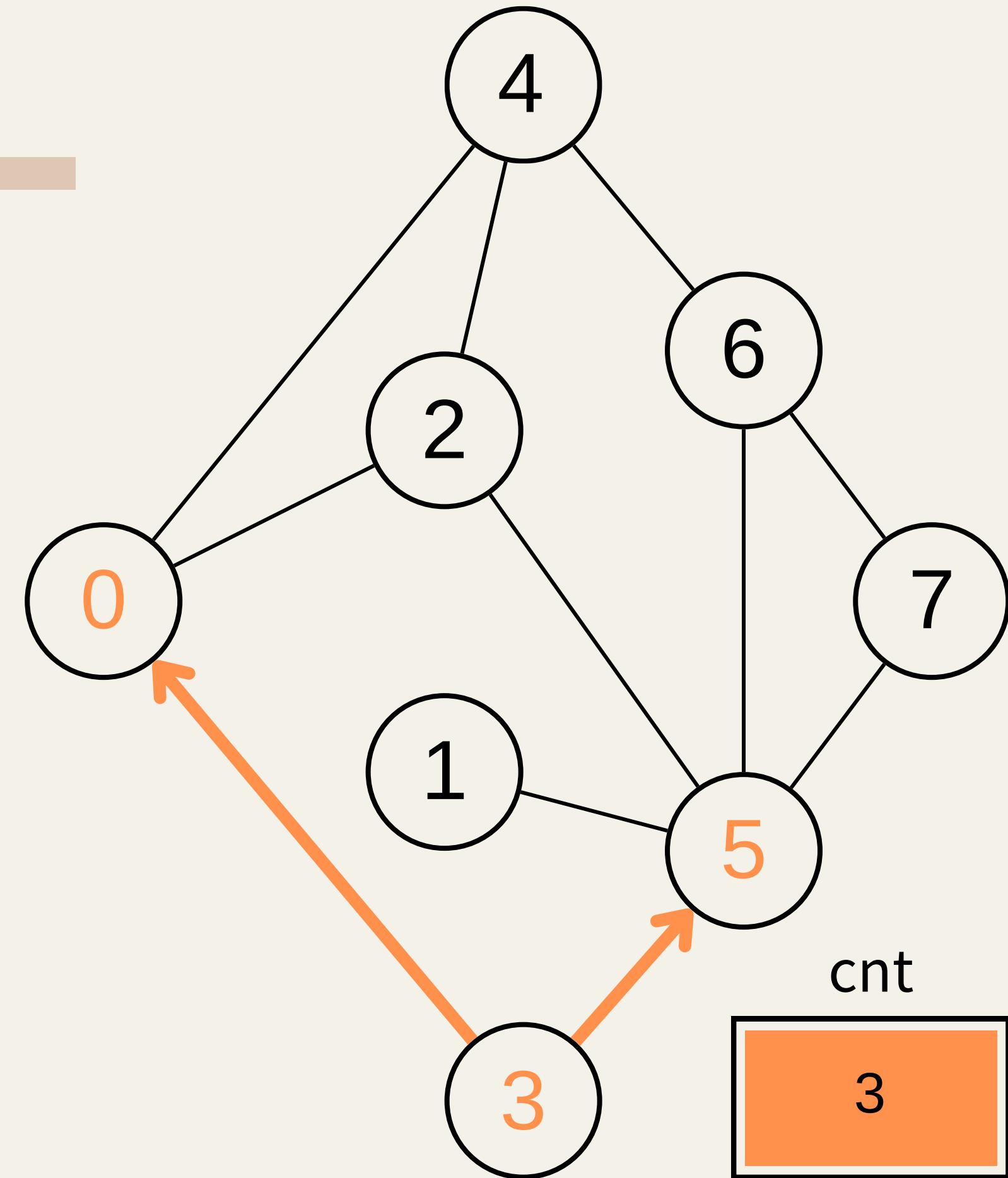
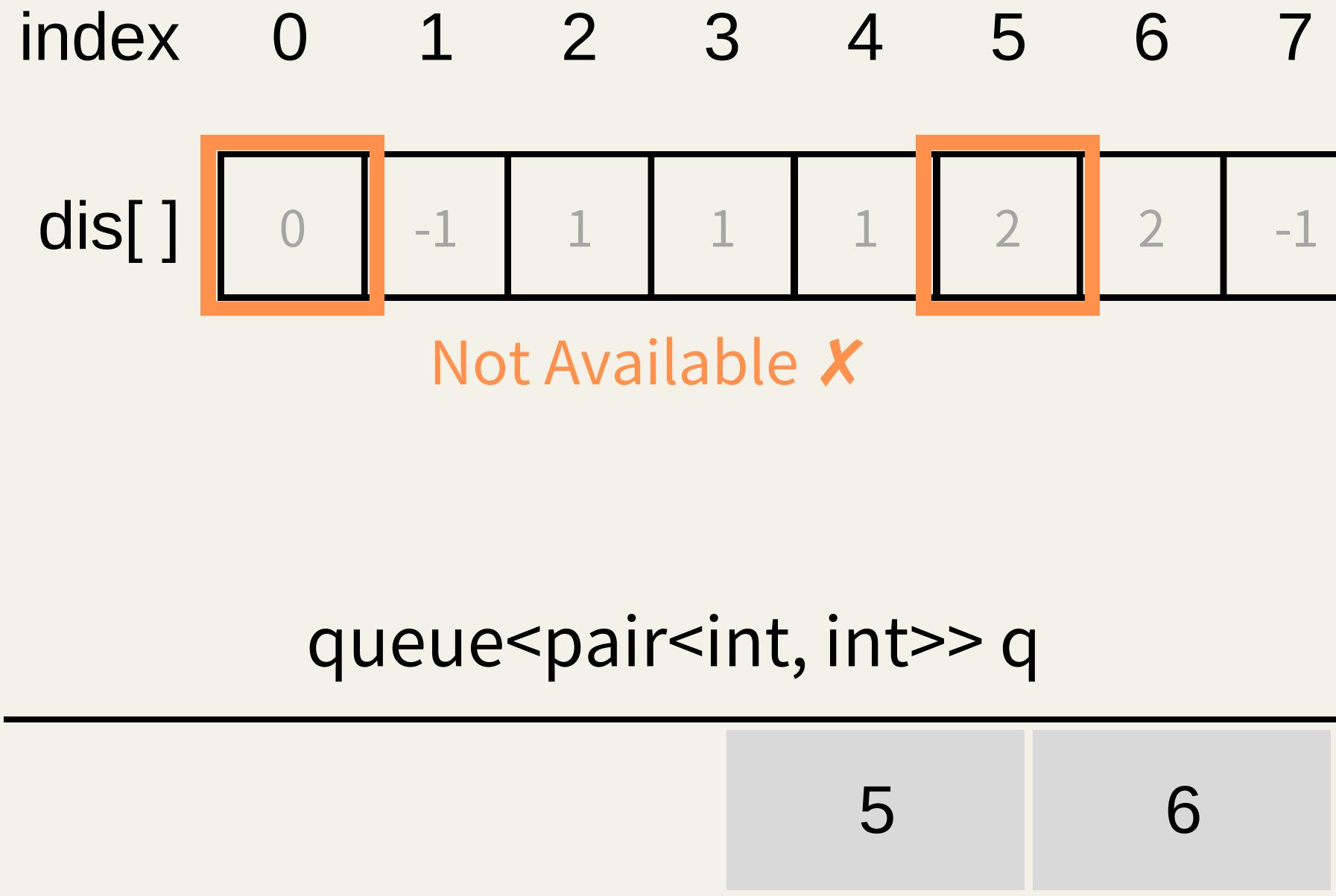
index 0 1 2 3 4 5 6 7

dis[] [0 | -1 | 1 | 1 | 1 | 2 | 2 | -1]

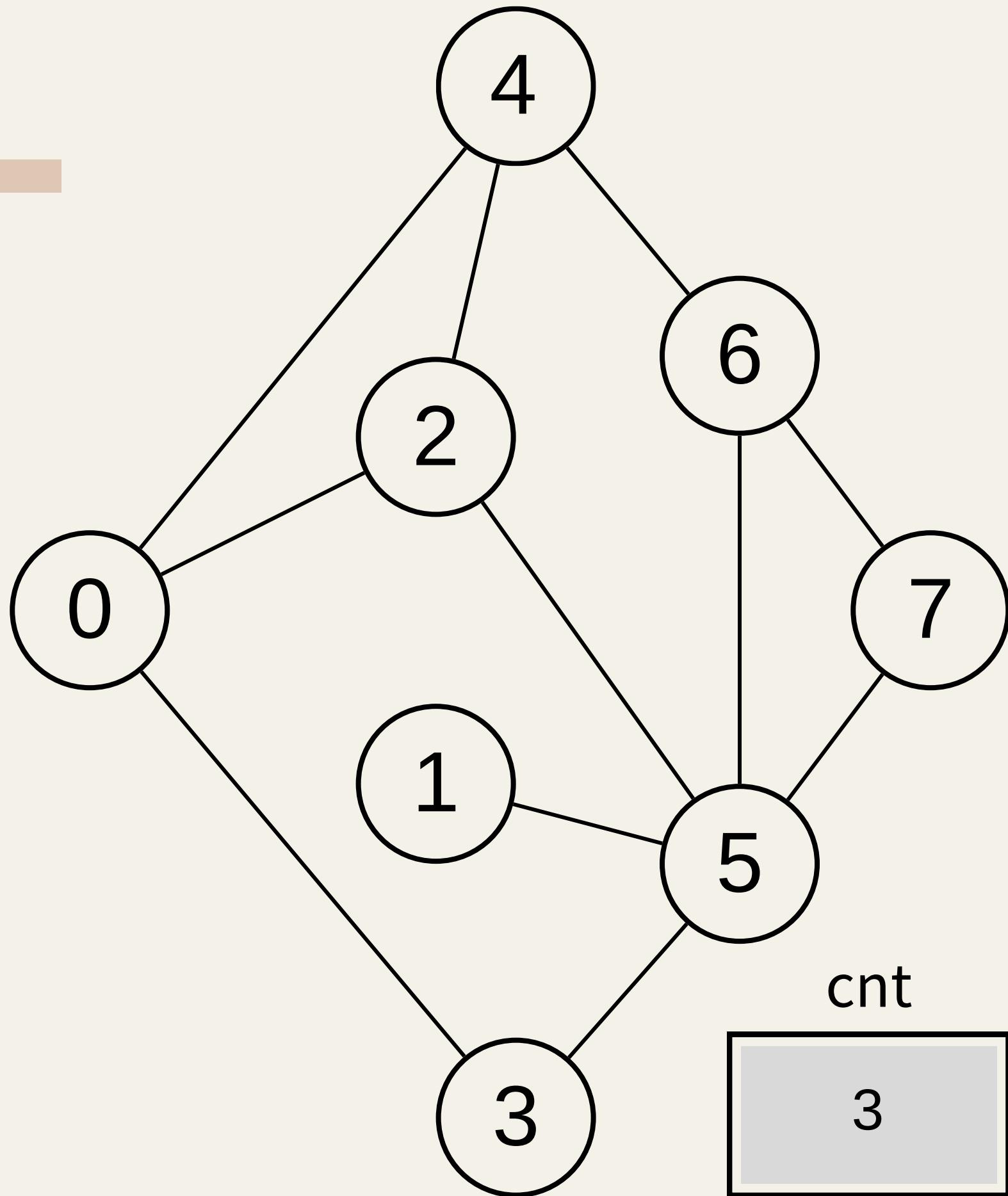
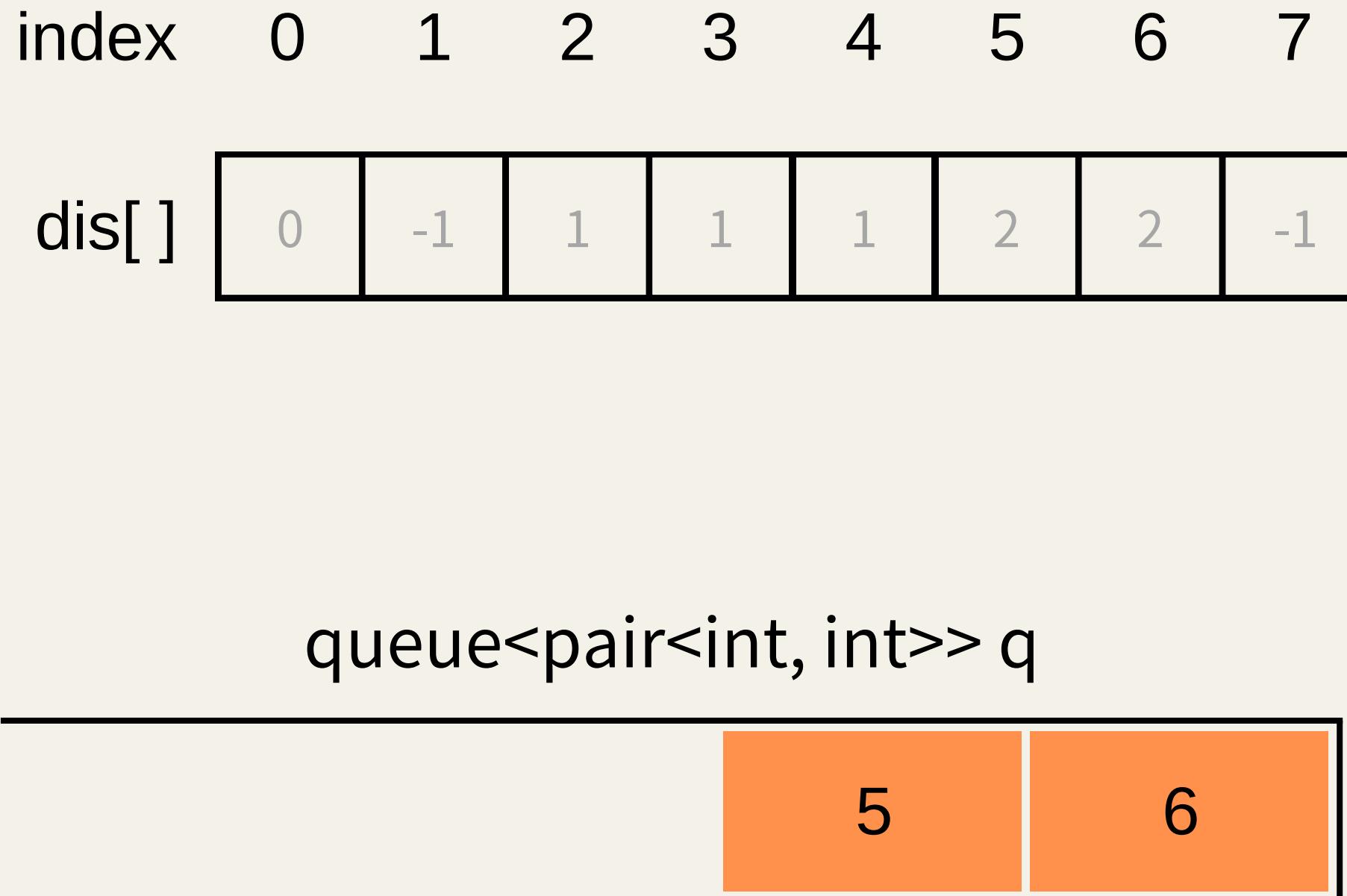
queue<pair<int, int>> q



Operating BFS



Operating BFS



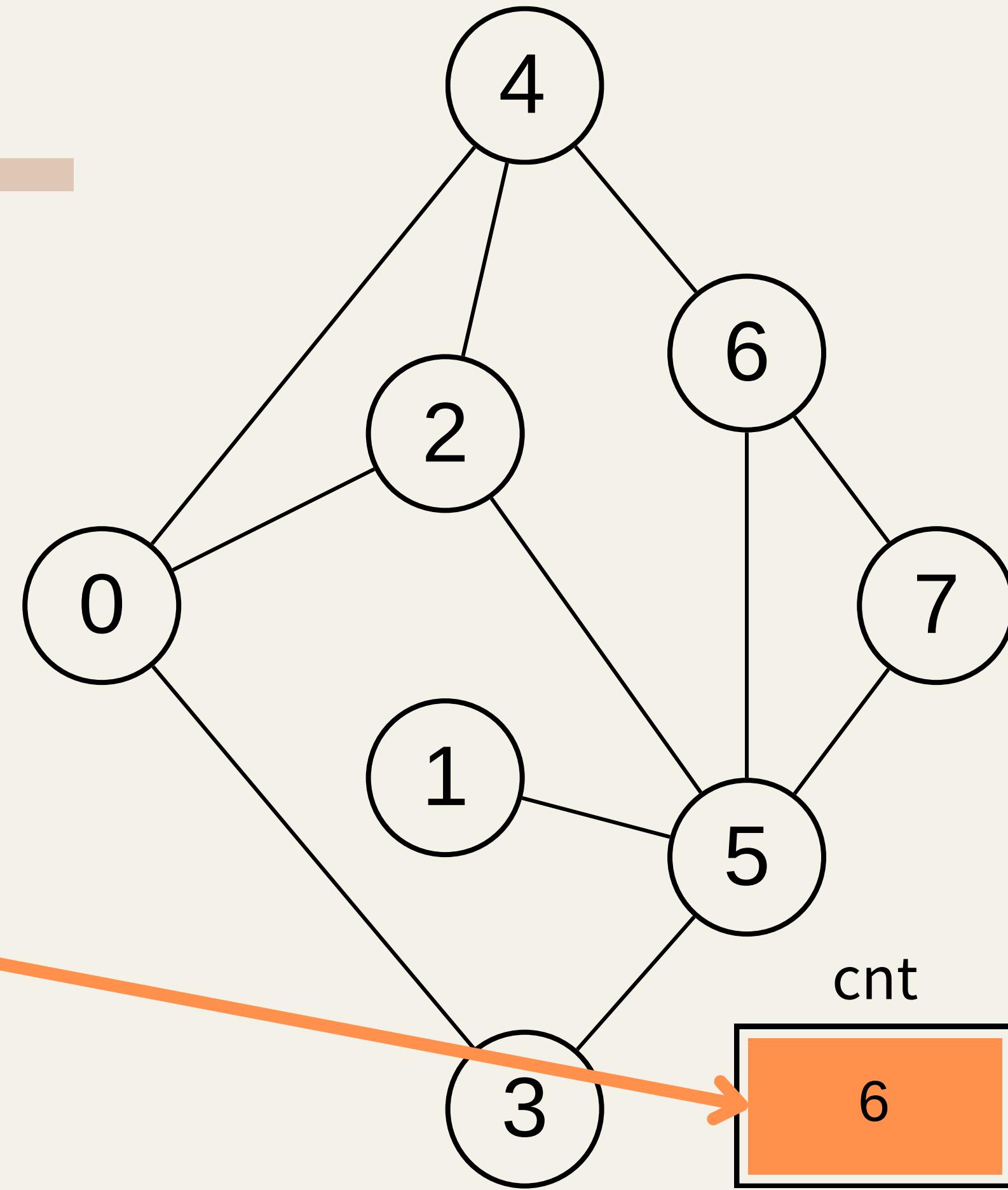
Operating BFS

index 0 1 2 3 4 5 6 7

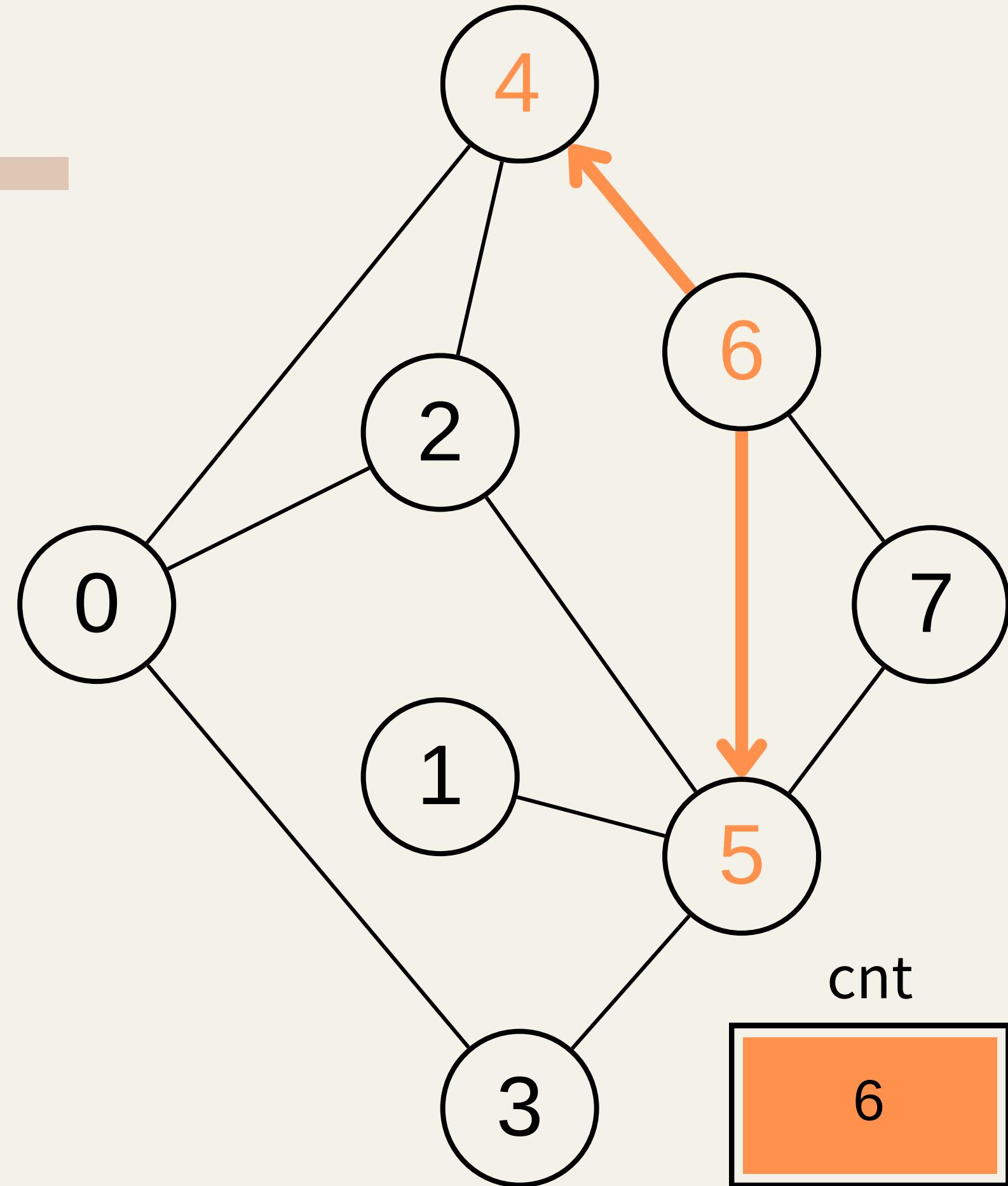
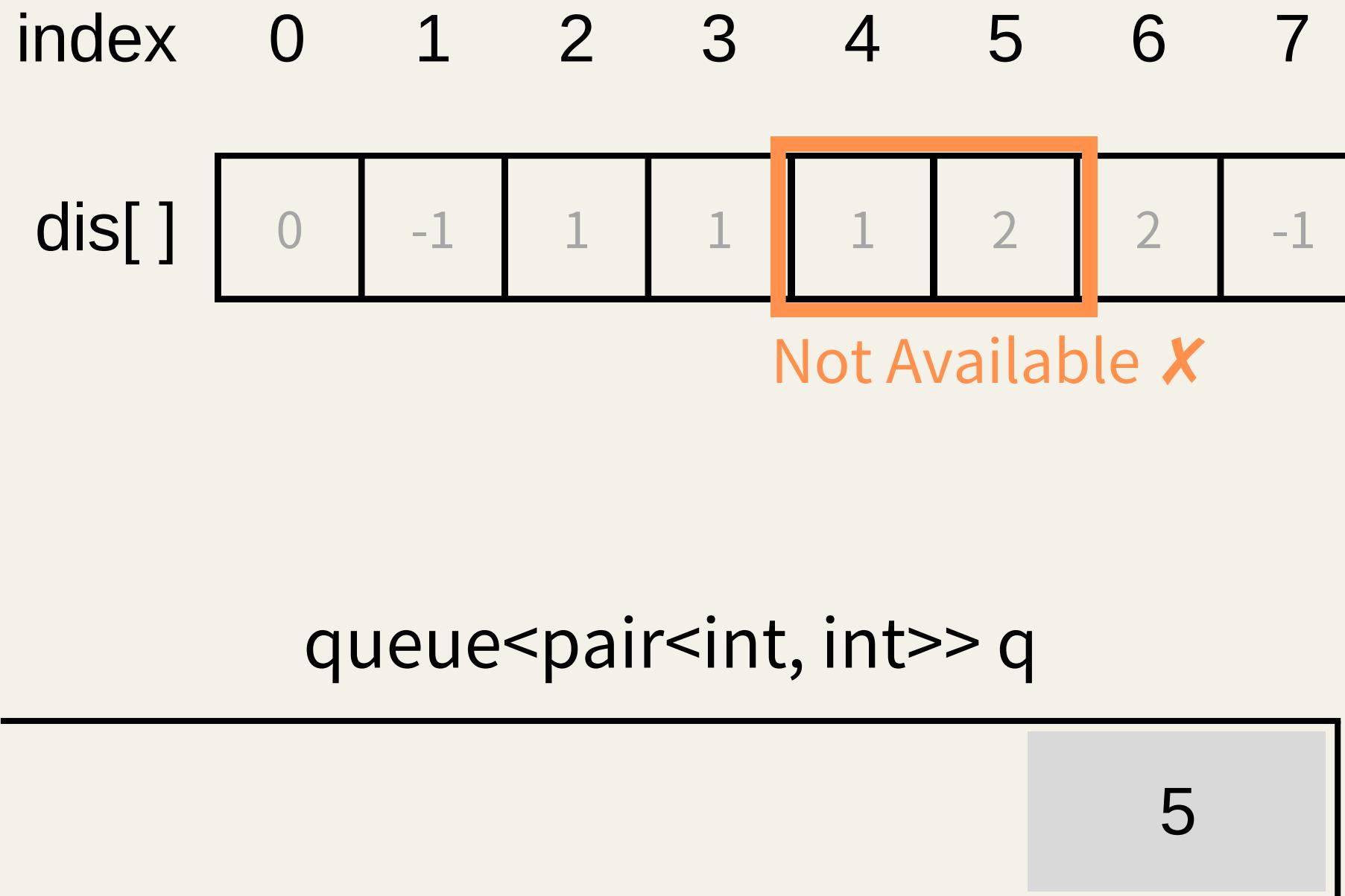
dis[] [0 | -1 | 1 | 1 | 1 | 2 | 2 | -1]

queue<pair<int, int>> q

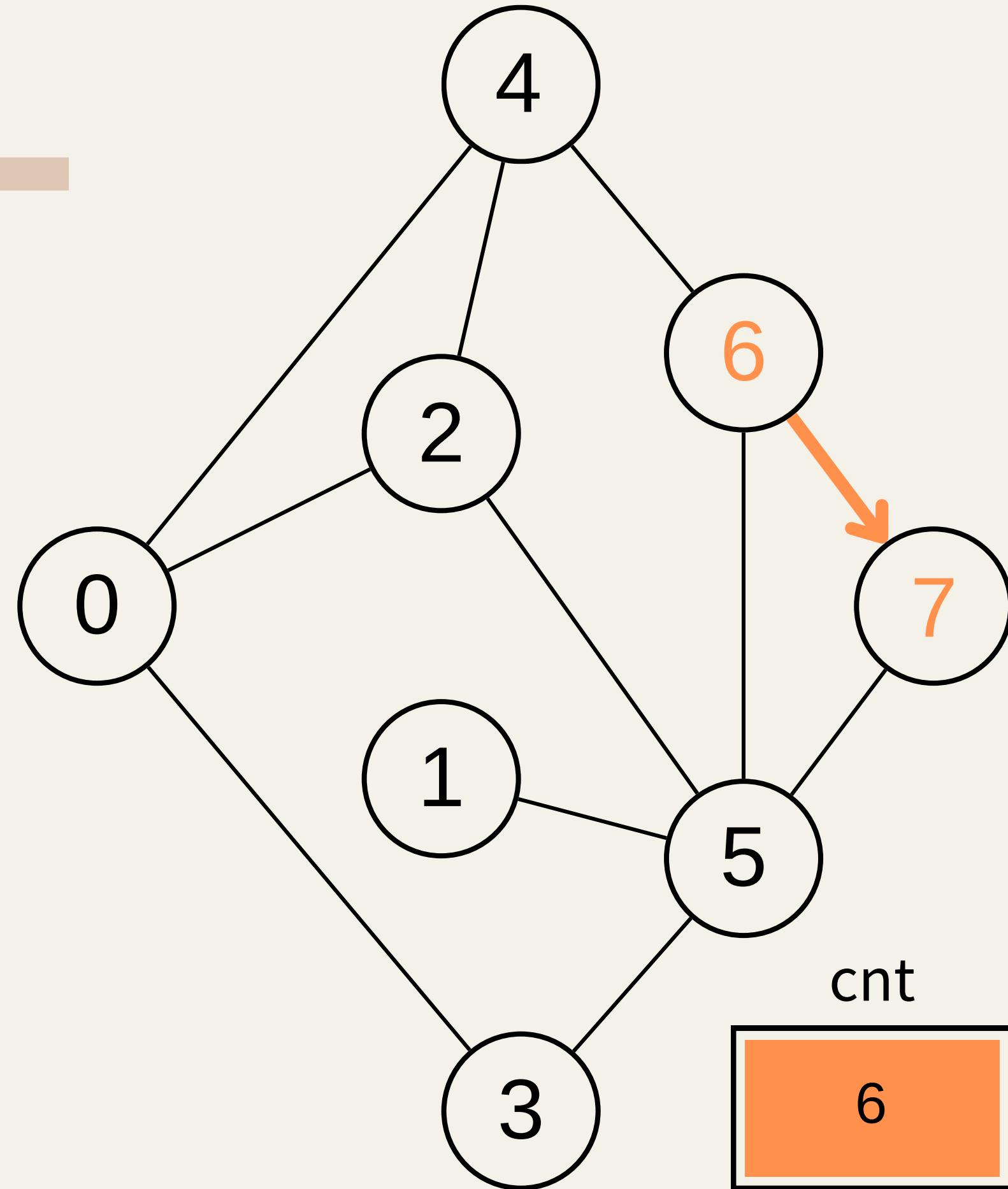
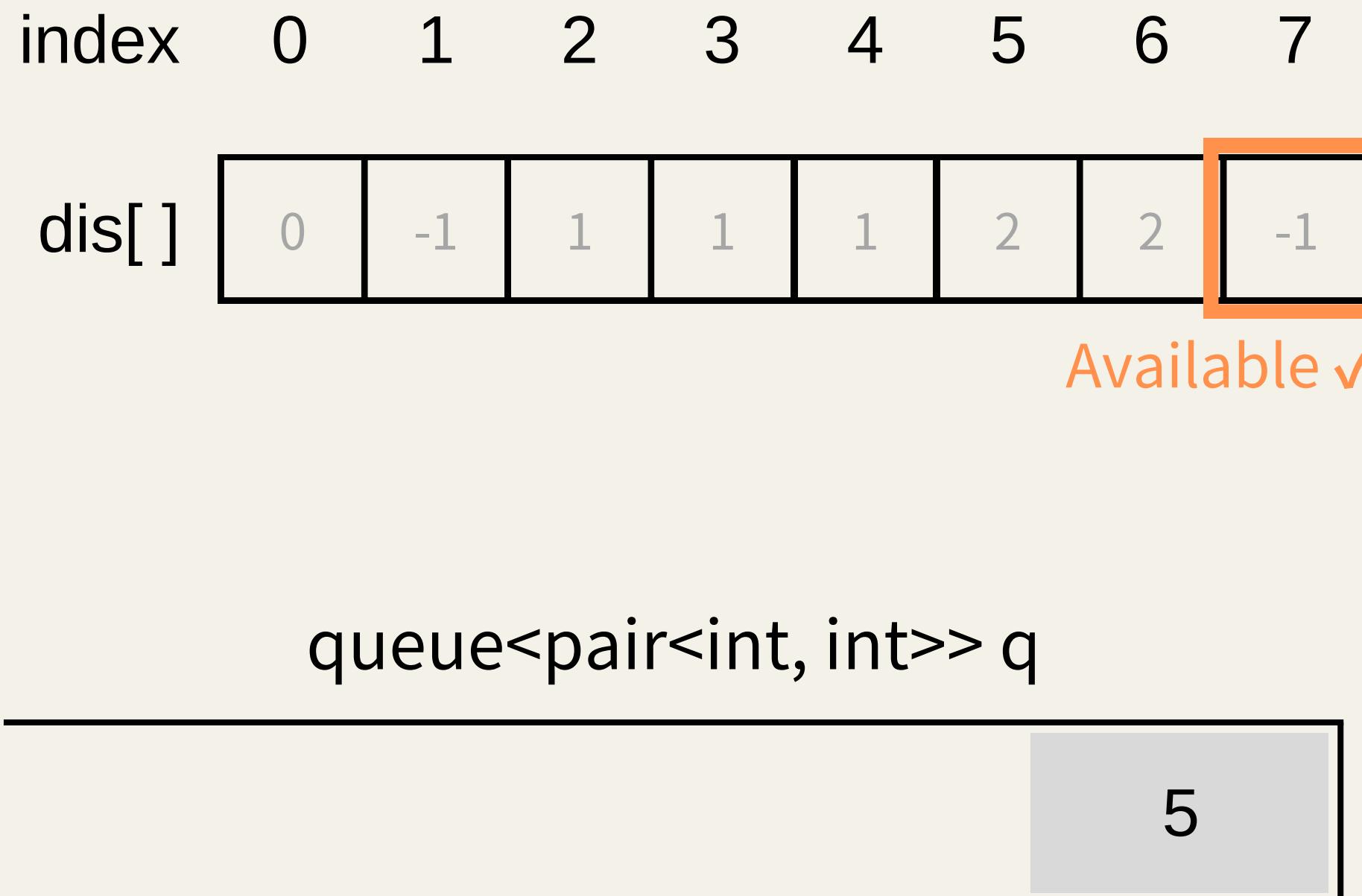
5



