# Slides for BST Q&A

Instructors: Subrahmanyam Kalyanasundaram
Karteek Sreenivasaiah

7th September 2020

# Plan

We saw why Binary Search Trees are important.

# Plan

We saw why Binary Search Trees are important.
Today we see how it implements the following

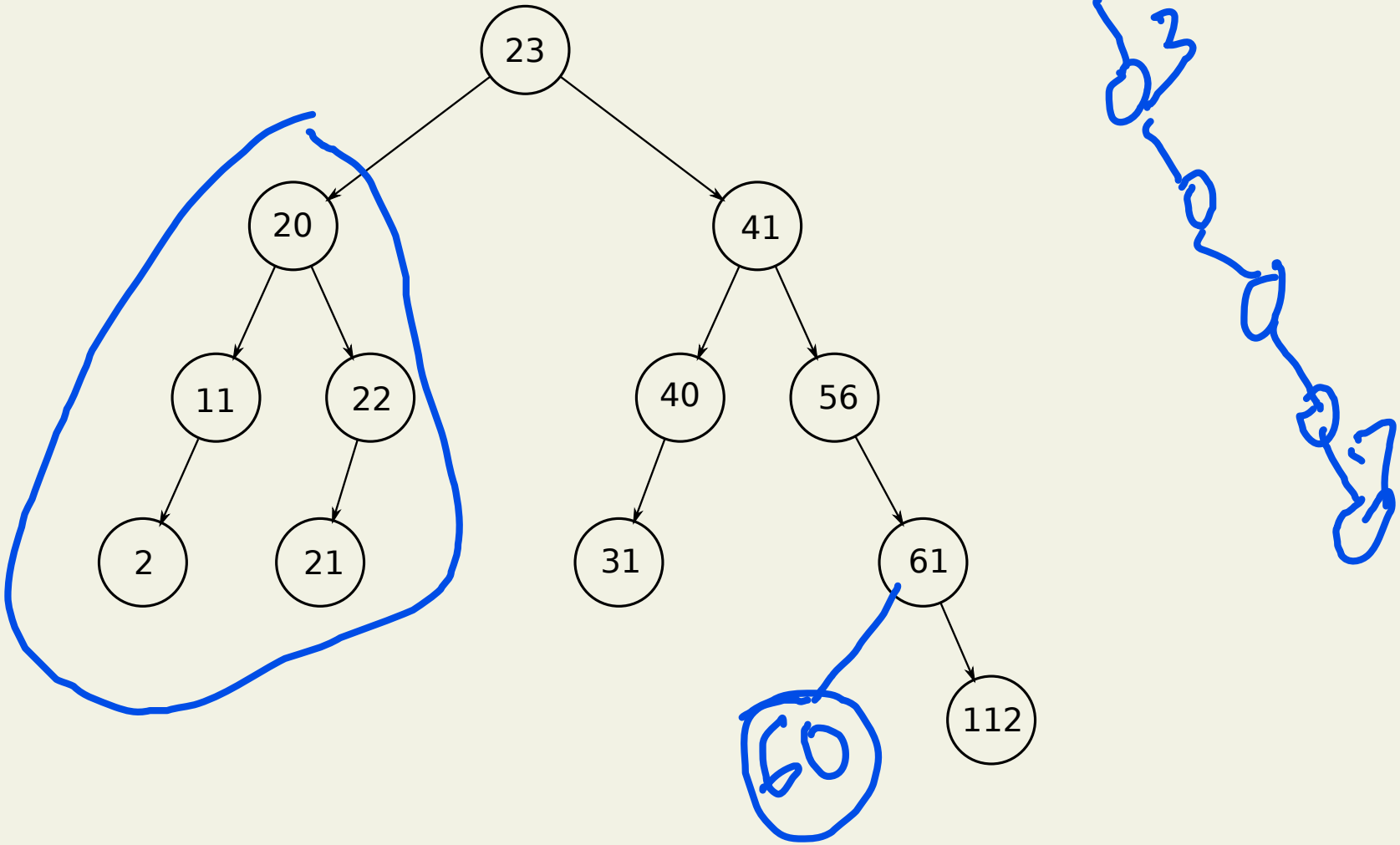
1. Search
2. Insert
3. Succ (also Pred)
4. Delete

# Binary Search Trees

Recall that a Binary Search Tree (BST) has the following crucial property:
For every node $X$ in the BST, we have:

- Every node in the left subtree of $X$ contains a value smaller than that of $X$.
- Every node in the right subtree of $X$ contains a value larger than that of $X$.

# Example BST



# INSERT procedure

The INSERT($node, x$) procedure:

- If $node = \text{NULL}$, create new node with $x$ and attach to parent.
- Else If $x <$ value($node$),
  - INSERT($node \rightarrow left, x$)
- Else If $x >$ value($node$) Then,
  - INSERT($node \rightarrow right, x$)

# SEARCH procedure

- SEARCH($node, x$):
- If $node$ = NULL, then return NULL
- Else If $x$ = value($node$), then return $node$
- Else If $x <$ value($node$), then
  - Return SEARCH($node \rightarrow left, x$)
- Else
  - Return SEARCH($node \rightarrow right.x$)

# Successor

The SUCC(VAL) procedure is as follows:

- Find the node which stores *val*. Refer to this node as "*node*".
- Two cases:

# Successor

The SUCC(VAL) procedure is as follows:

- Find the node which stores *val*. Refer to this node as "*node*".
- Two cases:

Case 1: *node* has a right child.
Find the smallest element of the right subtree.

# Successor

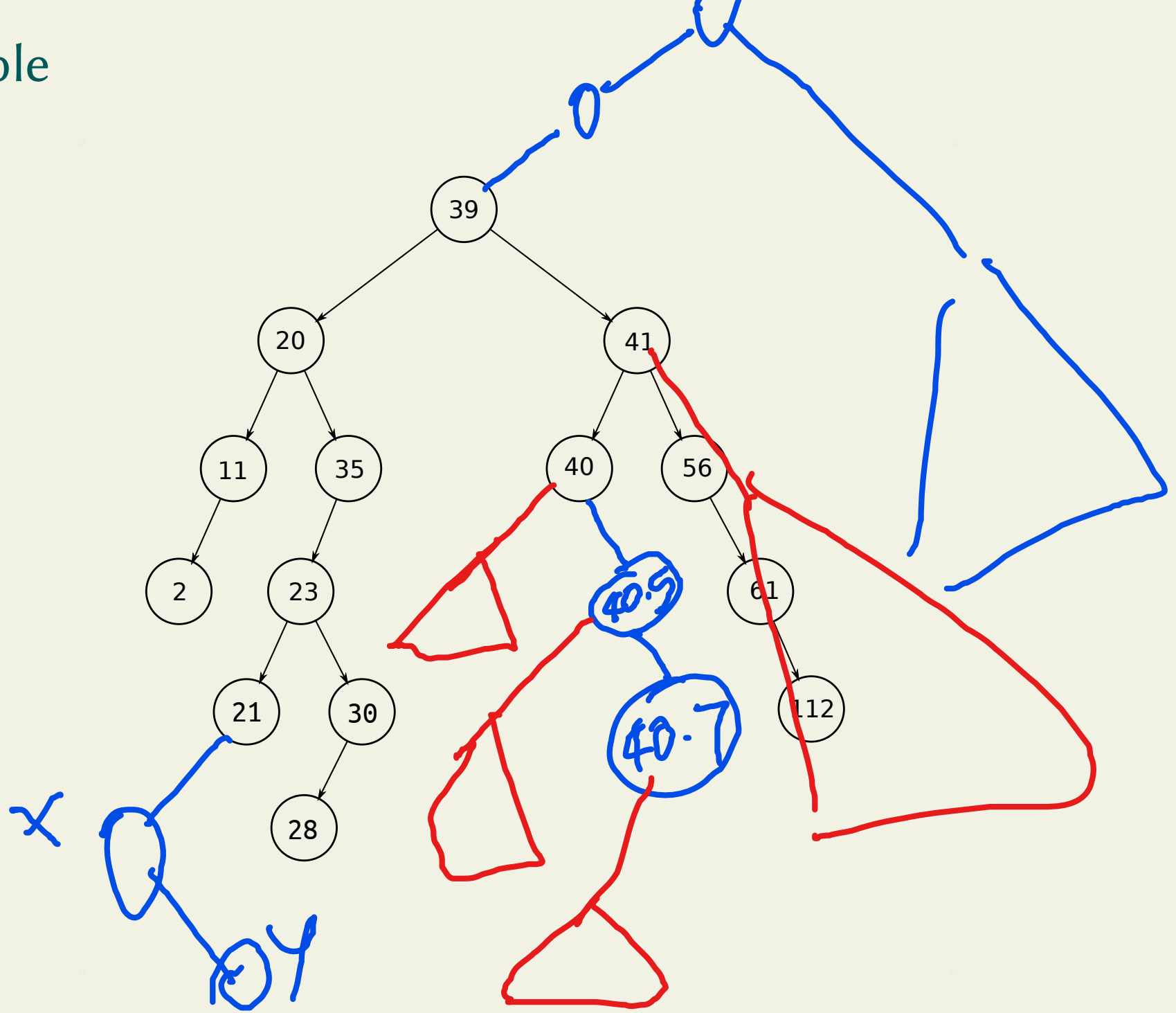
The SUCC(VAL) procedure is as follows:

- Find the node which stores *val*. Refer to this node as "*node*".
- Two cases:

Case 1: *node* has a right child.
Find the smallest element of the right subtree.

Case 2: *node* does not have a right child.
Keep going to the parent till we reach an ancestor $x$ that is a left child of its parent. The parent of $x$ is the successor.

