Depth First Search

cse2011

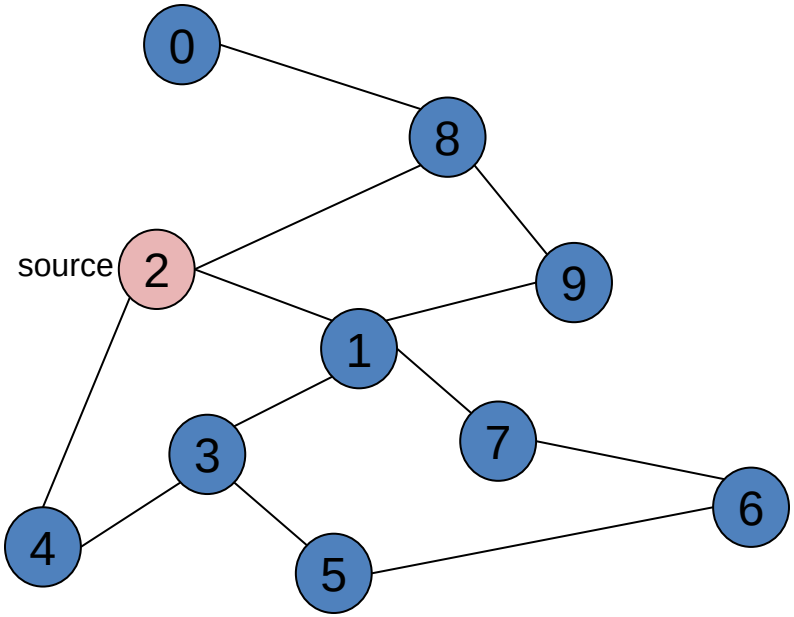
section 13.3 of textbook

Depth-First Search (DFS)

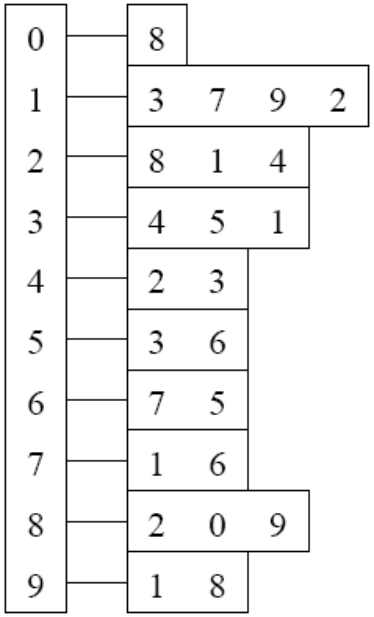
- DFS is another popular graph search strategy
  - Idea is similar to pre-order traversal (visit node, then visit children recursively)

- DFS will continue to visit neighbors in a recursive pattern
  - Whenever we visit v from u, we recursively visit all unvisited neighbors of v. Then we backtrack (return) to u.

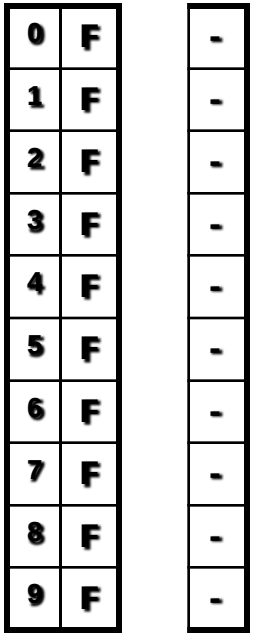
DFS Traversal Example



Adjacency List



Visited Table (T/F)



Pred

Initialize visited table (all False)

Initialize Pred to -1

DFS Algorithm Idea

- Similar to BFS algorithm, except that we use a stack instead of a queue for backtracking.

- In practice, we use recursion instead of an explicit stack.

BFS Algorithm

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

```
1.   for each vertex v
2.       do flag[v] := false;
3.   Q = empty queue;                    T = empty stack;
4.   flag[s] := true;
5.   enqueue(Q, s);                      T.push( s );
6.   while Q is not empty
7.      do v := dequeue(Q);              v = T.pop();
8.         for each w adjacent to v
9.            do if flag[w] = false
10.                 then flag[w] := true;
11.                      enqueue(Q, w)   T.push( w );
```

DFS Algorithm

Algorithm *DFS*(s)
1. **for** each vertex v
2. **do** $flag[v]$:= false;
3. *RDFS*(s);

Flag all vertices as not visited

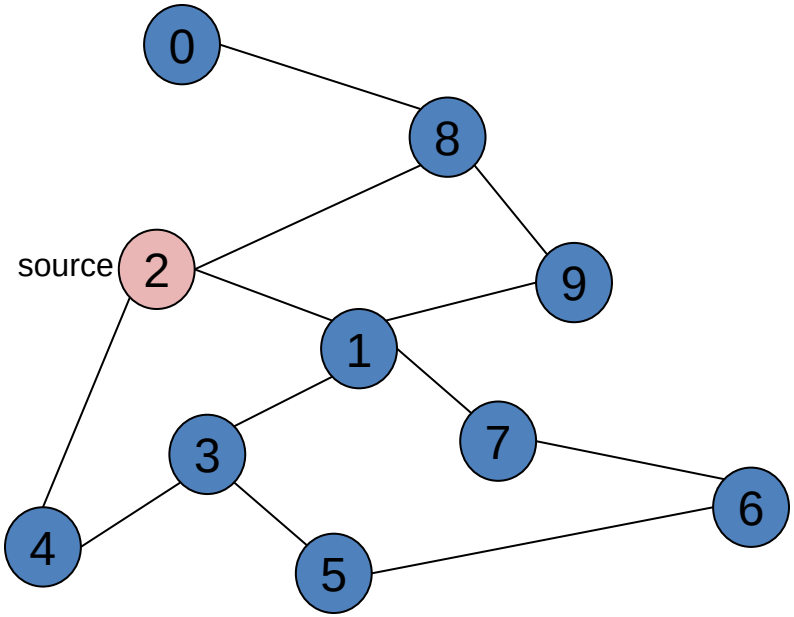
Algorithm *RDFS*(v)
1. $flag[v]$:= true; print v;
2. **for** each neighbor w of v
3. **do if** $flag[w]$ = false
4. **then** *RDFS*(w);

Flag v as visited; *print v*;

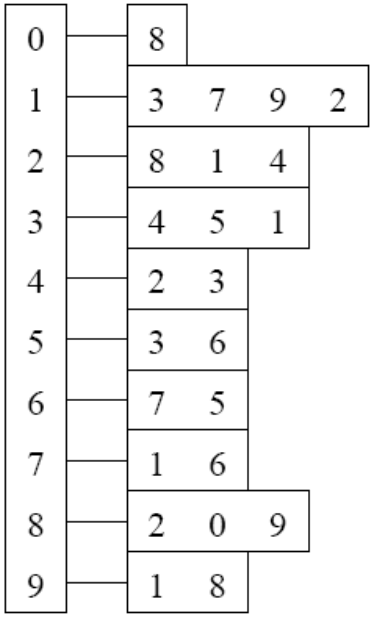
For unvisited neighbors, call *RDFS*(*w*) recursively

We can also record the paths using *prev*[ ].

