Binary Search Trees

Binary Search Trees

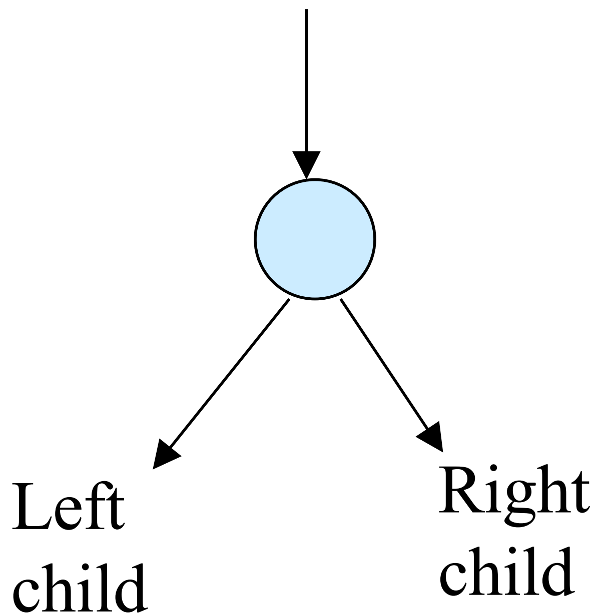
- View today as data structures that can support dynamic set operations.
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - Dictionaries.
  - Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

Definition Of Binary Search Tree

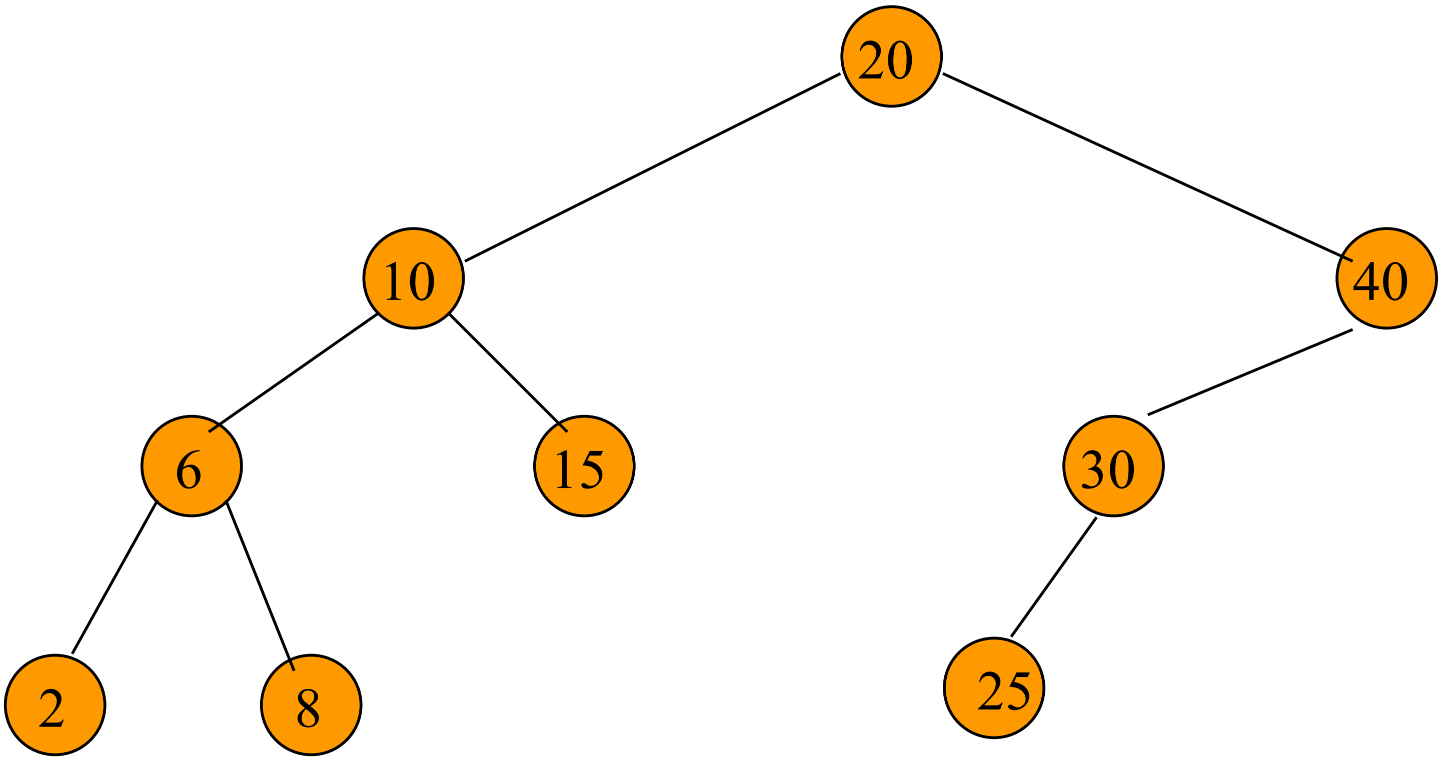
- A binary tree.
- Each node has a (key, value) pair.
- For every node x, all keys in the left subtree of x are smaller than (≤) that in x.
- For every node x, all keys in the right subtree of x are greater than (≥) that in x.

BST – Representation

- Represented by a linked data structure of nodes.
- *root*(T) points to the root of tree T.
- Each node contains fields:
  - *key*
  - *left* – pointer to left child
  - *right* – pointer to right child

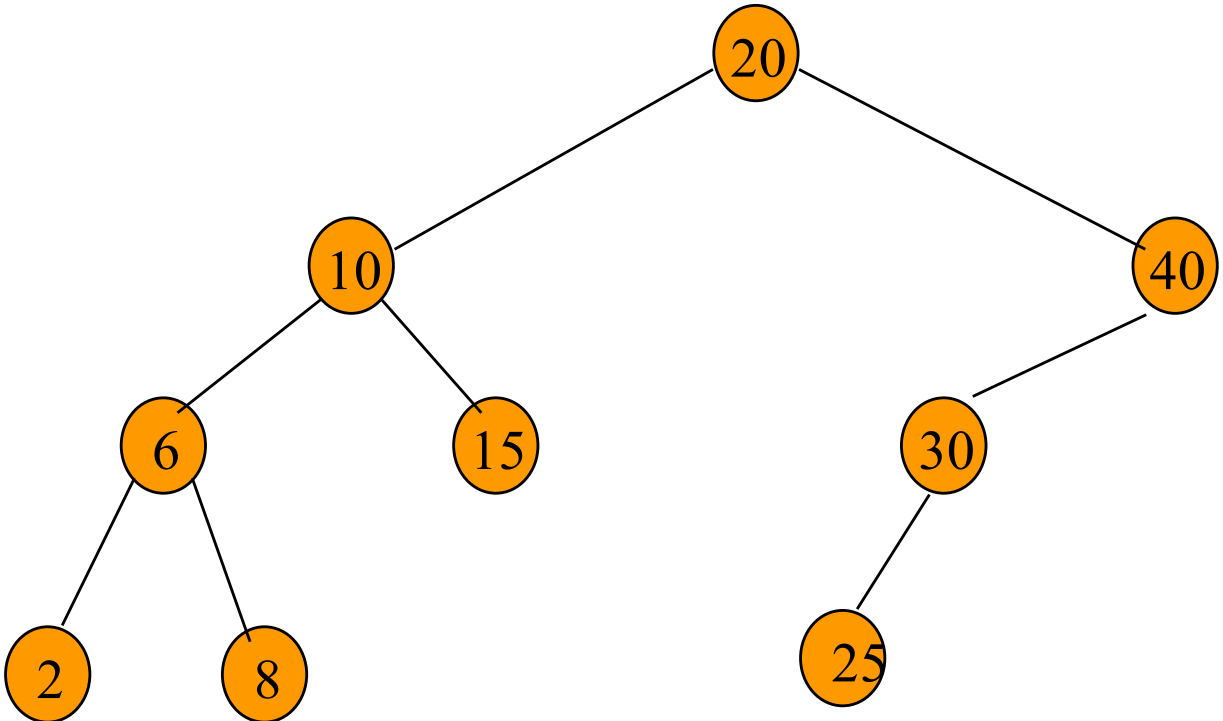


Example Binary Search Tree



Only keys are shown.

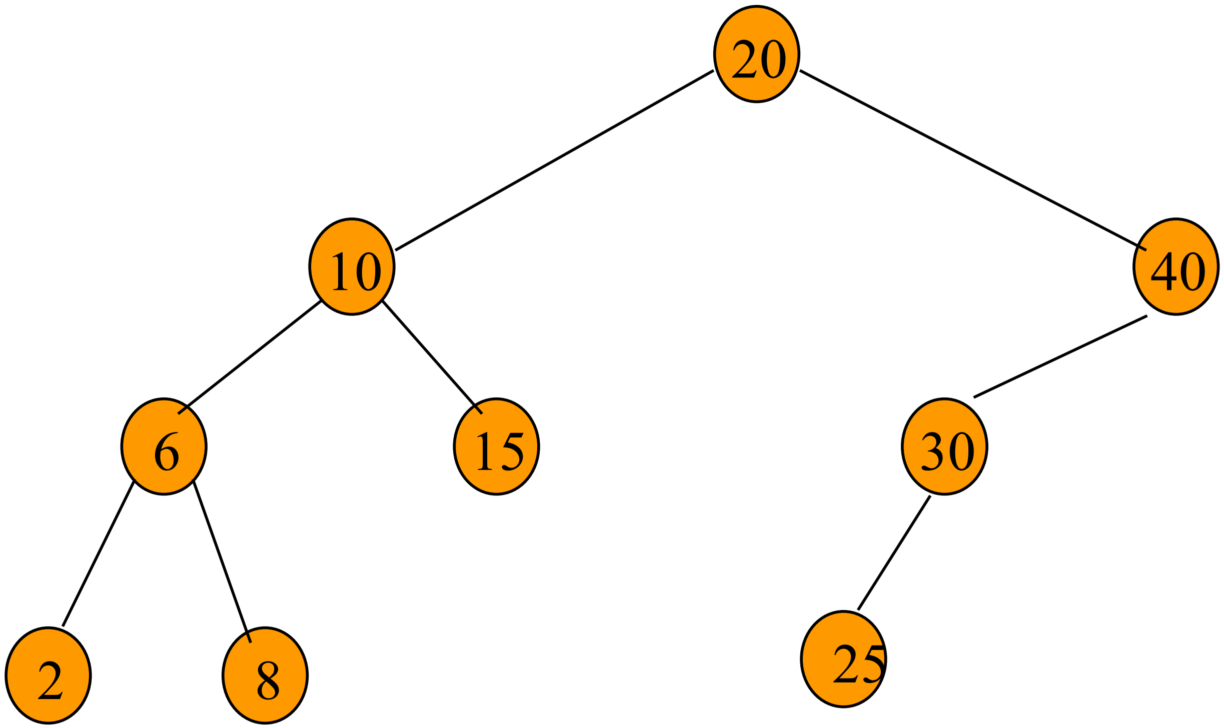
Inorder Traversal



Inorder-Tree-Walk (x)

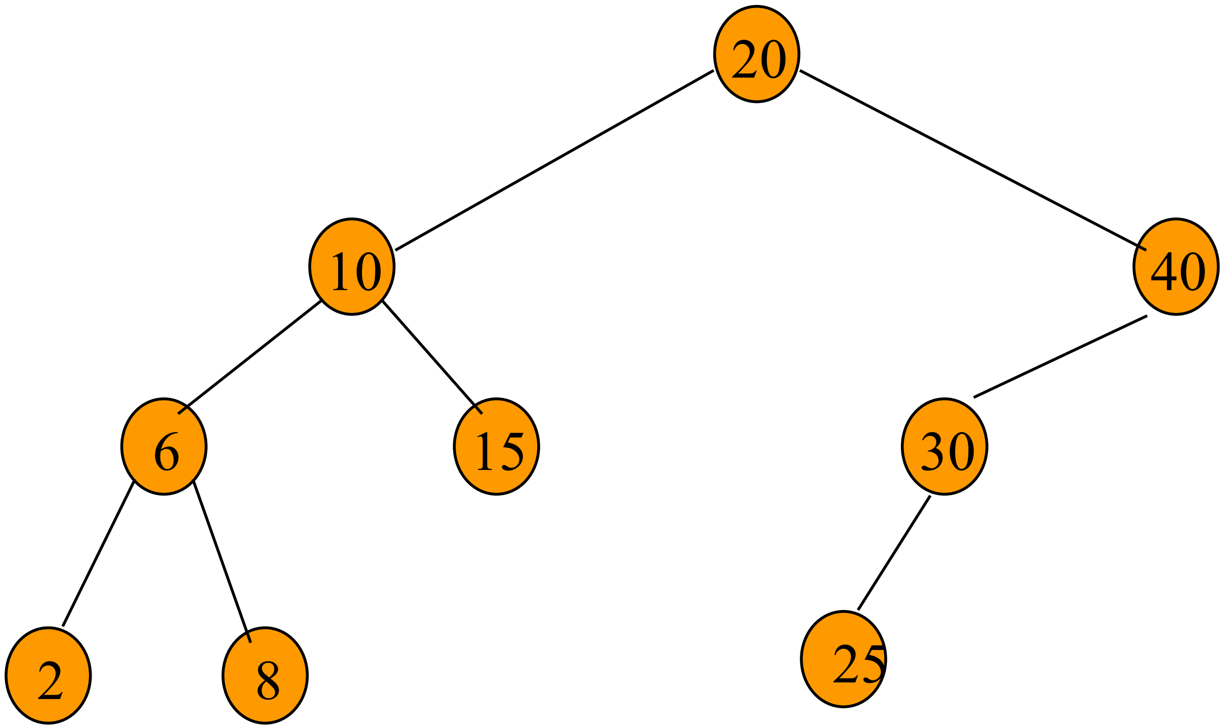
1. **if** $x \neq$ NIL
2. **then** Inorder-Tree-Walk(*left*[x])
3. print *key*[x]
4. Inorder-Tree-Walk(*right*[x])

