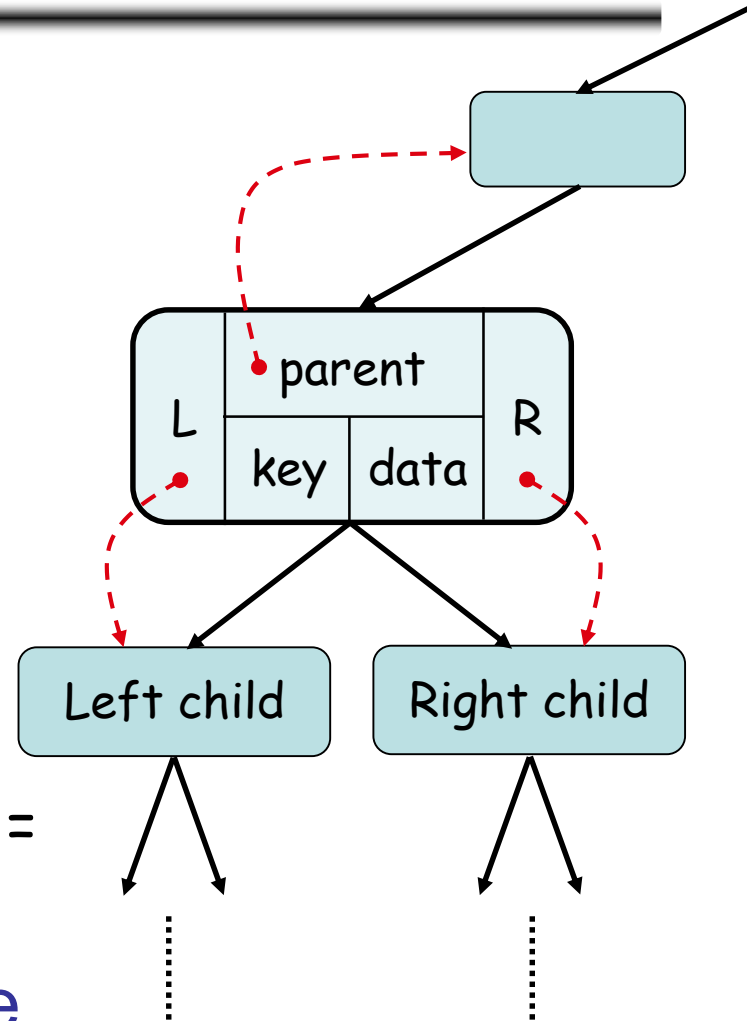


Binary Search Tree

Romi Fadillah Rahmat

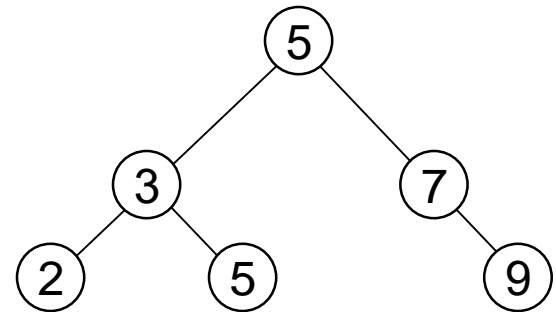
Binary Search Trees

- Tree representation:
 - A linked data structure in which each node is an object
- Node representation:
 - Key field
 - Satellite data
 - Left: pointer to left child
 - Right: pointer to right child
 - p: pointer to parent ($p[\text{root}[T]] = \text{NIL}$)
- Satisfies the binary-search-tree property !!



Binary Search Tree Property

- Binary search tree property:
 - If y is in left subtree of x ,
then $\text{key}[y] \leq \text{key}[x]$
 - If y is in right subtree of x ,
then $\text{key}[y] \geq \text{key}[x]$

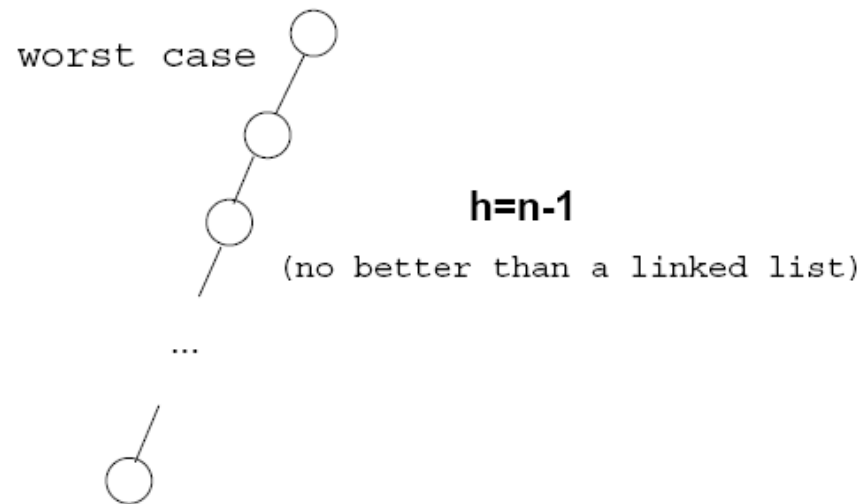


Binary Search Trees

- Support many dynamic set operations
 - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees
 - On average: $\Theta(\lg n)$
 - The expected height of the tree is $\lg n$
 - In the worst case: $\Theta(n)$
 - The tree is a linear chain of n nodes