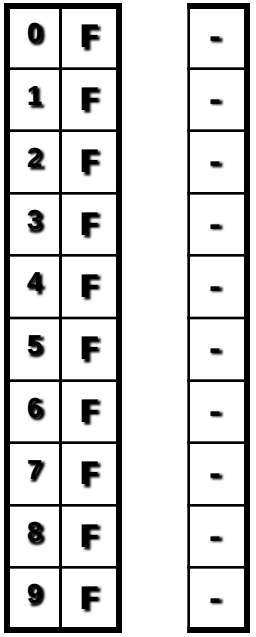
Example



Adjacency List



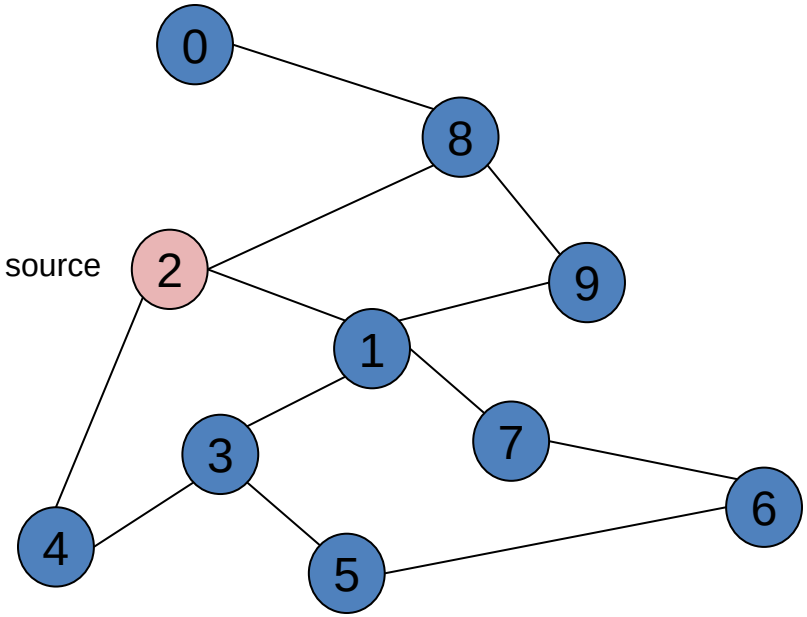
Visited Table (T/F)



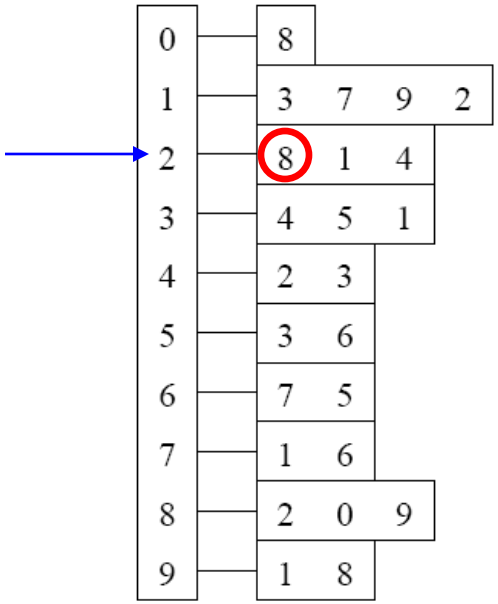
Pred

Initialize visited table (all False)

Initialize Pred to -1



Adjacency List



Visited Table (T/F)

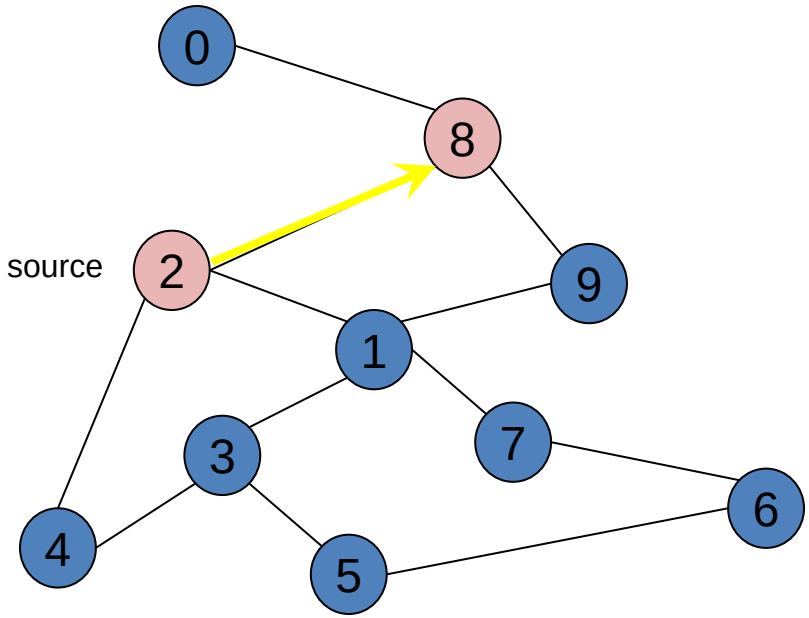
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Pred
-
-
-
-
-
-
-
-
-
-

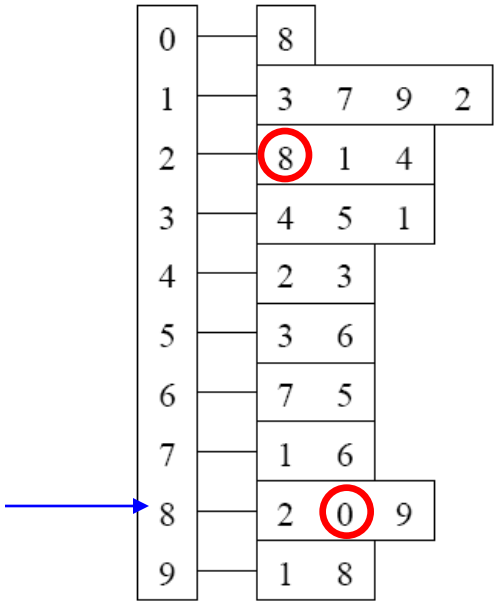
Mark 2 as visited

RDFS(2)

Now visit RDFS(8)



Adjacency List



Visited Table (T/F)

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

Pred

Mark 8 as visited

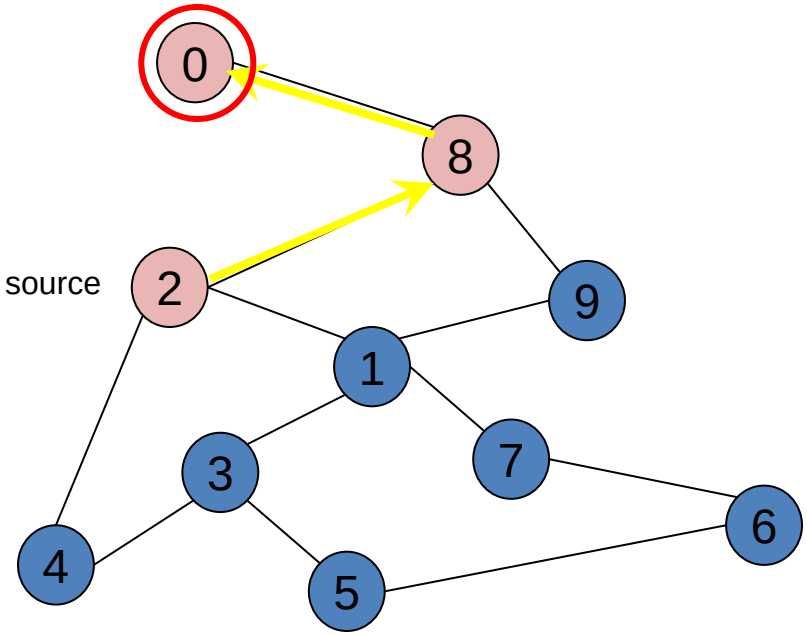
mark Pred[8]

Recursive calls

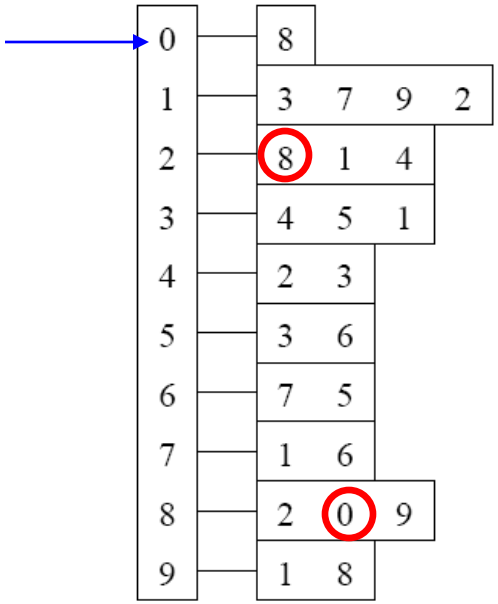
RDFS(2)

RDFS(8)

2 is already visited, so visit RDFS(0)



Adjacency List



Visited Table (T/F)

