Chapter 10: Search Trees

Data Structures

Binary search trees | Search and insertion | Deletion | O(h) analysis | Balance and rotation | AVL trees | Red-black trees | Splay trees | B-trees and B+ trees

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/data_structures/slides/Chapter_10_Search_Trees.html

INTRODUCTION

Learning Outcomes

BST operations

Carry out search, insertion, and deletion in a binary search tree.

Why balance

Explain how the height of a tree drives the cost of its operations.

Rebalancing

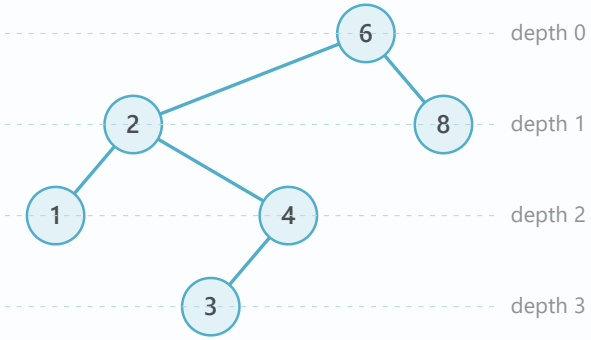
Read AVL rotations and apply the red-black tree rules.

Comparison

Choose among BSTs, AVL, red-black, splay, and B-trees for a given use.

Search Trees

- Each node holds an item, and each item has a **key**.
- Items are stored **in order** of their keys.
- A search starts at the root and follows a single branch at each node.
- The cost is proportional to the **height h** of the tree: **O(h)**.
- With n nodes, h is $\approx \log_2 n$ at best and $n - 1$ at worst.



The whole story of search trees: operations take O(h), so **keeping h small** (a balanced tree) makes them O(log n).

Basic Operations

Search

Returns true if the key is in the tree, false otherwise.

```
search(root, 5) → true
```

Insertion

Adds a new item as a leaf in the right place, preserving the order.

```
insert(root, 7)
```

Deletion

Removes the item with the given key and keeps the order intact.

```
delete(root, 40)
```

Traversal

Visits the items in a fixed order; inorder traversal yields the keys sorted.

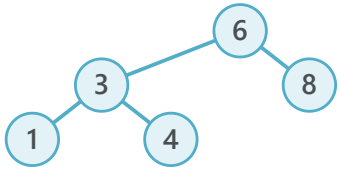
```
inorder(root) → 1 2 3 4 6 8
```

Also: the smallest / largest key, a key's successor / predecessor, and range queries. All O(h).

Kinds of Search Trees

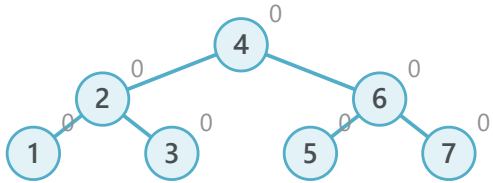
Binary search tree (BST)

Each key is greater than every key in its left subtree and less than every key in its right subtree. No balance guarantee.



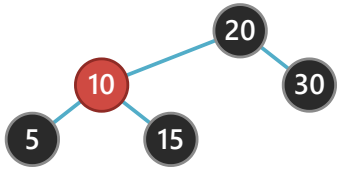
AVL tree

At every node the subtree heights differ by at most 1 (the balance factor). Rotations keep it balanced.



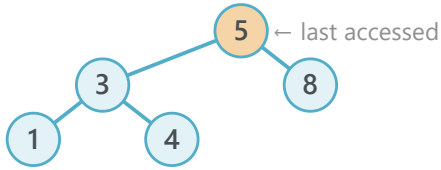
Red-black tree

Node colors and a few rules cap the height at $2 \cdot \log_2(n+1)$.



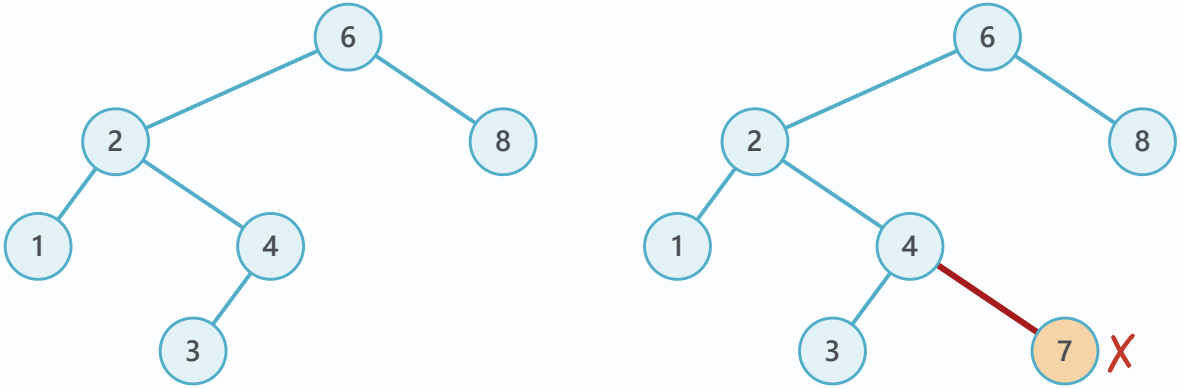
Splay tree

Rotates each accessed node up to the root; O(log n) amortized per operation.



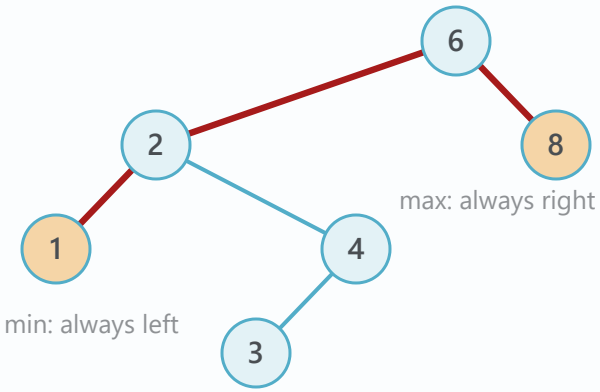
Binary Search Trees

For every node x: **all keys in the left subtree < x.key < all keys in the right subtree**.



- The property covers the **whole subtree**, not just the children: on the right, 7 > 4 holds, but 7 cannot sit left of 6.
- Duplicate keys are rejected or always sent to one side (say, the right).

Minimum, Maximum, and Sorted Traversal



An **inorder** traversal (left → root → right) yields the keys **in sorted order**:

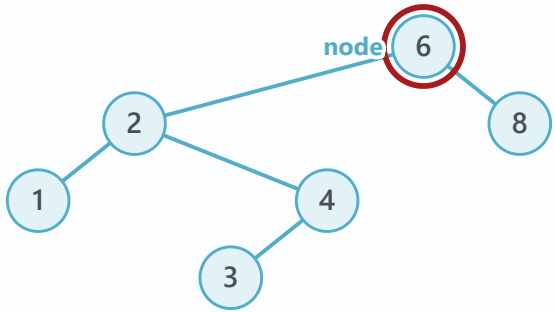
1 2 3 4 6 8

```
// Precondition: node != null
BinaryTreeNode<E> minValue(
        BinaryTreeNode<E> node) {
    while (node.leftChild != null)
        node = node.leftChild; // go left
    return node;  // the smallest
}
// maxValue: the same, with rightChild

void inorder(BinaryTreeNode<E> d) {
    if (d == null) return;
    inorder(d.leftChild);   // left
    System.out.print(d.data + " ");
    inorder(d.rightChild);  // right
}
```

Code Trace: `minValue` and `maxValue`

```
▶ BinaryTreeNode<E> minValue(
2         BinaryTreeNode<E> node) {
3     while (node.leftChild != null)
4         node = node.leftChild;  // go left
5     return node;    // the smallest
6 }
7 BinaryTreeNode<E> maxValue(
8         BinaryTreeNode<E> node) {
9     while (node.rightChild != null)
10        node = node.rightChild; // go right
11    return node;    // the largest
12 }
```



`minValue(root)` : node starts at the root (6). Precondition: the tree is not empty.

Step 1 / 12

VARIABLES

`node = node(6)`

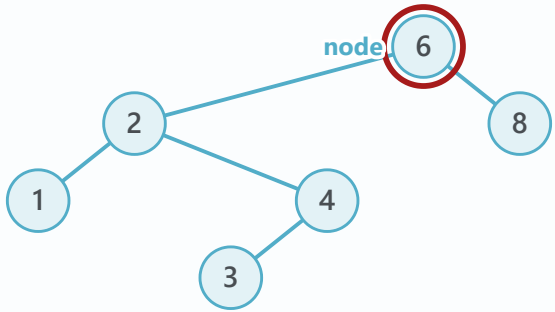
CALL STACK

`minValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;   // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;   // the largest
12 }
```



node.leftChild = 2 ≠ null → keep looping.

Step 2 / 12

VARIABLES

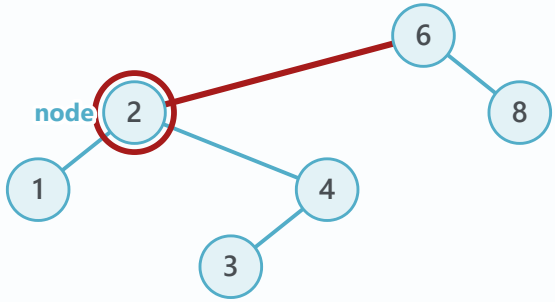
`node = node(6)` `node.leftChild = node(2)`

CALL STACK

- `minValue(6)`
- `main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
▶          node = node.leftChild;  // go left
5      return node;    // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;    // the largest
12 }
```



node = node.leftChild → 2. Moving left.

Step 3 / 12

VARIABLES

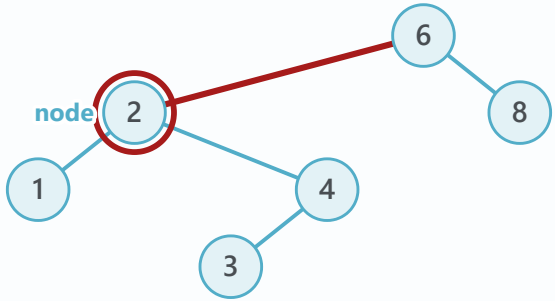
`node = node(2)`

CALL STACK

- `minValue(6)`
- `main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;   // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;   // the largest
12 }
```



node.leftChild = 1 ≠ null → keep looping.

Step 4 / 12

VARIABLES

`node = node(2)` `node.leftChild = node(1)`

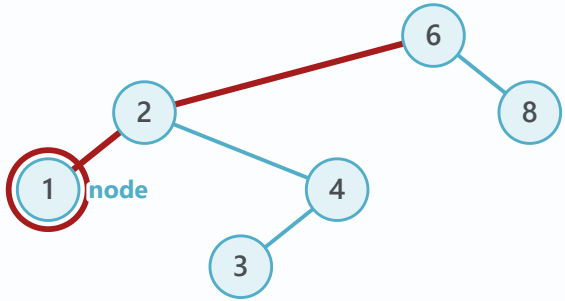
CALL STACK

`minValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
▶          node = node.leftChild;  // go left
5      return node;    // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;    // the largest
12 }
```



node = node.leftChild → 1. Moving left.

VARIABLES

`node = node(1)`

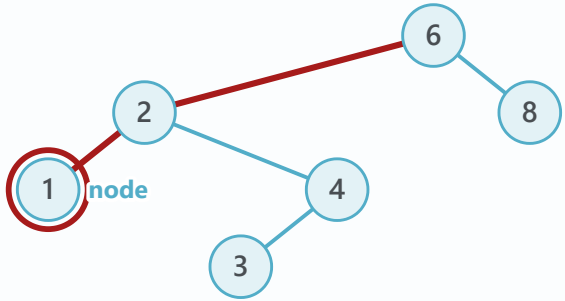
CALL STACK

`minValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;   // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;   // the largest
12 }
```



node.leftChild == null → the loop ends. 1 has no left child.

Step 6 / 12

VARIABLES

`node = node(1)` `node.leftChild = null`

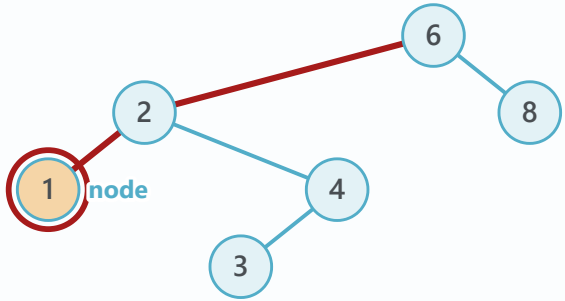
CALL STACK

`minValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
▶      return node;   // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;   // the largest
12 }
```



return 1: the smallest key. (3 is deeper but lies left of 4; the minimum is always the leftmost node.)

VARIABLES

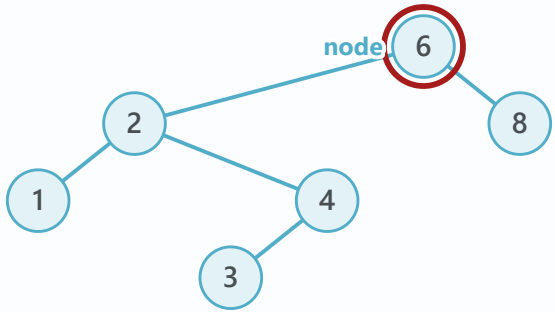
`node = node(1)` `result = 1`

CALL STACK

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;    // the smallest
6  }
   BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;    // the largest
12 }
```



`maxValue(root)` : node starts at the root (6). Precondition: the tree is not empty.

Step 8 / 12

VARIABLES

`node = node(6)`

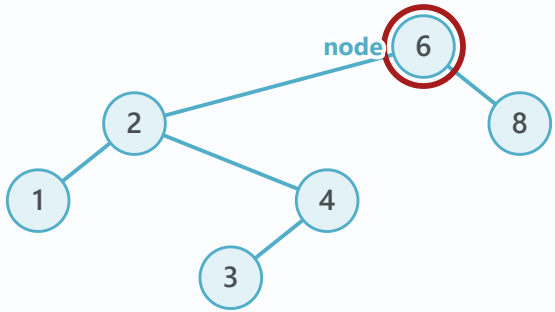
CALL STACK

`maxValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;    // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
▶      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;    // the largest
12 }
```



node.rightChild = 8 ≠ null → keep looping.

Step 9 / 12

VARIABLES

`node = node(6)` `node.rightChild = node(8)`

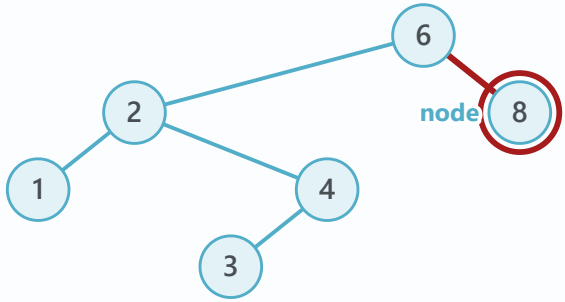
CALL STACK

`maxValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;    // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
▶          node = node.rightChild; // go right
11     return node;    // the largest
12 }
```



node = node.rightChild → 8. Moving right.

Step 10 / 12

VARIABLES

`node = node(8)`

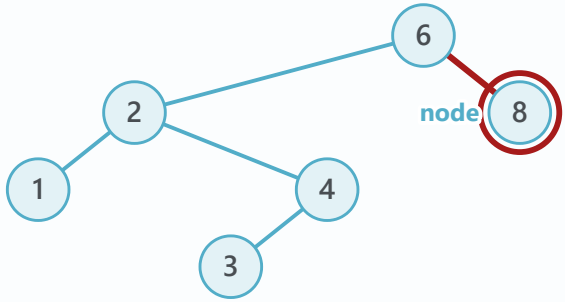
CALL STACK

`maxValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;    // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
▶      while (node.rightChild != null)
10         node = node.rightChild; // go right
11     return node;    // the largest
12 }
```



node.rightChild == null → the loop ends. 8 has no right child.

Step 11 / 12

VARIABLES

`node = node(8)` `node.rightChild = null`

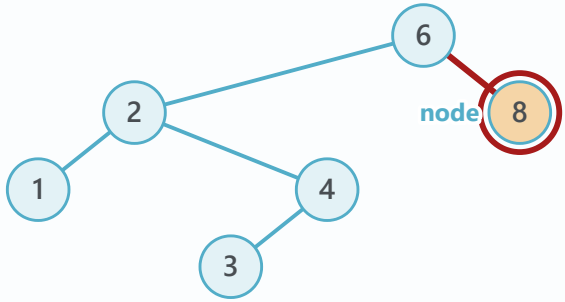
CALL STACK

`maxValue(6)`

`main()`

Code Trace: `minValue` and `maxValue`

```
1  BinaryTreeNode<E> minValue(
2          BinaryTreeNode<E> node) {
3      while (node.leftChild != null)
4          node = node.leftChild;  // go left
5      return node;   // the smallest
6  }
7  BinaryTreeNode<E> maxValue(
8          BinaryTreeNode<E> node) {
9      while (node.rightChild != null)
10         node = node.rightChild; // go right
▶      return node;   // the largest
12 }
```



return 8: the largest key. No comparisons, just links followed: **O(h)**.

Step 12 / 12

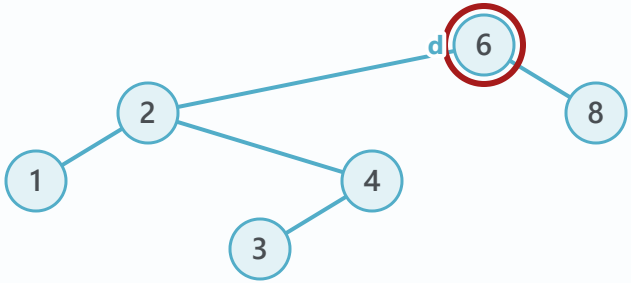
VARIABLES

`node = node(8)` `result = 8`

CALL STACK

`main()`

Code Trace: `inorder(root)`



`inorder(6)` : d ≠ null. **Left** subtree first:
`inorder(2)` ; 6 waits.

Step 1 / 20

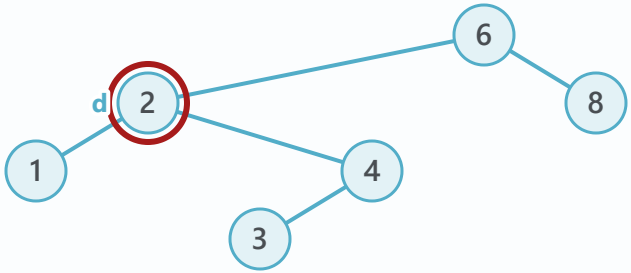
```
1 void inorder(BinaryTreeNode<E> d) {
▶     if (d == null) return;
▶     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

CALL STACK

- **`inorder(6)`**
- `main()`

OUTPUT

Code Trace: `inorder(root)`



`inorder(2)` : d ≠ null. **Left** subtree first:
`inorder(1)` ; 2 waits.

Step 2 / 20

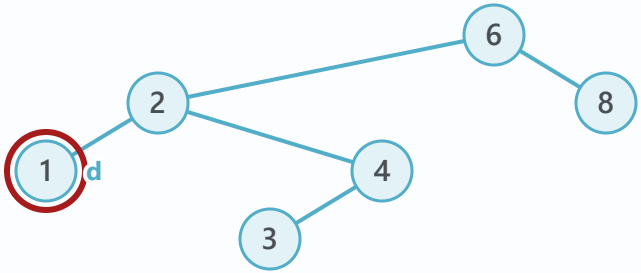
```
1 void inorder(BinaryTreeNode<E> d) {
▶     if (d == null) return;
▶     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

CALL STACK

- **inorder(2)**
- inorder(6)
- main()

OUTPUT

Code Trace: `inorder(root)`



`inorder(1)` : no left child; `inorder(null)` returns at line 2.

Step 3 / 20

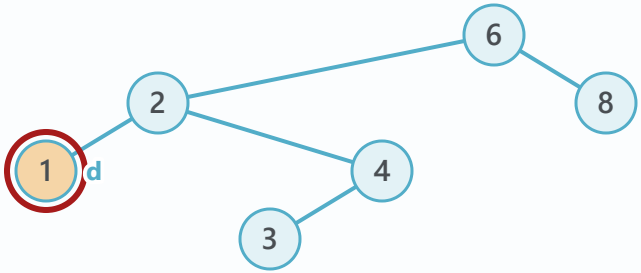
```
1 void inorder(BinaryTreeNode<E> d) {
▶     if (d == null) return;
▶     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

CALL STACK

- **inorder(1)**
- inorder(2)
- inorder(6)
- main()

OUTPUT

Code Trace: `inorder(root)`



Left side done → **print 1**. (Every key on its left is smaller.)

Step 4 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
▶     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

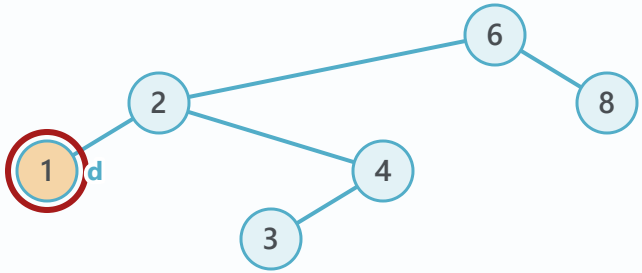
CALL STACK

- **inorder(1)**
- inorder(2)
- inorder(6)
- main()

OUTPUT

```
1
```

Code Trace: `inorder(root)`



No right child → the frame closes; `inorder(2)` resumes.

Step 5 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
▶     inorder(d.rightChild);  // right
▶ }
```

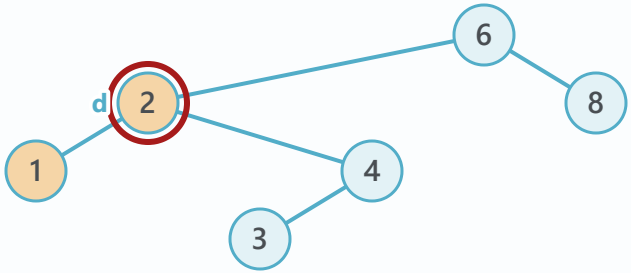
CALL STACK

- **inorder(1)**
- inorder(2)
- inorder(6)
- main()

OUTPUT

```
1
```

Code Trace: `inorder(root)`



Left side done → **print 2**. (Every key on its left is smaller.)

Step 6 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
▶     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

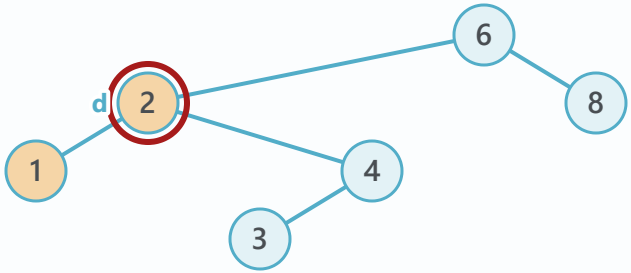
CALL STACK

- **inorder(2)**
- inorder(6)
- main()

OUTPUT

```
1 2
```

Code Trace: `inorder(root)`



Now the **right** subtree: `inorder(4)`.

Step 7 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
▶     inorder(d.rightChild);  // right
6 }
```

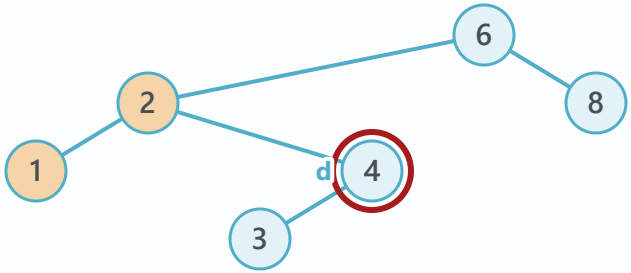
CALL STACK

- **inorder(2)**
- inorder(6)
- main()

OUTPUT

```
1 2
```

Code Trace: `inorder(root)`



`inorder(4)` : d ≠ null. **Left** subtree first: `inorder(3)` ; 4 waits.

Step 8 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
▶       if (d == null) return;
▶       inorder(d.leftChild);   // left
4       System.out.print(d.data + " ");
5       inorder(d.rightChild);  // right
6   }
```

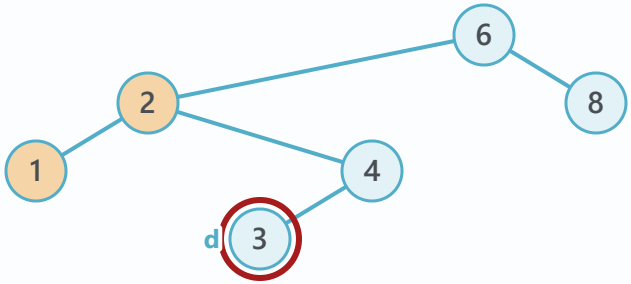
CALL STACK

- **`inorder(4)`**
- `inorder(2)`
- `inorder(6)`
- `main()`

OUTPUT

```
1 2
```

Code Trace: `inorder(root)`



`inorder(3)` : no left child; `inorder(null)` returns at line 2.

Step 9 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
▶     if (d == null) return;
▶     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

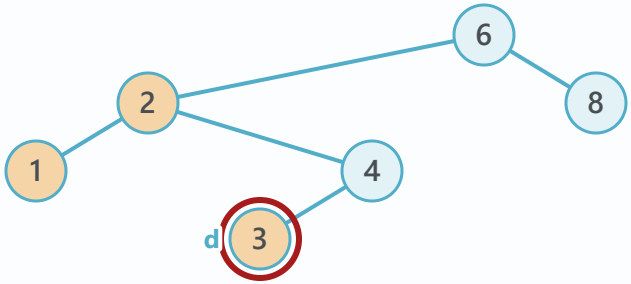
CALL STACK

- **inorder(3)**
- inorder(4)
- inorder(2)
- inorder(6)
- main()

OUTPUT

```
1 2
```

Code Trace: `inorder(root)`



Left side done → **print 3**. (Every key on its left is smaller.)

Step 10 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
▶     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

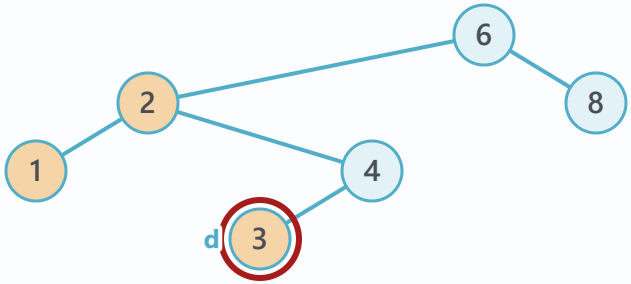
CALL STACK

- **inorder(3)**
- inorder(4)
- inorder(2)
- inorder(6)
- main()

OUTPUT

```
1 2 3
```

Code Trace: `inorder(root)`



No right child → the frame closes; `inorder(4)` resumes.

Step 11 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
▶     inorder(d.rightChild);  // right
▶ }
```

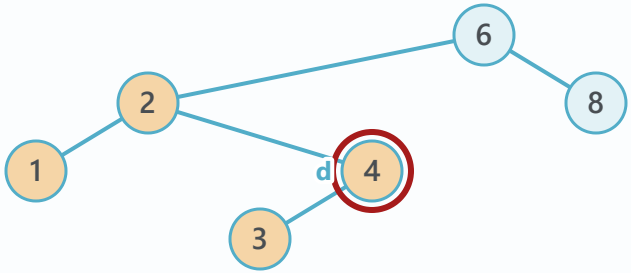
CALL STACK

- **inorder(3)**
- inorder(4)
- inorder(2)
- inorder(6)
- main()

OUTPUT

```
1 2 3
```

Code Trace: `inorder(root)`



Left side done → **print 4**. (Every key on its left is smaller.)

Step 12 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
▶     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

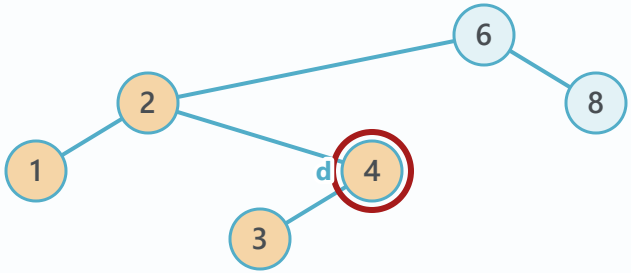
CALL STACK

- **inorder(4)**
- inorder(2)
- inorder(6)
- main()

OUTPUT

```
1 2 3 4
```

Code Trace: `inorder(root)`



No right child → the frame closes; `inorder(2)` resumes.

Step 13 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
▶     inorder(d.rightChild);  // right
▶ }
```

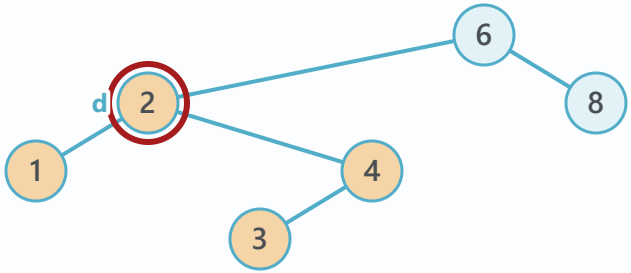
CALL STACK

- **inorder(4)**
- inorder(2)
- inorder(6)
- main()

OUTPUT

```
1 2 3 4
```

Code Trace: `inorder(root)`



2's right subtree is done too → `inorder(2)` closes, and its caller resumes.

Step 14 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
▶     inorder(d.rightChild);  // right
▶ }
```

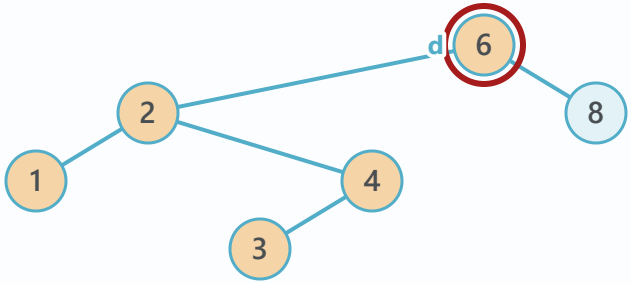
CALL STACK

- **inorder(2)**
- inorder(6)
- main()

OUTPUT

```
1 2 3 4
```

Code Trace: `inorder(root)`



Left side done → **print 6**. (Every key on its left is smaller.)

Step 15 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
▶     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

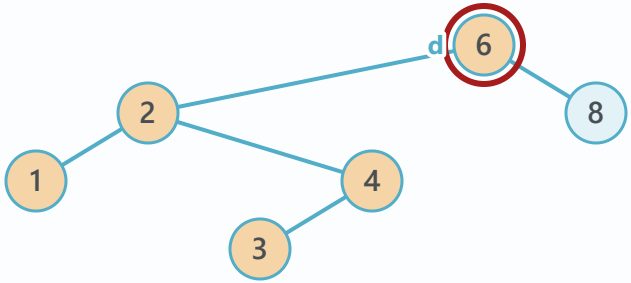
CALL STACK

- **inorder(6)**
- main()

OUTPUT

```
1 2 3 4 6
```

Code Trace: `inorder(root)`



Now the **right** subtree: `inorder(8)`.

Step 16 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
▶     inorder(d.rightChild);  // right
6 }
```

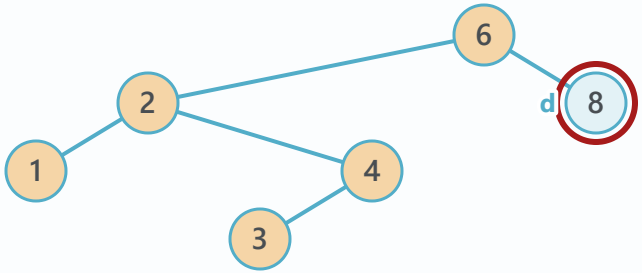
CALL STACK

- **inorder(6)**
- main()

OUTPUT

```
1 2 3 4 6
```

Code Trace: `inorder(root)`



`inorder(8)` : no left child; `inorder(null)` returns at line 2.

Step 17 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
▶     if (d == null) return;
▶     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

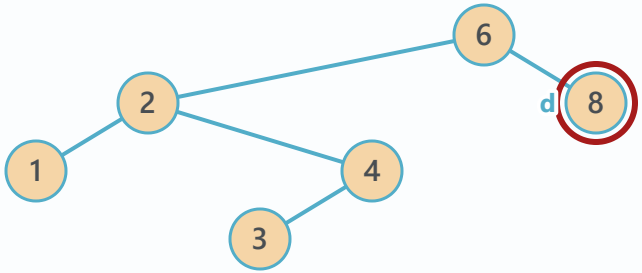
CALL STACK

- **inorder(8)**
- inorder(6)
- main()

OUTPUT

```
1 2 3 4 6
```

Code Trace: `inorder(root)`



Left side done → **print 8**. (Every key on its left is smaller.)

Step 18 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
▶     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

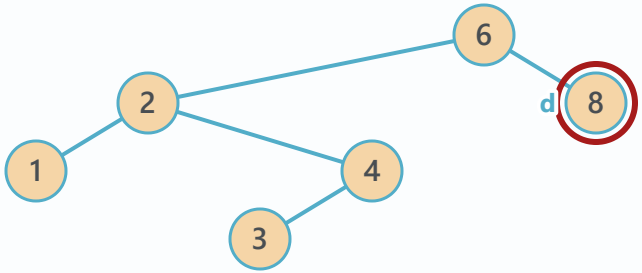
CALL STACK

- **inorder(8)**
- inorder(6)
- main()

OUTPUT

```
1 2 3 4 6 8
```

Code Trace: `inorder(root)`



No right child → the frame closes; `inorder(6)` resumes.

Step 19 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
▶     inorder(d.rightChild);  // right
▶ }
```

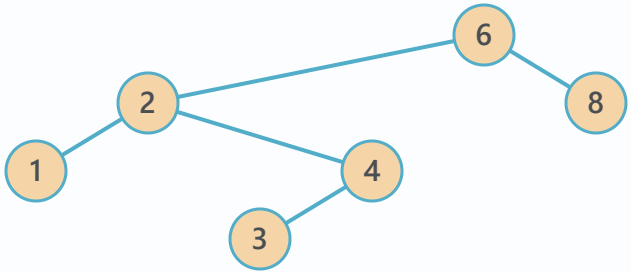
CALL STACK

- **inorder(8)**
- inorder(6)
- main()

OUTPUT

```
1 2 3 4 6 8
```

Code Trace: `inorder(root)`



All calls done; the output is **sorted**: 1 2 3 4 6 8. Each node printed once → **O(n)**.

Step 20 / 20

```
1 void inorder(BinaryTreeNode<E> d) {
2     if (d == null) return;
3     inorder(d.leftChild);   // left
4     System.out.print(d.data + " ");
5     inorder(d.rightChild);  // right
6 }
```

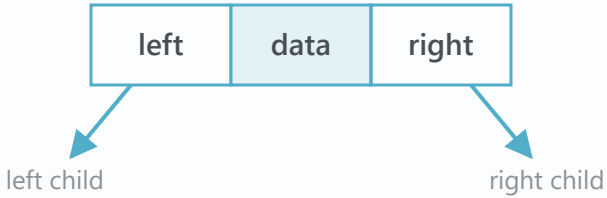
CALL STACK

`main()`

OUTPUT

```
1 2 3 4 6 8
```

The Tree Node



int keys

```
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

generic

```
class BinaryTreeNode<E extends Comparable<E>> {
    E data;                      // the object
    BinaryTreeNode<E> leftChild;  // left child
    BinaryTreeNode<E> rightChild; // right child

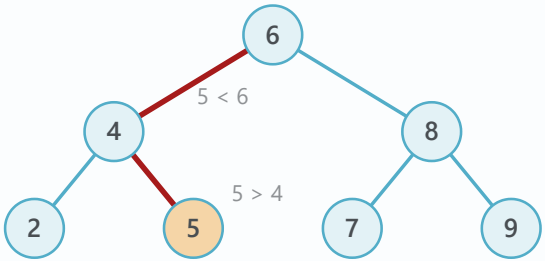
    BinaryTreeNode(E data) {
        this.data = data;
        // children start out null
    }
}
```

Keys are compared with `compareTo`, so E must be `Comparable<E>`.

BST OPERATIONS

Searching for a Key

- Start at the root; if the keys are equal, we are done.
- If smaller, continue in the left subtree; if larger, in the right.
- Reaching null means the key is not in the tree.




```
boolean search(BinaryTreeNode<E> root, E data) {
    if (root == null)
        return false;              // not found
    int k = data.compareTo(root.data);
    if (k == 0)
        return true;               // found
    else if (k < 0)
        return search(root.leftChild, data);
    else
        return search(root.rightChild, data);
}
```

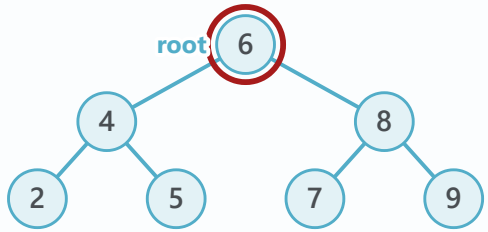
Compare objects with `compareTo` (or `equals`), not `==`, which only checks whether two references are identical.

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
▶      if (root == null)
3          return false;              // not found
▶      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```

CALL STACK

- `search(6, 5)`
- `main()`



`search(root, 5)` : root is node 6, not null. **k = compareTo = -1** since 5 < 6.

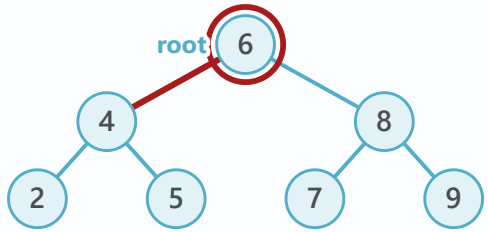
Step 1 / 20

VARIABLES

`root = node(6)` `data = 5` `k = -1`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;              // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
▶      else if (k < 0)
▶          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```



k < 0 → go left: `search(root.leftChild, 5)` opens a new frame (root = 4).

