Unit 14: Binary Search Trees

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

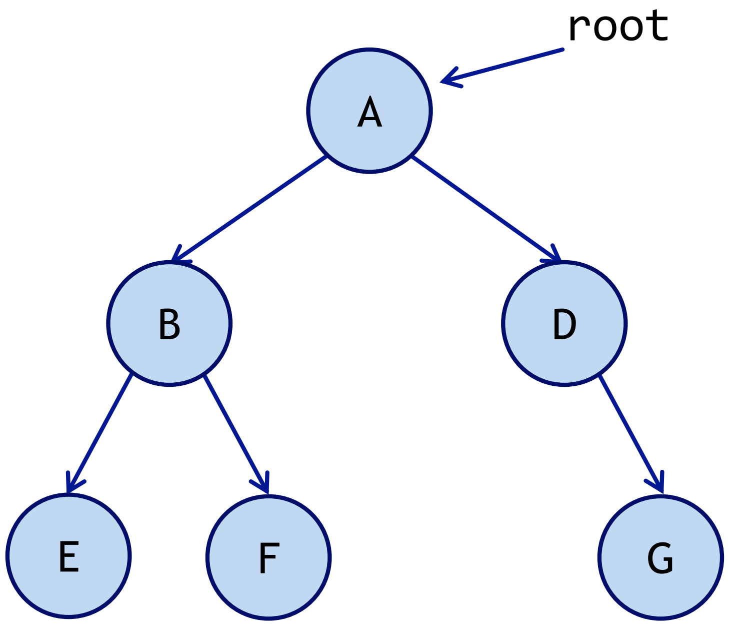
Unit 14 Overview

- Related Reading:
  - Textbook: Section 11.2
- Learning Objectives: (You should be able to…)
  - describe the binary **search** tree property
  - write code that inserts and removes elements from a binary search tree

Tree Terminology

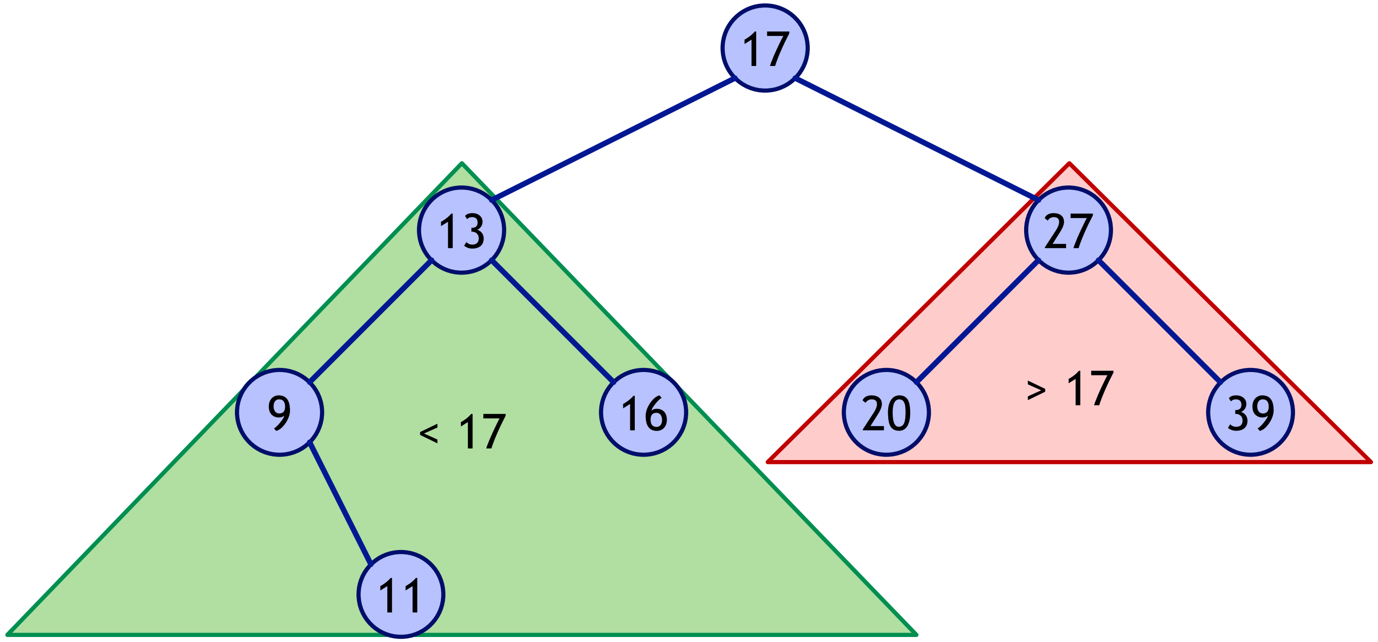
- A **binary tree** is a tree with at *most* two children per node
  - The children are typically referred to as *left* and *right*

B is A's **left** child



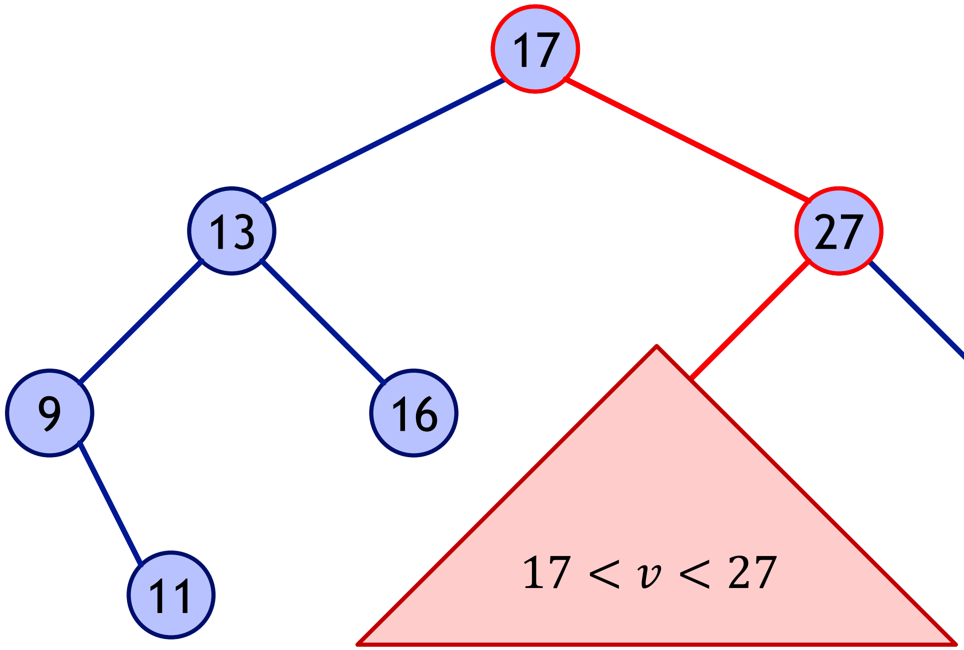
Binary Search Tree

- A **Binary *Search* Tree** (BST) is a binary tree with a special property:
- For every node n in the tree:
  - All node's in n's left subtree have keys less than n
  - All node's in n's right subtree have keys greater than n



Binary Search Tree

- A **Binary *Search* Tree** (BST) is a binary tree with a special property:
- For every node n in the tree:
  - All node's in n's left subtree have keys less than n
  - All node's in n's right subtree have keys greater than n



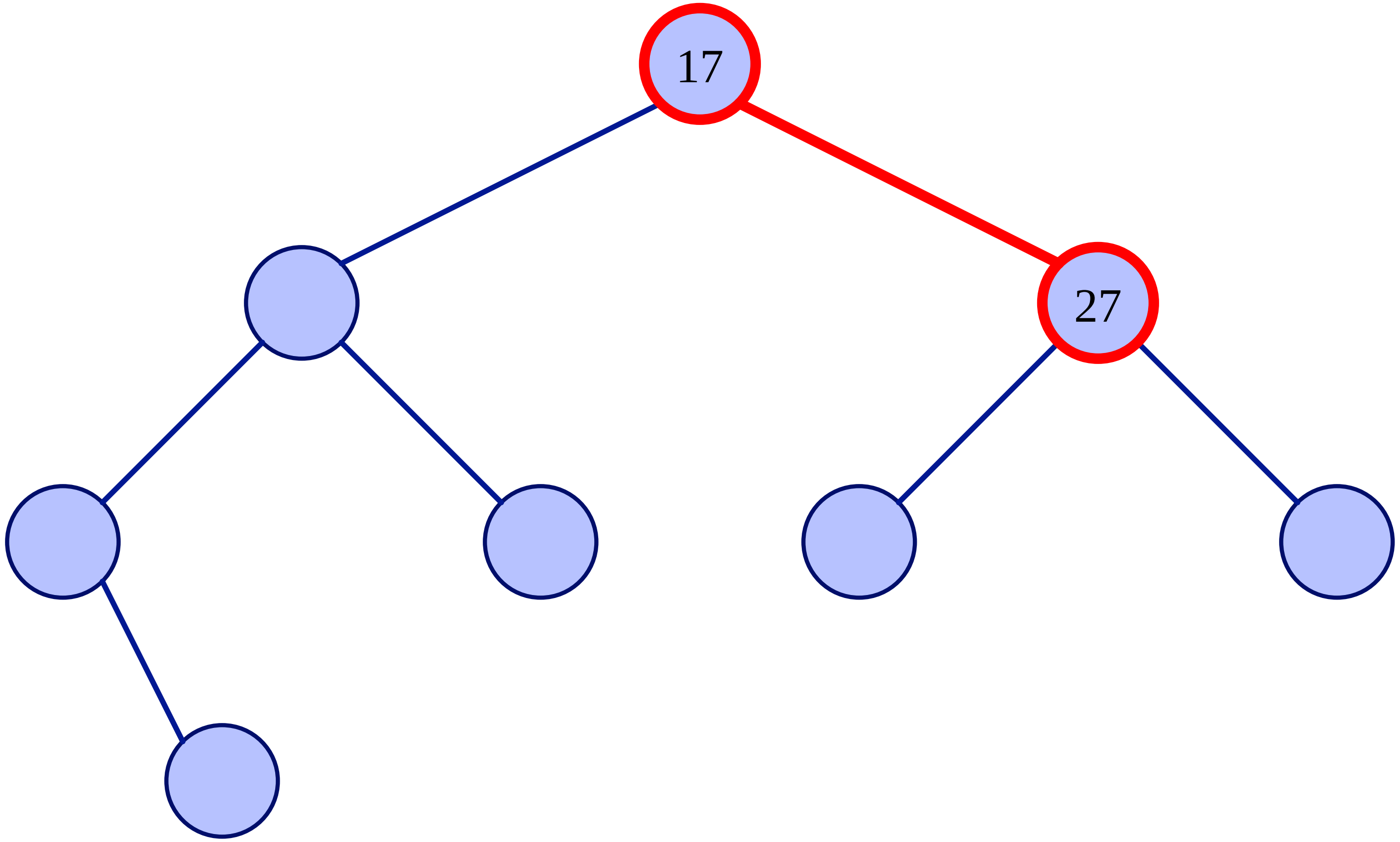
BST Search

- To find an element in a BST search beginning at node n:
  - If the target **key** is less than n's **key**, then search n's left subtree
  - If the target **key** is greater than n's **key**, then search n's right subtree
  - if n's key is equal to the target value, return **true** (or a pointer to the data)

- How many comparisons?
  - One for each node on the path
  - Worst case: height of the tree + 1

