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

Instructor

Dr Md Monirul Islam

BST Operation: Deletion

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

There are three possible scenarios:

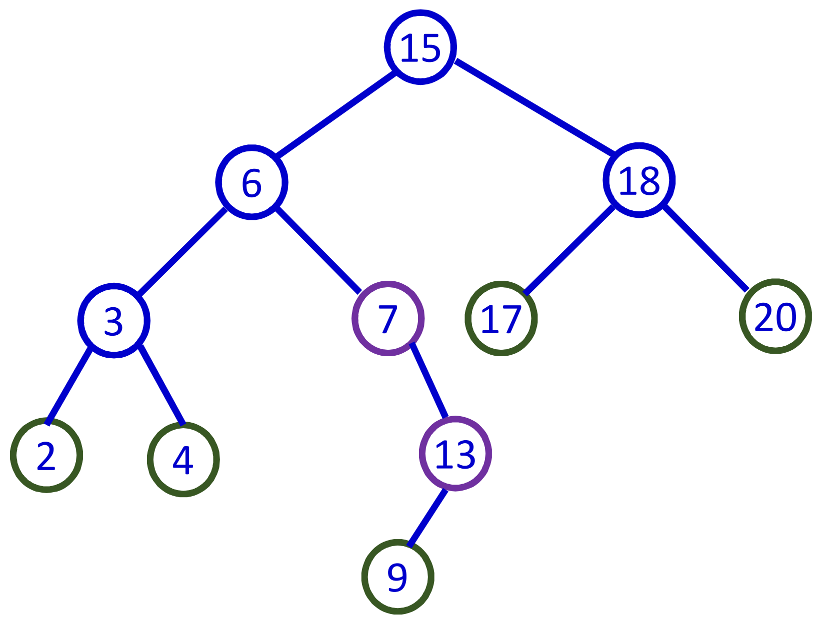
- The node is a leaf node
- It has exactly one child, or
- It has two children (it is a full node)

BST Operation: Deletion

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

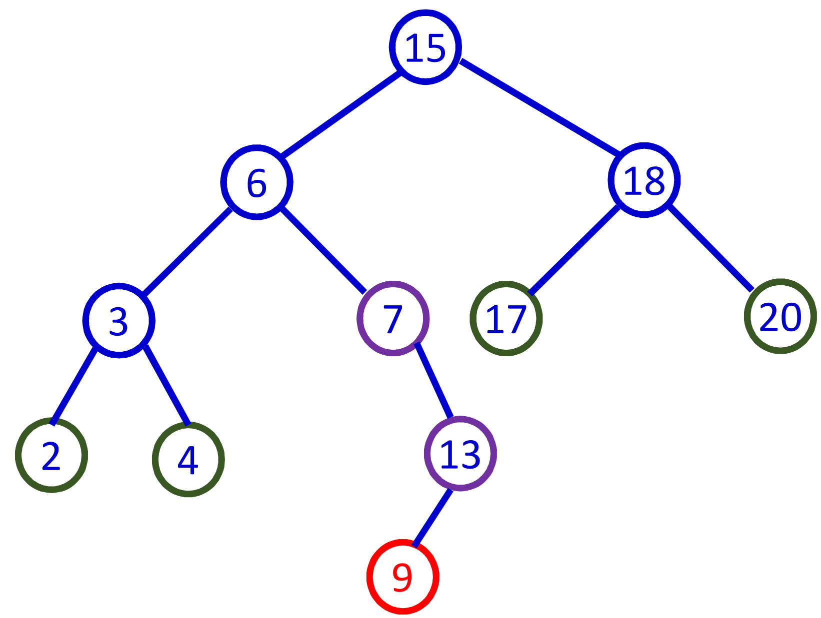
There are three possible scenarios:

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



BST Operation: Deletion

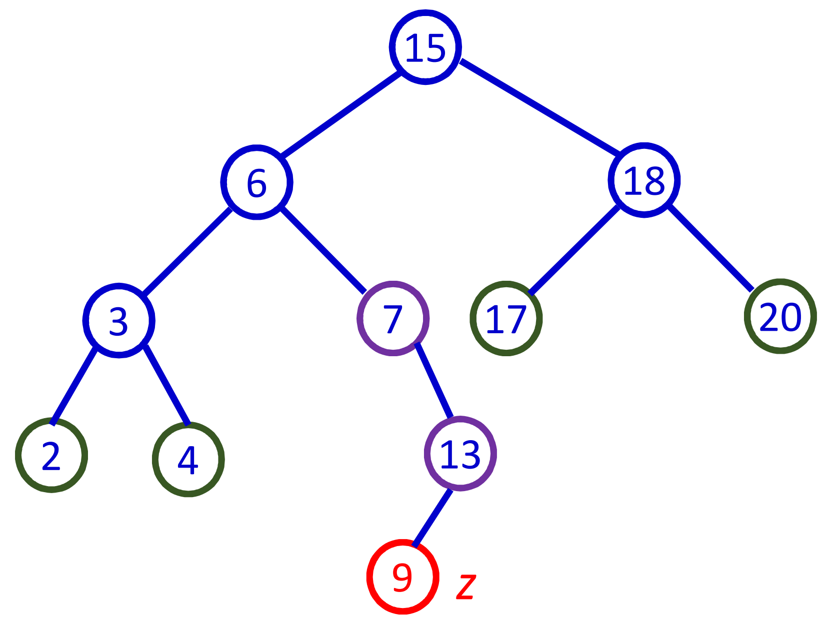
Removing a leaf node



BST Operation: Deletion

Removing a leaf node
Set left pointer of 13 as NULL

If $z == z \rightarrow parent \rightarrow left$
$z \rightarrow parent \rightarrow left = \text{NULL}$
else $z \rightarrow parent \rightarrow right = \text{NULL}$



BST Operation: Deletion

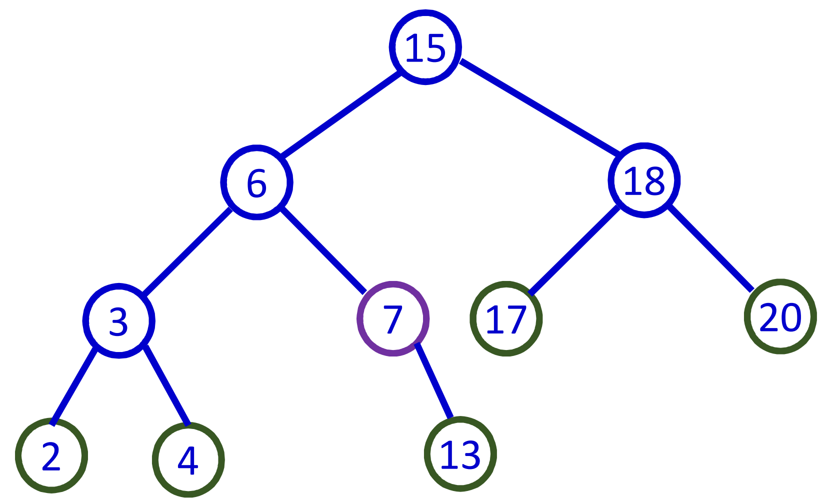
Removing a leaf node

Set left pointer of 13 as NULL

If $z == z \rightarrow parent \rightarrow left$

$z \rightarrow parent \rightarrow left$ =NULL

else $z \rightarrow parent \rightarrow right$ =NULL



BST Operation: Deletion

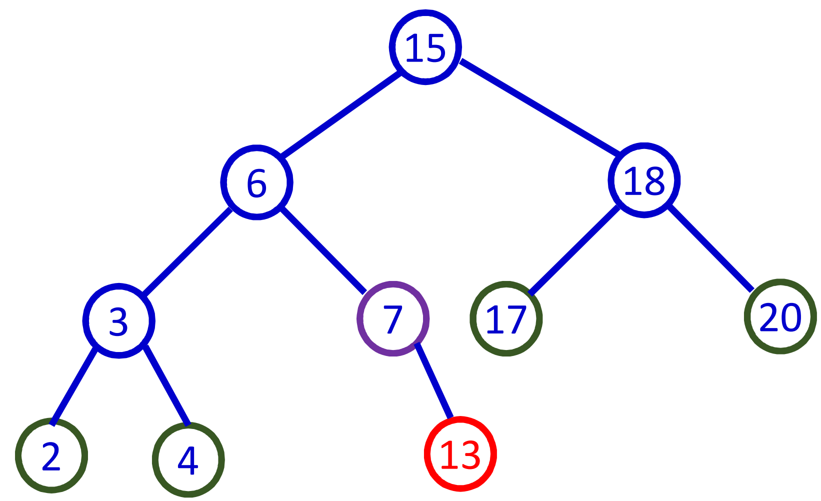
Removing a leaf node with key 13

Set right pointer of 7 as NULL

If $z == z \rightarrow parent \rightarrow left$

$z \rightarrow parent \rightarrow left$ =NULL

else $z \rightarrow parent \rightarrow right$ =NULL



BST Operation: Deletion

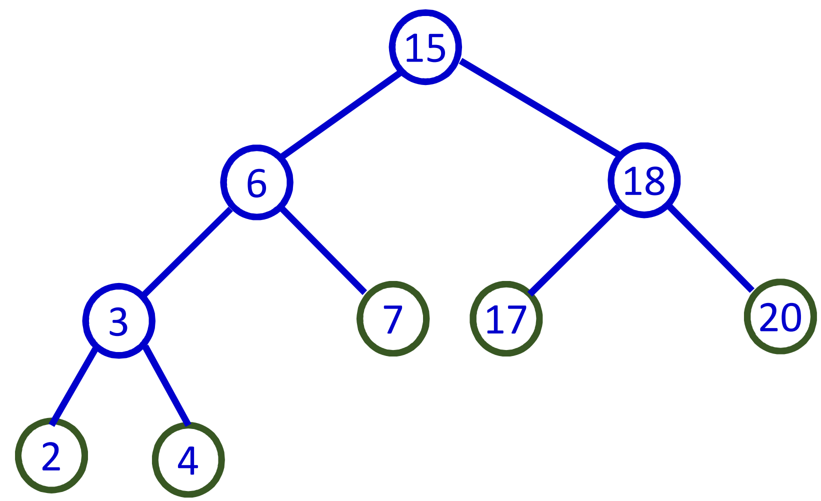
Removing a leaf node with key 13

Set right pointer of 7 as NULL

If $z == z \rightarrow parent \rightarrow left$

$z \rightarrow parent \rightarrow left = \text{NULL}$

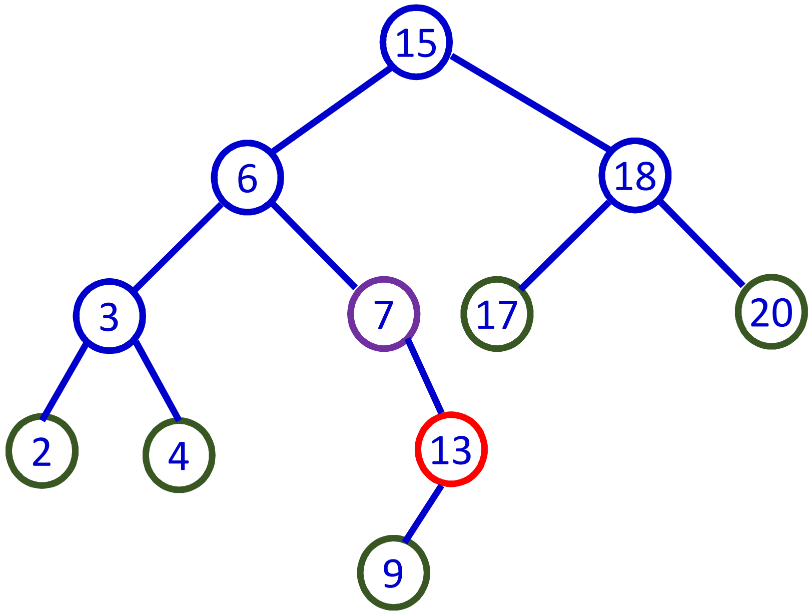
else $z \rightarrow parent \rightarrow right = \text{NULL}$



