Section 9

CSCI E-22

Will Begin Shortly

Binary Search Trees

- Recall that a *binary search tree* is a special binary tree in which the keys are kept in order. In particular, we call a tree a binary search tree if it satisfies the *binary search tree property*.

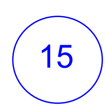
- A tree satisfies the *binary search tree property* if, for a root node with key *k*, all nodes in the left subtree have keys less than *k*, and all nodes in the right subtree have keys greater than or equal to *k*.

Insert the following sequence of keys into an empty binary search tree:

15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24

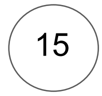
Insert the following sequence of keys into an empty binary search tree:

15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24



Insert the following sequence of keys into an empty binary search tree:

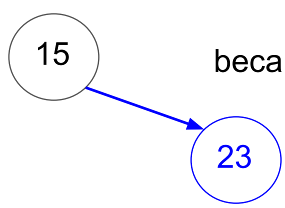
`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



23 < 15?   or   23 > 15?

Insert the following sequence of keys into an empty binary search tree:

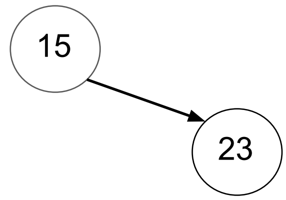
`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



because 23 > 15

Insert the following sequence of keys into an empty binary search tree:

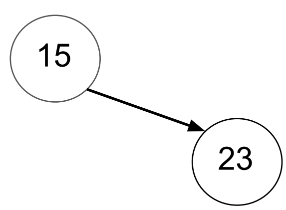
`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



20 < 15? or 20 > 15?

Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



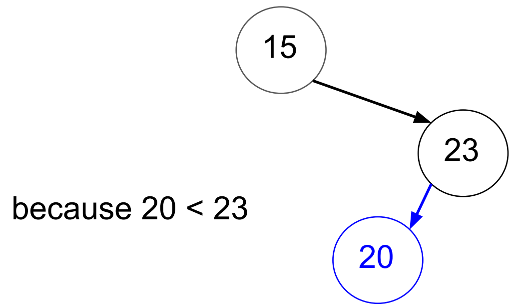
20 < 15? or 20 > 15?

So, we need to look in the right subtree.
Since 23 is already in the right subtree, we need to ask:

20 < 23? or 20 > 23?

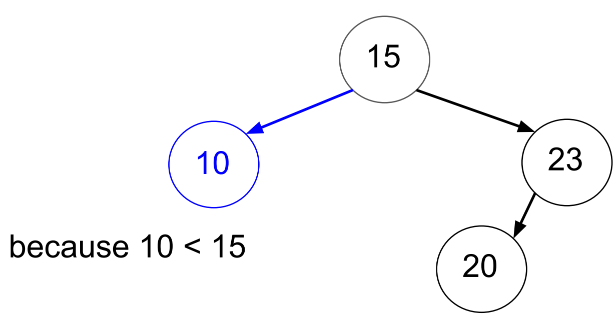
Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



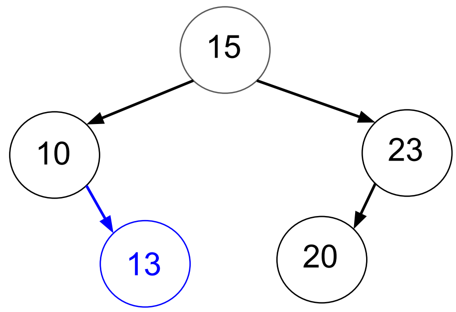
Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



Insert the following sequence of keys into an empty binary search tree:

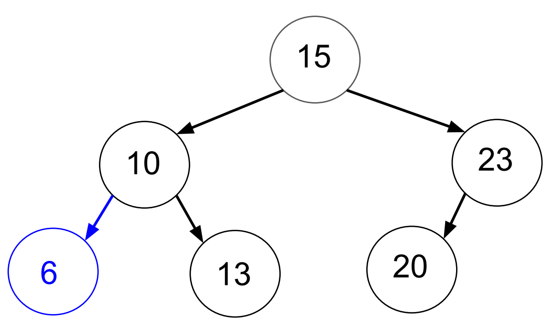
`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



because 13 < 15
and 13 > 10

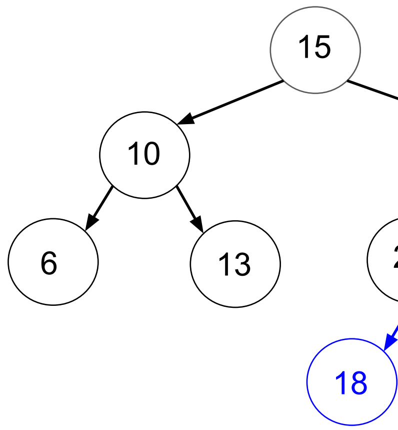
Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



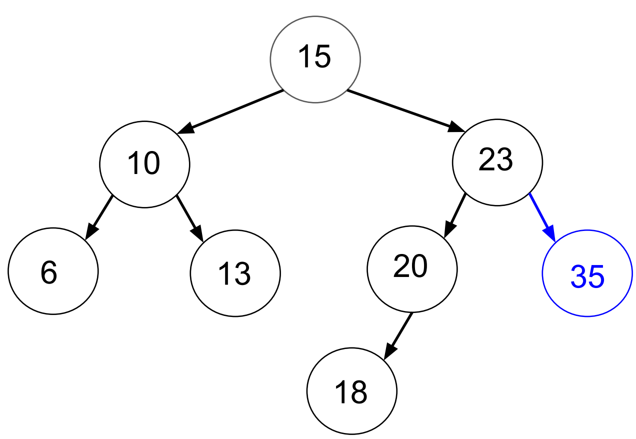
Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



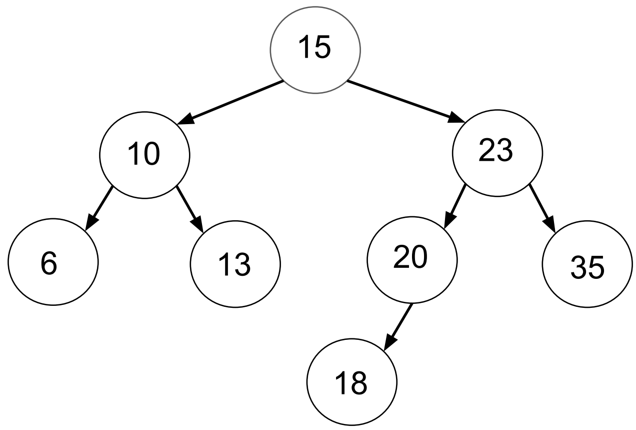
Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



Insert the following sequence of keys into an empty binary search tree:

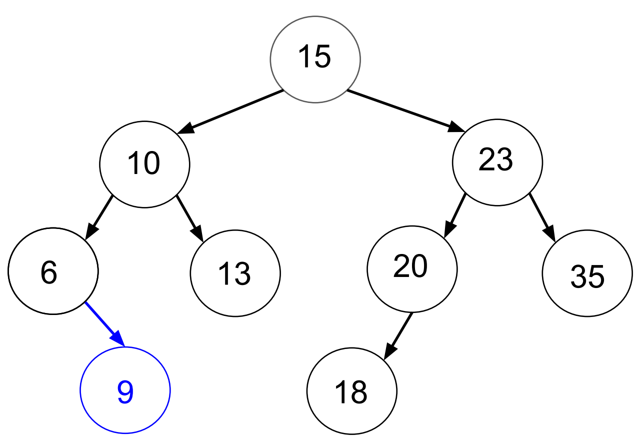
`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



As 23 is a duplicate, it is a unique case. It will still start at the root, and see that 23 > 15, and proceed to check the right subtree. However, it will see the root of the right subtree is 23, a match. It will therefore halt its insert as the number already exists in the tree.

