Binary Search Trees

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

April 18, 2023

Outline

- Binary search trees
- Randomized binary search trees

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

Binary Search 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

Binary Search 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.

- 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***

Binary Search Tree (2)

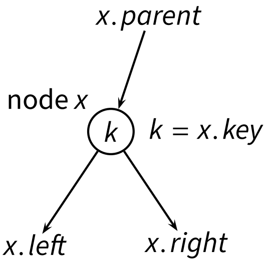
- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***
  - $T.root$ is the root node of tree T
  - or sometimes T refers directly to the root node

Binary Search Tree (2)

- *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

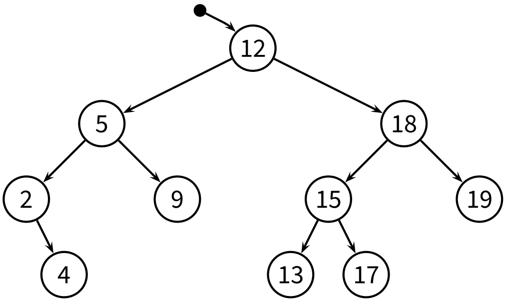
Node x

- $x.parent$ 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

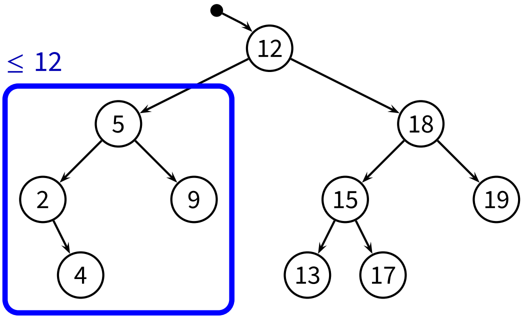


Binary Search Tree (3)

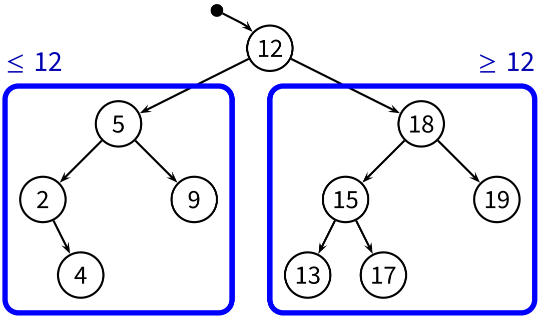
Binary Search Tree (3)



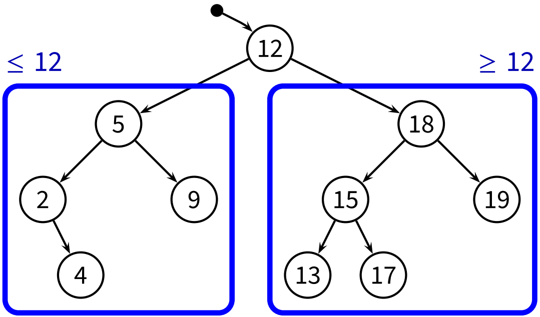
Binary Search Tree (3)



Binary Search Tree (3)

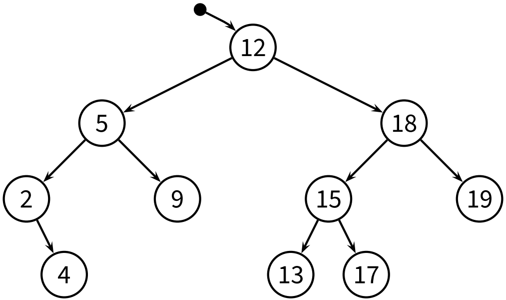


Binary Search Tree (3)

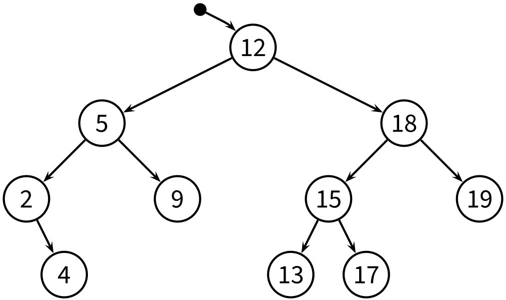


- ***Binary-search-tree property***
  - for all nodes x, y, and z
  - $y \in \mathit{left\text{-}subtree}(x) \Rightarrow y.\mathit{key} \leq x.\mathit{key}$
  - $z \in \mathit{right\text{-}subtree}(x) \Rightarrow z.\mathit{key} \geq x.\mathit{key}$

- 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

In-Order Tree Walk (2)

- A recursive algorithm

In-Order Tree Walk (2)

- A recursive algorithm

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

In-Order Tree Walk (2)