# Example



# Delete

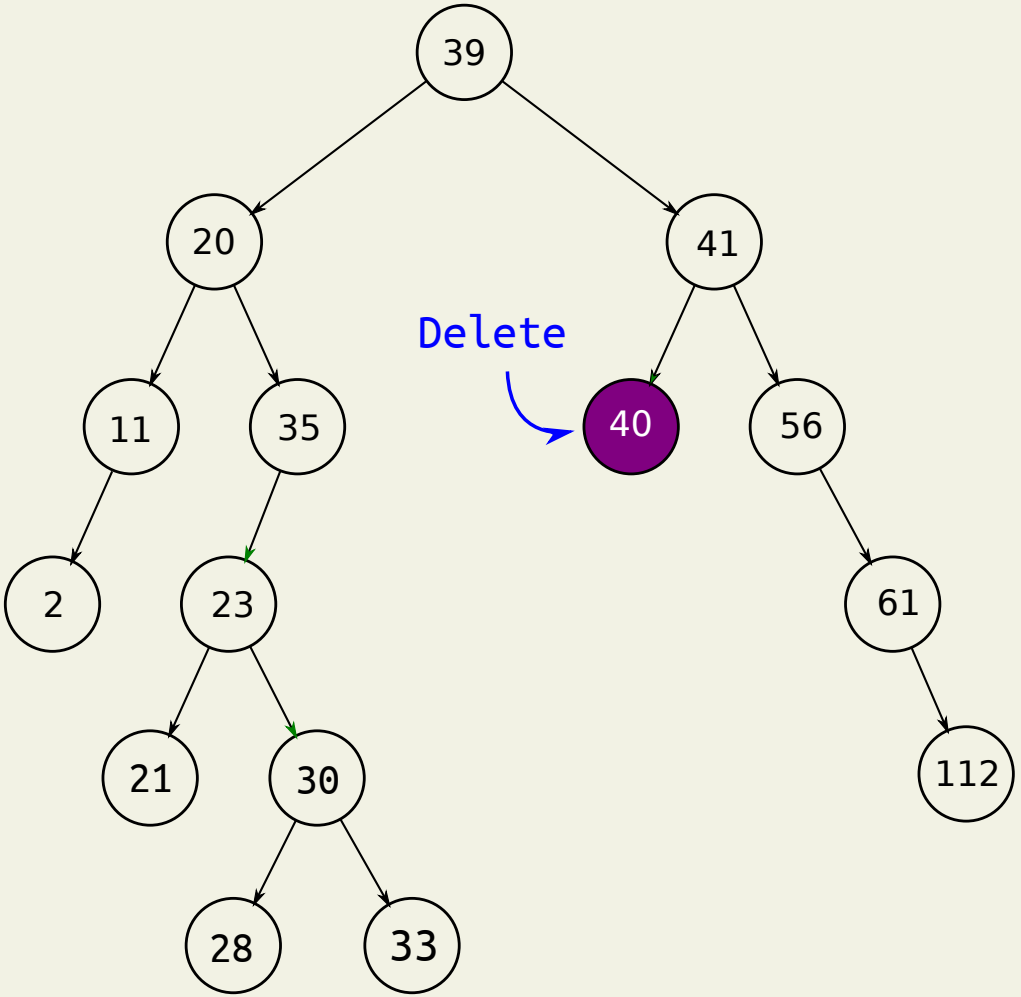
The Delete(val) procedure is as follows:
Find the node that has value *val*.

# Delete

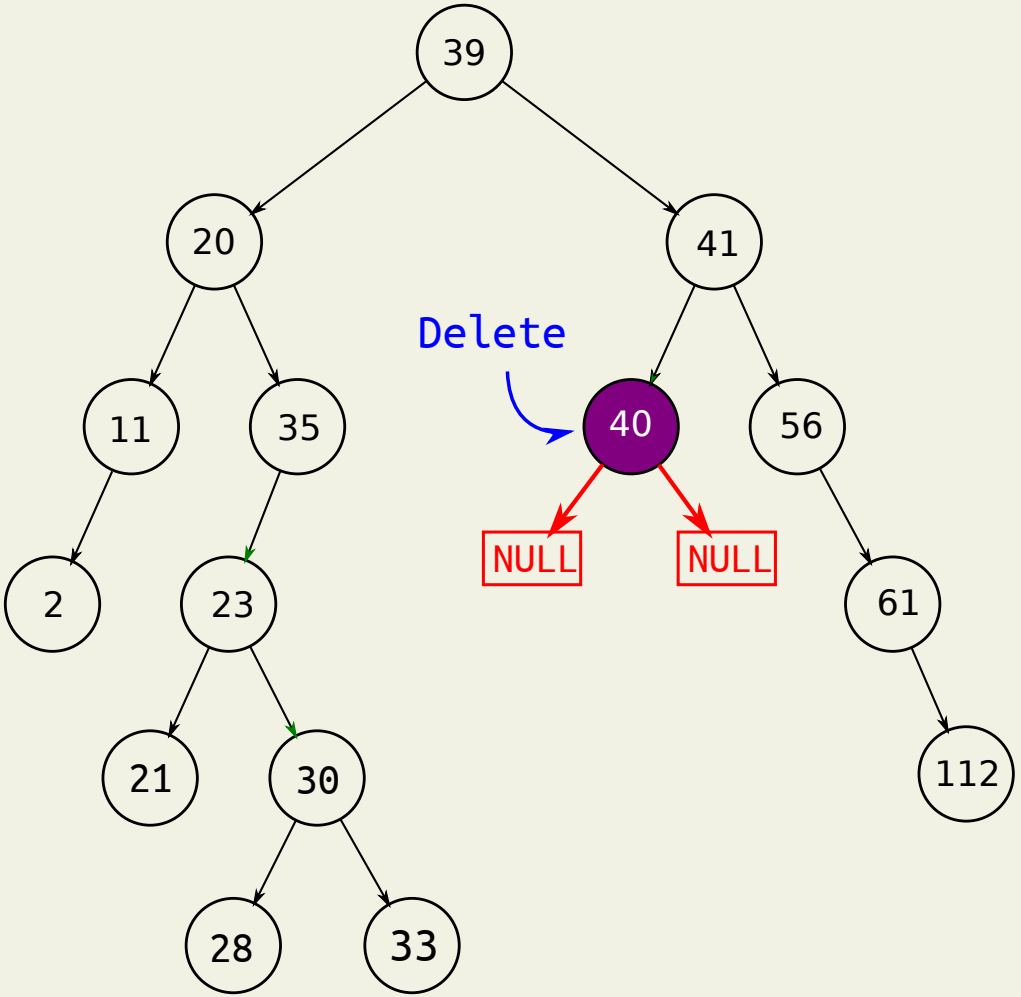
The Delete(val) procedure is as follows:
Find the node that has value *val*.
Three cases:

1. *node* has 0 children. (trivial)
2. *node* has 1 child. (splice)
3. *node* has 2 children:
   - Find successor node $X$ with value $x$.
   - Splice $X$ out of the tree.
   - Replace *val* with $x$.
   - Delete node $X$.