Operating BFS



Operating BFS



Operating BFS

index 0 1 2 3 4 5 6 7

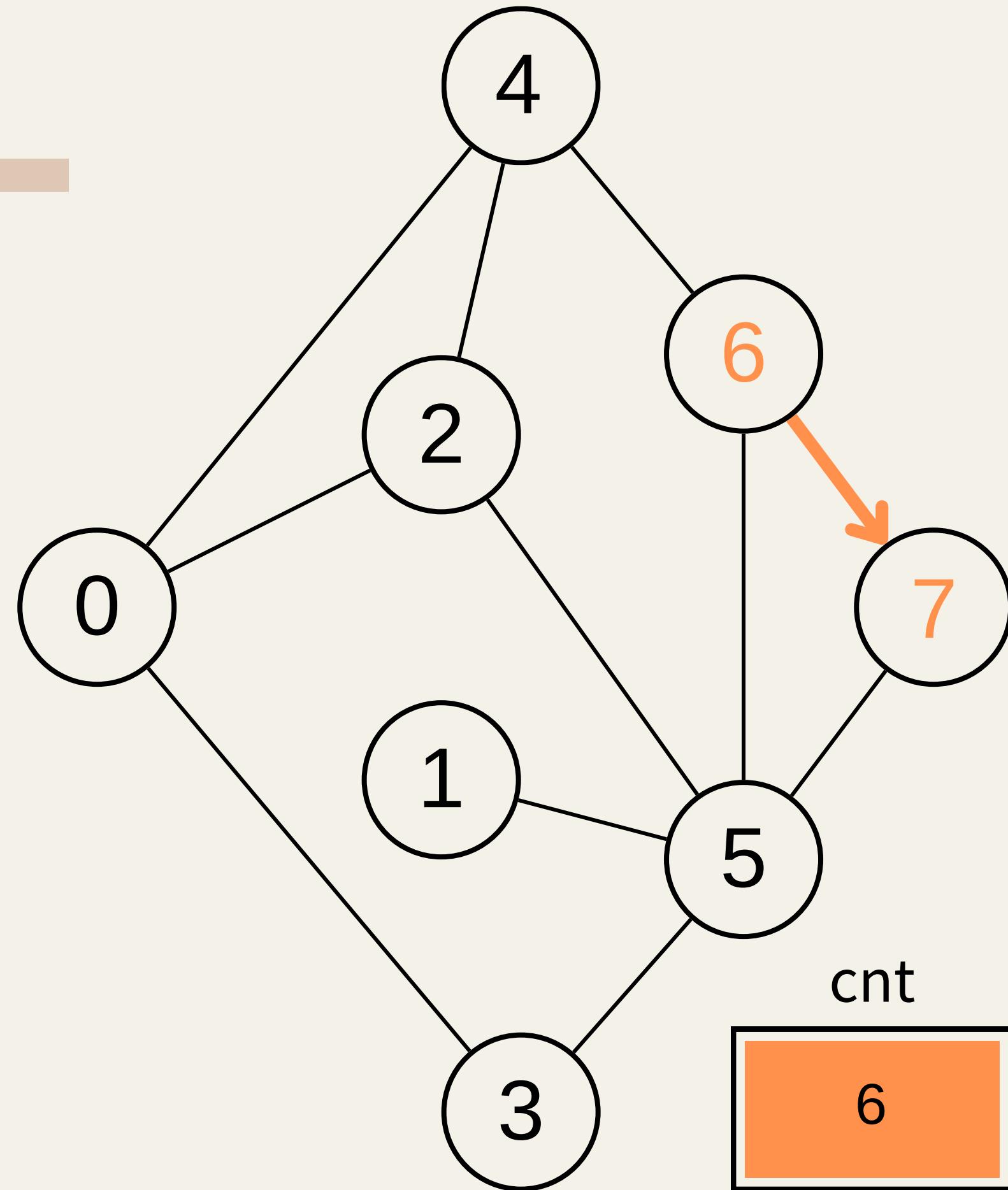
dis[]

0	-1	1	1	1	2	2	3
---	----	---	---	---	---	---	---

This is the shortest path !

queue<pair<int, int>> q

	5
--	---





Depth First Search

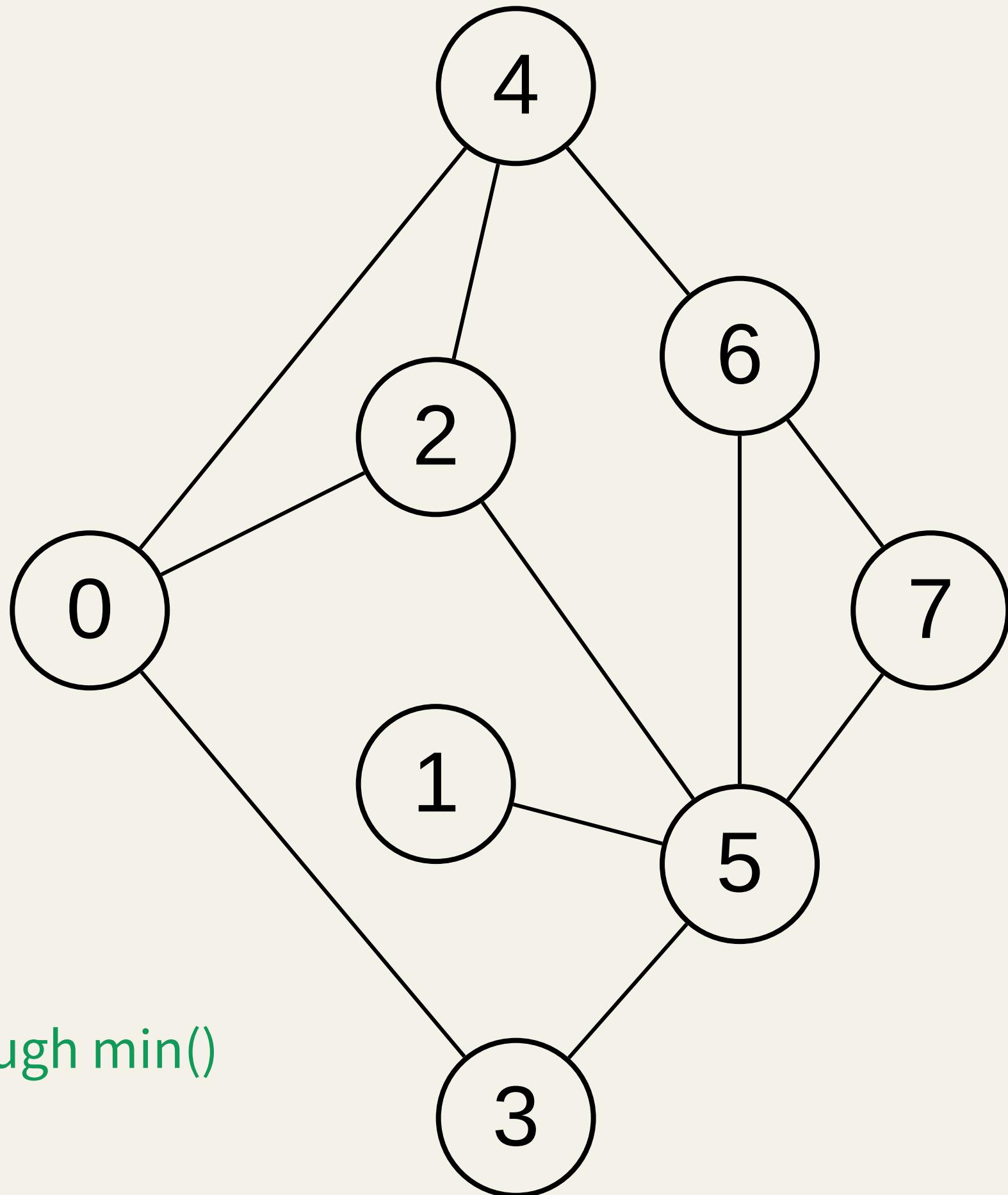
深度優先搜尋演算法

Initialize

index	0	1	2	3	4	5	6	7
used[]	False							
res	INF							

- ① Set all of the value in used array to default value → Define false for not been processed

- ② Set res to a large enough value
→ Because we have to constantly reduce it through min()



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False						
res	INF							

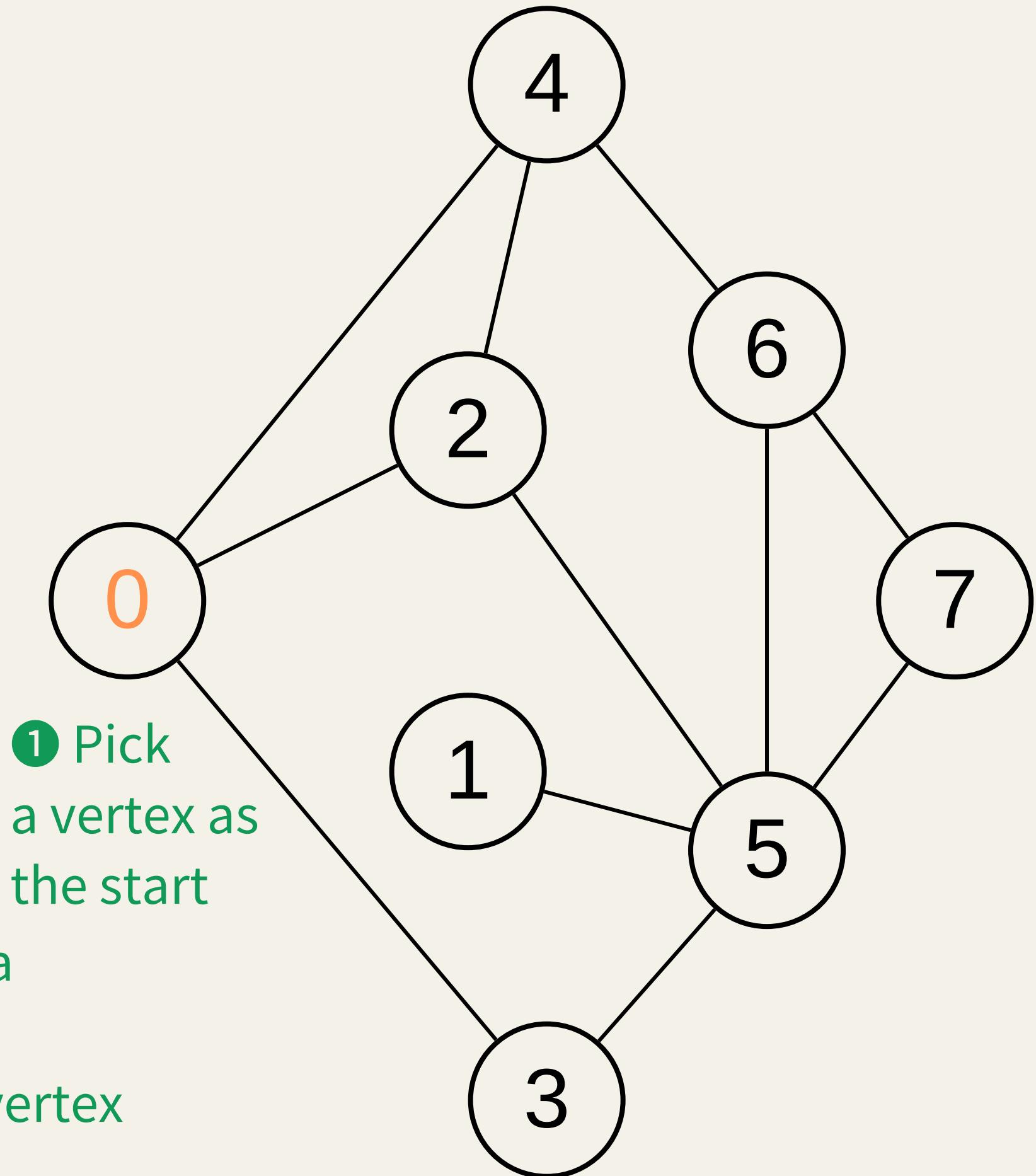
② Set the statement of starting vertex to true

cnt	dis
0	0

③ Declare two variable cnt and dis to store the data

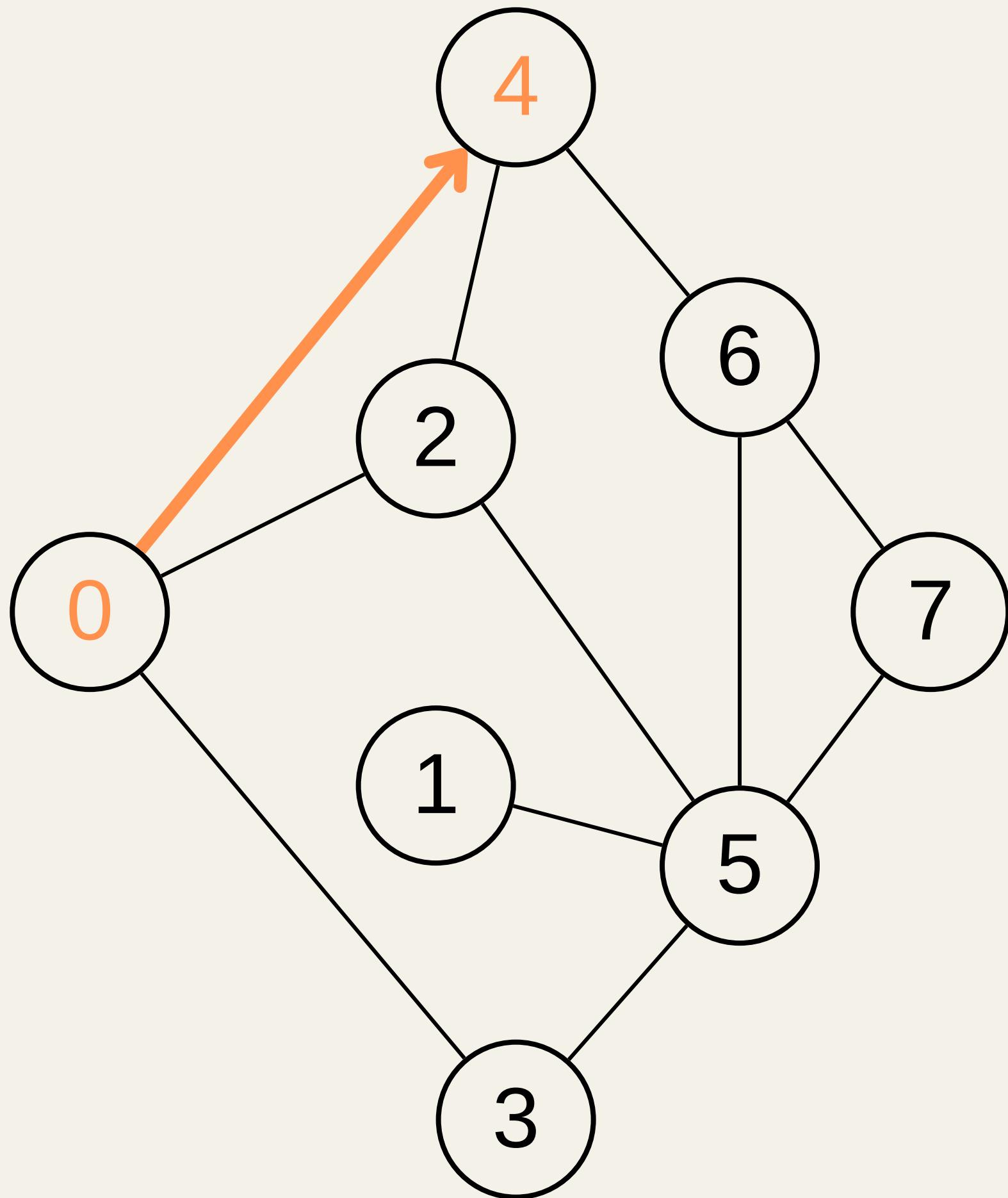
cnt → current vertex

dis → distance from starting vertex to current vertex



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False						
	Available ✓							
Check whether adjacent vertex already been used								
res	INF							
cnt		0						
dis			0					



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	False	False
res	INF							

- ① If cnt vertex hadn't been used
→ Update its statement

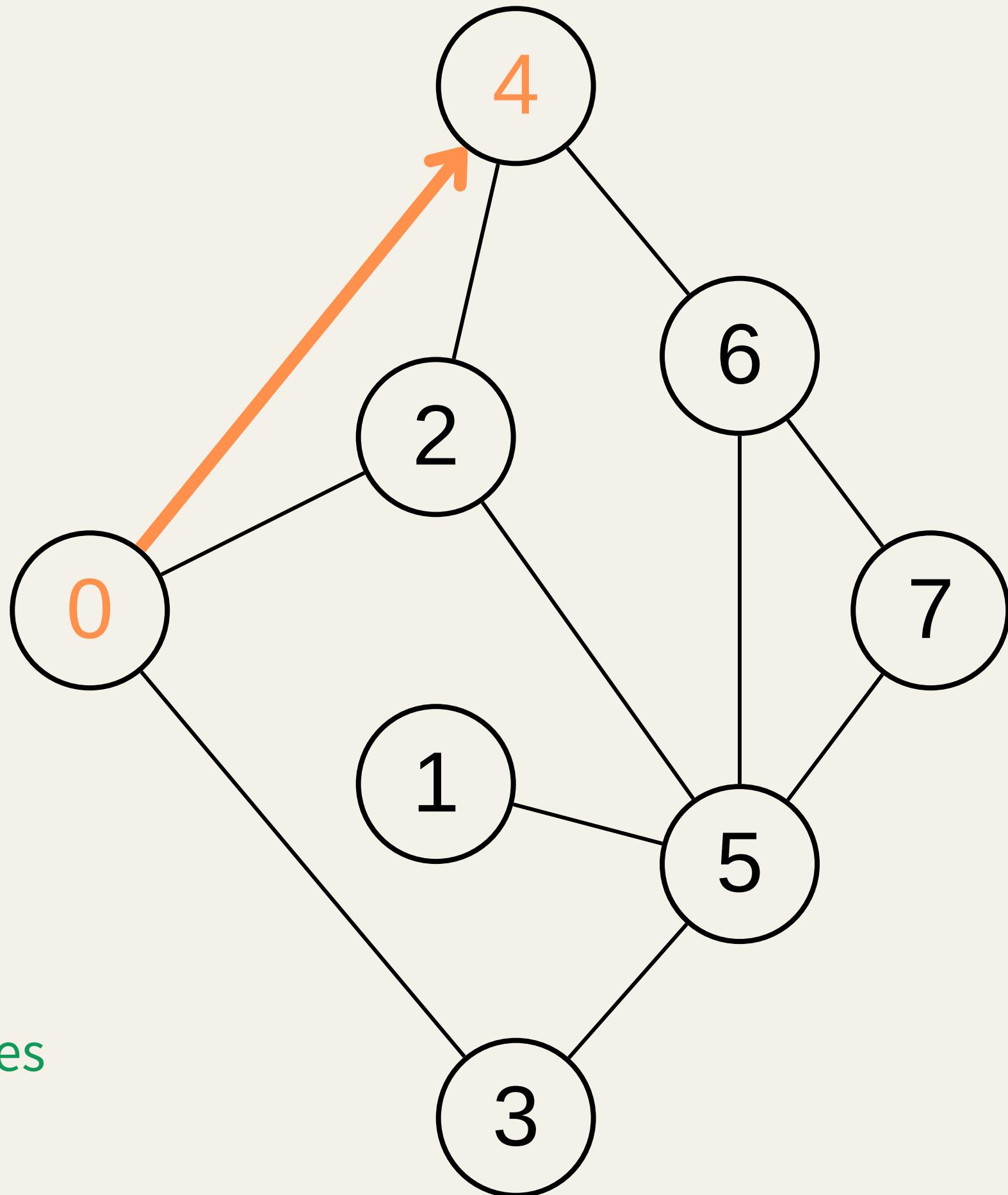
cnt

4

dis

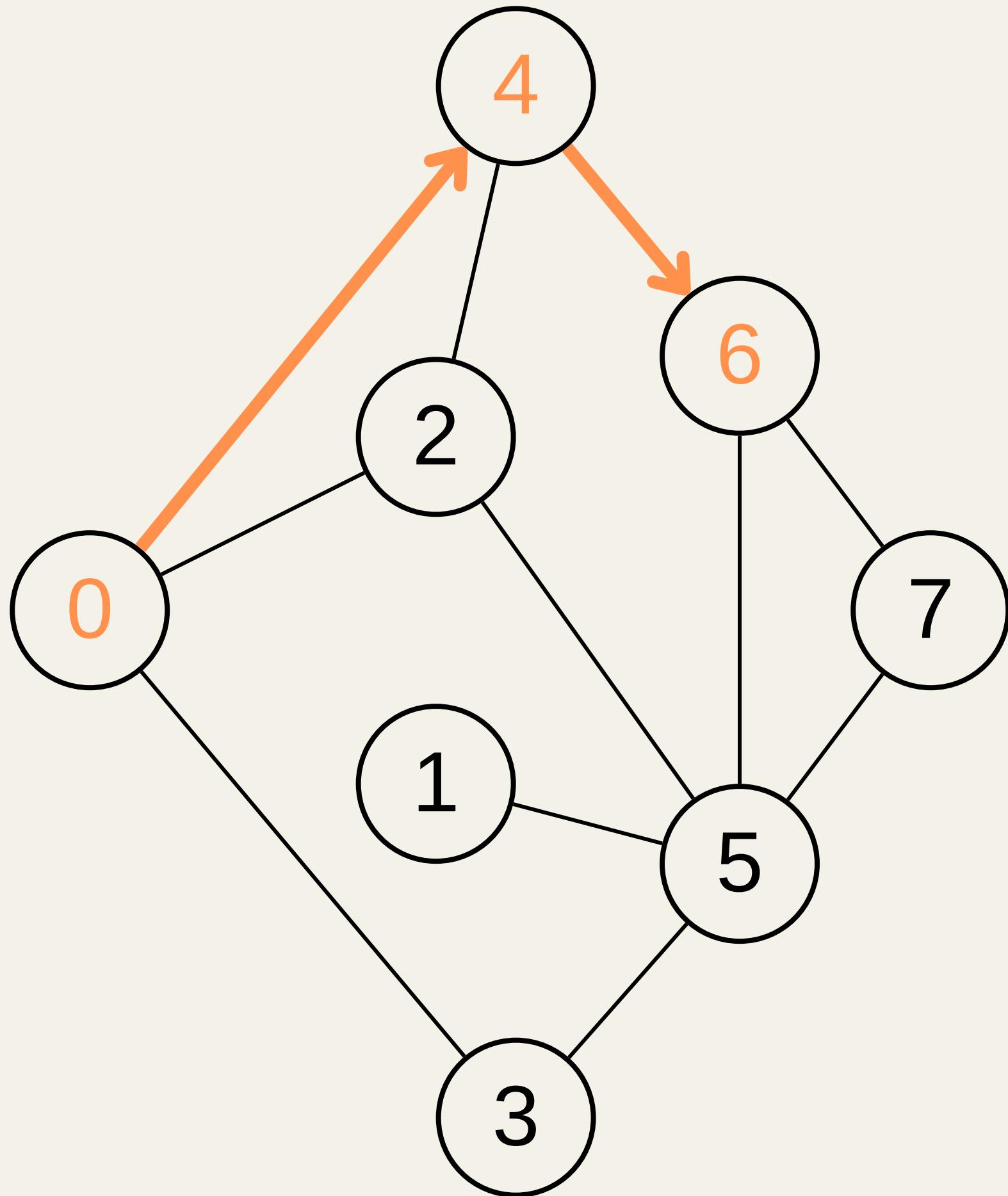
1

- ② If cnt vertex hadn't been used
→ Update cnt and dis variables



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	False	False
	Available ✓							
Check whether adjacent vertex already been used								
res	INF	cnt	4	dis	1			



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	True	False
res	INF							

- ① If cnt vertex hadn't been used
→ Update its statement

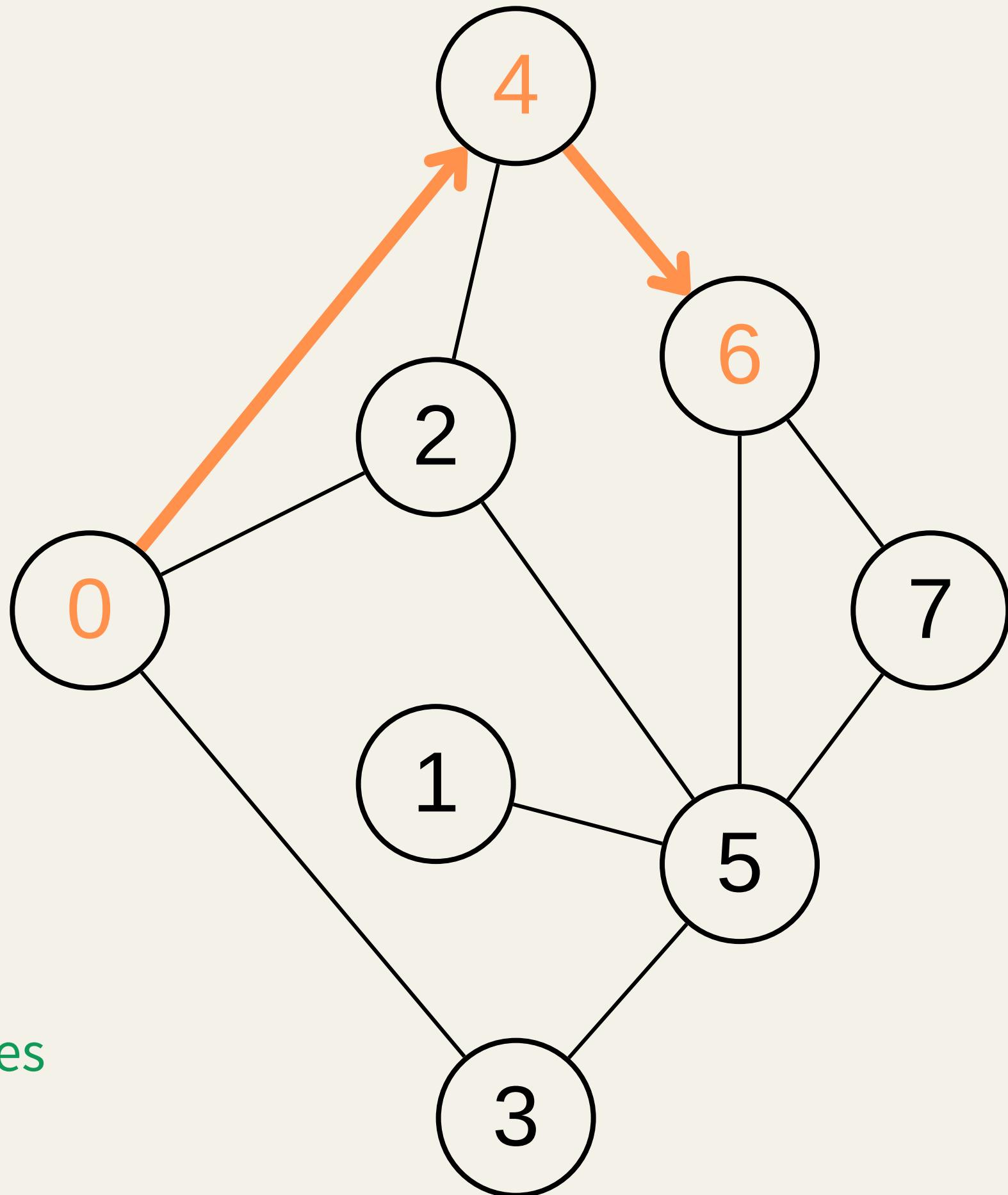
cnt

6

dis

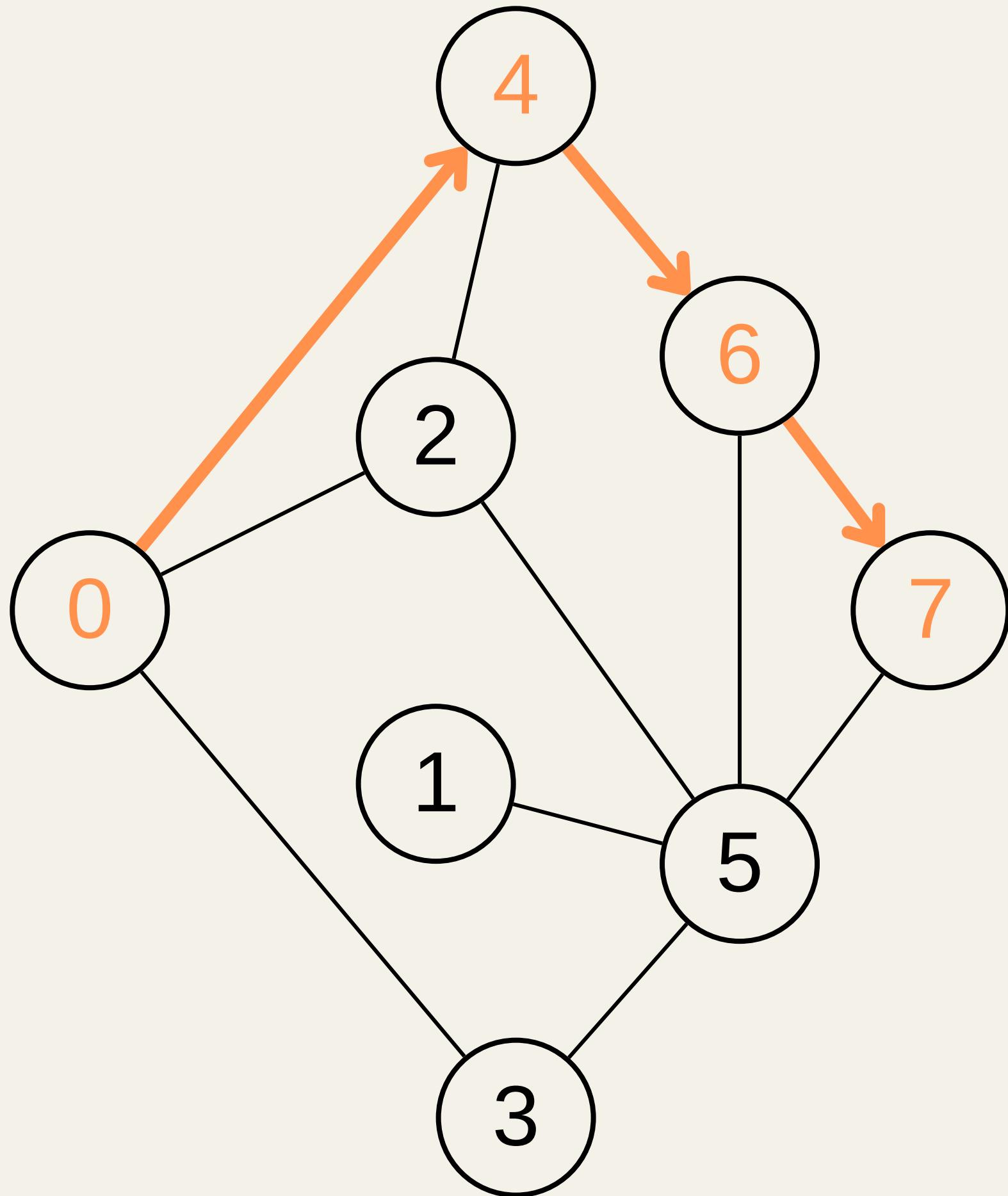
2

- ② If cnt vertex hadn't been used
→ Update cnt and dis variables



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	True	False
	Available ✓							
Check whether adjacent vertex already been used								
res	INF	cnt	6	dis	2			