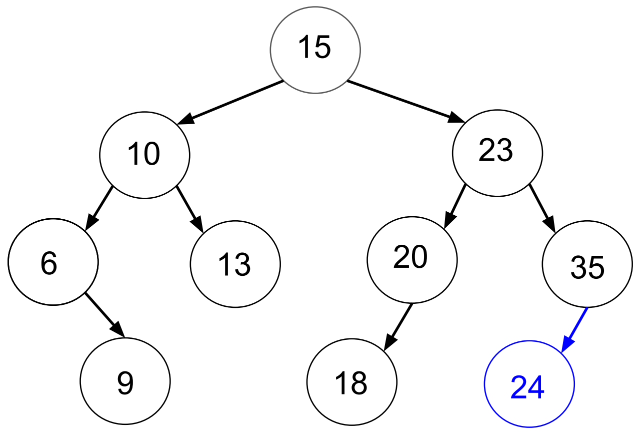
Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



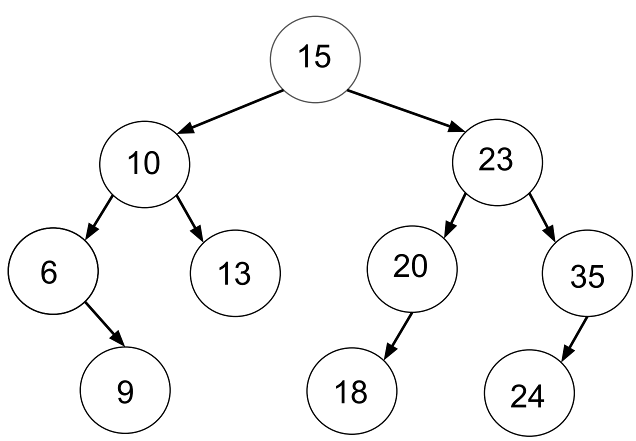
Insert the following sequence of keys into an empty binary search tree:

`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`

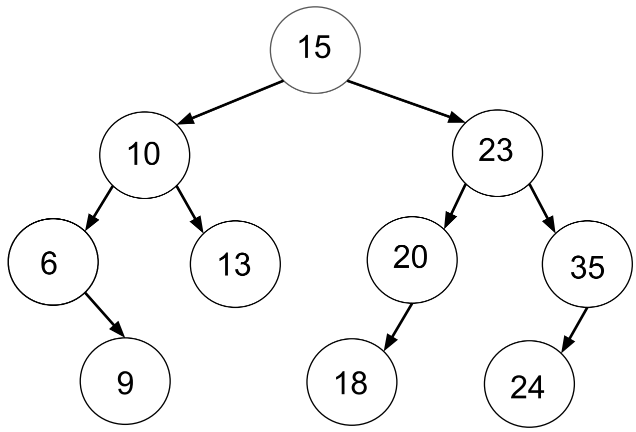


Insert the following sequence of keys into an empty binary search tree:

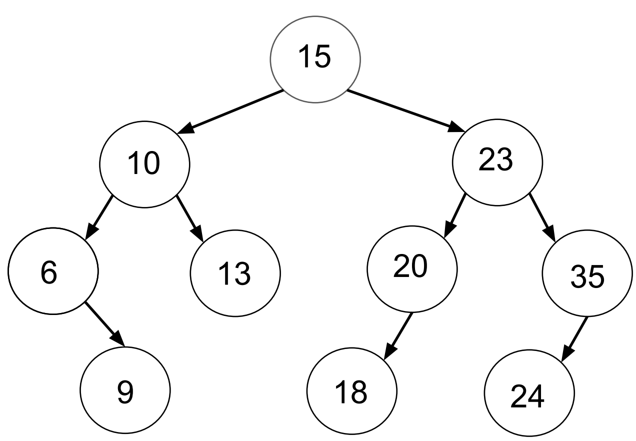
`15, 23, 20, 10, 13, 6, 18, 35, 23 (a duplicate), 9, 24`



Is this a balanced tree?

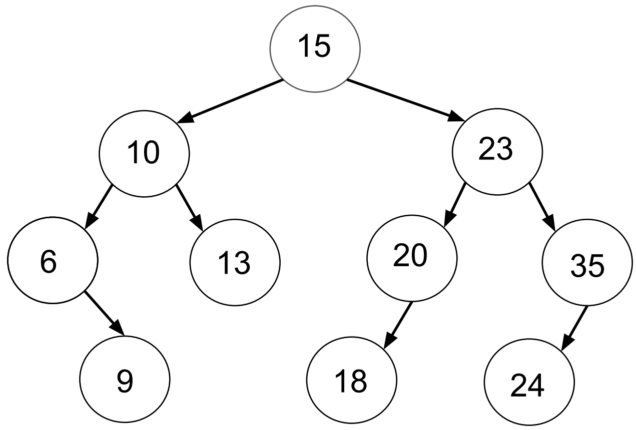


Is this a balanced tree?



*A tree is balanced if, **for each node**, the node's subtrees have the same height, or have heights that differ by 1.*

Is this a balanced tree?

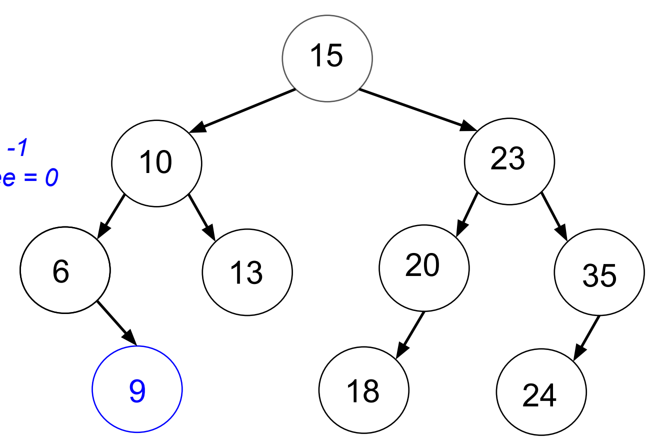


A tree is balanced if, ***for each node****, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