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

Pred
8
-
-
-
-
-
-
-
2
-

Mark 0 as visited

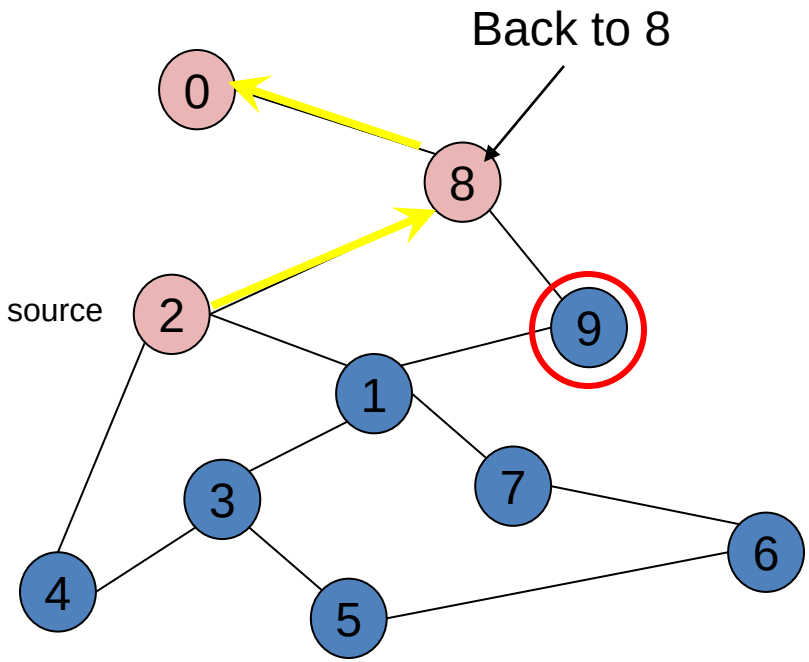
Mark Pred[0]

Recursive calls

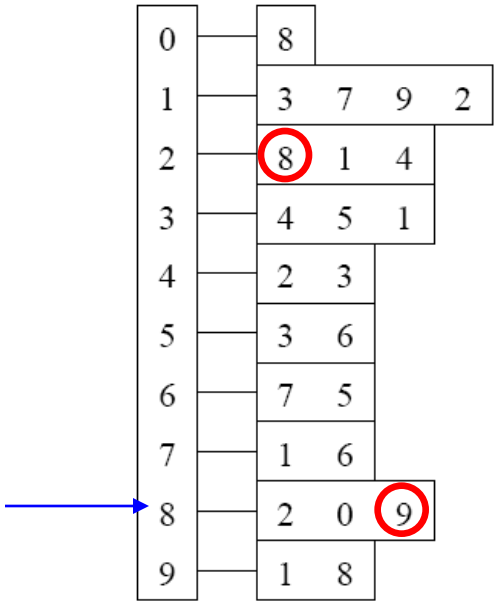
RDFS(2)

RDFS(8)

RDFS(0) -> no unvisited neighbors, return to call RDFS(8)



Adjacency List



Visited Table (T/F)

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

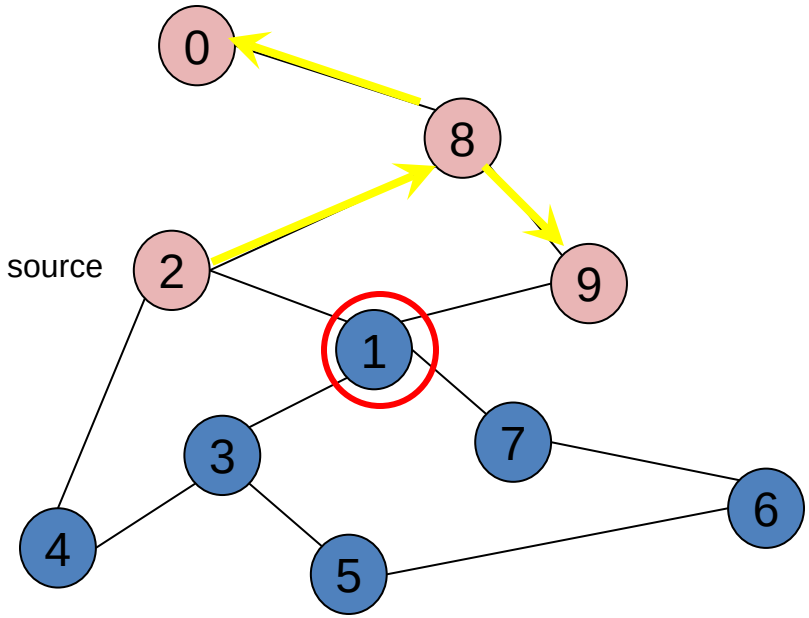
Pred
8
-
-
-
-
-
-
-
2
-

Recursive calls

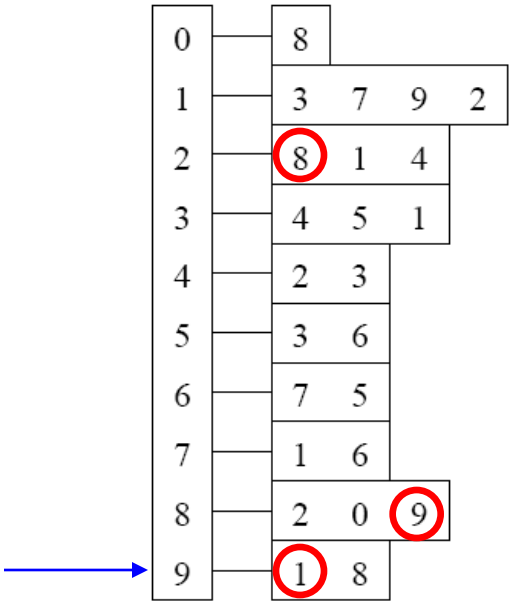
RDFS(2)

RDFS(8)

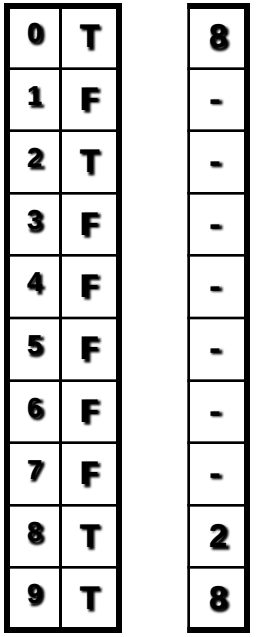
Now visit 9 -> RDFS(9)



Adjacency List



Visited Table (T/F)



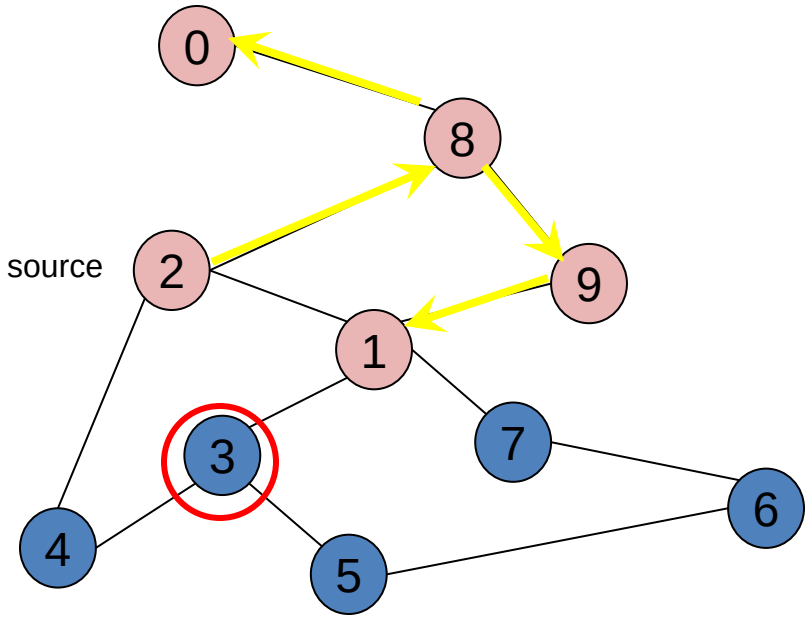
Pred

Mark 9 as visited

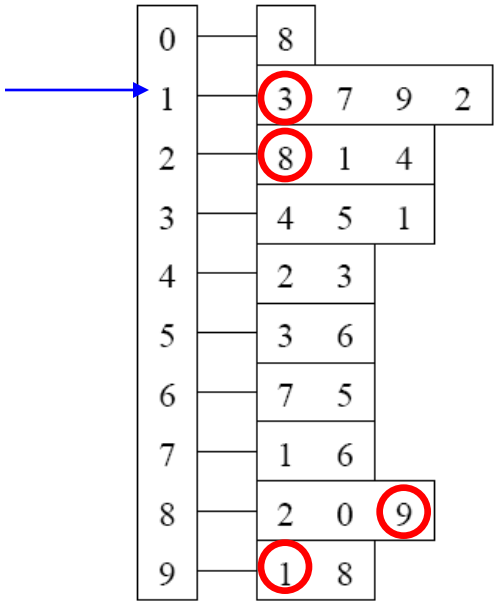
Mark Pred[9]

