AST20105 Data Structures & Algorithms

CHAPTER 8 – GRAPHS

Instructed by Garret Lai

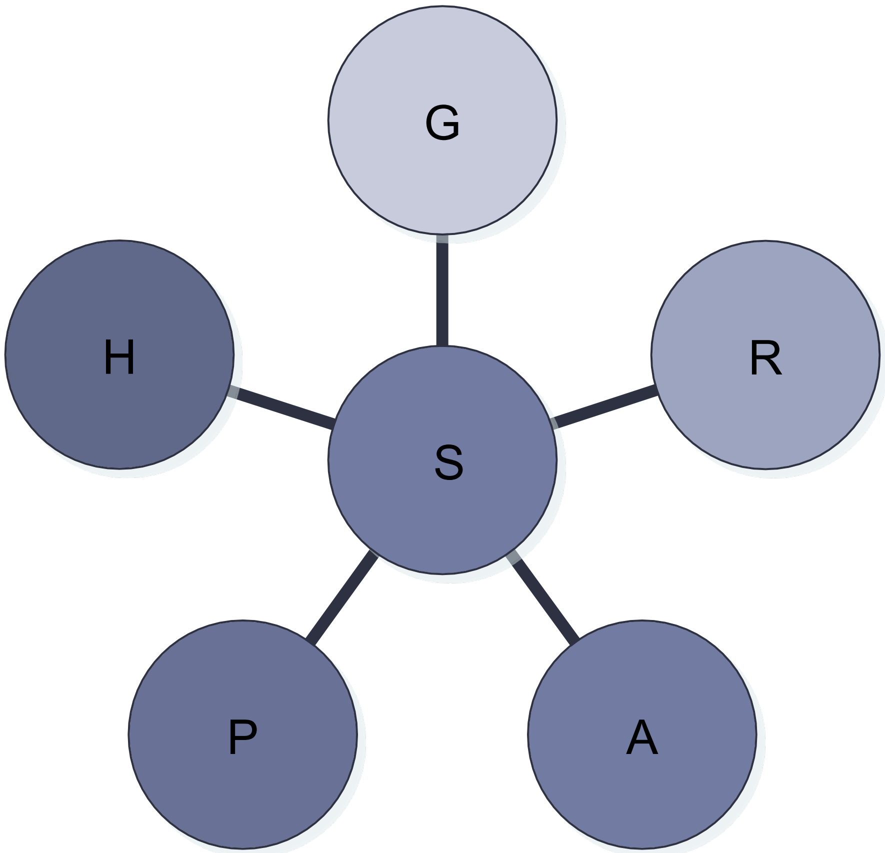
Before Start

- In spite of the flexibility of trees and the many different tree applications,
  - Trees, by their nature, have one limitation.
    - They can only represent relations of a hierarchical type,
      - such as relations between parent and child.
    - Other relations are only represented indirectly,
      - such as the relation of being a sibling.

Before Start

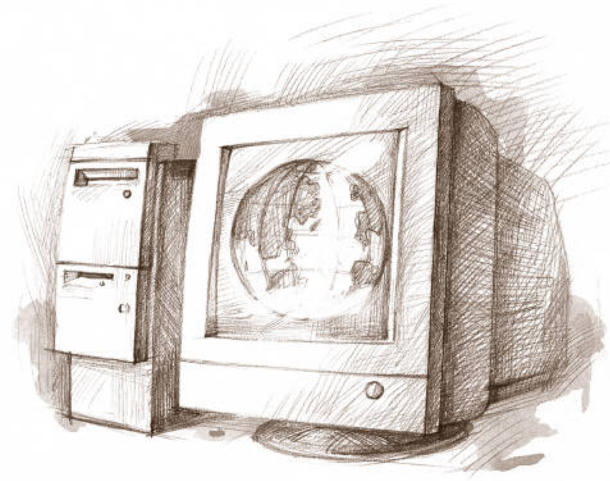
- A generalization of a tree, a graph, is a data structure in which this limitation is lifted.

Graphs



Graphs

- Graphs are widely-used structure in computer science and different computer applications.



Graphs

- Graphs mean to store and analyze *metadata*, the connections, which present in data.

- For instance, consider cities in your country.

  - Road network, which connects them, can be represented as a graph and then analyzed.

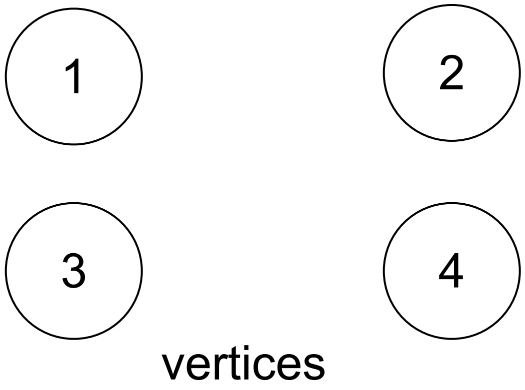
  - We can examine, if one city can be reached from another one or find the shortest route between two cities.

Introduction to graphs

Introduction

- There are two important sets of objects,
  - which specify graph and its structure.

- First set is **V**, which is called **vertex-set**.
  - In the example with road network, cities are vertices.
  - Each vertex can be drawn as a circle with vertex's number inside.

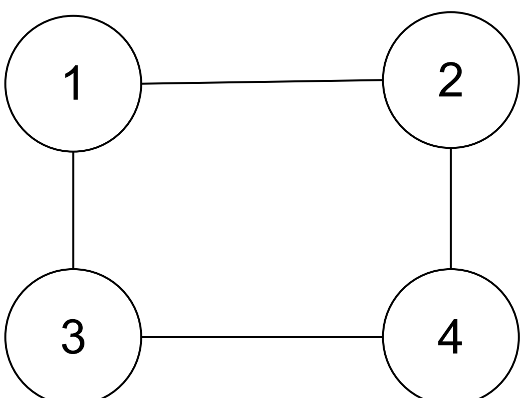


Introduction

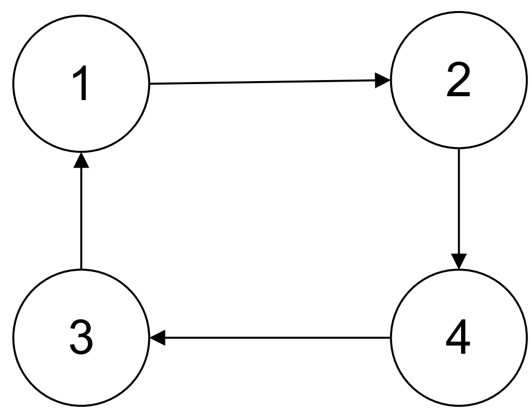
- Next important set is **E**, which is called **edge-set**.
  - **E** is a subset of **V x V**.
  - Simply speaking, each edge connects two vertices, including a case, when a vertex is connected to itself (such an edge is called *a loop*).

Introduction

- All graphs are divided into two big groups:
  - directed and
  - undirected graphs.



undirected graph



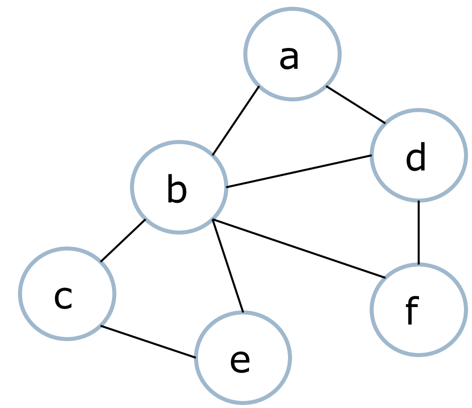
directed graph

Introduction

- The difference is that edges in directed graphs, called *arcs*, have a direction.
  - Edge can be drawn as a line.
  - If a graph is directed, each line has an arrow.

Terminology – Undirected Graph

- **Incident:**
  An edge (x,y) incidents upon vertices x and y
- **Adjacent:**
  a and b are adjacent if (x,y) is an edge in E
- **Degree of a node:**
  Number of distinct edges incident with it
- **Simple path:**
  No vertex appears twice in the path
- **Cycle:**
  A path in G contains at least 3 vertices, such that the last vertex in the sequence is adjacent to the first vertex in the sequence

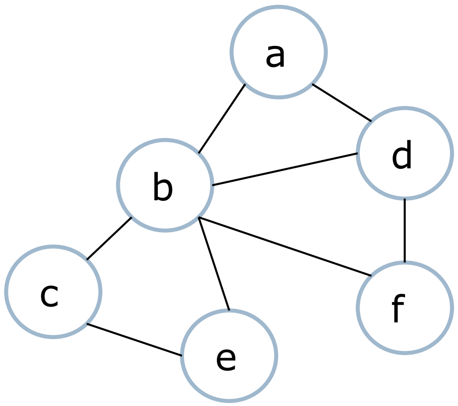


G = (V, E)
V = { a, b, c, d, e, f }
E = { (a,b), (a,d), (b,a), (b,d), (b,e), (b,f), (b,c), (c,b), (c,e), (d,a), (d,b), (d,f), (e,b), (e,c), (f,b), (f,d) }

- Incident: The edge (a,b) incidents vertices a and b
- Adjacent: a, b are adjacent, a, d are adjacent, etc.
- Degree of node b: 5
- Simple path: a, b, c
- Cycle: a, b, d, a

Terminology – Undirected Graph

- **Connected:**
  A graph G is connected any two vertices x and y in G has a path with first vertex x and last vertex y
- **Undirected complete graph:**
  An undirected graph G has an edge between every pair of vertices in G
- **Undirected cyclic graph:**
  An undirected graph with cycle
- **Undirected acyclic graph:**
  An undirected graph without cycle
- **Forest:**
  An acyclic graph whose connected components are trees

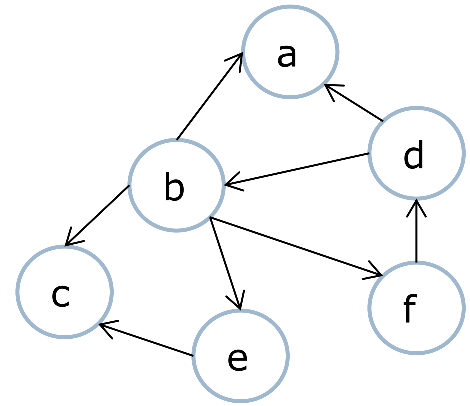


G = (V, E)
V = { a, b, c, d, e, f }
E = { (a,b), (a,d), (b,a), (b,d),
(b,e), (b,f), (b,c), (c,b),
(c,e), (d,a), (d,b), (d,f),
(e,b), (e,c), (f,b), (f,d) }

- The graph is connected
- The graph is NOT complete, but the subgraph formed by node a,b,d is complete
- The graph is a cyclic undirected graph, since there are cycle, e.g. c, b, e, c OR a, b, d, a, etc.

Terminology – Directed Graph

- **In-degree of a vertex:**
  Number of edges pointing into the node
- **Out-degree of a vertex:**
  Number of edges pointing out from the node
- **Directed path:**
  Sequence of distinct nodes, such that there is an edge from each vertex in the sequence to the next
- **Directed cycle:**
  A directed path in G that the last vertex in the sequence is pointing to the first vertex in the sequence

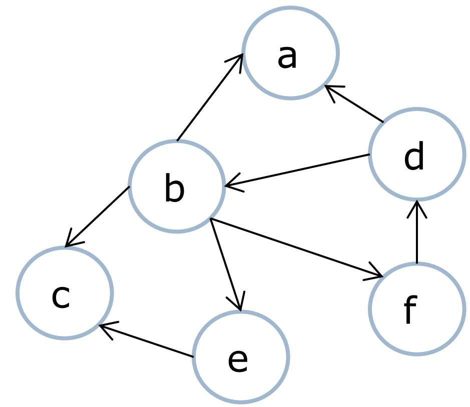


G = (V, E)
V = { a, b, c, d, e, f }
E = { (b,a), (b,e), (b,f), (b,c),
(d,a), (d,b), (e,c), (f,d) }

- In-degree of node b: 1
- Out-degree of node b: 4
- Directed path from d to c: d → b → c OR d → b → e → c
- Directed cycle: d → b → f → d

Terminology – Directed Graph

- **Connected:**
  A graph G is connected any two vertices x and y in G has a directed path with first vertex x and last vertex y
- **Directed complete graph:**
  A directed graph G has a directed edge between every pair of vertices in G
- **Directed cyclic graph:**
  A directed graph with directed cycle
- **Directed acyclic graph (DAG):**
  A directed graph without directed cycle



G = (V, E)
V = { a, b, c, d, e, f }
E = { (b,a), (b,e), (b,f), (b,c),
(d,a), (d,b), (e,c), (f,d) }

- The graph is NOT connected, since no path from e to a
- The graph is NOT complete, not every pair of vertices in G has directed edges, e.g. c and d
- The graph is a cyclic directed graph, since there are cycle, e.g. b → f → d → b.

Definitions

- Sequence of vertices, such that there is an edge from each vertex to the next in sequence, is called **path**.
  - First vertex in the path is called the; *start vertex*
  - Last vertex in the path is called the *end vertex*.
- If start and end vertices are the same, path is called **cycle**.

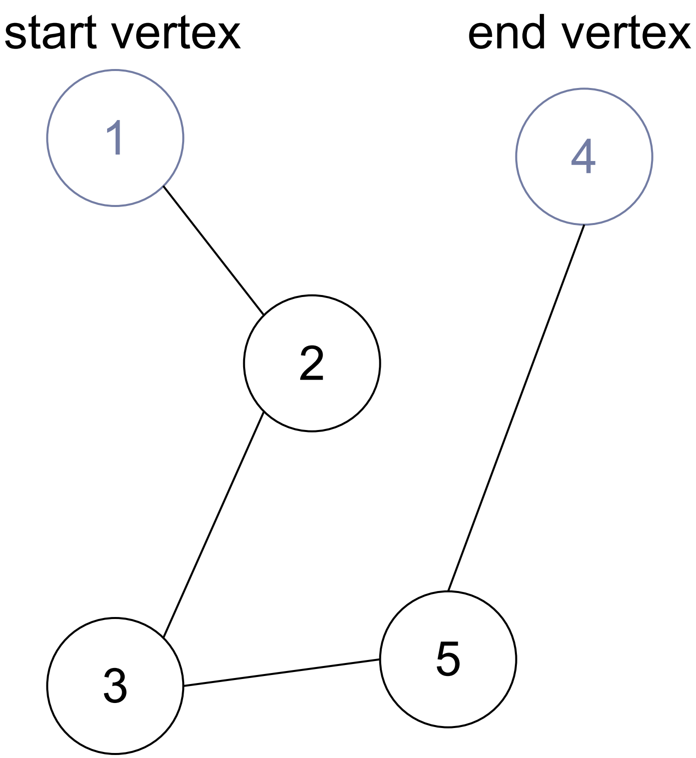
Definitions

- Path is called *simple*, if it includes every vertex only once.

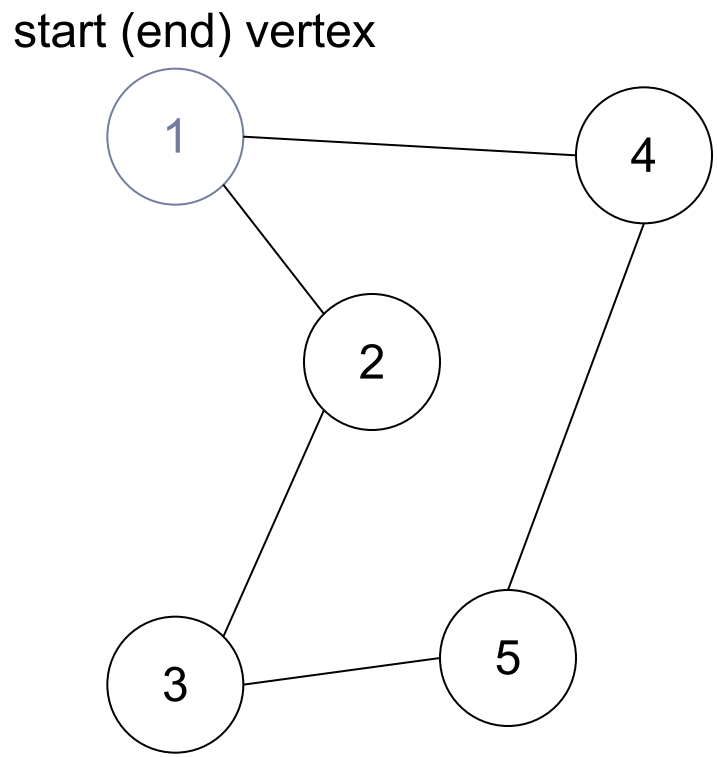
- Cycle is called *simple*, if it includes every vertex, except start (end) one, only once.

Definitions

- Let's see examples of path and cycle.



Path (simple)



Cycle (simple)

Definitions

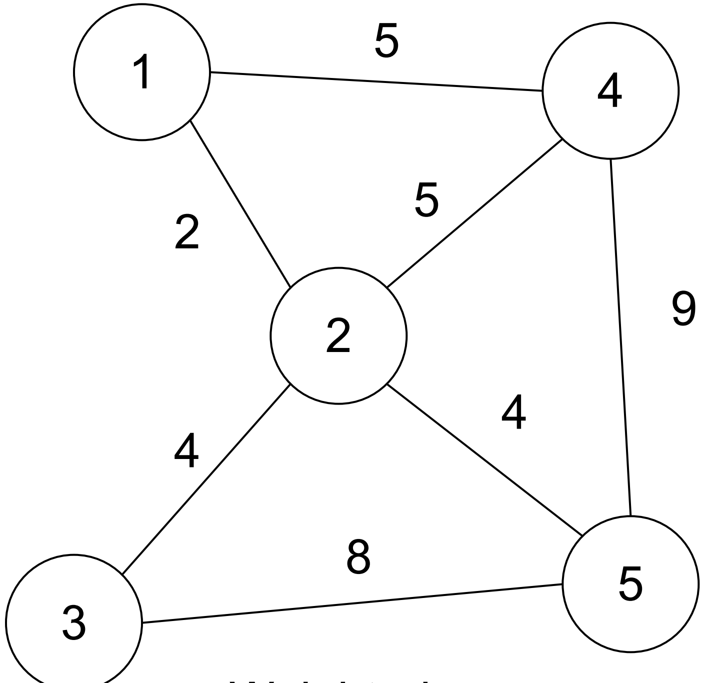
- The last definition we give here is a weighted graph.

- Graph is called *weighted*,

  - if every edge is associated with a real number, called edge weight.

Definitions

- For instance, in the road network example, weight of each road may be its length or minimal time needed to drive along.



