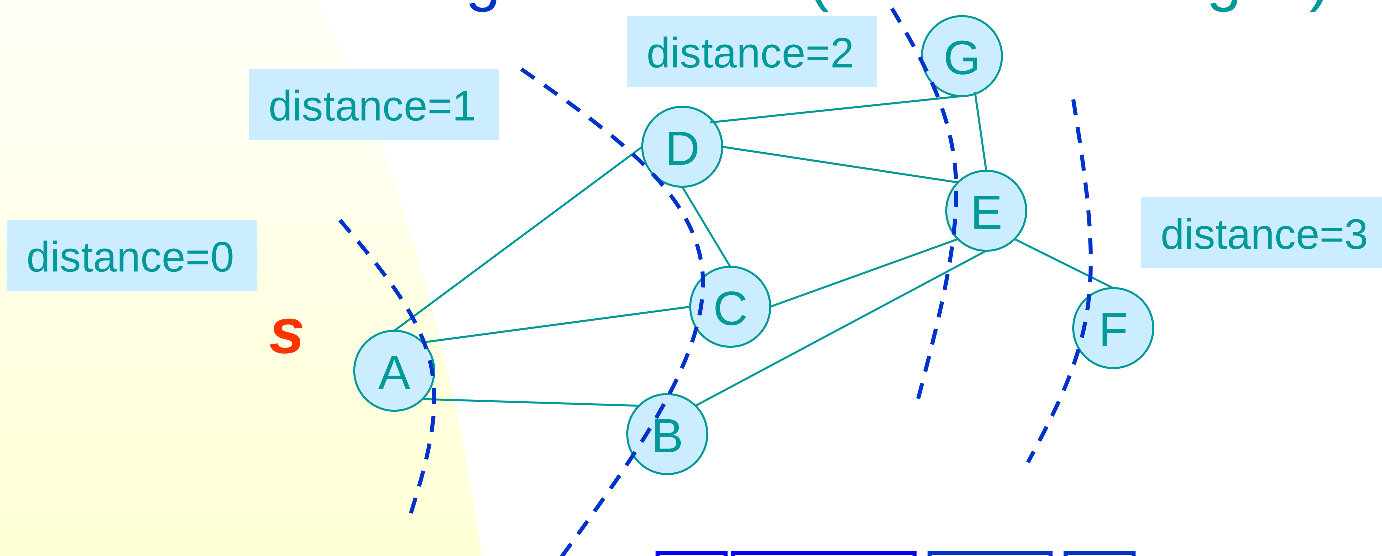
BFS : Breadth-First Search

- Given any source *s* (vertex), BFS visits the other vertices at increasing distances (number of edges) from *s*.



BFS visit sequence = A, D, B, C, G, E, F

BFS visit sequence = A, C, D, B, E, G, F

BFS : Breadth-First Search

- The situation is pretty much like a water drop falling into a pond.

- At all times, BFS maintains a subset of vertices at the frontier. This frontier moves outward to discover new vertices. Algorithmically, this frontier is maintained as a ***queue*** (FIFO) of vertices. Those vertices in the queue are waiting to be visited.

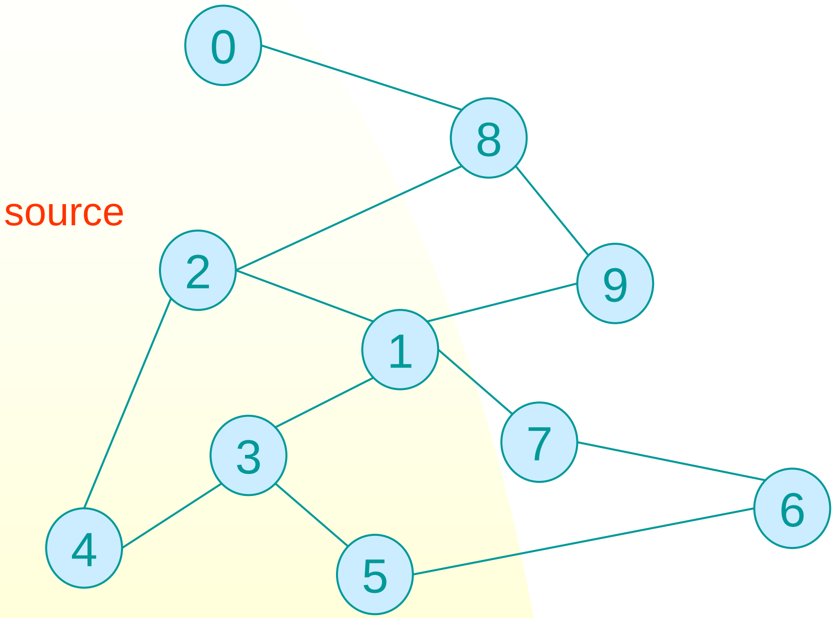
- In doing so, BFS discovers **(shortest)** paths from *s* to other vertices.

Algorithm

Algorithm *BFS*(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

for each vertex v
  do $\mathit{flag}[v]$:= false; initialization
Q = empty queue;
$\mathit{flag}[s]$:= true;
enqueue(Q, s);
while Q is not empty
  do $\boxed{v}$:= $\boxed{\mathit{dequeue}(Q)}$;
    for each $\boxed{w}$ adjacent to v *v* is visited here.
      do if $\mathit{flag}[w]$ = false
        then $\mathit{flag}[w]$:= true;
          $\boxed{\mathit{enqueue}}(Q, \boxed{w})$

Example



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Flag Table (T/F)

0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

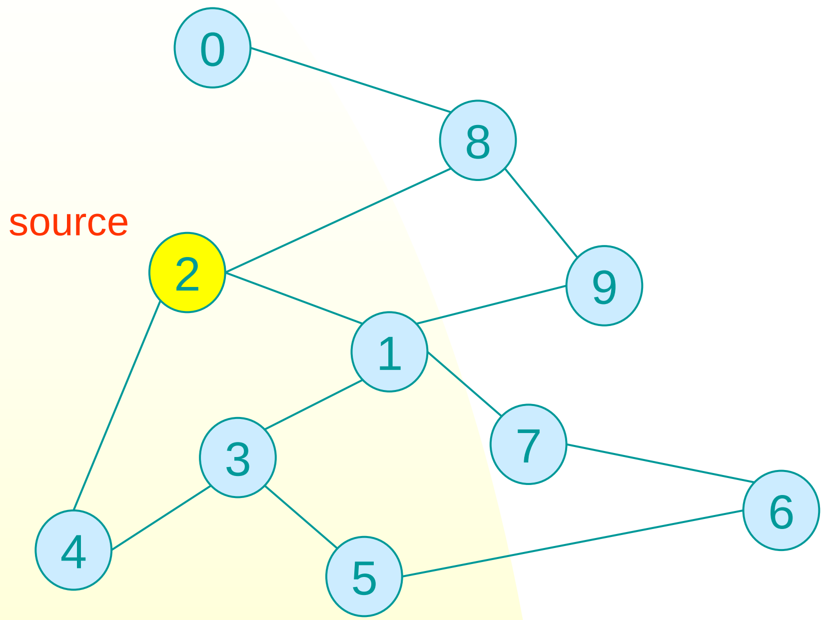
Initialize visited table (all empty F)

visit sequence ={ }

Q={ }

Initialize **Q** to be empty

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

mark Flag[2].

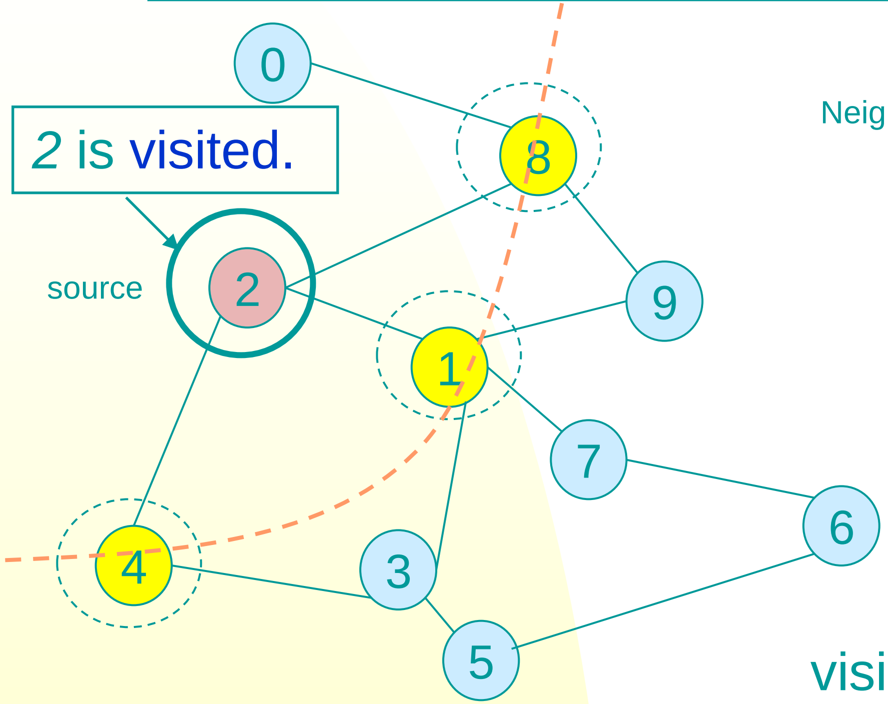
visit sequence ={ }

Q={ 2 }

Place source 2 on the queue.