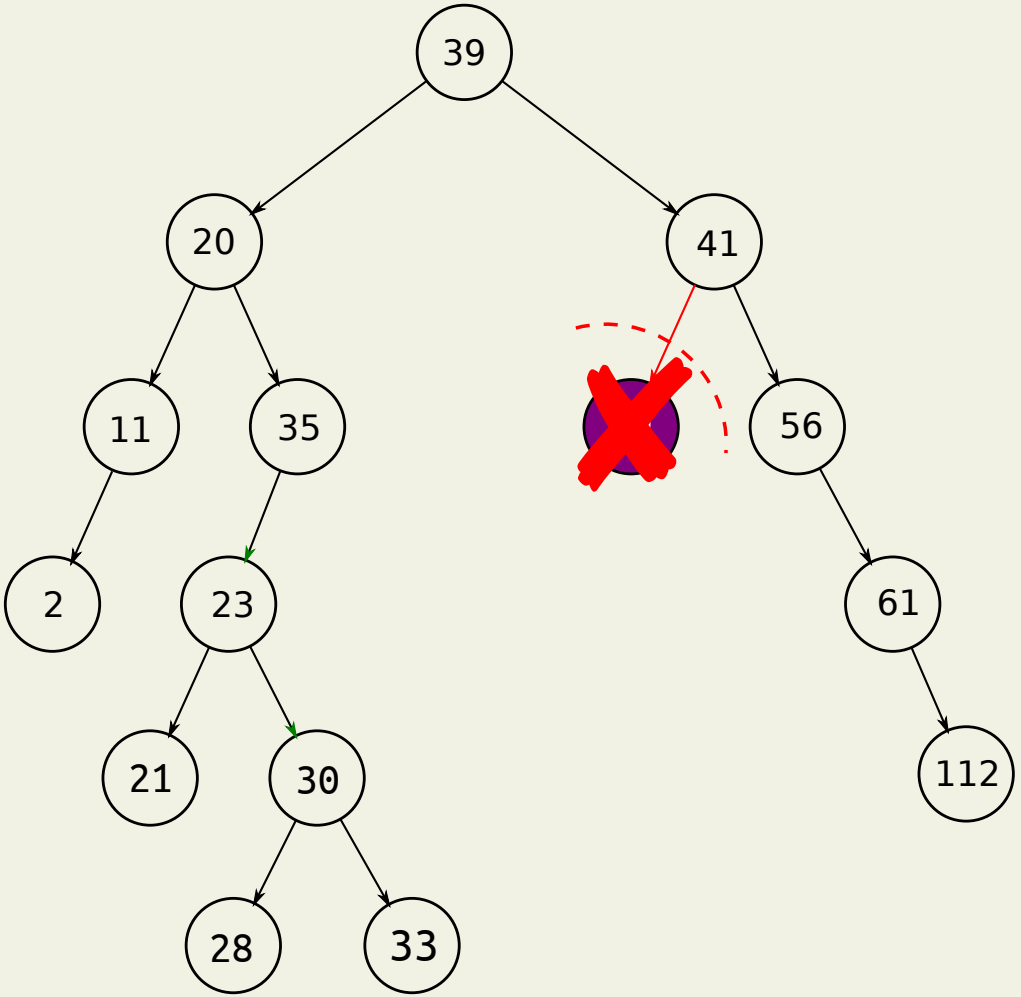
# Example



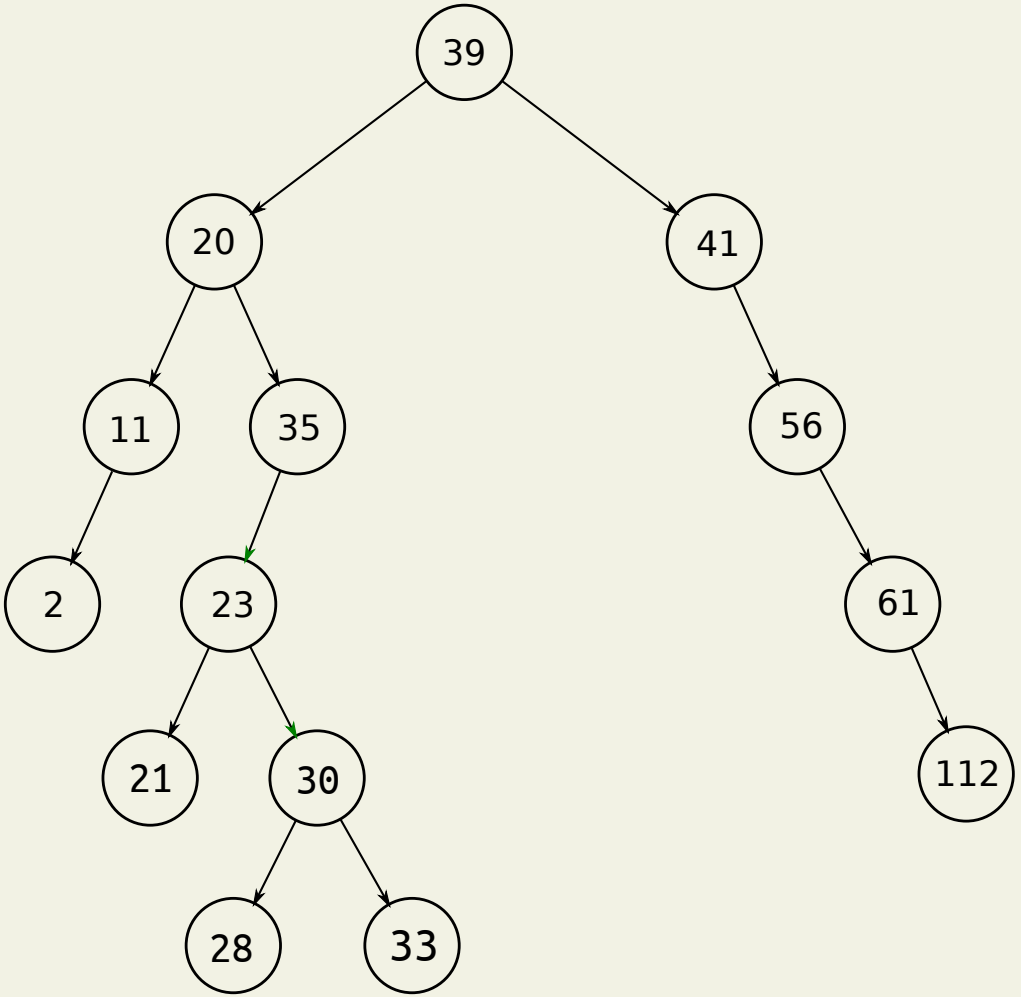
# Example



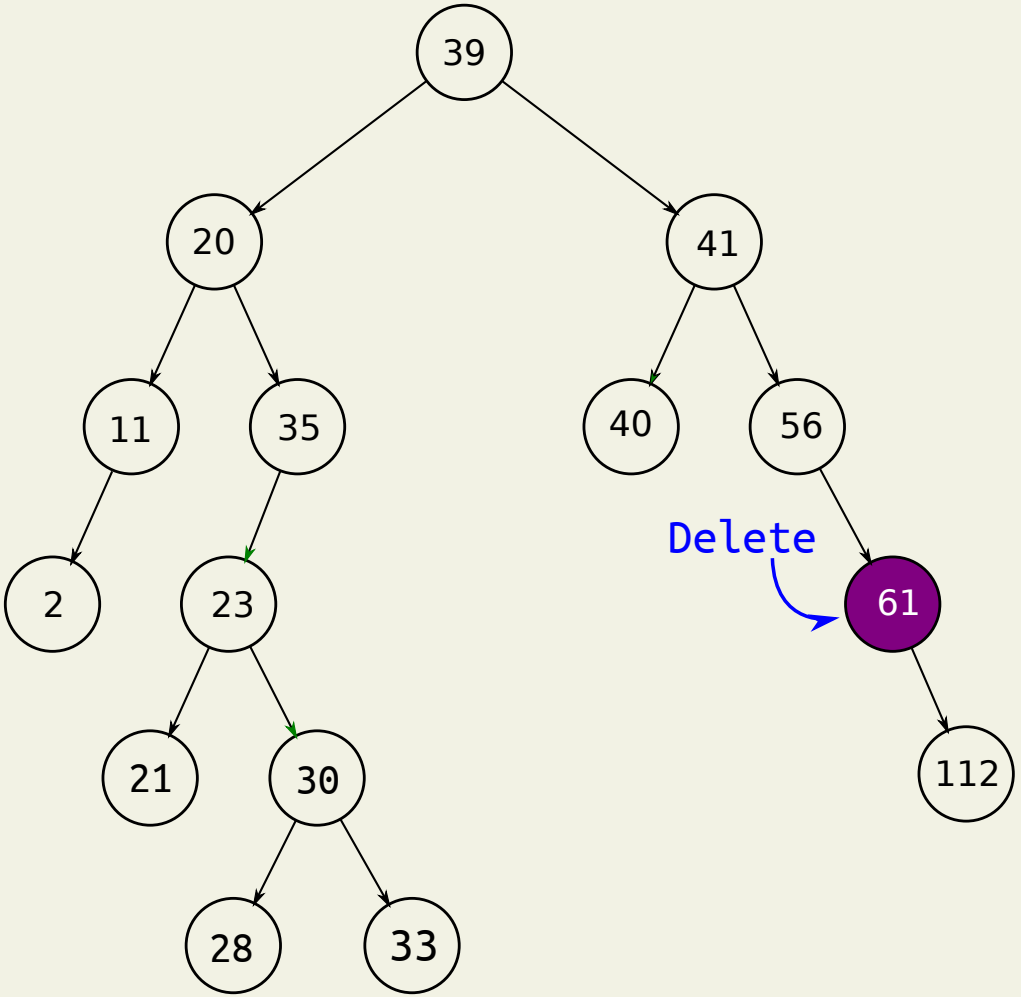
# Example



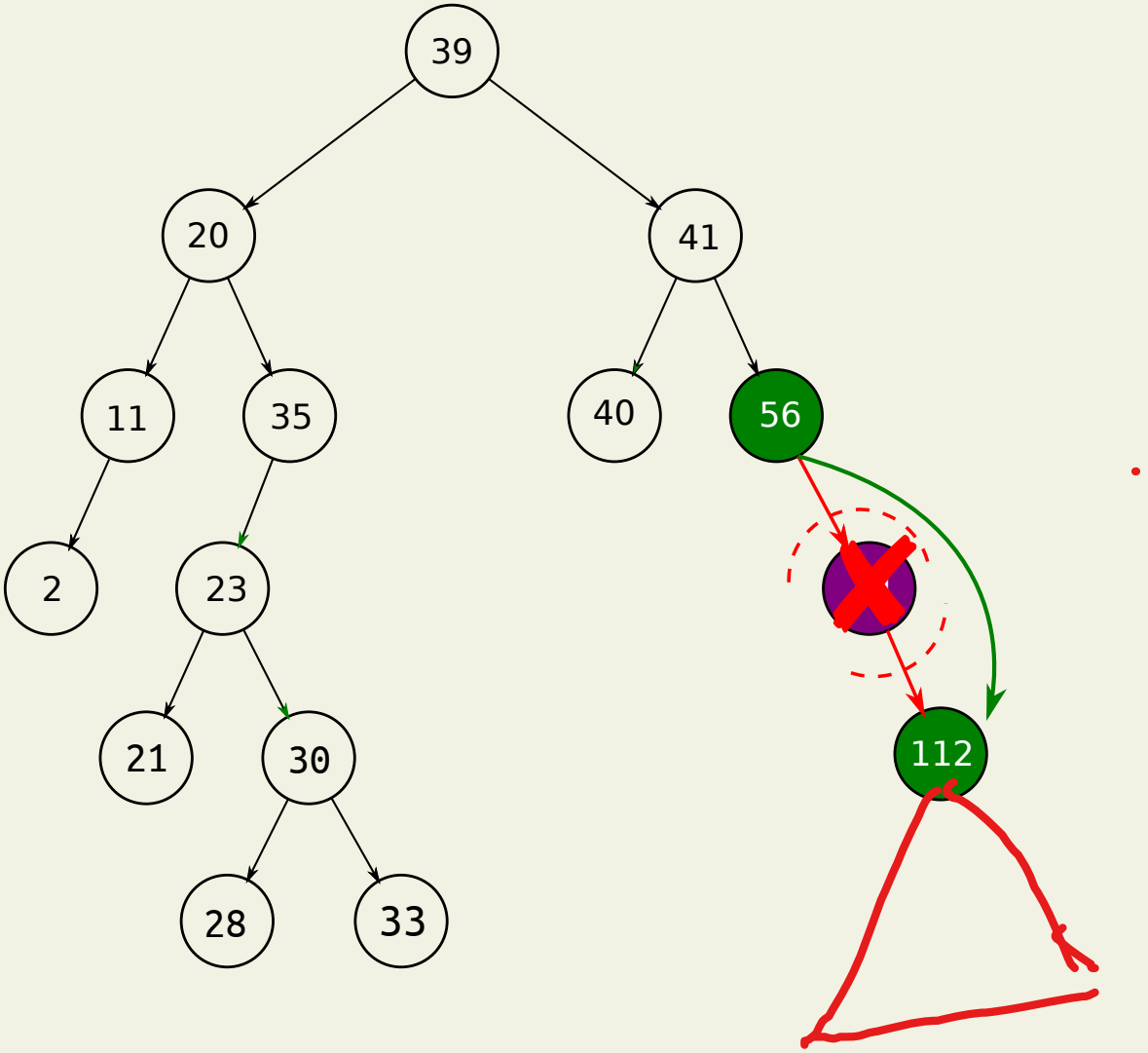