Inorder Traversal



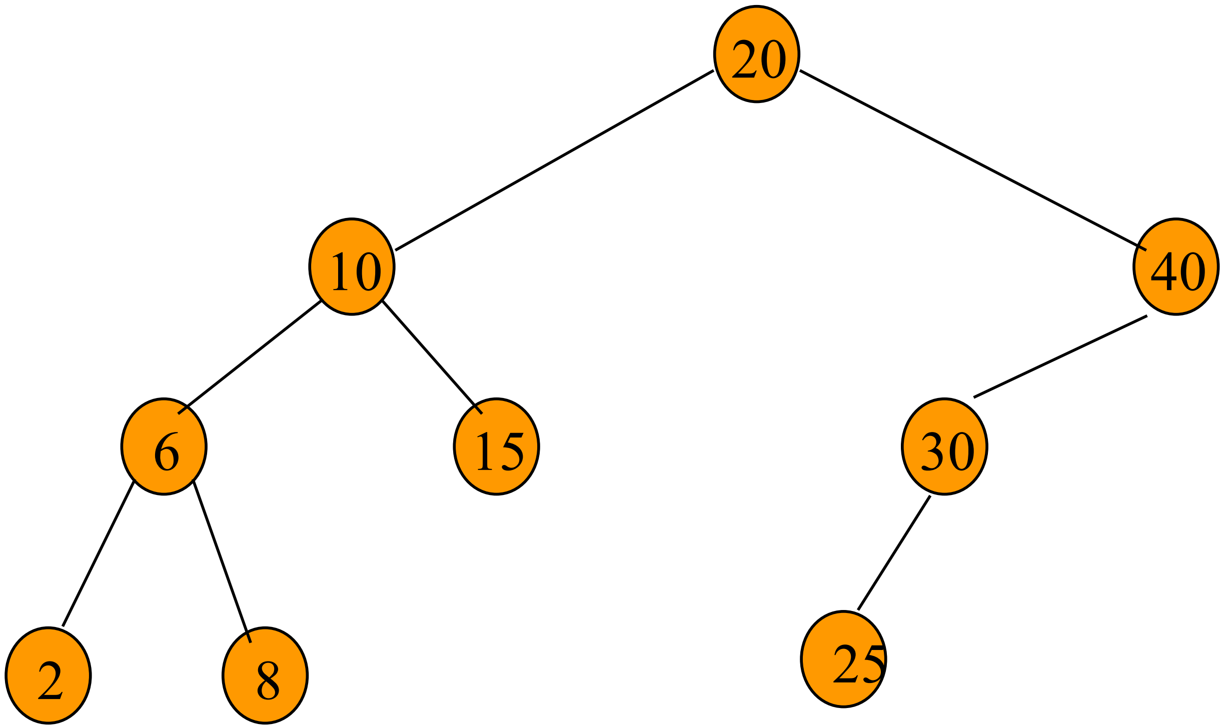
- What will be the output of inorder traversal?
- How can you characterize the output?
- How long does the traversal take?

Inorder Traversal



- What will be the output of inorder traversal?
- How can you characterize the output?
- How long does the traversal take? O(n)
- Does it imply that sorting can be done in O(n) time?

Inorder Traversal



- What will be the output of inorder traversal?
- How can you characterize the output?
- How long does the traversal take? O(n)
- Does it imply that sorting can be done in O(n) time?
  - No. Why?

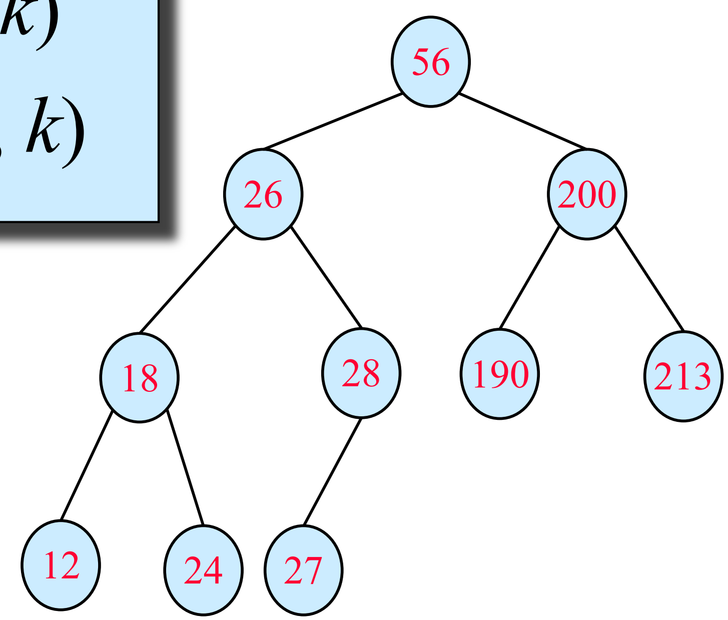
Querying a Binary Search Tree

- All dynamic-set search operations can be supported in $O(h)$ time.
- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

Question: When can such a BST be constructed?

Tree Search

```
Tree-Search(x, k)// search for key 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)
```

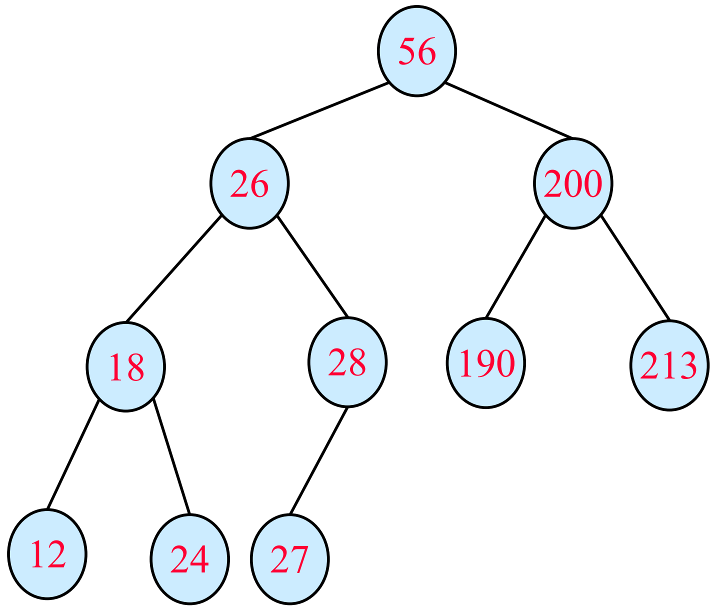


Running time: *O*(*h*)

Iterative Tree Search

Iterative-Tree-Search(x, k)

1. **while** $x \neq NIL$ **and** $k \neq key[x]$
2. **do if** $k < key[x]$
3. **then** $x \leftarrow left[x]$
4. **else** $x \leftarrow right[x]$
5. **return** x



The iterative tree search is more efficient on most computers.
The recursive tree search is more straightforward.

Finding Min & Max

- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.
  - » The maximum is located at the right-most node.

<u>Tree-Minimum(x)</u>

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

<u>Tree-Maximum(x)</u>

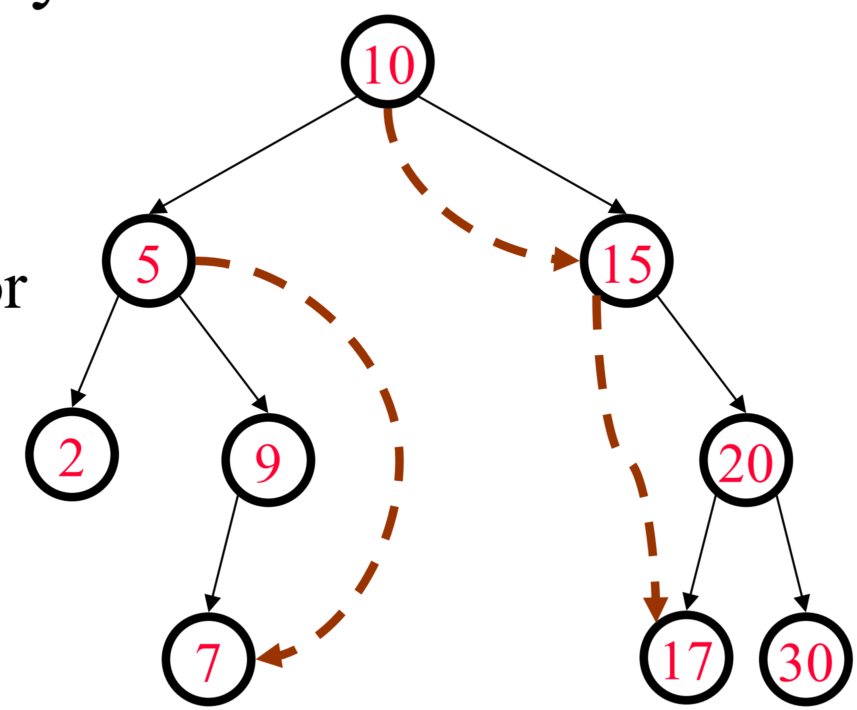
1. **while** $right[x] \neq NIL$
2. **do** $x \leftarrow right[x]$
3. **return** x

Q: How long do they take?

Predecessor and Successor

- Successor of node x is the node y such that $key[y]$ is the smallest key greater than $key[x]$.
- The successor of the largest key is NIL.
- Search consists of two cases.

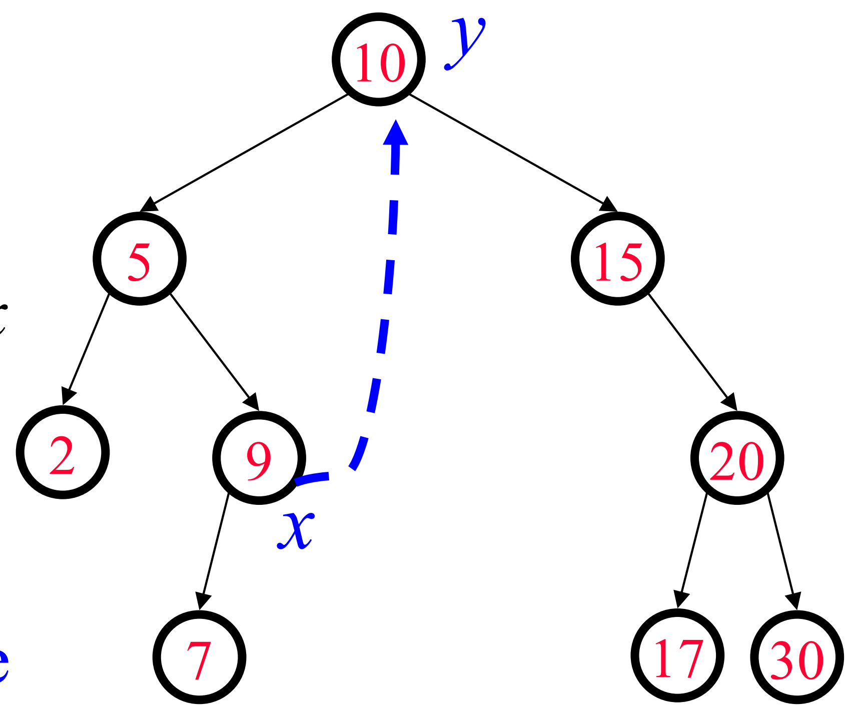
  Case 1: If x has a non-empty right subtree, then x' s successor is the minimum in the right subtree of x.



Predecessor and Successor

Case 2: If node x has an empty right subtree, then:

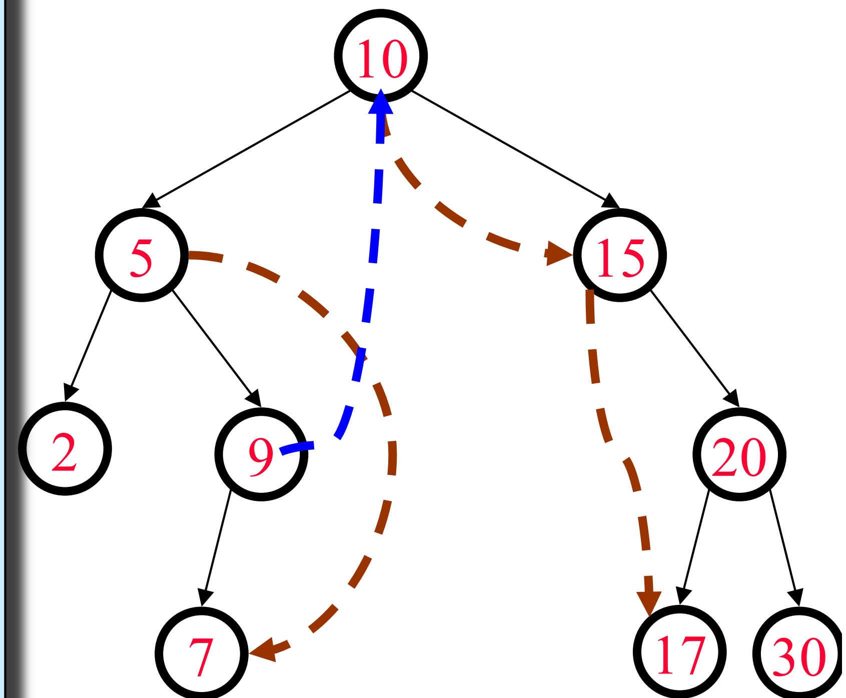
- As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
- x' s successor y is the node that x is the predecessor of (x is the maximum in y' s left subtree).
- In other words, x' s successor y, is the lowest ancestor of x whose left child is also an ancestor of x.



