

Binary Search Trees

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

April 2, 2020

- Binary search trees
- Randomized binary search trees

- A ***binary search tree*** implements a *dynamic set*
 - ▶ over a ***totally ordered domain***

- A **binary search tree** implements a *dynamic set*
 - ▶ over a **totally ordered domain**

- *Interface*
 - ▶ **TREE-INSERT**(T, k) adds a key k to the dictionary D
 - ▶ **TREE-DELETE**(T, k) removes key k from D
 - ▶ **TREE-SEARCH**(T, x) tells whether D contains a key k

- A **binary search tree** implements a *dynamic set*
 - ▶ over a **totally ordered domain**

- *Interface*
 - ▶ **TREE-INSERT**(T, k) adds a key k to the dictionary D
 - ▶ **TREE-DELETE**(T, k) removes key k from D
 - ▶ **TREE-SEARCH**(T, x) tells whether D contains a key k
 - ▶ *tree-walk*: **IN-ORDER-TREE-WALK**(T), etc.

- A **binary search tree** implements a *dynamic set*
 - ▶ over a **totally ordered domain**

- *Interface*
 - ▶ **TREE-INSERT**(T, k) adds a key k to the dictionary D
 - ▶ **TREE-DELETE**(T, k) removes key k from D
 - ▶ **TREE-SEARCH**(T, x) tells whether D contains a key k
 - ▶ *tree-walk*: **IN-ORDER-TREE-WALK**(T), etc.
 - ▶ **TREE-MINIMUM**(T) finds the smallest element in the tree
 - ▶ **TREE-MAXIMUM**(T) finds the largest element in the tree

- A **binary search tree** implements a *dynamic set*
 - ▶ over a **totally ordered domain**
- *Interface*
 - ▶ **TREE-INSERT**(T, k) adds a key k to the dictionary D
 - ▶ **TREE-DELETE**(T, k) removes key k from D
 - ▶ **TREE-SEARCH**(T, x) tells whether D contains a key k
 - ▶ *tree-walk*: **IN-ORDER-TREE-WALK**(T), etc.
 - ▶ **TREE-MINIMUM**(T) finds the smallest element in the tree
 - ▶ **TREE-MAXIMUM**(T) finds the largest element in the tree
 - ▶ *iteration*: **TREE-SUCCESSOR**(x) and **TREE-PREDECESSOR**(x) find the successor and predecessor, respectively, of an element x

- *Implementation*

- ▶ T represents the tree, which consists of a set of ***nodes***

■ *Implementation*

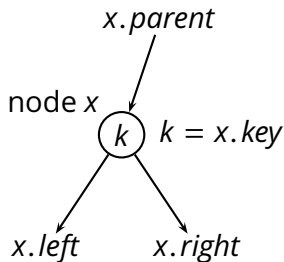
- ▶ T represents the tree, which consists of a set of **nodes**
- ▶ $T.root$ is the root node of tree T
- ▶ or sometimes T refers directly to the root node

■ Implementation

- ▶ T represents the tree, which consists of a set of **nodes**
- ▶ $T.root$ is the root node of tree T
- ▶ or sometimes T refers directly to the root node

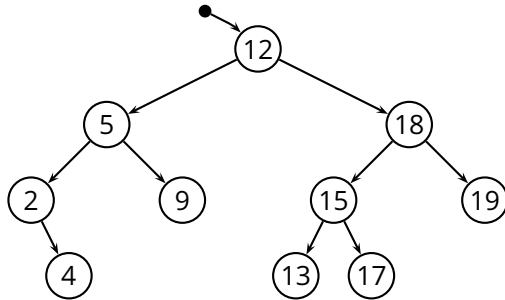
Node x

- ▶ $x.parent$ is the parent of node x
- ▶ $x.key$ is the key stored in node x
- ▶ $x.left$ is the left child of node x
- ▶ $x.right$ is the right child of node x

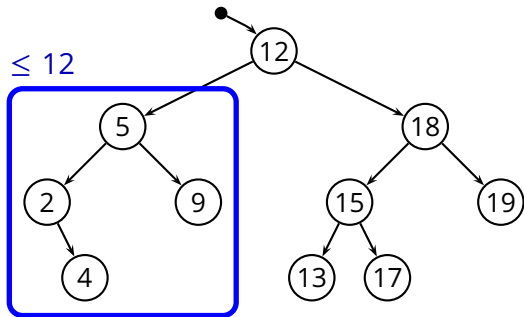


Binary Search Tree (3)

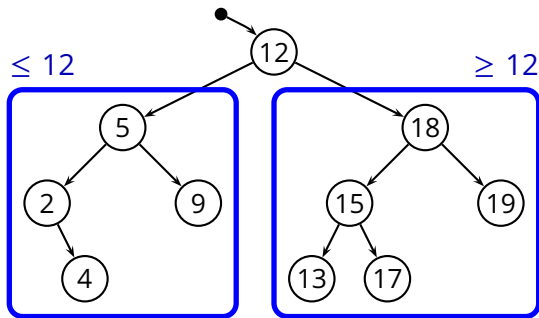
Binary Search Tree (3)

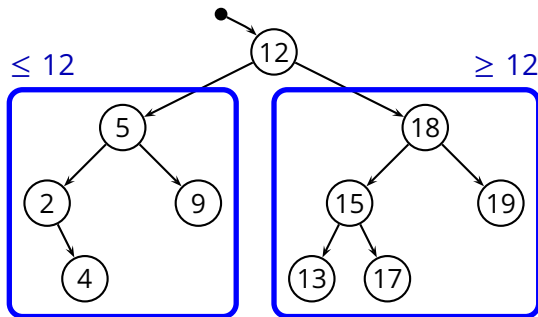


Binary Search Tree (3)



Binary Search Tree (3)

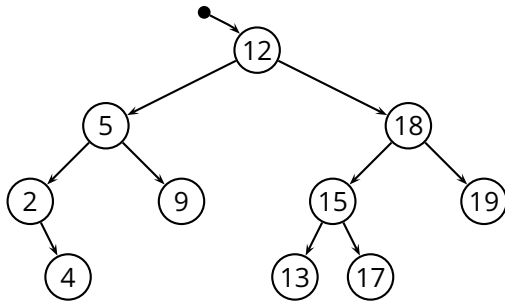




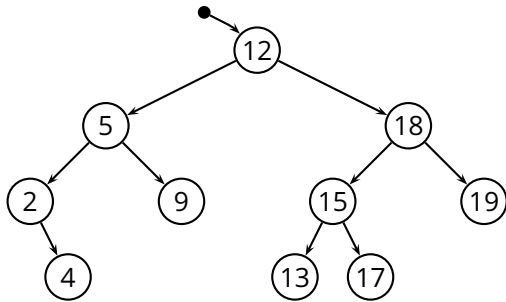
■ *Binary-search-tree property*

- ▶ for all nodes x , y , and z
- ▶ $y \in \text{left-subtree}(x) \Rightarrow y.\text{key} \leq x.\text{key}$
- ▶ $z \in \text{right-subtree}(x) \Rightarrow z.\text{key} \geq x.\text{key}$

- We want to go through the set of keys *in order*



- We want to go through the set of keys *in order*



2 4 5 9 12 13 15 17 18 19

- A recursive algorithm

- A recursive algorithm

```
INORDER-TREE-WALK(x)
```

```
1  if x ≠ NIL
```

```
2      INORDER-TREE-WALK(x.left)
```

```
3      print x.key
```

```
4      INORDER-TREE-WALK(x.right)
```

