Augmenting Data Structures

Augmenting Data Structures

Often useful to modify an existing data structure to perform additional operations

Augmenting Data Structures

Often useful to modify an existing data structure to perform additional operations

To do it:

1. Determine which additional info to store to do these additional ops

Augmenting Data Structures

Often useful to modify an existing data structure to perform additional operations

To do it:

1. Determine which additional info to store to do these additional ops
2. Check that this additional info can be cheaply maintained during each of the original operation

Augmenting Data Structures

Often useful to modify an existing data structure to perform additional operations

To do it:

1. Determine which additional info to store to do these additional ops
2. Check that this additional info can be cheaply maintained during each of the original operation
3. Use the additional info to efficiently implement new operation(s)

Augmenting Data Structures

Often useful to modify an existing data structure to perform additional operations

To do it:

1. Determine which additional info to store to do these additional ops
2. Check that this additional info can be cheaply maintained during each of the original operation
3. Use the additional info to efficiently implement new operation(s)

It is not an automatic process: needs creativity!

Dynamic Order Statistics

Maintain a dynamic set S of elements with distinct keys, supporting the following operations:

Dynamic Order Statistics

Maintain a dynamic set S of elements with distinct keys, supporting the following operations:

1. **Search**(x)
2. **Insert**(x)
3. **Delete**(x)

Dynamic Order Statistics

Maintain a dynamic set S of elements with distinct keys, supporting the following operations:

1. **Search**(x)
2. **Insert**(x)
3. **Delete**(x)

Dictionary !

Dynamic Order Statistics

Maintain a dynamic set S of elements with distinct keys, supporting the following operations:

1. **Search**(x)
2. **Insert**(x)
3. **Delete**(x)

Dictionary !

4. **Select**(k) : Find the k^{th} smallest element in S

Dynamic Order Statistics

Maintain a dynamic set S of elements with distinct keys, supporting the following operations:

1. **Search**(x)
2. **Insert**(x)
3. **Delete**(x)

Dictionary !

4. **Select**(k) : Find the k^{th} smallest element in S

   i.e. find the element with rank k

Dynamic Order Statistics

Maintain a dynamic set S of elements with distinct keys, supporting the following operations:

1. **Search**(x)
2. **Insert**(x)
3. **Delete**(x)

Dictionary !

4. **Select**(k) : Find the k^{th} smallest element in S

   i.e. find the element with <u>rank</u> k

Rank of x : Position of x in the sorted order of S

Dynamic Order Statistics

Maintain a dynamic set S of elements with distinct keys, supporting the following operations:

1. **Search**(x)
2. **Insert**(x)
3. **Delete**(x)

Dictionary !

4. **Select**(k) : Find the k^{th} smallest element in S

   i.e. find the element with rank k

5. **Rank**(x) : Given a pointer to x, find the rank of x

Rank of x : Position of x in the sorted order of S

Example

$S = \{5, 15, 27, 30, 56\}$

Example

S = {5, 15, 27, 30, 56}

Select(4) =

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Select(1) =

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Select(1) = 5

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Select(1) = 5

Rank(15) =

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Select(1) = 5

Rank(15) = 2

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Select(1) = 5

Rank(15) = 2

Rank(30) =

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Select(1) = 5

Rank(15) = 2

Rank(30) = 4

Example

S = {5, 15, 27, 30, 56}

Select(4) = 30

Select(1) = 5

Rank(15) = 2

Rank(30) = 4 (Note: Exactly 3 elements < 30)

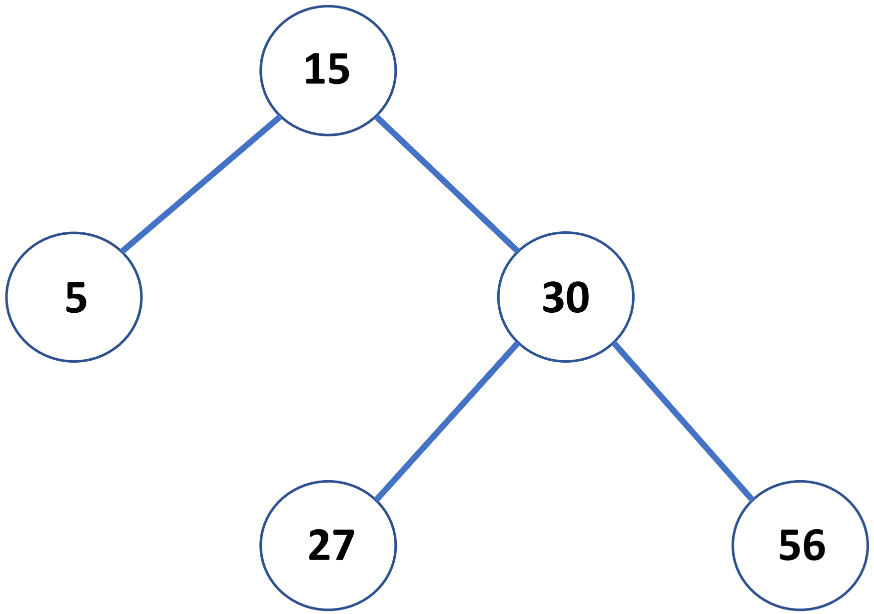
What data structure do we use?

What data structure do we use?

- For efficient **Search, Insert, Delete:**

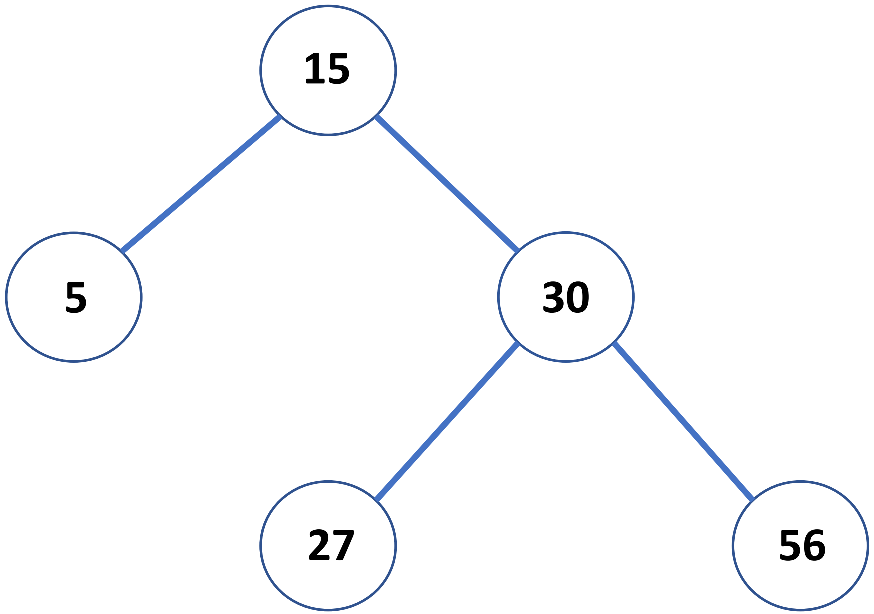
What data structure do we use?

- For efficient **Search, Insert, Delete:**
  keep S in an AVL tree



What data structure do we use?

- For efficient **Search, Insert, Delete:**
  keep S in an AVL tree (or any balanced BSTs)

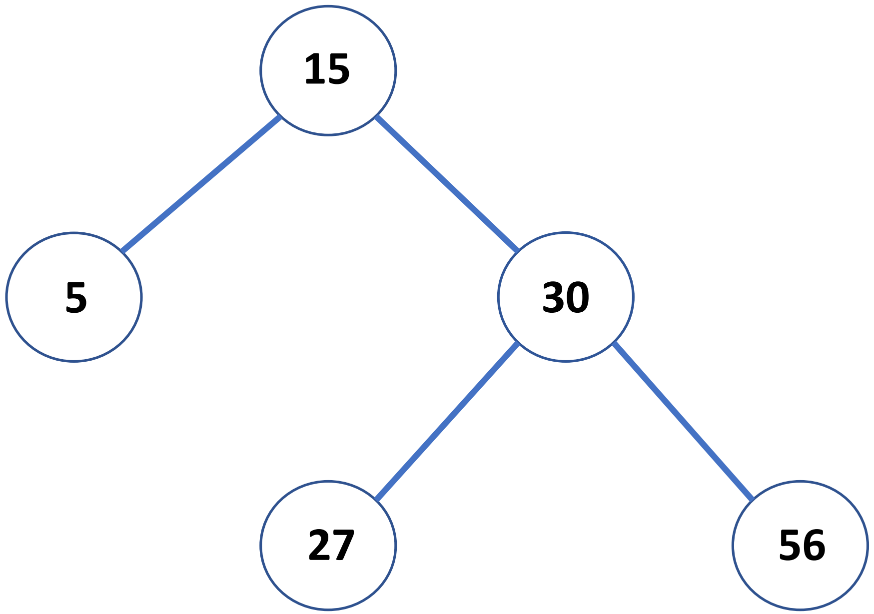


What data structure do we use?

- For efficient **Search, Insert, Delete:**

  keep S in an AVL tree (or any balanced BSTs)

- How to efficiently implement **Select, Rank** ?



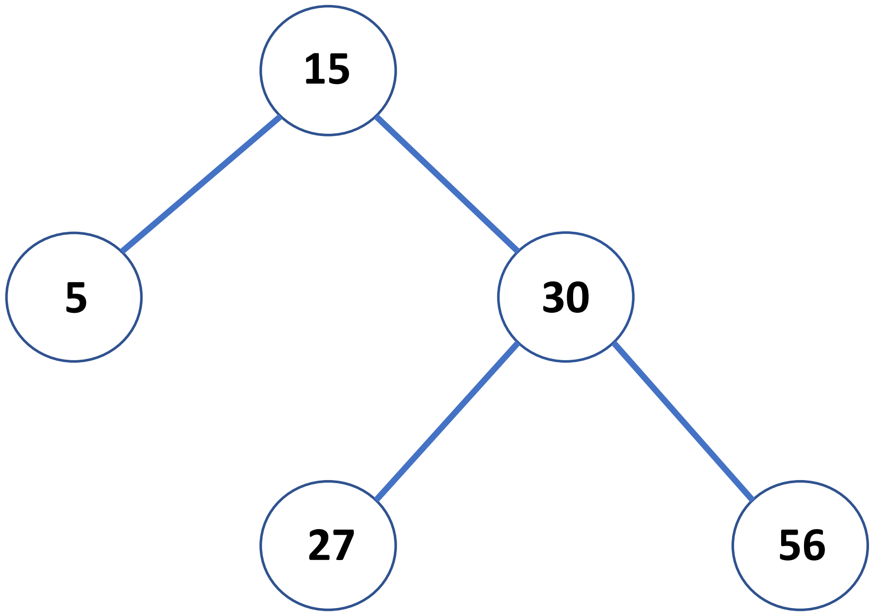
What data structure do we use?

- For efficient **Search, Insert, Delete:**

  keep S in an AVL tree (or any balanced BSTs)

- How to efficiently implement **Select, Rank** ?

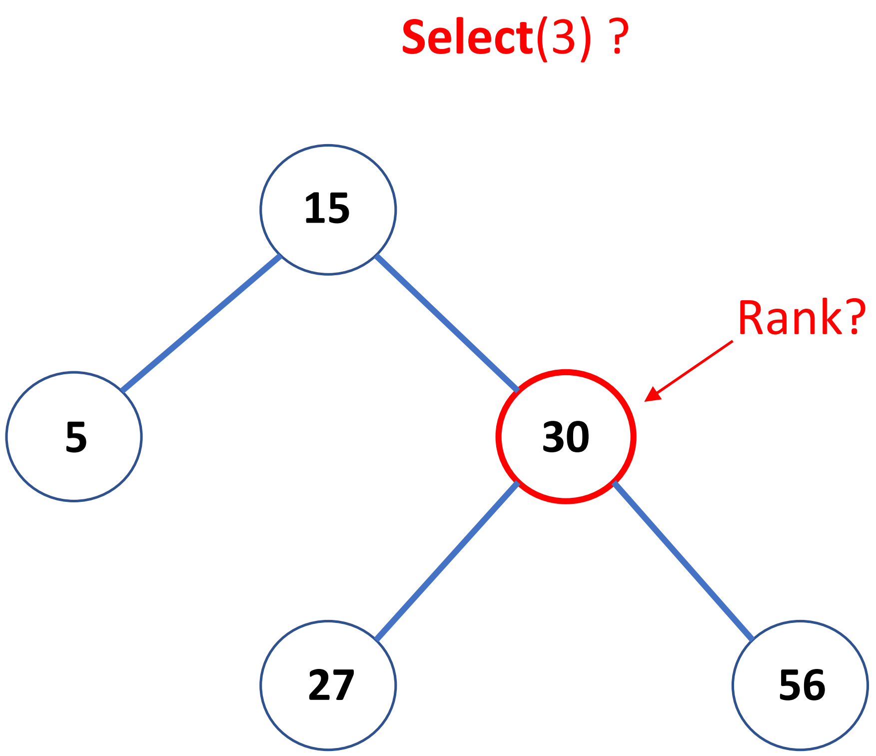
Select(3) ?



What data structure do we use?

- For efficient **Search, Insert, Delete:**
  keep S in an AVL tree (or any balanced BSTs)

- How to efficiently implement **Select, Rank** ?

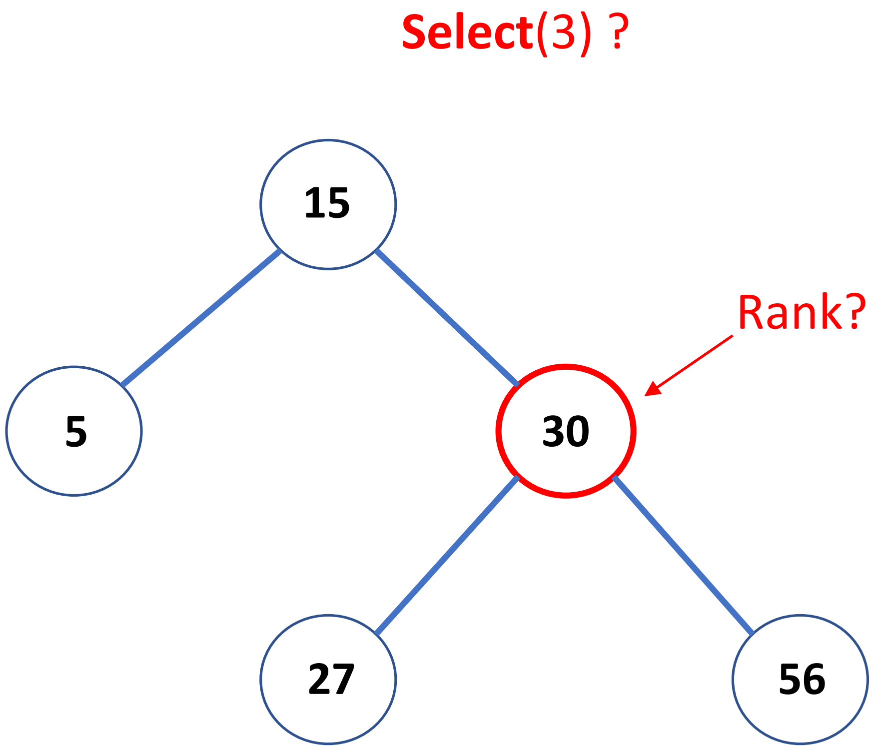


What data structure do we use?

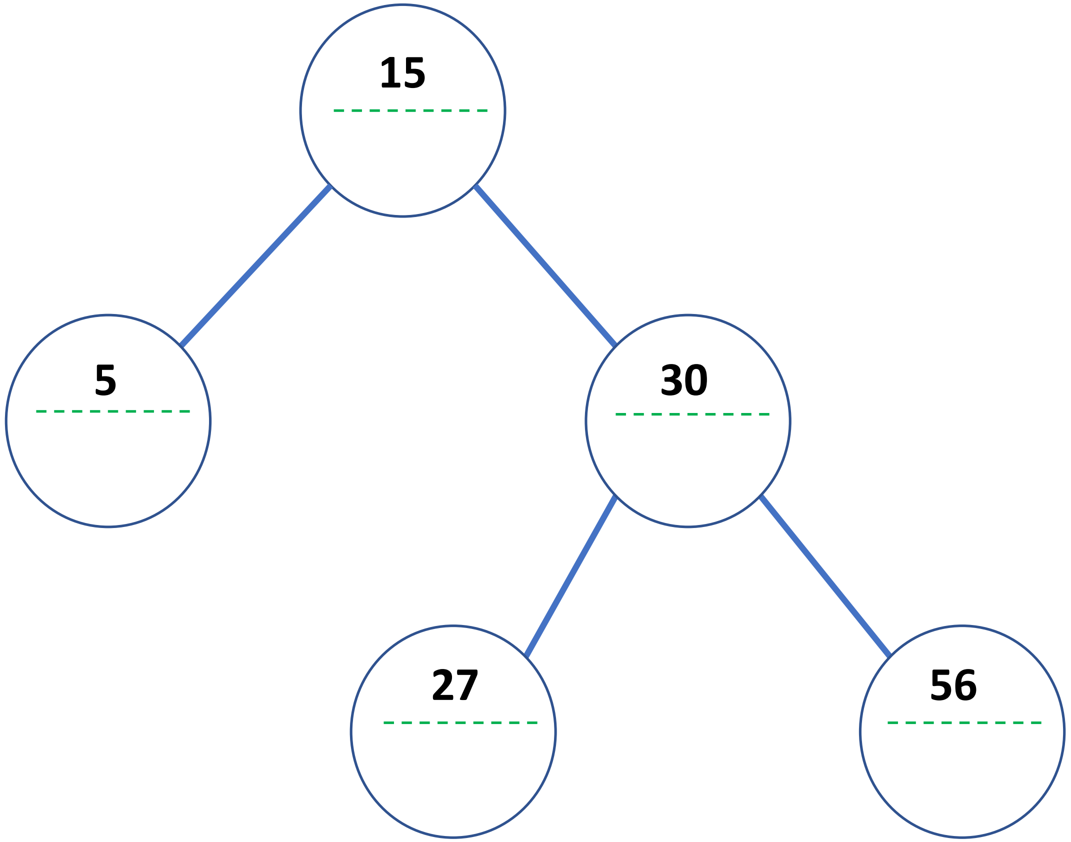
- For efficient **Search, Insert, Delete:**

  keep S in an AVL tree (or any balanced BSTs)

- How to efficiently implement **Select, Rank** ?
  - Augment the AVL tree!

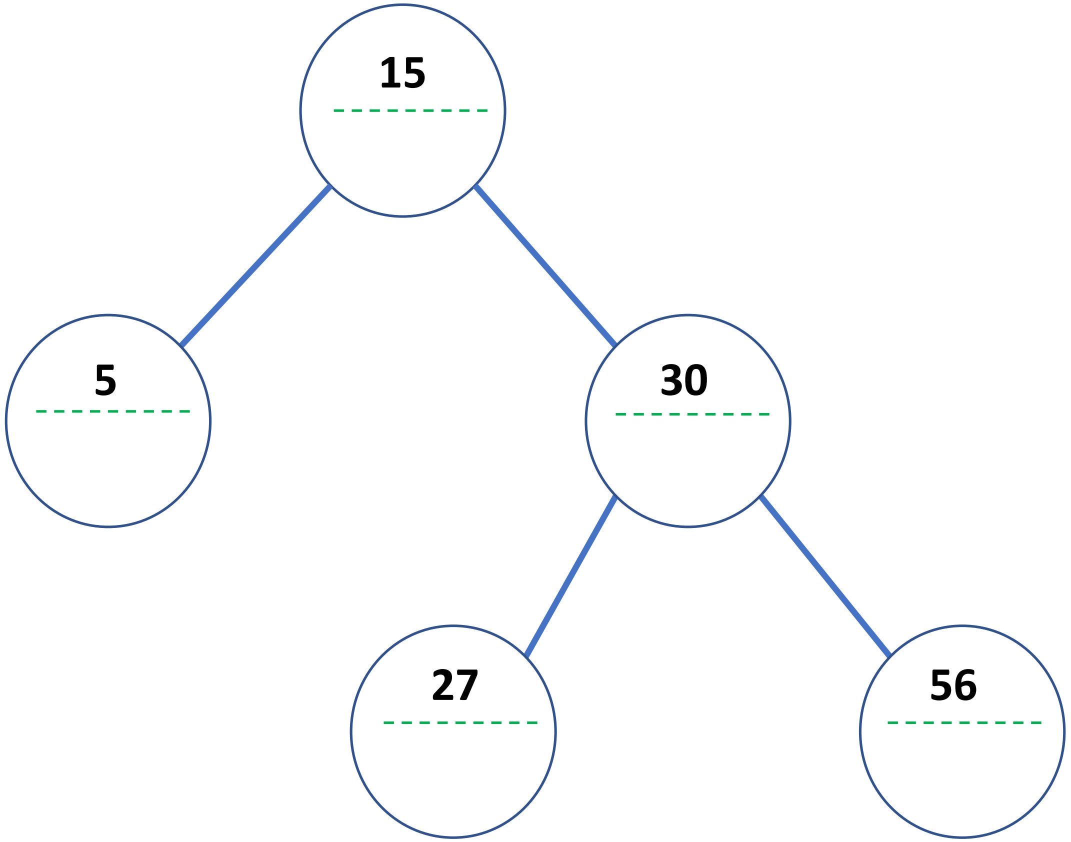


Naïve Augmentation



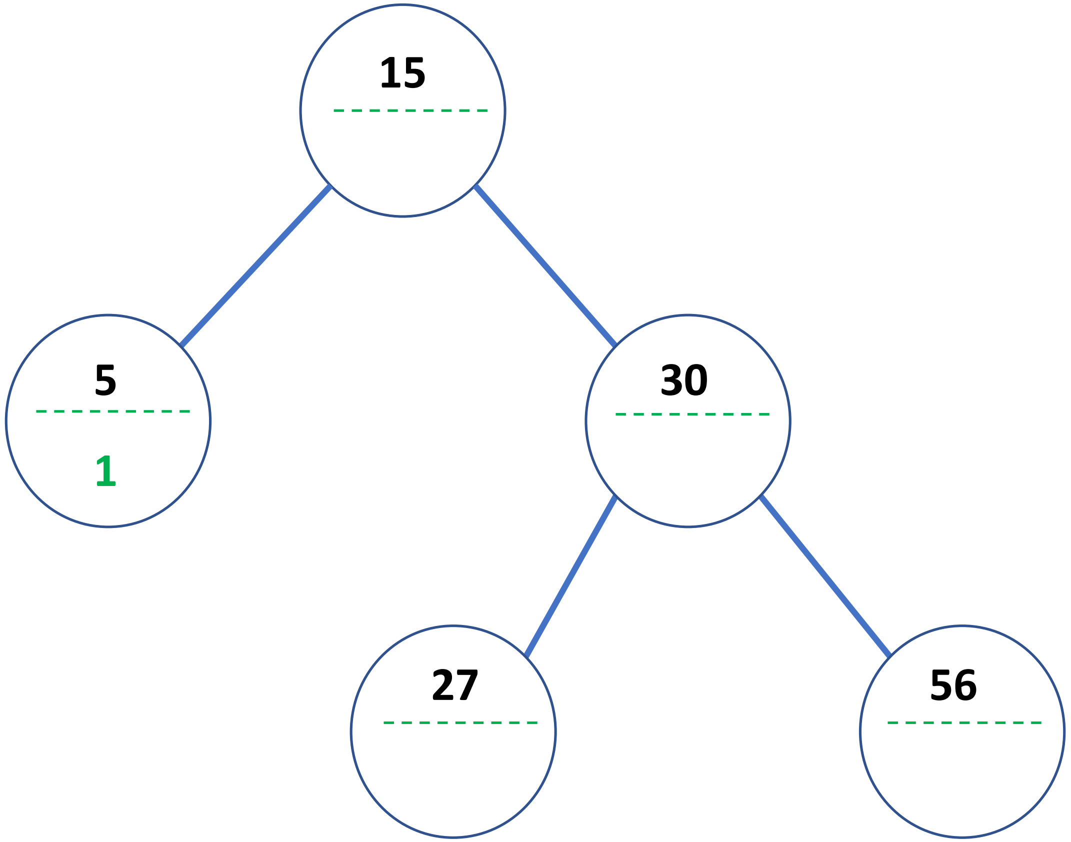
Naïve Augmentation

- At each node x, also store the rank of x



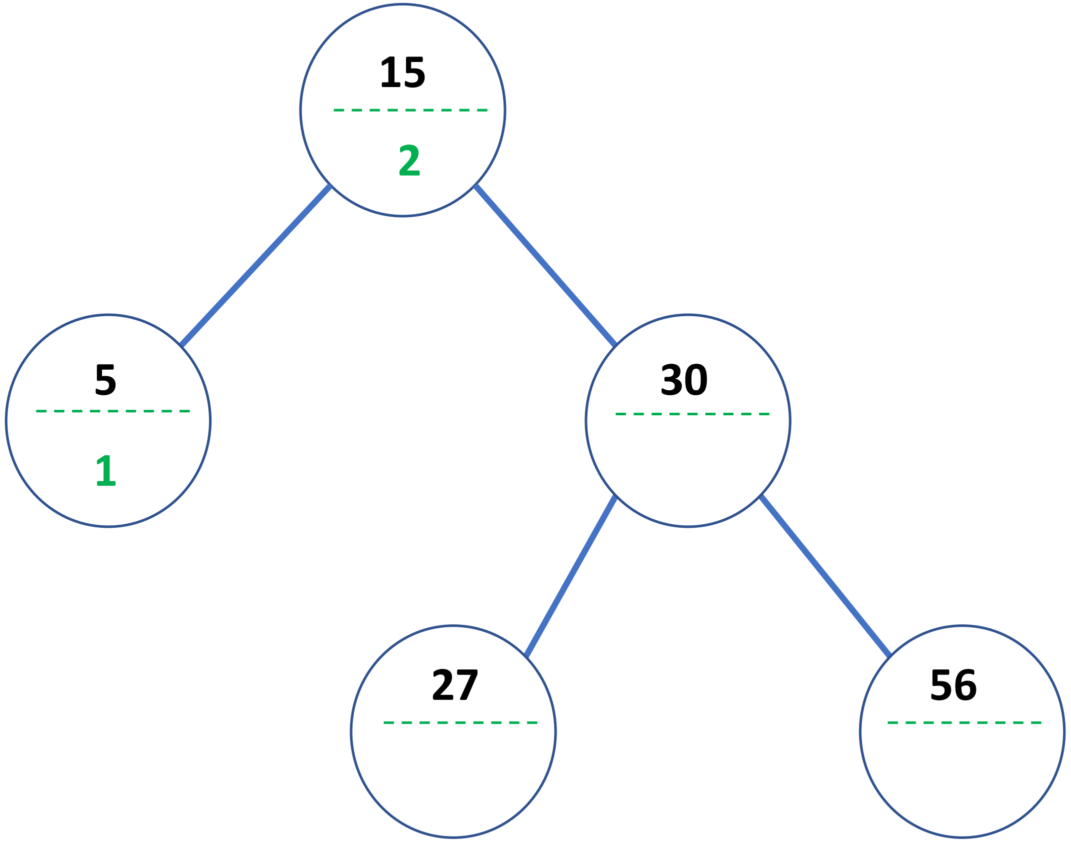
Naïve Augmentation

- At each node x, also store the rank of x



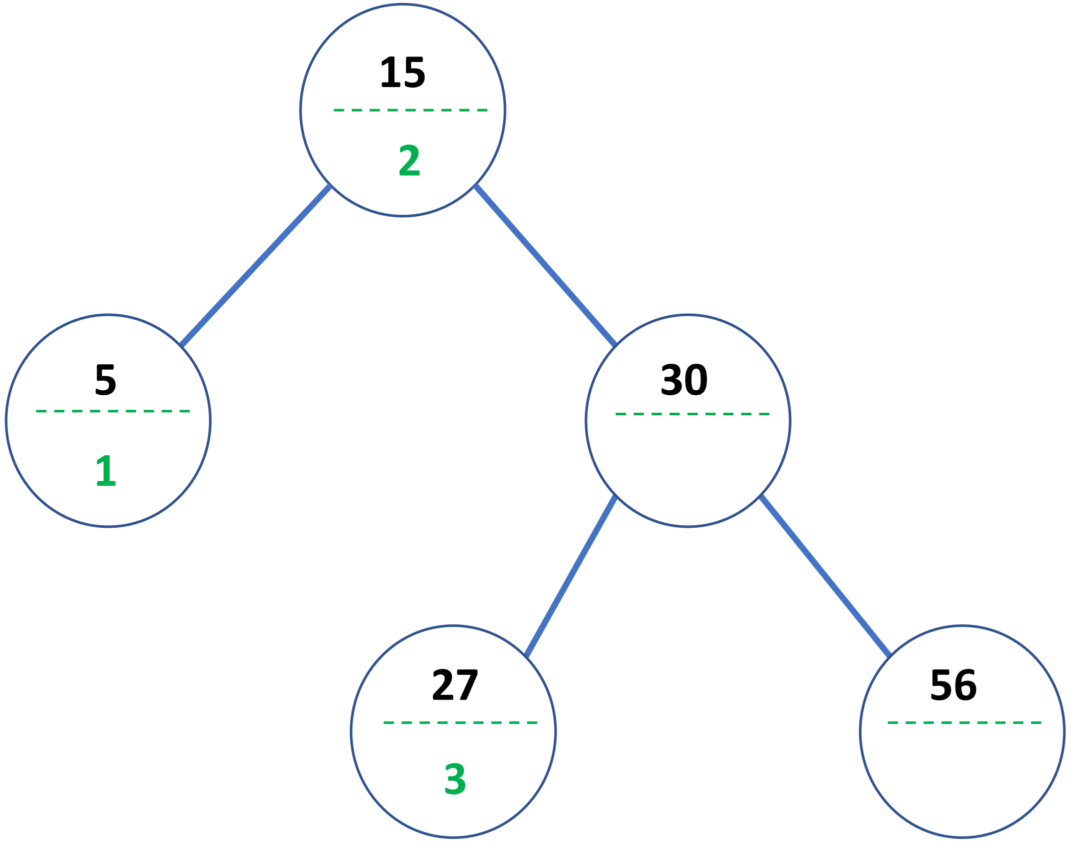
Naïve Augmentation

- At each node x, also store the rank of x



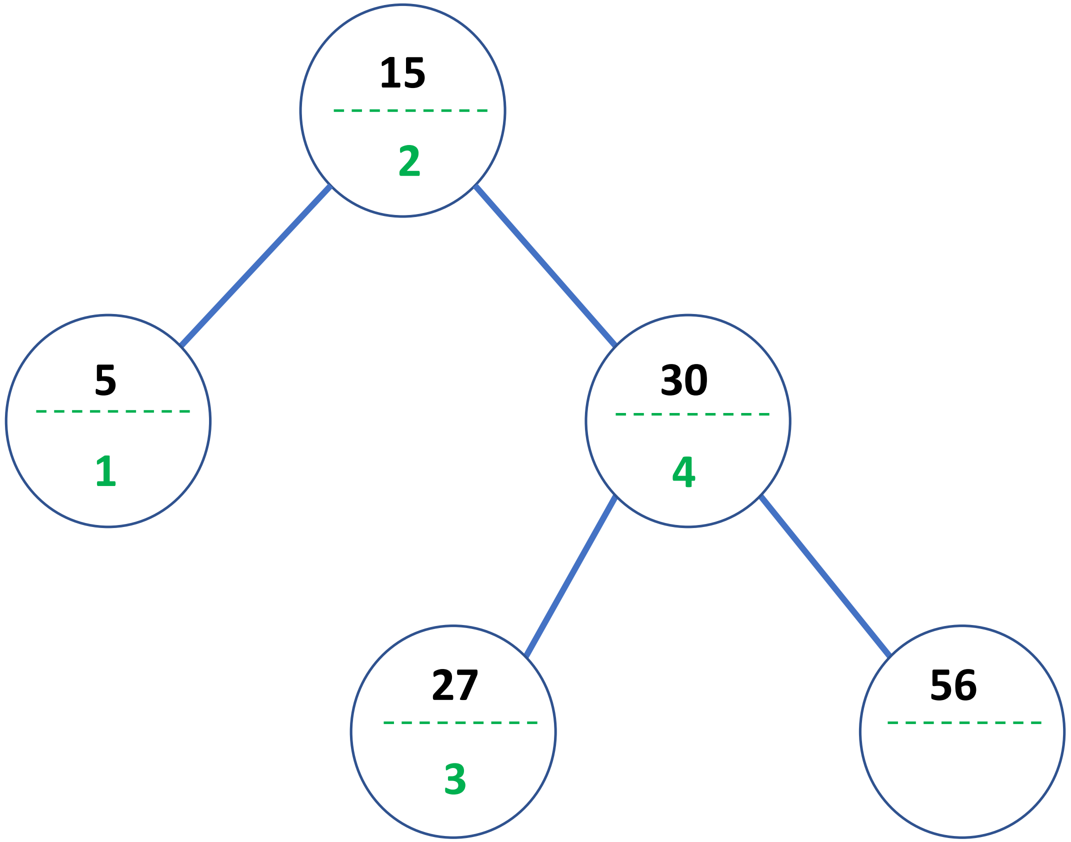
Naïve Augmentation

- At each node x, also store the rank of x



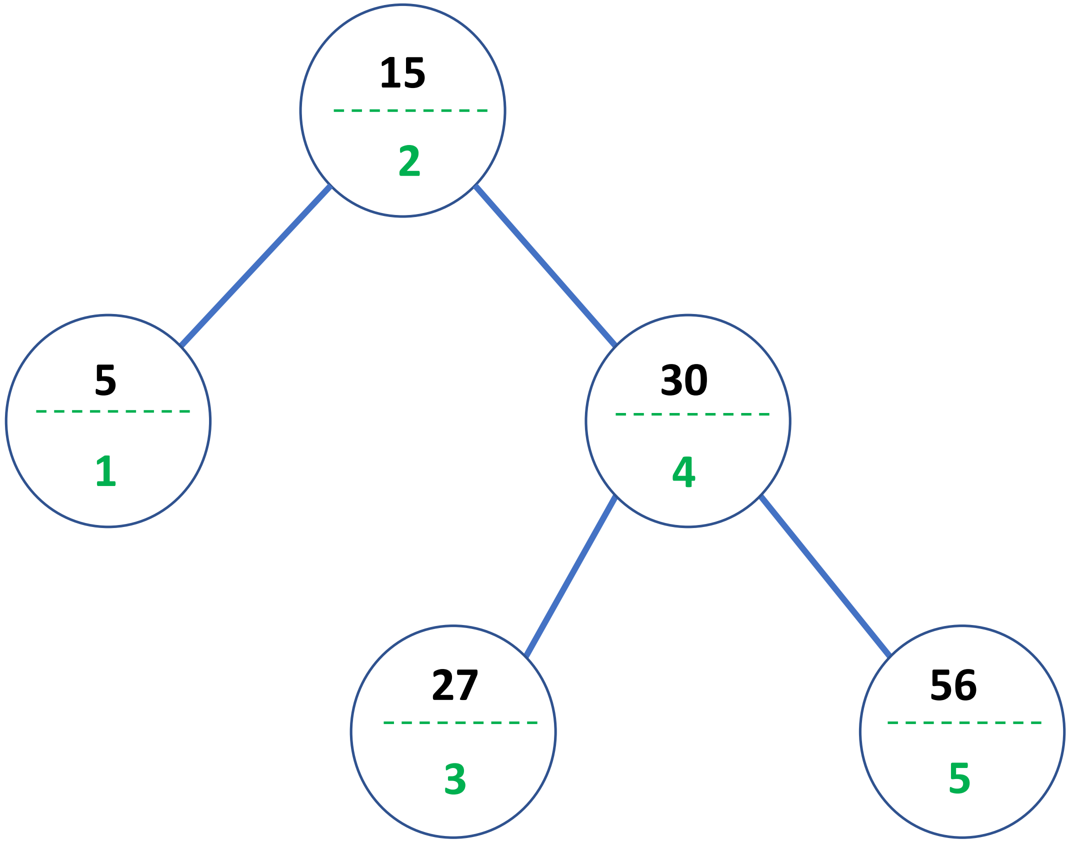
Naïve Augmentation

- At each node x, also store the rank of x



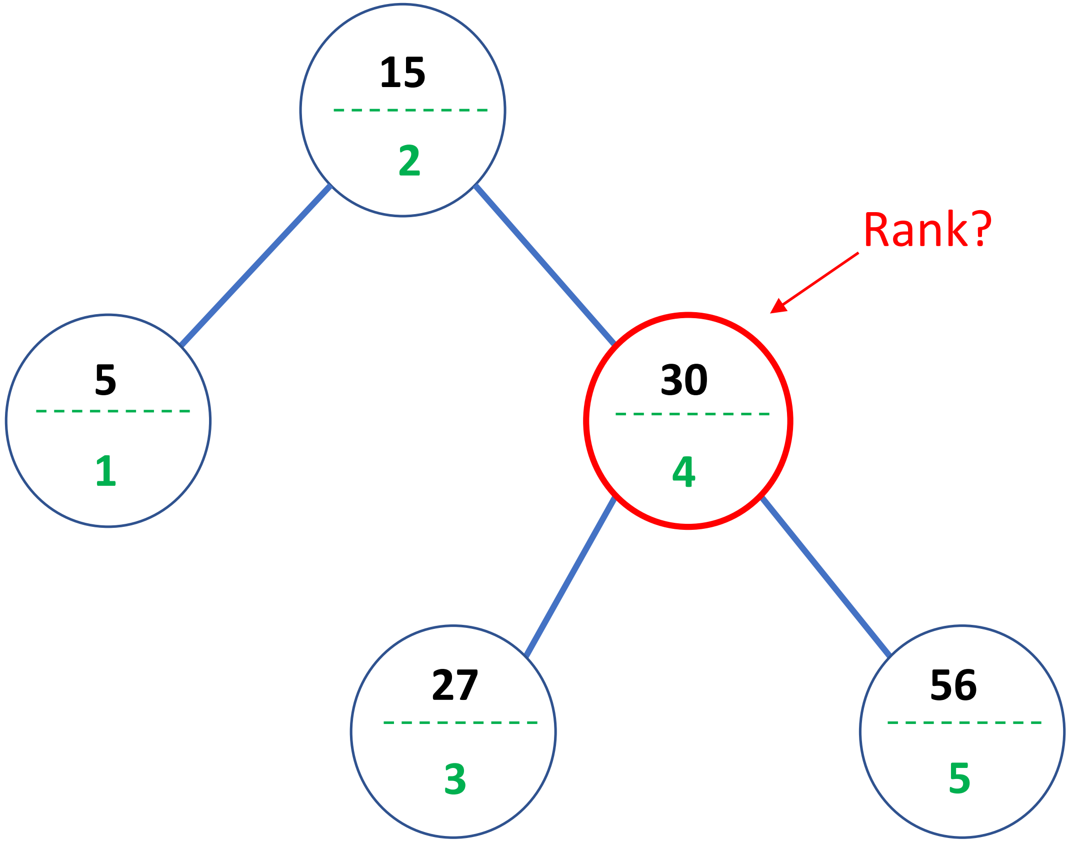
Naïve Augmentation

- At each node x, also store the rank of x



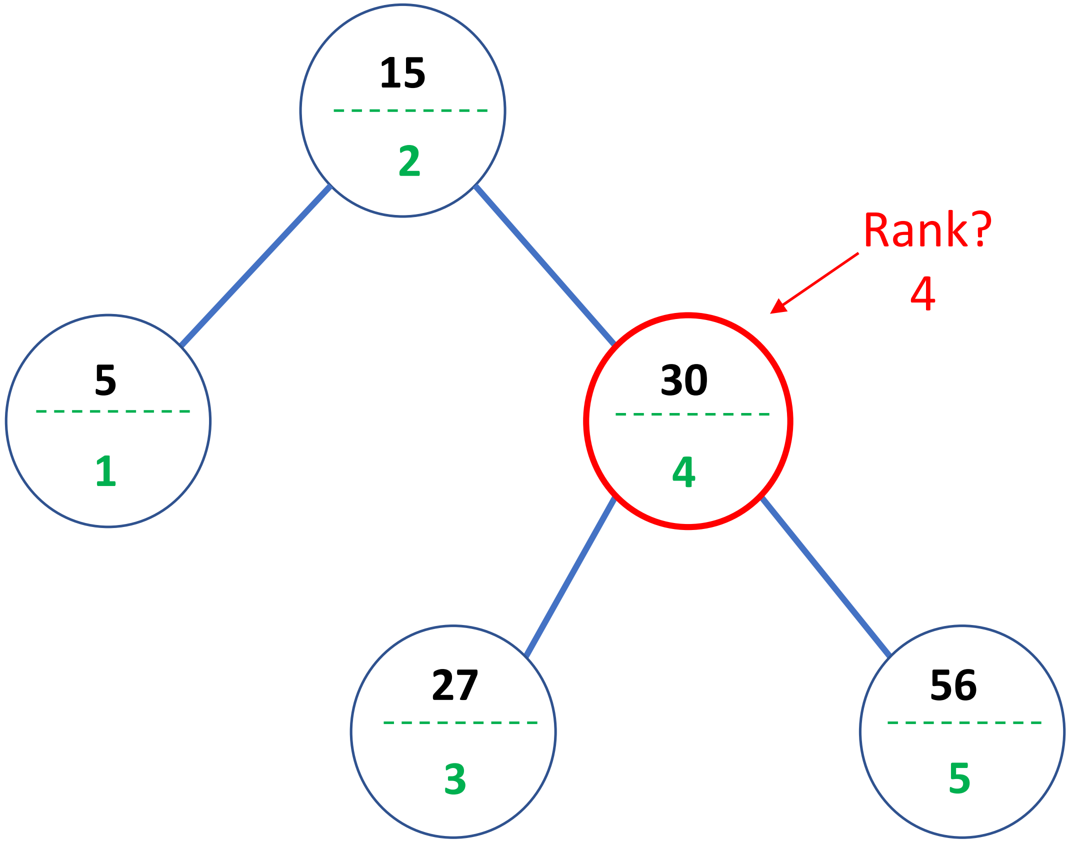
Naïve Augmentation

- At each node x, also store the rank of x



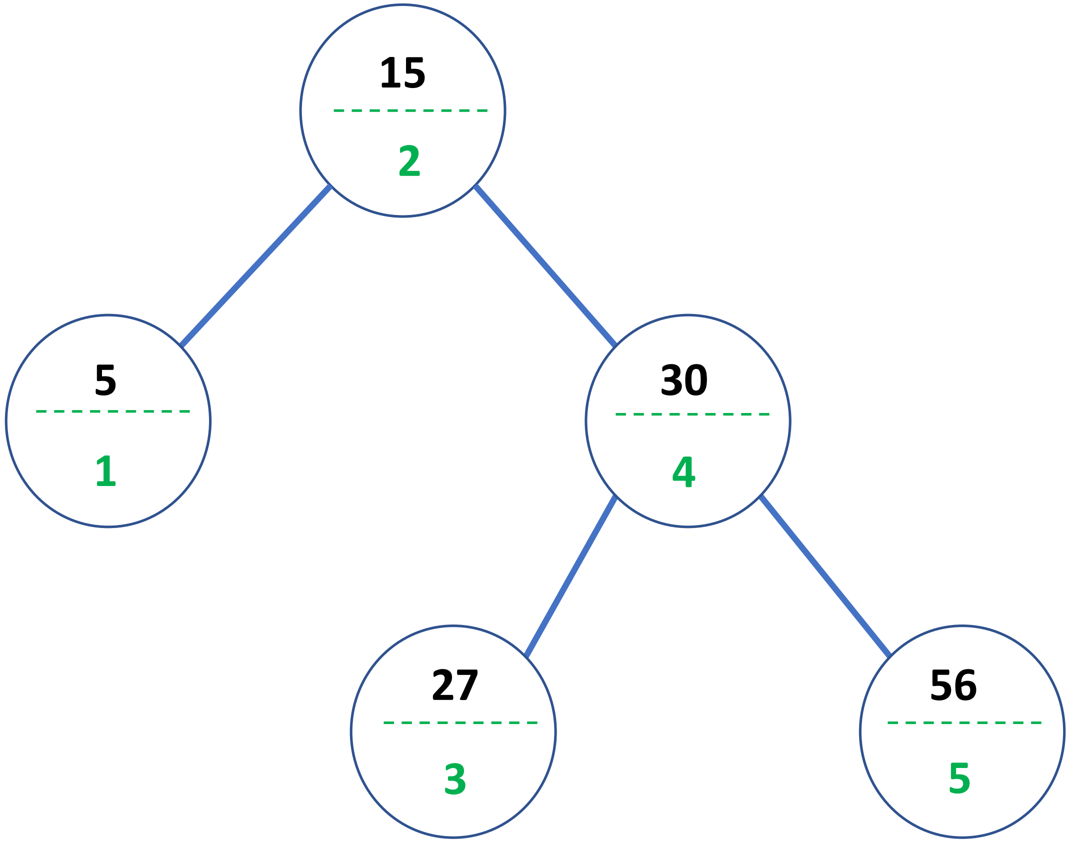
Naïve Augmentation

- At each node x, also store the rank of x



Naïve Augmentation

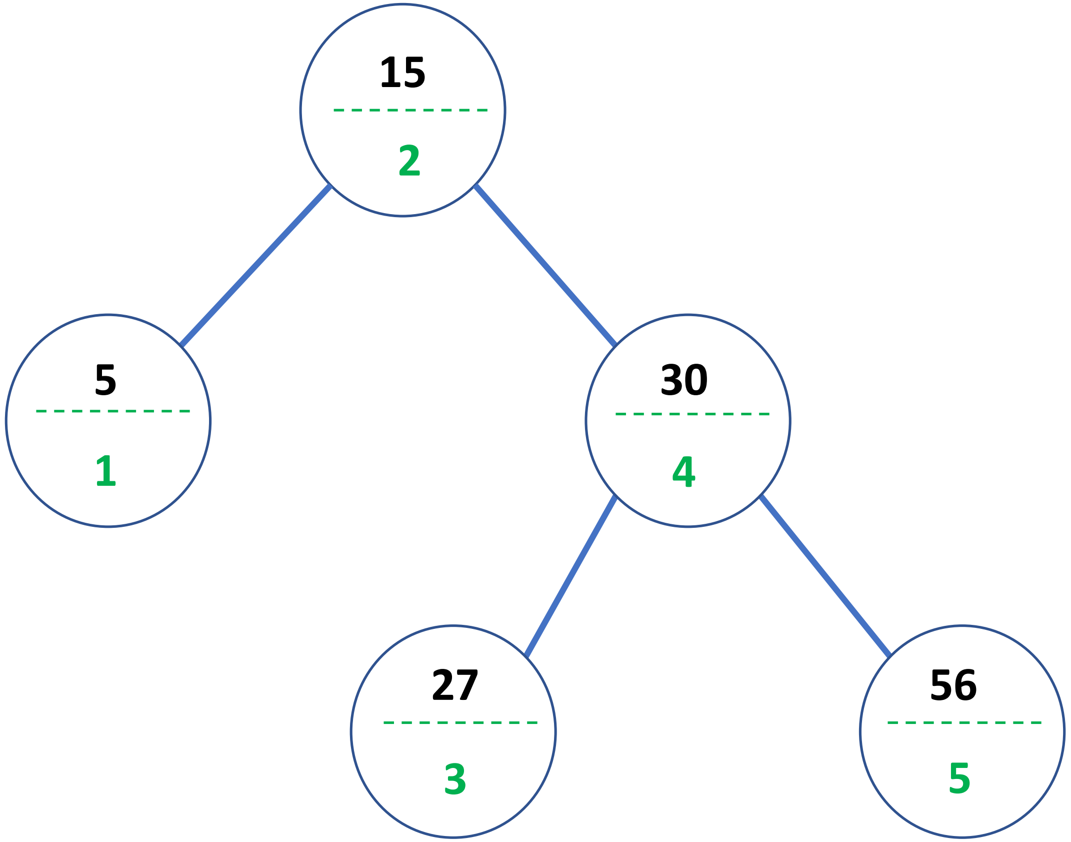
- At each node x, also store the rank of x
- Good: Efficient **Rank**(x)



Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x)

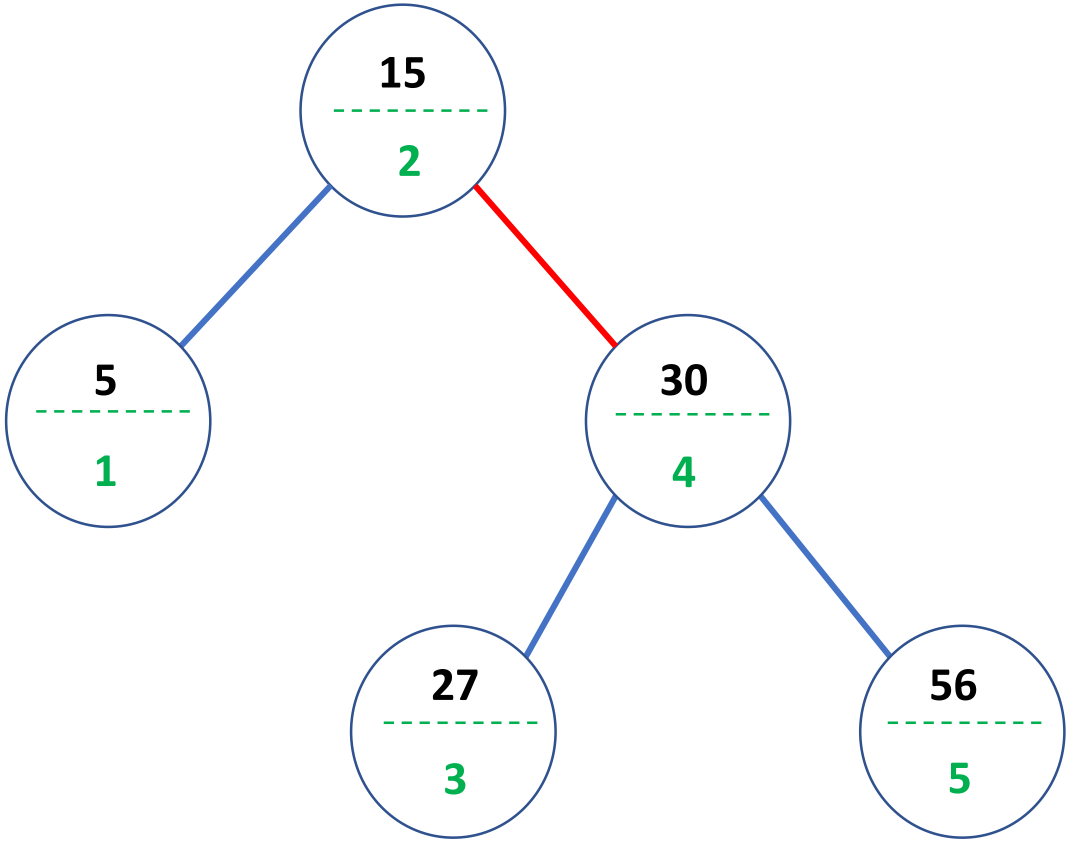
Select(3)?



