Print Copy

Print functionality is for user convenience only, not all interactive content is fully printable.

This print version is for subscriber's personal use only. Not to be posted or distributed.

Print Cancel
Rendering chapter...

©zyBooks 05/10/21 12:45 728163
Neha Maddali
IASTATECOMS228Spring2021

7.1 AVL: A balanced tree

Balanced BST

An AVL tree is a BST with a height balance property and specific operations to rebalance the tree when a node is inserted or removed. This section discusses the balance property; another section discusses the operations. A BST is height balanced if for any node, the heights of the node's left and right subtrees differ by only 0 or 1.

A node's balance factor is the left subtree height minus the right subtree height, which is 1, 0, or -1 in an AVL tree.

Recall that a tree (or subtree) with just one node has height 0. For calculating a balance factor, a non-existent left or right child's subtree's height is said to be -1.

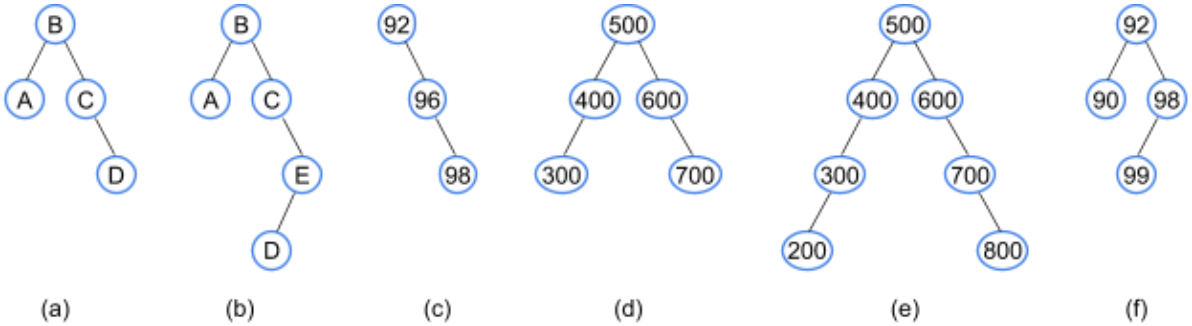
participation activity
7.1.1: An AVL tree is height balanced: For any node, left and right subtree heights differ by only 0 or 1.

Animation captions:

1. Every AVL tree node's balance factor (left minus right heights) is -1, 0, or 1.
2. If any node's subtree heights differ by 2 or more, the entire tree is not an AVL tree.

participation activity
7.1.2: AVL trees.

Indicate whether each tree is an AVL tree.



1)

(a)

- ○ Yes
- ○ No

2)

(b)

- ○ Yes
- ○ No

3)

(c)

- ○ Yes
- ○ No

4)

(d)

- ○ Yes
- ○ No

5)

(e)

- ○ Yes
- ○ No

6)

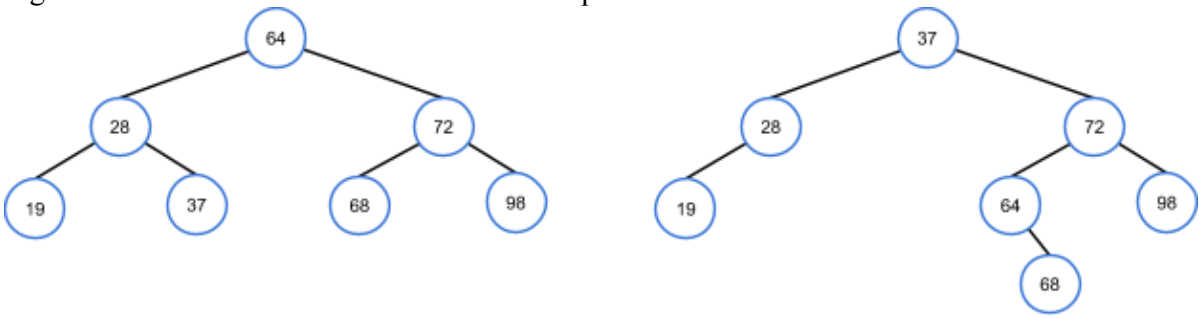
(f)

- ○ Yes
- ○ No

AVL tree height

Minimizing binary tree height yields fastest searches, insertions, and removals. If nodes are inserted and removed dynamically, maintaining a minimum height tree requires extensive tree rearrangements. In contrast, an AVL tree only requires a few local rotations (discussed in a later section), so is more computationally efficient, but doesn't guarantee a minimum height. However, theoreticians have shown that an AVL tree's worst case height is no worse than about 1.5x the minimum binary tree height, so the height is still O(log N) where N is the number of nodes. Furthermore, experiments show that AVL tree heights in practice are much closer to the minimum.

Figure 7.1.1: An AVL tree doesn't have to be a perfect BST.



Perfect BST with minimum possible height for 7 nodes

An AVL tree with a height close to the minimum

participation activity

7.1.3: AVL tree height.

1)

An AVL tree maintains the minimum possible height.

- ○ True
- ○ False

2)
What is the minimum possible height of an AVL tree with 7 nodes?

- ○ 2
- ○ 3
- ○ 5
- ○ 7

3)
What is the maximum possible height of an AVL tree with 7 nodes?

- ○ 2
- ○ 3
- ○ 5
- ○ 7

Storing height at each AVL node

An AVL tree implementation can store the subtree height as a member of each node. With the height stored as a member of each node, the balance factor for any node can be computed in O(1) time. When a node is inserted in or removed from an AVL tree, ancestor nodes may need the height value to be recomputed.

participation activity
7.1.4: Storing height at each AVL node.

Animation captions:

1. Adding node 55 requires height values for nodes 76 and 47 to be updated.
2. With updated height values at each node, balance factors can be computed. The height of any null subtree is -1.

participation activity
7.1.5: Storing height at each AVL node.
1)
What relationship does a node's height have to the node's balance factor?

- ○ Height equals balance factor.
- ○ A negative height implies a negative balance factor and a positive height implies a positive balance factor.
- ○ Absolute value of balance factor equals height.
- ○ No relationship.

2)
When adding a new node, what is true about the order in which the ancestor height values must be updated?

- ○ Height values must be updated starting at the node's parent, and moving up to the root.
- ○ Height values must be updated starting at the root and moving down to the node.
- ○ Height values can be updated top-down or bottom-up.
- ○ Height values can be updated in any order.

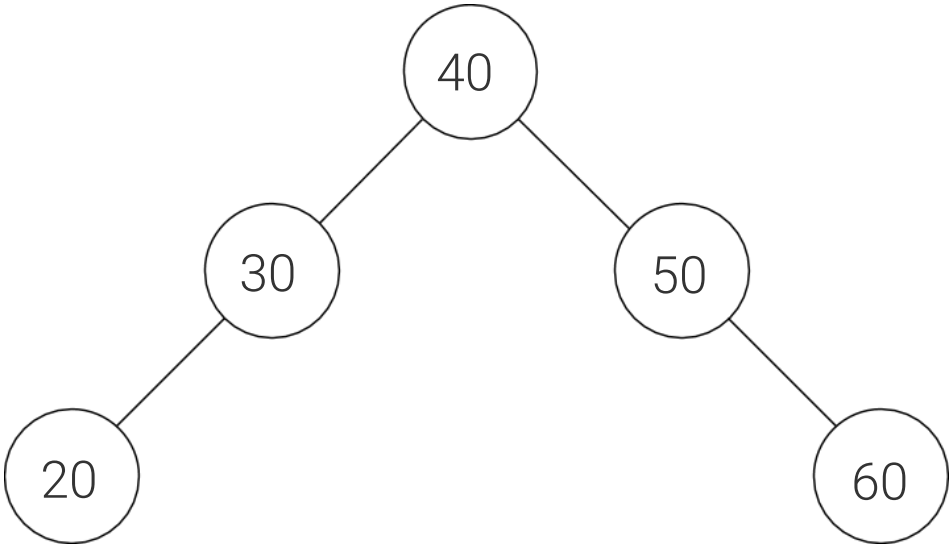
3)
When would inserting a new node to an AVL tree result in no height value changes for all ancestors?

- ○ None. Inserting a new node always changes the height in at least 1 ancestor node.
- ○ A new node is inserted as a child of a leaf node.
- ○ A new node is inserted as a child of an internal node with 1 child.
- ○ The new node is inserted as a child of an internal node with 2 children.

challenge activity

7.1.1: AVL tree properties.

