CSE 105:
Data Structures and Algorithms-I
(Part 2)

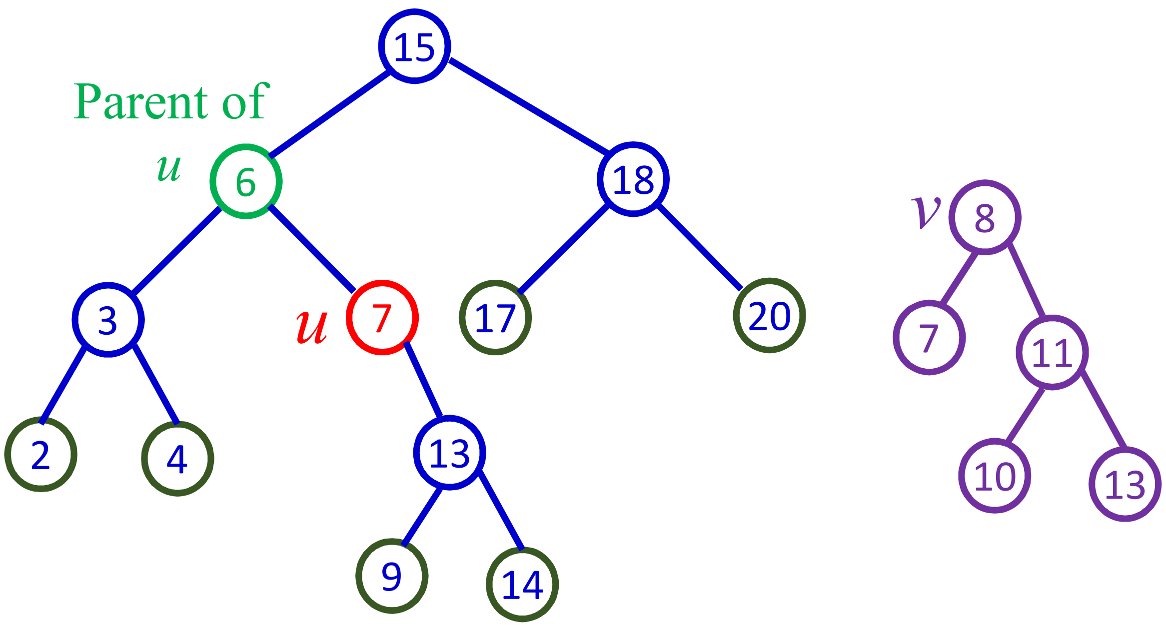
Instructor

Dr Md Monirul Islam

BST Operation: Deletion (2)

Review

This algorithm replaces node *u* by node *v*



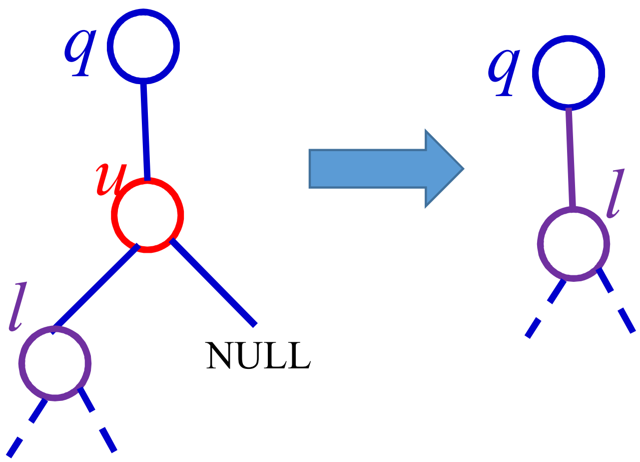

```
TRANSPLANT(T, u, v)
if u->parent == NULL          //special case
      T->root = v
elseif u = = u->parent->left  //set appropriate child
        u->parent->left = v
else u->parent->right =v
if v ≠ NULL                   //set parent
    v->parent = u->parent
```

BST Operation: Deletion (2)

Review

Node Deletion Cases

Node u has **NO RIGHT** child

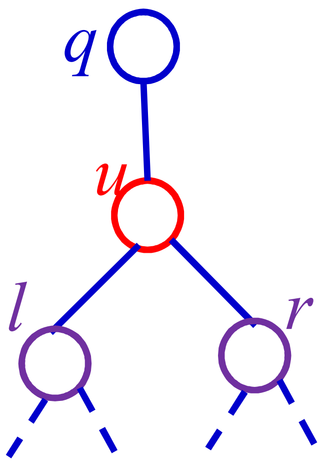



```
TREE_DELETE (T, u)
if u->left == NULL
    TRANSPLANT(T, u, u->right)
elseif  u->right == NULL
    TRANSPLANT (T, u, u->left)
```

BST Operation: Deletion (2)

Node Deletion Cases

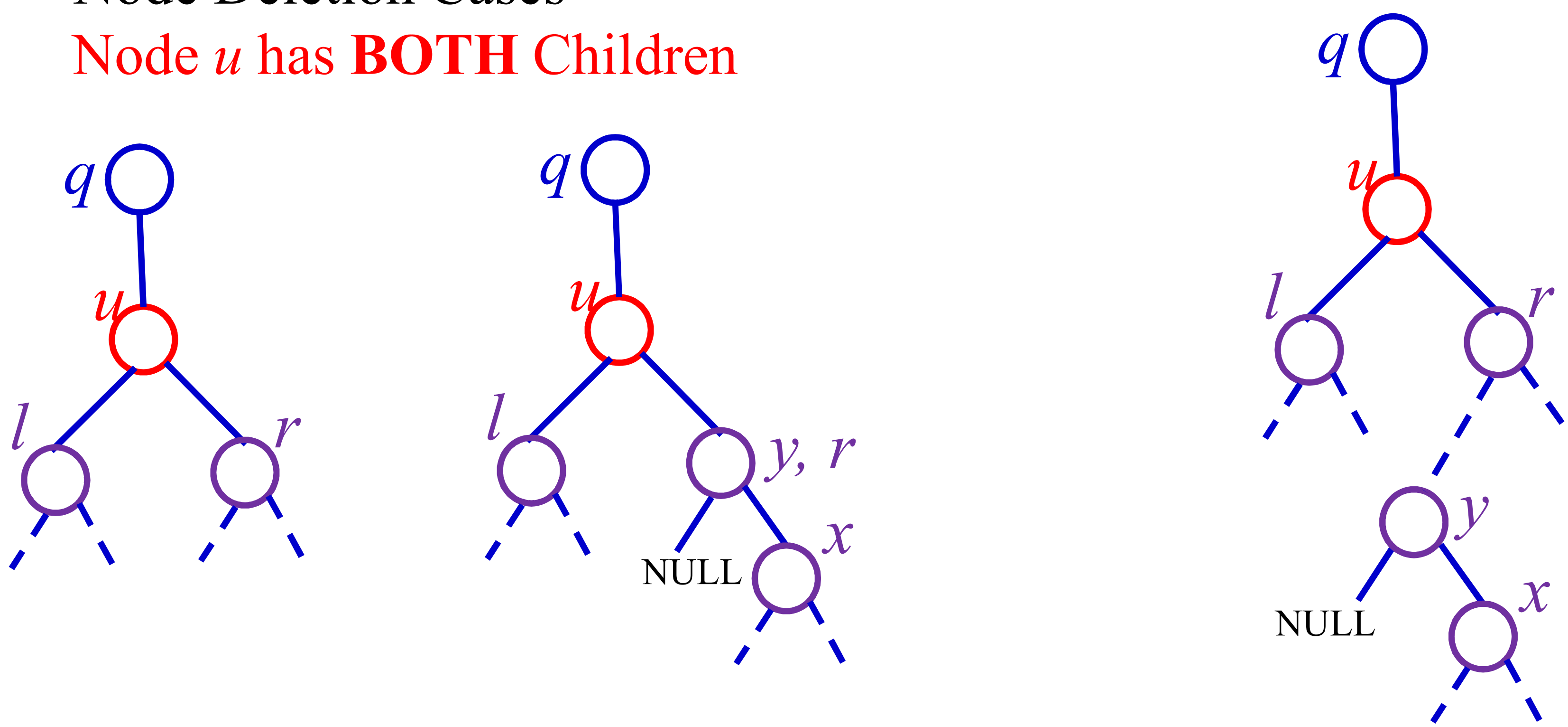
Node u has **BOTH** Children



BST Operation: Deletion (2)

Node Deletion Cases

Node u has **BOTH** Children

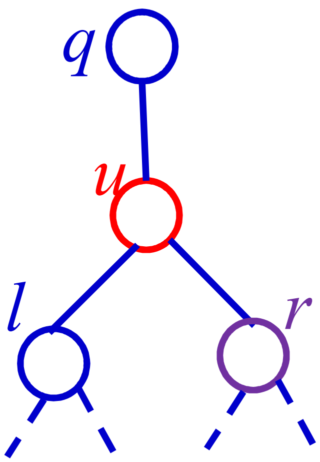


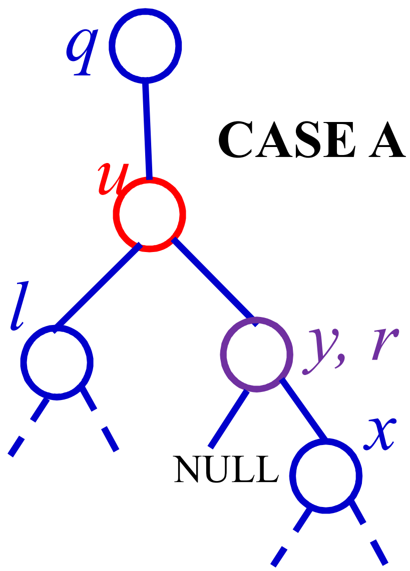
Find y = successor (next minimum) from RIGHT subtree

BST Operation: Deletion (2)

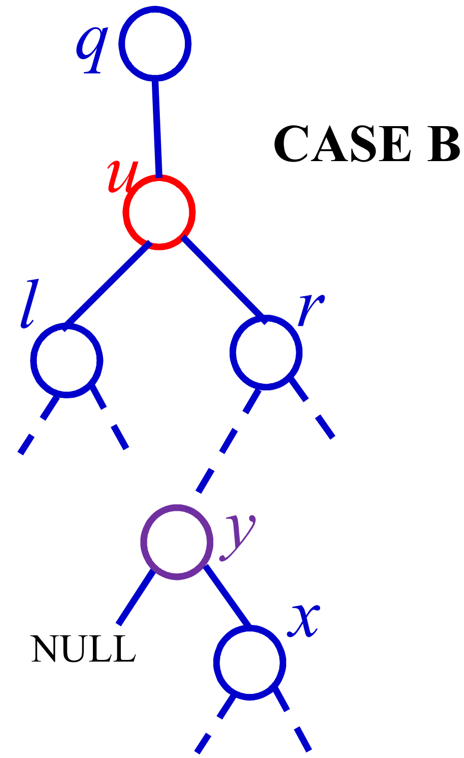
Node Deletion Cases

Node u has **BOTH** Children





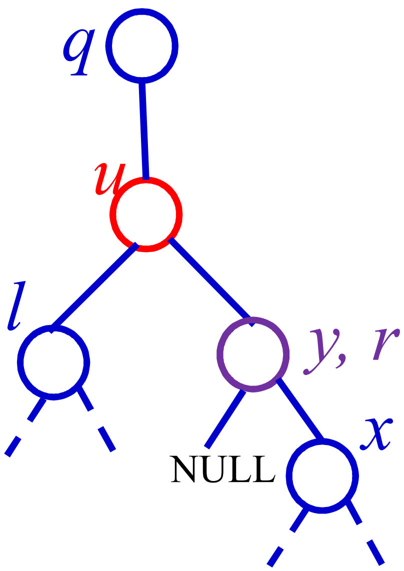
y is immediate RIGHT child of u



y is NOT immediate child of u

BST Operation: Deletion (2)

