COSC 222 Data Structure

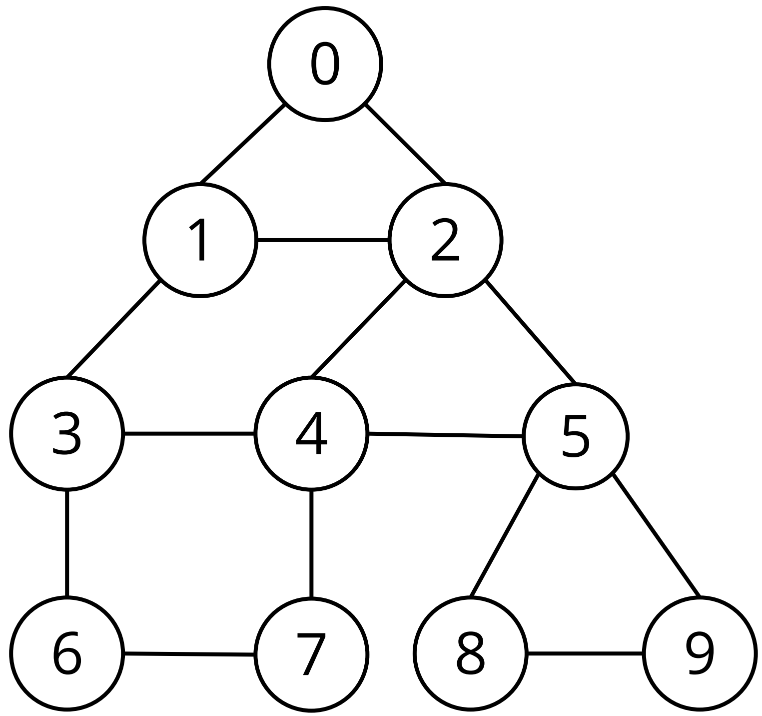
Graphs – Part 2

Depth-first search

- **Depth-first search (DFS)**: Finds a path between two vertices by exploring each possible path as far as possible before backtracking
  - Often implemented recursively.
  - Many graph algorithms involve visiting or marking vertices.

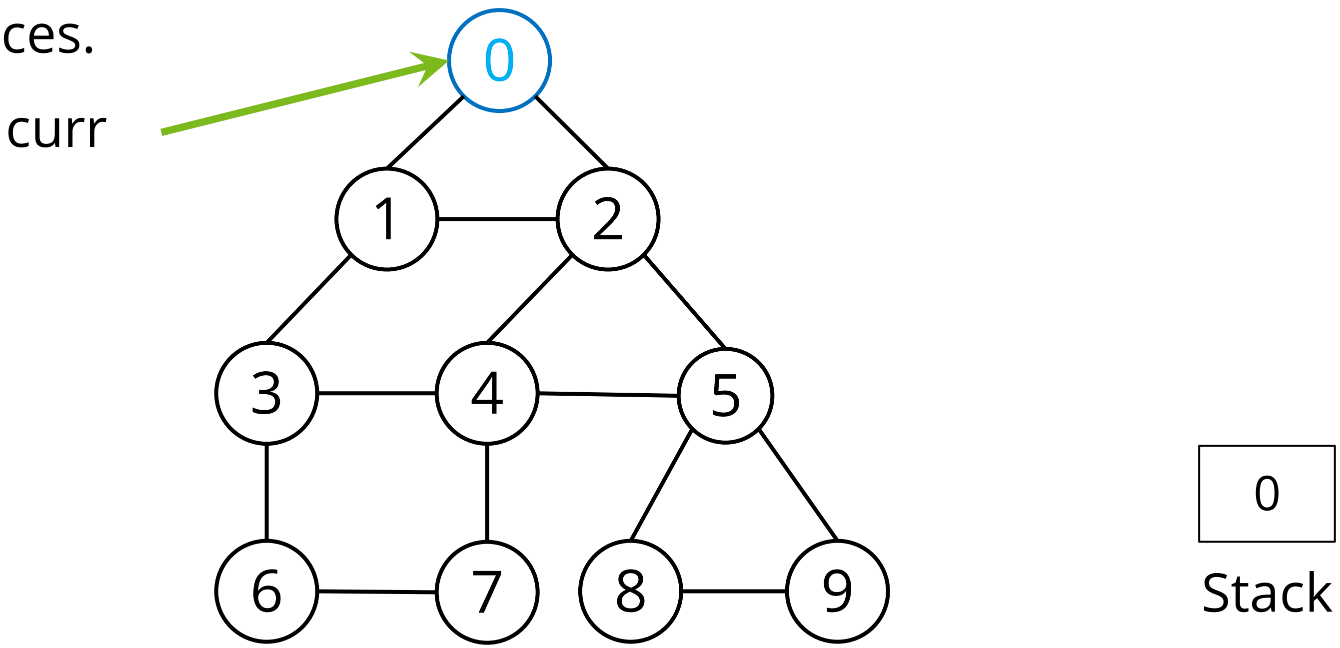
Depth-first search (DFS) example

- We will traverse the above graph starting at 0, with all vertices currently unvisited.



Depth-first search (DFS) example

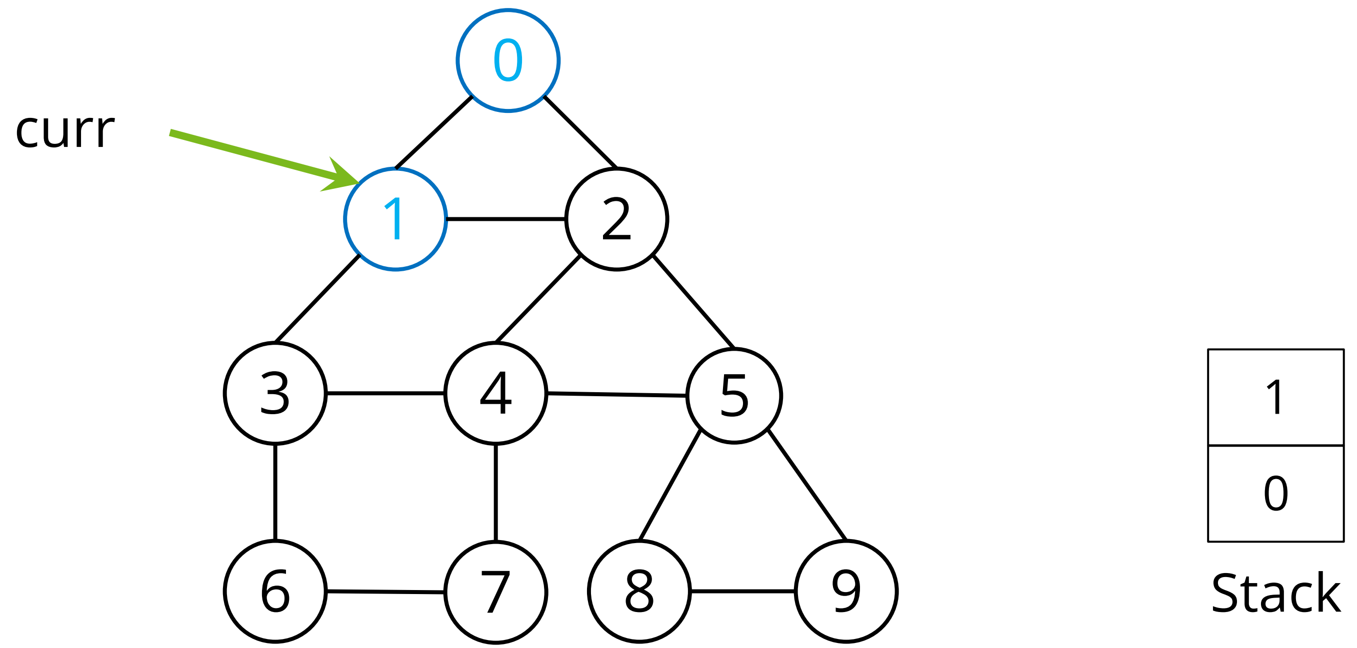
- To depth-first traverse at vertex 0:
  - Mark the current vertex (0) as visited, and then
  - Recursively depth-first traverse each of the adjacent unvisited vertices.



- Assume that we examine the adjacent vertices in sorted order (so we would first look at vertex 1, then at vertex 2). Vertex 1 is unvisited, so we next recursively depth-first traverse vertex 1.

Depth-first search (DFS) example

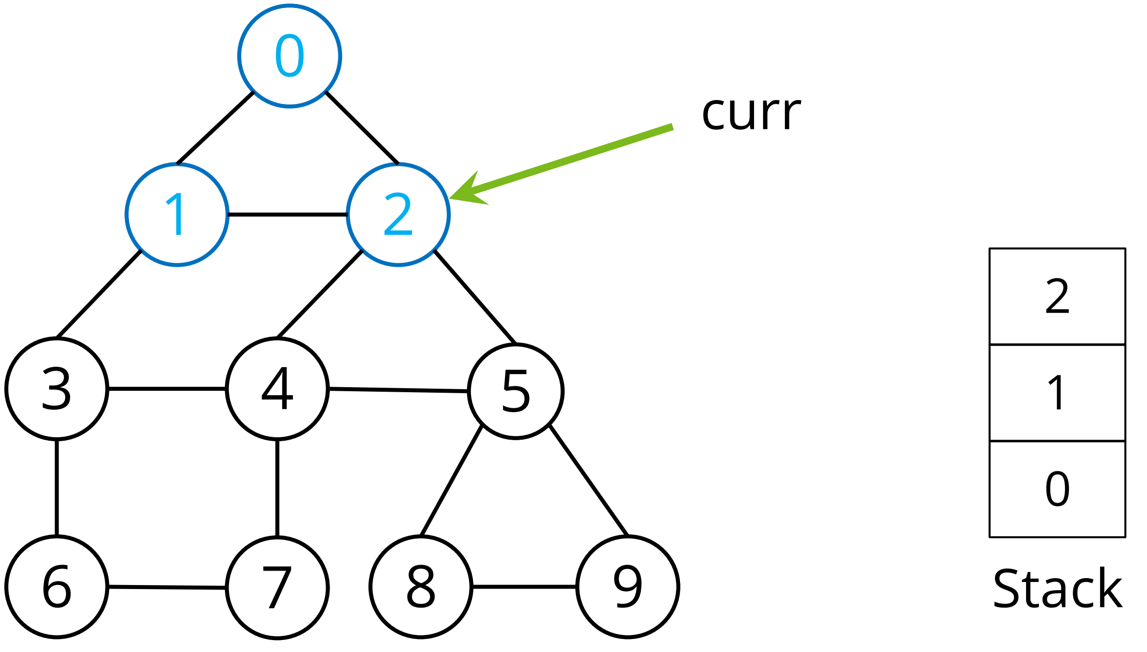
- Now mark the current vertex (1) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



- Adjacent vertex 2 is unvisited, so we next recursively depth-first traverse vertex 2.

Depth-first search (DFS) example

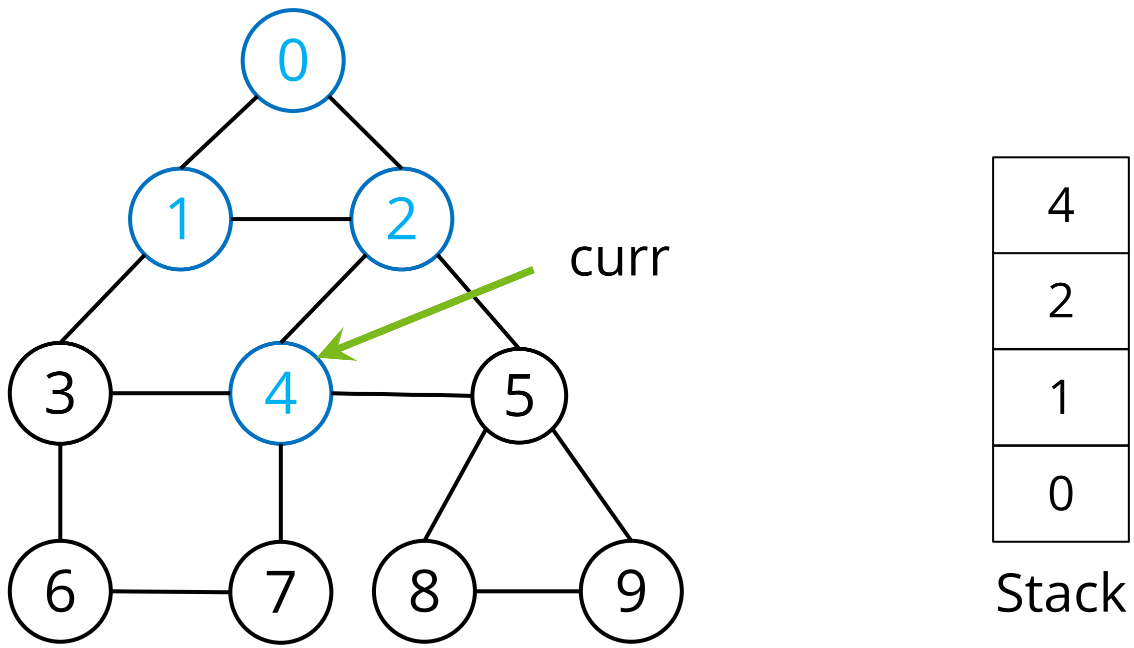
- Now mark the current vertex (2) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



- Adjacent vertex 4 is unvisited, so we next recursively depth-first traverse vertex 4.

Depth-first search (DFS) example

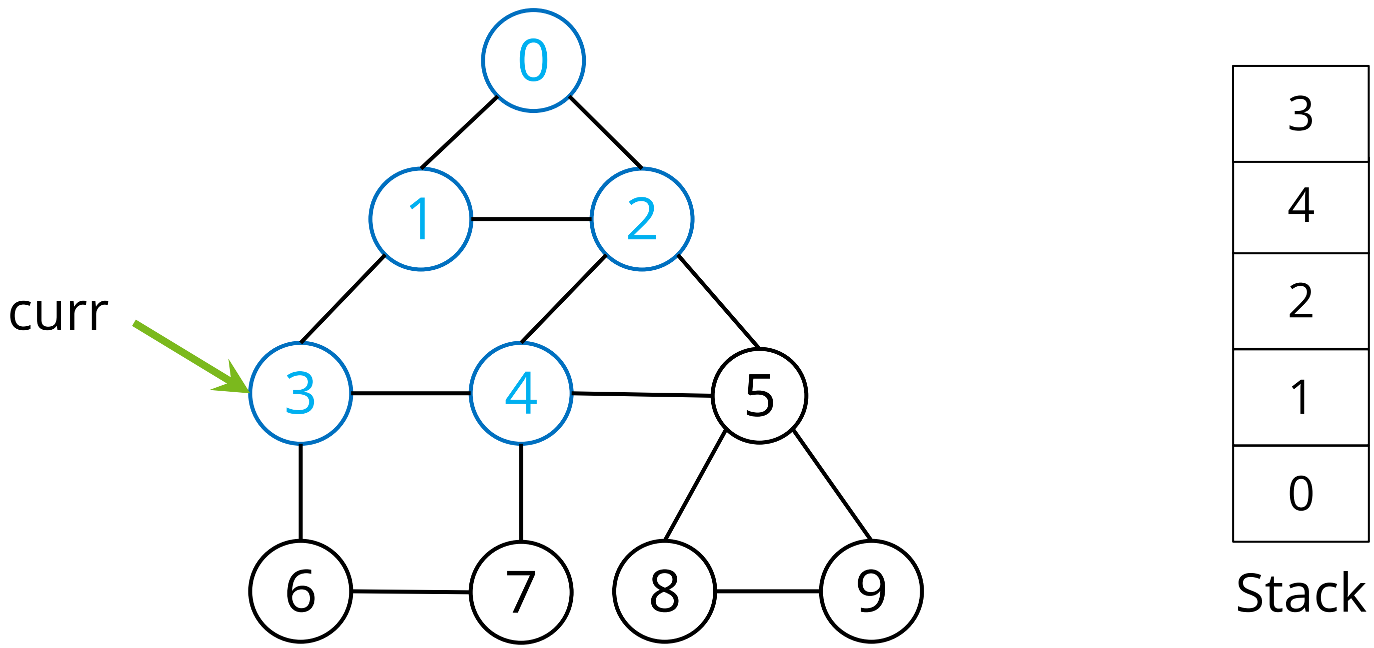
- Now mark the current vertex (4) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



- Adjacent vertex 3 is unvisited, so we next recursively depth-first traverse vertex 3.

Depth-first search (DFS) example

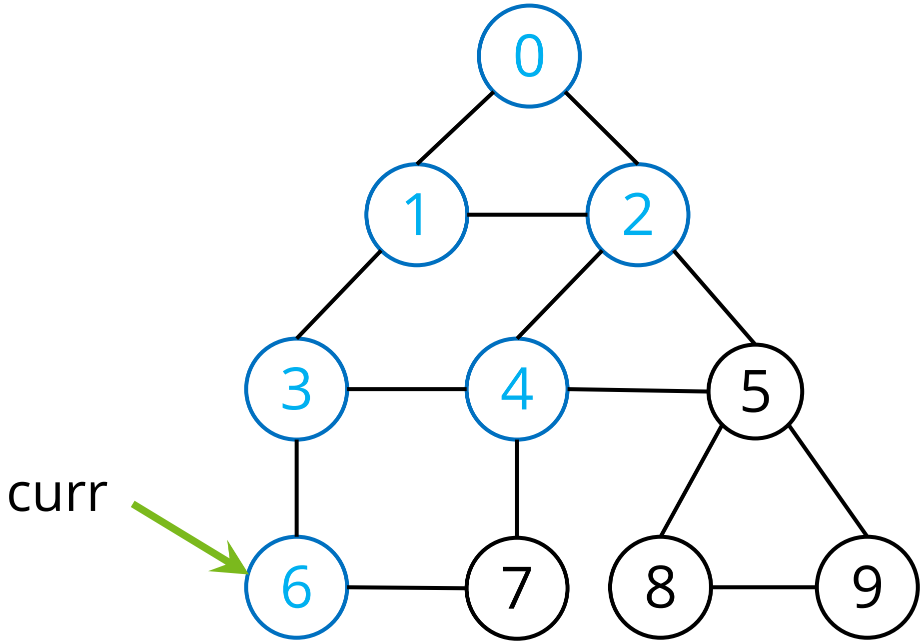
- Now mark the current vertex (3) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.

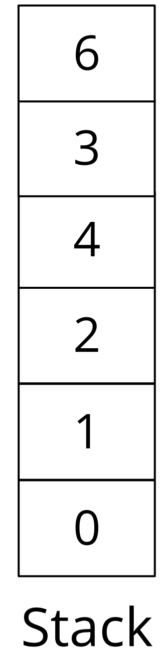


- Adjacent vertex 6 is unvisited, so we next recursively depth-first traverse vertex 6.

Depth-first search (DFS) example

- Now mark the current vertex (6) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.

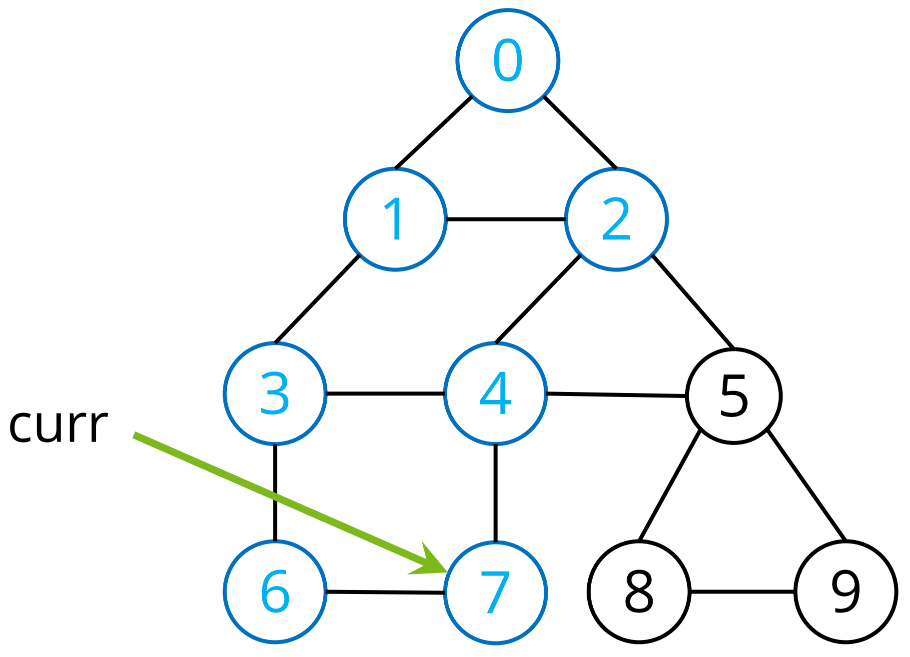


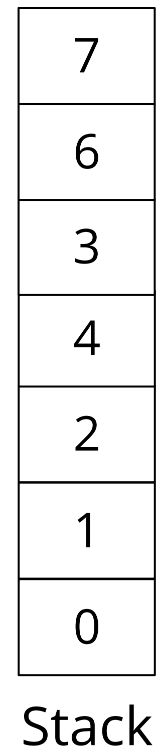


- Adjacent vertex 7 is unvisited, so we next recursively depth-first traverse vertex 7.