Step 2 / 20

CALL STACK

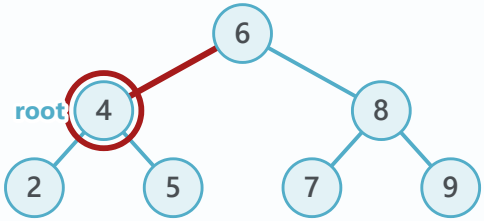
- `search(6, 5)`
- `main()`

VARIABLES

`root = node(6)` `data = 5` `k = -1`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
▶      if (root == null)
3          return false;              // not found
▶      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```



root ≠ null. **k = compareTo = 1** since 5 > 4.

Step 3 / 20

CALL STACK

- **search(4, 5)**
- search(6, 5)
- main()

VARIABLES

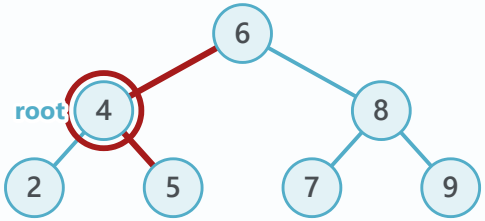
`root = node(4)` `data = 5` `k = 1`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;              // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
▶      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- **search(4, 5)**
- search(6, 5)
- main()



k > 0 → go right: `search(root.rightChild, 5)` opens a new frame (root = 5).

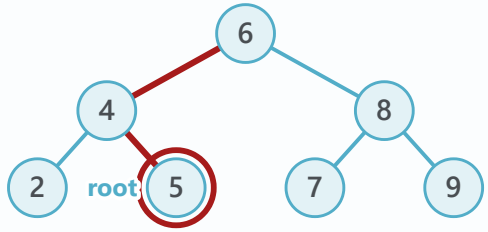
Step 4 / 20

VARIABLES

`root = node(4)` `data = 5` `k = 1`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
▶      if (root == null)
3          return false;              // not found
▶      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```



root ≠ null. **k = compareTo = 0** since 5 = 5.

Step 5 / 20

CALL STACK

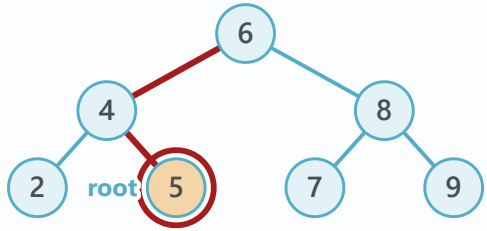
- **search(5, 5)**
- search(4, 5)
- search(6, 5)
- main()

VARIABLES

`root = node(5)` `data = 5` `k = 0`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;              // not found
4      int k = data.compareTo(root.data);
▶      if (k == 0)
▶          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```



k == 0 → key found: **return true**. This frame closes.

Step 6 / 20

CALL STACK

- **search(5, 5)**
- search(4, 5)
- search(6, 5)
- main()

VARIABLES

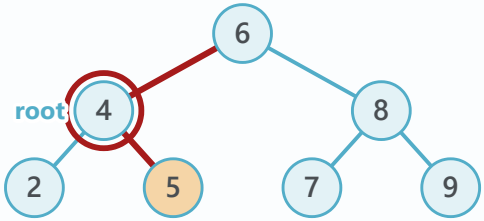
`root` = **node(5)** | `data` = **5** | `k` = **0**

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;              // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- **search(4, 5)**
- search(6, 5)
- main()



`search(5, 5)` returned **true**; the frame at 4 simply **return**s it.

Step 7 / 20

VARIABLES

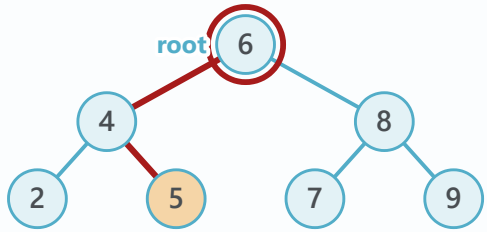
`root = node(4)` `data = 5` `k = 1` `returned = true`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;            // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;             // found
7      else if (k < 0)
▶          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```

CALL STACK

- `search(6, 5)`
- `main()`



`search(4, 5)` returned **true**; the frame at 6 simply **return**s it.

Step 8 / 20

VARIABLES

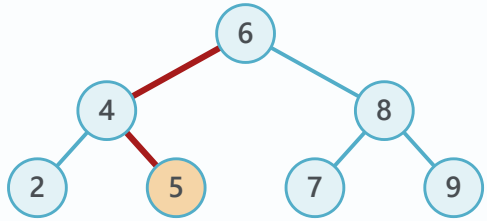
`root = node(6)` `data = 5` `k = -1` `returned = true`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
boolean search(BinaryTreeNode<E> root, E data) {
    if (root == null)
        return false;            // not found
    int k = data.compareTo(root.data);
    if (k == 0)
        return true;             // found
    else if (k < 0)
        return search(root.leftChild, data);
    else
        return search(root.rightChild, data);
}
```

CALL STACK

`main()`



main gets **true**. 3 nodes compared = depth of the found node + 1.

Step 9 / 20

VARIABLES

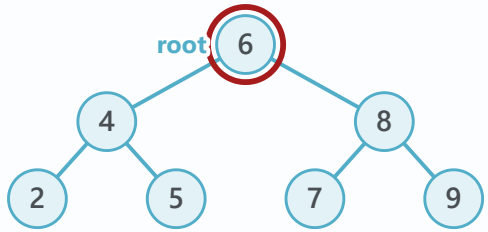
`result = true`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
▶      if (root == null)
3          return false;              // not found
▶      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```

CALL STACK

- `search(6, 10)`
- `main()`



`search(root, 10)` : root is node 6, not null. **k = compareTo = 1** since 10 > 6.

Step 10 / 20

VARIABLES

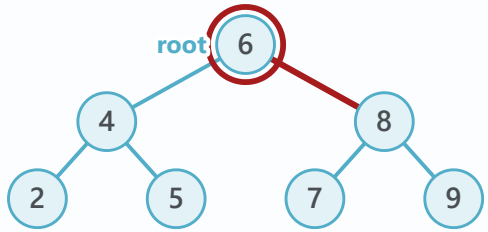
`root = node(6)` `data = 10` `k = 1`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;              // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
▶      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- `search(6, 10)`
- `main()`



k > 0 → go right: `search(root.rightChild, 10)` opens a new frame (root = 8).

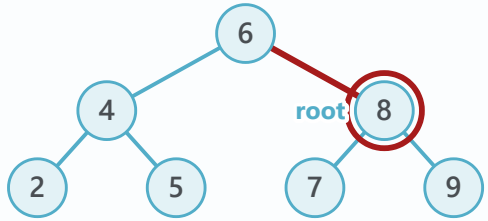
Step 11 / 20

VARIABLES

`root = node(6)` `data = 10` `k = 1`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
▶      if (root == null)
3          return false;              // not found
▶      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```



root ≠ null. **k = compareTo = 1** since 10 > 8.

Step 12 / 20

CALL STACK

- **search(8, 10)**
- search(6, 10)
- main()

VARIABLES

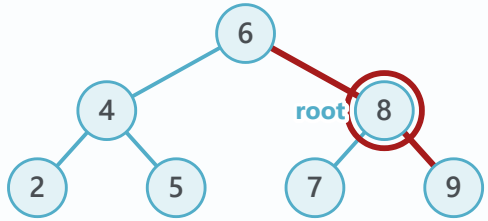
- root = **node(8)**
- data = **10**
- k = **1**

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;              // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
▶      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- **search(8, 10)**
- search(6, 10)
- main()



k > 0 → go right: `search(root.rightChild, 10)` opens a new frame (root = 9).

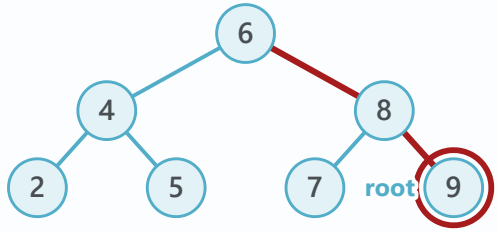
Step 13 / 20

VARIABLES

`root` = **node(8)** `data` = **10** `k` = **1**

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
▶      if (root == null)
3          return false;              // not found
▶      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```



root ≠ null. **k = compareTo = 1** since 10 > 9.

Step 14 / 20

CALL STACK

- **search(9, 10)**
- search(8, 10)
- search(6, 10)
- main()

VARIABLES

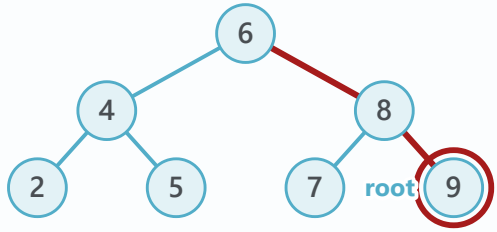
`root = node(9)` `data = 10` `k = 1`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;              // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
▶      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- **search(9, 10)**
- search(8, 10)
- search(6, 10)
- main()



k > 0 → go right: `search(root.rightChild, 10)` opens a new frame (root = null).

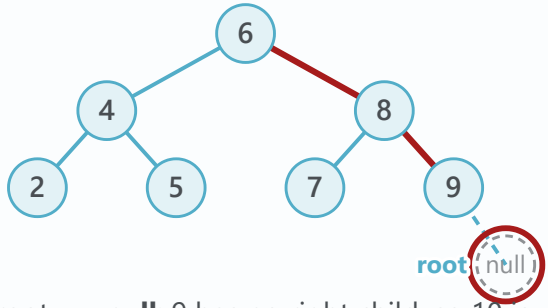
Step 15 / 20

VARIABLES

`root` = **node(9)** `data` = **10** `k` = **1**

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
▶      if (root == null)
▶          return false;              // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;               // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
10         return search(root.rightChild, data);
11 }
```



root == **null**: 9 has no right child, so 10 is **not** in the tree → **return false**.

CALL STACK

- **search(null, 10)**
- search(9, 10)
- search(8, 10)
- search(6, 10)
- main()

Step 16 / 20

VARIABLES

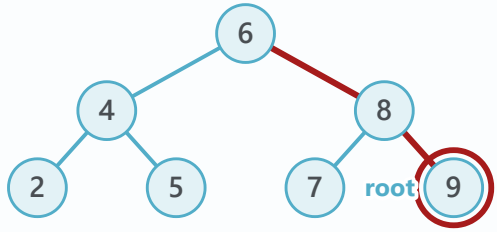
`root = null` `data = 10`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;            // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;             // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- **search(9, 10)**
- search(8, 10)
- search(6, 10)
- main()



`search(null, 10)` returned **false**; the frame at 9 simply **return**s it.

Step 17 / 20

VARIABLES

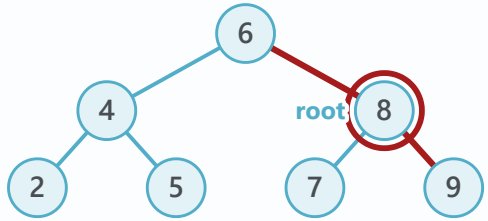
`root = node(9)` `data = 10` `k = 1` `returned = false`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;            // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;             // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- **search(8, 10)**
- search(6, 10)
- main()



`search(9, 10)` returned **false**; the frame at 8 simply **return**s it.

Step 18 / 20

VARIABLES

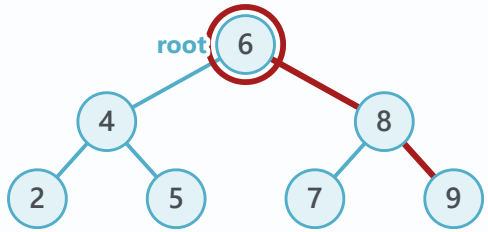
`root = node(8)` `data = 10` `k = 1` `returned = false`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
1  boolean search(BinaryTreeNode<E> root, E data) {
2      if (root == null)
3          return false;            // not found
4      int k = data.compareTo(root.data);
5      if (k == 0)
6          return true;             // found
7      else if (k < 0)
8          return search(root.leftChild, data);
9      else
▶          return search(root.rightChild, data);
11 }
```

CALL STACK

- `search(6, 10)`
- `main()`



`search(8, 10)` returned **false**; the frame at 6 simply **return**s it.

Step 19 / 20

VARIABLES

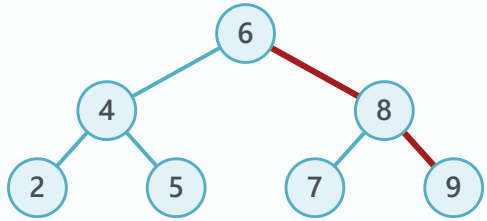
`root = node(6)` `data = 10` `k = 1` `returned = false`

Code Trace: `search(root, 5)` and `search(root, 10)`

```
▶ boolean search(BinaryTreeNode<E> root, E data) {
2     if (root == null)
3         return false;            // not found
4     int k = data.compareTo(root.data);
5     if (k == 0)
6         return true;             // found
7     else if (k < 0)
8         return search(root.leftChild, data);
9     else
10        return search(root.rightChild, data);
11 }
```

CALL STACK

`main()`



Result: **false**, after 3 nodes. At most h + 1 nodes → **O(h)**.

Step 20 / 20

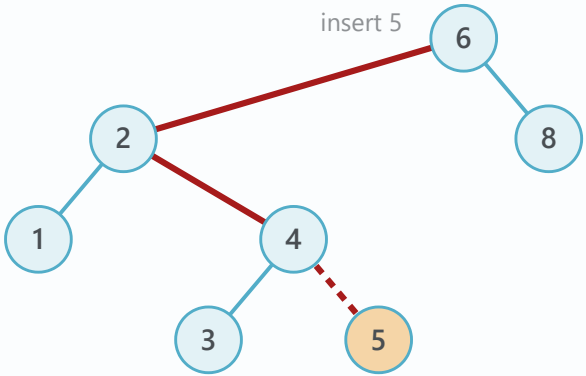
VARIABLES

`result = false`

BST OPERATIONS

Inserting a Key

- Descend as if searching; put the new leaf in the **null** spot where you land.
- The new node is always a **leaf**; O(h).




```
BinaryTreeNode<E> insert(
        BinaryTreeNode<E> root, E data) {
    if (root == null)
        return new BinaryTreeNode<>(data);
    int k = data.compareTo(root.data);
    if (k < 0)
        root.leftChild = insert(root.leftChild, data);
    else if (k > 0)
        root.rightChild = insert(root.rightChild, data)
    // k == 0: key already present, skip it
    return root;
}
// usage: root = insert(root, 5);
```

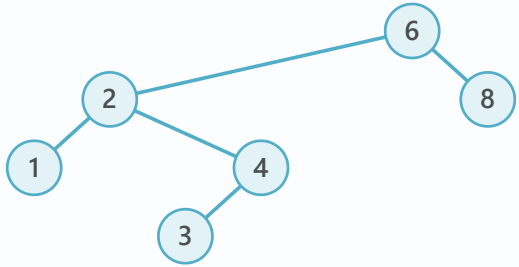
The recursive call's result is **assigned back to the child link**, so the new node replaces the null link.

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
▶  // usage: root = insert(root, 5);
```

CALL STACK

`main()`



main calls `root = insert(root, 5)`. Like search, insertion starts at the root.

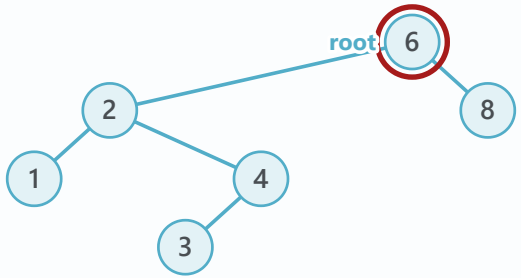
Step 1 / 13

VARIABLES

`data = 5`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null)
4          return new BinaryTreeNode<>(data);
▶      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```



root ≠ null (6). **k = -1** since 5 < 6.

Step 2 / 13

CALL STACK

- **insert(6, 5)**
- main()

VARIABLES

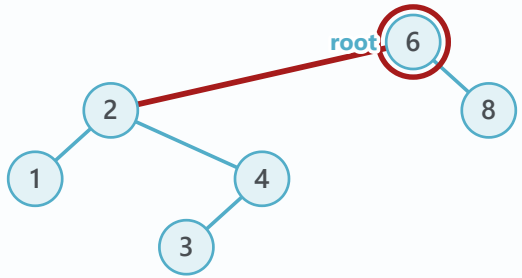
`root = node(6)` `data = 5` `k = −1`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
▶      if (k < 0)
▶          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```

CALL STACK

- **`insert(6, 5)`**
- `main()`



Go left: `insert(root.leftChild, 5)`; root = 2. The result will go into **root.leftChild**.

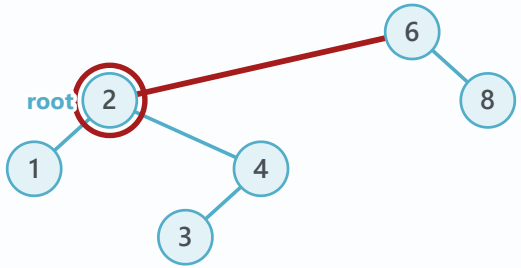
Step 3 / 13

VARIABLES

`root = node(6)` `data = 5` `k = −1`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null)
4          return new BinaryTreeNode<>(data);
▶      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```



root ≠ null (2). **k = 1** since 5 > 2.

Step 4 / 13

CALL STACK

- **insert(2, 5)**
- insert(6, 5)
- main()

VARIABLES

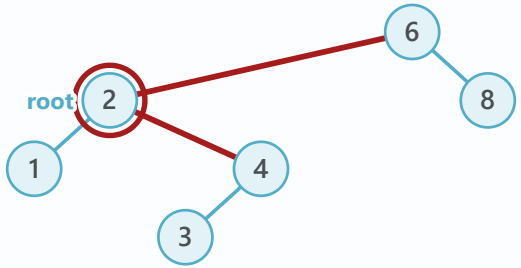
`root = node(2)` `data = 5` `k = 1`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
▶      else if (k > 0)
▶          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```

CALL STACK

- **`insert(2, 5)`**
- `insert(6, 5)`
- `main()`



Go right: `insert(root.rightChild, 5)`; root = 4. The result will go into **root.rightChild**.

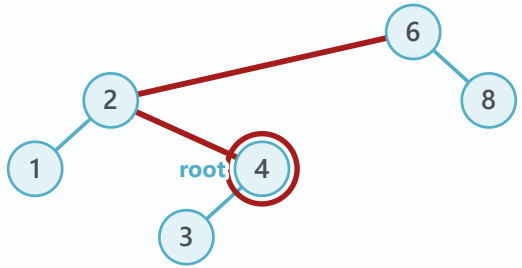
Step 5 / 13

VARIABLES

`root = node(2)` `data = 5` `k = 1`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null)
4          return new BinaryTreeNode<>(data);
▶      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```



root ≠ null (4). **k = 1** since 5 > 4.

Step 6 / 13

CALL STACK

- **insert(4, 5)**
- insert(2, 5)
- insert(6, 5)
- main()

VARIABLES

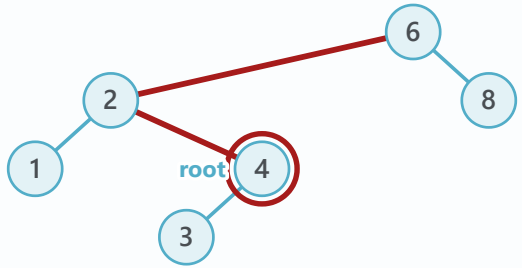
`root = node(4)` `data = 5` `k = 1`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
▶      else if (k > 0)
▶          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```

CALL STACK

- **insert(4, 5)**
- insert(2, 5)
- insert(6, 5)
- main()



Go right: `insert(root.rightChild, 5)`; root = null. The result will go into **root.rightChild**.

Step 7 / 13

VARIABLES

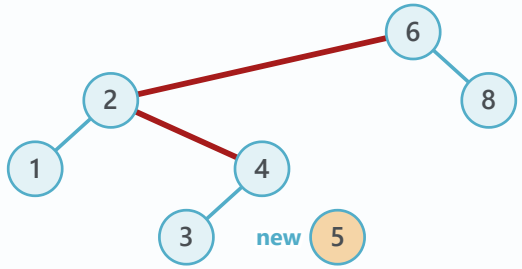
`root = node(4)` `data = 5` `k = 1`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null)
▶          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```

CALL STACK

- **insert(null, 5)**
- insert(4, 5)
- insert(2, 5)
- insert(6, 5)
- main()



root == **null**: an empty spot. `new` creates node 5 and **returns** it; no link points to it yet.

Step 8 / 13

VARIABLES

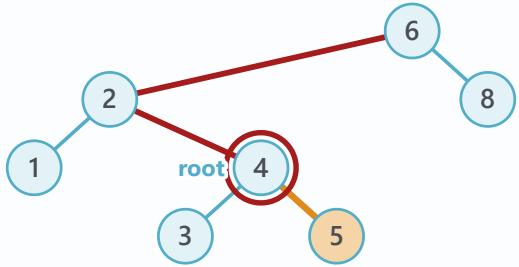
`root = null` `data = 5`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
▶          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
13 // usage: root = insert(root, 5);
```

CALL STACK

- **insert(4, 5)**
- insert(2, 5)
- insert(6, 5)
- main()



Back in 4's frame: **root.rightChild = node(5)**. The new node replaces the null link (orange).

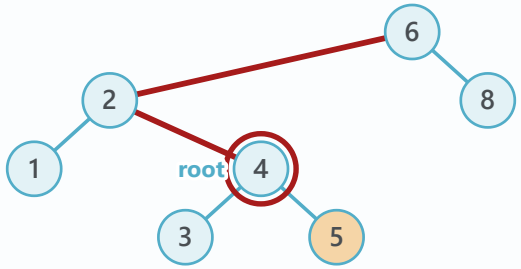
Step 9 / 13

VARIABLES

`root = node(4)` `data = 5` `k = 1` `returned = node(5)`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
▶      return root;
12 }
13 // usage: root = insert(root, 5);
```



return root: 4 goes back to its parent (2).

Step 10 / 13

CALL STACK

- **insert(4, 5)**
- insert(2, 5)
- insert(6, 5)
- main()

VARIABLES

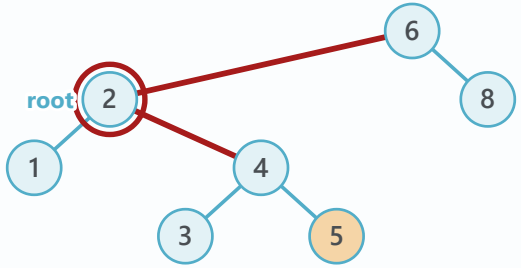
`root` = **node(4)** | `data` = **5** | `k` = **1**

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
▶          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
▶      return root;
12 }
13 // usage: root = insert(root, 5);
```

CALL STACK

- **`insert(2, 5)`**
- `insert(6, 5)`
- `main()`



At 2, `root.rightChild = node(4)` : same link, **no change**. Then return 2.

Step 11 / 13

VARIABLES

`root = node(2)` `data = 5` `k = 1`

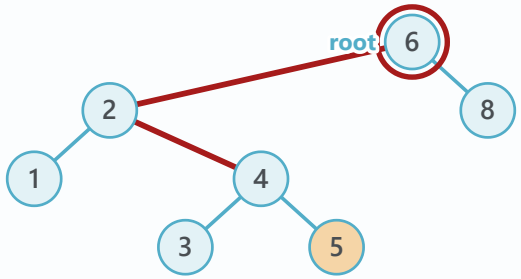
`returned = node(4)`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
▶          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
▶      return root;
12 }
13 // usage: root = insert(root, 5);
```

CALL STACK

- **insert(6, 5)**
- main()



At 6, `root.leftChild = node(2)` (the same link), then **return 6**.

Step 12 / 13

VARIABLES

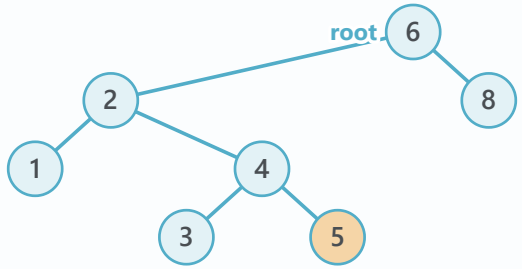
`root = node(6)` `data = 5` `k = −1` `returned = node(2)`

Code Trace: `insert(root, 5)`

```
1  BinaryTreeNode<E> insert(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null)
4          return new BinaryTreeNode<>(data);
5      int k = data.compareTo(root.data);
6      if (k < 0)
7          root.leftChild = insert(root.leftChild, data);
8      else if (k > 0)
9          root.rightChild = insert(root.rightChild, data);
10     // k == 0: key already present, skip it
11     return root;
12 }
▶ // usage: root = insert(root, 5);
```

CALL STACK

`main()`



main: `root = 6` (unchanged). A new node is always a **leaf**; one call per node on the path → **O(h)**.

Step 13 / 13

VARIABLES

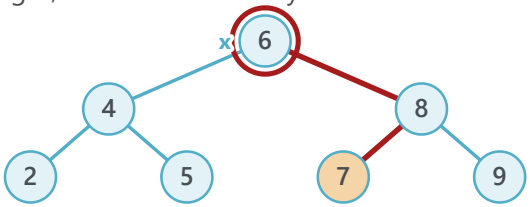
`root = node(6)`

BST OPERATIONS

Successor and Predecessor

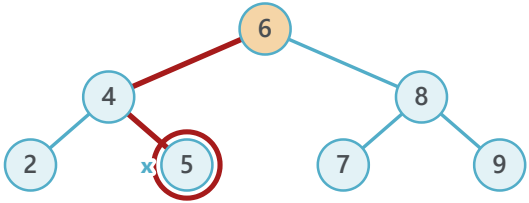
1) With a right subtree

successor = the **smallest key** in the right subtree: one step right, then left all the way.



2) With no right subtree

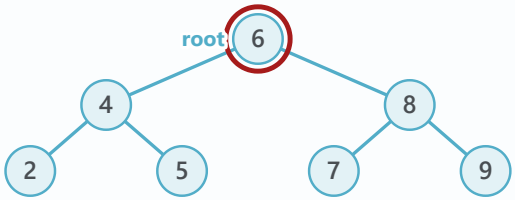
successor = the nearest ancestor **we reach from its left** (5 → 4 → 6: 5 is in 6's left subtree).



Operation	Meaning	Java `TreeMap`	In the tree above
successor	smallest key > x	`higherKey(x)`	successor(5) = 6
predecessor	largest key < x	`lowerKey(x)`	predecessor(6) = 5
ceiling	smallest key ≥ x	`ceilingKey(x)`	ceiling(3) = 4
floor	largest key ≤ x	`floorKey(x)`	floor(3) = 2

Each follows one path from the root: **O(h)**. A deleted node with two children is replaced by its successor.

Code Trace: `successor(root, 5)` and `successor(root, 6)`



`successor(root, 5)` : no candidate yet (`candidate = null`); start at the root (6).

Step 1 / 10

```
▶ TreeNode successor(TreeNode root, int x) {
▶     TreeNode candidate = null; // smallest key above x
3     while (root != null) {
4         if (x < root.data) {
5             candidate = root;  // root is a candidate
6             root = root.left;  // smaller ones are left
7         } else
8             root = root.right; // keys above x are right
9     }
10    return candidate;
11 }
```

VARIABLES

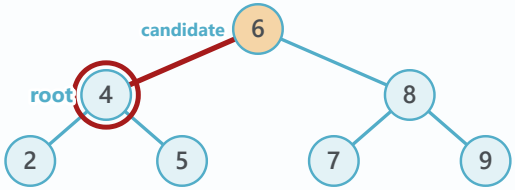
`x = 5` `root = node(6)` `candidate = null`

CALL STACK

`successor(root, 5)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



5 < 6 → **6 is a candidate** (> x); anything smaller is on the left → go left.

Step 2 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
▶          if (x < root.data) {
▶              candidate = root;  // root is a candidate
▶              root = root.left;  // smaller ones are left
7          } else
8              root = root.right; // keys above x are right
9      }
10     return candidate;
11 }
```

VARIABLES

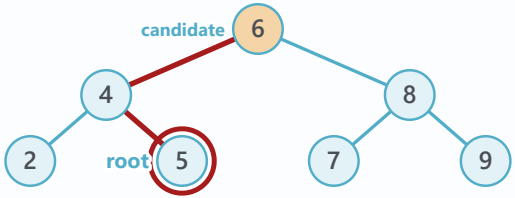
`x = 5` `root = node(4)` `candidate = node(6)`

CALL STACK

`successor(root, 5)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



5 ≥ 4 → 4 and everything to its left are ruled out; larger keys are on the **right** → go right.

Step 3 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
▶          if (x < root.data) {
5              candidate = root;  // root is a candidate
6              root = root.left;  // smaller ones are left
▶          } else
▶              root = root.right; // keys above x are right
9      }
10     return candidate;
11 }
```

VARIABLES

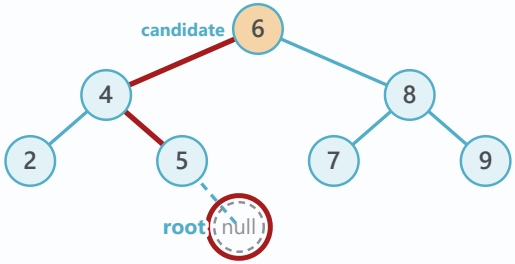
`x = 5` `root = node(5)` `candidate = node(6)`

CALL STACK

`successor(root, 5)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



5 ≥ 5 → 5 and everything to its left are ruled out; larger keys are on the **right** → go right. Now `root = null`.

Step 4 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
▶          if (x < root.data) {
5              candidate = root;  // root is a candidate
6              root = root.left;  // smaller ones are left
▶          } else
▶              root = root.right; // keys above x are right
9      }
10     return candidate;
11 }
```

VARIABLES

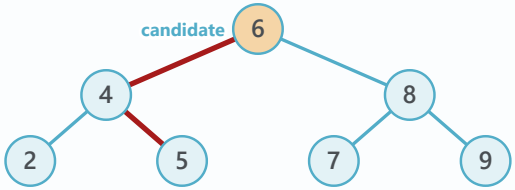
`x = 5` `root = null` `candidate = node(6)`

CALL STACK

`successor(root, 5)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



The loop ends (`root == null`) → **return node(6)**: successor(5) = **6**.

Step 5 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
4          if (x < root.data) {
5              candidate = root;  // root is a candidate
6              root = root.left;  // smaller ones are left
7          } else
8              root = root.right; // keys above x are right
9      }
▶      return candidate;
11 }
```

VARIABLES

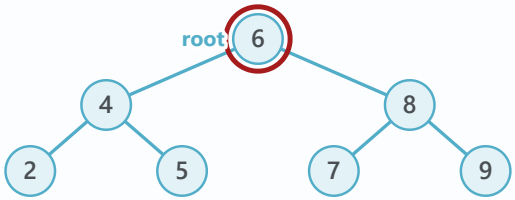
`x = 5` `root = null` `candidate = node(6)`

CALL STACK

`successor(root, 5)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



Second call, `successor(root, 6)`: no candidate yet (`candidate = null`); start at the root (6).

Step 6 / 10

```
▶ TreeNode successor(TreeNode root, int x) {
▶     TreeNode candidate = null; // smallest key above x
3     while (root != null) {
4         if (x < root.data) {
5             candidate = root;  // root is a candidate
6             root = root.left;  // smaller ones are left
7         } else
8             root = root.right; // keys above x are right
9     }
10    return candidate;
11 }
```

VARIABLES

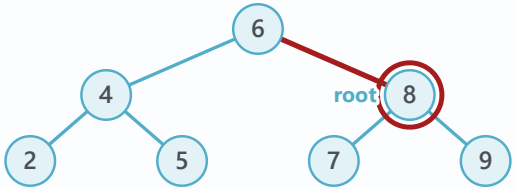
`x = 6` `root = node(6)` `candidate = null`

CALL STACK

`successor(root, 6)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



6 ≥ 6 → 6 and everything to its left are ruled out; larger keys are on the **right** → go right.

Step 7 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
▶          if (x < root.data) {
5              candidate = root;  // root is a candidate
6              root = root.left;  // smaller ones are left
▶          } else
▶              root = root.right; // keys above x are right
9      }
10     return candidate;
11 }
```

VARIABLES

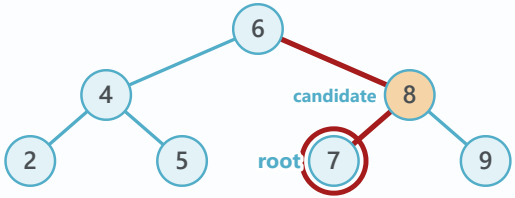
`x = 6` `root = node(8)` `candidate = null`

CALL STACK

`successor(root, 6)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



6 < 8 → **8 is a candidate** (> x); anything smaller is on the left → go left.

Step 8 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
▶          if (x < root.data) {
▶              candidate = root;  // root is a candidate
▶              root = root.left;  // smaller ones are left
7          } else
8              root = root.right; // keys above x are right
9      }
10     return candidate;
11 }
```

VARIABLES

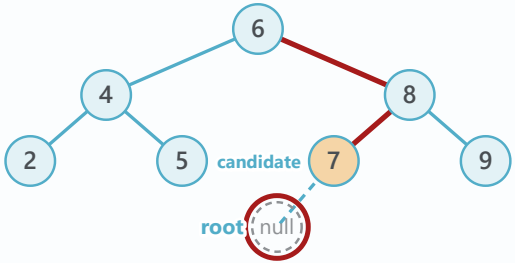
`x = 6` `root = node(7)` `candidate = node(8)`

CALL STACK

`successor(root, 6)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



6 < 7 → **7 is a candidate** (> x); anything smaller is on the left → go left. Now `root = null`.