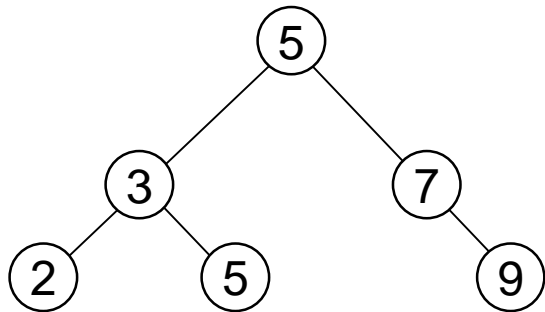
Worst Case

- If the tree is very **unbalanced**, then running time will be $\Theta(n)$



Traversing a Binary Search Tree

- **Inorder tree walk:**
 - Root is printed between the values of its left and right subtrees: **left, root, right**
 - Keys are printed in **sorted order**
- **Preorder tree walk:**
 - root printed first: **root, left, right**
- **Postorder tree walk:**
 - root printed last: **left, right, root**



Inorder: 2 3 5 5 7 9

Preorder: 5 3 2 5 7 9

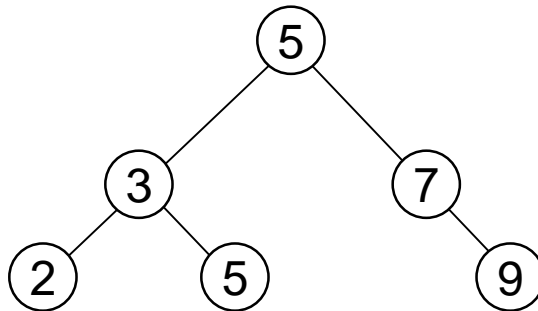
Postorder: 2 5 3 9 7 5

Traversing a Binary Search Tree

Alg: INORDER-TREE-WALK(x)

1. **if** $x \neq \text{NIL}$
2. **then** INORDER-TREE-WALK (left [x])
3. print key [x]
4. INORDER-TREE-WALK (right [x])

• *E.g.:*



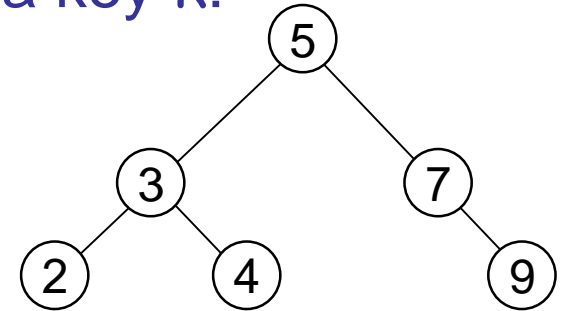
Output: 2 3 5 5 7 9

- Running time:
 - $\Theta(n)$, where n is the size of the tree rooted at x

Searching for a Key

- Given a pointer to the root of a tree and a key k :

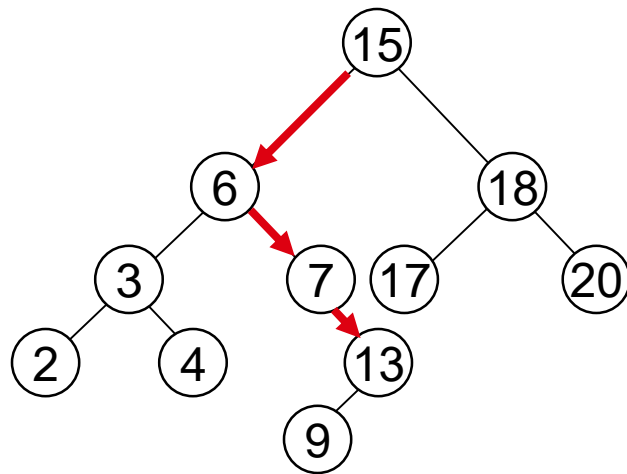
- Return a pointer to a node with key k if one exists
- Otherwise return NIL



- Idea

- Starting at the root: trace down a path by comparing k with the key of the current node:
 - If the keys are equal: we have found the key
 - If $k < \text{key}[x]$ search in the left subtree of x
 - If $k > \text{key}[x]$ search in the right subtree of x

Example: TREE-SEARCH

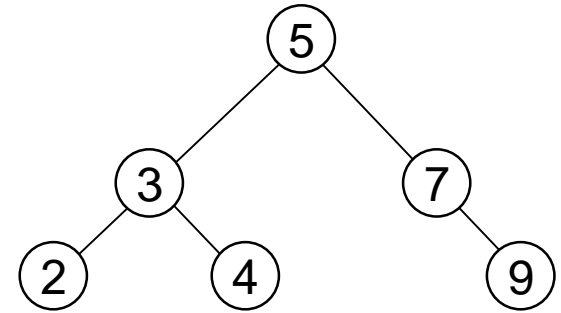


- Search for key 13:
 - $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

Searching for a Key

Alg: TREE-SEARCH(x, k)