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	True	True
res	INF							

- ① If cnt vertex hadn't been used
→ Update its statement

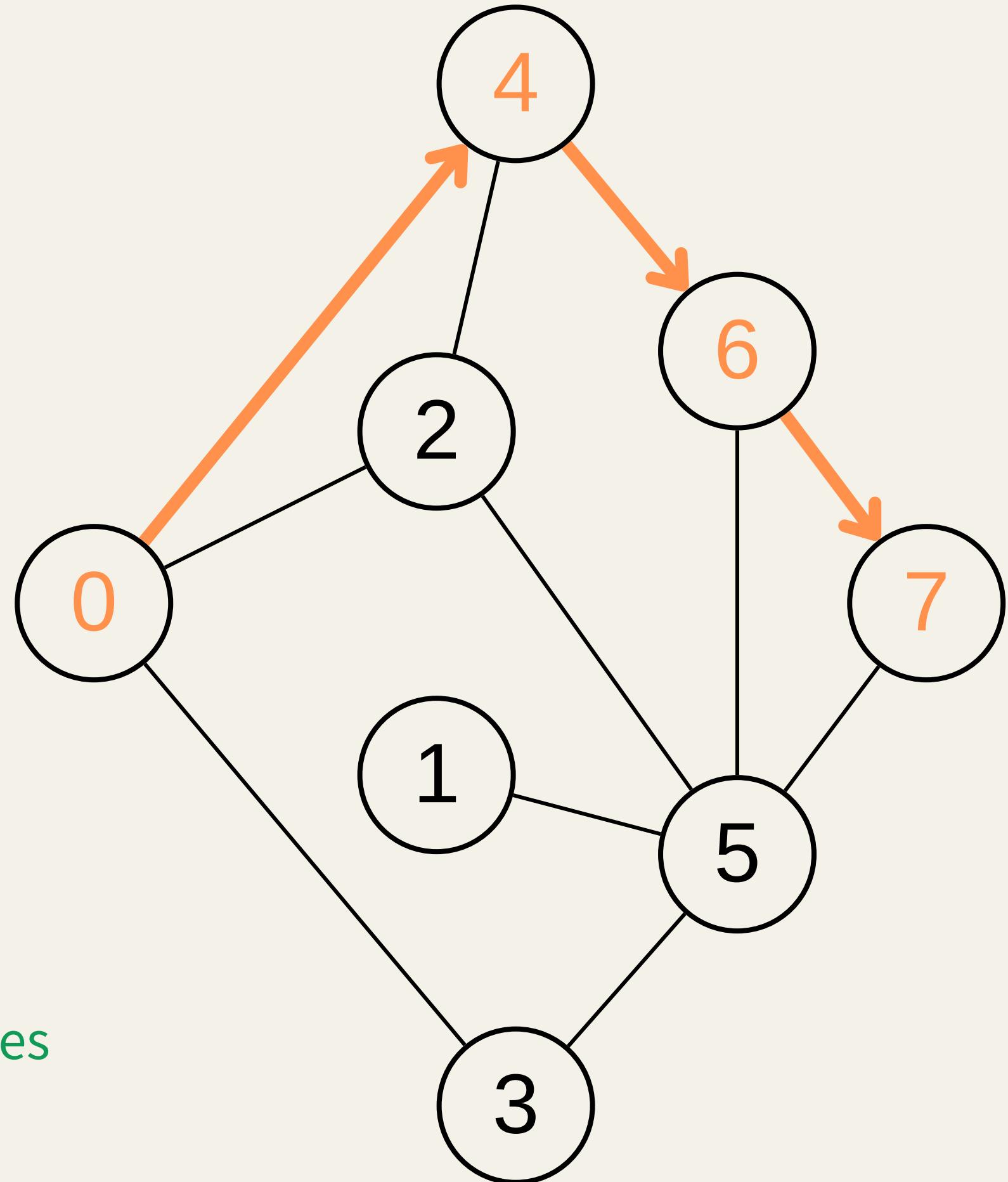
cnt

7

dis

3

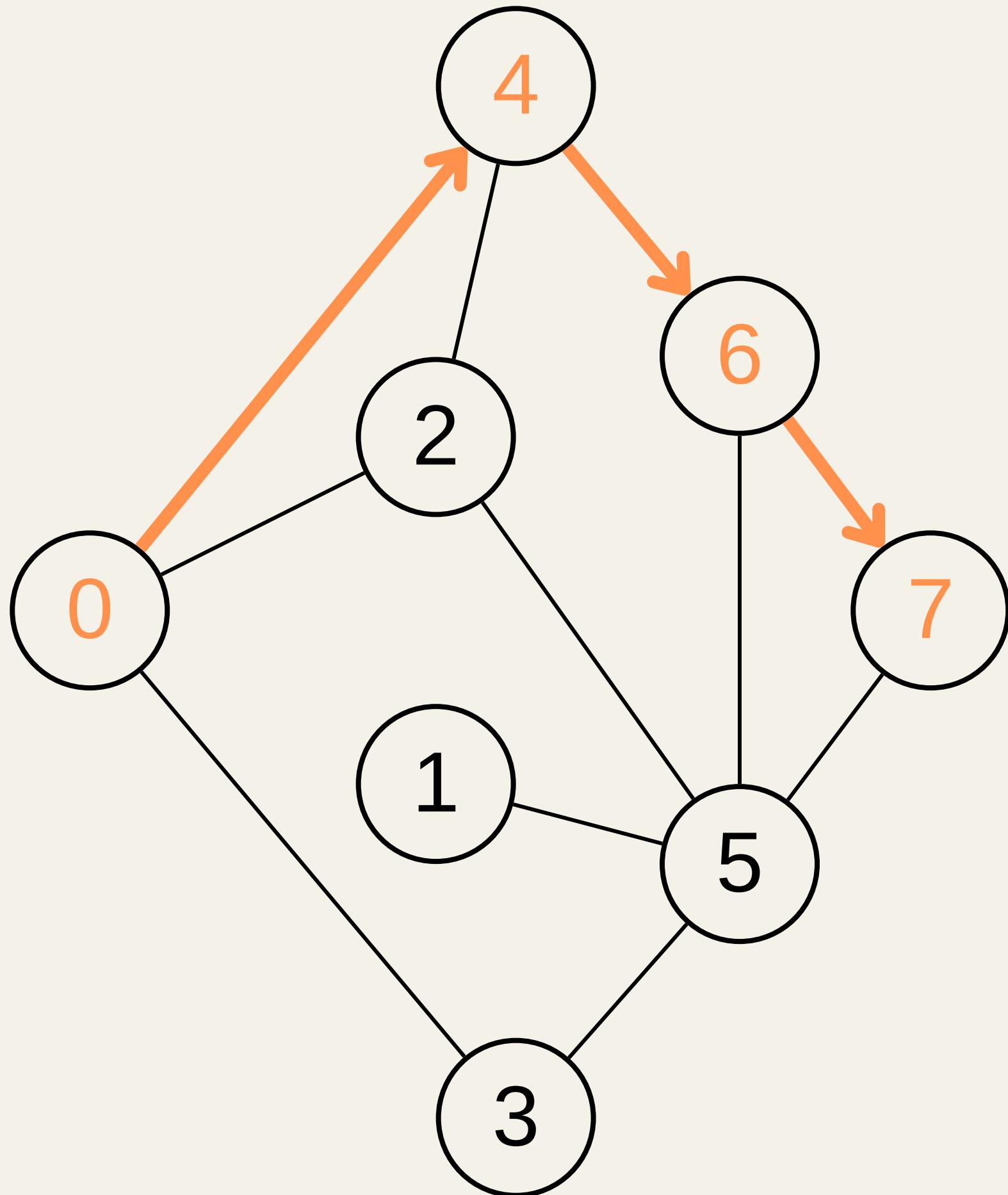
- ② If cnt vertex hadn't been used
→ Update cnt and dis variables



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	True	True
res	3							
cnt		7						
dis			3					

If cnt equals to the ending vertex
→ Update res value through $\min(\text{res}, \text{dis})$



Operating DFS

index	0	1	2	3	4	5	6	7
-------	---	---	---	---	---	---	---	---

used[]	True	False	False	False	True	False	True	False
---------	------	-------	-------	-------	------	-------	------	-------

- ① If cnt has no other adjacent vertices that haven't been visited → Trace back

res

cnt

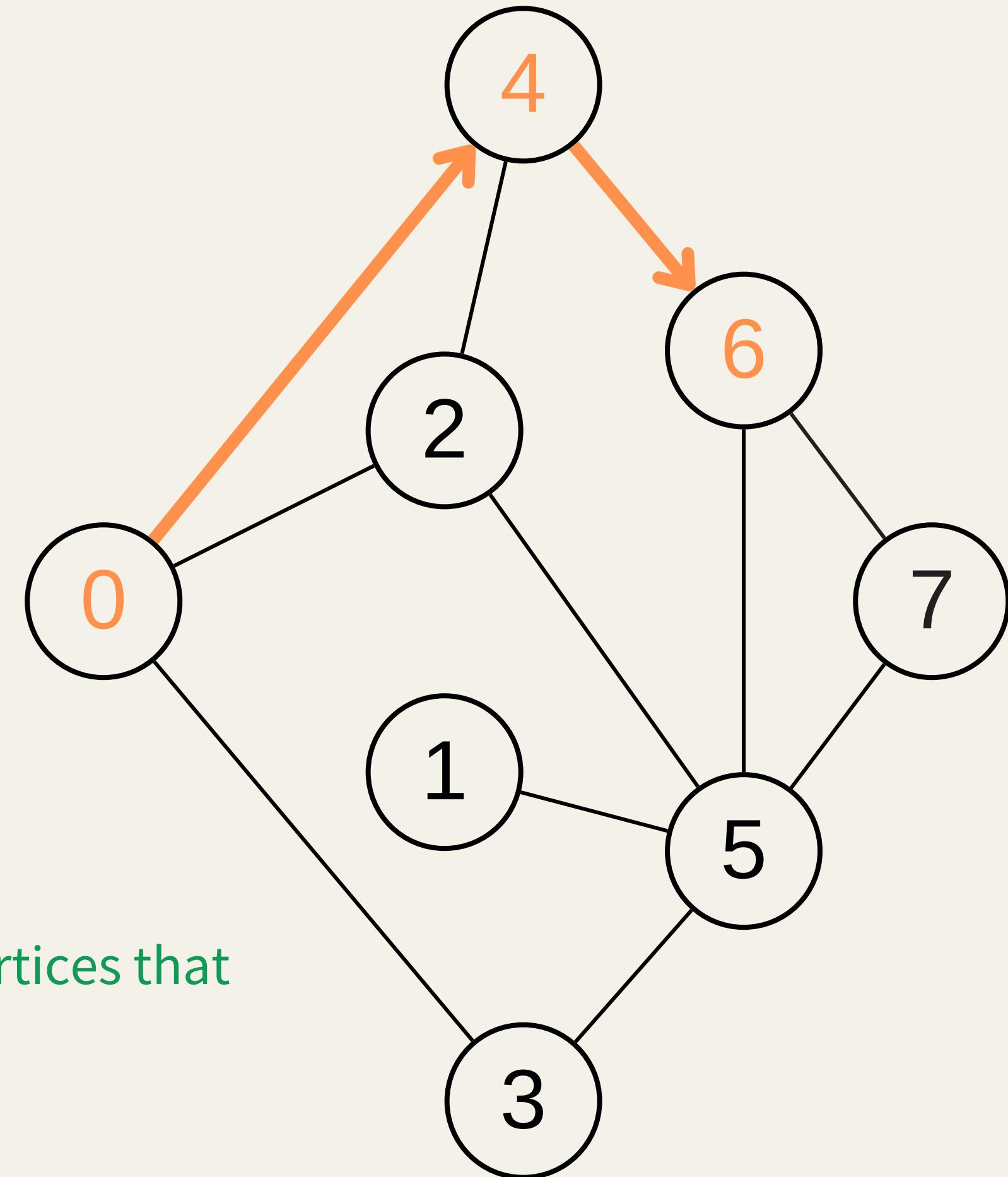
dis

3

6

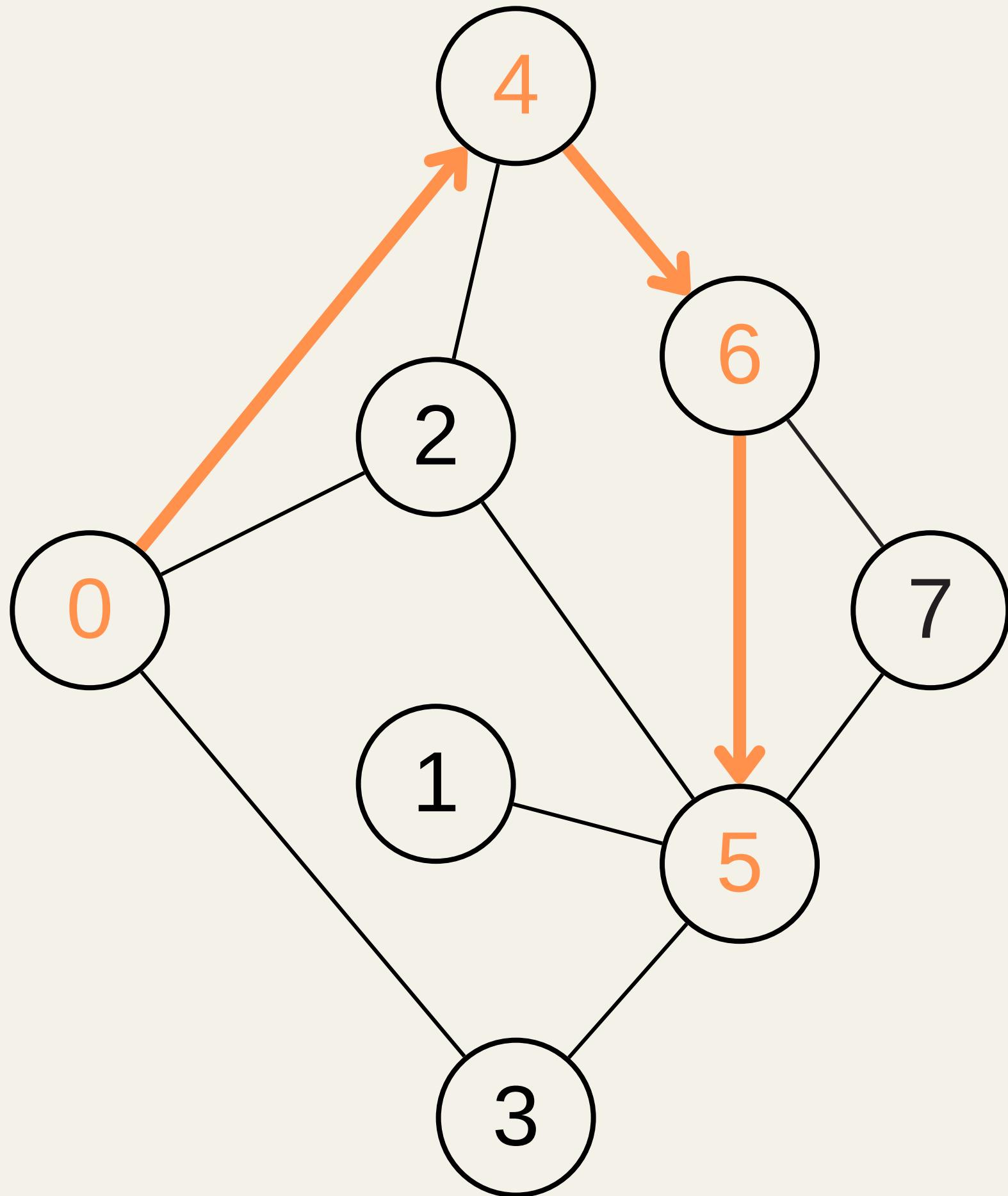
2

- ② If cnt has no other adjacent vertices that haven't been visited
→ Trace back cnt and dis



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	True	False
	Available ✓							
Keep checking the statement of adjacent vertex								
res	3							
cnt		6						
dis			2					



Operating DFS

index	0	1	2	3	4	5	6	7
-------	---	---	---	---	---	---	---	---

used[]	True	False	False	False	True	True	True	False
---------	------	-------	-------	-------	------	------	------	-------

- ① If cnt vertex hadn't been used
→ Update its statement

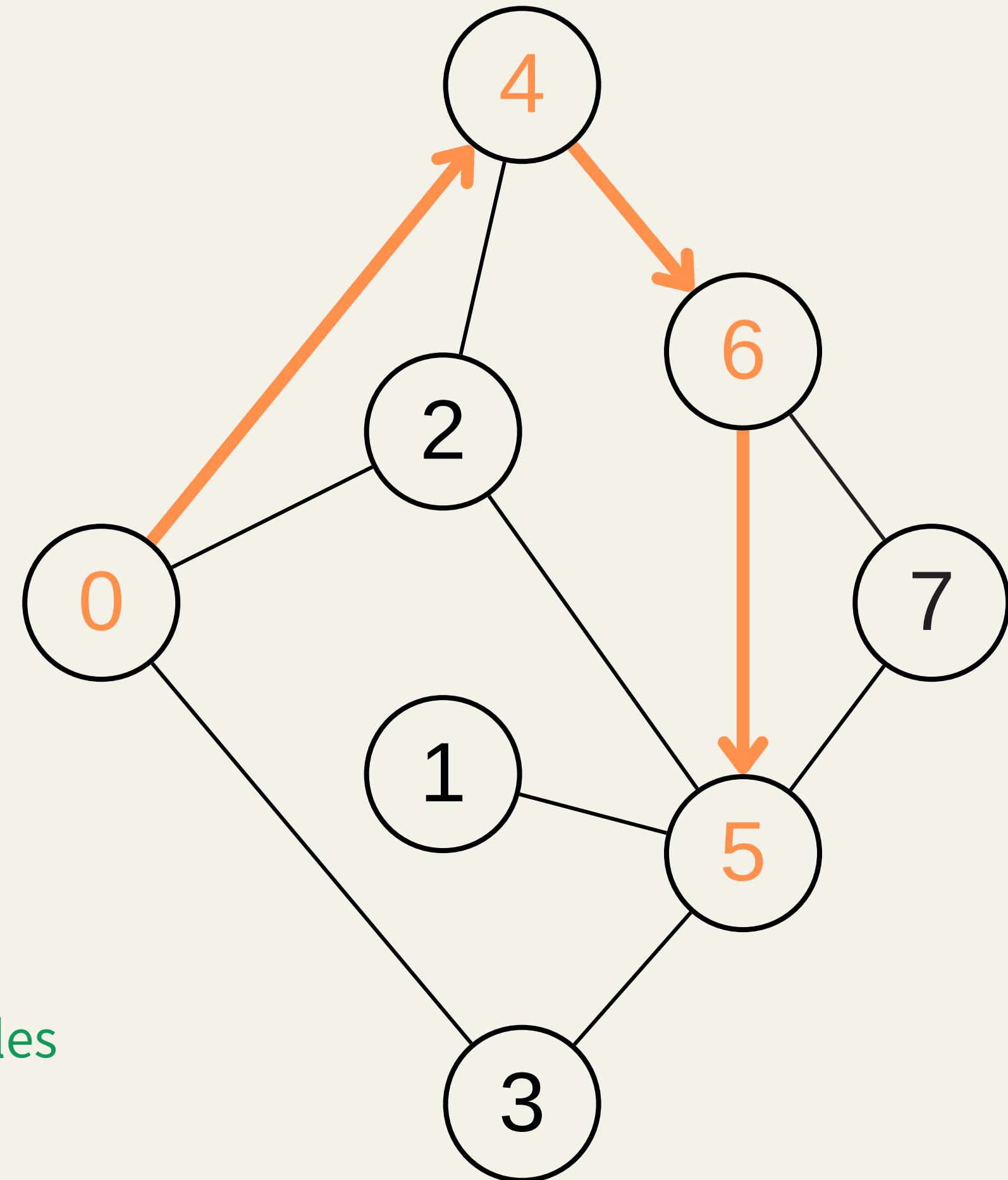
res

cnt

dis

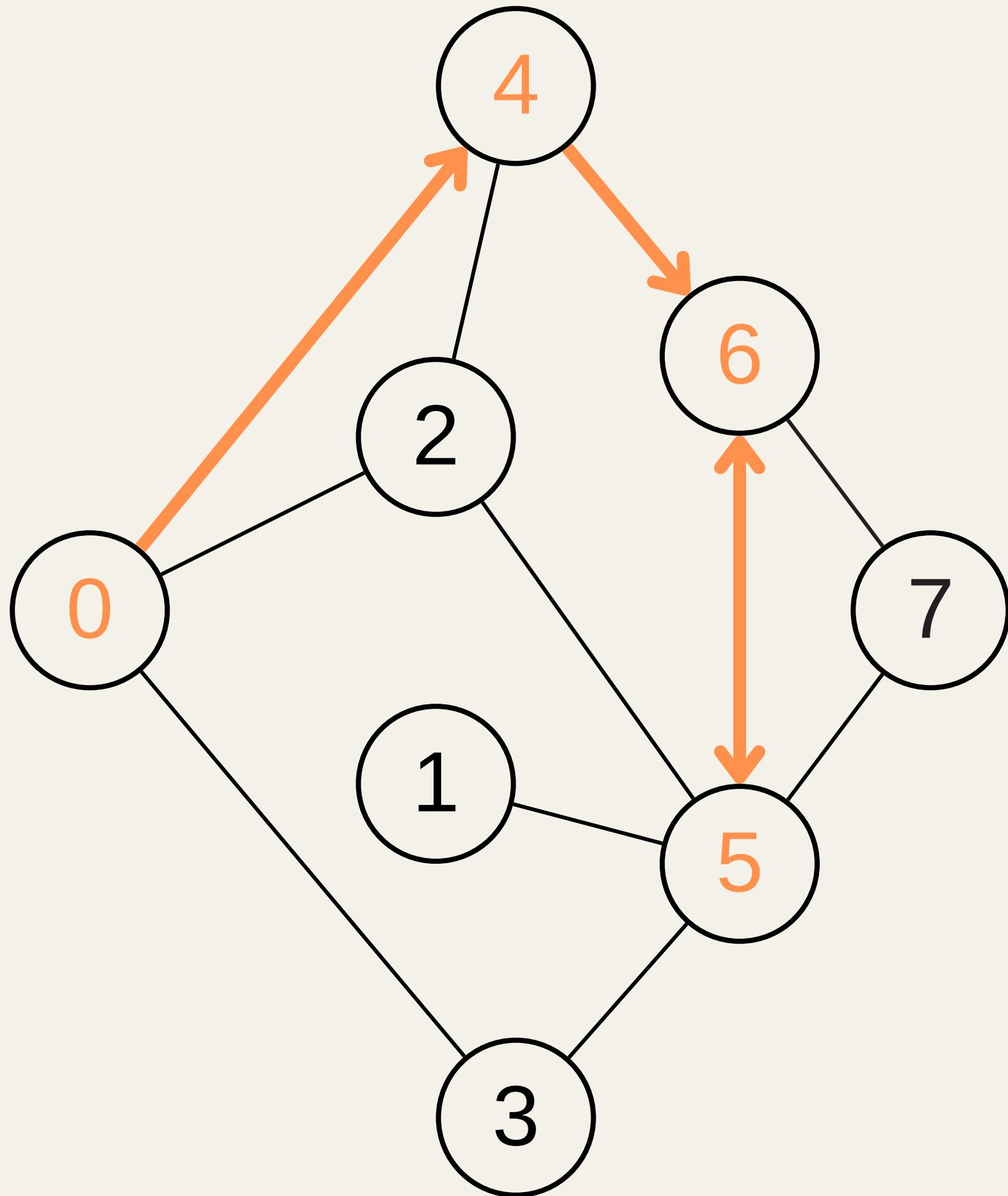
3

② If cnt vertex hadn't been used
→ Update cnt and dis variables



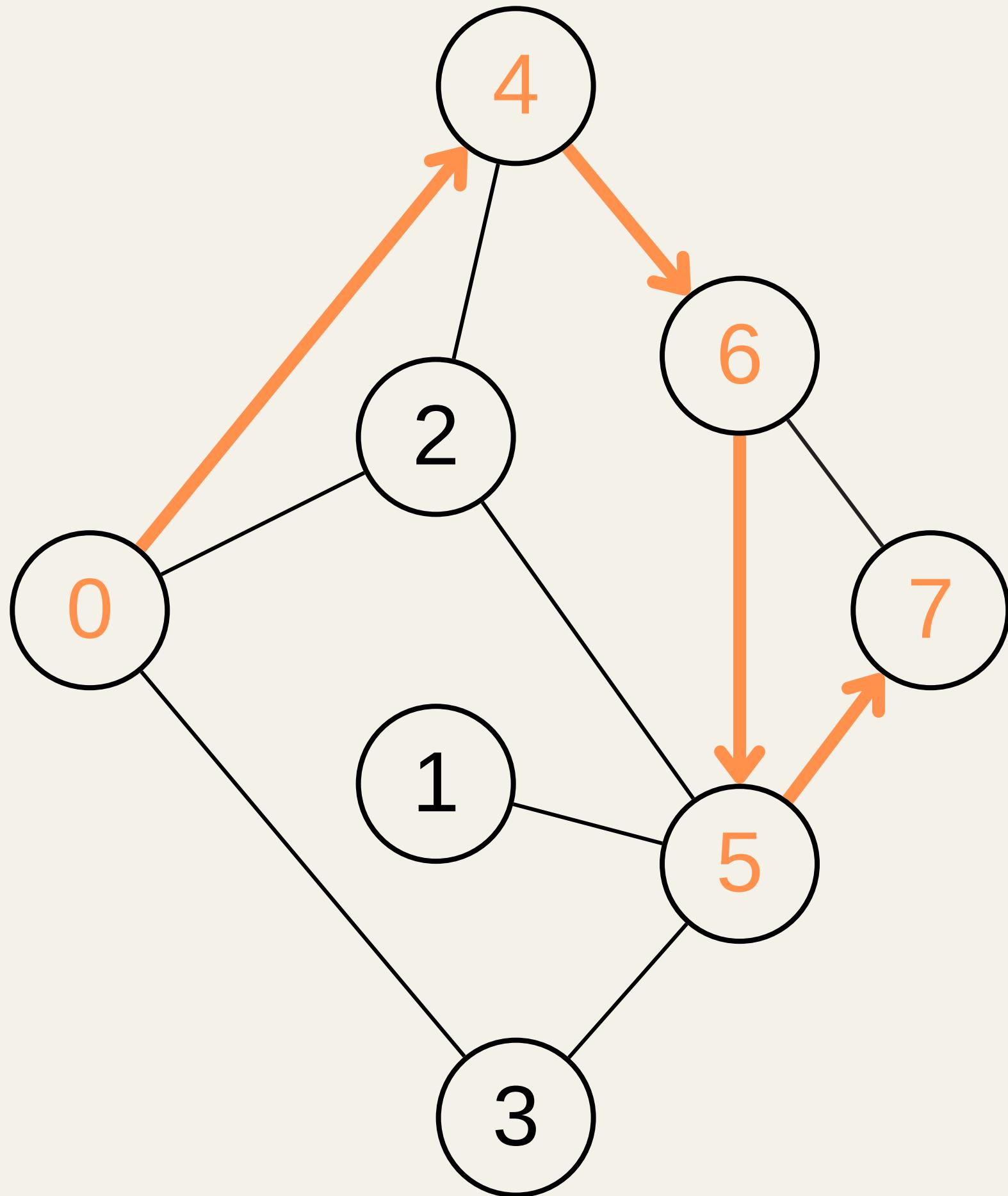
Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	True	True	False
Not Available ✗								
Check whether adjacent vertex already been used								
res	3							
cnt		5						
dis			3					



Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	True	True	False
	Available ✓							
Check whether adjacent vertex already been used								
res	3	cnt	7	dis	4			



Operating DFS

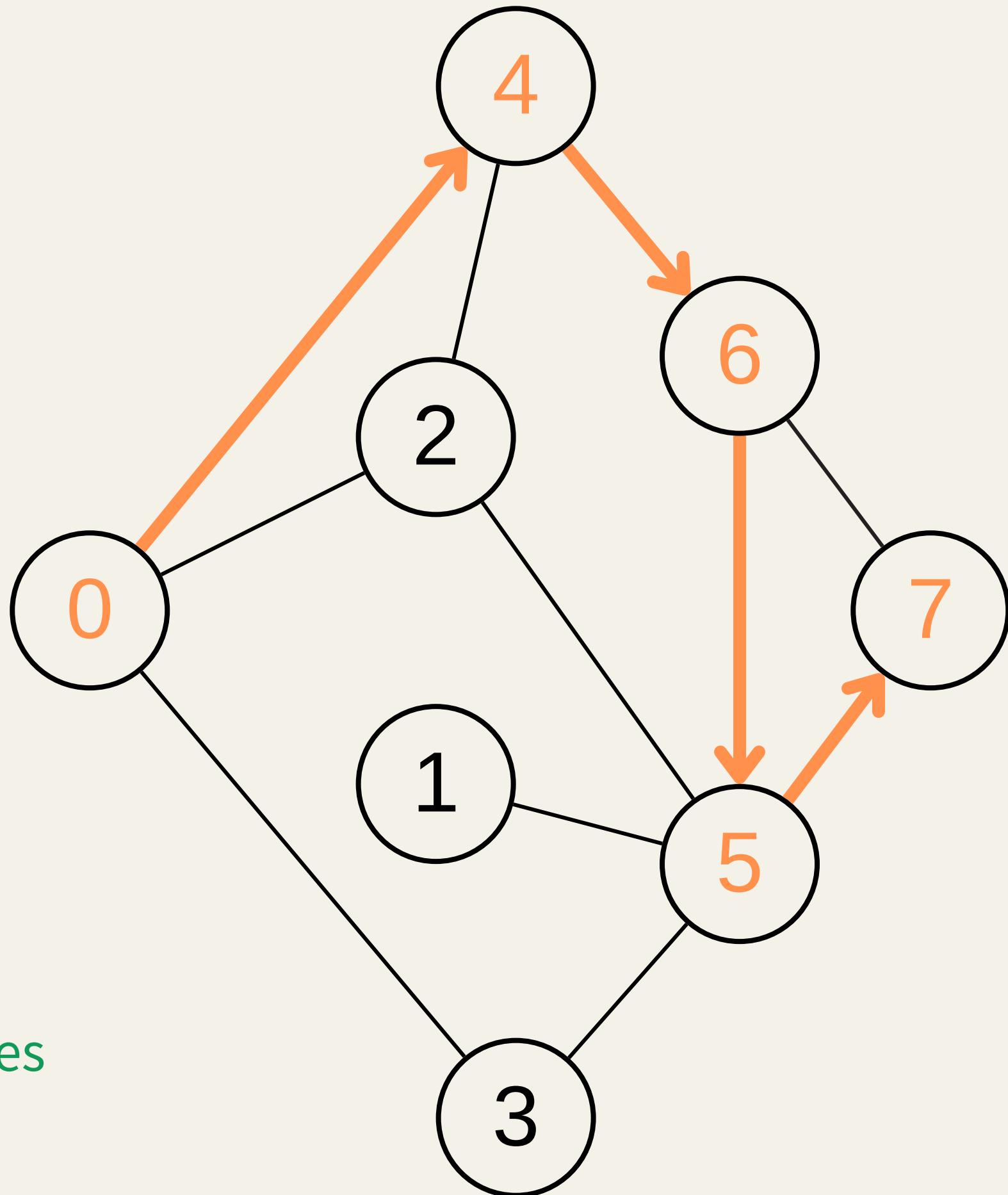
index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	True	True	True
res	3							

cnt dis

cnt	7
dis	4

- ① If cnt vertex hadn't been used
→ Update its statement

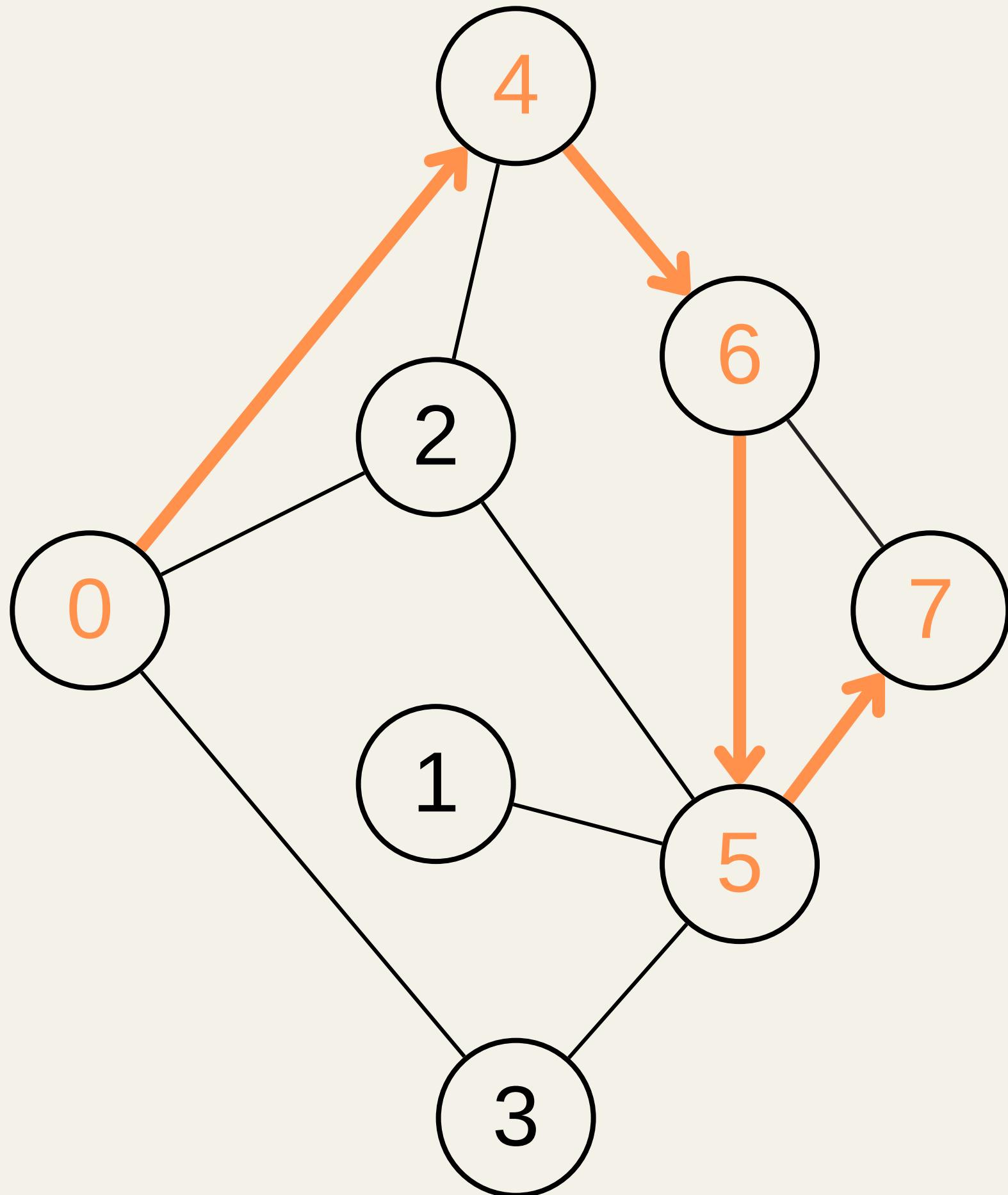
- ② If cnt vertex hadn't been used
→ Update cnt and dis variables