CASE A



y is immediate RIGHT child of u

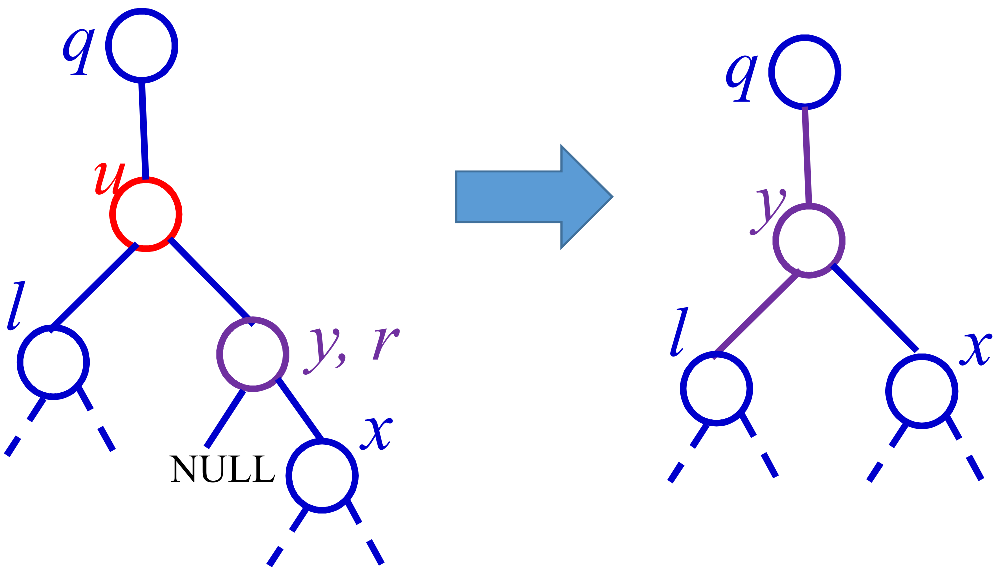
```
TREE_DELETE (T, u)
if u->left == NULL
    TRANSPLANT(T, u, u->right)
elseif  u->right == NULL
    TRANSPLANT (T, u, u->left)
else y = TREE_MINIMUM(u->right)
```

when y->$parent$ == u

```
    TRANSPLANT(T, u, y)
    y->left = u->left
    y->left->parent = y
```

BST Operation: Deletion (2)

CASE A



y is immediate RIGHT child of *u*

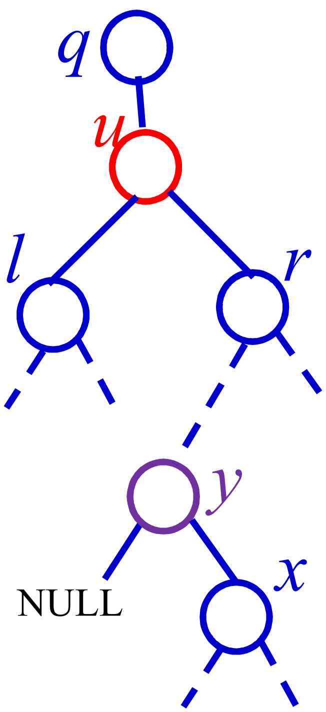
TREE_DELETE (*T, u*)
1 **if** *u->left* == NULL
2 TRANSPLANT(*T*, *u*, *u->right*)
3 **elseif** *u->right* == NULL
4 TRANSPLANT (*T*, *u*, *u->left*)
5 **else** *y* = TREE_MINIMUM(*u->right*)

when *y->parent* == *u*

10 TRANSPLANT(*T, u, y*)
11 *y->left* = *u->left*
12 *y->left->parent* = *y*

BST Operation: Deletion (2)

CASE B

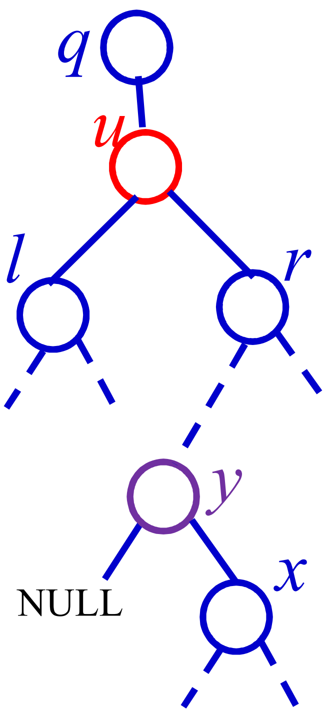


y is **NOT** immediate child of u

BST Operation: Deletion (2)

CASE B

$$y.key < \cdots < x.key \cdots < r.key$$

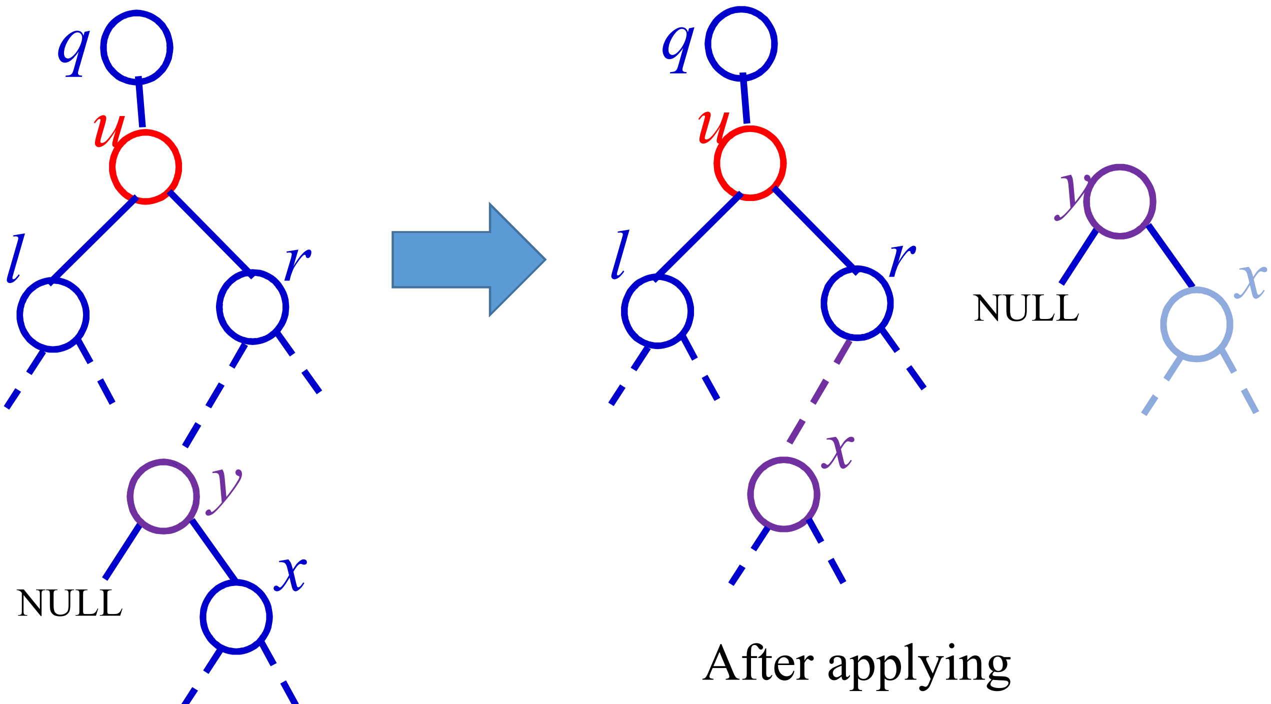


y is **NOT** immediate child of u

BST Operation: Deletion (2)

CASE B

$$y.key < \cdots < x.key \cdots < r.key$$

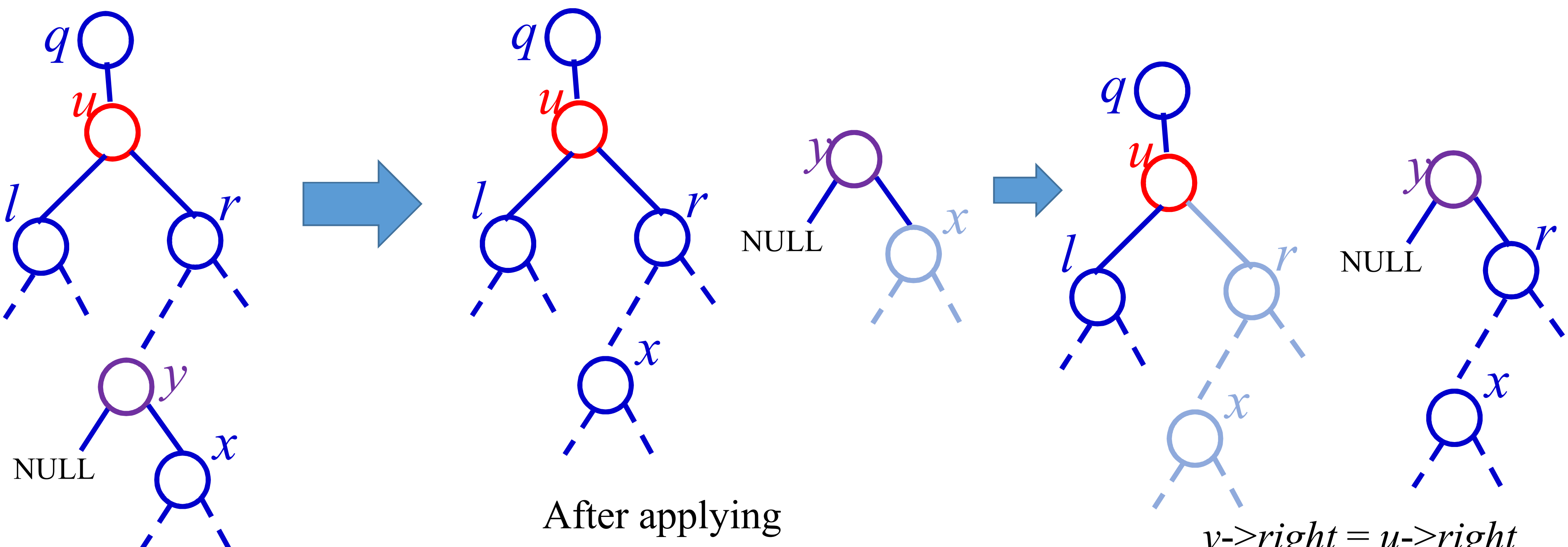


After applying
TRANSPLANT(T, y, y->$right$)

BST Operation: Deletion (2)