Weighted graph

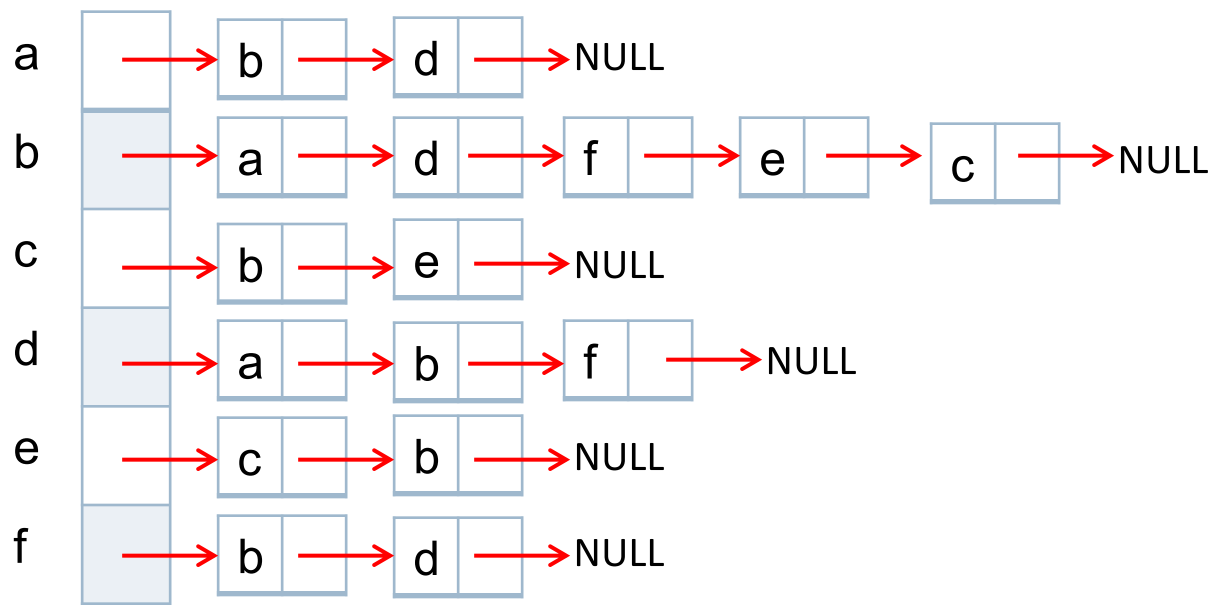
Graph Representation

	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Graph Representation

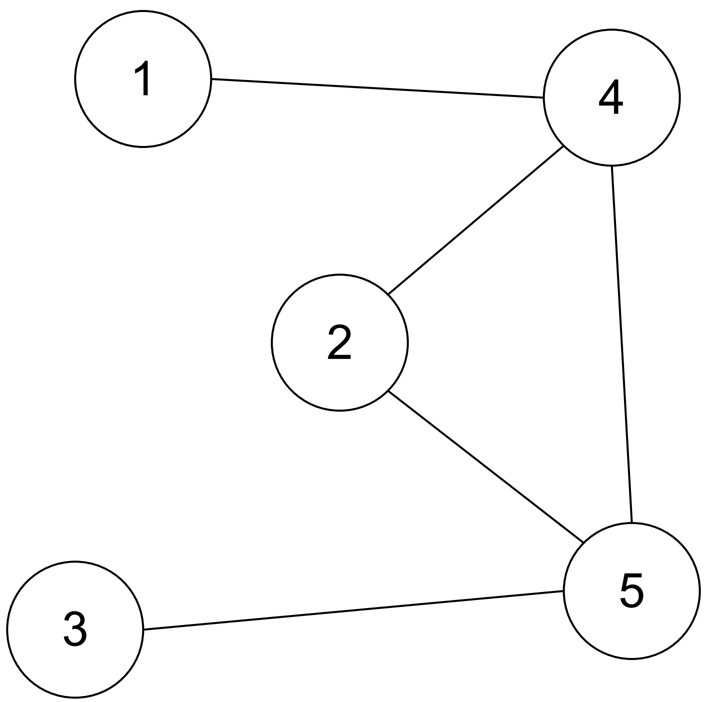
- There are several possible ways to represent a graph inside the computer. Two of them are:
  - Adjacency matrix and
  - Adjacency List

$$
M = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array}
\begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1
\end{bmatrix} \end{array}
$$



Adjacency Matrix

Graph



Adjacency matrix

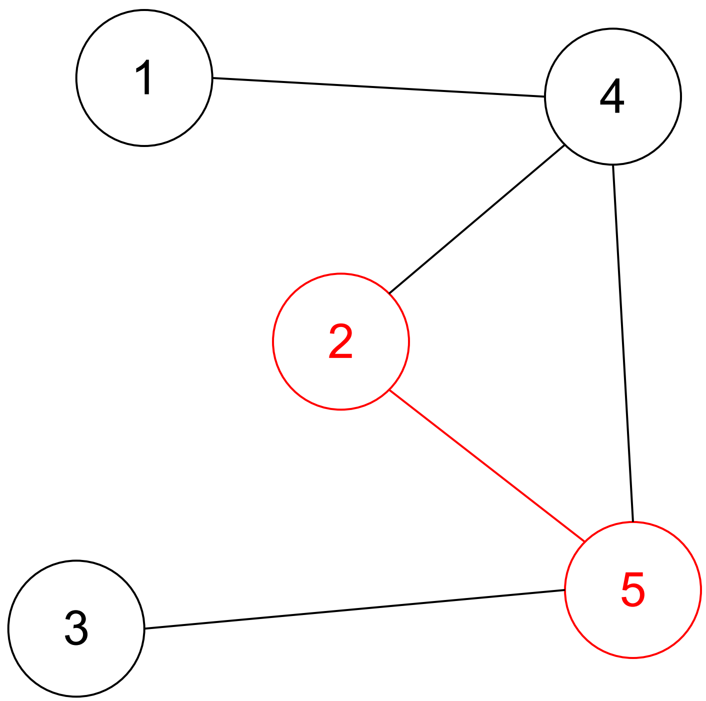
	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency Matrix

- Each cell a_{ij} of an adjacency matrix contains **0**.
- If there is an edge between i-th and j-th vertices, and **1** otherwise.

Adjacency Matrix

Edge(2, 5)

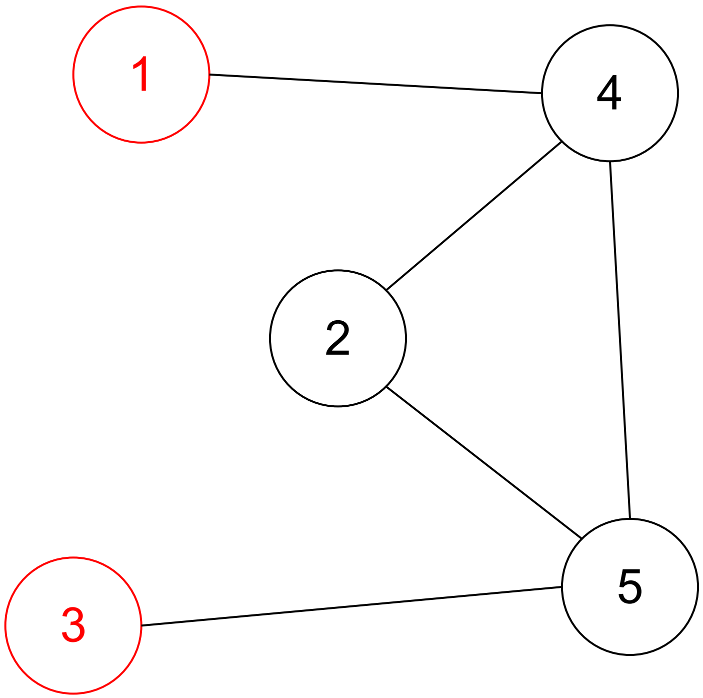


Cells for the edge(2, 5)

	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency Matrix

Edge(1, 3)



Cells for the edge(1, 3)

	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency Matrix

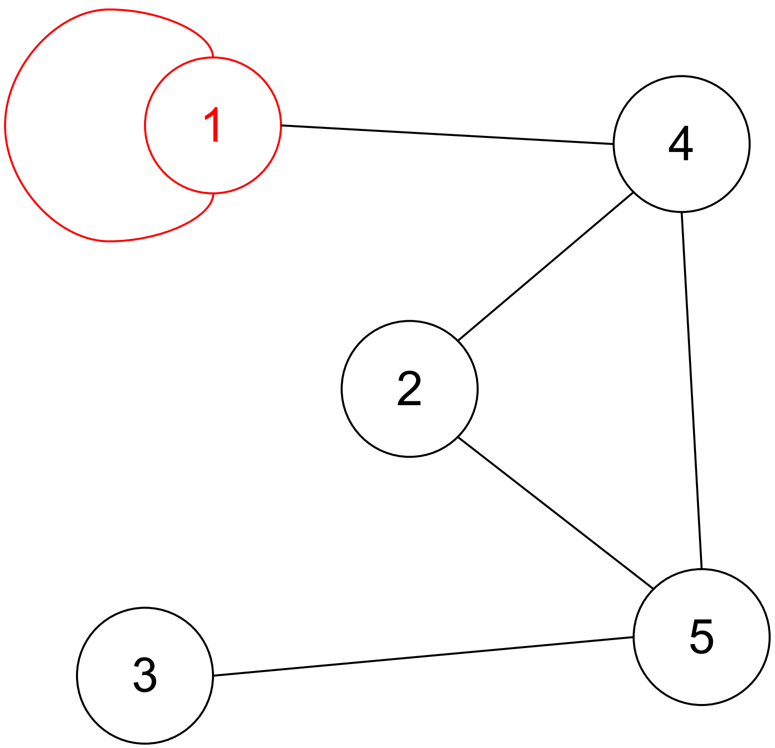
- The graph presented by example is undirected.
  - It means that its adjacency matrix is symmetric.
  - Indeed, in undirected graph, if there is an edge (2, 5) then there is also an edge (5, 2).
- This is also the reason, why there are two cells for every edge in the sample.

Adjacency Matrix

- Loops, if they are allowed in a graph, correspond to the diagonal elements of an adjacency matrix.

Adjacency Matrix

Edge(1, 3)

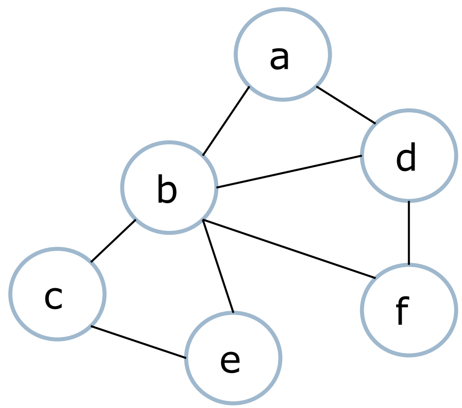


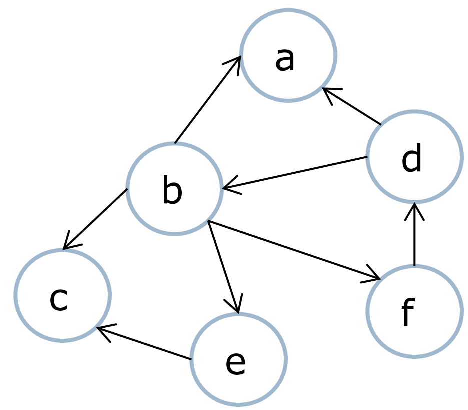
Cells for the edge(1, 3)

	1	2	3	4	5
1	1	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency Matrix

- Examples





$$M = \begin{array}{cc} & \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

$$M = \begin{array}{cc} & \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix

- **Advantages:**
  - Adjacency matrix is very convenient to work with.
  - Add (remove) an edge can be done very fast.
  - The same time is required to check, if there is an edge between two vertices.
  - Also it is very simple to program.

Adjacency Matrix

- **Disadvantages:**
  - Adjacency matrix consumes huge amount of memory for storing big graphs.
  - All graphs can be divided into two categories, *sparse* and *dense* graphs.
    - Sparse ones contain not much edges (number of edges is much less, that square of number of vertices, $|E| << |V|^2$).
    - On the other hand, dense graphs contain number of edges comparable with square of number of vertices.
  - Adjacency matrix is optimal for dense graphs, but for sparse ones it is superfluous.

Adjacency Matrix

- **Disadvantages:**
  - The next disadvantage is that adjacency matrix requires huge efforts for adding/removing a vertex.
  - In case, a graph is used for analysis only, it is not necessary, but if you want to construct fully dynamic structure, using of adjacency matrix make it quite slow for big graphs.

Adjacency Matrix

- To sum up
  - Adjacency matrix is a good solution for dense graphs, which implies having constant number of vertices.

Adjacency list

Adjacency list

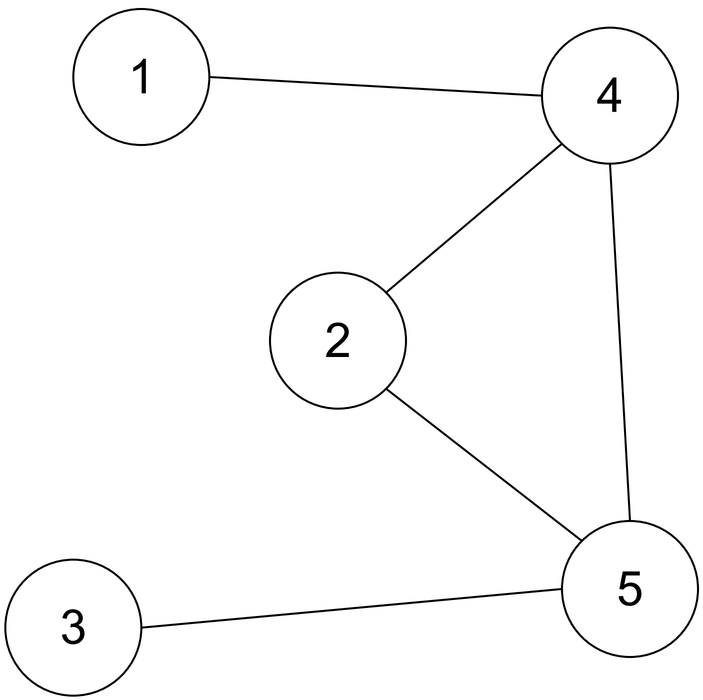
- This kind of the graph representation is one of the alternatives to adjacency matrix.

- It requires less amount of memory and, in particular situations even can outperform adjacency matrix.

- For every vertex adjacency list stores a list of vertices, which are adjacent to current one.

Adjacency list

Graph

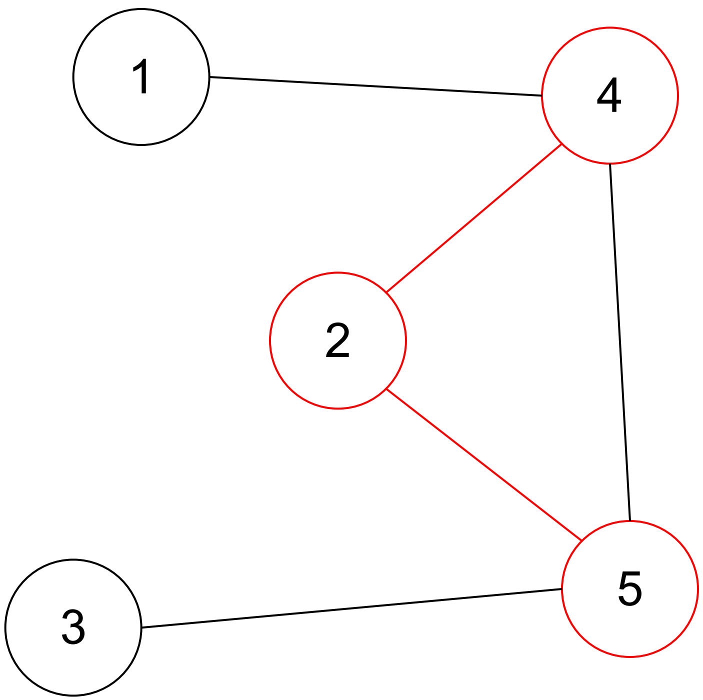


Adjacency list

1	4		
2	4	5	
3	5		
4	1	2	5
5	2	3	4

Adjacency list

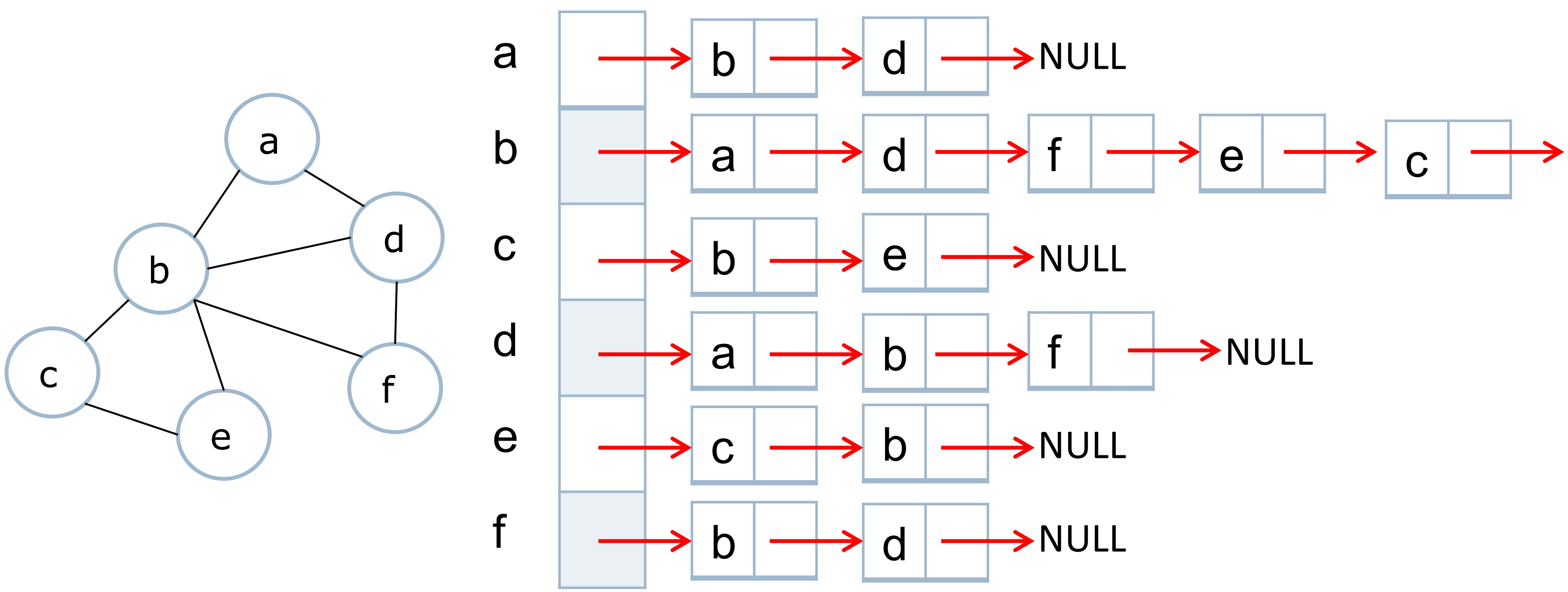
Vertices, adjacent to {2}



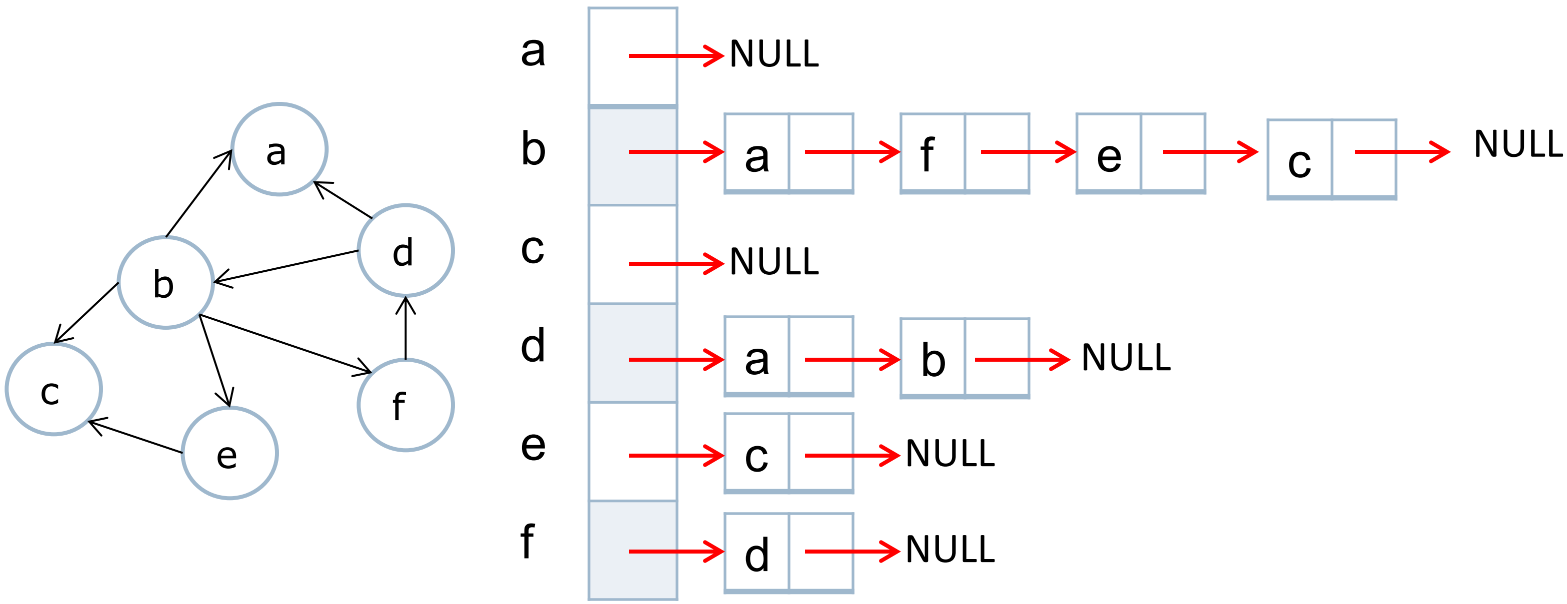
Row in the adjacency list

1	4		
2	4	5	
3	5		
4	1	2	5
5	2	3	4

Adjacency list



Adjacency list



Adjacency list

- **Advantages:**
  - Adjacent list allows us to store graph in more compact form, than adjacency matrix,
  - But the difference decreasing as a graph becomes denser.
  - Next advantage is that adjacent list allows to get the list of adjacent vertices very fast, which is a big advantage for some algorithms.

Adjacency list

- **Disadvantages:**
  - Adding/removing an edge to/from adjacent list is not so easy as for adjacency matrix.
  - Adjacent list doesn't allow us to make an efficient implementation

Adjacency list

- To sum up

  - Adjacency list is a good solution for sparse graphs and lets us changing number of vertices more efficiently, than if using an adjacent matrix.

  - But still there are better solutions to store fully dynamic graphs.