Example

a frontier of vertices waiting to be visited (marked yellow)



Adjacency List

Neighbors → 2

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

visit sequence ={2}

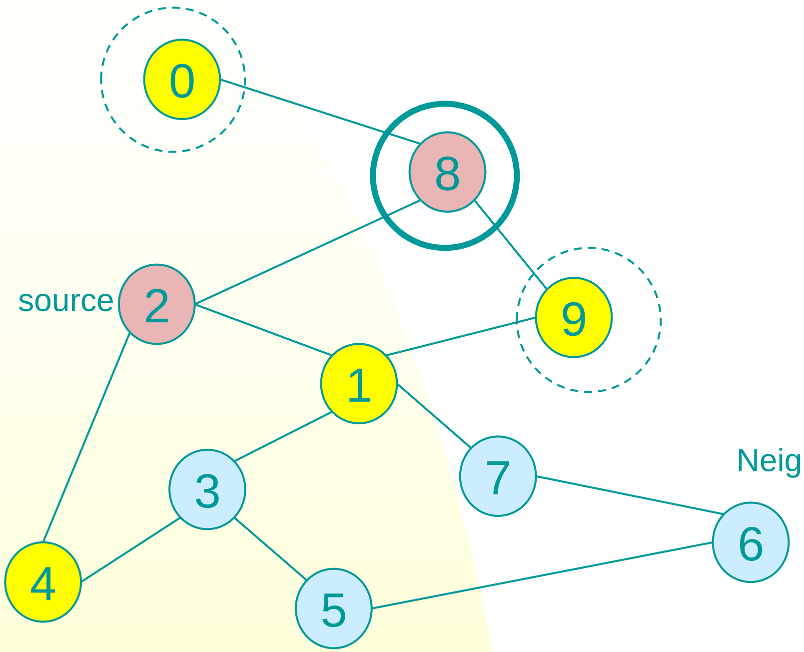
Mark unmarked neighbors.

Q={2} → { **8, 1, 4** }

Dequeue 2.

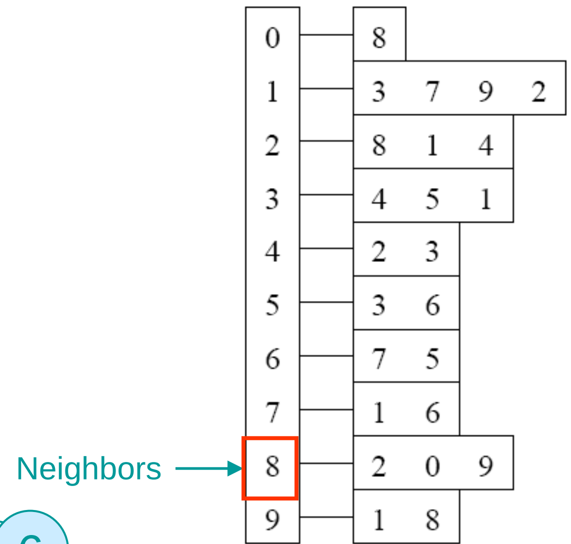
Place all previously unmarked neighbors of 2 on the queue.

Example

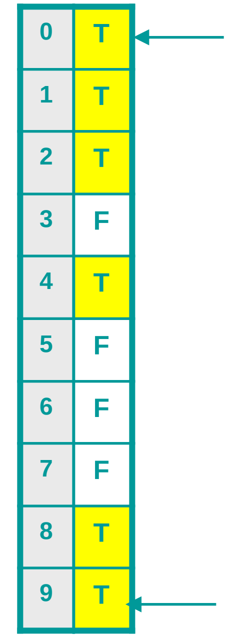


visit sequence ={2, 8}

Adjacency List



Flag Table (T/F)

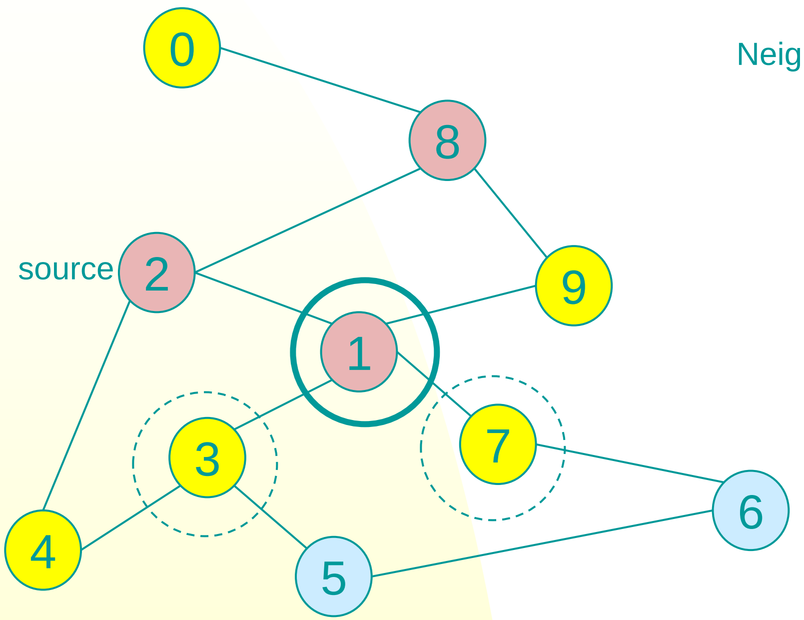


Mark unmarked neighbors.

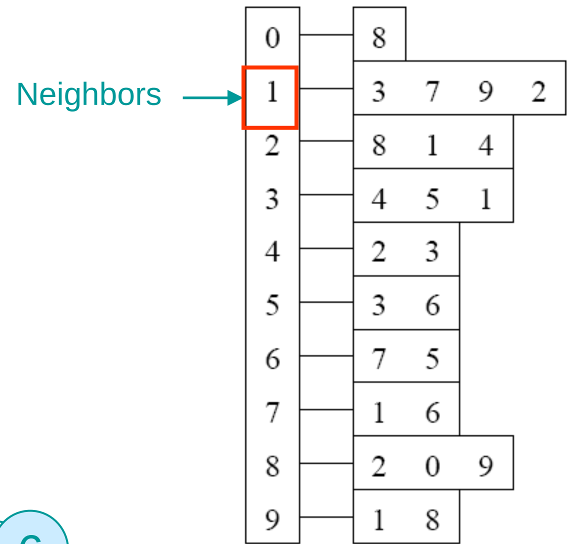
Q={ **8**, 1, 4 } → { 1, 4, **0**, **9** } (observe that 0, and 9 are placed AFTER 1 and 4)

Dequeue 8.
-- Place all unmarked neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, **as it has been marked before**!