Naïve Augmentation

- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x)

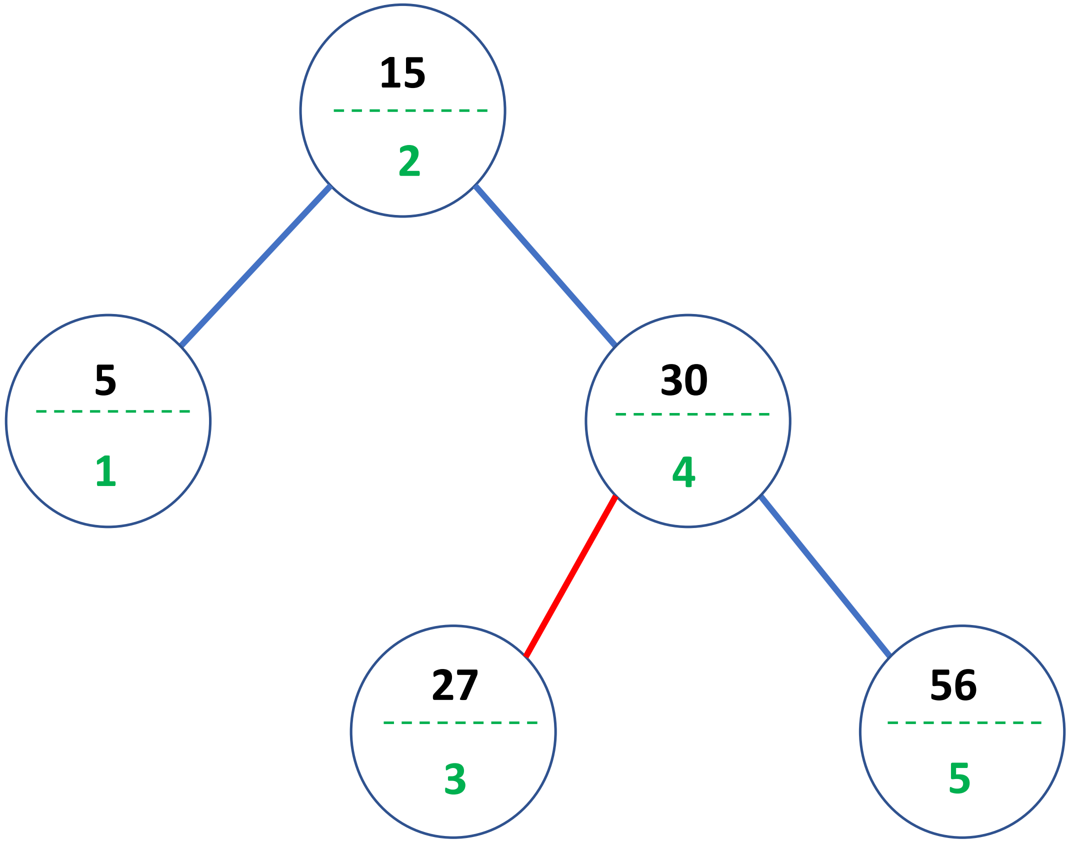
Select(3)?



Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x)

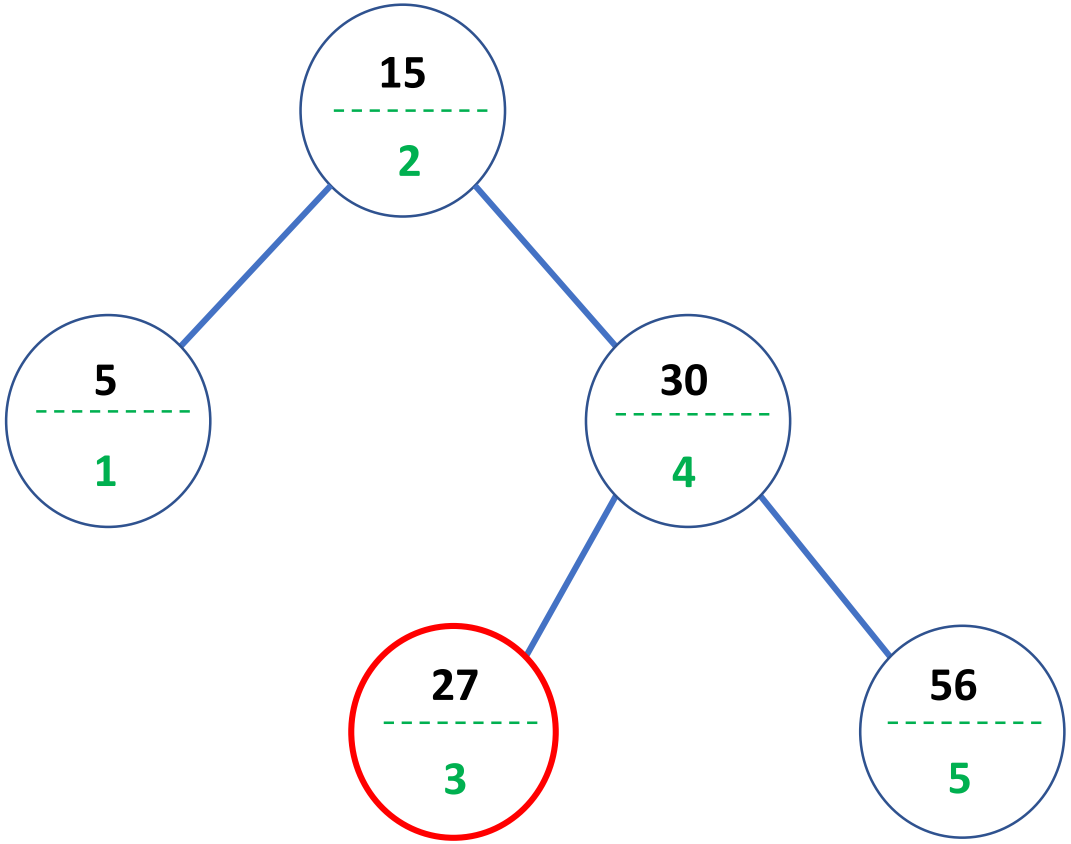
Select(3)?



Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x)

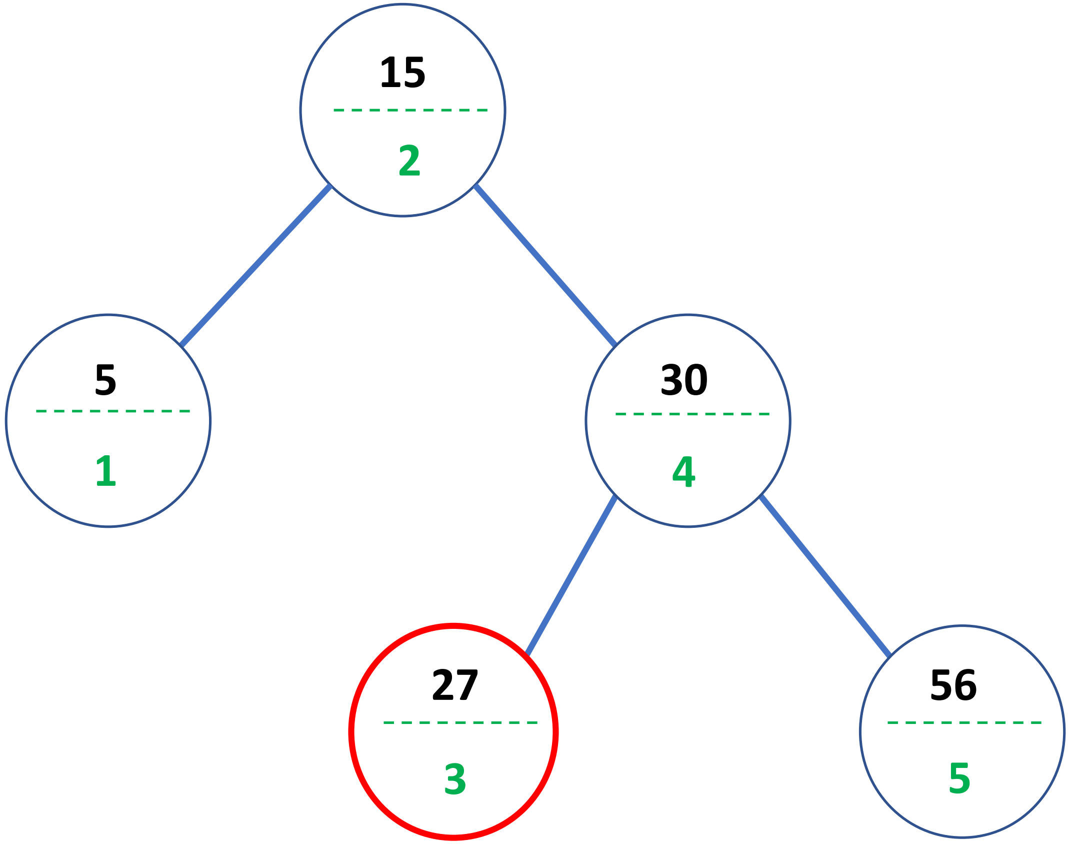
Select(3)?



Naïve Augmentation

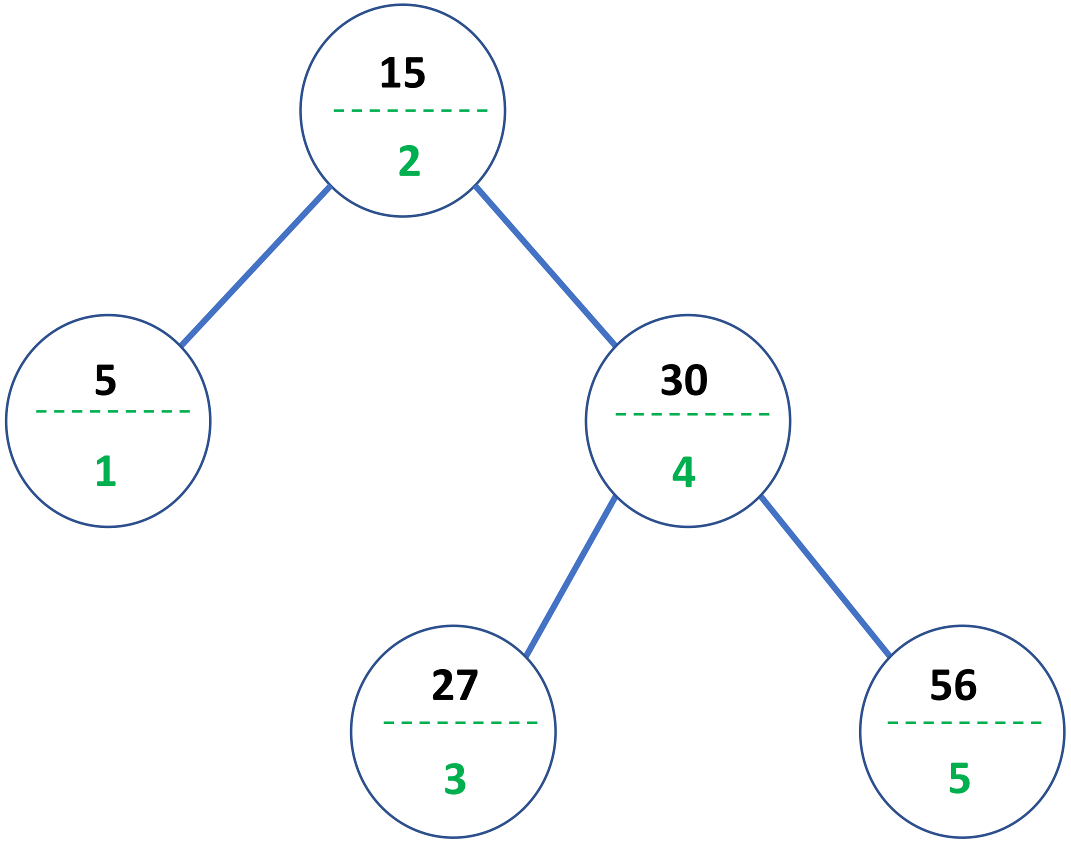
- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x)

Select(3)?
27



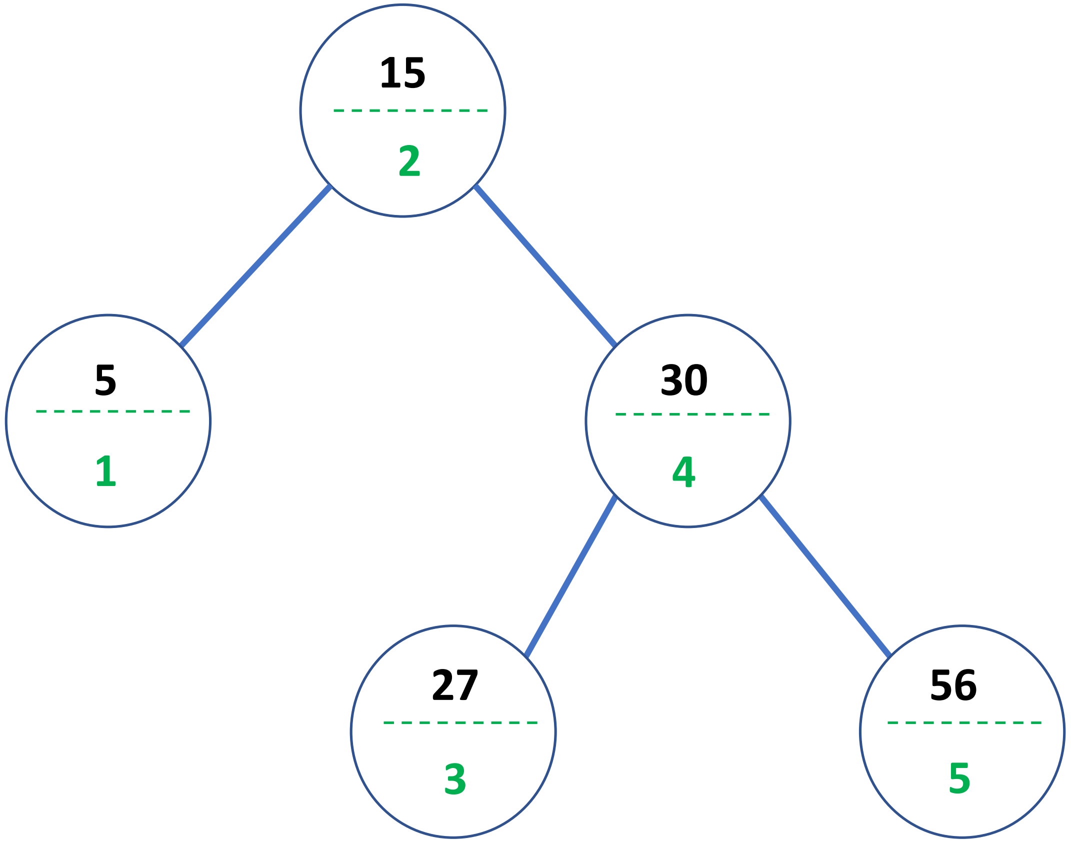
Naïve Augmentation

- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x) and **Select**(k)



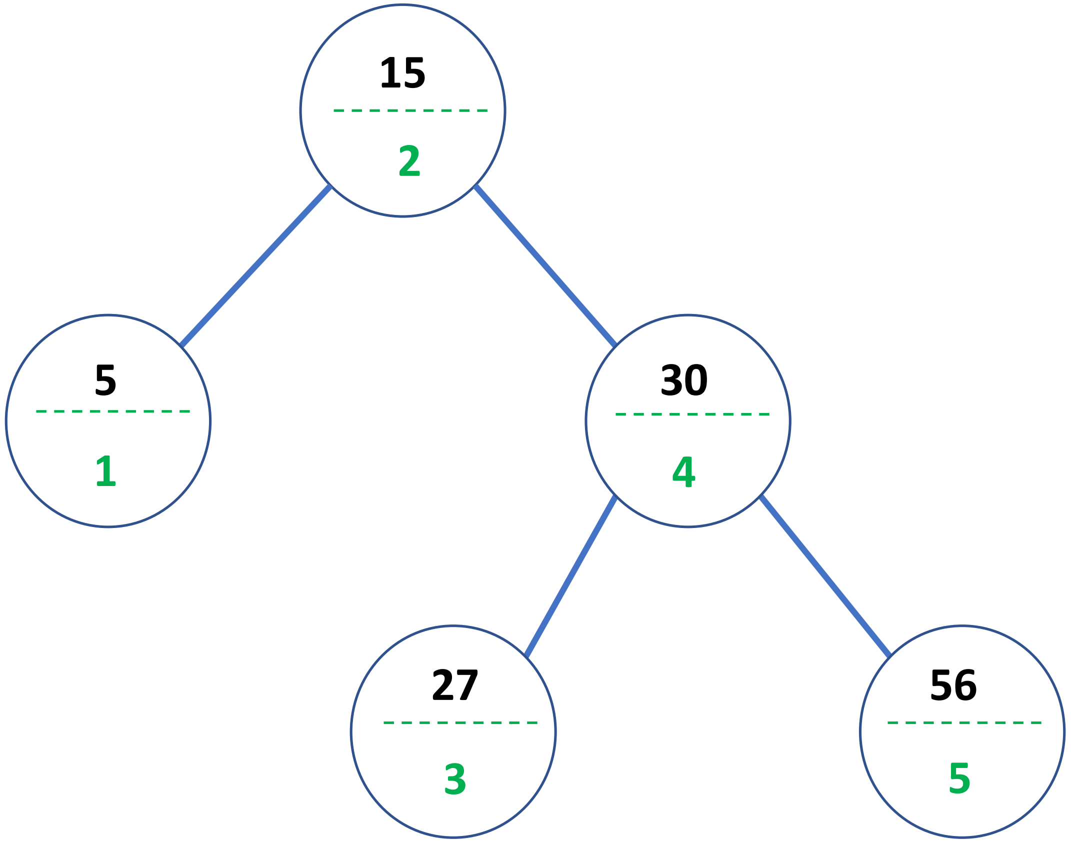
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)
- Are we done?



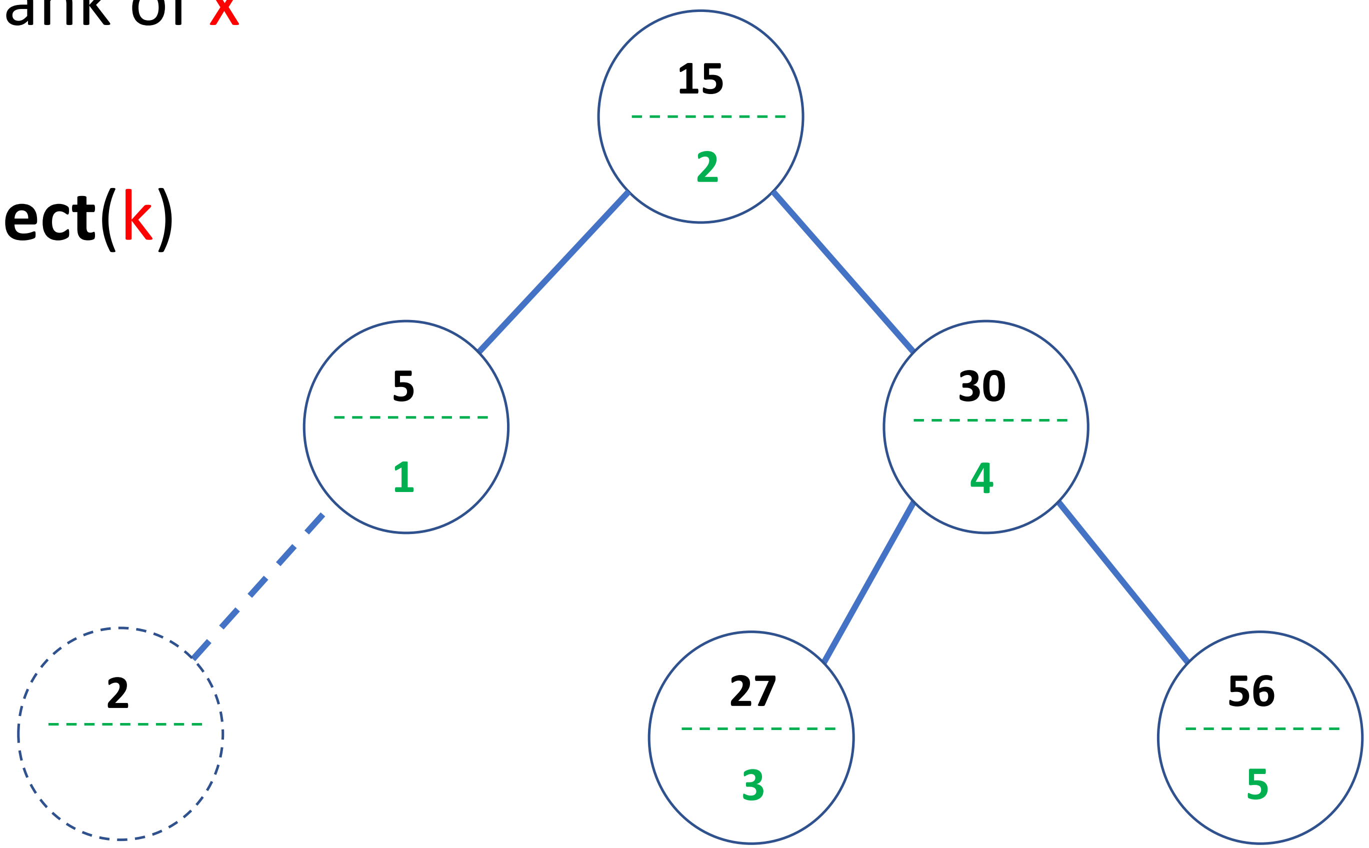
Naïve Augmentation

- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x) and **Select**(k)
- Are we done? No: maintaining the rank field is expensive!



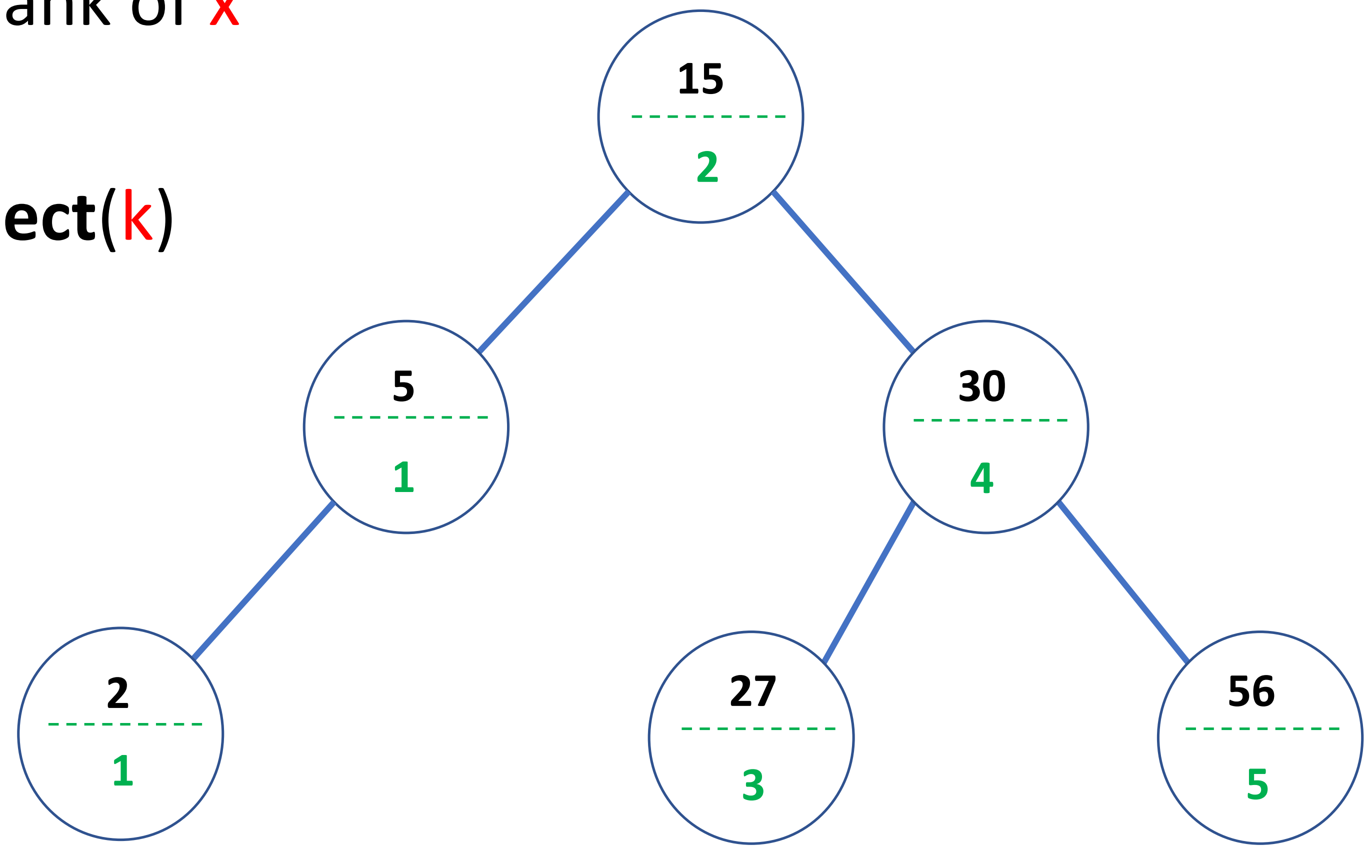
Naïve Augmentation

- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x) and **Select**(k)



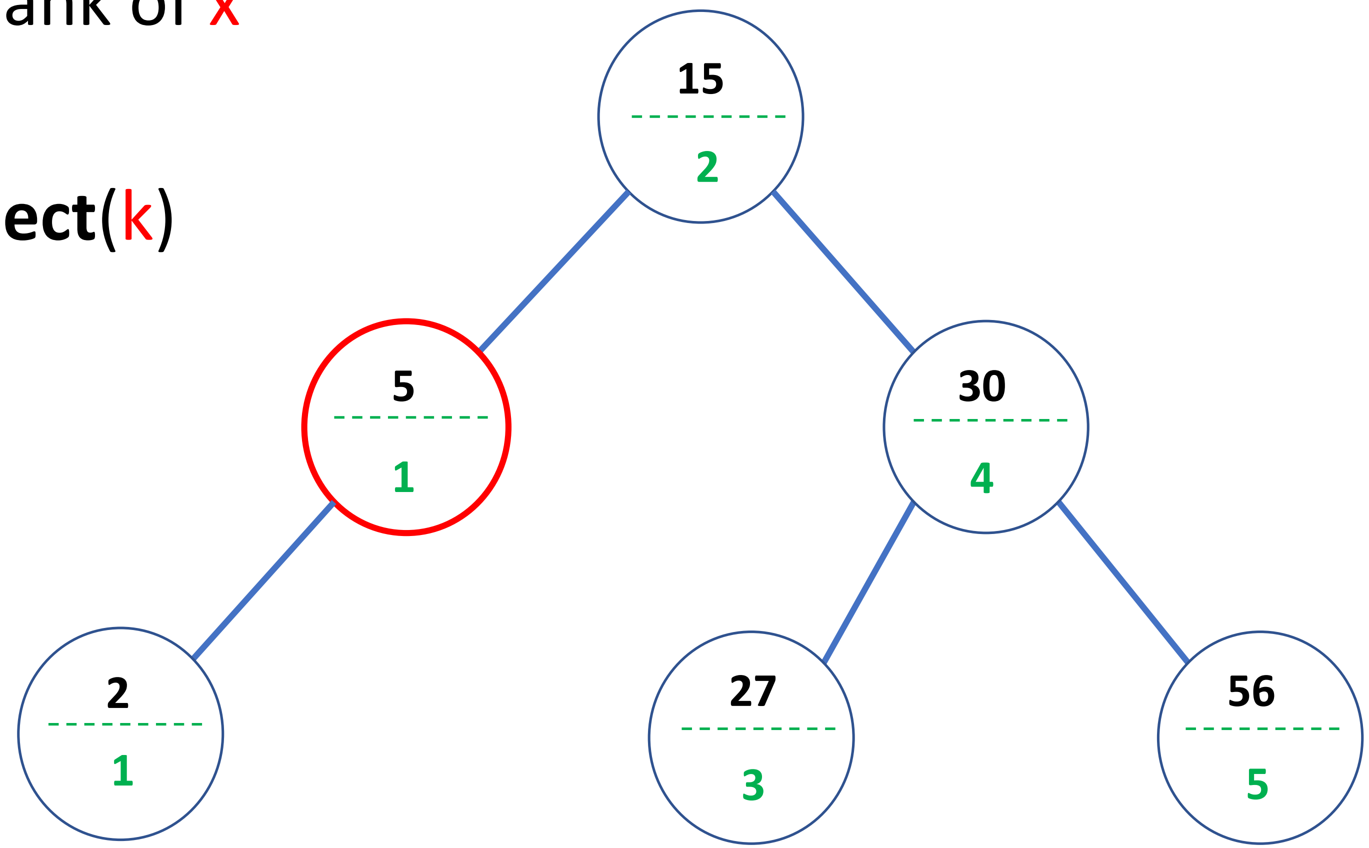
Naïve Augmentation

- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x) and **Select**(k)



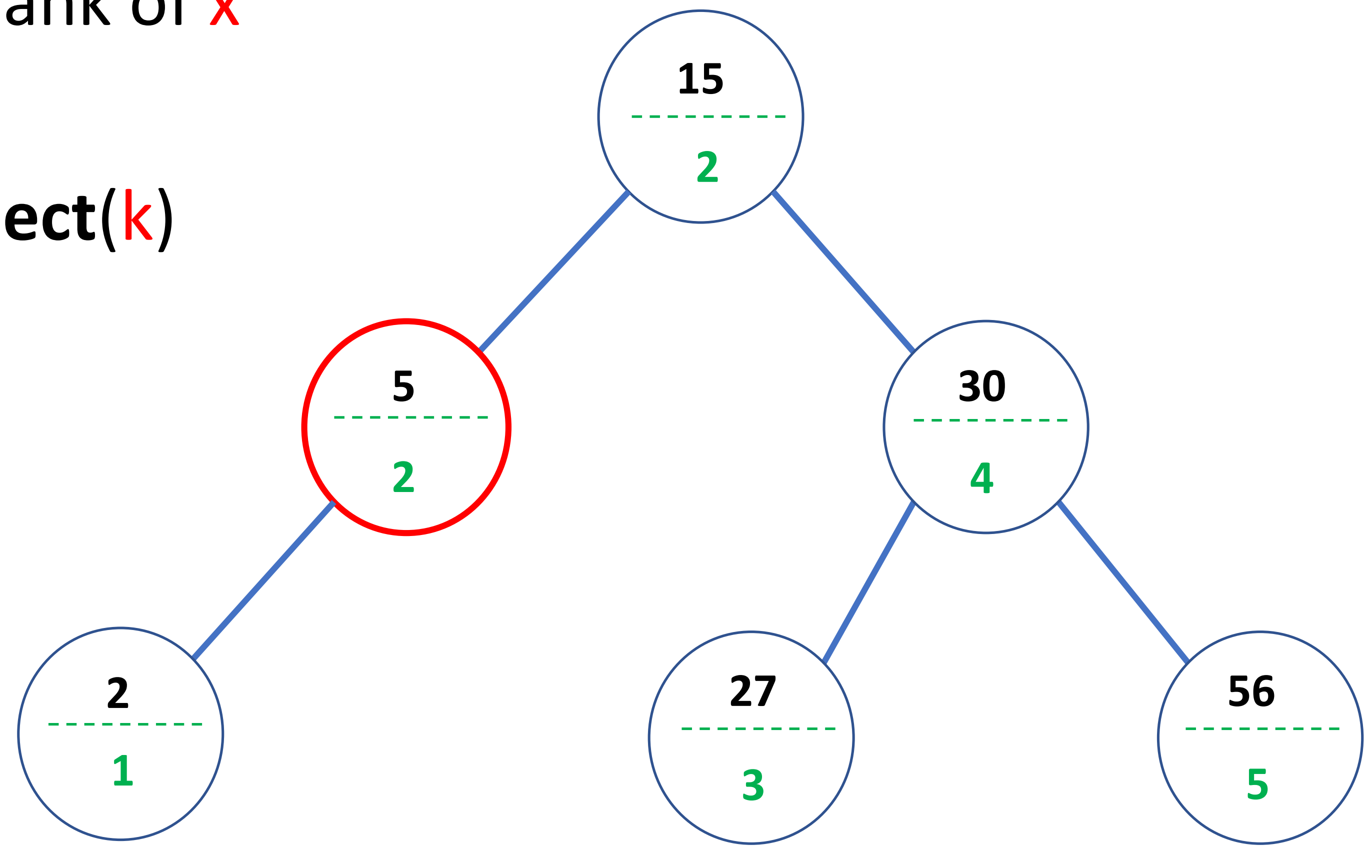
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)



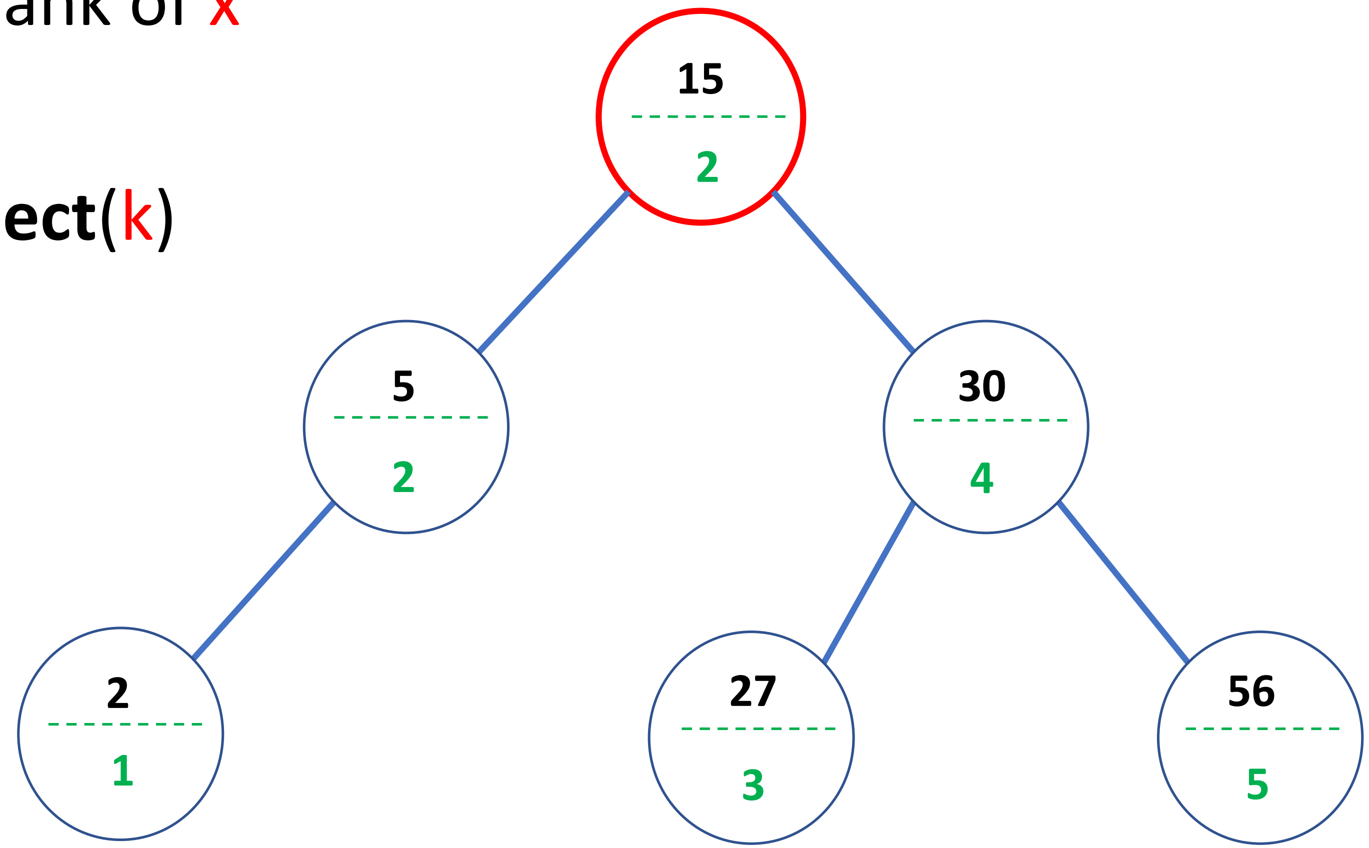
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)



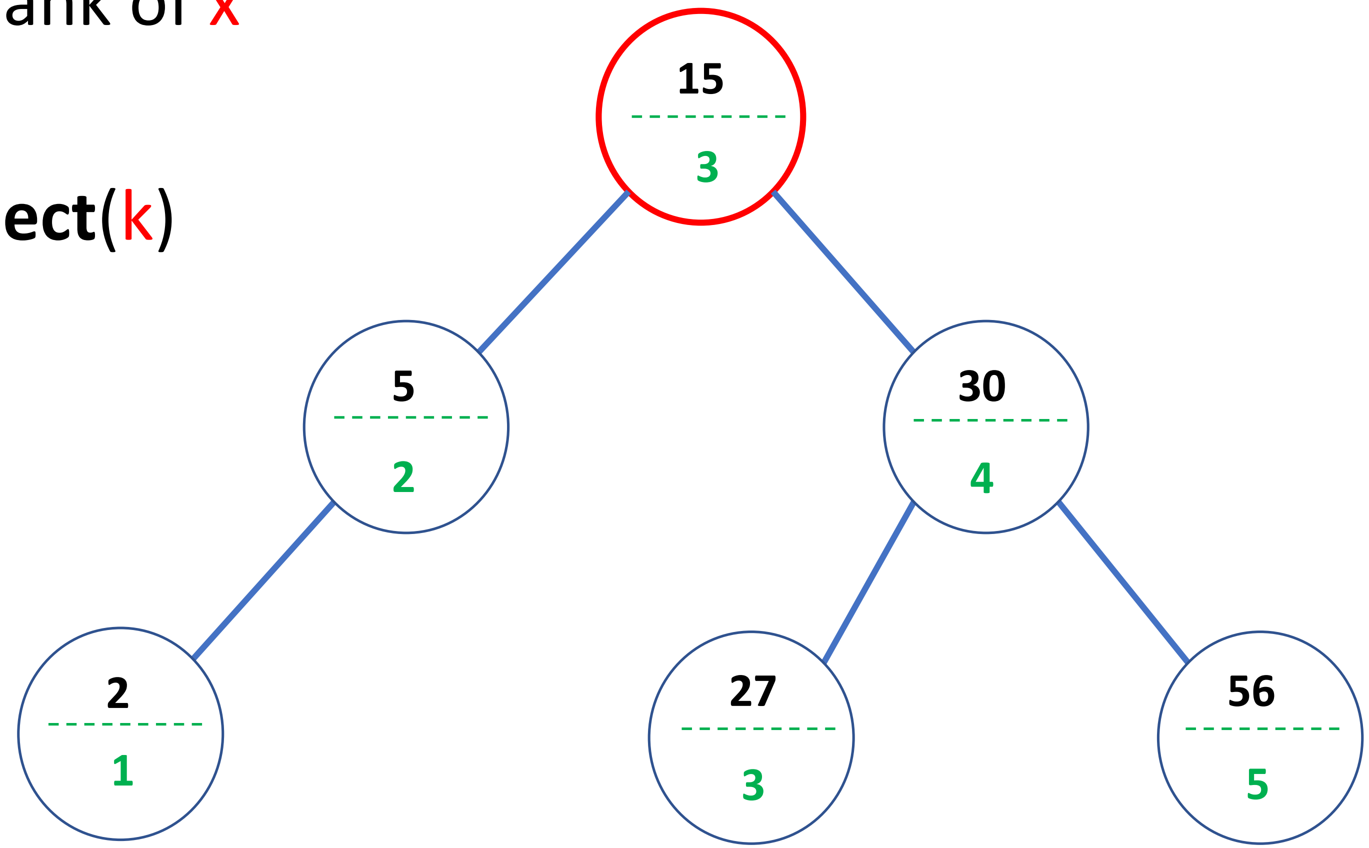
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)



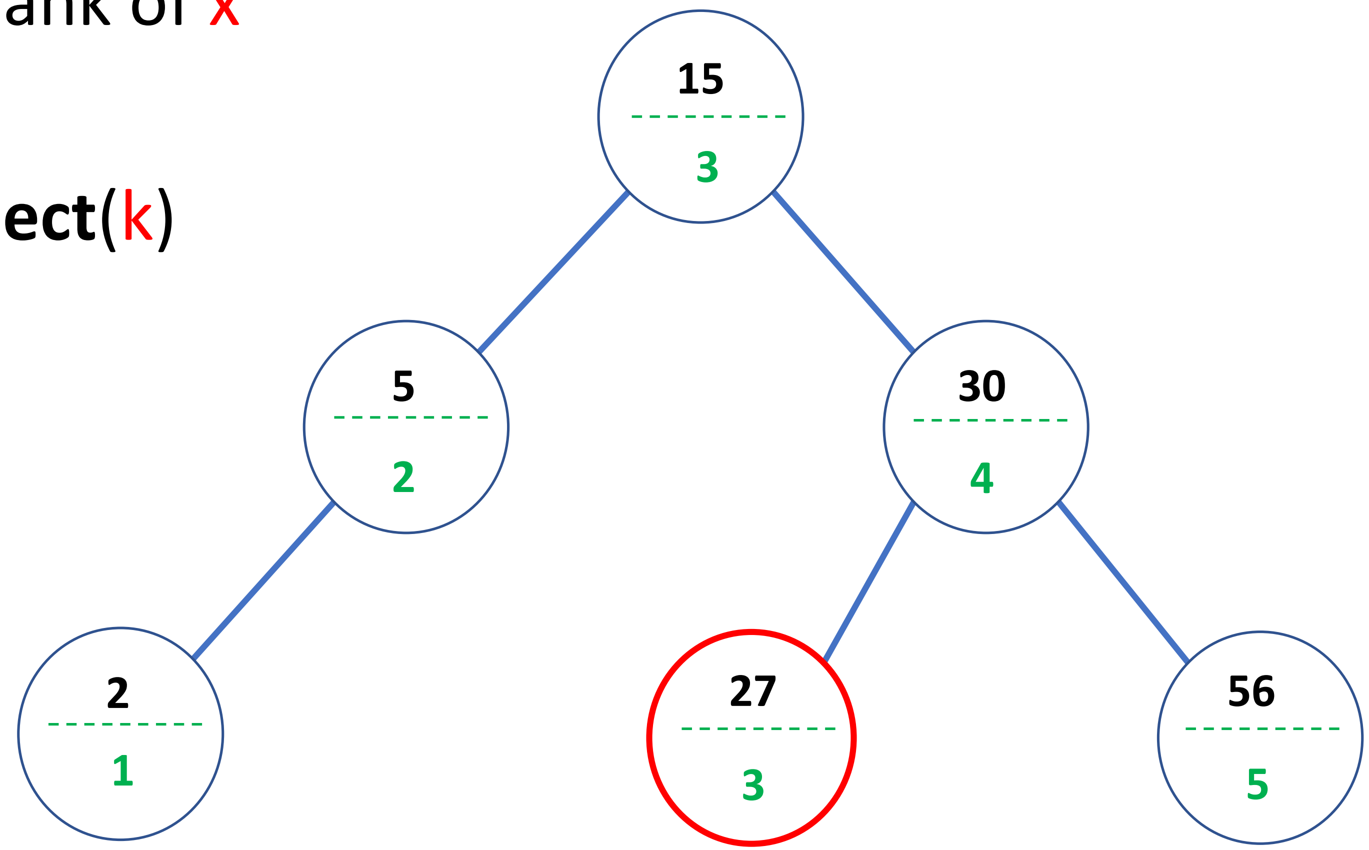
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)



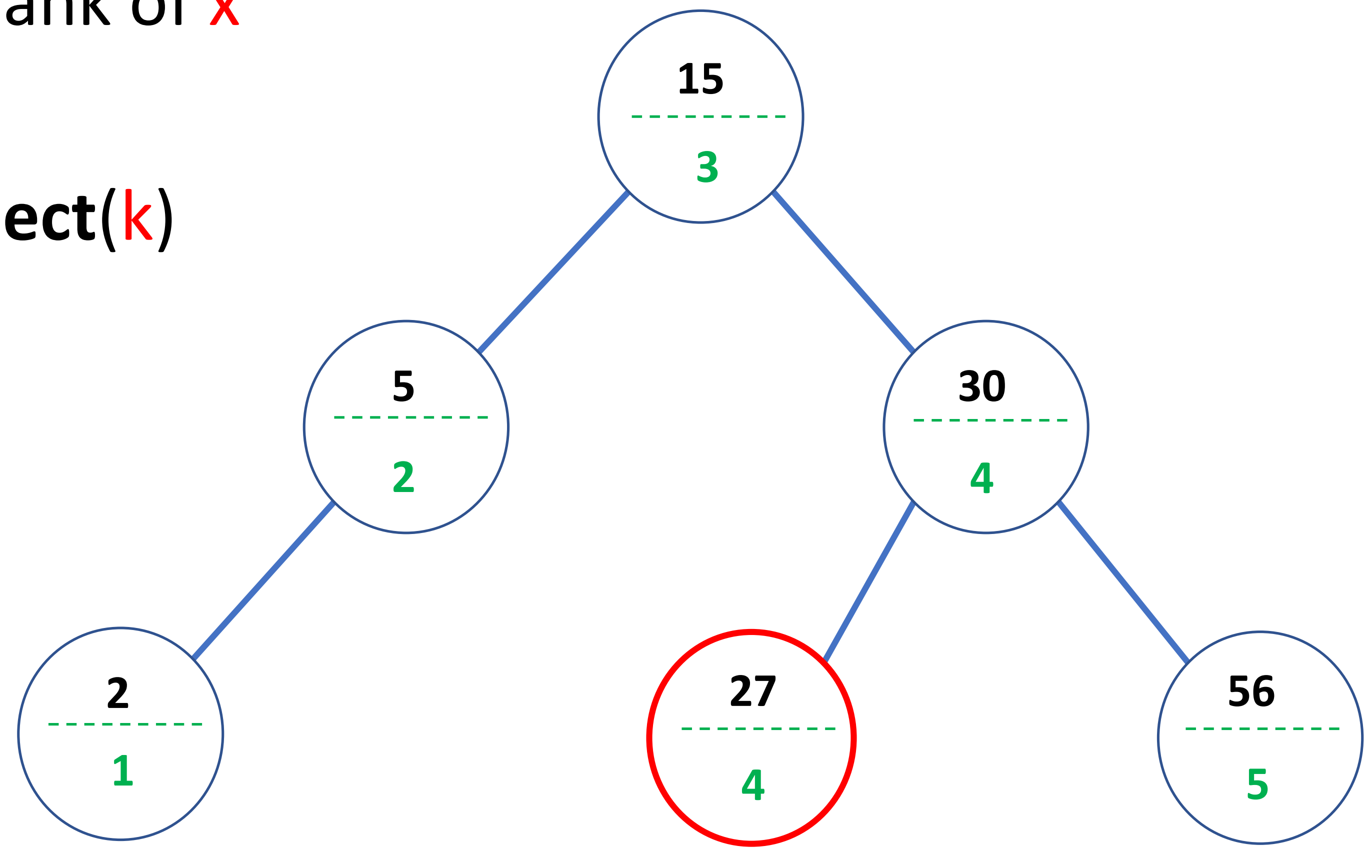
Naïve Augmentation

- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x) and **Select**(k)