Step 9 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
▶          if (x < root.data) {
▶              candidate = root;  // root is a candidate
▶              root = root.left;  // smaller ones are left
7          } else
8              root = root.right; // keys above x are right
9      }
10     return candidate;
11 }
```

VARIABLES

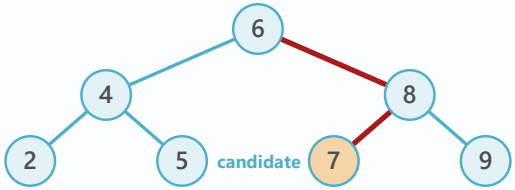
`x = 6` `root = null` `candidate = node(7)`

CALL STACK

`successor(root, 6)`

`main()`

Code Trace: `successor(root, 5)` and `successor(root, 6)`



The loop ends (`root == null`) → **return node(7)**: successor(6) = **7**.

Step 10 / 10

```
1  TreeNode successor(TreeNode root, int x) {
2      TreeNode candidate = null; // smallest key above x
▶      while (root != null) {
4          if (x < root.data) {
5              candidate = root;  // root is a candidate
6              root = root.left;  // smaller ones are left
7          } else
8              root = root.right; // keys above x are right
9      }
▶      return candidate;
11 }
```

VARIABLES

`x = 6` `root = null` `candidate = node(7)`

CALL STACK

`successor(root, 6)`

`main()`

BST OPERATIONS

Deleting a Key: Three Cases

First find the node (O(h)). Then, by number of children:

1. Leaf

No children. Set the parent's link to `null`.

`delete 70`

2. One child

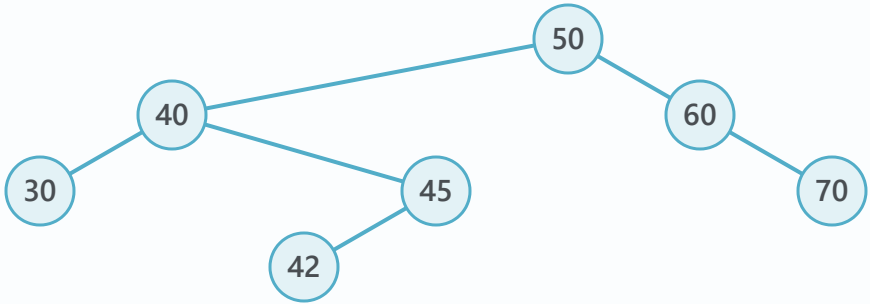
Bypass the node: link the parent straight to the only child.

`delete 45 · delete 60`

3. Two children

Replace the value with the inorder successor (smallest on the right) or predecessor (largest on the left).

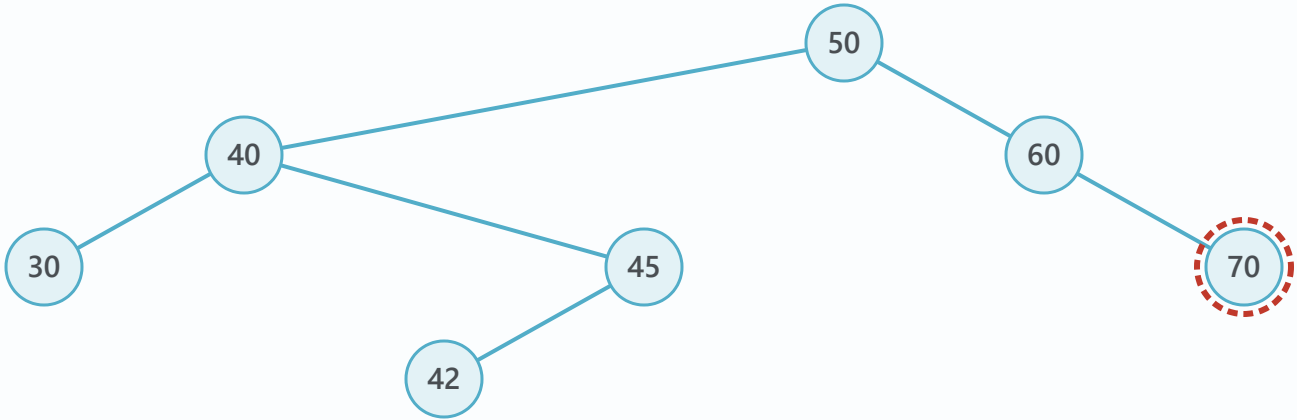
`delete 40`



Example tree

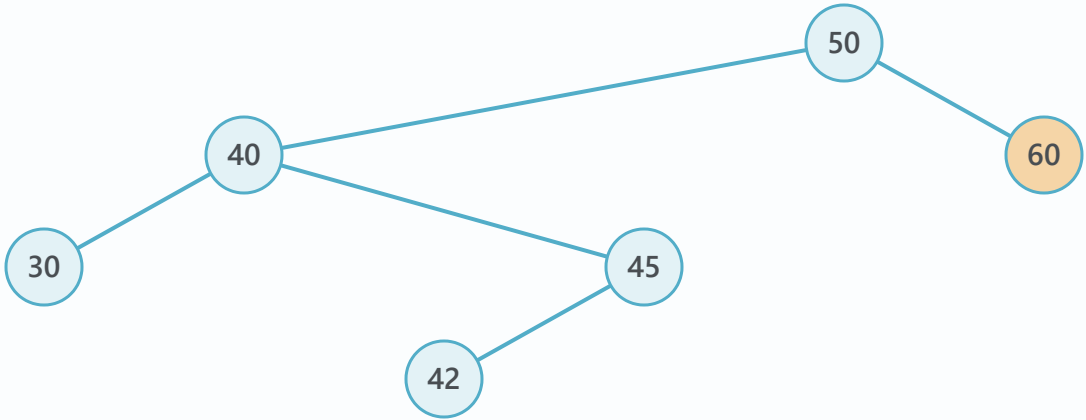
BST OPERATIONS

Deletion: A Leaf or One Child



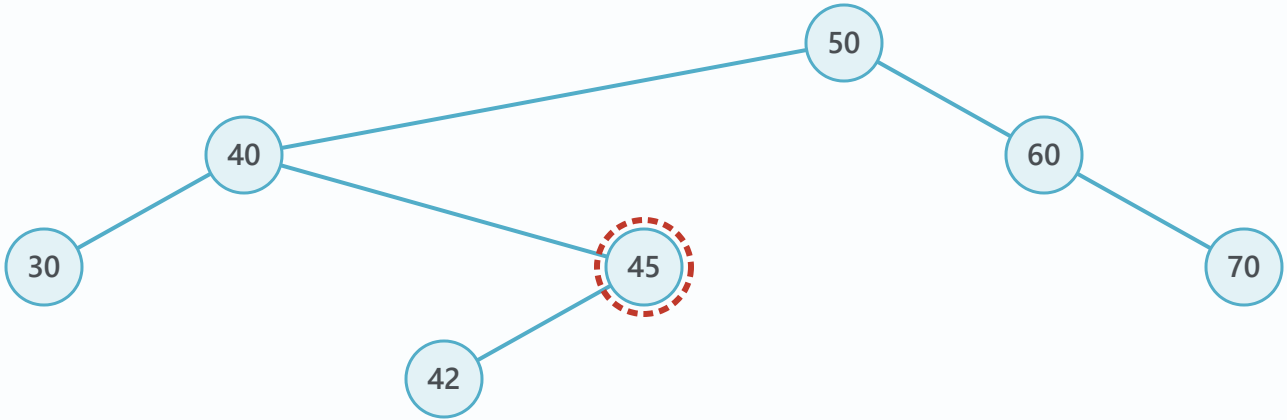
Leaf: 70 has no children. Its parent, 60, sets its right link to null.

Deletion: A Leaf or One Child



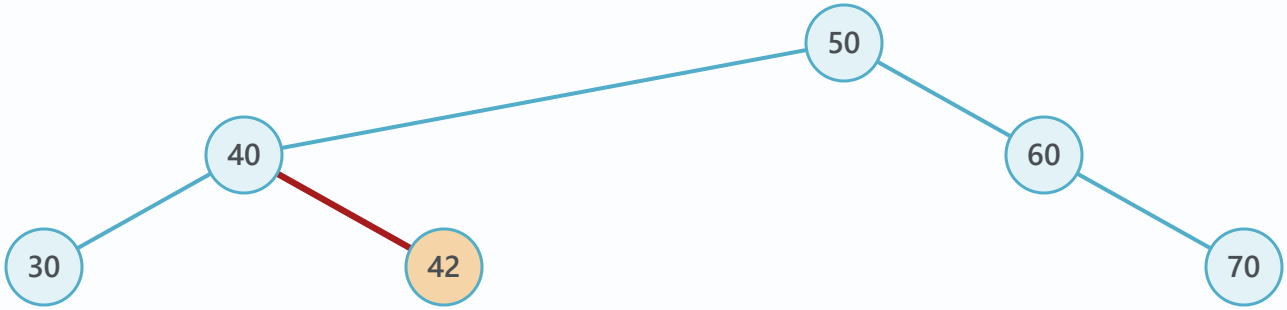
70 is gone; the rest of the tree is unchanged.

Deletion: A Leaf or One Child



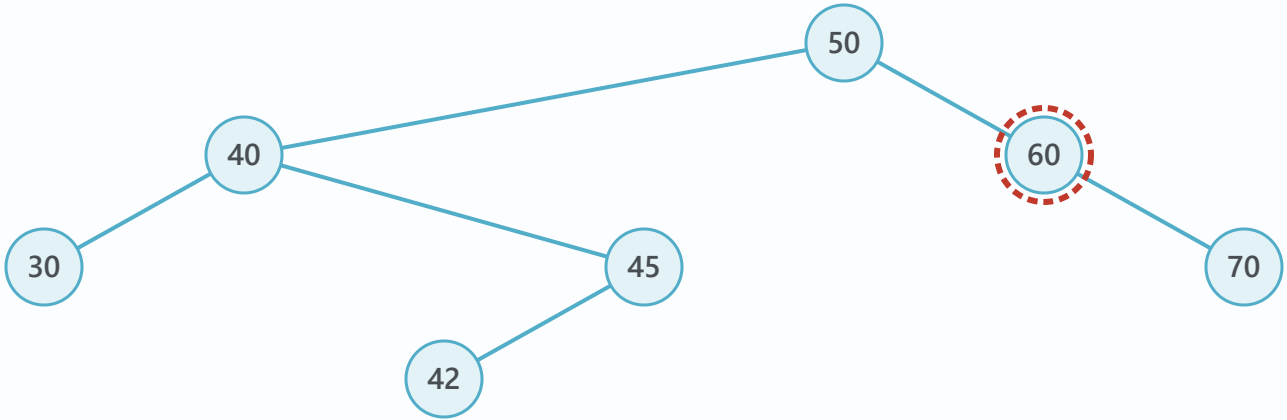
One child (left): 45 has only a left child, 42. (We start again from the original tree.)

Deletion: A Leaf or One Child



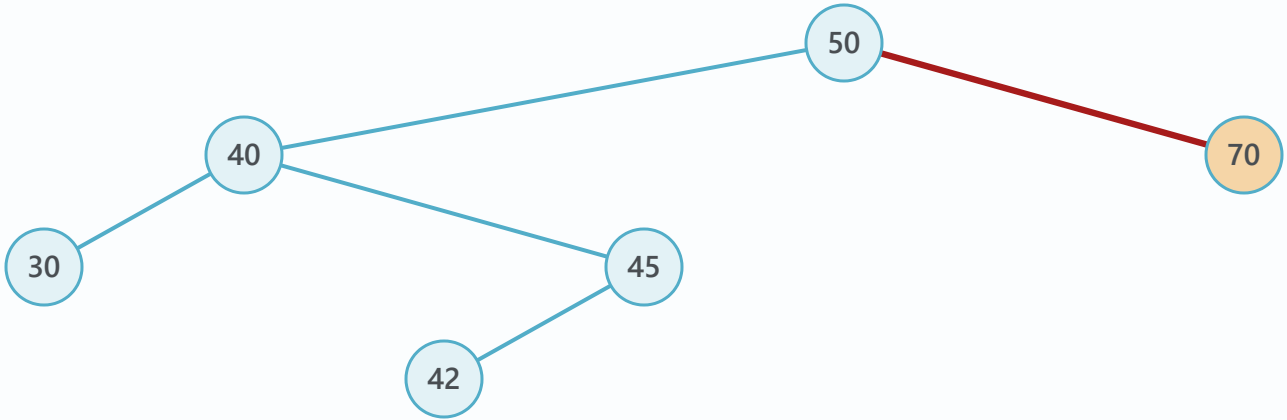
42 takes 45's place as 40's right child: the parent links straight to its grandchild.

Deletion: A Leaf or One Child



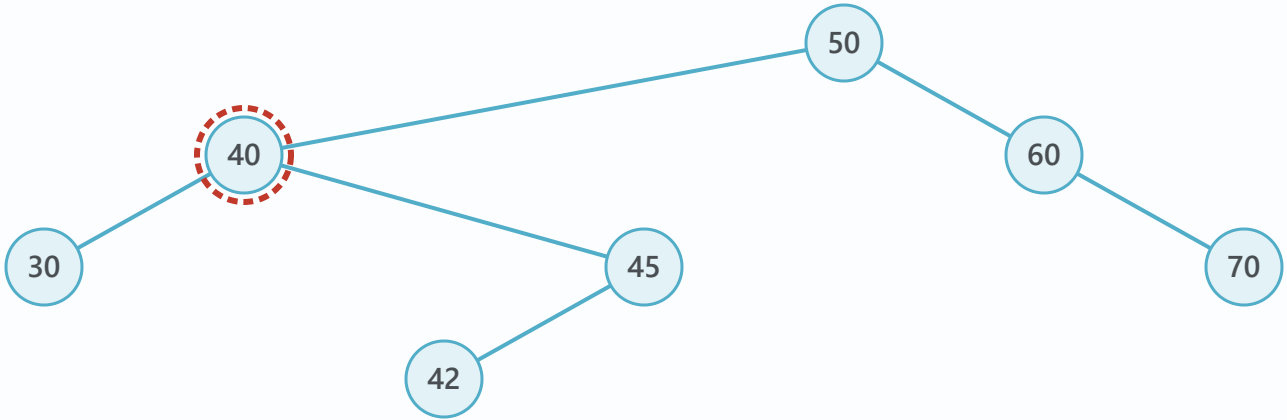
One child (right): 60 has only a right child, 70.

Deletion: A Leaf or One Child



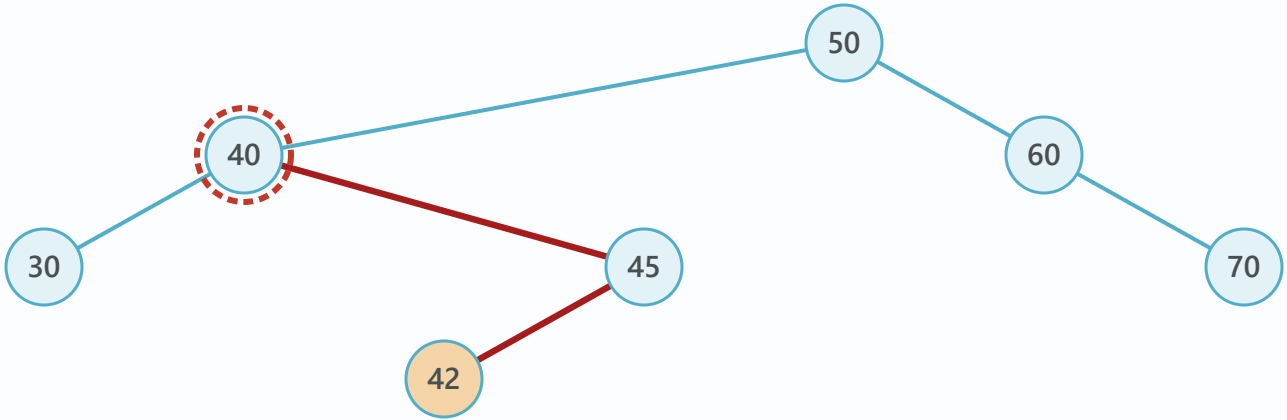
70 takes 60's place as 50's right child. The order is preserved.

Deletion: A Node with Two Children



Two children: 40 has a left (30) and a right (45) child, so it cannot simply be removed.

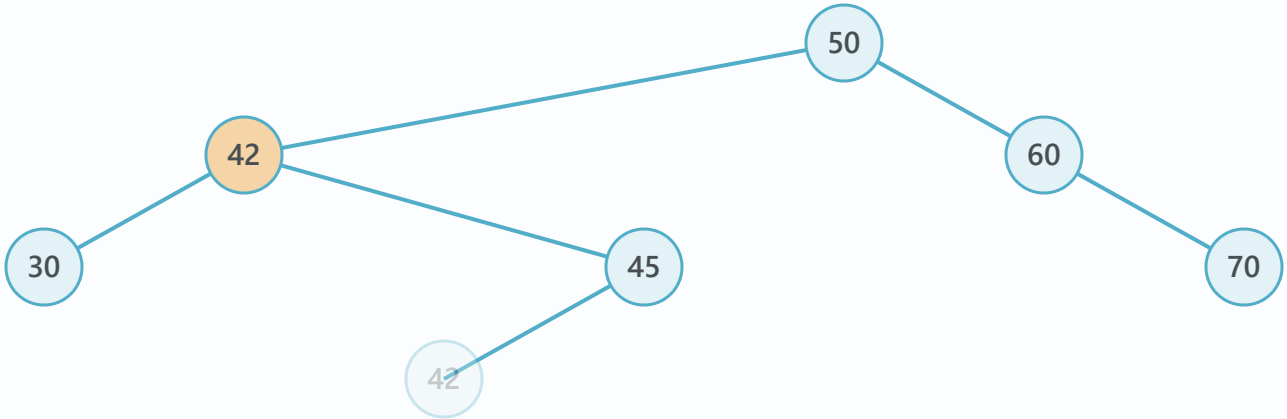
Deletion: A Node with Two Children



Inorder successor = smallest key in the right subtree: step to 45, then left all the way → 42.

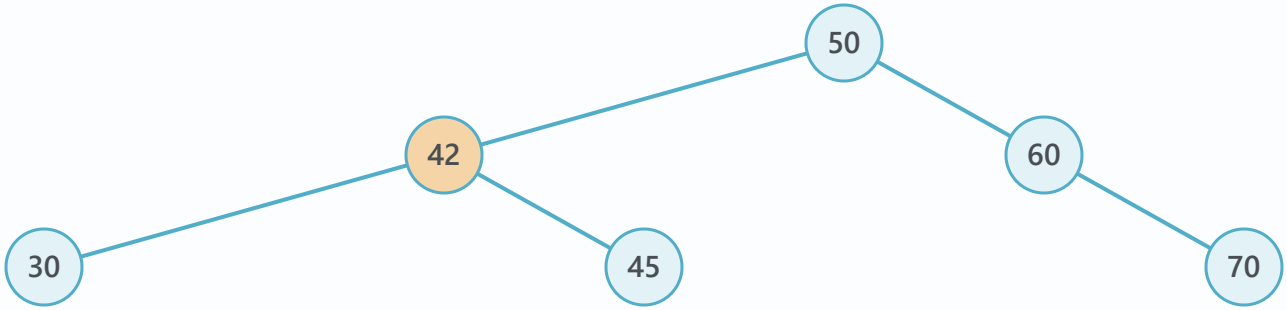
BST OPERATIONS

Deletion: A Node with Two Children



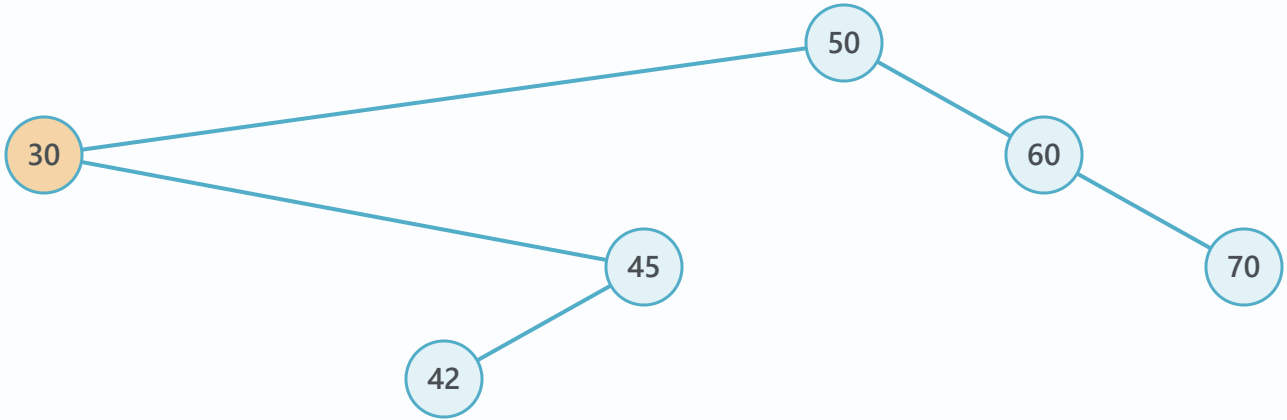
The successor's value (42) is copied into 40's node. The old 42 is now a duplicate.

Deletion: A Node with Two Children



Delete the old 42 from the right subtree; the successor has no left child, so this is always Case 1 or 2.

Deletion: A Node with Two Children



Alternatively, use the **inorder predecessor** (largest key on the left = 30). Both give a valid BST.

BST OPERATIONS

The Deletion Code

```
BinaryTreeNode<E> delete(
        BinaryTreeNode<E> root, E data) {
    if (root == null) return null;   // not found
    int k = data.compareTo(root.data);
    if (k < 0)
        root.leftChild = delete(root.leftChild, data);
    else if (k > 0)
        root.rightChild = delete(root.rightChild, data);
    else if (root.leftChild == null)
        return root.rightChild;  // leaf / right child only
    else if (root.rightChild == null)
        return root.leftChild;   // left child only
    else {                       // two children
        root.data = minValue(root.rightChild).data;
        root.rightChild = delete(root.rightChild, root.data);
    }
    return root;
}
```

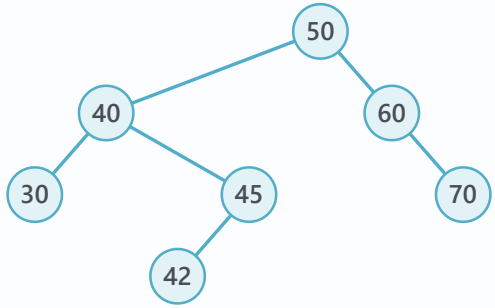
The return value goes into the parent's link: returning the only child or null cuts the node out. Usage: `root = delete(root, 40);` (if the root is deleted, this assignment installs the new one).

Code Trace: `delete(root, 70)` — a Leaf

```
▶ BinaryTreeNode<E> delete(
2         BinaryTreeNode<E> root, E data) {
3     if (root == null) return null;   // not found
4     int k = data.compareTo(root.data);
5     if (k < 0)
6         root.leftChild = delete(root.leftChild, data);
7     else if (k > 0)
8         root.rightChild = delete(root.rightChild, data);
9     else if (root.leftChild == null)
10        return root.rightChild;  // leaf / right child only
11    else if (root.rightChild == null)
12        return root.leftChild;   // left child only
13    else {                       // two children
14        root.data = minValue(root.rightChild).data;
15        root.rightChild = delete(root.rightChild, root.data);
16    }
17    return root;
18 }
```

VARIABLES

`data = 70`



main calls `root = delete(root, 70)`. First, find the node.

Step 1 / 11

CALL STACK

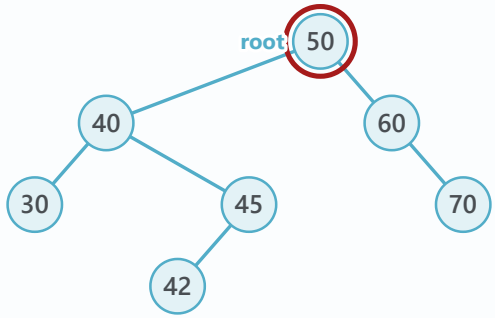
`main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)` `data = 70` `k = 1`



root ≠ null. **k = 1** since 70 > 50.

Step 2 / 11

CALL STACK

`delete(50, 70)`

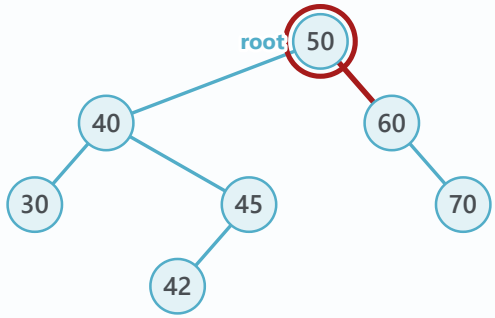
`main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
▶      else if (k > 0)
▶          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)` `data = 70` `k = 1`



Go right: `delete(root.rightChild, 70)`. The result will go into **root.rightChild**.

Step 3 / 11

CALL STACK

`delete(50, 70)`

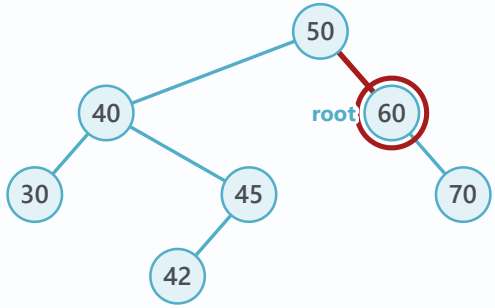
`main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(60)` `data = 70` `k = 1`



root ≠ null. **k = 1** since 70 > 60.

Step 4 / 11

CALL STACK

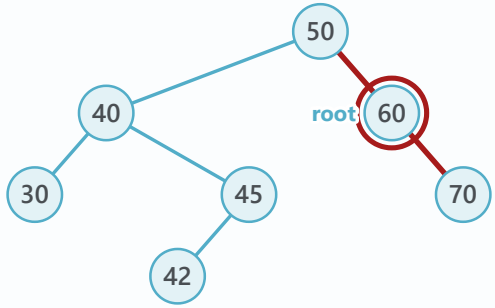
- `delete(60, 70)`
- `delete(50, 70)`
- `main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
▶      else if (k > 0)
▶          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(60)` `data = 70` `k = 1`



Go right: `delete(root.rightChild, 70)`. The result will go into **root.rightChild**.

Step 5 / 11

CALL STACK

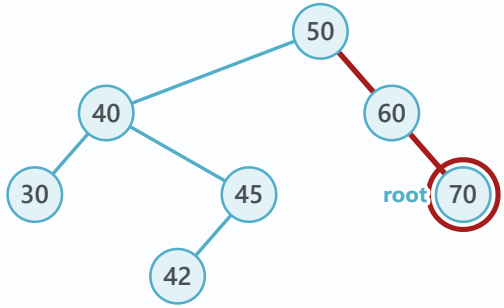
- `delete(60, 70)`
- `delete(50, 70)`
- `main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(70)` `data = 70` `k = 0`



root ≠ null. **k = 0** since 70 = 70.

Step 6 / 11

CALL STACK

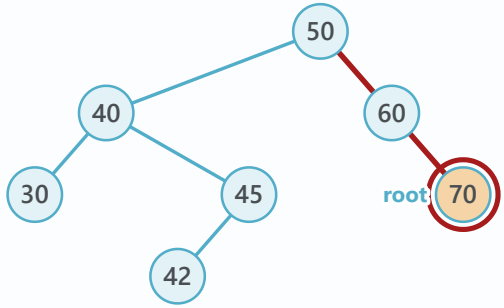
- `delete(70, 70)`
- `delete(60, 70)`
- `delete(50, 70)`
- `main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
▶      else if (root.leftChild == null)
▶          return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(70)` `data = 70` `k = 0`



k = 0 → found the node to delete. No left child → **return root.rightChild** = null. (A leaf returns null.)

Step 7 / 11

CALL STACK

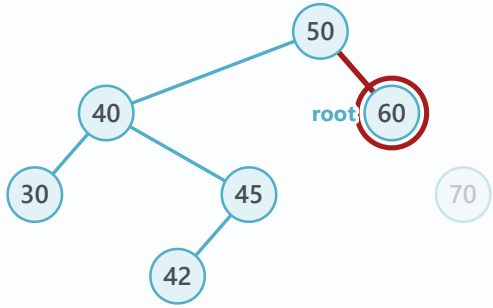
- `delete(70, 70)`
- `delete(60, 70)`
- `delete(50, 70)`
- `main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
▶          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(60)` `data = 70` `k = 1` `returned = null`



Changed link: **root.rightChild = null**. Nothing points to the deleted node now; the garbage collector reclaims it.

Step 8 / 11

CALL STACK

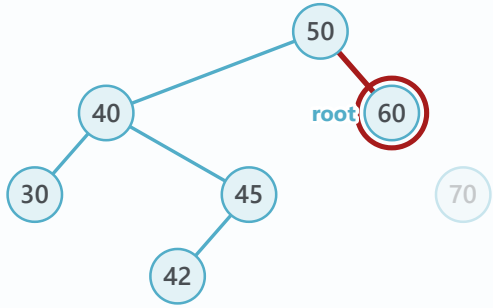
- `delete(60, 70)`
- `delete(50, 70)`
- `main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
▶      return root;
18 }
```

VARIABLES

`root = node(60)` `data = 70` `k = 1`



return root: 60 goes back to its parent.

Step 9 / 11

CALL STACK

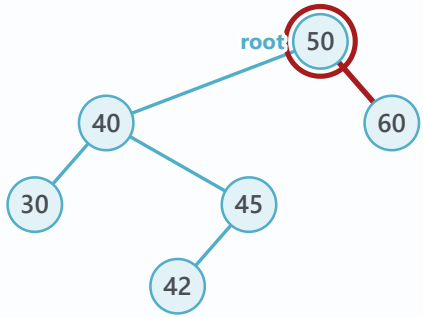
- `delete(60, 70)`
- `delete(50, 70)`
- `main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
▶          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
▶      return root;
18 }
```

VARIABLES

`root = node(50)` `data = 70` `k = 1` `returned = node(60)`



`root.rightChild = node(60)` : same link, **no change**. Then **return root** (50).

Step 10 / 11

CALL STACK

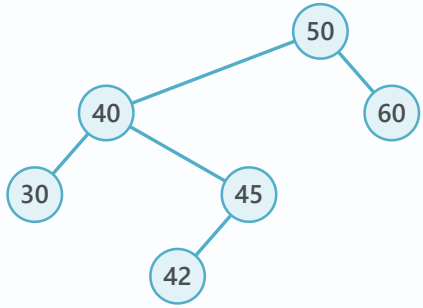
- `delete(50, 70)`
- `main()`

Code Trace: `delete(root, 70)` — a Leaf

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)`



main: `root = 50`. 70 was a leaf, so only 60's right link changed (to null).

Step 11 / 11

CALL STACK

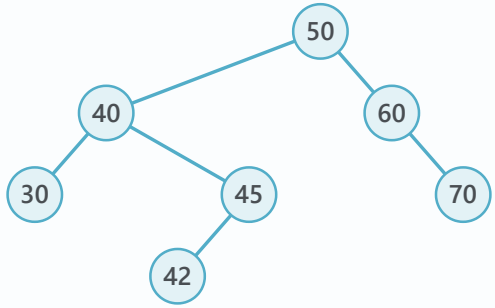
`main()`

Code Trace: `delete(root, 45)` — One Child

```
▶ BinaryTreeNode<E> delete(
2         BinaryTreeNode<E> root, E data) {
3     if (root == null) return null;   // not found
4     int k = data.compareTo(root.data);
5     if (k < 0)
6         root.leftChild = delete(root.leftChild, data);
7     else if (k > 0)
8         root.rightChild = delete(root.rightChild, data);
9     else if (root.leftChild == null)
10        return root.rightChild;  // leaf / right child only
11    else if (root.rightChild == null)
12        return root.leftChild;   // left child only
13    else {                       // two children
14        root.data = minValue(root.rightChild).data;
15        root.rightChild = delete(root.rightChild, root.data);
16    }
17    return root;
18 }
```

VARIABLES

`data = 45`



main calls `root = delete(root, 45)`. First, find the node.

Step 1 / 11

CALL STACK

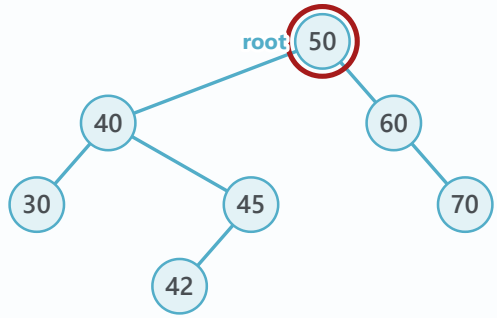
`main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)` `data = 45` `k = -1`



root ≠ null. **k = -1** since 45 < 50.

Step 2 / 11

CALL STACK

`delete(50, 45)`

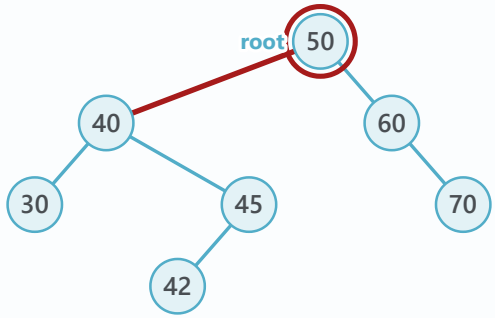
`main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
▶      if (k < 0)
▶          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)` `data = 45` `k = -1`



Go left: `delete(root.leftChild, 45)`. The result will go into **root.leftChild**.

Step 3 / 11

CALL STACK

`delete(50, 45)`

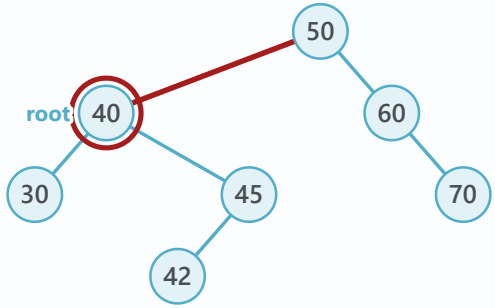
`main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(40)` `data = 45` `k = 1`



root ≠ null. **k = 1** since 45 > 40.

Step 4 / 11

CALL STACK

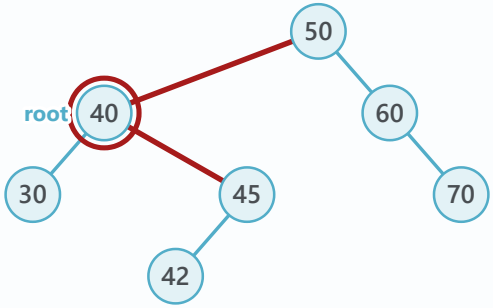
- `delete(40, 45)`
- `delete(50, 45)`
- `main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
▶      else if (k > 0)
▶          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(40)` `data = 45` `k = 1`



Go right: `delete(root.rightChild, 45)`. The result will go into **root.rightChild**.

CALL STACK

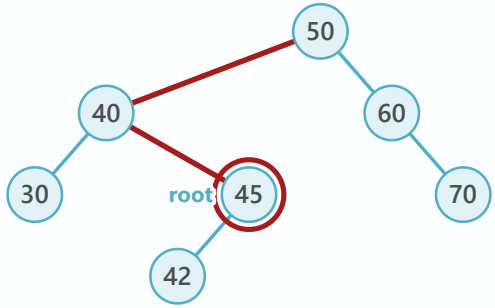
- `delete(40, 45)`
- `delete(50, 45)`
- `main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(45)` `data = 45` `k = 0`



root ≠ null. **k = 0** since 45 = 45.

Step 6 / 11

CALL STACK

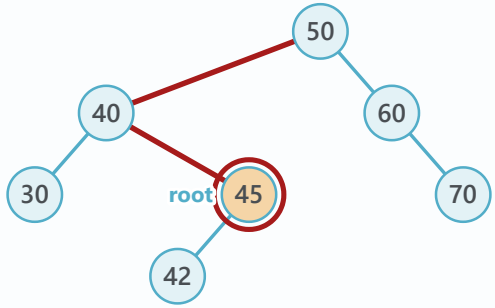
- `delete(45, 45)`
- `delete(40, 45)`
- `delete(50, 45)`
- `main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
▶      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
▶      else if (root.rightChild == null)
▶          return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(45)` `data = 45` `k = 0`



k = 0 → found. Left child, **no right child** → **return root.leftChild** = node(42): the only child goes to the parent.

Step 7 / 11

CALL STACK

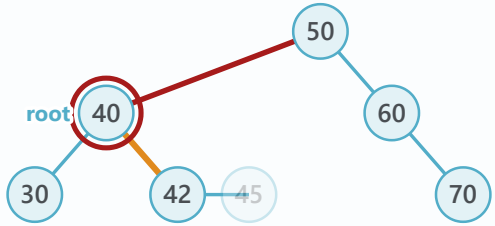
- `delete(45, 45)`
- `delete(40, 45)`
- `delete(50, 45)`
- `main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
▶          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(40)` `data = 45` `k = 1` `returned = node(42)`



Changed link: **root.rightChild = 42**. The parent bypasses the deleted node and links to its child.

Step 8 / 11

CALL STACK

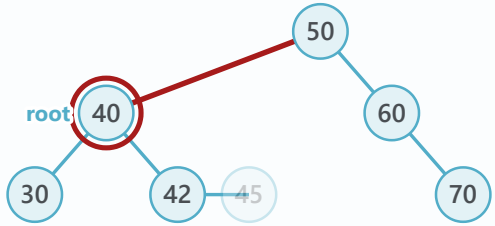
- `delete(40, 45)`
- `delete(50, 45)`
- `main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
▶      return root;
18 }
```

VARIABLES

`root = node(40)` `data = 45` `k = 1`



return root: 40 goes back to its parent.

