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

Instructor

Dr Md Monirul Islam

Graphs and Trees: Representation and Search

BST Operations

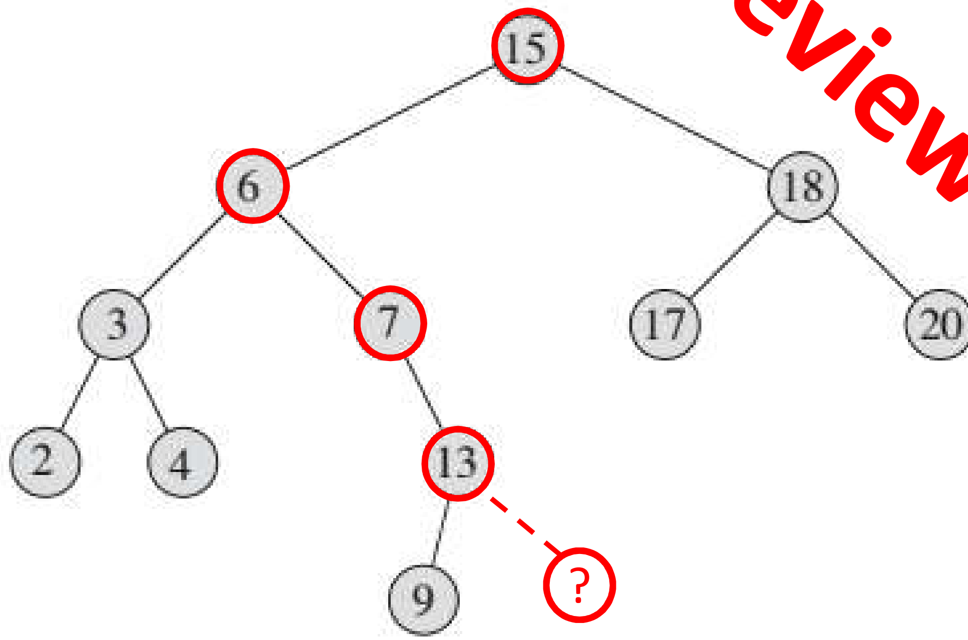
Review

- Search for a key
- Minimum
- Maximum
- Successor
- Predecessor
- Insert
- Delete

BST Operation: Search

Review

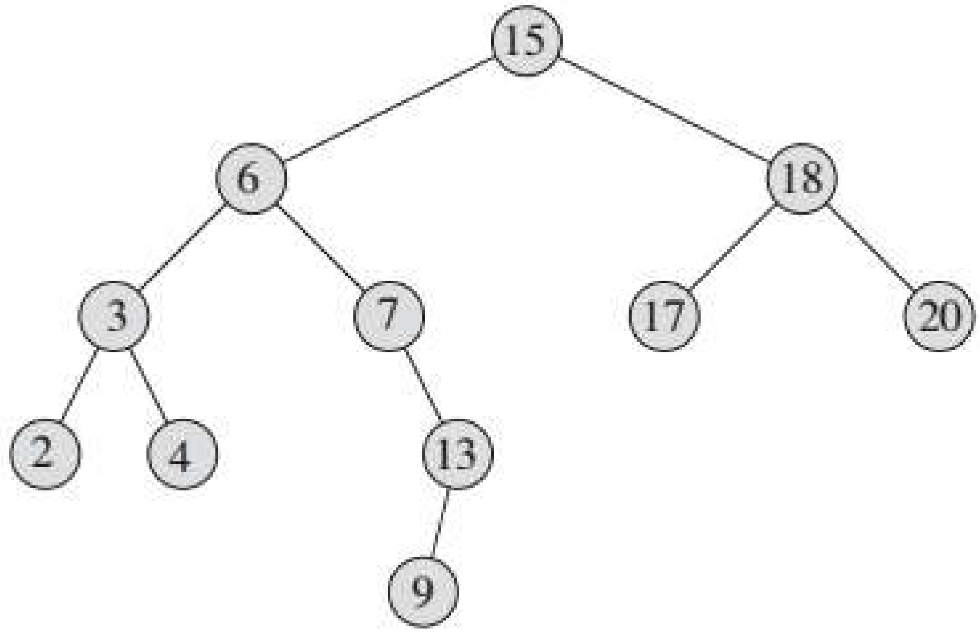
```
TREE_SEARCH (x, k)
if x == NULL or k == x->key
    return x
if k < x->key
    return TREE_SEARCH(x->left, k)
else return TREE_SEARCH(x->right, k)
```



Complexity: O(*h*)

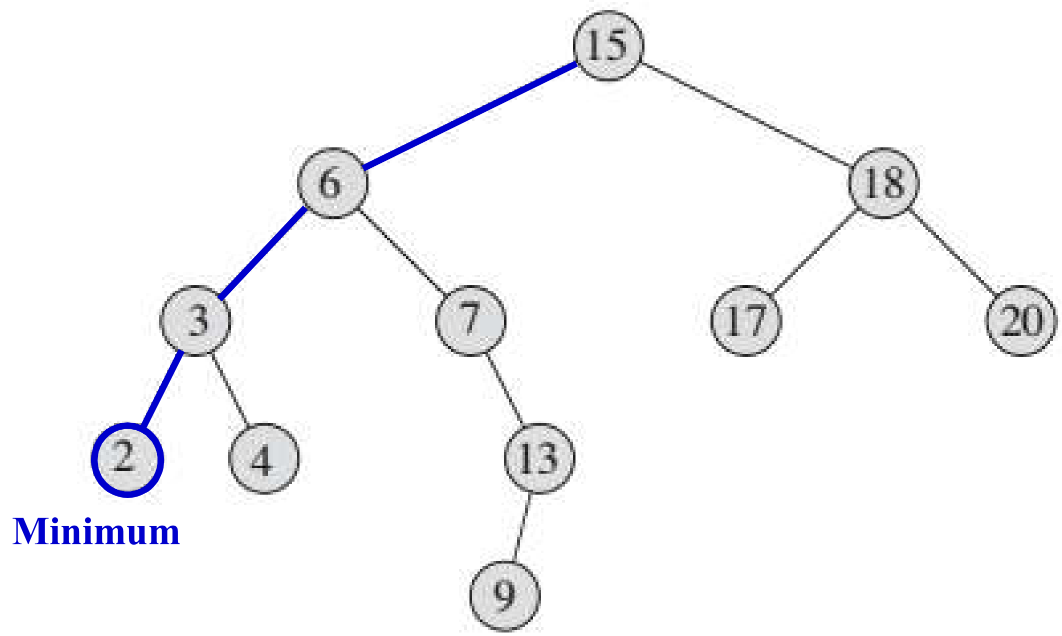
BST Operation: Minimum

Where?



