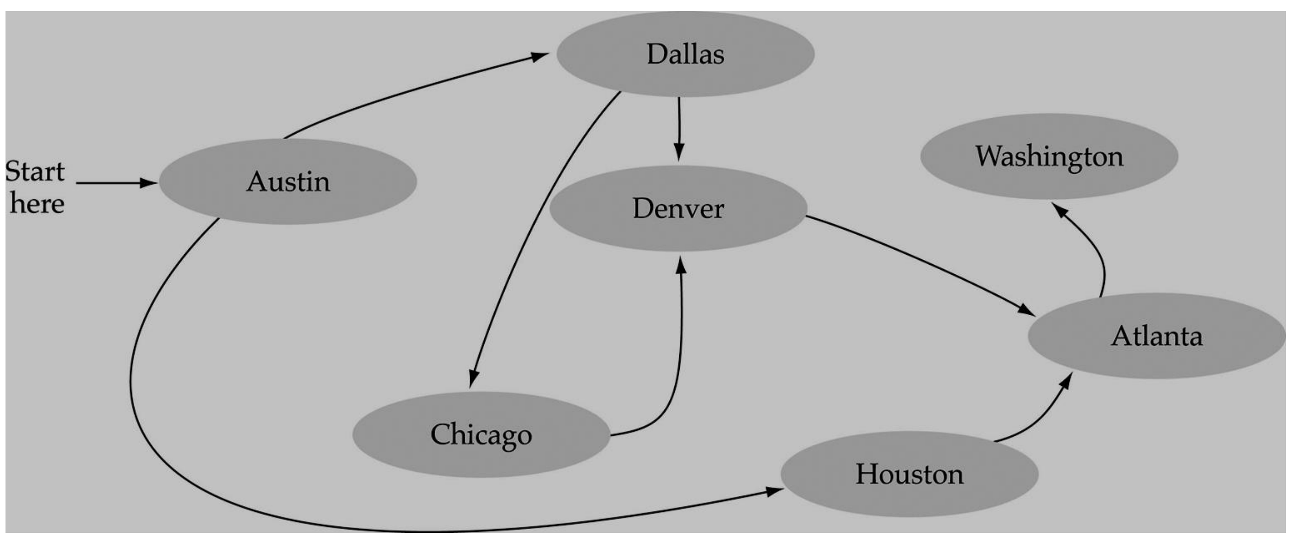
Week 17
Graph Data Structure

Data Structures CS-218

Instructor: Anam Qureshi

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph G is defined as follows:

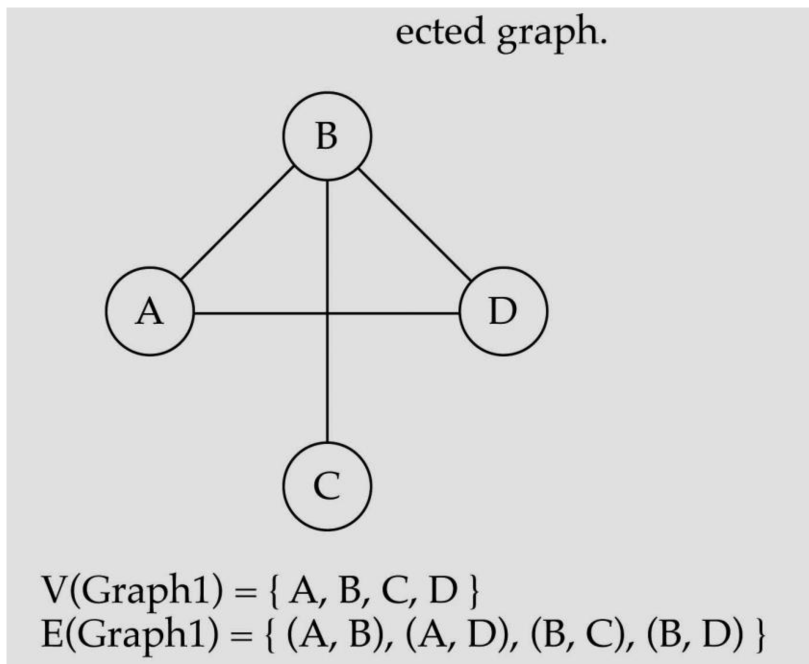
$$G=(V,E)$$

$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)

Directed vs. undirected graphs

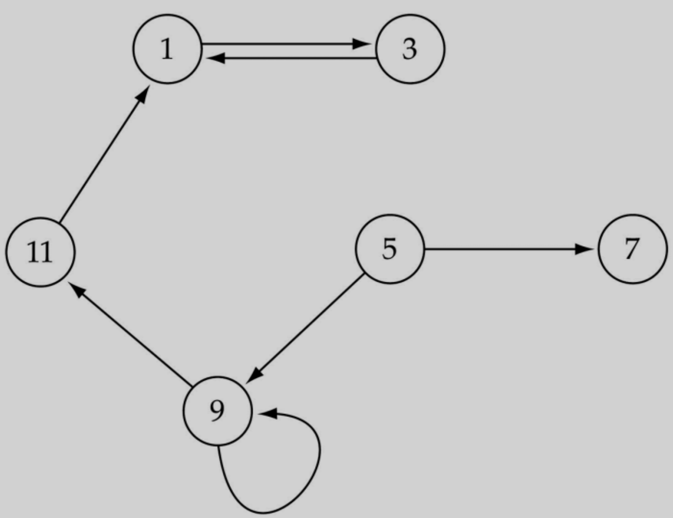
- When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.

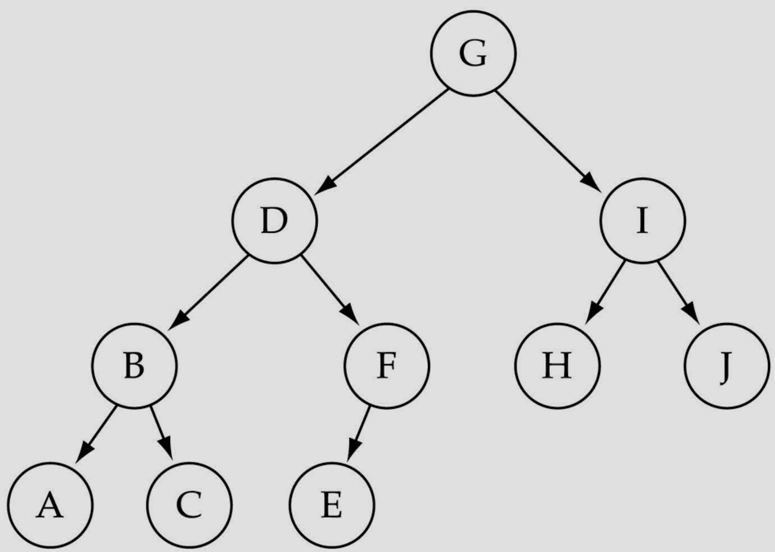


V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7), (9, 9), (11, 1) }

Trees vs graphs

- Trees are special cases of graphs!!

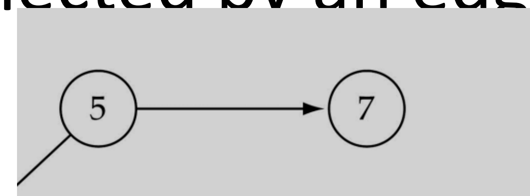
(c) Graph3 is a directed graph.



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology

- <u>Adjacent nodes</u>: two nodes are adjacent if they are connected by an edge



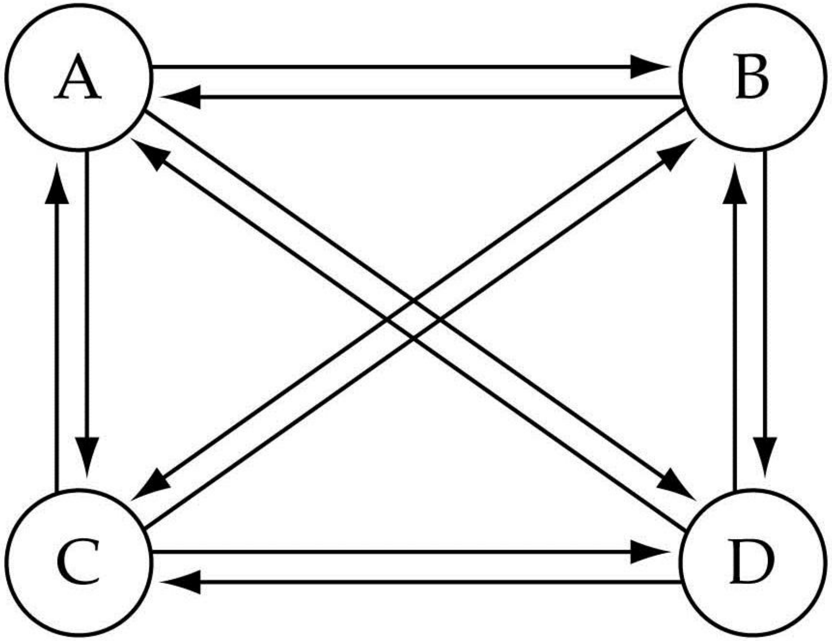
- <u>Path</u>: a sequence of vertices that connect two nodes in a graph
- <u>Complete graph</u>: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$O(N^2)$



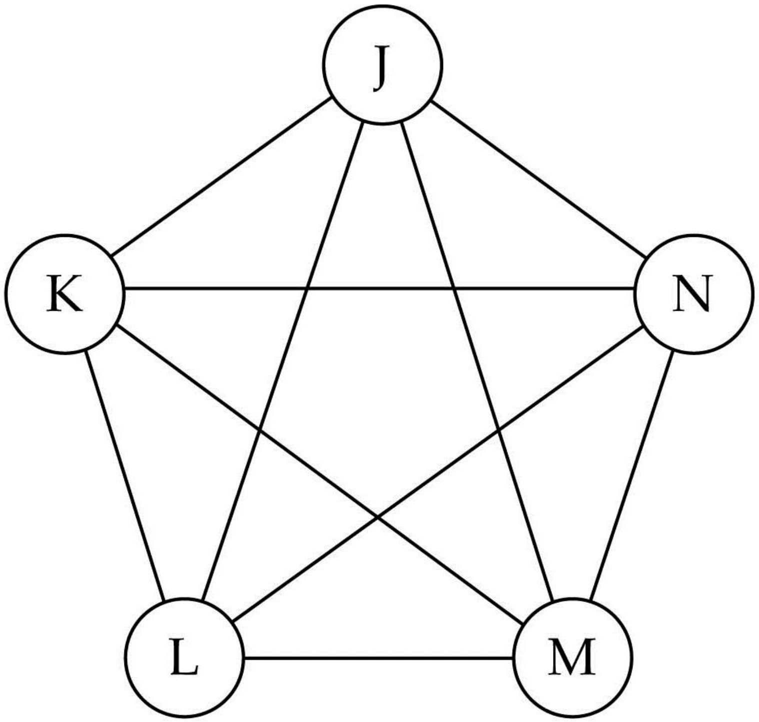
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

  *N * (N-1) / 2*

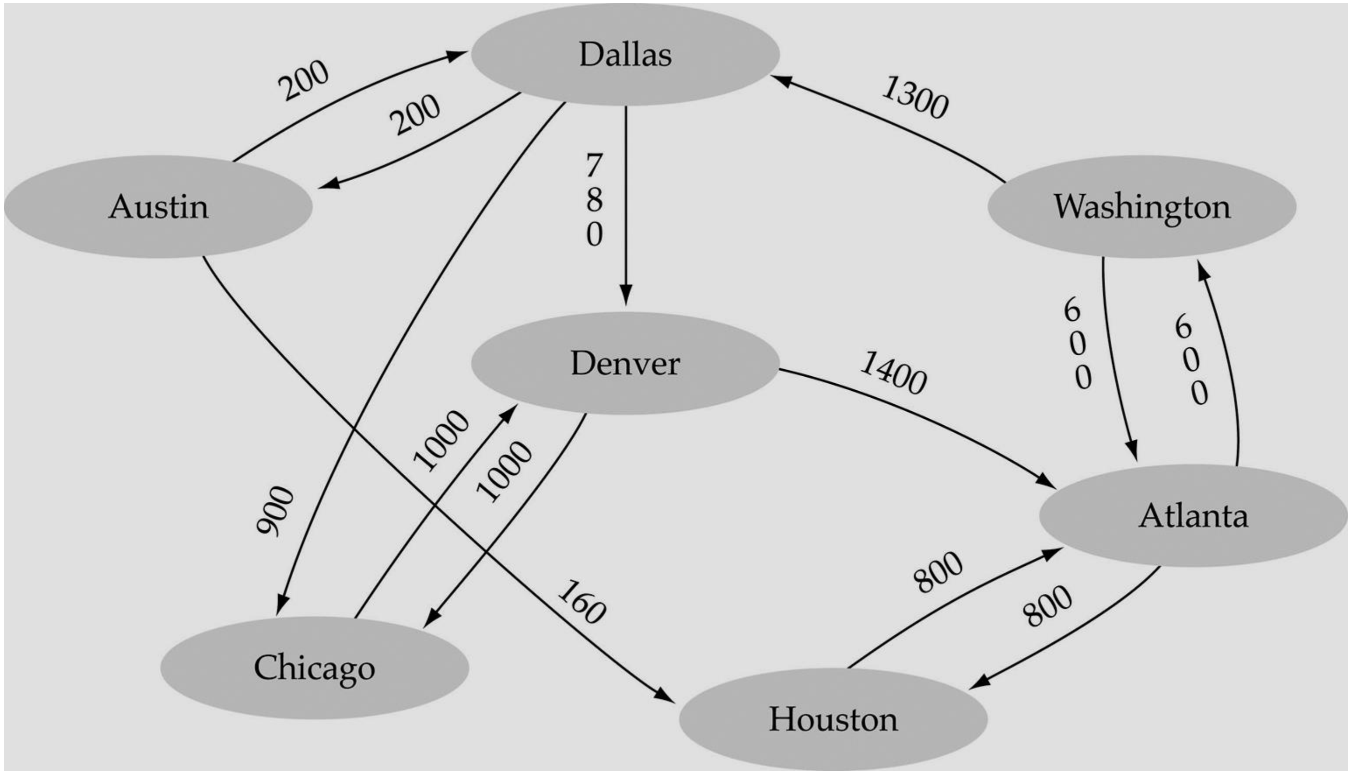
  $O(N^2)$



(b) Complete undirected graph.

