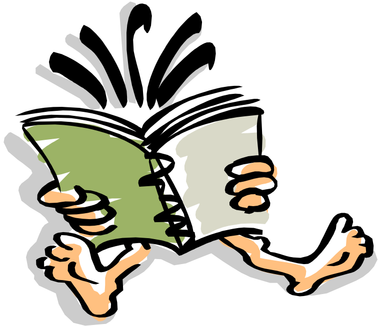
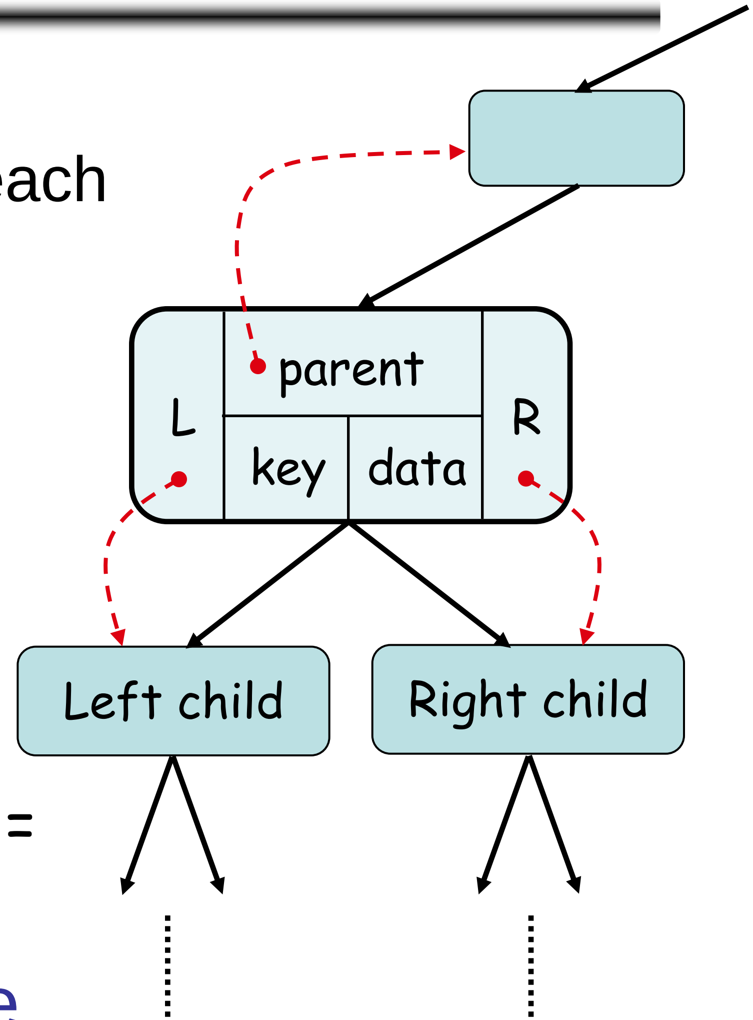
Binary Search Trees



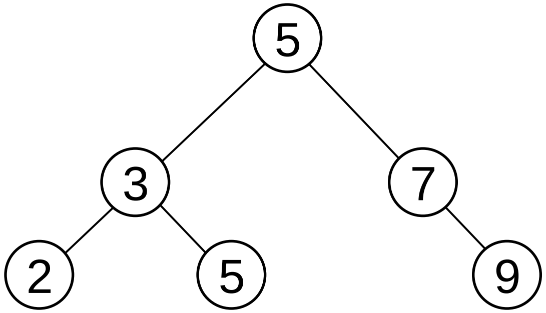
Binary Search Trees

- 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 [root [T]] = NIL)
- Satisfies the binary-search-tree property !!