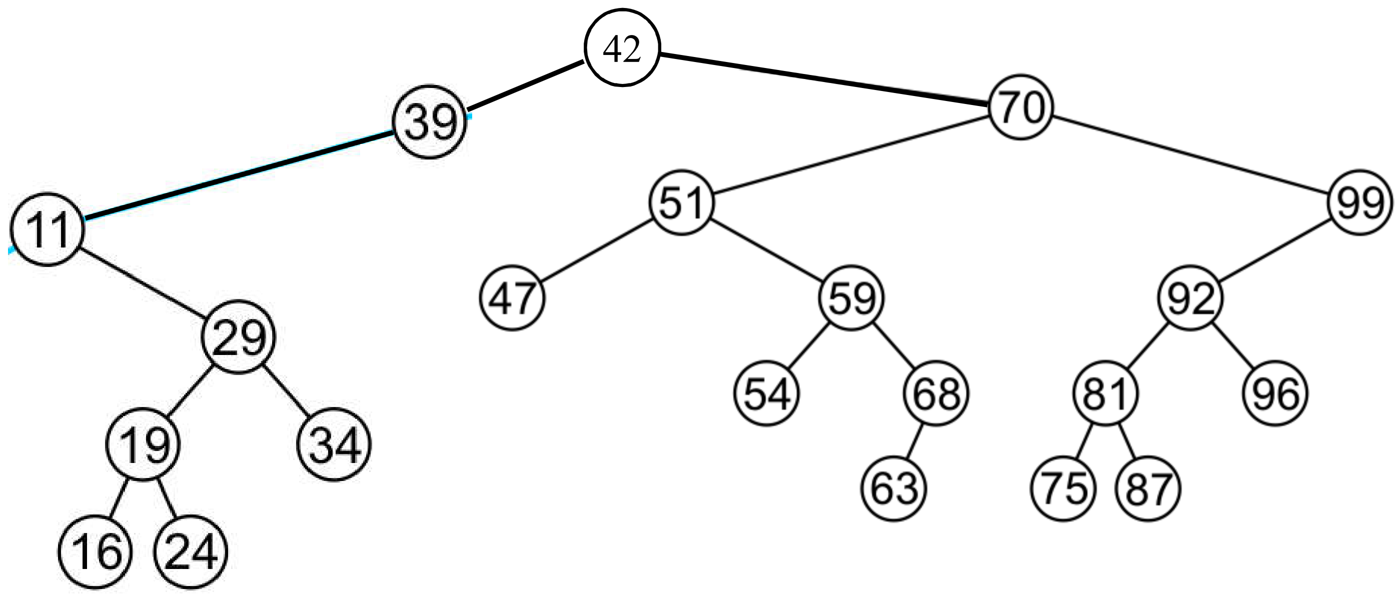
BST Operation: Minimum

Must be in the left subtree



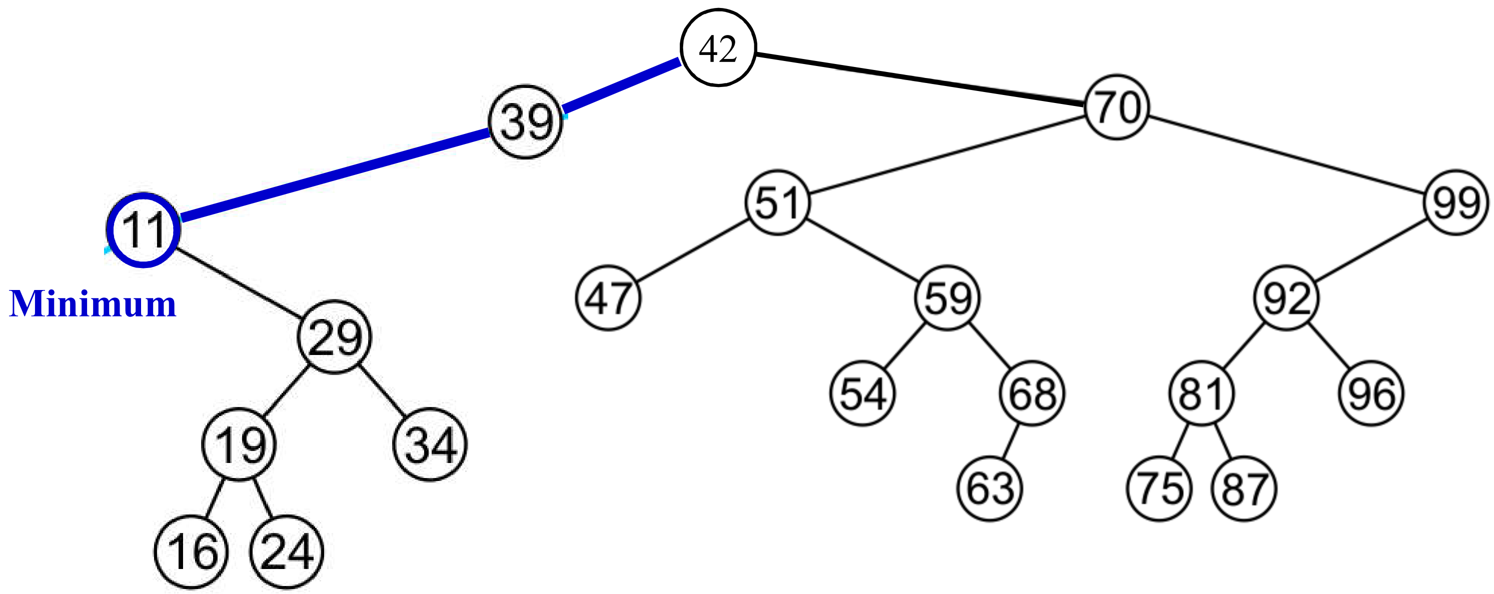
BST Operation: Minimum

Not necessarily in the leaf node



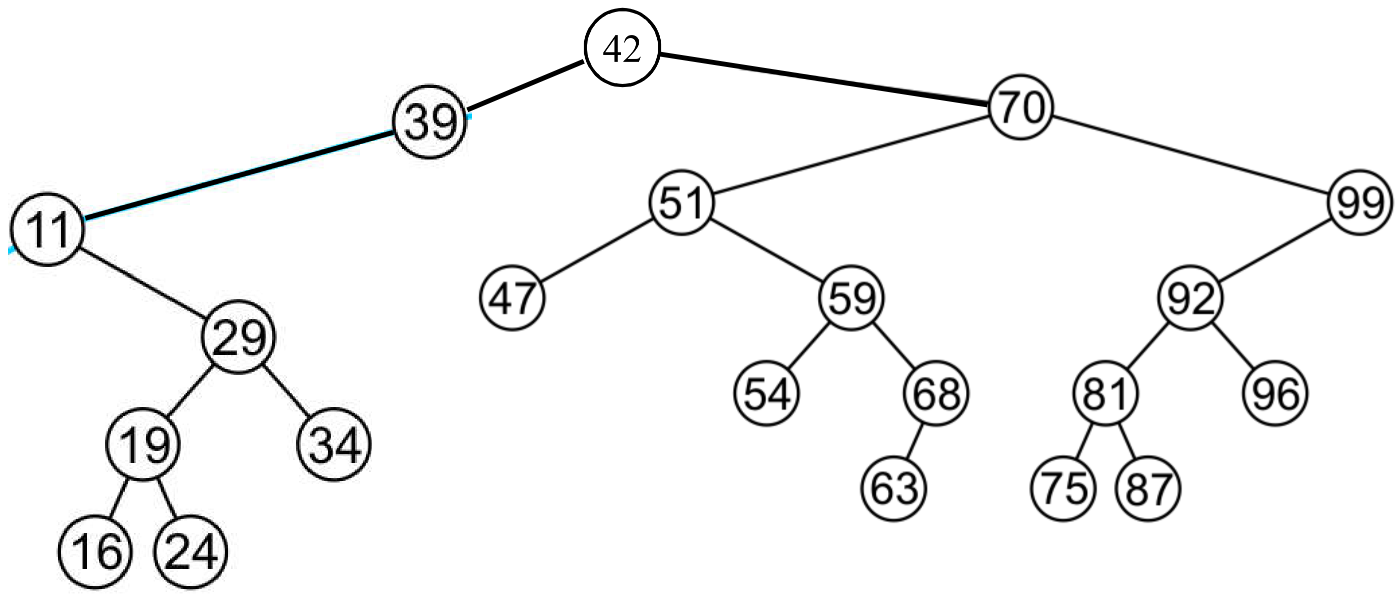
BST Operation: Minimum

Not necessarily in the leaf node

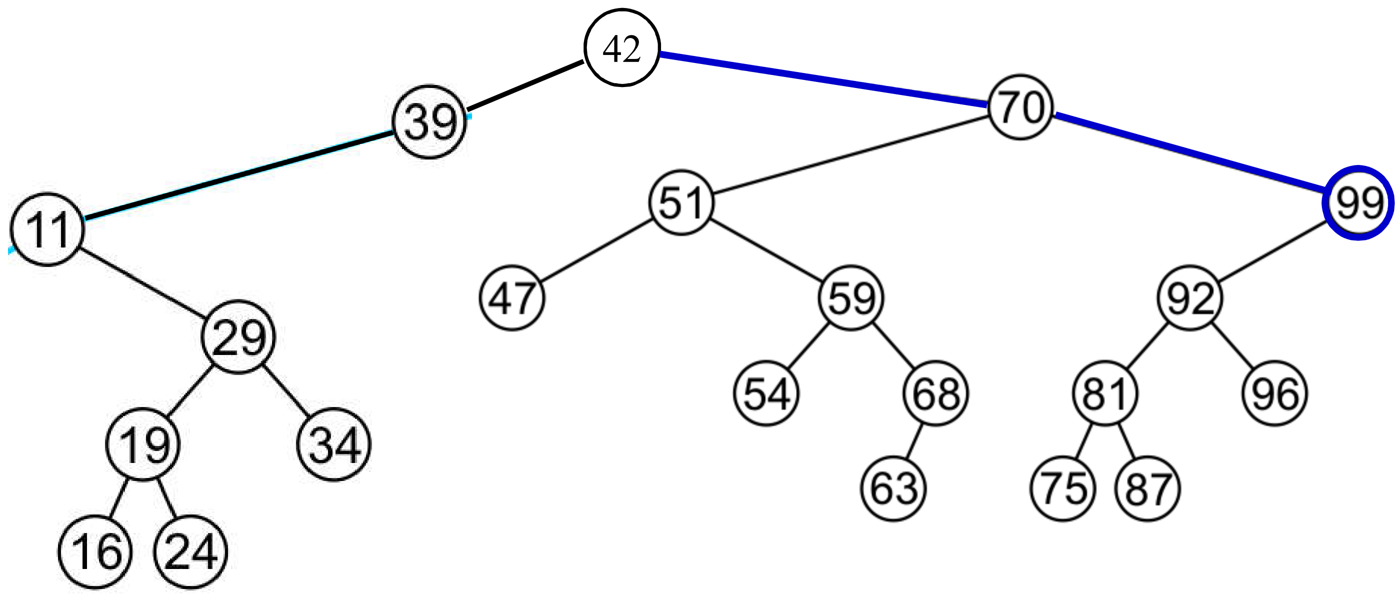


BST Operation: Minimum

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

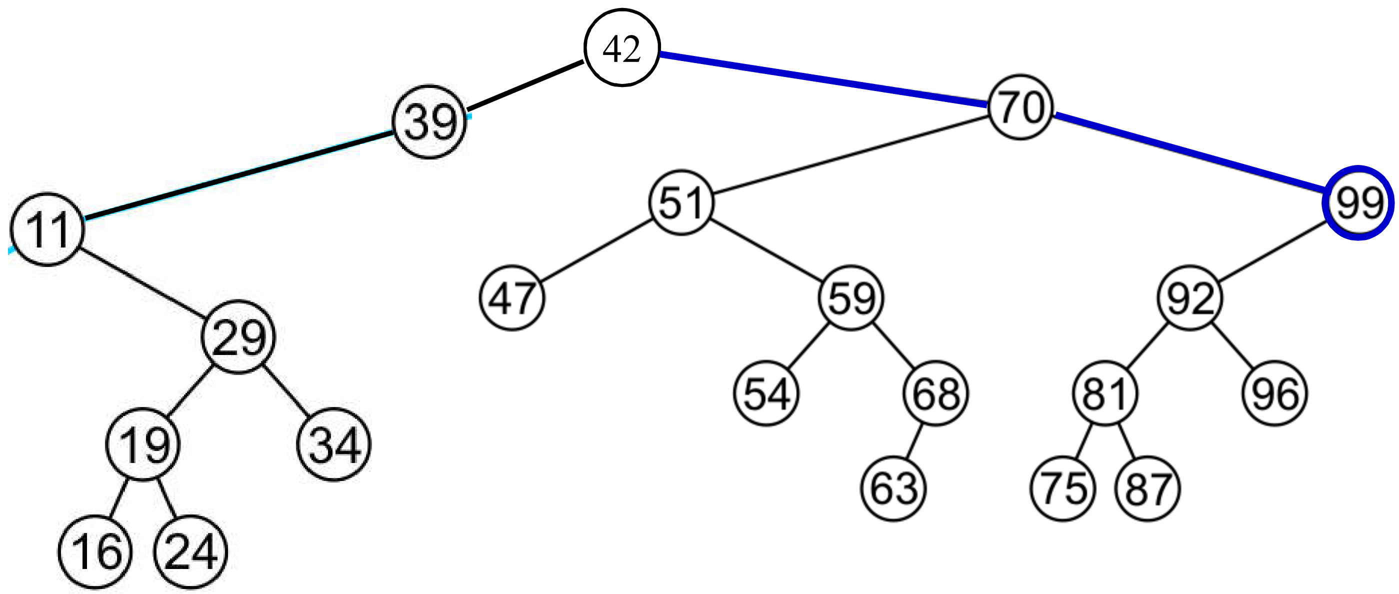


BST Operation: Maximum



BST Operation: Maximum

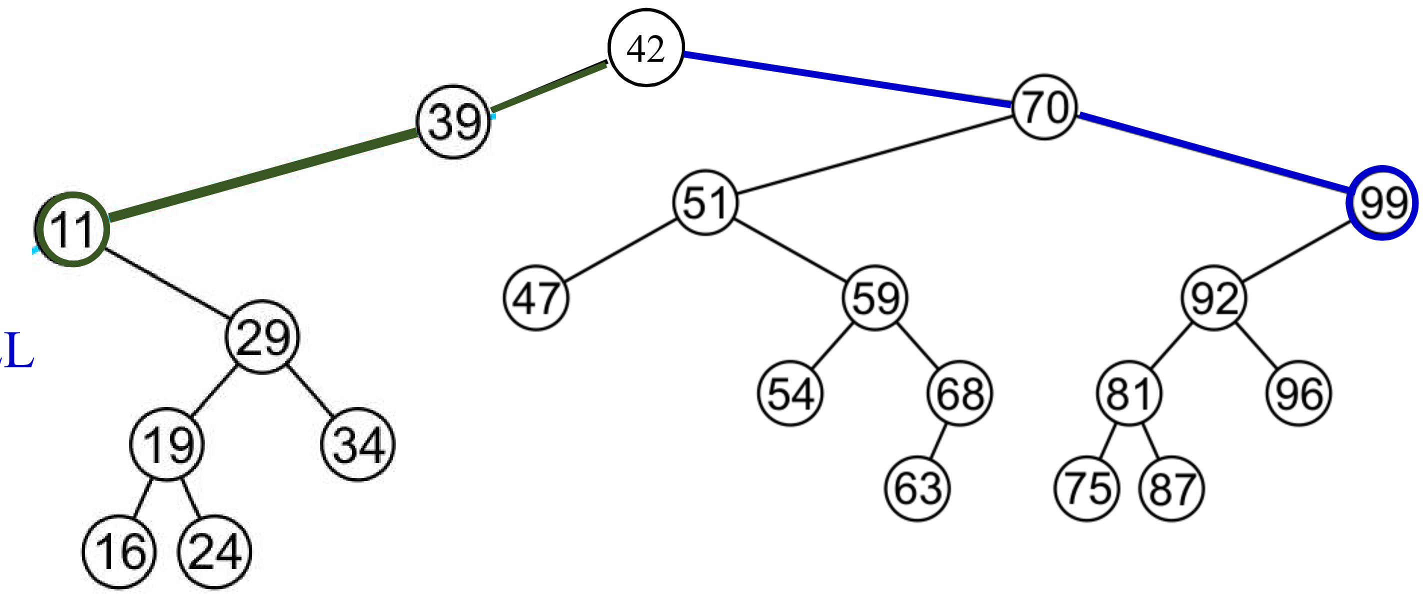
```
TREE_MXIMUM (x)
if x == NULL return NULL
while x->right ≠ NULL
      x = x->right
return x
```



BST Operation: Minimum and Maximum

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

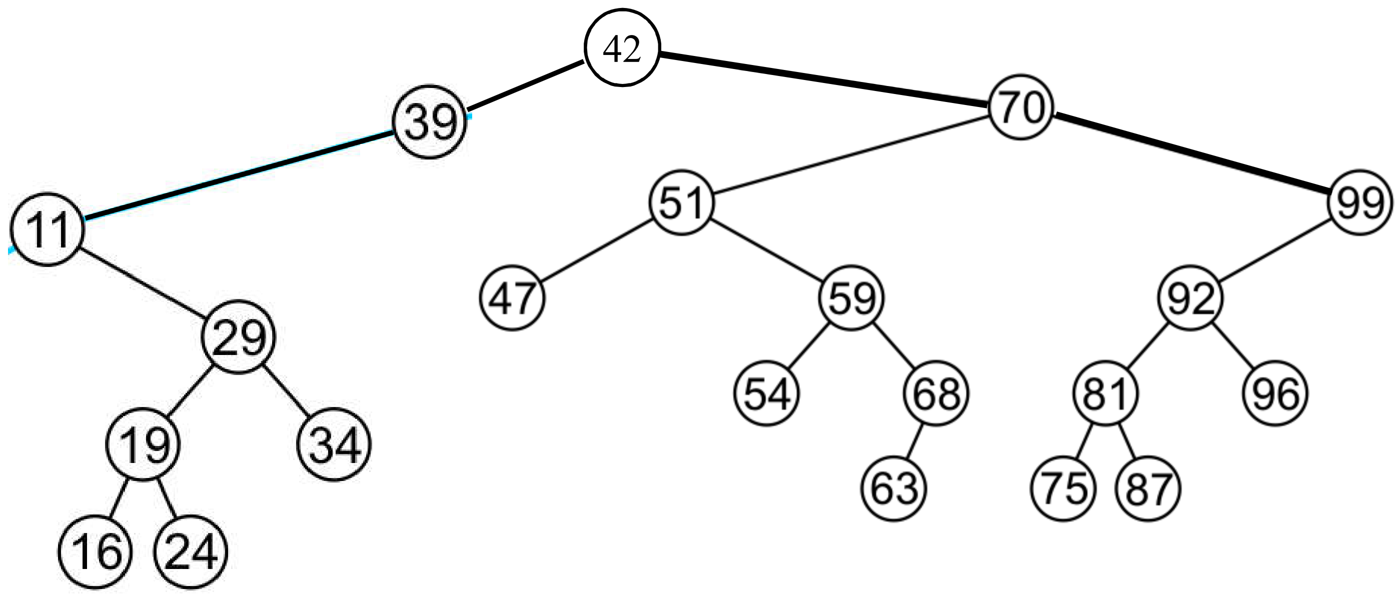
Complexity: O(*h*)




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

BST Operation: Successor

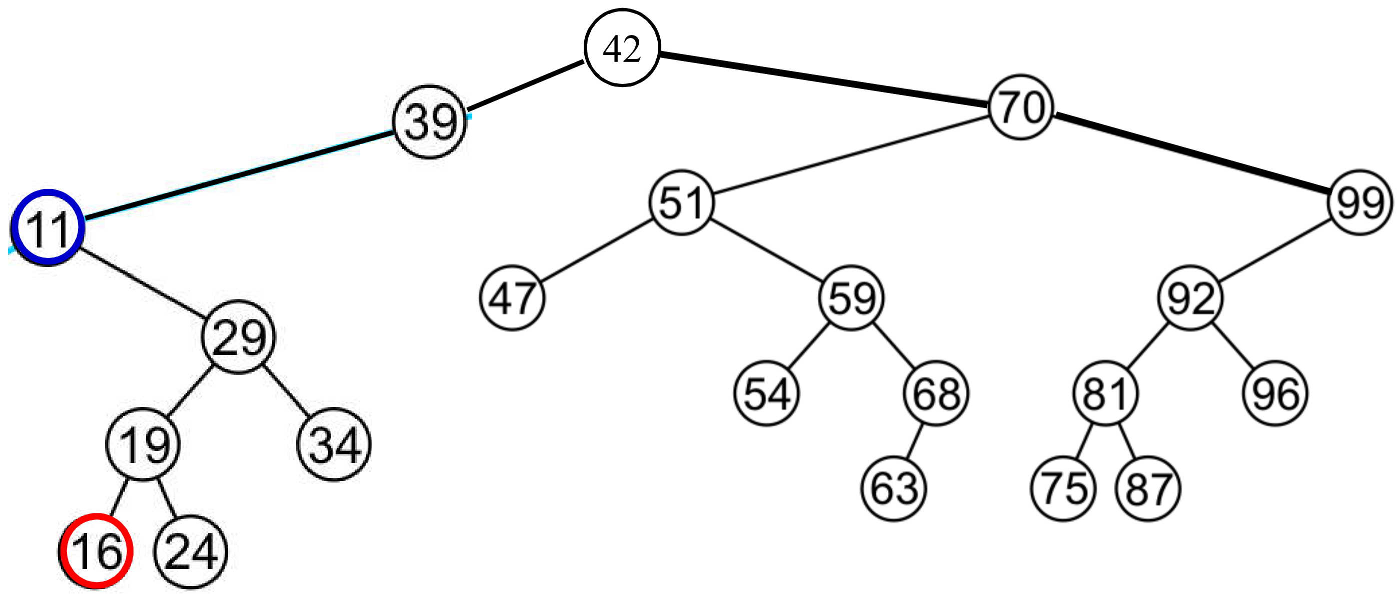
successor of a node x : the node with the smallest key greater than $x.key$



BST Operation: Successor

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

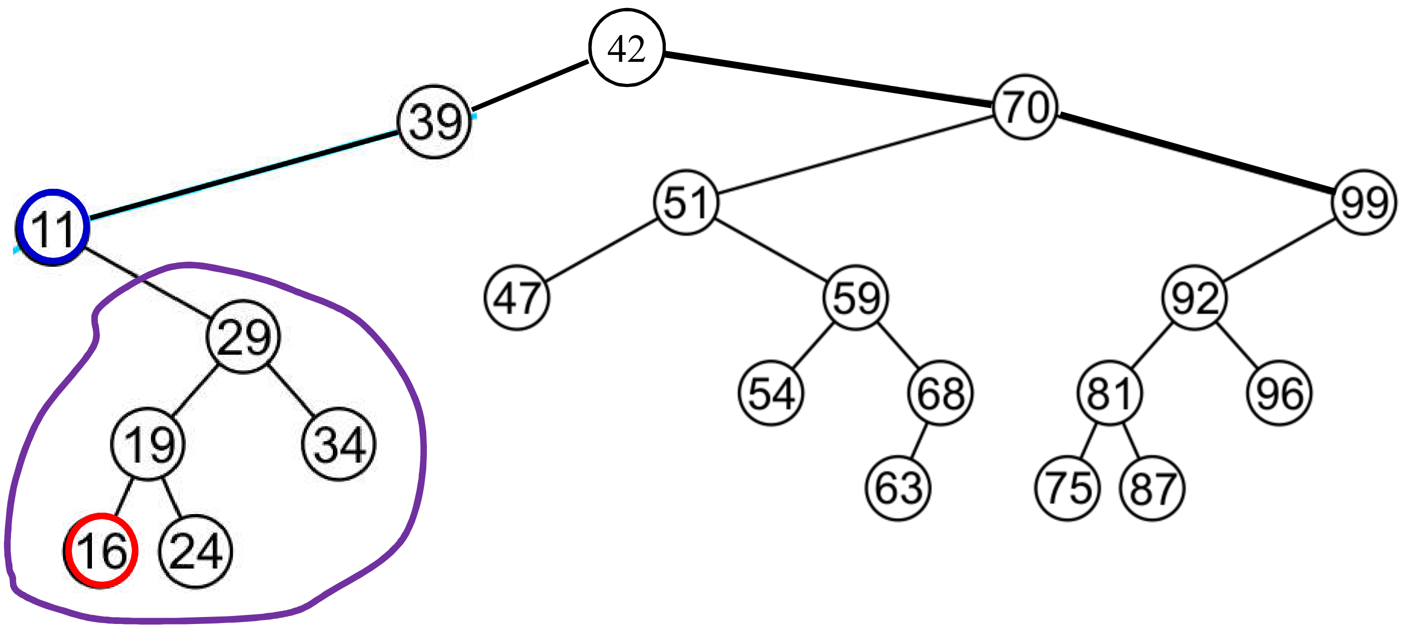
successor of the node with 11 : the node with 16



BST Operation: Successor

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

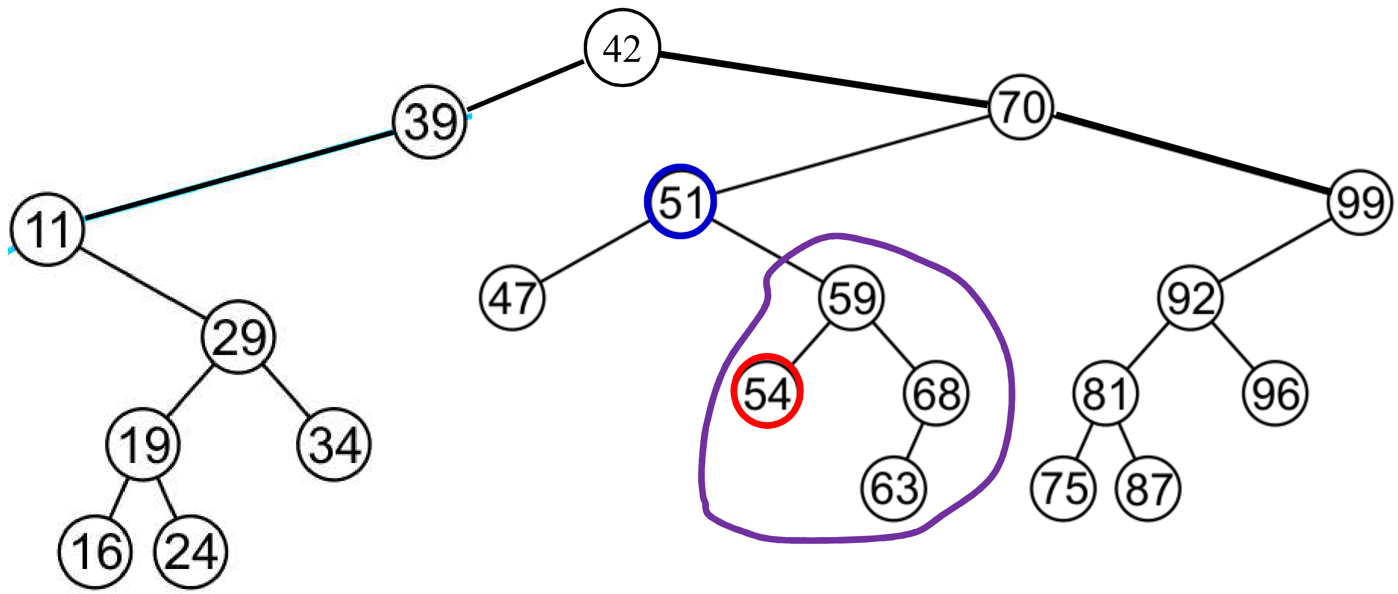
successor of the node with 11 : the node with 16 (minimum of right subtree)