- A recursive algorithm

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

And then we need a “starter” procedure

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

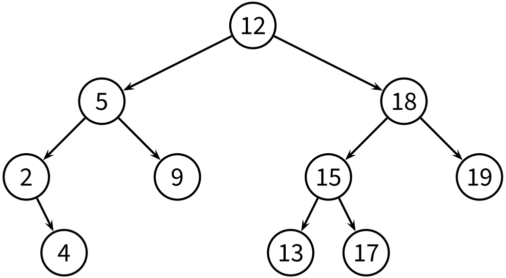
Pre-Order Tree Walk

Pre-Order Tree Walk

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

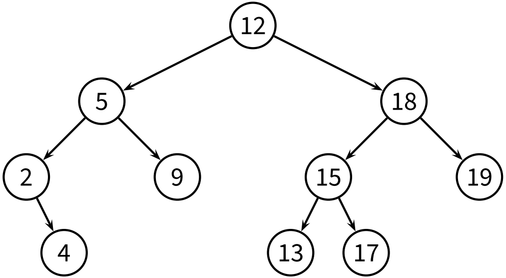
Pre-Order Tree Walk

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



Pre-Order Tree Walk

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



12 5 2 4 9 18 15 13 17 19

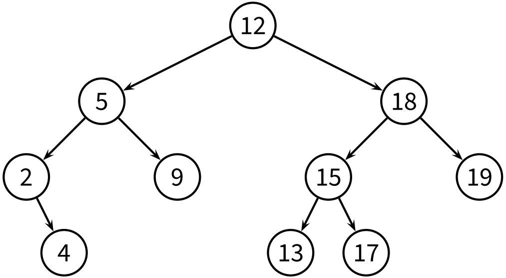
Post-Order Tree Walk

Post-Order Tree Walk

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

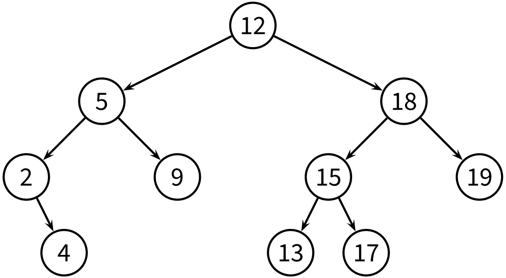
Post-Order Tree Walk

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



Post-Order Tree Walk

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



4 2 9 5 13 17 15 19 18 12

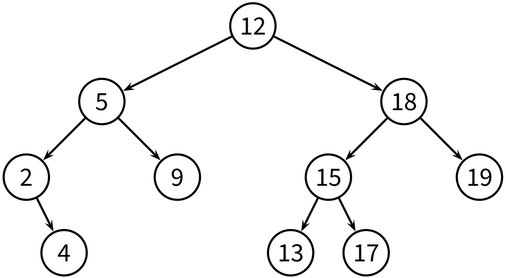
Reverse-Order Tree Walk

Reverse-Order Tree Walk

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

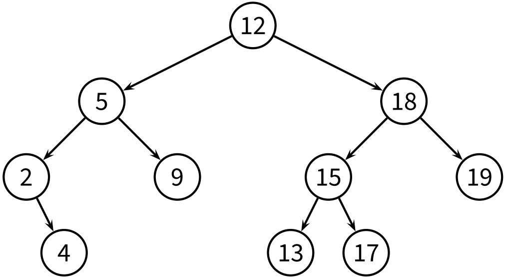
Reverse-Order Tree Walk

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



Reverse-Order Tree Walk

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



19 18 17 15 13 12 9 5 4 2

Complexity of Tree Walks

- The general recurrence is

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

Complexity of Tree Walks

- 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
  - $y \in \mathit{left\text{-}subtree}(x) \Rightarrow y.\mathit{key} \leq x.\mathit{key}$
  - $z \in \mathit{right\text{-}subtree}(x) \Rightarrow z.\mathit{key} \geq x.\mathit{key}$