Depth-first search (DFS) example

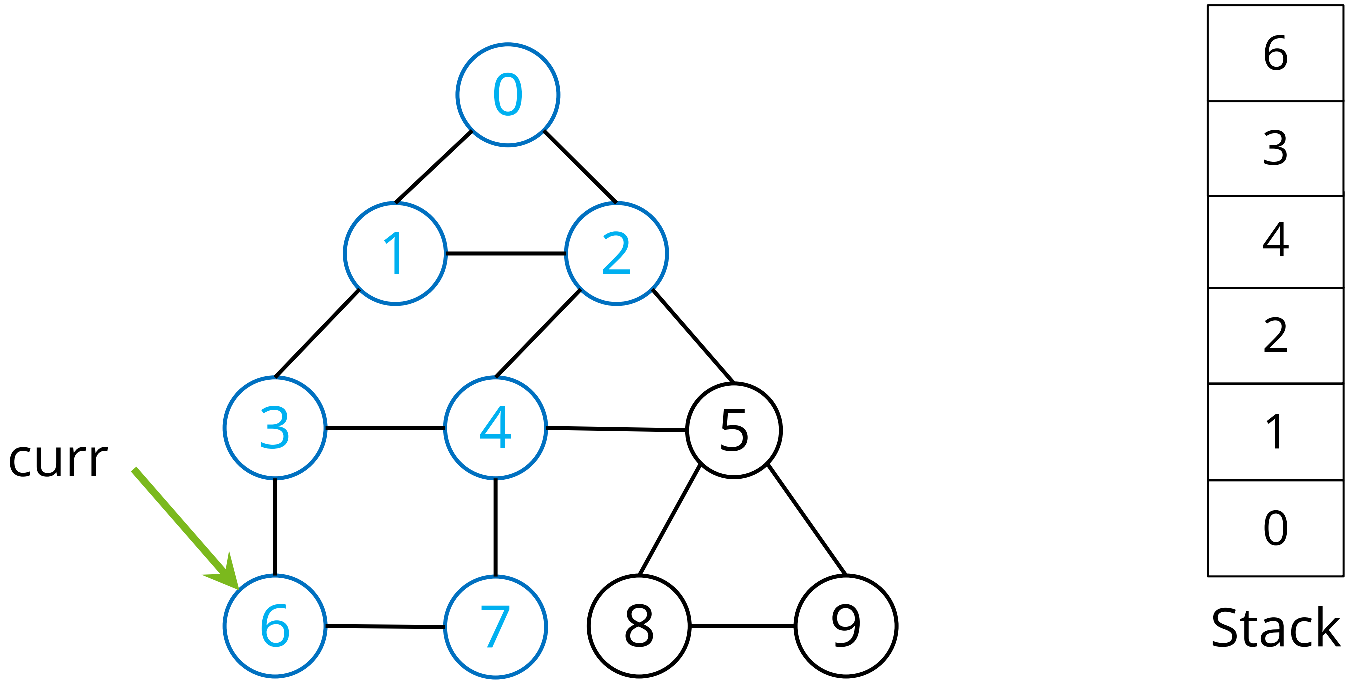
- Mark 7 as visited, and recursively depth-first traverse the adjacent unvisited vertices.





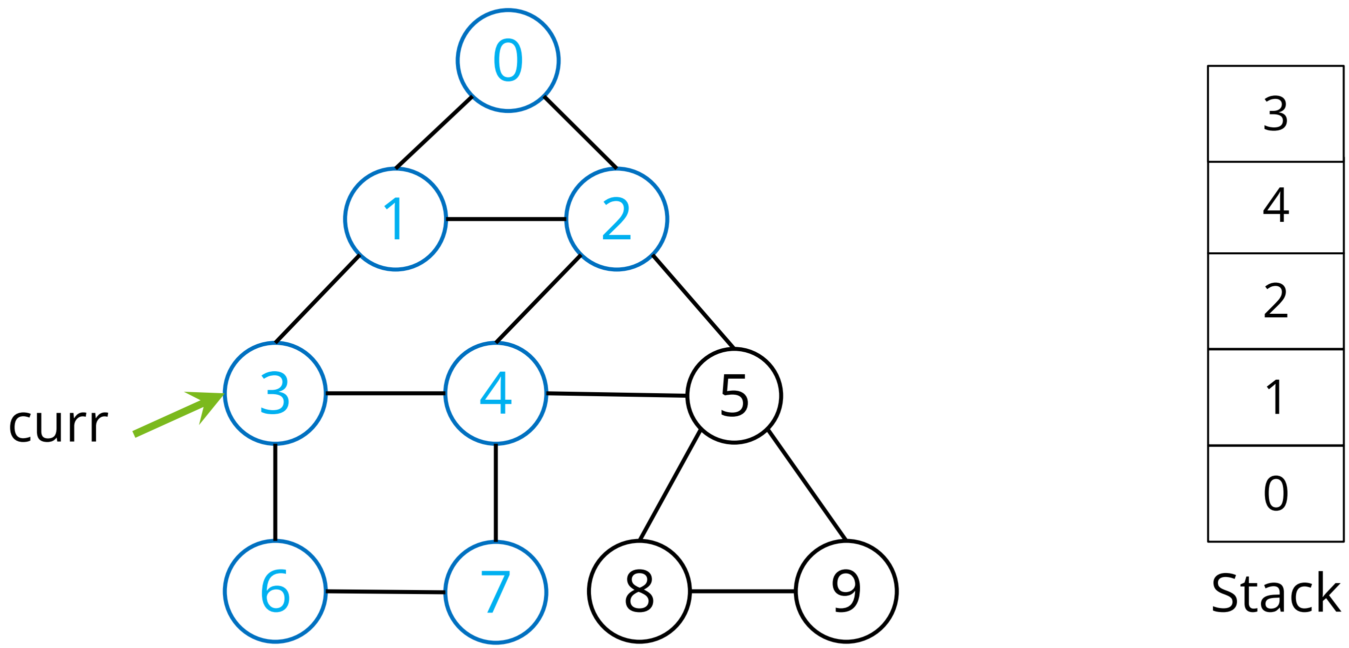
- **No** adjacent vertices are unvisited, so pop the stack to return to a previous vertex and look for unvisited adjacent vertices there.

Depth-first search (DFS) example



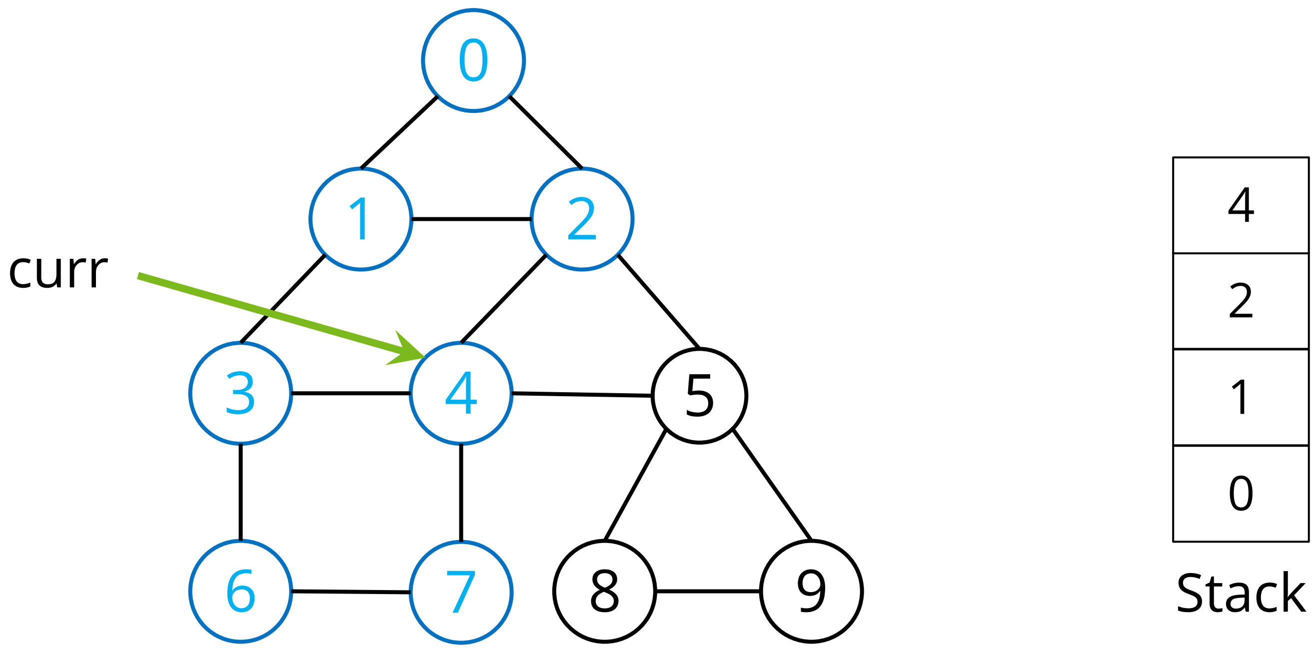
- No vertices adjacent to 6 are unvisited, so pop the stack again.

Depth-first search (DFS) example



- No vertices adjacent to 3 are unvisited, so pop the stack again.

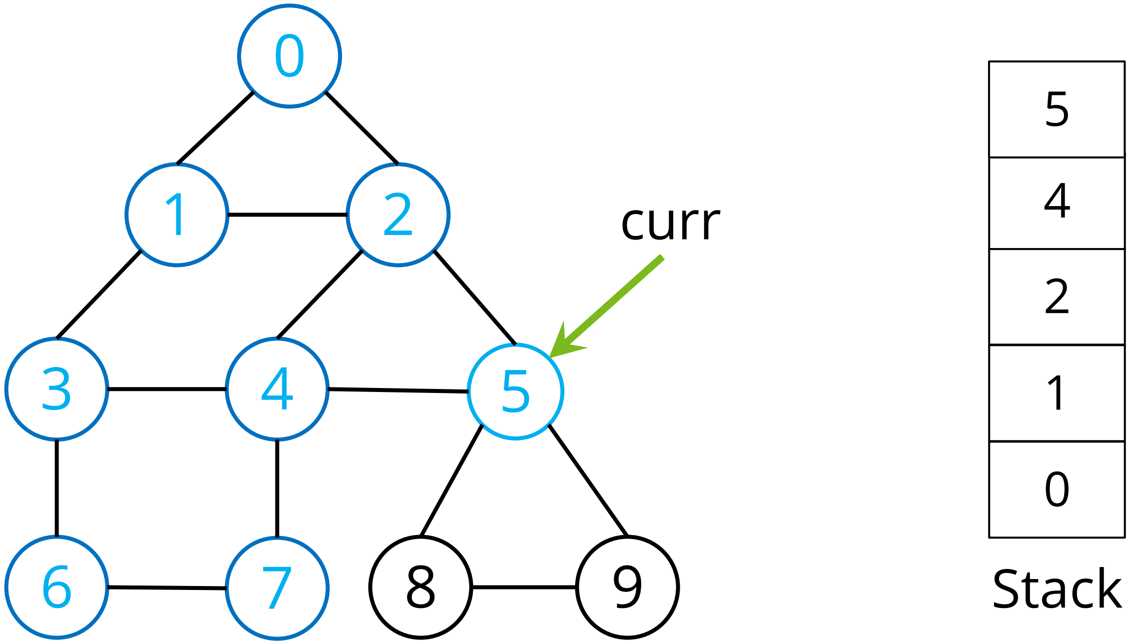
Depth-first search (DFS) example



- Vertex 5 is adjacent to 4 and is unvisited, so depth-first traverse 5

Depth-first search (DFS) example

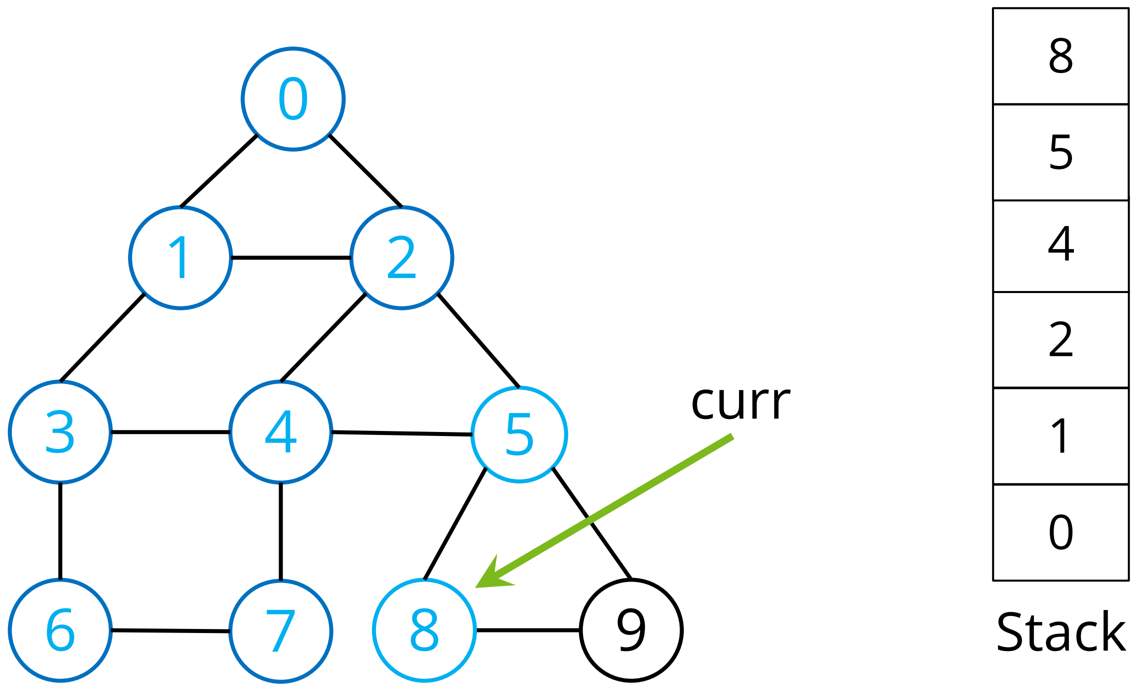
- Mark 7 as visited, and recursively depth-first traverse the adjacent unvisited vertices.



- Vertex 8 is adjacent to 5 and is unvisited, so depth-first traverse 8.

Depth-first search (DFS) example

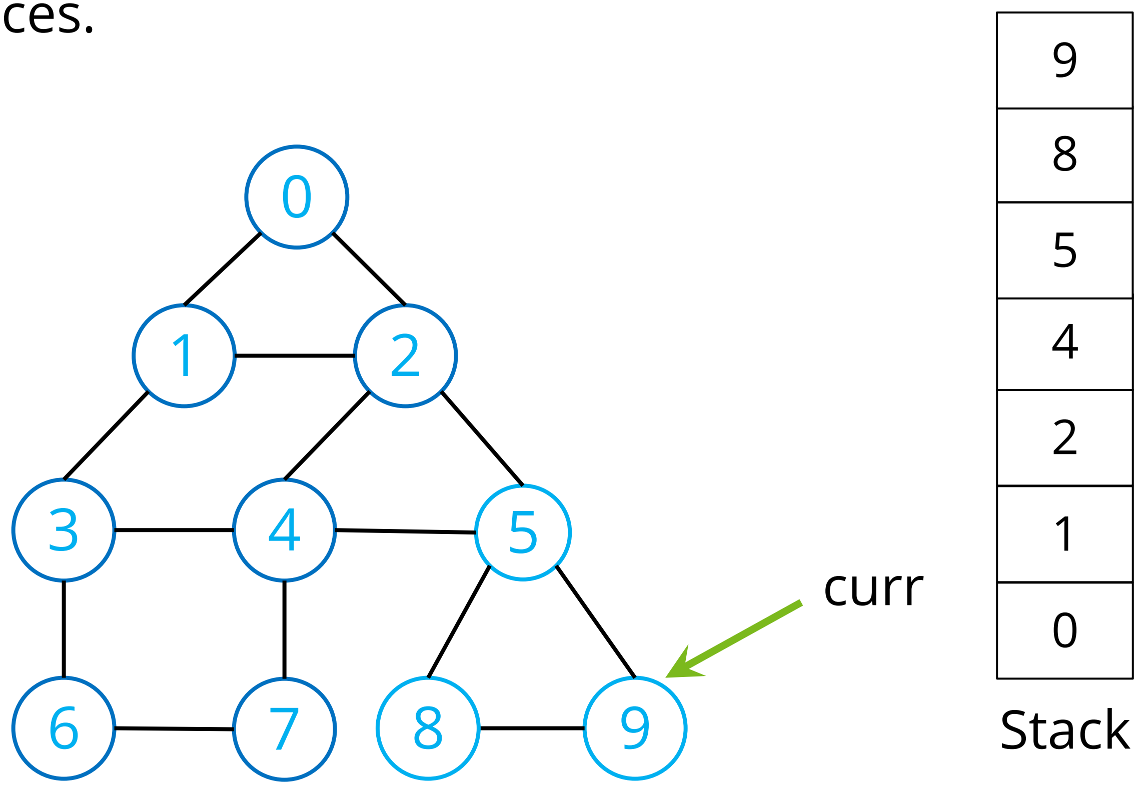
- Mark 8 as visited, and recursively depth-first traverse the adjacent unvisited vertices.



- Vertex 9 is adjacent to 8 and is unvisited, so depth-first traverse 9.

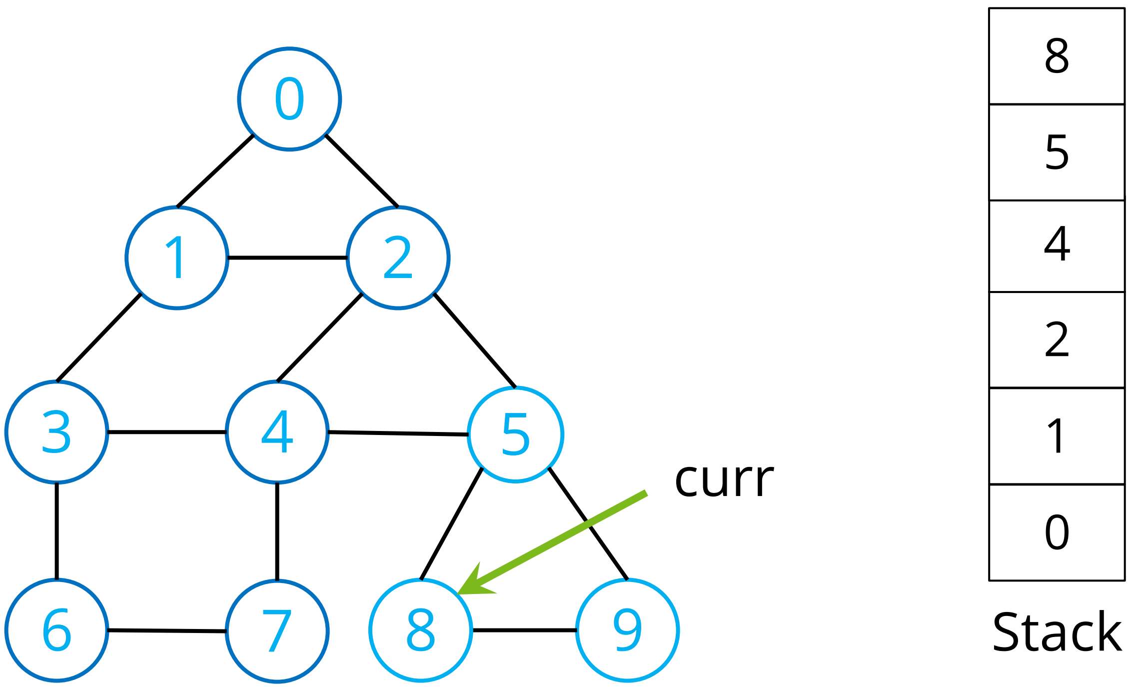
Depth-first search (DFS) example

- Mark 9 as visited, and recursively depth-first traverse the adjacent unvisited vertices.



