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

Instructor

Dr Md Monirul Islam

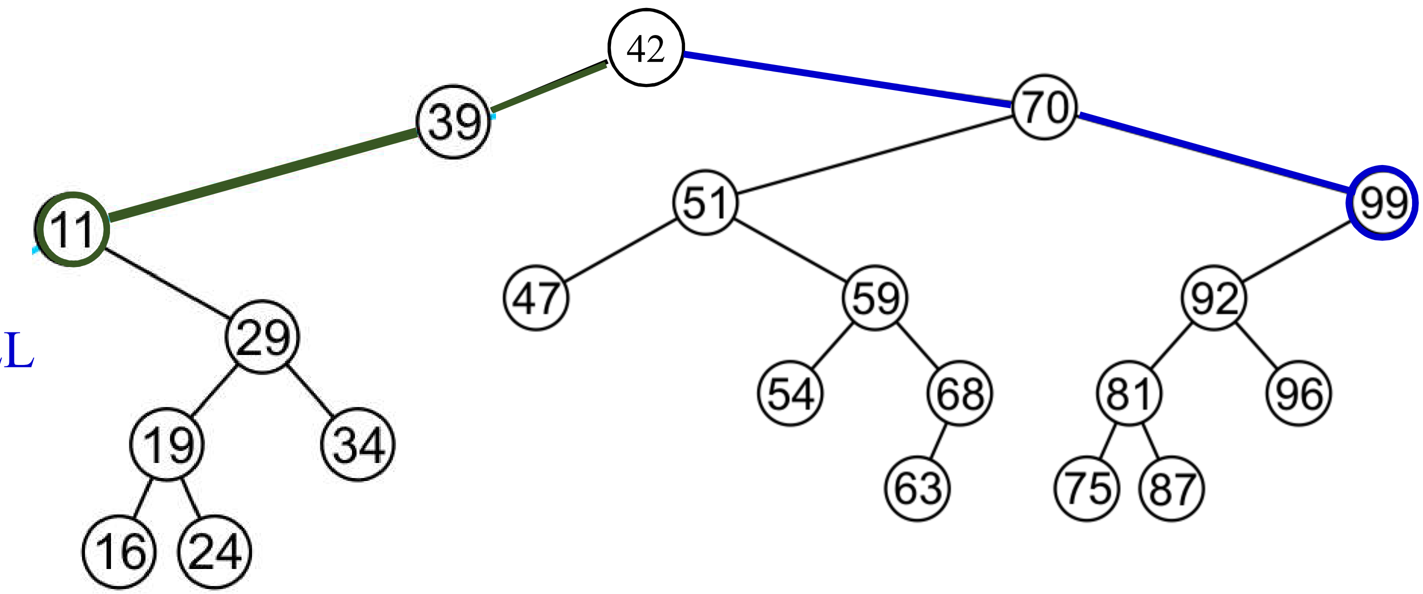
Graphs and Trees: Representation and Search

BST Operation: Minimum and Maximum

Review

```
TREE_MINIMUM (x)
if x == NULL return NULL
while x->left ≠ NULL
      x = x->left
return x
```

Complexity: O(*h*)



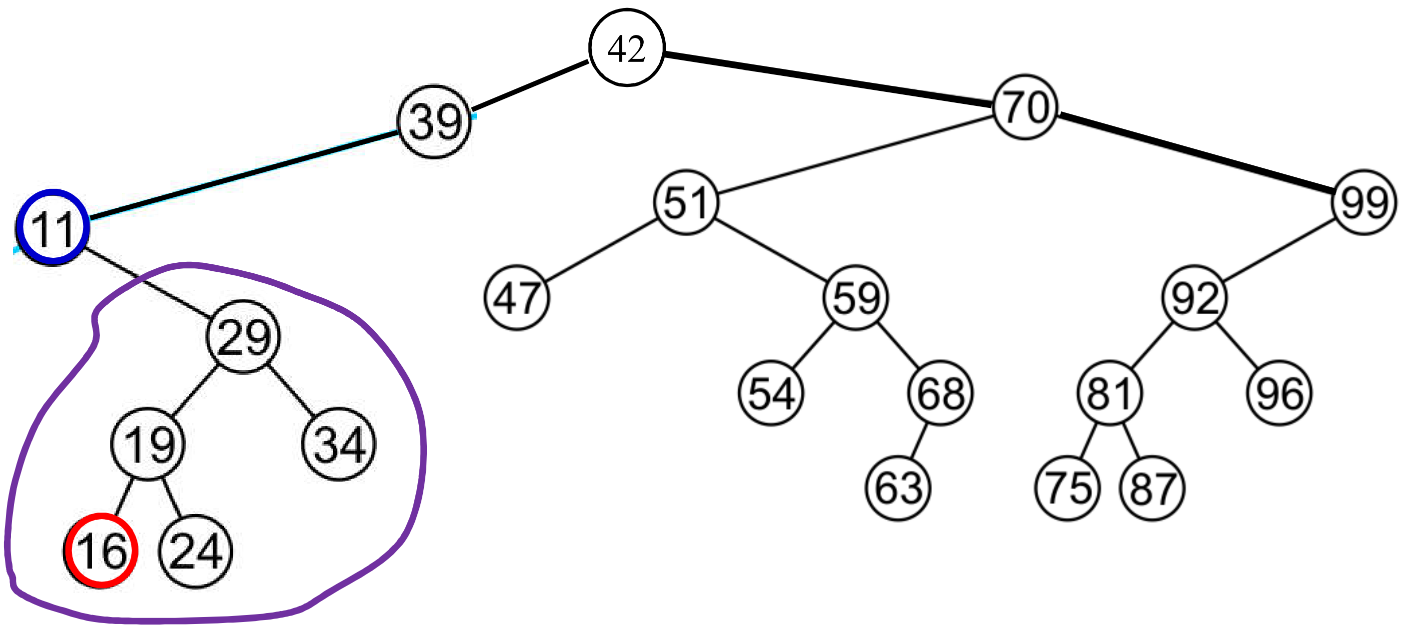

```
TREE_MXIMUM (x)
if x == NULL return NULL
while x->right ≠ NULL
      x = x->right
return x
```

BST Operation: Successor

Review

successor of a node *x* : the node with the smallest key greater than *x.key*

successor of the node with 11 : the node with 16 (minimum of right subtree)



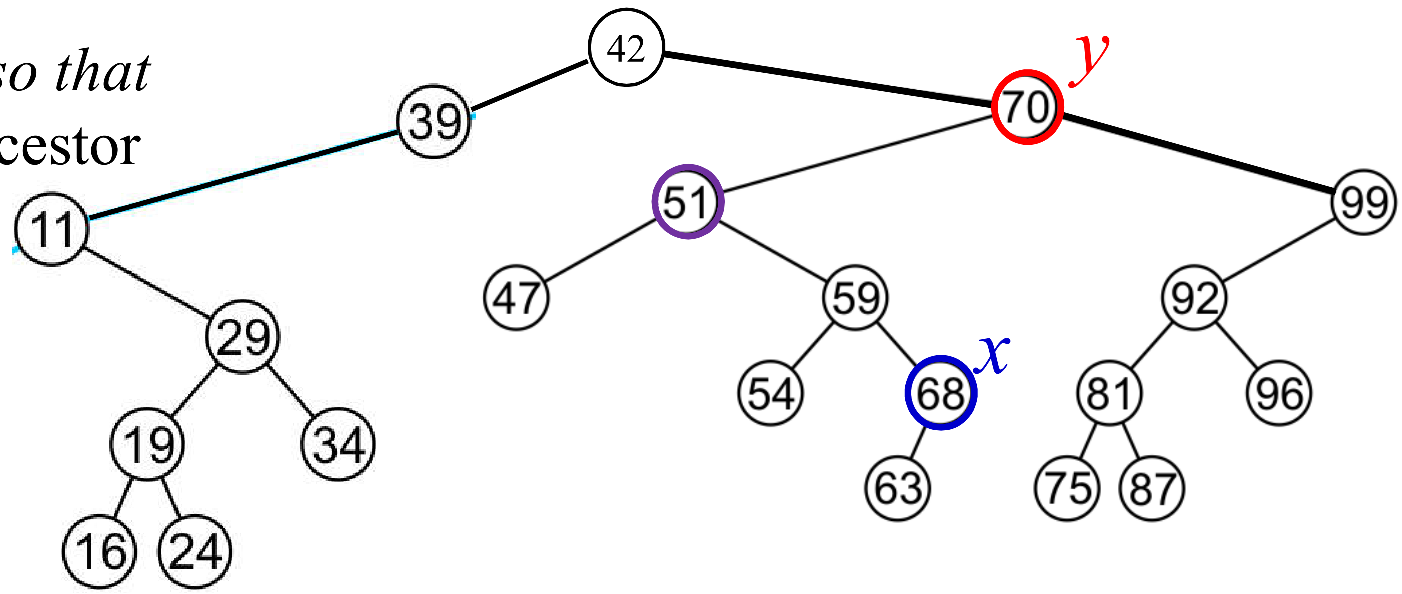
BST Operation: Successor

Review

successor of a node x : the node with the smallest key greater than $x.key$

successor of the node with 68 : the node with 70 (right subtree is NULL)

y is successor of x
y is lowest ancestor of x *so that*
y's left child is also an ancestor of x



BST Operation: Deletion

Review

A node being deleted is not always going to be a leaf node

There are three possible scenarios:

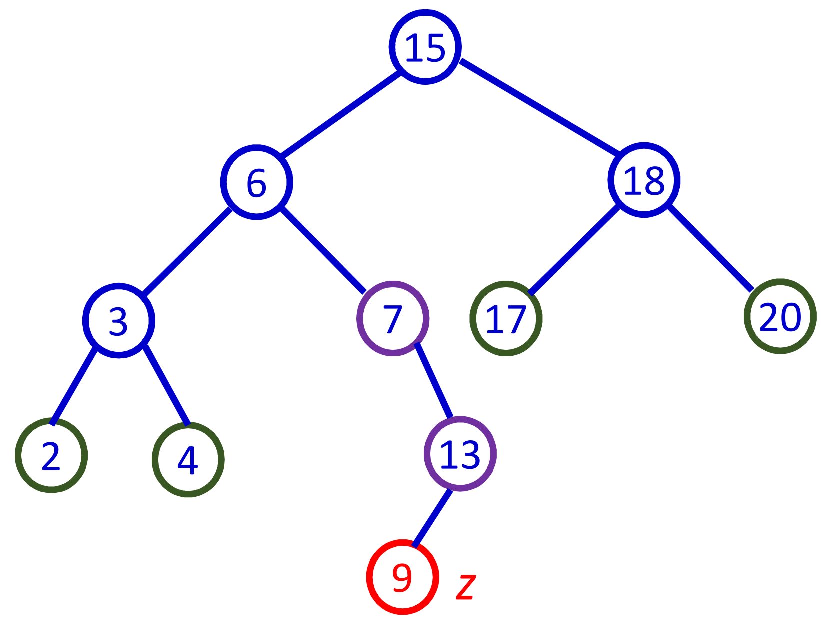
- The node is a leaf node
- It has exactly one child, or
- It has two children (it is a full node)

BST Operation: Deletion

Review

Removing a leaf node
Set left pointer of 13 as NULL

If *z== z->parent ->left*
    z->parent ->left =NULL
else *z->parent ->right* =NULL



