Binary Search Trees

Pontus Ekberg

Uppsala University

(Based on previous material by Mohamed Faouzi Atig and Parosh Aziz Abdulla)

Introduction

Binary Search Trees

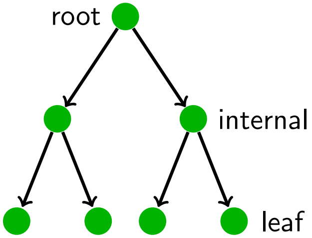
- Data structures that support many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE.

- Can be used as both dictionary and as priority queue

- Basic operations take time proportional to the height of the tree:
  - For complete binary tree with n nodes: worst case $\Theta(log_2(n))$
  - For linear chain with n nodes: worst case $\Theta(n))$

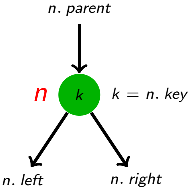
Binary Trees

- A binary tree is a tree such that:
  - Each node has at most two child nodes, distinguished by left and right.
  - The trees we consider here need not be complete binary trees.



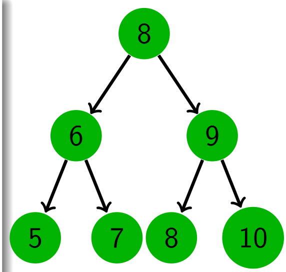
Implementation of Binary Trees

- A binary tree T can be represented as a linked data structure
- Each node n is represented by an object having the following attributes:
  - $n.key$ is the key stored at the node n.
  - ($n.value$ is the value stored at the node n.)
  - $n.left$ points to the left child of n.
  - $n.right$ points to the right child of n.
  - $n.parent$ points to the parent of n.
- If a child or the parent is missing, the appropriate attribute contains the value NIL
- The root node of the tree T is pointed to by the attribute $T.root$.

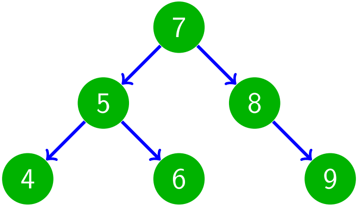


Binary Search Tree

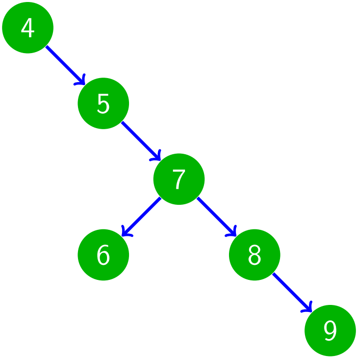
- A binary search tree T is a binary tree such that:
  - For every x and y two nodes of T:
    - If y is in the left subtree of x then $y.key \leq x.key$.
    - If y is in the right subtree of x then $x.key \leq y.key$.



Binary Search Trees: Example (1)



Binary Search Trees: Example (2)



Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- INORDER-TREE-WALK(x): Print out all the keys of the subtree rooted at x in a sorted order.
- SEARCH(x, k): Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- TREE-MINIMUM(x): Return a pointer to the node with smallest key in the subtree of x.
- TREE-MAXIMUM(x): Return a pointer to the node with largest key in the subtree of x.
- TREE-SUCCESSOR(x): Return a pointer to the node with the smallest key larger than $x.key$.
- TREE-INSERT(T, x): Insert x in T such that the binary search property is preserved.
- TREE-DELETE(T, x): Delete x from T such that the binary search property is preserved.

TREE-WALK

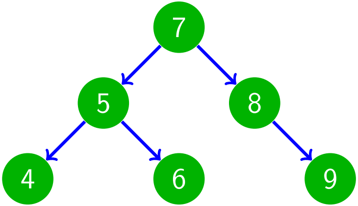
INORDER-TREE-WALK(x): Print out all the keys of the tree rooted at x in a sorted order.

INORDER-TREE-WALK(x)

```
if x ≠ NIL
   then INORDER-TREE-WALK(x. left)
        print x. key
        INORDER-TREE-WALK(x. right)
```

INORDER-TREEE-WALK: Principle

- Check whether the node x is not *NIL*
- Recursively, print the keys of the nodes in the left subtree of x
- Print the key of x
- Recursively, print the keys of the nodes in the right subtree of x



TREE-WALK

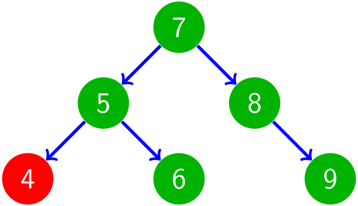
INORDER-TREE-WALK(x): Print out all the keys of the tree rooted at x in a sorted order.

INORDER-TREE-WALK(x)

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

INORDER-TREEE-WALK: Principle

- Check whether the node x is not NIL
- Recursively, print the keys of the nodes in the left subtree of x
- Print the key of x
- Recursively, print the keys of the nodes in the right subtree of x



4

TREE-WALK

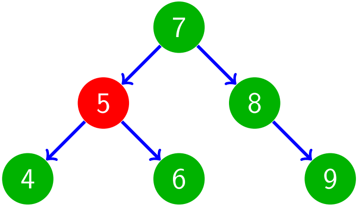
INORDER-TREE-WALK(x): Print out all the keys of the tree rooted at x in a sorted order.

INORDER-TREE-WALK(x)

```
if x ≠ NIL
    then INORDER-TREE-WALK(x.left)
         print x.key
         INORDER-TREE-WALK(x.right)
```

INORDER-TREEE-WALK: Principle

- Check whether the node x is not *NIL*
- Recursively, print the keys of the nodes in the left subtree of x
- Print the key of x
- Recursively, print the keys of the nodes in the right subtree of x



4 5

TREE-WALK

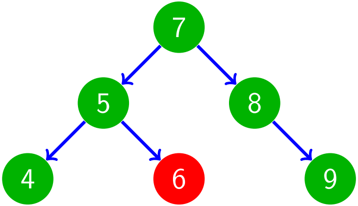
INORDER-TREE-WALK(x): Print out all the keys of the tree rooted at x in a sorted order.

INORDER-TREE-WALK(x)

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

INORDER-TREEE-WALK: Principle

- Check whether the node x is not NIL
- Recursively, print the keys of the nodes in the left subtree of x
- Print the key of x
- Recursively, print the keys of the nodes in the right subtree of x



4 5 6

TREE-WALK

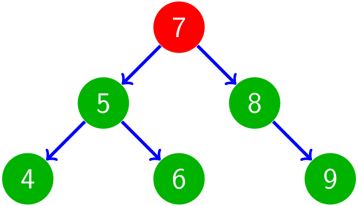
INORDER-TREE-WALK(x): Print out all the keys of the tree rooted at x in a sorted order.

INORDER-TREE-WALK(x)

```
if x ≠ NIL
   then INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)
```

INORDER-TREEE-WALK: Principle

- Check whether the node x is not *NIL*
- Recursively, print the keys of the nodes in the left subtree of x
- Print the key of x
- Recursively, print the keys of the nodes in the right subtree of x



4 5 6 7

TREE-WALK

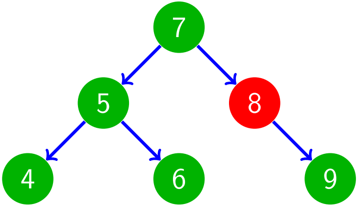
INORDER-TREE-WALK(x): Print out all the keys of the tree rooted at x in a sorted order.

INORDER-TREE-WALK(x)

```
if x ≠ NIL
   then INORDER-TREE-WALK(x. left)
        print x. key
        INORDER-TREE-WALK(x. right)
```

INORDER-TREEE-WALK: Principle

- Check whether the node x is not *NIL*
- Recursively, print the keys of the nodes in the left subtree of x
- Print the key of x
- Recursively, print the keys of the nodes in the right subtree of x



4 5 6 7 8

TREE-WALK

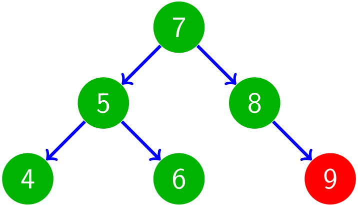
INORDER-TREE-WALK(x): Print out all the keys of the tree rooted at x in a sorted order.