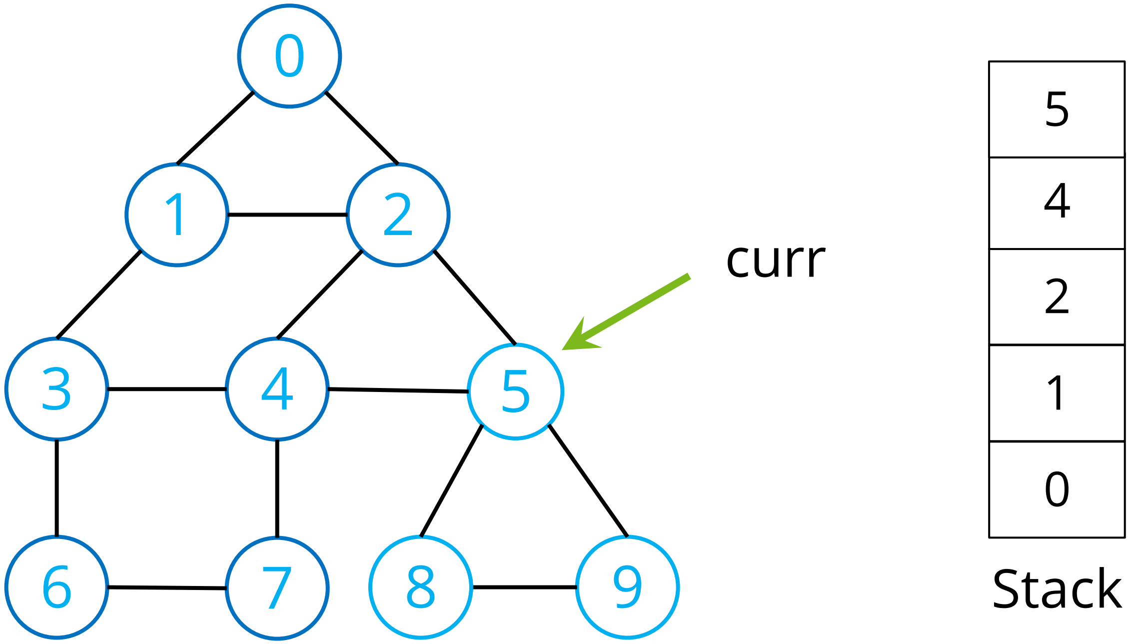
- No adjacent vertex is unvisited, so pop the stack.

Depth-first search (DFS) example



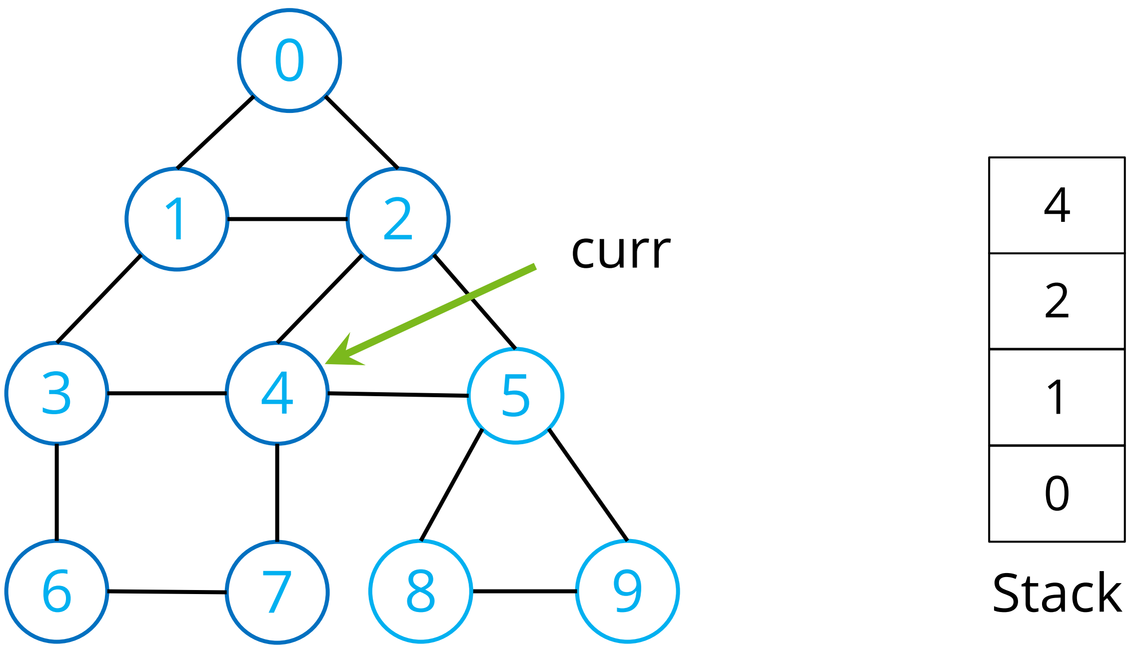
- No adjacent vertex is unvisited, so pop the stack.

Depth-first search (DFS) example



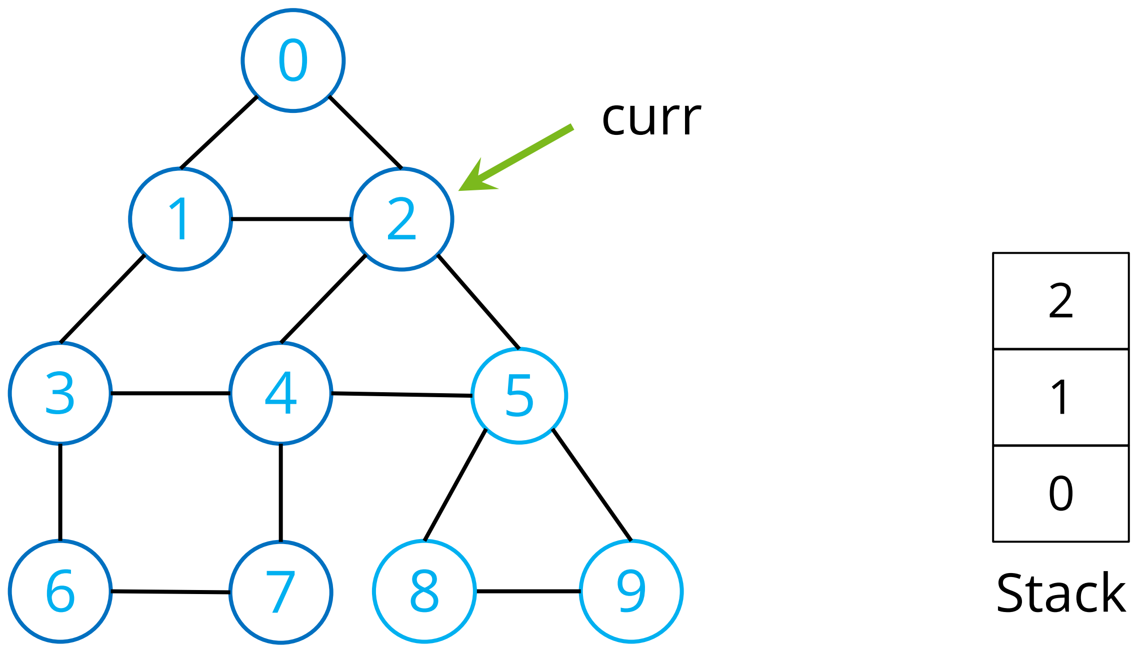
- No adjacent vertex is unvisited, so pop the stack.

Depth-first search (DFS) example



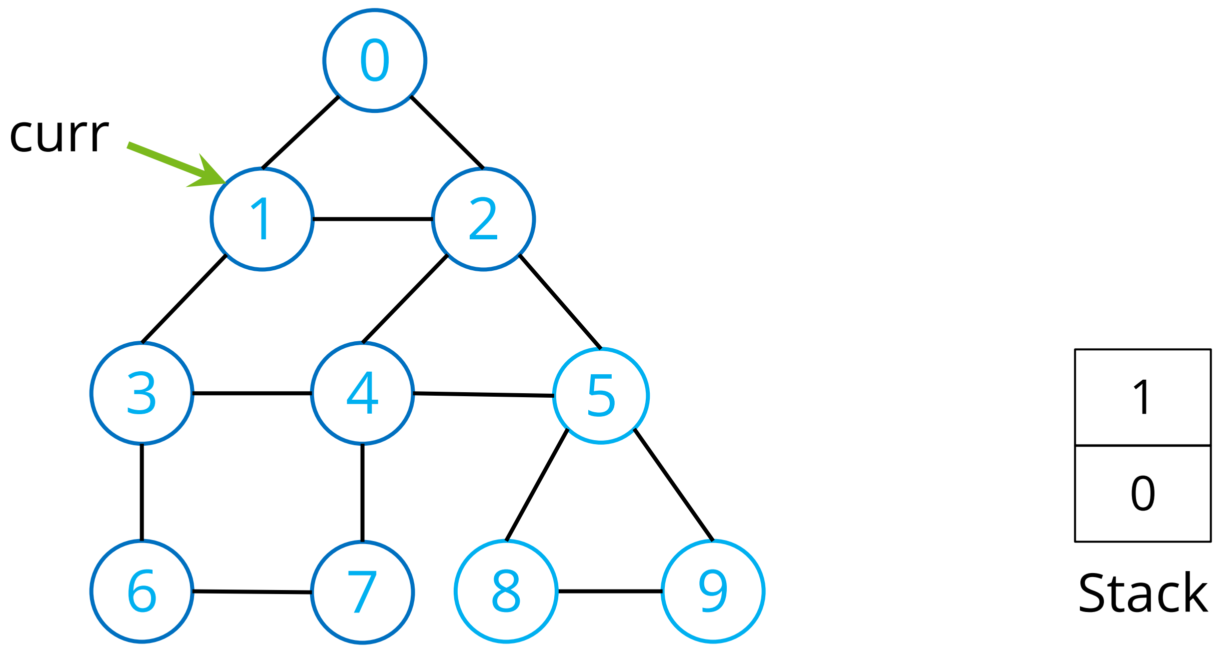
- No adjacent vertex is unvisited, so pop the stack.

Depth-first search (DFS) example



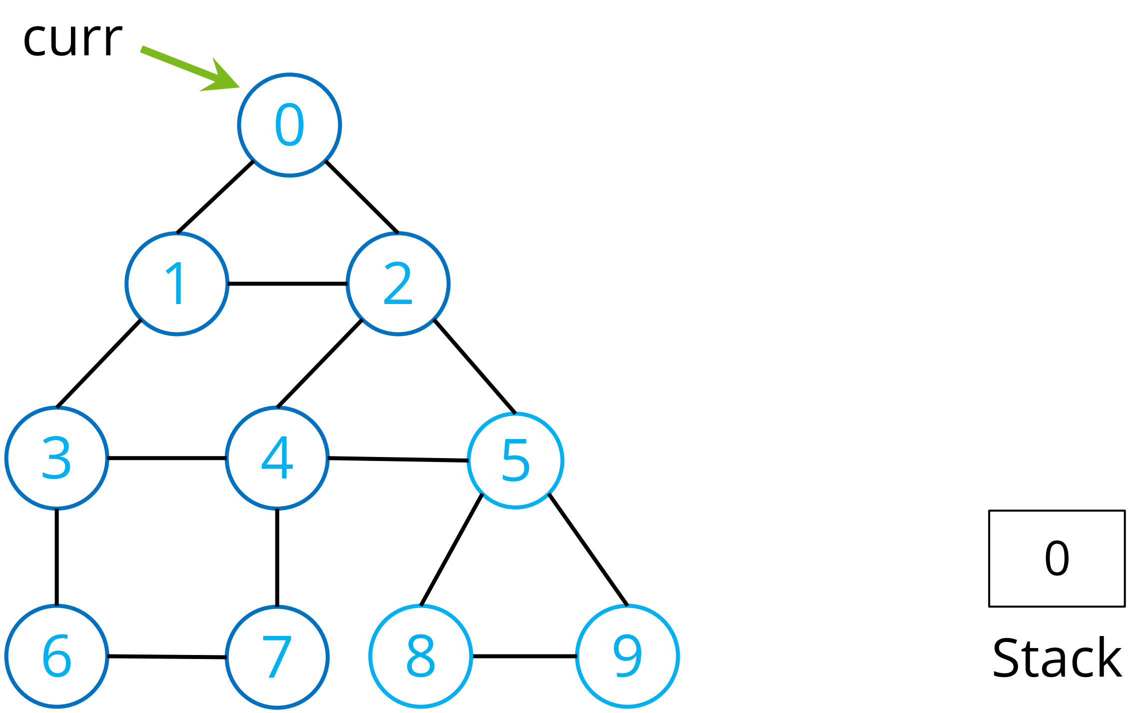
- No adjacent vertex is unvisited, so pop the stack.

Depth-first search (DFS) example



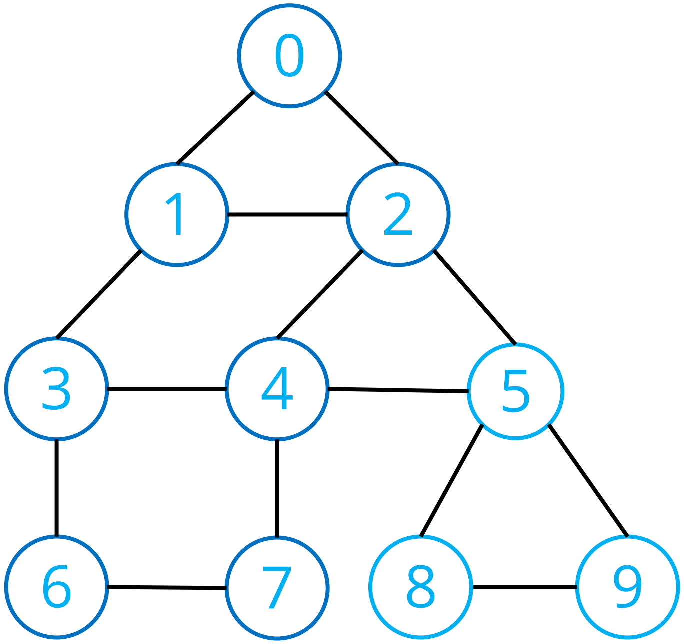
- No adjacent vertex is unvisited, so pop the stack.

Depth-first search (DFS) example



- No adjacent vertex is unvisited, so pop the stack.

Depth-first search (DFS) example

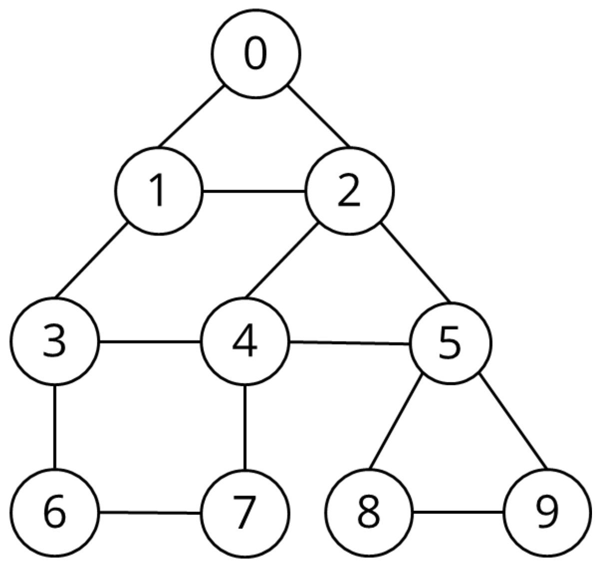


Stack

- Now the stack is empty, so we have traversed the whole graph.

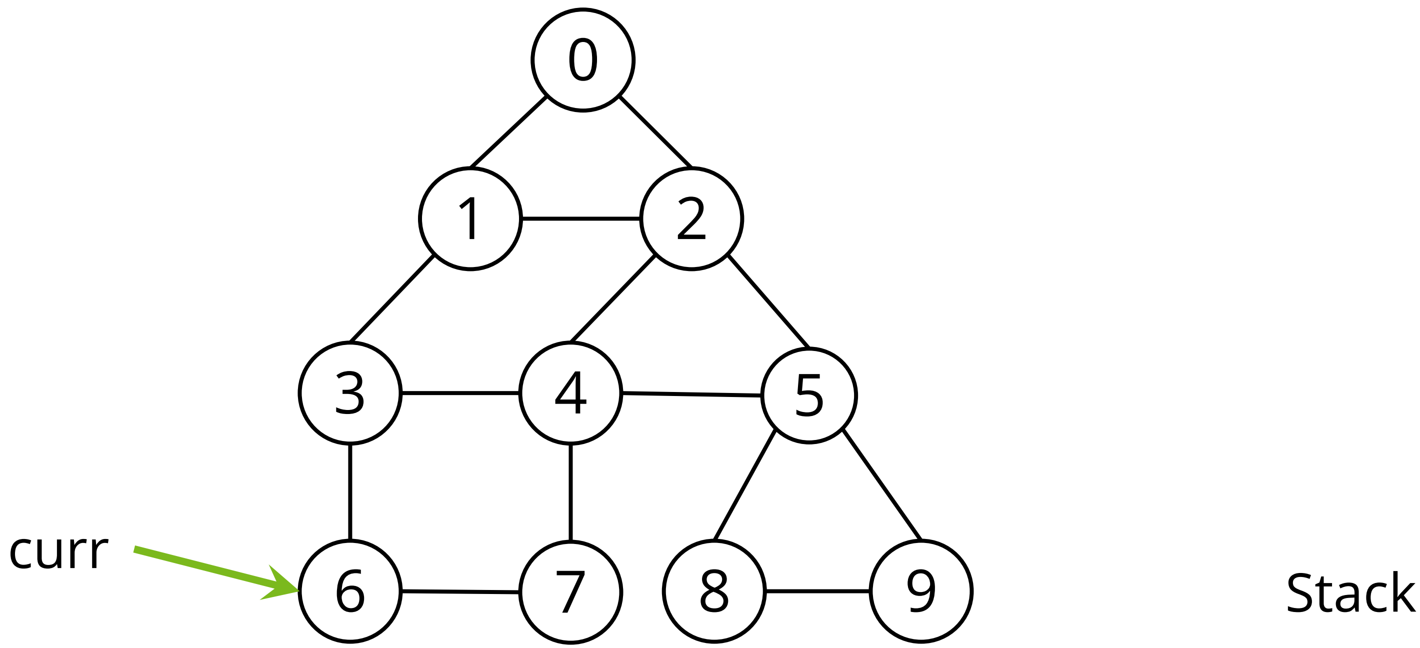
Printing in a Depth-First Traversal

- We “visit" a vertex when we mark it as visited.
- In our example, we did nothing when we visited a vertex.
- If we print out the contents of the vertex when we visit it, then the output would be
  - 0, 1, 2, 4, 3, 6, 7, 5, 8, 9



Depth-first Traversal: Example

- What would the output be if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



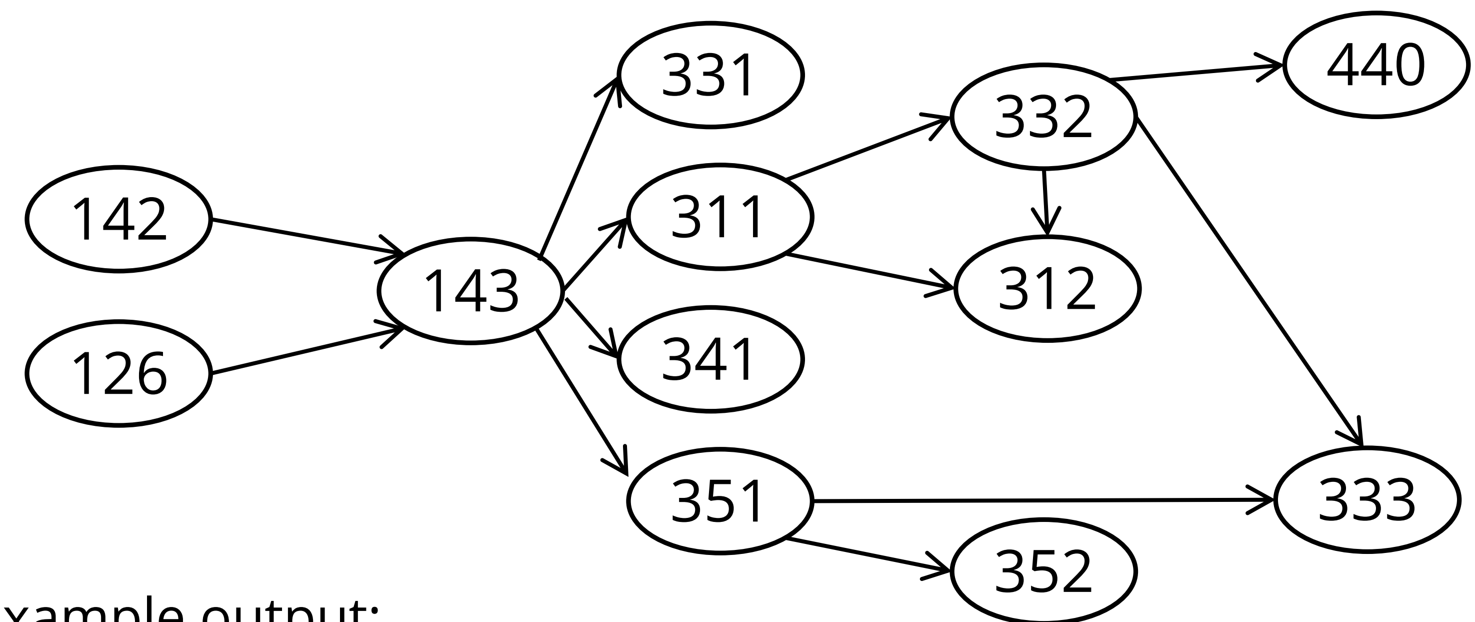
- Answer: 6, 3, 1, 0, 2, 4, 5, 8, 9, 7

Depth-first Traversal Pseudocode

```
public void depthFirstTraveral(vertex curr) {
      mark vertex curr as visited;
      visit curr (e.g., print);
      for each vertex v adjacent to curr
         if v is unvisited
            depthFirstTraversal(v);
} // end depthFirstTraversal
```