BST Operation: Successor

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

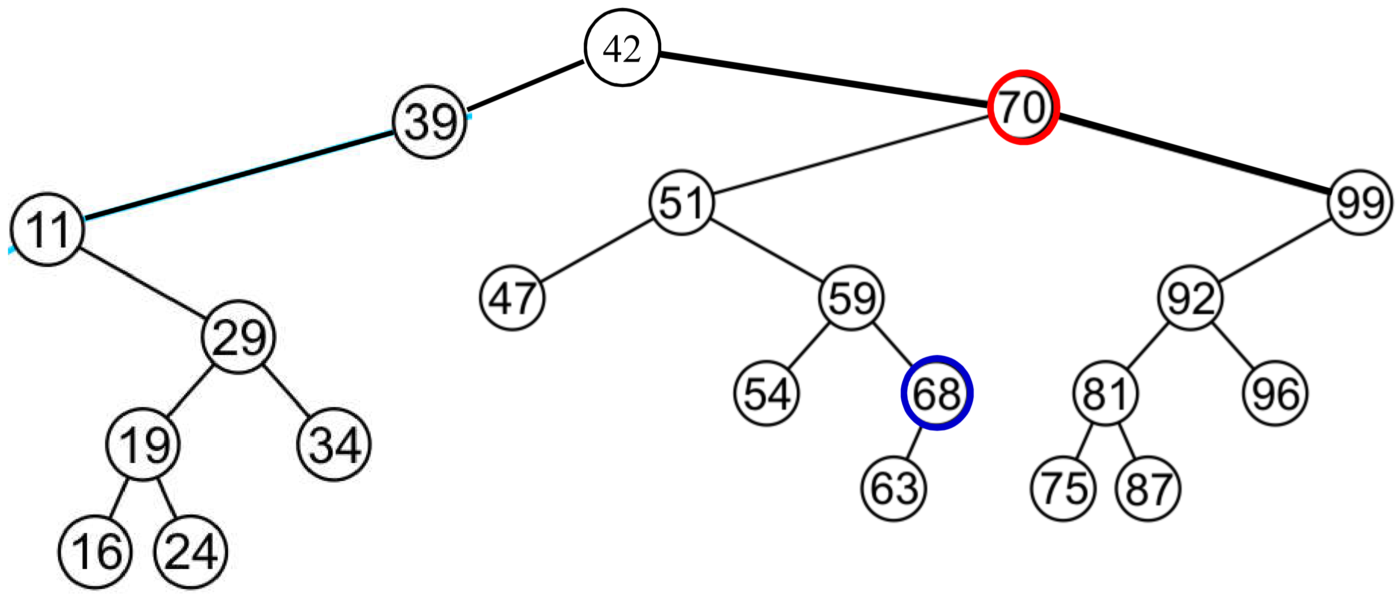
successor of the node with 51 : the node with 54 (minimum of right subtree)



BST Operation: Successor

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

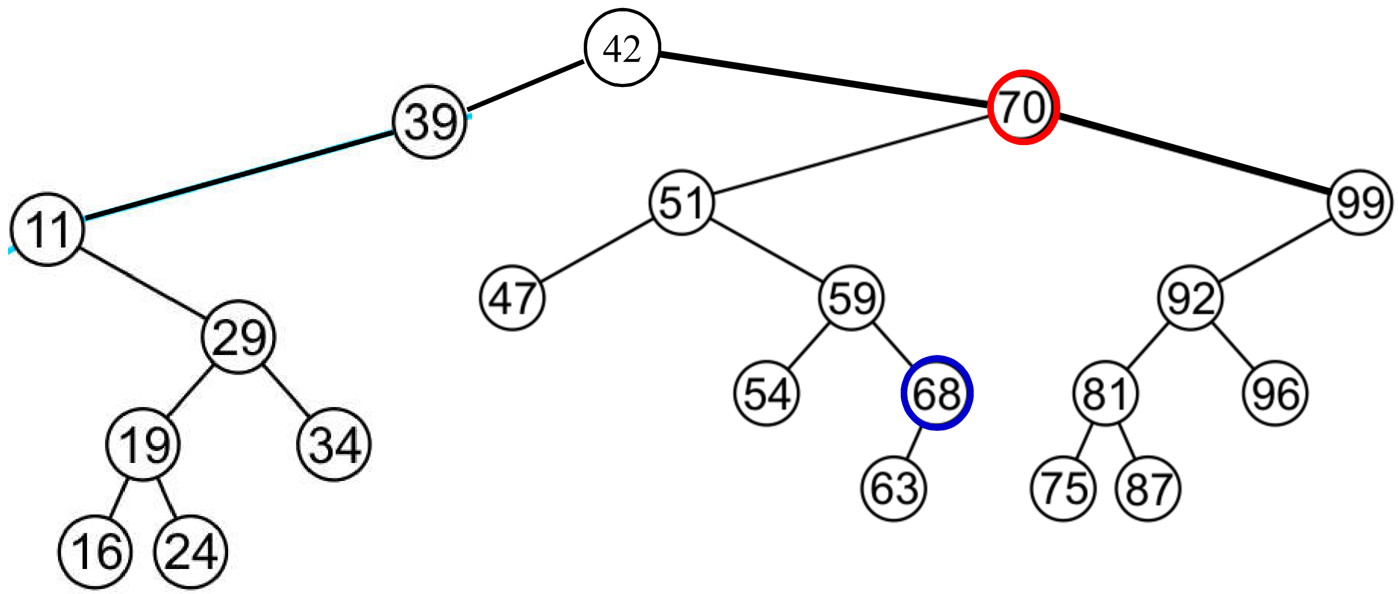
successor of the node with 68 : the node with 70



BST Operation: Successor

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)

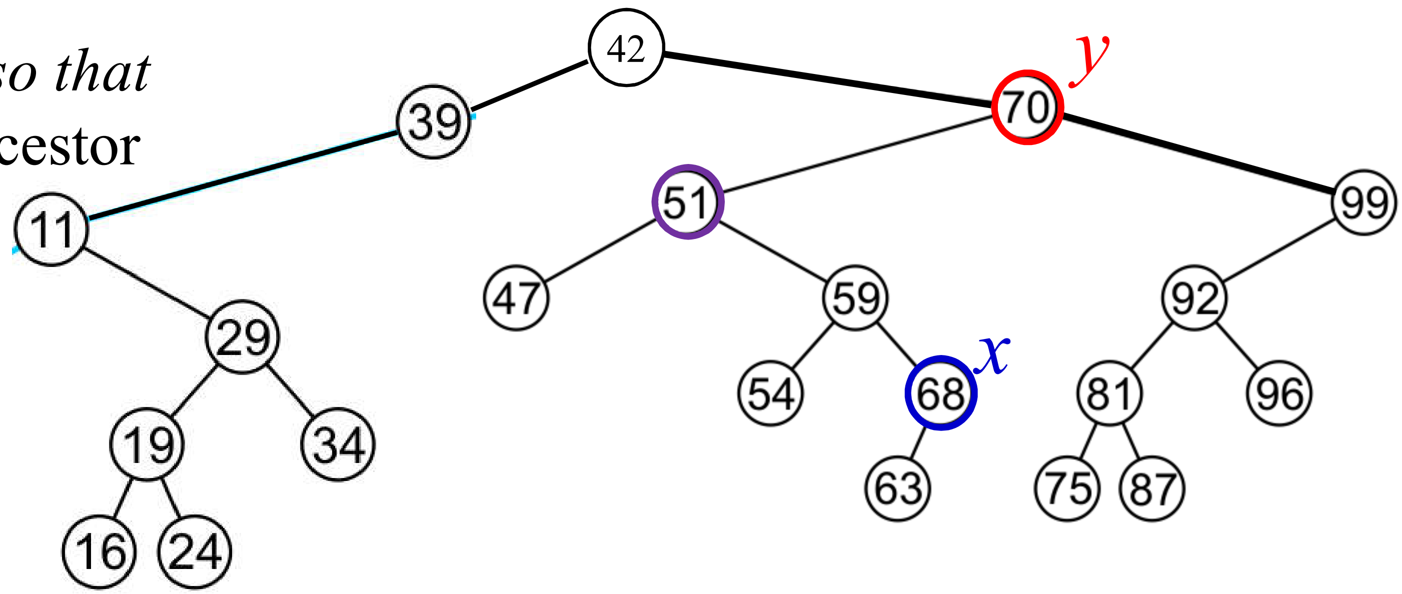


BST Operation: Successor

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

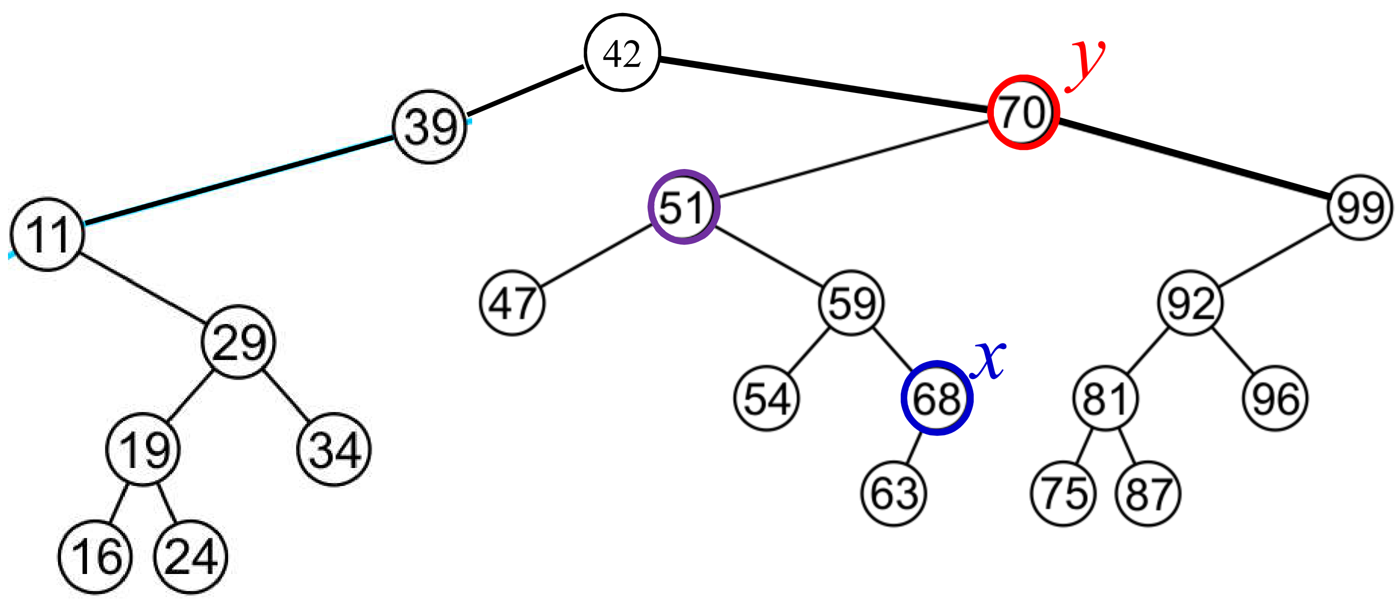
successor of the node with 68 : the node with 70 (right subtree is NULL)

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



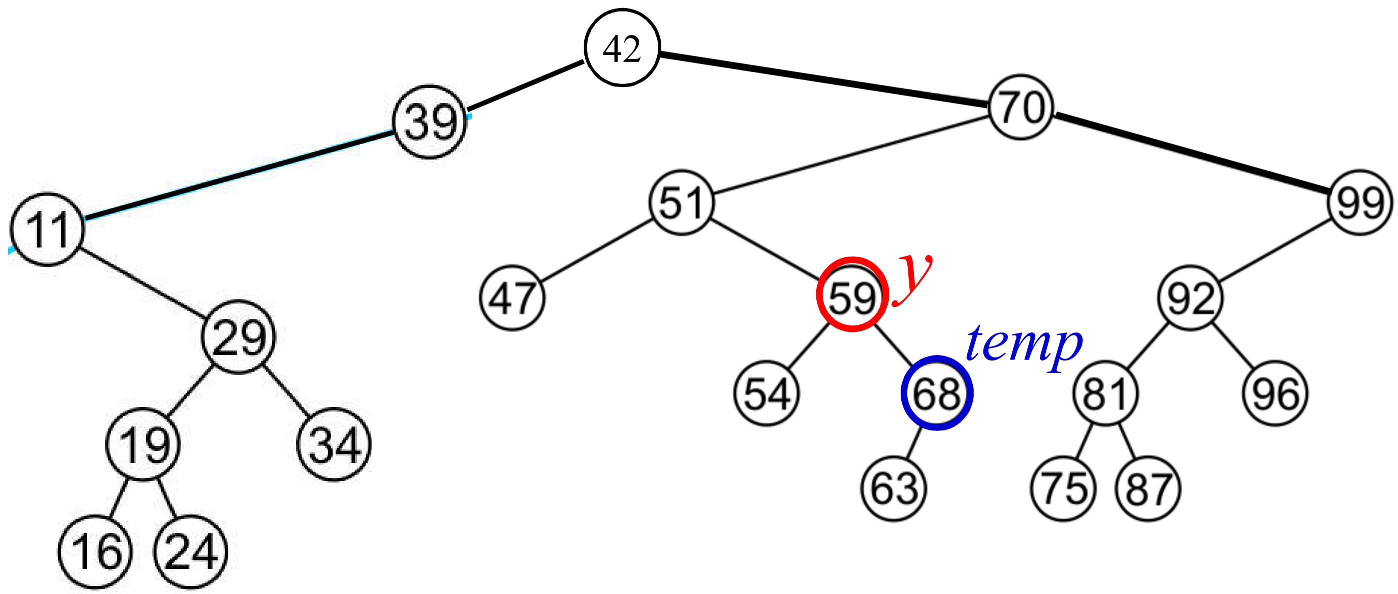
BST Operation: Successor

```
TREE_SUCCESSOR (x)
if x->right ≠ NULL
        return TREE_MINIMUM (x->right)
temp = x; y = temp->parent
while y ≠ NULL and temp == y->right
        temp = y
        y = y-> parent
return y
```



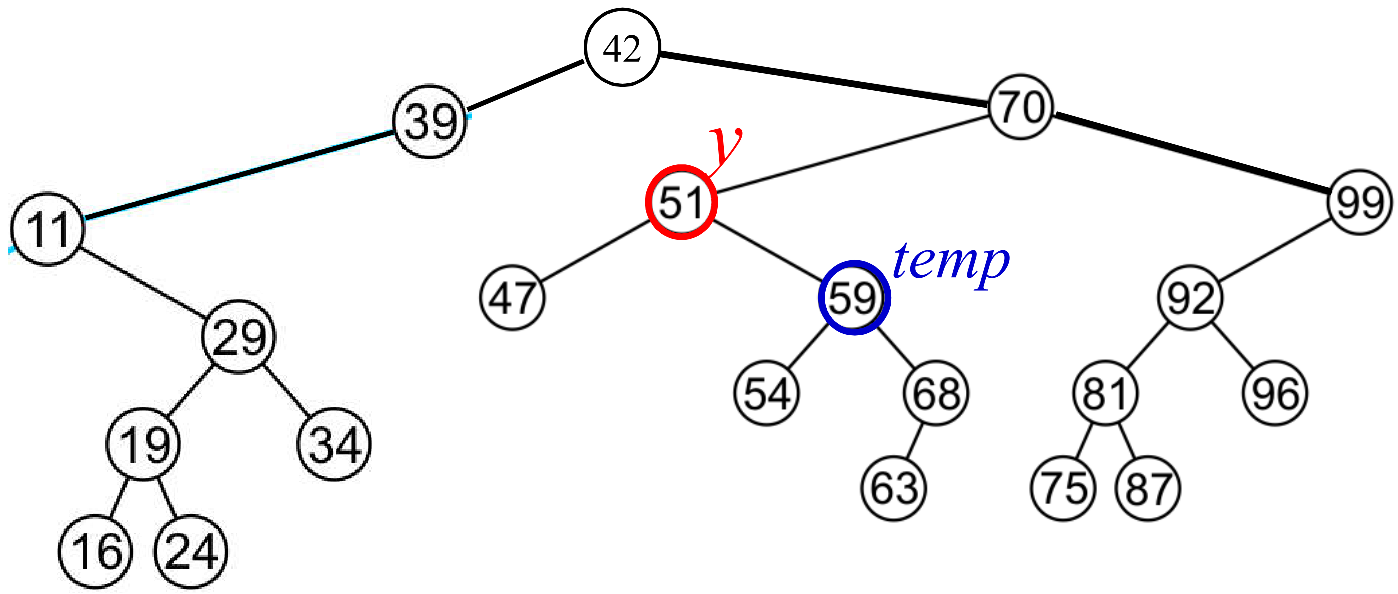
BST Operation: Successor

```
TREE_SUCCESSOR (x)
if x->right ≠ NULL
        return TREE_MINIMUM (x->right)
temp = x; y = temp->parent
while y ≠ NULL and temp == y->right
        temp = y
        y = y-> parent
return y
```