Minimum and Maximum Keys

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

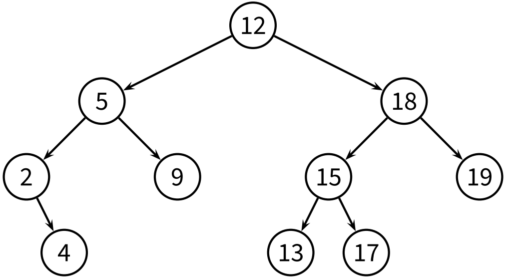
```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

Successor and Predecessor

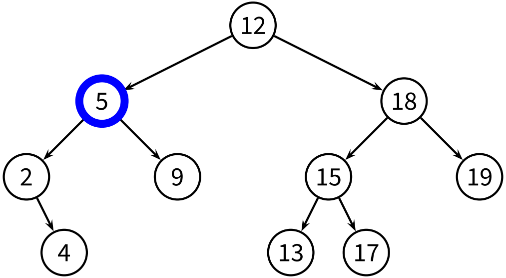
- Given a node *x*, find the node containing the next key value

Successor and Predecessor

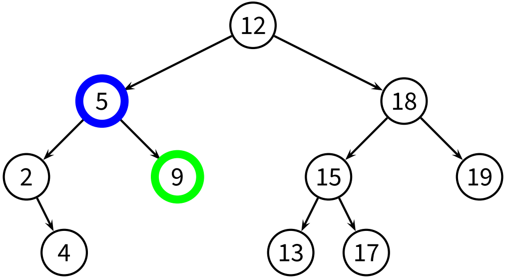
- Given a node *x*, find the node containing the next key value



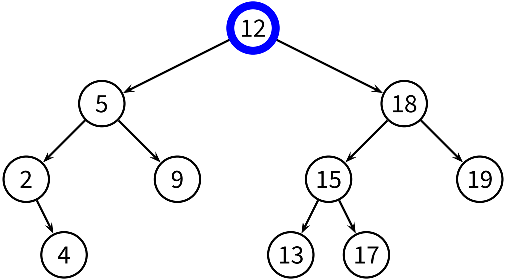
- Given a node *x*, find the node containing the next key value



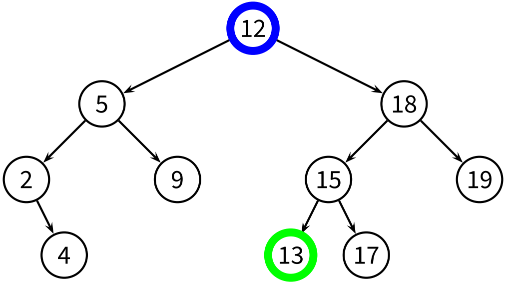
- Given a node *x*, find the node containing the next key value



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

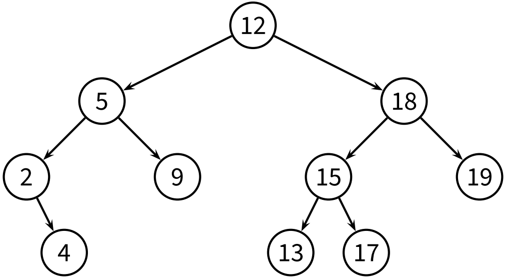


- Given a node *x*, find the node containing the next key value



Successor and Predecessor

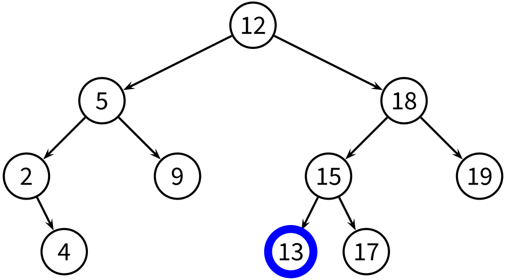
- Given a node *x*, find the node containing the next key value



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

Successor and Predecessor

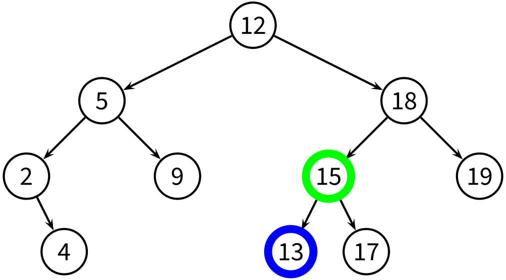
- Given a node *x*, find the node containing the next key value



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

Successor and Predecessor

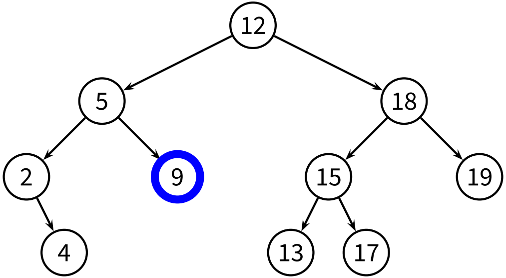
- Given a node *x*, find the node containing the next key value



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

Successor and Predecessor

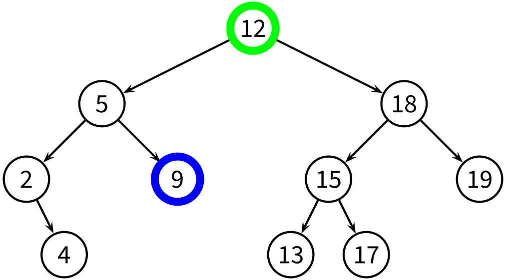
- Given a node *x*, find the node containing the next key value



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

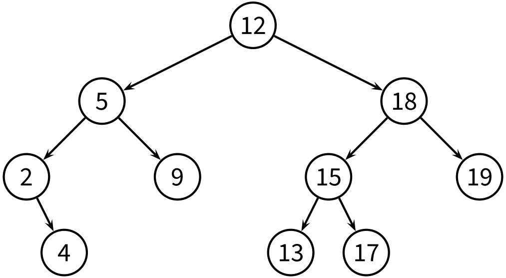
Successor and Predecessor

- 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

- 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

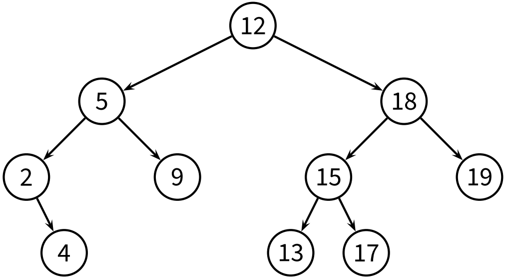
- Otherwise it is the *first ancestor a* of *x* such that *x* falls in the *left* subtree of *a*

Successor and Predecessor(2)

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

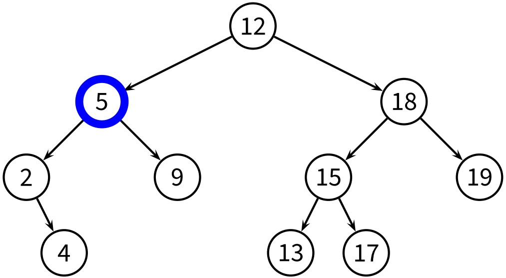
Successor and Predecessor(2)

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



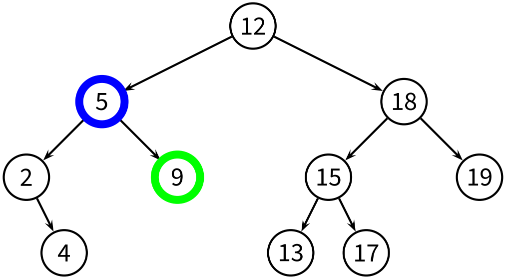
Successor and Predecessor(2)

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



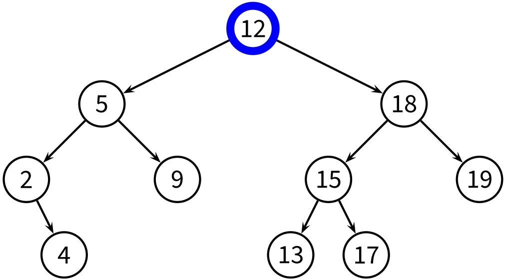
Successor and Predecessor(2)