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
-> visit 1, RDFS(1)



Adjacency List



Visited Table (T/F)

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

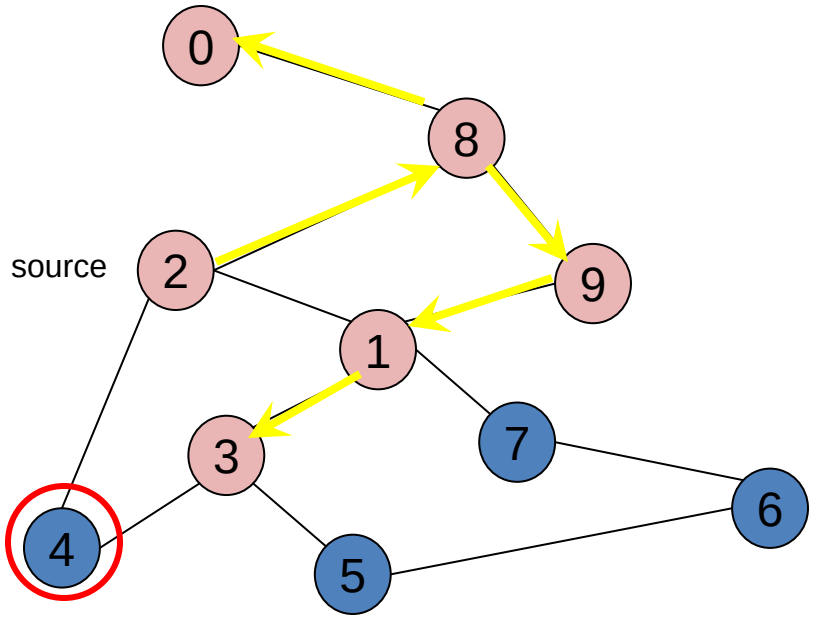
Pred
8
9
-
-
-
-
-
-
2
8

Mark 1 as visited

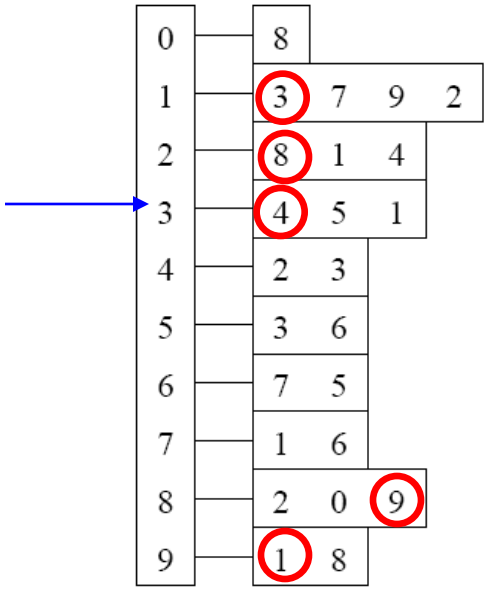
Mark Pred[1]

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
visit RDFS(3)



Adjacency List



Visited Table (T/F)

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

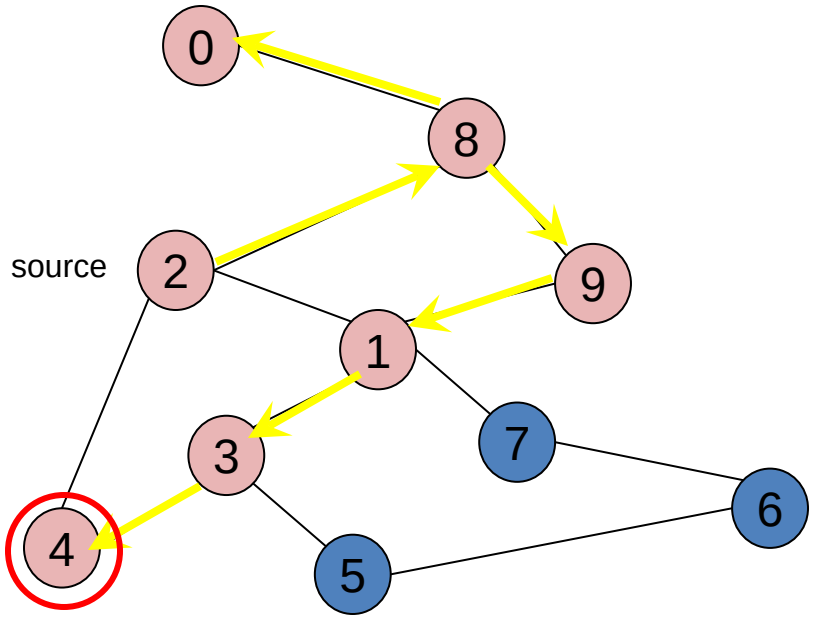
Pred
8
9
-
1
-
-
-
-
2
8

Mark 3 as visited

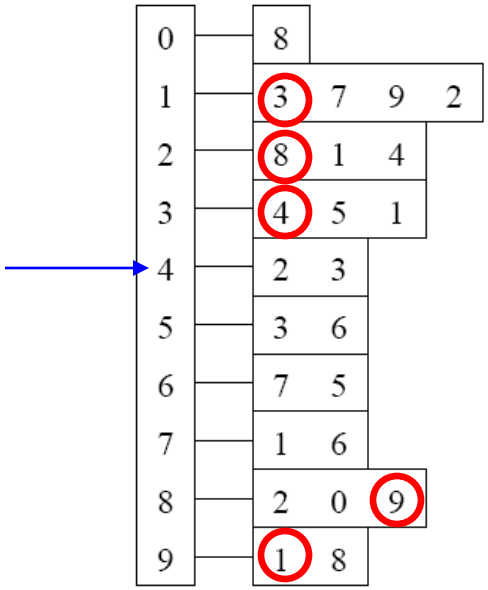
Mark Pred[3]

Recursive calls

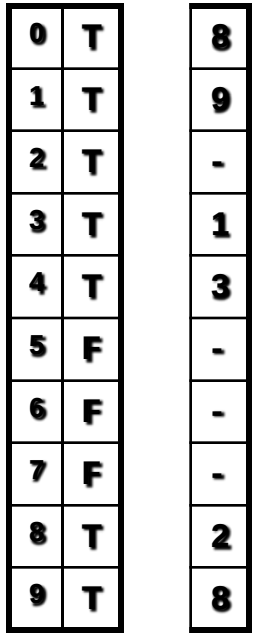
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit RDFS(4)



Adjacency List



Visited Table (T/F)