Pseudo-code for Successor

<u>Tree-Successor(x)</u>

- **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: $O(h)$

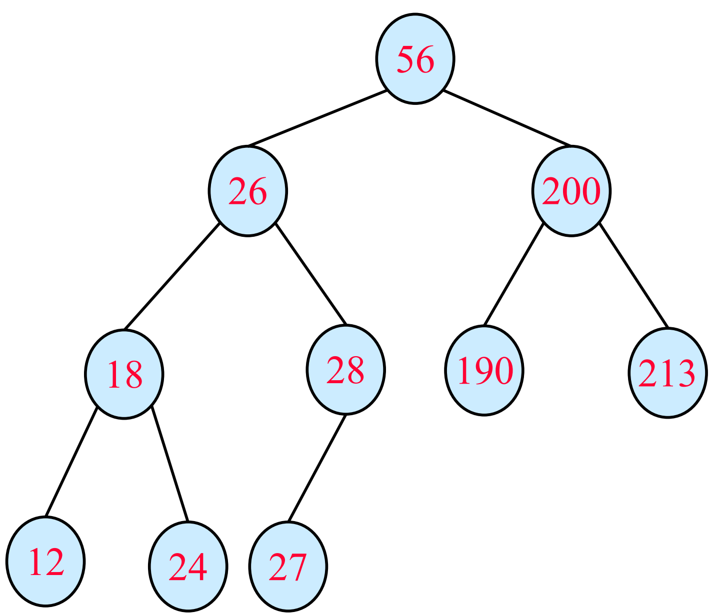
Practice problem: Write pseudo code for finding predecessor of a node.

BST Insertion

- Ensure BST property after insertion.
- Insertion is easier than deletion.
- Like search: search for the key to be inserted and attach it to the appropriate parent.

BST Insertion

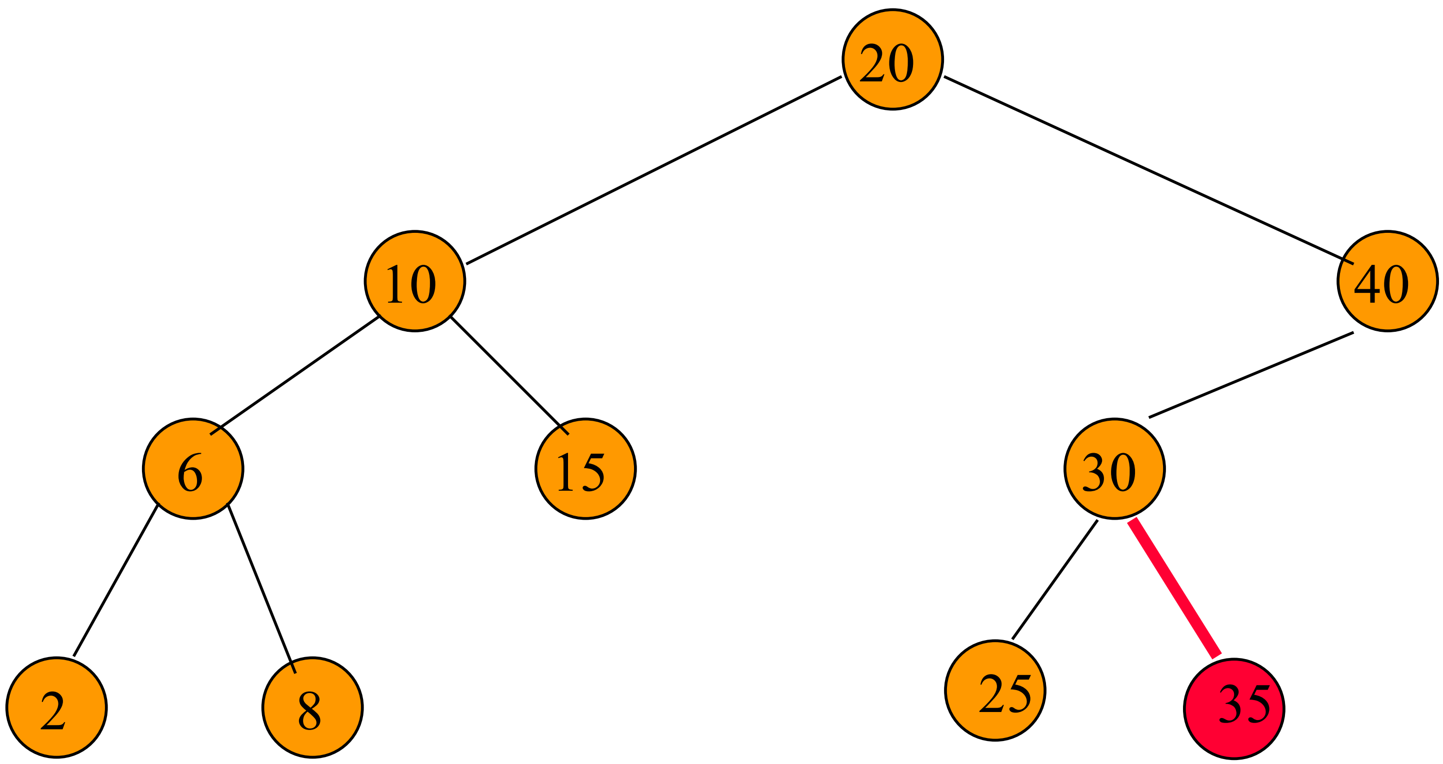
- Ensure BST property after insertion.
- Insertion is easier than deletion.
- Like search: search for the key to be inserted and attach it to the appropriate parent.



Tree-Insert(*T*, *z*)

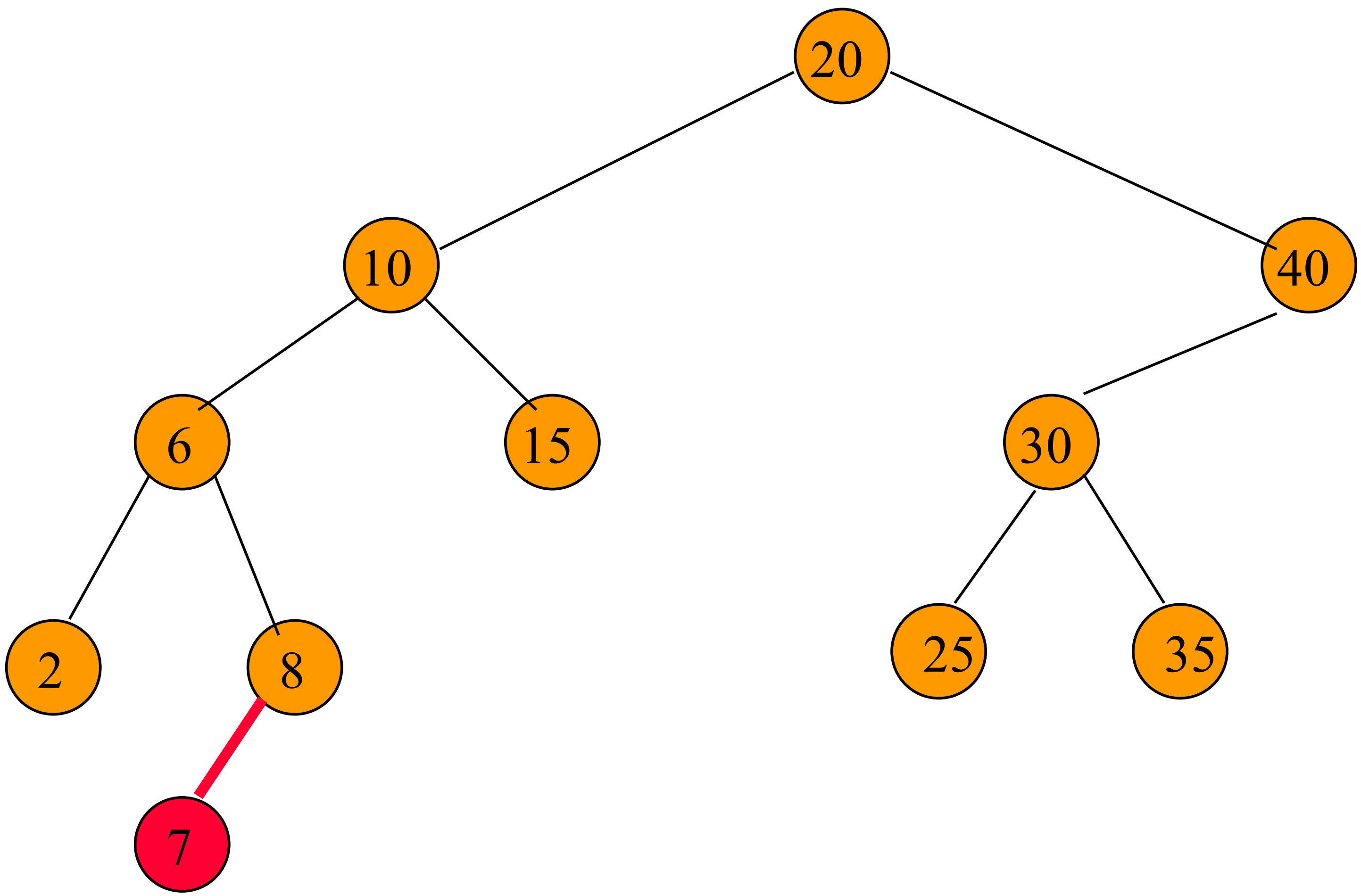
```
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
        else x ← right[x]
p[z] ← y
if y = NIL
    then root[t] ← z
    else if key[z] < key[y]
            then  left[y] ← z
        else right[y] ← z
```

The Operation Insert()



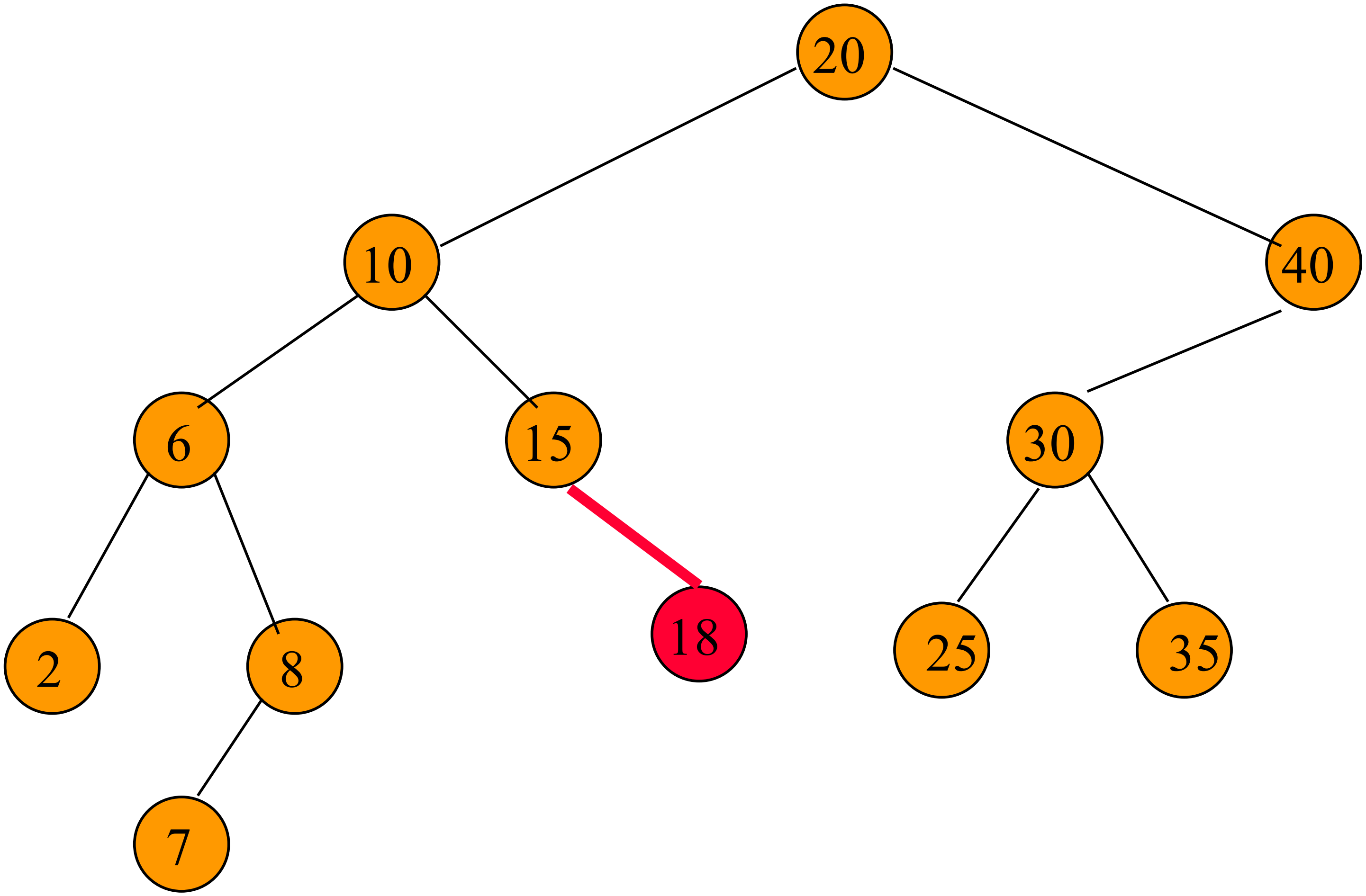
Insert a pair whose key is 35.