Example



Adjacency List



Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

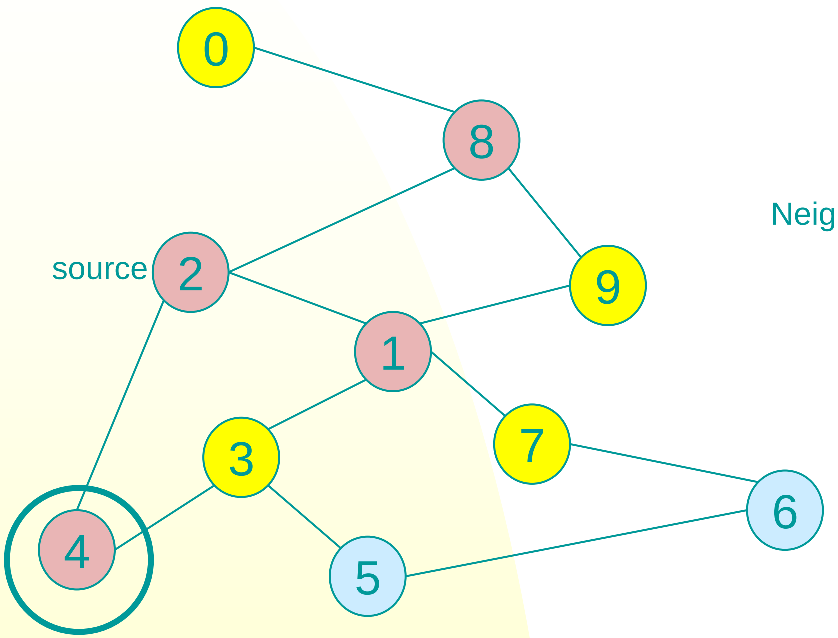
Mark unmarked neighbors.

visit sequence ={2, 8, 1}

Q={ **1**, 4, 0, 9 } → { 4, 0, 9, **3, 7** }

Dequeue 1.
-- Place all previously unmarked neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven't been marked previously.

Example



Adjacency List

Neighbors → 4

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

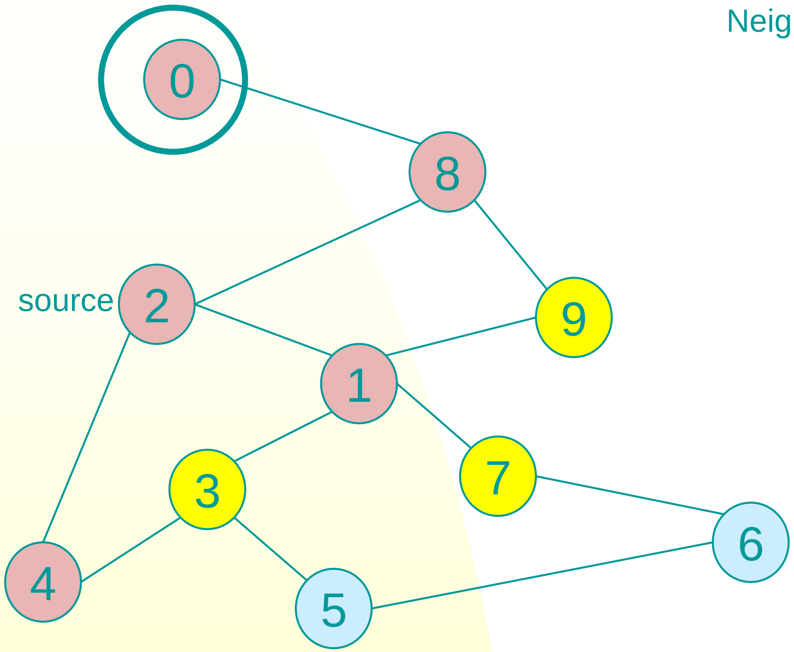
visit sequence ={2, 8, 1, 4}

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.

-- 4 has no unmarked neighbors!

Example



Adjacency List

Neighbors

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

visit sequence ={2, 8, 1, 4, 0}

Q = { **0**, 9, 3, 7 } → { 9, 3, 7 }

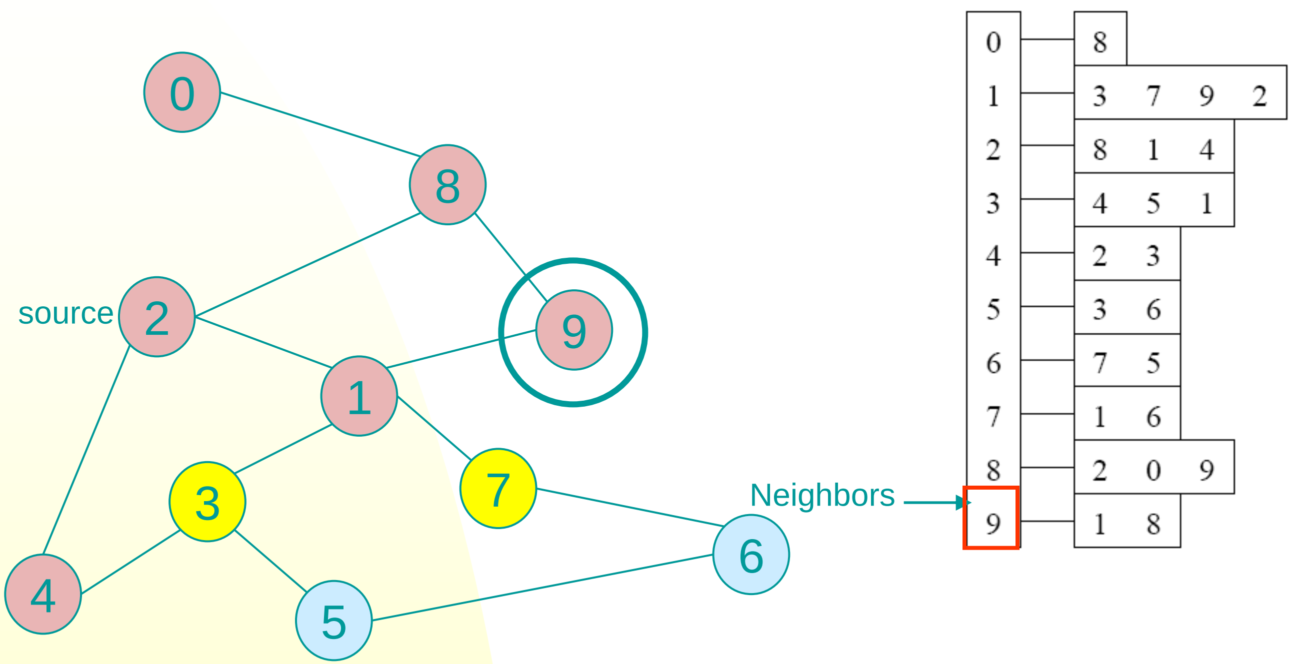
Dequeue 0.

-- 0 has no unmarked neighbors!

Example

Adjacency List

Flag Table (T/F)



0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

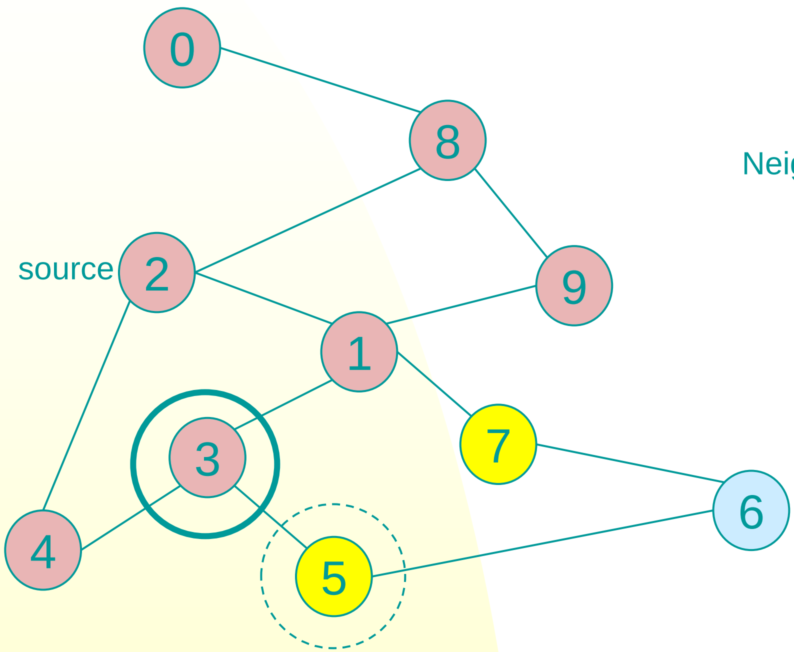
visit sequence ={2, 8, 1, 4, 0, 9}

Q={ **9**, 3, 7 } → { 3, 7 }

Dequeue 9.

-- 9 has no unmarked neighbors!

Example



Adjacency List

Neighbors

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

Mark unmarked neighbor.

