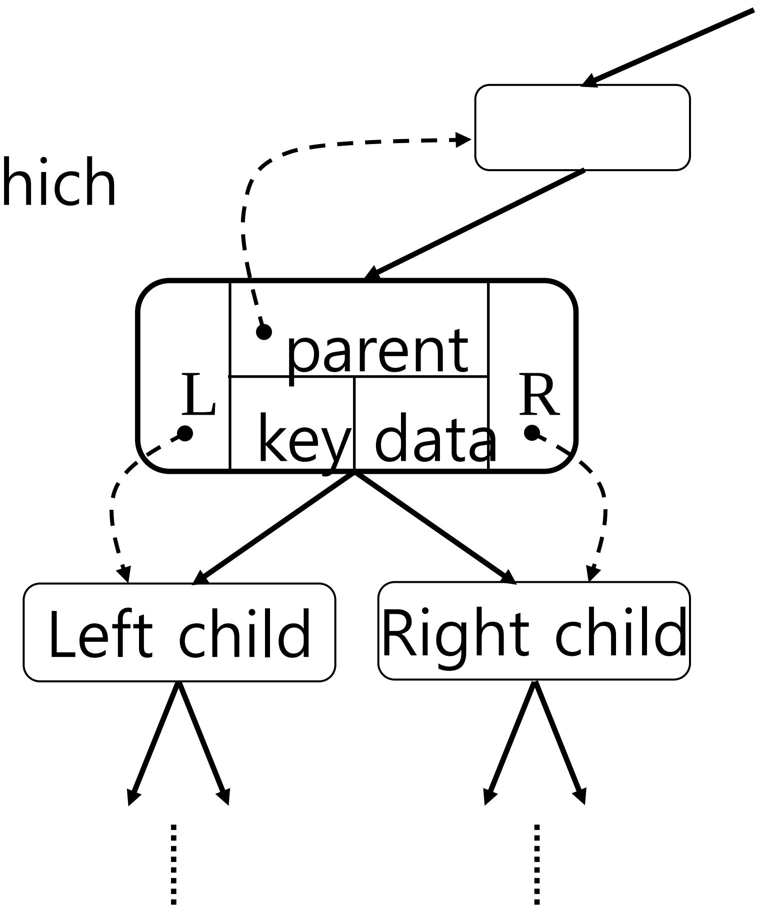
Chapter12. Binary Search Tree

- Binary Search Tree Representation
- Binary Search Tree Property
- Operations on binary search trees

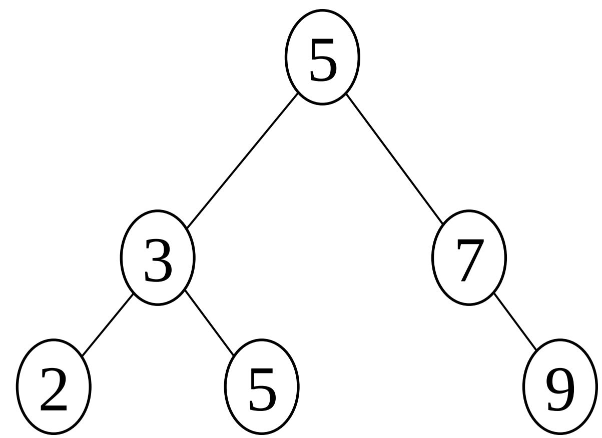
Binary Search Tree Representation

- 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)



Binary Search Tree Property

- Binary search tree property:
  - If y is in left subtree of x,
    then key [y] ≤ key [x]
  - If y is in right subtree of x,
    then key [y] ≥ key [x]