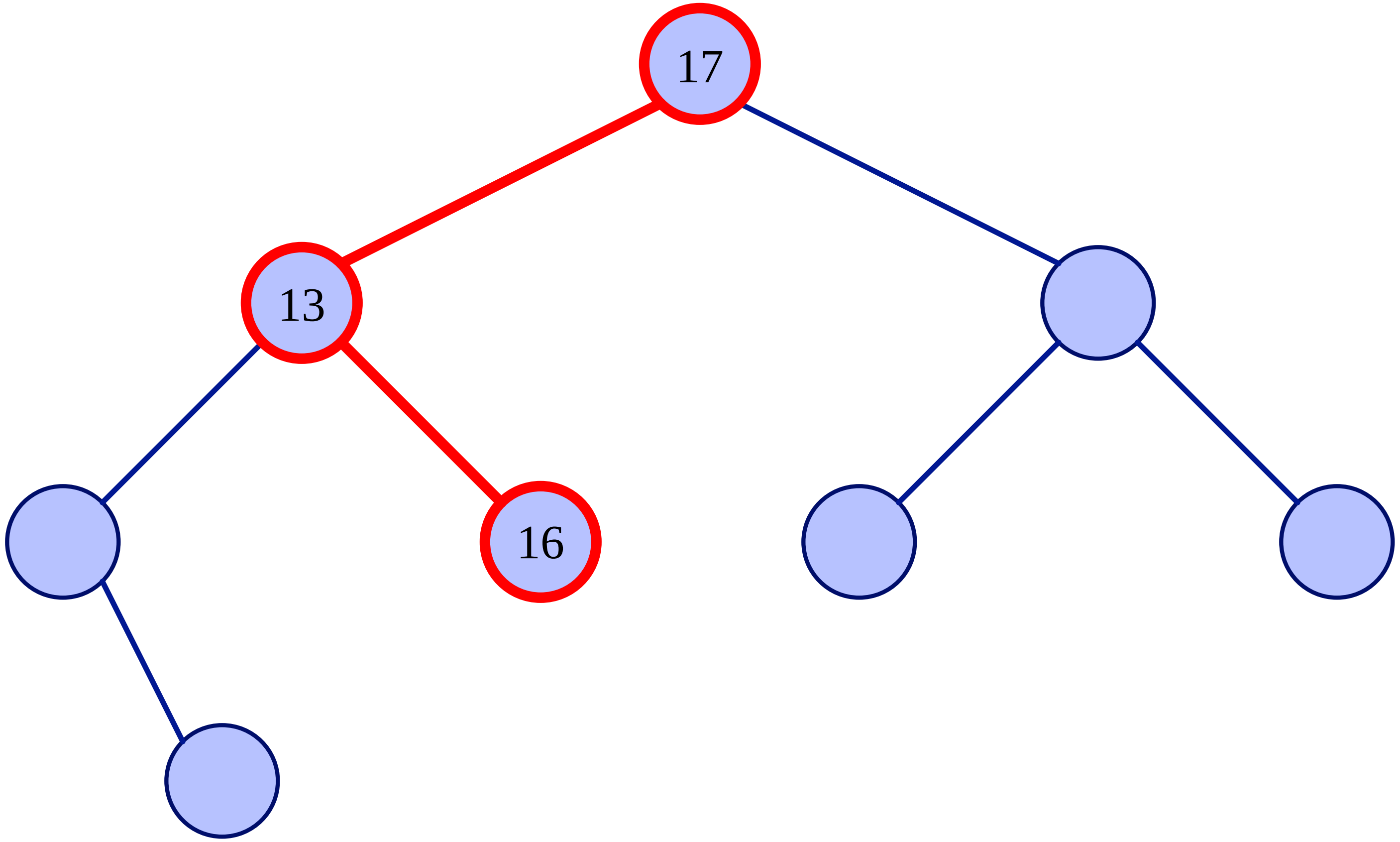
BST Search Example

```
search(27);
```



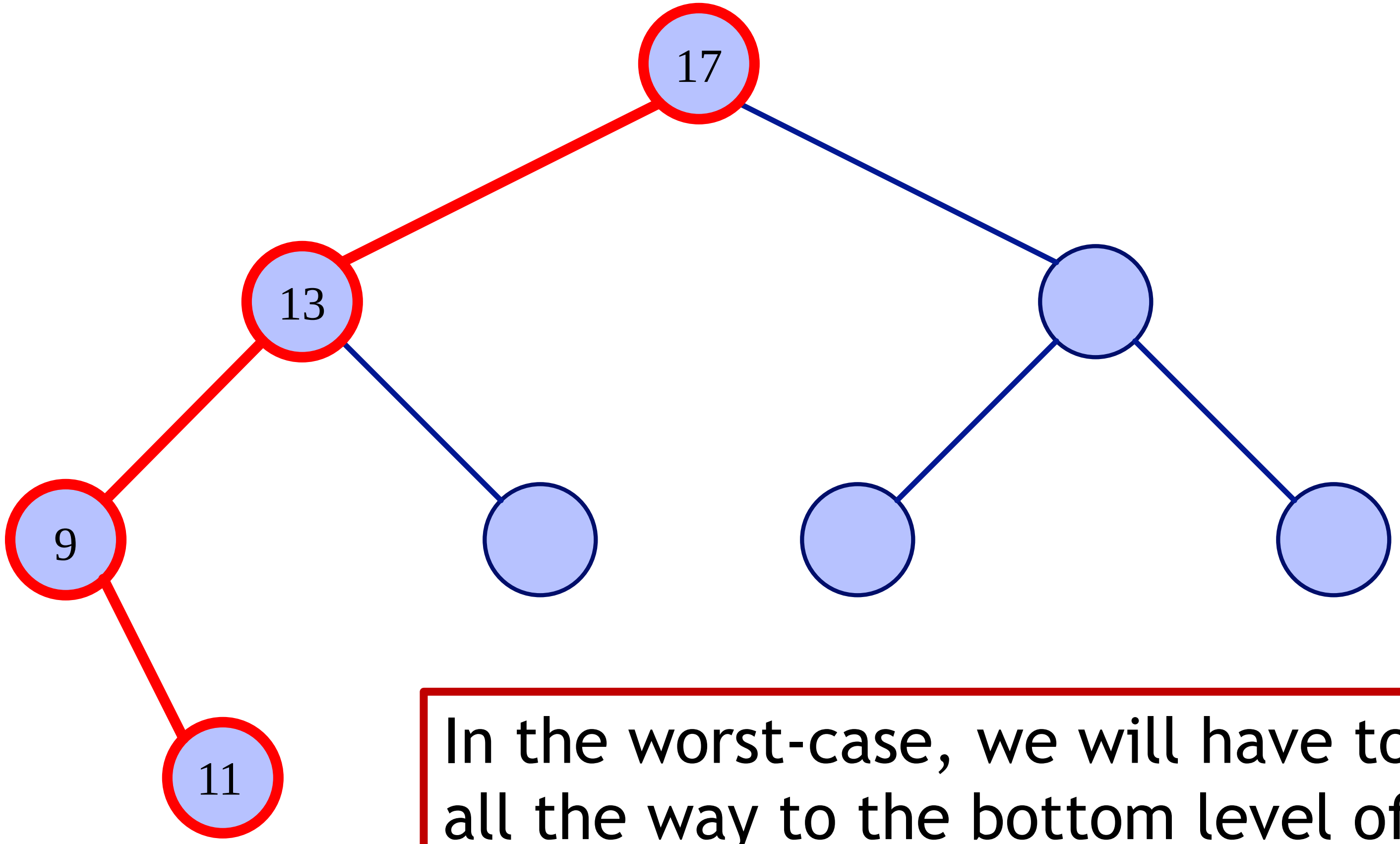
BST Search Example

```
search(16);
```



BST Search Example

```
search(12);
```



In the worst-case, we will have to search all the way to the bottom level of the tree

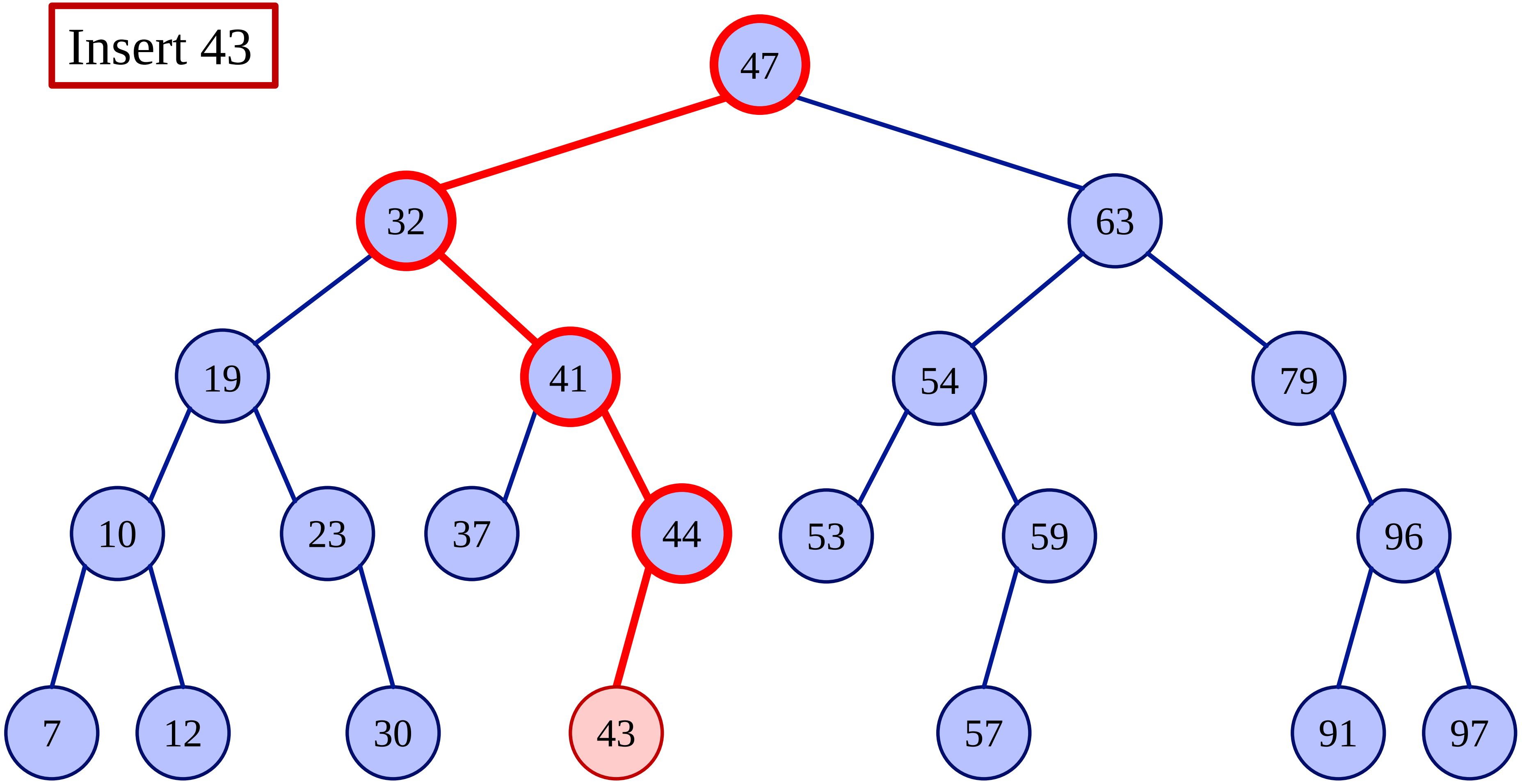
BST Insertion

- The BST property must hold after each insertion
  - Observation from search: there is only one valid place to insert a new node

- Make sure the node is inserted in the correct position
  - The position is determined by performing a search
  - If the search ends at the null left child of a node n, insert the new node so that it is n's left child
  - If the search ends at the null right child of a node n, insert the new node so that it is n's right child