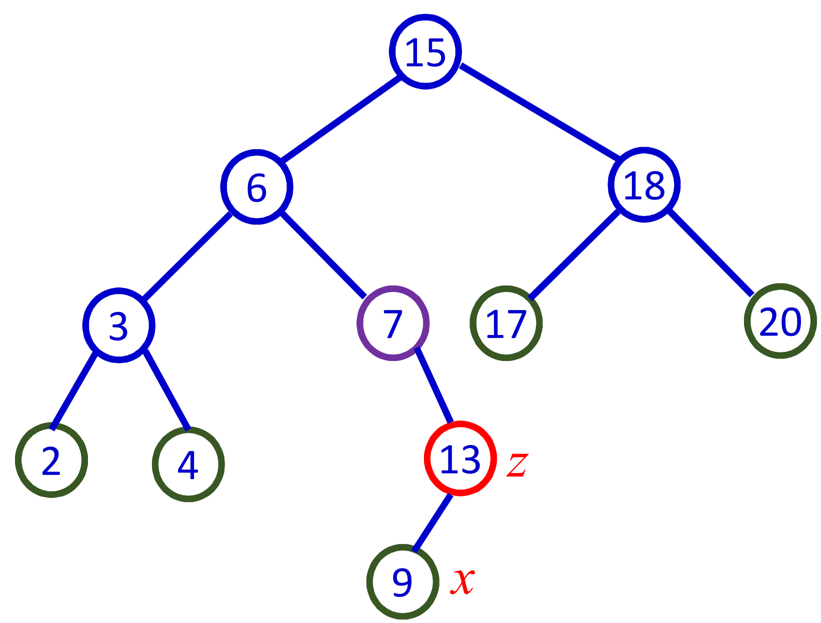
BST Operation: Deletion

Review

Removing a node with exactly one child
Remove node with key 13 which has a left subtree ONLY
Promote the left subtree

If *z->left* =NULL
    x= z->right
else *x= z->left*

If *z== z->parent ->left*
    z->parent ->left =x
else *z->parent ->right =x*
x->parent = z->parent



BST Operation: Deletion

Review

Removing a node with exactly one child

Remove node with key 7 which has a **RIGHT** subtree ONLY

If *z->left* =NULL

    x= z->right

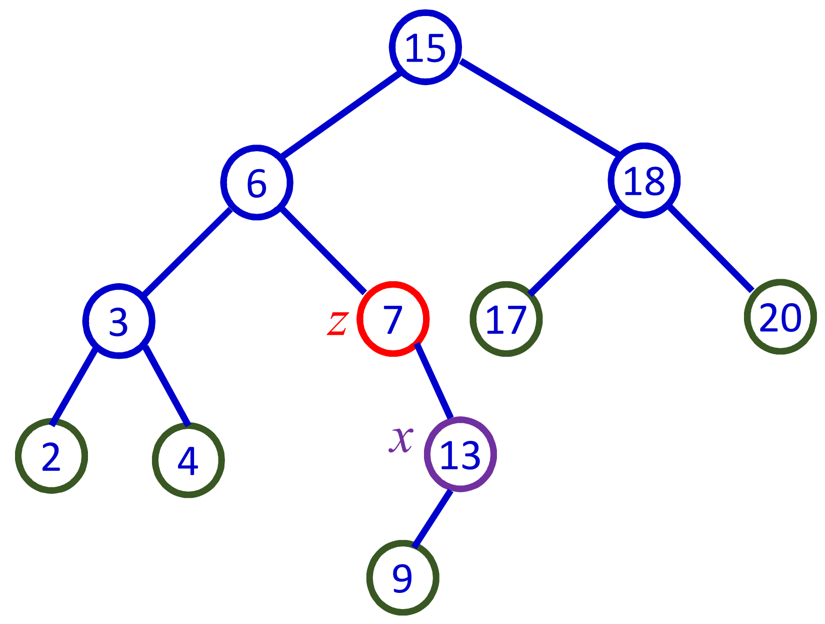
else *x= z->left*

If *z== z->parent ->left*

    z->parent ->left =x

else *z->parent ->right =x*

x->parent = z->parent



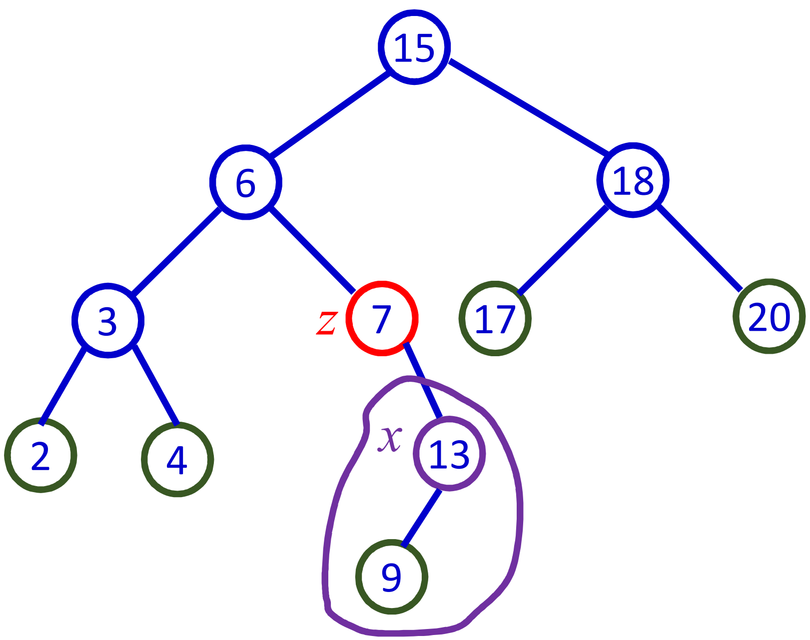
BST Operation: Deletion

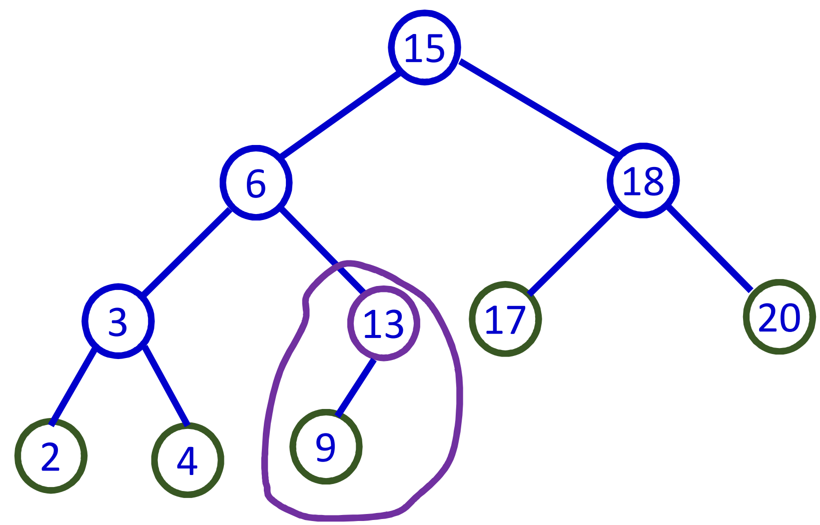
Review

Removing a node with exactly one child

Remove node with key 7 which has a **RIGHT** subtree ONLY

BST property holds





BST Operation: Deletion

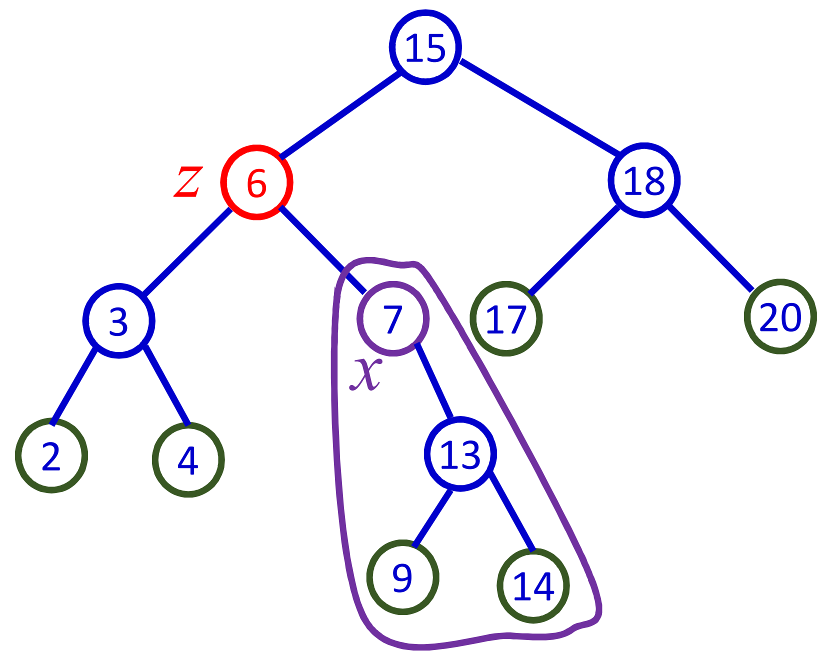
Review

Removing a node having two children (full node)
Remove node with 6

the successor is minimum in the **right** subtree

The minimum will be a **leaf node** OR
a **node with NO left** child

If x would have a left child
that would be the minimum of the subtree



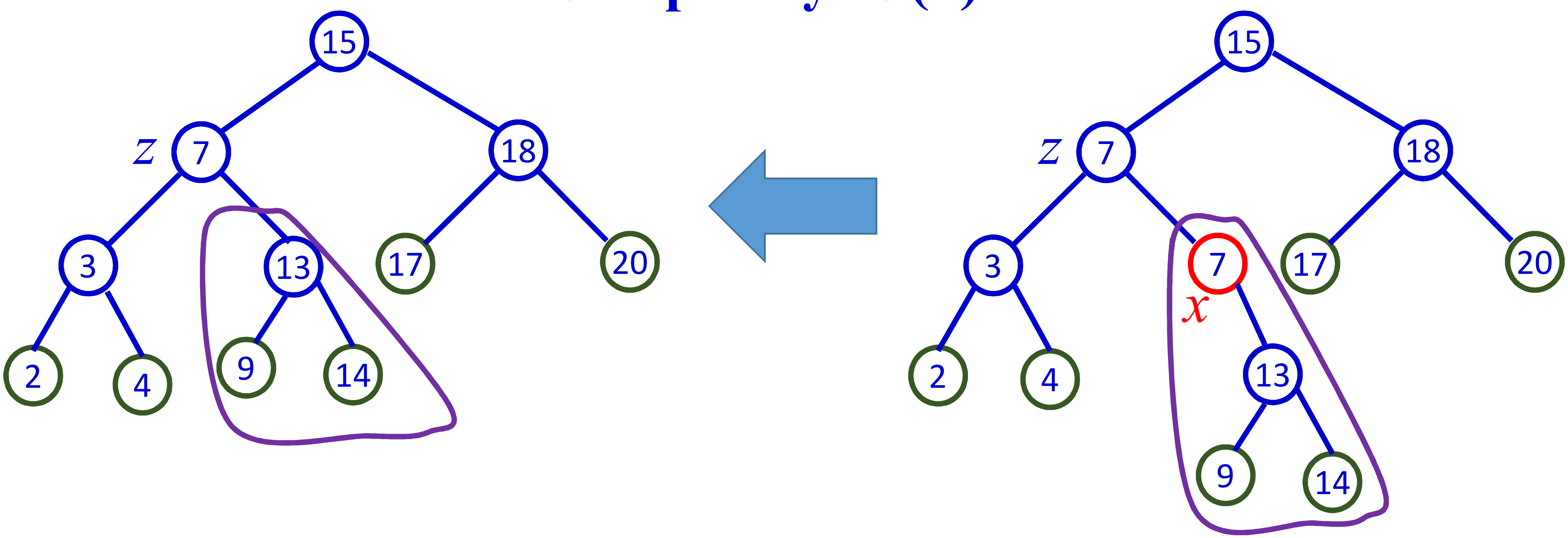
BST Operation: Deletion

Review

Removing a node having two children (full node)

