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.

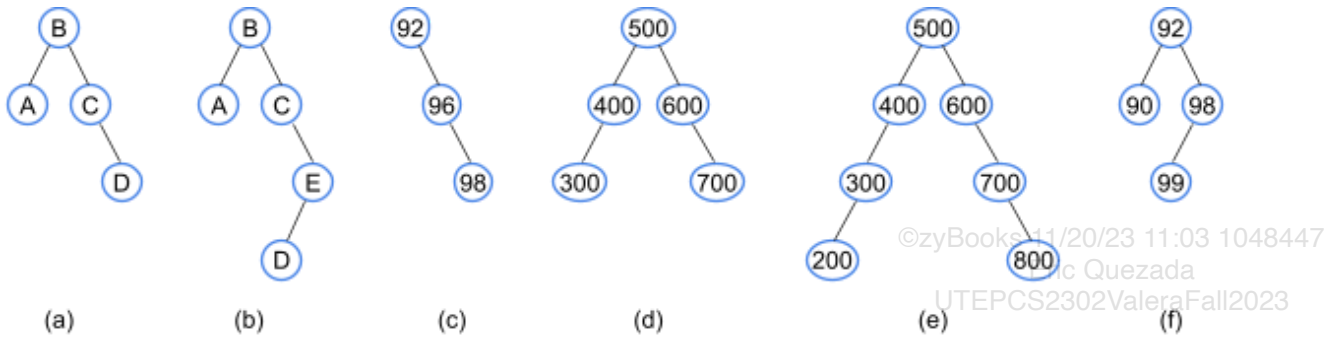
PARTICIPATION ACTIVITY 7.1.1: An AVL tree is height balanced: For any node, left and right subtree heights differ by only 0 or 1.

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

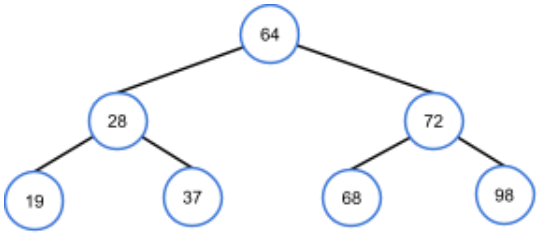
6) (f)

- Yes
- No

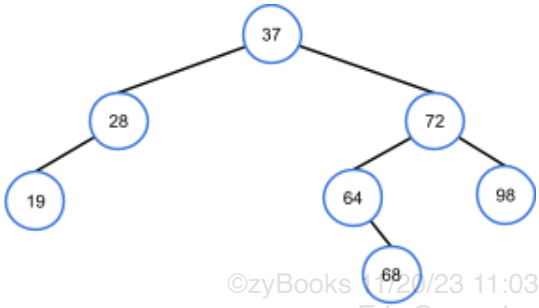
AVL tree height

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.

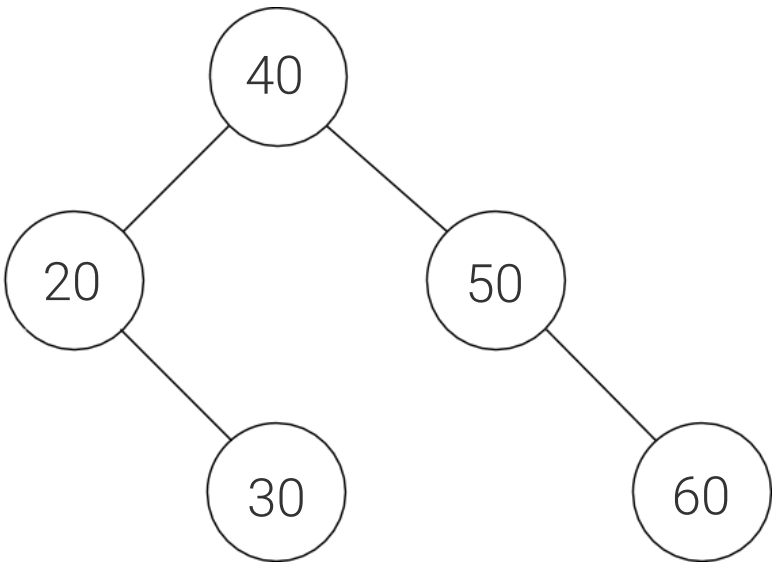
3) When would inserting a new node to an AVL tree result in no height value changes for all ancestors?

- 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.

502696.2096894.qx3zqy7

Start



What is the height of 20? Ex: 5

20's left subtree height:

20'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

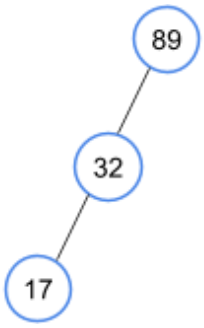
Animation captions:

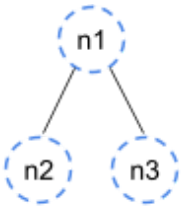
1. This BST violates the AVL height balance property.
2. A right rotation balances the tree while maintaining the BST ordering property.

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.





If unable to drag and drop, refresh the page.

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.

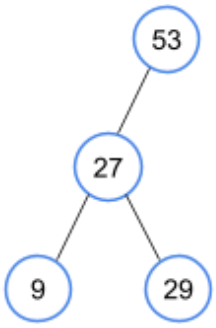
PARTICIPATION ACTIVITY 7.2.3: In a right rotate, B's former right child C becomes D's left child, to maintain the BST ordering property.

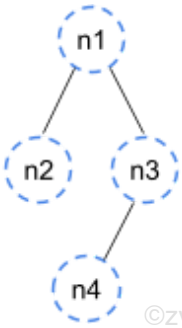
Animation captions:

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.





If unable to drag and drop, refresh the page.

27 53 29 9

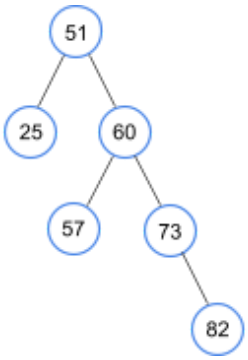
	n1
	n2
	n3
	n4

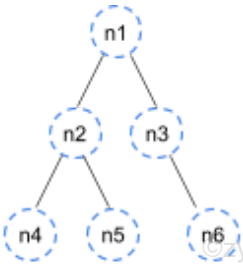
Reset

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.





If unable to drag and drop, refresh the page.

51 57 25 73 82 60

	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.