Start



What is the height of 50? Ex: 5

50's left subtree height:

50's right subtree height:

1 2 3 4 5

Check Next

Exploring further:

- AVL is named for inventors Adelson-Velsky and Landis, who described the data structure in a 1962 paper: "An Algorithm for the Organization of Information", often cited as the first balanced tree structure. AVL tree: Wikipedia

7.2 AVL rotations

Tree rotation to keep balance

Inserting an item into an AVL tree may require rearranging the tree to maintain height balance. A rotation is a local rearrangement of a BST that maintains the BST ordering property while rebalancing the tree.

participation activity
7.2.1: A simple right rotation in an AVL tree.

Animation content:

undefined

Animation captions:

1. This BST violates the AVL height balance property.
2. A right rotation balances the tree while maintaining the BST ordering property.

Rotating is said to be done "at" a node. Ex: Above, the rotation is at node 86.

participation activity
7.2.2: AVL rotate right: 3 nodes.

Rotate right at node 89. Match the node value to the corresponding location in the rotated AVL tree template on the right.

- 89
- 32
- 17

---

n1
n2
n3
Reset

In the above animation, node 75 becomes the root, and node 86 becomes node 75's new right child. If node 75 had a right child node, that node becomes node 86's left child, to maintain the BST ordering property.

Below, the leaf nodes may instead be subtrees, and the rotation moves the entire subtree along with the node. Likewise, the shown tree may be a subtree, meaning the shown root may have a parent.

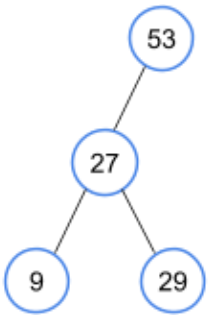
participation activity
7.2.3: In a right rotate, B's former right child C becomes D's left child, to maintain the BST ordering property.

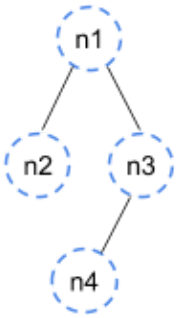
Animation captions:

1. If right-rotating B to the root, B's former right child becomes D's left child to maintain the BST ordering property.

participation activity
7.2.4: AVL rotate right: 4 nodes.

Rotate right at node 53. Match the node value to the node location in the rotated AVL tree template on the right.





- 53
- 29
- 27
- 9

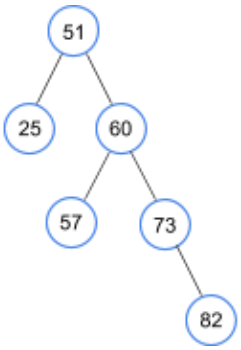
---

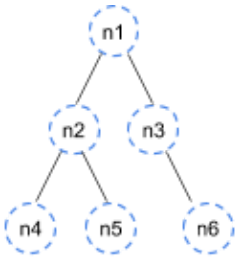
n1
n2
n3
n4
Reset

A left rotation is also possible and is symmetrical to the right rotation.

participation activity
7.2.5: AVL rotate left.

Rotate left at node 51. Match the node value to the node location in the AVL tree template.





- 51
- 25
- 73
- 82
- 60
- 57

---

n1
n2
n3
n4
n5
n6
Reset

Algorithms supporting AVL trees

The `AVLTreeUpdateHeight` algorithm updates a node's height value by taking the maximum of the child subtree heights and adding 1.

The `AVLTreeSetChild` algorithm sets a node as the parent's left or right child, updates the child's parent pointer, and updates the parent node's height.

The `AVLTreeReplaceChild` algorithm replaces one of a node's existing child pointers with a new value, utilizing `AVLTreeSetChild` to perform the replacement.

The `AVLTreeGetBalance` algorithm computes a node's balance factor by subtracting the right subtree height from the left subtree height.

Figure 7.2.1: AVLTreeUpdateHeight, AVLTreeSetChild, AVLTreeReplaceChild, and AVLTreeGetBalance algorithms.

```
AVLTreeUpdateHeight(node) {
   leftHeight = -1
   if (node→left != null)
      leftHeight = node→left→height
   rightHeight = -1
   if (node→right != null)
      rightHeight = node→right→height
   node→height = max(leftHeight, rightHeight) + 1
}


AVLTreeSetChild(parent, whichChild, child) {
   if (whichChild != "left" && whichChild != "right")
      return false

   if (whichChild == "left")
      parent→left = child
   else
      parent→right = child
   if (child != null)
      child→parent = parent

   AVLTreeUpdateHeight(parent)
   return true
}


AVLTreeReplaceChild(parent, currentChild, newChild) {
   if (parent→left == currentChild)
      return AVLTreeSetChild(parent, "left", newChild)
   else if (parent→right == currentChild)
      return AVLTreeSetChild(parent, "right", newChild)
   return false
}


AVLTreeGetBalance(node) {
   leftHeight = -1
   if (node→left != null)
      leftHeight = node→left→height
   rightHeight = -1
   if (node→right != null)
      rightHeight = node→right→height
```

```
   return leftHeight - rightHeight
}
```

participation activity
7.2.6: AVL tree utility algorithms.
1)
AVLTreeGetBalance has a precondition that the node parameter is non-null.

- ○ True
- ○ False

2)
AVLTreeGetBalance has a precondition that the node's children are both non-null.

- ○ True
- ○ False

3)
AVLTreeUpdateHeight has a precondition that the node's children both have correct height values.

- ○ True
- ○ False

4)
AVLTreeSetChild has a precondition that the child's height value is correct.

- ○ True
- ○ False

5)
AVLTreeSetChild calls AVLTreeReplaceChild.

- ○ True
- ○ False

Right rotation algorithm

A right rotation algorithm is defined on a subtree root (node D) which must have a left child (node B). The algorithm reassigns child pointers,assigning B's right child with D, and assigning D's left child with C (B's original right child, which may be null). If D's parent is non-null, then the parent's child D is replaced with B. Other tree parts (T1..T4 below) naturally stay with their parent nodes.

participation activity
7.2.7: Right rotation algorithm.

Animation content:

```
undefined
```

Animation captions:

1. A right rotation at node D changes P's child from D to B, D's left child from B to C, and B's right child from C to D.

participation activity
7.2.8: Right rotation algorithm.

Refer to the above AVL tree right rotation algorithm.

1)

The algorithm works even if node B's right child is null.

- ○ True
- ○ False

2)
The algorithm works even if node D's left child is null.

- ○ True
- ○ False

3)
Node D may be a subtree of a larger tree. The algorithm updates node D's parent to point to node B, the new root of the subtree.

- ○ True
- ○ False

AVL tree balancing

When an AVL tree node has a balance factor of 2 or -2, which only occurs after an insertion or removal, the node must be rebalanced via rotations. The `AVLTreeRebalance` algorithm updates the height value at a node, computes the balance factor, and rotates if the balance factor is 2 or -2.

Figure 7.2.2: AVLTreeRebalance algorithm.

```
AVLTreeRebalance(tree, node) {
   AVLTreeUpdateHeight(node)
   if (AVLTreeGetBalance(node) == -2) {
      if (AVLTreeGetBalance(node→right) == 1) {
         // Double rotation case.
         AVLTreeRotateRight(tree, node→right)
      }
      return AVLTreeRotateLeft(tree, node)
   }
   else if (AVLTreeGetBalance(node) == 2) {
      if (AVLTreeGetBalance(node→left) == -1) {
         // Double rotation case.
         AVLTreeRotateLeft(tree, node→left)
      }
      return AVLTreeRotateRight(tree, node)
   }
   return node
}
```

participation activity
7.2.9: AVLTreeRebalance algorithm.
1)
AVLTreeRebalance rebalances all ancestors from the node up to the root.

- ○ True
- ○ False

2)
AVLTreeRebalance recomputes the height values for each non-null child.

- ○ True
- ○ False

3)

AVLTreeRebalance recomputes the height for the node.

- ○ True
- ○ False

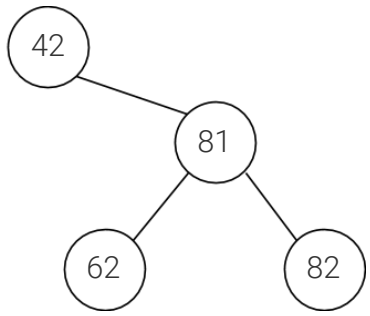
4)
AVLTreeRebalance takes no action if a node's balance factor is 1, 0, or -1.

- ○ True
- ○ False

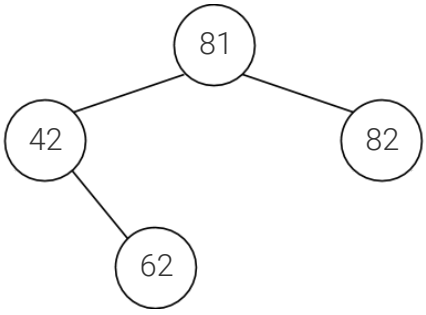
challenge activity
7.2.1: AVL rotations.