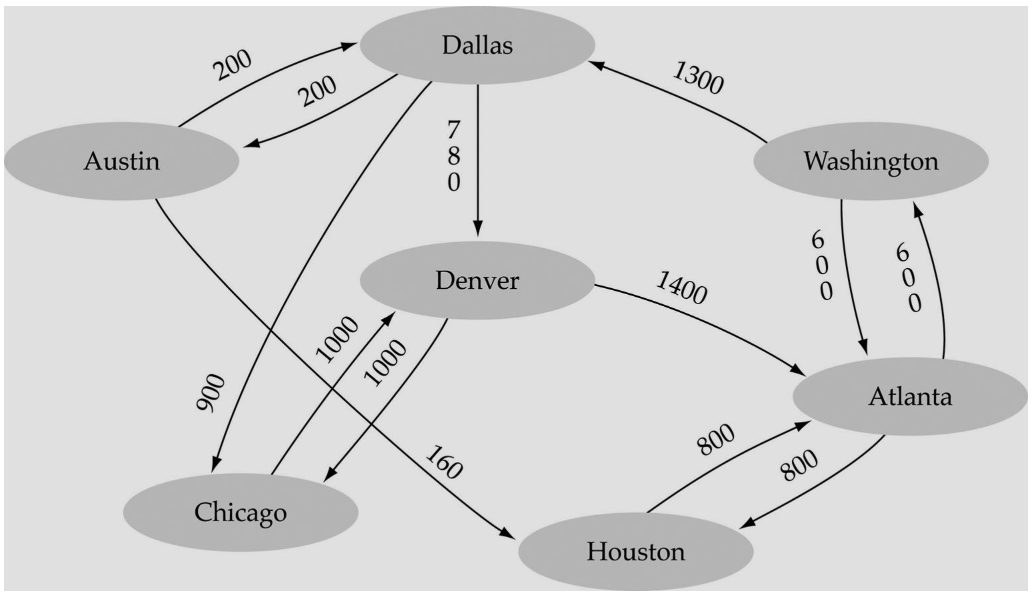
Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



Graph Representation

- <u>Array-based implementation</u>
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges



graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

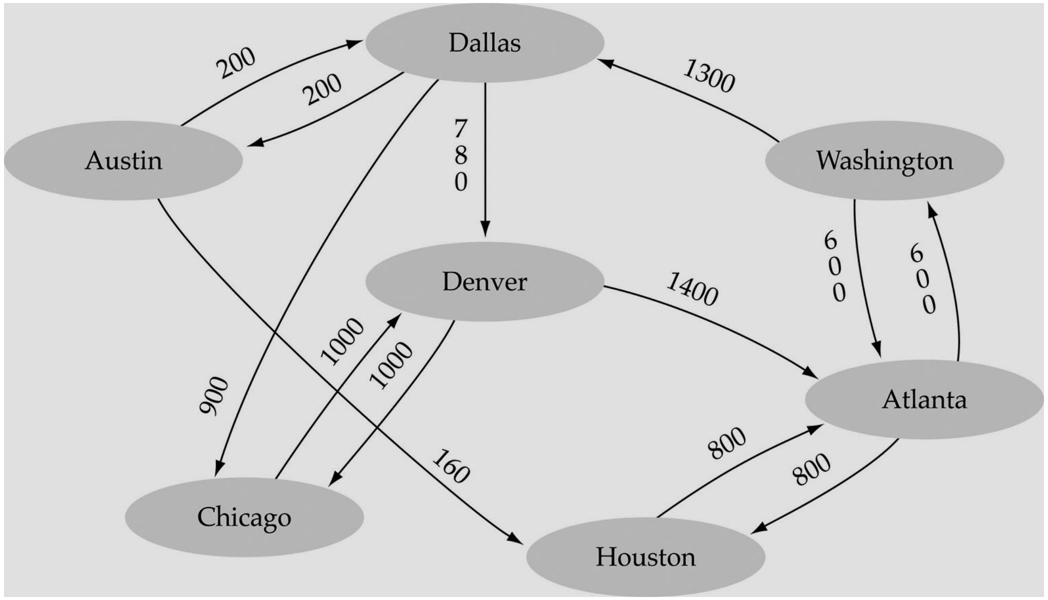
.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

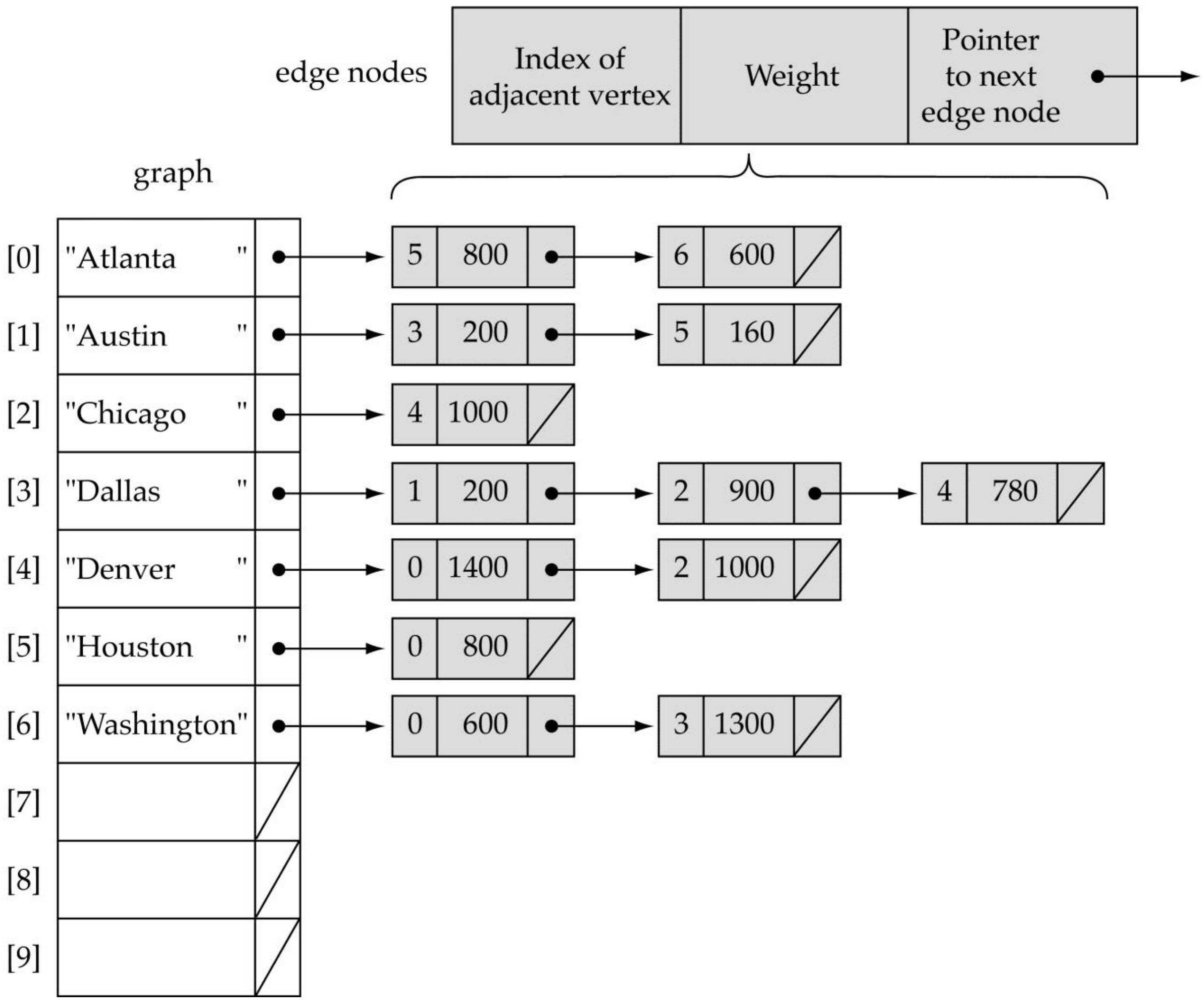
(Array positions marked '•' are undefined)

Graph Representation (cont.)

- <u>Linked-list implementation</u>
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)



(a)


edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node
graph
[0]
"Atlanta "
5
800
6
600
[1]
"Austin "
3
200
5
160
[2]
"Chicago "
4
1000
[3]
"Dallas "
1
200
2
900
4
780
[4]
"Denver "
0
1400
2
1000
[5]
"Houston "
0
800
[6]
"Washington"
0
600
3
1300
[7]
[8]
[9]

Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
  - Good for dense graphs
  - Connectivity between two vertices can be tested quickly
- **Adjacency list**
  - Good for sparse graphs
  - Vertices adjacent to another vertex can be found quickly

Graph searching

- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)
- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)

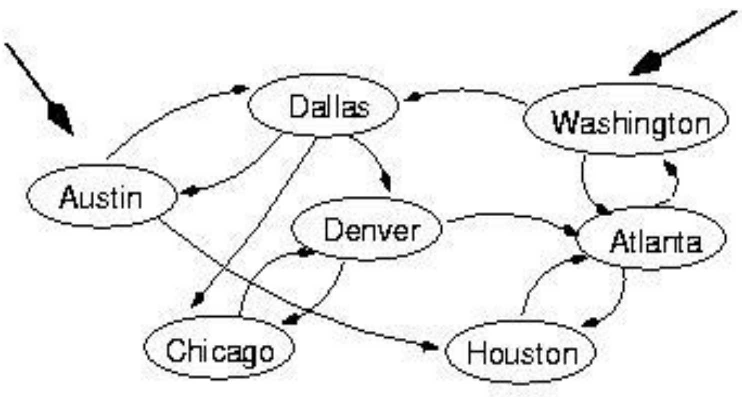
Depth-First-Search (DFS)

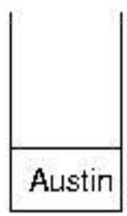
- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a *stack*

Depth-First-Search (DFS) *(cont.)*

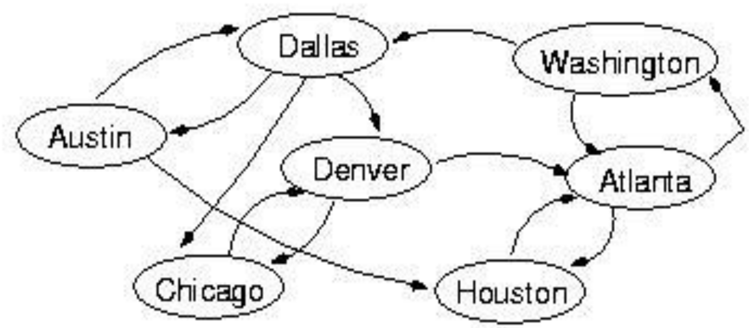
```
Set found to false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Push all adjacent vertices
   onto stack
WHILE !stack.IsEmpty() AND
   !found

IF(!found)
  Write "Path does not exist"
```

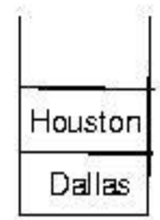


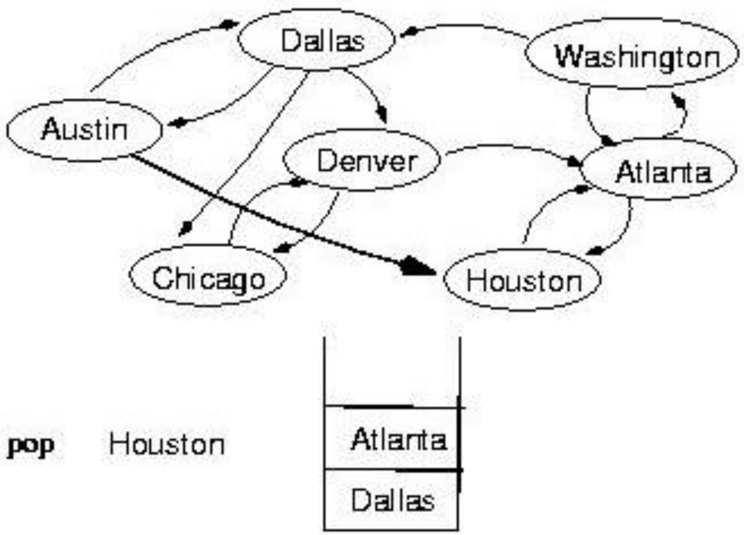


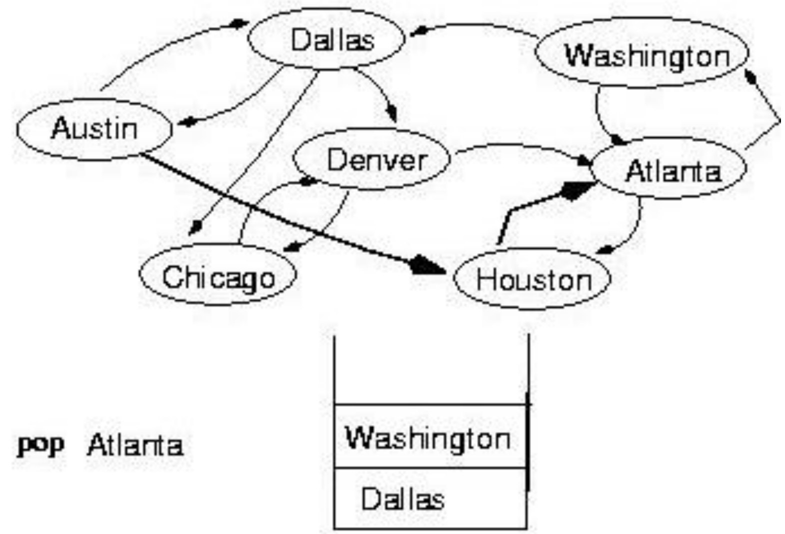
(initialization)

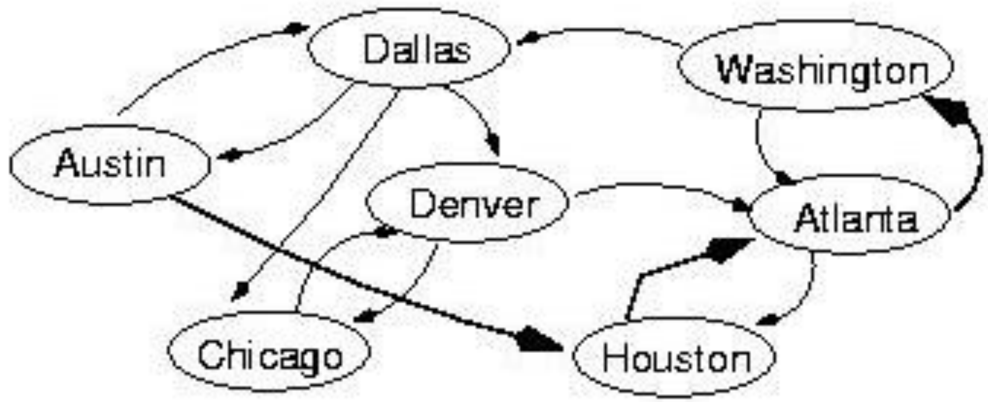


pop Austin

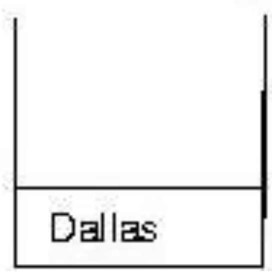



Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Houston
Atlanta
Dallas


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Atlanta
Washington
Dallas



pop Washington



Breadth-First-Searching (BFS)

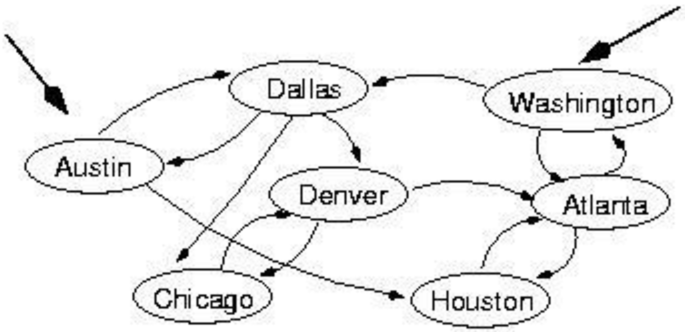
- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

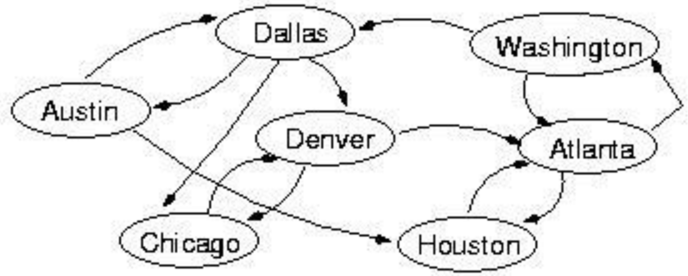
Breadth-First-Searching (BFS) (cont.)