# Example



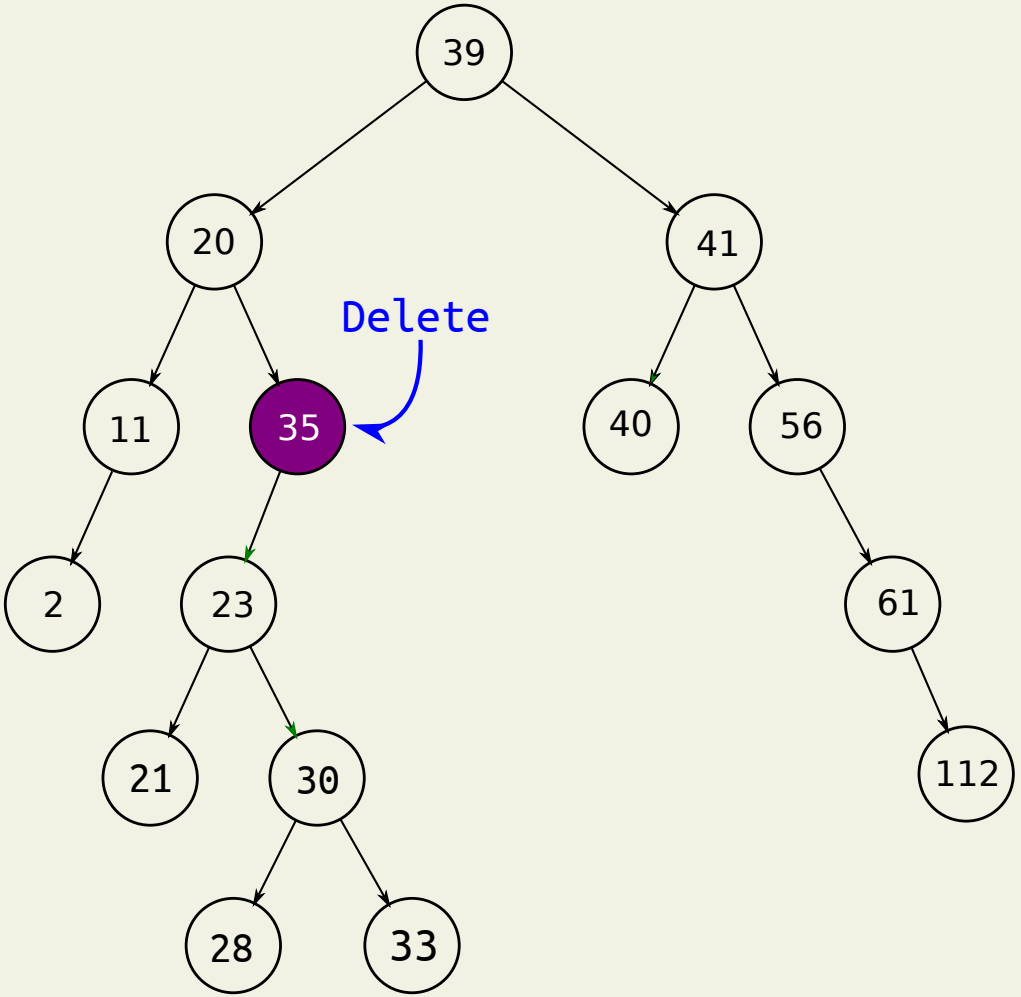
# Example



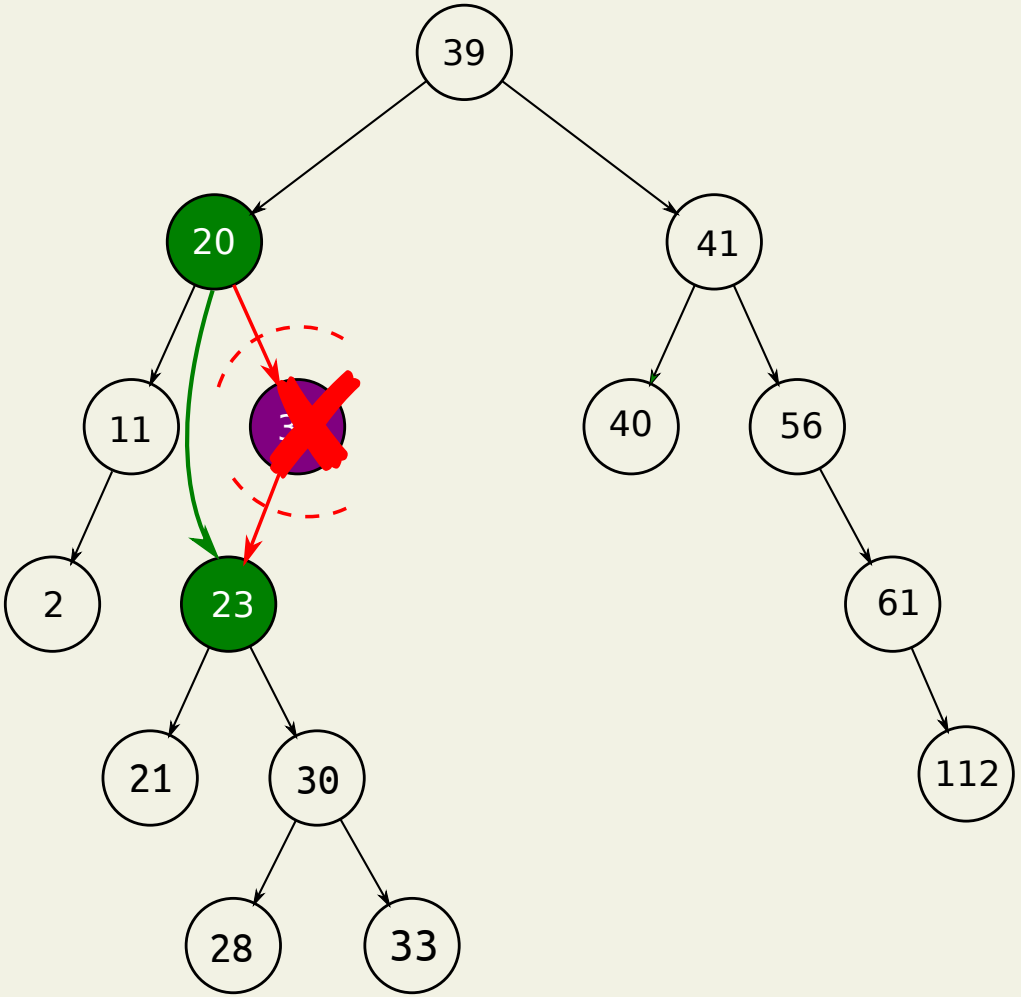
# Example



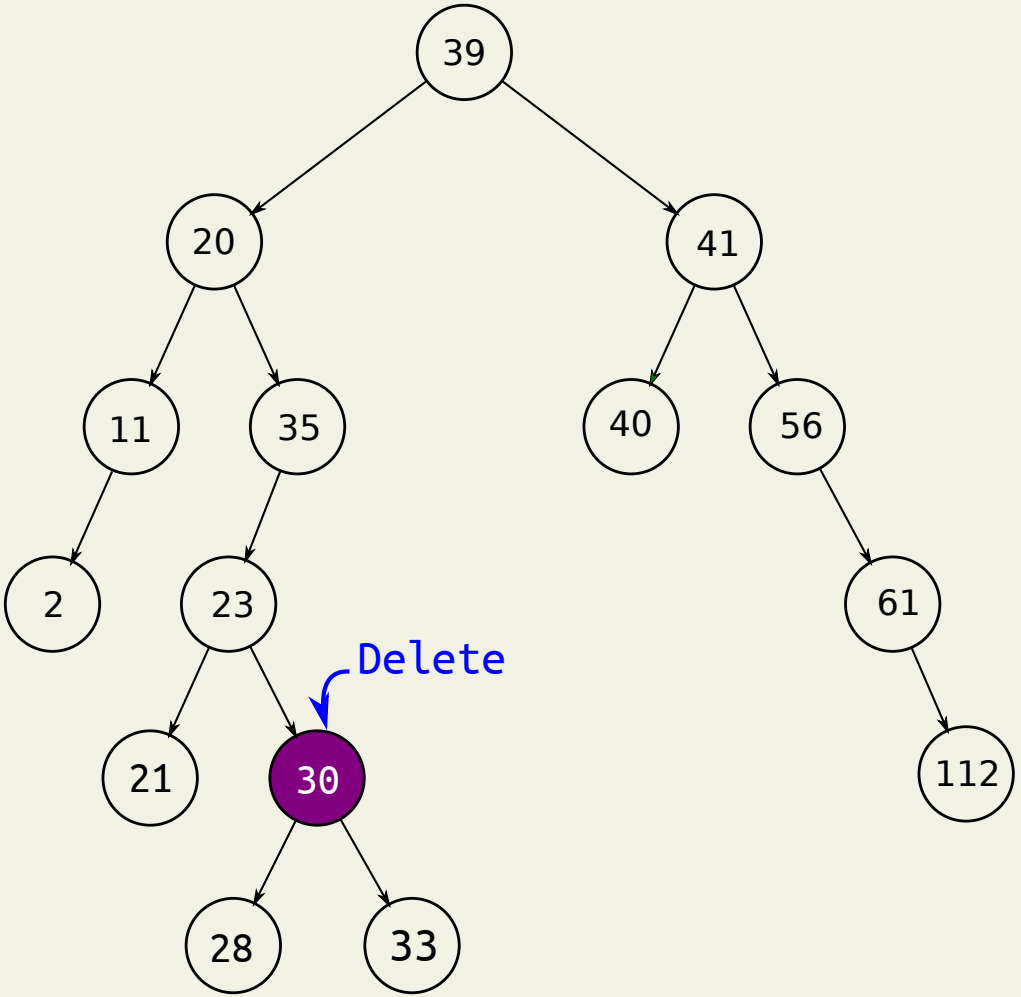