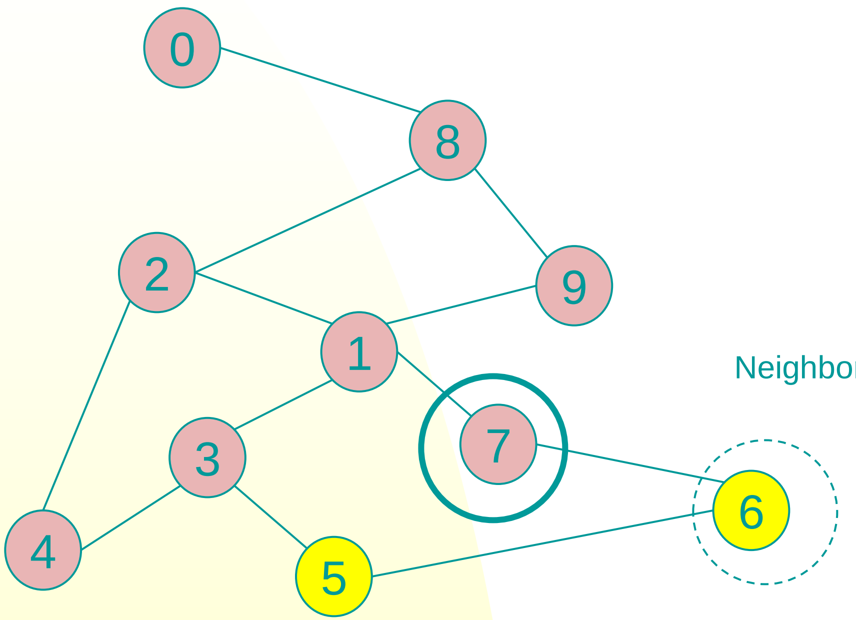
visit sequence ={2, 8, 1, 4, 0, 9, 3}

Q={ **3**, 7 } → { 7, **5** }

Dequeue 3.

-- place neighbor 5 on the queue.

Example



Neighbors →

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Mark unmarked neighbor.

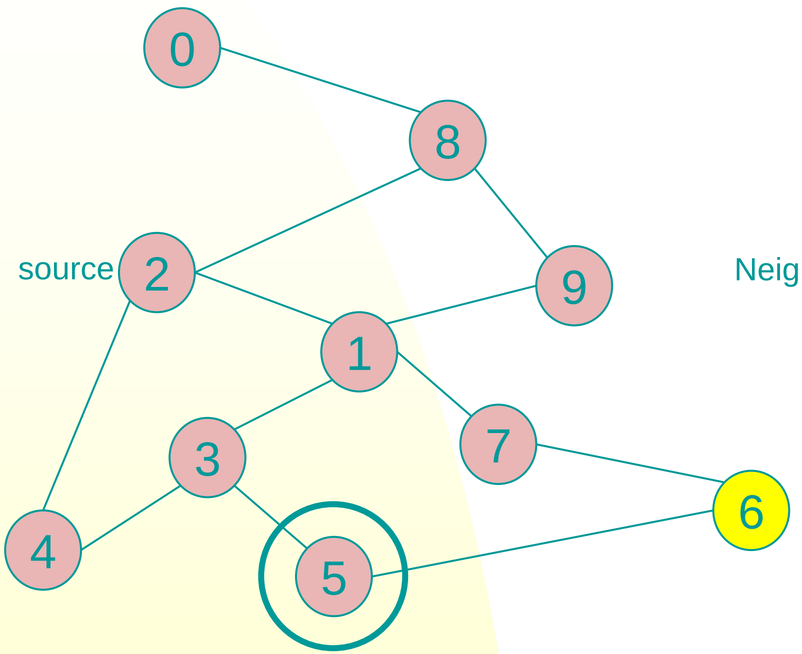
visit sequence ={2, 8, 1, 4, 0, 9, 3, 7}

Q = { **7**, 5 } → { 5, **6** }

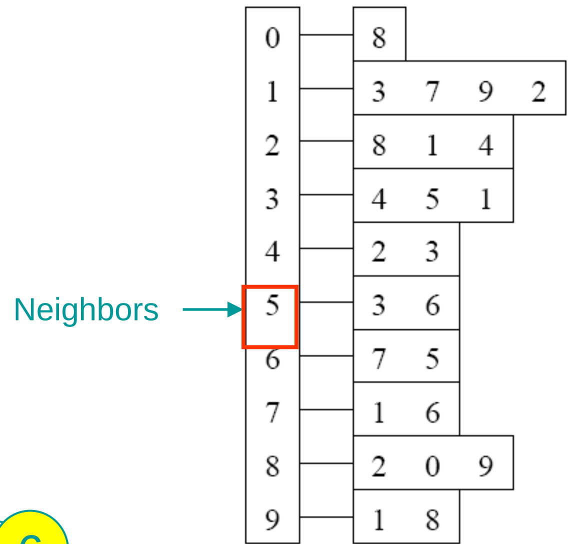
Dequeue 7.

-- place neighbor 6 on the queue.

Example



Adjacency List



Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

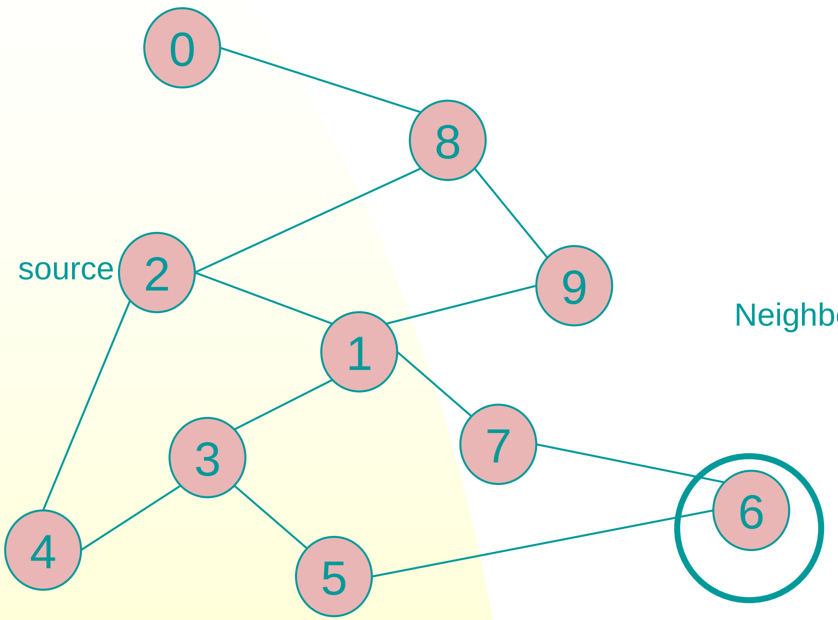
visit sequence ={2, 8, 1, 4, 0, 9, 3, 7, 5}

Q = { **5**, 6} → { 6 }

Dequeue 5.

-- no unmarked neighbors of 5.

Example



Adjacency List

Neighbors

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

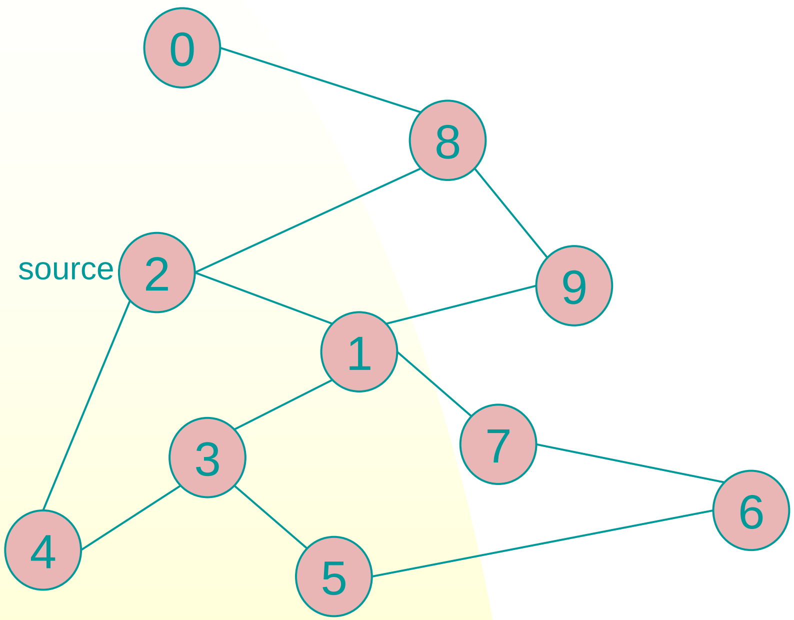
visit sequence ={2, 8, 1, 4, 0, 9, 3, 7, 5, 6}

Q={ **6** } → { }

Dequeue 6.

-- no unmarked neighbors of 6.

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

visit sequence = {2, 8, 1, 4, 0, 9, 3, 7, 5, 6}

There exist a path from source vertex 2 to all vertices in the graph!