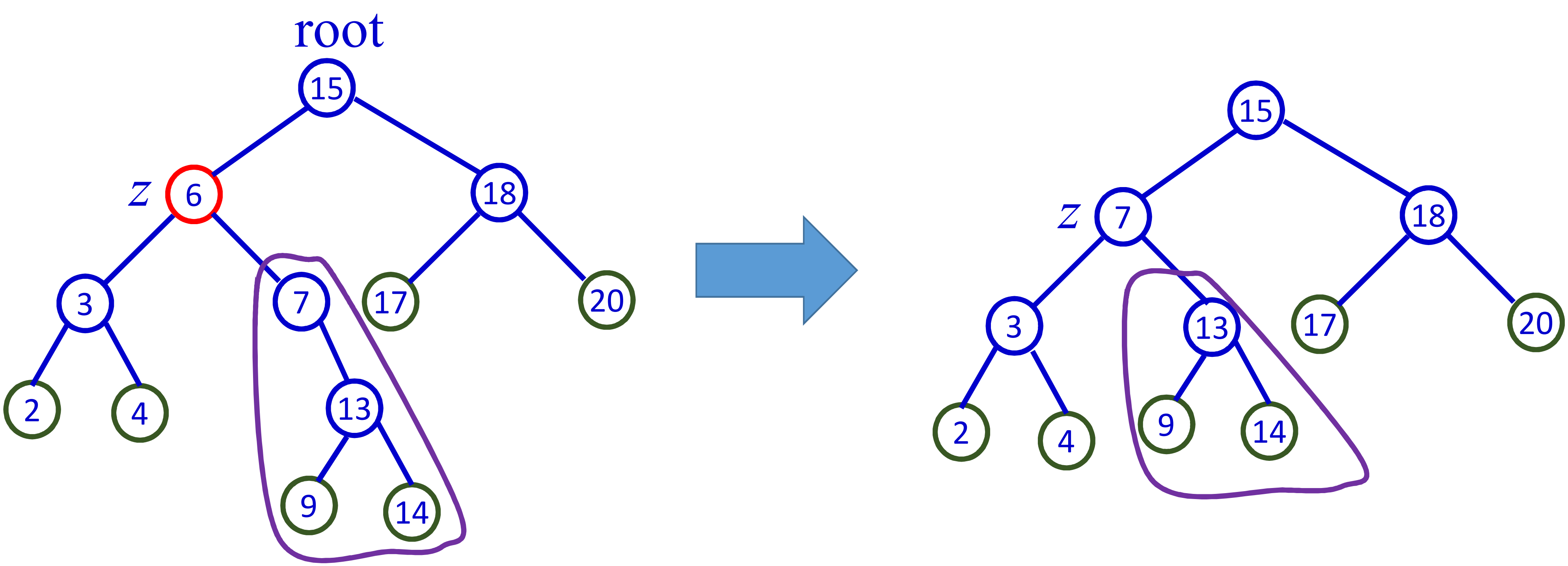
Remove node with 6

Complexity: O(*h*)



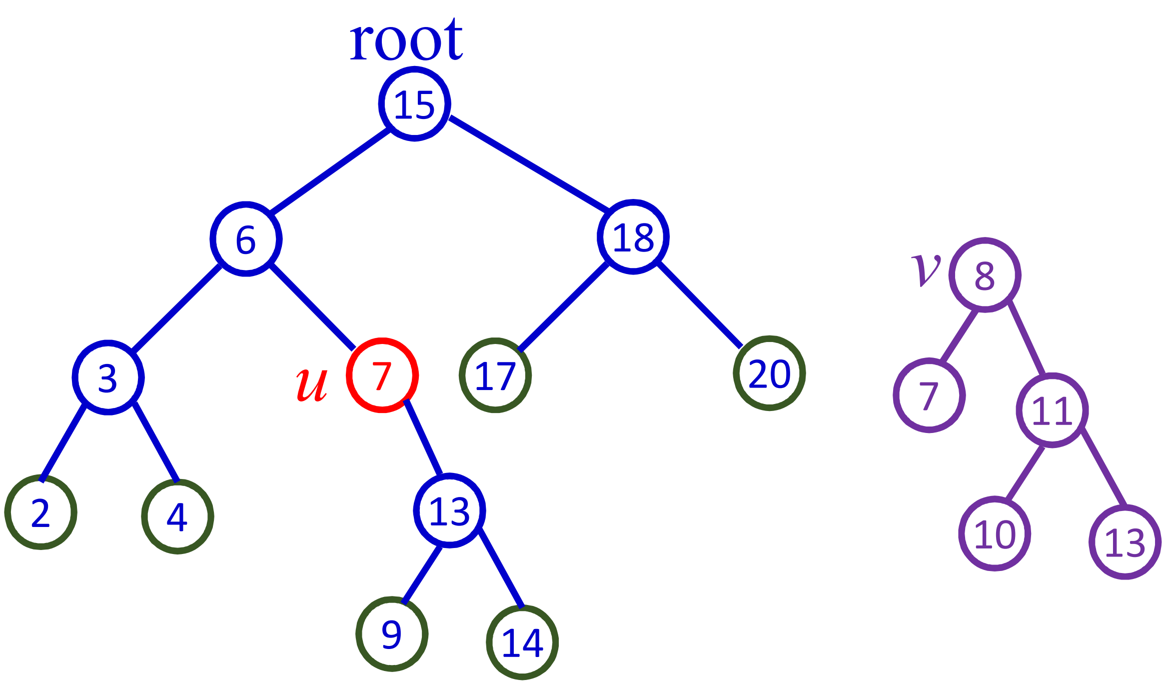
BST Operation: Deletion (2)

Removes a node directly even if it has two children (full node)



BST Operation: Deletion (2)

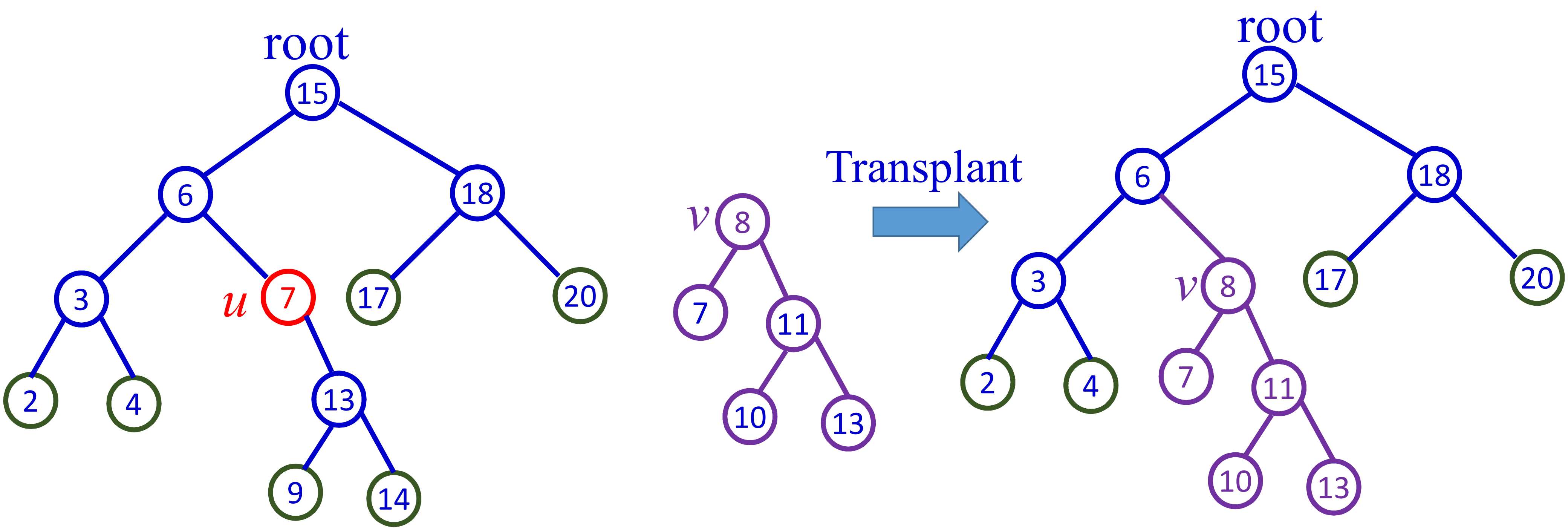
First try to replace node *u* by node *v*



BST Operation: Deletion (2)

First try to replace node *u* by node *v*

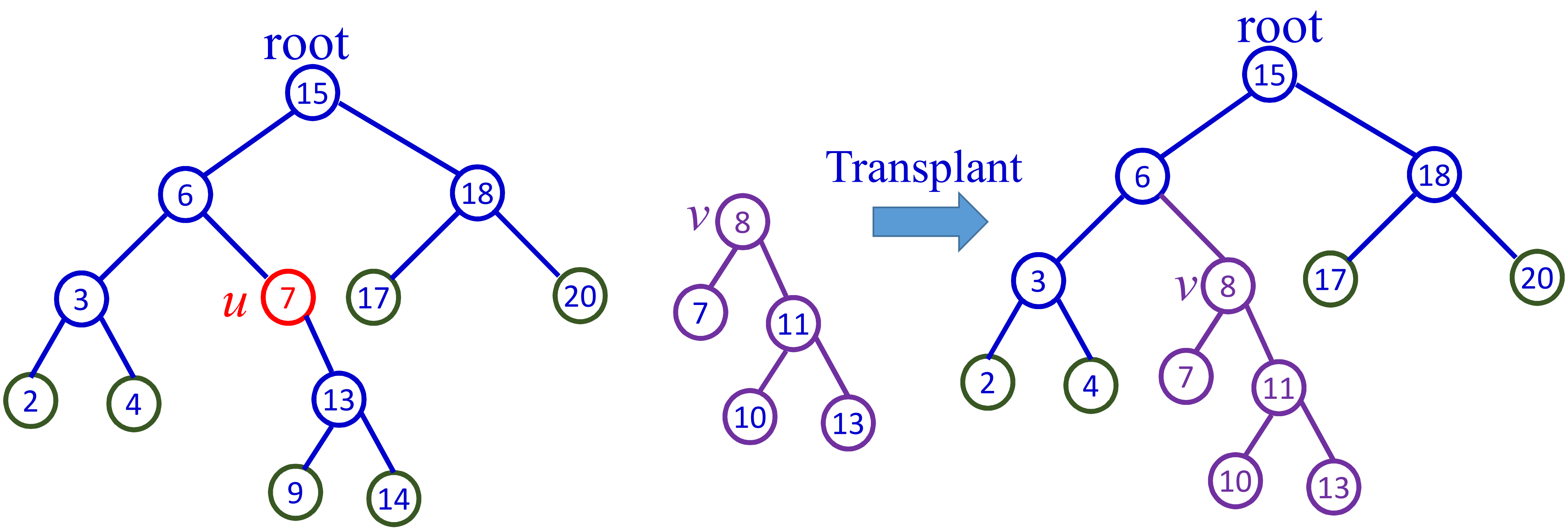
Removes a node directly even if it has two children (full node)



BST Operation: Deletion (2)

Removes a node directly even if it has two children (full node)

