Binary Search Trees

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

April 27, 2021

Outline

- Binary search trees
- Randomized binary search trees

Binary Search Tree

- 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.

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.
  - **Tree-Minimum**(T) finds the smallest element in the tree
  - **Tree-Maximum**(T) finds the largest element in the tree

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.
  - **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

Binary Search Tree (2)

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

Binary Search Tree (2)

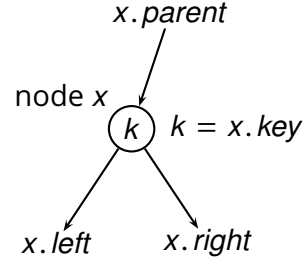
- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***
  - $T.root$ is the root node of tree T
  - or sometimes T refers directly to the root node

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

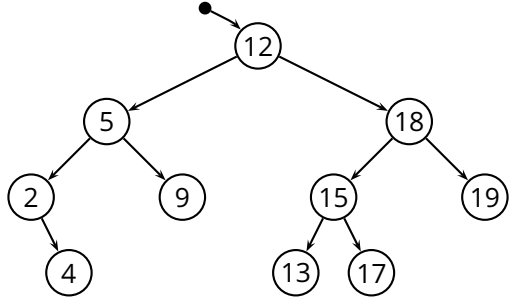
Node x

- $x.parent$ 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

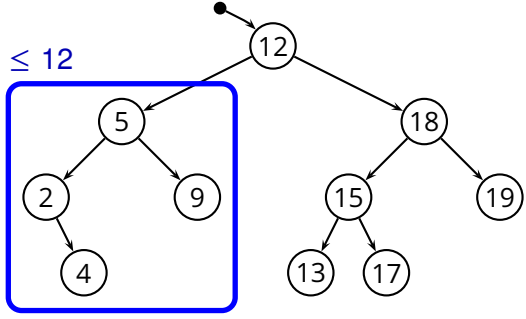


Binary Search Tree (3)

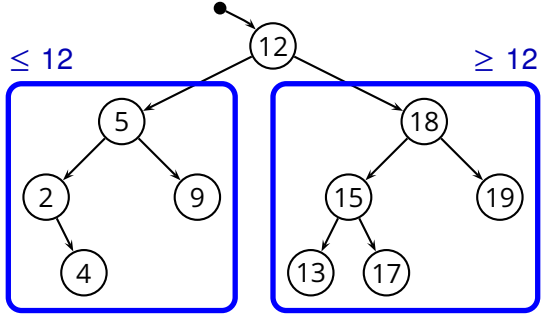
Binary Search Tree (3)



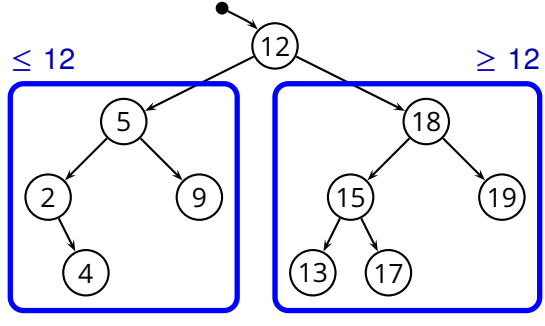
Binary Search Tree (3)



Binary Search Tree (3)



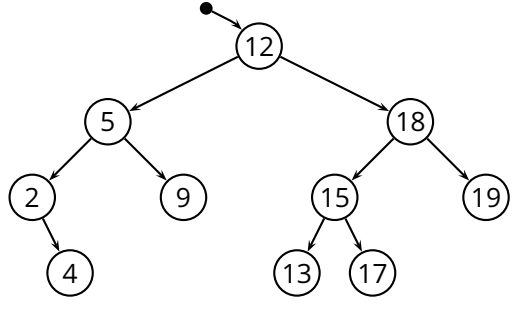
Binary Search Tree (3)



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

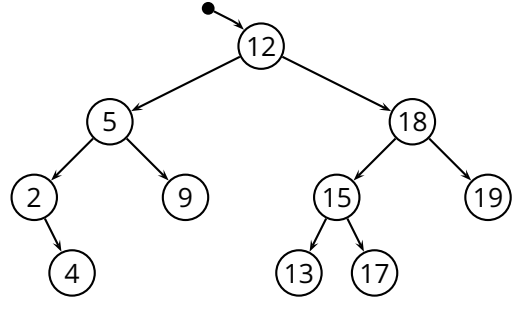
In-Order Tree Walk

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



In-Order Tree Walk

- 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)
1  if x ≠ nil
2       Inorder-Tree-Walk(x.left)
3       print x.key
4       Inorder-Tree-Walk(x.right)
```

In-Order Tree Walk (2)

- 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)
```

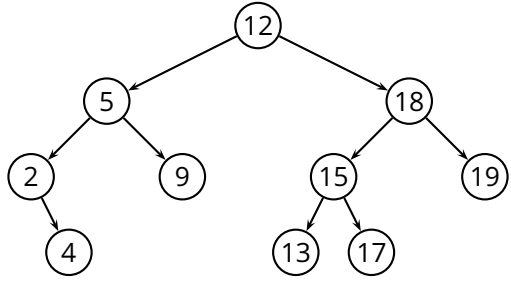
Pre-Order Tree Walk

Pre-Order Tree Walk

```
Preorder-Tree-Walk(x)
1  if x ≠ nil
2        print x.key
3        Preorder-Tree-Walk(x.left)
4        Preorder-Tree-Walk(x.right)
```

