Algorithms and Data Structures (II)

Gabriel Istrate

March 18, 2020

- Wrap up hash tables.
- Skip lists.
- Binary search trees
- Randomized binary search trees

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*

- *Interface* (generic interface)
  - **INSERT**(D, k) adds a key k to the dictionary D
  - **DELETE**(D, k) removes key k from D
  - **SEARCH**(D, k) tells whether D contains a key k

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*
- *Interface* (generic interface)
  - **INSERT**(D, k) adds a key k to the dictionary D
  - **DELETE**(D, k) removes key k from D
  - **SEARCH**(D, k) tells whether D contains a key k
- *Implementation (so far)*
  - direct access tables. Linked lists. Hash tables. Skip Lists. Binary Search Trees.

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

- A ***binary search tree*** implements of 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 of 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:* **INORDER-TREE-WALK**(T), etc.

- A ***binary search tree*** implements of 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:* **INORDER-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 of 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:* **INORDER-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

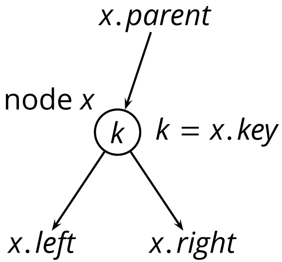
- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***

- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***
  - $T.root$ is the root node of tree T

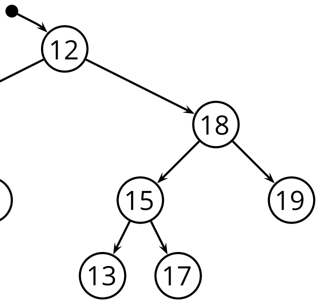
- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***
  - $T.root$ is the root node of tree T

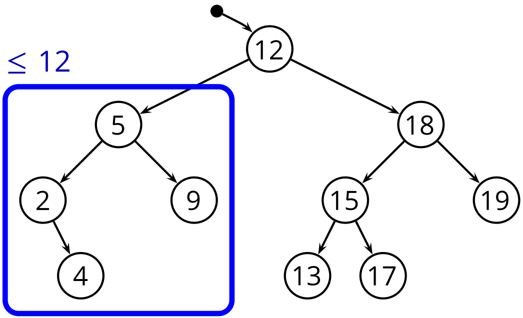
Node x

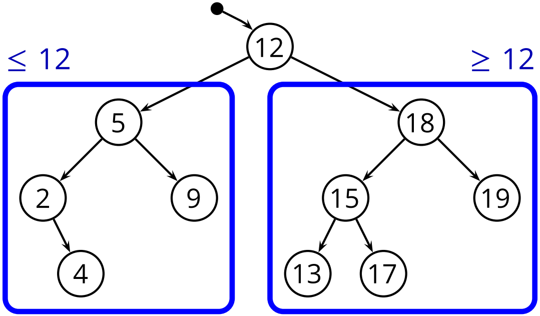
- $x.parent$ is the parent of node x
- $x.key$ is the key stored in node x
- $x.left$ is the left child of node x
- $x.right$ is the right child of node x



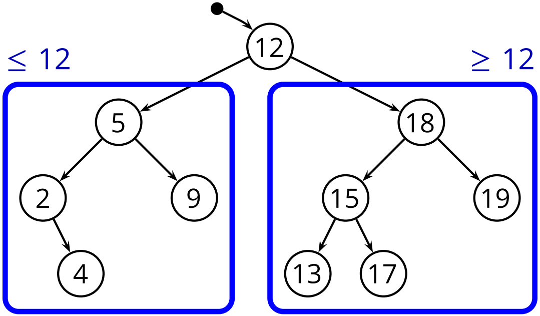
Binary Search Trees (3)


12
5
18
2
9
15
19
4
13
17


≤ 12
12
5
18
2
9
15
19
4
13
17



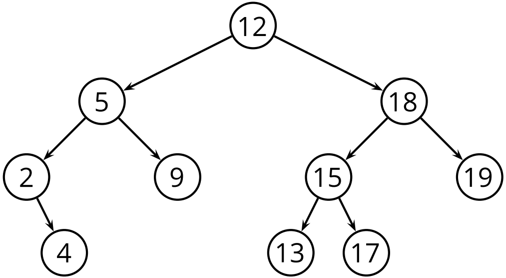
12
≤ 12
≥ 12
5
18
2
9
15
19
4
13
17



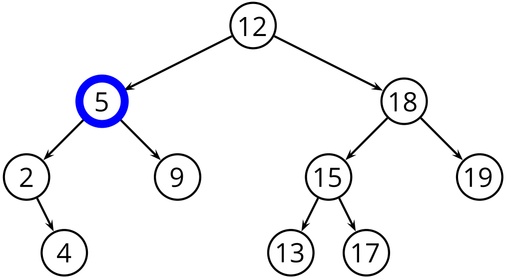
- ***Binary-search-tree property***
  - for all nodes x, y, and z
  - $y \in \textit{left-subtree}(x) \Rightarrow y.\textit{key} \leq x.\textit{key}$
  - $z \in \textit{right-subtree}(x) \Rightarrow z.\textit{key} \geq x.\textit{key}$