INORDER-TREE-WALK(x)

```
if x ≠ NIL
   then INORDER-TREE-WALK(x. left)
        print x. key
        INORDER-TREE-WALK(x. right)
```

INORDER-TREEE-WALK: Principle

- Check whether the node x is not *NIL*
- Recursively, print the keys of the nodes in the left subtree of x
- Print the key of x
- Recursively, print the keys of the nodes in the right subtree of x



4 5 6 7 8 9

TREE-WALK

- Let n be the number of nodes in the subtree rooted at x
- Let $T(n)$ be the time taken by INORDER-TREE-WALK(x)
- Each of lines 1 and 3 takes constant time.
- Let k be the number of nodes of the left subtree of x then we have:

INORDER-TREE-WALK(x)

```
1  if x ≠ NIL
2     then INORDER-TREE-WALK(x. left)
3          print x. key
4          INORDER-TREE-WALK(x. right)
```

$$T(n) = T(k) + T(n - k - 1) + \Theta(1)$$

TREE-WALK

- Let n be the number of nodes in the subtree rooted at x
- Let $T(n)$ be the time taken by INORDER-TREE-WALK(x)
- Each of lines 1 and 3 takes constant time.
- Let k be the number of nodes of the left subtree of x then we have:

$$T(n) = T(k) + T(n-k-1) + \Theta(1)$$

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(x. left)
3          print x. key
4          INORDER-TREE-WALK(x. right)
```

$$T(n) = \Theta(n)$$

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- INORDER-TREE-WALK(x): Print out all the keys of the subtree rooted at x in a sorted order.
- SEARCH(x, k): Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return *NIL*
- TREE-MINIMUM(x): Return a pointer to the node with smallest key in the subtree of x.
- TREE-MAXIMUM(x): Return a pointer to the node with largest key in the subtree of x.
- TREE-SUCCESSOR(x): Return a pointer to the node with the smallest key larger than $x.key$.
- TREE-INSERT(T, x): Insert x in T such that the binary search property is preserved.
- TREE-DELETE(T, x): Delete x from T such that the binary search property is preserved.

TREE-SEARCH

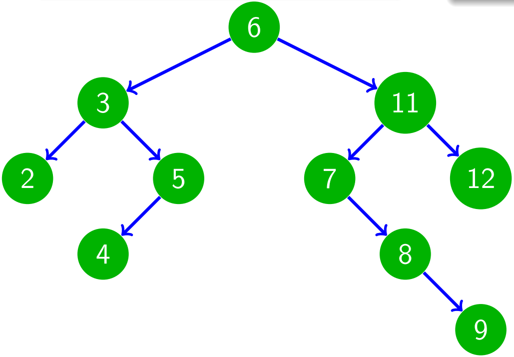
TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH: Principle

For each node x it encounters, it compares the key k and $x.key$:

- If the two keys are equal, the search terminates.
- If k is smaller than $x.key$, the search continues in the left subtree of x
- If k is larger than $x.key$, the search continues in the right subtree of x



TREE-SEARCH($T.root$, 4)

TREE-SEARCH

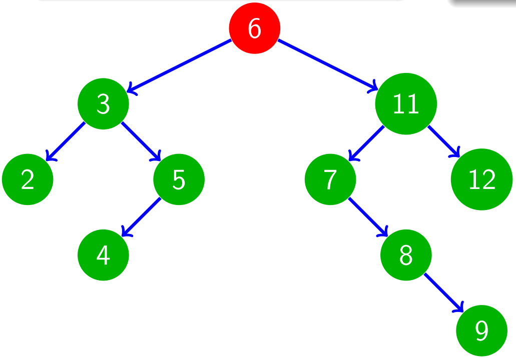
TREE-SEARCH(x, k)

if x = NIL or $k = x.\,key$
    then return x
if $k \leq x.\,key$
    then return TREE-SEARCH($x.left$, k)
    else return TREE-SEARCH($x.right$, k)

TREE-SEARCH: Principle

For each node x it encounters, it compares the key k and $x.\,key$:

- If the two keys are equal, the search terminates.
- If k is smaller than $x.\,key$, the search continues in the left subtree of x
- If k is larger than $x.\,key$, the search continues in the right subtree of x



TREE-SEARCH($T.\,root$, 4)

TREE-SEARCH

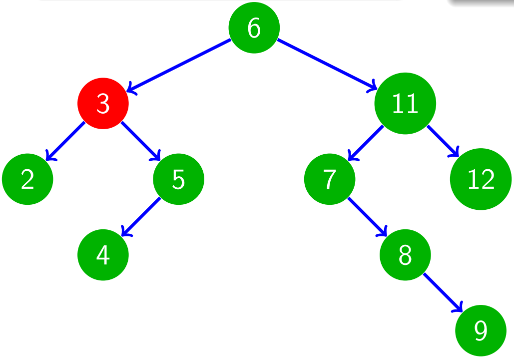
TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH: Principle

For each node x it encounters, it compares the key k and $x.key$:

- If the two keys are equal, the search terminates.
- If k is smaller than $x.key$, the search continues in the left subtree of x
- If k is larger than $x.key$, the search continues in the right subtree of x



TREE-SEARCH($T.root.left$, 4)

TREE-SEARCH

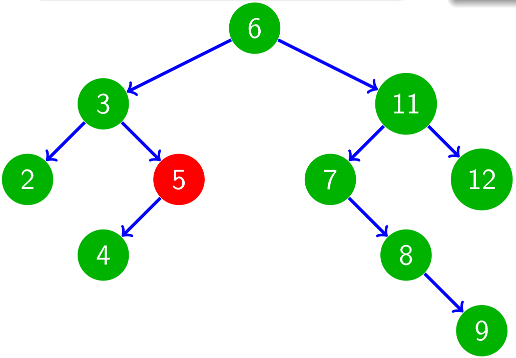
TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH: Principle

For each node x it encounters, it compares the key k and $x.key$:

- If the two keys are equal, the search terminates.
- If k is smaller than $x.key$, the search continues in the left subtree of x
- If k is larger than $x.key$, the search continues in the right subtree of x



TREE-SEARCH($T.root.left.right$, 4)

TREE-SEARCH

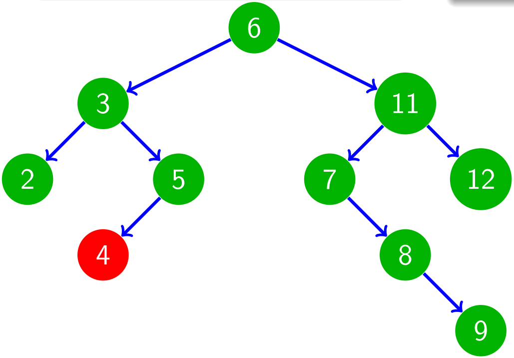
TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH: Principle

For each node x it encounters, it compares the key k and $x.key$:

- If the two keys are equal, the search terminates.
- If k is smaller than $x.key$, the search continues in the left subtree of x
- If k is larger than $x.key$, the search continues in the right subtree of x



TREE-SEARCH($T.root.left.right.left$, 4)

TREE-SEARCH

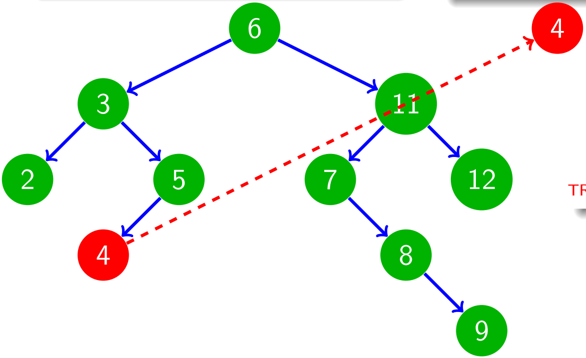
TREE-SEARCH(x, k)

if $x = NIL$ or $k = x.\,key$
    then return x
if $k \leq x.\,key$
    then return TREE-SEARCH($x.left$, k)
    else return TREE-SEARCH($x.right$, k)

TREE-SEARCH: Principle

For each node x it encounters, it compares the key k and $x.\,key$:

- If the two keys are equal, the search terminates.
- If k is smaller than $x.\,key$, the search continues in the left subtree of x
- If k is larger than $x.\,key$, the search continues in the right subtree of x

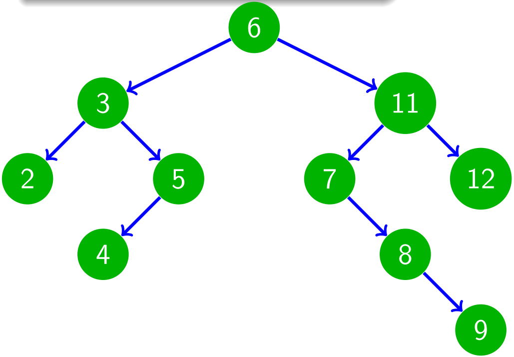


TREE-SEARCH($T.\,root.\,left.\,right.\,left$, 4)

Binary Search Trees

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH($T.\mathit{root}$, 10)

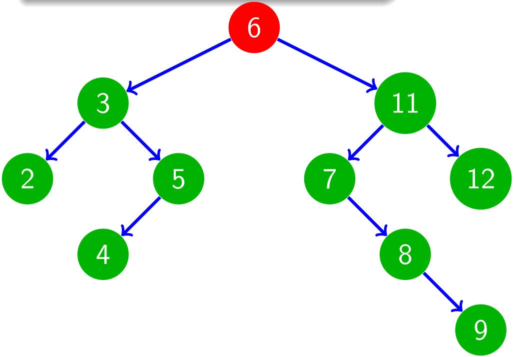


Binary Search Trees

TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH($T.root$, 10)

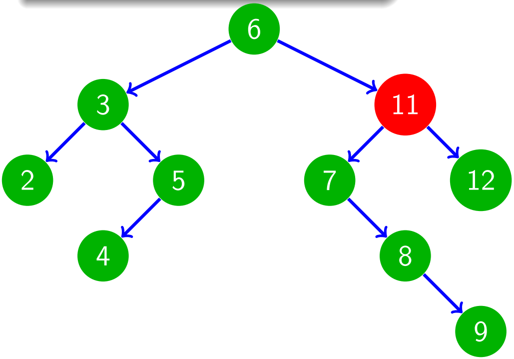


Binary Search Trees

TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH($T.root.right$, 10)

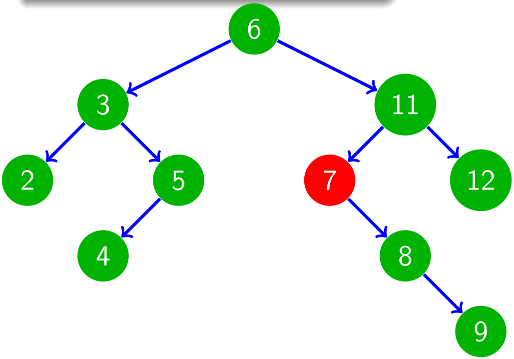


Binary Search Trees

TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH$(T.root.right.left, 10)$

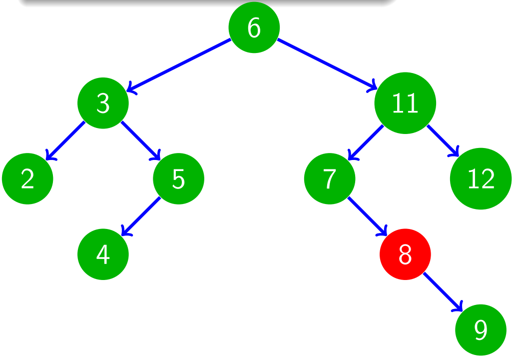


Binary Search Trees

TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH($T.root.right.left.right$, 10)

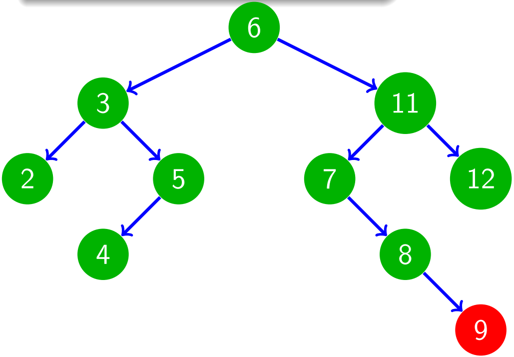


Binary Search Trees

TREE-SEARCH(x, k)