©zyBooks 11/20/23 11:03 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
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
}
```


©zyBooks 11/20/23 11:03 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

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

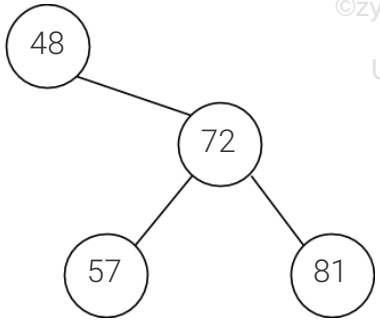
4) AVLTreeRebalance takes no action if a node's balance factor is 1, 0, or -1.
   - True
   - False

CHALLENGE ACTIVITY 7.2.1: AVL rotations.

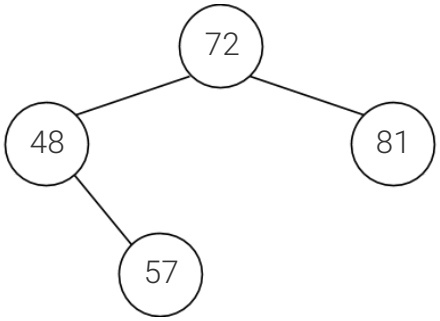
502696.2096894.qx3zqy7

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.

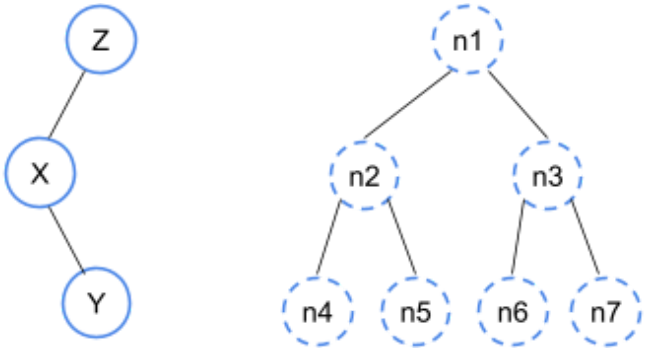
PARTICIPATION ACTIVITY 7.3.2: Sometimes a double rotation is necessary to rebalance.

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

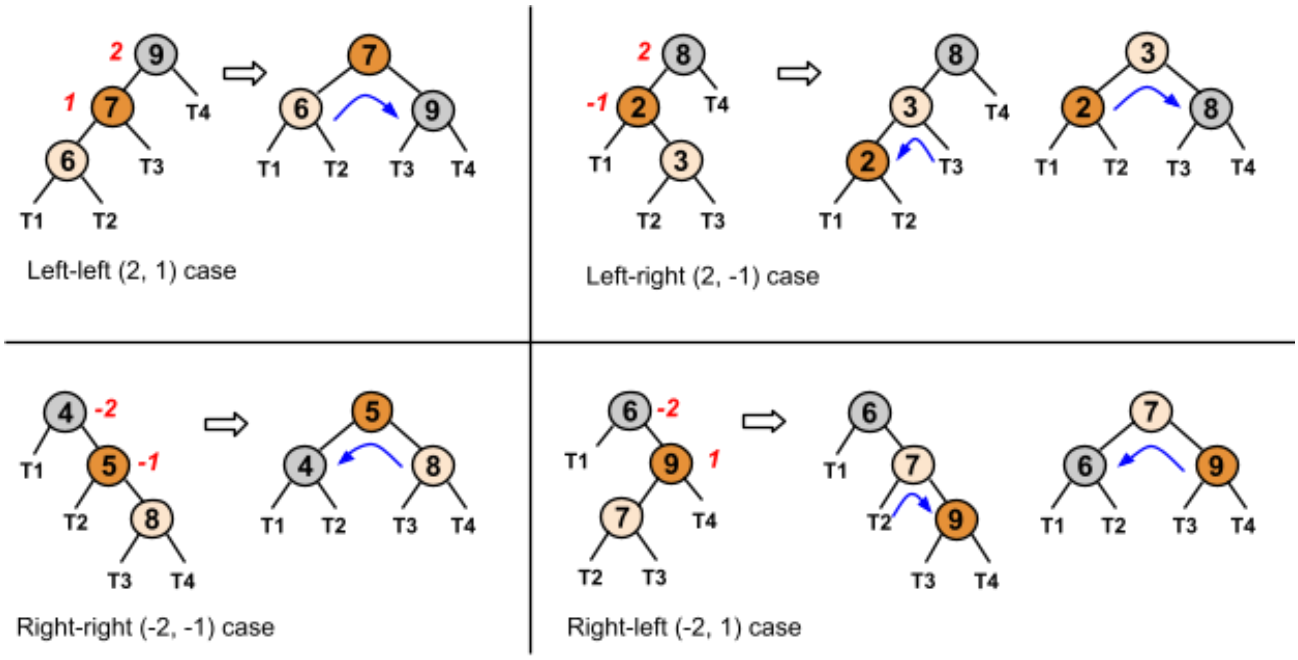
After inserting a node, nodes on the path from the new node to the root should be checked for a balance factor of 2 or -2. The first such node P triggers rebalancing. Four cases exist, distinguishable by the balance factor of node P and one of P's children.

PARTICIPATION ACTIVITY 7.3.4: Four AVL imbalance cases are possible after inserting a new node.

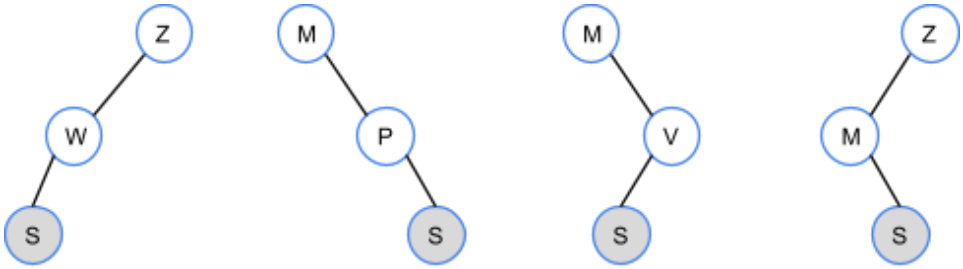
Animation captions:

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: Imbalance cases and appropriate rotations.



(a) (b) (c) (d)

1) In (a), Z's balance factor is _____ and W's balance factor is _____.

- 2, 1
- 2, -1
- -2, -1
- -2, 1

2) In (b), M's balance factor is _____ and P's balance factor is _____.

- 2, 1
- 2, -1
- -2, -1
- -2, 1

3) In (c), M's balance factor is _____ and V's balance factor is _____.

- 2, 1
- 2, -1
- -2, -1
- -2, 1

4) In (d), Z's balance factor is _____ and M's balance factor is _____.

- 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

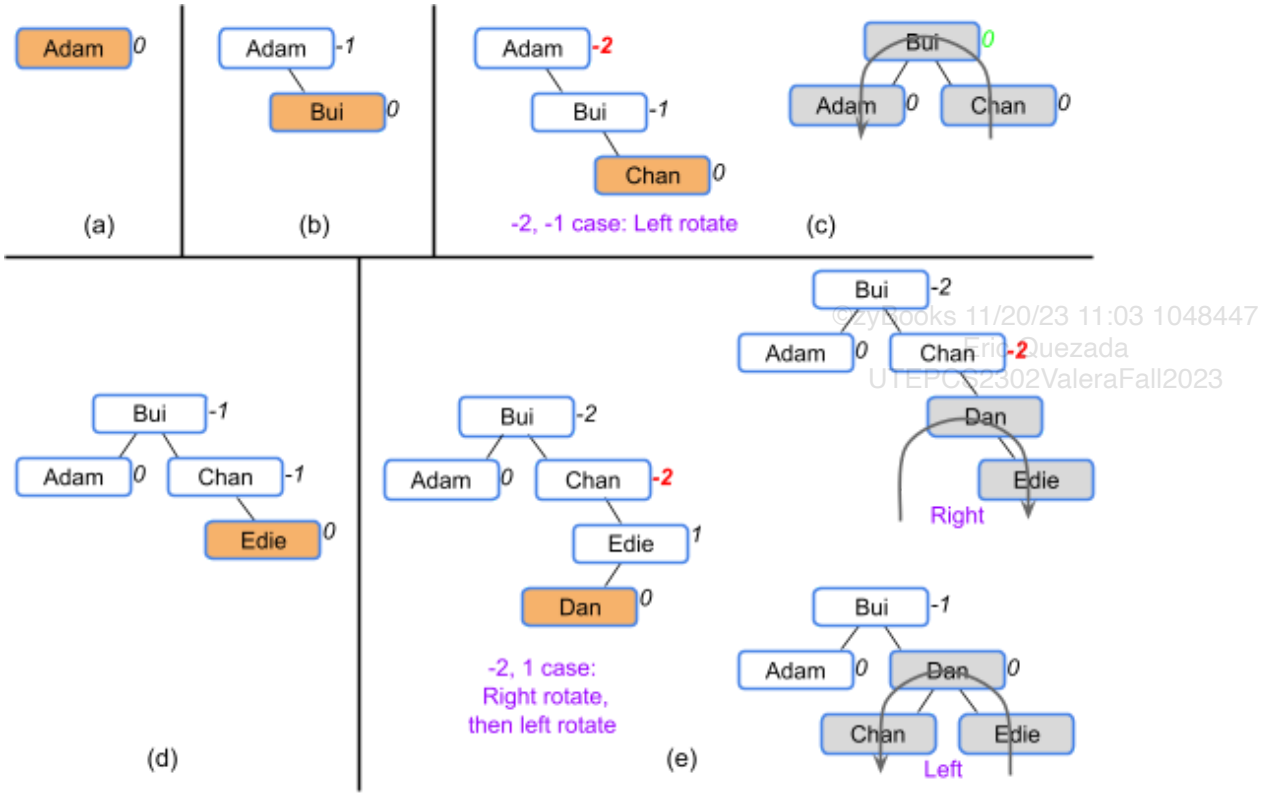
Insertion with rebalancing

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

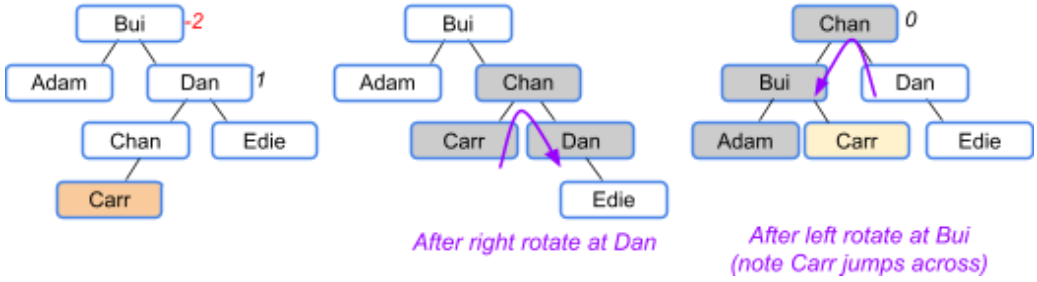
Balance factor updates are only needed on nodes ascending along the path from the inserted node up to the root, since no other nodes' balance could be affected. Each node's balance factor can be recomputed by determining left and right subtree heights, or for speed can be stored in each node and then 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

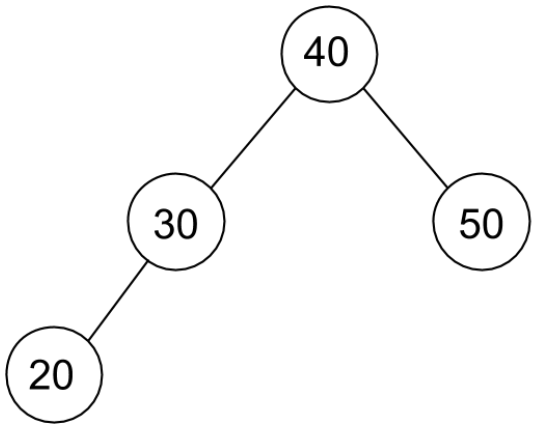
The AVL insertion algorithm traverses the tree from the root to a leaf node to find the insertion point, then traverses back up to the root to rebalance. One node is visited per level, and at most 2 rotations are needed for a single node. Each rotation is an O(1) operation. Therefore, the runtime complexity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