1. **if** $x = \text{NIL}$ or $k = \text{key}[x]$
2. **then return** x
3. **if** $k < \text{key}[x]$
4. **then return** TREE-SEARCH(left $[x], k$)
5. **else return** TREE-SEARCH(right $[x], k$)



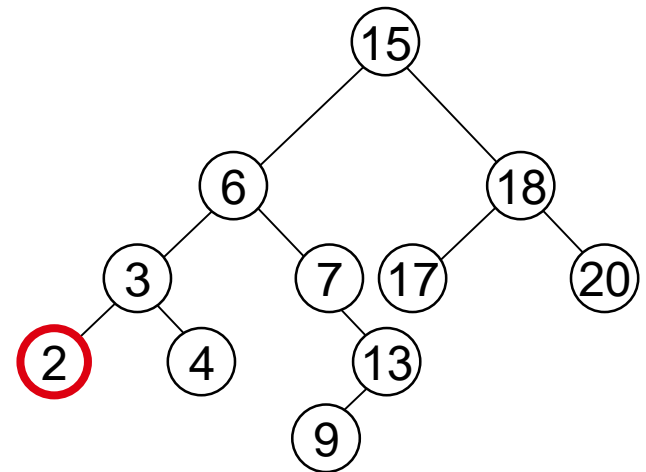
Running Time: $O(h)$,
 h – the height of the tree

Finding the Minimum in a Binary Search Tree

- Goal: find the minimum value in a BST
 - Following left child pointers from the root, until a NIL is encountered

Alg: TREE-MINIMUM(x)

1. **while** left [x] \neq NIL
2. **do** $x \leftarrow$ left [x]
3. **return** x



Minimum = 2

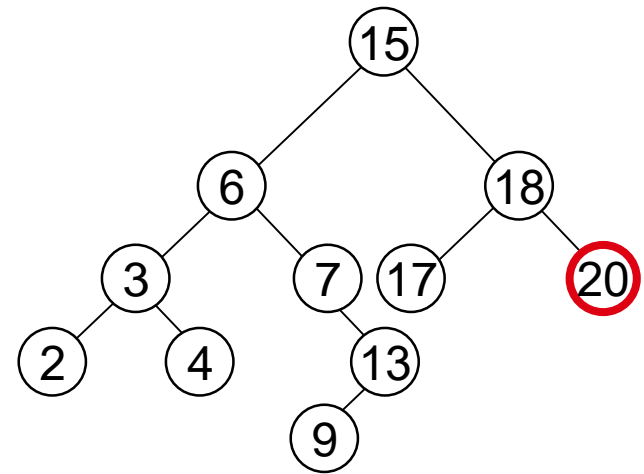
Running time: $O(h)$, h – height of tree

Finding the Maximum in a Binary Search Tree

- Goal: find the maximum value in a BST
 - Following right child pointers from the root, until a NIL is encountered

Alg: TREE-MAXIMUM(x)

1. **while** right [x] \neq NIL
2. **do** $x \leftarrow$ right [x]
3. **return** x



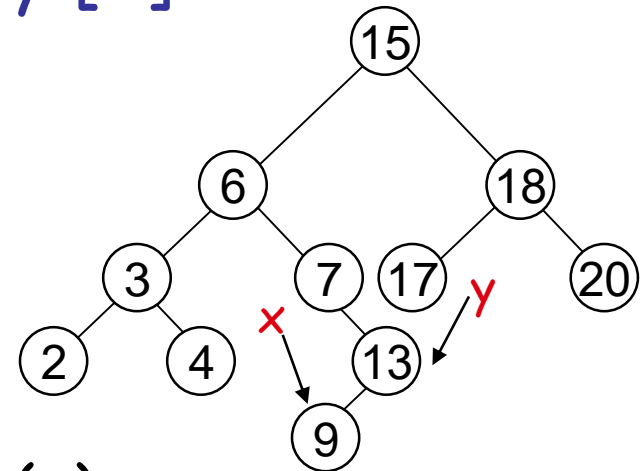
Maximum = 20

- Running time: $O(h)$, h – height of tree

Successor

Def: $\text{successor}(x) = y$, such that $\text{key}[y]$ is the smallest key $> \text{key}[x]$

- *E.g.:* $\text{successor}(15) = 17$
 $\text{successor}(13) = 15$
 $\text{successor}(9) = 13$