BST Operation: Deletion

Removing a node with exactly **one child**

Remove node with key 13 which has a left subtree ONLY



BST Operation: Deletion

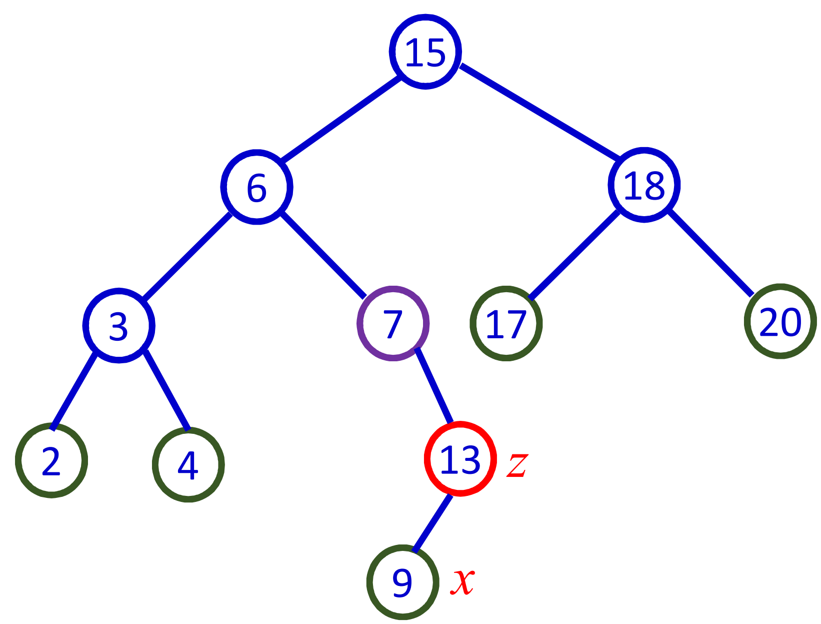
Removing a node with exactly one child

1. Remove node with key 13 which has a left subtree ONLY
2. Promote the left subtree

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

Locate the child

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



BST Operation: Deletion

Removing a node with exactly one child

1. Remove node with key 13 which has a left subtree ONLY
2. Promote the left subtree

If *z->left* =NULL

x= z->right

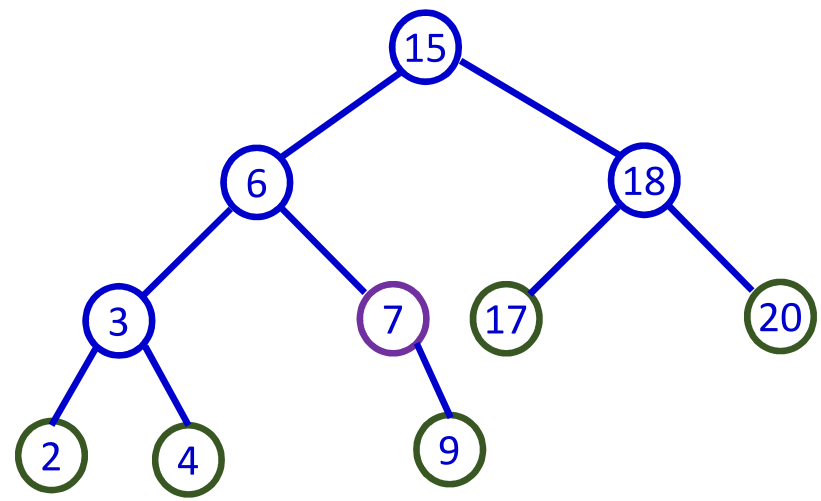
else *x= z->left*

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

z->parent ->left =x

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

x->parent = z->parent



BST Operation: Deletion

Removing a node with exactly one child

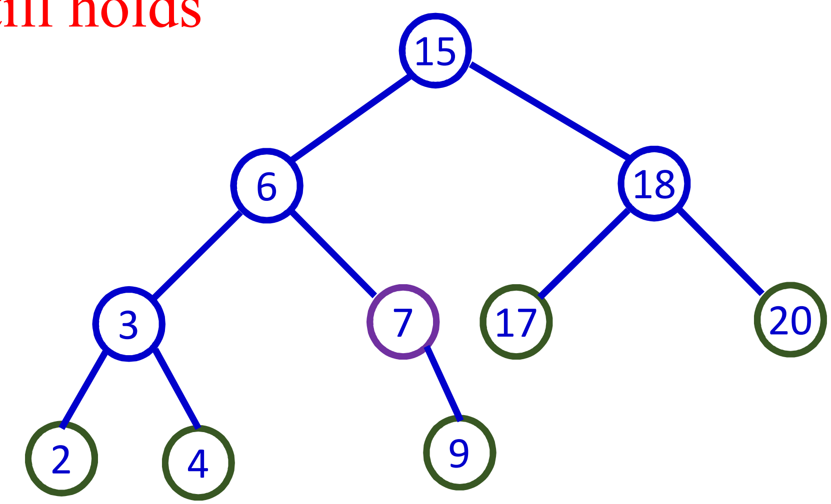
Remove node with key 13 which has a left subtree ONLY