Operating DFS

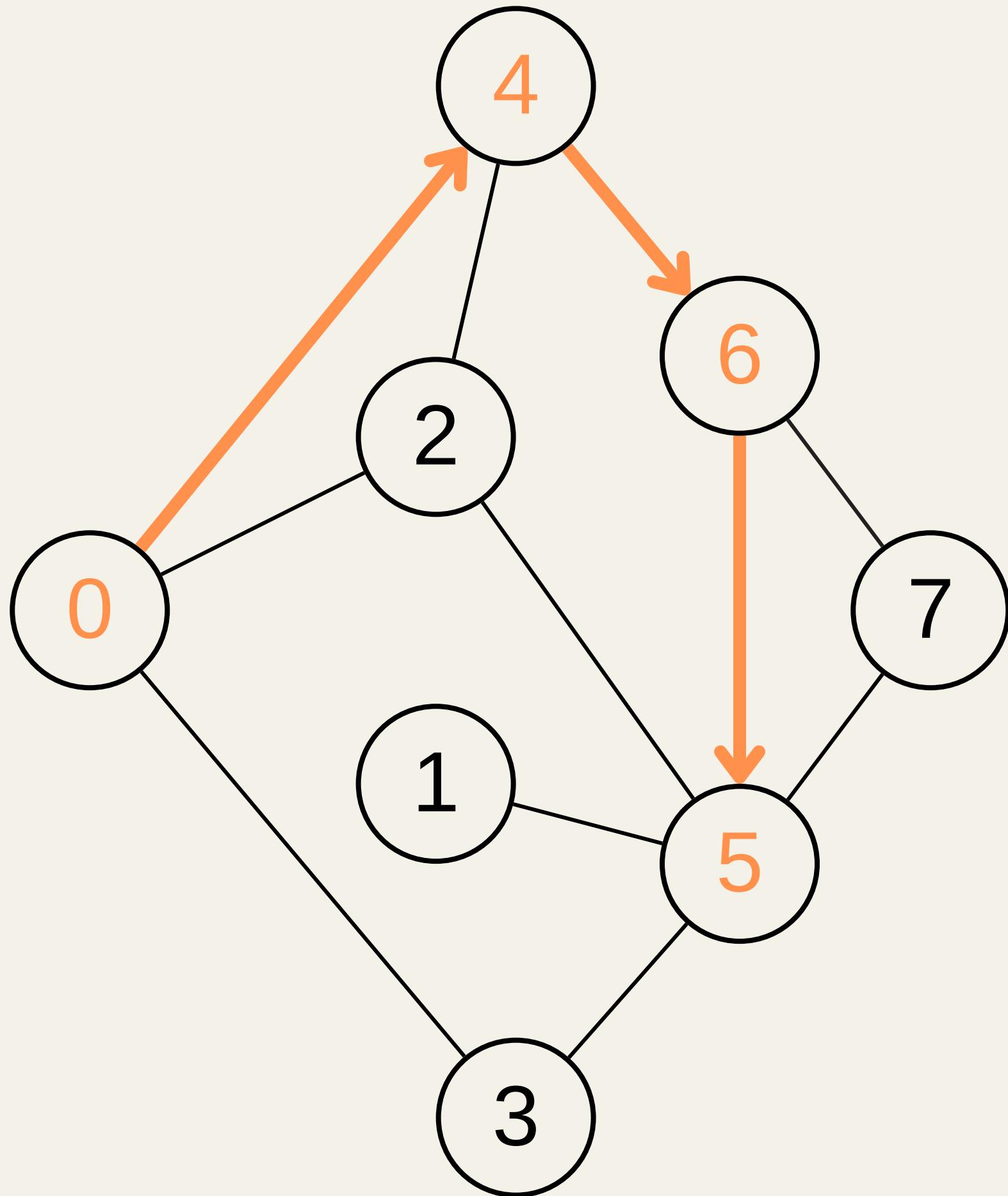
index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	True	True	True
res	3							
cnt		7						
dis			4					

If cnt equals to the ending vertex
→ Update res value through $\min(\text{res}, \text{dis})$



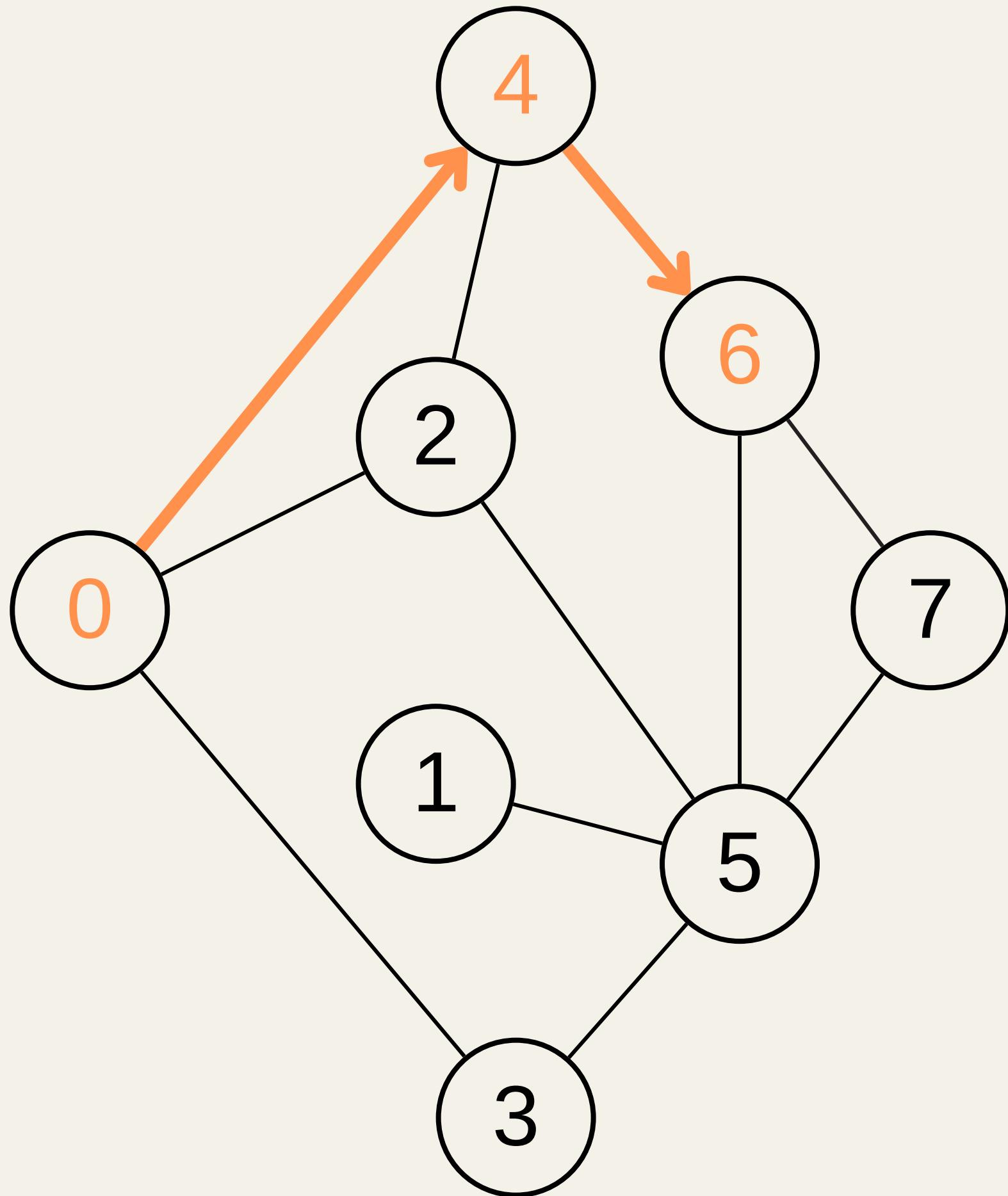
Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	True	True	False
res	3							
cnt		7						
dis			4					



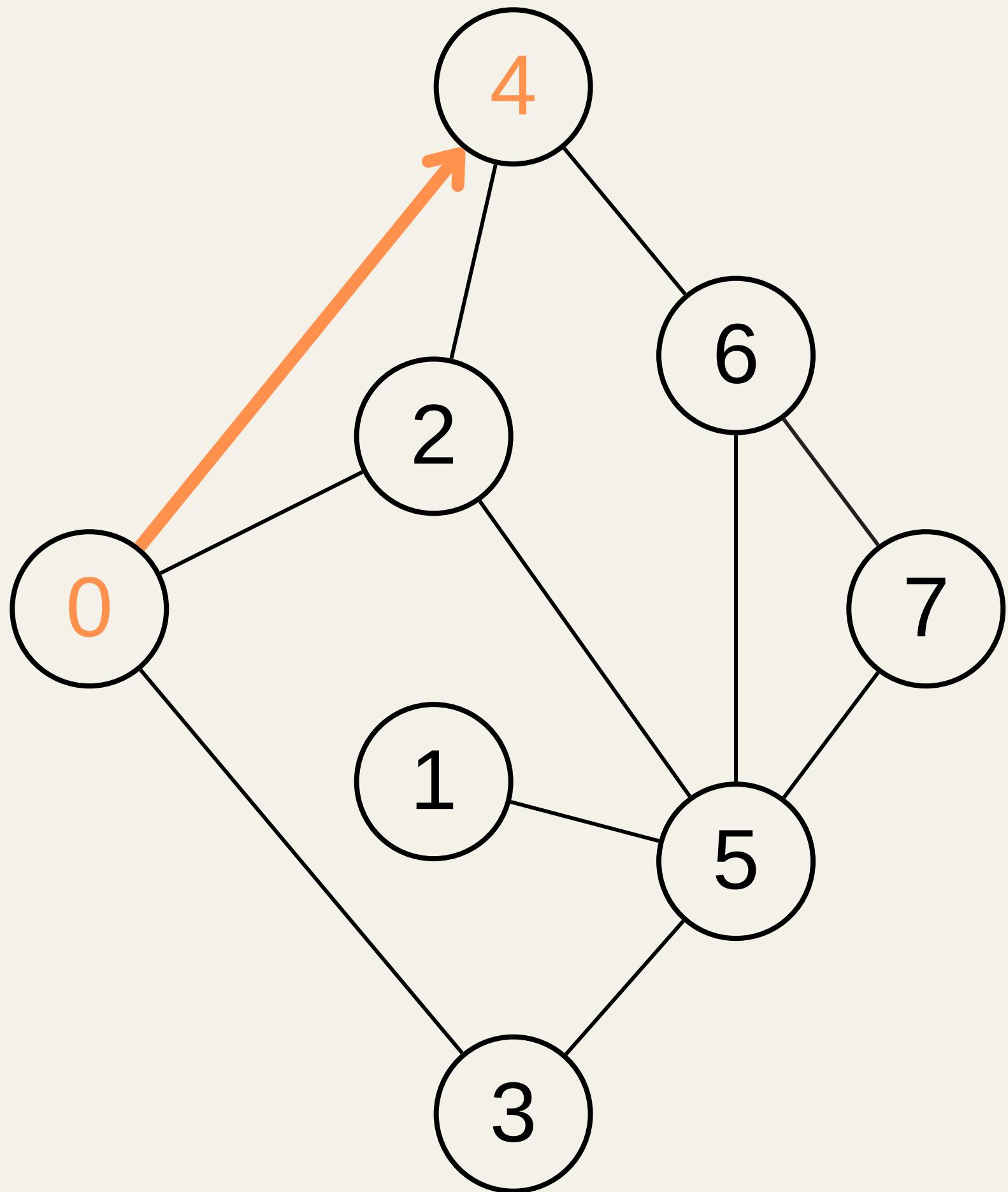
Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	True	False
res	3							
cnt		7						
dis			4					



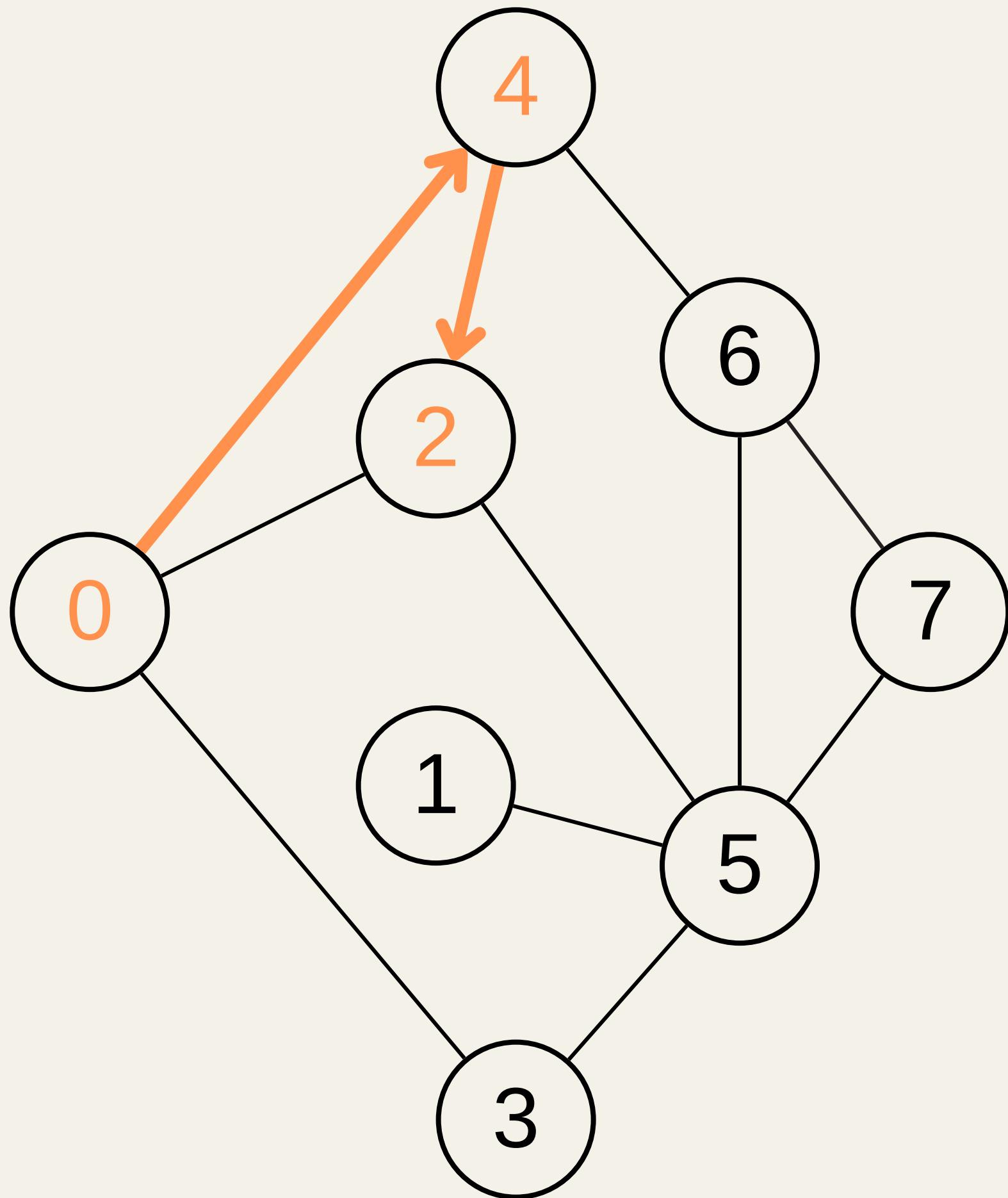
Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	False	False
res	3							
cnt		7						
dis			4					



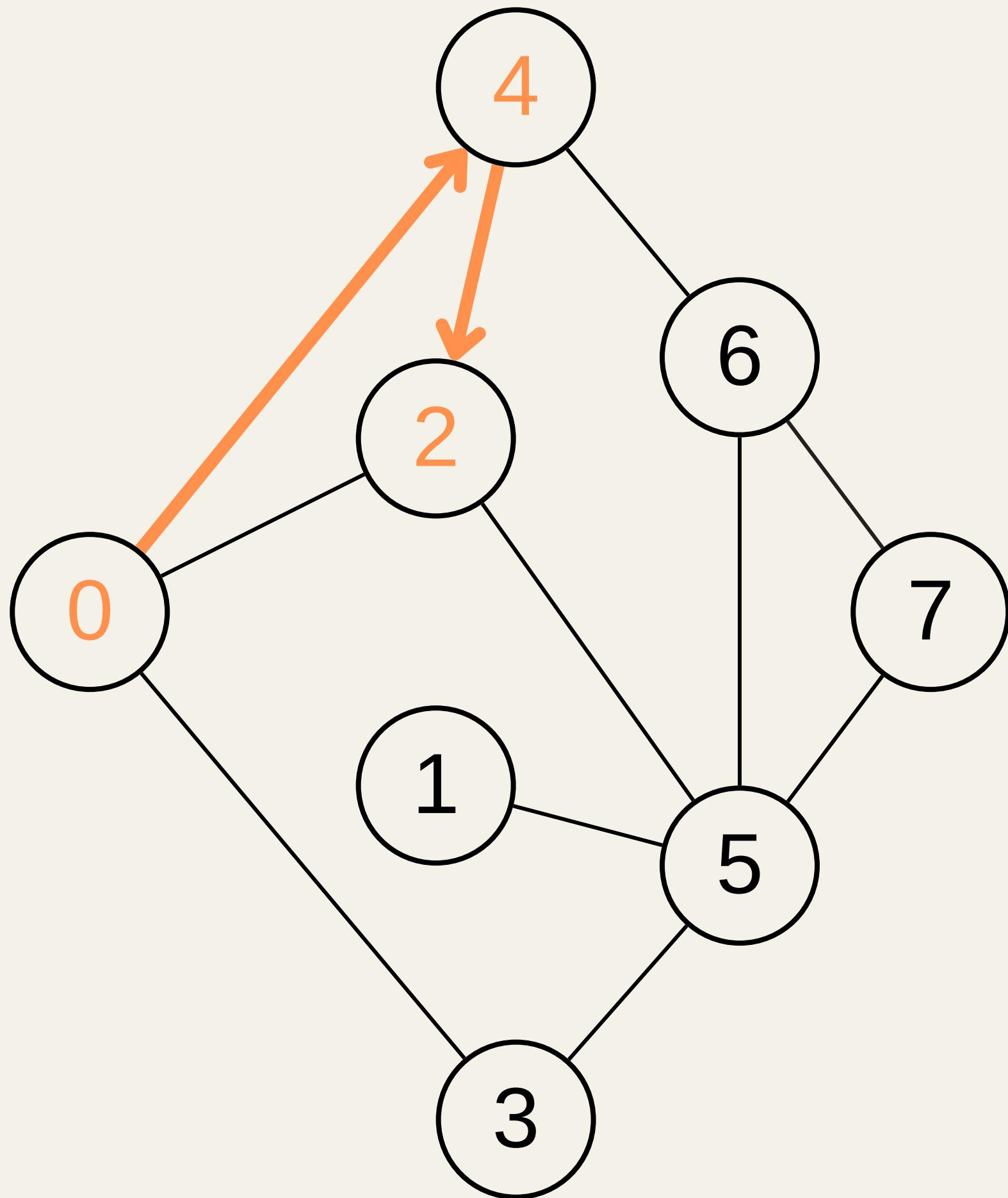
Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	False	False	True	False	False	False
	Available ✓							
res	3							
cnt		7						
dis			4					



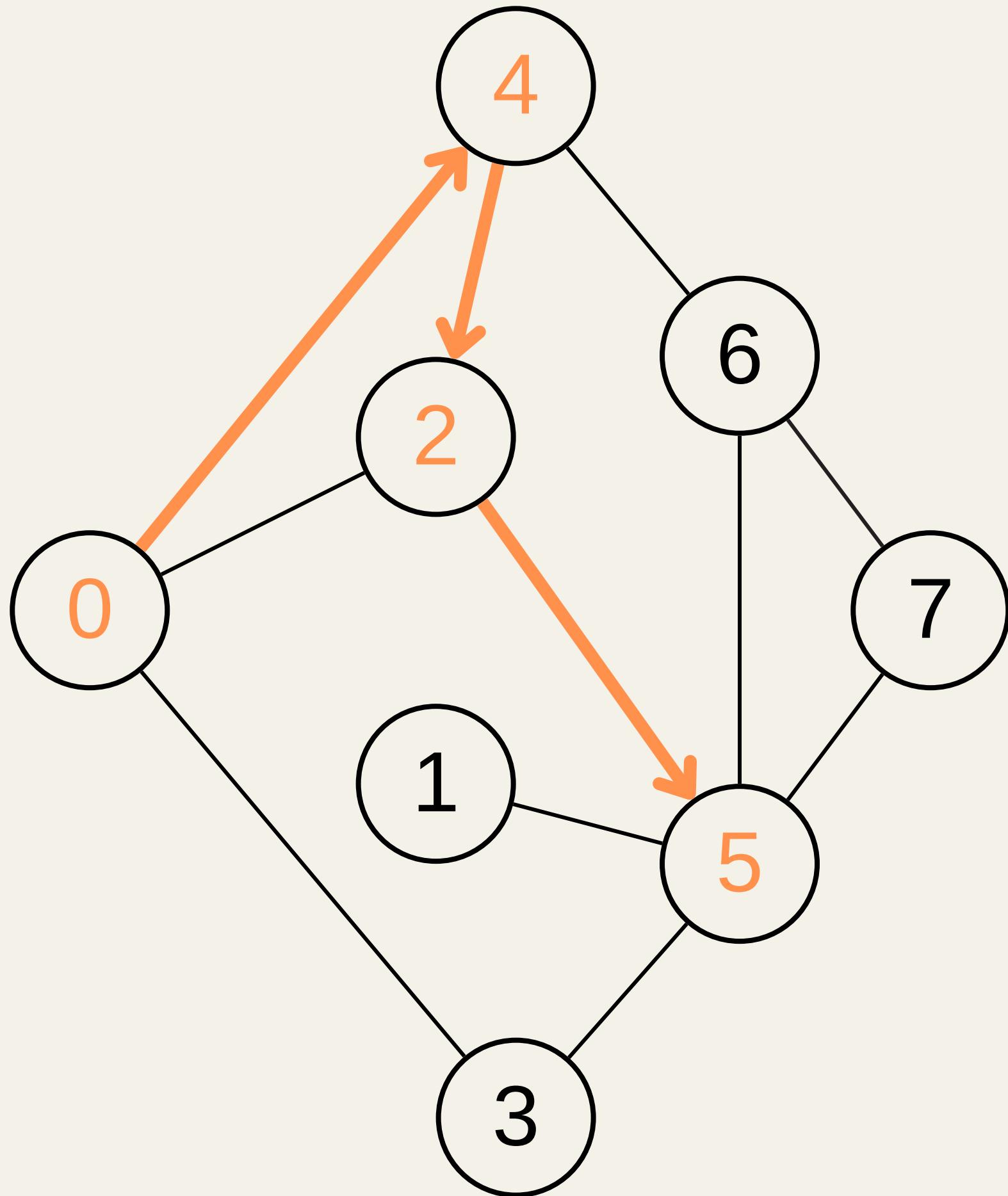
Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	False	False	False
res	3							
cnt		2						
dis			2					



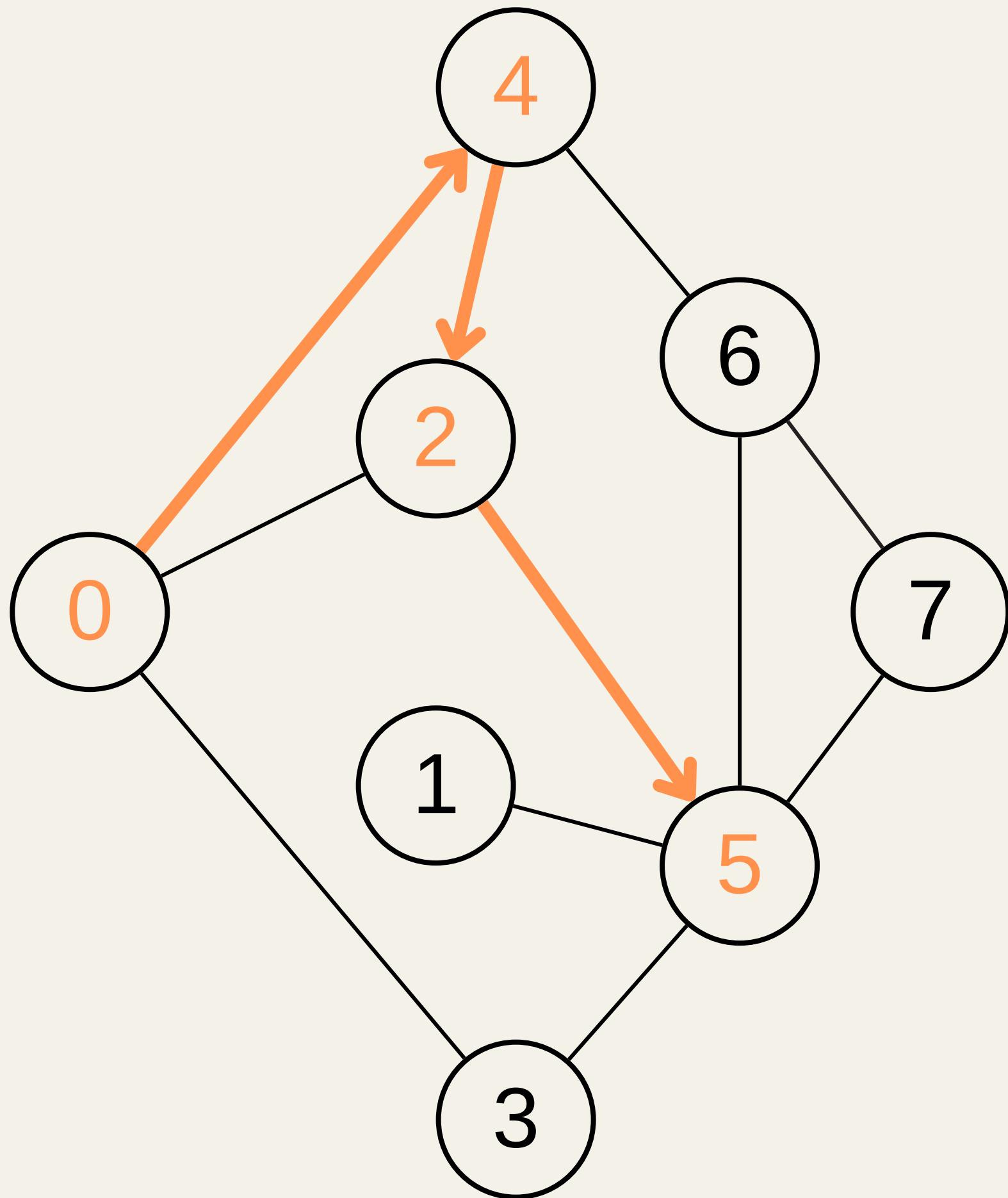
Operating DFS

index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	False	False	False
						Available ✓		
res	3							
cnt		2						
dis			2					



Operating DFS

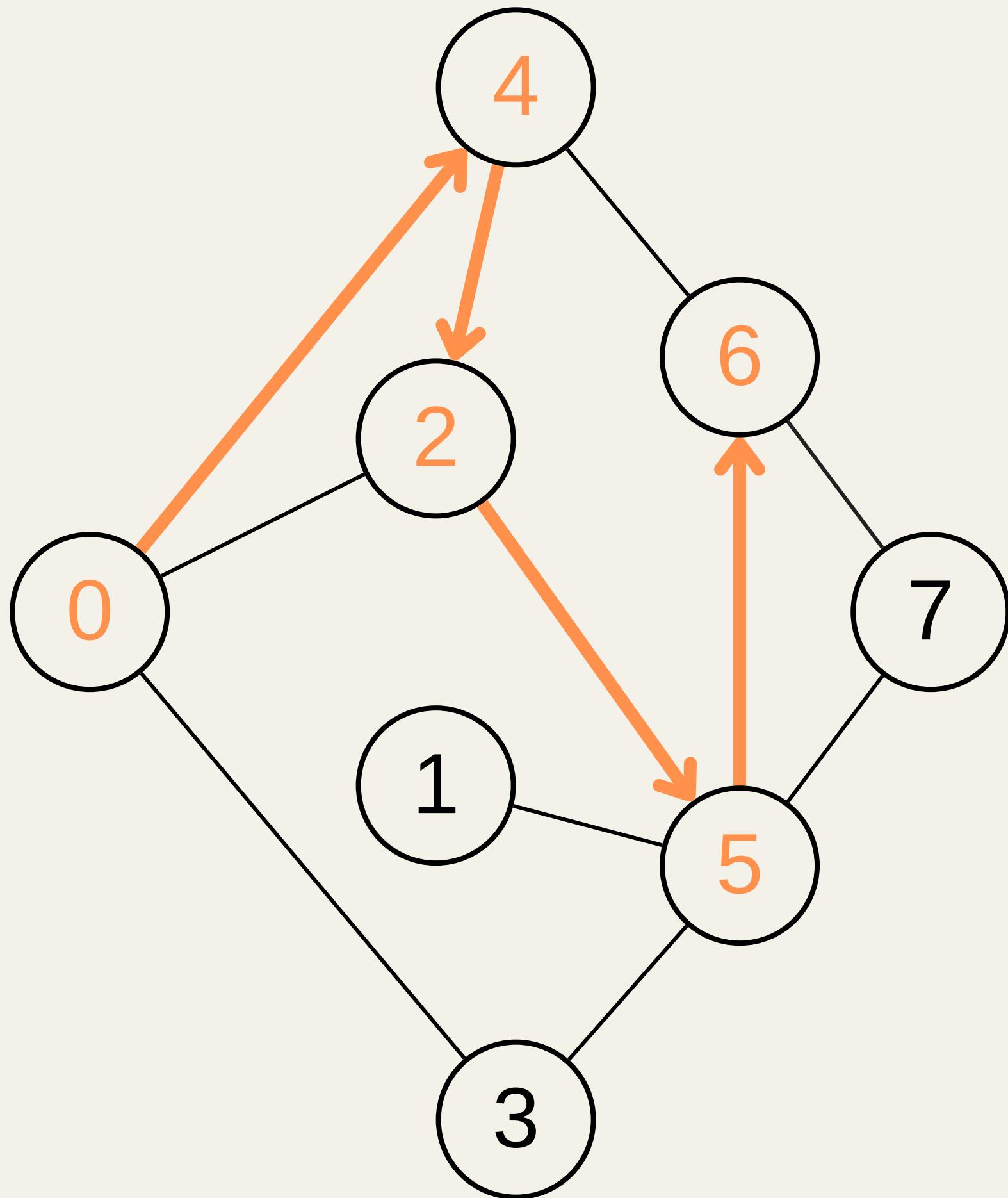
index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	True	False	False
res	3							
cnt		5						
dis			3					



Operating DFS

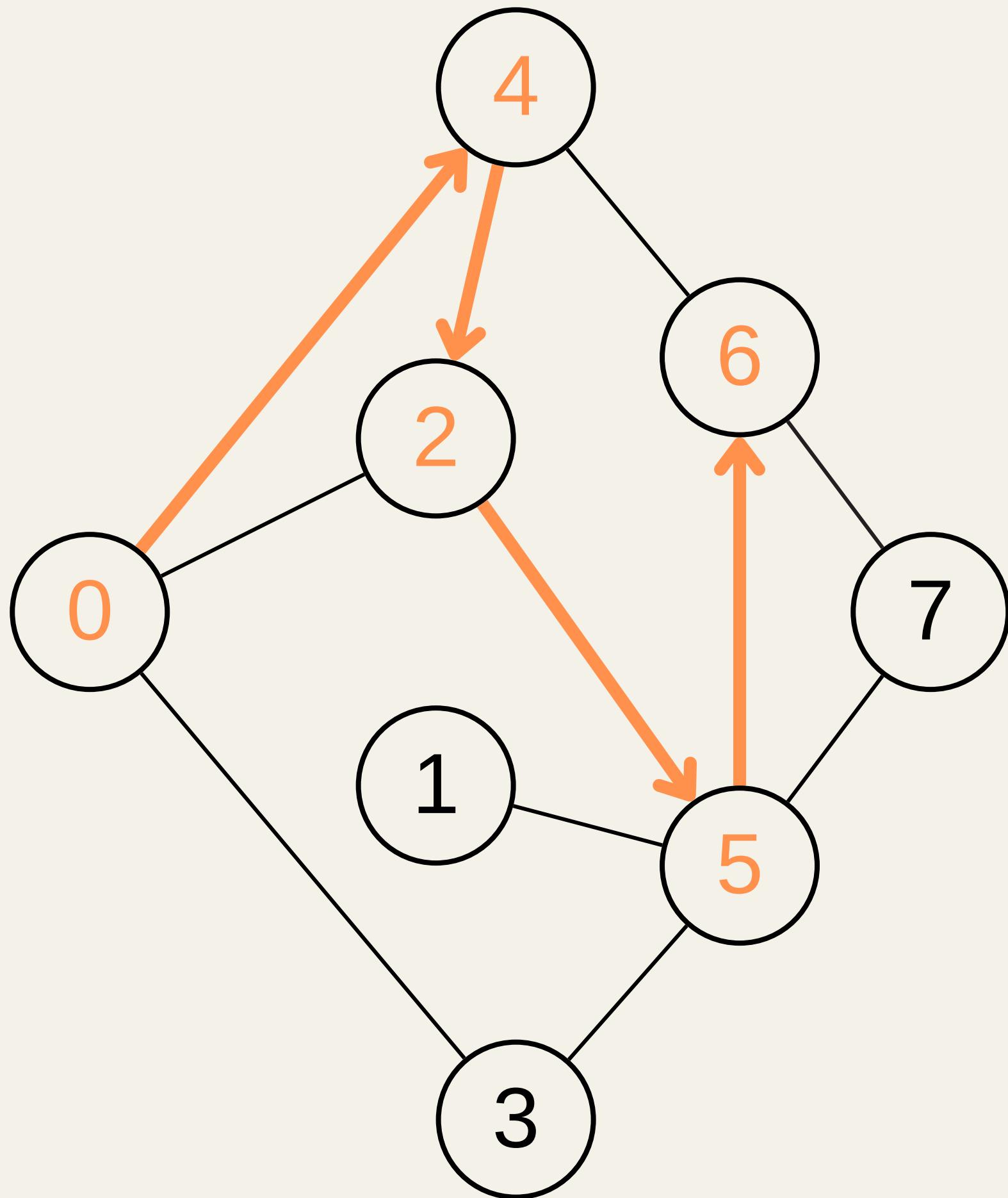
index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	True	False	False
res	3							
cnt								
dis								

