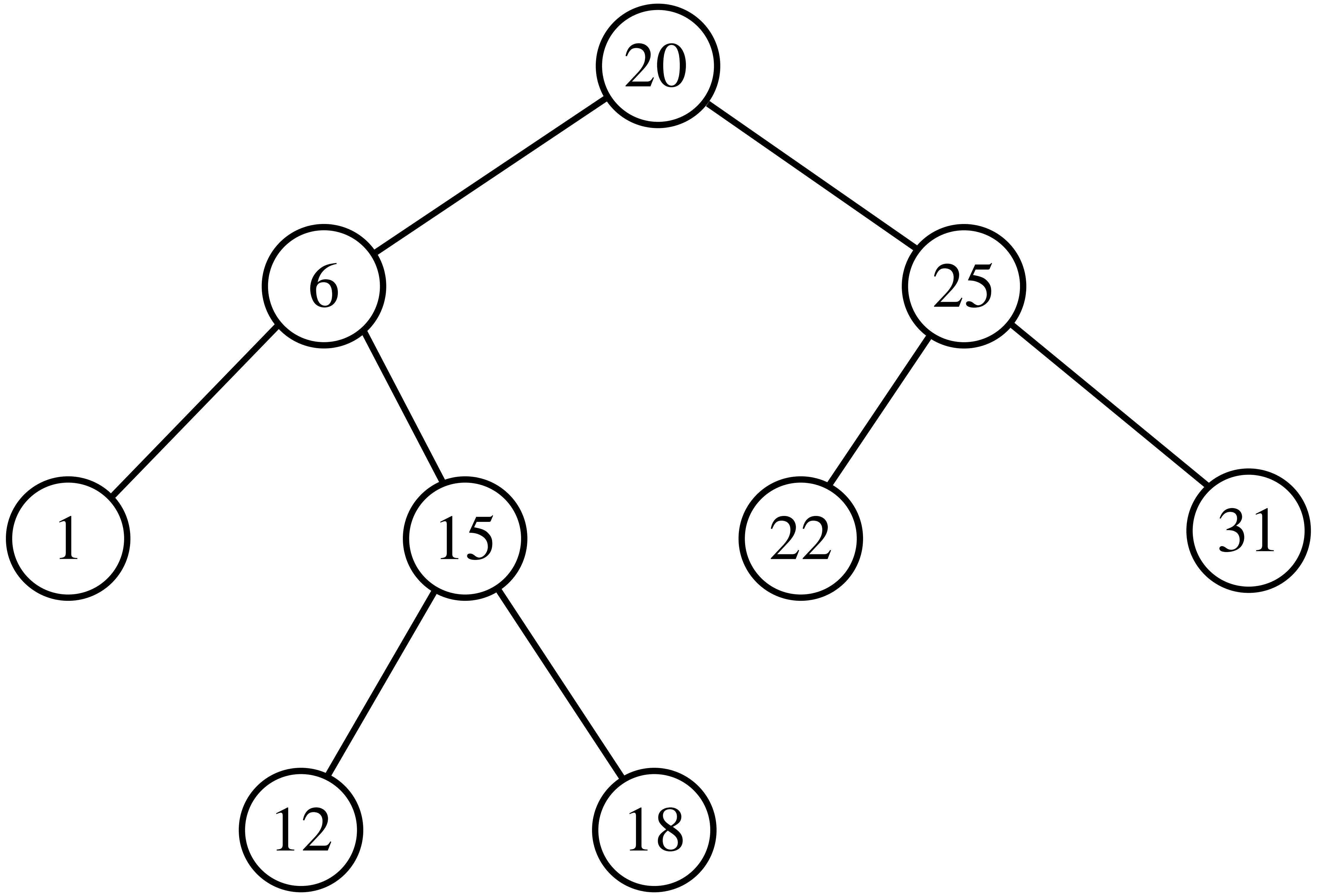
Lecture 4

BST: Insertion & Deletion, Intro to Red-Black Trees

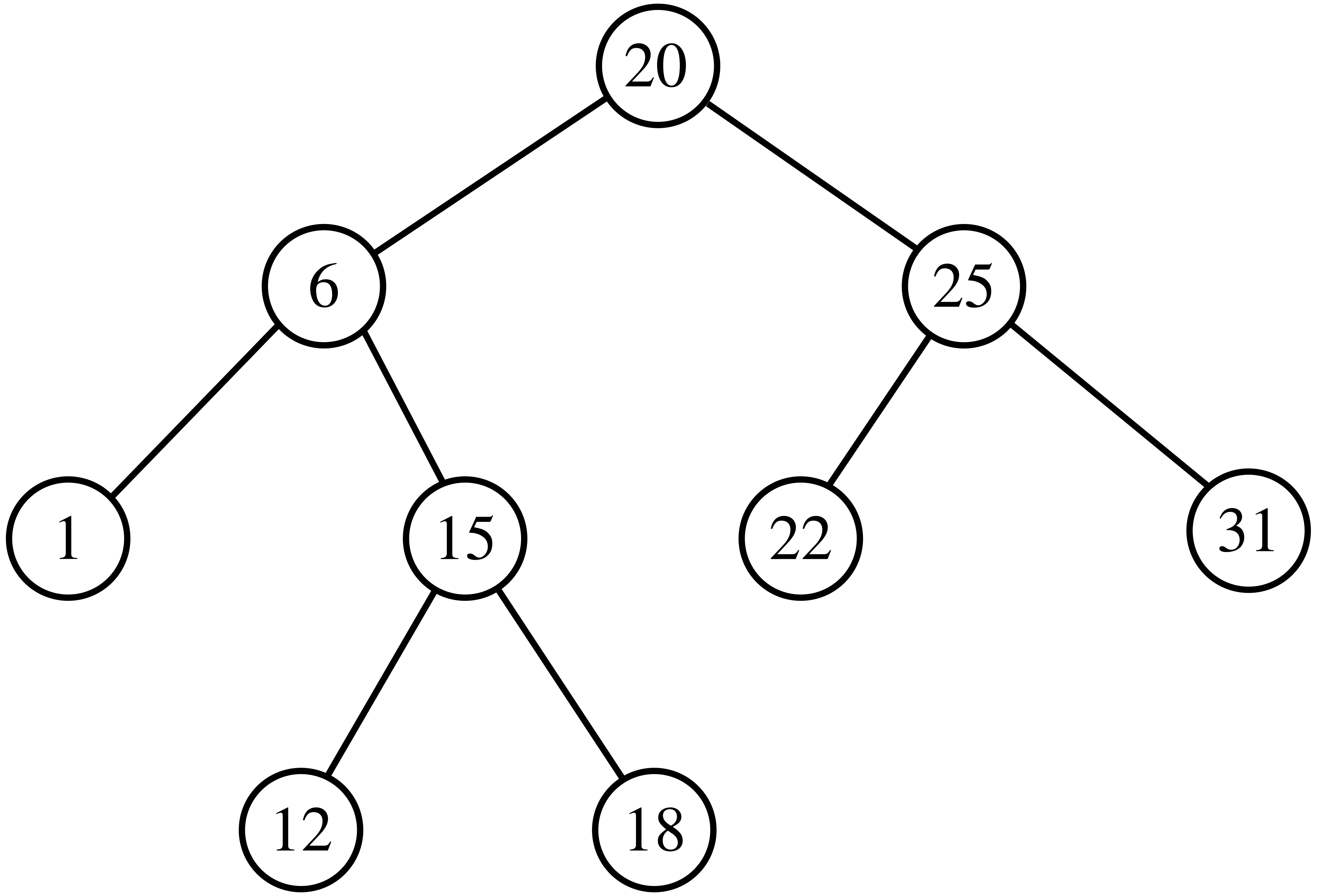
Insertion in a BST

Insertion in a BST



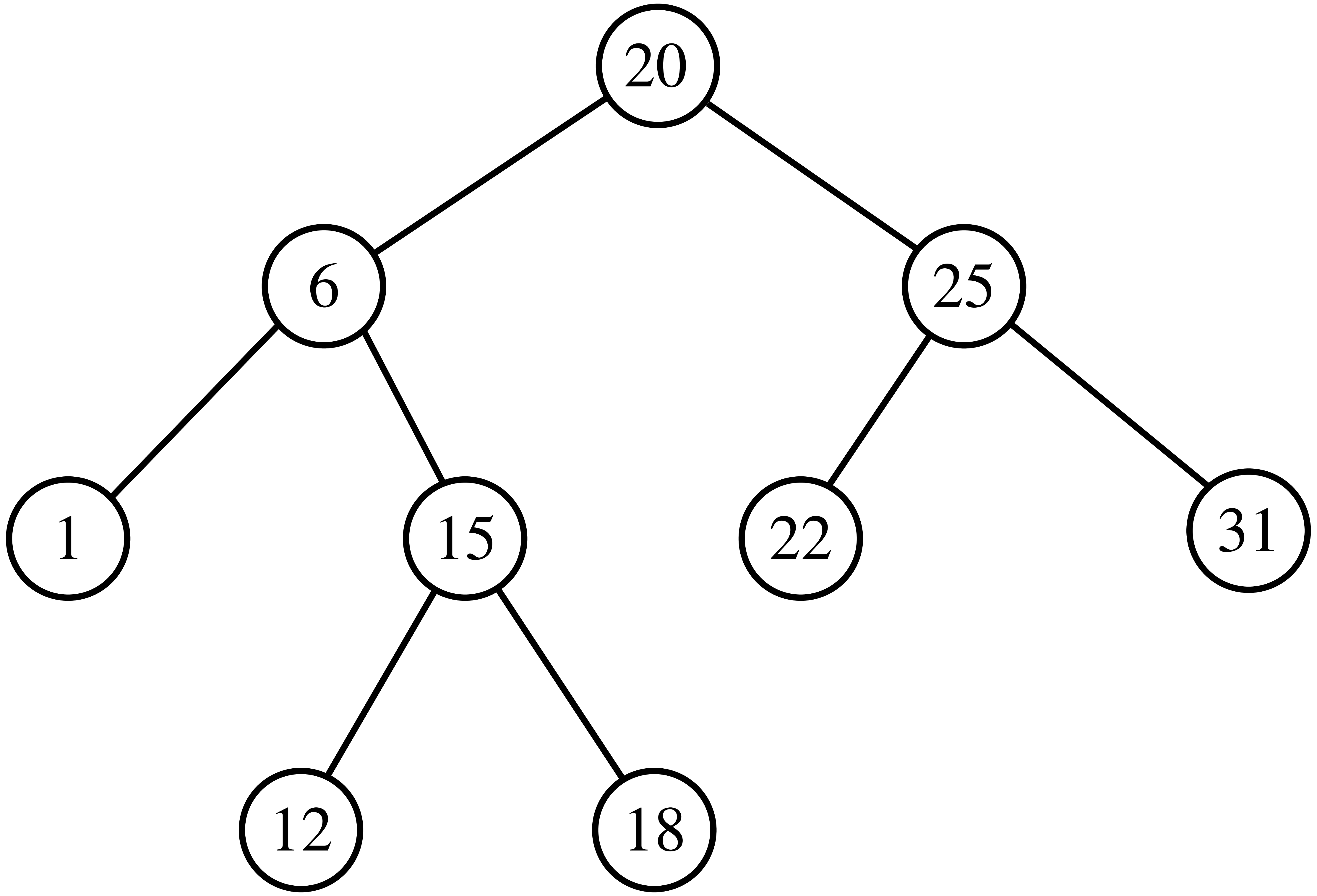
Insertion in a BST

Example:



Insertion in a BST

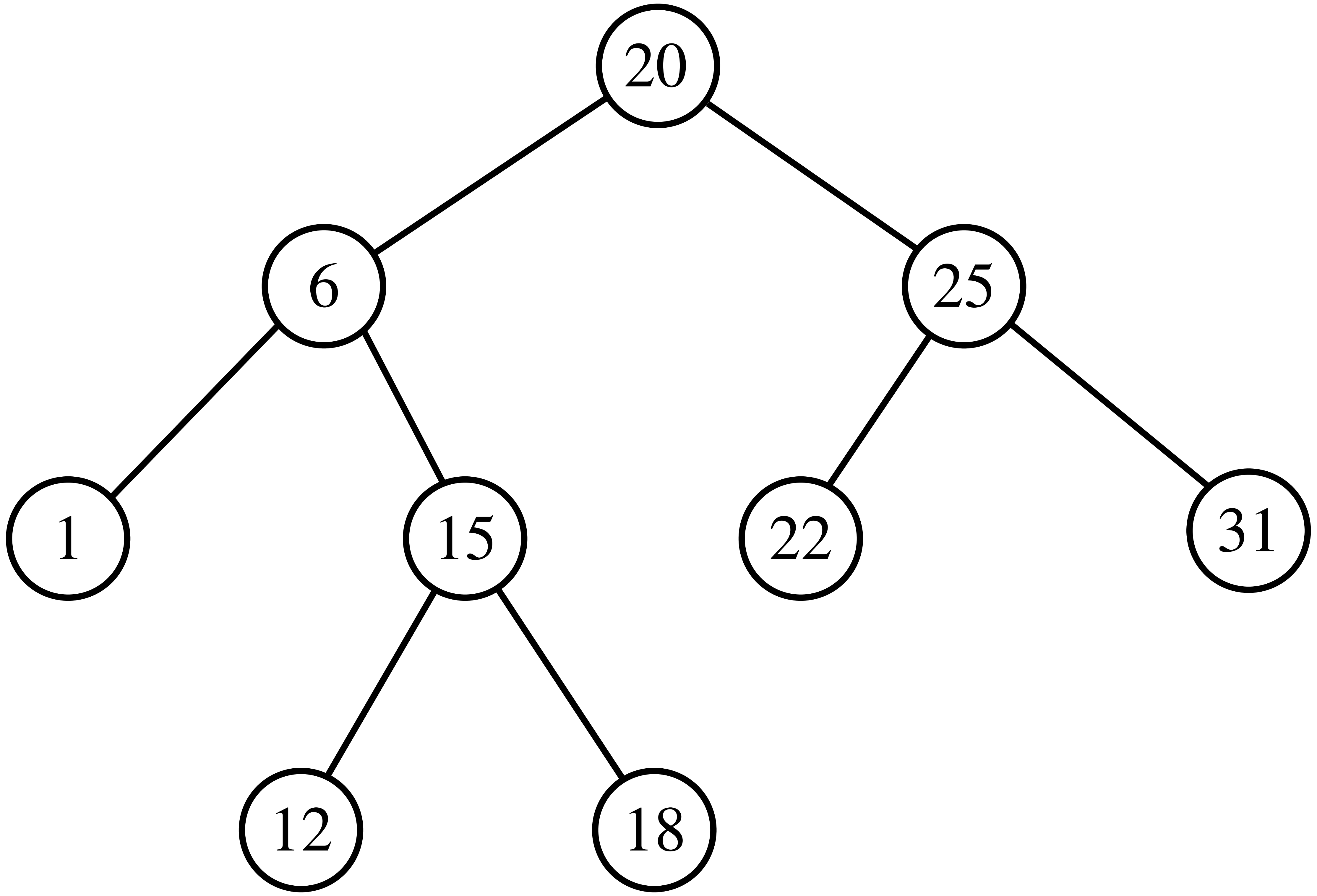
Example: Insert a node with 24 as key in the following BST.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

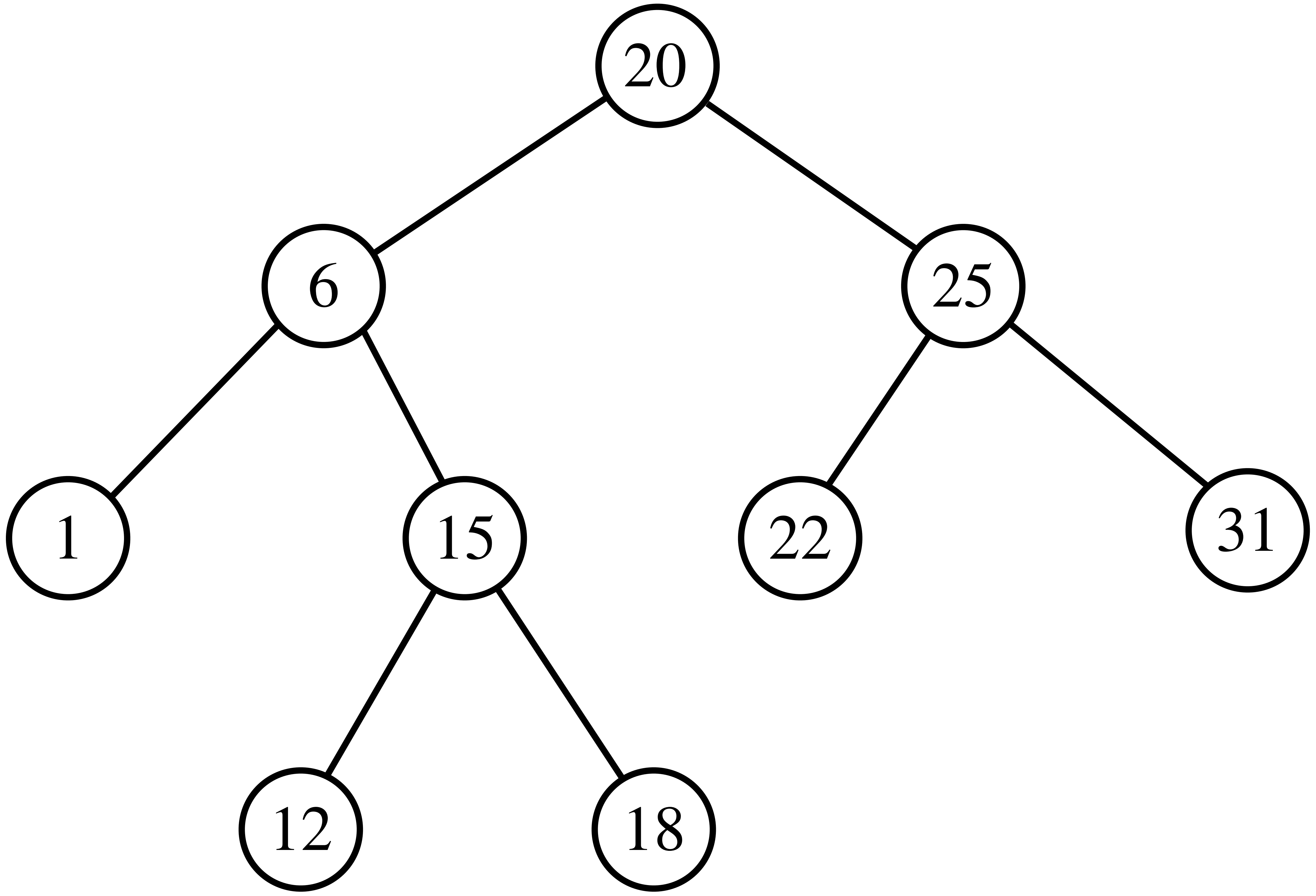
Idea:



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

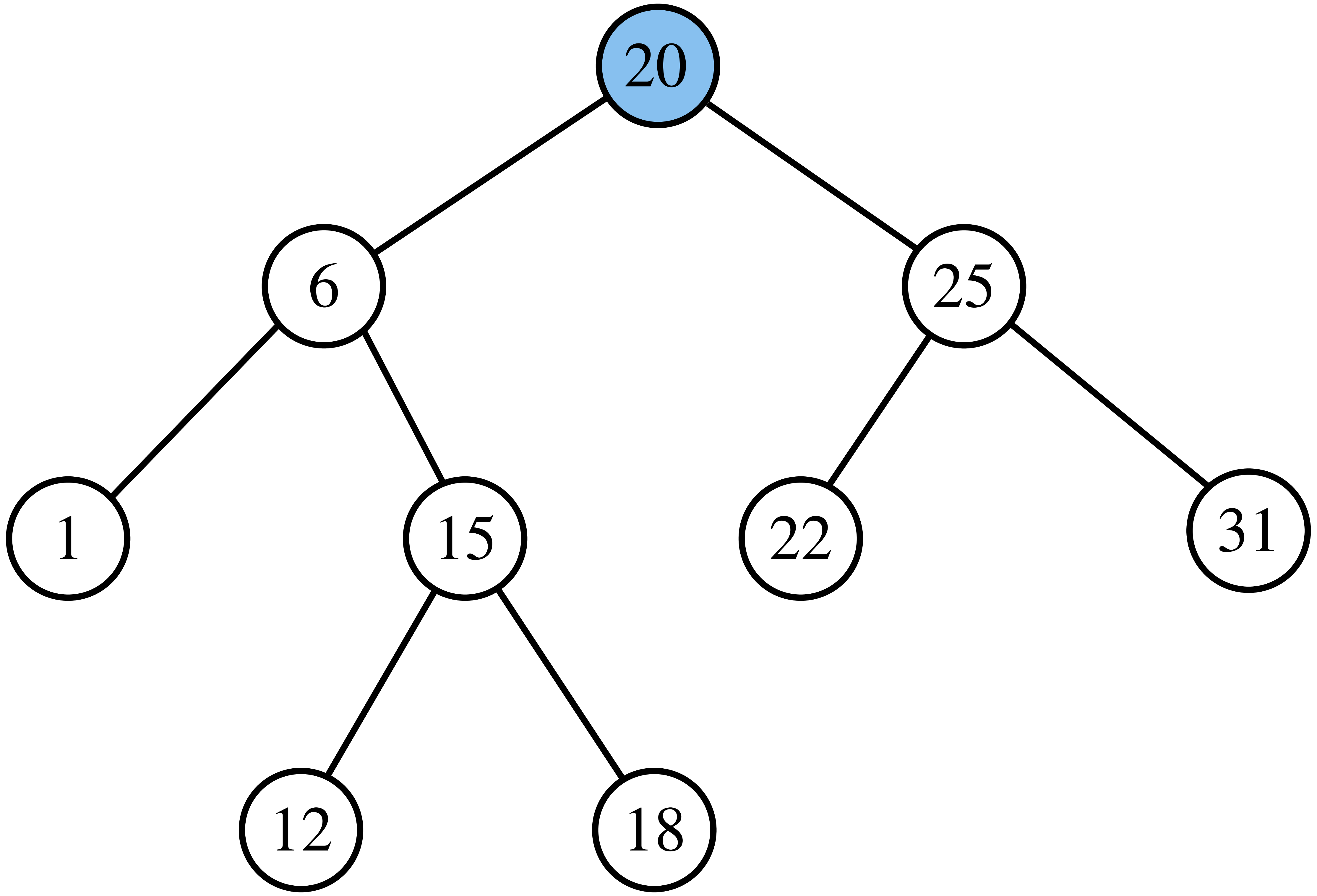
Idea: Find the correct leaf where it can be inserted.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

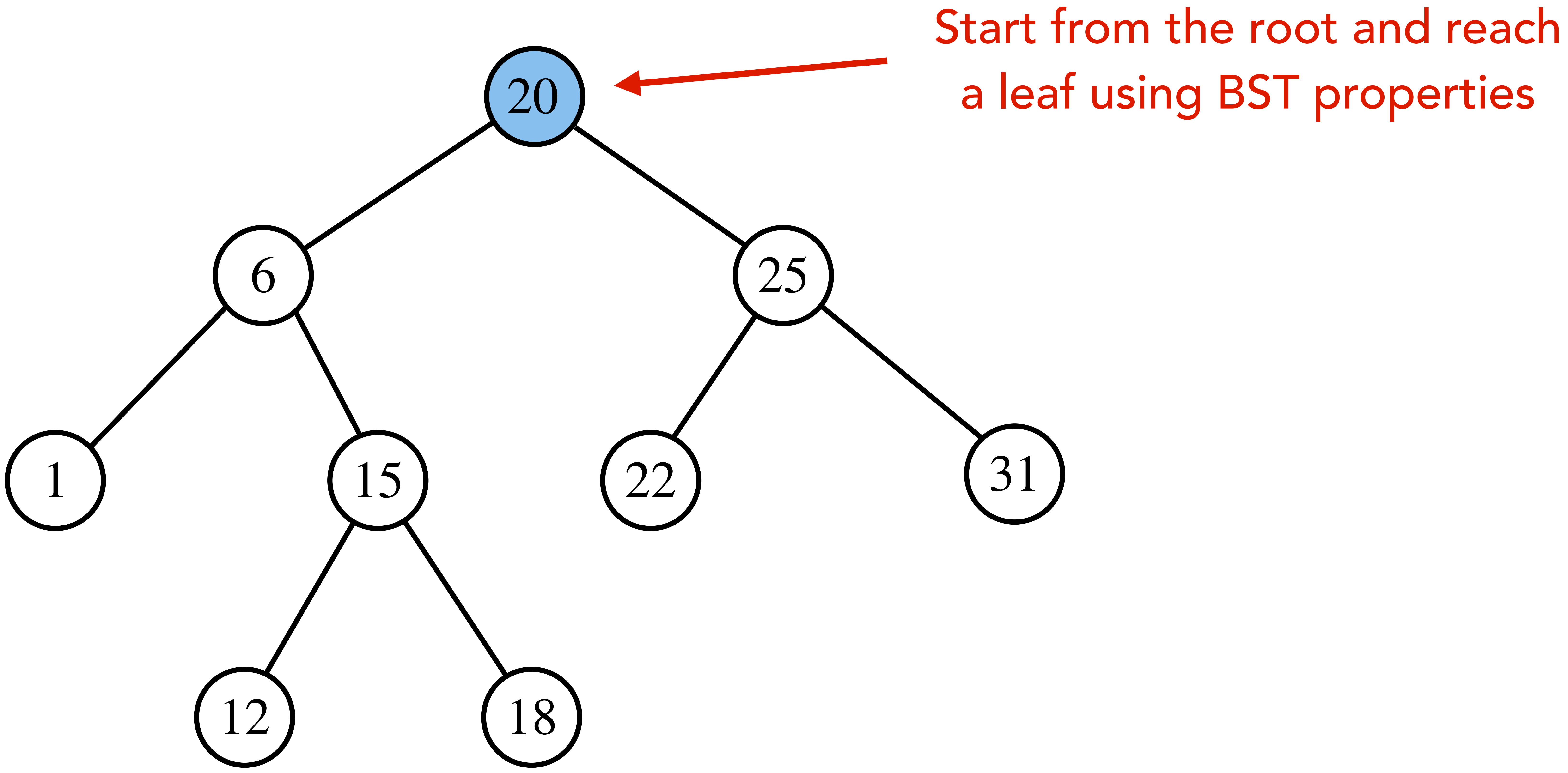
Idea: Find the correct leaf where it can be inserted.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