CASE B

$$y.key < \cdots < x.key \cdots < r.key$$

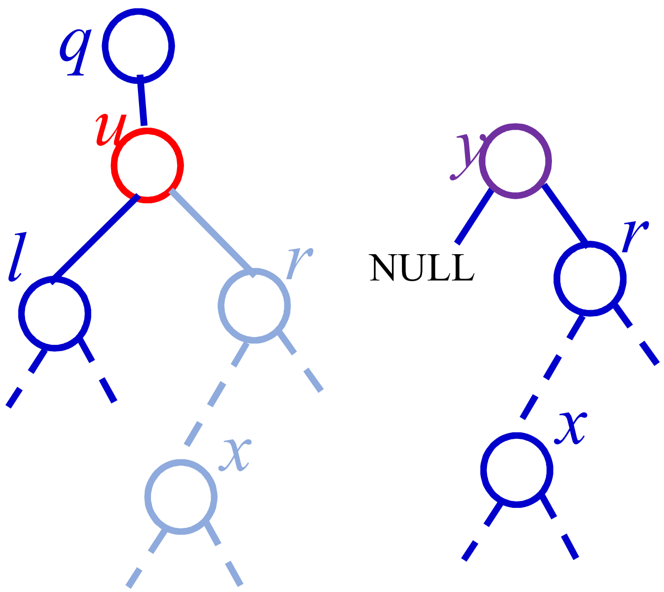


After applying
TRANSPLANT(T, y, y->$right$)

y->$right$ = u->$right$
y->$right$->$parent$= y

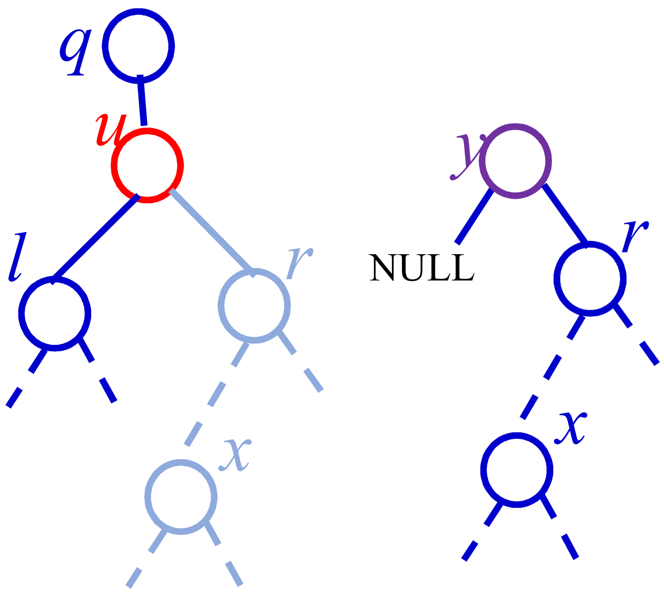
BST Operation: Deletion (2)

CASE B

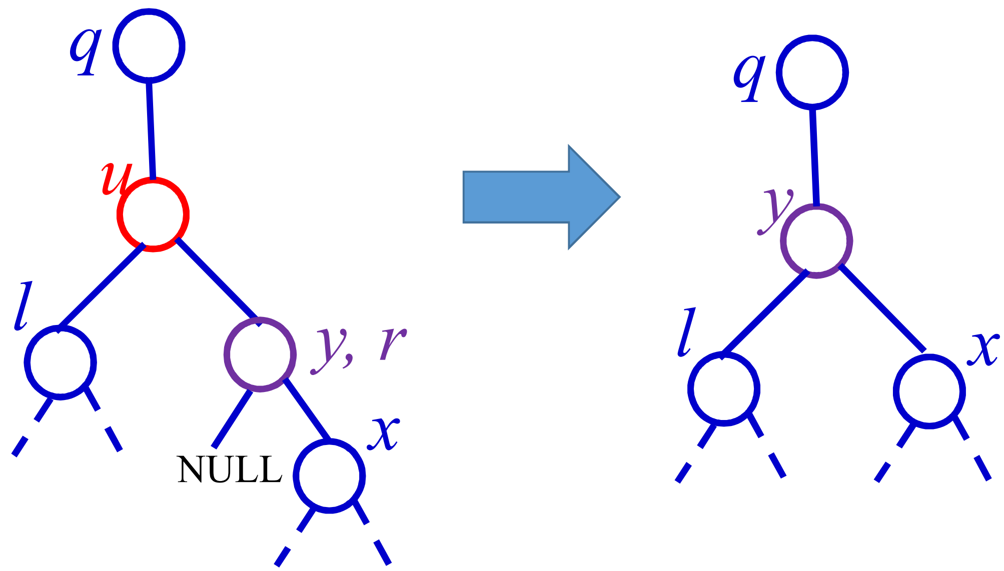


BST Operation: Deletion (2)

CASE B
Outcome



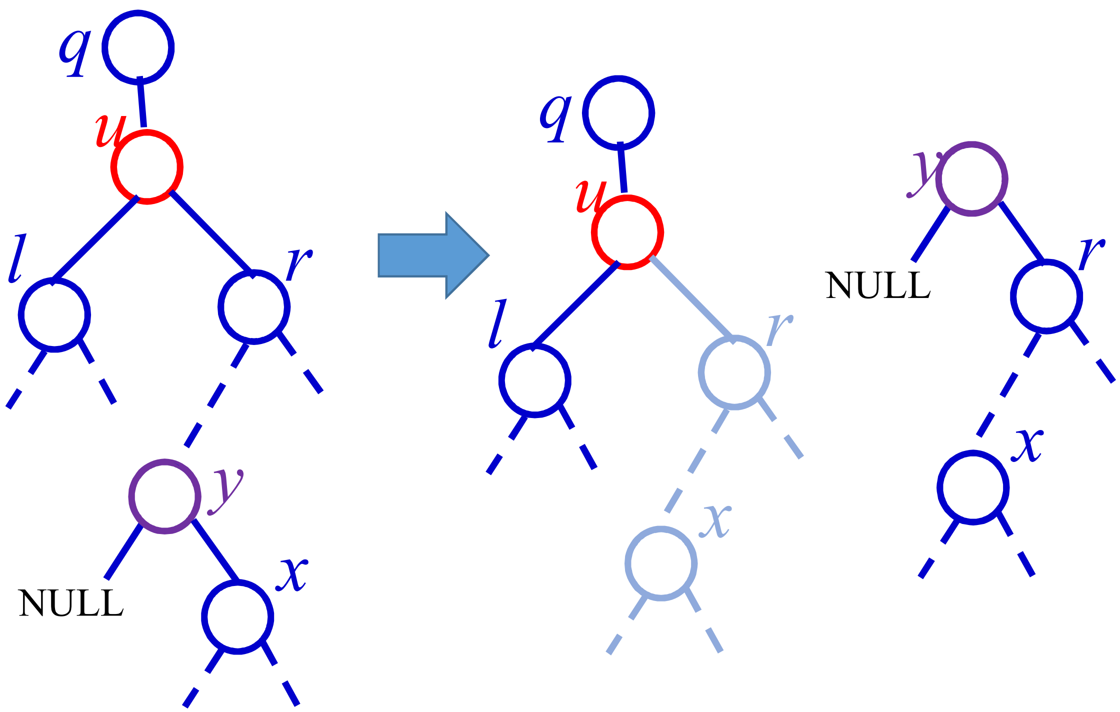
CASE A



y is immediate RIGHT child of *u*

BST Operation: Deletion (2)

CASE B



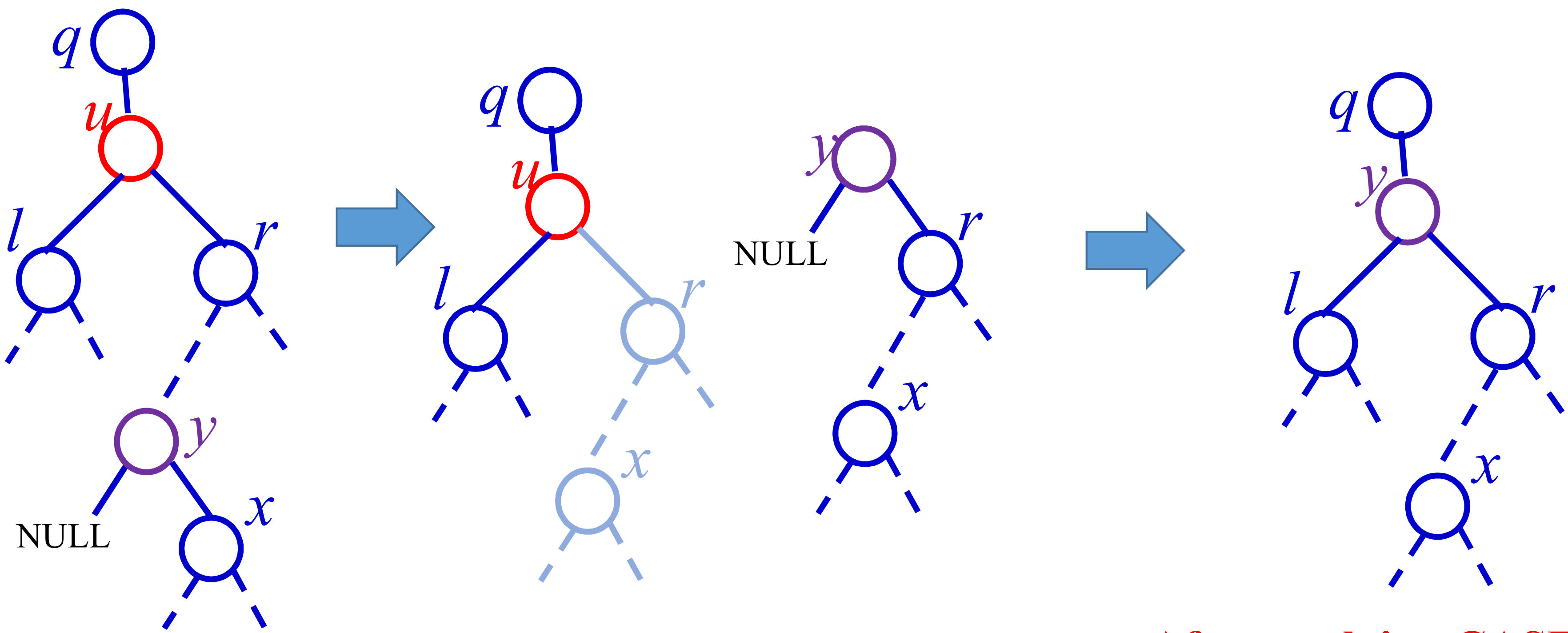
Starting state

Converted to CASE A

```
TREE_DELETE (T, u)
if u->left == NULL
    TRANSPLANT(T, u, u->right)
elseif   u->right == NULL
    TRANSPLANT (T, u, u->left)
else y = TREE_MINIMUM(u->right)
    if y->parent ≠u
        TRANSPLANT(T, y, y->right)
        y->right = u->right
        y->right->parent= y
    TRANSPLANT(T, u, y)
    y->left = u->left
    y->left->parent = y
```

BST Operation: Deletion (2)

CASE B



Starting state

Converted to CASE A

After applying CASE A Code

Back to Graph

Adjacency Matrix Representation

Review

- Pros:
  - Simple to implement
  - Easy and fast to tell if a pair (i, j) is an edge: simply check if A[i, j] is 1 or 0
  - Can be very efficient for small graphs
  - Good for dense graphs (why?)

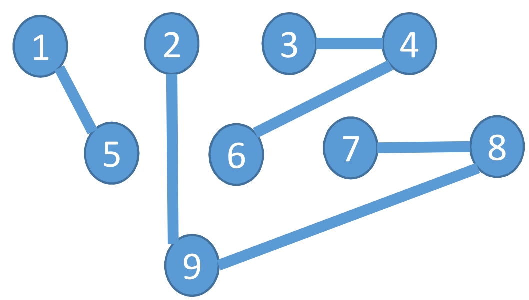
- Cons:
  - No matter how few edges the graph has, the matrix takes $O(n^2)$, i.e., $O(|V|^2)$ in memory

$|V| = n$

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

$|V| = n$

$n \times n$ matrix

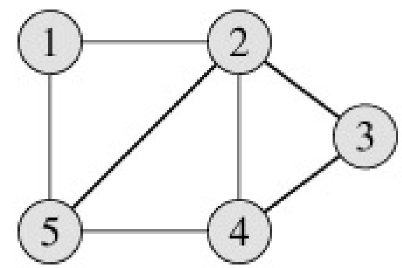


Adjacency Lists Representation

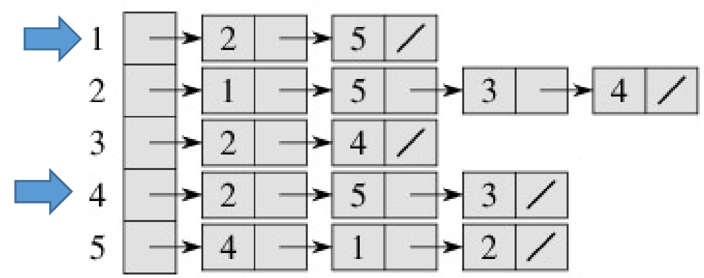
Review

- Pros:
  - Saves on space (memory): the representation takes O(|V|+|E|) memory.
  - Good for large, sparse graphs (e.g., planar maps)

How to find whether there is an edge (4,1)?