```
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

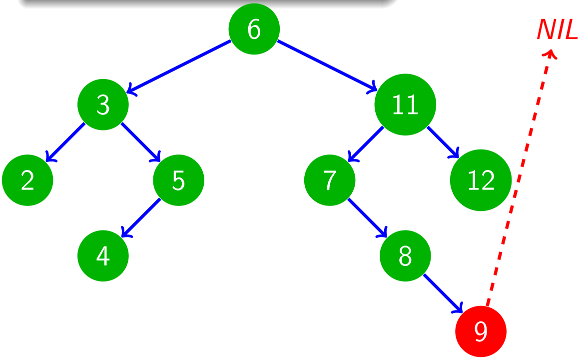
TREE-SEARCH($T.root.right.left.right.right$, 10)



Binary Search Trees

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

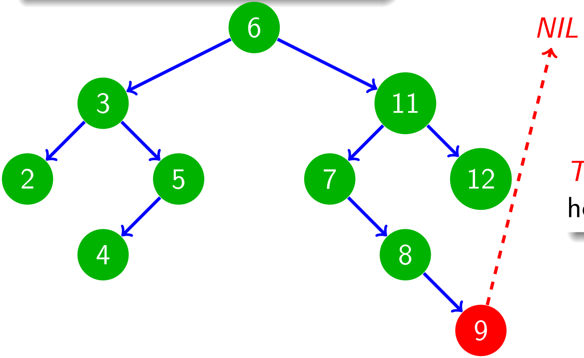
TREE-SEARCH($T.root.right.left.right.right$, 10)



Binary Search Trees

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
    then return x
if k ≤ x.key
    then return TREE-SEARCH(x.left, k)
    else  return TREE-SEARCH(x.right, k)
```

TREE-SEARCH($T.root.right.left.right.right$, 10)



$T(n) = O(h)$ where h is the height of the tree

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- INORDER-TREE-WALK(x): Print out all the keys of the subtree rooted at x in a sorted order.
- SEARCH(x, k): Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- TREE-MINIMUM(x): Return a pointer to the node with smallest key in the subtree of x.
- TREE-MAXIMUM(x): Return a pointer to the node with largest key in the subtree of x.
- TREE-SUCCESSOR(x): Return a pointer to the node with the smallest key larger than $x.key$.
- TREE-INSERT(T, x): Insert x in T such that the binary search property is preserved.
- TREE-DELETE(T, x): Delete x from T such that the binary search property is preserved.

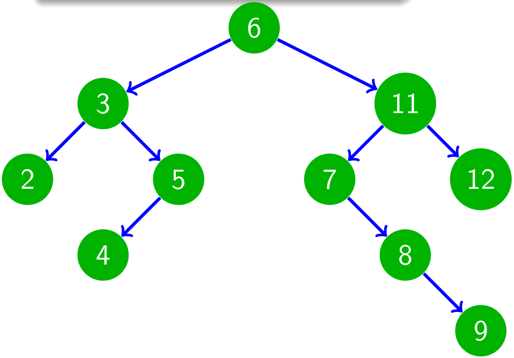
TREE-MINIMUM

TREE-MINIMUM(x)

```
while x.left ≠ NIL
    do x ⟵ x.left
return x
```

TREE-MINIMUM: Principle

The minimum key of a binary search tree is located at the leftmost node



TREE-MINIMUM($T.root$)

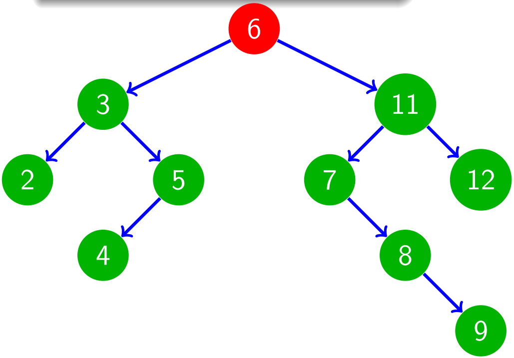
TREE-MINIMUM

TREE-MINIMUM(x)

while $x.\,left \neq NIL$
    do $x \longleftarrow x.\,left$
return x

TREE-MINIMUM: Principle

The minimum key of a binary search tree is located at the leftmost node



TREE-MINIMUM($T.\,root$)

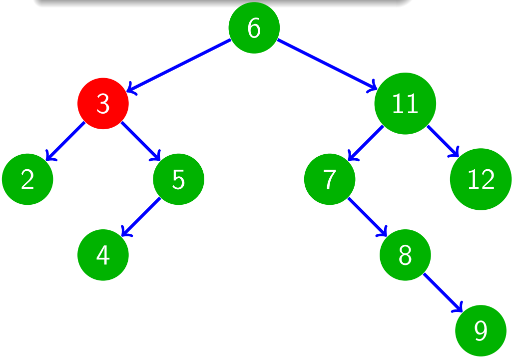
TREE-MINIMUM

TREE-MINIMUM(x)

while $x.\mathit{left} \neq \mathit{NIL}$
    do $x \longleftarrow x.\mathit{left}$
return x

TREE-MINIMUM: Principle

The minimum key of a binary search tree is located at the leftmost node



TREE-MINIMUM($T.\mathit{root}.\mathit{left}$)

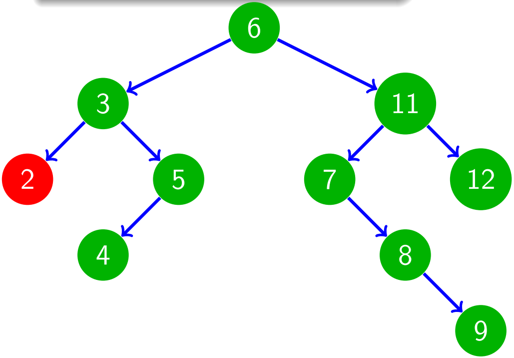
TREE-MINIMUM

TREE-MINIMUM(x)

```
while x.left ≠ NIL
      do x ⟵ x.left
return x
```

Tree-Minimum: Principle

The minimum key of a binary search tree is located at the leftmost node



TREE-MINIMUM($T.root.left.left$)

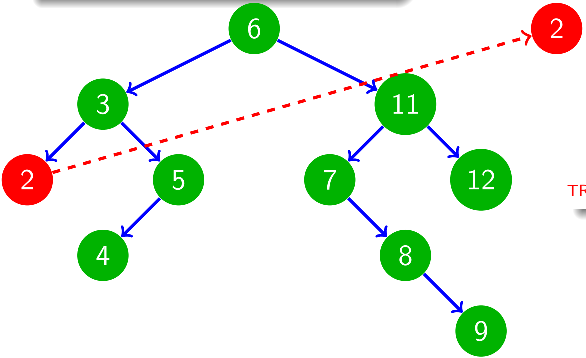
TREE-MINIMUM

TREE-MINIMUM(x)

while $x.\ left \neq NIL$
  do $x \longleftarrow x.\ left$
return x

TREE-MINIMUM: Principle

The minimum key of a binary search tree is located at the leftmost node



TREE-MINIMUM($T.\ root\ .\ left\ .\ left$)

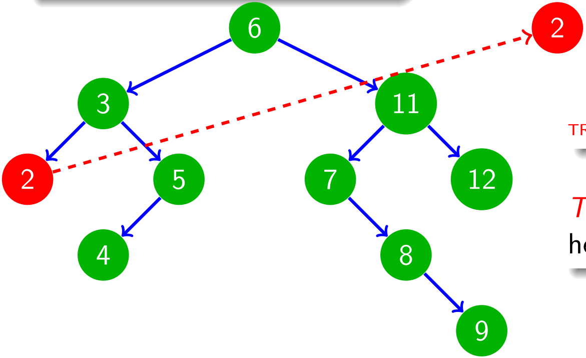
TREE-MINIMUM

TREE-MINIMUM(x)

1 **while** $x.\,left \neq NIL$
2 **do** $x \longleftarrow x.\,left$
3 **return** x

TREE-MINIMUM: Principle

The minimum key of a binary search tree is located at the leftmost node



TREE-MINIMUM($T.\,root\,.\,left\,.\,left$)

$T(n) = O(h)$ where h is the height of the tree

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

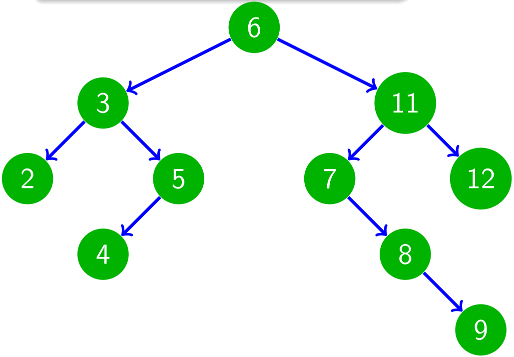
TREE-MAXIMUM

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
      do x ⟵ x.right
return x
```