Binary Search Tree Property

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

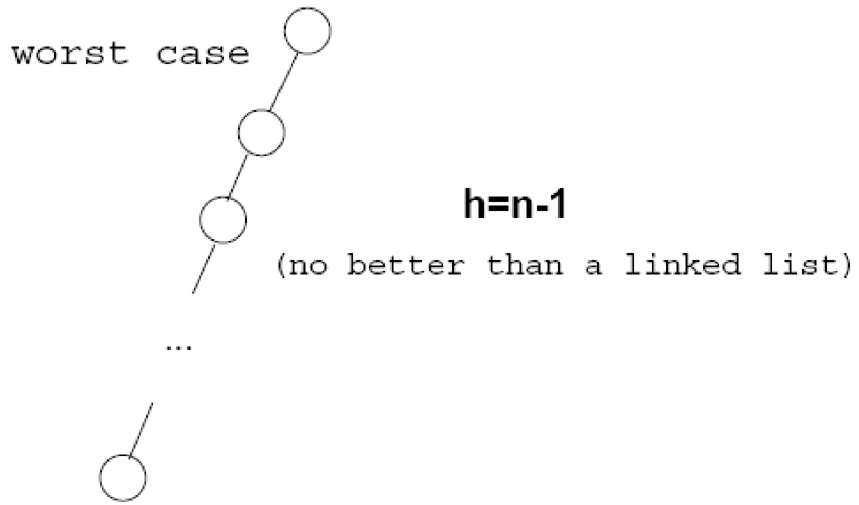


Binary Search Trees

- 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

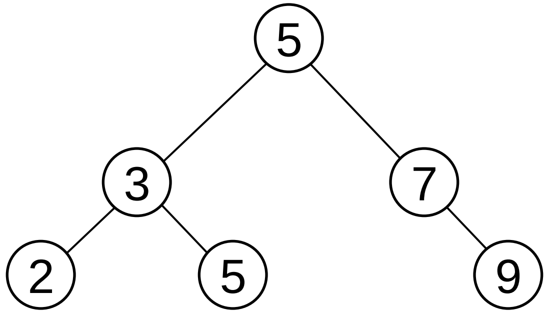
Worst Case

- 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

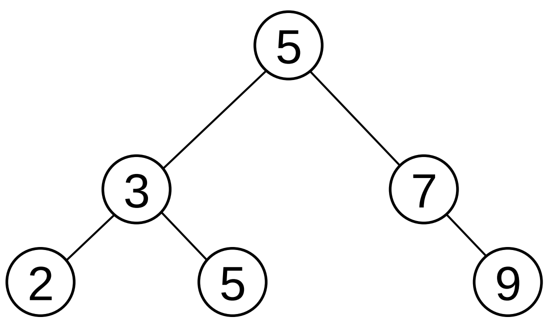
Postorder: 2 5 3 9 7 5

Traversing a Binary Search Tree

Alg: INORDER-TREE-WALK(x)

```
if x ≠ NIL
   then INORDER-TREE-WALK ( left [x] )
        print key [x]
        INORDER-TREE-WALK ( right [x] )
```

- *E.g.:*

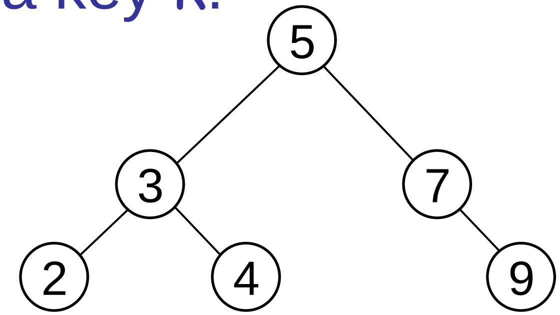


Output: 2 3 5 5 7 9

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

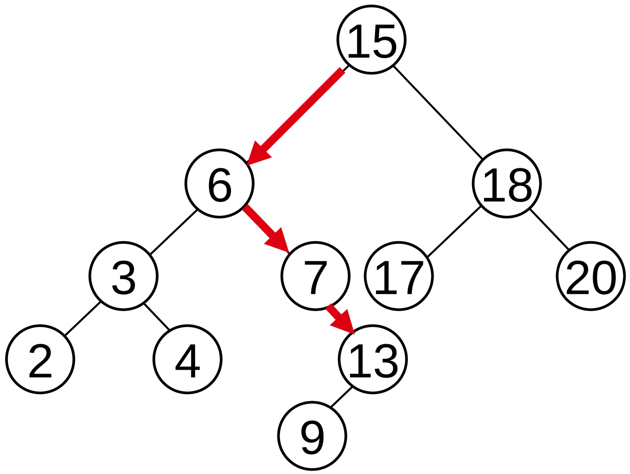
Searching for a Key

- 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 $k < key[x]$ search in the left subtree of x
    - If $k > key[x]$ search in the right subtree of x

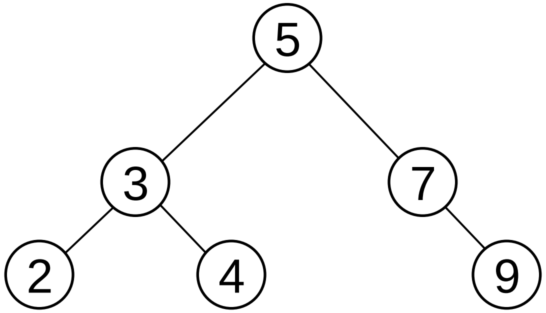
Example: TREE-SEARCH



- Search for key 13:
  - 15 → 6 → 7 → 13

Searching for a Key

Alg: TREE-SEARCH(x, k)



