CS302 – Analysis and Design of Algorithms

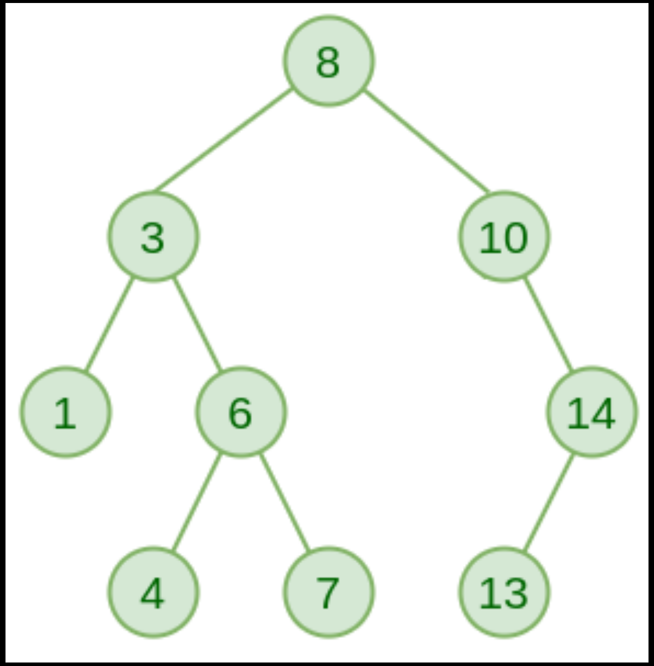
Binary Search Tree

Content

Content
➡ Binary Search Tree
Querying Binary Search Tree
Insertion and Deletion
Exercises

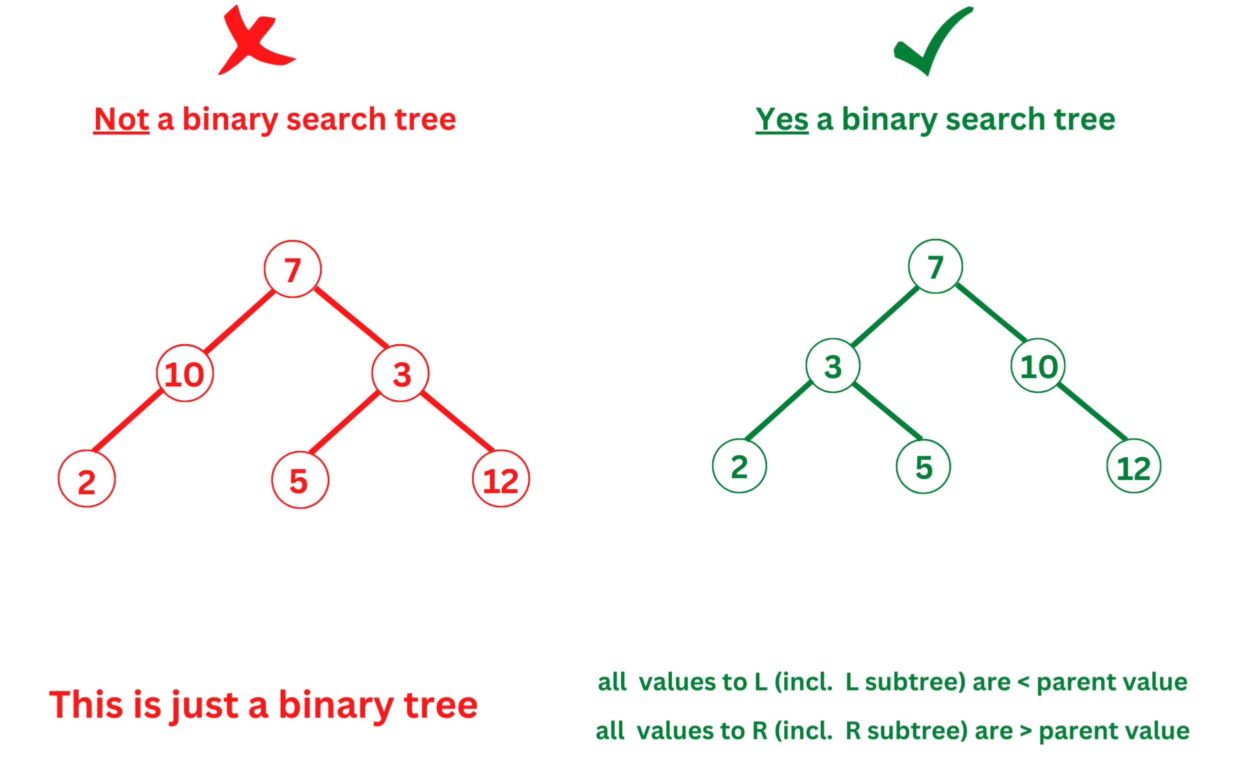
Binary Search Trees

- A BST is a data structure used for storing data in a sorted manner.
- Each node in has at most two children:
  - Left child – containing values less than the parent node
  - Right child – containing values greater than the parent node.



Binary Search Trees

- Left subtree contain keys less than the parent.
- Right subtree contain keys greater than the parent.



Not a binary search tree

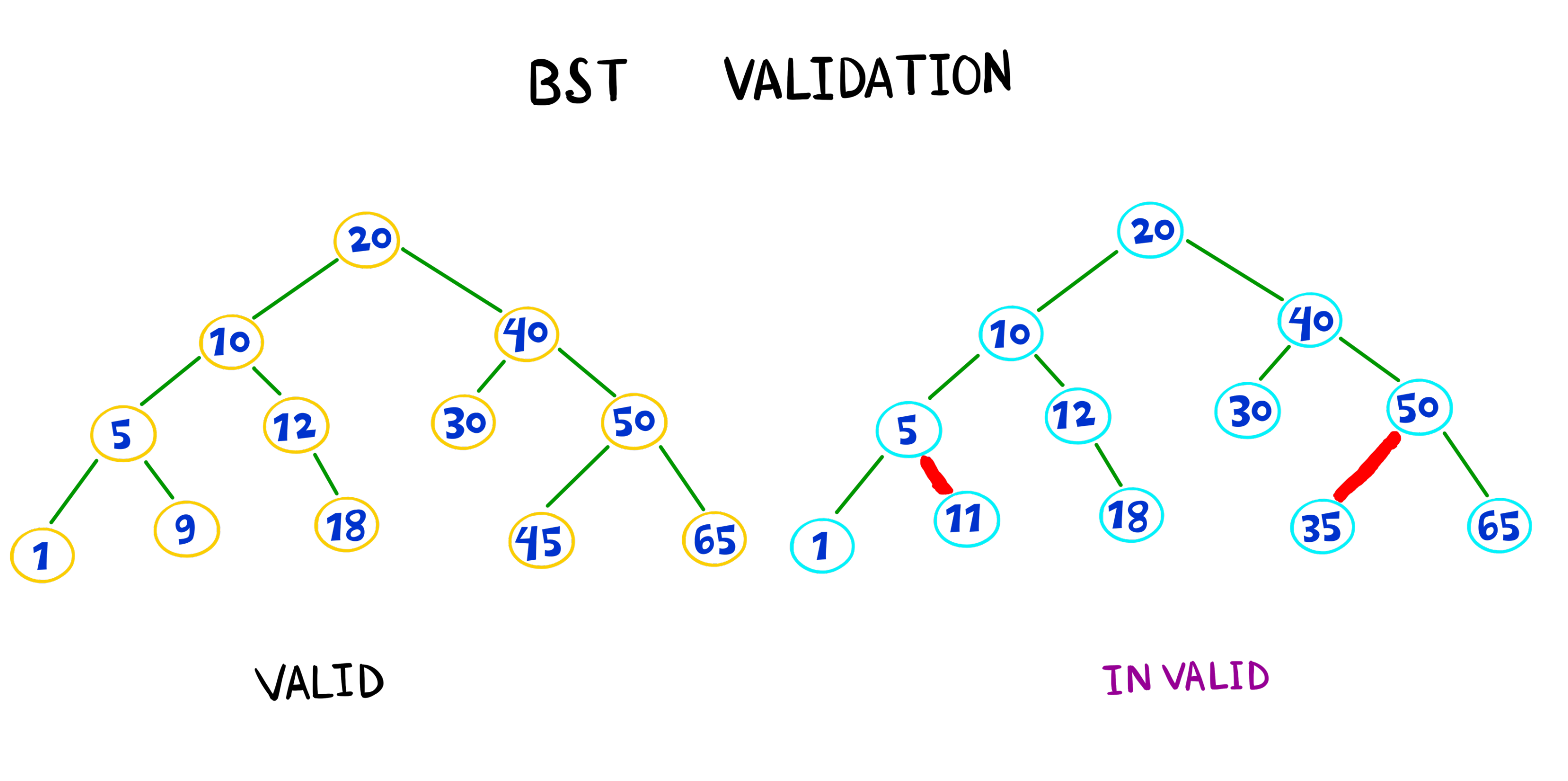
This is just a binary tree

Yes a binary search tree

all values to L (incl. L subtree) are < parent value

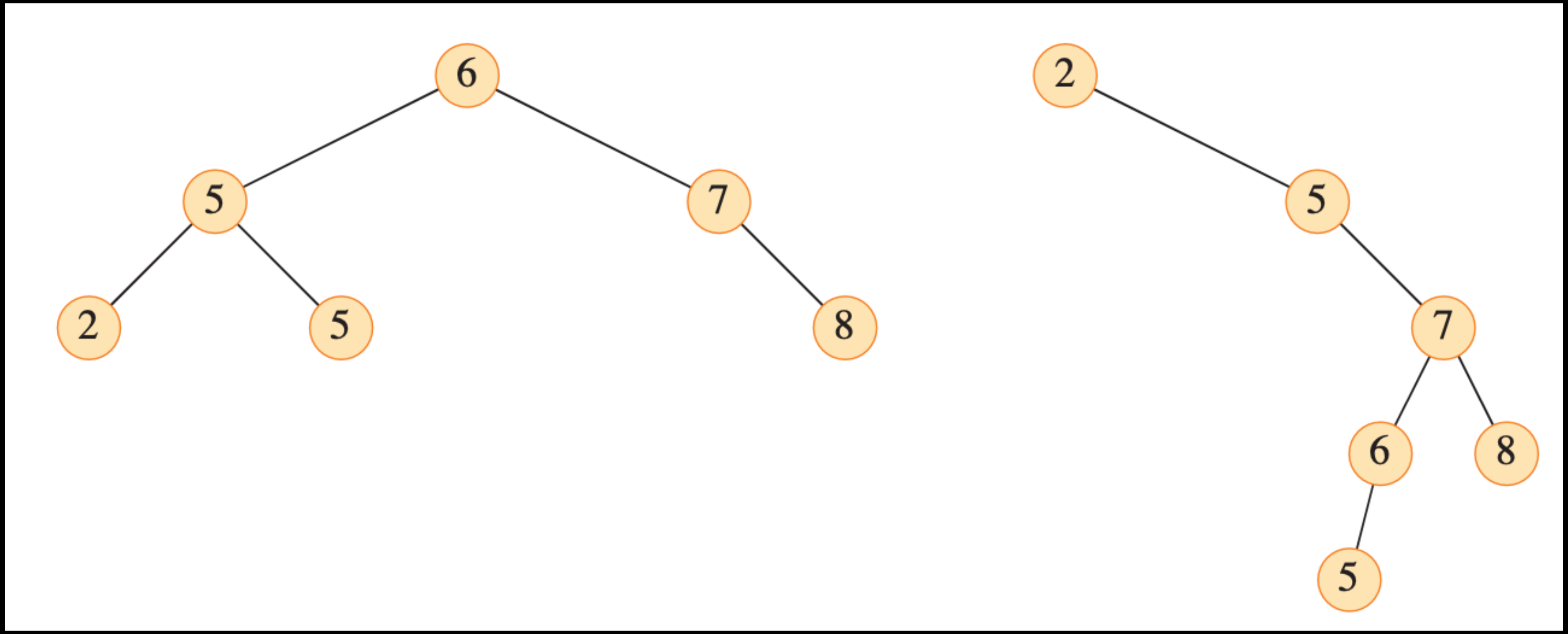
all values to R (incl. R subtree) are > parent value

Binary Search Trees



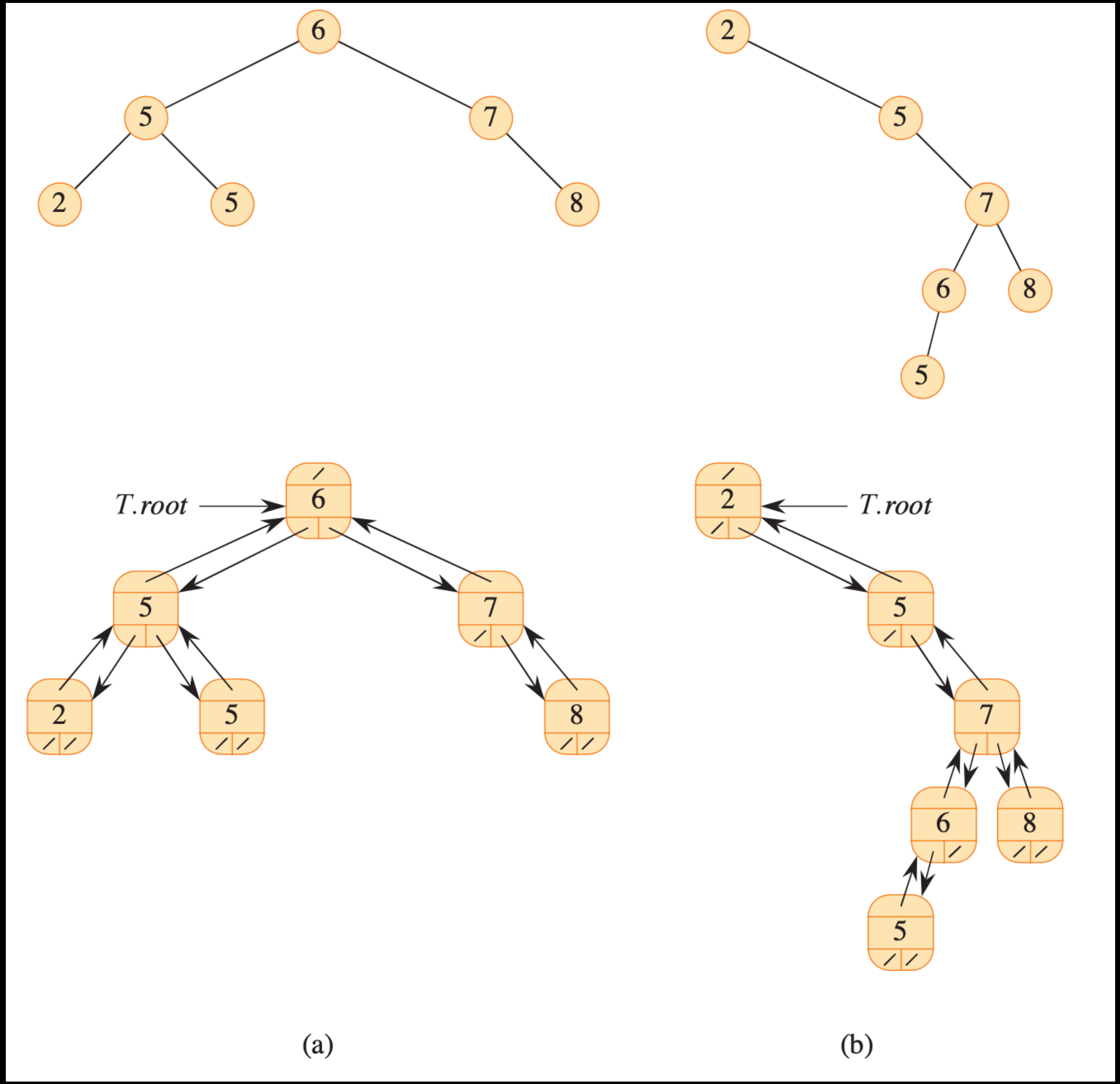
Binary Search Trees

- Basic operations on a BST take time proportional to the height of the tree.
  - Complete BST operations take $\Theta(\lg n)$ worst-case runtime.
  - Linear chain of nodes BST operations take $\Theta(n)$ worst-case runtime.



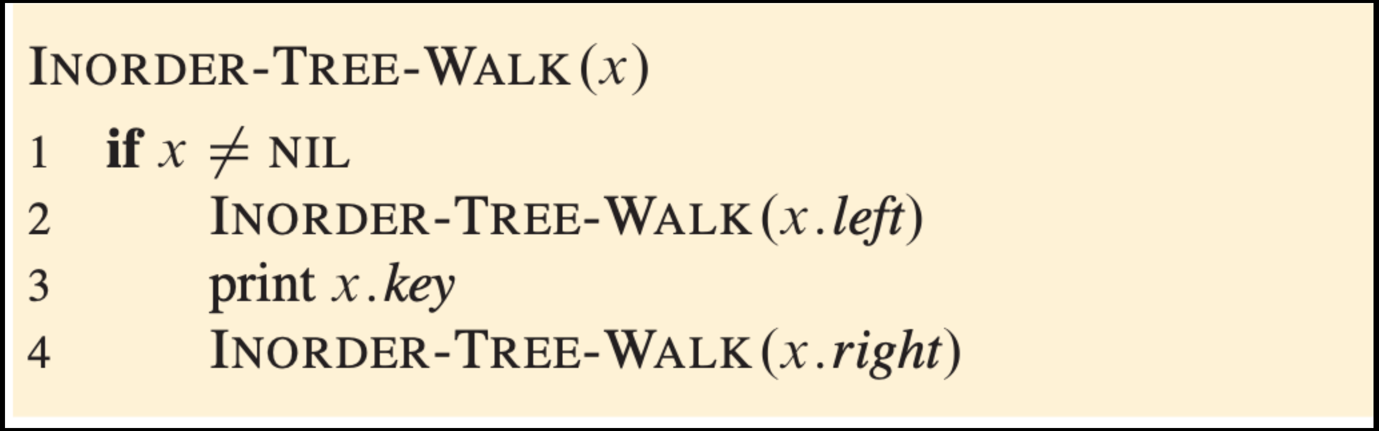
Binary Search Trees

- Represented with a linked list
- Each node contains:
  - A key
  - A pointer to the parent.
  - A pointer to the left child.
  - A pointer to the right child.
- Missing child or parent is NIL
  - $\mathrm{T}.root.parent = NIL$



Binary Search Trees

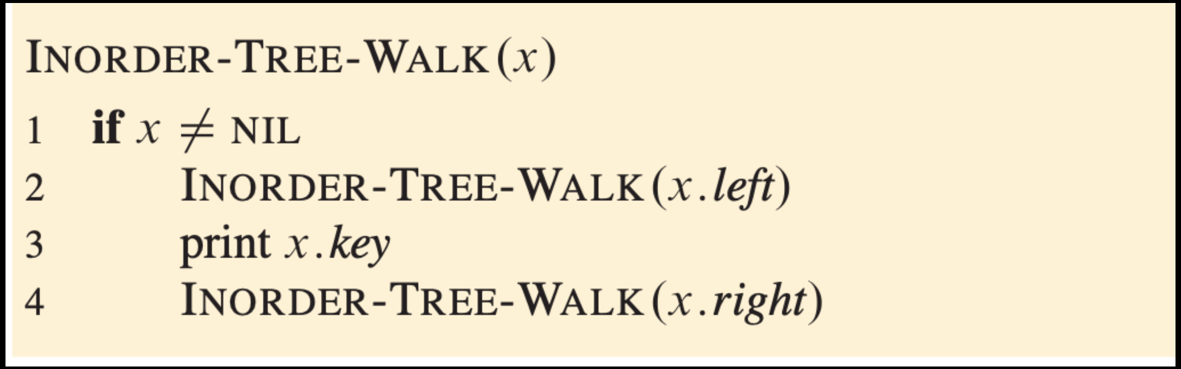
- To print out the sorted elements, use *inorder tree walk* algorithm.



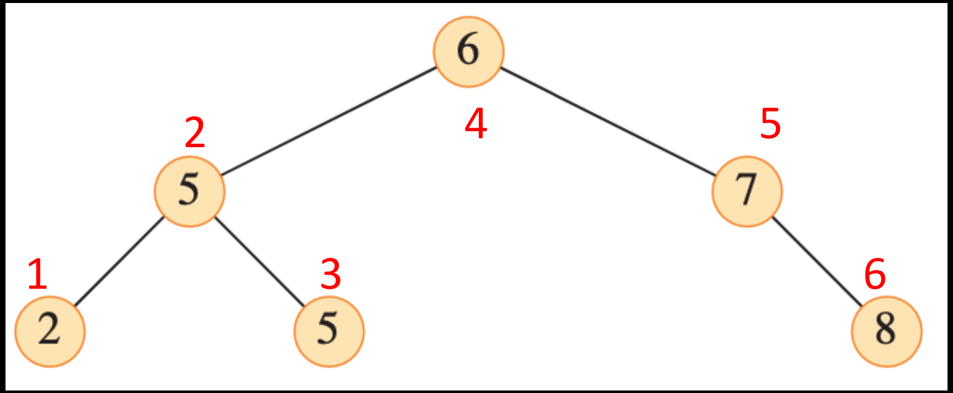
```
INORDER-TREE-WALK($x$)
if $x \neq$ NIL
    INORDER-TREE-WALK($x.left$)
    print $x.key$
    INORDER-TREE-WALK($x.right$)
```

- Time complexity: $\Theta(n)$

Binary Search Trees



```
INORDER-TREE-WALK(x)
if $x \neq$ NIL
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)
```



2, 5, 5, 6, 7, 8

Content

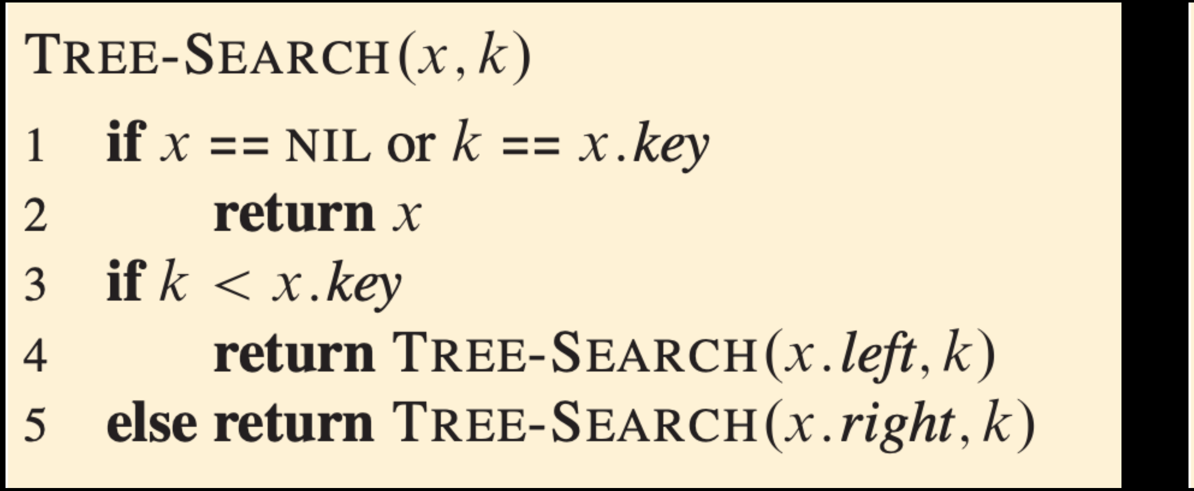
Content
Binary Search Tree
➡ Querying Binary Search Tree
Insertion and Deletion
Exercises

Querying Binary Search Trees

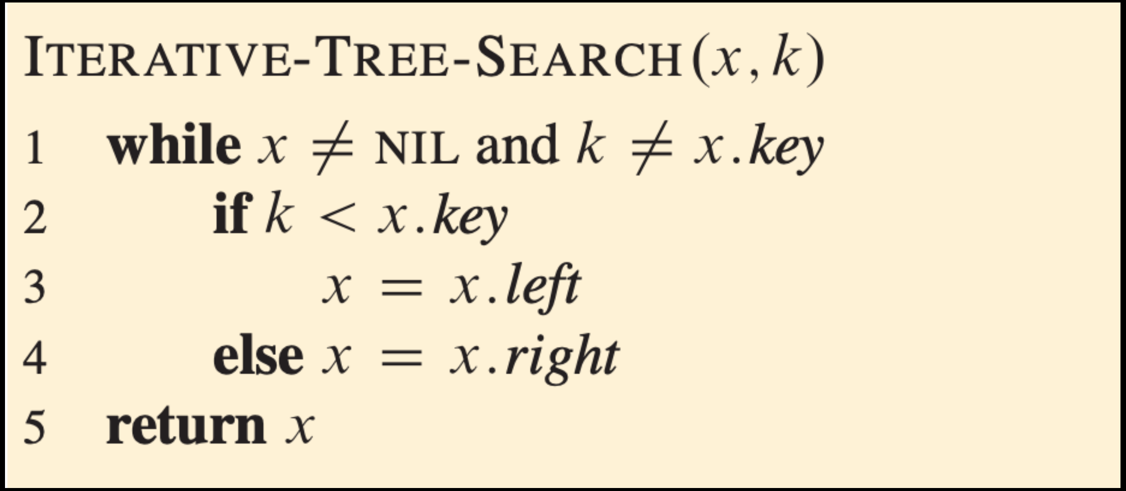
- A BST supports the following operations in time $O(\lg_2 n)$
  - Minimum and Maximum
  - Successor and Predecessor
  - Search

Querying Binary Search Trees

- Tree-Search procedure



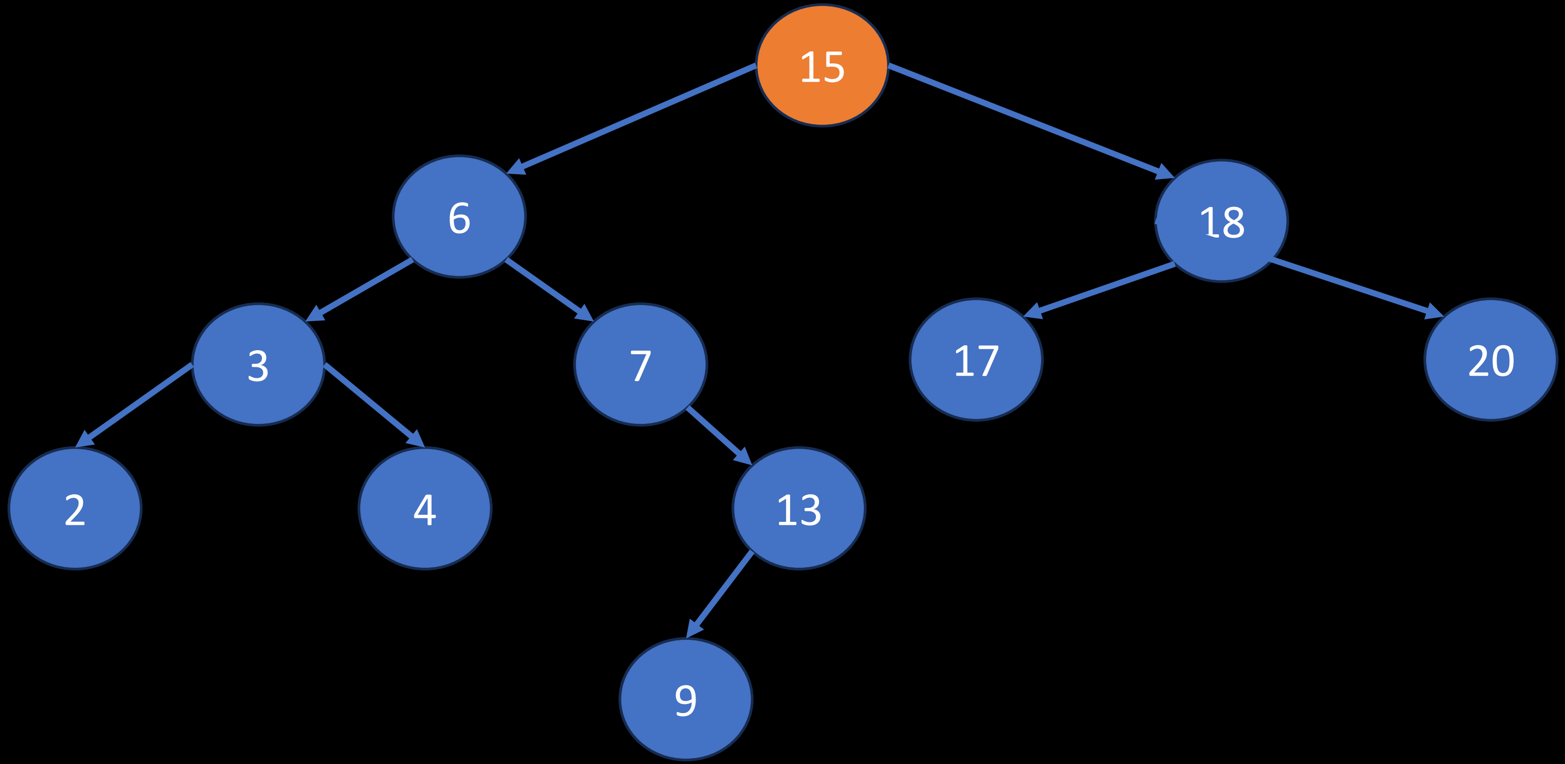
```
TREE-SEARCH$(x, k)$
if $x == \text{NIL}$ or $k == x.key$
    return $x$
if $k < x.key$
    return TREE-SEARCH$(x.left, k)$
else return TREE-SEARCH$(x.right, k)$
```



```
ITERATIVE-TREE-SEARCH$(x, k)$
while $x \neq \text{NIL}$ and $k \neq x.key$
    if $k < x.key$
        $x = x.left$
    else $x = x.right$
return $x$
```