- The runtime is also bound by the tree's height: $O(height)$

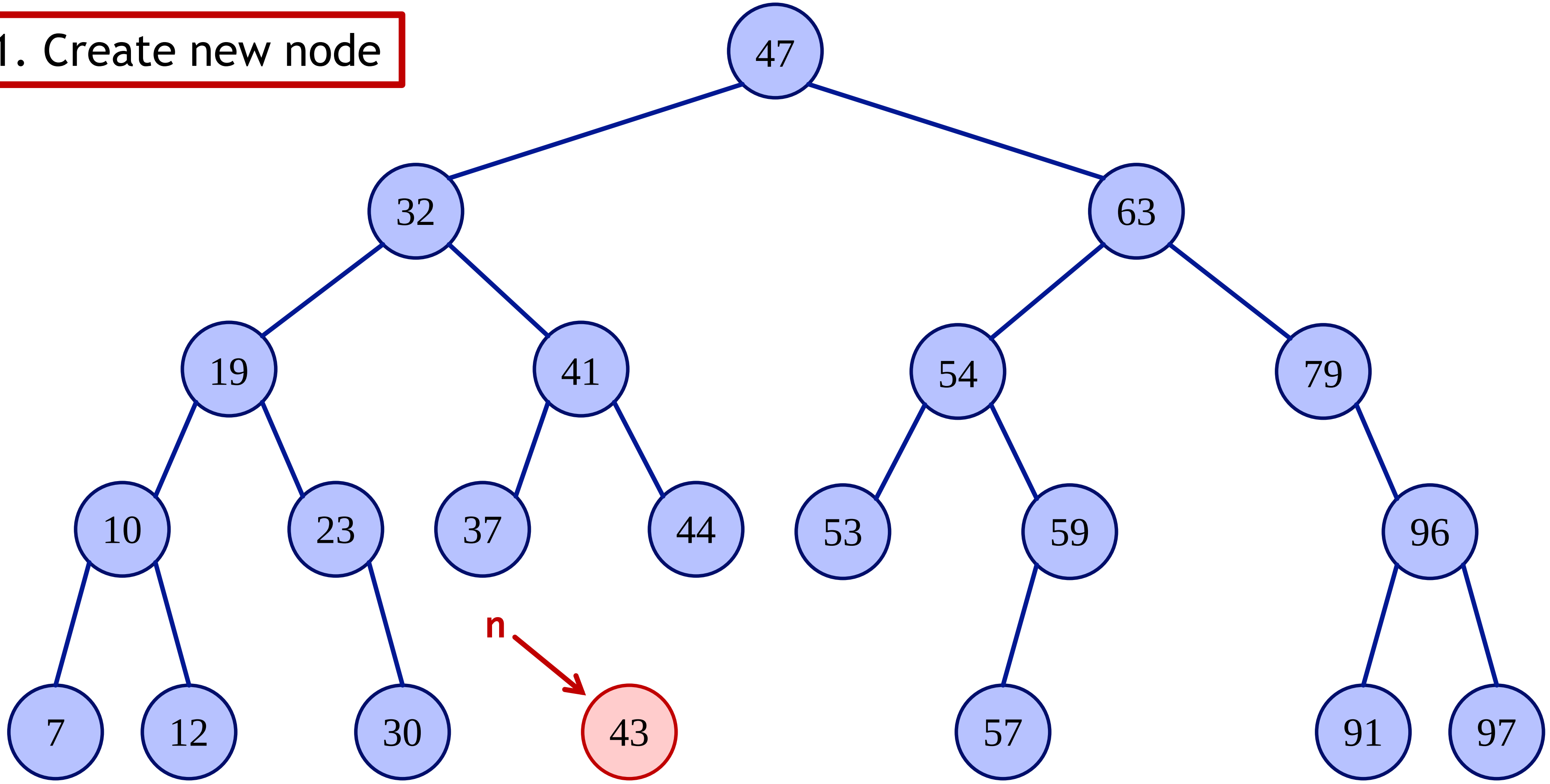
BST Insertion Example



BST Insertion Example

Insert 43

1. Create new node



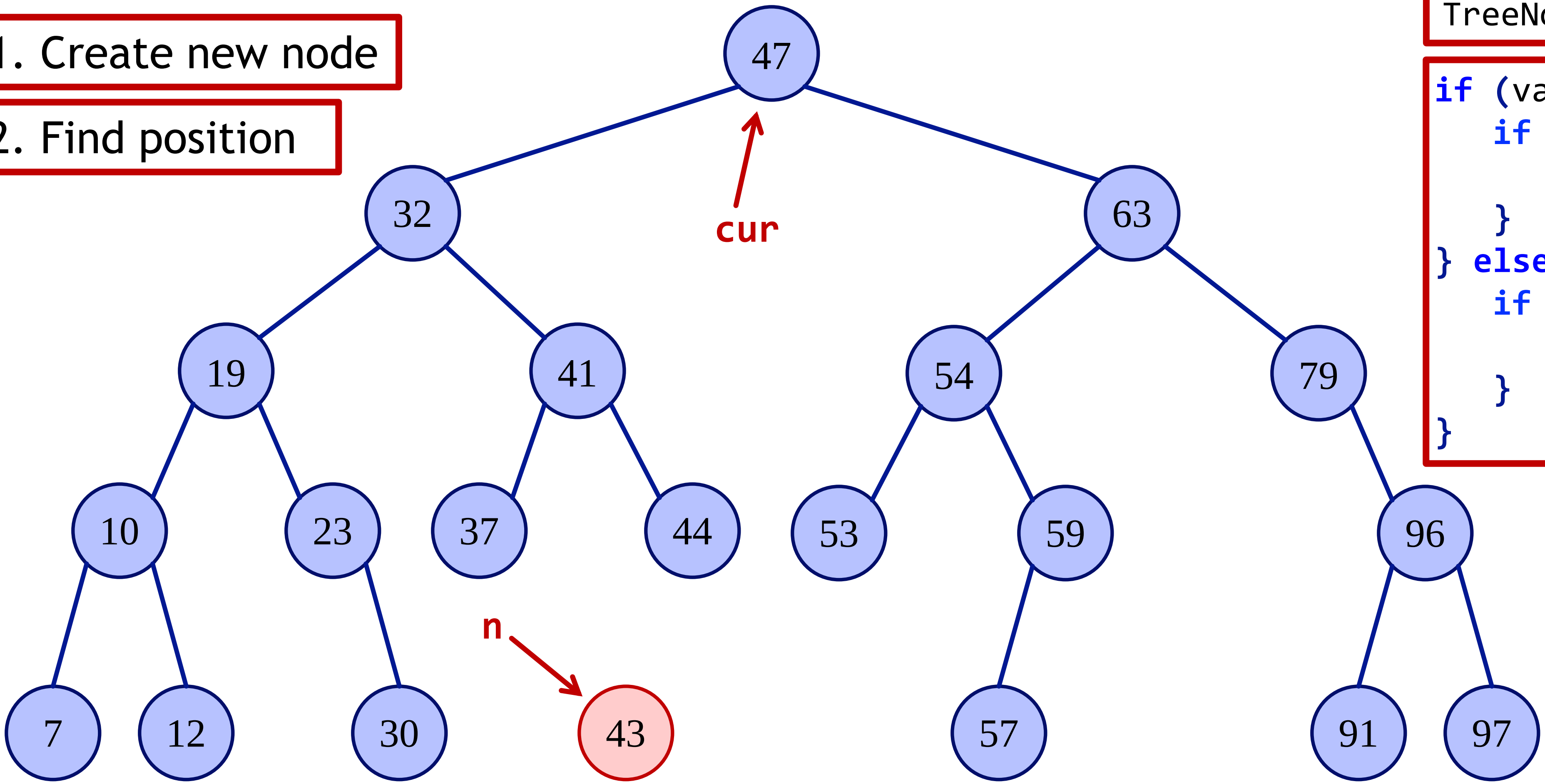

```
TreeNode n = new TreeNode(val);
```

BST Insertion Example

Insert 43

1. Create new node

2. Find position




```
TreeNode cur = root;
```

```
if (val < cur.data) {
   if (cur.left != null) {
      cur = cur.left;
   }
} else if (val > cur.data) {
   if (cur.right != null) {
      cur = cur.right;
   }
}
```

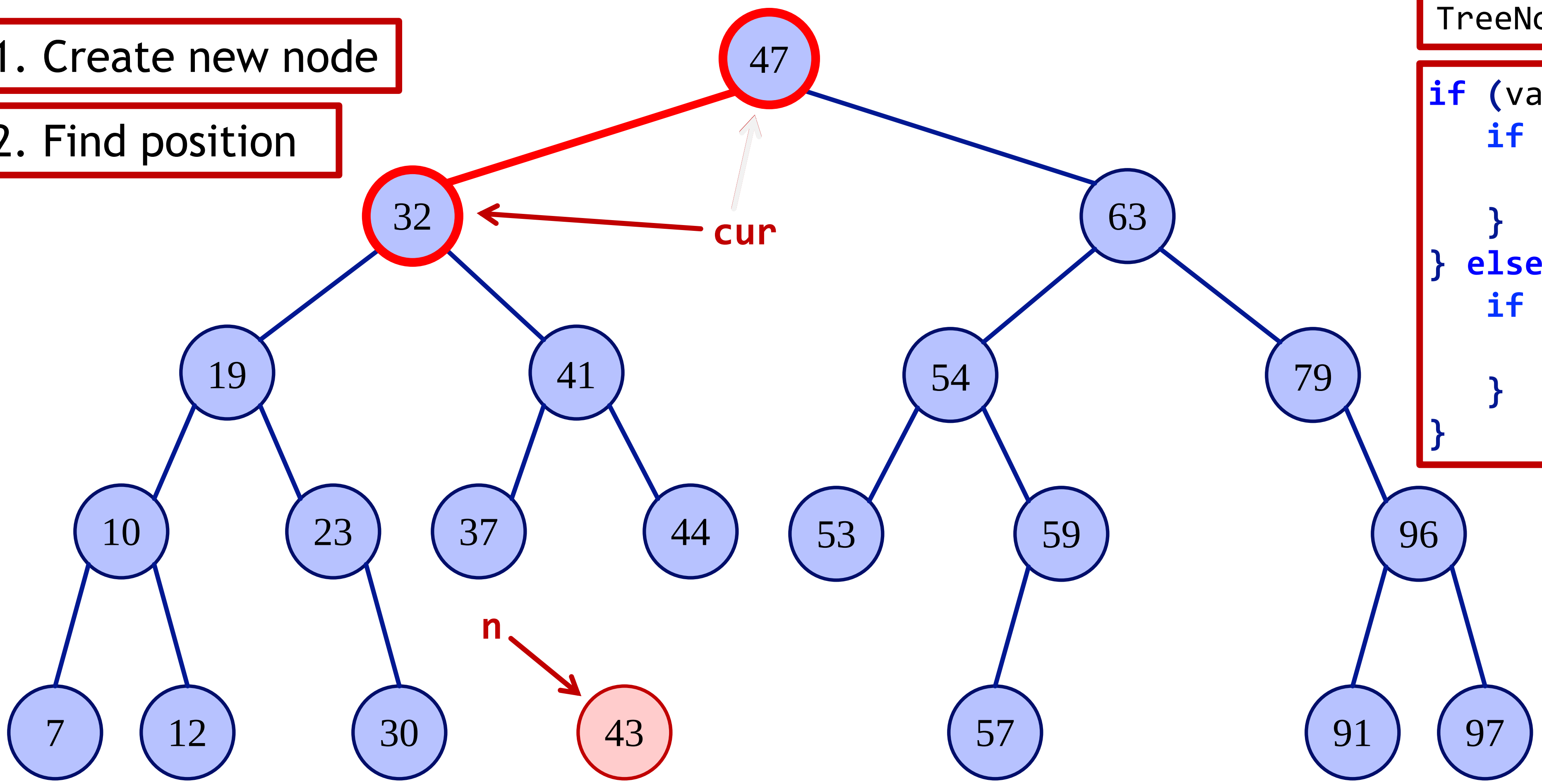
```
TreeNode n = new TreeNode(val);
```

BST Insertion Example

Insert 43

1. Create new node

2. Find position




```
TreeNode cur = root;
```

```
if (val < cur.data) {
   if (cur.left != null) {
      cur = cur.left;
   }
} else if (val > cur.data) {
   if (cur.right != null) {
      cur = cur.right;
   }
}
```

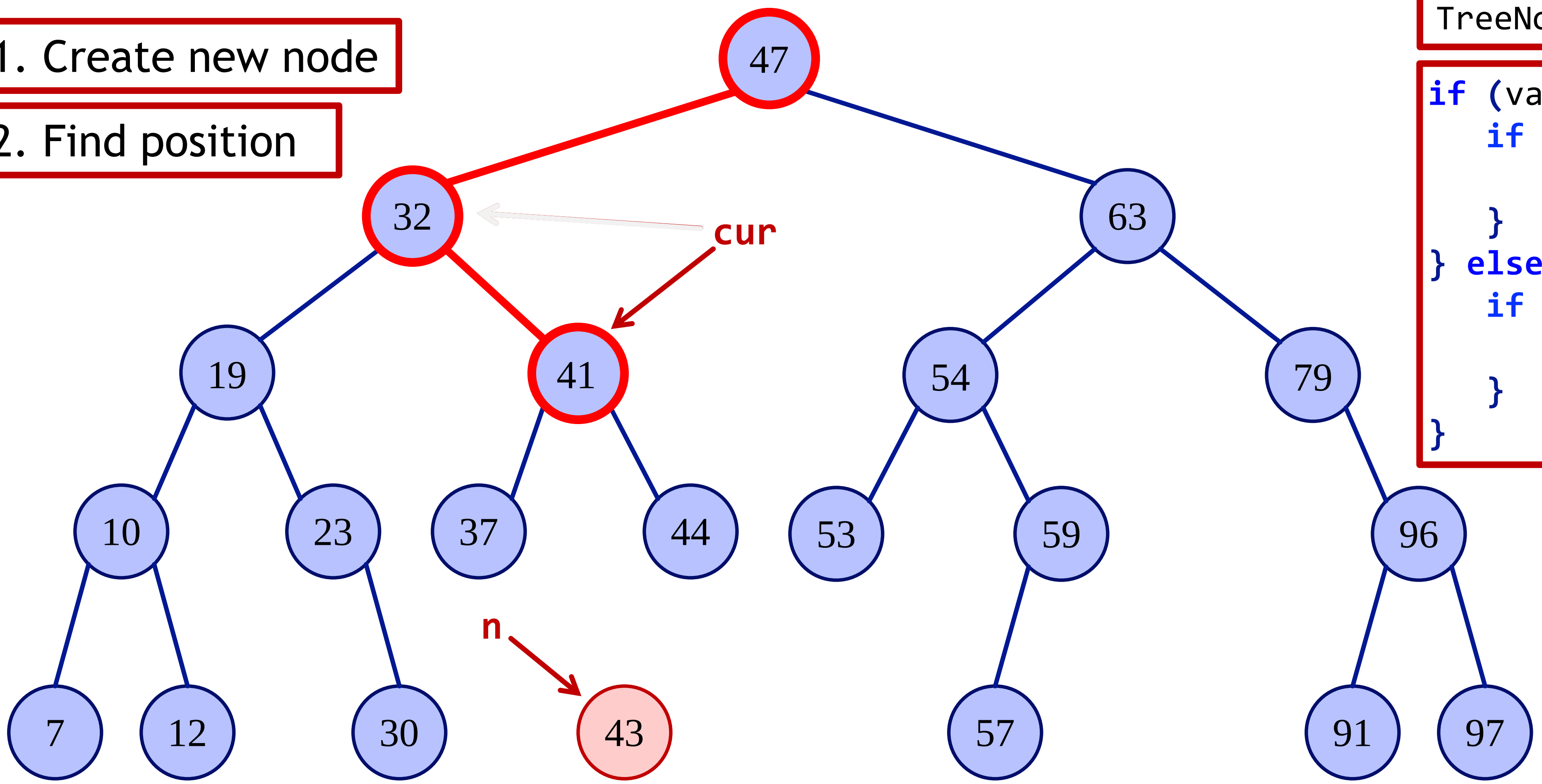
```
TreeNode n = new TreeNode(val);
```

BST Insertion Example

Insert 43

1. Create new node

2. Find position




```
TreeNode cur = root;
```

```
if (val < cur.data) {
   if (cur.left != null) {
      cur = cur.left;
   }
} else if (val > cur.data) {
   if (cur.right != null) {
      cur = cur.right;
   }
}
```