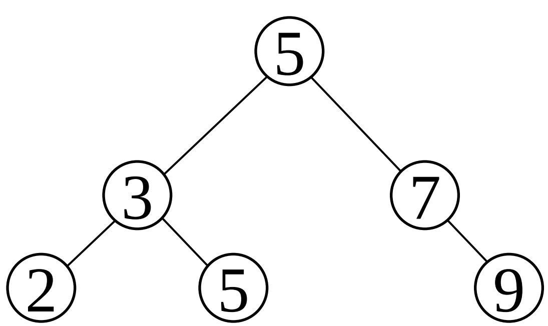


Binary Search Tree

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

Traversing a Binary Search Tree

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



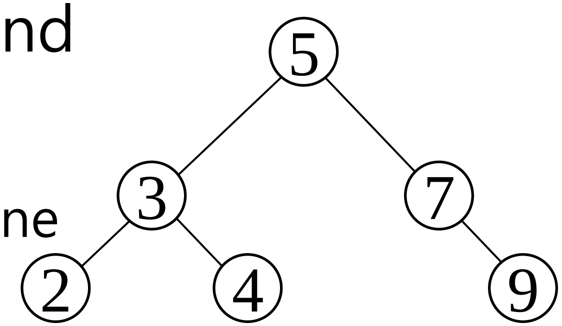
Inorder: 2 3 5 5 7 9

Preorder: 5 3 2 5 7 9

Postorder: 2 5 3 9 7 5

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

Searching for a Key

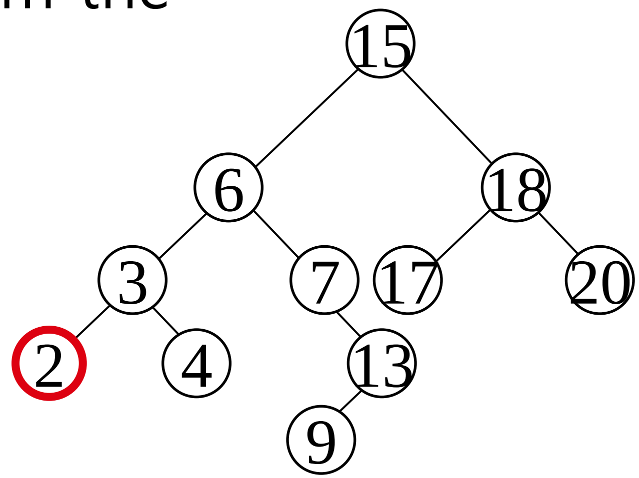
TREE-SEARCH(x, k)

if x = NIL or k = key [x]
    then return x
if k < key [x]
    then return TREE-SEARCH(left [x], k)
    else return TREE-SEARCH(right [x], k)

- Running Time: O (h) , where h is the height of the tree

Finding the Minimum

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



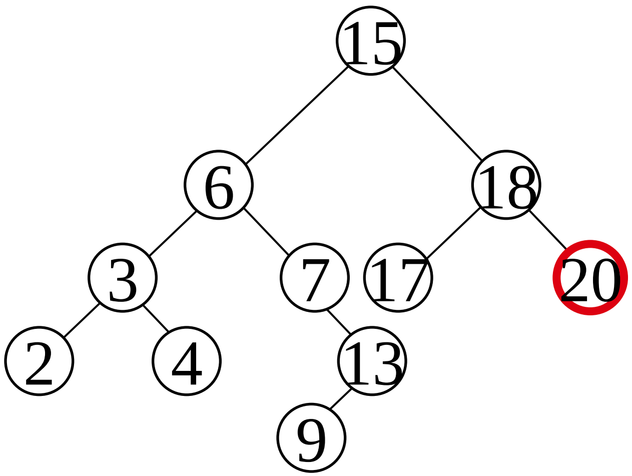
TREE-MINIMUM(x)