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

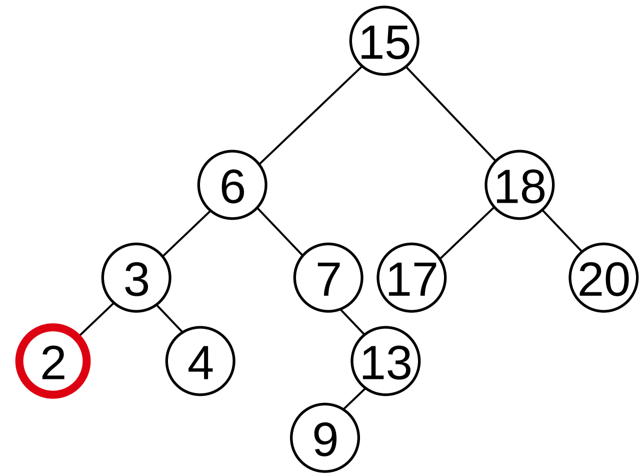
Running Time: *O* (h),
h – the height of the tree

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

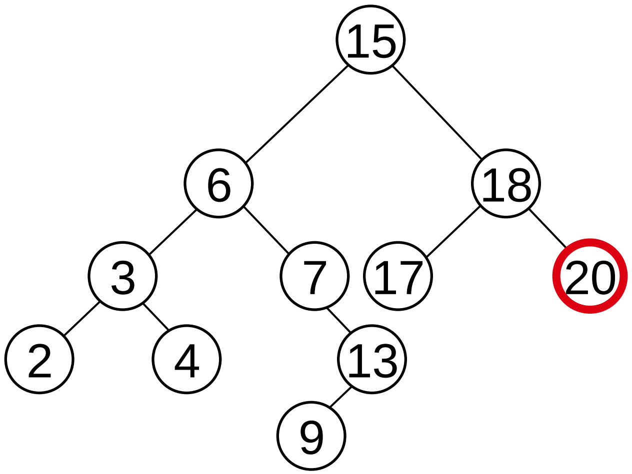
Running time: *O(h), h* – height of tree

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

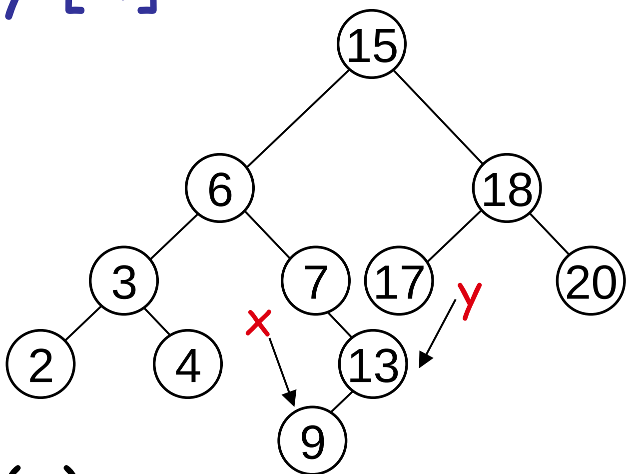
Successor

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

- *E.g.:* $successor(15) = 17$

  $successor(13) = 15$

  $successor(9) = 13$

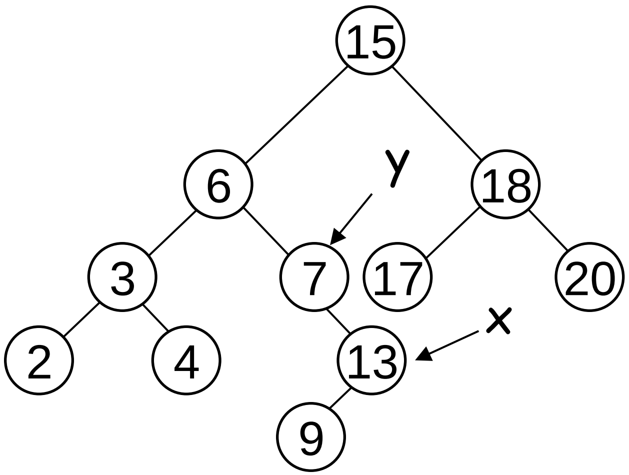


- Case 1: right (x) is non empty
  - $successor(x)$ = the minimum in right (x)
- Case 2: right (x) is empty
  - go up the tree until the current node is a left child: $successor(x)$ is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element