CHALLENGE ACTIVITY 7.3.1: AVL insertions.

502696.2096894.qx3zqy7

Start



Inserting which nodes would unbalance the AVL tree?

- [ ] 19
- [ ] 33
- [ ] 21
- [ ] 43
- [ ] 24
- [ ] 64

1 2 3 4 5

Check Next

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 content:

Step 1: An AVL tree with seven nodes is shown. Root's key is 84. Root's left child has key 61 and the right child has key 89. Node 61's left child has key 21 and the right child has key 63. Node 89's right child has key 93. Node 21's left child has key 12.
A label indicating the removal operation to perform appears: "Remove 63". Label "Node to remove" appears by node 63 and "Parent" by node 61. Node 63 and the accompanying label then disappear.

Step 2: Node 61's balance factor is 2, so a right rotation occurs at node 61. The resulting tree: root node 84, root's left child is node 21, root's right child is node 89, node 21's left child is node 12, node 21s right child is node 61, and node 89's right child is node 93.
After the rotation, ancestor nodes 21 and 84 have a balance factor of 0. So no further modifications occur and removal of 63 is done.

Step 3: A label indicating the removal operation to perform appears: "Remove 89". A label below says "Successor is 89". Key 89 is copied from the root's right child into the root.

Step 4: The root's right child is removed, resulting in an AVL tree with five nodes: root has key 89, root's left child has key 21, root's right child has key 93, node 21's left child has key 12, and node 21's right child has key 61. The root's balance factor is 1, so no further modifications occur and removal of 84 is done.

Step 5: A label indicating the removal operation to perform appears: "Remove 93". Node 93 disappears, leaving the root with no right child and a balance factor of 2. A right rotation at node 89 occurs, yielding an AVL tree with four nodes: root's key is 21, root's left child has key 12, root's right child has key 89, and node 89's left child has key 61. Node 21's balance factor is -1, so no further modifications occur and removal of 93 is done.
Final frame shows original AVL tree on the left and final AVL tree on the right.

Animation captions:

1. Removing 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 occur if the balance factor (BF) is 2 or -2.
3. Removal of 84 starts by replacing key 84 with the successor's key, 89.
4. Node 89 is then removed from the right subtree. The root's balance factor becomes 1, so no rotations are needed.
5. After the standard BST removal algorithm removes node 93, the root is left with a balance factor of 2. A right rotation at 89 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)
}
```