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



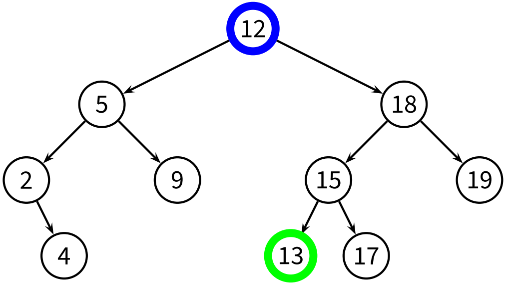
Successor and Predecessor(2)

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



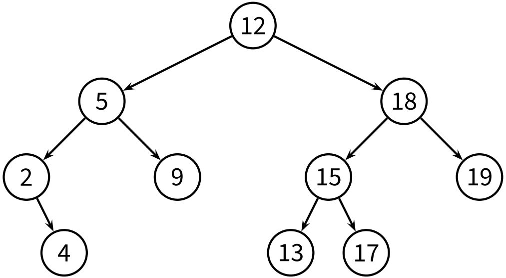
Successor and Predecessor(2)

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



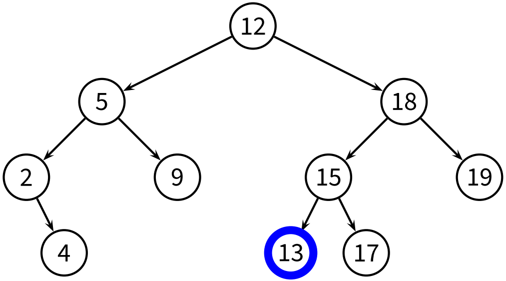
Successor and Predecessor(2)

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



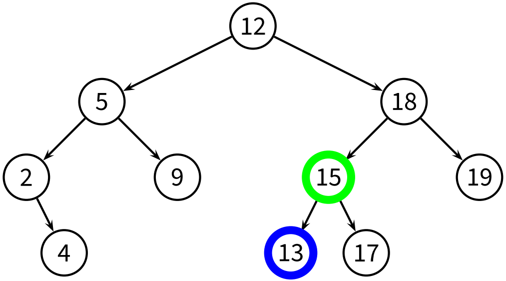
Successor and Predecessor(2)

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



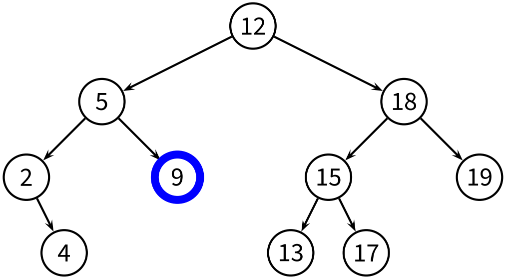
Successor and Predecessor(2)

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



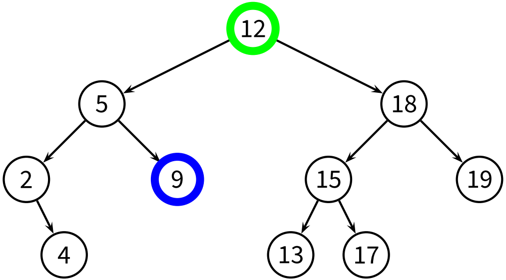
Successor and Predecessor(2)

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



Successor and Predecessor(2)

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



Search

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

Search

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

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

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

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

- Is this correct?

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

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

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

Search

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

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

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

Search

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

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

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

- Complexity?

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

Search

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

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

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

- Complexity?

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

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

- Iterative *binary search*

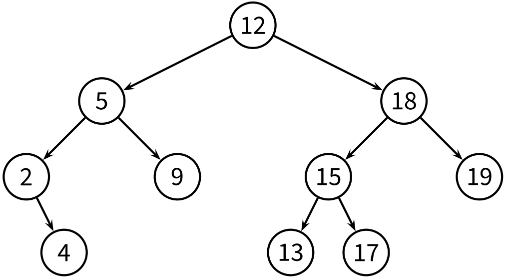
Search (2)

- Iterative *binary search*

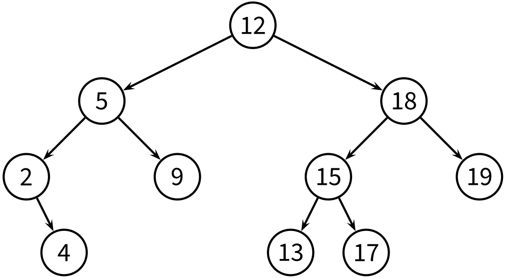
```
ITERATIVE-TREE-SEARCH(T, k)
x = T.root
while x ≠ NIL ∧ k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Insertion