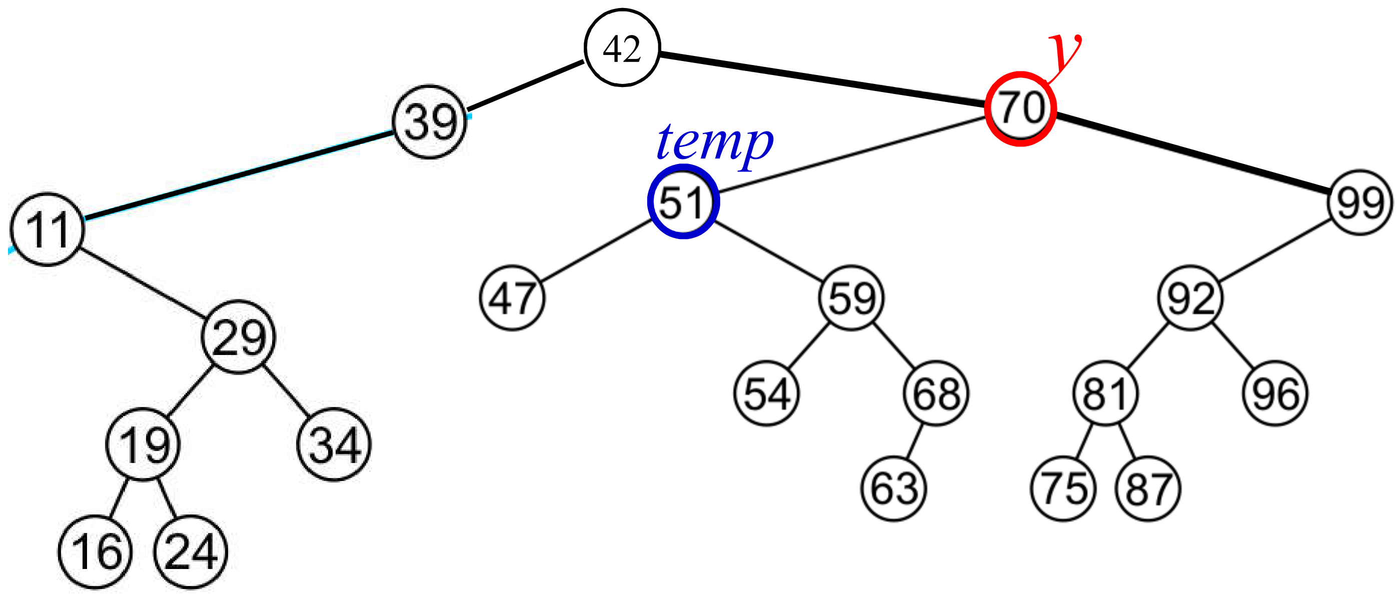
BST Operation: Successor

```
TREE_SUCCESSOR (x)
if x->right ≠ NULL
        return TREE_MINIMUM (x->right)
temp = x; y = temp->parent
while y ≠ NULL and temp == y->right
        temp = y
        y = y-> parent
return y
```



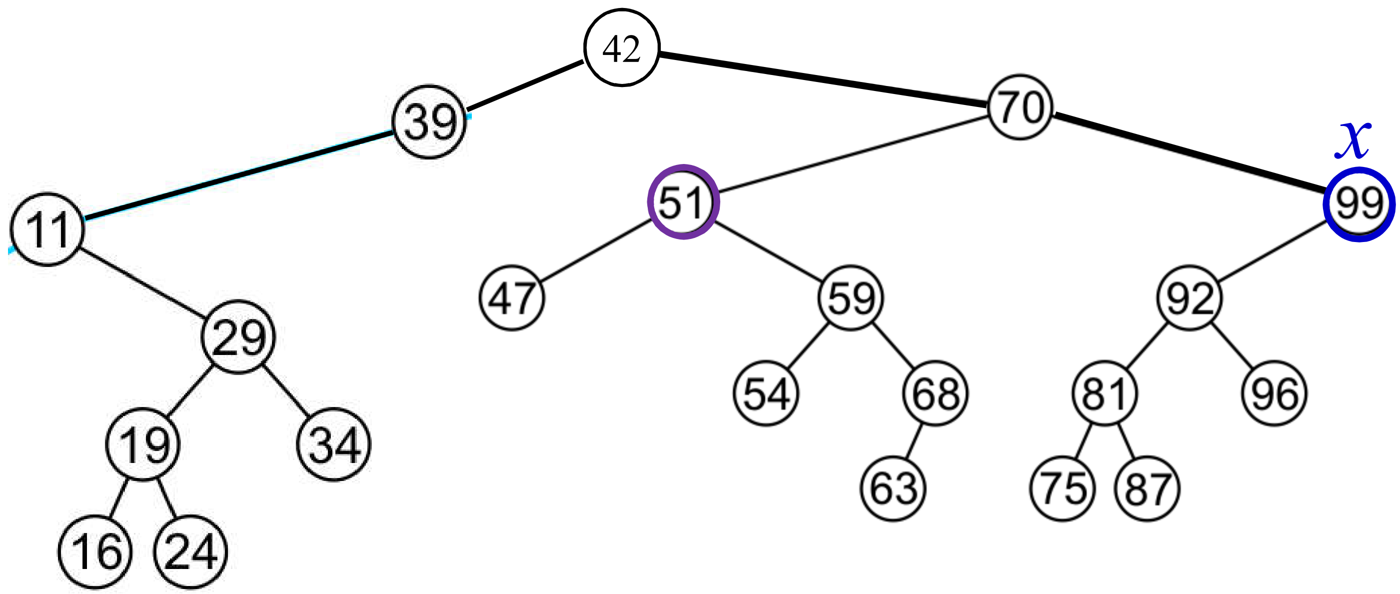
BST Operation: Successor

```
TREE_SUCCESSOR (x)
if x->right ≠ NULL
        return TREE_MINIMUM (x->right)
temp = x; y = temp->parent
while y ≠ NULL and temp == y->right
        temp = y
        y = y-> parent
return y
```



BST Operation: Successor

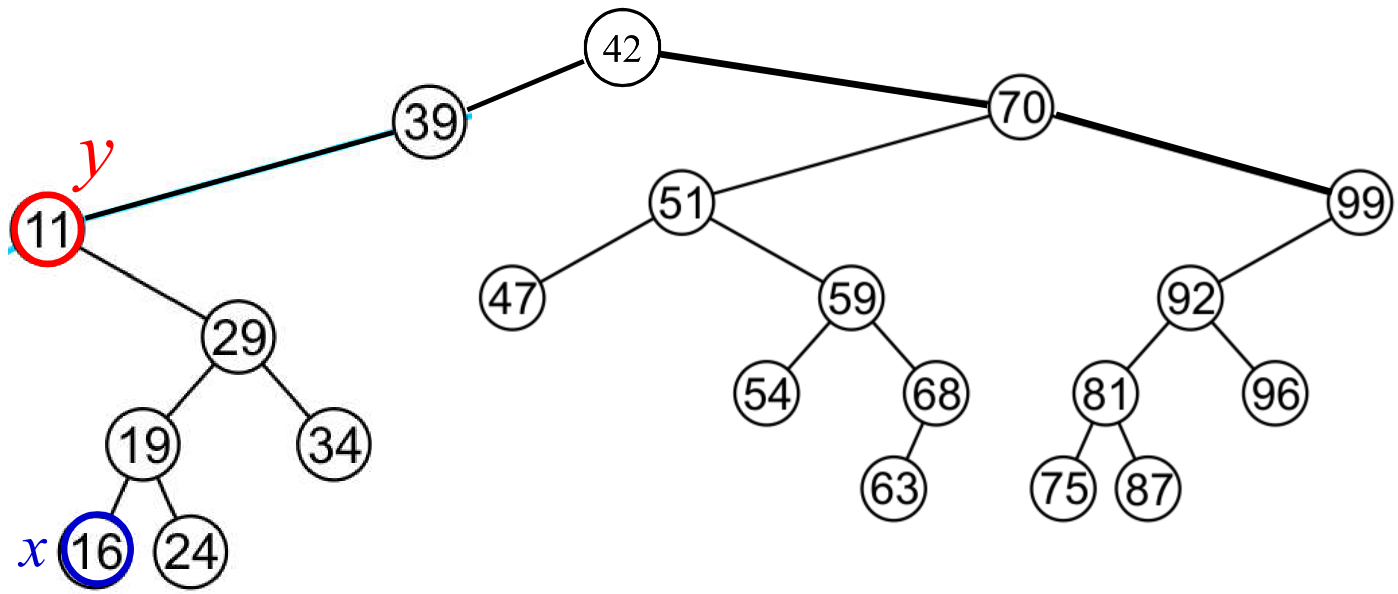
```
TREE_SUCCESSOR (x)
if x->right ≠ NULL
        return TREE_MINIMUM (x->right)
temp = x; y = temp->parent
while y ≠ NULL and temp == y->right
        temp = y
        y = y-> parent
return y
```



Predecessor of x ?

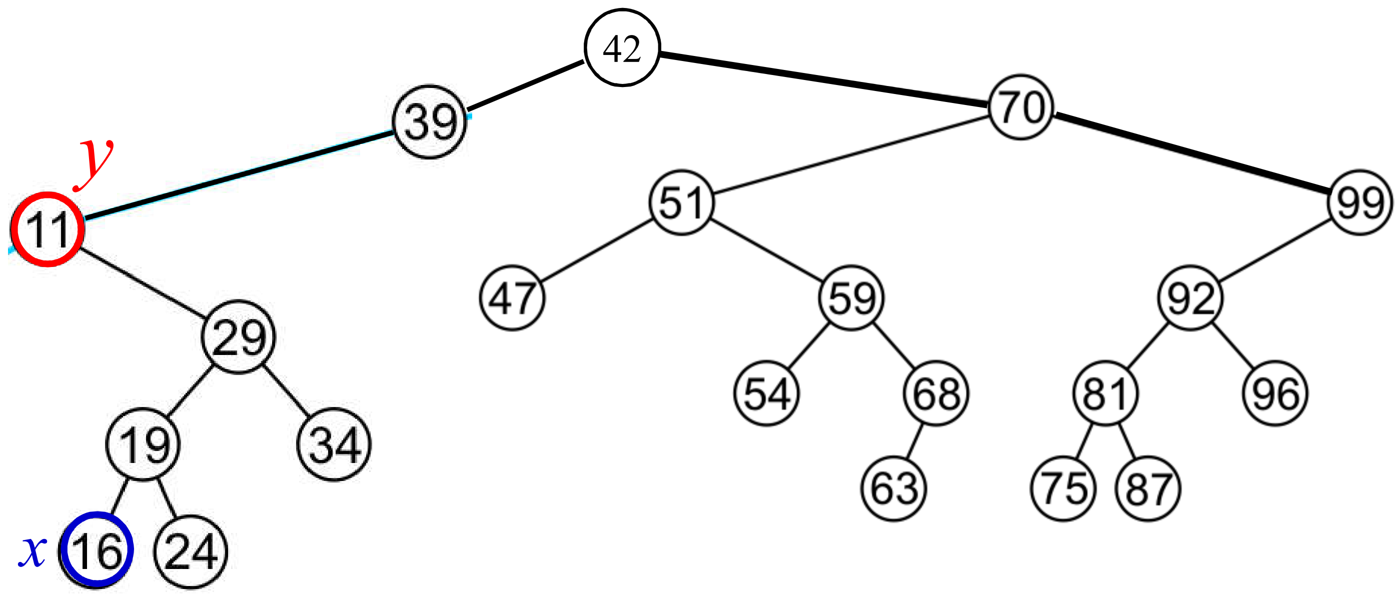
BST Operation: Predecessor

```
TREE_PREDECESSOR (x)
if x->left ≠ NULL
        return TREE_MAXIMUM (x->left)
temp = x; y = temp->parent
while y ≠ NULL and temp == y->left
        temp = y
        y = y-> parent
return y
```



BST Operation: Predecessor

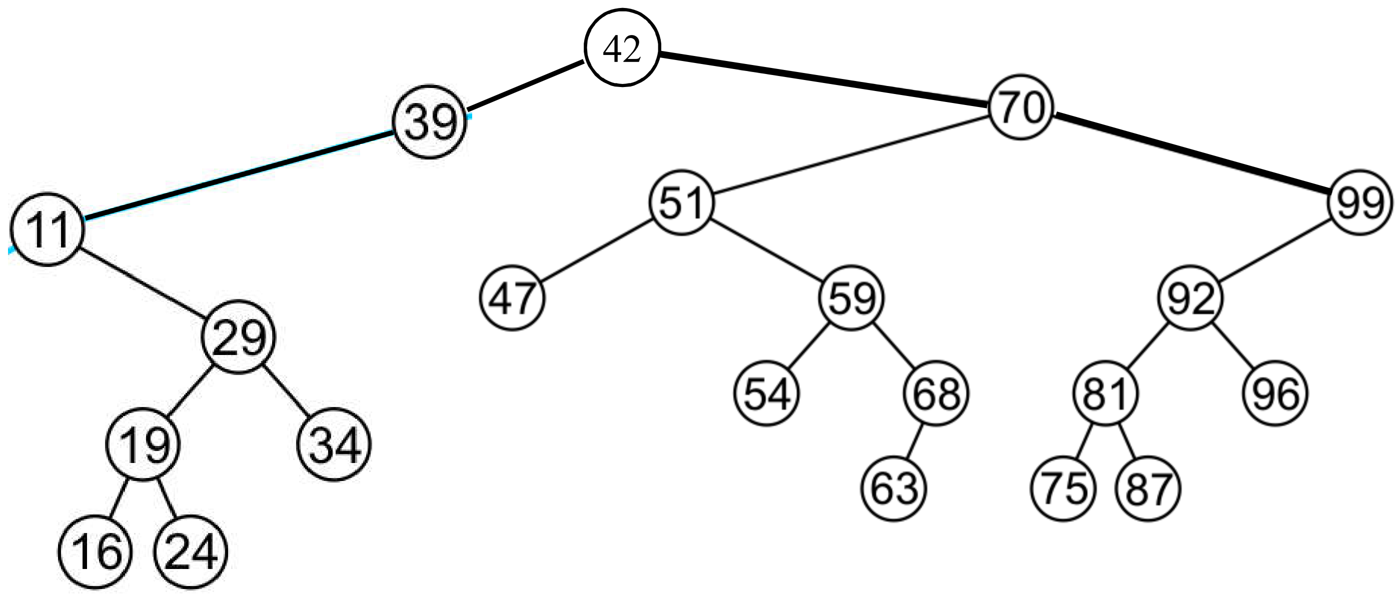
```
TREE_PREDECESSOR (x)
if x->left ≠ NULL
        return TREE_MAXIMUM (x->left)
temp = x; y = temp->parent
while y ≠ NULL and temp == y->left
        temp = y
        y = y-> parent
return y
```



Complexity O(h)