- Cons:
  - It can take up to O(n) time to determine if a pair of nodes (i, j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i].



Graph Searching

Graph Searching

- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: **build a tree** on the graph
- General Procedure:
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

Graph Searching

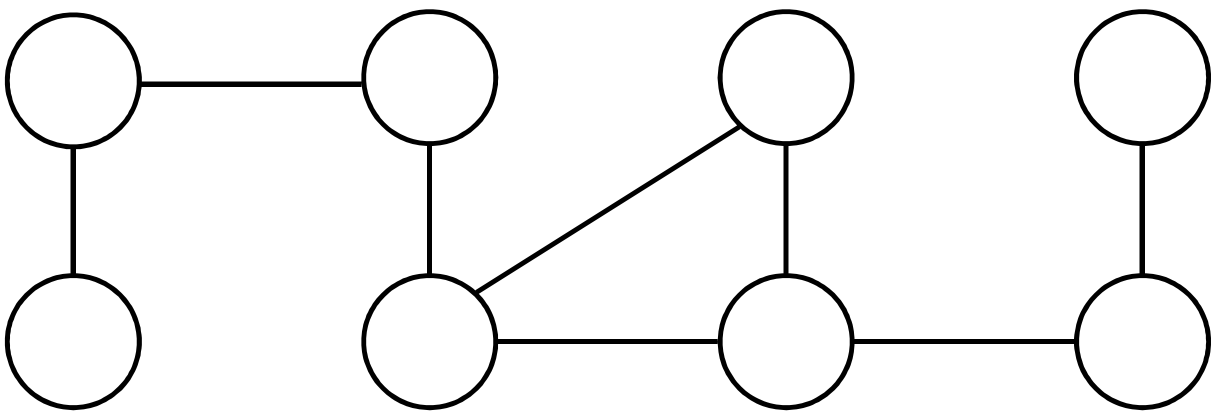
- There are two standard graph traversal techniques:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

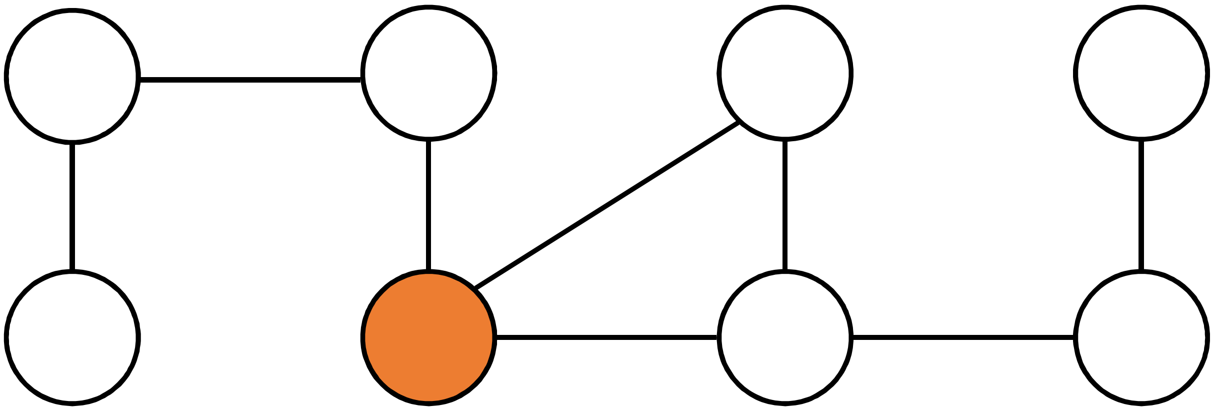
Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



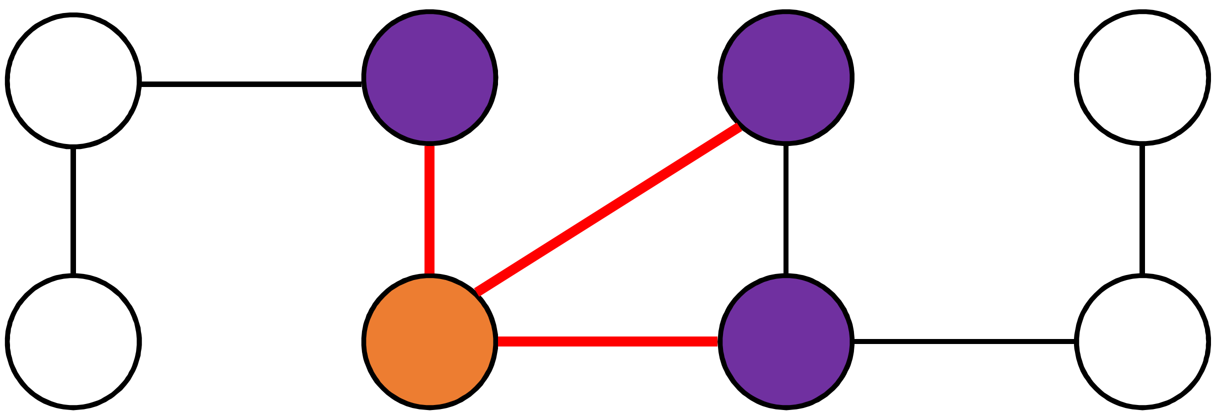
Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



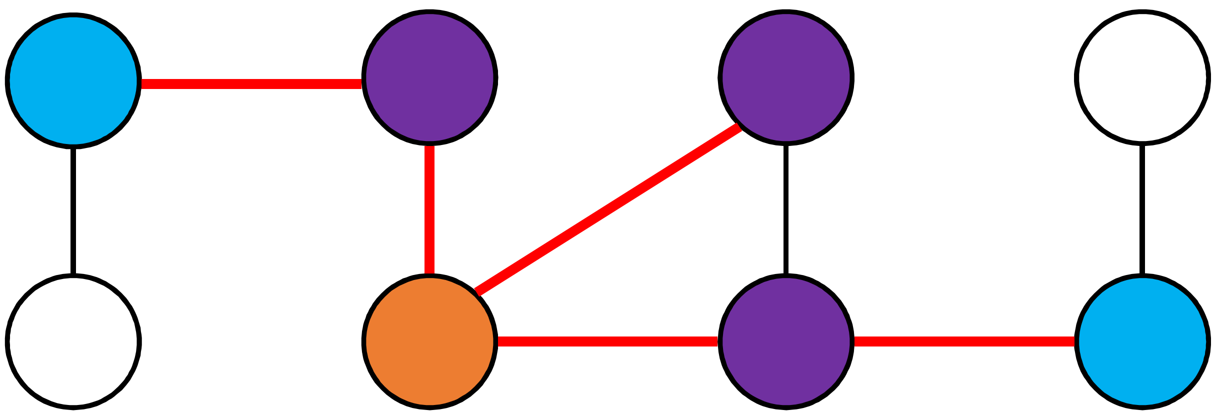
Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



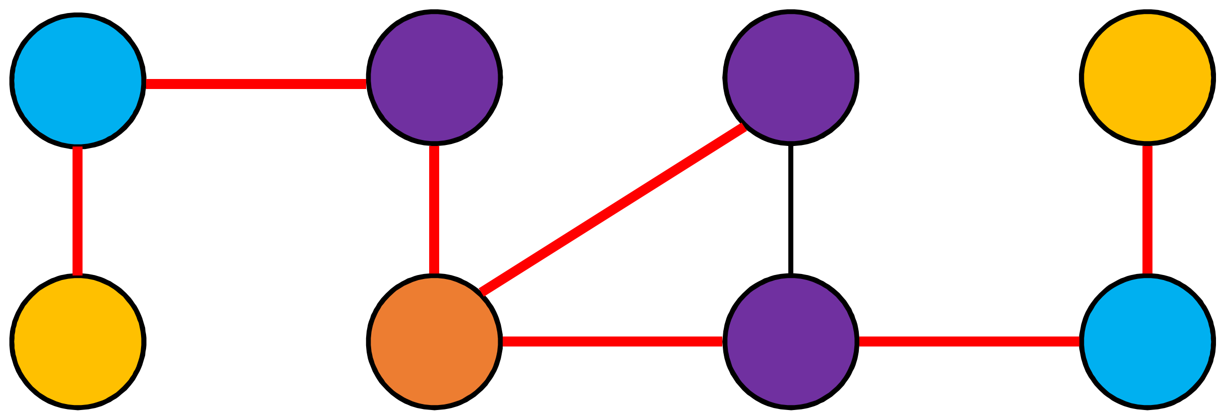
Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



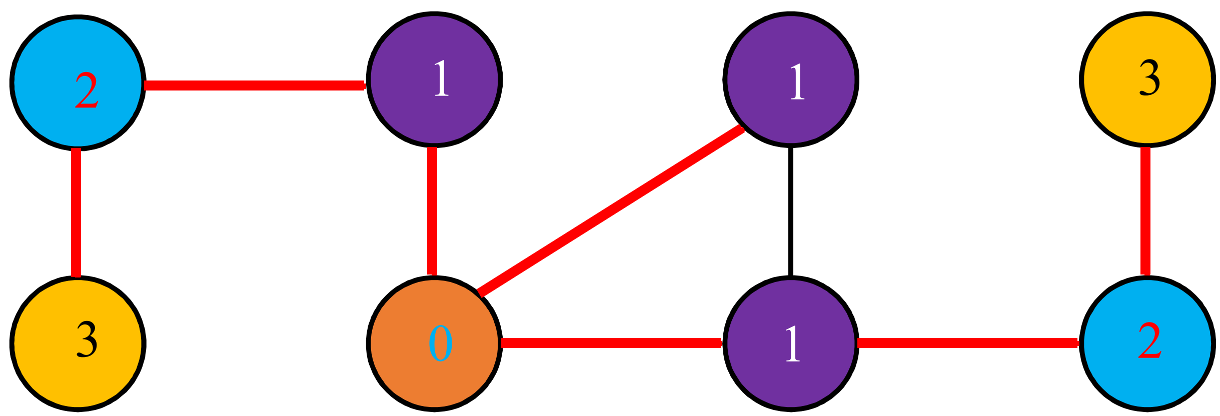
Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