Start

Given the following BST violating the AVL height balance property:



A [ ] rotation at node [ ] yields:



1 2 3

Check Next

7.3 AVL insertions

Insertions requiring rotations to rebalance

Inserting an item into an AVL tree may cause the tree to become unbalanced. A rotation can rebalance the tree.

participation activity
7.3.1: After an insert, a rotation may rebalance the tree.

Animation captions:

1. Inserting a node may temporarily violate the AVL height balance property.
2. A rotation, at the problem node closest to the new node, restores balance.

Sometimes, the imbalance is due to an insertion on the *inside* of a subtree, rather than on the *outside* as above. One rotation won't rebalance. A double rotation is needed.

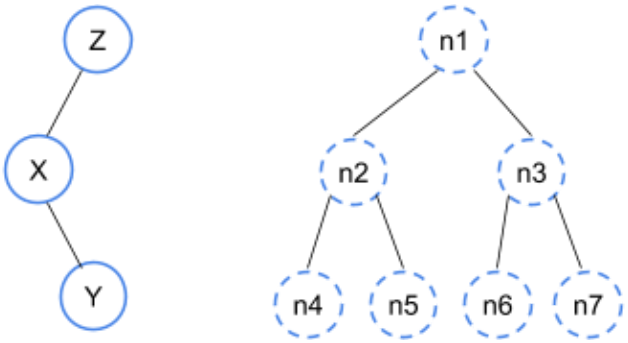
participation activity
7.3.2: Sometimes a double rotation is necessary to rebalance.

Animation captions:

1. Inserting a node may temporarily violate the AVL height balance property.
2. In this case, after a single rotate, the tree is still unbalanced.
3. A double "left-then-right" rotate is necessary. First rotate left at A...
4. ...and then rotate right at C. Tree is now balanced.

participation activity
7.3.3: Double rotate: Left-then-right.

When performing a double rotation (left-then-right), indicate each node's new location using the template tree's labels (n1, n2, n3, n4, n5, n6, or n7).



1)
After the initial left rotation, where is Y?

Check Show answer

2)
After the initial left rotation, where is X?

Check Show answer

3)
After the left rotation, where is the right rotation performed: at X, Y, or Z?

Check Show answer

4)
After the left rotation and then the right rotation, where is Z?

Check Show answer

5)
After the left rotation and then the right rotation, where is Y?

Check Show answer

6)
After the left rotation and then the right rotation, where is X?

Check Show answer

7)
If the left rotation had NOT first been performed, a right rotation at Z would have made X the new root, and made Z X's right child. To where would Y have been moved?

Check Show answer

Four imbalance cases

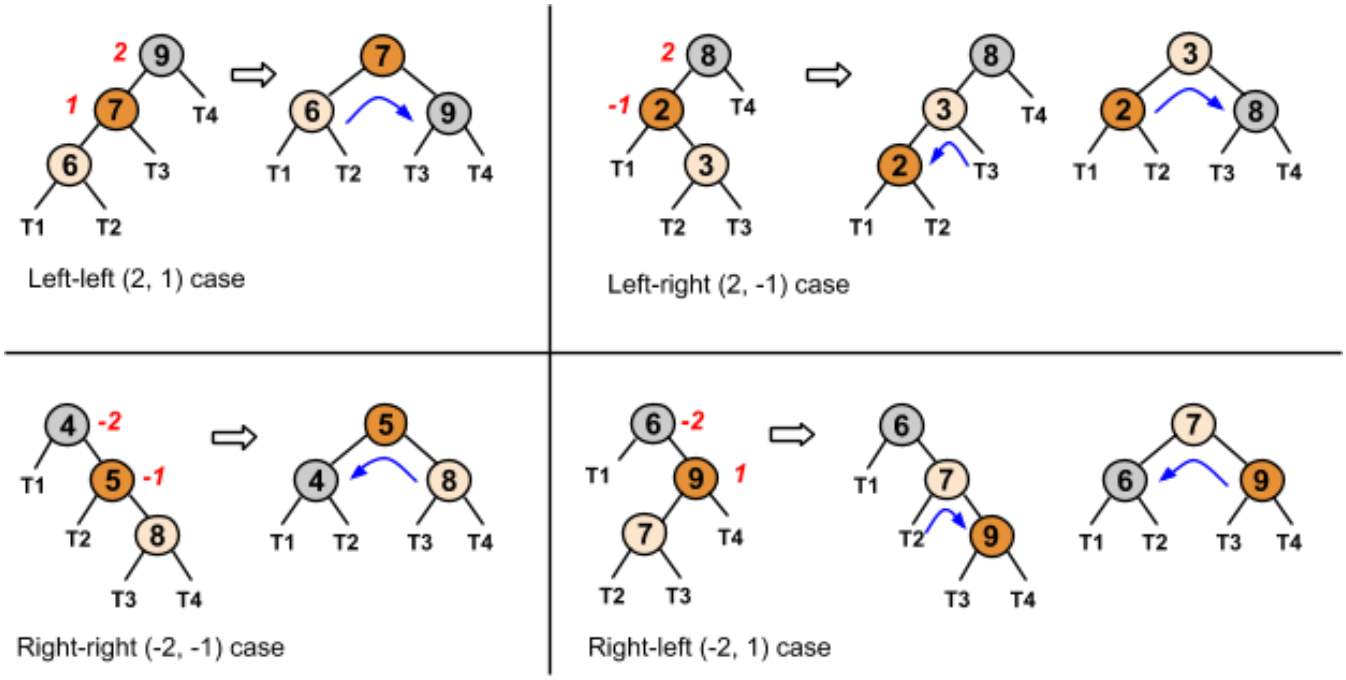
After inserting a node, nodes on the path from the new node to the root should be checked for a balance factor of 2 or -2. The first such node P triggers rebalancing. Four cases exist, distinguishable by the balance factor of node P and one of P's children.

participation activity
7.3.4: Four AVL imbalance cases are possible after inserting a new node.

Animation captions:

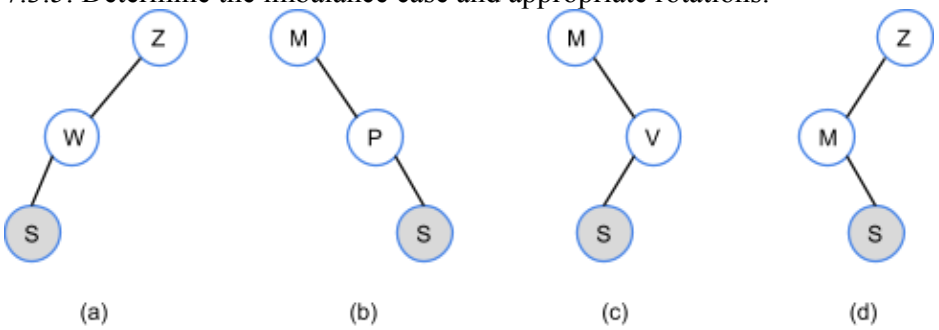
1. Four imbalance cases can arise from inserting a new node into an AVL tree. Inserting node 12 leads to a left-left imbalance case.
2. Inserting node 38 to the original AVL tree results in a left-right imbalance case.
3. Similarly, right-right and right-left imbalance cases also exist.

Figure 7.3.1: Four imbalance cases and rotations (indicated by blue arrow) to rebalance.



participation activity

7.3.5: Determine the imbalance case and appropriate rotations.



1)

(a)

- ○ (2, 1)
- ○ (2, -1)
- ○ (-2, -1)
- ○ (-2, 1)

2)

(b)

- ○ (2, 1)
- ○ (2, -1)
- ○ (-2, -1)
- ○ (-2, 1)

3)

(c)

- ○ (2, 1)
- ○ (2, -1)
- ○ (-2, -1)
- ○ (-2, 1)

4)

(d)

- ○ (2, 1)

- ○ (2, -1)
- ○ (-2, -1)
- ○ (-2, 1)

5)
What is the proper rotation for (a)?

- ○ Left at Z
- ○ Right at Z
- ○ Right at W, then right at Z.

6)
What is the proper rotation for a 2, -1 case?