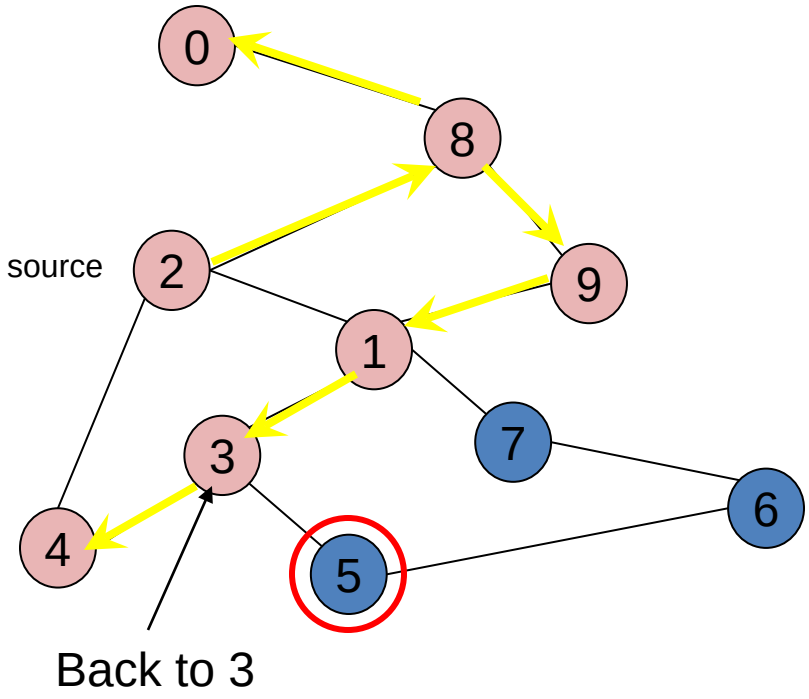
Pred

Recursive calls

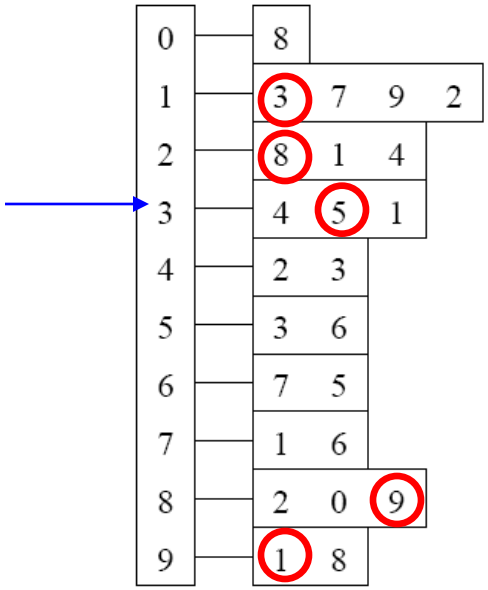
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(4) → STOP all of 4's neighbors have been visited
return back to call RDFS(3)

Mark 4 as visited

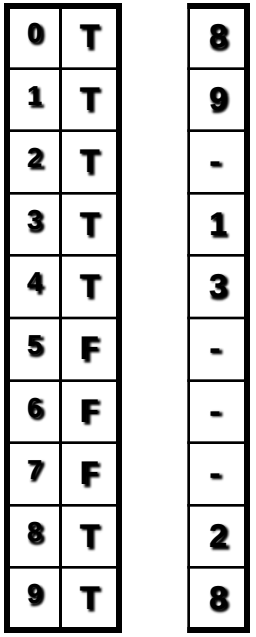
Mark Pred[4]



Adjacency List

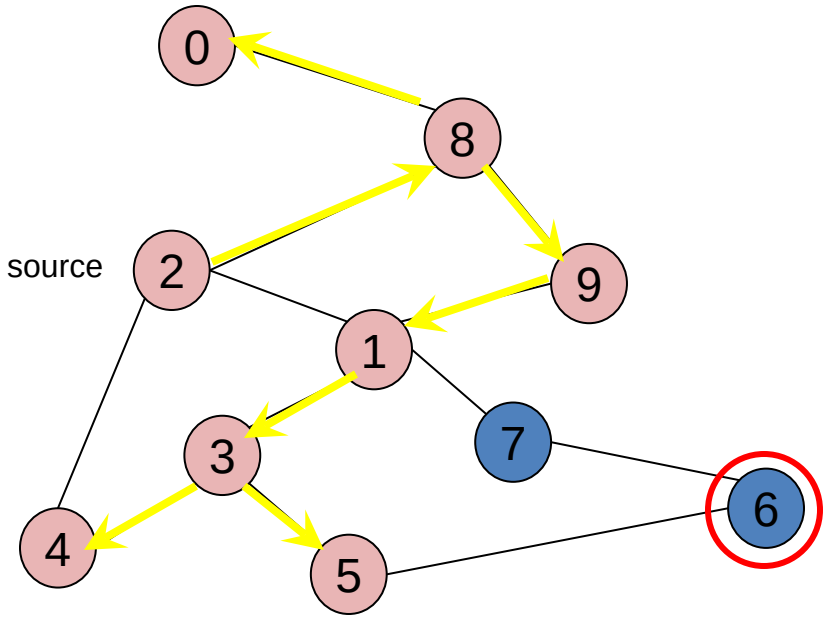


Visited Table (T/F)

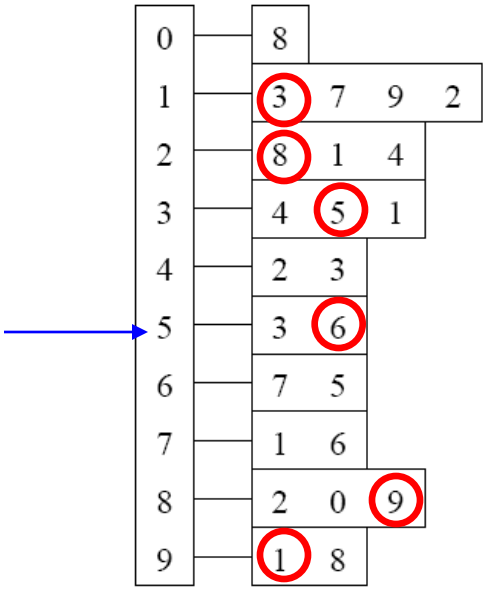


Recursive calls

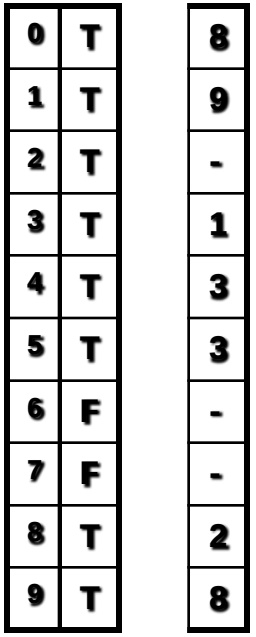
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit 5 -> RDFS(5)



Adjacency List



Visited Table (T/F)



Pred

RDFS(2)

RDFS(8)

RDFS(9)

RDFS(1)

RDFS(3)

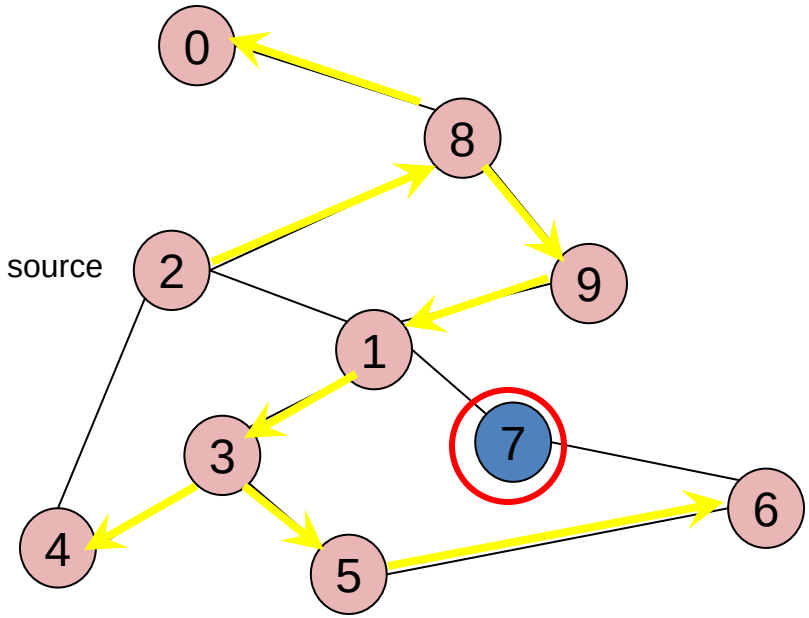
RDFS(5)

3 is already visited, so visit 6 -> RDFS(6)