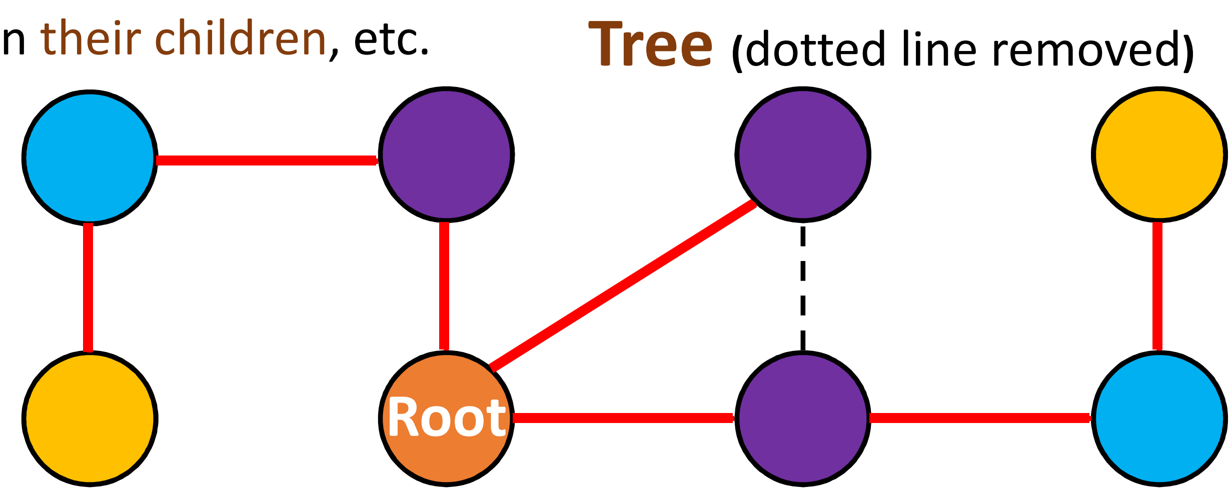
Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



Breadth-First Search

- It associates vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and grey vertices
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Whitening

Breadth-First Search

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = Ø
ENQUEUE(Q, s)
while Q ≠ Ø
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Whitening

Enqueue the root

Breadth-First Search

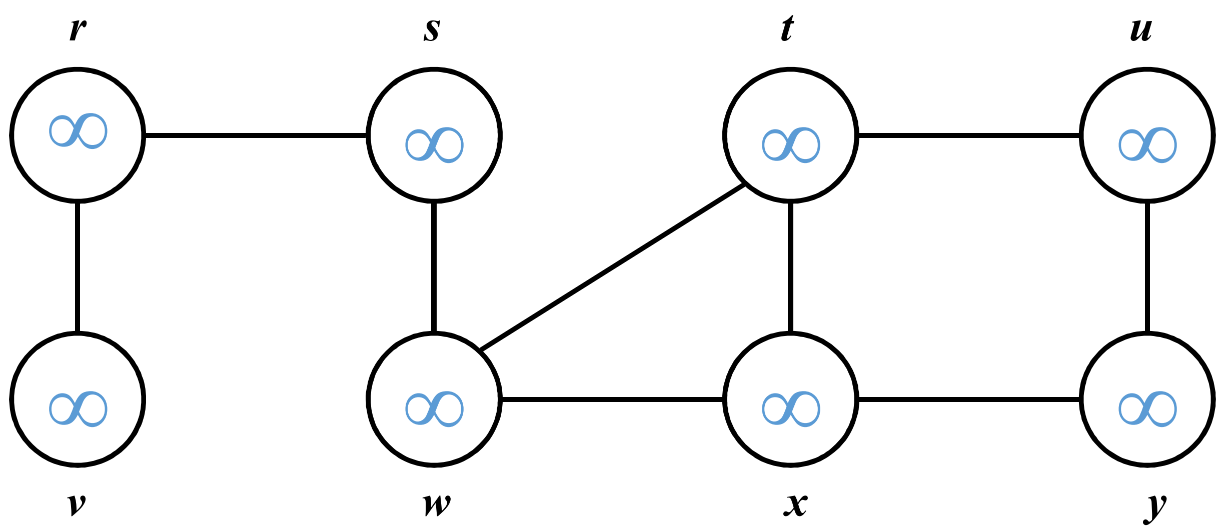
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Whitening

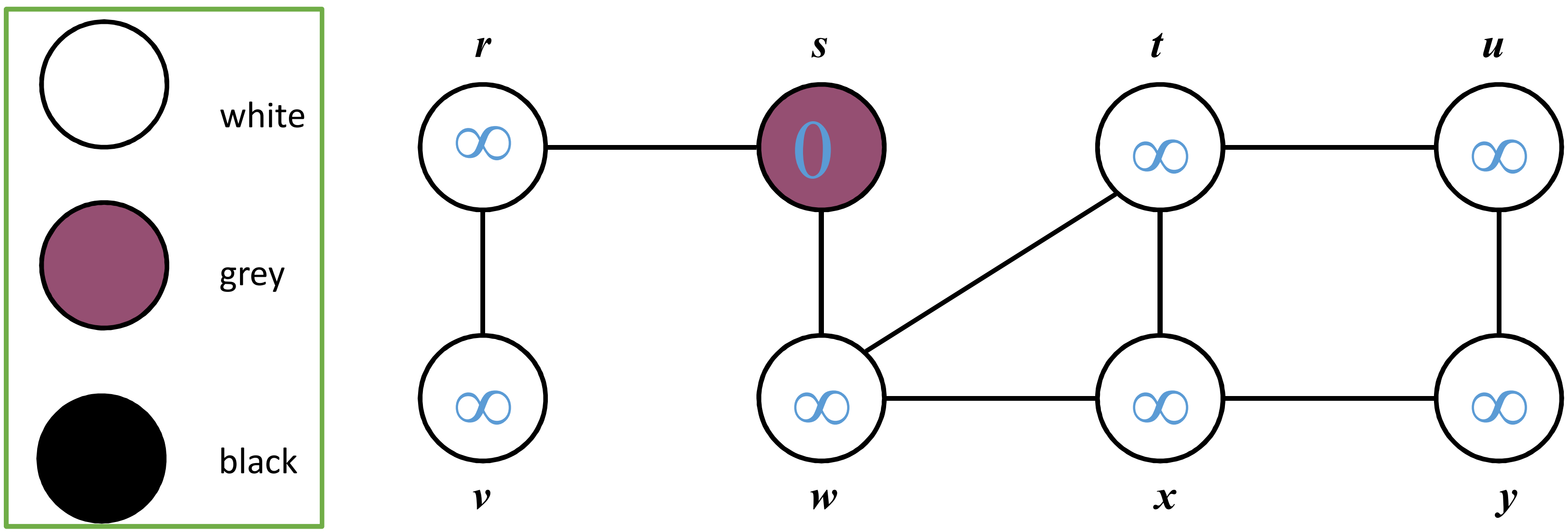
Enqueue the root

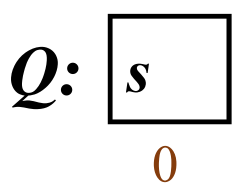
runs until queue is empty

Breadth-First Search: Example



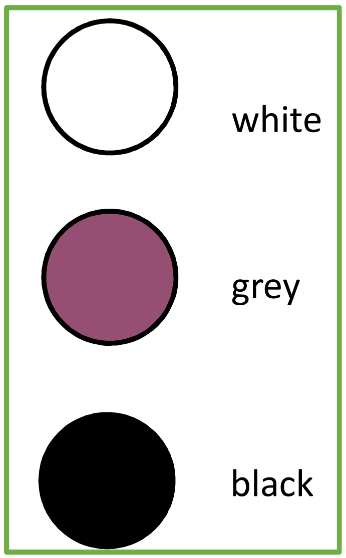
Breadth-First Search: Example

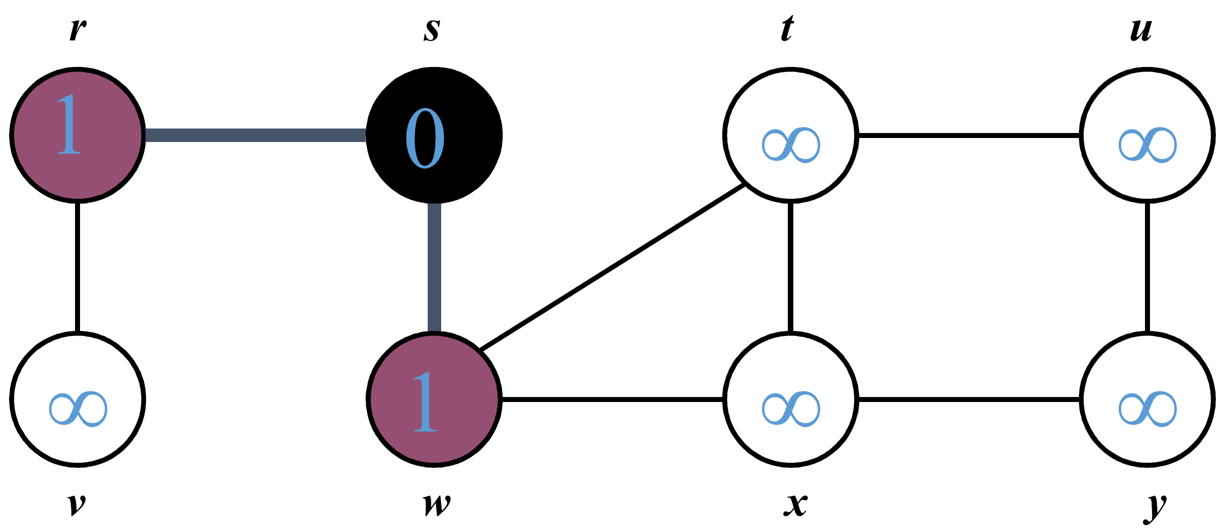


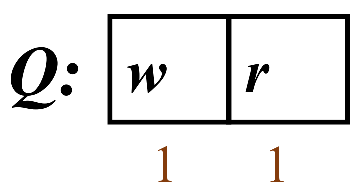



```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search: Example

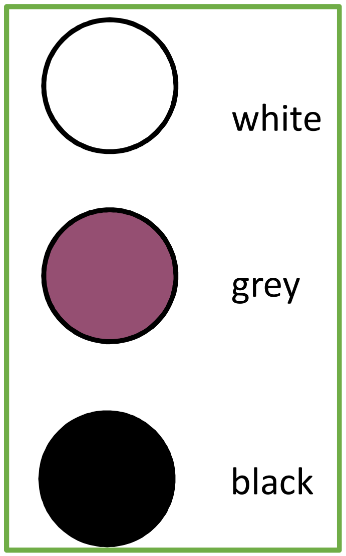


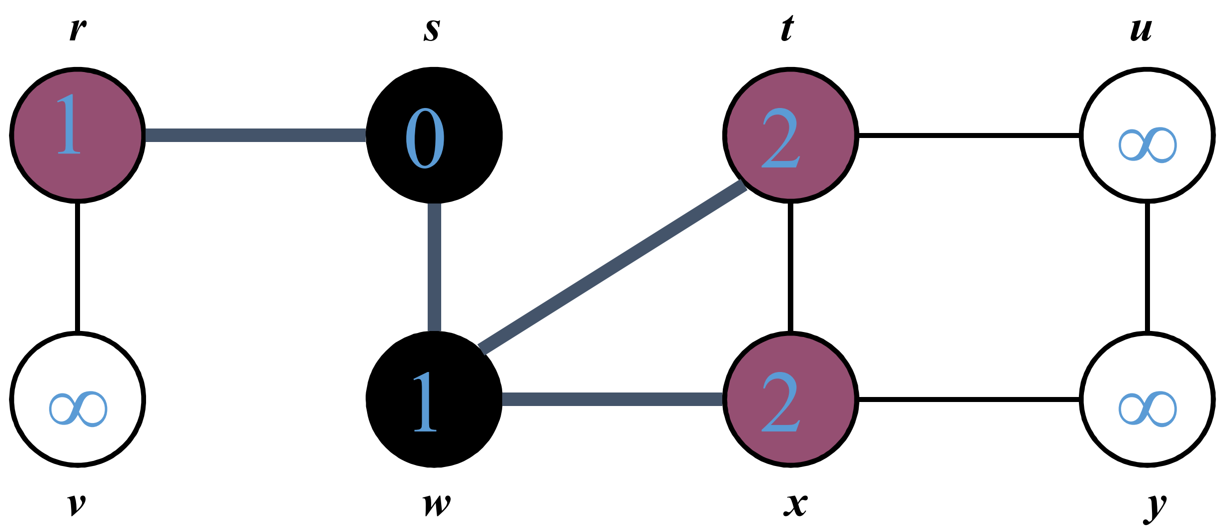


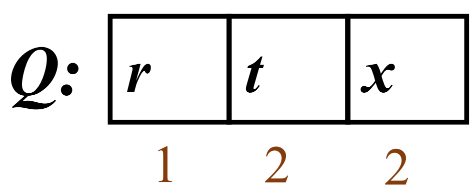