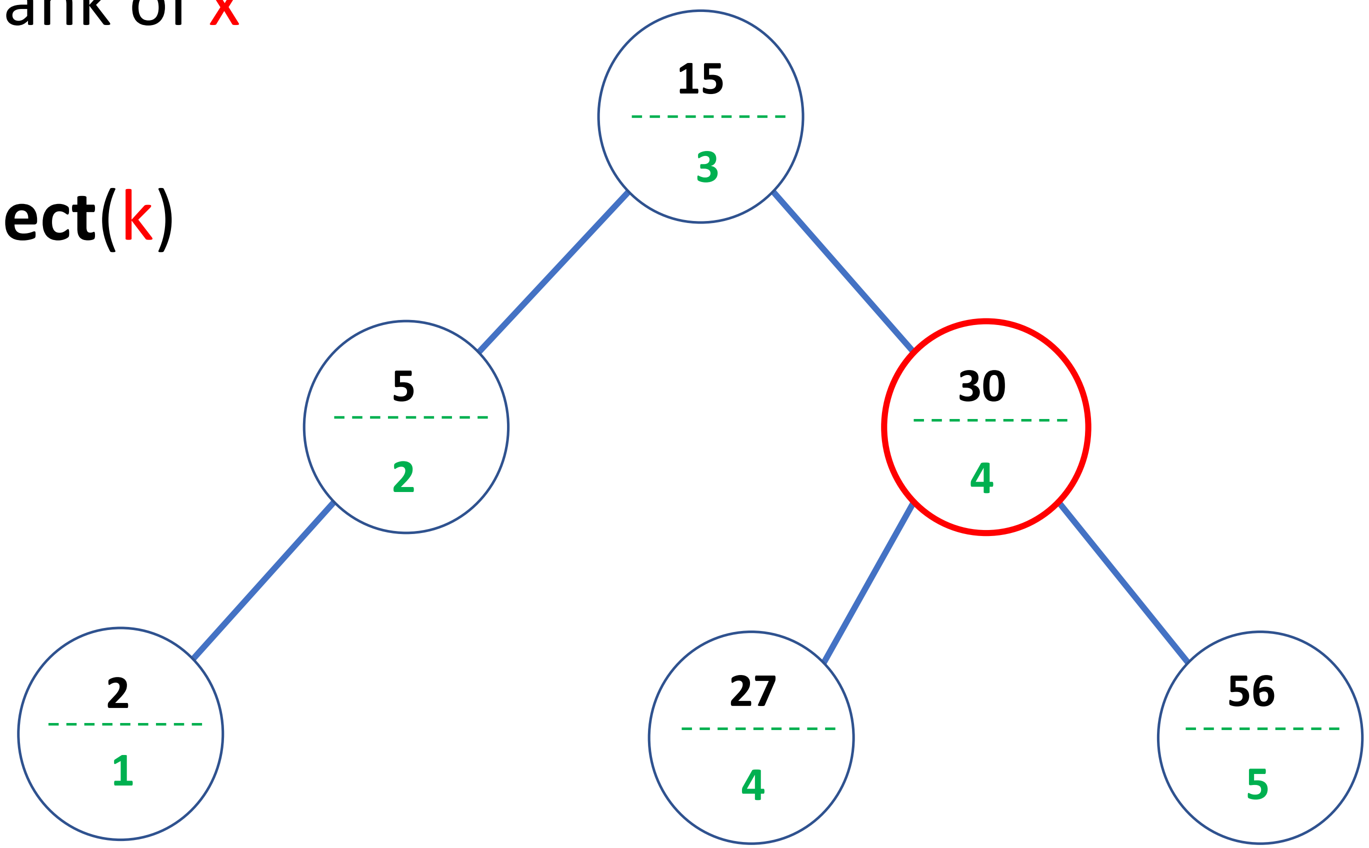
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)



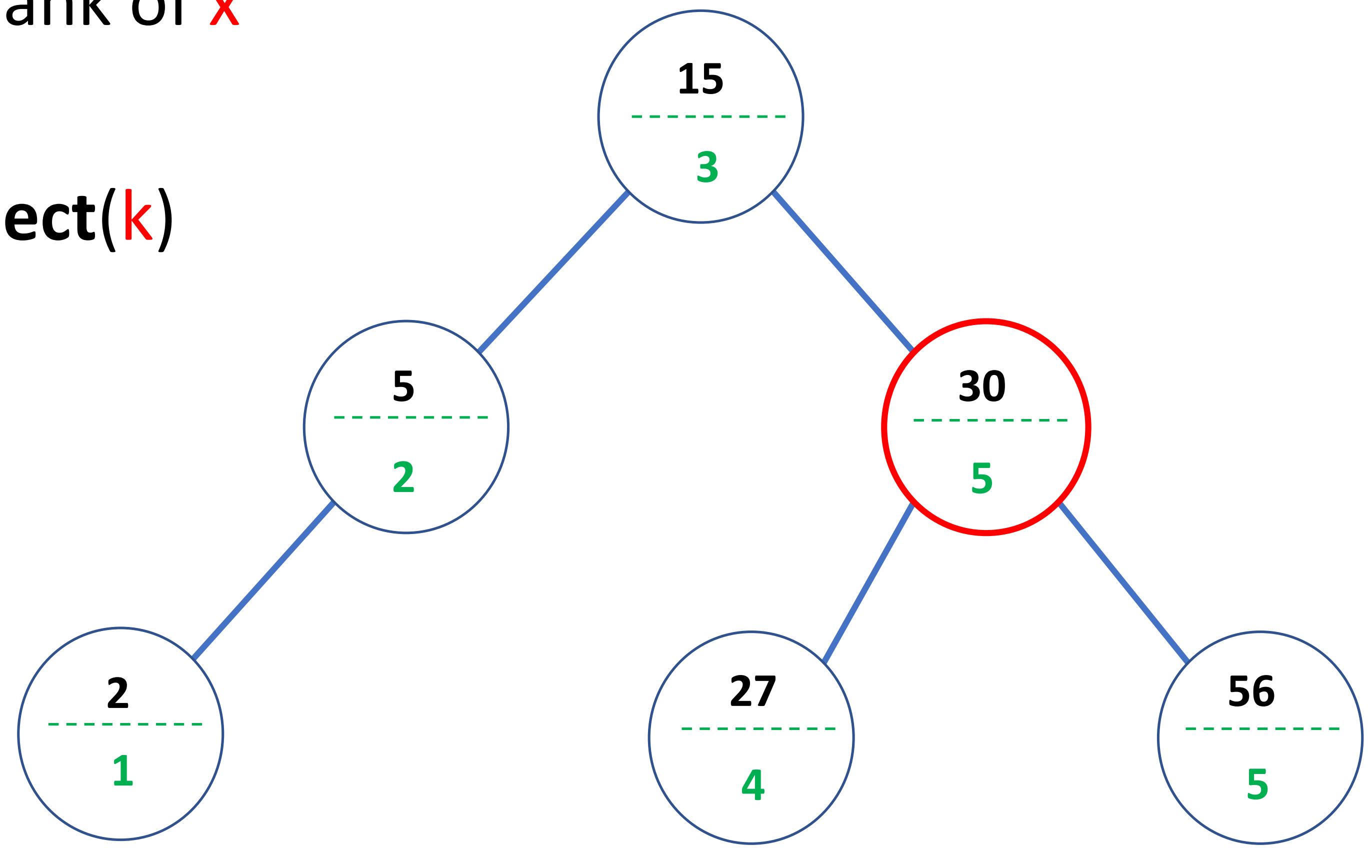
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)



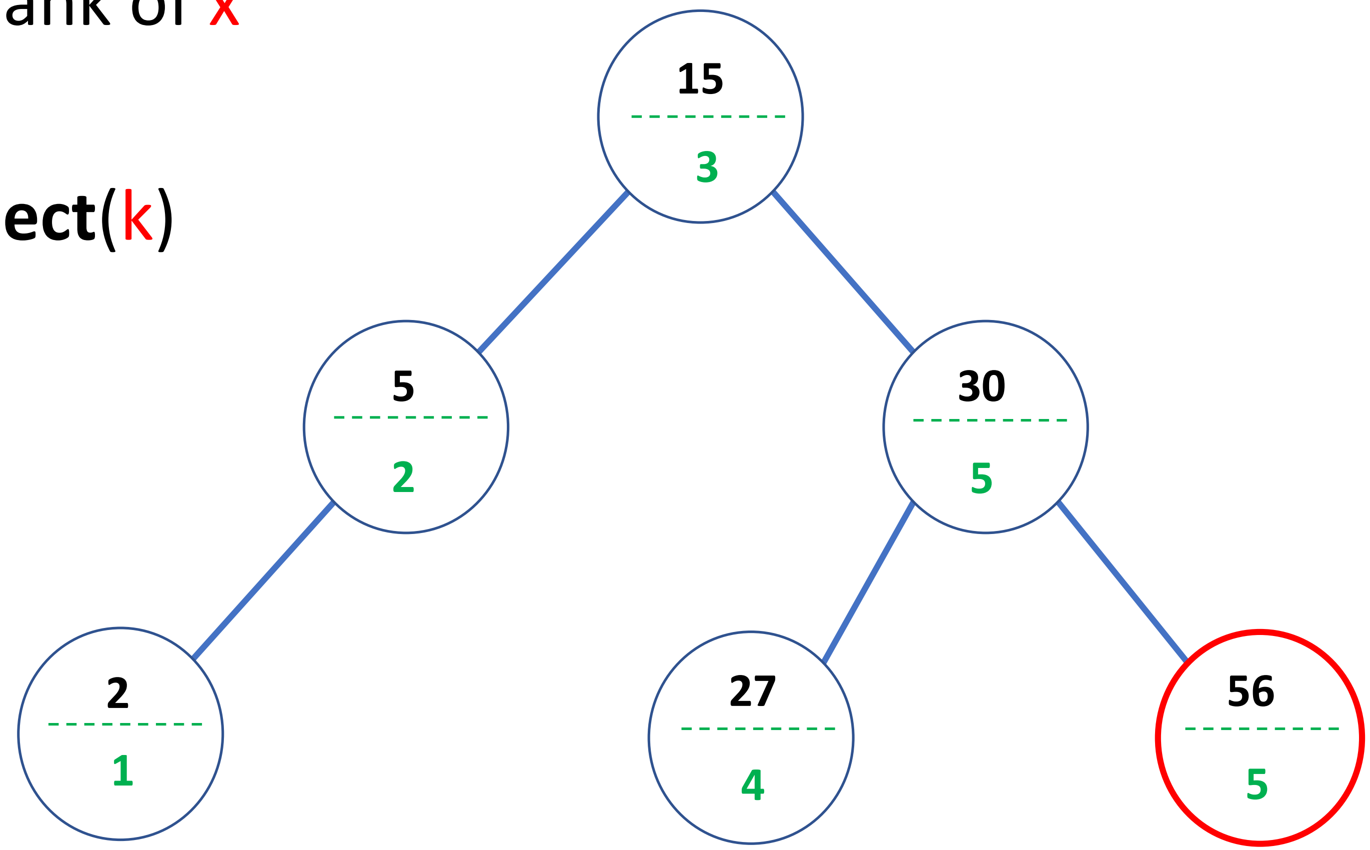
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)



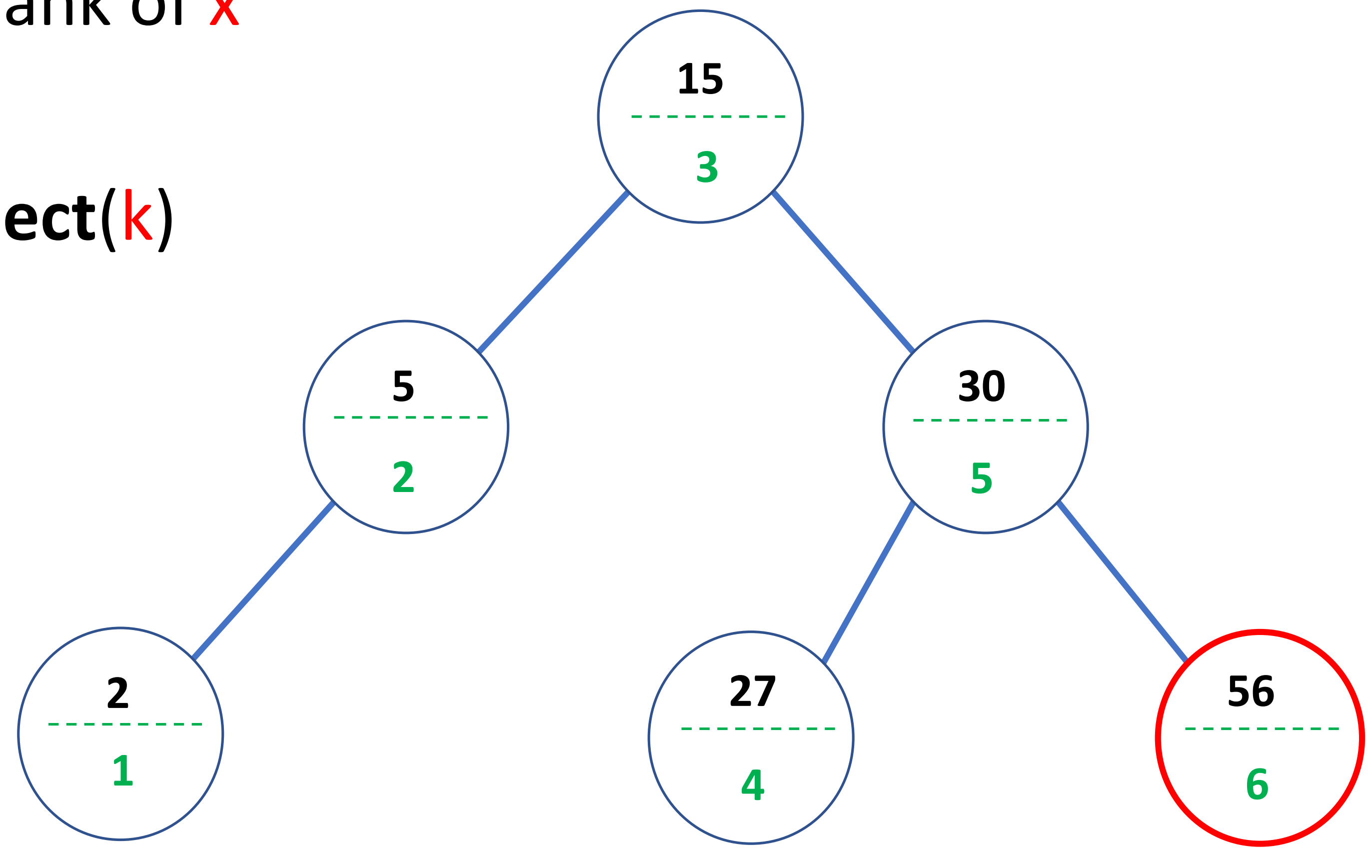
Naïve Augmentation

- At each node x, also store the rank of x
- <u>Good:</u> Efficient **Rank**(x) and **Select**(k)



Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)

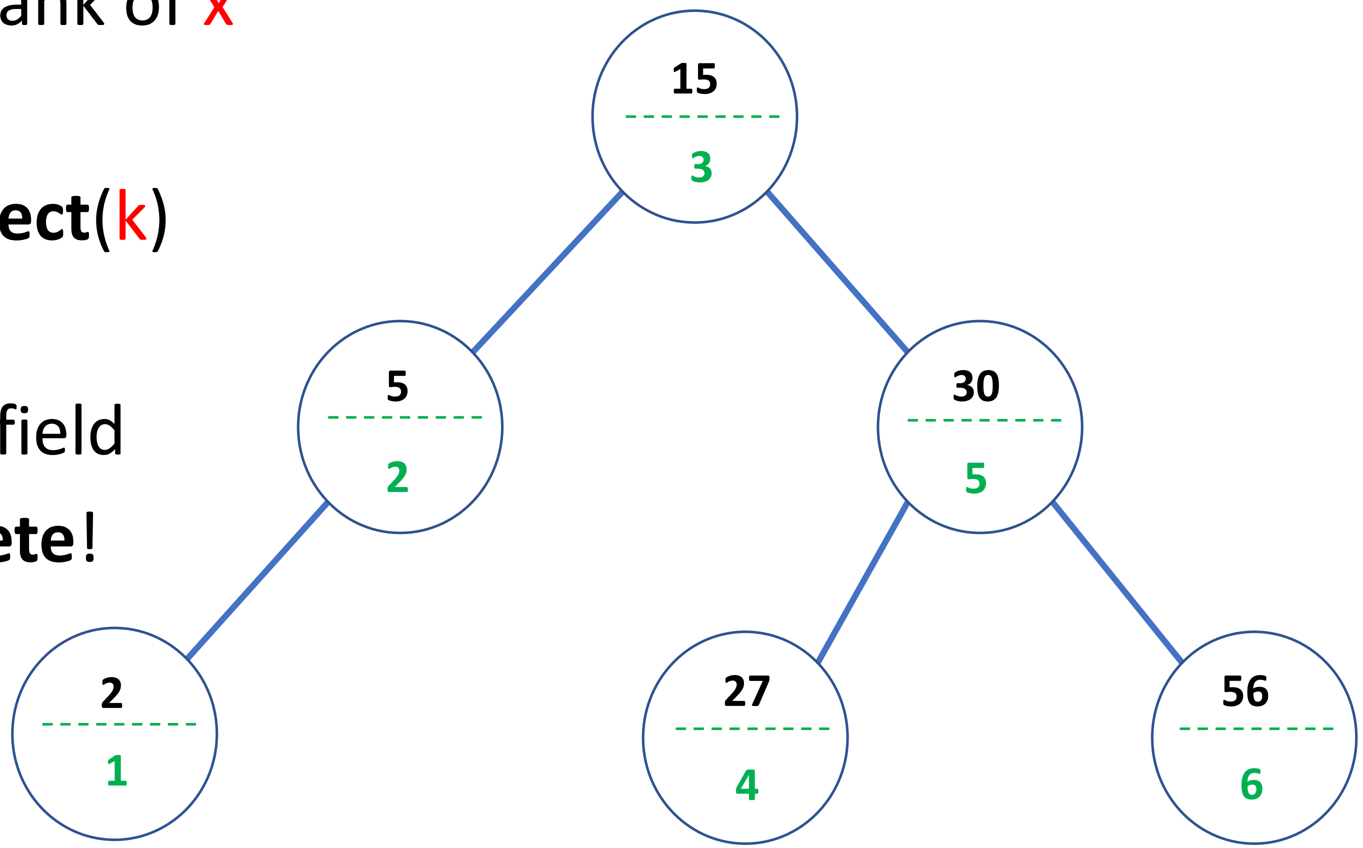


Naïve Augmentation

- At each node x, also store the rank of x

- Good: Efficient **Rank**(x) and **Select**(k)

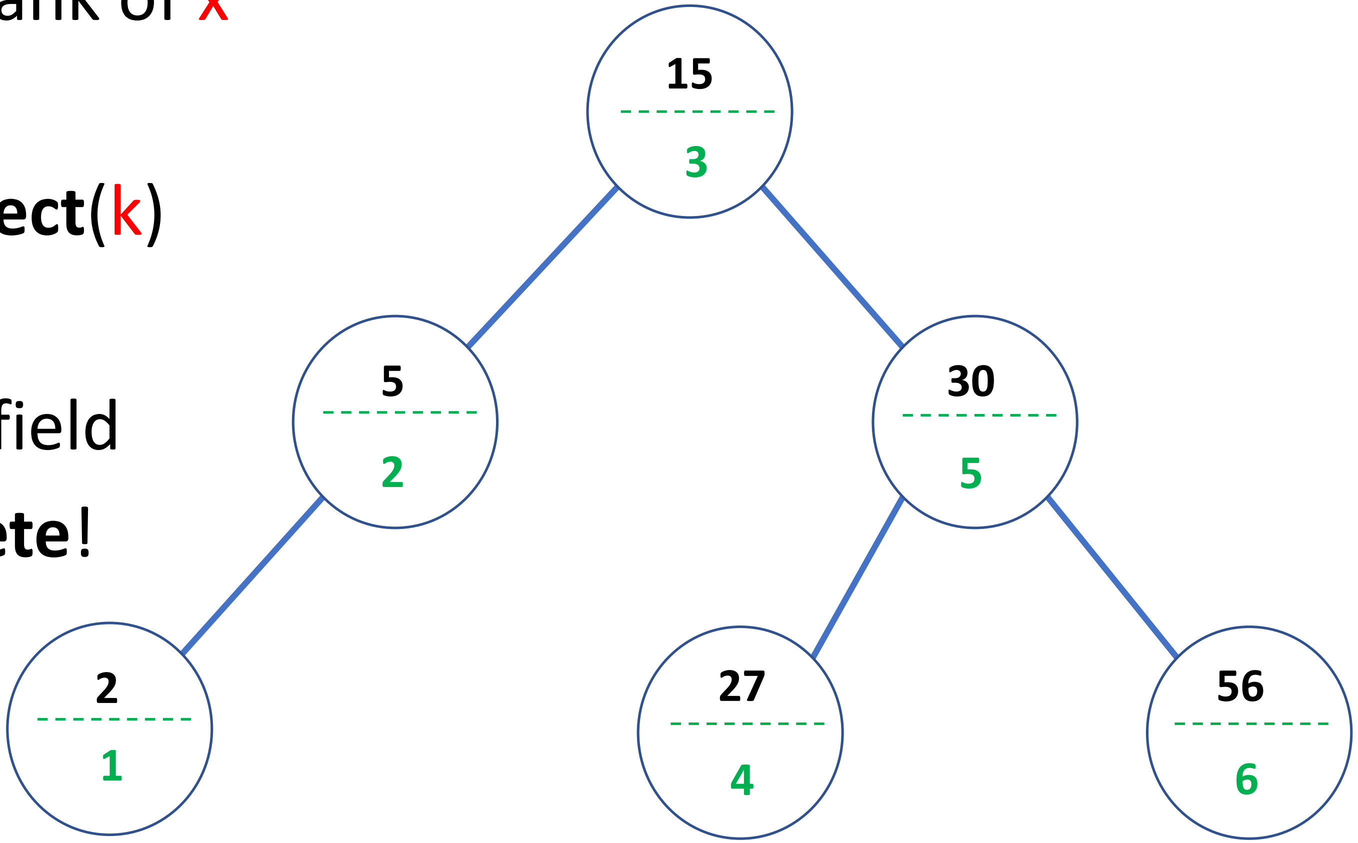
- Bad: Expensive to update rank field
  when doing **Insert** or **Delete**!



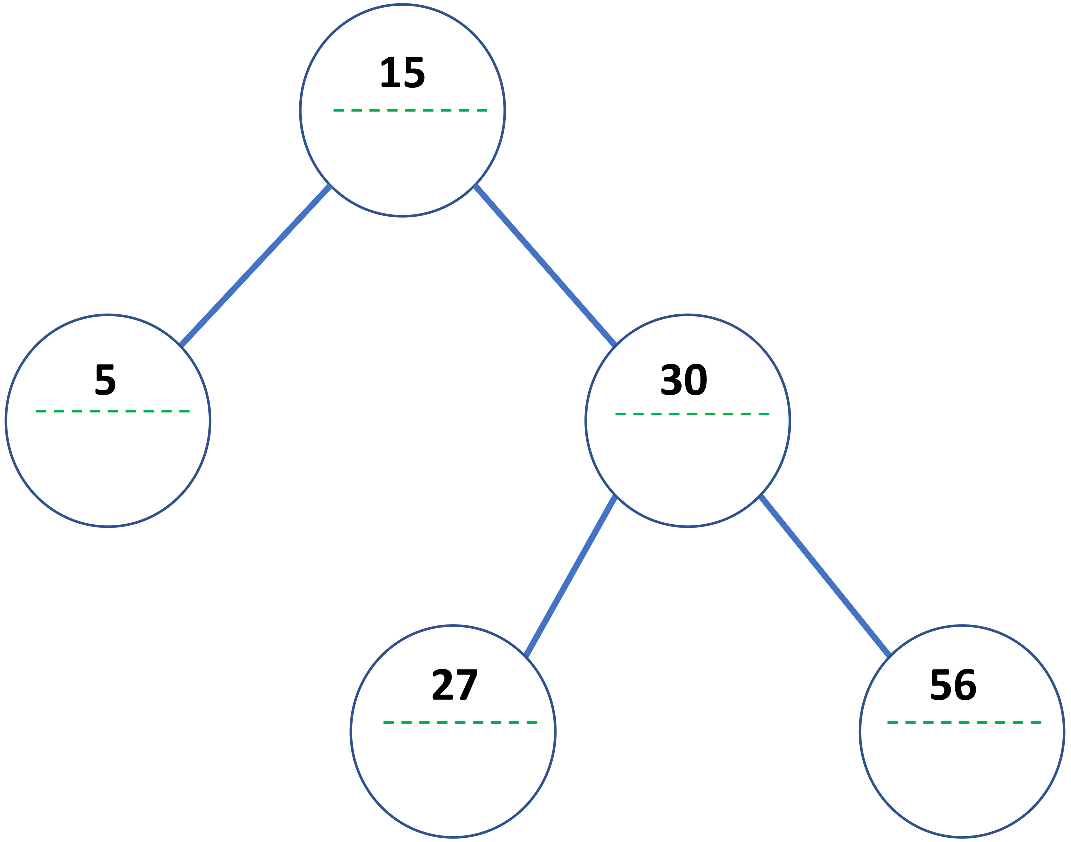
Naïve Augmentation

- At each node x, also store the rank of x
- Good: Efficient **Rank**(x) and **Select**(k)
- Bad: Expensive to update rank field when doing **Insert** or **Delete**!