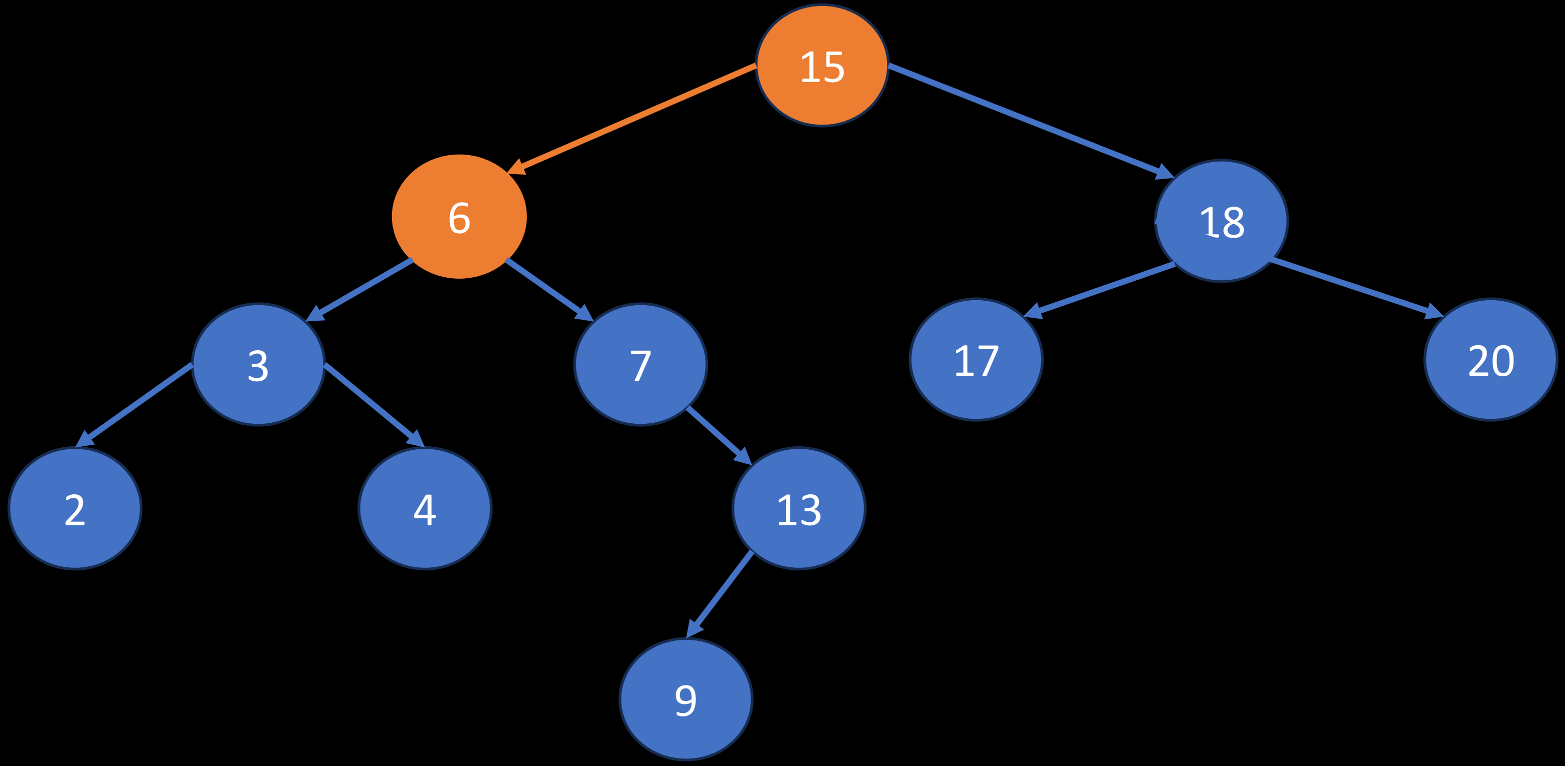
Querying Binary Search Trees

- Tree-Search procedure – search for 13



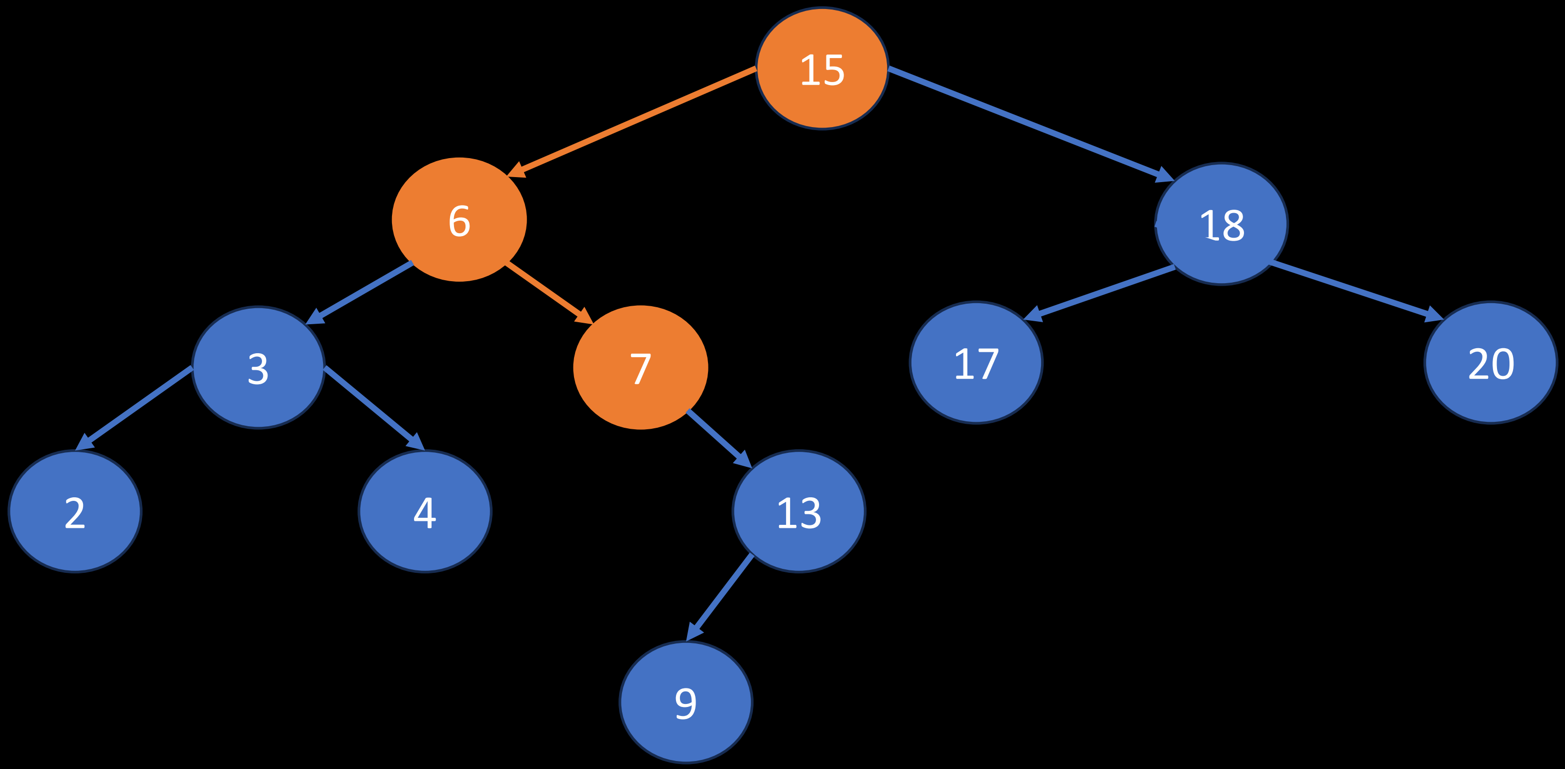
Querying Binary Search Trees

- Tree-Search procedure – search for 13



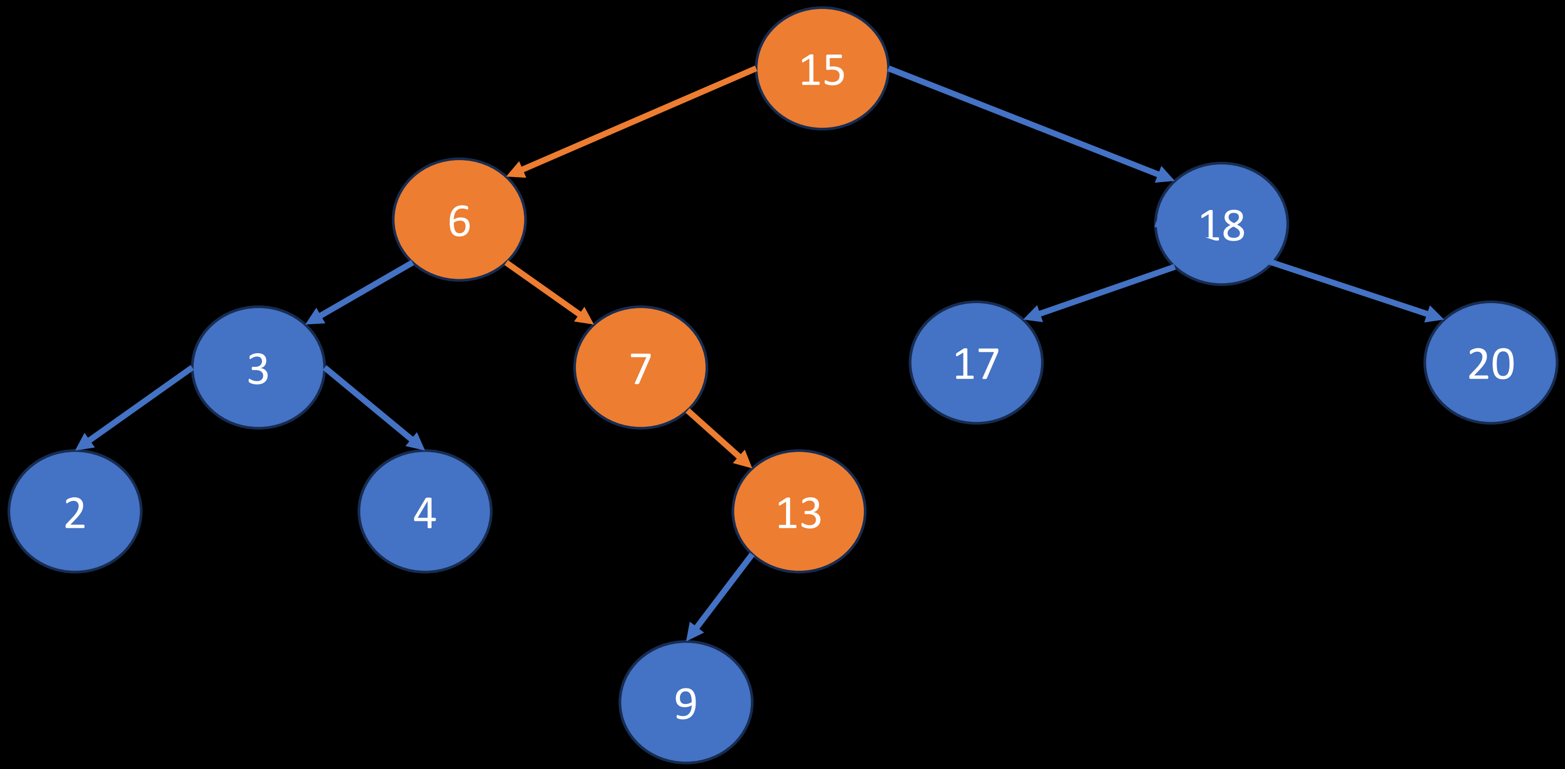
Querying Binary Search Trees

- Tree-Search procedure – search for 13



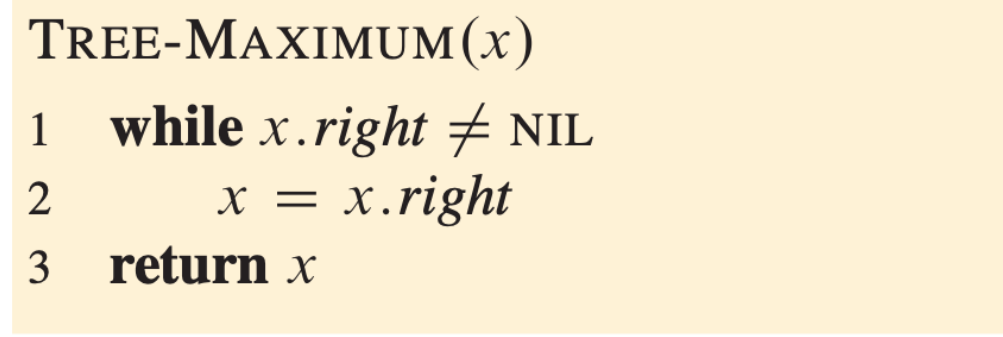
Querying Binary Search Trees

- Tree-Search procedure – search for 13

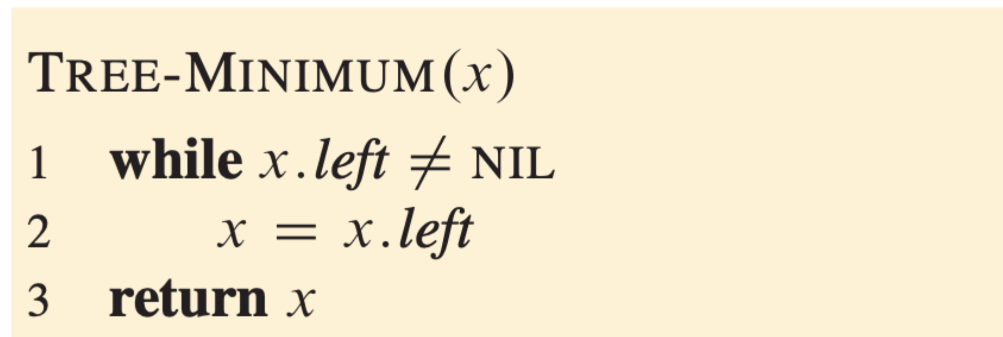


Querying Binary Search Trees

- Minimum and Maximum procedures.
    - To find the minimum, follow the left child pointers until you find NIL.
    - To find the maximum, follow the right child pointer until you find NIL.



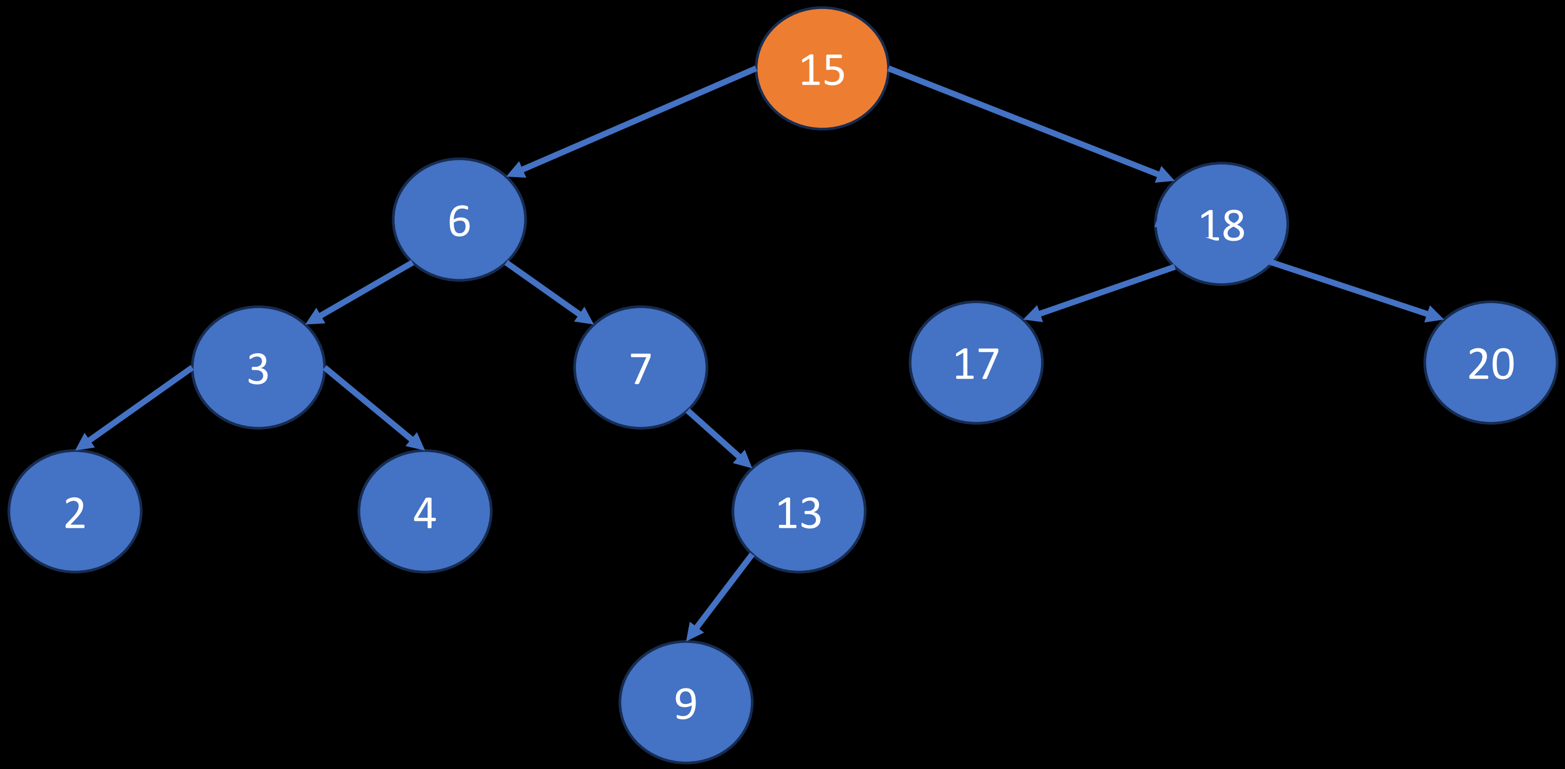
```
TREE-MAXIMUM($x$)
while $x.right \neq$ NIL
    $x = x.right$
return $x$
```



```
TREE-MINIMUM($x$)
while $x.left \neq$ NIL
    $x = x.left$
return $x$
```

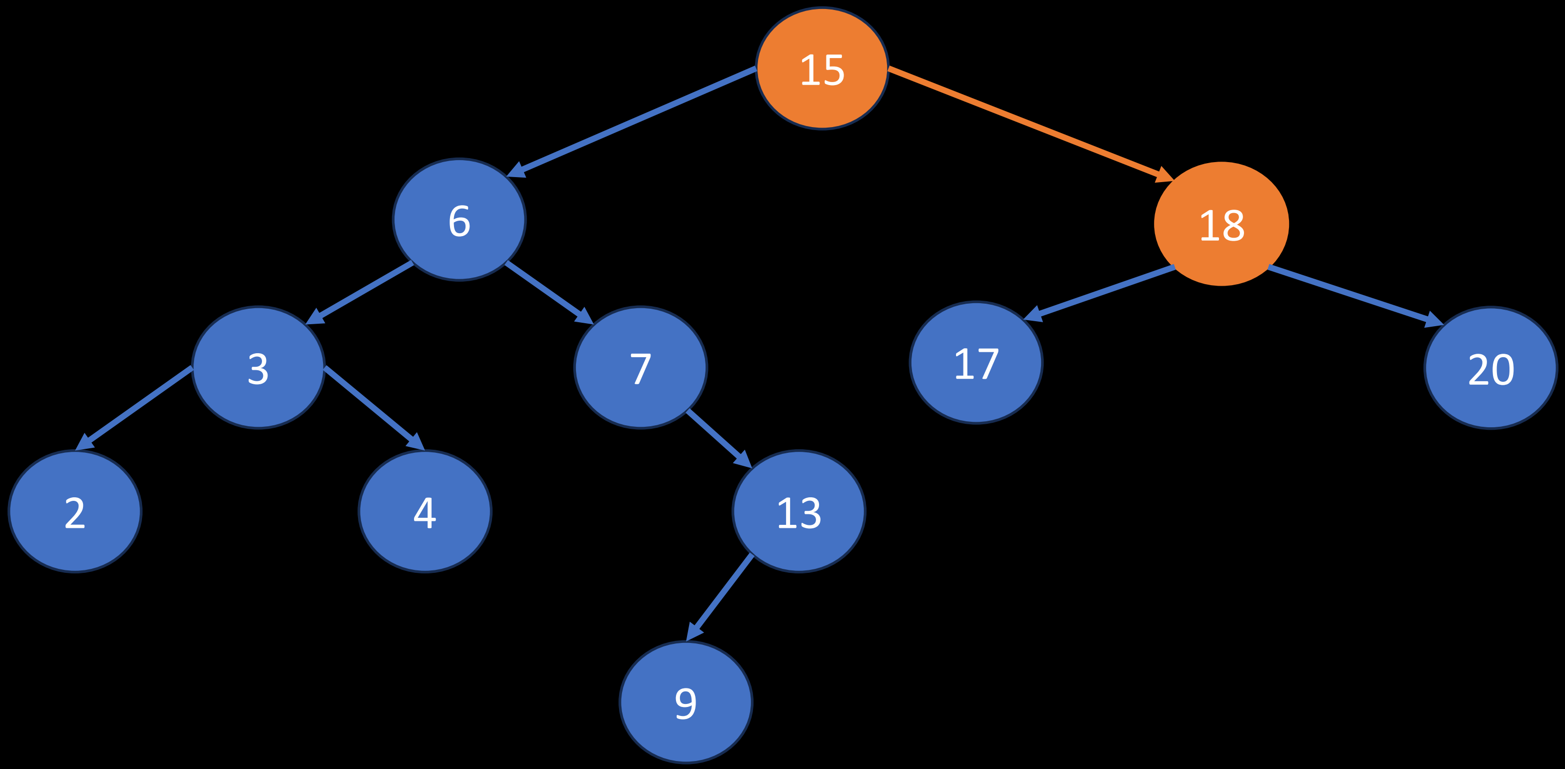
Querying Binary Search Trees

- Minimum and Maximum procedures – find maximum



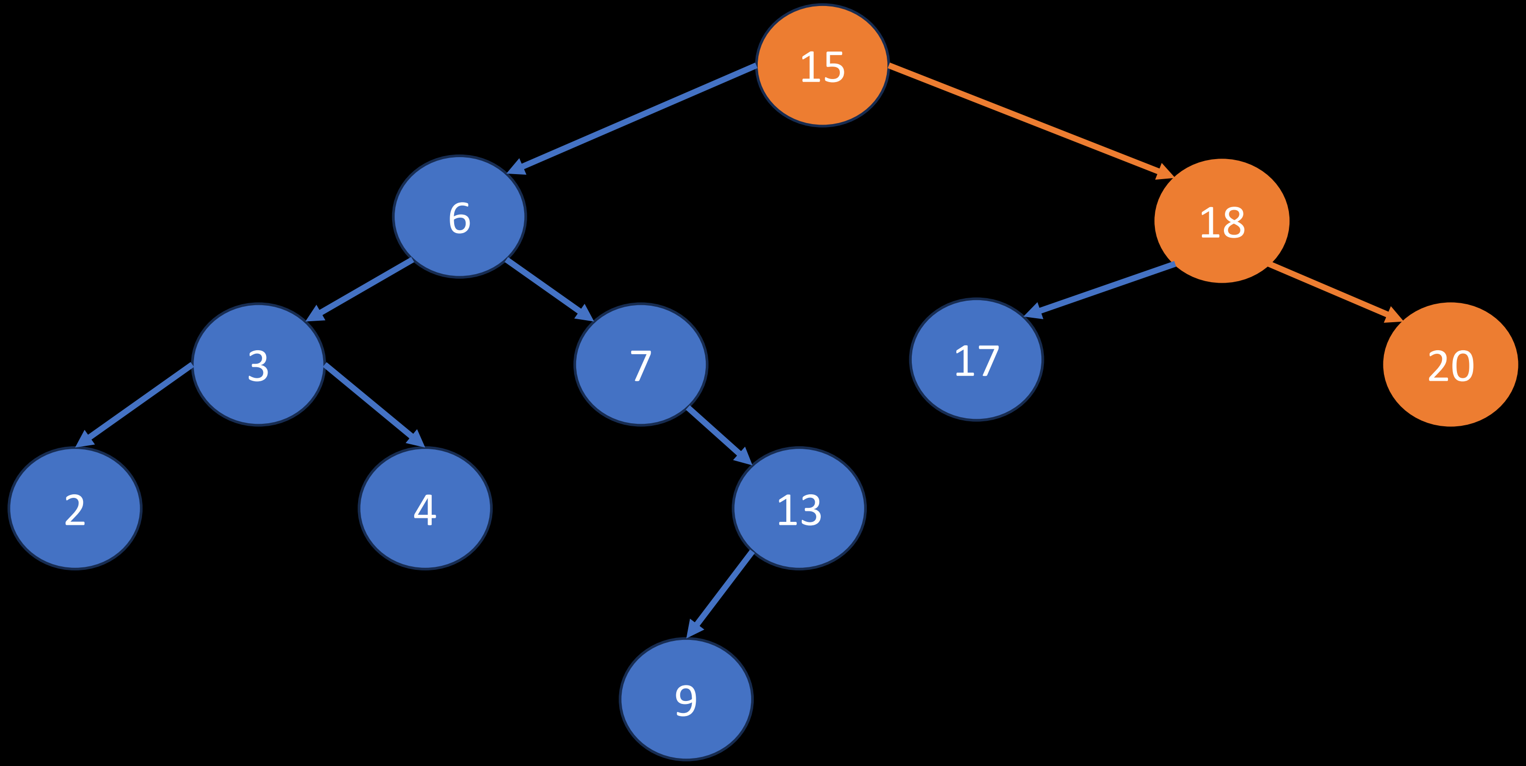
Querying Binary Search Trees

- Minimum and Maximum procedures – find maximum



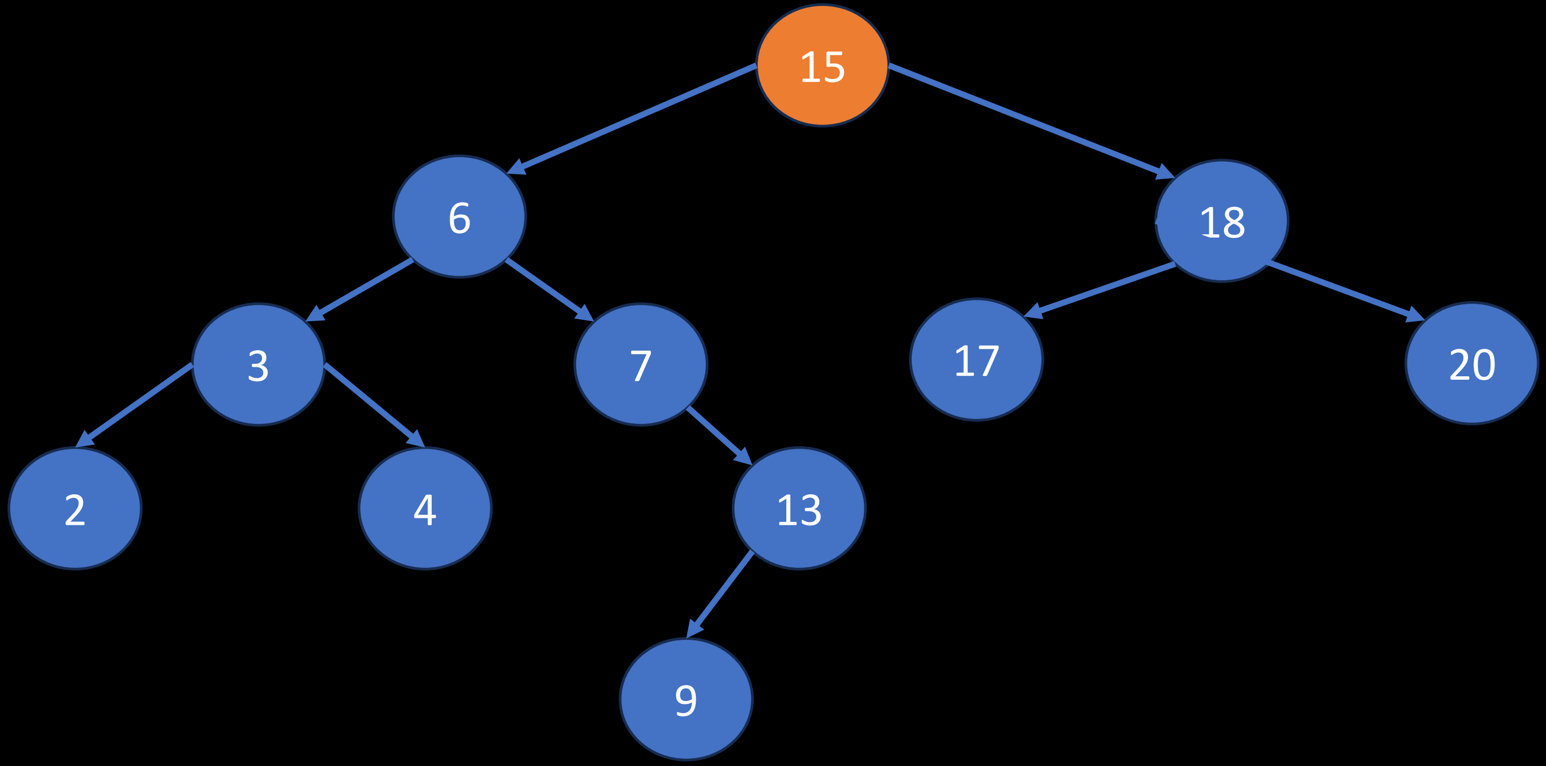
Querying Binary Search Trees

- Minimum and Maximum procedures – find maximum



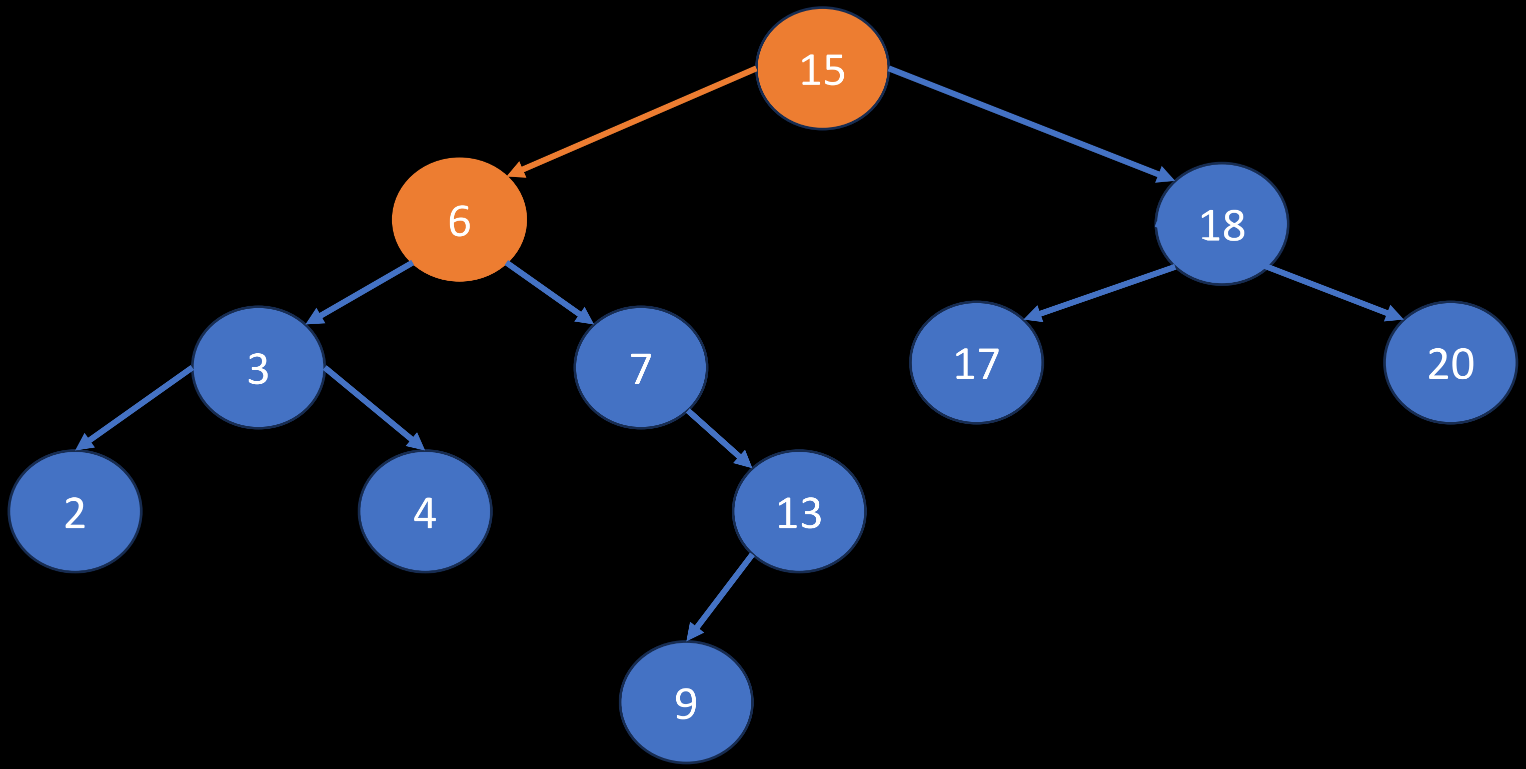
Querying Binary Search Trees

- Minimum and Maximum procedures – find minimum



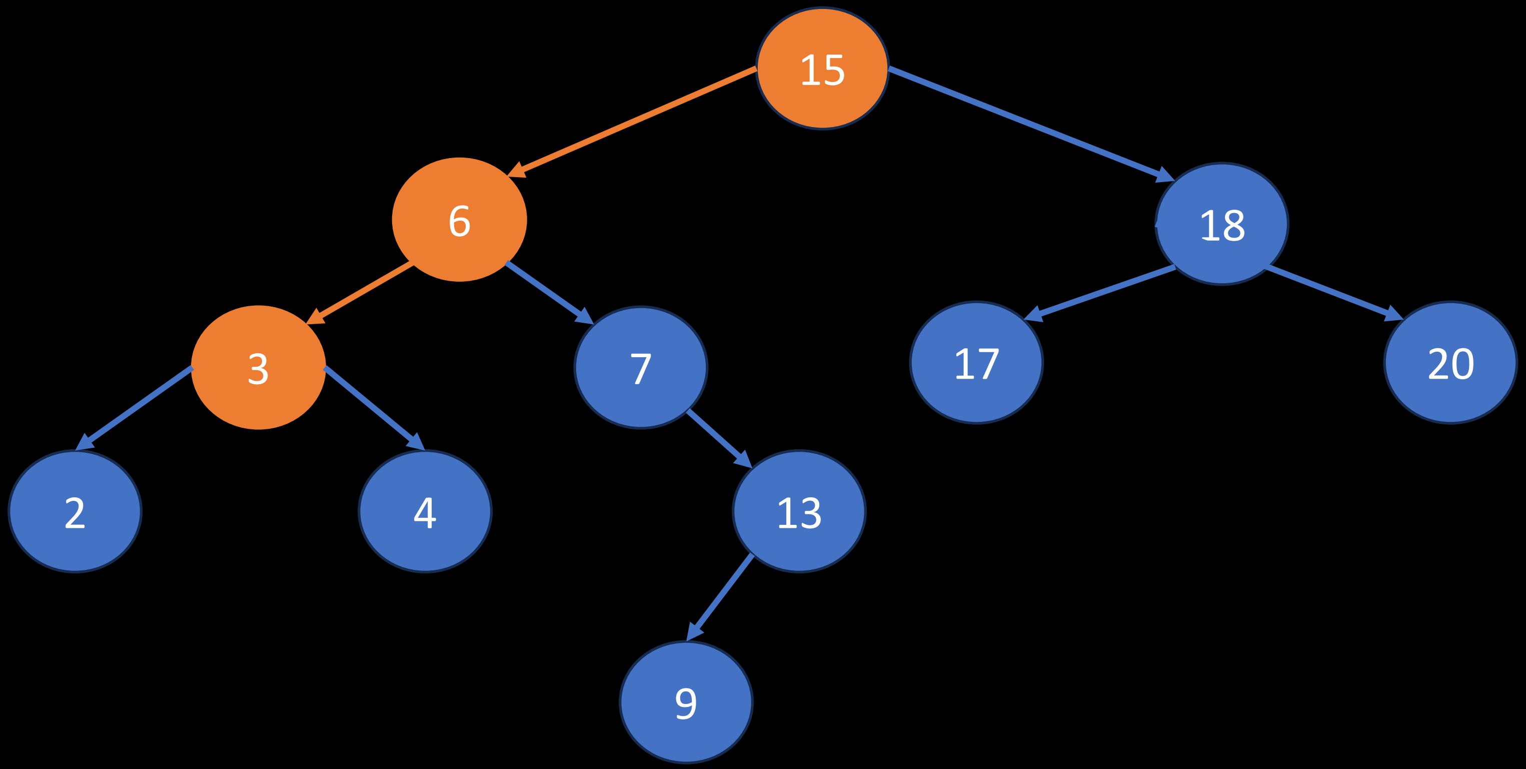
Querying Binary Search Trees

- Minimum and Maximum procedures – find minimum



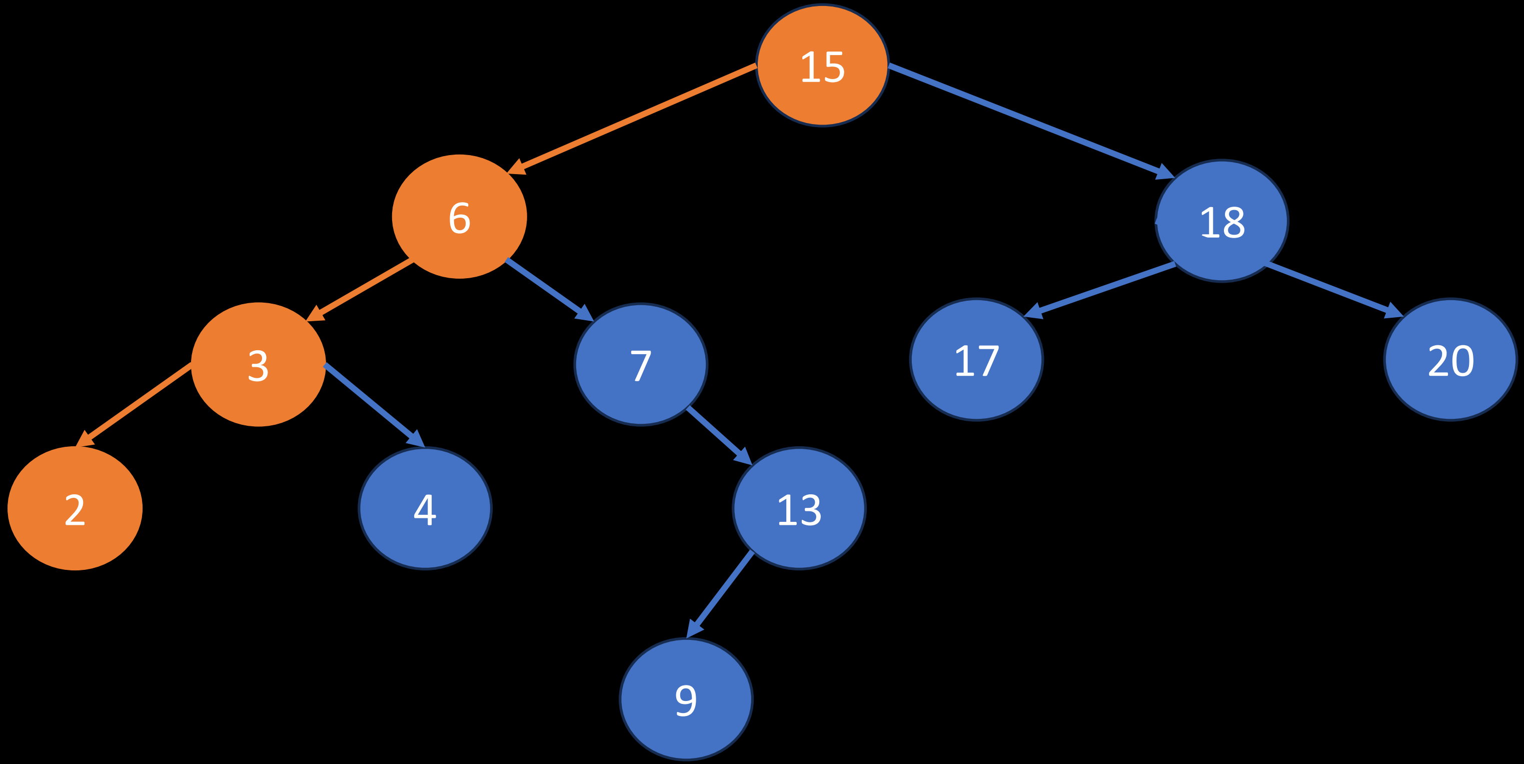
Querying Binary Search Trees

- Minimum and Maximum procedures – find minimum



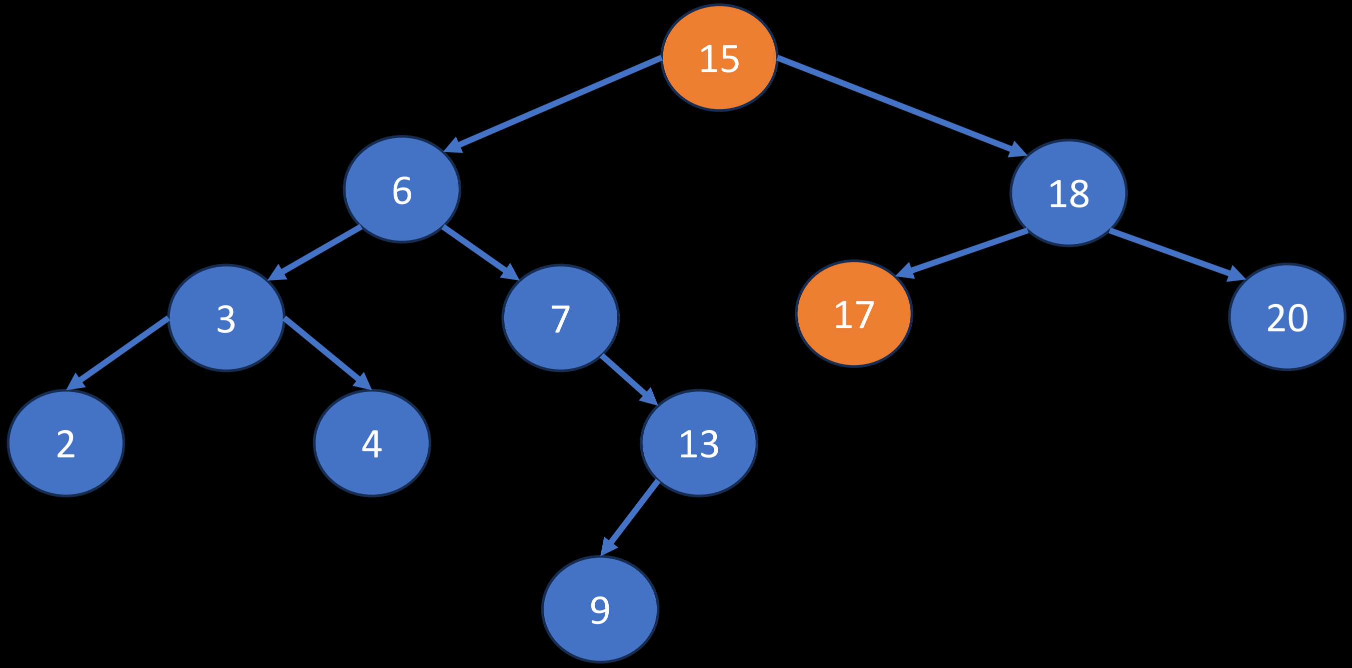
Querying Binary Search Trees

- Minimum and Maximum procedures – find minimum



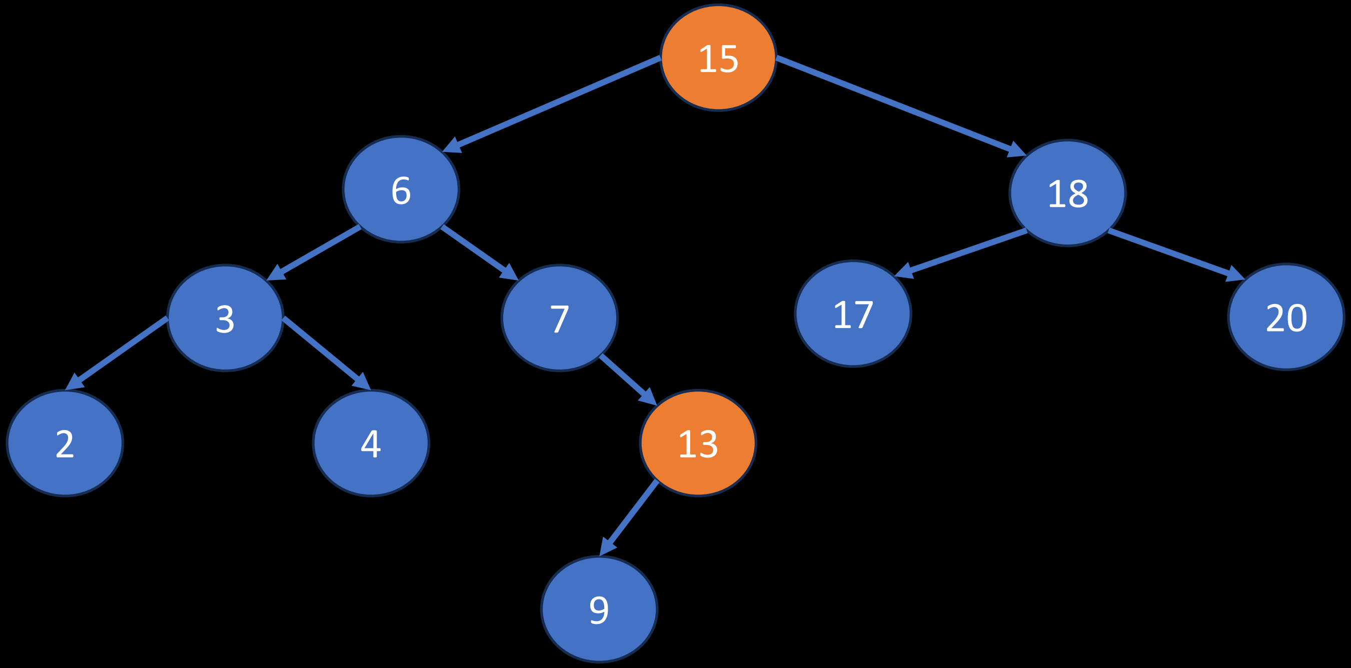
Querying Binary Search Trees

- **Successor** of a node is the next node visited in an inorder tree walk.
  - The **left most child** of right subtree or right child itself.
- Example: successor(15) is 17