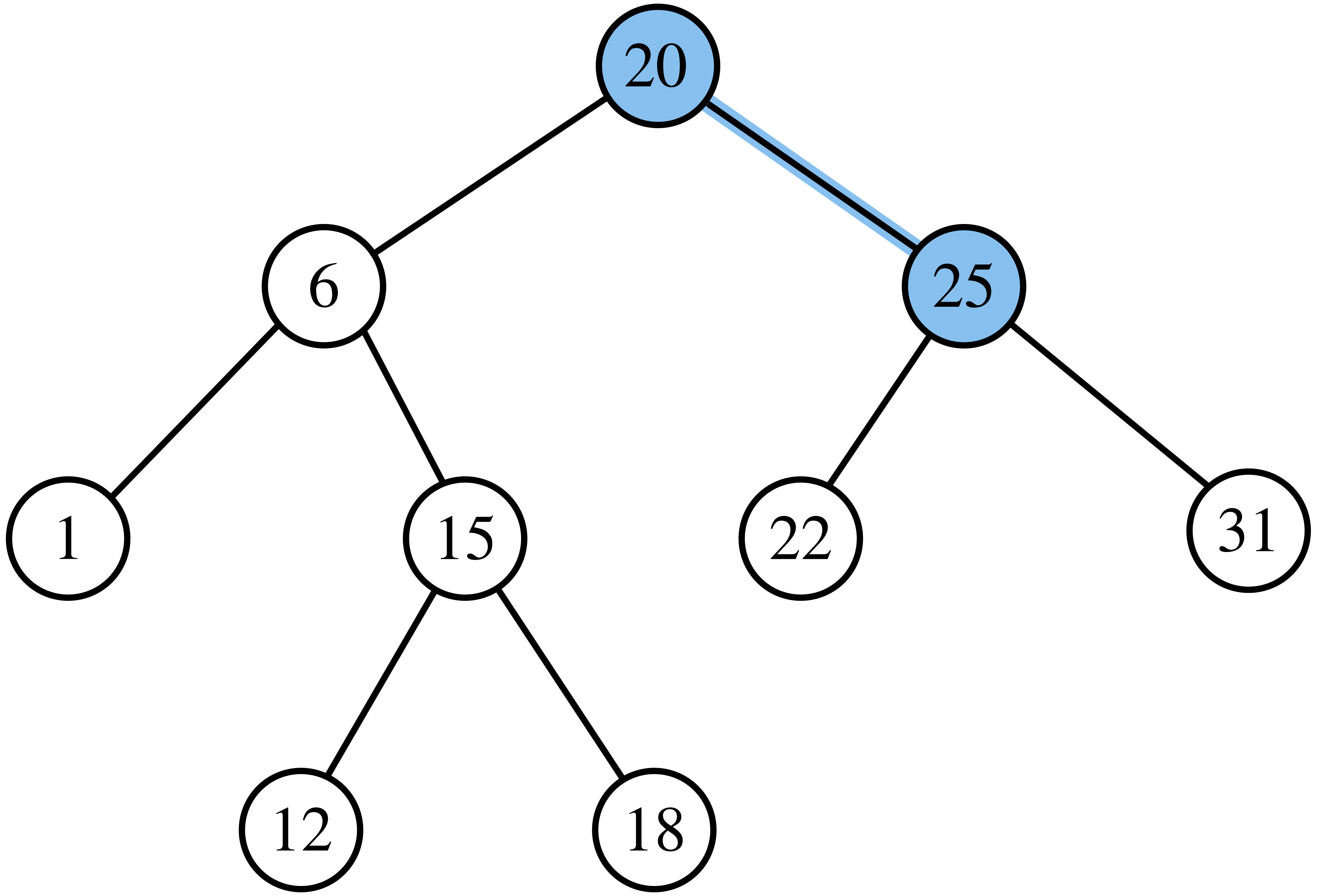
Idea: Find the correct leaf where it can be inserted.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

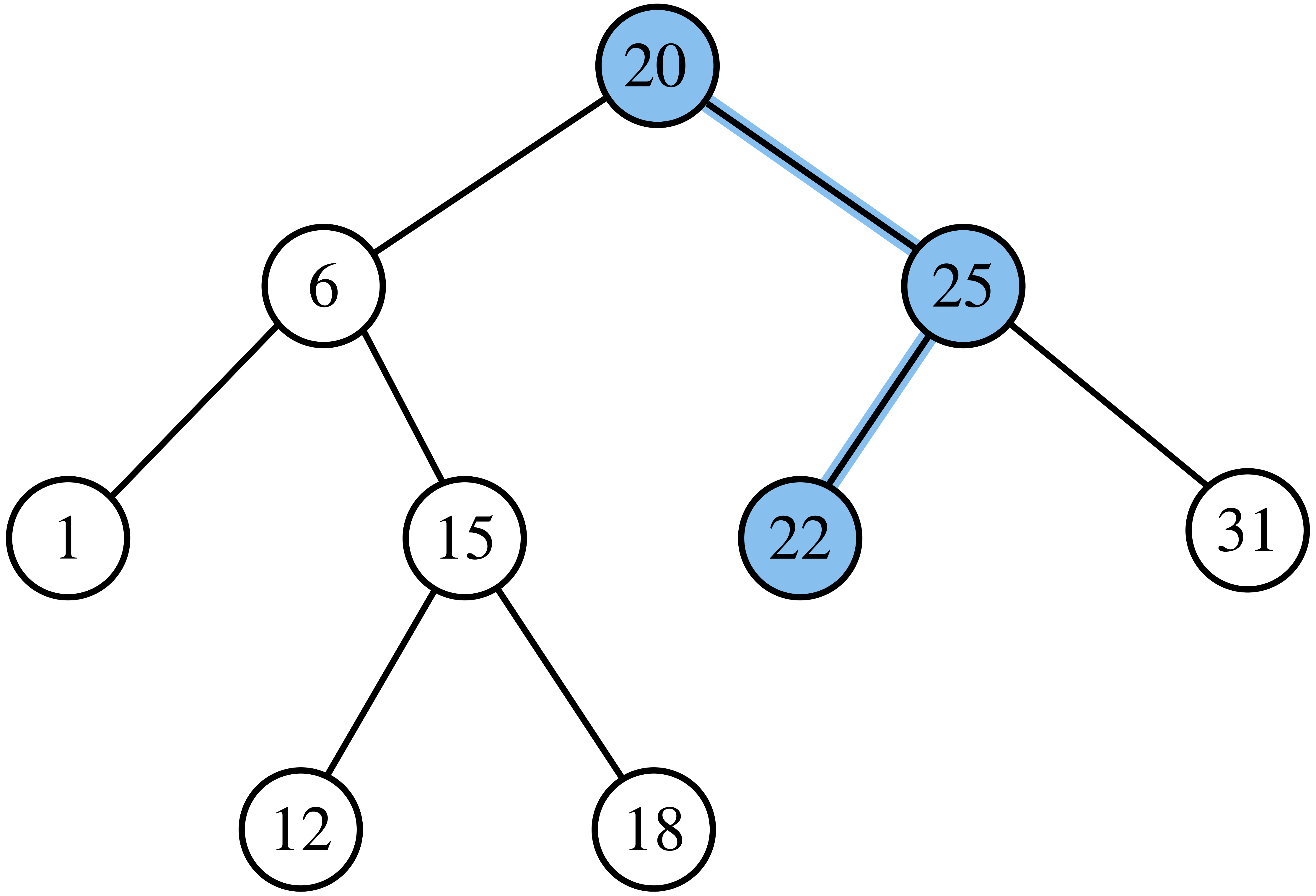
Idea: Find the correct leaf where it can be inserted.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

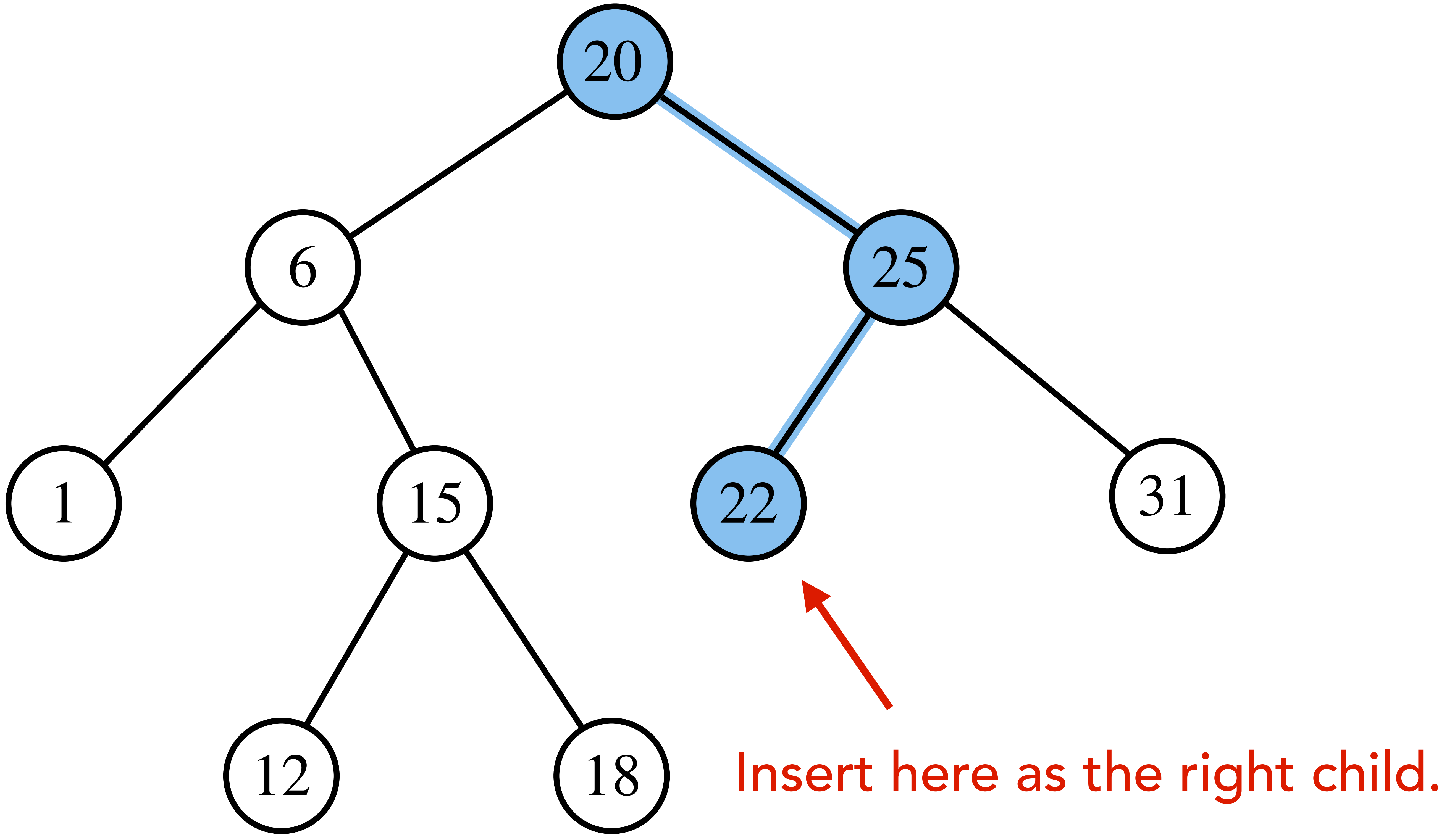
Idea: Find the correct leaf where it can be inserted.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

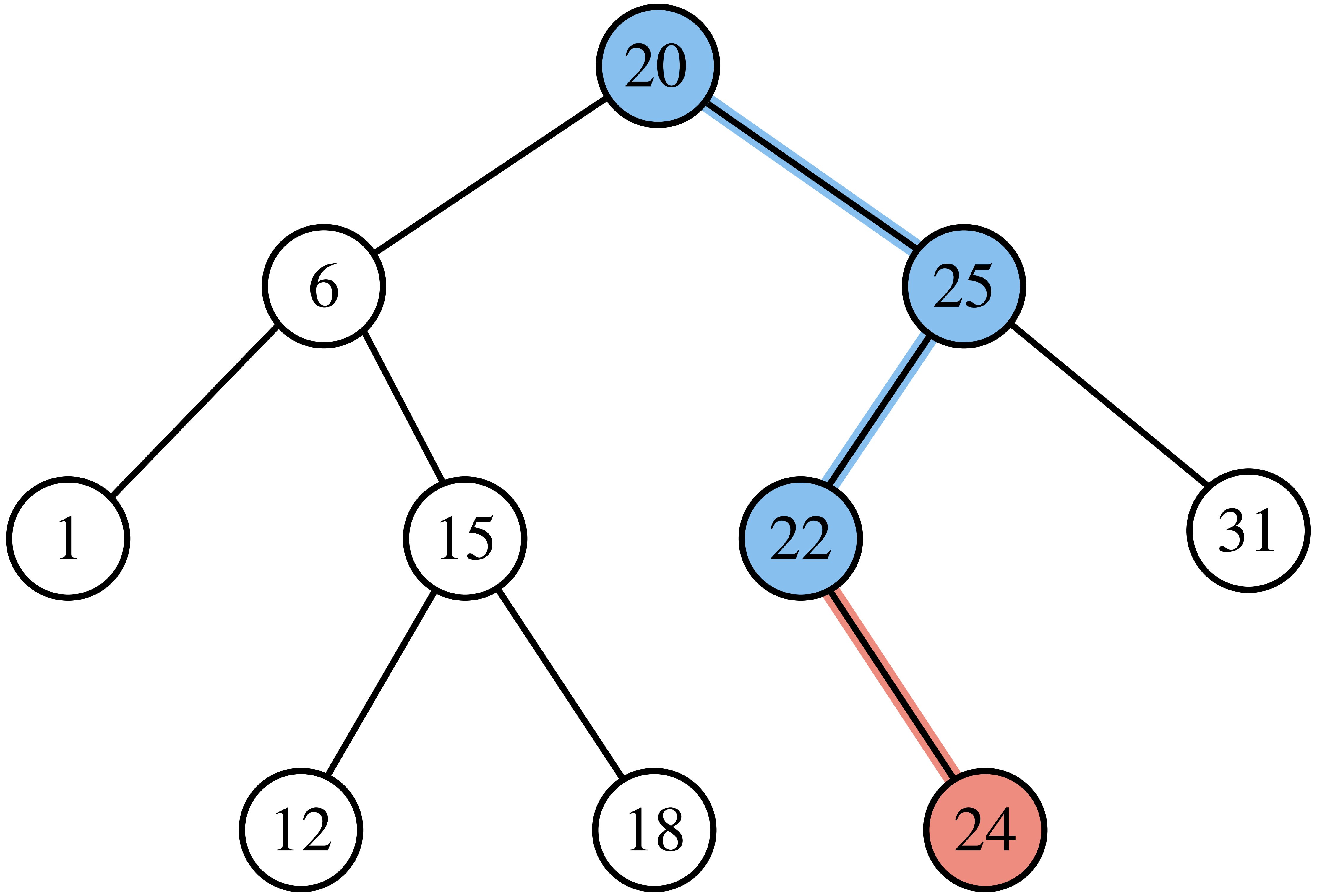
Idea: Find the correct leaf where it can be inserted.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

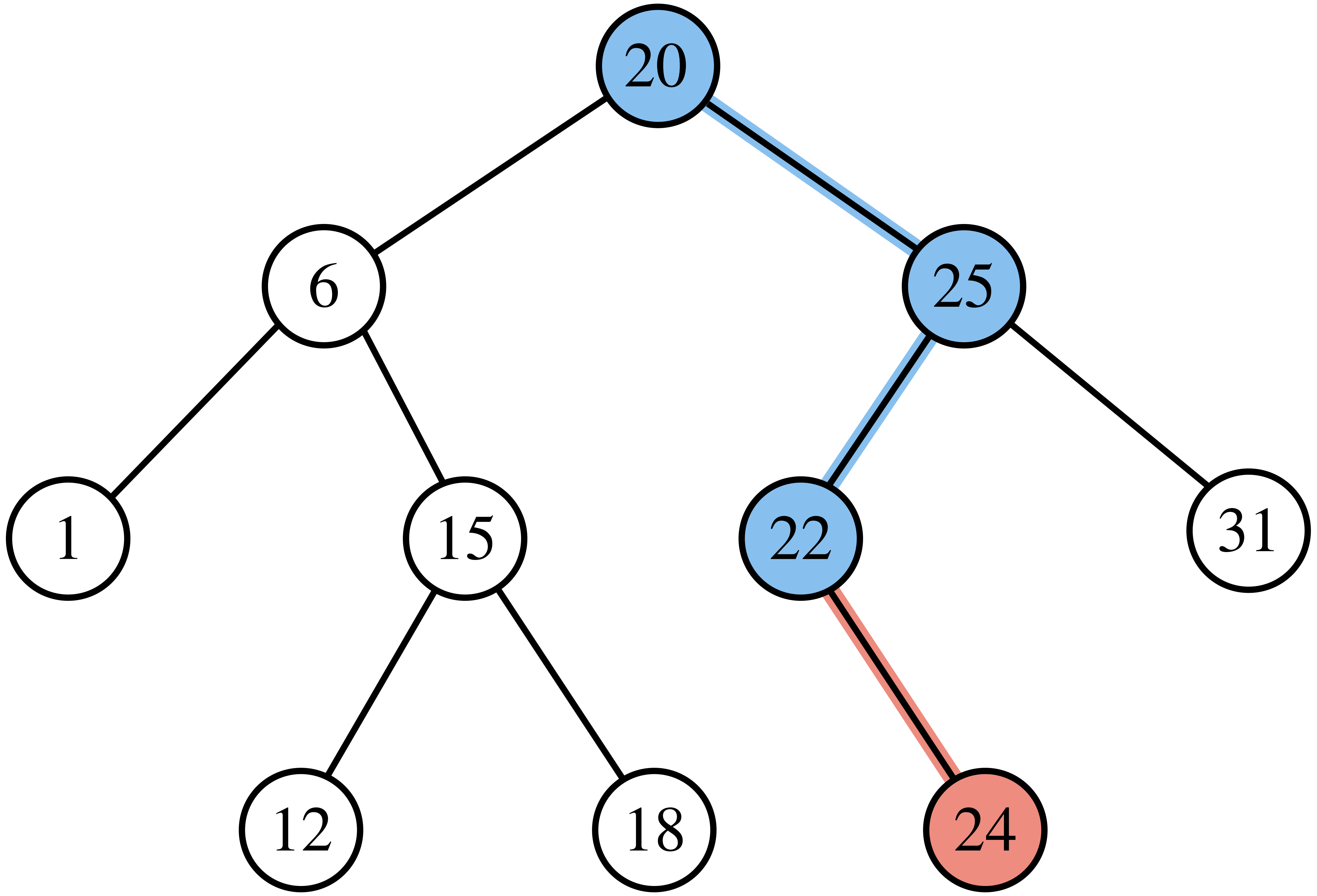
Idea: Find the correct leaf where it can be inserted.



Insertion in a BST

Example: Insert a node with 24 as key in the following BST.

Idea: Find the correct leaf where it can be inserted.



Searching for the correct leaf and insertion takes $\Theta(h)$ time.

Deletion in a BST

Deletion in a BST

Deletion can be more tricky than Insertion.

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete.

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete. Then, the following cases are possible:

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.
- **Case** 3: z has two children.

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.
- **Case** 3: z has two children.

Easy

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.
- **Case** 3: z has two children.

Easy (Cases 1 and 2)

Less easy (Case 3)

Deletion in a BST

Deletion can be more tricky than Insertion.

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.
- **Case** 3: z has two children.

Easy (Cases 1 and 2)

Less easy (Case 3)

Note: Node z is provided as the input.

Deletion in a BST

Deletion in a BST

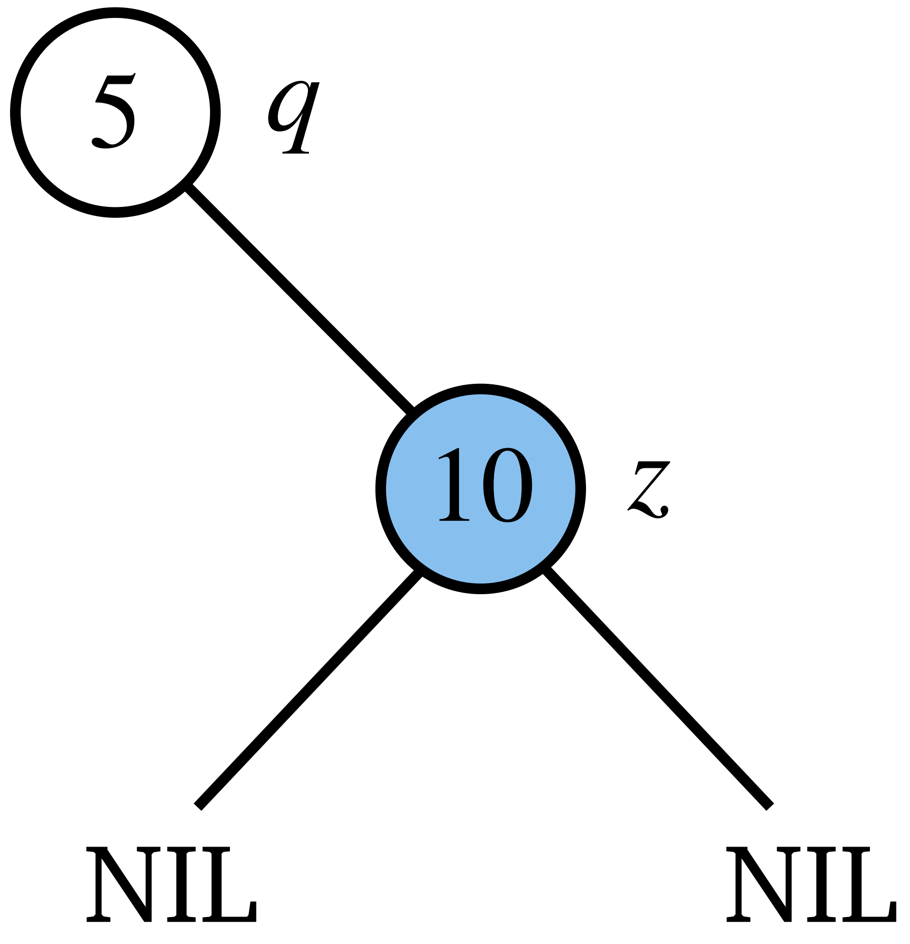
Case 1: z has no children.

Deletion in a BST

Case 1: z has no children. (WLOG assume z is a right child.)

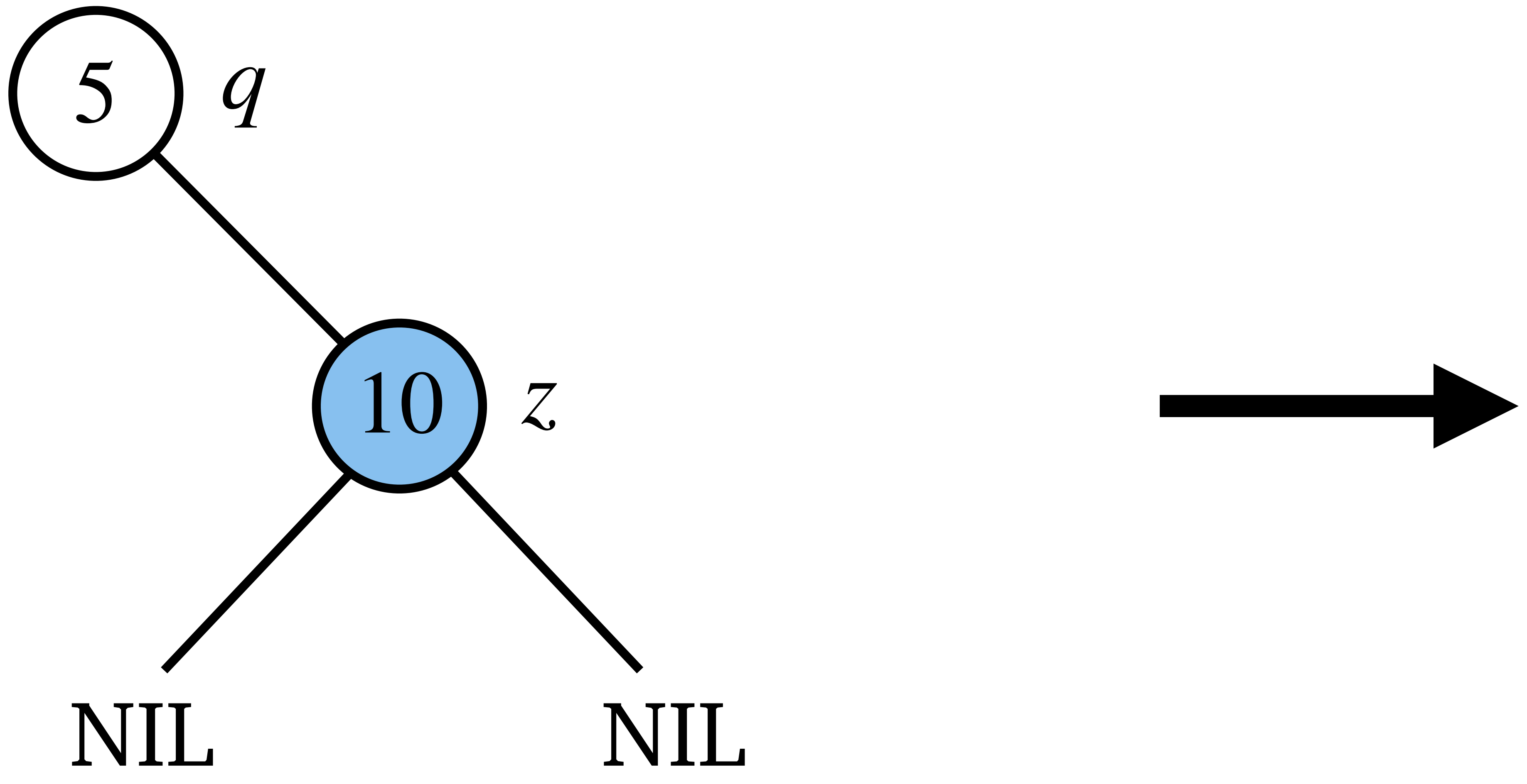
Deletion in a BST

Case 1: z has no children. (WLOG assume z is a right child.)



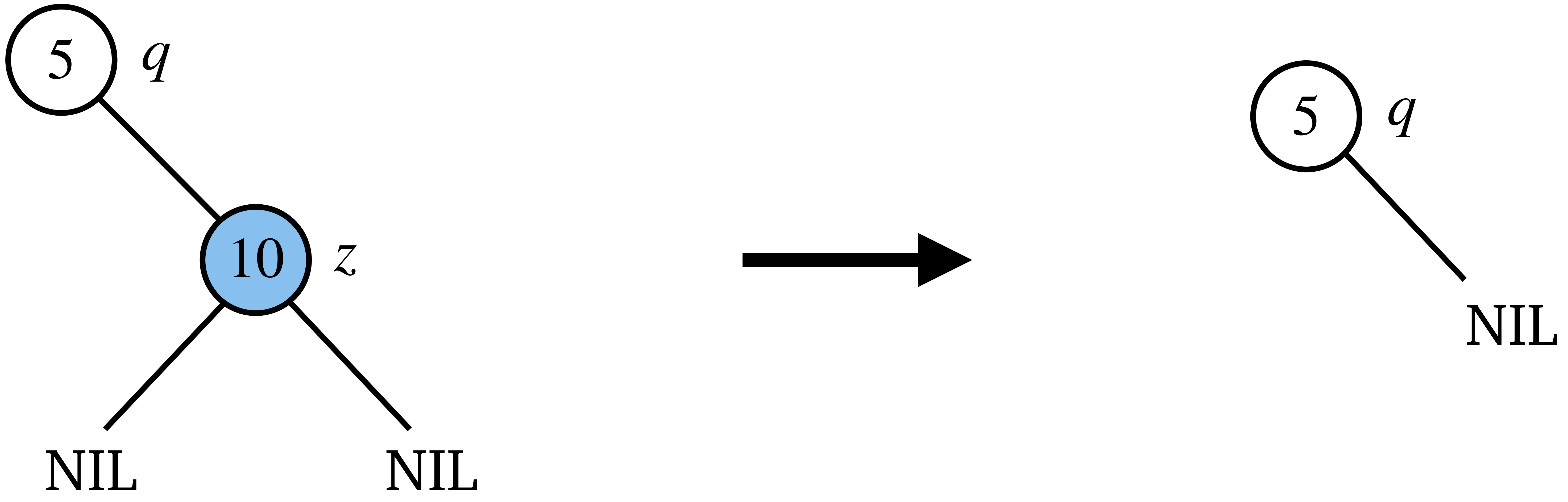
Deletion in a BST

Case 1: z has no children. (WLOG assume z is a right child.)