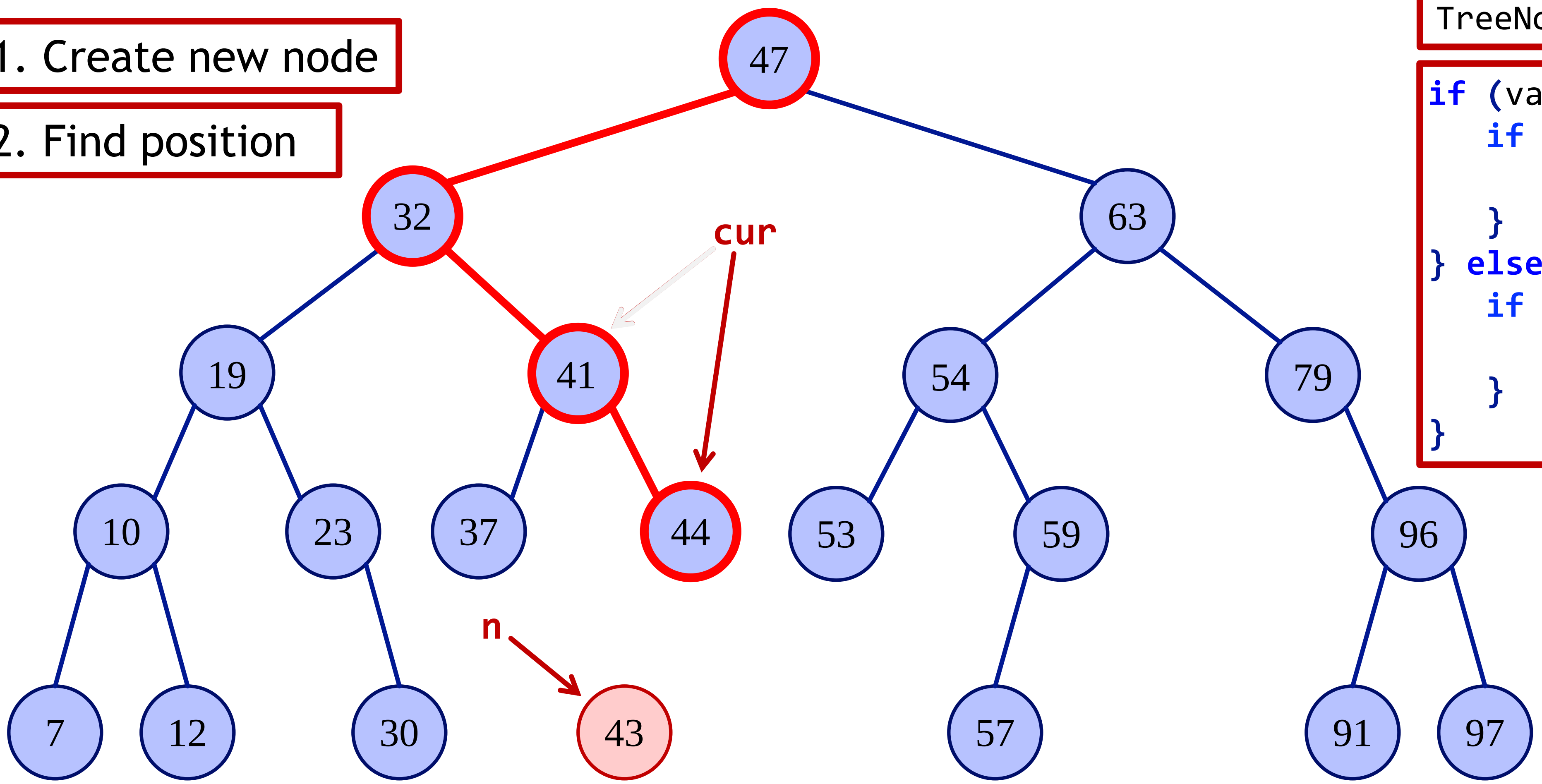
```
TreeNode n = new TreeNode(val);
```

BST Insertion Example

Insert 43

1. Create new node

2. Find position




```
TreeNode cur = root;
```

```
if (val < cur.data) {
   if (cur.left != null) {
      cur = cur.left;
   }
} else if (val > cur.data) {
   if (cur.right != null) {
      cur = cur.right;
   }
}
```

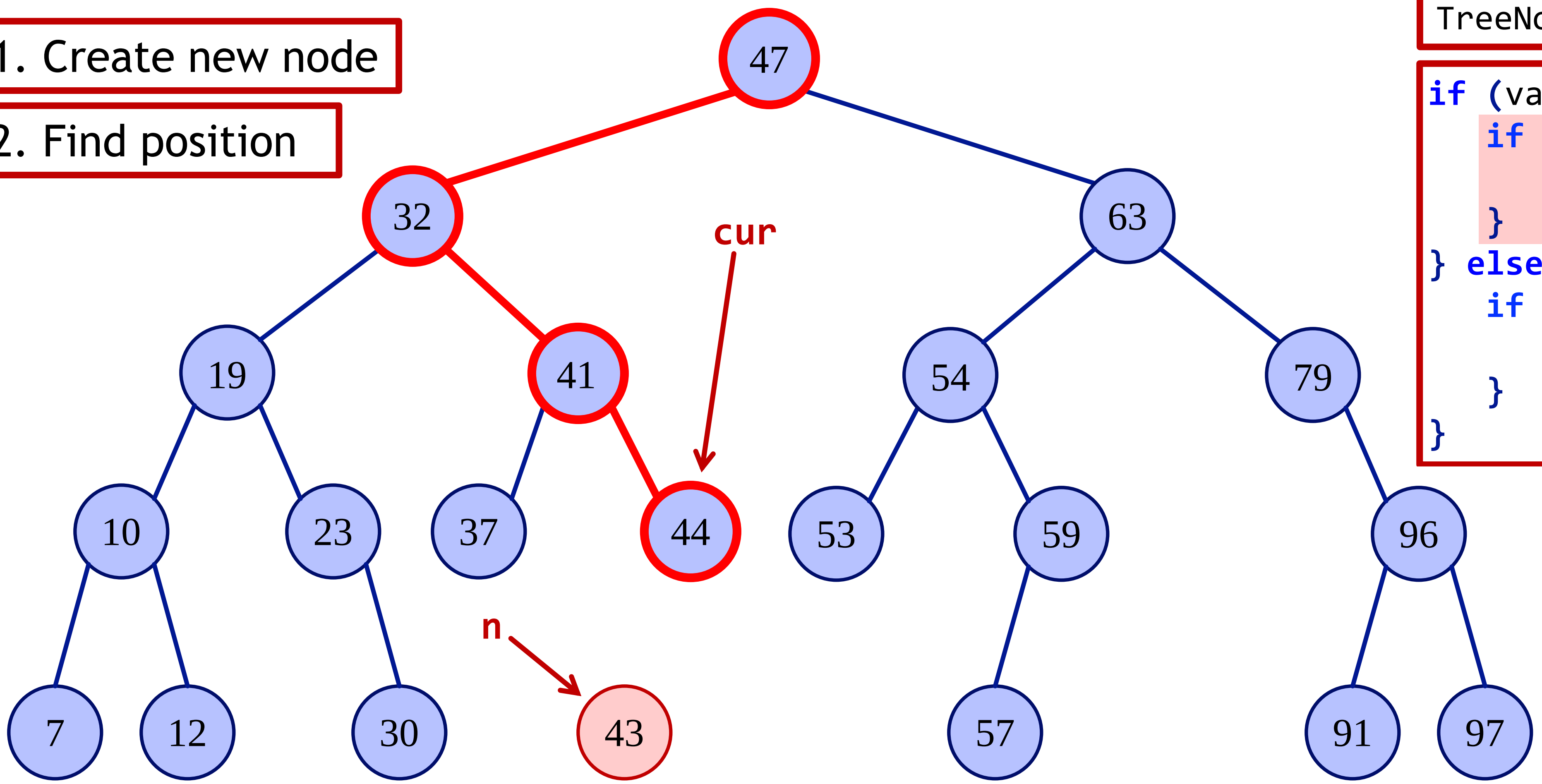
```
TreeNode n = new TreeNode(val);
```

BST Insertion Example

Insert 43

1. Create new node

2. Find position




```
TreeNode cur = root;
```

```
if (val < cur.data) {
   if (cur.left != null) {
      cur = cur.left;
   }
} else if (val > cur.data) {
   if (cur.right != null) {
      cur = cur.right;
   }
}
```

```
TreeNode n = new TreeNode(val);
```