```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search: Example

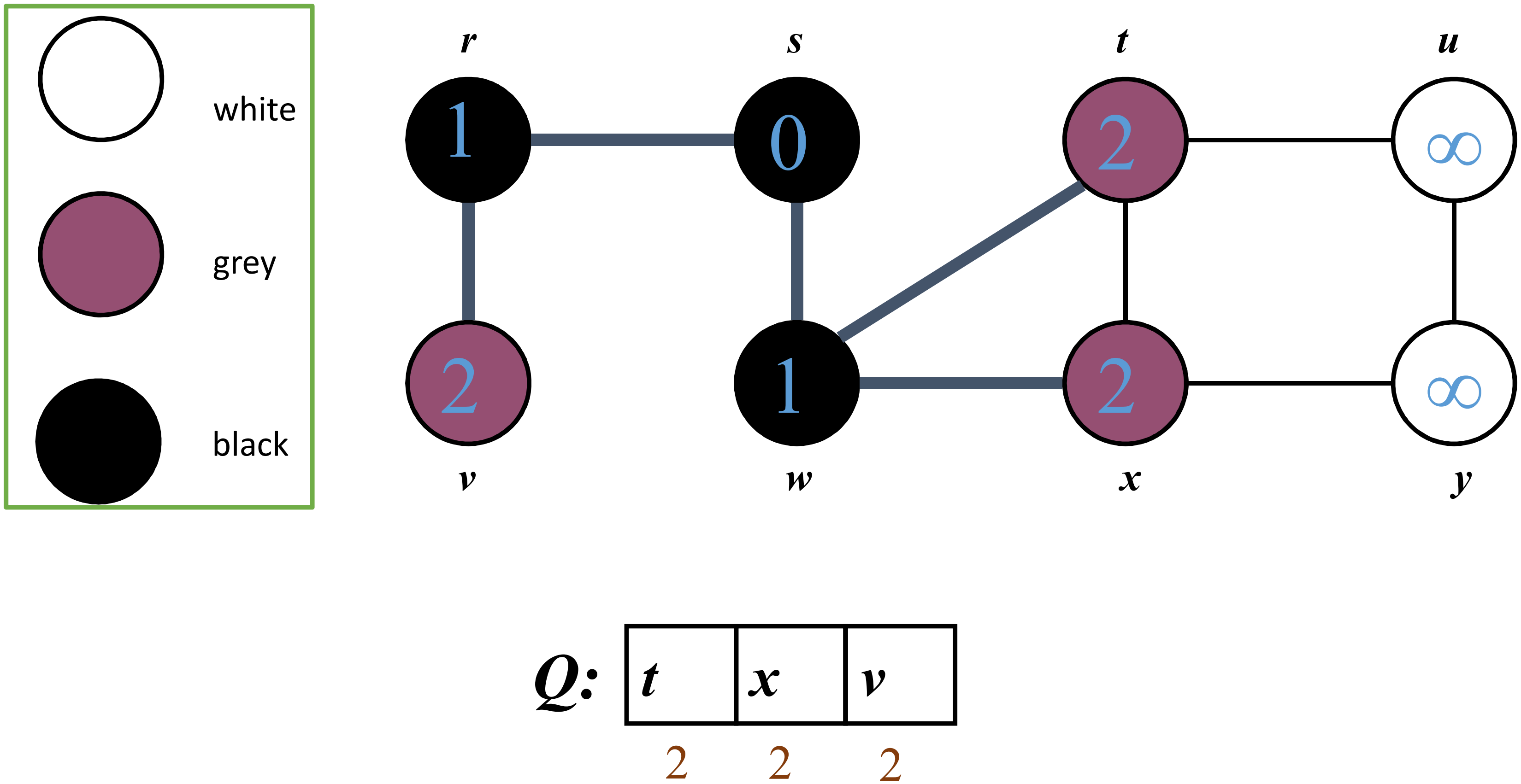






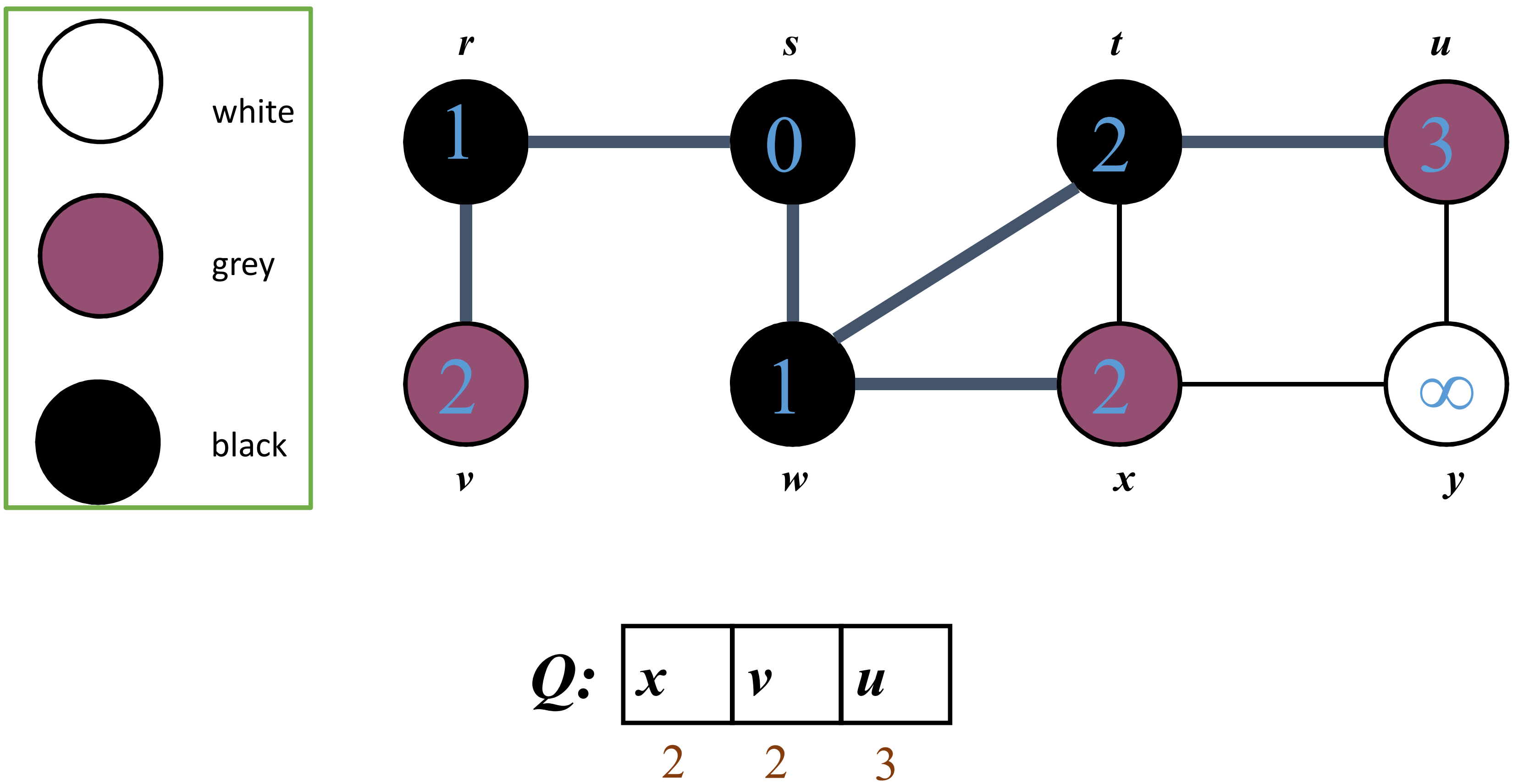

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search: Example



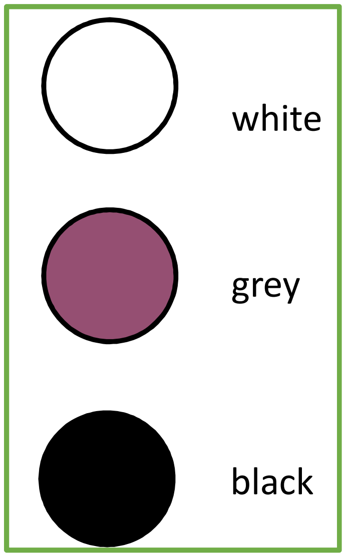

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

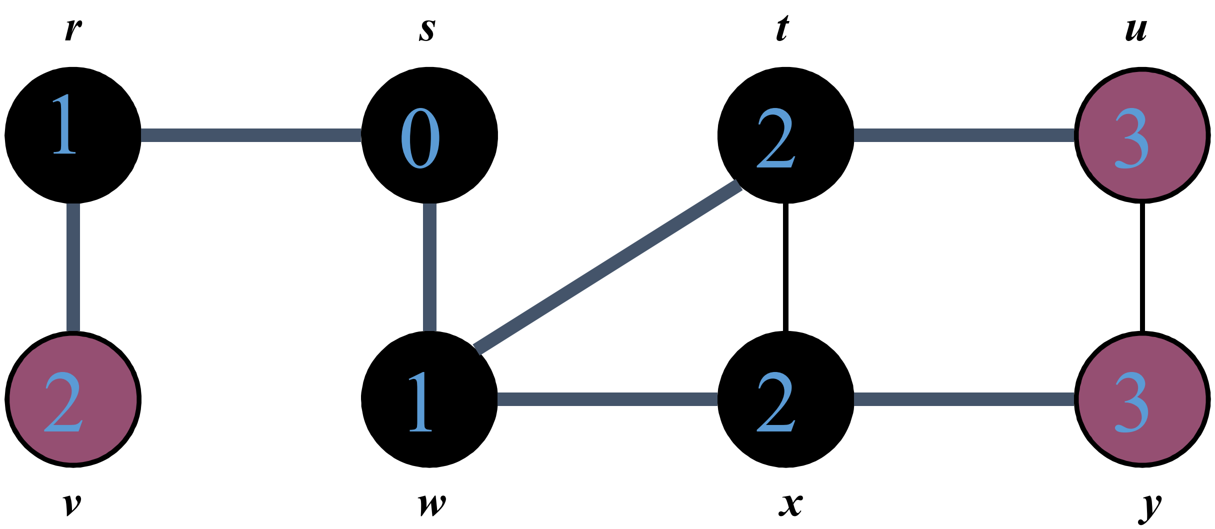
Breadth-First Search: Example

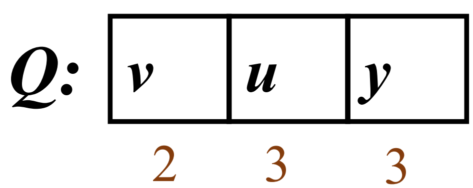



```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search: Example

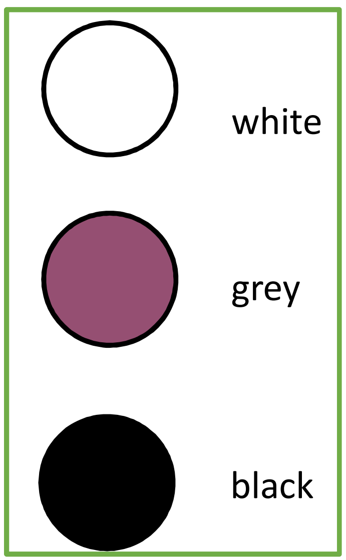


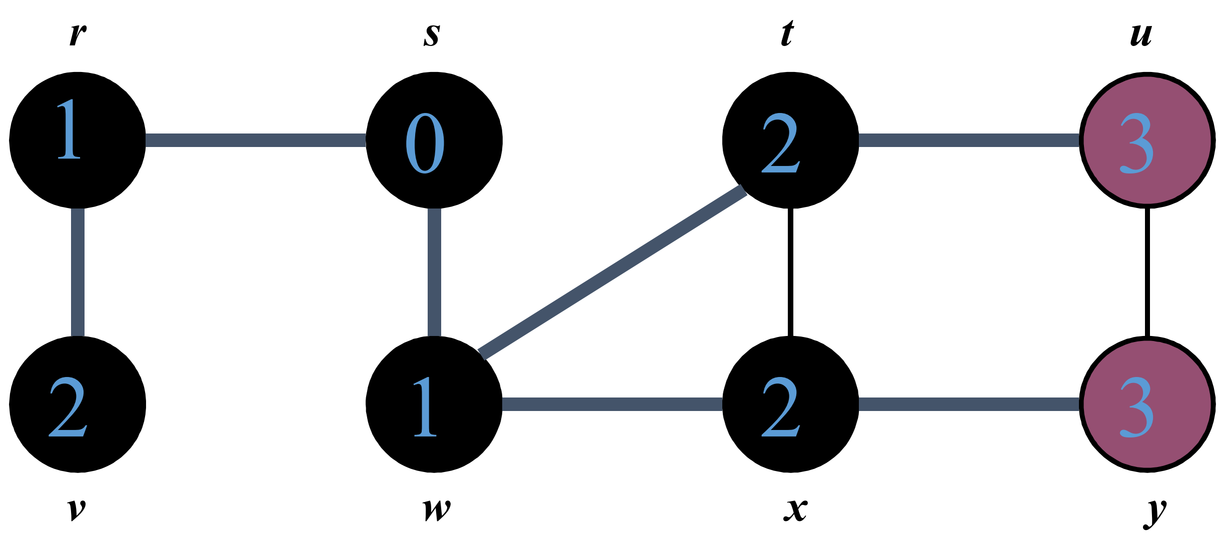


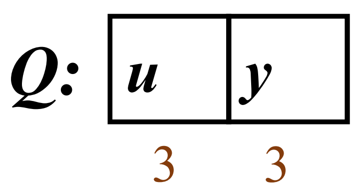



```
BFS(G, s)
  for each vertex u ∈ G.V − {s}
      u.color = WHITE
      u.d = ∞
      u.π = NIL
  s.color = GRAY
  s.d = 0
  s.π = NIL
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ G.Adj[u]
          if v.color == WHITE
              v.color = GRAY
              v.d = u.d + 1
              v.π = u
              ENQUEUE(Q, v)
      u.color = BLACK
```

Breadth-First Search: Example

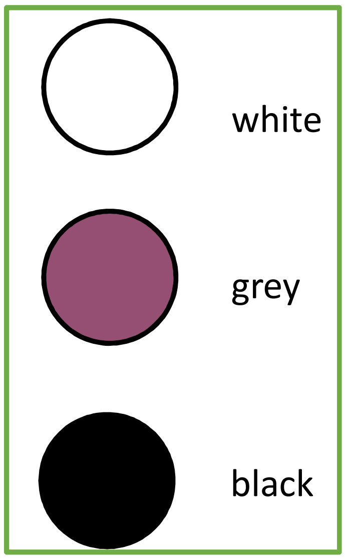


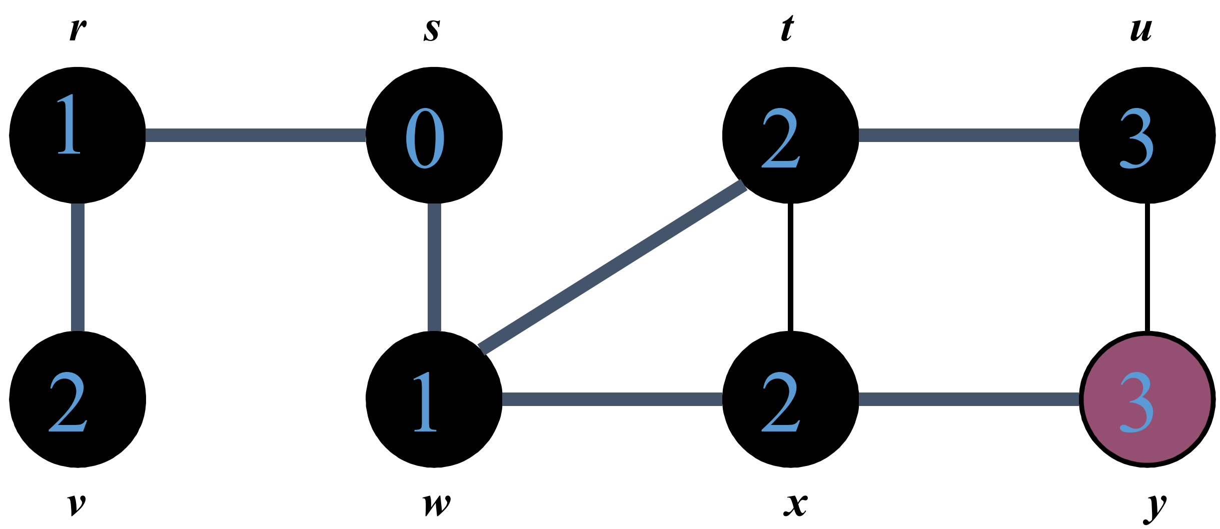





```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search: Example



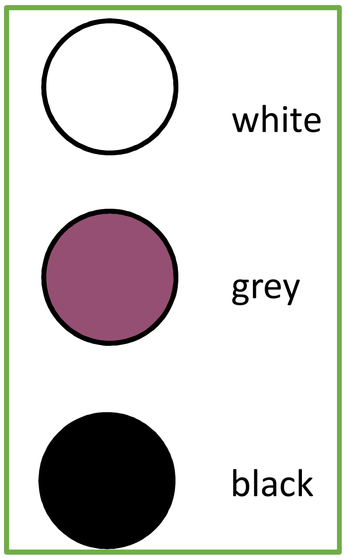


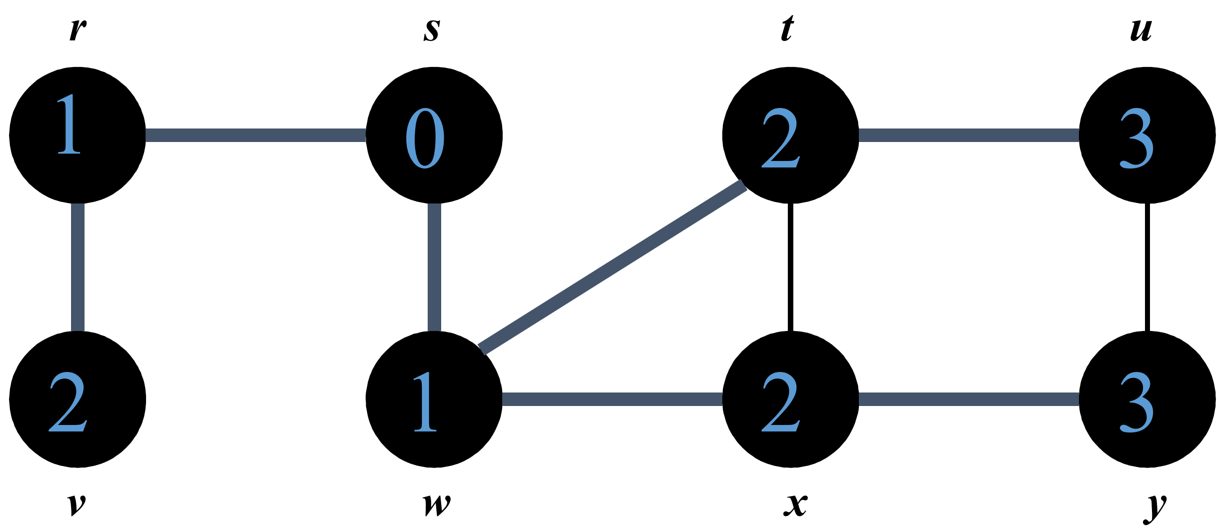
Q: $\boxed{y}$

3

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search: Example