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

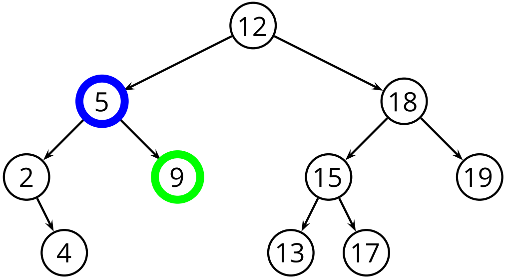
- Given a node *x*, find the node containing the next key value



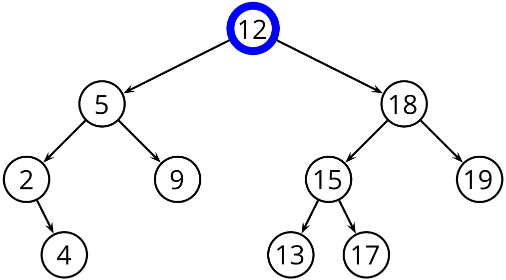
- Given a node *x*, find the node containing the next key value



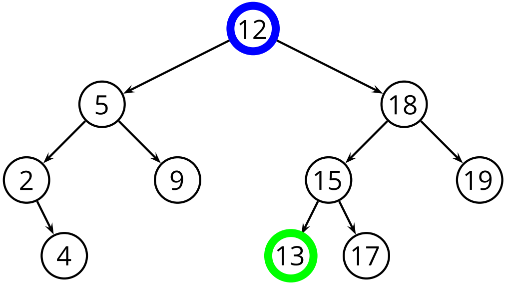
- Given a node x, find the node containing the next key value



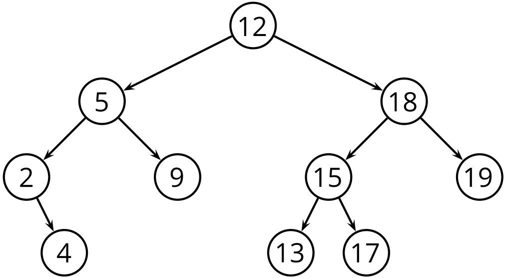
- Given a node *x*, find the node containing the next key value



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

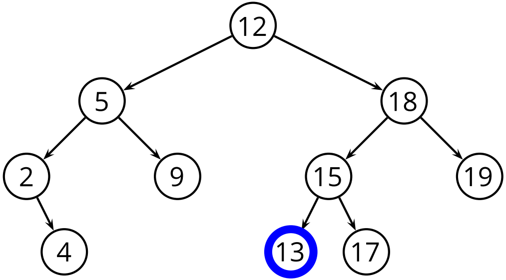


- Given a node x, find the node containing the next key value



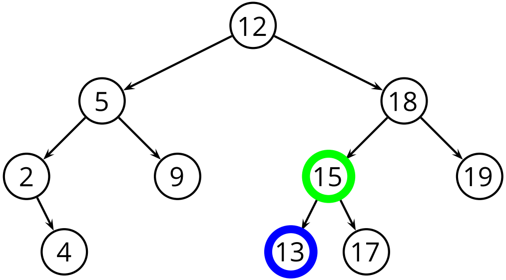
- The successor of x is the *minimum* of the *right* subtree of x, if that exists

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



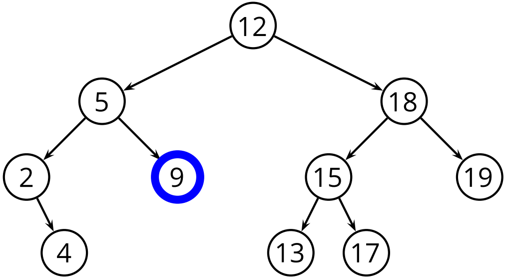
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

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



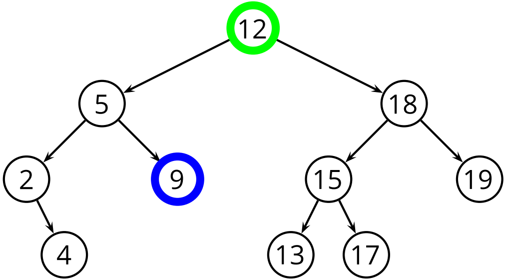
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