BST Insertion Example

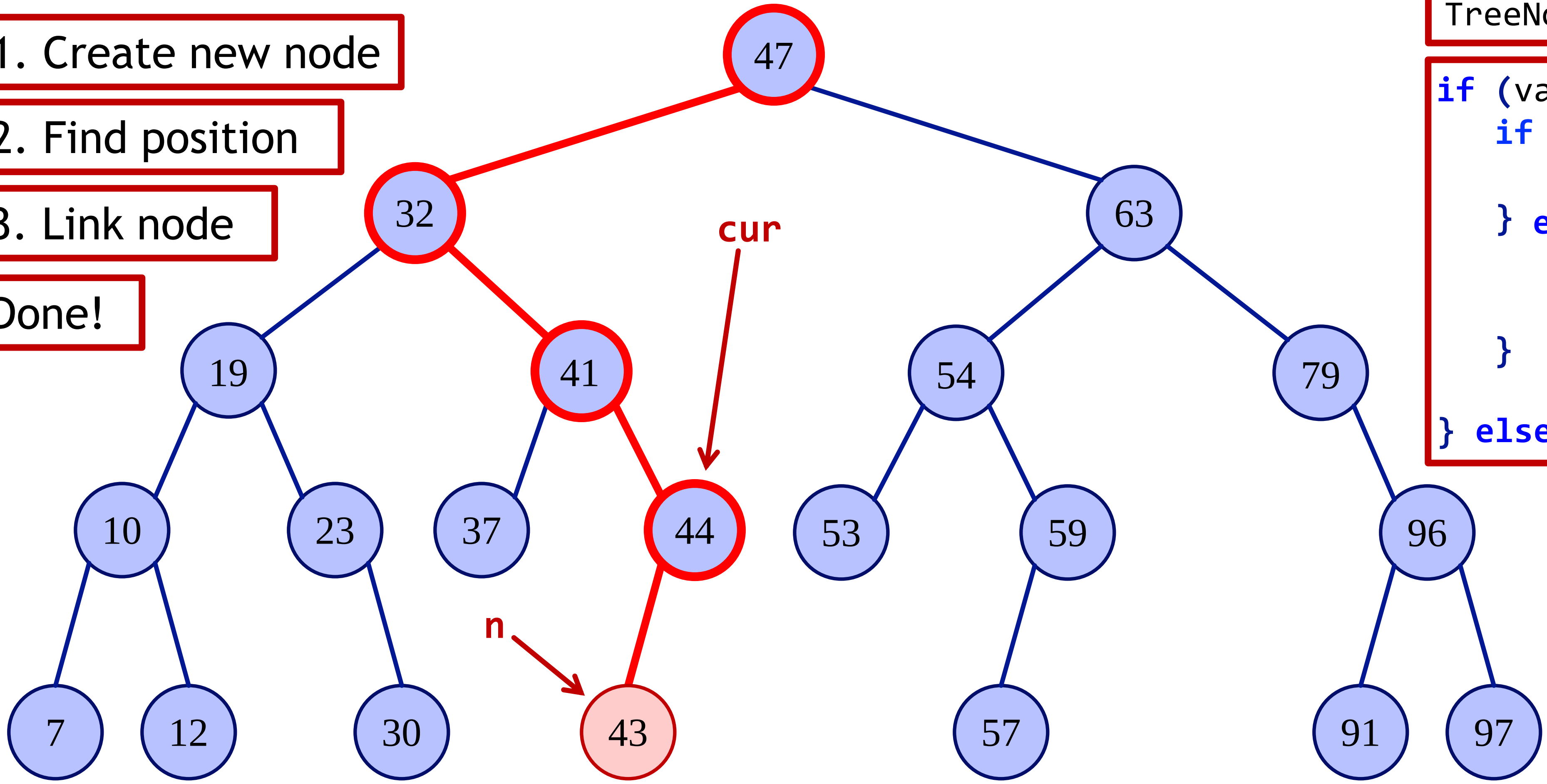
Insert 43

1. Create new node

2. Find position

3. Link node

Done!




```
TreeNode cur = root;
```

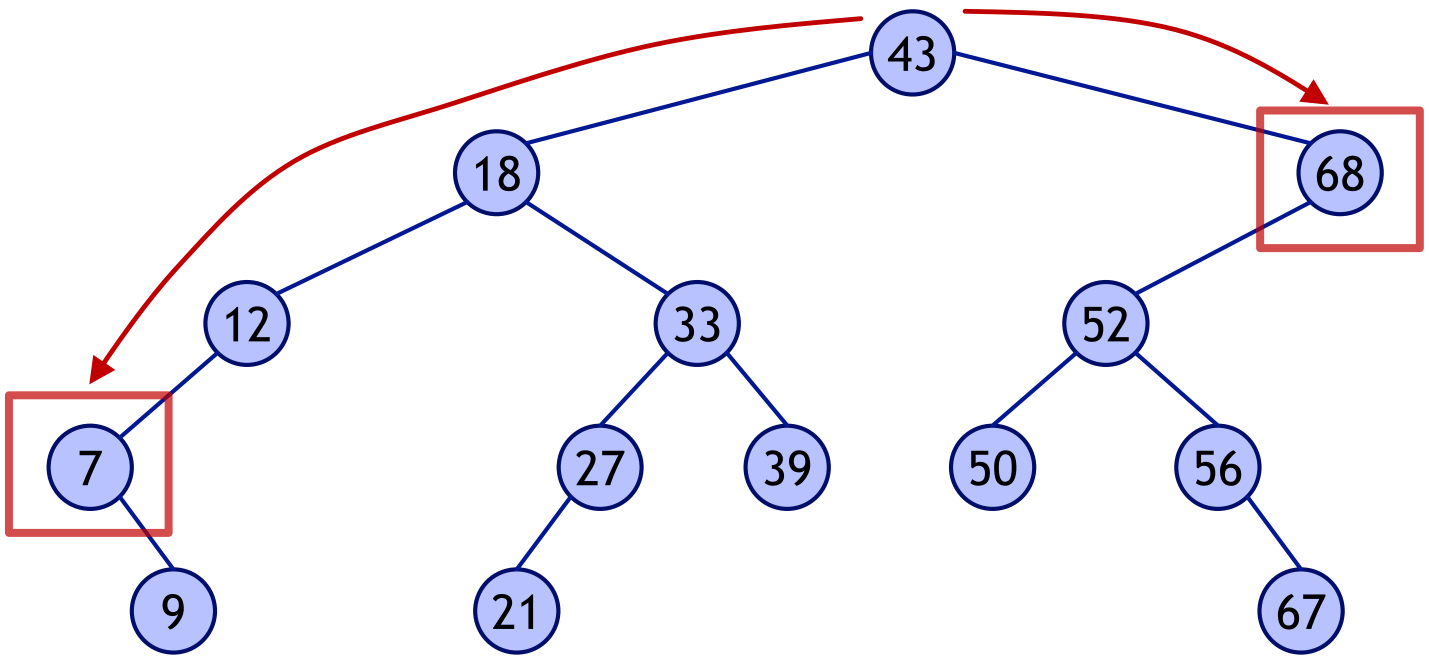
```
if (val < cur.data) {
   if (cur.left != null) {
      cur = cur.left;
   } else {
      cur.left = n;
      return;
   }

} else if (val > cur.data) {
```

```
TreeNode n = new TreeNode(val);
```

Finding the Minimum and Maximum Key

- Find minimum:
  - From the root, follow left reference arrows until no more left children exist
- Find maximum:
  - From the root, follow right reference arrows until no more right children exist



BST Removal

- The BST property must hold after each removal
- Removal is not as straightforward as search or insertion
  - With insertion the strategy is to insert a new leaf node
  - This avoids changing the internal structure of the tree
  - Unless a leaf node is removed, this isn’t possible with the removal operation
    - and we don’t know where the element we want to remove is located within the tree

BST Removal Cases

- There are a number of different cases we need to consider

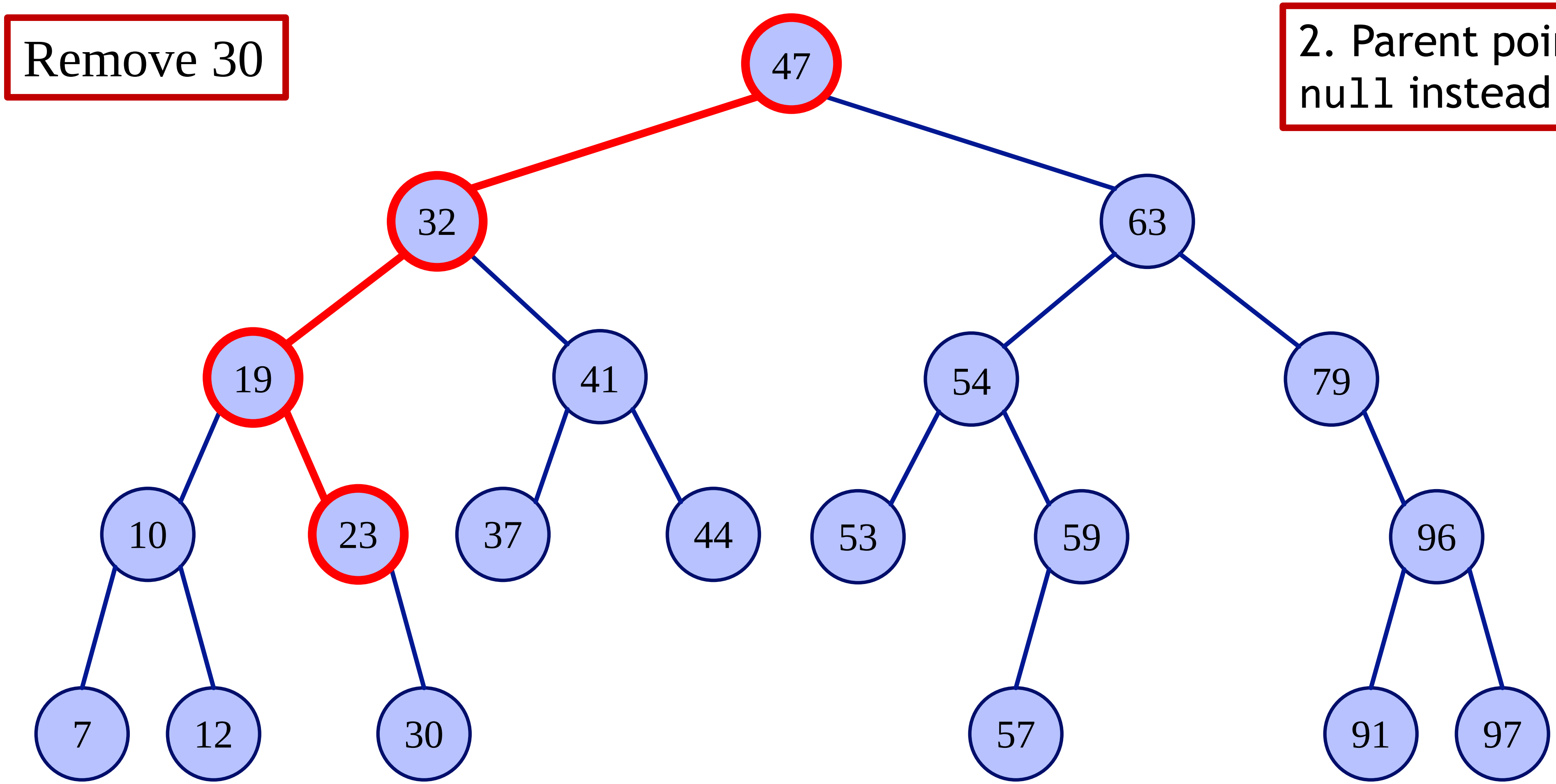
1. The node to be removed has no children
   - remove the node
   - assign its parent to point to null instead of the node
2. The node to be removed has one child
   - replace the node with the subtree rooted by the child
3. The node to be removed has two children
   - Choose a node to replace the node with… ?

BST Removal Example

1. Target is a leaf node (no children)

1. Locate the target

2. Parent points to null instead of node



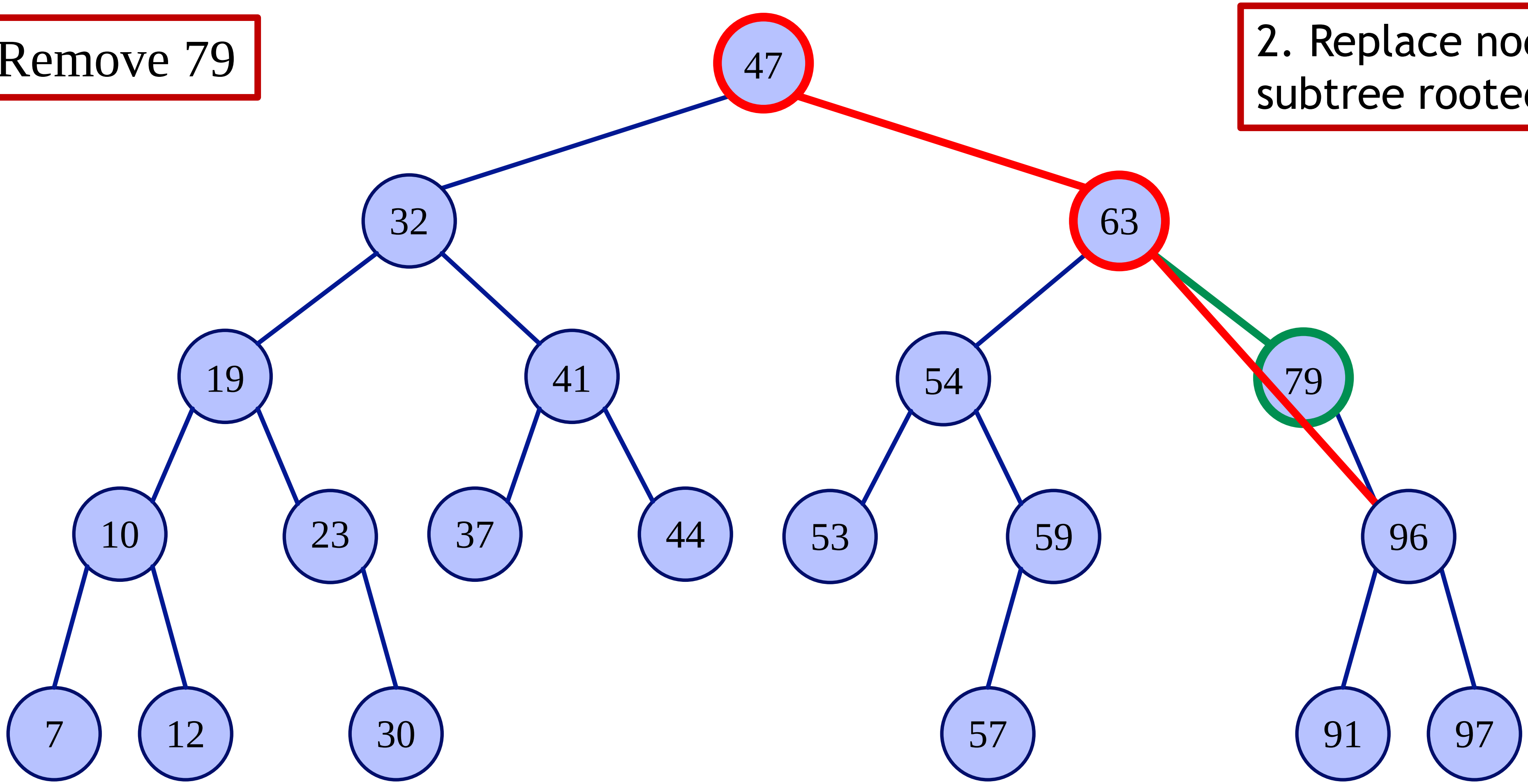
BST Removal Example

2. Target has one child

1. Locate the target

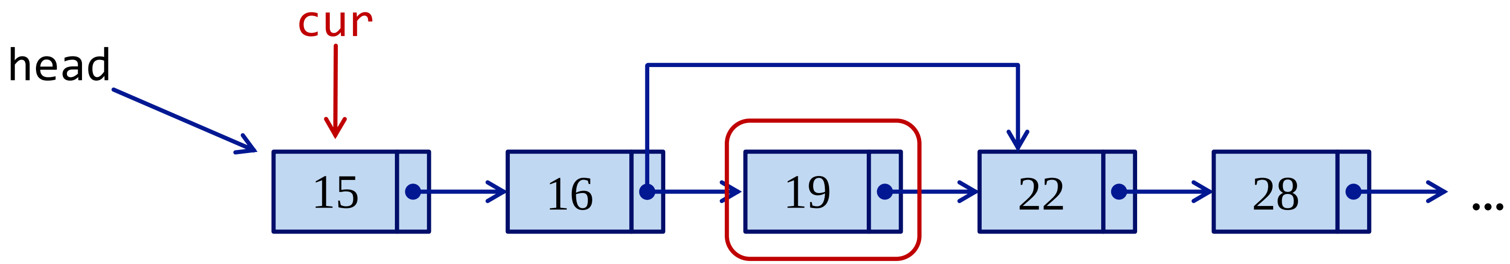
2. Replace node with subtree rooted by child