Tree-Maximum: Principle

The maximum key of a binary search tree is located at the rightmost node



TREE-MAXIMUM($T.root$)

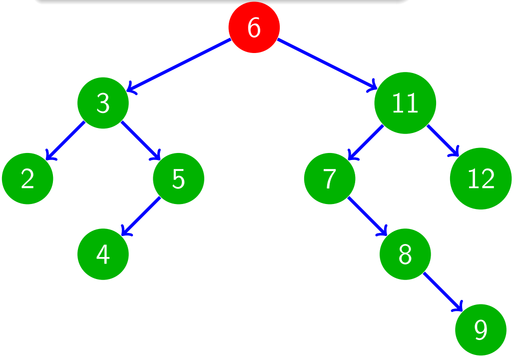
TREE-MAXIMUM

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    do x ⟵ x.right
return x
```

TREE-MAXIMUM: Principle

The maximum key of a binary search tree is located at the rightmost node



TREE-MAXIMUM($T.root$)

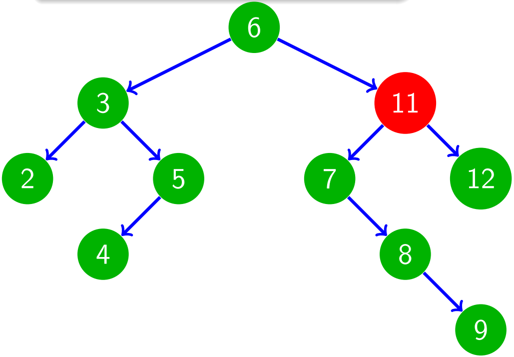
TREE-MAXIMUM

TREE-MAXIMUM(x)

while $x.\ right \neq NIL$
    do $x \longleftarrow x.\ right$
return x

Tree-Maximum: Principle

The maximum key of a binary search tree is located at the rightmost node



TREE-MAXIMUM($T.\ root\ .\ right$)

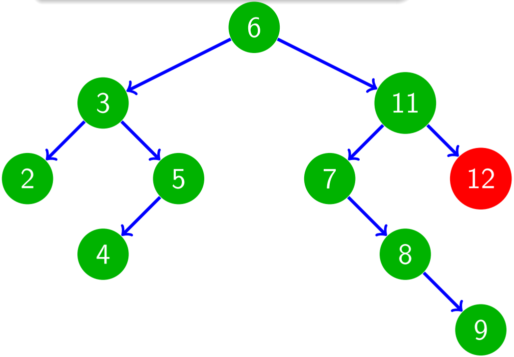
TREE-MAXIMUM

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    do x ⟵ x.right
return x
```

TREE-MAXIMUM: Principle

The maximum key of a binary search tree is located at the rightmost node



TREE-MAXIMUM($T.root.right.right$)

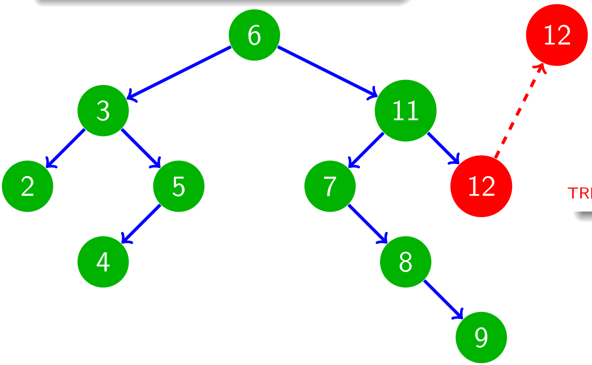
TREE-MAXIMUM

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
      do x ⟵ x.right
return x
```

Tree-Maximum: Principle

The maximum key of a binary search tree is located at the rightmost node



TREE-MAXIMUM($T.root.right.right$)

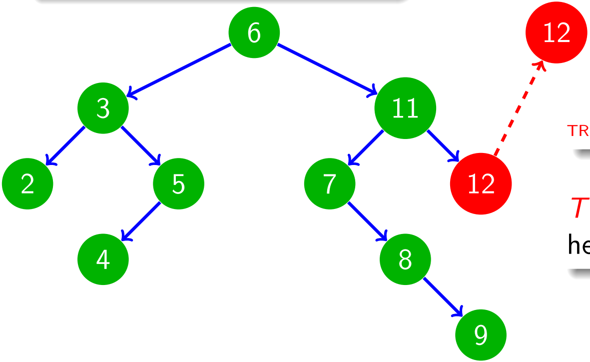
TREE-MAXIMUM

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    do x ⟵ x.right
return x
```

TREE-MAXIMUM: Principle

The maximum key of a binary search tree is located at the rightmost node



TREE-MAXIMUM($T.root.right.right$)

$T(n) = O(h)$ where h is the height of the tree

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return *NIL*
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

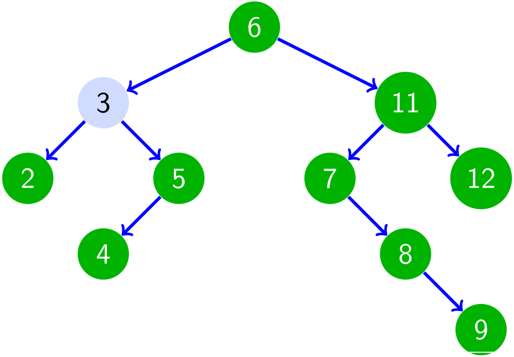
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



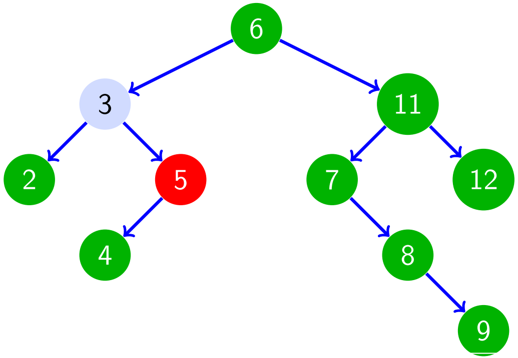
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

if $x.\,right \neq NIL$
    then return TREE-MINIMUM($x.\,right$)
$y \leftarrow x.\,parent$
while $y \neq NIL$ and $x = y.\,right$
    do $x \leftarrow y$
        $y \leftarrow y.\,parent$
return (y)

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.\,key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



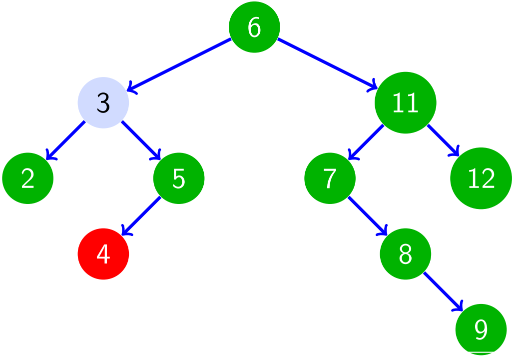
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

Tree-Successor: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
    - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
    - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



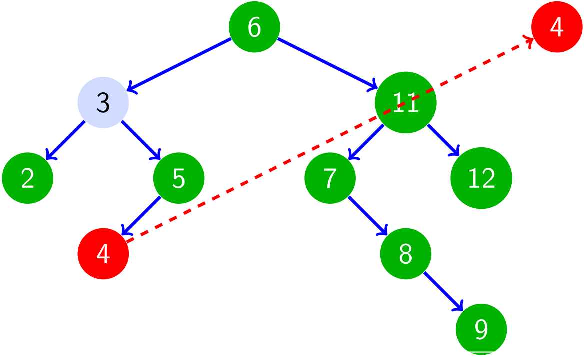
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



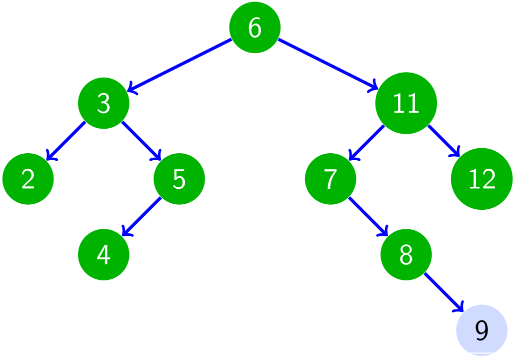
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