- BFS can be implemented efficiently using a *queue*

```
Set found to false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
```

- Should we mark a vertex when it is enqueued or when it is dequeued ?



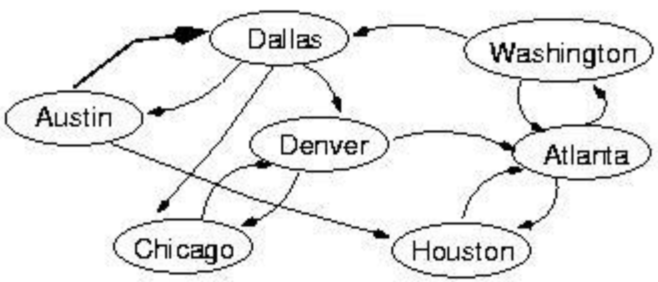


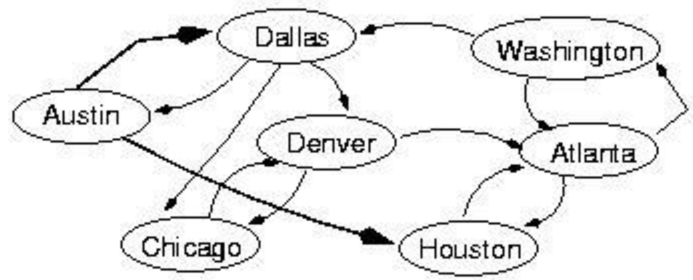
(initialization)

			Austin

dequeue Austin

		Dallas	Houston



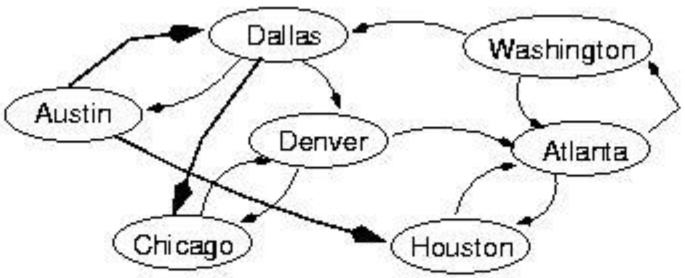


dequeue Dallas

	Houston	Chicago	Denver

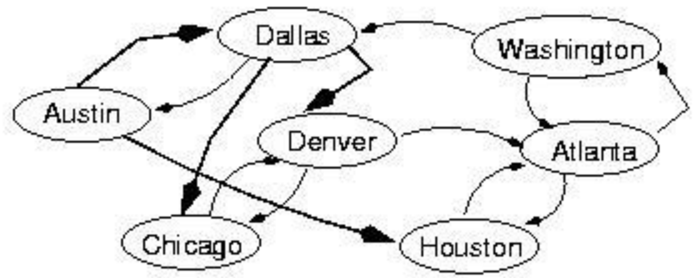
dequeue Houston

	Chicago	Denver	Atlanta



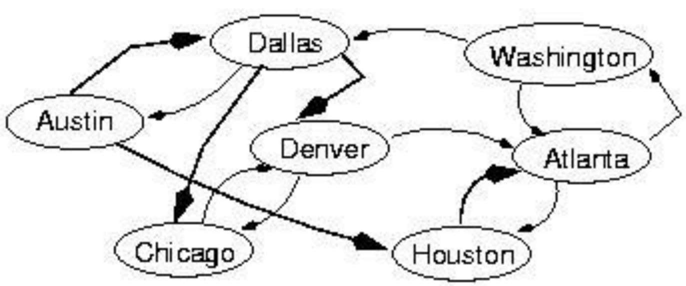
dequeue Chicago

	Denver	Atlanta	Denver



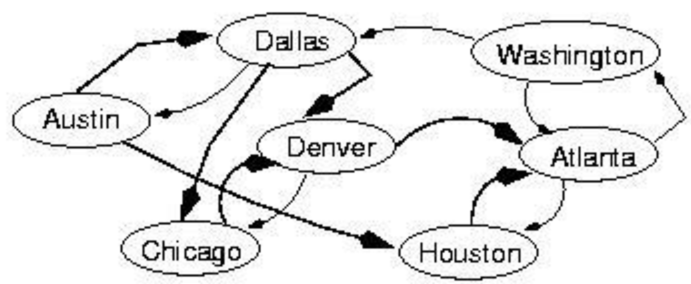
dequeue Denver

	Atlanta	Denver	Atlanta



dequeue Atlanta

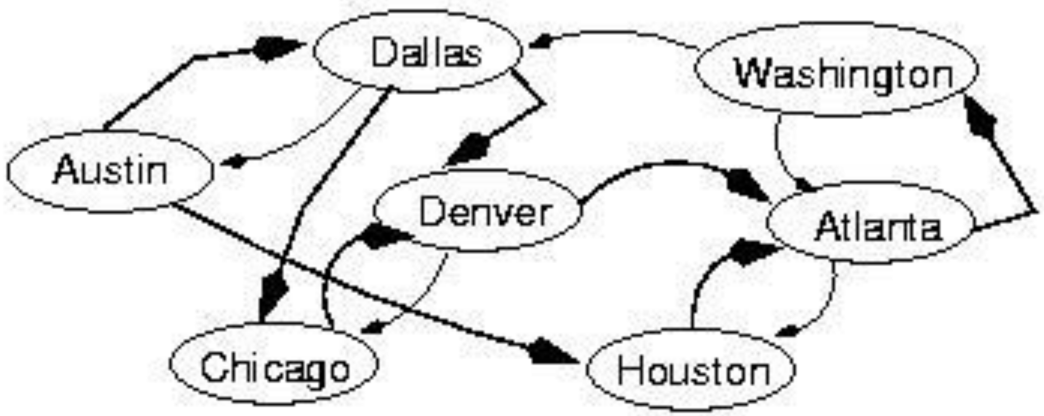
	Denver	Atlanta	Washington



dequeue Denver,

next: Atlanta

	Washington	Washington



dequeue Washington

		Washington

Single-source shortest-path problem

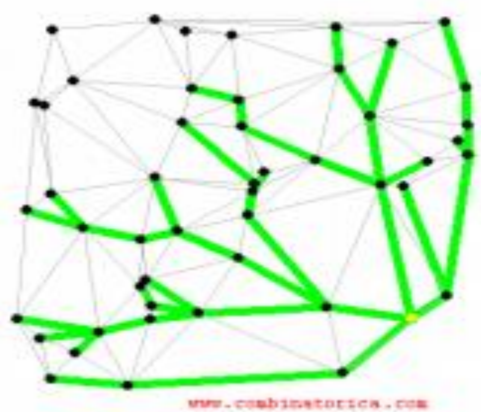
- There are multiple paths from a source vertex to a destination vertex
- *Shortest path*: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
  - Austin->Houston->Atlanta->Washington: 1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles

Single-source shortest-path problem (cont.)

The problem of finding shortest paths from a source vertex v to all other vertices in the graph.

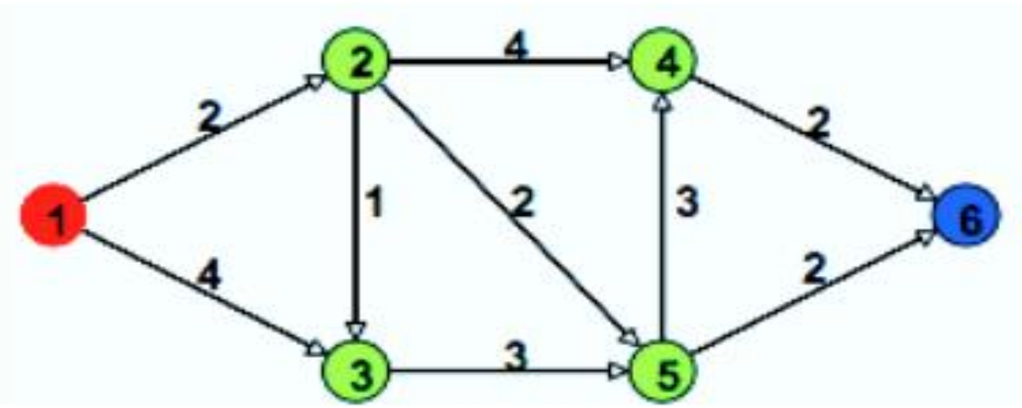
Weighted graph $G = (E,V)$

Source vertex $s \in V$ to all vertices $v \in V$



Dijkstra’s Algorithm

- Solution to the single-source shortest path problem in graph theory
  - Both directed and undirected graphs
  - All edges must have nonnegative weights
  - Graph must be connected

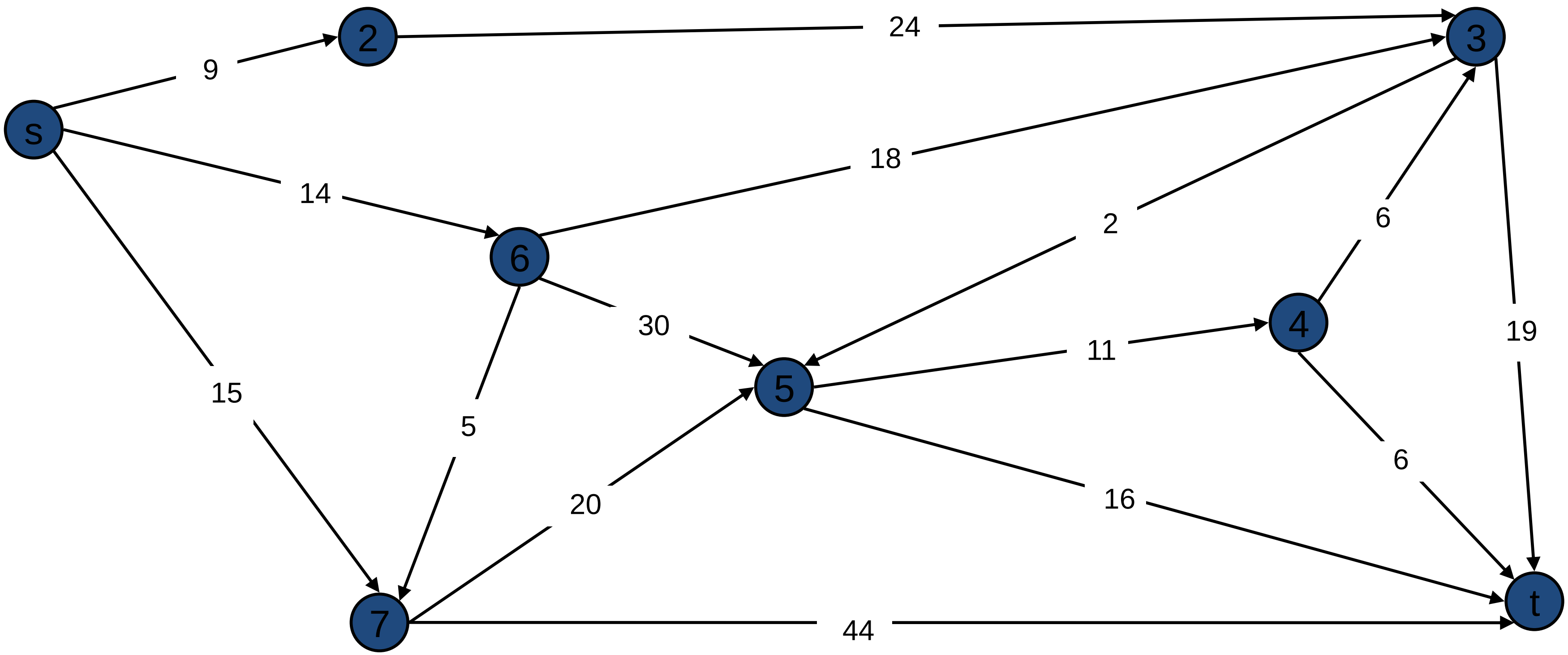


Pseudocode

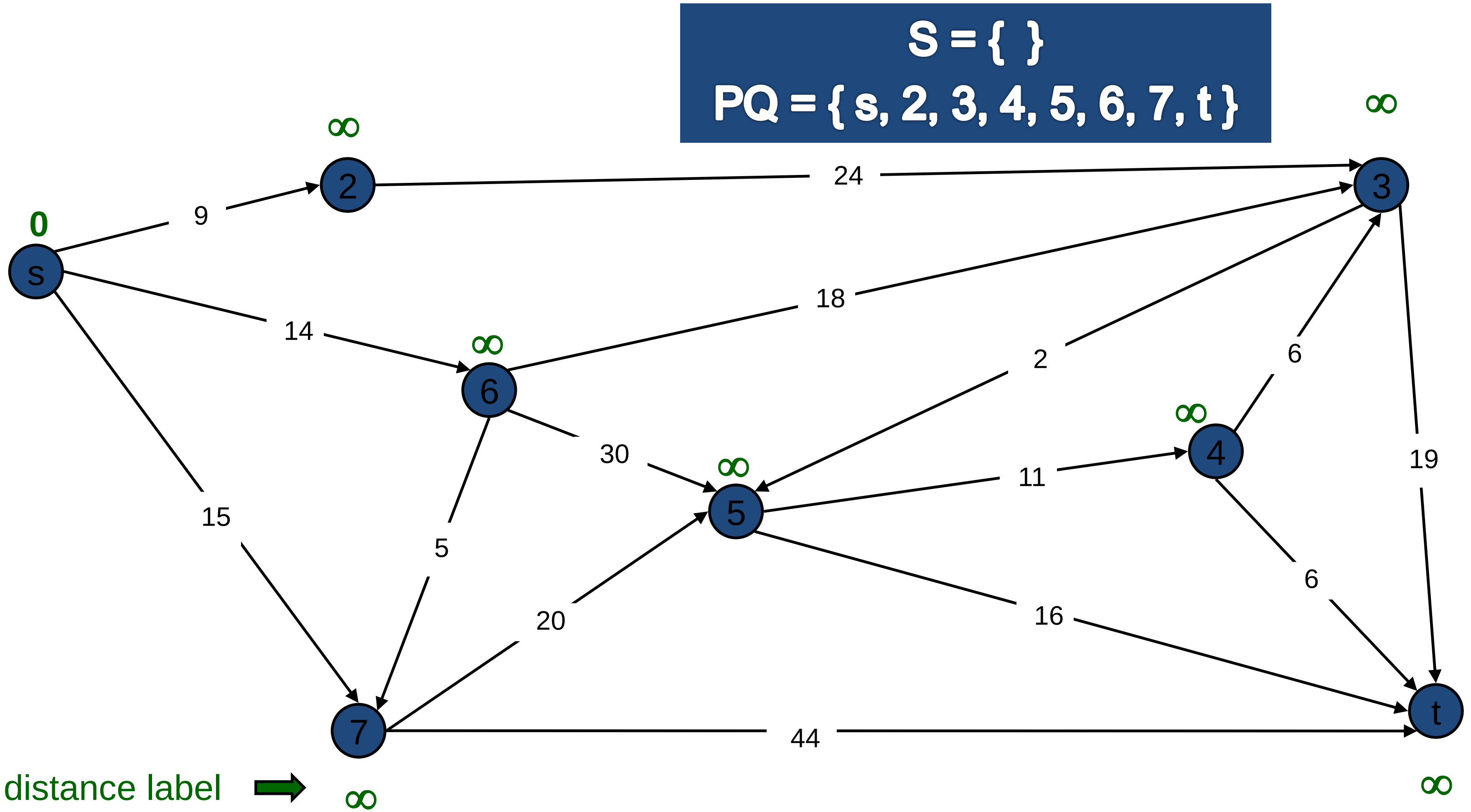
```
dist[s] ←0                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
     do  dist[v] ←∞                         (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
    S←S∪{u}                                 (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do  if   dist[v] > dist[u] + w(u, v)        (if new shortest path found)
                   then     d[v] ←d[u] + w(u, v)      (set new value of shortest path)
                                                      (if desired, add traceback code)
return dist
```