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



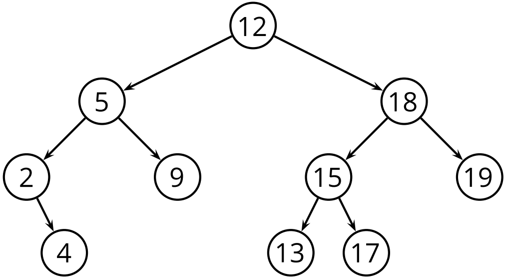
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

- 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



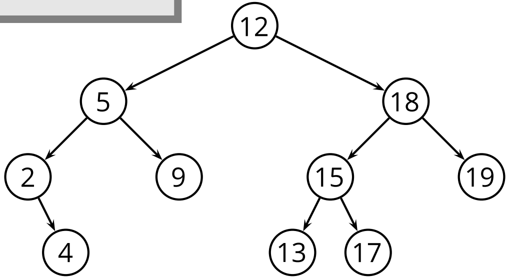
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists
- Otherwise it is the *first ancestor a* of *x* such that *x* falls in the *left* subtree of *a*

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
```

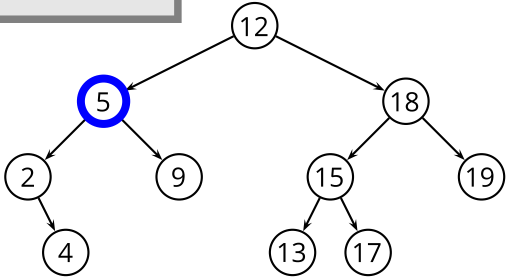
Successor and Predecessor(2)

```
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
```

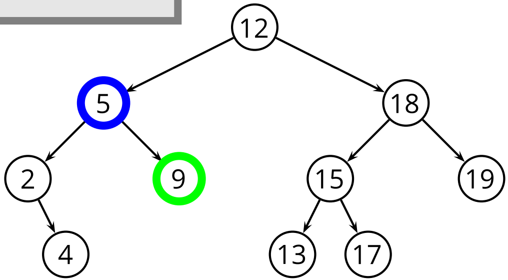


Successor and Predecessor(2)

```
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
```

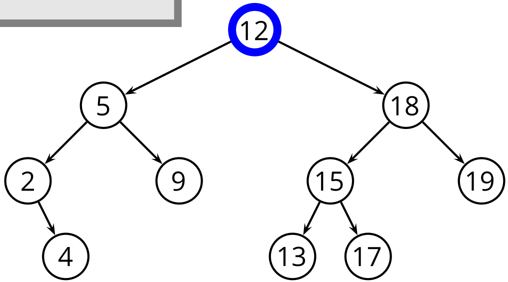


```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```



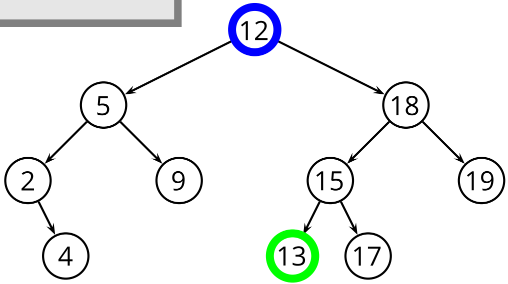
Successor and Predecessor(2)

```
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
```



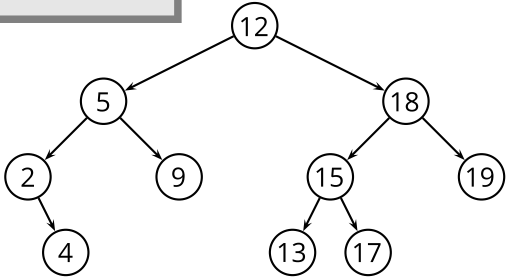
Successor and Predecessor(2)

```
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
```

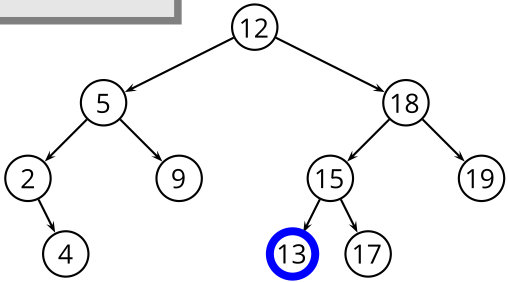


Successor and Predecessor(2)

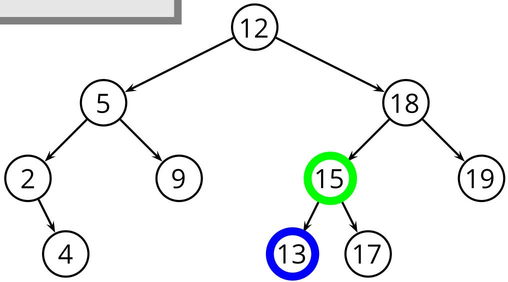
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
    x = y
    y = y.parent
return y
```