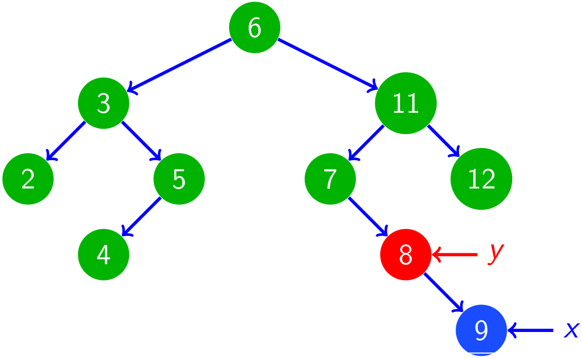
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

Tree-Successor: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



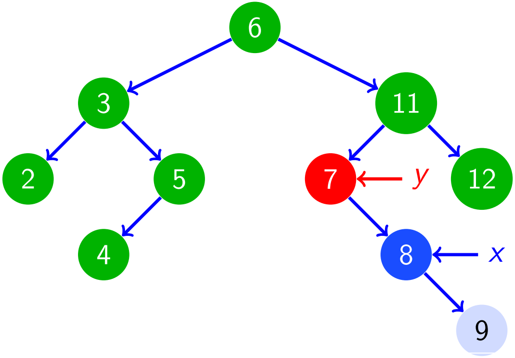
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



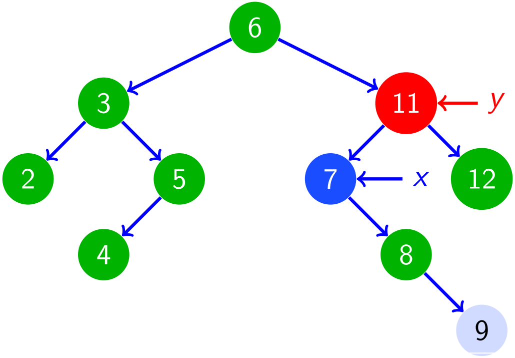
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



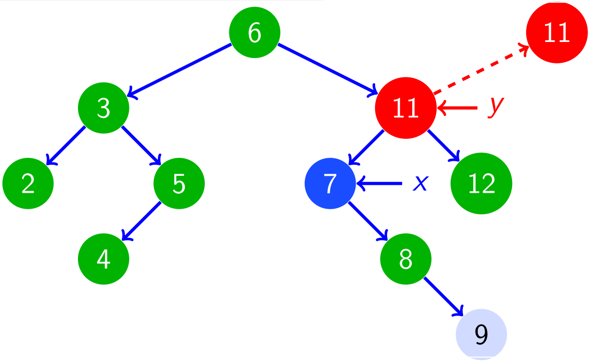
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



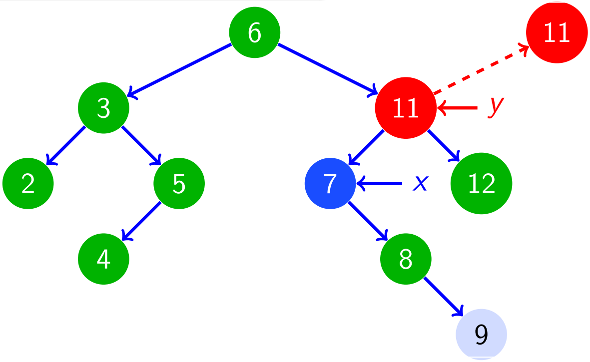
TREE-SUCCESSOR

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y ← x.parent
while y ≠ NIL and x = y.right
      do x ← y
         y ← y.parent
return (y)
```

TREE-SUCCESSOR: Principle

- Assumption: All keys are different
- To find the node with the smallest key larger than $x.key$
  - If the right subtree of x is nonempty, then the successor of x is the minimum in the right subtree of x
  - If the right subtree of x is empty, then the successor is the lowest ancestor of x whose left child is also ancestor of x (or x itself).



$T(n) = O(h)$ where h is the height of the tree

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, x)$: Delete x from T such that the binary search property is preserved.

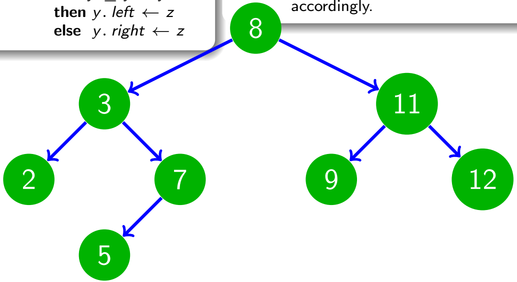
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
            then y.left ← z
            else y.right ← z
```

TREE-INSERT: Principle

- Begin at the root and maintain two pointers:
    - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
    - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$$z.key = 4$$

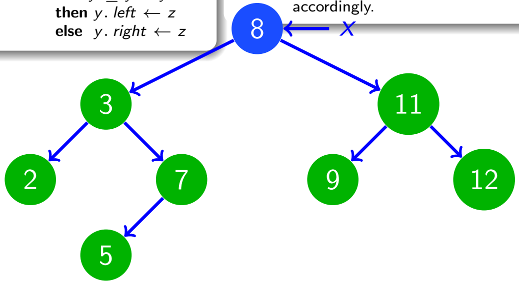
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else  x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else  if z.key ≤ y.key
            then y.left ← z
            else  y.right ← z
```

Tree-Insert: Principle

- Begin at the root and maintain two pointers:
    - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
    - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 4$

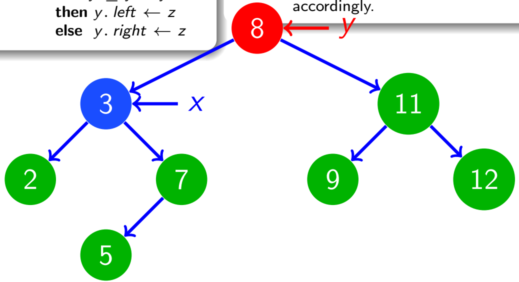
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else  x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else  if z.key ≤ y.key
            then y.left ← z
            else  y.right ← z
```

TREE-INSERT: Principle

- Begin at the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 4$

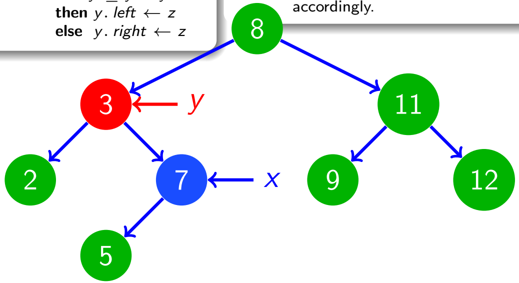
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
            then y.left ← z
            else y.right ← z
```

Tree-Insert: Principle

- Begin at the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$$z.key = 4$$

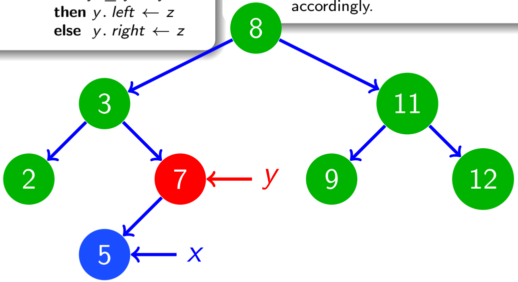
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
           then y.left ← z
           else y.right ← z
```

TREE-INSERT: Principle

- Begin at the root and maintain two pointers:
    - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
    - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 4$

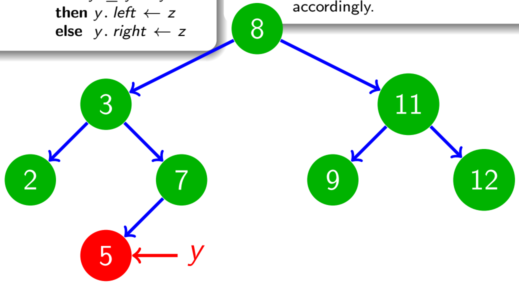
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else  x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
           then y.left ← z
           else  y.right ← z
```

Tree-Insert: Principle

- Begin at the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 4$

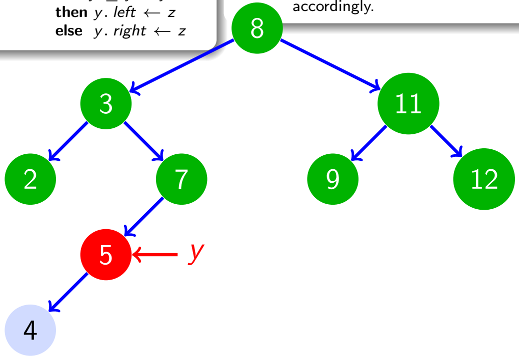
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
            then y.left ← z
            else y.right ← z
```

TREE-INSERT: Principle

- Begin at the root and maintain two pointers:
    - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
    - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 4$

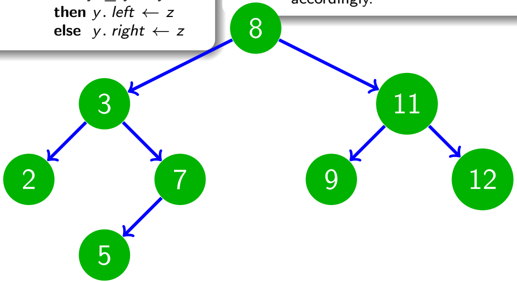
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
            then y.left ← z
            else y.right ← z
```

TREE-INSERT: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a *NIL* to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is *NIL*, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 6$

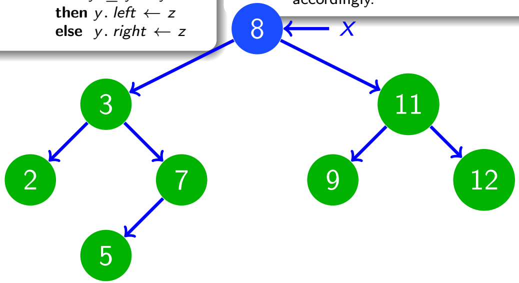
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
            then y.left ← z
            else y.right ← z
```