Dijkstra's Shortest Path Algorithm

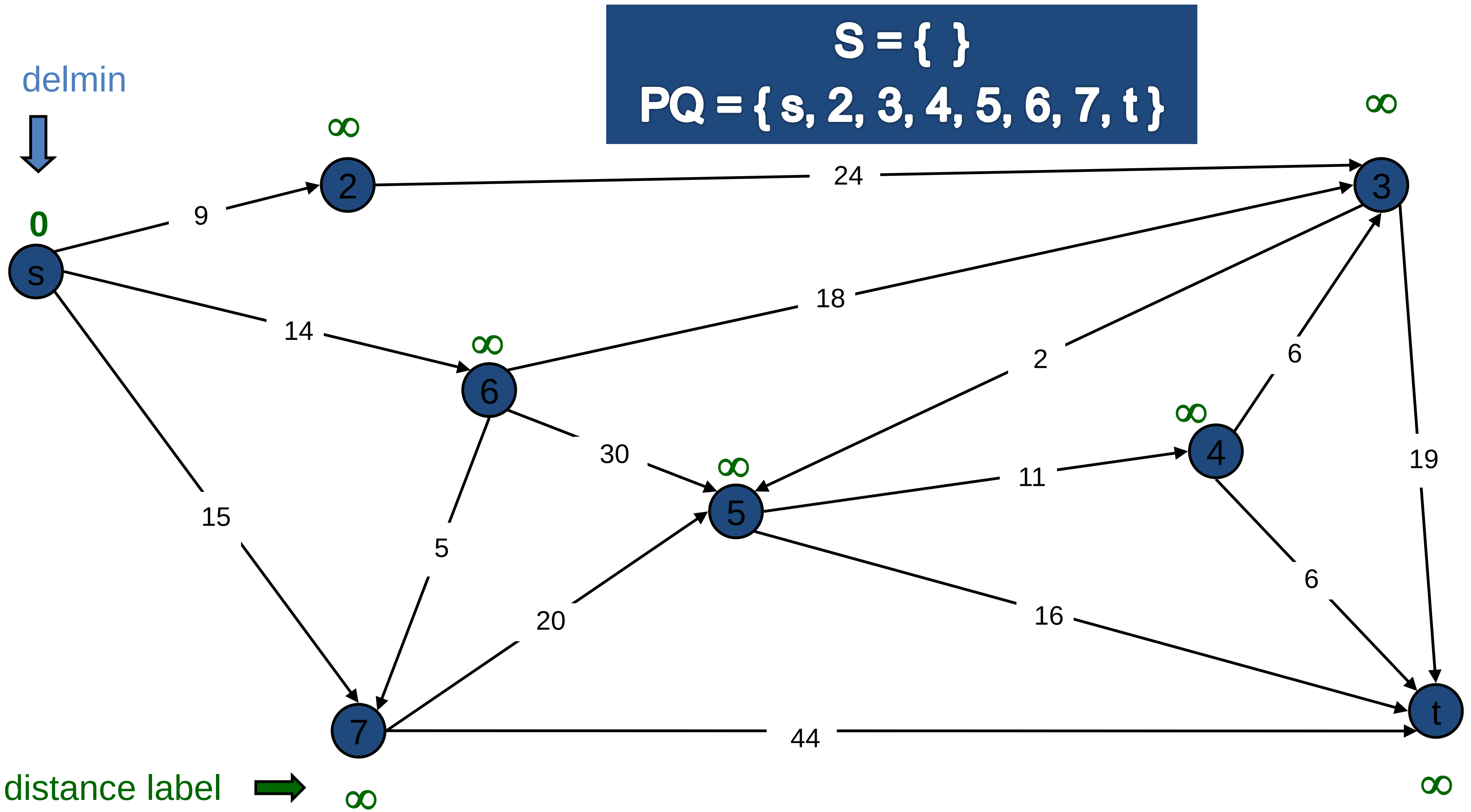
- Find shortest path from s to t.