Topological Sort

- Problem: Given a DAG **G=(V,E)**, output all the vertices in order such that if no vertex appears before any other vertex that has an edge to it. Only possible for a directed acyclic graph.
- Example input: course prerequisites

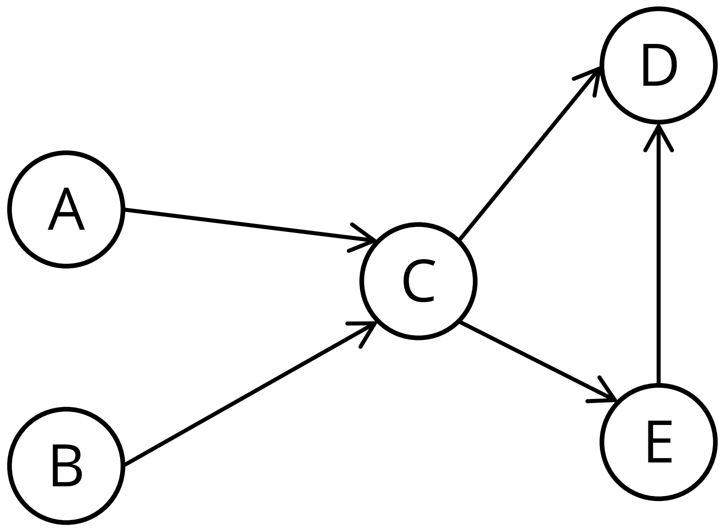


- Example output:

142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352

Topological sort

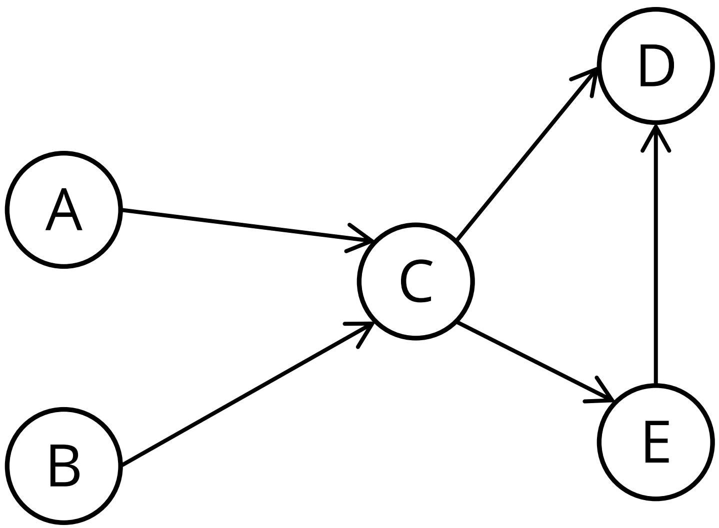
- **Setup**
  - Look at each vertex and record its in-degree somewhere



Node:	A	B	C	D	E
In-degree:	0	0	2	2	1

Topological sort

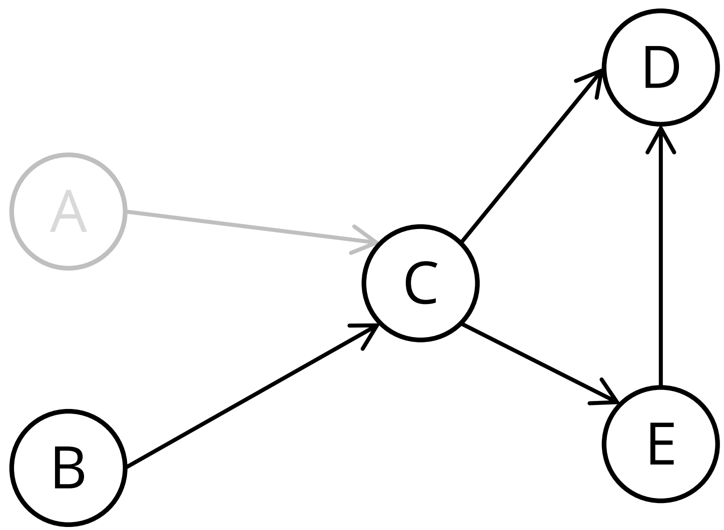
- **Core loop**
  - Choose an arbitrary vertex “a” with in-degree 0
  - Output “a” and conceptually remove it from the graph
  - For each vertex “b” adjacent to “a”, decrement the in-degree of “b”
  - Repeat



Node:	A	B	C	D	E
In-degree:	0	0	2	2	1

Topological sort

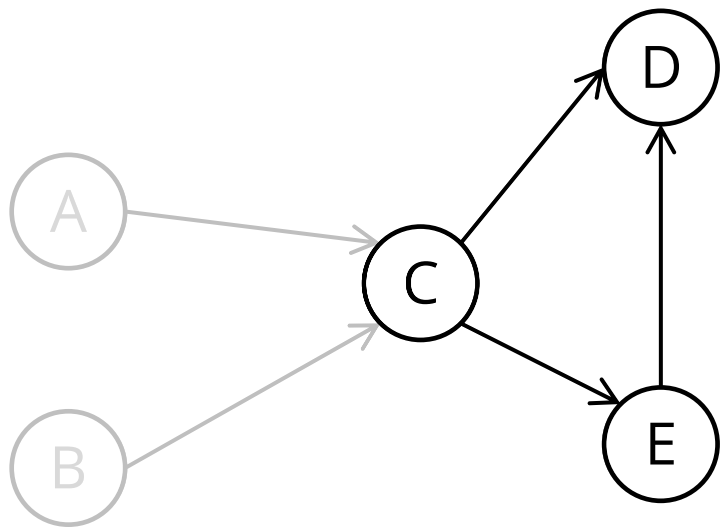
- **Core loop**
  - Choose an arbitrary vertex "a" with in-degree 0
  - Output "a" and conceptually remove it from the graph
  - For each vertex "b" adjacent to "a", decrement the in-degree of "b"
  - Repeat



Node:	A	B	C	D	E
In-degree:	0	0	1	2	1
Remove?	X				

Topological sort

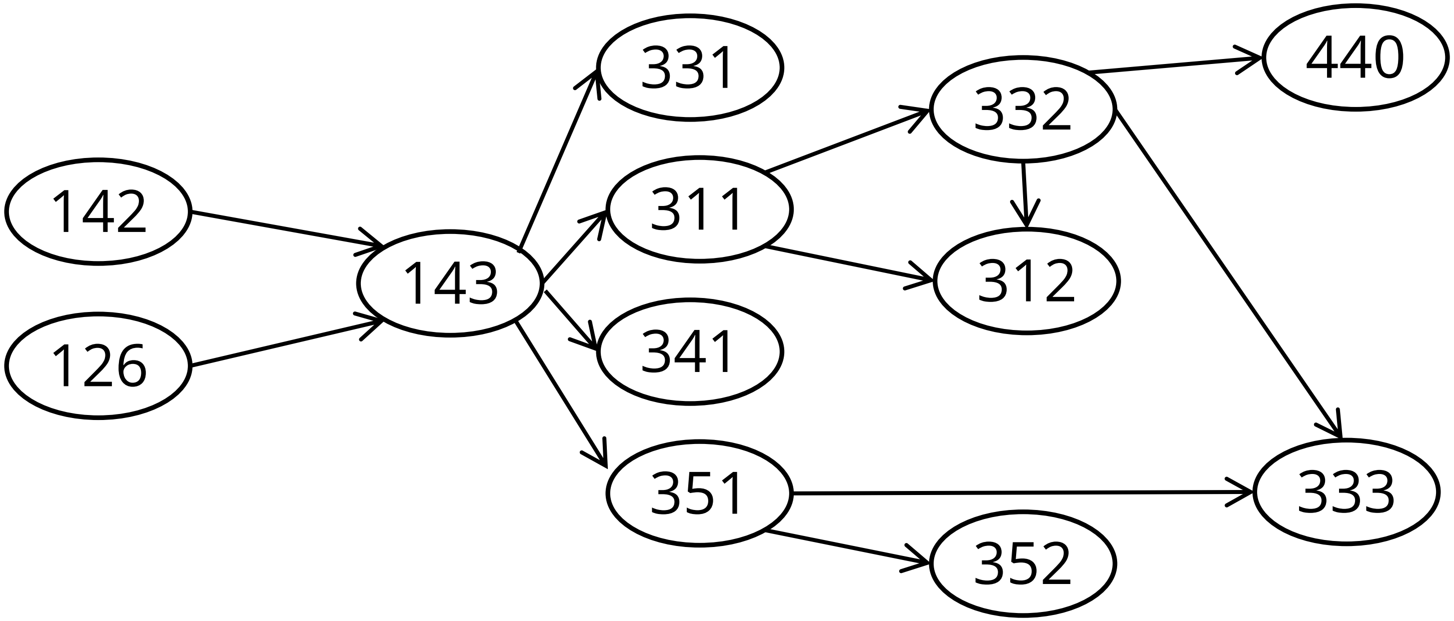
- **Core loop**
  - Choose an arbitrary vertex “a” with in-degree 0
  - Output “a” and conceptually remove it from the graph
  - For each vertex “b” adjacent to “a”, decrement the in-degree of “b”
  - Repeat



Node:	A	B	C	D	E
In-degree:	0	0	0	2	1
Remove?	X	X			

Example

- Output:

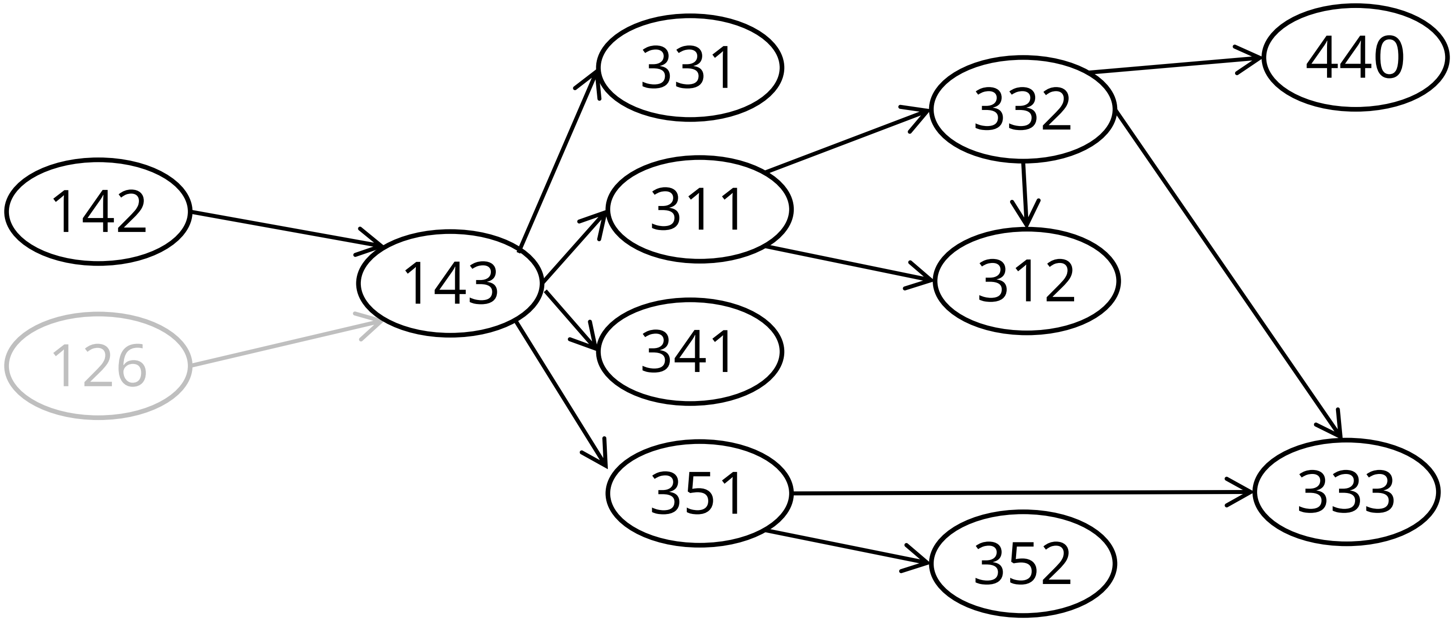


▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	2	1	2	1	1	2	1	1	1	1
▪ Removed?												

Example

- Output:

126

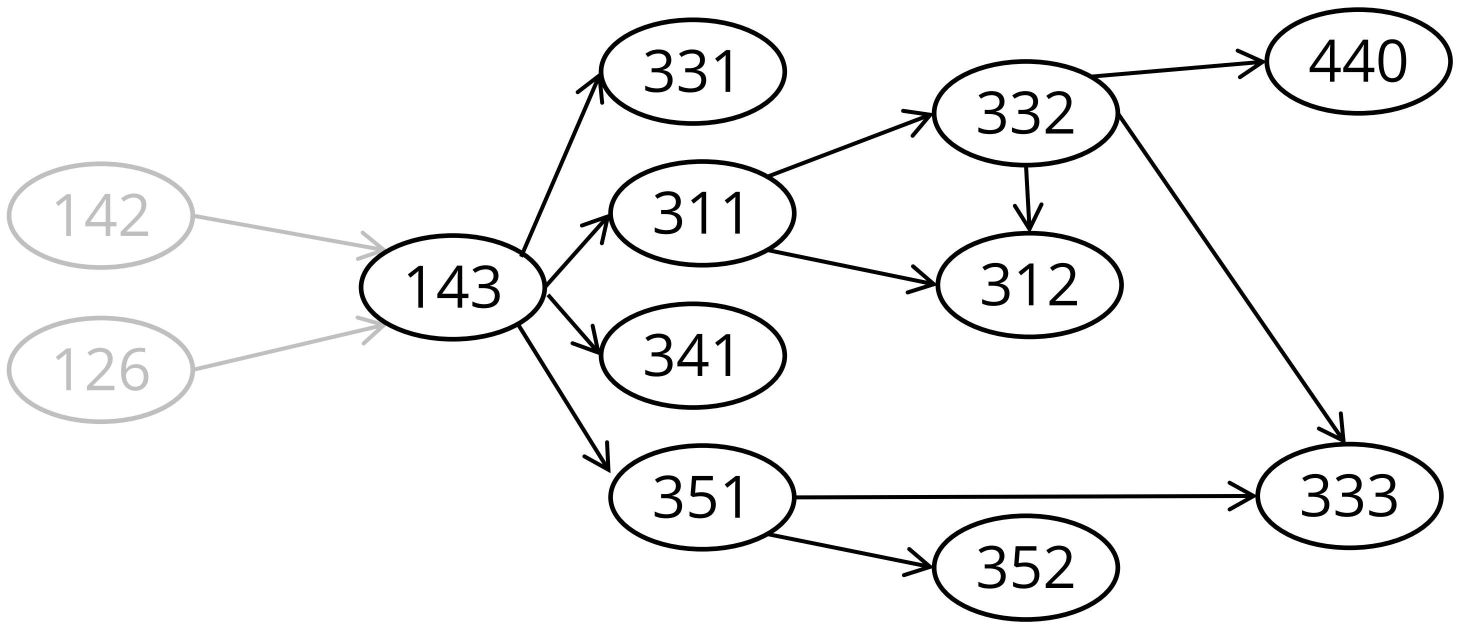


▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	1	1	2	1	1	2	1	1	1	1
▪ Removed?	X											

Example

- Output:

126
142

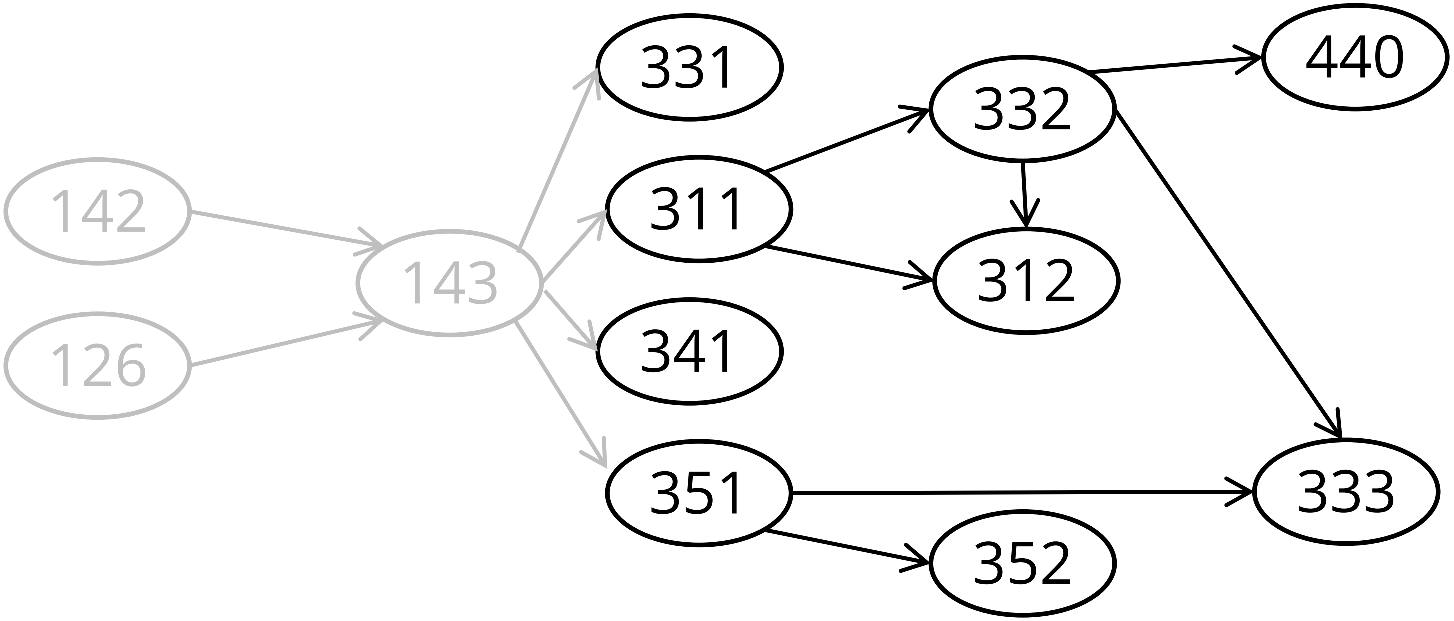


▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	0	1	2	1	1	2	1	1	1	1
▪ Removed?	X	X										

Example

- Output:

126
142
143

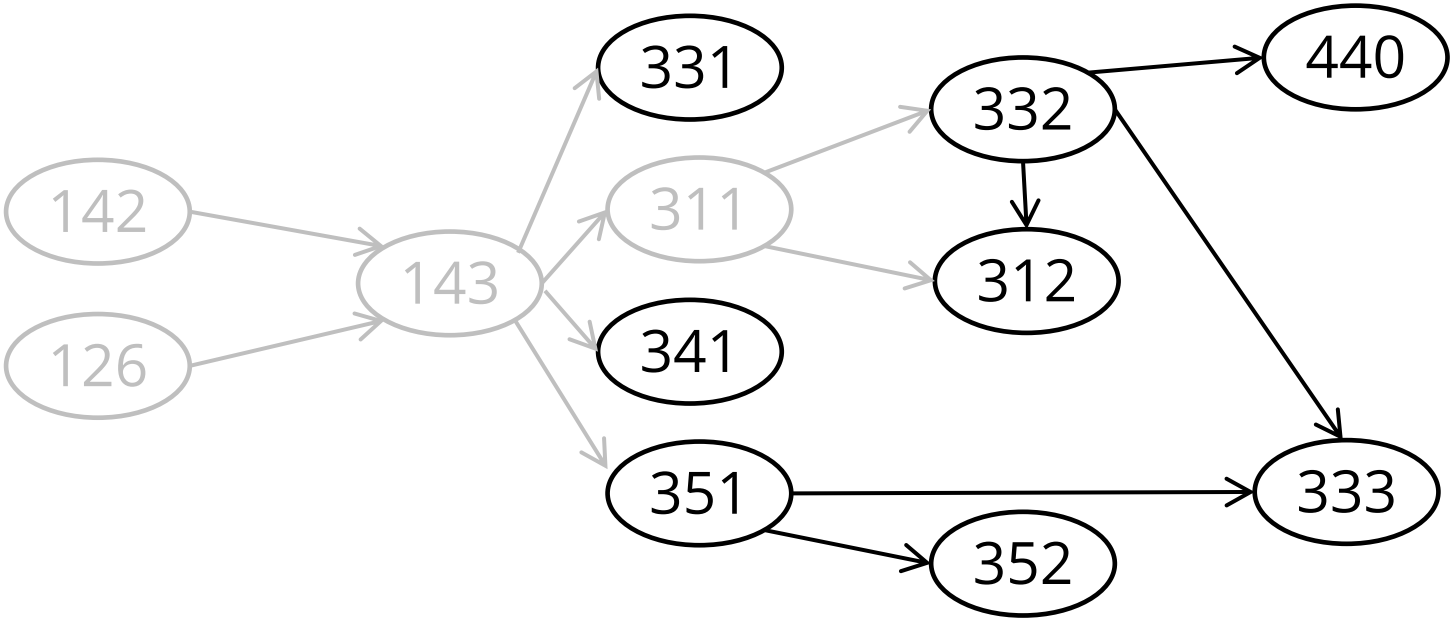


▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	0	0	2	0	1	2	0	0	1	1
▪ Removed?	X	X	X									