Tree-Insert: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a *NIL* to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.\,key$ to $z.\,key$, and move to the left or right child accordingly.
- When x is *NIL*, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.\,key = 6$

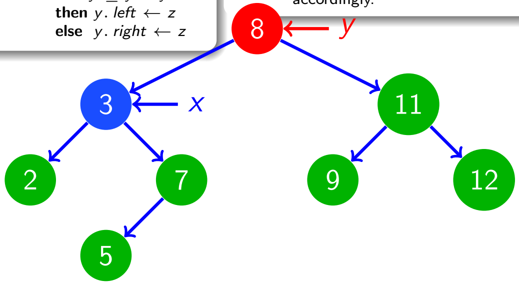
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
           then y.left ← z
           else y.right ← z
```

Tree-Insert: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$$z.key = 6$$

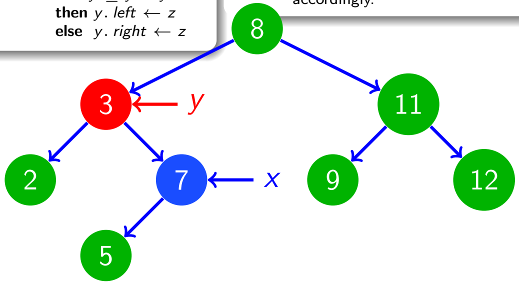
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
            then y.left ← z
            else y.right ← z
```

TREE-INSERT: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$$z.key = 6$$

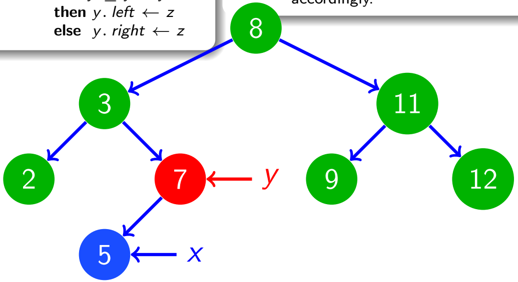
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
           then y.left ← z
           else y.right ← z
```

TREE-INSERT: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 6$

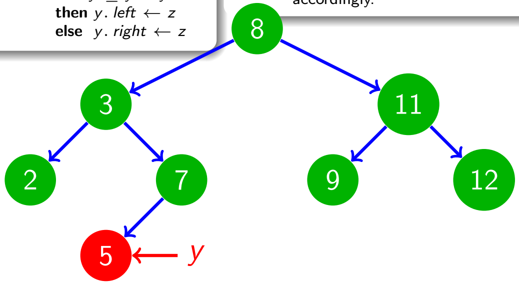
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else  x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else  if z.key ≤ y.key
            then y.left ← z
            else  y.right ← z
```

TREE-INSERT: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 6$

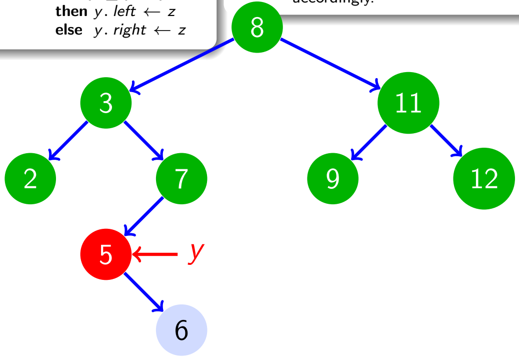
TREE-INSERT

TREE-INSERT(T, z)

```
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
            then y.left ← z
            else y.right ← z
```

Tree-Insert: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



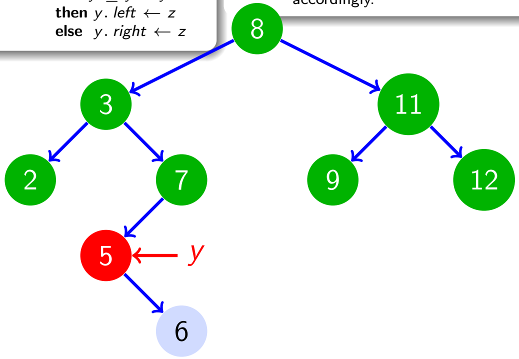
$$z.key = 6$$

TREE-INSERT

```
TREE-INSERT(T, z)
y ← NIL
x ← T.root
while x ≠ NIL
      do y ← x
         if z.key ≤ x.key
            then x ← x.left
            else x ← x.right
z.parent ← y
if y = NIL
   then T.root ← z
   else if z.key ≤ y.key
           then y.left ← z
           else y.right ← z
```

Tree-Insert: Principle

- Begin at the root of the root and maintain two pointers:
  - Pointer x traces a simple path downward looking for a NIL to replace it with the node z
  - Pointer y: Trailing pointer to keep track of the parent of x
- Traverse the tree downward by comparing $x.key$ to $z.key$, and move to the left or right child accordingly.
- When x is NIL, it is at the right position to insert z
- Compare the key of z to the one of y, and insert z o the left or right of y accordingly.



$z.key = 6$

$T(n) = O(h)$ where h is the height of the tree

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, z)$: Delete z from T such that the binary search property is preserved.

Binary Search Trees: Operations:

Let T be a tree, x be a node in T, and a key k:

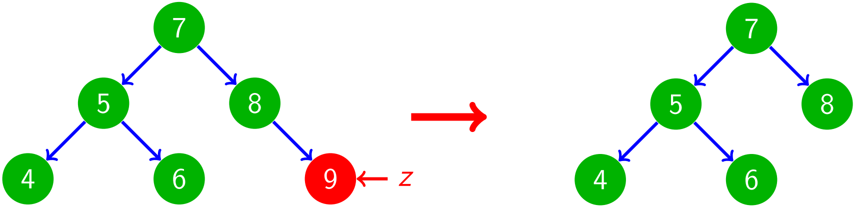
- $\textsc{Inorder-Tree-Walk}(x)$: Print out all the keys of the subtree rooted at x in a sorted order.
- $\textsc{Search}(x, k)$: Return a pointer to a node with key k in the subtree of x if one exists; otherwise, return NIL
- $\textsc{Tree-Minimum}(x)$: Return a pointer to the node with smallest key in the subtree of x.
- $\textsc{Tree-Maximum}(x)$: Return a pointer to the node with largest key in the subtree of x.
- $\textsc{Tree-Successor}(x)$: Return a pointer to the node with the smallest key larger than $x.key$.
- $\textsc{Tree-Insert}(T, x)$: Insert x in T such that the binary search property is preserved.
- $\textsc{Tree-Delete}(T, z)$: Delete z from T such that the binary search property is preserved.

TREE-DELETE: Principle

TREE-DELETE(T, z): Delete z from T such that the binary search property is preserved.

There are three cases to consider depending on the position of the node z:

- Case 1: The node z has no children
  - Delete z by making the parent of z point to *NIL*, instead of to z.

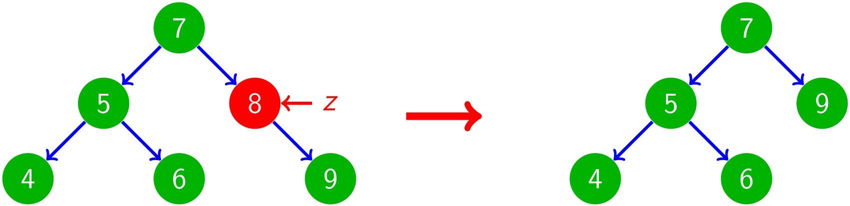


TREE-DELETE: Principle

TREE-DELETE(T, z): Delete z from T such that the binary search property is preserved.

There are three cases to consider depending on the position of the node z:

- Case 2: The node z has one child
  - Delete z by making the parent of z point to z's child, instead of to z.

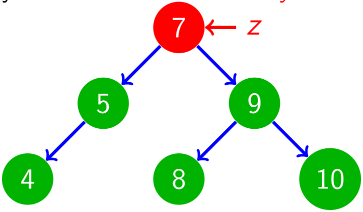


TREE-DELETE: Principle

TREE-DELETE(T, z): Delete z from T such that the binary search property is preserved.

There are three cases to consider depending on the position of the node z:

- Case 3: The node z has two children
  - Compute y the successor of z (y has at most one child).
  - Delete y from the tree (via case 1 or 2)
  - Replace z's key and satellite data with y's

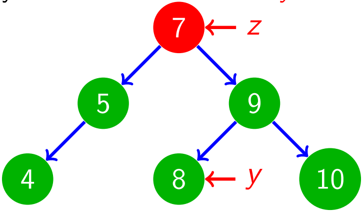


TREE-DELETE: Principle

TREE-DELETE(T, z): Delete z from T such that the binary search property is preserved.

There are three cases to consider depending on the position of the node z:

- Case 3: The node z has two children
  - Compute y the successor of z (y has at most one child).
  - Delete y from the tree (via case 1 or 2)
  - Replace z's key and satellite data with y's

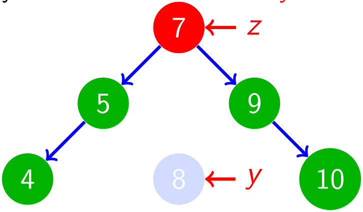


TREE-DELETE: Principle

TREE-DELETE(T, z): Delete z from T such that the binary search property is preserved.