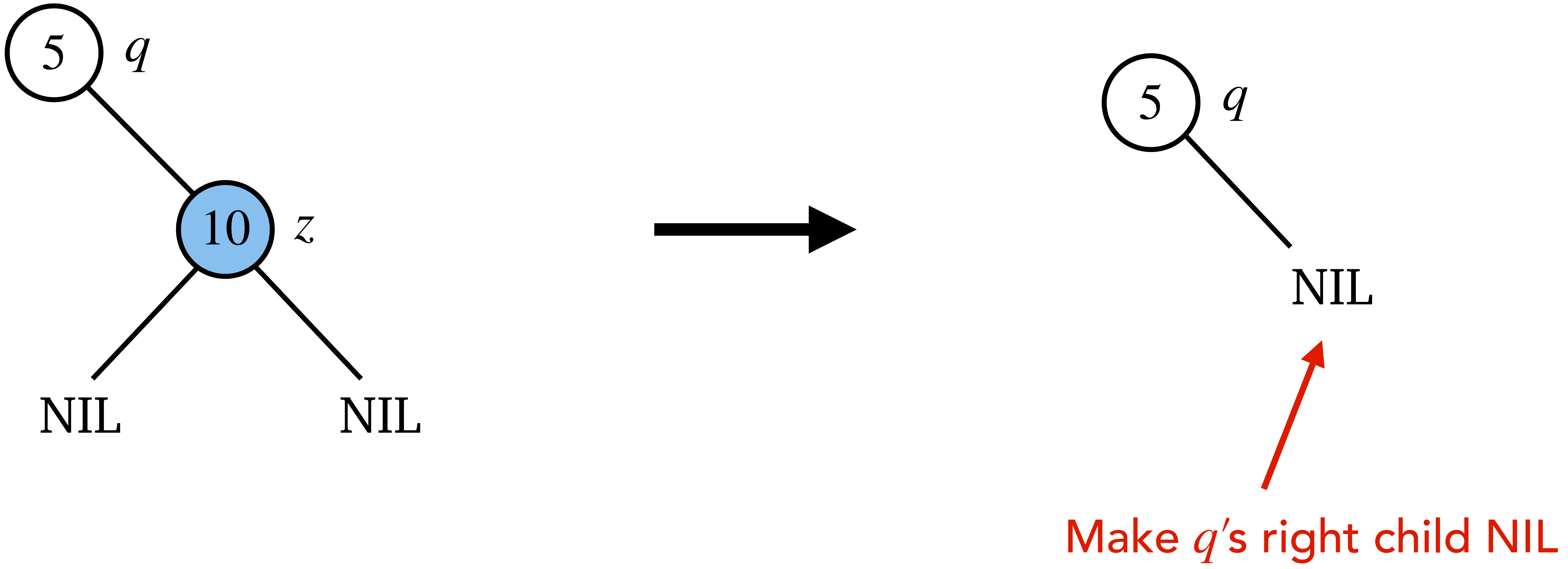
Deletion in a BST

Case 1: z has no children. (WLOG assume z is a right child.)



Deletion in a BST

Case 1: z has no children. (WLOG assume z is a right child.)



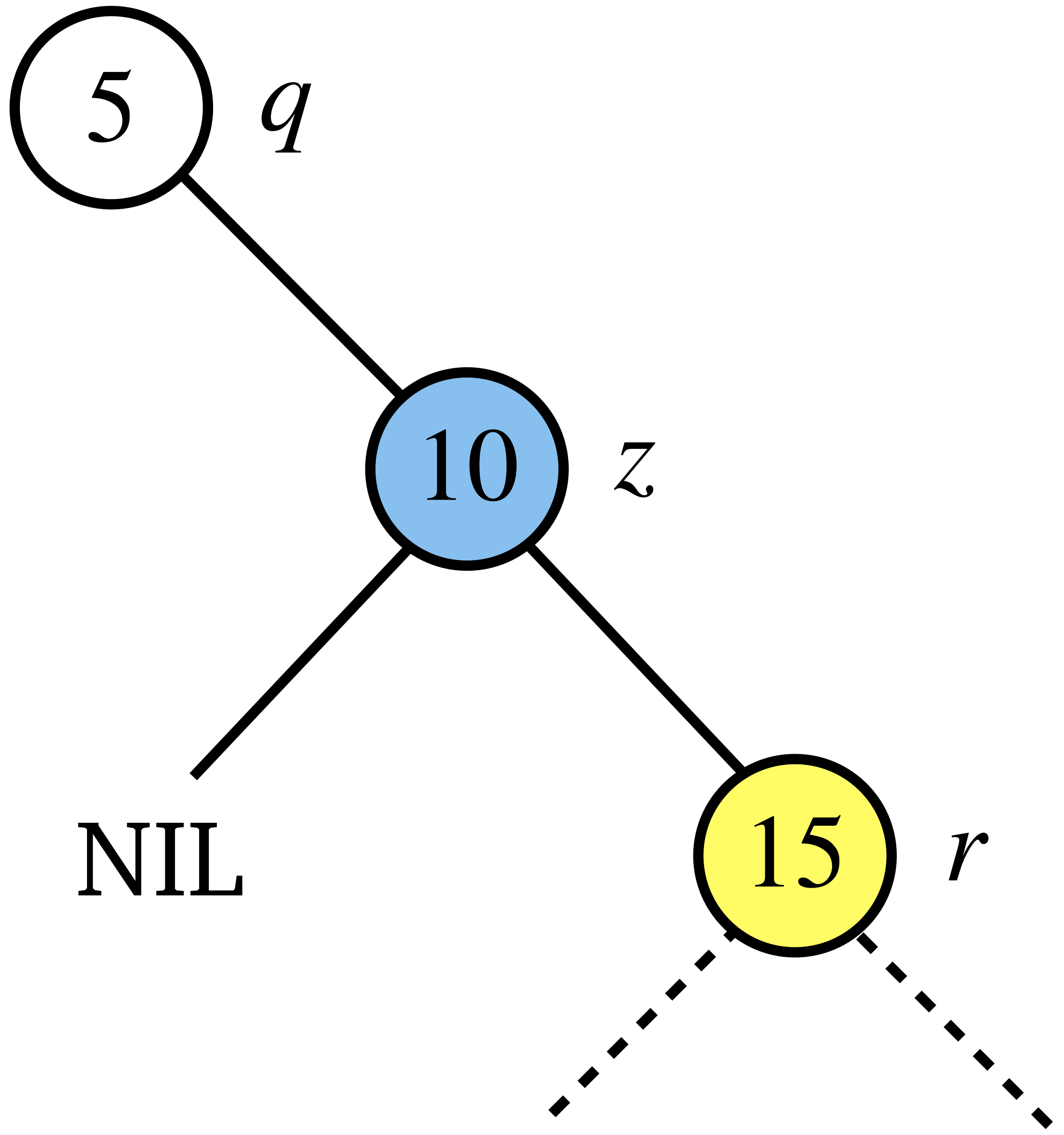
Deletion in a BST

Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)

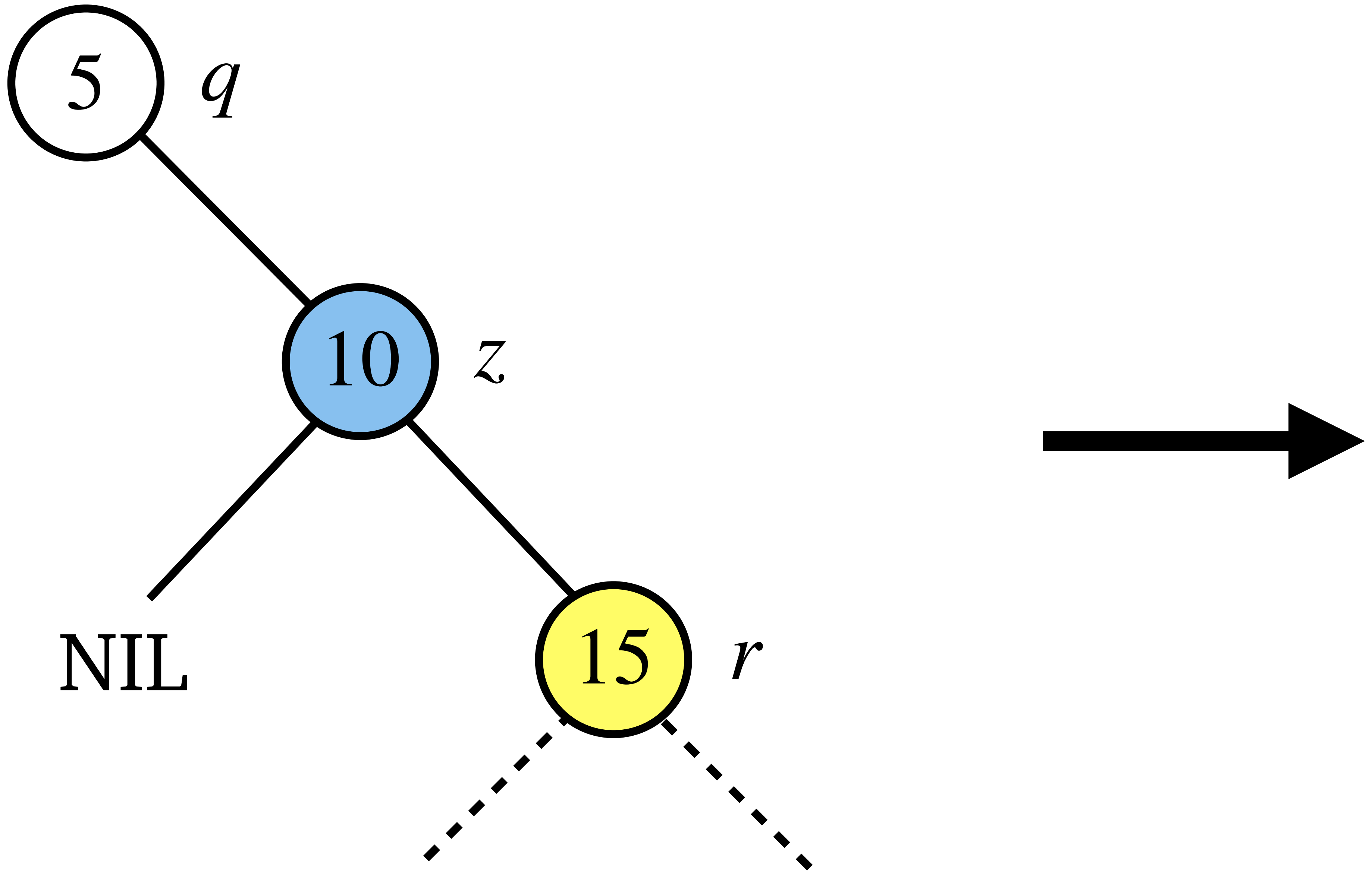
Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)



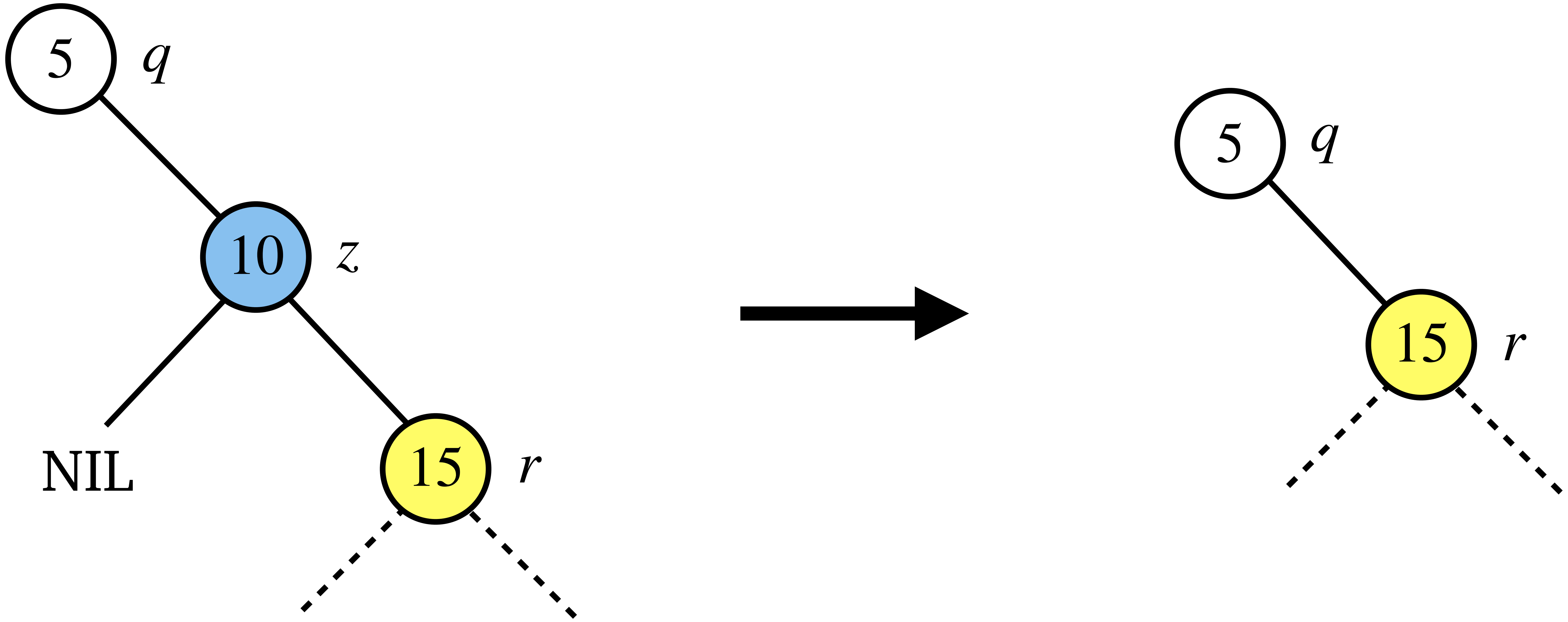
Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)



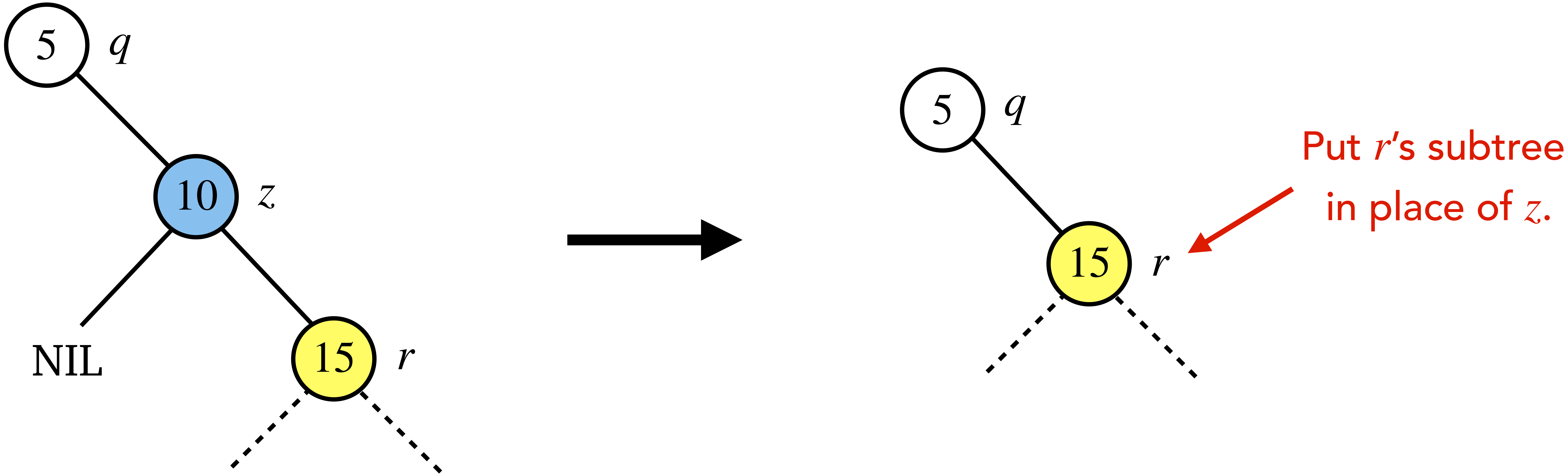
Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)



Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)

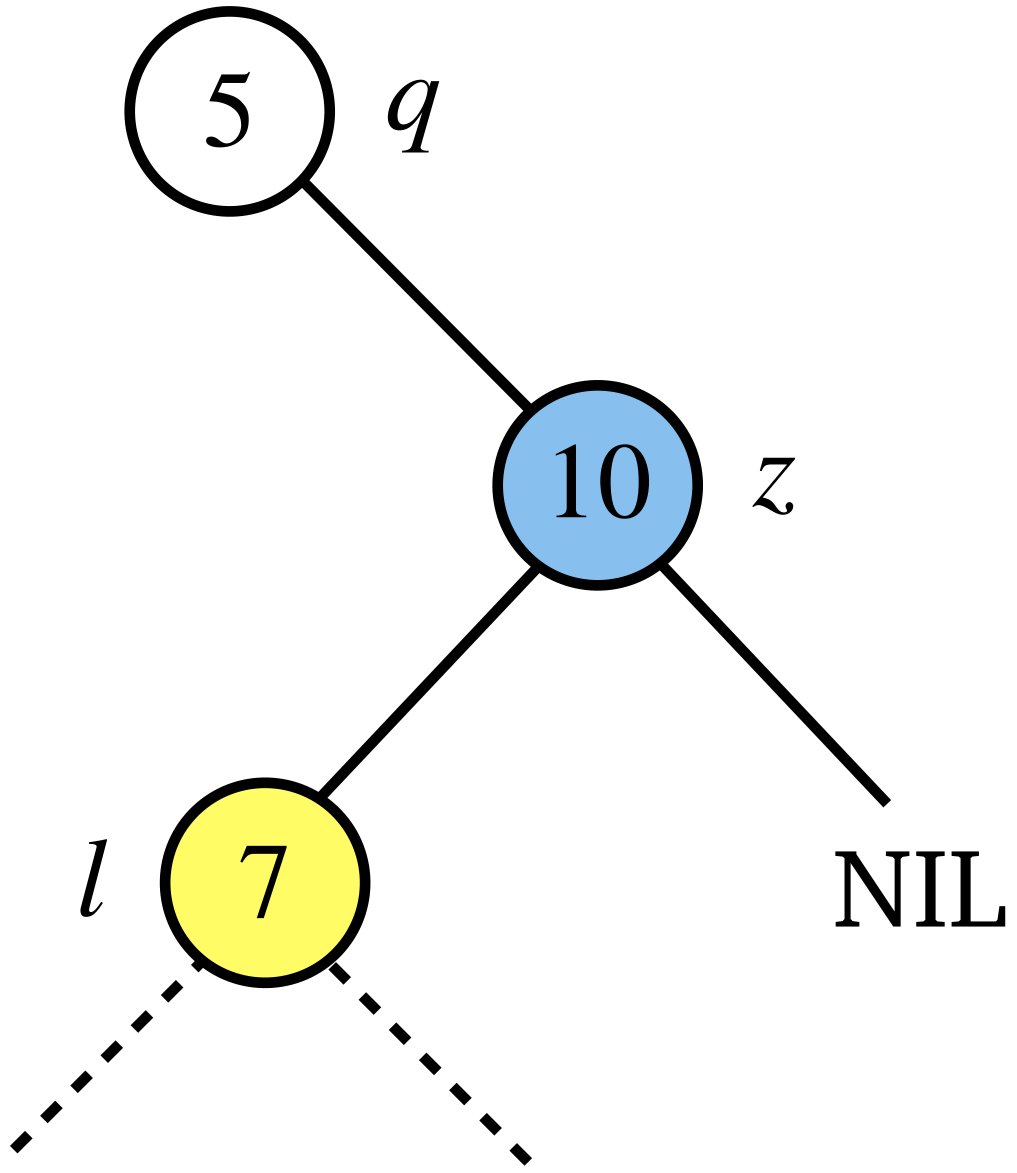


Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)

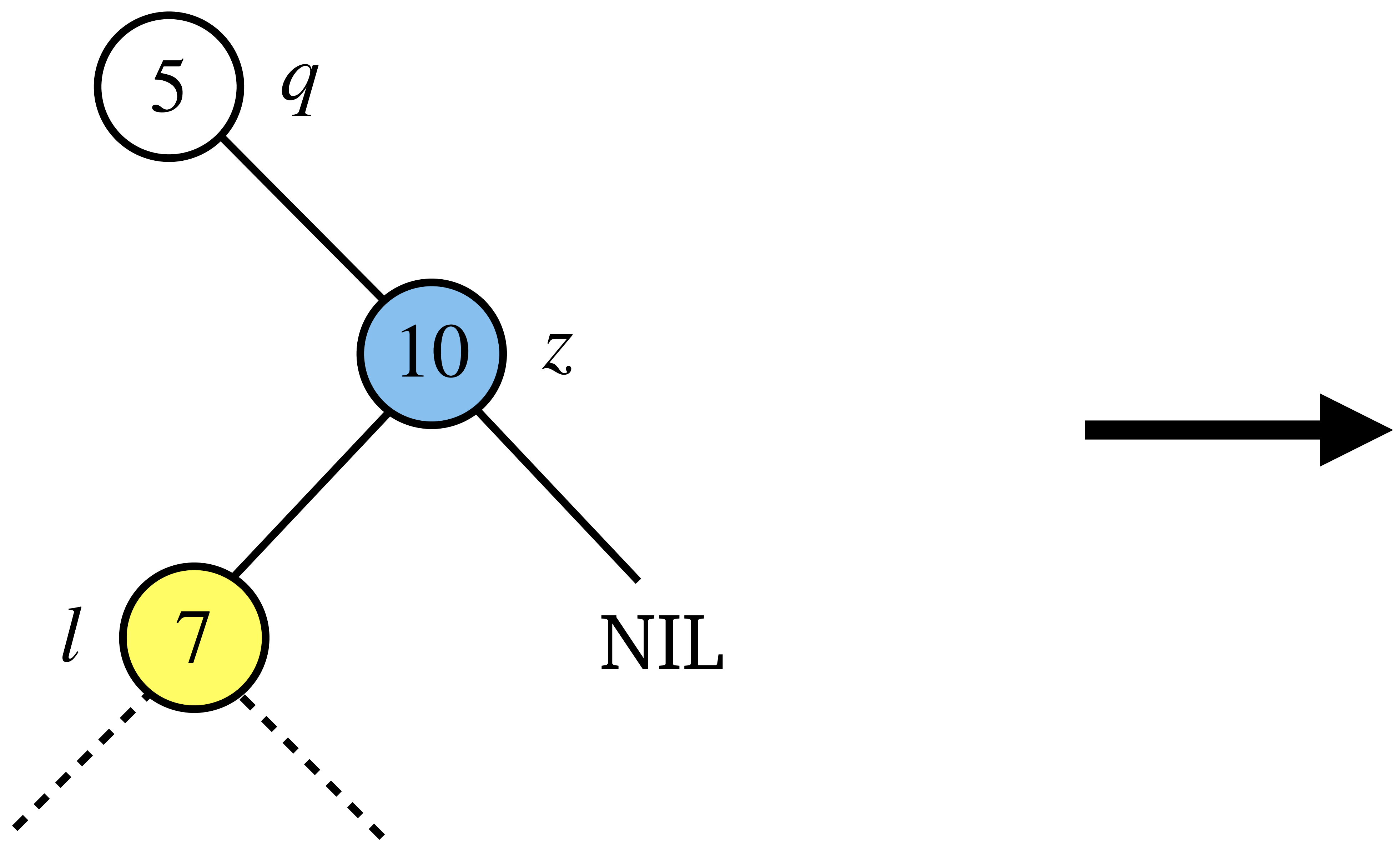
Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)



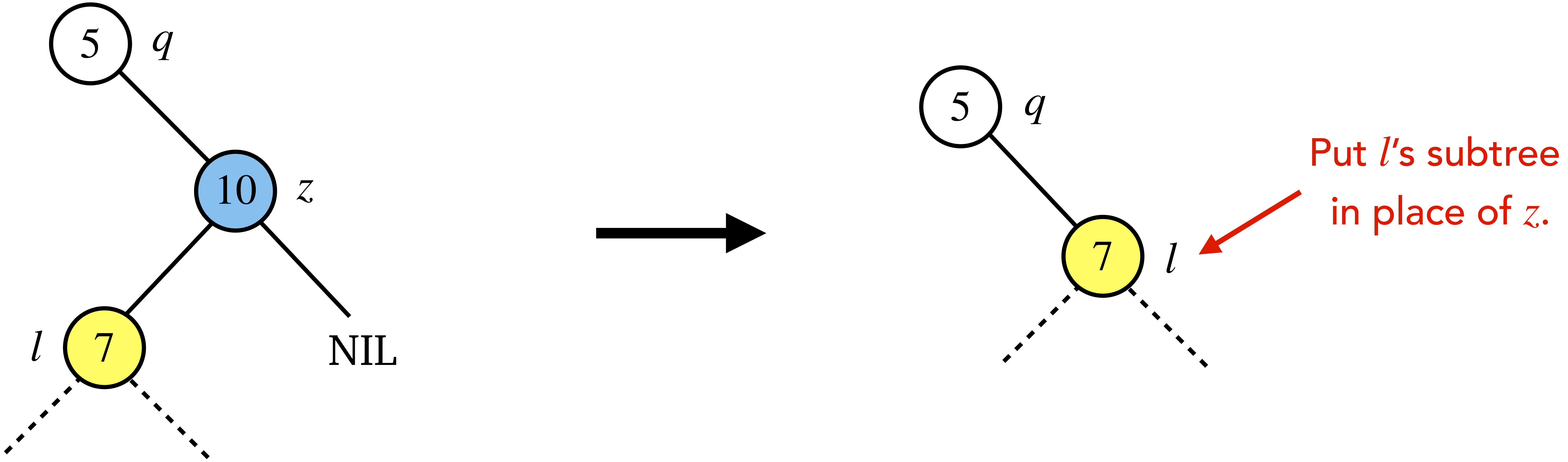
Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)



Deletion in a BST

Case 2: z has one child. (WLOG assume z is a right child.)