```
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
```

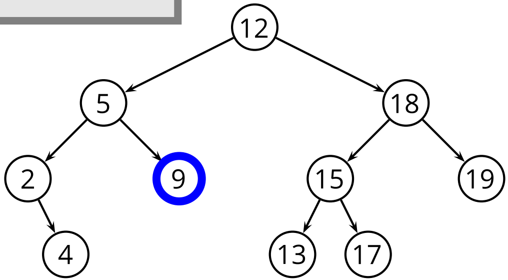


```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```



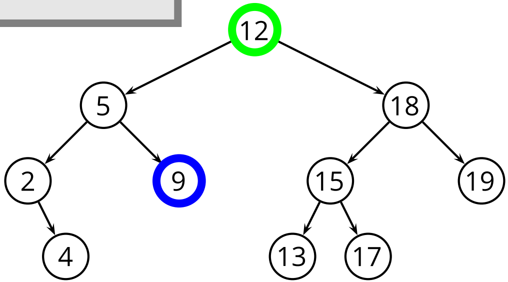
Successor and Predecessor(2)

```
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
```



- *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)
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*

- *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?

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

- *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?

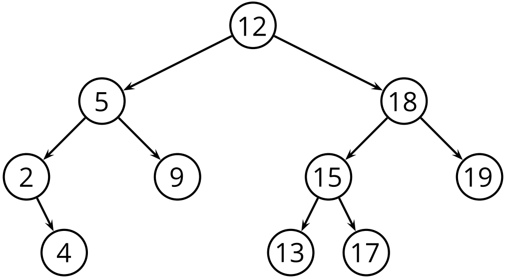
$$T(n) = \Theta(\textit{depth of the tree})$$

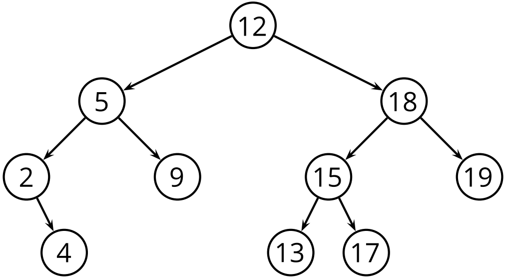
$$T(n) = O(n)$$

- Iterative *binary search*

- 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
```


12
5
18
2
9
15
19
4
13
17



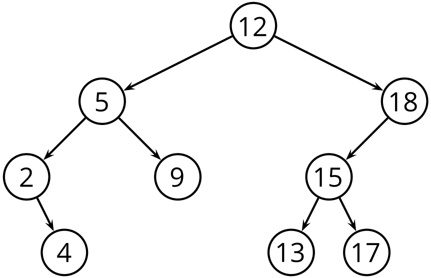
- *Idea*
  - in order to insert x, we *search* for x (more precisely $x.key$)
  - if we don't find it, we add it where the search stopped

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
```

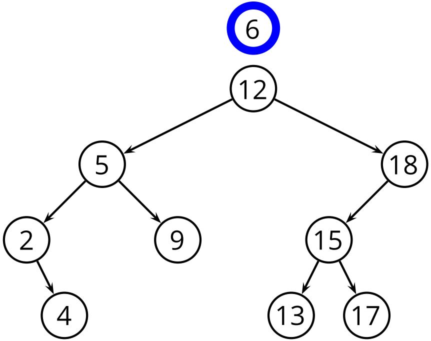
Insertion (2)

```
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
```



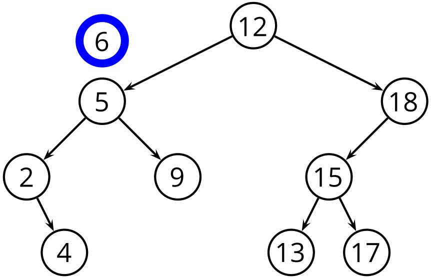
Insertion (2)

```
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
```



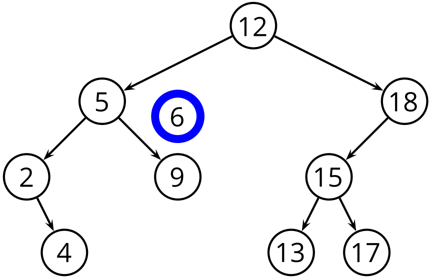
Insertion (2)

```
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
```