Takes O(n) per insert/delete

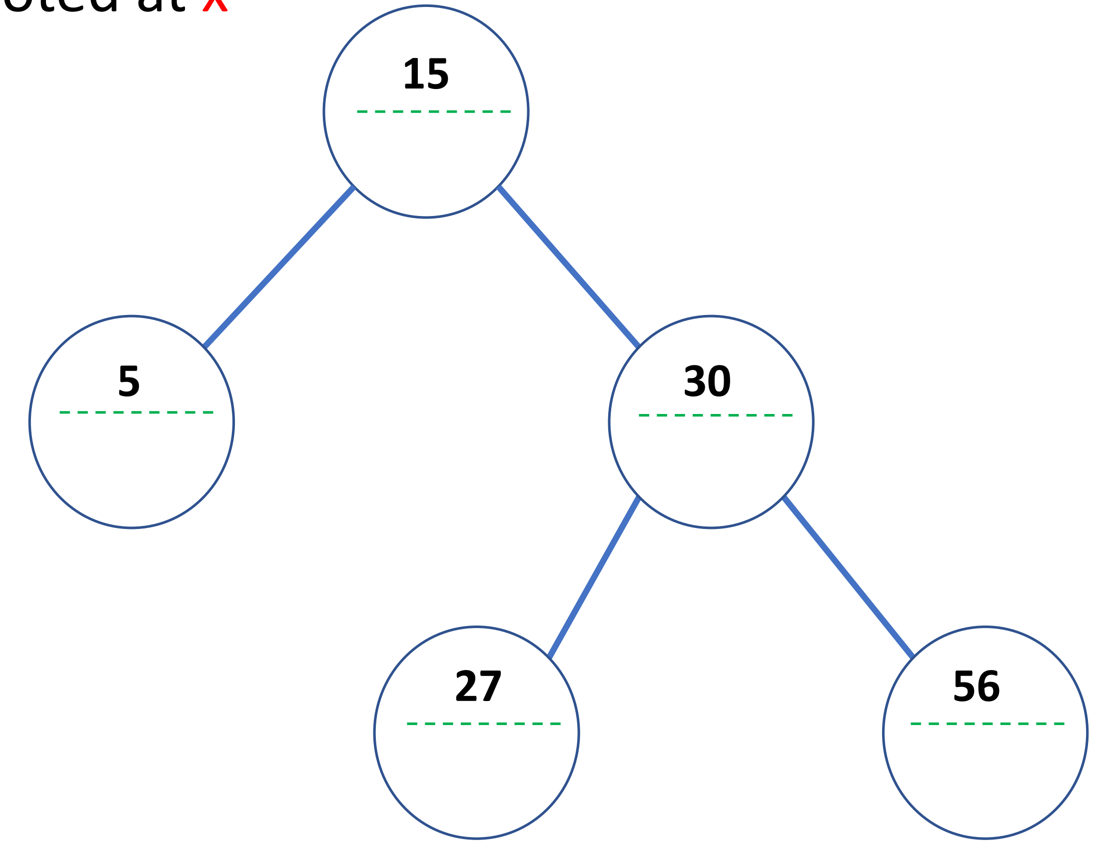


A better augmentation



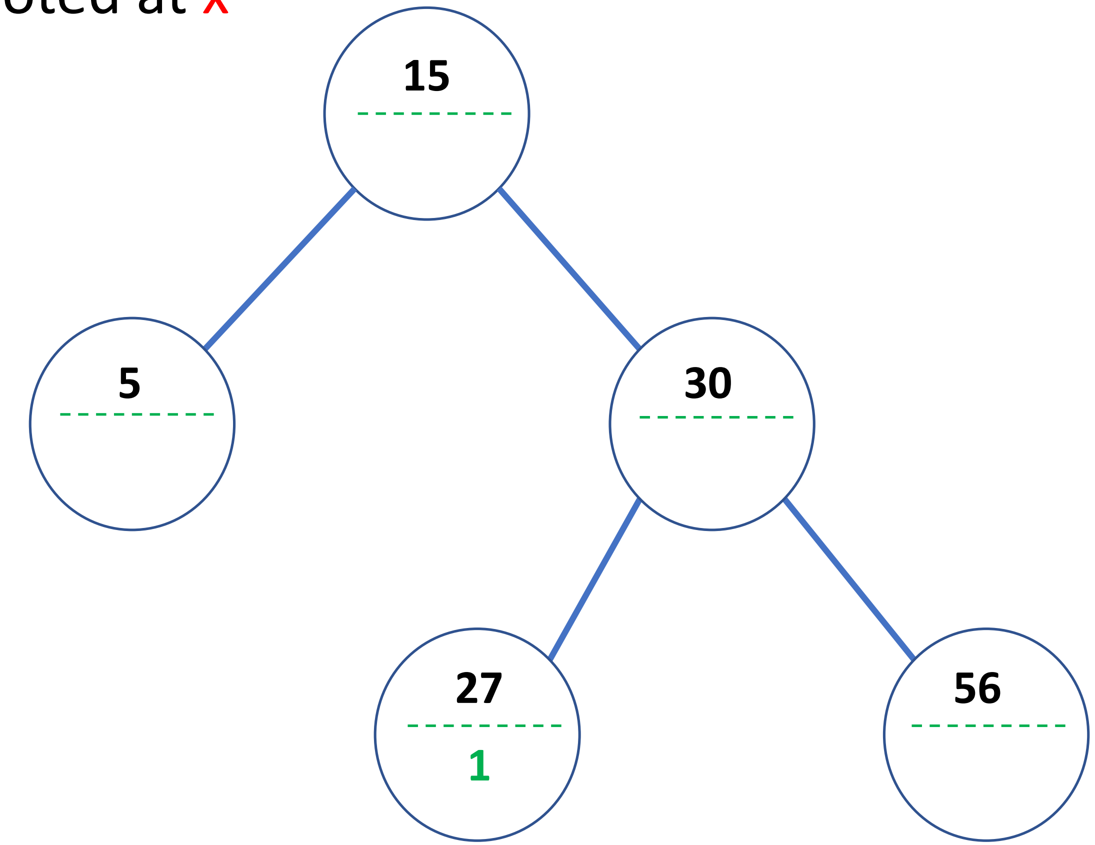
A better augmentation

- At each node x, store the size of the subtree rooted at x



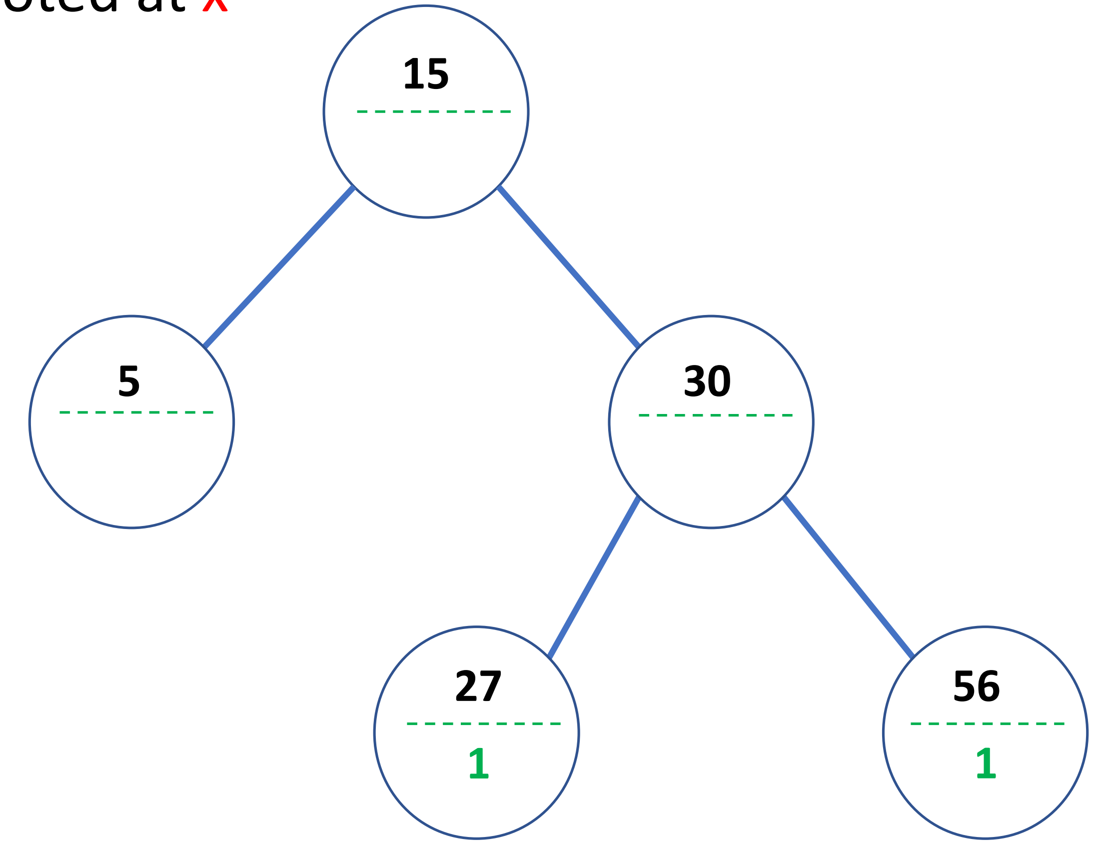
A better augmentation

- At each node x, store the size of the subtree rooted at x



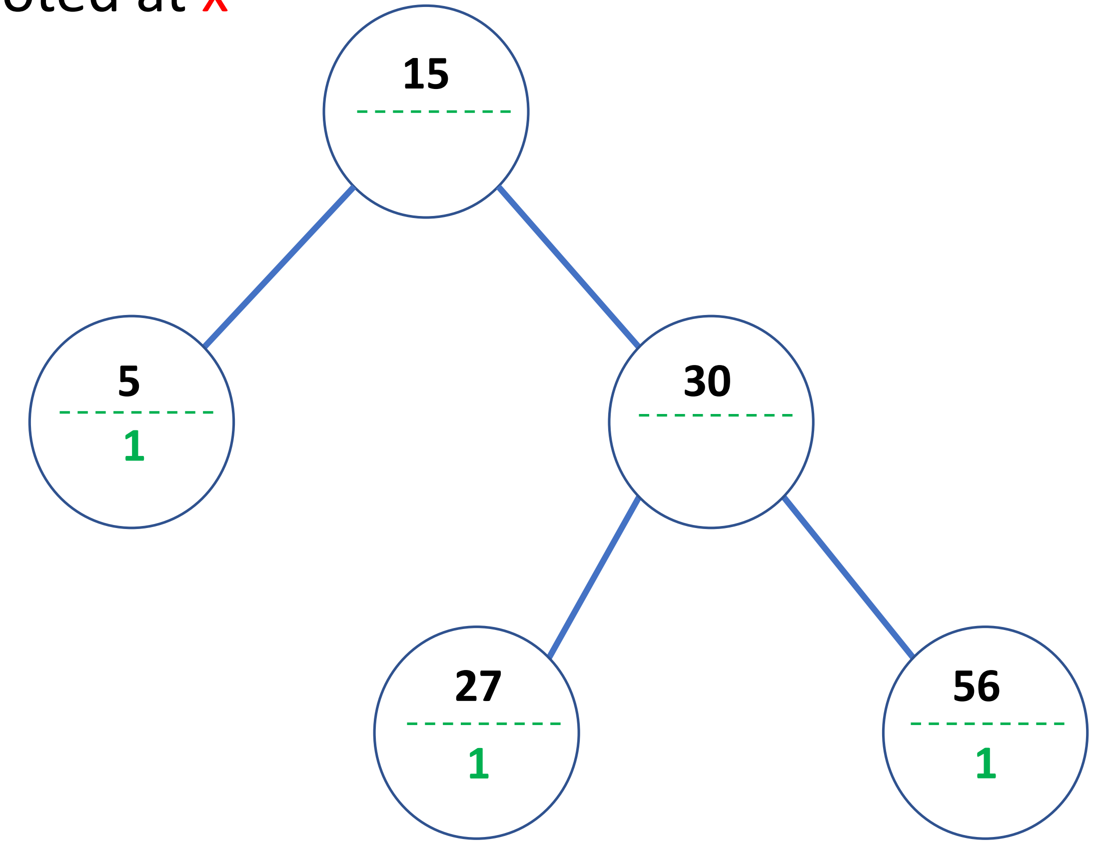
A better augmentation

- At each node x, store the size of the subtree rooted at x



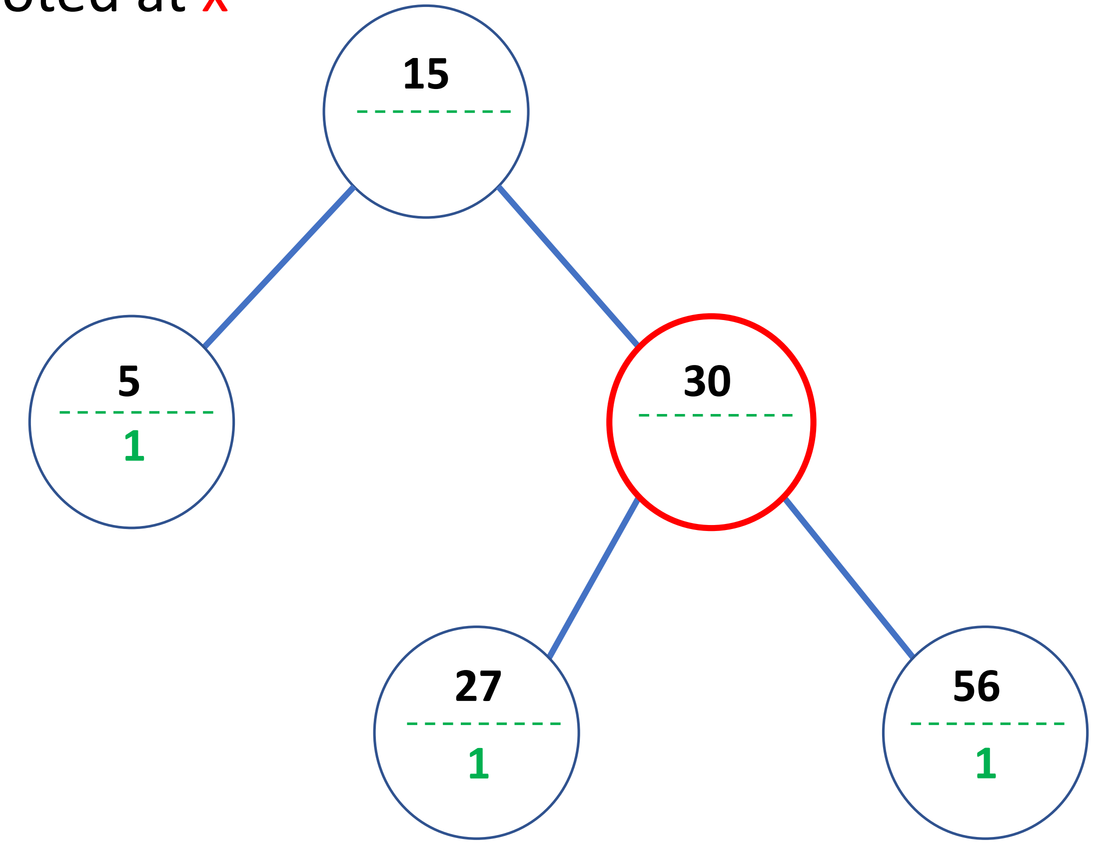
A better augmentation

- At each node x, store the size of the subtree rooted at x



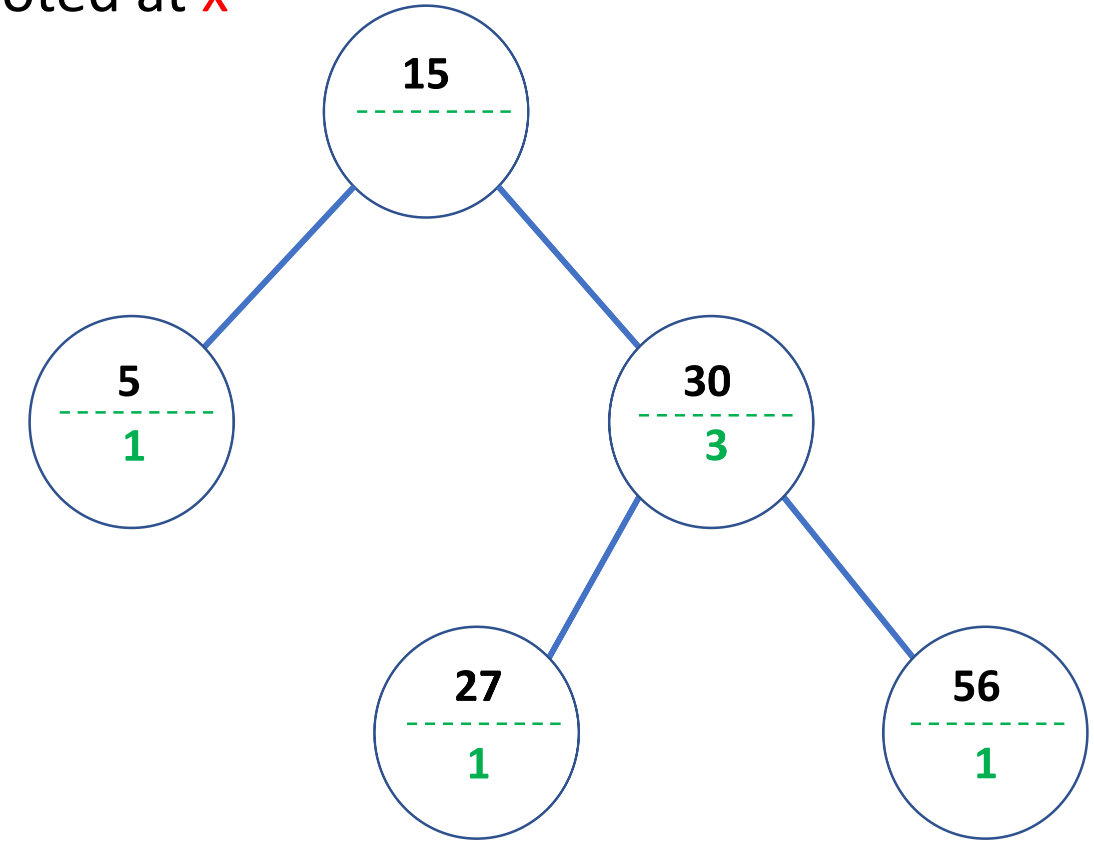
A better augmentation

- At each node x, store the size of the subtree rooted at x



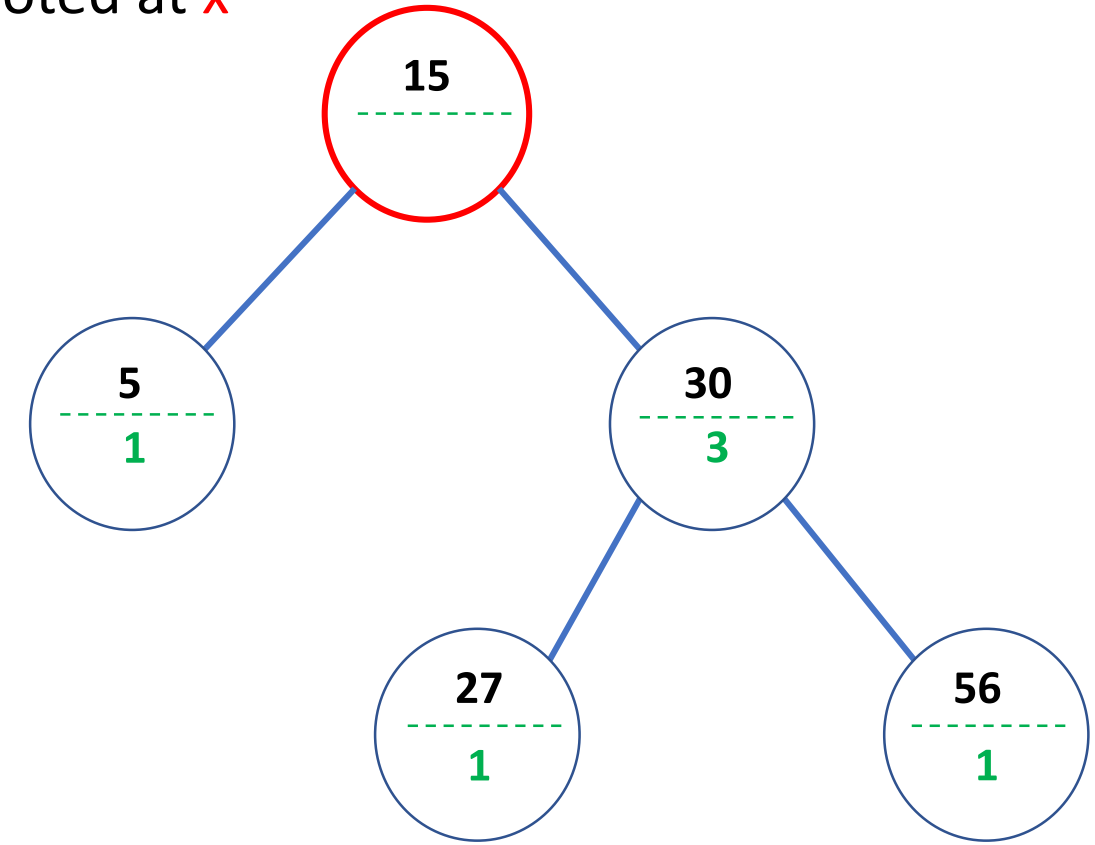
A better augmentation

- At each node x, store the size of the subtree rooted at x



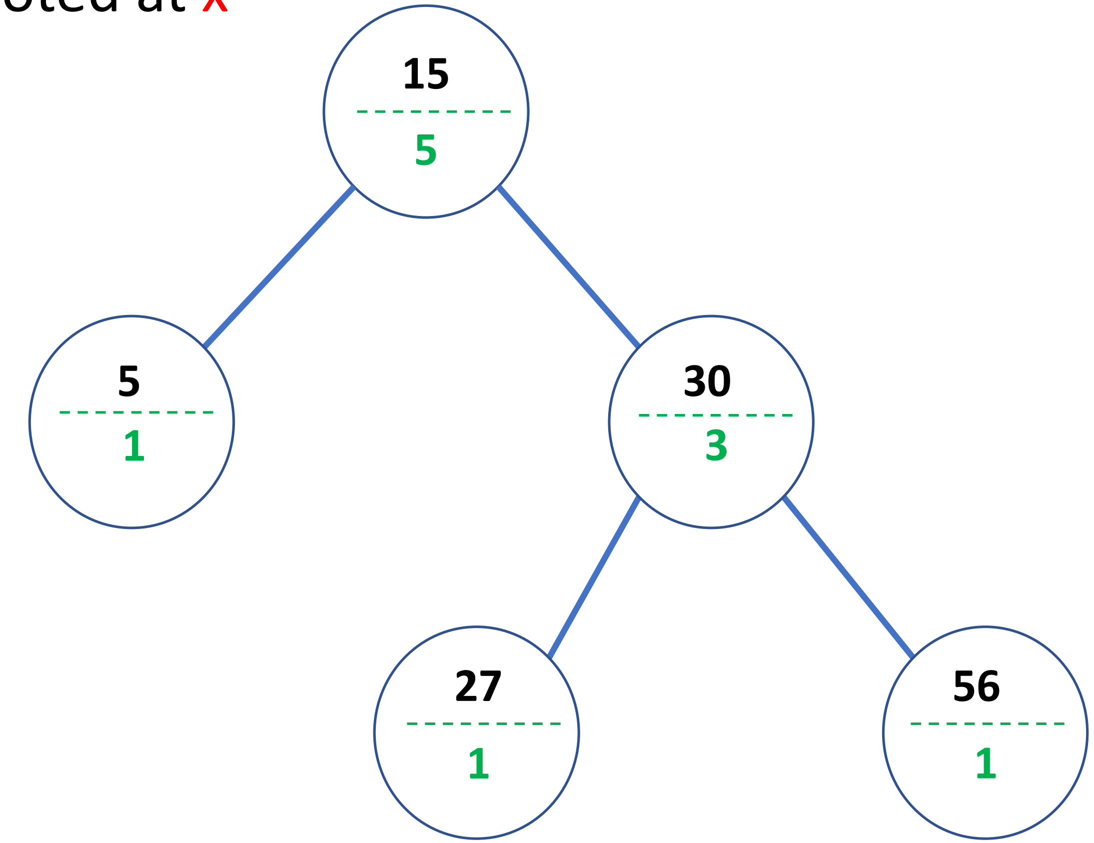
A better augmentation

- At each node x, store the size of the subtree rooted at x



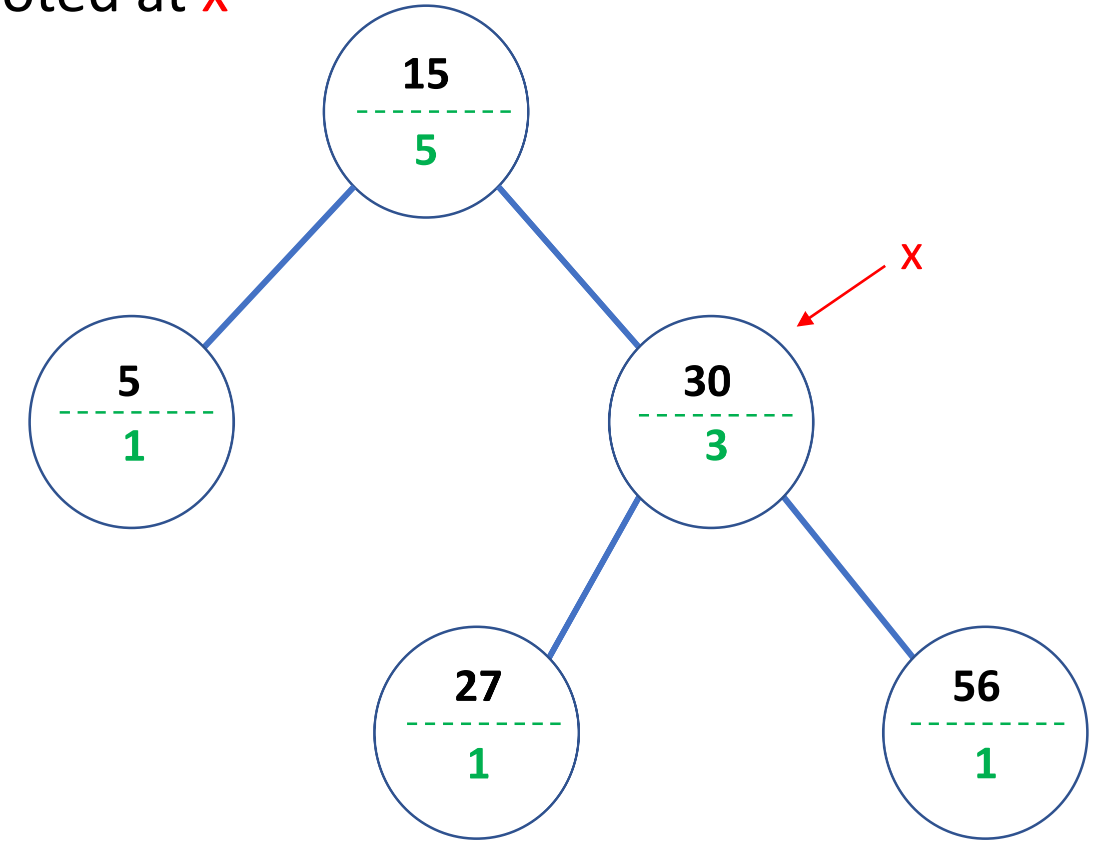
A better augmentation

- At each node x, store the size of the subtree rooted at x



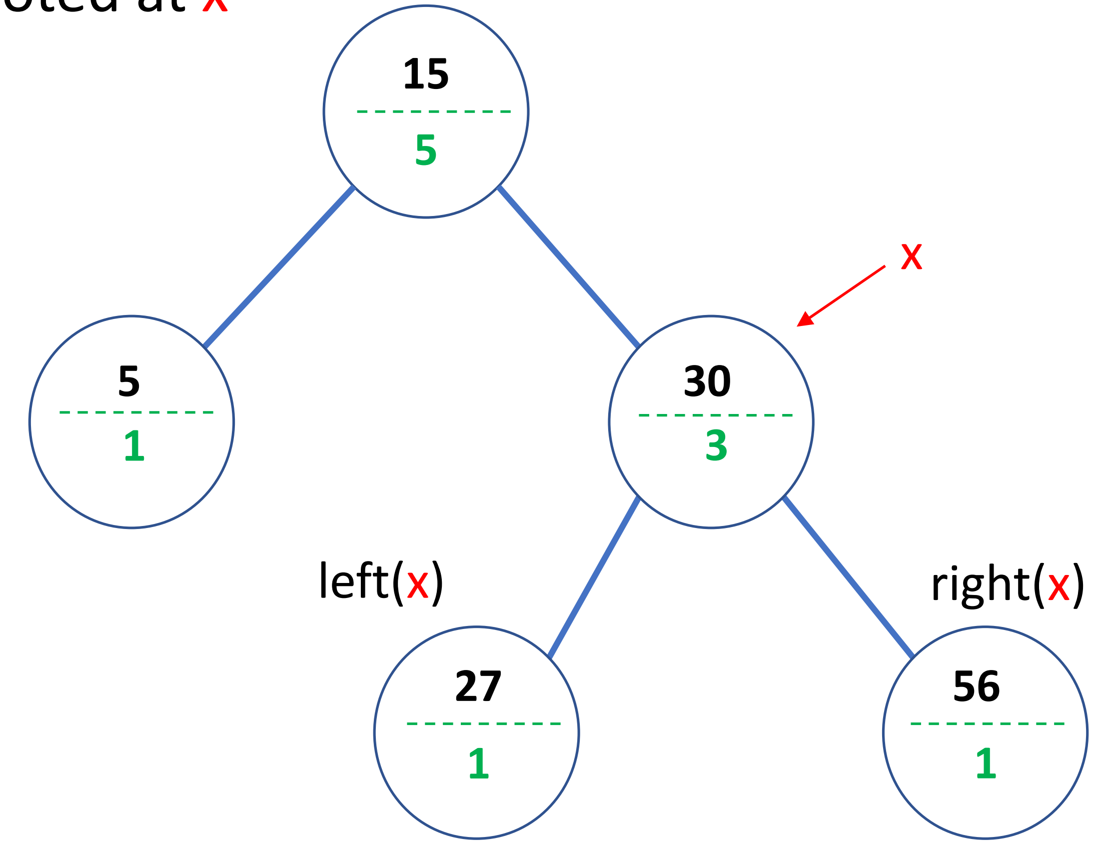
A better augmentation

- At each node x, store the size of the subtree rooted at x



A better augmentation

- At each node x, store the size of the subtree rooted at x

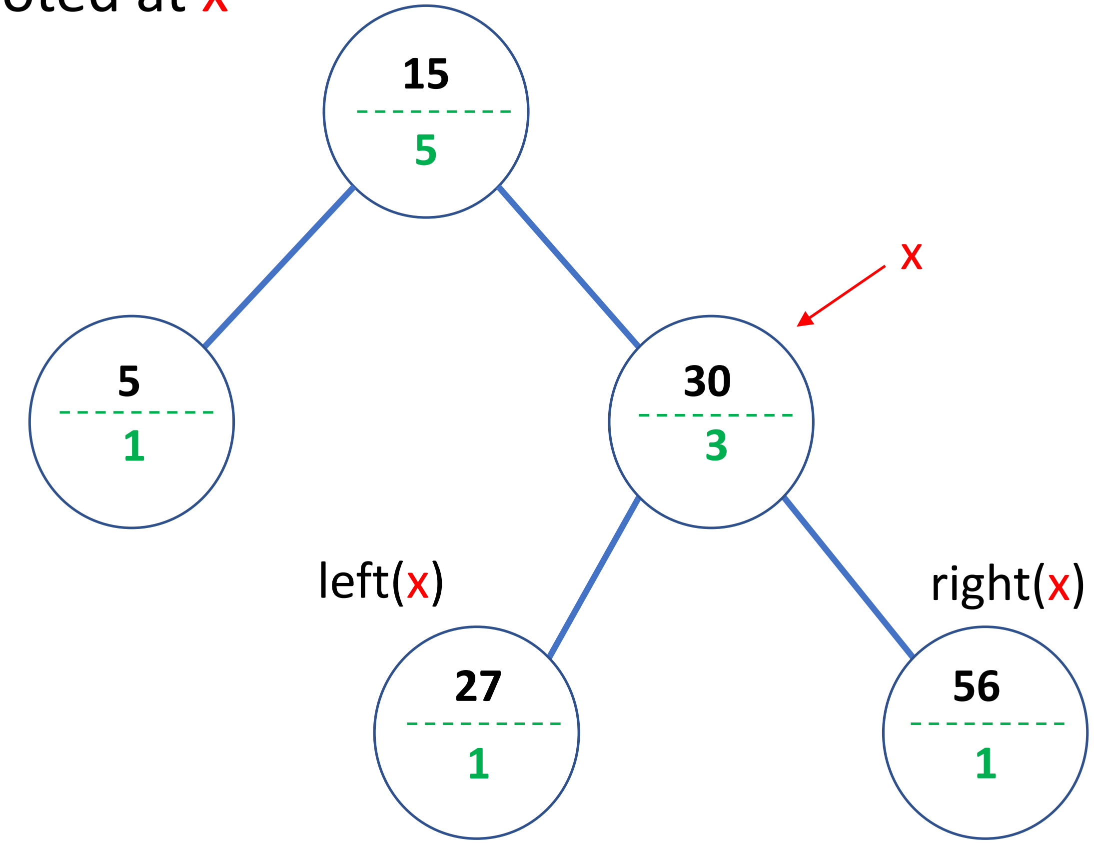


A better augmentation

- At each node x, store the size of the subtree rooted at x

For every node x,

$$\boxed{\mathbf{size(x) = size(left(x)) + size(right(x)) + 1}}$$



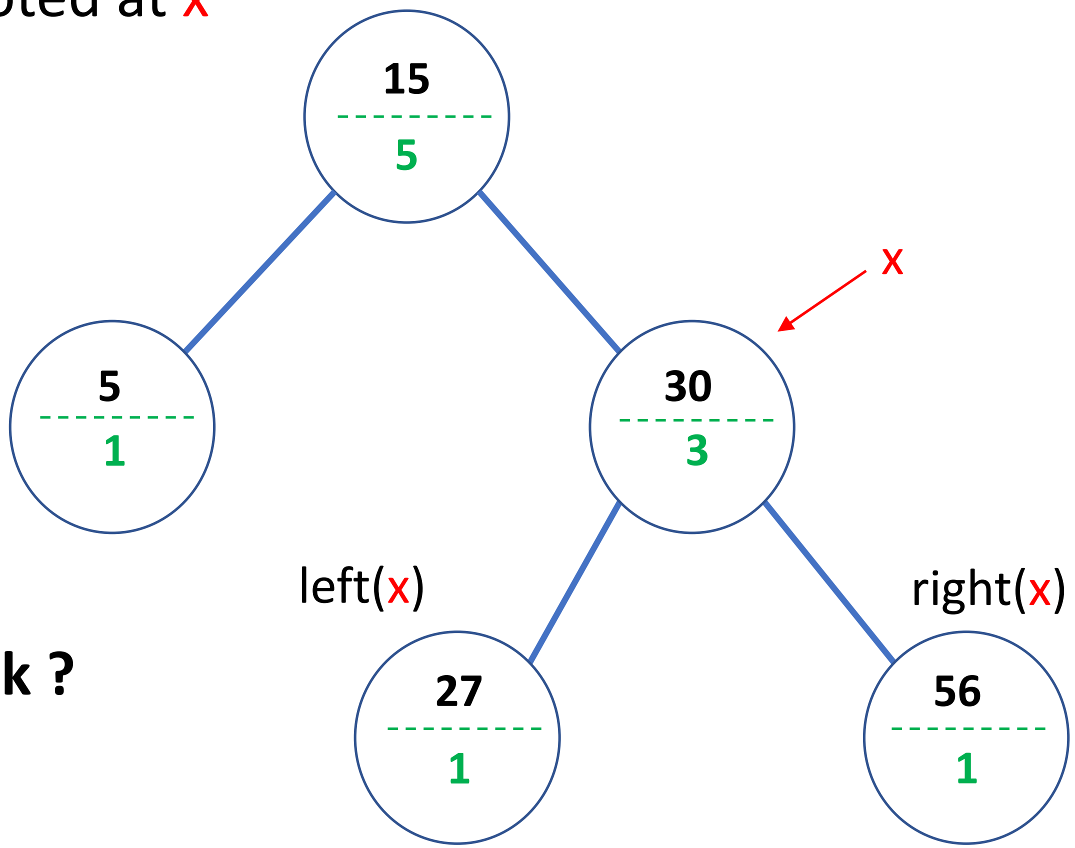
A better augmentation

- At each node x, store the size of the subtree rooted at x

For every node x,

$$\boxed{\textbf{size}(x) = \textbf{size}(\textbf{left}(x)) + \textbf{size}(\textbf{right}(x)) + 1}$$

1. How to efficiently implement **Select** and **Rank** ?

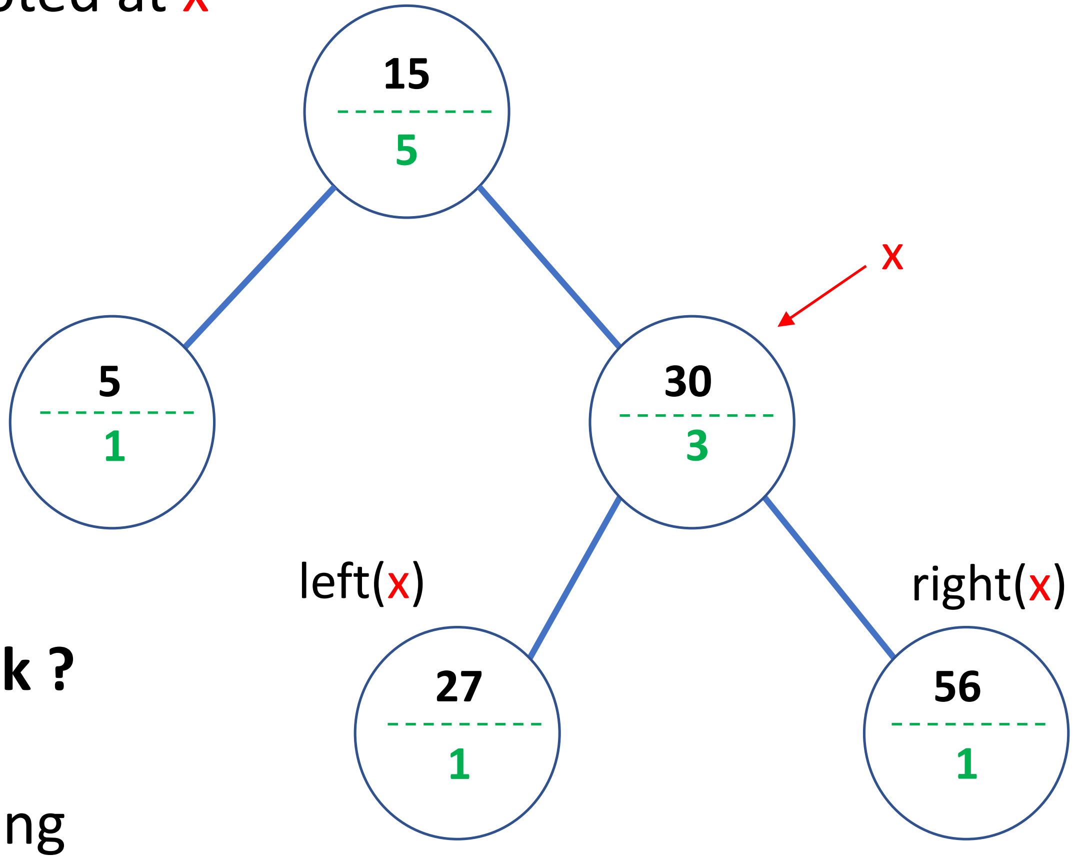


A better augmentation

- At each node x, store the size of the subtree rooted at x

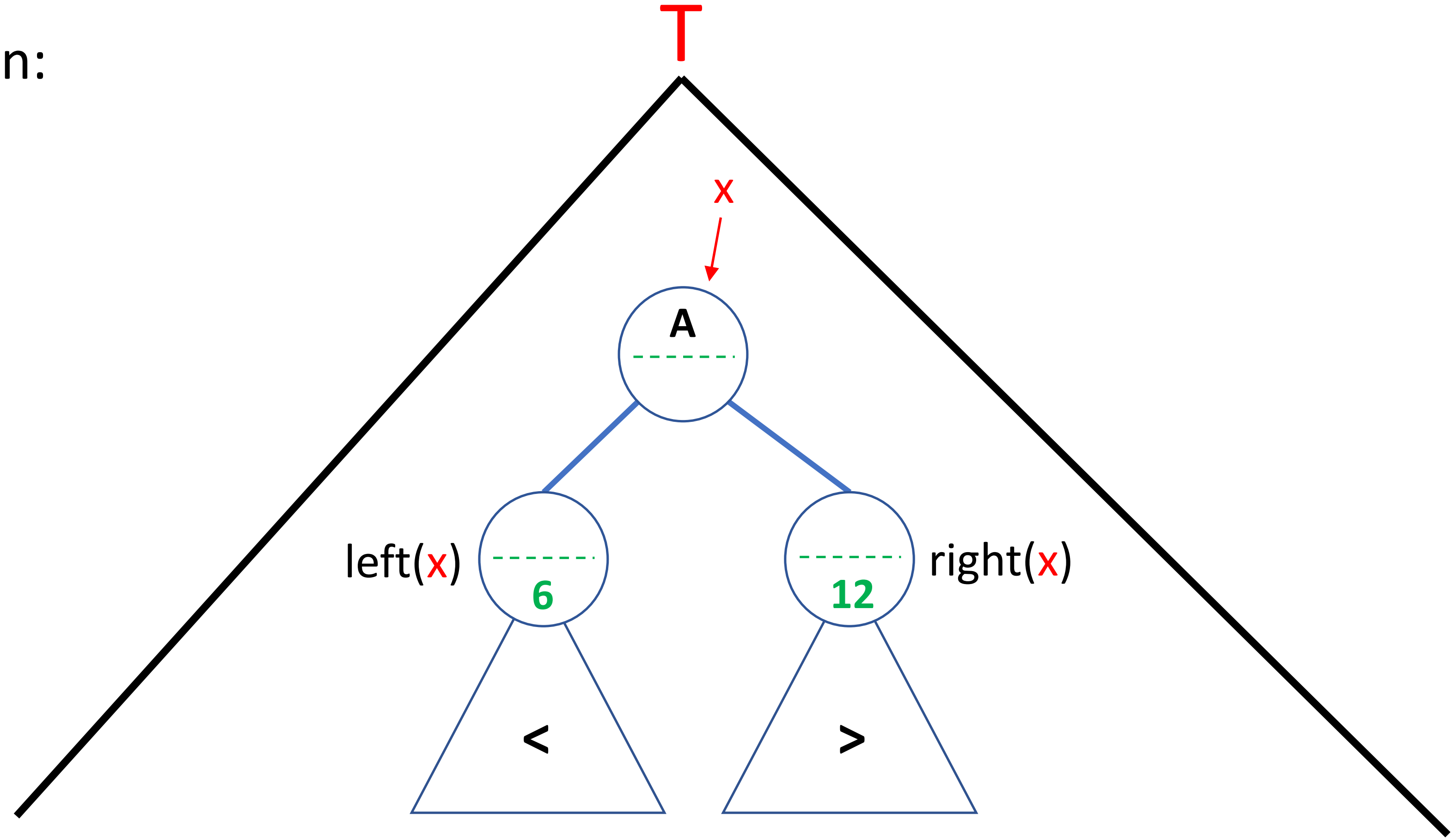
For every node x,

$$\boxed{\mathbf{size}(x) = \mathbf{size}(\mathbf{left}(x)) + \mathbf{size}(\mathbf{right}(x)) + 1}$$

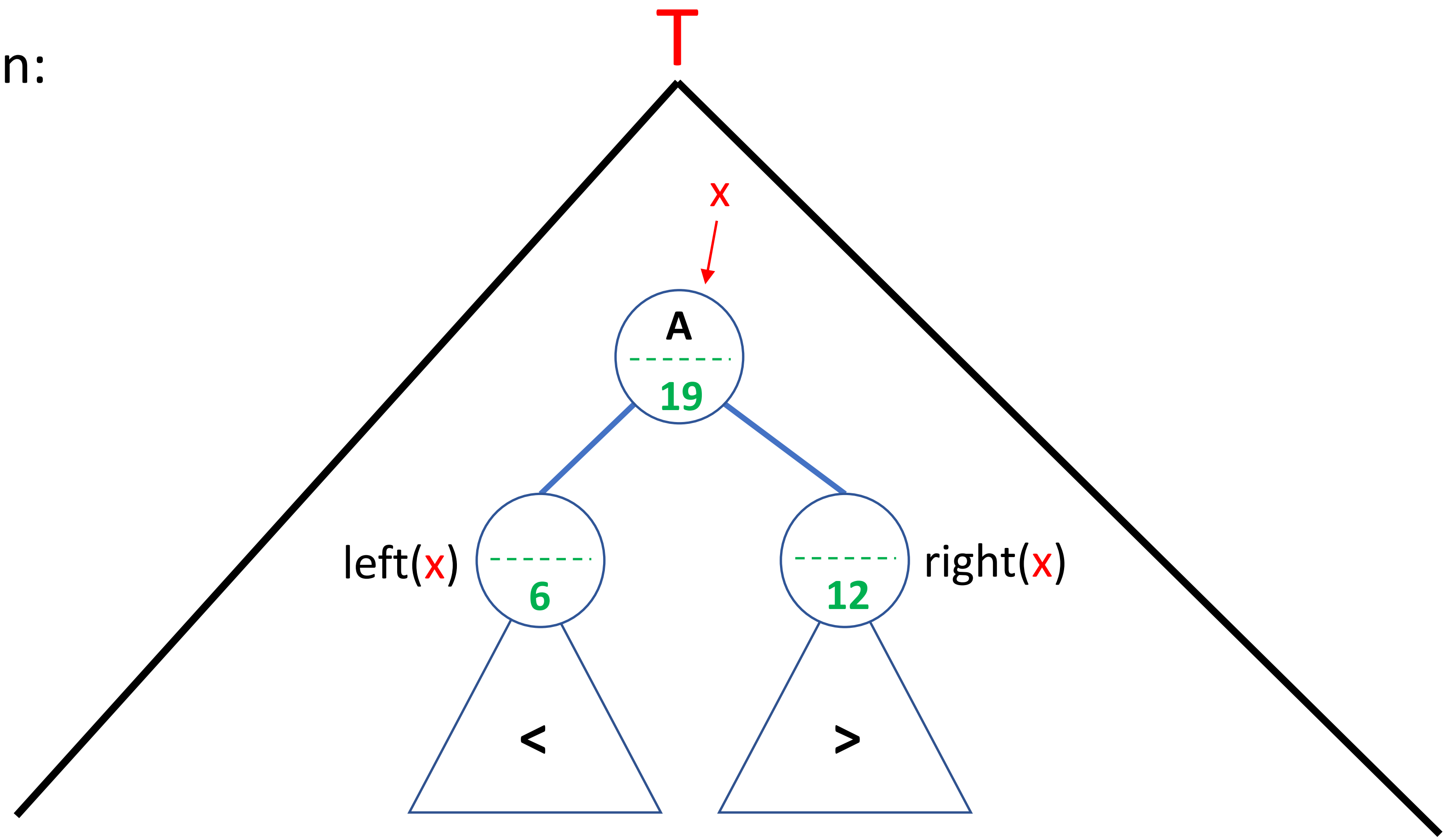


1. How to efficiently implement **Select** and **Rank ?**
2. How to efficiently maintain the size field during **Insert** and **Delete**?

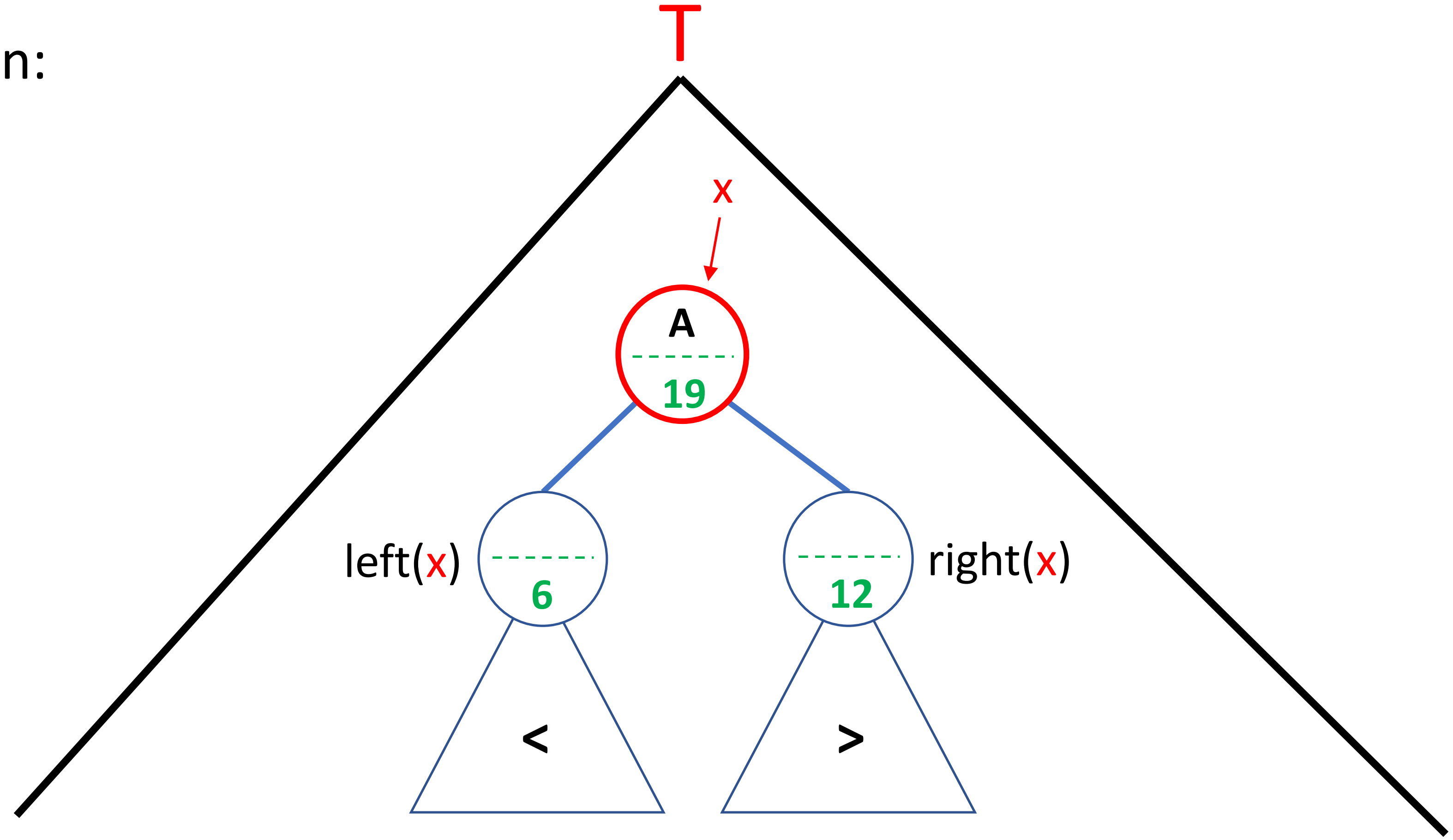
Basic Observation:



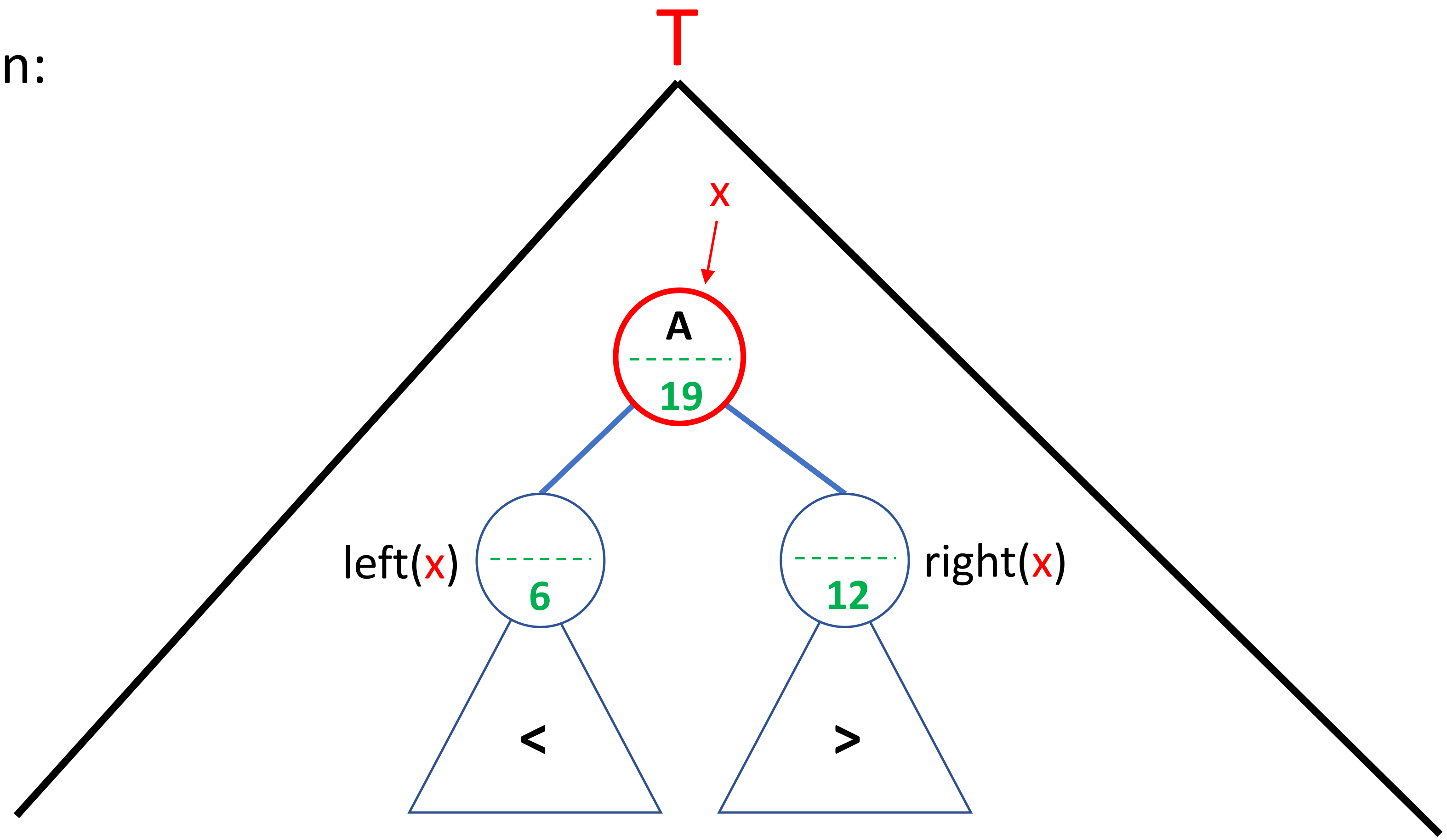
Basic Observation:



Basic Observation:

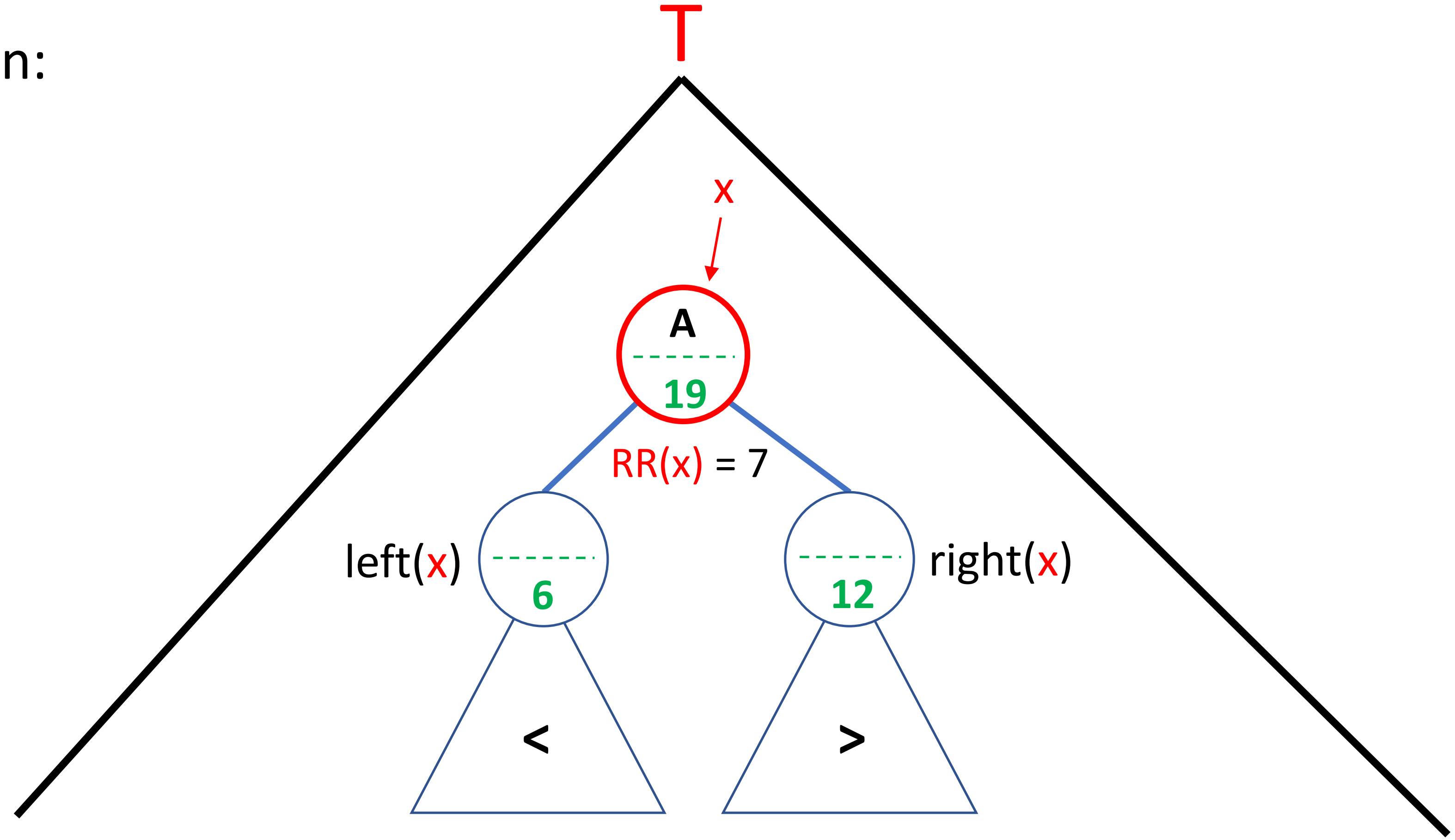


Basic Observation:



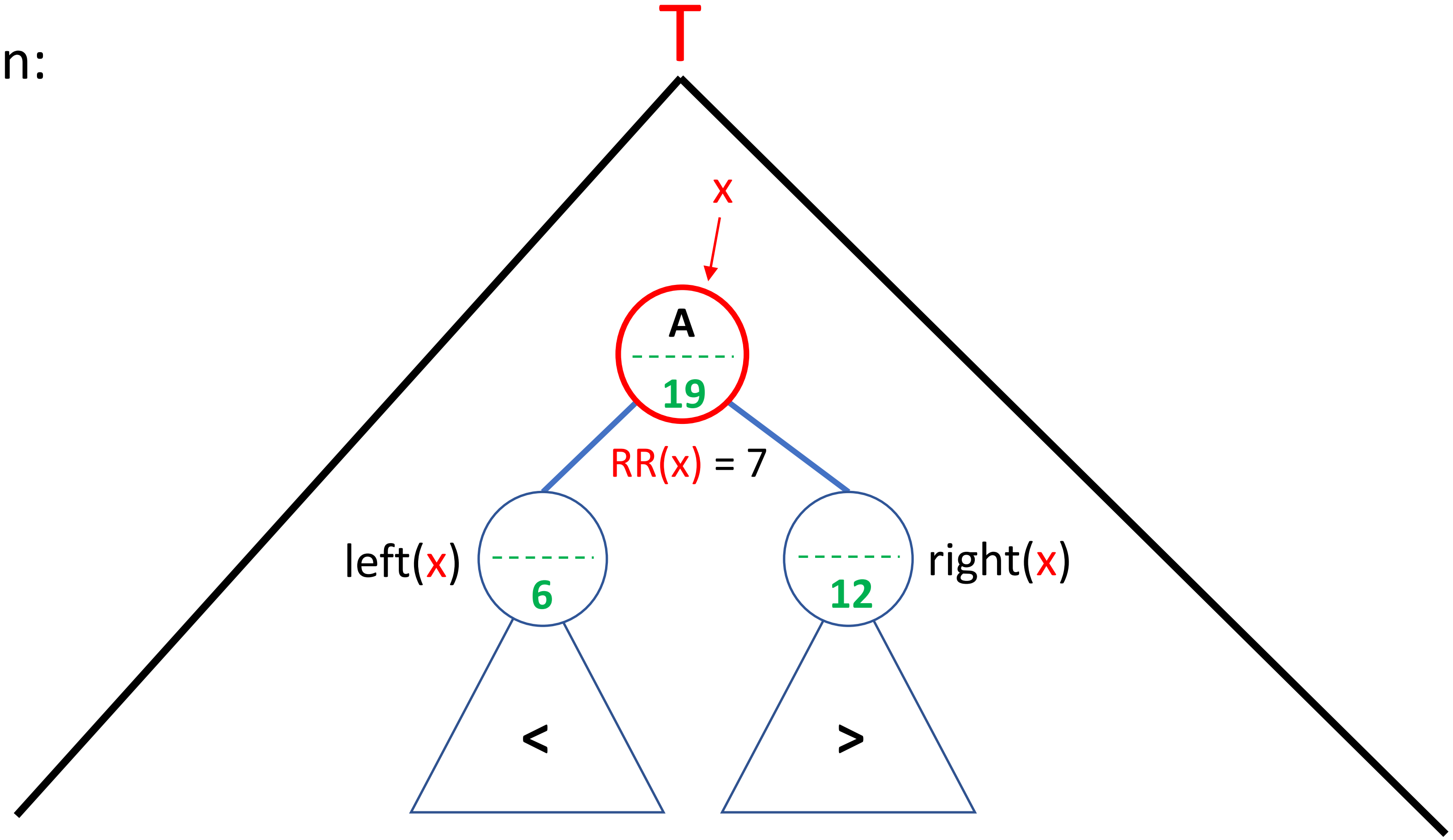
RR(x): Relative Rank of x in the subtree rooted at x

Basic Observation:



RR(x): Relative Rank of x in the subtree rooted at x

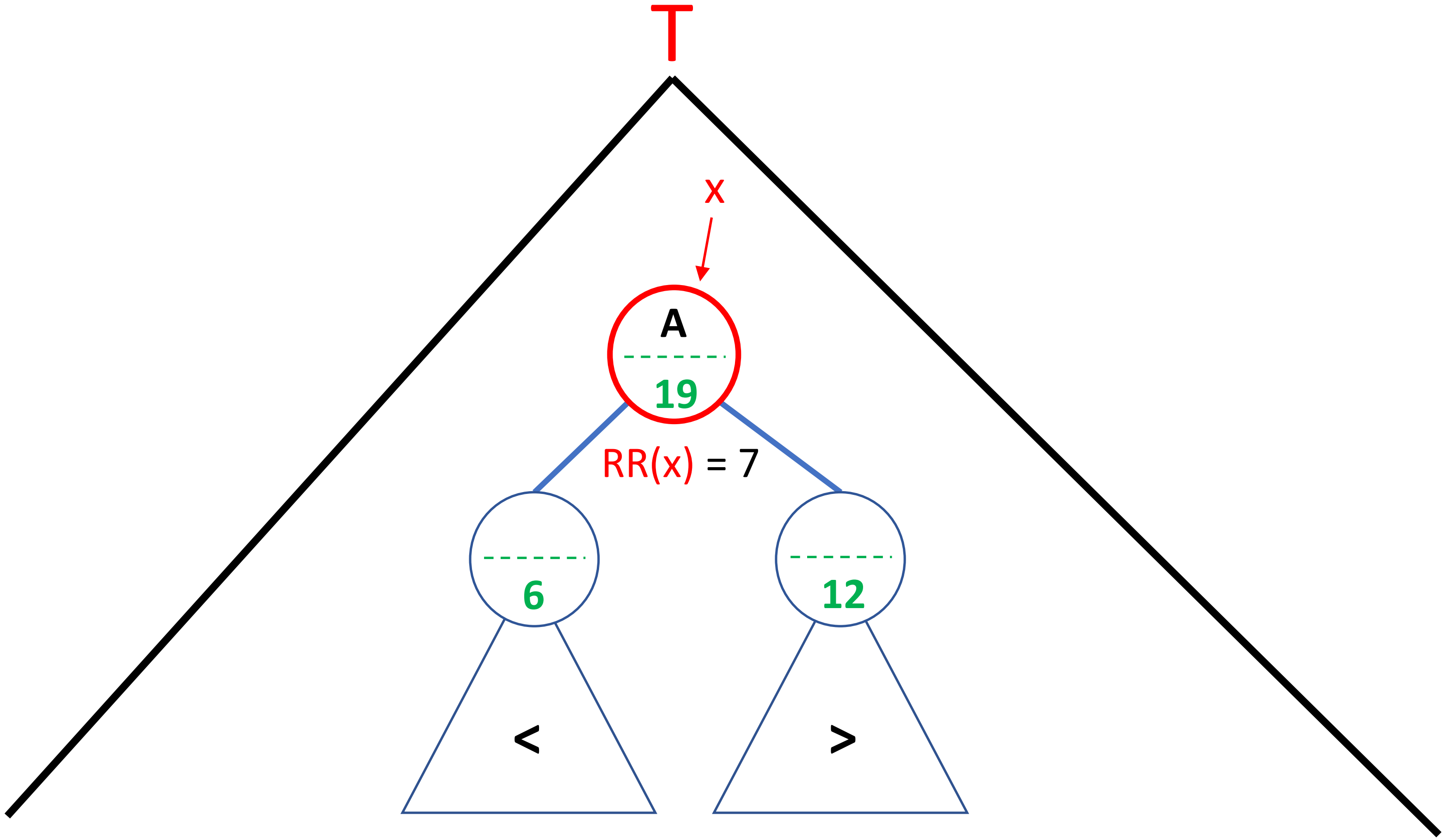
Basic Observation:



RR(x): Relative Rank of x in the subtree rooted at x

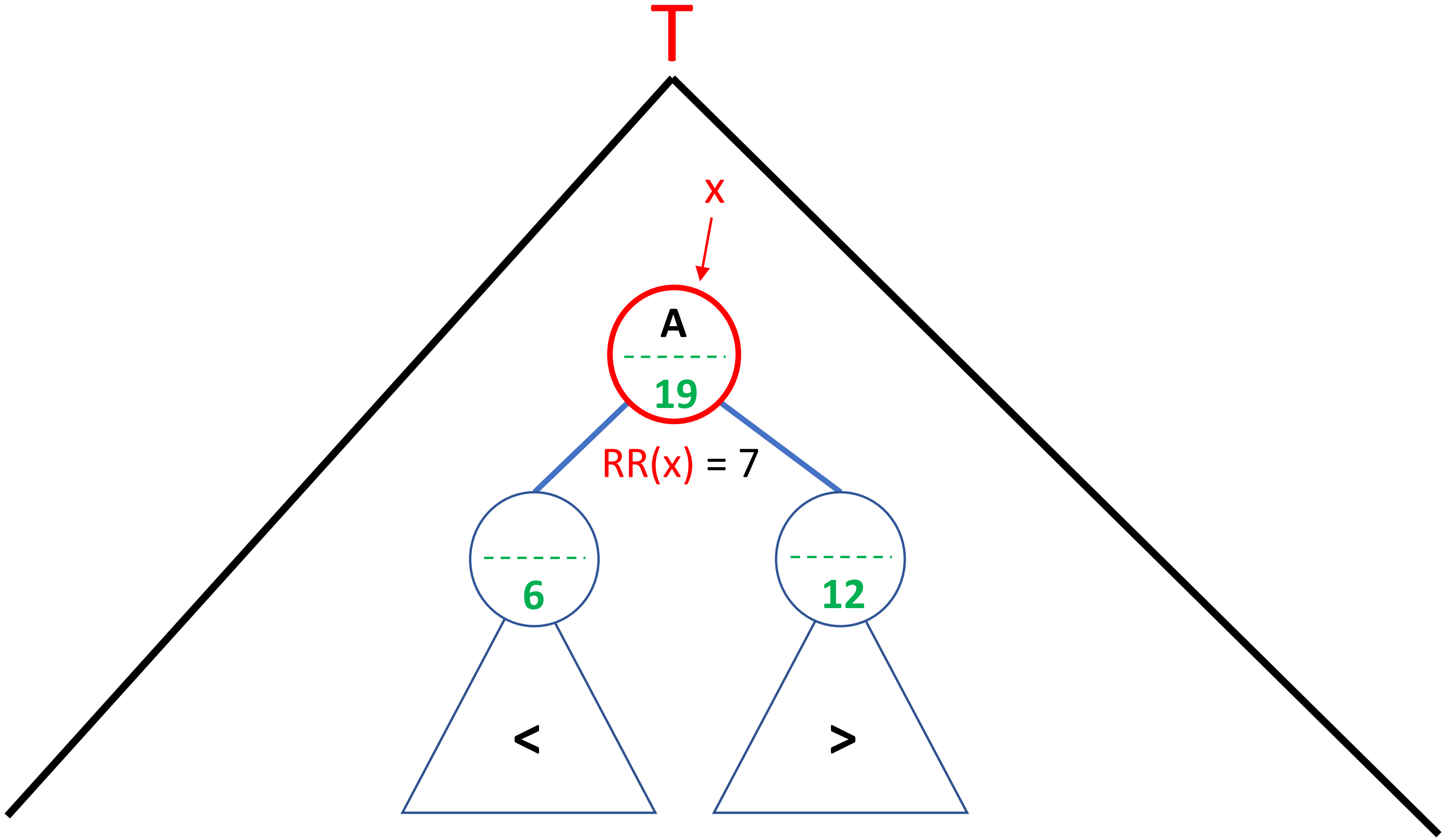
RR(x) = size(left(x)) + 1

RR(x) = 7



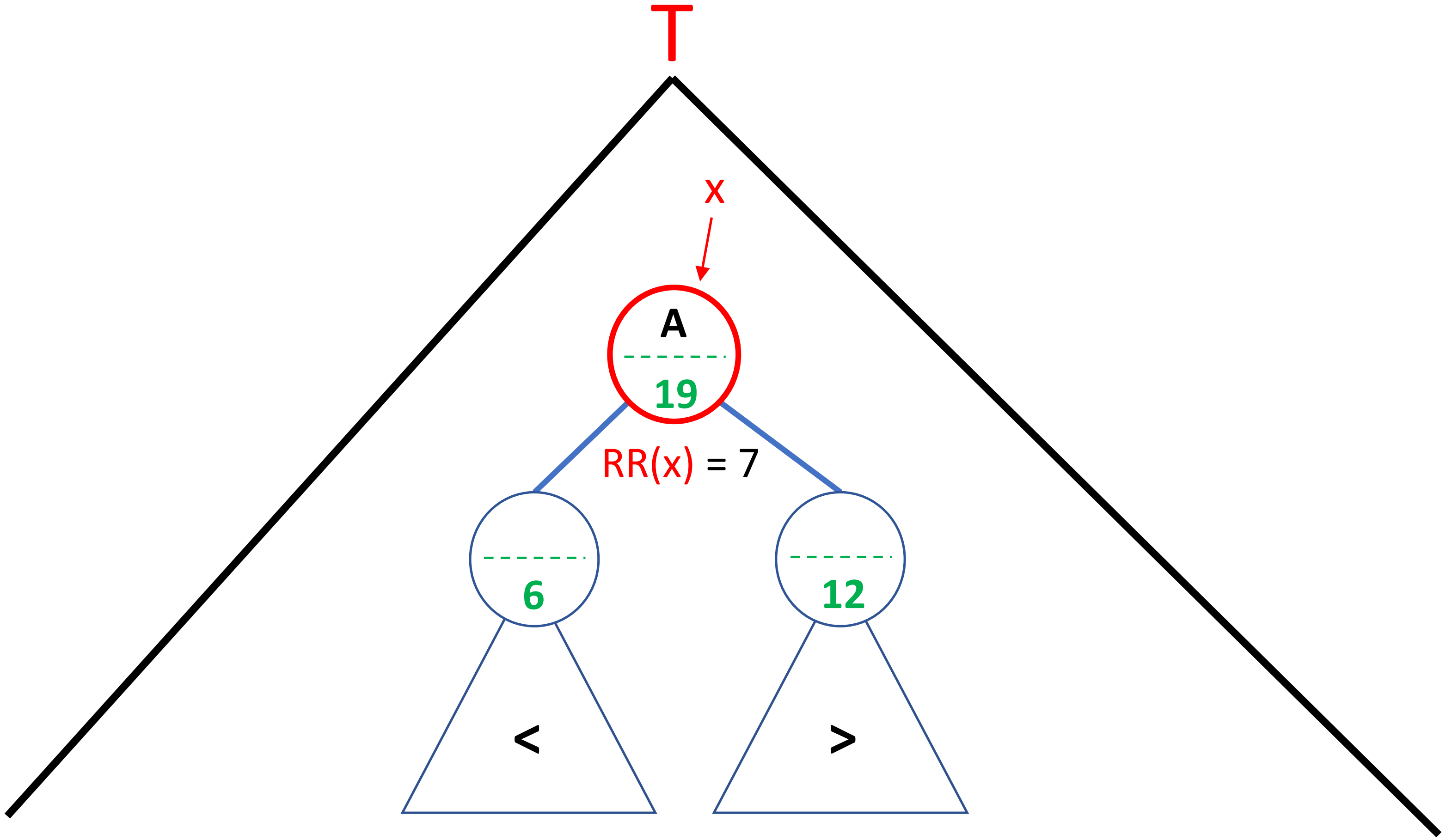
$RR(x) = 7$

Select(x, 7)