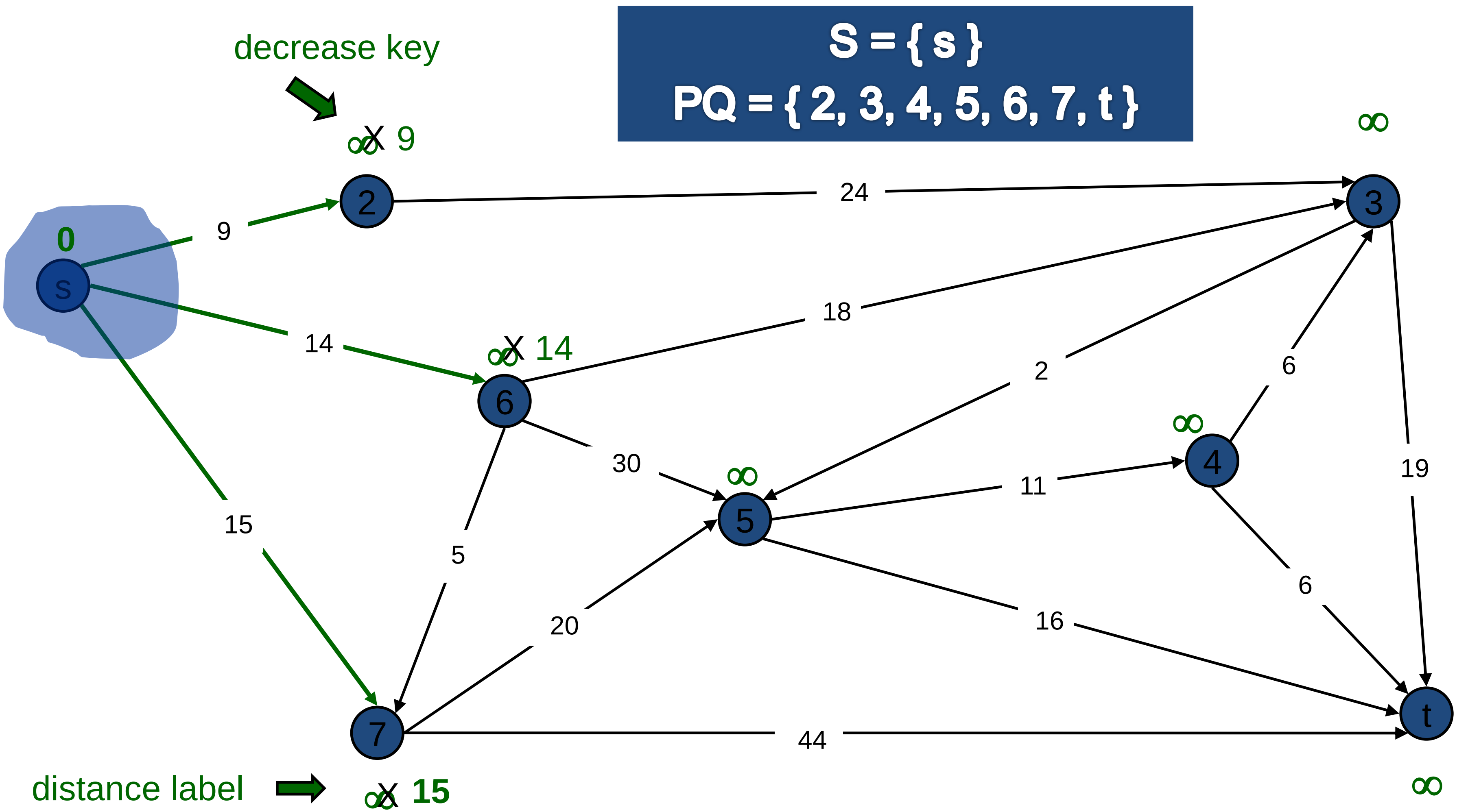
Dijkstra's Shortest Path Algorithm



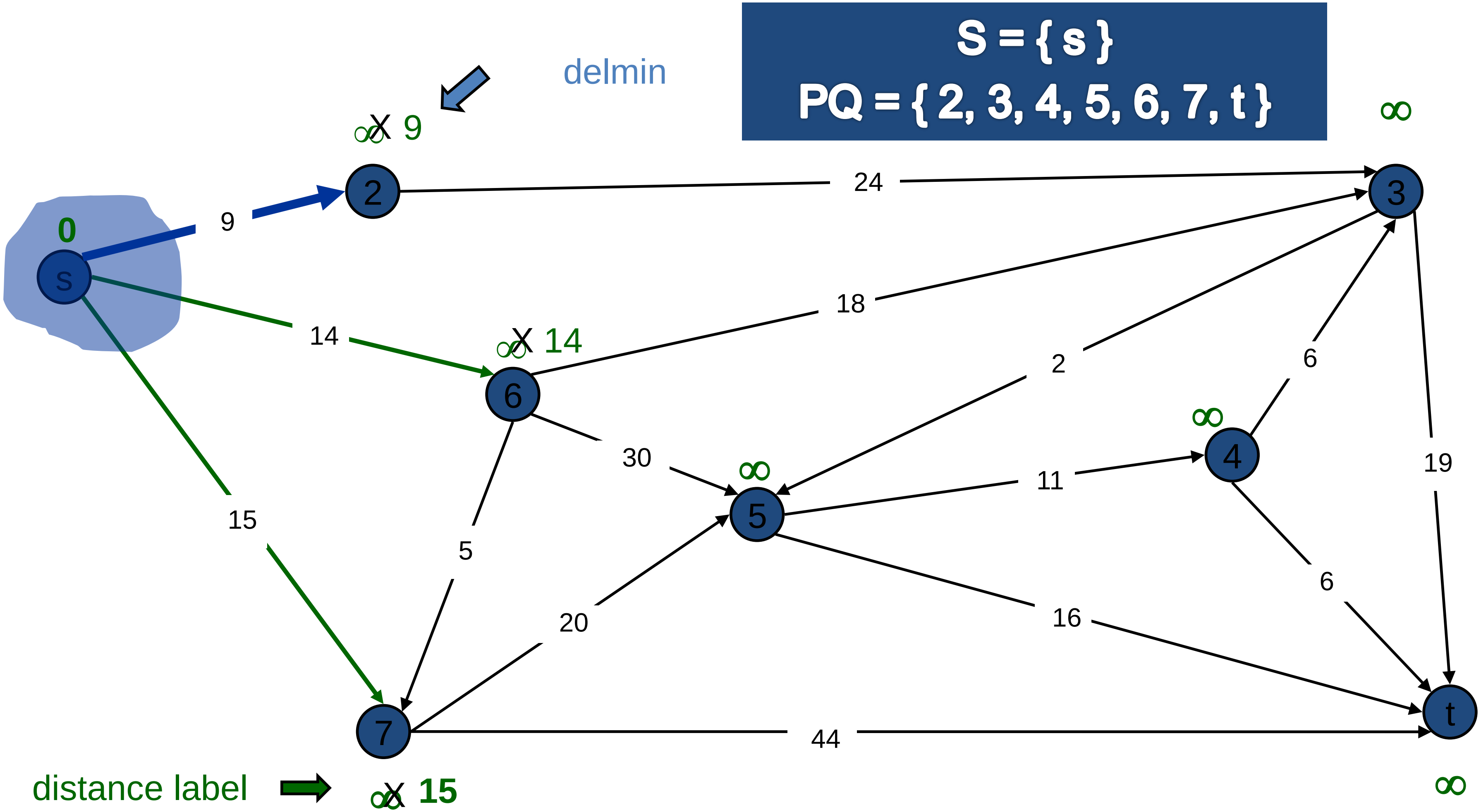
Dijkstra's Shortest Path Algorithm



Dijkstra's Shortest Path Algorithm

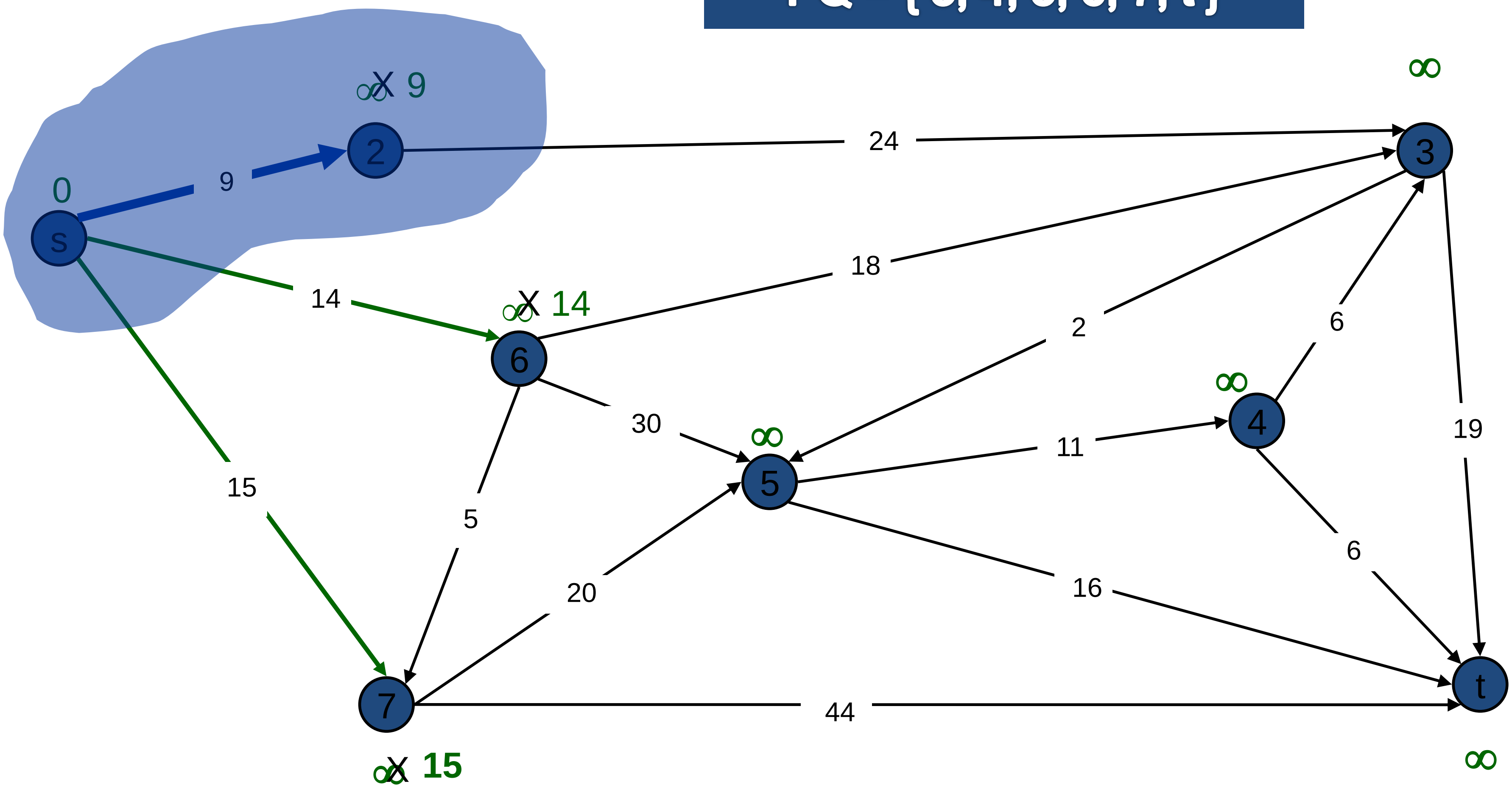


Dijkstra's Shortest Path Algorithm



Dijkstra's Shortest Path Algorithm

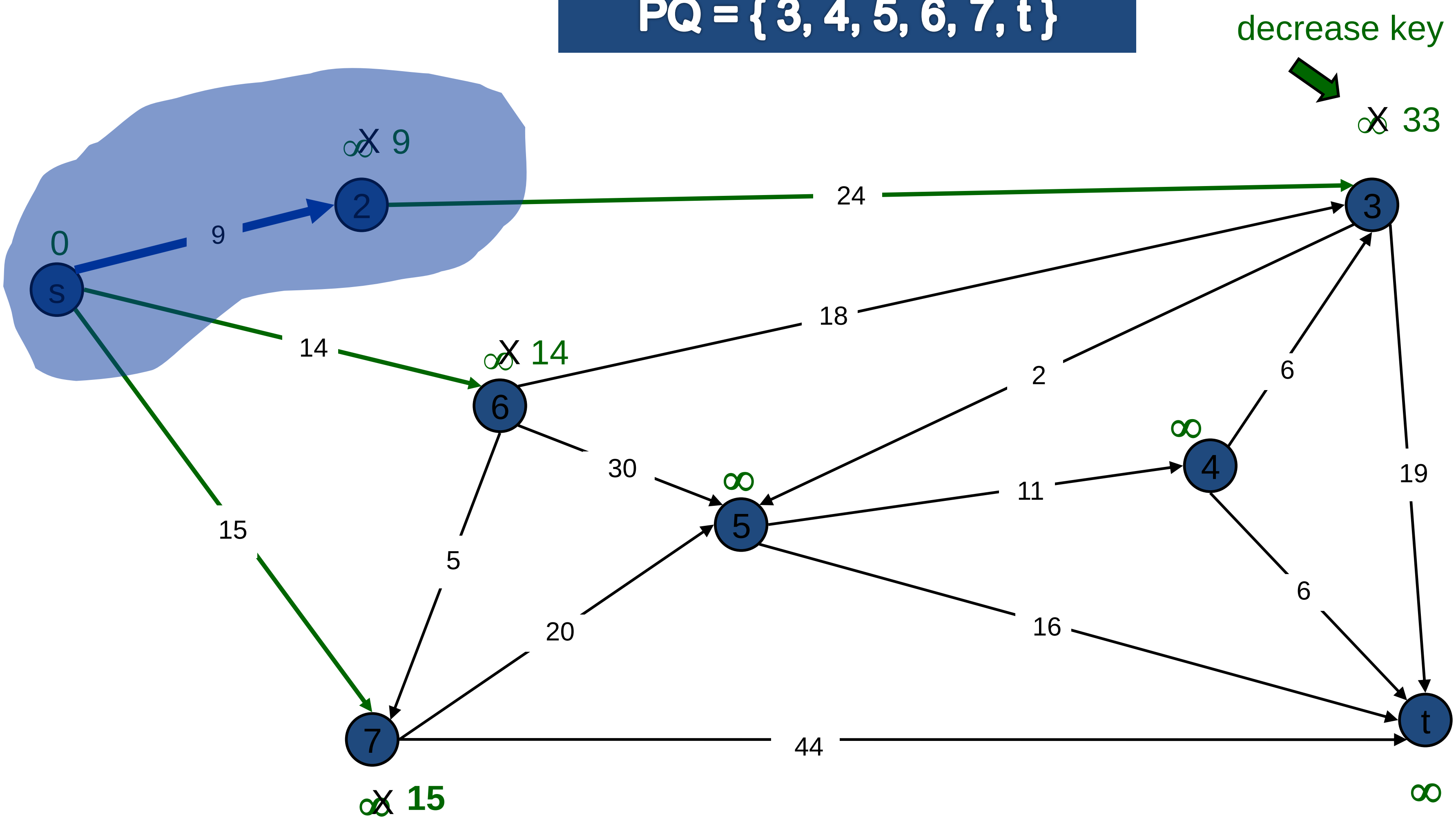
S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



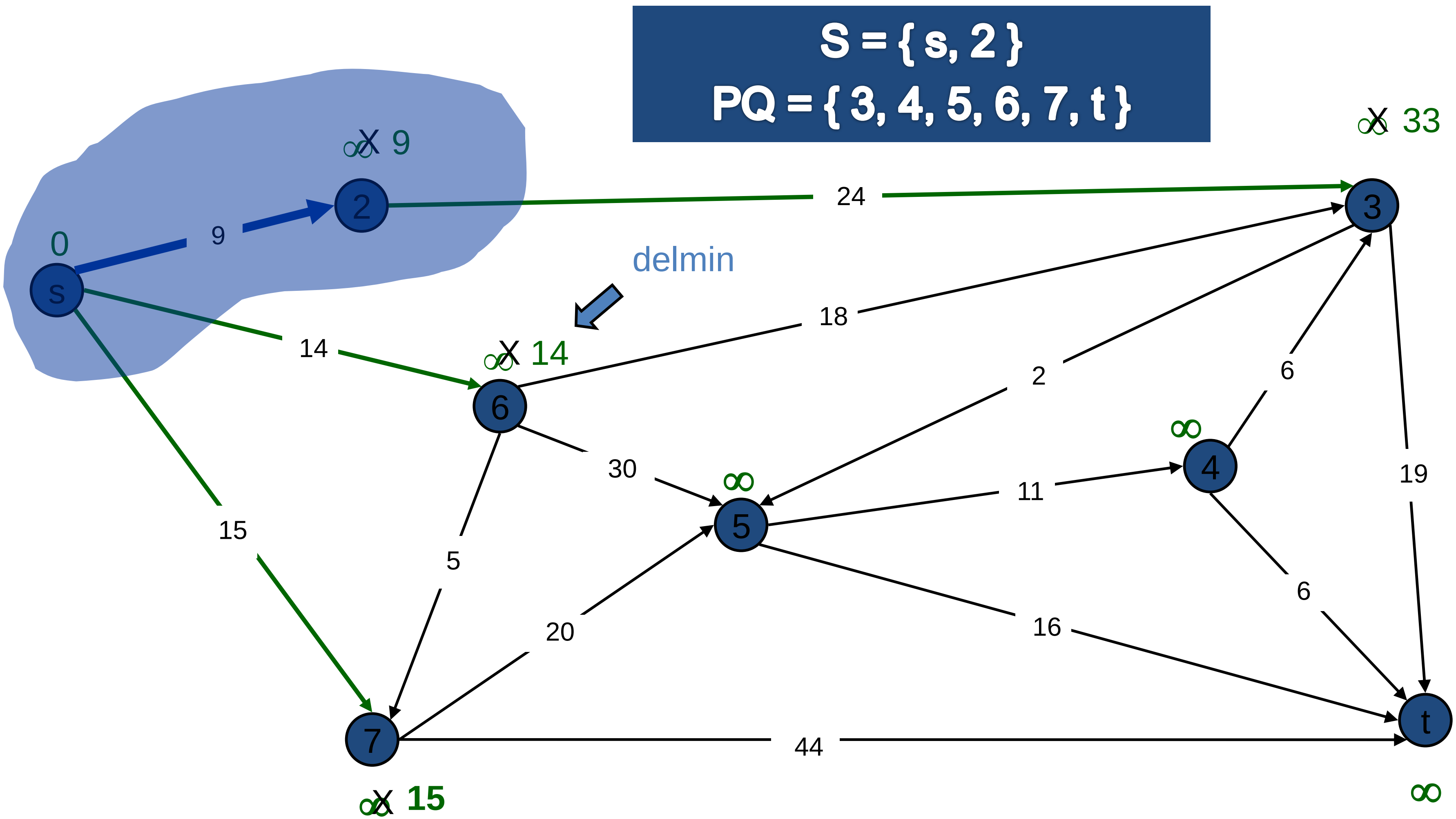
Dijkstra's Shortest Path Algorithm

S = { s, 2 }

PQ = { 3, 4, 5, 6, 7, t }

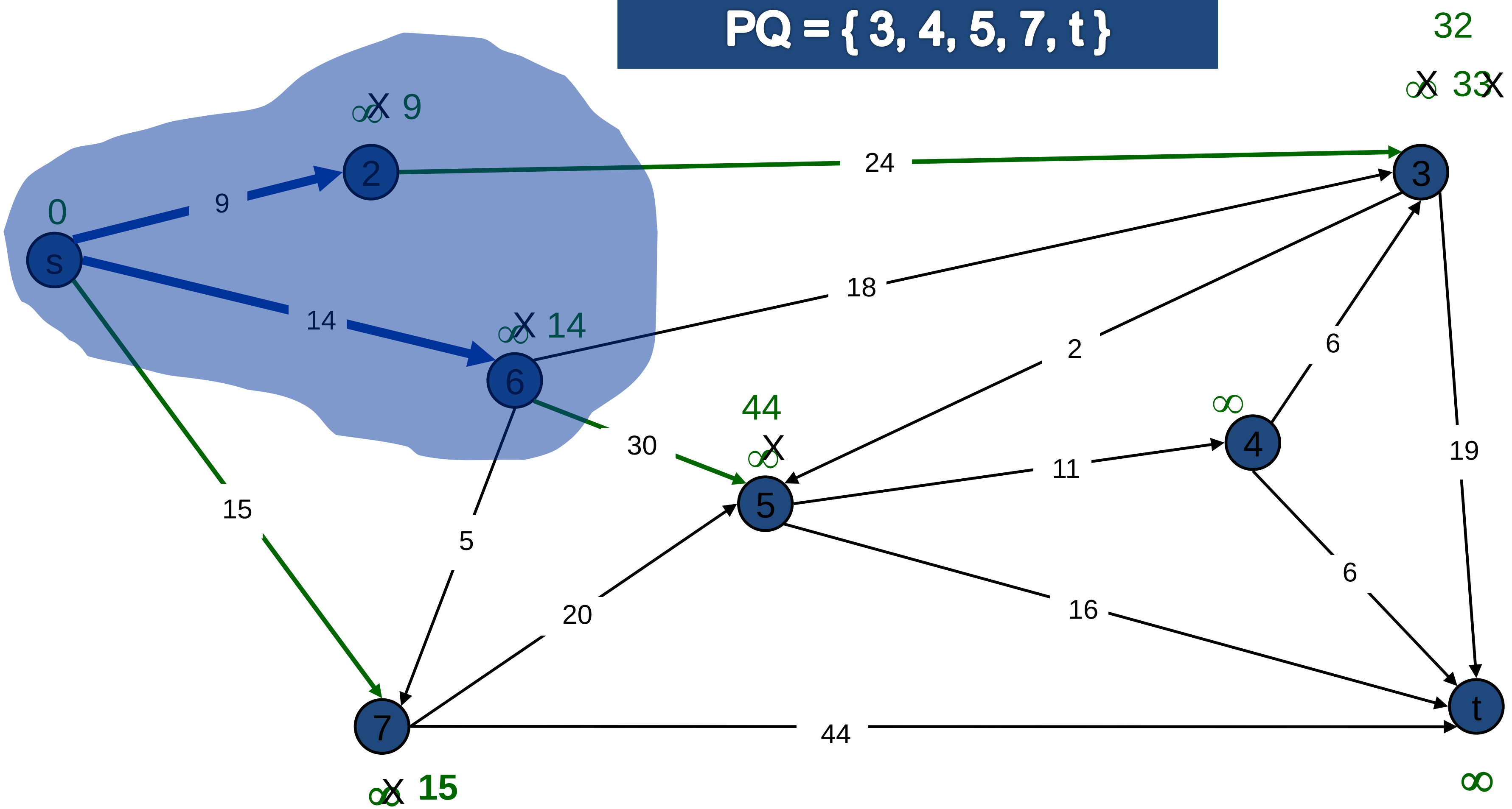


Dijkstra's Shortest Path Algorithm



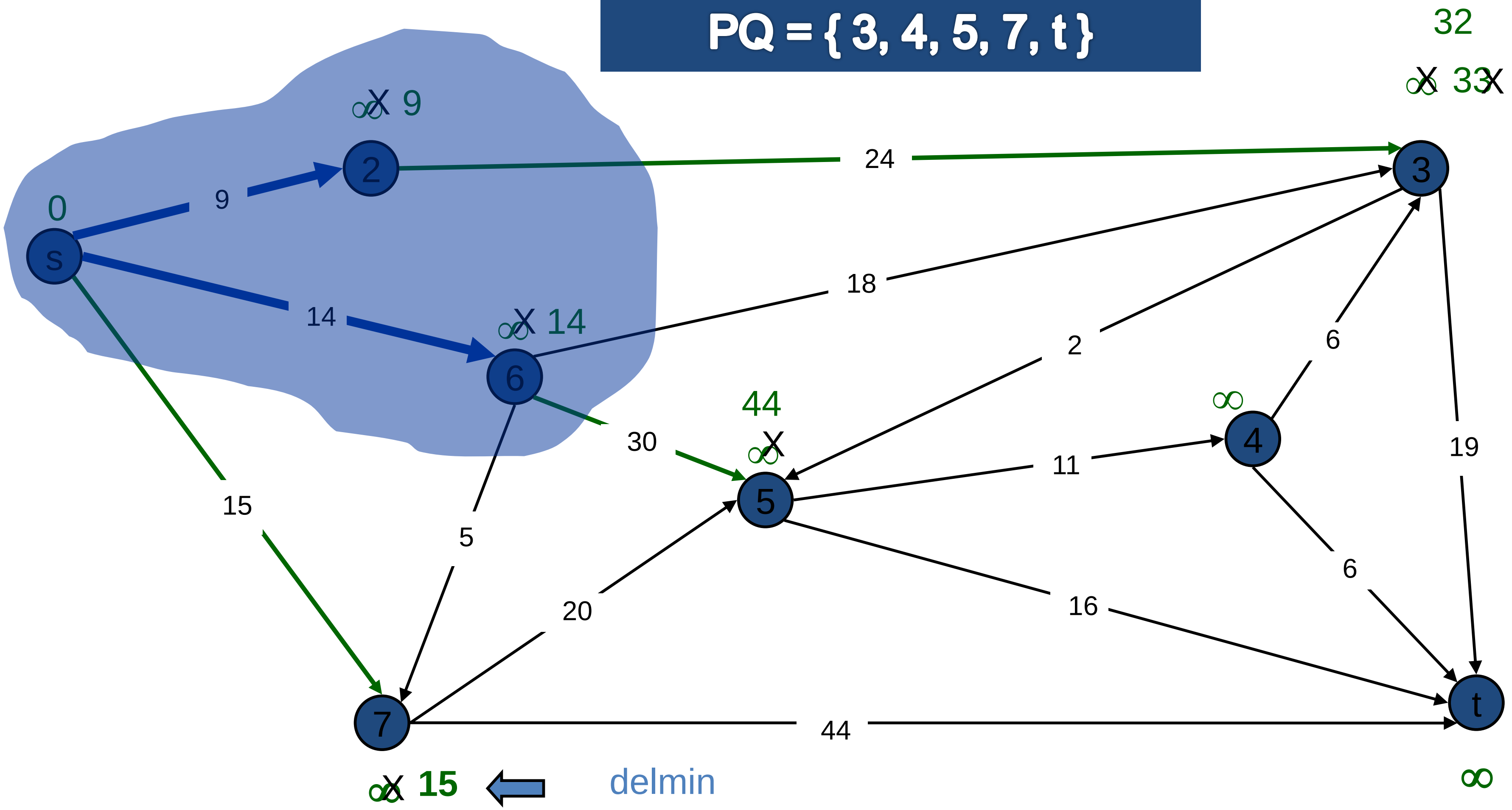
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }

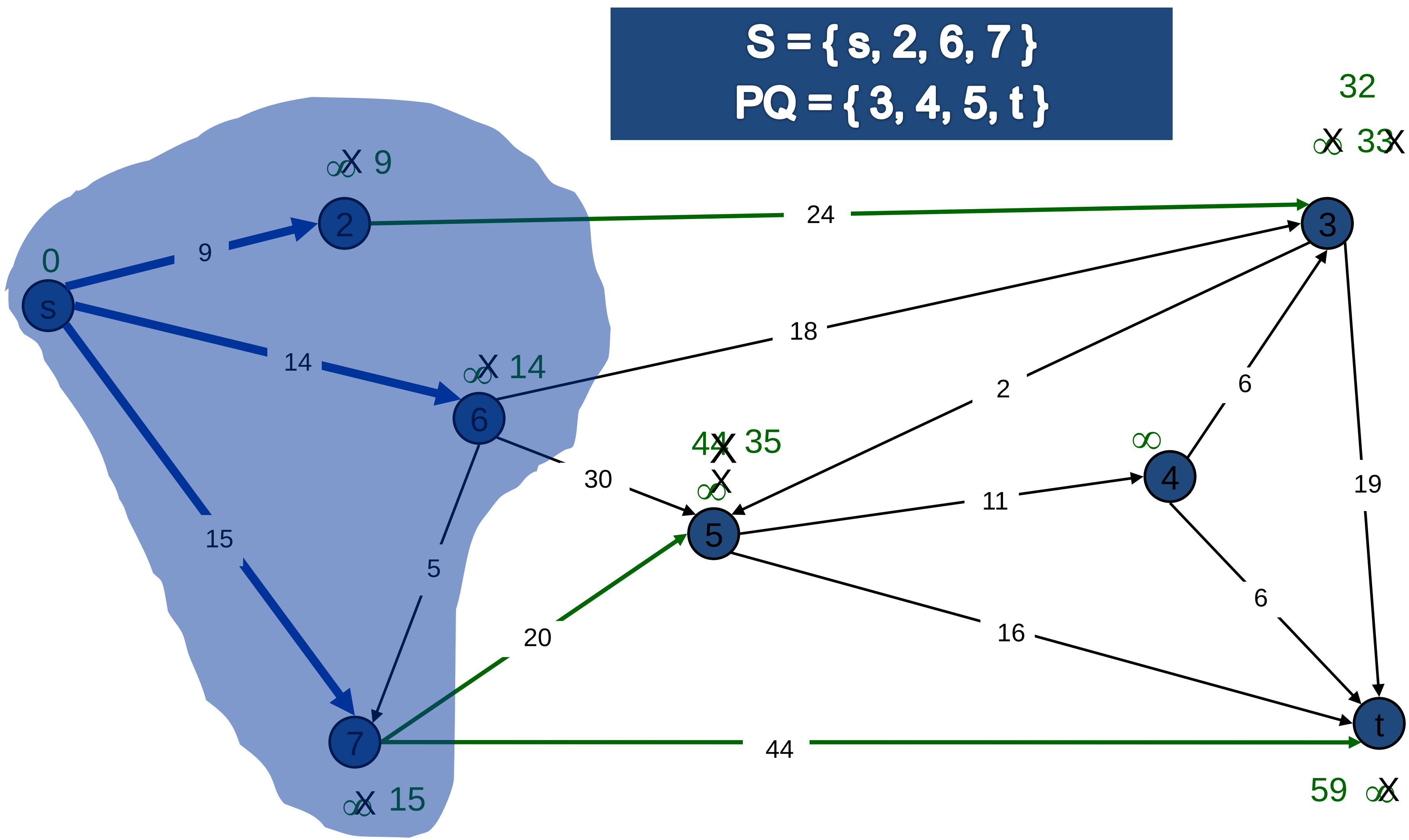


Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }

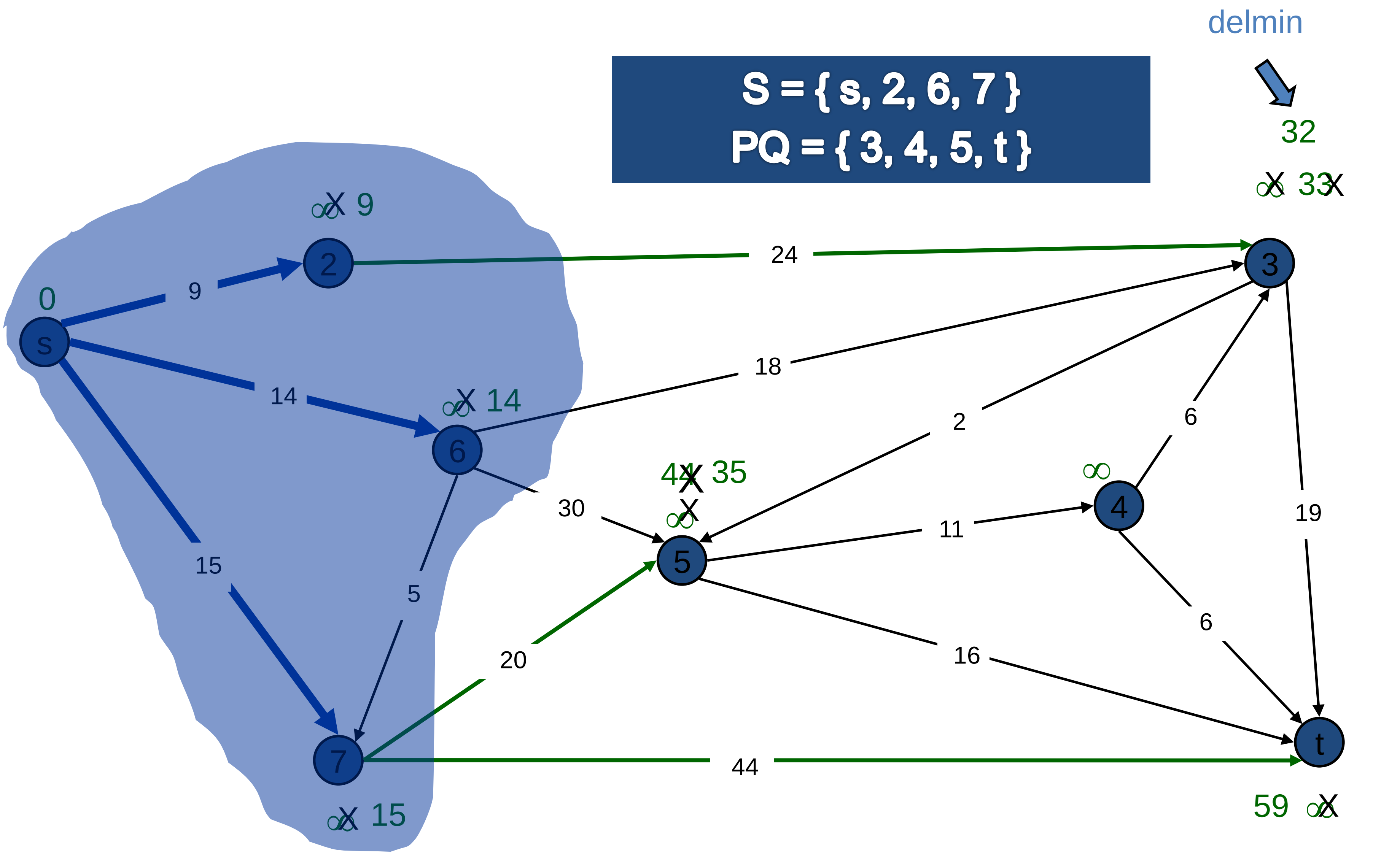


Dijkstra's Shortest Path Algorithm



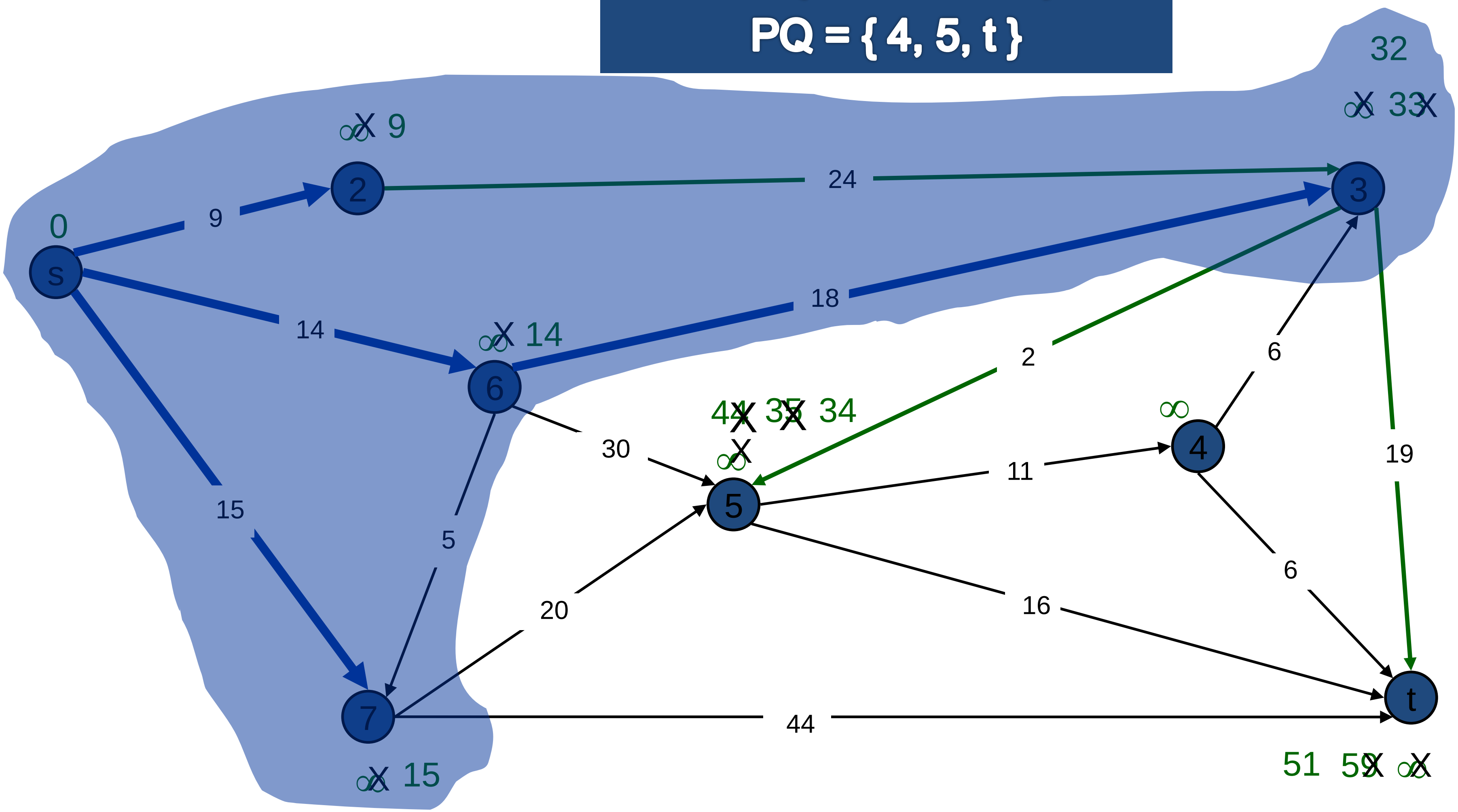
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }

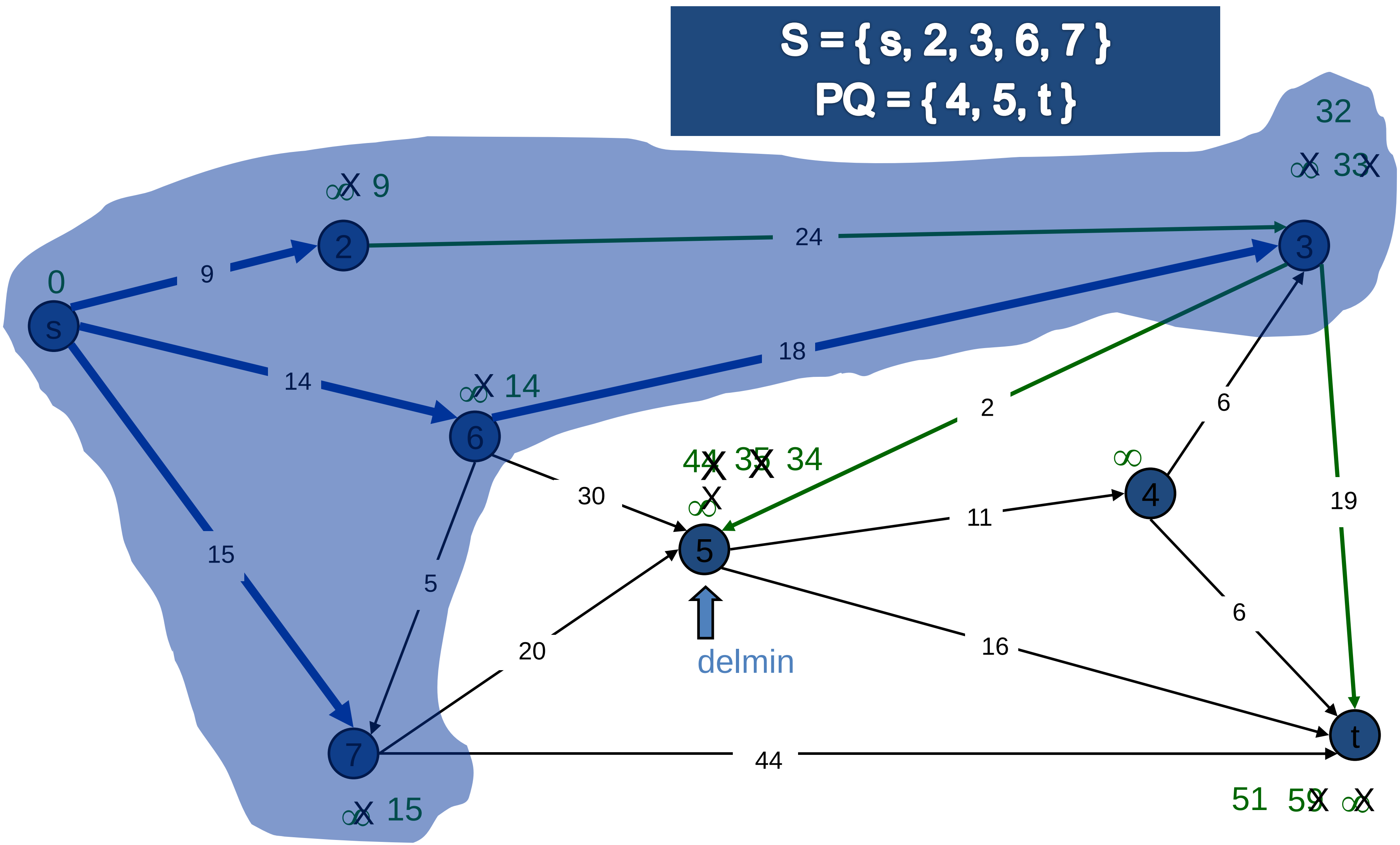


Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }

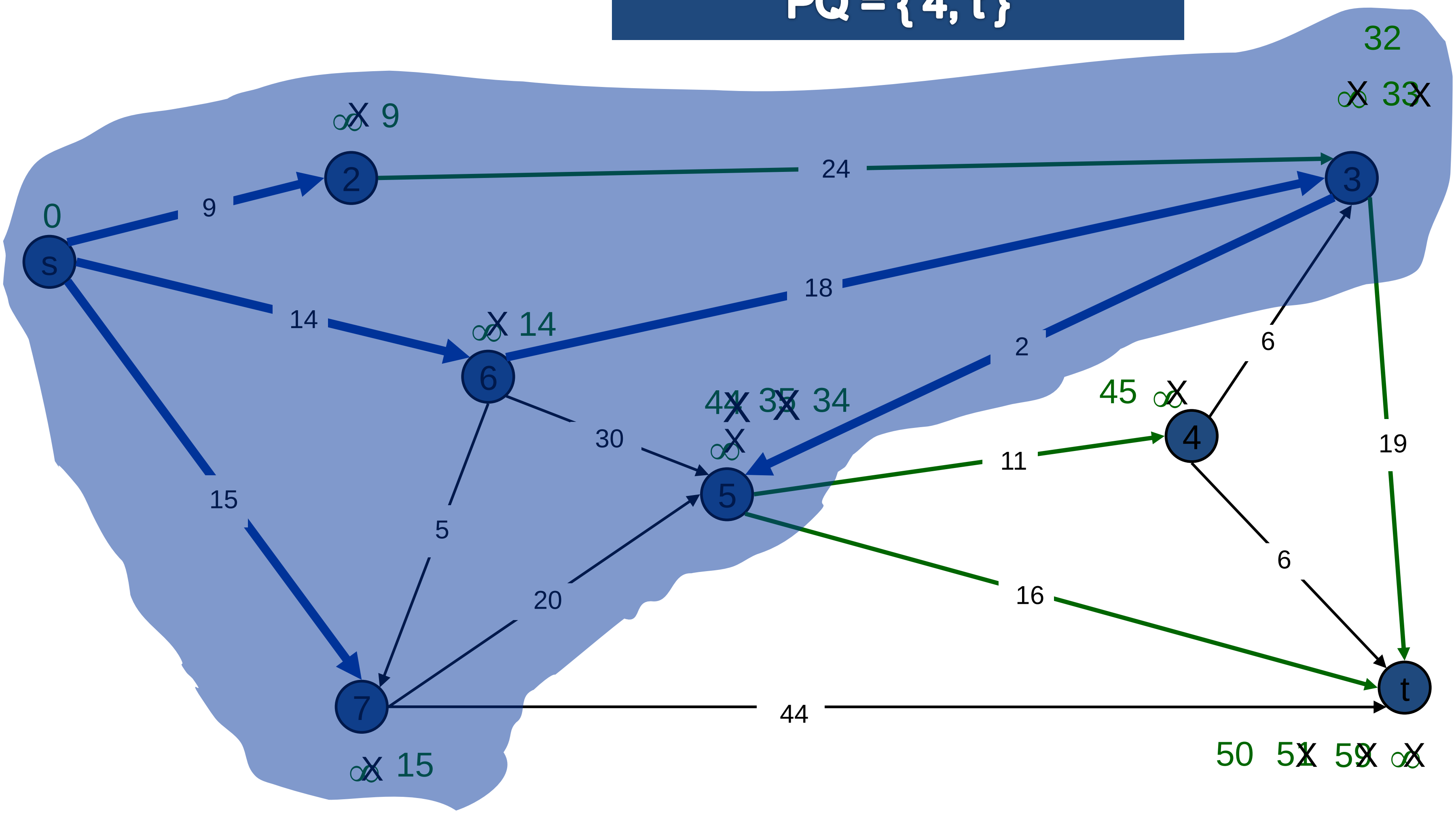


Dijkstra's Shortest Path Algorithm



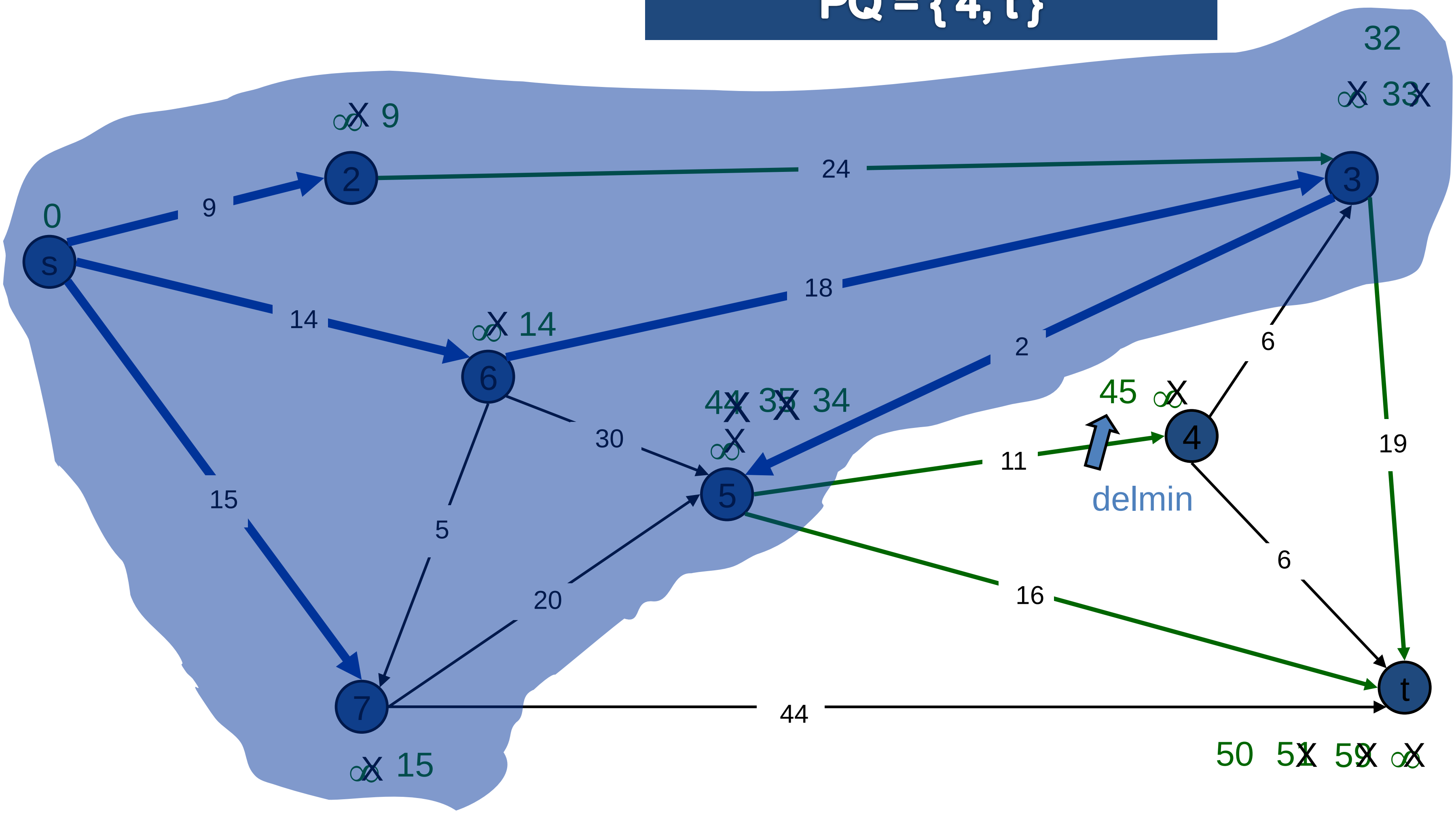
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



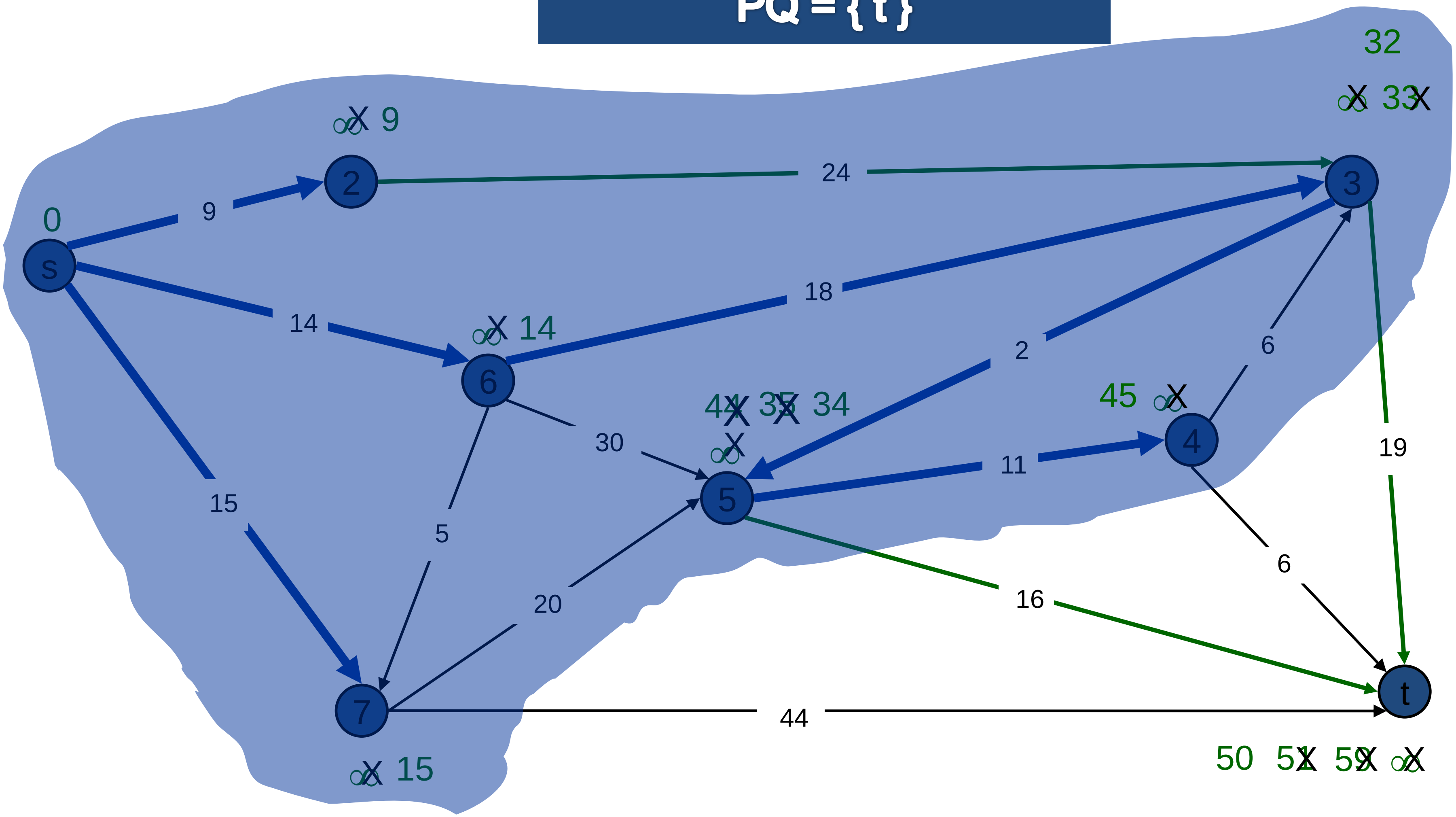
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



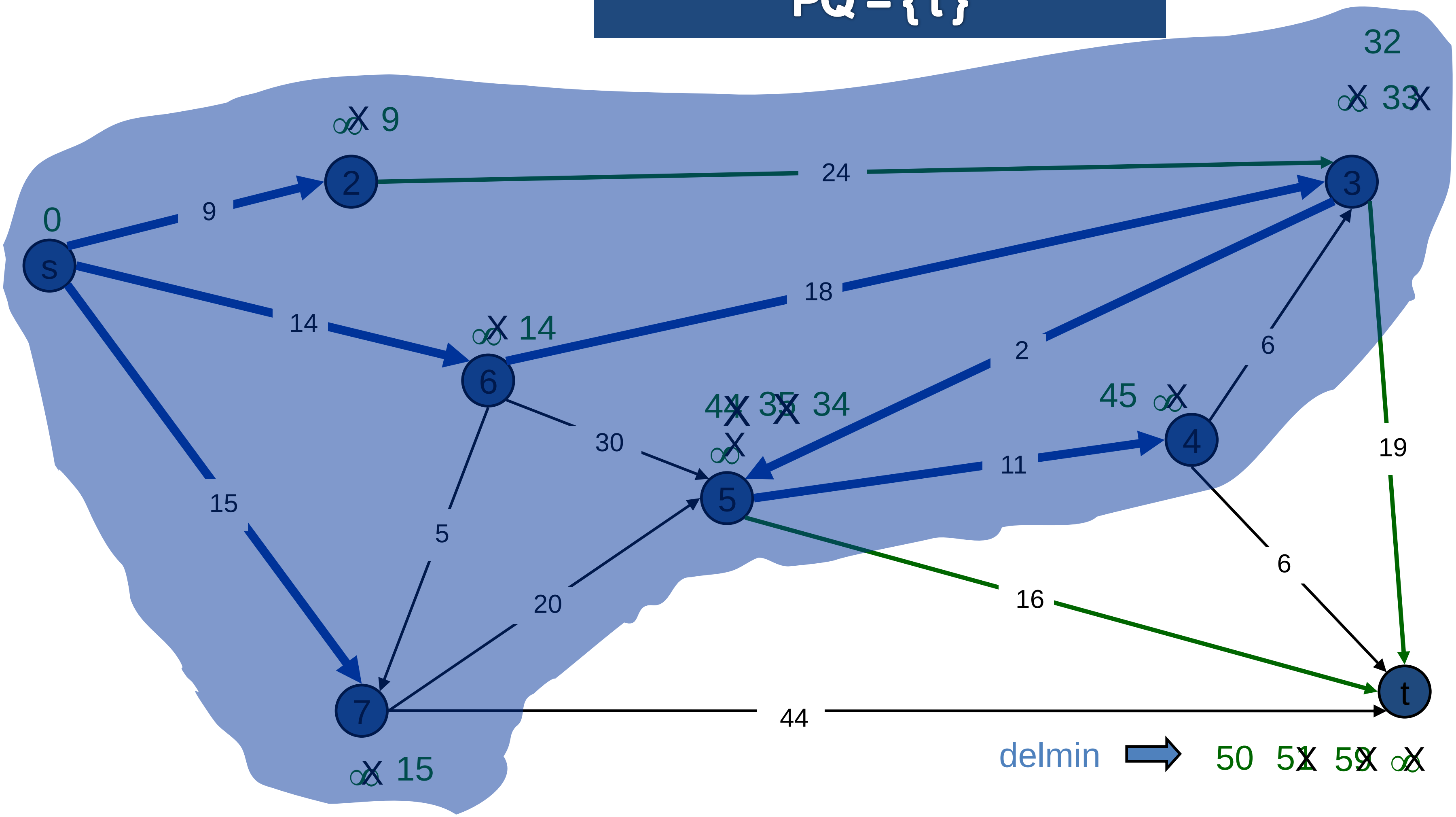
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