# Example



# Example

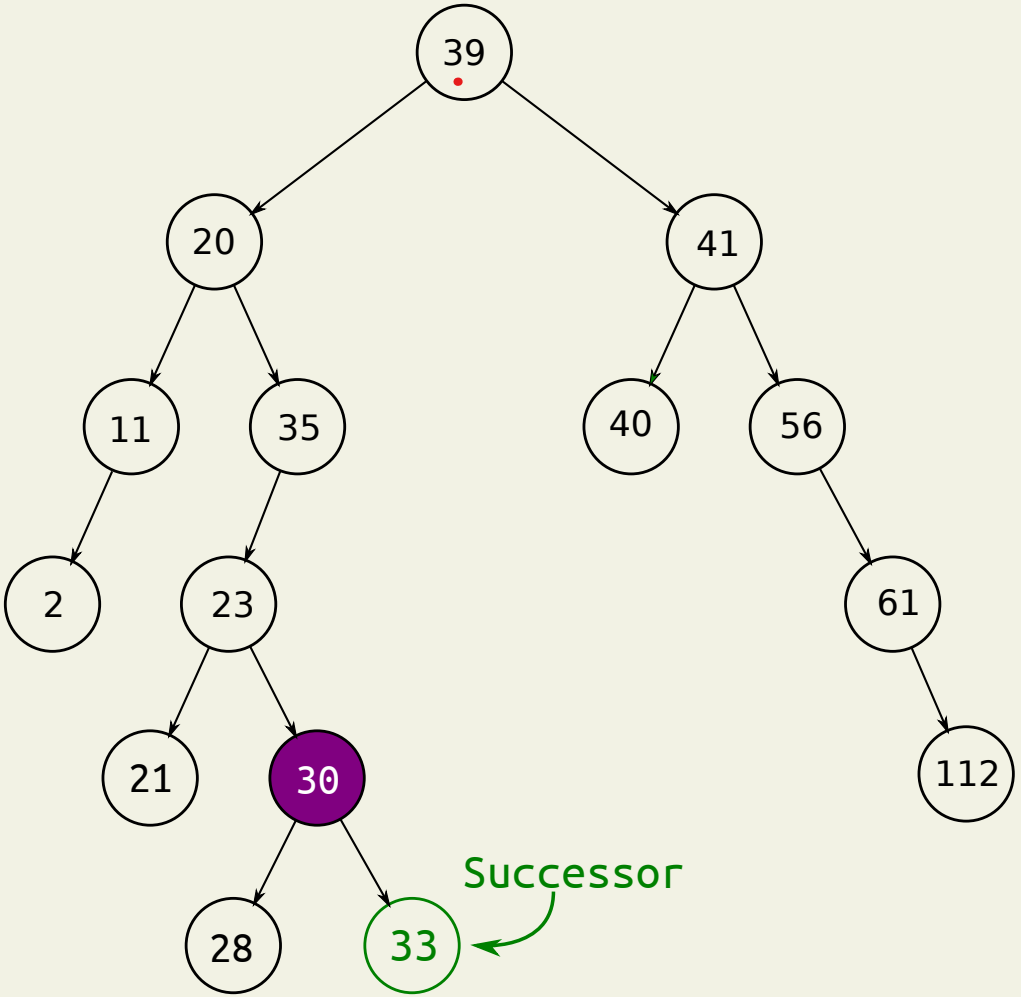


# Example

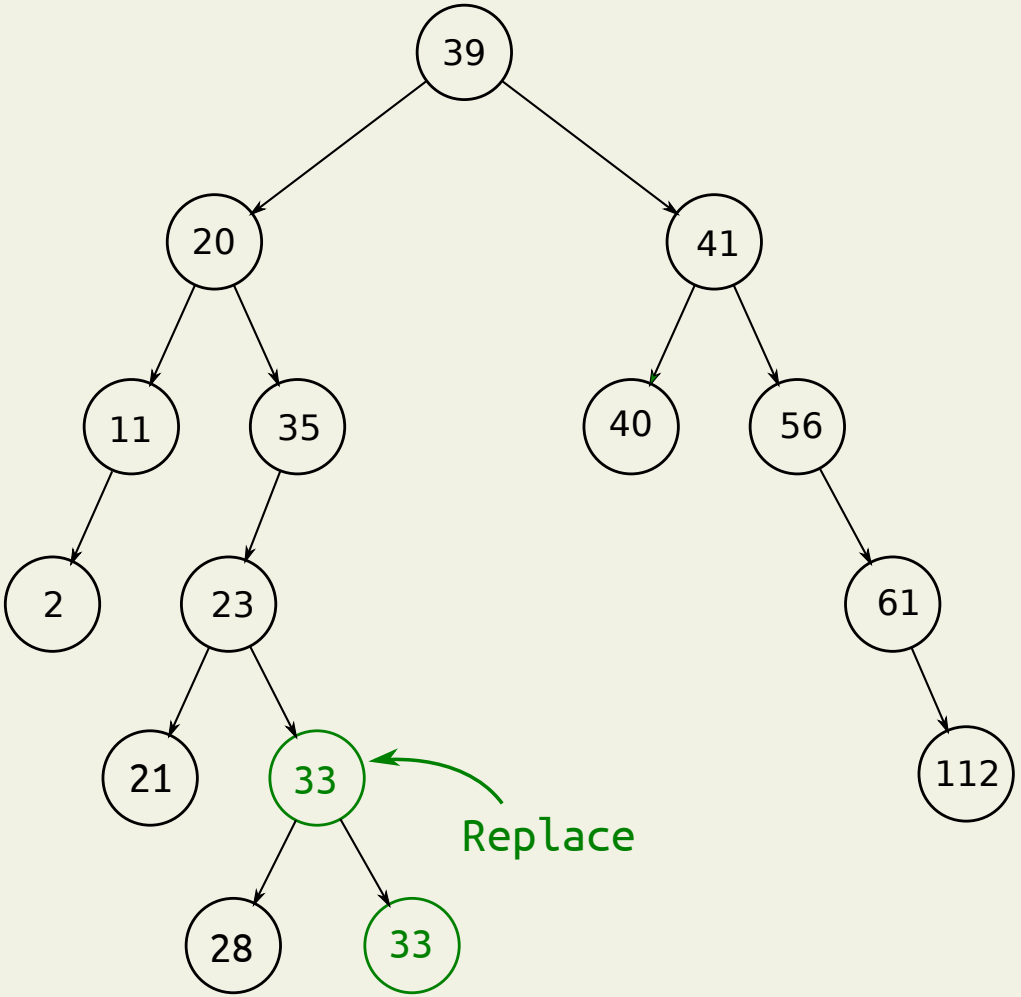


# Example

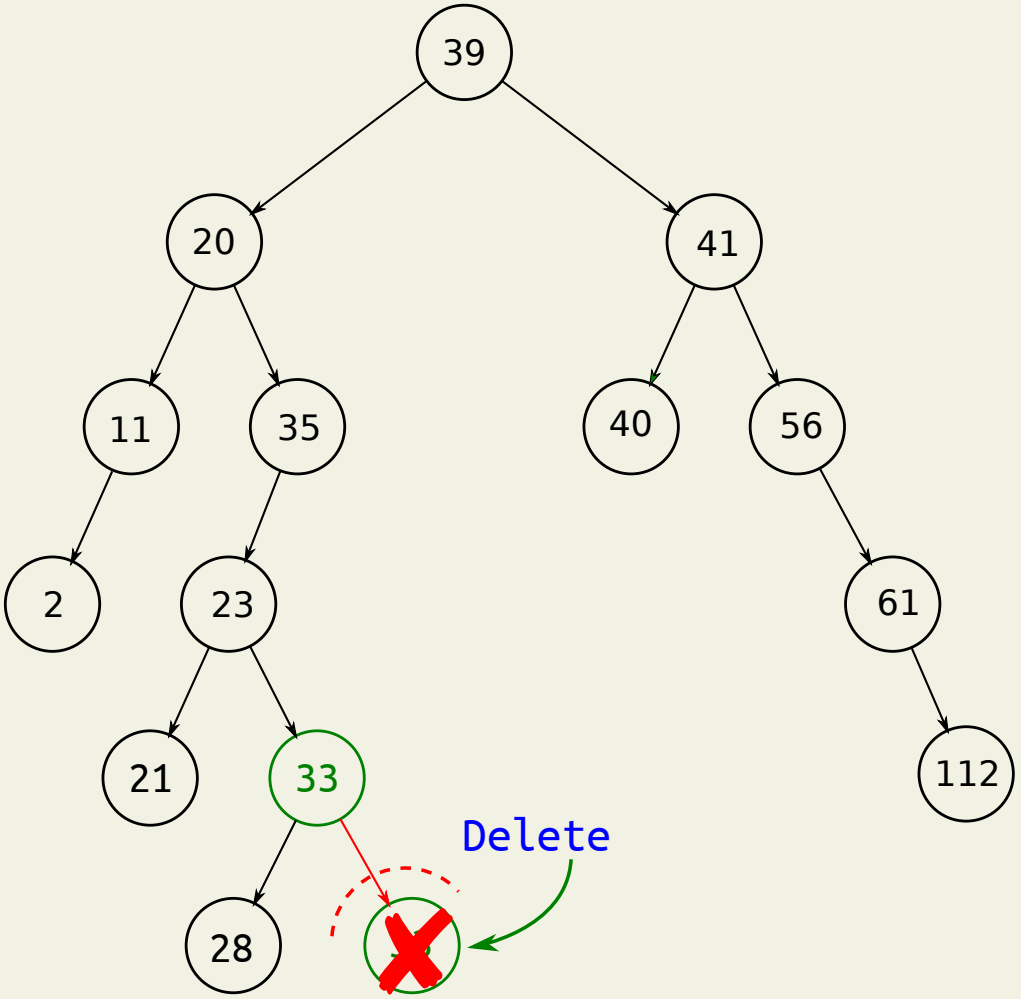
1, 2, 3, 4, 5, 18, 20, 26, ..