RR(x) = 7

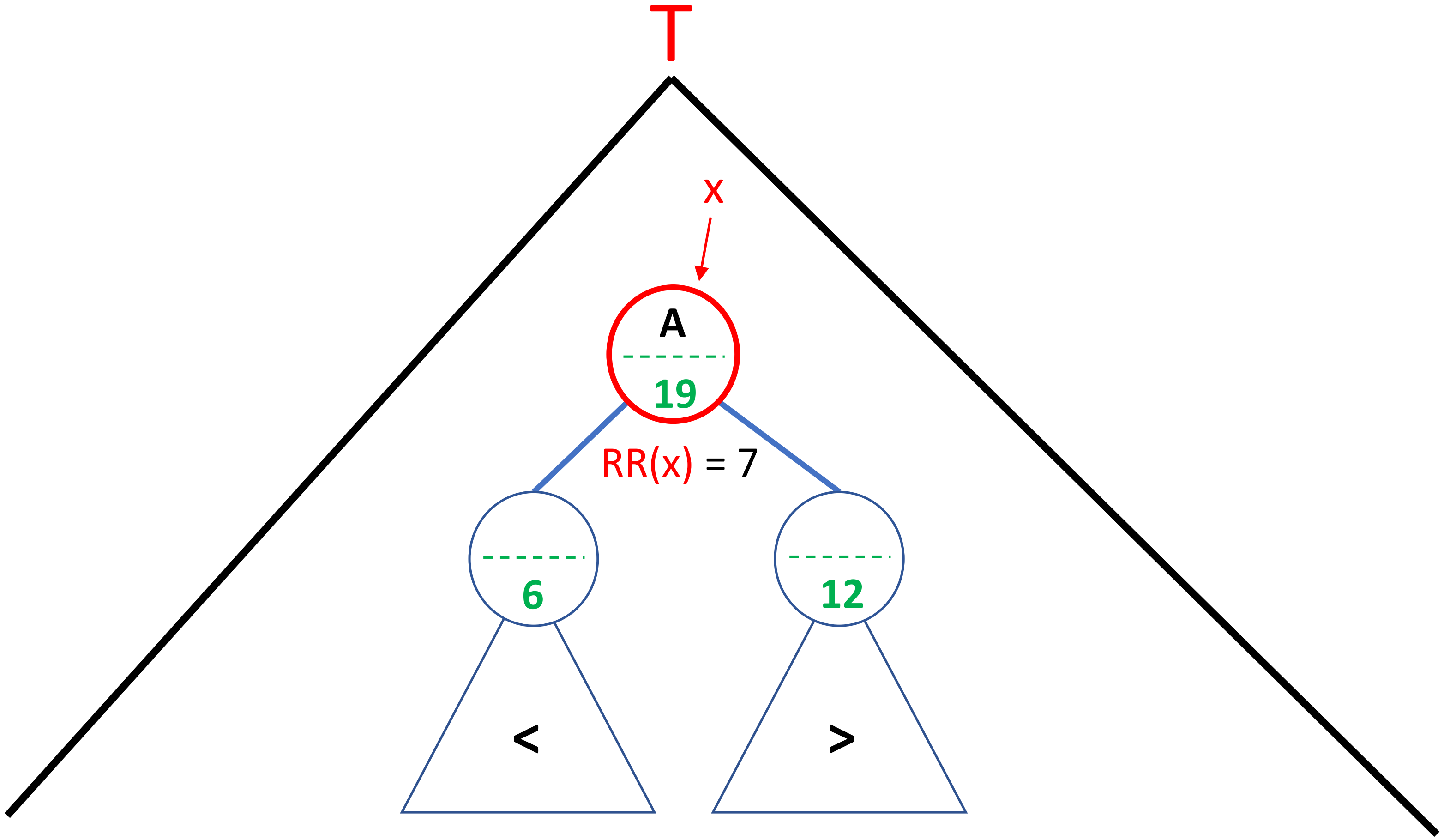
Select(x, 7) : Returns x



RR(x) = 7

Select(x, 7) : Returns x

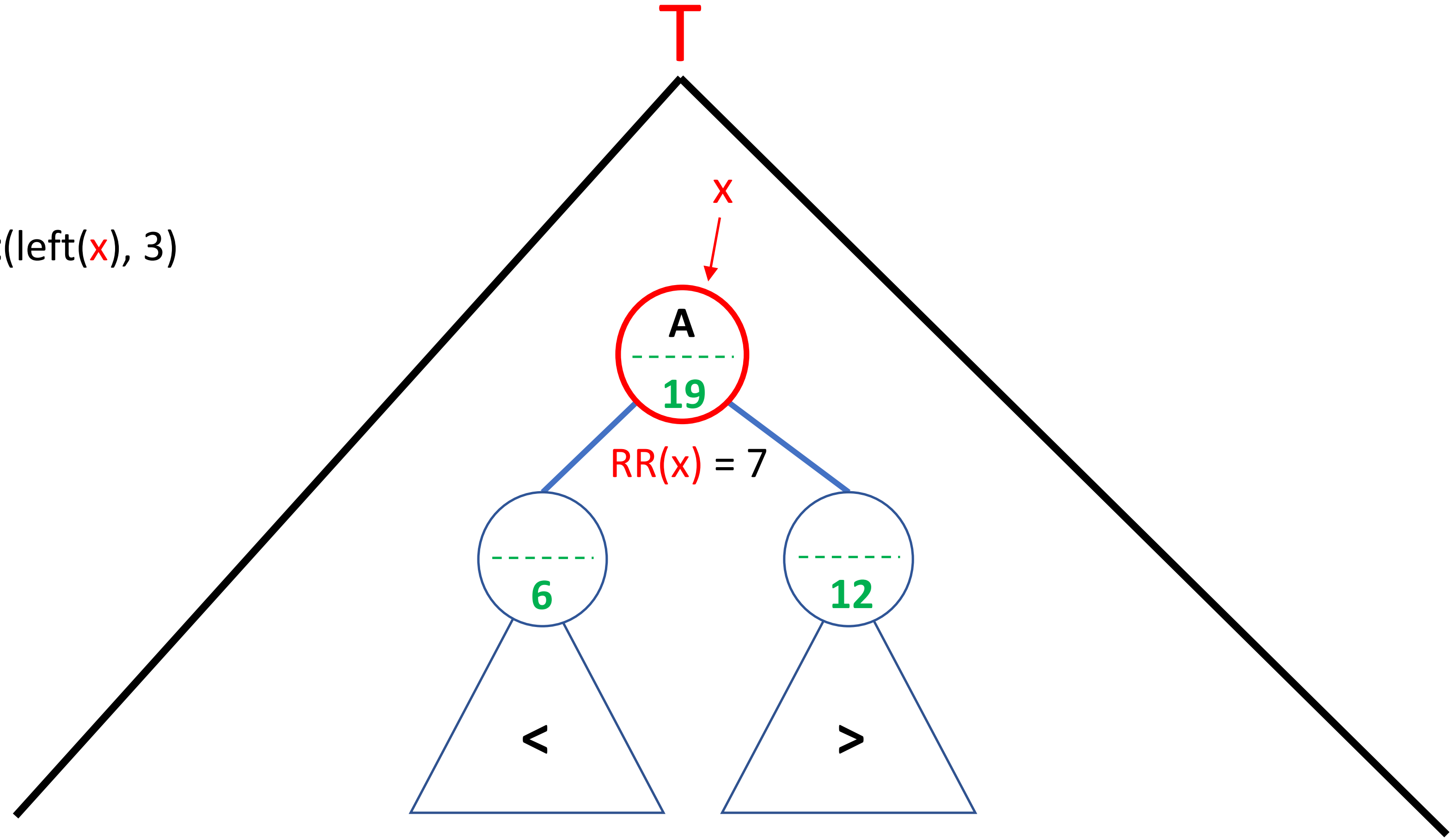
Select(x, 3)



RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

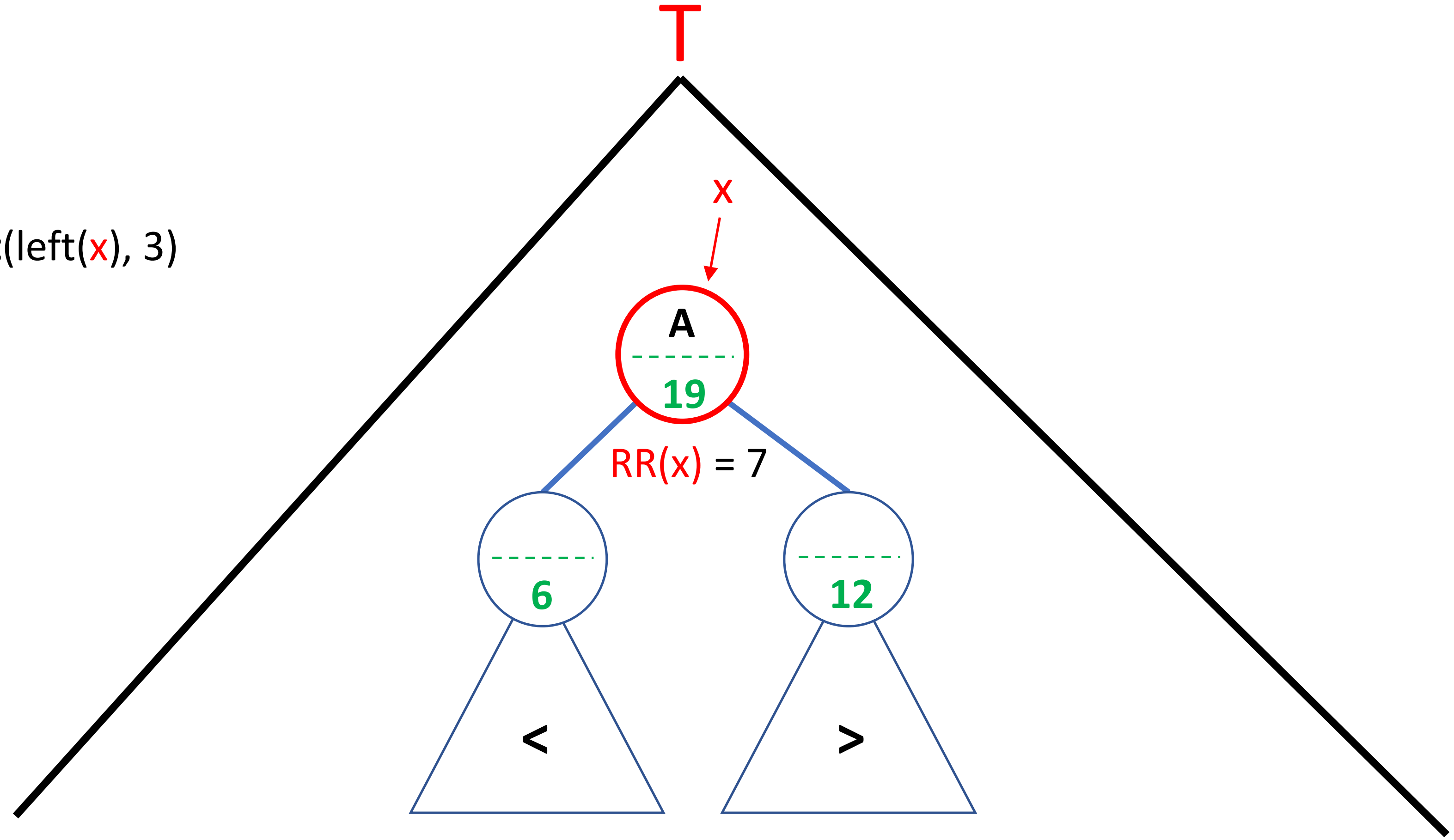


RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

Select(x, 8)

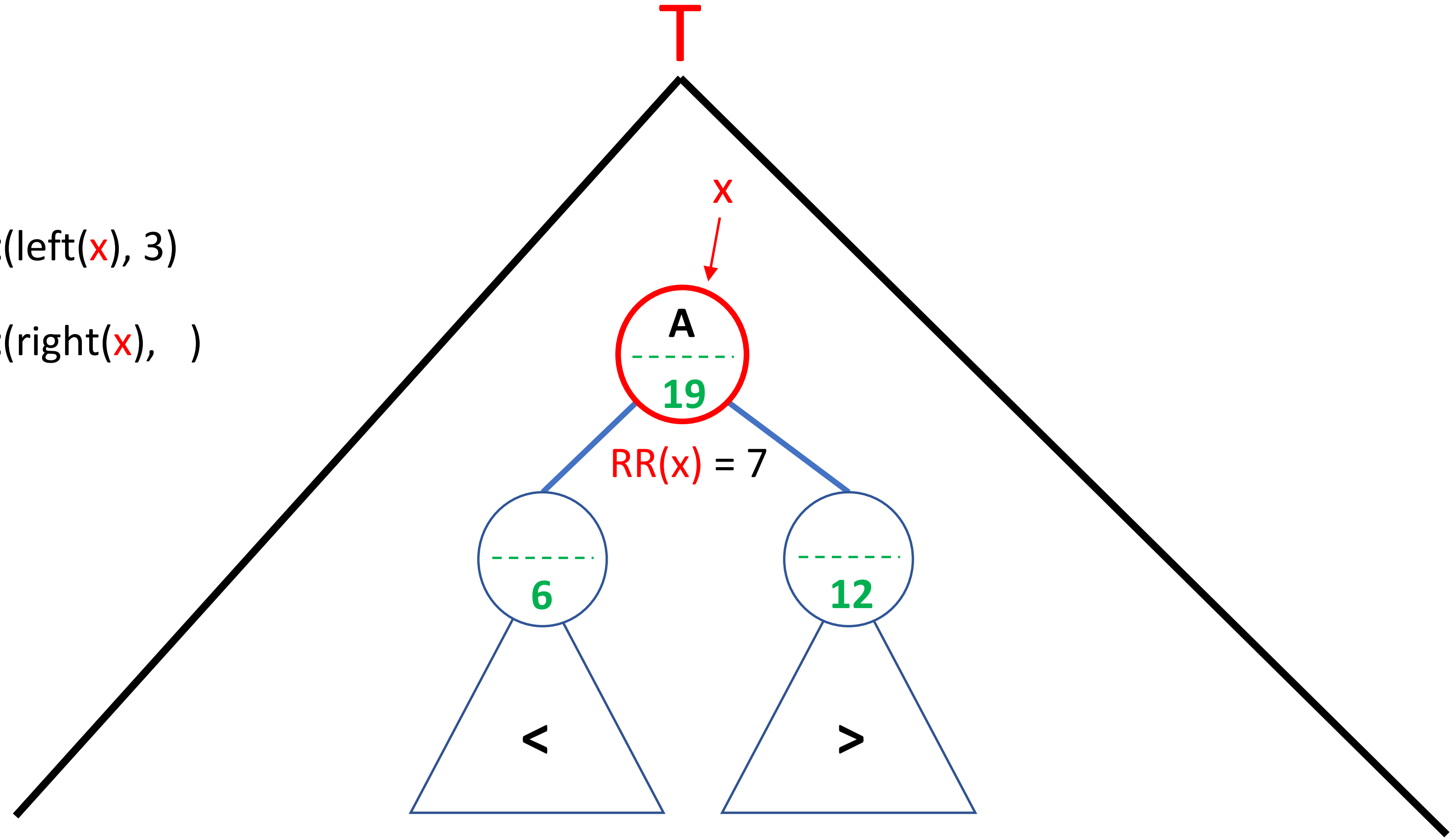


RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

Select(x, 8) : Calls **Select**(right(x),   )

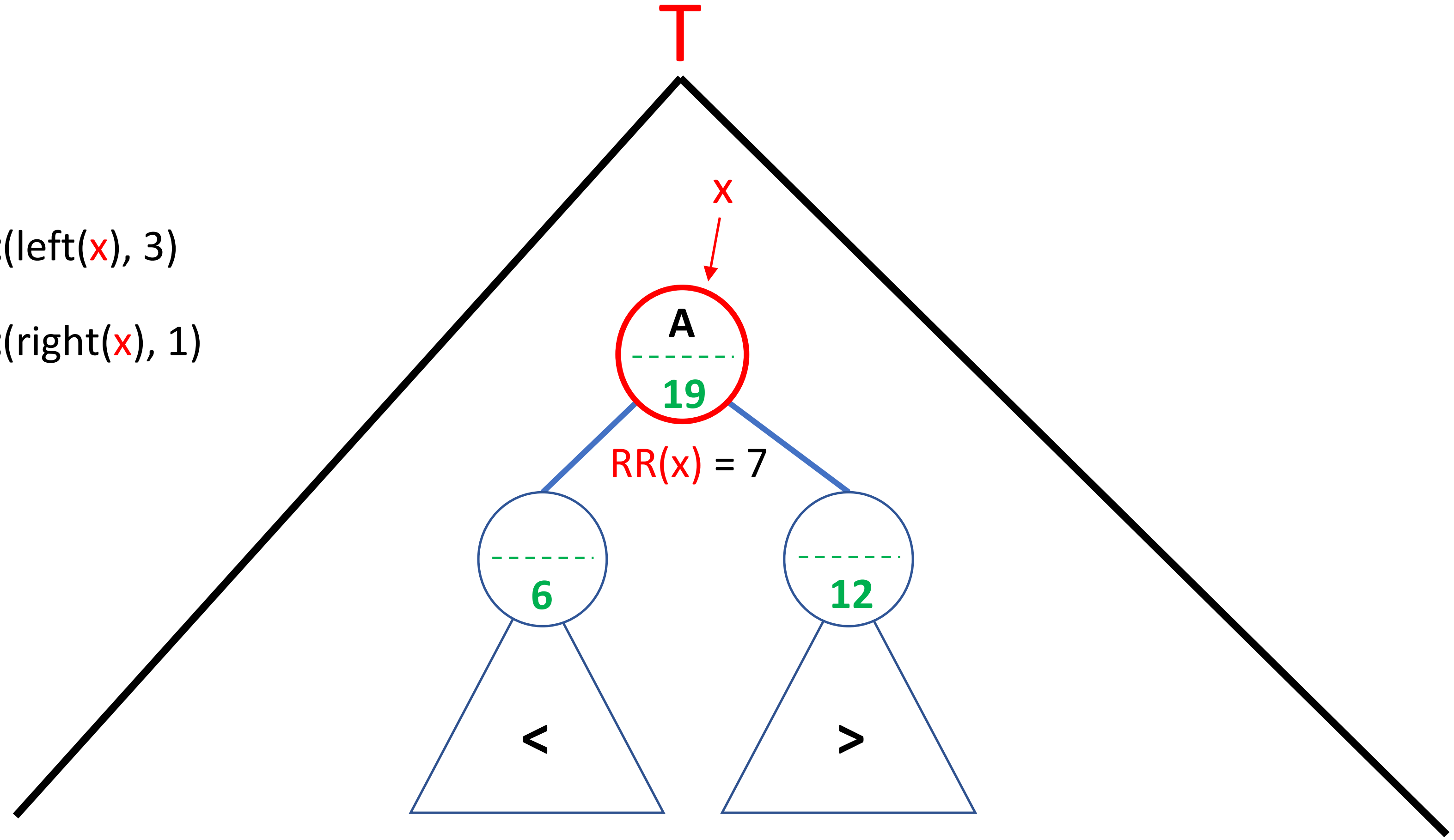


RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

Select(x, 8) : Calls **Select**(right(x), 1)



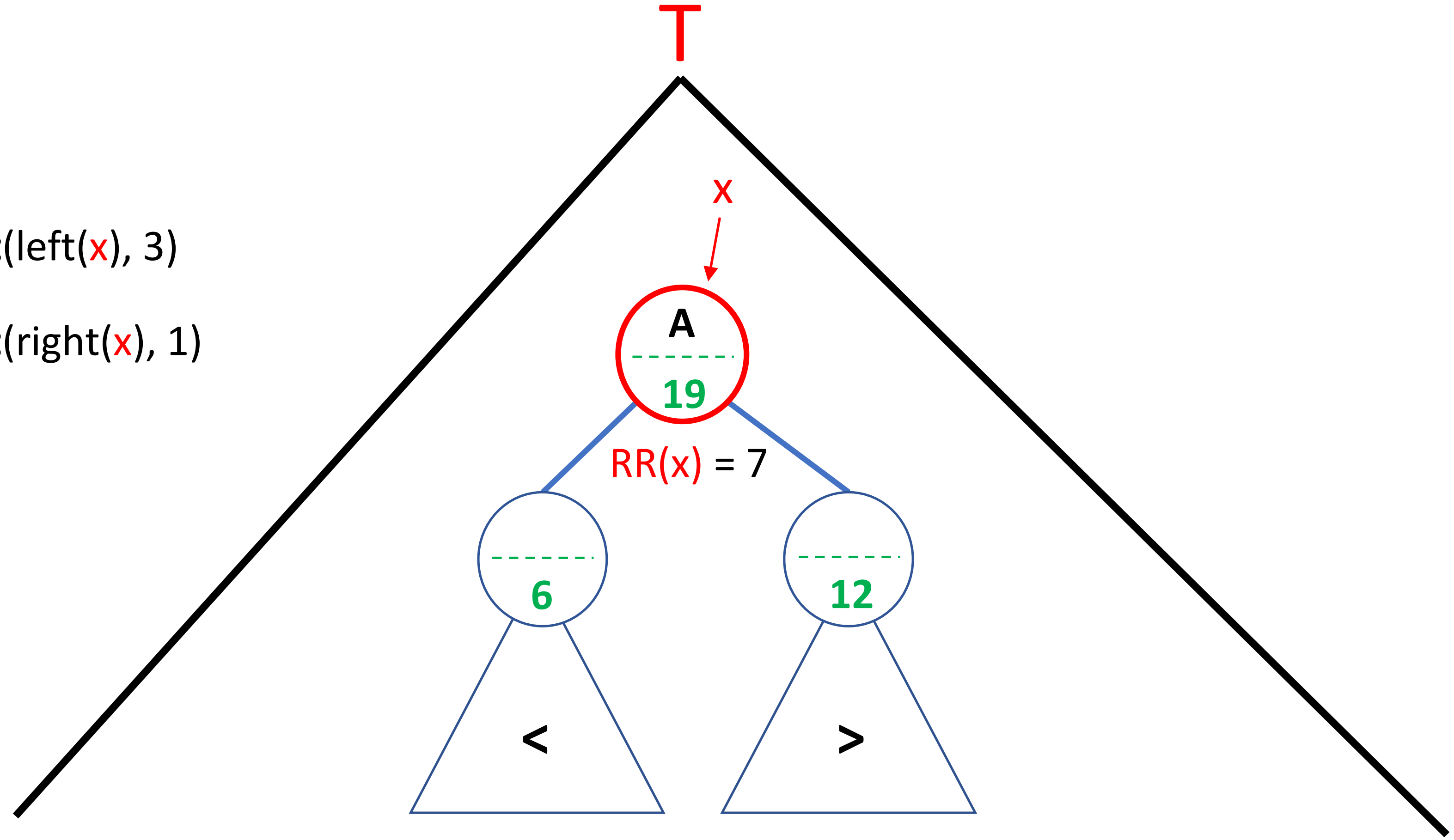
RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

Select(x, 8) : Calls **Select**(right(x), 1)

Select(x, 11)



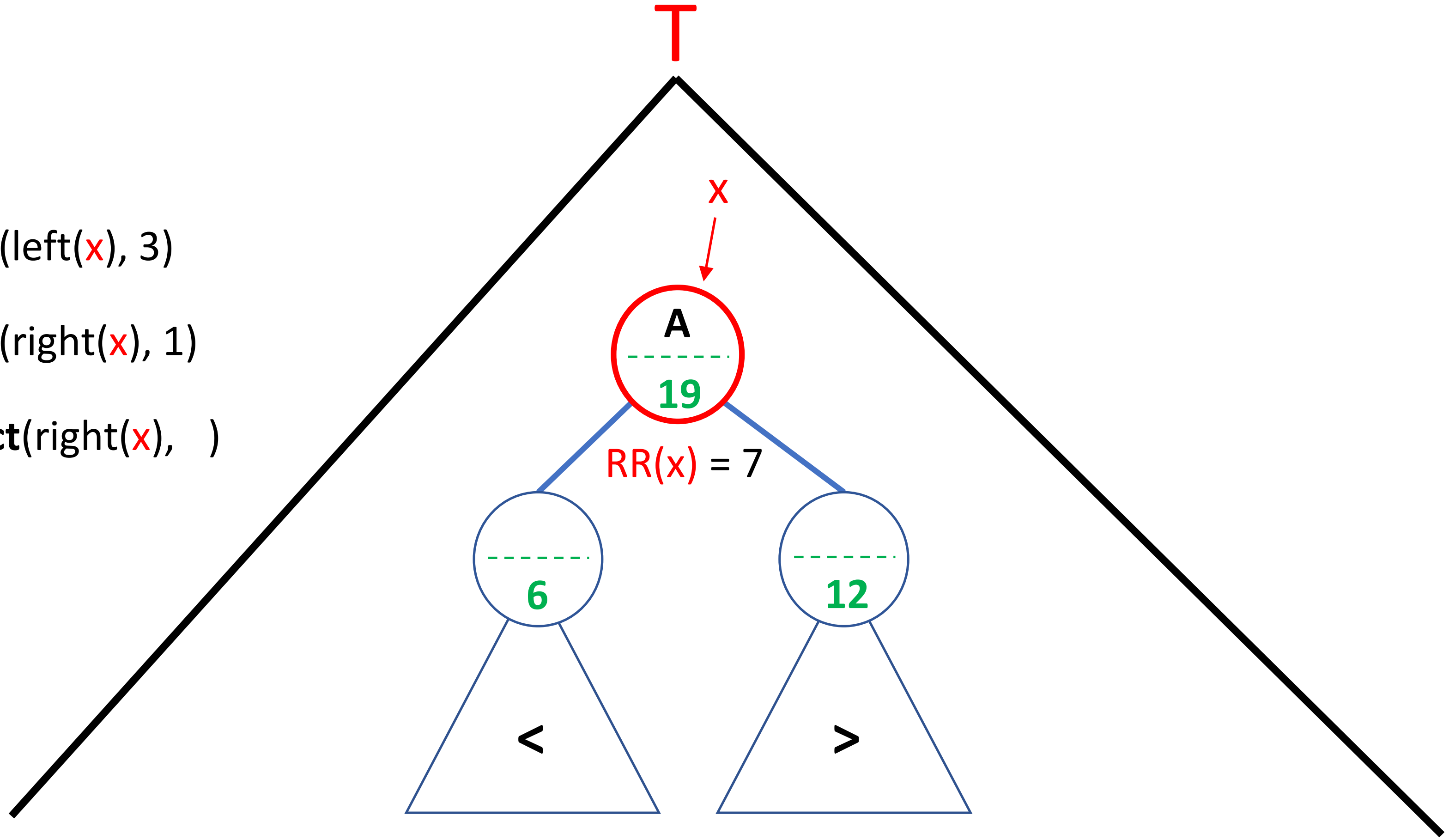
RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

Select(x, 8) : Calls **Select**(right(x), 1)

Select(x, 11) : Calls **Select**(right(x),   )



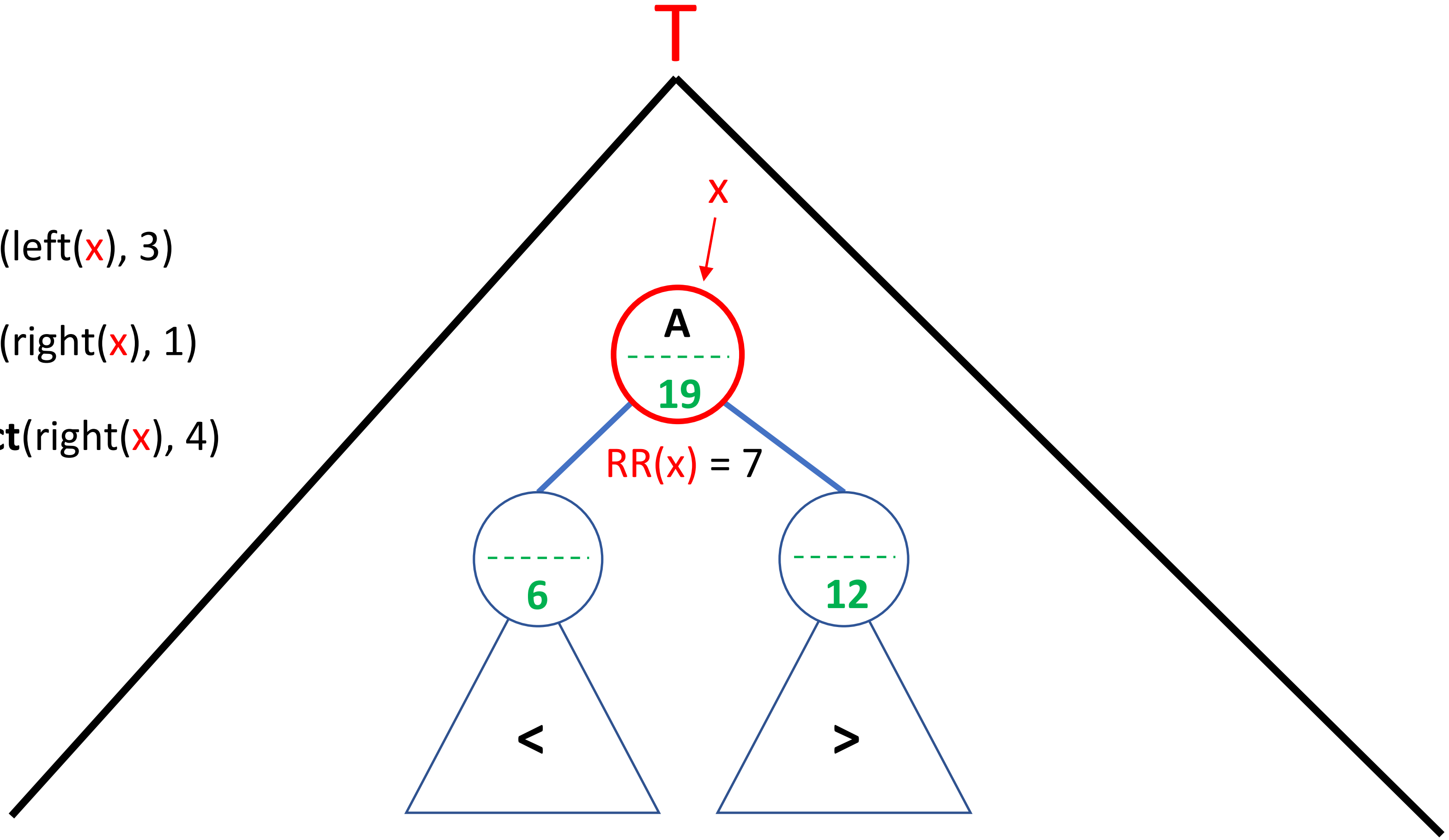
RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

Select(x, 8) : Calls **Select**(right(x), 1)

Select(x, 11) : Calls **Select**(right(x), 4)



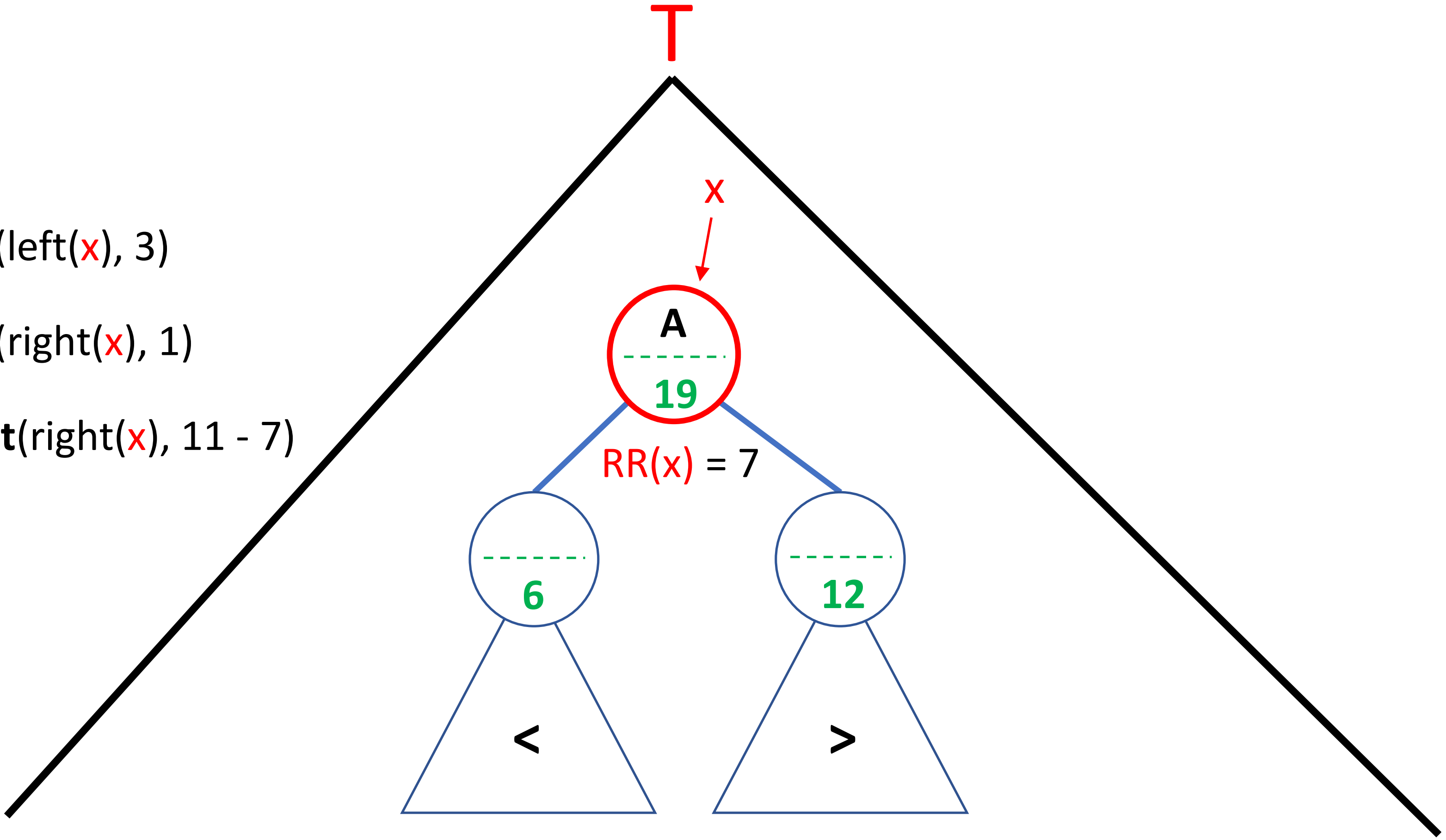
RR(x) = 7

Select(x, 7) : Returns x

Select(x, 3) : Calls **Select**(left(x), 3)

Select(x, 8) : Calls **Select**(right(x), 1)

Select(x, 11) : Calls **Select**(right(x), 11 - 7)



Select(x, k) : Return element with rank k in subtree rooted at x

Select(x, k) : Return element with rank k in subtree rooted at x

RR(x) $\leftarrow$ size(left(x)) + 1

Select(x, k) : Return element with rank k in subtree rooted at x

$RR(x) \leftarrow \text{size}(\text{left}(x)) + 1$

if $k = RR(x)$

if $k < RR(x)$

if $k > RR(x)$

Select(x, k) : Return element with rank k in subtree rooted at x

$RR(x) \leftarrow \text{size}(\text{left}(x)) + 1$

if $k = RR(x)$ then return x

if $k < RR(x)$

if $k > RR(x)$

Select(x, k) : Return element with rank k in subtree rooted at x

RR(x) ← size(left(x)) + 1

if k = RR(x) then return x

if k < RR(x) then **Select**(left(x),)

if k > RR(x)

Select(x, k) : Return element with rank k in subtree rooted at x

$RR(x) \leftarrow \text{size}(\text{left}(x)) + 1$

if $k = RR(x)$ then return x

if $k < RR(x)$ then **Select**(left(x), k)

if $k > RR(x)$

Select(x, k) : Return element with rank k in subtree rooted at x

RR(x) ← size(left(x)) + 1

if k = RR(x) then return x

if k < RR(x) then **Select**(left(x), k)

if k > RR(x) then **Select**(right(x),)

Select(x, k) : Return element with rank k in subtree rooted at x

RR(x) ← size(left(x)) + 1

if k = RR(x) then return x

if k < RR(x) then **Select**(left(x), k)

if k > RR(x) then **Select**(right(x), k – RR(x))

Select(x, k) : Return element with rank k in subtree rooted at x

$RR(x) \leftarrow size(left(x)) + 1$

if $k = RR(x)$ then return x

if $k < RR(x)$ then **Select**(left(x), k)

if $k > RR(x)$ then **Select**(right(x), $k - RR(x)$)

Select(T, k) = **Select**(x, k) where x is the root of T

Select(x, k) : Return element with rank k in subtree rooted at x

$RR(x) \leftarrow$ size(left(x)) + 1

if $k = RR(x)$ then return x

if $k < RR(x)$ then **Select**(left(x), k)

if $k > RR(x)$ then **Select**(right(x), $k - RR(x)$)

Select(T, k) = **Select**(x, k) where x is the root of T

Worst-Case Time Complexity of **Select**(T, k):

Select(x, k) : Return element with rank k in subtree rooted at x

RR(x) ← size(left(x)) + 1

if k = RR(x) then return x

if k < RR(x) then **Select**(left(x), k)

if k > RR(x) then **Select**(right(x), k – RR(x))

Select(T, k) = **Select**(x, k) where x is the root of T

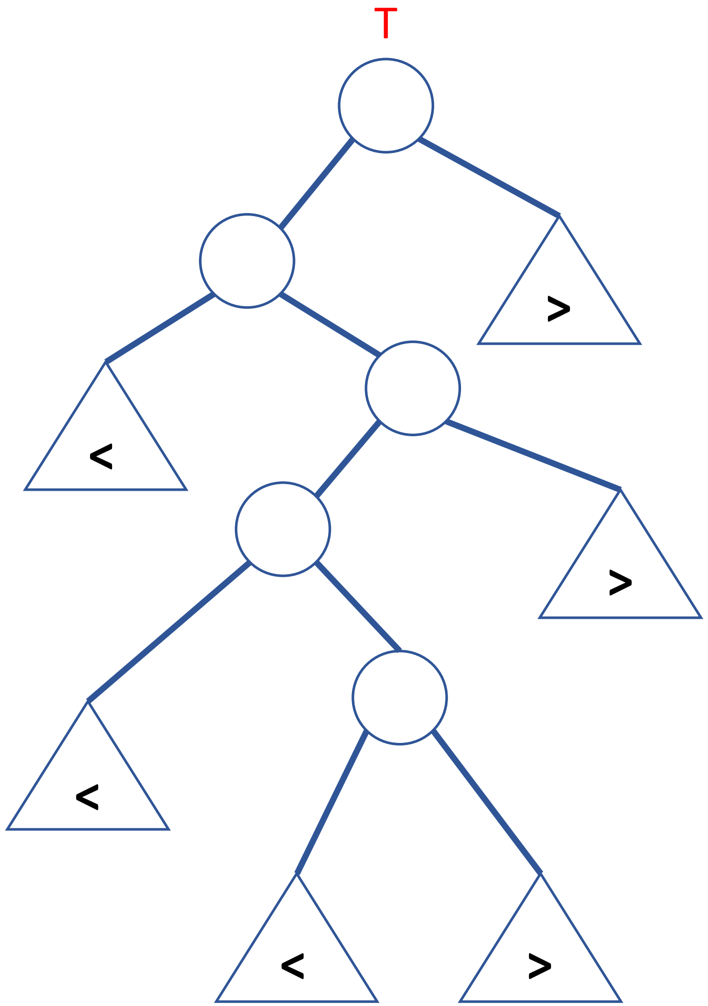
Worst-Case Time Complexity of **Select**(T, k):

- Each **Select** call goes down one level in T (or returns)
- Height(T) is O(log n)
- Hence **Select** takes O(log n)