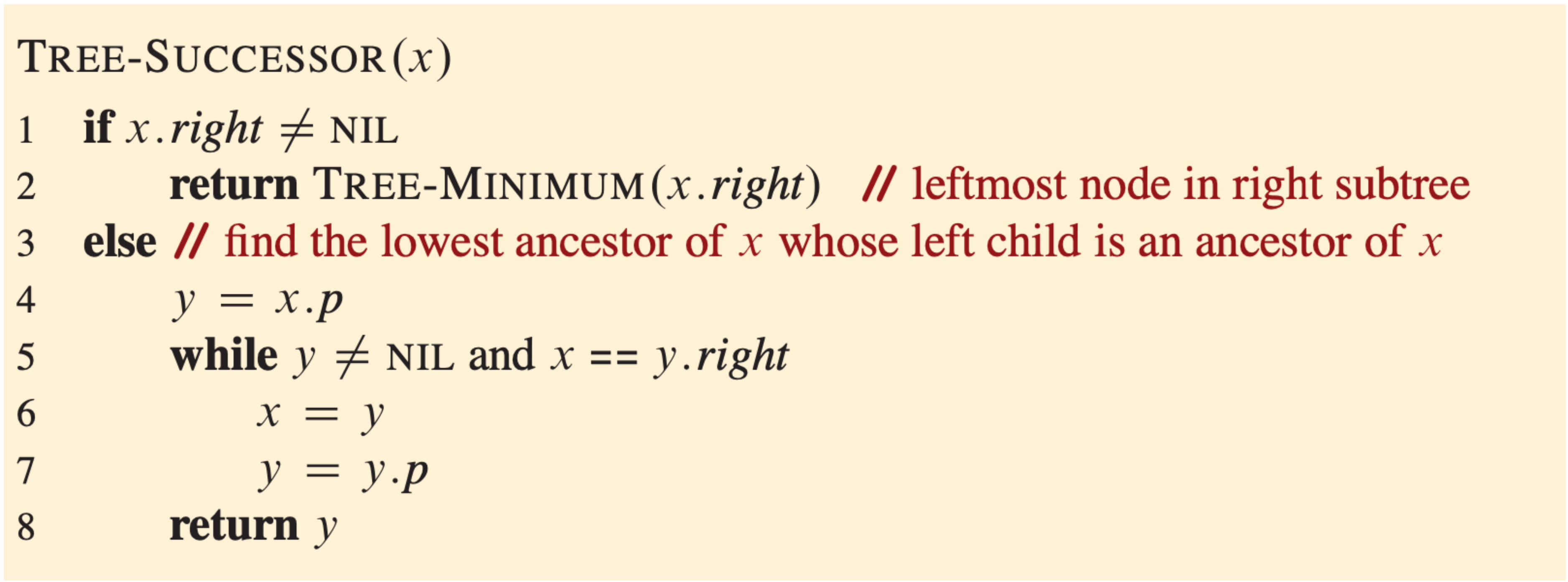
Querying Binary Search Trees

- **Predecessor** of a node is the preceding node visited in an inoreder tree walk.
  - The **right most child** of left subtree or left child itself.
- Example: predecessor(15) is 13



Querying Binary Search Trees

- Tree-Successor(x)



```
TREE-SUCCESSOR($x$)
if $x.right \neq$ NIL
    return TREE-MINIMUM($x.right$)   // leftmost node in right subtree
else // find the lowest ancestor of $x$ whose left child is an ancestor of $x$
    $y = x.p$
    while $y \neq$ NIL and $x == y.right$
        $x = y$
        $y = y.p$
    return $y$
```

Content

Content
Binary Search Tree
Querying Binary Search Tree
Insertion and Deletion
Exercises

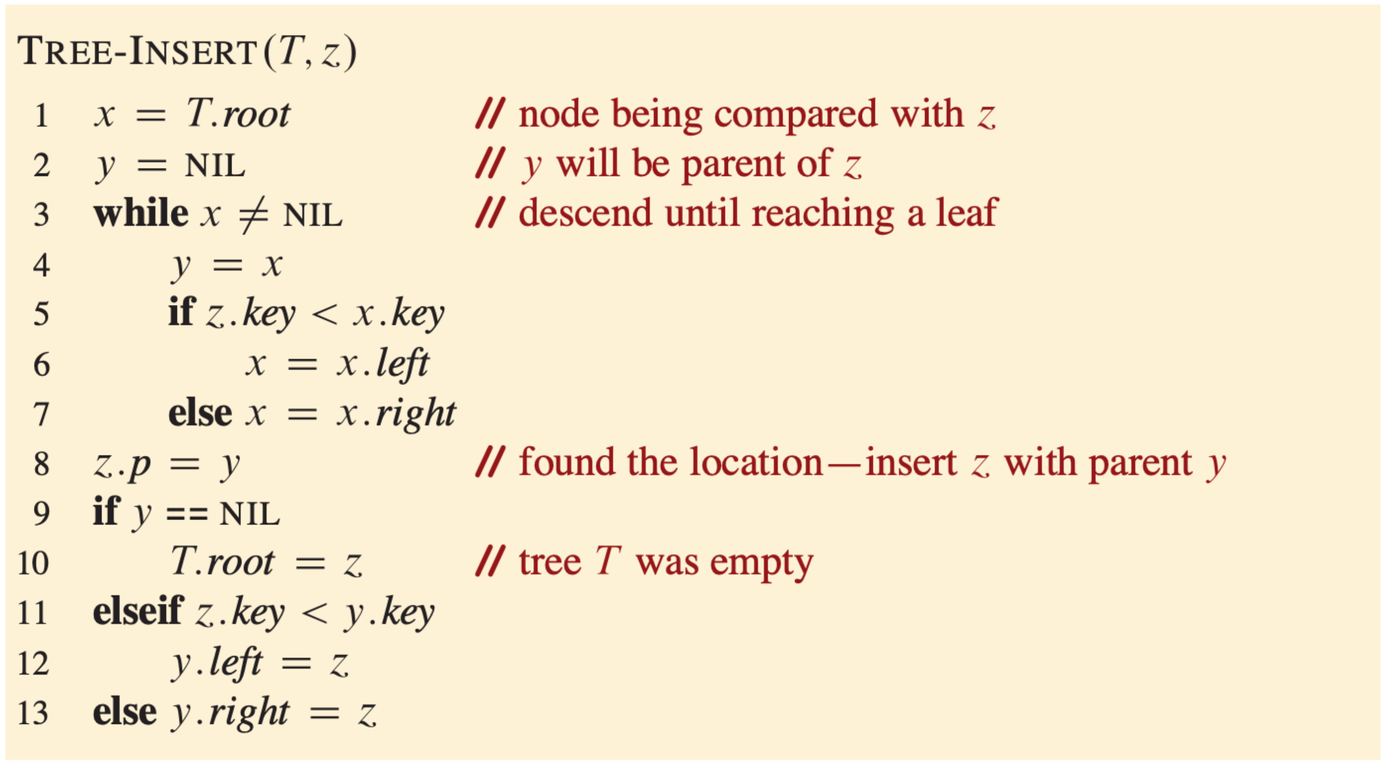
Insertion and Deletion

- When inserting or deleting elements, ensure that the BST property is held.
  - Every parent is greater than its left child.
  - Every parent is smaller than its right child.
  - Left nodes are smaller than their right siblings.

Insertion and Deletion

$Tree - Insert(T, z)$

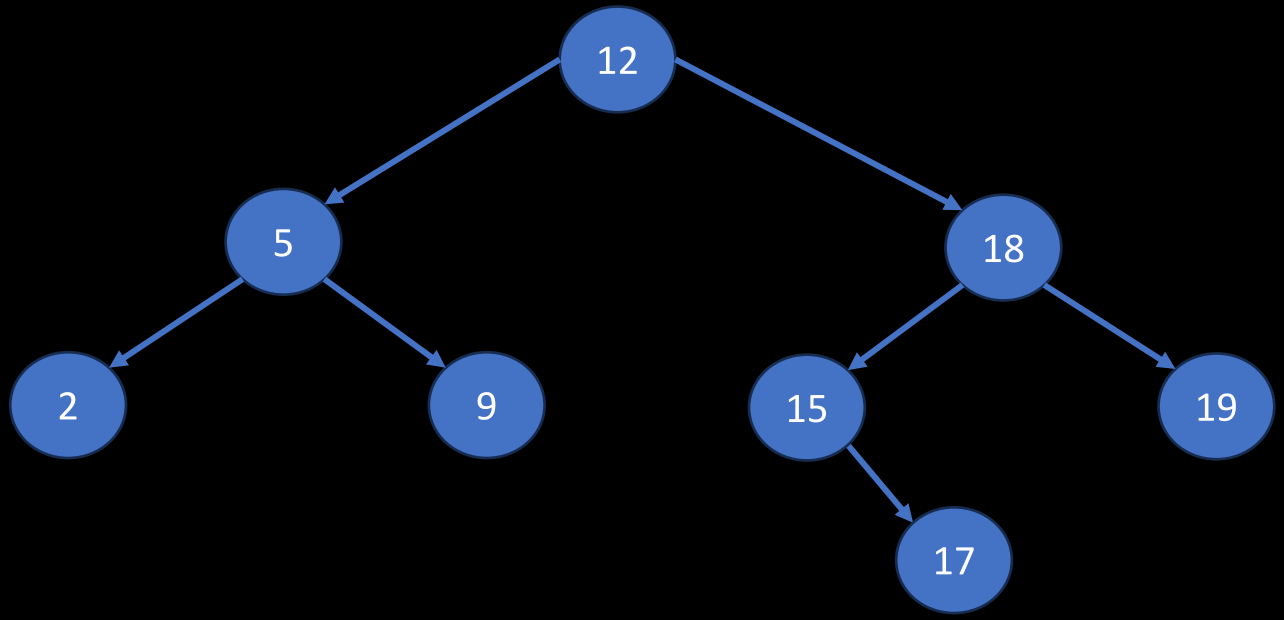
- Input a binary tree T and a node z, for which $z.left = NIL$ and $z.right = NIL$



```
TREE-INSERT(T, z)
$x = T.root$                   // node being compared with $z$
$y =$ NIL                      // $y$ will be parent of $z$
while $x \neq$ NIL             // descend until reaching a leaf
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$                      // found the location—insert $z$ with parent $y$
if $y ==$ NIL
    $T.root = z$               // tree $T$ was empty
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
```

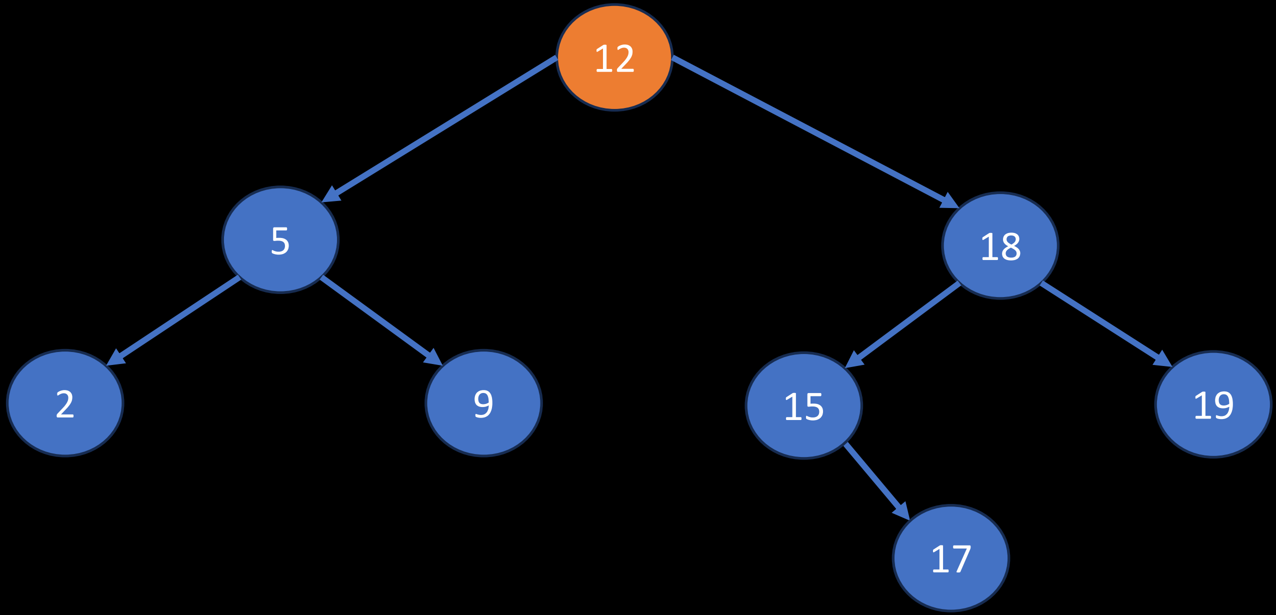
Insertion and Deletion

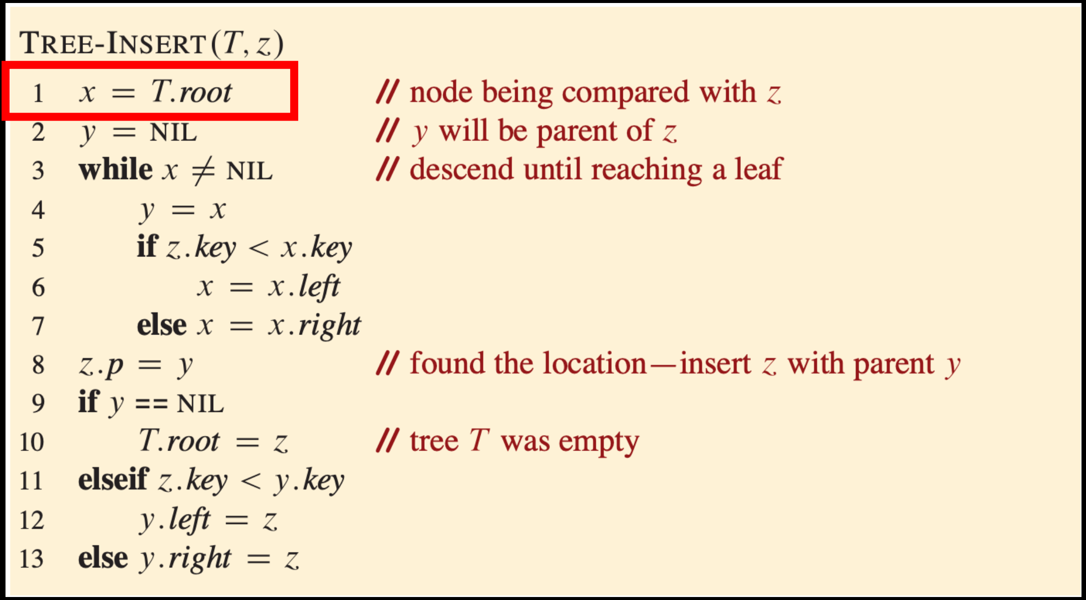
- Example: insert 13.



Insertion and Deletion

- Example: insert 13.

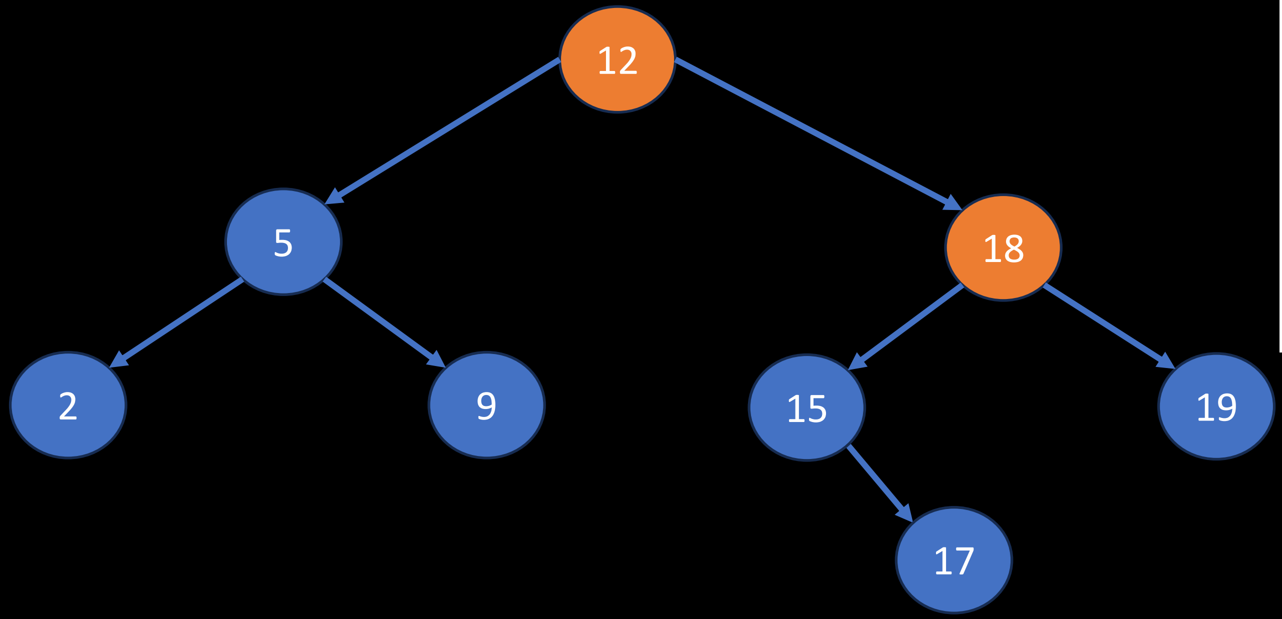


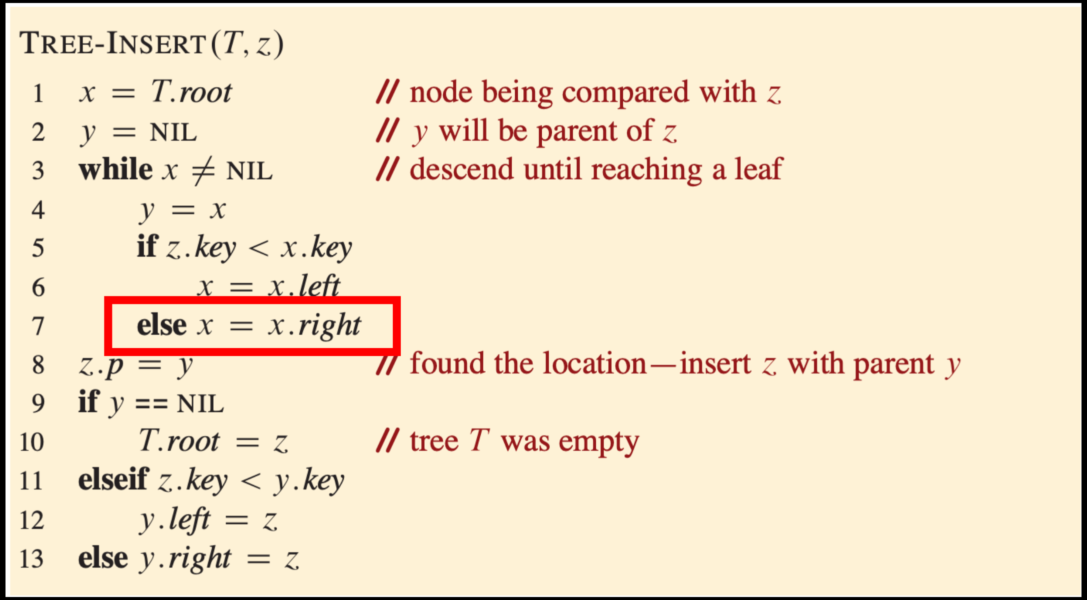


```
TREE-INSERT(T, z)
x = T.root                          // node being compared with z
y = NIL                             // y will be parent of z
while x ≠ NIL                       // descend until reaching a leaf
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                             // found the location—insert z with parent y
if y == NIL
    T.root = z                      // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Insertion and Deletion

- Example: insert 13.

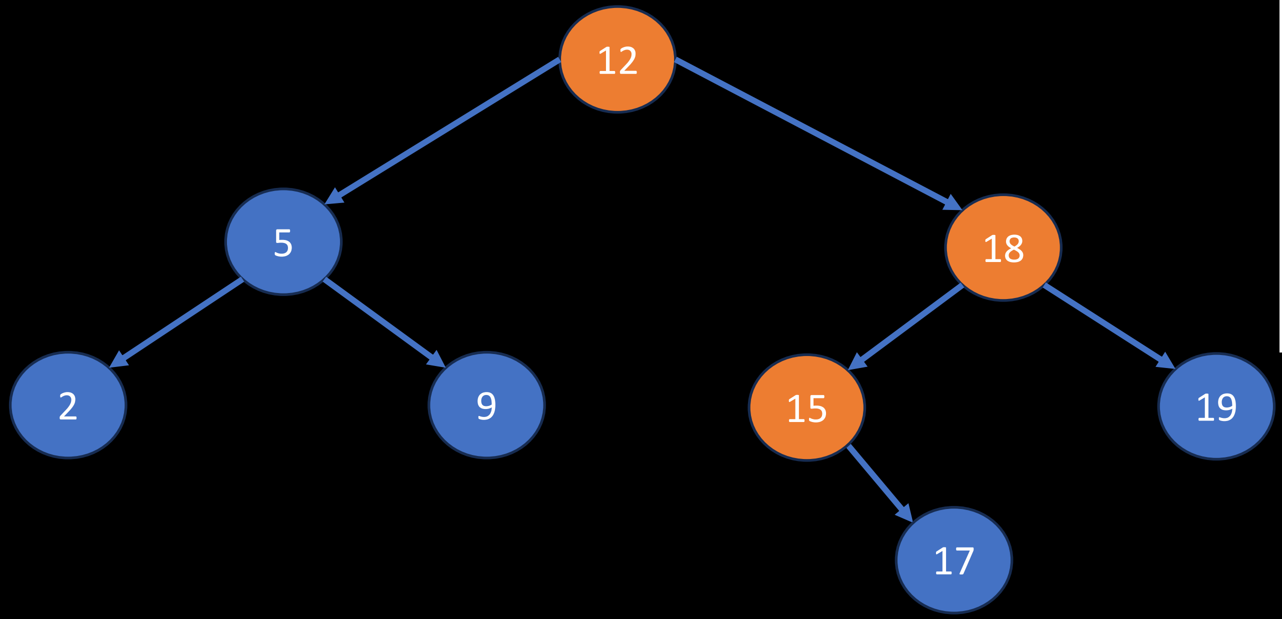


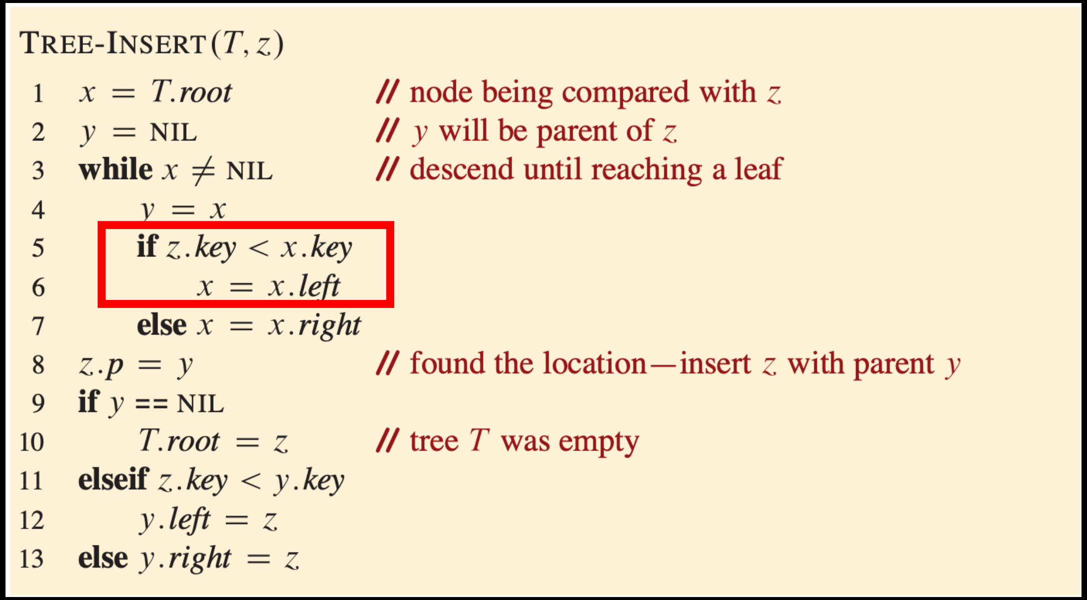


```
TREE-INSERT(T, z)
x = T.root                  // node being compared with z
y = NIL                     // y will be parent of z
while x ≠ NIL               // descend until reaching a leaf
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                     // found the location—insert z with parent y
if y == NIL
    T.root = z              // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Insertion and Deletion

- Example: insert 13.

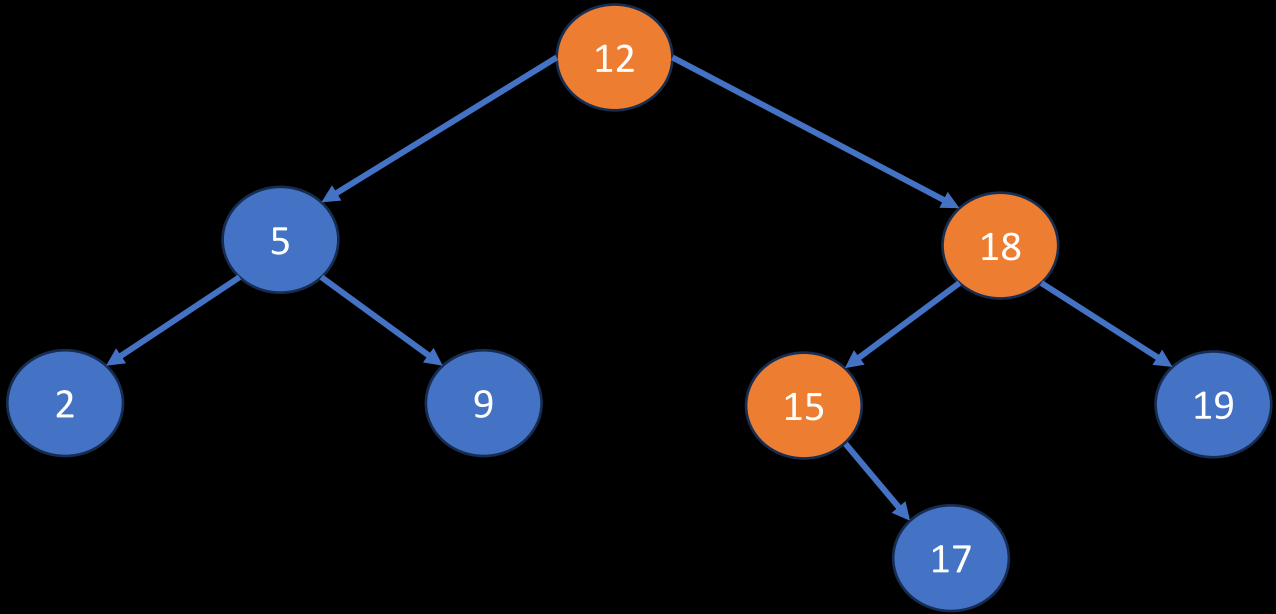


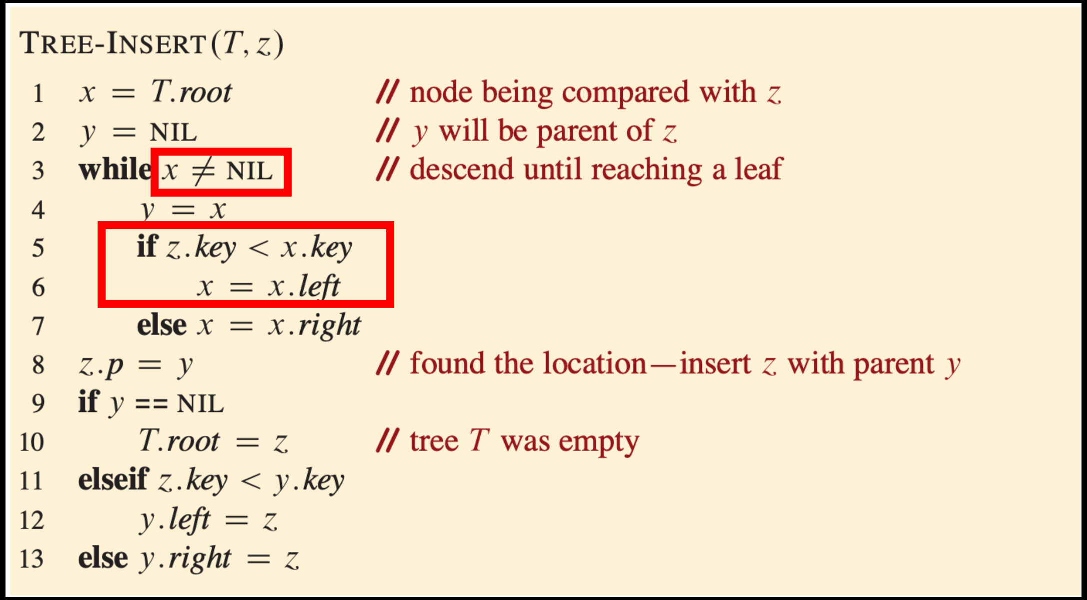


```
TREE-INSERT$(T, z)$
$x = T.root$                    // node being compared with $z$
$y =$ NIL                       // $y$ will be parent of $z$
while $x \neq$ NIL              // descend until reaching a leaf
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$                       // found the location—insert $z$ with parent $y$
if $y ==$ NIL
    $T.root = z$                // tree $T$ was empty
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
```

Insertion and Deletion

- Example: insert 13.

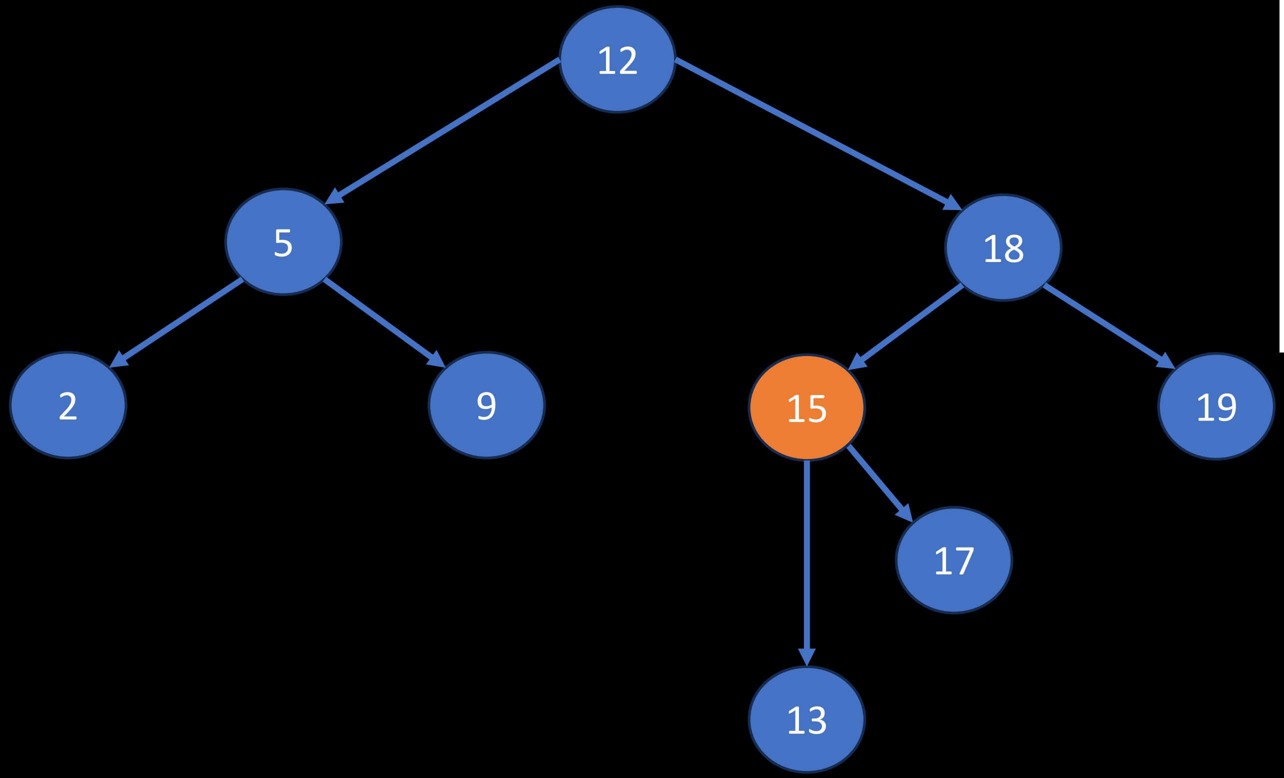


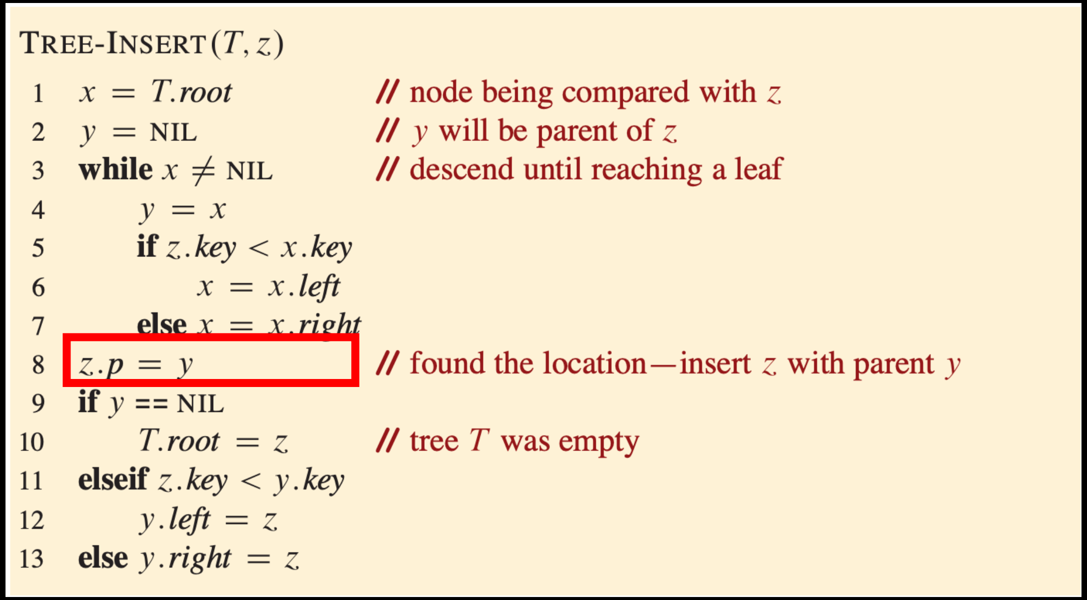


```
TREE-INSERT(T, z)
x = T.root                        // node being compared with z
y = NIL                           // y will be parent of z
while x ≠ NIL                     // descend until reaching a leaf
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                           // found the location—insert z with parent y
if y == NIL
    T.root = z                    // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Insertion and Deletion

- Example: insert 13.

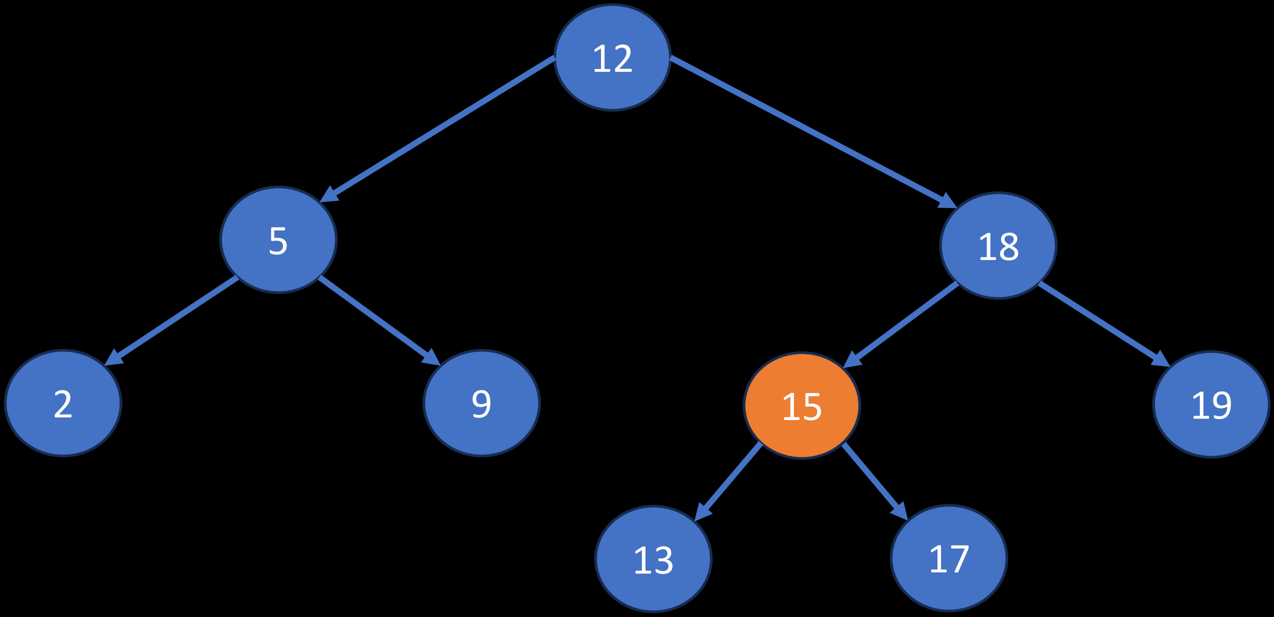


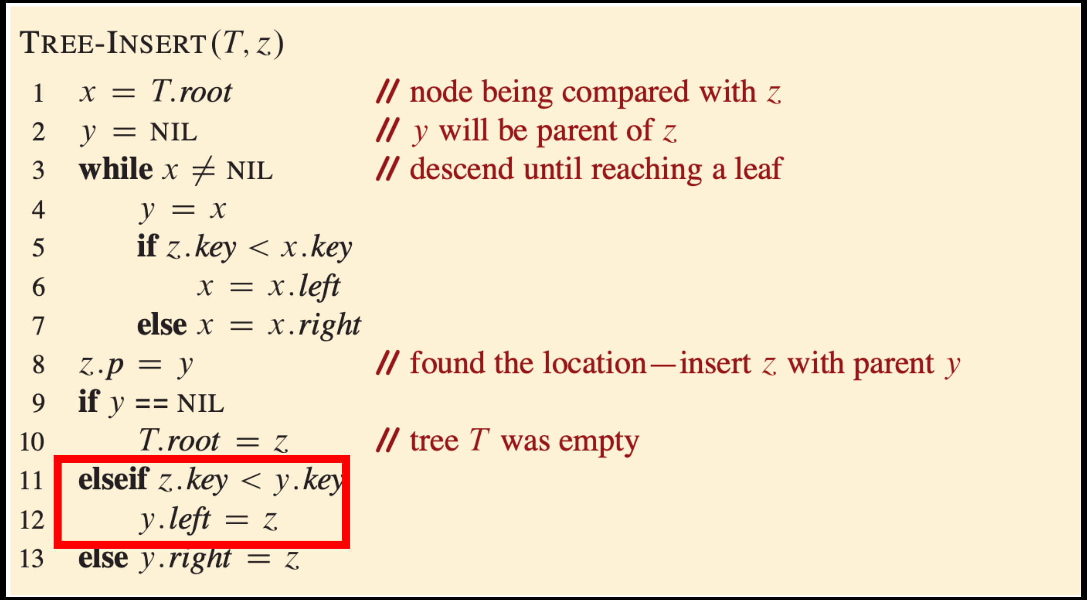