Available ✓



Operating DFS

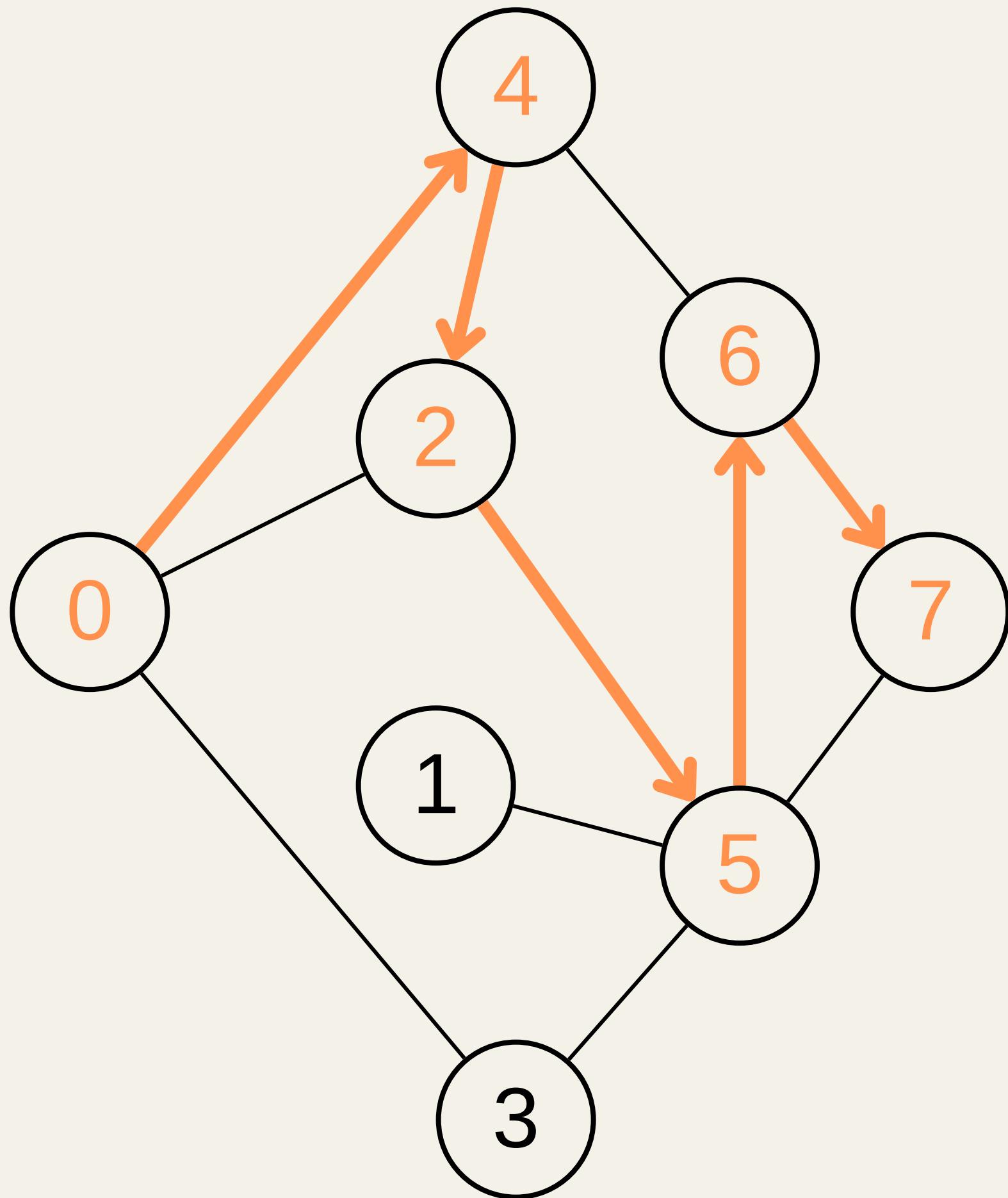
index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	True	True	False
res	3							
cnt		6						
dis			4					



Operating DFS

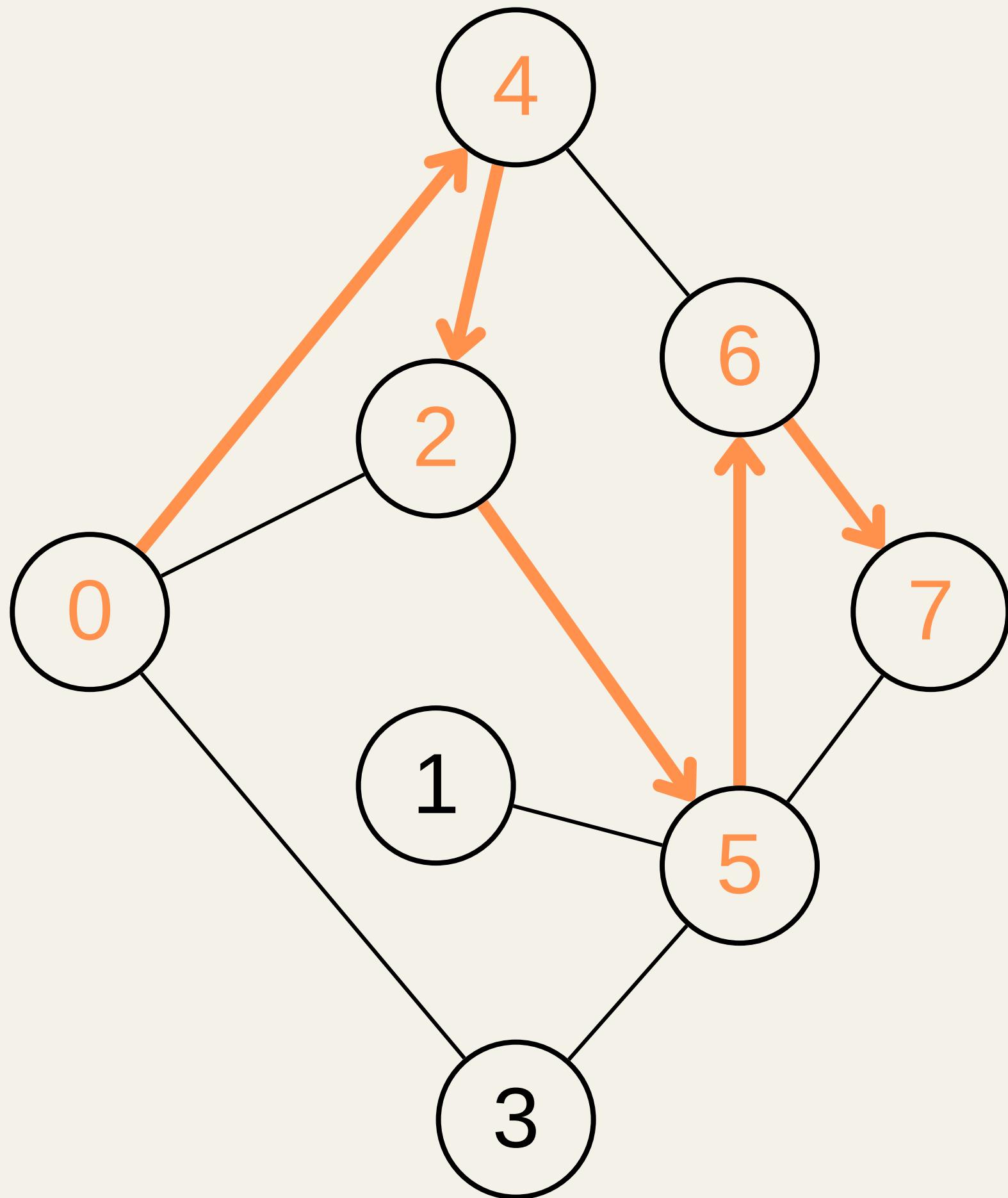
index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	True	True	False
res	3							
cnt								
dis								

Available ✓



Operating DFS

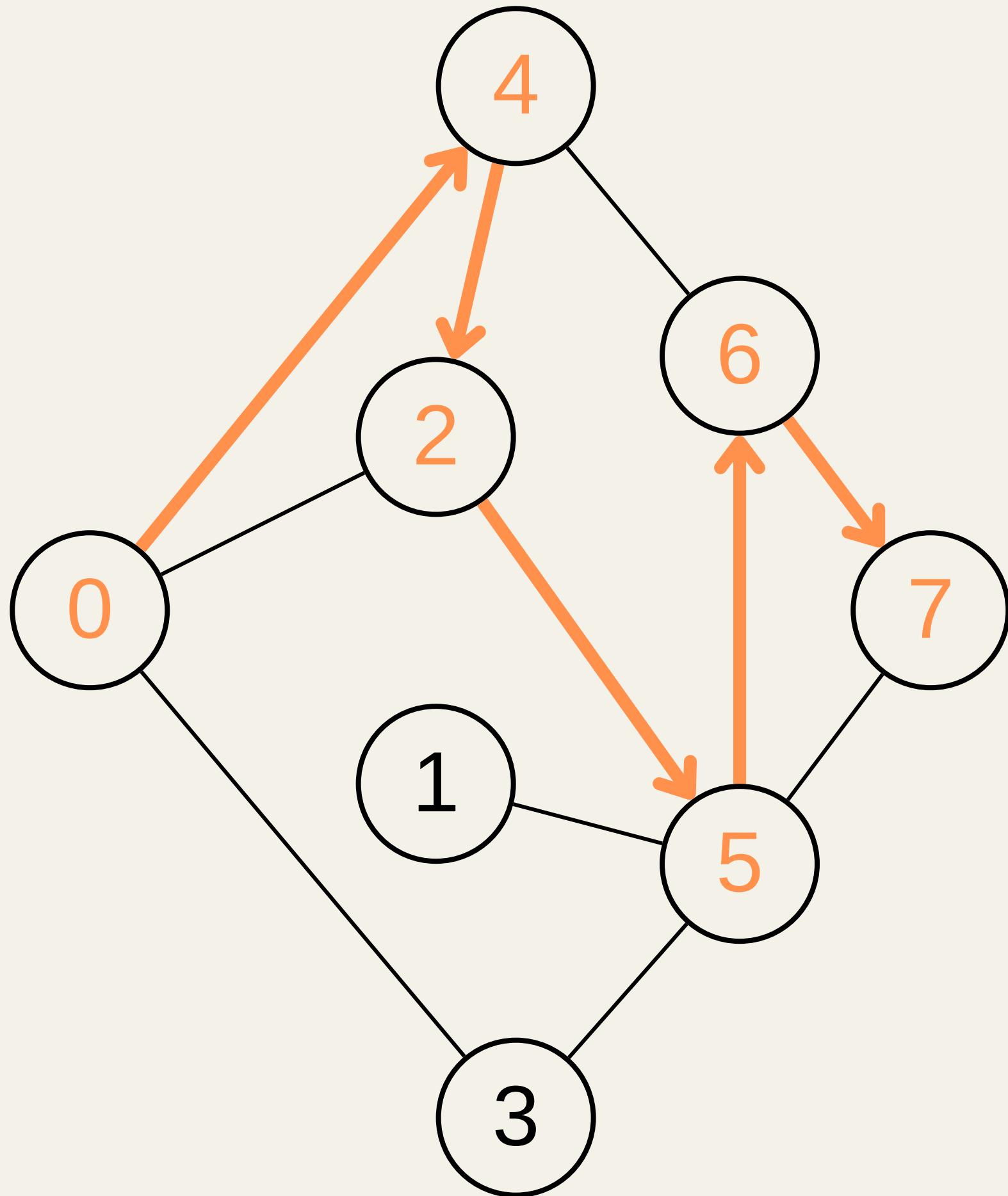
index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	True	True	True
res	3							
cnt		7						
dis			5					



Operating DFS

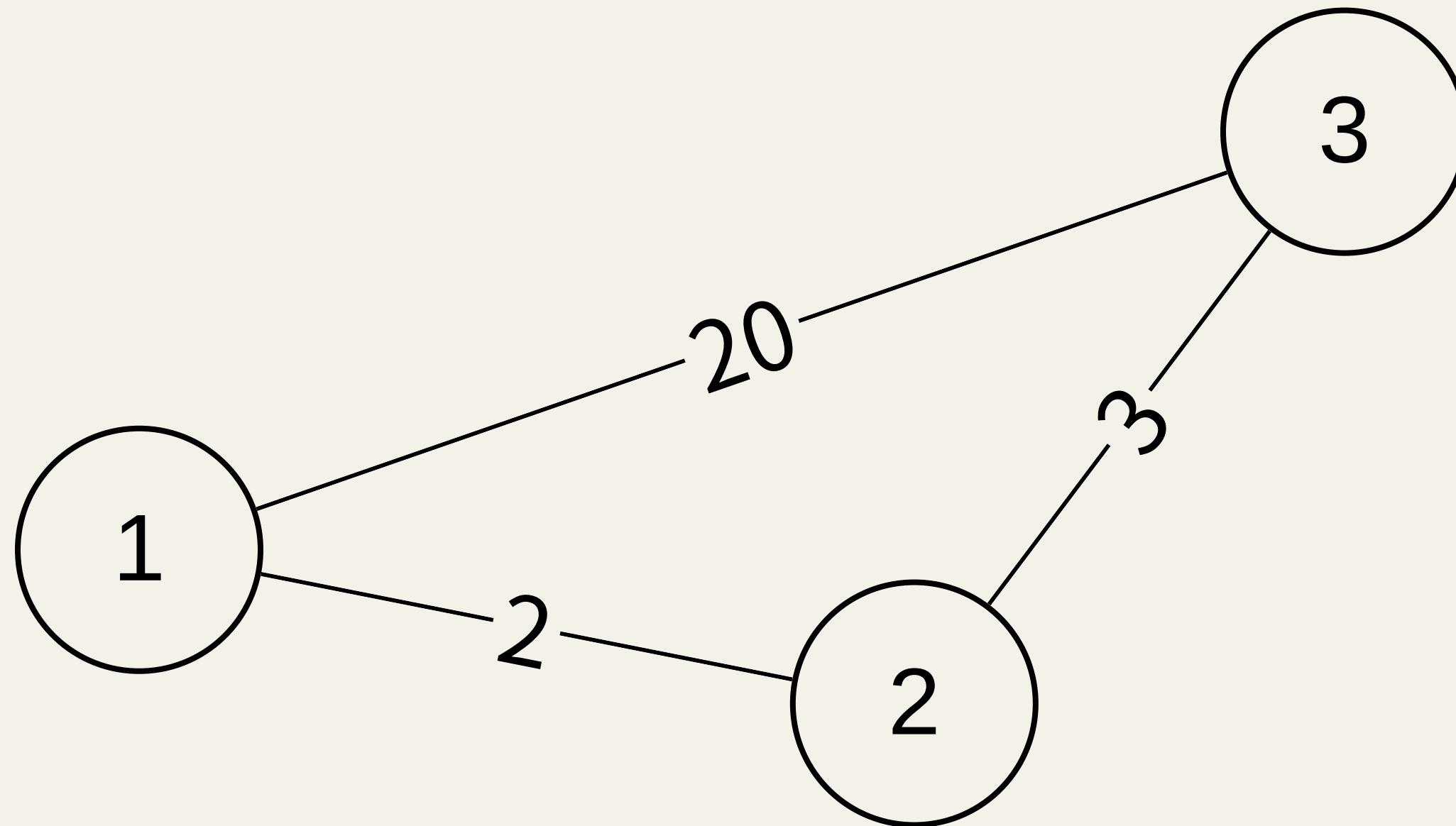
index	0	1	2	3	4	5	6	7
used[]	True	False	True	False	True	True	True	True
res	3							
cnt		7						
dis			5					

If cnt equals to the ending vertex
→ Update res value through $\min(\text{res}, \text{dis})$
This is the shortest path !

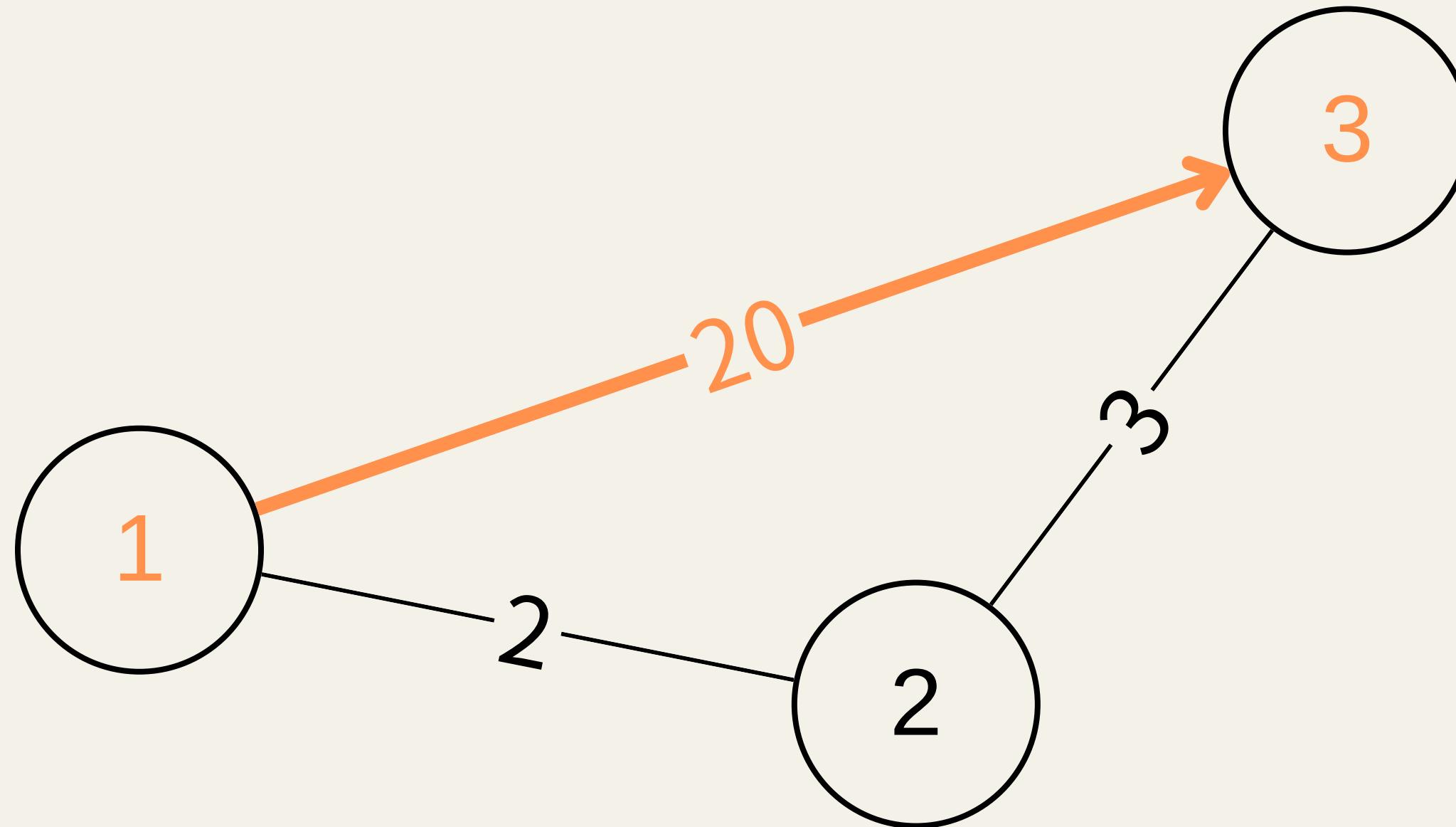




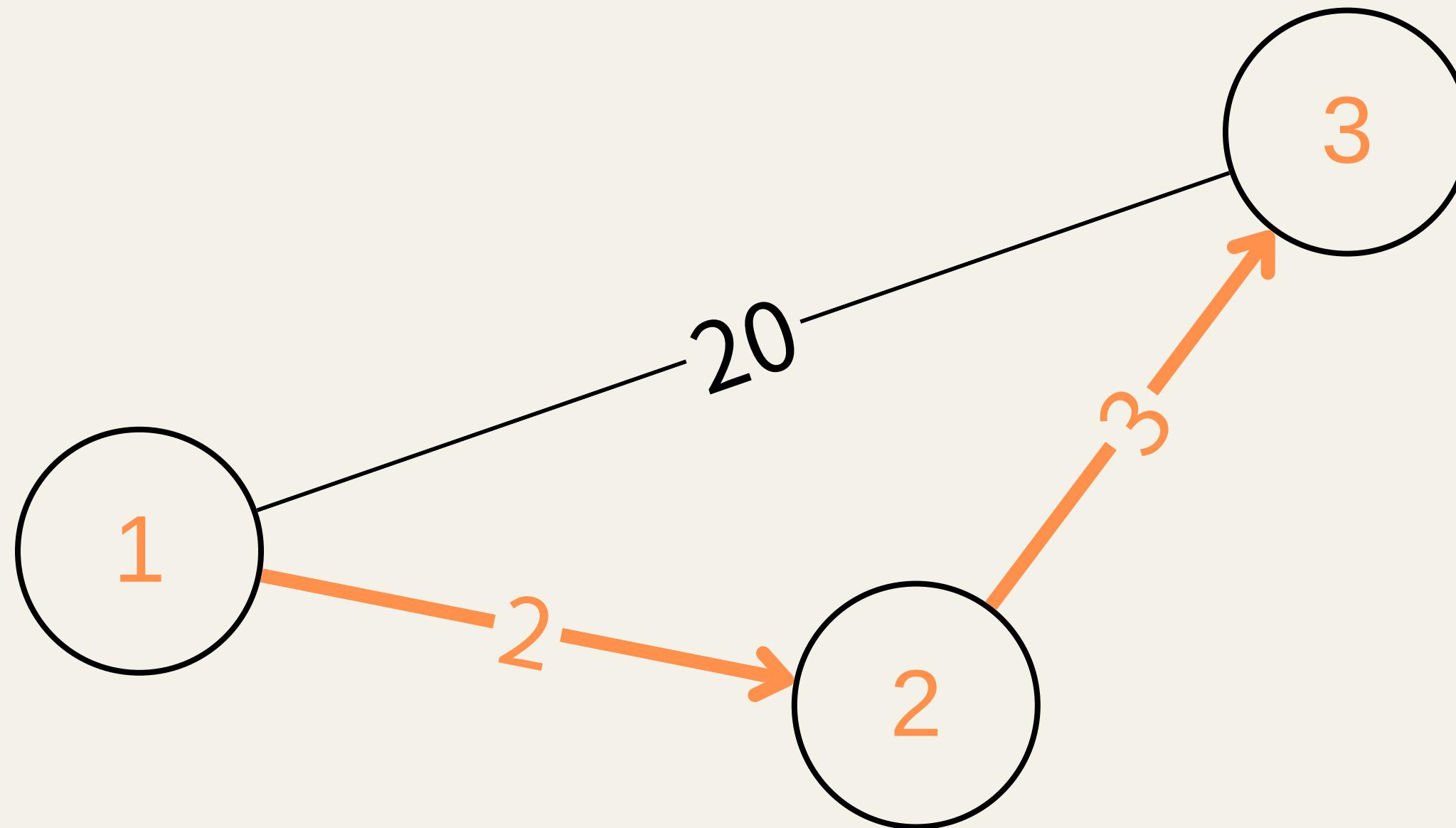
Relax Operation



Relax Operation



Relax Operation



Relax Operation



Initialize

index 0 1 2 3 4 5

dis[]

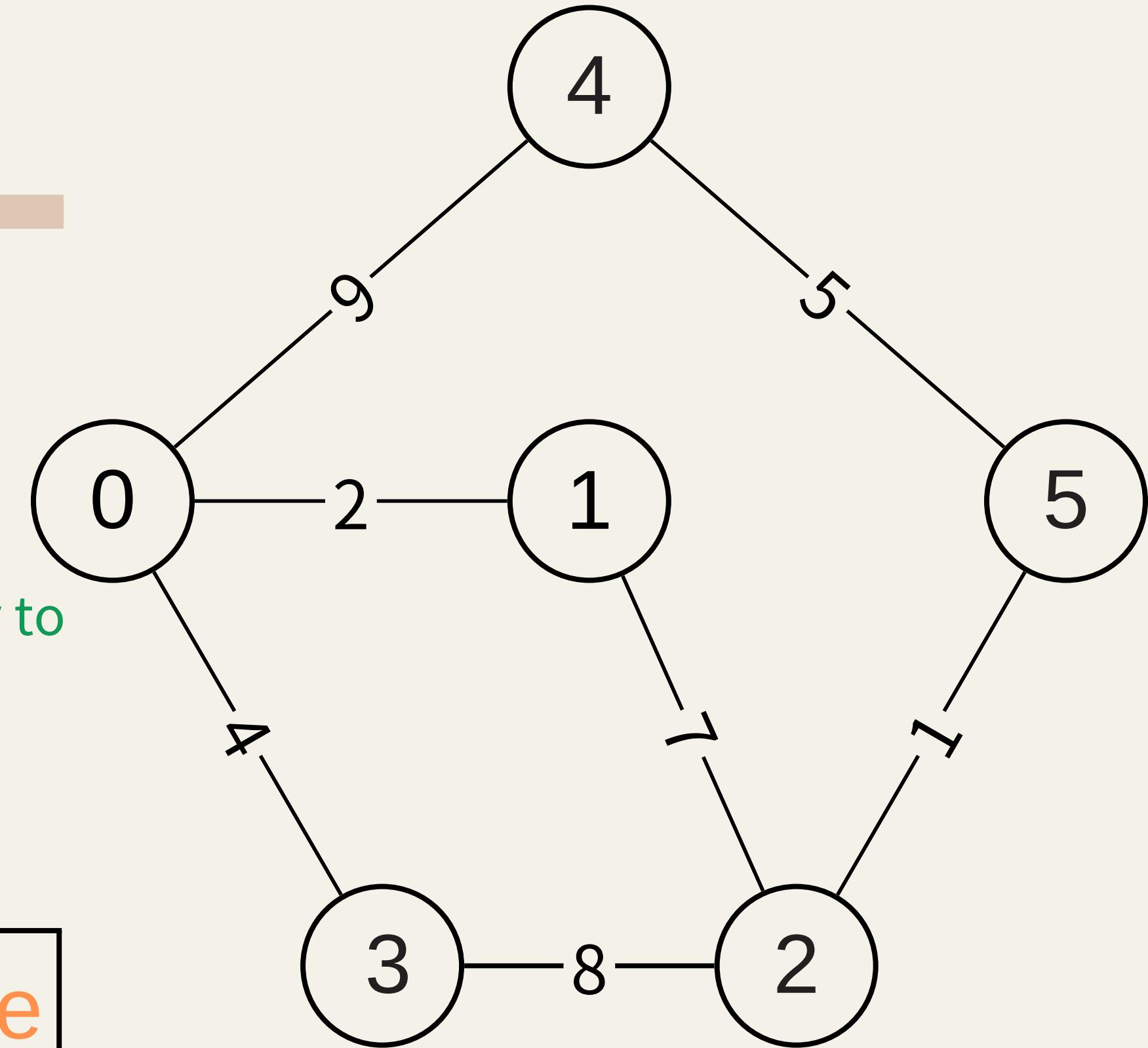
INF	INF	INF	INF	INF	INF
-----	-----	-----	-----	-----	-----

- ① Set all of the value in distance array to default value → Infinite

priority queue<pair<int, int>> pq

pq.empty() = True

- ② Make sure the priority queue we are going to use is empty → declare a new one

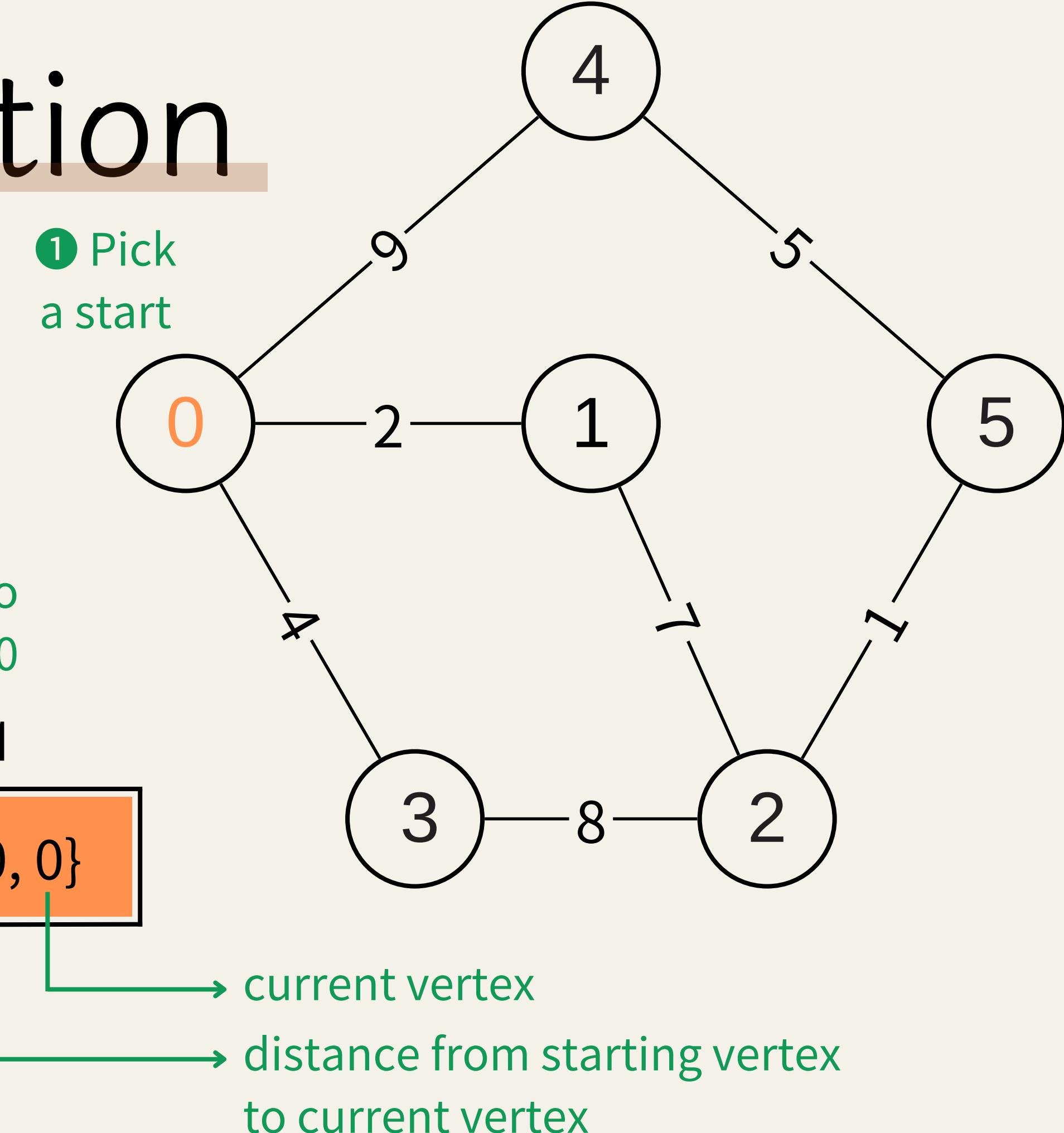


Initial Operation

index	0	1	2	3	4	5
dis[]	0	INF	INF	INF	INF	INF

- ② Set the distance of starting vertex to 0
→ The distance from starting vertex to starting vertex will definitely be 0

priority queue<pair<int, int>, int> q



- ③ Push starting vertex into priority queue
→ This means we are ready to deal with this vertex

Dijkstra

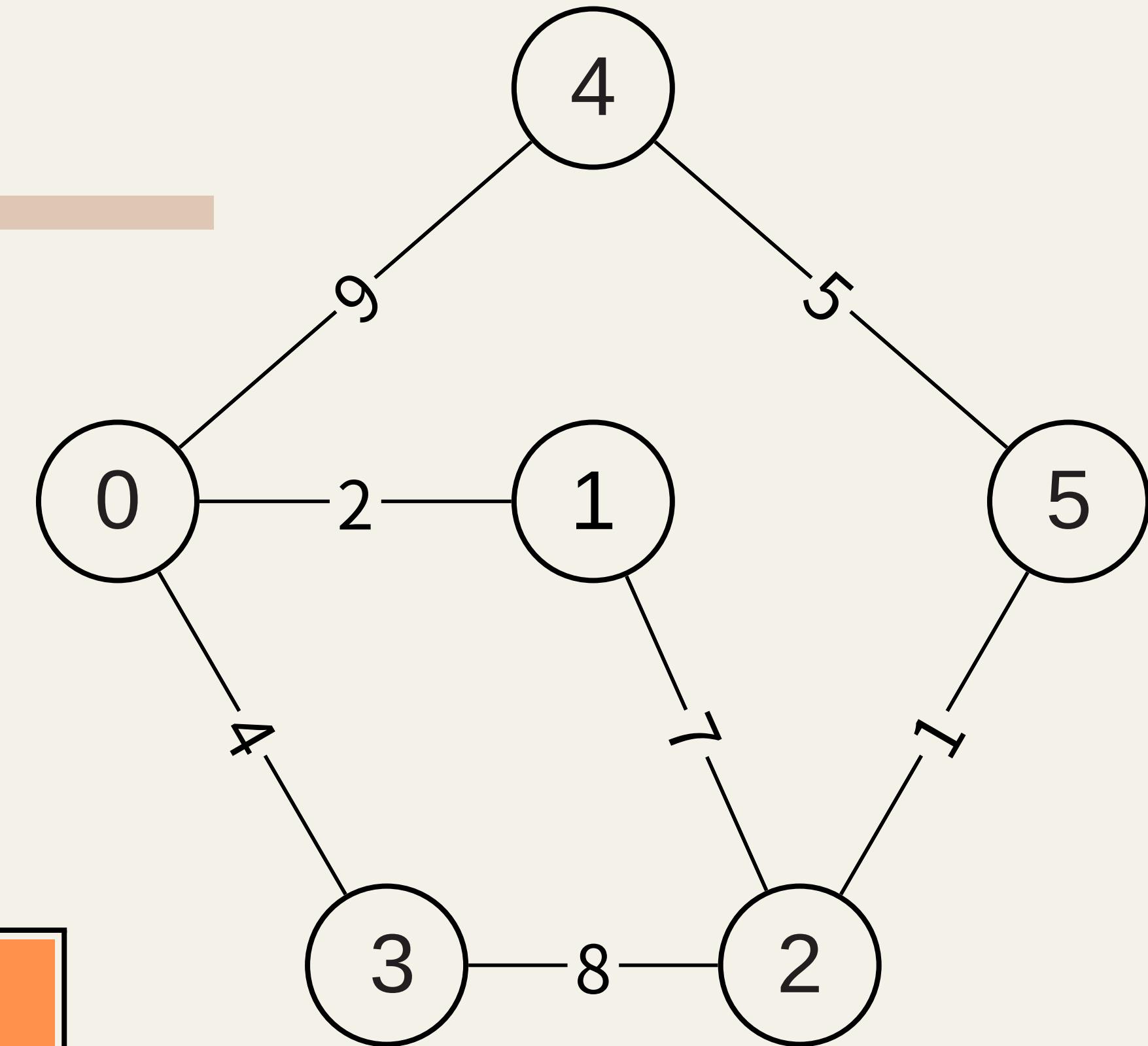
index 0 1 2 3 4 5

dis[]