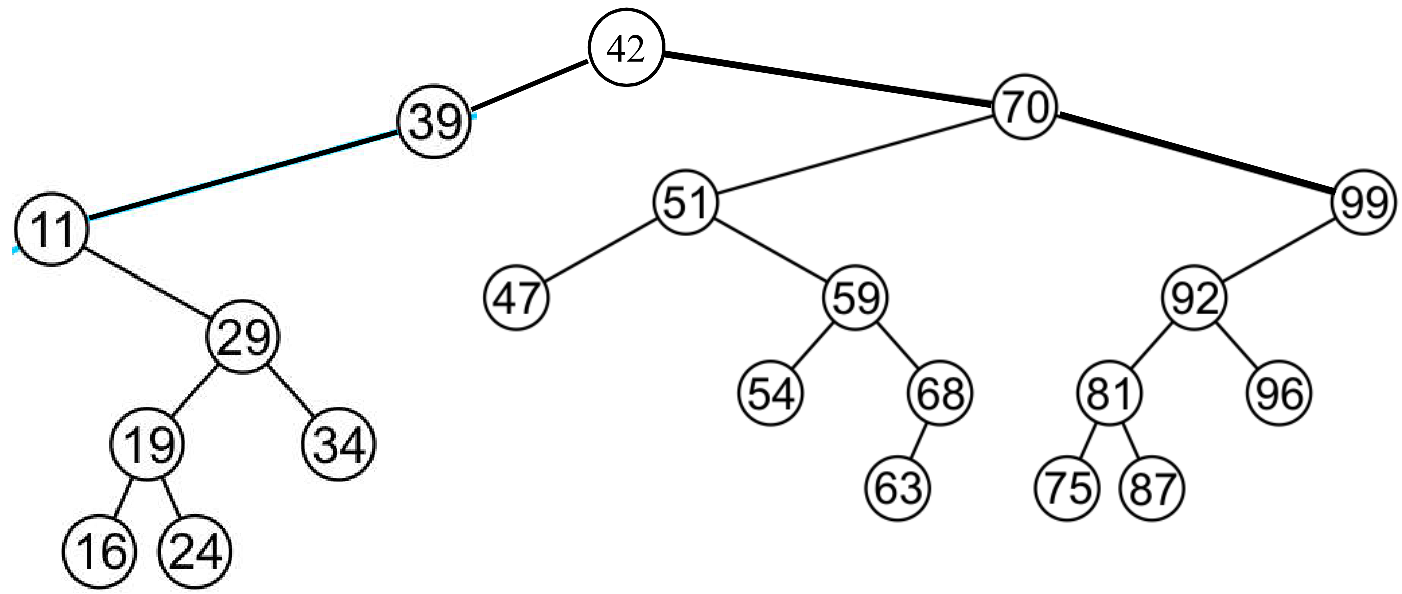
BST Operation: Insertion

An insertion will be performed at a leaf node



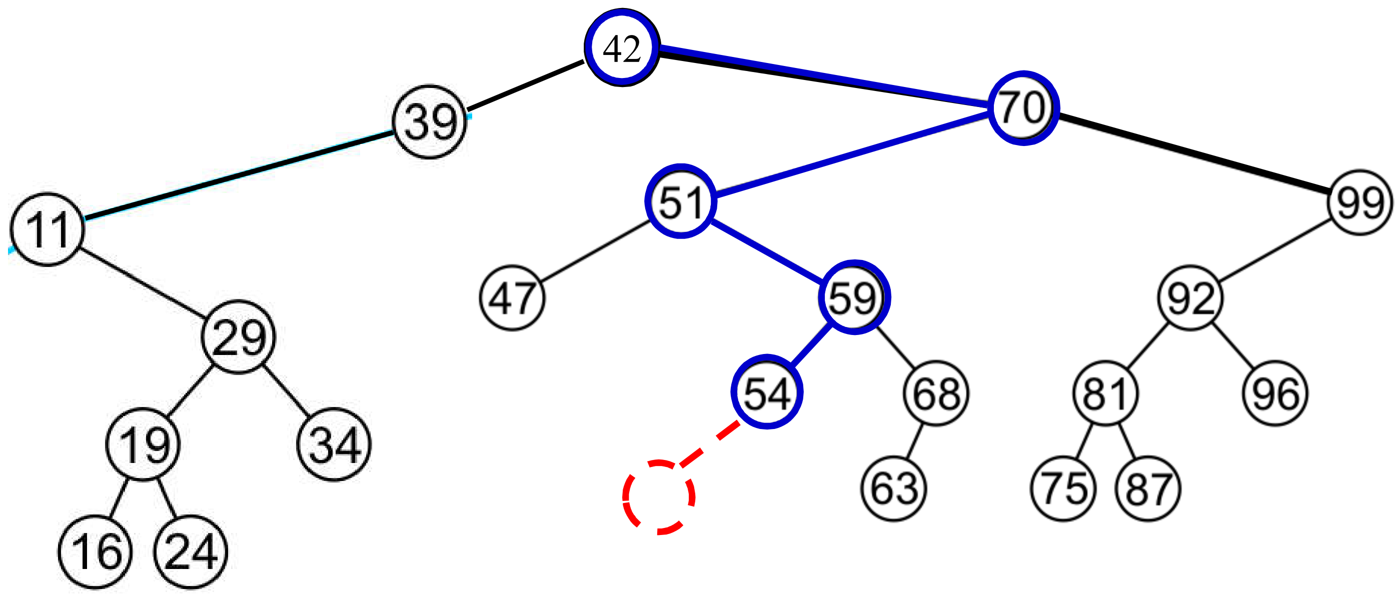
BST Operation: Insertion

Given a *key* to insert, find the location if it were in the tree



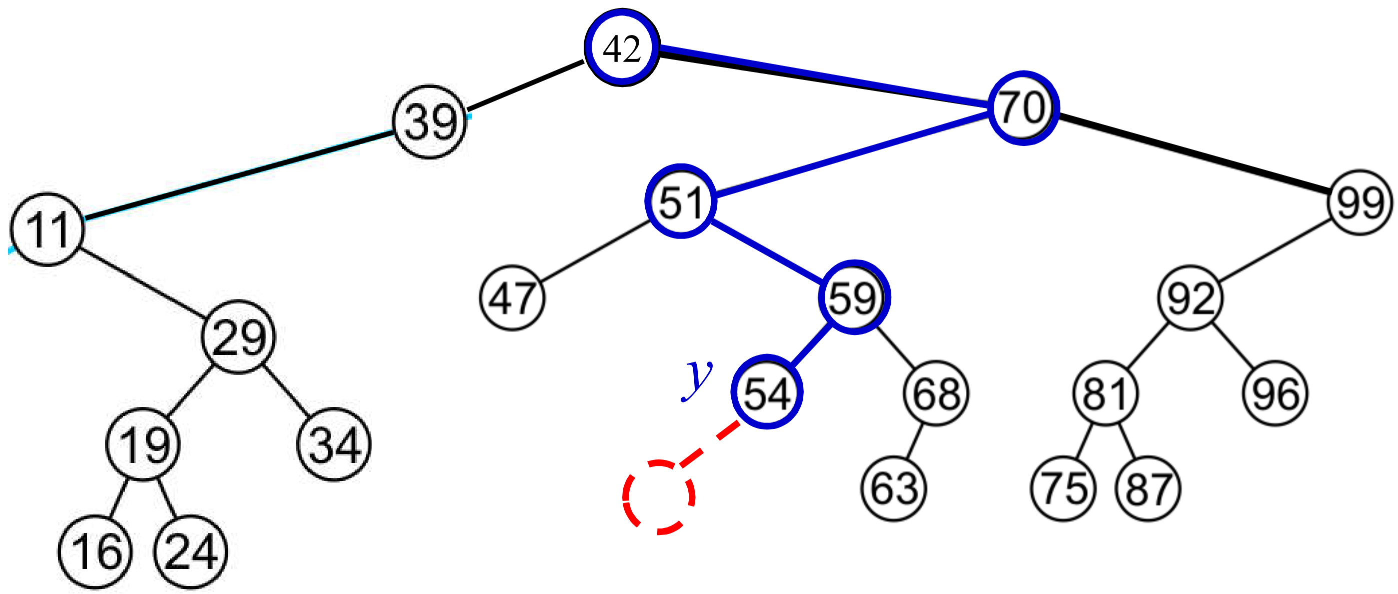
BST Operation: Insertion

To insert a node *z* with key 52



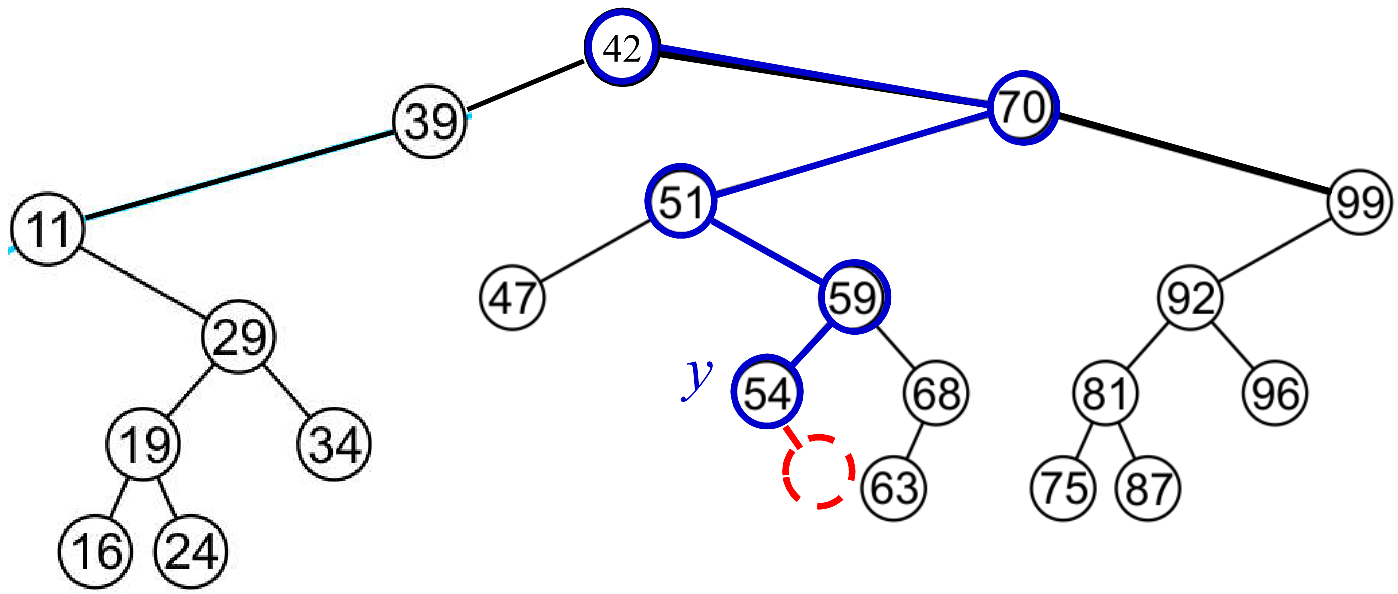
BST Operation: Insertion

To insert a node z with key 52



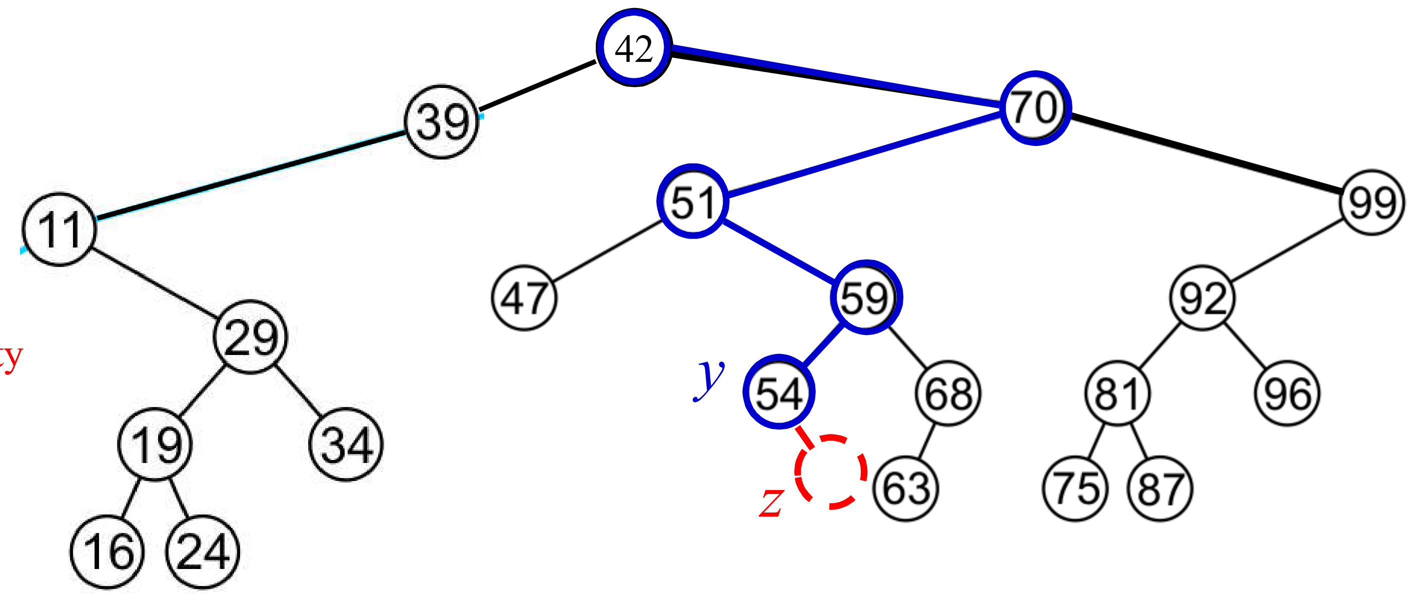
BST Operation: Insertion

To insert a node z with key 55



BST Operation: Insertion

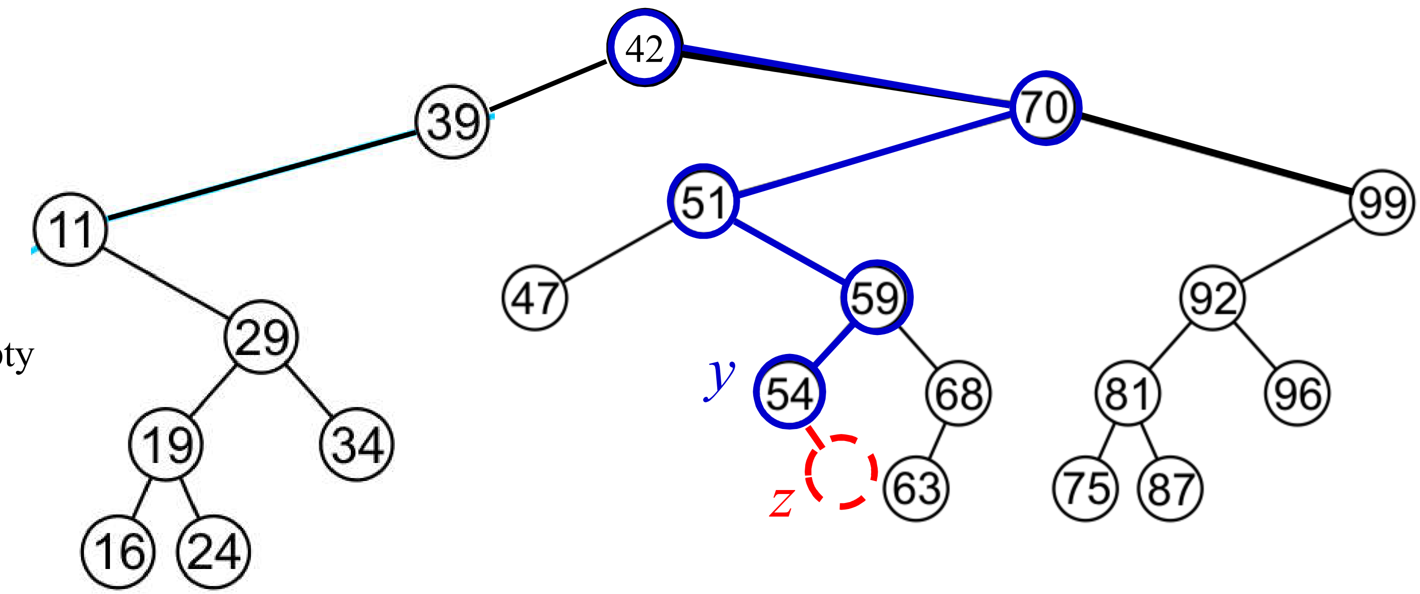
```
TREE_INSERT (T, z)

y = NULL
x = T->root

while x ≠¤ NULL
      y = x
      if z->key < x->key
            x = x->left
      else x = x->right

z->parent = y

if y == NULL
    T->root =z // tree T was empty
elseif z->key < y->key
      y->left =z
else y->right =z
```



BST Operation: Insertion

```
TREE_INSERT (T, z)

y = NULL
x = T->root

while x ≠¤ NULL
    y = x
    if z->key < x->key
        x = x->left
    else x = x->right

z->parent = y

if y == NULL
    T->root =z // tree T was empty
elseif z->key < y->key
    y->left =z
else y->right =z
```

Complexity: O(h)



BST Operation: Deletion

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

There are three possible scenarios:

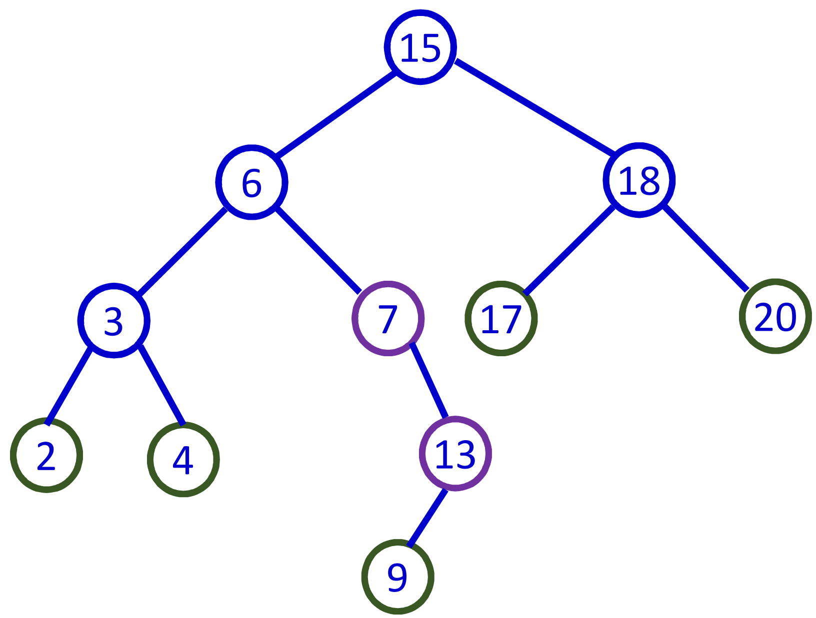
- The node is a leaf node
- It has exactly one child, or
- It has two children (it is a full node)

BST Operation: Deletion

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

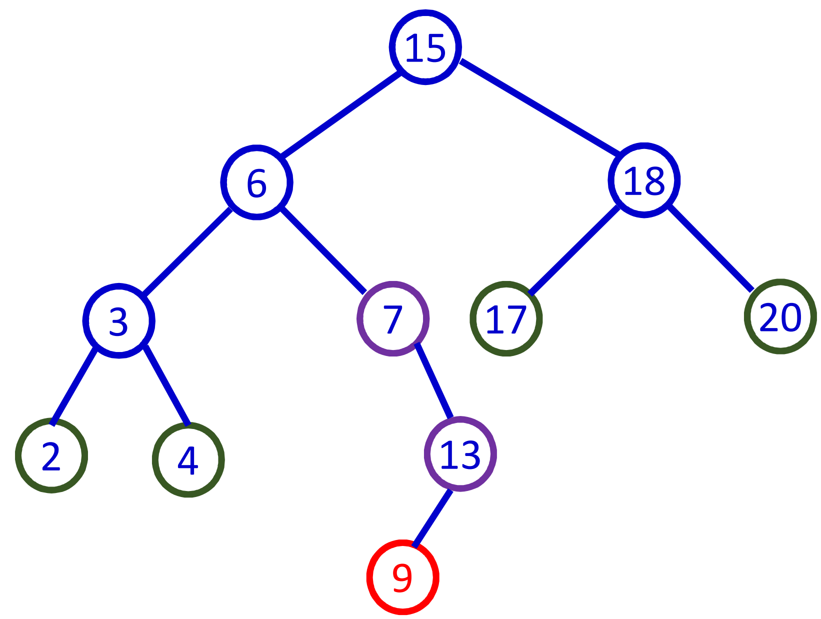
There are three possible scenarios:

- 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

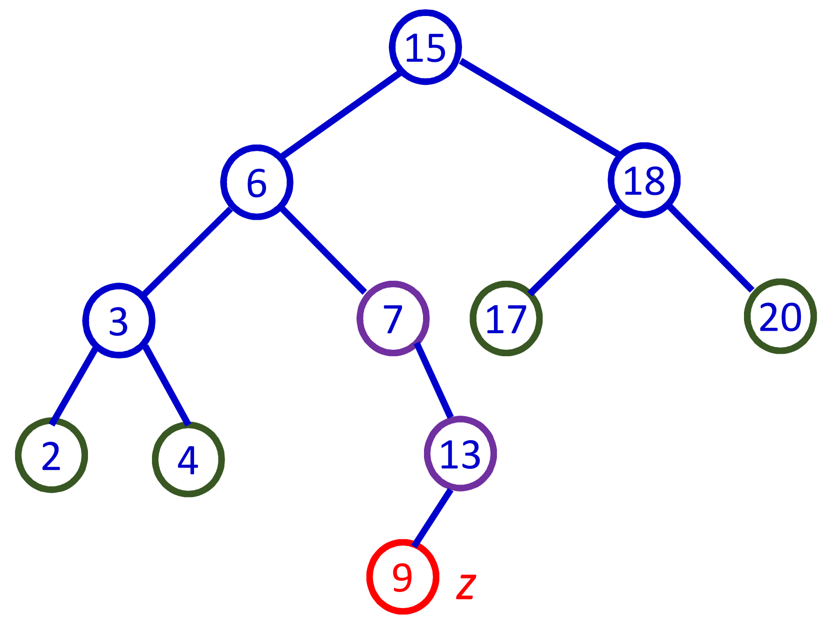
Removing a leaf node

Set left pointer of 13 as NULL

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

z->parent ->left =NULL

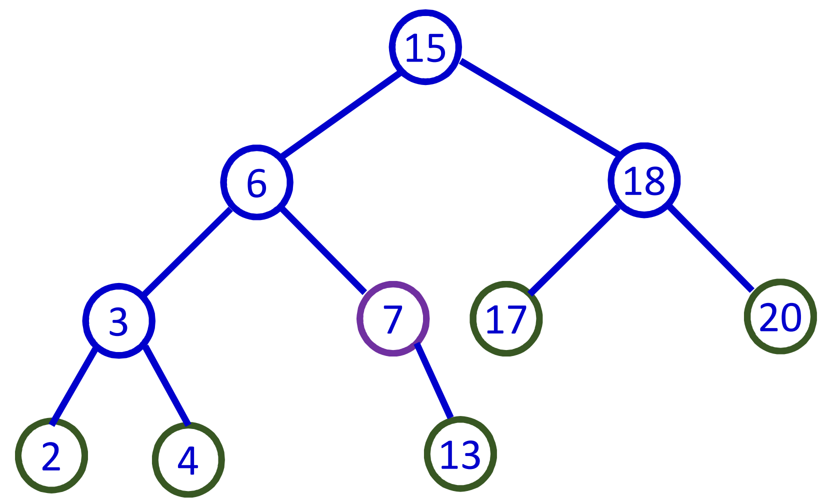
else *z->parent ->right* =NULL



BST Operation: Deletion

Removing a leaf node
Set left pointer of 13 as NULL

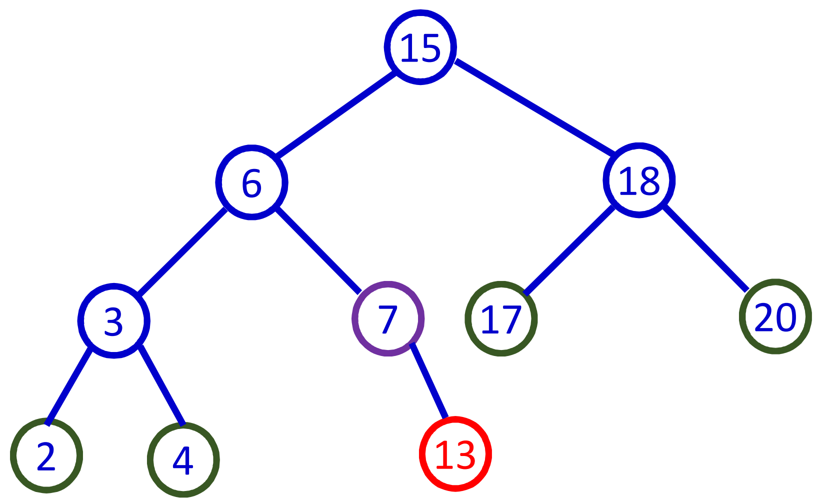
If *z== z->parent ->left*
z->parent ->left =NULL
else *z->parent ->right* =NULL