```
TREE-INSERT$(T, z)$
$x = T.root$                   // node being compared with $z$
$y =$ NIL                      // $y$ will be parent of $z$
while $x \neq$ NIL             // descend until reaching a leaf
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$                      // found the location—insert $z$ with parent $y$
if $y ==$ NIL
    $T.root = z$               // tree $T$ was empty
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
```

Insertion and Deletion

- Example: insert 13.

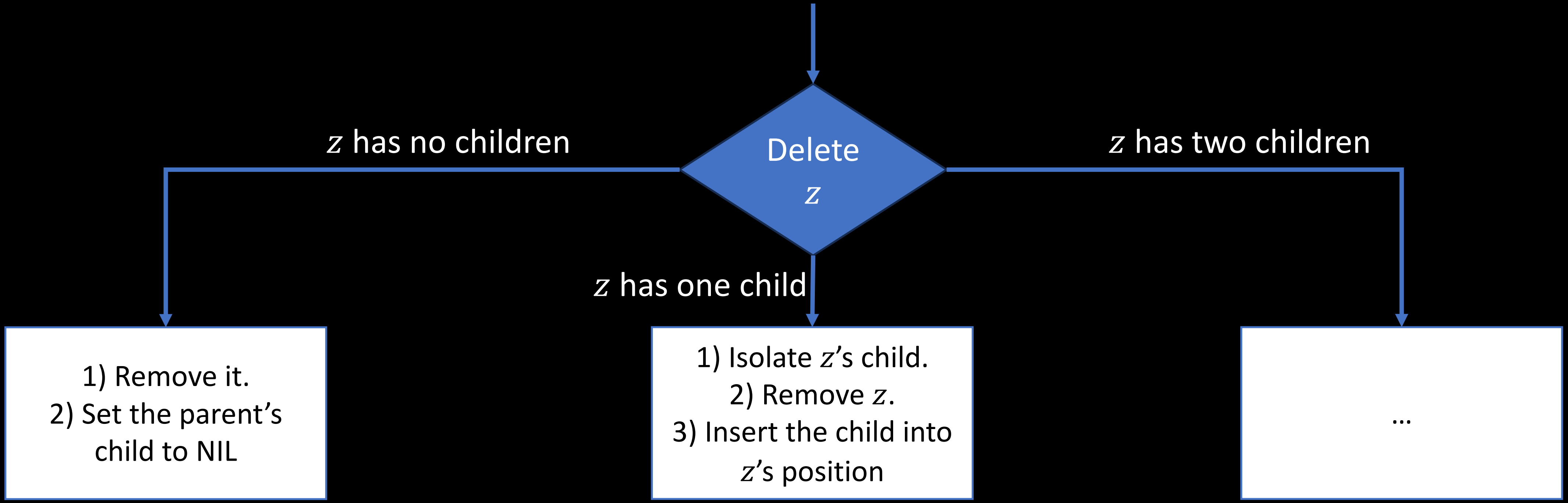




```
TREE-INSERT(T, z)
x = T.root                  // node being compared with z
y = NIL                     // y will be parent of z
while x ≠ NIL               // descend until reaching a leaf
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                     // found the location—insert z with parent y
if y == NIL
    T.root = z              // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Insertion and Deletion

- Deleting a node from the BST has three cases:



1) Remove it.
2) Set the parent's child to NIL

1) Isolate z's child.
2) Remove z.
3) Insert the child into z's position

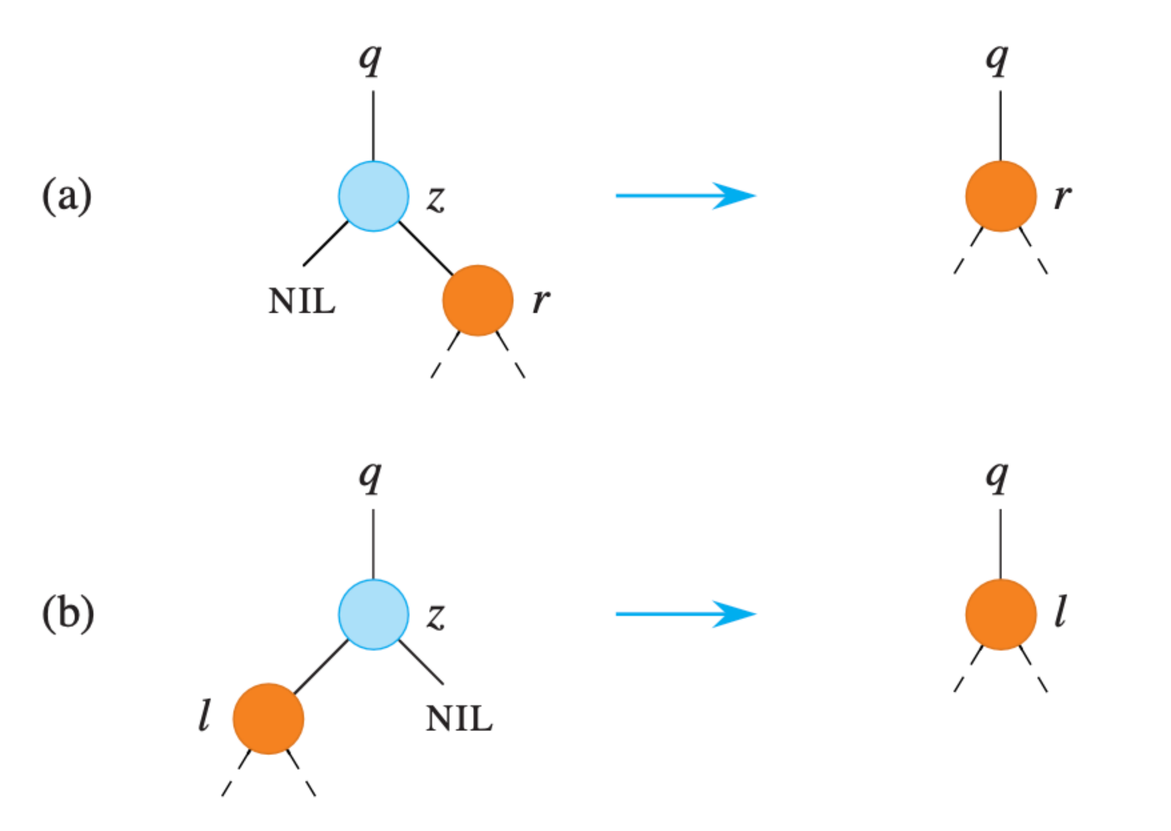
…

Insertion and Deletion

- Deleting a node z with two children:
  1. Find the successor of z, call it y
     1. y must belong to z's right subtree
  2. Move y to take the position of z
  3. The right subtree of z becomes the new right subtree of y
  4. The left subtree of z becomes the new left subtree of y

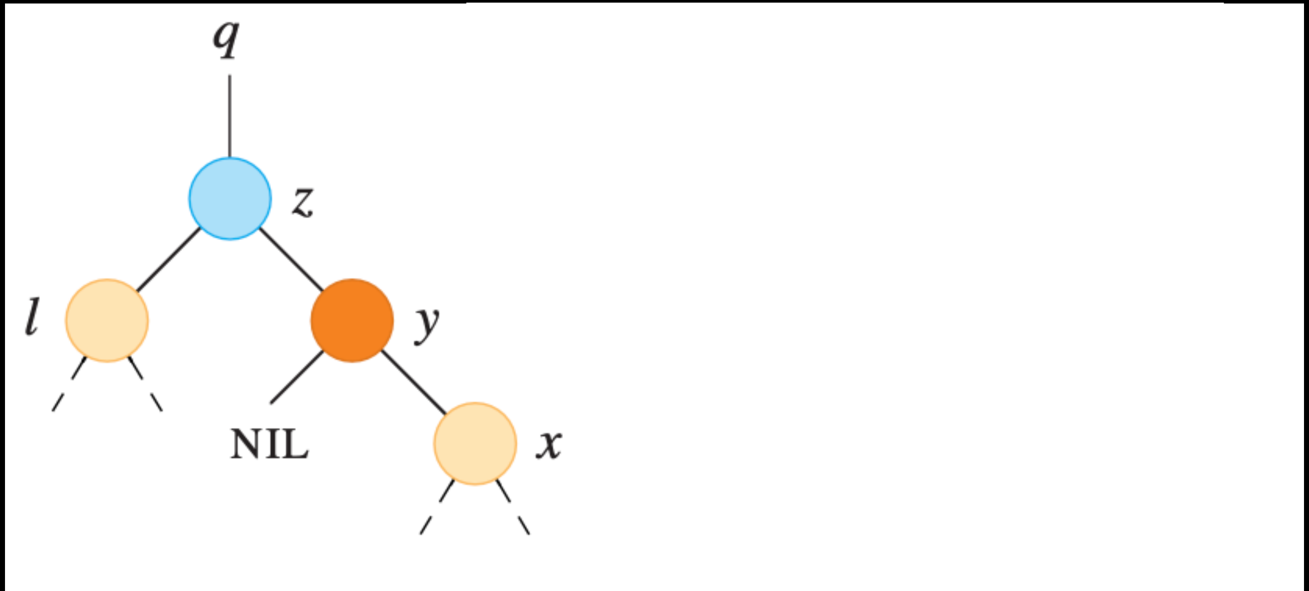
Insertion and Deletion

- Deleting a node with one child



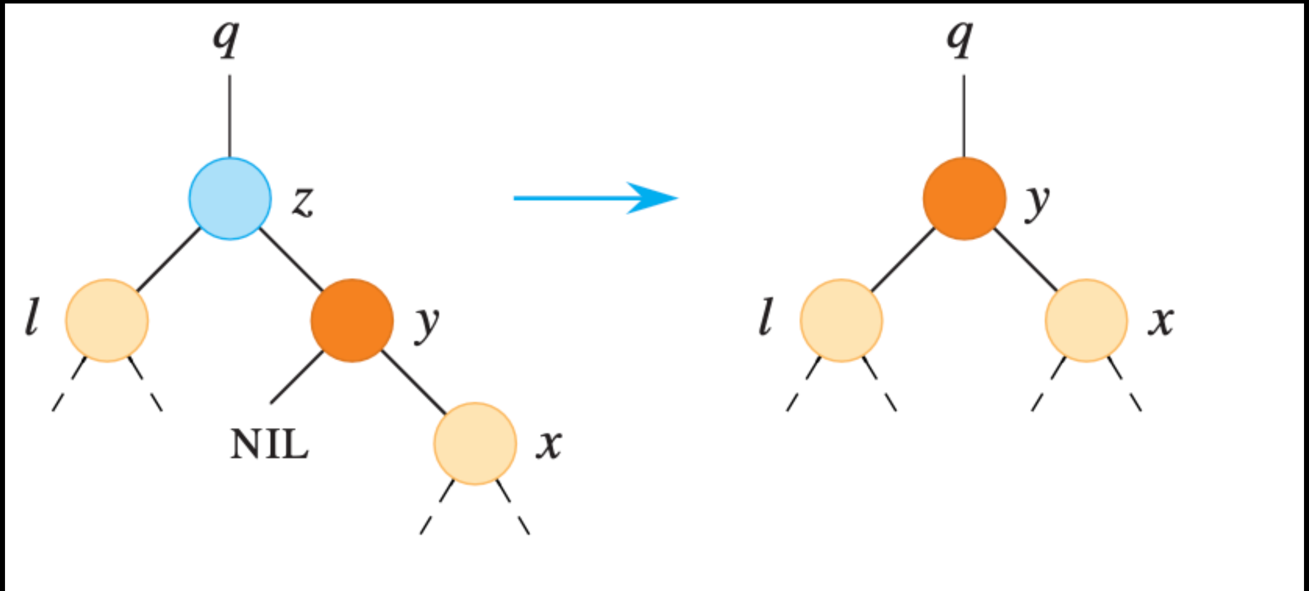
Insertion and Deletion

- Deleting a node with two children



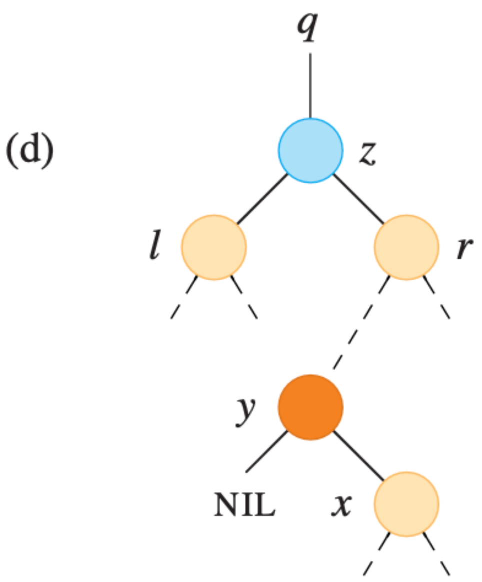
Insertion and Deletion

- Deleting a node with two children
  - Replace z with its right subtree and make the old left subtree to be the left subtree of the new node.



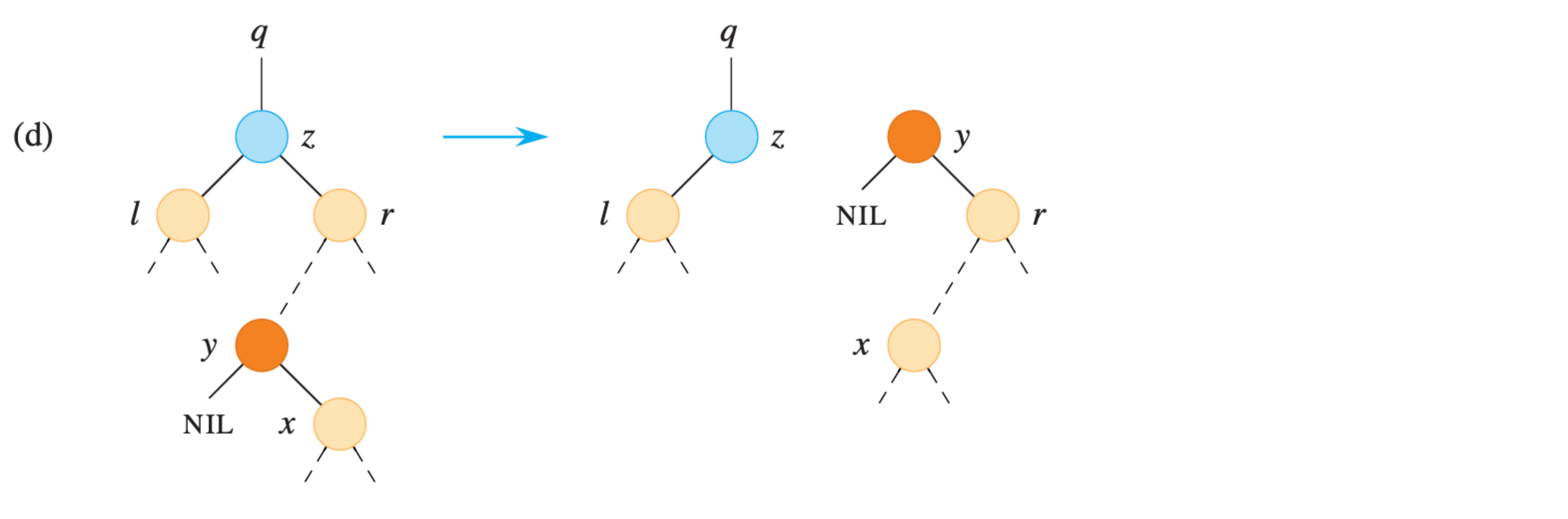
Insertion and Deletion

- Deleting a node with two children



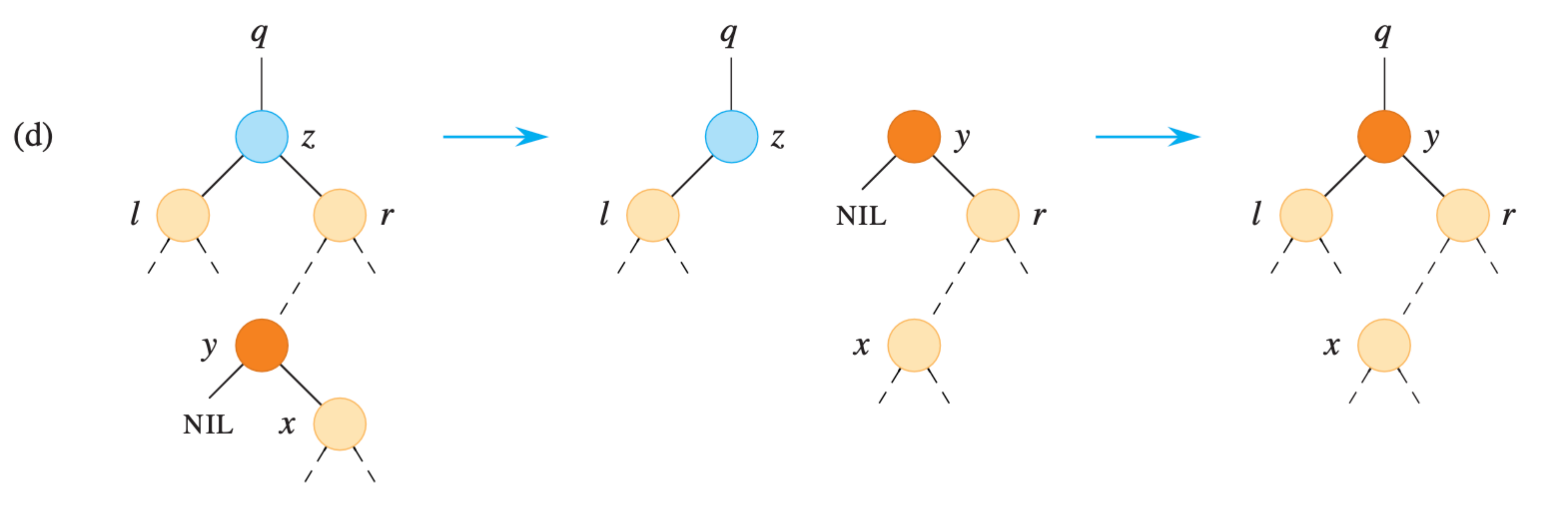
Insertion and Deletion

- Deleting a node with two children
  - Replace the successor of z, which is y, with its parent.



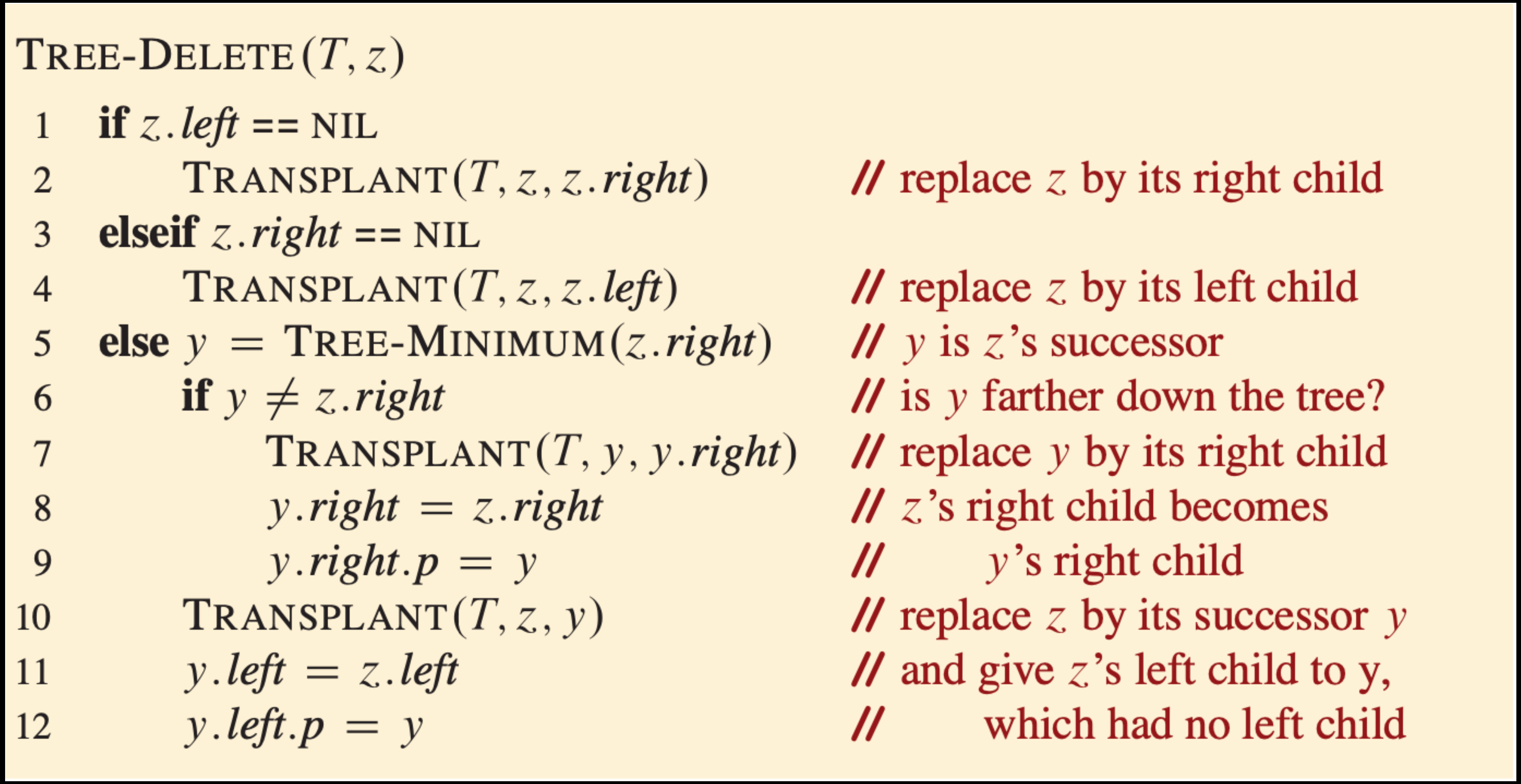
Insertion and Deletion

- Deleting a node with two children
  - Replace the successor of z, which is y, with its parent.
  - Set y to be in z's position (child of q)



Insertion and Deletion

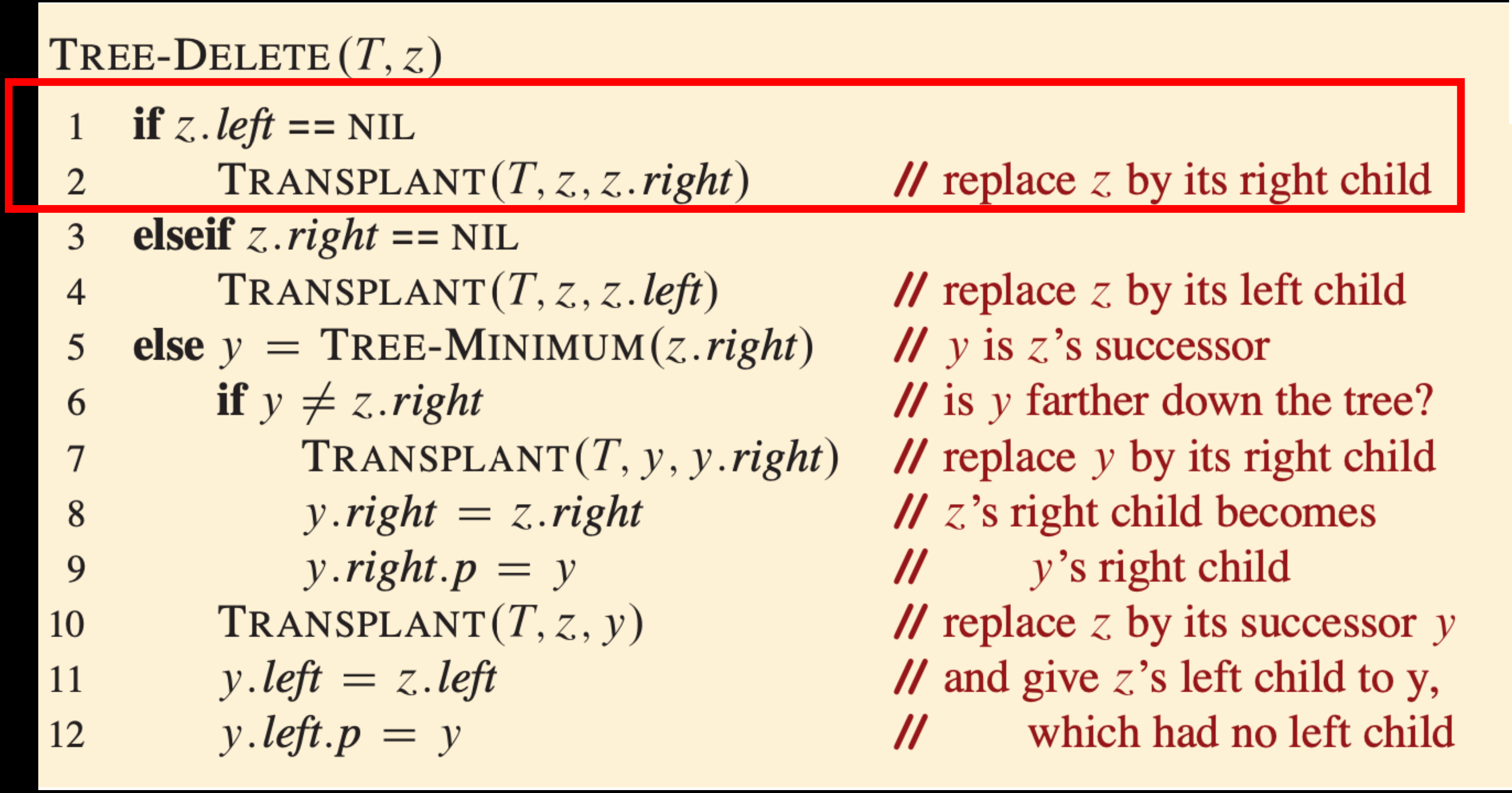
- TREE-DELETE(T, z)



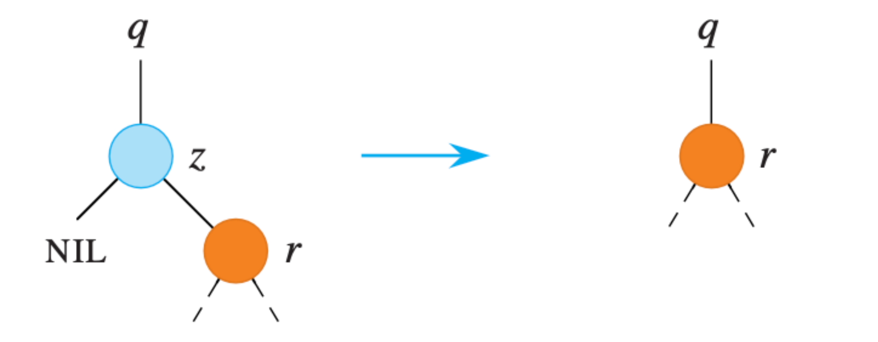
```
TREE-DELETE($T, z$)
if $z.left$ == NIL
    TRANSPLANT($T, z, z.right$)              // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT($T, z, z.left$)               // replace $z$ by its left child
else $y = $ TREE-MINIMUM($z.right$)          // $y$ is $z$'s successor
    if $y \neq z.right$                      // is $y$ farther down the tree?
        TRANSPLANT($T, y, y.right$)          // replace $y$ by its right child
        $y.right = z.right$                  // $z$'s right child becomes
        $y.right.p = y$                      //     $y$'s right child
    TRANSPLANT($T, z, y$)                    // replace $z$ by its successor $y$
    $y.left = z.left$                        // and give $z$'s left child to y,
    $y.left.p = y$                           //     which had no left child
```

Insertion and Deletion

- TREE-DELETE(T, z)



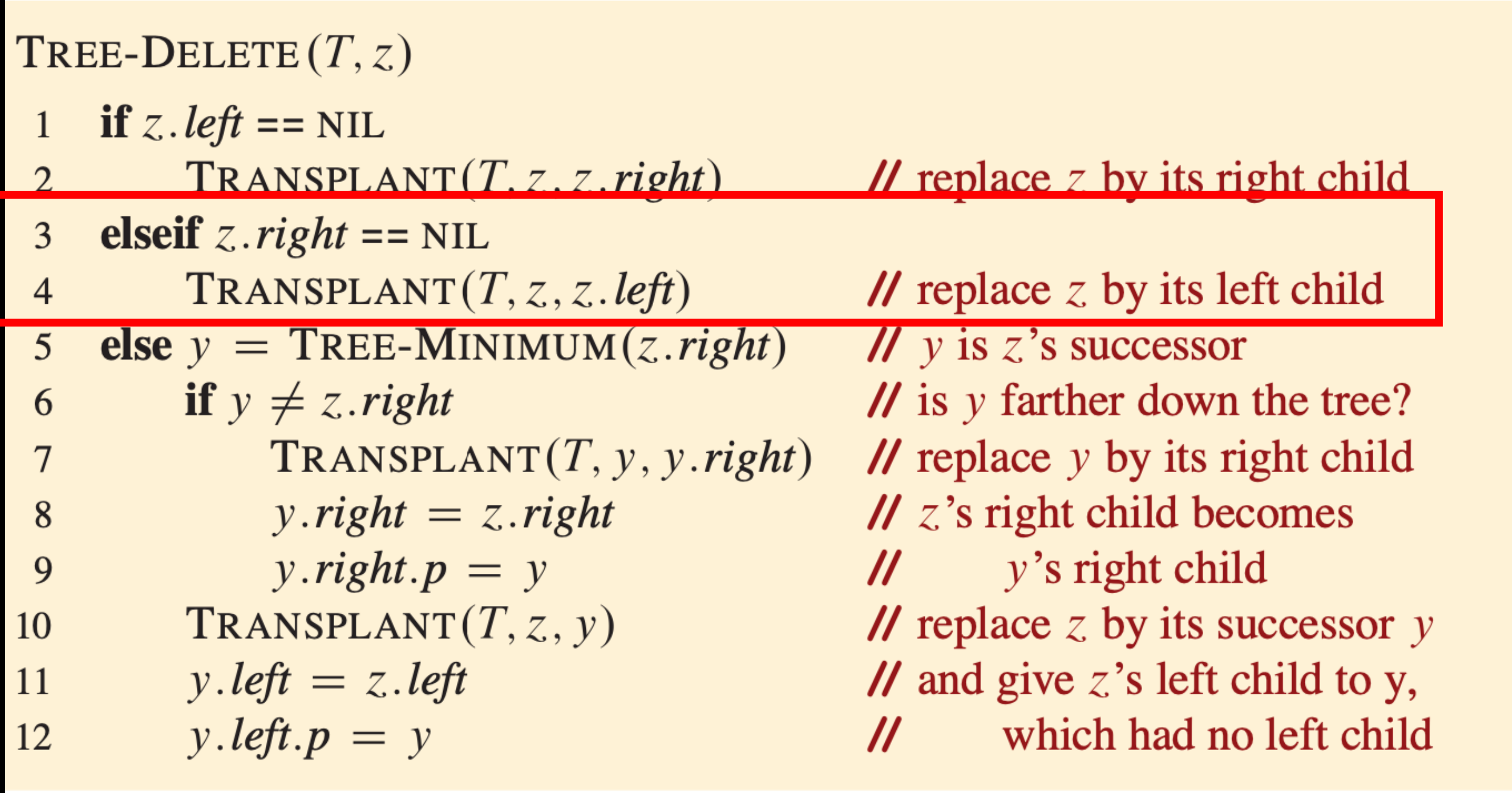
```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)          // replace z by its right child
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)           // replace z by its left child
else y = TREE-MINIMUM(z.right)         // y is z's successor
    if y ≠ z.right                     // is y farther down the tree?
        TRANSPLANT(T, y, y.right)      // replace y by its right child
        y.right = z.right              // z's right child becomes
        y.right.p = y                  //        y's right child
    TRANSPLANT(T, z, y)                // replace z by its successor y
    y.left = z.left                    // and give z's left child to y,
    y.left.p = y                       //        which had no left child
```



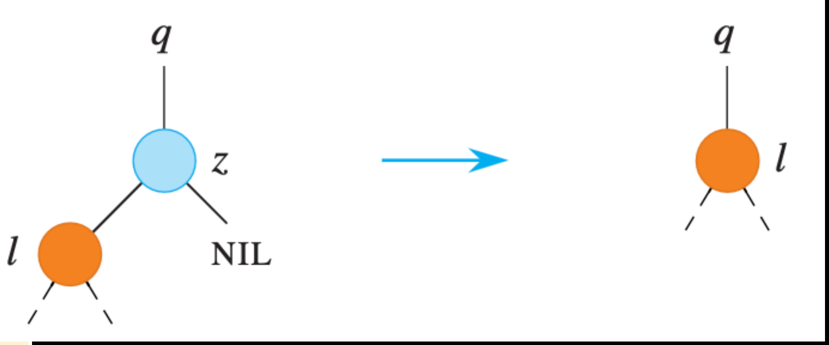
Handles the case where *z* has a right child but no left child

Insertion and Deletion

- TREE-DELETE(T, z)