- A recursive algorithm

```
INORDER-TREE-WALK(x)
```

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

And then we need a “starter” procedure

```
INORDER-TREE-WALK-START(T)
```

```
1 INORDER-TREE-WALK(T.root)
```

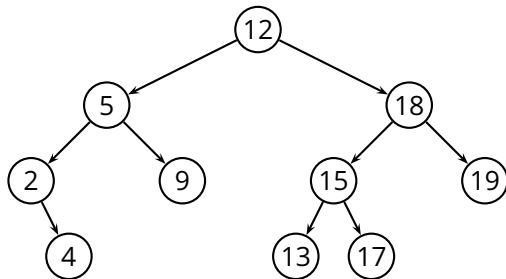
Pre-Order Tree Walk

PREORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      print  $x.\text{key}$ 
3      PREORDER-TREE-WALK( $x.\text{left}$ )
4      PREORDER-TREE-WALK( $x.\text{right}$ )
```

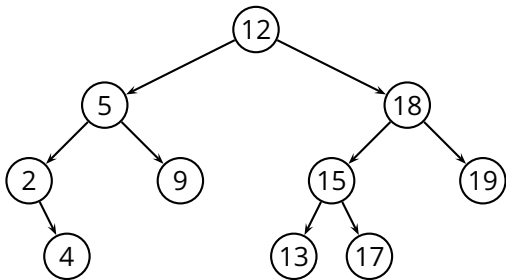
PREORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      print  $x.\text{key}$ 
3      PREORDER-TREE-WALK( $x.\text{left}$ )
4      PREORDER-TREE-WALK( $x.\text{right}$ )
```



PREORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      print  $x.\text{key}$ 
3      PREORDER-TREE-WALK( $x.\text{left}$ )
4      PREORDER-TREE-WALK( $x.\text{right}$ )
```



12 5 2 4 9 18 15 13 17 19

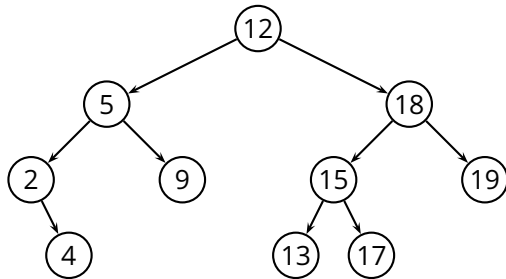
Post-Order Tree Walk

POSTORDER-TREE-WALK(*x*)

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

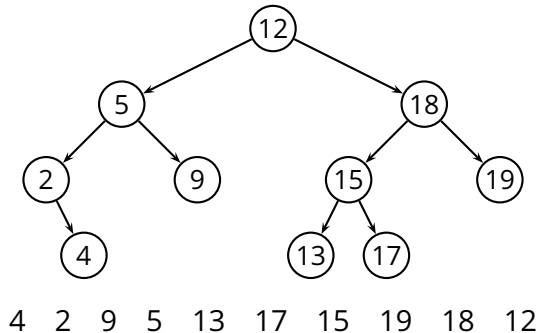
POSTORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$   
2      POSTORDER-TREE-WALK( $x.\text{left}$ )  
3      POSTORDER-TREE-WALK( $x.\text{right}$ )  
4      print  $x.\text{key}$ 
```



POSTORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$   
2      POSTORDER-TREE-WALK( $x.\text{left}$ )  
3      POSTORDER-TREE-WALK( $x.\text{right}$ )  
4      print  $x.\text{key}$ 
```



Reverse-Order Tree Walk

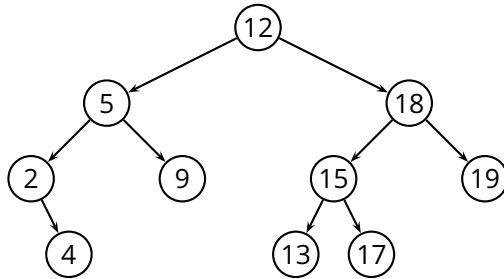
REVERSE-ORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      REVERSE-ORDER-TREE-WALK( $x.\text{right}$ )
3      print  $x.\text{key}$ 
4      REVERSE-ORDER-TREE-WALK( $x.\text{left}$ )
```

Reverse-Order Tree Walk

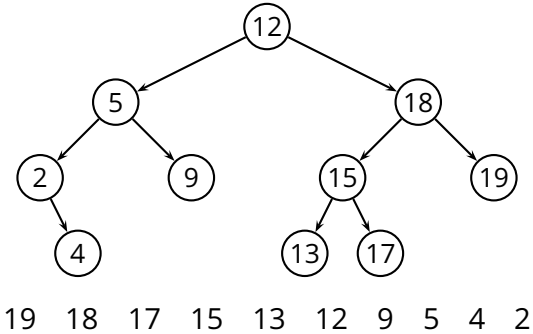
REVERSE-ORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      REVERSE-ORDER-TREE-WALK( $x.\text{right}$ )
3      print  $x.\text{key}$ 
4      REVERSE-ORDER-TREE-WALK( $x.\text{left}$ )
```



Reverse-Order Tree Walk

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



Complexity of Tree Walks

- The general recurrence is

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

- The general recurrence is

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

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

- The general recurrence is

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

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

We could prove this using the *substitution method*

- The general recurrence is

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

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

We could prove this using the *substitution method*

- Can we do better?

- The general recurrence is

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

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

We could prove this using the *substitution method*

- Can we do better? No!
 - ▶ the length of the output is $\Theta(n)$

Minimum and Maximum Keys

- Recall the *binary-search-tree property*
 - ▶ for all nodes x, y , and z
 - ▶ $y \in \text{left-subtree}(x) \Rightarrow y.\text{key} \leq x.\text{key}$
 - ▶ $z \in \text{right-subtree}(x) \Rightarrow z.\text{key} \geq x.\text{key}$

Minimum and Maximum Keys

- Recall the *binary-search-tree property*
 - ▶ for all nodes x, y , and z
 - ▶ $y \in \text{left-subtree}(x) \Rightarrow y.\text{key} \leq x.\text{key}$
 - ▶ $z \in \text{right-subtree}(x) \Rightarrow z.\text{key} \geq x.\text{key}$
- So, the minimum key is in all the way to the left
 - ▶ similarly, the maximum key is all the way to the right