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 property still holds



BST Operation: Deletion

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

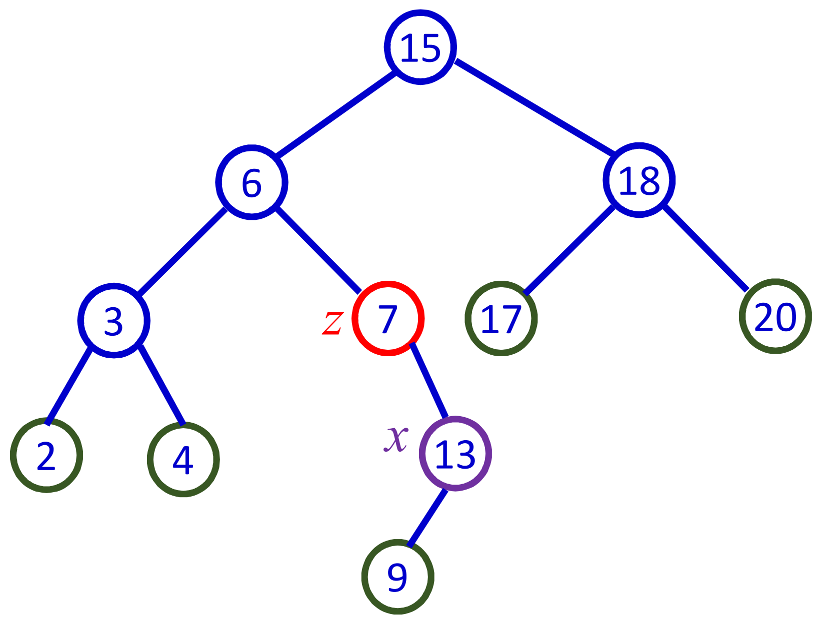
else *x= z->left*

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

z->parent ->left =x

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

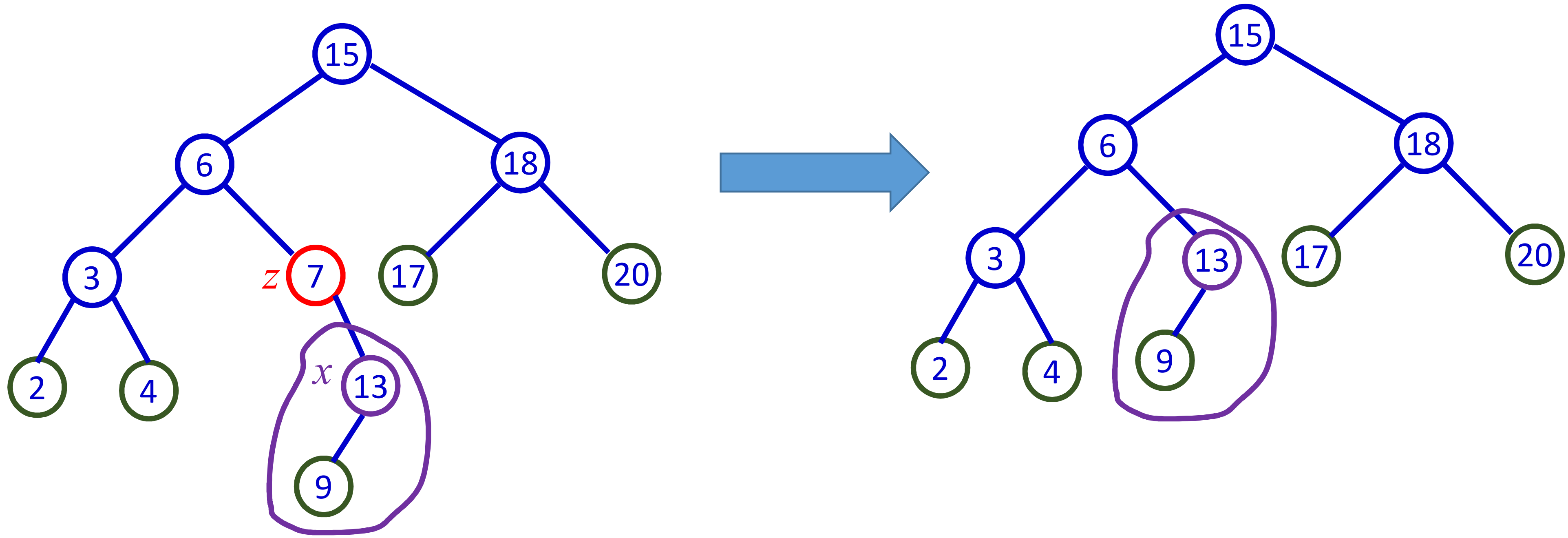
x->parent = z->parent



BST Operation: Deletion

Removing a node with exactly one child

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

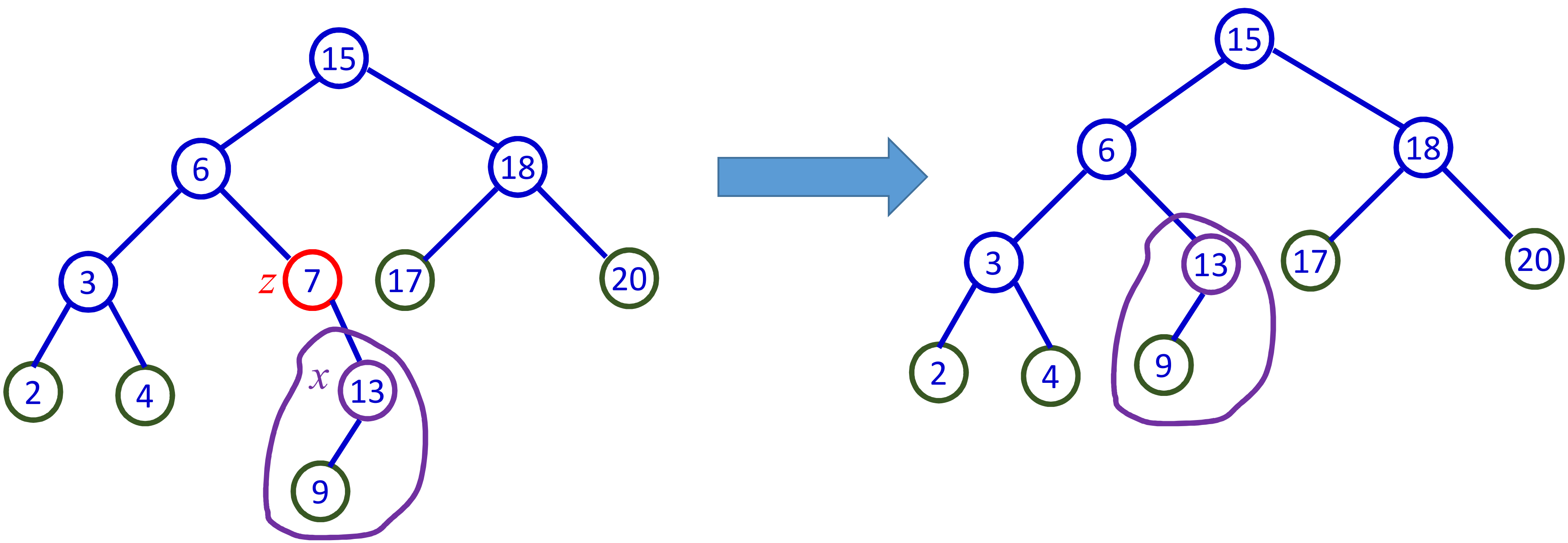


BST Operation: Deletion

Removing a node with exactly one child

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

BST property holds

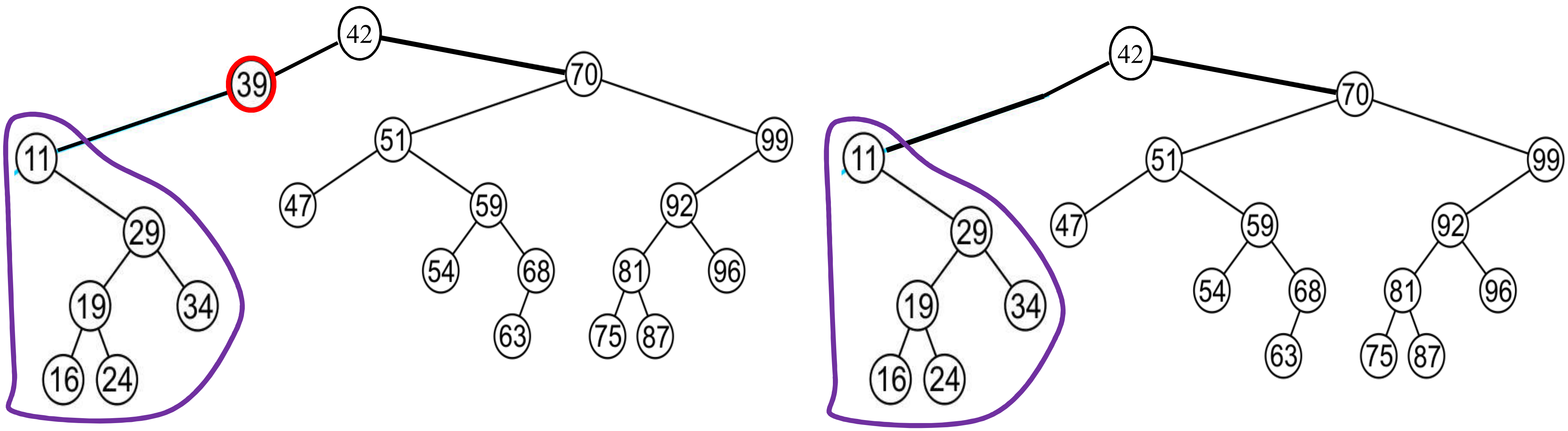


BST Operation: Deletion

Removing a node with exactly one child

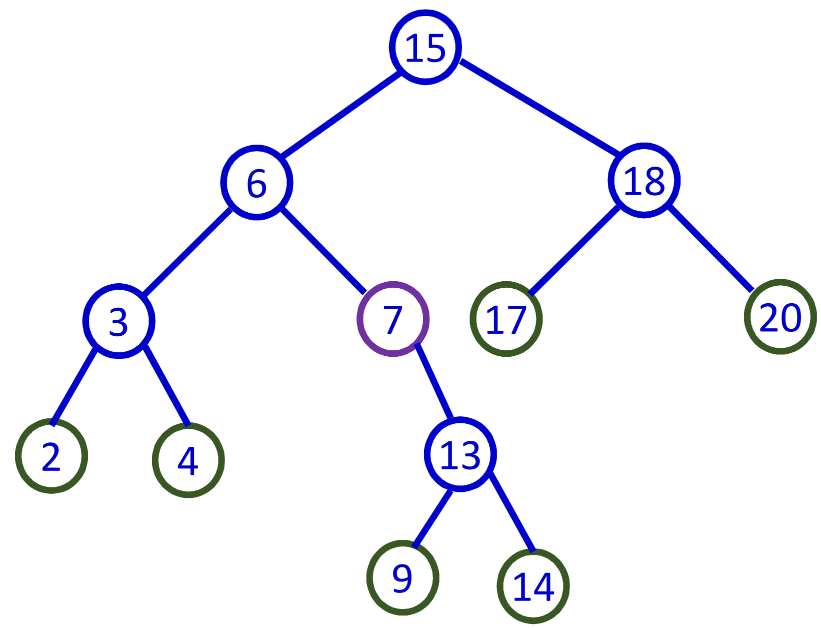
Remove node with key **39** which has a **LEFT** subtree ONLY

BST property holds



BST Operation: Deletion

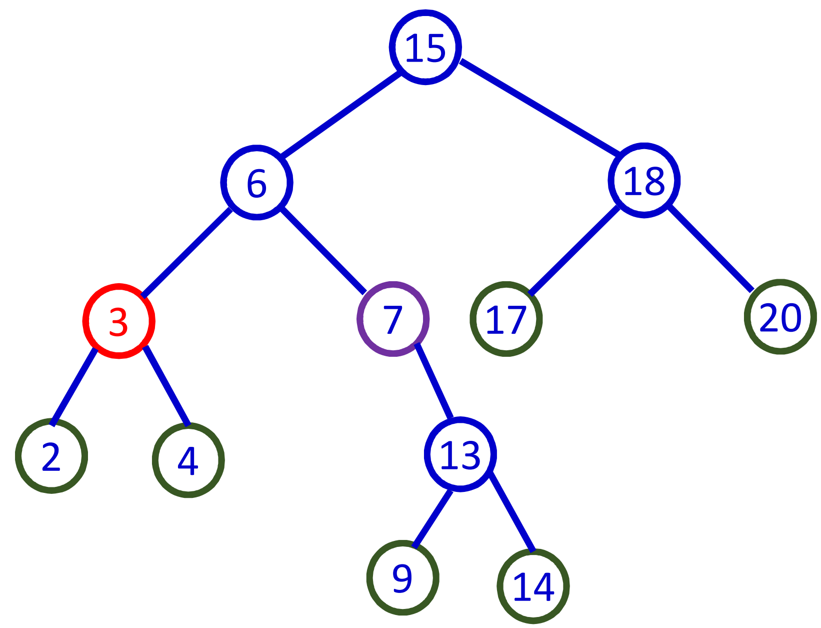
Removing a node having **two children** (full node)



BST Operation: Deletion

Removing a node having **two children** (full node)

Remove node with 3



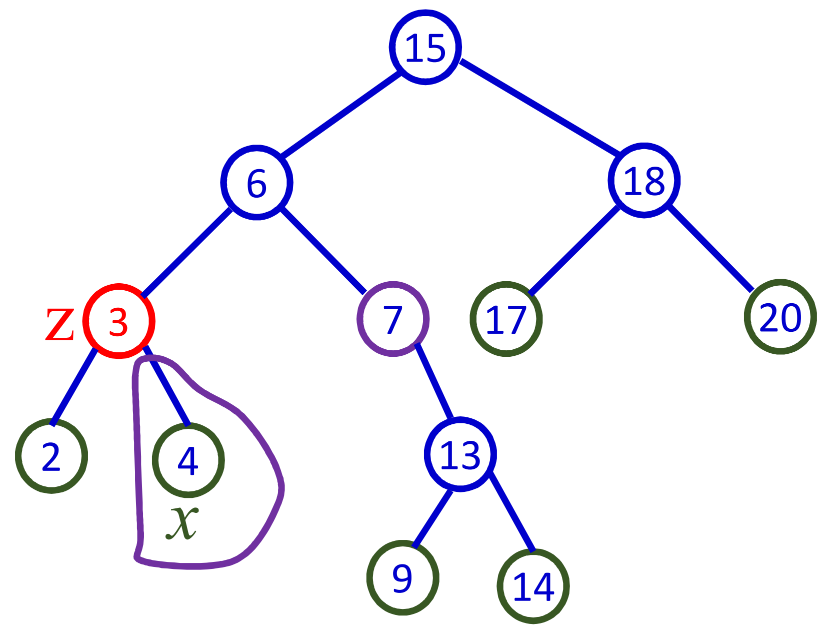
BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 3

Idea:

1. Find the successor of the node with key 3
   the successor must be in right subtree
2. Copy successor's key to node z
3. Delete successor x



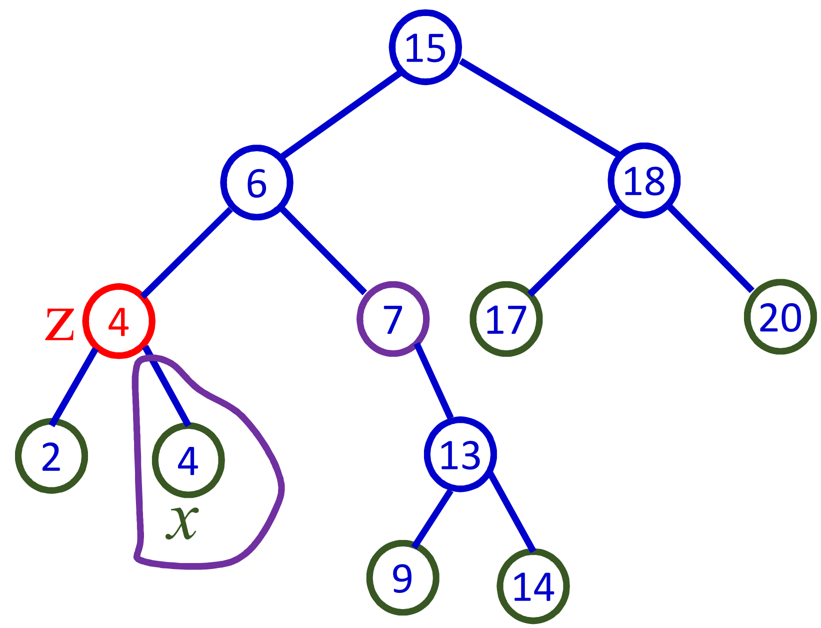
BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 3