Remove 79



Looking at the **next** node

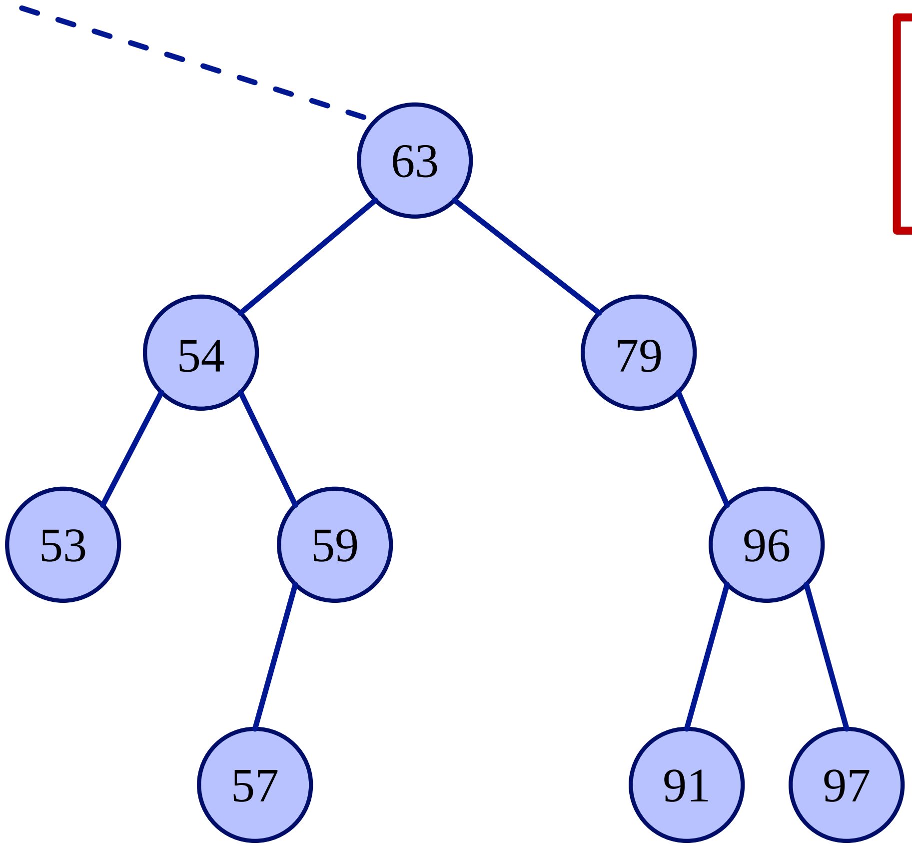
- One of the issues with implementing a BST is the necessity to look ahead at both children in order to update the reference arrows
  - this is similar to singly-linked lists when we needed to look ahead for insertion and removal




```
cur.next = cur.next.next;
```

Looking ahead

Remove 59



1. Locate the node to remove and its parent

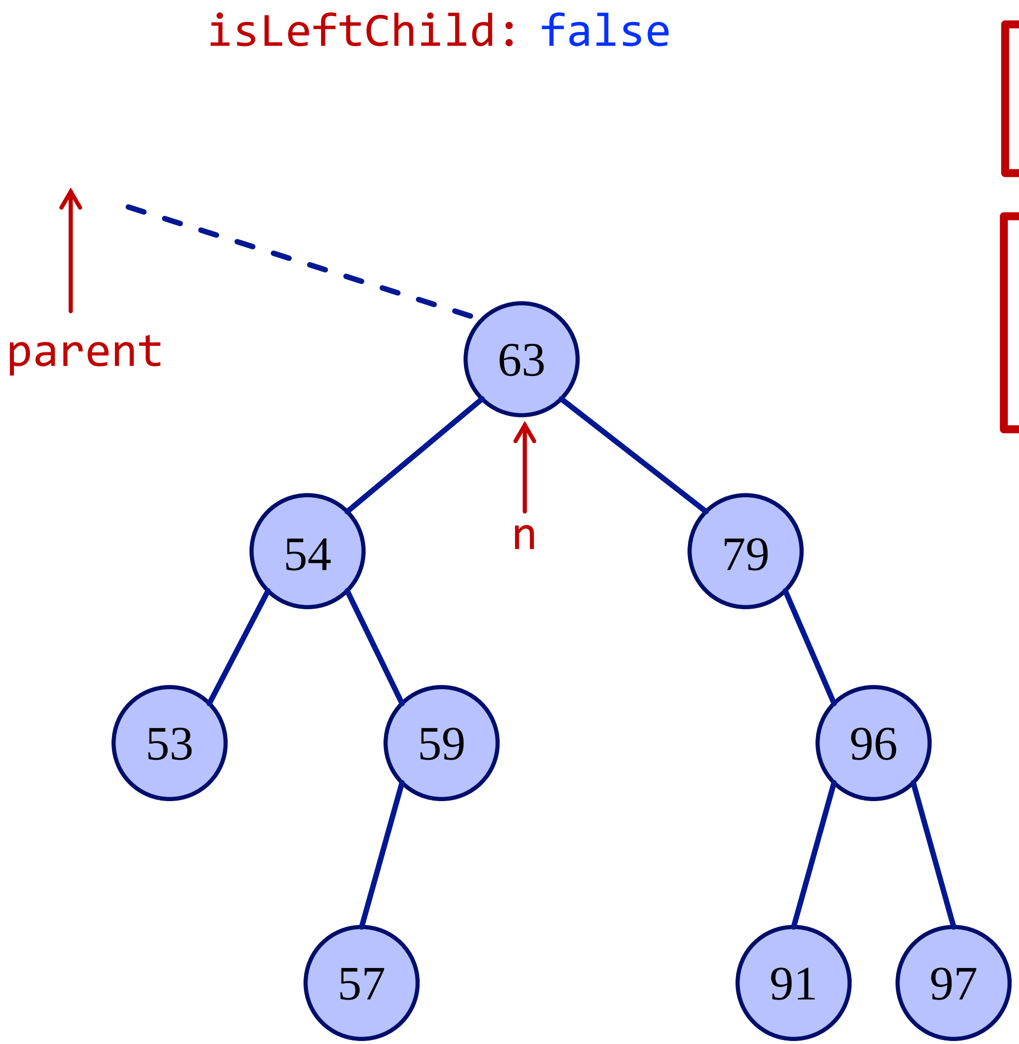
2. To make the correct link, we need to know if the node to be removed is a left or right child

```
if (n == NULL) {
   return;
}

if (target < n.data) {
   parent = n;
   n = n.left;
   isLeftChild = true;
} else {
   parent = n;
   n = n.right;
   isLeftChild = false;
}
```

Looking ahead

Remove 59



1. Locate the node to remove and its parent

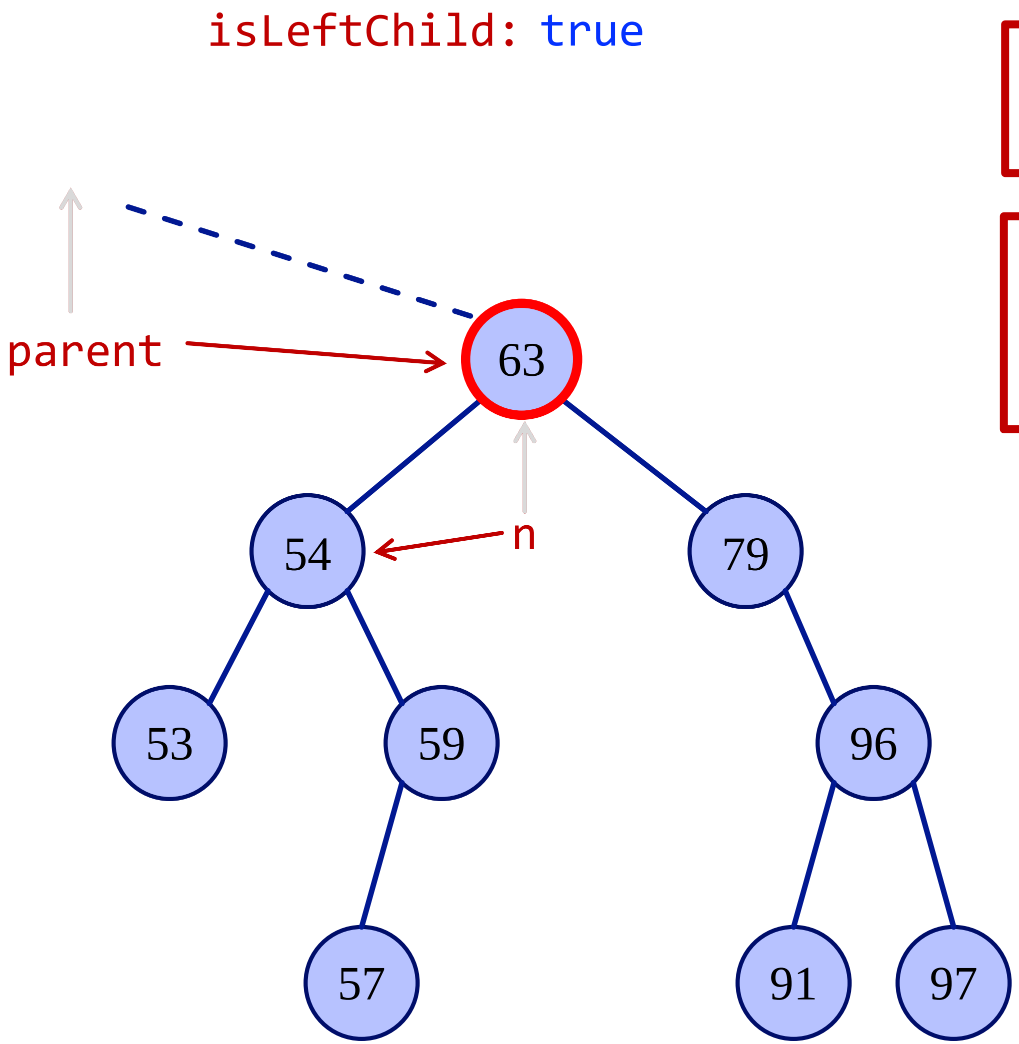
2. To make the correct link, we need to know if the node to be removed is a left or right child

```
if (n == NULL) {
   return;
}

if (target < n.data) {
   parent = n;
   n = n.left;
   isLeftChild = true;
} else {
   parent = n;
   n = n.right;
   isLeftChild = false;
}
```

Looking ahead

Remove 59



1. Locate the node to remove and its parent

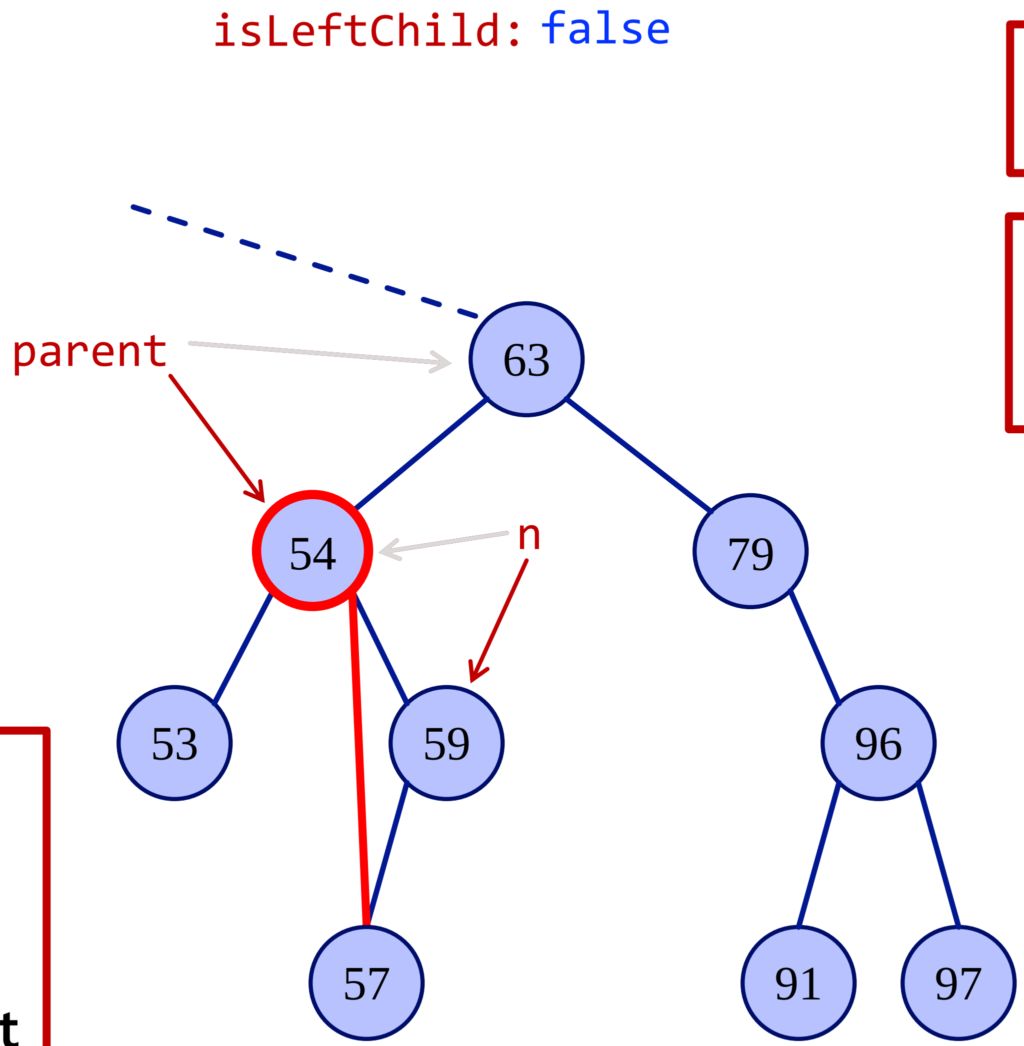
2. To make the correct link, we need to know if the node to be removed is a left or right child

```
if (n == NULL) {
   return;
}

if (target < n.data) {
   parent = n;
   n = n.left;
   isLeftChild = true;
} else {
   parent = n;
   n = n.right;
   isLeftChild = false;
}
```

Looking ahead

Remove 59



1. Locate the node to remove and its parent

2. To make the correct link, we need to know if the node to be removed is a left or right child

```
if (n == NULL) {
   return;
}

if (target < n.data) {
   parent = n;
   n = n.left;
   isLeftChild = true;
} else {
   parent = n;
   n = n.right;
   isLeftChild = false;
}
```

Now when the node to remove is found, there is enough information to remove node 59 and create a link going **right** from node 54.