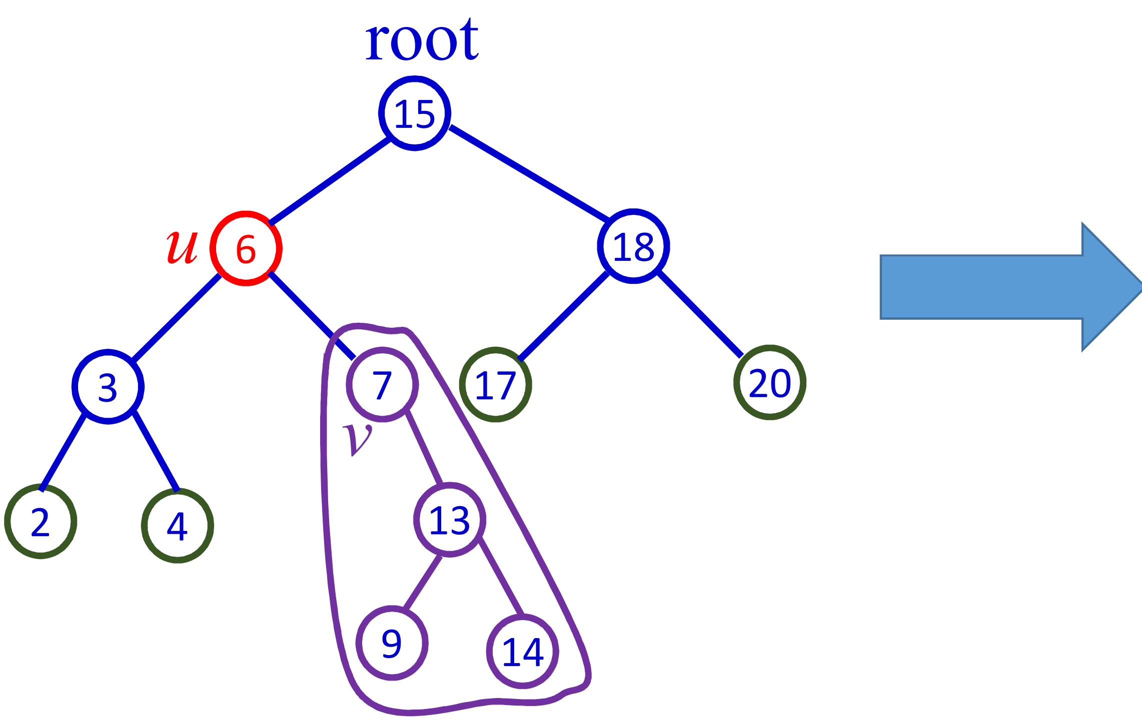
First try to replace node u by node v

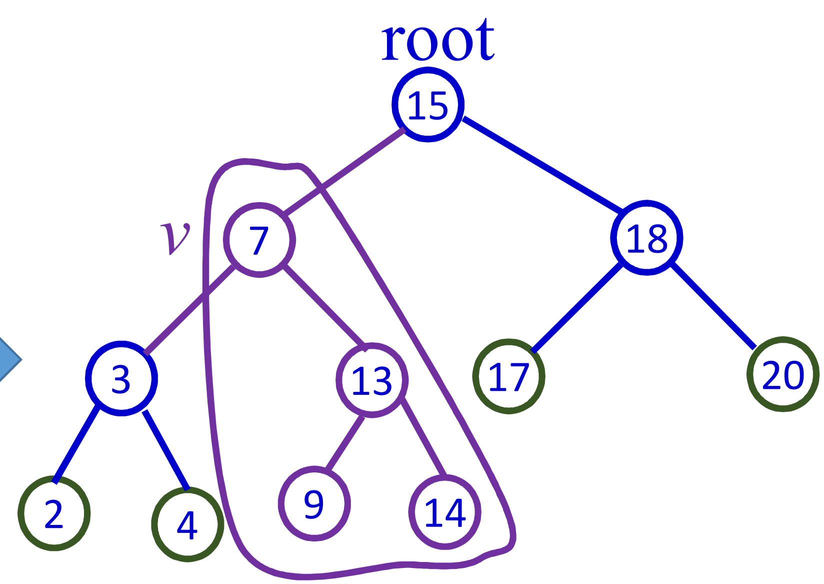


We did not change children of v

BST Operation: Deletion (2)

This is also transplant

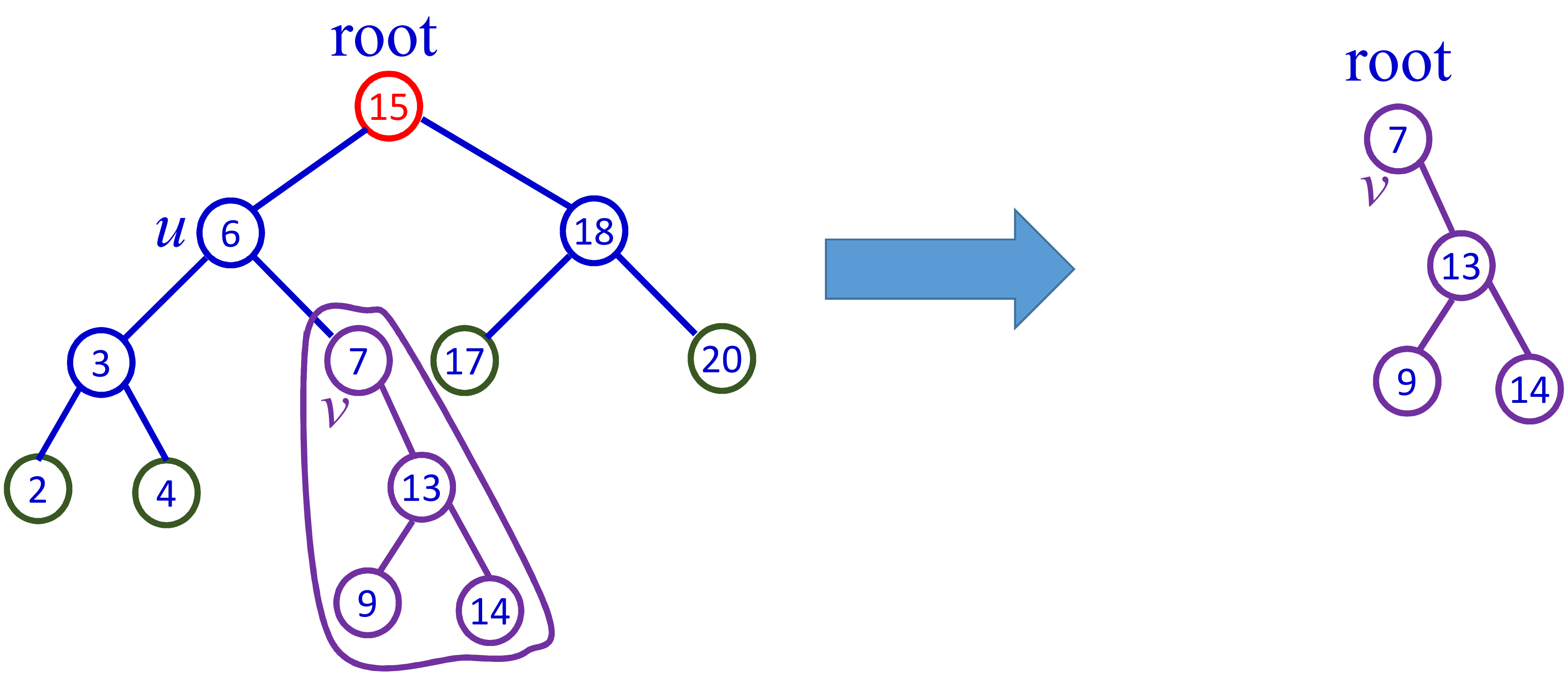




We changed one child of v

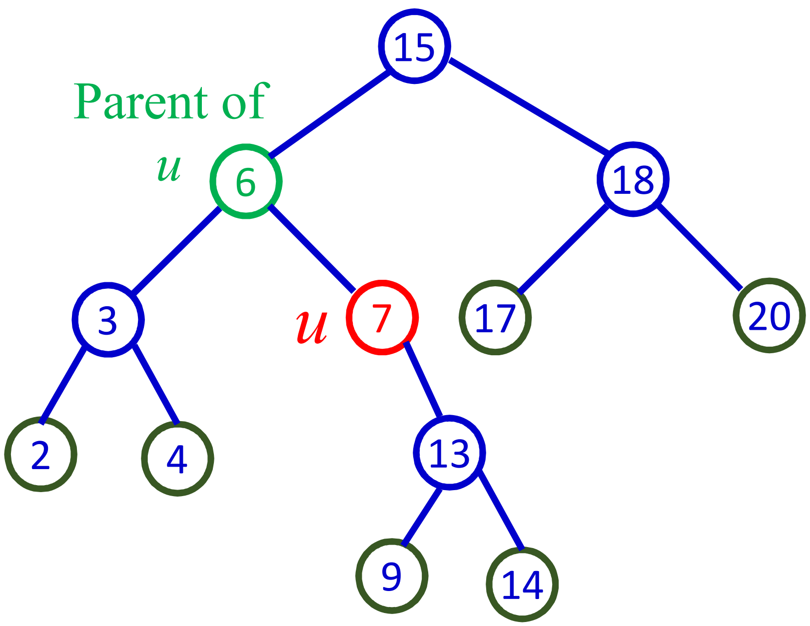
BST Operation: Deletion (2)

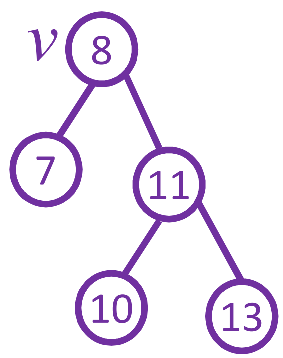
Even we can replace the root



BST Operation: Deletion (2)

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



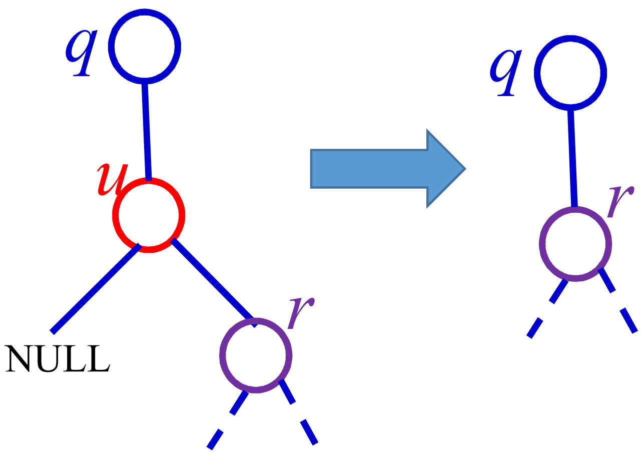



```
TRANSPLANT(T, u, v)
if u->parent == NULL          //special case
      T->root = v
elseif u = = u->parent->left   //set appropriate child
      u->parent->left = v
else u->parent->right =v
if v ≠ NULL                   //set parent
   v->parent = u->parent
```

BST Operation: Deletion (2)

Node Deletion Cases

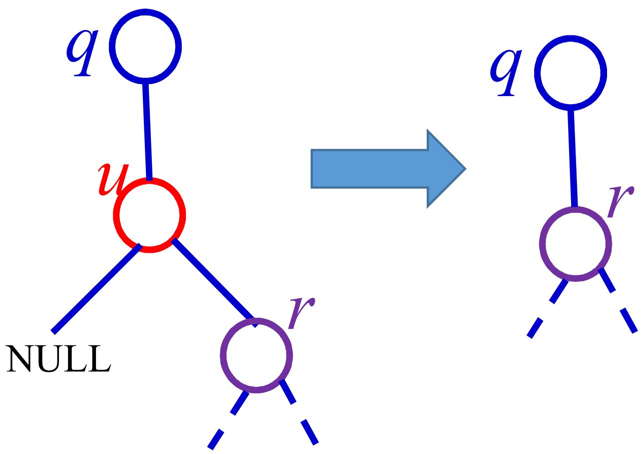
Node u has **NO LEFT** child



BST Operation: Deletion (2)

Node Deletion Cases

Node u has **NO LEFT** child

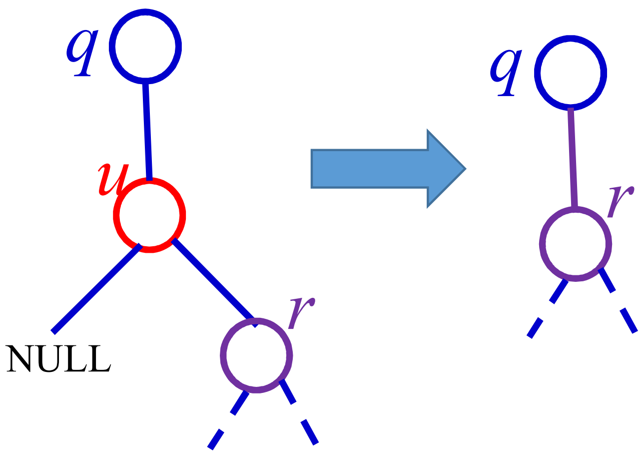


This also covers if both children are empty

BST Operation: Deletion (2)