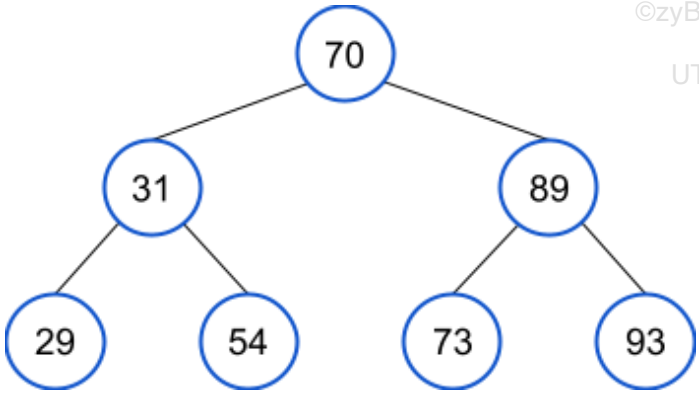

Figure 7.4.3: AVLTreeRemoveNode algorithm.

©zyBooks 11/20/23 11:03 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
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 key from the successor node
      node→key = succNode→key

      // 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 != null) {
         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:



If unable to drag and drop, refresh the page.

6 4 5 2 1 Never 3

	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 Python: AVL Trees

The Node class for AVL Trees

Because AVL Trees are a form of a binary search tree, the AVLTree class is similar to the BinarySearchTree class. A Node class is used that contains data members key, left, and right, as well as two new data members:

- parent - A pointer to the parent node, or None for the root.
- height - The height of the subtree at the node. A single node has a height of 0.

The height data member is used to detect imbalances in the tree after insertions or removals, while the parent data member is used during rotations to correct imbalances. The Node class contains methods that are useful for AVL operations:

- get_balance() - Returns the node's balance factor. The balance factor is the left child's height minus the right child's height. A height of -1 is used in the calculation if the child is None.
- update_height() - Calculates the node's current height and assigns the height data member with the new value.
- set_child() - Assigns either the left or right child data members with a new node.
- replace_child() - Replaces a current child node with a new node.

Figure 7.5.1: Node class for AVL trees.

©zyBooks 11/20/23 11:03 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
class Node:
    # Constructor with a key
parameter creates the Node object.
    def __init__(self, key):
        self.key = key
        self.parent = None
        self.left = None
        self.right = None
        self.height = 0

    # Calculate the current nodes'
balance factor,
    # defined as height(left
subtree) - height(right subtree)
    def get_balance(self):
        # Get current height of
left subtree, or -1 if None
        left_height = -1
        if self.left is not None:
            left_height =
self.left.height

        # Get current height of
right subtree, or -1 if None
        right_height = -1
        if self.right is not None:
            right_height =
self.right.height

        # Calculate the balance
factor.
        return left_height -
right_height

    # Recalculate the current
height of the subtree rooted at
    # the node, usually called
after a subtree has been
    # modified.
    def update_height(self):
        # Get current height of
left subtree, or -1 if None
        left_height = -1
        if self.left is not None:
            left_height =
self.left.height

        # Get current height of
right subtree, or -1 if None
        right_height = -1
        if self.right is not None:
            right_height =
self.right.height

        # Assign self.height with
calculated node height.
        self.height =
max(left_height, right_height) + 1

# Assign either the left or right
data member with a new
# child. The parameter which_child
is expected to be the
# string "left" or the string
"right". Returns True if
# the new child is successfully
assigned to this node, False
# otherwise.
def set_child(self, which_child,
child):
    # Ensure which_child is properly
assigned.
    if which_child != "left" and
which_child != "right":
        return False

    # Assign the left or right data
member.
    if which_child == "left":
        self.left = child
    else:
        self.right = child

    # Assign the parent data member
of the new child,
    # if the child is not None.
    if child is not None:
        child.parent = self

    # Update the node's height,
since the subtree's structure
    # may have changed.
    self.update_height()
    return True

# Replace a current child with a new
child. Determines if
# the current child is on the left
or right, and calls
# set_child() with the new node
appropriately.
# Returns True if the new child is
assigned, False otherwise.
def replace_child(self,
current_child, new_child):
    if self.left is current_child:
        return
self.set_child("left", new_child)
    elif self.right is
current_child:
        return
self.set_child("right", new_child)

    # If neither of the above cases
applied, then the new child
    # could not be attached to this
node.
    return False
```

PARTICIPATION ACTIVITY 7.5.1: Working with Nodes.

For questions 1 - 3, provide the values based on the code below.