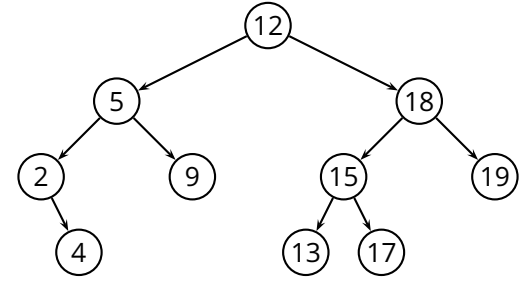
Pre-Order Tree Walk

```
Preorder-Tree-Walk(x)
1  if x ≠ nil
2        print x.key
3        Preorder-Tree-Walk(x.left)
4        Preorder-Tree-Walk(x.right)
```



Pre-Order Tree Walk

```
Preorder-Tree-Walk(x)
1  if x ≠ nil
2        print x.key
3        Preorder-Tree-Walk(x.left)
4        Preorder-Tree-Walk(x.right)
```



12 5 2 4 9 18 15 13 17 19

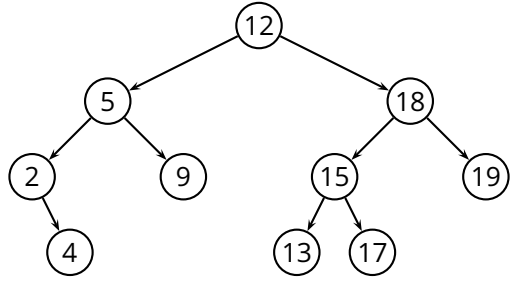
Post-Order Tree Walk

Post-Order Tree Walk

```
Postorder-Tree-Walk(x)
1  if x ≠ nil
2        Postorder-Tree-Walk(x.left)
3        Postorder-Tree-Walk(x.right)
4        print x.key
```

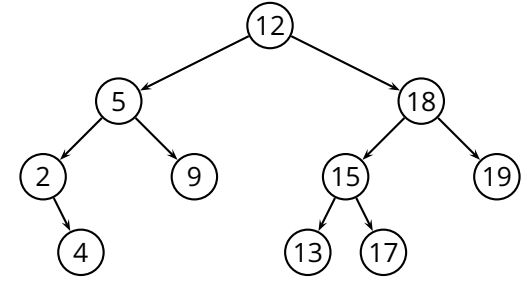
Post-Order Tree Walk

```
Postorder-Tree-Walk(x)
1   if x ≠ nil
2         Postorder-Tree-Walk(x.left)
3         Postorder-Tree-Walk(x.right)
4         print x.key
```



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
```



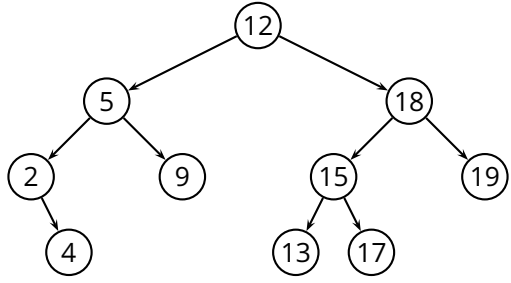
Reverse-Order Tree Walk

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)
```

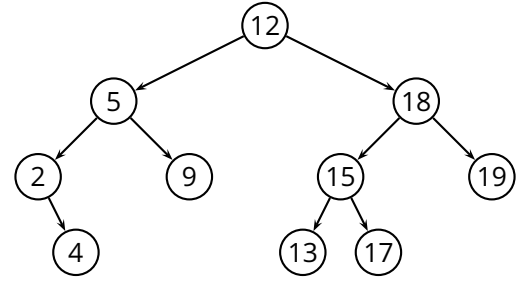
Reverse-Order Tree Walk

```
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)
```



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

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	$\Theta(n)$

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	$\Theta(n)$

We could prove this using the *substitution method*

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	$\Theta(n)$

  We could prove this using the *substitution method*

- Can we do better?

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	$\Theta(n)$

  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

Minimum and Maximum Keys

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

Minimum and Maximum Keys

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

Tree-Maximum(x)

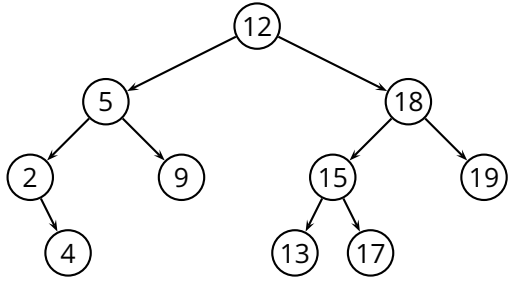
```
1  while x.right ≠ nil
2        x = x.right
3  return x
```