Node Deletion Cases

Node u has **NO LEFT** child

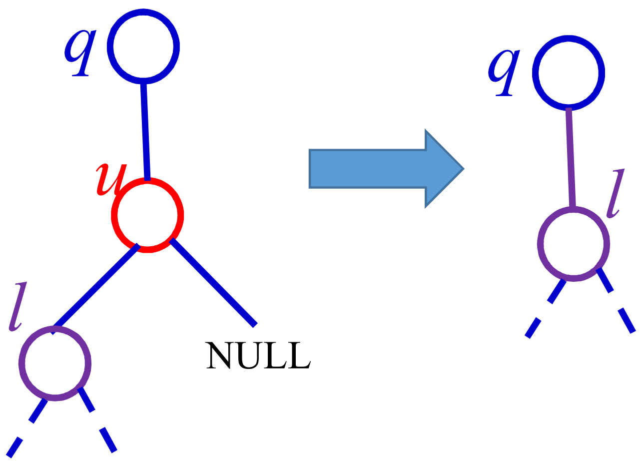



```
TREE_DELETE (T, u)
if u->left == NULL
      TRANSPLANT(T, u, u->right)
```

BST Operation: Deletion (2)

Node Deletion Cases

Node *u* has **NO RIGHT** child

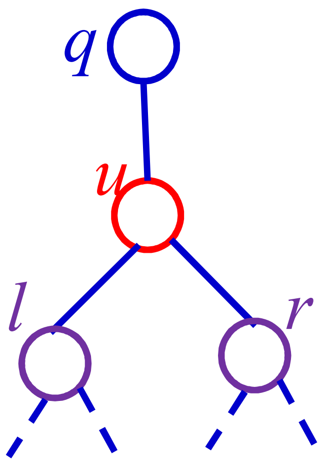


TREE_DELETE (T, u)
1 **if** *u->left* == NULL
2 TRANSPLANT(T, u, *u->right*)
3 **elseif** *u->right* == NULL
4 TRANSPLANT (T, u, *u->left*)

BST Operation: Deletion (2)

Node Deletion Cases

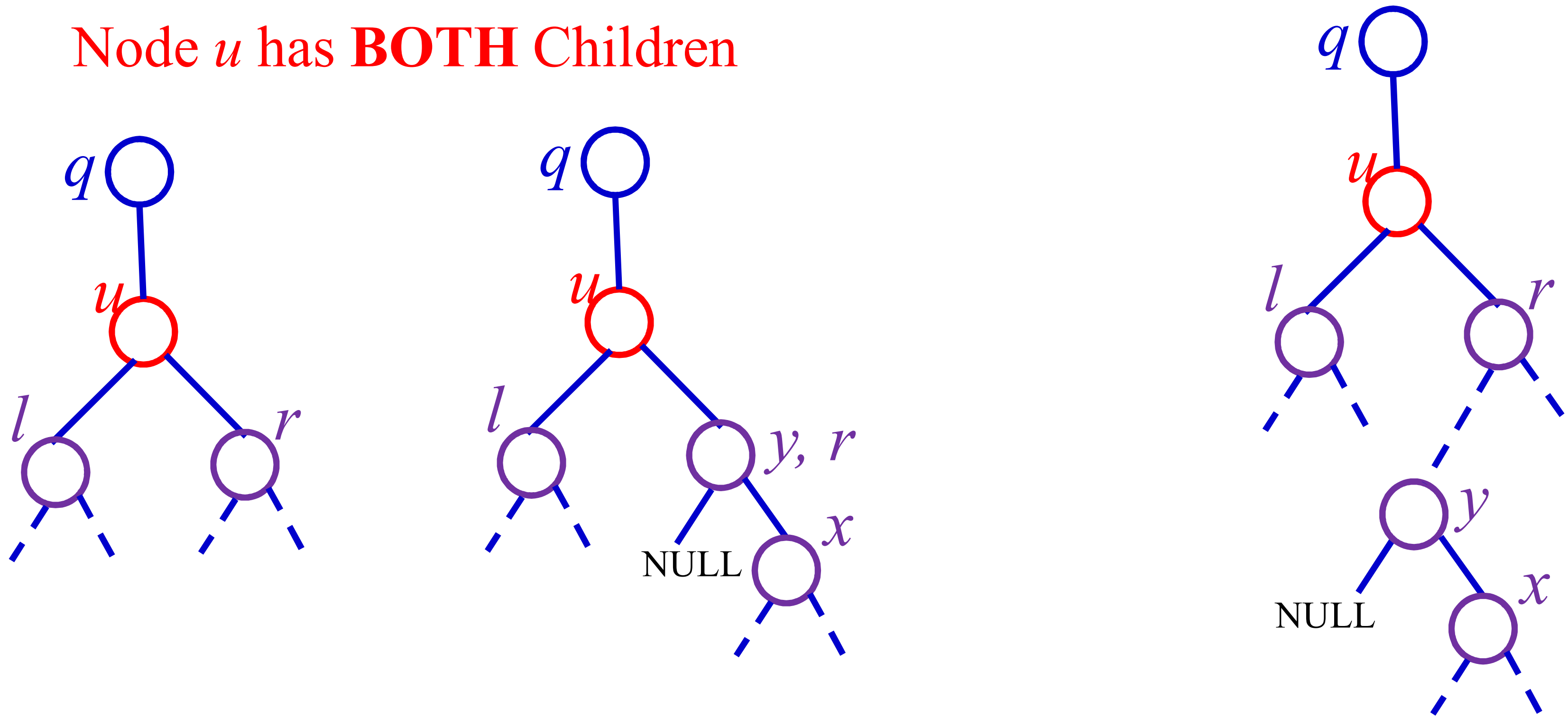
Node u has **BOTH** Children



BST Operation: Deletion (2)

Node Deletion Cases

Node *u* has **BOTH** Children

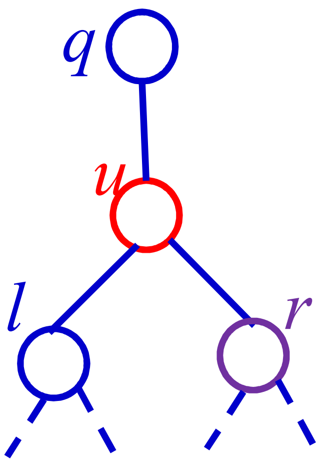


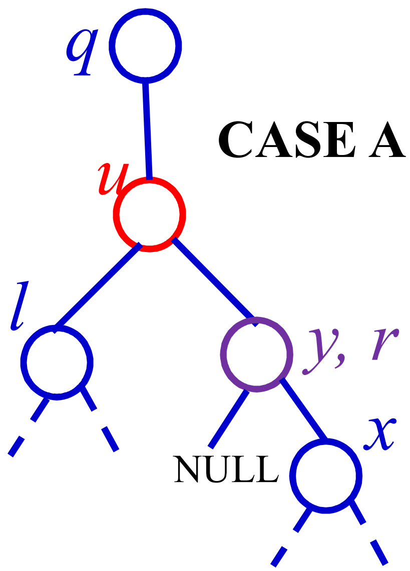
Find *y* = successor (next minimum) from RIGHT subtree

BST Operation: Deletion (2)

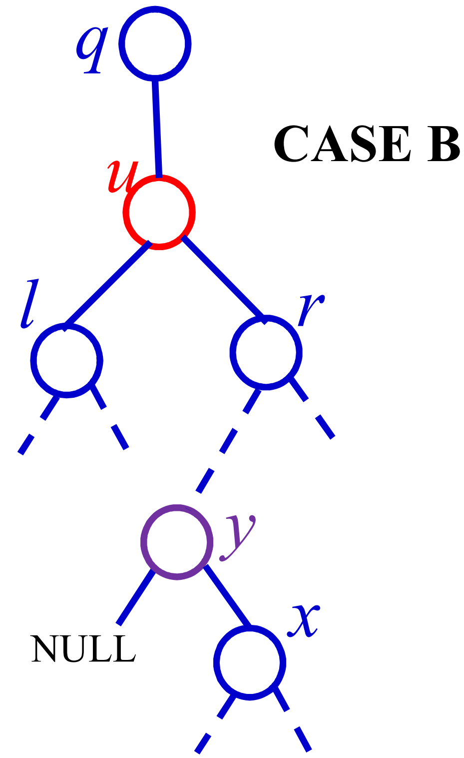
Node Deletion Cases

Node u has **BOTH** Children





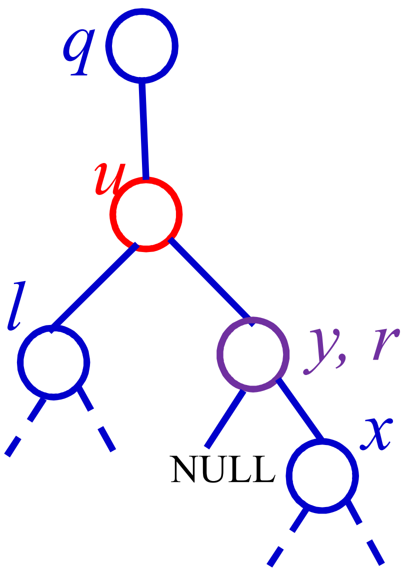
y is immediate RIGHT child of u



y is NOT immediate child of u

BST Operation: Deletion (2)

CASE A



y is immediate RIGHT child of *u*

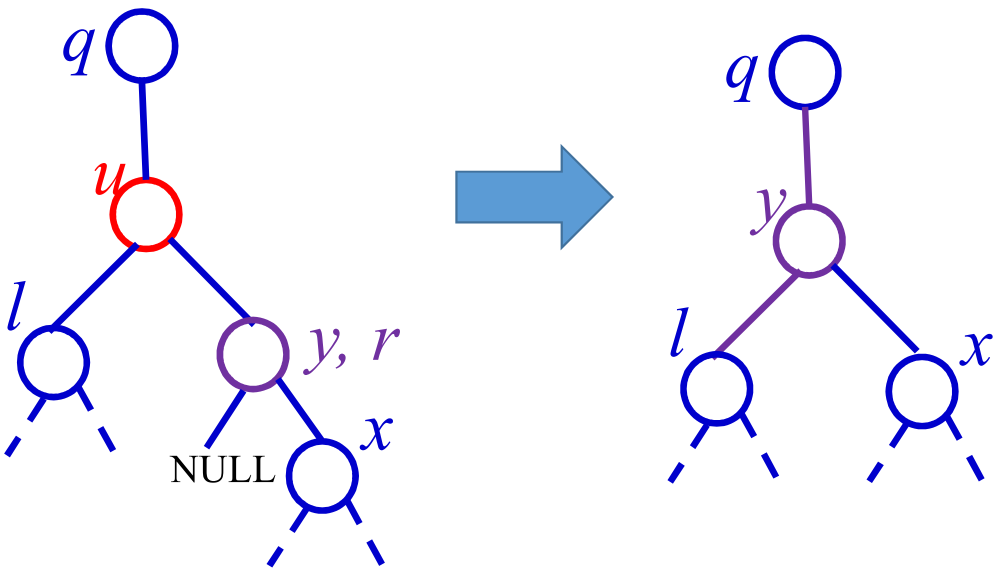
```
TREE_DELETE (T, u)
if u->left == NULL
    TRANSPLANT(T, u, u->right)
elseif  u->right == NULL
    TRANSPLANT (T, u, u->left)
else y = TREE_MINIMUM(u->right)
```

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*

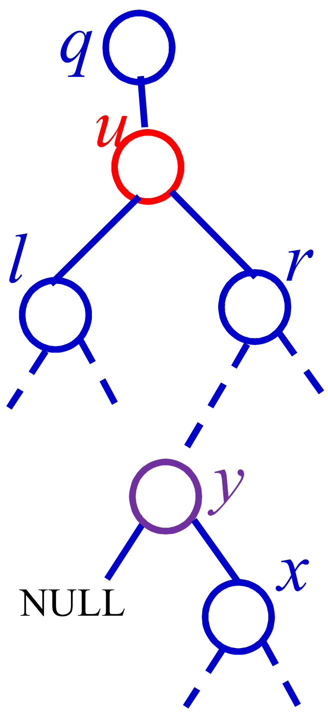
```
TREE_DELETE (T, u)
1 if u->left == NULL
2       TRANSPLANT(T, u, u->right)
3 elseif   u->right == NULL
4        TRANSPLANT (T, u, u->left)
5 else y = TREE_MINIMUM(u->right)
```