The Operation Insert ()



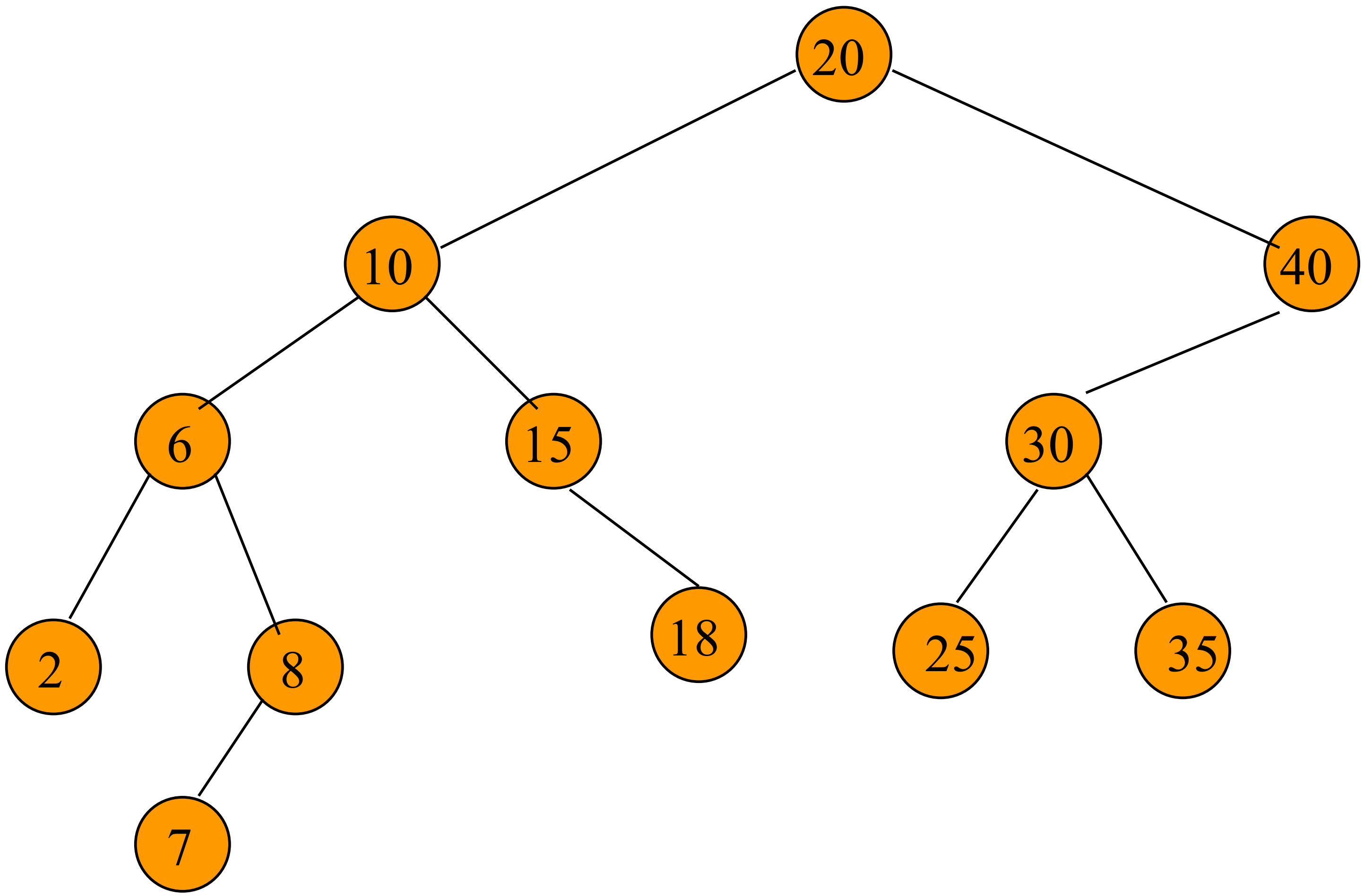
Insert a pair whose key is 7.

The Operation Insert()



Insert a pair whose key is 18.

The Operation Insert()



Complexity of Insert() is O(height).

Remove (T, x)

if x has no children (leaf) ♦ case 0

then remove x

if x has one child (degree 1) ♦ case 1

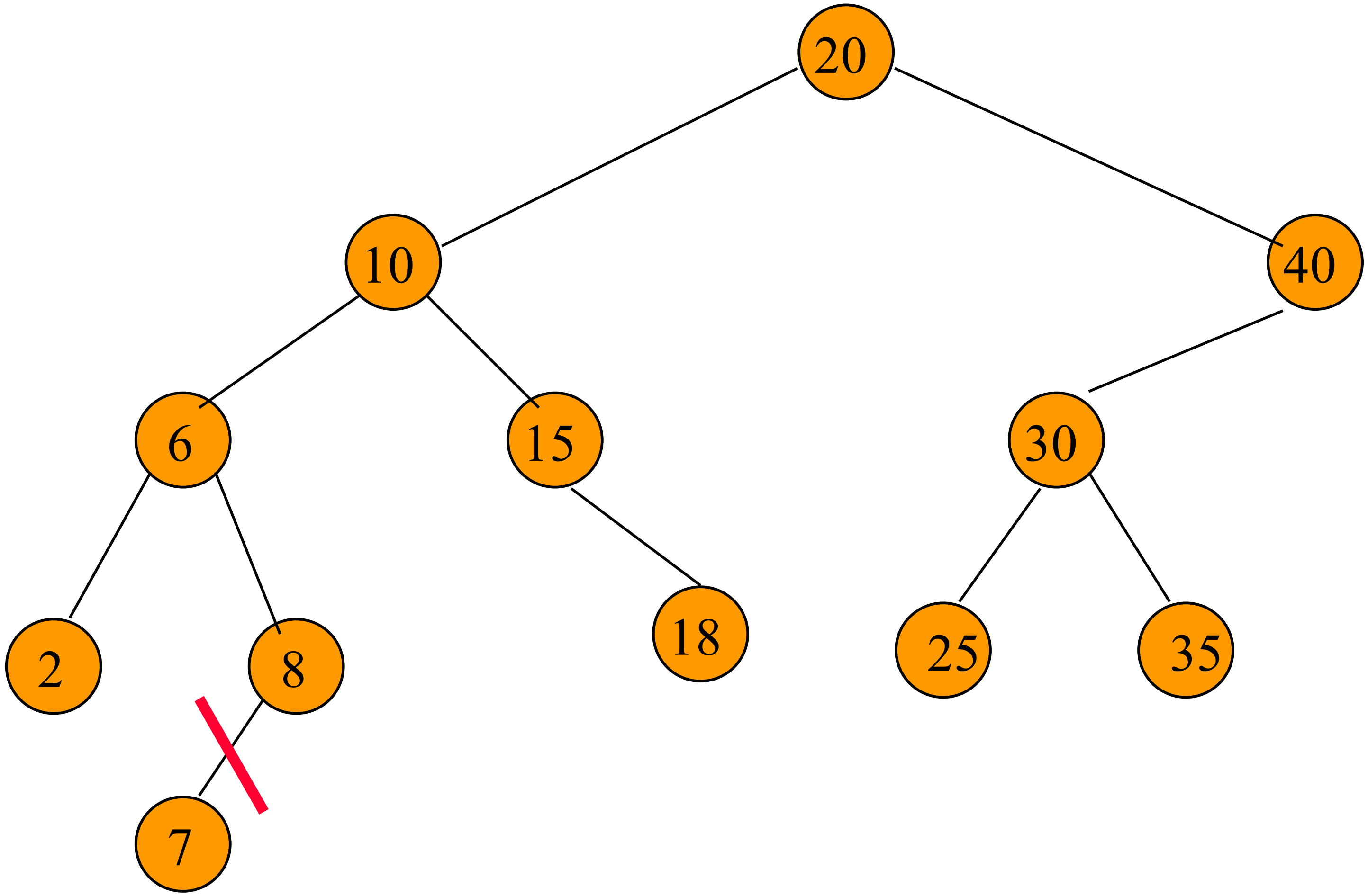
then make $p[x]$ point to child

if x has two children (degree 2) ♦ case 2

then swap x with max(x[left]) or min(x[right])

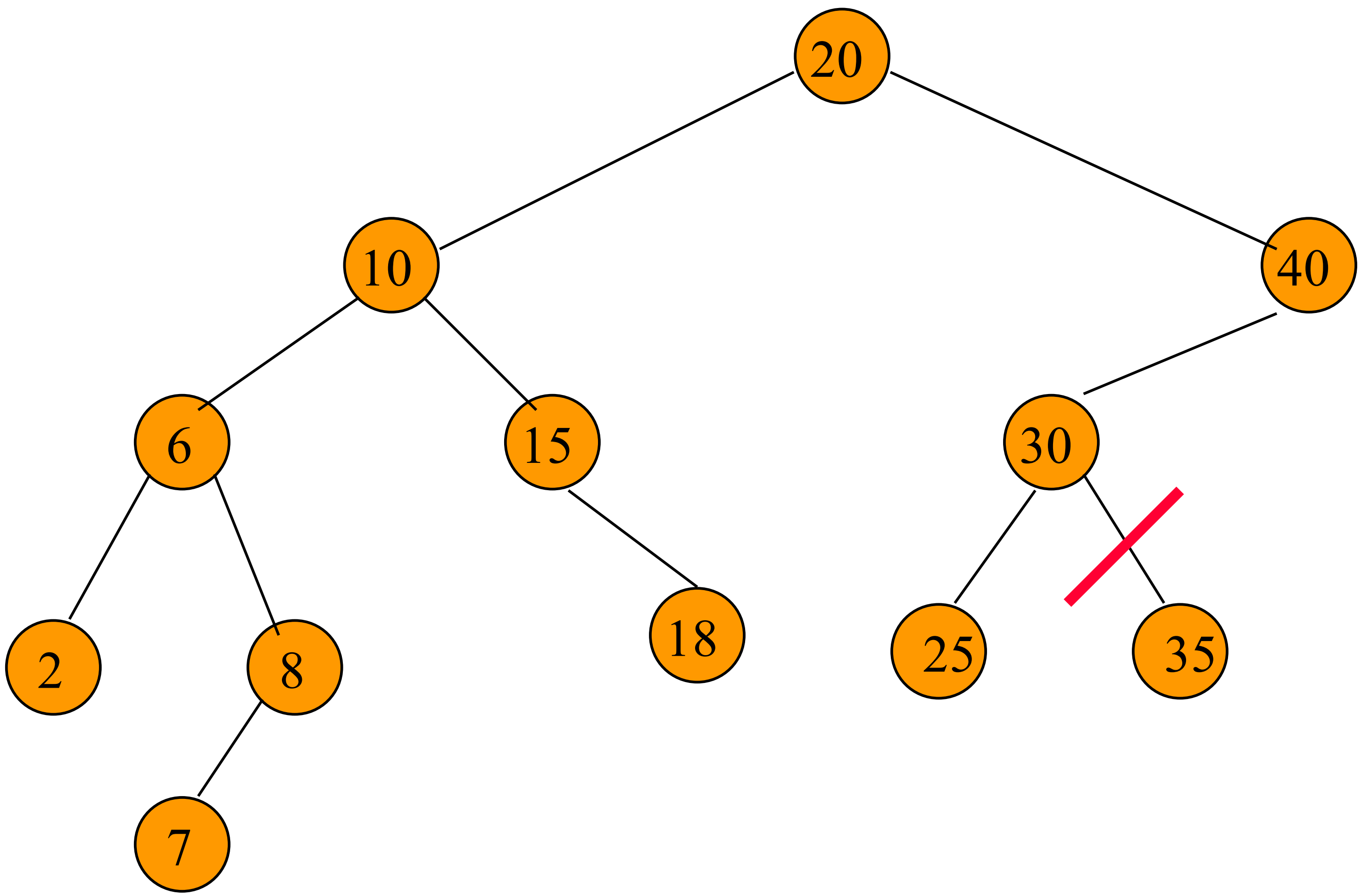
Thus reduces to case 0 or case 1.