Idea:

1. Find the successor of the node with key 3
   the successor must be in right subtree
2. Copy successor's key to node z
3. Delete successor x



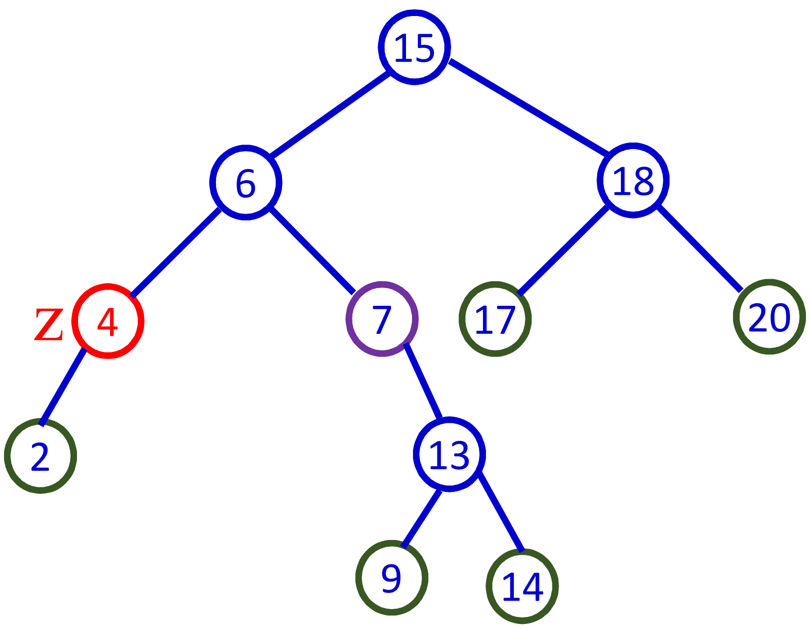
BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 3

Idea:

1. Find the successor of the node with key 3
   the successor must be in right subtree
2. Copy successor's key to node z
3. Delete successor *x*