Q = { } Q is empty, exit the while loop.

Remarks

- The unmarked neighbors enter to ***Q*** in the same order as in appear in the adjacent list.
  - If follows that if the order in the adjacent list is different, the output visit sequence will also be different.

- Starting at source *s*, BFS visits all the other (connected) vertices at increasing distance from s.

Running Time

Assume the graph is represented by an **adjacency list**. Let n and m represent the number of vertices and edges respectively.

```
for each vertex v
      do flag[v] := false;
Q = empty queue;
flag[s] := true;
enqueue(Q, s);
while Q is not empty
      do v := dequeue(Q);
           for each w adjacent to v
                do if flag[w] = false
                     then flag[w] := true;
                          enqueue(Q, w)
```

It loops O(n) times.

For a particular v, the for-loop loops exactly O(degree(v)) times (which is the size of that linked-list).

For a particular v, it loops at most O(degree(v)) times (which is the number of neighbors).

Running time

- Observe that whenever a vertex is marked for the first time, it is put inside *Q* in line 11. A marked vertex in *Q* will eventually be dequeued in line 7 and it will never be put inside Q again.
  - a vertex can only be dequeued (enqueued) one time

- Whenever a vertex *v* is dequeued,
  - we first find out all neighbors of *v*. For adjacency list representation, it needs to access the whole linked-list which has size O(deg(v)).
  - It follows the total time needed for all vertex is :

$$\sum_{\text{vertex } v} O(\deg(v)) = O(2m) = O(m)$$

Running time

- Moreover,
  - the neighbors (w) may be enqueued. For one vertex v, then it may has O(deg(v)) operations. However, since every vertex is enqueued (dequeued) exactly once, it follows the total number of enqueued (dequeue) operations is

$$O(n)$$

- Hence the running time for BFS for adjacency list representation is

$$\underbrace{O(n)}_{\text{initialization}} + \underbrace{O(n)}_{\text{enqueued/dequeued operations}} + \underbrace{O(2m)}_{\text{find out all the neighbors}} = O(n+m)$$

Running time

If the graph is represented by an adjacent matrix, the analysis is the same, except:

- To find out all neighbors of *v*, for adjacency matrix representation, it needs to access a row in the matrix, which has size O(n).

  It follows the total time needed for all vertex is :

$$\sum_{\text{vertex } v} O(n) = O(n^2)$$

- Hence the total running time for BFS is

$$\underbrace{O(n)}_{\text{initialization}} + \underbrace{O(n)}_{\text{enqueued/dequeued operation}} + \underbrace{O(n^2)}_{\text{find out all the neighbors}} = O(n^2)$$

Path recording

- BFS only tells us if a path exists from source s to other vertices v.
  - It doesn’t tell us the path!
  - We need to modify the algorithm to record the (shortest) path from s to v.

- The trick is to keep one additional piece of information with each vertex.

Path recording

- Let pred[0..n-1] be an array indexed by the vertices. The entry pred[w] contains the vertex v from where w is discovered, i.e., w was put inside the Q in line 11 because w is discovered by v.

```
6.  while Q is not empty
7.      do v := dequeue(Q);
8.          for each w adjacent to v
9.              do if flag[w] = false
10.                 then flag[w] := true;
11.                     enqueue(Q, w)
```

w is '***discovered***' by **v**, hence the path from **s** to **w** must pass through **v**, i.e.,

s $\rightarrow \dots \rightarrow$ **v** $\rightarrow$ **w**

BFS and Path recording

Algorithm *BFS*(s)

for each vertex v
  do $\mathit{flag}(v) :=$ false;
    $\boxed{\mathit{pred}[v] := -1;}$

initialization

$Q =$ empty queue;
$\mathit{flag}[s] :=$ true;
enqueue(Q, s);
while Q is not empty
  do $v :=$ *dequeue*(Q);
    for each w adjacent to v
      do if $\mathit{flag}[w] =$ false
        then $\mathit{flag}[w] :=$ true;
          $\boxed{\mathit{pred}[w] := v;}$
          enqueue(Q, w)

prev[w] stores which vertex discovers w.

Path Reporting

- After running the modified BFS, if flag[w] = true (it means there exists a path from s to w), one can call Path(w) to output the vertices on the path from s to w **in this order**.

Algorithm *Path*($\boxed{w}$)

1. **if** $pred[w] \neq -1$
2. **then**
3. $\boxed{Path(pred[w]);}$
4. $\boxed{\text{output } w}$

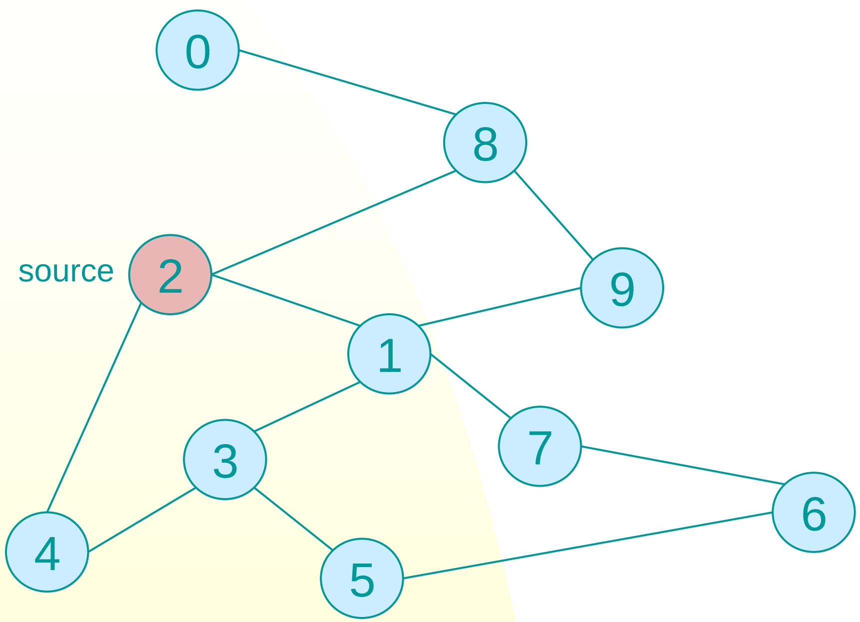
Notice the recursive structure which outputs a shortest path from s to w (not from w to s).

Shortest Path Reporting

- The running time is proportional to the length of the path from s to w.

- The path returned is actually the shortest from s to w. That is, among all possible paths from s to w, it has the minimum number of edges.

Example



Q = { }

Initialize **Q** to be empty

Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

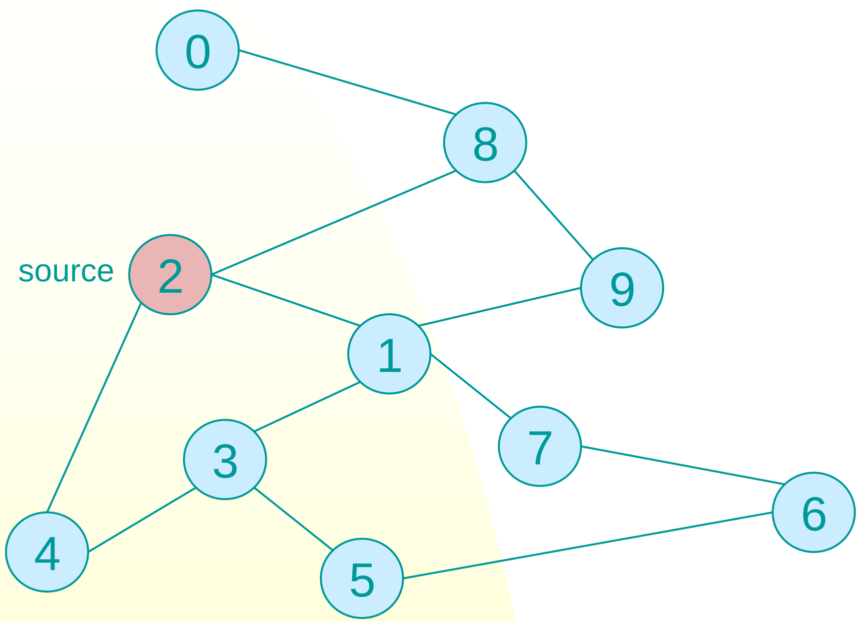
Flag Table (T/F)

		Pred
0	F	-1
1	F	-1
2	F	-1
3	F	-1
4	F	-1
5	F	-1
6	F	-1
7	F	-1
8	F	-1
9	F	-1

Initialize flag table (all F)