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

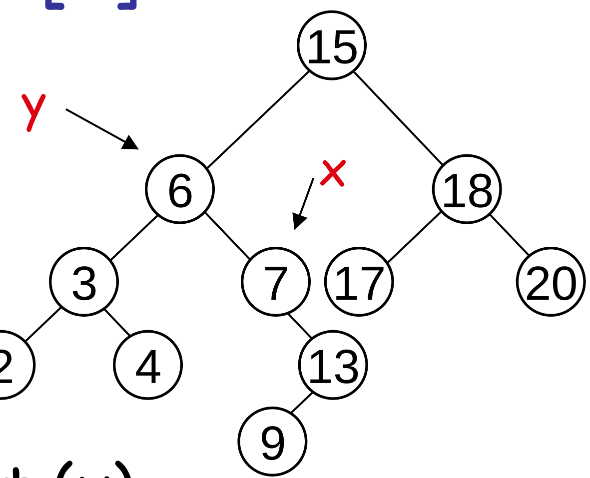


Running time: *O* (h), h – height of the tree

Predecessor

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

- *E.g.: predecessor (15) = 13*
  predecessor (9) = 7
  predecessor (7) = 6



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

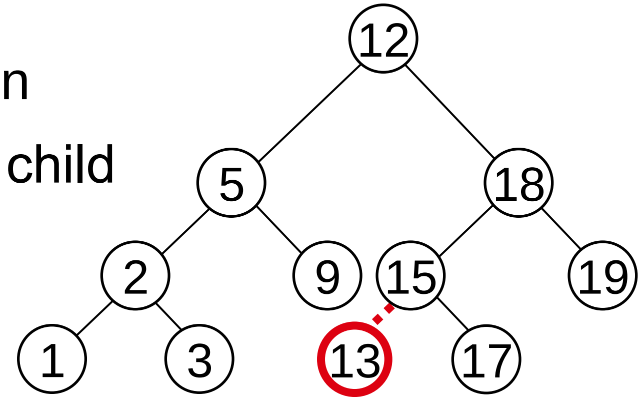
Insertion

- Goal:
  - Insert value v into a binary search tree
- Idea:
  - If 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 < key [y] insert the new node as y's left child