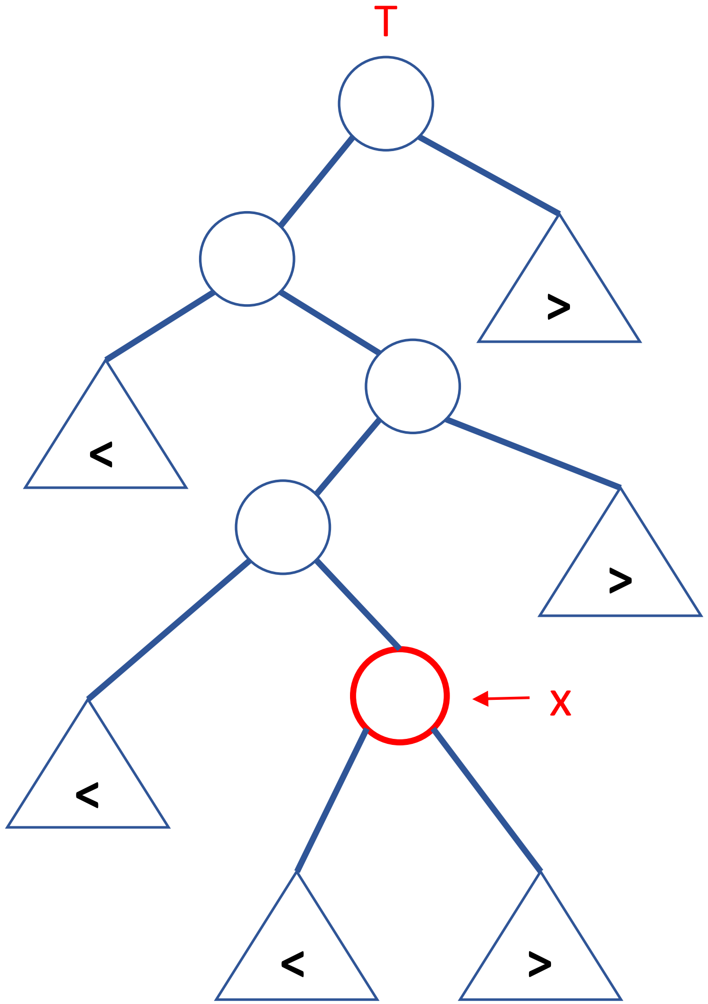
Augmenting AVL

- **Select** operation ✓
- **Rank** operation
- Maintain size() field

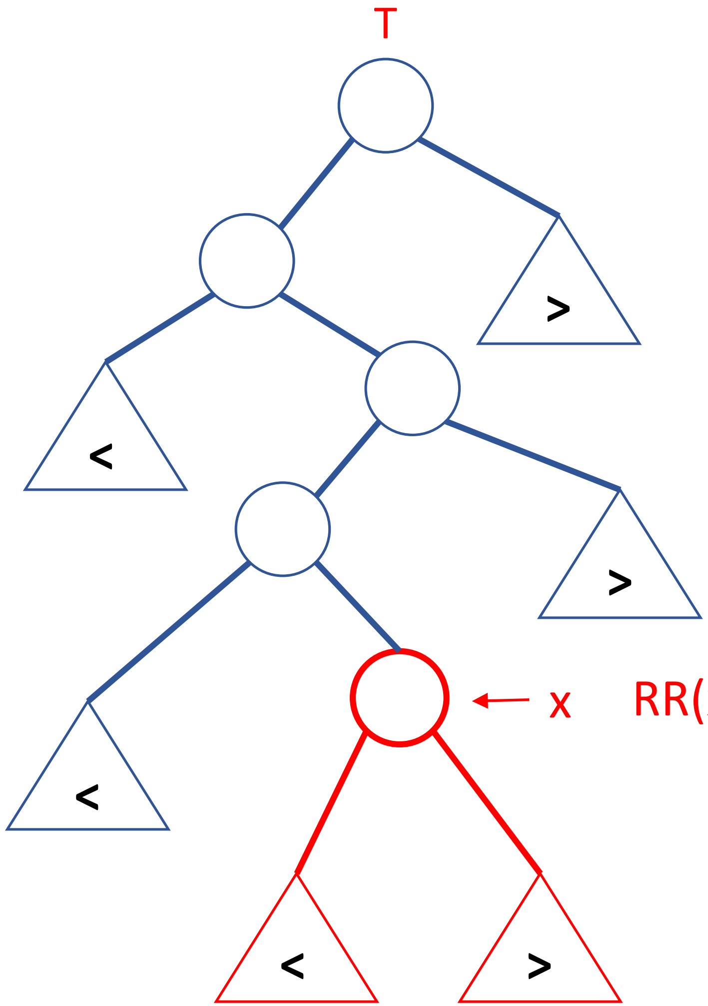
Rank(T, x): return rank of x in T



Rank(T, x): return rank of x in T

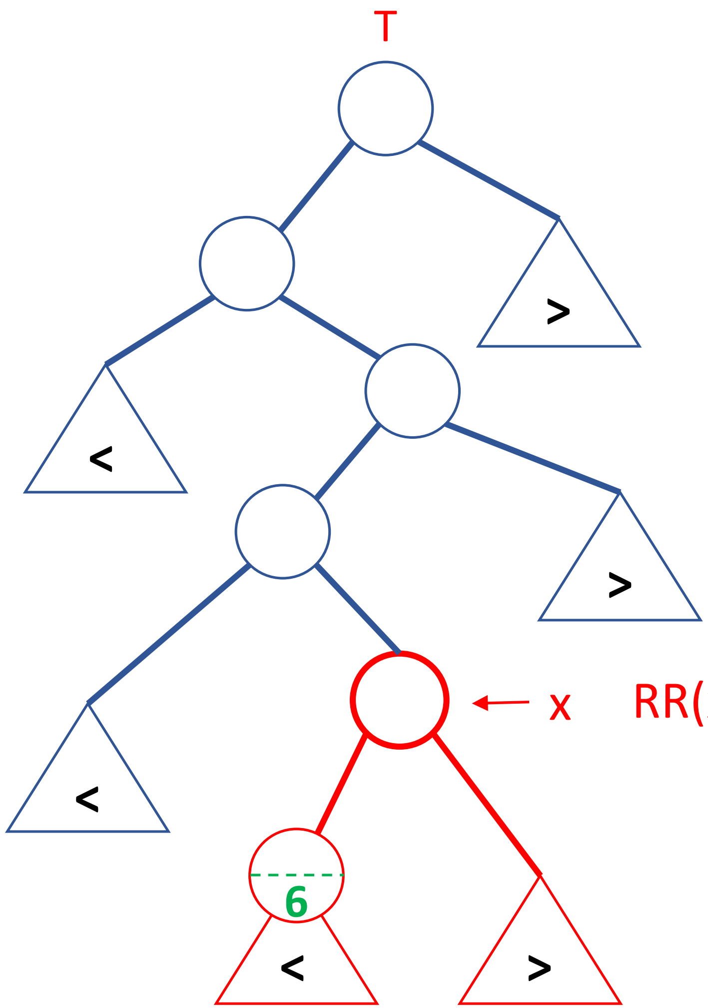


Rank(T, x): return rank of x in T



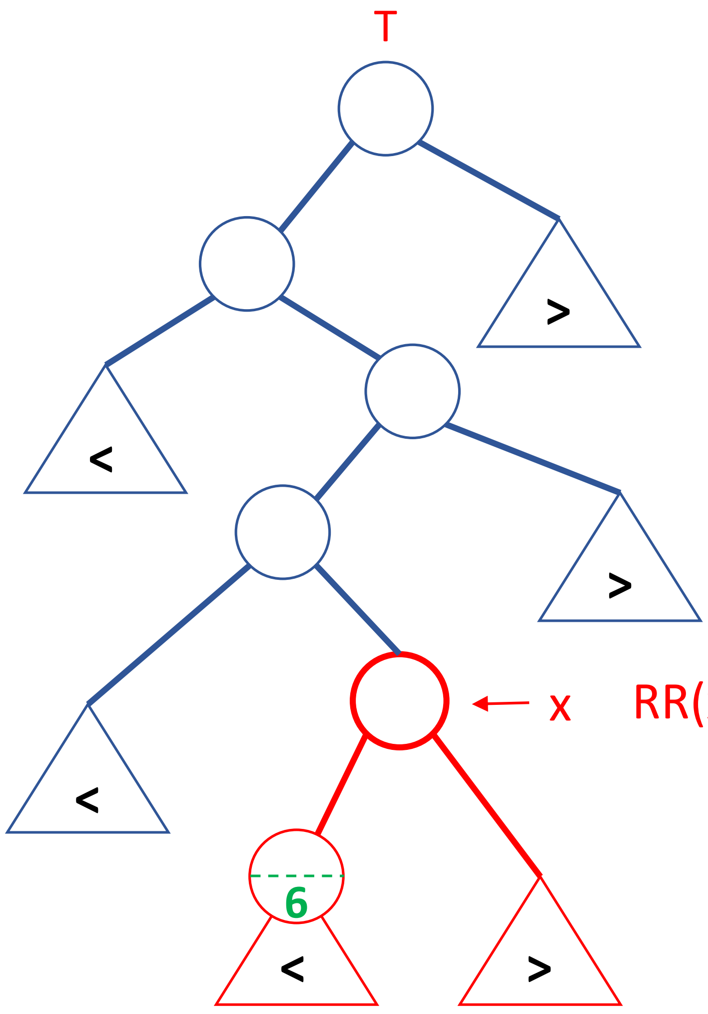
RR(x) in x-subtree =

Rank(T, x): return rank of x in T



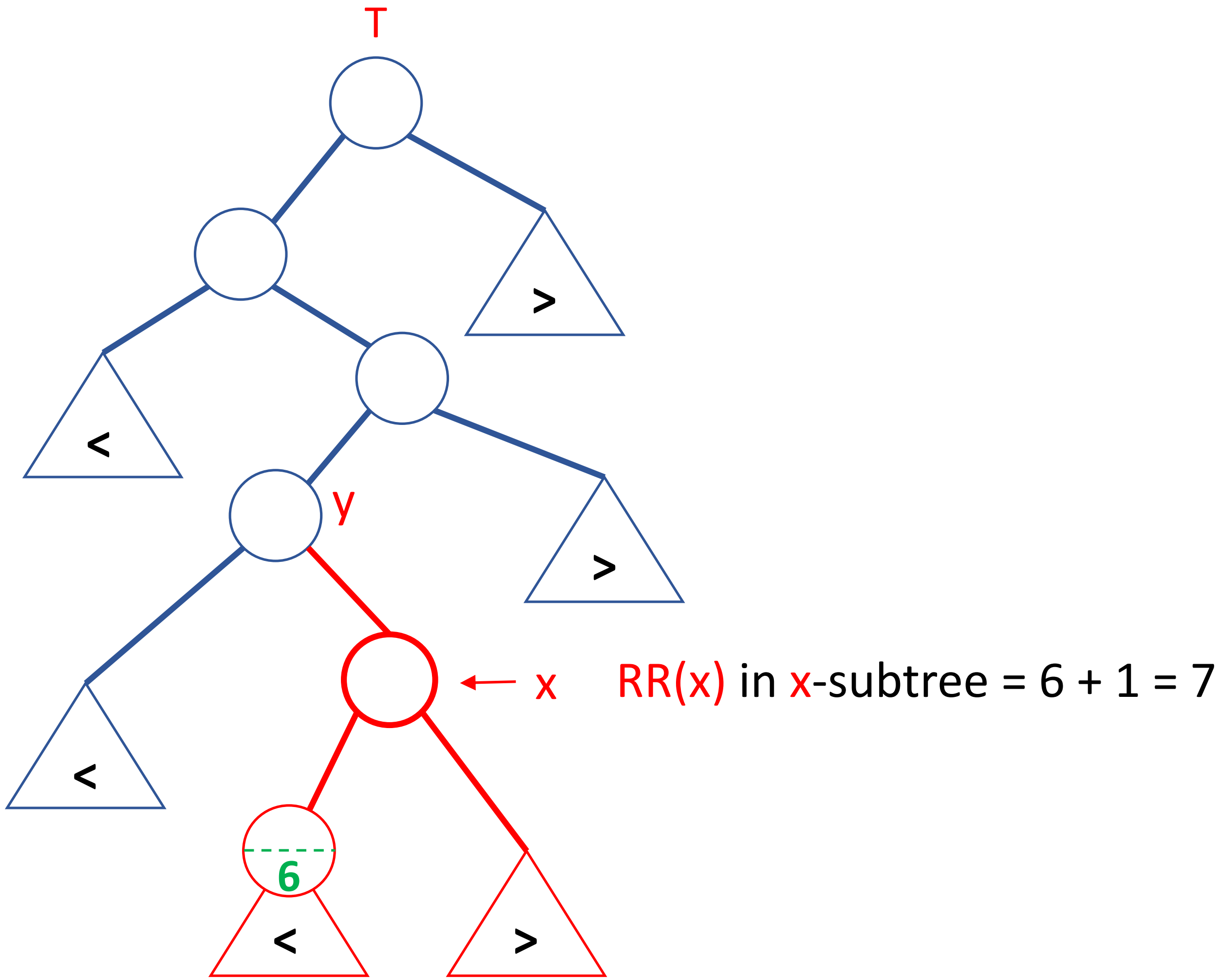
RR(x) in x-subtree =

Rank(T, x): return rank of x in T



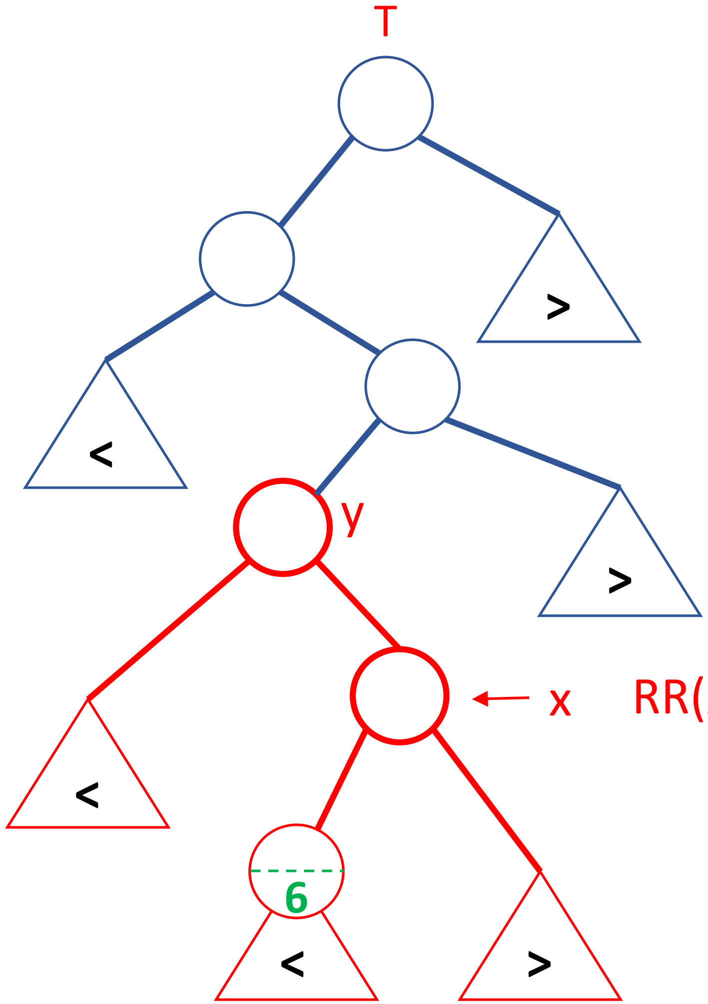
RR(x) in x-subtree = 6 + 1 = 7

Rank(T, x): return rank of x in T



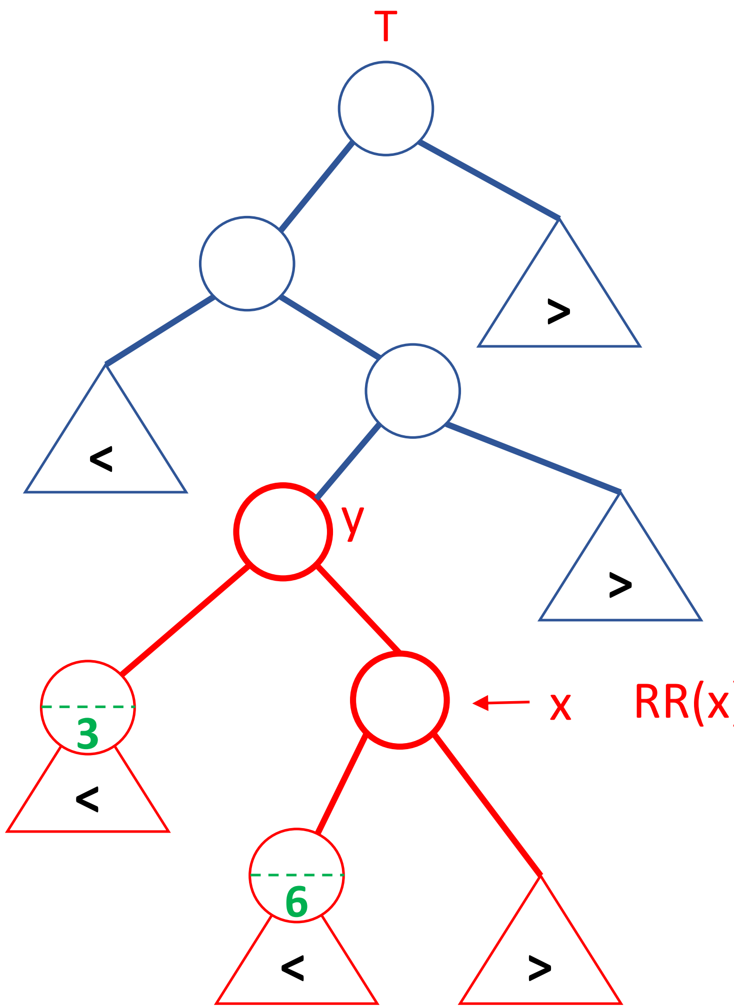
Rank(T, x): return rank of x in T

RR(x) in y-subtree =



RR(x) in x-subtree = 6 + 1 = 7

Rank(T, x): return rank of x in T

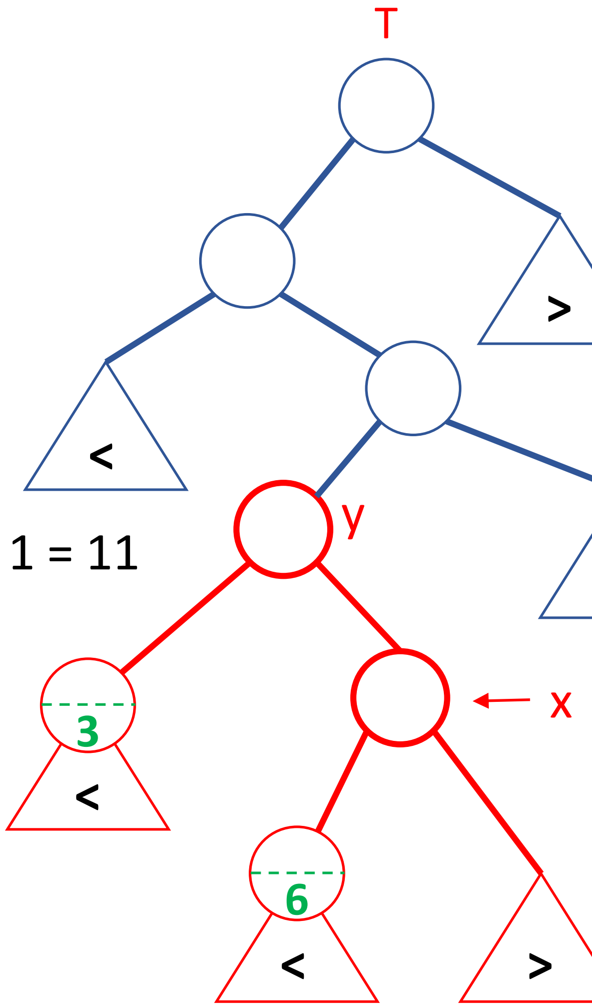


RR(x) in y-subtree =

RR(x) in x-subtree = 6 + 1 = 7

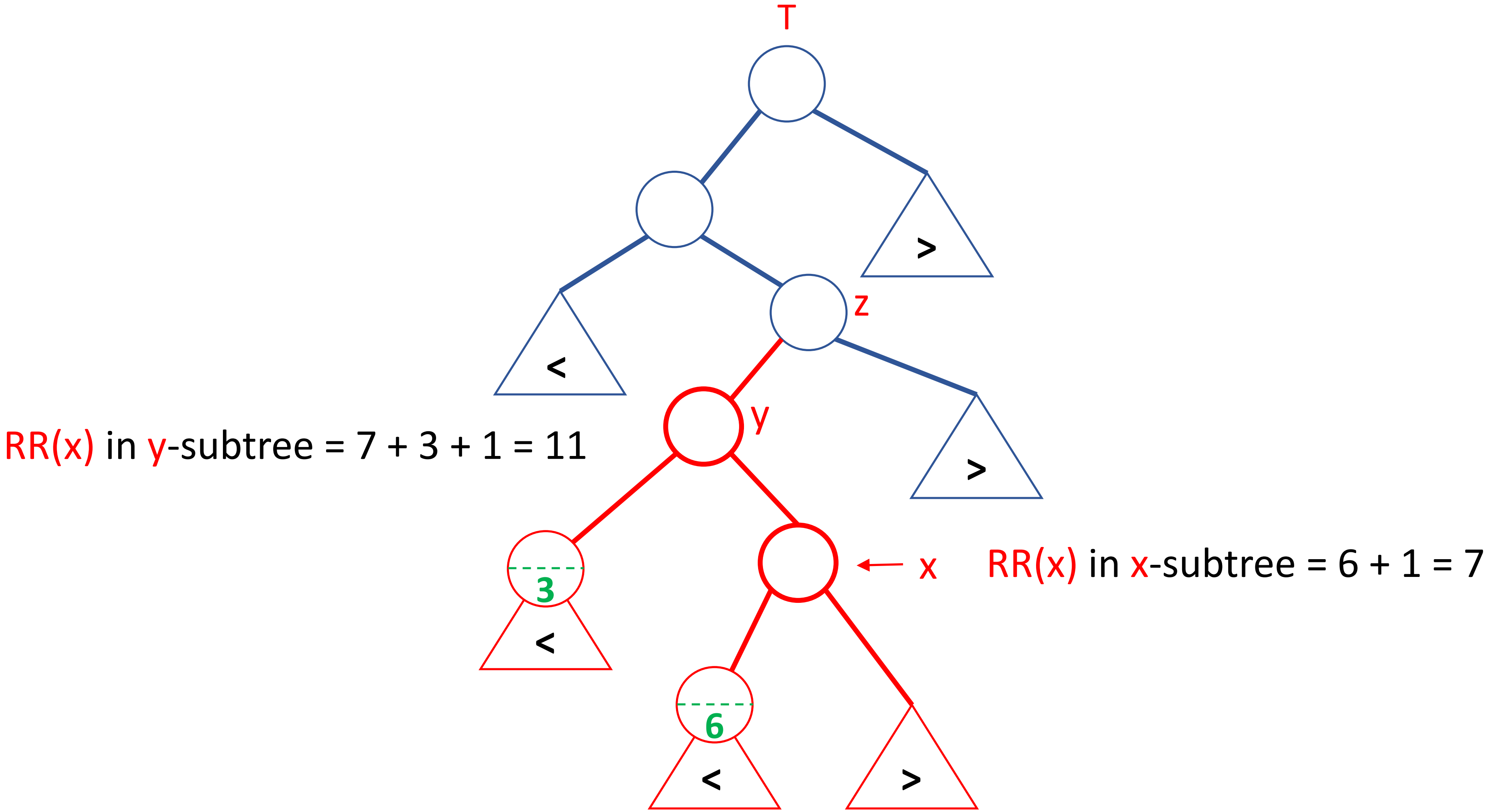
Rank(T, x): return rank of x in T

RR(x) in y-subtree = 7 + 3 + 1 = 11

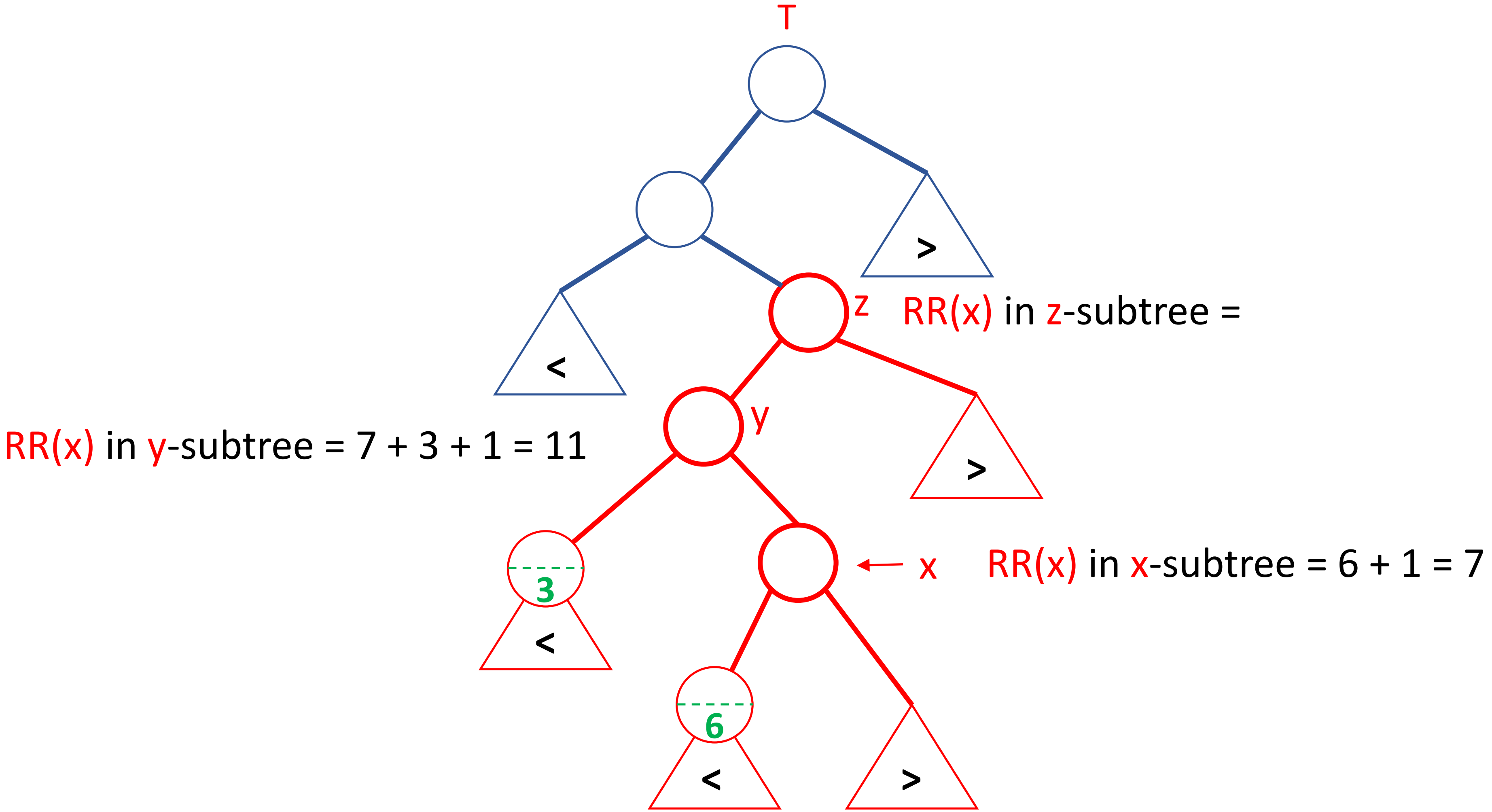


RR(x) in x-subtree = 6 + 1 = 7

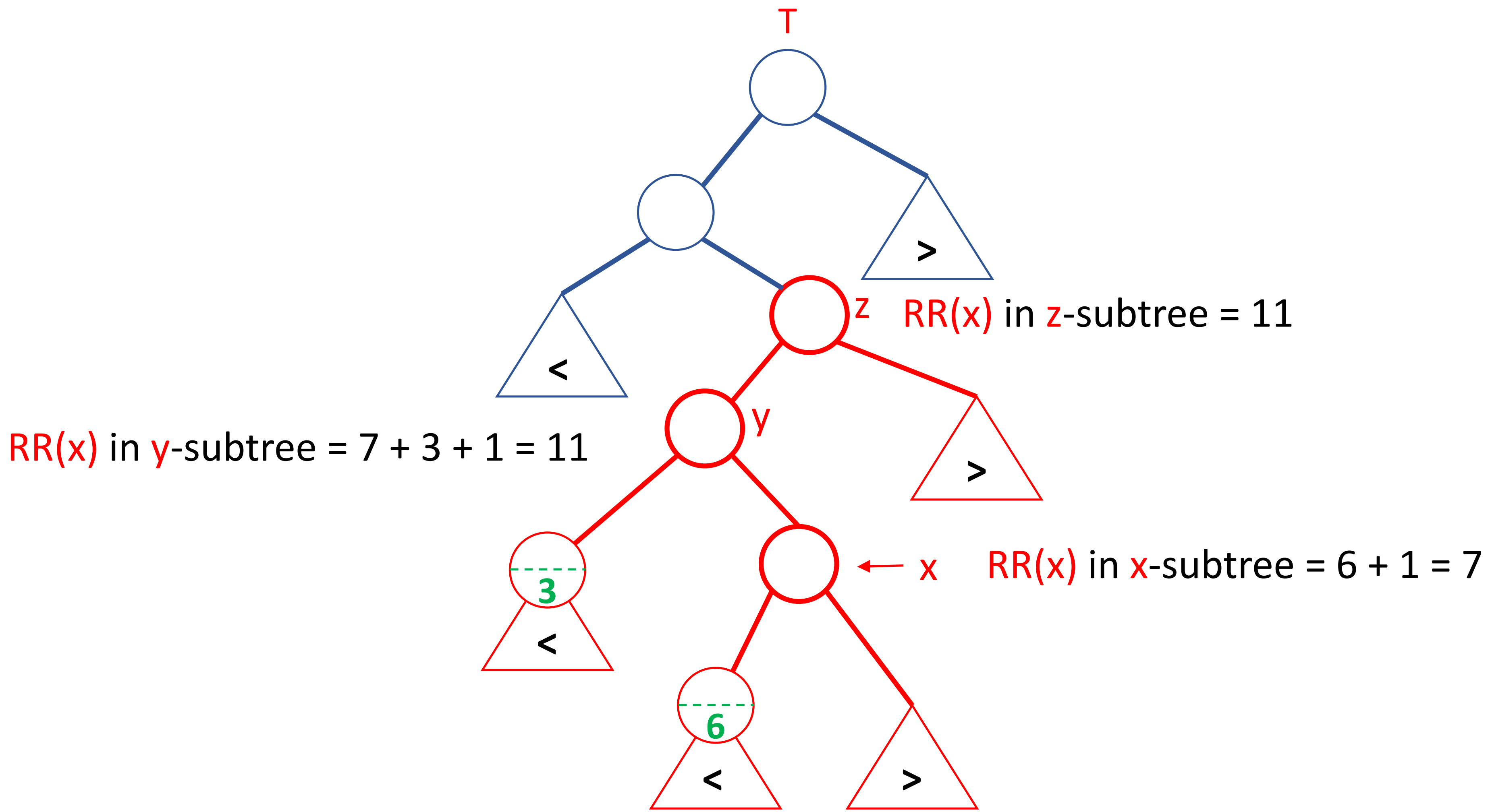
Rank(T, x): return rank of x in T



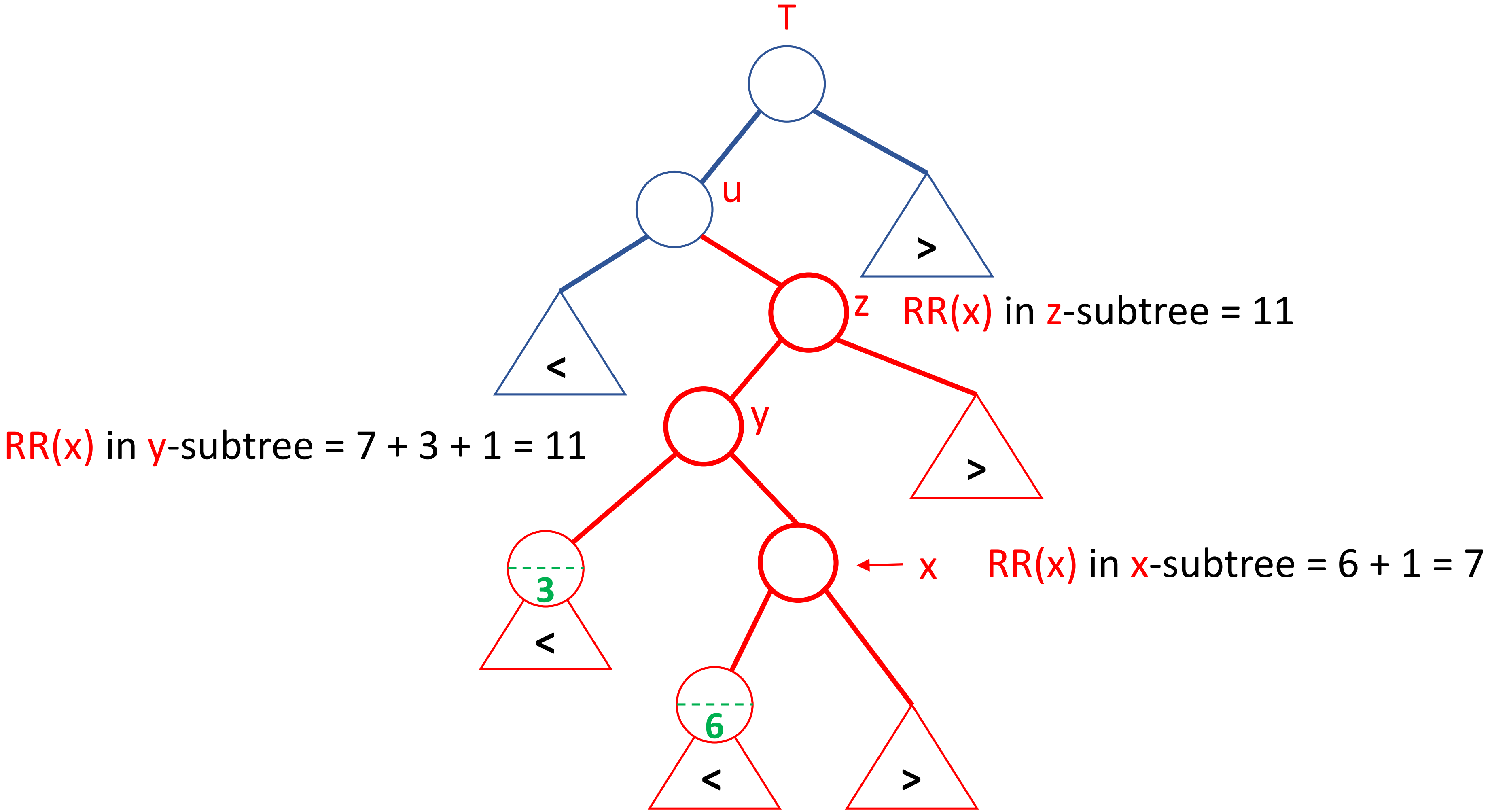
Rank(T, x): return rank of x in T



Rank(T, x): return rank of x in T

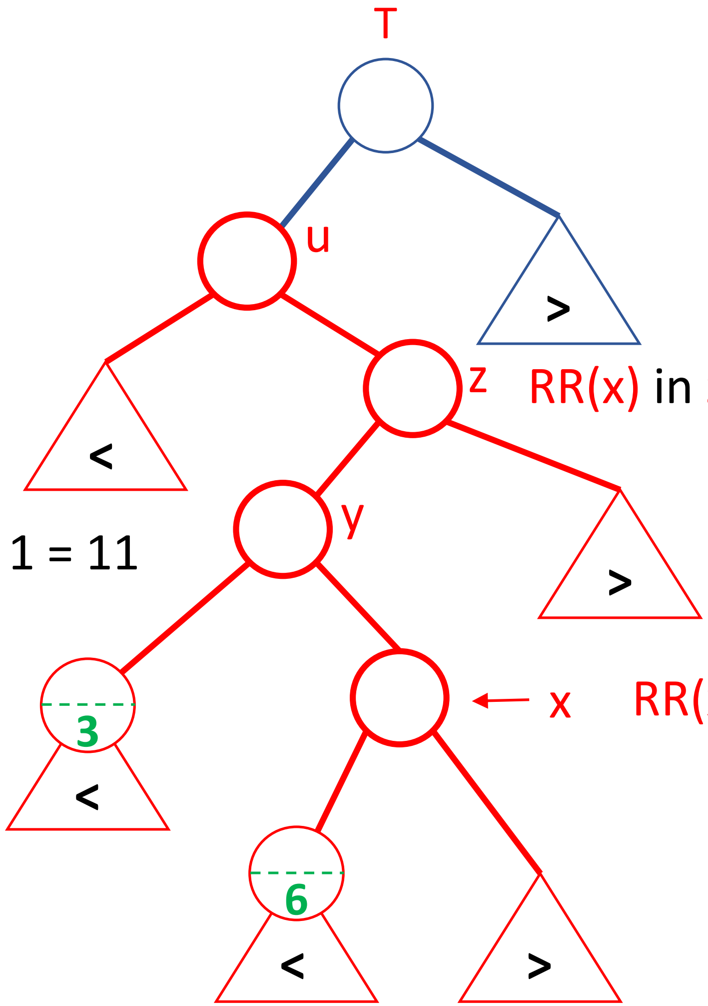


Rank(T, x): return rank of x in T



Rank(T, x): return rank of x in T

RR(x) in u-subtree =

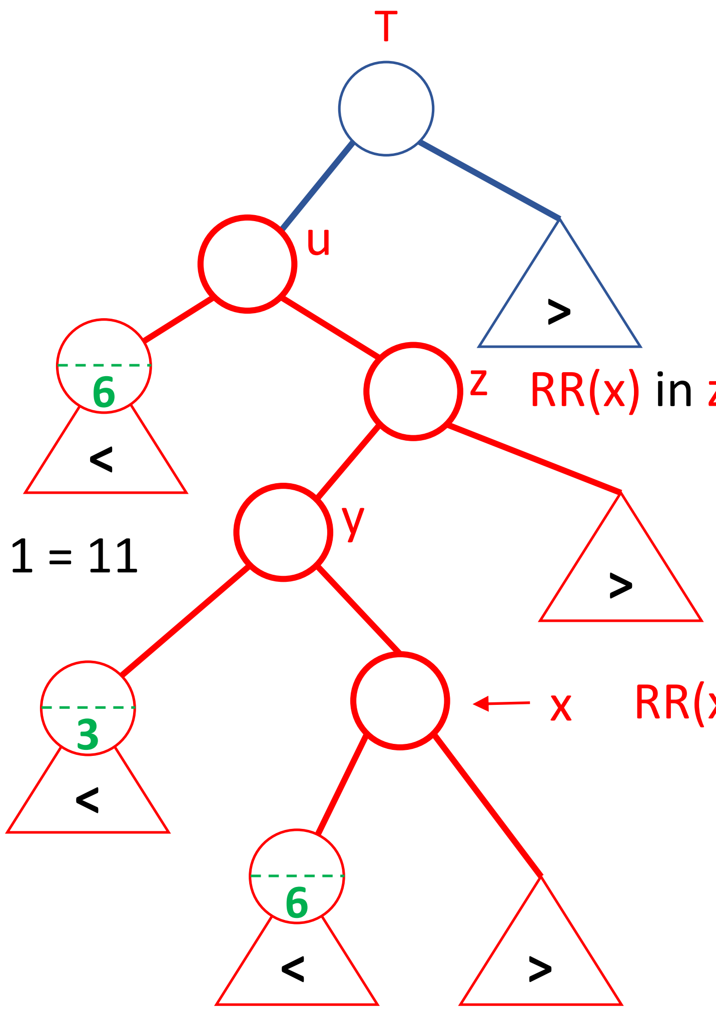


RR(x) in z-subtree = 11

RR(x) in y-subtree = 7 + 3 + 1 = 11

RR(x) in x-subtree = 6 + 1 = 7

Rank(T, x): return rank of x in T



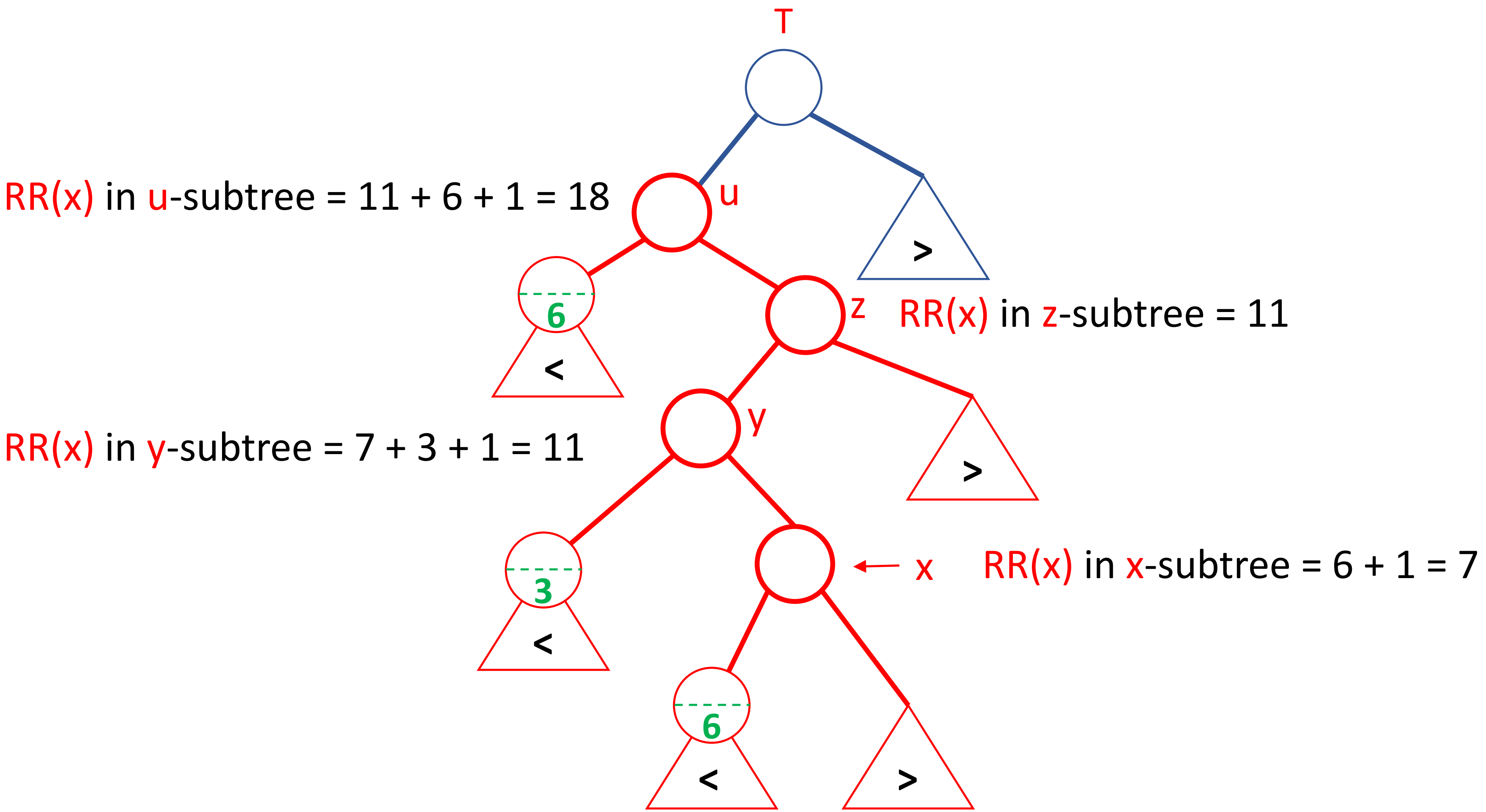
RR(x) in u-subtree =

RR(x) in z-subtree = 11

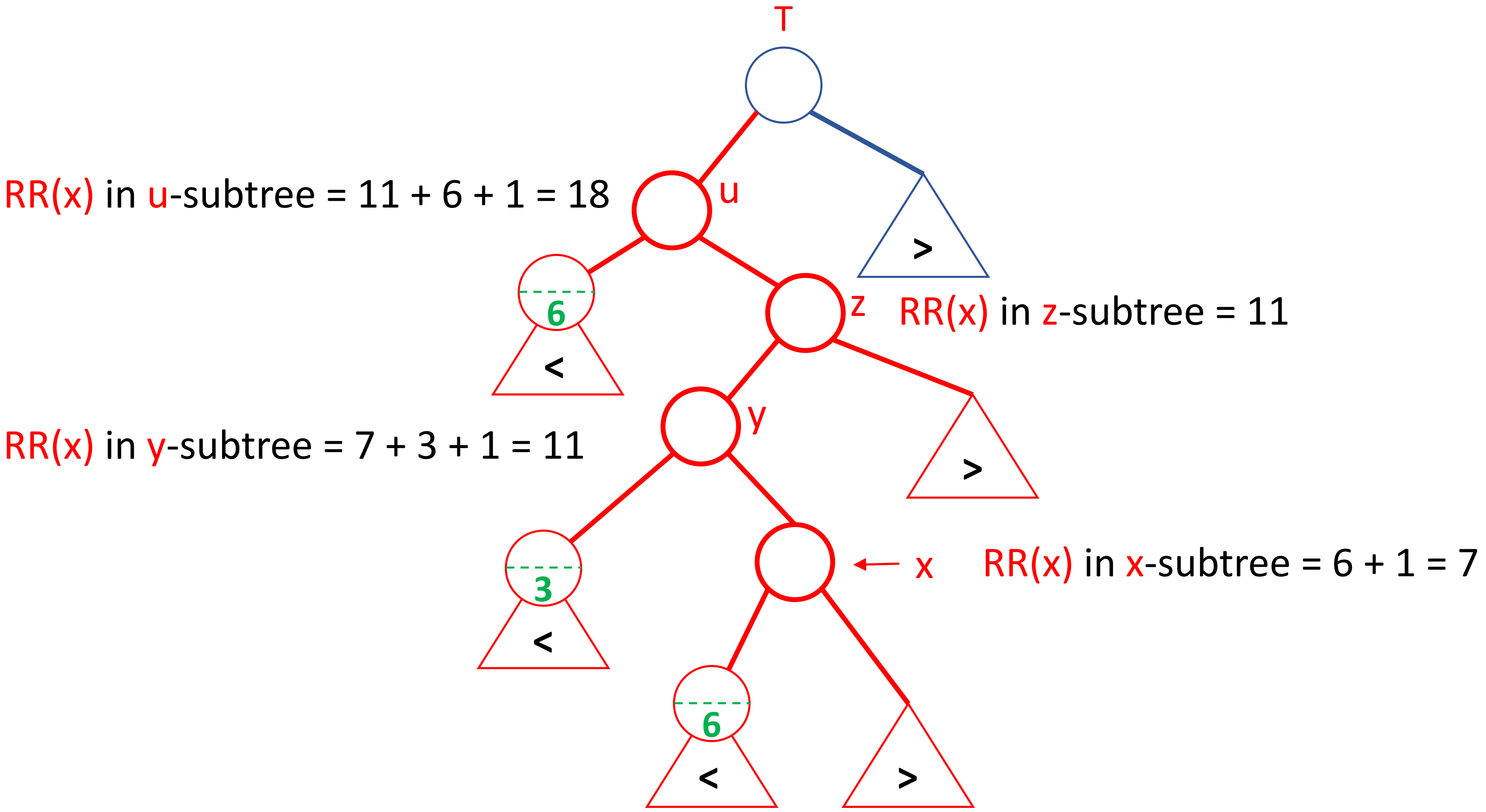
RR(x) in y-subtree = 7 + 3 + 1 = 11

RR(x) in x-subtree = 6 + 1 = 7

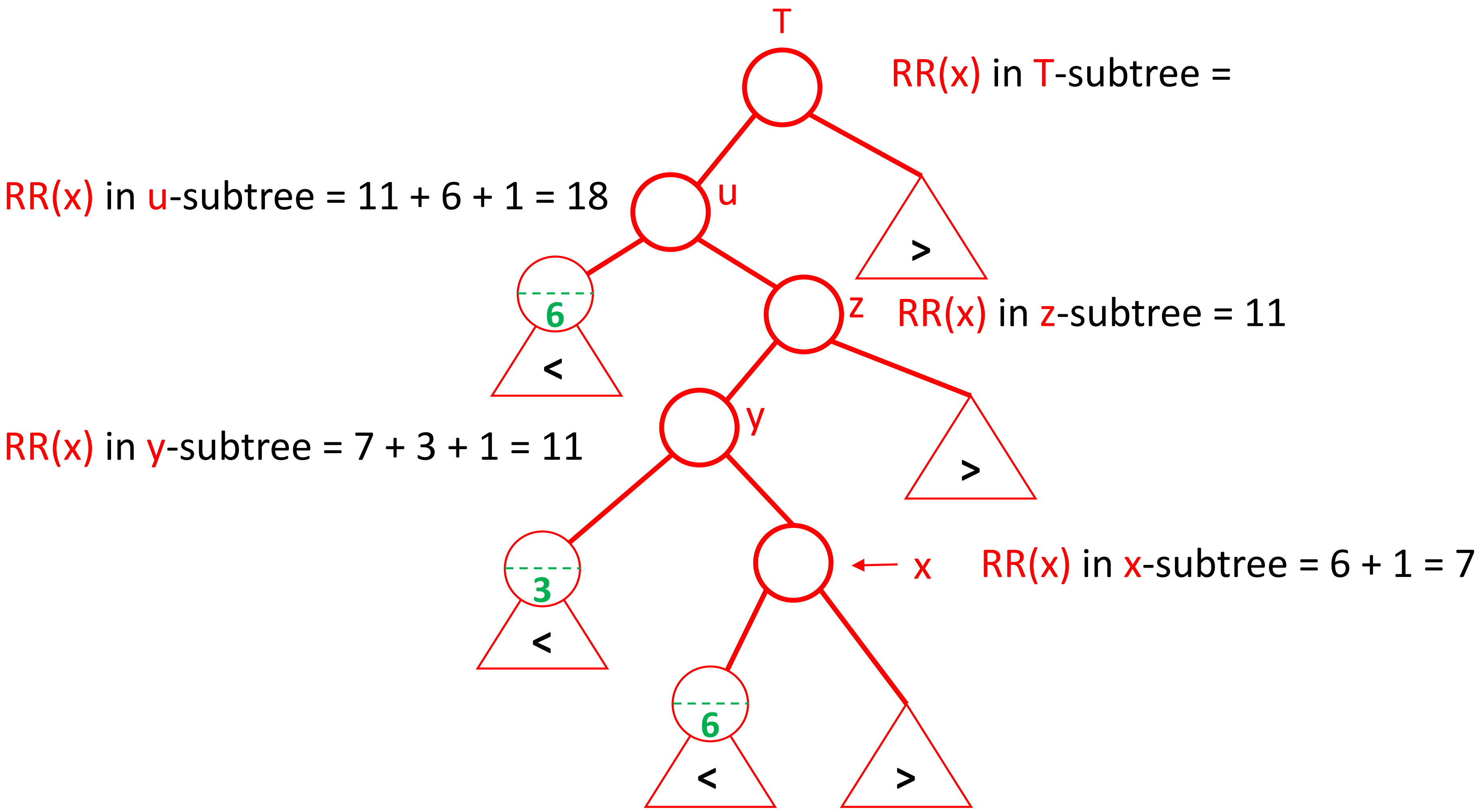
Rank(T, x): return rank of x in T



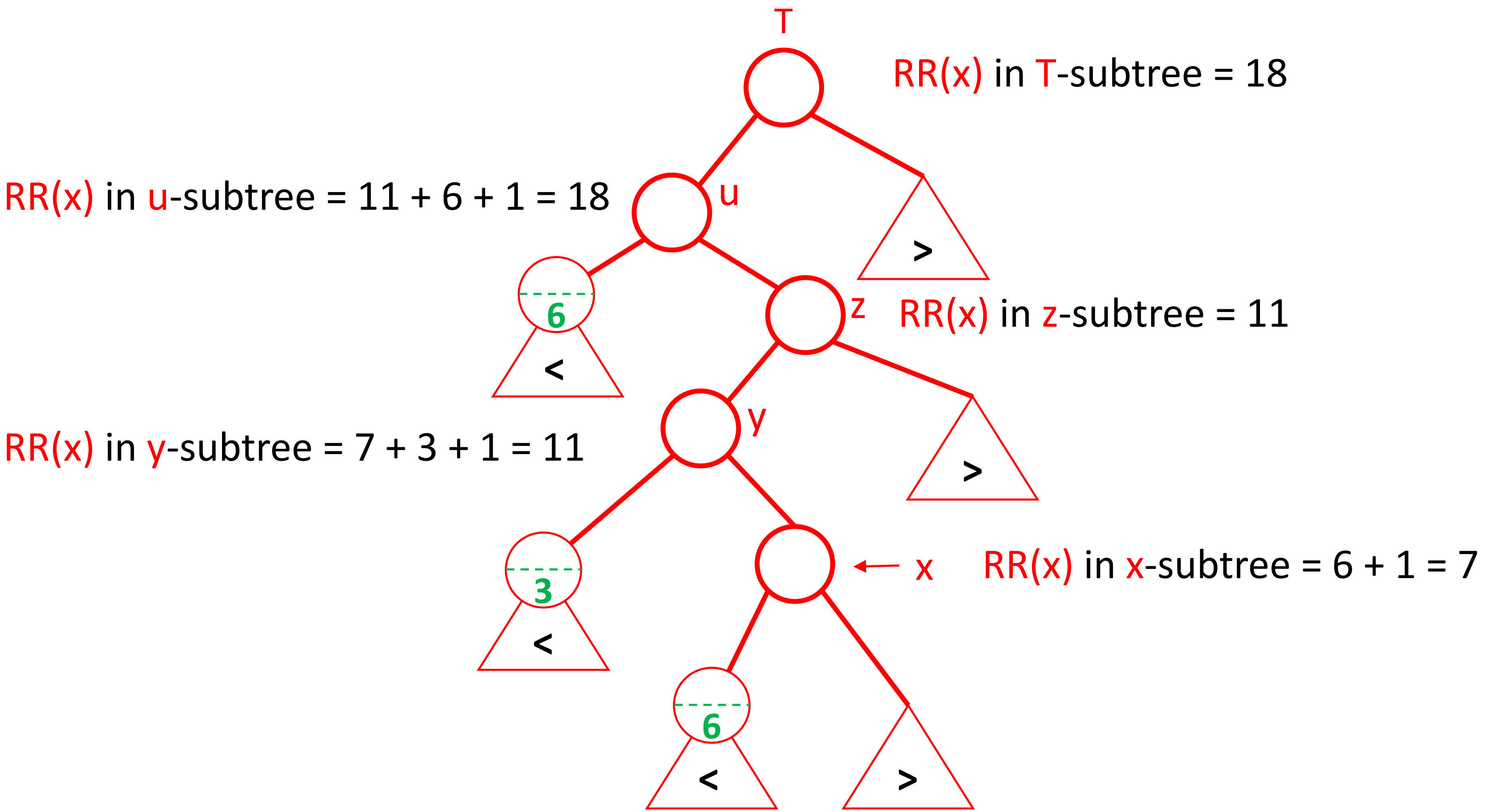
Rank(T, x): return rank of x in T



Rank(T, x): return rank of x in T



Rank(T, x): return rank of x in T



Rank(T, x): return rank of x in T

- Find the rank of x in x-subtree:

  RR(x) ← size(left(x)) + 1

Rank(T, x): return rank of x in T

- Find the rank of x in x-subtree:

  RR(x) ← size(left(x)) + 1

- For each node y in path x to root of T :

  Compute rank of x in y-subtree as shown in previous example

Rank(T, x): return rank of x in T

- Find the rank of x in x-subtree:

  RR(x) ← size(left(x)) + 1

- For each node y in path x to root of T :

  Compute rank of x in y-subtree as shown in previous example

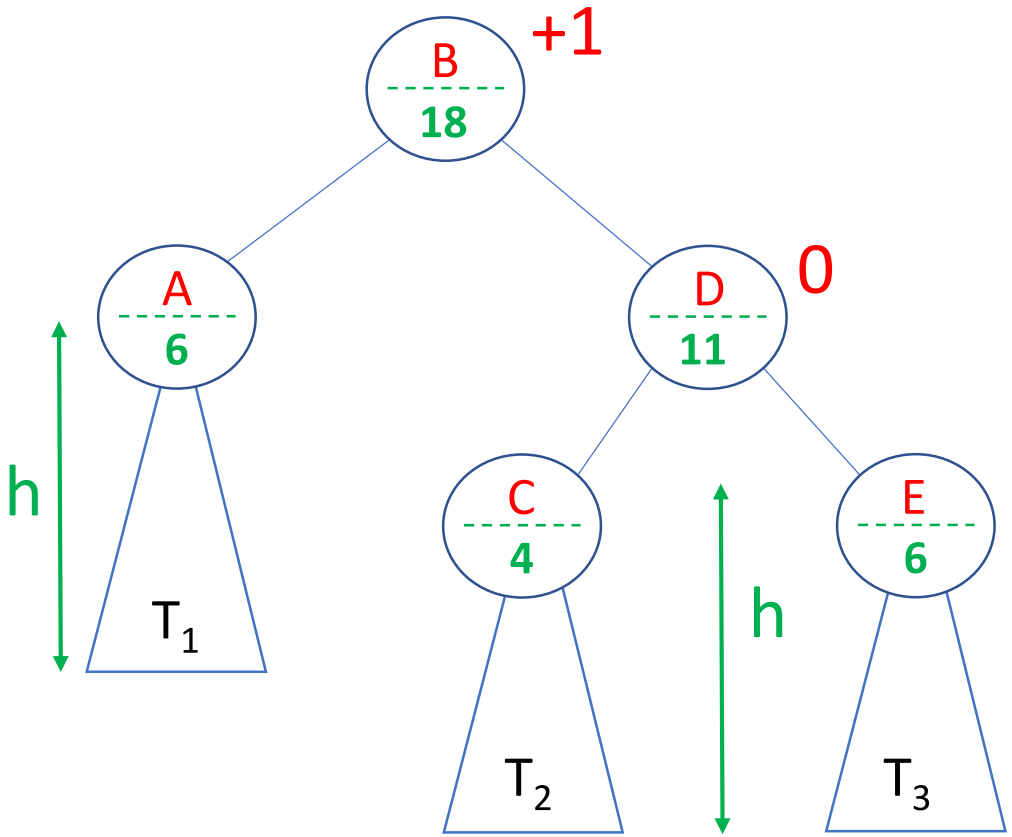
Worst-Case Time Complexity of **Rank**(T, k):

- Constant time for each level of T
- Height(T) is O(log n)
- Hence **Rank** takes O(log n)

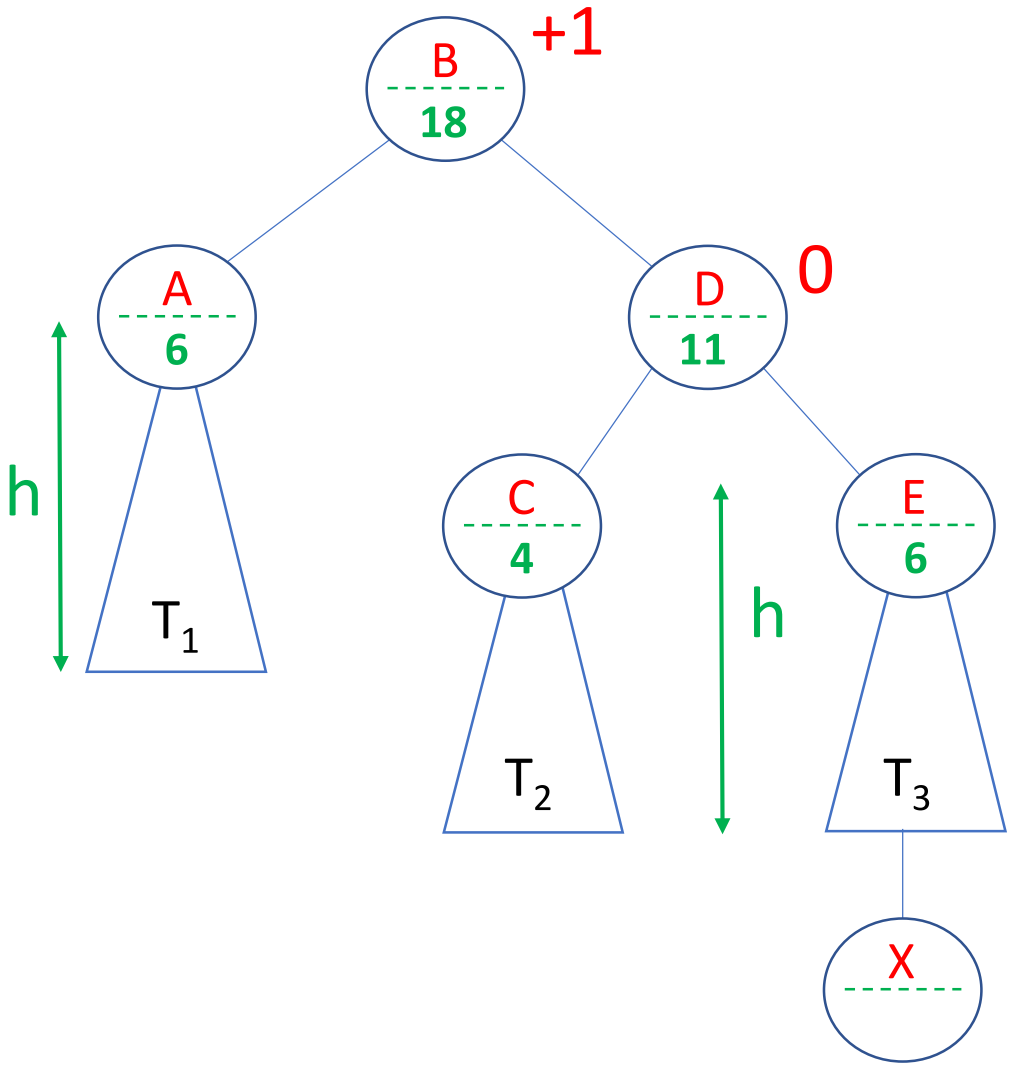
Augmenting AVL

- **Select** operation ✓
- **Rank** operation ✓
- Maintain size() field
  after **Insert or Delete**