left: -1
right: -1

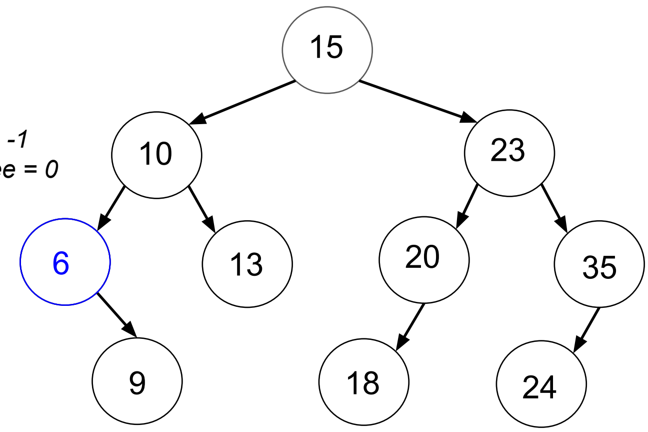
diff: 0

A tree is balanced if, ***for each node****, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

left: -1
right: 0

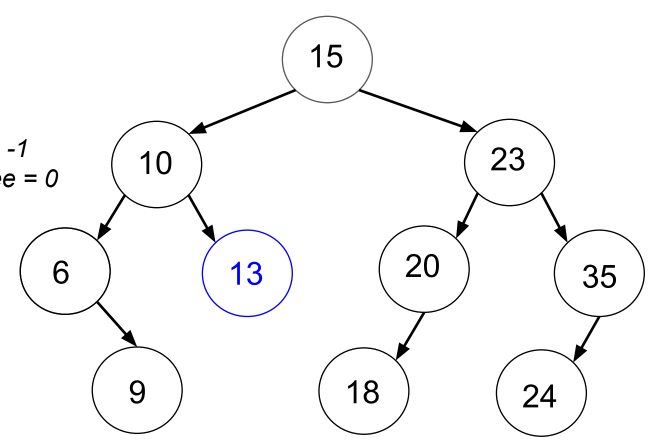
diff: 1

A tree is balanced if, ***for each node****, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

left: -1
right: -1

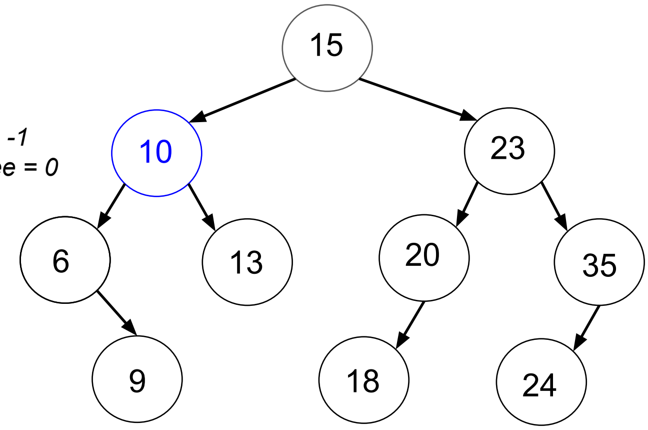
diff: 0

A tree is balanced if, ***for each node****, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

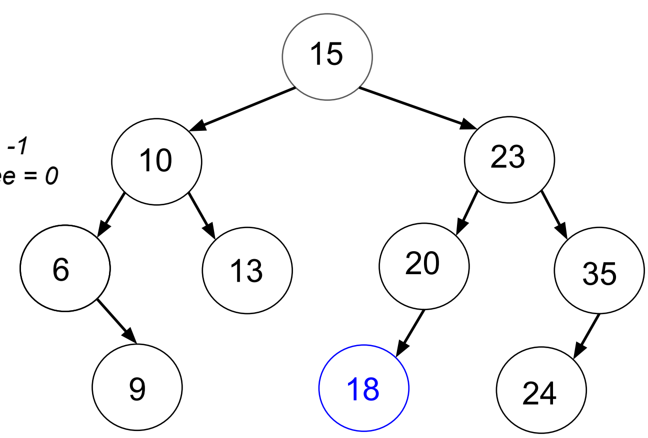
left: 1
right: 0 **diff: 1**

*A tree is balanced if, **for each node**, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

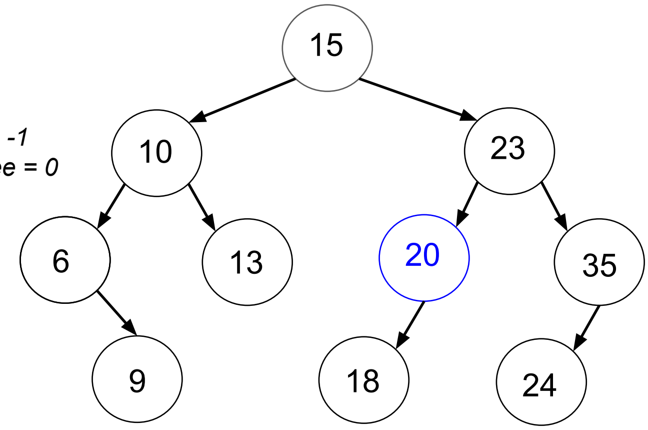
left: -1
right: -1 **diff: 0**

*A tree is balanced if, **for each node**, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

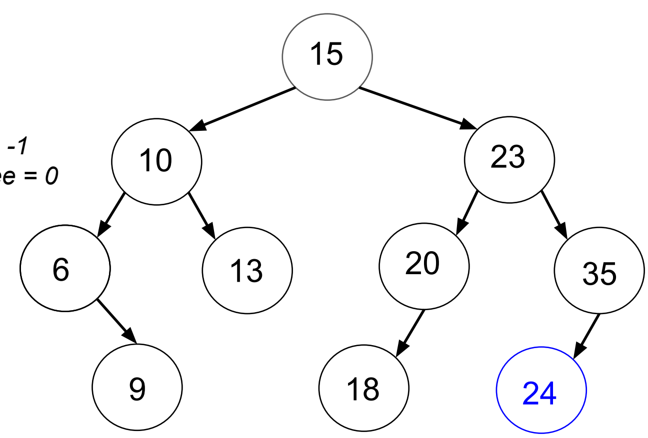
left: 0
right: -1
diff: 1

*A tree is balanced if, **for each node**, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

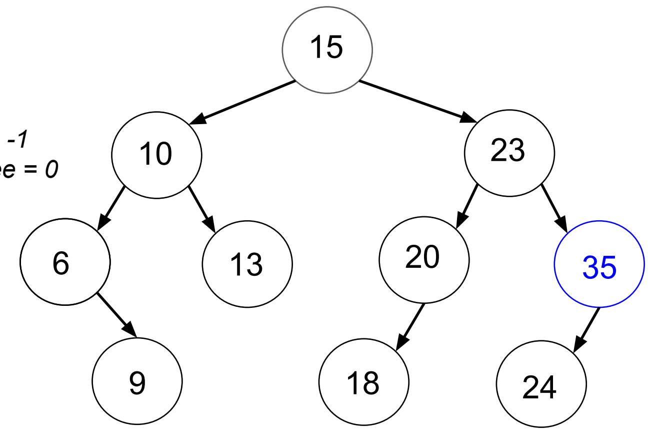
left: -1
right: -1
diff: 0

*A tree is balanced if, **for each node**, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree?

height of empty tree = -1
height of one-node tree = 0



subtree heights

left: 0
right: -1

diff: 1

A tree is balanced if, ***for each node****, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