There are three cases to consider depending on the position of the node z:

- Case 3: The node z has two children
  - Compute y the successor of z (y has at most one child).
  - Delete y from the tree (via case 1 or 2)
  - Replace z's key and satellite data with y's

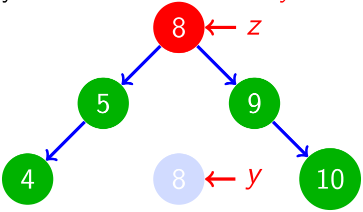


TREE-DELETE: Principle

TREE-DELETE(T, z): Delete z from T such that the binary search property is preserved.

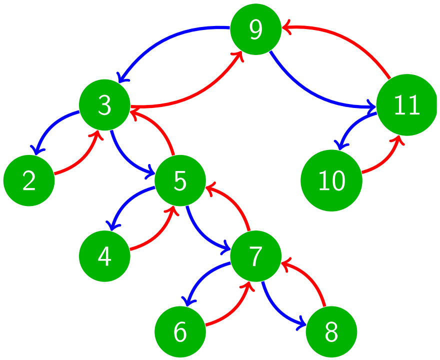
There are three cases to consider depending on the position of the node z:

- Case 3: The node z has two children
  - Compute y the successor of z (y has at most one child).
  - Delete y from the tree (via case 1 or 2)
  - Replace z's key and satellite data with y's



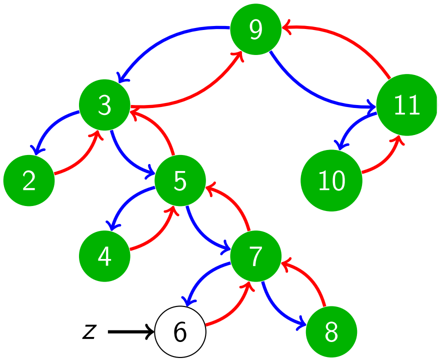
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



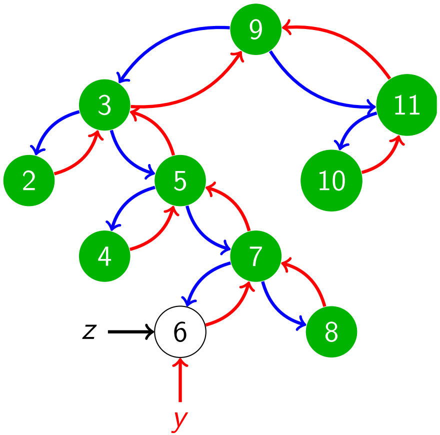
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

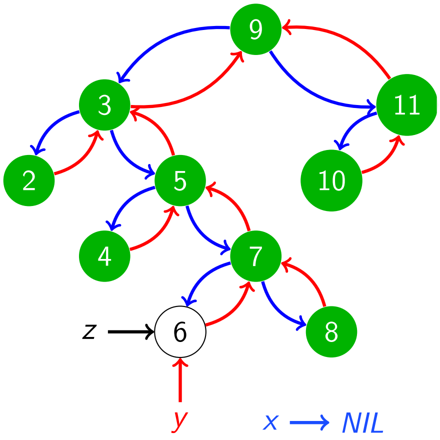
```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else if y = y.parent.left
             then y.parent.left ← x
             else y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

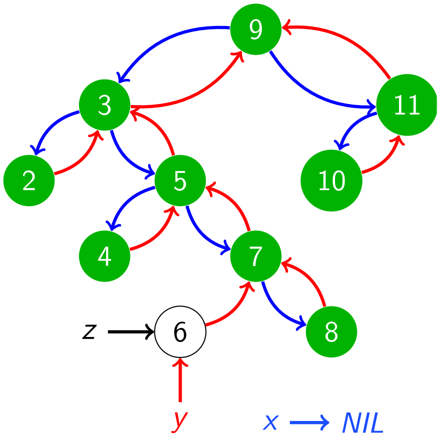
TREE-DELETE(T, z)

```
▷ Determine which node y to splice out:
  either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
  if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
  pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
  copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



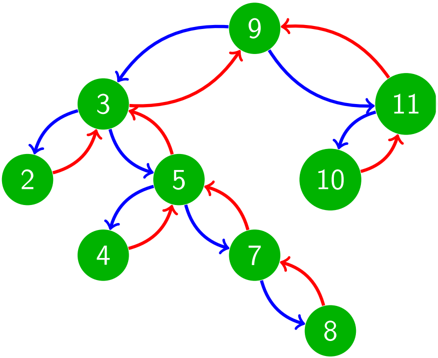
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



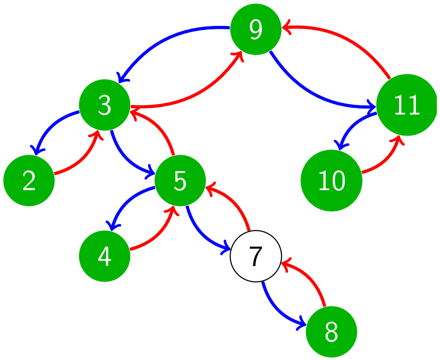
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out: either z or z's successor
if z. left = NIL or z. right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children
if y. left ≠ NIL
    then x ← y. left
    else  x ← y. right
if x ≠ NIL
    then x. parent ← y. parent
▷ y is removed from the tree by manipulating pointers of y. parent and x
if y. parent = NIL
    then T. root = x
    else  if y = y. parent . left
              then y. parent . left ← x
              else  y. parent . right ← x
▷ If it was z's successor that was spliced out, copy its data into z
if y ≠ z
    then z. key = y. key
         copy y's data into z
return y
```



TREE-DELETE

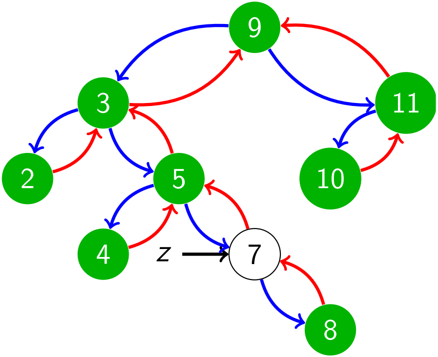
```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

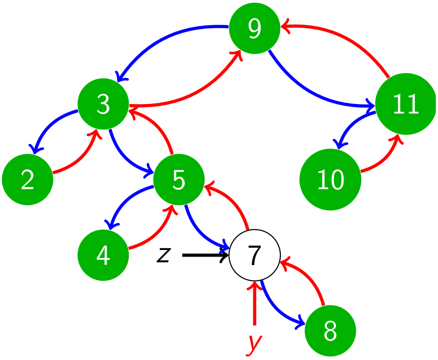
TREE-DELETE(T, z)

▷ Determine which node y to splice out: either z or z's successor
if $z.\mathit{left} = \mathit{NIL}$ or $z.\mathit{right} = \mathit{NIL}$
  then $y \leftarrow z$
  else $y \leftarrow$ TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children
if $y.\mathit{left} \neq \mathit{NIL}$
  then $x \leftarrow y.\mathit{left}$
  else $x \leftarrow y.\mathit{right}$
if $x \neq \mathit{NIL}$
  then $x.\mathit{parent} \leftarrow y.\mathit{parent}$
▷ y is removed from the tree by manipulating pointers of $y.\mathit{parent}$ and x
if $y.\mathit{parent} = \mathit{NIL}$
  then $T.\mathit{root} = x$
  else if $y = y.\mathit{parent}.\mathit{left}$
    then $y.\mathit{parent}.\mathit{left} \leftarrow x$
    else $y.\mathit{parent}.\mathit{right} \leftarrow x$
▷ If it was z's successor that was spliced out, copy its data into z
if $y \neq z$
  then $z.\mathit{key} = y.\mathit{key}$
    copy y's data into z
return y



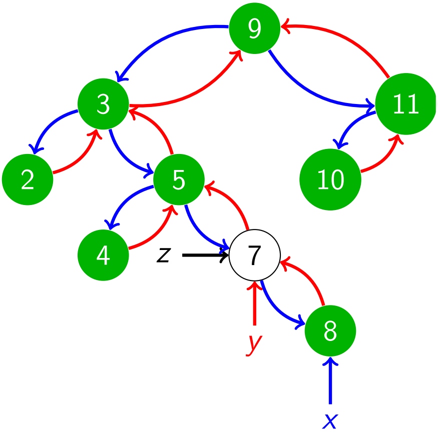
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