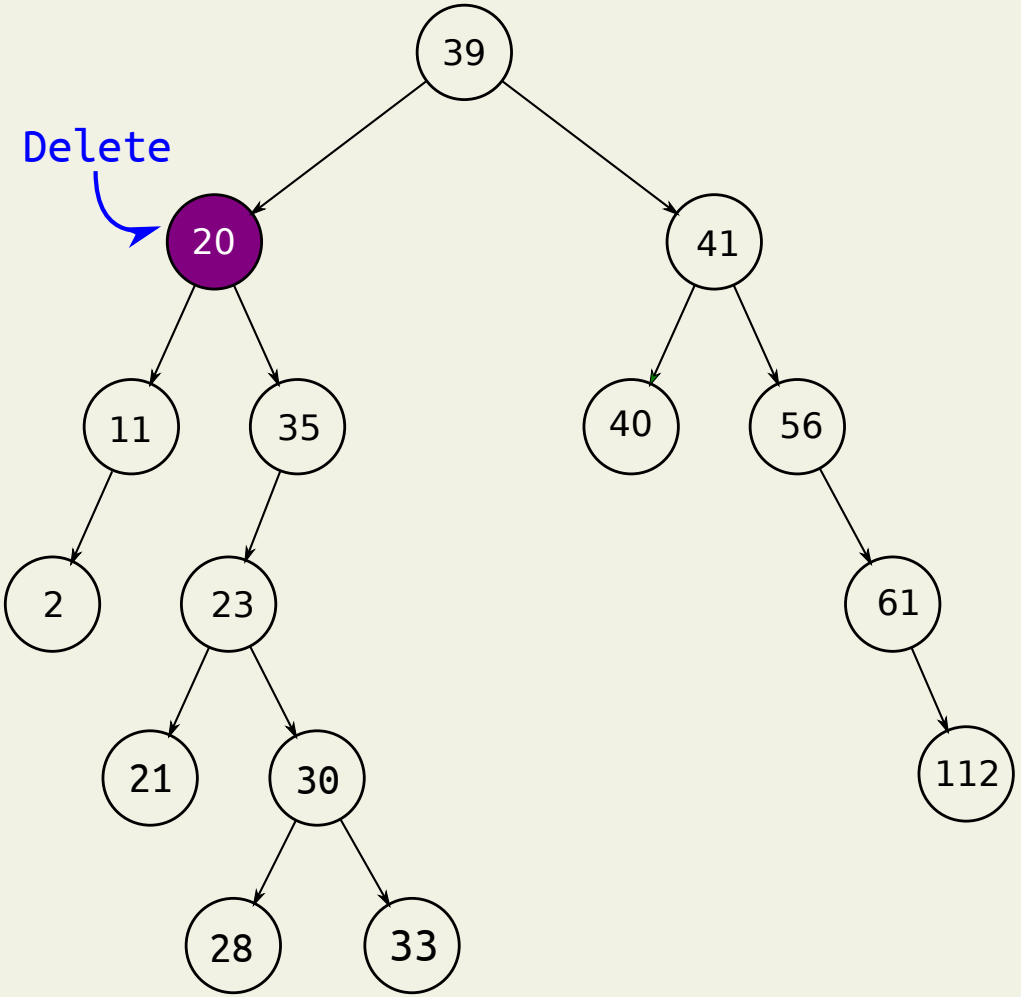
# Example



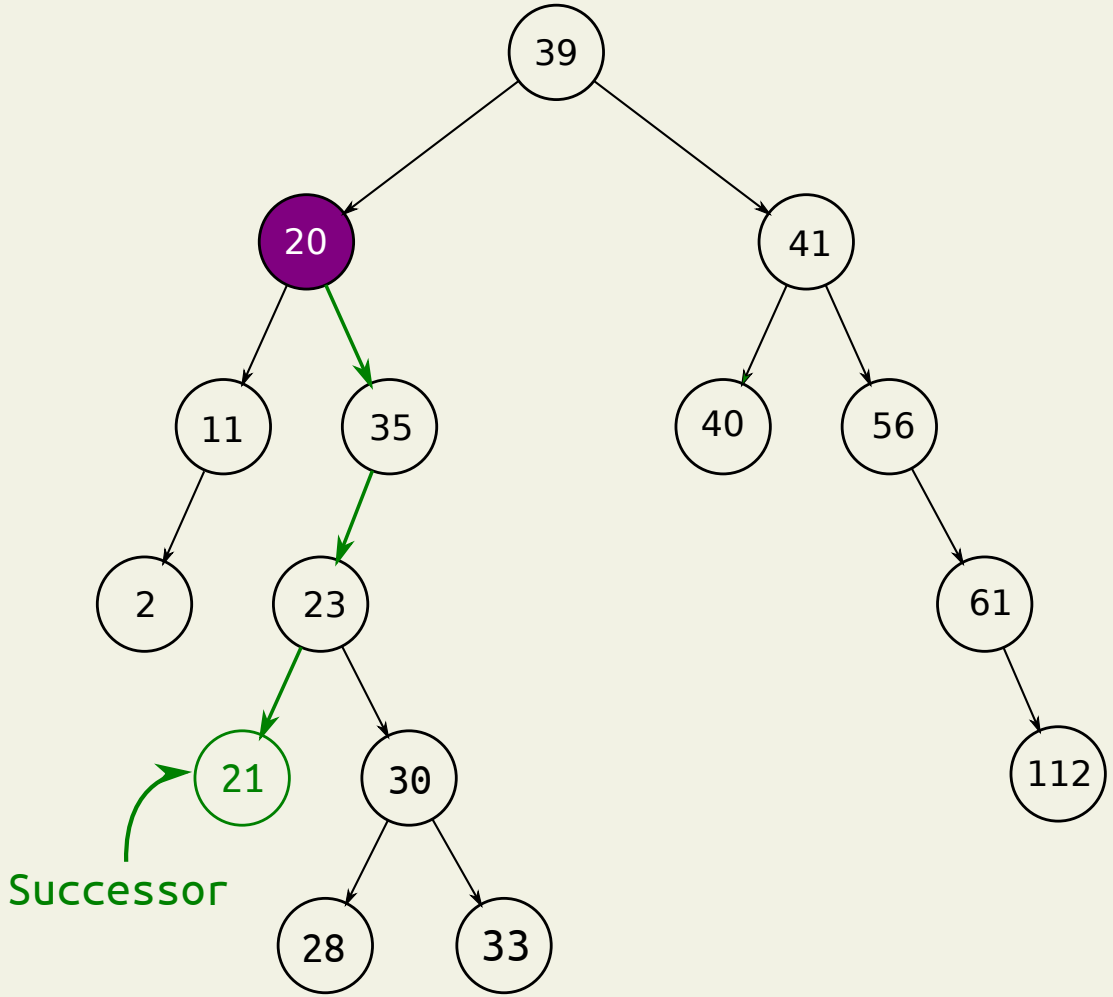
# Example



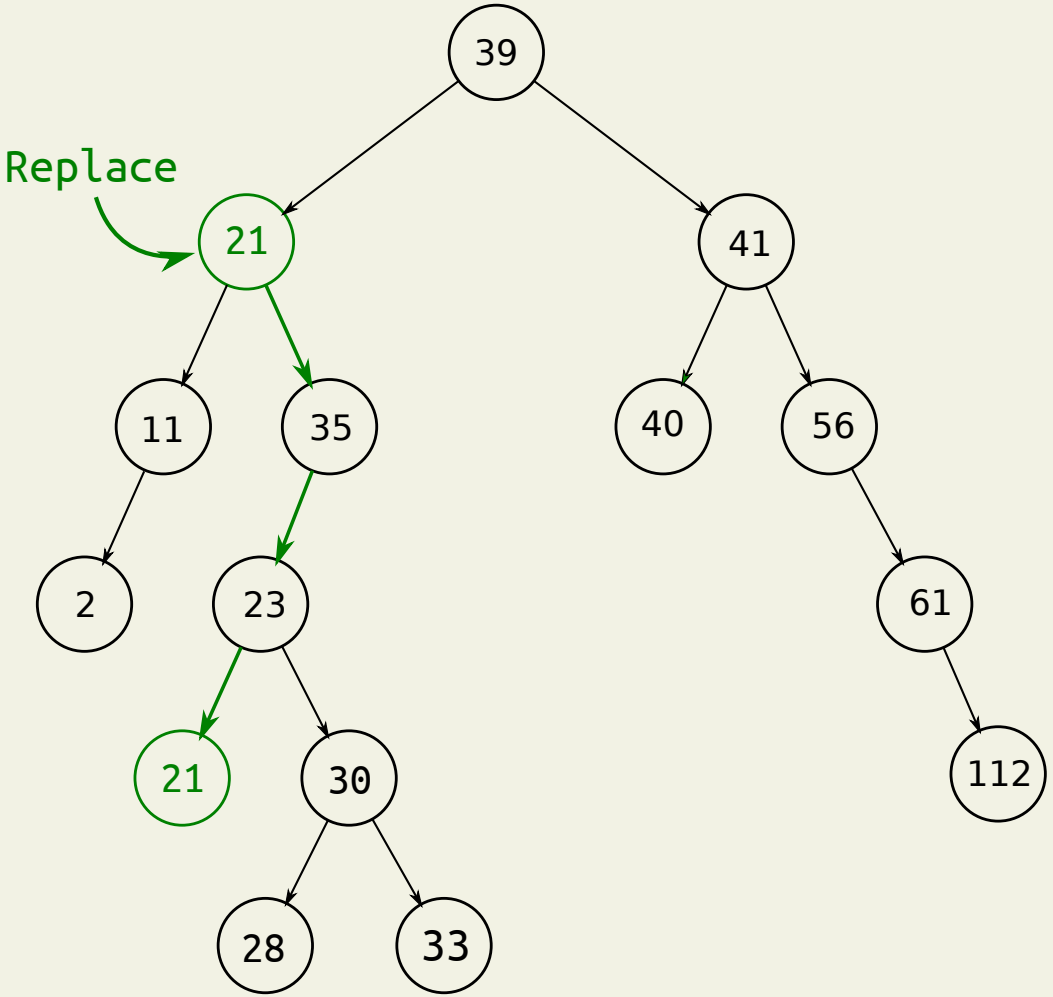
# Example



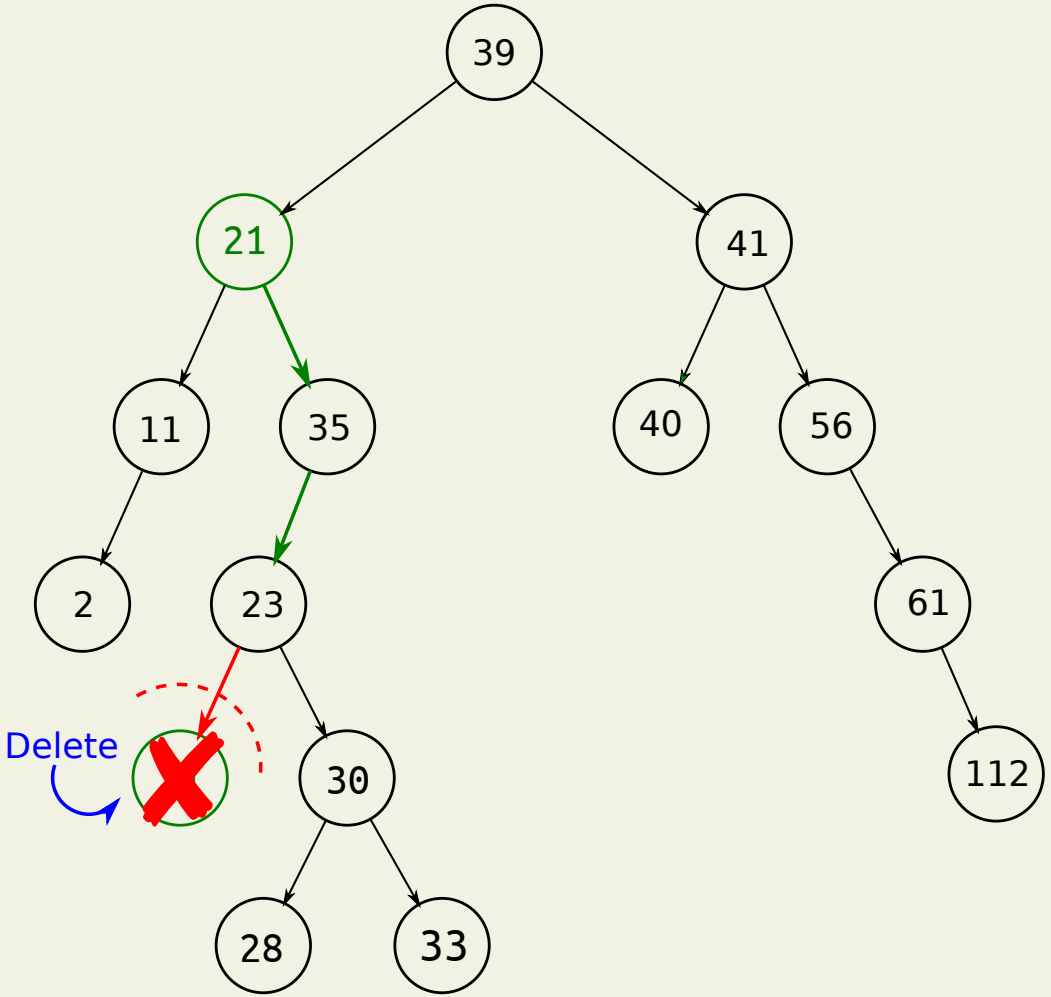
# Example



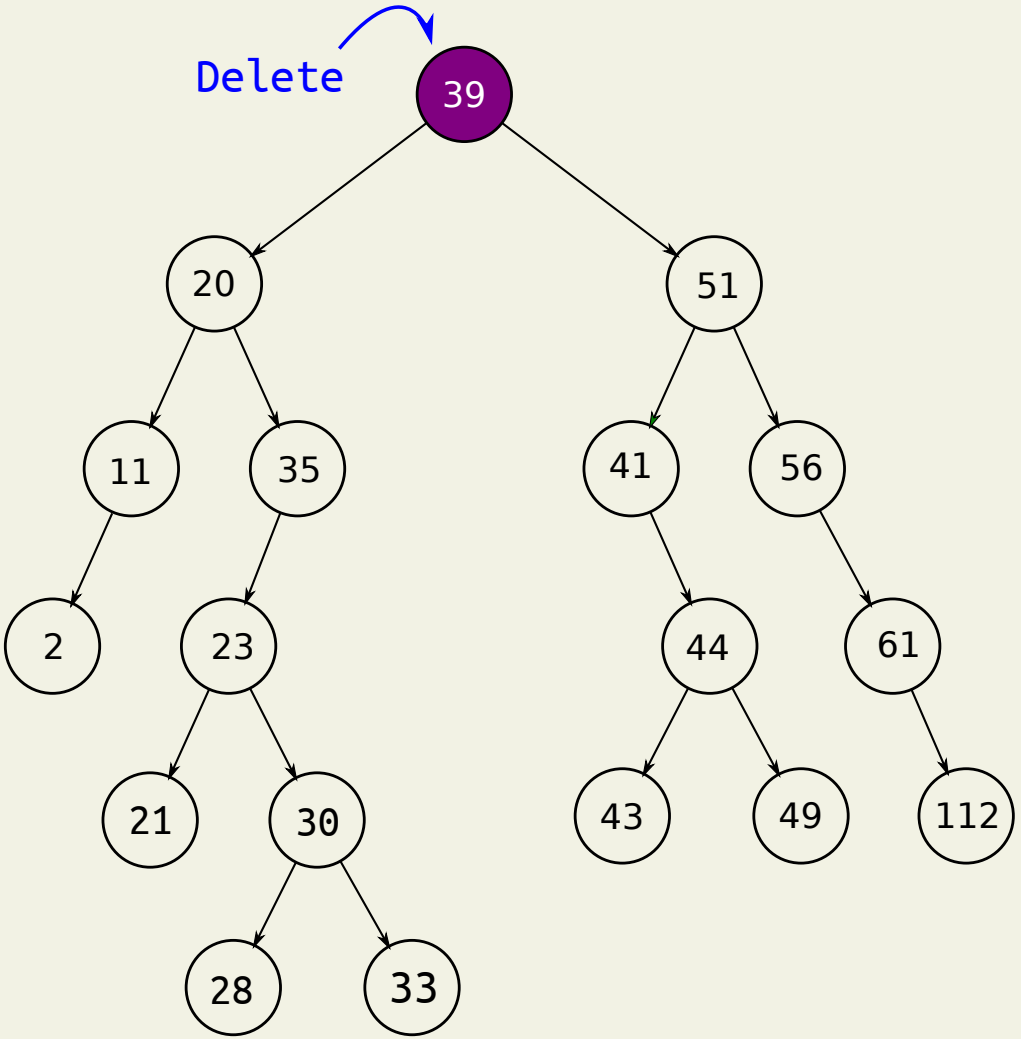
# Example



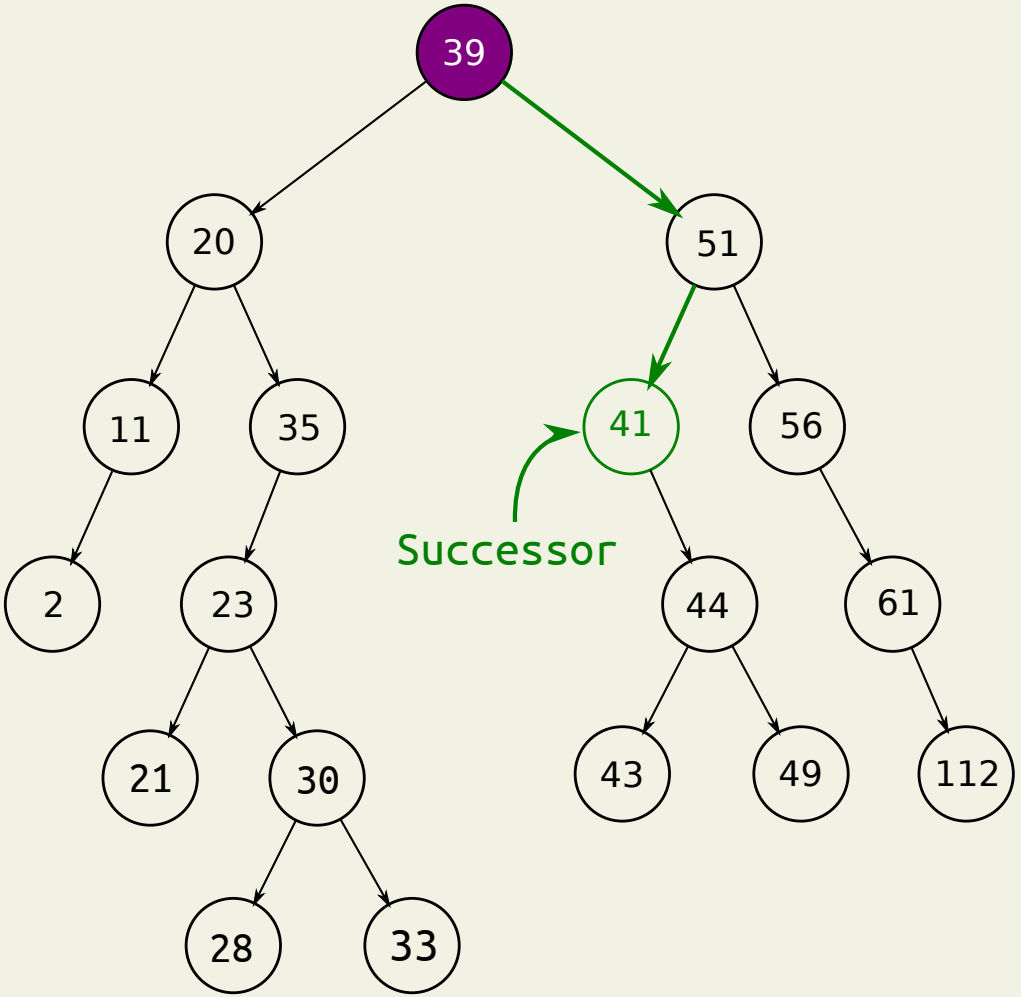
# Example



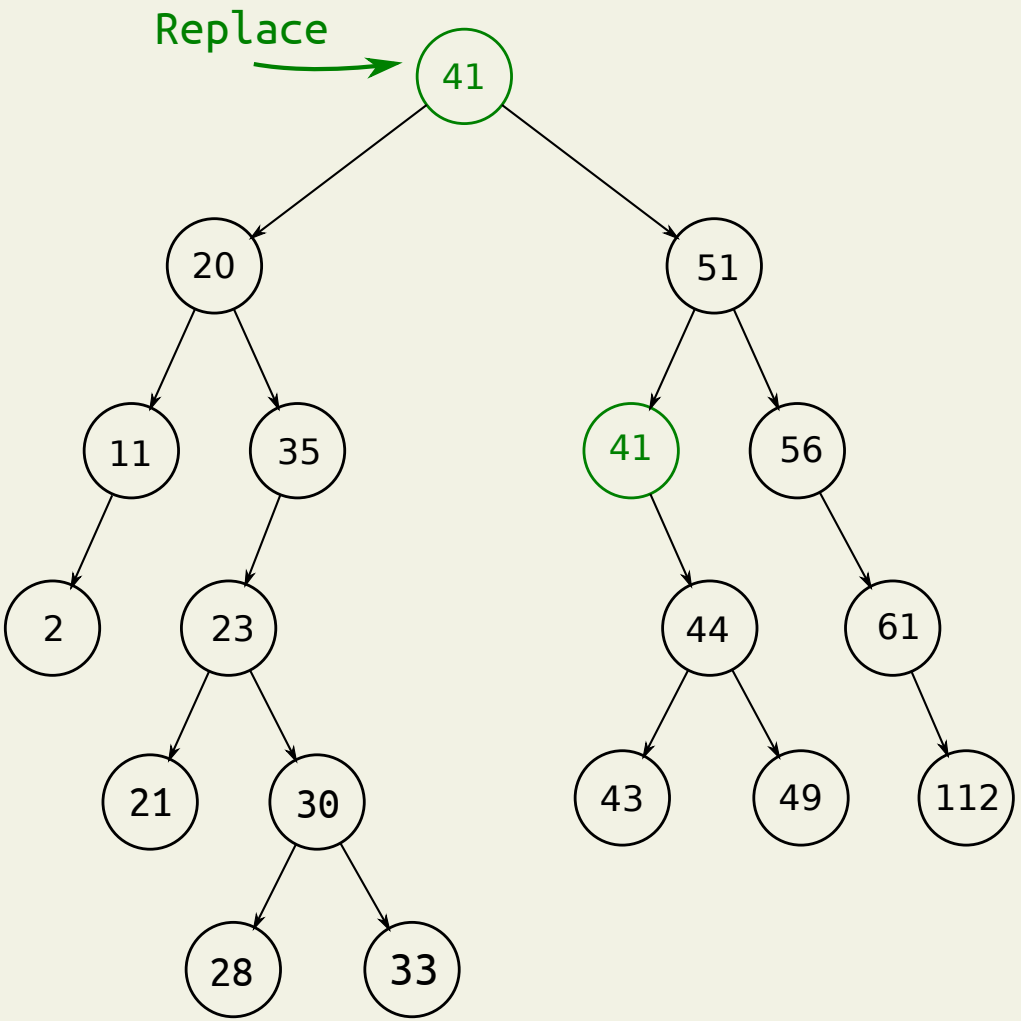
# Example



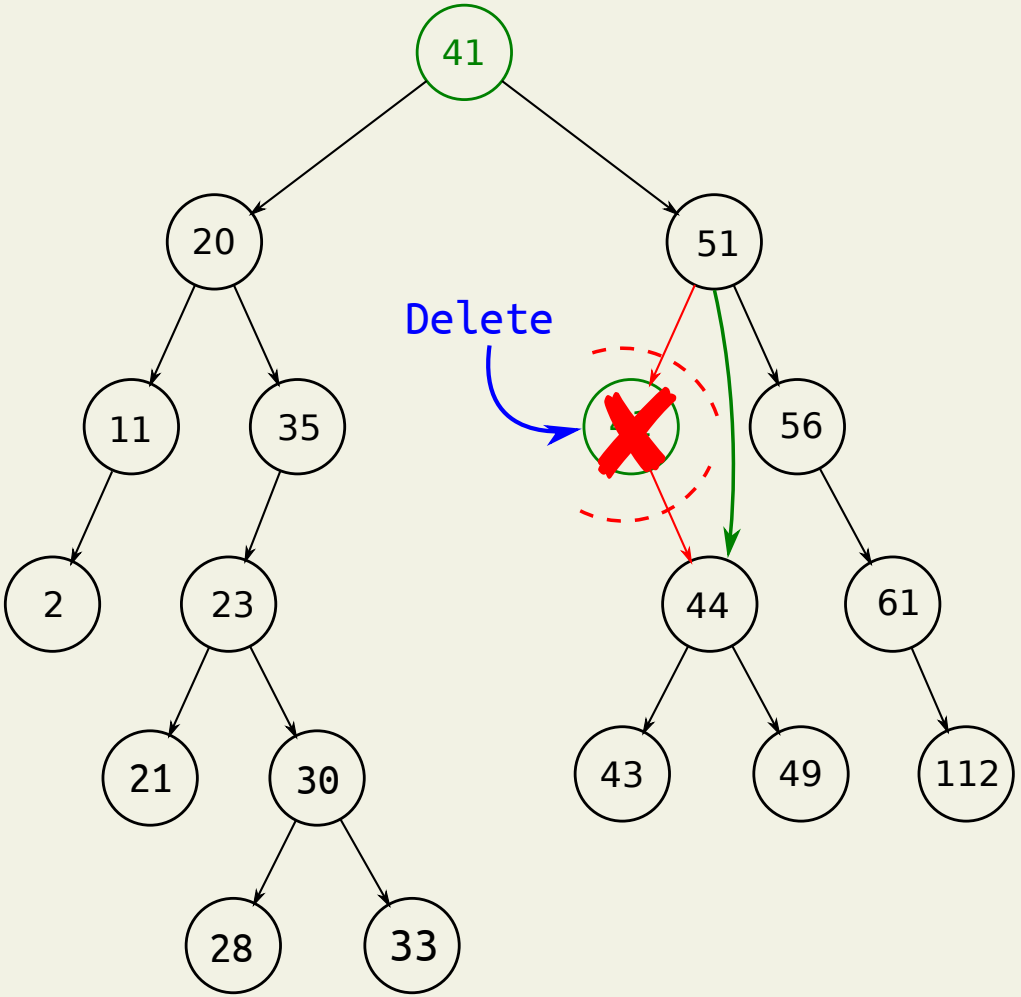
# Example



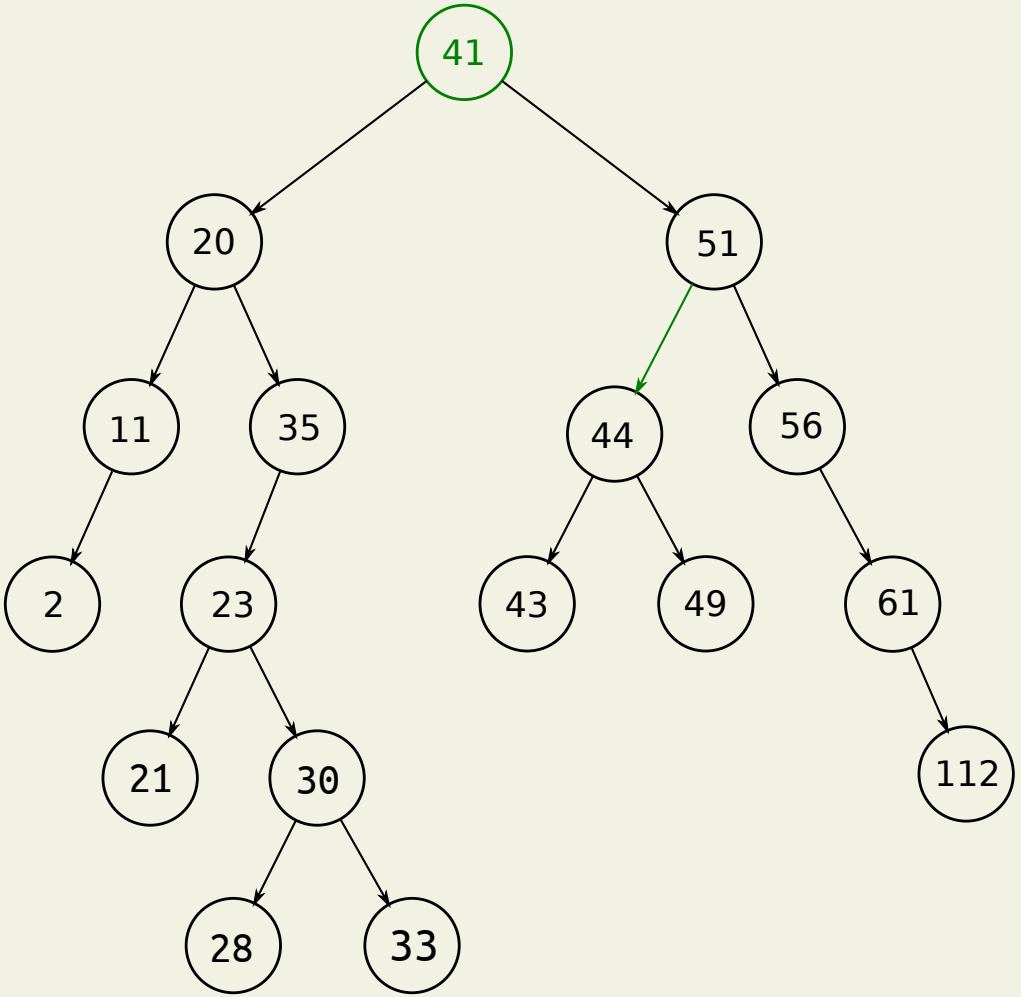