Example

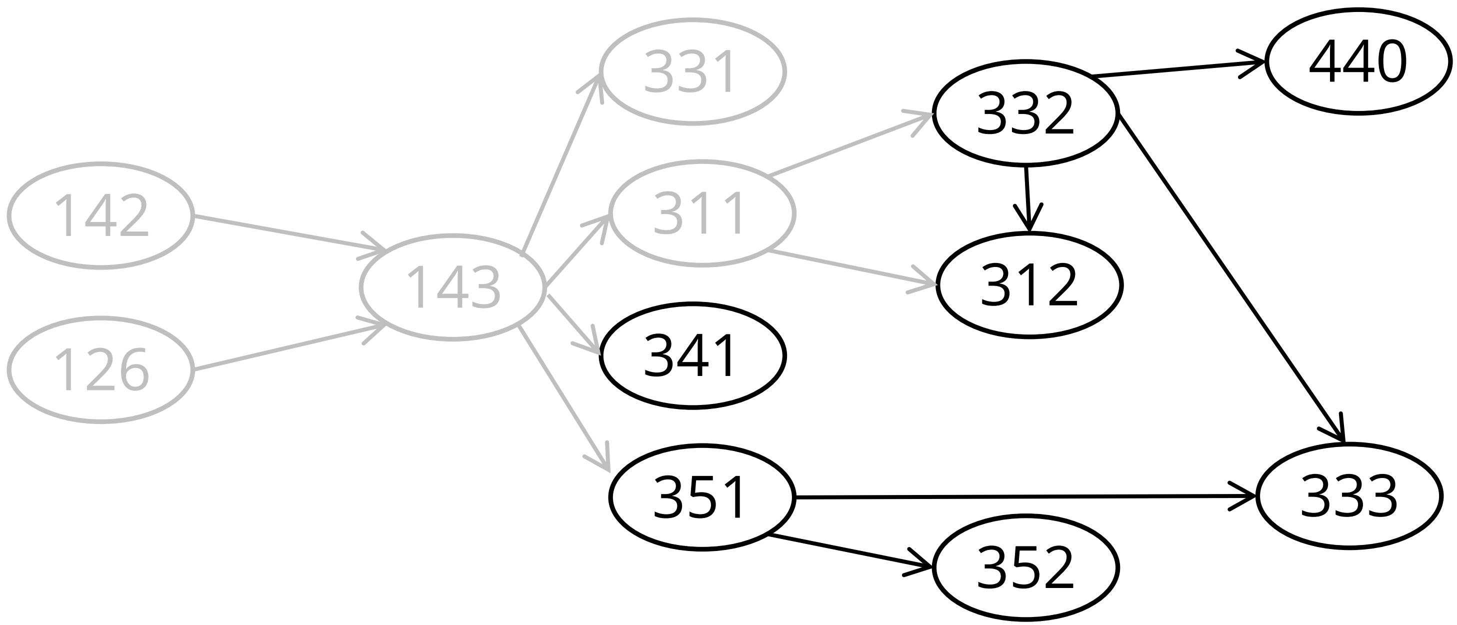
- Output:

126
142
143
311



▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	0	0	1	0	0	2	0	0	1	1
▪ Removed?	X	X	X	X								

Example

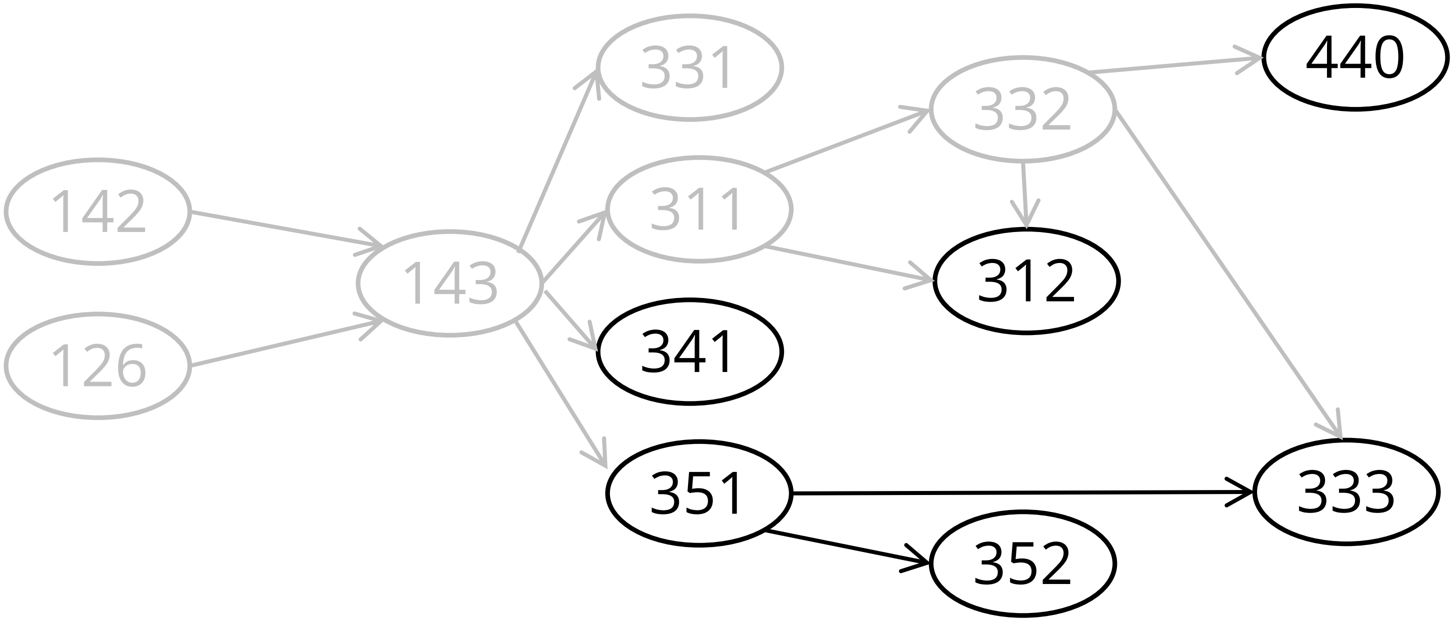


- Output:

126
142
143
311
331

▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	0	0	1	0	0	2	0	0	1	1
▪ Removed?	X	X	X	X		X						

Example

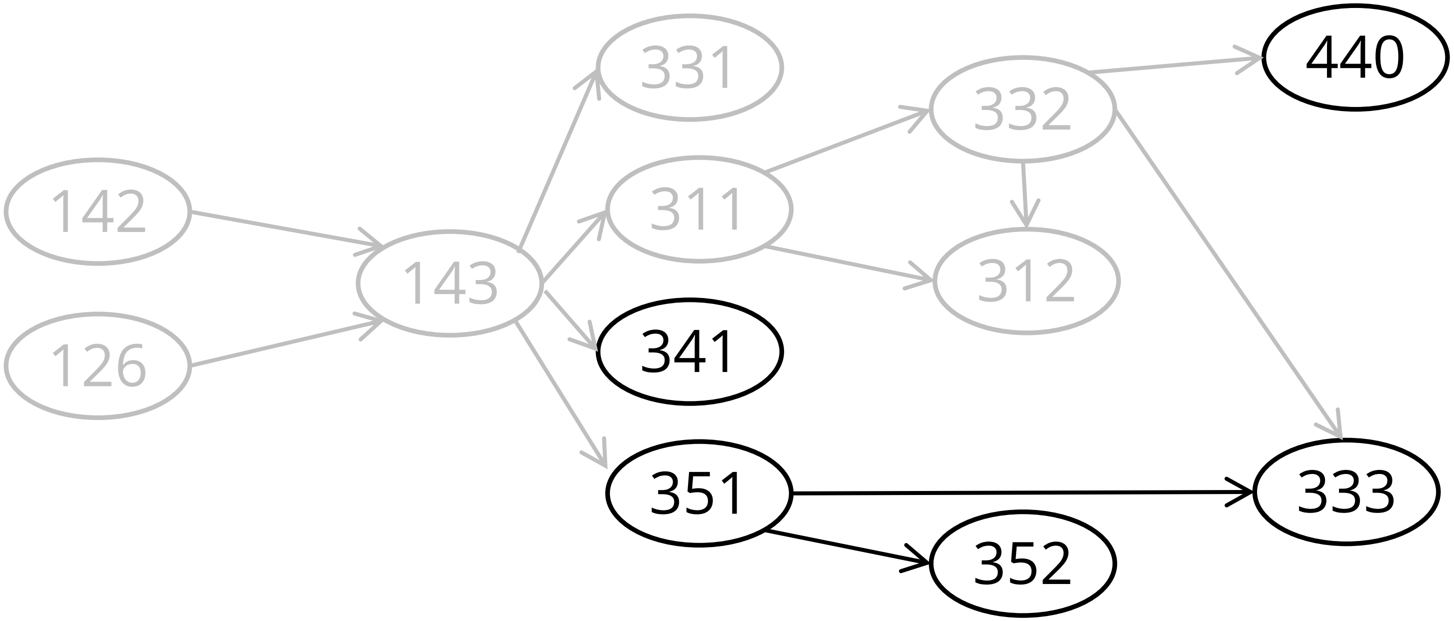


- Output:

126
142
143
311
331
332

Node:	126	142	143	311	312	331	332	333	341	351	352	440
In-degree:	0	0	0	0	0	0	0	1	0	0	1	0
Removed?	X	X	X	X		X	X					

Example

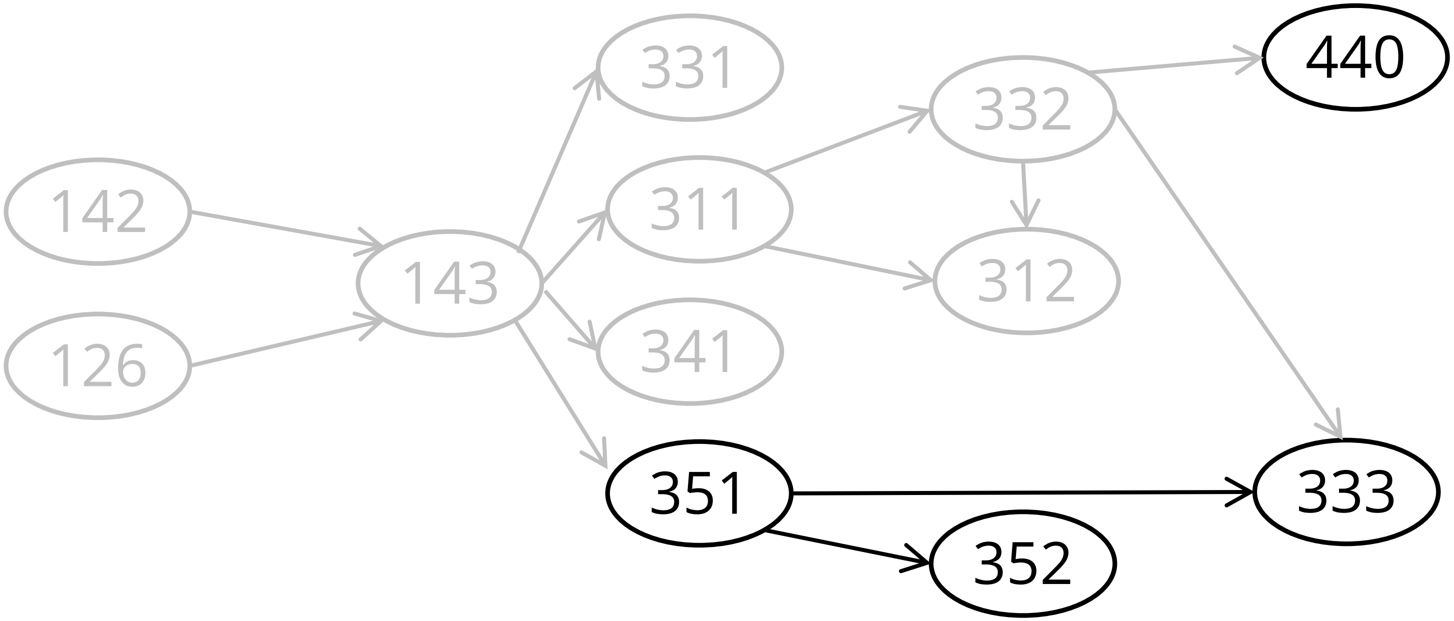


- Output:

126
142
143
311
331
332
312

Node:	126	142	143	311	312	331	332	333	341	351	352	440
In-degree:	0	0	0	0	0	0	0	1	0	0	1	0
Removed?	X	X	X	X	X	X	X					

Example

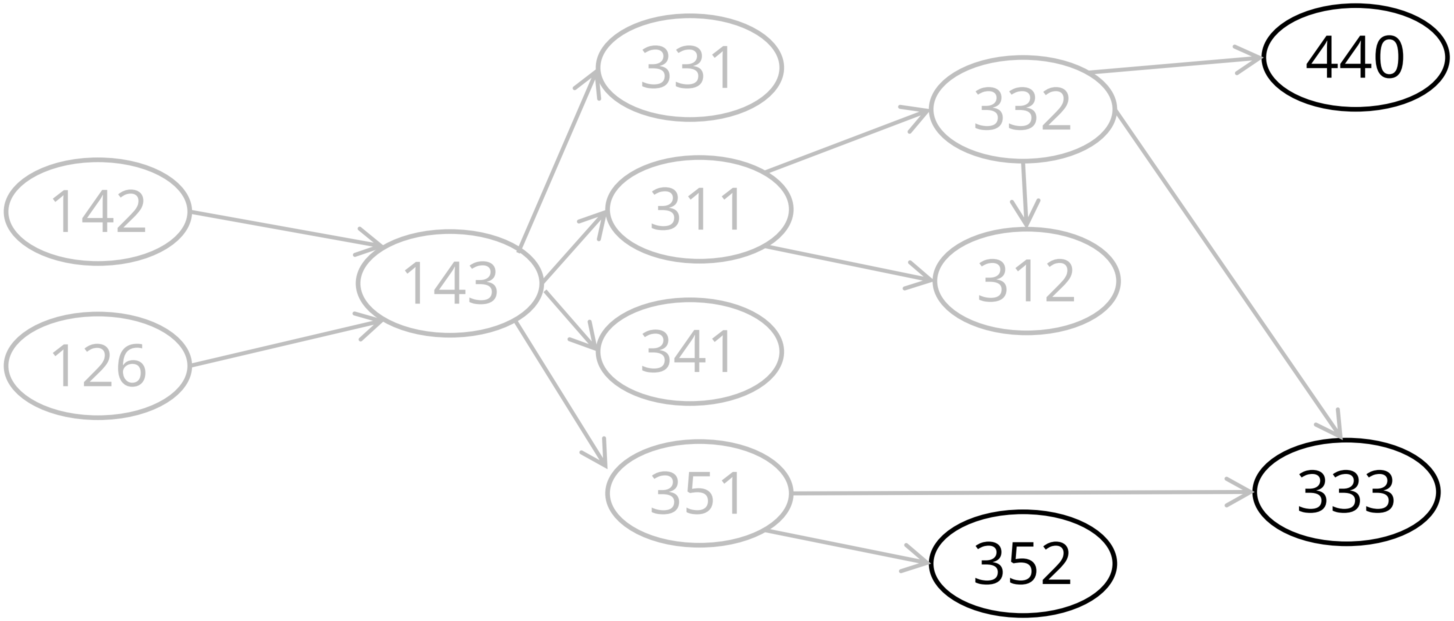


- Output:

126
142
143
311
331
332
312
341

Node:	126	142	143	311	312	331	332	333	341	351	352	440
In-degree:	0	0	0	0	0	0	0	1	0	0	1	0
Removed?	X	X	X	X	X	X	X		X			

Example

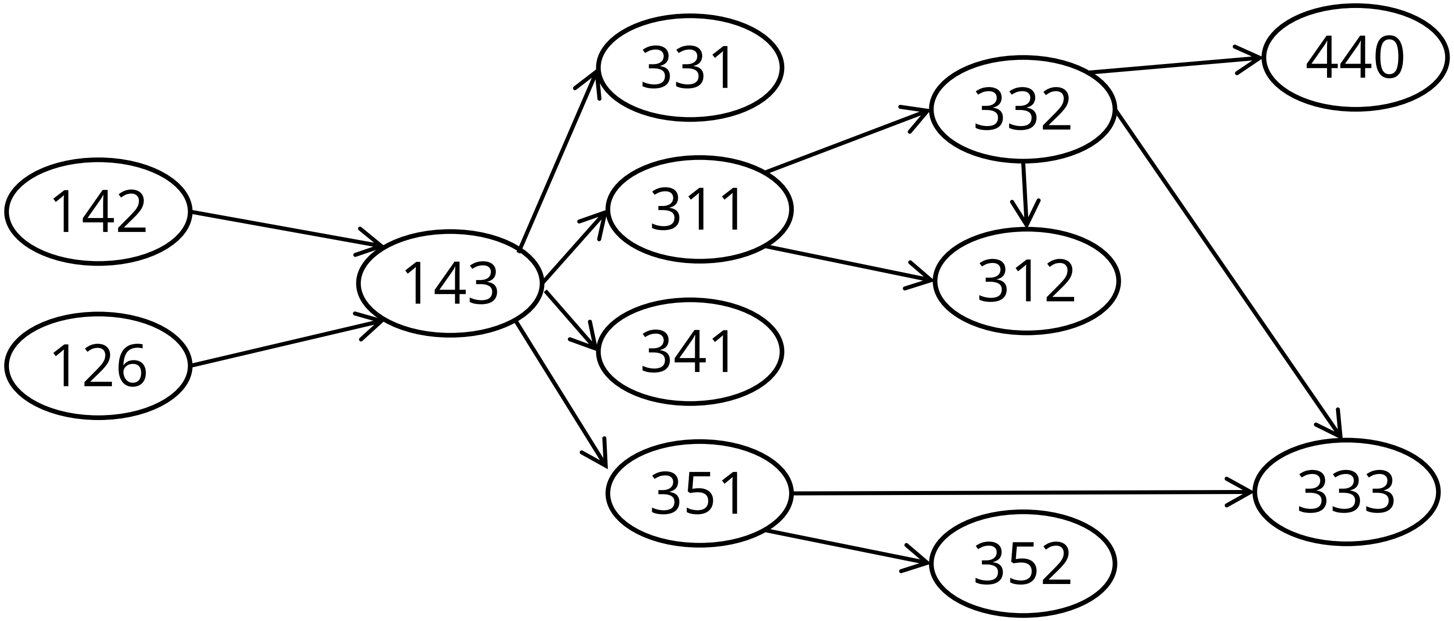


- Output:

126
142
143
311
331
332
312
341
351

▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	0	0	0	0	0	0	0	0	0	0
▪ Removed?	X	X	X	X	X	X	X		X	X		

Example



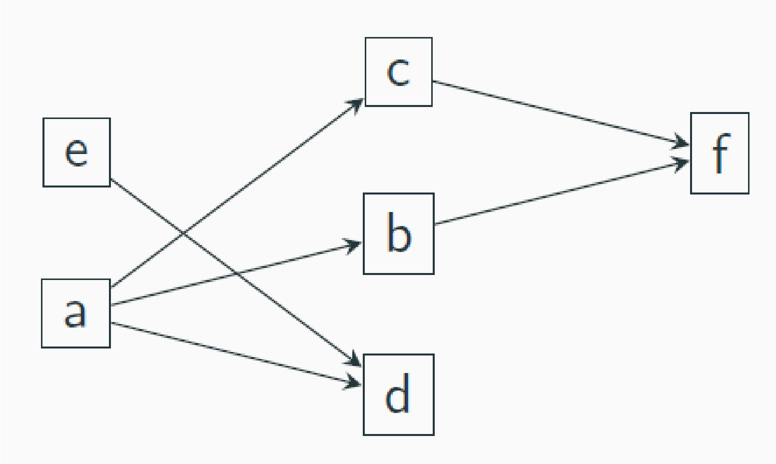
- Output:

126
142
143
311
331
332
312
341
351
333
352
440

▪ Node:	126	142	143	311	312	331	332	333	341	351	352	440
▪ In-degree:	0	0	0	0	0	0	0	0	0	0	0	0
▪ Removed?	X	X	X	X	X	X	X	X	X	X	X	X

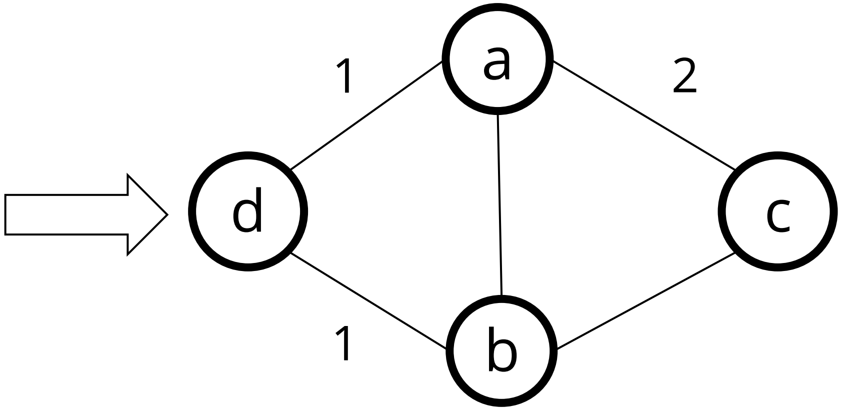
Question

- Find a **correct** topological sort output of the following graph:

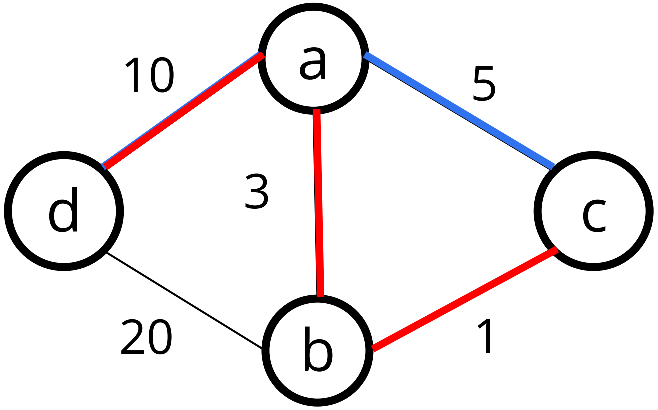


Pathfinding

- We can use BFS to find shortest path between two nodes



- ...but it fails if the graph is weighted!