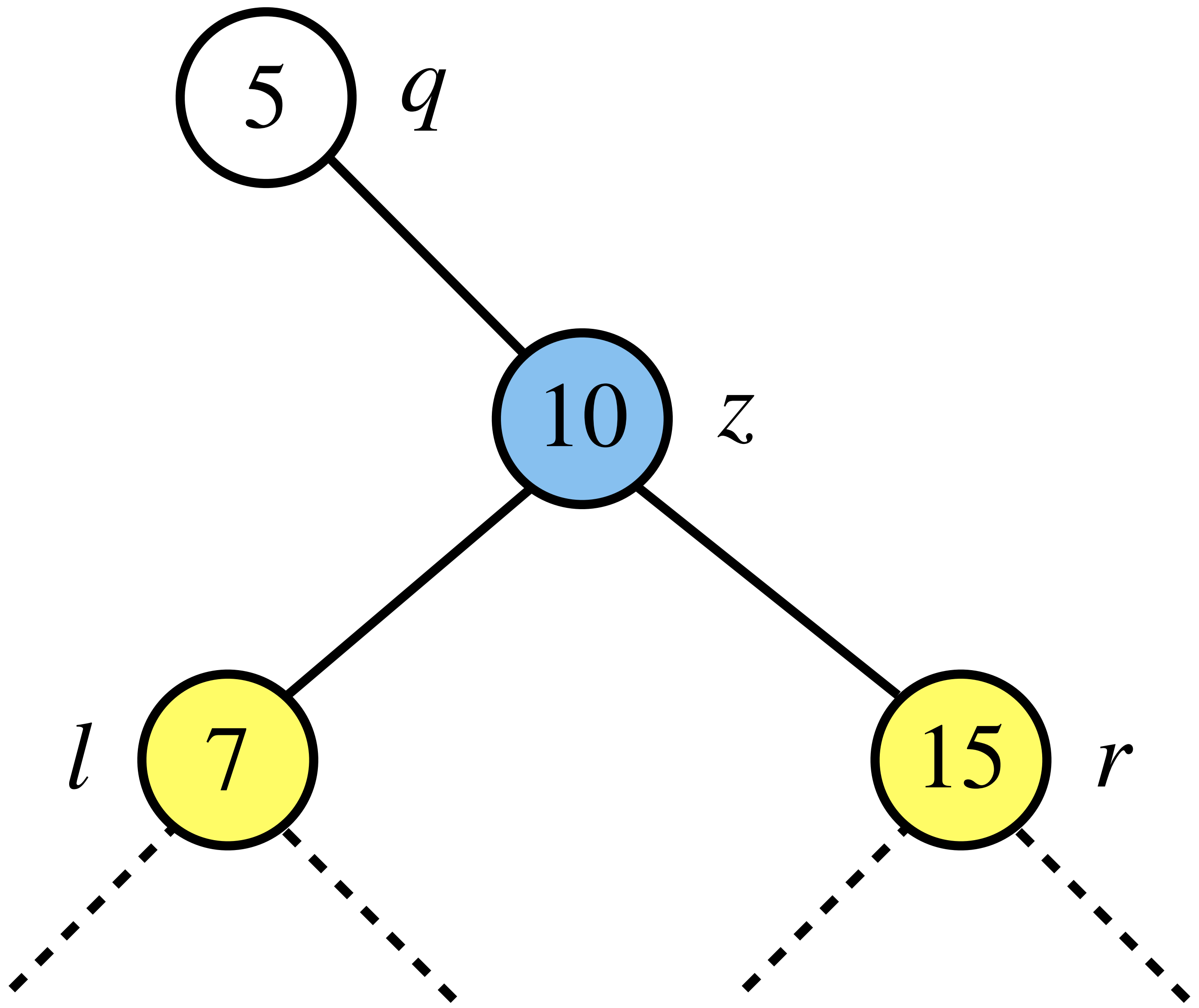
Deletion in a BST

Deletion in a BST

Case 3: z has two children. (WLOG assume z is a right child.)

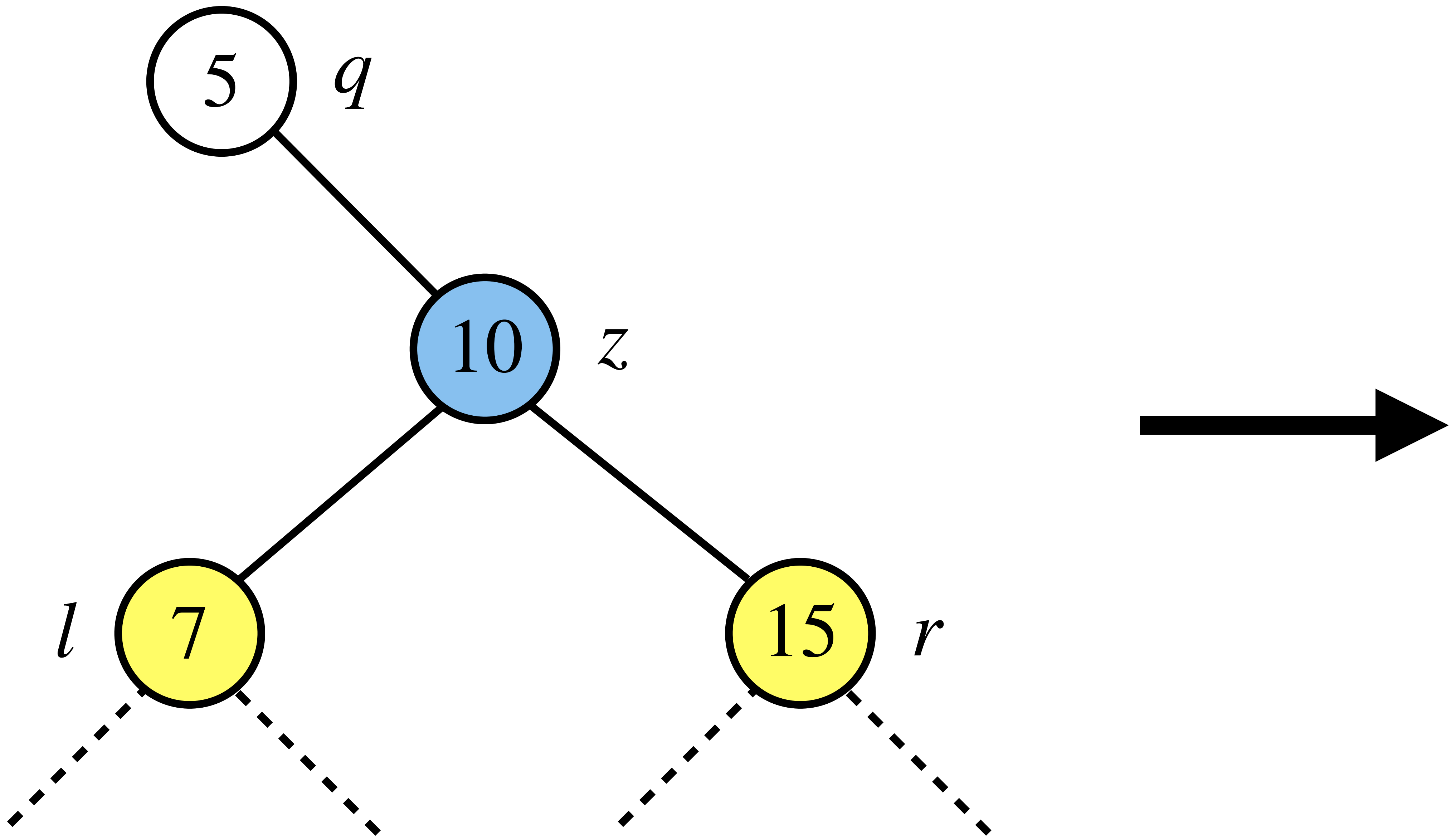
Deletion in a BST

Case 3: z has two children. (WLOG assume z is a right child.)



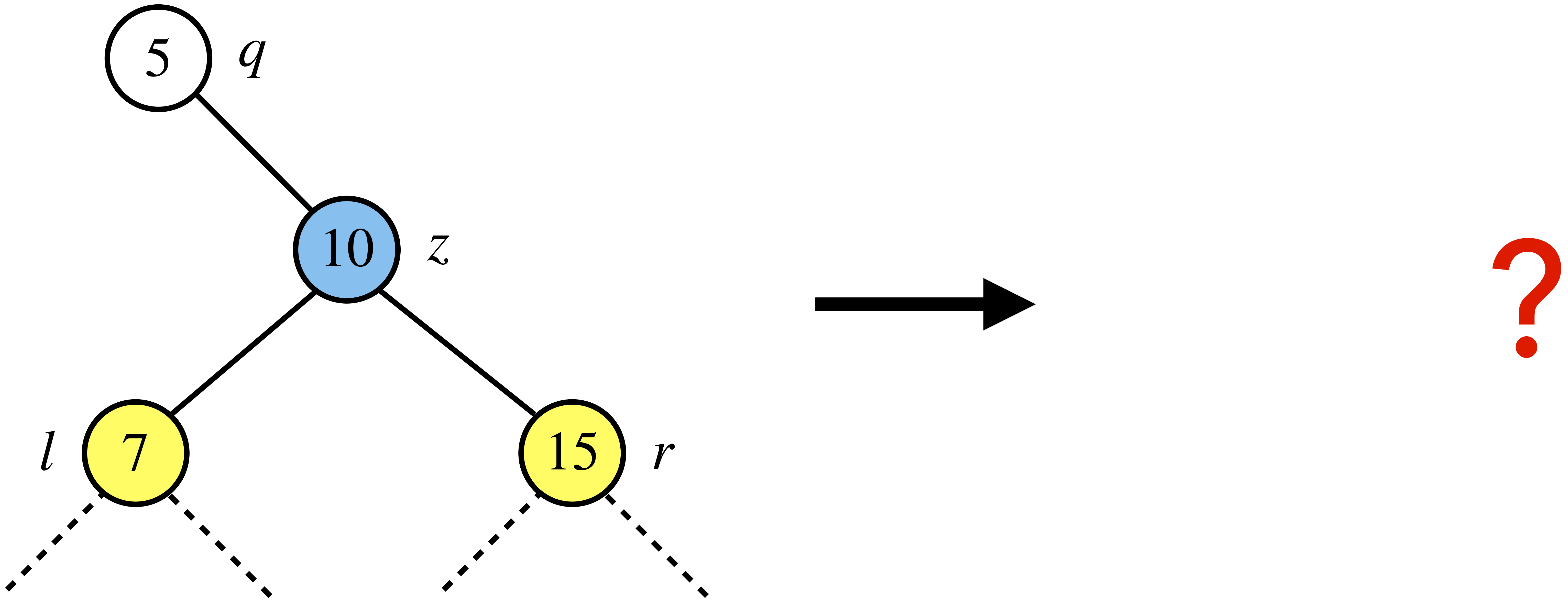
Deletion in a BST

Case 3: z has two children. (WLOG assume z is a right child.)



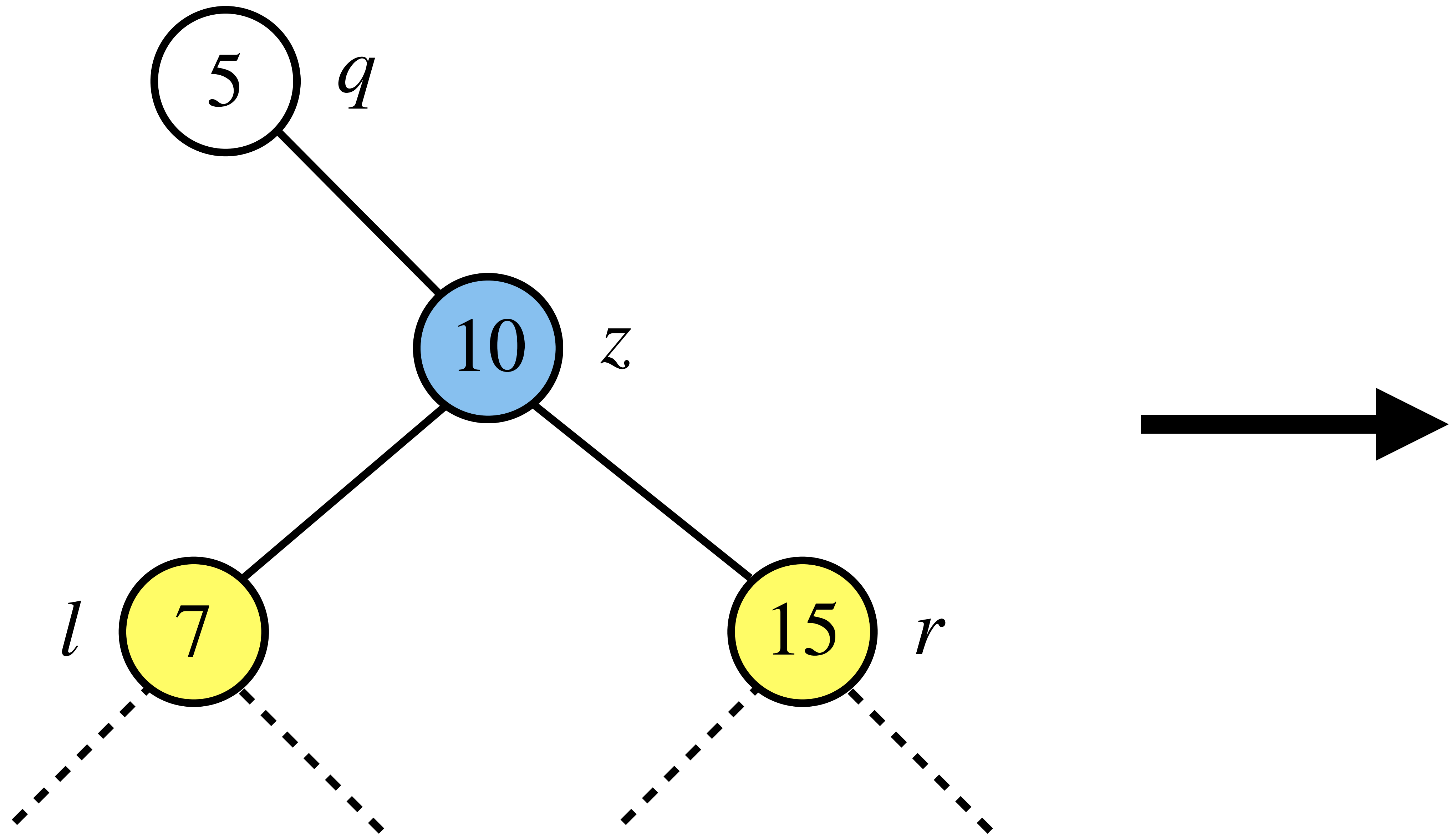
Deletion in a BST

Case 3: z has two children. (WLOG assume z is a right child.)



Deletion in a BST

Case 3: z has two children. (WLOG assume z is a right child.)



Two sub-cases:

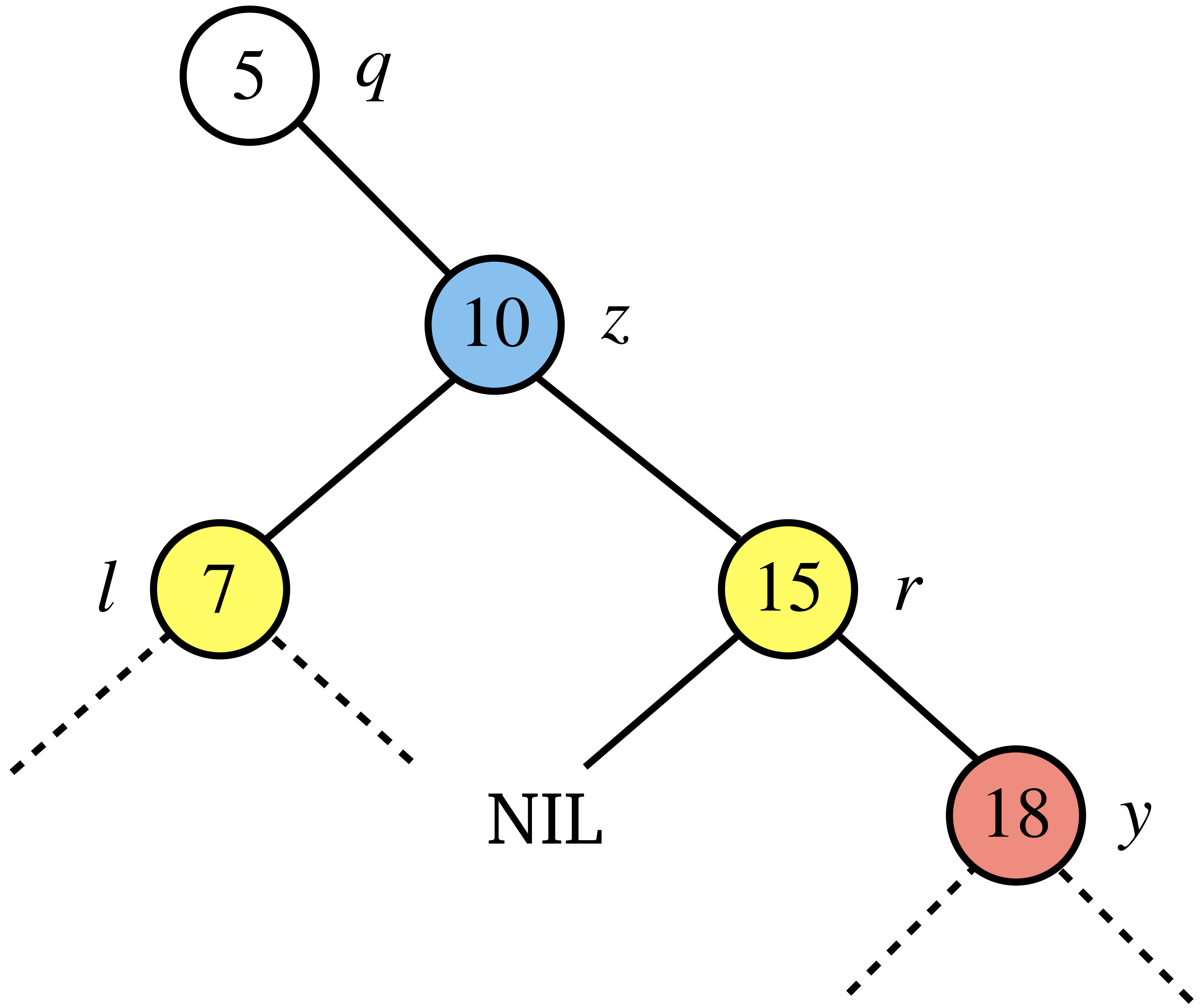
- r has no left child.
- r has a left child.

Deletion in a BST

Case 3a: z has two children where its right child has no left child.

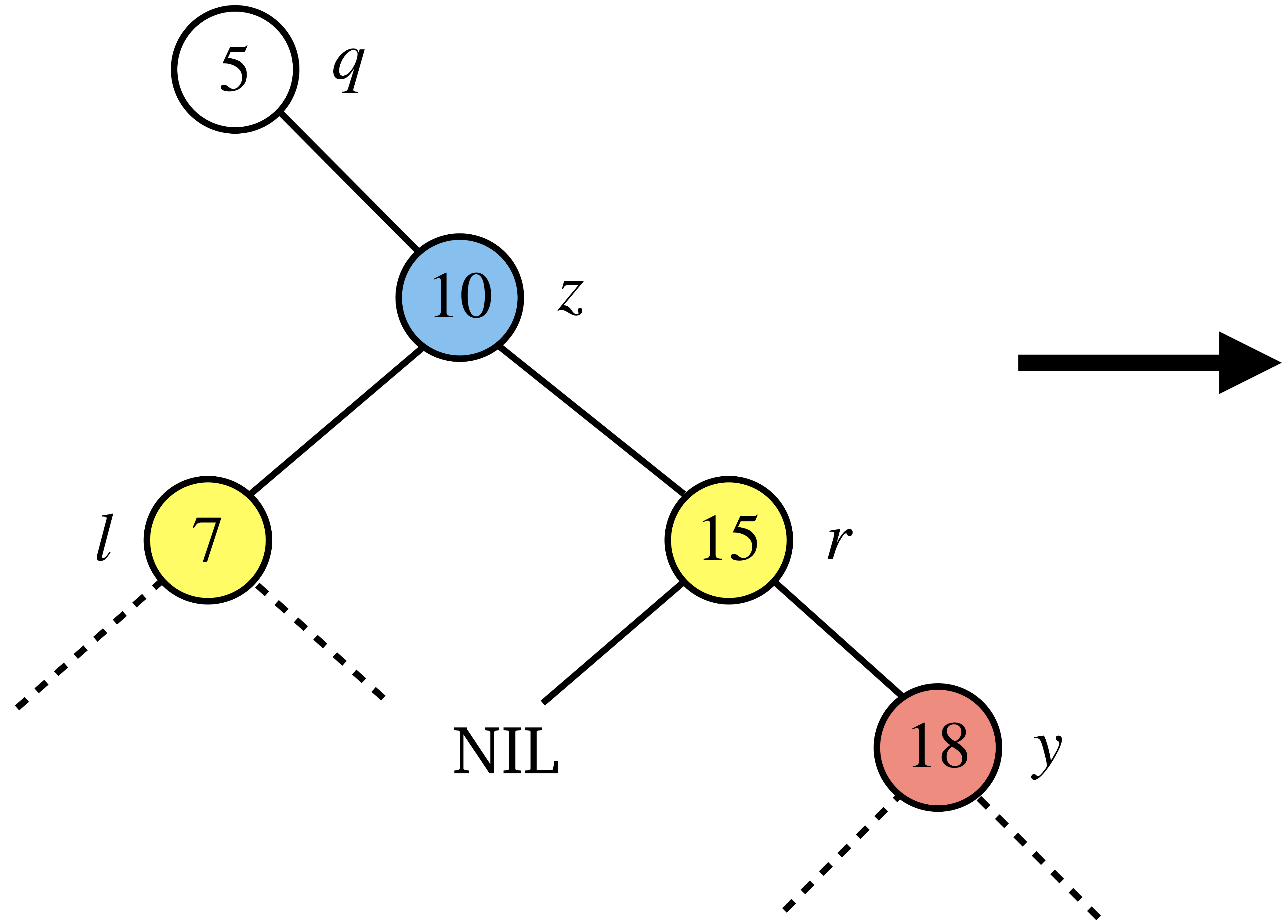
Deletion in a BST

Case 3**a**: z has two children where its right child has no left child.



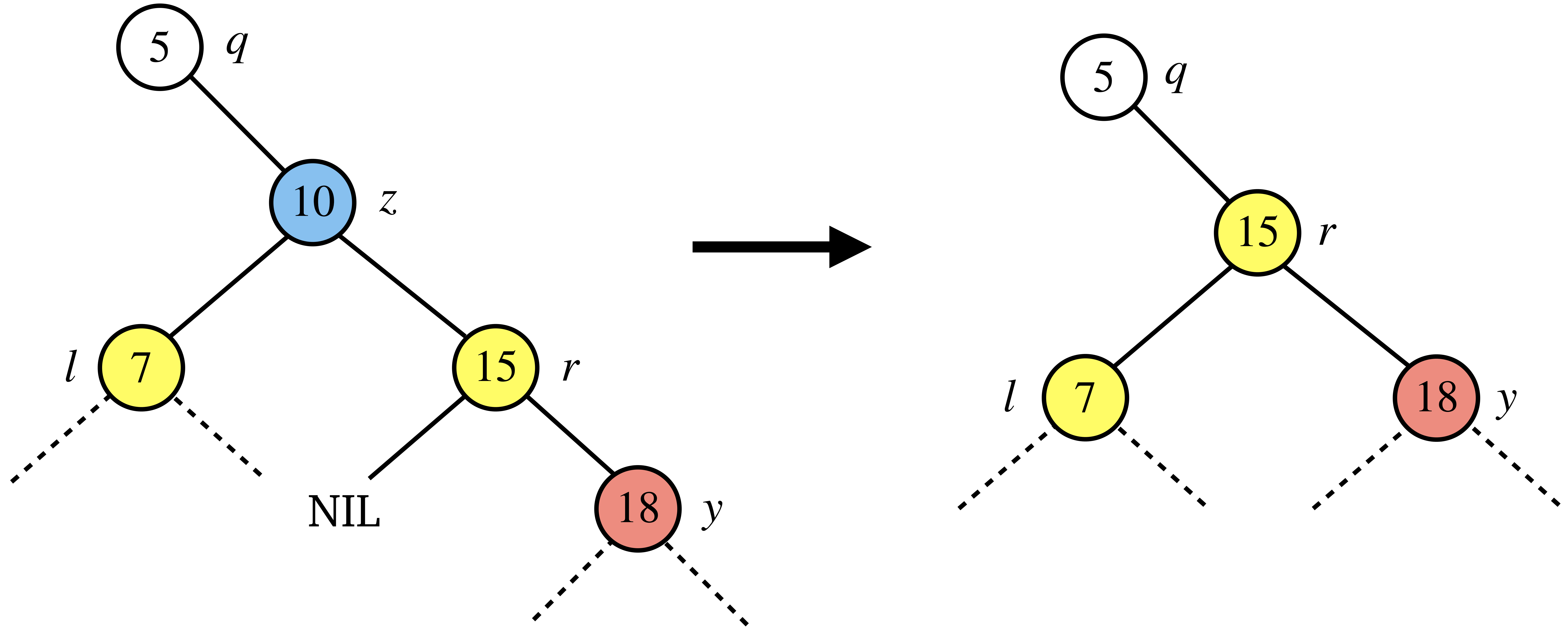
Deletion in a BST

Case 3**a**: z has two children where its right child has no left child.



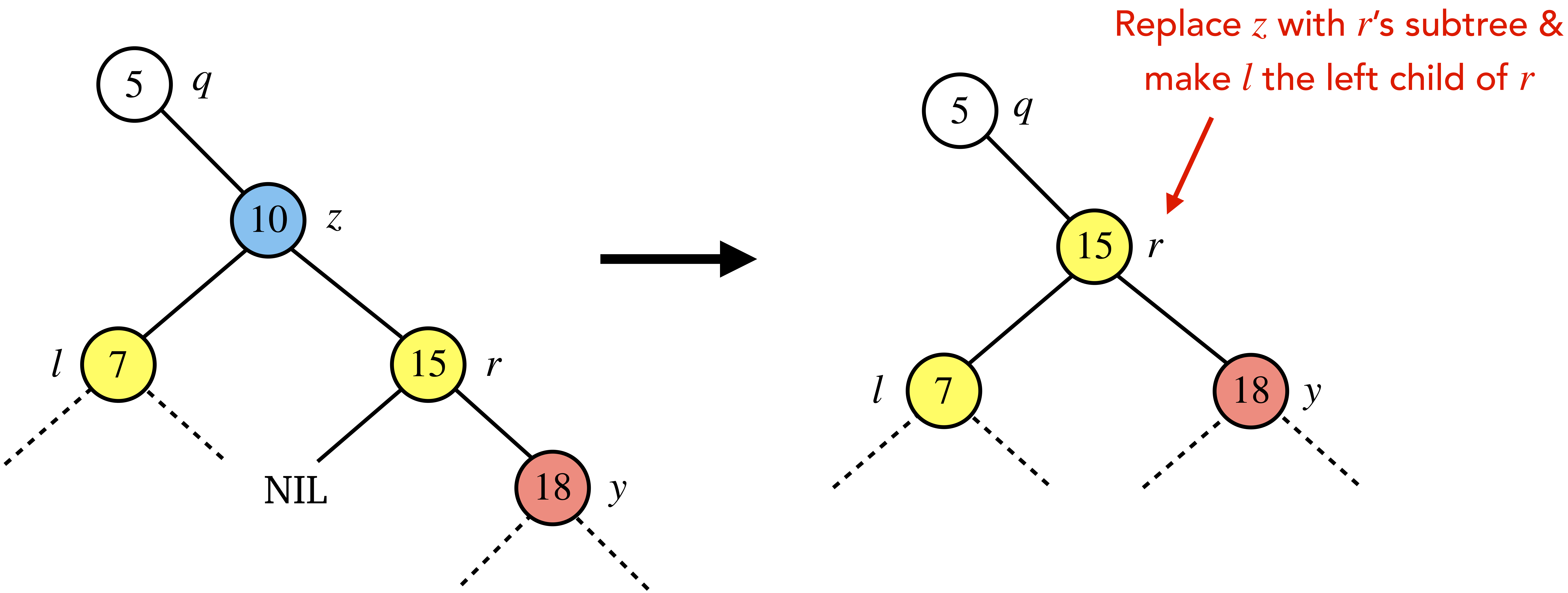
Deletion in a BST

Case 3a: z has two children where its right child has no left child.



Deletion in a BST

Case 3a: z has two children where its right child has no left child.