Graph Traversal

- Traverse a graph means
  - Visit all the graph nodes / vertices
  - The order of visit depends on the traversal algorithms

- Traversal algorithms
  - Breath-First Search traversal (BFS)
  - Depth-First Search traversal (DFS)

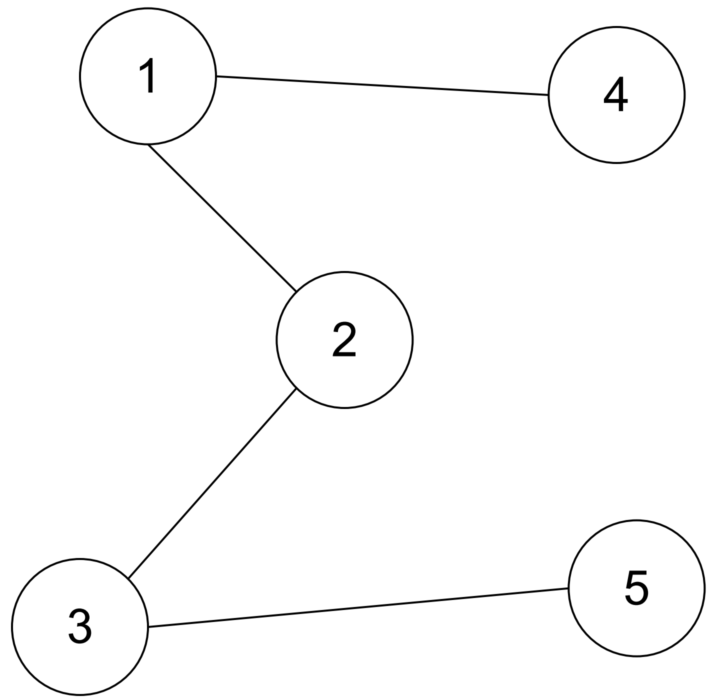
Breadth-First Search

Breadth-First Search

- **Breadth-First Search** of a graph is similar to traversing a binary tree level-by-level.
  - All the nodes at any level, i, are visited before visiting the nodes at level i+1.

Breadth-First Search

- The breadth-first ordering of the vertices of the following graph is as follows:
  - 1 2 4 3 5



Breadth-First Search

- The breadth-first search traverses the graph from each vertex that is not visited.
  - Starting at the first vertex, the graph is traversed as much as possible
  - Then go to the next vertex that has not been visited.

Breadth-First Search

- To implement the breadth-first search algorithm, we use a queue.

- The general algorithm is as follows:

1. for each vertex v in the graph

   if v is not visited

   add v to the queue

2. Mark v as visited

BREADTH-FIRST SEARCH

- The general algorithm is as follows (cont'):

3. while the queue is not empty

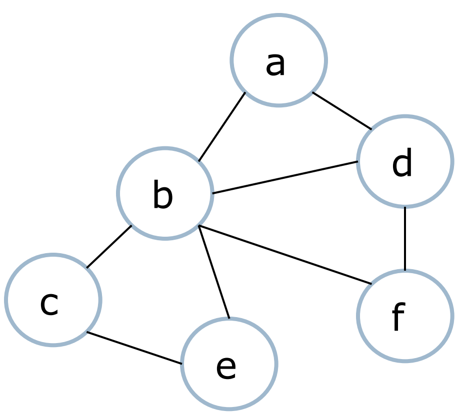
   1. Remove vertex u from the queue

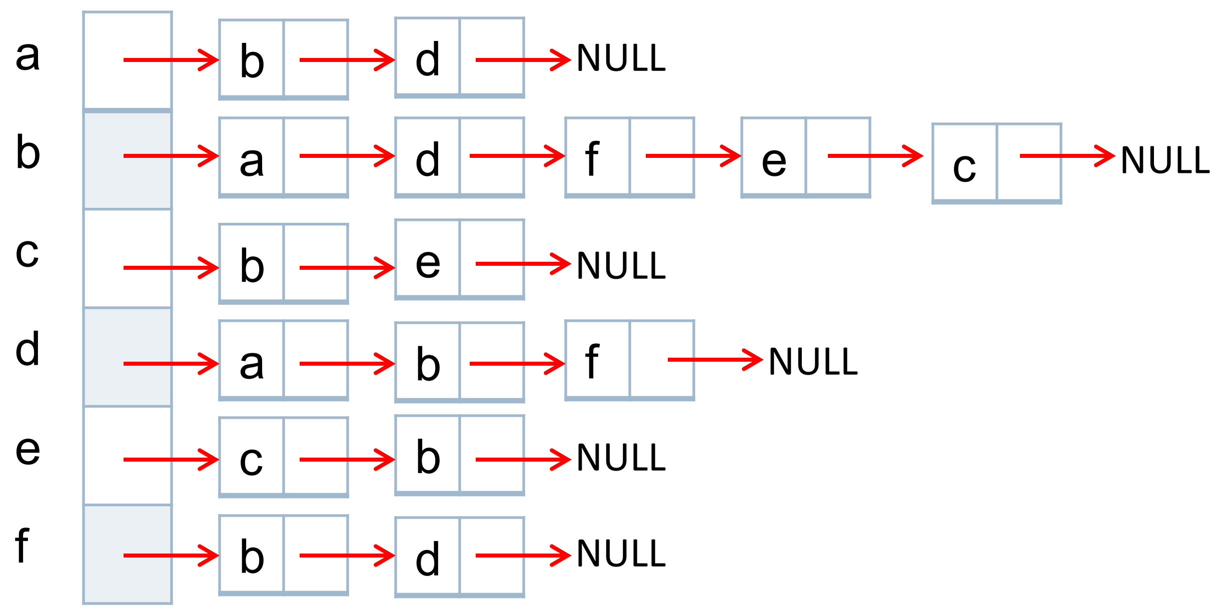
   2. Retrieve the vertices adjacent to u

   3. for each vertex w that is adjacent to u
      if w is not visited
      Add w to the queue
      Mark w as visited

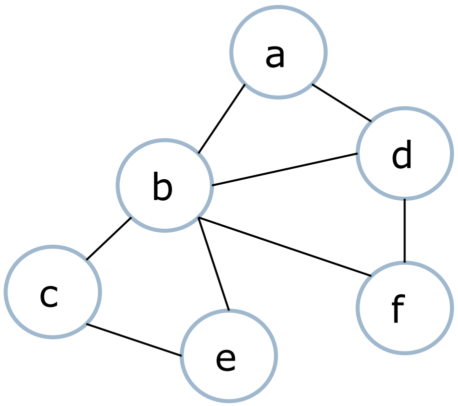
Breadth-First Search - Example

- Assume
  - Start from node a
  - Use adjacency list as graph representation



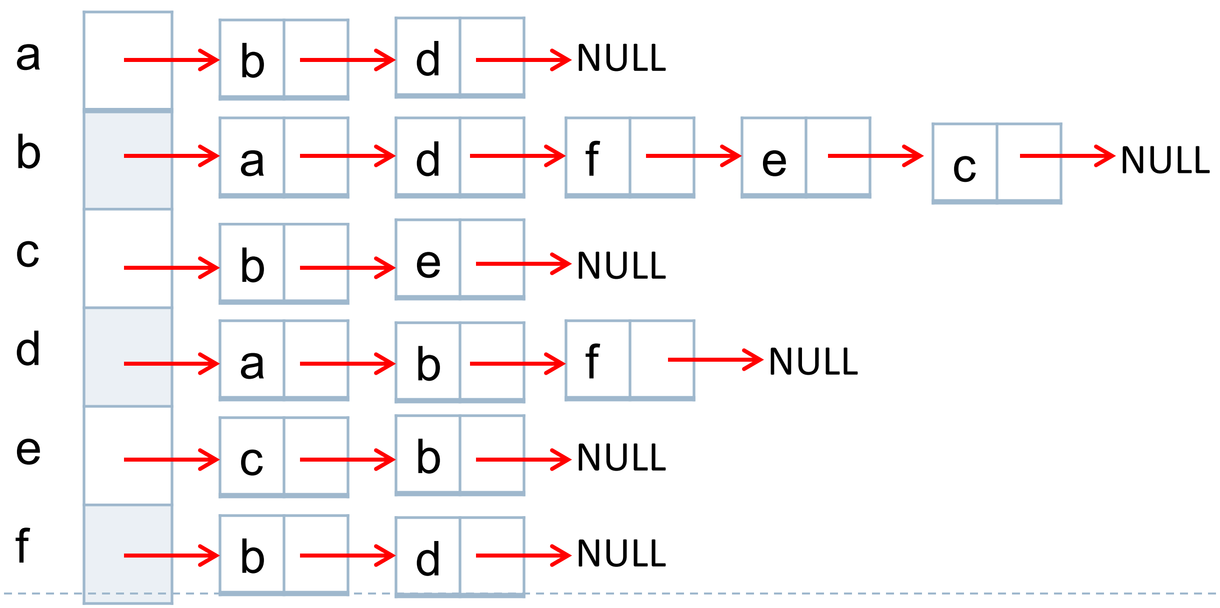


Breadth-First Search – Example (Initial)



Q = { }

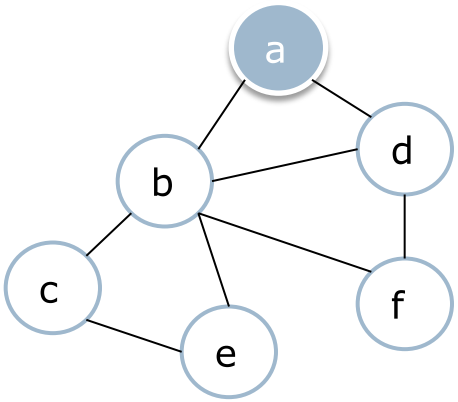
Order of visit:



Visited Table

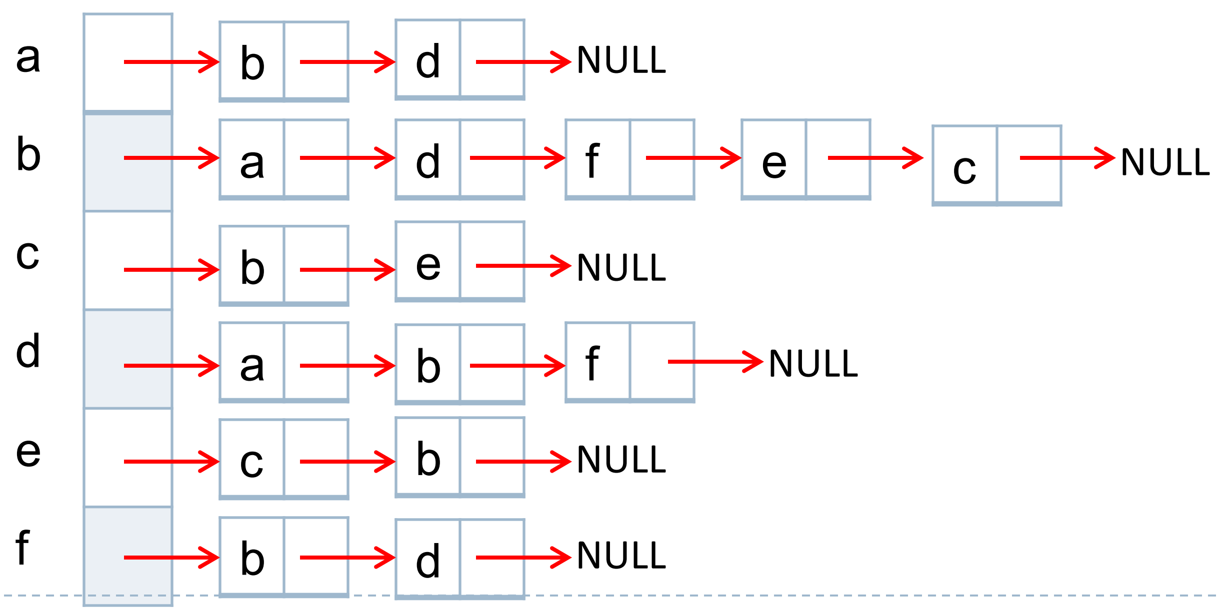
a	F
b	F
c	F
d	F
e	F
f	F

Breadth-First Search – Example (Step 1)



Q = { a }

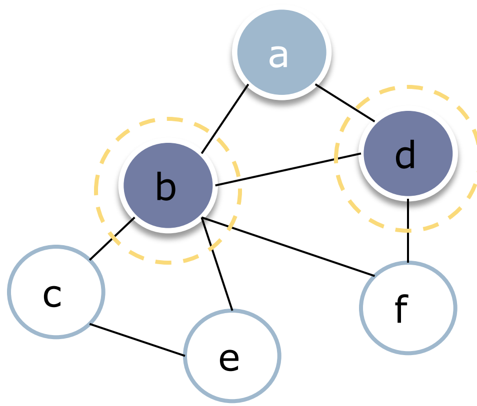
Order of visit: a



Visited Table

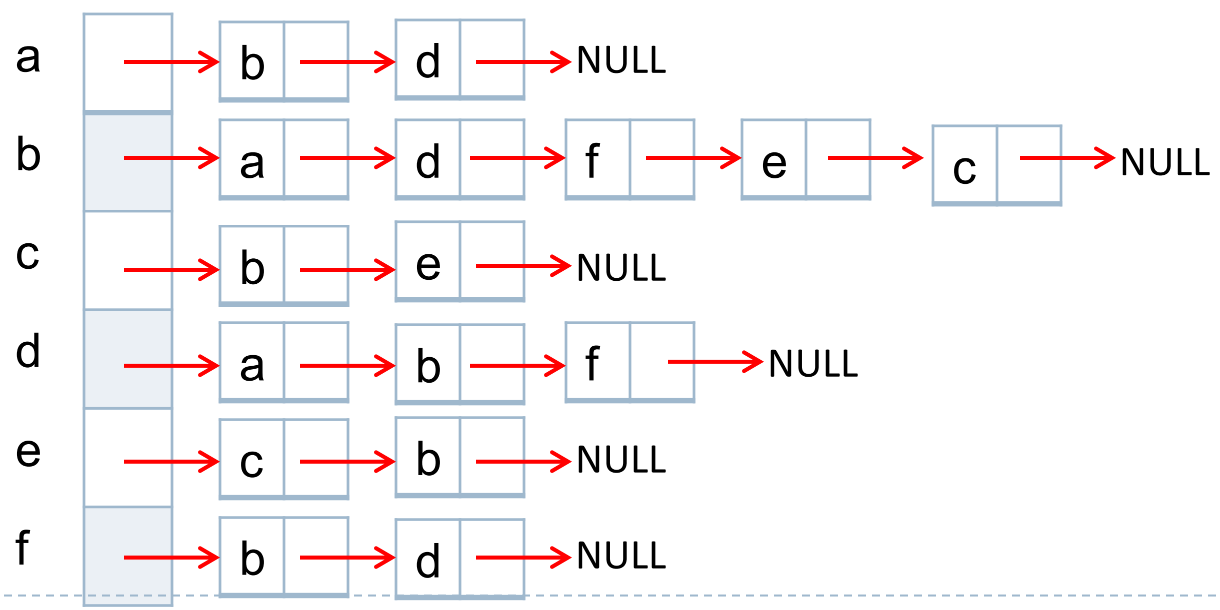
a	T
b	F
c	F
d	F
e	F
f	F

Breadth-First Search – Example (Step 2)



Q = { b, d }

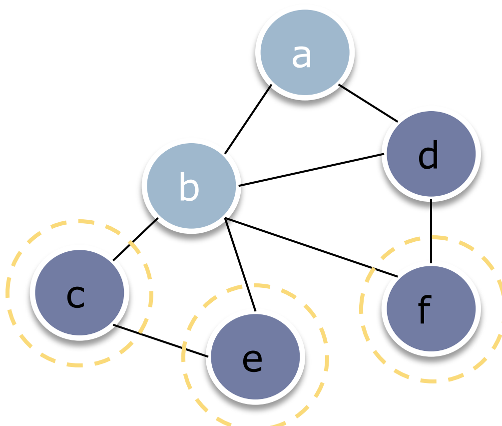
Order of visit: a



Visited Table

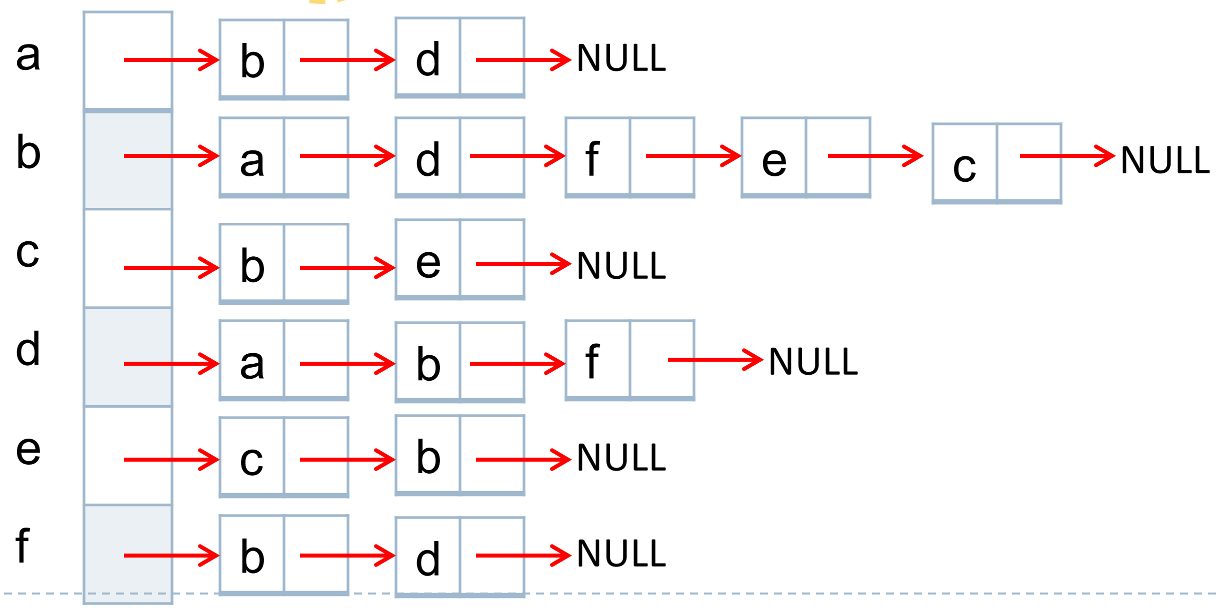
a	T
b	T
c	F
d	T
e	F
f	F

Breadth-First Search – Example (Step 3)



Q = { d, f, e, c }

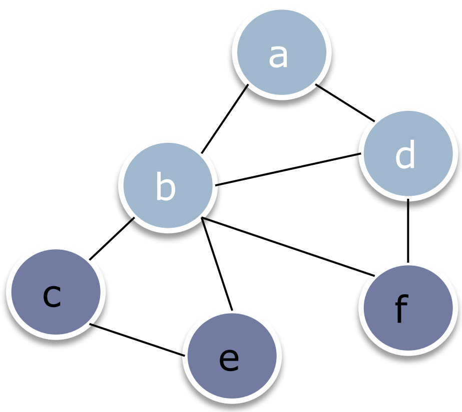
Order of visit: a, b



Visited Table

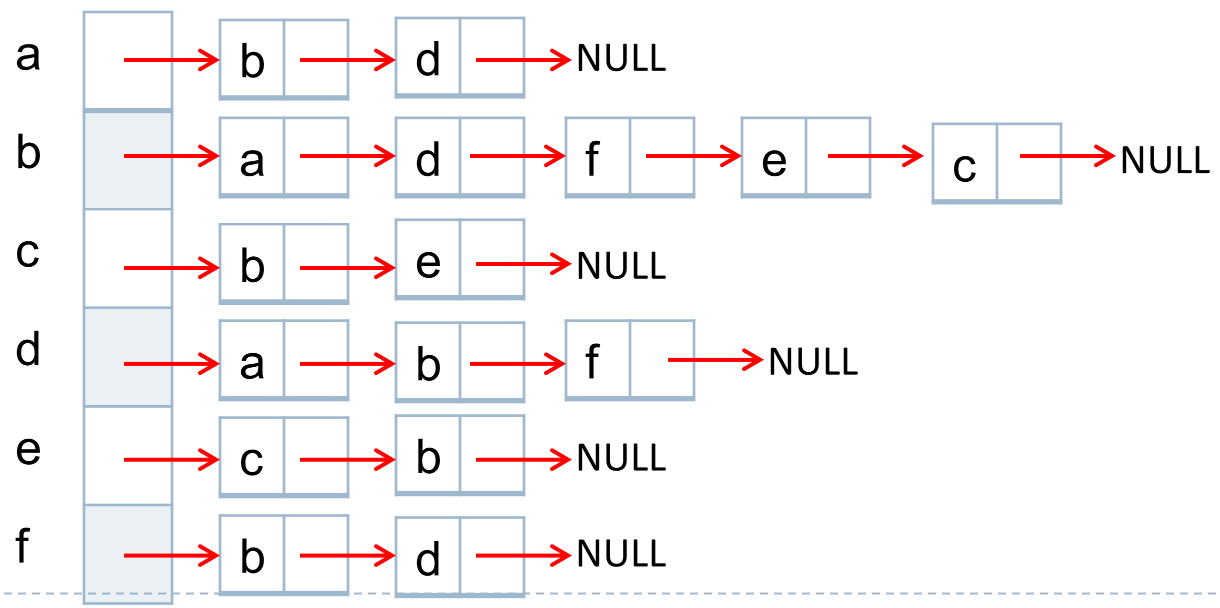
a	T
b	T
c	T
d	T
e	T
f	T

Breadth-First Search – Example (Step 4)