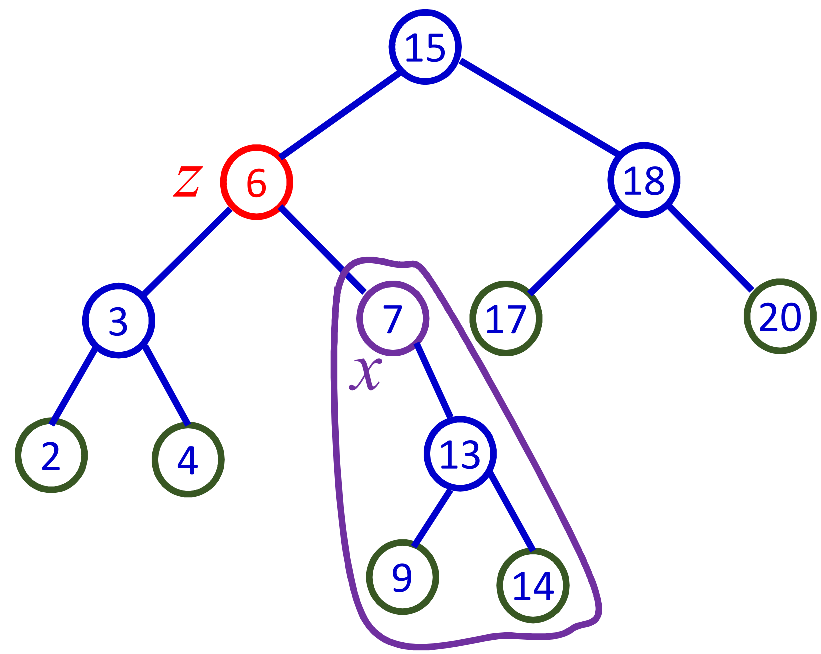
BST Property holds

BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 6

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

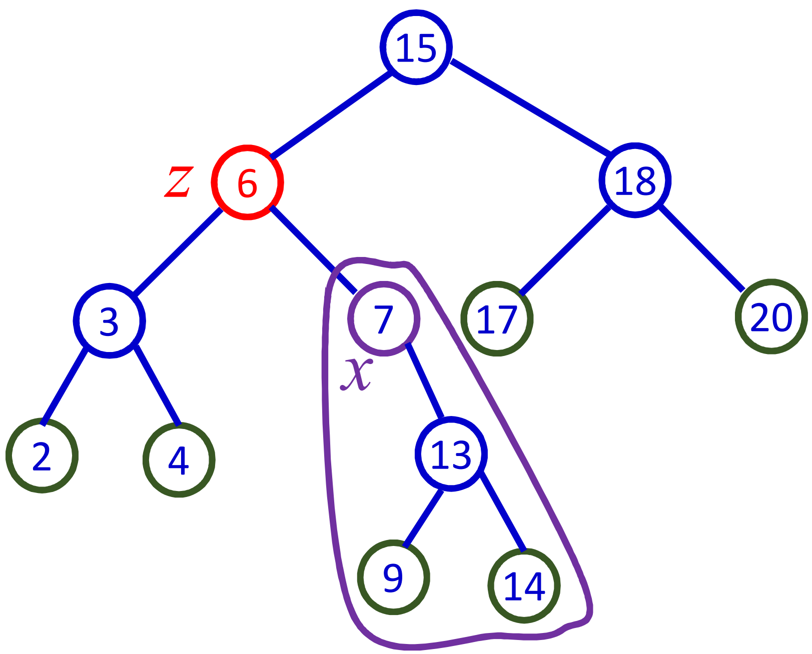


BST Operation: Deletion

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



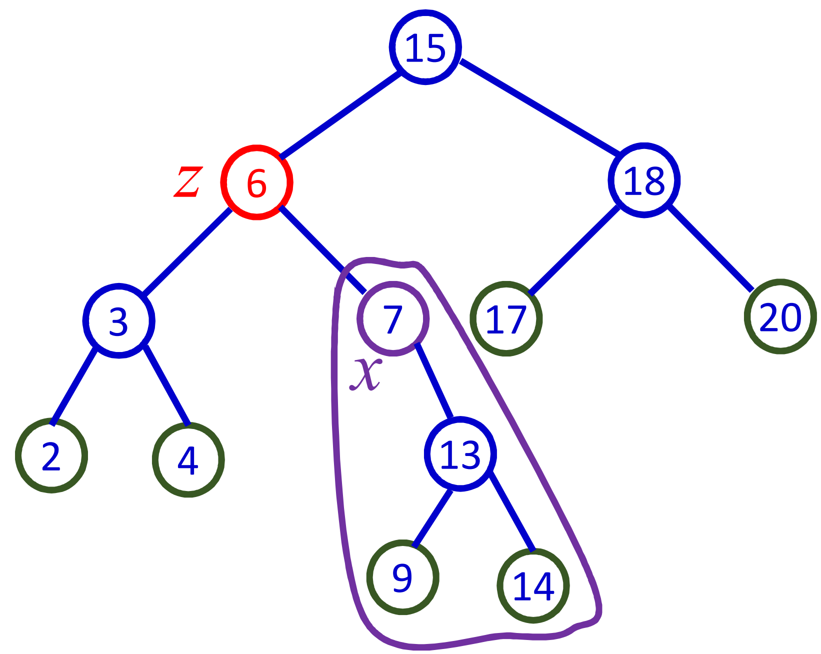
BST Operation: Deletion

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

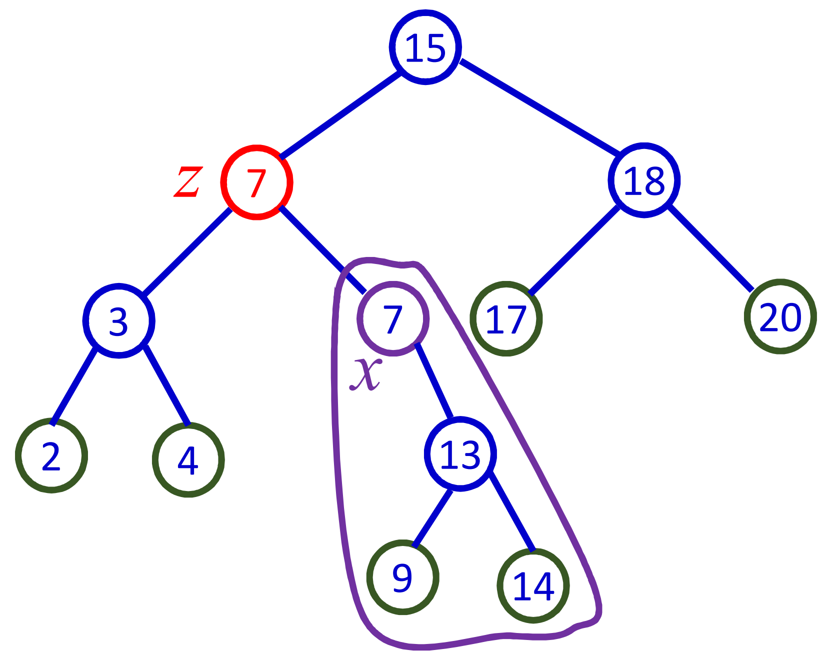


BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 6

Now copy key of x to z

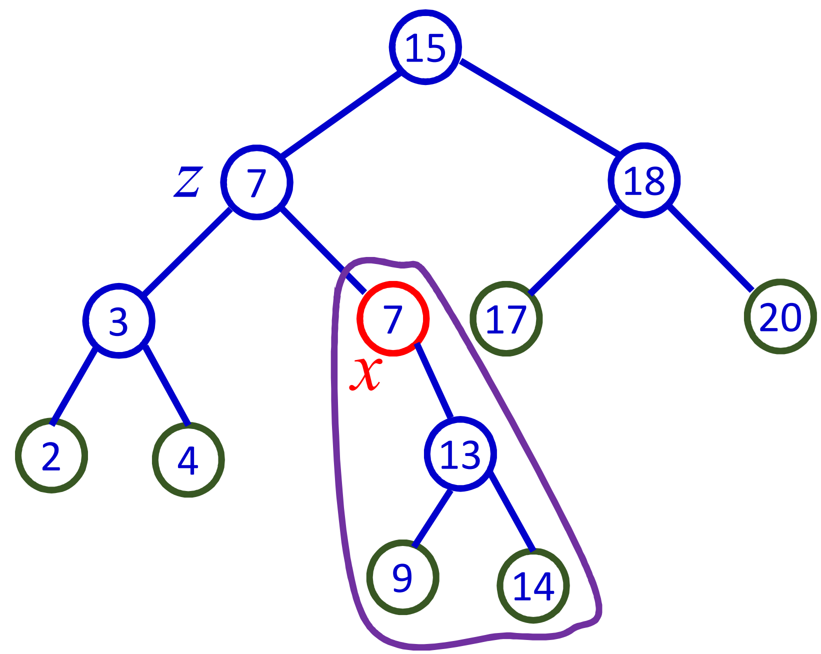


BST Operation: Deletion

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

Now copy key of *x* to *z*

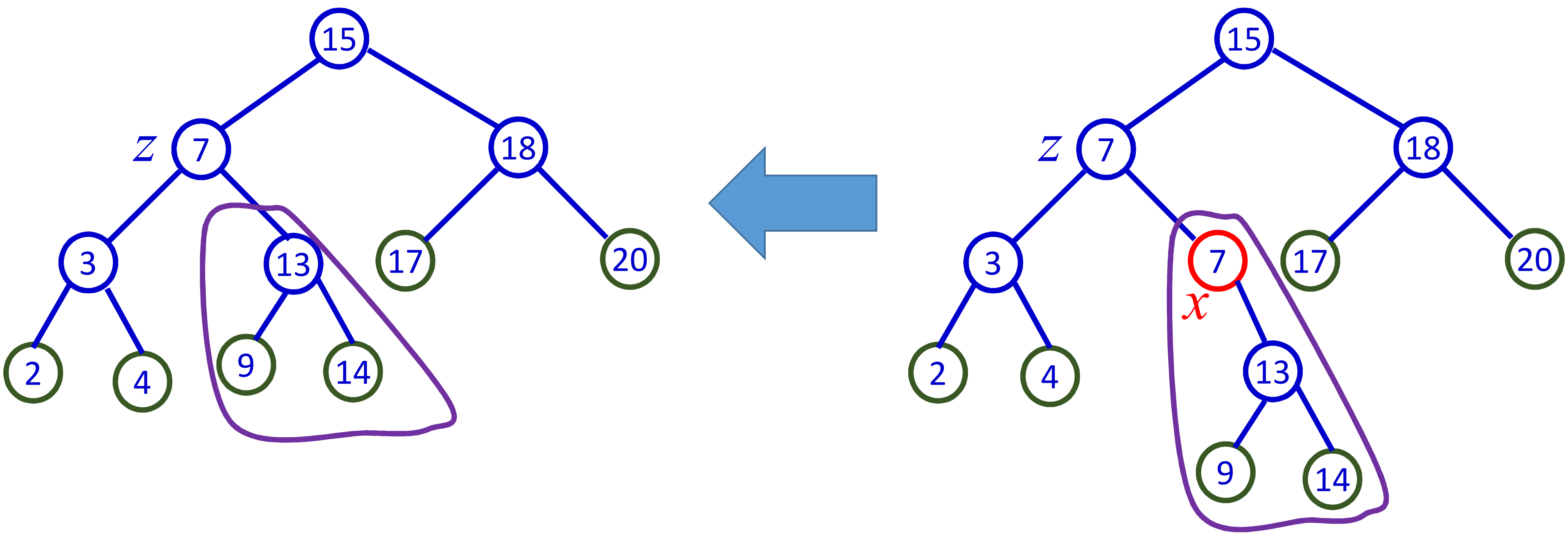
Delete node *x* (it has a single child ONLY)



BST Operation: Deletion

Removing a node having two children (full node)

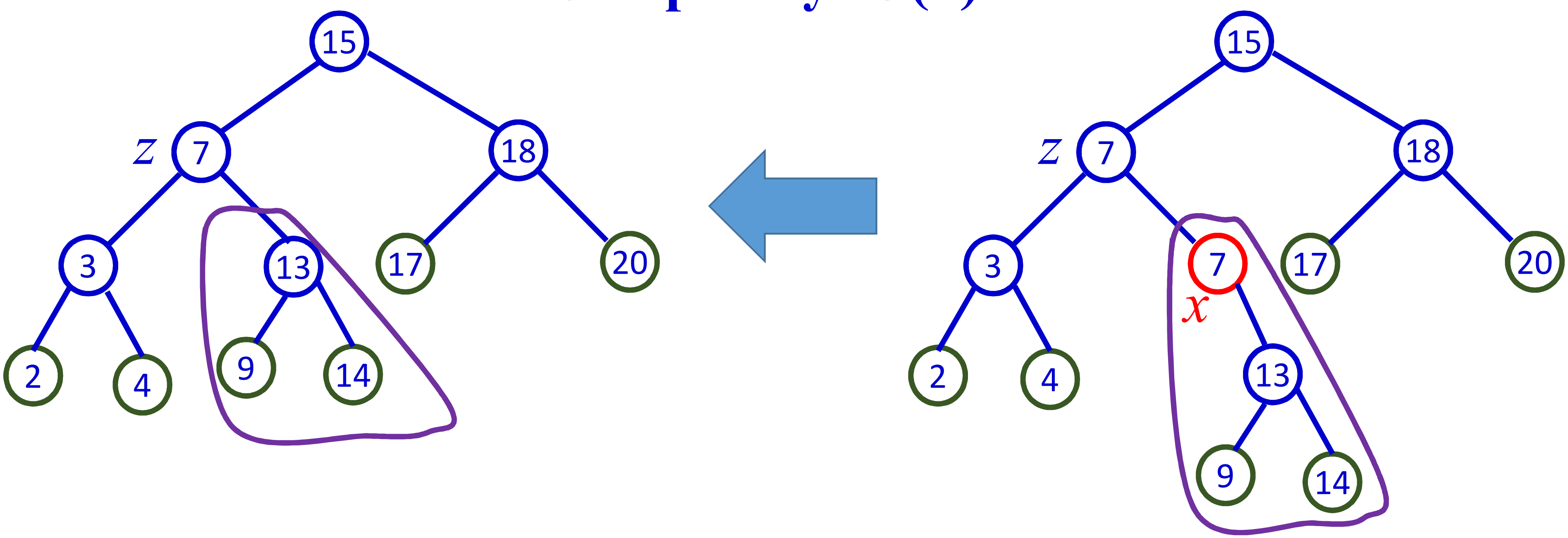
Remove node with 6



BST Operation: Deletion

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

Complexity: O(*h*)



BST Operations: Complexity

Search: O(h)

Maximum / Minimum : O(h)

Predecessor / Successor: O(h)

Insert / Delete: O(h)

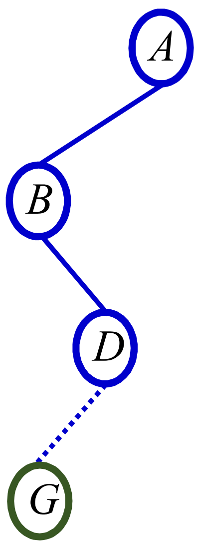
h = depth of the deepest node in the BST, i.e.,

$\approx$ height of the tree.

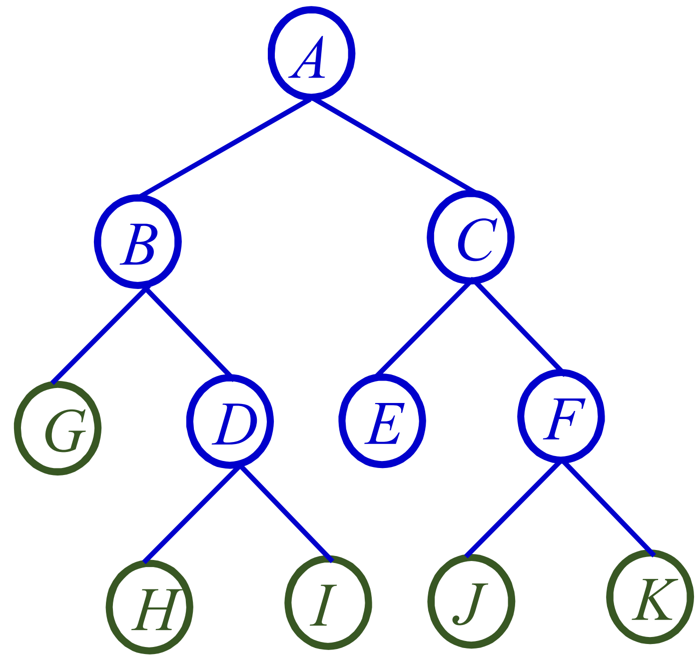
$\approx$ log n if tree is balanced.

What is the worst case?

BST Operations: Complexity

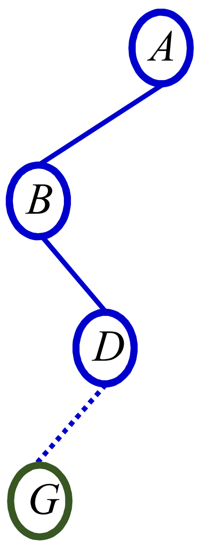


Chain, Imbalanced: height, $h \approx n$
All complexity: $O(n)$



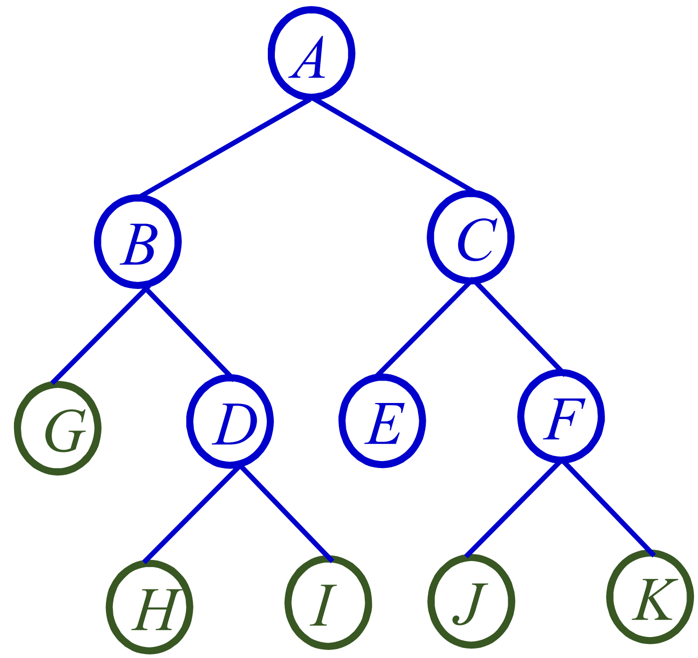
Balanced: height, $h = \log(n)$
All complexity: $O(\log(n))$