0	INF	INF	INF	INF	INF
---	-----	-----	-----	-----	-----

priority queue<pair<int, int>> q

{0, 0}

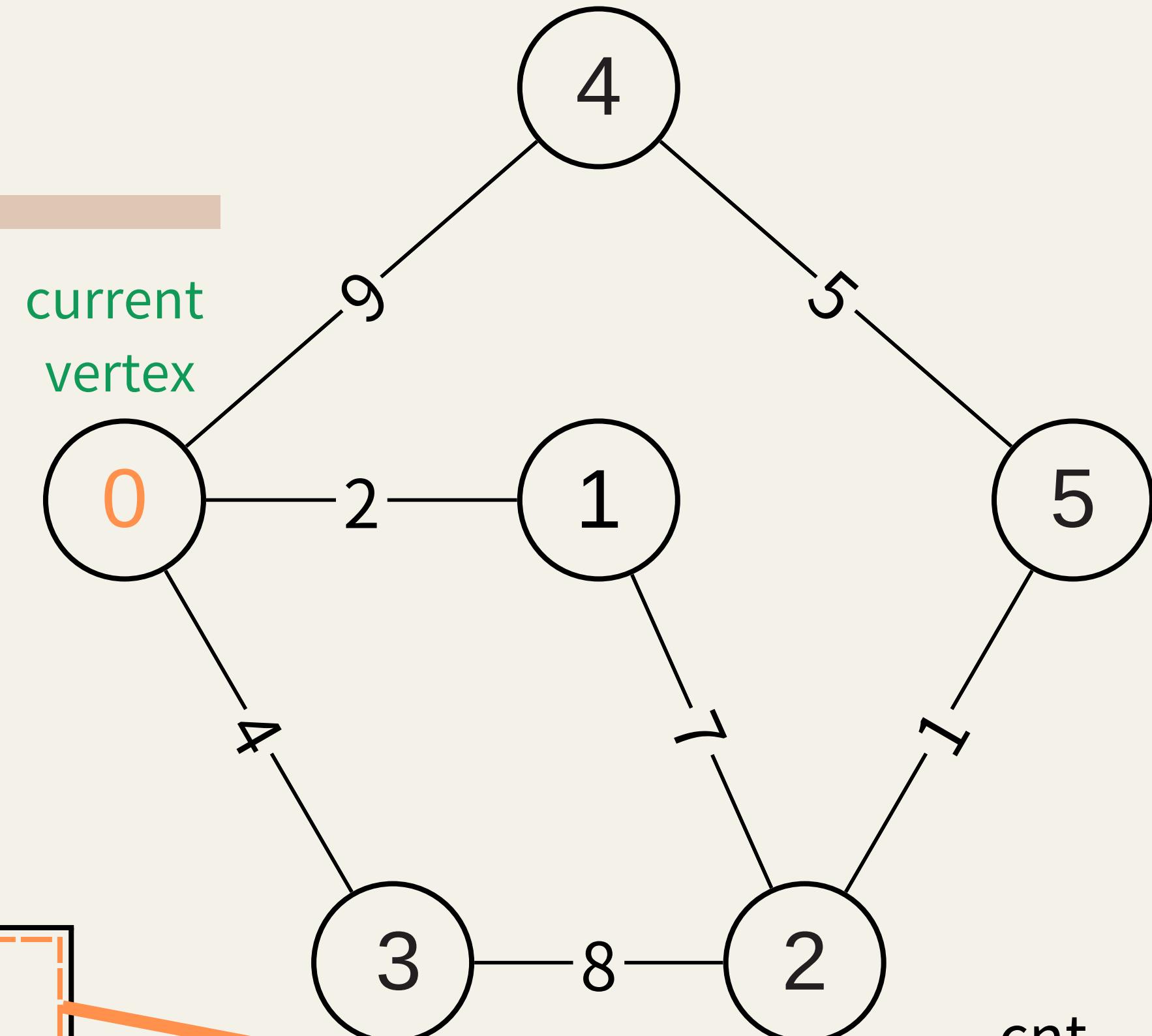


Once the priority queue is not empty
→ Keep going with the remaining vertices

Dijkstra

index	0	1	2	3	4	5
dis[]	0	INF	INF	INF	INF	INF

priority queue<pair<int, int>> q



cnt

{0, 0}

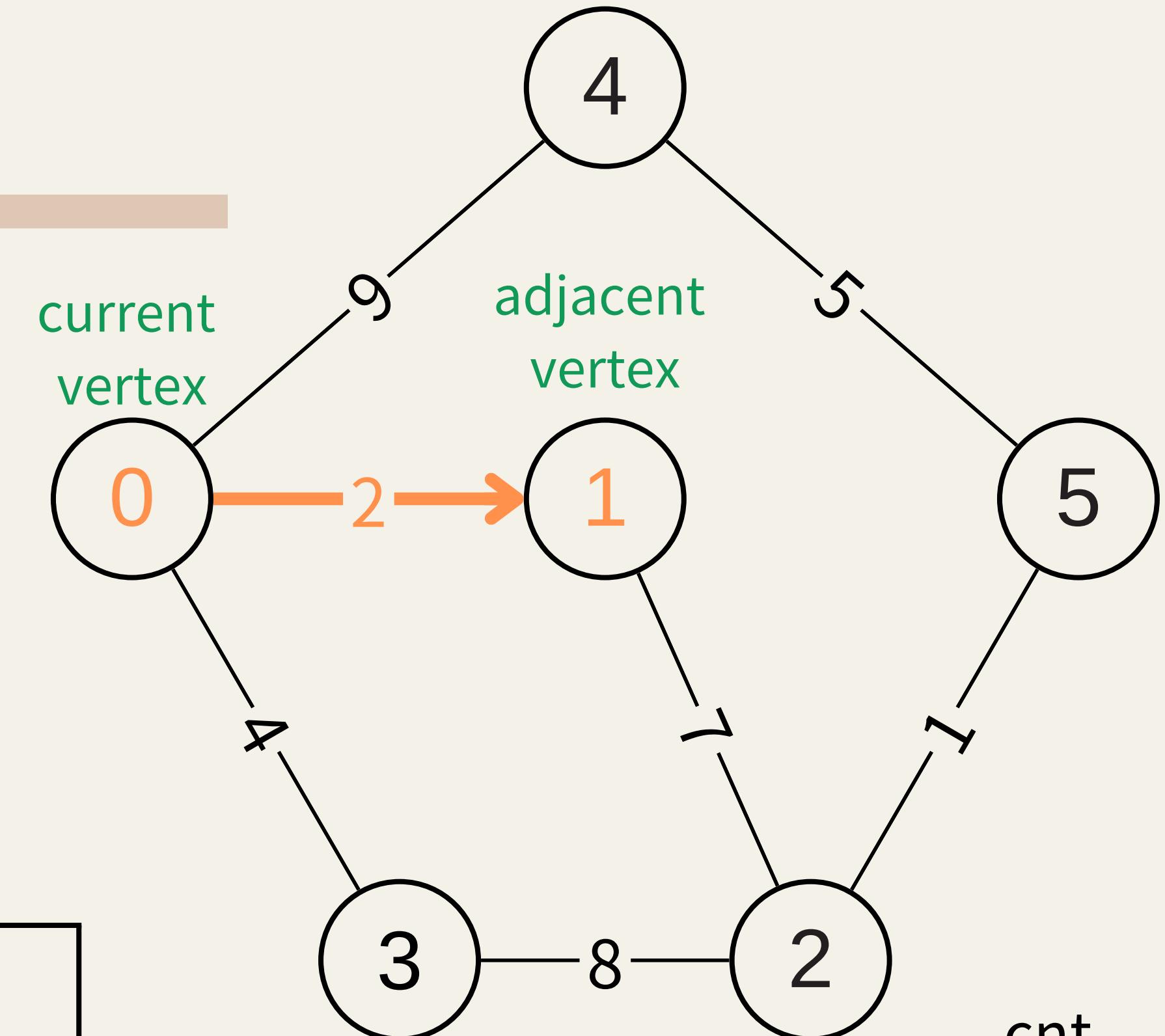
Dijkstra

index	0	1	2	3	4	5
dis[]	0	INF	INF	INF	INF	INF

Available ✓

Check the adjacent vertex
if can be relaxed or not

priority queue<pair<int, int>> q

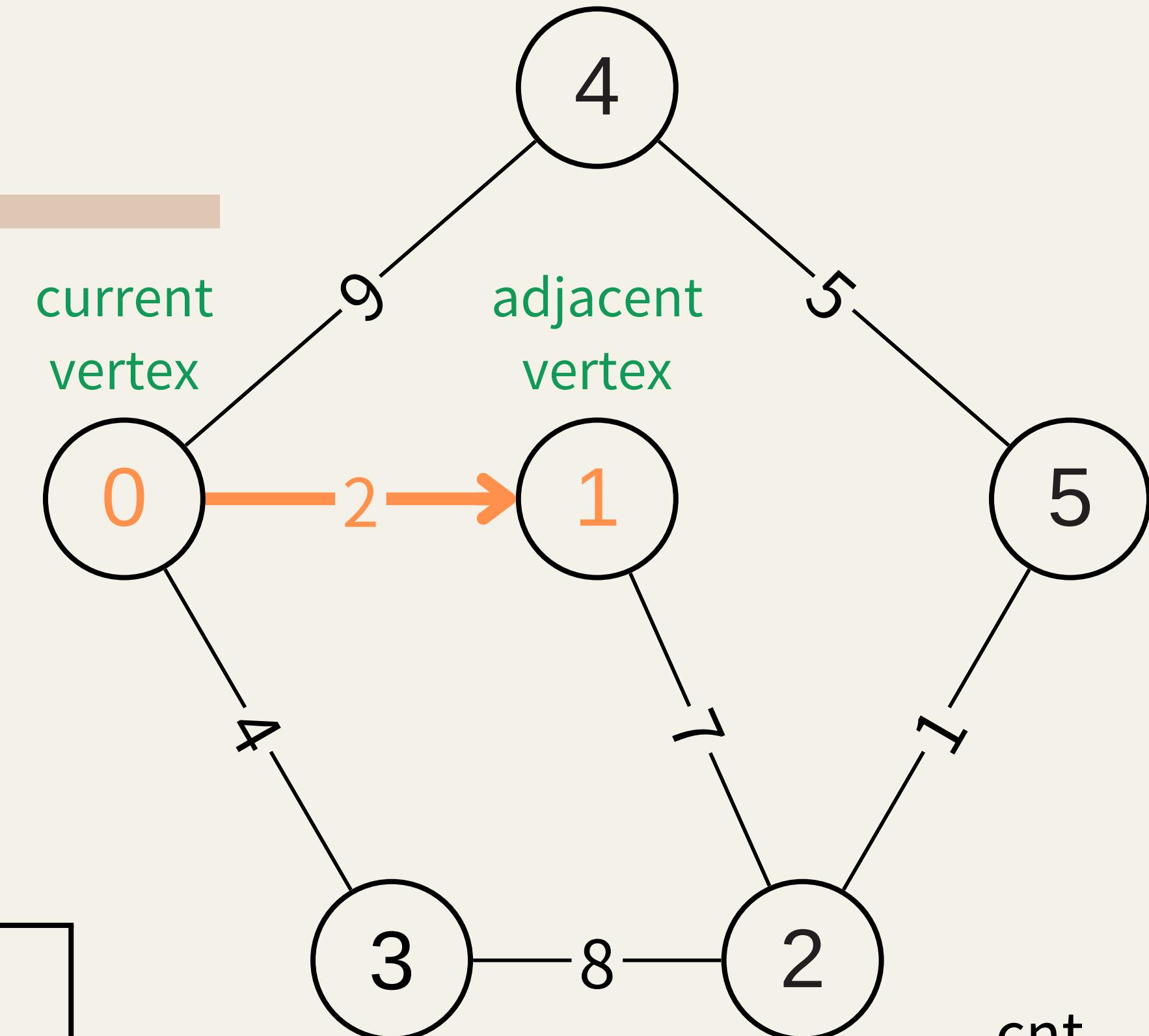


{0, 0}

Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	INF	INF

priority queue<pair<int, int>> q



cnt

{0, 0}

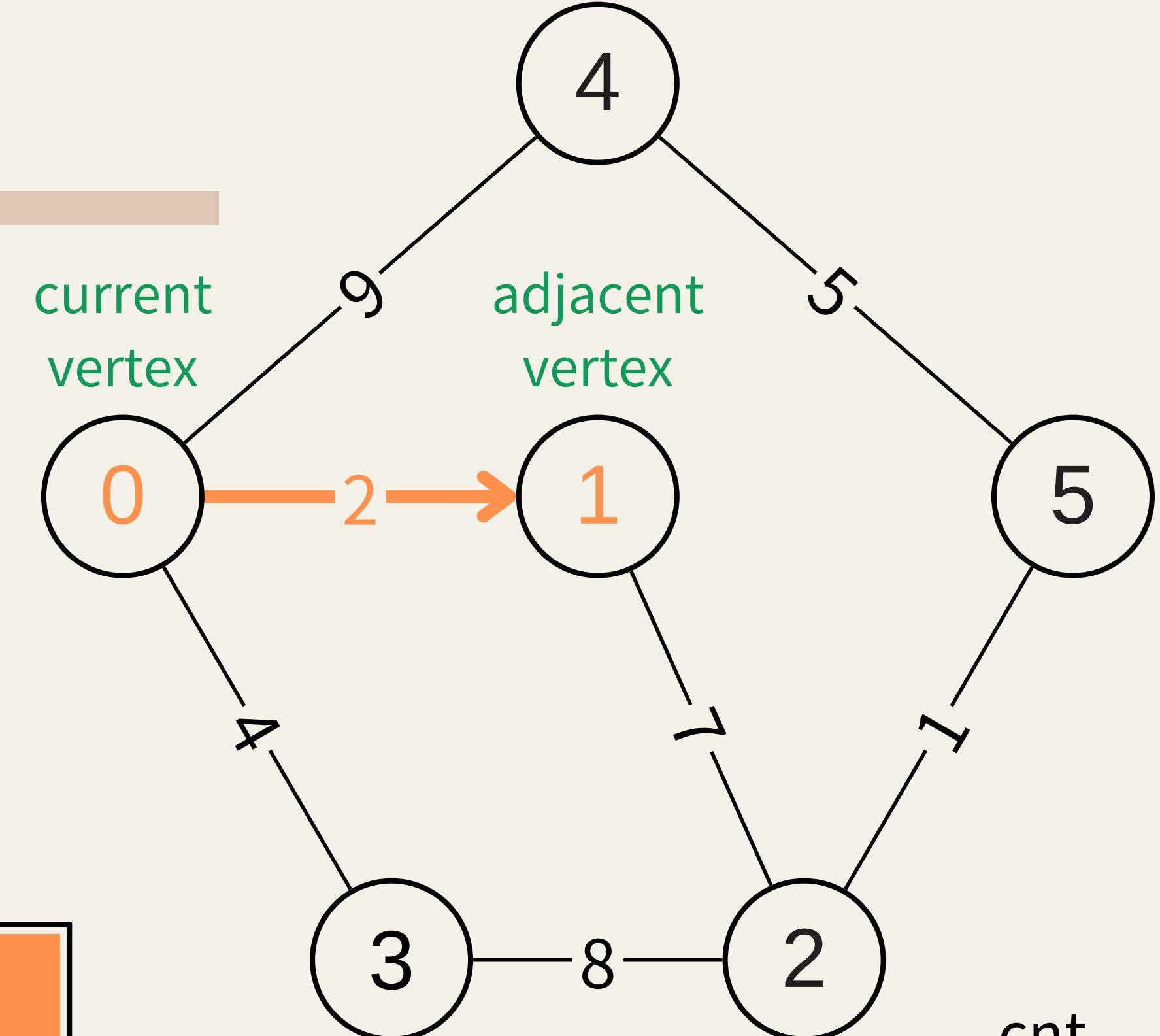
Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	INF	INF

priority queue<pair<int, int>> q



{2, 1}



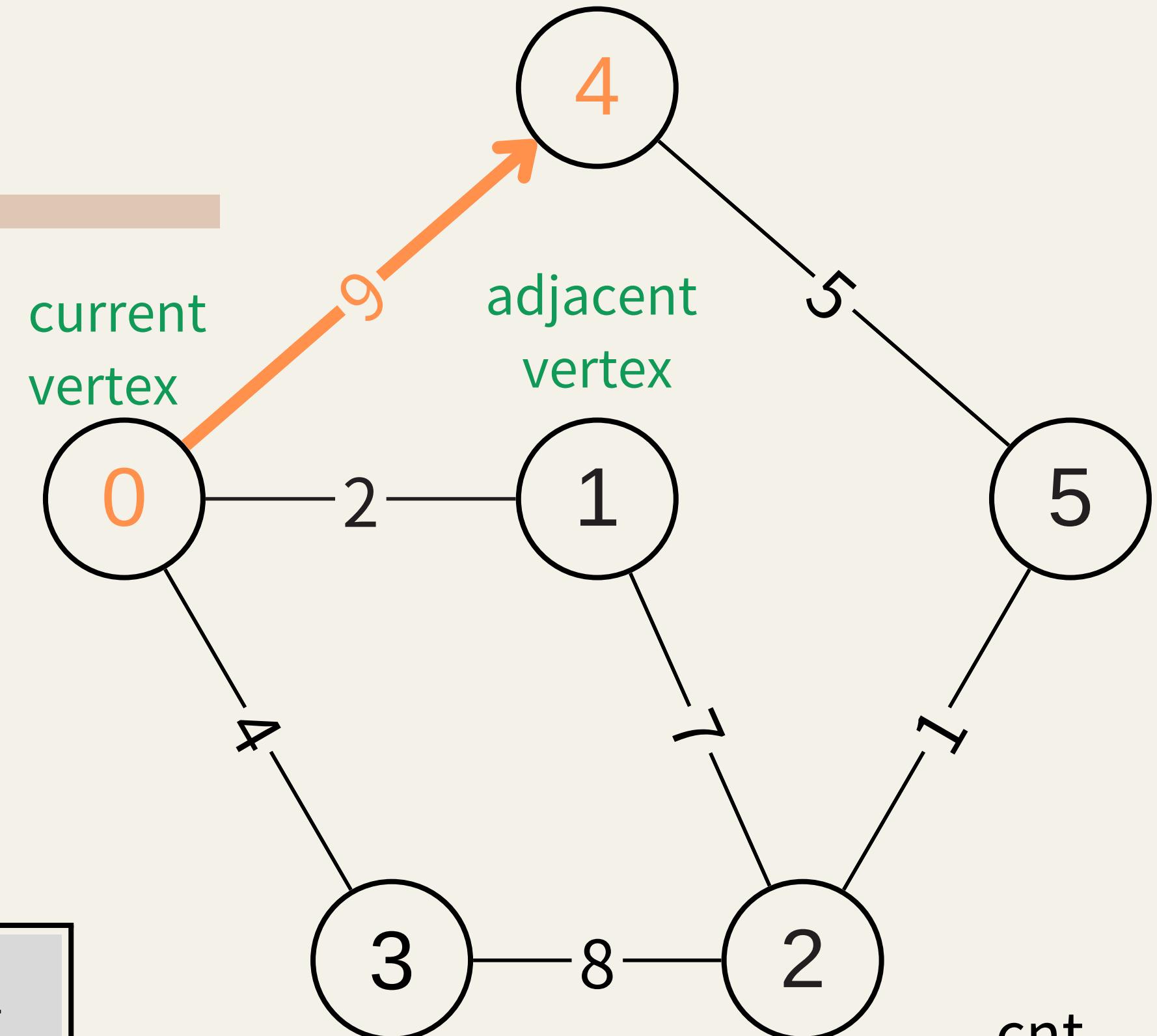
{0, 0}

Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	INF	INF

priority queue<pair<int, int>> q

{2, 1}



cnt

{0, 0}

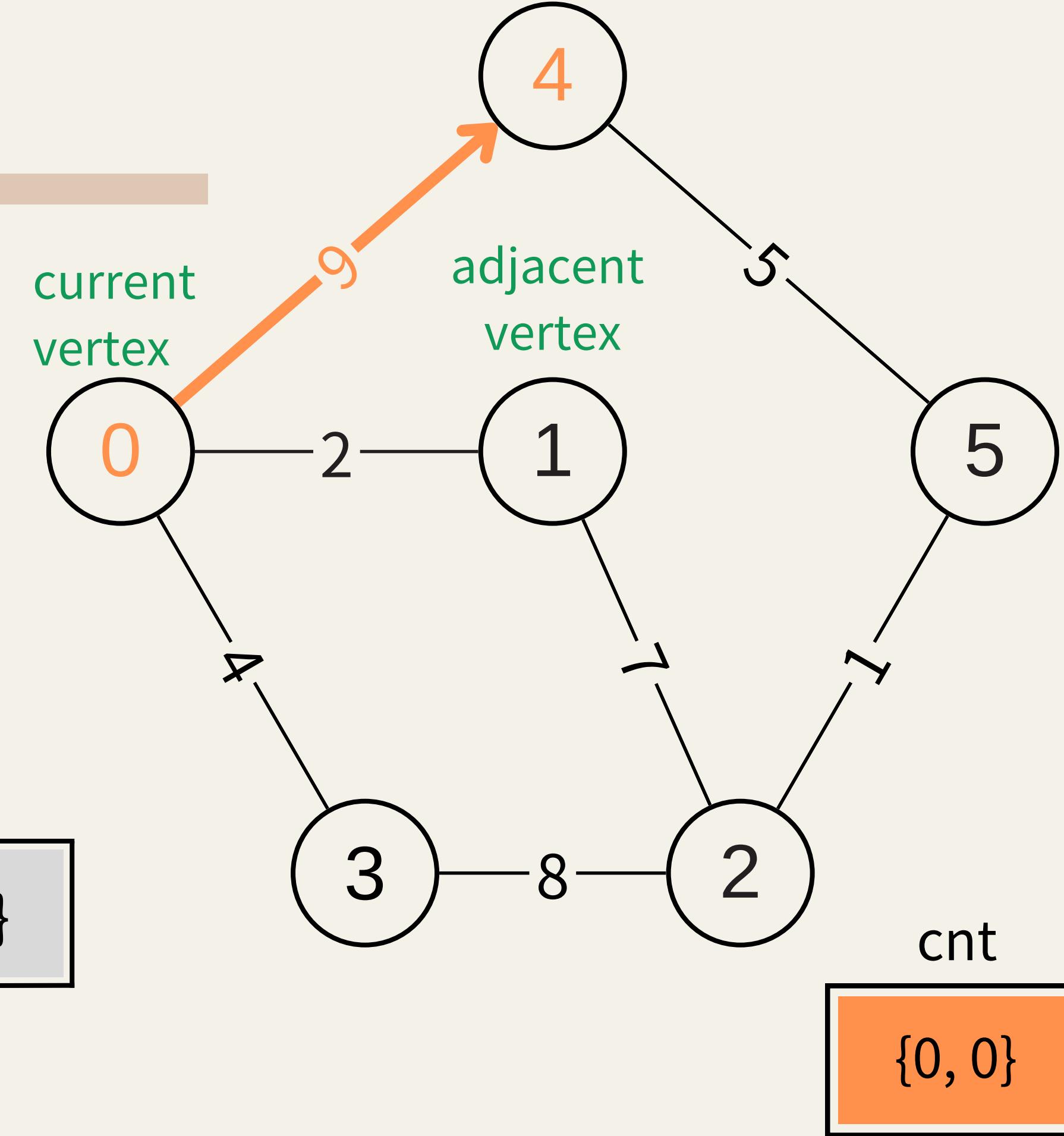
Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	INF	INF

Available ✓

priority queue<pair<int, int>> q

{2, 1}

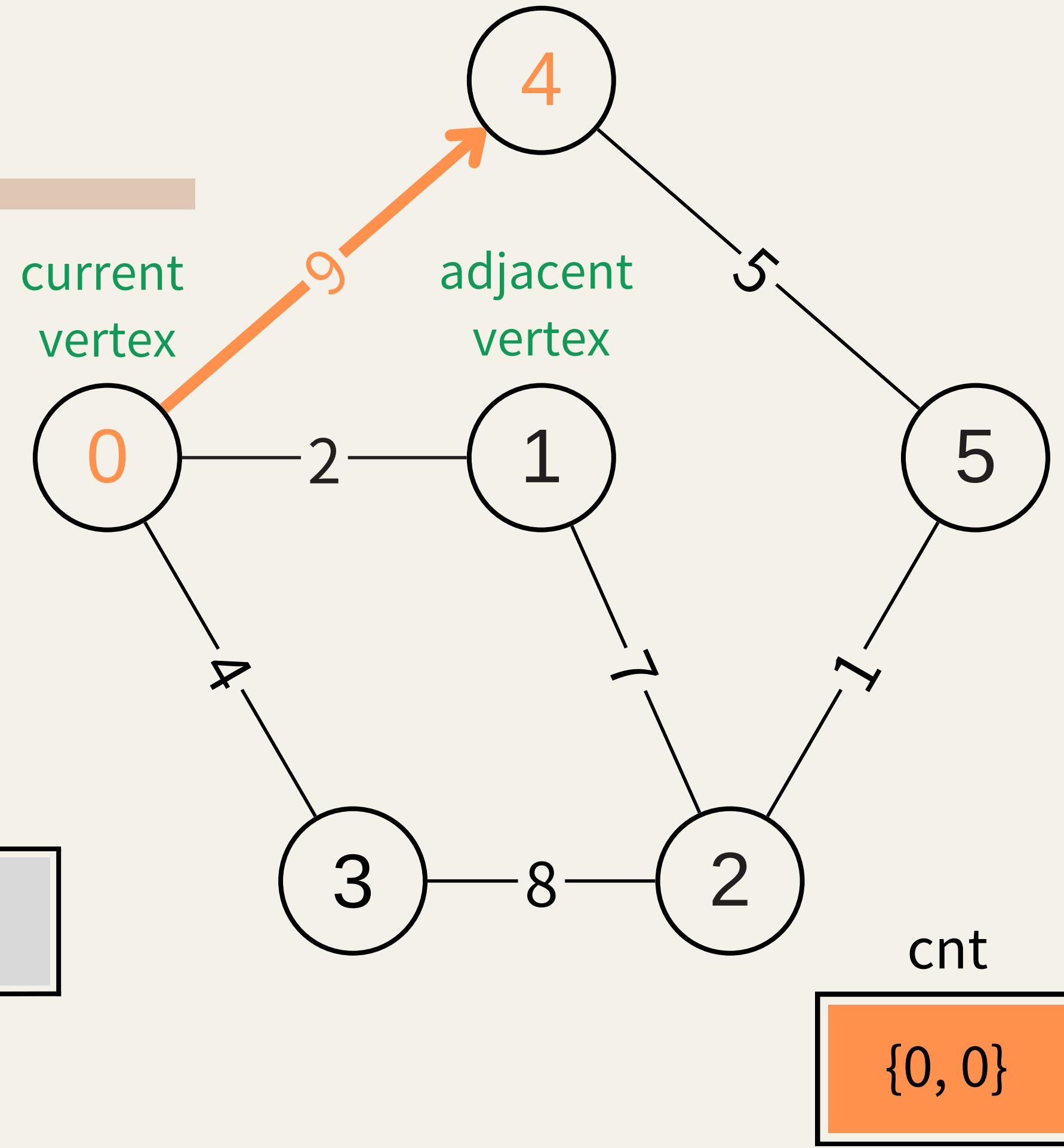


Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	9	INF

```
priority queue<pair<int, int>> q
```

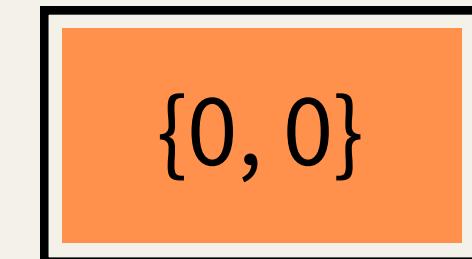
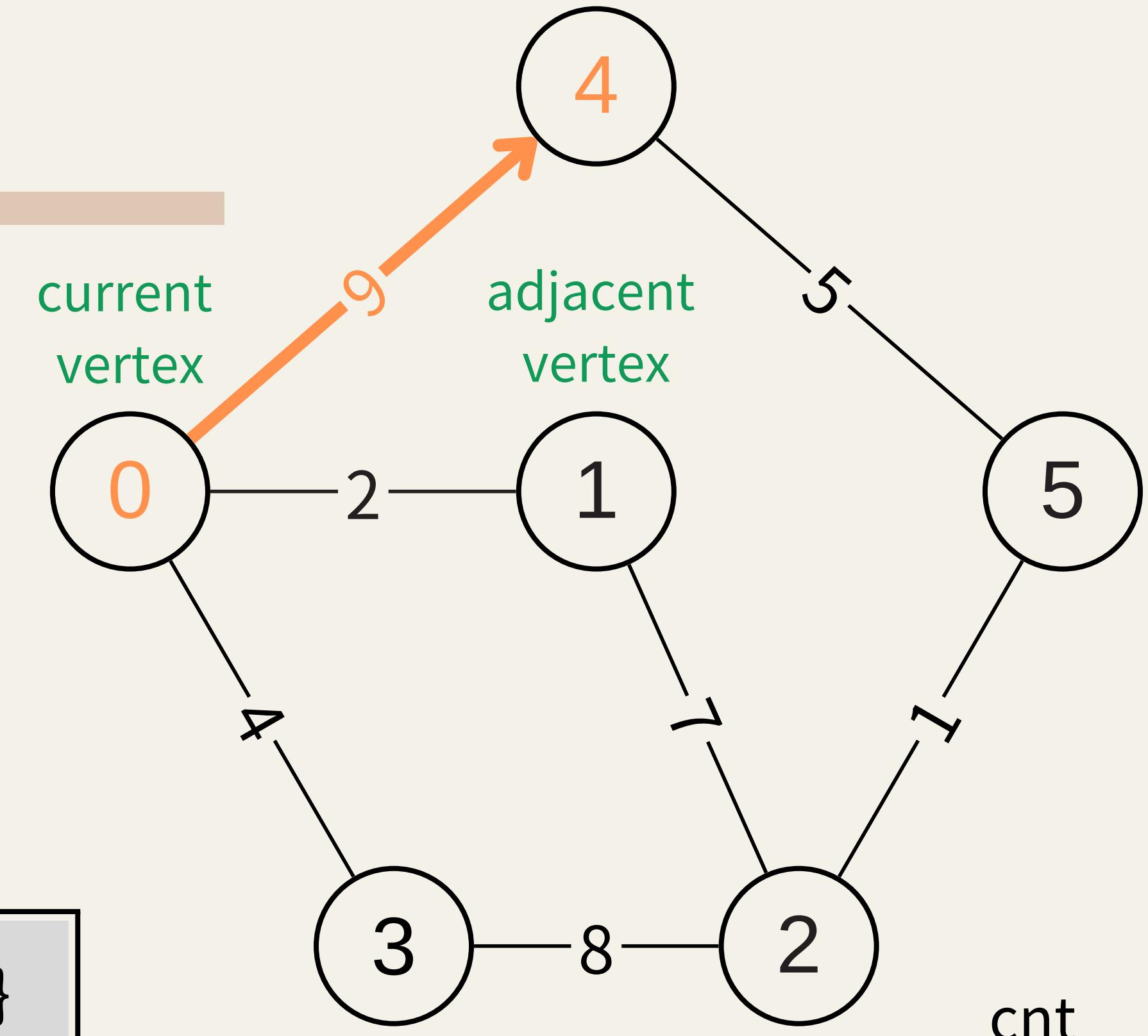
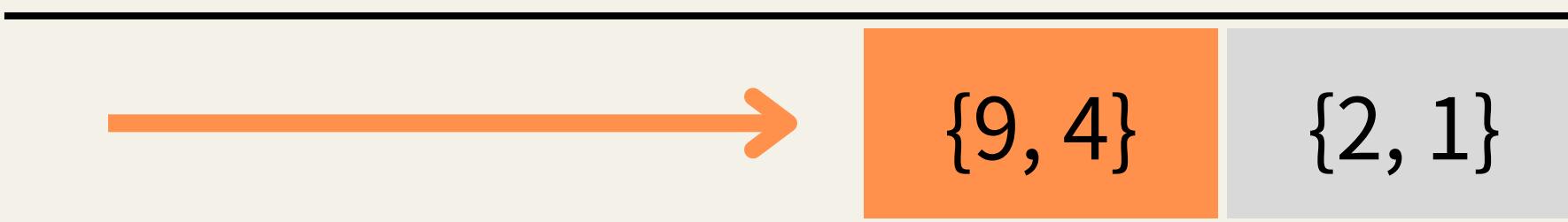
{2, 1}



Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	9	INF

priority queue<pair<int, int>> q

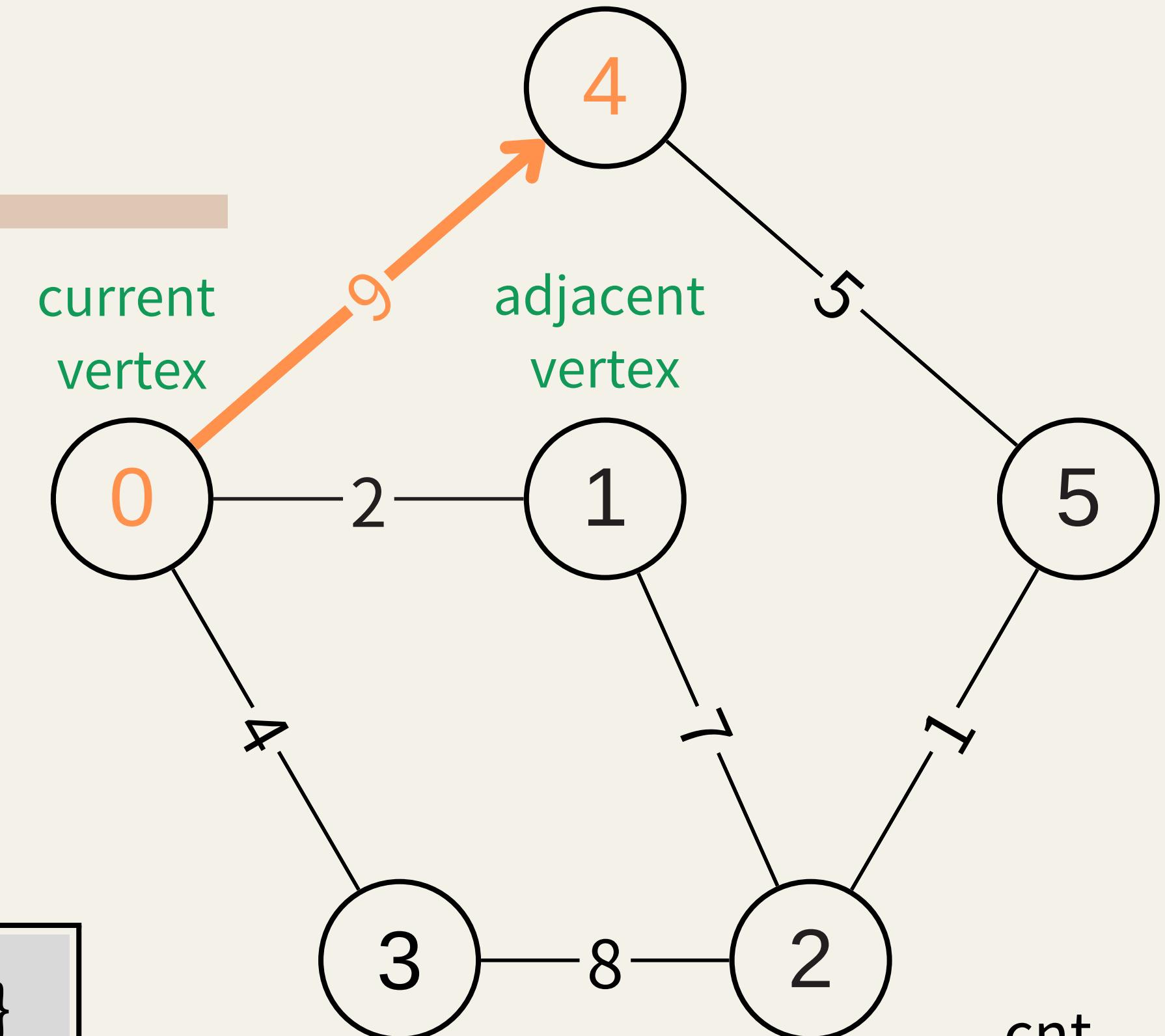


Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	9	INF

priority queue<pair<int, int>> q

{9, 4}	{2, 1}
--------	--------



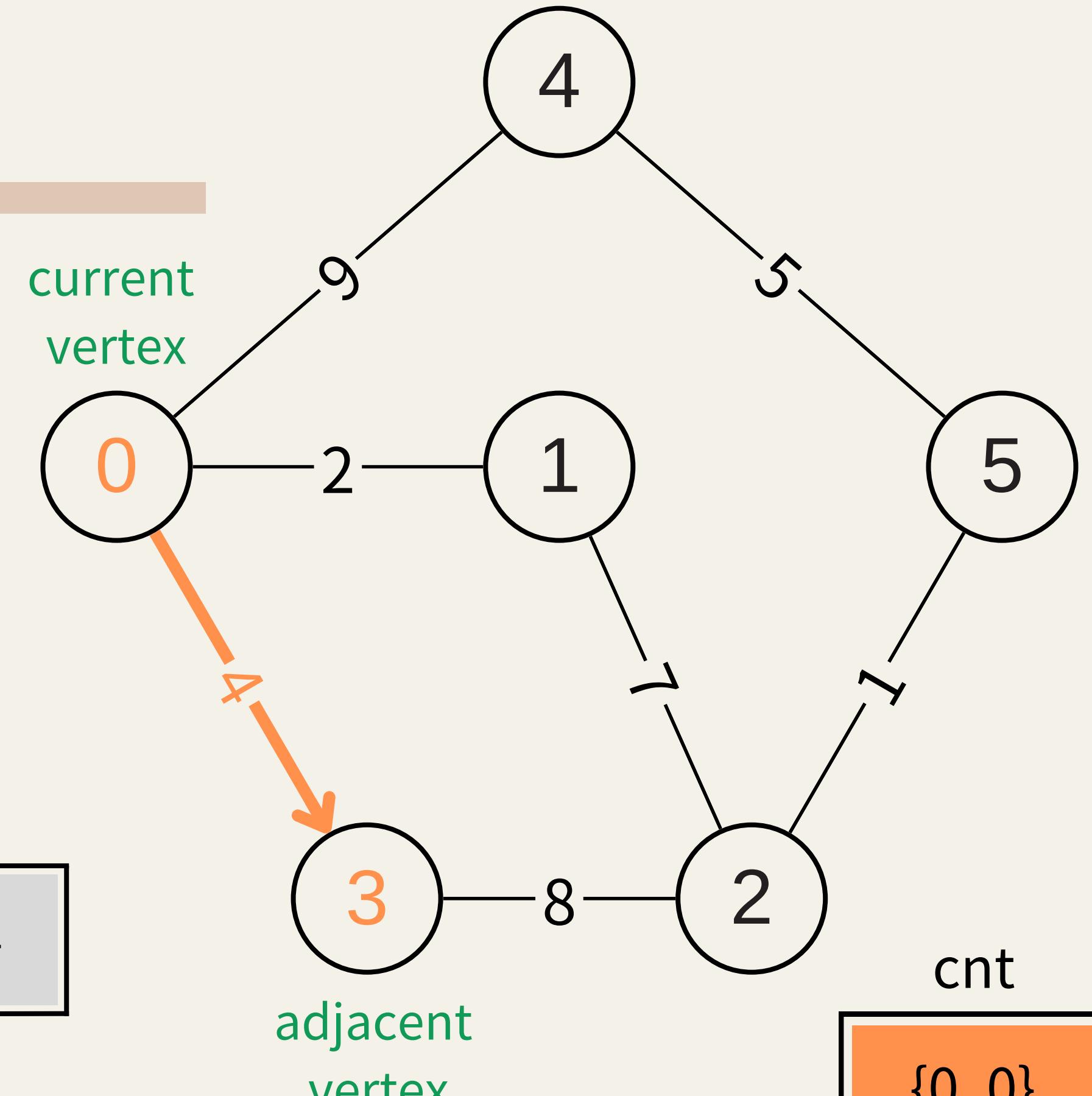
{0, 0}

Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	9	INF

priority queue<pair<int, int>> q

{9, 4}	{2, 1}
--------	--------



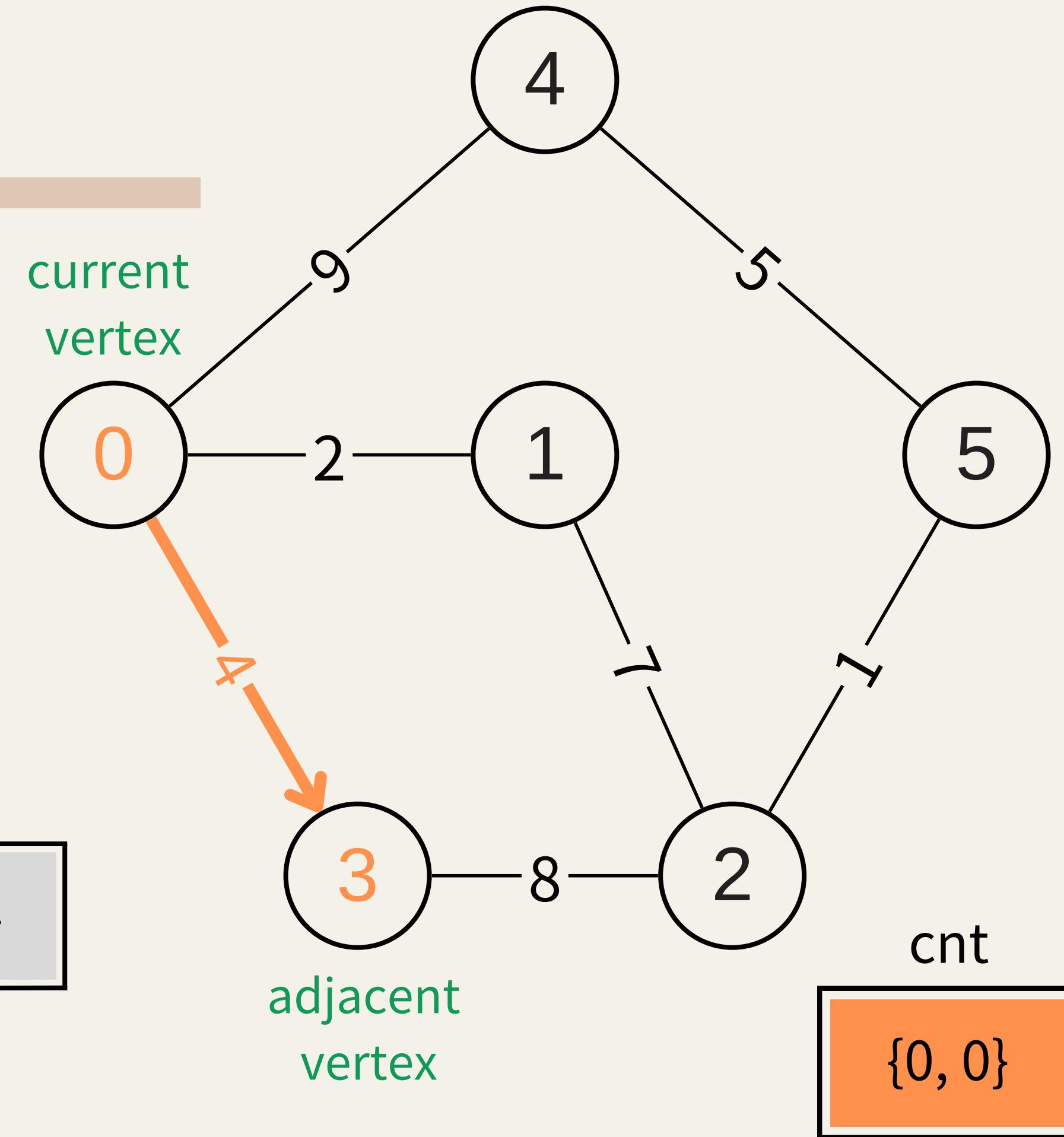
Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	INF	9	INF

Available ✓

priority queue<pair<int, int>> q

{9, 4} {2, 1}



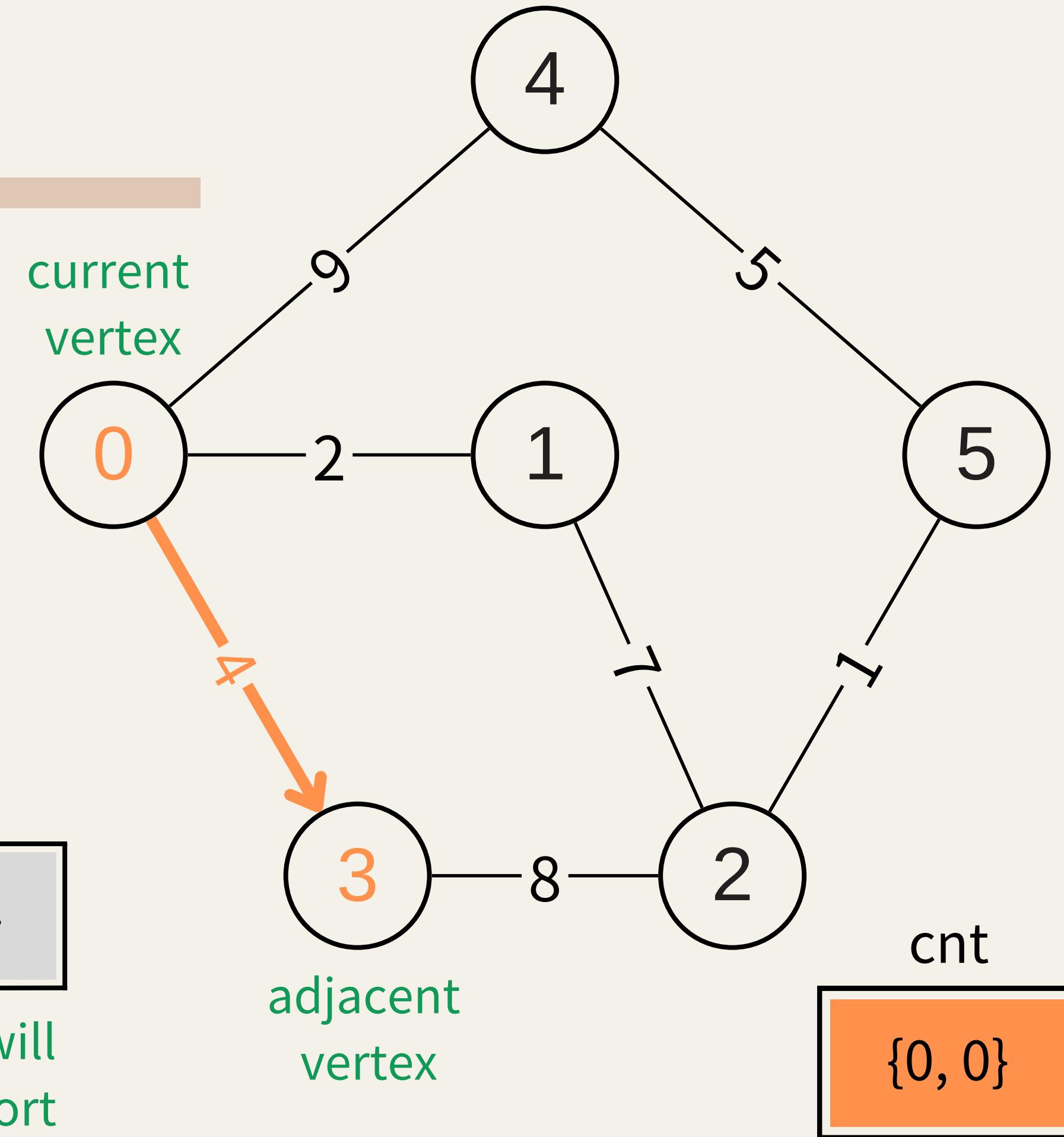
Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	INF	4	9	INF

priority queue<pair<int, int>> q



It is worth noting that priority queue will automatically sort



Dijkstra

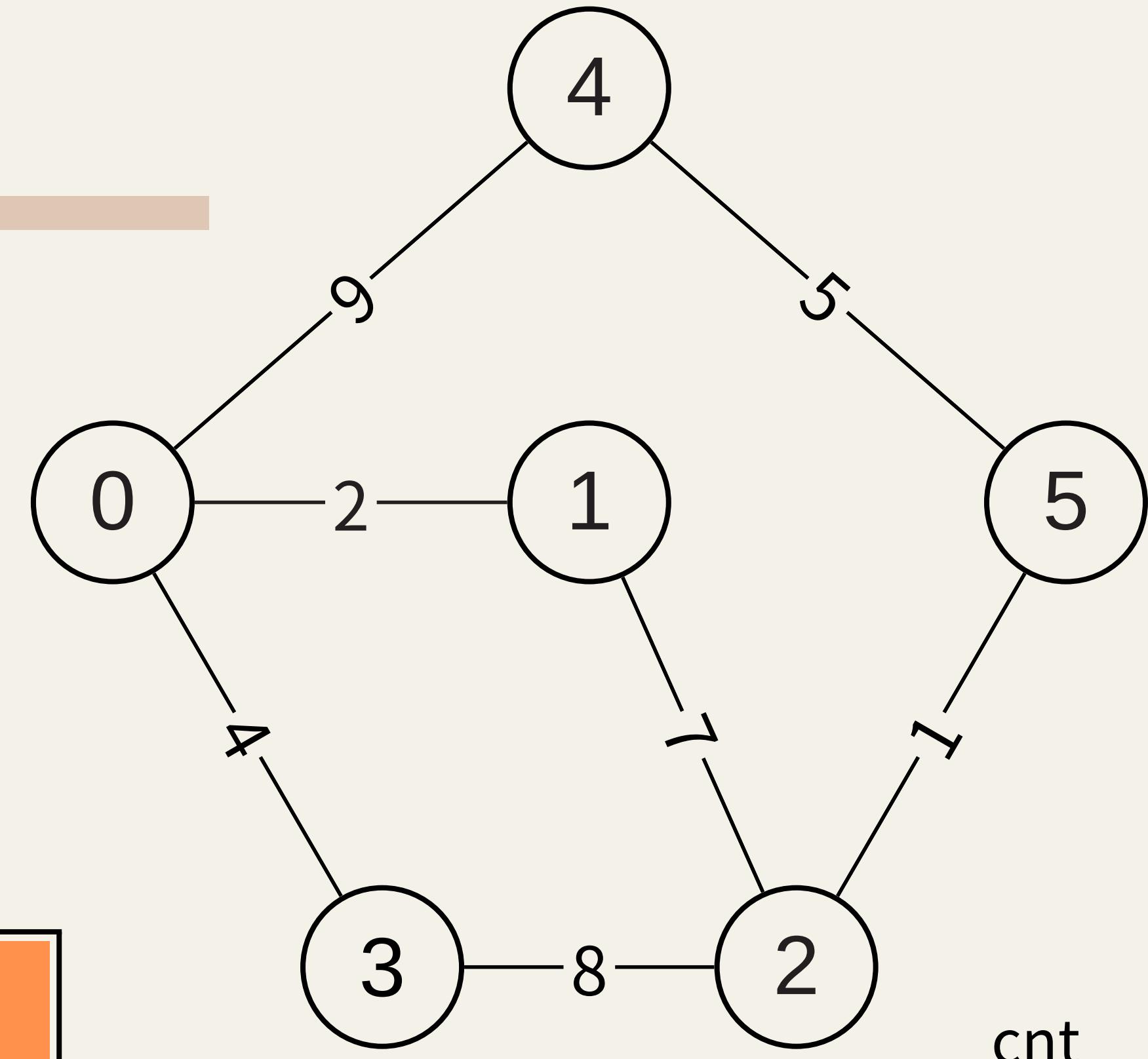
index 0 1 2 3 4 5

dis[]

0	2	INF	4	9	INF
---	---	-----	---	---	-----

priority queue<pair<int, int>> q

{9, 4}	{4, 3}	{2, 1}
--------	--------	--------



When all vertex adjacent to cnt have been processed

Then, Check the whether queue is empty or not

→ If true, keep going with the remaining vertices

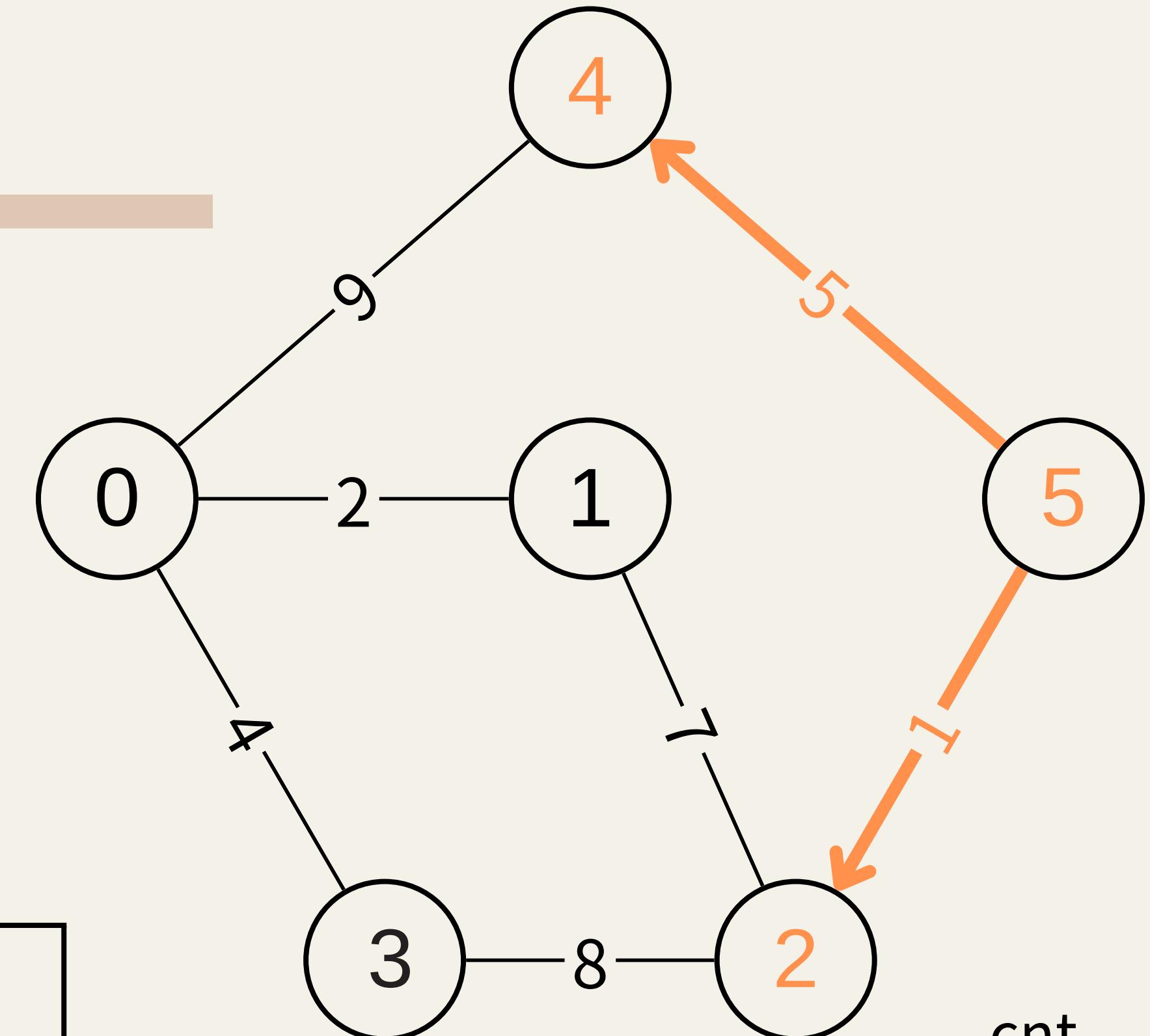
{0, 0}

Dijkstra

index	0	1	2	3	4	5
dis[]	0	2	9	4	9	10

This is the shortest path !

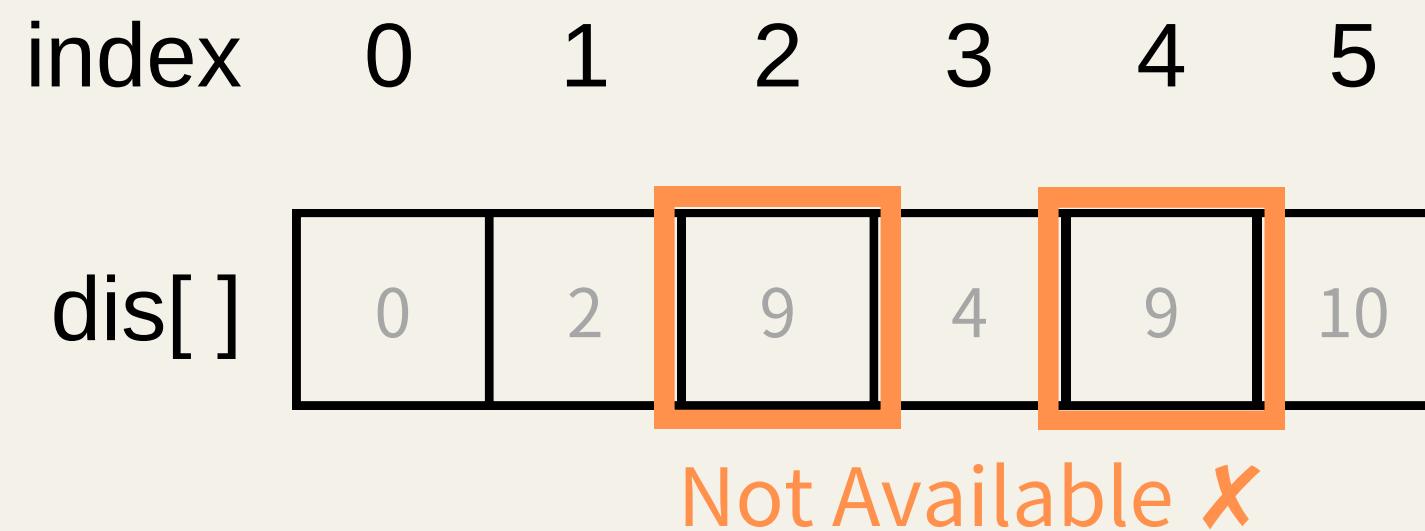
priority queue<pair<int, int>> q



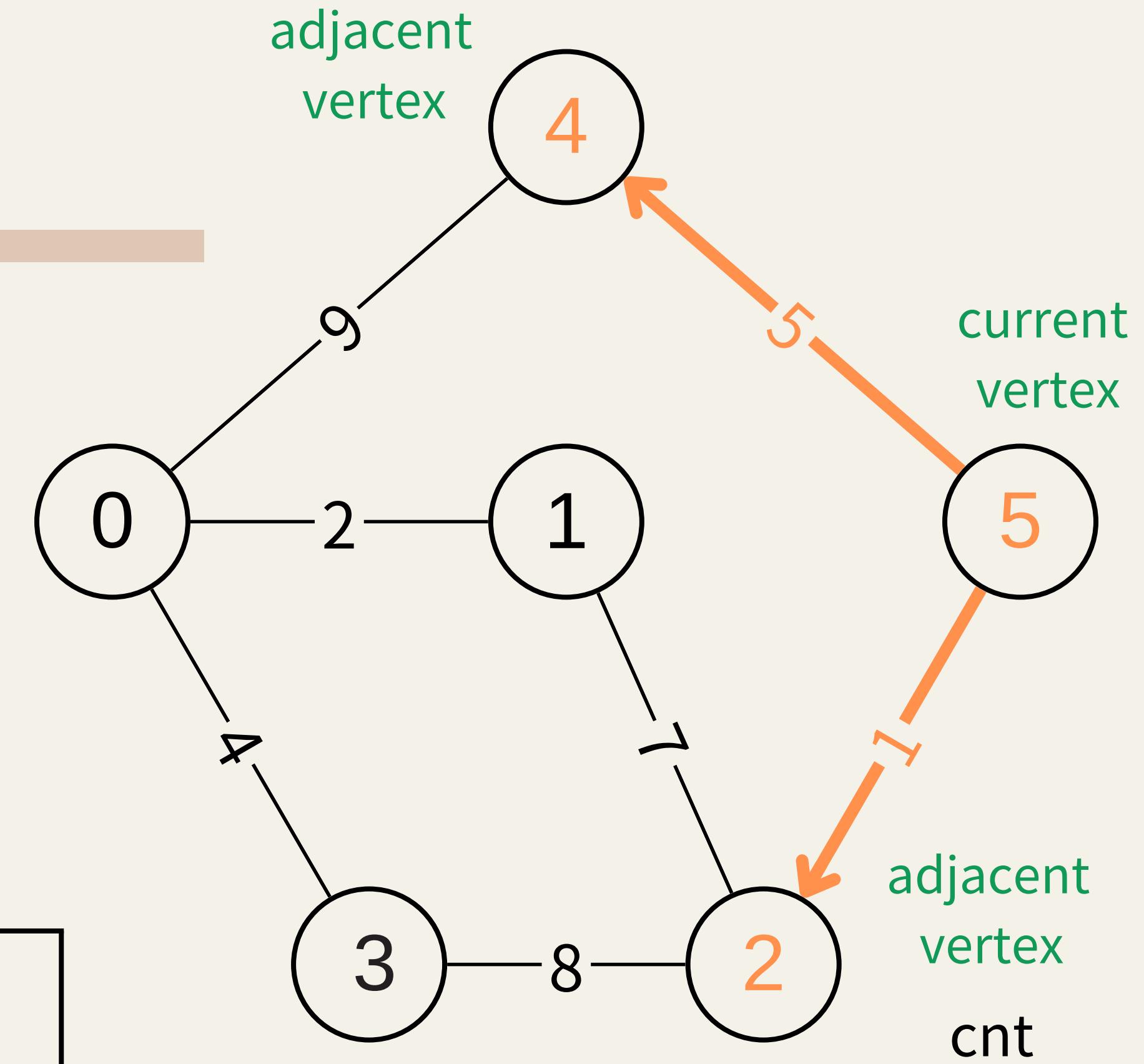
cnt

{10, 5}

Dijkstra



priority queue<pair<int, int>> q



When all vertex adjacent to cnt have been processed
Then, Check the whether queue is empty or not
→ If true, keep going with the remaining vertices



Bellman-Ford

貝爾曼-福特演算法

Initialize

index	0	1	2	3
dis[]	0	INF	INF	INF

① Set all of the value distance array to default
value → Infinite

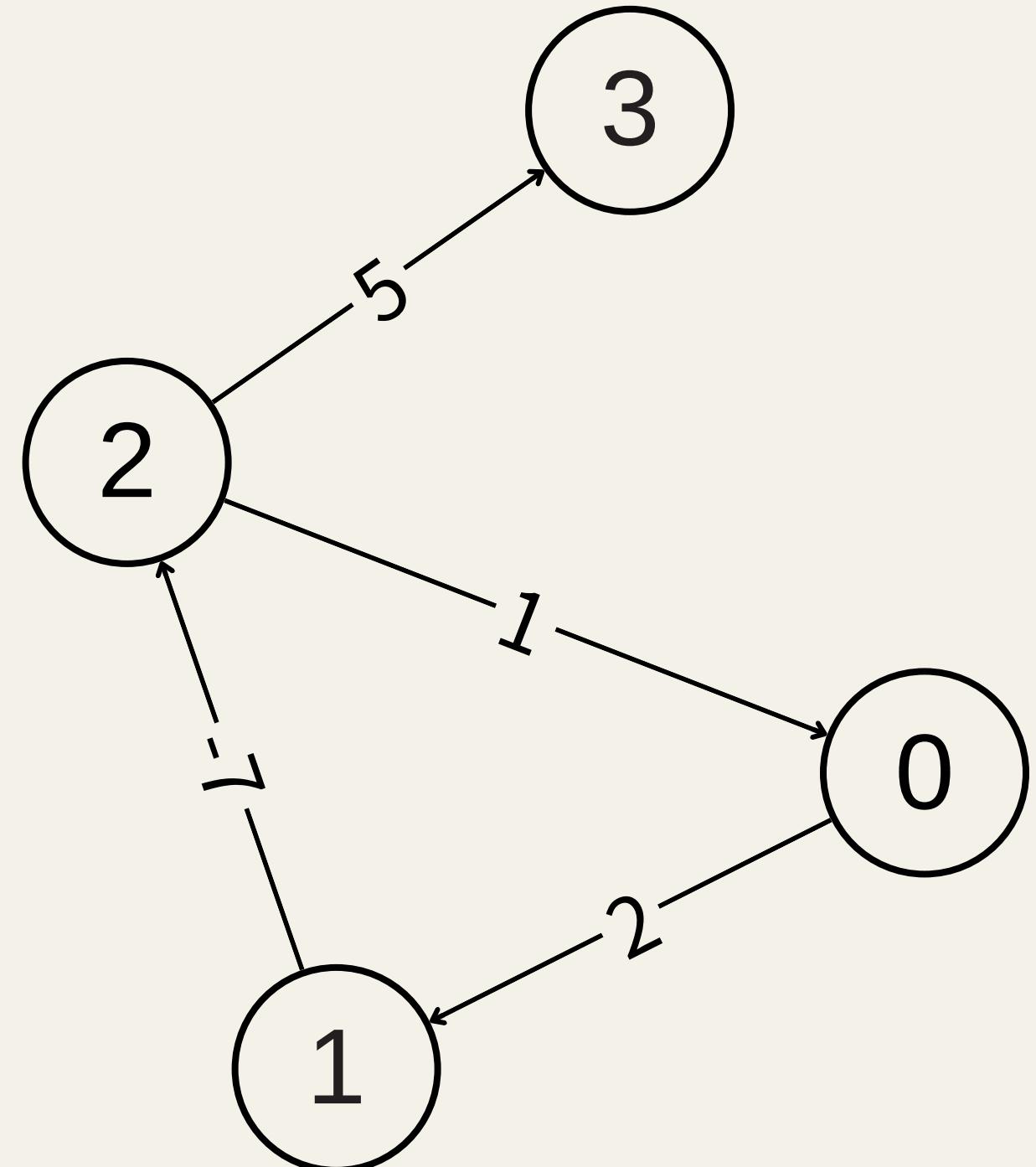
② Set dis[start] → 0

vector<vector<int>> edge

{-7, 1, 2}	{1, 2, 0}	{5, 2, 3}	{2, 0, 1}
------------	-----------	-----------	-----------

edge[3] edge[2] edge[1] edge[0]