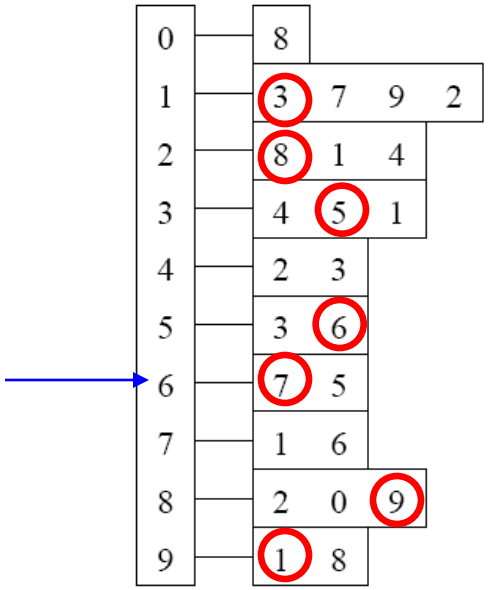
Recursive calls

Mark 5 as visited

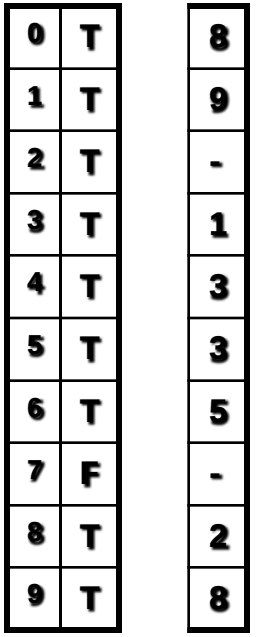
Mark Pred[5]



Adjacency List



Visited Table (T/F)



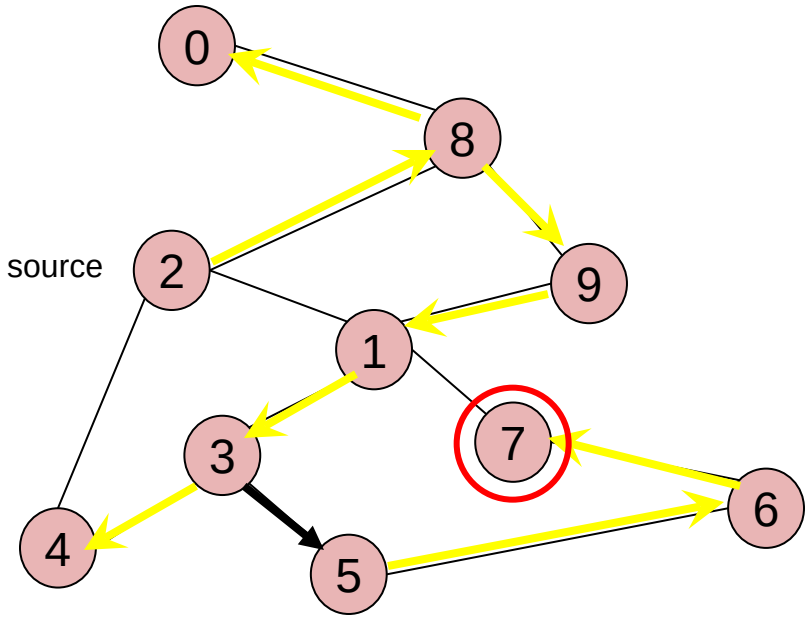
Pred

Recursive calls

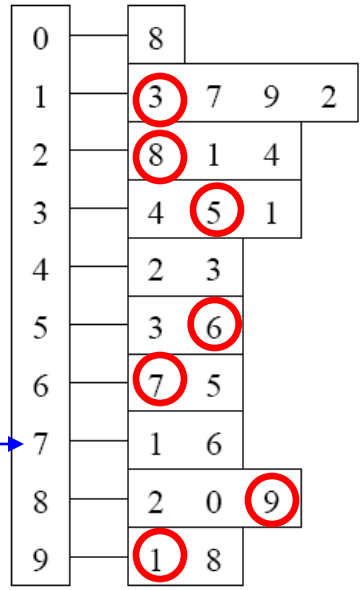
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
visit 7 -> RDFS(7)

Mark 6 as visited

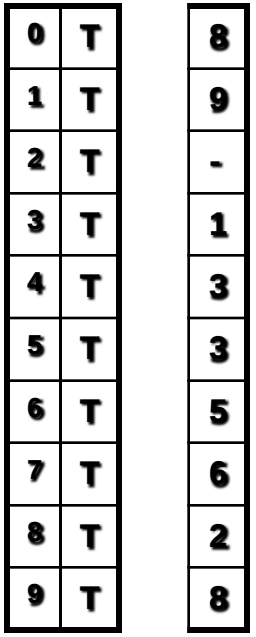
Mark Pred[6]



Adjacency List



Visited Table (T/F)



RDFS(2)

RDFS(8)

RDFS(9)

RDFS(1)

RDFS(3)

RDFS(5)

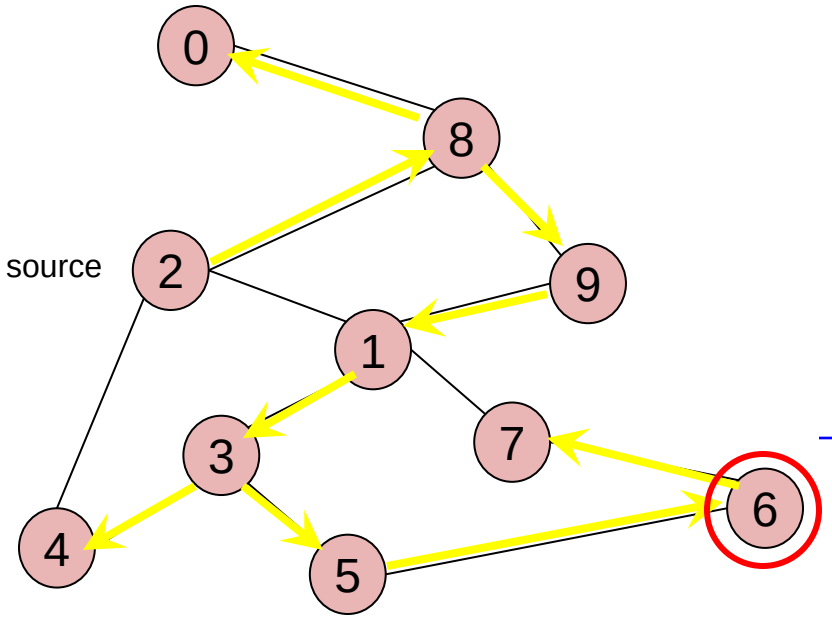
RDFS(6)

RDFS(7) -> Stop no more unvisited neighbors

Recursive calls

Mark 7 as visited

Mark Pred[7]



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

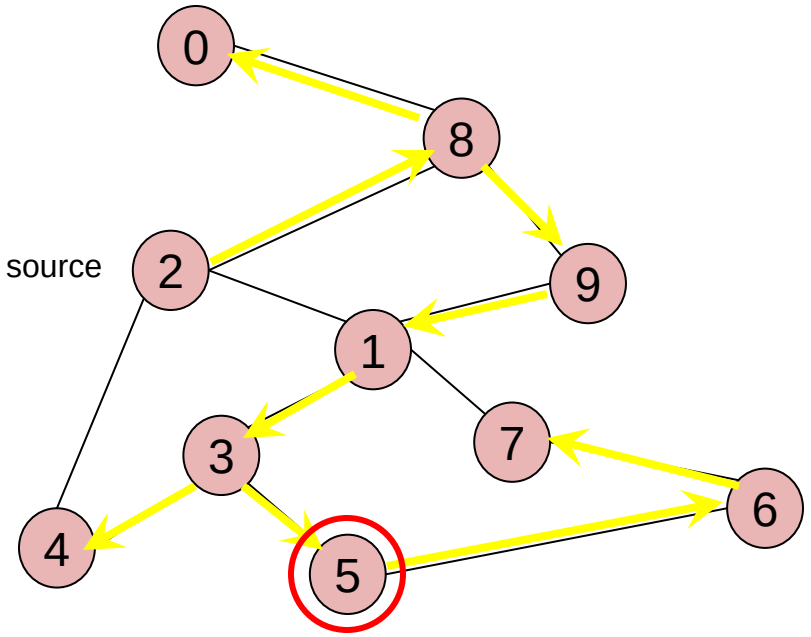
Visited Table (T/F)