```
TREE-MINIMUM( $x$ )  
1  while  $x.\text{left} \neq \text{NIL}$   
2       $x = x.\text{left}$   
3  return  $x$ 
```

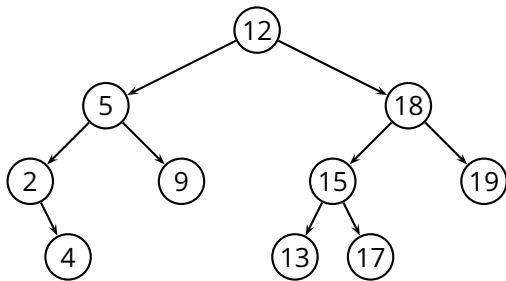
```
TREE-MAXIMUM( $x$ )  
1  while  $x.\text{right} \neq \text{NIL}$   
2       $x = x.\text{right}$   
3  return  $x$ 
```

Successor and Predecessor

- Given a node x , find the node containing the next key value

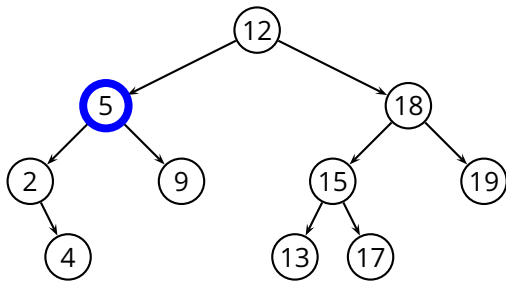
Successor and Predecessor

- Given a node x , find the node containing the next key value



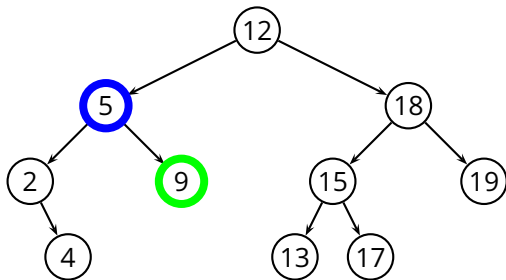
Successor and Predecessor

- Given a node x , find the node containing the next key value



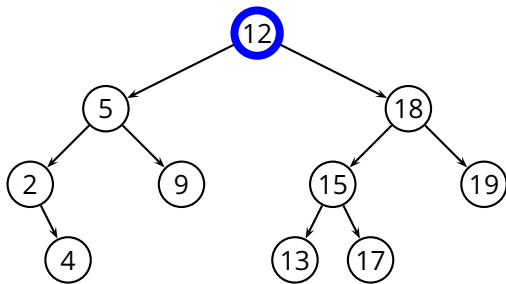
Successor and Predecessor

- Given a node x , find the node containing the next key value



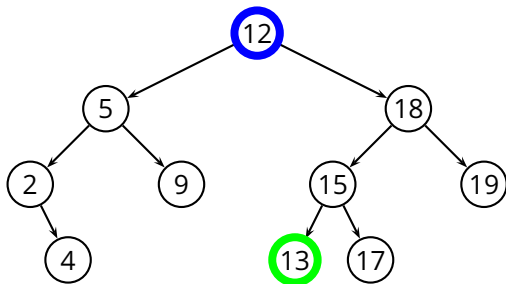
Successor and Predecessor

- Given a node x , find the node containing the next key value



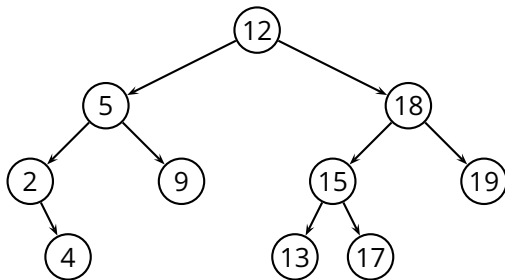
Successor and Predecessor

- Given a node x , find the node containing the next key value



Successor and Predecessor

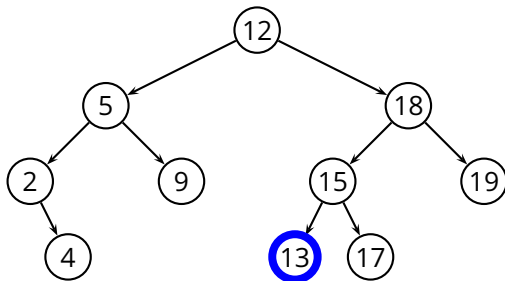
- Given a node x , find the node containing the next key value



- The successor of x is the *minimum* of the *right* subtree of x , if that exists

Successor and Predecessor

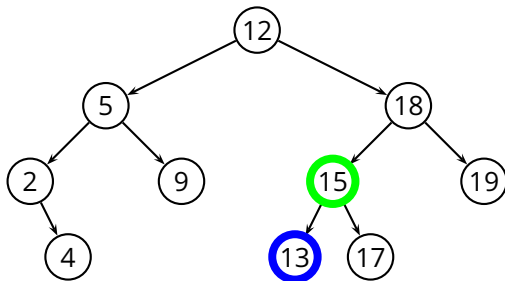
- Given a node x , find the node containing the next key value



- The successor of x is the *minimum* of the *right* subtree of x , if that exists

Successor and Predecessor

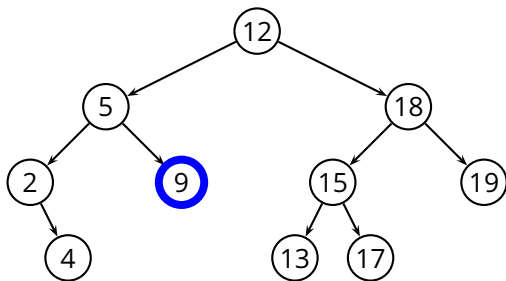
- Given a node x , find the node containing the next key value



- The successor of x is the *minimum* of the *right* subtree of x , if that exists

Successor and Predecessor

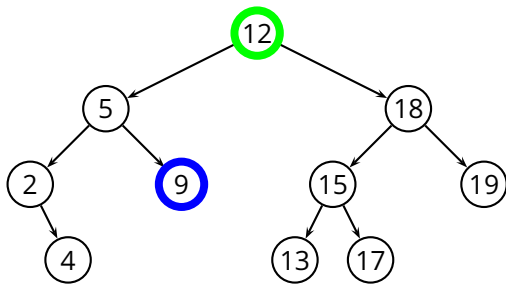
- Given a node x , find the node containing the next key value



- The successor of x is the *minimum* of the *right* subtree of x , if that exists

Successor and Predecessor

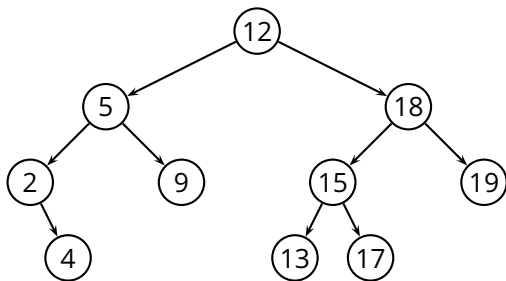
- Given a node x , find the node containing the next key value



- The successor of x is the *minimum* of the *right* subtree of x , if that exists

Successor and Predecessor

- Given a node x , find the node containing the next key value



- The successor of x is the *minimum* of the *right* subtree of x , if that exists
- Otherwise it is the *first ancestor* a of x such that x falls in the *left* subtree of a

Successor and Predecessor(2)

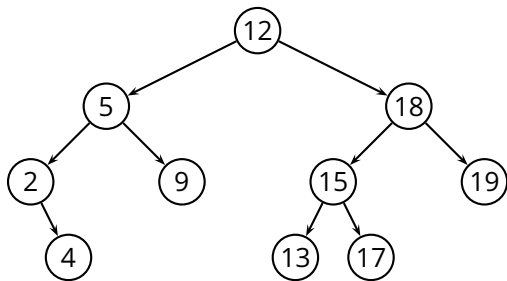
TREE-SUCCESSOR(x)

```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```

Successor and Predecessor(2)

TREE-SUCCESSOR(x)

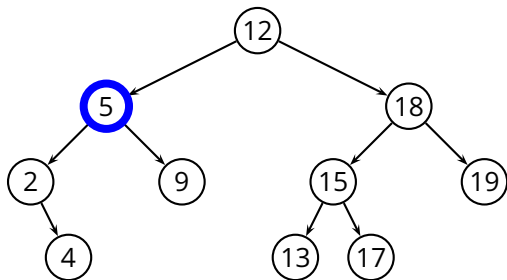
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

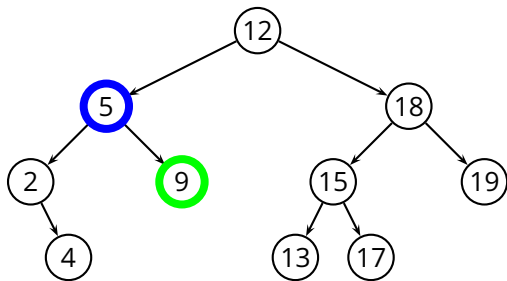
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