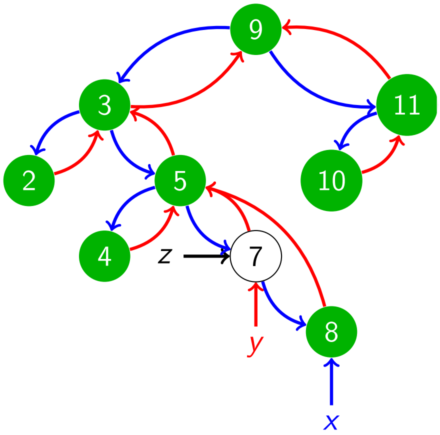
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
  either z or z's successor
if z.left = NIL or z.right = NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
  if y has no children
if y.left ≠ NIL
   then x ← y.left
   else x ← y.right
if x ≠ NIL
   then x.parent ← y.parent
▷ y is removed from the tree by manipulating
  pointers of y.parent and x
if y.parent = NIL
   then T.root = x
   else if y = y.parent.left
           then y.parent.left ← x
           else y.parent.right ← x
▷ If it was z's successor that was spliced out,
  copy its data into z
if y ≠ z
   then z.key = y.key
        copy y's data into z
return y
```



TREE-DELETE

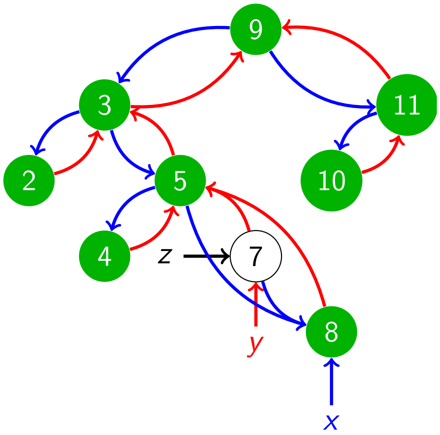
```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else if y = y.parent.left
            then y.parent.left ← x
            else y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
        copy y's data into z
return y
```



TREE-DELETE

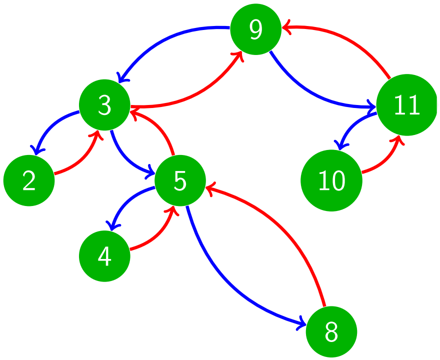
TREE-DELETE(T, z)

```
▷ Determine which node y to splice out: either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
             then y.parent.left ← x
             else  y.parent.right ← x
▷ If it was z's successor that was spliced out, copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



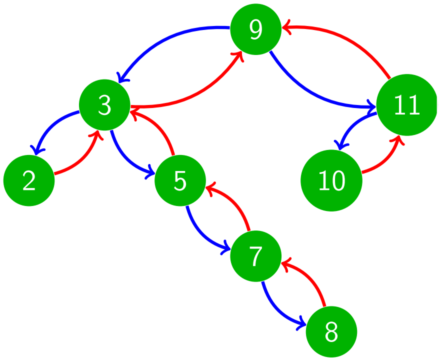
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

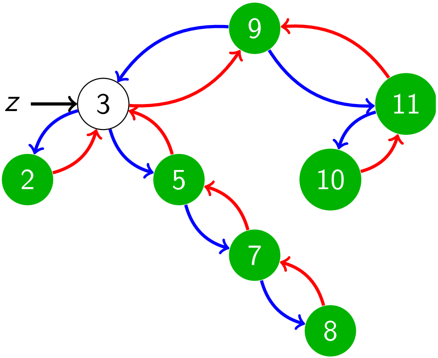
```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else if y = y.parent.left
             then y.parent.left ← x
             else y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

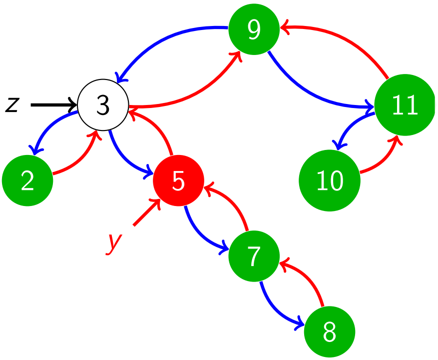
TREE-DELETE(T, z)

```
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

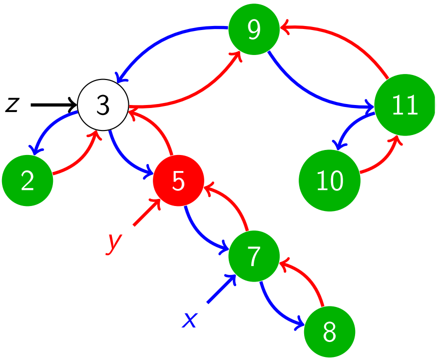
```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

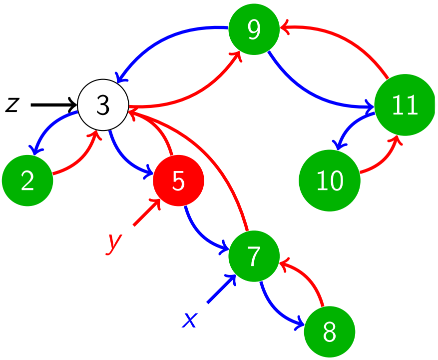
TREE-DELETE(T, z)

```
▷ Determine which node y to splice out:
  either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
  if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
  pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
  copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



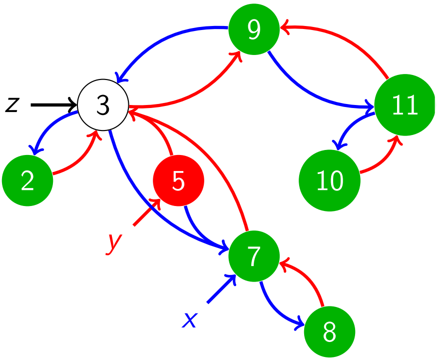
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else if y = y.parent.left
            then y.parent.left ← x
            else y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



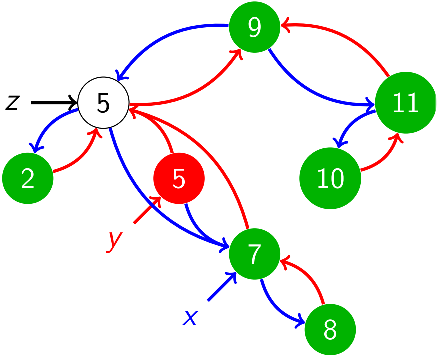
TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else if y = y.parent.left
            then y.parent.left ← x
            else y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

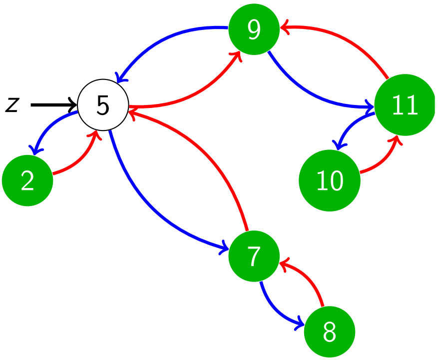
```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



TREE-DELETE

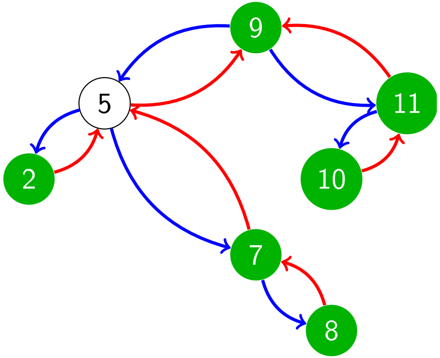
TREE-DELETE(T, z)

```
▷ Determine which node y to splice out:
  either z or z's successor
if z.left = NIL or z.right = NIL
   then y ← z
   else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
  if y has no children
if y.left ≠ NIL
   then x ← y.left
   else  x ← y.right
if x ≠ NIL
   then x.parent ← y.parent
▷ y is removed from the tree by manipulating
  pointers of y.parent and x
if y.parent = NIL
   then T.root = x
   else  if y = y.parent.left
            then y.parent.left ← x
            else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
  copy its data into z
if y ≠ z
   then z.key = y.key
        copy y's data into z
return y
```



TREE-DELETE

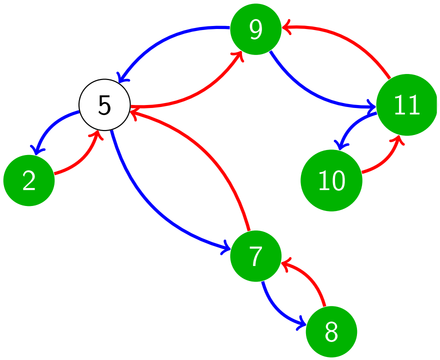
```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
            copy y's data into z
return y
```



$T(n) = O(h)$ where h is the height of the tree

TREE-DELETE

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out:
either z or z's successor
if z.left = NIL or z.right = NIL
    then y ← z
    else  y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL
if y has no children
if y.left ≠ NIL
    then x ← y.left
    else  x ← y.right
if x ≠ NIL
    then x.parent ← y.parent
▷ y is removed from the tree by manipulating
pointers of y.parent and x
if y.parent = NIL
    then T.root = x
    else  if y = y.parent.left
              then y.parent.left ← x
              else  y.parent.right ← x
▷ If it was z's successor that was spliced out,
copy its data into z
if y ≠ z
    then z.key = y.key
         copy y's data into z
return y
```



Note: In the 3'rd edition of CLRS, they use a different implementation of TREE-DELETE

Binary Search Trees

- Almost all the operations can be performed in time $O(h)$ where h is the height of the tree (the exception is INORDER-TREE-WALK, which is $\Theta(n)$).
- If the tree contains n nodes then we have:
  - Worst-Case: $h = n - 1 = O(n)$
    - For linear chain of n nodes
  - Best-Case: $h = O(log_2(n))$
    - For a complete binary tree with n nodes
  - Average-Case: $h = O(log_2(n))$
    - After inserting the keys in a uniformly random order