Insertion



Insertion



- *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

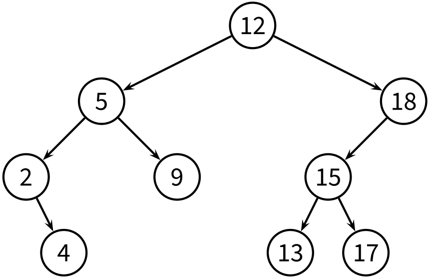
Insertion (2)

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

Insertion (2)

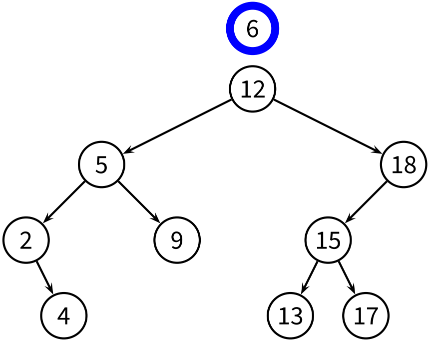
Tree-Insert(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y = NIL
    T.root = z
else if z.key < y.key
        y.left = z
    else y.right = z
```



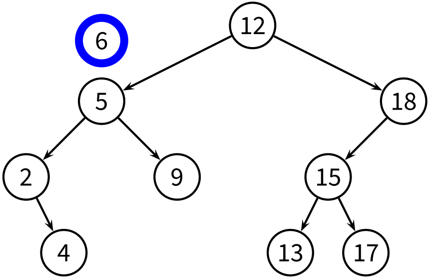
Insertion (2)

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



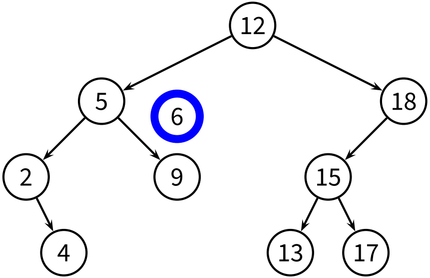
Insertion (2)

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

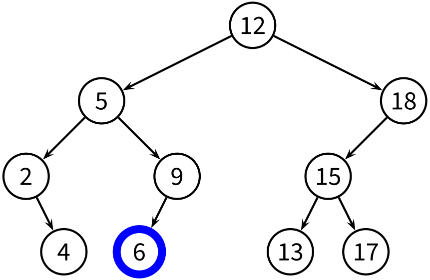


Insertion (2)

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

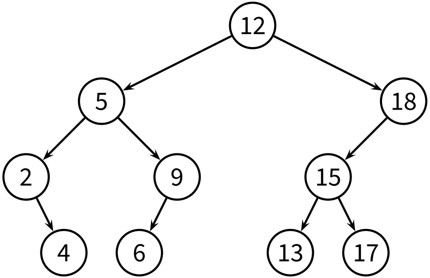