    else insert it as y's right child

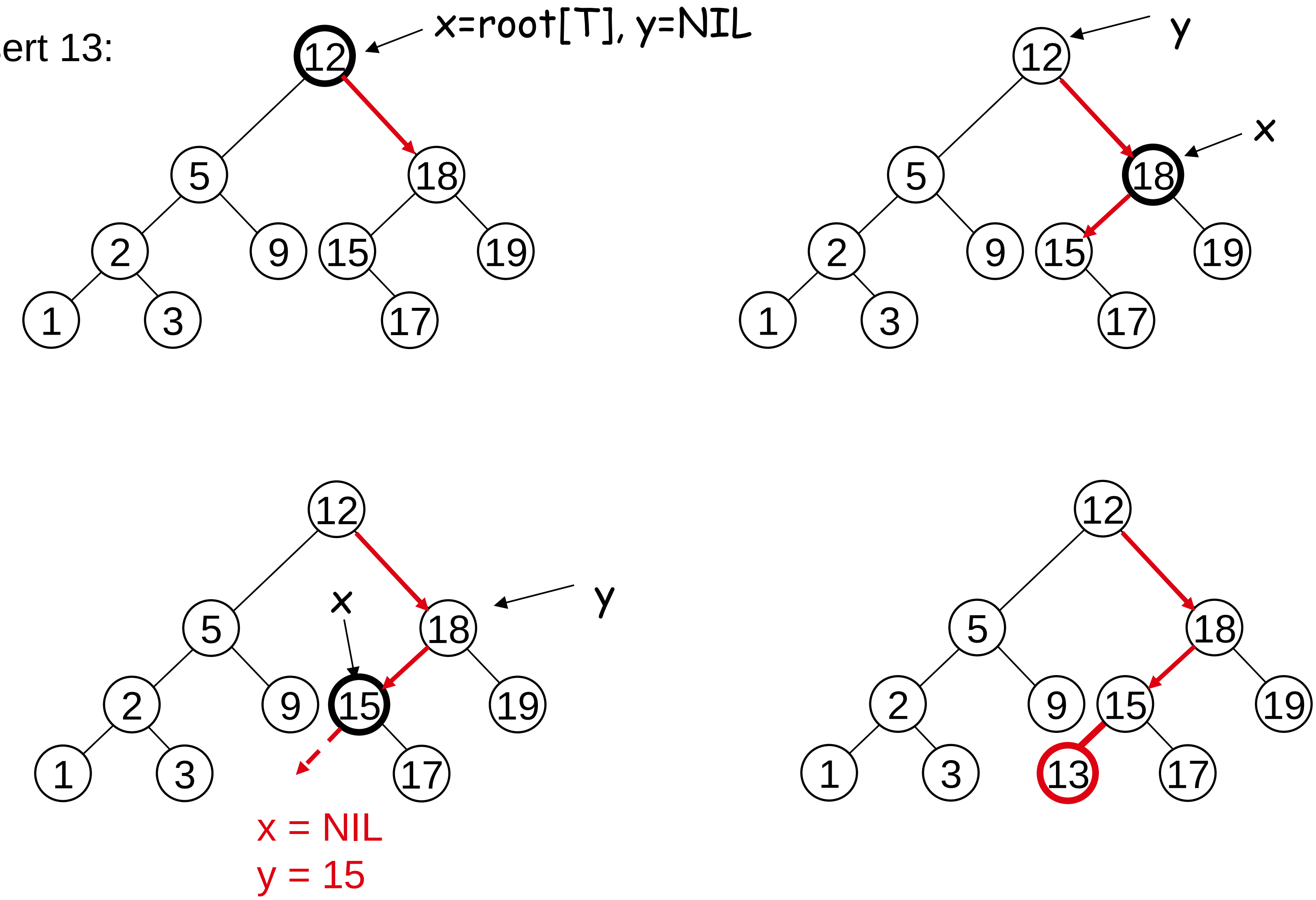
Insert value 13



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

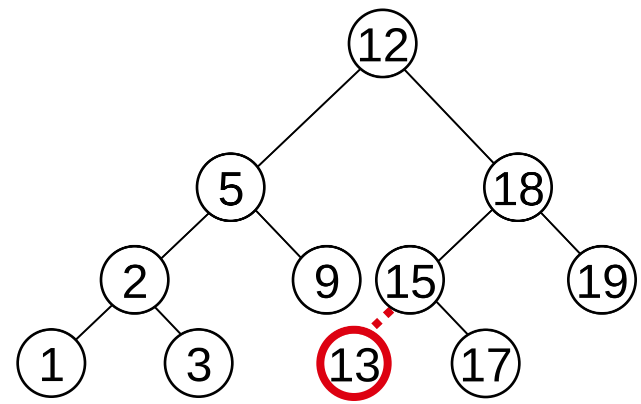
Example: TREE-INSERT

Insert 13:



Alg: TREE-INSERT(T, z)

```
y ← NIL
x ← root [T]
while x ≠ NIL
     do y ← x
          if key [z] < key [x]
             then x ← left [x]
             else x ← right [x]
p[z] ← y
if y = NIL
   then root [T] ← z
   else if key [z] < key [y]
           then left [y] ← z
           else right [y] ← z
```