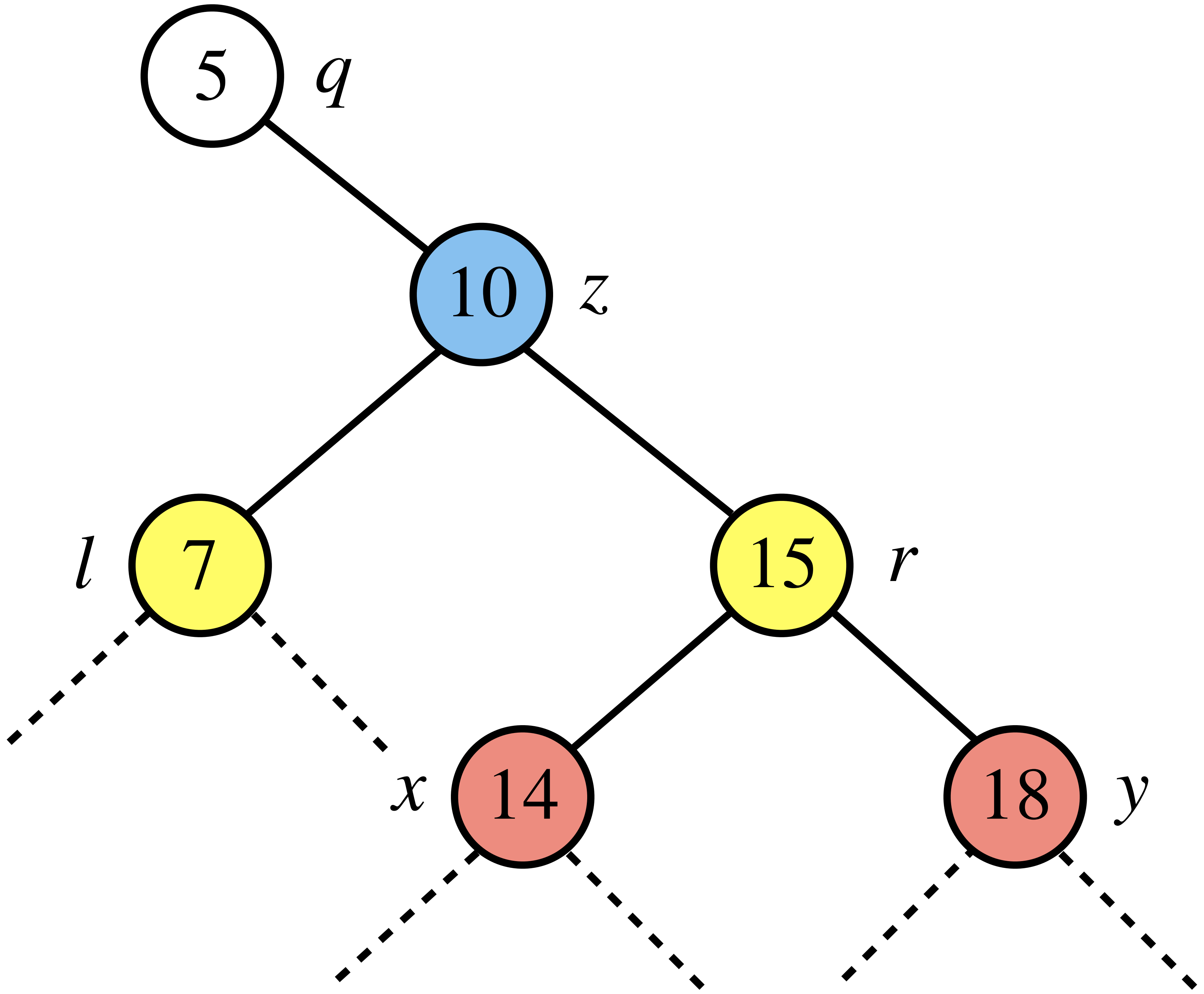
Deletion in a BST

Deletion in a BST

Case 3b: z has two children where its right child has a left child.

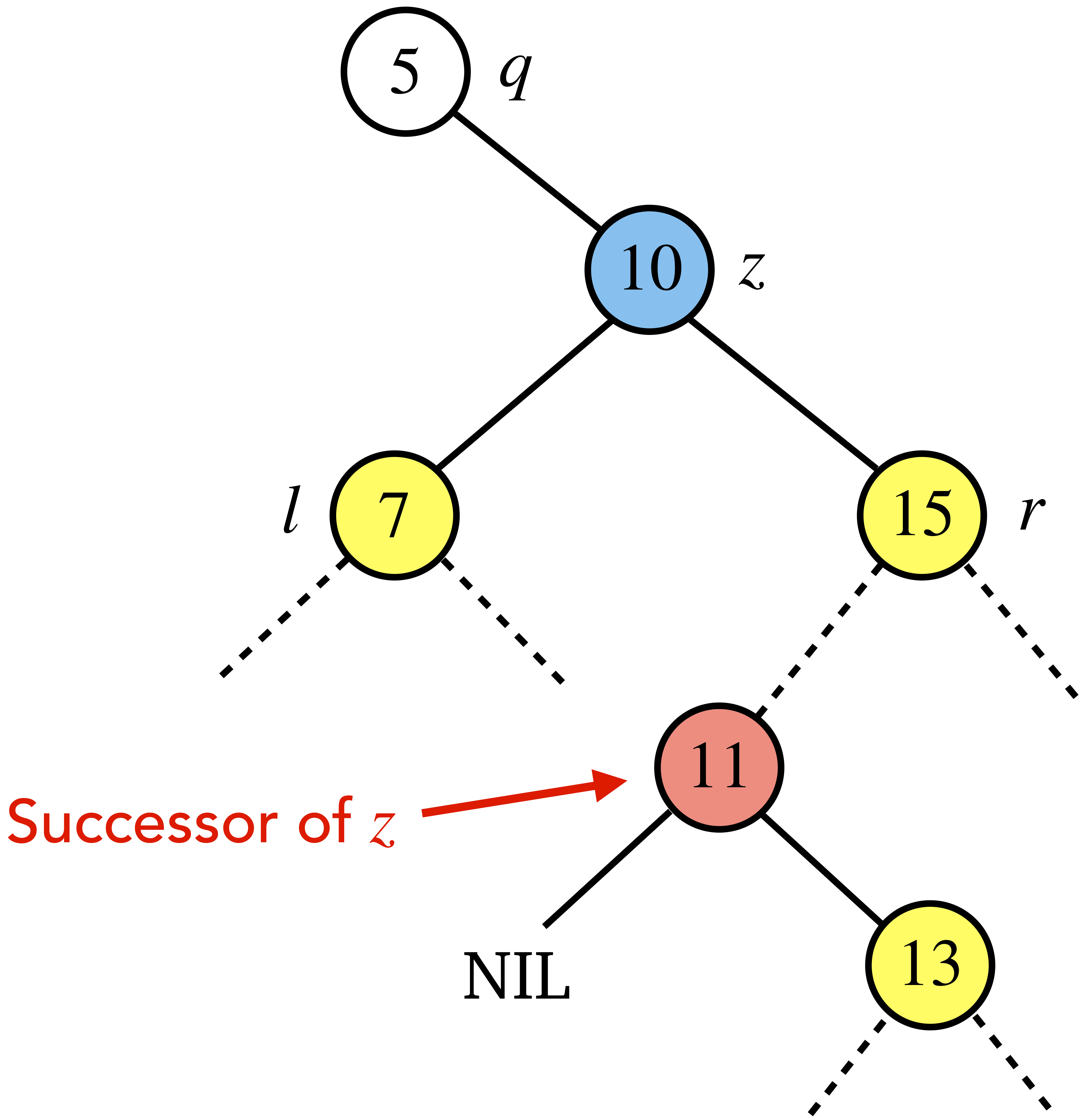
Deletion in a BST

Case 3**b**: z has two children where its right child has a left child.



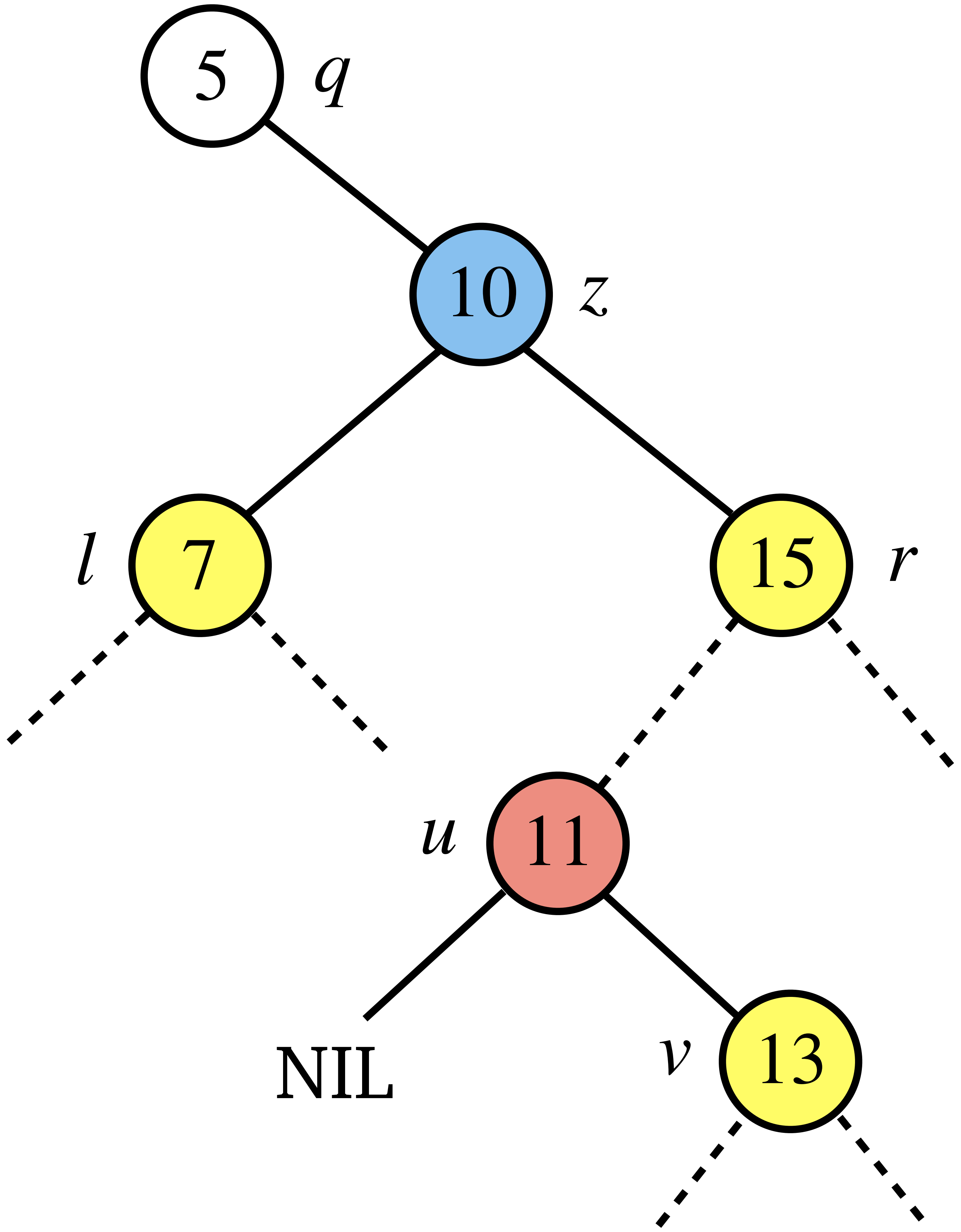
Deletion in a BST

Case 3**b**: z has two children where its right child has a left child.



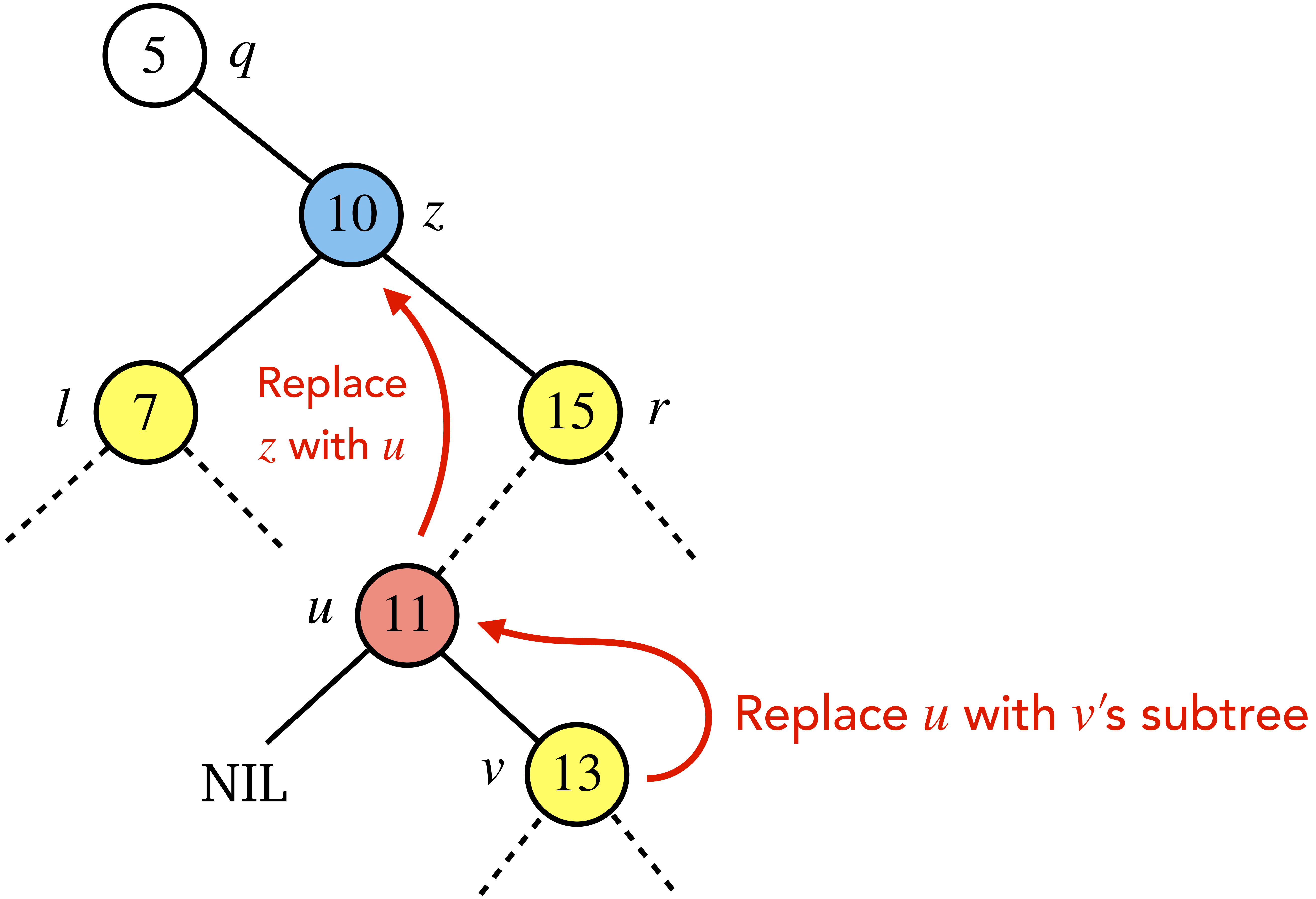
Deletion in a BST

Case 3**b**: z has two children where its right child has a left child.



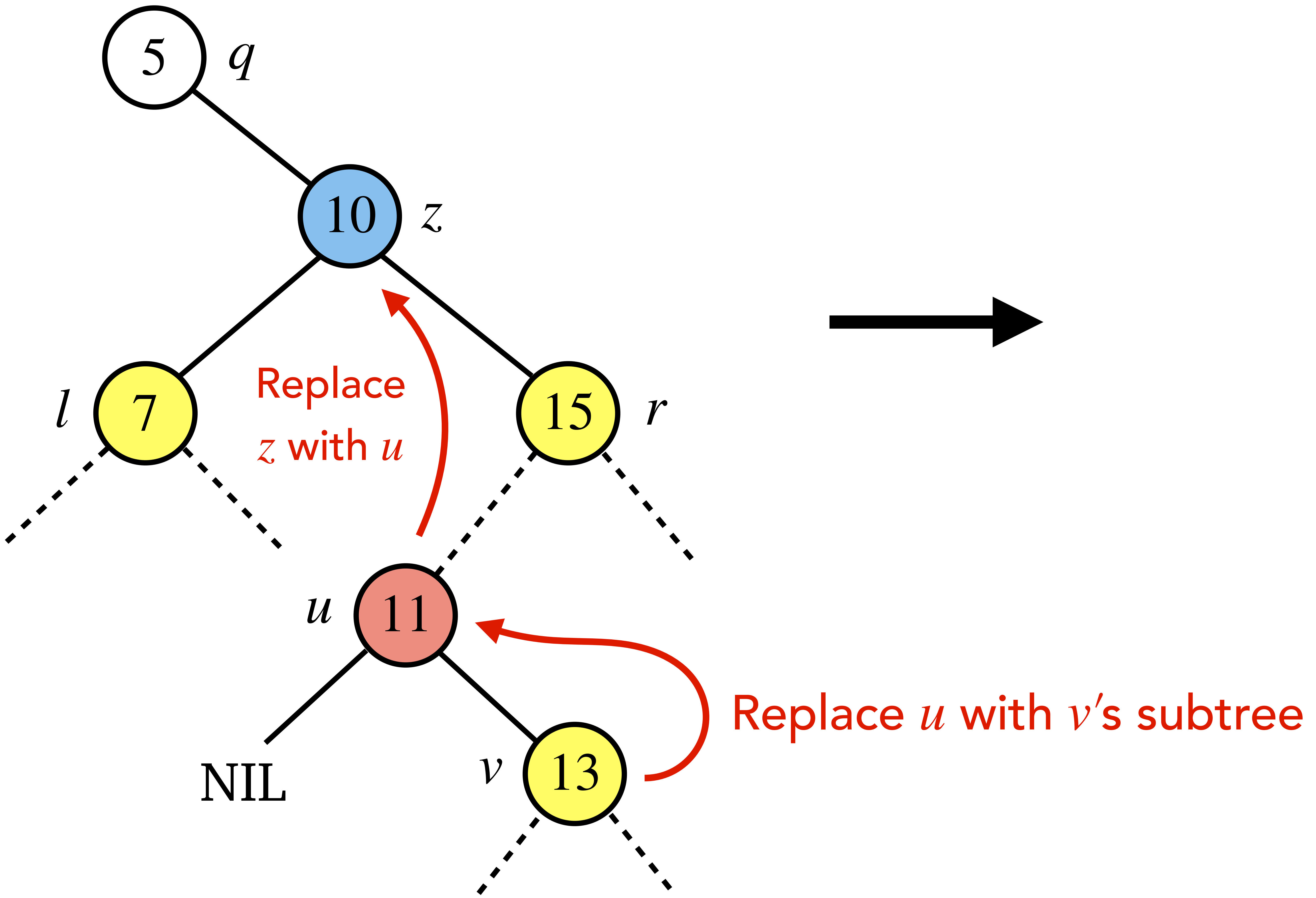
Deletion in a BST

Case 3**b:** z has two children where its right child has a left child.



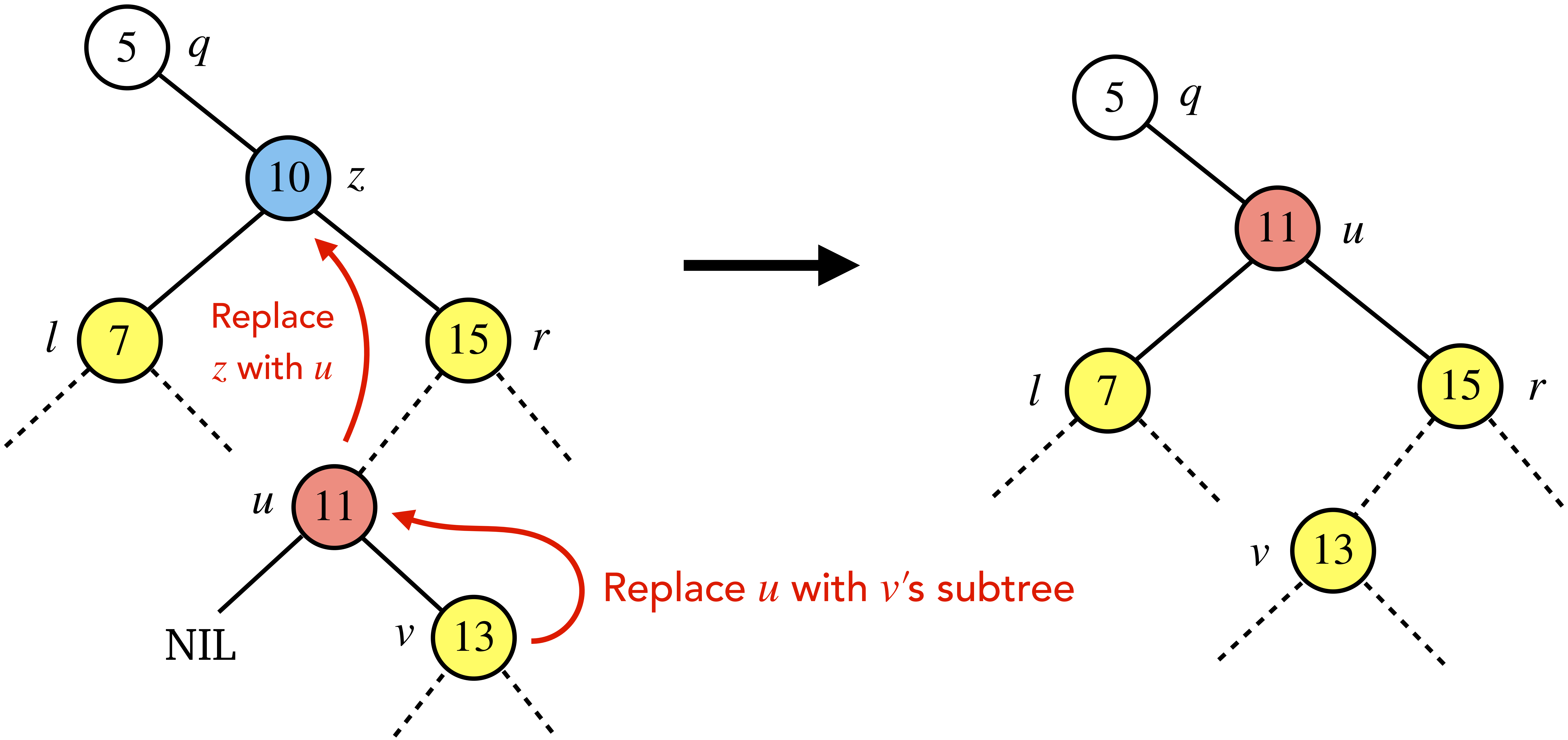
Deletion in a BST

Case 3**b**: z has two children where its right child has a left child.



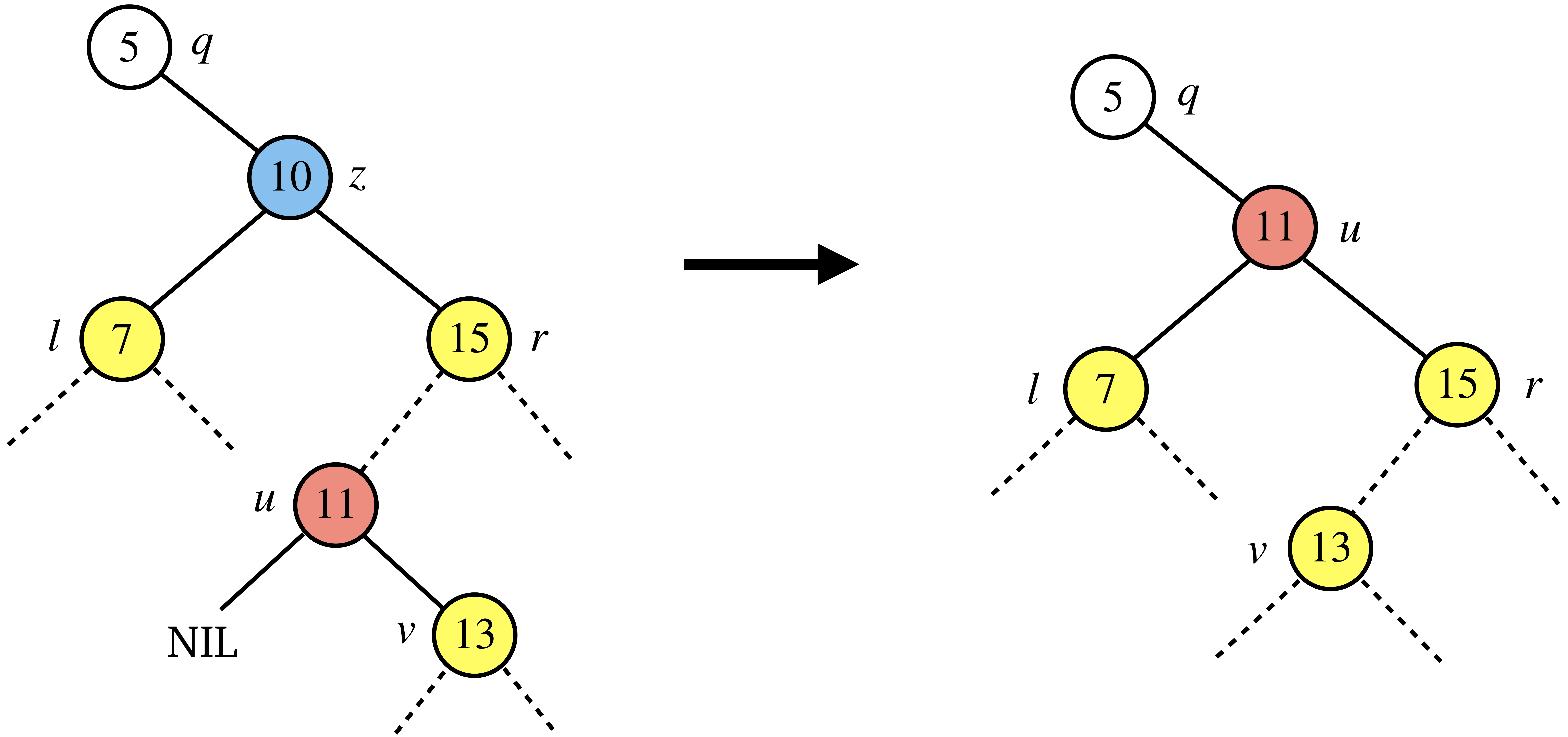
Deletion in a BST

Case 3**b**: z has two children where its right child has a left child.