```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
        y = x
        if z.key < x.key
                x = x.left
        else x = x.right
z.parent = y
if y = NIL
        T.root = z
else if z.key < y.key
                y.left = z
        else y.right = z
```



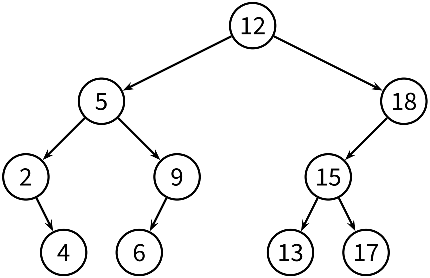
Insertion (2)

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



Insertion (2)

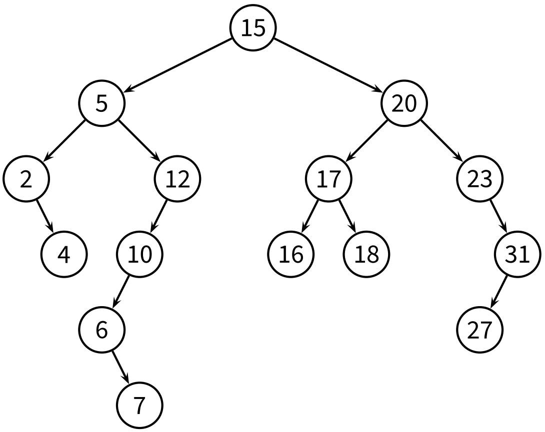
```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y = NIL
    T.root = z
else if z.key < y.key
    y.left = z
else y.right = z
```

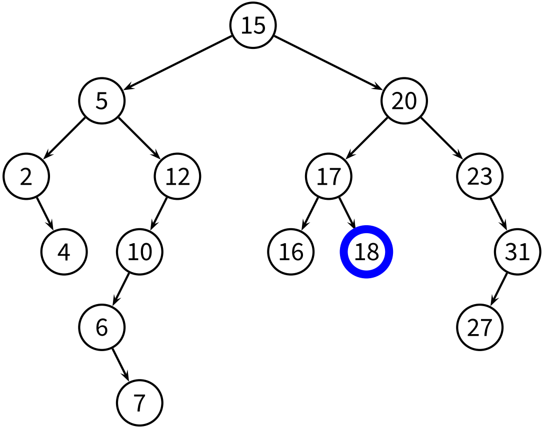


$$T(n) = \Theta(h)$$

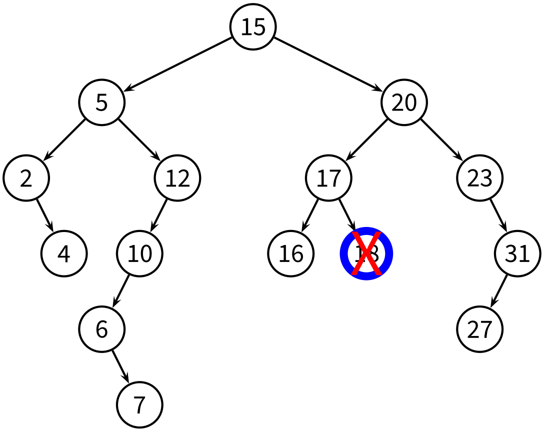
Deletion

Deletion



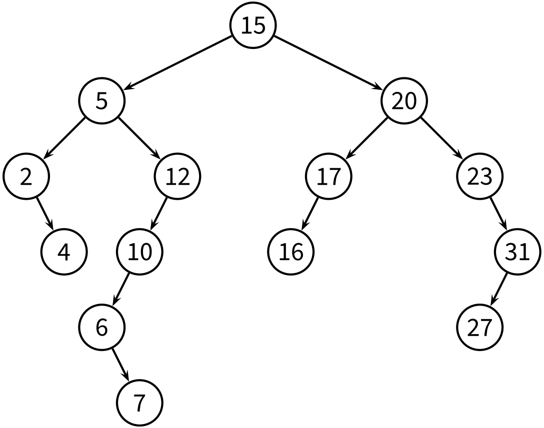


1. z has no children



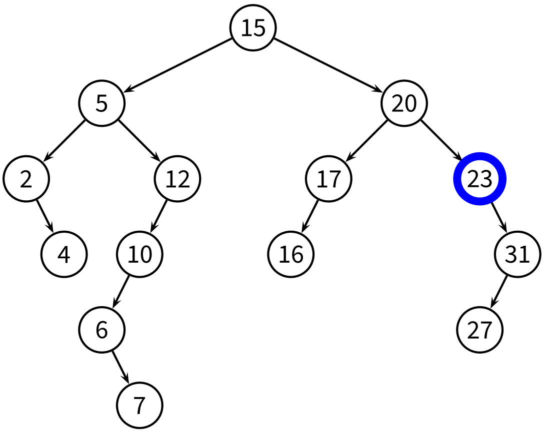
1. *z* has no children
   - simply remove *z*

Deletion

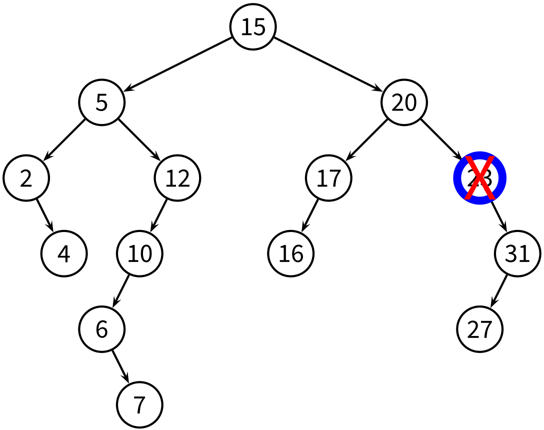


1. *z* has no children
   - simply remove *z*

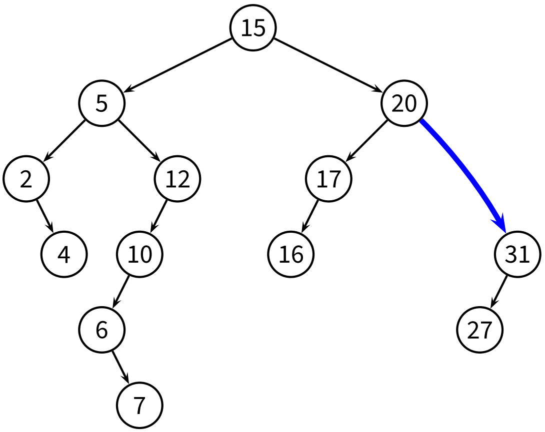
Deletion



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

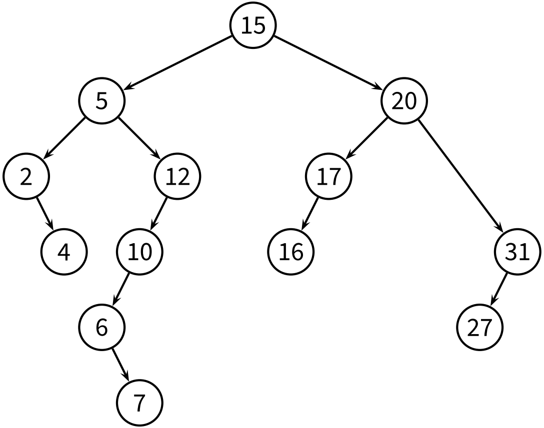


1. *z* has no children
   - simply remove *z*
2. *z* has one child
   - remove *z*

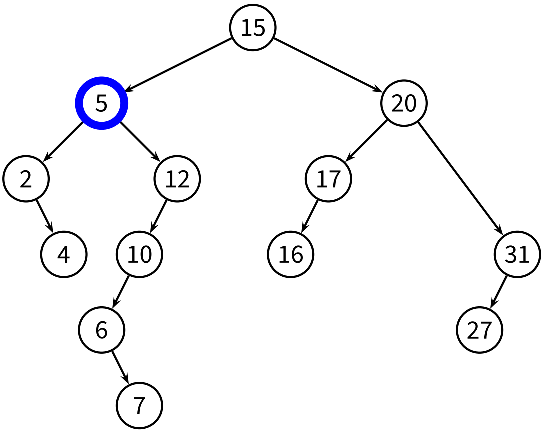


1. *z* has no children
   - simply remove *z*
2. *z* has one child
   - remove *z*
   - connect $z.parent$ to $z.right$

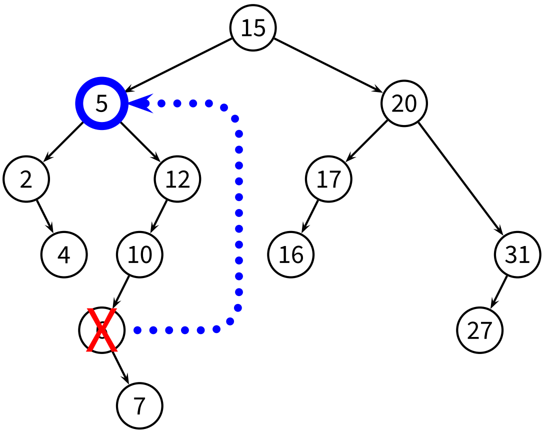
Deletion



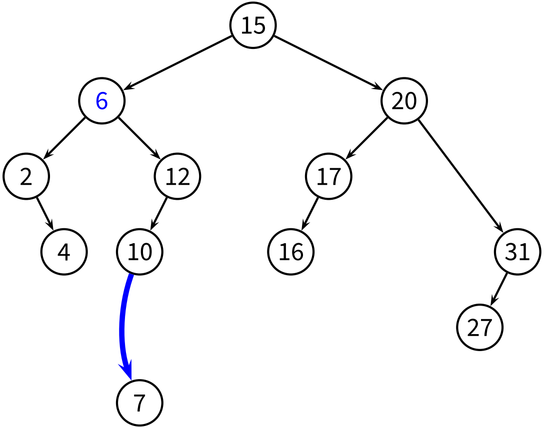
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$



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 = $ **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 $y = \textbf{Tree-Successor}(z)$
   - remove y (1 child!)
   - connect $y.parent$ to $y.right$

Deletion (2)