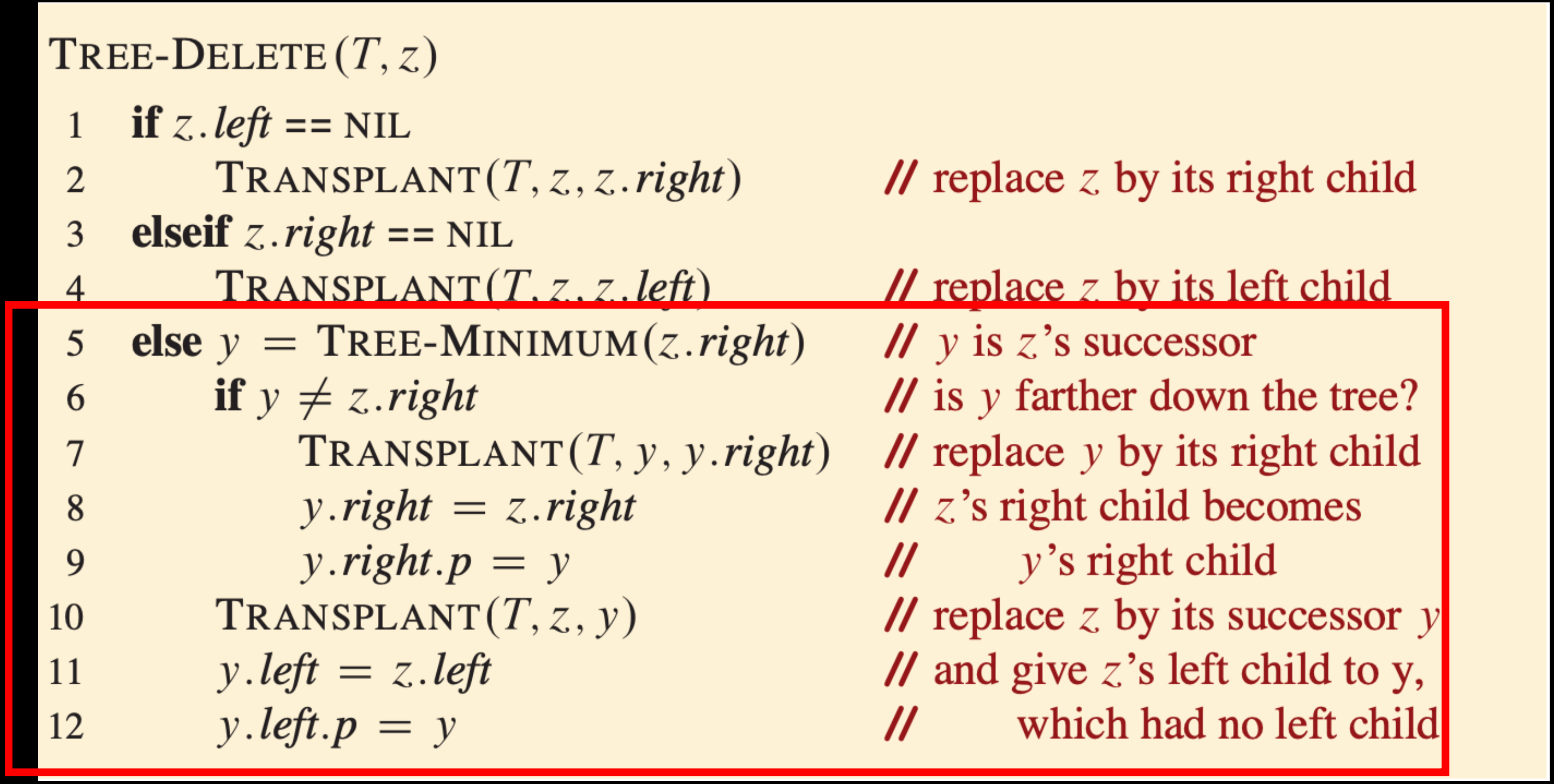
```
TREE-DELETE($T, z$)
if $z.left$ == NIL
    TRANSPLANT($T, z, z.right$)        // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT($T, z, z.left$)         // replace $z$ by its left child
else $y = $ TREE-MINIMUM($z.right$)    // $y$ is $z$'s successor
    if $y \neq z.right$                // is $y$ farther down the tree?
        TRANSPLANT($T, y, y.right$)    // replace $y$ by its right child
        $y.right = z.right$            // $z$'s right child becomes
        $y.right.p = y$                //       $y$'s right child
    TRANSPLANT($T, z, y$)              // replace $z$ by its successor $y$
    $y.left = z.left$                  // and give $z$'s left child to y,
    $y.left.p = y$                     //       which had no left child
```



Handles the case where *z* has a left child but no right child

Insertion and Deletion

- TREE-DELETE(T, z)

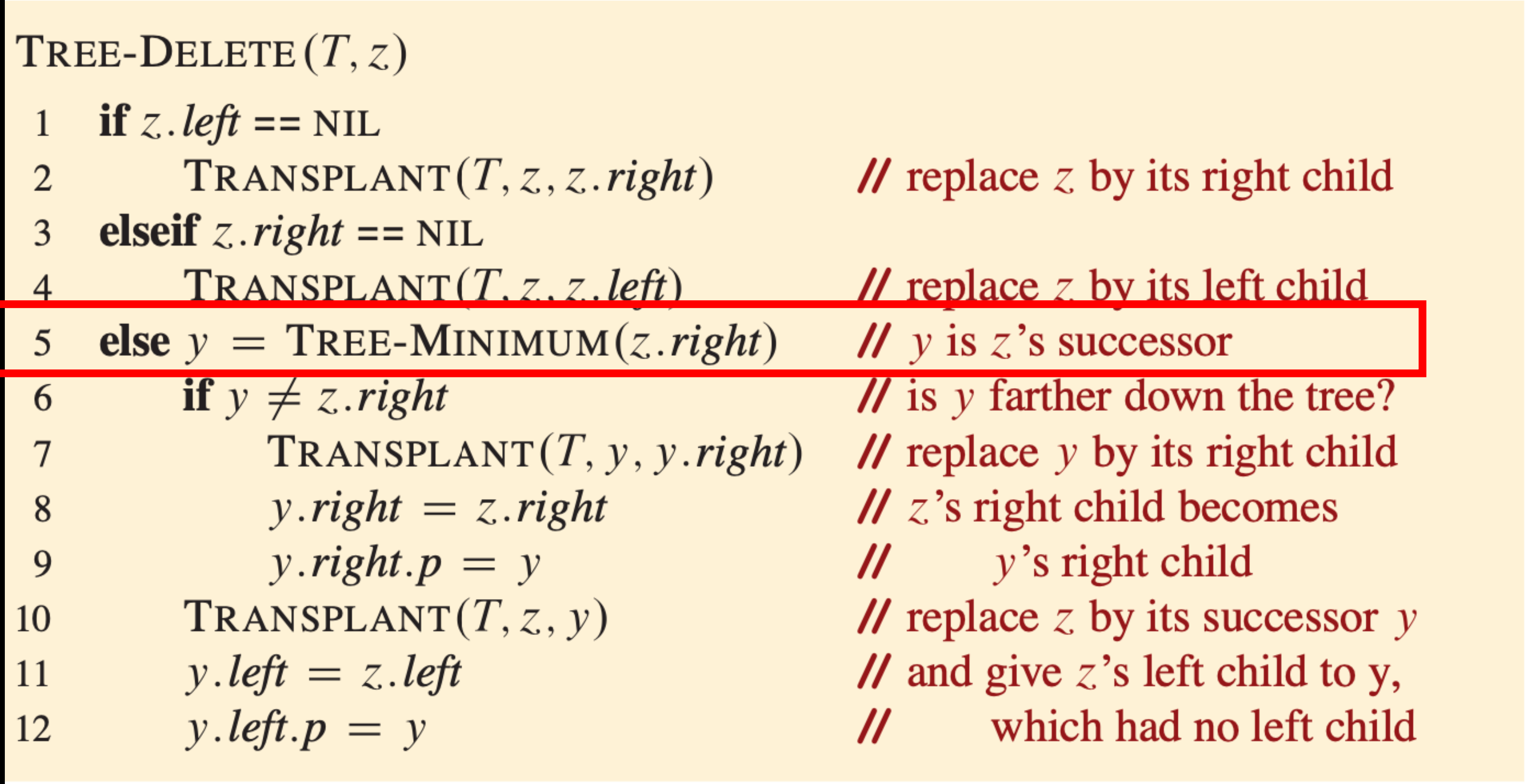


```
TREE-DELETE(T, z)
if $z.left$ == NIL
    TRANSPLANT($T, z, z.right$)          // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT($T, z, z.left$)           // replace $z$ by its left child
else $y$ = TREE-MINIMUM($z.right$)       // $y$ is $z$'s successor
    if $y \neq z.right$                  // is $y$ farther down the tree?
        TRANSPLANT($T, y, y.right$)      // replace $y$ by its right child
        $y.right = z.right$              // $z$'s right child becomes
        $y.right.p = y$                  //       $y$'s right child
    TRANSPLANT($T, z, y$)                // replace $z$ by its successor $y$
    $y.left = z.left$                    // and give $z$'s left child to y,
    $y.left.p = y$                       //       which had no left child
```

Handles the remaining two cases, in which z has two children.

Insertion and Deletion

- TREE-DELETE(T, z)



```
TREE-DELETE($T, z$)
if $z.left$ == NIL
    TRANSPLANT($T, z, z.right$)          // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT($T, z, z.left$)           // replace $z$ by its left child
else $y$ = TREE-MINIMUM($z.right$)       // $y$ is $z$'s successor
    if $y \neq z.right$                  // is $y$ farther down the tree?
        TRANSPLANT($T, y, y.right$)      // replace $y$ by its right child
        $y.right = z.right$              // $z$'s right child becomes
        $y.right.p = y$                  //       $y$'s right child
    TRANSPLANT($T, z, y$)                // replace $z$ by its successor $y$
    $y.left = z.left$                    // and give $z$'s left child to y,
    $y.left.p = y$                       //       which had no left child
```

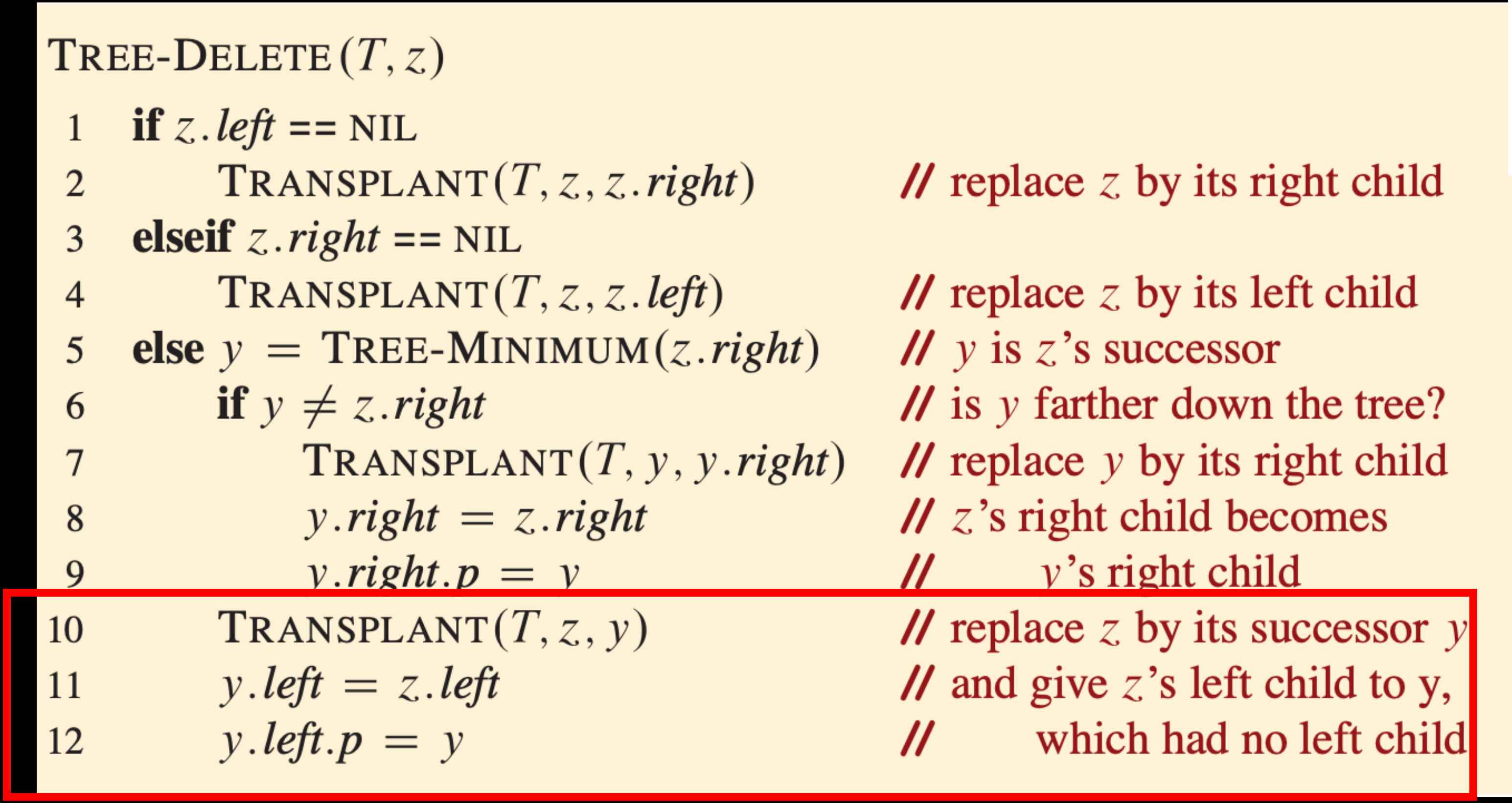
Find the successor of z, call it y.

Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key

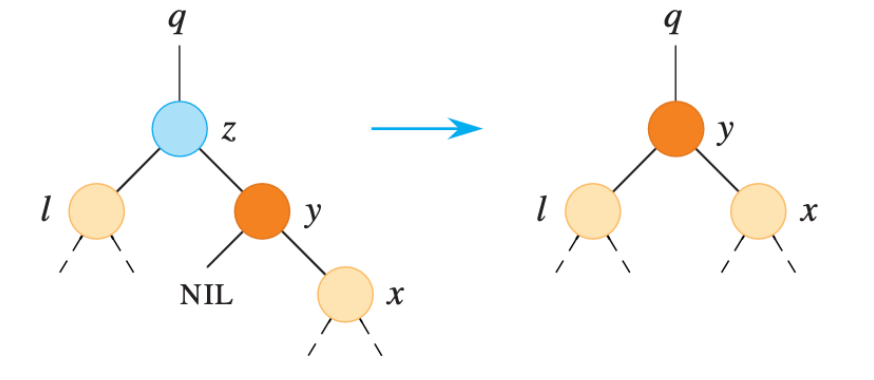
y have no left child

Insertion and Deletion

- TREE-DELETE(T, z)



```
TREE-DELETE$(T, z)$
if $z.left$ == NIL
    TRANSPLANT$(T, z, z.right)$          // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT$(T, z, z.left)$           // replace $z$ by its left child
else $y = $ TREE-MINIMUM$(z.right)$      // $y$ is $z$'s successor
    if $y \neq z.right$                  // is $y$ farther down the tree?
        TRANSPLANT$(T, y, y.right)$      // replace $y$ by its right child
        $y.right = z.right$              // $z$'s right child becomes
        $y.right.p = y$                  //       $y$'s right child
    TRANSPLANT$(T, z, y)$                // replace $z$ by its successor $y$
    $y.left = z.left$                    // and give $z$'s left child to y,
    $y.left.p = y$                       //       which had no left child
```



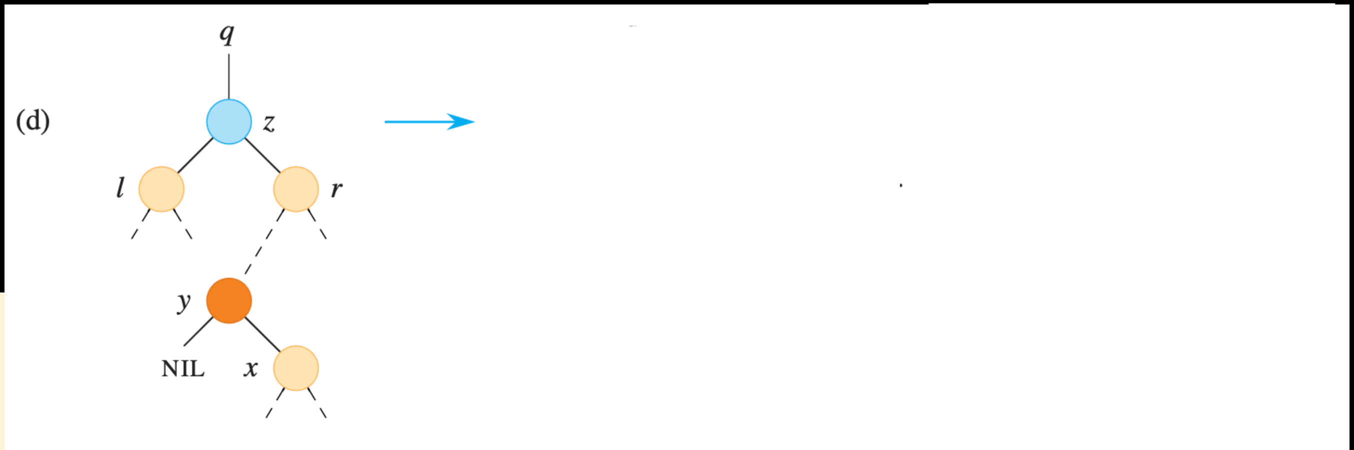
If y is the right child of z:

1) Replace z by y
2) Replace the left child of y by the left child of z
3) Node y keeps its right child

Insertion and Deletion

- TREE-DELETE(T, z)

```
TREE-DELETE($T, z$)
if $z.left$ == NIL
    TRANSPLANT($T, z, z.right$)        // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT($T, z, z.left$)         // replace $z$ by its left child
else $y = $ TREE-MINIMUM($z.right$)    // $y$ is $z$'s successor
    if $y \neq z.right$                // is $y$ farther down the tree?
        TRANSPLANT($T, y, y.right$)    // replace $y$ by its right child
        $y.right = z.right$            // $z$'s right child becomes
        $y.right.p = y$                //         $y$'s right child
    TRANSPLANT($T, z, y$)              // replace $z$ by its successor $y$
    $y.left = z.left$                  // and give $z$'s left child to y,
    $y.left.p = y$                     //         which had no left child
```

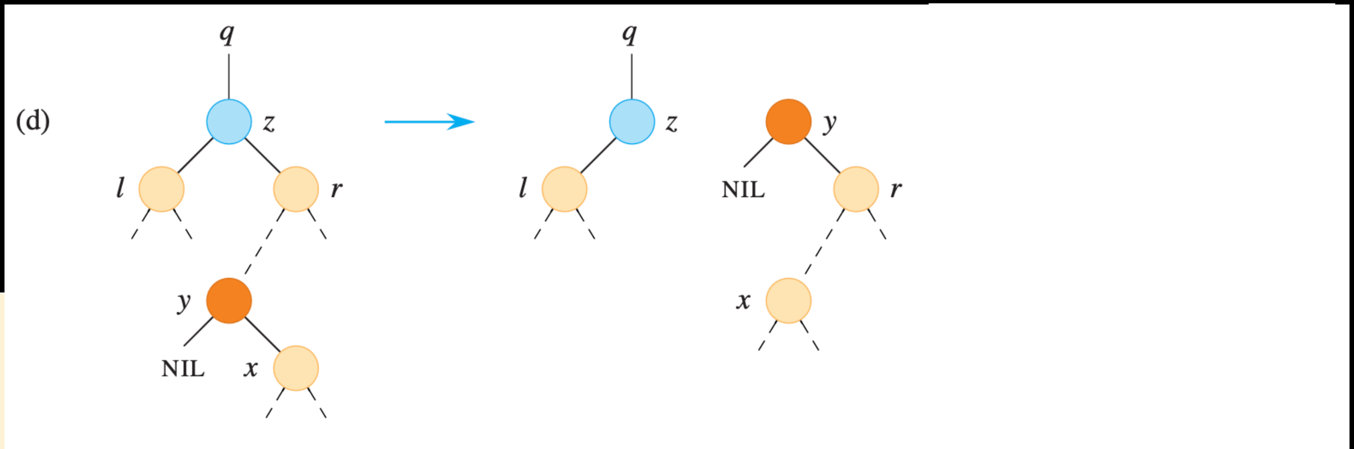


If y is NOT the right child of z,

Insertion and Deletion

- TREE-DELETE(T, z)

```
TREE-DELETE($T, z$)
if $z.left$ == NIL
    TRANSPLANT($T, z, z.right$)        // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT($T, z, z.left$)         // replace $z$ by its left child
else $y = $ TREE-MINIMUM($z.right$)    // $y$ is $z$'s successor
    if $y \neq z.right$                // is $y$ farther down the tree?
        TRANSPLANT($T, y, y.right$)    // replace $y$ by its right child
        $y.right = z.right$            // $z$'s right child becomes
        $y.right.p = y$                //       $y$'s right child
    TRANSPLANT($T, z, y$)              // replace $z$ by its successor $y$
    $y.left = z.left$                  // and give $z$'s left child to y,
    $y.left.p = y$                     //       which had no left child
```

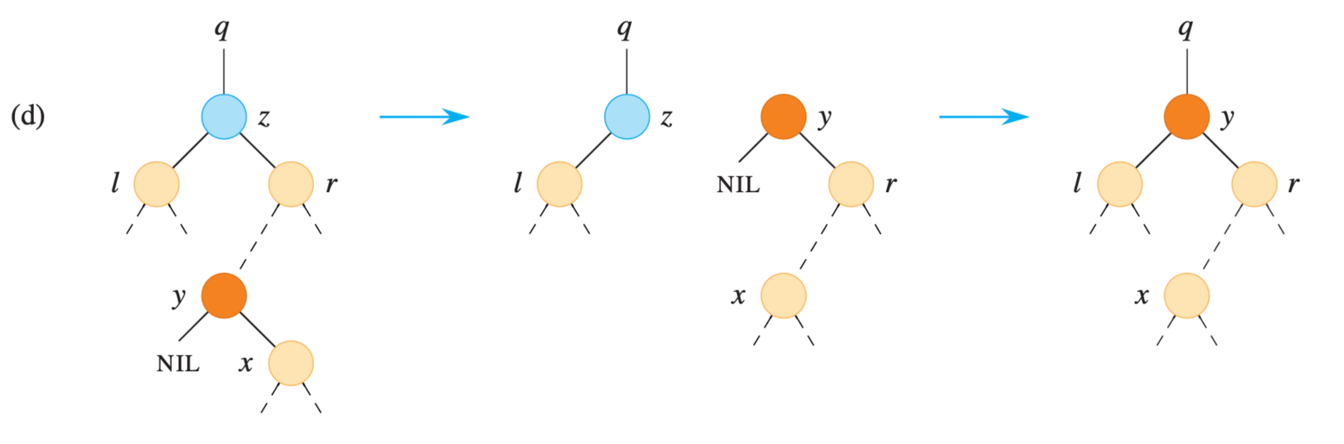


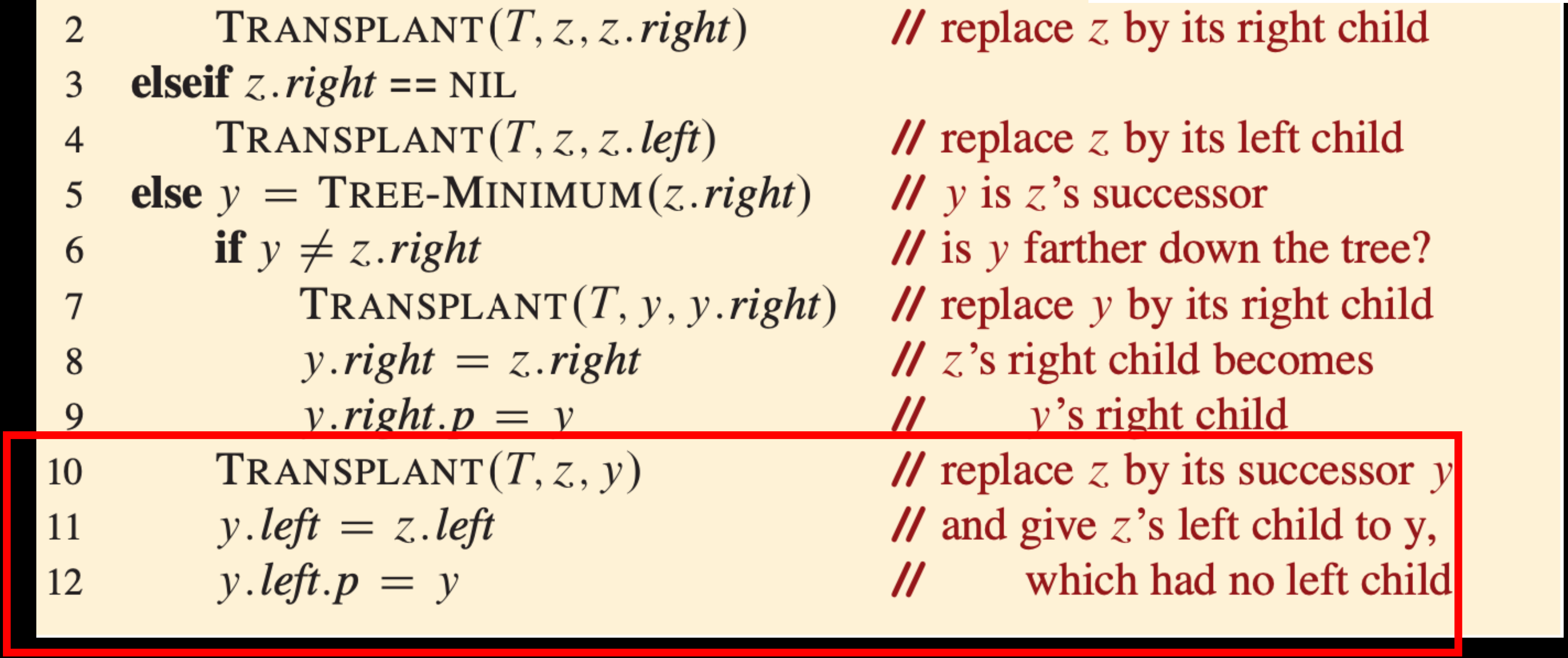
If y is NOT the right child of z, then two nodes must move:

1) Replace y by its right child
2) Set the right child of y to be the right child of z

Insertion and Deletion

- TREE-DELETE(T, z)





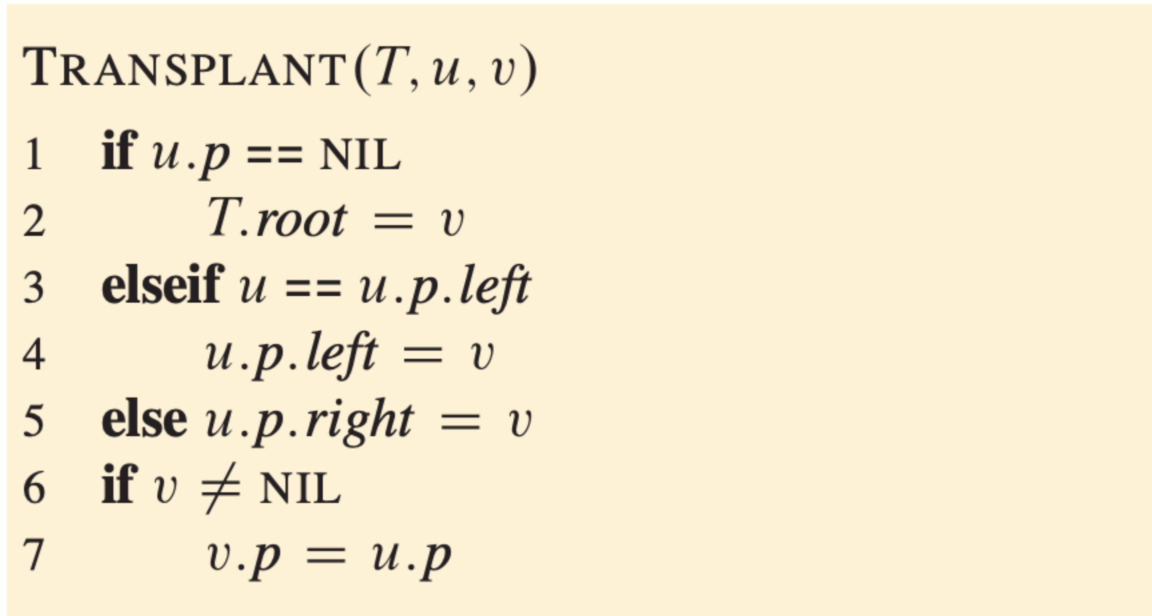
```
TREE-DELETE(T, z)
if $z.left$ == NIL
    TRANSPLANT($T, z, z.right$)        // replace $z$ by its right child
elseif $z.right$ == NIL
    TRANSPLANT($T, z, z.left$)         // replace $z$ by its left child
else $y = $ TREE-MINIMUM($z.right$)    // $y$ is $z$'s successor
    if $y \neq z.right$                // is $y$ farther down the tree?
        TRANSPLANT($T, y, y.right$)    // replace $y$ by its right child
        $y.right = z.right$            // $z$'s right child becomes
        $y.right.p = y$                //         $y$'s right child
    TRANSPLANT($T, z, y$)              // replace $z$ by its successor $y$
    $y.left = z.left$                  // and give $z$'s left child to y,
    $y.left.p = y$                     //         which had no left child
```

If y is NOT the right child of z, then two nodes must move:

1) Replace y by its right child
2) Set the right child of y to be the right child of z
3) Put y with its subtrees in place of z

Insertion and Deletion

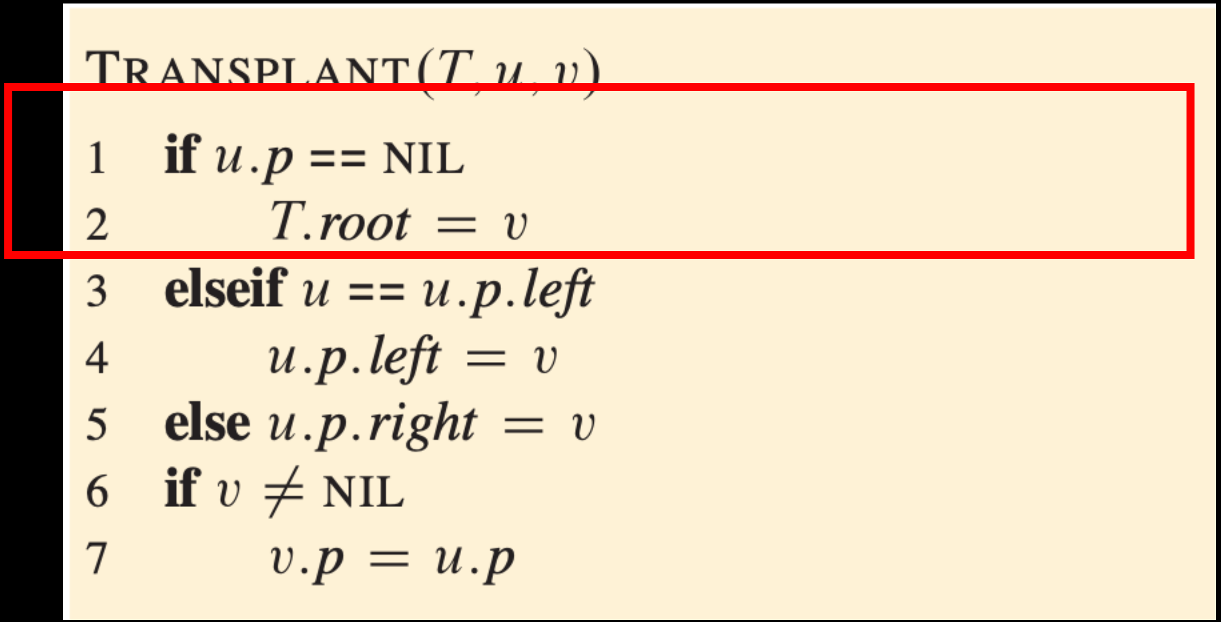
- TRANSPLANT(T, u, v): replaces the subtree rooted at node u with the subtree rooted at node v



```
TRANSPLANT($T, u, v$)
if $u.p$ == NIL
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$
```

Insertion and Deletion

- TRANSPLANT(T, u, v): replaces the subtree rooted at node u with the subtree rooted at node v

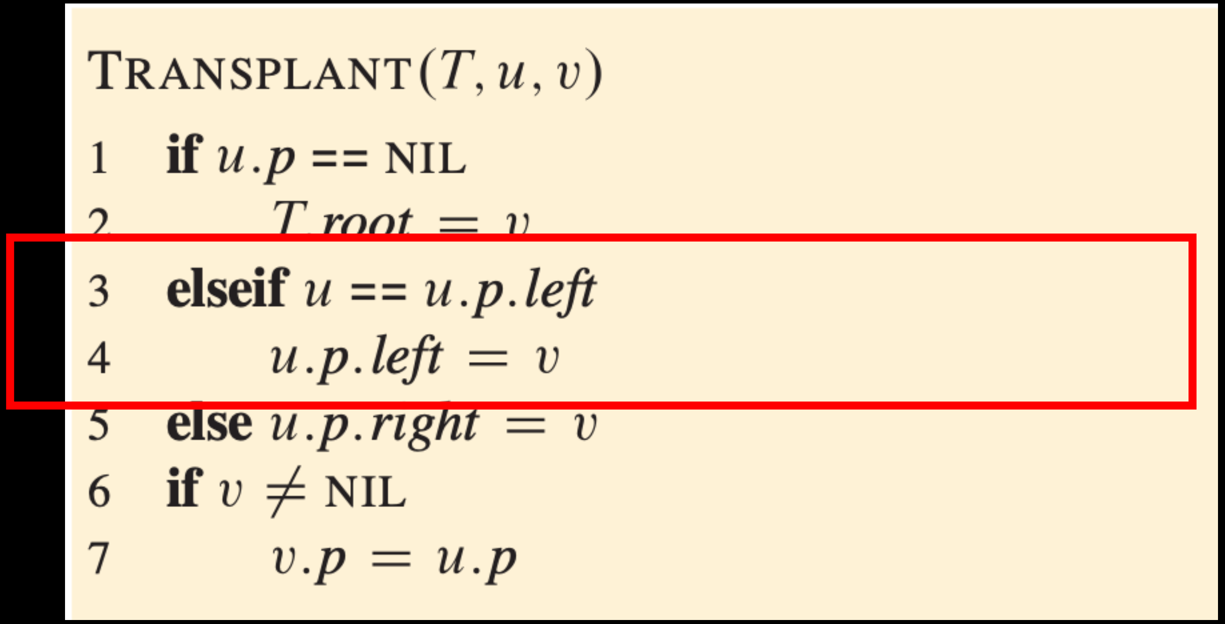


```
TRANSPLANT($T, u, v$)
if $u.p$ == NIL
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$
```

Handles the case in which u is the root. Otherwise, u is either a left child or a right child of its parent.