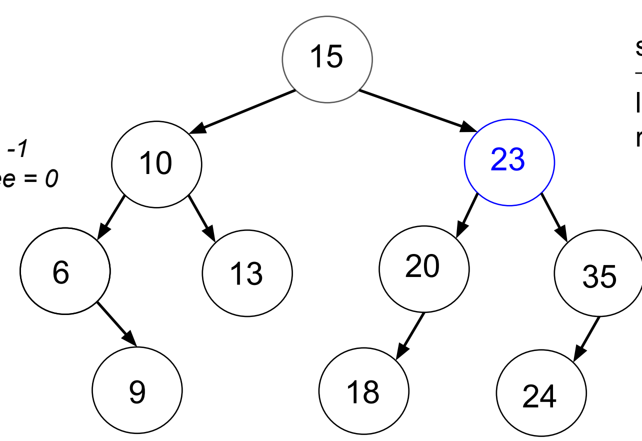
Is this a balanced tree?

subtree heights

left: 1
right: 1

diff: 0

height of empty tree = -1
height of one-node tree = 0



A tree is balanced if, ***for each node****, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

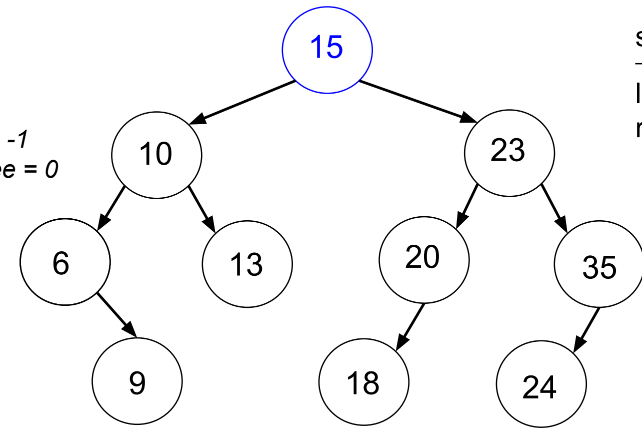
Is this a balanced tree?

subtree heights

left: 2
right: 2

diff: 0

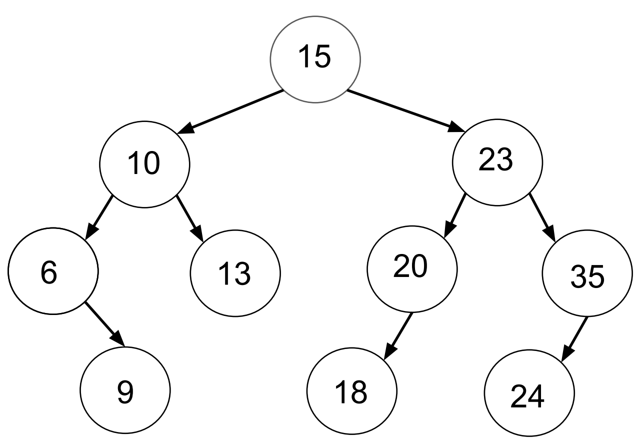
height of empty tree = -1
height of one-node tree = 0



*A tree is balanced if, **for each node**, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

Is this a balanced tree? Yes. For each node, the heights of the subtrees differ by at most one.

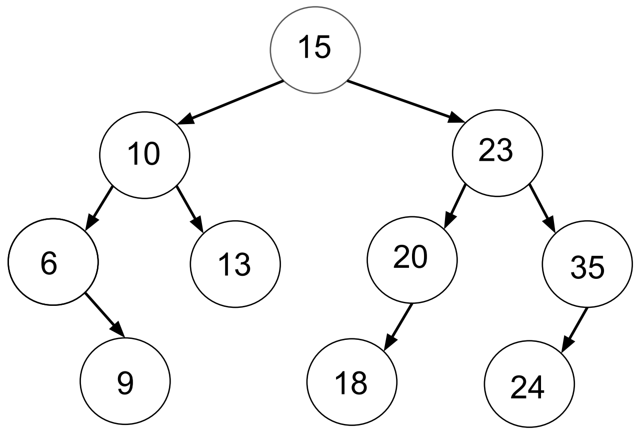


*A tree is balanced if, **for each node**, the node's subtrees have the same height, or have heights that differ by 1.*

height of a tree = maximum depth of its nodes
depth = # of edges on path from it to root

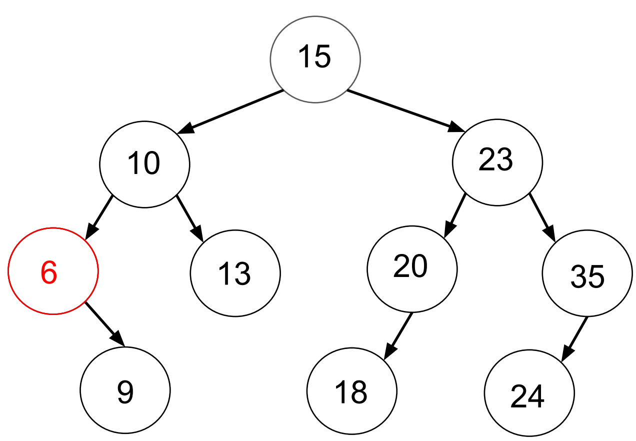
What will the tree look like after deleting the following items?:

`6, 15, 20`



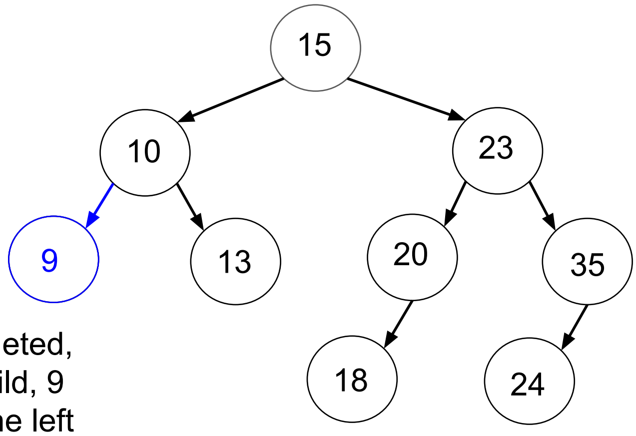
What will the tree look like after deleting the following items?:

`6, 15, 20`



What will the tree look like after deleting the following items?:

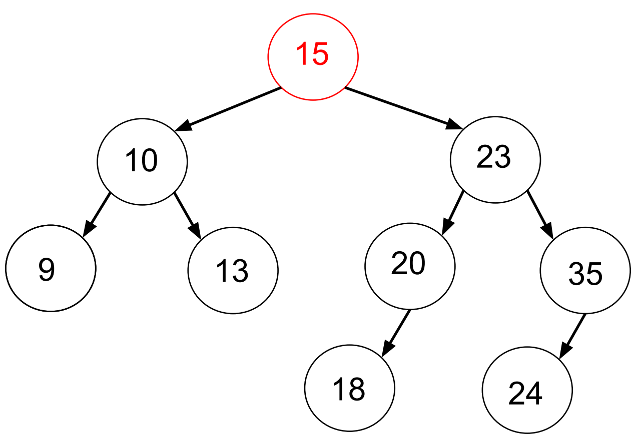
`6, 15, 20`



Since 6 is being deleted, and 9 is its only child, 9 takes its place as the left child of the node containing 10