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

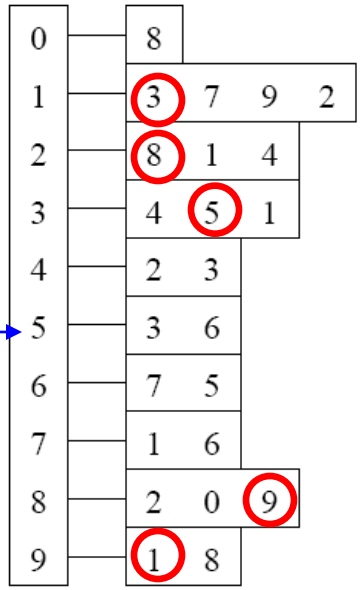
Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) -> Stop



Adjacency List



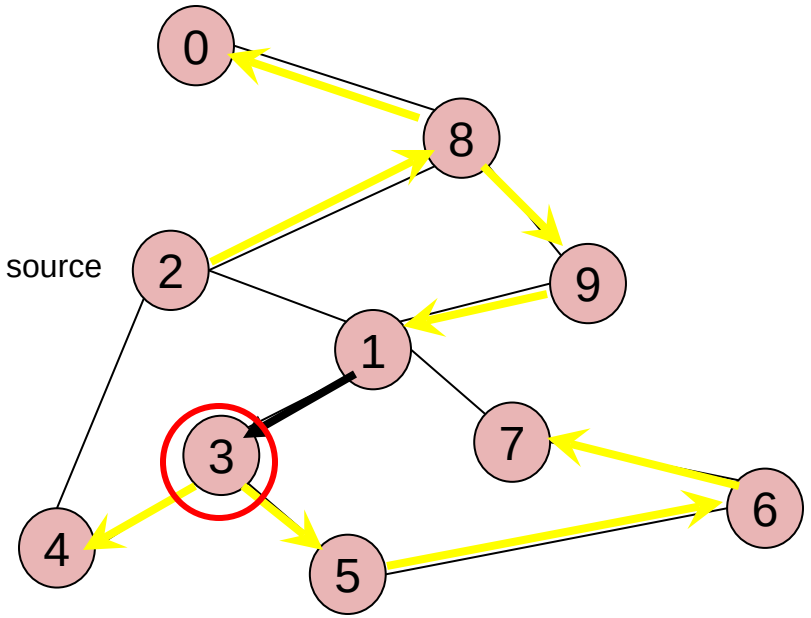
Visited Table (T/F)

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

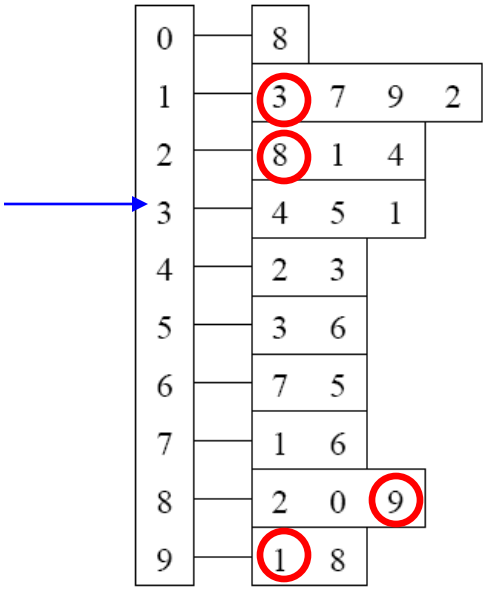
Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

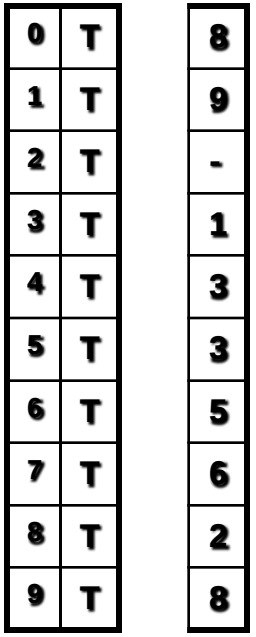
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5) -> Stop



Adjacency List



Visited Table (T/F)



Pred

RDFS(2)

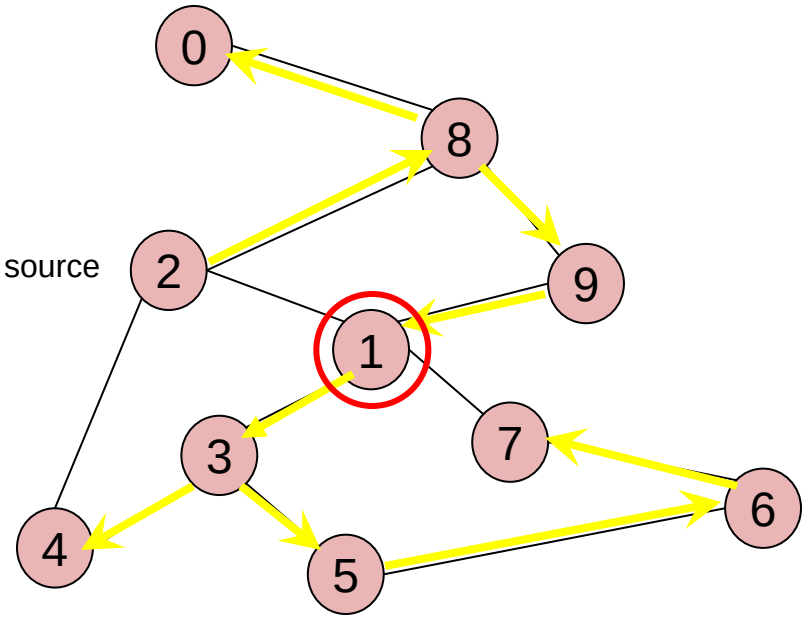
RDFS(8)

RDFS(9)

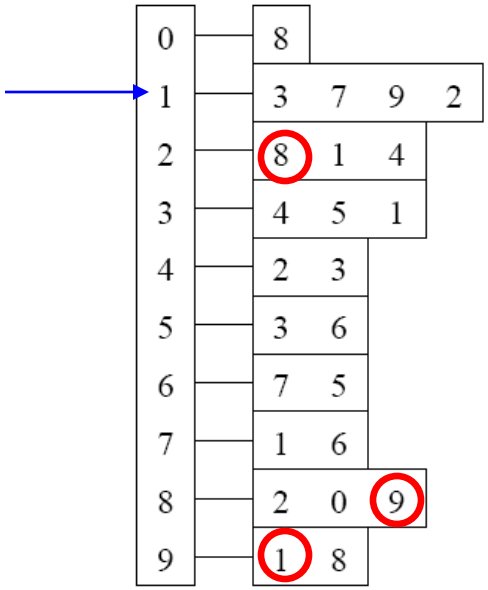
RDFS(1)

RDFS(3) -> Stop

Recursive calls



Adjacency List



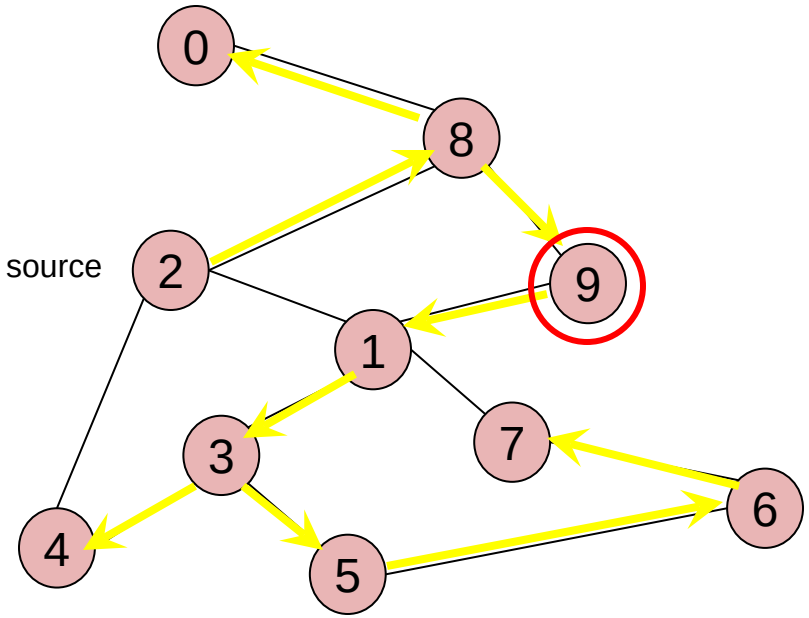
Visited Table (T/F)