Removing a Node with two Children

- The most difficult case is when the node to be removed has 2 children
  - We can’t just replace the node with its only child
- Which child should we replace the node with?
  - Can we just arbitrarily pick one?
  - What happens if the replacement nodes have children?
- Let’s take a look!

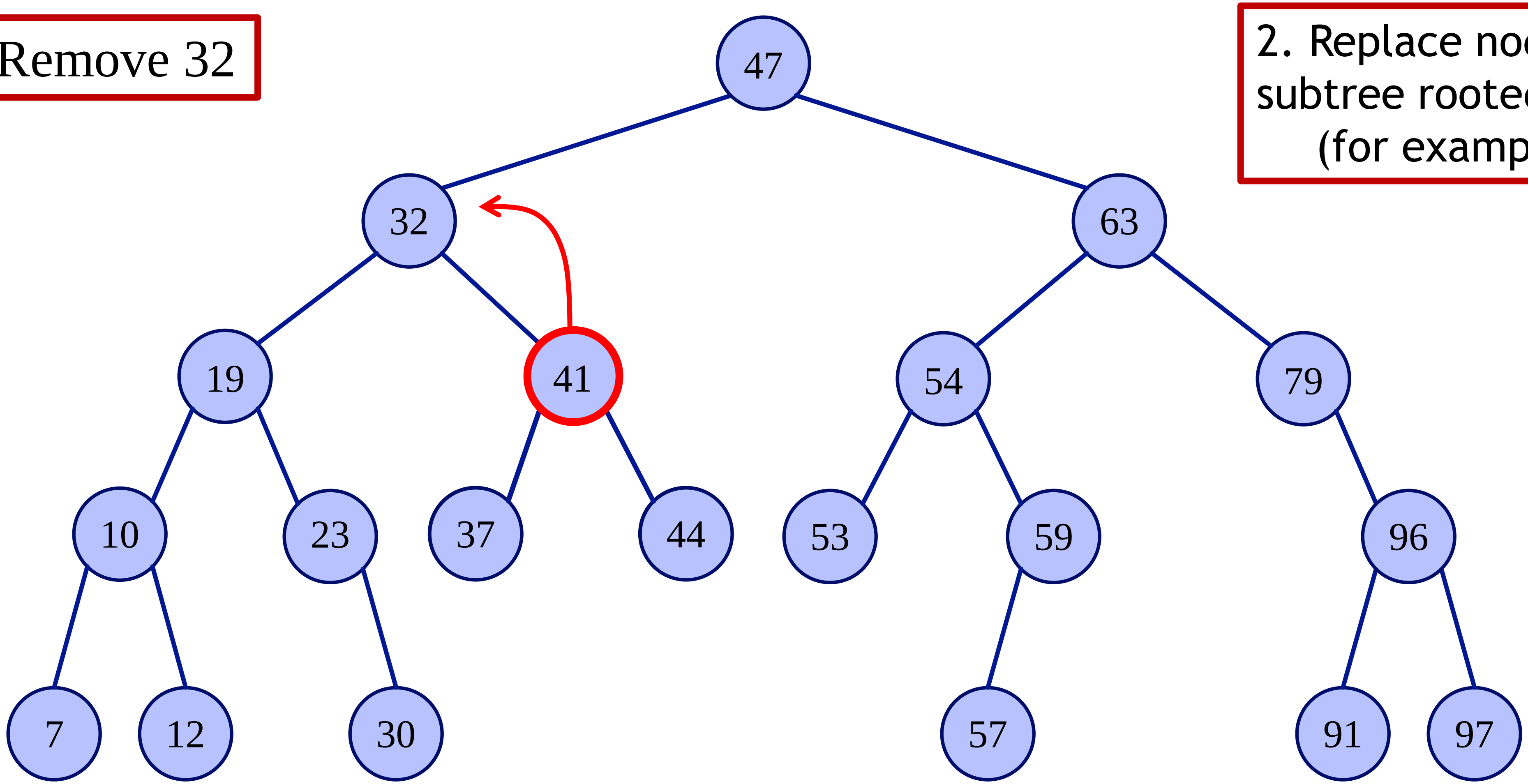
BST Removal Example

3. Target has two children

1. Locate the target

2. Replace node with subtree rooted by child (for example, 41)

Remove 32



BST Removal Example

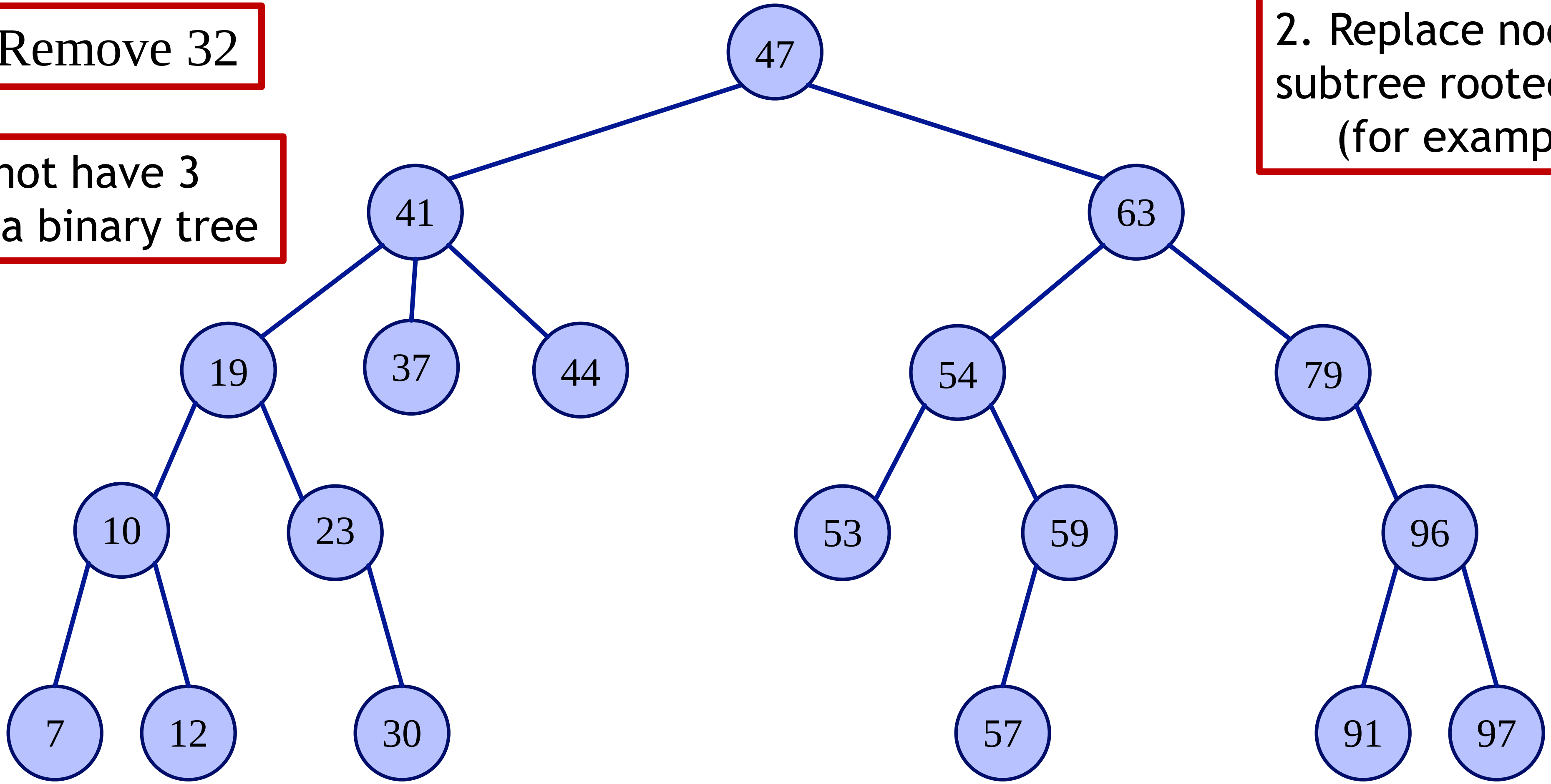
3. Target has two children

1. Locate the target

Remove 32

2. Replace node with subtree rooted by child (for example, 41)

A node cannot have 3 children in a binary tree



BST Removal Example

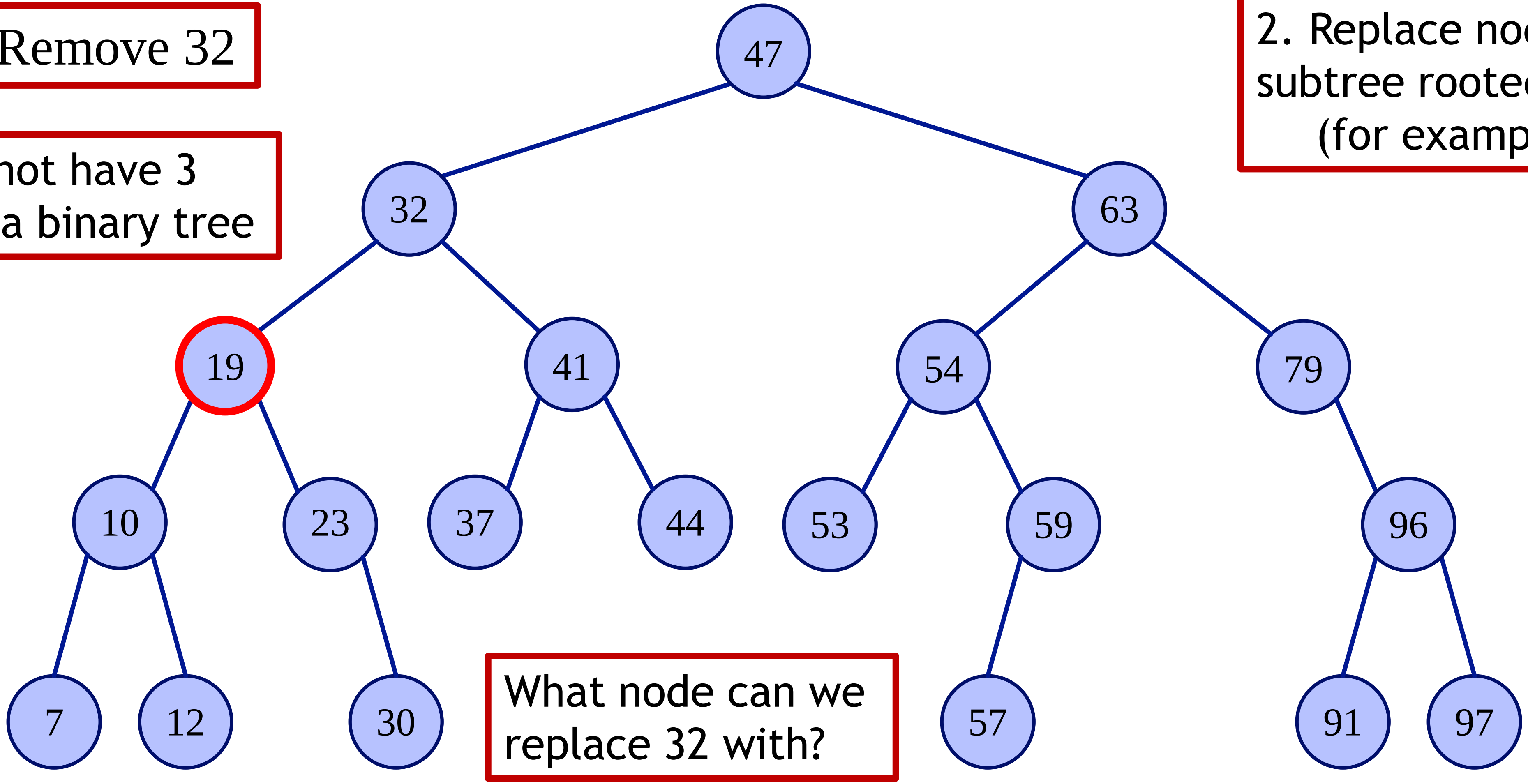
3. Target has two children

1. Locate the target

2. Replace node with subtree rooted by child (for example, 41)

Remove 32

A node cannot have 3 children in a binary tree



What node can we replace 32 with?