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

Running time: O(h), where h is the height of tree

Find the Maximum

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



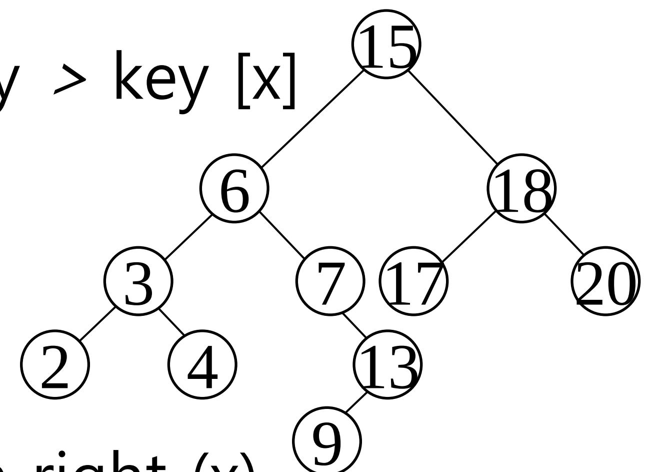
TREE-MAXIMUM*(x)*

1. **while** right [x] ≠ NIL
2. **do** x ← right [x]
3. **return** x

- Running time: O(h), where h is the height of tree

Find the Successor

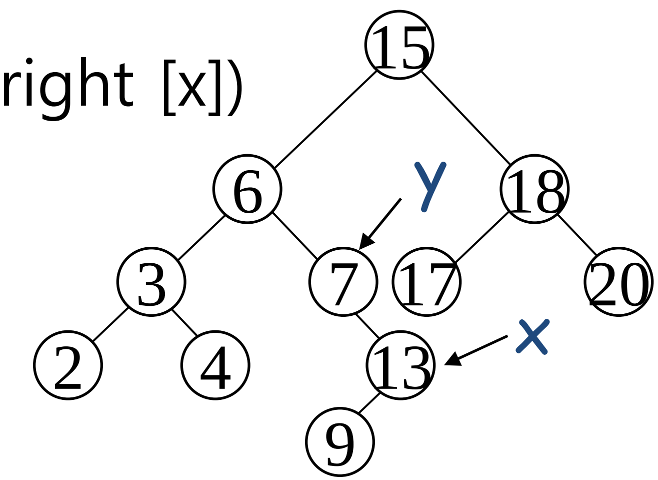
Def: successor (x) = y,
such that key [y] is the smallest key > key [x]



- 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

Find the Successor

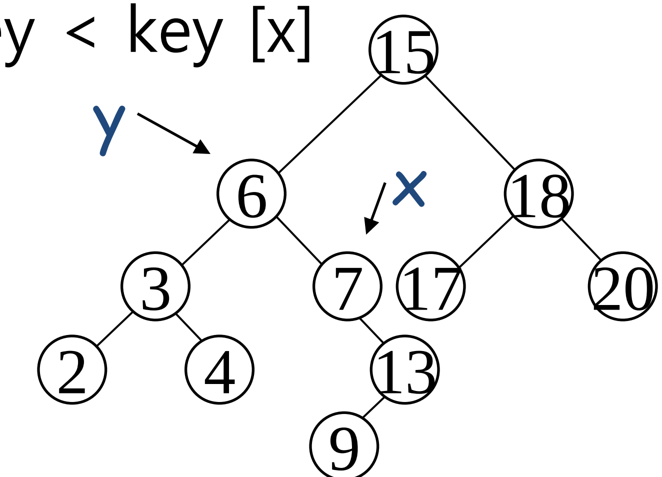
```
TREE-SUCCESSOR(x)
    if right [x] ≠ NIL
        then return TREE-MINIMUM(right [x])
    y ← p[x]
    while y ≠ NIL and x = right [y]
        do x ← y
            y ← p[y]
    return y
```



Running time: O (h), where h is the height of the tree

Find the Predecessor

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



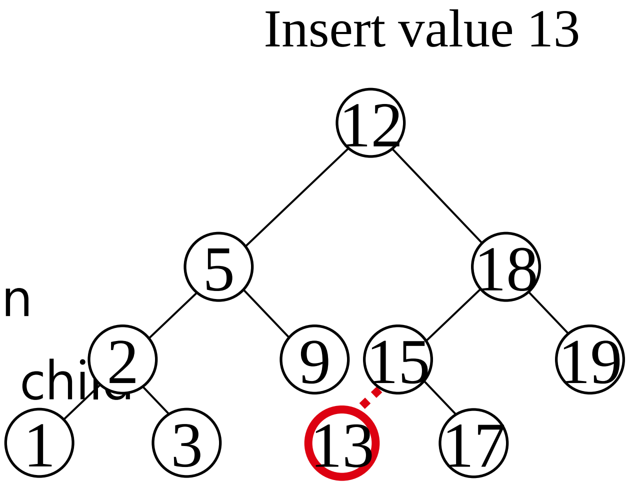
- E.g.: predecessor (15) = 13
  predecessor (9) = 7
  predecessor (7) = 6
- Case 1: left (x) is non empty
  - predecessor (x) = the maximum in left (x)
- Case 2: left (x) is empty
  - go up the tree until the current node is a right child: predecessor (x) is the parent of the current node
  - if you 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,