Q = { f, e, c }

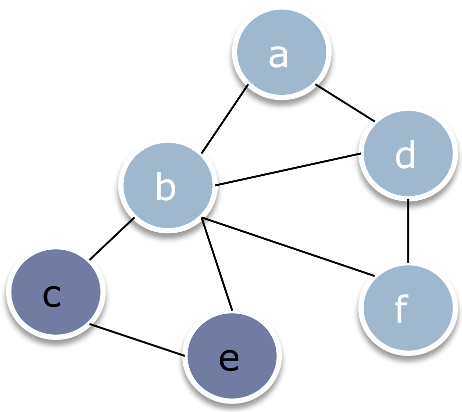
Order of visit: a, b, d



Visited Table

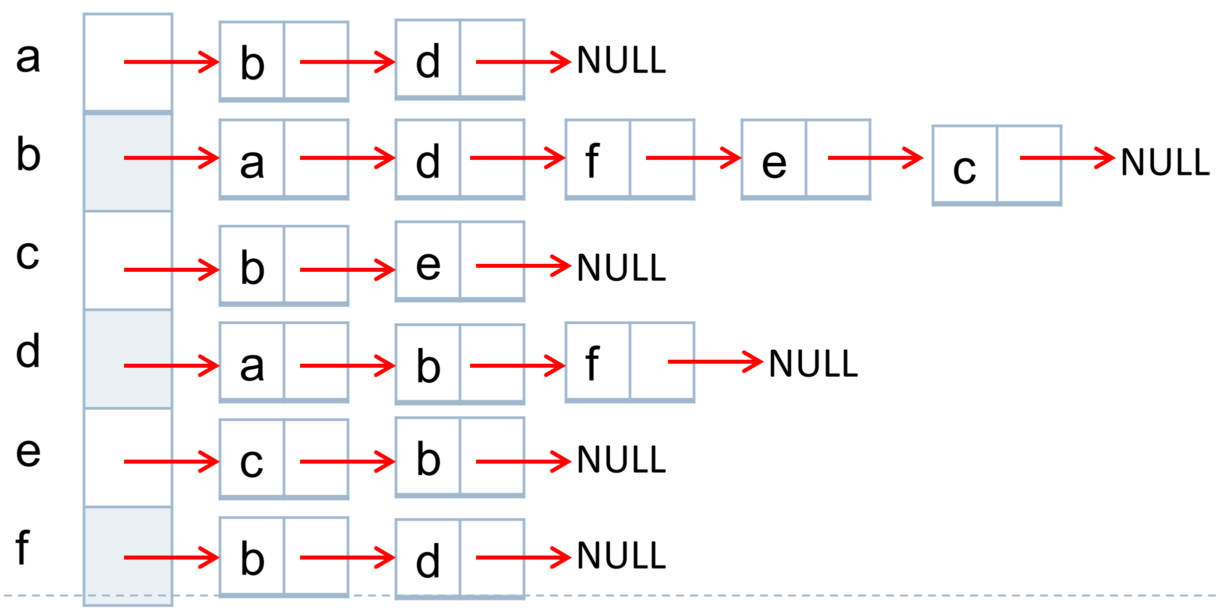
a	T
b	T
c	T
d	T
e	T
f	T

Breadth-First Search – Example (Step 5)



Q = { e, c }

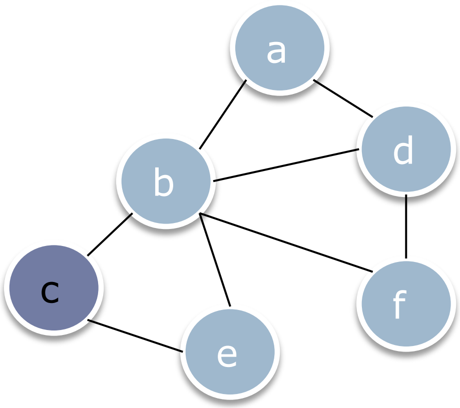
Order of visit: a, b, d, f



Visited Table

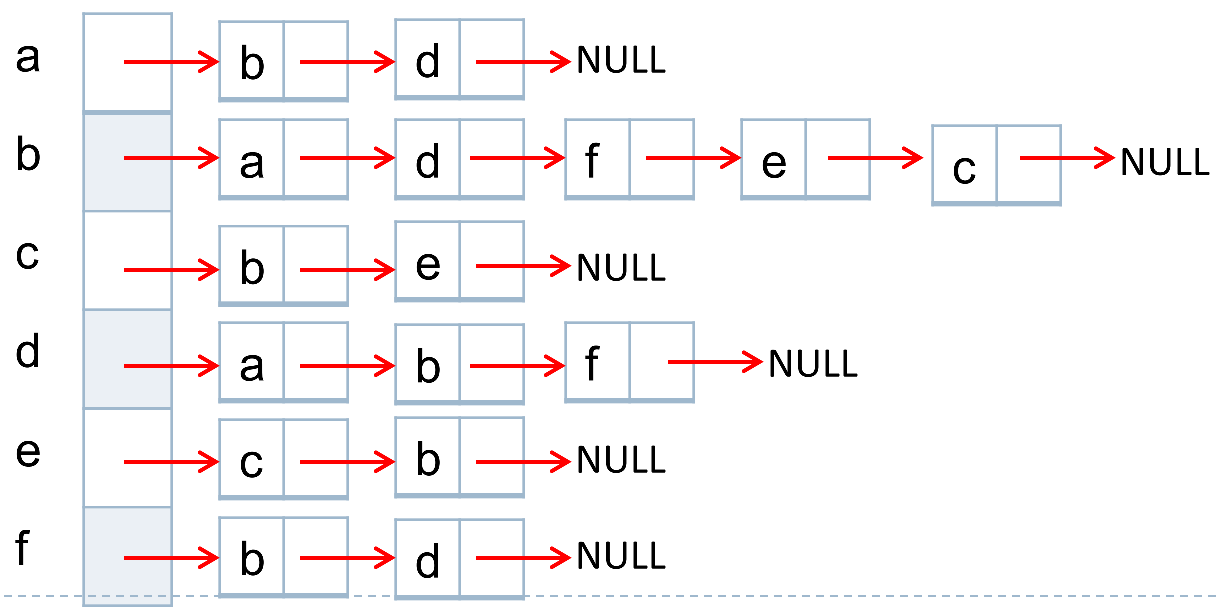
a	T
b	T
c	T
d	T
e	T
f	T

Breadth-First Search – Example (Step 6)



Q = { c }

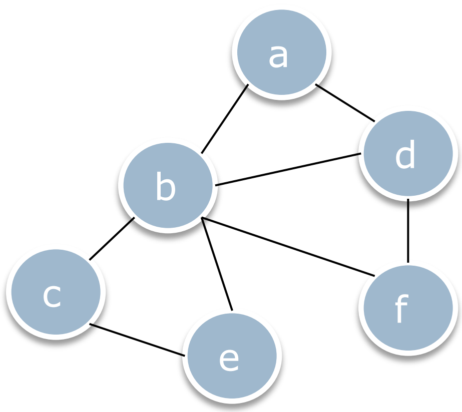
Order of visit: a, b, d, f, e



Visited Table

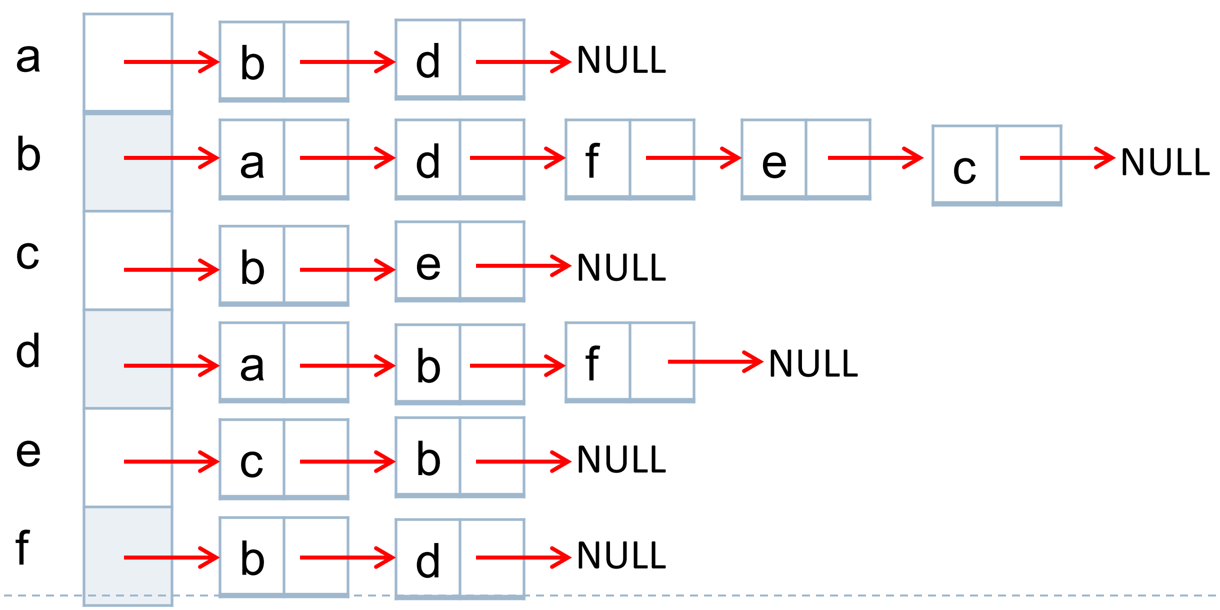
a	T
b	T
c	T
d	T
e	T
f	T

Breadth-First Search – Example (Step 7)



Q = { }

Order of visit: a, b, d, f, e, c



Visited Table

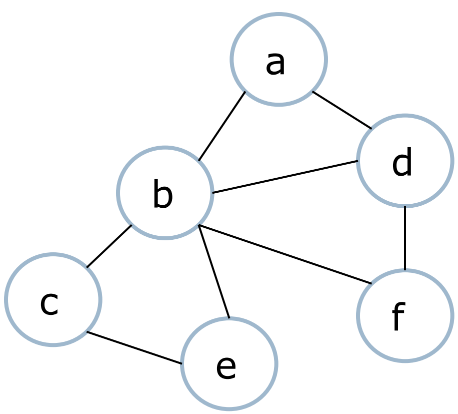
a	T
b	T
c	T
d	T
e	T
f	T

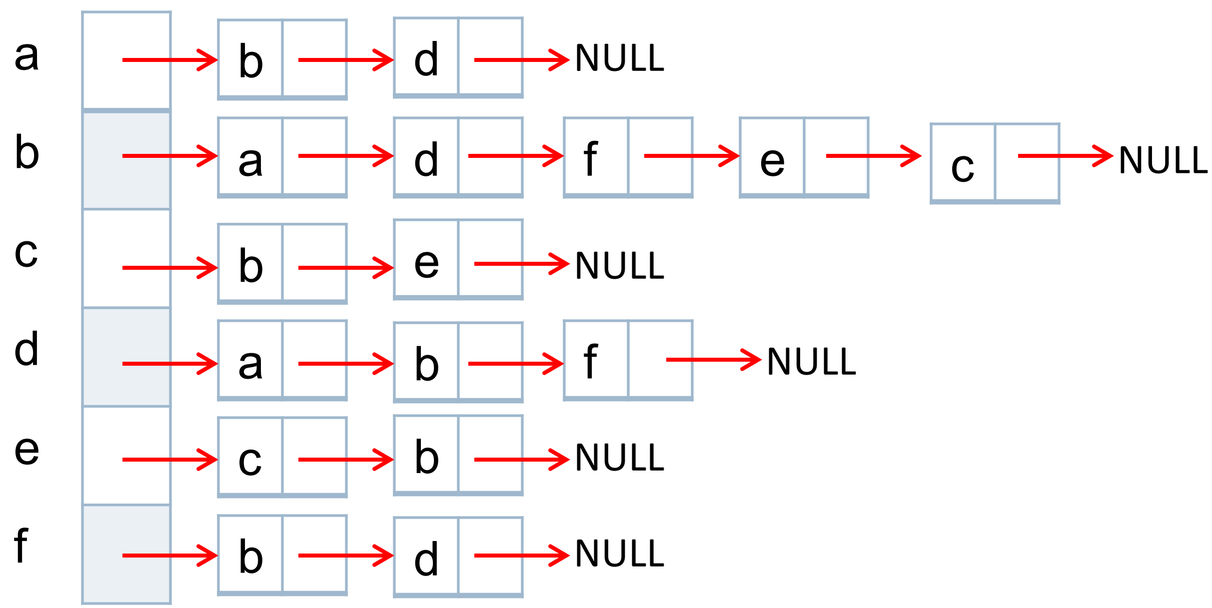
Shortest Path Finding using Breadth-First Search (BFS)

- The BFS introduced just now only let us know whether a path exists from the source to other nodes, but it doesn't record the paths
- We could slightly modify the algorithm to record to the path from s to each node and the shortest path length
- Algorithm:
  - For each node / vertex v in graph, mark every vertex as unvisited, **set all entries of predecessor array to NULL and distance array to infinity**
  - Mark the start node S as visited and distance from s to 0
  - enqueue S to a queue Q
  - while(Q is NOT empty)
    v = dequeue Q
    for each w adjacent to v
      if w is not visited, then
        mark it to visited, **set the predecessor to v, d(w) = d(v)+1** &
          enqueue it to Q

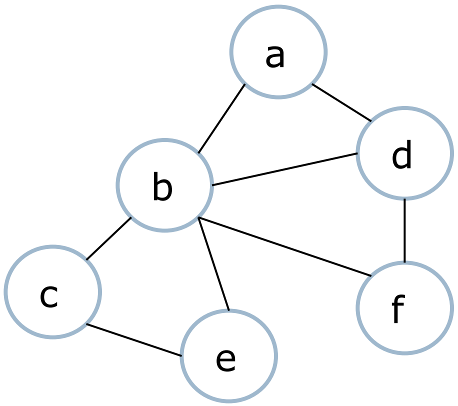
Breadth-First Search - Example

- Assume
  - Start from node a
  - Use adjacency list as graph representation





Breadth-First Search – Example (Initial)



Q = { }

Order of visit:

a → b → d → NULL
b → a → d → f → e → c → NULL
c → b → e → NULL
d → a → b → f → NULL
e → c → b → NULL
f → b → d → NULL

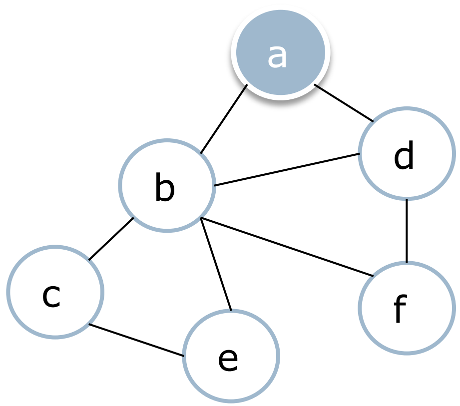
Visited Table

a	F
b	F
c	F
d	F
e	F
f	F

Predecessor and distance table

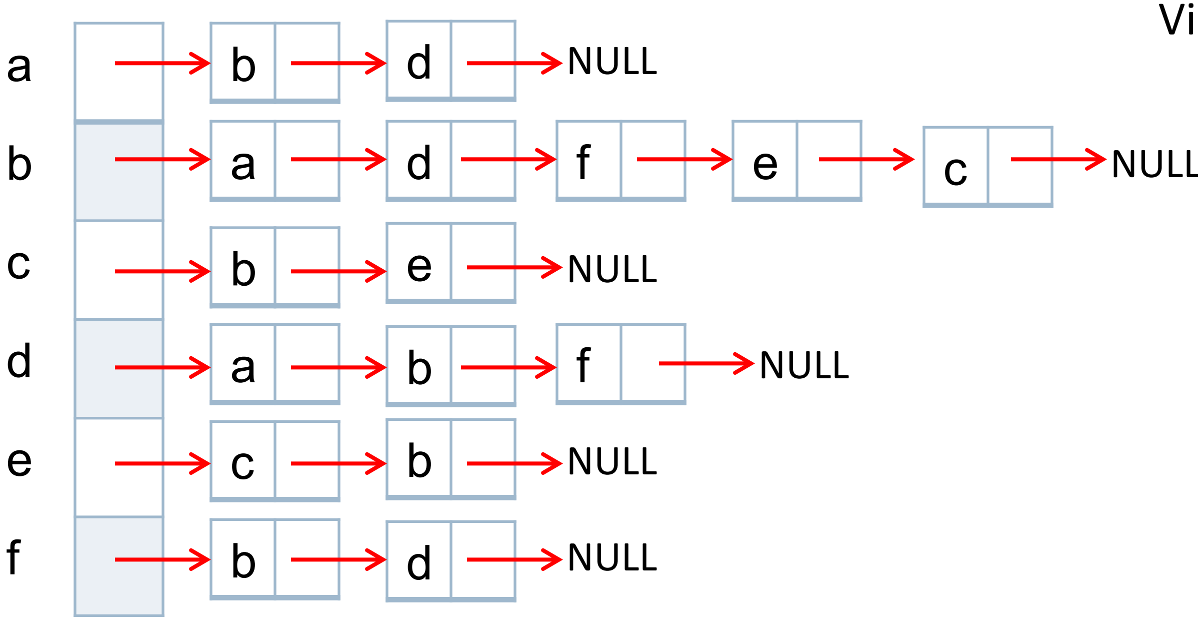
a	NULL	∞
b	NULL	∞
c	NULL	∞
d	NULL	∞
e	NULL	∞
f	NULL	∞

Breadth-First Search – Example (Step 1)



Q = { a }

Order of visit: a



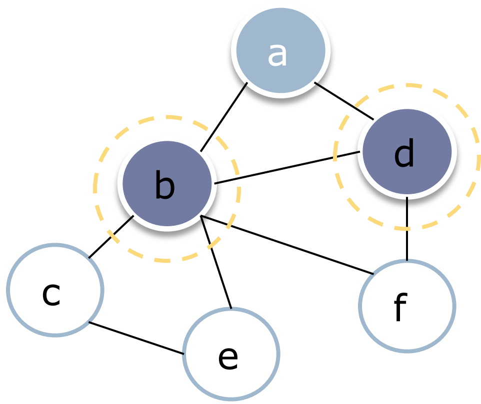
Visited Table

a	T
b	F
c	F
d	F
e	F
f	F

Predecessor and distance table

a	NULL	0
b	NULL	∞
c	NULL	∞
d	NULL	∞
e	NULL	∞
f	NULL	∞

Breadth-First Search – Example (Step 2)



Q = { b, d }

Order of visit: a

a → b → d → NULL

b → a → d → f → e → c → NULL

c → b → e → NULL

d → a → b → f → NULL

e → c → b → NULL

f → b → d → NULL

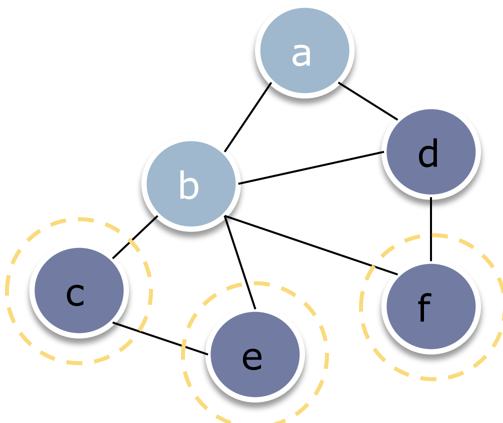
Visited Table

a	T
b	T
c	F
d	T
e	F
f	F

Predecessor and distance table

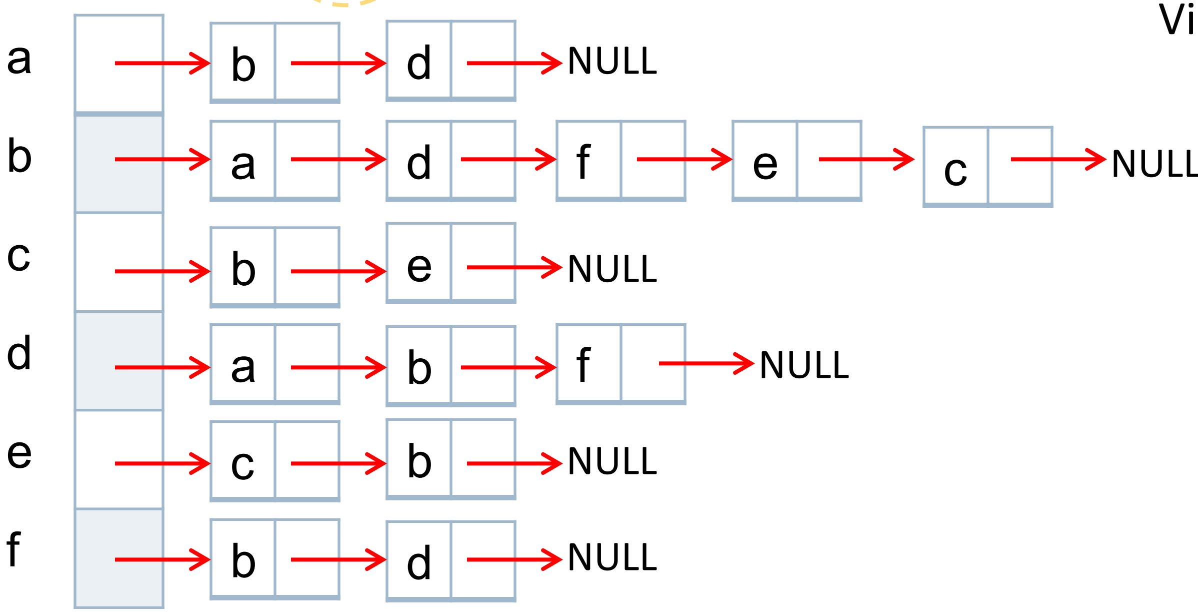
a	NULL	0
b	a	1
c	NULL	∞
d	a	1
e	NULL	∞
f	NULL	∞