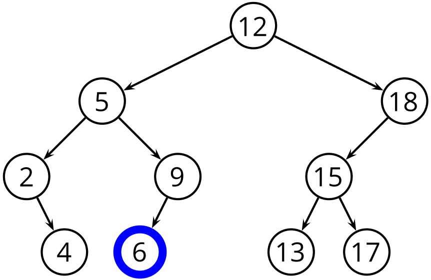
Insertion (2)

```
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
```



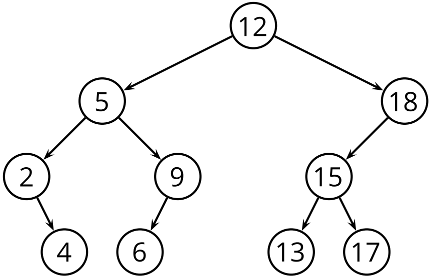
Insertion (2)

```
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
```

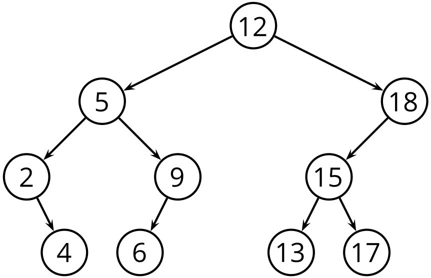


Insertion (2)

```
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
```



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

- Both insertion and search operations have complexity h, where h is the height of the tree

Observation

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

- Both insertion and search operations have complexity 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

Observation

- Both insertion and search operations have complexity 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

Observation

- Both insertion and search operations have complexity 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

- Both insertion and search operations have complexity 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 in the average case

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

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

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

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

```
TREE-RANDOMIZED-INSERT1(T, z)
  r = uniformly rand. val. 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?

```
TREE-RANDOMIZED-INSERT1(T, z)
  r = uniformly rand. val. 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

```
TREE-RANDOMIZED-INSERT1(T, z)
  r = uniformly rand. val. 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

Randomized Insertion (2)

TREE-RANDOMIZED-INSERT1(T, z)

```
r = uniformly rand. val. from {1, ..., t.size + 1}
if r = 1
      TREE-ROOT-INSERT(T, z)
else TREE-INSERT(T, z)
```

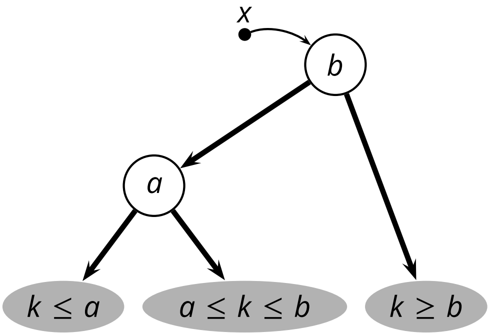
- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?
  - no, clearly the last element can only go to the top or to the bottom

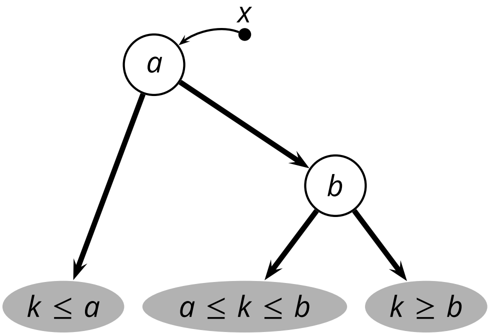
- It is true that any node has the same probability of being inserted at the top
  - this suggests a recursive application of this same procedure

```
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
```

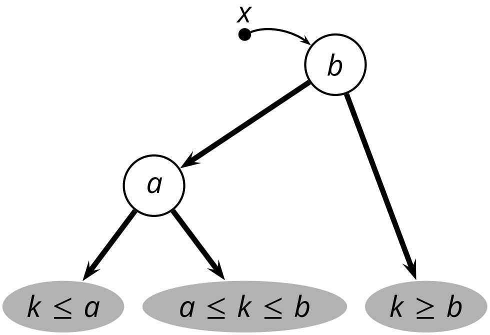
```
TREE-RANDOMIZED-INSERT(t, z)
if t = NIL
        return z
r = uniformly random value from {1, . . . , t.size + 1
if r = 1                                  // Pr[r = 1] = 1/(t.size + 1)
        z.size = t.size + 1
        return TREE-ROOT-INSERT(t, z)
if z.key < t.key
        t.left = TREE-RANDOMIZED-INSERT(t.left,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