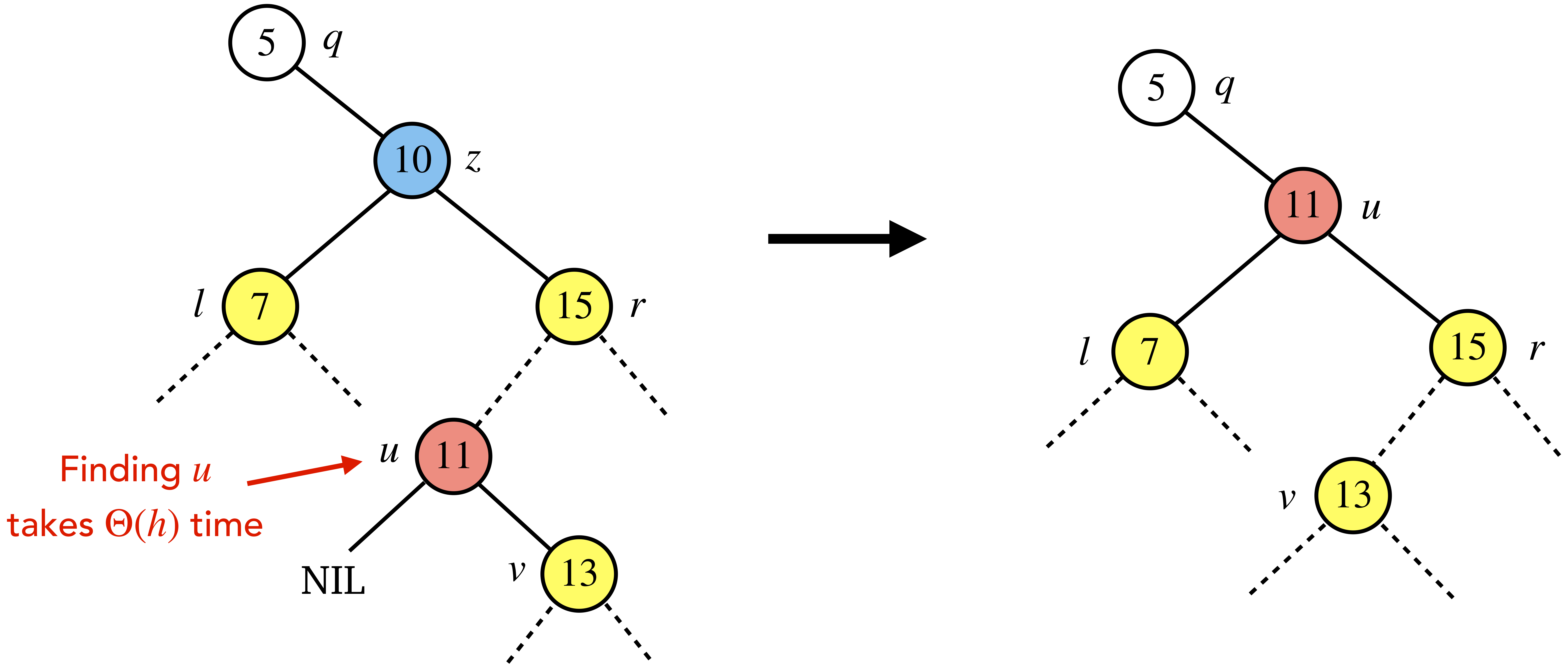
Deletion in a BST

Case 3**b**: z has two children where its right child has a left child.



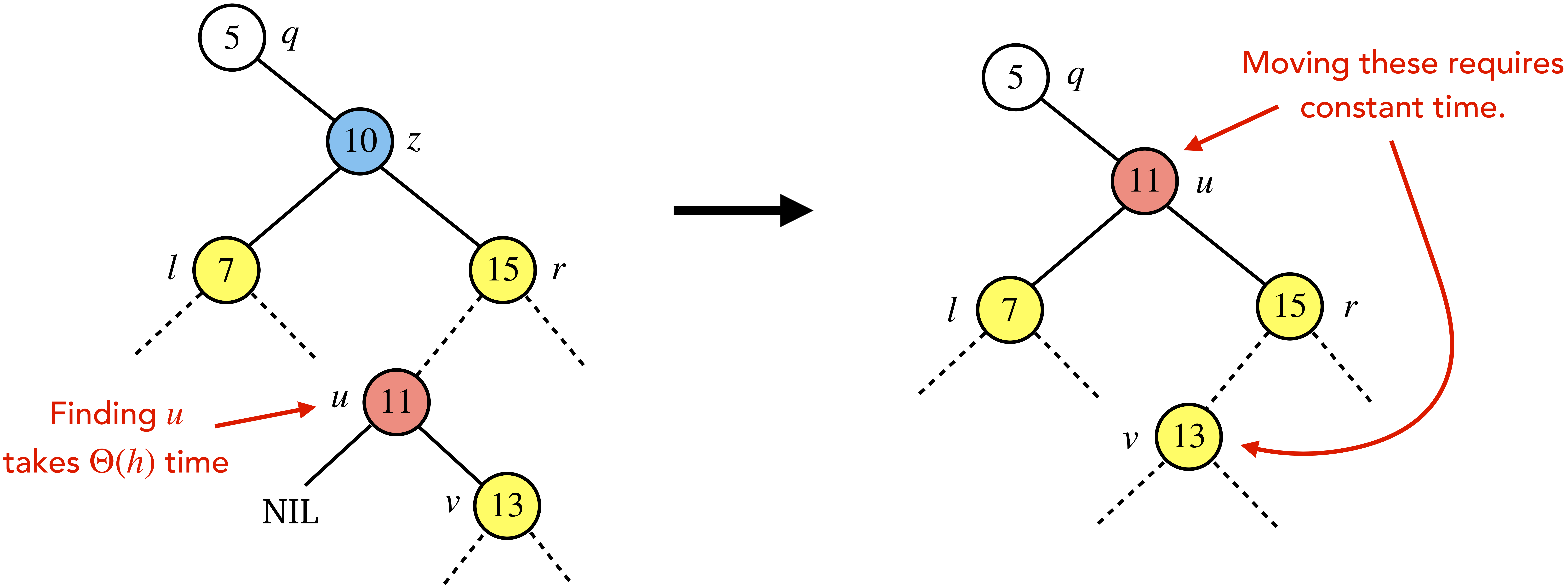
Deletion in a BST

Case 3**b**: z has two children where its right child has a left child.



Deletion in a BST

Case 3**b:** z has two children where its right child has a left child.



Are BSTs Good Enough?

Are BSTs Good Enough?

BSTs can perform Insert, Delete, Search, etc., in $\Theta(h)$ time.

Are BSTs Good Enough?

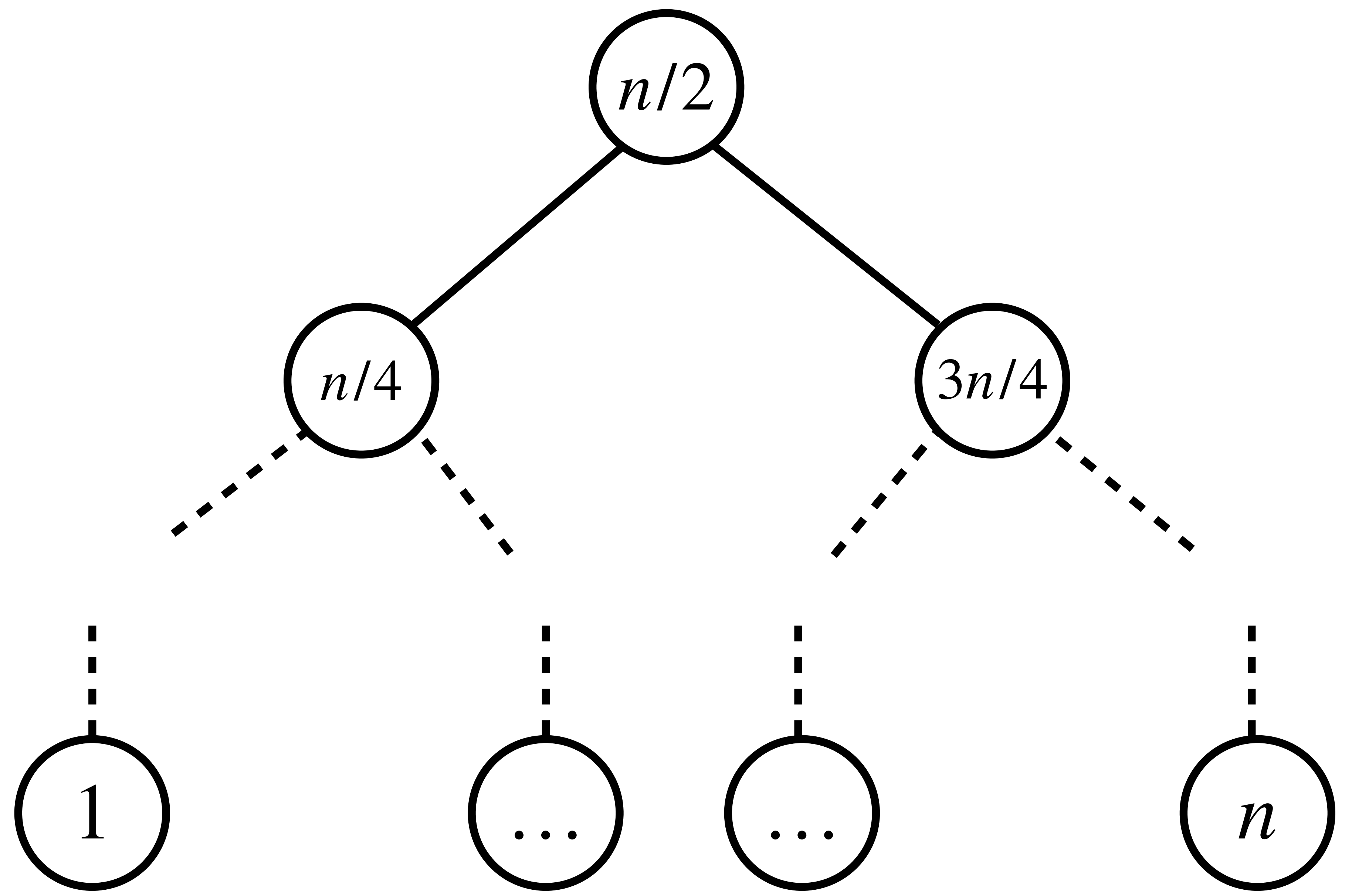
BSTs can perform Insert, Delete, Search, etc., in $\Theta(h)$ time.

But,

Are BSTs Good Enough?

BSTs can perform Insert, Delete, Search, etc., in $\Theta(h)$ time.

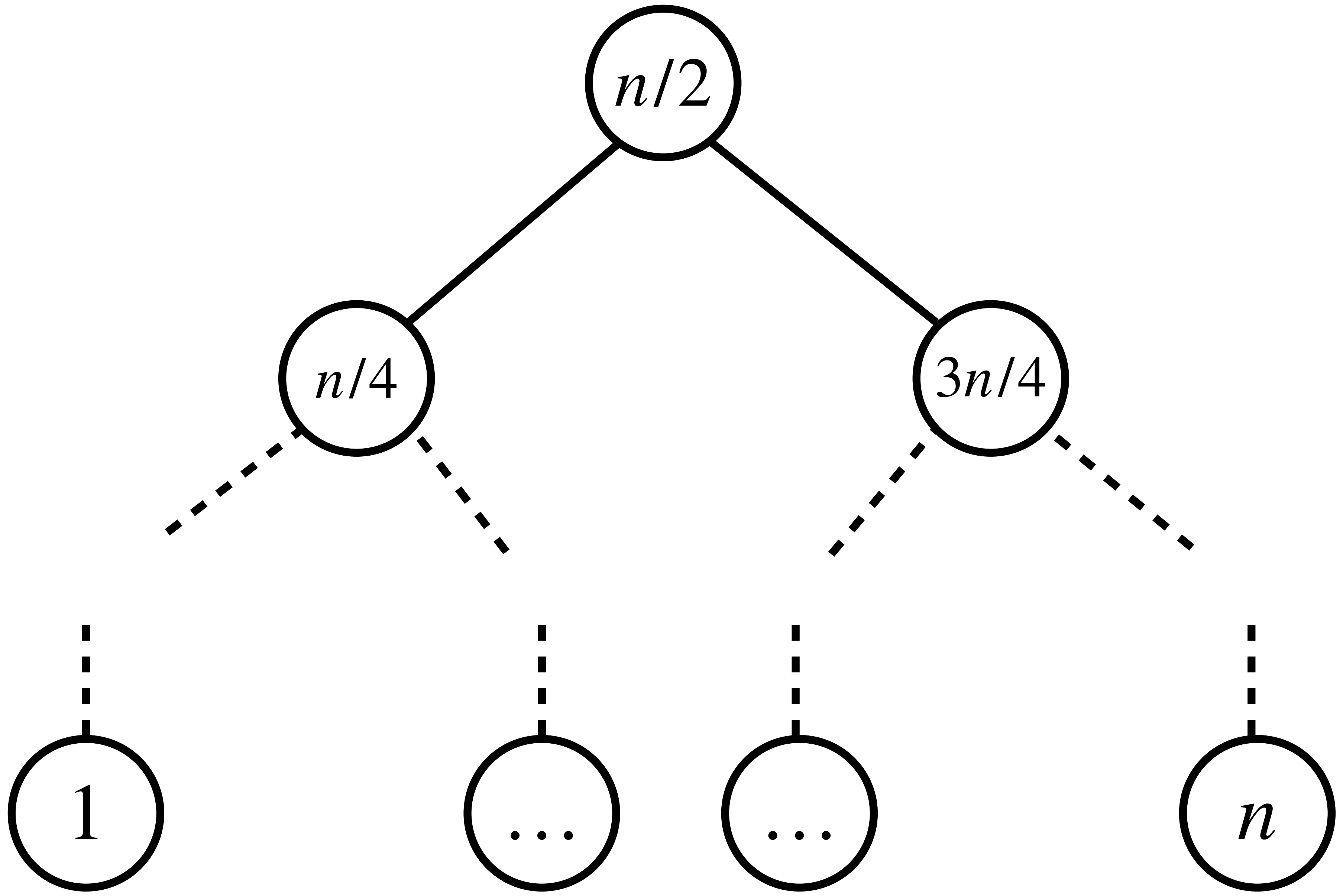
But,



Are BSTs Good Enough?

BSTs can perform Insert, Delete, Search, etc., in $\Theta(h)$ time.

But,

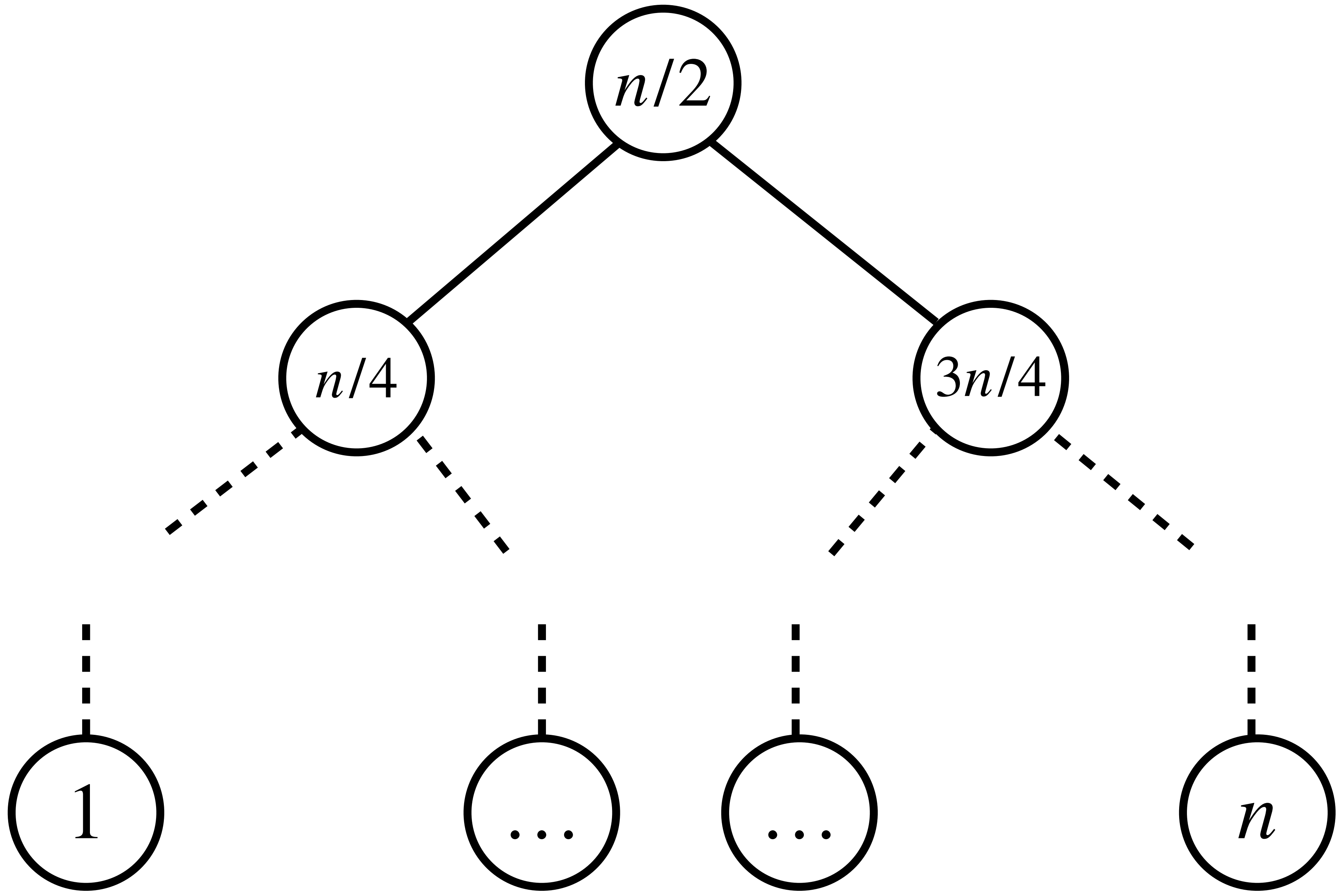


Best case: $h = \Theta(\log n)$

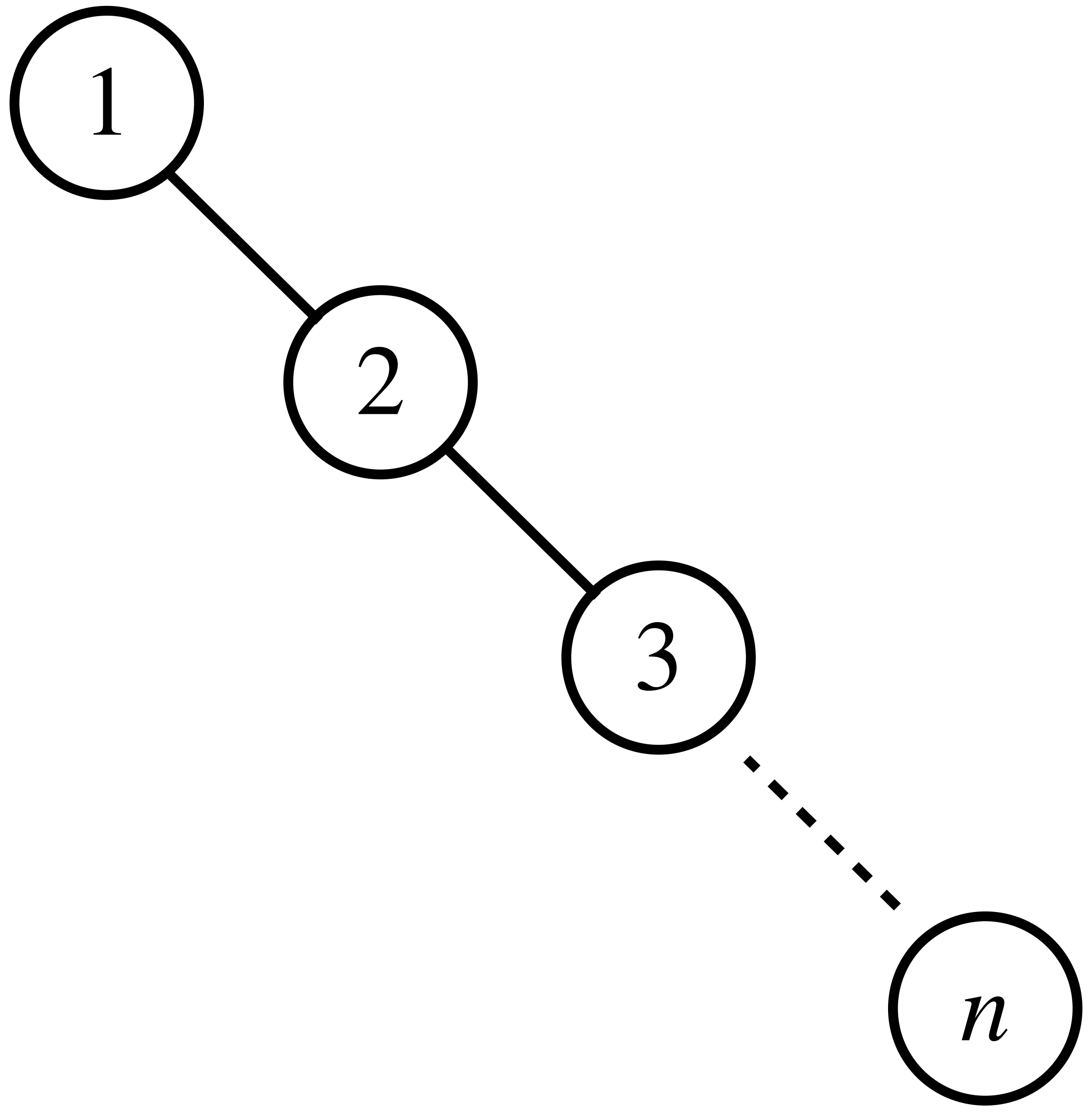
Are BSTs Good Enough?

BSTs can perform Insert, Delete, Search, etc., in $\Theta(h)$ time.

But,



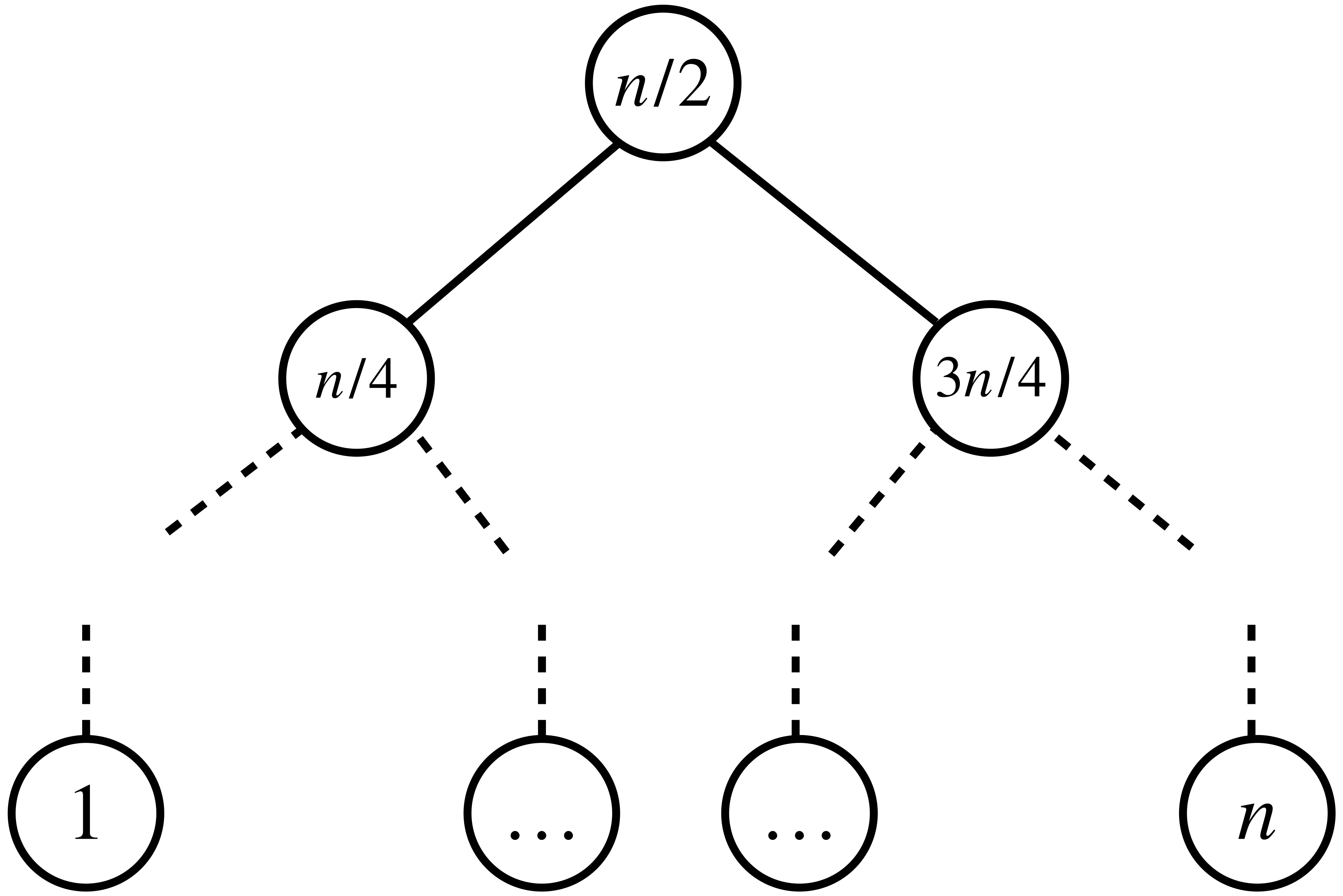
Best case: $h = \Theta(\log n)$



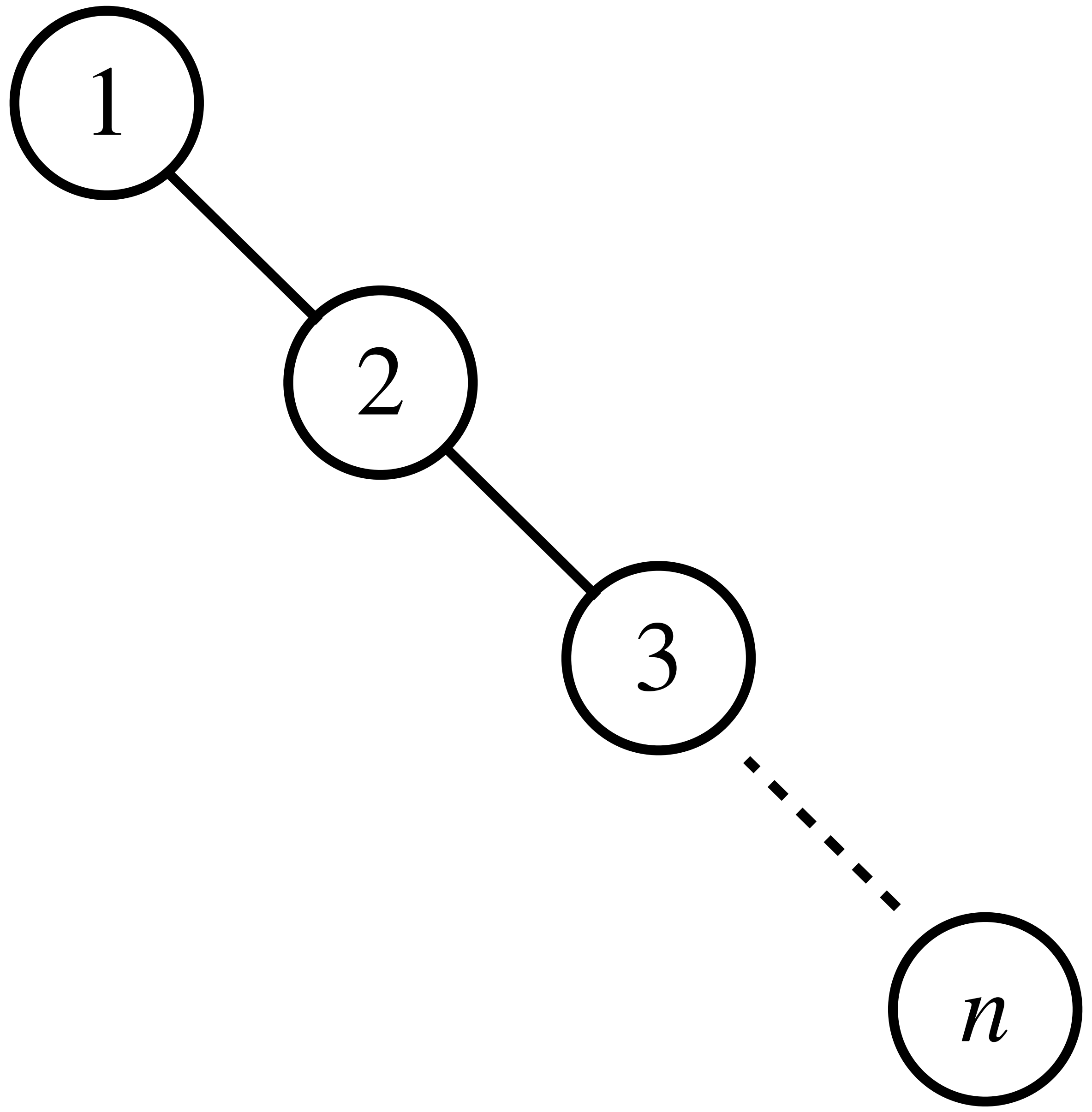
Are BSTs Good Enough?