Breadth-First Search – Example (Step 3)



Q = { d, f, e, c }

Order of visit: a, b



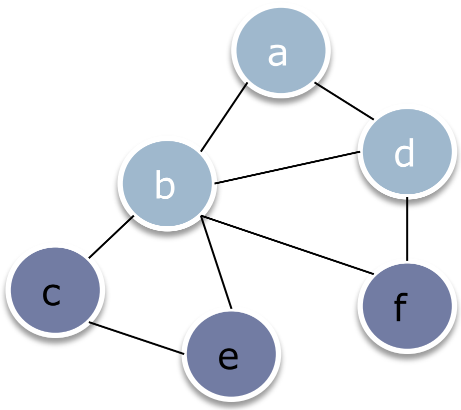
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

Predecessor and distance table

a	NULL	0
b	a	1
c	b	2
d	a	1
e	b	2
f	b	2

Breadth-First Search – Example (Step 4)



Q = { f, e, c }

Order of visit: a, b, d

a → b → d → NULL

b → a → d → f → e → c → NULL

c → b → e → NULL

d → a → b → f → NULL

e → c → b → NULL

f → b → d → NULL

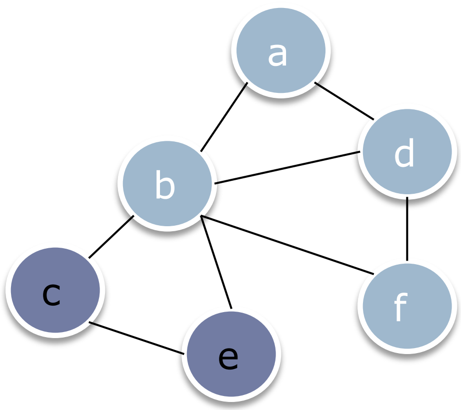
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

Predecessor and distance table

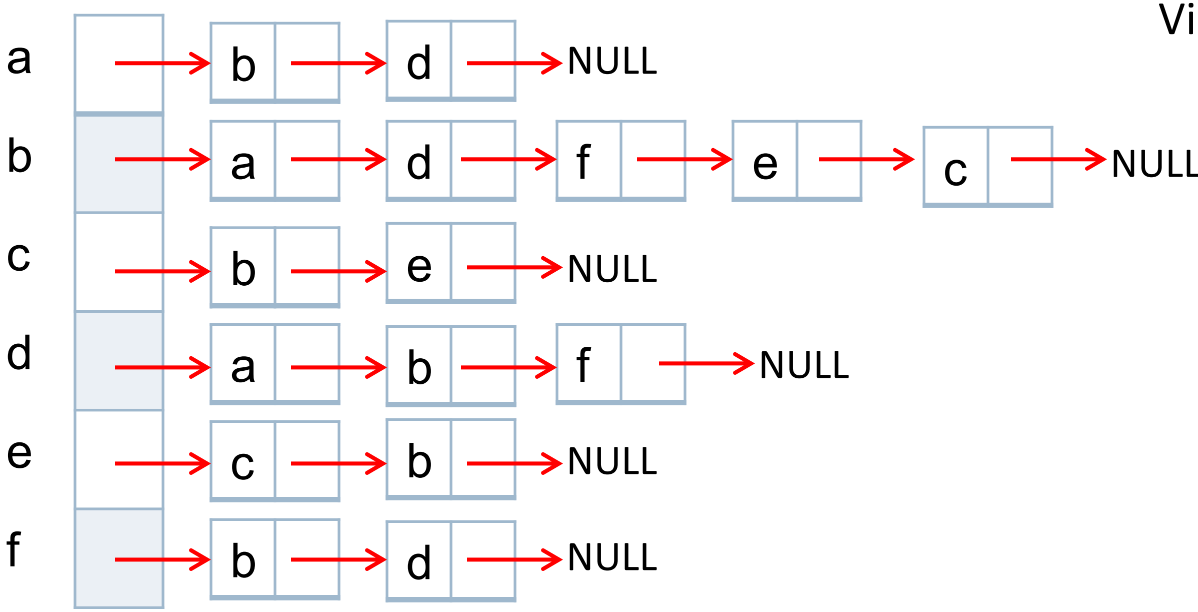
a	NULL	0
b	a	1
c	b	2
d	a	1
e	b	2
f	b	2

Breadth-First Search – Example (Step 5)



Q = { e, c }

Order of visit: a, b, d, f



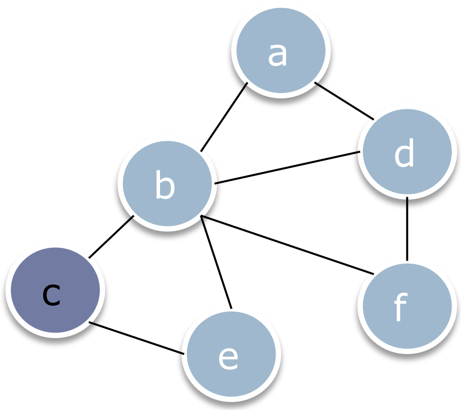
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

Predecessor and distance table

a	NULL	0
b	a	1
c	b	2
d	a	1
e	b	2
f	b	2

Breadth-First Search – Example (Step 6)



Q = { c }

Order of visit: a, b, d, f, e

a → b → d → NULL

b → a → d → f → e → c → NULL

c → b → e → NULL

d → a → b → f → NULL

e → c → b → NULL

f → b → d → NULL

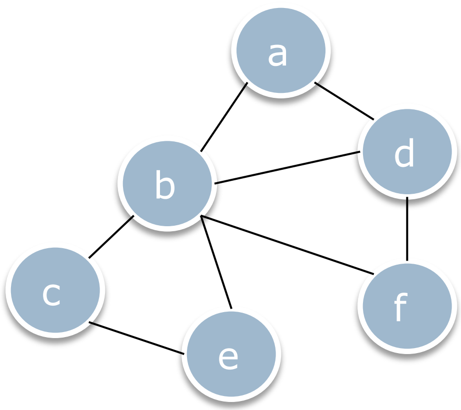
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

Predecessor and distance table

a	NULL	0
b	a	1
c	b	2
d	a	1
e	b	2
f	b	2

Breadth-First Search – Example (Step 7)



Q = { }

Order of visit: a, b, d, f, e, c

a → b → d → NULL
b → a → d → f → e → c → NULL
c → b → e → NULL
d → a → b → f → NULL
e → c → b → NULL
f → b → d → NULL

Visited Table

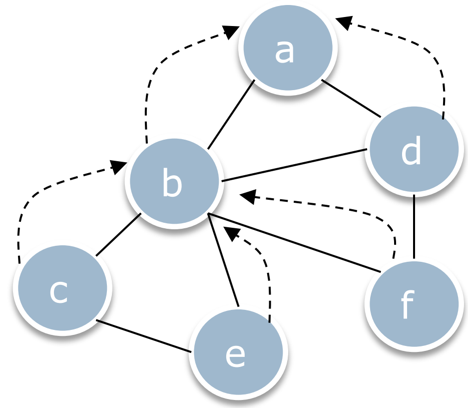
a	T
b	T
c	T
d	T
e	T
f	T

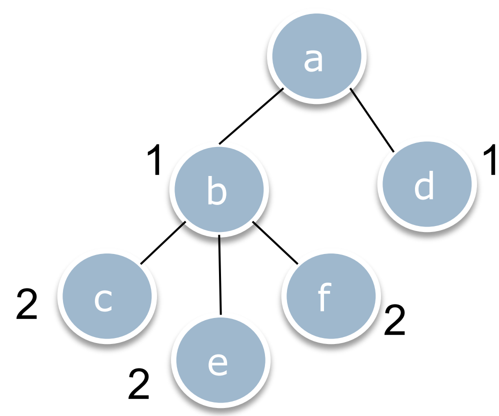
Predecessor and distance table

a	NULL	0
b	a	1
c	b	2
d	a	1
e	b	2
f	b	2

Paths from Source to Each Node

- Algorithm:
  Path(w)
  - If predecessor[w] is not NULL
    Path(predecessor[w])
  - Output w





BFS Tree

Predecessor and distance table

a	NULL	0
b	a	1
c	b	2
d	a	1
e	b	2
f	b	2

Depth-First Search

Depth-First Search

- The principle of the algorithm is quite simple:
  to go forward (in depth) while there is such possibility,
  otherwise to backtrack.

Depth-First Search

- **Algorithm**

  - In DFS, each vertex has three possible colors representing its state:

  - white: vertex is unvisited;

  - gray: vertex is in progress;

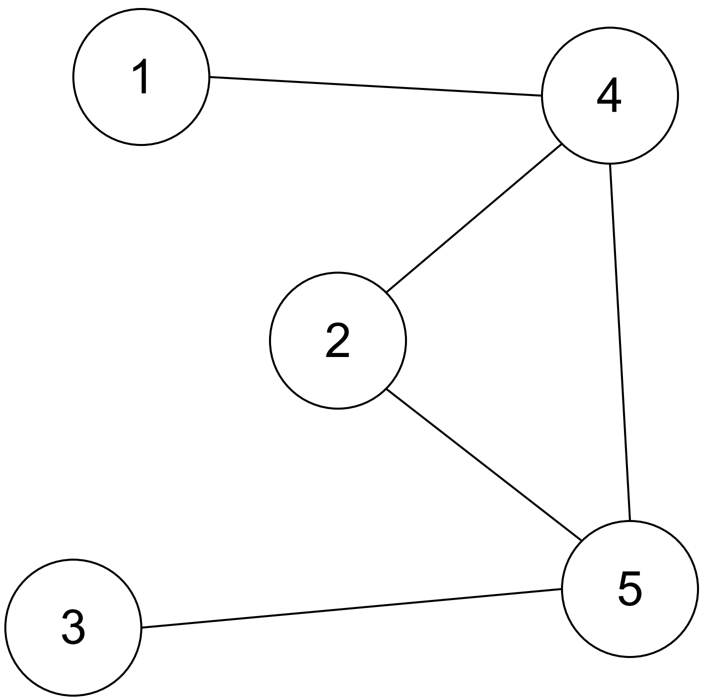
  - black: DFS has finished processing the vertex.

Depth-First Search

- Initially all vertices are white (unvisited).
  DFS starts in arbitrary vertex and runs as follows:

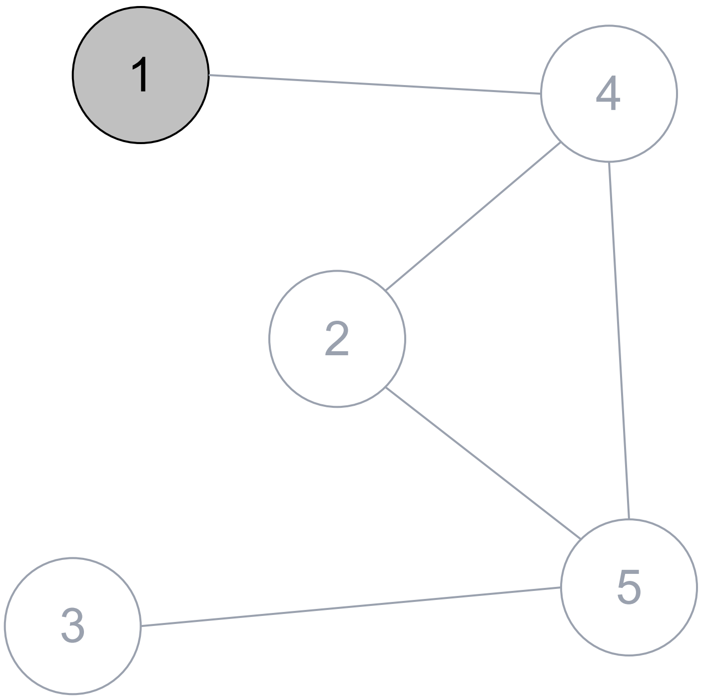
1. Mark vertex **u** as gray (visited).
2. For each edge **(u, v)**, where **v** is white, run depth-first search for **v** recursively.
3. Mark vertex **u** as black and backtrack to the parent.

Depth-First Search



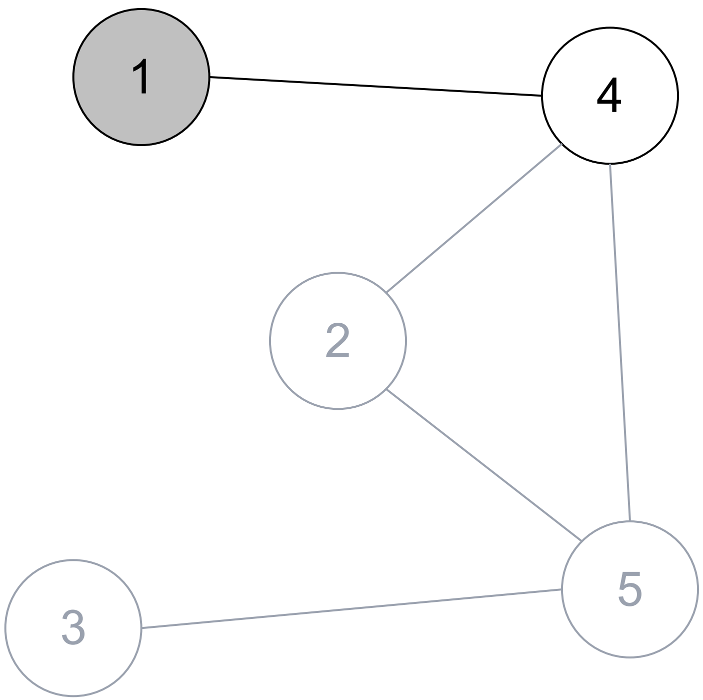
- Start from a vertex with number 1

Depth-First Search



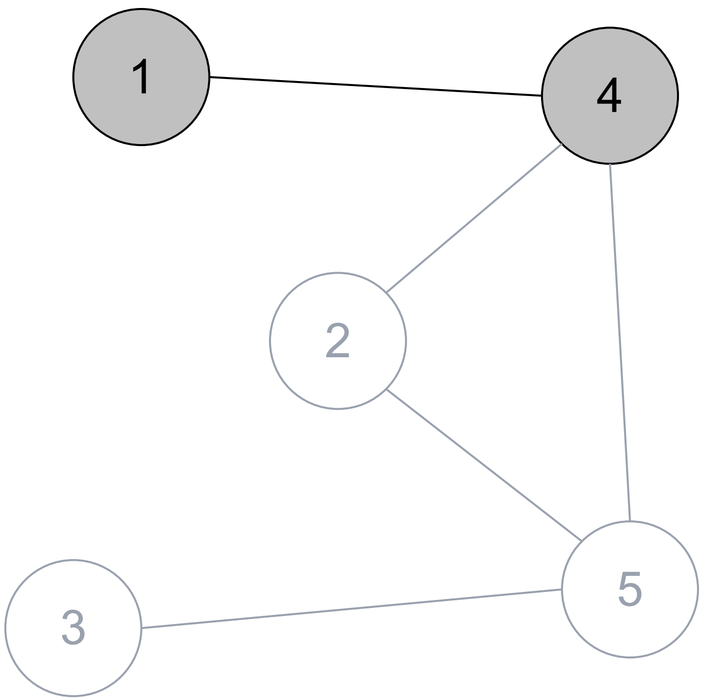
- Mark vertex 1 as gray.

Depth-First Search



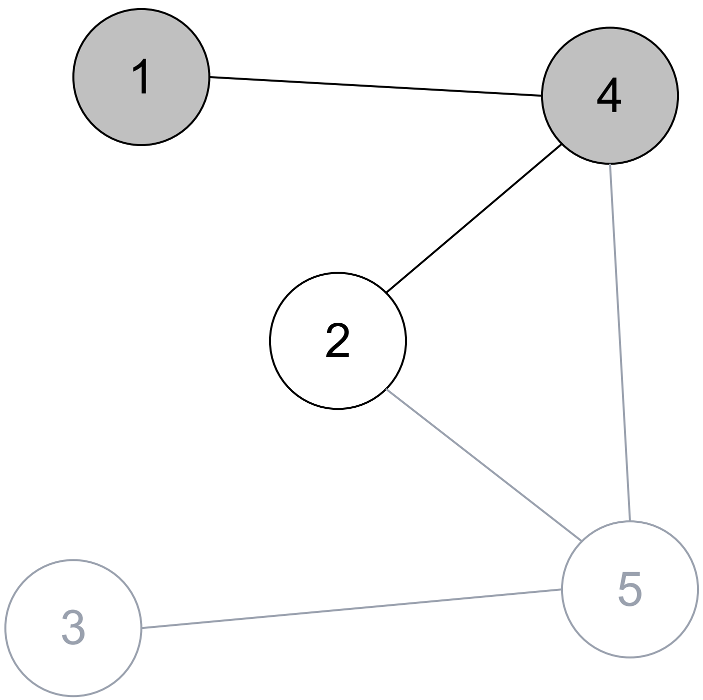
- There is an edge (1, 4) and a vertex 4 is unvisited.
- Go there.

Depth-First Search



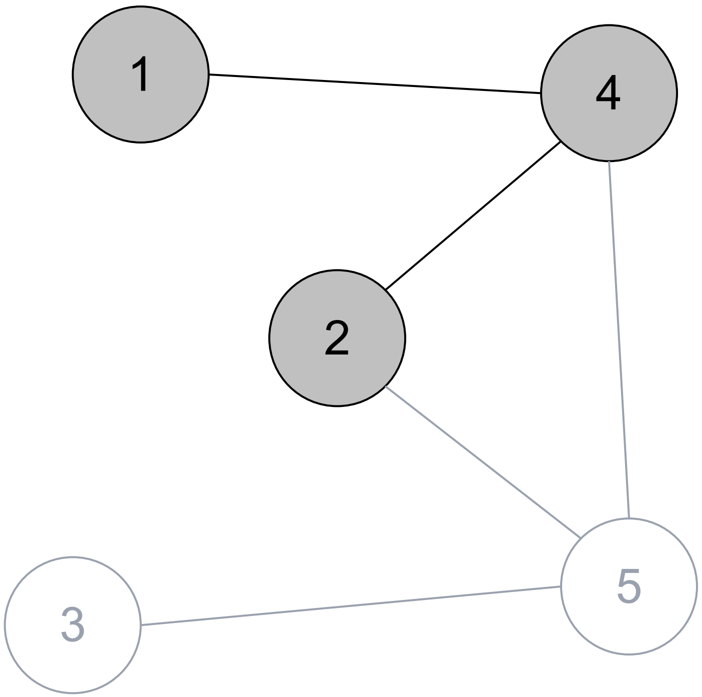
- Mark the vertex 4 as gray.

Depth-First Search



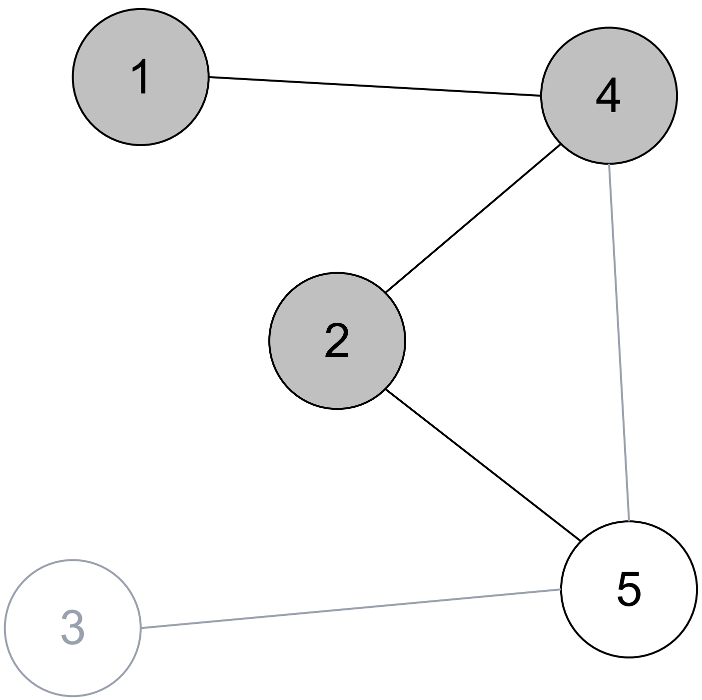
- There is an edge (4, 2) and vertex a 2 is unvisited.
- Go there.