BST Operations: Complexity of Creation of BST



A BST of n nodes
Insert one after another

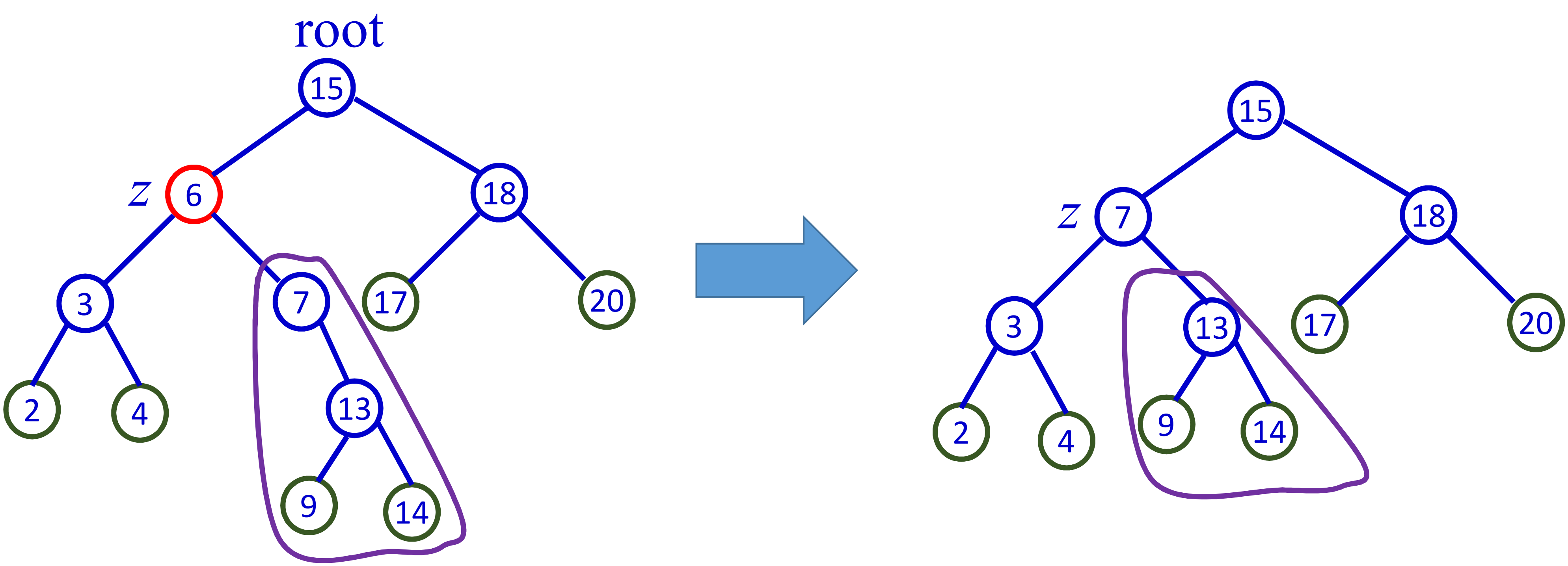
Chain, Imbalanced: height, $h \approx n$
Complexity: $\sum_{i=1}^{n} i = \mathrm{O}(n^2)$



Balanced: height, h =log(n)
Complexity: O($n$log(n))

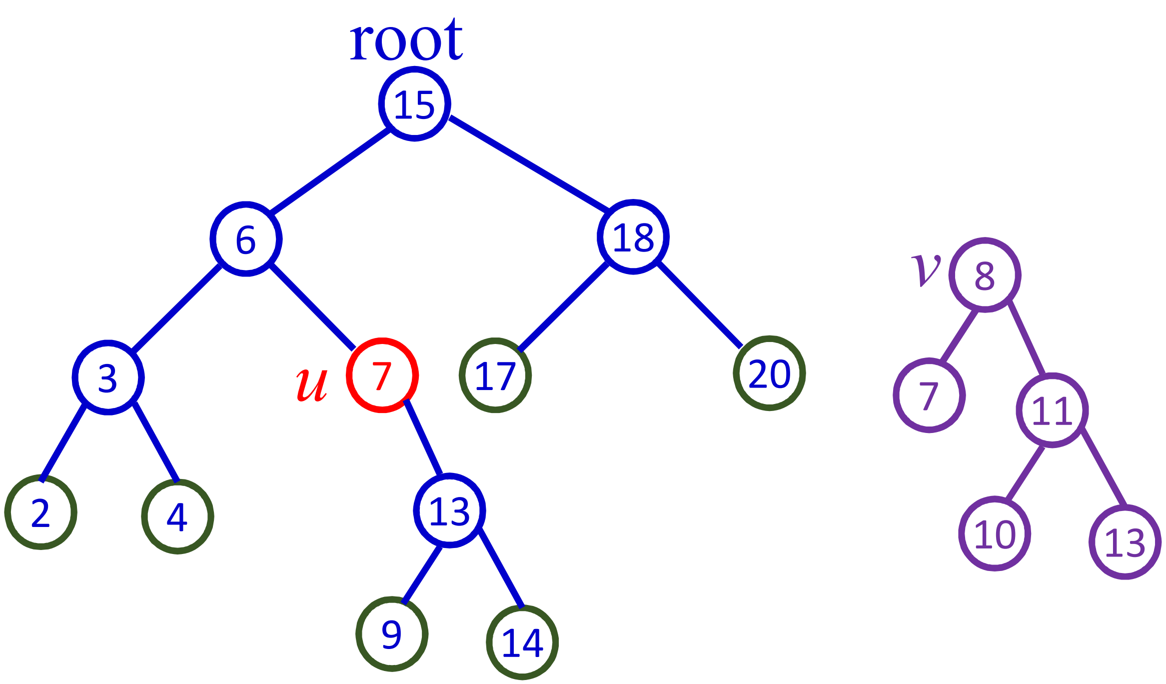
BST Operation: Deletion - **Alternate Method (2)**

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



BST Operation: Deletion (2)

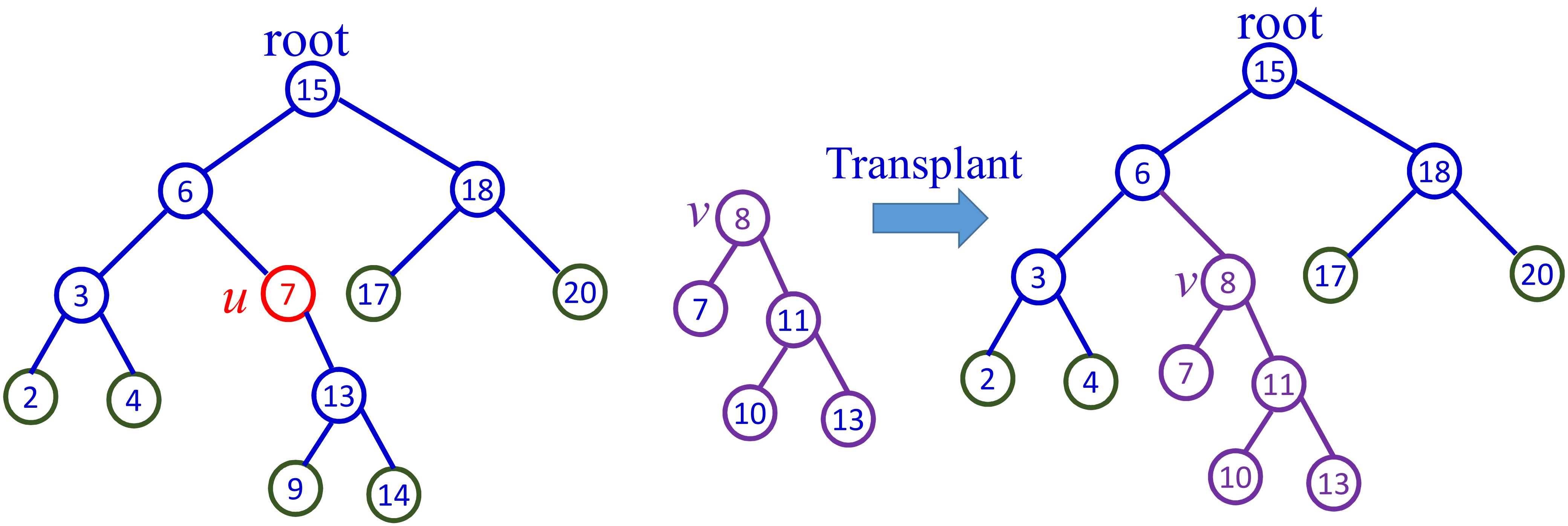
First try to replace node u by node v



BST Operation: Deletion (2)

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

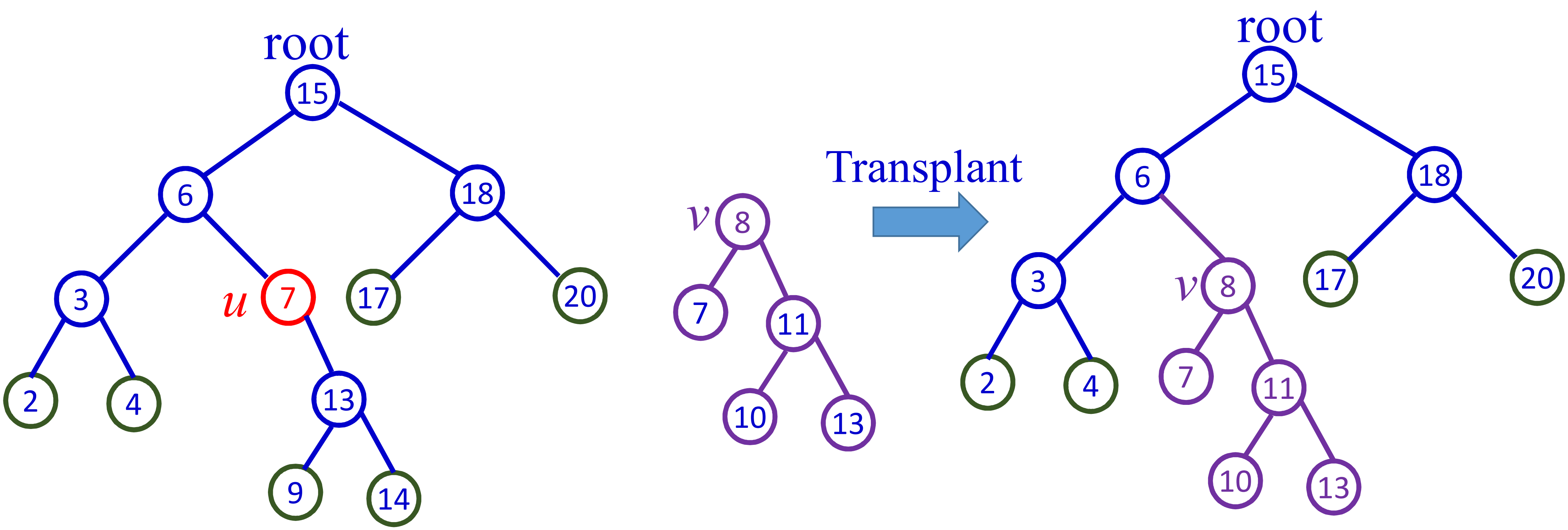
Removes a node directly even if it has two children (full node)



BST Operation: Deletion (2)