Step 9 / 11

CALL STACK

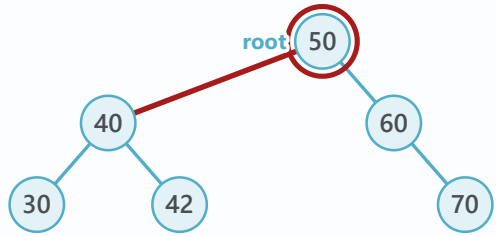
- `delete(40, 45)`
- `delete(50, 45)`
- `main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
▶          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
▶      return root;
18 }
```

VARIABLES

`root = node(50)` `data = 45` `k = -1` `returned = node(40)`



`root.leftChild = node(40)` : same link, **no change**. Then **return root** (50).

Step 10 / 11

CALL STACK

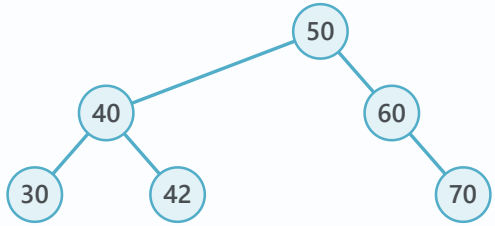
- `delete(50, 45)`
- `main()`

Code Trace: `delete(root, 45)` — One Child

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)`



main: `root = 50`. 45 is gone; its only child, 42, is now 40's right child. Inorder sequence preserved.

Step 11 / 11

CALL STACK

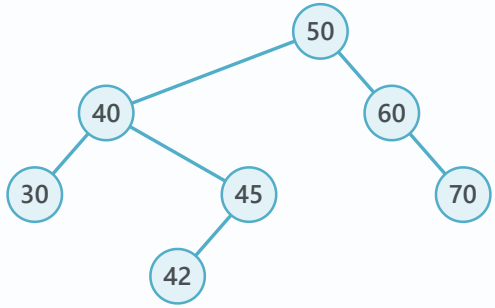
`main()`

Code Trace: `delete(root, 40)` — Two Children

```
▶ BinaryTreeNode<E> delete(
2         BinaryTreeNode<E> root, E data) {
3     if (root == null) return null;   // not found
4     int k = data.compareTo(root.data);
5     if (k < 0)
6         root.leftChild = delete(root.leftChild, data);
7     else if (k > 0)
8         root.rightChild = delete(root.rightChild, data);
9     else if (root.leftChild == null)
10        return root.rightChild;  // leaf / right child only
11    else if (root.rightChild == null)
12        return root.leftChild;   // left child only
13    else {                       // two children
14        root.data = minValue(root.rightChild).data;
15        root.rightChild = delete(root.rightChild, root.data);
16    }
17    return root;
18 }
```

VARIABLES

`data = 40`



main calls `root = delete(root, 40)`. First, find the node.

Step 1 / 17

CALL STACK

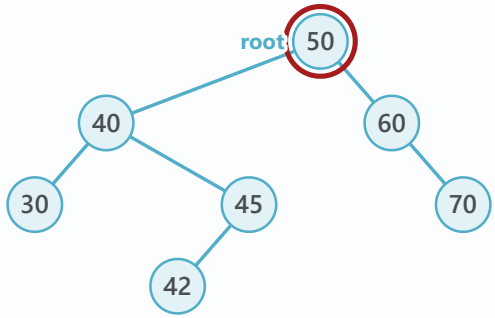
`main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)` `data = 40` `k = -1`



root ≠ null. **k = -1** since 40 < 50.

Step 2 / 17

CALL STACK

`delete(50, 40)`

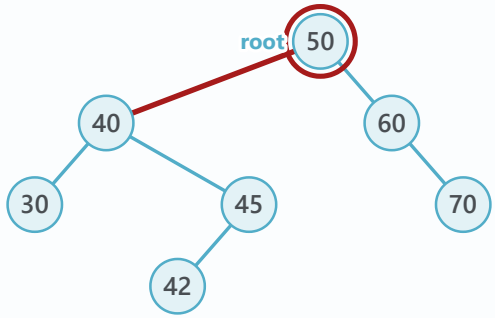
`main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
▶      if (k < 0)
▶          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)` `data = 40` `k = -1`



Go left: `delete(root.leftChild, 40)`. The result will go into **root.leftChild**.

Step 3 / 17

CALL STACK

`delete(50, 40)`

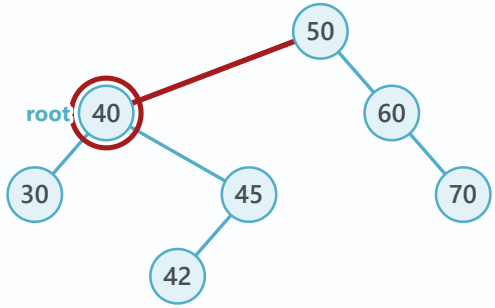
`main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(40)` `data = 40` `k = 0`



root ≠ null. **k = 0** since 40 = 40.

Step 4 / 17

CALL STACK

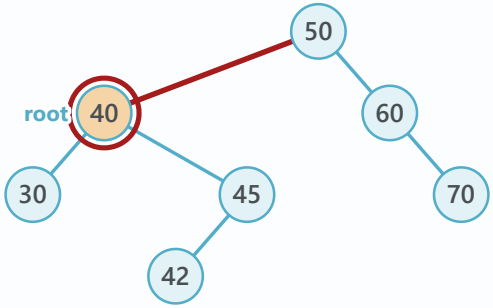
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
▶      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
▶      else if (root.rightChild == null)
12         return root.leftChild;   // left child only
▶      else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(40)` `data = 40` `k = 0`



k = 0 → found, but it has **two children** (30 and 45), so it cannot simply be removed.

CALL STACK

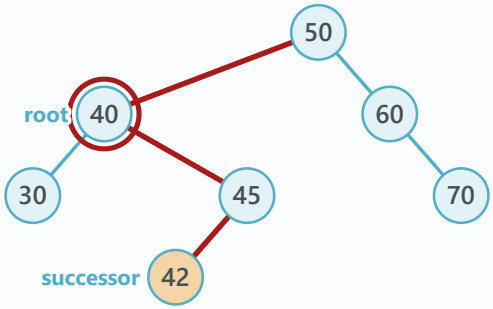
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
▶          root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(40)` `data = 40` `successor = node(42)`



`minValue(root.rightChild)` : one step right (45), then left all the way → **successor = 42** (smallest on the right).

Step 6 / 17

CALL STACK

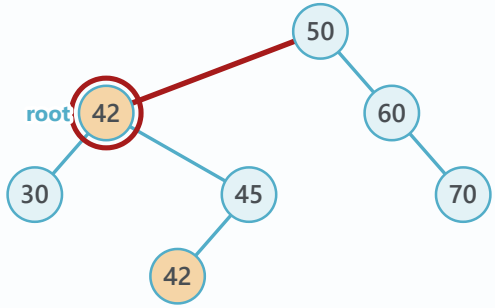
- `minValue(45)`
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
▶          root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(42)` `data = 40`



root.data = 42: the successor's value is copied into the target node. 42 now appears twice; the lower copy must go.

Step 7 / 17

CALL STACK

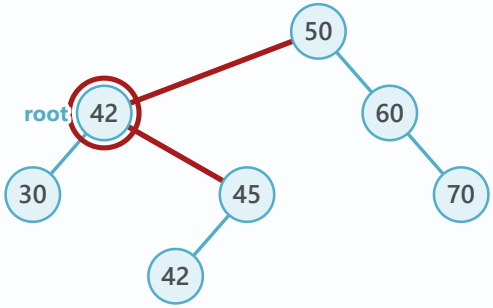
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
▶          root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(42)` `data = 40`



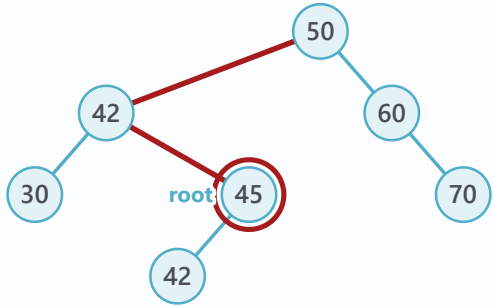
`delete(root.rightChild, 42)` : delete the old 42 from the right subtree. The successor has no left child, so this is always the leaf or one-child case.

CALL STACK

- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```



root ≠ null. **k = -1** since 42 < 45.

Step 9 / 17

VARIABLES

`root = node(45)` `data = 42` `k = -1`

CALL STACK

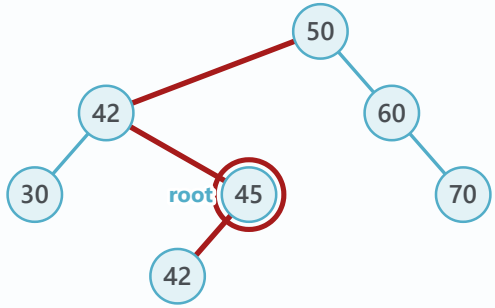
- `delete(45, 42)`
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
▶      if (k < 0)
▶          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(45)` `data = 42` `k = -1`



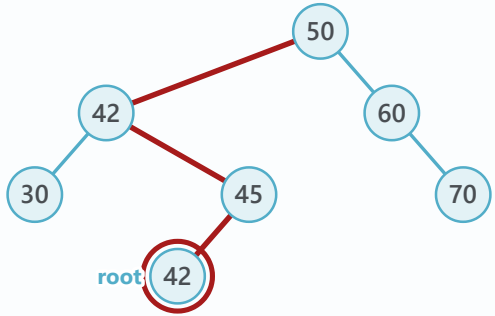
Go left: `delete(root.leftChild, 42)`. The result will go into **root.leftChild**.

CALL STACK

- `delete(45, 42)`
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
▶      if (root == null) return null;   // not found
▶      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```



root ≠ null. **k = 0** since 42 = 42.

Step 11 / 17

VARIABLES

`root = node(42)` `data = 42` `k = 0`

CALL STACK

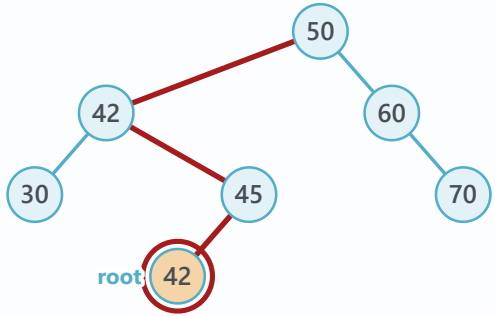
- `delete(42, 42)`
- `delete(45, 42)`
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
▶      else if (root.leftChild == null)
▶          return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(42)` `data = 42` `k = 0`



k = 0 → found the node to delete. No left child → **return root.rightChild** = null. (A leaf returns null.)

CALL STACK

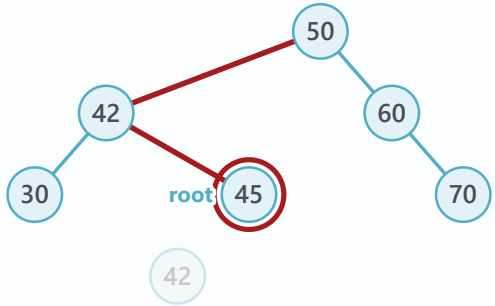
- `delete(42, 42)`
- `delete(45, 42)`
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
▶          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(45)` `data = 42` `k = -1` `returned = null`



Changed link: **root.leftChild = null**. Nothing points to the deleted node now; the garbage collector reclaims it.

Step 13 / 17

CALL STACK

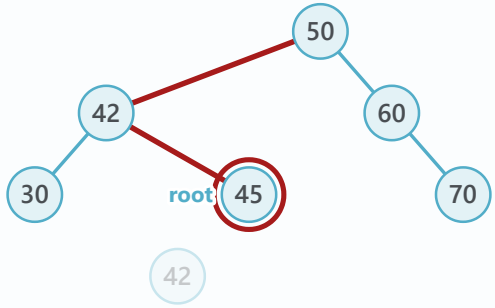
- `delete(45, 42)`
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
▶      return root;
18 }
```

VARIABLES

`root = node(45)` `data = 42` `k = -1`



return root: 45 goes back to its parent.

Step 14 / 17

CALL STACK

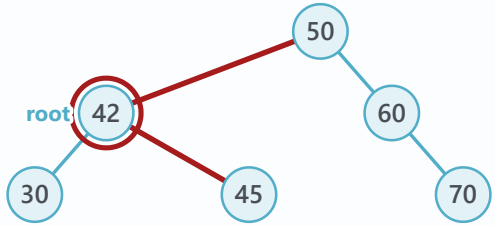
- `delete(45, 42)`
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
▶          root.rightChild = delete(root.rightChild, root.data);
16     }
▶      return root;
18 }
```

VARIABLES

`root = node(42)` `data = 40` `returned = node(45)`



`root.rightChild = node(45)` (the same link), **return root**. 42 now stands where the deleted 40 was.

Step 15 / 17

CALL STACK

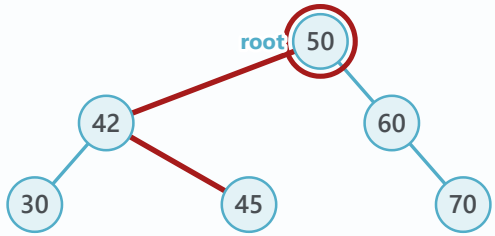
- `delete(40, 40)`
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
▶          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
▶      return root;
18 }
```

VARIABLES

`root = node(50)` `data = 40` `k = -1` `returned = node(42)`



`root.leftChild = node(42)` : same link, **no change**. Then **return root** (50).

CALL STACK

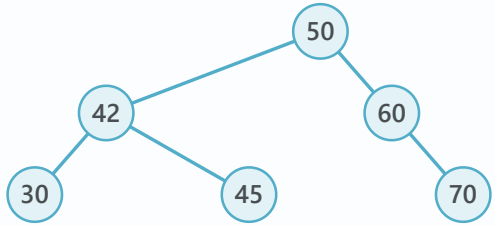
- `delete(50, 40)`
- `main()`

Code Trace: `delete(root, 40)` — Two Children

```
1  BinaryTreeNode<E> delete(
2          BinaryTreeNode<E> root, E data) {
3      if (root == null) return null;   // not found
4      int k = data.compareTo(root.data);
5      if (k < 0)
6          root.leftChild = delete(root.leftChild, data);
7      else if (k > 0)
8          root.rightChild = delete(root.rightChild, data);
9      else if (root.leftChild == null)
10         return root.rightChild;  // leaf / right child only
11     else if (root.rightChild == null)
12         return root.leftChild;   // left child only
13     else {                       // two children
14         root.data = minValue(root.rightChild).data;
15         root.rightChild = delete(root.rightChild, root.data);
16     }
17     return root;
18 }
```

VARIABLES

`root = node(50)`



main: `root = 50`. 40 is gone and its successor 42 stands in its place; one node fewer, still a BST. Cost: O(h).

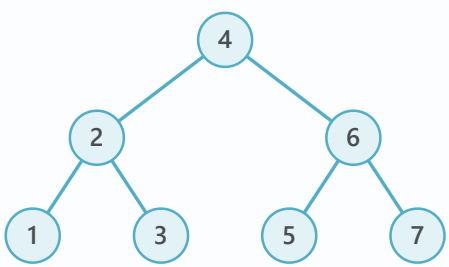
Step 17 / 17

CALL STACK

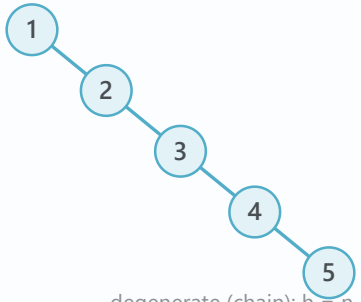
`main()`

Analyzing BST Operations

- Search, insertion, and deletion take time proportional to **the depth of the deepest node reached**: O(h).



balanced: h = 2 ≈ $\log_2 n$

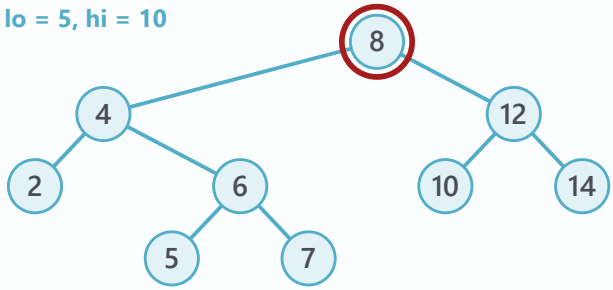


degenerate (chain): h = n − 1

Case	Height h	Search / Insert / Delete	Steps for n = 10^6
Balanced (best)	≈ $\log_2 n$	O(log n)	≈ 20
Random order (average)	≈ 2.99 $\log_2 n$	O(log n)	≈ 28 (average)
Degenerate (worst, sorted input)	n − 1	O(n)	10^6

With random insertion order, the **average depth** of a node is ≈ 2 ln n ≈ 1.39 $\log_2 n$ (≈ 28 steps for n = 10^6), while the **expected height** is ≈ 4.31 ln n ≈ 2.99 $\log_2 n$ (≈ 60).

Code Trace: the Range Query `range(root, 5, 10)`



`range(8)` : 5 < 8 → the left may hold keys in range: `range(k.left)` .

Step 1 / 19

```
1 void range(TreeNode k, int lo, int hi) {
▶     if (k == null) return;
▶     if (lo < k.data)           // candidates on the left
▶         range(k.left, lo, hi);
5     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
7     if (k.data < hi)           // candidates on the right
8         range(k.right, lo, hi);
9 }
```

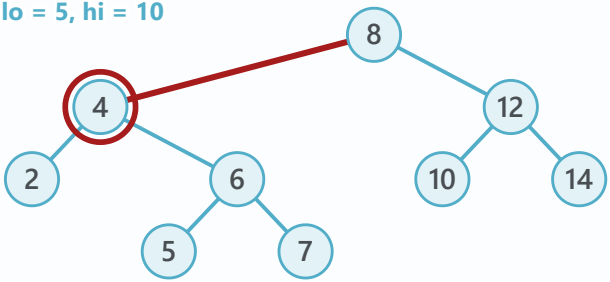
CALL STACK

- `range(8)`
- `main()`

OUTPUT

Code Trace: the Range Query `range(root, 5, 10)`

lo = 5, hi = 10



`range(4)` : 5 < 4 is false → every key on the left is < 4 ≤ lo: **left subtree pruned** (never visited).

Step 2 / 19

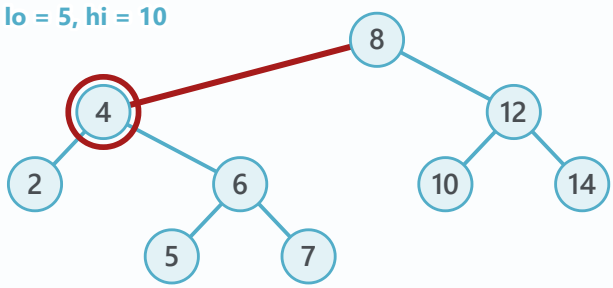
```
1  void range(TreeNode k, int lo, int hi) {
▶      if (k == null) return;
▶      if (lo < k.data)           // candidates on the left
4          range(k.left, lo, hi);
5      if (lo <= k.data && k.data <= hi)
6          System.out.print(k.data + " ");
7      if (k.data < hi)           // candidates on the right
8          range(k.right, lo, hi);
9  }
```

CALL STACK

- `range(4)`
- `range(8)`
- `main()`

OUTPUT

Code Trace: the Range Query `range(root, 5, 10)`



Left subtree pruned (4 ≤ lo = 5), no call made; `range(4)` continues. 4 is out of range, not printed. 4 < 10 → try the right too.

Step 3 / 19

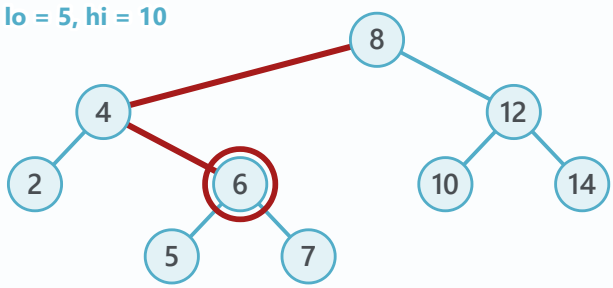
```
1 void range(TreeNode k, int lo, int hi) {
2     if (k == null) return;
3     if (lo < k.data)           // candidates on the left
4         range(k.left, lo, hi);
▶     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
▶     if (k.data < hi)           // candidates on the right
▶         range(k.right, lo, hi);
9 }
```

CALL STACK

- **range(4)**
- range(8)
- main()

OUTPUT

Code Trace: the Range Query `range(root, 5, 10)`



`range(6)` : 5 < 6 → the left may hold keys in range: `range(k.left)` .

Step 4 / 19

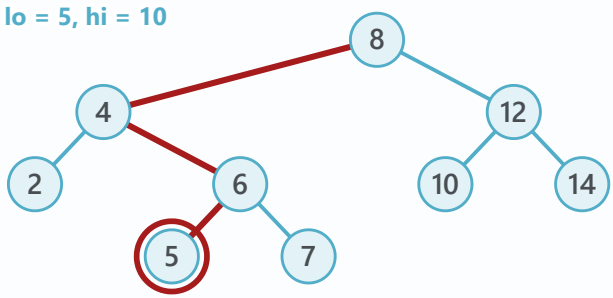
```
1 void range(TreeNode k, int lo, int hi) {
▶     if (k == null) return;
▶     if (lo < k.data)           // candidates on the left
▶         range(k.left, lo, hi);
5     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
7     if (k.data < hi)           // candidates on the right
8         range(k.right, lo, hi);
9 }
```

CALL STACK

- **range(6)**
- range(4)
- range(8)
- main()

OUTPUT

Code Trace: the Range Query `range(root, 5, 10)`



`range(5)` : 5 < 5 is false → every key on the left is < 5 ≤ lo: **left subtree pruned** (never visited).

Step 5 / 19

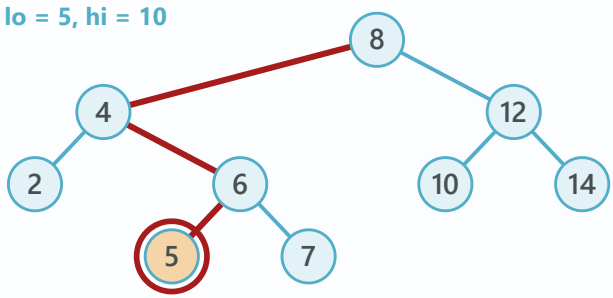
```
1 void range(TreeNode k, int lo, int hi) {
▶     if (k == null) return;
▶     if (lo < k.data)           // candidates on the left
4         range(k.left, lo, hi);
5     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
7     if (k.data < hi)           // candidates on the right
8         range(k.right, lo, hi);
9 }
```

CALL STACK

- **`range(5)`**
- `range(6)`
- `range(4)`
- `range(8)`
- `main()`

OUTPUT

Code Trace: the Range Query `range(root, 5, 10)`



Left subtree pruned (5 ≤ lo = 5), no call made; `range(5)` continues. 5 ∈ [5, 10] → **print**. 5 < 10 → try the right too.

Step 6 / 19

```
1 void range(TreeNode k, int lo, int hi) {
2     if (k == null) return;
3     if (lo < k.data)           // candidates on the
  left
4         range(k.left, lo, hi);
▶     if (lo <= k.data && k.data <= hi)
▶         System.out.print(k.data + " ");
▶     if (k.data < hi)           // candidates on the
  right
▶         range(k.right, lo, hi);
9 }
```

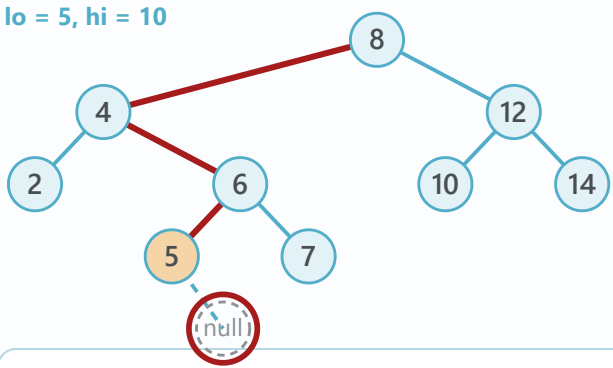
CALL STACK

- **`range(5)`**
- `range(6)`
- `range(4)`
- `range(8)`
- `main()`

OUTPUT

```
5
```

Code Trace: the Range Query `range(root, 5, 10)`



`range(null)` : empty subtree → **return** at once.

Step 7 / 19

```
1 void range(TreeNode k, int lo, int hi) {
▶       if (k == null) return;
3       if (lo < k.data)           // candidates on the
  left
4           range(k.left, lo, hi);
5       if (lo <= k.data && k.data <= hi)
6           System.out.print(k.data + " ");
7       if (k.data < hi)           // candidates on the
  right
8           range(k.right, lo, hi);
9   }
```

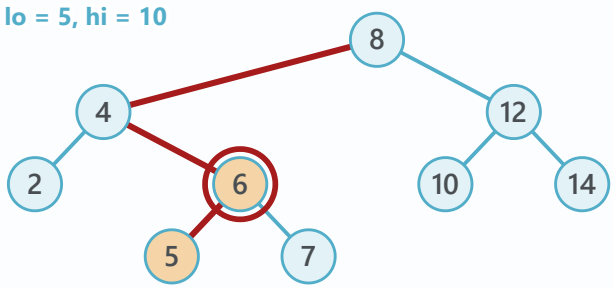
CALL STACK

- `range(null)`
- `range(5)`
- `range(6)`
- `range(4)`
- `range(8)`
- `main()`

OUTPUT

```
5
```

Code Trace: the Range Query `range(root, 5, 10)`



Left side done; back in `range(6)`. 6 ∈ [5, 10] → **print**. 6 < 10 → try the right too.

Step 8 / 19

```
1 void range(TreeNode k, int lo, int hi) {
2     if (k == null) return;
3     if (lo < k.data)           // candidates on the left
4         range(k.left, lo, hi);
▶     if (lo <= k.data && k.data <= hi)
▶         System.out.print(k.data + " ");
▶     if (k.data < hi)           // candidates on the right
▶         range(k.right, lo, hi);
9 }
```

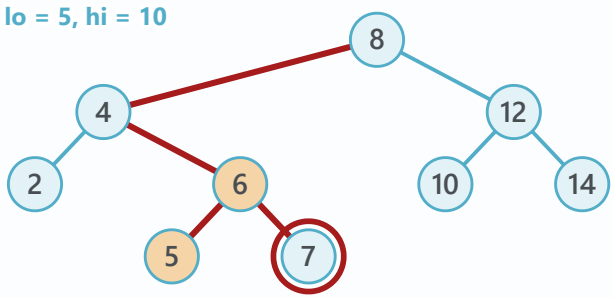
CALL STACK

- **range(6)**
- range(4)
- range(8)
- main()

OUTPUT

```
5 6
```

Code Trace: the Range Query `range(root, 5, 10)`



`range(7)` : 5 < 7 → the left may hold keys in range: `range(k.left)` .

Step 9 / 19

```
1 void range(TreeNode k, int lo, int hi) {
▶     if (k == null) return;
▶     if (lo < k.data)           // candidates on the left
▶         range(k.left, lo, hi);
5     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
7     if (k.data < hi)           // candidates on the right
8         range(k.right, lo, hi);
9 }
```

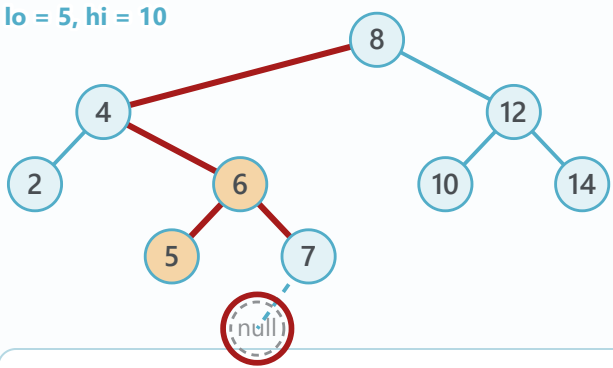
CALL STACK

- **range(7)**
- range(6)
- range(4)
- range(8)
- main()

OUTPUT

```
5 6
```

Code Trace: the Range Query `range(root, 5, 10)`



`range(null)` : empty subtree → **return** at once.

Step 10 / 19

```
1 void range(TreeNode k, int lo, int hi) {
▶       if (k == null) return;
3       if (lo < k.data)           // candidates on the
  left
4           range(k.left, lo, hi);
5       if (lo <= k.data && k.data <= hi)
6           System.out.print(k.data + " ");
7       if (k.data < hi)           // candidates on the
  right
8           range(k.right, lo, hi);
9   }
```

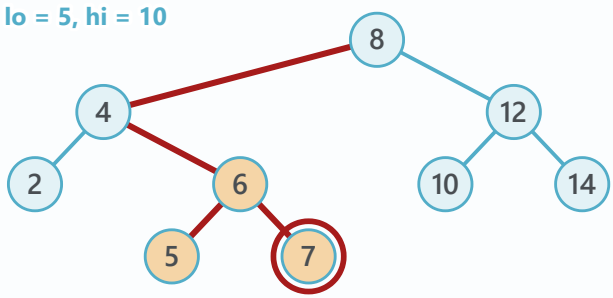
CALL STACK

- `range(null)`
- `range(7)`
- `range(6)`
- `range(4)`
- `range(8)`
- `main()`

OUTPUT

```
5 6
```

Code Trace: the Range Query `range(root, 5, 10)`



Left side done; back in `range(7)`. 7 ∈ [5, 10] → **print**. 7 < 10 → try the right too.


```
1 void range(TreeNode k, int lo, int hi) {
2     if (k == null) return;
3     if (lo < k.data)           // candidates on the
  left
4         range(k.left, lo, hi);
▶     if (lo <= k.data && k.data <= hi)
▶         System.out.print(k.data + " ");
▶     if (k.data < hi)           // candidates on the
  right
▶         range(k.right, lo, hi);
9 }
```

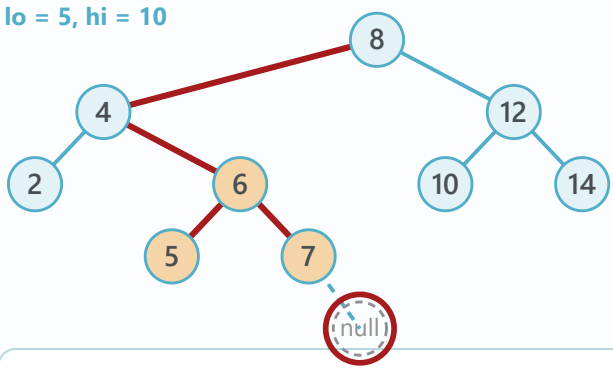
CALL STACK

- **range(7)**
- range(6)
- range(4)
- range(8)
- main()

OUTPUT

```
5 6 7
```

Code Trace: the Range Query `range(root, 5, 10)`



`range(null)` : empty subtree → **return** at once.

Step 12 / 19

```
1 void range(TreeNode k, int lo, int hi) {
▶     if (k == null) return;
3     if (lo < k.data)           // candidates on the
  left
4         range(k.left, lo, hi);
5     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
7     if (k.data < hi)           // candidates on the
  right
8         range(k.right, lo, hi);
9 }
```

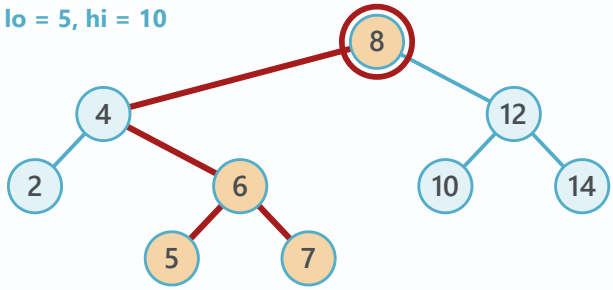
CALL STACK

- **`range(null)`**
- `range(7)`
- `range(6)`
- `range(4)`
- `range(8)`
- `main()`

OUTPUT

```
5 6 7
```

Code Trace: the Range Query `range(root, 5, 10)`



Left side done; back in `range(8)`. 8 ∈ [5, 10] → **print**. 8 < 10 → try the right too.

Step 13 / 19

```
1  void range(TreeNode k, int lo, int hi) {
2      if (k == null) return;
3      if (lo < k.data)           // candidates on the
   left
4          range(k.left, lo, hi);
▶      if (lo <= k.data && k.data <= hi)
▶          System.out.print(k.data + " ");
▶      if (k.data < hi)           // candidates on the
   right
▶          range(k.right, lo, hi);
9  }
```

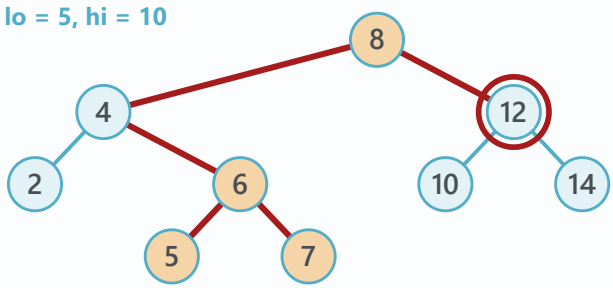
CALL STACK

- `range(8)`
- `main()`

OUTPUT

```
5 6 7 8
```

Code Trace: the Range Query `range(root, 5, 10)`



`range(12)` : 5 < 12 → the left may hold keys in range: `range(k.left)` .


```
1 void range(TreeNode k, int lo, int hi) {
▶     if (k == null) return;
▶     if (lo < k.data)           // candidates on the left
▶         range(k.left, lo, hi);
5     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
7     if (k.data < hi)           // candidates on the right
8         range(k.right, lo, hi);
9 }
```

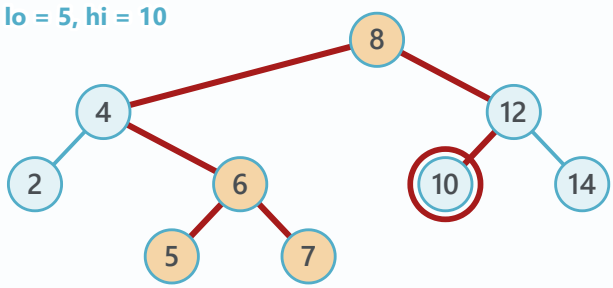
CALL STACK

- **range(12)**
- range(8)
- main()

OUTPUT

```
5 6 7 8
```

Code Trace: the Range Query `range(root, 5, 10)`



`range(10)` : 5 < 10 → the left may hold keys in range: `range(k.left)` .

Step 15 / 19

```
1  void range(TreeNode k, int lo, int hi) {
▶      if (k == null) return;
▶      if (lo < k.data)           // candidates on the
   left
▶          range(k.left, lo, hi);
5      if (lo <= k.data && k.data <= hi)
6          System.out.print(k.data + " ");
7      if (k.data < hi)           // candidates on the
   right
8          range(k.right, lo, hi);
9  }
```

CALL STACK

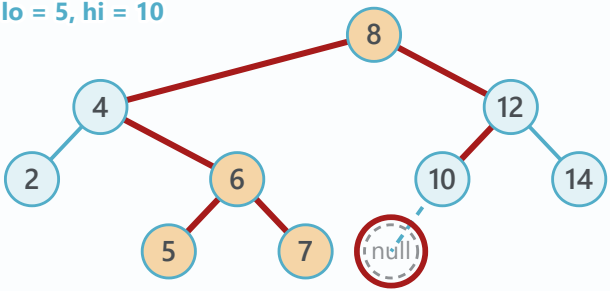
- **`range(10)`**
- `range(12)`
- `range(8)`
- `main()`

OUTPUT

```
5 6 7 8
```

Code Trace: the Range Query `range(root, 5, 10)`

lo = 5, hi = 10



`range(null)` : empty subtree → **return** at once.

Step 16 / 19

```
1 void range(TreeNode k, int lo, int hi) {
▶       if (k == null) return;
3       if (lo < k.data)           // candidates on the
  left
4           range(k.left, lo, hi);
5       if (lo <= k.data && k.data <= hi)
6           System.out.print(k.data + " ");
7       if (k.data < hi)           // candidates on the
  right
8           range(k.right, lo, hi);
9   }
```

CALL STACK

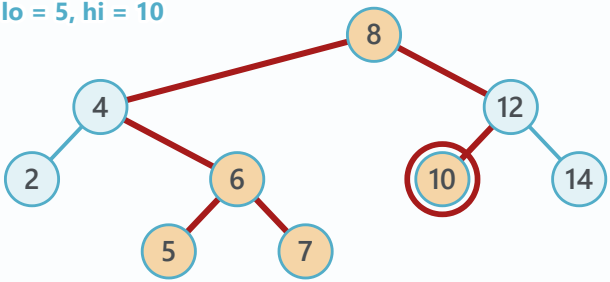
- `range(null)`
- `range(10)`
- `range(12)`
- `range(8)`
- `main()`

OUTPUT

```
5 6 7 8
```

Code Trace: the Range Query `range(root, 5, 10)`

lo = 5, hi = 10



Left side done; back in `range(10)`. 10 ∈ [5, 10] → **print**. 10 ≥ 10 → right subtree pruned.

Step 17 / 19

```
1  void range(TreeNode k, int lo, int hi) {
2      if (k == null) return;
3      if (lo < k.data)           // candidates on the
   left
4          range(k.left, lo, hi);
▶      if (lo <= k.data && k.data <= hi)
▶          System.out.print(k.data + " ");
▶      if (k.data < hi)           // candidates on the
   right
8          range(k.right, lo, hi);
9  }
```

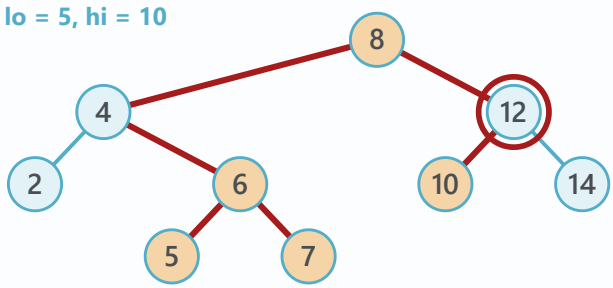
CALL STACK

- `range(10)`
- `range(12)`
- `range(8)`
- `main()`

OUTPUT

```
5 6 7 8 10
```

Code Trace: the Range Query `range(root, 5, 10)`



Left side done; back in `range(12)`. 12 is out of range, not printed. 12 ≥ 10 → right subtree pruned.

Step 18 / 19

```
1 void range(TreeNode k, int lo, int hi) {
2     if (k == null) return;
3     if (lo < k.data)           // candidates on the
  left
4         range(k.left, lo, hi);
▶     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
▶     if (k.data < hi)           // candidates on the
  right
8         range(k.right, lo, hi);
9 }
```

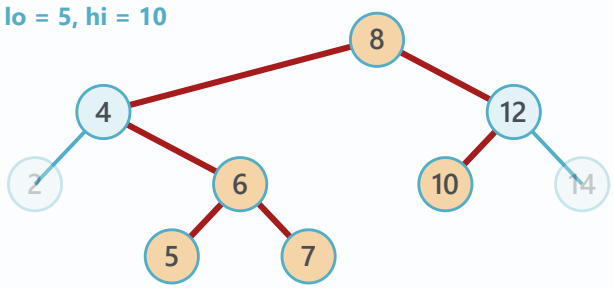
CALL STACK

- **range(12)**
- range(8)
- main()

OUTPUT

```
5 6 7 8 10
```

Code Trace: the Range Query `range(root, 5, 10)`



Done. The output is in sorted (inorder) order: **5 6 7 8 10**. 2 and 14 were **never visited** (faded). Cost: **O(h + s)**, s = keys printed.

Step 19 / 19

```
▶ void range(TreeNode k, int lo, int hi) {
2     if (k == null) return;
3     if (lo < k.data)           // candidates on the
  left
4         range(k.left, lo, hi);
5     if (lo <= k.data && k.data <= hi)
6         System.out.print(k.data + " ");
7     if (k.data < hi)           // candidates on the
  right
8         range(k.right, lo, hi);
9 }
```

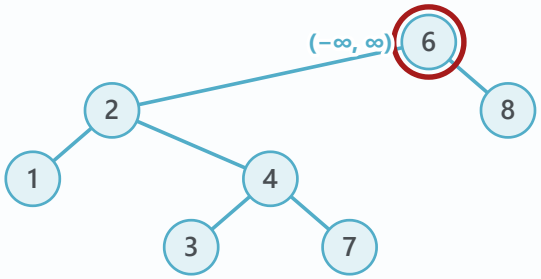
CALL STACK

`main()`

OUTPUT

```
5 6 7 8 10
```

Code Trace: Is This Tree a BST? `isBST()`



`isBST(6)` : 6 ∈ (−∞, ∞) ✓. Left subtree gets upper bound **6**: `isBST(k.left, −∞, 6)` .

Step 1 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
2      if (k == null) return true;
▶      if (k.data <= min || k.data >= max)
4          return false;          // out of range
▶      return isBST(k.left, min, k.data)
6          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

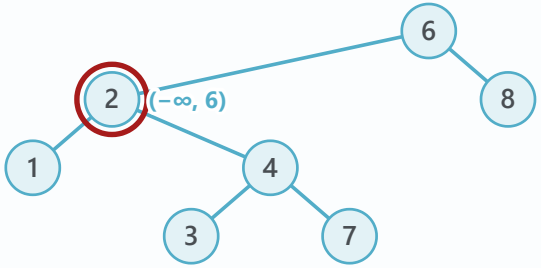
CALL STACK

- `isBST(6)`
- `main()`

VARIABLES

`k = node(6)` `min = −∞` `max = ∞`

Code Trace: Is This Tree a BST? `isBST()`



`isBST(2)` : 2 ∈ (−∞, 6) ✓. Left subtree gets upper bound **2**: `isBST(k.left, −∞, 2)` .