BSTs can perform Insert, Delete, Search, etc., in $\Theta(h)$ time.

But,



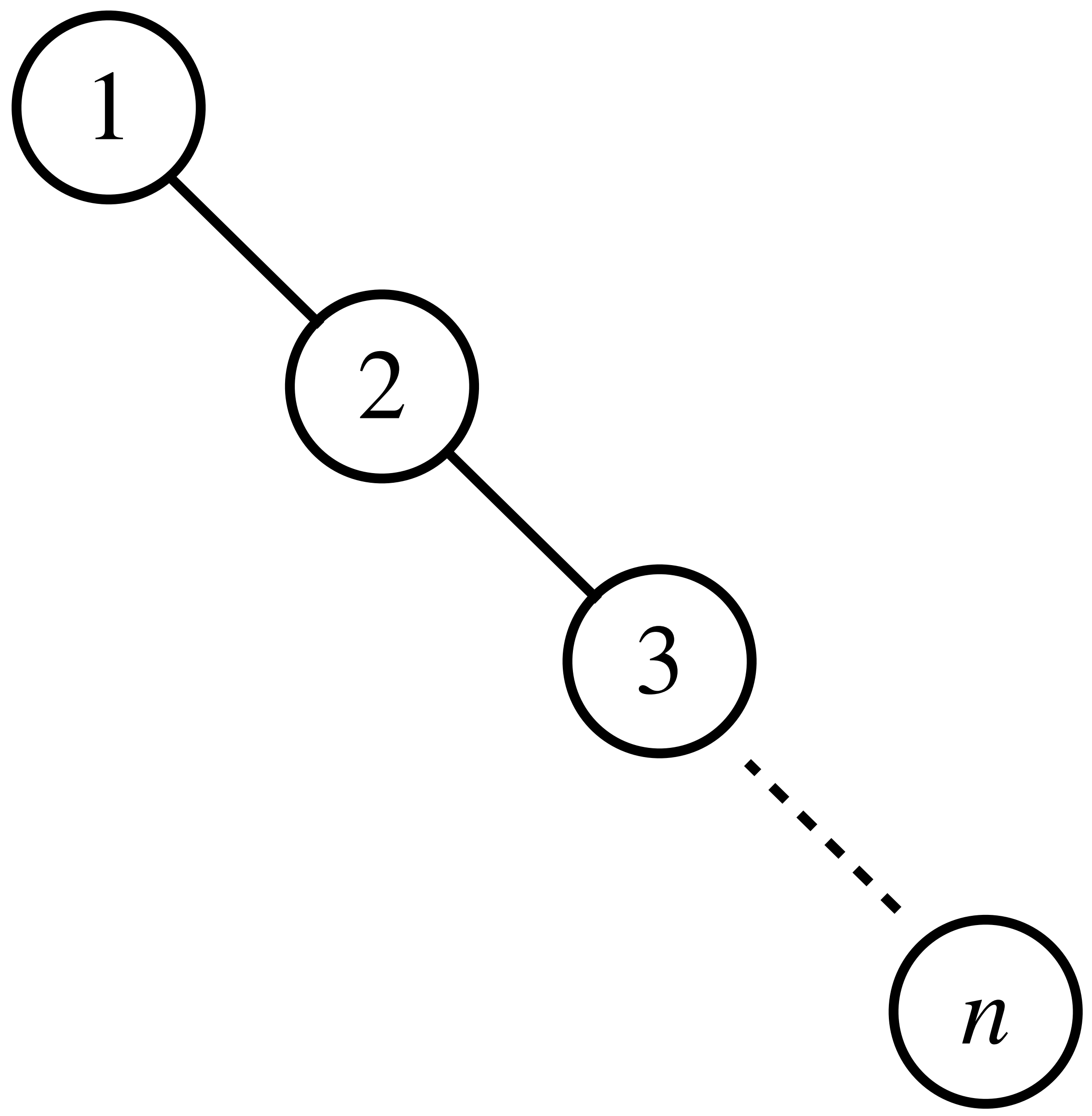
Best case: $h = \Theta(\log n)$



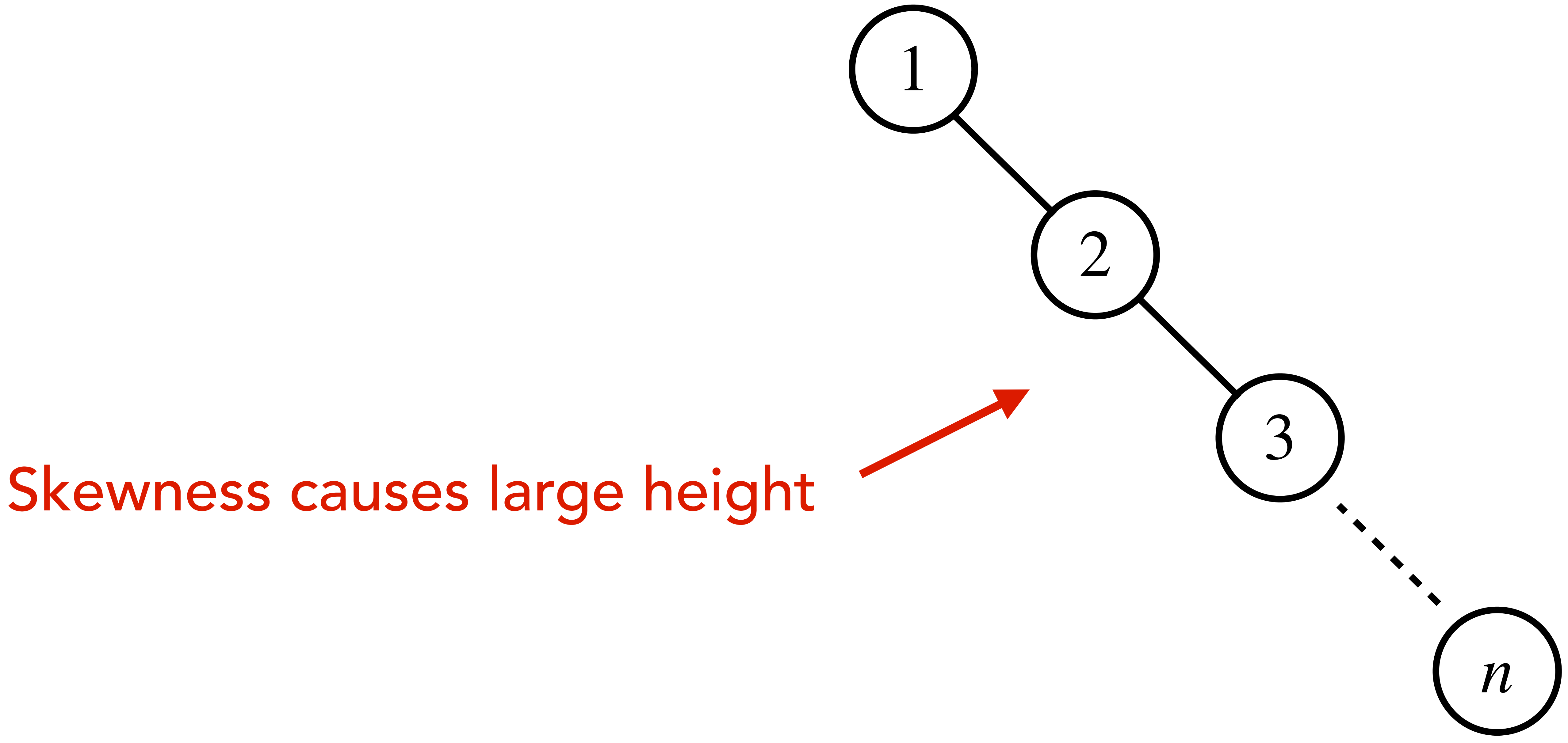
Worst case: $h = \Theta(n)$

How to Restrict Height in BSTs?

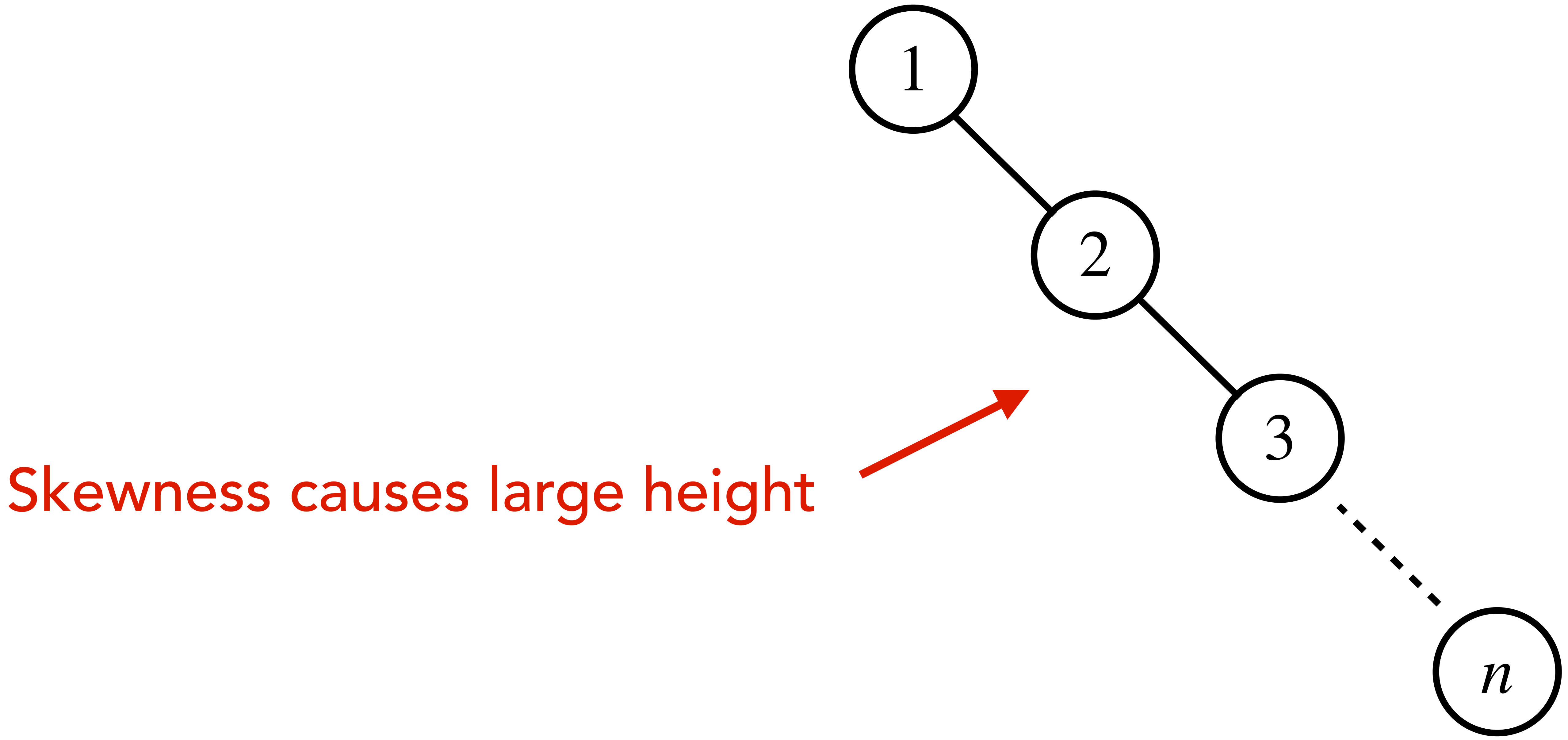
How to Restrict Height in BSTs?



How to Restrict Height in BSTs?

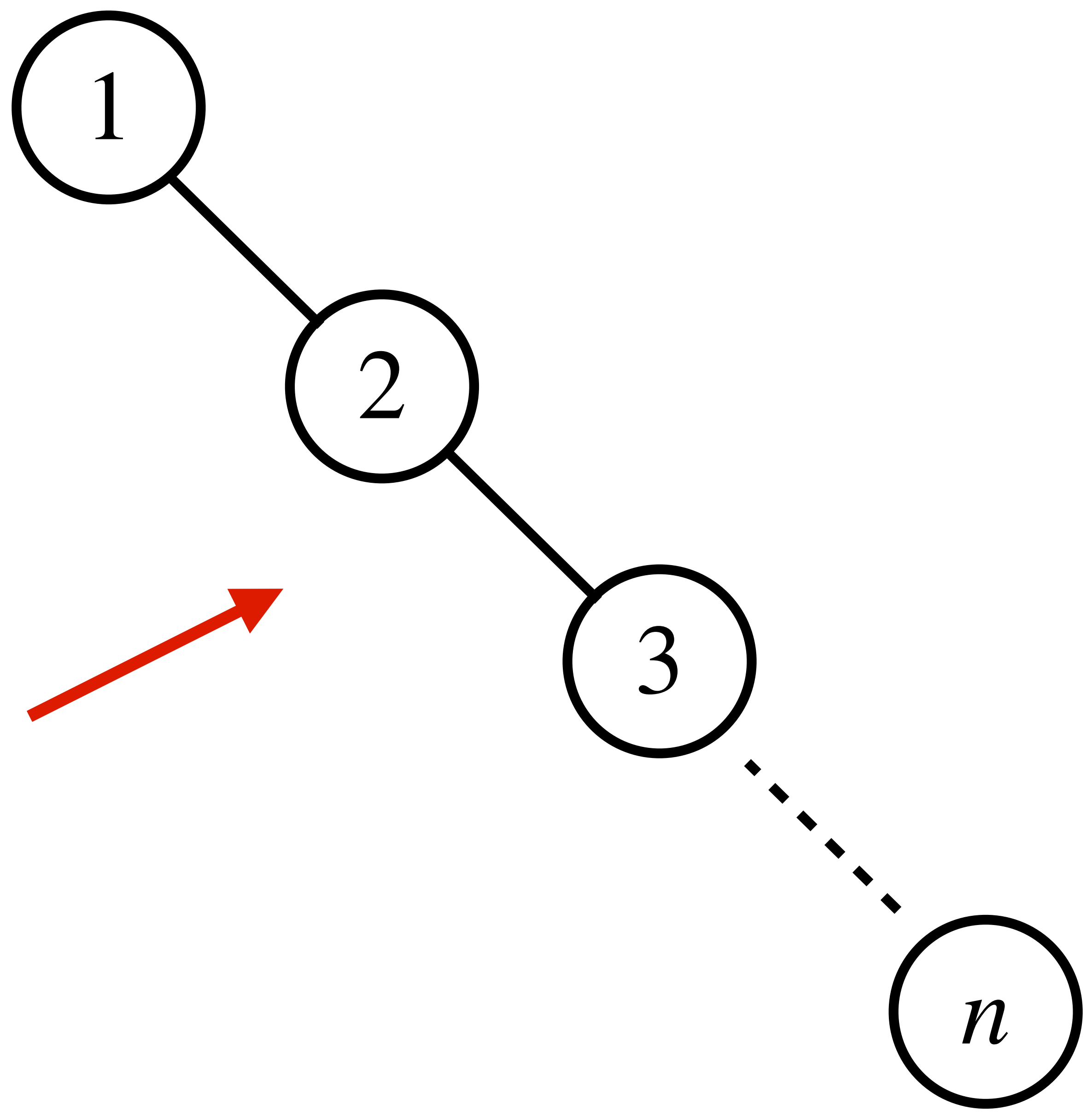


How to Restrict Height in BSTs?



Idea: We can restrict the maximum height by keeping the BST balanced.

How to Restrict Height in BSTs?



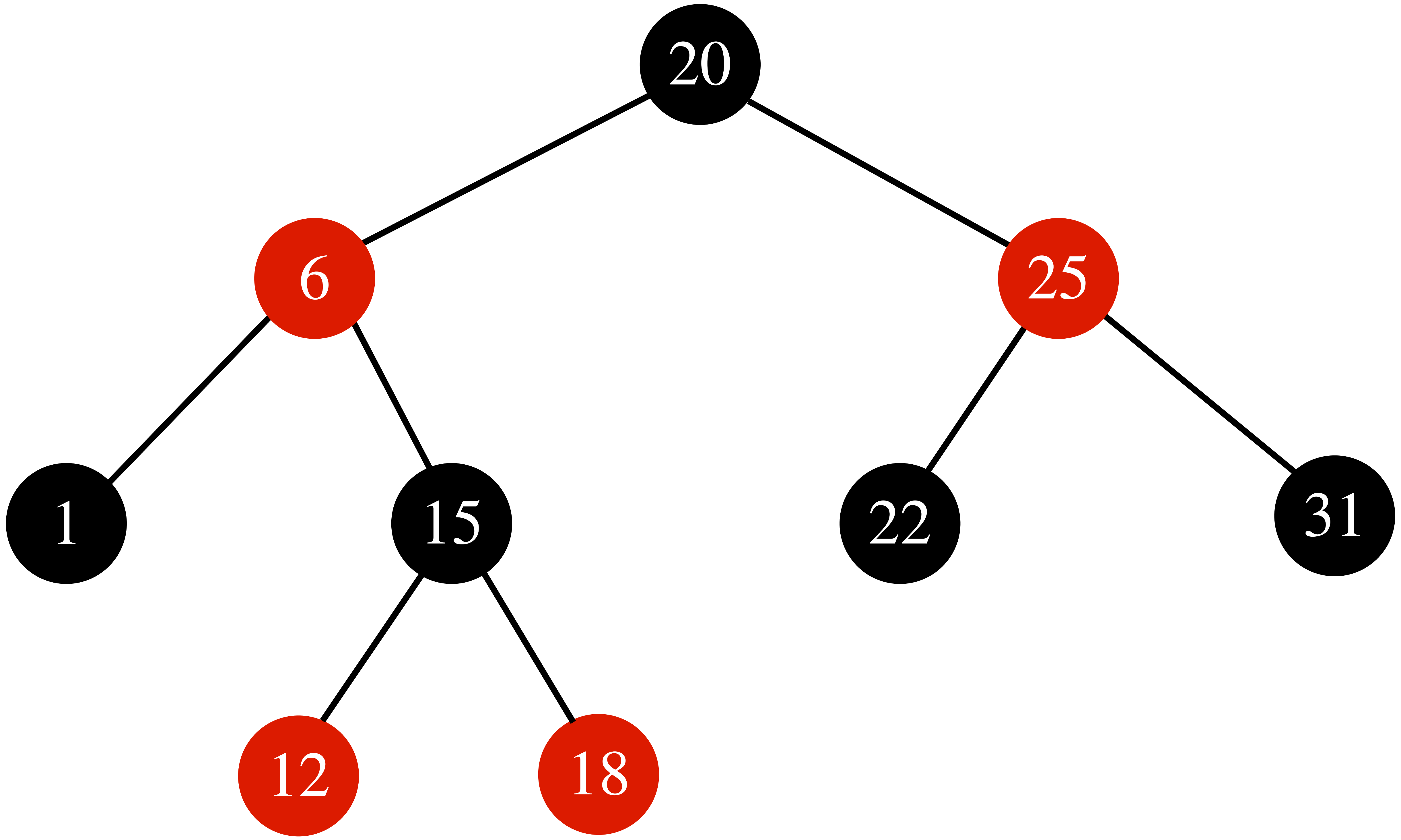
Skewness causes large height

Idea: We can restrict the maximum height by keeping the BST balanced.

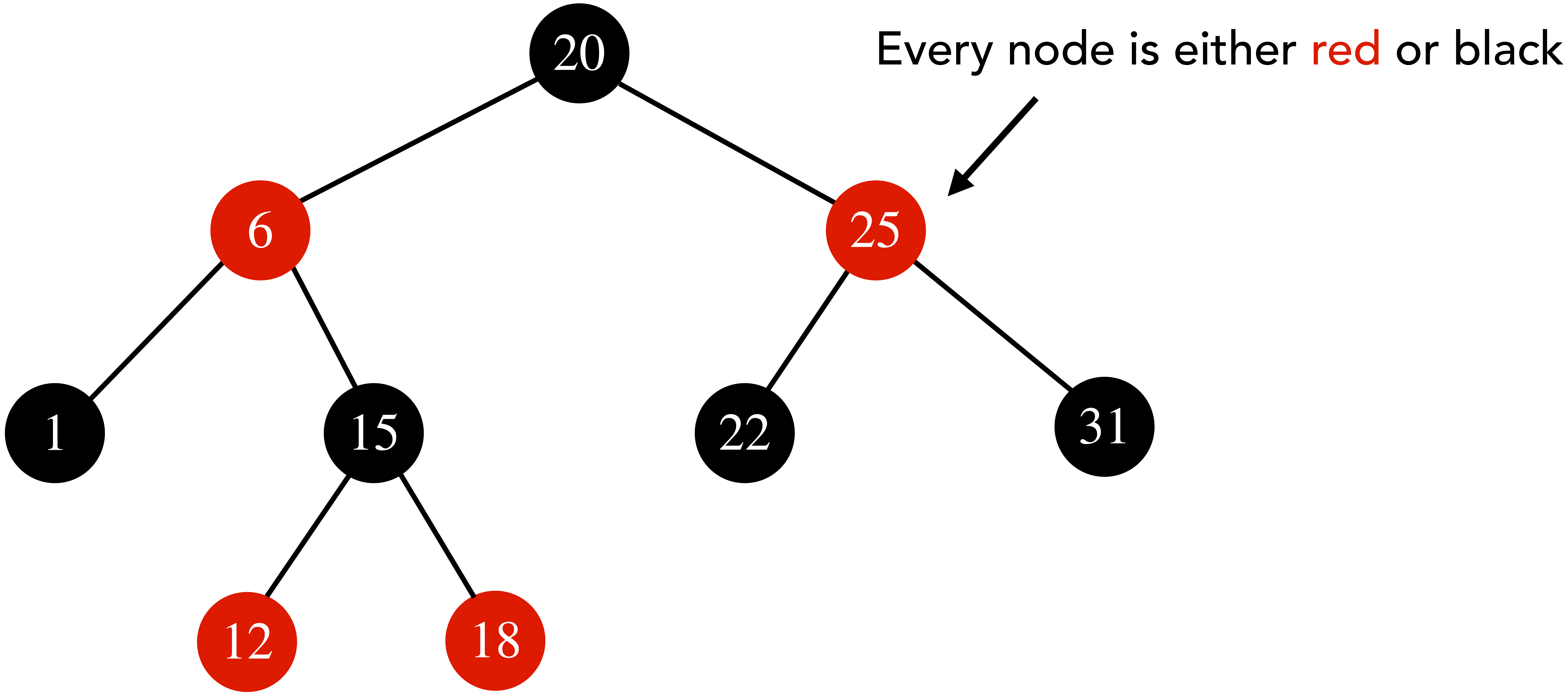
For any node x, number of nodes in left-subtree(x) should not be too small or large than number of nodes in right-subtree(x)

RB-Trees: How do they look like?

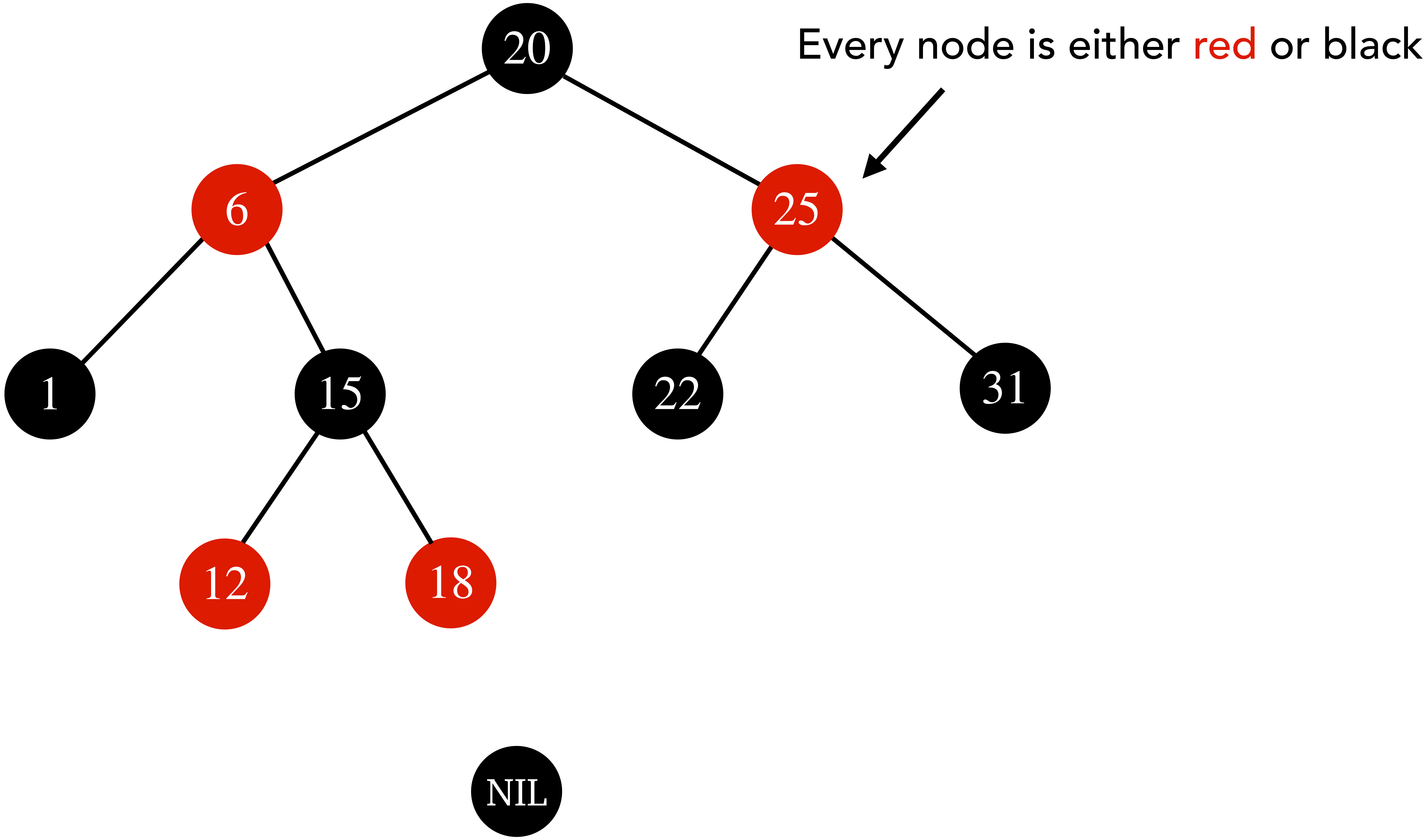
RB-Trees: How do they look like?