- ○ Right
- ○ Right-left
- ○ Left-right

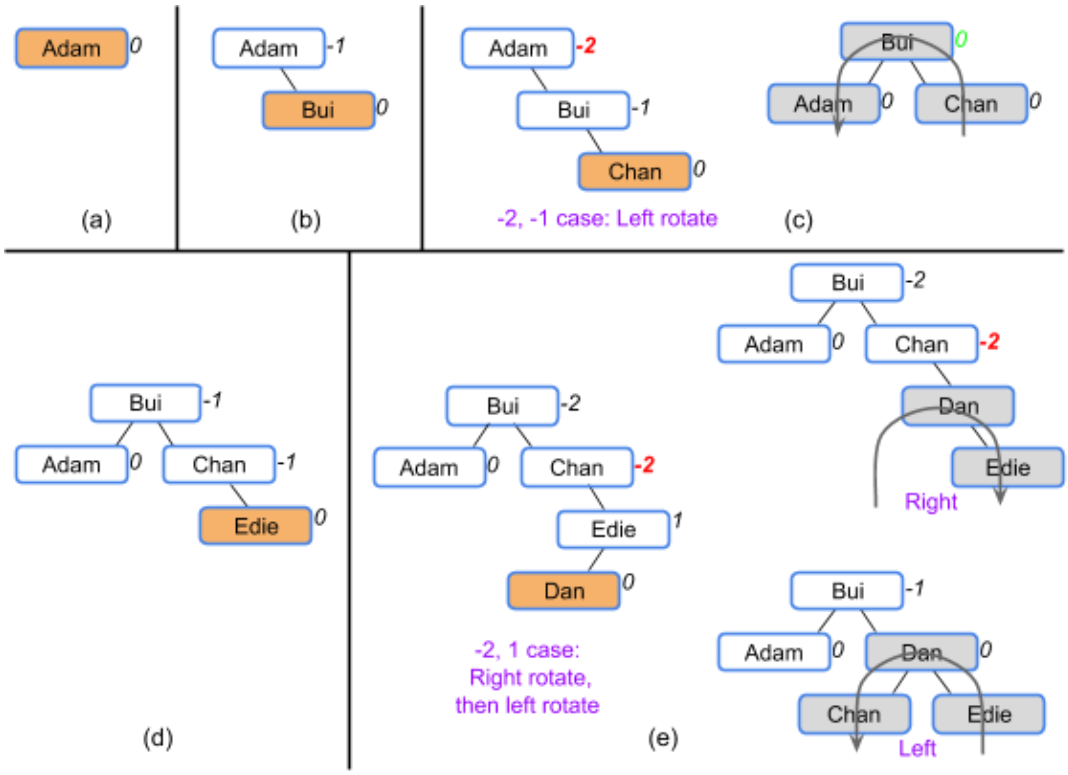
Insertion with rebalancing

An AVL tree insertion involves searching for the insert location, inserting the new node, updating balance factors, and rebalancing.

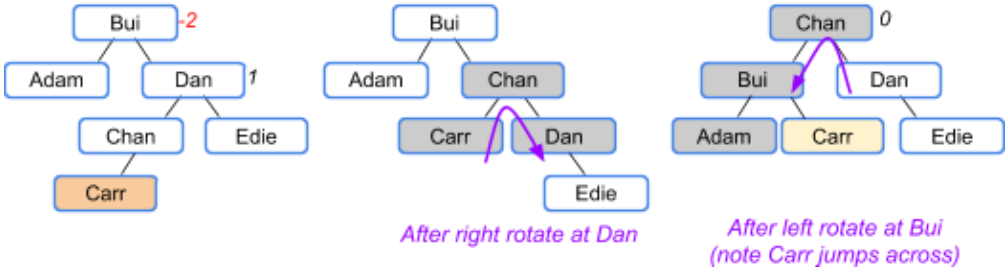
Balance factor updates are only needed on nodes ascending along the path from the inserted node up to the root, since no other nodes' balance could be affected. Each node's balance factor can be recomputed by determining left and right subtree heights, or for speed can be stored in each node and then incrementally updated: +1 if ascending from a left child, -1 if from a right child. If a balance factor update yields 2 or -2, the imbalance case is determined via that node's left (for 2) or right (for -2) child's balance factor, and the appropriate rotations performed.

Example 7.3.1: AVL example: Phone book.

A phone book program stores names in an AVL tree. A program user enters a series of names, which just happen to be in sorted order (perhaps copying from a previously sorted list). The tree is kept balanced by occasional rebalancing rotations, thus enabling fast searches. Without rebalancing, the BST's final height would be 4 instead of just 2.



Adding another item shows the interesting case of rotations occurring higher up in the tree.



participation activity
7.3.6: AVL balance.

1)
If an AVL tree has X levels, the first X-1 levels will be full.

- ○ True
- ○ False

2)
For n nodes, an AVL tree has height equal to floor(log(n)).

- ○ True
- ○ False

3)
For n nodes, an AVL tree has height O(log(n)).

- ○ True
- ○ False

4)
An AVL insert operation involves a search, an insert, and possibly some rotations. An insert operation is thus O(log(n)).

- ○ True
- ○ False

5)
After inserting a node into a tree, all tree nodes must have their balance factors updated.

- ○ True
- ○ False

6)
Conceivably, inserting 100 items into an AVL tree may not require *any* rotations.

- ○ True
- ○ False

AVL insertion algorithm

Insertion starts with the standard BST insertion algorithm. After inserting a node, all ancestors of the inserted node, from the parent up to the root, are rebalanced. A node is rebalanced by first computing the node's balance factor, then performing rotations if the balance factor is outside of the range [-1,1].

Figure 7.3.2: AVLTreeInsert algorithm.

```
AVLTreeInsert(tree, node) {
   if (tree→root == null) {
      tree→root = node
      node→parent = null
      return
```

```
    }

    cur = tree→root
    while (cur != null) {
       if (node→key < cur→key) {
          if (cur→left == null) {
             cur→left = node
             node→parent = cur
             cur = null
          }
          else {
             cur = cur→left
          }
       }
       else {
          if (cur→right == null) {
             cur→right = node
             node→parent = cur
             cur = null
          }
          else
             cur = cur→right
       }
    }

    node = node→parent
    while (node != null) {
       AVLTreeRebalance(tree, node)
       node = node→parent
    }
}
```

participation activity
7.3.7: AVLTreeInsert algorithm.

1)
AVLTreeInsert updates heights on all ancestors before inserting the node.

- ○ True
- ○ False

2)
The node passed to AVLTreeInsert must be a leaf node.

- ○ True
- ○ False

3)
AVLTreeInsert works to insert a node into an empty tree.

- ○ True
- ○ False

4)
AVLTreeInsert adds the new node as a child to an existing node in the tree, but the new node's parent pointer is not set and must be handled outside of the function.

- ○ True
- ○ False

5)

AVLTreeInsert sets the height in the newly inserted node to 0 and the node's left and right child pointers to null.

- ○ True
- ○ False

AVL insertion algorithm complexity

The AVL insertion algorithm traverses the tree from the root to a leaf node to find the insertion point, then traverses back up to the root to rebalance. One node is visited per level, and at most 2 rotations are needed for a single node. Each rotation is an O(1) operation. Therefore, the runtime complexity of insertion is O(log N).

Because a fixed number of temporary pointers are needed for the AVL insertion algorithm, including any rotations, the space complexity is O(1).

Exploring further:

- AVL tree simulator

7.4 AVL removals

Removing nodes in AVL trees

Given a key, an AVL tree remove operation removes the first-found matching node, restructuring the tree to preserve all AVL tree requirements. Removal begins by removing the node using the standard BST removal algorithm. After removing a node, all ancestors of the removed node, from the nodes' parent up to the root, are rebalanced. A node is rebalanced by first computing the node's balance factor, then performing rotations if the balance factor is 2 or -2.

participation activity
7.4.1: AVL tree removal.

Animation captions:

1. Removing node 63 begins with the standard BST removal. A pointer to node 63's parent is kept.
2. After removal, the balance factor of each node from the parent up to the root is checked. Rotations are used if the balance factor (BF) is 2 or -2.
3. Removing node 84 replaces the node with node 84's successor, node 89. Node 89 is then removed from the right subtree. No rotations are necessary because the root's balance factor changes to 1.
4. After the standard BST removal algorithm removes node 93, the root is left with a balance factor of 2. A right rotation at 21 rebalances the tree.

participation activity
7.4.2: AVL tree removal.
1)
The BST removal algorithm is used as part of AVL tree removal.

- ○ True
- ○ False

2)
After removing a node from an AVL tree using the standard BST removal algorithm, all nodes in the tree must be rebalanced.

- ○ True
- ○ False

3)
During rebalancing, encountering nodes with balance factors of 2 or -2 implies that a rotation must occur.