Insertion and Deletion

- TRANSPLANT(T, u, v): replaces the subtree rooted at node u with the subtree rooted at node v

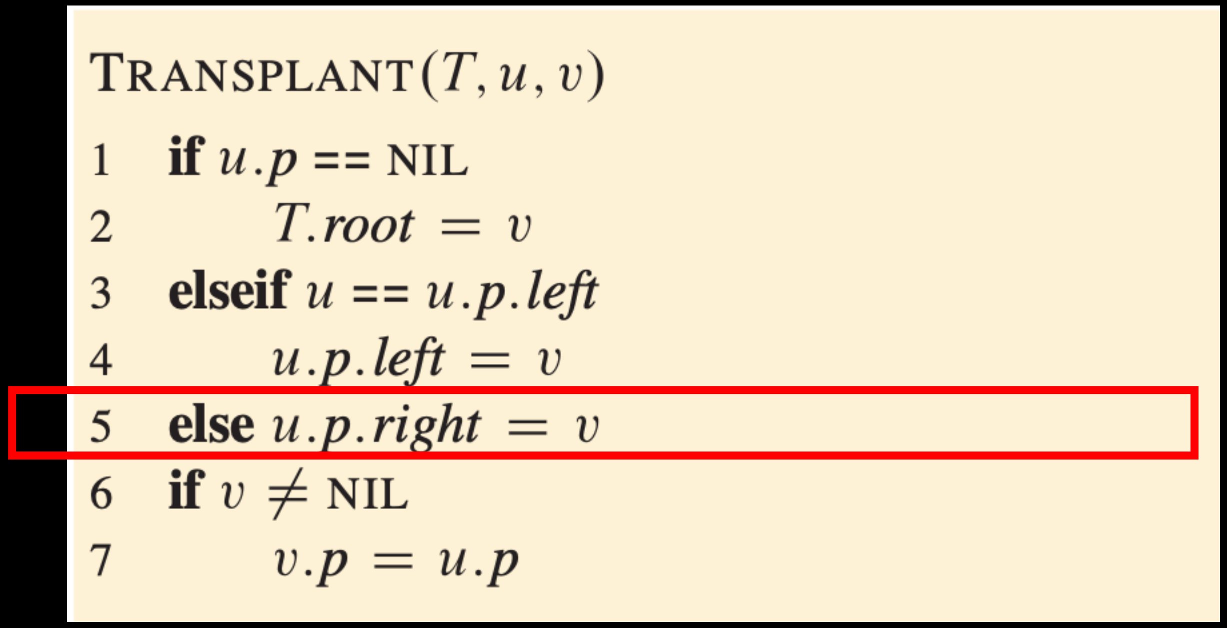


```
TRANSPLANT($T, u, v$)
if $u.p$ == NIL
    $T.root = v$
elseif $u$ == $u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$
```

If u is a left child of its parent, then make the parent’s left child be v

Insertion and Deletion

- TRANSPLANT(T, u, v): replaces the subtree rooted at node u with the subtree rooted at node v

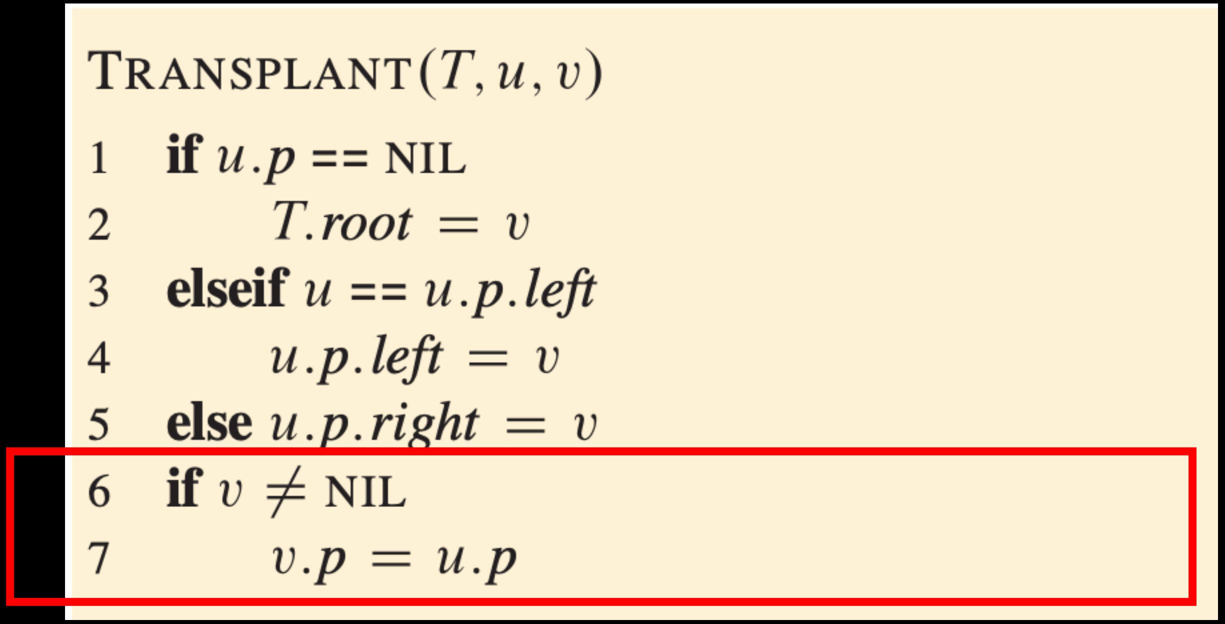


```
TRANSPLANT$(T, u, v)$
if $u.p$ == NIL
    $T.root = v$
elseif $u$ == $u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$
```

If u is a right child of its parent, then make its parent be the node v

Insertion and Deletion

- TRANSPLANT(T, u, v): replaces the subtree rooted at node u with the subtree rooted at node v



```
TRANSPLANT($T, u, v$)
if $u.p$ == NIL
        $T.root = v$
elseif $u == u.p.left$
        $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
        $v.p = u.p$
```

If v is not NIL, then make it points to its parent, which is u's parent.

Content

Content
Binary Search Tree
Querying Binary Search Tree
Insertion and Deletion
➡ Exercises

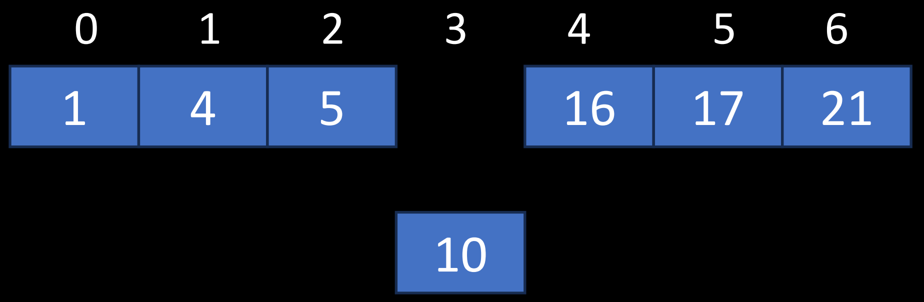
Exercises

- For the set {1,4,5,10,16,17,21} of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

Exercises

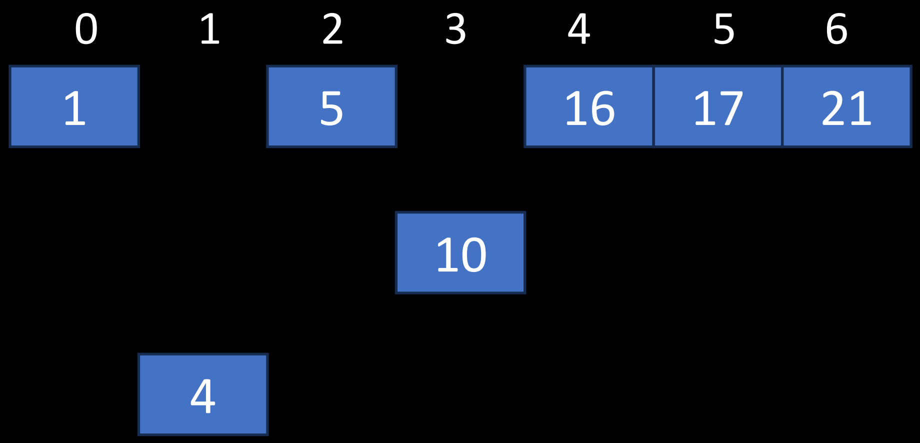
0	1	2	3	4	5	6
1	4	5	10	16	17	21

Exercises



0	1	2	3	4	5	6
1	4	5		16	17	21
			10			

Exercises

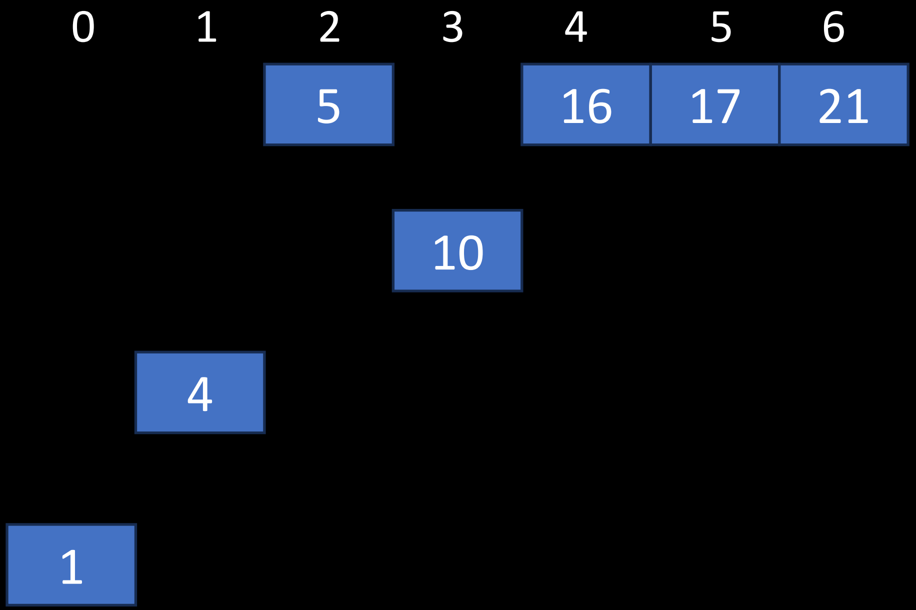


0	1	2	3	4	5	6
1		5		16	17	21

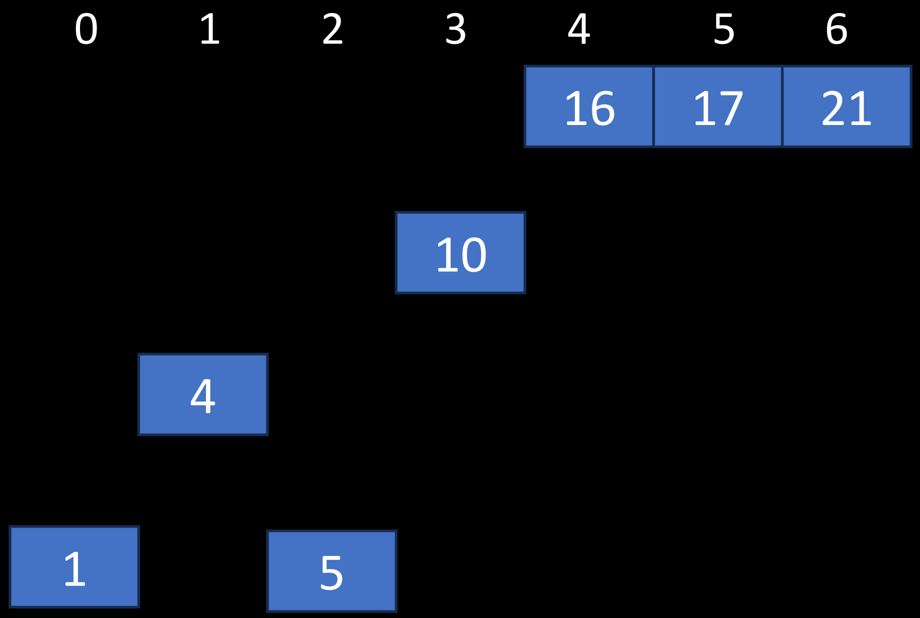
10

4

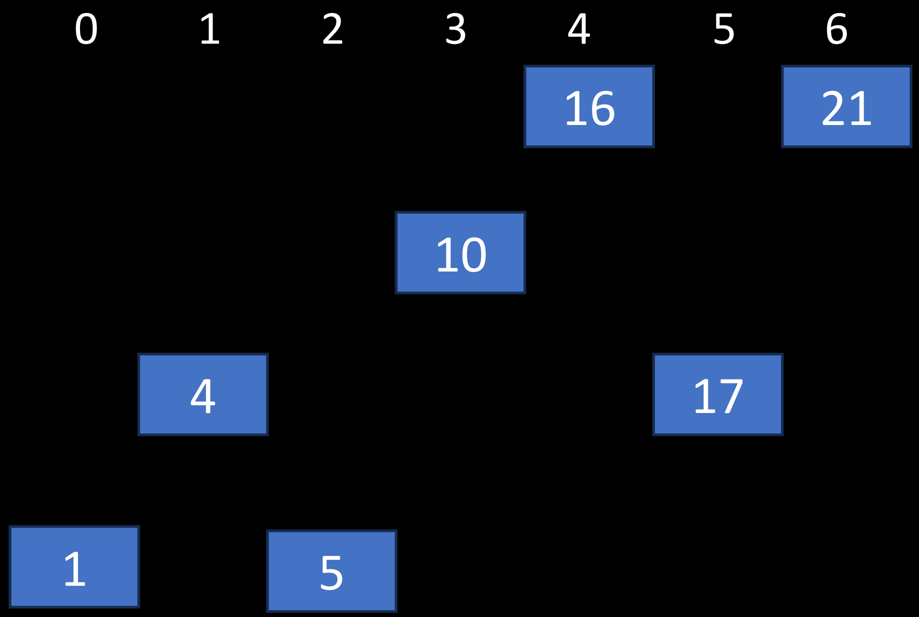
Exercises



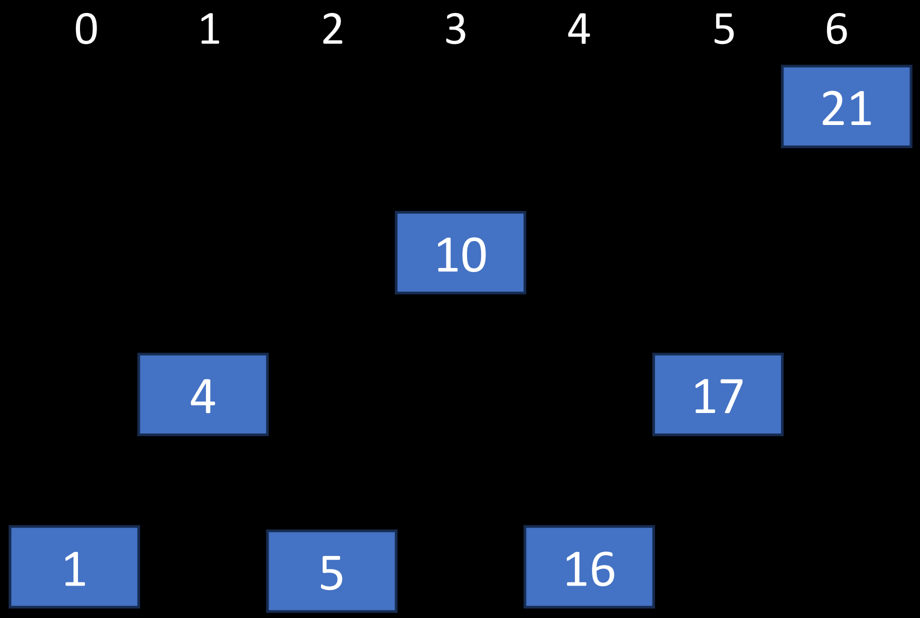
Exercises



Exercises



Exercises



0 1 2 3 4 5 6

21

10

4

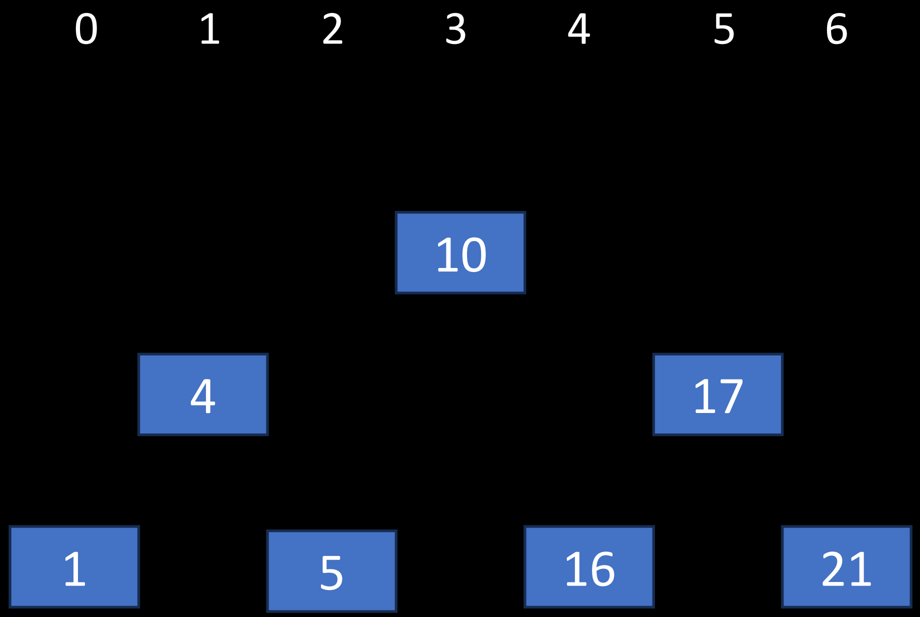
17

1

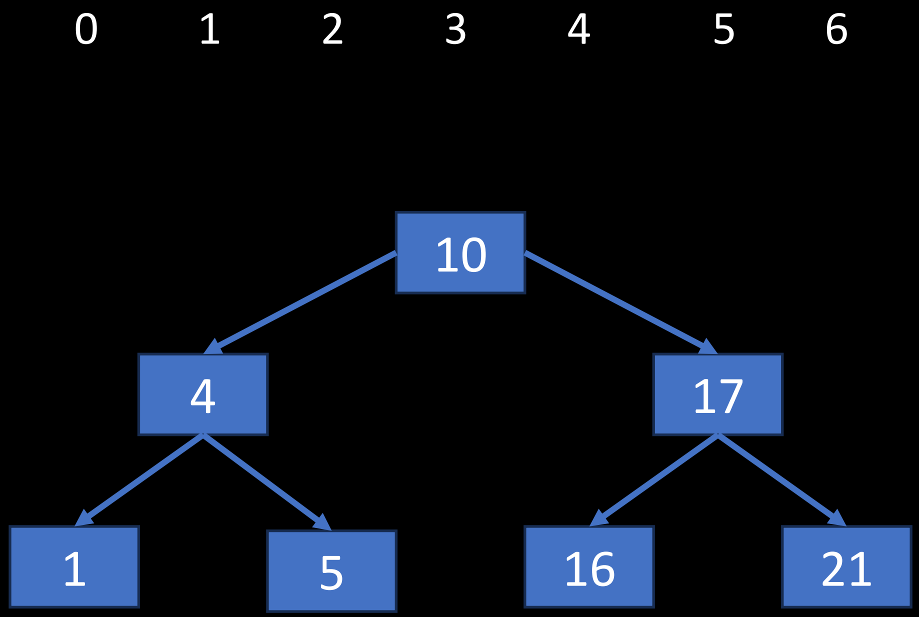
5

16

Exercises

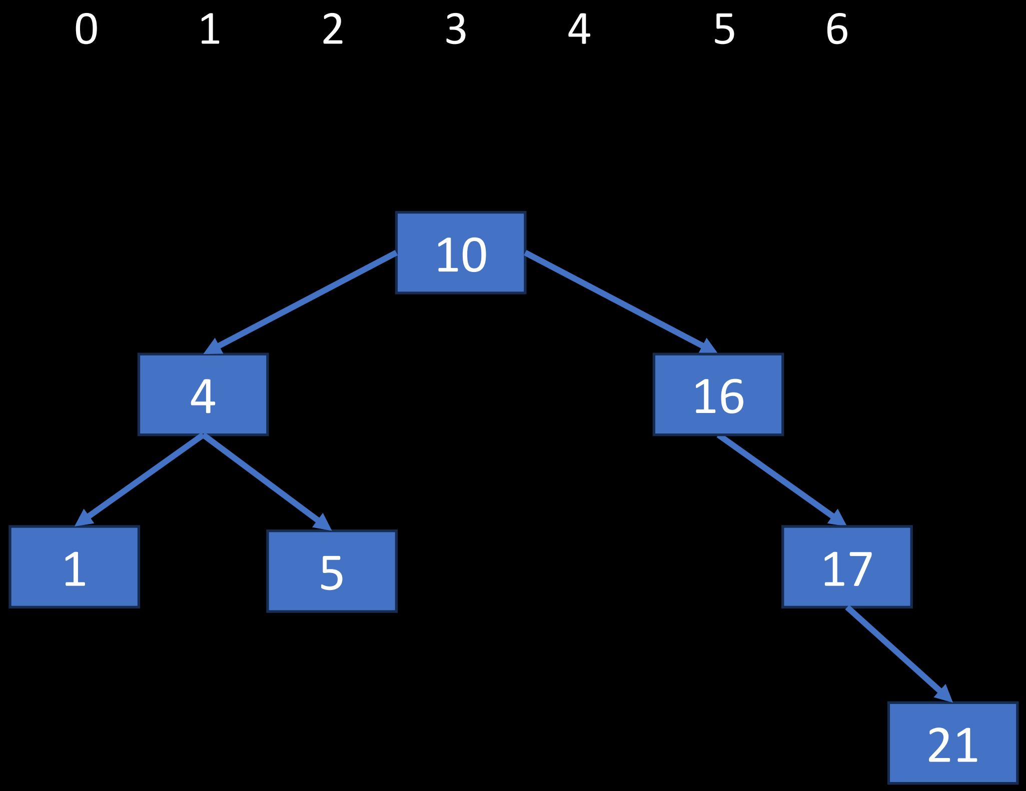


Exercises



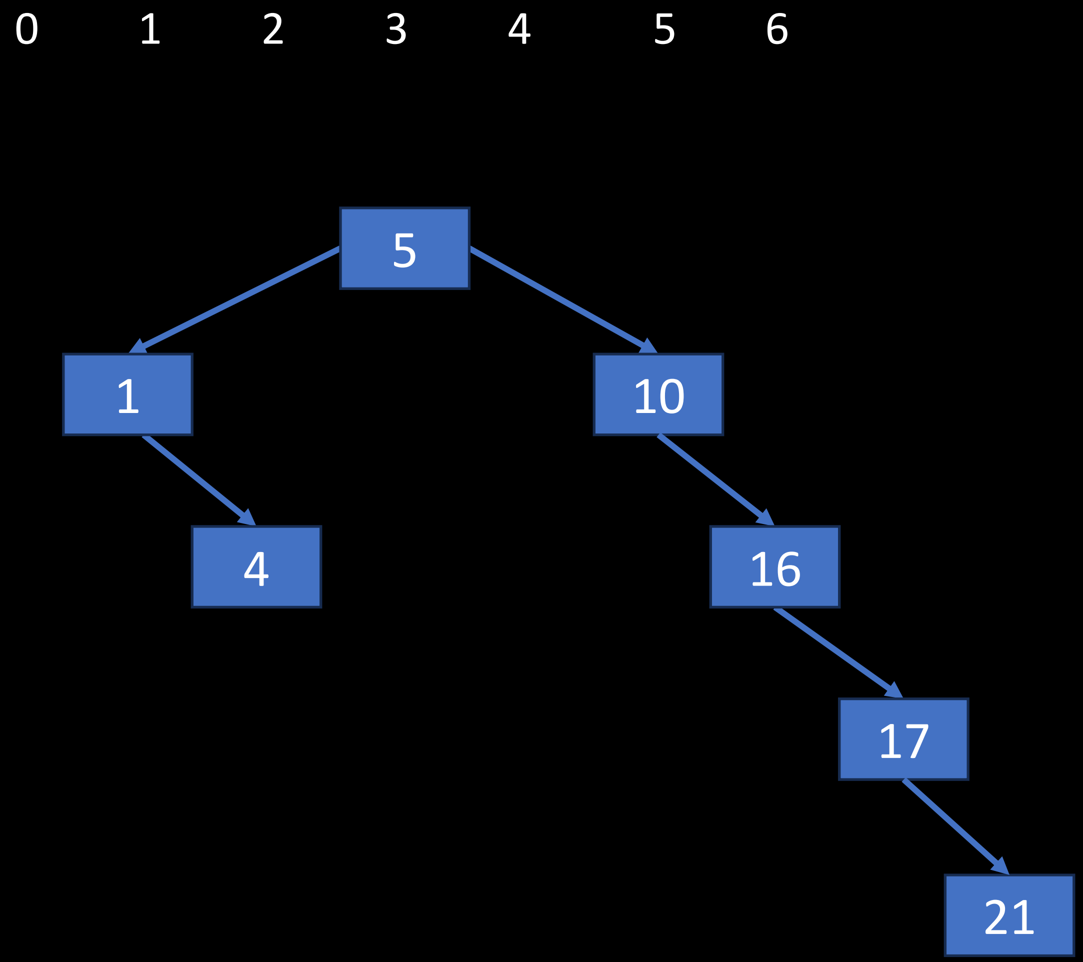
Height = 2

Exercises



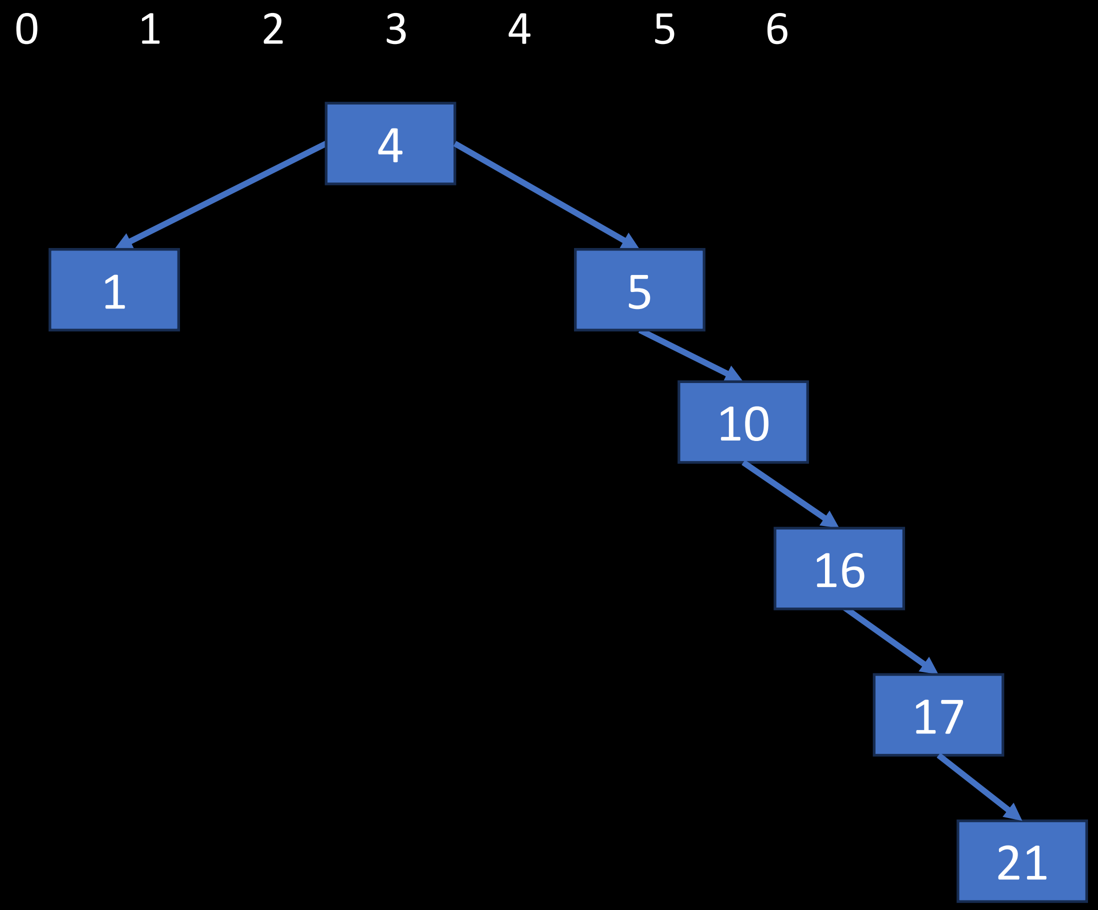
Height = 3

Exercises



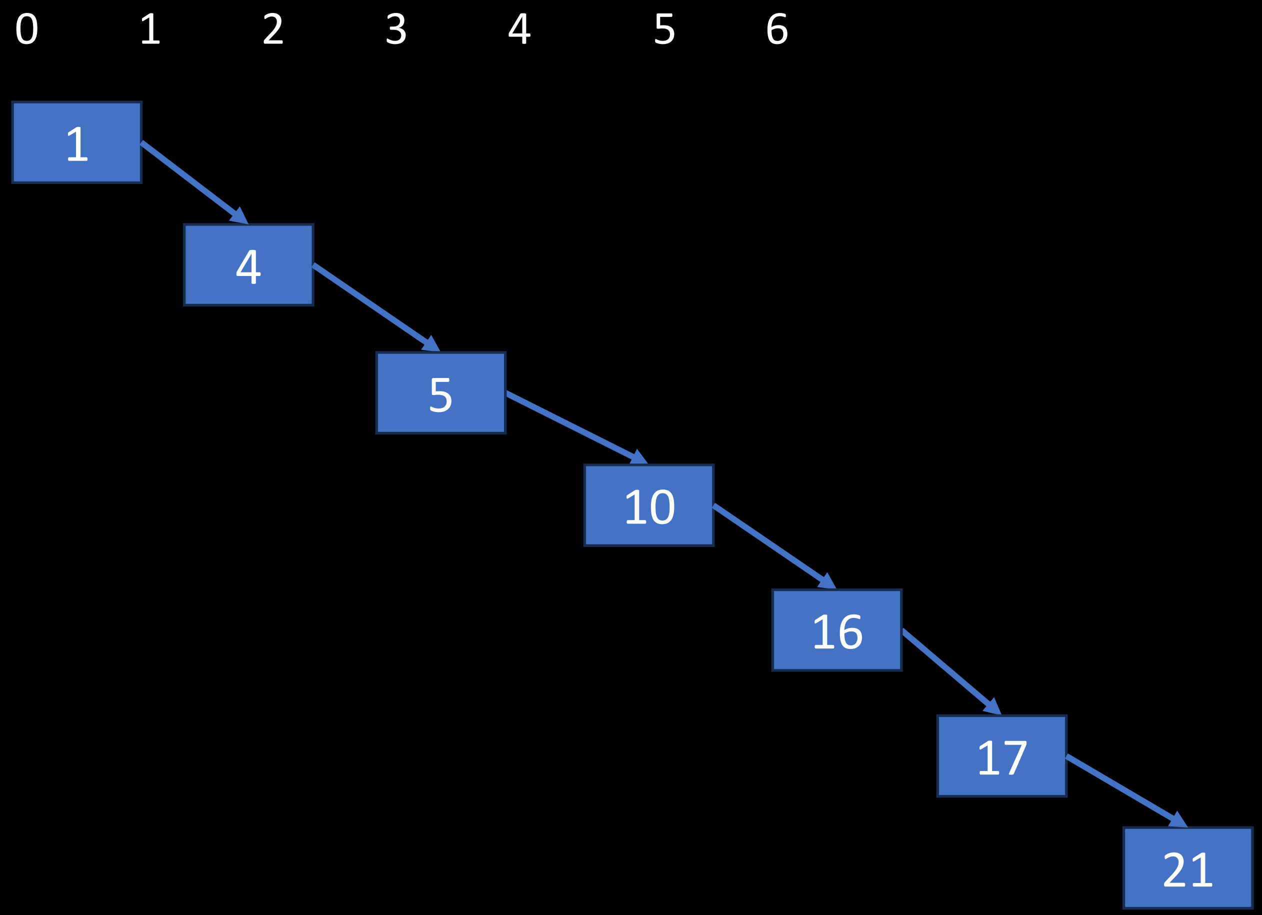
Height = 4

Exercises



Height = 5

Exercises

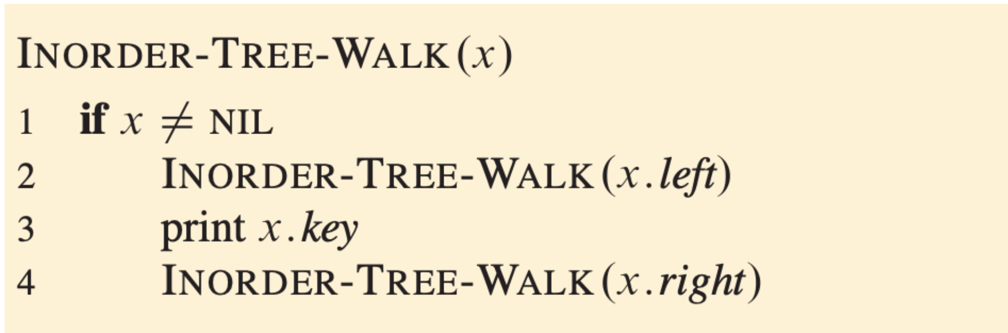


Height = 6

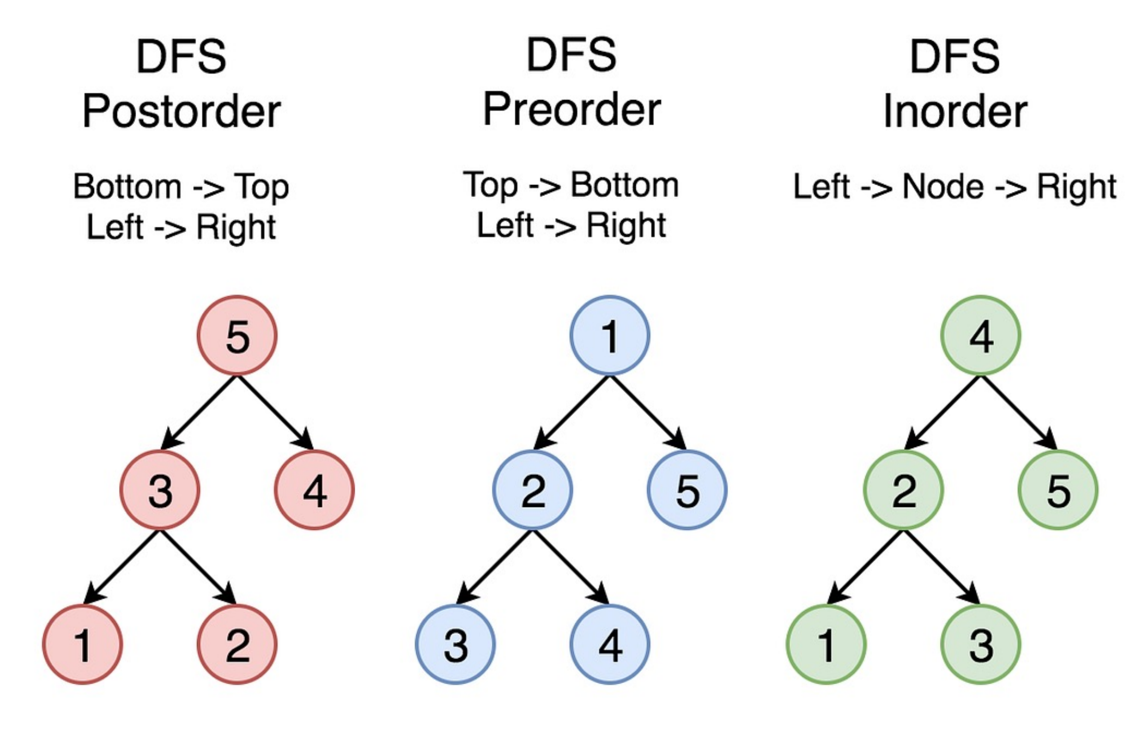
Exercises

- Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

Exercises



```
INORDER-TREE-WALK(x)
if $x \neq$ NIL
    INORDER-TREE-WALK($x.left$)
    print $x.key$
    INORDER-TREE-WALK($x.right$)
```



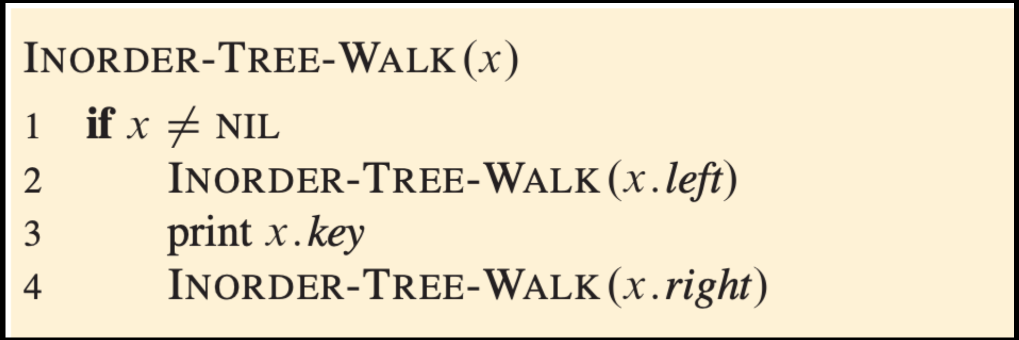
Postorder: Left → Right → root :: 1, 2, 3, 4, 5

Preorder: root → left → right :: 1, 2, 3, 4, 5

Inorder: left → root → right :: 1, 2, 3, 4, 5

Src: https://samanbatool08.medium.com/trees-binary-search-trees-and-traversal-methods-the-difference-and-why-c52edd53cc31

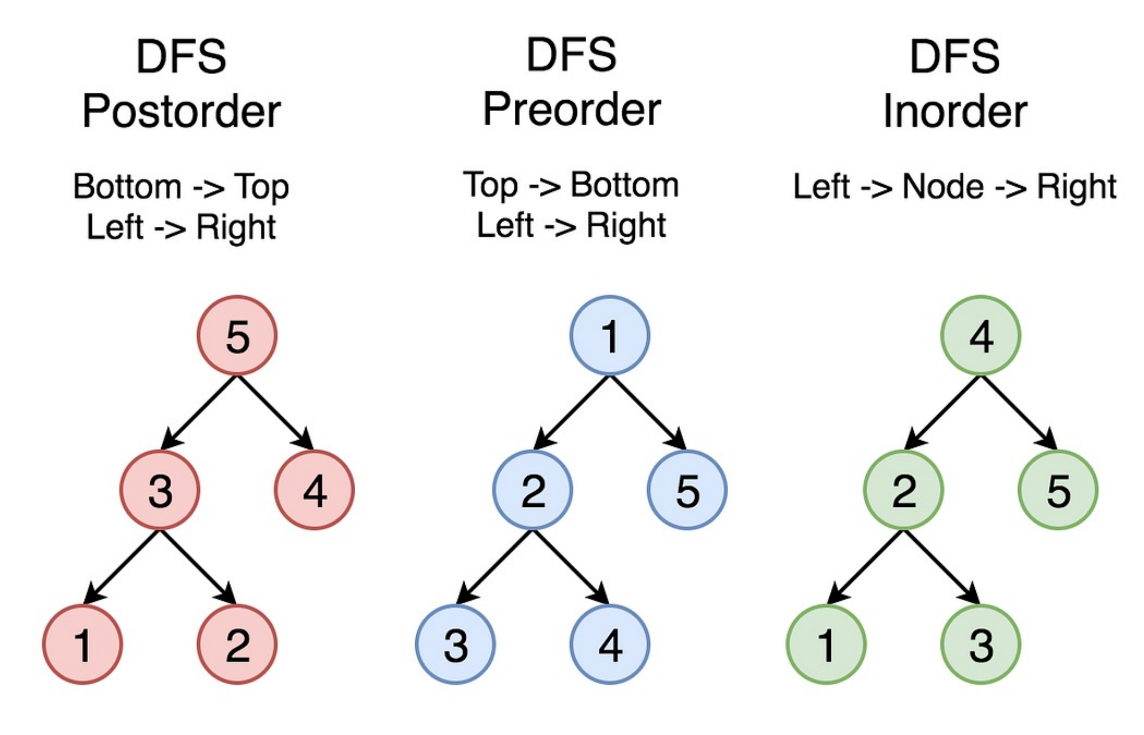
Exercises



```
INORDER-TREE-WALK(x)
if $x \neq$ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

