Trees

Topical Review Session 05

CS 61B // Spring 2021

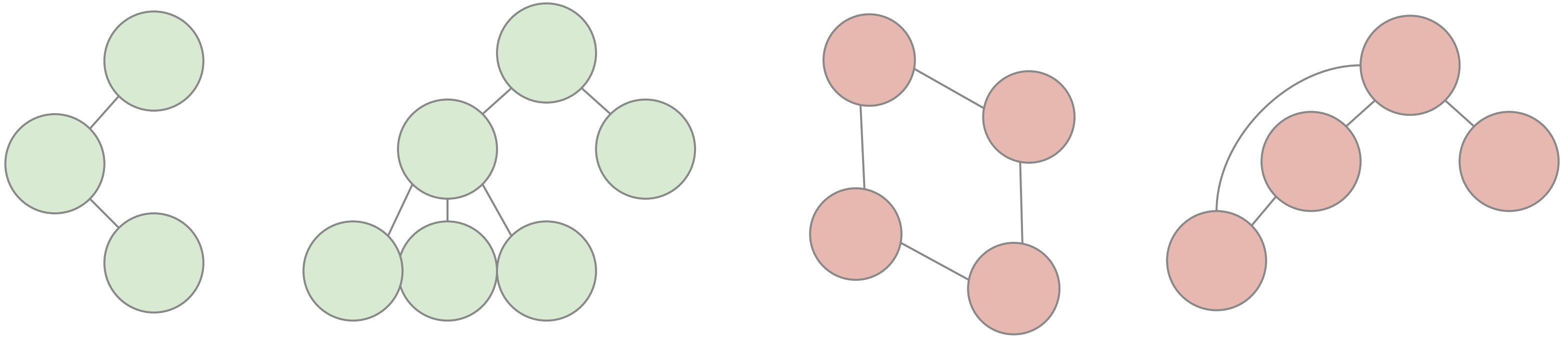
Agenda

Trees

Introduction to Trees

Introduction to Trees

- A **tree** is a special kind of graph with some extra constraints
- All the following rules apply:
    - A tree has N nodes and N-1 edges
    - A tree is a fully connected, acyclic graph
    - If you delete any edge in a tree, it becomes disconnected
    - There is exactly 1 path connecting any two nodes

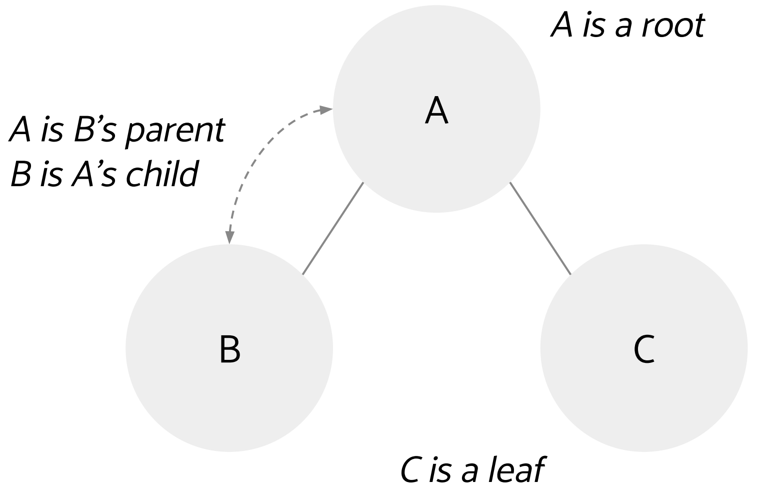


Introduction to Trees

A parent node points towards its child.

A root node is one that has no parent.

A leaf node is one that has no child.



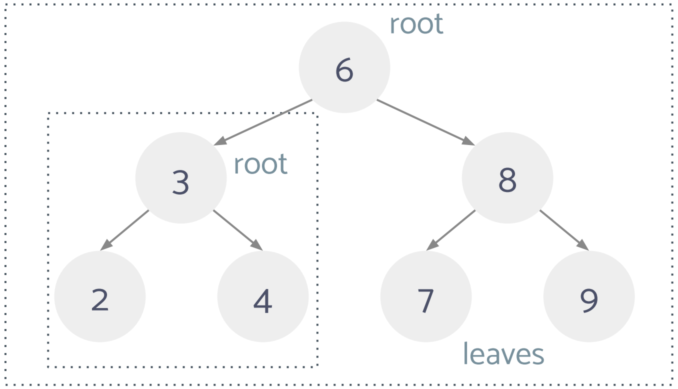
Binary Search Trees

Binary Search Trees

A **Binary Search Tree** (BST) is a data structure that allows us to easily maintain comparable elements.

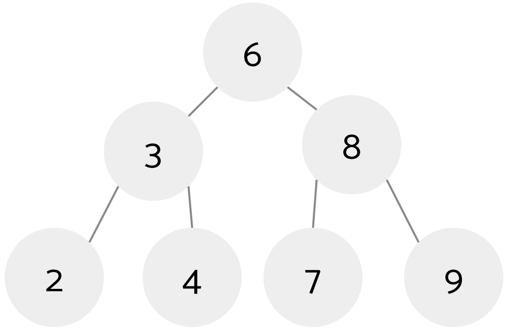
Key Properties:

1. Every node is the root of a smaller sub tree with two children
2. Every node's left child has a value lesser than the parent node `n.left < n`
3. Every node's right child has a value greater than the parent node `n.right > n`



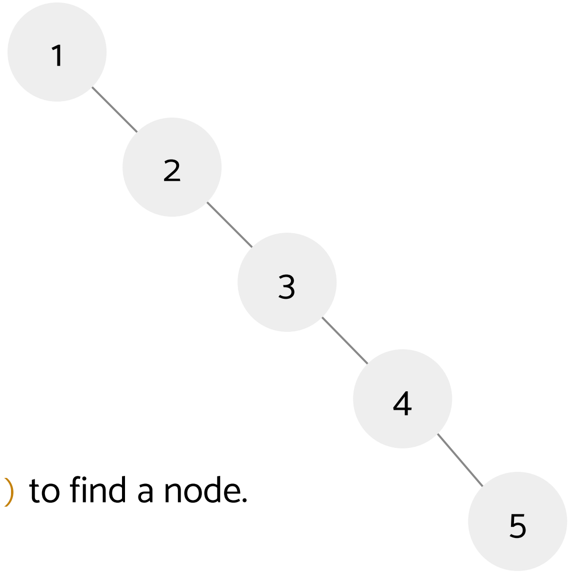
Binary Search Trees

Bushy BSTs are extremely fast because every node has two children.



`Θ(log N)` to find a node.

Spindly BSTs are extremely slow because every node has one child. Think linked list.

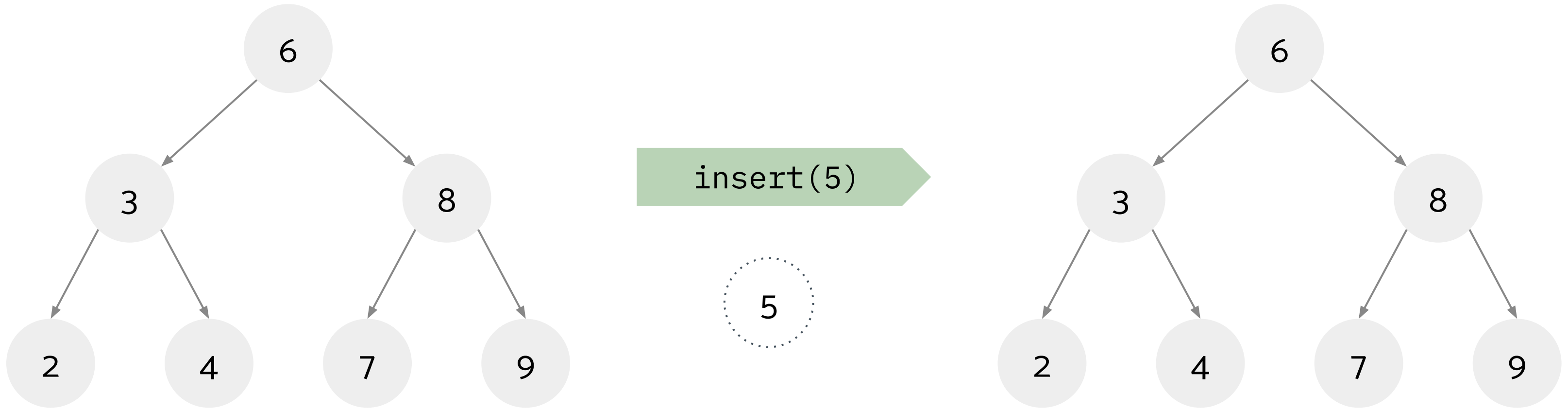


`Θ(N)` to find a node.

In general, the time complexity to find a node is `O(height)`

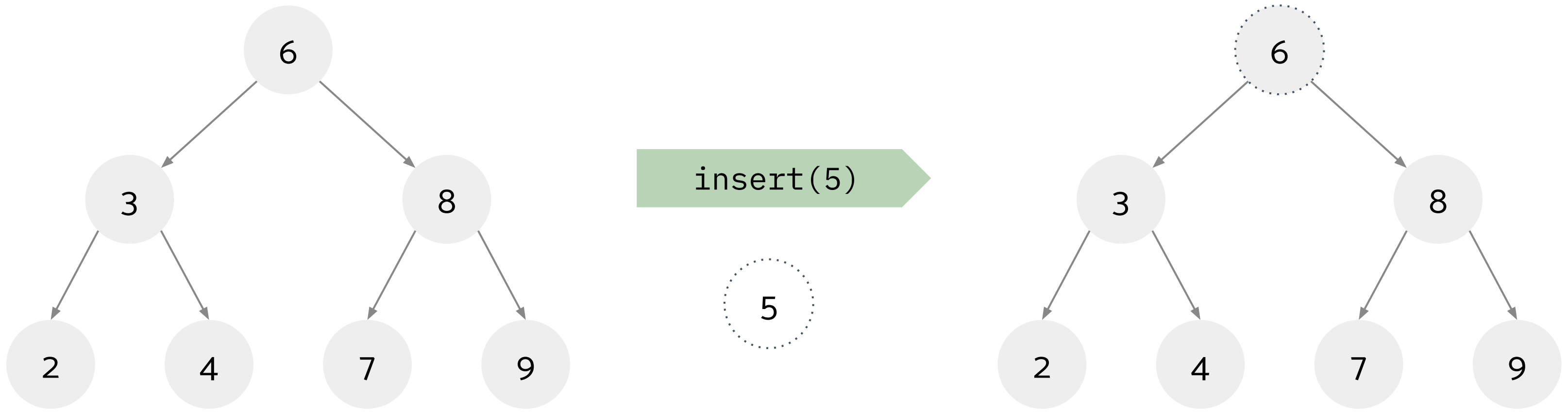
BST Insertion

New elements are always inserted as leaves.