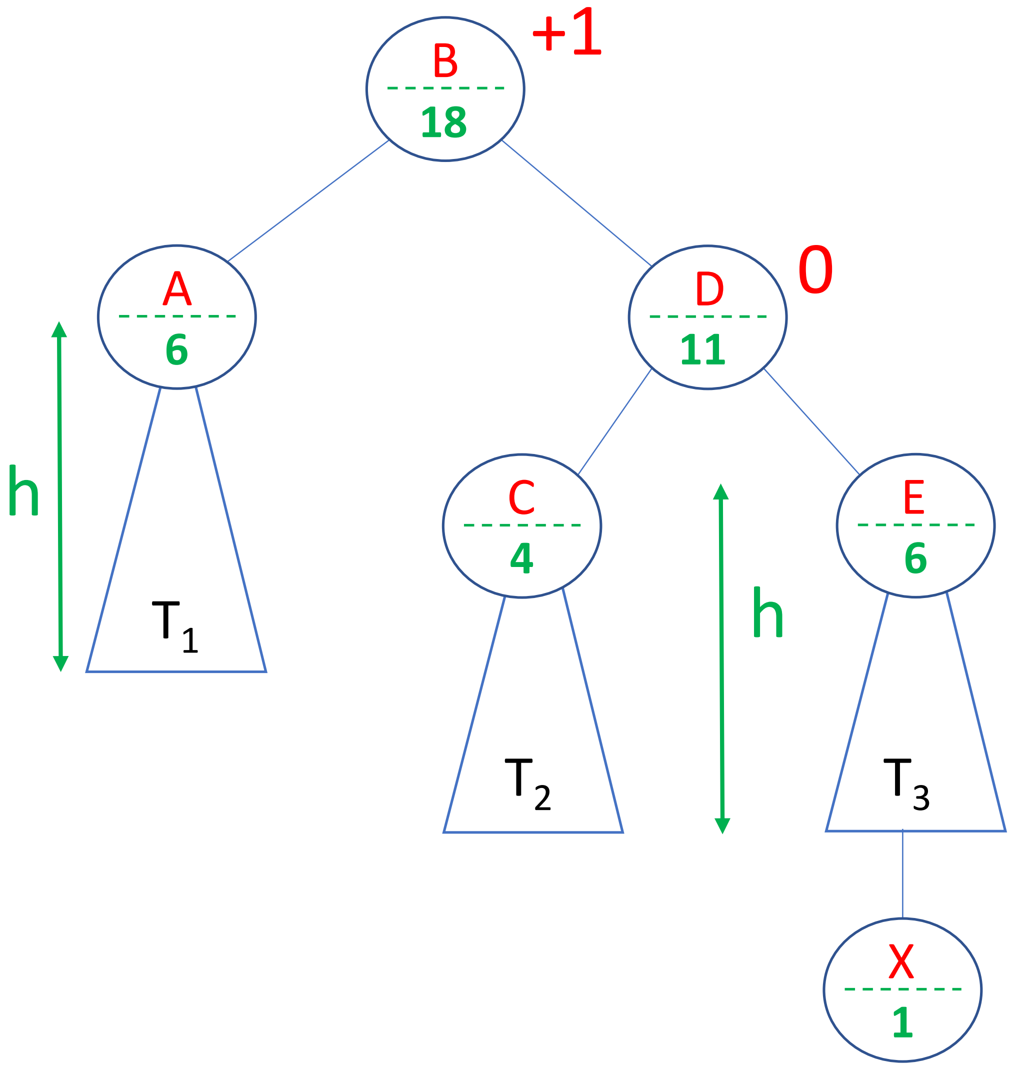
Maintaining size() field: Insert Example



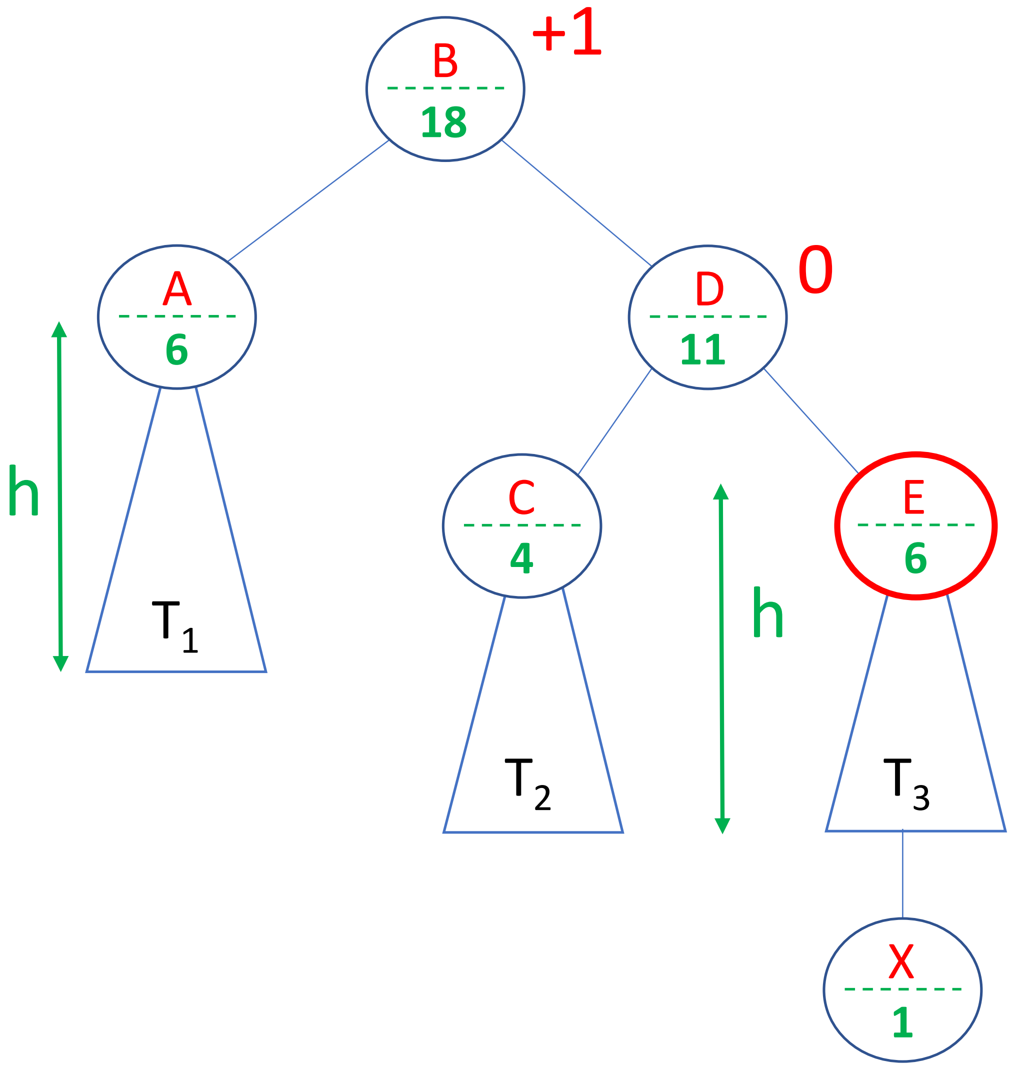
Maintaining size() field: Insert Example



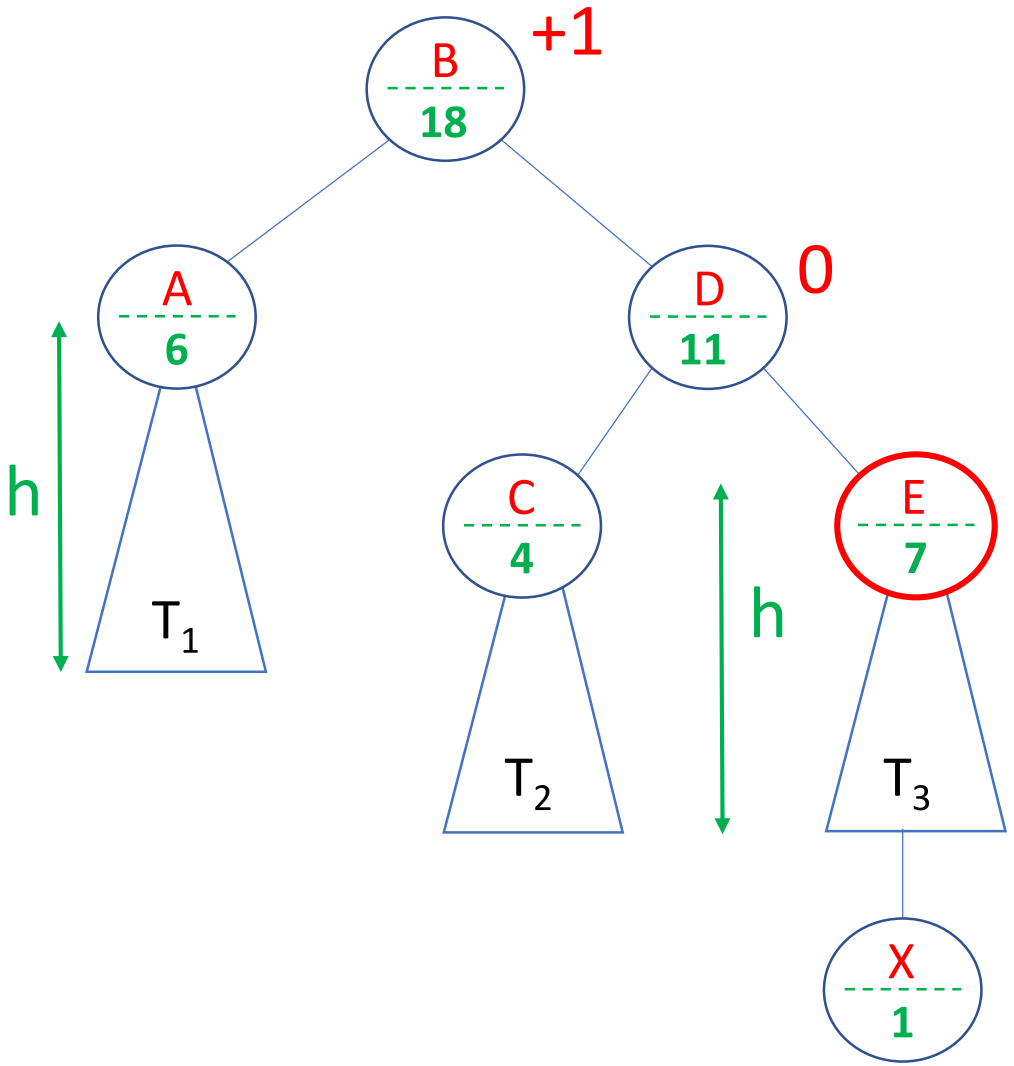
Maintaining size() field: Insert Example



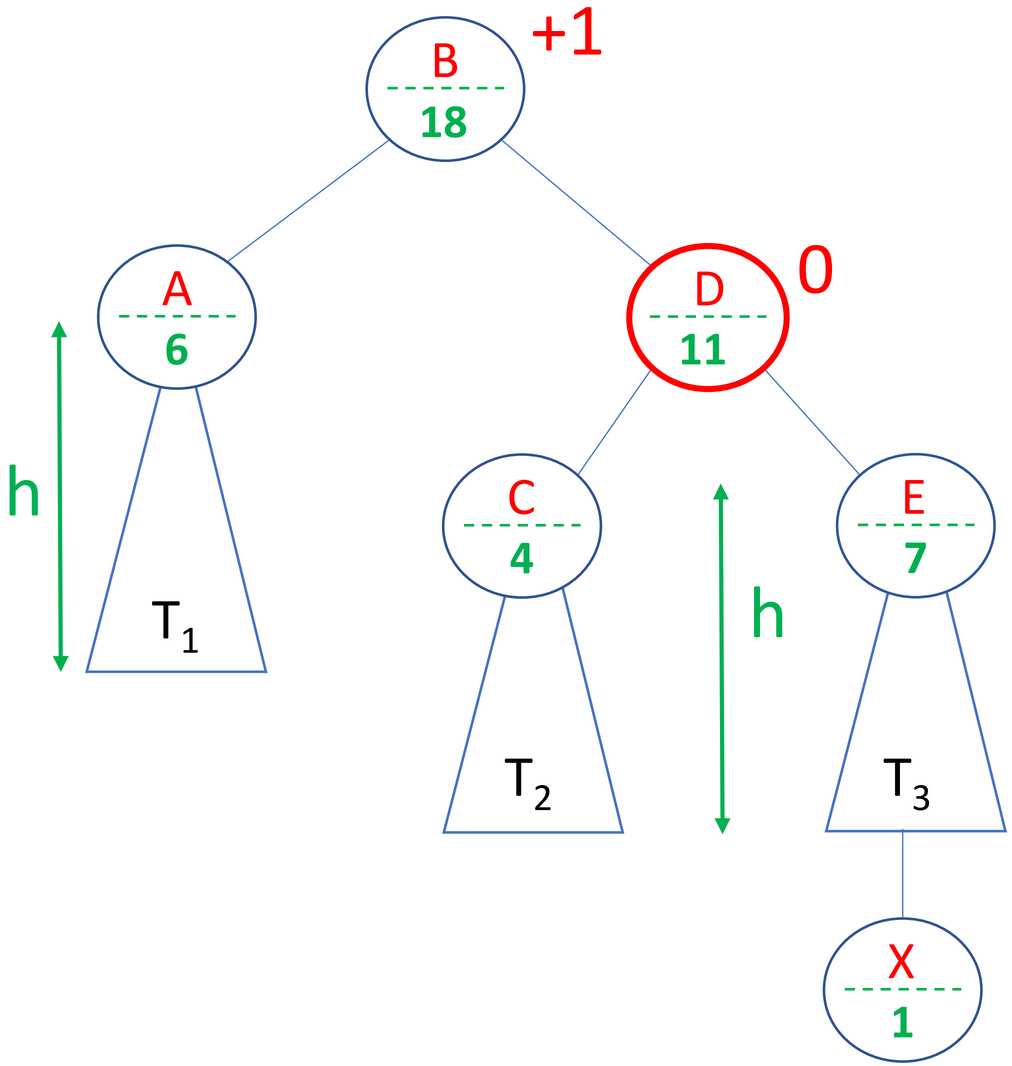
Maintaining size() field: Insert Example



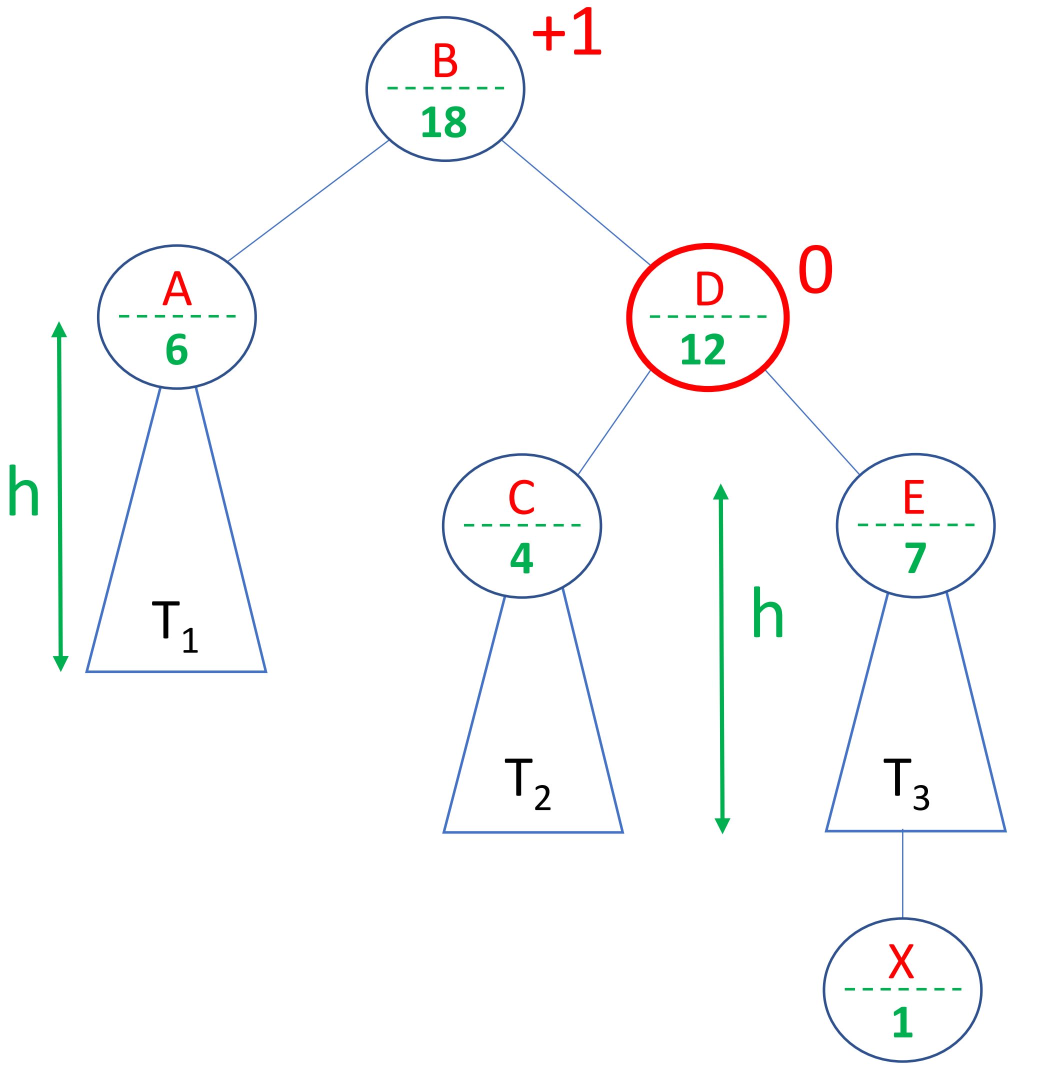
Maintaining size() field: Insert Example



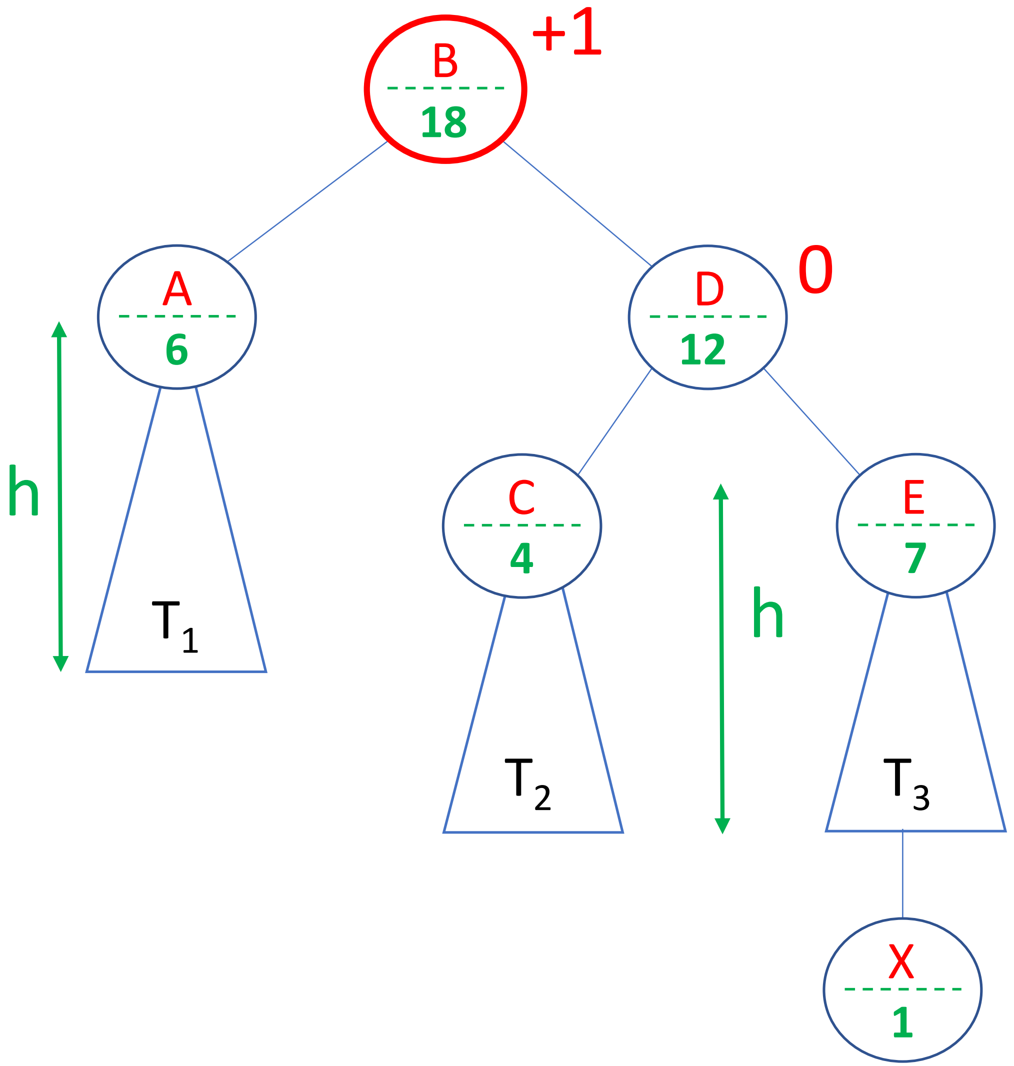
Maintaining size() field: Insert Example



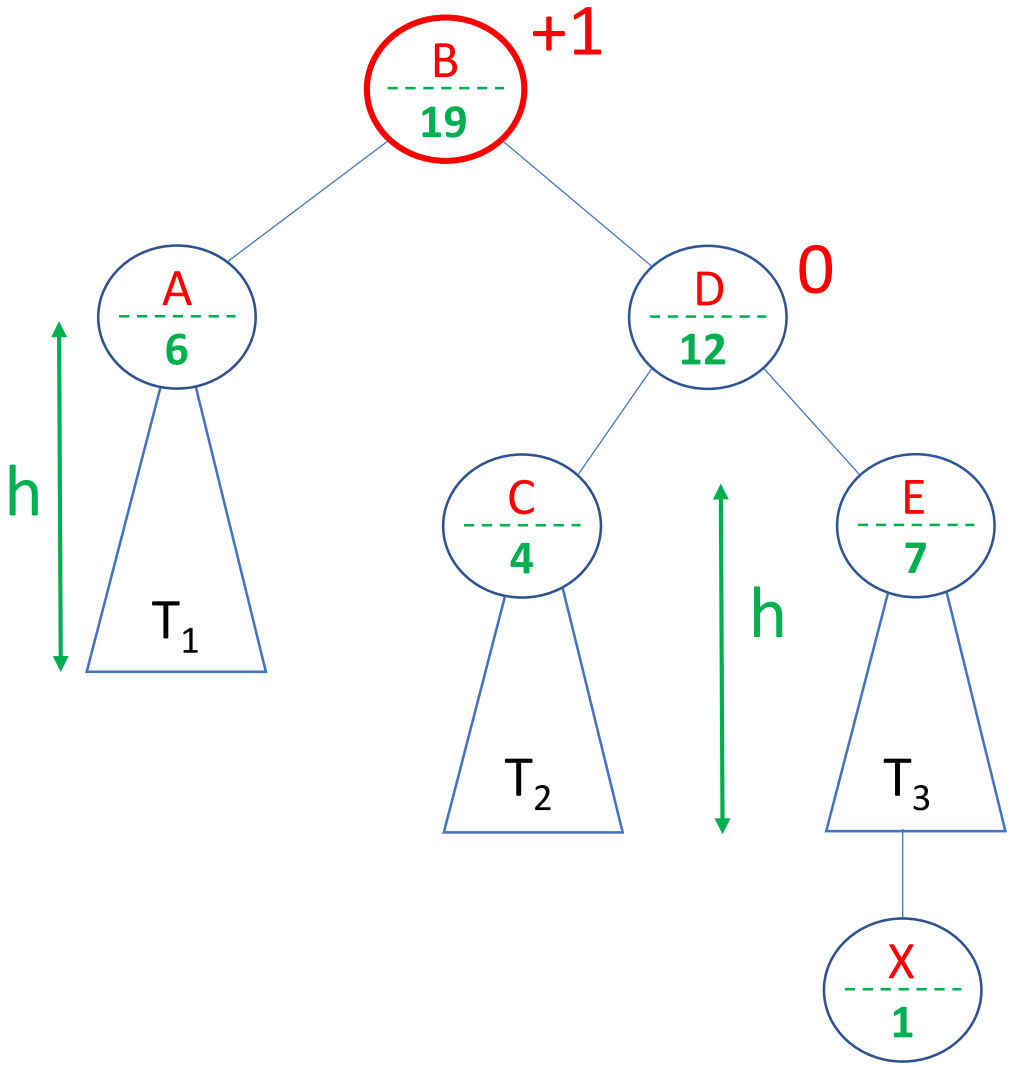
Maintaining size() field: Insert Example



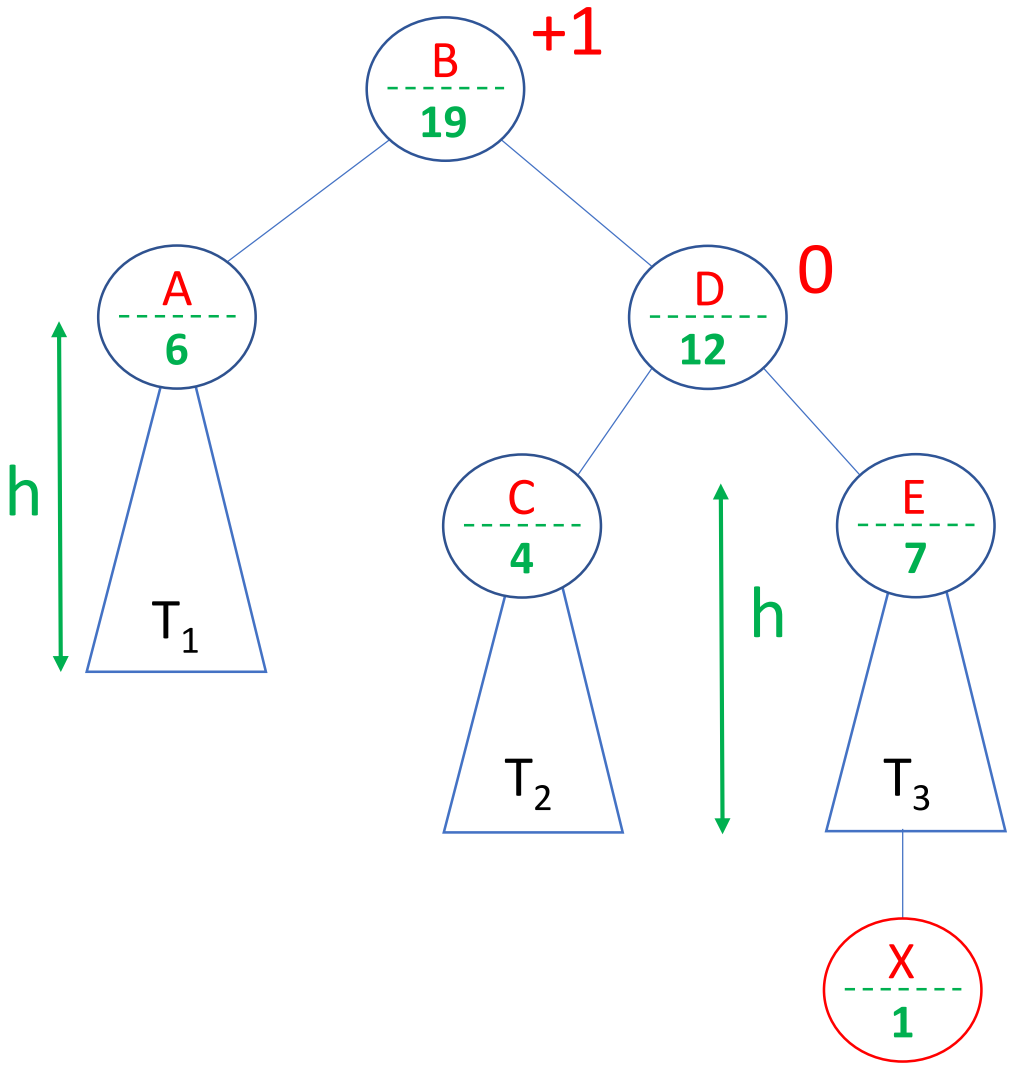
Maintaining size() field: Insert Example



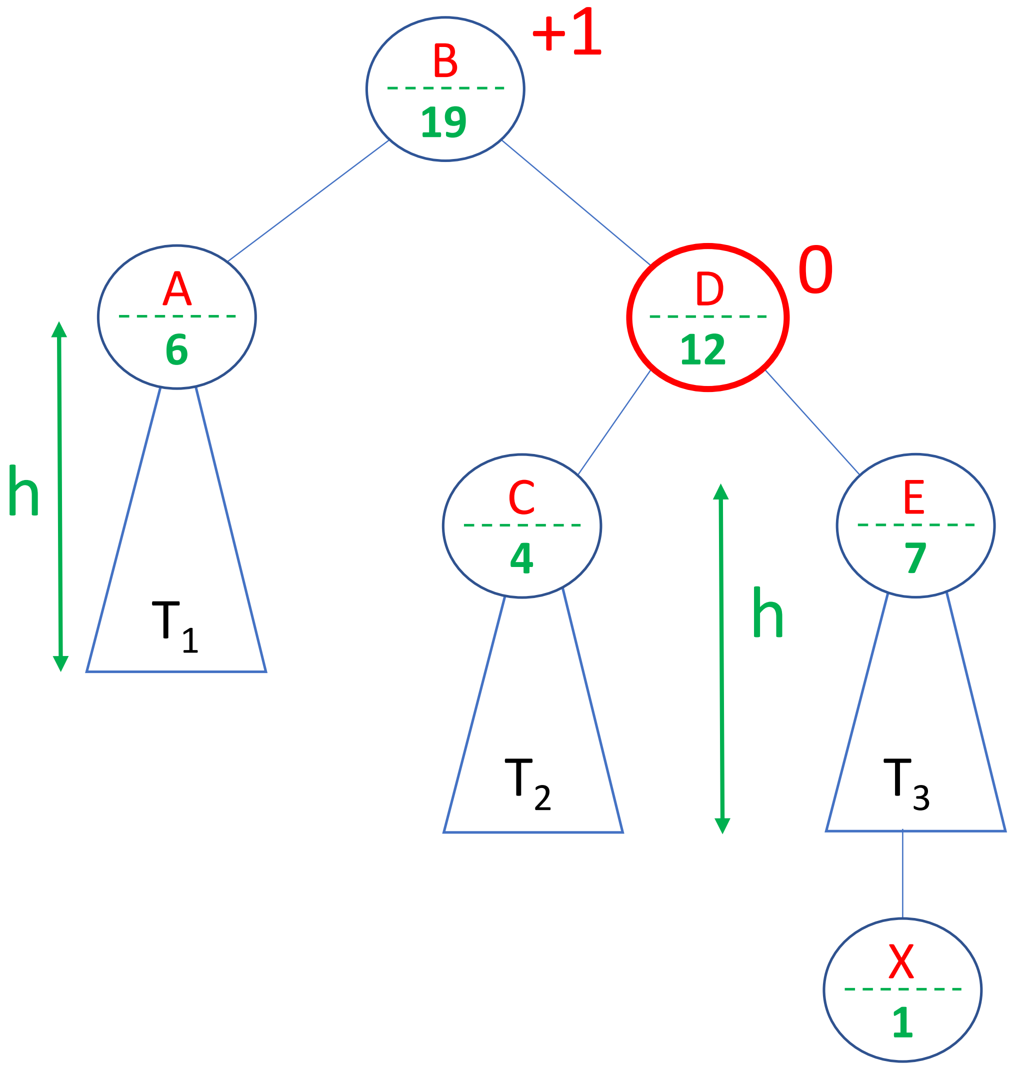
Maintaining size() field: Insert Example



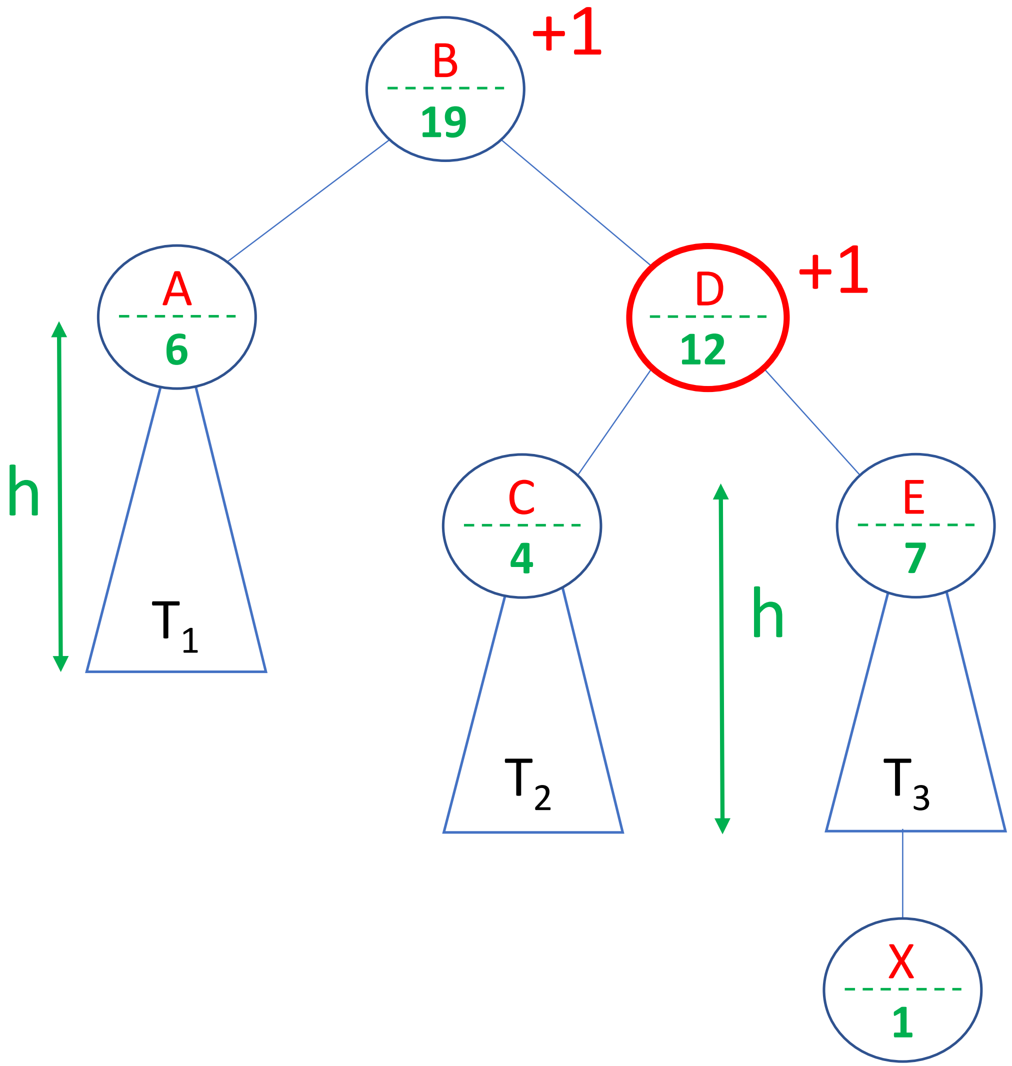
Maintaining size() field: Insert Example



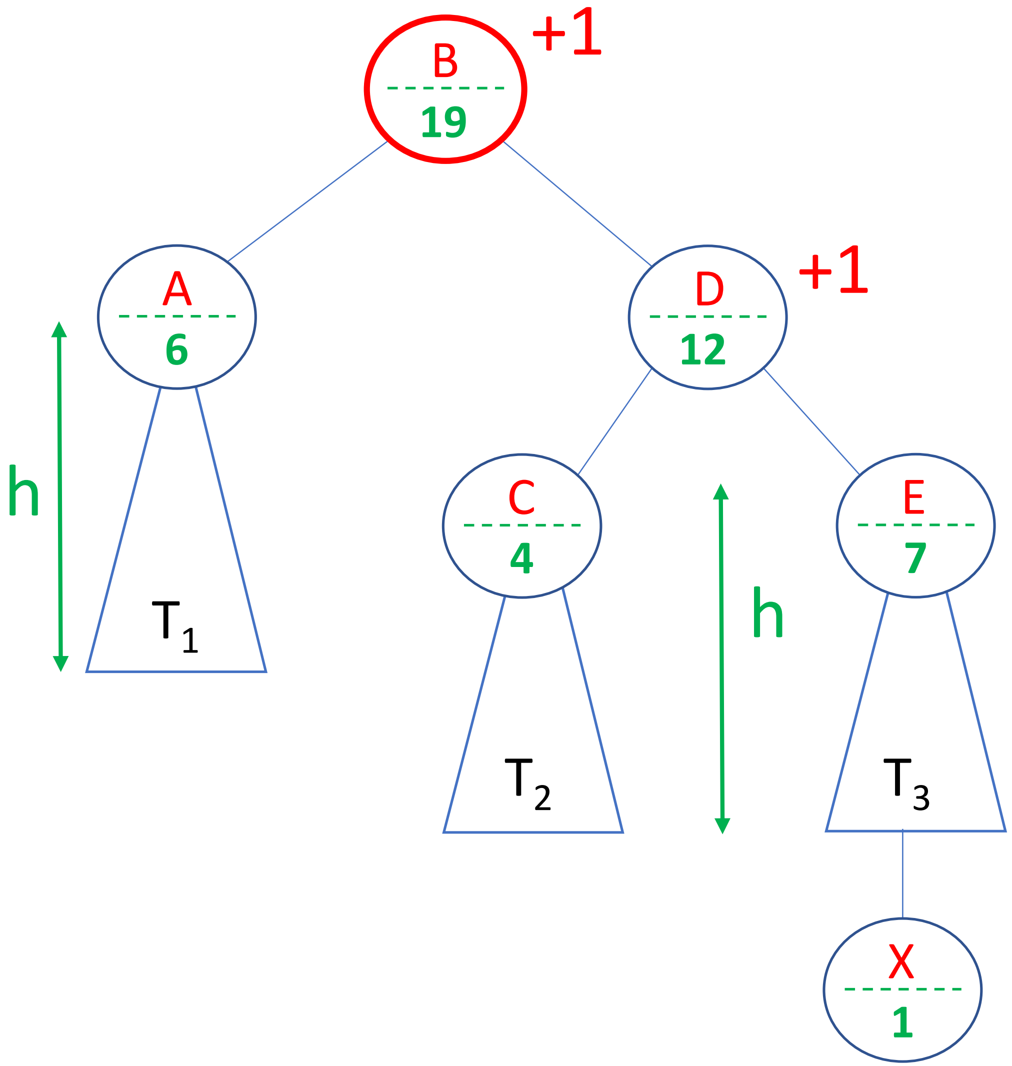
Maintaining size() field: Insert Example



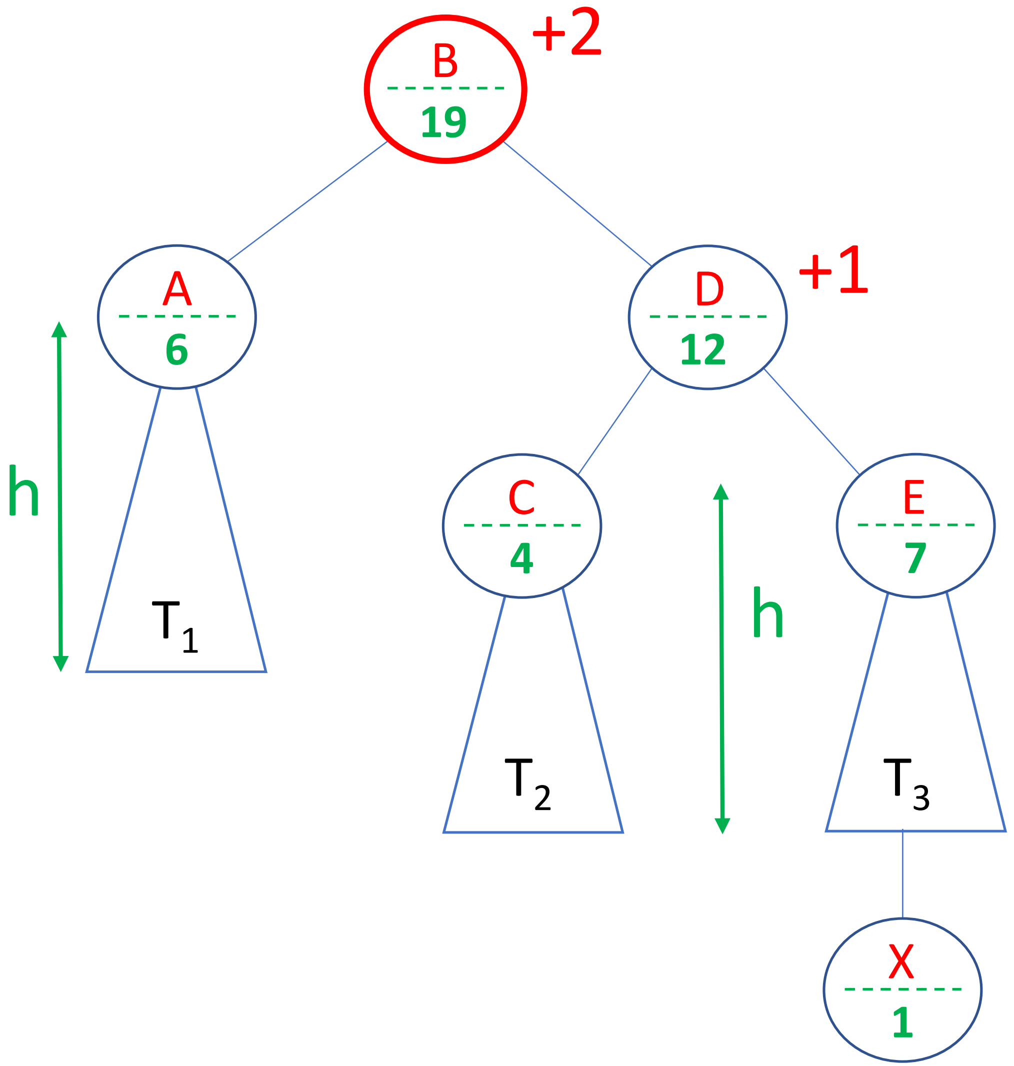
Maintaining size() field: Insert Example



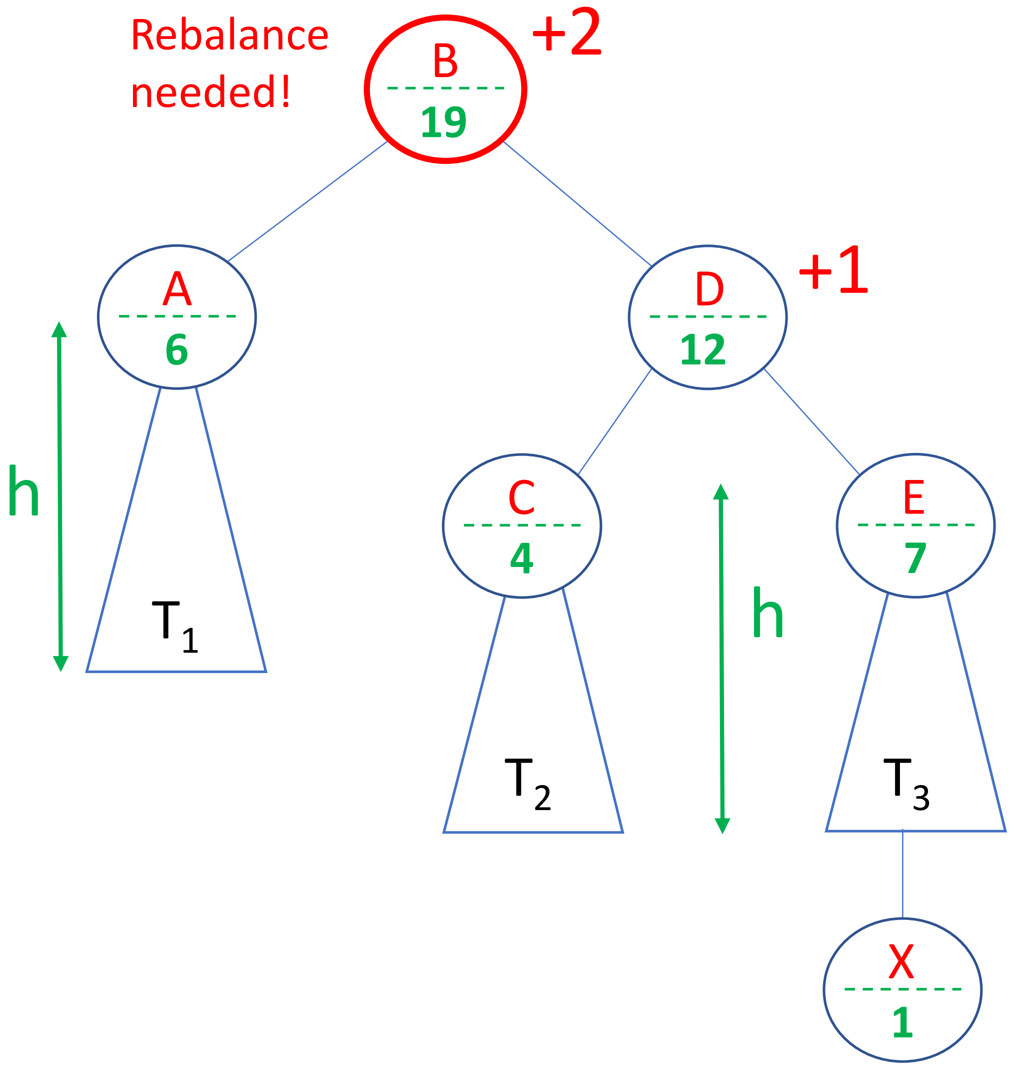
Maintaining size() field: Insert Example



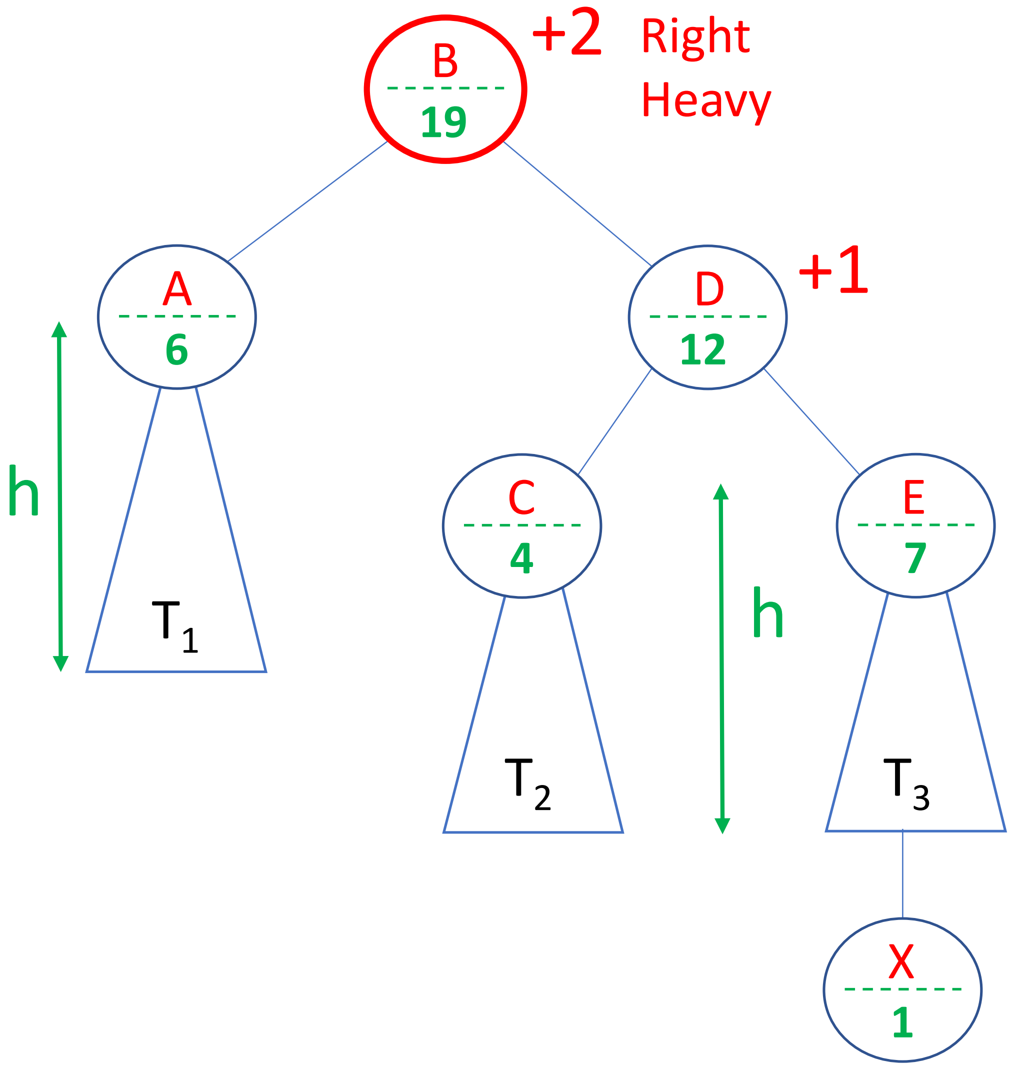
Maintaining size() field: Insert Example