```
node_a = Node(1)
node_b = Node(2)
node_c = Node(3)
node_d = Node(4)
node_e = Node(5)
node_f = Node(6)

node_a.set_child('left', node_b)
node_a.set_child('right', node_c)
node_b.set_child('left', node_d)
node_c.set_child('left', node_e)
node_e.set_child('right', node_f)
```

1) `node_a.height`

Check Show answer

2) `node_b.right`

Check Show answer

3) `node_e.parent.parent.left.key`

Check Show answer

4) Is new_node the left or right child of node_c? `new_node = Node(6)`
`node_c.replace_child(node_e, new_node)`

Check Show answer

Rotations and rebalancing

Tree rotations are required to correct any problems where the left and right subtrees of a node have heights that differ by more than 1. After a new node is inserted into an AVL tree, either one or two rotations will fix any imbalance that happens. The rotate_left() and rotate_right() methods perform these operations. The rebalance() method examines the structure of the subtree of a node, and determines which rotations to do if a height imbalance exists at the node.

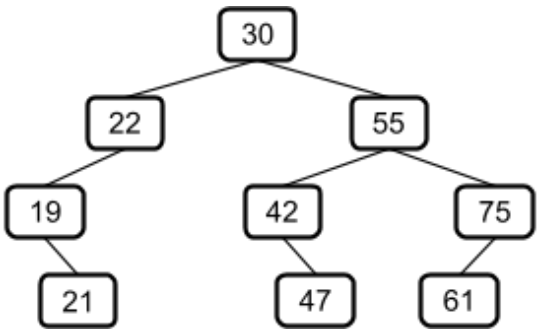
Figure 7.5.2: The AVLTree class with rotate_left(), rotate_right() and rebalance() methods.

```
class AVLTree:
    # Constructor to create an empty AVLTree. There is only
    # one data member, the tree's root Node, and it starts
    # out as None.
    def __init__(self):
        self.root = None

    # Performs a left rotation at the given node. Returns the
    # new root of the subtree.
    def rotate_left(self, node):
        # Define a convenience pointer to the left child of the
        # right child.
        right_left_child = node.right.left

        # Step 1 - the right child moves up to the node's position.
        # This detaches node from the tree, but it will be
reattached
        # later.
        if node.parent is not None:
            node.parent.replace_child(node, node.right)
        else:  # node is root
            self.root = node.right
            self.root.parent = None

        # Step 2 - the node becomes the left child of what used
        # to be its right child, but is now its parent. This will
        # detach right_left_child from the tree.
        node.right.set_child('left', node)

        # Step 3 - reattach right_left_child as the right child of
node.
        node.set_child('right', right_left_child)

        return node.parent

    # Performs a right rotation at the given node. Returns the
    # subtree's new root.
    def rotate_right(self, node):
        # Define a convenience pointer to the right child of the
        # left child.
        left_right_child = node.left.right

        # Step 1 - the left child moves up to the node's position.
        # This detaches node from the tree, but it will be
reattached
        # later.
        if node.parent is not None:
            node.parent.replace_child(node, node.left)
        else:  # node is root
            self.root = node.left
            self.root.parent = None

        # Step 2 - the node becomes the right child of what used
        # to be its left child, but is now its parent. This will
        # detach left_right_child from the tree.
        node.left.set_child('right', node)

        # Step 3 - reattach left_right_child as the left child of
node.
        node.set_child('left', left_right_child)
```

```
        return node.parent

    # Updates the given node's height and rebalances the subtree if
    # the balancing factor is now -2 or +2. Rebalancing is done by
    # performing a rotation. Returns the subtree's new root if
    # a rotation occurred, or the node if no rebalancing was
required.
    def rebalance(self, node):

        # First update the height of this node.
        node.update_height()

        # Check for an imbalance.
        if node.get_balance() == -2:

            # The subtree is too big to the right.
            if node.right.get_balance() == 1:
                # Double rotation case. First do a right rotation
                # on the right child.
                self.rotate_right(node.right)

            # A left rotation will now make the subtree balanced.
            return self.rotate_left(node)

        elif node.get_balance() == 2:

            # The subtree is too big to the left
            if node.left.get_balance() == -1:
                # Double rotation case. First do a left rotation
                # on the left child.
                self.rotate_left(node.left)

            # A right rotation will now make the subtree balanced.
            return self.rotate_right(node)

        # No imbalance, so just return the original node.
        return node
```


PARTICIPATION ACTIVITY 7.5.2: AVL tree rebalancing and rotations.

For the following questions, the method `tree.rebalance(node)` is called where `node` as a pointer to the node with key 22 in the tree below. The key 21 has just been inserted, causing an imbalance.



1) What is the value of `node.get_balance()`?

- -2
- -1
- 2

2) What is the value of `node.left.get_balance()`?

- -1
- 0
- 1

3) What rotation method(s) will be called by rebalance()?

- rotate_left() only
- rotate_right() only
- rotate_left() followed by rotate_right()
- rotate_right() followed by rotate_left()

4) What is the key in the node returned by the rebalance() method?

- 19
- 21
- 22
- 30

Insertions

AVL insertions are performed in two steps:

1. Insert into the tree using the normal binary search tree insertion algorithm.
2. Call rebalance() on all nodes along a path from the new node's parent up to the root.

Step 1 requires steps, since an AVL tree has levels, and the loop makes the current node go down one level with each iteration. Step 2 requires steps, again since the path back up to the root has levels to visit, and rotations are operations. Thus, the insert() method has a worst-case runtime of .

Figure 7.5.3: The AVLTree insert() method.

```
def insert(self, node):

    # Special case: if the tree is empty, just set the root to
    # the new node.
    if self.root is None:
        self.root = node
        node.parent = None

    else:
        # Step 1 - do a regular binary search tree insert.
        current_node = self.root
        while current_node is not None:
            # Choose to go left or right
            if node.key < current_node.key:
                # Go left. If left child is None, insert the new
                # node here.
                if current_node.left is None:
                    current_node.left = node
                    node.parent = current_node
                    current_node = None
                else:
                    # Go left and do the loop again.
                    current_node = current_node.left
            else:
                # Go right. If the right child is None, insert the
                # new node here.
                if current_node.right is None:
                    current_node.right = node
                    node.parent = current_node
                    current_node = None
                else:
                    # Go right and do the loop again.
                    current_node = current_node.right