Initialize Pred to -1

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

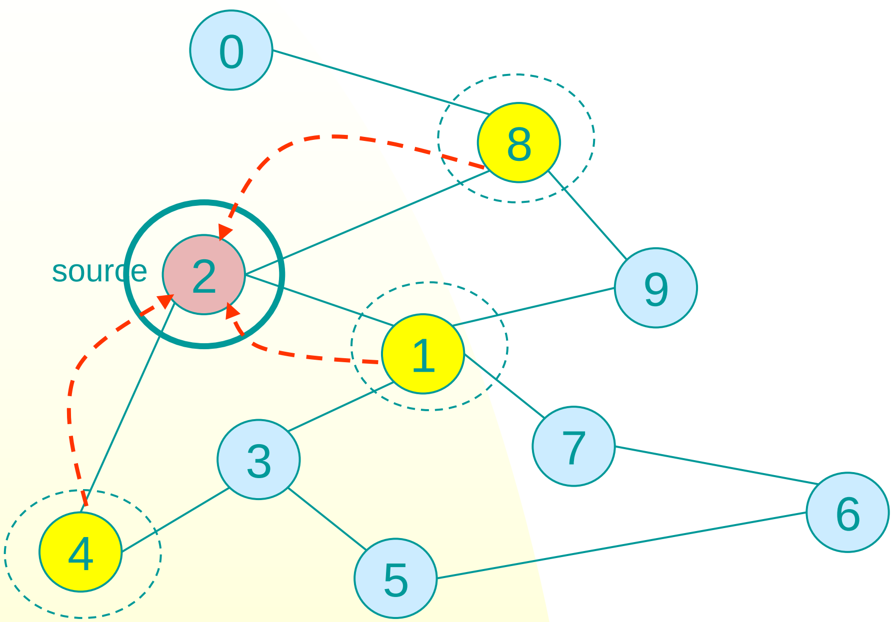
		Pred
0	F	-1
1	F	-1
2	T	-1
3	F	-1
4	F	-1
5	F	-1
6	F	-1
7	F	-1
8	F	-1
9	F	-1

Flag that 2 has been marked.

Q = { 2 }

Place source 2 on the queue.

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

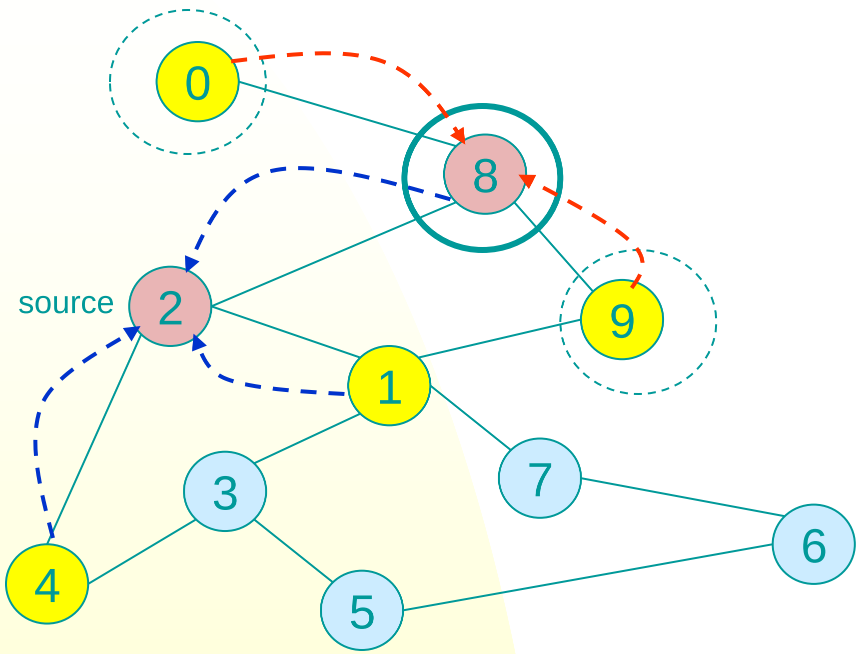
	Flag	*Pred*
0	F	-1
1	T	2
2	T	-1
3	F	-1
4	T	2
5	F	-1
6	F	-1
7	F	-1
8	T	2
9	F	-1

Record in Pred who was marked (discovered) by 2.

Q = {2} → { 8, 1, 4 }

Dequeue 2.
Place all unmarked neighbors of 2 on the queue

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	F	-1
4	T	2
5	F	-1
6	F	-1
7	F	-1
8	T	2
9	T	8

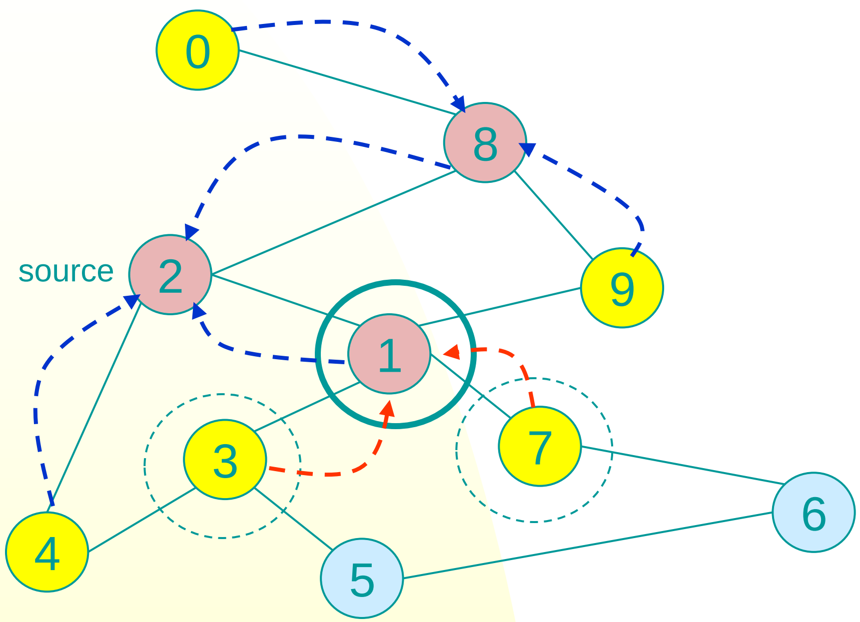
Mark unmarked Neighbors.

Record in Pred who was marked by 8.

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Dequeue 8.
-- Place all unmarked neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	F	-1
6	F	-1
7	T	1
8	T	2
9	T	8

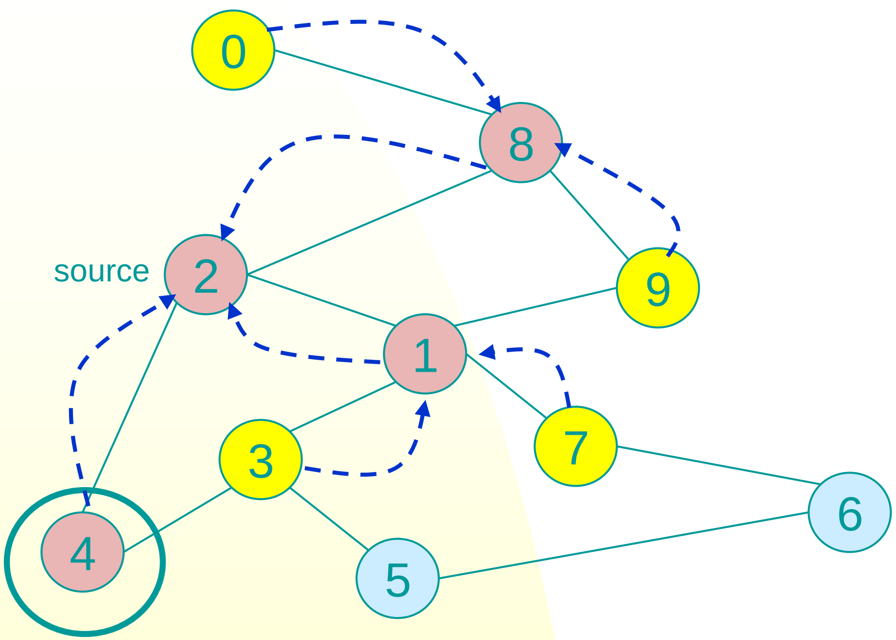
Mark unmarked Neighbors.

Record in Pred who was marked by 1.

Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.
-- Place all unmarked neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven't been marked yet.