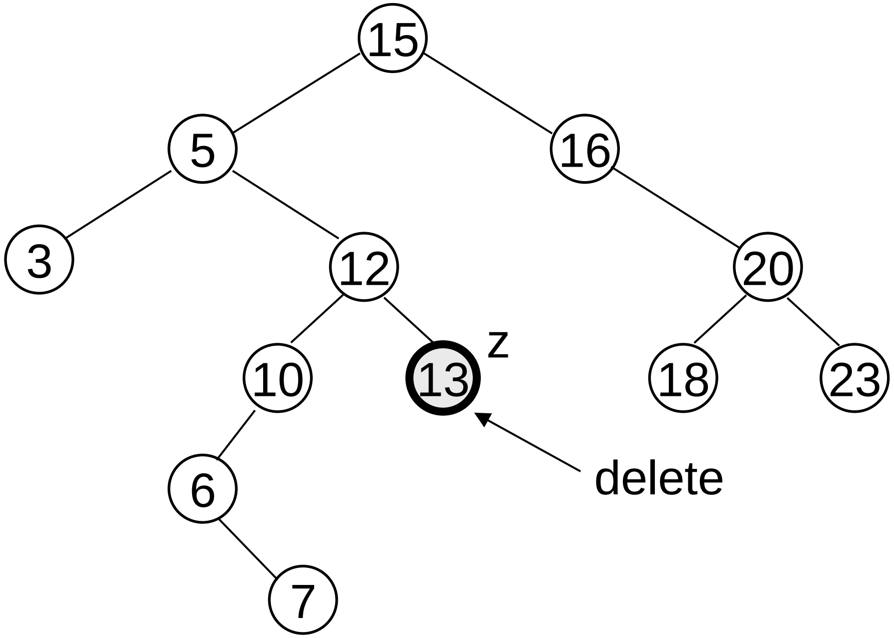


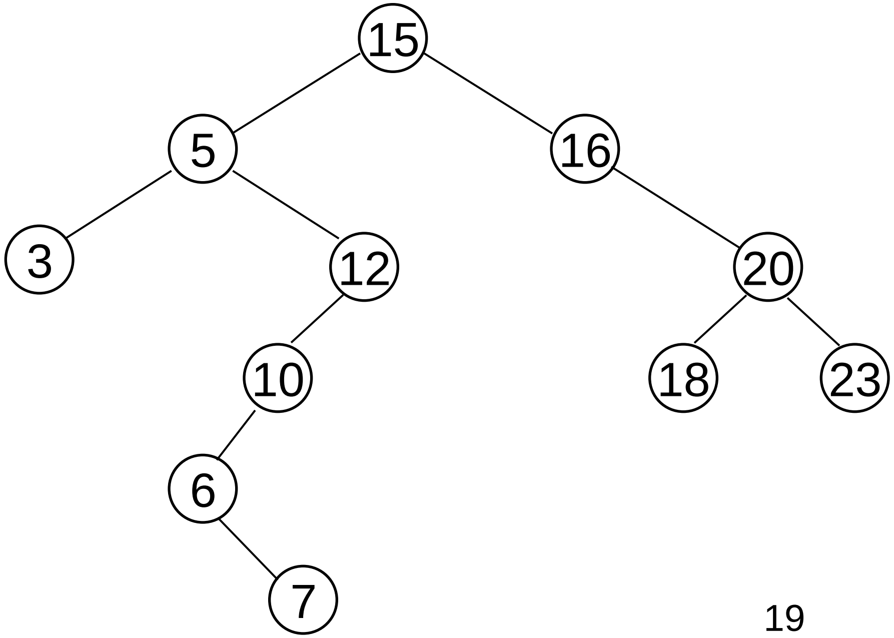
▷ Tree T was empty

Running time: $O(h)$

Deletion

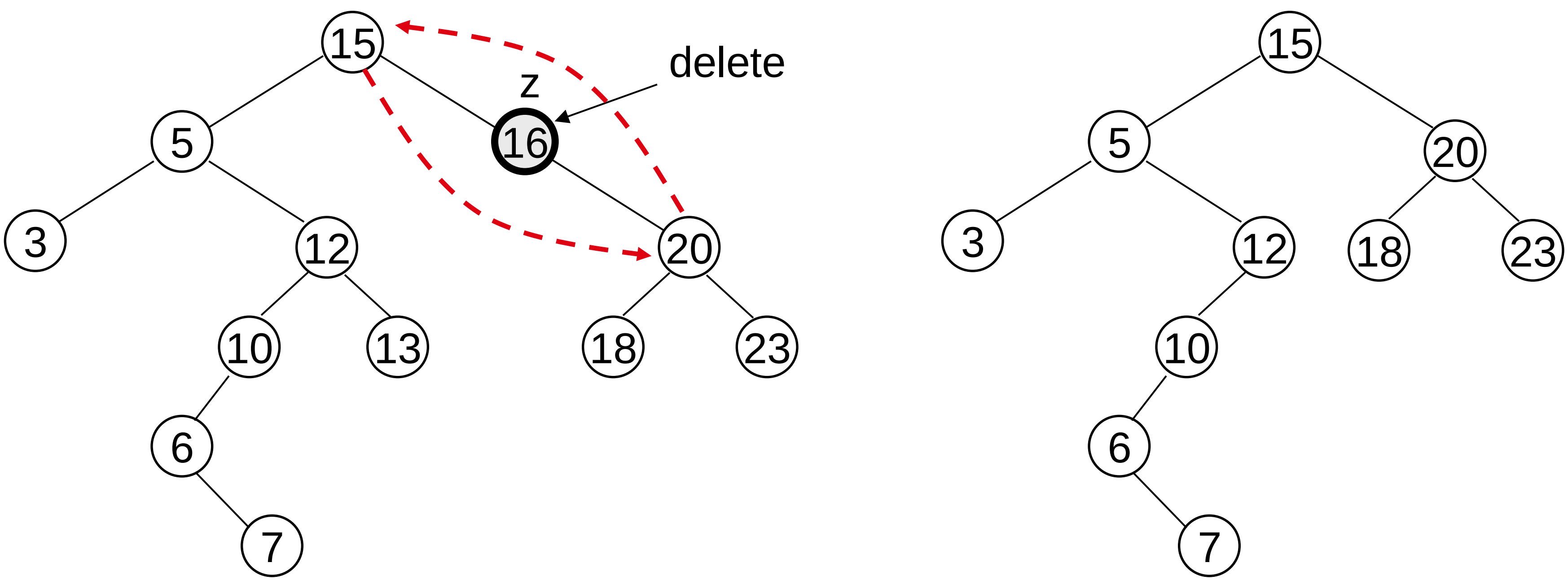
- Goal:
  – Delete a given node **z** from a binary search tree
- Idea:
  – **Case 1: z** has no children
    - Delete **z** by making the parent of **z** point to NIL