Step 2 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
2      if (k == null) return true;
▶      if (k.data <= min || k.data >= max)
4          return false;          // out of range
▶      return isBST(k.left, min, k.data)
6          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

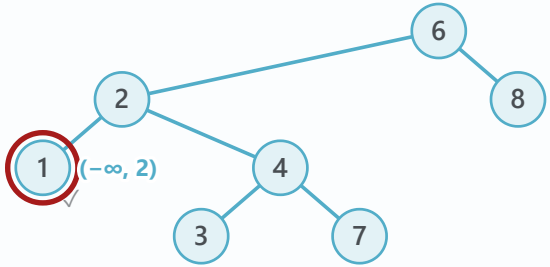
CALL STACK

- **isBST(2)**
- isBST(6)
- main()

VARIABLES

`k = node(2)` `min = −∞` `max = 6`

Code Trace: Is This Tree a BST? `isBST()`



`isBST(1)` : 1 ∈ (−∞, 2) ✓. Leaf: both null children return **true** → **return true**.

Step 3 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
▶      if (k == null) return true;
▶      if (k.data <= min || k.data >= max)
4          return false;          // out of range
▶      return isBST(k.left, min, k.data)
▶          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

CALL STACK

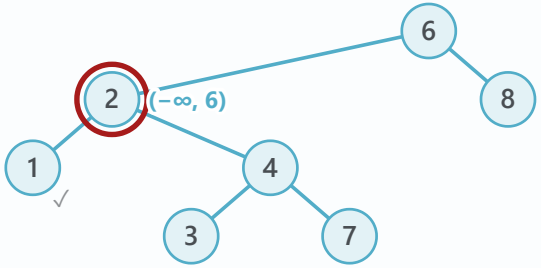
- **isBST(1)**
- isBST(2)
- isBST(6)
- main()

VARIABLES

`k = node(1)` `min = −∞` `max = 2`

`returned = true`

Code Trace: Is This Tree a BST? `isBST()`



Left side **true**. Right subtree gets lower bound **2**: `isBST(k.right, 2, 6)`.

Step 4 / 12

```
1 boolean isBST(TreeNode k, long min, long max) {
2     if (k == null) return true;
3     if (k.data <= min || k.data >= max)
4         return false;          // out of range
5     return isBST(k.left, min, k.data)
▶         && isBST(k.right, k.data, max);
7 }
8 // first call: isBST(root, Long.MIN_VALUE,
9 //                   Long.MAX_VALUE)
```

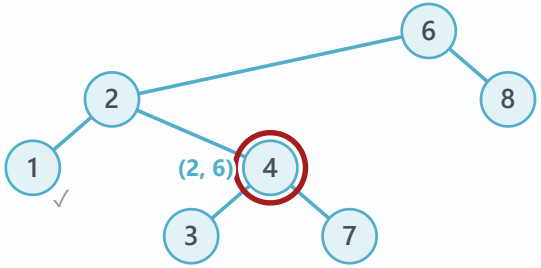
CALL STACK

- **isBST(2)**
- isBST(6)
- main()

VARIABLES

`k = node(2)` `min = −∞` `max = 6`

Code Trace: Is This Tree a BST? `isBST()`



`isBST(4)` : 4 ∈ (2, 6) ✓. Left subtree gets upper bound **4**: `isBST(k.left, 2, 4)` .

Step 5 / 12

```
1 boolean isBST(TreeNode k, long min, long max) {
2     if (k == null) return true;
▶     if (k.data <= min || k.data >= max)
4         return false;           // out of range
▶     return isBST(k.left, min, k.data)
6         && isBST(k.right, k.data, max);
7 }
8 // first call: isBST(root, Long.MIN_VALUE,
9 //                   Long.MAX_VALUE)
```

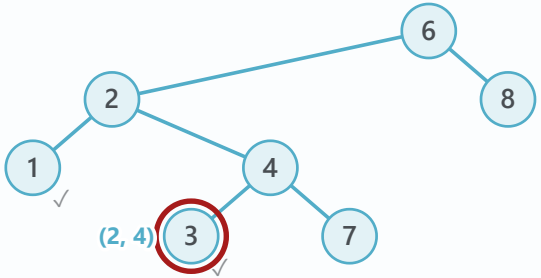
CALL STACK

- **isBST(4)**
- isBST(2)
- isBST(6)
- main()

VARIABLES

`k = node(4)` `min = 2` `max = 6`

Code Trace: Is This Tree a BST? `isBST()`



`isBST(3)` : 3 ∈ (2, 4) ✓. Leaf: both null children return **true** → **return true**.

Step 6 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
▶      if (k == null) return true;
▶      if (k.data <= min || k.data >= max)
4          return false;          // out of range
▶      return isBST(k.left, min, k.data)
▶          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

CALL STACK

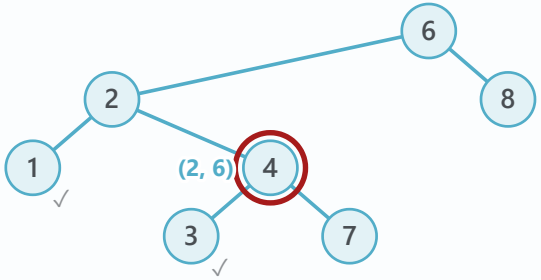
- **isBST(3)**
- isBST(4)
- isBST(2)
- isBST(6)
- main()

VARIABLES

`k = node(3)` `min = 2` `max = 4`

`returned = true`

Code Trace: Is This Tree a BST? `isBST()`



Left side **true**. Right subtree gets lower bound **4**: `isBST(k.right, 4, 6)`.

Step 7 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
2      if (k == null) return true;
3      if (k.data <= min || k.data >= max)
4          return false;          // out of range
5      return isBST(k.left, min, k.data)
▶          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

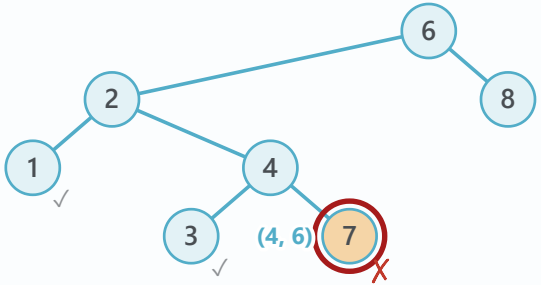
CALL STACK

- **isBST(4)**
- isBST(2)
- isBST(6)
- main()

VARIABLES

`k = node(4)` `min = 2` `max = 6`

Code Trace: Is This Tree a BST? `isBST()`



`isBST(7)`, allowed range **(4, 6)**: 7 ≥ 6 → **return false**. Right of 4, 7 must exceed 4 (✓); in 6's left subtree, it must also be **less** than 6.

Step 8 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
2      if (k == null) return true;
▶      if (k.data <= min || k.data >= max)
▶          return false;          // out of range
5      return isBST(k.left, min, k.data)
6          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

CALL STACK

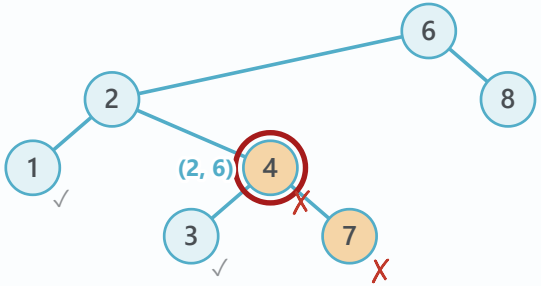
- **isBST(7)**
- isBST(4)
- isBST(2)
- isBST(6)
- main()

VARIABLES

`k = node(7)` `min = 4` `max = 6`

`returned = false`

Code Trace: Is This Tree a BST? `isBST()`



false from the right → `isBST(4)` = true && false = **false**.

Step 9 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
2      if (k == null) return true;
3      if (k.data <= min || k.data >= max)
4          return false;           // out of range
▶      return isBST(k.left, min, k.data)
▶          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

CALL STACK

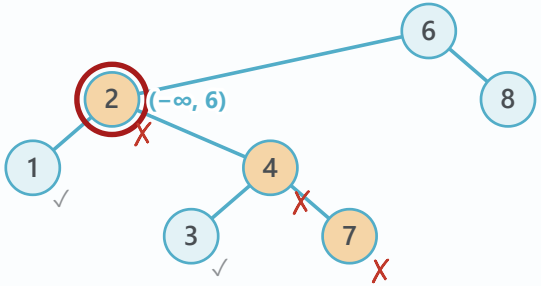
- **isBST(4)**
- isBST(2)
- isBST(6)
- main()

VARIABLES

`k = node(4)` `min = 2` `max = 6`

`returned = false`

Code Trace: Is This Tree a BST? `isBST()`



false from the right → `isBST(2)` = true && false = **false**.

Step 10 / 12

```
1 boolean isBST(TreeNode k, long min, long max) {
2     if (k == null) return true;
3     if (k.data <= min || k.data >= max)
4         return false;           // out of range
▶     return isBST(k.left, min, k.data)
▶         && isBST(k.right, k.data, max);
7 }
8 // first call: isBST(root, Long.MIN_VALUE,
9 //                    Long.MAX_VALUE)
```

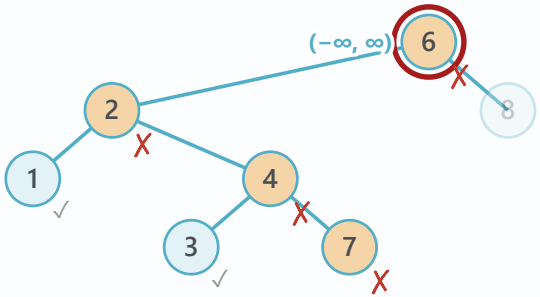
CALL STACK

- **isBST(2)**
- isBST(6)
- main()

VARIABLES

`k = node(2)` `min = −∞` `max = 6` `returned = false`

Code Trace: Is This Tree a BST? `isBST()`



Left result **false** → `&&` short-circuits: the right subtree is never checked; `isBST(6)` returns **false**.

Step 11 / 12

```
1  boolean isBST(TreeNode k, long min, long max) {
2      if (k == null) return true;
3      if (k.data <= min || k.data >= max)
4          return false;          // out of range
▶      return isBST(k.left, min, k.data)
▶          && isBST(k.right, k.data, max);
7  }
8  // first call: isBST(root, Long.MIN_VALUE,
9  //                   Long.MAX_VALUE)
```

CALL STACK

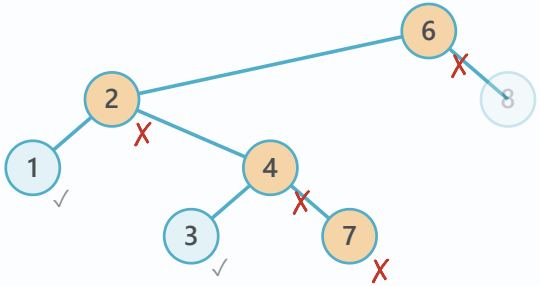
- `isBST(6)`
- `main()`

VARIABLES

`k = node(6)` `min = −∞` `max = ∞`

`returned = false`

Code Trace: Is This Tree a BST? `isBST()`



Result: **false**; not a BST. A check of **parent–child** pairs alone (4 < 7 ✓) would wrongly say **true**; the range must be passed down. Cost: O(n).

Step 12 / 12

```
1 boolean isBST(TreeNode k, long min, long max) {
2     if (k == null) return true;
3     if (k.data <= min || k.data >= max)
4         return false;          // out of range
5     return isBST(k.left, min, k.data)
6         && isBST(k.right, k.data, max);
7 }
▶ // first call: isBST(root, Long.MIN_VALUE,
▶ //                  Long.MAX_VALUE)
```

CALL STACK

`main()`

VARIABLES

`result = false`

Application: Student Records

The **student ID** is the key; each node carries the whole student object.

```
public class Student {
    int id;     // key
    String fullName;

    public Student(int id,
                   String fullName) {
        this.id = id;
        this.fullName = fullName;
    }
}
```

```
public class StudentNode {
    Student st;           // data
    StudentNode leftChild;
    StudentNode rightChild;

    public StudentNode(Student st) {
        this.st = st;
        // children start out null
    }
}
```

Java's `TreeMap<Integer, Student>` packages the same idea: a key → value map backed by a balanced (red-black) search tree.

BST OPERATIONS

Inserting and Finding Students

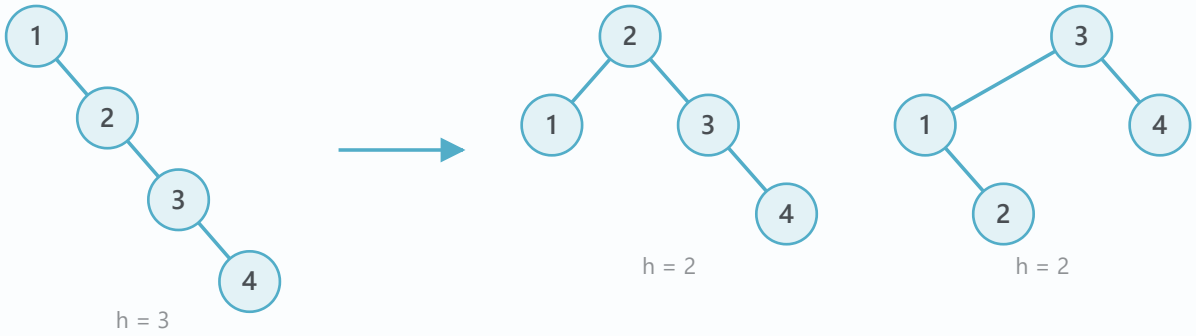
```
StudentNode insert(StudentNode root, Student st) {
    if (root == null) return new StudentNode(st);
    if (st.id < root.st.id)
        root.leftChild = insert(root.leftChild, st);     // continue left
    else if (st.id > root.st.id)
        root.rightChild = insert(root.rightChild, st);   // continue right
    return root;                                         // duplicate IDs are skipped
}
```

```
Student search(StudentNode root, int id) {
    if (root == null) return null;                // no such record
    if (id == root.st.id) return root.st; // found
    if (id < root.st.id)
        return search(root.leftChild, id);         // continue left
    return search(root.rightChild, id);            // continue right
}
```

`search` returns the student object, not a boolean, so a hit also gives the name and the rest of the record. With an `int` key, `==` and `<` work directly.

Why Balance Matters

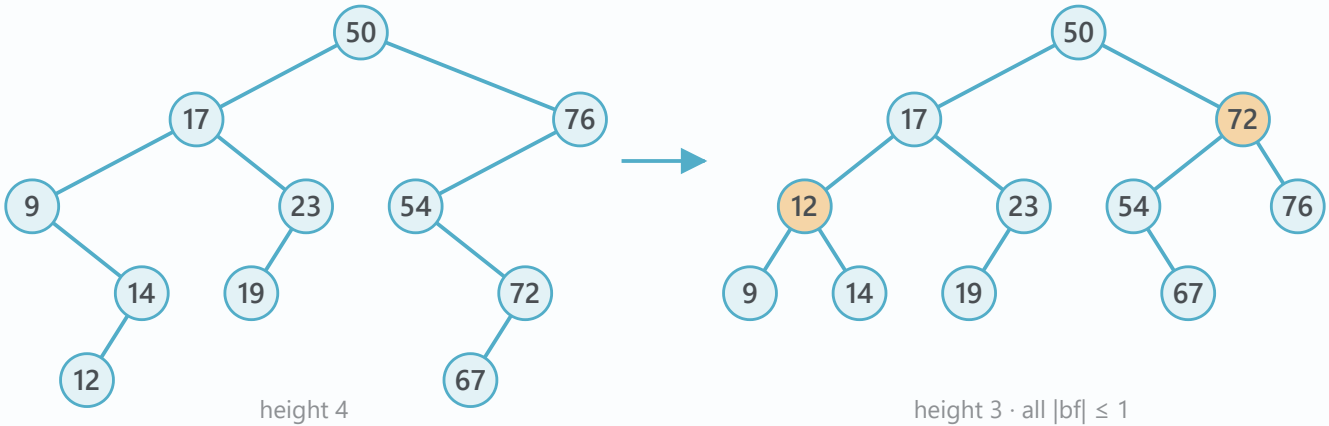
- A BST's shape depends on the **insertion order**: insert 1, 2, 3, 4 in order and it becomes a linked list.
- Arranged differently, the same keys fit in a shorter tree (two options on the right).



Self-balancing search trees repair themselves after each insertion and deletion with small local changes (rotations, recoloring), guaranteeing h = O(log n).

A Rebalancing Example

On the left, the branches 9 → 14 → 12 and 76 → 54 → 72 → 67 have grown long. Rearranging those subtrees (new subtree roots in orange) fits the same 11 keys into a shorter tree:



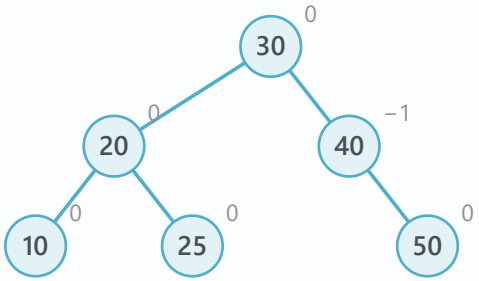
- Rearranging **does not change the inorder sequence**: 9 12 14 17 19 23 50 54 67 72 76.
- Rebuilding the tree each time would cost O(n); AVL and red-black trees work only along the update path, in O(log n).

AVL Trees

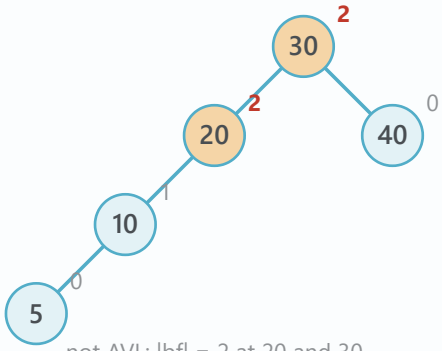
- Adelson-Velsky and Landis (1962): the first balanced BST.
- **Balance factor**: bf(x) = h(left) − h(right). An empty subtree has height −1.
- AVL condition: **bf ∈ {−1, 0, +1} at every node**.
- Consequence: $h < 1.44 \log_2(n + 2)$, so every operation is O(log n).
- Each node stores its height; after an update, bf is checked on the way back up to the root.

How to read it

The number at a node's upper right is its bf: positive means left-heavy, negative right-heavy. We rotate at the first (lowest) node with |bf| = 2.



AVL ✓



not AVL: |bf| = 2 at 20 and 30

Why Is the Height of an AVL Tree O(log n)?

Question: what is the **fewest** nodes an AVL tree of height h can have? In the sparsest one, the two subtrees of every internal node differ in height by exactly 1.

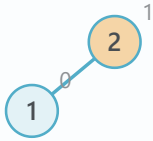


h = 0: a single node. **N(0) = 1**.

$N(h) = 1 + N(h-1) + N(h-2) > 2 \cdot N(h-2) \rightarrow N(h) > 2^{h/2} \rightarrow$ **$h < 2 \log_2 n$**. Fibonacci gives a tighter bound: $h < 1.44 \log_2(n + 2)$.

Why Is the Height of an AVL Tree O(log n)?

Question: what is the **fewest** nodes an AVL tree of height h can have? In the sparsest one, the two subtrees of every internal node differ in height by exactly 1.



N(0) = 1

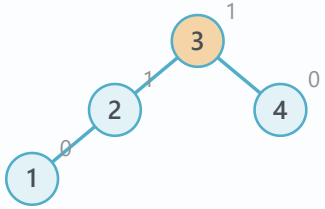
N(1) = 2

h = 1: a root and one child. **N(1) = 2**.

$N(h) = 1 + N(h-1) + N(h-2) > 2 \cdot N(h-2) \rightarrow N(h) > 2^{h/2} \rightarrow$ **$h < 2 \log_2 n$**. Fibonacci gives a tighter bound: $h < 1.44 \log_2(n + 2)$.

Why Is the Height of an AVL Tree O(log n)?

Question: what is the **fewest** nodes an AVL tree of height h can have? In the sparsest one, the two subtrees of every internal node differ in height by exactly 1.



N(0) = 1

N(1) = 2

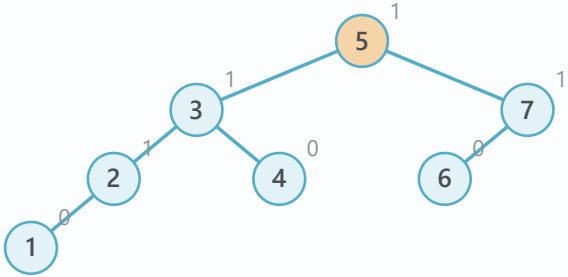
N(2) = 4

h = 2: root (orange) + sparsest tree of height 1 (left) + sparsest tree of height 0 (right). **N(2) = 1 + N(1) + N(0) = 1 + 2 + 1 = 4**.

$N(h) = 1 + N(h-1) + N(h-2) > 2 \cdot N(h-2) \rightarrow N(h) > 2^{h/2} \rightarrow$ **$h < 2 \log_2 n$**. Fibonacci gives a tighter bound: $h < 1.44 \log_2(n + 2)$.

Why Is the Height of an AVL Tree O(log n)?

Question: what is the **fewest** nodes an AVL tree of height h can have? In the sparsest one, the two subtrees of every internal node differ in height by exactly 1.



N(0) = 1

N(1) = 2

N(2) = 4

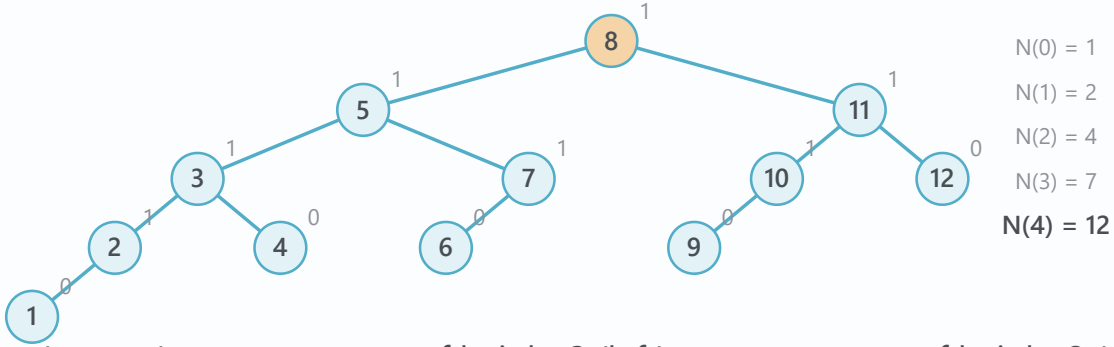
N(3) = 7

h = 3: root (orange) + sparsest tree of height 2 (left) + sparsest tree of height 1 (right). **N(3) = 1 + N(2) + N(1) = 1 + 4 + 2 = 7**.

$N(h) = 1 + N(h-1) + N(h-2) > 2 \cdot N(h-2) \rightarrow N(h) > 2^{h/2} \rightarrow$ **$h < 2 \log_2 n$**. Fibonacci gives a tighter bound: $h < 1.44 \log_2(n + 2)$.

Why Is the Height of an AVL Tree O(log n)?

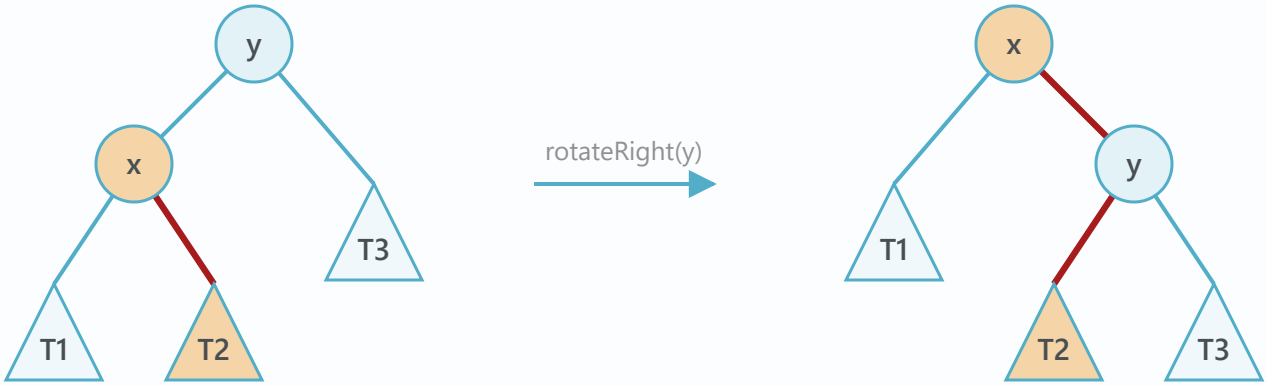
Question: what is the **fewest** nodes an AVL tree of height h can have? In the sparsest one, the two subtrees of every internal node differ in height by exactly 1.



h = 4: root (orange) + sparsest tree of height 3 (left) + sparsest tree of height 2 (right). **N(4) = 1 + N(3) + N(2) = 1 + 7 + 4 = 12**. The counts grow like Fibonacci numbers: N(h) = F(h+3) − 1.

N(h) = 1 + N(h−1) + N(h−2) > 2·N(h−2) → N(h) > $2^{h/2}$ → **h < 2 $\log_2$ n**. Fibonacci gives a tighter bound: h < 1.44 $\log_2(n + 2)$.

Rotation



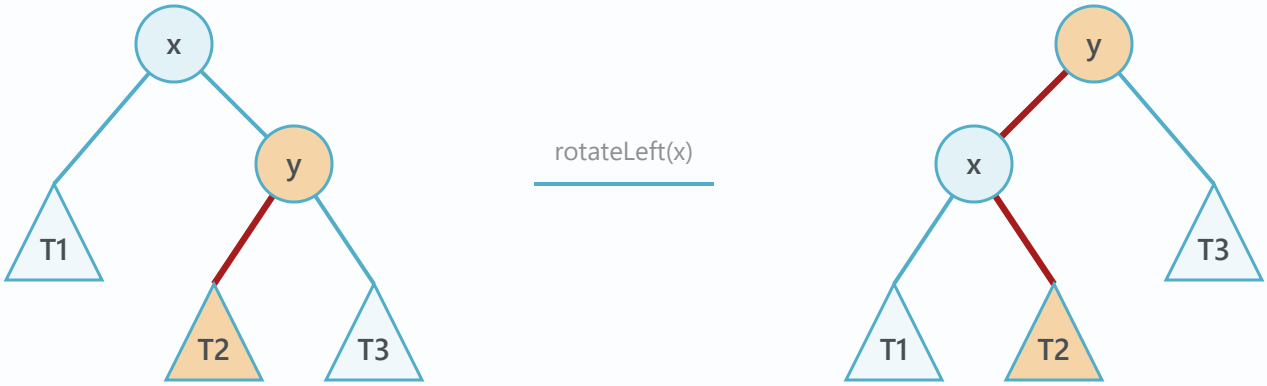
same inorder sequence on both sides: T1 · x · T2 · y · T3

Right rotation (y left-heavy): x moves up, y becomes x's right child, and x's right subtree T2 moves to y's left.

Step 1 / 2

A rotation preserves the BST property (same inorder sequence) but changes subtree heights: one side loses a level, the other gains one.

Rotation



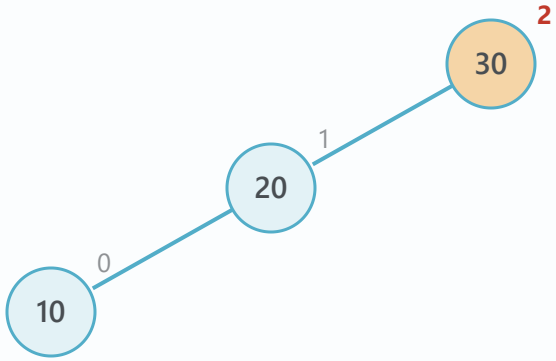
same inorder sequence on both sides: T1 · x · T2 · y · T3

Left rotation (x right-heavy): the mirror image. Only 3 links change (thick lines) → O(1).

Step 2 / 2

A rotation preserves the BST property (same inorder sequence) but changes subtree heights: one side loses a level, the other gains one.

Single Rotations: LL and RR

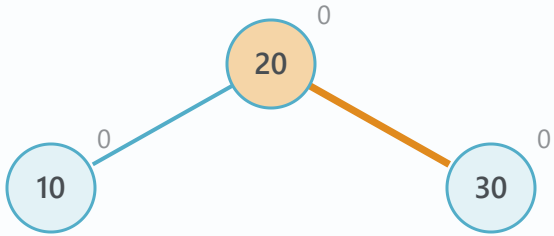


LL: 10 went to the **left** of the left child (20). bf(30) = 2 *X* and bf(20) = 1: both nodes lean the same way (left).

Step 1 / 4

The number at a node's upper right is its bf. LL takes a **single** right rotation, RR a single left rotation; the orange link is the one the rotation changes.

Single Rotations: LL and RR

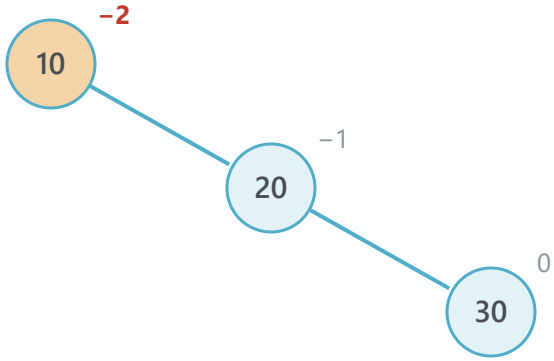


Rotate right at 30: 20 moves up and 30 becomes its right child (orange link). All bf = 0.

Step 2 / 4

The number at a node's upper right is its bf. LL takes a **single** right rotation, RR a single left rotation; the orange link is the one the rotation changes.

Single Rotations: LL and RR

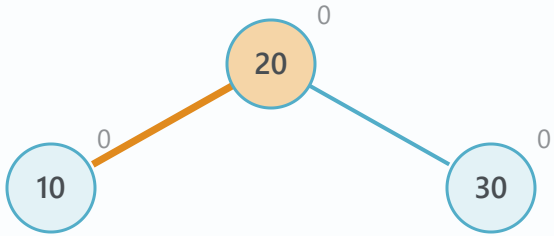


RR: 30 went to the **right** of the right child (20). bf(10) = −2 *X* and bf(20) = −1: both lean right.

Step 3 / 4

The number at a node's upper right is its bf. LL takes a **single** right rotation, RR a single left rotation; the orange link is the one the rotation changes.

Single Rotations: LL and RR

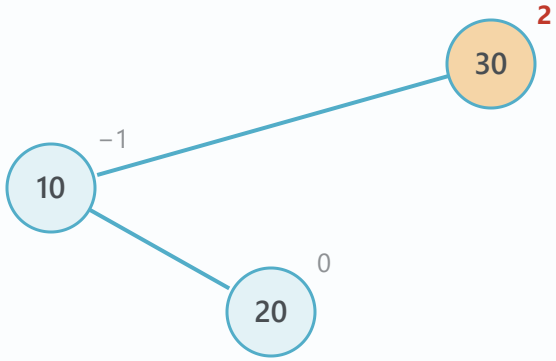


Rotate left at 10: 20 moves up and 10 becomes its left child. All bf = 0.

Step 4 / 4

The number at a node's upper right is its bf. LL takes a **single** right rotation, RR a single left rotation; the orange link is the one the rotation changes.

Double Rotations: LR and RL

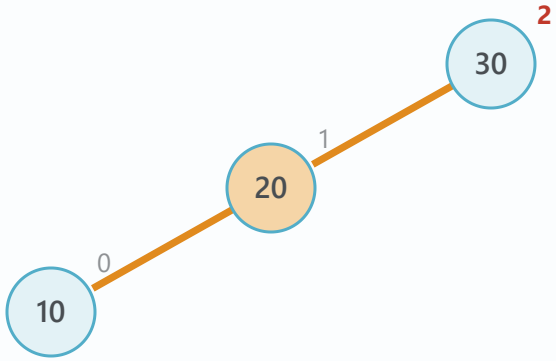


LR: 20 went to the **right** of the left child (10). bf(30) = 2 but bf(10) = −1: opposite leans (a triangle). A single right rotation still leaves a triangle.

Step 1 / 6

LR: rotate left at the child, then right at the unbalanced node. RL: right at the child, then left.

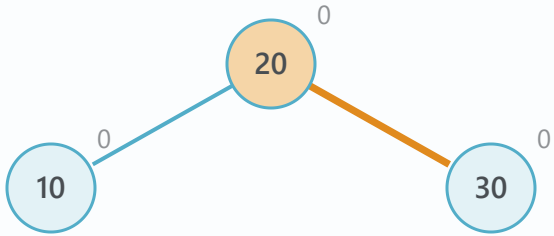
Double Rotations: LR and RL



Rotation 1: **rotate left** at 10 → 20 moves up; the triangle becomes a straight line (LL). bf(30) is still 2.

LR: rotate left at the child, then right at the unbalanced node. RL: right at the child, then left.

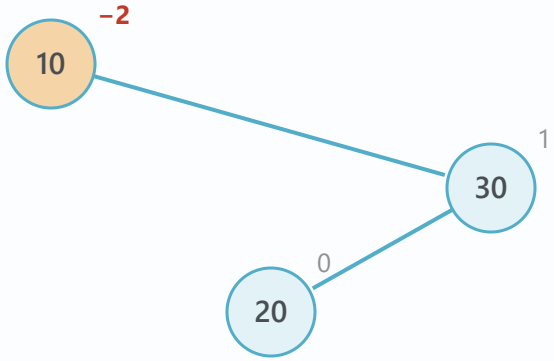
Double Rotations: LR and RL



Rotation 2: **rotate right** at 30 → the middle key, 20, becomes the root; all bf = 0.

LR: rotate left at the child, then right at the unbalanced node. RL: right at the child, then left.

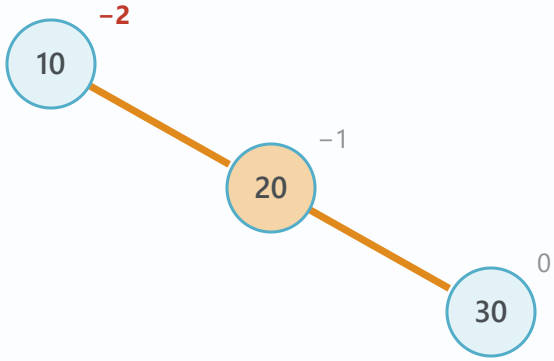
Double Rotations: LR and RL



RL: 20 went to the **left** of the right child (30). bf(10) = −2, bf(30) = 1: a triangle.

LR: rotate left at the child, then right at the unbalanced node. RL: right at the child, then left.

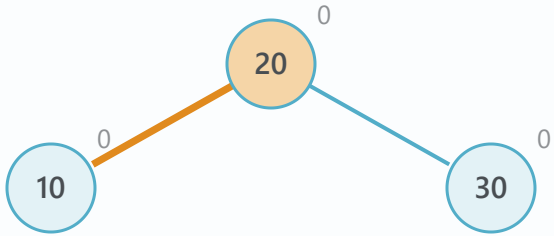
Double Rotations: LR and RL



Rotation 1: **rotate right** at 30 → a straight line (RR). bf(10) is still −2.

LR: rotate left at the child, then right at the unbalanced node. RL: right at the child, then left.

Double Rotations: LR and RL



Rotation 2: **rotate left** at 10 → 20 is the root; all bf = 0.

LR: rotate left at the child, then right at the unbalanced node. RL: right at the child, then left.

AVL Rotation and Rebalancing Code

```
int h(AvlNode d) {
    return d == null ? -1 : d.height;
}
int bf(AvlNode d) {
    return h(d.left) - h(d.right);
}
void update(AvlNode d) {
    d.height = 1 +
        Math.max(h(d.left), h(d.right));
}
AvlNode rotateRight(AvlNode y) {
    AvlNode x = y.left;
    y.left = x.right;  // T2 → y.left
    x.right = y;       // y → x.right
    update(y);         // lower node first
    update(x);
    return x;          // new root
}
```

```
AvlNode rebalance(AvlNode d) {
    update(d);
    if (bf(d) > 1) {       // left-heavy
        if (bf(d.left) < 0)   // LR
            d.left = rotateLeft(d.left);
        return rotateRight(d); // LL
    }
    if (bf(d) < -1) {      // right-heavy
        if (bf(d.right) > 0)  // RL
            d.right = rotateRight(d.right);
        return rotateLeft(d);  // RR
    }
    return d;
}
// rotateLeft: mirror image (left ↔ right)
// insert/delete as in a BST; on return:
//   return rebalance(root);
```

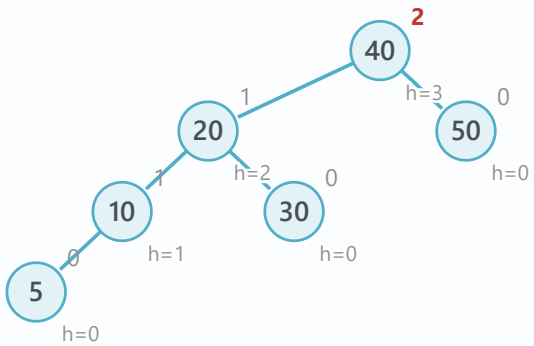
As the recursion unwinds, `rebalance` runs at each node on the path; a deletion may rotate at several nodes (at most O(log n)).