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

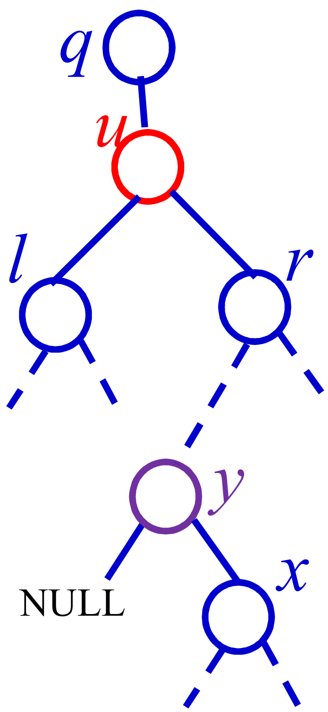


y is **NOT** immediate child of u

BST Operation: Deletion (2)

CASE B

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

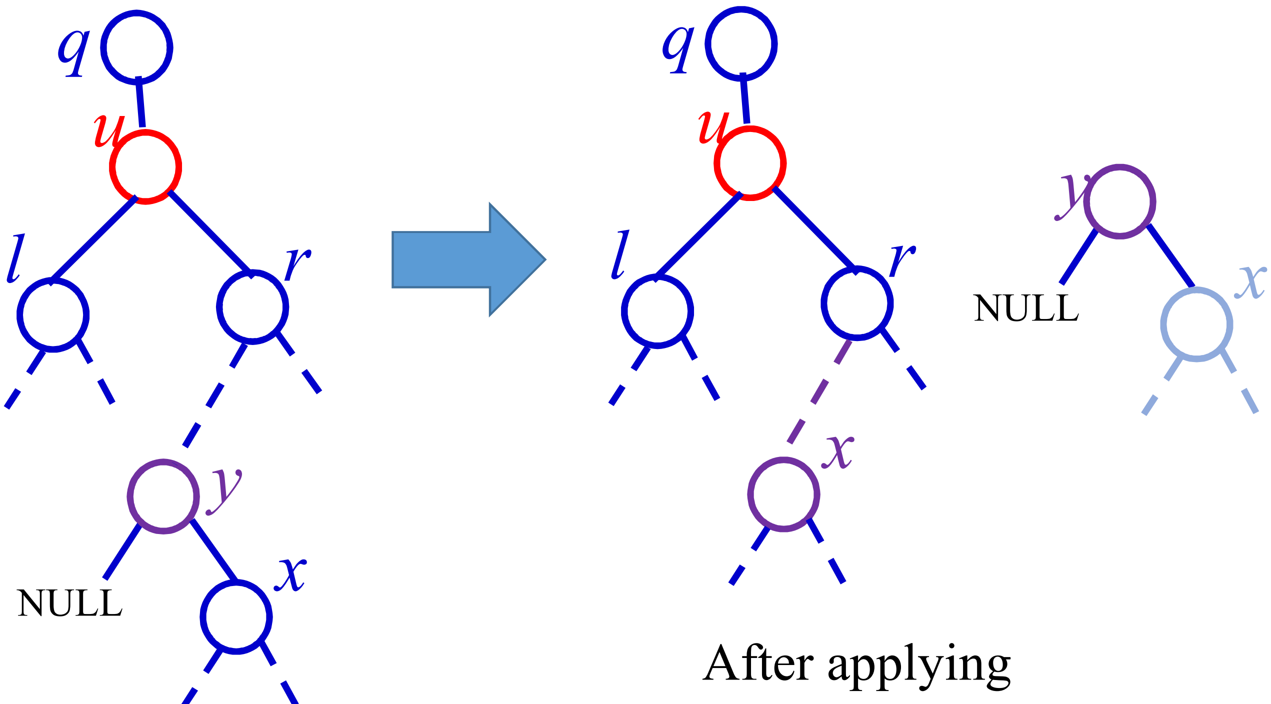


y is **NOT** immediate child of u

BST Operation: Deletion (2)

CASE B

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

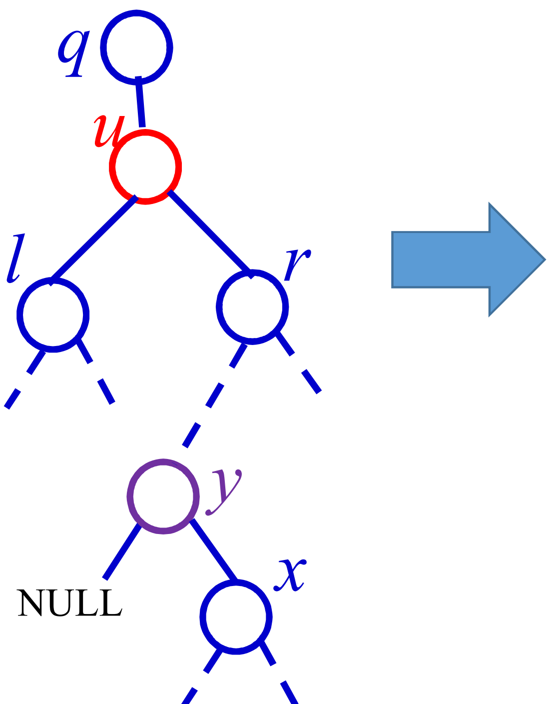


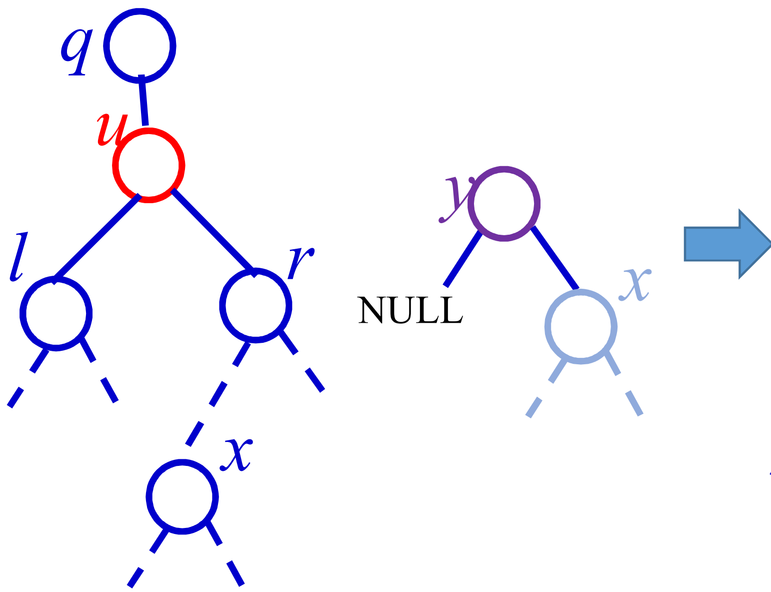
After applying
TRANSPLANT(*T*, *y*, *y->right*)

BST Operation: Deletion (2)

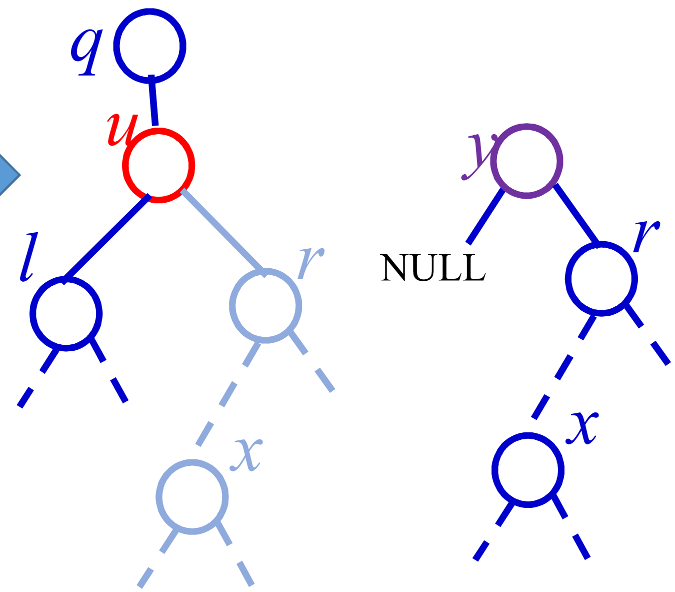
CASE B

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





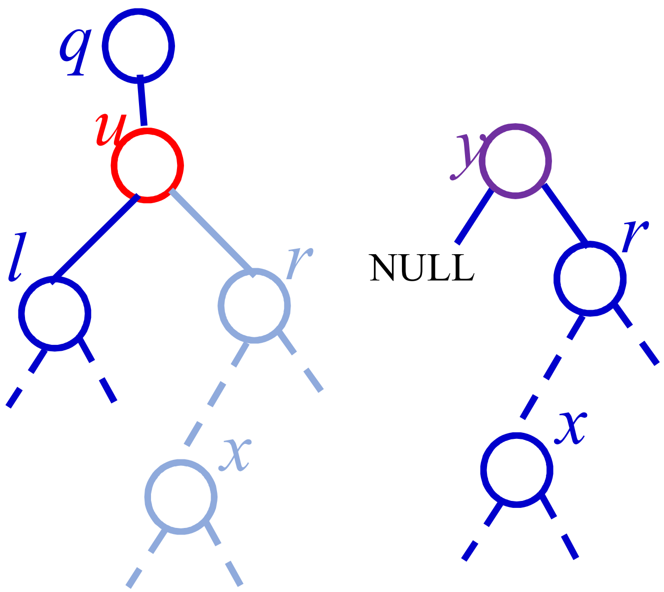
After applying
TRANSPLANT(T, y, y->$right$)



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

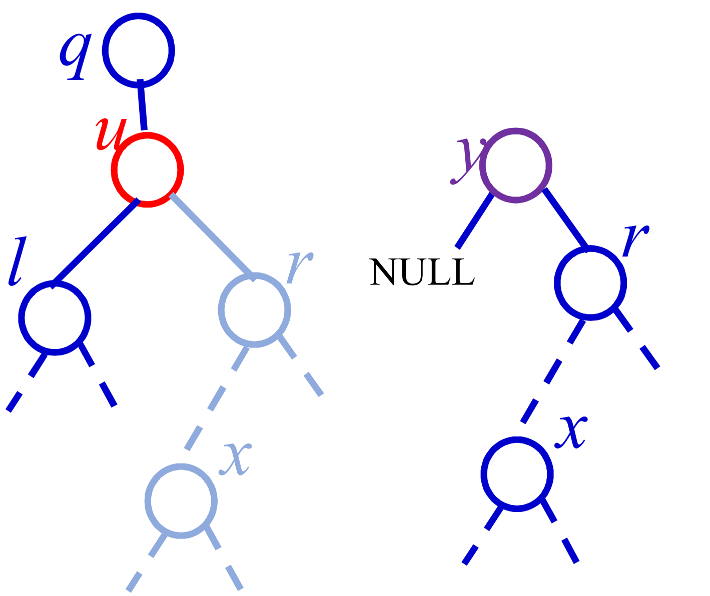
BST Operation: Deletion (2)