- ○ True

○ False

4)
Removal of an internal node with 2 children always requires a rotation to rebalance.

○ True
○ False

AVL tree removal algorithm

To remove a key, the AVL tree removal algorithm first locates the node containing the key using `BSTSearch`. If the node is found, AVLTreeRemoveNode is called to remove the node. Standard BST removal logic is used to remove the node from the tree. Then AVLTreeRebalance is called for all ancestors of the removed node, from the parent up to the root.

participation activity
7.4.3: AVL tree removal algorithm.

Animation content:

`undefined`

Animation captions:

1. Removal of 75 starts with the standard BST removal, replacing the child and updating the height at node 50.
2. The node has not changed, so the balancing begins at the parent and continues up parent pointers until null.
3. Removal of 88 again starts with the standard BST removal. The root's balance factor changes to 2, requiring a rotation to rebalance.
4. After removals, the tree maintains O(log n) height.

Figure 7.4.1: AVLTreeRebalance algorithm.

```
AVLTreeRebalance(tree, node) {
   AVLTreeUpdateHeight(node)
   if (AVLTreeGetBalance(node) == -2) {
      if (AVLTreeGetBalance(node⇢right) == 1) {
         // Double rotation case.
         AVLTreeRotateRight(tree, node⇢right)
      }
      return AVLTreeRotateLeft(tree, node)
   }
   else if (AVLTreeGetBalance(node) == 2) {
      if (AVLTreeGetBalance(node⇢left) == -1) {
         // Double rotation case.
         AVLTreeRotateLeft(tree, node⇢left)
      }
      return AVLTreeRotateRight(tree, node)
   }
   return node
}
```

Figure 7.4.2: AVLTreeRemoveKey algorithm.

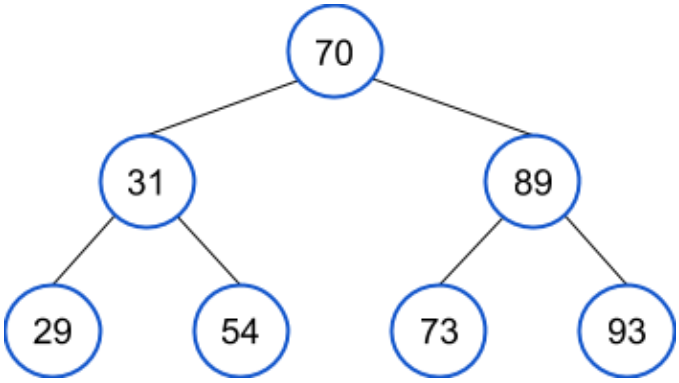
```
AVLTreeRemoveKey(tree, key) {
   node = BSTSearch(tree, key)
   return AVLTreeRemoveNode(tree, node)
}
```

Figure 7.4.3: AVLTreeRemoveNode algorithm.

```
AVLTreeRemoveNode(tree, node) {
   if (node == null)
      return false

   // Parent needed for rebalancing
   parent = node⇢parent

   // Case 1: Internal node with 2 children
   if (node⇢left != null && node⇢right != null) {
      // Find successor
      succNode = node⇢right
      while (succNode⇢left != null)
         succNode = succNode⇢left

      // Copy the value from the node
      node = Copy succNode

      // Recursively remove successor
      AVLTreeRemoveNode(tree, succNode)

      // Nothing left to do since the recursive call will have rebalanced
      return true
   }

   // Case 2: Root node (with 1 or 0 children)
   else if (node == tree⇢root) {
      if (node⇢left != null)
         tree⇢root = node⇢left
      else
         tree⇢root = node⇢right

      if (tree⇢root)
         tree⇢root⇢parent = null

      return true
   }

   // Case 3: Internal with left child only
   else if (node⇢left != null)
      AVLTreeReplaceChild(parent, node, node⇢left)

   // Case 4: Internal with right child only OR leaf
   else
      AVLTreeReplaceChild(parent, node, node⇢right)

   // node is gone. Anything that was below node that has persisted is already correctly
   // balanced, but ancestors of node may need rebalancing.
   node = parent
   while (node != null) {
      AVLTreeRebalance(tree, node)
      node = node⇢parent
   }
   return true
}
```

participation activity
7.4.4: AVL tree removal algorithm.

Select the order of tree-altering operations that occur as a result of calling AVLTreeRemoveKey to remove 70 from this tree:



- 6
- 3
- 1
- Never
- 4
- 5
- 2

Node 89's left child is set to null.
The tree's root pointer is set to the root's left child.
The node being removed is compared against the tree's root and is not equal.
Node 73 is found as the successor to node 70.
AVLTreeRebalance is called on node 73, which has a balance factor of 0, so no rotations are necessary.
Key 73 is copied into the root node.
AVLTreeRebalance is called on node 89, which has a balance factor of -1, so no rotations are necessary.
Reset
AVL removal algorithm complexity

In the worst case scenario, the AVL removal algorithm traverses the tree from the root to the lowest level to find the node to remove, then traverses back up to the root to rebalance. One node is visited per level, and at most 2 rotations are needed for a single node. Each rotation is an O(1) operation. Therefore, the runtime complexity of an AVL tree removal is O(log N).

Because a fixed number of temporary pointers are needed for the AVL removal algorithm, including any rotations, the space complexity is O(1).

7.5 Red-black tree: A balanced tree

A red-black tree is a BST with two node types, namely red and black, and supporting operations that ensure the tree is balanced when a node is inserted or removed. The below red-black tree's requirements ensure that a tree with N nodes will have a height of O(log N).

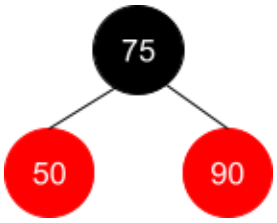
- Every node is colored either red or black.
- The root node is black.
- A red node's children cannot be red.
- A null child is considered to be a black leaf node.
- All paths from a node to any null leaf descendant node must have the same number of black nodes.

participation activity
7.5.1: Red-black tree rules.

Animation captions:

1. The null child pointer of a leaf node is considered a null leaf node and is always black. Visualizing null leaf nodes helps determine if a tree is a valid red-black tree.
2. Each requirement must be met for the tree to be a valid red-black tree.
3. A tree that violates any requirement is not a valid red-black tree.

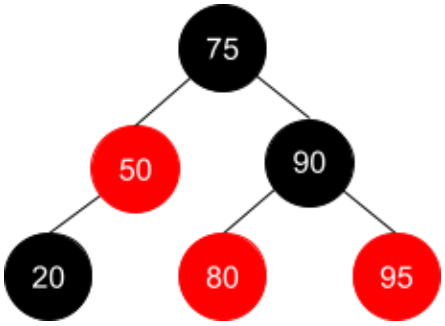
participation activity
7.5.2: Red-black tree rules.
1)



Which red-black tree requirement does this BST not satisfy?

- ○ Root node must be black.
- ○ A red node's children cannot be red.
- ○ All paths from a node to null leaf nodes must have the same number of black nodes.
- ○ None.

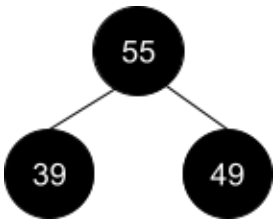
2)



Which red-black tree requirement does this BST not satisfy?

- ○ Root node must be black.
- ○ A red node's children cannot be red.
- ○ Not all levels are full.
- ○ All paths from a node to null leaf nodes must have the same number of black nodes.

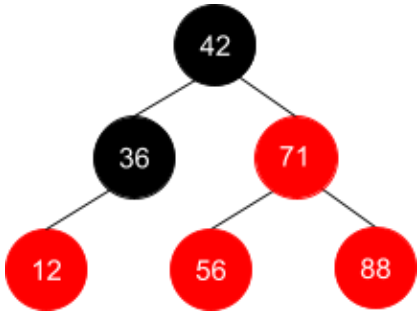
3)



The tree below is a valid red-black tree.

- ○ True
- ○ False

4)



What single color change will make the below tree a valid red-black tree?

- ○ Change node 36's color to red.
- ○ Change node 71's color to black.
- ○ Change node 88's color to black.
- ○ No single color change will make this a valid red-black tree..

5)
A black node's children will always be the same color.

- ○ True
- ○ False

6)
All valid red-black trees will have more red nodes than black nodes.

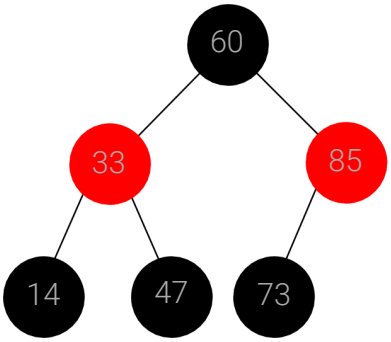
- ○ True
- ○ False

7)
Any BST can be made a red-black tree by coloring all nodes black.