        # Step 2 - Rebalance along a path from the new node's
parent up
        # to the root.
        node = node.parent
        while node is not None:
            self.rebalance(node)
            node = node.parent
```


PARTICIPATION ACTIVITY 7.5.3: AVL insertion.

1) A rotation is a operation.

- True
- False

2) AVL insertion starts with the same algorithm as binary search tree insertion.

- True
- False

3) Rotations can cause an imbalance in the left or right subtree, so rebalance() must be called on each child node.

- True
- False

Removals

Removal from an AVL tree is a two-step process, similar to insertion:

1. Remove the node in the same way nodes are removed from a binary search tree. One of four cases is identified to determine how to remove the node. In the most general case, the node's key is replaced by the successor node's key in the tree, and the successor is then more easily removed.
2. Call rebalance() on all the nodes on the path from the removed node's parent up to the root. If the node's successor was ultimately removed, the rebalancing begins from the successor's parent, not the original target node.

As with insertion, each step requires operations to be performed, first on a path from the root down to a leaf, and then on a path near a leaf back up to the root. Because the height of an AVL tree is guaranteed to be , the entire remove algorithm runs in worst-case time.

Figure 7.5.4: The AVLTree remove_node() method.

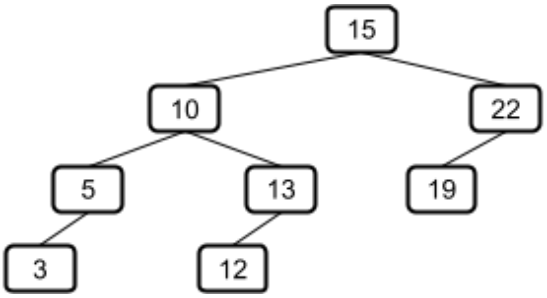
```
def remove_node(self, node):
    # Base case:
    if node is None:
        return False

    # Parent needed for rebalancing.
    parent = node.parent

    # Case 1: Internal node with 2 children
    if node.left is not None and node.right is not None:
        # Find successor
        successor_node = node.right
        while successor_node.left != None:
            successor_node = successor_node.left

        # Copy the value from the node
        node.key = successor_node.key

        # Recursively remove successor
        self.remove_node(successor_node)

        # Nothing left to do since the recursive call will have rebalanced
        return True

    # Case 2: Root node (with 1 or 0 children)
    elif node is self.root:
        if node.left is not None:
            self.root = node.left
        else:
            self.root = node.right

        if self.root is not None:
            self.root.parent = None

        return True

    # Case 3: Internal with left child only
    elif node.left is not None:
        parent.replace_child(node, node.left)

    # Case 4: Internal with right child only OR leaf
    else:
        parent.replace_child(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 is not None:
        self.rebalance(node)
        node = node.parent

    return True
```

Often the user does not know where the desired node object to be removed is in the tree, or even if the node exists at all. The remove_key() method can be used to first search for the node using the BinarySearchTree search() method, and then call remove_node() only if search()returns a Node pointer.

Figure 7.5.5: The search() and remove_key() methods.


```
# Searches for a node with a matching key. Does a regular
# binary search tree search operation. Returns the node with the
# matching key if it exists in the tree, or None if there is no
# matching key in the tree.
def search(self, key):
    current_node = self.root
    while current_node is not None:
        # Compare the current node's key with the target key.
        # If it is a match, return the current key; otherwise go
        # either to the left or right, depending on whether the
        # current node's key is smaller or larger than the target key.
        if current_node.key == key: return current_node
        elif current_node.key < key: current_node = current_node.right
        else: current_node = current_node.left

# Attempts to remove a node with a matching key. If no node has a matching
key
# then nothing is done and False is returned; otherwise the node is
removed and
# True is returned.
def remove_key(self, key):
    node = self.search(key)
    if node is None:
        return False
    else:
        return self.remove_node(node)
```

PARTICIPATION ACTIVITY 7.5.4: AVL removal.



1) Removing node 10:

- replaces the 10 node's key with value 12, and removes the node 12.
- moves node 13 up into node 10's position.
- causes a rotation at node 10.

2) Removing node 13:

- replaces the 10 node's key with value 12, and removes the node 13.
- moves node 12 up into node 13's position.
- causes a rotation at node 10.

3) Removing node 19:

- replaces the 15 node's key with value 22, and removes the node 19.
- moves node 19 up into node 22's position.
- causes a rotation at node 15.

zyDE 7.5.1: Exploring the AVLTree class.

The following program inserts some nodes into an AVL tree and then removes some nod
Try the following activities to see if you can predict how the tree will change:

- Add the value 30 in the middle of the keys list (after the 15). How does the insertion change the initial tree?
- What key could be added to the end of the keys list that would cause at least one rotation when the node is inserted?
- Remove the root (15) immediately after all the nodes are inserted. Does the remova change the tree's height?
- Which nodes will trigger a rotation when removed?

Current file: main.py Load default templat

```
1 from AVLTree import AVLTree, Node
2
```

```
# Create an empty AVLTree object.
tree = AVLTree()

# Insert some random-looking integers into the tree.
keys = [ 10, 20, 5, 22, 15, 47, 19, 3, 12, 18 ]
for key in keys:
    node = Node(key)
    tree.insert(node)