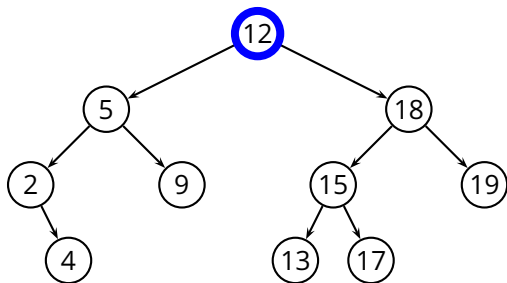
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

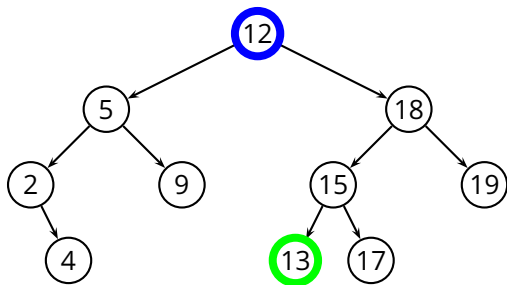
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

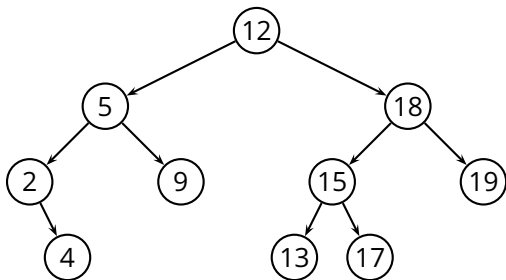
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

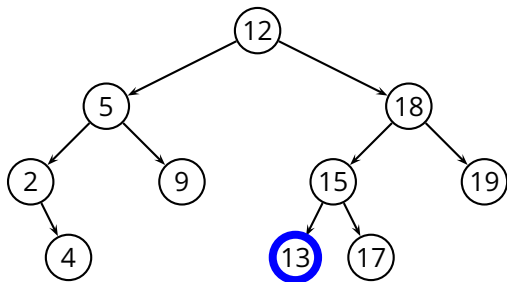
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

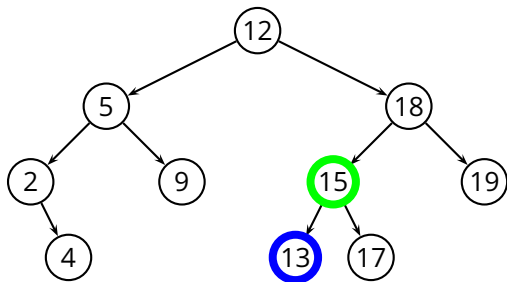
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

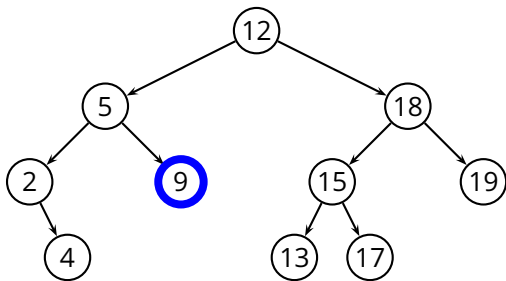
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

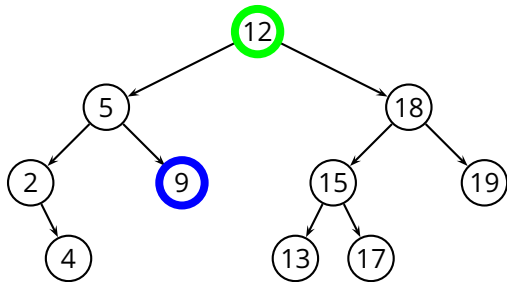
```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



Successor and Predecessor(2)

TREE-SUCCESSOR(x)

```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq \text{NIL}$  and  $x = y.right$ 
5       $x = y$ 
6       $y = y.parent$ 
7  return  $y$ 
```



- *Binary search* (thus the name of the tree)

- *Binary search* (thus the name of the tree)

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

- *Binary search* (thus the name of the tree)

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

- Is this correct?

- *Binary search* (thus the name of the tree)

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

- Is this correct? Yes, thanks to the *binary-search-tree property*

- *Binary search* (thus the name of the tree)

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

- Is this correct? Yes, thanks to the *binary-search-tree property*
- Complexity?

- *Binary search* (thus the name of the tree)

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

- Is this correct? Yes, thanks to the *binary-search-tree property*
- Complexity?

$$T(n) = \Theta(\text{depth of the tree})$$

- *Binary search* (thus the name of the tree)

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

- Is this correct? Yes, thanks to the *binary-search-tree property*
- Complexity?

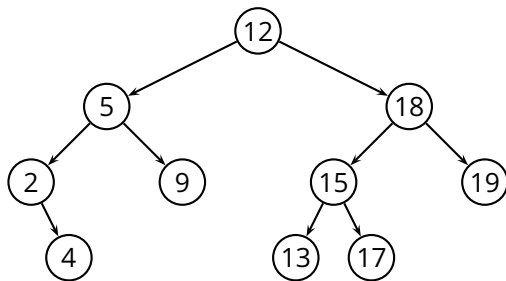
$$T(n) = \Theta(\text{depth of the tree})$$

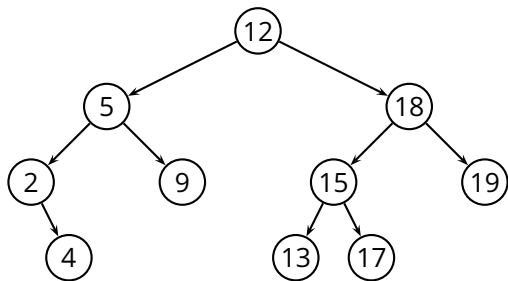
$$T(n) = O(n)$$

- Iterative *binary search*

■ Iterative *binary search*

```
ITERATIVE-TREE-SEARCH( $T, k$ )  
1  $x = T.root$   
2 while  $x \neq NIL \wedge k \neq x.key$   
3     if  $k < x.key$   
4          $x = x.left$   
5     else  $x = x.right$   
6 return  $x$ 
```



■ Idea

- ▶ in order to insert x , we *search* for x (more precisely $x.key$)
- ▶ if we don't find it, we add it where the search stopped

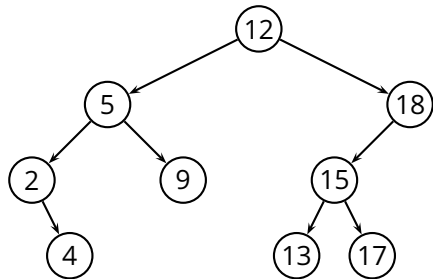
TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.\text{parent} = y$ 
9  if  $y = \text{NIL}$ 
10      $T.\text{root} = z$ 
11  else if  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 
```

Insertion (2)

TREE-INSERT(T, z)

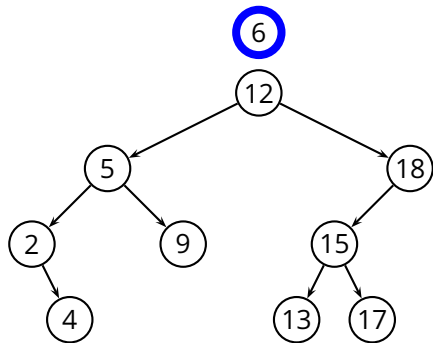
```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.\text{parent} = y$ 
9  if  $y = \text{NIL}$ 
10      $T.\text{root} = z$ 
11  else if  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 
```