③ Push all of the edge into vector

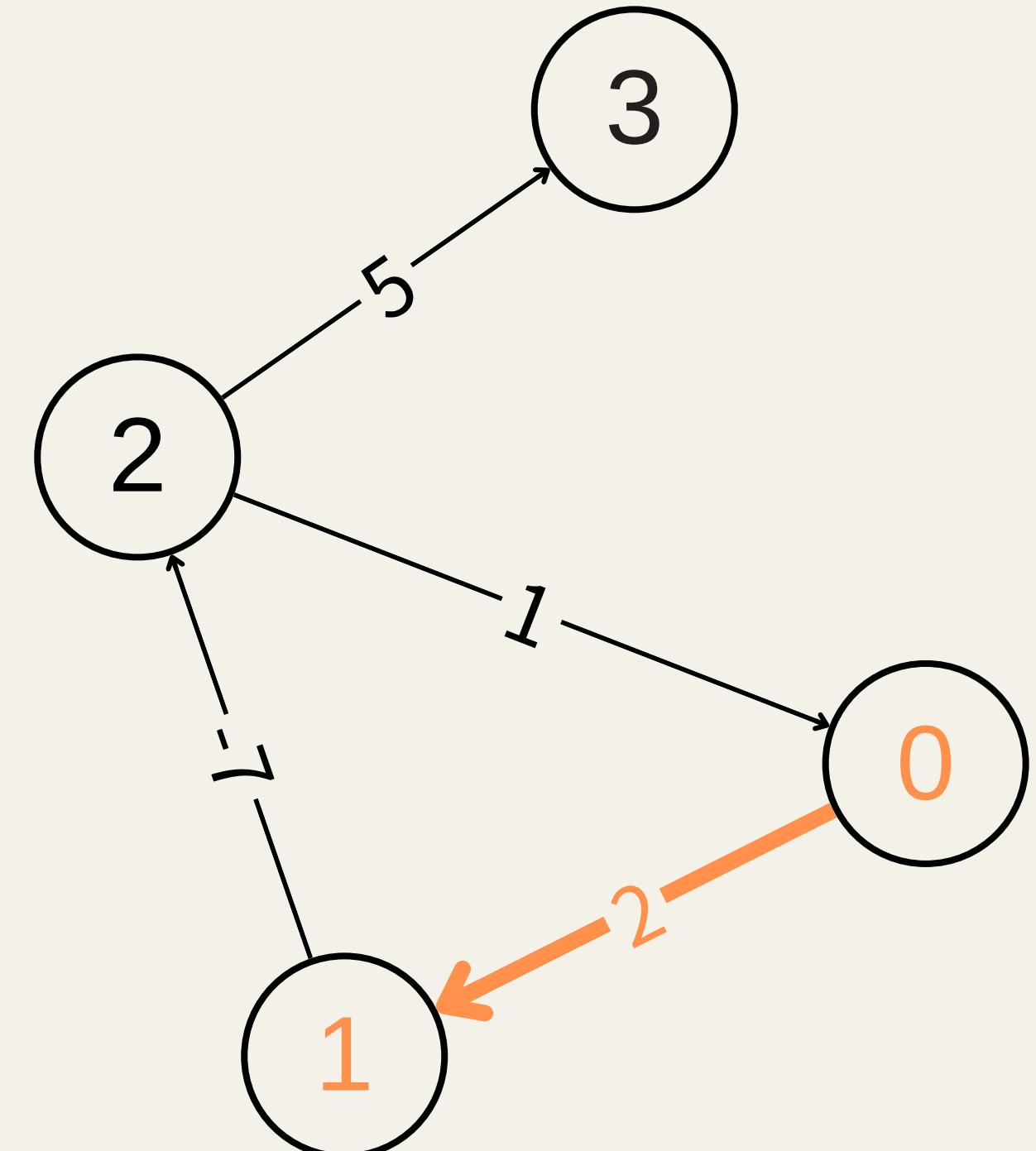
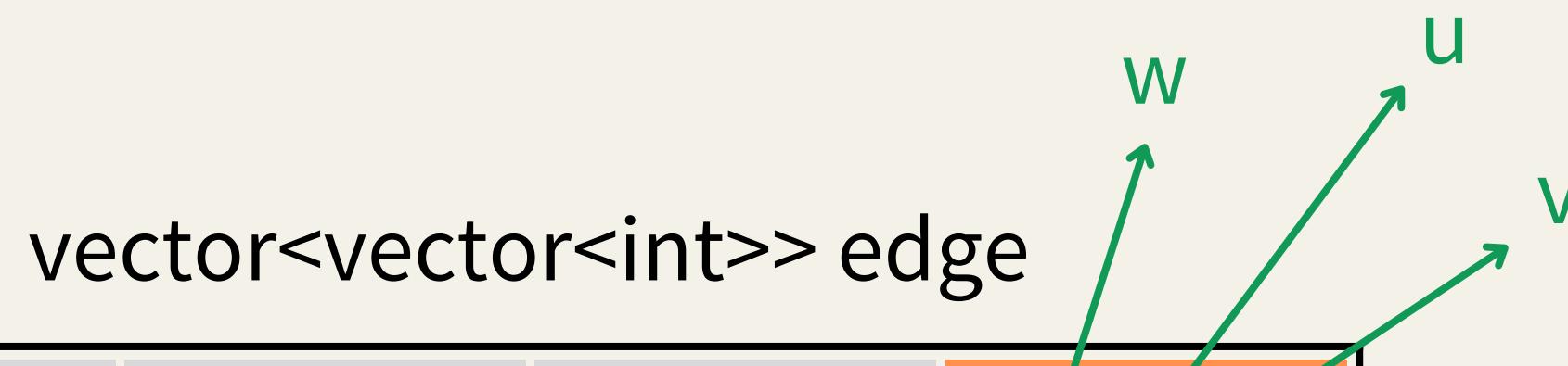


④ Define $\forall \text{edge}[i] \in \text{edge}$
 $\text{edge}[i][0] \rightarrow \text{weight}$
 $\text{edge}[i][1] \rightarrow \text{vertex u}$
 $\text{edge}[i][2] \rightarrow \text{vertex v}$

Bellman-ford

index	0	1	2	3
dis[]	0	2	INF	INF

② If $\text{edge}[i]$ is relaxable \rightarrow relax the edge



① Check If $\text{edge}[i]$ is relaxable
 $\rightarrow \text{dis}[u] + w < \text{dis}[v]$

Bellman-ford

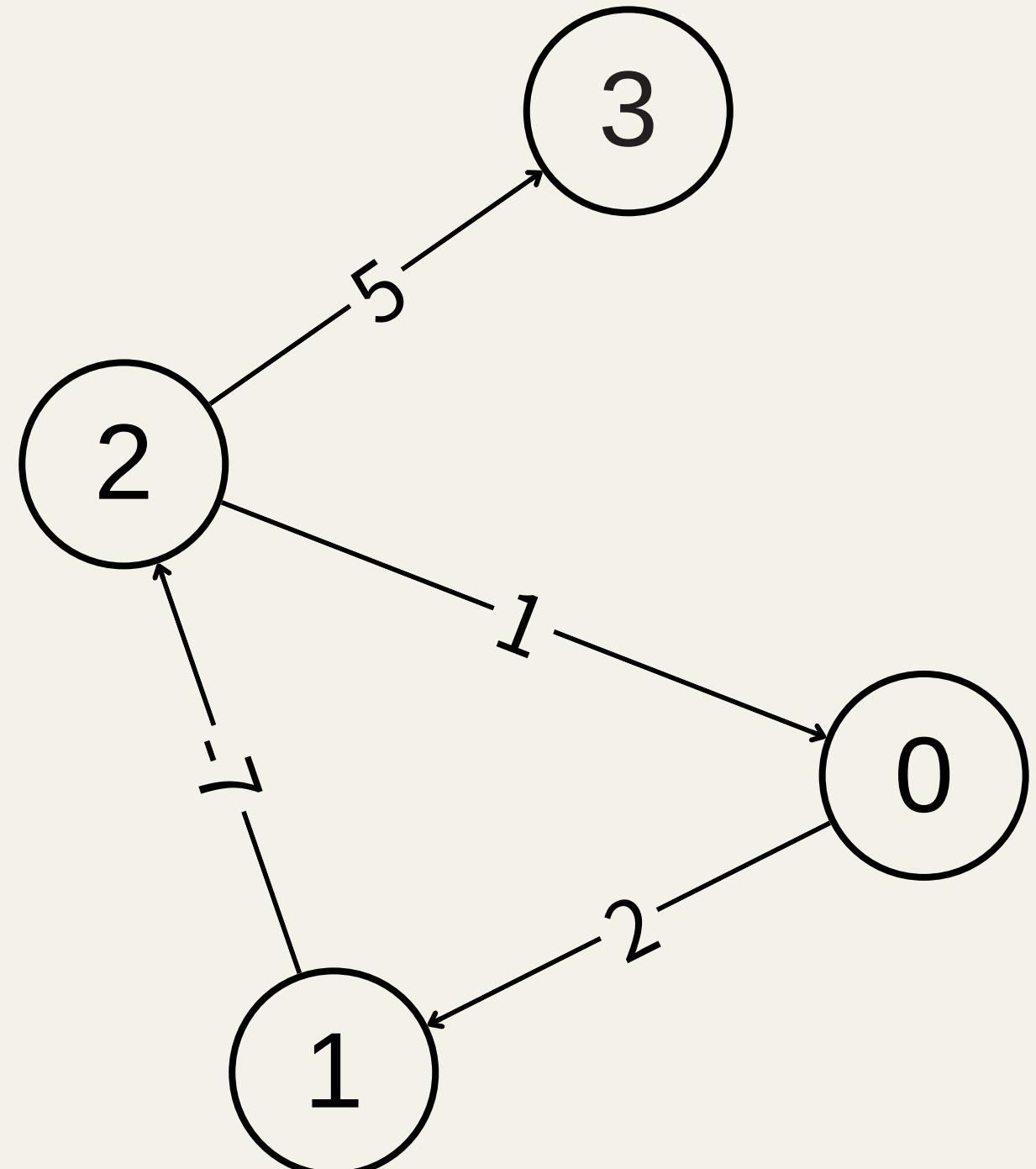
index	0	1	2	3
dis[]	0	2	-5	INF

→ After 1 round of relax operation

vector<vector<int>> edge

{-7, 1, 2}	{1, 2, 0}	{5, 2, 3}	{2, 0, 1}
------------	-----------	-----------	-----------

edge[3] edge[2] edge[1] edge[0]



Bellman-ford

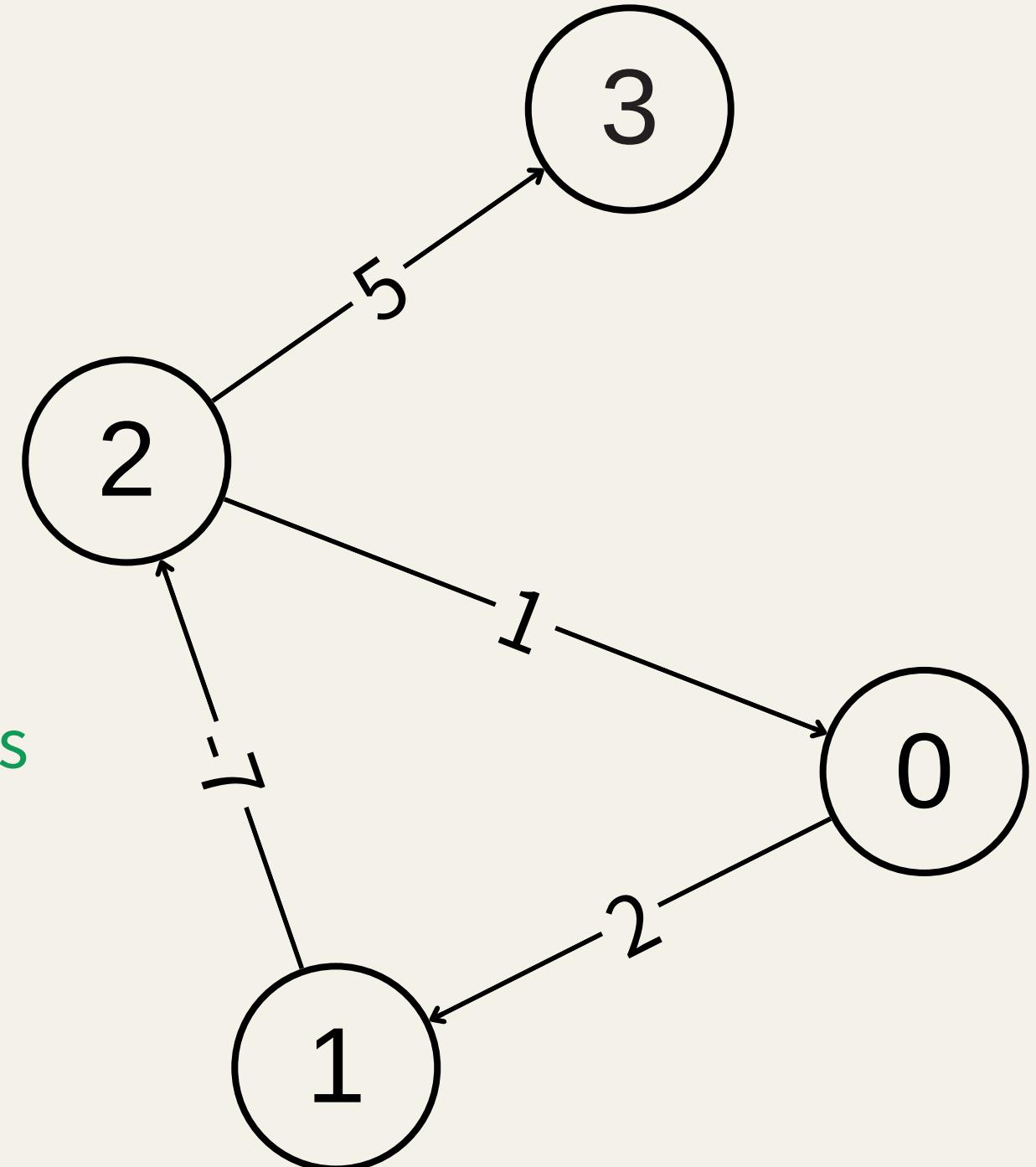
index	0	1	2	3
dis[]	-8	-2	-9	-4

After N-1 rounds of relax operation. If there are edges that are still relaxable → The graph contains a cycle

vector<vector<int>> edge

{-7, 1, 2}	{1, 2, 0}	{5, 2, 3}	{2, 0, 1}
------------	-----------	-----------	-----------

edge[3] edge[2] edge[1] edge[0]





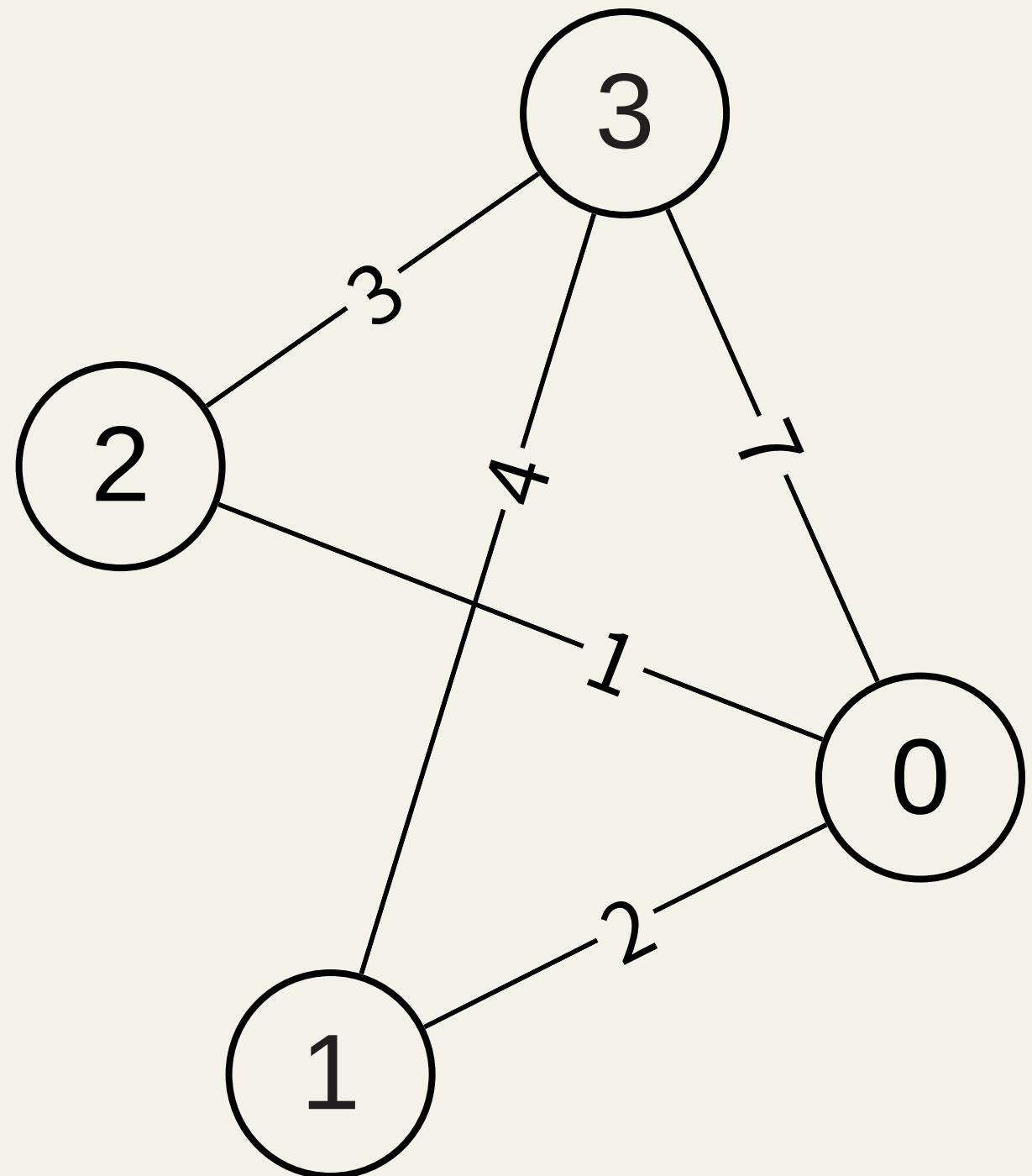
Floyd-Warshall

佛洛依德演算法

Initialize

index	0	1	2	3
dis[][]	0	2	1	7
	2	0	INF	4
	1	INF	0	3
	7	4	3	0

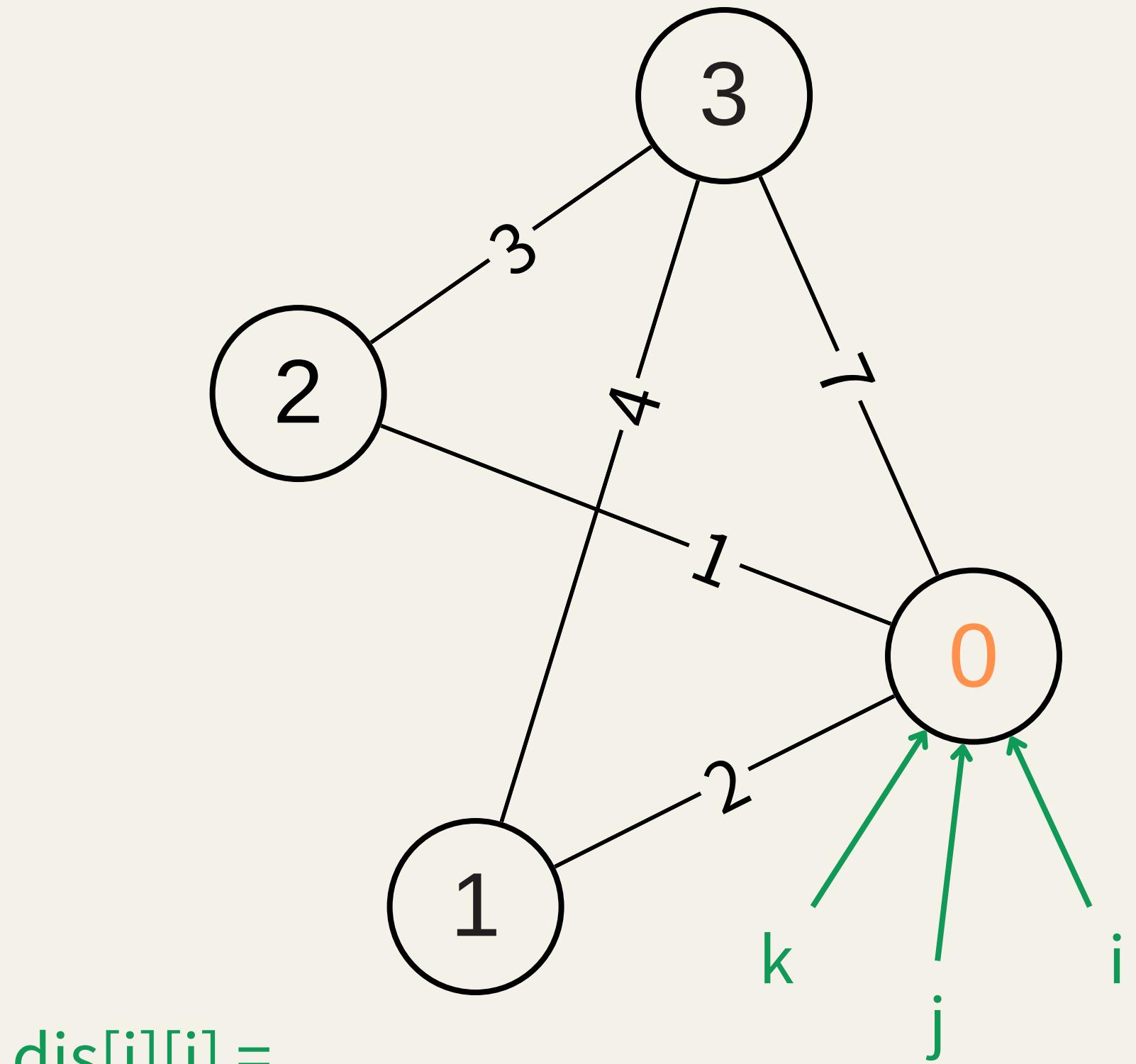
0
1
2
3



Floyd-Washall

index	0	1	2	3
dis[][]	0	2	1	7
	2	0	INF	4
	1	INF	0	3
	7	4	3	0

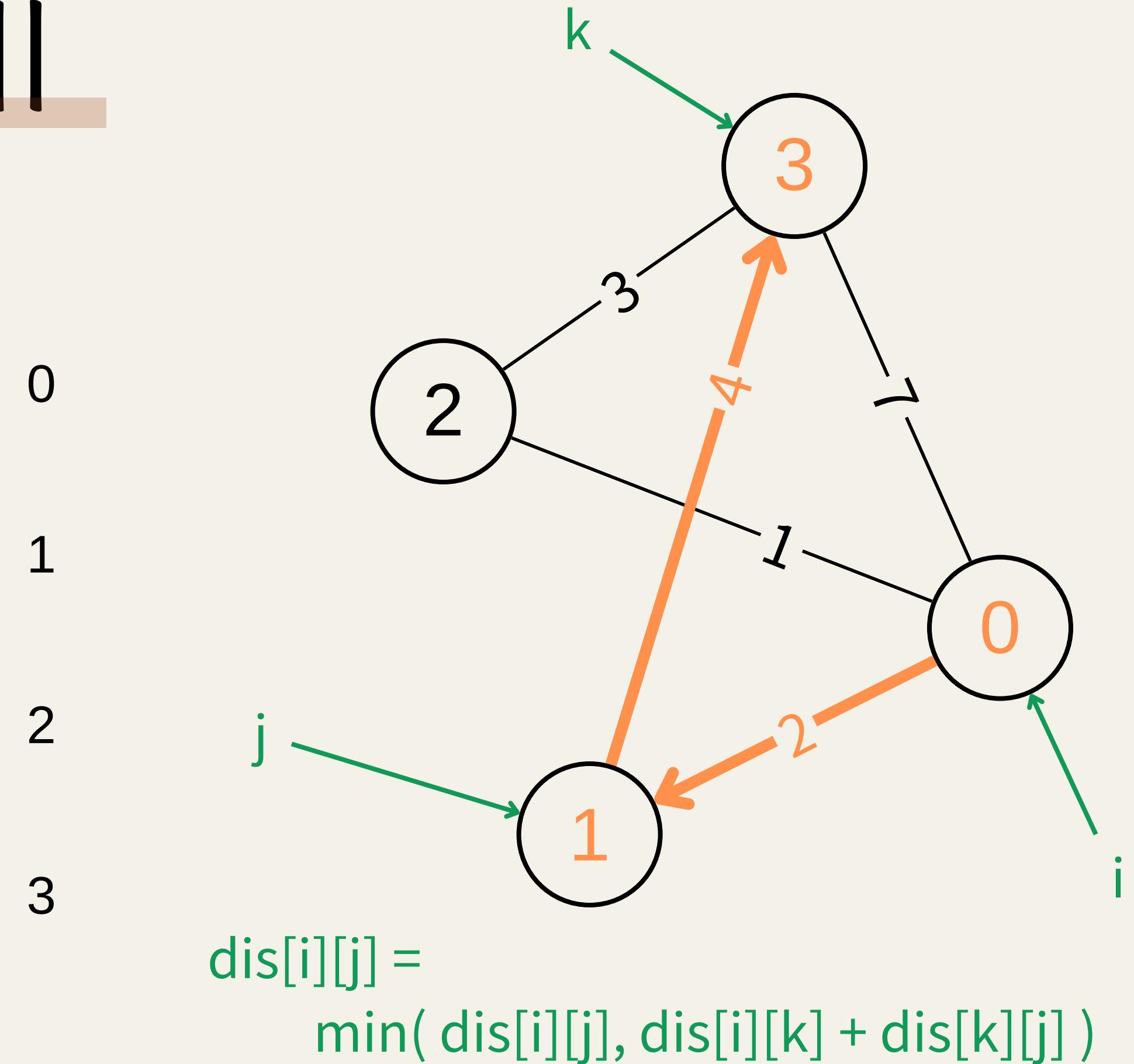
0
1
2
3



$$dis[i][j] = \min(dis[i][j], dis[i][k] + dis[k][j])$$

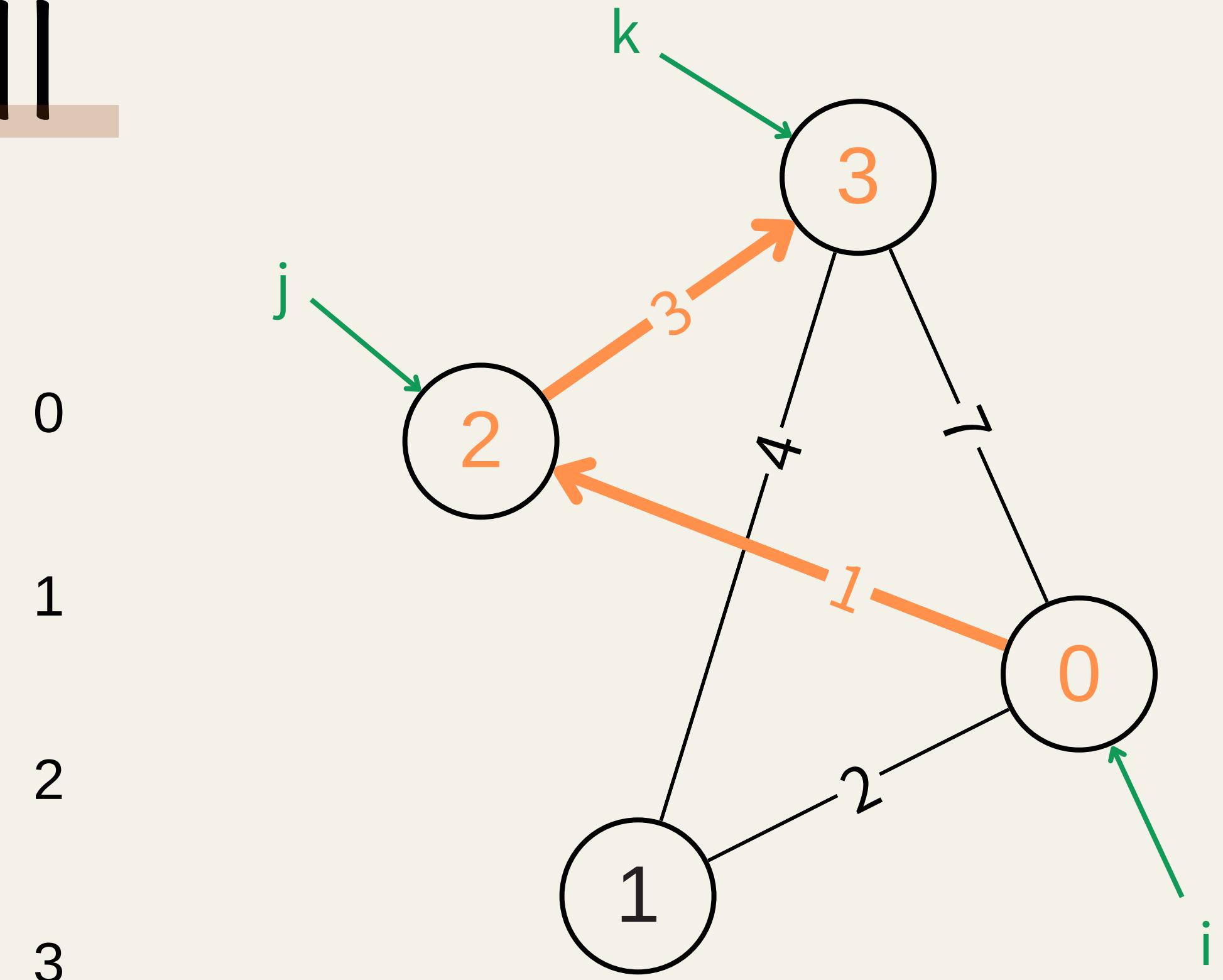
Floyd-Washall

index	0	1	2	3
dis[][]	0	2	1	6
	2	0	INF	4
	1	INF	0	3
	6	4	3	0



Floyd-Washall

index	0	1	2	3
dis[][]	0	2	1	4
	2	0	INF	4
	1	INF	0	3
	4	4	3	0

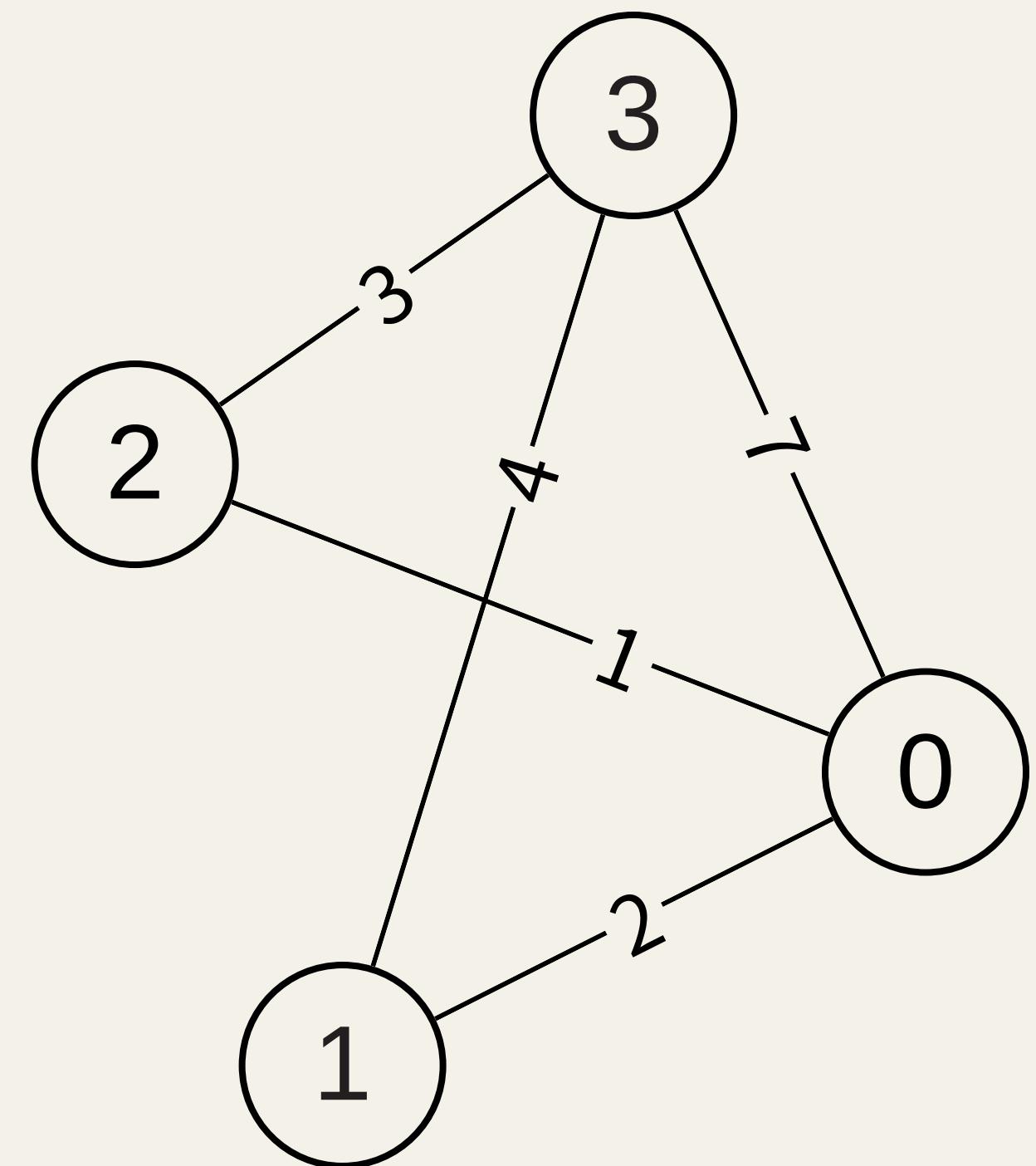


$$dis[i][j] = \min(dis[i][j], dis[i][k] + dis[k][j])$$

Floyd-Washall

index	0	1	2	3
dis[][]	0	2	1	4
	2	0	INF	4
	1	INF	0	3
	4	4	3	0

0
1
2
3





COMPARISON

	Time Complexity	Shortest Path Type
BFS	$O(V+E)$ or $O(V^2)$	Single Source
DFS	$O(V+E)$ or $O(V^2)$	Single Source
Bellman-Ford	$O(VE)$	Single Source
Dijkstra	$O(E \log_2 E)$	Single Source
Floyd-Warshall	$O(V^3)$	Mutiple Source

	Detect Negative Cycle	Data Structure Used
BFS	Not Available	Queue 、 Vector / Array
DFS	Not Available	Vector / Array
Bellman-Ford	Available	Vector / Array
Dijkstra	Not Available	PQueue 、 Vector / Array
Floyd-Warshall	Available	Vector / Array

Implementation

BFS

```
create a queue Q  
mark v as visited and put v into Q  
while Q is non-empty  
    remove the head u of Q  
    mark all (unvisited) neighbours of u  
    enqueue all (unvisited) neighbours of u
```

DFS

```
DFS(G, u)  
    u.visited = true  
    for each v ∈ G.Adj[u]  
        if v.visited == false  
            DFS(G,v)  
  
init() {  
    For each u ∈ G  
        u.visited = false  
    For each u ∈ G  
        DFS(G, u)  
}
```

Dijkstra

```
function Dijkstra ( Graph, source )
```

```
    for each vertex v in Graph.Vertices
```

```
        dist[v]  $\leftarrow$  INFINITY
```

```
        prev[v]  $\leftarrow$  UNDEFINED
```

```
        add v to Q
```

```
        dist[source]  $\leftarrow$  0
```

```
    while Q is not empty
```

```
        u  $\leftarrow$  vertex in Q with min dist[u]
```

```
        remove u from Q
```

```
    for each neighbor v of u still in Q:
```

```
        alt  $\leftarrow$  dist[u] + Graph.Edges(u, v)
```

```
        if alt < dist[v]:
```

```
            dist[v]  $\leftarrow$  alt
```

```
            prev[v]  $\leftarrow$  u
```

```
    return dist[], prev[]
```

Bellman-Ford

```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        distance[S] <- 0

    for each vertex V in G
        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        if distance[U] + edge_weight(U, V) < distance[V]
            Error: Negative Cycle Exists

    return distance[], previous[]
```

Eloyd-Marschal

```
n = no of vertices  
A = matrix of dimension n*n  
for k = 1 to n  
    for i = 1 to n  
        for j = 1 to n  
            A[i, j] = min (A[i, j], A[i, k] + A[k, j])  
return A
```

A photograph of a traditional Japanese garden featuring a long, narrow wooden walkway made of planks. The walkway is flanked by dark wooden railings and leads towards a small, ornate lantern at its far end. The scene is bathed in warm, golden light, likely from the setting sun, which creates long shadows and a peaceful atmosphere.

Thank You

For Tuning In