Depth-First Search



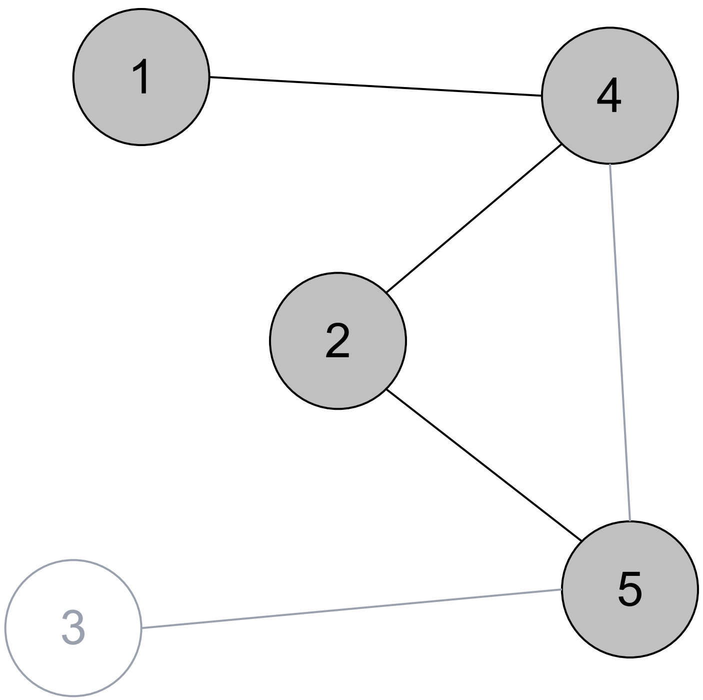
- Mark the vertex 2 as gray.

Depth-First Search



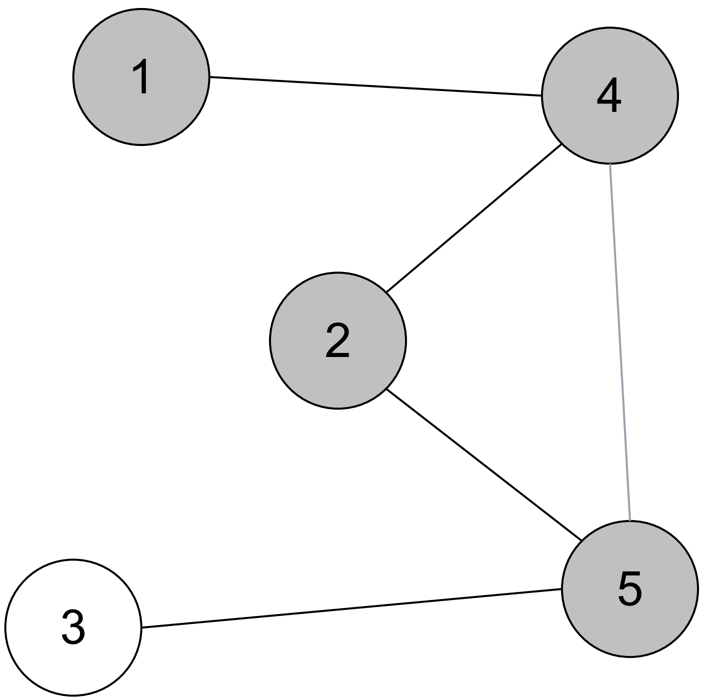
- There is an edge (2, 5) and a vertex 5 is unvisited.
- Go there.

Depth-First Search



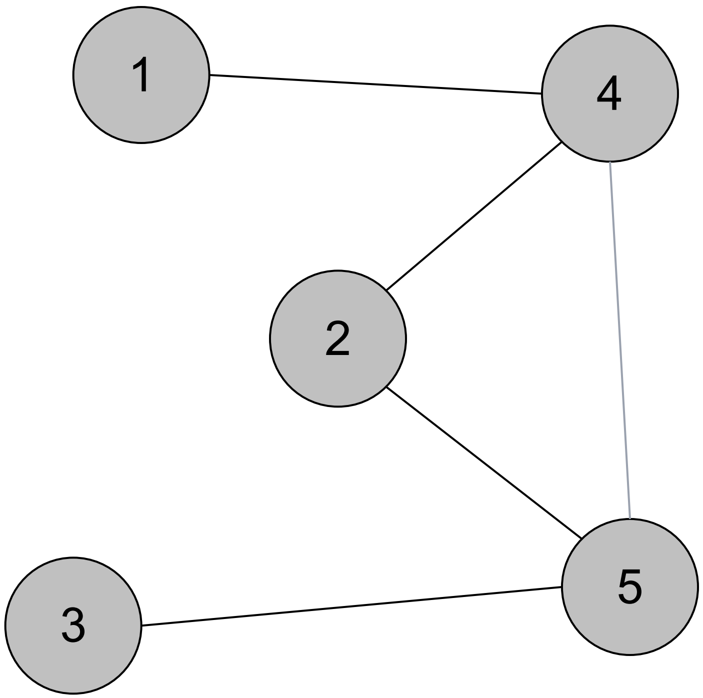
- Mark the vertex 5 as gray.

Depth-First Search



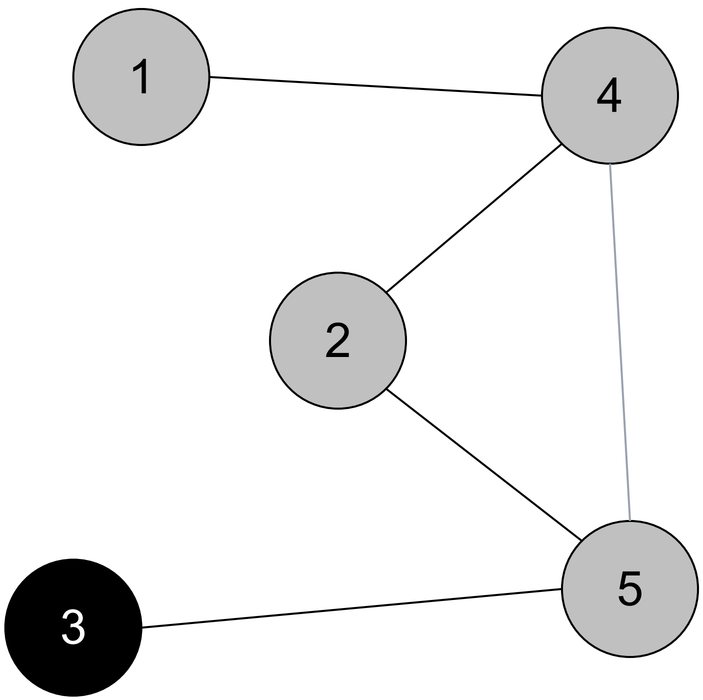
- There is an edge (5, 3) and a vertex 3 is unvisited.
- Go there.

Depth-First Search



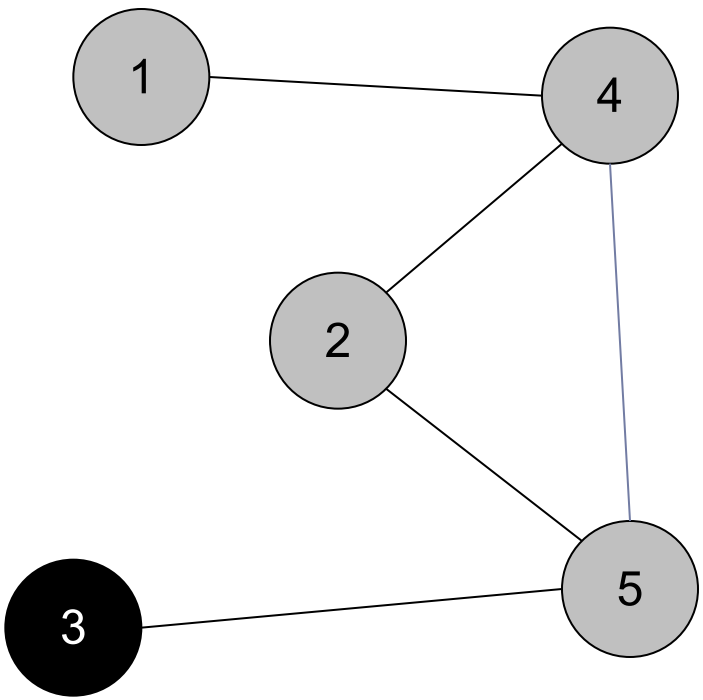
- Mark the vertex 3 as gray.

Depth-First Search



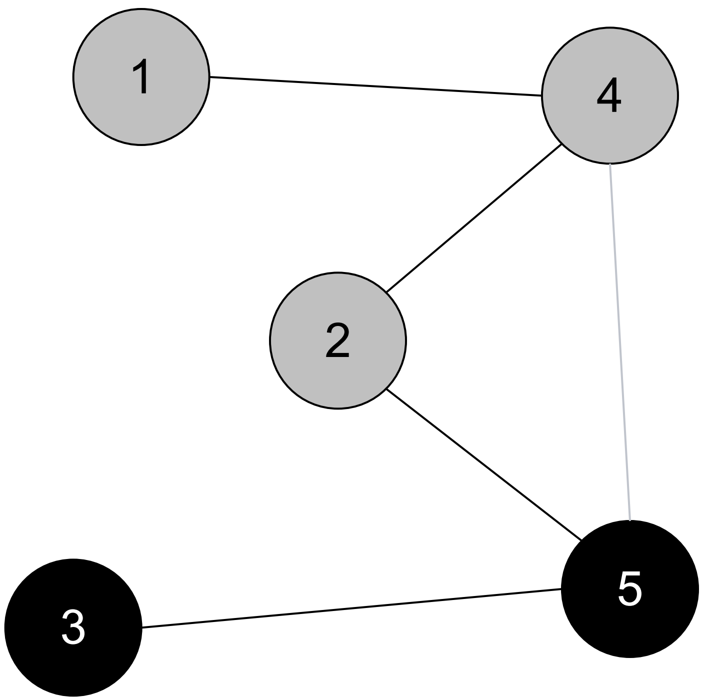
- There are no ways to go from the vertex **3**.
- Mark it as black and backtrack to the vertex **5**.

Depth-First Search



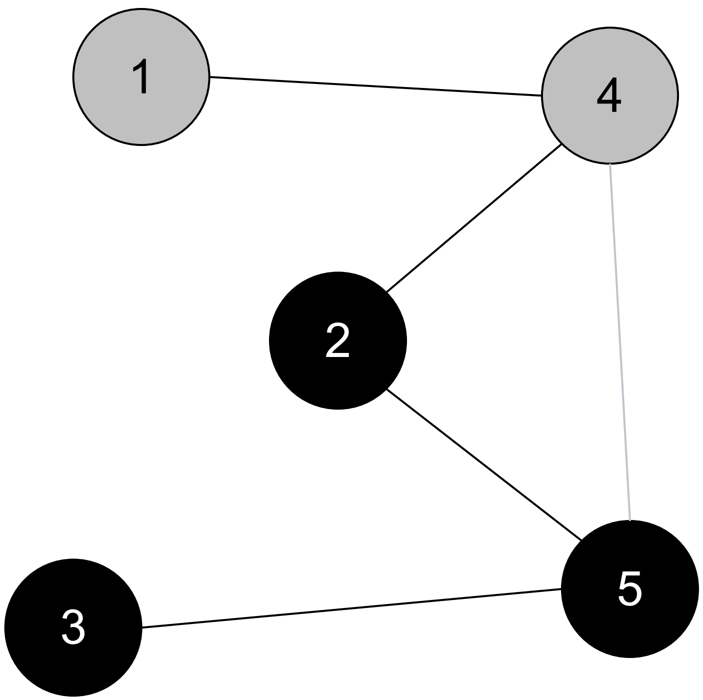
- There is an edge **(5, 4)**, but the vertex 4 is gray.

Depth-First Search



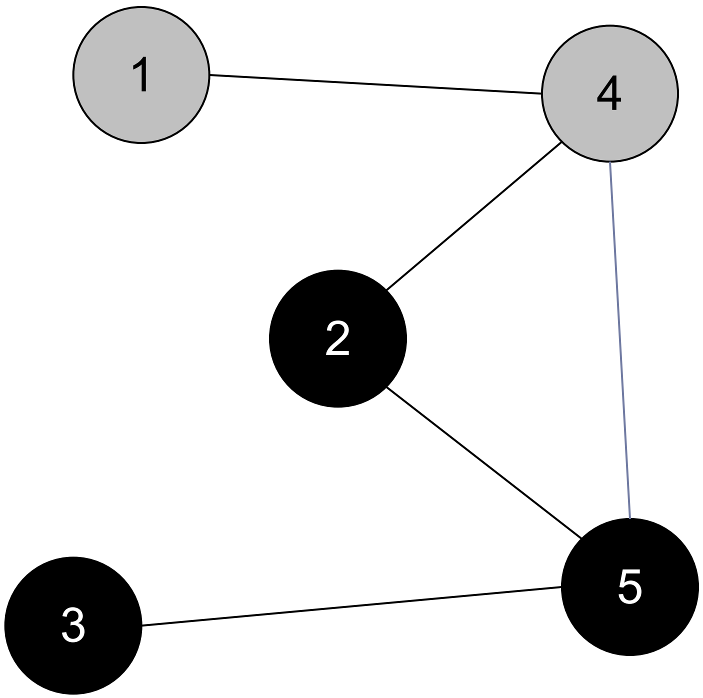
- There are no ways to go from the vertex **5.**
- Mark it as black and backtrack to the vertex **2**.

Depth-First Search



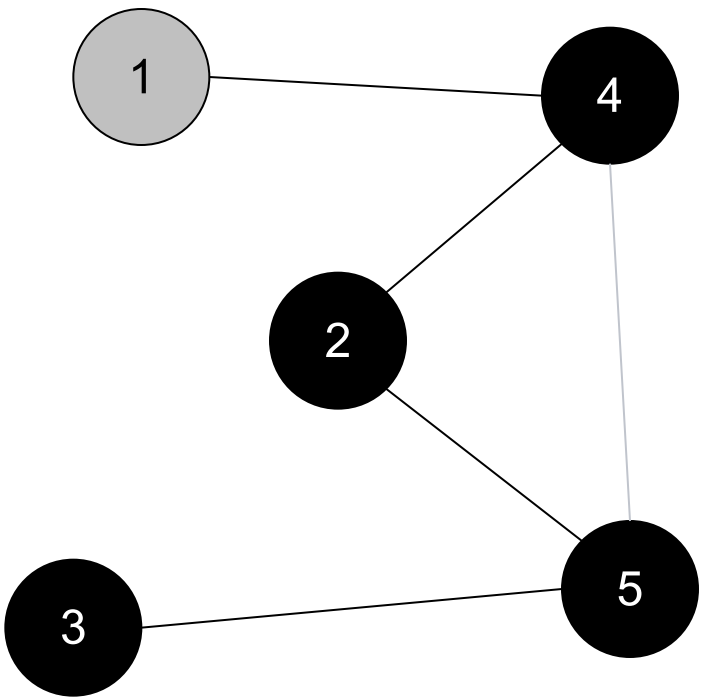
- There are no more edges, adjacent to vertex **2.**
- Mark it as black and backtrack to the vertex **4**.

Depth-First Search



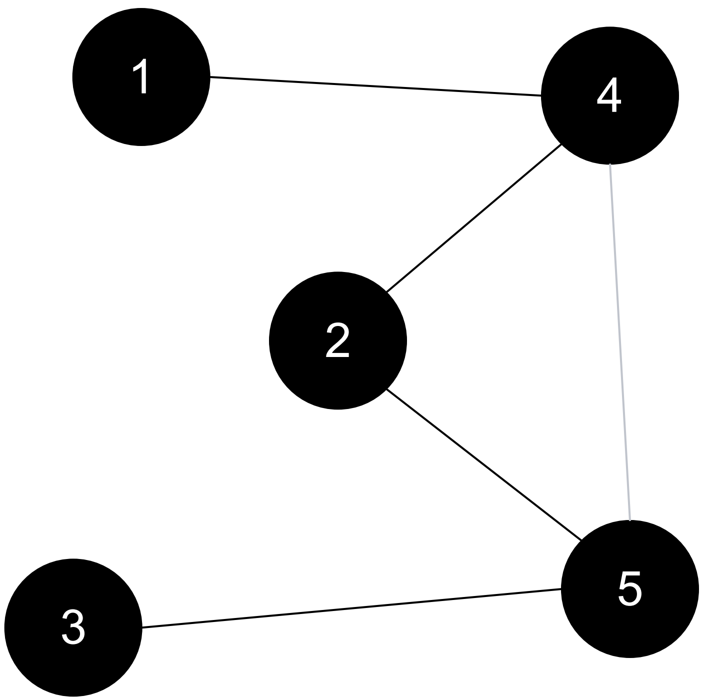
- There is an edge **(4, 5)**, but the vertex 5 is black.

Depth-First Search



- There are no more edges, adjacent to the vertex **4.**
- Mark it as black and backtrack to the vertex **1**.

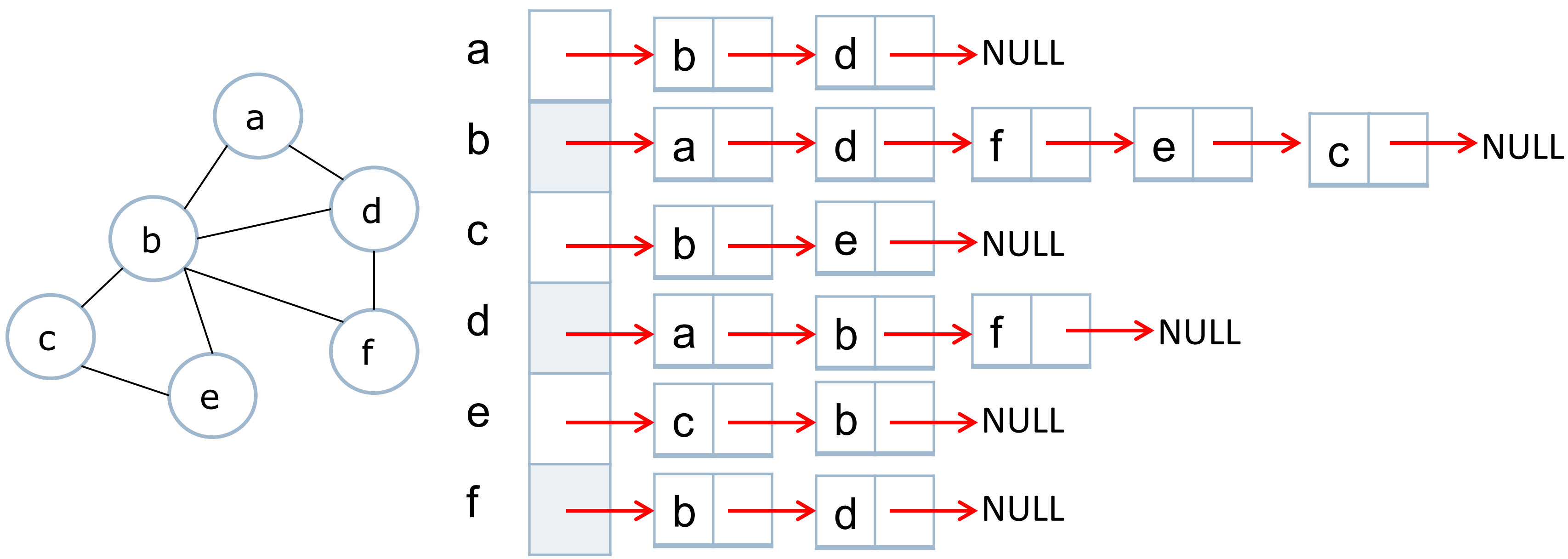
Depth-First Search



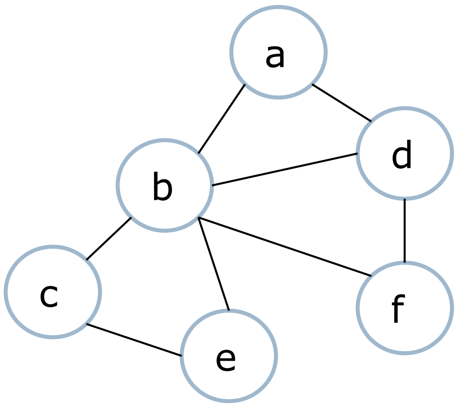
- There are no more edges, adjacent to the vertex **1**.
- Mark it as black.
- DFS is over.