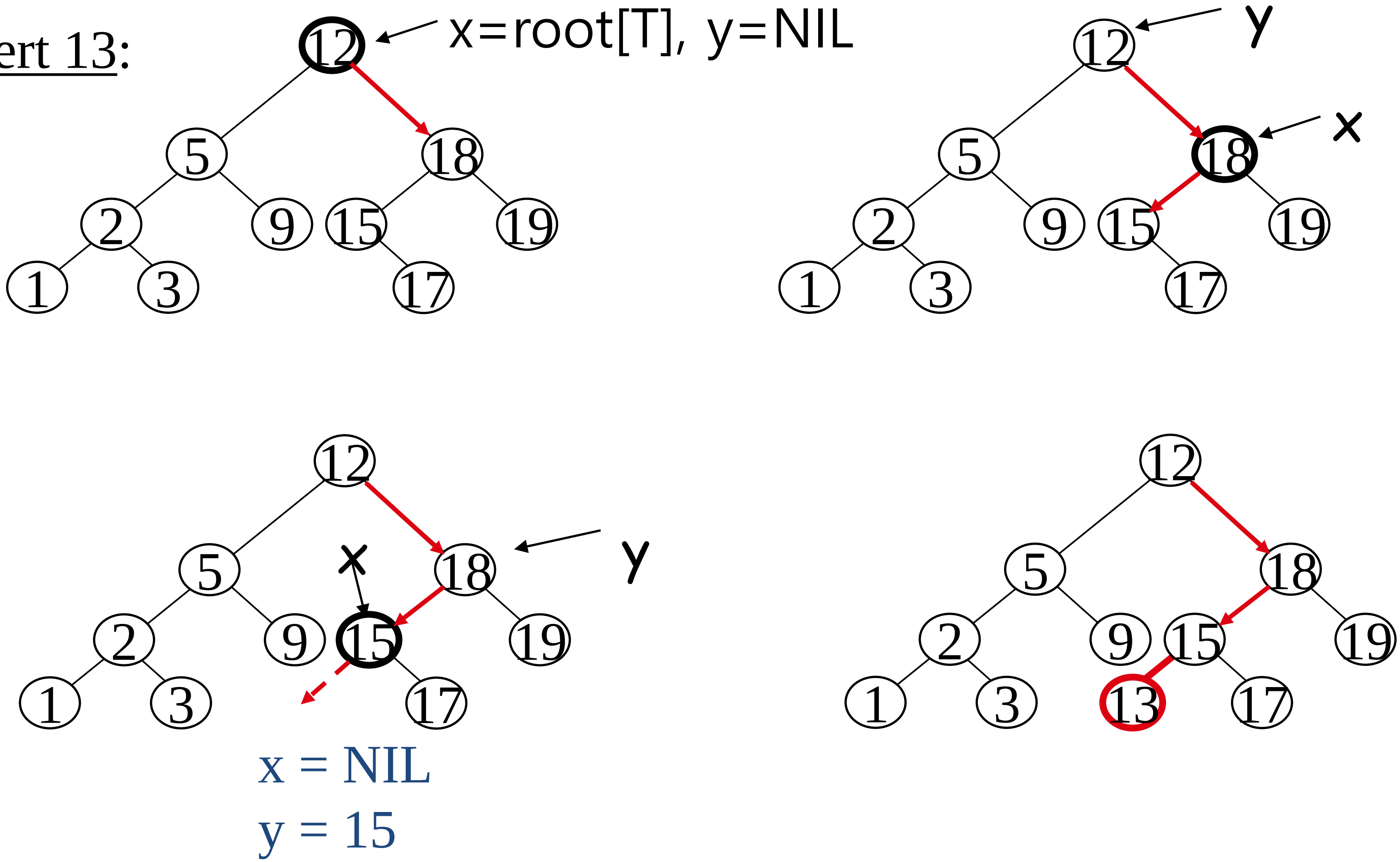
    else move to the left child of x
  – When x is NIL, we found the correct position
  – If v < key [y] insert the new node as y's left child

    else insert it as y's right child
  – 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")