Insertion (2)

TREE-INSERT(T, z)

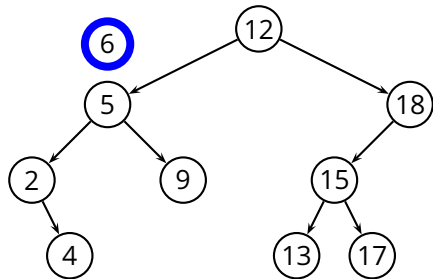
```
1  $y = \text{NIL}$ 
2  $x = T.\text{root}$ 
3 while  $x \neq \text{NIL}$ 
4    $y = x$ 
5   if  $z.\text{key} < x.\text{key}$ 
6      $x = x.\text{left}$ 
7   else  $x = x.\text{right}$ 
8  $z.\text{parent} = y$ 
9 if  $y = \text{NIL}$ 
10    $T.\text{root} = z$ 
11 else if  $z.\text{key} < y.\text{key}$ 
12    $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```



Insertion (2)

TREE-INSERT(T, z)

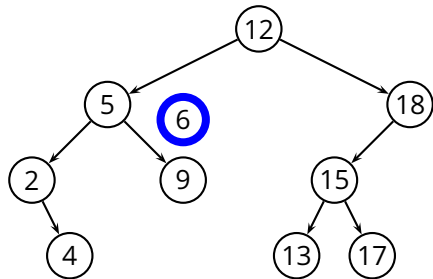
```
1  $y = \text{NIL}$ 
2  $x = T.\text{root}$ 
3 while  $x \neq \text{NIL}$ 
4    $y = x$ 
5   if  $z.\text{key} < x.\text{key}$ 
6      $x = x.\text{left}$ 
7   else  $x = x.\text{right}$ 
8  $z.\text{parent} = y$ 
9 if  $y = \text{NIL}$ 
10    $T.\text{root} = z$ 
11 else if  $z.\text{key} < y.\text{key}$ 
12    $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```



Insertion (2)

TREE-INSERT(T, z)

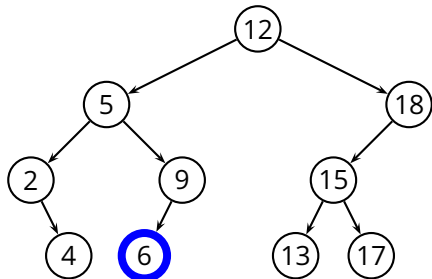
```
1  $y = \text{NIL}$ 
2  $x = T.\text{root}$ 
3 while  $x \neq \text{NIL}$ 
4    $y = x$ 
5   if  $z.\text{key} < x.\text{key}$ 
6      $x = x.\text{left}$ 
7   else  $x = x.\text{right}$ 
8  $z.\text{parent} = y$ 
9 if  $y = \text{NIL}$ 
10    $T.\text{root} = z$ 
11 else if  $z.\text{key} < y.\text{key}$ 
12    $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```



Insertion (2)

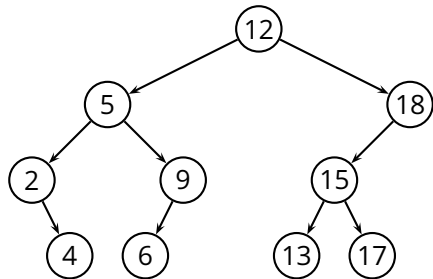
TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.\text{parent} = y$ 
9  if  $y = \text{NIL}$ 
10      $T.\text{root} = z$ 
11  else if  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 
```



TREE-INSERT(T, z)

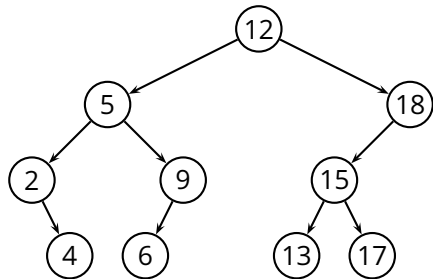
```
1  $y = \text{NIL}$ 
2  $x = T.\text{root}$ 
3 while  $x \neq \text{NIL}$ 
4    $y = x$ 
5   if  $z.\text{key} < x.\text{key}$ 
6      $x = x.\text{left}$ 
7   else  $x = x.\text{right}$ 
8  $z.\text{parent} = y$ 
9 if  $y = \text{NIL}$ 
10    $T.\text{root} = z$ 
11 else if  $z.\text{key} < y.\text{key}$ 
12    $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```



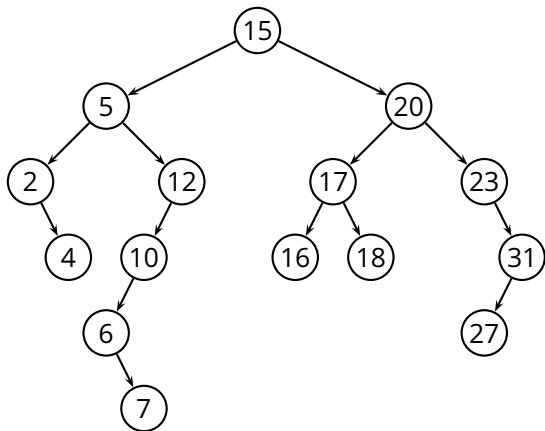
Insertion (2)

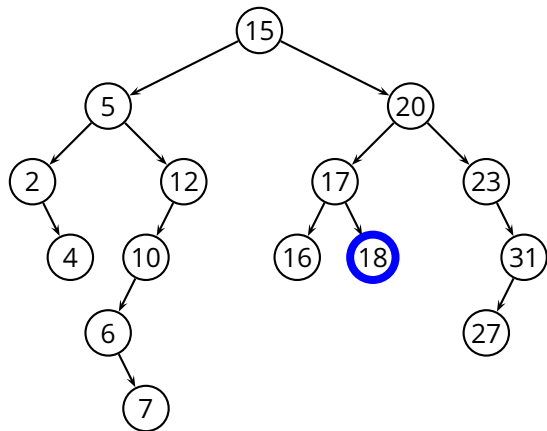
TREE-INSERT(T, z)

```
1  $y = \text{NIL}$ 
2  $x = T.\text{root}$ 
3 while  $x \neq \text{NIL}$ 
4    $y = x$ 
5   if  $z.\text{key} < x.\text{key}$ 
6      $x = x.\text{left}$ 
7   else  $x = x.\text{right}$ 
8  $z.\text{parent} = y$ 
9 if  $y = \text{NIL}$ 
10    $T.\text{root} = z$ 
11 else if  $z.\text{key} < y.\text{key}$ 
12    $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```