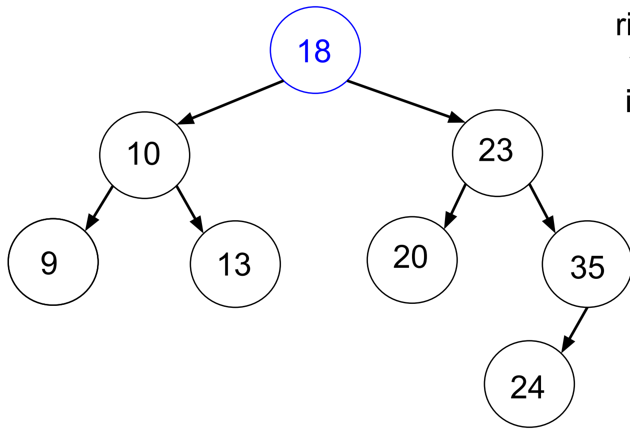
What will the tree look like after deleting the following items?:

`6, 15, 20`



What will the tree look like after deleting the following items?:

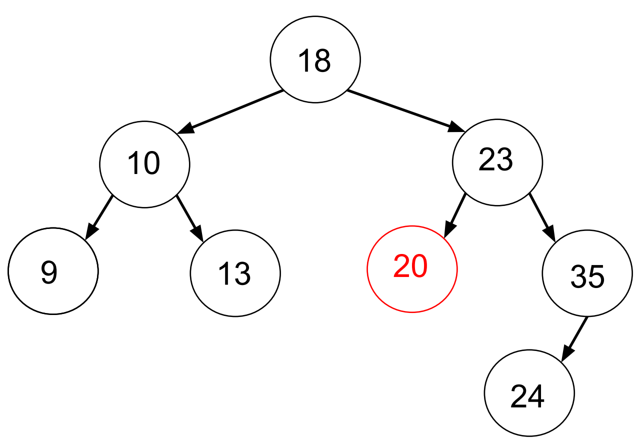
`6, 15, 20`



Since 15 has a left and right child, we replace it with the smallest item in its right subtree, 18, and then delete the node containing 18

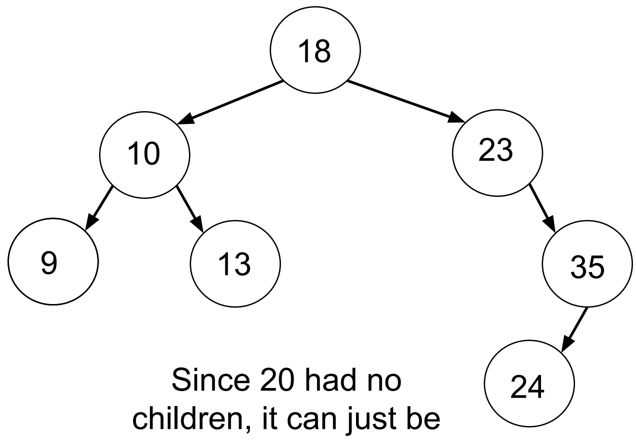
What will the tree look like after deleting the following items?:

`6, 15, 20`



What will the tree look like after deleting the following items?:

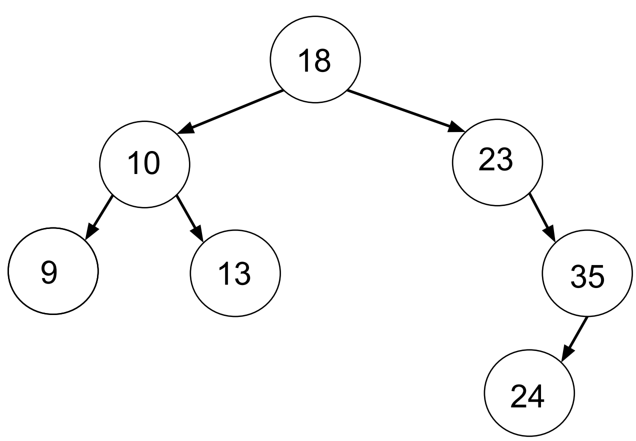
`6, 15, 20`



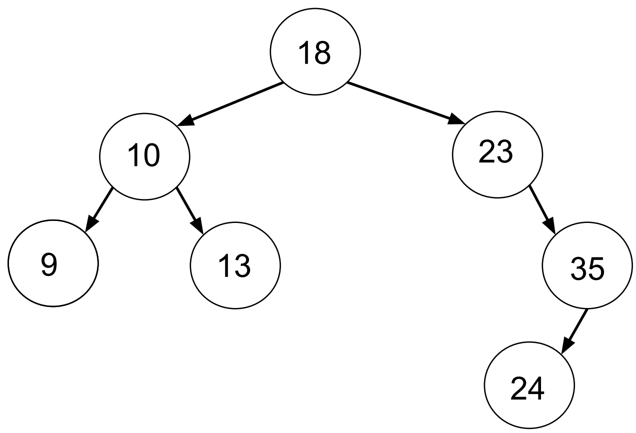
Since 20 had no children, it can just be removed

What will the tree look like after deleting the following items?:

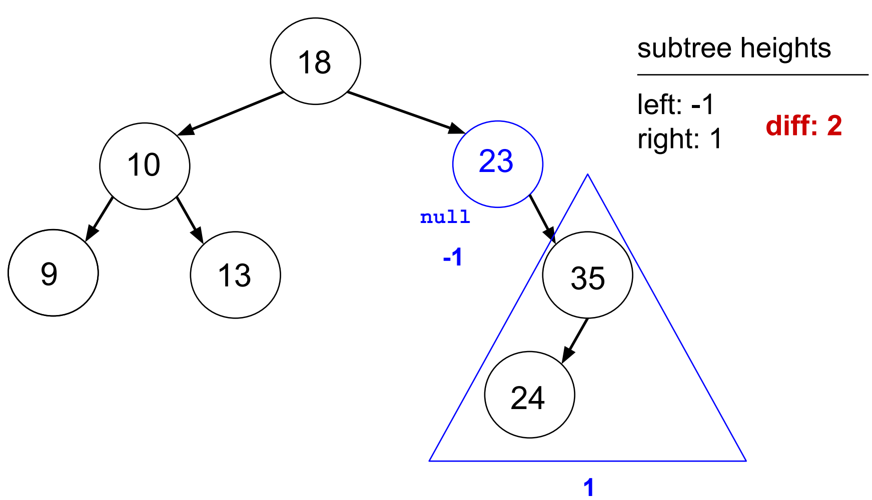
`6, 15, 20`



Is this a balanced tree?



Is this a balanced tree? No. The 23 has an empty left subtree (which can be thought of as having a height of -1) but its right subtree has a height of 1 - so the heights of its subtrees differ by more than 1.



Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

We can approach this problem somewhat like recursion over a linked list. The main difference will be that instead of only recursively processing `node.next`, we'll need to process *both* `node.left` and `node.right`.

Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

Assume we have a node `root` which is the root node of a tree. How can we recursively define the number of leaf nodes in that tree?

Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

Assume we have a node `root` which is the root node of a tree. How can we recursively define the number of leaf nodes in that tree?

The number of leaves in a tree for which `root` is the root node is the sum of the number of leaves in `root`'s left subtree and the number of leaves in `root`'s right subtree. In other words, we can say that

numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

What will be our base case(s)?

Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

What will be our base case(s)?

We need to terminate the recursion if we are either at a leaf or if we are given an empty tree.

Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

What will our function return?

An integer value.

Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

Review:

- Return an integer value
- We need to terminate the recursion if we are either given an empty tree or at a leaf.
- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

- Return an integer value
- We need to terminate the recursion if we are either given an empty tree or at a leaf.
- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
```

- We need to terminate the recursion if we are either given an empty tree or at a leaf.
- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
```

- We need to terminate the recursion if we are either given an empty tree or at a leaf.
- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
    if (root == null) {
            // we're given an empty tree (no nodes)
            return 0;
    }
```

- We need to terminate the recursion if we are either given an empty tree or at a leaf.
- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
    if (root == null) {
            // we're given an empty tree (no nodes)
            return 0;
    }
```

- We need to terminate the recursion if we are either given an empty tree or at a leaf.
- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
    if (root == null) {
            // we're given an empty tree (no nodes)
            return 0;
    }else if (root.left == null && root.right == null) {
            // we're given a leaf node
            return 1;
    }
```

- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
    if (root == null) {
            // we're given an empty tree (no nodes)
            return 0;
    }else if (root.left == null && root.right == null) {
            // we're given a leaf node
            return 1;
    }
```

- numberOfLeaves(root) = numberOfLeaves(root.left) + numberOfLeaves(root.right)

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
    if (root == null) {
            // we're given an empty tree (no nodes)
            return 0;
    }else if (root.left == null && root.right == null) {
            // we're given a leaf node
            return 1;
    }else {
            return numLeaves(root.left)+ numLeaves(root.right);
    }
```

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

```
public int numLeaves(Node root) {
    if (root == null) {
            // we're given an empty tree (no nodes)
            return 0;
    }else if (root.left == null && root.right == null) {
            // we're given a leaf node
            return 1;
    }else {
            return numLeaves(root.left)+ numLeaves(root.right);
    }
```

```
public int numLeaves(Node root) {
    if (root == null) {
            // we're given an empty tree (no nodes)
            return 0;
    }else if (root.left == null && root.right == null) {
            // we're given a leaf node
            return 1;
    }else {
            return leafCount(root.left)+ leafCount(root.right);
    }
```

What would we need to do if we wanted to write this method iteratively? What sort of data structures would we need?