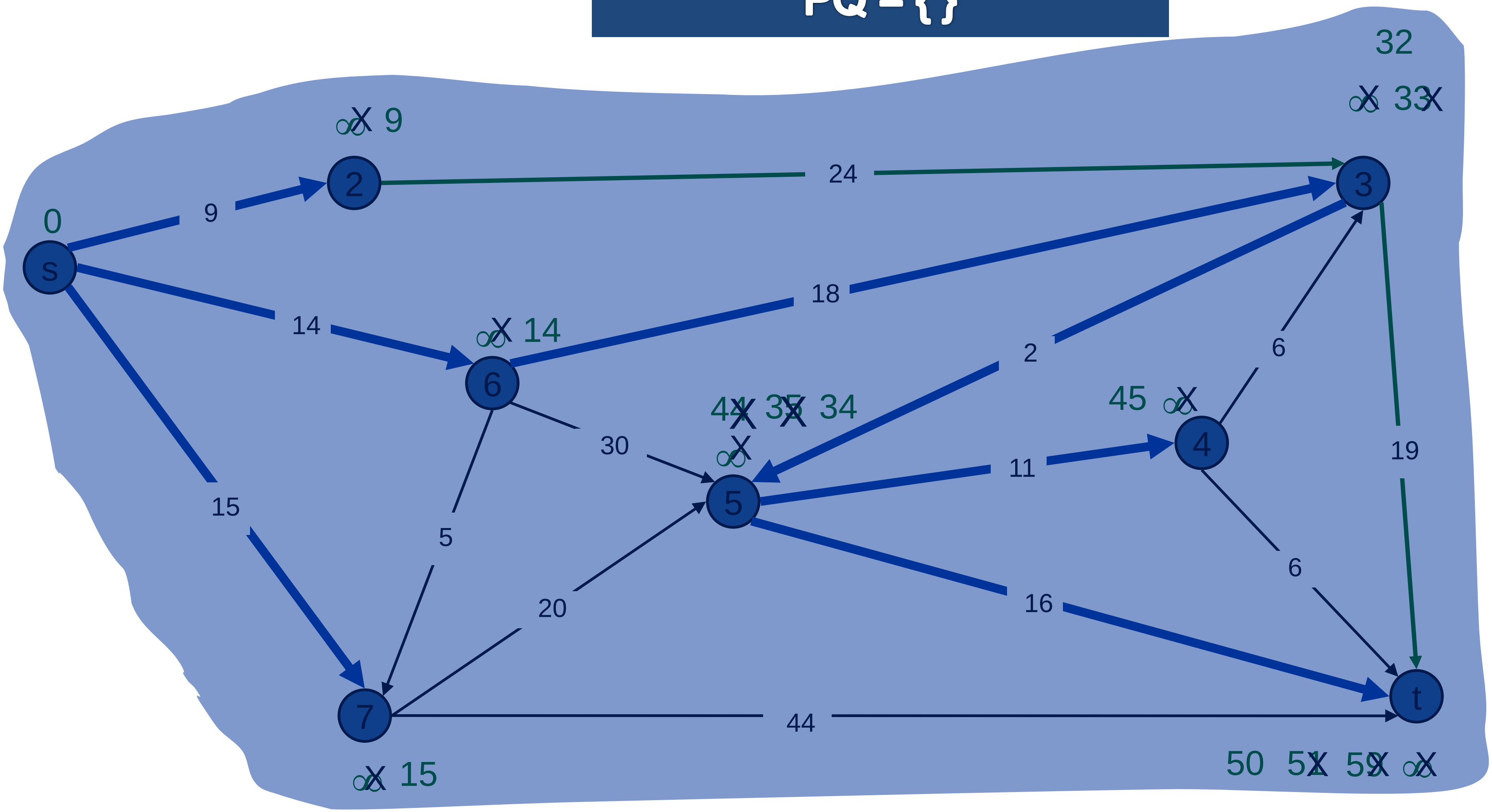
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }

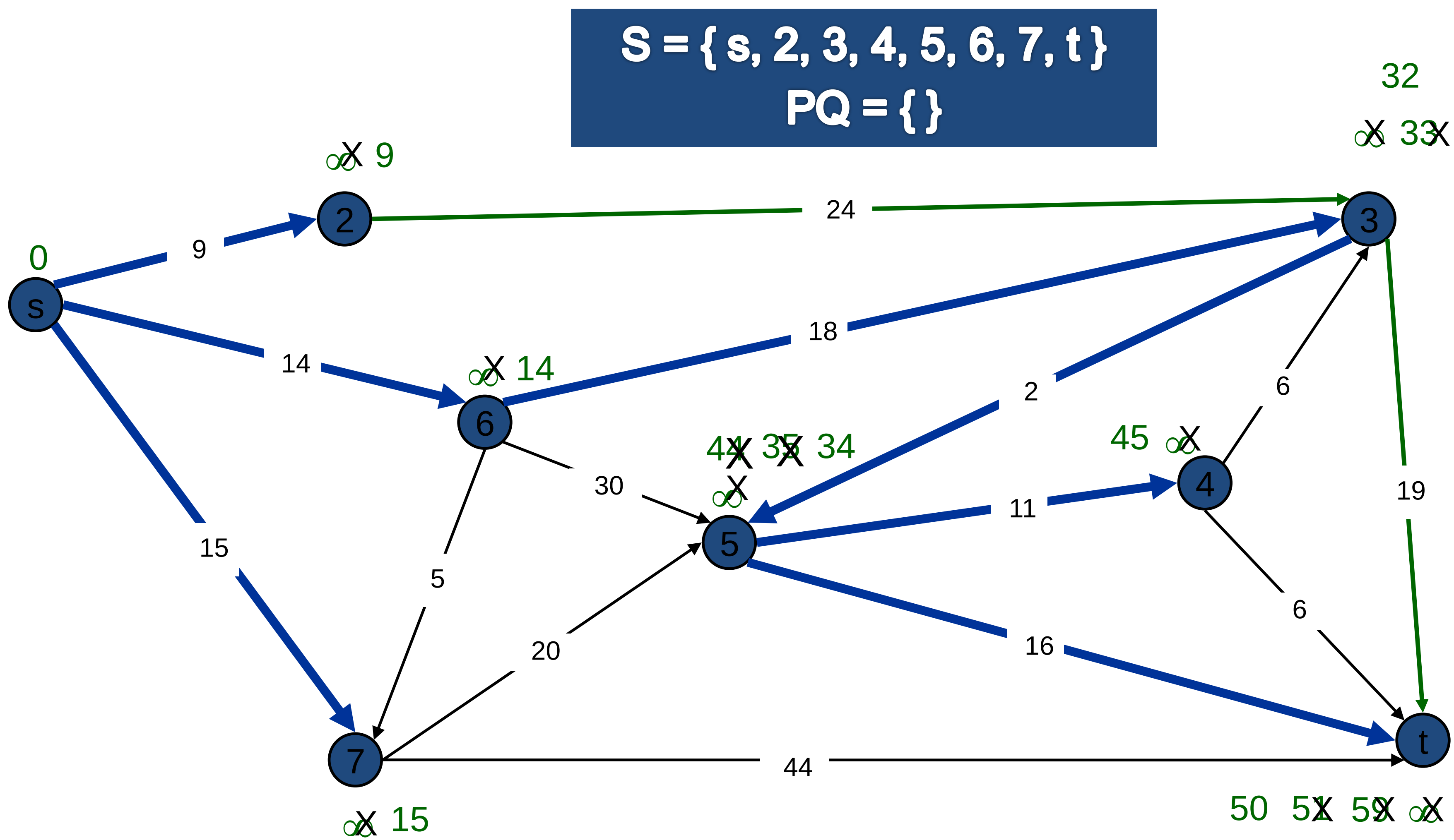


Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }

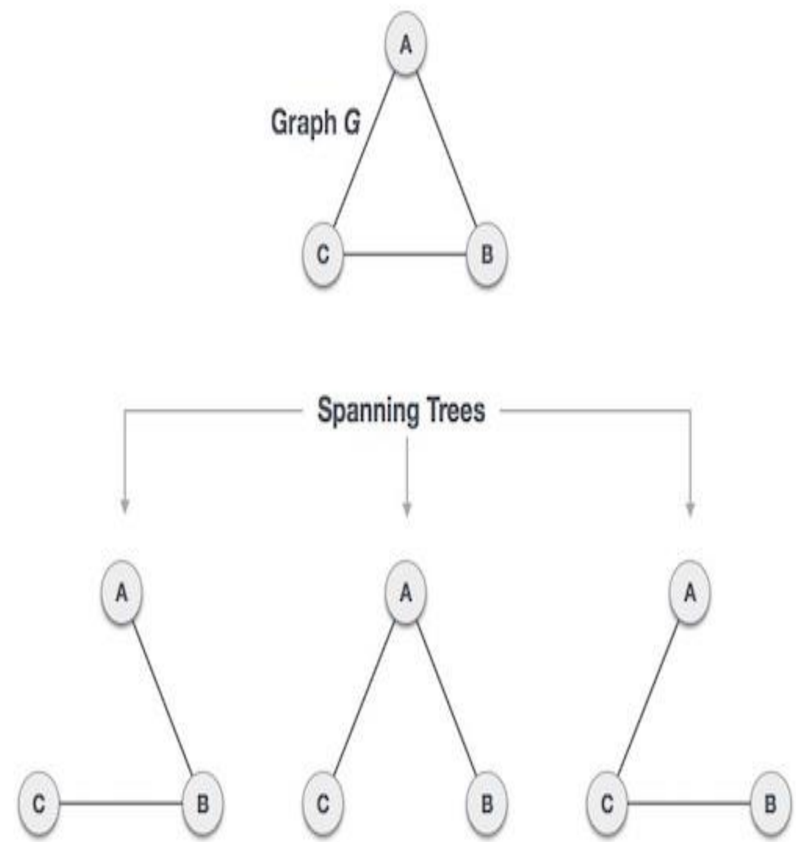


Dijkstra's Shortest Path Algorithm



Spanning Tree

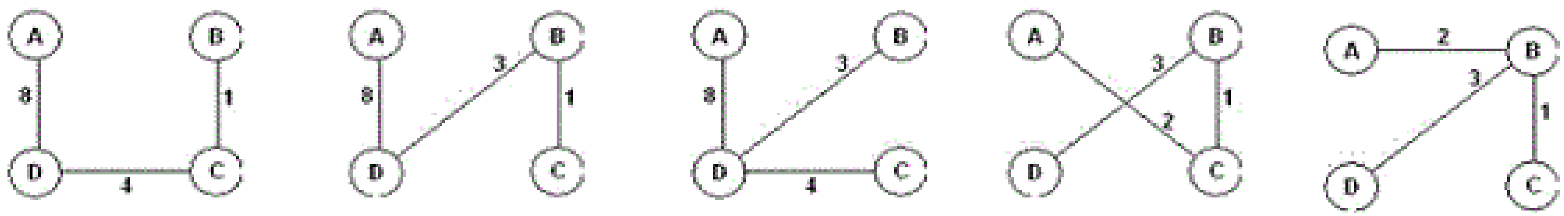
- Given an undirected and connected graph G(V,E)
- A spanning tree is a sub graph of G, which must include all the vertices.



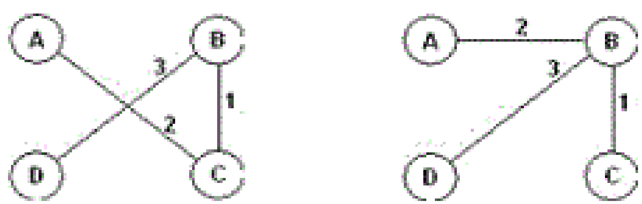
Minimum Spanning Tree

- A minimum spanning tree of a connected undirected weighted graph.
- It has a subset of the edges from the original graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Has 16 spanning trees. Some are:



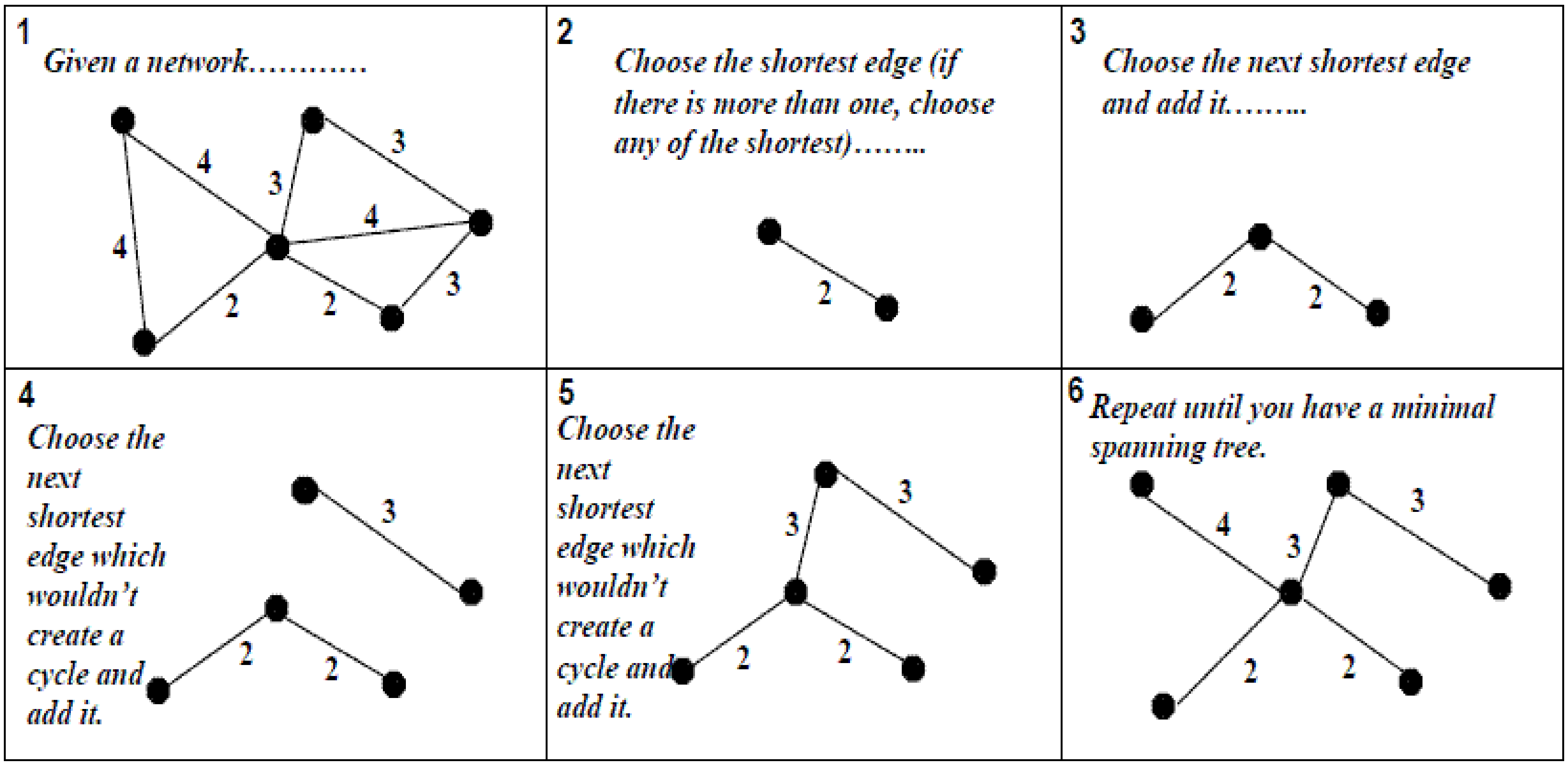
The graph has two minimum-cost spanning trees, each with a cost of 6:



Algorithms for MST

- Kruskal’s Algorithm
- Prim’s Algorithm

Kruskal's Algorithm



Prim's Algorithm