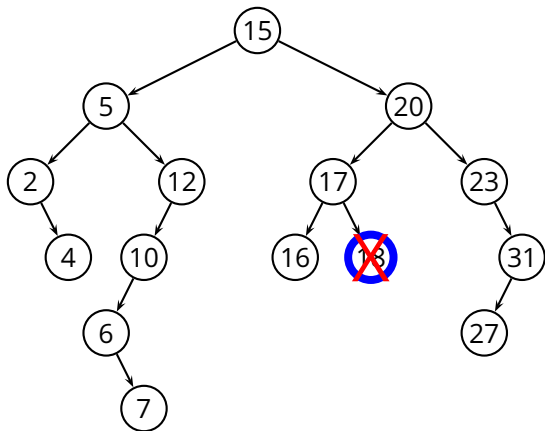


$$T(n) = \Theta(h)$$



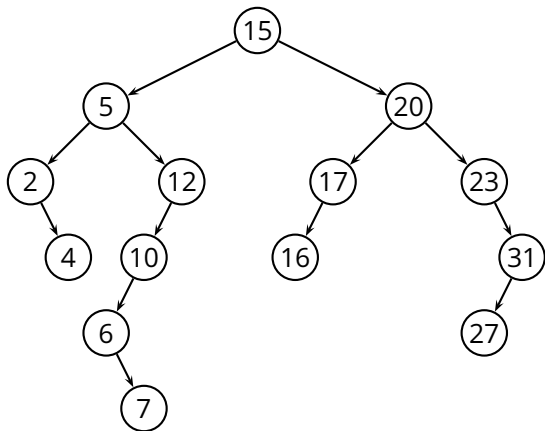


1. z has no children

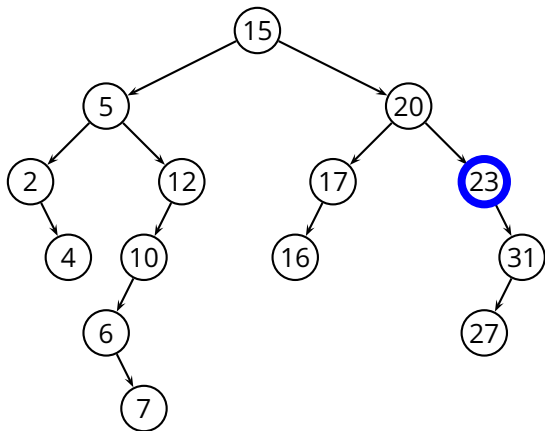


1. z has no children

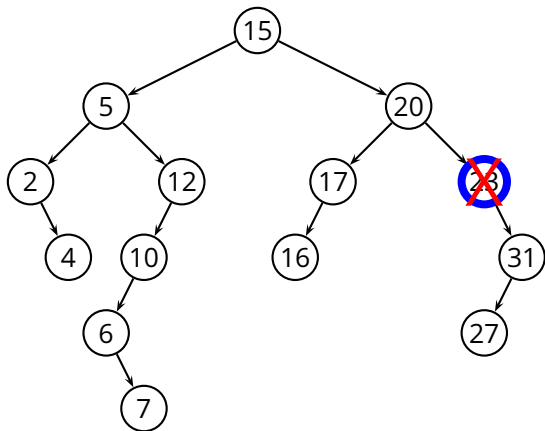
- ▶ simply remove z



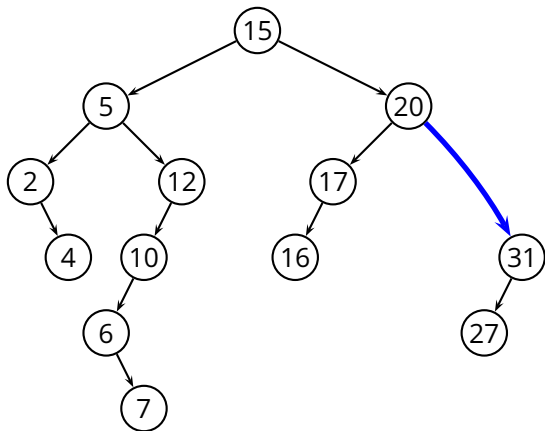
1. z has no children
 - ▶ simply remove z



1. z has no children
 - ▶ simply remove z
2. z has one child



1. z has no children
 - ▶ simply remove z
2. z has one child
 - ▶ remove z

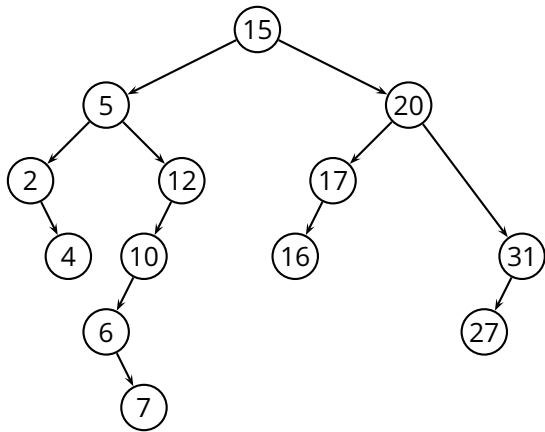


1. z has no children

- ▶ simply remove z

2. z has one child

- ▶ remove z
- ▶ connect *z.parent* to *z.right*

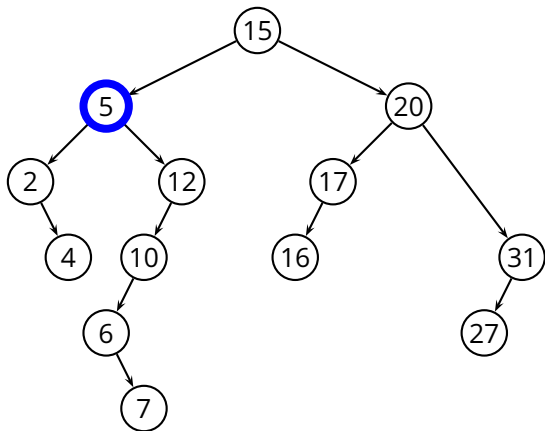


1. z has no children

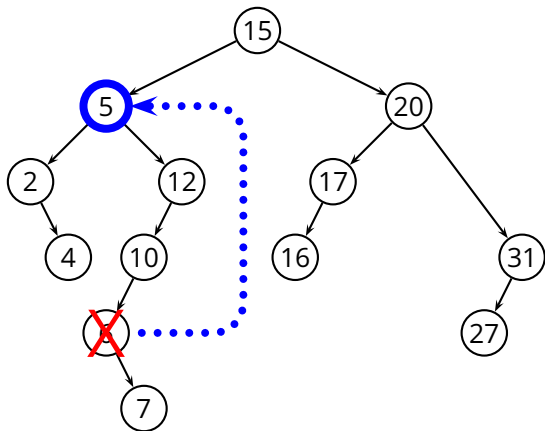
- ▶ simply remove z

2. z has one child

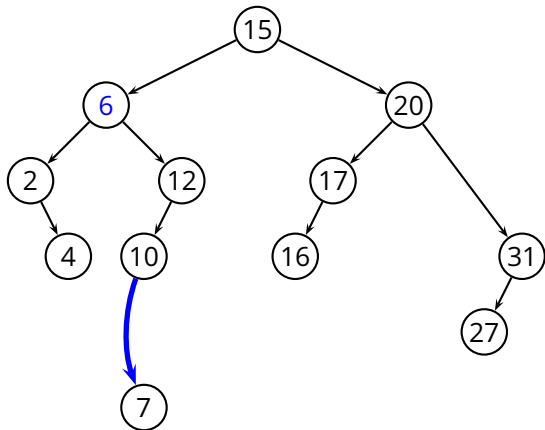
- ▶ remove z
- ▶ connect z.*parent* to z.*right*



1. z has no children
 - ▶ simply remove z
2. z has one child
 - ▶ remove z
 - ▶ connect z.*parent* to z.*right*
3. z has two children



1. z has no children
 - ▶ simply remove z
2. z has one child
 - ▶ remove z
 - ▶ connect $z.parent$ to $z.right$
3. z has two children
 - ▶ replace z with $y = \mathbf{TREE-SUCCESSOR}(z)$
 - ▶ remove y (1 child!)