Remove From A Leaf



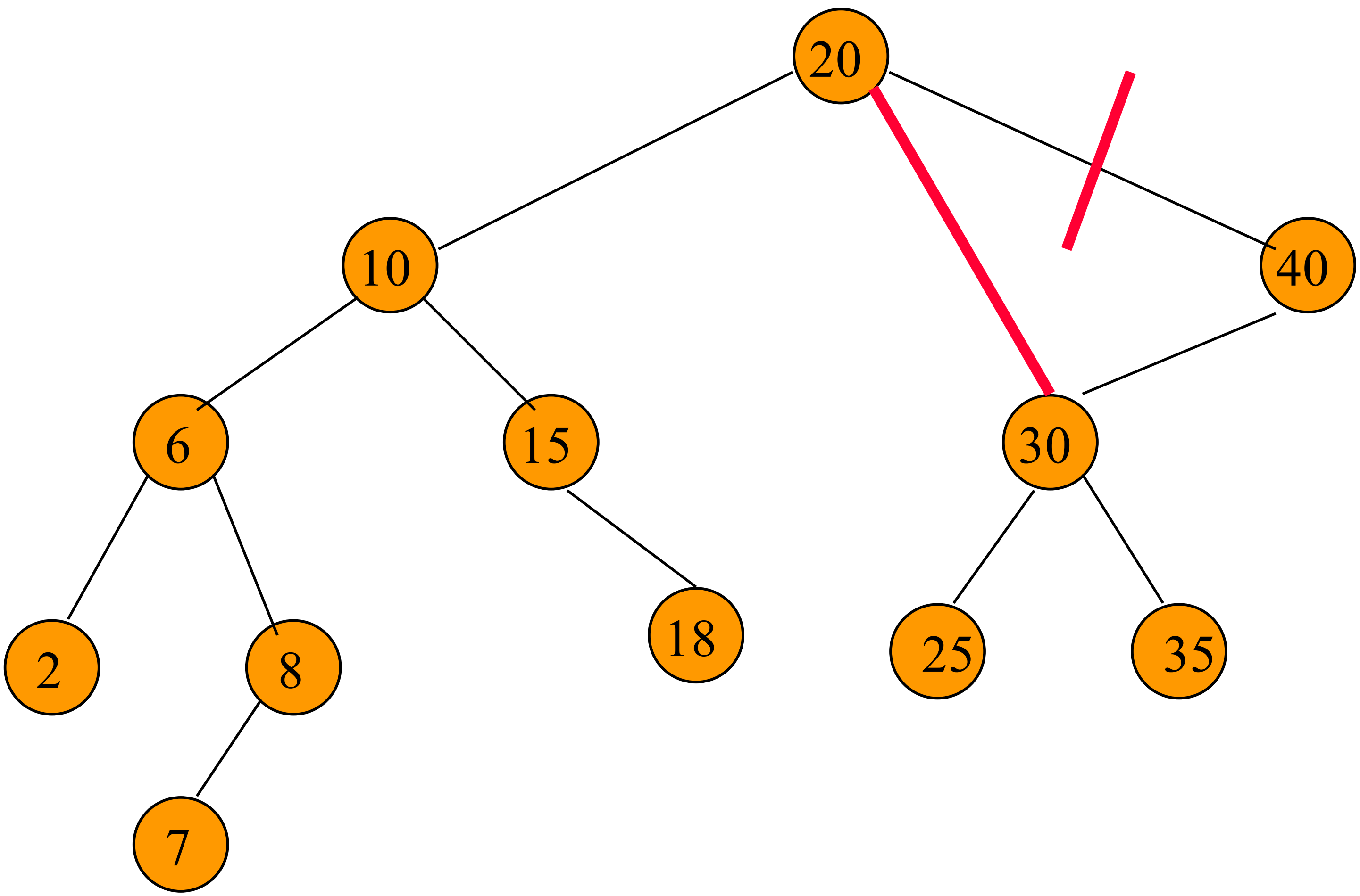
Remove a leaf element. key = 7

Remove From A Leaf (contd.)



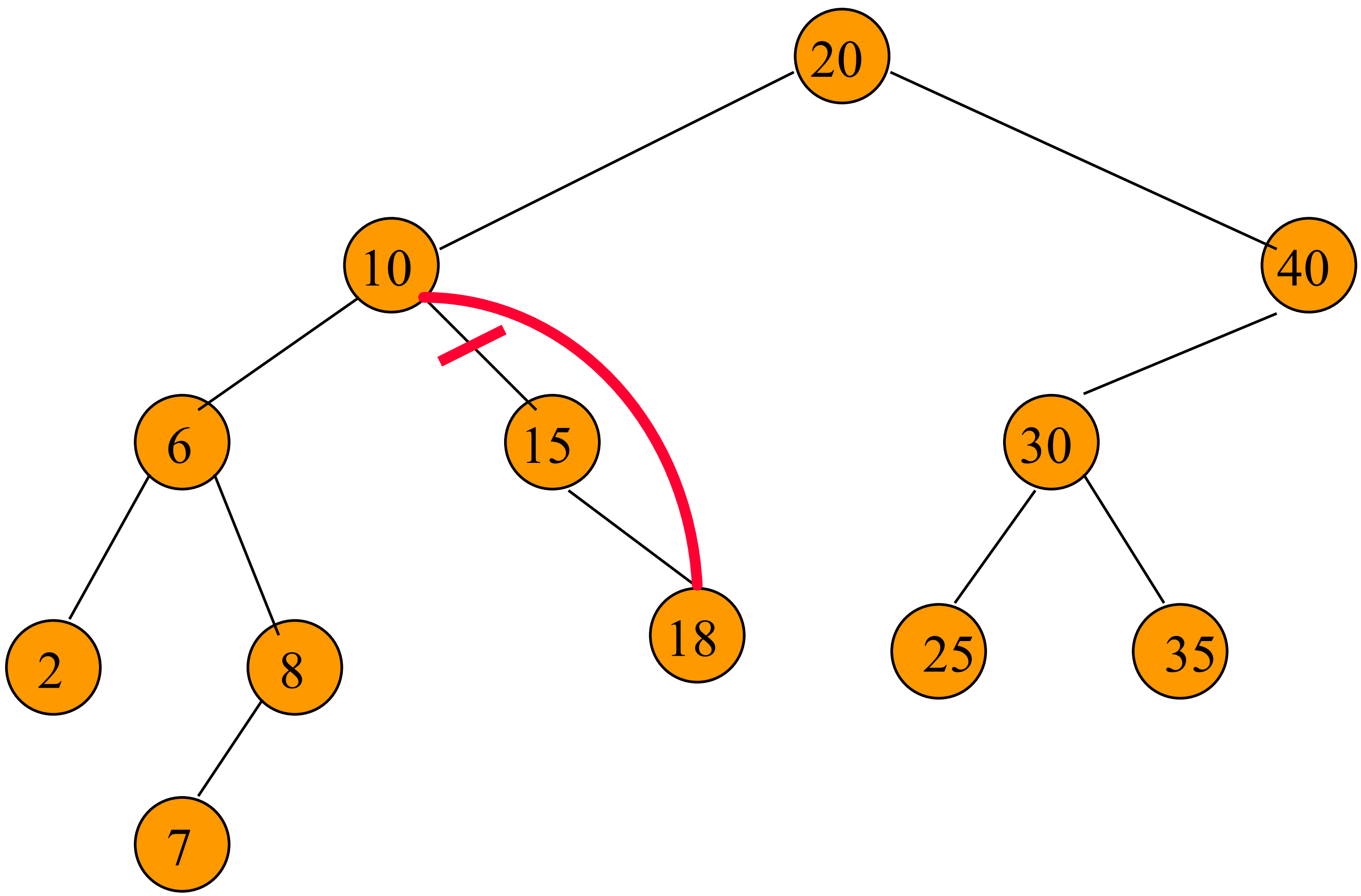
Remove a leaf element. key = 35

Remove From A Degree 1 Node



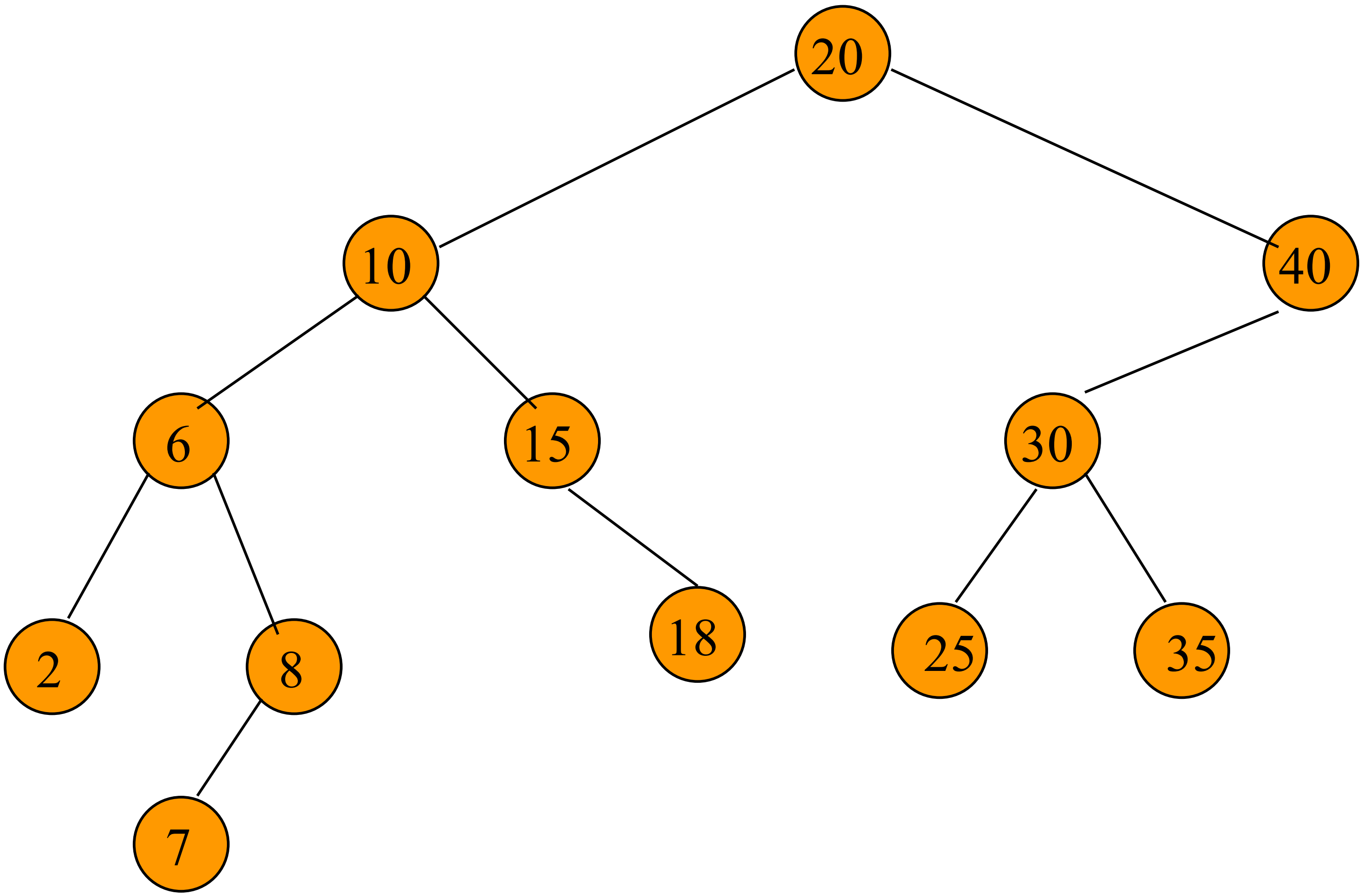
Remove from a degree 1 node. key = 40

Remove From A Degree 1 Node (contd.)