Depth-First Search - Example

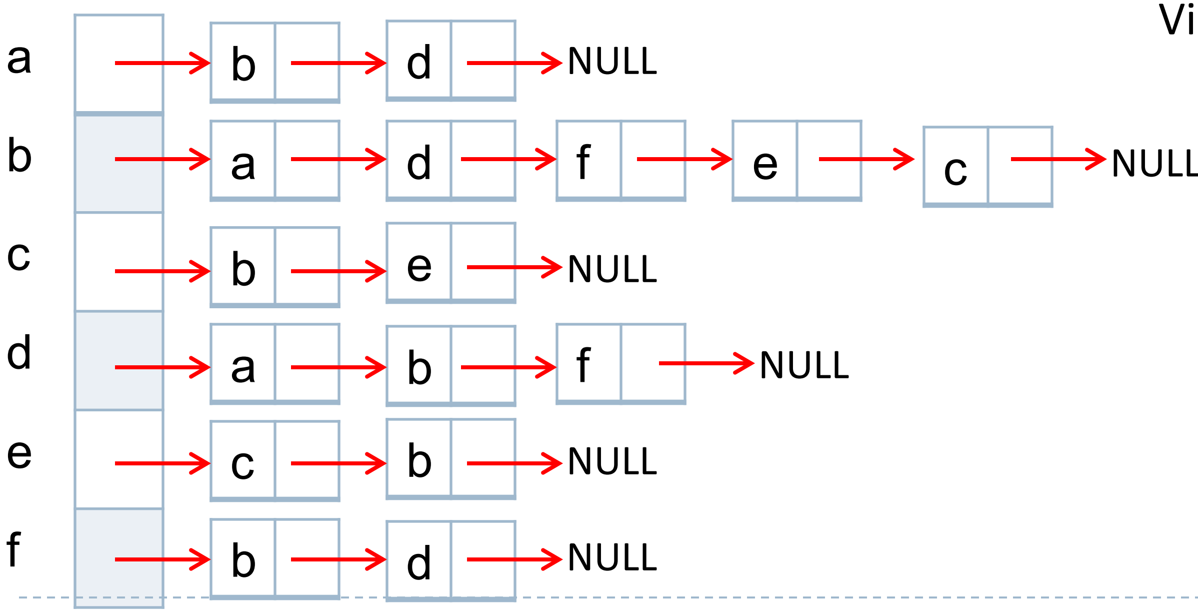
- Assume
  - Start from node a
  - Use adjacency list as graph representation



Depth-First Search – Example (Initial)



Order of visit:



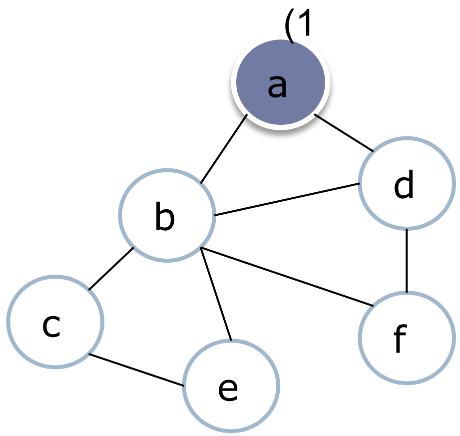
Visited Table

a	F
b	F
c	F
d	F
e	F
f	F

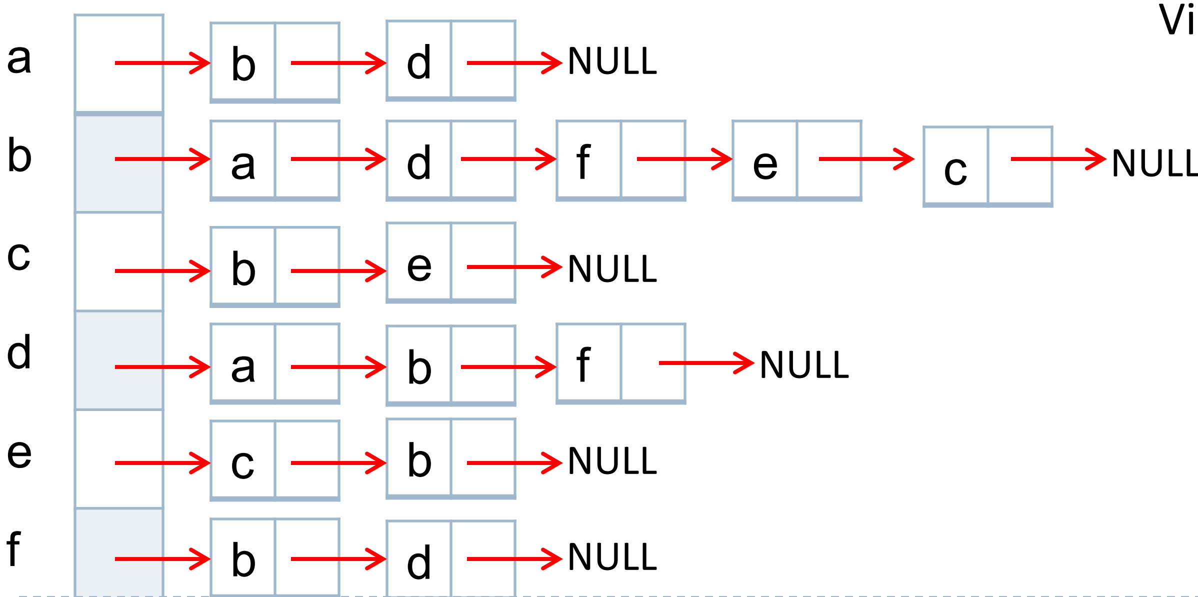
Predecessor table

a	NULL
b	NULL
c	NULL
d	NULL
e	NULL
f	NULL

Depth-First Search – Example (Step 1)



Order of visit:



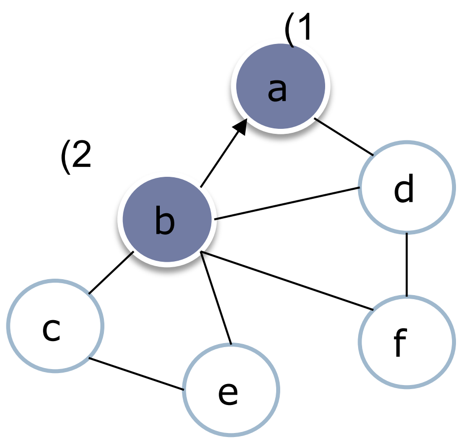
Visited Table

a	T
b	F
c	F
d	F
e	F
f	F

Predecessor table

a	NULL
b	NULL
c	NULL
d	NULL
e	NULL
f	NULL

Depth-First Search – Example (Step 2)



Order of visit:

a → b → d → NULL
b → a → d → f → e → c → NULL
c → b → e → NULL
d → a → b → f → NULL
e → c → b → NULL
f → b → d → NULL

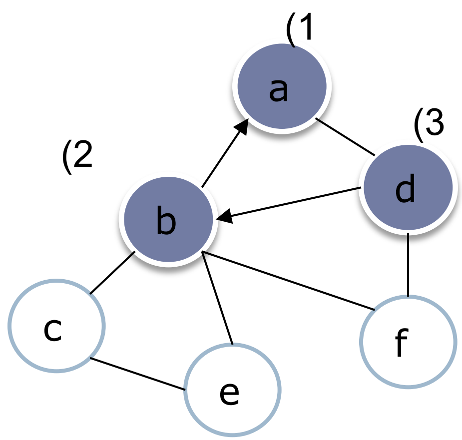
Visited Table

a	T
b	T
c	F
d	F
e	F
f	F

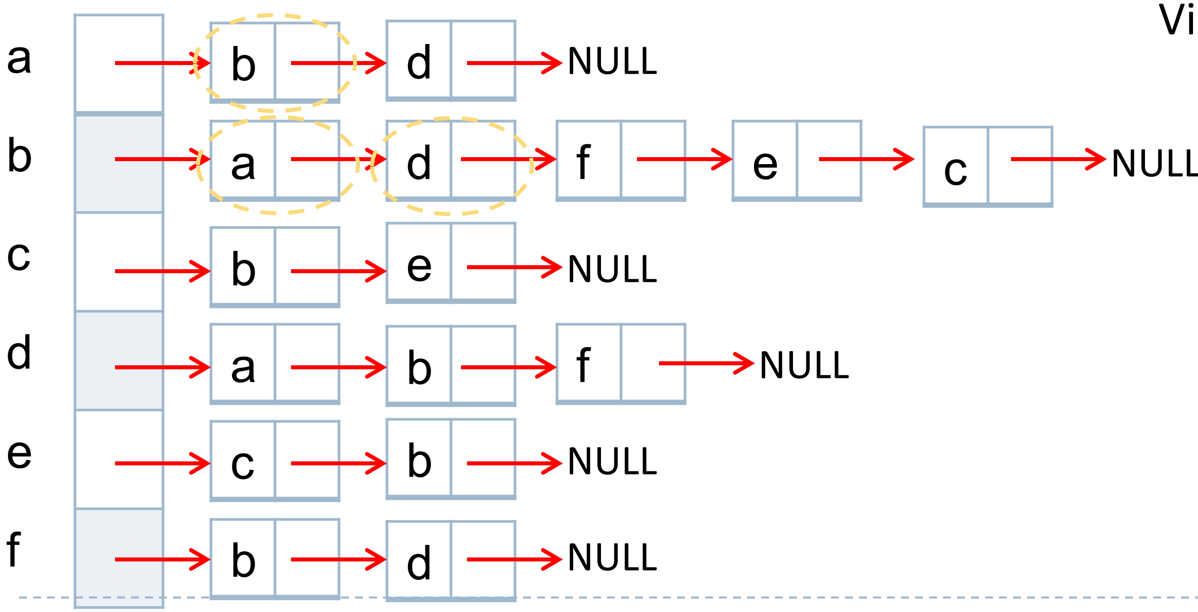
Predecessor table

a	NULL
b	a
c	NULL
d	NULL
e	NULL
f	NULL

Depth-First Search – Example (Step 3)



Order of visit:



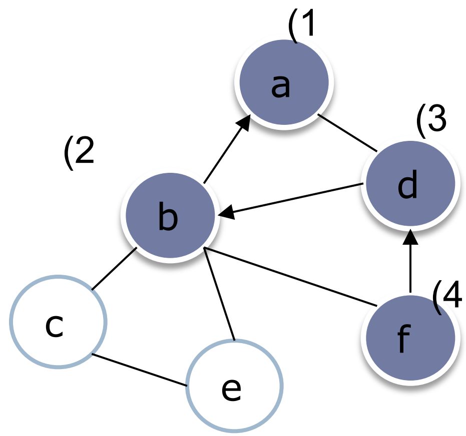
Visited Table

a	T
b	T
c	F
d	T
e	F
f	F

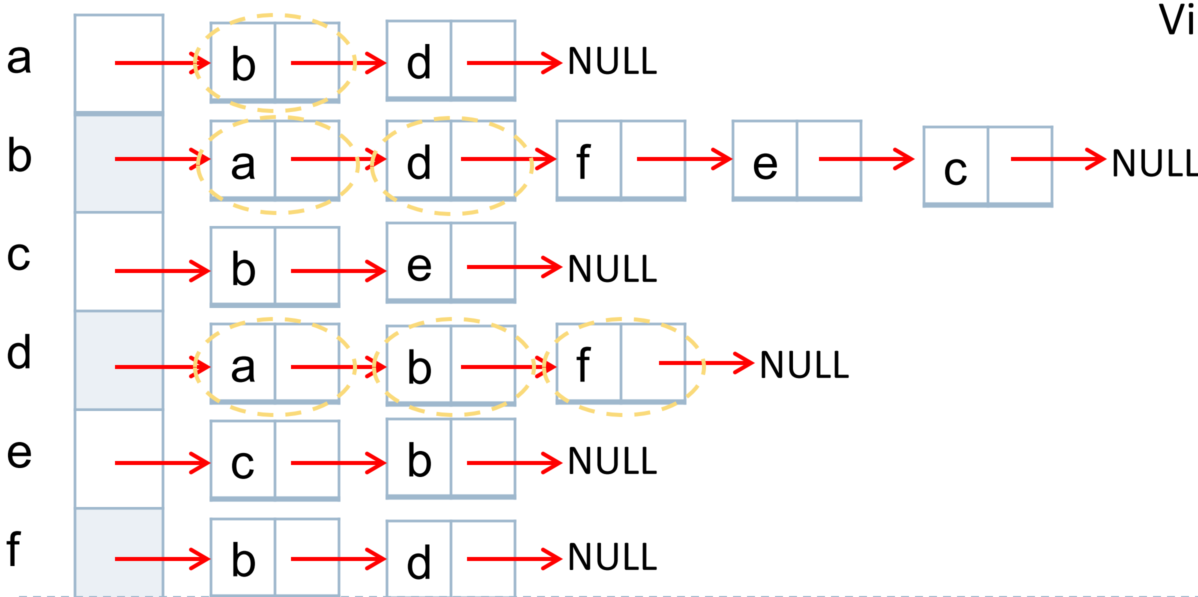
Predecessor table

a	NULL
b	a
c	NULL
d	b
e	NULL
f	NULL

Depth-First Search – Example (Step 4)



Order of visit:



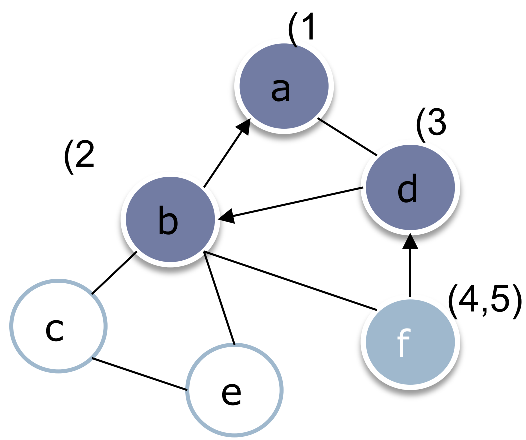
Visited Table

a	T
b	T
c	F
d	T
e	F
f	T

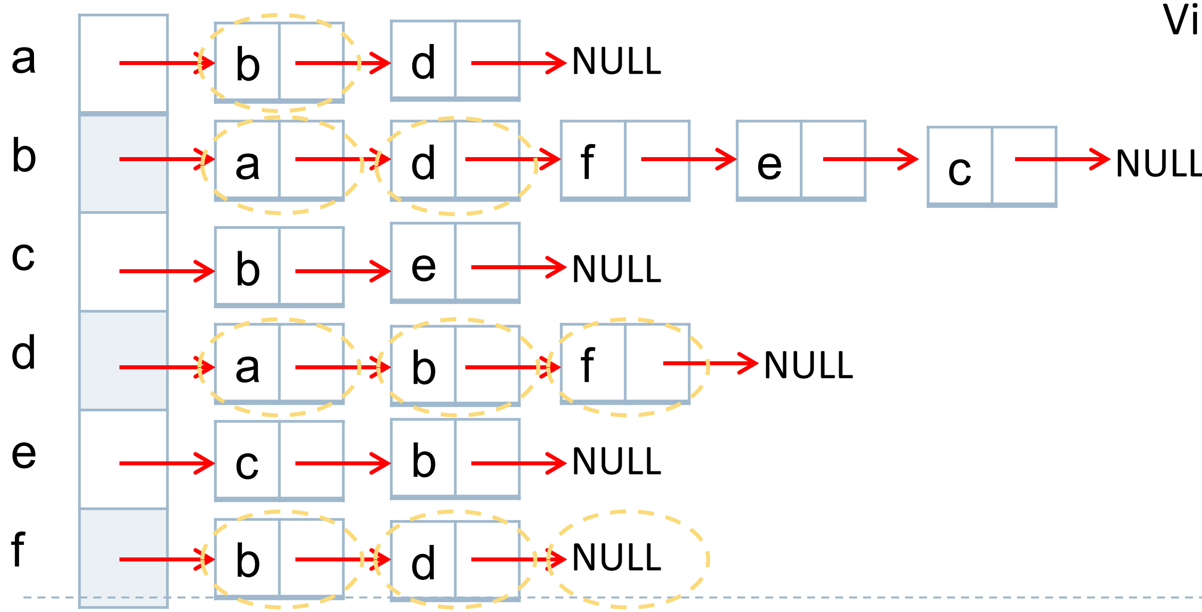
Predecessor table

a	NULL
b	a
c	NULL
d	b
e	NULL
f	d

Depth-First Search – Example (Step 5)



Order of visit: f



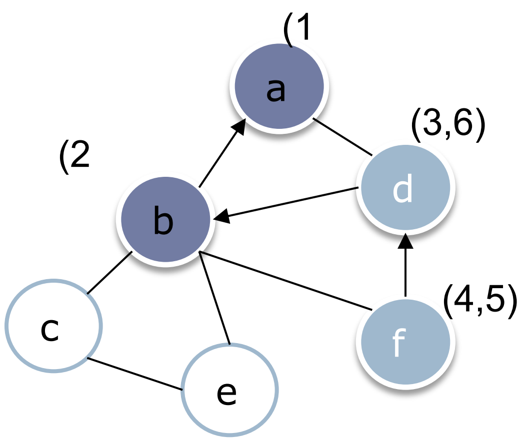
Visited Table

a	T
b	T
c	F
d	T
e	F
f	T

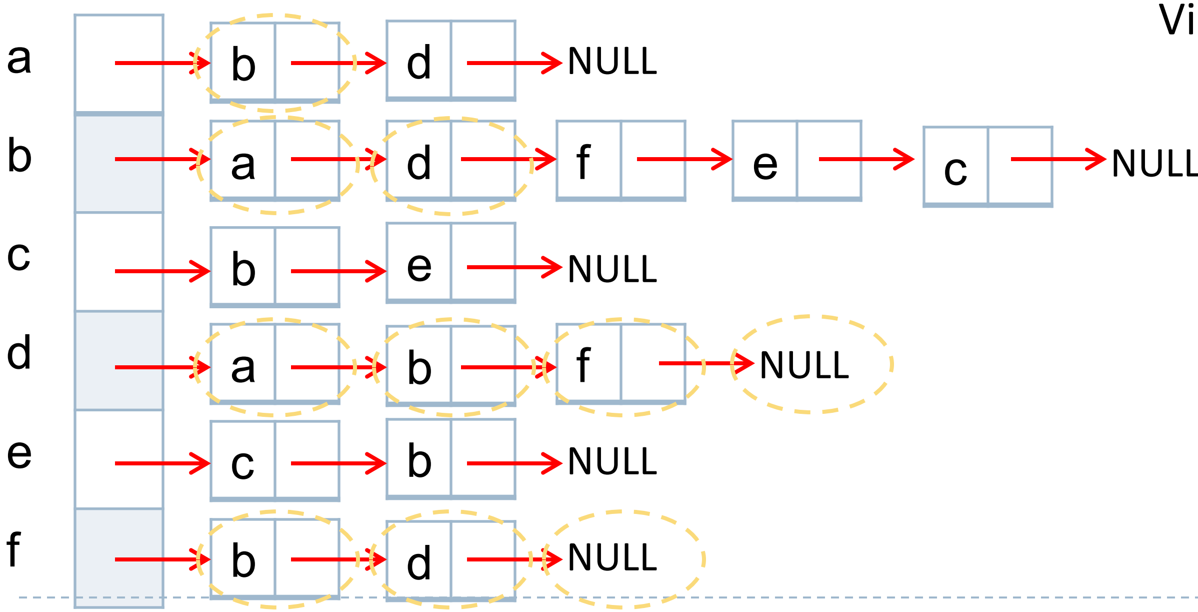
Predecessor table

a	NULL
b	a
c	NULL
d	b
e	NULL
f	d

Depth-First Search – Example (Step 6)