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

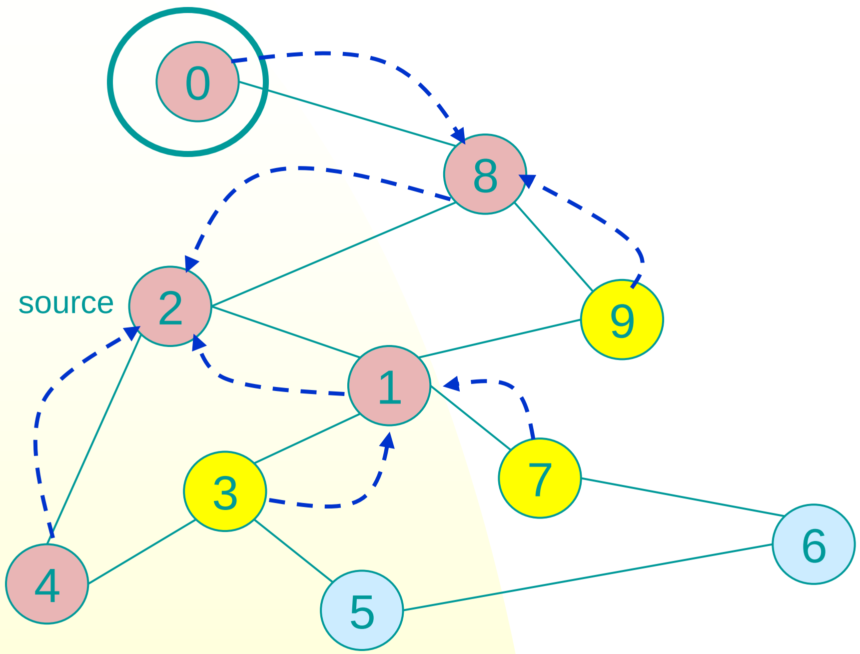
	Flag	*Pred*
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	F	-1
6	F	-1
7	T	1
8	T	2
9	T	8

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.

-- 4 has no unmarked neighbors!

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

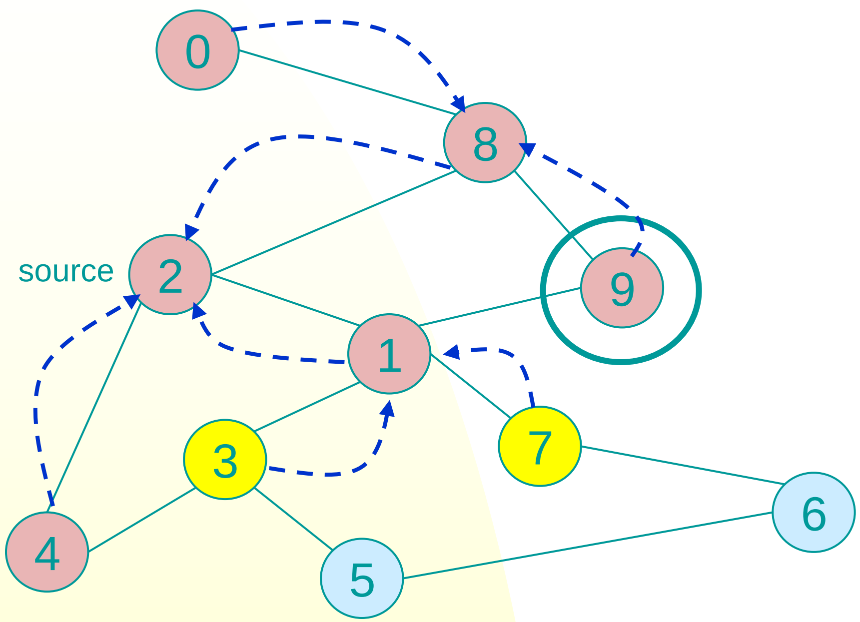
Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	F	-1
6	F	-1
7	T	1
8	T	2
9	T	8

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Dequeue 0.
-- 0 has no unmarked neighbors!

Example



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Flag Table (T/F)

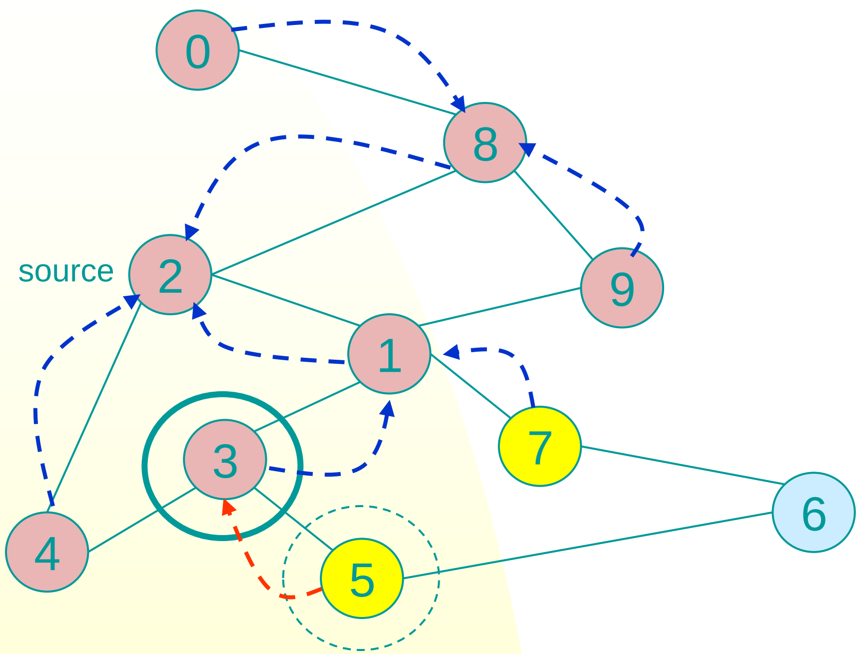
		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	F	-1
6	F	-1
7	T	1
8	T	2
9	T	8

Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.

-- 9 has no unmarked neighbors!

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	T	3
6	F	-
7	T	1
8	T	2
9	T	8

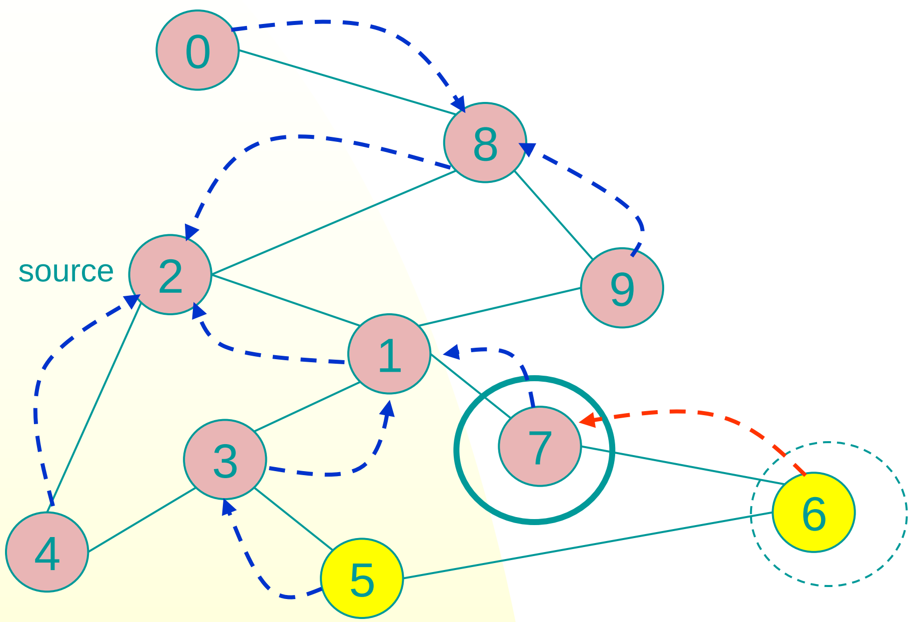
Mark unmarked Vertex 5.

Record in Pred who was marked by 3.

Q = { 3, 7 } → { 7, 5 }

Dequeue 3.
-- place neighbor 5 on the queue.