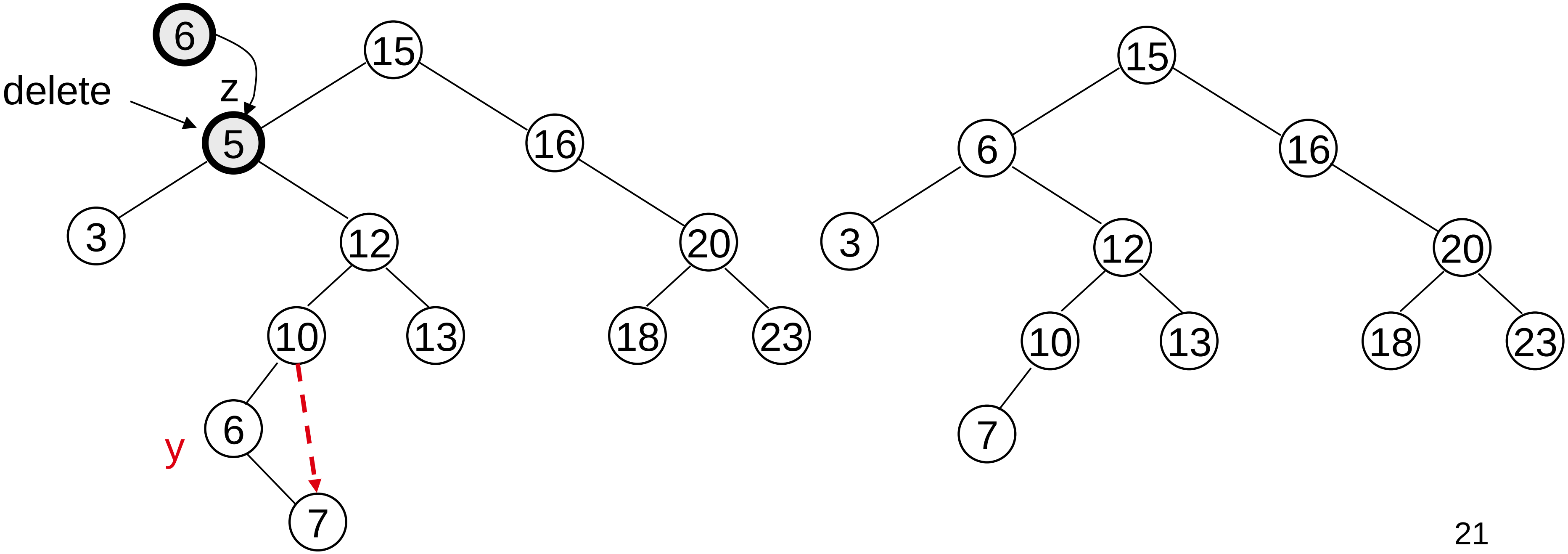
Deletion

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



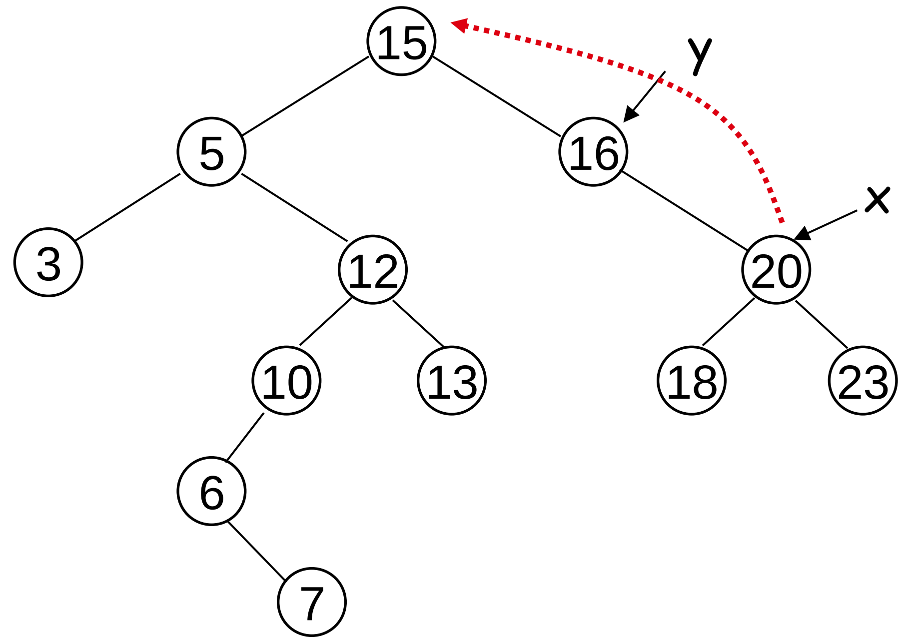
Deletion

- **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.



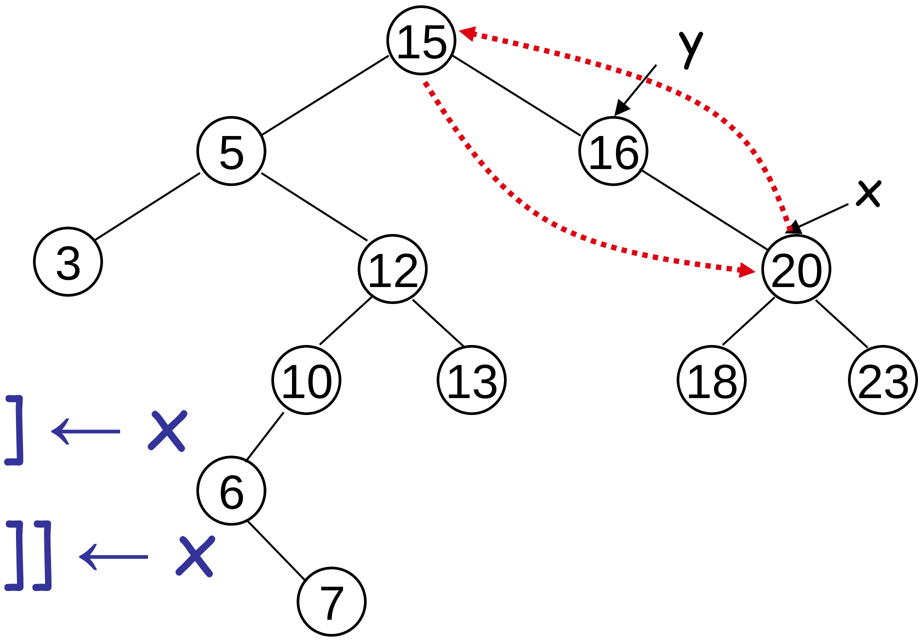
TREE-DELETE(T, z)

1. **if** left[z] = NIL or right[z] = NIL
2. **then** y ← z — z has one child
3. **else** y ← TREE-SUCCESSOR(z) — z has 2 children
4. **if** left[y] ≠ NIL
5. **then** x ← left[y]
6. **else** x ← right[y]
7. **if** x ≠ NIL
8. **then** p[x] ← p[y]



TREE-DELETE(T, z) – cont.

```
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
           copy y’s satellite data into z
return y
```



Running time: $O(h)$

Binary Search Trees - Summary

- Operations on binary search trees:
  - SEARCH $O(h)$
  - PREDECESSOR $O(h)$
  - SUCCESOR $O(h)$
  - 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

Problems

- **Exercise 12.1-2 (page 256)** What is the difference between the MAX-HEAP property and the binary search tree property?

- **Exercise 12.1-2 (page 256)** Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time?

- Can you use the heap property to design an efficient algorithm that searches for an item in a binary tree?

Problems

- Let x be the root node of a binary search tree (BST). Write an algorithm BSTHeight(x) that determines the height of the tree. What would be its running time?

```
Alg: BSTHeight(x)
    if (x==NULL)
        return -1;
    else
        return max (BSTHeight(left[x]),
                    BSTHeight(right[x]))+1;
```

Problems

- **(Exercise 12.3-5, page 264)** In a binary search tree, are the insert and delete operations commutative?
- Insert:
  - Try to insert 4 followed by 6, then insert 6 followed by 4
- Delete
  - Delete 5 followed by 6, then 6 followed by 5 in the following tree