Successor and Predecessor

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

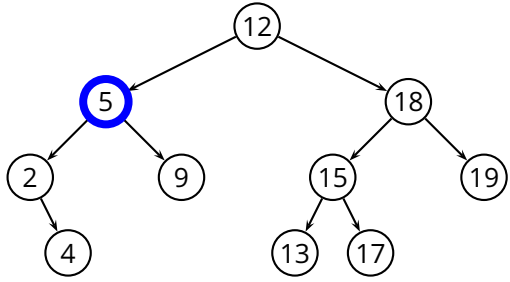
Successor and Predecessor

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



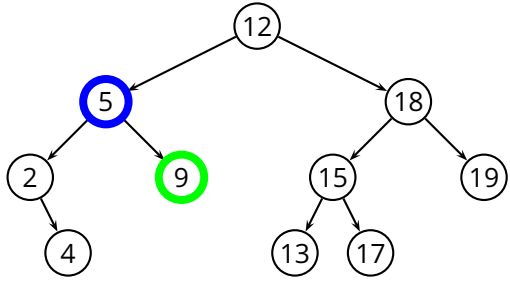
Successor and Predecessor

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



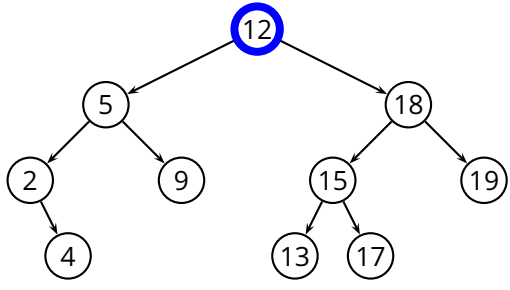
Successor and Predecessor

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



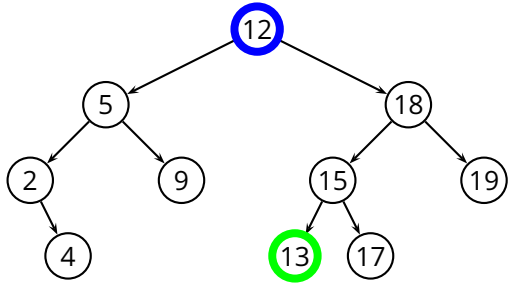
Successor and Predecessor

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



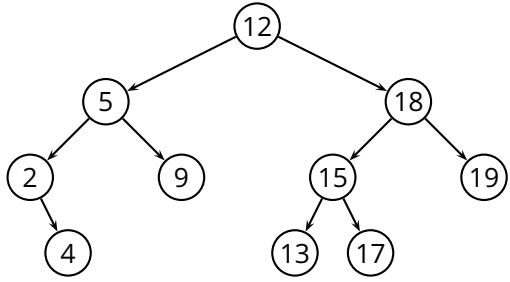
Successor and Predecessor

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



Successor and Predecessor

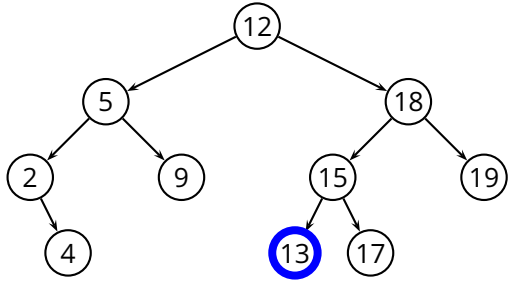
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

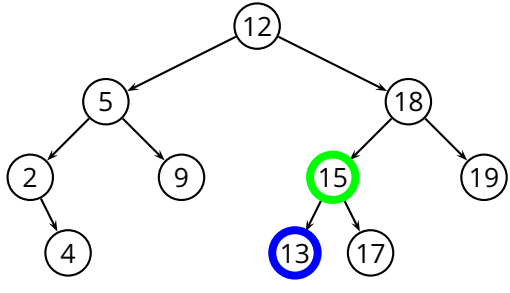
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

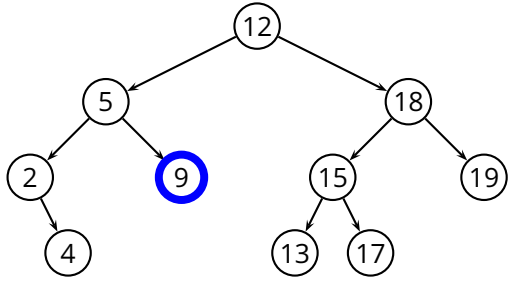
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

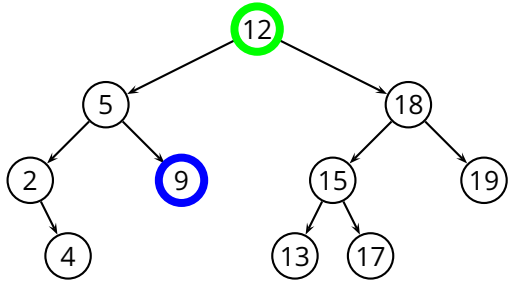
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

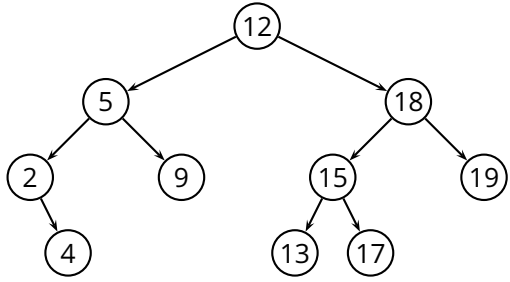
- Given a node x, find the node containing the next key value



- 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



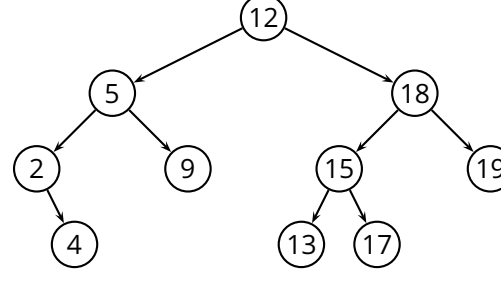
- The successor of x is the *minimum* of the *right* subtree of x, if that exists
- Otherwise it is the *first ancestor* a of x such that x falls in the *left* subtree of a

Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```

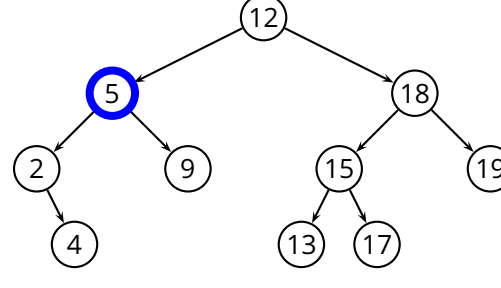
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



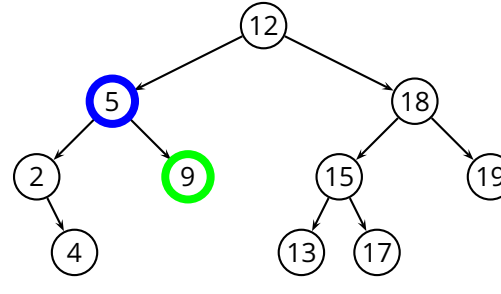
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



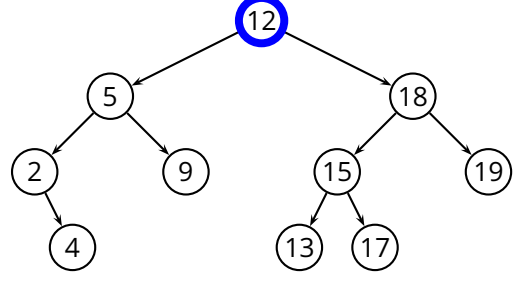
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



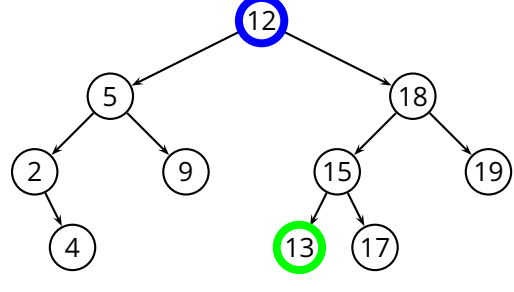
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



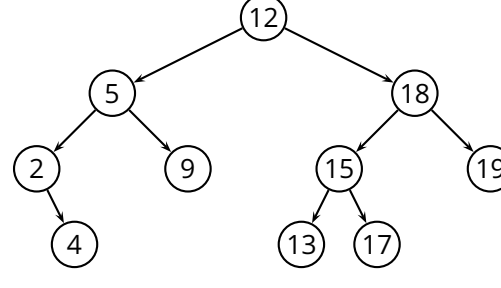
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



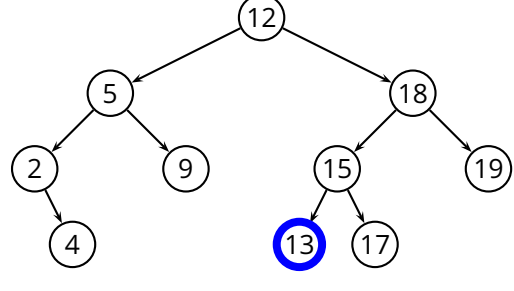
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



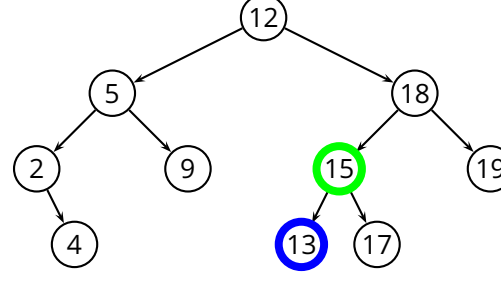
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



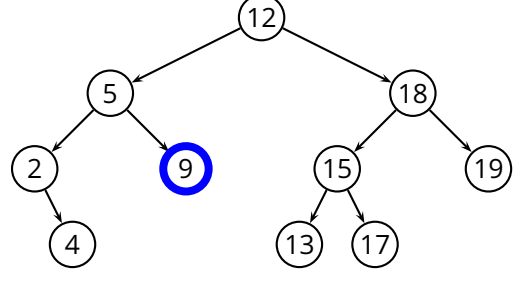
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



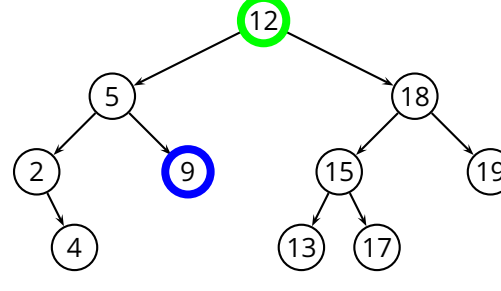
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ 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 ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