Shortest Paths

- The shortest path problem is about finding a path between vertices in a graph such that the total sum of the edges weights is minimum.
- Different algorithms are available to find shortest path:
  - Dijkstra's Algorithm
- Dijkstra's algorithm finds the shortest path between two vertices in a graph.

Dijkstra's algorithm

- **Initialization:**

1. Assign each node an initial cost of ∞
2. Set our starting node's cost to 0

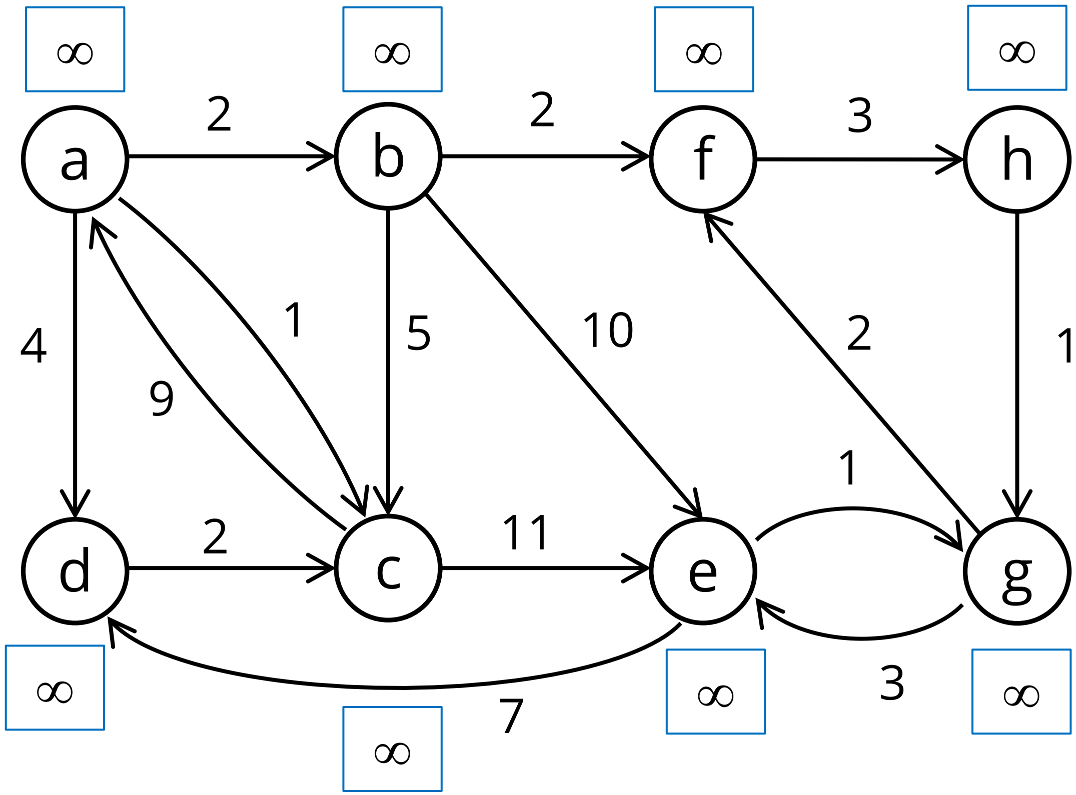
- **Core loop:**

1. Get the next (unvisited) node that has the smallest cost
2. Update all adjacent vertices (if applicable)
3. Mark current node as "visited"

Idea: *Greedily* pick node with smallest cost, then update everything possible. Repeat.

Dijkstra's algorithm

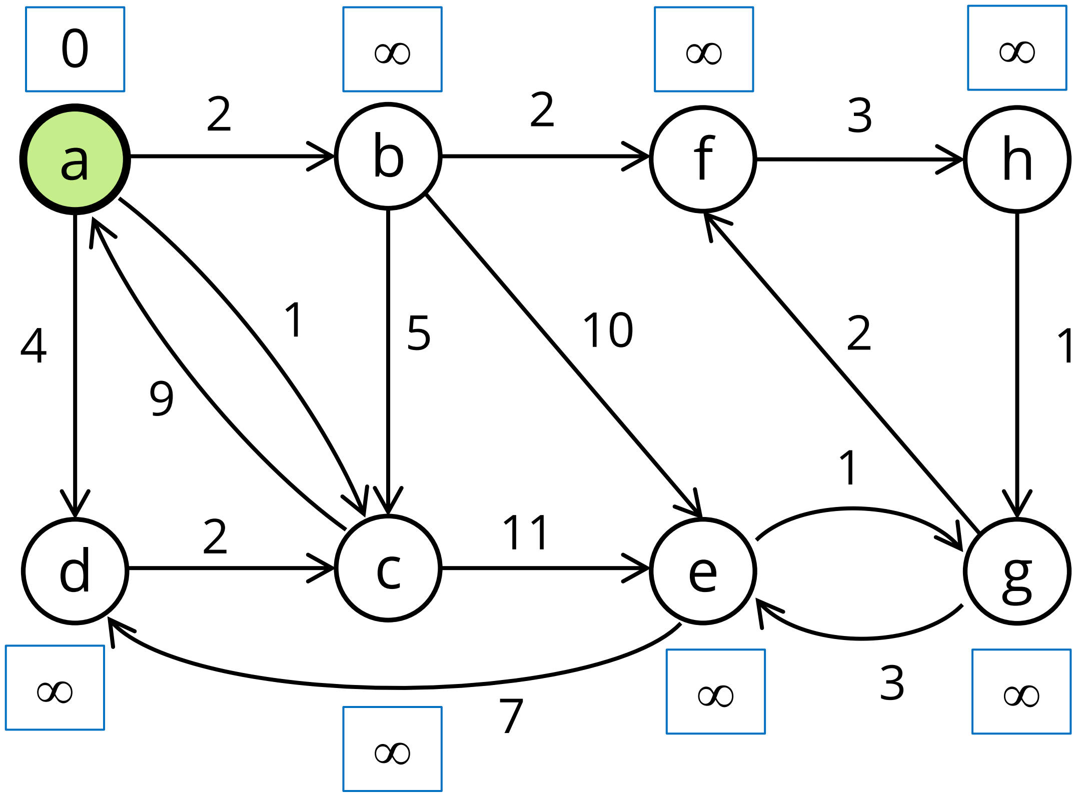
- Suppose we start at vertex "a":



- We initially assign all nodes a cost of infinity. Next, assign the starting node a cost of 0.

Dijkstra’s algorithm

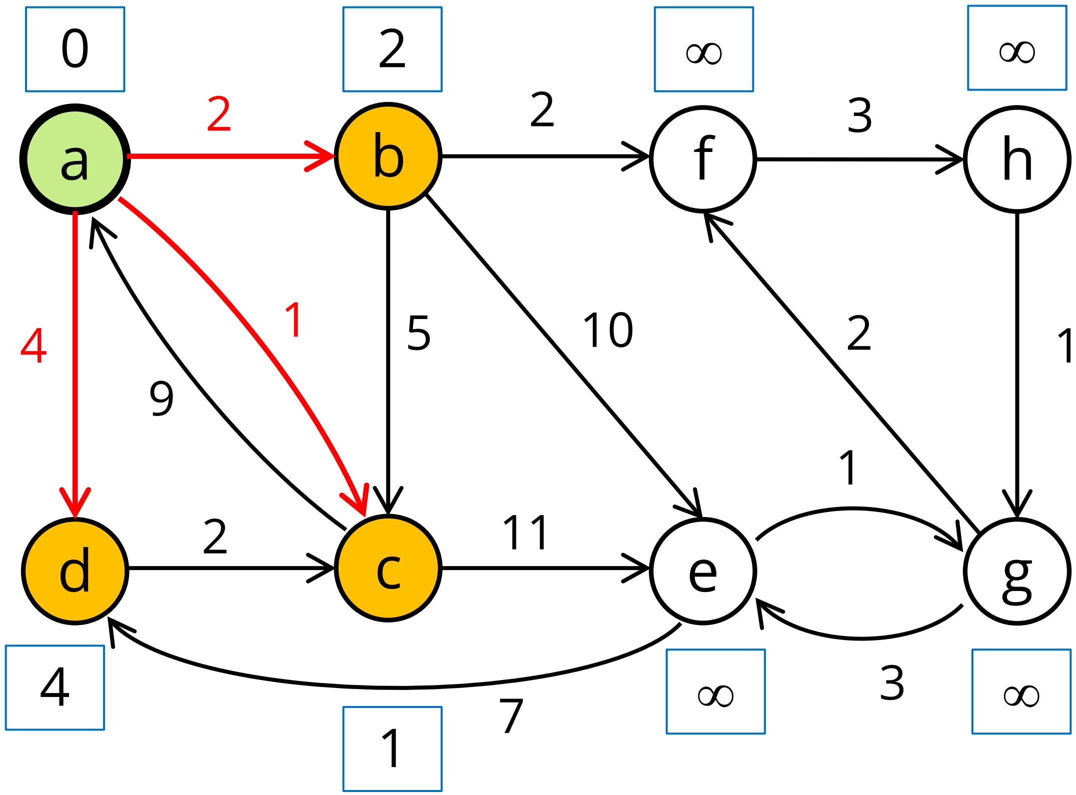
- Suppose we start at vertex “a”:



- Next, assign the starting node a cost of 0.

Dijkstra's algorithm

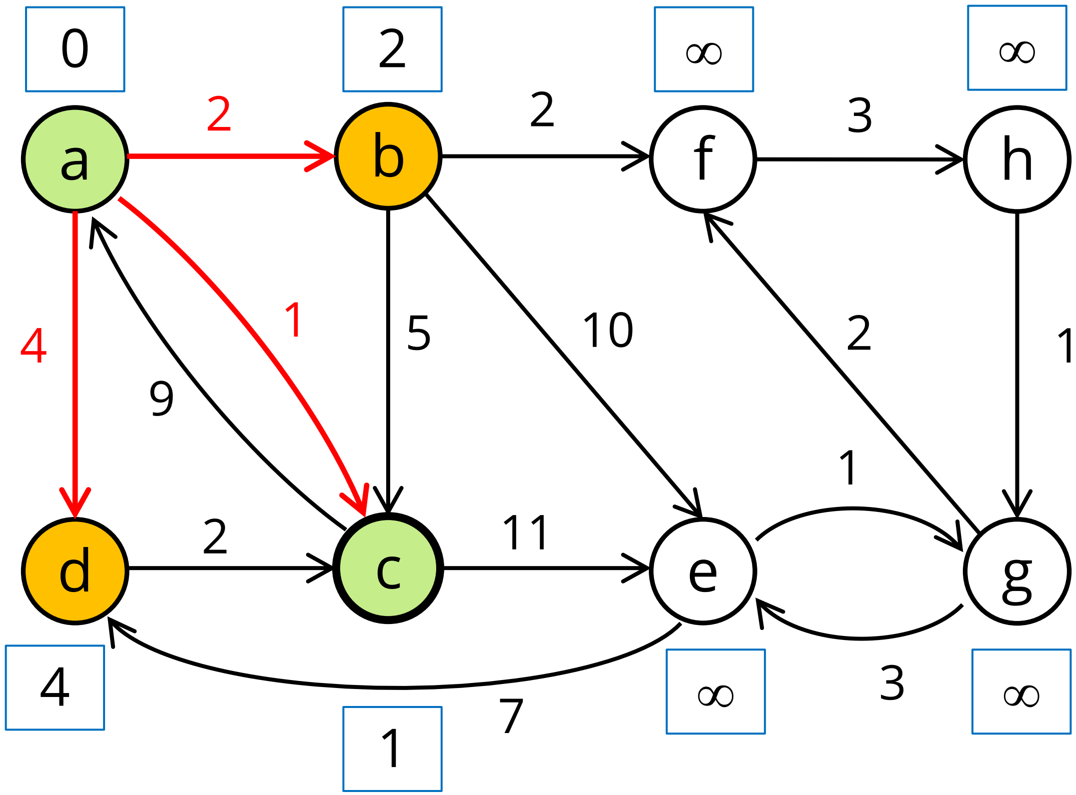
- Suppose we start at vertex "a":



- Next, update all adjacent node costs

Dijkstra's algorithm

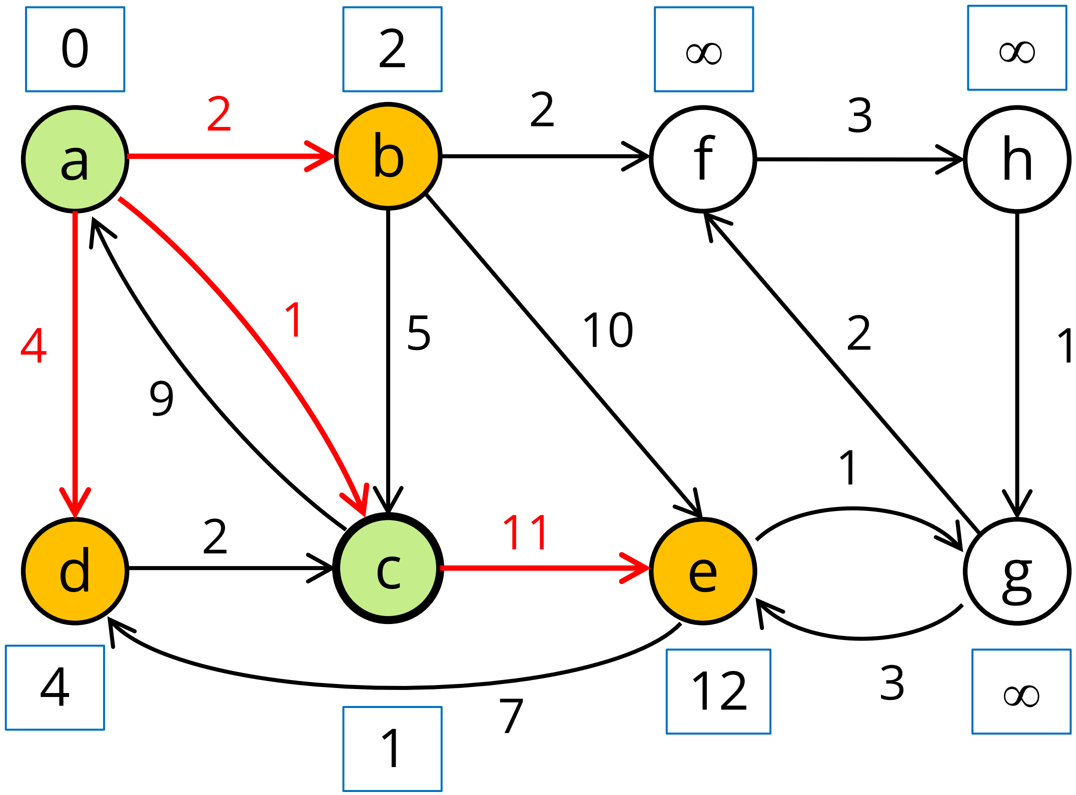
- Suppose we start at vertex "a":



- The node with the smallest cost is *c*, so we visit that next.

Dijkstra's algorithm

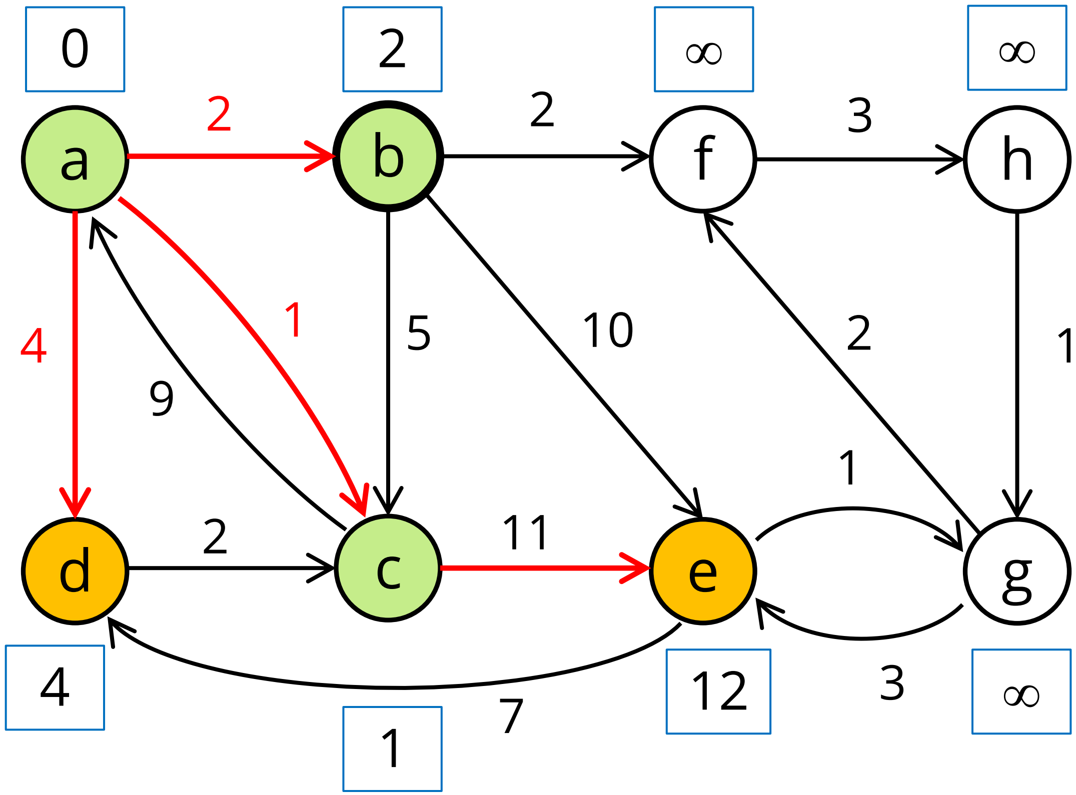
- Suppose we start at vertex "a":



- We consider all adjacent nodes. *a* is fixed, we need to update *e*. The new cost of *e* is the sum of the weights for $a \rightarrow c$ and $c \rightarrow e$.