# Example



# Example



# Example



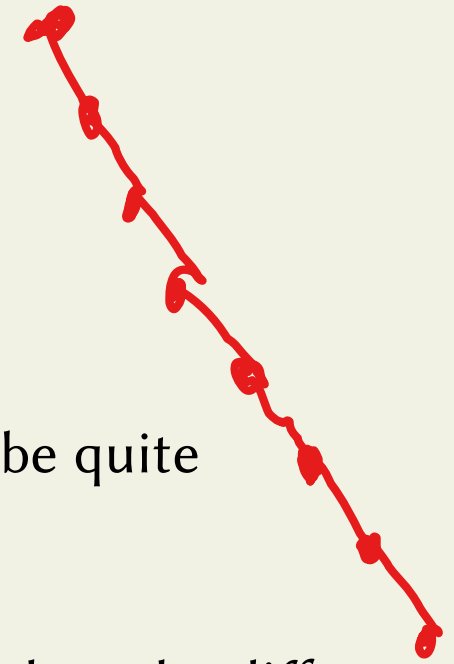
# Running Time

Worst case running times for a BST of height $h$:

- Insert – $O(h)$.
- Succ – $O(h)$.
- Search – $O(h)$.
- Delete – $O(h)$.

The height of a BST depends on the input sequence and can be $n$ after inserting $n$ elements in the worst case.

# Balancing a BST



The biggest drawback of BSTs are that they can be quite "unbalanced".

One way to measure if a tree is balanced is to look at the difference between the longest path from root to leaf and the shortest path from root to leaf.

# Balancing a BST

The biggest drawback of BSTs are that they can be quite "unbalanced".

One way to measure if a tree is balanced is to look at the difference between the longest path from root to leaf and the shortest path from root to leaf.

We want to make sure this difference does not get too large.

Thank You!