Search

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

- *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)
```

- *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?

- *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*

- *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*
- Complexity?

- *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*
- Complexity?

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

- *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*
- Complexity?

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

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

Search (2)

Search (2)

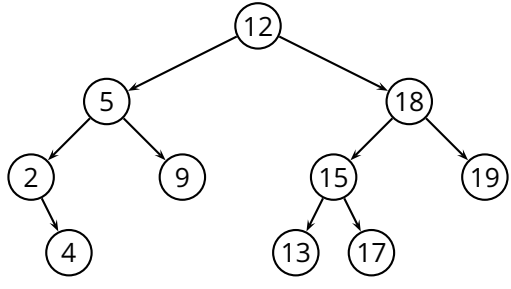
- Iterative *binary search*

- Iterative *binary search*

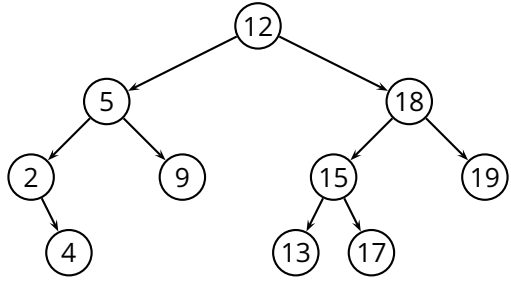
```
Iterative-Tree-Search(T, k)
1  x = T.root
2  while x ≠ nil ∧ k ≠ x.key
3        if k < x.key
4              x = x.left
5        else x = x.right
6  return x
```

Insertion

Insertion



Insertion



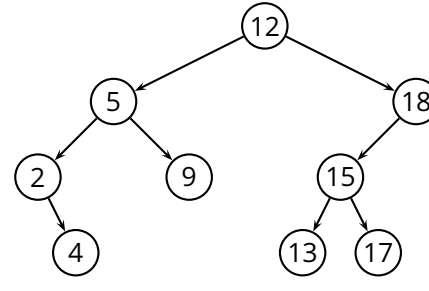
- *Idea*
  - in order to insert x, we *search* for x (more precisely $x.key$)
  - if we don't find it, we add it where the search stopped

Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```

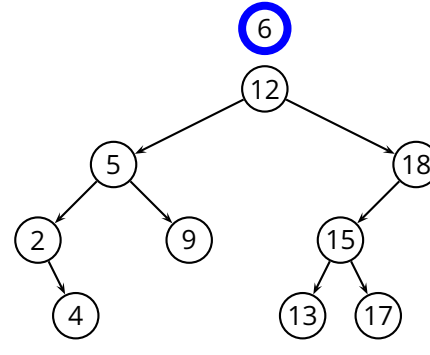
Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



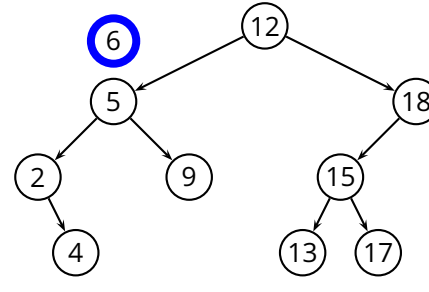
Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



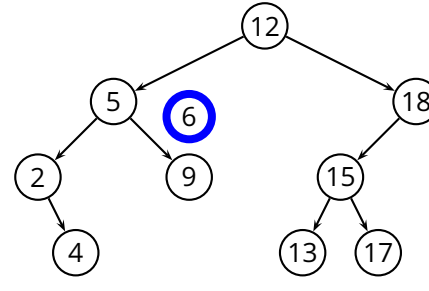
Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



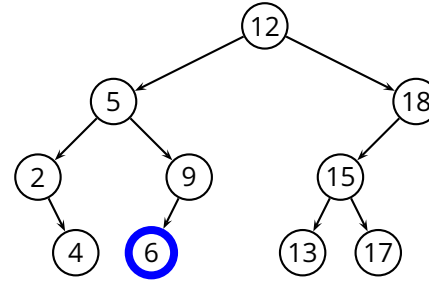
Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



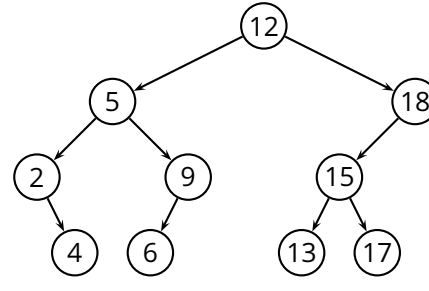
Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



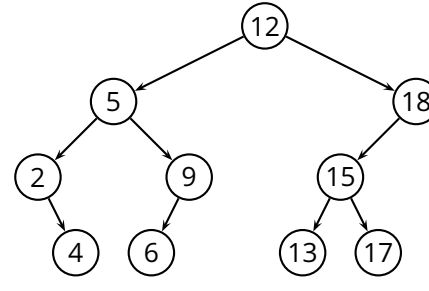
Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



Insertion (2)

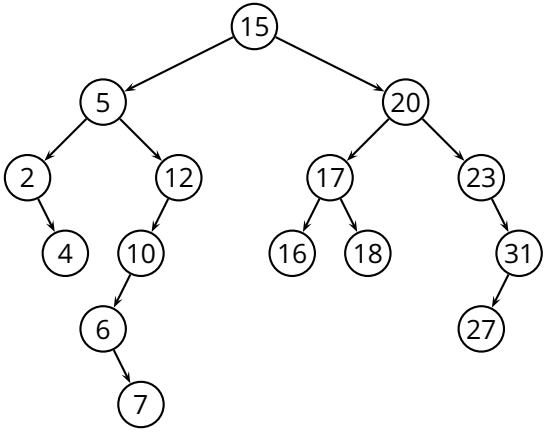
```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