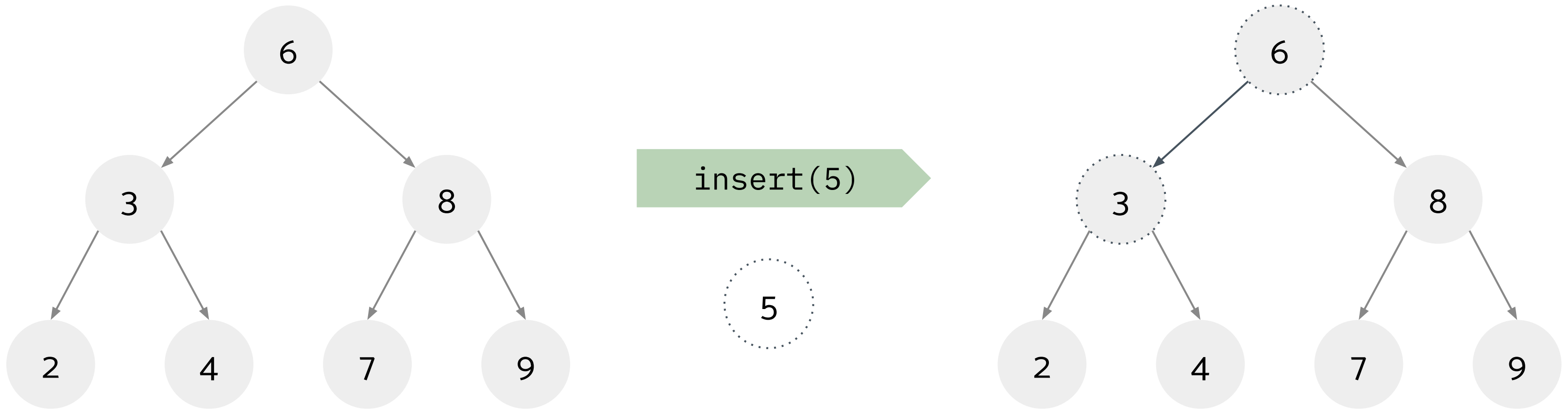
BST Insertion

New elements are always inserted as leaves.



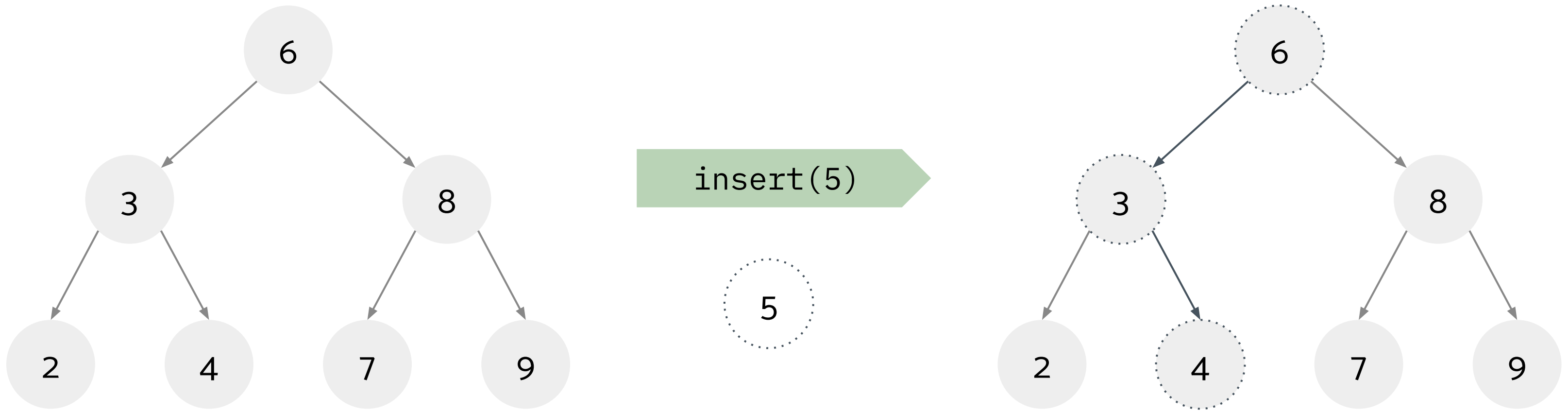
BST Insertion

New elements are always inserted as leaves.



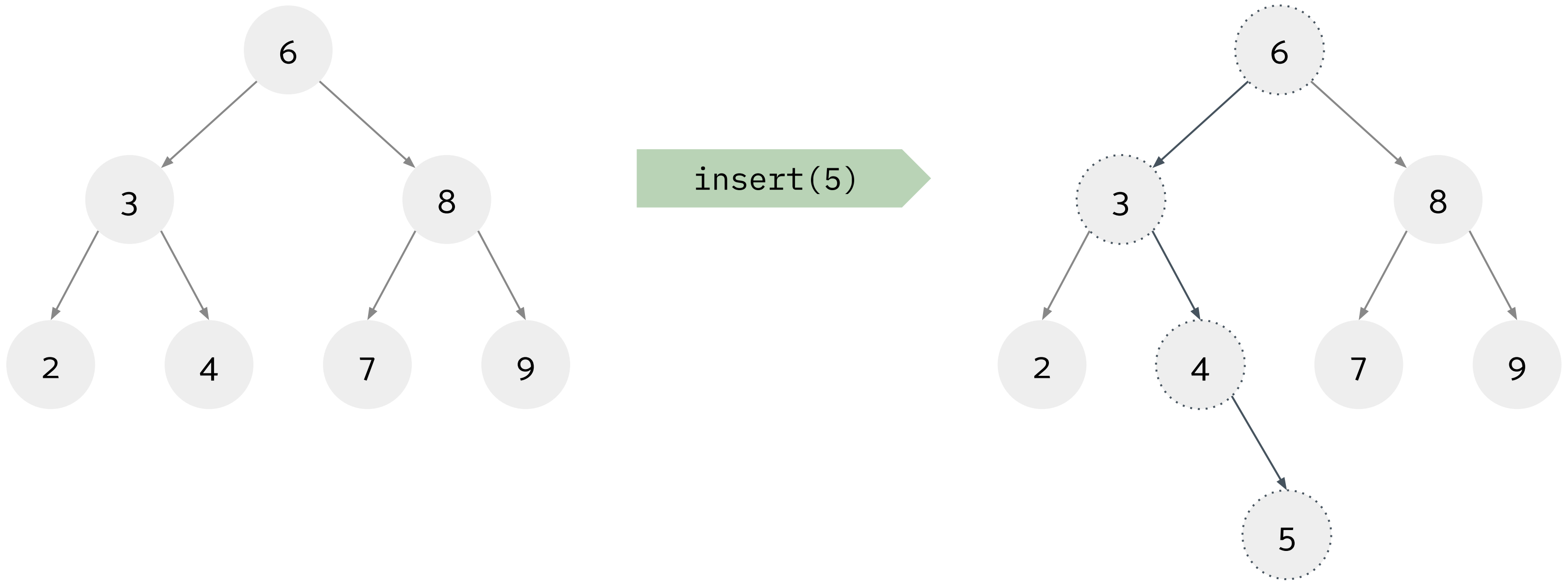
BST Insertion

New elements are always inserted as leaves.



BST Insertion

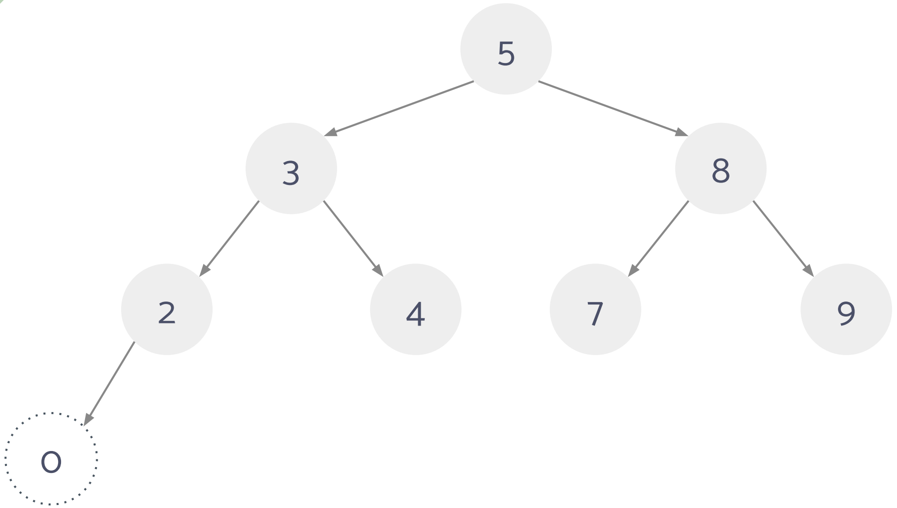
New elements are always inserted as leaves. This takes `O(log N)` time.



BST Deletion: Zero Children 🧍‍♂️

Deleting with no children is the simplest case - the node just disappears.

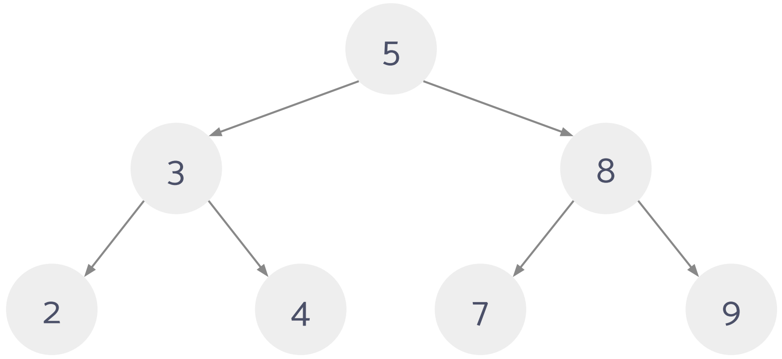
`delete(0)`



BST Deletion: Zero Children

Deleting with no children is the simplest case - the node just disappears.