CASE B

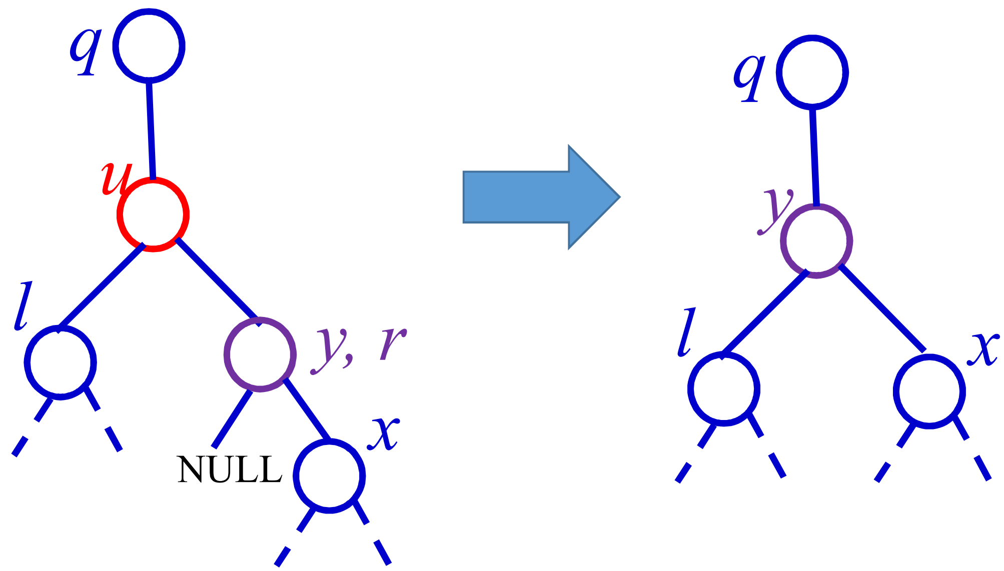


BST Operation: Deletion (2)

CASE B Outcome



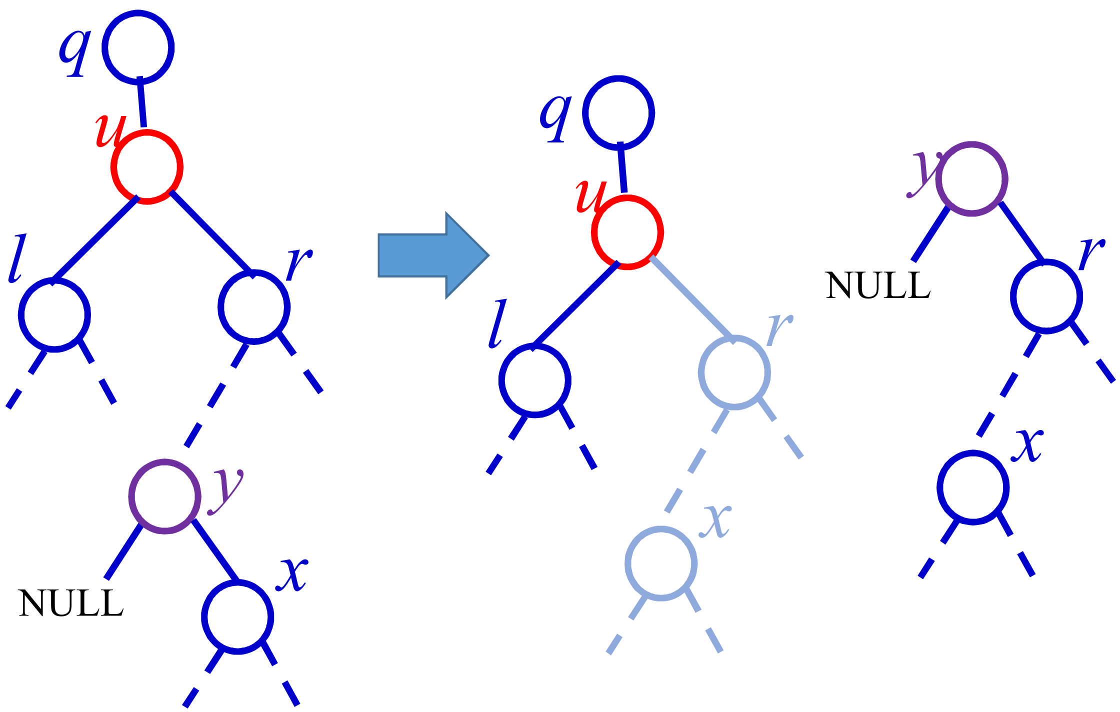
CASE A



y is immediate RIGHT child of *u*

BST Operation: Deletion (2)

CASE B



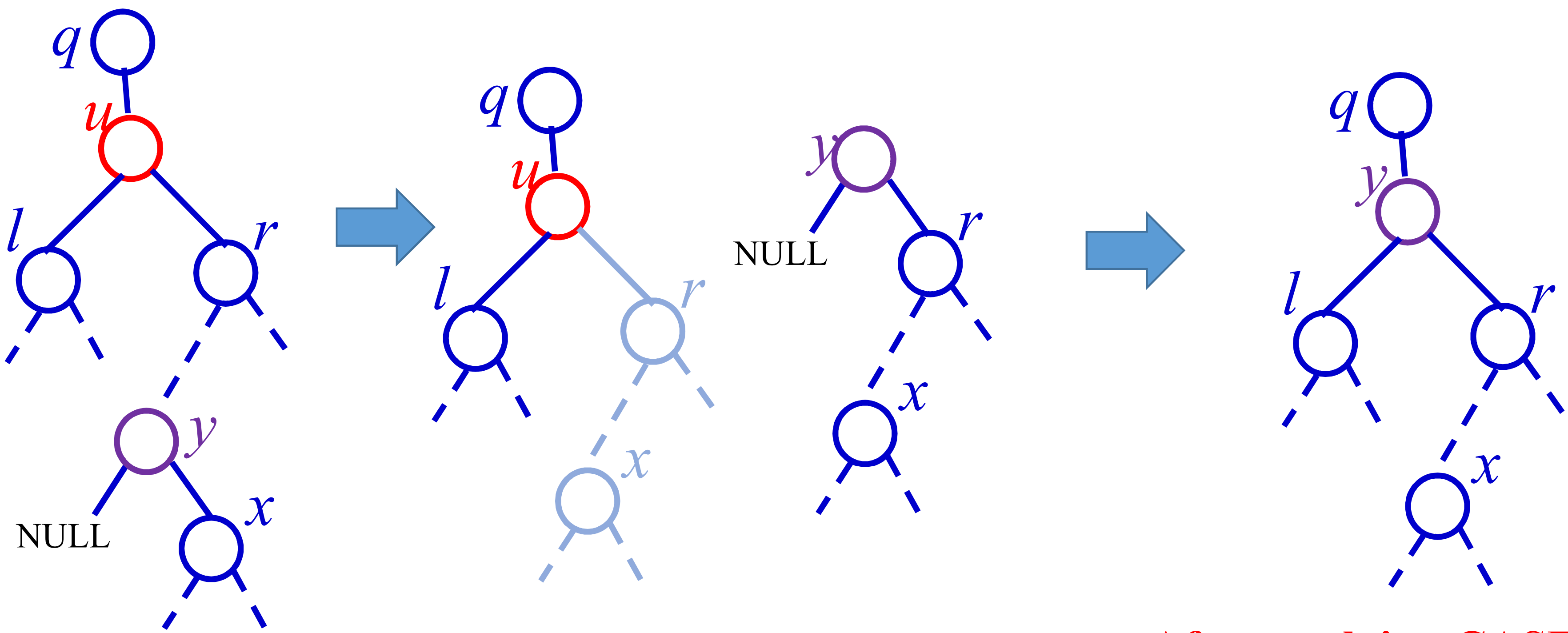
Starting state

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

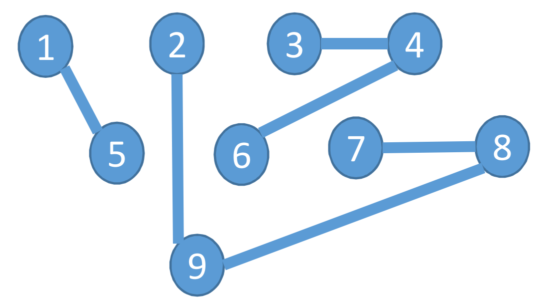
- Pros:
  - Simple to implement
  - Easy and fast to tell if a pair (i, j) is an edge: simply check if A[i, j] is 1 or 0
  - Can be very efficient for small graphs
  - Good for dense graphs (why?)
- Cons:
  - No matter how few edges the graph has, the matrix takes $O(n^2)$, i.e., $O(|V|^2)$ in memory

$|V| = n$

$|V| = n$

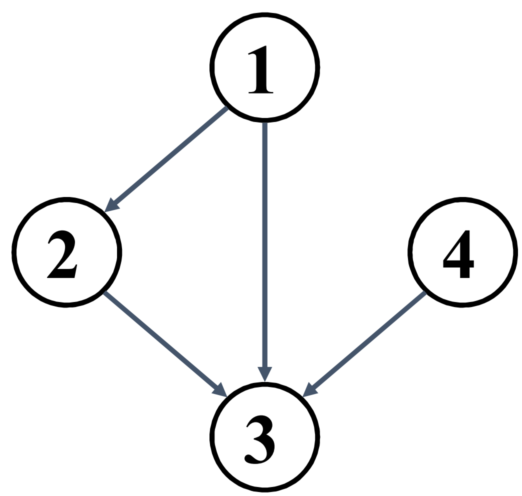
	1	2	3	4	5	6	7	8	9
1	0	0	0	0	1	0	0	0	0
2	0	0	0	0	0	0	0	0	1
3	0	0	0	1	0	0	0	0	0
4	0	0	1	0	0	1	0	0	0
5	1	0	0	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	1	0	1
9	0	1	0	0	0	0	0	1	0

$n \times n$ matrix

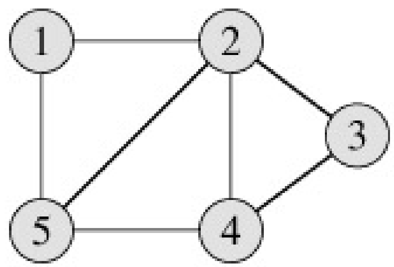


Adjacency Lists Representation

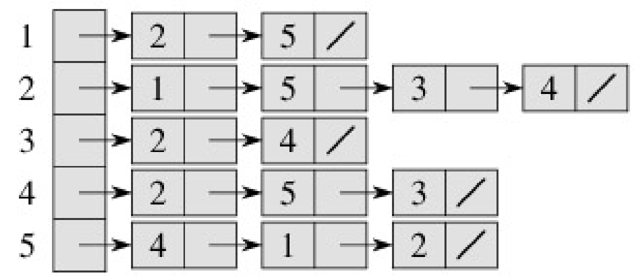
- A graph is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent to node i.
  - The nodes in the list L[i] are in NO particular order