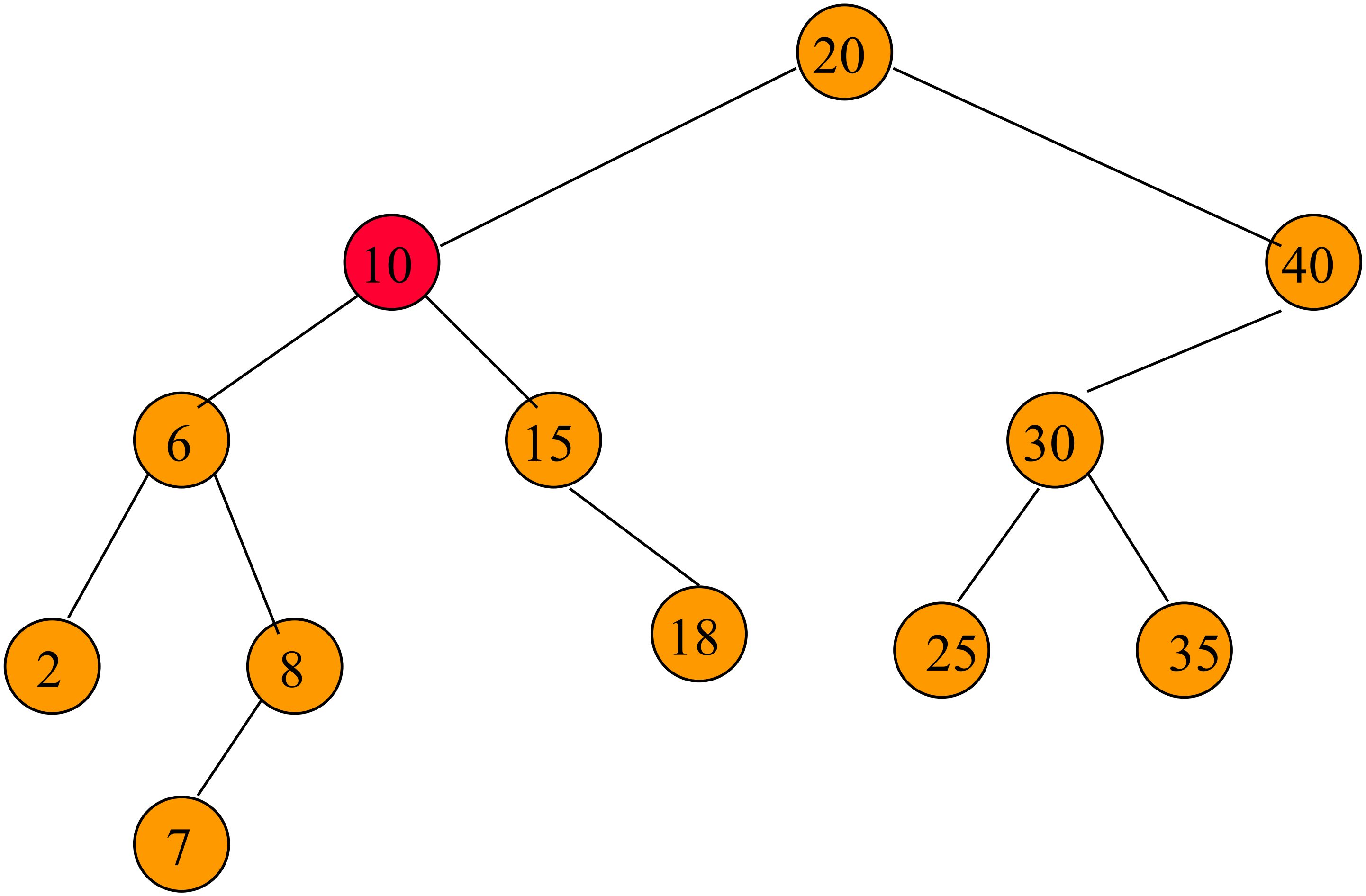
Remove from a degree 1 node. key = 15

Remove From A Degree 2 Node



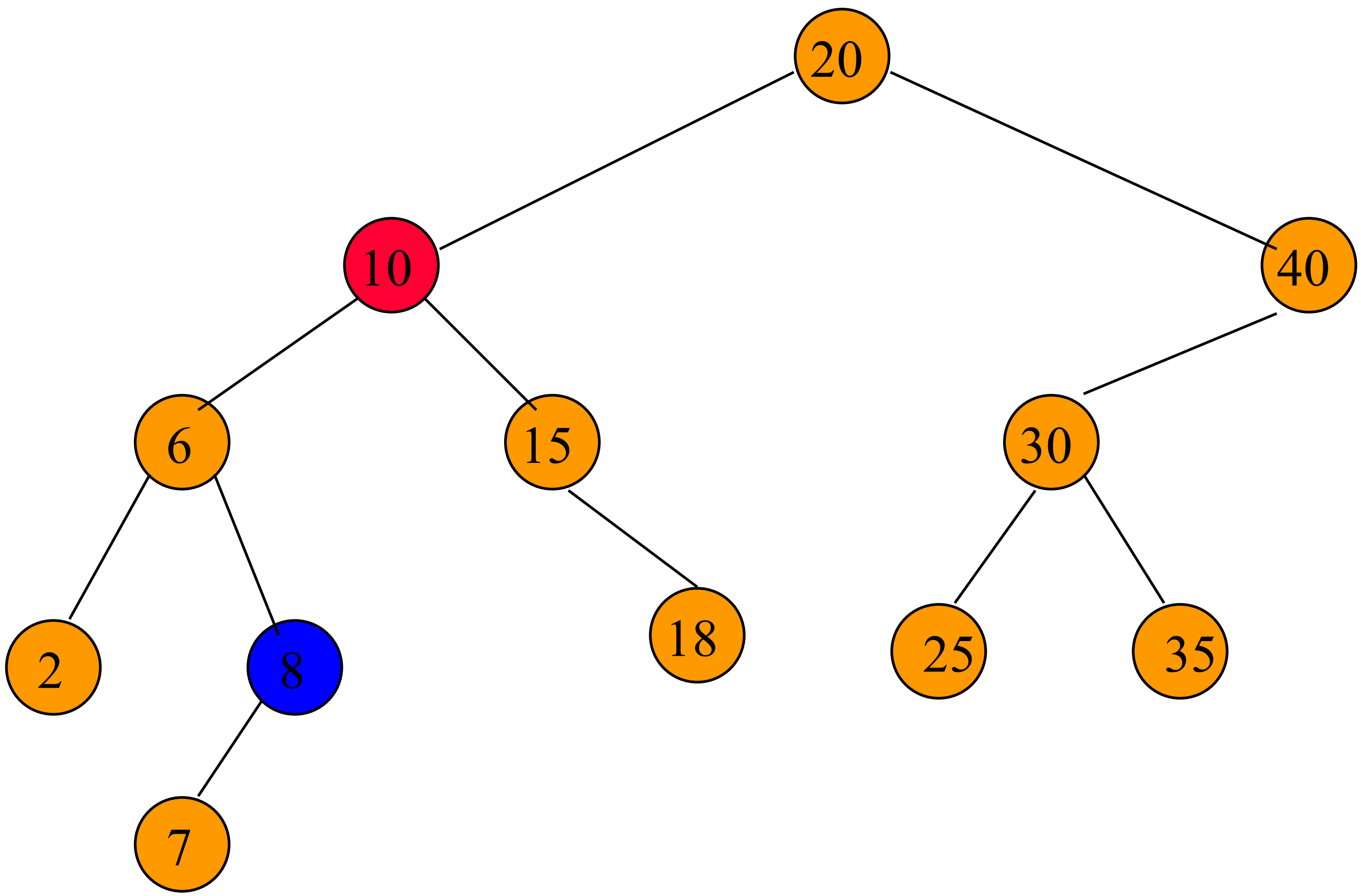
Remove from a degree 2 node. key = 10

Remove From A Degree 2 Node



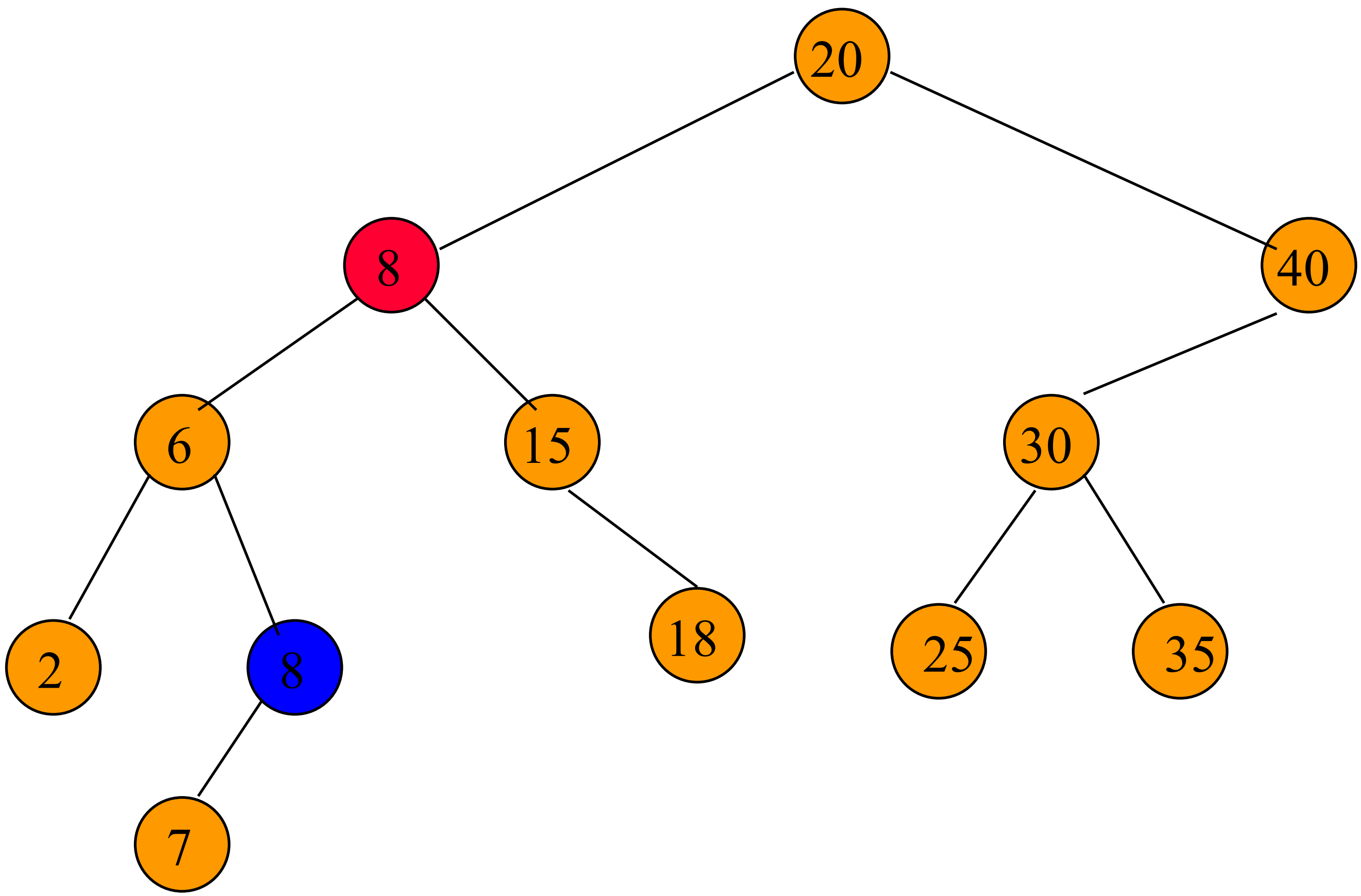
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



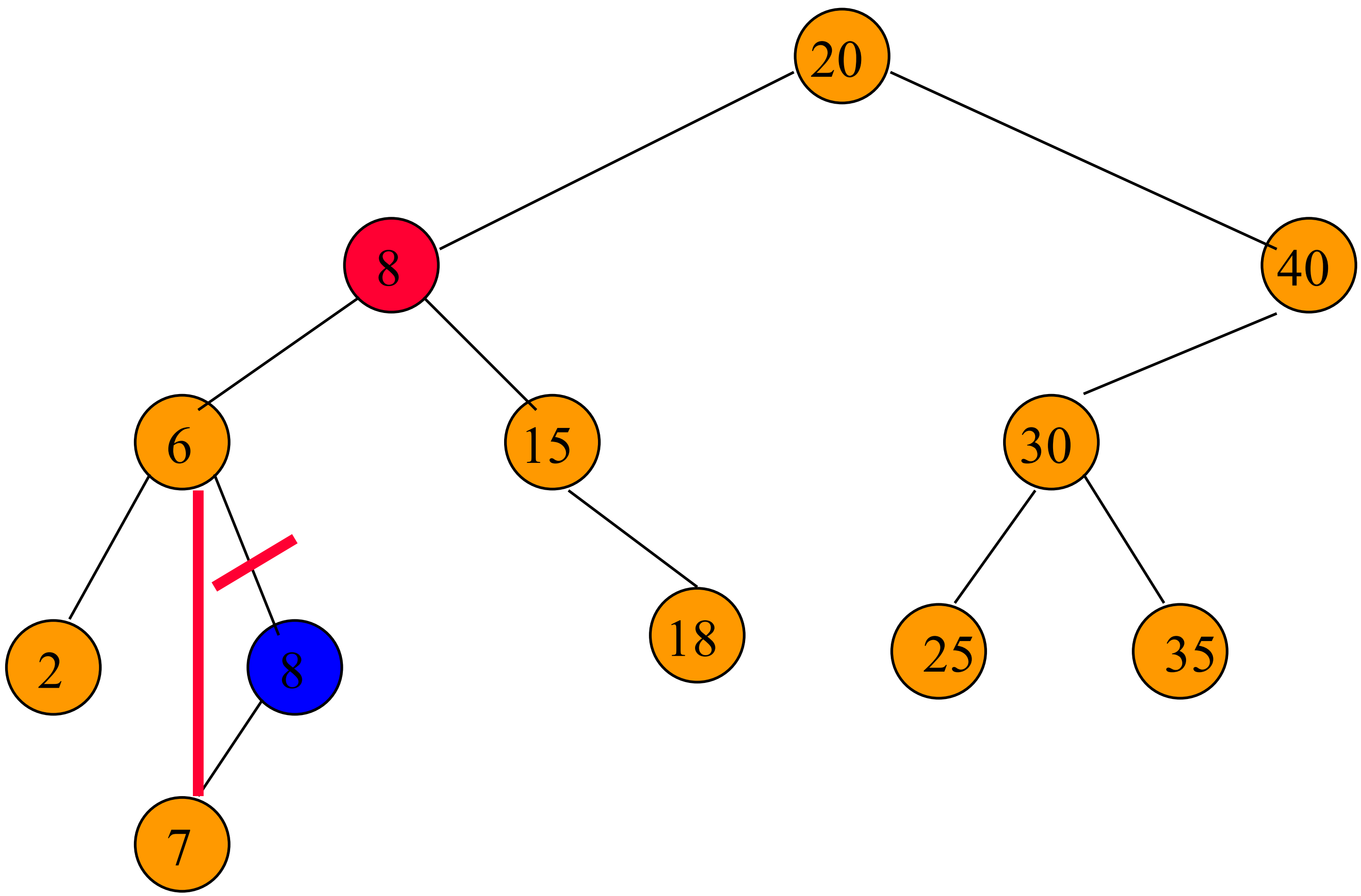
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