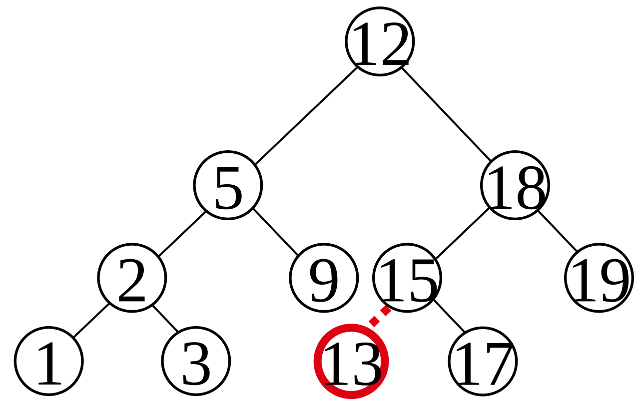
Example: TREE-INSERT

Insert 13:



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
```

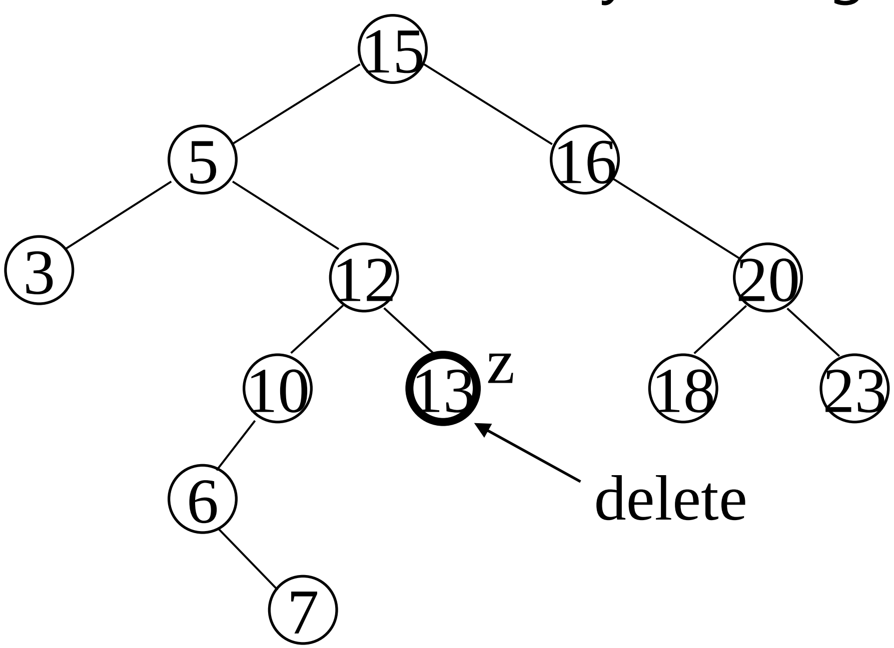


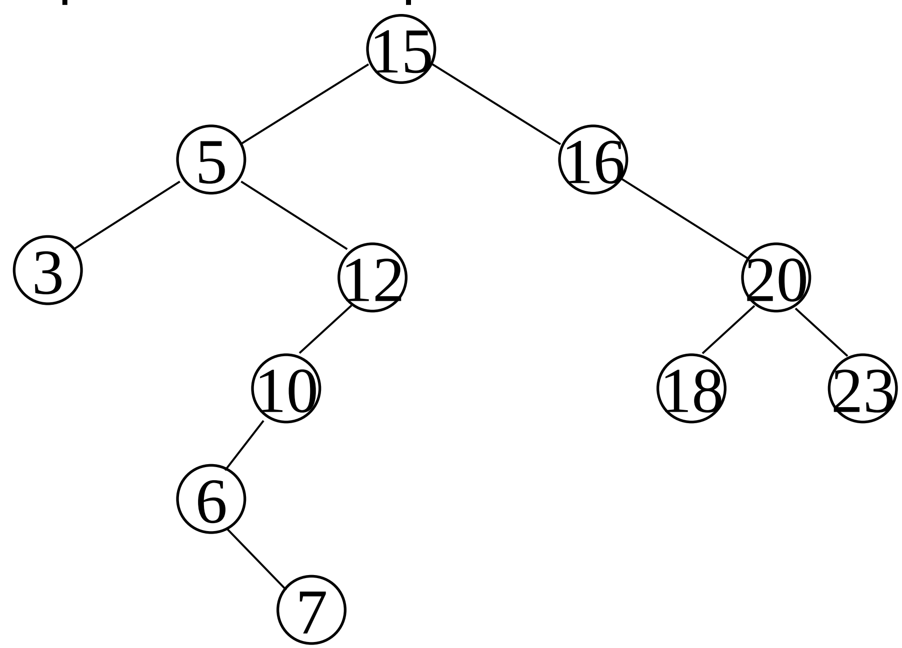
Tree T was empty

Running time: O(h)

Deletion

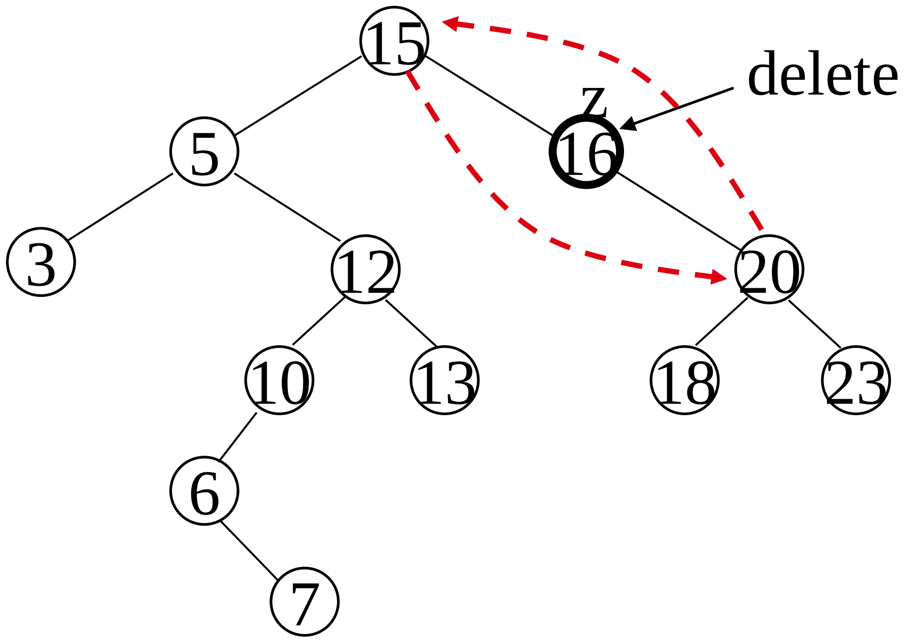