Q: Ø

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = Ø
ENQUEUE(Q, s)
while Q ≠ Ø
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

BFS: Analysis

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

O(V)

O(1)

BFS: Analysis

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

O(V)

O(1)

Each vertex is enqueued/dequeued once: O(V) in total

Once dequeued, the adjacency list of a vertex is explored: O(E) in total

BFS: Analysis

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

O(V)

O(1)

Each vertex is enqueued/dequeued once: O(V) in total

Once dequeued, the adjacency list of a vertex is explored: O(E) in total

O(V+E)

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* from the source node
  - *Shortest-path distance* $\delta(s, v)$

    = minimum number of edges from s to v, OR

    = ∞, if v is NOT reachable from s

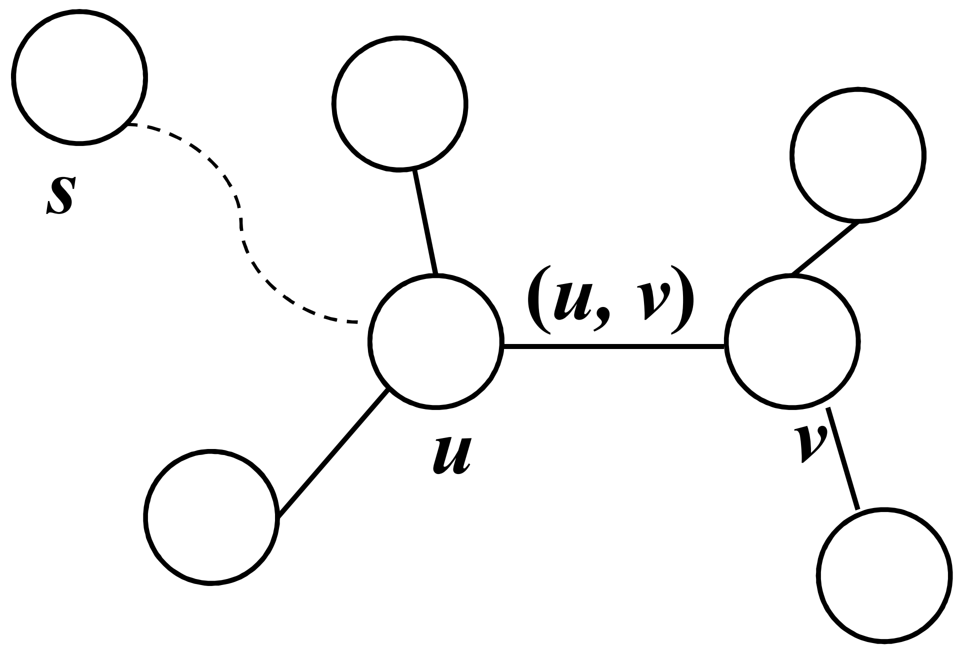
Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* from the source node
  - *Shortest-path distance* $\delta(s, v)$
    = minimum number of edges from s to v, OR
    = ∞, if v is NOT reachable from s
- BFS builds *breadth-first tree*, in which paths from root represent shortest paths in G
  - Thus we can use BFS to calculate shortest path from one vertex to another in $O(V+E)$ time in an unweighted graph