Dijkstra's algorithm

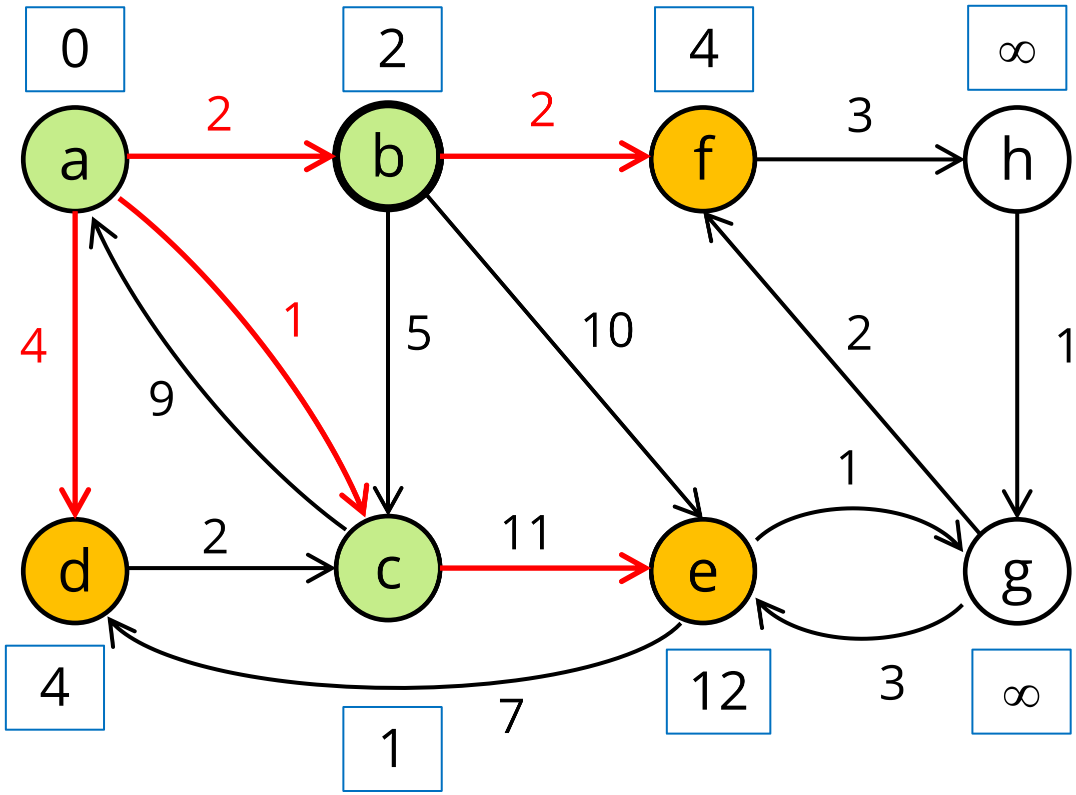
- Suppose we start at vertex "a":



- *b* is the next pending node with smallest cost.

Dijkstra's algorithm

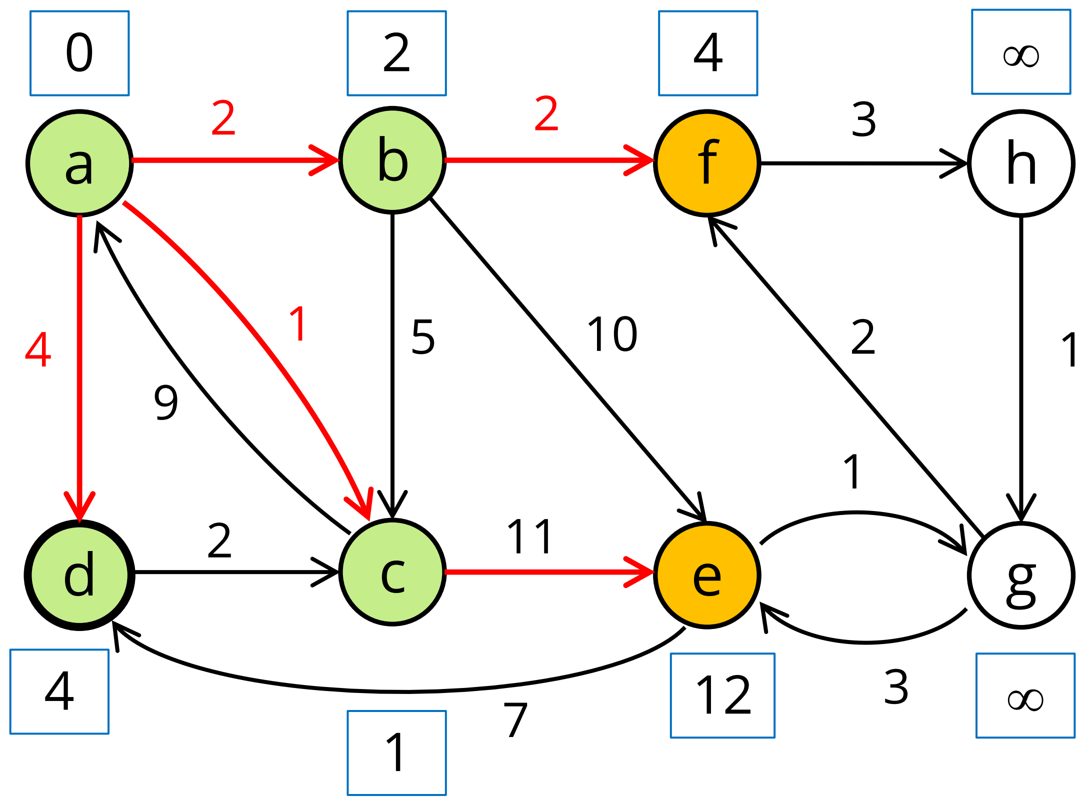
- Suppose we start at vertex "a":



- The adjacent nodes are c, e, and f . The only node where we can update the cost is f .

Dijkstra’s algorithm

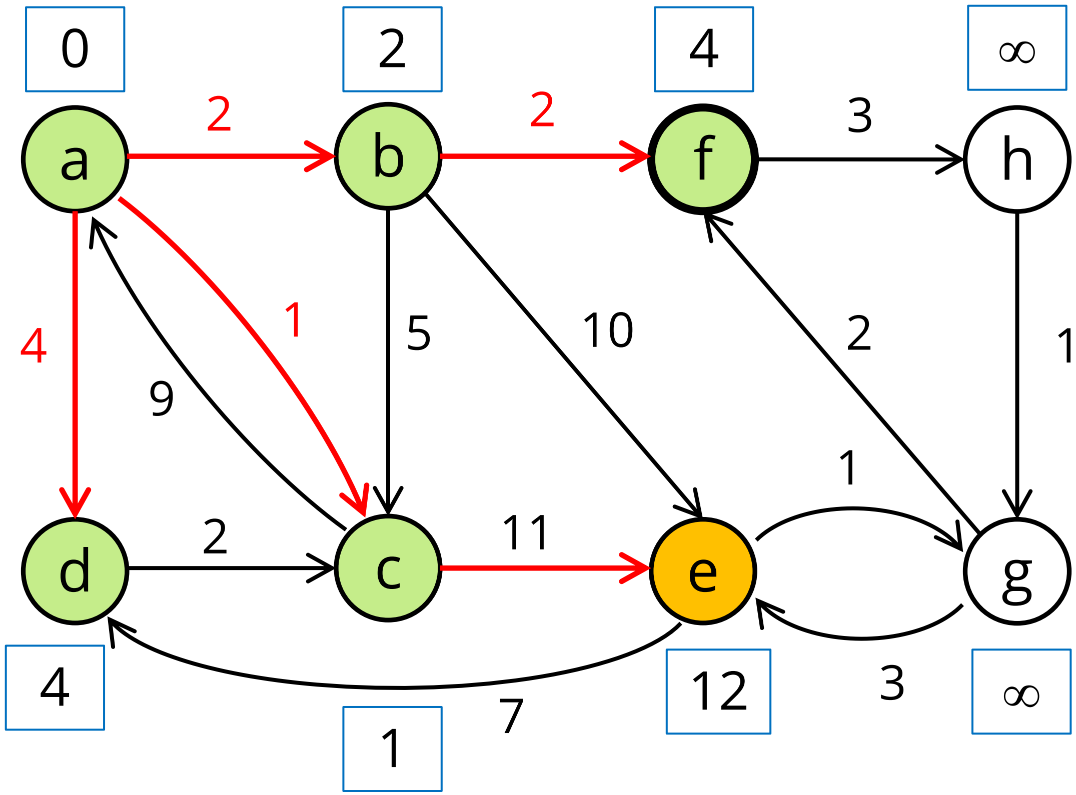
- Suppose we start at vertex “a”:



- Both *d* and *f* have the same cost, so let’s (arbitrarily) pick *d* next. Note that we can’t adjust any of our neighbors.

Dijkstra's algorithm

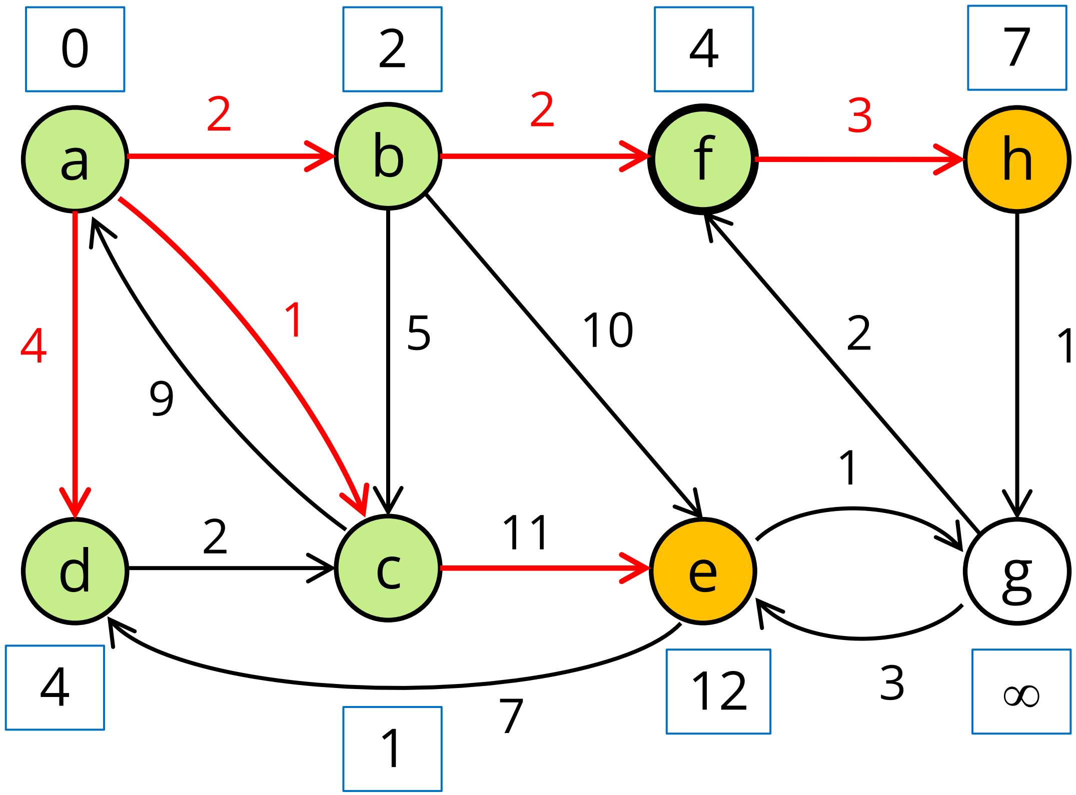
- Suppose we start at vertex "a":



- Next up is f .

Dijkstra's algorithm

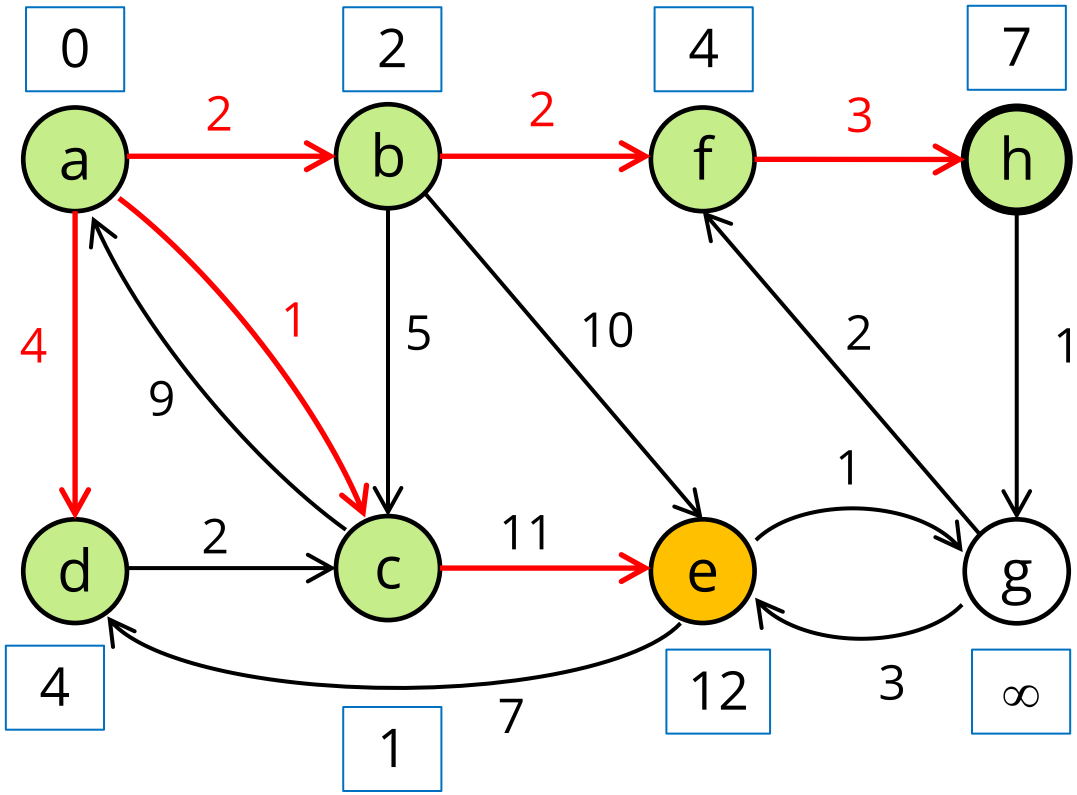
- Suppose we start at vertex "a":



- Next up is f .

Dijkstra's algorithm

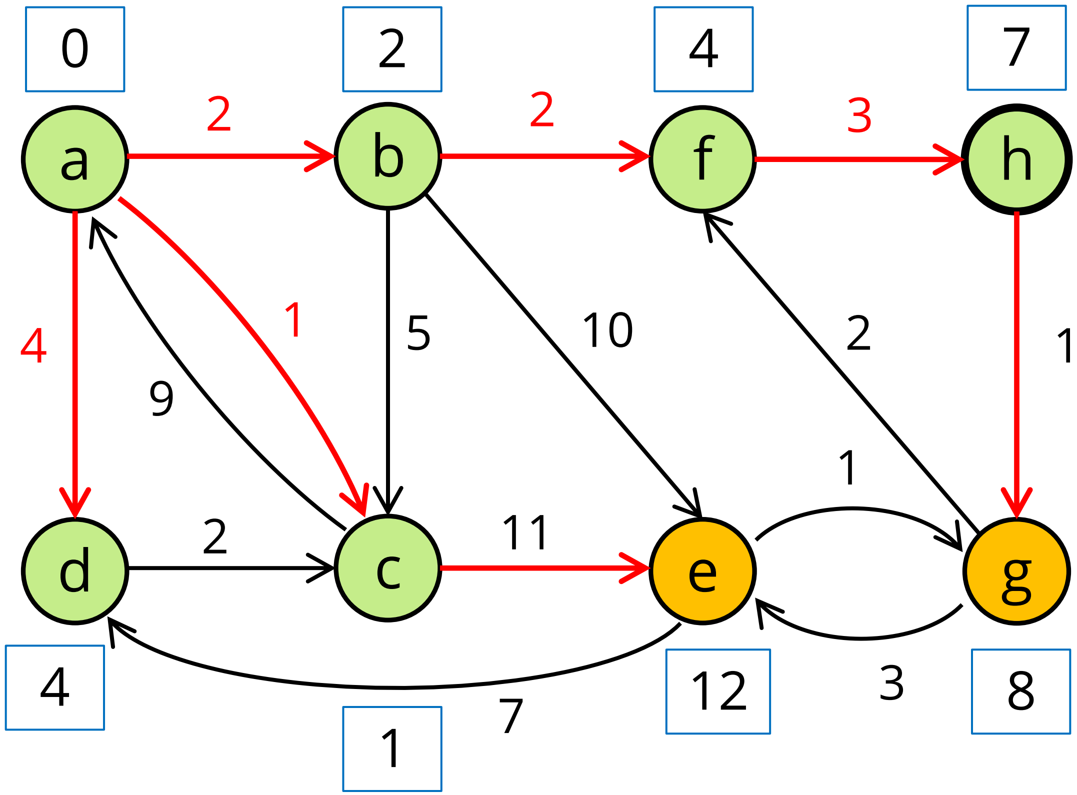
- Suppose we start at vertex "a":



- *h* has the smallest cost now.

Dijkstra's algorithm

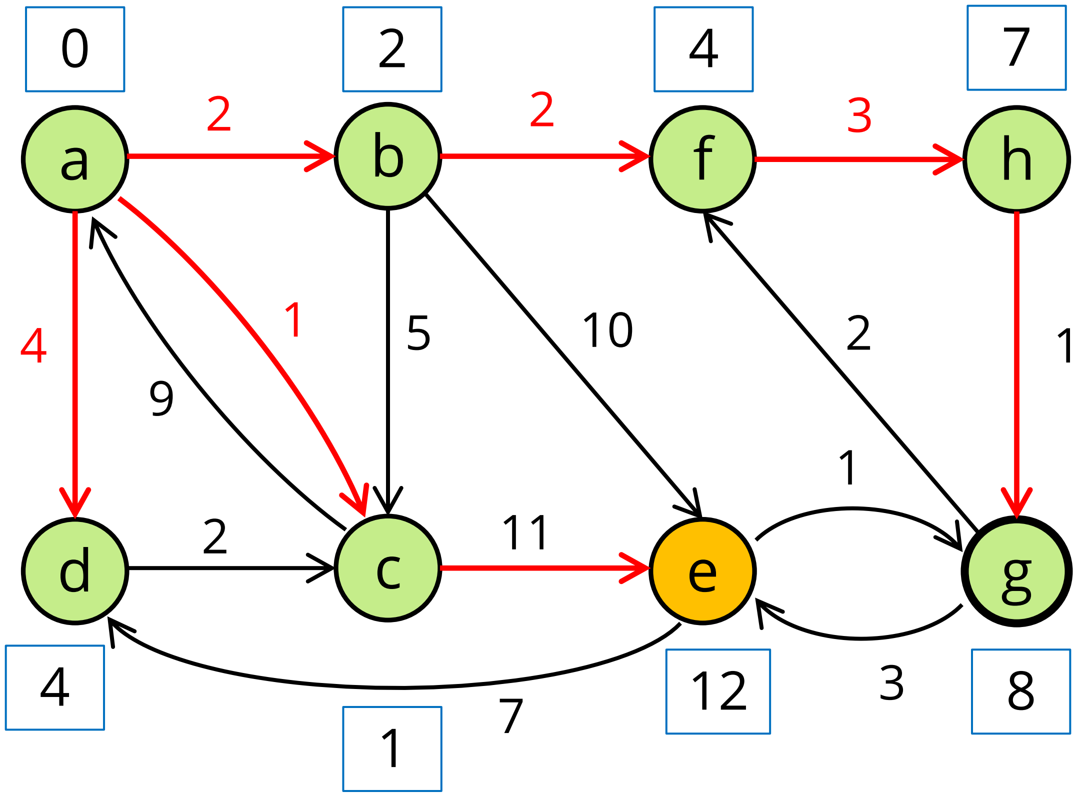
- Suppose we start at vertex "a":



- *h* has the smallest cost now.