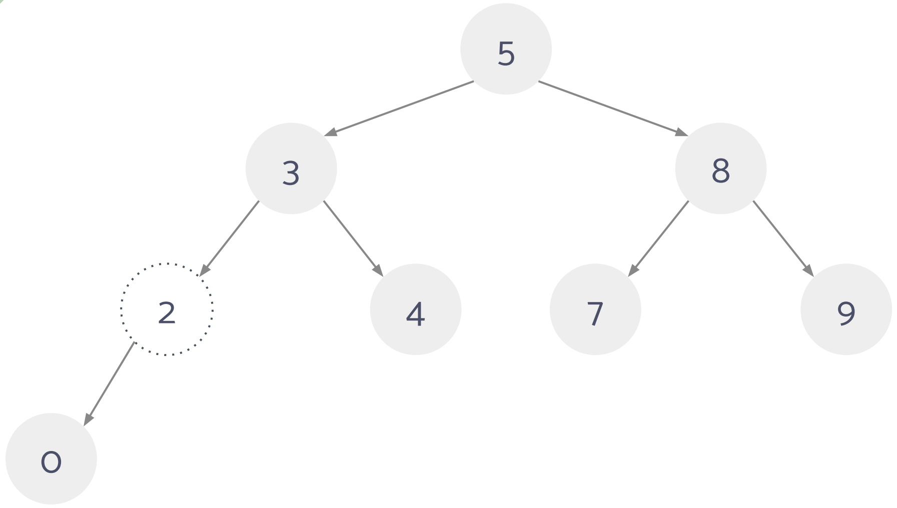
`delete(0)`



BST Deletion: One Child 👩‍👦

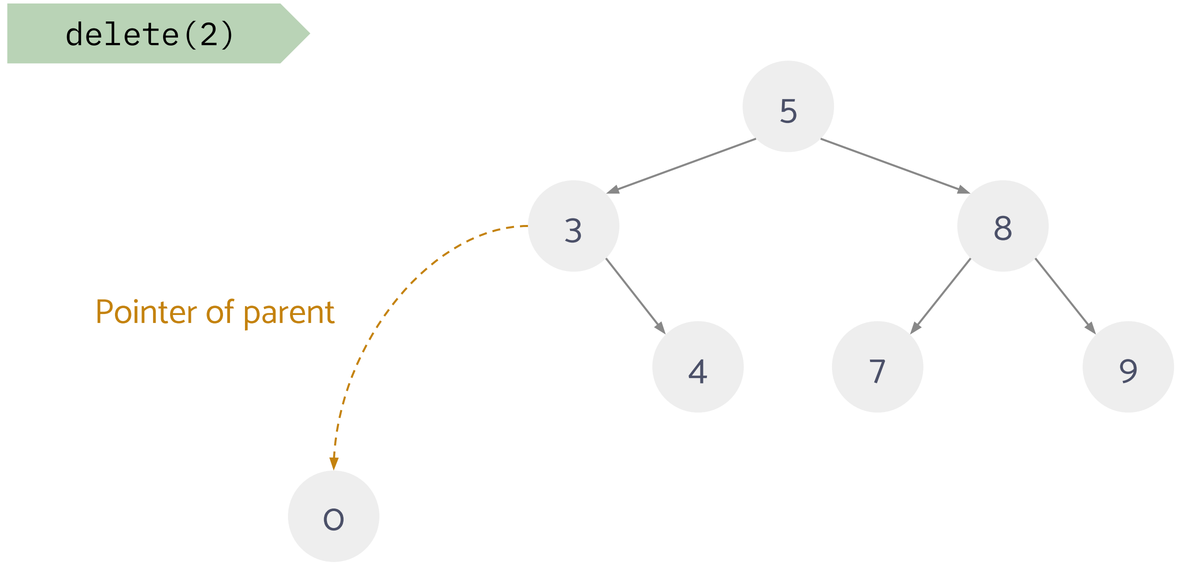
The child can replace the parent that gets deleted.

`delete(2)`



BST Deletion: One Child 👩‍👦

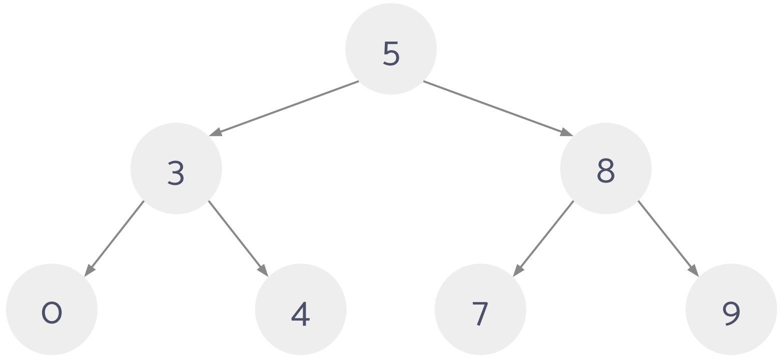
The child can replace the parent that gets deleted.



BST Deletion: One Child 👩‍👦

The child can replace the parent that gets deleted.

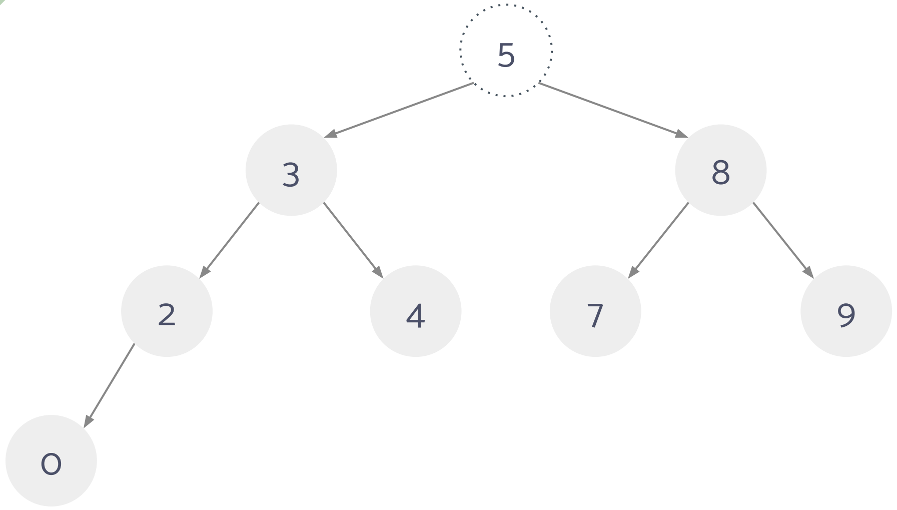
`delete(2)`



BST Deletion: Two Children 👨‍👧‍👦

We replace the node with either the largest descendent on the left or the smallest descendent on the right.

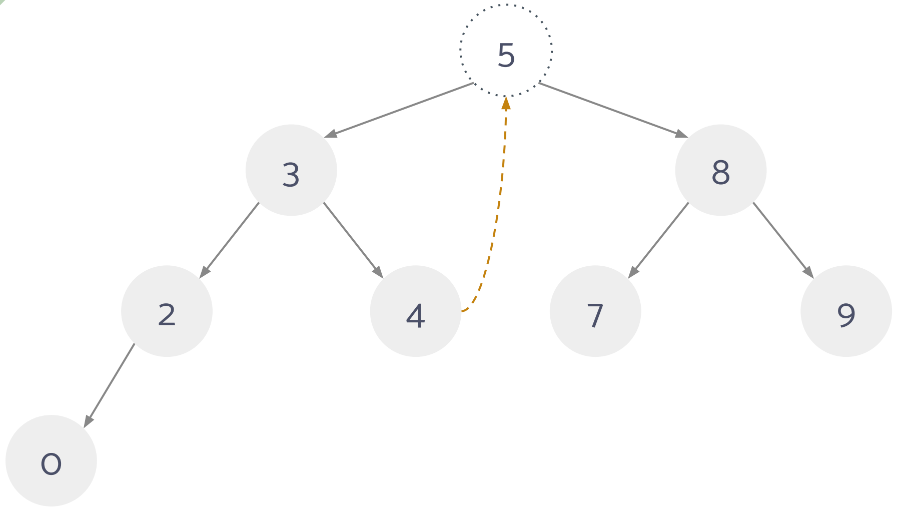
`delete(5)`



BST Deletion: Two Children 👨‍👧‍👦

We replace the node with either the largest descendent on the left or the smallest descendent on the right.

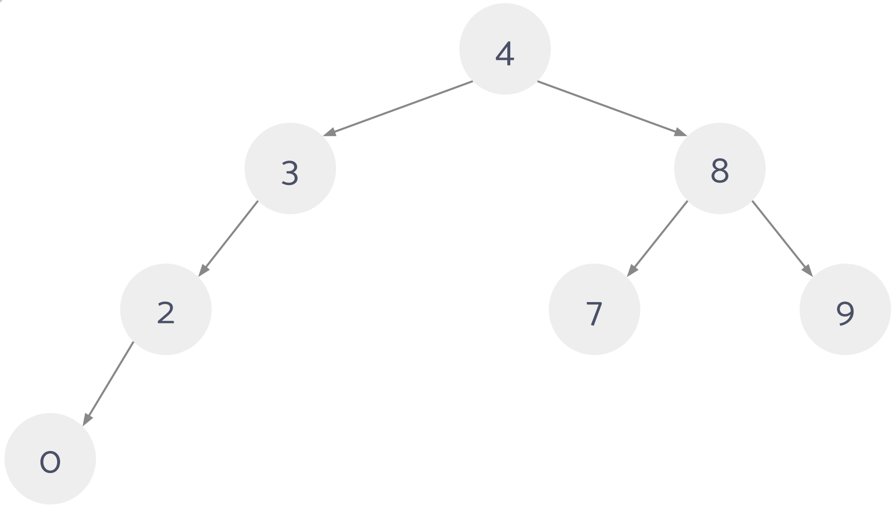
`delete(5)`



BST Deletion: Two Children 👨‍👧‍👦

We replace the node with either the largest descendent on the left or the smallest descendent on the right.