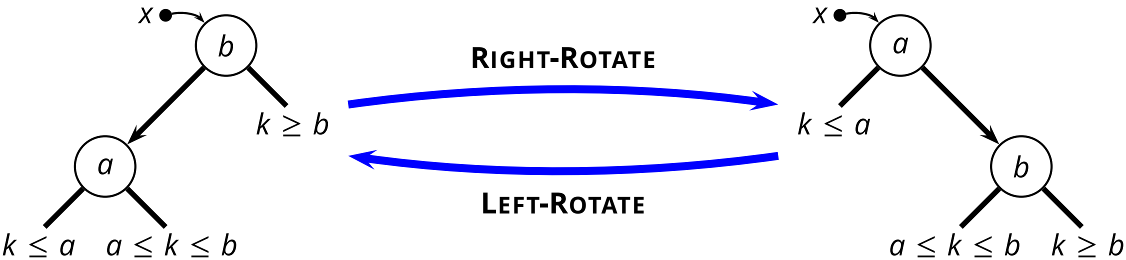

x
b
a
k ≤ a
a ≤ k ≤ b
k ≥ b



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

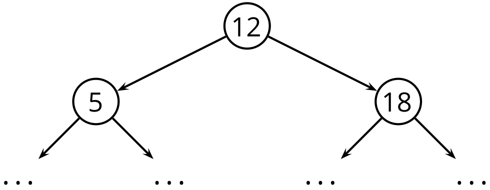


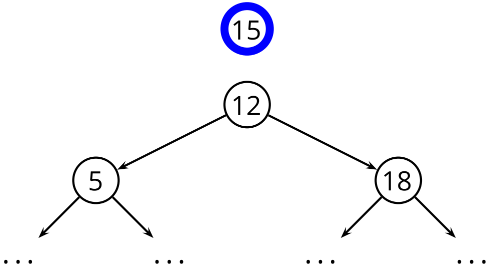
- x = **RIGHT-ROTATE**(x)
- x = **LEFT-ROTATE**(x)

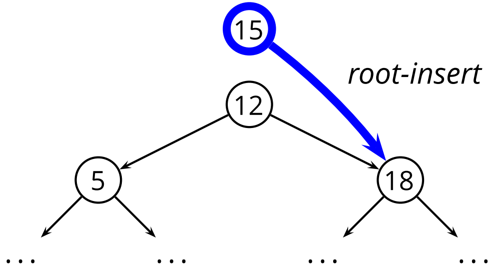



```
RIGHT-ROTATE(x)
    l = x.left
    x.left = l.right
    l.right = x
    return l
```

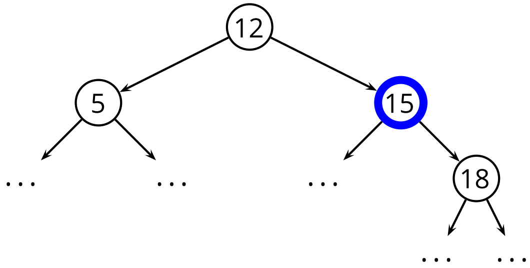
```
LEFT-ROTATE(x)
    r = x.right
    x.right = r.left
    r.left = x
    return r
```


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

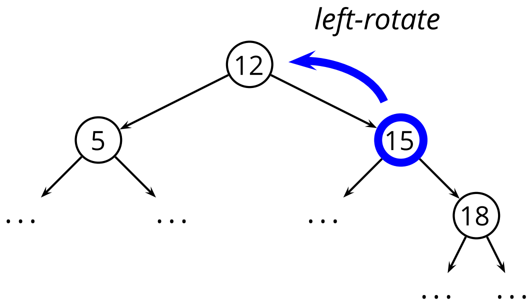

15
12
5
18
. . .
. . .
. . .
. . .



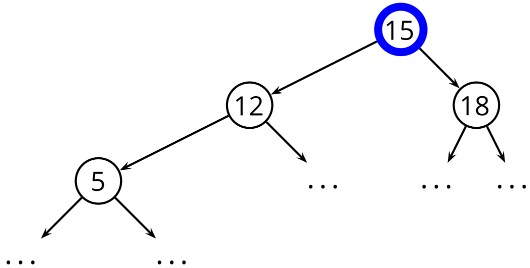
1. Recursively insert *z* at the root of the appropriate subtree (right)



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)

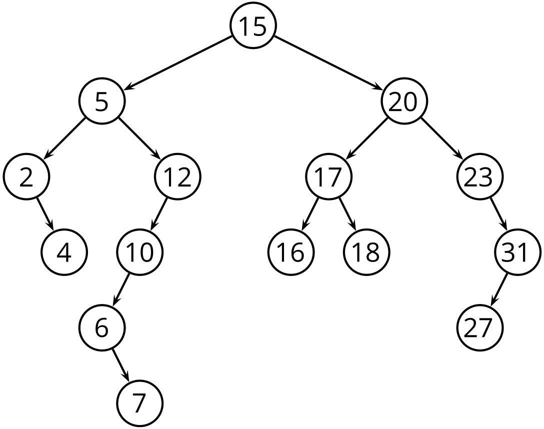
```
TREE-ROOT-INSERT(x, z)
if x = NIL
        return z
if z.key < x.key
        x.left = TREE-ROOT-INSERT(x.left, z)
        return RIGHT-ROTATE(x)
else x.right = TREE-ROOT-INSERT(x.right, z)
        return LEFT-ROTATE(x)
```