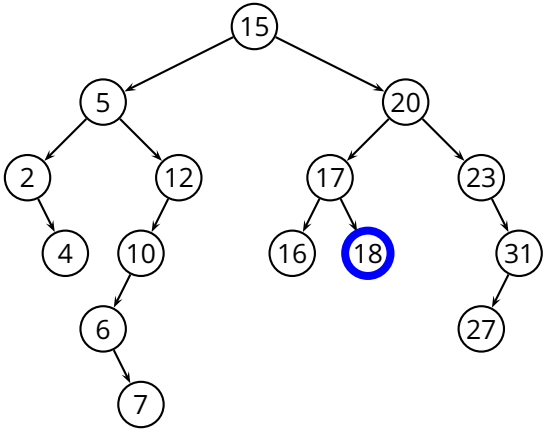
$T(n) = \Theta(h)$

Deletion

Deletion

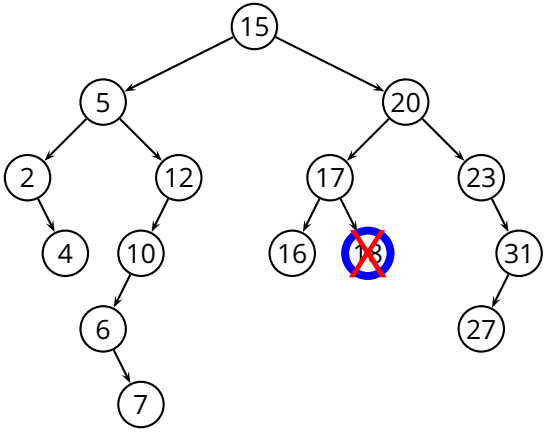


Deletion



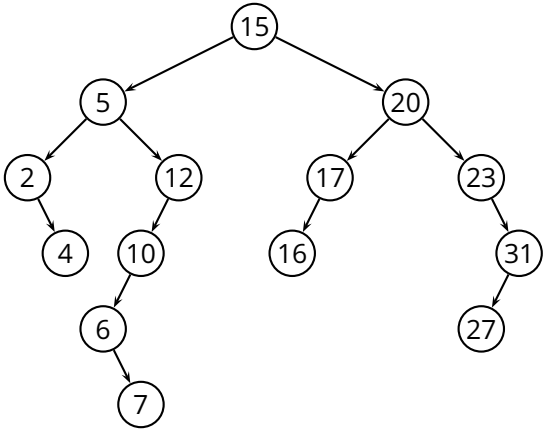
1. z has no children

Deletion



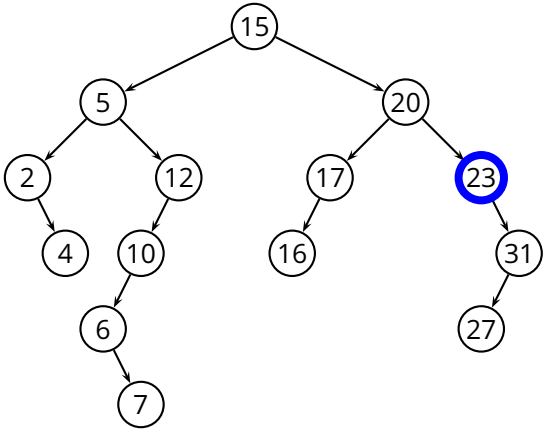
1. z has no children
   - simply remove z

Deletion



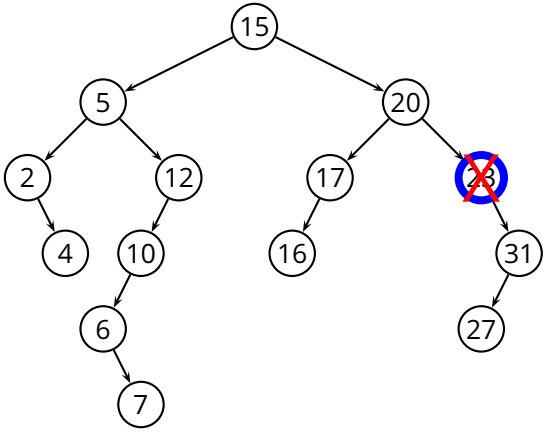
1. z has no children
   - simply remove z

Deletion



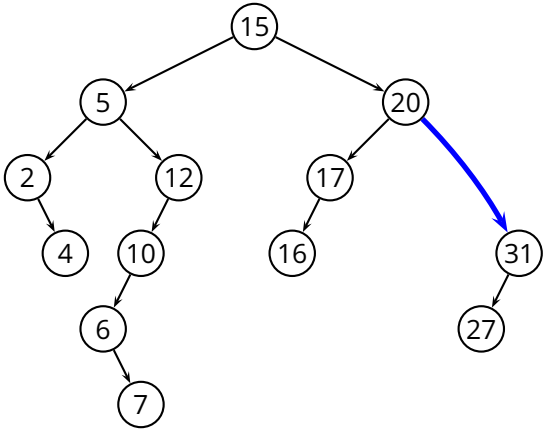
1. z has no children
   - simply remove z
2. z has one child