```
TREE-DELETE(T, z)
if z.left = NIL or z.right = NIL
        y = z
else y = TREE-SUCCESSOR(z)
if y.left ≠ NIL
        x = y.left
else x = y.right
if x ≠ NIL
        x.parent = y.parent
if y.parent == NIL
        T.root = x
else if y = y.parent.left
                y.parent.left = x
        else y.parent.right = x
if y ≠ z
        z.key = y.key
        copy any other data from y into z
```

Complexity

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

Complexity

- 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

Complexity

- 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

Randomized Insertion

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

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

Randomized Insertion

- ***Idea 1:*** insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, 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 $A = \langle 1, 2, 3, \ldots, n \rangle$, 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 $A = \langle 1, 2, 3, \ldots, n \rangle$, 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)
r = uniformly random value from {1, ..., t.size + 1}
if r = 1
        TREE-ROOT-INSERT(T, z)
else TREE-INSERT(T, z)
```

Randomized Insertion (2)

TREE-RANDOMIZED-INSERT1(T, z)

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

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

Randomized Insertion (2)

```
TREE-RANDOMIZED-INSERT1(T, z)
r = uniformly random value from {1, ..., t.size + 1}
if r = 1
        TREE-ROOT-INSERT(T, z)
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

Randomized Insertion (2)

```
TREE-RANDOMIZED-INSERT1(T, z)
r = uniformly random value from {1, ..., t.size + 1}
if r = 1
        TREE-ROOT-INSERT(T, z)
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)
r = uniformly random value from {1, ..., t.size + 1}
if r = 1
        TREE-ROOT-INSERT(T, z)
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)

Randomized Insertion (3)

```
TREE-RANDOMIZED-INSERT(t, z)
if t = NIL
        return z
r = uniformly random value from {1, . . . , t.size + 1}
if r = 1                                    // Pr[r = 1] = 1/(t.size + 1)
        z.size = t.size + 1
        return TREE-ROOT-INSERT(t, z)
if z.key < t.key
        t.left = TREE-RANDOMIZED-INSERT(t.left, z)
else t.right = TREE-RANDOMIZED-INSERT(t.right, z)
t.size = t.size + 1
return t
```

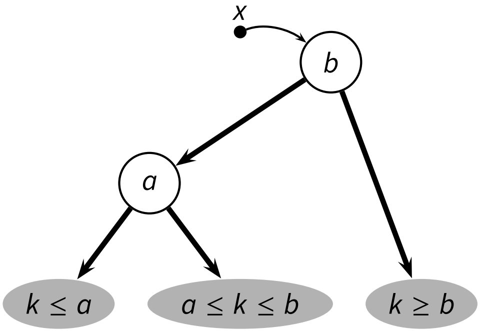
Randomized Insertion (3)

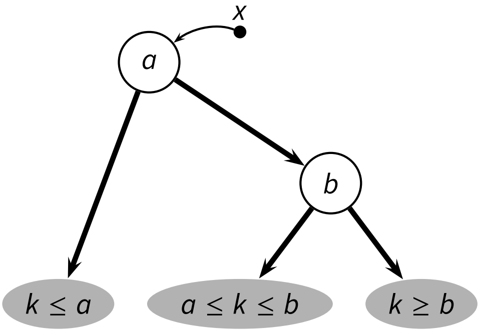
```
TREE-RANDOMIZED-INSERT(t, z)
if t = NIL
        return z
r = uniformly random value from {1, . . . , t.size + 1}
if r = 1                              // Pr[r = 1] = 1/(t.size + 1)
        z.size = t.size + 1
        return TREE-ROOT-INSERT(t, z)
if z.key < t.key
        t.left = TREE-RANDOMIZED-INSERT(t.left, z)
else t.right = TREE-RANDOMIZED-INSERT(t.right, z)
t.size = t.size + 1
return t
```