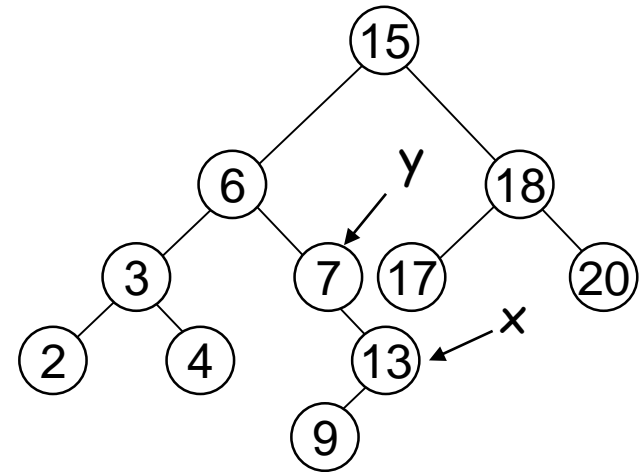


- **Case 1: $\text{right}(x)$ is non empty**
 - $\text{successor}(x)$ = the minimum in $\text{right}(x)$
- **Case 2: $\text{right}(x)$ is empty**
 - go up the tree until the current node is a left child:
 $\text{successor}(x)$ is the parent of the current node
 - if you cannot go further (and you reached the root):
 x is the largest element

Finding the Successor

Alg: TREE-SUCCESSOR(x)

1. **if** right [x] \neq NIL
2. **then return** TREE-MINIMUM(right [x])
3. $y \leftarrow p[x]$
4. **while** $y \neq$ NIL and $x =$ right [y]
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

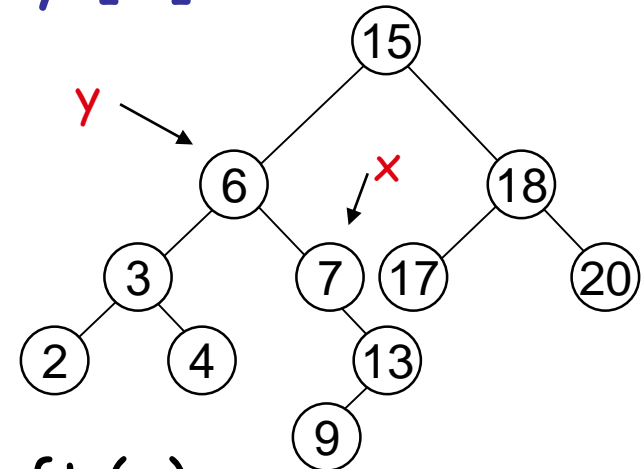


Running time: $O(h)$, h – height of the tree

Predecessor

Def: $\text{predecessor}(x) = y$, such that key $[y]$ is the biggest key $< \text{key}[x]$

- *E.g.:* $\text{predecessor}(15) = 13$
 $\text{predecessor}(9) = 7$
 $\text{predecessor}(7) = 6$



- **Case 1: $\text{left}(x)$ is non empty**
 - $\text{predecessor}(x) = \text{the maximum in } \text{left}(x)$
- **Case 2: $\text{left}(x)$ is empty**
 - go up the tree until the current node is a right child:
 $\text{predecessor}(x)$ is the parent of the current node
 - if you cannot go further (and you reached the root):
 x is the smallest element

Insertion

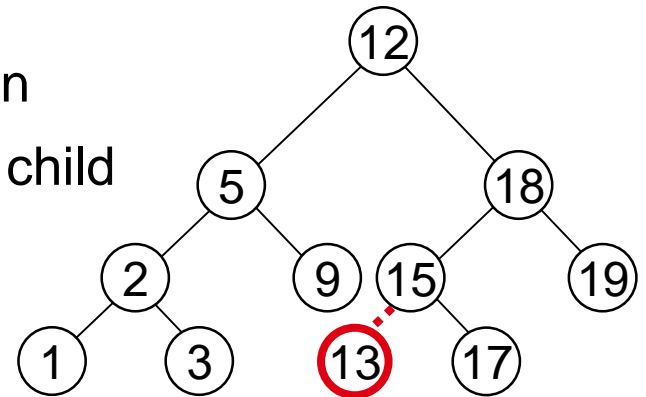
- Goal:

- Insert value v into a binary search tree

- Idea:

- If $\text{key}[x] < v$ move to the right child of x ,
else move to the left child of x
- When x is NIL, we found the correct position
- If $v < \text{key}[y]$ insert the new node as y 's left child
else insert it as y 's right child