```
PREORDER-TREE-WALK(x)
  if x != NIL
    print x.key
    PREORDER-TREE-WALK(x.left)
    PREORDER-TREE-WALK(x.right)
```

```
POSTORDER-TREE-WALK(x)
  if x != NIL
    POSTORDER-TREE-WALK(x.left)
    POSTORDER-TREE-WALK(x.right)
    print x.key
```



Postorder: Left → Right → root :: 1, 2, 3, 4, 5

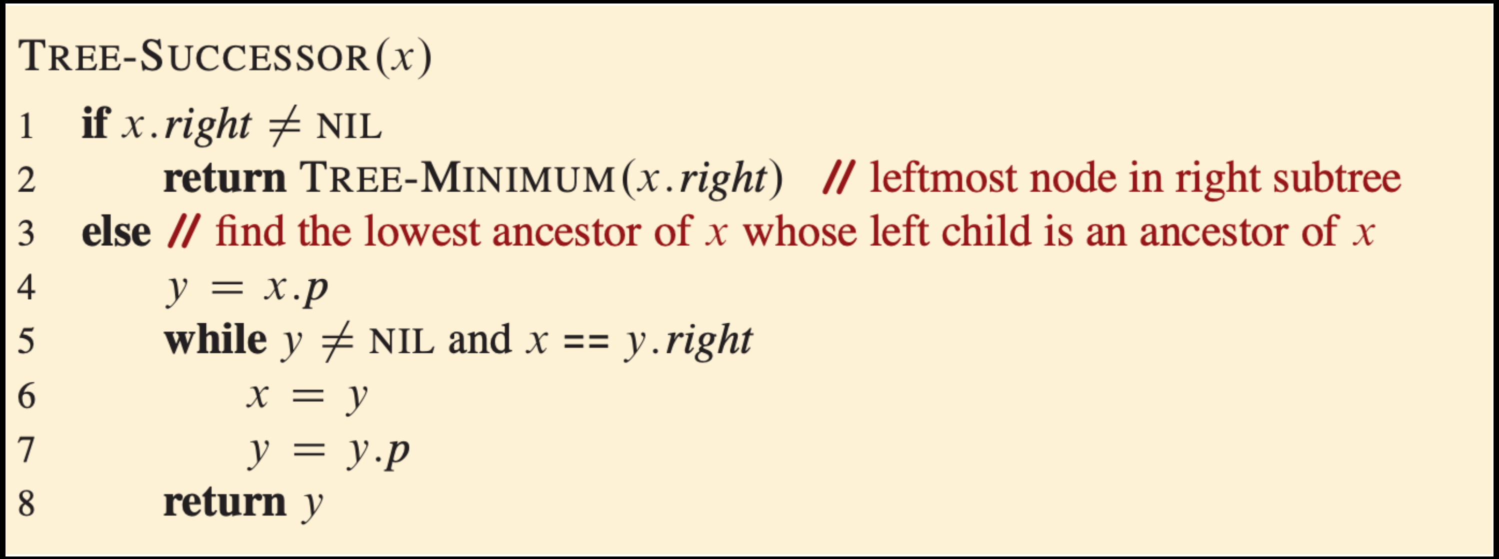
Preorder: root → left → right :: 1, 2, 3, 4, 5

Inorder: left → root → right :: 1, 2, 3, 4, 5

Src: https://samanbatool08.medium.com/trees-binary-search-trees-and-traversal-methods-the-difference-and-why-c52edd53cc31

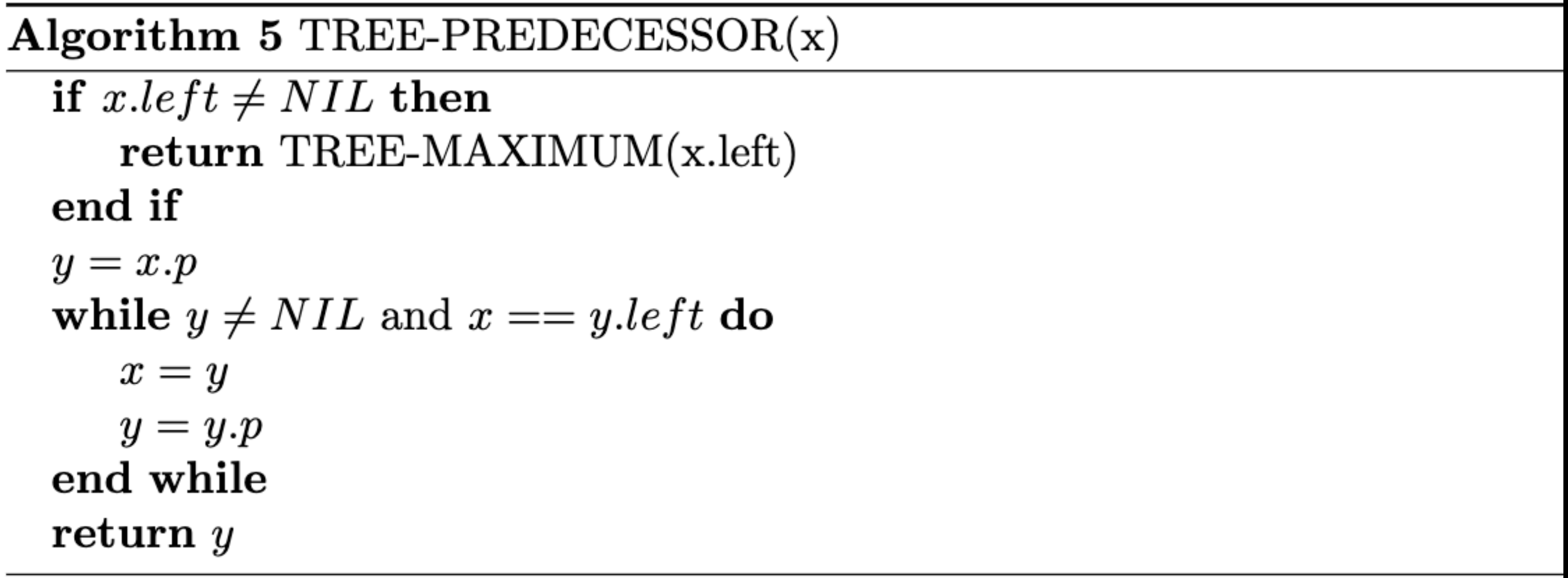
Exercises

- Write the TREE-PREDECESSOR procedure.



```
TREE-SUCCESSOR$(x)$
if $x.right \neq$ NIL
    return TREE-MINIMUM$(x.right)$    // leftmost node in right subtree
else // find the lowest ancestor of $x$ whose left child is an ancestor of $x$
    $y = x.p$
    while $y \neq$ NIL and $x == y.right$
        $x = y$
        $y = y.p$
    return $y$
```

Exercises



Algorithm 5 TREE-PREDECESSOR(x)