Dijkstra's algorithm

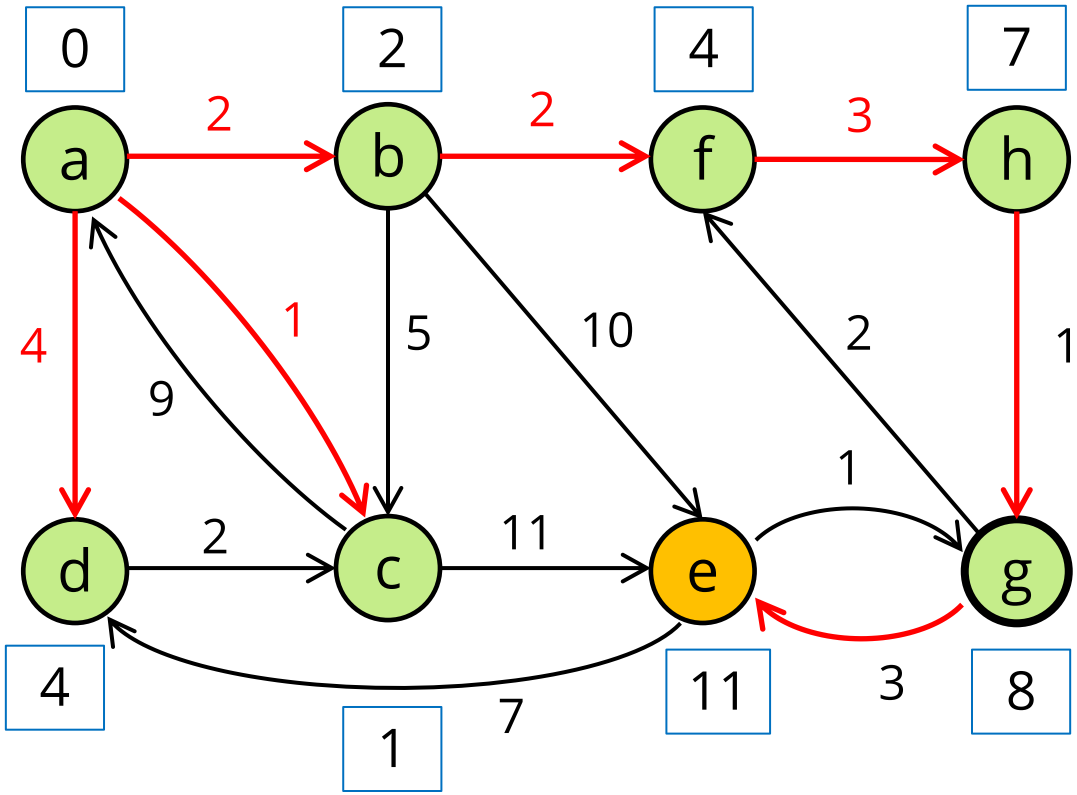
- Suppose we start at vertex "a":



- The two adjacent nodes are *f* and *e*. *f* is fixed, we update *e*: our current route is cheaper then the previous route.

Dijkstra's algorithm

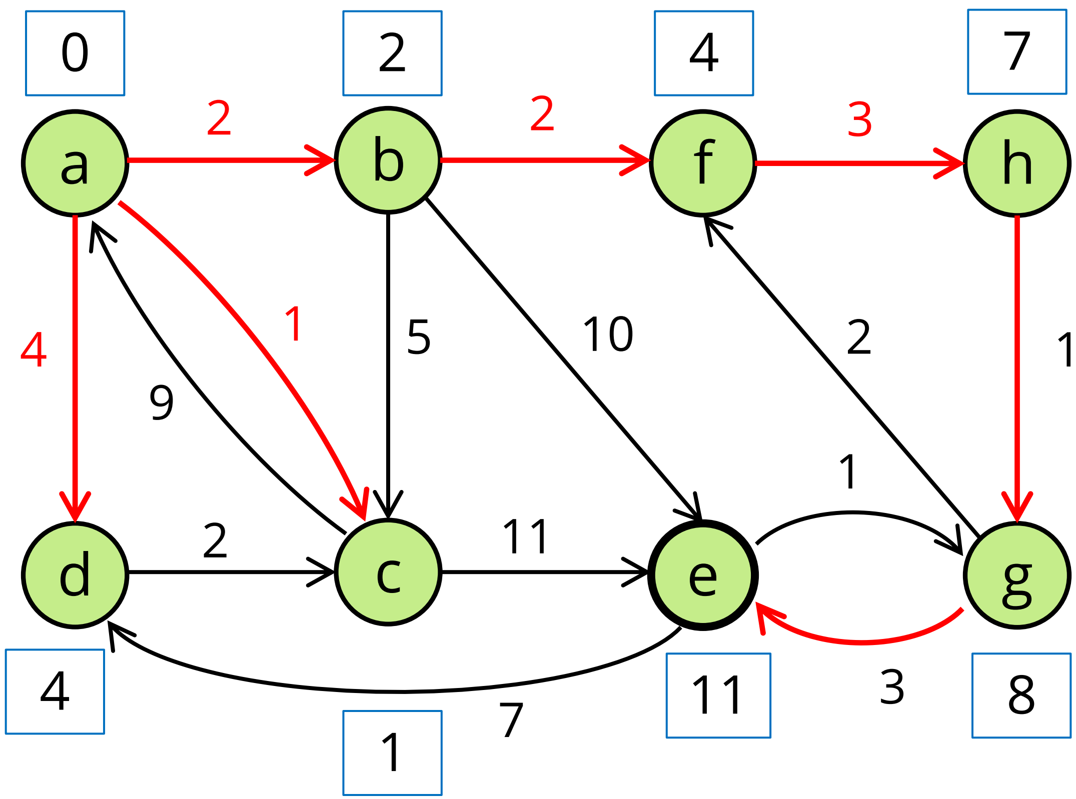
- Suppose we start at vertex "a":



- The two adjacent nodes are *f* and *e*. *f* is fixed, we update *e*: our current route is cheaper then the previous route.

Dijkstra's algorithm

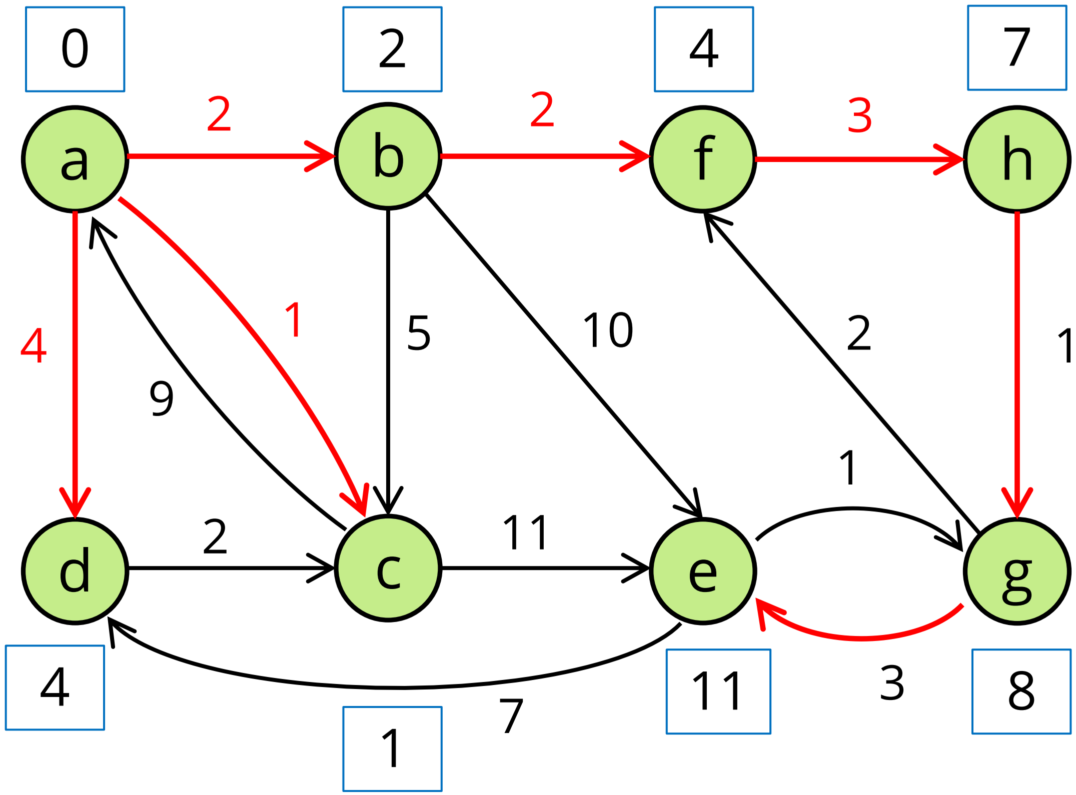
- Suppose we start at vertex "a":



- The last pending node is *e*. We visit it, and check for any unfixed adjacent nodes (there are none).

Dijkstra's algorithm

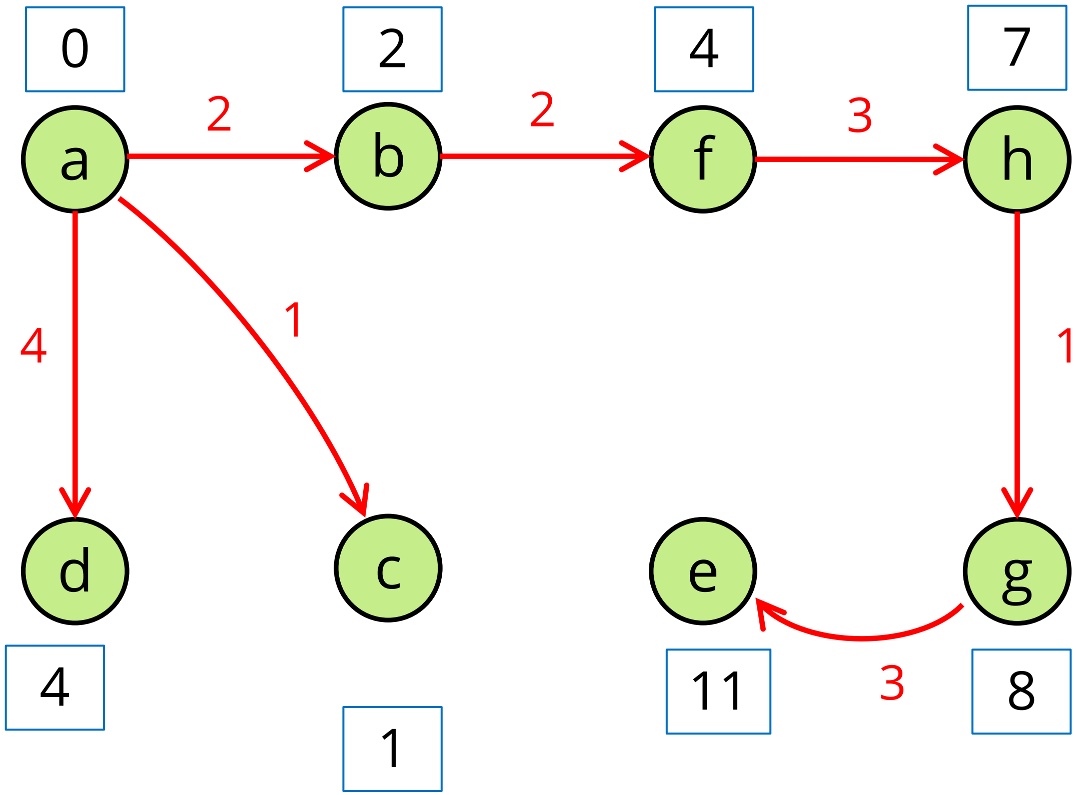
- Suppose we start at vertex "a":



- And we're done!

Dijkstra’s algorithm

- Suppose we start at vertex “a”:



- And we’re done!

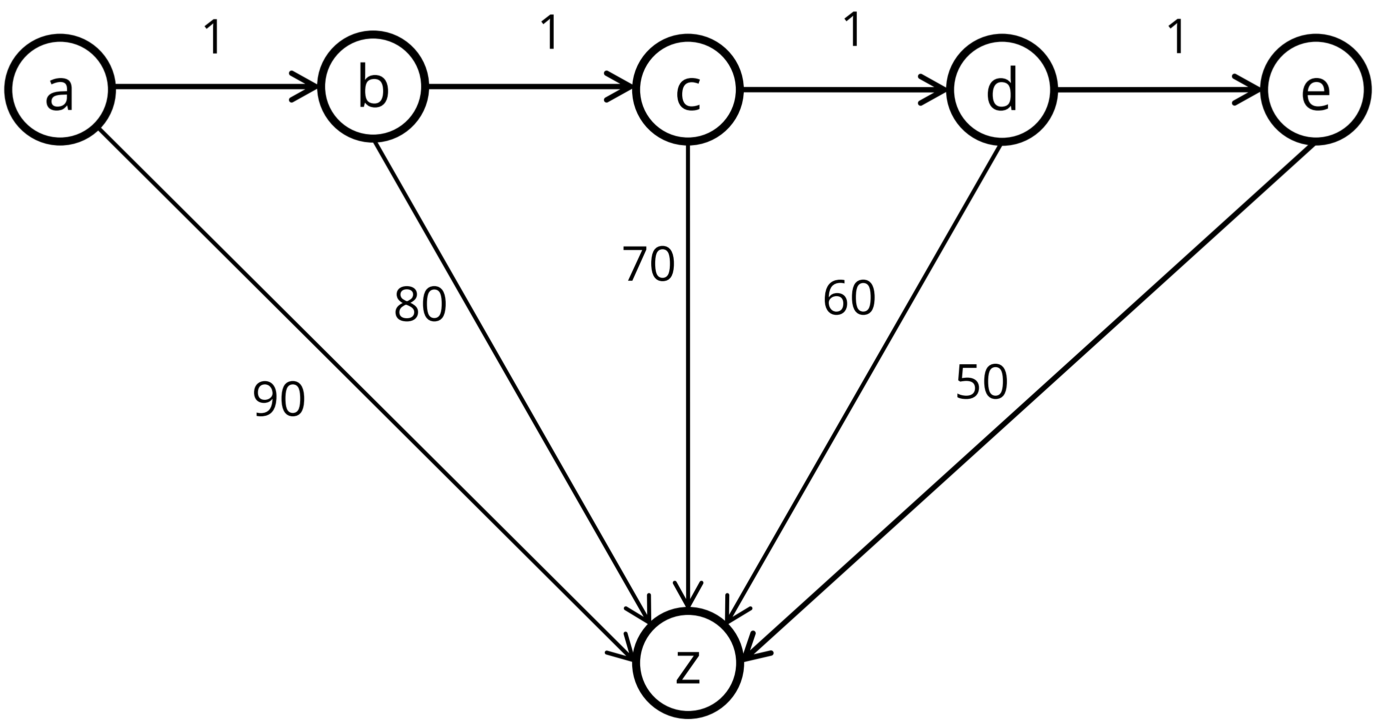
Dijkstra's algorithm

```
def dijkstra(start):
    for (v : vertices):
        set cost(v) to infinity
        set cost(start) to 0

    while (we still have unvisited nodes):
        current = get next smallest node
        for (edge : current.getOutEdges()):
            newCost = min(cost(current) + edge.cost, cost(edge.dest))
            update cost(edge.dest) to newCost
```

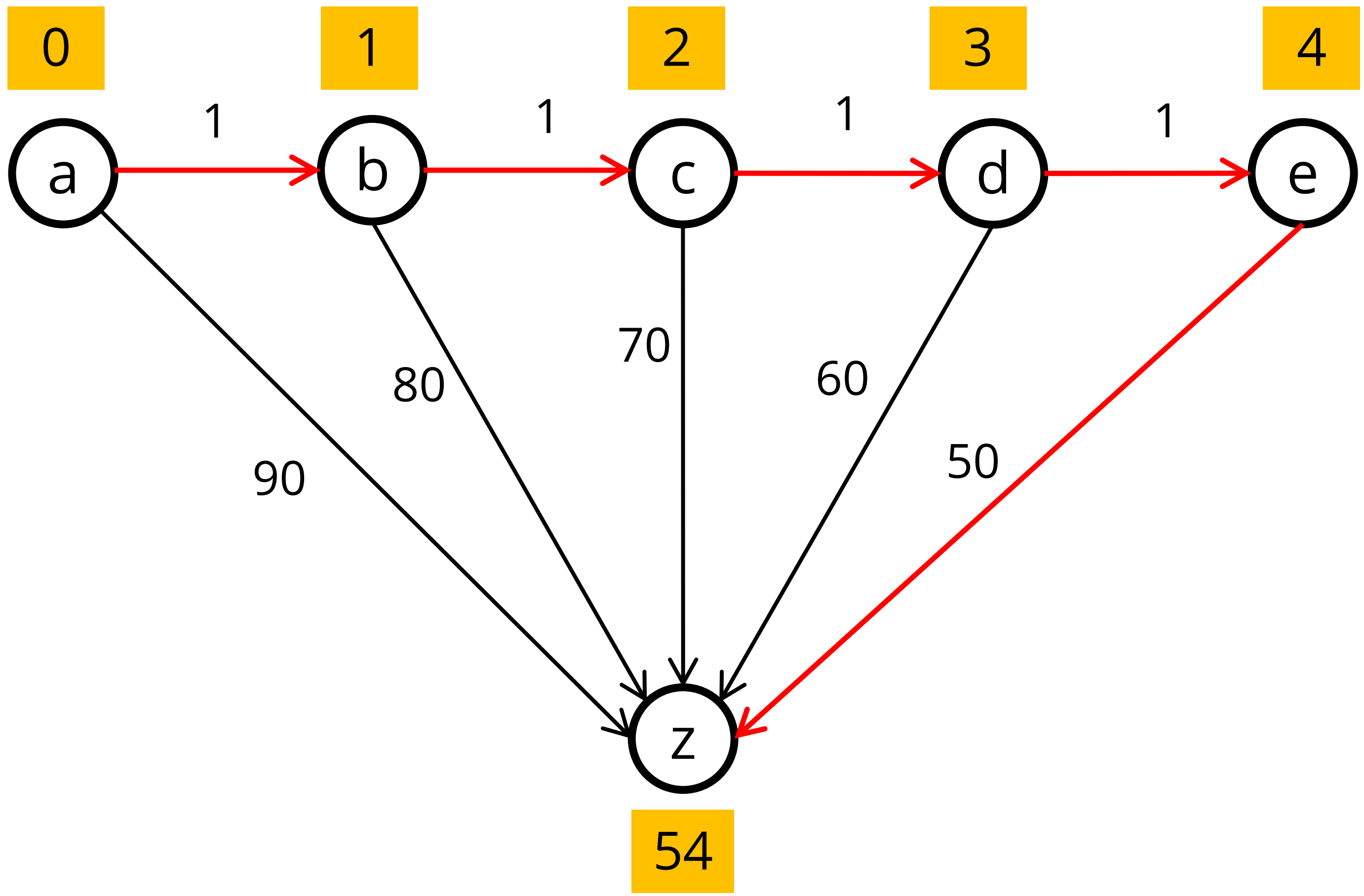
Example

- What does Dijkstra’s algorithm do when run on vertex **a**?



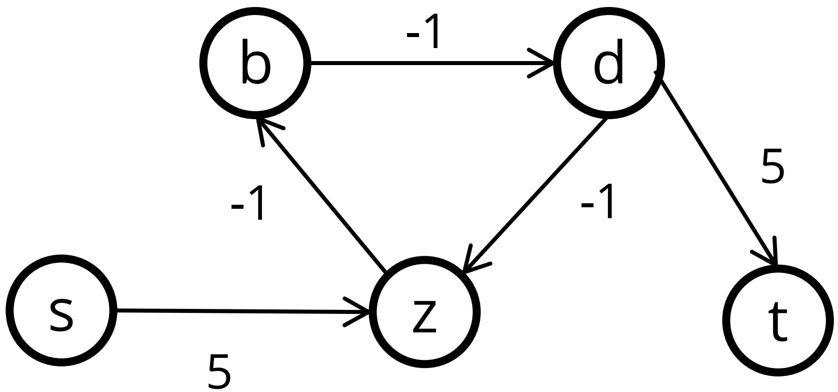
Example

- What does Dijkstra's algorithm do when run on vertex **a**?



Dijkstra's: negative edges

- What's the shortest path now?



- If there are negative edges, Dijkstra's doesn't work
- (There exist other algorithms that can handle negative edges. e.g., see Bellman-Ford.)

Thank you

Review

1. The value of the postfix expression 5 3 3 2 + – * is
   a) 0
   b) -18
   c) -10
   d) none of the above

2. Consider the following operation performed on a stack of size 5. After the completion of all operations, the number of elements present in stack is?
   a) 0
   b) 1
   c) 2
   d) 3
   e) none of the above

```
push(1);
pop();
push(2);
peek();
push(3);
pop();
push(4);
peek()
pop();
pop();
push(5);
```

Review

3. In a circular queue, how do we increment the rear of the queue?
   a) rear++
   b) rear = (rear + 1) % queue.length
   c) rear = (rear % queue.length) + 1
   d) none of the above
4. What is the worst-case time complexity for search operations (i.e., search time) in a Binary Search Tree?
   a) O(n)
   b) O(logn)
   c) O(nlogn)
   d) none of the above
5. What is the worst-case time complexity for delete operations in a Binary Search Tree?
   a) O(n)
   b) O(logn)
   c) O(nlogn)
   d) none of the above

Review

6. The worst-case time complexity to search for an element in a balanced tree with n elements is
   a) O(1)
   b) O(n)
   c) O(logn)
   d) O(nlogn)
   e) none of the above

7. A binary tree is an AVL tree if it maintains a balance factor in each node of
   a) 0
   b) 1
   c) -1
   d) all of the above

8. What happens when no base condition is defined in a case of recursion?
   a) Stack overflow
   b) Stack underflow
   c) Both overflow and underflow
   d) none of the above

Review

9. What's the output of the following code for **`func(2,3)`**? Assume that we have other necessary code written to run the program properly.

```
int func(int a, int b)
 {
 if (b==1)
    return a;
 else
    return a + func(a+1,b-1);
 }
```

10. Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order.

```
70
11
47
81
20
61
10
12
13
62
```