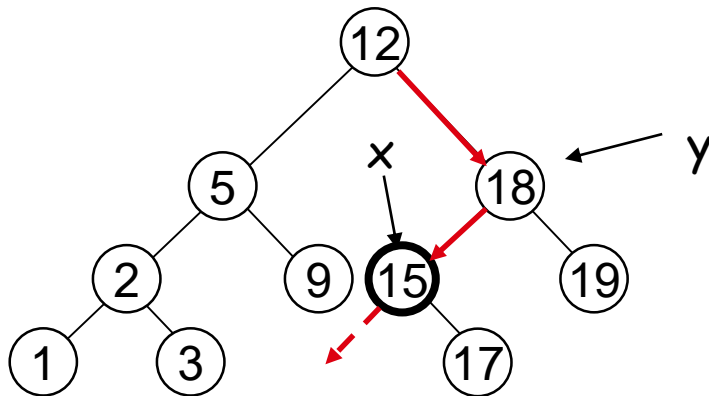
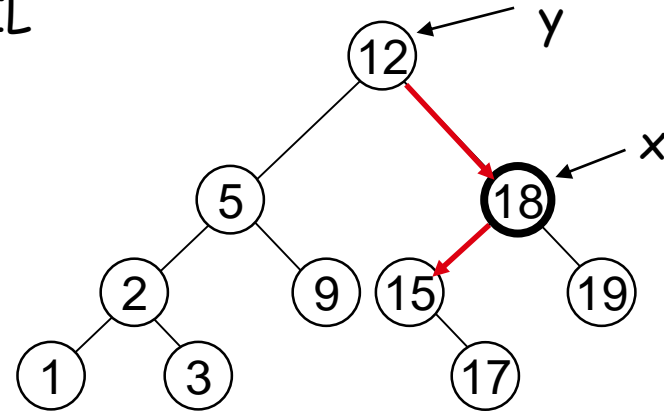
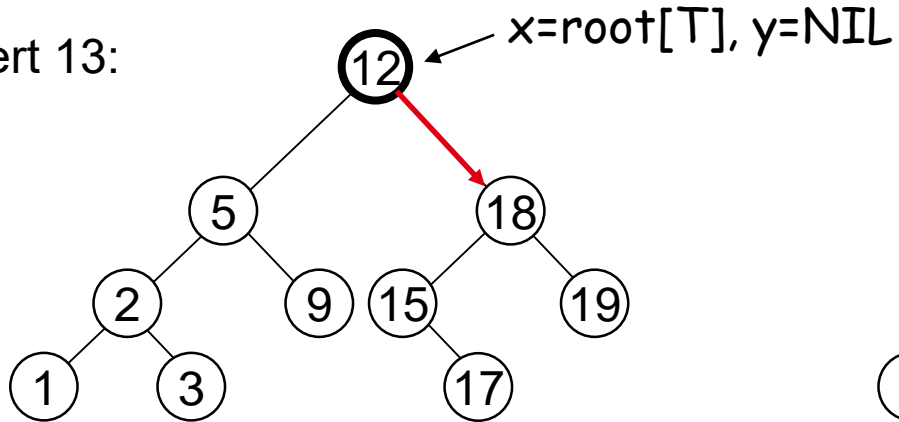
Insert value 13



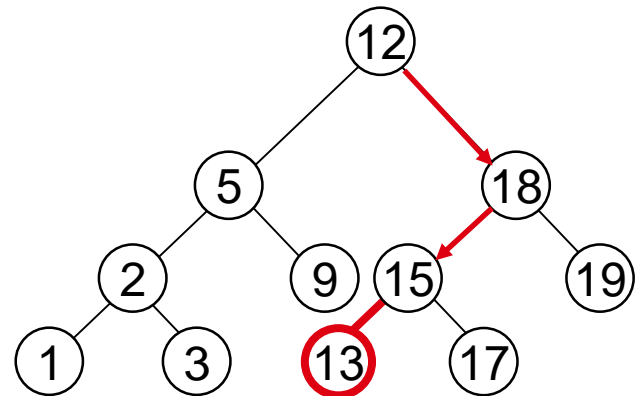
- Beginning at the root, go down the tree and maintain:
 - Pointer x : traces the downward path (current node)
 - Pointer y : parent of x (“trailing pointer”)

Example: TREE-INSERT

Insert 13:

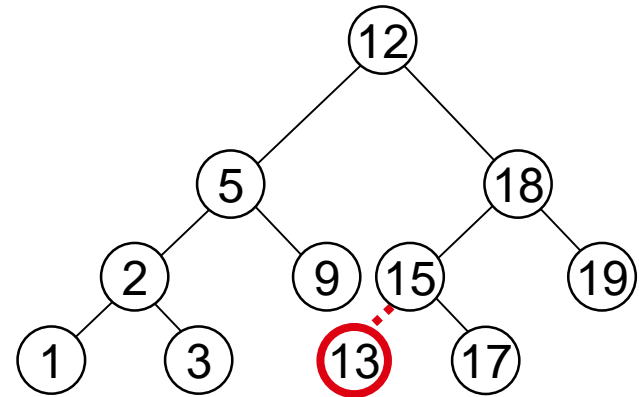


$x = \text{NIL}$
 $y = 15$



Alg: TREE-INSERT(T, z)

1. $y \leftarrow \text{NIL}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{NIL}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{NIL}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