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Pred

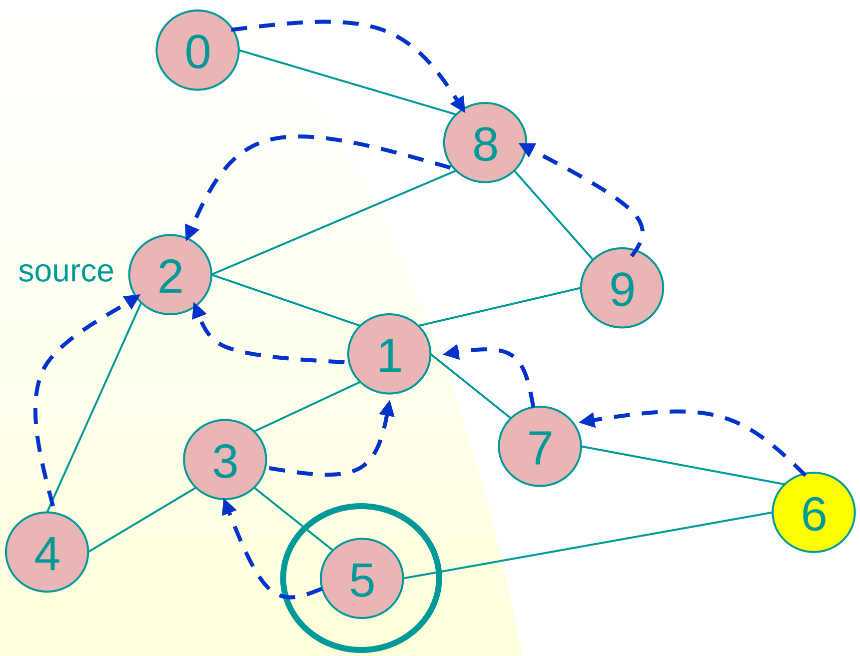
Mark unmarked Vertex 6.

Record in Pred who was marked by 7.

Q = { 7, 5 } → { 5, 6 }

Dequeue 7.
-- place neighbor 6 on the queue.

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

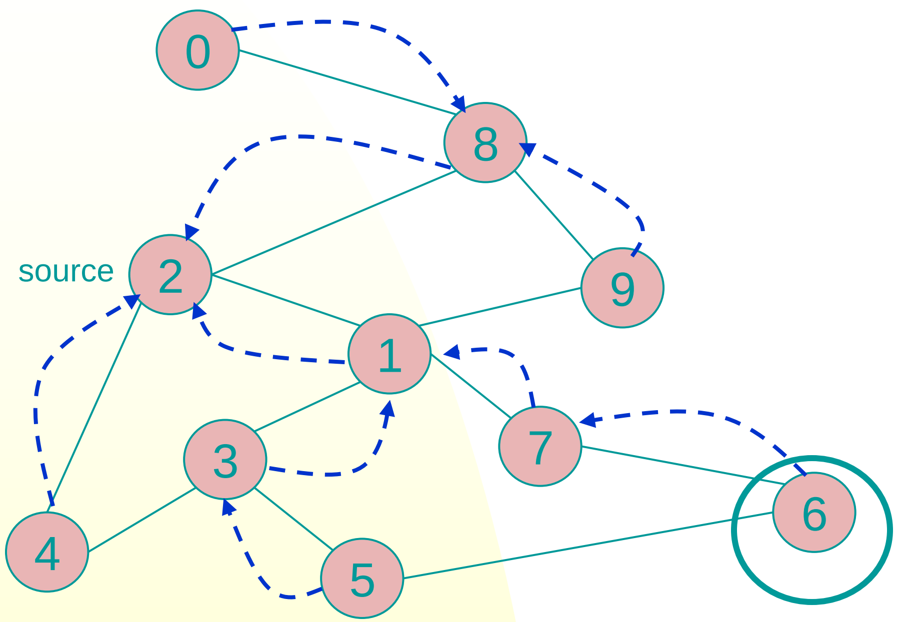
Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Q = { 5, 6} → { 6 }

Dequeue 5.
-- no unmarked neighbors of 5.

Example



Adjacency List

0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

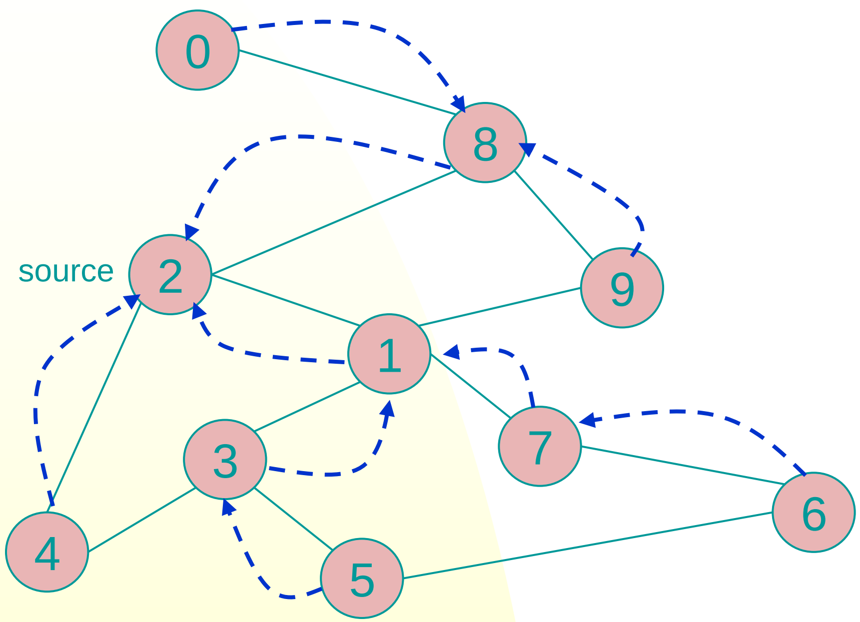
Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

Q = { 6 } → { }

Dequeue 6.
-- no unmarked neighbors of 6.

Example



Adjacency List

0	8
1	3 7 9 2
2	8 1 4
3	4 5 1
4	2 3
5	3 6
6	7 5
7	1 6
8	2 0 9
9	1 8

Flag Table (T/F)

		Pred
0	T	8
1	T	2
2	T	-1
3	T	1
4	T	2
5	T	3
6	T	7
7	T	1
8	T	2
9	T	8

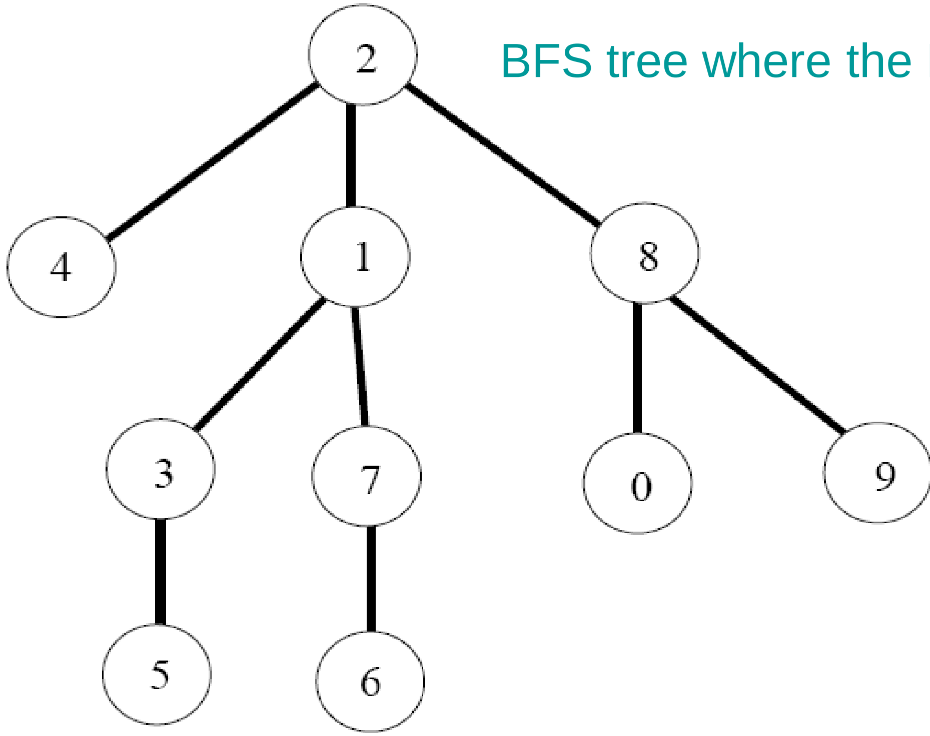
Pred

Pred now stores all the paths!

Q = { } STOP!!! Q is empty!!!

BFS tree

- We often draw the BFS paths as a tree, where ***s*** is the root.



BFS tree where the BFS start at vertex 2.

The root (s) to v path in the BFS tree represents the shortest from s to v in the original graph, and the level of v represents the length of such shortest path.