- Goal:
  - Delete a given node **z** from a binary search tree
- Idea:
  - **<u>Case 1</u>:** z has no children
    - Delete z by making the parent of z point to NIL

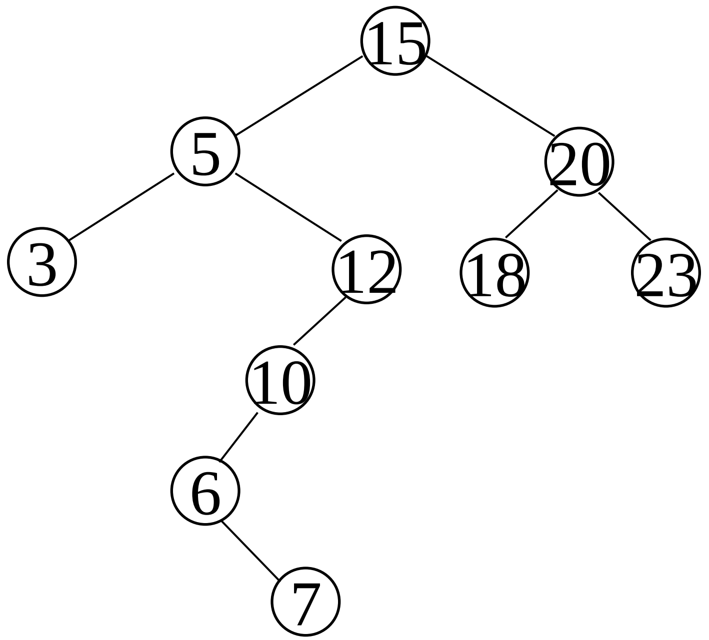




Deletion

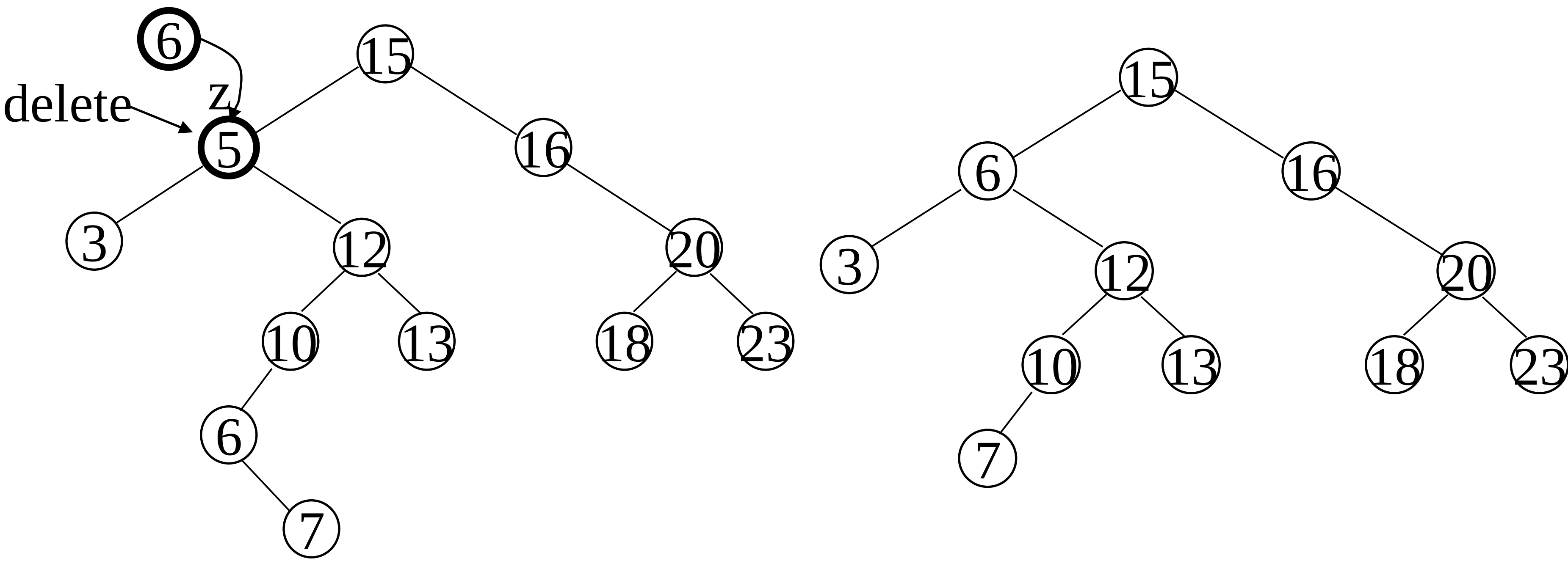