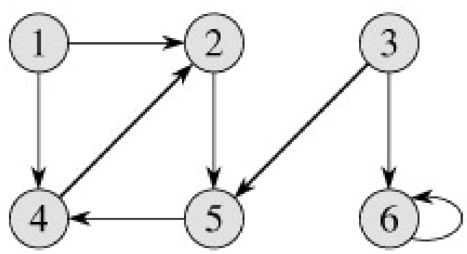


Adjacency Lists Representation

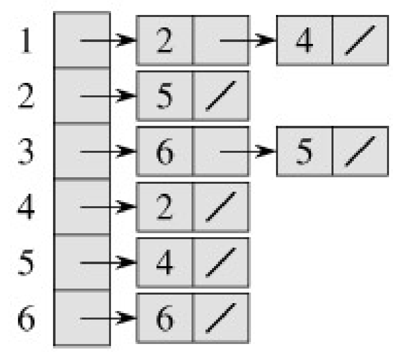


Undirected Graph





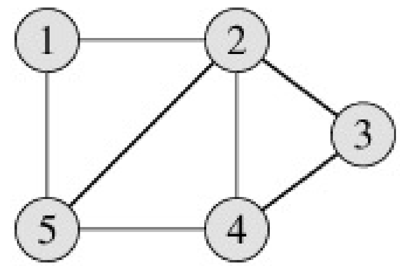
Directed Graph



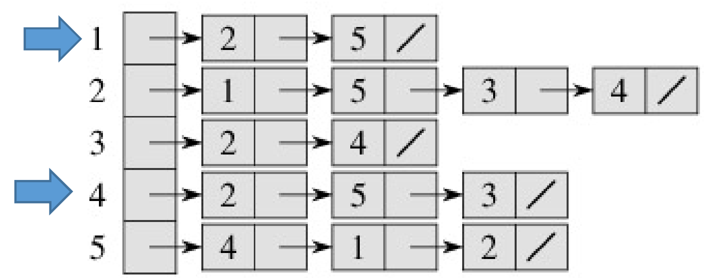
Adjacency Lists Representation

- 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

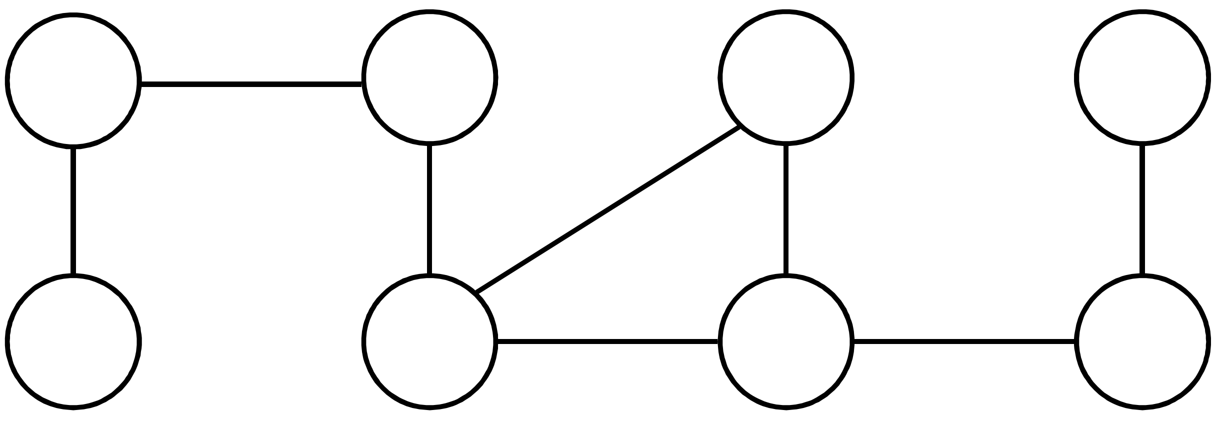
- There are two standard graph traversal techniques:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

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.

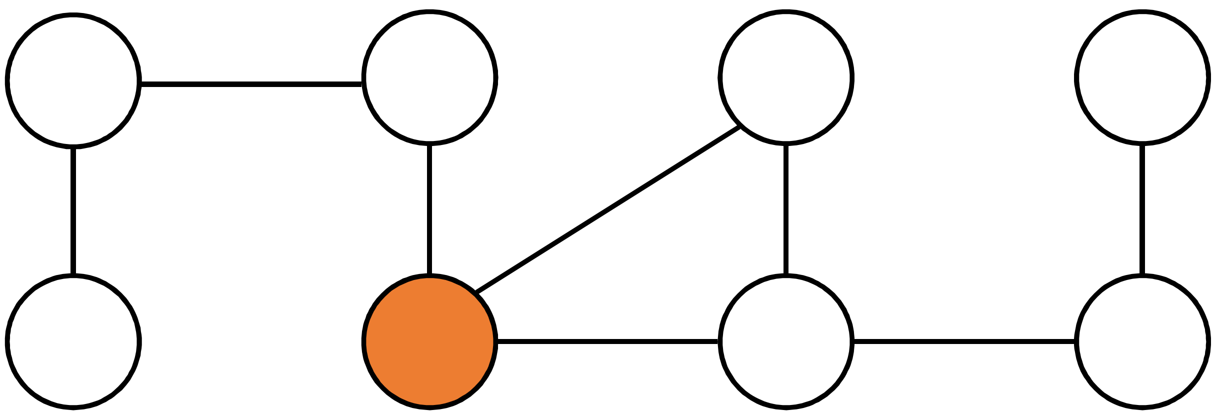
Breadth-First Search

- “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.



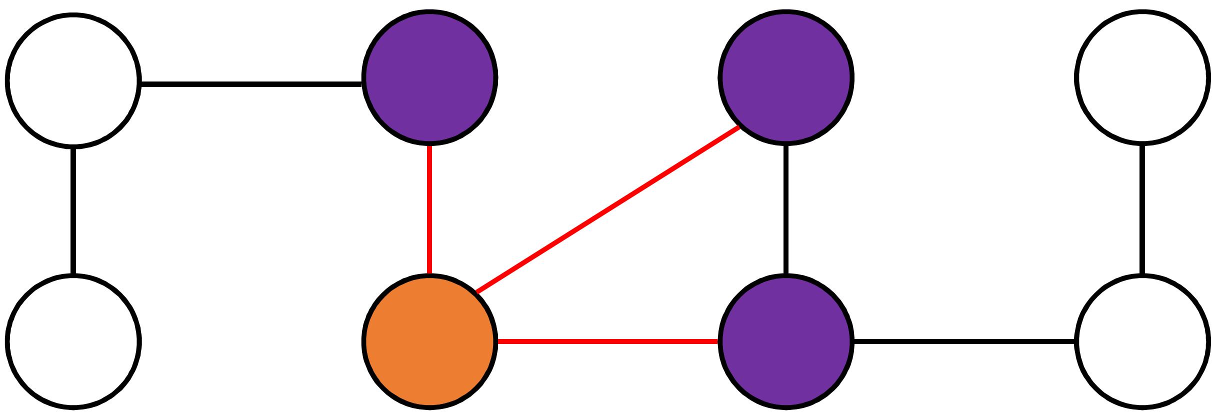
Breadth-First Search

- "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.



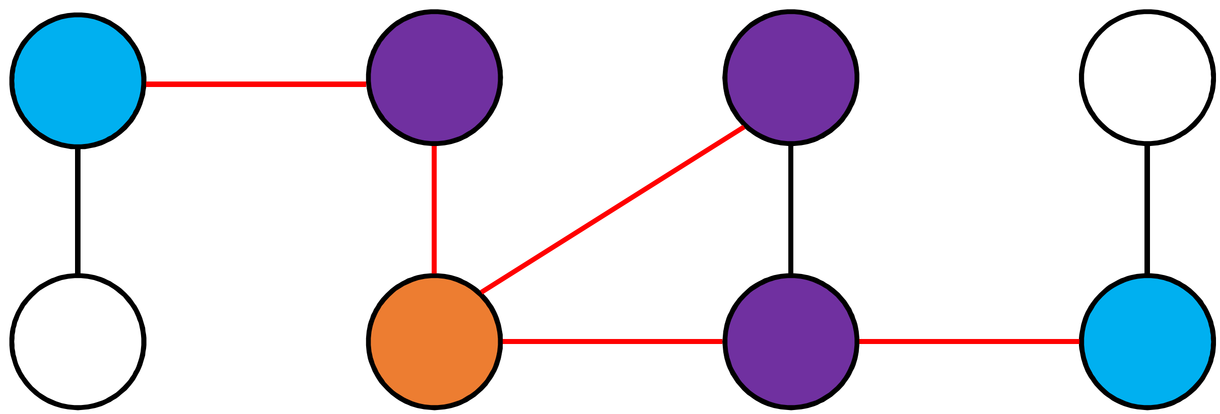
Breadth-First Search

- "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.



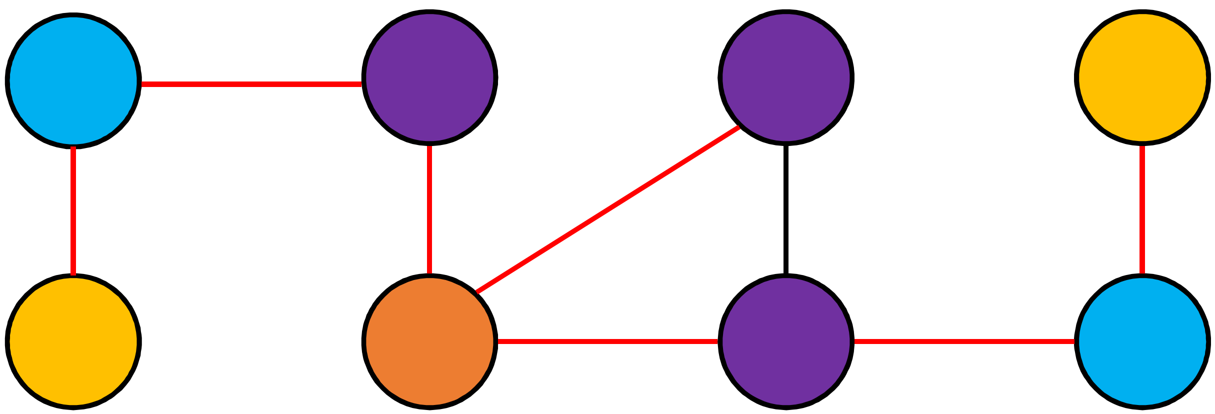
Breadth-First Search

- “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.



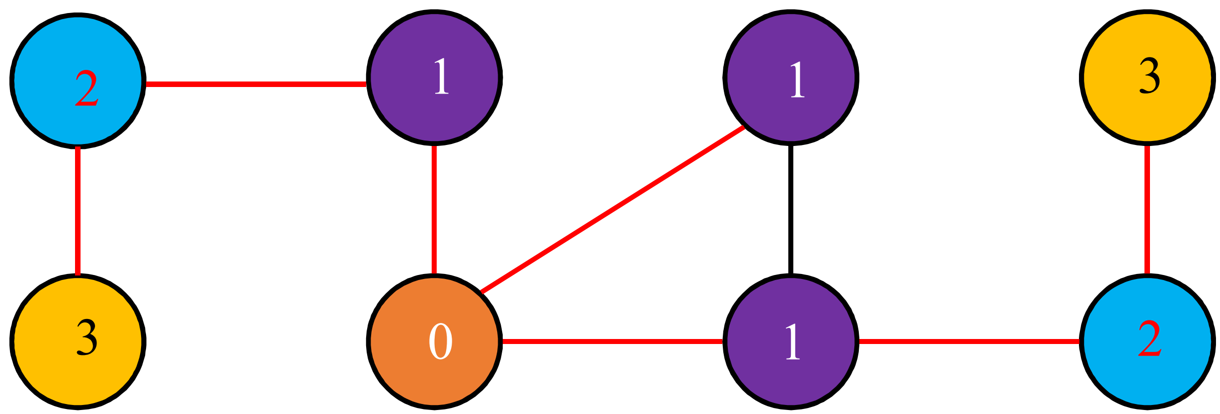
Breadth-First Search

- "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

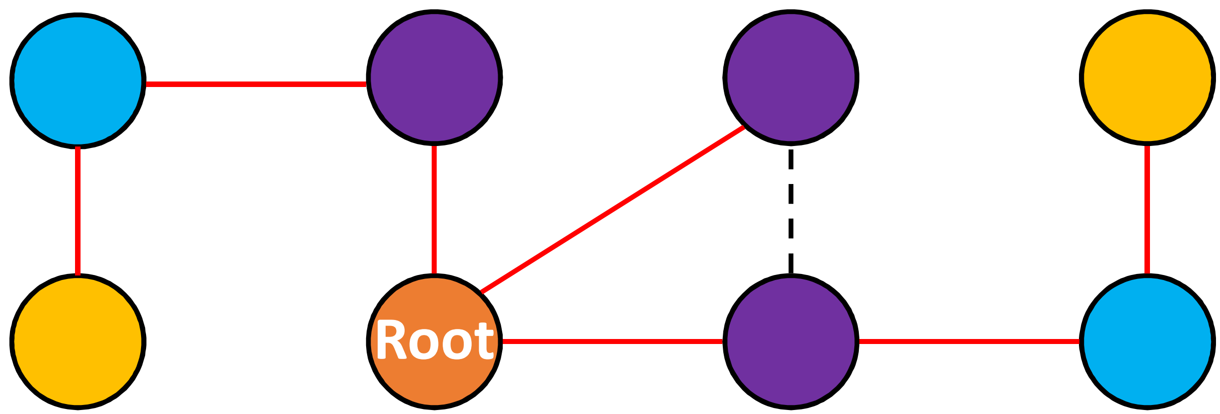
- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



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.

Tree (dotted line removed)



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