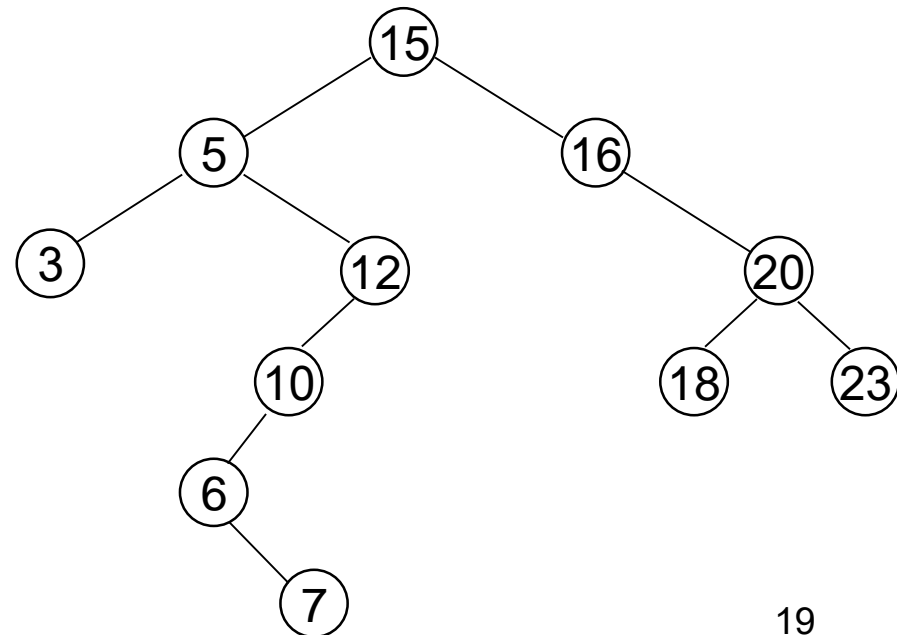
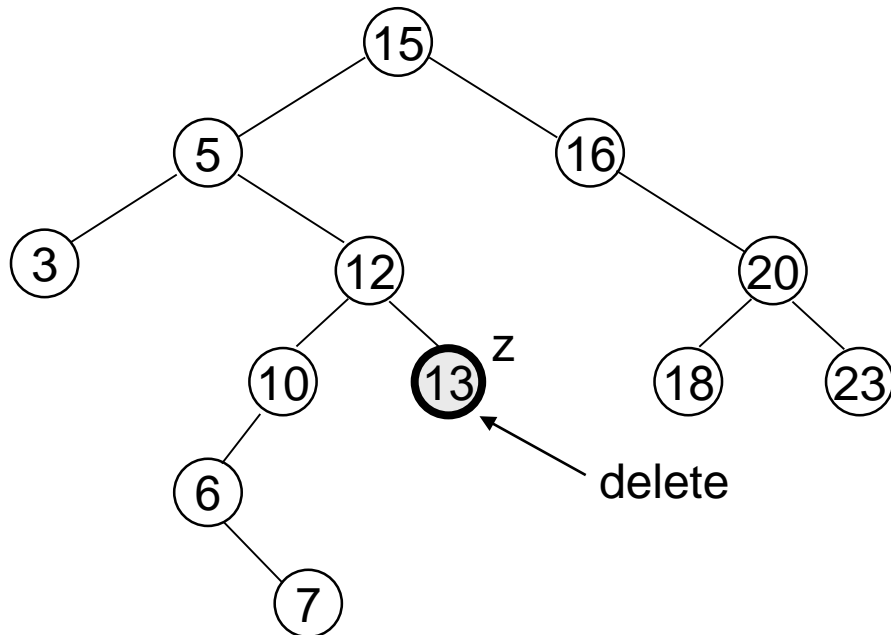


▷ Tree T was empty

Running time: $O(h)$

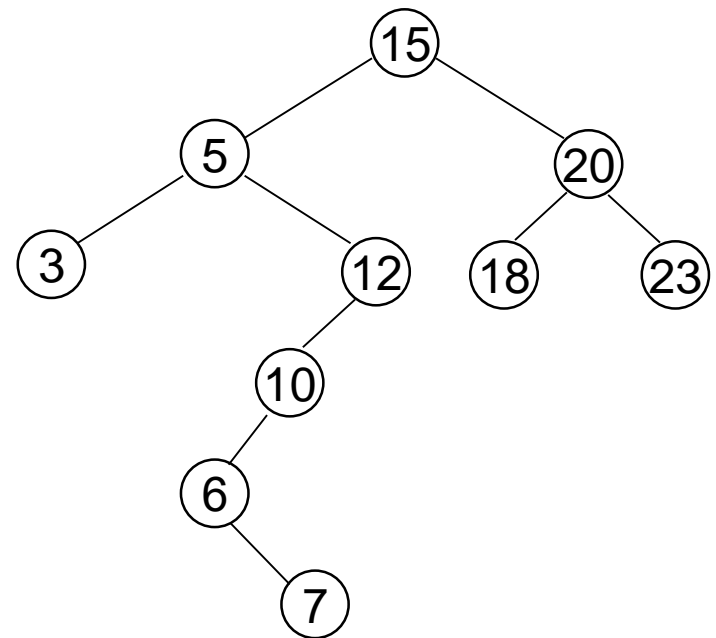
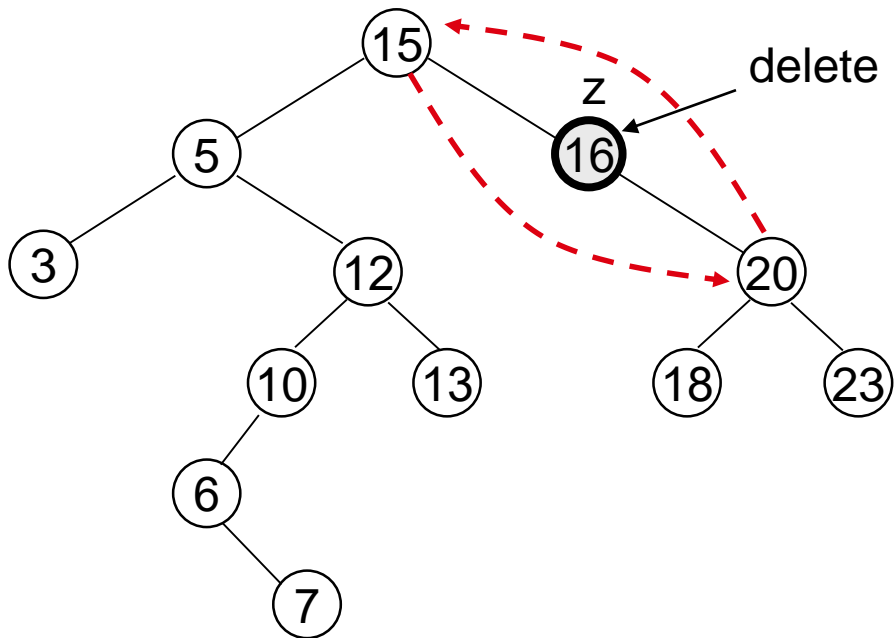
Deletion