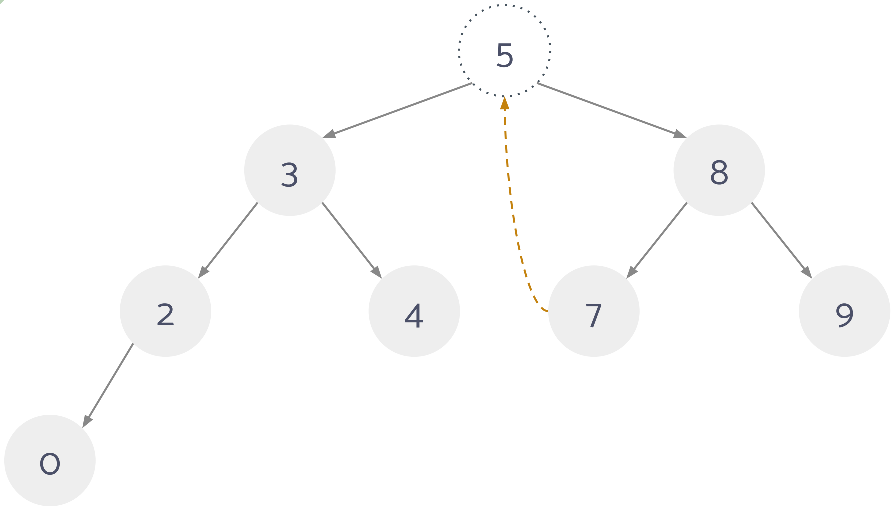
`delete(5)`



BST Deletion: Two Children 👨‍👧‍👦

We replace the node with either the largest descendent on the left or the smallest descendent on the right.

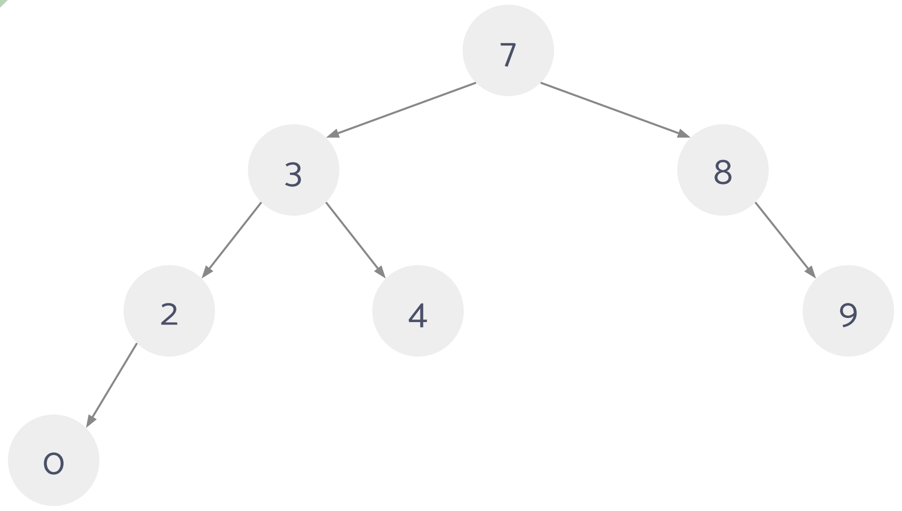
`delete(5)`



BST Deletion: Two Children 👨‍👧‍👦

We replace the node with either the largest descendent on the left or the smallest descendent on the right.

`delete(5)`



Asymptotics of BST

Bushy BST - Best Case

Search: `Θ(log N)`

Insertion: `Θ(log N)`

Deletion: `Θ(log N)`

☺️

Spindly BST - Worst Case

Search: `Θ(N)`

Insertion: `Θ(N)`

Deletion: `Θ(N)`

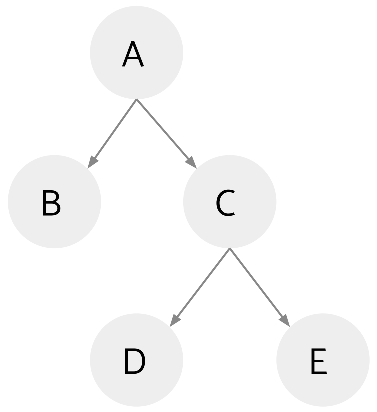
☹️

Tree Traversals

Breadth First Search

Breadth first search means we visit the nodes of a tree level by level, also known as level order traversals.

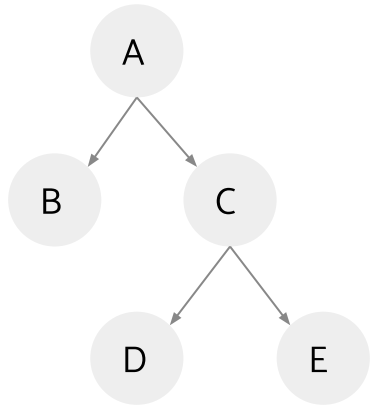
BFS is usually done using a queue.



Breadth First Search

Breadth first search means we visit the nodes of a tree level by level, also known as level order traversals.

BFS is usually done using a queue.



A B C D E

Depth First Search

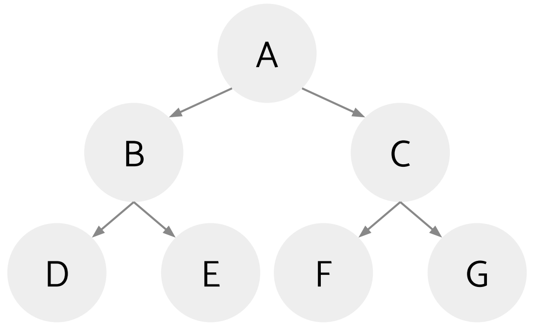
Three types of DFS: **Preorder**, **Inorder**, **Postorder**. Overall, we traverse deeper nodes before nodes that are closer to the root. Usually, we can implement this recursively or iteratively.

DFS is usually done iteratively using a stack.

Depth First Search: Preorder

In **preorder traversal**, we visit a node, then traverse its children.

```
preOrder(BSTNode x) {
    if (x == null)
        return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

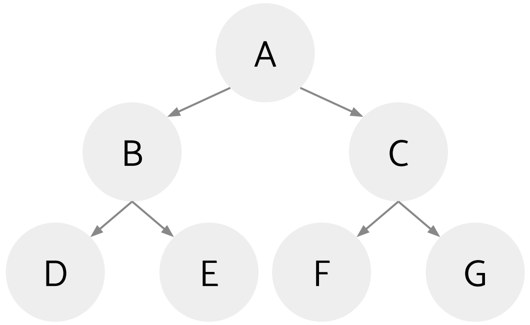


A B D E C F G

Depth First Search: Inorder

In **inorder**, we traverse left child, visit, then traverse right child.

```
inOrder(BSTNode x) {
    if (x == null)
       return;
    inOrder(x.left)
    print(x.key)
    inOrder(x.right)
}
```

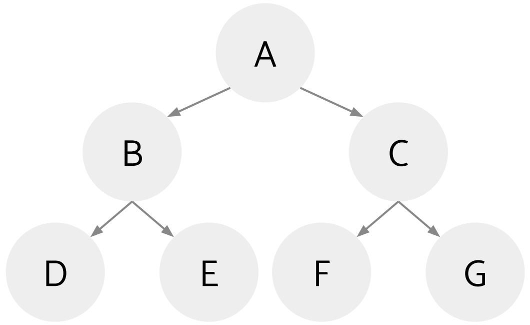


D B E A F C G

Depth First Search: Postorder

In **postorder**, we traverse left, traverse right, then visit