Code Trace: `rotateRight(40)` — the LL Case

```
▶ AvlNode rotateRight(AvlNode y) {
2     AvlNode x = y.left;
3     y.left = x.right;  // T2 → y.left
4     x.right = y;       // y → x.right
5     update(y);         // lower node first
6     update(x);
7     return x;          // new root
8 }
9 void update(AvlNode d) {
10    d.height = 1 +
11        Math.max(h(d.left), h(d.right));
12 }
```

VARIABLES

`y = node(40)`



After inserting 5, rebalance(40) finds bf = 2; its left child 20 has bf = 1 → **LL**. Call `rotateRight(y)`.

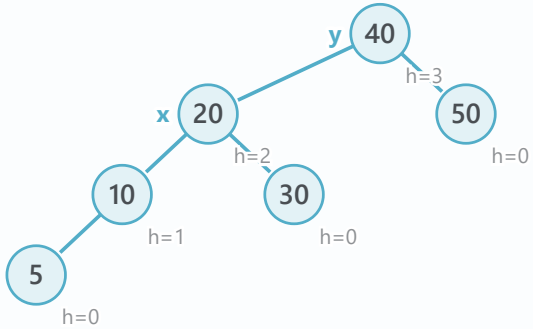
Step 1 / 7

CALL STACK

- `rotateRight(40)`
- `rebalance(40)`
- `insert(40, 5)`

Code Trace: `rotateRight(40)` — the LL Case

```
1  AvlNode rotateRight(AvlNode y) {
▶      AvlNode x = y.left;
3      y.left = x.right;  // T2 → y.left
4      x.right = y;       // y → x.right
5      update(y);         // lower node first
6      update(x);
7      return x;          // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```



x = y.left: 20 is the child that will move up. No link has changed yet.

Step 2 / 7

VARIABLES

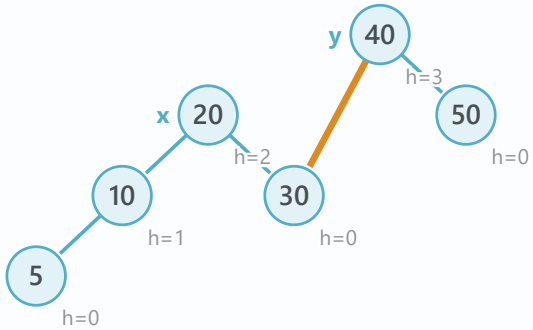
`y = node(40)` `x = node(20)` `y.left = node(20)` `x.right = node(30)` `y.height = 3` `x.height = 2`

CALL STACK

- `rotateRight(40)`
- `rebalance(40)`
- `insert(40, 5)`

Code Trace: `rotateRight(40)` — the LL Case

```
1  AvlNode rotateRight(AvlNode y) {
2      AvlNode x = y.left;
▶      y.left = x.right;   // T2 → y.left
4      x.right = y;        // y → x.right
5      update(y);          // lower node first
6      update(x);
7      return x;           // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```



y.left = x.right: 40's left link now points to T2 (30). 20 is detached; only **x** still reaches it.

Step 3 / 7

VARIABLES

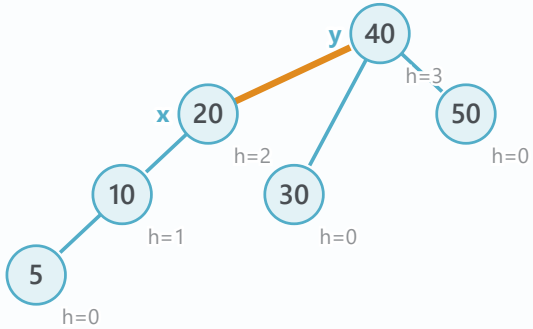
`y = node(40)` `x = node(20)` `y.left = node(30)` `x.right = node(30)` `y.height = 3` `x.height = 2`

CALL STACK

- `rotateRight(40)`
- `rebalance(40)`
- `insert(40, 5)`

Code Trace: `rotateRight(40)` — the LL Case

```
1  AvlNode rotateRight(AvlNode y) {
2      AvlNode x = y.left;
3      y.left = x.right;  // T2 → y.left
▶      x.right = y;       // y → x.right
5      update(y);         // lower node first
6      update(x);
7      return x;          // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```



x.right = y: x.right: 30 → 40. 40 is now 20's right child (the orange arrow points up).

Step 4 / 7

VARIABLES

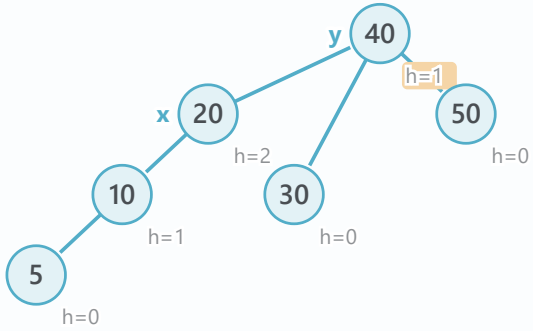
`y = node(40)` `x = node(20)` `y.left = node(30)` `x.right = node(40)` `y.height = 3` `x.height = 2`

CALL STACK

- `rotateRight(40)`
- `rebalance(40)`
- `insert(40, 5)`

Code Trace: `rotateRight(40)` — the LL Case

```
1  AvlNode rotateRight(AvlNode y) {
2      AvlNode x = y.left;
3      y.left = x.right;  // T2 → y.left
4      x.right = y;       // y → x.right
▶      update(y);         // lower node first
6      update(x);
7      return x;          // new root
8  }
9  void update(AvlNode d) {
▶      d.height = 1 +
▶          Math.max(h(d.left), h(d.right));
12 }
```



`update(y)` : 40.height = 1 + max(0, 0) = **1** (was 3). The lower node goes first, since x's height depends on it.

Step 5 / 7

VARIABLES

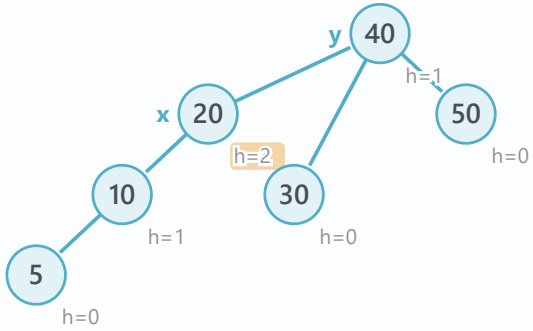
`y = node(40)` `x = node(20)` `y.left = node(30)` `x.right = node(40)` `y.height = 1` `x.height = 2`

CALL STACK

- **update(40)**
- rotateRight(40)
- rebalance(40)
- insert(40, 5)

Code Trace: `rotateRight(40)` — the LL Case

```
1  AvlNode rotateRight(AvlNode y) {
2      AvlNode x = y.left;
3      y.left = x.right;  // T2 → y.left
4      x.right = y;       // y → x.right
5      update(y);         // lower node first
▶      update(x);
7      return x;          // new root
8  }
9  void update(AvlNode d) {
▶      d.height = 1 +
▶          Math.max(h(d.left), h(d.right));
12 }
```



`update(x)` : 20.height = 1 + max(1, 1) = **2** (unchanged, but must be recomputed).

Step 6 / 7

VARIABLES

`y = node(40)` `x = node(20)` `y.left = node(30)` `x.right = node(40)` `y.height = 1` `x.height = 2`

CALL STACK

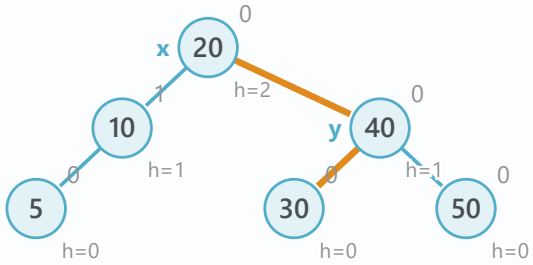
- **update(20)**
- rotateRight(40)
- rebalance(40)
- insert(40, 5)

Code Trace: `rotateRight(40)` — the LL Case

```
1  AvlNode rotateRight(AvlNode y) {
2      AvlNode x = y.left;
3      y.left = x.right;  // T2 → y.left
4      x.right = y;       // y → x.right
5      update(y);         // lower node first
6      update(x);
▶      return x;          // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```

VARIABLES

`y = node(40)` `x = node(20)` `y.left = node(30)` `x.right = node(40)` `y.height = 1` `x.height = 2`



return x: 20 is the subtree's new root (same links, redrawn). All bf ∈ {−1, 0, 1}; inorder sequence unchanged. The method changed 2 links; the caller sets the 3rd (from the parent) → **O(1)**.

Step 7 / 7

CALL STACK

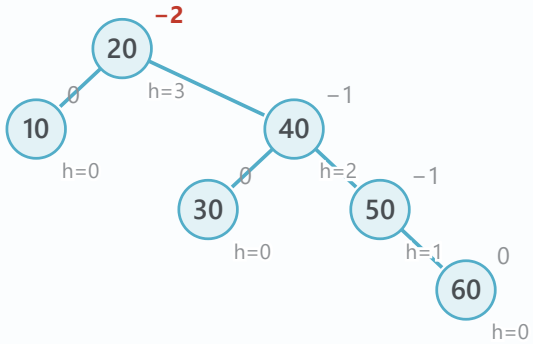
- `rebalance(40)`
- `insert(40, 5)`

Code Trace: `rotateLeft(20)` — the RR Case

```
▶ AvlNode rotateLeft(AvlNode x) {
2     AvlNode y = x.right;
3     x.right = y.left;  // T2 → x.right
4     y.left = x;        // x → y.left
5     update(x);         // lower node first
6     update(y);
7     return y;          // new root
8 }
9 void update(AvlNode d) {
10    d.height = 1 +
11        Math.max(h(d.left), h(d.right));
12 }
```

VARIABLES

`x = node(20)`



After inserting 60, rebalance(20) finds bf = −2; its right child 40 has bf = −1 → **RR**. Call `rotateLeft(x)`.

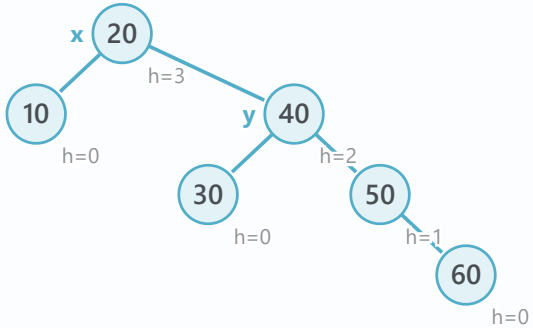
Step 1 / 7

CALL STACK

- `rotateLeft(20)`
- `rebalance(20)`
- `insert(20, 60)`

Code Trace: `rotateLeft(20)` — the RR Case

```
1  AvlNode rotateLeft(AvlNode x) {
▶      AvlNode y = x.right;
3      x.right = y.left;  // T2 → x.right
4      y.left = x;        // x → y.left
5      update(x);         // lower node first
6      update(y);
7      return y;          // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```



y = x.right: 40 is the child that will move up. No link has changed yet.

Step 2 / 7

VARIABLES

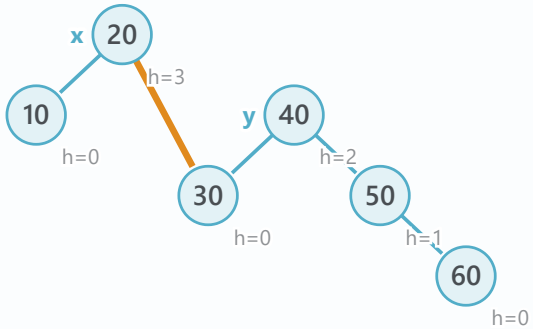
`x = node(20)` `y = node(40)` `x.right = node(40)` `y.left = node(30)` `x.height = 3` `y.height = 2`

CALL STACK

- **`rotateLeft(20)`**
- `rebalance(20)`
- `insert(20, 60)`

Code Trace: `rotateLeft(20)` — the RR Case

```
1  AvlNode rotateLeft(AvlNode x) {
2      AvlNode y = x.right;
▶      x.right = y.left;   // T2 → x.right
4      y.left = x;         // x → y.left
5      update(x);          // lower node first
6      update(y);
7      return y;           // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```



x.right = y.left: 20's right link now points to T2 (30). 40 is detached; only **y** still reaches it.

Step 3 / 7

VARIABLES

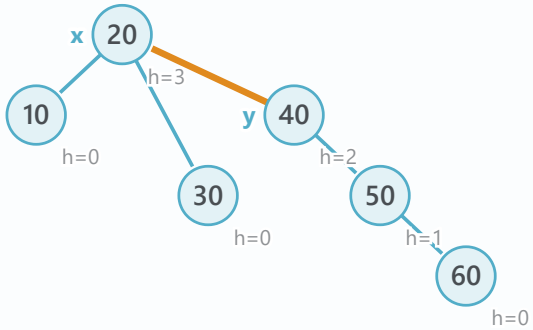
`x = node(20)` `y = node(40)` `x.right = node(30)` `y.left = node(30)` `x.height = 3` `y.height = 2`

CALL STACK

- `rotateLeft(20)`
- `rebalance(20)`
- `insert(20, 60)`

Code Trace: `rotateLeft(20)` — the RR Case

```
1  AvlNode rotateLeft(AvlNode x) {
2      AvlNode y = x.right;
3      x.right = y.left;  // T2 → x.right
▶      y.left = x;        // x → y.left
5      update(x);         // lower node first
6      update(y);
7      return y;          // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```



y.left = x: y.left: 30 → 20. 20 is now 40's left child (the orange arrow points up).

Step 4 / 7

VARIABLES

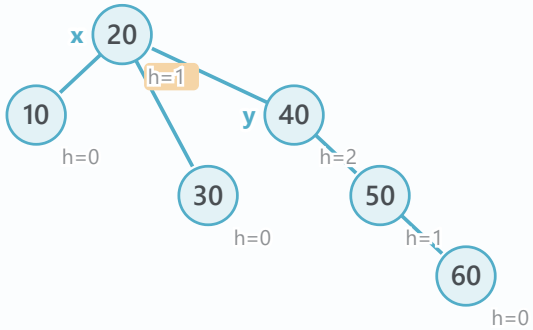
`x = node(20)` `y = node(40)` `x.right = node(30)` `y.left = node(20)` `x.height = 3` `y.height = 2`

CALL STACK

- `rotateLeft(20)`
- `rebalance(20)`
- `insert(20, 60)`

Code Trace: `rotateLeft(20)` — the RR Case

```
1  AvlNode rotateLeft(AvlNode x) {
2      AvlNode y = x.right;
3      x.right = y.left;  // T2 → x.right
4      y.left = x;        // x → y.left
▶      update(x);         // lower node first
6      update(y);
7      return y;          // new root
8  }
9  void update(AvlNode d) {
▶      d.height = 1 +
▶          Math.max(h(d.left), h(d.right));
12 }
```



`update(x)` : 20.height = 1 + max(0, 0) = **1** (was 3). The lower node goes first, since y's height depends on it.

Step 5 / 7

VARIABLES

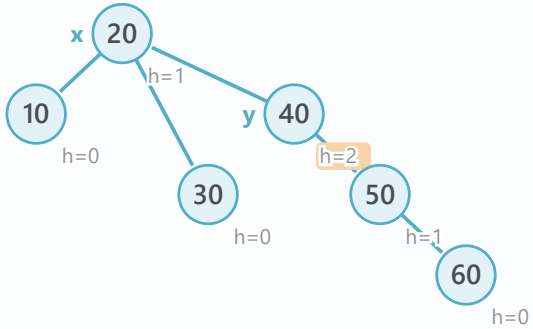
`x` = **node(20)** | `y` = **node(40)** | `x.right` = **node(30)** | `y.left` = **node(20)** | `x.height` = **1** | `y.height` = **2**

CALL STACK

- **update(20)**
- rotateLeft(20)
- rebalance(20)
- insert(20, 60)

Code Trace: `rotateLeft(20)` — the RR Case

```
1  AvlNode rotateLeft(AvlNode x) {
2      AvlNode y = x.right;
3      x.right = y.left;  // T2 → x.right
4      y.left = x;        // x → y.left
5      update(x);         // lower node first
▶      update(y);
7      return y;          // new root
8  }
9  void update(AvlNode d) {
▶      d.height = 1 +
▶          Math.max(h(d.left), h(d.right));
12 }
```



`update(y)` : 40.height = 1 + max(1, 1) = **2** (unchanged, but must be recomputed).

Step 6 / 7

VARIABLES

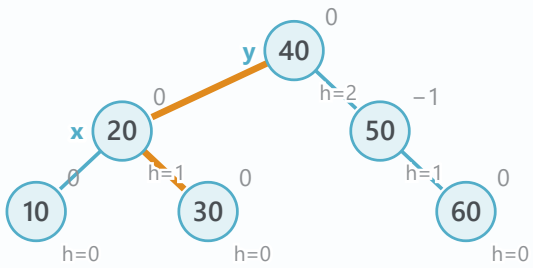
`x = node(20)` `y = node(40)` `x.right = node(30)` `y.left = node(20)` `x.height = 1` `y.height = 2`

CALL STACK

- **update(40)**
- rotateLeft(20)
- rebalance(20)
- insert(20, 60)

Code Trace: `rotateLeft(20)` — the RR Case

```
1  AvlNode rotateLeft(AvlNode x) {
2      AvlNode y = x.right;
3      x.right = y.left;  // T2 → x.right
4      y.left = x;        // x → y.left
5      update(x);         // lower node first
6      update(y);
▶      return y;          // new root
8  }
9  void update(AvlNode d) {
10     d.height = 1 +
11         Math.max(h(d.left), h(d.right));
12 }
```



return y: 40 is the subtree's new root (same links, redrawn). All bf ∈ {−1, 0, 1}; inorder sequence unchanged. The method changed 2 links; the caller sets the 3rd (from the parent) → **O(1)**.

Step 7 / 7

VARIABLES

`x = node(20)` `y = node(40)` `x.right = node(30)` `y.left = node(20)` `x.height = 1` `y.height = 2`

CALL STACK

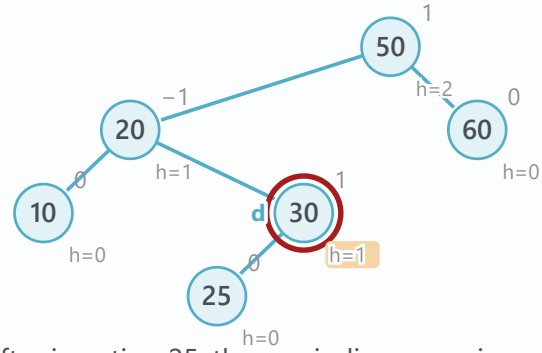
- `rebalance(20)`
- `insert(20, 60)`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
▶      update(d);
3      if (bf(d) > 1) {          // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {         // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`d = node(30)` `d.height = 1` `bf(d) = 1`



After inserting 25, the unwinding recursion calls `rebalance(30)`. update: height 0 → **1**, bf = 1.

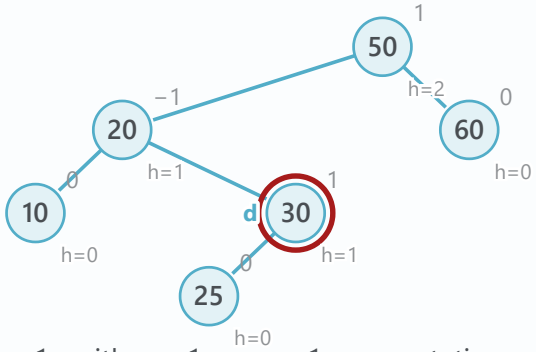
Step 1 / 10

CALL STACK

- **`rebalance(30)`**
- `insert(30, 25)`
- `insert(20, 25)`
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
2      update(d);
▶      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
▶      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
▶      return d;
14 }
```



bf = 1: neither > 1 nor < −1 → no rotation; **return d** to the caller.

Step 2 / 10

VARIABLES

`d = node(30)` `d.height = 1` `bf(d) = 1`

CALL STACK

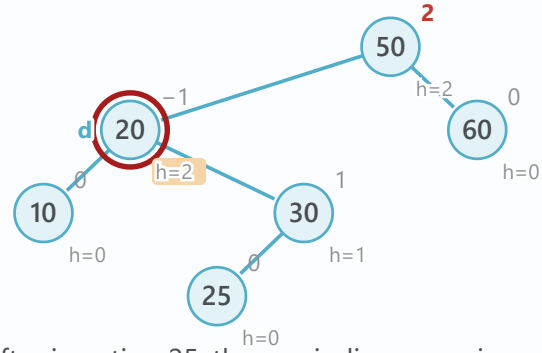
- **`rebalance(30)`**
- `insert(30, 25)`
- `insert(20, 25)`
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
▶      update(d);
3      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`d = node(20)` `d.height = 2` `bf(d) = -1`



After inserting 25, the unwinding recursion calls `rebalance(20)`. update: height 1 → **2**, bf = −1.

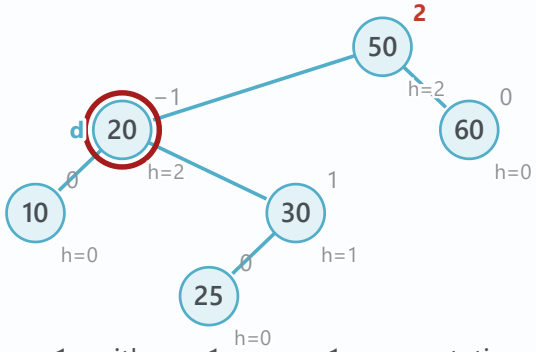
Step 3 / 10

CALL STACK

- **`rebalance(20)`**
- `insert(20, 25)`
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
2      update(d);
▶      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
▶      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
▶      return d;
14 }
```



bf = −1: neither > 1 nor < −1 → no rotation; **return d** to the caller.

Step 4 / 10

VARIABLES

`d = node(20)` `d.height = 2` `bf(d) = -1`

CALL STACK

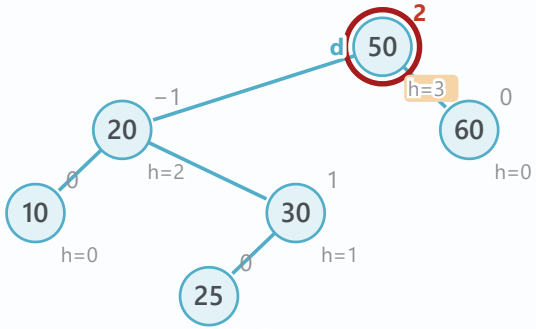
- **`rebalance(20)`**
- `insert(20, 25)`
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
▶      update(d);
3      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`d = node(50)` `d.height = 3` `bf(d) = 2`



`rebalance(50)`: update → height **3**, bf = h(20) − h(60) = 2 − 0 = **2**.

Step 5 / 10

CALL STACK

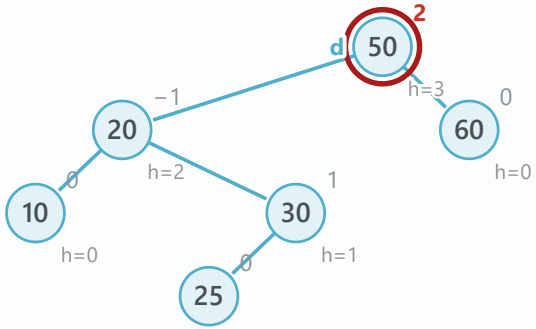
- **`rebalance(50)`**
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
2      update(d);
▶      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`d = node(50)` `d.height = 3` `bf(d) = 2`



bf(d) = 2 > 1 → **left-heavy**. Which rotation? Look at how the left child leans.

Step 6 / 10

CALL STACK

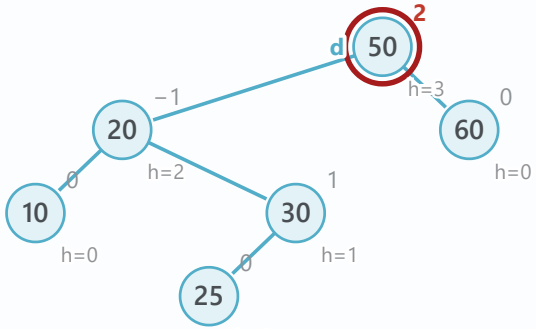
- `rebalance(50)`
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
2      update(d);
3      if (bf(d) > 1) {          // left-heavy
▶          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {         // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`d = node(50)` `d.height = 3` `bf(d) = 2` `bf(d.left) = -1`



bf(d.left) = bf(20) = -1 < 0: the left child leans **right** → **LR** (a triangle). A single right rotation is not enough.

Step 7 / 10

CALL STACK

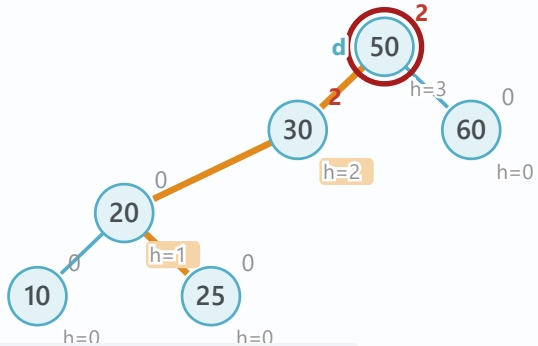
- **`rebalance(50)`**
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
2      update(d);
3      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
▶              d.left = rotateLeft(d.left);
6          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`d = node(50)` `d.height = 3` `bf(d) = 2` `bf(d.left) = 2`



`d.left = rotateLeft(20)` : 30 moves up and 25 moves to 20's right. The triangle is now a straight line (LL); bf(50) is still 2.

Step 8 / 10

CALL STACK

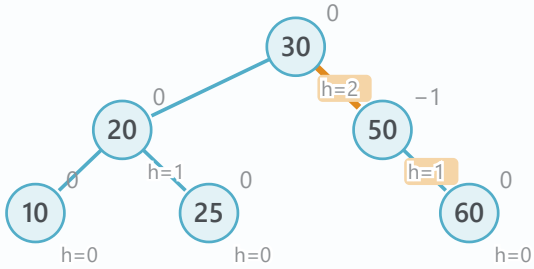
- **`rotateLeft(20)`**
- `rebalance(50)`
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
2      update(d);
3      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
▶          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`d = node(50)` `d.height = 1` `bf(d) = -1`



`return rotateRight(50)` : 30 becomes the root, 50 drops to its right, and 30's old right (null) becomes 50's left.

Step 9 / 10

CALL STACK

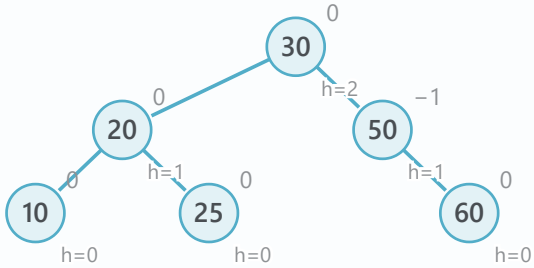
- **`rotateRight(50)`**
- `rebalance(50)`
- `insert(50, 25)`
- `main()`

Code Trace: `rebalance(50)` — the LR Case

```
1  AvlNode rebalance(AvlNode d) {
2      update(d);
3      if (bf(d) > 1) {        // left-heavy
4          if (bf(d.left) < 0)   // LR
5              d.left = rotateLeft(d.left);
▶          return rotateRight(d); // LL
7      }
8      if (bf(d) < -1) {       // right-heavy
9          if (bf(d.right) > 0)  // RL
10             d.right = rotateRight(d.right);
11         return rotateLeft(d);  // RR
12     }
13     return d;
14 }
```

VARIABLES

`returned = node(30)` `bf(30) = 0`



`rebalance(50)` returns the new subtree root **30**; `insert` hands it to its caller (here `root`). Height drops from 3 to 2.

Step 10 / 10

CALL STACK

`insert(50, 25)`

`main()`

Tracing AVL Insertion: 10, 20, 30



Empty tree: `insert(null, 10)` → d == null, a new node is returned; **root = 10**, bf = 0.

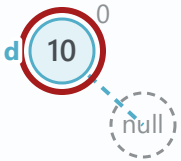
Step 1 / 8

```
1 AvlNode insert(AvlNode d, int k) {
▶     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;          // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

k = **10** d = **null**

Tracing AVL Insertion: 10, 20, 30



`insert(root, 20)` descends as in a BST: 20 > 10 → right.

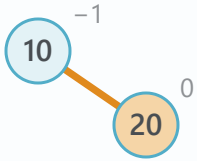
Step 2 / 8

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
▶     else if (k > d.key)
▶         d.right = insert(d.right, k);
7     else return d;           // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

`k = 20` `d = node(10)`

Tracing AVL Insertion: 10, 20, 30



d == null → new leaf **20** is returned and becomes 10's right child. The bf values on the path have changed.

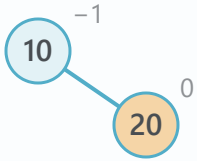
Step 3 / 8

```
1 AvlNode insert(AvlNode d, int k) {
▶       if (d == null) return new AvlNode(k);
3       if (k < d.key)
4           d.left = insert(d.left, k);
5       else if (k > d.key)
6           d.right = insert(d.right, k);
7       else return d;          // no duplicates
8       return rebalance(d);
9   }
```

VARIABLES

k = **20** d = **null**

Tracing AVL Insertion: 10, 20, 30



Unwinding, each ancestor runs `return rebalance(d)`: bf(10) = −1 → all |bf| ≤ 1, no rotation.

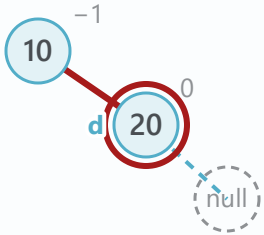
Step 4 / 8

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;          // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 20` `d = node(10)` `bf(d) = −1`

Tracing AVL Insertion: 10, 20, 30



`insert(root, 30)` descends as in a BST: 30 > 10 → right; 30 > 20 → right.

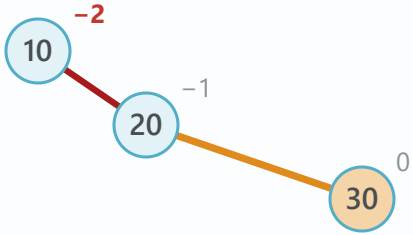
Step 5 / 8

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
▶     else if (k > d.key)
▶         d.right = insert(d.right, k);
7     else return d;           // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

`k = 30` `d = node(20)`

Tracing AVL Insertion: 10, 20, 30



d == null → new leaf **30** is returned and becomes 20's right child. The bf values on the path have changed.

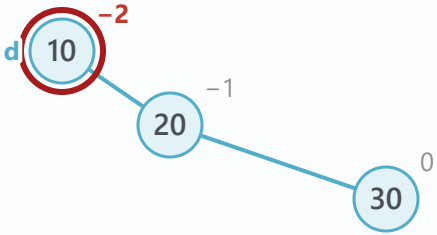
Step 6 / 8

```
1 AvlNode insert(AvlNode d, int k) {
▶     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;          // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

k = 30 d = null

Tracing AVL Insertion: 10, 20, 30



`rebalance(10)` : bf = −2 ✗ (lowest unbalanced ancestor). Right child 20 has bf = −1 → **RR case**.

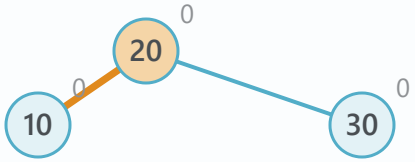
Step 7 / 8

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;          // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 30` `d = node(10)` `bf(d) = −2` `bf(child) = −1`

Tracing AVL Insertion: 10, 20, 30



rotateLeft at 10: 20 moves up and 10 becomes its left child. All bf ∈ {−1, 0, 1}.

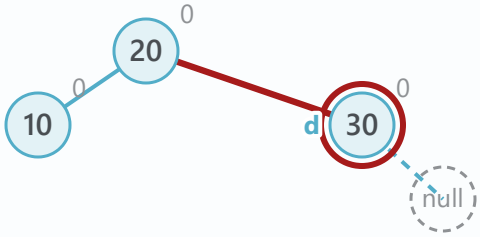
Step 8 / 8

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;           // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 30` `d = node(20)` `bf(d) = 0`

Tracing AVL Insertion (continued): 40, 50, 25



`insert(root, 40)` descends as in a BST: 40 > 20 → right; 40 > 30 → right.

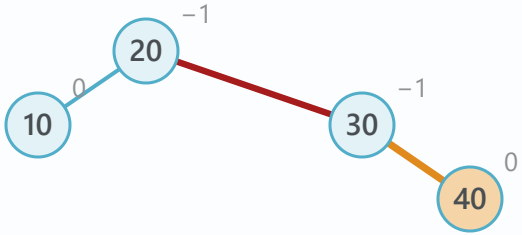
Step 1 / 12

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
▶     else if (k > d.key)
▶         d.right = insert(d.right, k);
7     else return d;           // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

k = **40** d = **node(30)**

Tracing AVL Insertion (continued): 40, 50, 25



d == null → new leaf **40** is returned and becomes 30's right child. The bf values on the path have changed.

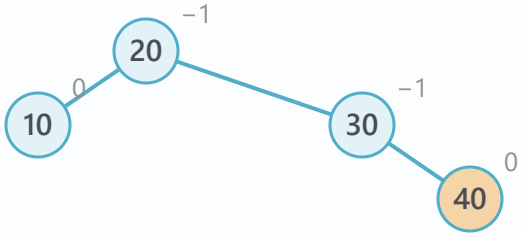
Step 2 / 12

```
1 AvlNode insert(AvlNode d, int k) {
▶     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;          // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

k = **40** d = **null**

Tracing AVL Insertion (continued): 40, 50, 25



Unwinding, each ancestor runs `return rebalance(d)` : bf(30) = −1, bf(20) = −1 → all |bf| ≤ 1, no rotation.

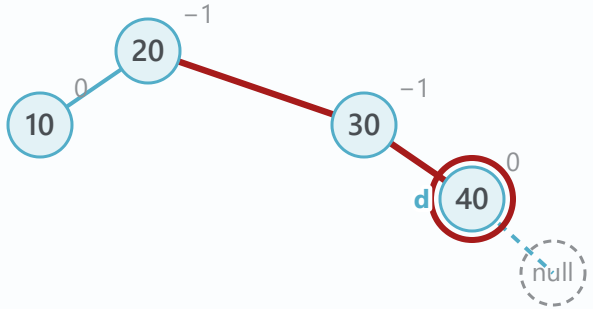
Step 3 / 12

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;           // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 40` `d = node(20)` `bf(d) = -1`

Tracing AVL Insertion (continued): 40, 50, 25



`insert(root, 50)` descends as in a BST: 50 > 20 → right; 50 > 30 → right; 50 > 40 → right.

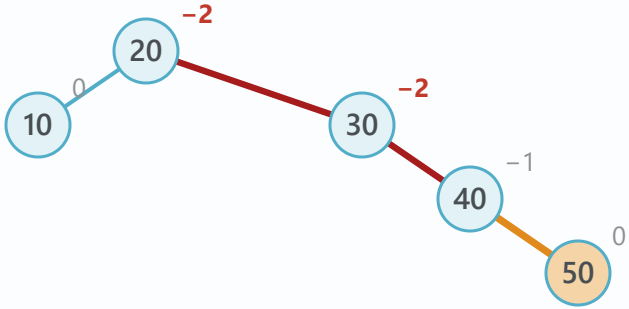
Step 4 / 12

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
▶     else if (k > d.key)
▶         d.right = insert(d.right, k);
7     else return d;           // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

`k = 50` `d = node(40)`

Tracing AVL Insertion (continued): 40, 50, 25



d == null → new leaf **50** is returned and becomes 40's right child. The bf values on the path have changed.

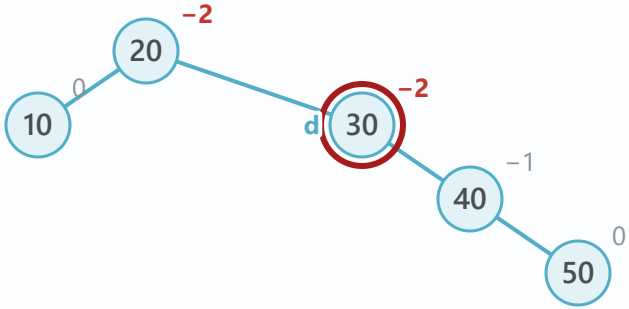
Step 5 / 12

```
1 AvlNode insert(AvlNode d, int k) {
▶       if (d == null) return new AvlNode(k);
3       if (k < d.key)
4           d.left = insert(d.left, k);
5       else if (k > d.key)
6           d.right = insert(d.right, k);
7       else return d;          // no duplicates
8       return rebalance(d);
9   }
```

VARIABLES

k = **50** d = **null**

Tracing AVL Insertion (continued): 40, 50, 25



`rebalance(30)`: bf = −2 ✗ (lowest unbalanced ancestor). Right child 40 has bf = −1 → **RR case**.

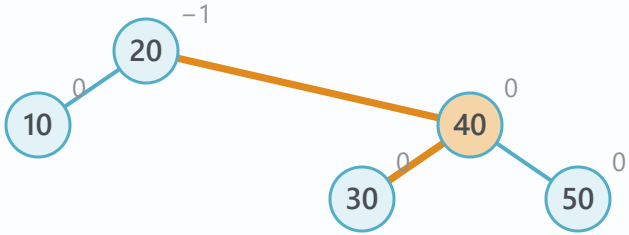
Step 6 / 12

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;           // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 50` `d = node(30)` `bf(d) = −2` `bf(child) = −1`

Tracing AVL Insertion (continued): 40, 50, 25



rotateLeft at 30: 40 moves up and 30 becomes its left child. All bf ∈ {−1, 0, 1}.