- Goal:
 - Delete a given node z from a binary search tree
- Idea:
 - **Case 1:** z has no children
 - Delete z by making the parent of z point to NIL



Deletion

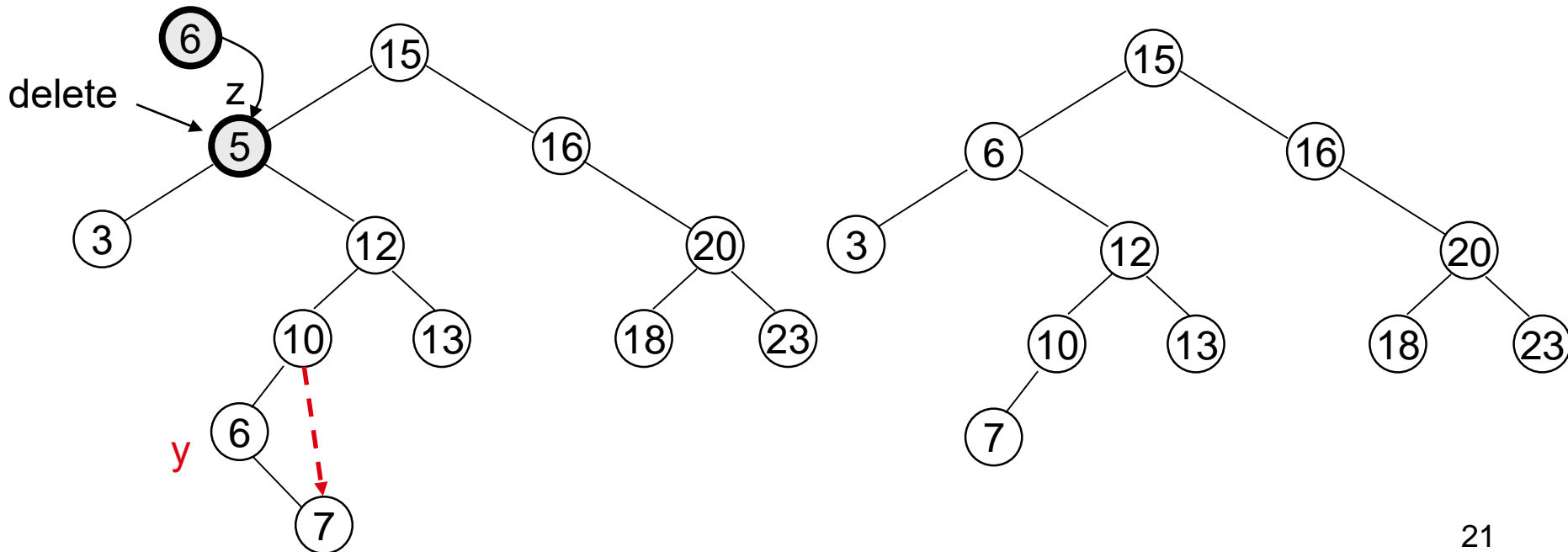
- **Case 2: z has one child**
 - Delete z by making the parent of z point to z's child, instead of to z