```
postOrder(BSTNode x) {
    if (x == null)
       return;
    postOrder(x.left)
    postOrder(x.right)
    print(x.key)
}
```

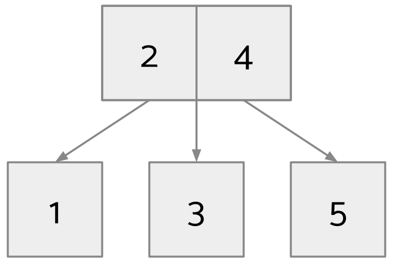


D E B F G C A

B-Trees

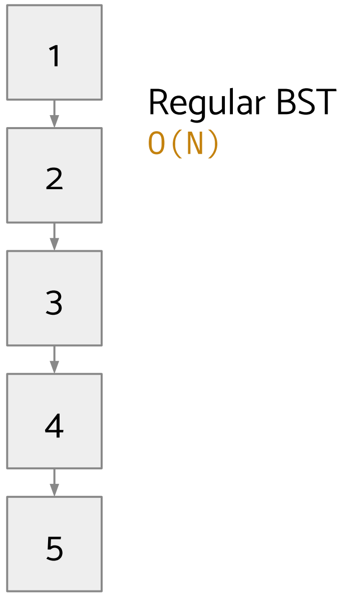
2-3 Trees

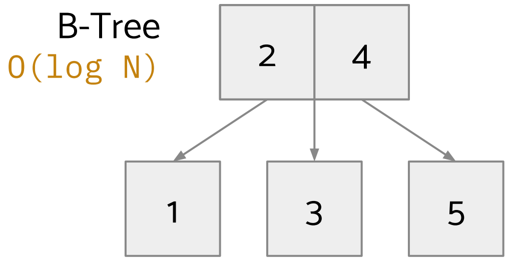
- **2-3 Trees** are a modified representation of BSTs, so they still follow the left child smaller than parent, right child larger than parent rule.
- A node may contain up to 2 values
- Non-leaf nodes can have 2 or 3 children



Why B-Trees?

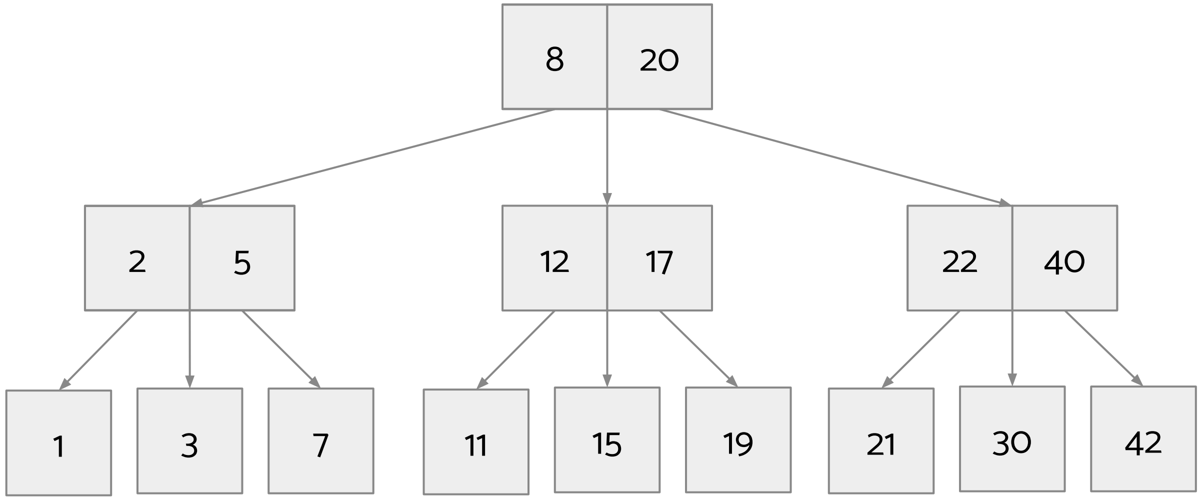
Let’s say we insert [1, 2, 3, 4, 5] in order.





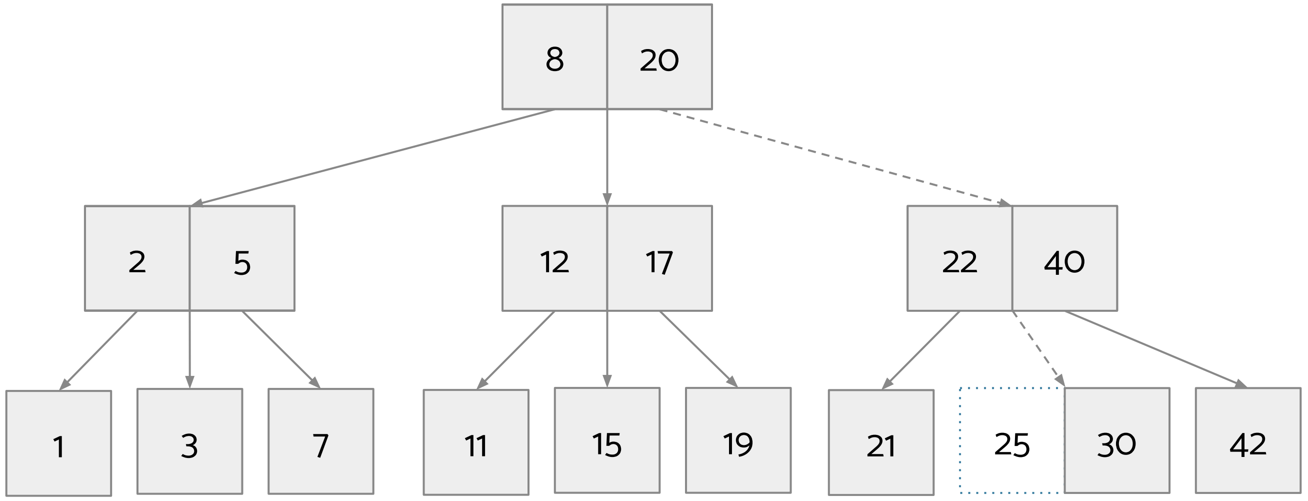
B-Tree Insertion

`insert(25)`



B-Tree Insertion

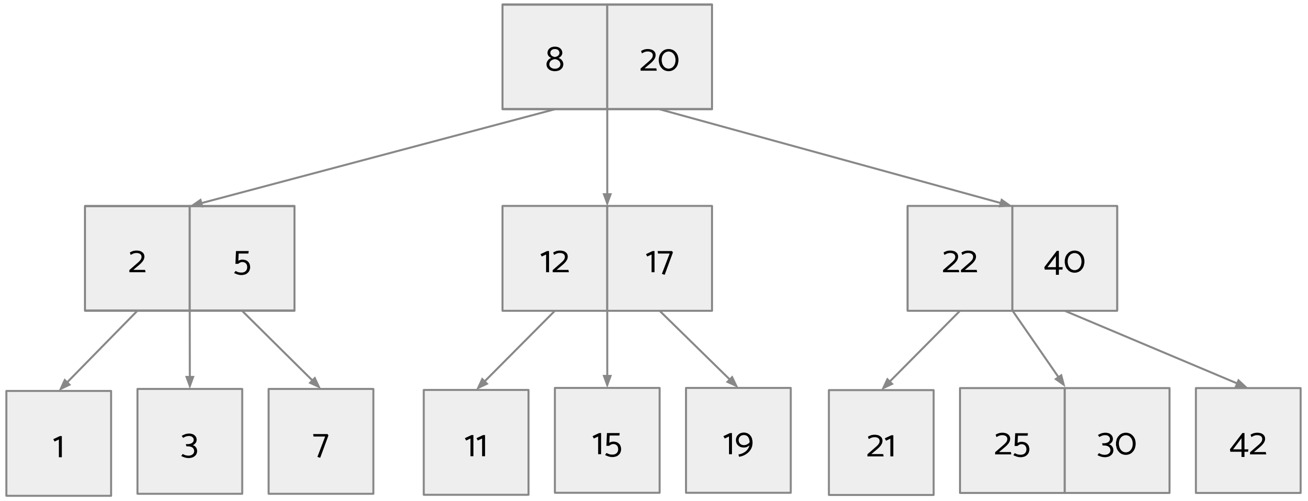
`insert(25)`



`25` is smaller than `30`, but larger than `22`, so it would be inserted in the leaf node to the left of `30`.
Because the node only has two elements, no additional steps are required.

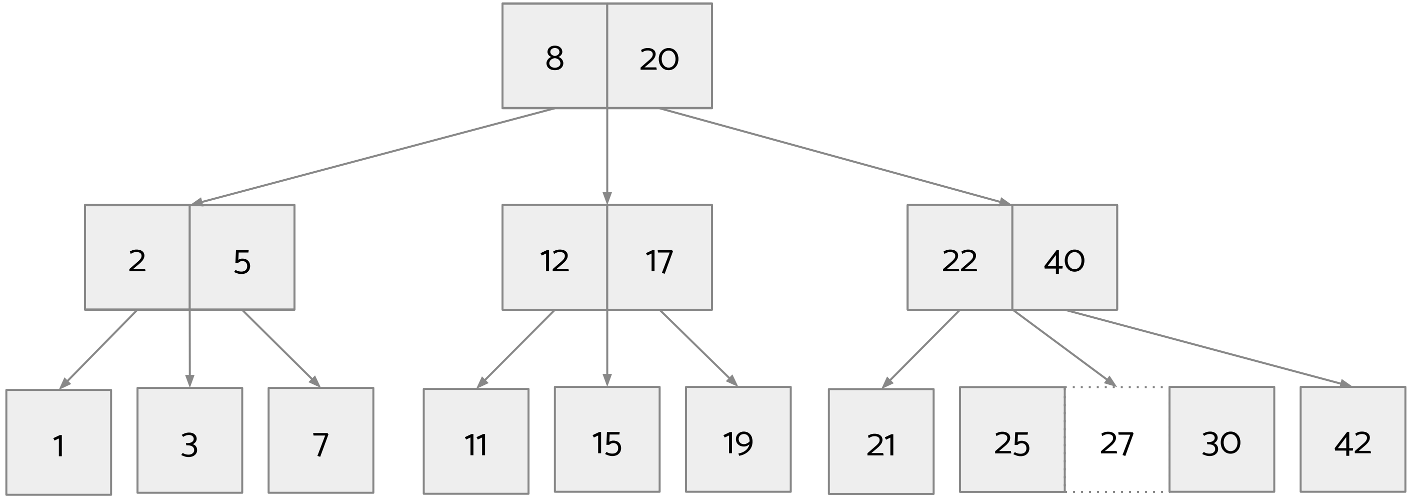
B-Tree Insertion

`insert(27)`



B-Tree Insertion

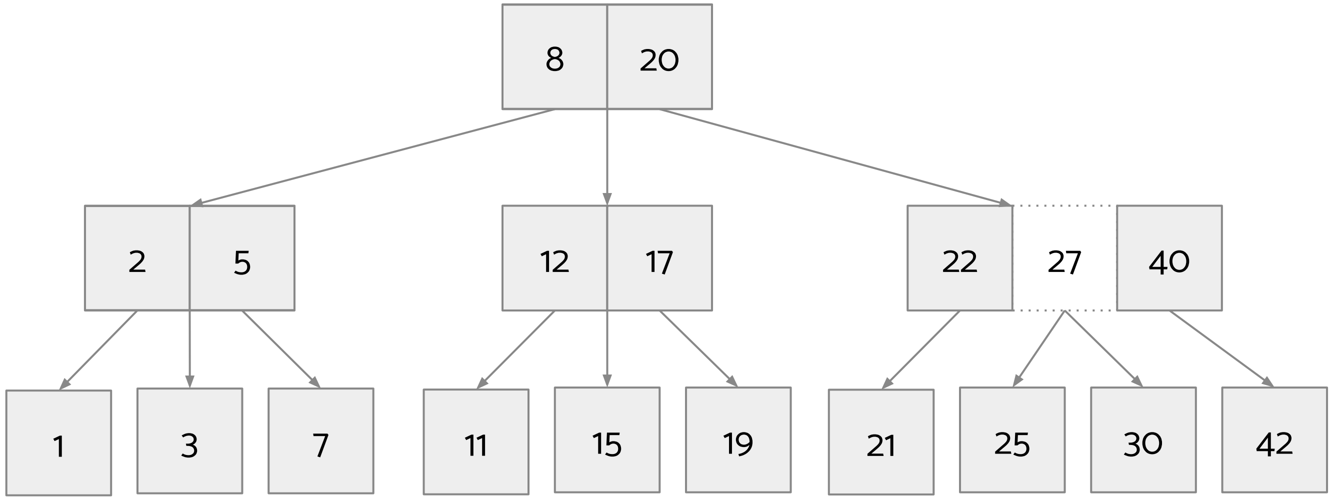
`insert(27)`



`27` goes in between `25` and `30`, but now we have a node with three elements, so we have to push the middle element (27) up.

B-Tree Insertion

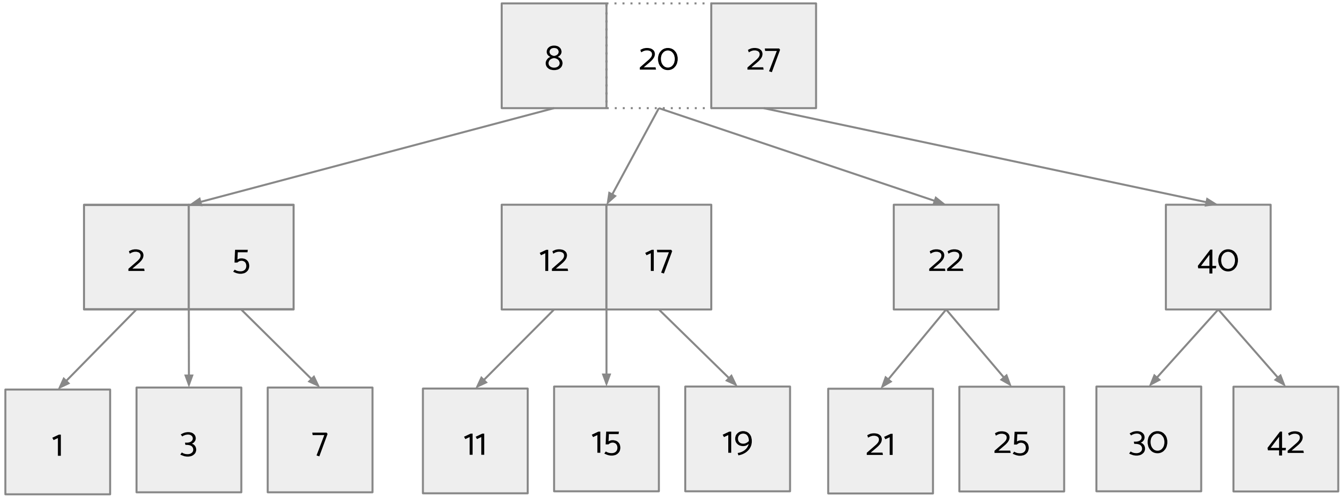
`insert(27)`