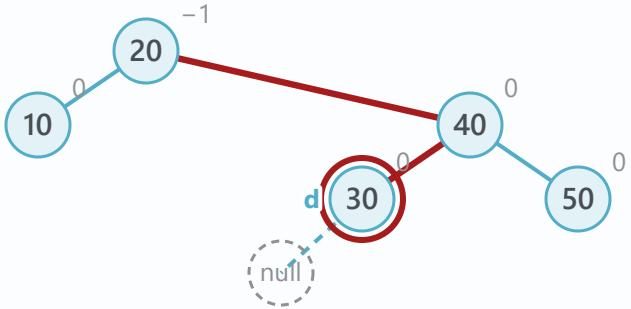
Step 7 / 12

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;          // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 50` `d = node(40)` `bf(d) = 0`

Tracing AVL Insertion (continued): 40, 50, 25



`insert(root, 25)` descends as in a BST: 25 > 20 → right; 25 < 40 → left; 25 < 30 → left.

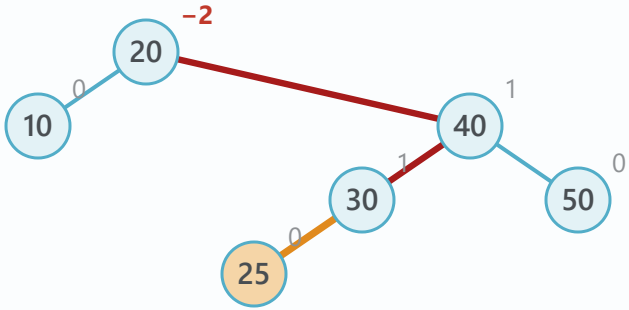
Step 8 / 12

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
▶     if (k < d.key)
▶         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;          // no duplicates
8     return rebalance(d);
9 }
```

VARIABLES

`k = 25` `d = node(30)`

Tracing AVL Insertion (continued): 40, 50, 25



d == null → new leaf **25** is returned and becomes 30's left child. The bf values on the path have changed.

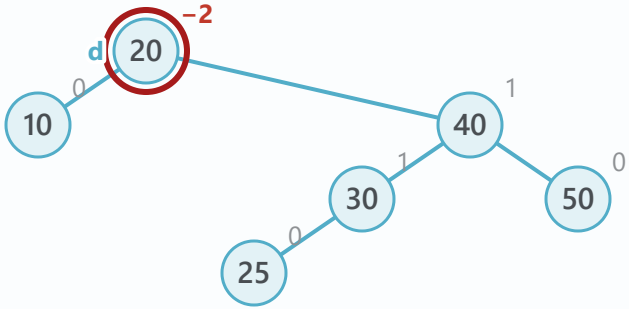
Step 9 / 12

```
1 AvlNode insert(AvlNode d, int k) {
▶       if (d == null) return new AvlNode(k);
3       if (k < d.key)
4           d.left = insert(d.left, k);
5       else if (k > d.key)
6           d.right = insert(d.right, k);
7       else return d;          // no duplicates
8       return rebalance(d);
9   }
```

VARIABLES

k = **25** d = **null**

Tracing AVL Insertion (continued): 40, 50, 25



`rebalance(20)`: bf = −2 ✗ (lowest unbalanced ancestor). Right child 40 has bf = 1 → **RL case**.

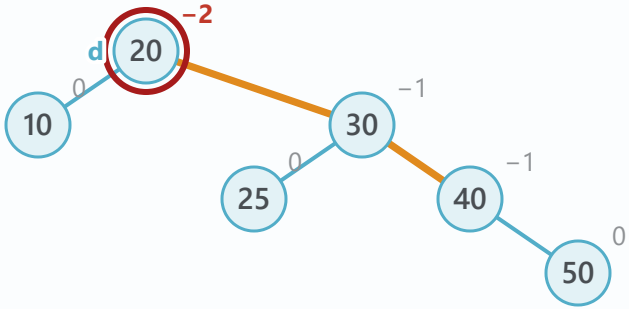
Step 10 / 12

```
1  AvlNode insert(AvlNode d, int k) {
2      if (d == null) return new AvlNode(k);
3      if (k < d.key)
4          d.left = insert(d.left, k);
5      else if (k > d.key)
6          d.right = insert(d.right, k);
7      else return d;           // no duplicates
▶      return rebalance(d);
9  }
```

VARIABLES

`k = 25` `d = node(20)` `bf(d) = −2` `bf(child) = 1`

Tracing AVL Insertion (continued): 40, 50, 25



Rotation 1: **rotateRight** at 40 → the triangle becomes a straight line (right-right). bf(20) is still −2.

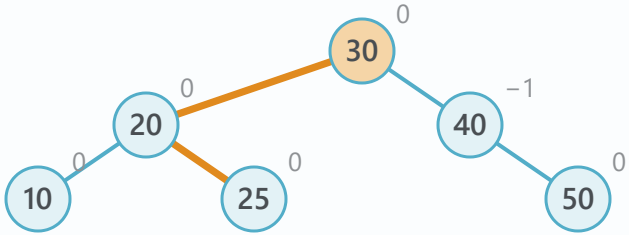
Step 11 / 12

```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;           // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 25` `d = node(20)` `bf(d) = −2`

Tracing AVL Insertion (continued): 40, 50, 25



Rotation 2: **rotateLeft** at 20: 30 moves up and 25 moves to 20's right. The tree is AVL again. All bf ∈ {−1, 0, 1}.

Step 12 / 12

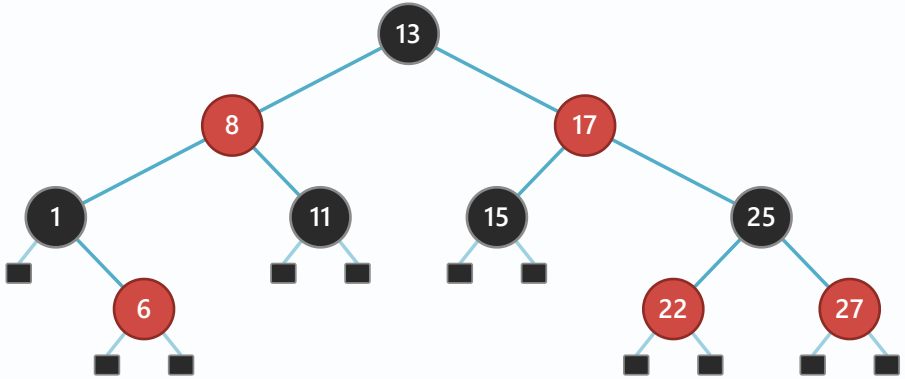
```
1 AvlNode insert(AvlNode d, int k) {
2     if (d == null) return new AvlNode(k);
3     if (k < d.key)
4         d.left = insert(d.left, k);
5     else if (k > d.key)
6         d.right = insert(d.right, k);
7     else return d;           // no duplicates
▶     return rebalance(d);
9 }
```

VARIABLES

`k = 25` `d = node(30)` `bf(d) = 0`

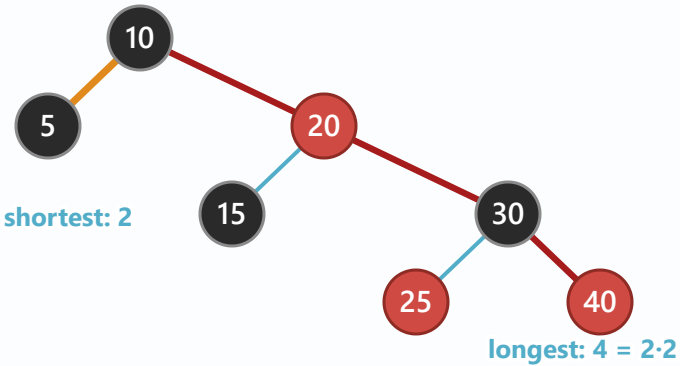
Red-Black Trees

1. Every node is **red** or **black**.
2. The **root** is black.
3. All leaves (**NIL**, the empty children) count as black.
4. A red node's children are black: **no two reds in a row**.
5. **Every path** from a node down to its NIL leaves has **the same number of black nodes** (the black height).



Example: every path from the root to a NIL has 2 black nodes (counting the root, not the NIL)

Why Do the Rules Guarantee Balance?



- **Shortest path:** can be all black → b nodes (10–5 in the figure: b = 2).
- **Longest path:** reds cannot be adjacent, so at most one red per black → at most 2b nodes (10–20–30–40).
- **Consequence:** longest path ≤ 2 × shortest path, which gives $\mathbf{h \le 2 \cdot \log_2(n + 1)}$.
- **Cost:** search, insertion, deletion in O(log n); at most 2 rotations per insertion, 3 per deletion; the rest is recoloring.

A red-black tree is balanced **more loosely** than an AVL tree (it may be a bit taller), but updates are cheaper. Java's `TreeMap` / `TreeSet` and C++'s `std::map` use it.

Inserting into a Red-Black Tree

Insert the new node as in a BST and color it **red** (leaving the black height intact). If its parent is red too, rule 4 breaks; look at the **uncle** (the parent's sibling):

Red uncle

Recolor: parent and uncle black, grandparent red. The problem may move up → continue from the grandparent.

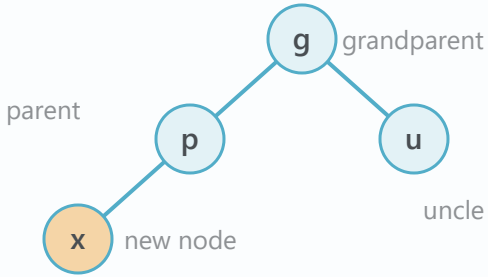
Black uncle, triangle

If node, parent, and grandparent zigzag, rotate at the parent → the line case.

Black uncle, line

Rotate at the grandparent; the old parent becomes black and the old grandparent red. Done.

- If the parent is black, nothing needs doing. At the end, the root is always colored black.
- A NIL uncle counts as black.
- Deletion is similar but has more cases (the "double black" fix).



Fixing Up After Insertion

```
void fixUp(Node x) {       // x: new, red
    while (x != root && isRed(x.parent)) {
        Node p = x.parent, g = p.parent;
        Node u = (p == g.left) ? g.right : g.left;
        if (isRed(u)) {                // red uncle
            p.color = u.color = BLACK;
            g.color = RED;
            x = g;                     // move up to g
        } else {                       // black uncle
            if (isTriangle(x, p, g)) { // zigzag
                rotate(p, x);          // make it a line
                Node t = x; x = p; p = t;
            }
            rotate(g, p);              // line
            p.color = BLACK; g.color = RED;
        }
    }
    root.color = BLACK;
}
```

```
// Insertion: add a leaf as in a BST,
// link its parent, color it red:
//   x = bstInsert(key);
//   x.color = RED; fixUp(x);

// NIL (null) counts as black
boolean isRed(Node n) {
    return n != null
        && n.color == RED;
}
// is x–p–g a zigzag (triangle)?
boolean isTriangle(Node x, Node p,
                   Node g) {
    return (p == g.left)
        != (x == p.left);
}
// rotate(a, b): b is a child of a;
// moves b into a's place (rotate
// right if b is a left child, else
// left; parent links included)
```

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {      // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



10 is inserted as in a BST (as the root) and colored **red**; then `fixUp(x)`.

Step 1 / 19

VARIABLES

`x = node(10) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {      // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x is the root → the loop condition is false; the loop is skipped.

Step 2 / 19

VARIABLES

`x = node(10) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {      // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
▶      root.color = BLACK;
19 }
```



root.color = BLACK: the root is always black (rule 2).

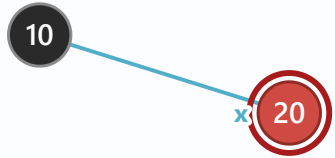
Step 3 / 19

VARIABLES

`x = node(10) B`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
▶ void fixUp(Node x) {      // x: new, red
2     while (x != root && isRed(x.parent)) {
3         Node p = x.parent, g = p.parent;
4         Node u = (p == g.left) ? g.right : g.left;
5         if (isRed(u)) {                // red uncle
6             p.color = u.color = BLACK;
7             g.color = RED;
8             x = g;                     // move up to
  g
9         } else {                       // black uncle
10            if (isTriangle(x, p, g)) { // zigzag
11                rotate(p, x);          // make it a
  line
12                Node t = x; x = p; p = t;
13            }
14            rotate(g, p);              // line
15            p.color = BLACK; g.color = RED;
16        }
17    }
18    root.color = BLACK;
19 }
```



20 is inserted as in a BST (10's right child) and colored **red**; then `fixUp(x)`.

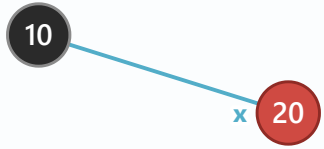
Step 4 / 19

VARIABLES

`x = node(20) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {      // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
▶      root.color = BLACK;
19 }
```



x.parent (10) is **black** → no red-red pair; the loop ends. Line 18: the root is already black.

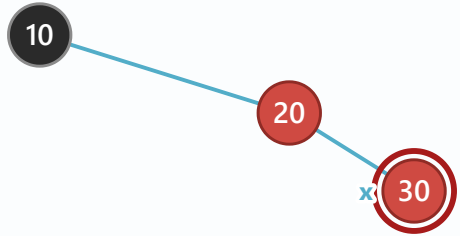
Step 5 / 19

VARIABLES

`x = node(20) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
▶ void fixUp(Node x) {        // x: new, red
2     while (x != root && isRed(x.parent)) {
3         Node p = x.parent, g = p.parent;
4         Node u = (p == g.left) ? g.right : g.left;
5         if (isRed(u)) {                 // red uncle
6             p.color = u.color = BLACK;
7             g.color = RED;
8             x = g;                      // move up to
  g
9         } else {                        // black uncle
10            if (isTriangle(x, p, g)) { // zigzag
11                rotate(p, x);           // make it a
  line
12                Node t = x; x = p; p = t;
13            }
14            rotate(g, p);               // line
15            p.color = BLACK; g.color = RED;
16        }
17    }
18    root.color = BLACK;
19 }
```



30 is inserted as in a BST (20's right child) and colored **red**; then `fixUp(x)`.

Step 6 / 19

VARIABLES

`x = node(30) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {      // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x.parent (20) is **red** too: two reds in a row ✗ (rule 4) → enter the loop.

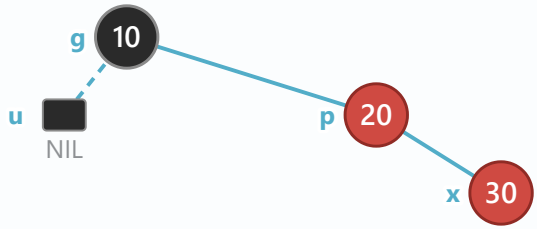
Step 7 / 19

VARIABLES

`x = node(30) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
▶          Node p = x.parent, g = p.parent;
▶          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



p = parent (20), g = grandparent (10), u = uncle = NIL (empty, counts as black).

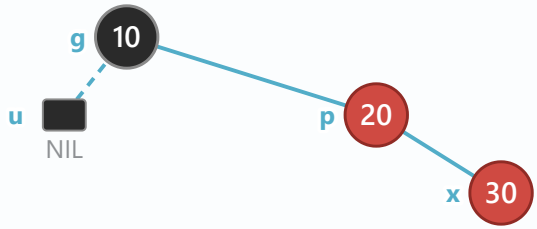
Step 8 / 19

VARIABLES

`x = node(30) R` `p = node(20) R` `g = node(10) B` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
▶          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
▶          } else {                         // black uncle
▶              if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



The uncle is **black** → a rotation is needed. x–p–g is a **straight line**: isTriangle() is false; jump to line 14.

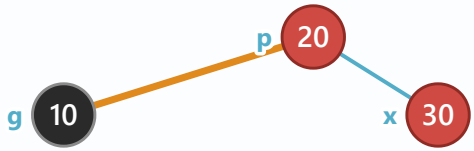
Step 9 / 19

VARIABLES

`x = node(30) R` `p = node(20) R` `g = node(10) B` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
▶              rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



`rotate(g, p)` : **left rotation** at grandparent 10 → p (20) moves up, g becomes its child.

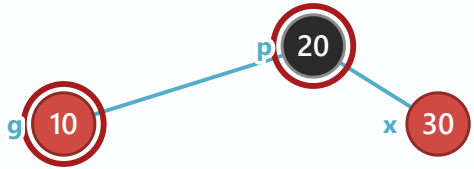
Step 10 / 19

VARIABLES

`x = node(30) R` `p = node(20) R` `g = node(10) B` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {       // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
▶              p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



Colors: **p black, g red**. No red-red pair remains; the black height is preserved.

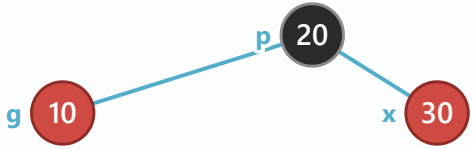
Step 11 / 19

VARIABLES

`x = node(30) R` `p = node(20) B` `g = node(10) R` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {      // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
▶      root.color = BLACK;
19 }
```



x.parent (20) is **black** → no red-red pair; the loop ends. Line 18: the root is already black.

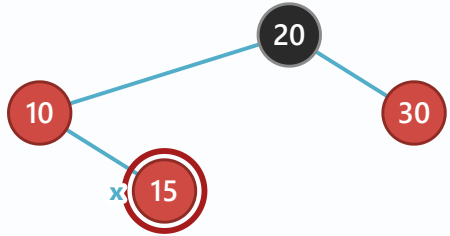
Step 12 / 19

VARIABLES

`x = node(30) R` `p = node(20) B` `g = node(10) R` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
▶ void fixUp(Node x) {      // x: new, red
2     while (x != root && isRed(x.parent)) {
3         Node p = x.parent, g = p.parent;
4         Node u = (p == g.left) ? g.right : g.left;
5         if (isRed(u)) {                 // red uncle
6             p.color = u.color = BLACK;
7             g.color = RED;
8             x = g;                      // move up to
  g
9         } else {                        // black uncle
10            if (isTriangle(x, p, g)) { // zigzag
11                rotate(p, x);           // make it a
  line
12                Node t = x; x = p; p = t;
13            }
14            rotate(g, p);               // line
15            p.color = BLACK; g.color = RED;
16        }
17    }
18    root.color = BLACK;
19 }
```



15 is inserted as in a BST (10's right child) and colored **red**; then `fixUp(x)`.

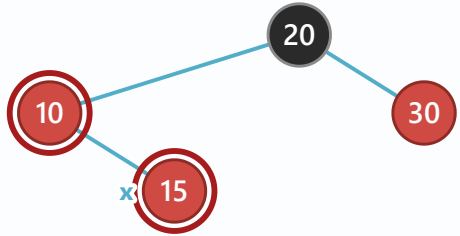
Step 13 / 19

VARIABLES

`x = node(15) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {      // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x.parent (10) is **red** too: two reds in a row ✗ (rule 4) → enter the loop.

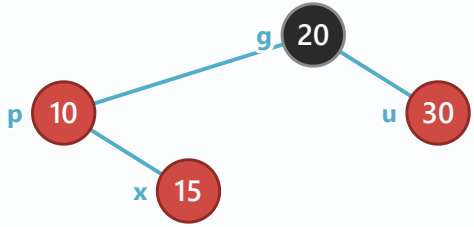
Step 14 / 19

VARIABLES

`x = node(15) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
▶          Node p = x.parent, g = p.parent;
▶          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



p = parent (10), g = grandparent (20), u = uncle = 30.

Step 15 / 19

VARIABLES

x = node(15) R

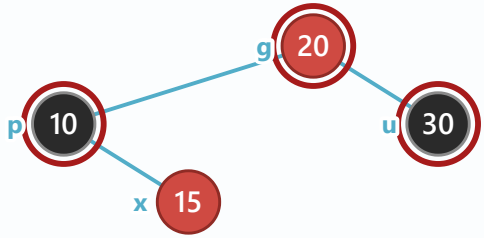
p = node(10) R

g = node(20) B

u = node(30) R

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
▶          if (isRed(u)) {                  // red uncle
▶              p.color = u.color = BLACK;
▶              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



The uncle is red → just recolor: p and u black, g red. Every path keeps the same number of blacks.

Step 16 / 19

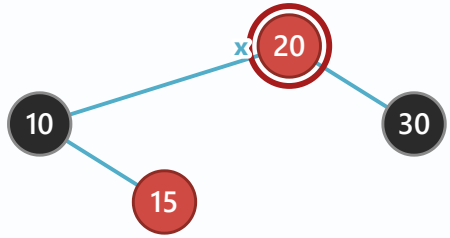
VARIABLES

`x = node(15) R` `p = node(10) B`

`g = node(20) R` `u = node(30) B`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {       // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                 // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
▶              x = g;                      // move up to g
9          } else {                        // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);           // make it a line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);               // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x = g (20): g is now red, and its parent may be red too → the loop continues from there.

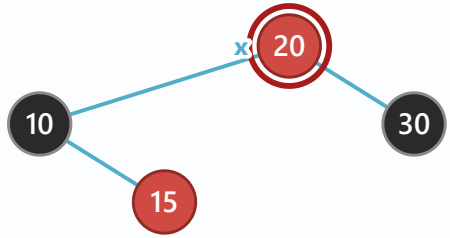
Step 17 / 19

VARIABLES

`x = node(20) R`

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {        // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x is the root → the loop condition is false; the loop is skipped.

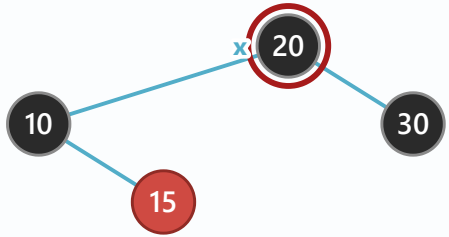
Step 18 / 19

VARIABLES

x = **node(20) R**

Code Trace: `fixUp` — Inserting 10, 20, 30, 15

```
1  void fixUp(Node x) {       // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                     // move up to
   g
9          } else {                       // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);          // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);              // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
▶      root.color = BLACK;
19 }
```



root.color = BLACK: the root is always black (rule 2). The black height grows by 1.

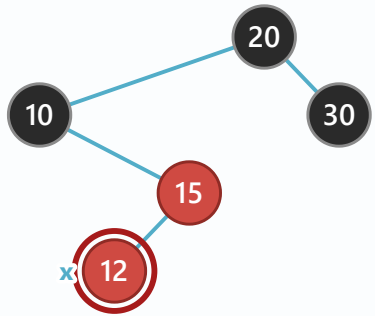
Step 19 / 19

VARIABLES

`x = node(20) B`

Code Trace: `fixUp` — Inserting 12 and 5

```
▶ void fixUp(Node x) {      // x: new, red
2     while (x != root && isRed(x.parent)) {
3         Node p = x.parent, g = p.parent;
4         Node u = (p == g.left) ? g.right : g.left;
5         if (isRed(u)) {                 // red uncle
6             p.color = u.color = BLACK;
7             g.color = RED;
8             x = g;                      // move up to
  g
9         } else {                        // black uncle
10            if (isTriangle(x, p, g)) { // zigzag
11                rotate(p, x);           // make it a
  line
12                Node t = x; x = p; p = t;
13            }
14            rotate(g, p);               // line
15            p.color = BLACK; g.color = RED;
16        }
17    }
18    root.color = BLACK;
19 }
```



12 is inserted as in a BST (15's left child) and colored **red**; then `fixUp(x)`.

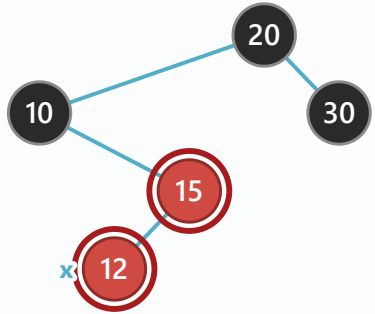
Step 1 / 15

VARIABLES

`x = node(12) R`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                 // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                      // move up to
   g
9          } else {                        // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);           // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);               // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x.parent (15) is **red** too: two reds in a row ✗ (rule 4) → enter the loop.

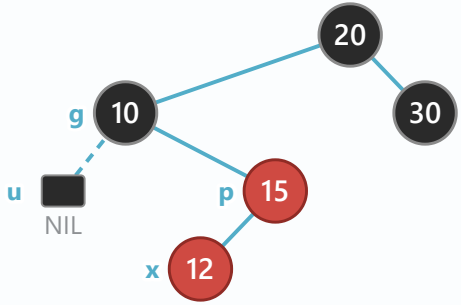
Step 2 / 15

VARIABLES

`x = node(12) R`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
▶          Node p = x.parent, g = p.parent;
▶          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



p = parent (15), g = grandparent (10), u = uncle = NIL (empty, counts as black).

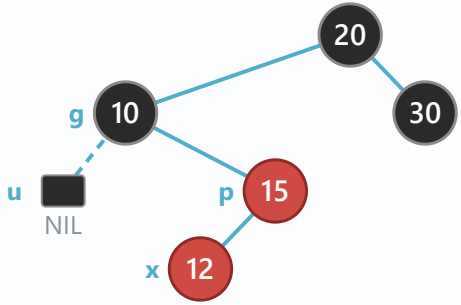
Step 3 / 15

VARIABLES

`x = node(12) R` `p = node(15) R` `g = node(10) B` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
▶          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to g
▶          } else {                         // black uncle
▶              if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



The uncle is **black** → a rotation is needed. x–p–g is a **triangle** (zigzag): x and p are children on opposite sides.

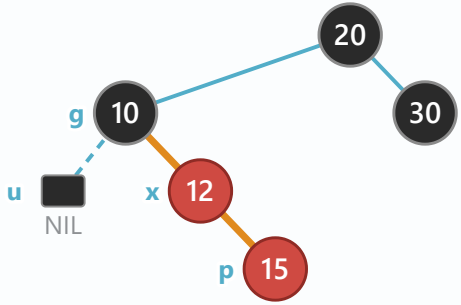
Step 4 / 15

VARIABLES

`x = node(12) R` `p = node(15) R` `g = node(10) B` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
▶                  rotate(p, x);           // make it a line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



`rotate(p, x)` : a right rotation at 15 → x goes up, p goes down. The triangle is now a straight line (still two reds in a row).

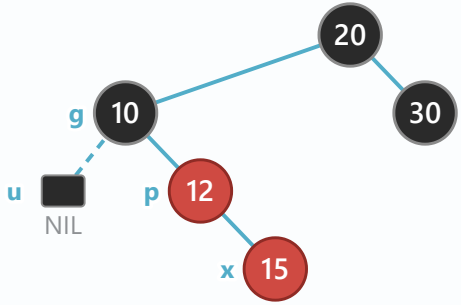
Step 5 / 15

VARIABLES

- x = **node(12) R**
- p = **node(15) R**
- g = **node(10) B**
- u = **null (NIL, B)**

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a line
▶                  Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x and p swap roles: now **x = 15** (the lower one) and **p = 12**.

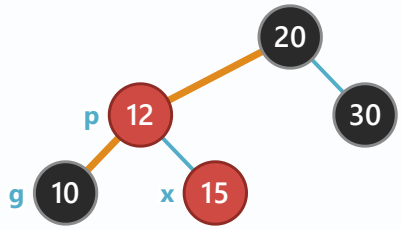
Step 6 / 15

VARIABLES

`x = node(15) R` `p = node(12) R` `g = node(10) B` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
▶              rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



`rotate(g, p)` : **left rotation** at grandparent 10 → p (12) moves up, g becomes its child.

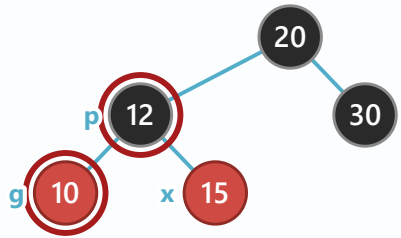
Step 7 / 15

VARIABLES

- x = **node(15) R**
- p = **node(12) R**
- g = **node(10) B**
- u = **null (NIL, B)**

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
▶              p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



Colors: **p black, g red**. No red-red pair remains; the black height is preserved.

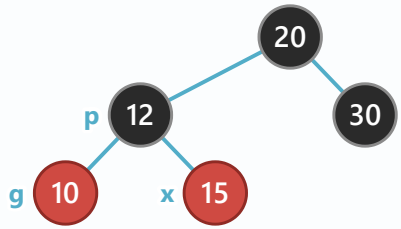
Step 8 / 15

VARIABLES

`x = node(15) R` `p = node(12) B` `g = node(10) R` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                 // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                      // move up to
   g
9          } else {                        // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);           // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);               // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
▶      root.color = BLACK;
19 }
```



x.parent (12) is **black** → no red-red pair; the loop ends. Line 18: the root is already black.

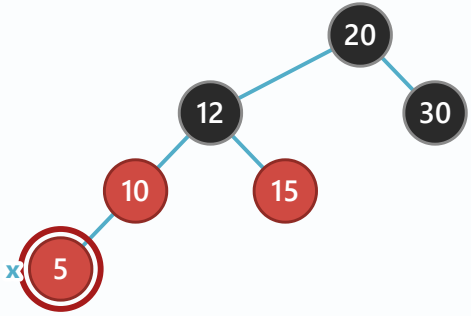
Step 9 / 15

VARIABLES

`x = node(15) R` `p = node(12) B` `g = node(10) R` `u = null (NIL, B)`

Code Trace: `fixUp` — Inserting 12 and 5

```
▶ void fixUp(Node x) {      // x: new, red
2     while (x != root && isRed(x.parent)) {
3         Node p = x.parent, g = p.parent;
4         Node u = (p == g.left) ? g.right : g.left;
5         if (isRed(u)) {                 // red uncle
6             p.color = u.color = BLACK;
7             g.color = RED;
8             x = g;                      // move up to g
9         } else {                        // black uncle
10            if (isTriangle(x, p, g)) { // zigzag
11                rotate(p, x);           // make it a line
12                Node t = x; x = p; p = t;
13            }
14            rotate(g, p);               // line
15            p.color = BLACK; g.color = RED;
16        }
17    }
18    root.color = BLACK;
19 }
```



5 is inserted as in a BST (10's left child) and colored **red**; then `fixUp(x)`.

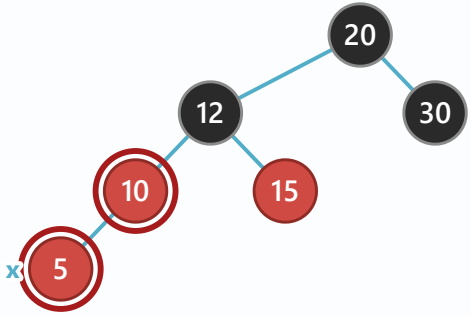
Step 10 / 15

VARIABLES

`x = node(5) R`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                 // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                      // move up to
   g
9          } else {                        // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);           // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);               // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x.parent (10) is **red** too: two reds in a row ✗ (rule 4) → enter the loop.

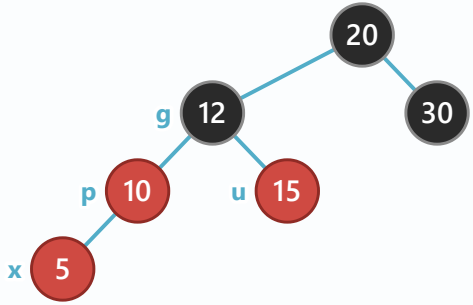
Step 11 / 15

VARIABLES

`x = node(5) R`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
▶          Node p = x.parent, g = p.parent;
▶          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                       // move up to
   g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);            // make it a
   line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



p = parent (10), g = grandparent (12), u = uncle = 15.

Step 12 / 15

VARIABLES

x = node(5) R

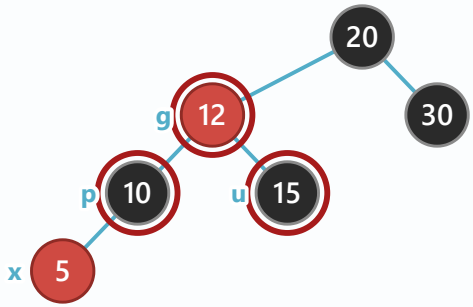
p = node(10) R

g = node(12) B

u = node(15) R

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
▶          if (isRed(u)) {                  // red uncle
▶              p.color = u.color = BLACK;
▶              g.color = RED;
8              x = g;                       // move up to g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) {   // zigzag
11                 rotate(p, x);            // make it a line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);                // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



The uncle is red → just recolor: p and u black, g red. Every path keeps the same number of blacks.

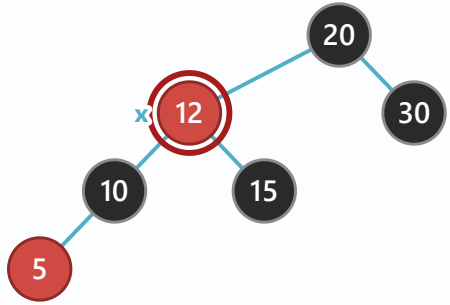
Step 13 / 15

VARIABLES

`x = node(5) R` `p = node(10) B` `g = node(12) R` `u = node(15) B`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
2      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                  // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
▶              x = g;                       // move up to g
9          } else {                         // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);           // make it a line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);               // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
18     root.color = BLACK;
19 }
```



x = g (12): g is now red, and its parent may be red too → the loop continues from there.

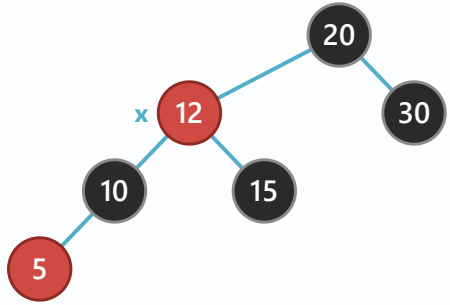
Step 14 / 15

VARIABLES

`x = node(12) R`

Code Trace: `fixUp` — Inserting 12 and 5

```
1  void fixUp(Node x) {        // x: new, red
▶      while (x != root && isRed(x.parent)) {
3          Node p = x.parent, g = p.parent;
4          Node u = (p == g.left) ? g.right : g.left;
5          if (isRed(u)) {                 // red uncle
6              p.color = u.color = BLACK;
7              g.color = RED;
8              x = g;                      // move up to g
9          } else {                        // black uncle
10             if (isTriangle(x, p, g)) { // zigzag
11                 rotate(p, x);           // make it a line
12                 Node t = x; x = p; p = t;
13             }
14             rotate(g, p);               // line
15             p.color = BLACK; g.color = RED;
16         }
17     }
▶      root.color = BLACK;
19 }
```



x.parent (20) is **black** → no red-red pair; the loop ends. Line 18: the root is already black.

Step 15 / 15

VARIABLES

`x = node(12) R`

Deletion in AVL and Red-Black Trees

Common first step

An ordinary BST deletion. A node with two children is replaced by its successor or predecessor, so the actual removal is a leaf or one-child case.

Why the extra work?

Removing a node can break the AVL height condition or the red-black black-height rule.

AVL fix

Update heights from the deleted node up to the root. At every node with |bf| = 2, apply an LL, RR, LR, or RL rotation. Several ancestors may need fixing.

Red-black fix

Deleting a red node is easy. If a black goes missing, recolor or rotate based on the sibling's color and children; the missing black may travel up to the root.

Structure	Upward pass	Rotations	Total
AVL	O(log n)	up to O(log n)	O(log n)
Red-black	O(log n)	at most 3	O(log n)

Insertion and deletion keep the BST order; rebalancing changes only the shape of the tree and its bookkeeping fields.

AVL or Red-Black?

Property	AVL	Red-black
Balance criterion	$\lvert bf \rvert \le 1$ at every node	color rules, equal black height
Maximum height	$\approx 1.44 \log_2 n$	$2 \log_2(n + 1)$
Search	O(log n), slightly faster (shorter tree)	O(log n)
Rotations per insertion	at most 2 (one single or one double)	at most 2
Rotations per deletion	up to O(log n)	at most 3
Extra data per node	height (or bf)	1 color bit
Best for	search-heavy workloads	frequent updates, libraries

Splay Trees

- Sleator and Tarjan (1985): a **self-adjusting** BST.
- Every access (search, insertion, deletion) **rotates the node to the root**: the *splay* operation.
- Nodes store no balance information (no height, no color).
- One operation can take O(n), but any m operations take O(m log n): **O(log n) amortized**.

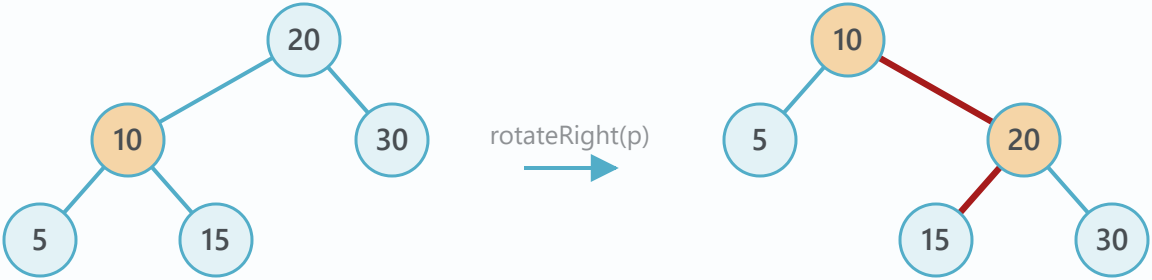
Locality
Frequently accessed keys stay near the root, so the tree acts like a cache.

Simplicity
No extra fields; one splay routine serves every operation.

Drawback
Even a search changes the tree, and a single operation has no guarantee.

Splay Steps: Zig, Zig-Zig, Zig-Zag

x: the node being splayed, p: its parent, g: its grandparent. Repeat until x is the root.