- ○ True
- ○ False

challenge activity
7.5.1: Red-black tree: A balanced tree.

Start



Is the above a valid red-black tree? Select all that apply.

- ☐ Yes
- ☐ No, root node must be black
- ☐ No, a red node's child cannot be red
- ☐ No, all paths from a node to null leaf nodes must have the same number of black nodes
- ☐ No, BST ordering property violated

Check Next

7.6 Red-black tree: Rotations

Introduction to rotations

A rotation is a local rearrangement of a BST that maintains the BST ordering property while rebalancing the tree. Rotations are used during the insert and remove operations on a red-black tree to ensure that red-black tree requirements hold. Rotating is said to be done "at" a node. A left rotation at a node causes the node's right child to take the node's place in the tree. A right rotation at a node causes the node's left child to take the node's place in the tree.

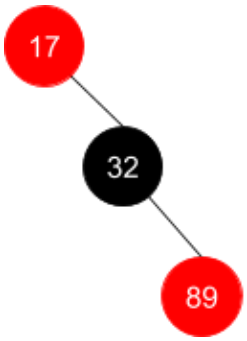
participation activity
7.6.1: A simple left rotation in a red-black tree.

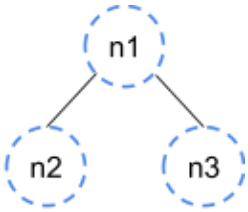
Animation captions:

1. This BST is not a valid red-black tree. From the root, paths down to null leaves are inconsistent in terms of number of black nodes.
2. A left rotation at node 16 creates a valid red-black tree.

participation activity
7.6.2: Red-black tree rotate left: 3 nodes.

Rotate left at node 17. Match the node value to the corresponding location in the rotated red-black tree template on the right.





- 89
- 17
- 32

---

n2
n1
n3
Reset

Left rotation algorithm

A rotation requires altering up to 3 child subtree pointers. A left rotation at a node requires the node's right child to be non-null. Two utility functions are used for red-black tree rotations. The `RBTreeSetChild` utility function sets a node's left child, if the whichChild parameter is "left", or right child, if the whichChild parameter is "right", and updates the child's parent pointer. The `RBTreeReplaceChild` utility function replaces a node's left or right child pointer with a new value.

Figure 7.6.1: RBTreeSetChild utility function.

```
RBTreeSetChild(parent, whichChild, child) {
   if (whichChild != "left" && whichChild != "right")
      return false

   if (whichChild == "left")
      parent→left = child
   else
      parent→right = child
   if (child != null)
      child→parent = parent
   return true
}
```

Figure 7.6.2: RBTreeReplaceChild utility function.

```
RBTreeReplaceChild(parent, currentChild, newChild) {
   if (parent→left == currentChild)
      return RBTreeSetChild(parent, "left", newChild)
   else if (parent→right == currentChild)
      return RBTreeSetChild(parent, "right", newChild)
   return false
}
```

The `RBTreeRotateLeft` function performs a left rotation at the specified node by updating the right child's left child to point to the node, and updating the node's right child to point to the right child's former left child. If non-null, the node's parent has the child pointer changed from node to the node's right child. Otherwise, if the node's parent is null, then the tree's root pointer is updated to point to the node's right child.

Figure 7.6.3: RBTreeRotateLeft pseudocode.

```
RBTreeRotateLeft(tree, node) {
   rightLeftChild = node→right→left
   if (node→parent != null)
      RBTreeReplaceChild(node→parent, node, node→right)
   else { // node is root
      tree→root = node→right
      tree→root→parent = null
   }
   RBTreeSetChild(node→right, "left", node)
   RBTreeSetChild(node, "right", rightLeftChild)
}
```

participation activity
7.6.3: RBTreeRotateLeft algorithm.

- Node with null right child
- Node with null left child
- Red node
- Root node

RBTreeRotateLeft will not work when called at this type of node.
RBTreeRotateLeft called at this node requires the tree's root pointer to be updated.
After calling RBTreeRotateLeft at this node, the node will have a null left child.
After calling RBTreeRotateLeft at this node, the node will be colored red.

Reset

Right rotation algorithm

Right rotation is analogous to left rotation. A right rotation at a node requires the node's left child to be non-null.

participation activity
7.6.4: RBTreeRotateRight algorithm.

Animation content:

undefined

Animation captions:

1. A right rotation at node 80 causes node 61 to become the new root, and nodes 40 and 80 to become the root's left and right children, respectively.
2. The rotation results in a valid red-black tree.

participation activity
7.6.5: Right rotation algorithm.

1)
A rotation will never change the root node's value.

- ○ True
- ○ False

2)
A rotation at a node will only change properties of the node's descendants, but will never change properties of the node's ancestors.

- ○ True
- ○ False

3)
RBTreeRotateRight works even if the node's parent is null.

- ○ True
- ○ False

4)
RBTreeRotateRight works even if the node's left child is null.

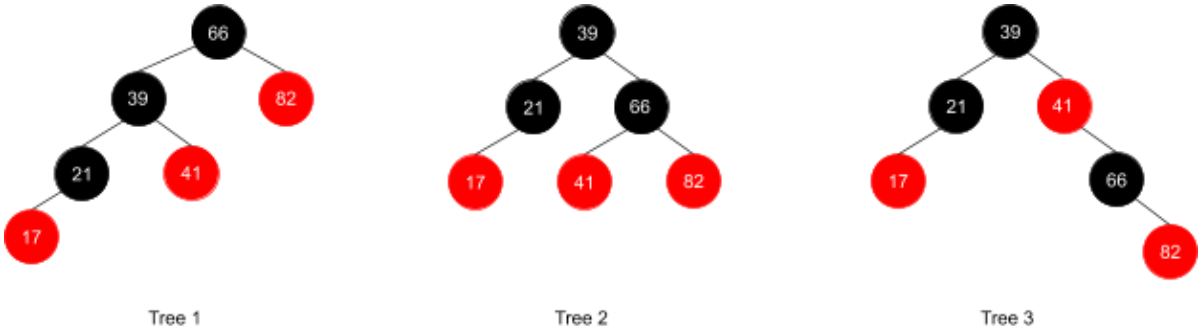
- ○ True
- ○ False

5)
RBTreeRotateRight works even if the node's right child is null.

- ○ True
- ○ False

participation activity
7.6.6: Red-black tree rotations.

Consider the 3 trees below:



1)
Which trees are valid red-black trees?

- ○ Tree 1 only.
- ○ Tree 2 only.
- ○ Tree 3 only.
- ○ All are valid red-black trees.

2)
Which operation on tree 1 would produce tree 2?

- ○ Rotate right at node 82.
- ○ Rotate left at node 66.
- ○ Rotate right at node 66.
- ○ Rotate left at node 39.

3)
Which operation on tree 3 would produce tree 2?

- ○ Rotate left at node 21.
- ○ Rotate left at node 39.
- ○ Rotate left at node 41.
- ○ Rotate left at node 66.

4)
A right rotation at node 21 in tree 2 would result in a valid red-black tree.

- ○ True
- ○ False

7.7 Red-black tree: Insertion

Given a new node, a red-black tree insert operation inserts the new node in the proper location such that all red-black tree requirements still hold after the insertion completes.

Red-black tree insertion begins by calling `BSTInsert` to insert the node using the BST insertion rules. The newly inserted node is colored red and then a balance operation is performed on this node.

Figure 7.7.1: RBTreeInsert algorithm.

```
RBTreeInsert(tree, node) {
   BSTInsert(tree, node)
   node→color = red
   RBTreeBalance(tree, node)
}
```

The red-black balance operation consists of the steps below.

1. Assign `parent` with `node`'s parent, `uncle` with `node`'s uncle, which is a sibling of `parent`, and `grandparent` with `node`'s grandparent.
2. If `node` is the tree's root, then color `node` black and return.
3. If `parent` is black, then return without any alterations.
4. If `parent` and `uncle` are both red, then color `parent` and `uncle` black, color `grandparent` red, recursively balance `grandparent`, then return.
5. If `node` is `parent`'s right child and `parent` is `grandparent`'s left child, then rotate left at `parent`, assign `node` with `parent`, assign `parent` with `node`'s parent, and go to step 7.
6. If `node` is `parent`'s left child and `parent` is `grandparent`'s right child, then rotate right at `parent`, assign `node` with `parent`, assign `parent` with `node`'s parent, and go to step 7.
7. Color `parent` black and `grandparent` red.
8. If `node` is `parent`'s left child, then rotate right at `grandparent`, otherwise rotate left at `grandparent`.