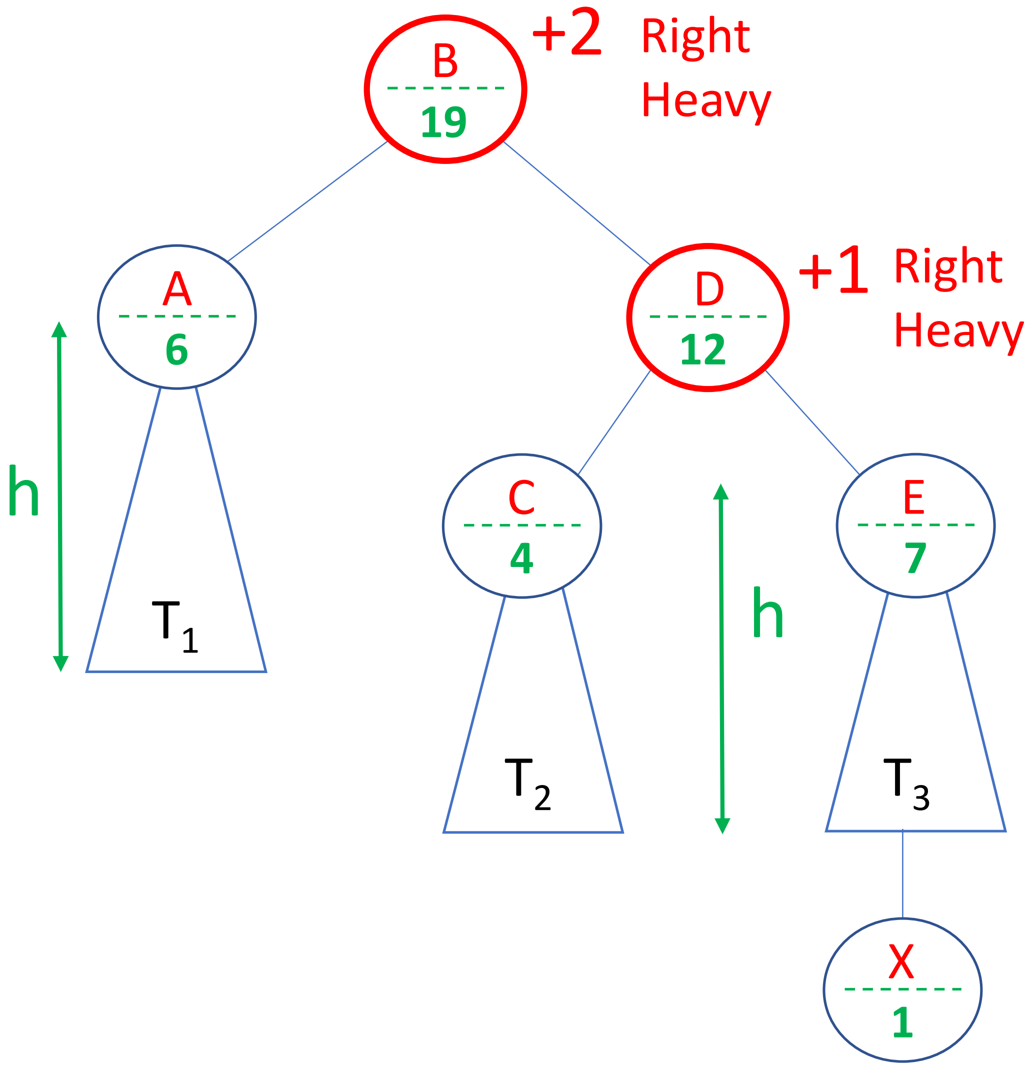
Maintaining size() field: Insert Example



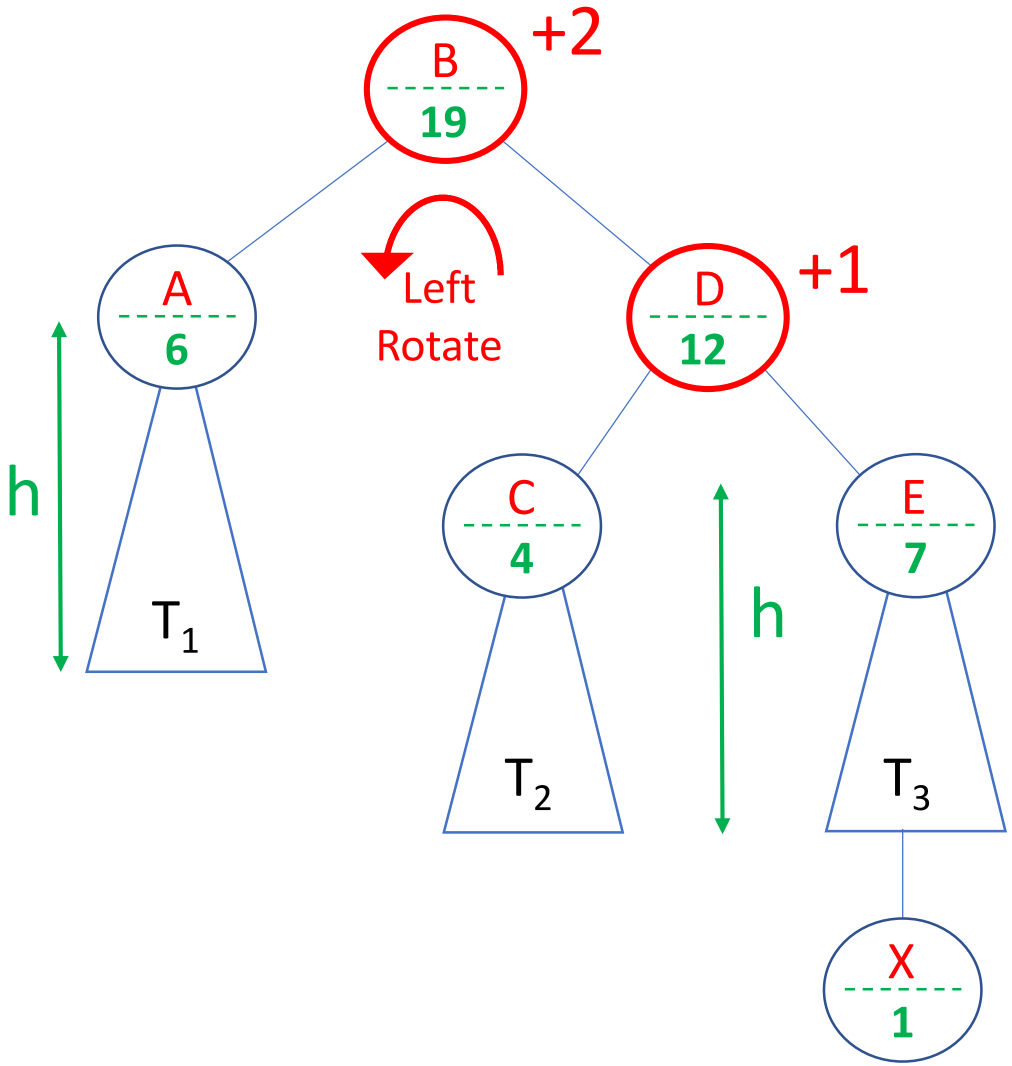
Maintaining size() field: Insert Example



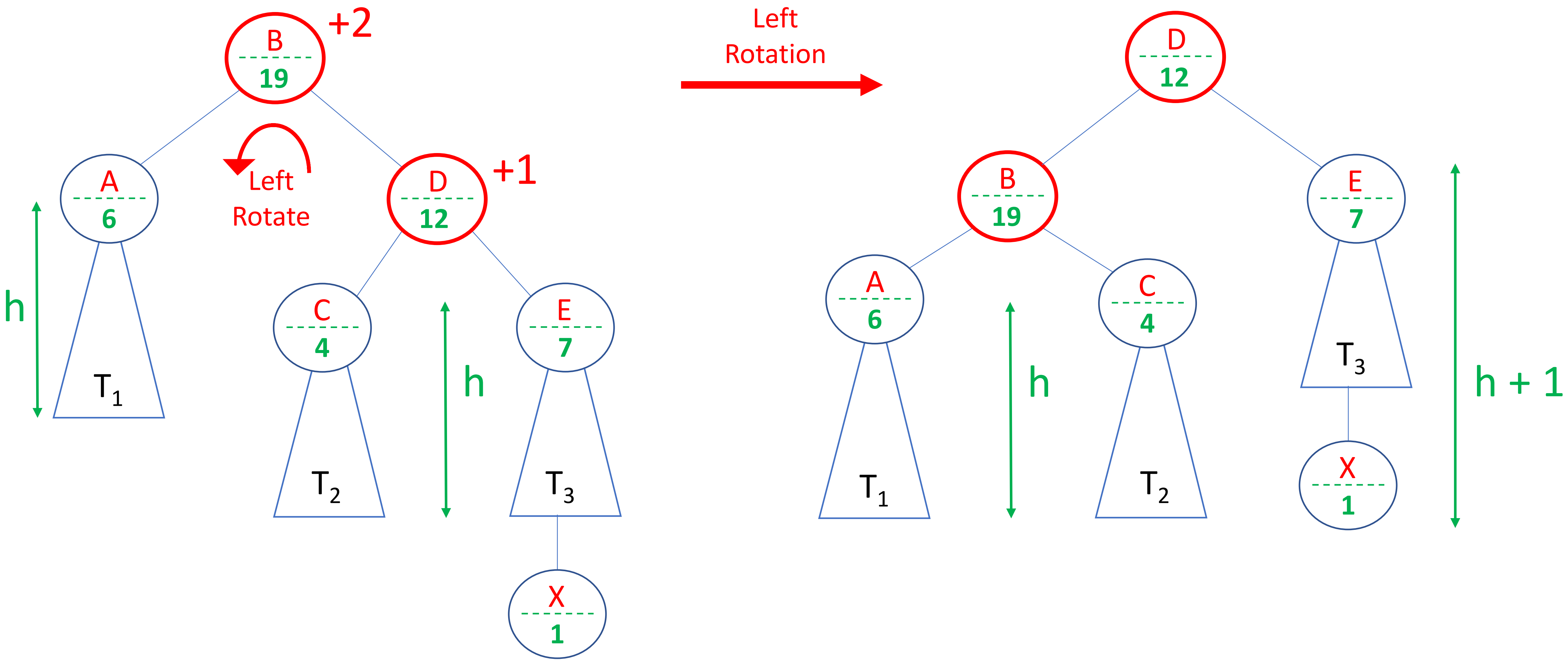
Maintaining size() field: Insert Example



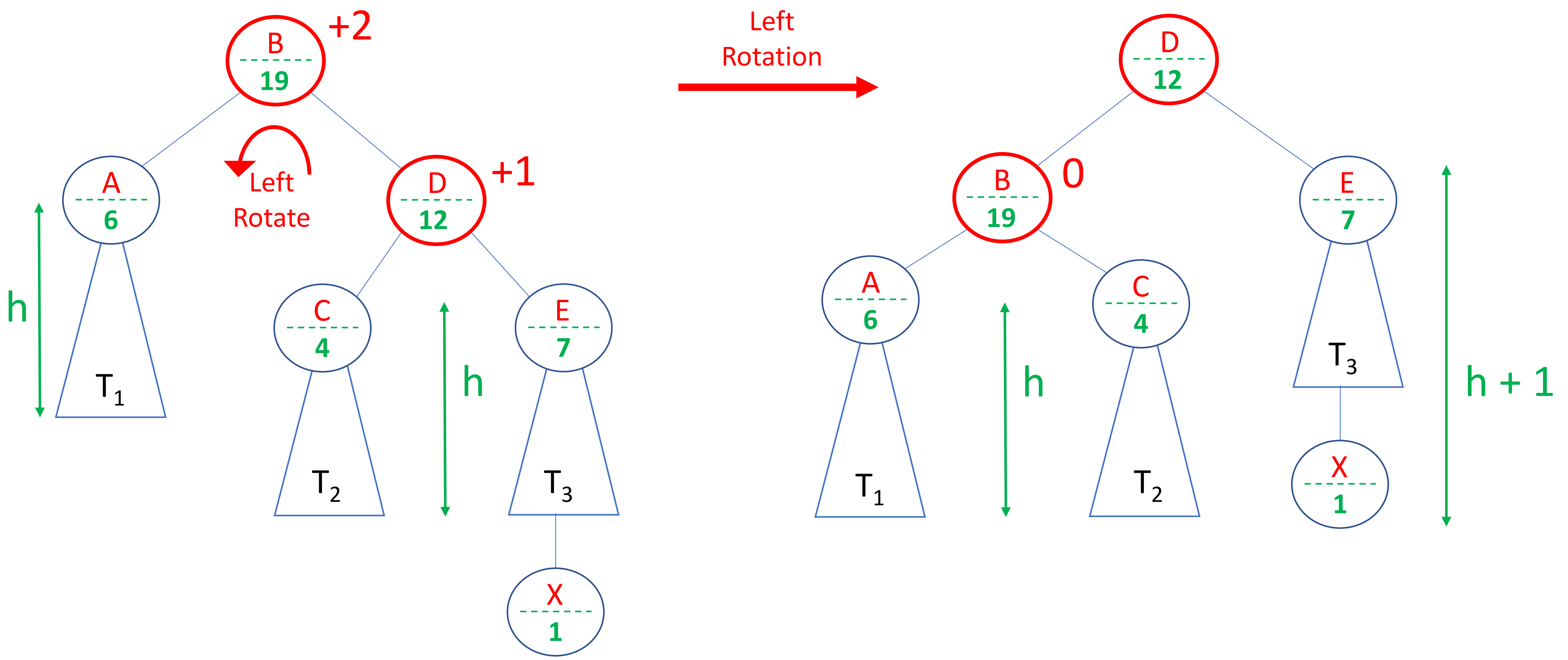
Maintaining size() field: Insert Example



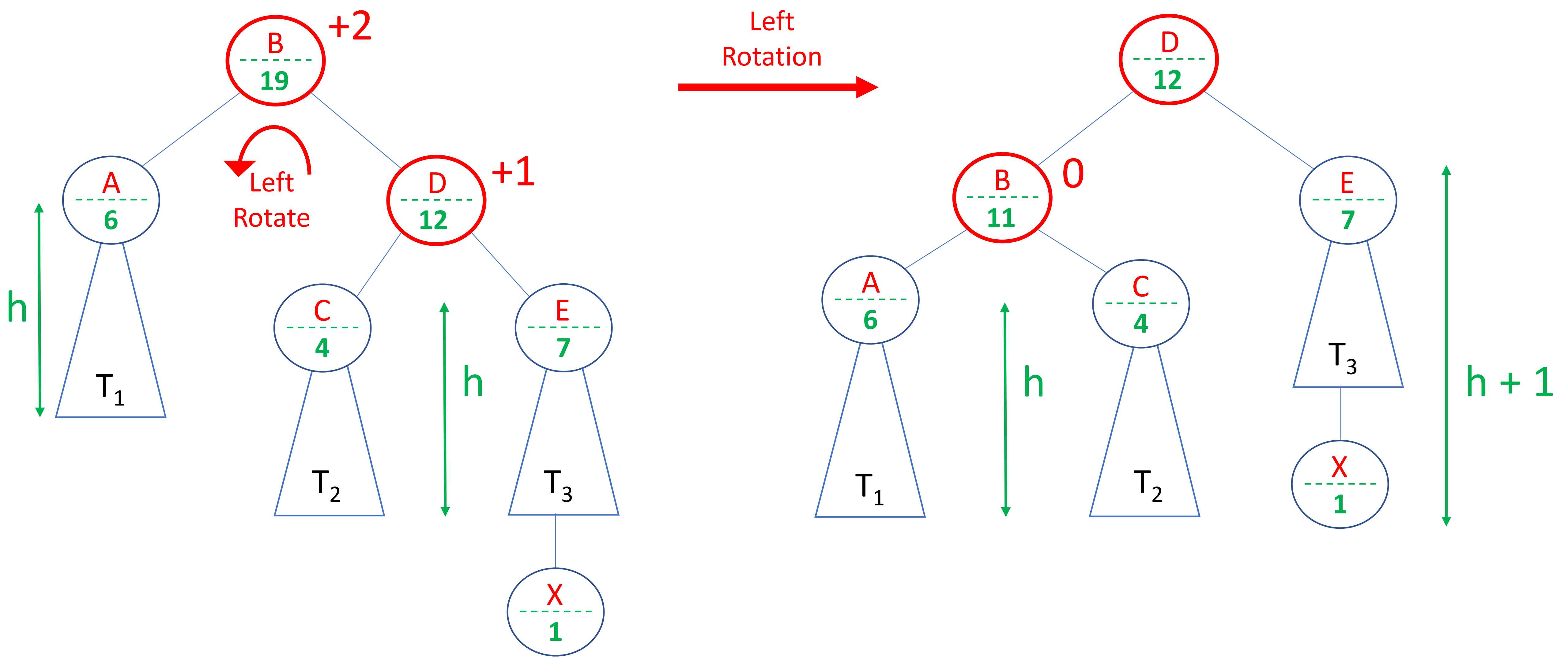
Maintaining size() field: Insert Example



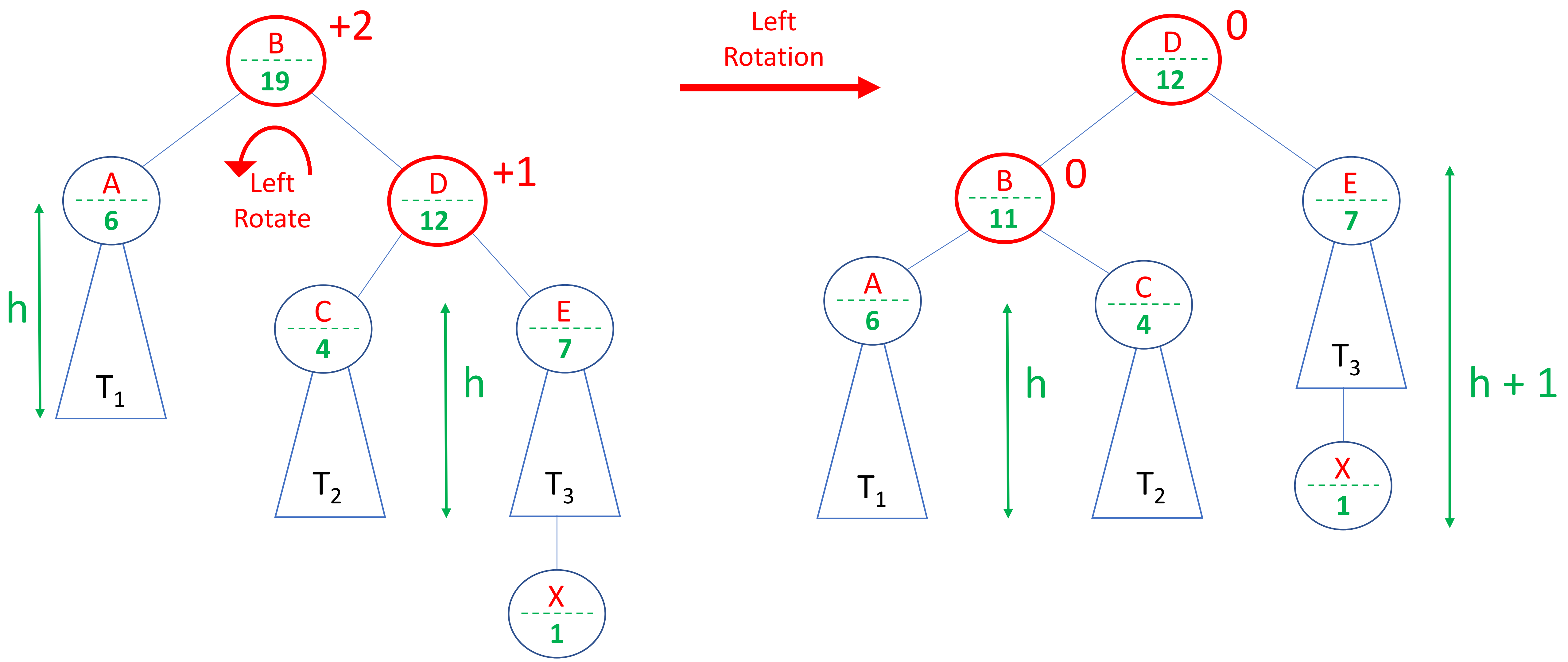
Maintaining size() field: Insert Example



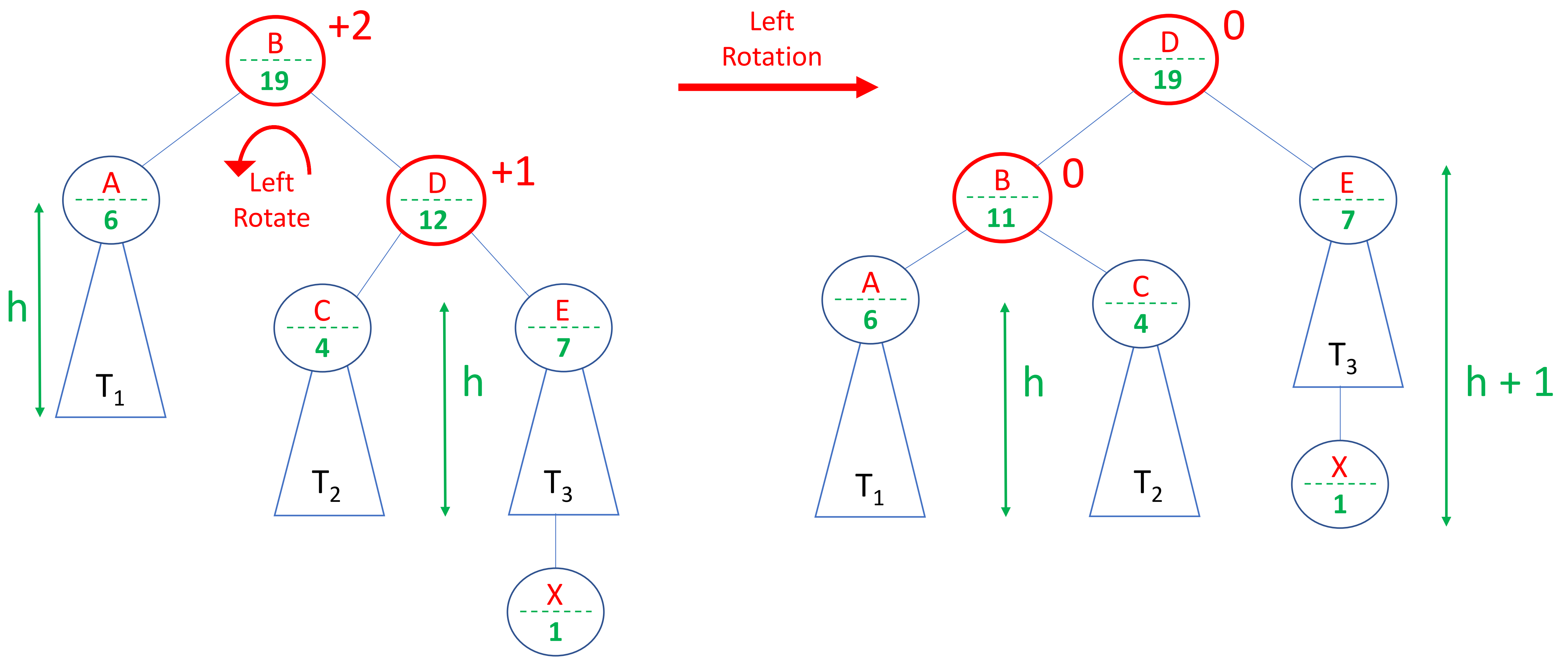
Maintaining size() field: Insert Example



Maintaining size() field: Insert Example



Maintaining size() field: Insert Example

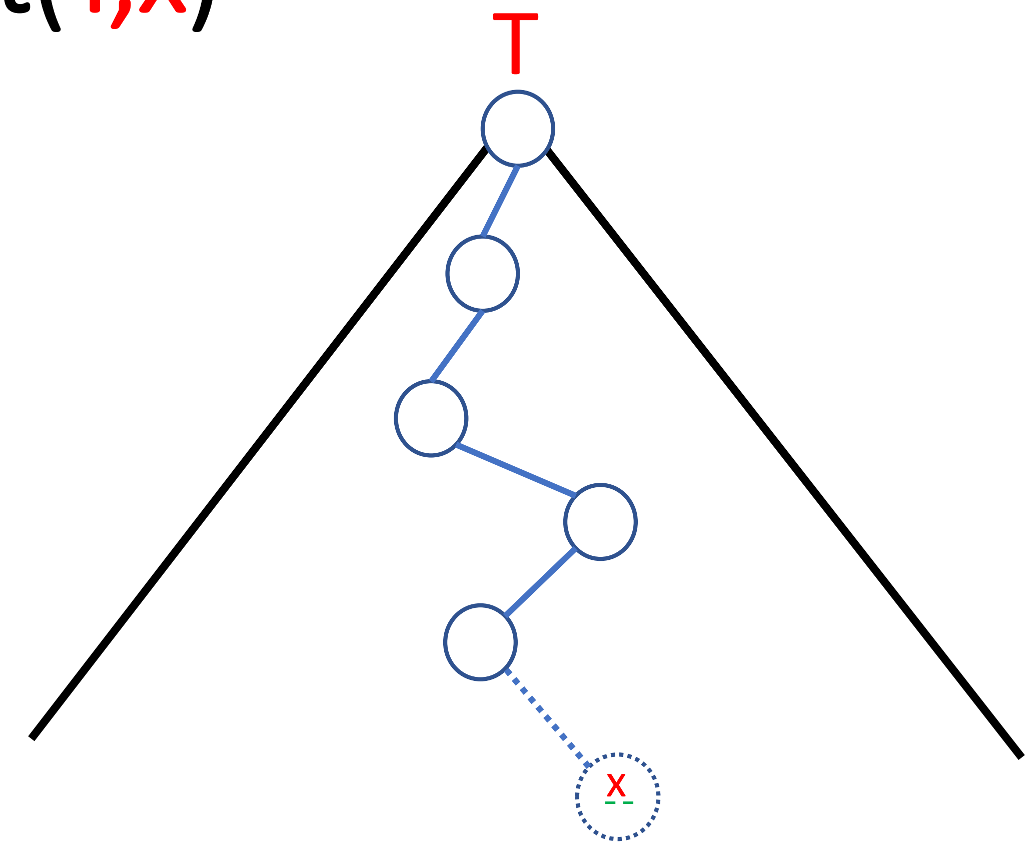


Maintaining size() field: Insert(T,x)

Maintaining size() field: Insert(T,x)

- Insert x into T as in any BST :
    - x is now a leaf

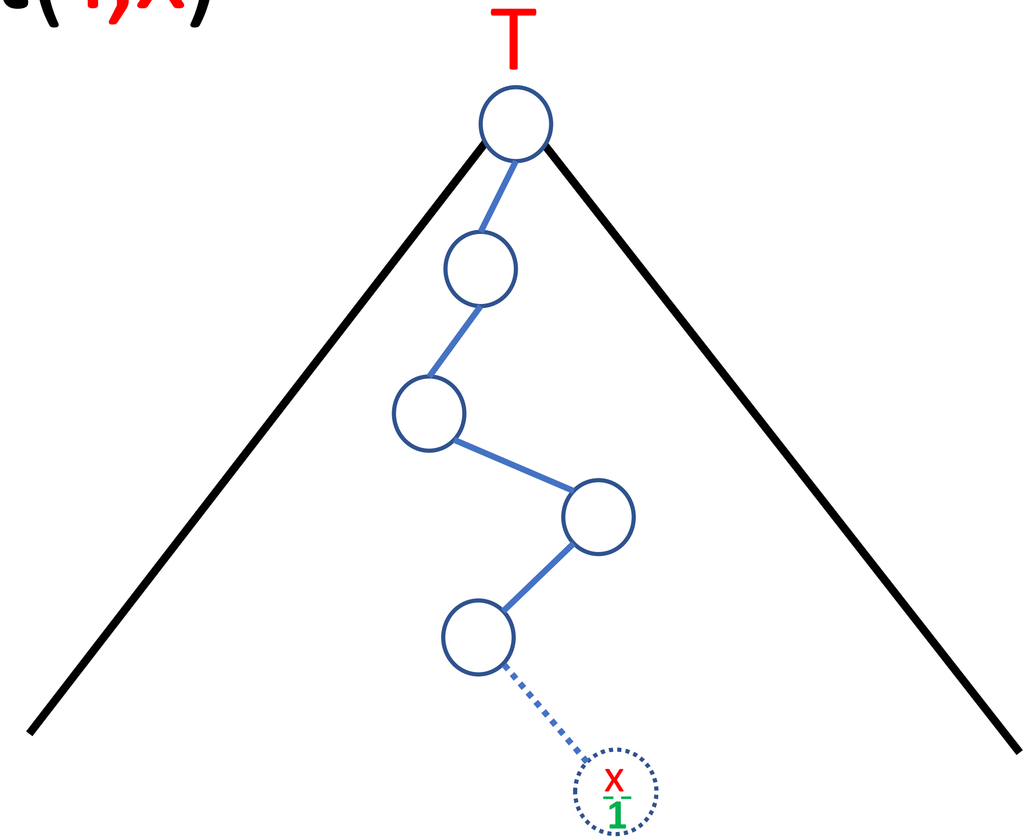
Phase 1



Maintaining size() field: Insert(T,x)

- Insert x into T as in any BST :
  - x is now a leaf
  - Set size(x) = 1

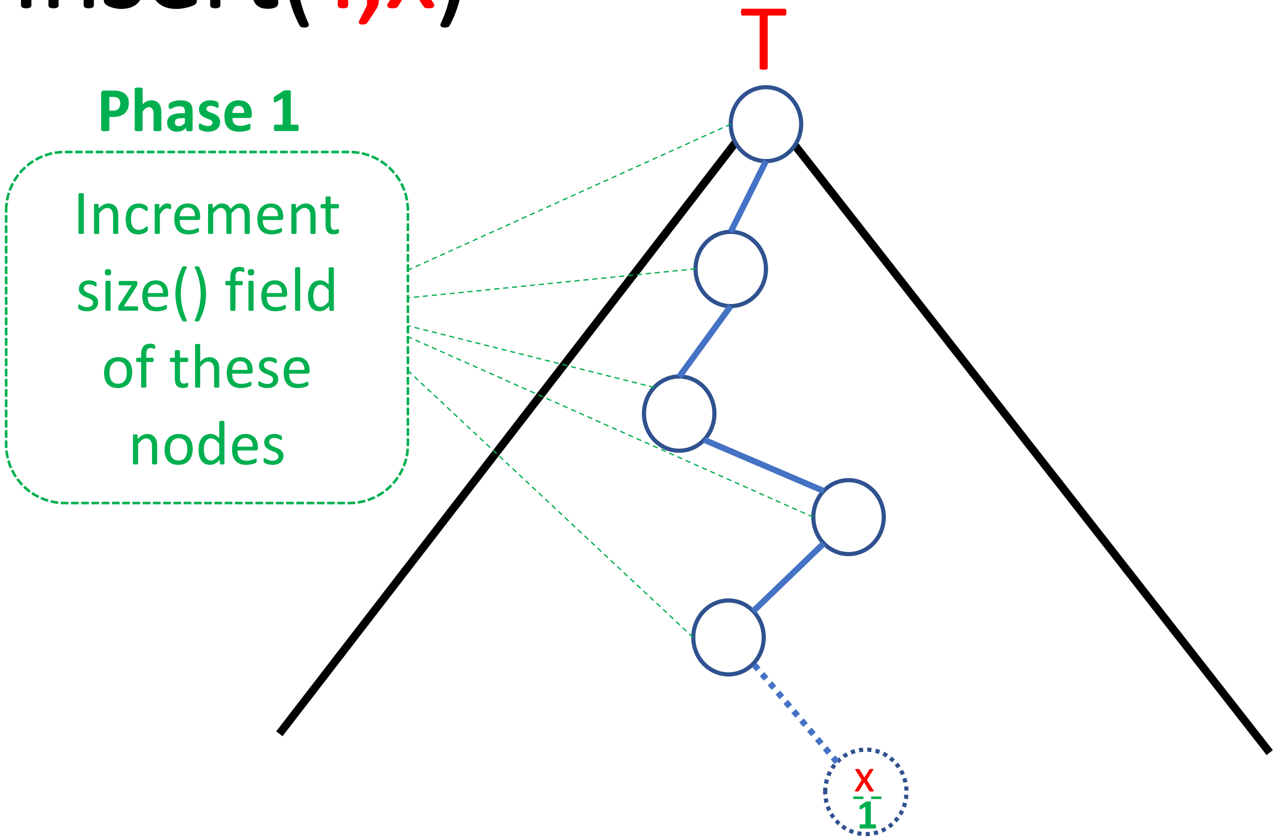
Phase 1



Maintaining size() field: Insert(T,x)

- Insert x into T as in any BST :
  - x is now a leaf
  - Set size(x) = 1

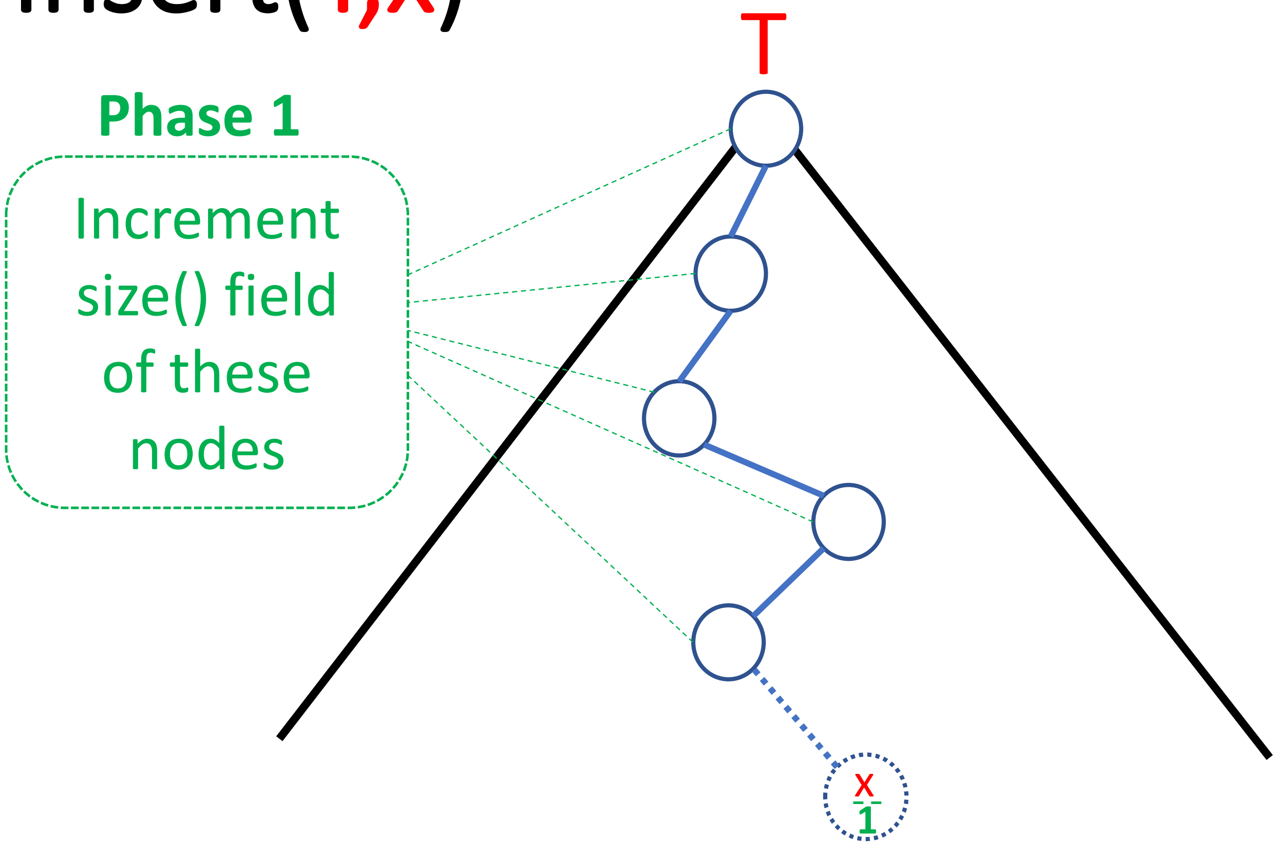
Phase 1



Maintaining size() field: Insert(T,x)

Phase 1

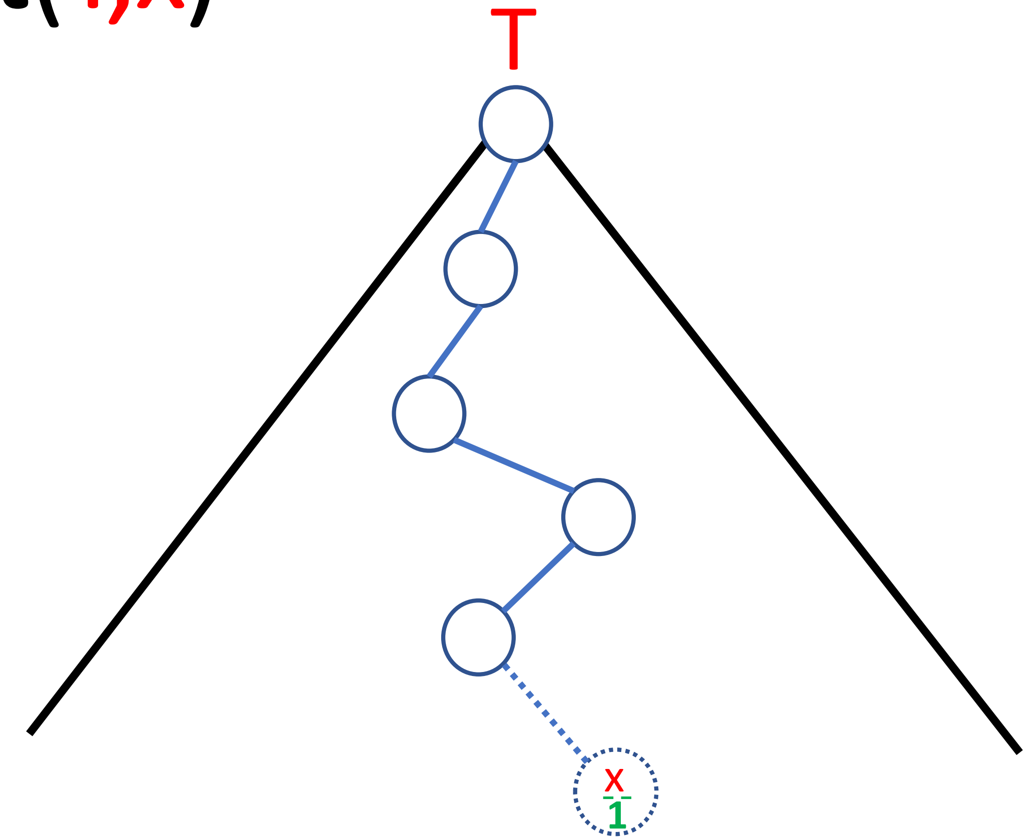
- Insert x into T as in any BST :
  - x is now a leaf
  - Set size(x) = 1
  - For each node y on path from x to root
    - Increment size(y)



Maintaining size() field: Insert(T,x)

- Insert x into T as in any BST :
  - x is now a leaf
  - Set size(x) = 1
  - For each node y on path from x to root
    - Increment size(y)

Phase 1

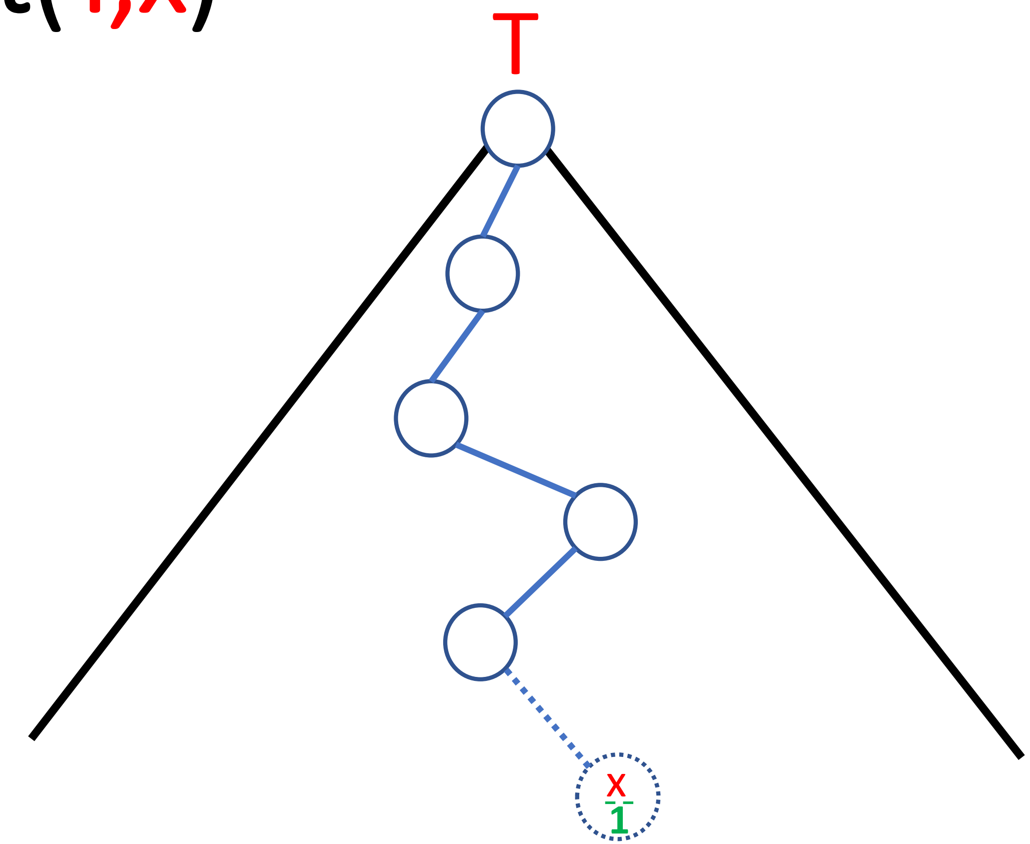


Maintaining size() field: Insert(T,x)

- Insert x into T as in any BST :
  - x is now a leaf
  - Set size(x) = 1
  - For each node y on path from x to root
    - Increment size(y)
- Go up from x to the root and for each node :
  - Adjust the BF
  - Rebalance with rotation if needed

Phase 1

Phase 2



Maintaining size() field: Insert(T,x)

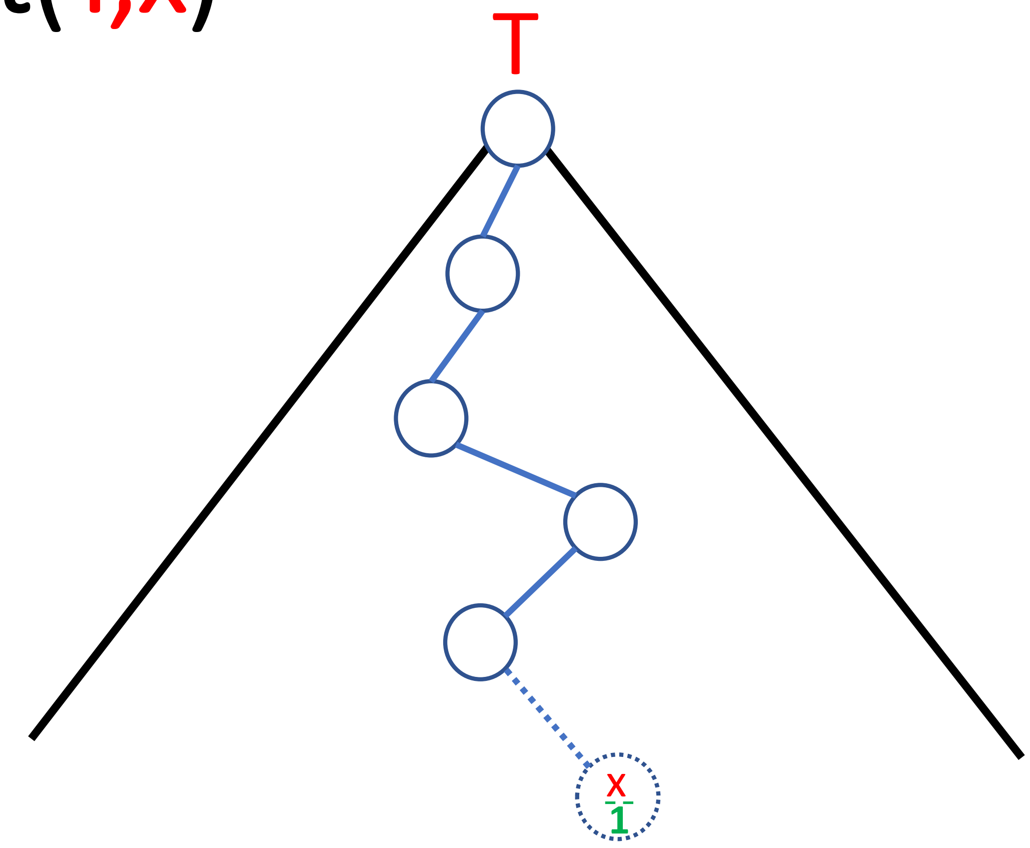
- Insert x into T as in any BST :
  - x is now a leaf
  - Set size(x) = 1
  - For each node y on path from x to root
    - Increment size(y)

Phase 1

- Go up from x to the root and for each node :
  - Adjust the BF
  - Rebalance with rotation if needed
  - If rotation is needed, update size() where necessary using the invariant:

$$\mathbf{size(z) = size(left(z)) + size(right(z)) + 1}$$

Phase 2



Maintaining size() field: Insert(T,x)

- Insert x into T as in any BST :
  - x is now a leaf
  - Set size(x) = 1
  - For each node y on path from x to root
    - Increment size(y)

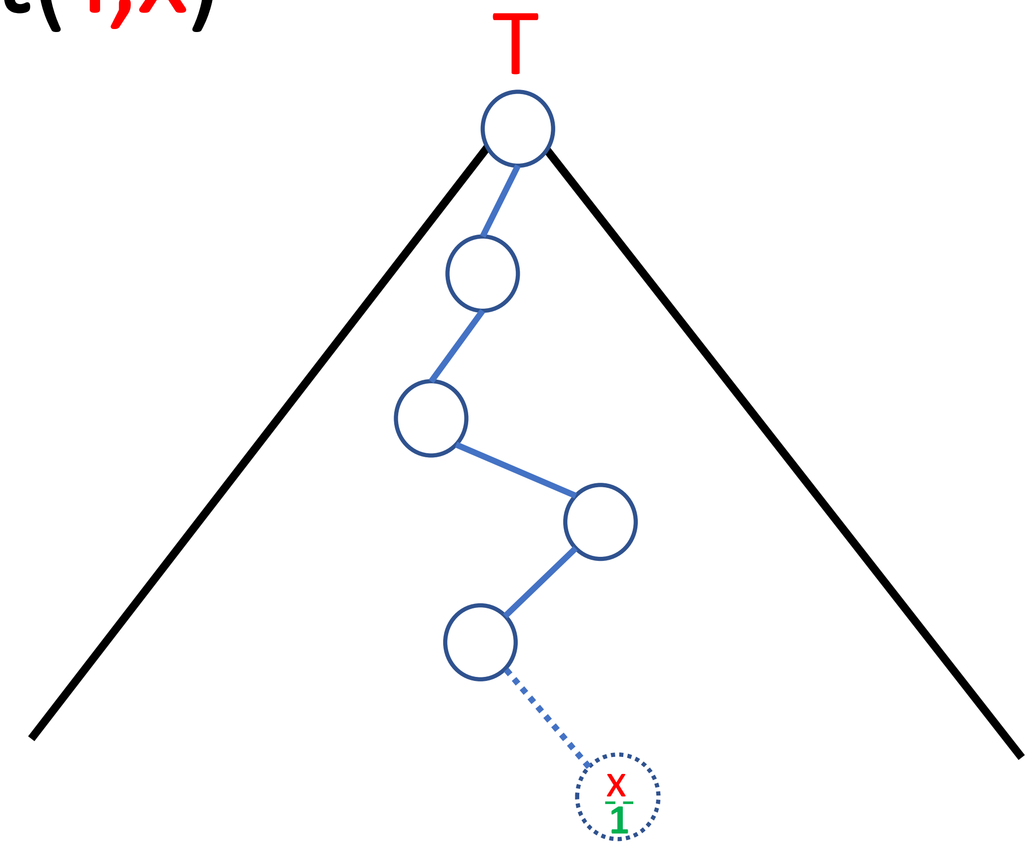
Phase 1

- Go up from x to the root and for each node :
  - Adjust the BF
  - Rebalance with rotation if needed
  - If rotation is needed, update size() where necessary using the invariant:

size(z) = size(left(z)) + size(right(z)) + 1

Phase 2

This adds constant work for each rotation



Augmenting AVL

- **Select** operation ✓
- **Rank** operation ✓
- Maintain size() field after **Insert or Delete** ✓