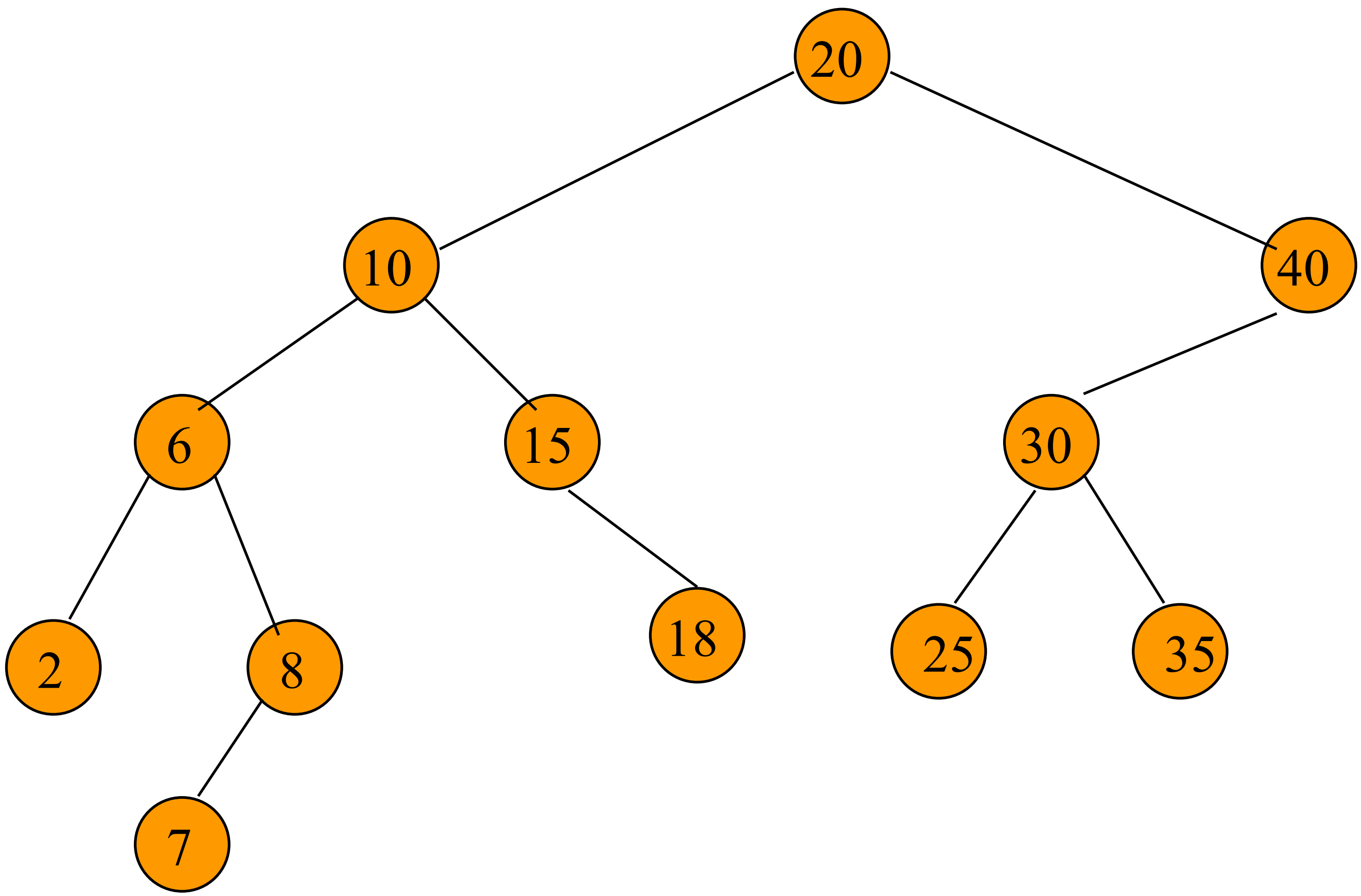
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



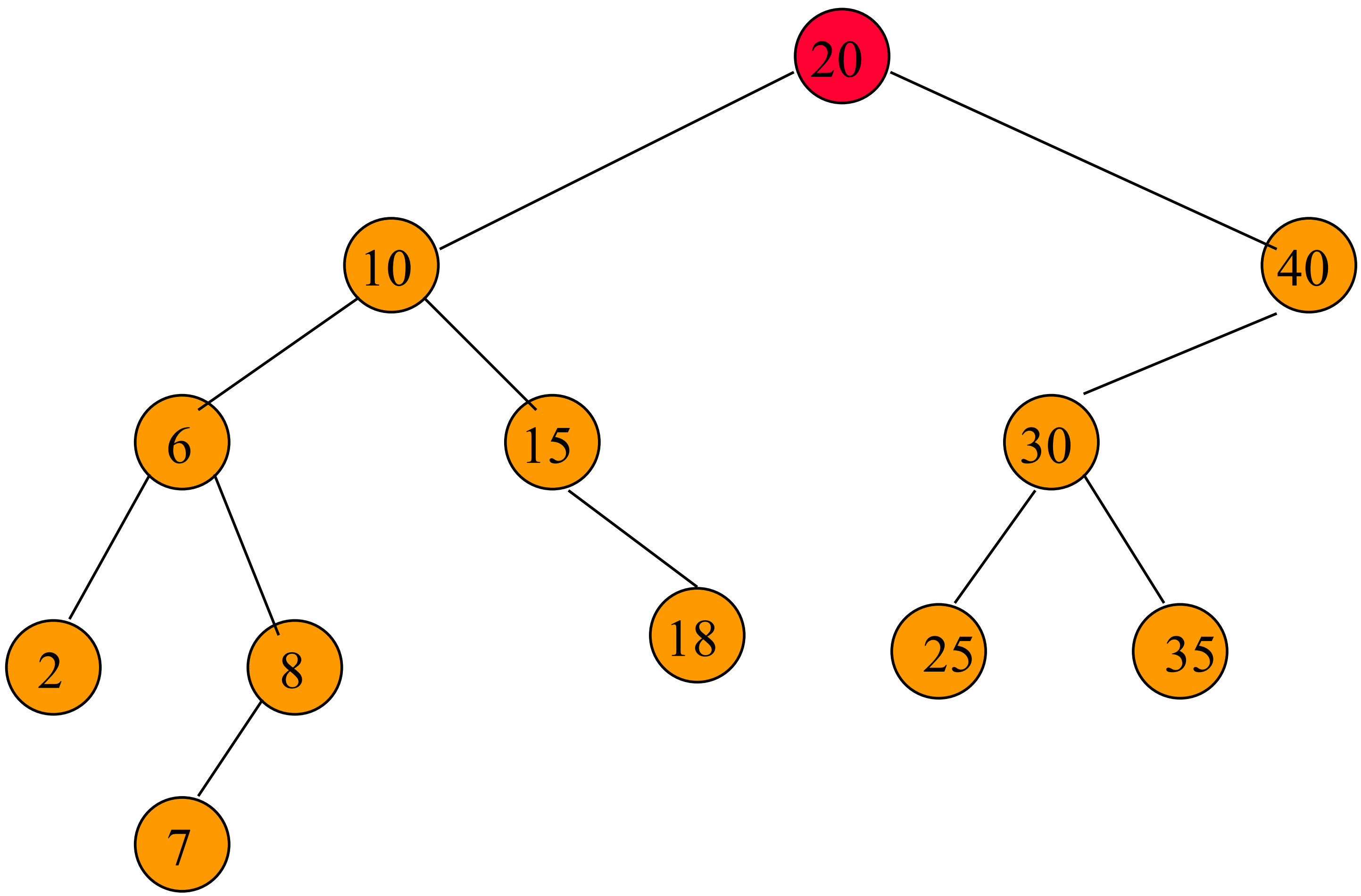
Largest key must be in a leaf or degree 1 node.

Another Remove From A Degree 2 Node



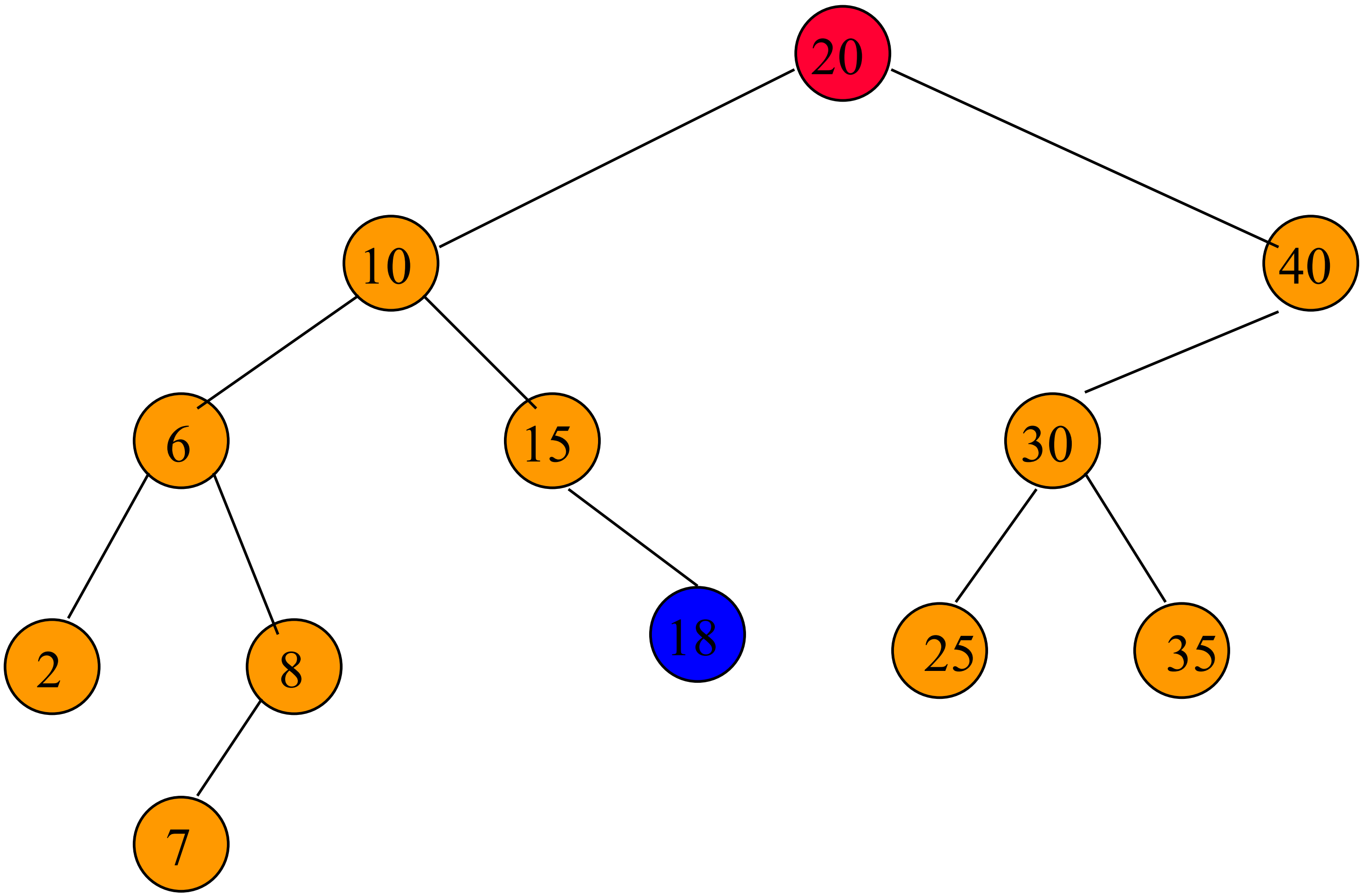
Remove from a degree 2 node. key = 20

Remove From A Degree 2 Node



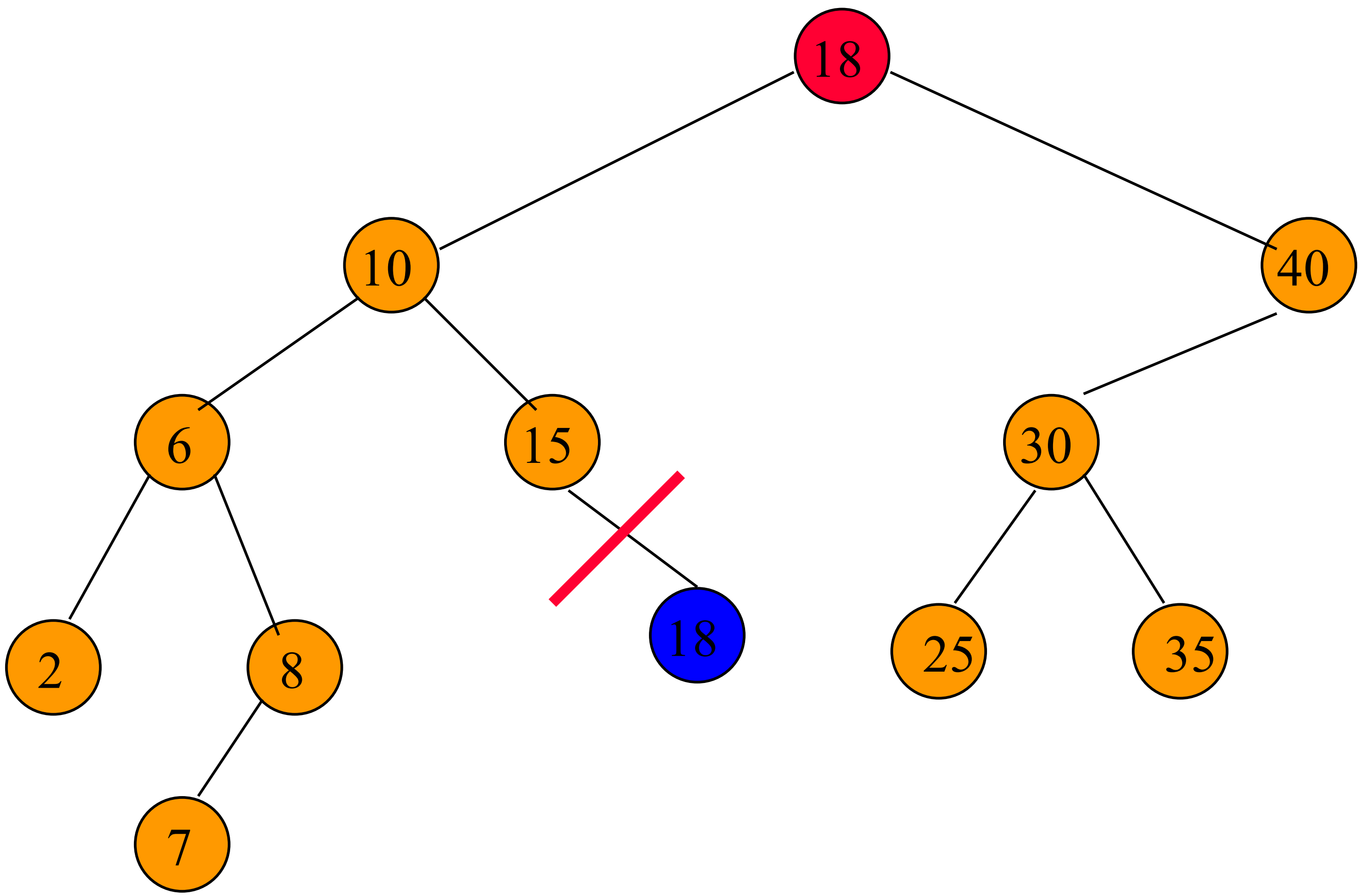
Replace with largest in left subtree.