Replacement Nodes

- When a node has two children, instead of replacing it with one of its children, it will need to be replaced with its **predecessor** or **successor**

- A predecessor is the rightmost node in a node's left subtree
  - The predecessor is the node in the left subtree with the highest key
- A successor is the leftmost node in a node's right subtree
  - The successor is the node in the right subtree with the lowest key

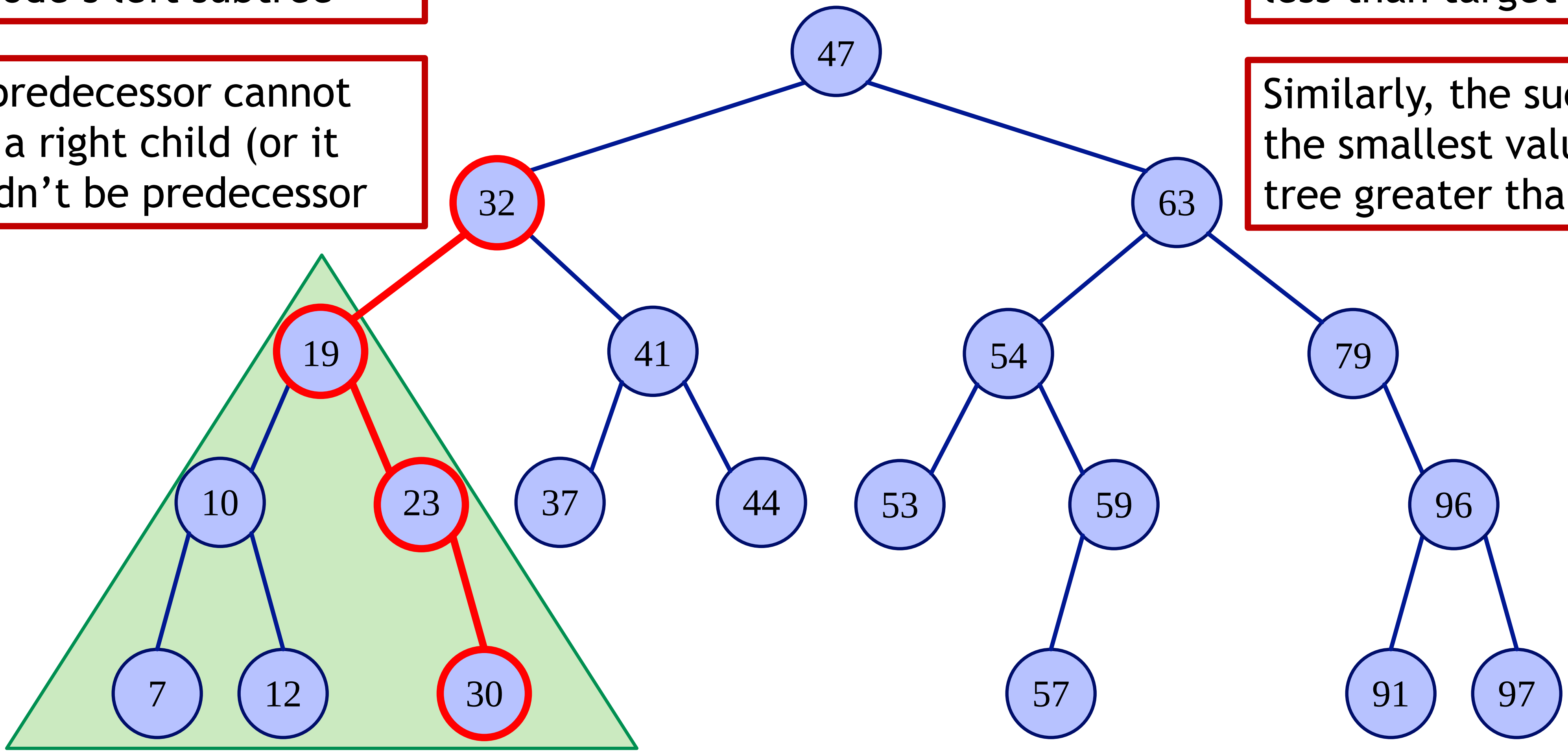
- The predecessor and successor are the only two nodes that can replace the node and maintain the binary search tree property
  - They are also good choices because neither of them have 2 children

Predecessor Node

The predecessor of a node is the rightmost node in the node's left subtree

The predecessor cannot have a right child (or it wouldn't be predecessor

Remove 32



In a BST, the predecessor is the largest value in the tree less than target node

Similarly, the successor is the smallest value in the tree greater than the target

Predecessor and Successor

- Both are good candidates to replace the node to remove with
  - They are easy to locate
  - They don’t have two children
  - The maintain the BST property

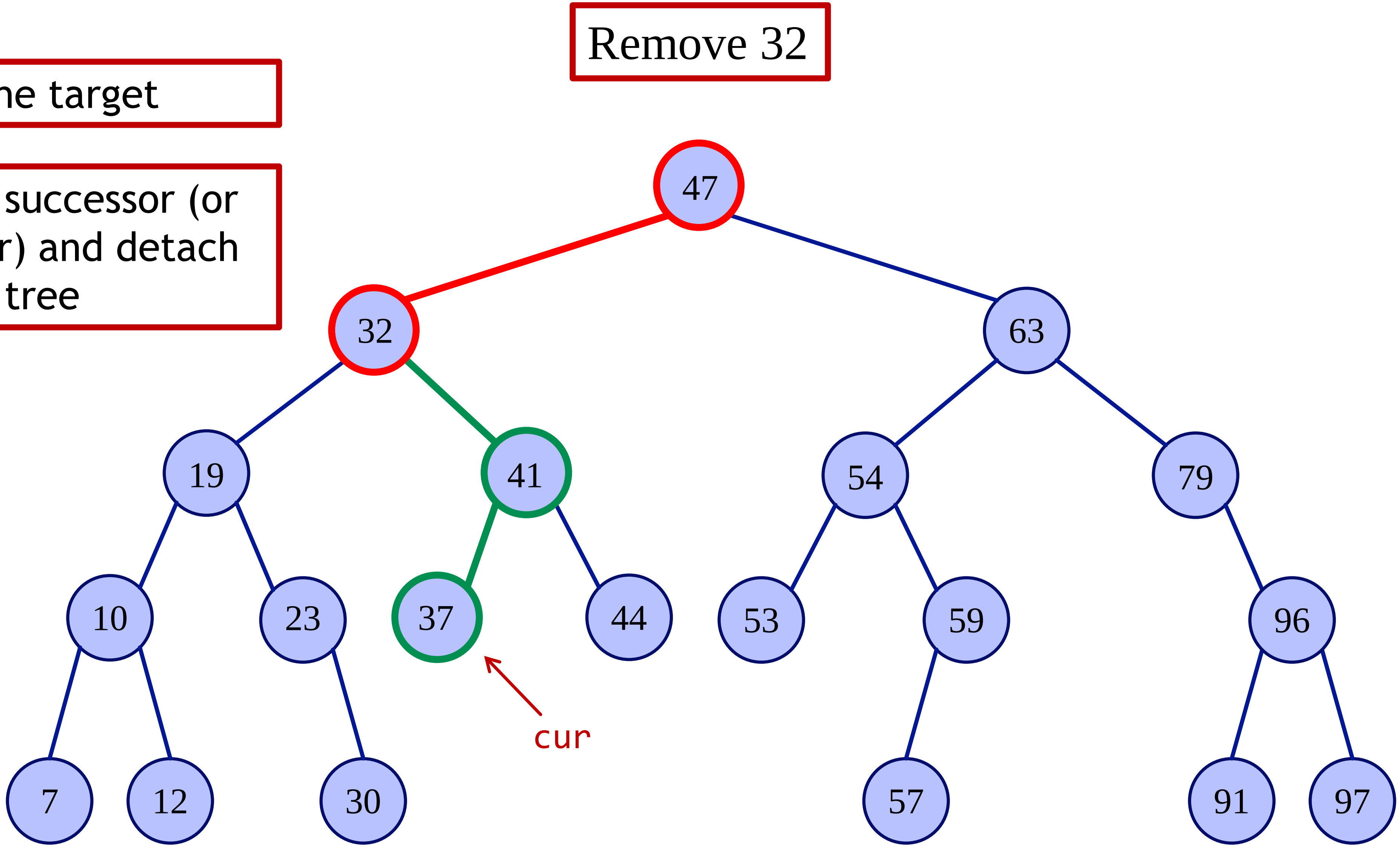
- Pick one to use for a removal of a node with two children
  - Pick either one, but be consistent!

- Let’s revisit our example of removing a node with two children

BST Removal Example

1. Locate the target

2. Find the successor (or predecessor) and detach it from the tree



BST Removal Example

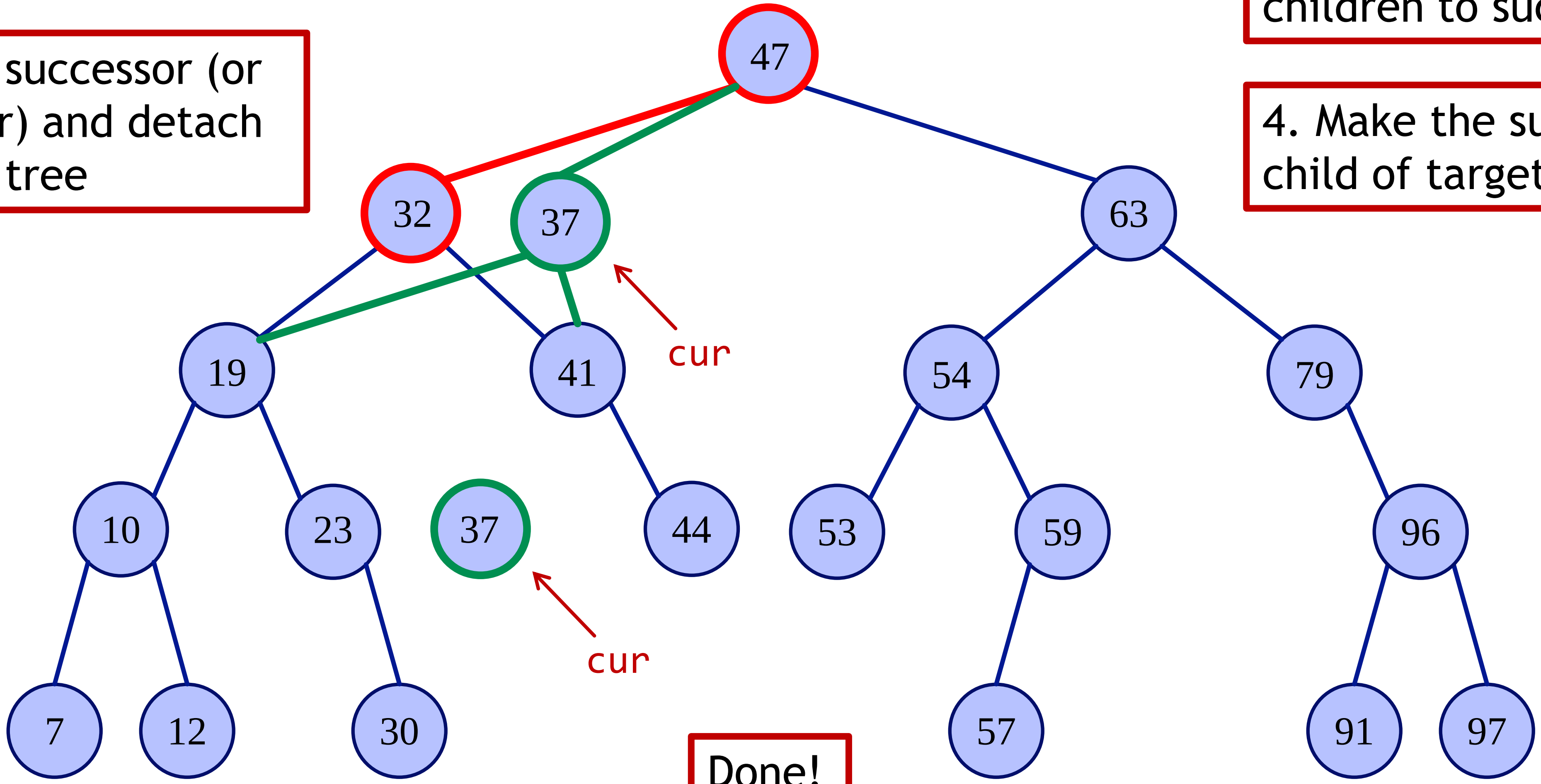
1. Locate the target

2. Find the successor (or predecessor) and detach it from the tree

3. Attach removed node's children to successor

4. Make the successor the child of target's parent

Remove 32



Done!