BST Operation: Deletion

Removing a leaf node with key 13
Set right pointer of 7 as NULL

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



BST Operation: Deletion

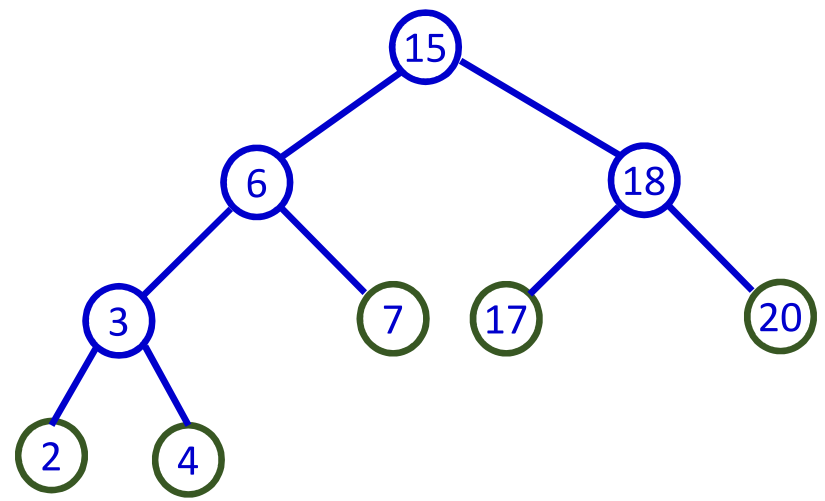
Removing a leaf node with key 13

Set right pointer of 7 as NULL

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

z->parent ->left =NULL

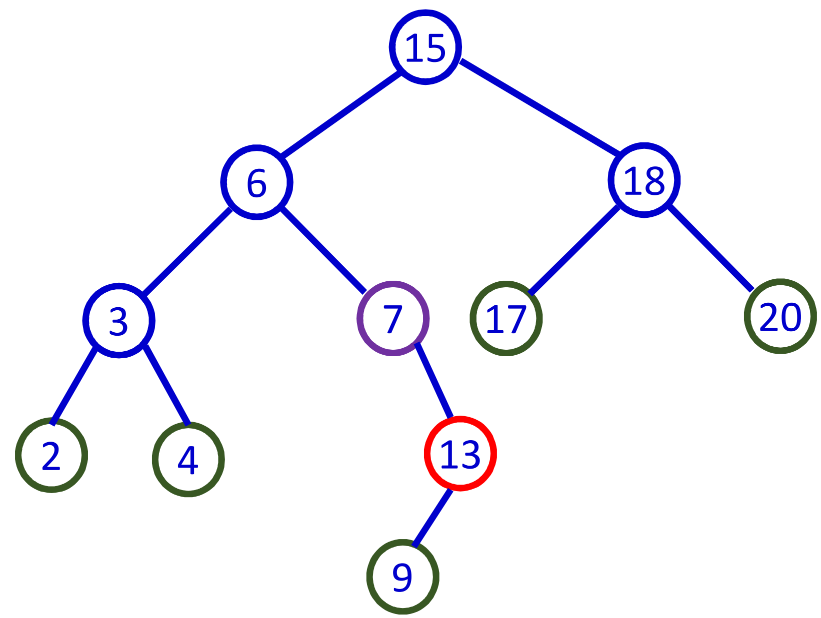
else *z->parent ->right* =NULL



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

Remove node with key 13 which has a left subtree ONLY

Promote the left subtree

If *z->left* =NULL

x= z->right

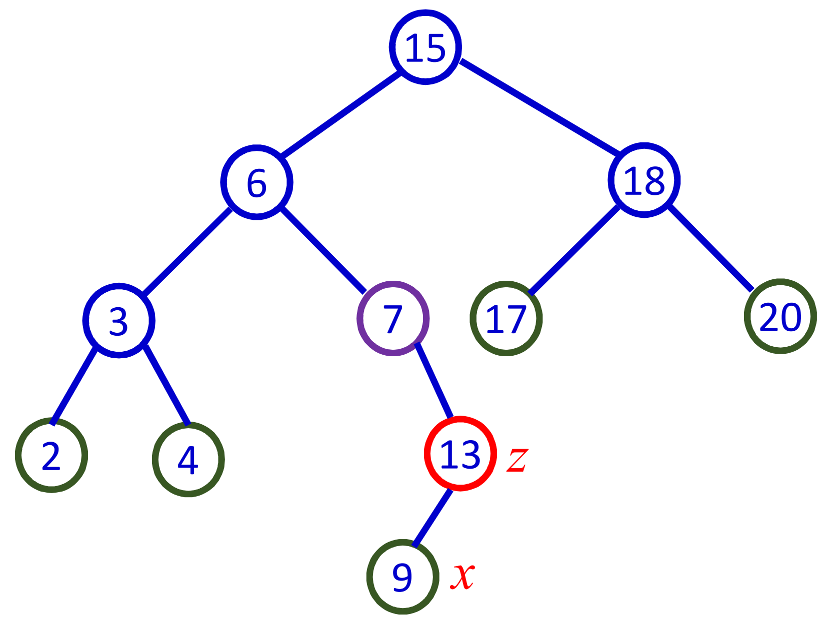
else *x= z->left*

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

z->parent ->left =x

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

x->parent = z->parent



BST Operation: Deletion

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*

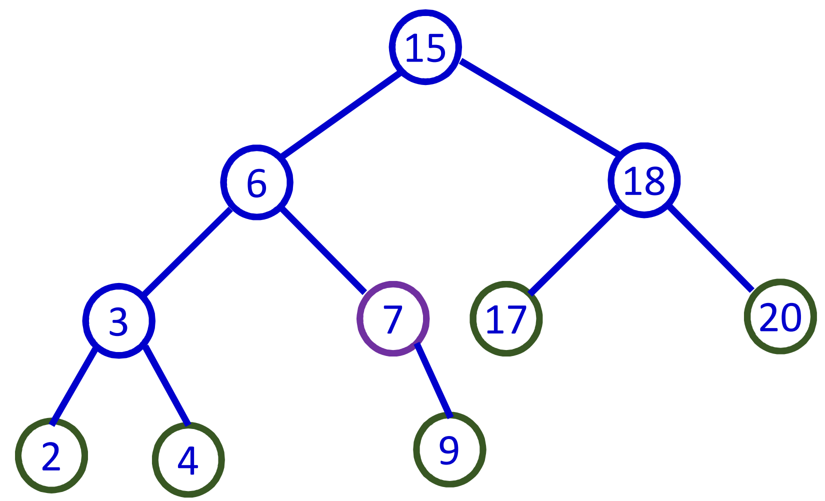
else *x*= *z*->*left*

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

    z->*parent* ->*left* =*x*

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

x->*parent* = *z*->*parent*



BST Operation: Deletion

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*

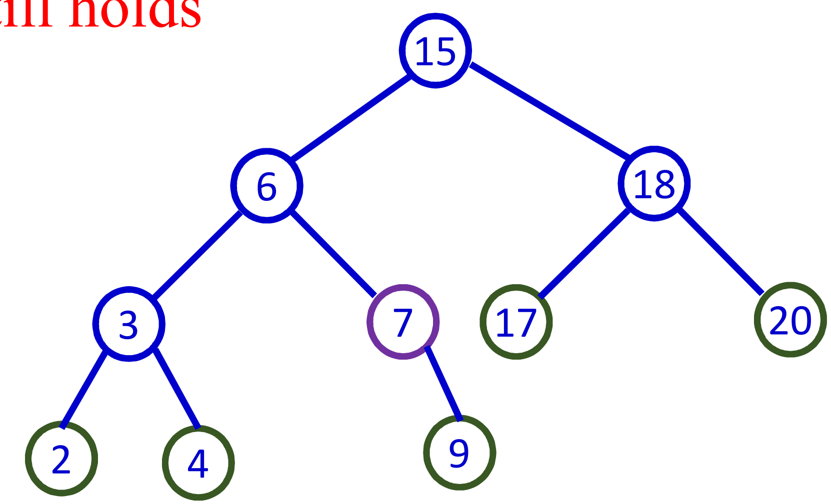
If *z== z->parent ->left*

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*

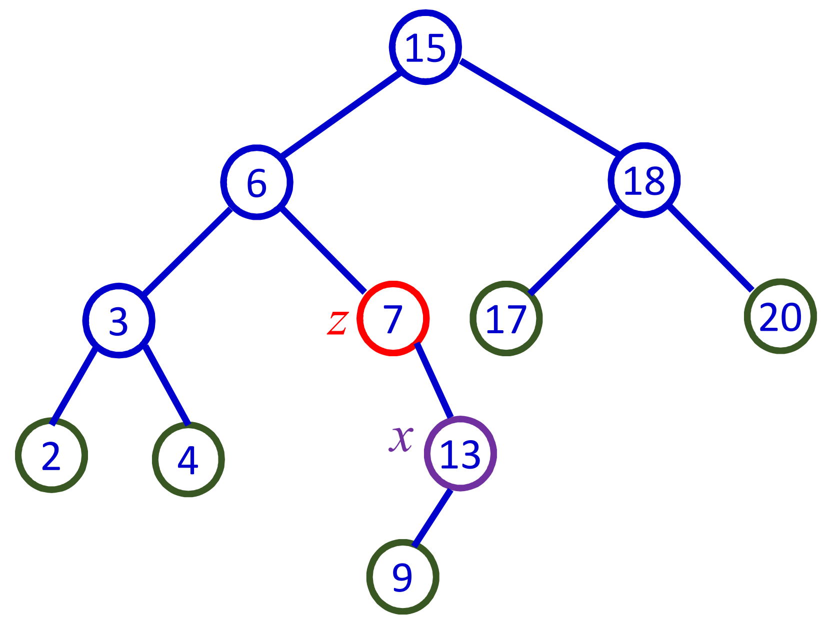
else *x*= *z*->*left*

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

    z->*parent* ->*left* =*x*

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

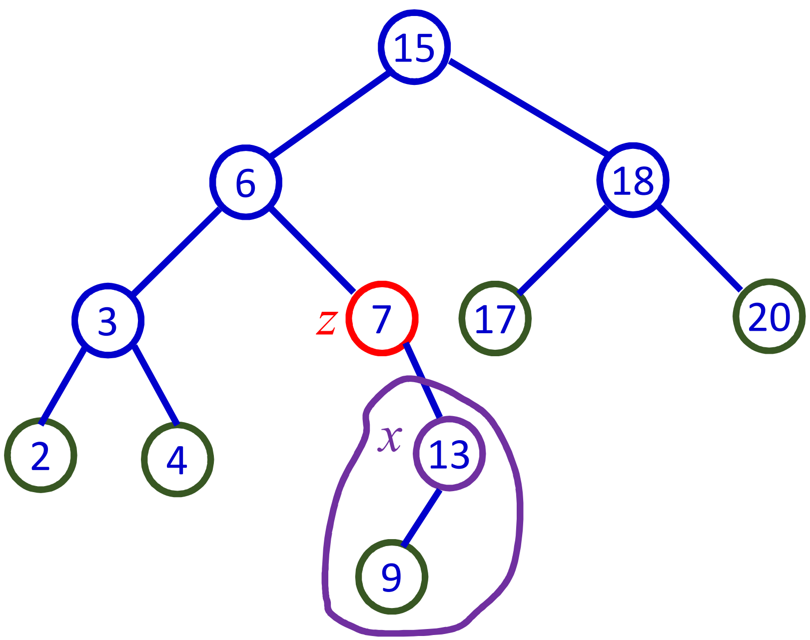
x->*parent* = *z*->*parent*

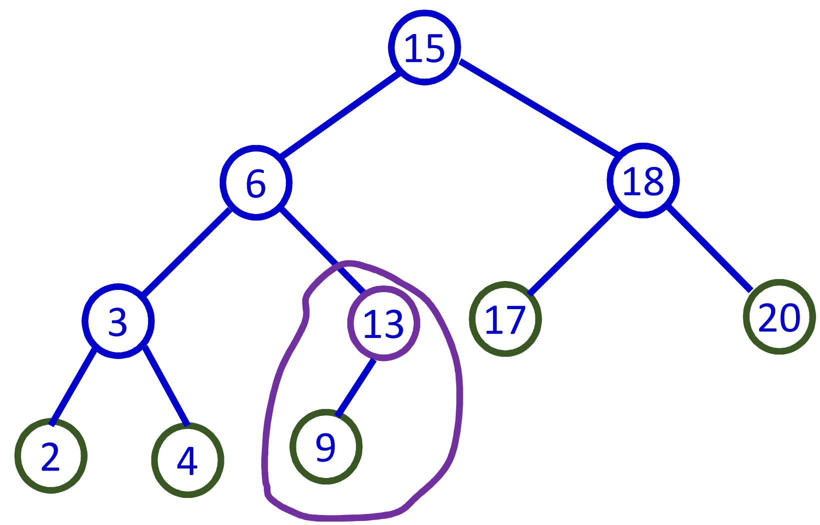


BST Operation: Deletion

Removing a node with exactly one child

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



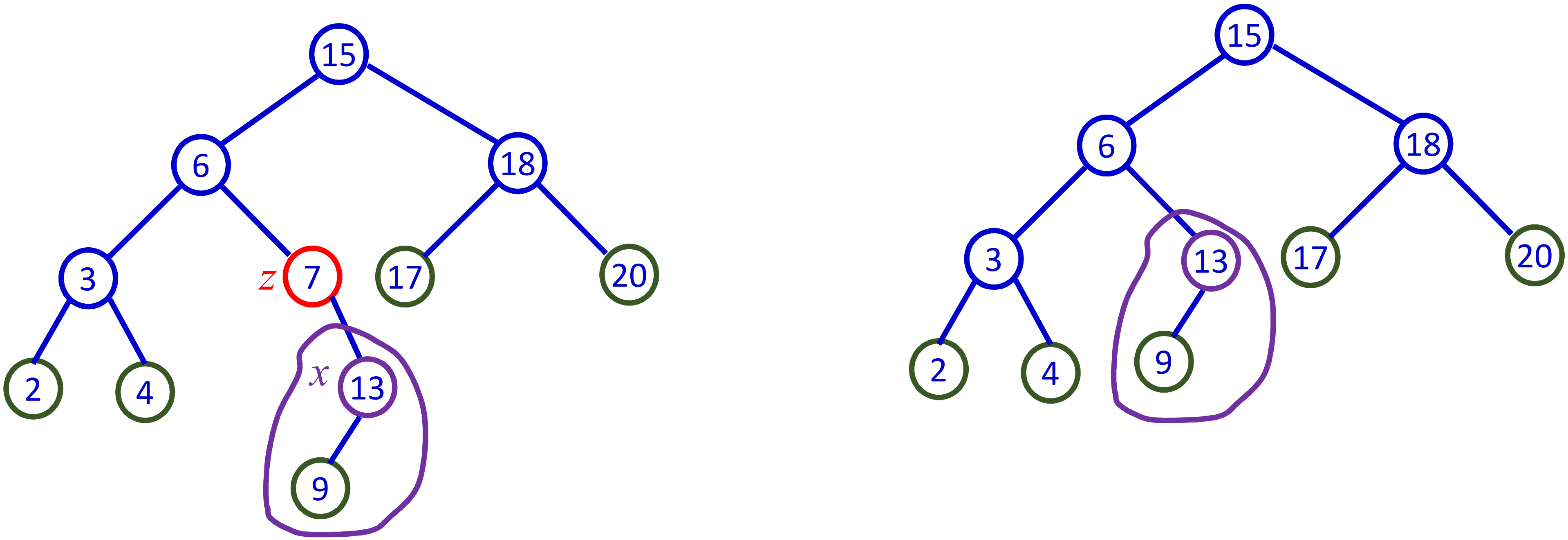


BST Operation: Deletion

Removing a node with exactly one child

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

BST property holds

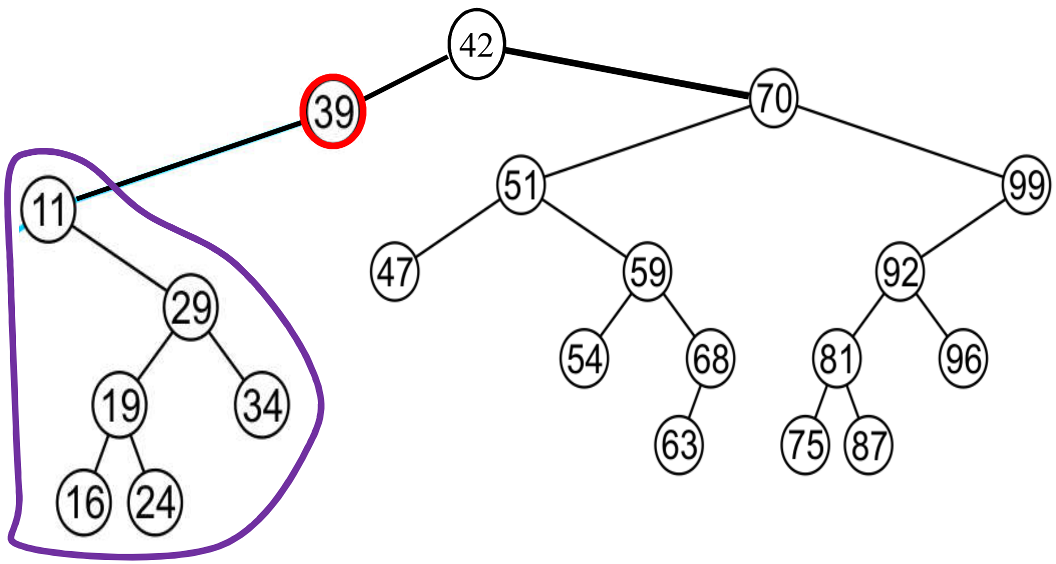


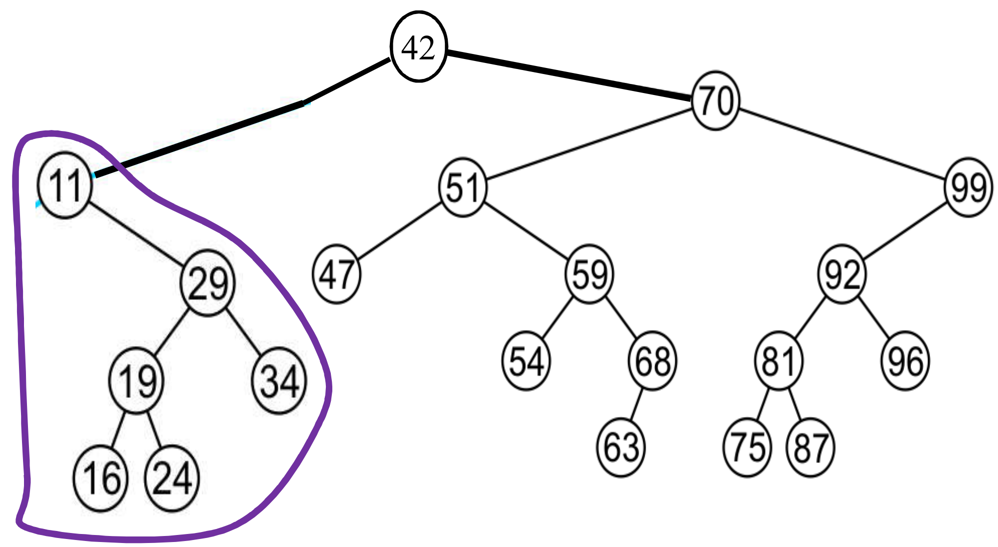
BST Operation: Deletion

Removing a node with exactly one child

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

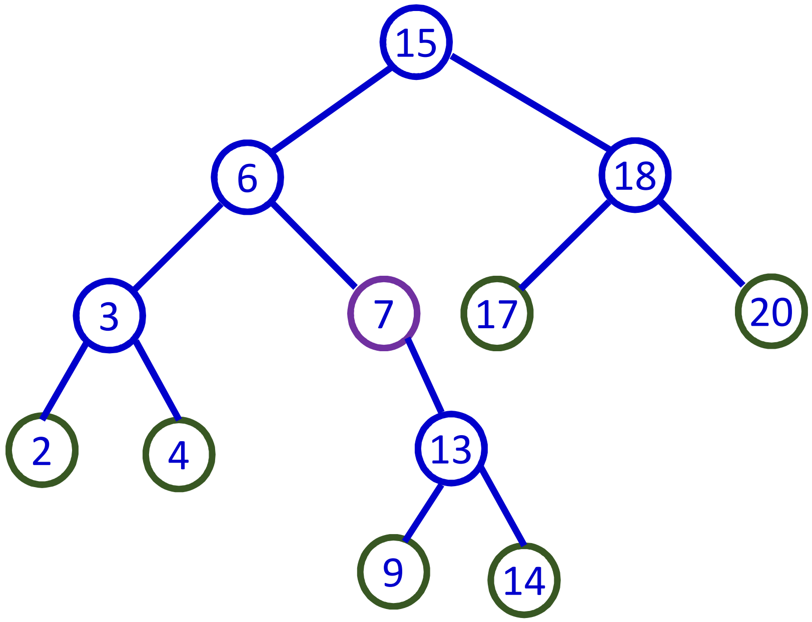
BST property holds





BST Operation: Deletion

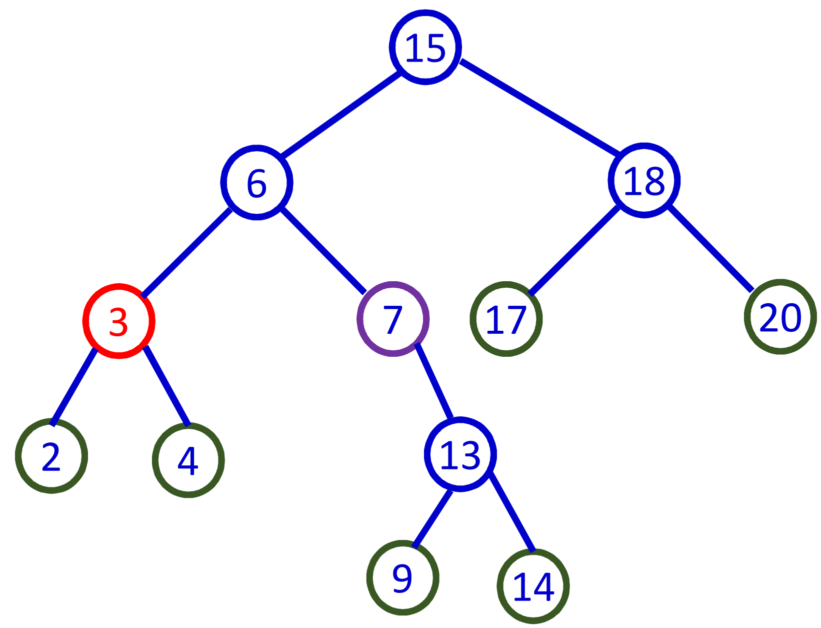
Removing a node having two children (full node)



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

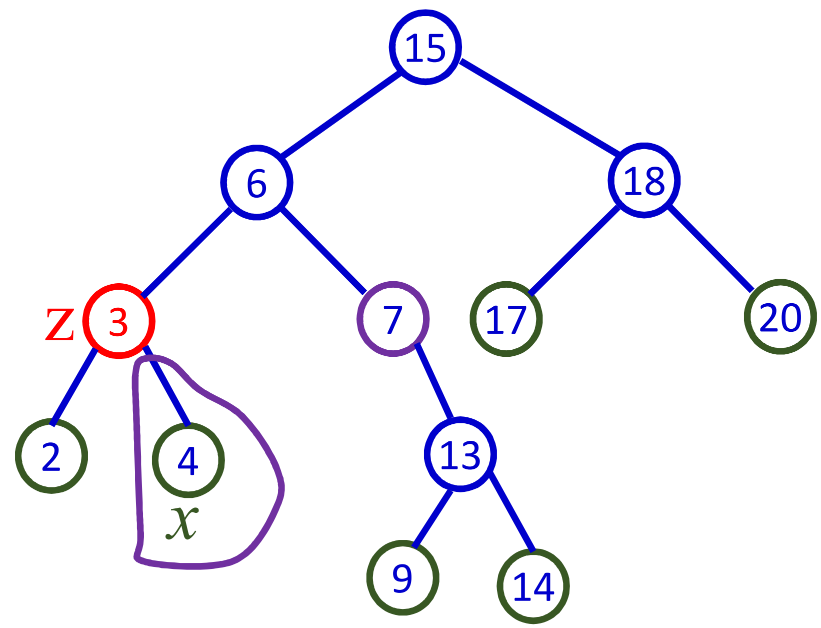
Idea:

Find the successor of the node with key 3

the successor must be in right subtree

Copy successor's key to node z

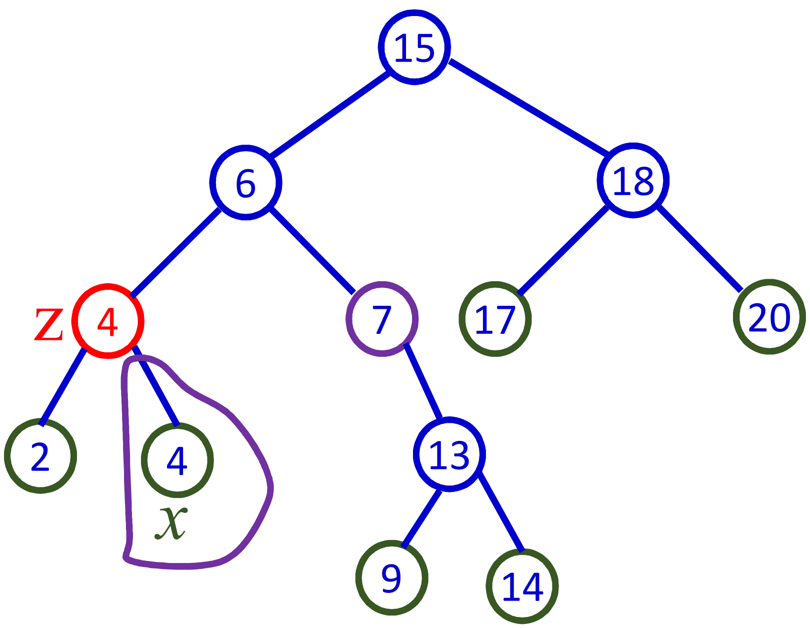
Delete successor x



BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 3
Idea:
Find the successor of the node with key 3
the successor must be in right subtree
Copy successor's key to node z
Delete successor x



BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 3

Idea:

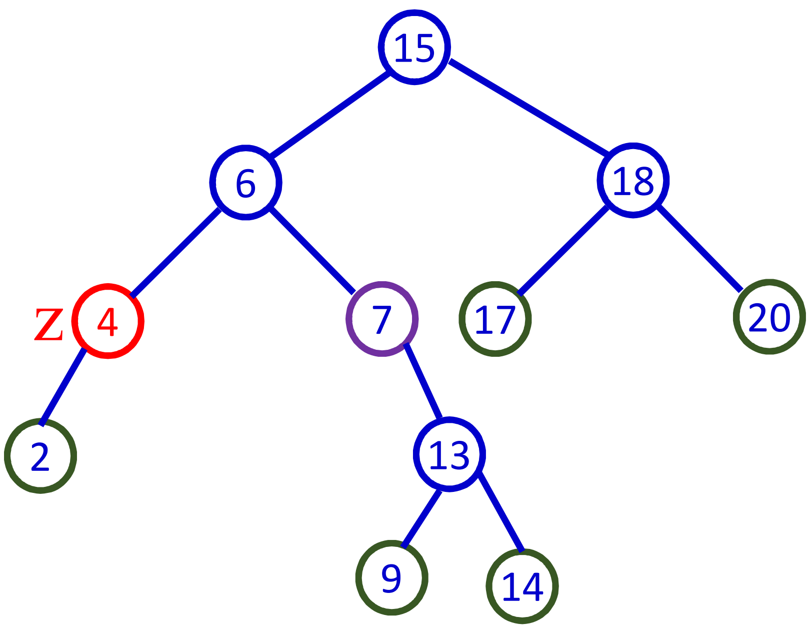
Find the successor of the node with key 3