- Looks like this one really simulates a random permutation…

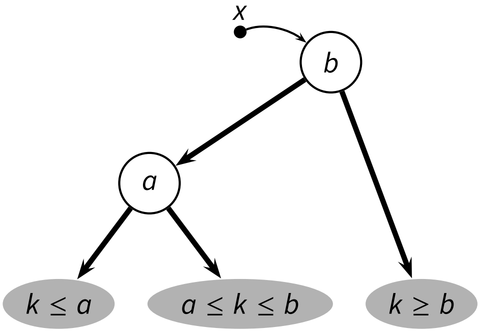
Rotation

Rotation



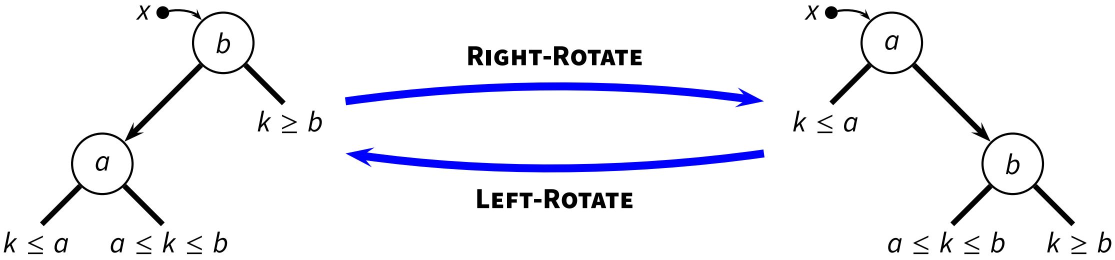


- x = **Right-Rotate**(x)



- x = **Right-Rotate**(x)
- x = **Left-Rotate**(x)

Rotation

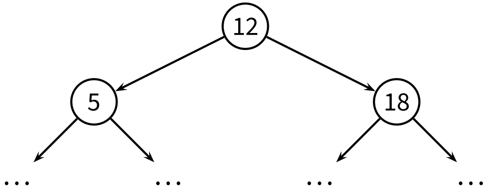


RIGHT-ROTATE(x)

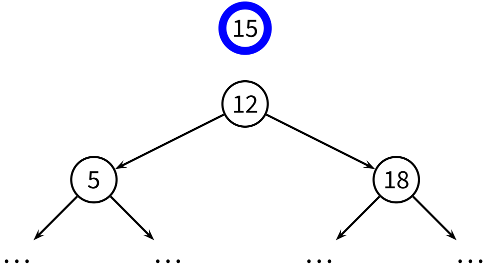
```
l = x.left
x.left = l.right
l.right = x
return l
```

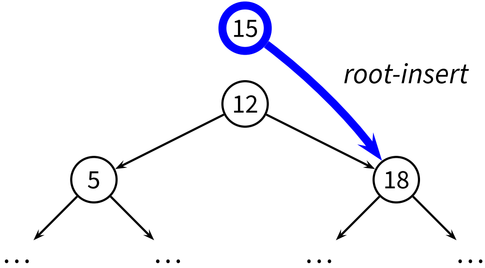
LEFT-ROTATE(x)

```
r = x.right
x.right = r.left
r.left = x
return r
```

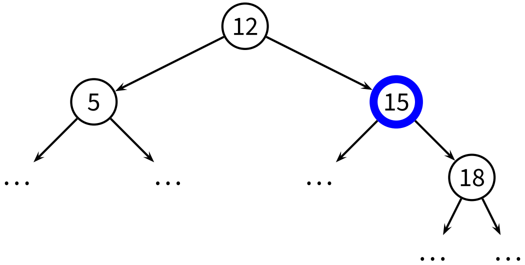

12
5
18
...
...
...
...

Root Insertion

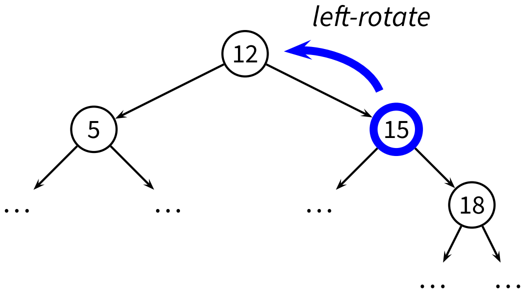




1. Recursively insert z at the root of the appropriate subtree (right)

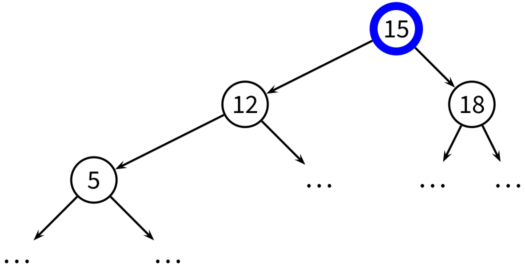


1. Recursively insert z at the root of the appropriate subtree (right)



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)

Root Insertion (2)

```
TREE-ROOT-INSERT(x, z)
if x = NIL
        return z
if z.key < x.key
        x.left = TREE-ROOT-INSERT(x.left, z)
        return RIGHT-ROTATE(x)
else x.right = TREE-ROOT-INSERT(x.right, z)
        return LEFT-ROTATE(x)
```

Observation

- General strategies to deal with complexity in the worst case

Observation

- 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

Observation

- 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