The RBTreeBalance function uses the RBTreeGetGrandparent and RBTreeGetUncle utility functions to determine a node's grandparent and uncle, respectively.

Figure 7.7.2: RBTreeGetGrandparent and RBTreeGetUncle utility functions.

```
RBTreeGetGrandparent(node) {
   if (node→parent == null)
      return null
   return node→parent→parent
}

RBTreeGetUncle(node) {
   grandparent = null
   if (node→parent != null)
      grandparent = node→parent→parent
   if (grandparent == null)
      return null
   if (grandparent→left == node→parent)
      return grandparent→right
   else
      return grandparent→left
}
```

participation activity
7.7.1: RBTreeBalance algorithm.

Animation content:

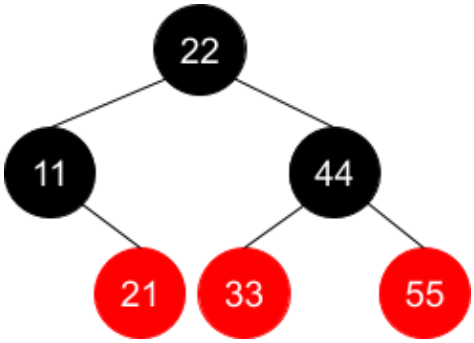
undefined

Animation captions:

1. Insertion of 22 as the root starts with the normal BST insertion, followed by coloring the node red. The balance operation simply changes the root node to black.
2. Insertion of 11 and 33 do not require any node color changes or rotations.
3. Insertion of 55 requires recoloring the parent, uncle, and grandparent, then recursively balancing the grandparent.
4. Inserting 44 requires two rotations. The first rotation is a right rotation at the parent, node 55. The second rotation is a left rotation at the grandparent, node 33.

participation activity
7.7.2: Red-black tree: insertion.

Consider the following tree:



1)
Starting at and including the root node, how many black nodes are encountered on any path down to and including the null leaf nodes?

- ○ 1
- ○ 2
- ○ 3
- ○ 4

2)
Insertion of which value will require at least 1 rotation?

- ○ 10
- ○ 20
- ○ 30
- ○ 45

3)
The values 11, 21, 22, 33, 44, 55 can be inserted in any order and the above tree will always be the result.

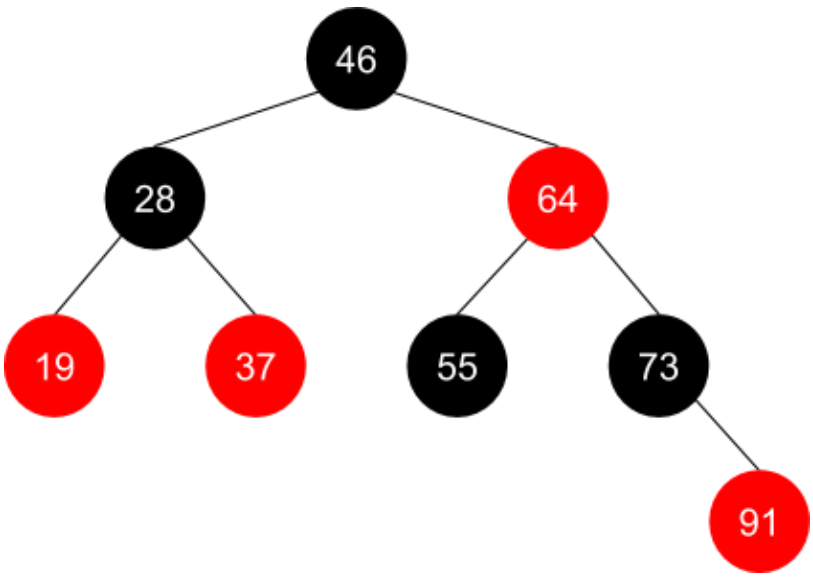
- ○ True
- ○ False

4)
All red nodes could be recolored to black and the above tree would still be a valid red-black tree.

- ○ True
- ○ False

participation activity
7.7.3: RBTreeInsert algorithm.

Select the order of tree-altering operations that occur as a result of calling RBTreeInsert to insert 82 into this tree:



- 4
- 1
- Never
- 2
- 3
- 5

Rotate left at node 73.
Insert red node 82 as node 91's left child.
Color grandparent node red.
Color parent node black.
Rotate right at node 91.
Call RBTreeBalance recursively on node 73.
Reset

7.8 Red-black tree: Removal

Removal overview

Given a key, a red-black tree remove operation removes the first-found matching node, restructuring the tree to preserve all red-black tree requirements. First, the node to remove is found using `BSTSearch`. If the node is found, RBTreeRemoveNode is called to remove the node.

Figure 7.8.1: RBTreeRemove algorithm.

```
RBTreeRemove(tree, key) {
   node = BSTSearch(tree, key)
   if (node != null)
      RBTreeRemoveNode(tree, node)
}
```

The RBTreeRemove algorithm consists of the following steps:

1. If the node has 2 children, copy the node's predecessor to a temporary value, recursively remove the predecessor from the tree, replace the node's key with the temporary value, and return.
2. If the node is black, call `RBTreePrepareForRemoval` to restructure the tree in preparation for the node's removal.
3. Remove the node using the standard BST `BSTRemove` algorithm.

Figure 7.8.2: RBTreeRemoveNode algorithm.

```
RBTreeRemoveNode(tree, node) {
   if (node→left != null && node→right != null) {
      predecessorNode = RBTreeGetPredecessor(node)
      predecessorKey = predecessorNode→key
      RBTreeRemoveNode(tree, predecessorNode)
      node→key = predecessorKey
      return
   }

   if (node→color == black)
      RBTreePrepareForRemoval(node)
   BSTRemove(tree, node→key)
}
```

Figure 7.8.3: RBTreeGetPredecessor utility function.

```
RBTreeGetPredecessor(node) {
   node = node⇢left
   while (node⇢right != null) {
      node = node⇢right
   }
   return node
}
```

participation activity
7.8.1: Removal concepts.

1)
The red-black tree removal algorithm uses the normal BST removal algorithm.

- ○ True
- ○ False

2)
RBTreeRemove uses the BST search algorithm.

- ○ True
- ○ False

3)
Removing a red node with RBTreeRemoveNode will never cause RBTreePrepareForRemoval to be called.

- ○ True
- ○ False

4)
Although RBTreeRemoveNode uses the node's predecessor, the algorithm could also use the successor.

- ○ True
- ○ False

Removal utility functions

Utility functions help simplify red-black tree removal code. The `RBTreeGetSibling` function returns the sibling of a node. The `RBTreeIsNonNullAndRed` function returns true only if a node is non-null and red, false otherwise. The `RBTreeIsNullOrBlack` function returns true if a node is null or black, false otherwise. The `RBTreeAreBothChildrenBlack` function returns true only if both of a node's children are black. Each utility function considers a null node to be a black node.

Figure 7.8.4: RBTreeGetSibling algorithm.

```
RBTreeGetSibling(node) {
   if (node⇢parent != null) {
      if (node == node⇢parent⇢left)
         return node⇢parent⇢right
      return node⇢parent⇢left
   }
   return null
}
```

Figure 7.8.5: RBTreeIsNonNullAndRed algorithm.

```
RBTreeIsNonNullAndRed(node) {
   if (node == null)
      return false
```

```
   return (node→color == red)
}
```

Figure 7.8.6: RBTreeIsNullOrBlack algorithm.

```
RBTreeIsNullOrBlack(node) {
   if (node == null)
      return true
   return (node→color == black)
}
```

Figure 7.8.7: RBTreeAreBothChildrenBlack algorithm.

```
RBTreeAreBothChildrenBlack(node) {
   if (node→left != null && node→left→color == red)
      return false
   if (node→right != null && node→right→color == red)
      return false
   return true
}
```

participation activity
7.8.2: Removal utility functions.

1)
Under what circumstance will RBTreeAreBothChildrenBlack always return true?

- ○ When both of the node's children are null
- ○ When both of the node's children are non-null
- ○ When the node's left child is null
- ○ When the node's right child is null

2)
RBTreeIsNonNullAndRed will not work properly when passed a null node.

- ○ True
- ○ False

3)
What will be returned when RBTreeGetSibling is called on a node with a null parent?

- ○ A pointer to the node
- ○ null
- ○ A pointer to the tree's root
- ○ Undefined/unknown

4)
RBTreeIsNullOrBlack requires the node to be a leaf.