the successor must be in right subtree

Copy successor's key to node z

Delete successor *x*

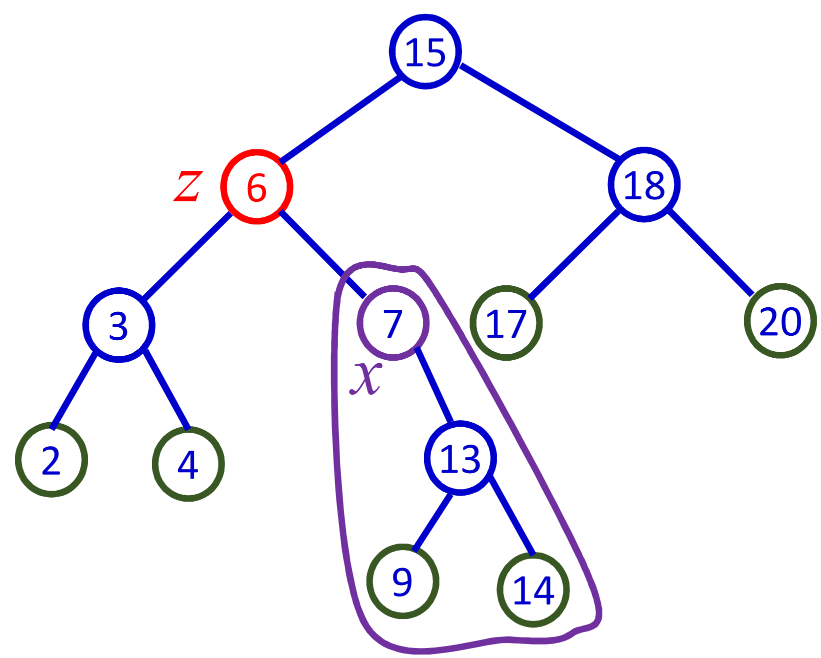
BST Property holds



BST Operation: Deletion

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

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

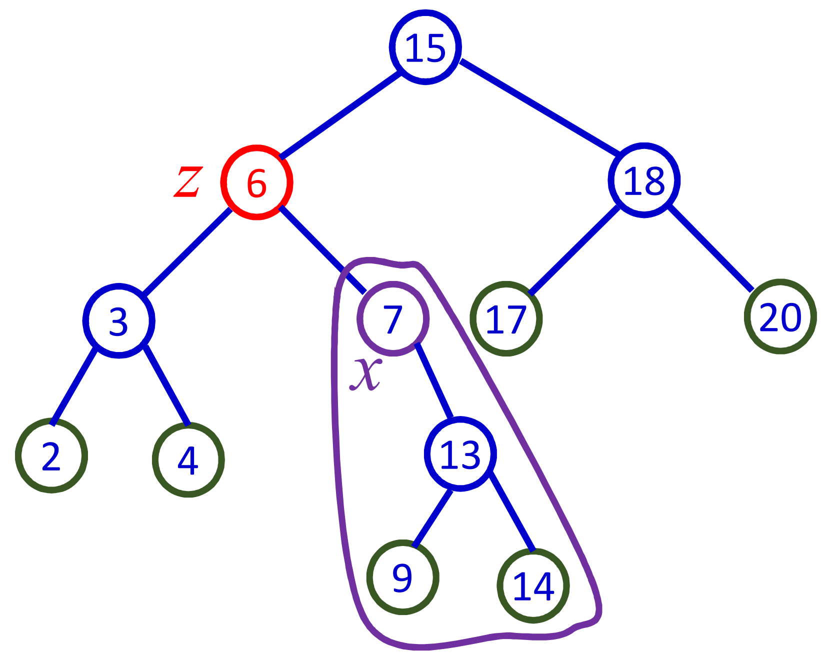


BST Operation: Deletion

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



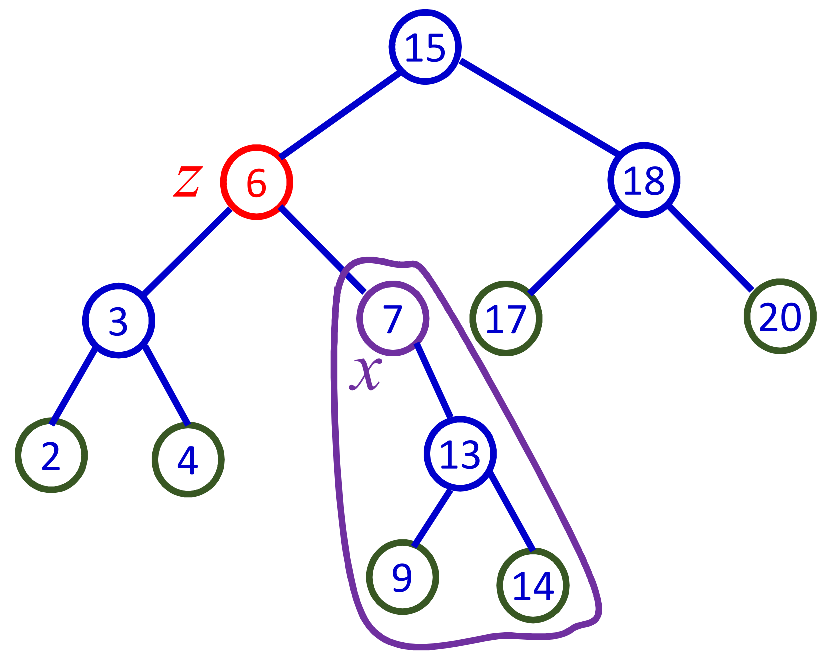
BST Operation: Deletion

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

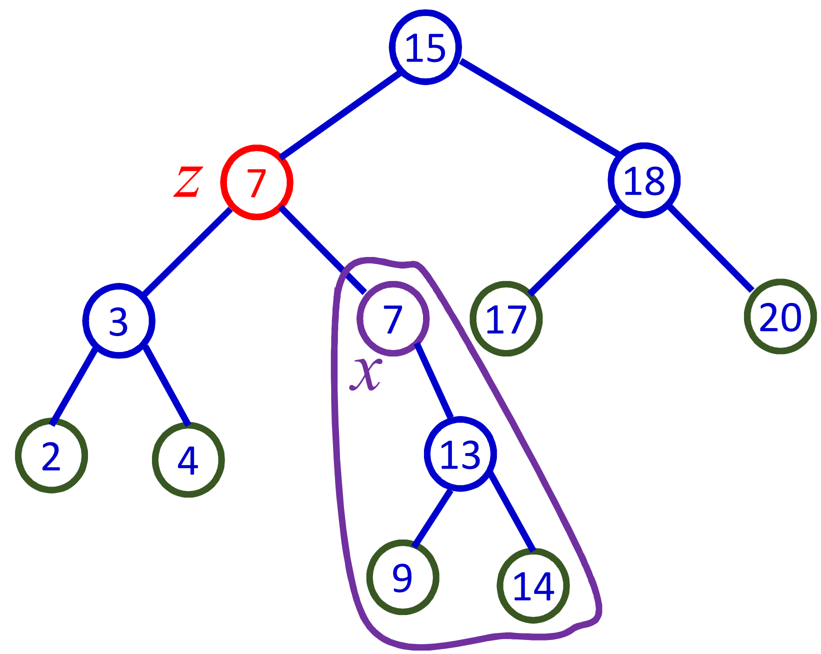
If x would have a left child
that would be the minimum of the subtree



BST Operation: Deletion

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

Now copy key of *x* to *z*



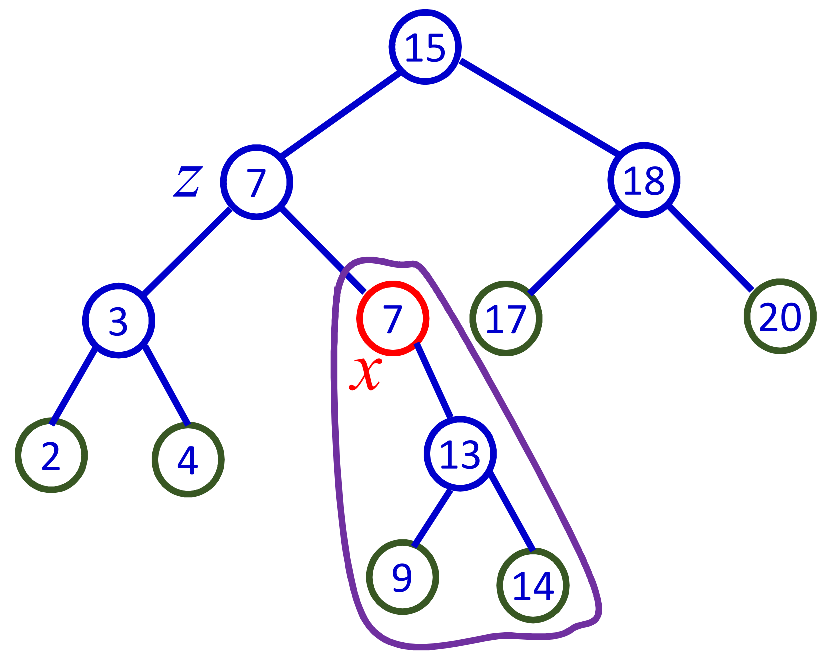
BST Operation: Deletion

Removing a node having two children (full node)

Remove node with 6

Now copy key of x to z

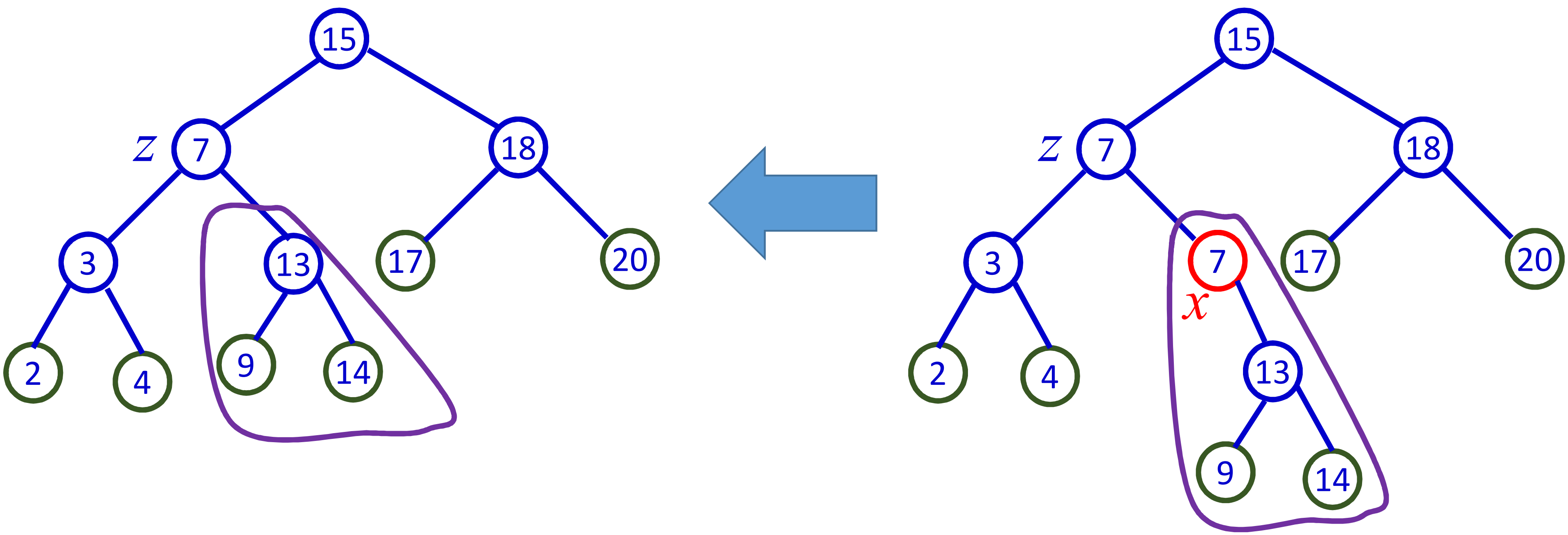
Delete node x (it has a single child ONLY)



BST Operation: Deletion

Removing a node having two children (full node)

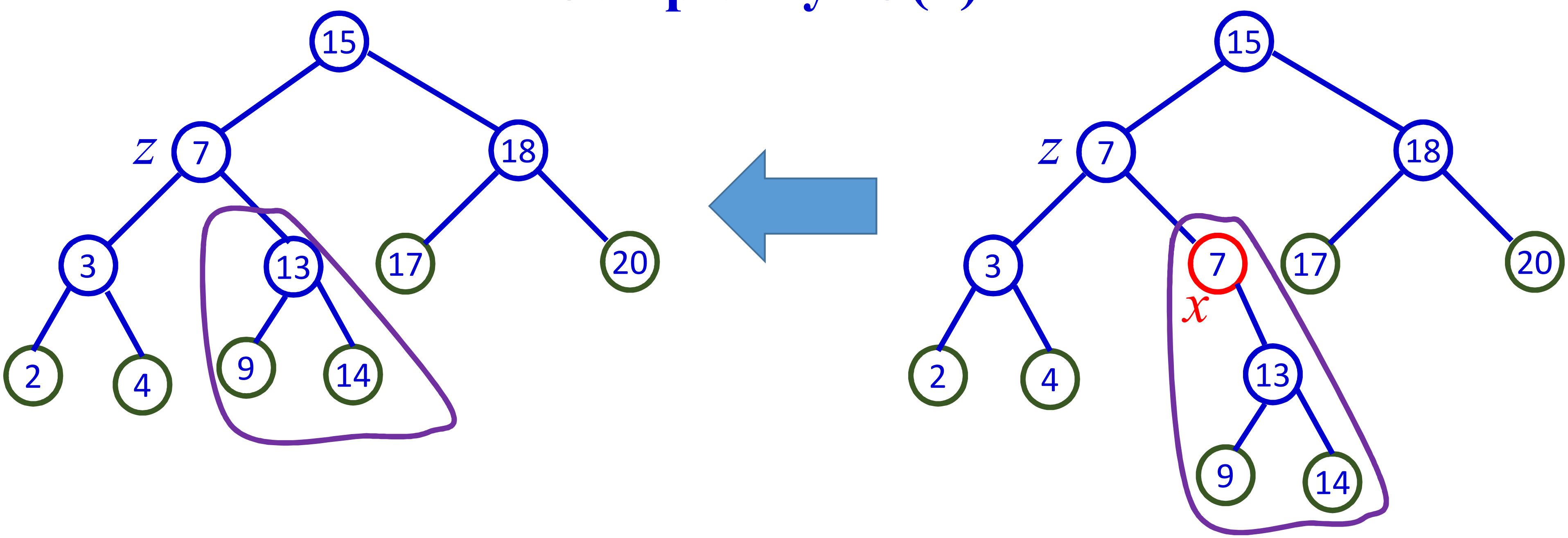
Remove node with 6



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)

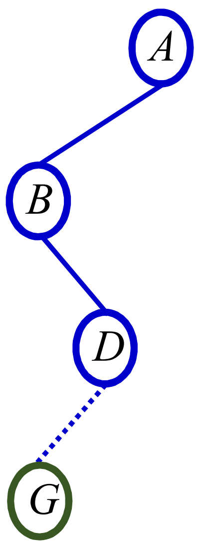
h = depth of the deepest node in the BST, i.e.,

$\approx$ height of the tree.

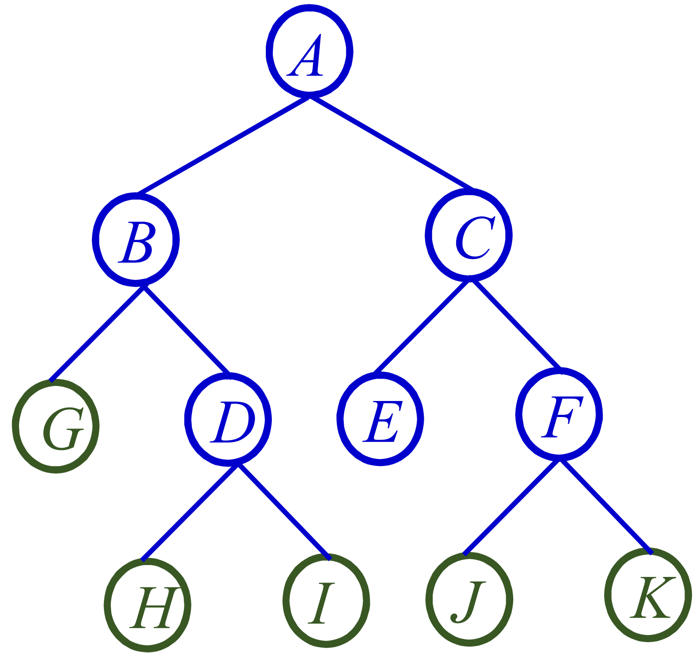
$\approx \log n$ if tree is balanced.

What is the worst case?

BST Operations: Complexity



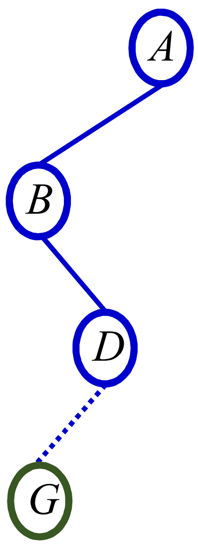
Chain, Imbalanced: height, $h \approx n$
All complexity: O(n)



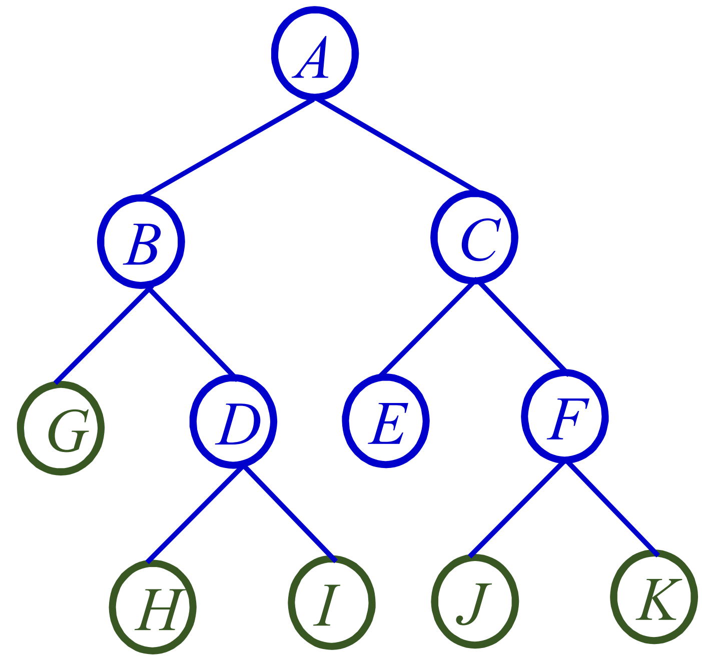
Balanced: height, h =log(n)
All complexity: O(log(n))

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))