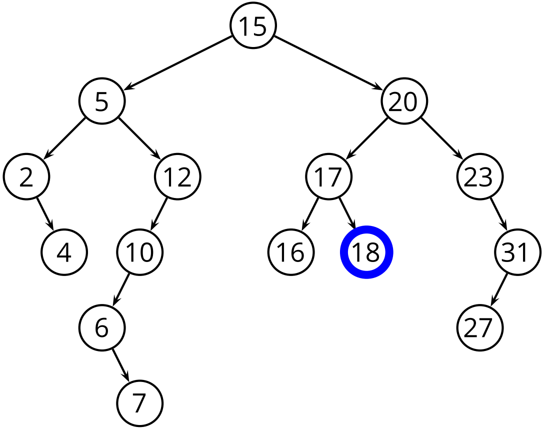
- General strategies to deal with complexity in the worst case

- 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

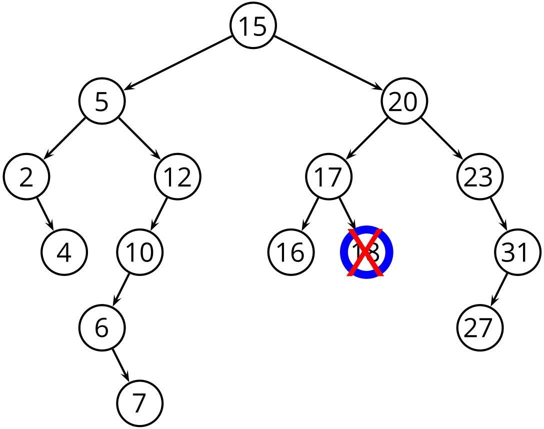
- General strategies to deal with complexity in the worst case
  - *randomization:* turns any case into the average case
    - the worst case is still possible, but it is extremely improbable
  - *amortized maintenance:* e.g., balancing a BST or resizing a hash table
    - relatively expensive but "amortized"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

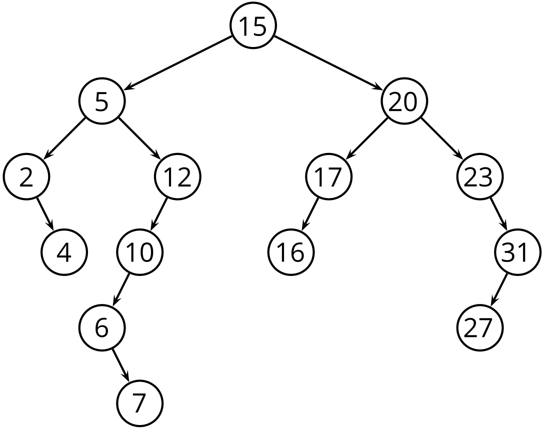

15
5
20
2
12
17
23
4
10
16
18
31
6
27
7



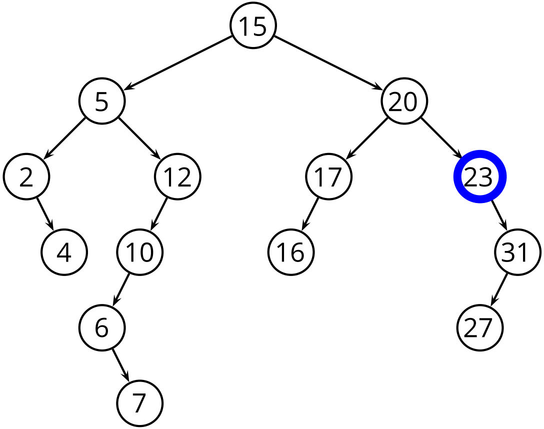
1. z has no children



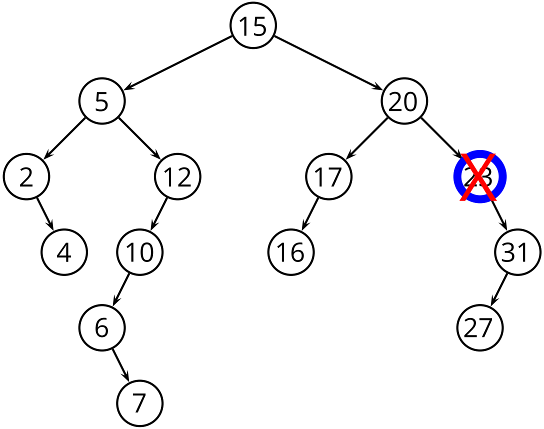
1. *z* has no children
   - simply remove *z*