```java
public int numLeaves(Node root) {
     if (root == null) {
             // we're given an empty tree (no nodes)
             return 0;
     }else if (root.left == null && root.right == null) {
             // we're given a leaf node
             return 1;
     }else {
              return leafCount(root.left)+ leafCount(root.right);
     }
```

What would we need to do if we wanted to write this method iteratively? What sort of data structures would we need? We would need to maintain a stack onto which we could push the left and right subtree's root nodes as we iterated.

2-3 Trees

Insert the following sequence of keys into an empty 2-3 tree:

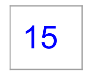
`15, 23, 20, 10, 13, 6, 18, 35, 27, 9`

Remember:

- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and 0 or 2 children
    - a **3-node**, which contains two data items and 0 or 3 children
  - the keys form a search tree

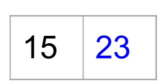
Insert the following sequence of keys into an empty 2-3 tree:

15, 23, 20, 10, 13, 6, 18, 35, 27, 9



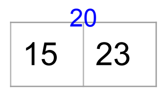
Insert the following sequence of keys into an empty 2-3 tree:

15, 23, 20, 10, 13, 6, 18, 35, 27, 9



Insert the following sequence of keys into an empty 2-3 tree:

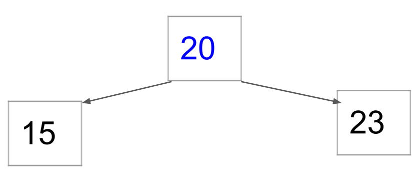
`15, 23, 20, 10, 13, 6, 18, 35, 27, 9`



Inserting 20 requires a split, which creates a new root containing the middle item

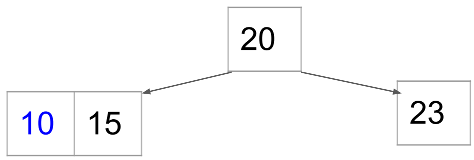
Insert the following sequence of keys into an empty 2-3 tree:

`15, 23, 20, 10, 13, 6, 18, 35, 27, 9`



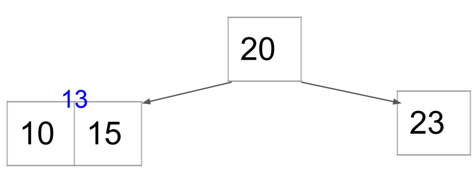
Insert the following sequence of keys into an empty 2-3 tree:

15, 23, 20, 10, 13, 6, 18, 35, 27, 9



Insert the following sequence of keys into an empty 2-3 tree:

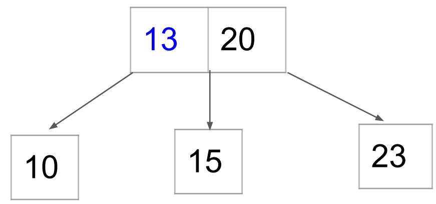
15, 23, 20, 10, 13, 6, 18, 35, 27, 9



Inserting 13 requires a split, sending the middle item up a level to join the root

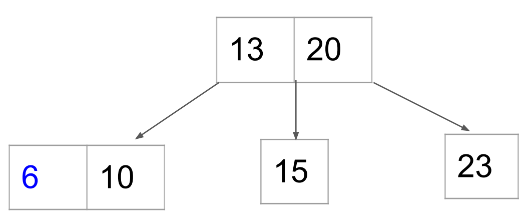
Insert the following sequence of keys into an empty 2-3 tree:

15, 23, 20, 10, 13, 6, 18, 35, 27, 9



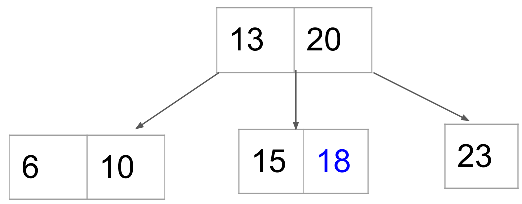
Insert the following sequence of keys into an empty 2-3 tree:

15, 23, 20, 10, 13, 6, 18, 35, 27, 9



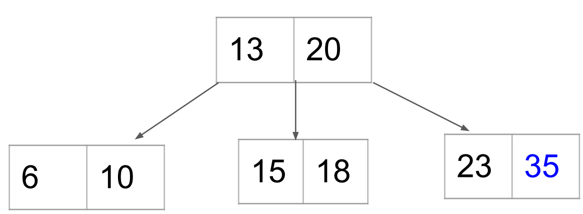
Insert the following sequence of keys into an empty 2-3 tree:

15, 23, 20, 10, 13, 6, 18, 35, 27, 9

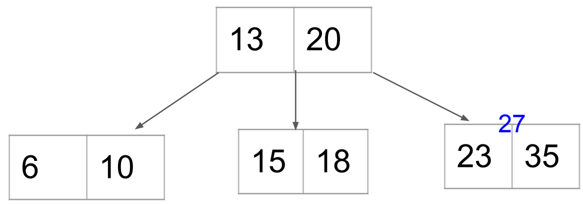


Insert the following sequence of keys into an empty 2-3 tree:

15, 23, 20, 10, 13, 6, 18, 35, 27, 9

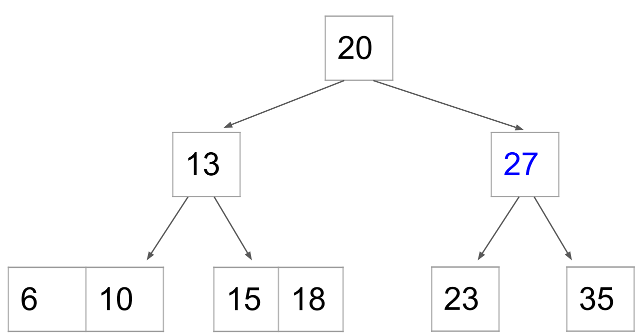


Insert the following sequence of keys into an empty 2-3 tree:
15, 23, 20, 10, 13, 6, 18, 35, 27, 9



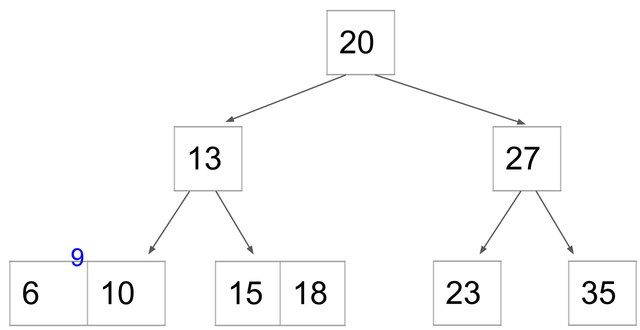
Inserting 27 requires two splits. First, it splits the node containing 23 and 35, sending the middle item, 27, up a level. However, once up a level, it once again splits a node (this time, that containing 13 and 20), and sends the new middle item, 20, up another level

Insert the following sequence of keys into an empty 2-3 tree:
15, 23, 20, 10, 13, 6, 18, 35, 27, 9



Insert the following sequence of keys into an empty 2-3 tree:

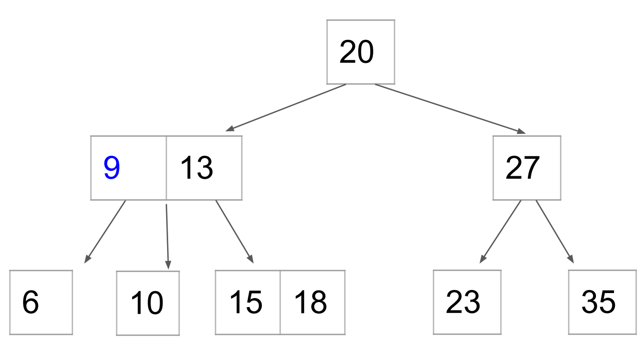
`15, 23, 20, 10, 13, 6, 18, 35, 27, 9`



Inserting 9 requires one split, sending the middle element, 9, up a level where it joins 13 as the root of the left subtree