```
if $x.left \neq NIL$ then
    return TREE-MAXIMUM(x.left)
end if
$y = x.p$
while $y \neq NIL$ and $x == y.left$ do
    $x = y$
    $y = y.p$
end while
return $y$
```

Exercises

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

Exercises

- Worst case: The tree formed has height n because we were inserting them in already sorted order. This will result in a runtime of $\Theta(n^2)$.
  - Reading elements × Inserting the elements into the tree + tree traversal = $n \times n + n = \Theta(n^2) + \Theta(n) = \Theta(n^2)$

- Best case: The tree formed is approximately balanced. So, its height doesn't exceed O(lg(n)). This will result in a runtime of $O(n \lg(n))$
  - Reading elements × Inserting the elements into the tree + tree traversal = $n \times \lg(n) + n = \Theta(n \lg n) + \Theta(n) = \Theta(n \lg n\ )$

Exercises

You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences cannot be the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363

b. 924, 220, 911, 244, 898, 258, 362, 363

c. 925, 202, 911, 240, 912, 245, 363

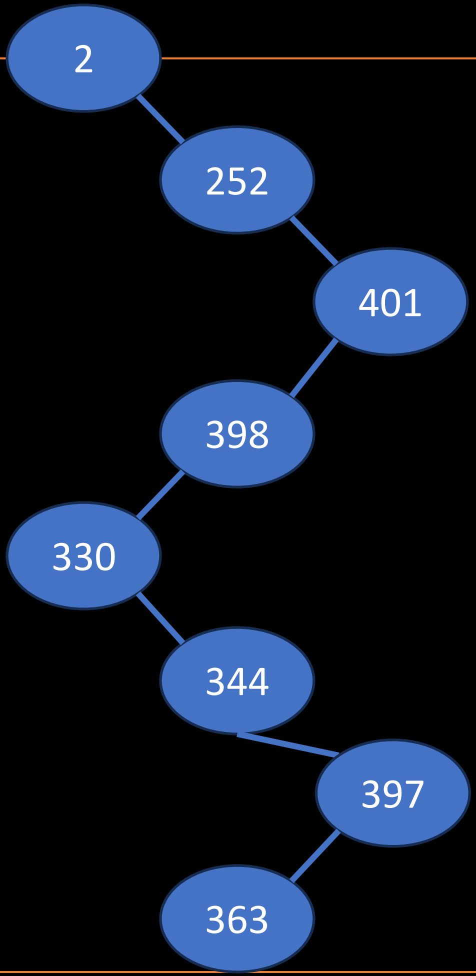
d. 2, 399, 387, 219, 266, 382, 381, 278, 363

e. 935, 278, 347, 621, 299, 392, 358, 363

Exercises

a. 2, 252, 401, 398, 330, 344, 397, 363

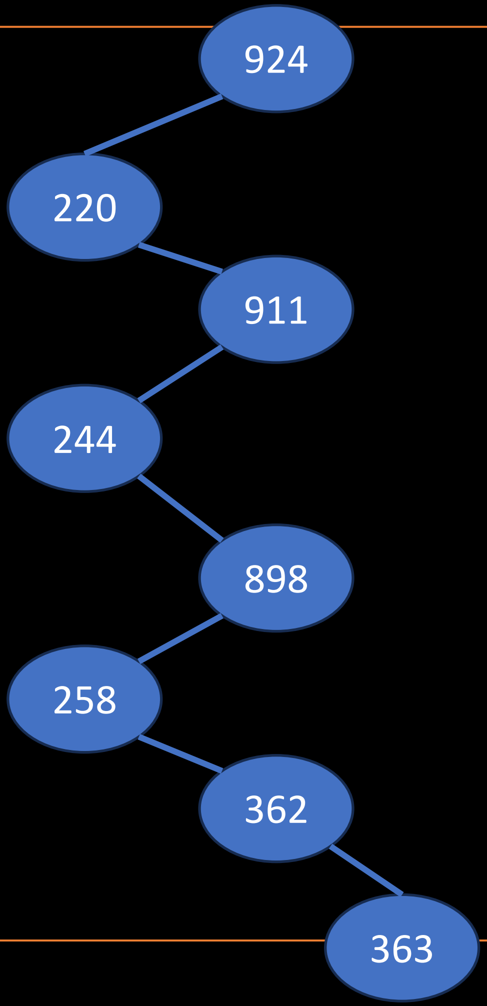
Valid tree, can be queried



Exercises

b. 924, 220, 911, 244, 898, 258, 362, 363

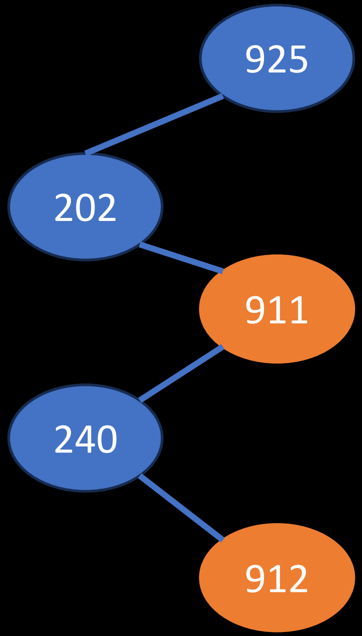
Valid tree, can be queried



Exercises

c. 925, 202, 911, 240, 912, 245, 363

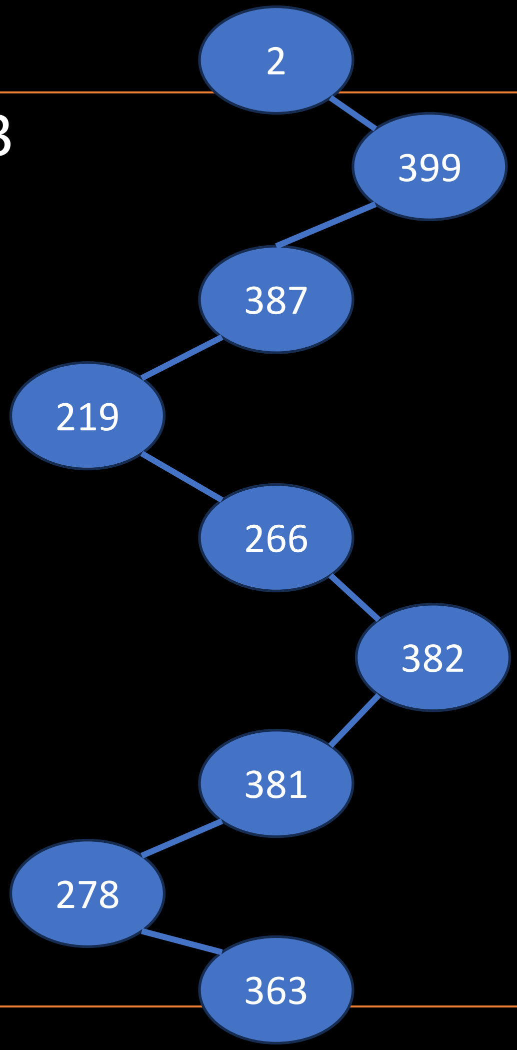
Invalid tree, cannot be queried



Exercises

d. 2, 399, 387, 219, 266, 382, 381, 278, 363

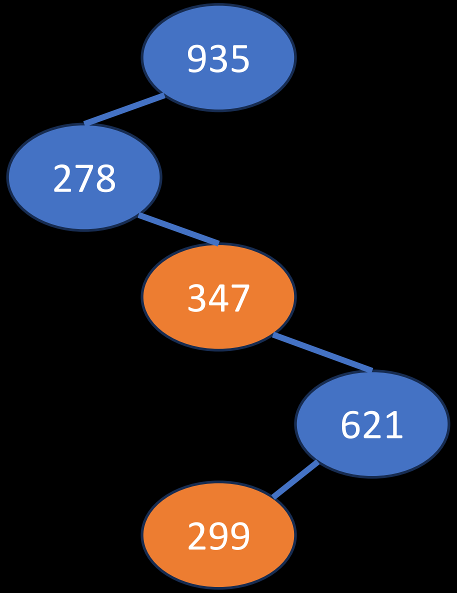
Valid tree, can be queried



Exercises

e. 935, 278, 347, 621, 299, 392, 358, 363

Invalid tree, cannot be queried

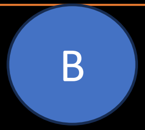


Exercises

Starting with an empty binary search tree, nodes with keys B, R, A, N, C, H, E, and S are are inserted into the tree in that order. Draw the final binary tree.

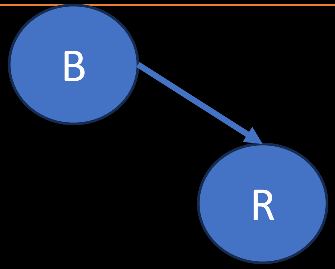
Exercises

B, R, A, N, C, H, E, S



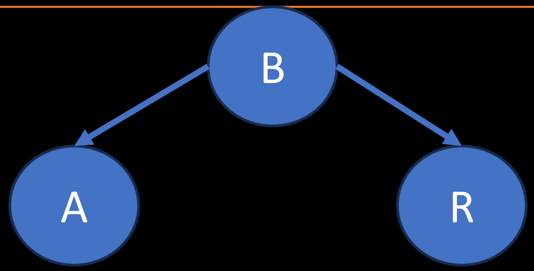
Exercises

B, R, A, N, C, H, E, S



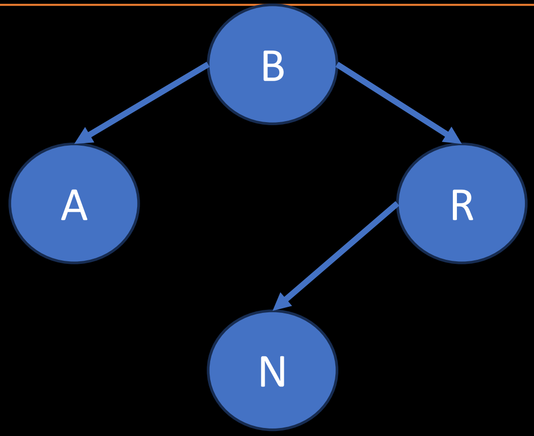
Exercises

B, R, A, N, C, H, E, S



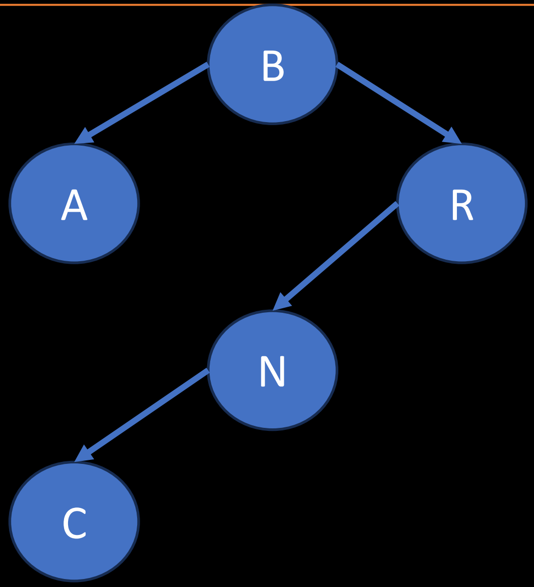
Exercises

B, R, A, N, C, H, E, S



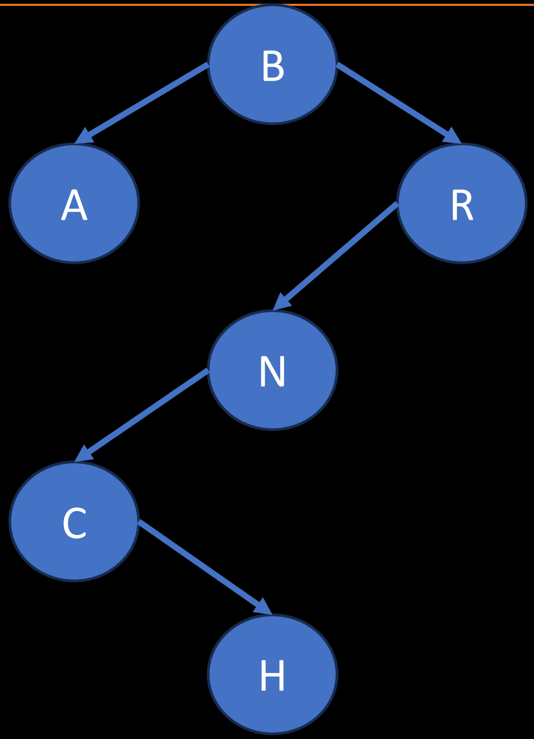
Exercises

B, R, A, N, C, H, E, S



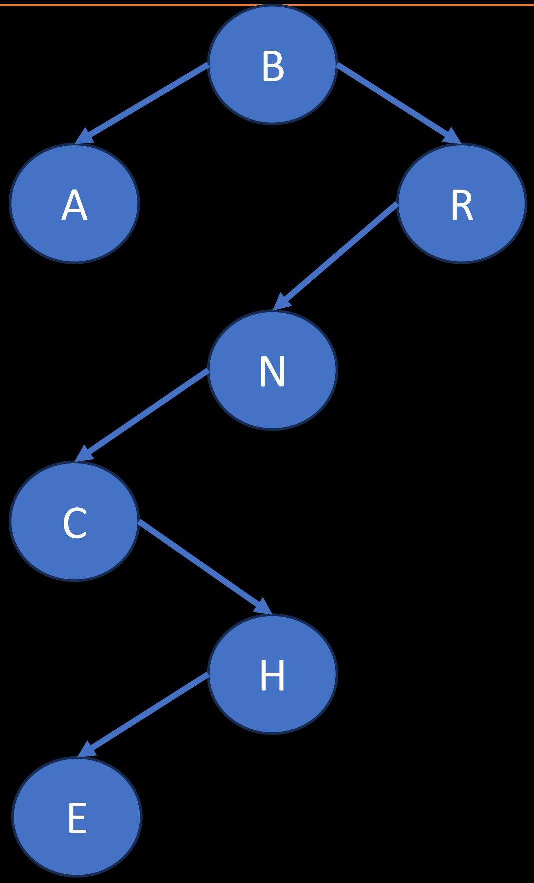
Exercises

B, R, A, N, C, H, E, S



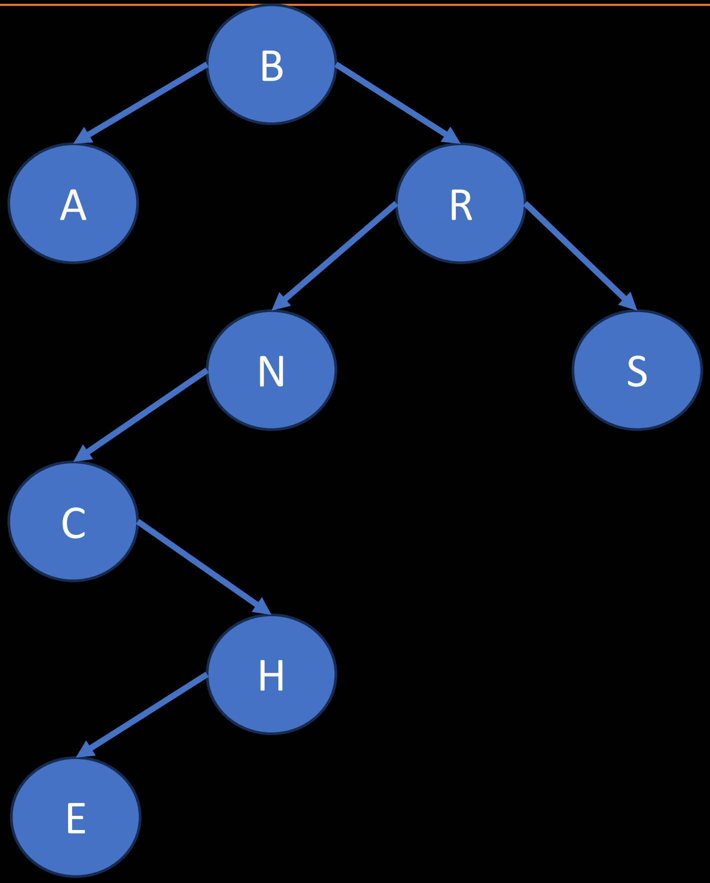
Exercises

B, R, A, N, C, H, <u>E</u>, S



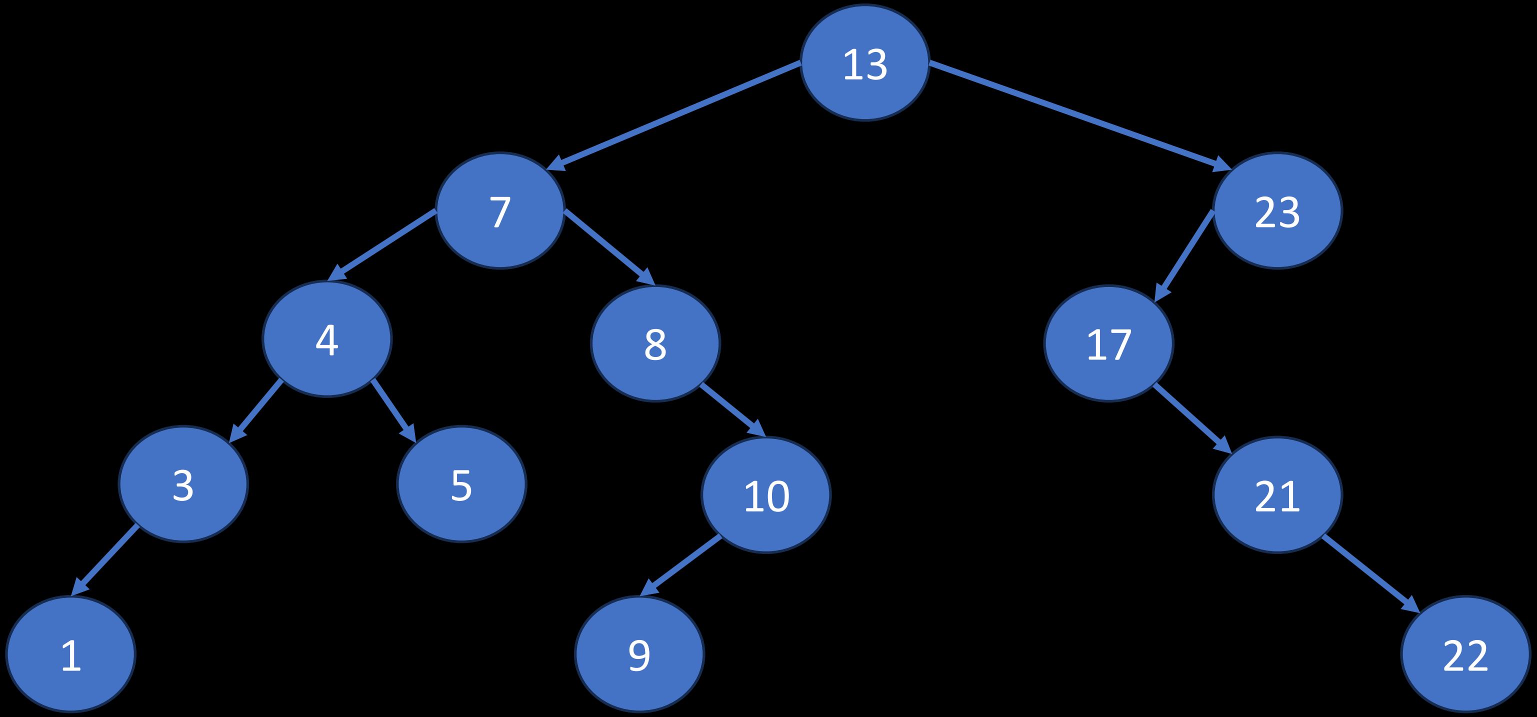
Exercises

B, R, A, N, C, H, E, S

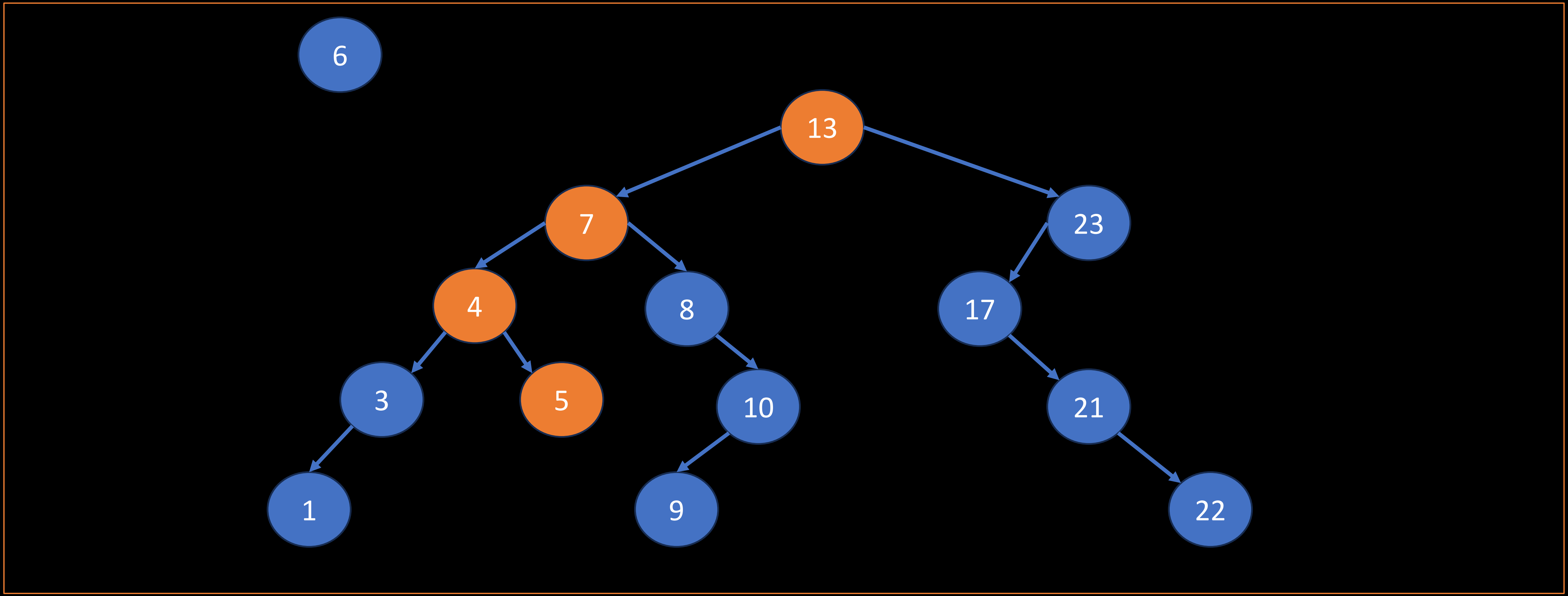


Exercises

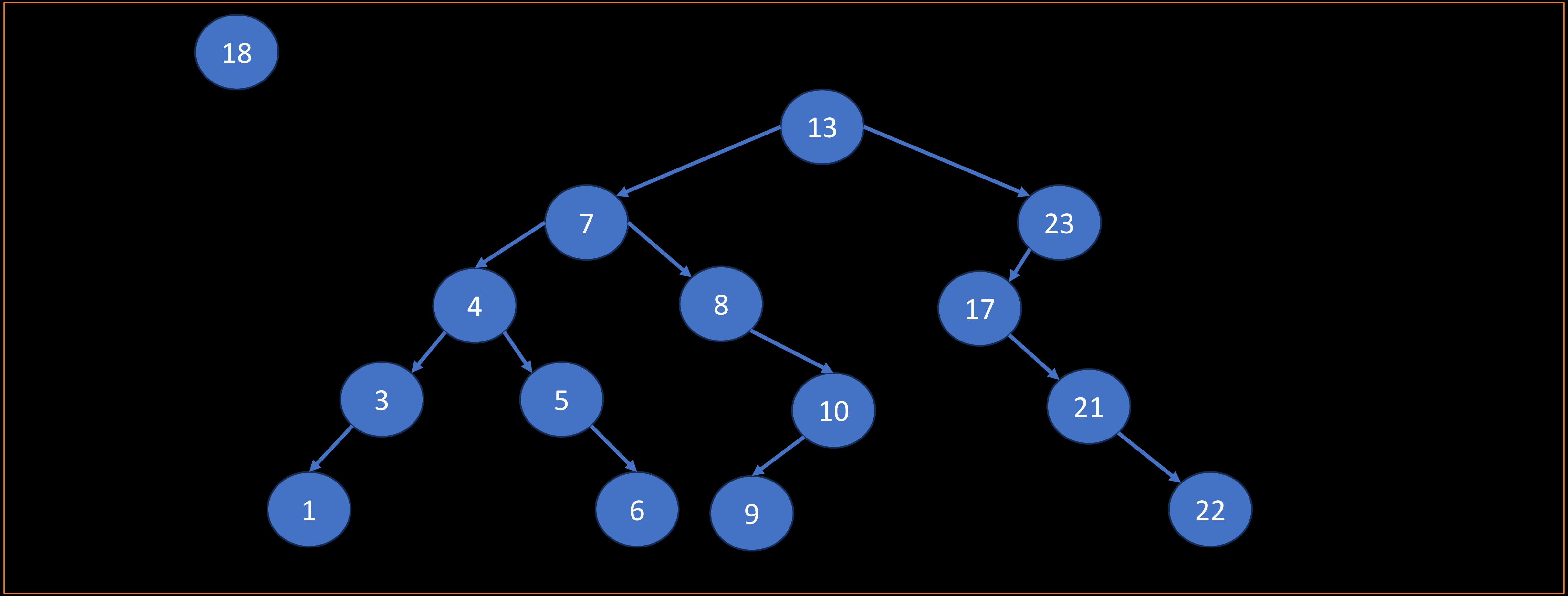
Insert node 6 then 18 in the following tree



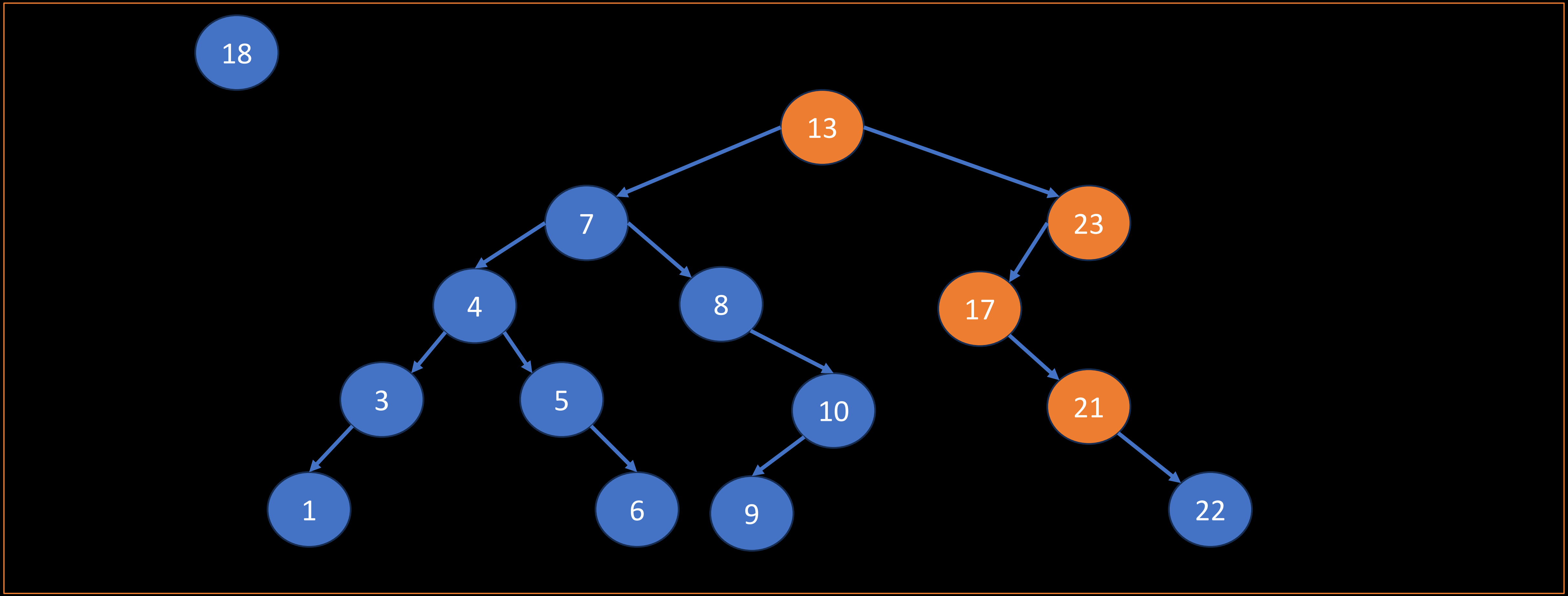
Exercises



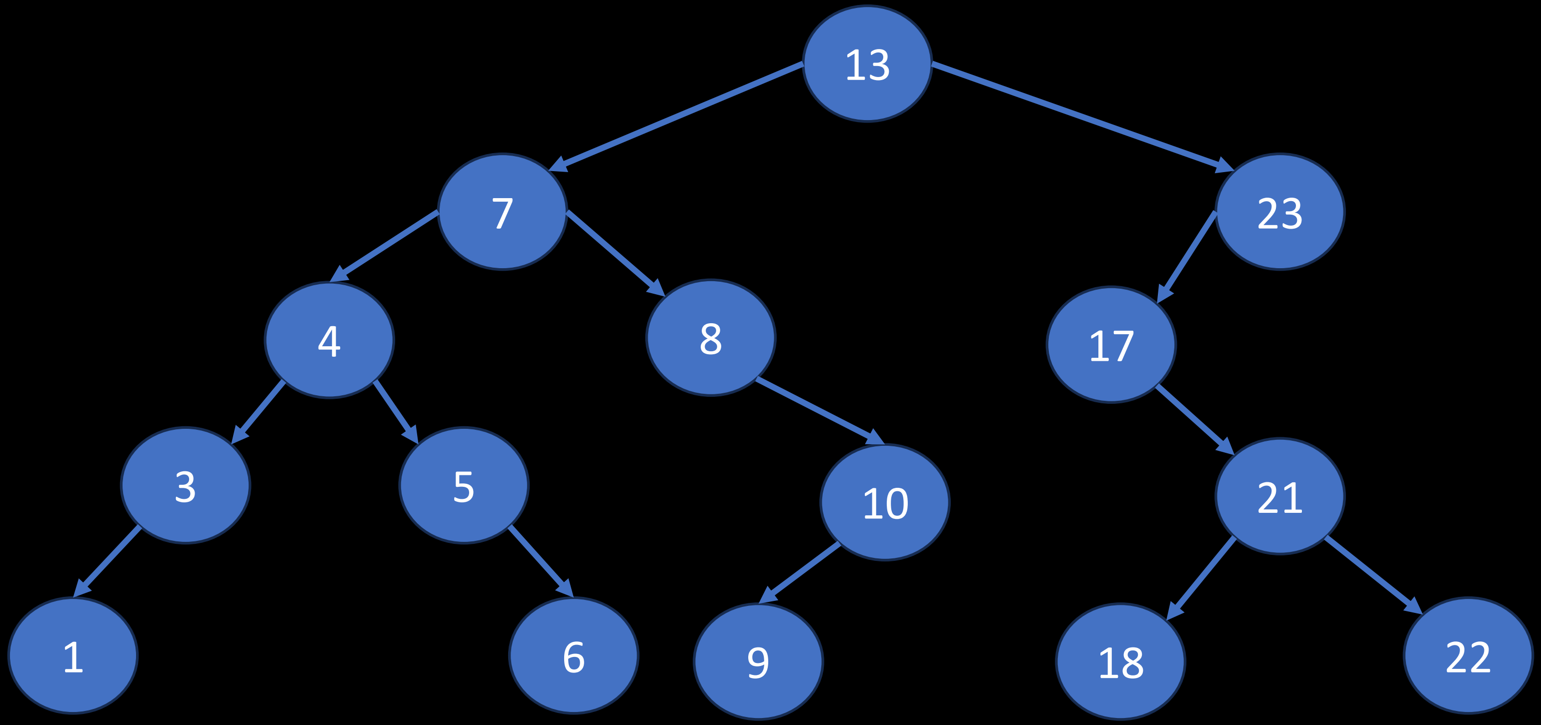
Exercises



Exercises

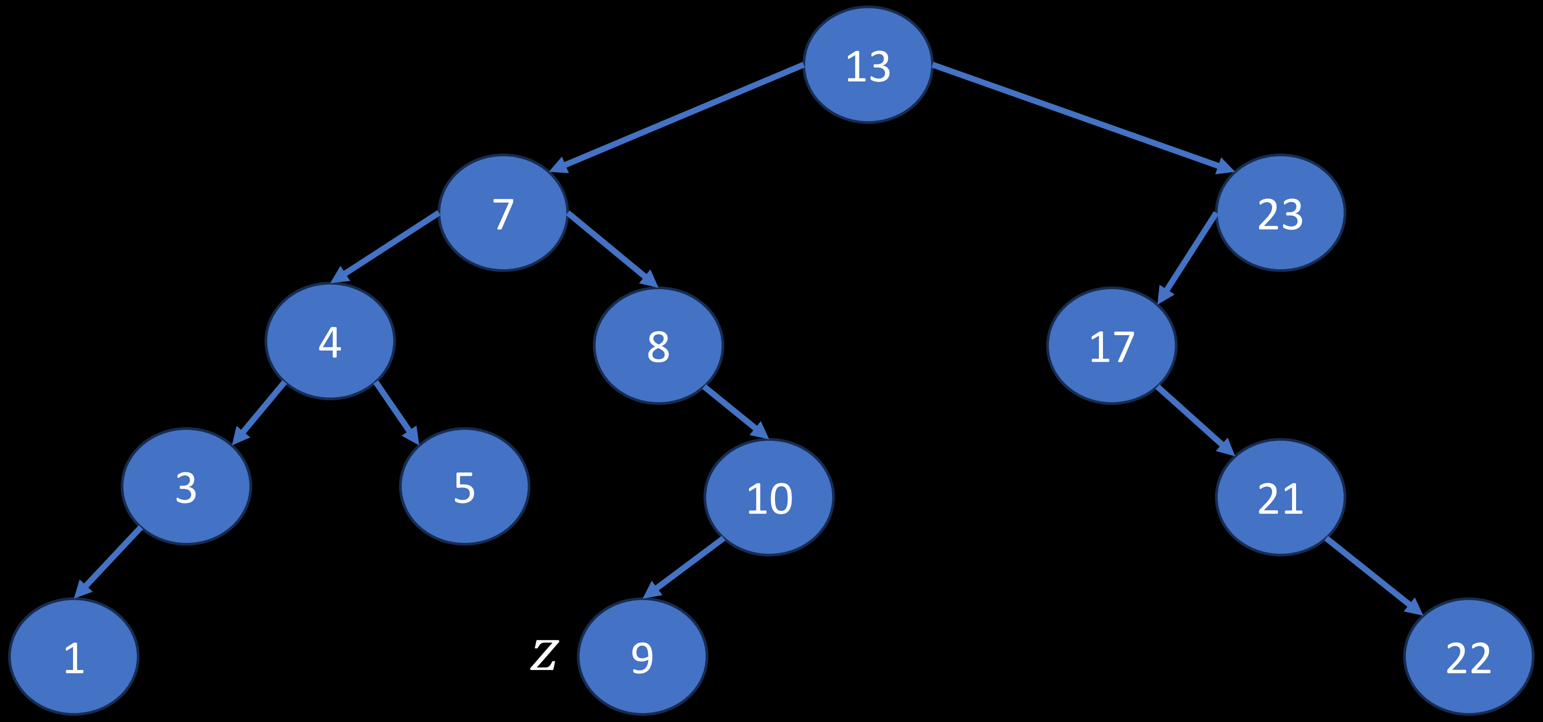


Exercises

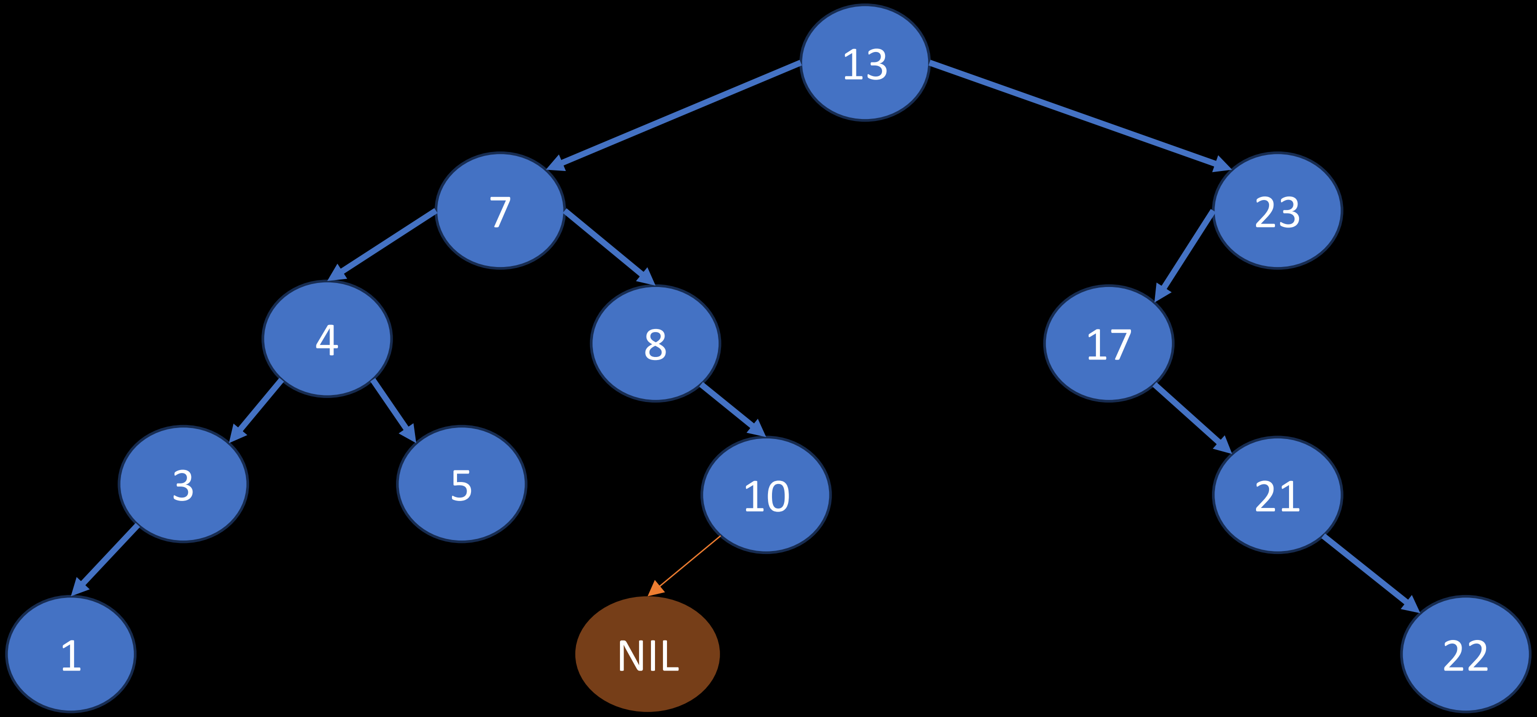


Exercises

Delete node z

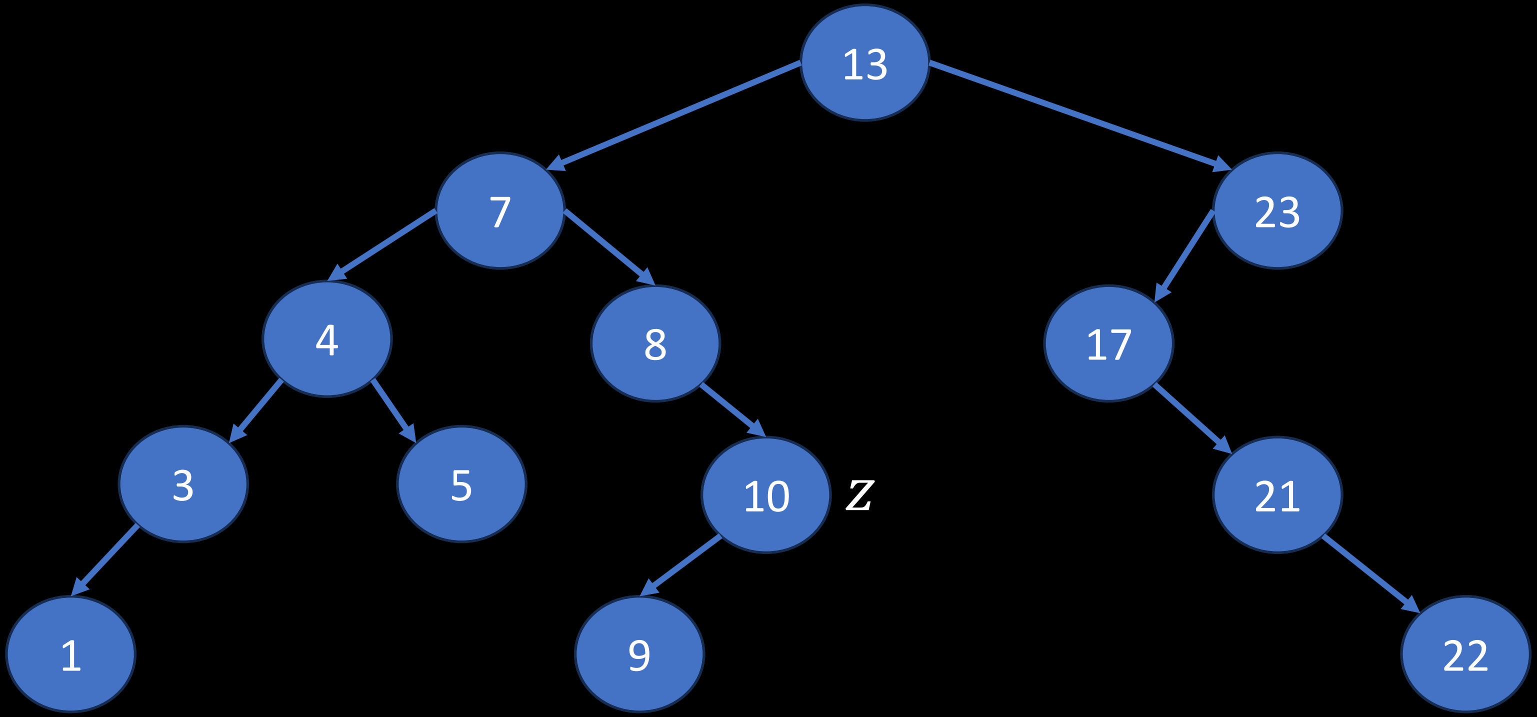


Exercises

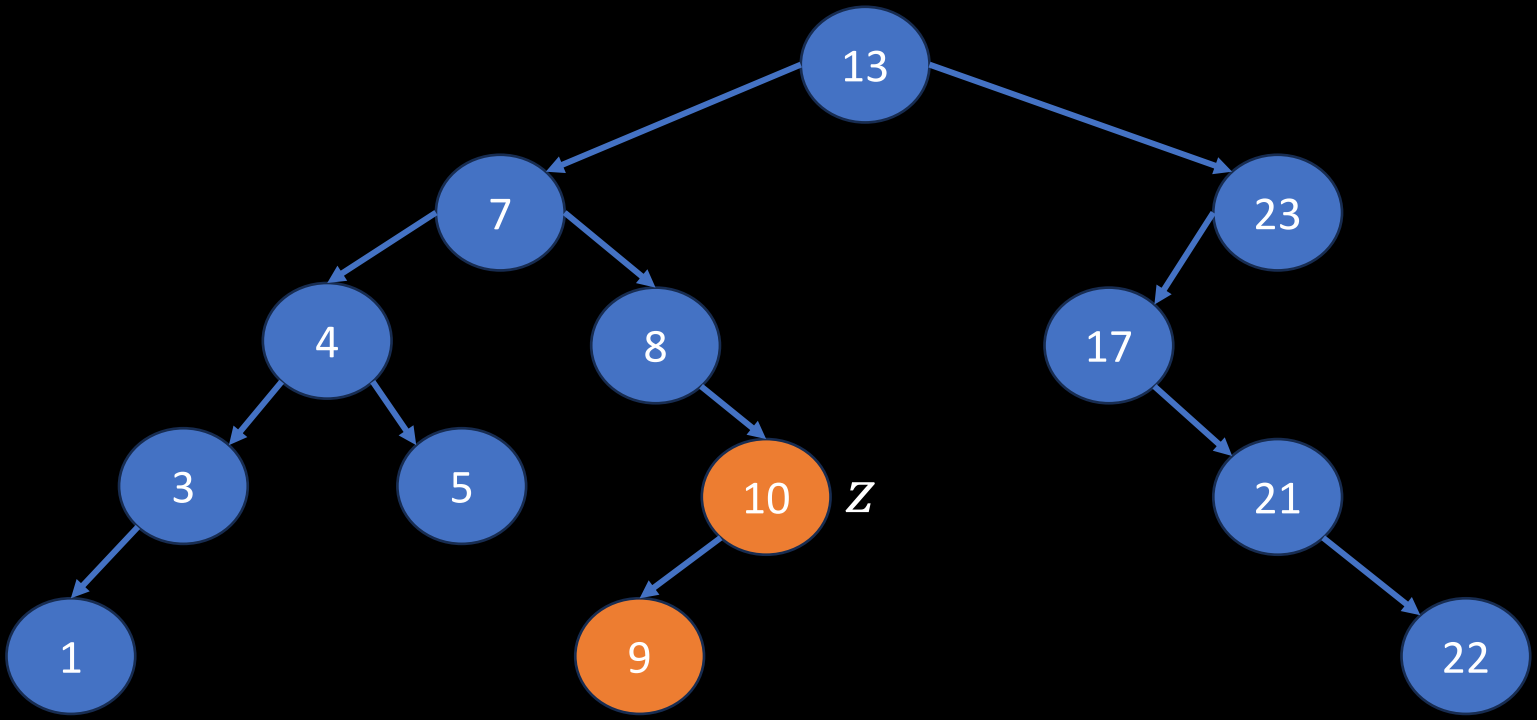


Exercises

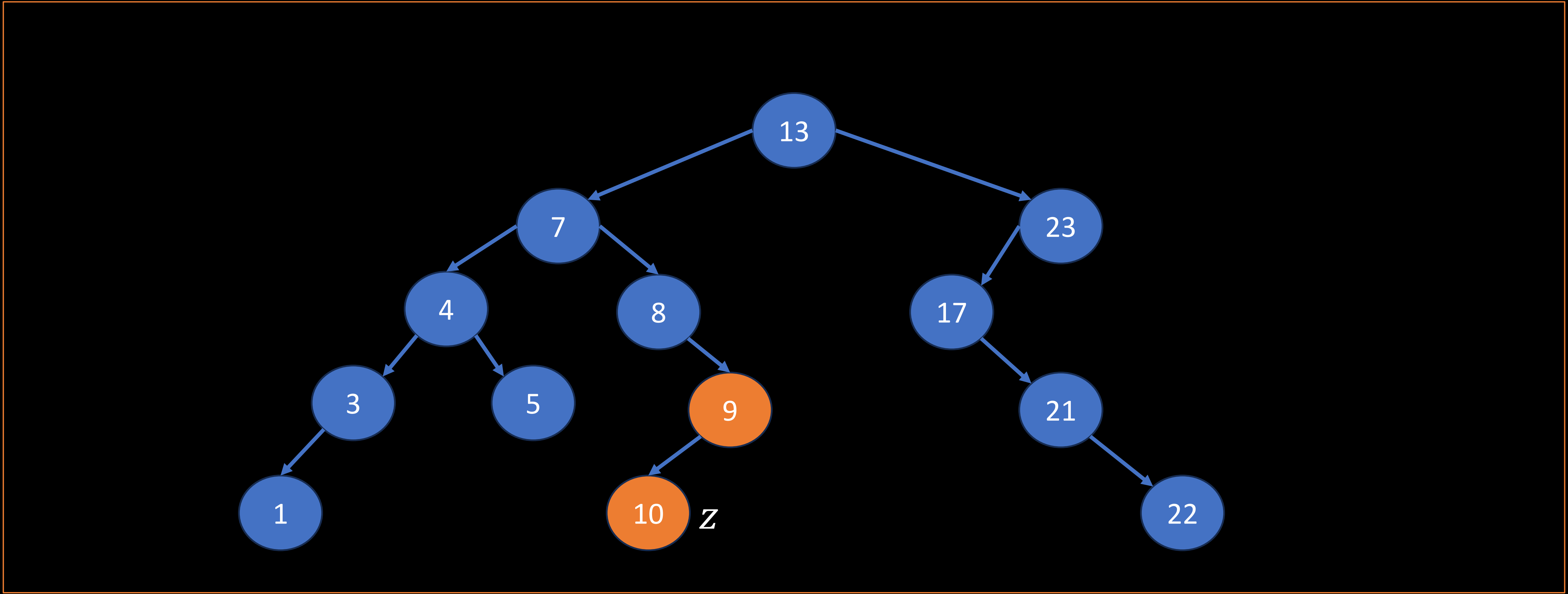
Delete node z



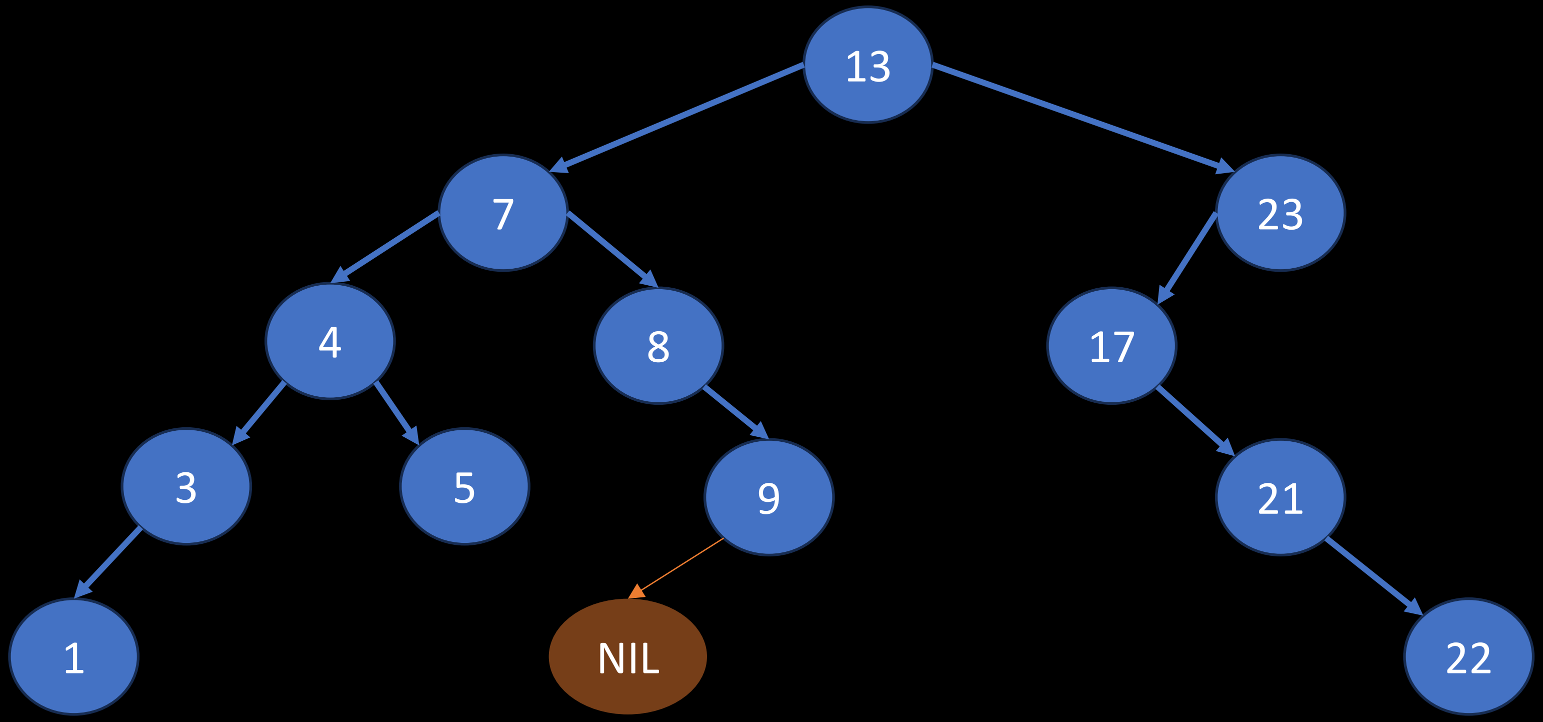
Exercises



Exercises

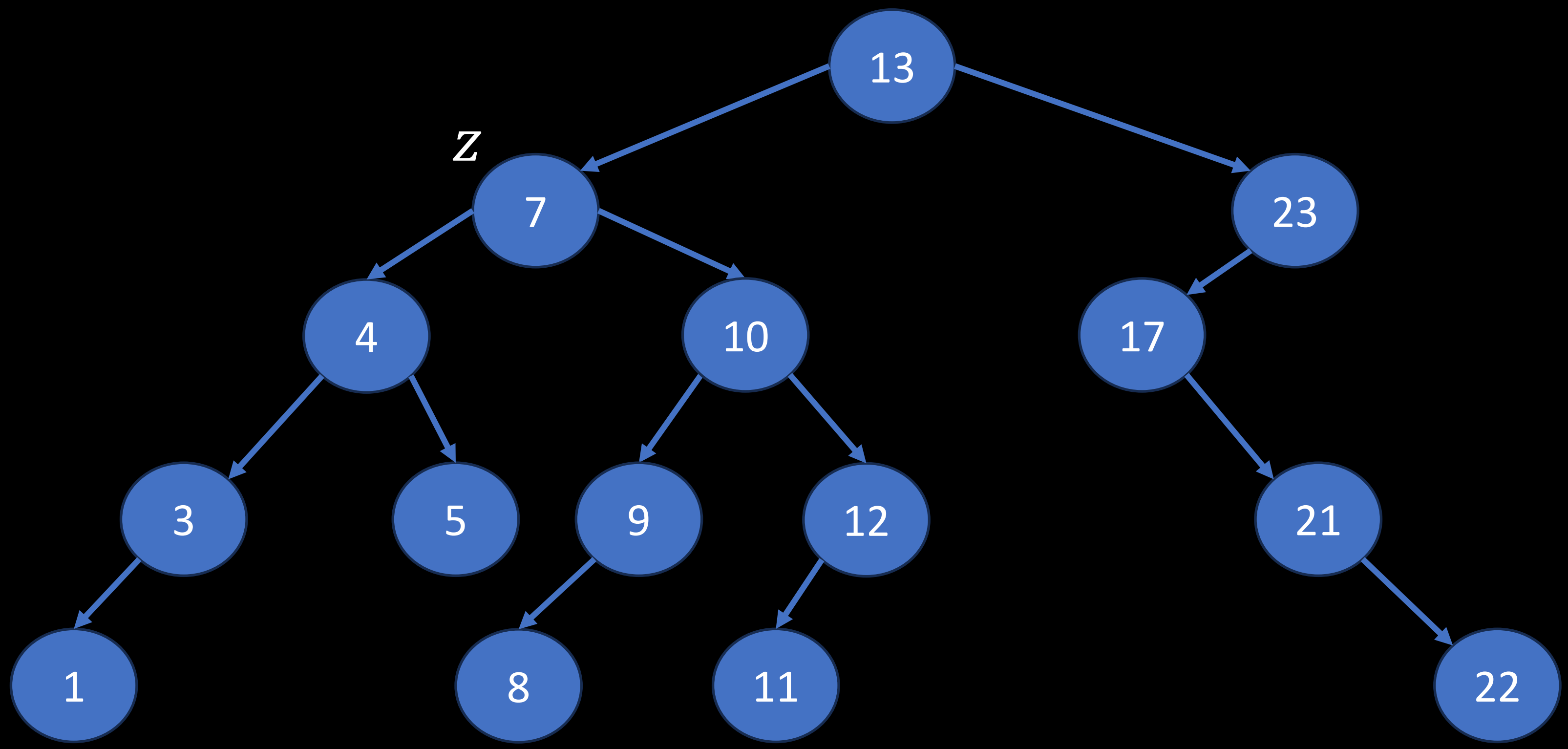


Exercises

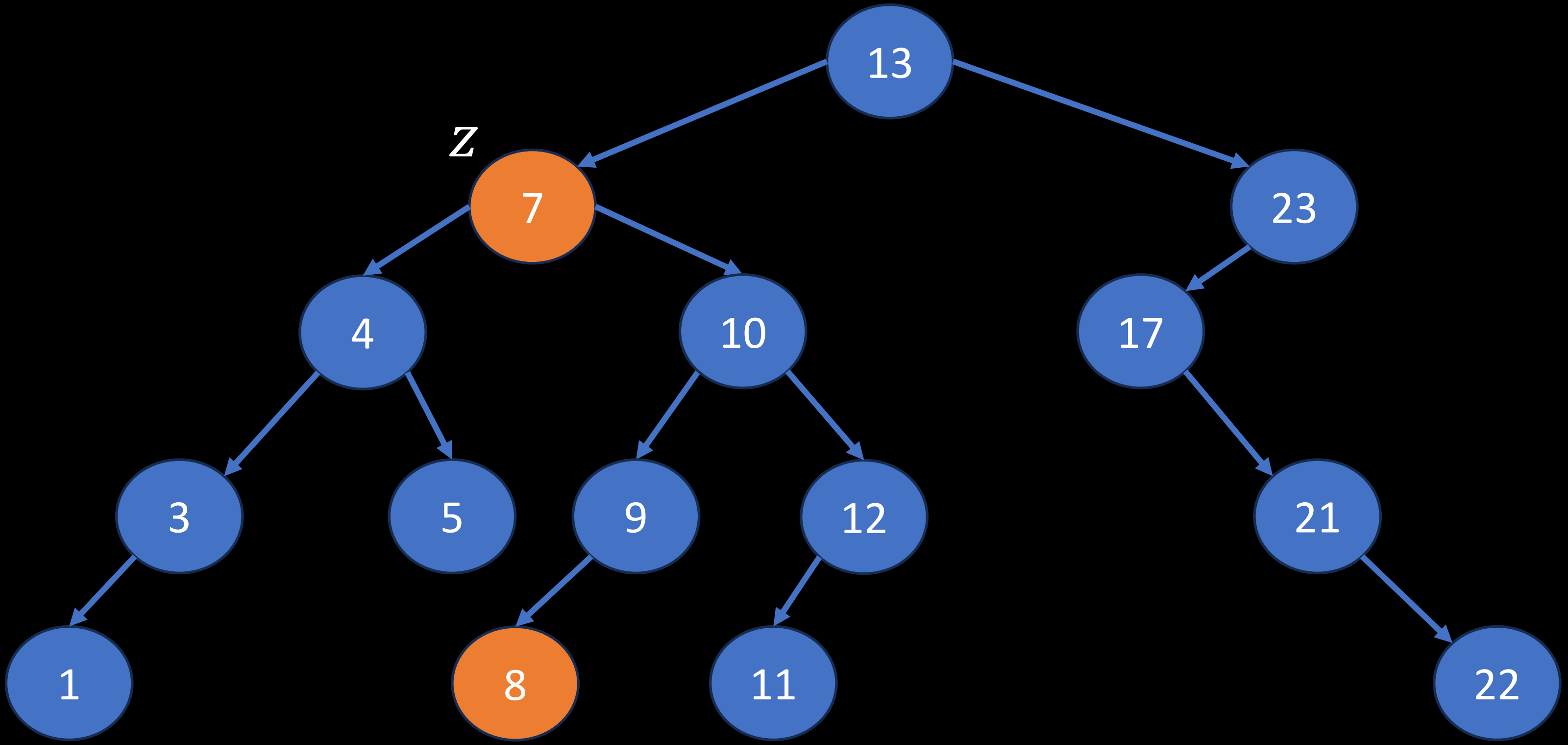


Exercises

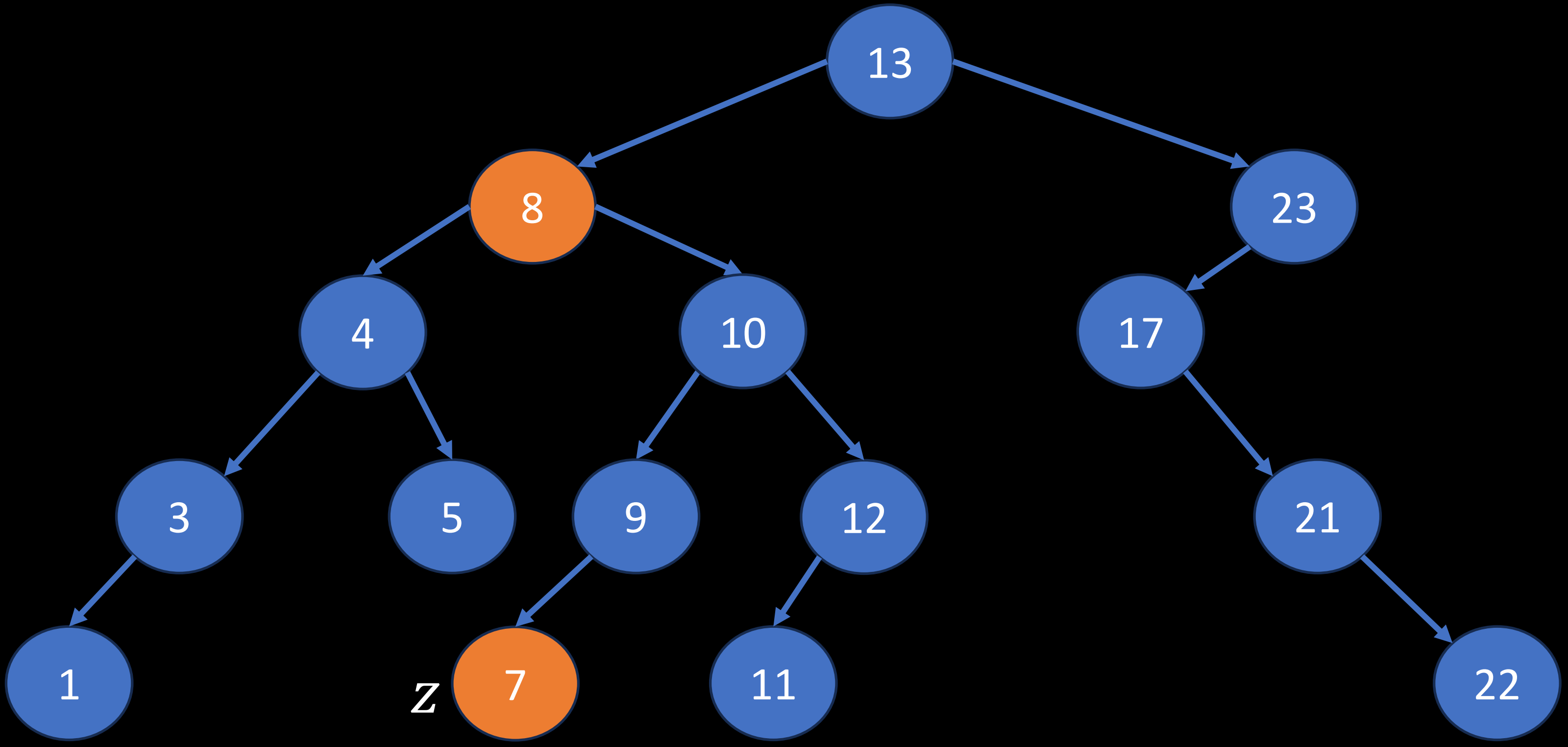
Delete node z



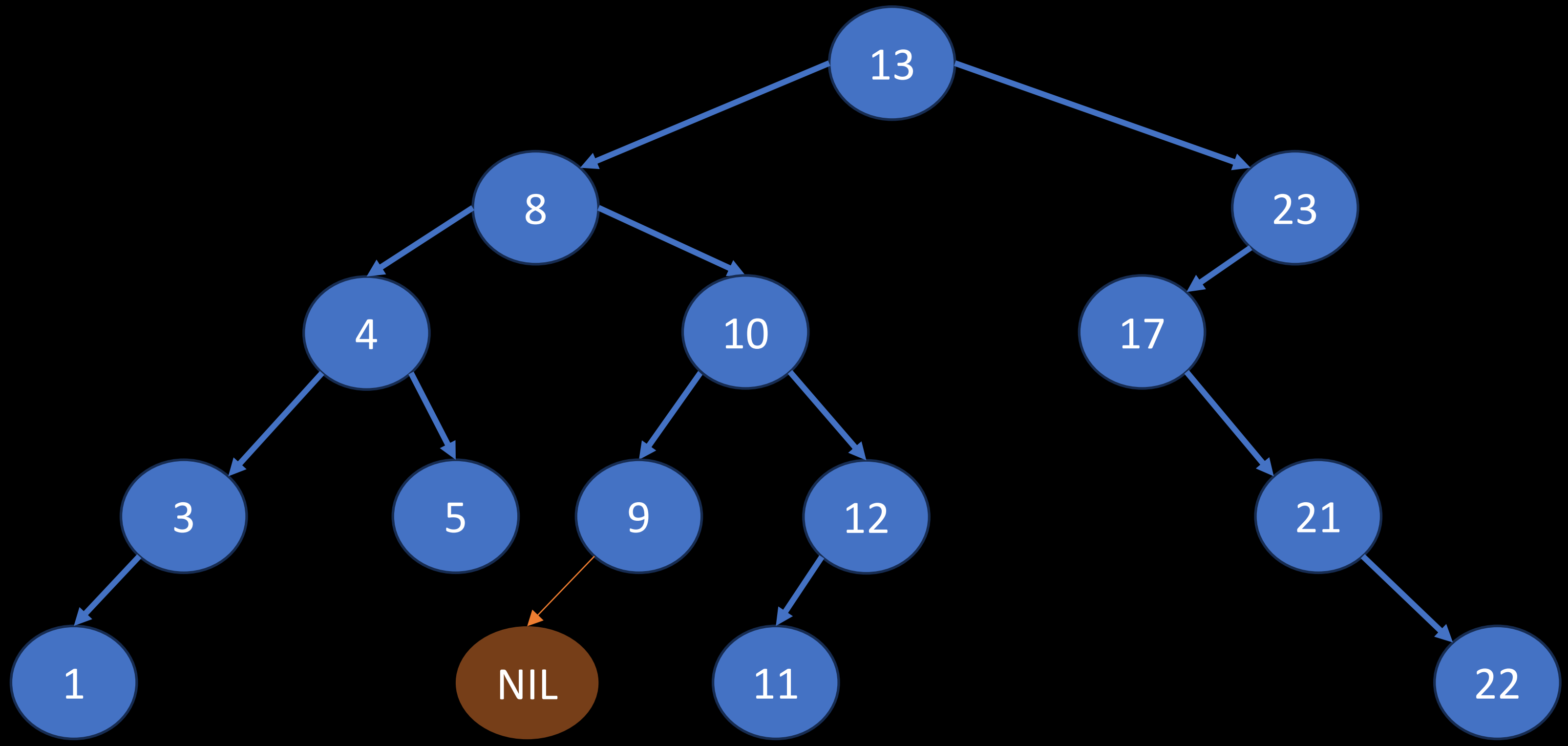
Exercises



Exercises



Exercises


13
8
23
4
10
17
3
5
9
12
21
1
NIL
11
22