Lemma 22.1

Let G = (V, E) be a directed or undirected graph, and let s Ɛ V be an arbitrary vertex. Then, for any edge (u, v) Ɛ E,

$\delta(s, v) \leq \delta(s, u) + 1$

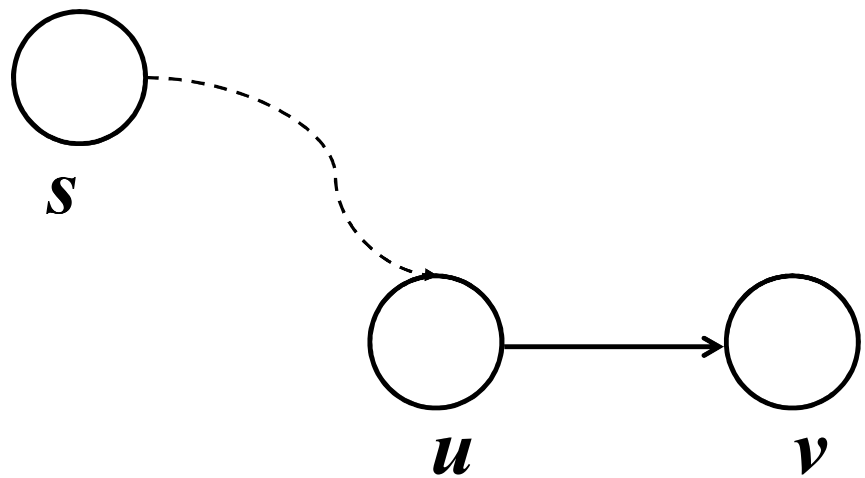


Lemma 22.1

Let G = (V, E) be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$\delta(s, v) \leq \delta(s, u) + 1$

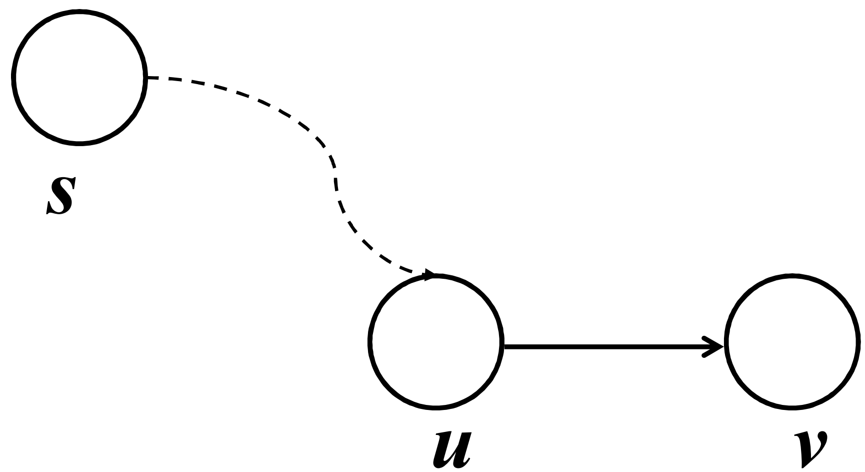
If u is reachable from s, so is v



Lemma 22.1

Let G = (*V*, *E*) be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$\delta(s, v) \leq \delta(s, u) + 1$

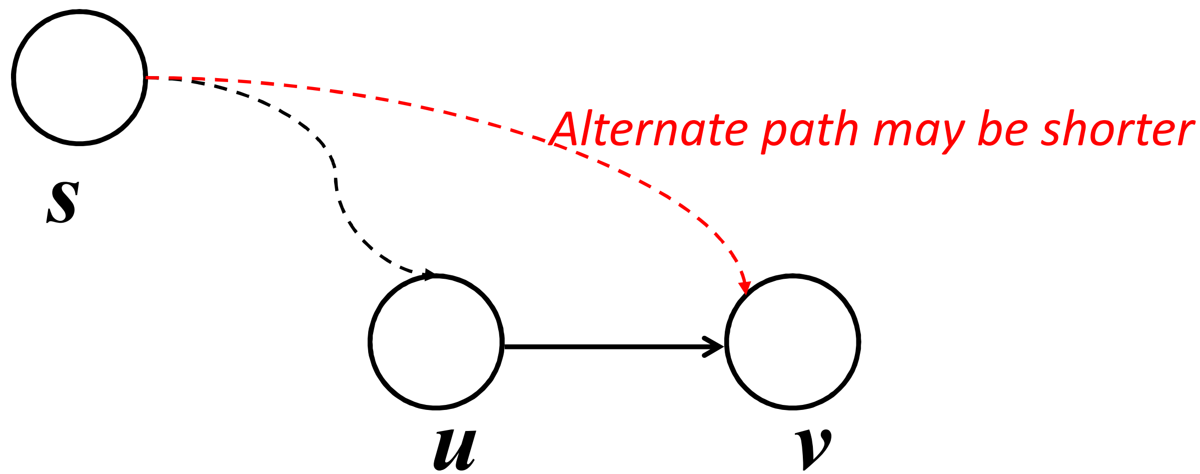


Shortest-path to v *cannot be longer than shortest path to u* plus edge (*u, v)*

Lemma 22.1

Let G = (V, E) be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$\delta(s, v) \leq \delta(s, u) + 1$



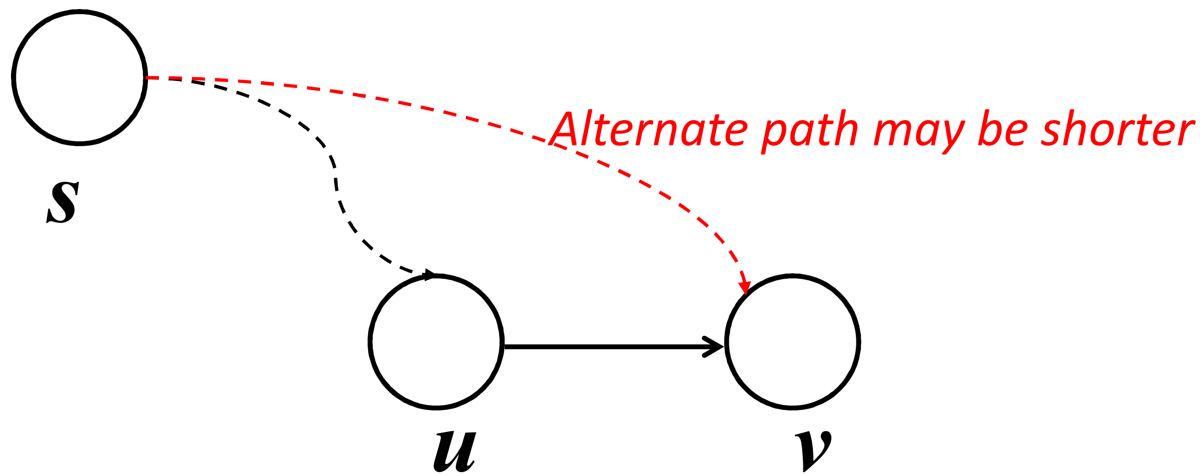
Shortest-path to v *cannot be longer than shortest path to* u plus edge (u, v)

Lemma 22.1

Let G = (V, E) be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$\delta(s, v) \leq \delta(s, u) + 1$

So we proved



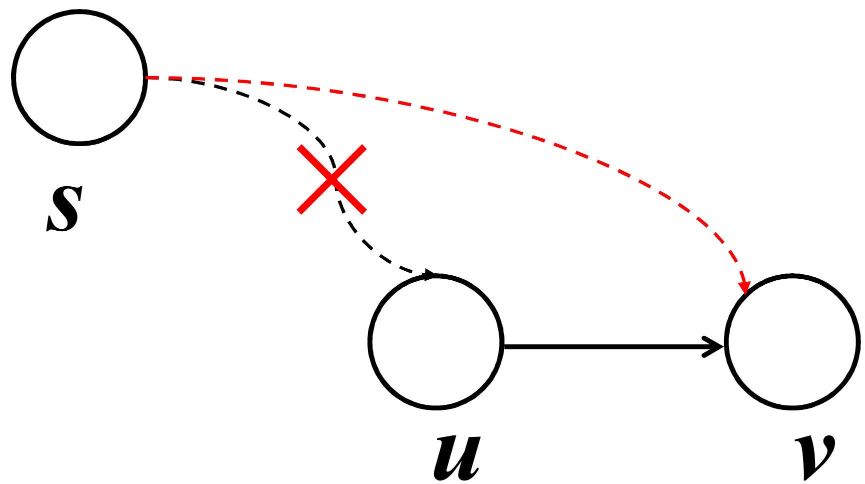
Shortest-path to v *cannot be longer than shortest path to* u plus edge (u, v)

Lemma 22.1

Let G = (V, E) be a directed or undirected graph, and let s Ɛ V be an arbitrary vertex. Then, for any edge (u, v) Ɛ E,

$\delta(s, v) \leq \delta(s, u) + 1$

If u is NOT reachable from s

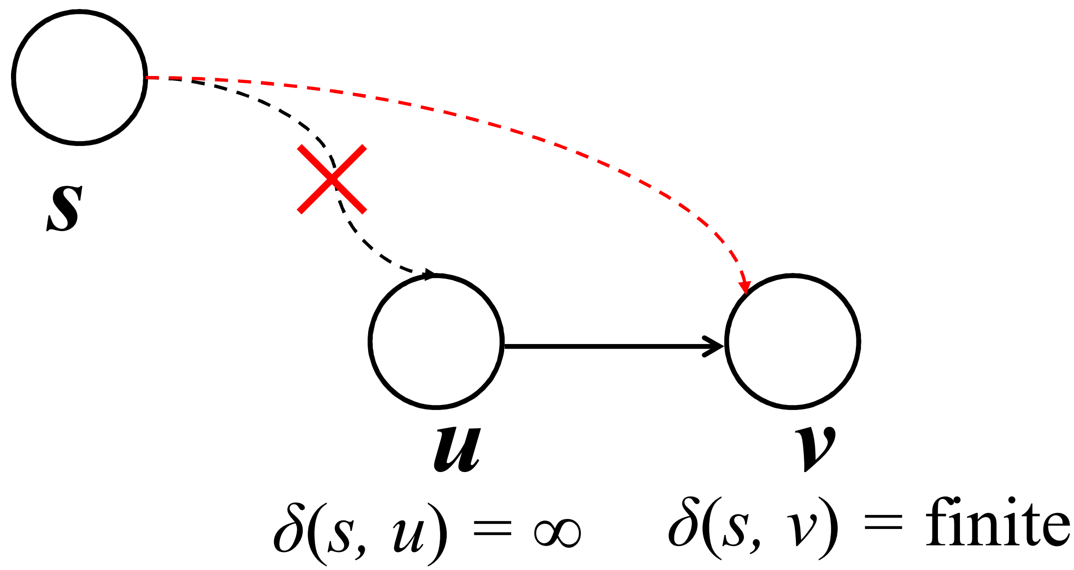


Lemma 22.1

Let G = (V, E) be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$\delta(s, v) \le \delta(s, u) + 1$

If u is NOT reachable from s



Lemma 22.1

Let G = (V, E) be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$\delta(s, v) \leq \delta(s, u) + 1$

we proved again

If u is NOT reachable from s