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

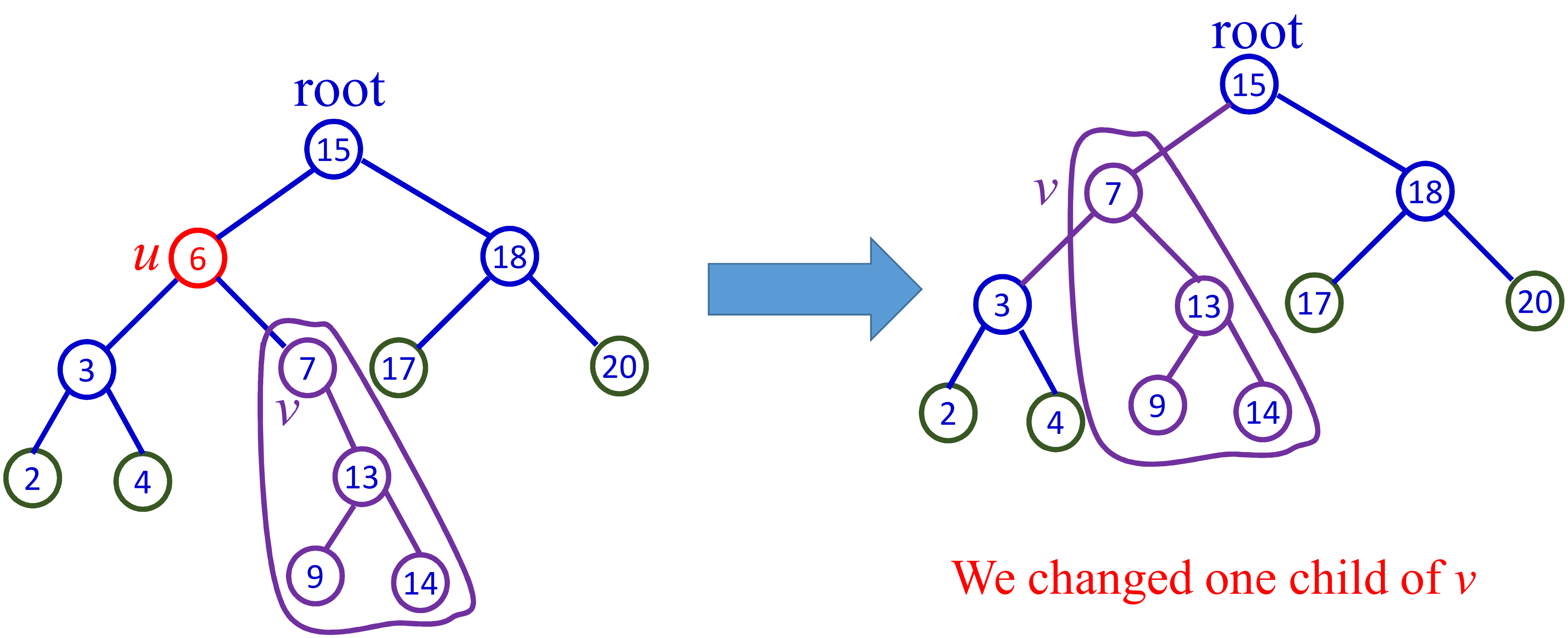
First try to replace node *u* by node *v*



We did not change children of *v*

BST Operation: Deletion (2)

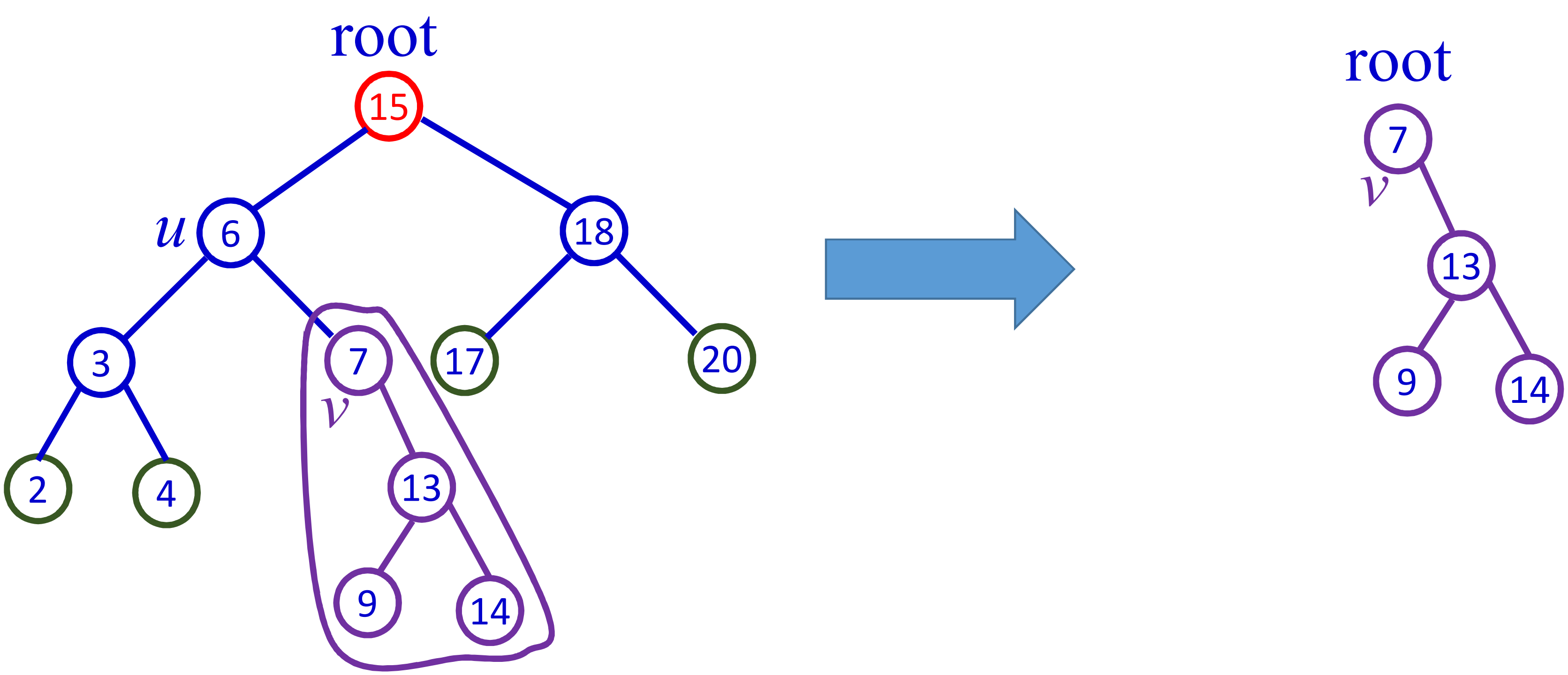
This is also transplant



We changed one child of v

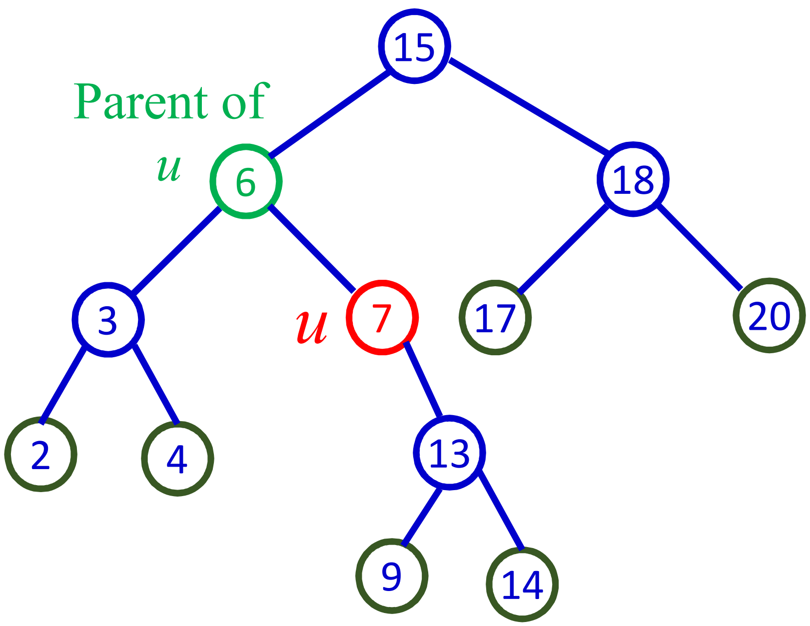
BST Operation: Deletion (2)

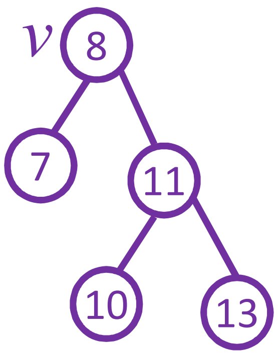
Even we can replace the root



BST Operation: Deletion (2)

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



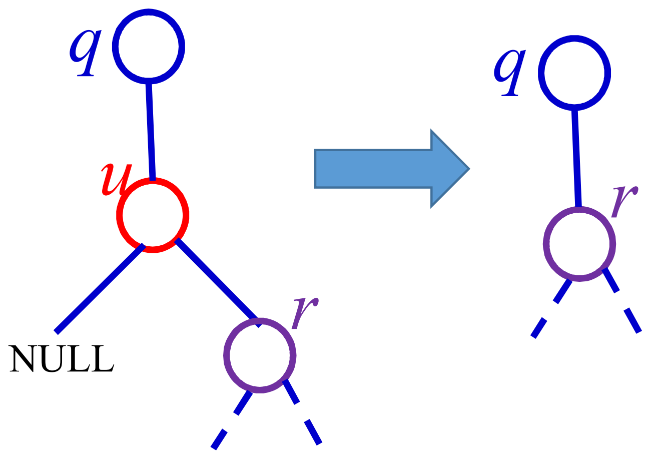



```
TRANSPLANT(T, u, v)
if u->parent == NULL          //special case
      T->root = v
elseif u = = u->parent->left   //set appropriate child
        u->parent->left = v
else u->parent->right =v
if v ≠ NULL                   //set parent
    v->parent = u->parent
```

BST Operation: Deletion (2)

Node Deletion Cases

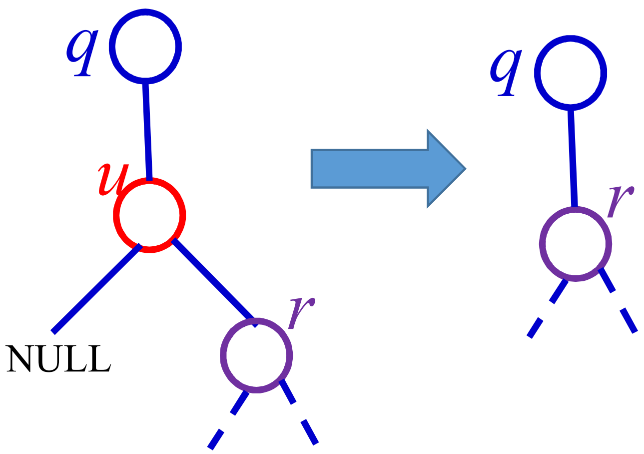
Node u has **NO LEFT** child



BST Operation: Deletion (2)

Node Deletion Cases

Node u has **NO LEFT** child

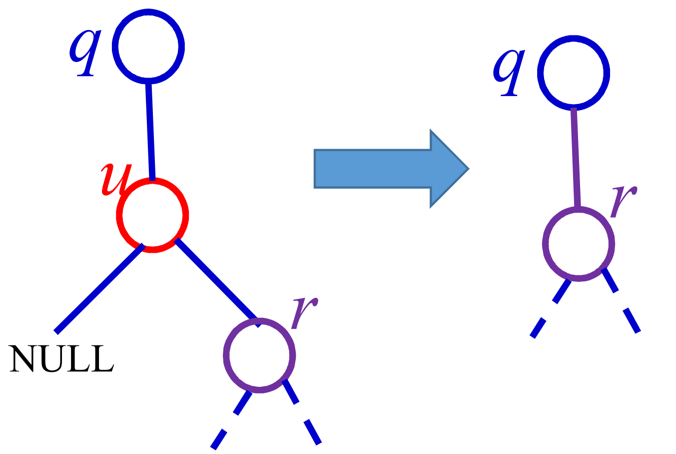


This also covers if both children are empty

BST Operation: Deletion (2)