Deletion



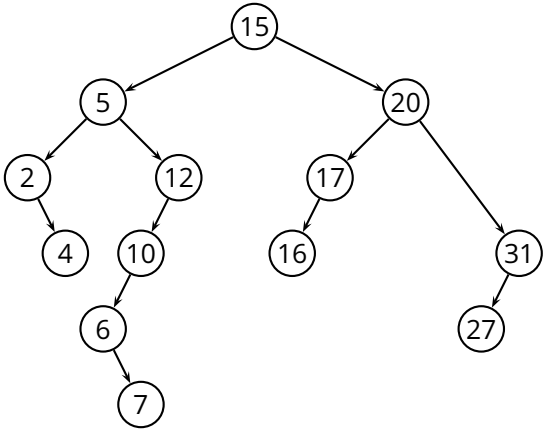
1. z has no children
   - simply remove z
2. z has one child
   - remove z

Deletion



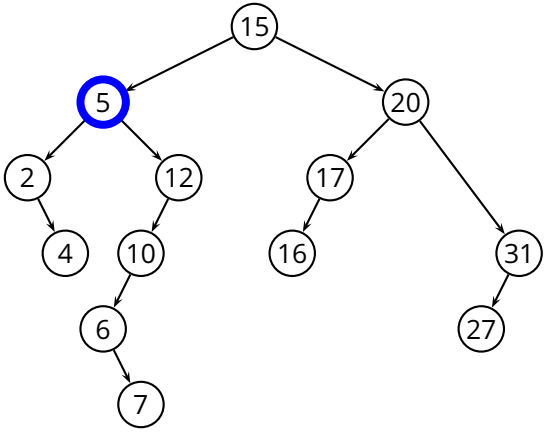
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$

Deletion



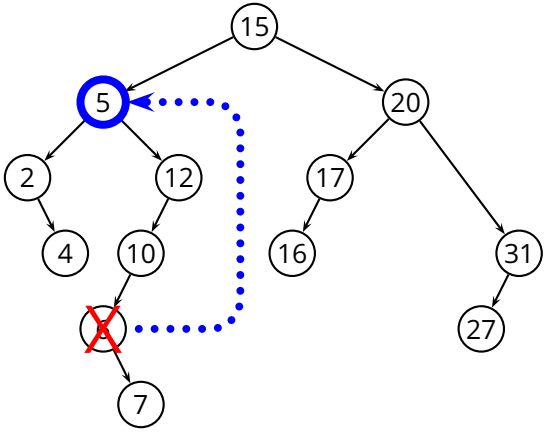
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$

Deletion



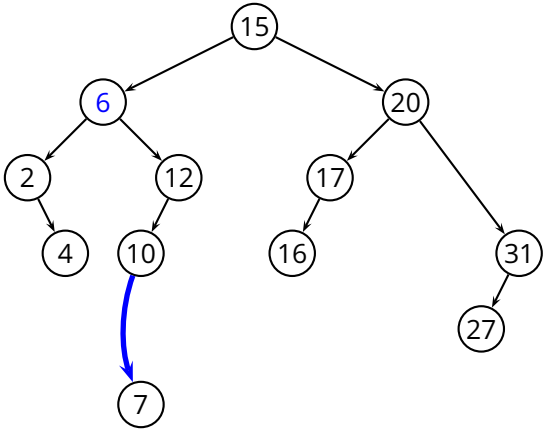
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$
3. z has two children

Deletion



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.\mathit{parent}$ to $z.\mathit{right}$
3. z has two children
   - replace z with $y = \textbf{Tree-Successor}(z)$
   - remove y (1 child!)

Deletion



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)
 1  if z.left = nil or z.right = nil
 2        y = z
 3  else y = Tree-Successor(z)
 4  if y.left ≠ nil
 5        x = y.left
 6  else x = y.right
 7  if x ≠ nil
 8        x.parent = y.parent
 9  if y.parent == 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 ≠ z
15        z.key = y.key
16        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

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
  - i.e., with ordered sequences

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
  - i.e., with ordered sequences
  - the problem is that the “worst” case is not that uncommon

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
  - 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

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

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

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 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

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 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, ..., t.size + 1}
2  if r = 1
3        Tree-Root-Insert(T, z)
4  else Tree-Insert(T, z)
```

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly random value from {1, ..., 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?

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly random value from {1, ..., 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

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly random value from {1, ..., 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

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly random value from {1, ..., 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)

Randomized Insertion (3)

```
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
```

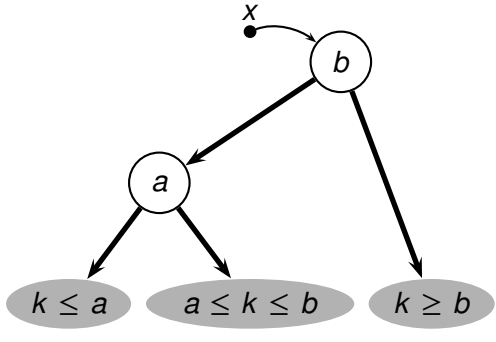
Randomized Insertion (3)