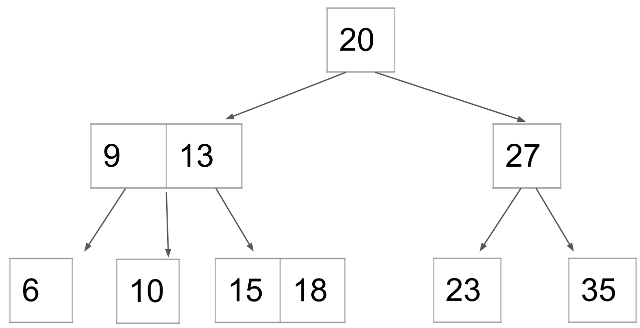
Insert the following sequence of keys into an empty 2-3 tree:

`15, 23, 20, 10, 13, 6, 18, 35, 27, 9`



Insert the following sequence of keys into an empty 2-3 tree:

`15, 23, 20, 10, 13, 6, 18, 35, 27, 9`



B-Trees

We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`

*Remember, with B-trees:

- A B-tree of order *m* is a tree in which each node has:
  - At most 2*m* entries (and, for internal nodes, 2*m*+1 children)
  - At least *m* entries (and, for internal nodes, *m*+1 children)
  - Exception: the root node may have as few as 1 entry
  - A 2-3 tree is essentially a B-tree of order 1

B-Trees

We want to construct a B-tree where *m = 2* for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`

Remember:

- A B-tree of order *m* is a tree in which each node has:
  - At most 4 entries (and, for internal nodes, 5 children)
  - At least 2 entries (and, for internal nodes, 3 children)
  - Exception: the root node may have as few as 1 entry
  - A 2-3 tree is essentially a B-tree of order 1

We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`

We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14

3	51

We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14

3	10	51

We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14

3	10	51	77

We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14

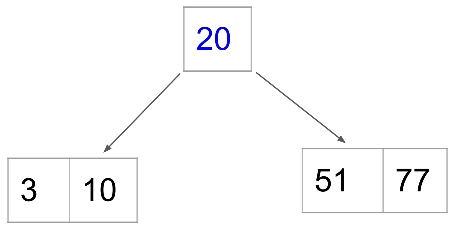
20

3	10	51	77

Inserting 20 requires a split

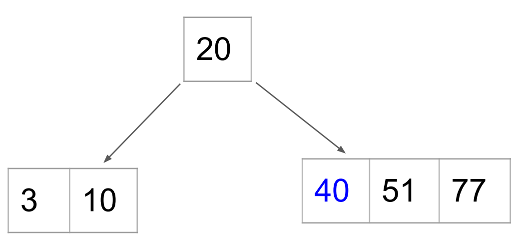
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14



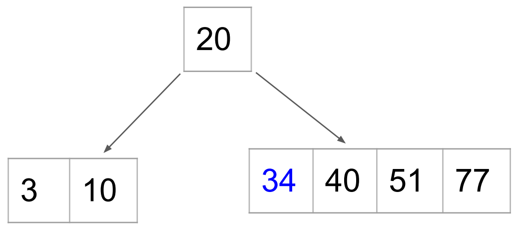
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14



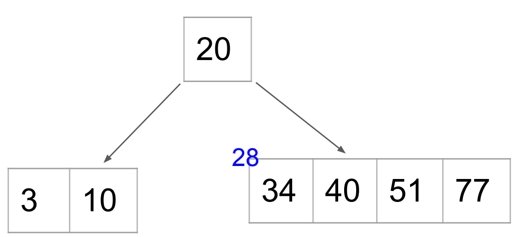
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

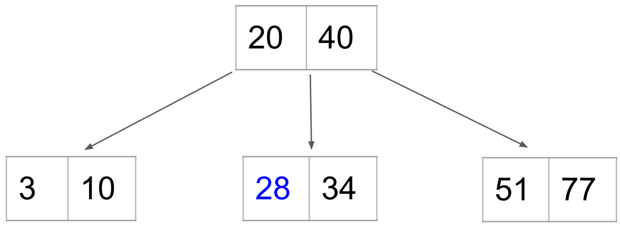
`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



Inserting 28 requires a split, sending the middle item up a level

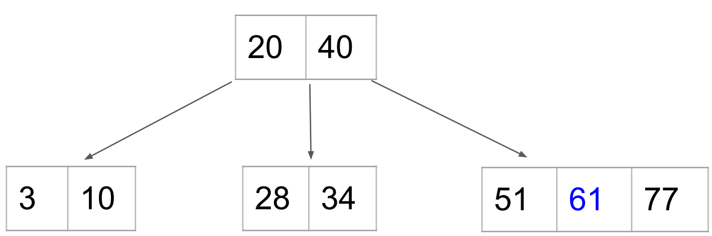
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14



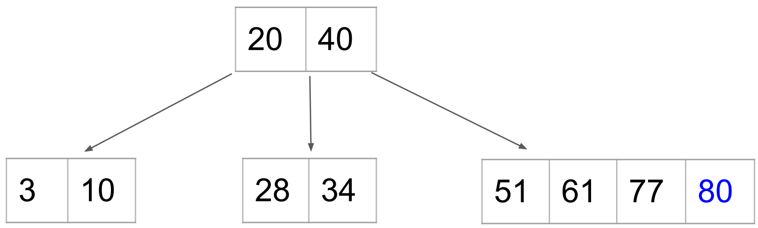
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14



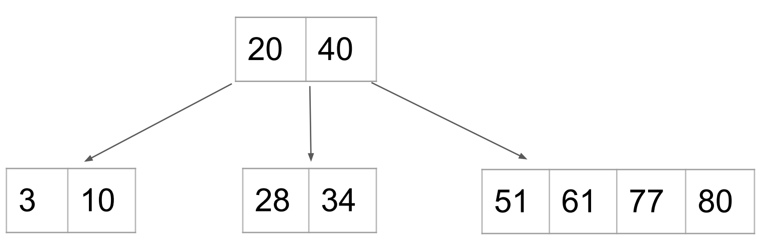
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14



We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

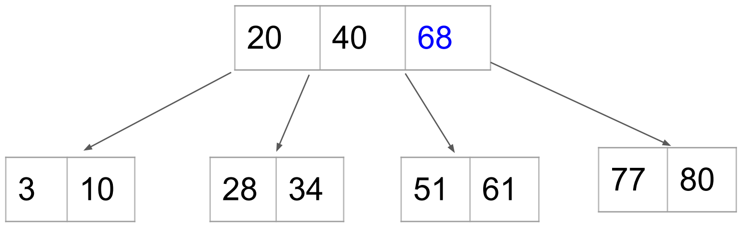
51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14



Inserting 68 requires another split, sending the middle item up a level

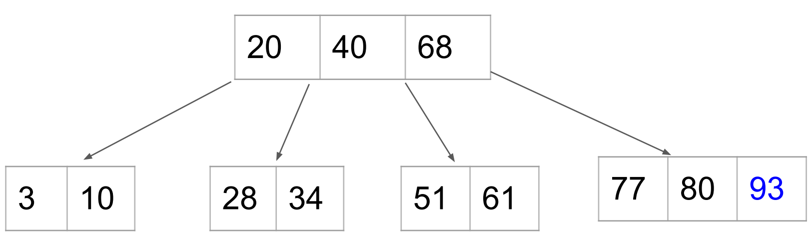
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



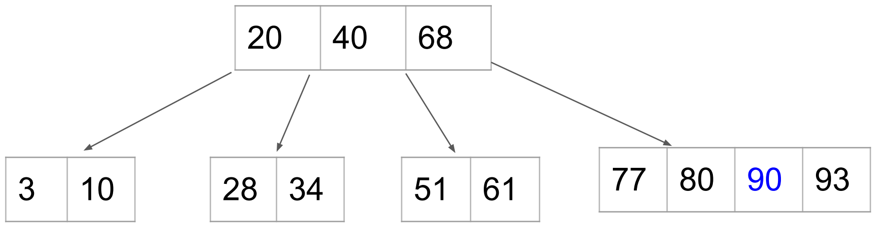
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



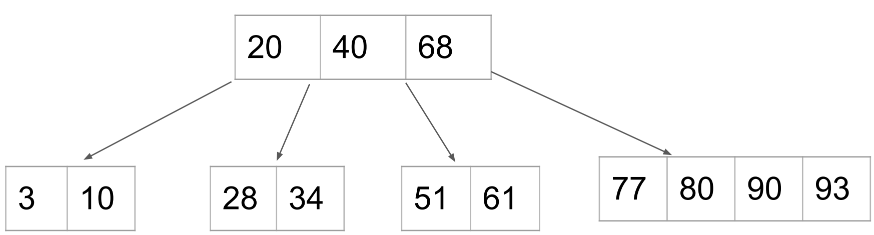
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

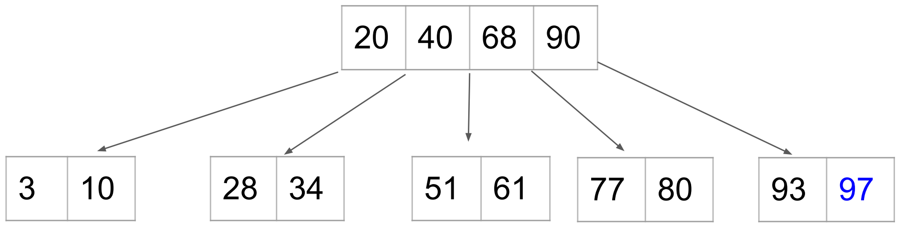
`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



Inserting 97 requires another split, sending the middle item up a level

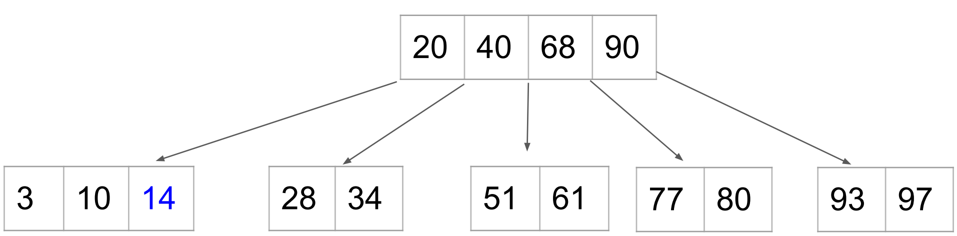
We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

`51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14`



We want to construct a B-tree where *m* = 2 for the following keys, in the specified order, from left to right:

```
51, 3, 10, 77, 20, 40, 34, 28, 61, 80, 68, 93, 90, 97, 14
```