Node Deletion Cases

Node *u* has **NO LEFT** child

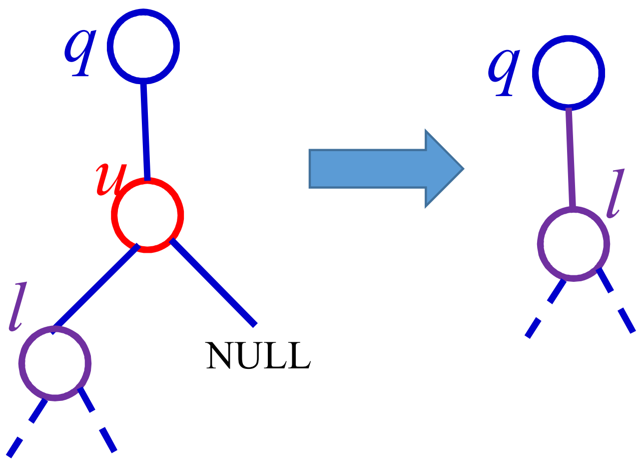


TREE_DELETE (*T*, *u*)
1 **if** *u*->*left* == NULL
2 TRANSPLANT(*T*, *u*, *u*->*right*)

BST Operation: Deletion (2)

Node Deletion Cases

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

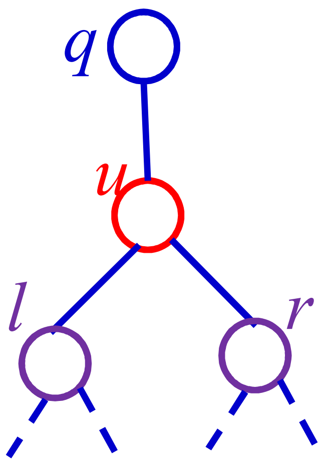


TREE_DELETE (*T*, *u*)
1 **if** *u->left* == NULL
2 TRANSPLANT(*T*, *u*, *u->right*)
3 **elseif** *u->right* == NULL
4 TRANSPLANT (*T*, *u*, *u->left)*

BST Operation: Deletion (2)

Node Deletion Cases

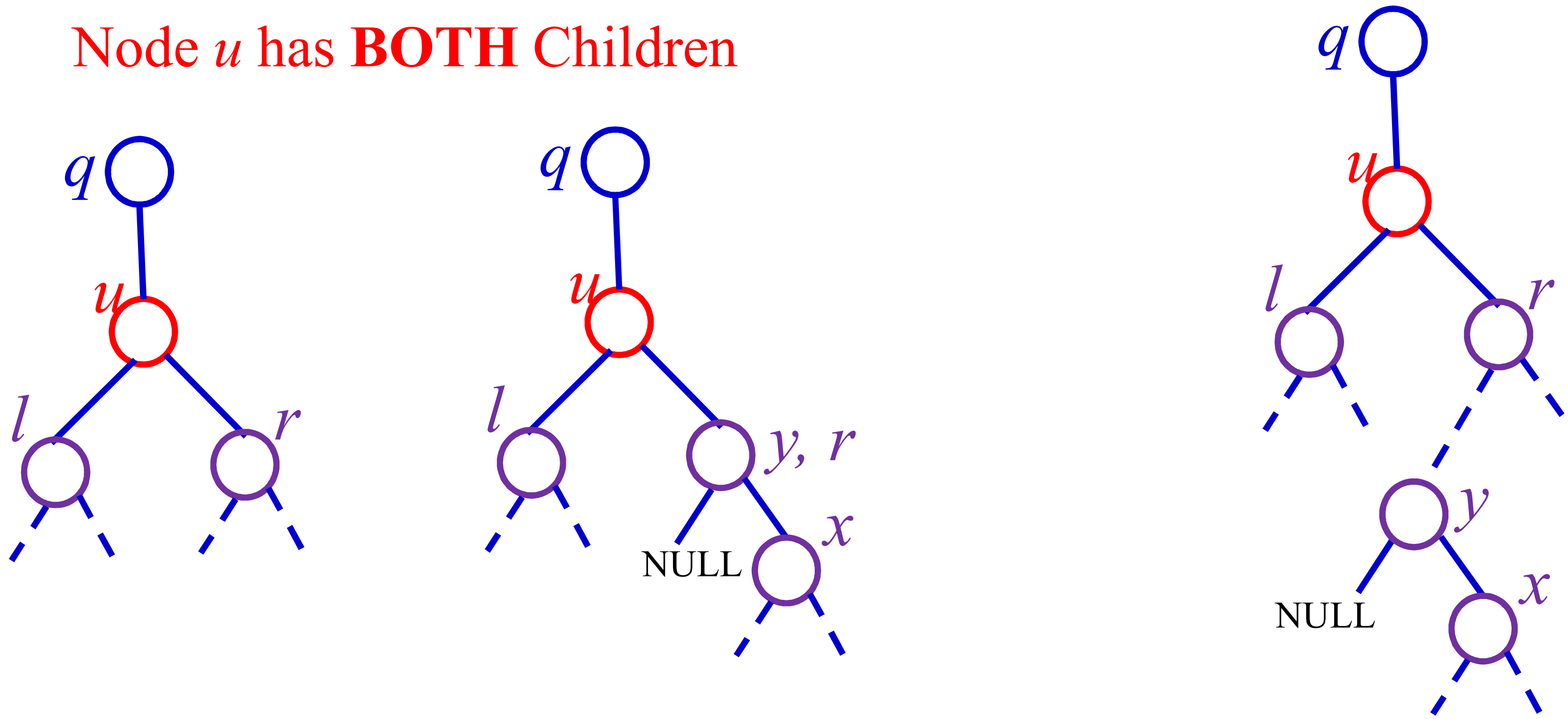
Node u has **BOTH** Children



BST Operation: Deletion (2)

Node Deletion Cases

Node *u* has **BOTH** Children

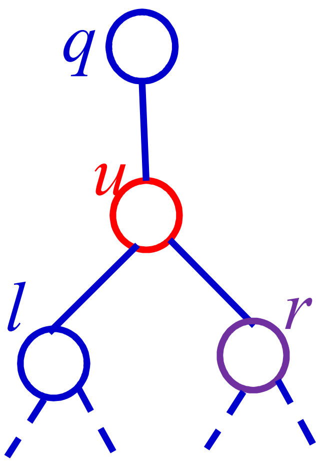


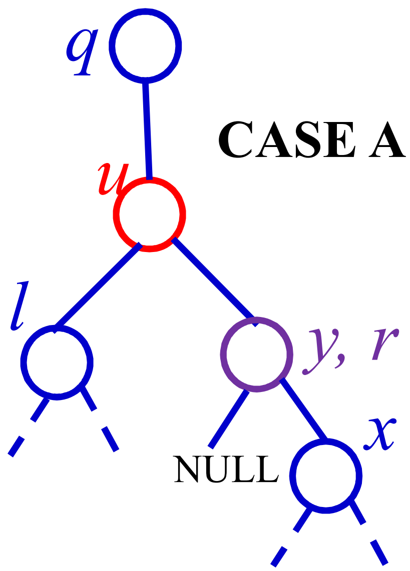
Find *y* = successor (next minimum) from RIGHT subtree

BST Operation: Deletion (2)

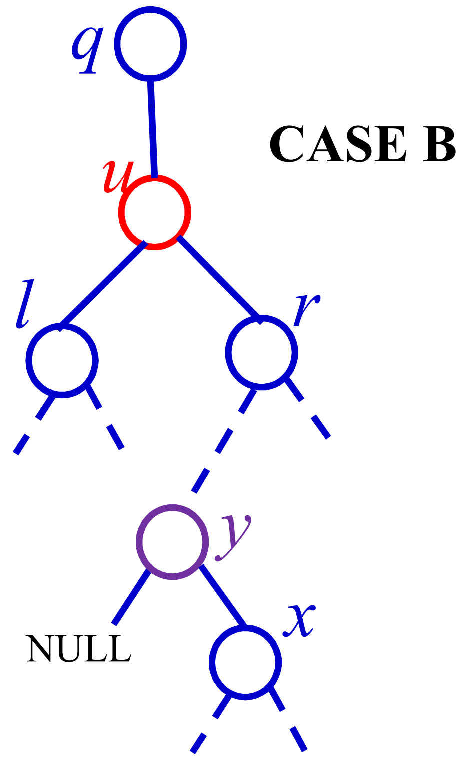
Node Deletion Cases

Node u has **BOTH** Children





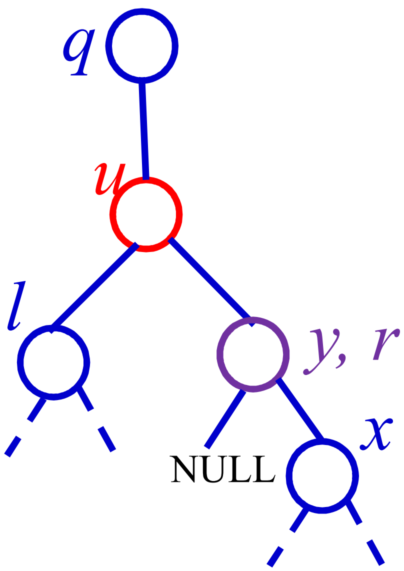
y is immediate RIGHT child of u



y is NOT immediate child of u

BST Operation: Deletion (2)

CASE A



y is immediate RIGHT child of *u*

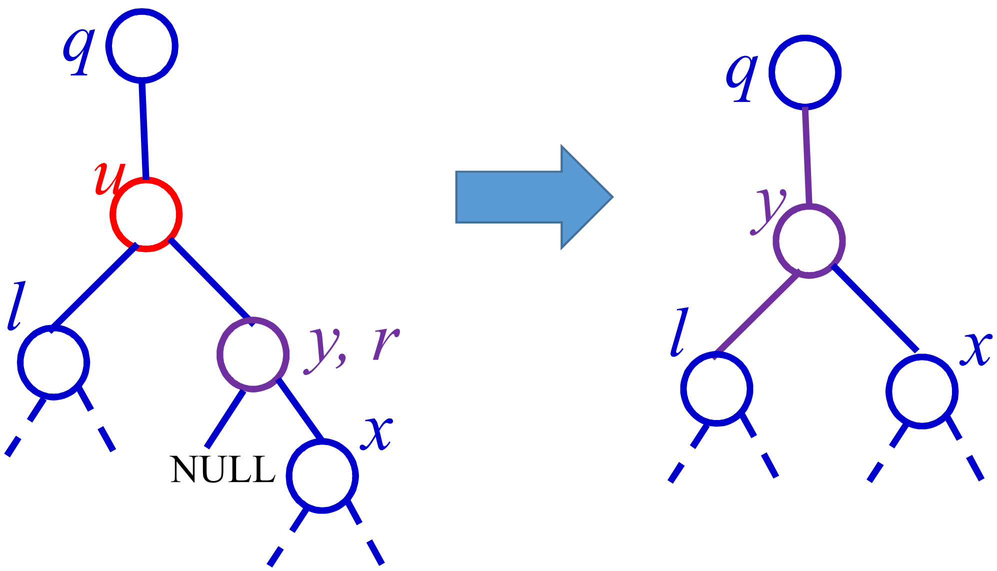
```
TREE_DELETE (T, u)
if u->left == NULL
    TRANSPLANT(T, u, u->right)
elseif   u->right == NULL
    TRANSPLANT (T, u, u->left)
else y = TREE_MINIMUM(u->right)
```

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)
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
```