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

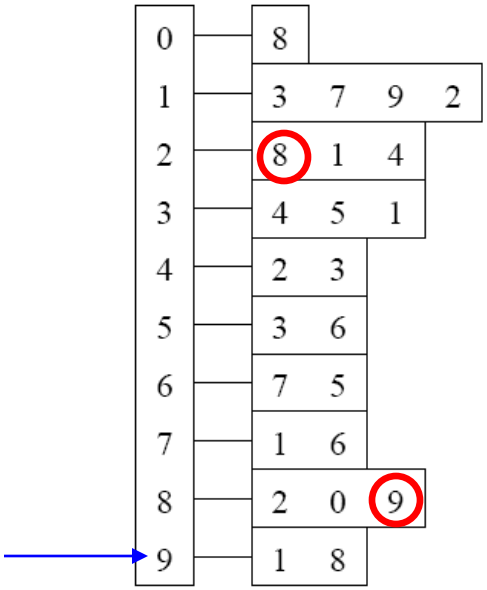
Pred
8
9
-
1
3
3
5
6
2
8

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1) -> Stop

Recursive calls



Adjacency List



Visited Table (T/F)

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

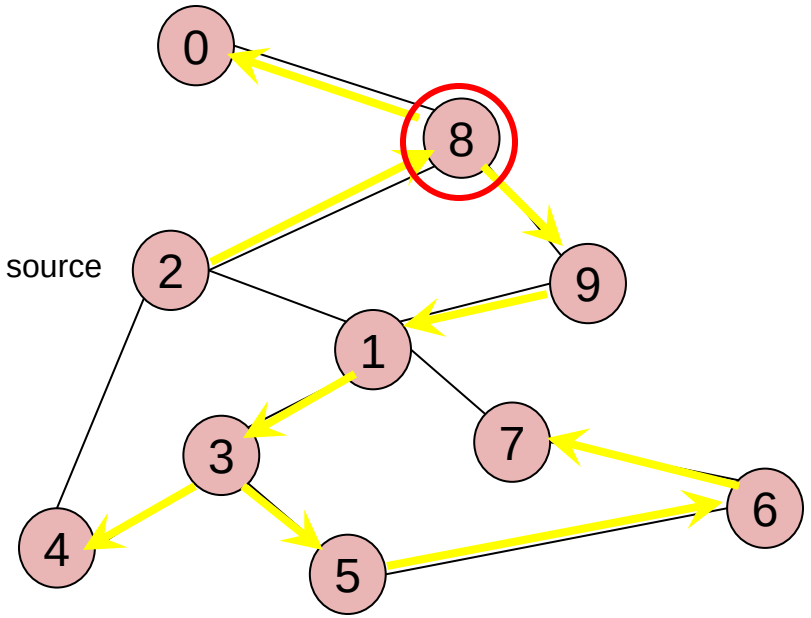
Pred
8
9
-
1
3
3
5
6
2
8

RDFS(2)

RDFS(8)

RDFS(9) -> Stop

Recursive calls



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

Visited Table (T/F)

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

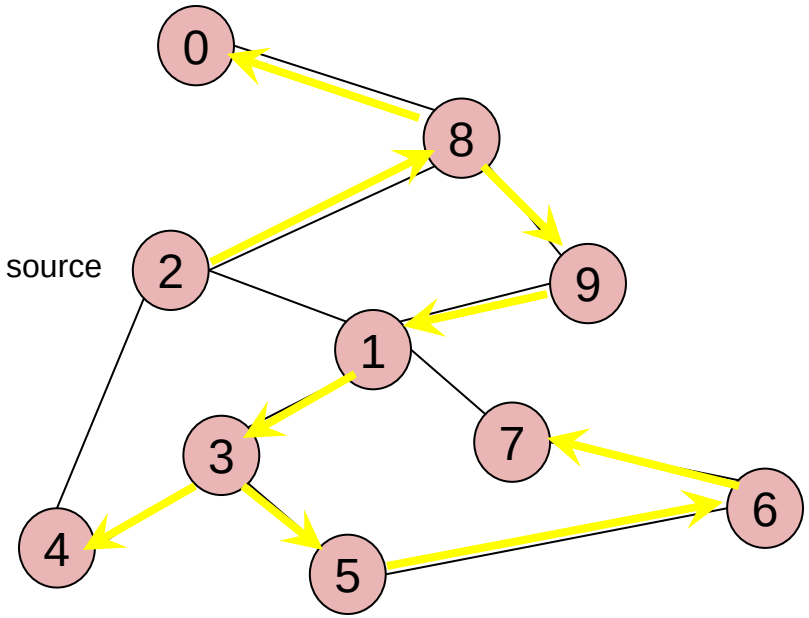
Pred

RDFS(2)

RDFS(8) -> Stop

Recursive calls

Example Finished



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

Visited Table (T/F)

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

Pred

RDFS(2) -> Stop

Recursive calls finished

Time Complexity of DFS

- We never visited a vertex more than once.
- We had to examine the adjacency lists of all vertices.
  - $\Sigma_{\text{vertex } v}$ degree(v) = 2E
- So, the running time of DFS is proportional to the number of edges and number of vertices (same as BFS)
  - O(V + E)

Enhanced DFS Algorithm

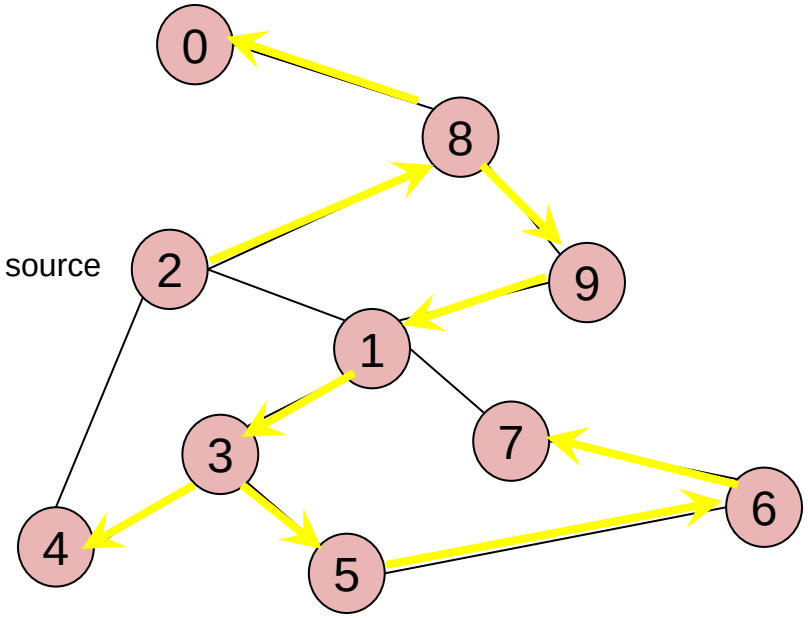
- What if a graph is not connected (strongly connected)?
  - Use an enhanced version of DFS, which is similar to the enhanced BFS algorithm.

```
DFSearch( G ) {
i = 1;     // component number
for every vertex v
    flag[v] = false;
for every vertex v
    if ( flag[v] == false ) {
      print ( "Component " +  i++ );
      RDFS( v );
    }
}
```

Applications of DFS

- Is there a path from source *s* to a vertex *v*?
- Is an undirected graph connected?
- Is a directed graph strongly connected?
- To output the contents (e.g., the vertices) of a graph
- To find the connected components of a graph
- To find out if a graph contains cycles and report cycles.
- To construct a DSF tree/forest from a graph

DFS Path Tracking



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

Visited Table (T/F)

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

Pred
8
9
-
1
3
3
5
6
2
8

DFS finds out path too.

Algorithm *Path*(w)

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

Try some examples.
Path(0) ->
Path(6) ->
Path(7) ->