- ○ True
- ○ False

5)
Which function(s) have a precondition that the node parameter must be non-null?

- ○ All 4 functions have a precondition that the node parameter must be non-null
- ○ RBTreeGetSibling only
- ○ RBTreeIsNonNullAndRed and RBTreeIsNullOrBlack
- ○ RBTreeGetSibling and RBTreeAreBothChildrenBlack

6)
If RBTreeGetSibling returns a non-null, red node, then the node's parent must be non-null and black.

○ True
○ False

Prepare-for-removal algorithm overview

Preparation for removing a black node requires altering the number of black nodes along paths to preserve the black-path-length property. The `RBTreePrepareForRemoval` algorithm uses 6 utility functions that analyze the tree and make appropriate alterations when each of the 6 cases is encountered. The utility functions return true if the case is encountered, and false otherwise. If case 1, 3, or 4 is encountered, `RBTreePrepareForRemoval` will return after calling the utility function. If case 2, 5, or 6 is encountered, additional cases must be checked.

Figure 7.8.8: RBTreePrepareForRemoval pseudocode.

```
RBTreePrepareForRemoval(tree, node) {
   if (RBTreeTryCase1(tree,node))
      return

   sibling = RBTreeGetSibling(node)
   if (RBTreeTryCase2(tree, node, sibling))
      sibling = RBTreeGetSibling(node)
   if (RBTreeTryCase3(tree, node, sibling))
      return
   if (RBTreeTryCase4(tree, node, sibling))
      return
   if (RBTreeTryCase5(tree, node, sibling))
      sibling = RBTreeGetSibling(node)
   if (RBTreeTryCase6(tree, node, sibling))
      sibling = RBTreeGetSibling(node)

   sibling⇢color = node⇢parent⇢color
   node⇢parent⇢color = black
   if (node == node⇢parent⇢left) {
      sibling⇢right⇢color = black
      RBTreeRotateLeft(tree, node⇢parent)
   }
   else {
      sibling⇢left⇢color = black
      RBTreeRotateRight(tree, node⇢parent)
   }
}
```

participation activity
7.8.3: Prepare-for-removal algorithm.
1)
If the condition for any of the first 6 cases is met, then an adjustment specific to the case is made and the algorithm returns without processing any additional cases.

○ True
○ False

2)
Why is no preparation action required if the node is red?

○ A red node will never have children
○ A red node will never be the root of the tree
○ A red node always has a black parent node
○ Removing a red node will not change the number of black nodes along any path

3)
Re-computation of the sibling node after case RBTreeTryCase2, RBTreeTryCase5, or RBTreeTryCase6 implies that these functions may be doing what?

- ○ Recoloring the node or the node's parent
- ○ Recoloring the node's uncle or the node's sibling
- ○ Rotating at one of the node's children
- ○ Rotating at the node's parent or the node's sibling

4)
RBTreePrepareForRemoval performs the check `node→parent == null` on the first line. What other check is equivalent and could be used in place of the code `node→parent == null`?

- ○ `tree→root == null`
- ○ `tree→root == node`
- ○ `node→color == black`
- ○ `node→color == red`

Prepare-for-removal algorithm cases

Preparation for removing a node first checks for each of the six cases, performing the operations below.

1. If the node is red or the node's parent is null, then return.
2. If the node has a red sibling, then color the parent red and the sibling black. If the node is the parent's left child then rotate left at the parent, otherwise rotate right at the parent. Continue to the next step.
3. If the node's parent is black and both children of the node's sibling are black, then color the sibling red, recursively call on the node's parent, and return.
4. If the node's parent is red and both children of the node's sibling are black, then color the parent black, color the sibling red, then return.
5. If the sibling's left child is red, the sibling's right child is black, and the node is the left child of the parent, then color the sibling red and the left child of the sibling black. Then rotate right at the sibling and continue to the next step.
6. If the sibling's left child is black, the sibling's right child is red, and the node is the right child of the parent, then color the sibling red and the right child of the sibling black. Then rotate left at the sibling and continue to the next step.
7. Color the sibling the same color as the parent and color the parent black.
8. If the node is the parent's left child, then color the sibling's right child black and rotate left at the parent. Otherwise color the sibling's left child black and rotate right at the parent.

Table 7.8.1: Prepare-for-removal algorithm case descriptions.

Case #	Condition	Action if condition is true	Process additional cases after action?
1	Node is red or node's parent is null.	None.	No
2	Sibling node is red.	Color parent red and sibling black. If node is left child of parent, rotate left at parent node, otherwise rotate right at parent node.	Yes
3	Parent is black and both of sibling's children are black.	Color sibling red and call removal preparation function on parent.	No
4	Parent is red and both of sibling's children are black.	Color parent black and sibling red.	No
5	Sibling's left child is red, sibling's right child is black, and node is left child of parent.	Color sibling red and sibling's left child black. Rotate right at sibling.	Yes
6	Sibling's left child is black, sibling's right child is red, and node is right	Color sibling red and sibling's right child black. Rotate left at sibling.	Yes

child of parent.

Table 7.8.2: Prepare-for-removal algorithm case code.

Case #	Code

1	`RBTreeTryCase1(tree, node) {`
	`  if (node→color == red || node→parent == null)`
	`  return true`
	`  else`
	`  return false // not case 1`
	`}`

2	`RBTreeTryCase2(tree, node, sibling) {`
	`  if (sibling→color == red) {`
	`  node→parent→color = red`
	`  sibling→color = black`
	`  if (node == node→parent→left)`
	`  RBTreeRotateLeft(tree, node→parent)`
	`  else`
	`  RBTreeRotateRight(tree, node→parent)`
	`  return true`
	`  }`
	`  return false // not case 2`
	`}`

3	`RBTreeTryCase3(tree, node, sibling) {`
	`  if (node→parent→color == black &&`
	`  RBTreeAreBothChildrenBlack(sibling)) {`
	`  sibling→color = red`
	`  RBTreePrepareForRemoval(tree, node→parent)`
	`  return true`
	`  }`
	`  return false // not case 3`
	`}`

4	`RBTreeTryCase4(tree, node, sibling) {`
	`  if (node→parent→color == red &&`
	`  RBTreeAreBothChildrenBlack(sibling)) {`
	`  node→parent→color = black`
	`  sibling→color = red`
	`  return true`
	`  }`
	`  return false // not case 4`
	`}`

5	`RBTreeTryCase5(tree, node, sibling) {`
	`  if (RBTreeIsNonNullAndRed(sibling-→left) &&`
	`  RBTreeIsNullOrBlack(sibling→right) &&`
	`  node == nodeparent→left) {`
	`  sibling→color = red`
	`  sibling→left→color = black`
	`  RBTreeRotateRight(tree, sibling)`
	`  return true`

```
         }
         return false // not case 5
      }


      RBTreeTryCase6(tree, node, sibling) {
         if (RBTreeIsNullOrBlack(sibling→left) &&
             RBTreeIsNonNullAndRed(sibling→right) &&
             node == node→parent→right) {
            sibling→color = red
6           sibling→right→color = black
            RBTreeRotateLeft(tree, sibling)
            return true
         }
         return false // not case 6
      }
```

participation activity
7.8.4: Removal preparation, case 4.

Animation content:

undefined

Animation captions:

1. In the above tree, all paths from root to null leaves have 3 black nodes.
2. Preparation for removal of node 62 encounters case 4, since the node's parent is red and both children of the sibling are black (null).
3. The parent is colored black and the sibling is colored red.
4. The preparation leaves the tree in a state where node 62 can be removed and all red-black tree requirements would be met.

participation activity
7.8.5: Removal preparation for a node can encounter more than 1 case.

Animation content:

undefined

Animation captions:

1. Preparation for removal of node 75 first encounters case 2 in RBTreePrepareForRemoval.
2. After making alterations for case 2, the code proceeds to additional case checks, ending after case 4 alterations.
3. In the resulting tree, node 75 can be removed via BSTRemove and all red-black tree requirements will hold.

participation activity
7.8.6: Prepare-for-removal algorithm cases.

- RBTreeTryCase1
- RBTreeTryCase5
- RBTreeTryCase2
- RBTreeTryCase4
- RBTreeTryCase3
- RBTreeTryCase6

This case function always returns true if passed a node with a red sibling.
This case function finishes preparation exclusively by recoloring nodes.
This case function never returns true if the node is the right child of the node's parent.
This case function never alters the tree.
When this case function returns true, a left rotation at the node's sibling will have just taken place.
This case function recursively calls RBTreePrepareForRemoval if the node's parent and both children of the node's sibling are black.

Reset