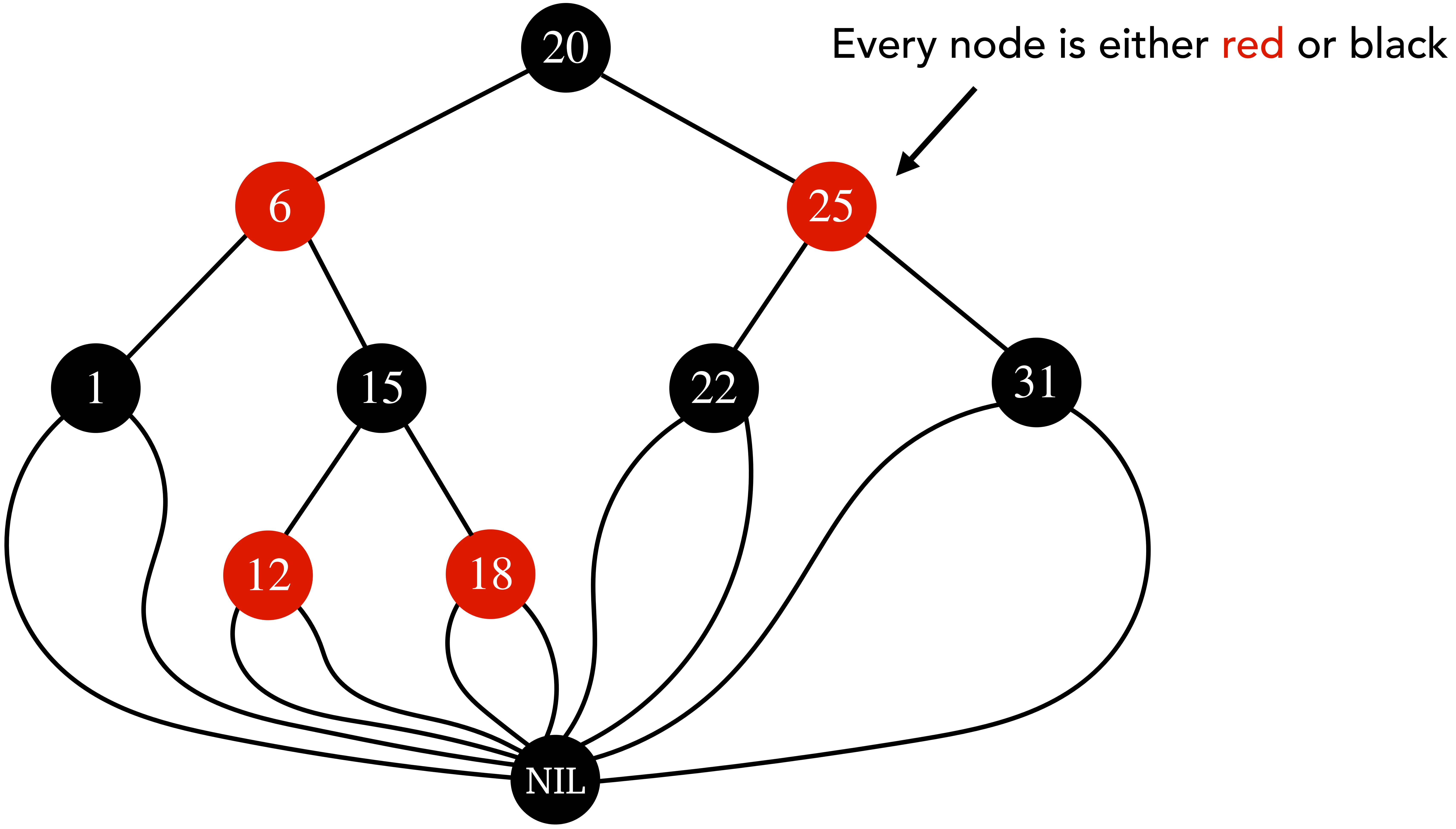
RB-Trees: How do they look like?



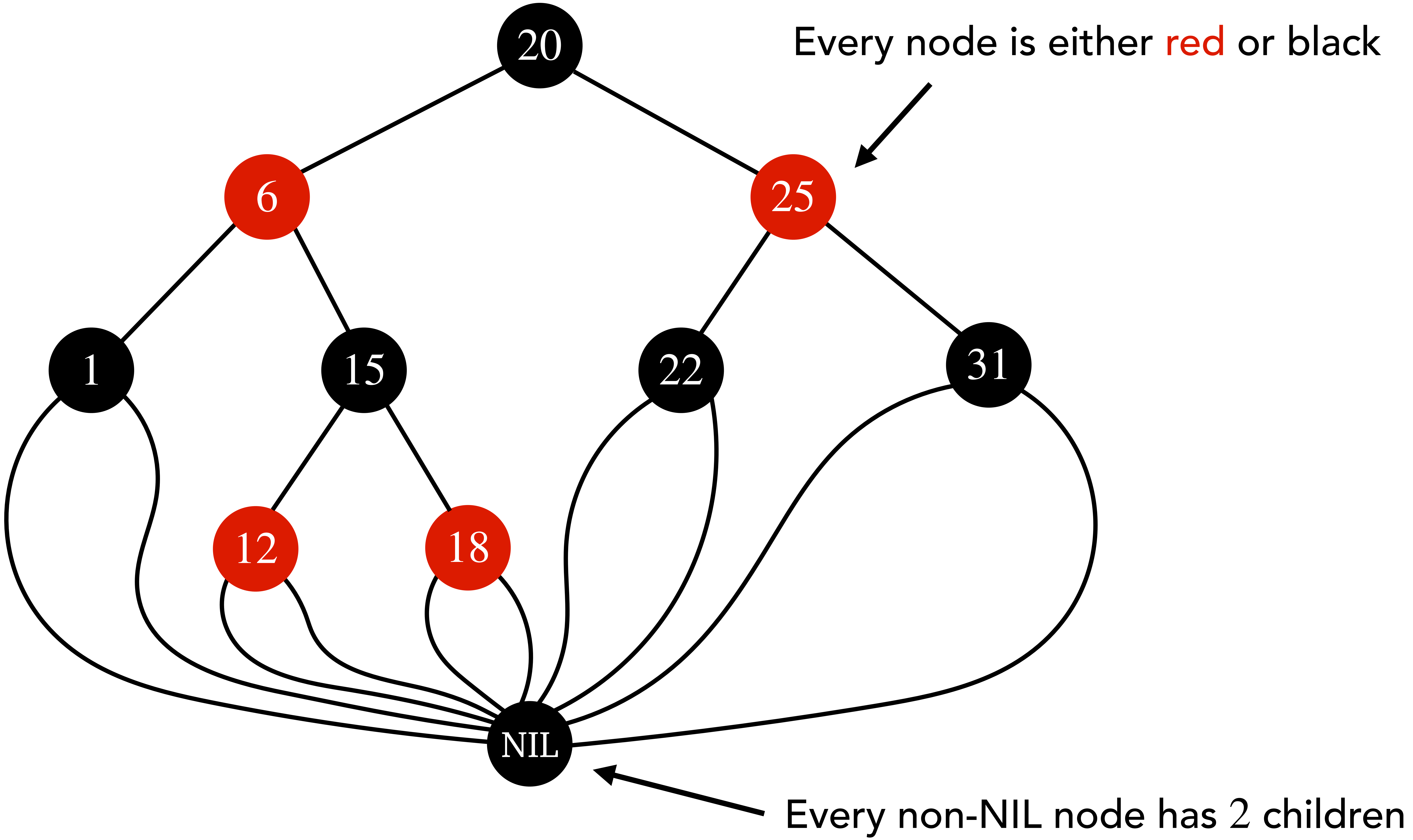
RB-Trees: How do they look like?



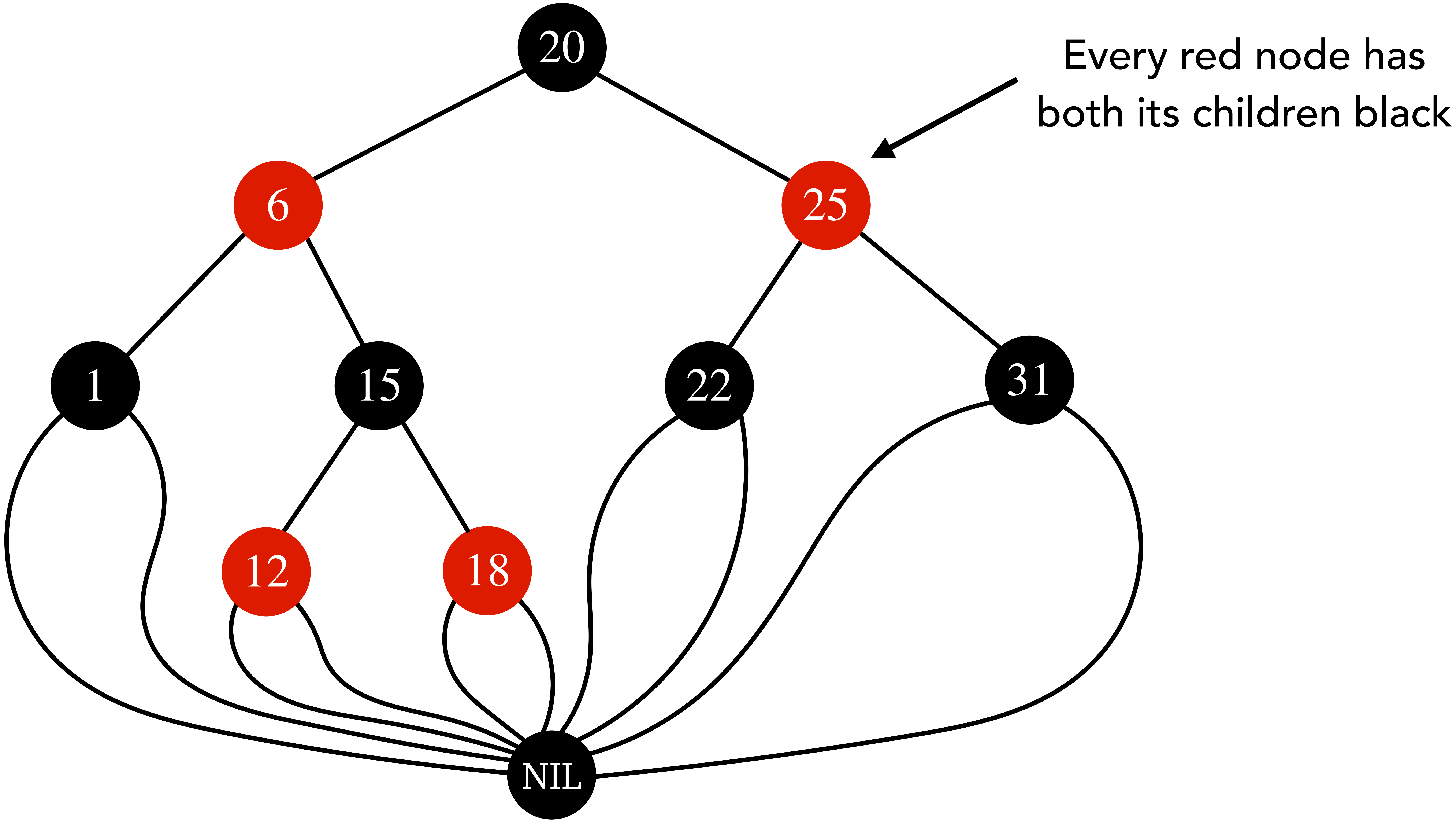
RB-Trees: How do they look like?



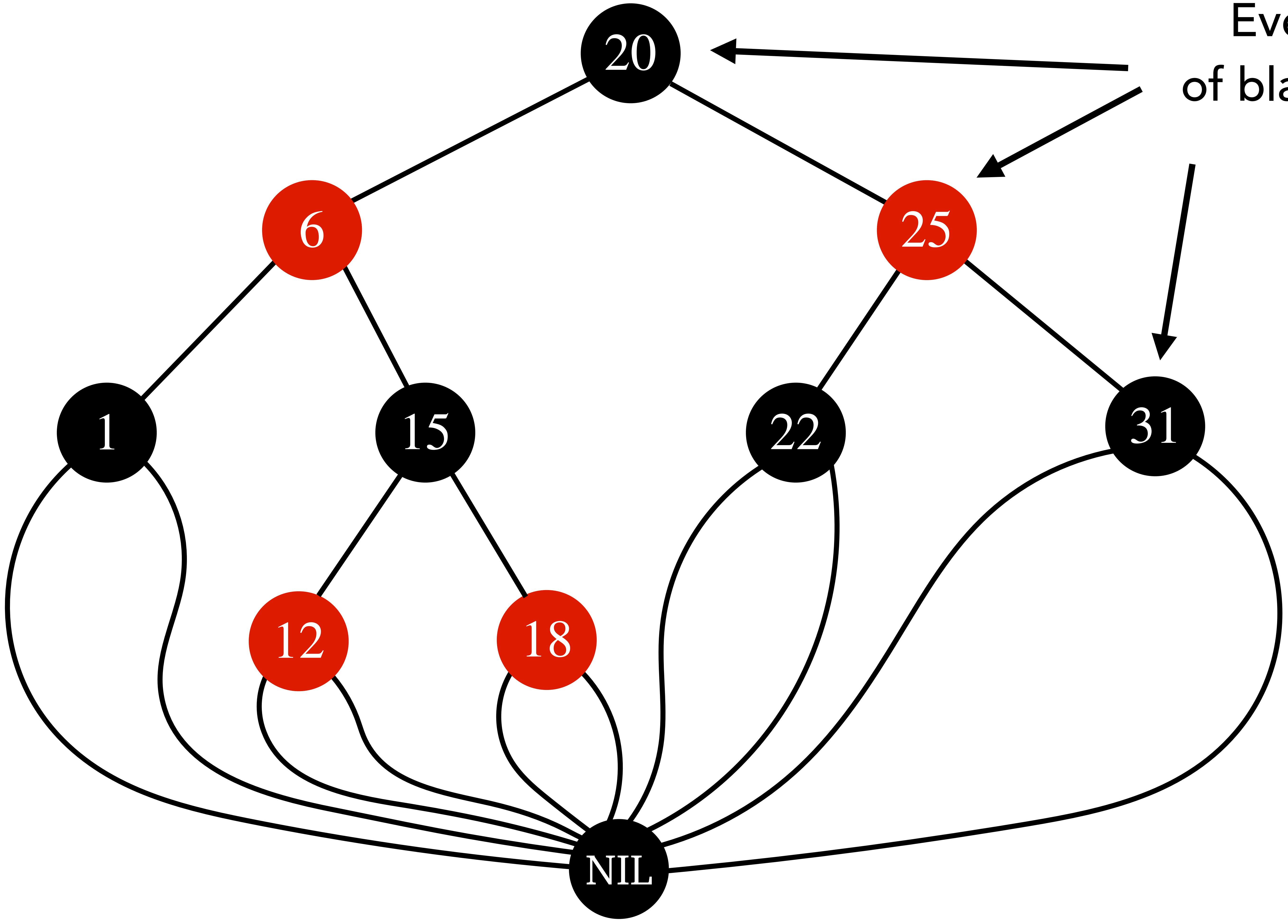
RB-Trees: How do they look like?



RB-Trees: How do they look like?



RB-Trees: How do they look like?



Every node has same number of black nodes on paths to leaves.

RB-Trees: Formal Description

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

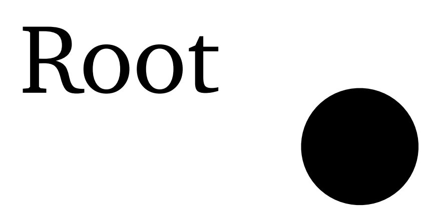
- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same number of **black** nodes.

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same number of **black** nodes.

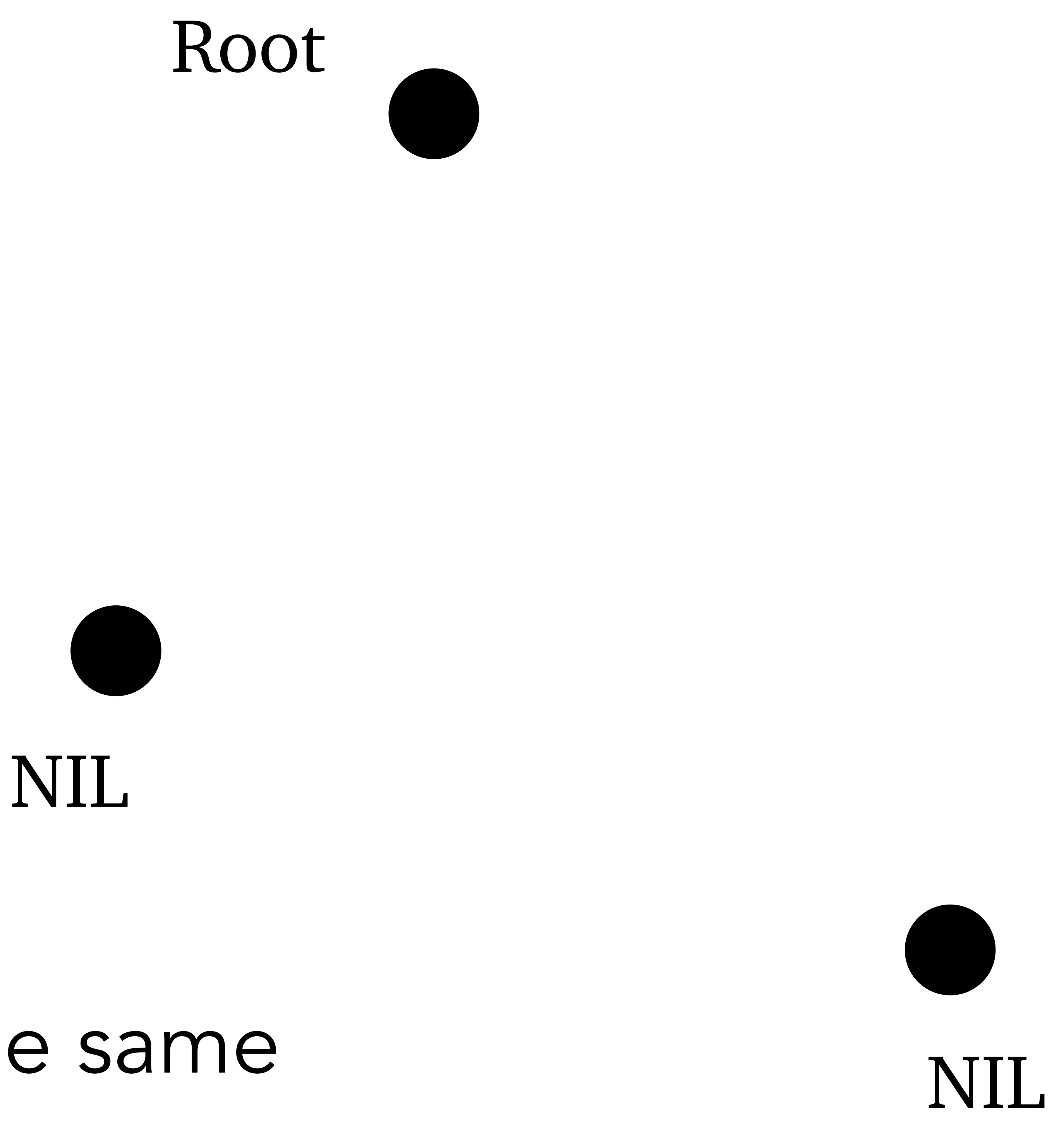
RB-Trees: Formal Description



RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same number of **black** nodes.

RB-Trees: Formal Description



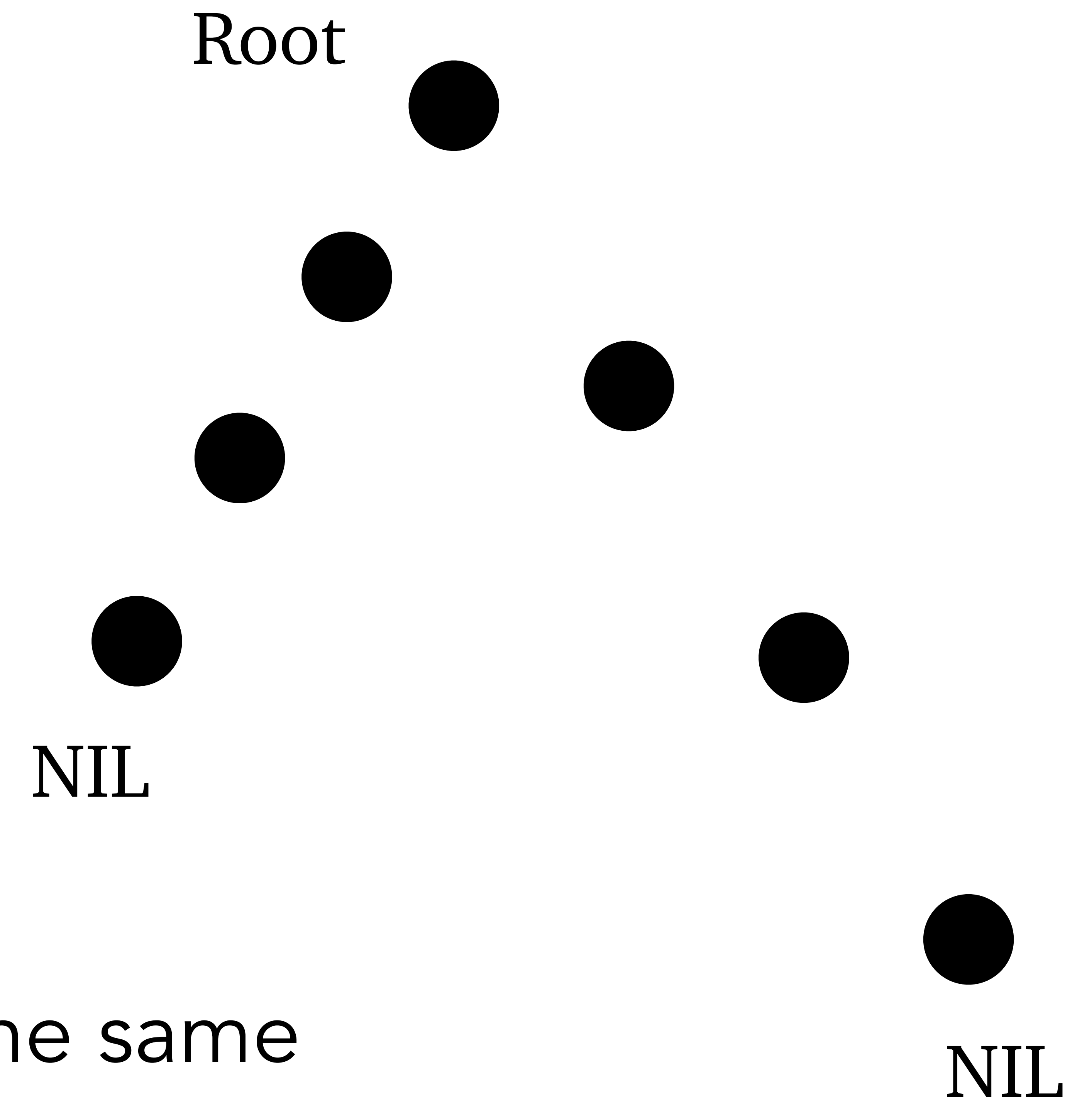
RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either **red** or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is **red**, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same number of **black** nodes.

RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

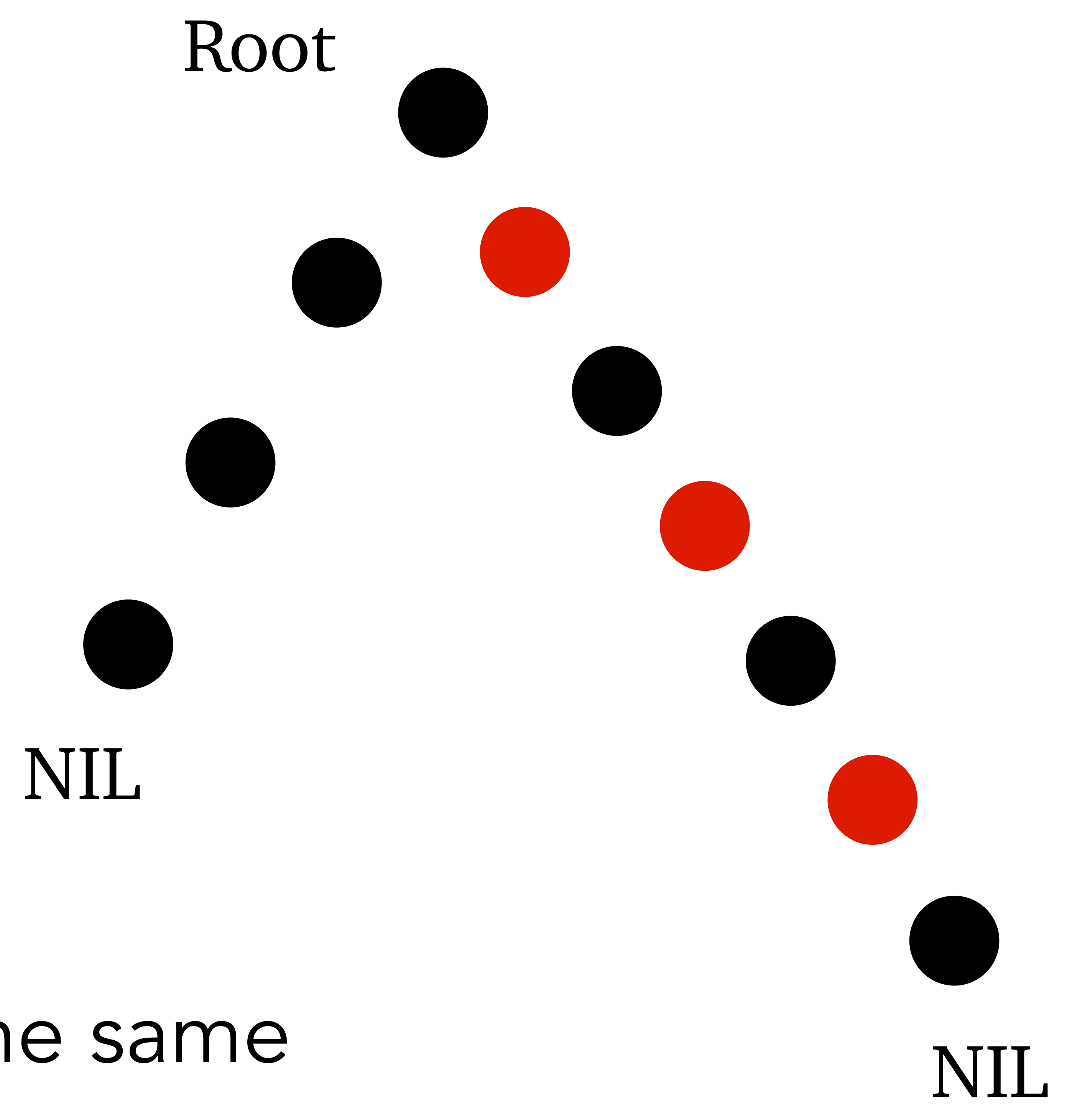
- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same number of **black** nodes.



RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same number of **black** nodes.



RB-Trees: Formal Description

RB-trees are BSTs which satisfy the following properties:

- Every node has a colour either red or **black**.
- Root is **black**.
- Leaf nodes are NIL nodes which are **black** in colour.
- If a node is red, then both its children are **black**.
- For every node, all the paths from the node to leaves contain the same number of **black** nodes.

Both these properties ensure that no path from root to a leaf is more than twice as long as any other.