Deletion

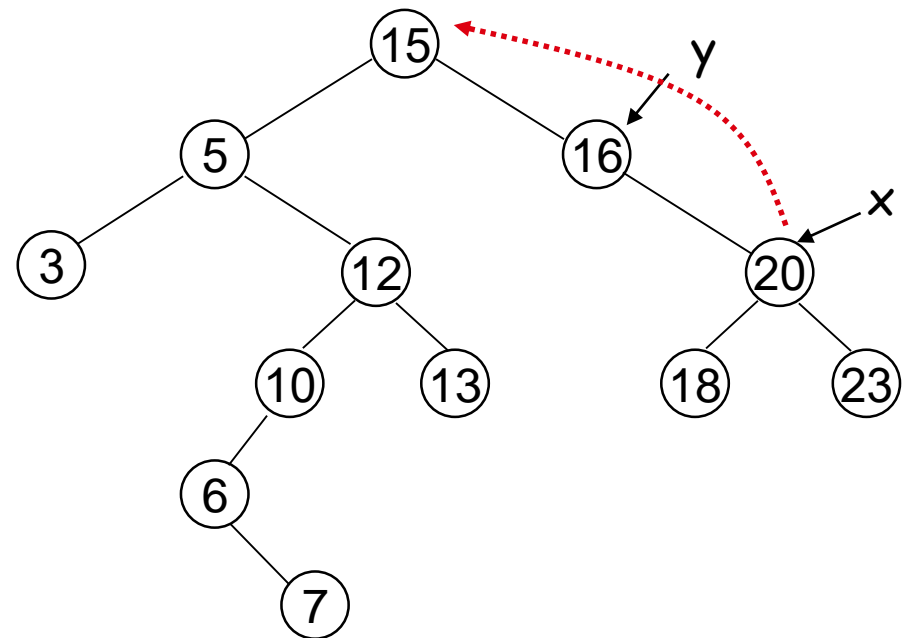
- **Case 3: z has two children**

- z's successor (y) is the minimum node in z's right subtree
- y has either no children or one right child (but no left child)
- Delete y from the tree (via Case 1 or 2)
- Replace z's key and satellite data with y's.



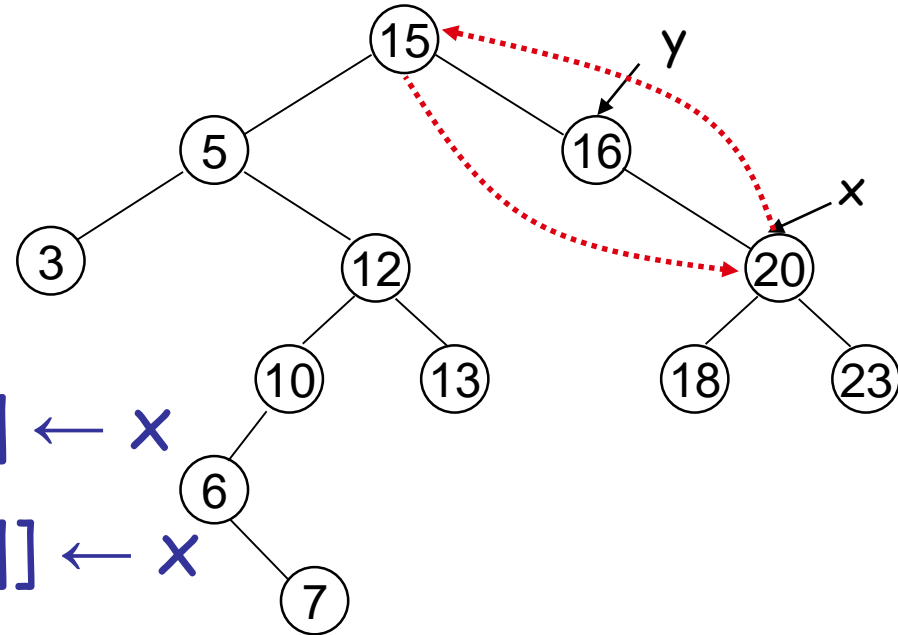
TREE-DELETE(T, z)

1. **if** $\text{left}[z] = \text{NIL}$ or $\text{right}[z] = \text{NIL}$
2. **then** $y \leftarrow z$ z has one child
3. **else** $y \leftarrow \text{TREE-SUCCESSOR}(z)$ z has 2 children
4. **if** $\text{left}[y] \neq \text{NIL}$
5. **then** $x \leftarrow \text{left}[y]$
6. **else** $x \leftarrow \text{right}[y]$
7. **if** $x \neq \text{NIL}$
8. **then** $p[x] \leftarrow p[y]$



TREE-DELETE(T, z) – cont.

9. **if** $p[y] = \text{NIL}$
10. **then** $\text{root}[T] \leftarrow x$
11. **else if** $y = \text{left}[p[y]]$
12. **then** $\text{left}[p[y]] \leftarrow x$
13. **else** $\text{right}[p[y]] \leftarrow x$
14. **if** $y \neq z$
15. **then** $\text{key}[z] \leftarrow \text{key}[y]$
16. copy y 's satellite data into z
17. **return** y



Running time: $O(h)$

Binary Search Trees - Summary

- Operations on binary search trees:
 - SEARCH $O(h)$
 - PREDECESSOR $O(h)$
 - SUCCESSION $O(h)$
 - MINIMUM $O(h)$
 - MAXIMUM $O(h)$
 - INSERT $O(h)$
 - DELETE $O(h)$
- These operations are fast if the height of the tree is **small** – otherwise their performance is similar to that of a linked list