1. z has no children
 - ▶ simply remove z
2. z has one child
 - ▶ remove z
 - ▶ connect $z.parent$ to $z.right$
3. z has two children
 - ▶ replace z with $y = \mathbf{TREE-SUCCESSOR}(z)$
 - ▶ remove y (1 child!)
 - ▶ connect $y.parent$ to $y.right$

TREE-DELETE(T, z)

```
1  if  $z.left = \text{NIL}$  or  $z.right = \text{NIL}$ 
2       $y = z$ 
3  else  $y = \text{TREE-SUCCESSOR}(z)$ 
4  if  $y.left \neq \text{NIL}$ 
5       $x = y.left$ 
6  else  $x = y.right$ 
7  if  $x \neq \text{NIL}$ 
8       $x.parent = y.parent$ 
9  if  $y.parent == \text{NIL}$ 
10      $T.root = x$ 
11 else if  $y = y.parent.left$ 
12      $y.parent.left = x$ 
13     else  $y.parent.right = x$ 
14 if  $y \neq z$ 
15      $z.key = y.key$ 
16     copy any other data from  $y$  into  $z$ 
```

- Insertion, search, and deletion operations have complexity $\Theta(h)$, where h is the height of the tree

- Insertion, search, and deletion operations have complexity $\Theta(h)$, where h is the height of the tree
- $h = O(\log n)$ in the average case
 - ▶ i.e., with a random insertion order

- Insertion, search, and deletion operations have complexity $\Theta(h)$, where h is the height of the tree
- $h = O(\log n)$ in the average case
 - ▶ i.e., with a random insertion order
- $h = O(n)$ in some particular cases

- Insertion, search, and deletion operations have complexity $\Theta(h)$, where h is the height of the tree
- $h = O(\log n)$ in the average case
 - ▶ i.e., with a random insertion order
- $h = O(n)$ in some particular cases
 - ▶ i.e., with ordered sequences

- Insertion, search, and deletion operations have complexity $\Theta(h)$, where h is the height of the tree
- $h = O(\log n)$ in the average case
 - ▶ i.e., with a random insertion order
- $h = O(n)$ in some particular cases
 - ▶ i.e., with ordered sequences
 - ▶ the problem is that the “worst” case is not that uncommon

- Insertion, search, and deletion operations have complexity $\Theta(h)$, where h is the height of the tree
- $h = O(\log n)$ in the average case
 - ▶ i.e., with a random insertion order
- $h = O(n)$ in some particular cases
 - ▶ i.e., with ordered sequences
 - ▶ the problem is that the “worst” case is not that uncommon
- **Idea:** use randomization to turn all cases into the average case

- ***Idea 1:*** insert every sequence as a random sequence

- **Idea 1:** insert every sequence as a random sequence
 - ▶ i.e., given $A = \langle 1, 2, 3, \dots, n \rangle$, insert a *random permutation* of A

- **Idea 1:** insert every sequence as a random sequence
 - ▶ i.e., given $A = \langle 1, 2, 3, \dots, n \rangle$, insert a *random permutation* of A
 - ▶ problem: A is not necessarily known in advance

- **Idea 1:** insert every sequence as a random sequence
 - ▶ i.e., given $A = \langle 1, 2, 3, \dots, n \rangle$, insert a *random permutation* of A
 - ▶ problem: A is not necessarily known in advance
- **Idea 2:** we can obtain a random permutation of the input sequence by randomly alternating two insertion procedures
 - ▶ *tail insertion*: this is what **TREE-INSERT** does

- **Idea 1:** insert every sequence as a random sequence
 - ▶ i.e., given $A = \langle 1, 2, 3, \dots, n \rangle$, insert a *random permutation* of A
 - ▶ problem: A is not necessarily known in advance
- **Idea 2:** we can obtain a random permutation of the input sequence by randomly alternating two insertion procedures
 - ▶ *tail insertion*: this is what **TREE-INSERT** does
 - ▶ *head insertion*: for this we need a new procedure **TREE-ROOT-INSERT**
 - ▶ inserts n in T as if n was inserted as the first element

Randomized Insertion (2)

TREE-RANDOMIZED-INSERT1(T, z)

- 1 $r =$ uniformly random value from $\{1, \dots, t.size + 1\}$
- 2 **if** $r = 1$
- 3 **TREE-ROOT-INSERT**(T, z)
- 4 **else** **TREE-INSERT**(T, z)

TREE-RANDOMIZED-INSERT1(T, z)

```
1  $r =$  uniformly random value from  $\{1, \dots, t.size + 1\}$   
2 if  $r = 1$   
3     TREE-ROOT-INSERT( $T, z$ )  
4 else TREE-INSERT( $T, z$ )
```

- Does this really simulate a random permutation?
 - ▶ i.e., with all permutations being equally likely?

TREE-RANDOMIZED-INSERT1(T, z)

```
1  $r =$  uniformly random value from  $\{1, \dots, t.size + 1\}$   
2 if  $r = 1$   
3     TREE-ROOT-INSERT( $T, z$ )  
4 else TREE-INSERT( $T, z$ )
```

- Does this really simulate a random permutation?
 - ▶ i.e., with all permutations being equally likely?
 - ▶ no, clearly the last element can only go to the top or to the bottom

TREE-RANDOMIZED-INSERT1(T, z)

```
1  $r =$  uniformly random value from  $\{1, \dots, t.size + 1\}$ 
2 if  $r = 1$ 
3     TREE-ROOT-INSERT( $T, z$ )
4 else TREE-INSERT( $T, z$ )
```

- Does this really simulate a random permutation?
 - ▶ i.e., with all permutations being equally likely?
 - ▶ no, clearly the last element can only go to the top or to the bottom
- It is true that any node has the same probability of being inserted at the top

TREE-RANDOMIZED-INSERT1(T, z)

```
1  $r =$  uniformly random value from  $\{1, \dots, t.size + 1\}$ 
2 if  $r = 1$ 
3     TREE-ROOT-INSERT( $T, z$ )
4 else TREE-INSERT( $T, z$ )
```

- Does this really simulate a random permutation?
 - ▶ i.e., with all permutations being equally likely?
 - ▶ no, clearly the last element can only go to the top or to the bottom
- It is true that any node has the same probability of being inserted at the top
 - ▶ this suggests a recursive application of this same procedure

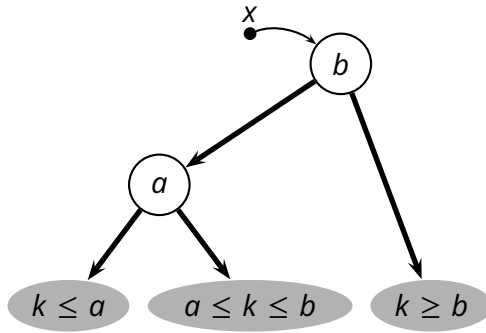
Randomized Insertion (3)