Now the parent node of `22`, `27`, and `40` also has three elements, so we push the middle element (`27`) up again. Note how the `25` and the `30` were split up to maintain the correct relationship between the parent node and its left/right children.

B-Tree Insertion

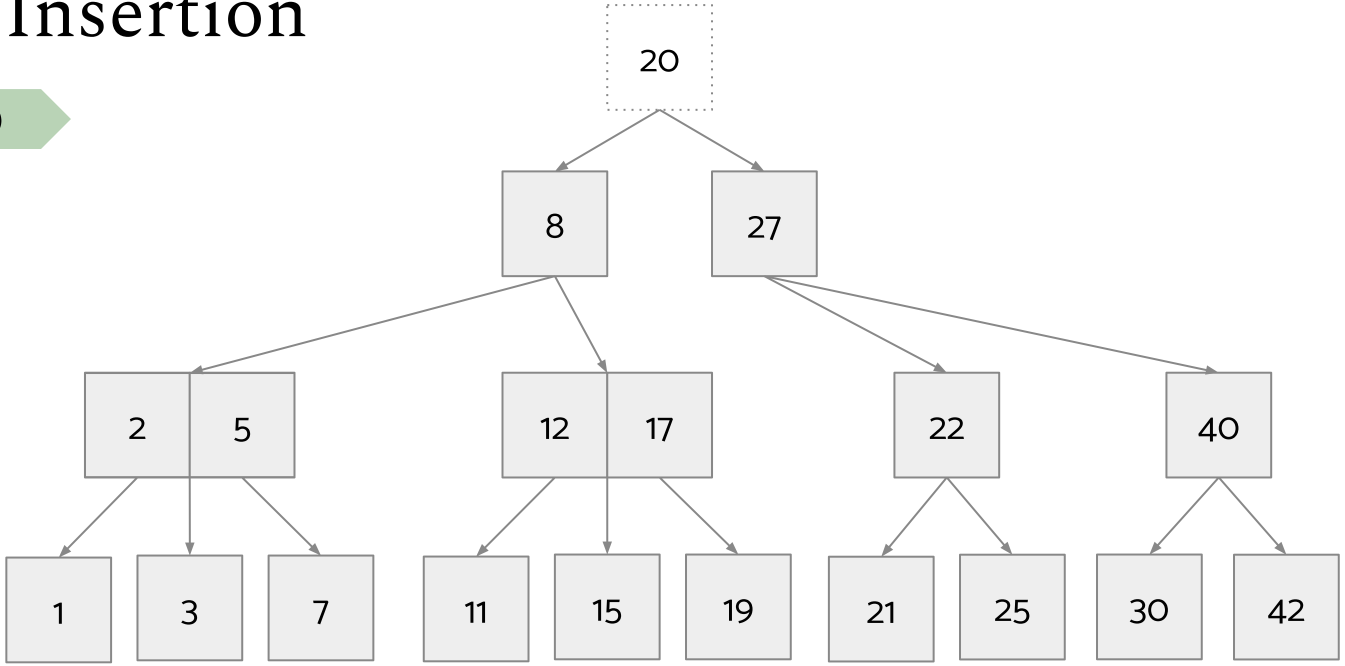
`insert(27)`



Now the root with `8`, `20`, and `27` has three elements, so we repeat the pushing process one more time.

B-Tree Insertion

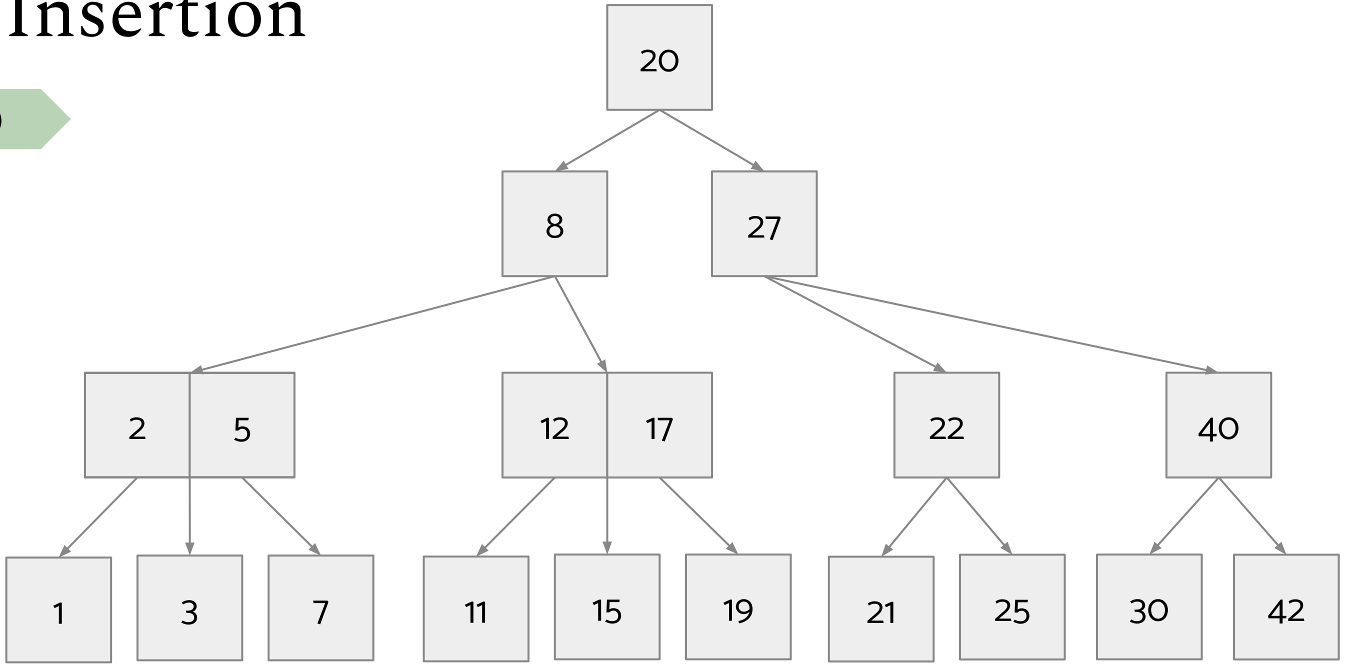
`insert(27)`



And we're done! The tree is still balanced :)

B-Tree Insertion

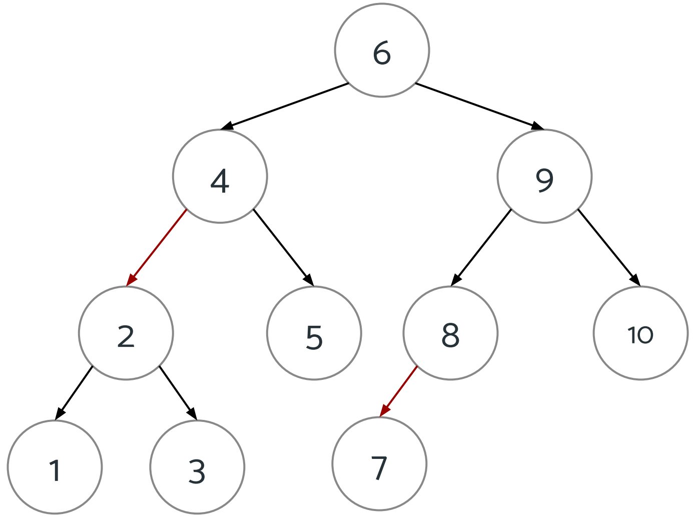
`insert(27)`



Red Black Trees

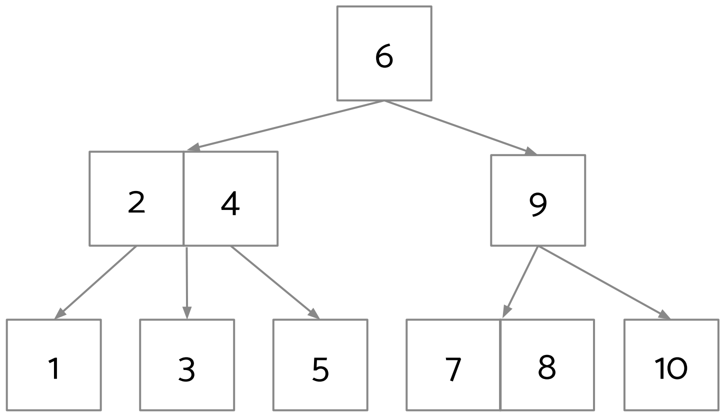
Red Black Trees

- **Red Black Trees** are BSTs that retain the properties from B-Trees
- **Black Edges** are the equivalent of a normal edge in the corresponding 2-3 tree
- **Red Edges** are the equivalent of the implicit connections between values in the same node in the corresponding 2-3 tree
- The version we use in this class is called a **Left Leaning Red Black Trees** because our red edges only point to the left.

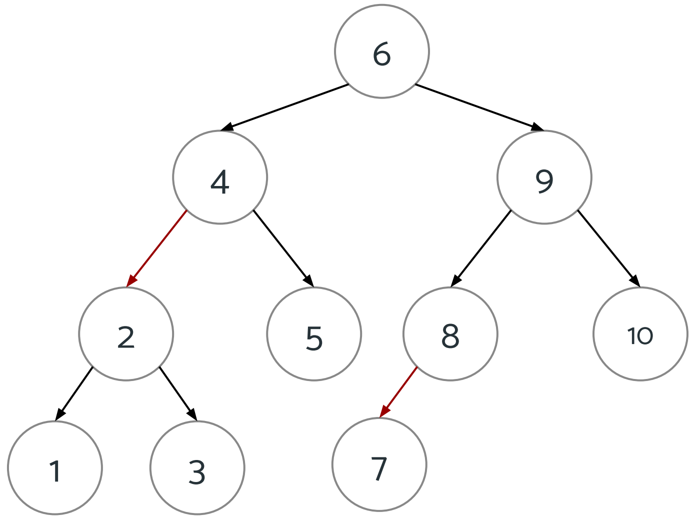


Converting Between Trees

2-3 Tree

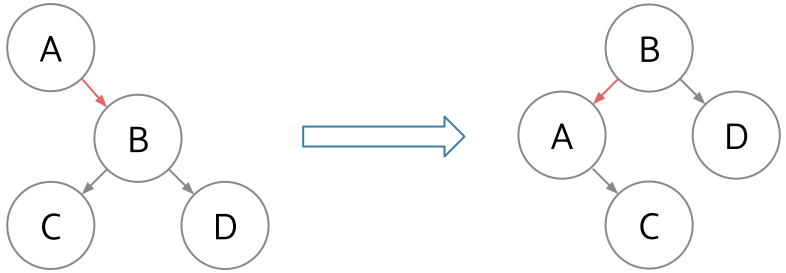


Red Black Tree

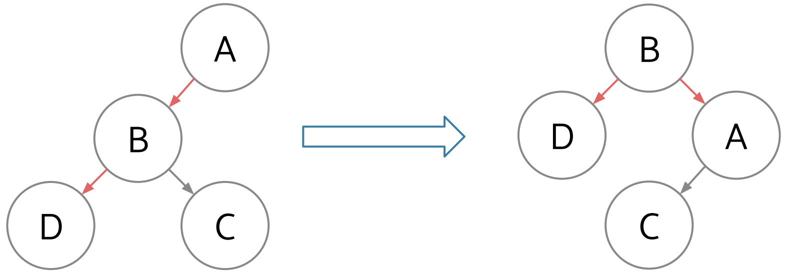


LLRB Balancing Rotations

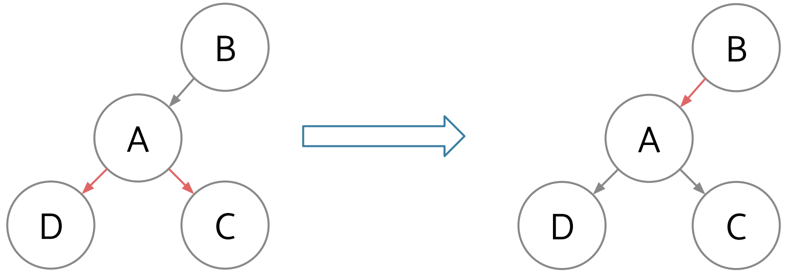
```
rotateLeft(A);
```