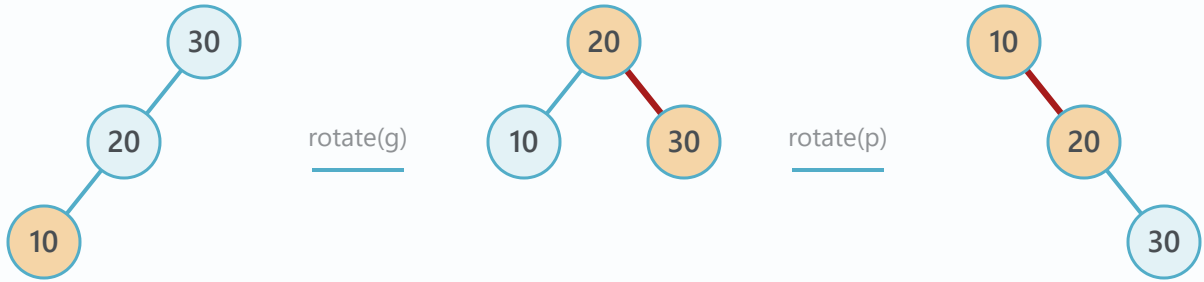
x = 10; parent p = 20 is root

x is root; 15 moved left of p

Zig: x's parent p is the root → a single rotation at p. (This can only be the last step.)

Splay Steps: Zig, Zig-Zig, Zig-Zag

x: the node being splayed, p: its parent, g: its grandparent. Repeat until x is the root.

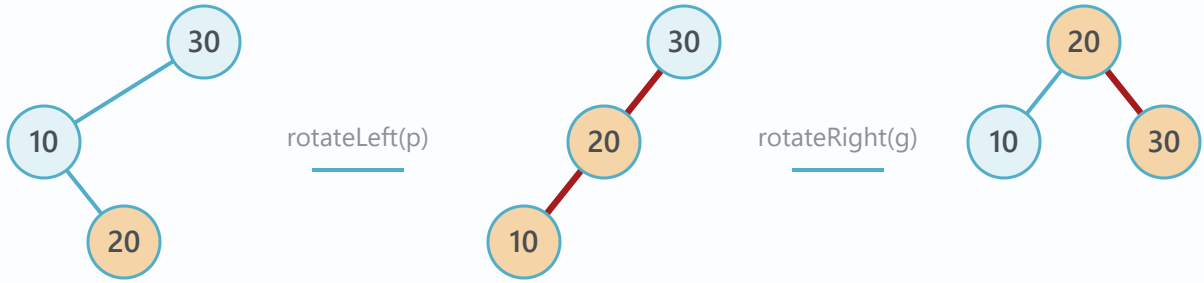


x, p, g in a line (left-left) — grandparent g = 30 first — then parent p = 20

Zig-zig: x and p are children on the same side → rotate **at g first**, then at p. (Not the same as an AVL single rotation!)

Splay Steps: Zig, Zig-Zig, Zig-Zag

x: the node being splayed, p: its parent, g: its grandparent. Repeat until x is the root.



x right, p left child (zigzag) — parent p = 10 first — then grandparent g = 30

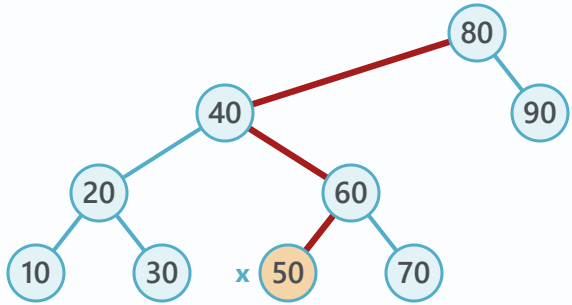
Zig-zag: x and p are on opposite sides → rotate at p, then at g (exactly an AVL double rotation).

Code Trace: `splay(50)`, then `splay(10)`

```
▶ void splay(Node x) {
2     while (x != root) {
3         Node p = x.parent, g = p.parent;
4         if (g == null)                    // zig
5             rotate(p, x);
6         else if (sameSide(x, p, g)) {  // zig-zig
7             rotate(g, p);   // grandparent first
8             rotate(p, x);
9         } else {                          // zig-zag
10            rotate(p, x);   // parent first
11            rotate(g, x);
12        }
13    }
14 }
```

VARIABLES

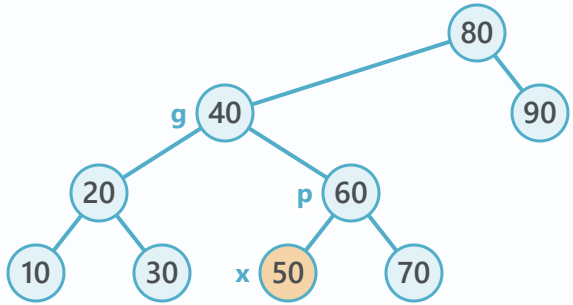
`x = node(50)`



`splay(50)`: first an ordinary BST search, 80 → 40 → 60 → 50; the node found is x. (sameSide: are x and p children on the same side? rotate: the single rotation from the red-black code.)

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
▶      while (x != root) {
▶          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                          // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```



x is not the root. p = 60, g = 40.

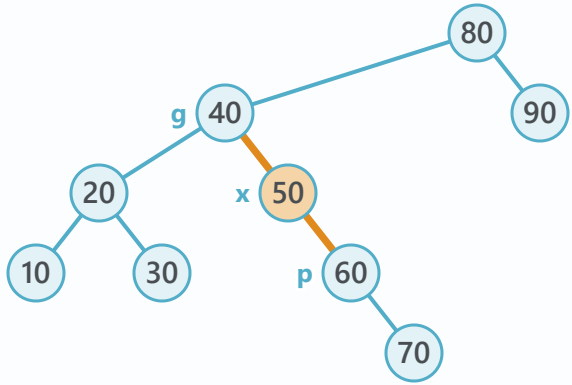
Step 2 / 14

VARIABLES

`x = node(50)` `p = node(60)` `g = node(40)`

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
2      while (x != root) {
3          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
           else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
           } else {                          // zig-zag
               rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```



x and p are on opposite sides → **zig-zag**.
The **parent** first: rotate right at 60.

VARIABLES

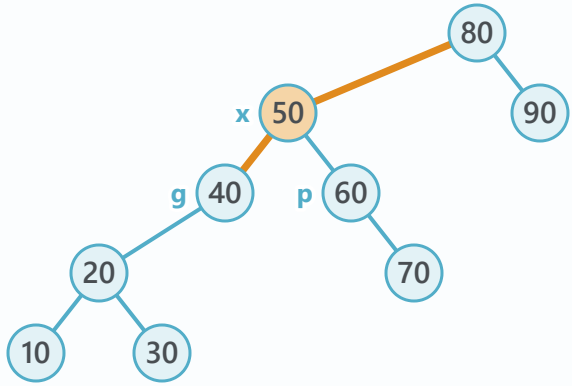
`x = node(50)` `p = node(60)` `g = node(40)`

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
2      while (x != root) {
3          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                          // zig-zag
10             rotate(p, x);   // parent first
▶              rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(50)` `p = node(60)` `g = node(40)`



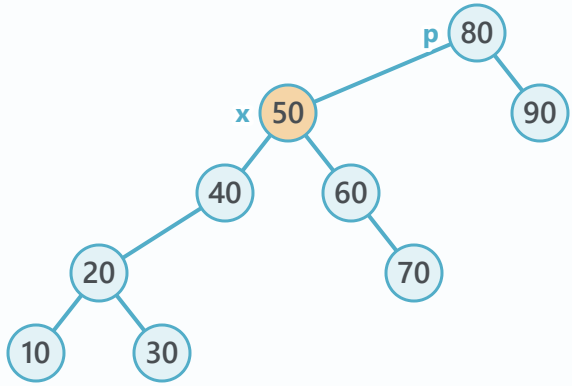
Then the grandparent: rotate left at 40 → x becomes the parent of both p and g.

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
▶      while (x != root) {
▶          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                          // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(50)` `p = node(80)` `g = null`



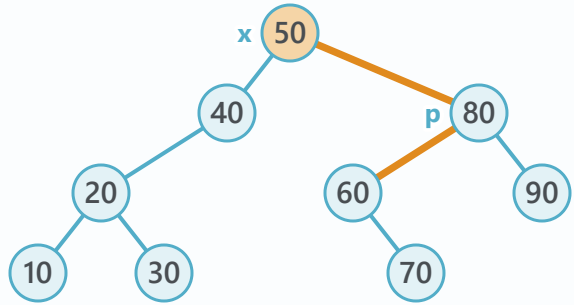
x is not the root. p = 80, g = null (p is the root).

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
2      while (x != root) {
3          Node p = x.parent, g = p.parent;
▶          if (g == null)                  // zig
▶              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                        // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(50)` `p = node(80)` `g = null`



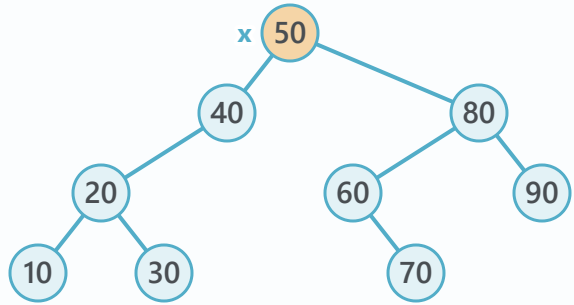
g == null → **zig**: a right rotation at the root, 80; x becomes the root.

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
▶      while (x != root) {
3          Node p = x.parent, g = p.parent;
4          if (g == null)                  // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                        // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(50)`



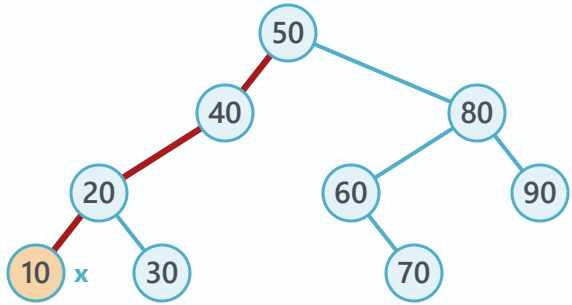
x == root → the loop ends. 50 is now the root; frequently accessed keys stay near the root.

Code Trace: `splay(50)`, then `splay(10)`

```
void splay(Node x) {
    while (x != root) {
        Node p = x.parent, g = p.parent;
        if (g == null)                  // zig
            rotate(p, x);
        else if (sameSide(x, p, g)) {   // zig-zig
            rotate(g, p);   // grandparent first
            rotate(p, x);
        } else {                        // zig-zag
            rotate(p, x);   // parent first
            rotate(g, x);
        }
    }
}
```

VARIABLES

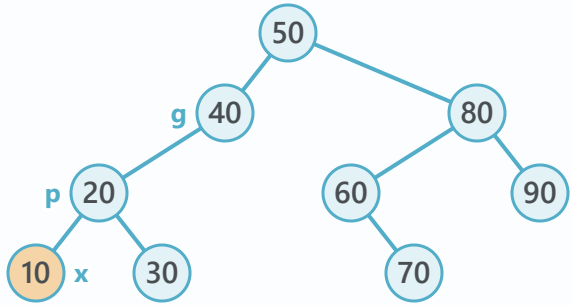
`x = node(10)`



`splay(10)`: first an ordinary BST search, 50 → 40 → 20 → 10; the node found becomes x.

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
▶      while (x != root) {
▶          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                          // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```



x is not the root. p = 20, g = 40.

VARIABLES

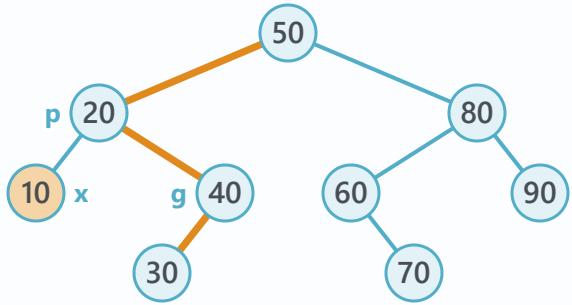
`x = node(10)` `p = node(20)` `g = node(40)`

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
2      while (x != root) {
3          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
▶          else if (sameSide(x, p, g)) {  // zig-zig
▶              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                          // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(10)` `p = node(20)` `g = node(40)`



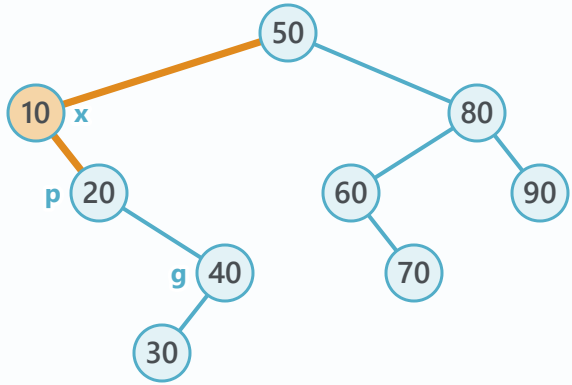
x and p are on the same side → **zig-zig**. The **grandparent** first: rotate right at 40 (p moves up).

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
2      while (x != root) {
3          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
▶              rotate(p, x);
9          } else {                          // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(10)` `p = node(20)` `g = node(40)`



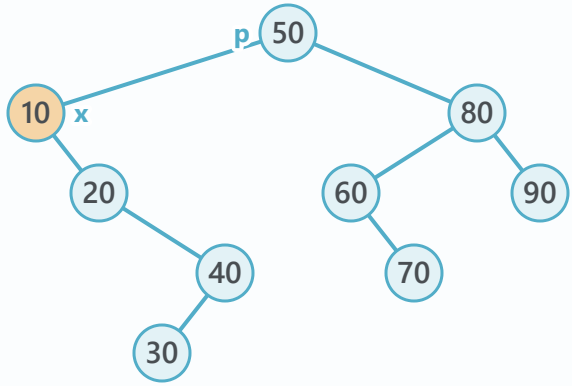
Then the parent: rotate right at 20 → x has risen two levels.

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
▶      while (x != root) {
▶          Node p = x.parent, g = p.parent;
4          if (g == null)                    // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                          // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(10)` `p = node(50)` `g = null`



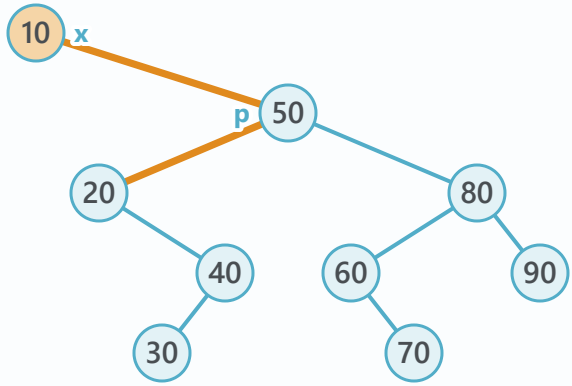
x is not the root. p = 50, g = null (p is the root).

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
2      while (x != root) {
3          Node p = x.parent, g = p.parent;
▶          if (g == null)                  // zig
▶              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                        // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

`x = node(10)` `p = node(50)` `g = null`



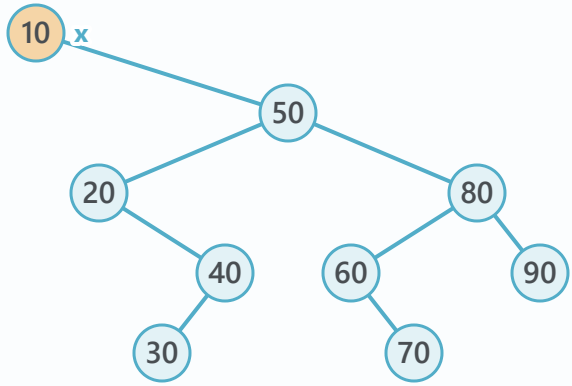
g == null → **zig**: a right rotation at the root, 50; x becomes the root.

Code Trace: `splay(50)`, then `splay(10)`

```
1  void splay(Node x) {
▶      while (x != root) {
3          Node p = x.parent, g = p.parent;
4          if (g == null)                  // zig
5              rotate(p, x);
6          else if (sameSide(x, p, g)) {  // zig-zig
7              rotate(g, p);   // grandparent first
8              rotate(p, x);
9          } else {                        // zig-zag
10             rotate(p, x);   // parent first
11             rotate(g, x);
12         }
13     }
14 }
```

VARIABLES

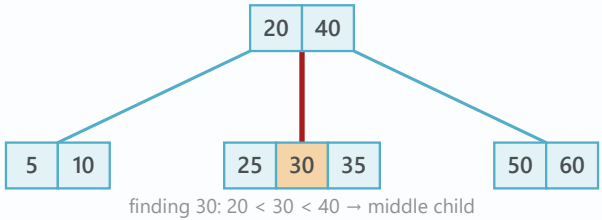
`x = node(10)`



x == root → the loop ends. 10 is the root; 50, accessed just before, is still near it (depth 1). One operation may take O(n); amortized, O(log n).

B-Trees

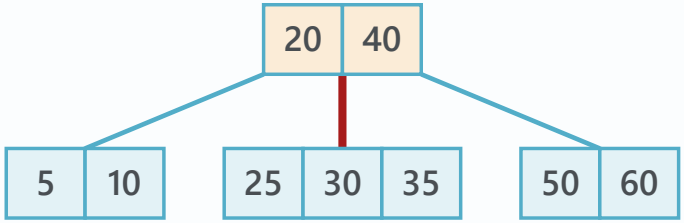
- A balanced multiway search tree whose nodes hold **several keys** and many children.
- In a B-tree of order m, a node has at most m children and m − 1 keys; each non-root **internal node** has at least ⌈m/2⌉ children.
- Keys within a node are sorted, and **all leaves are at the same depth**.
- The height stays O(log n); insertions and deletions split or merge nodes.



Why? One node fills one disk block. Wide and shallow = few disk reads. The basis of database and file-system indexes.

A B-tree of order 4 (a 2-3-4 tree) is equivalent to a red-black tree: merge each black node with its red children and you get a 2-3-4 tree. Insertion and splitting step by step: B-tree simulator.

Search and Insertion in a B-Tree



root: 20 < 30 < 40 → go to the middle child

1 · Search within the node. To find 30, compare it with the root's keys in order; 20 < 30 < 40 → descend to the **middle child**, which covers that gap.

Step 1 / 4

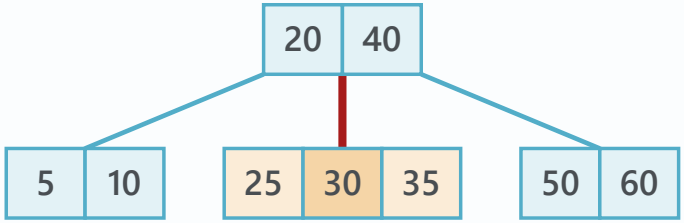
When the root splits

A new root appears and the height grows by just one. All leaves remain at the same depth.

Deletion

An underfull node borrows a key from a sibling or merges with it; balance and fill limits are preserved.

Search and Insertion in a B-Tree



leaf: 25, 30 → found (2 nodes read)

30 is found in the leaf. Each level reads just **one node (one disk block)** → the cost is proportional to the height: $O(\log_m n)$.

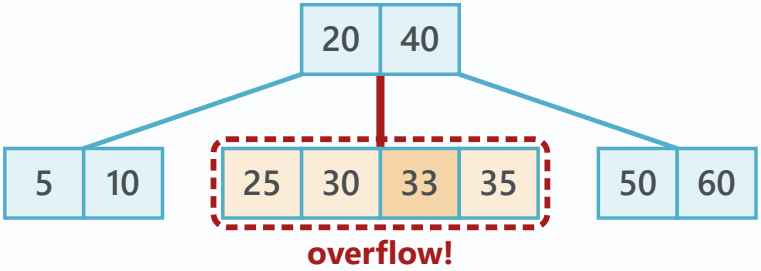
When the root splits

A new root appears and the height grows by just one. All leaves remain at the same depth.

Deletion

An underfull node borrows a key from a sibling or merges with it; balance and fill limits are preserved.

Search and Insertion in a B-Tree



m = 4: at most 3 keys per node

2 · Insert into the leaf. 33 follows the same path and goes into the leaf **in sorted position**. The leaf now holds 4 keys, over its capacity (m − 1 = 3).

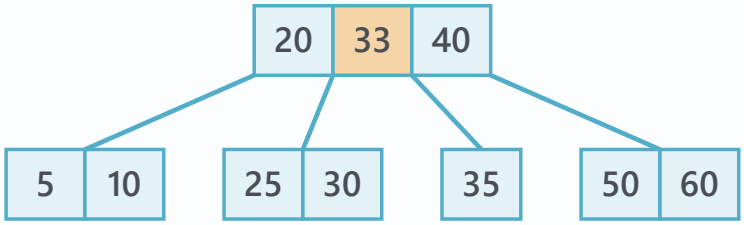
When the root splits

A new root appears and the height grows by just one. All leaves remain at the same depth.

Deletion

An underfull node borrows a key from a sibling or merges with it; balance and fill limits are preserved.

Search and Insertion in a B-Tree



25 30 | 33 ↑ | 35

3 · Split the overflow. The middle key (33) **moves up to the parent**; the rest split into two nodes (25 30 and 35). If the parent overflows too, splitting continues upward; if the root splits, a new root appears and the height grows by one.

When the root splits

A new root appears and the height grows by just one. All leaves remain at the same depth.

Deletion

An underfull node borrows a key from a sibling or merges with it; balance and fill limits are preserved.

B-Tree Insertion and Splitting (m = 3)

Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Insert 10. It goes into the root leaf in sorted order; 1 key (≤ 2), no split.

B-Tree Insertion and Splitting (m = 3)

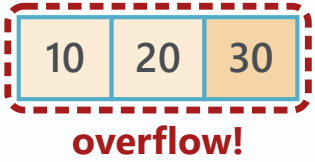
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Insert 20. It goes into the root leaf in sorted order; 2 keys (≤ 2), no split.

B-Tree Insertion and Splitting (m = 3)

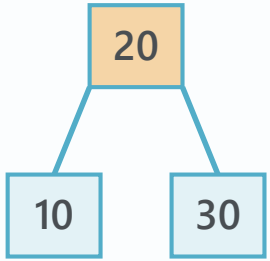
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Insert 30 → node [10, 20, 30]: 3 keys, over capacity (m − 1 = 2) → **split**.

B-Tree Insertion and Splitting (m = 3)

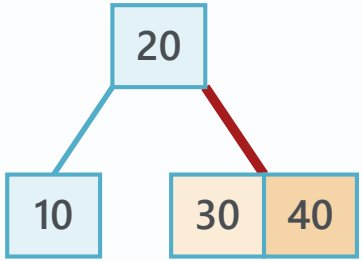
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



The root splits: the median **20** moves up as a new root; 10 and 30 become separate nodes. **The height grows by one**; all leaves stay at the same depth.

B-Tree Insertion and Splitting (m = 3)

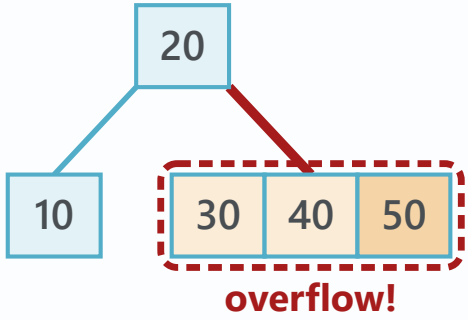
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Insert 40. Go right from the root and add it to the leaf in sorted order; 2 keys (≤ 2), no split.

B-Tree Insertion and Splitting (m = 3)

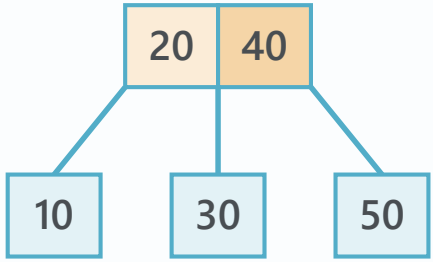
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Insert 50 → node [30, 40, 50]: 3 keys, over capacity (m − 1 = 2) → **split**.

B-Tree Insertion and Splitting (m = 3)

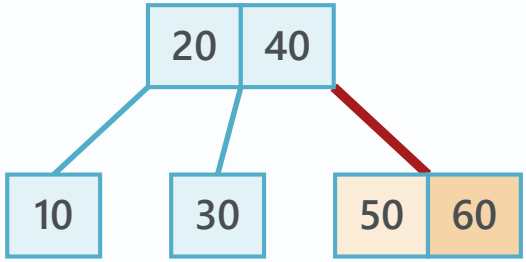
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



The median, **40**, moves up to the parent; the rest split into two nodes, 30 and 50. The parent has room → the split stops here.

B-Tree Insertion and Splitting (m = 3)

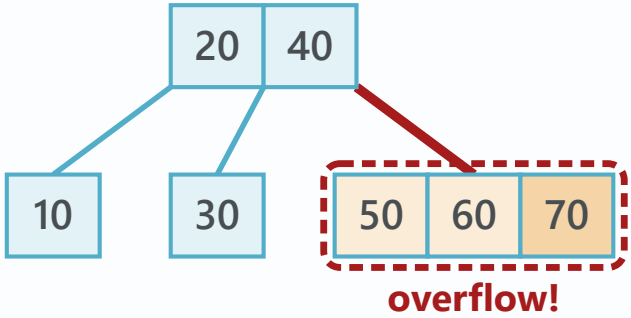
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Insert 60. Go right from the root and add it to the leaf in sorted order; 2 keys (≤ 2), no split.

B-Tree Insertion and Splitting (m = 3)

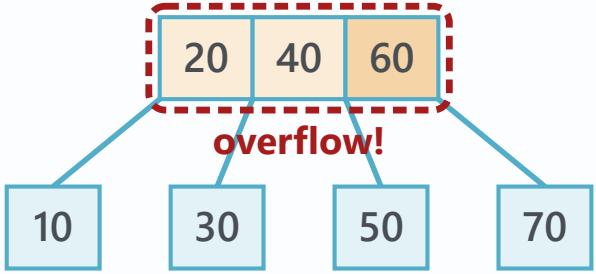
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Insert 70 → node [50, 60, 70]: 3 keys, over capacity (m − 1 = 2) → **split**.

B-Tree Insertion and Splitting (m = 3)

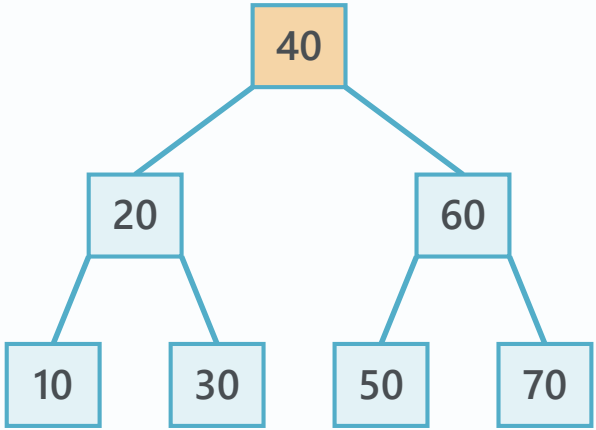
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



The median, **60**, moves up to the parent and the node splits into 50 | 70. But the parent [20, 40, 60] now has 3 keys → **the split propagates upward**.

B-Tree Insertion and Splitting (m = 3)

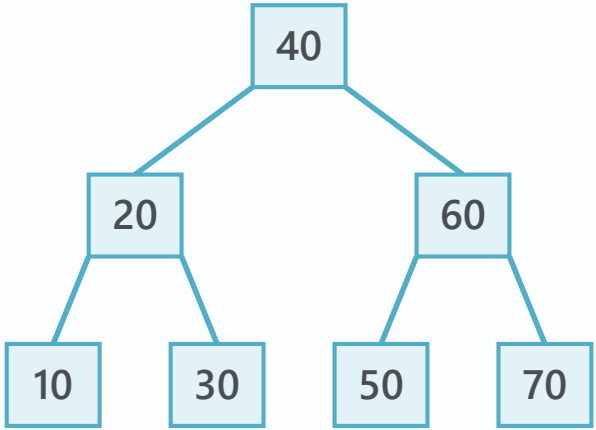
Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



The root splits: the median **40** moves up as a new root; 20 and 60 become separate nodes. **The height grows by one**; all leaves stay at the same depth.

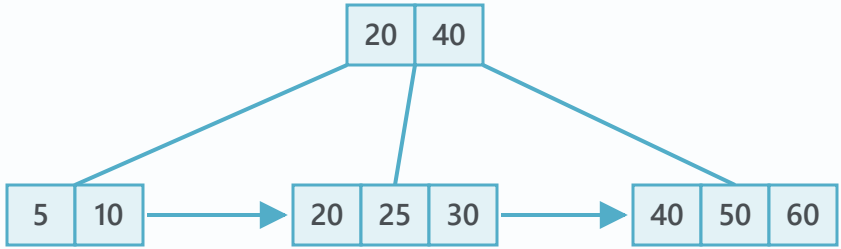
B-Tree Insertion and Splitting (m = 3)

Keys inserted in order: **10, 20, 30, 40, 50, 60, 70** · order m = 3 → at most 2 keys and 3 children per node.



Result: 7 keys on 3 levels; every internal node has 2 children; **all leaves at the same depth**. A B-tree grows upward: its height increases only when the root splits.

B+ Trees



leaves form a linked list: a range query reads left to right

Data in the leaves

All keys (and records) live in the leaves; internal nodes hold only routing keys.

Linked leaves

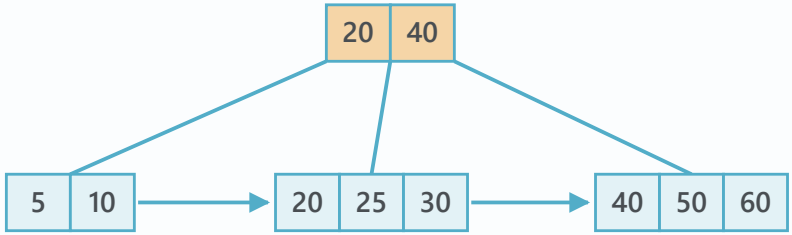
The leaves are linked together, so range queries such as "between 20 and 50" are fast.

Wider internal nodes

Internal nodes carry no records, so more keys fit in a block → a shallower tree.

Most relational database indexes are B+ trees.

Range Query in a B+ Tree: 10 to 45

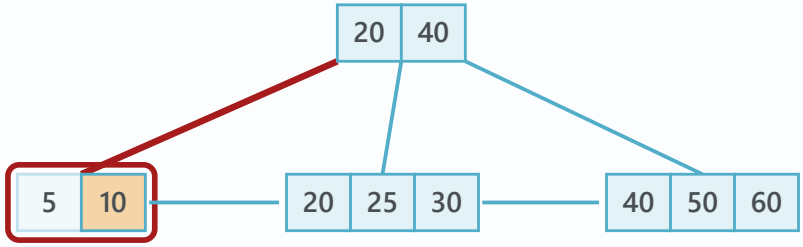


query: 10 ≤ key ≤ 45

result: —

Start at the root: 10 < 20 → the leaf holding the lower bound 10 is under the **leftmost** child. (Just like a search: O(log n) levels.)

Range Query in a B+ Tree: 10 to 45

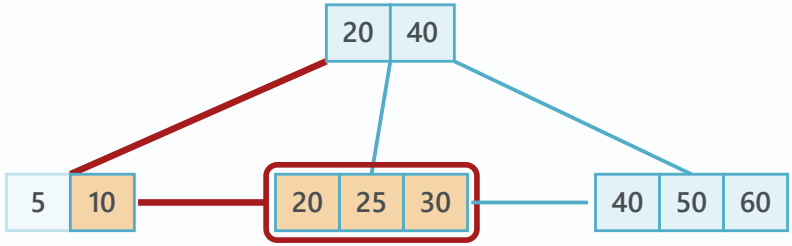


query: 10 ≤ key ≤ 45

result: 10

First leaf, [5 10]: skip 5 < 10 and take **10**. At the end of the leaf → follow the **next-leaf** link (without climbing back up).

Range Query in a B+ Tree: 10 to 45

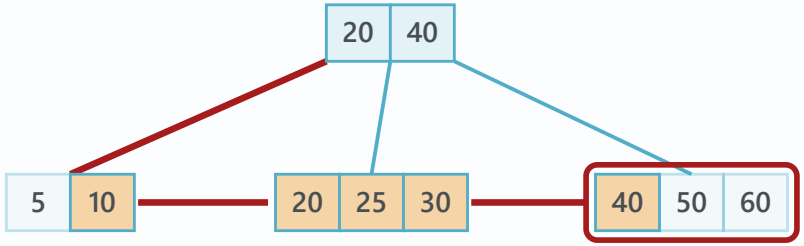


query: 10 ≤ key ≤ 45

result: 10 20 25 30

Second leaf: 20, 25, and 30 are all in range → take them. Follow the link again.

Range Query in a B+ Tree: 10 to 45

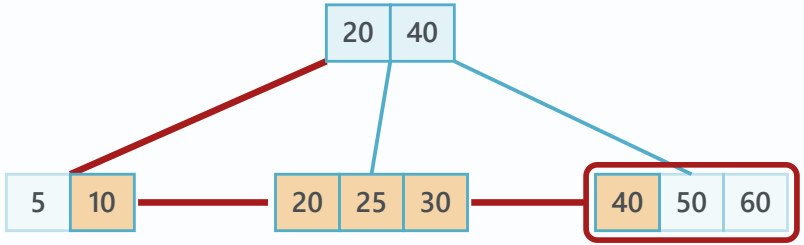


query: 10 ≤ key ≤ 45

result: 10 20 25 30 40

Third leaf: take **40**; 50 > 45 → **stop**. Result: 10 20 25 30 40.

Range Query in a B+ Tree: 10 to 45



query: 10 ≤ key ≤ 45

result: 10 20 25 30 40

Cost: one descent, O(log n), plus the leaves read. A B-tree would have to climb up and down through internal nodes again and again; the B+ tree's leaf chain gives **O(log n + k)** (k: number of results).

SUMMARY

Summary: Operation Costs in Search Trees

Structure	Height	Search	Insert	Delete	How it stays balanced
BST	log n … n	O(h)	O(h)	O(h)	it doesn't; worst case O(n)
AVL	≤ 1.44 log n	O(log n)	O(log n)	O(log n)	bf ∈ {−1, 0, 1}; LL, RR, LR, RL rotations
Red-black	≤ 2 log(n+1)	O(log n)	O(log n)	O(log n)	recoloring + at most 2–3 rotations
Splay	unbounded	O(log n)*	O(log n)*	O(log n)*	accessed node moves to the root by zig / zig-zig / zig-zag
B-tree / B+	≈ $\log_m n$	O(log n)	O(log n)	O(log n)	overflowing nodes split; all leaves at the same depth

* Amortized; one operation can take O(n). In every structure, inorder traversal is O(n) and yields the keys sorted; min/max are O(h).

BST property
left < root < right, for whole subtrees.

Deletion
leaf → remove; one child → bypass; two children → successor/predecessor.

Rotation
O(1); preserves the inorder sequence and changes heights.

SUMMARY

Interactive Simulators

Binary Search Tree Simulator

Search, insertion, the three deletion cases, and traversals.

BST Traversal Simulator

Insertion, search, deletion; preorder, inorder, postorder, and level-order traversals.

AVL Tree Simulator

Balance factors and the LL / RR / LR / RL rotations.

Red-Black Tree Simulator

Insertion with recoloring and rotations.

Splay Tree Simulator

Bringing the accessed node to the root with zig, zig-zig, and zig-zag.

B-Tree Simulator

How nodes split as keys are added and the tree grows in balance.

B+ Tree Simulator

A multiway tree that keeps its data in linked leaves.

SUMMARY

Common Mistakes

Dropping the return value

`delete(root, 40);` breaks the tree when the root is deleted. Write `root = delete(root, 40);` (likewise for insert).

Checking only the children

The BST property covers whole subtrees, so a check must pass the allowed **(min, max)** range down (`isBST`).

Comparing objects with `==`

That compares references. Order keys with `compareTo` and test equality with `equals`.

Wrong height-update order

After an AVL rotation, update the node that **moved down** first, then the new root; the reverse gives wrong bf values.

"A BST is always O(log n)"

Wrong: sorted input gives h = n − 1. O(log n) is guaranteed only for balanced trees (AVL, red-black).

Forgetting null children

An empty subtree has height −1, and in a red-black tree NIL counts as black; hence helpers like `h(null)` and `isRed(null)`.

SUMMARY

Sample Problems

1. Insert 50, 30, 70, 20, 40, 60, 80 in order into an empty BST. What is the height? What does an inorder traversal print?
2. If the same keys arrive in the order 20, 30, 40, 50, 60, 70, 80, what are the height and the search cost?
3. Delete the root 50 from the tree in problem 1 using the inorder successor. What does the tree look like?
4. Insert 3, 2, 1 into an empty AVL tree. Which case arises, and which rotation fixes it?
5. Insert 10, 30, 20 into an empty AVL tree. Which case arises, and what is the resulting tree?
6. In a red-black tree, the longest root-to-NIL path is at most how many times the shortest? Why?
7. In what order are the rotations done in a splay tree's zig-zig step?

▶ **Show answers**

SUMMARY

Exercises

1. Write `floor(root, x)`, which returns the largest key ≤ x (or null if there is none).
2. Write a method that finds the **k-th smallest** key in a BST. What is its time complexity?
3. In what order should 1, 2, …, 7 be inserted into an empty BST to get a perfect tree of height 2?
4. Insert 1, 2, …, 7 in order into an empty AVL tree. What is the final tree? How many rotations are done?
5. What is the minimum number of nodes in an AVL tree of height 5?
6. Why does `range(root, lo, hi)` take O(h + s) rather than O(n)?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011.

The End

Chapter 10: Search Trees

Next deck: Chapter 11: Priority Queues and Heaps · Study questions · Simulators: Binary Search Tree, BST Traversal, Red-Black Tree, AVL Tree, Splay Tree, B-Tree, B+ Tree · Glossary