# Print the tree after all inserts are complete.
print("Tree after initial insertions:")
print(tree)
```


Run

7.6 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.

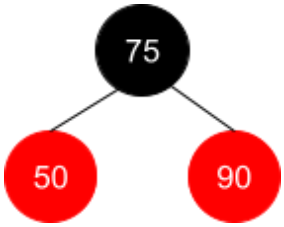
PARTICIPATION ACTIVITY 7.6.1: Red-black tree rules.

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.

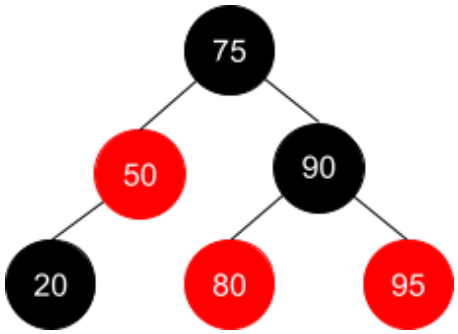
PARTICIPATION ACTIVITY 7.6.2: Red-black tree rules.

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



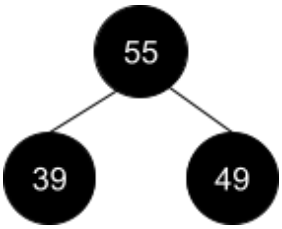
- Root node must be black.
- A red node's children cannot be red.
- All paths from a node to null leaf nodes must have the same number of black nodes.
- None.

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



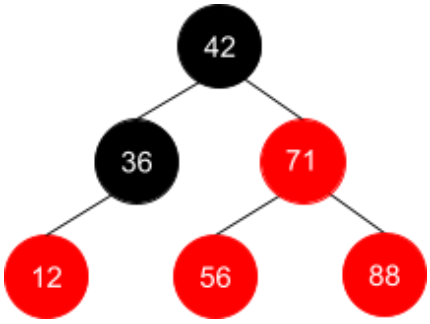
- Root node must be black.
- A red node's children cannot be red.
- Not all levels are full.
- All paths from a node to null leaf nodes must have the same number of black nodes.

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

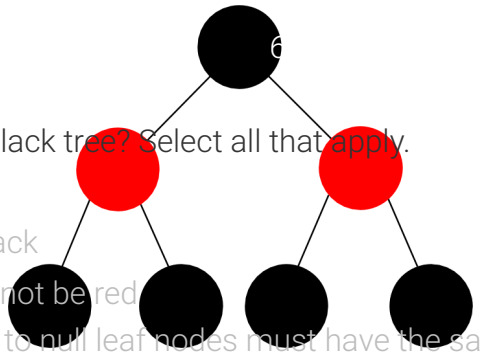
7) Any BST can be made a red-black tree by coloring all nodes black.

- True
- False

CHALLENGE ACTIVITY 7.6.1: Red-black tree: A balanced tree.

502696.2096894.qx3zqy7

Start



Is this binary tree 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

1 2 3

Check Next

7.7 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.

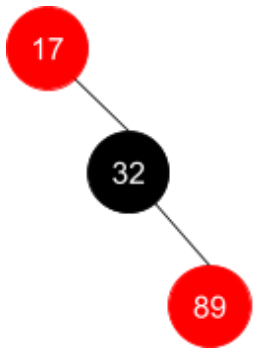
PARTICIPATION ACTIVITY 7.7.1: A simple left rotation in a red-black tree.

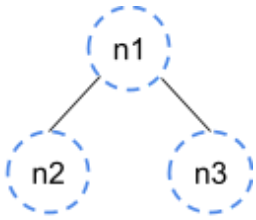
Animation captions:

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 7.7.2: 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.





If unable to drag and drop, refresh the page.

32 17 89

	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 7.7.1: 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 7.7.2: 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 7.7.3: 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 7.7.3: RBTreeRotateLeft algorithm.

If unable to drag and drop, refresh the page.

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

	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 7.7.4: 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 7.7.5: 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.

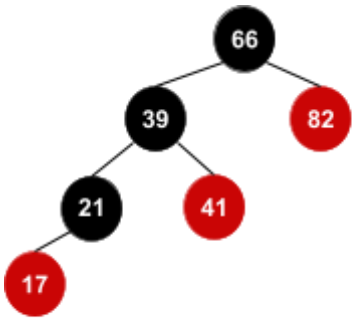
- True
- False

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

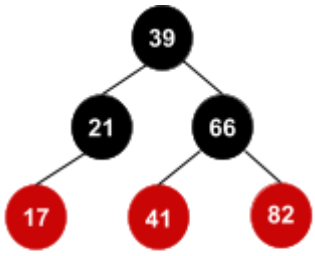
- True
- False

PARTICIPATION ACTIVITY 7.7.6: Red-black tree rotations.

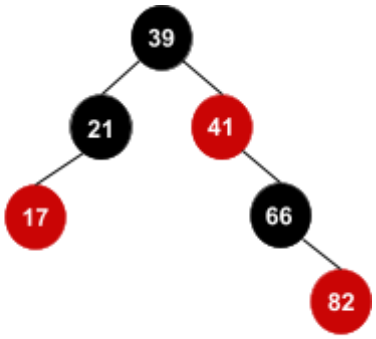
Consider the three trees below:



Tree 1



Tree 2



Tree 3

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 yields 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 yields a valid red-black tree.

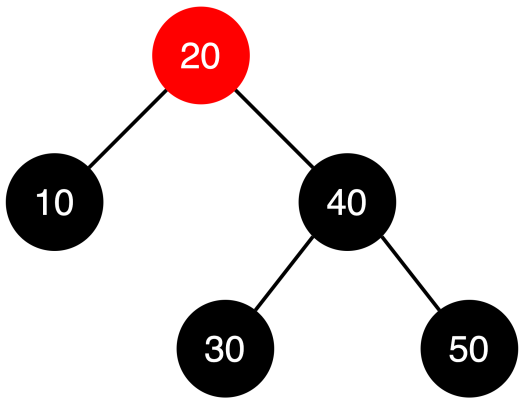
- True
- False

CHALLENGE ACTIVITY 7.7.1: Red-black tree: Rotations.

502696.2096894.qx3zqy7

Start

An invalid red-black tree is shown below.



A [ ] rotation at node [ ] yields a valid red-black tree.

Check Next

7.8 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 7.8.1: 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 7.8.2: 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 7.8.1: RBTreeBalance algorithm.

Animation content:

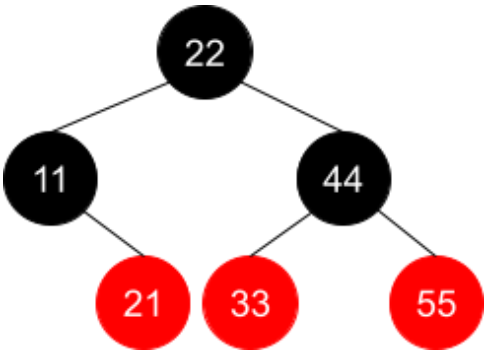
undefined

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 7.8.2: 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.

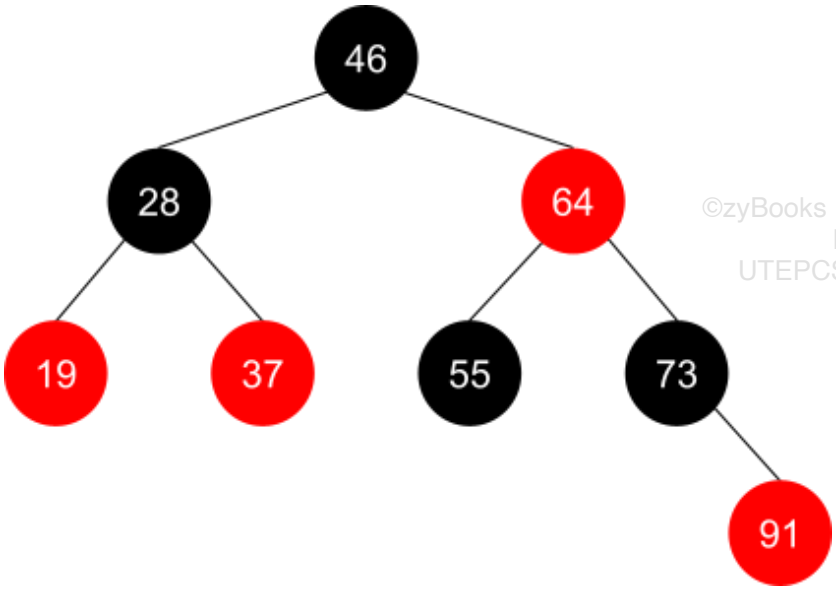
- True
- False

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 7.8.3: RBTreeInsert algorithm.

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



If unable to drag and drop, refresh the page.

Never | 5 | 4 | 2 | 1 | 3

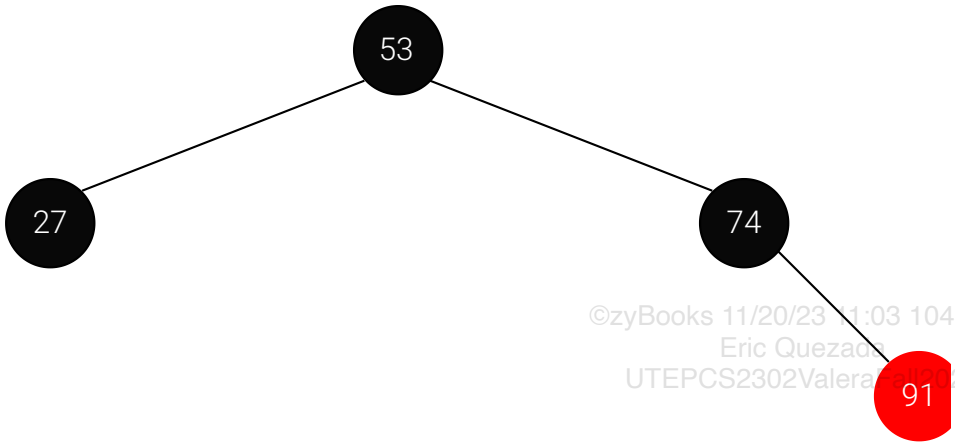
	Rotate left at node 73.
	Insert red node 82 as node 91's left child.
	Color grandparent node red.
	Color parent node black.
	Rotate right at node 91.
	Call RBTreeBalance recursively on node 73.

Reset

CHALLENGE ACTIVITY 7.8.1: Red-black tree: Insertion.

502696.2096894.qx3zqy7

Start



Node 18 is inserted. Prior to any rotations, what is node 18's _____?

Parent node: Ex: 10 or null

Grandparent node:

Uncle node:

Is a rotation required to complete the insertion?

1 2 3

Check Next

7.9 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 7.9.1: RBTreeRemove algorithm.

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


Given a key, a red-black tree ***remove-key*** operation removes the key from the tree, if present, restructuring as needed to preserve all red-black tree requirements. First, BSTSearch() is called to find the node containing the key. If the node is found, RBTreeRemoveNode() is called to remove the node.

RBTreeRemoveNode() consists of the following steps:

1. If the node has two children, copy the key from 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 removal algorithm.

Figure 7.9.2: 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 7.9.3: RBTreeGetPredecessor utility function.

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


PARTICIPATION ACTIVITY 7.9.1: 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 7.9.4: RBTreeGetSibling algorithm.

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


Figure 7.9.5: RBTreeIsNonNullAndRed algorithm.