```
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
```

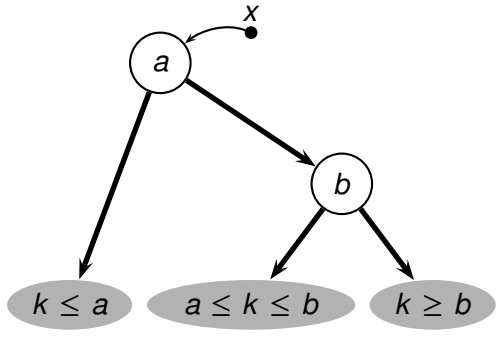
- Looks like this one really simulates a random permutation...

Rotation

Rotation

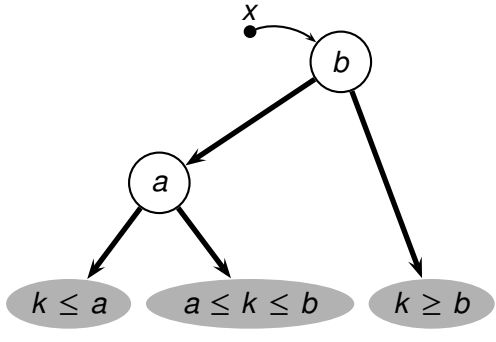


Rotation



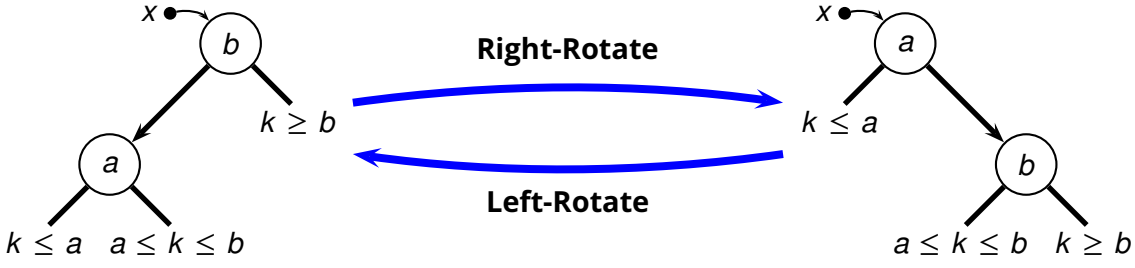
- $x =$ **Right-Rotate**(x)

Rotation



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

Rotation



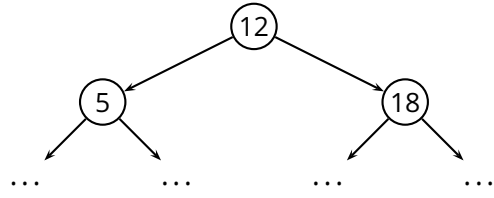
Right-Rotate(x)

```
1  l = x.left
2  x.left = l.right
3  l.right = x
4  return l
```

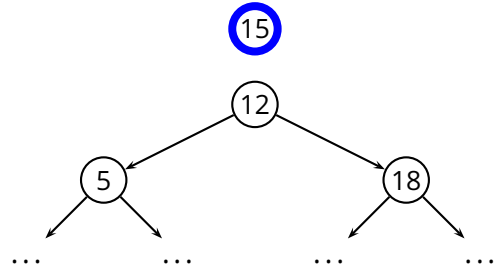
Left-Rotate(x)

```
1  r = x.right
2  x.right = r.left
3  r.left = x
4  return r
```

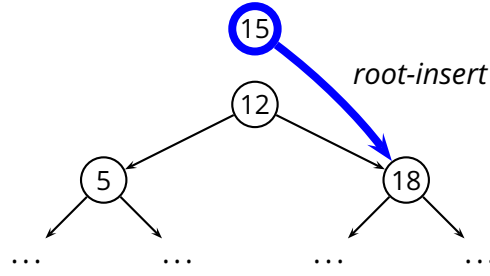
Root Insertion



Root Insertion

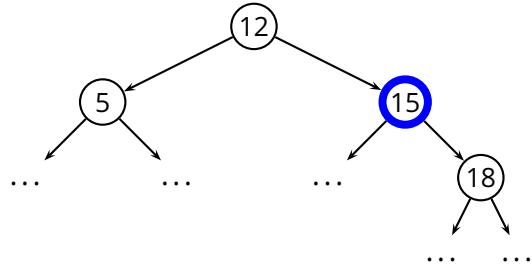


Root Insertion



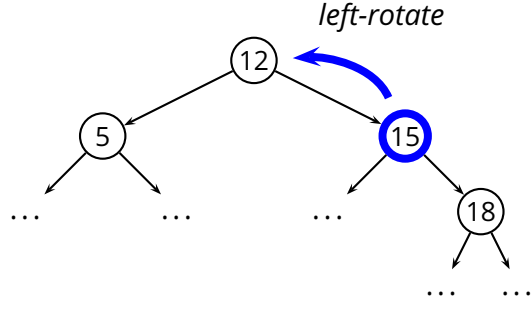
1. Recursively insert z at the root of the appropriate subtree (right)

Root Insertion



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

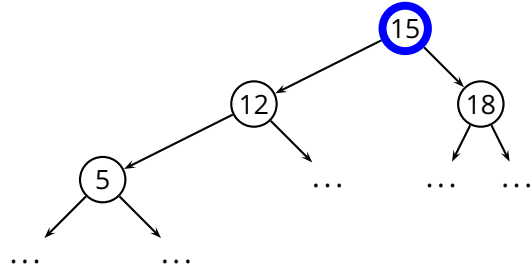
Root Insertion



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

2. Rotate *x* with *z* (left-rotate)

Root Insertion



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)
1  if x = nil
2        return z
3  if z.key < x.key
4        x.left = Tree-Root-Insert(x.left, z)
5        return Right-Rotate(x)
6  else x.right = Tree-Root-Insert(x.right, z)
7        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

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

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