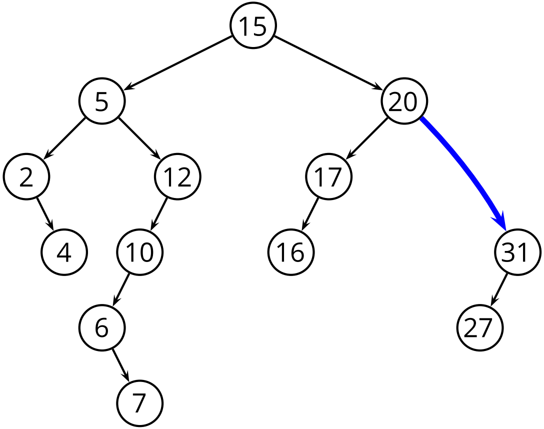
1. z has no children
   - simply remove z



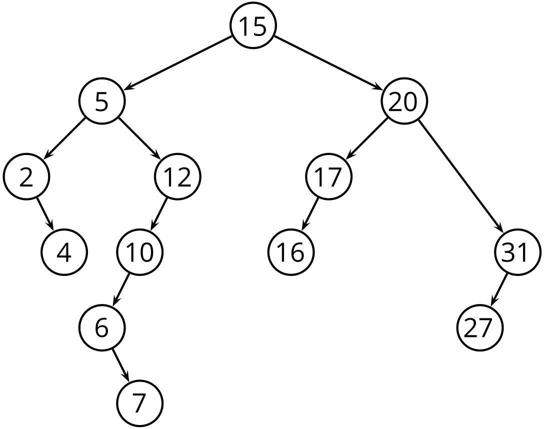
1. z has no children
   - simply remove z
2. z has one child



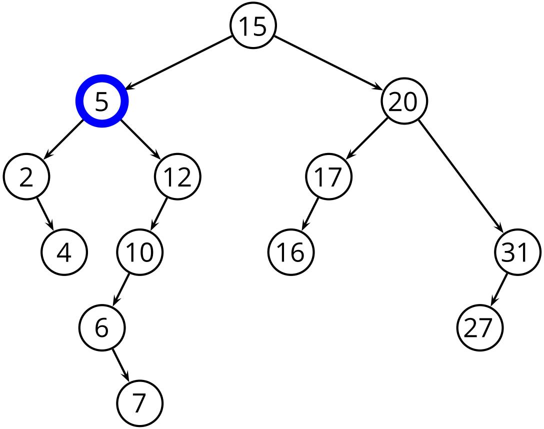
1. *z* has no children
   - simply remove *z*
2. *z* has one child
   - remove *z*



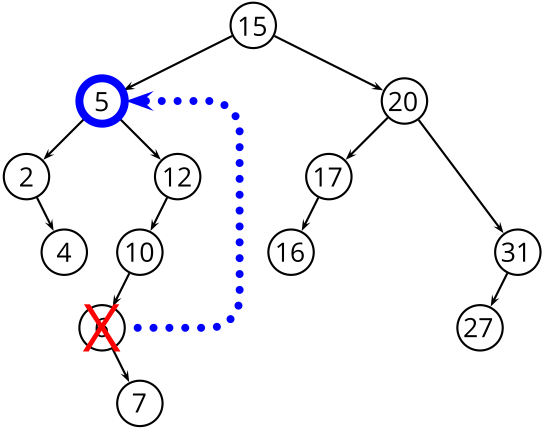
1. *z* has no children
   - simply remove *z*
2. *z* has one child
   - remove *z*
   - connect $z.parent$ to $z.right$



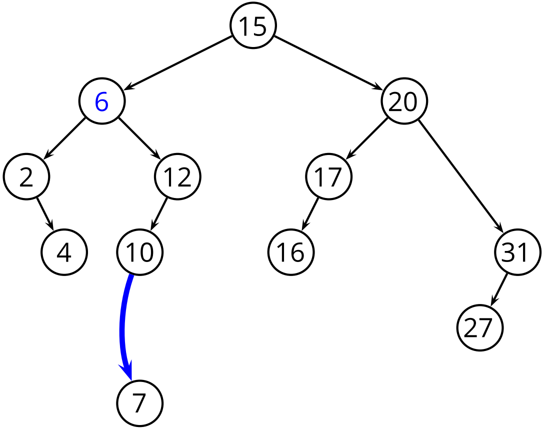
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$



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 $y = \textbf{Tree-Successor}(z)$
   - 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 $y = \textsc{Tree-Successor}(z)$
   - 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
```