Remove From A Degree 2 Node



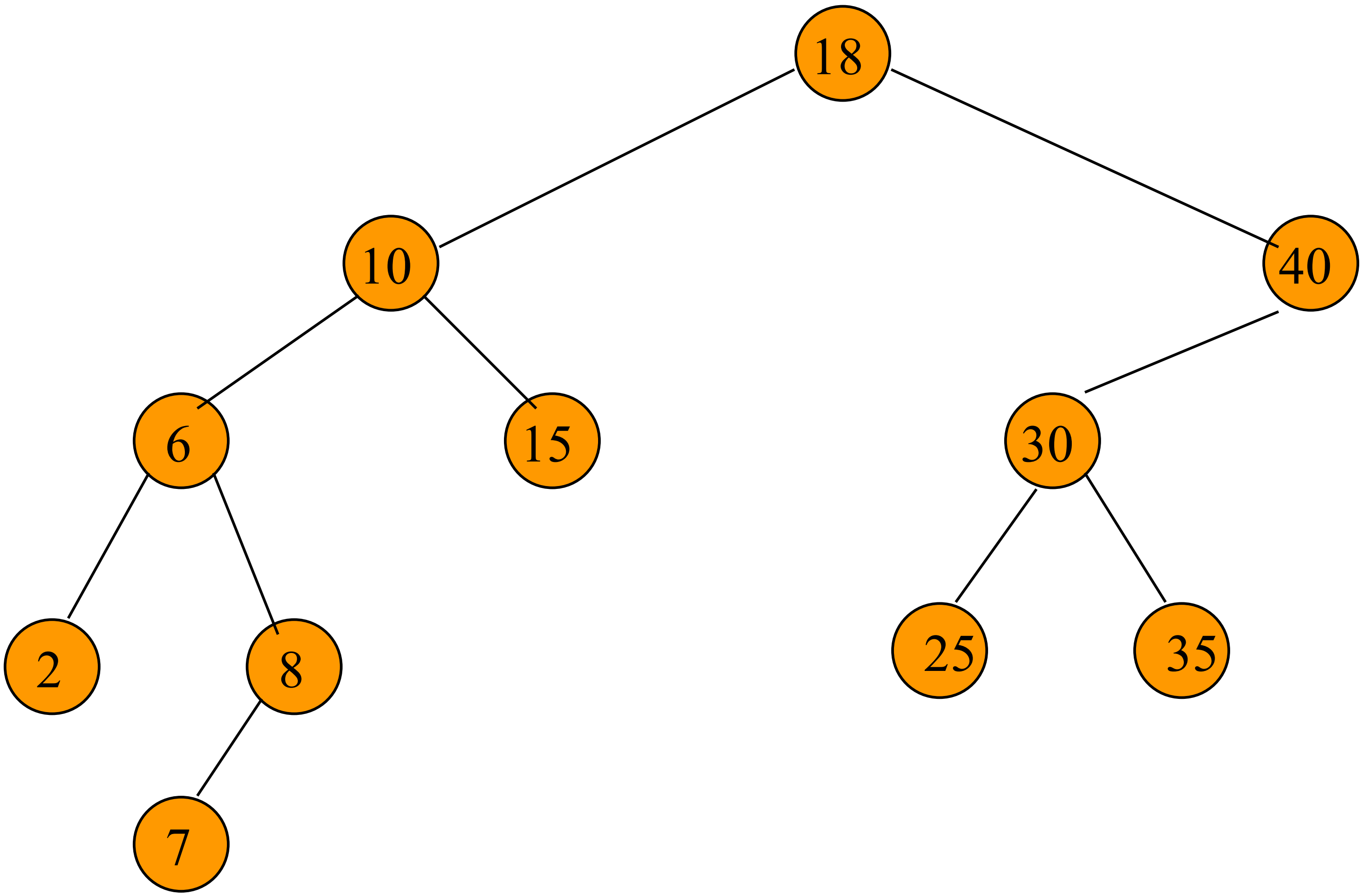
Replace with largest in left subtree.

Remove From A Degree 2 Node



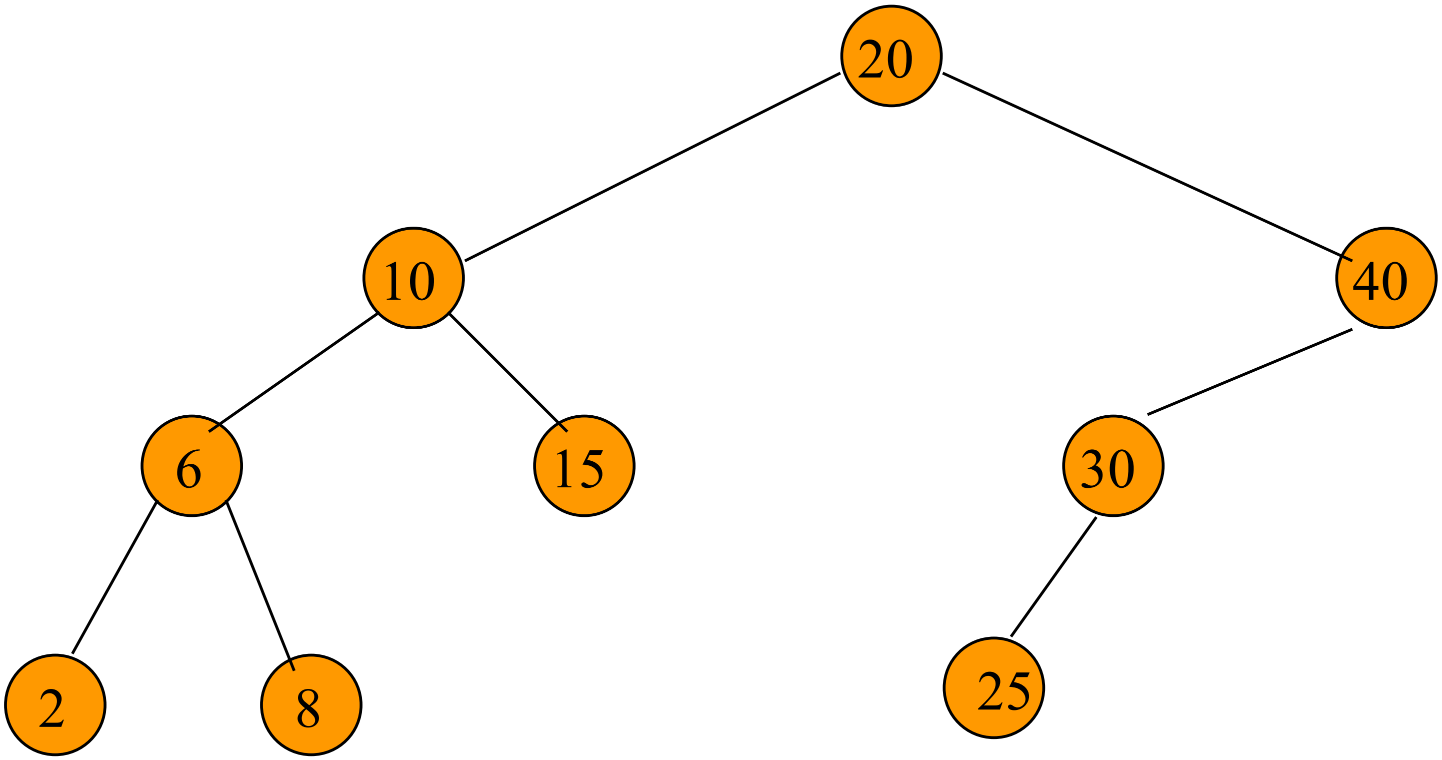
Replace with largest in left subtree.

Remove From A Degree 2 Node



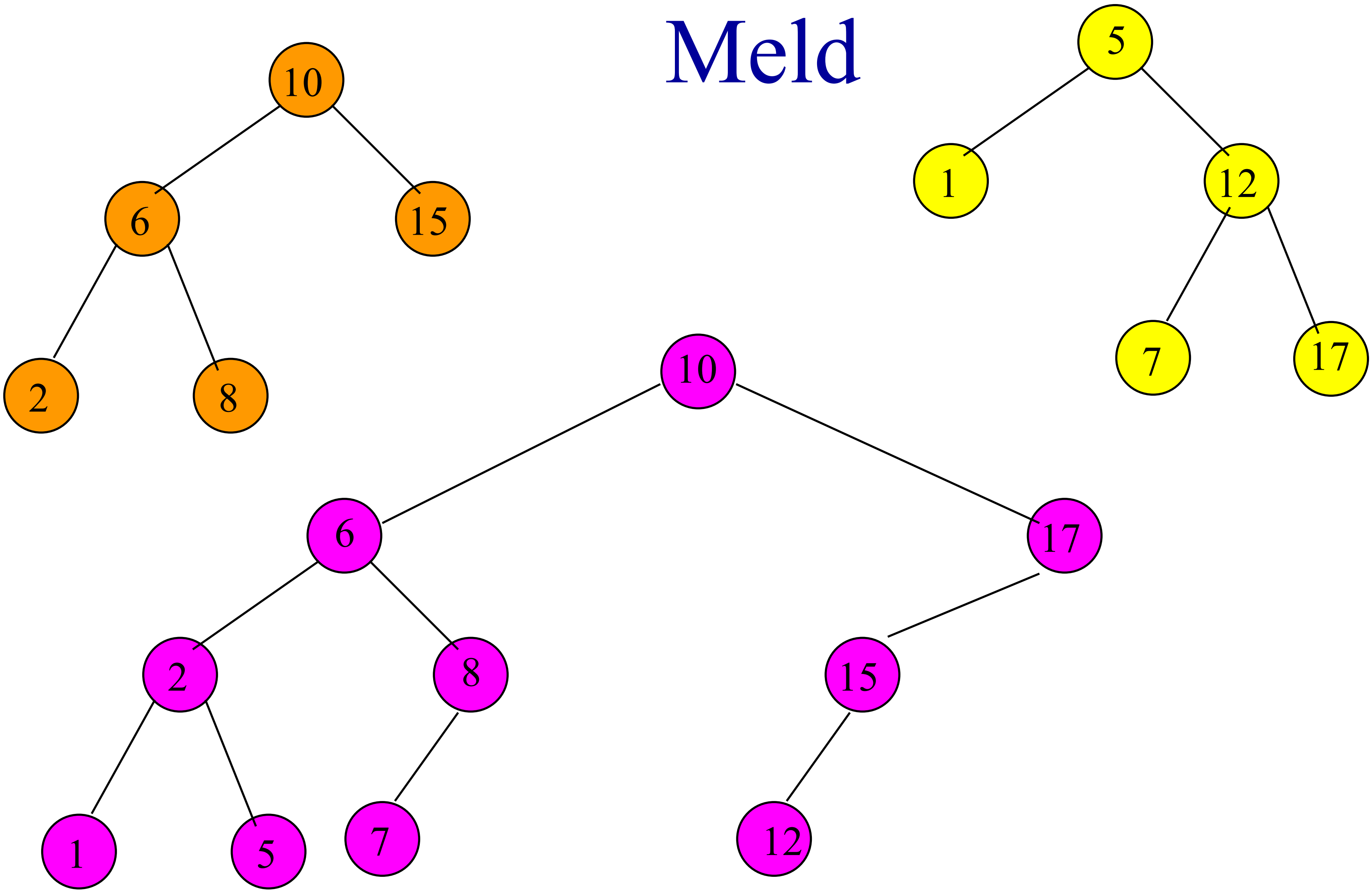
Complexity is O(height).

Initialize



- Sort n elements.
  - Initialize search tree.
  - Output in inorder (O(n)).
- Initialize must take O(n log n) time, because it isn't possible to sort faster than O(n log n).

Meld



Balanced Search Trees

- Height balanced.
  - ✓AVL (Adelson-Velsky and Landis) trees
  - ✓Red-black trees
- Degree Balanced.
  - ✓2-3 trees
  - ✓2-3-4 trees
  - ✓B-trees
  - ✓Red-black trees