- **<u>Case 2</u>:** z has one child
  - Delete z by making the parent of z point to z's child, instead of to z





Deletion

- **<u>Case 3</u>:** 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)

if left[z] = NIL or right[z] = NIL

    then y ← z   z has one child

    else y ← TREE-SUCCESSOR(z)   z has 2 children

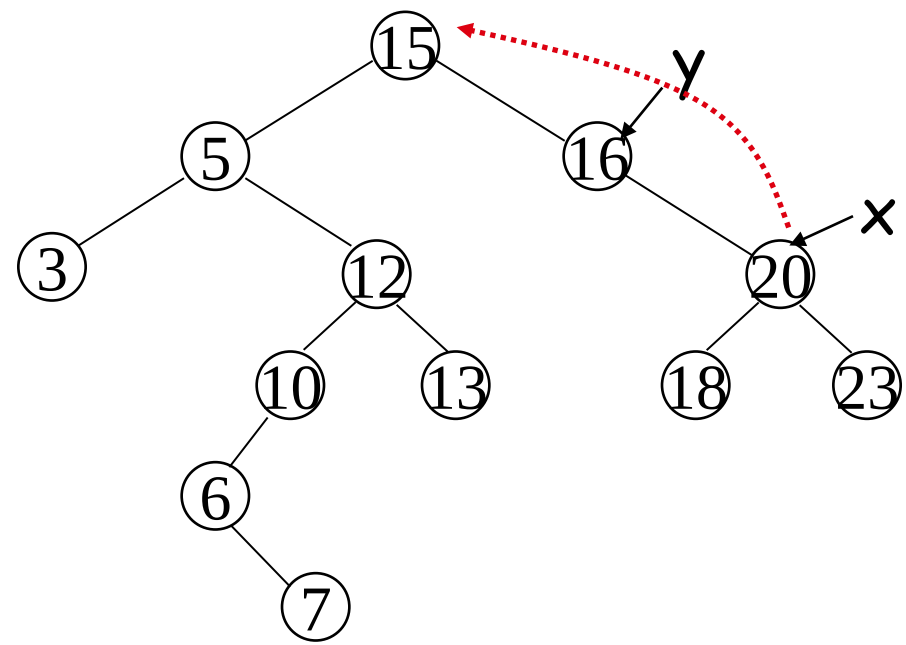
if left[y] ≠ NIL

    then x ← left[y]

    else x ← right[y]

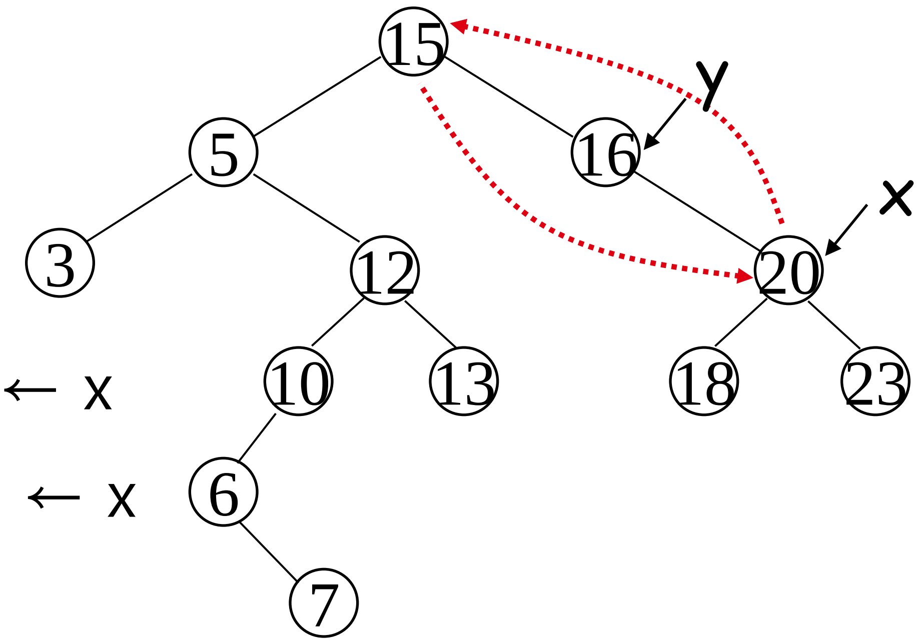
if x ≠ NIL

    then p[x] ← p[y]



TREE-DELETE(T, z)

```
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