Order of visit: f, d



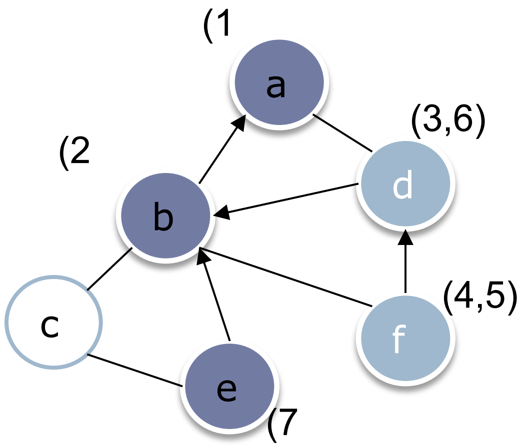
Visited Table

a	T
b	T
c	F
d	T
e	F
f	T

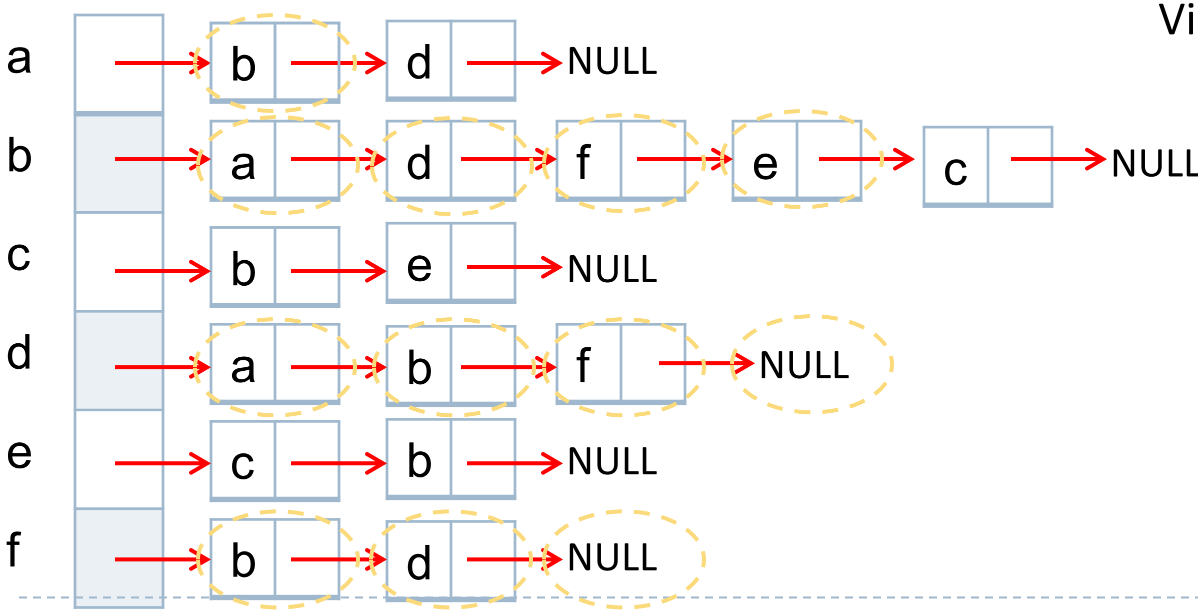
Predecessor table

a	NULL
b	a
c	NULL
d	b
e	NULL
f	d

Depth-First Search – Example (Step 7)



Order of visit: f, d



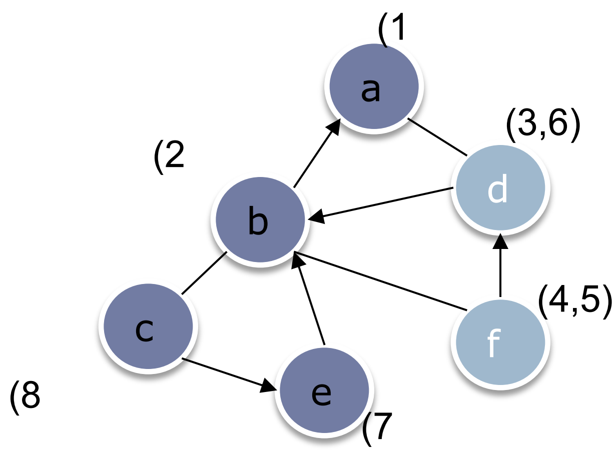
Visited Table

a	T
b	T
c	F
d	T
e	T
f	T

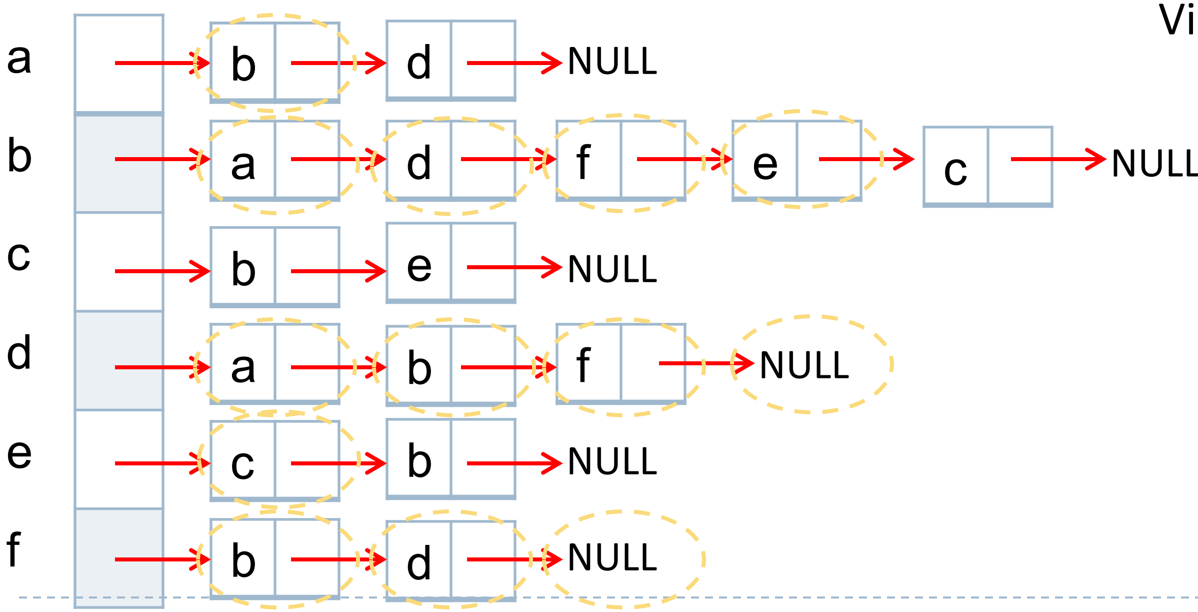
Predecessor table

a	NULL
b	a
c	NULL
d	b
e	b
f	d

Depth-First Search – Example (Step 8)



Order of visit: f, d



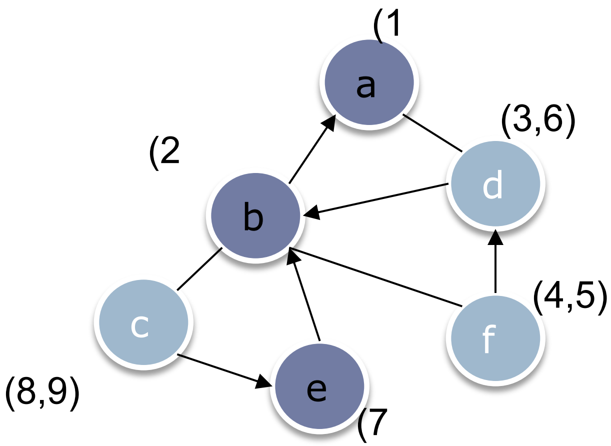
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

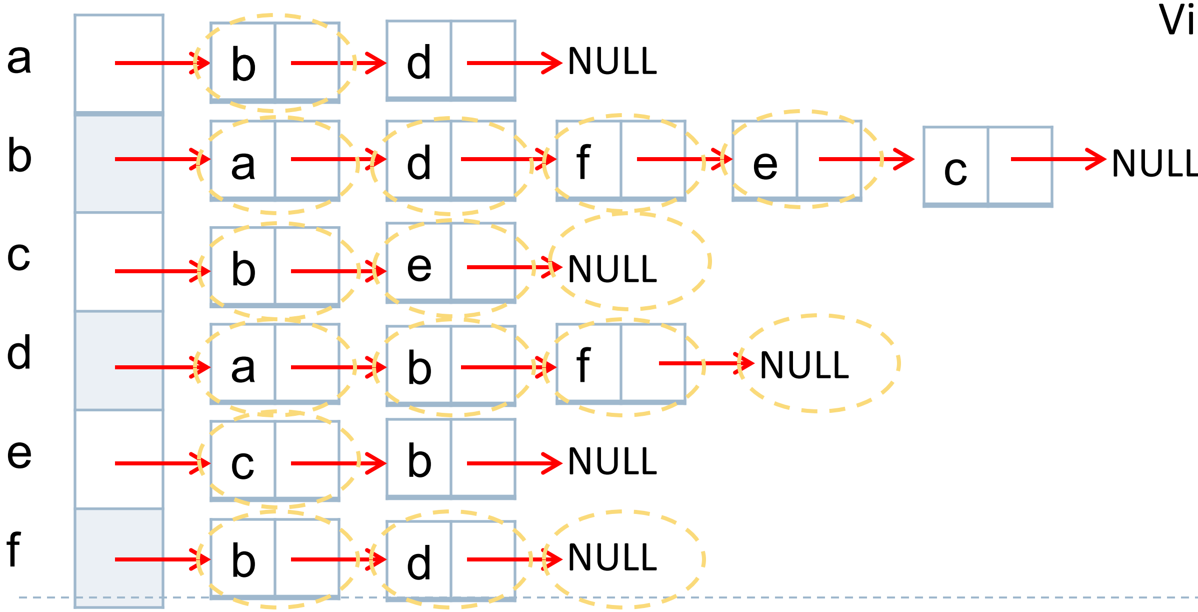
Predecessor table

a	NULL
b	a
c	e
d	b
e	b
f	d

Depth-First Search – Example (Step 9)



Order of visit: f, d, c



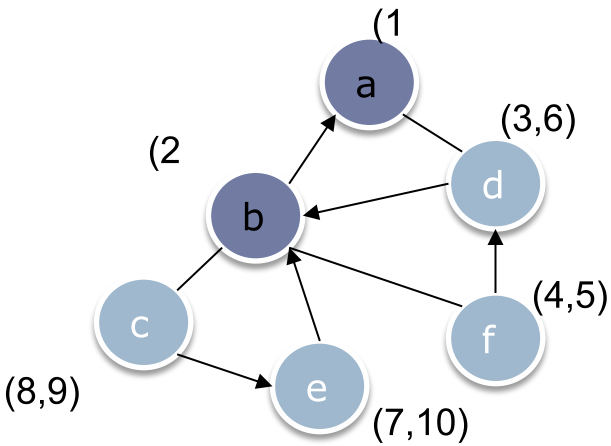
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

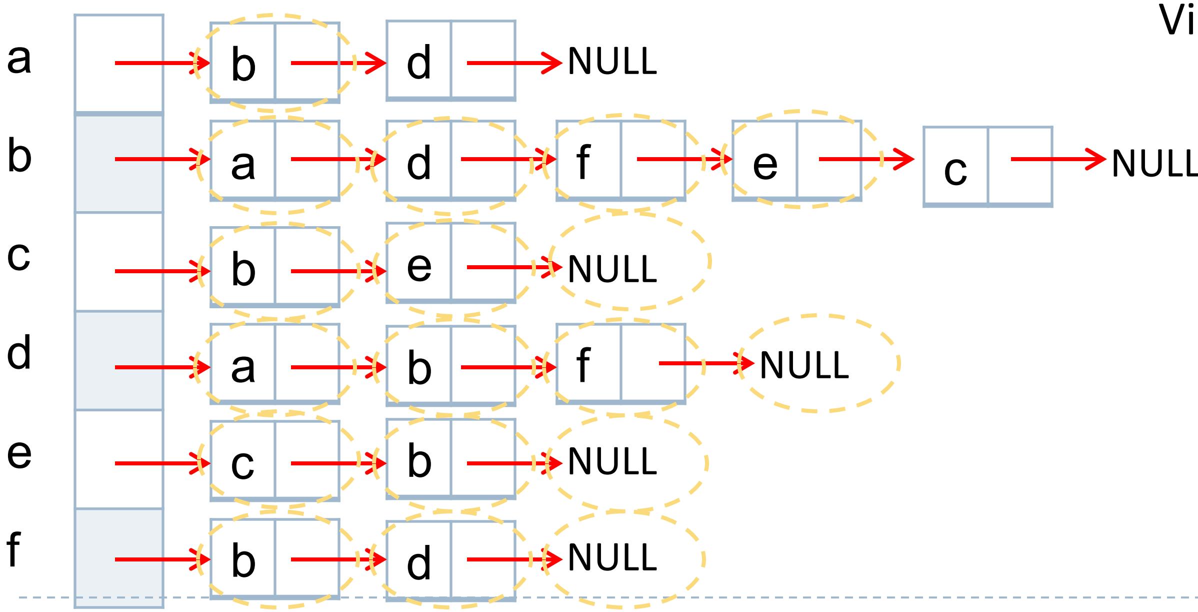
Predecessor table

a	NULL
b	a
c	e
d	b
e	b
f	d

Depth-First Search – Example (Step 10)



Order of visit: f, d, c, e



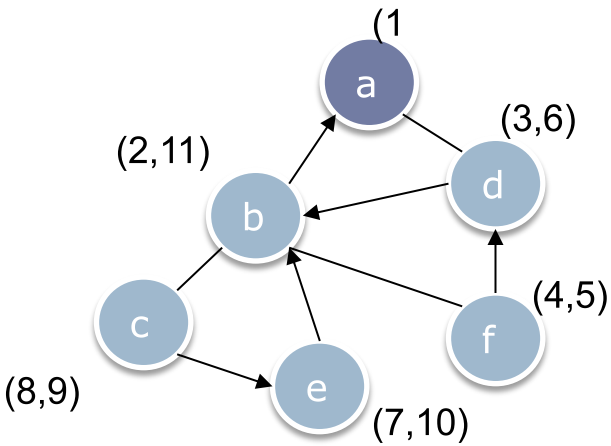
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

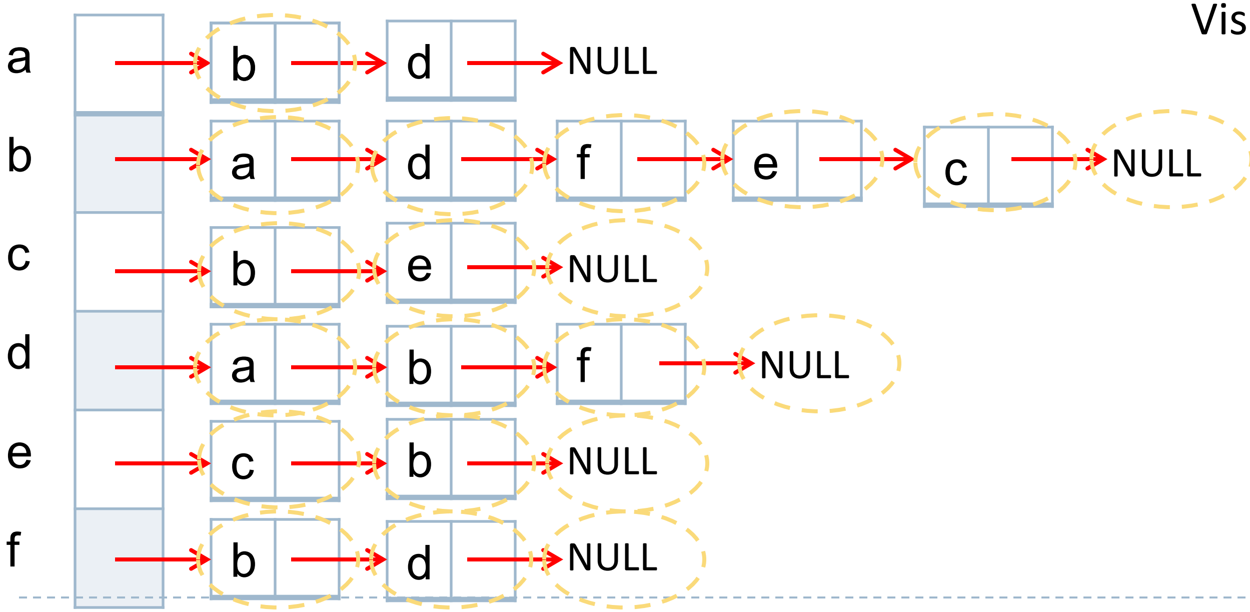
Predecessor table

a	NULL
b	a
c	e
d	b
e	b
f	d

Depth-First Search – Example (Step 11)



Order of visit: f, d, c, e, b



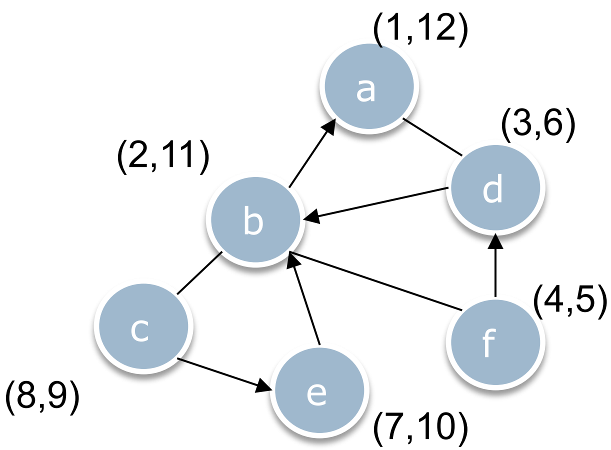
Visited Table

a	T
b	T
c	T
d	T
e	T
f	T

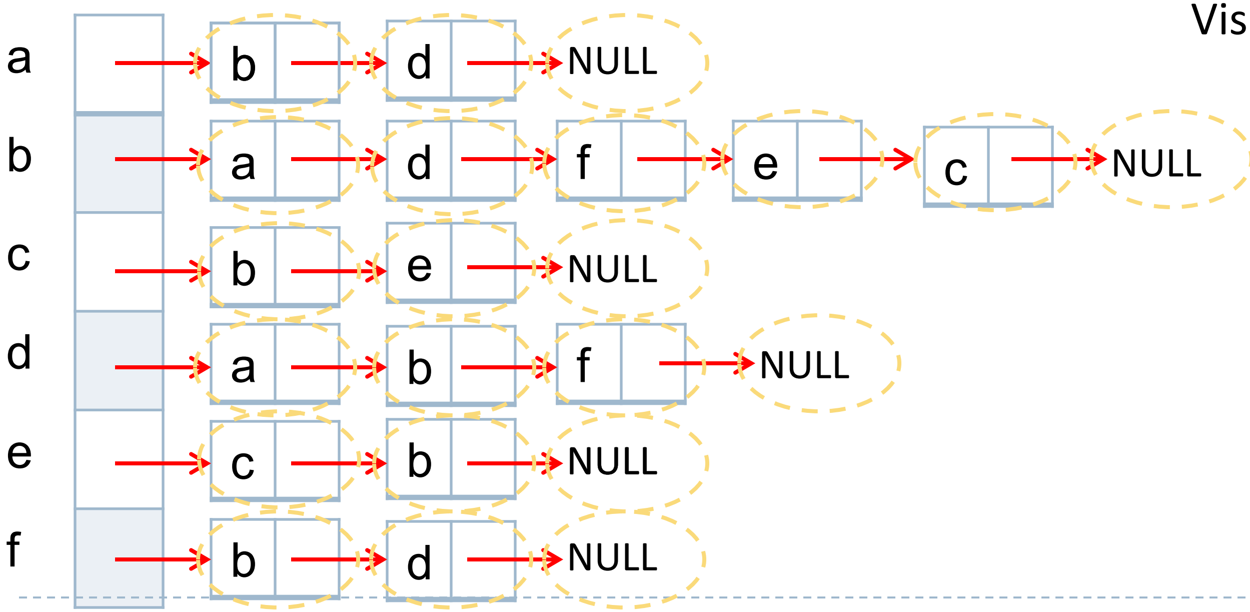
Predecessor table

a	NULL
b	a
c	e
d	b
e	b
f	d

Depth-First Search – Example (Step 12)



Order of visit: f, d, c, e, b, a



Visited Table

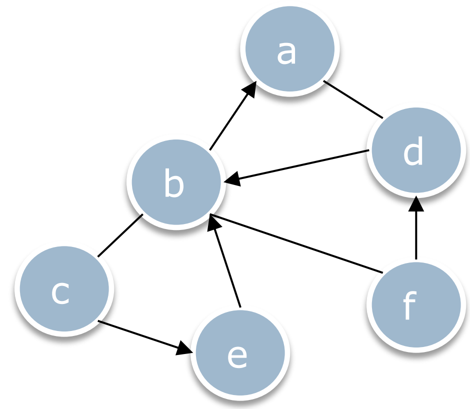
a	T
b	T
c	T
d	T
e	T
f	T

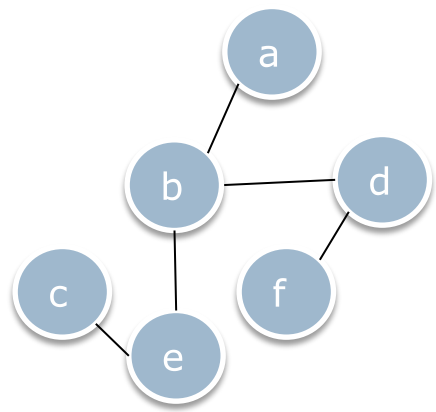
Predecessor table

a	NULL
b	a
c	e
d	b
e	b
f	d

Paths from Source to Each Node

- Algorithm:
  Path(w)
  - If predecessor[w] is not NULL
    Path(predecessor[w])
  - Output w





DFS Tree

Predecessor table

a	NULL
b	a
c	e
d	b
e	b
f	d

Depth-First Search

- As you can see from the example, DFS doesn't go through all edges.

- The vertices and edges, which depth-first search has visited is a **tree**.

  - This tree contains all vertices of the graph (if it is connected) and is called ***graph spanning tree***.

CHAPTER 8 END