```
rotateRight(A);
```

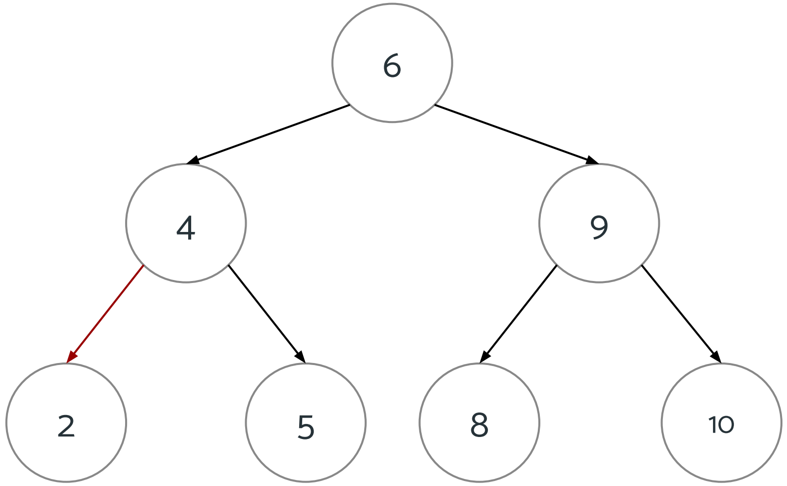


```
colorFlip(A);
```



LLRB Insertion

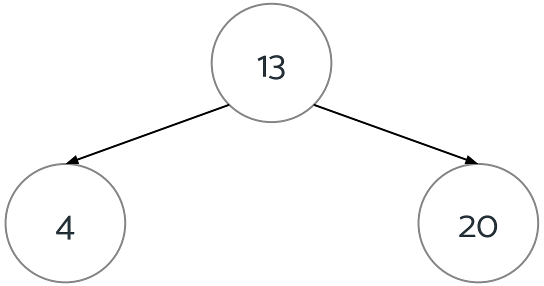
Nodes are always added as red connections.



LLRB Insertion

Nodes are always added as red connections.

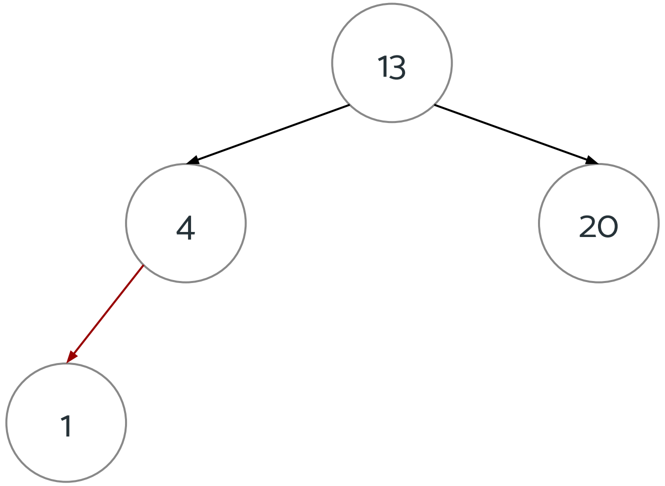
`insert(1)`



LLRB Insertion

Nodes are always added as red connections.

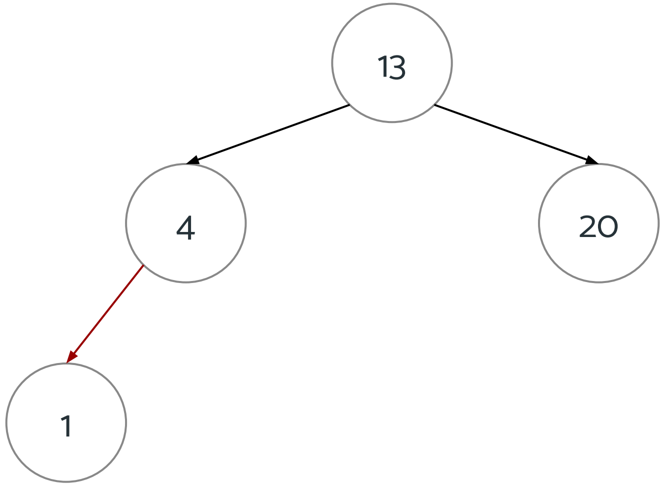
`insert(1)`



LLRB Insertion

If nodes are added to the right, the tree must be rotated until it is left leaning again.

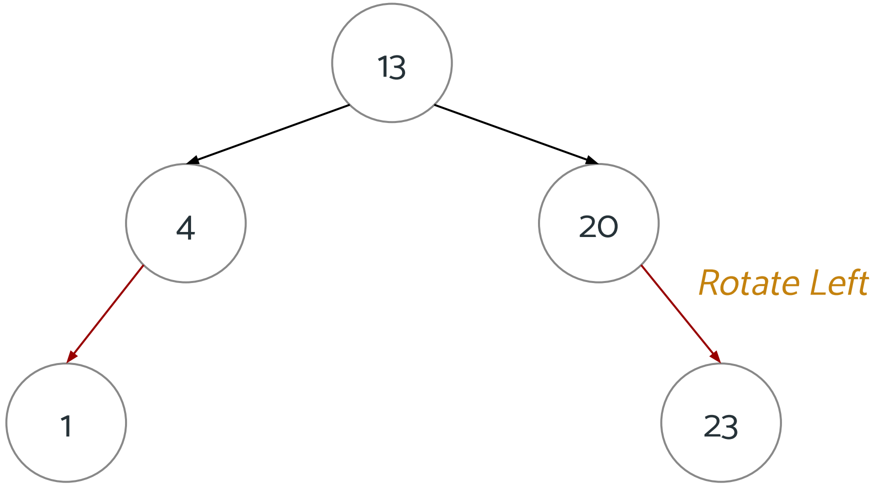
`insert(23)`



LLRB Insertion

If nodes are added to the right, the tree must be rotated until it is left leaning again.

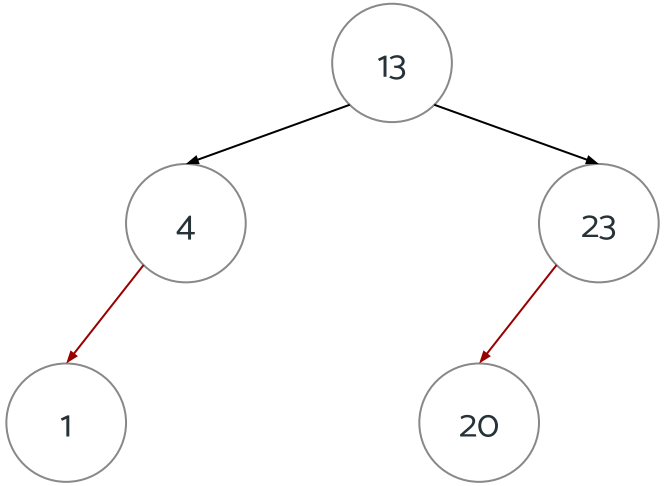
`insert(23)`



LLRB Insertion

If nodes are added to the right, the tree must be rotated until it is left leaning again.

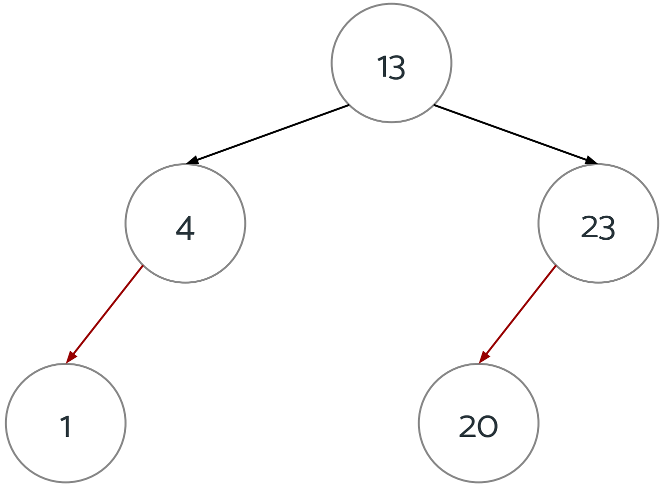
`insert(23)`



LLRB Insertion

If both the left and right branches are red, we can perform a color flip.

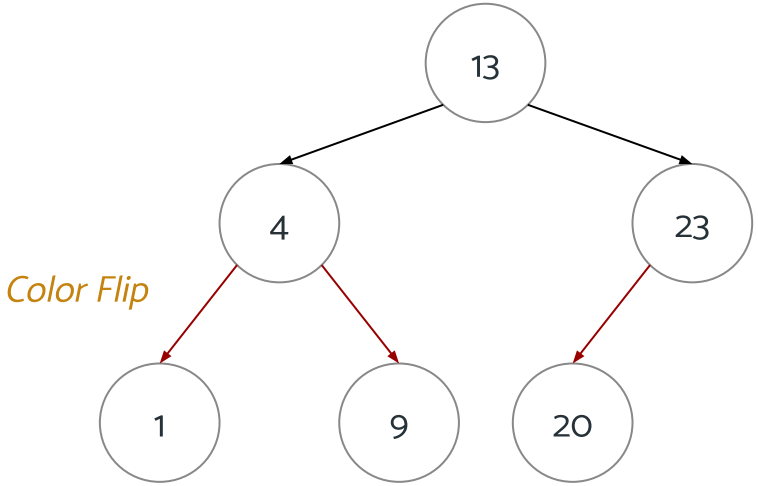
`insert(9)`



LLRB Insertion

If both the left and right branches are red, we can perform a color flip.

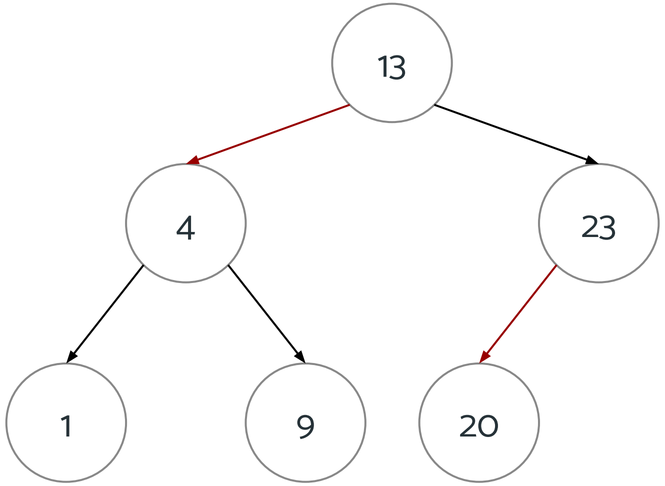