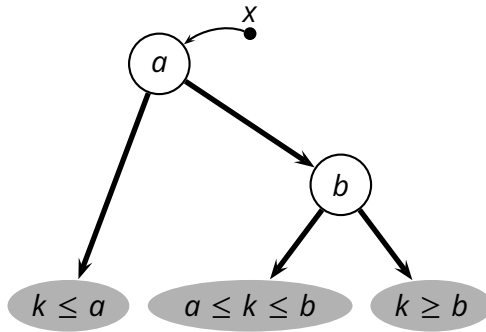
TREE-RANDOMIZED-INSERT(t, z)

```
1  if  $t = \text{NIL}$ 
2      return  $z$ 
3   $r =$  uniformly random value from  $\{1, \dots, t.size + 1\}$ 
4  if  $r = 1$                                 //  $\text{Pr}[r = 1] = 1/(t.size + 1)$ 
5       $z.size = t.size + 1$ 
6      return TREE-ROOT-INSERT( $t, z$ )
7  if  $z.key < t.key$ 
8       $t.left =$  TREE-RANDOMIZED-INSERT( $t.left, z$ )
9  else  $t.right =$  TREE-RANDOMIZED-INSERT( $t.right, z$ )
10  $t.size = t.size + 1$ 
11 return  $t$ 
```

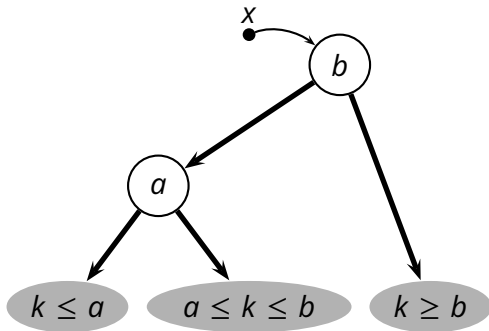
```
TREE-RANDOMIZED-INSERT(t, z)
1  if t = NIL
2      return z
3  r = uniformly random value from {1, ..., t.size + 1}
4  if r = 1                                // Pr[r = 1] = 1/(t.size + 1)
5      z.size = t.size + 1
6      return TREE-ROOT-INSERT(t, z)
7  if z.key < t.key
8      t.left = TREE-RANDOMIZED-INSERT(t.left, z)
9  else t.right = TREE-RANDOMIZED-INSERT(t.right, z)
10 t.size = t.size + 1
11 return t
```

- Looks like this one really simulates a random permutation...



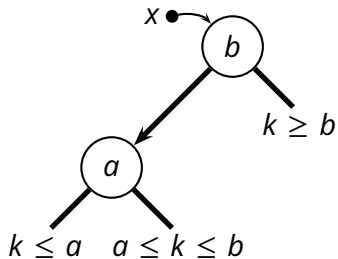


■ $x = \text{RIGHT-ROTATE}(x)$



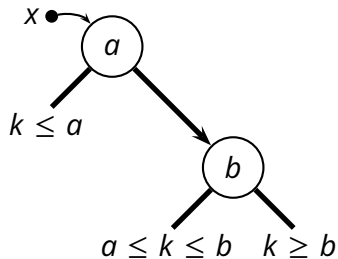
■ $x = \text{RIGHT-ROTATE}(x)$

■ $x = \text{LEFT-ROTATE}(x)$



RIGHT-ROTATE

LEFT-ROTATE

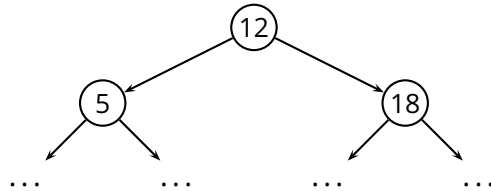
**RIGHT-ROTATE**(x)

- 1 $l = x.left$
- 2 $x.left = l.right$
- 3 $l.right = x$
- 4 **return** l

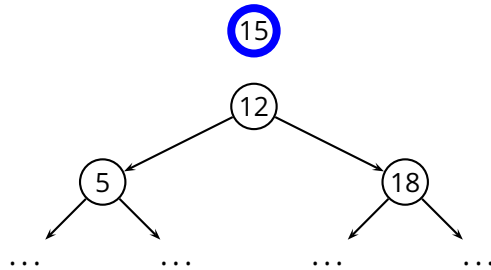
LEFT-ROTATE(x)

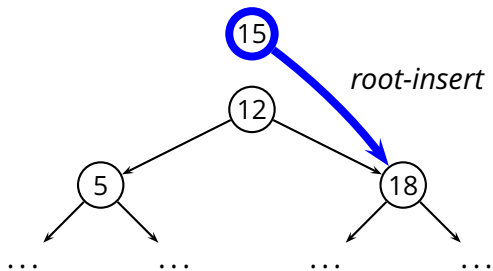
- 1 $r = x.right$
- 2 $x.right = r.left$
- 3 $r.left = x$
- 4 **return** r

Root Insertion

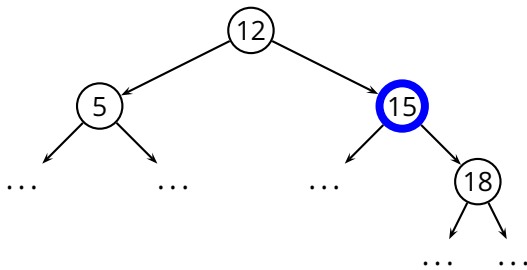


Root Insertion

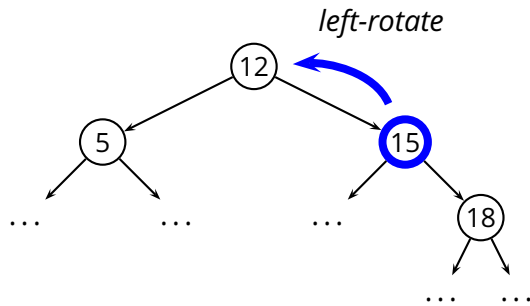




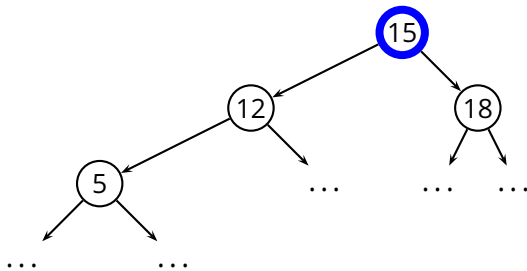
1. Recursively insert z at the root of the appropriate subtree (right)



1. Recursively insert z at the root of the appropriate subtree (right)



1. Recursively insert z at the root of the appropriate subtree (right)
2. Rotate x with z (left-rotate)



1. Recursively insert z at the root of the appropriate subtree (right)
2. Rotate x with z (left-rotate)

TREE-ROOT-INSERT(x, z)

```
1  if  $x = \text{NIL}$ 
2      return  $z$ 
3  if  $z.\text{key} < x.\text{key}$ 
4       $x.\text{left} = \text{TREE-ROOT-INSERT}(x.\text{left}, z)$ 
5      return RIGHT-ROTATE( $x$ )
6  else  $x.\text{right} = \text{TREE-ROOT-INSERT}(x.\text{right}, z)$ 
7      return LEFT-ROTATE( $x$ )
```

- General strategies to deal with complexity in the worst case

- General strategies to deal with complexity in the worst case
 - ▶ *randomization*: turns any case into the average case
 - ▶ the worst case is still possible, but it is extremely improbable

- General strategies to deal with complexity in the worst case
 - ▶ *randomization*: turns any case into the average case
 - ▶ the worst case is still possible, but it is extremely improbable
 - ▶ *amortized maintenance*: e.g., balancing a BST or resizing a hash table
 - ▶ relatively expensive but “amortized” operations

- General strategies to deal with complexity in the worst case
 - ▶ *randomization*: turns any case into the average case
 - ▶ the worst case is still possible, but it is extremely improbable
 - ▶ *amortized maintenance*: e.g., balancing a BST or resizing a hash table
 - ▶ relatively expensive but “amortized” operations
 - ▶ *optimized data structures*: a self-balanced data structure
 - ▶ guaranteed $O(\log n)$ complexity bounds