`insert(9)`



LLRB Insertion

If both the left and right branches are red, we can perform a color flip.

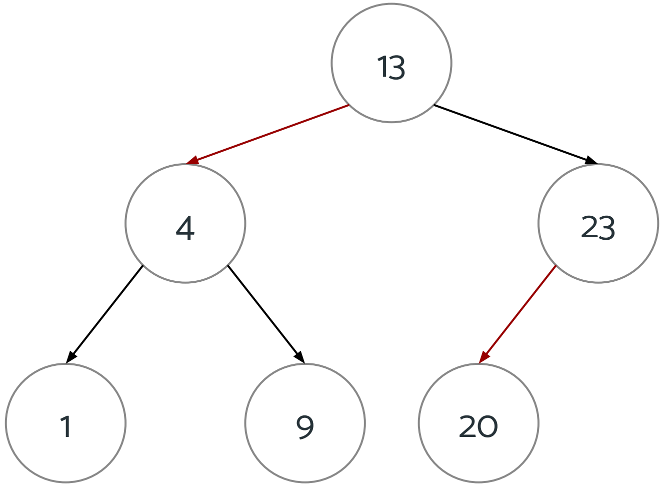
`insert(9)`



LLRB Insertion

Sometimes you will need to do multiple operations.

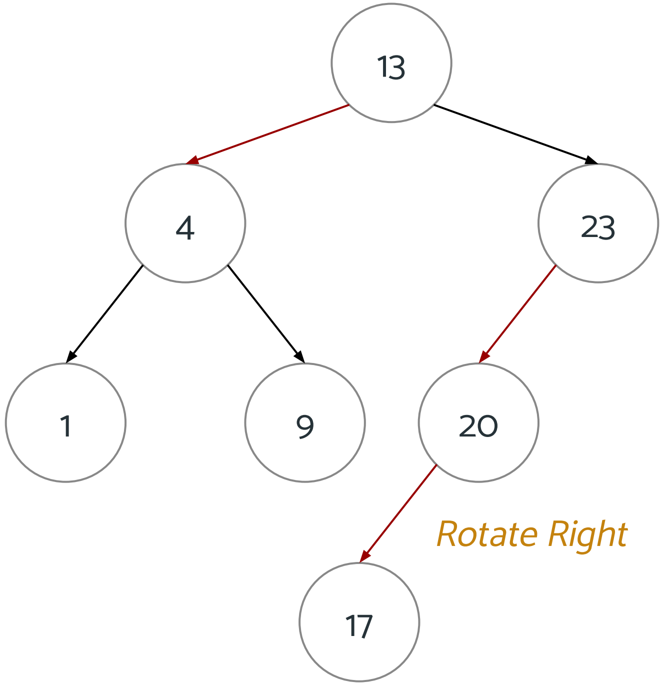
`insert(17)`



LLRB Insertion

Sometimes you will need to do multiple operations.

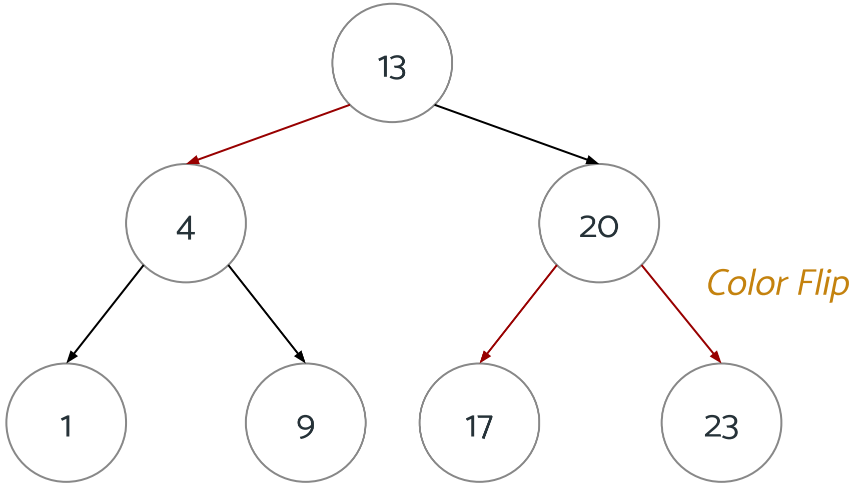
`insert(17)`



LLRB Insertion

Sometimes you will need to do multiple operations.

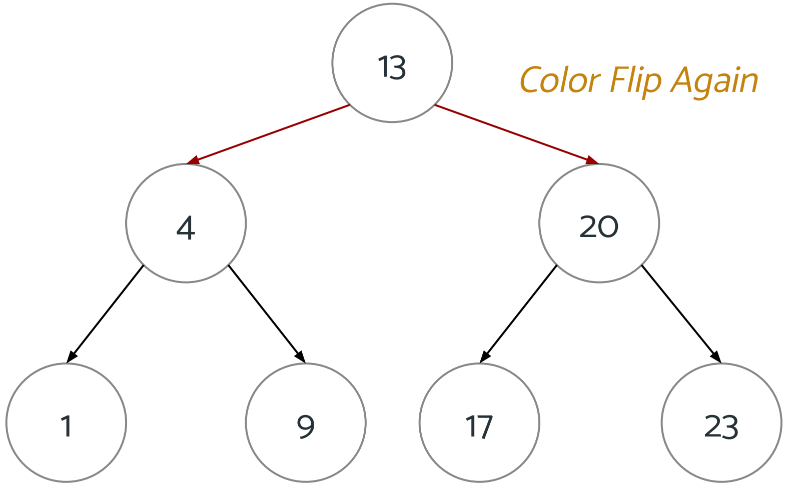
`insert(17)`



LLRB Insertion

Sometimes you will need to do multiple operations.

`insert(17)`

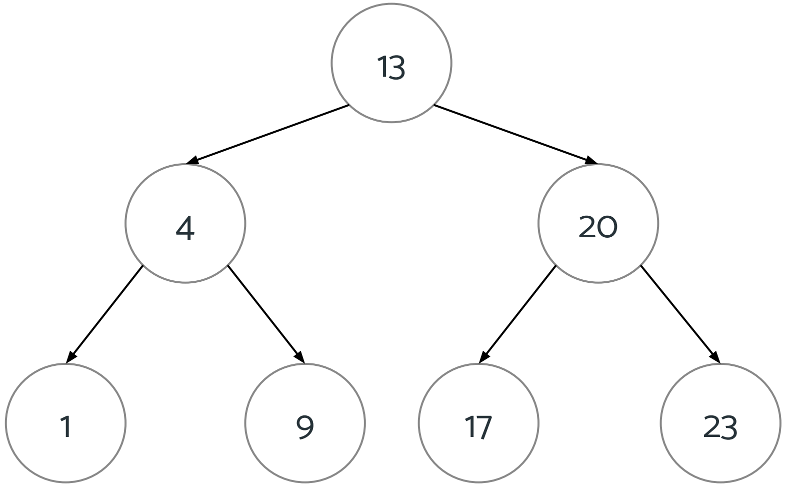


Color Flip Again

LLRB Insertion

Sometimes you will need to do multiple operations.

`insert(17)`



Summary of Red Black Trees

A balancing operation is necessary under the following conditions:

1. A right leaning branch is present
2. There are two left leaning branches in a row

How do we make sure an LLRB is valid?

1. It is still a BST - all those rules apply
2. Once you convert it to a B-Tree, it should follow all the rules laid out for B-Trees
3. Every path from the root to a null reference must have the same number of Black links