```
RBTreeIsNonNullAndRed(node)
{
   if (node == null)
      return false
   return (node→color ==
red)
}
```

Figure 7.9.6: RBTreeIsNullOrBlack algorithm.

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

Figure 7.9.7: 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 7.9.2: 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 7.9.8: 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 7.9.3: 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 7.9.1: 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 child of parent.	Color sibling red and sibling's right child black. Rotate left at sibling.	Yes

Table 7.9.2: 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`
	`}`

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

PARTICIPATION ACTIVITY 7.9.4: 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 7.9.5: 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 7.9.6: Prepare-for-removal algorithm cases.

If unable to drag and drop, refresh the page.

RBTreeTryCase6 **RBTreeTryCase5** **RBTreeTryCase2** **RBTreeTryCase4**

RBTreeTryCase1 **RBTreeTryCase3**

	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

7.10 Python: Red-black trees

The RBTNode class for red-black trees

The RedBlackTree class is similar to the BinarySearchTree class because a red-black tree is a form of a binary search tree. The RBTNode class, representing red-black tree nodes, contains data members key, left, and right, as well as two new data members:

- parent - A pointer to the parent node, or None for the root.
- color - A string representing the node's color, set to either "red" or "black".

The RBTNode class also contains methods useful for various operations:

Table 7.10.1: RBTNode class method descriptions.

Method name	Description
are_both_children_black()	Returns true if both child nodes are black. A child set to None is considered to be black, as per the red-black tree requirements.
count()	Returns the number of nodes in this subtree, including the node itself.
get_grandparent()	Returns this node's grandparent, or None if this node has no grandparent.
get_predecessor()	Returns the predecessor from this node's left subtree.
get_sibling()	Returns this node's sibling, which is the other child of this node's parent. Returns None if this node has no sibling.

get_uncle()	Returns this node's uncle, which is the sibling of this node's parent. Returns None if this node has no uncle.
is_black()	Returns True if this node is black, False otherwise.
is_red()	Returns True if this node is red, False otherwise.
replace_child()	Replaces a current child node with a new node. Takes two RBTNode arguments: current_child and new_child, and if current_child is one of the node's children, replaces current_child with new_child.
set_child()	Assigns either the left or right child data members with a new node. Takes two arguments: which_child (string) and child (RBTNode), and sets child as the node's left child if which_child is "left", or sets child as the right child if which_child is "right".

Figure 7.10.1: RBTNode class for red-black trees.

```
# RBTNode class - represents a node in a red-black tree
class RBTNode:
    def __init__(self, key, parent, is_red = False, left = None, right =
None):
        self.key = key
        self.left = left
        self.right = right
        self.parent = parent

        if is_red:
            self.color = "red"
        else:
            self.color = "black"

    # Returns true if both child nodes are black. A child set to None is
considered
    # to be black.
    def are_both_children_black(self):
        if self.left != None and self.left.is_red():
            return False
        if self.right != None and self.right.is_red():
            return False
        return True

    def count(self):
        count = 1
        if self.left != None:
            count = count + self.left.count()
        if self.right != None:
            count = count + self.right.count()
        return count

    # Returns the grandparent of this node
    def get_grandparent(self):
        if self.parent is None:
            return None
        return self.parent.parent

    # Gets this node's predecessor from the left child subtree
    # Precondition: This node's left child is not None
    def get_predecessor(self):
        node = self.left
        while node.right is not None:
            node = node.right
        return node

    # Returns this node's sibling, or None if this node does not have a
sibling
    def get_sibling(self):
        if self.parent is not None:
            if self is self.parent.left:
                return self.parent.right
            return self.parent.left
        return None

    # Returns the uncle of this node
    def get_uncle(self):
        grandparent = self.get_grandparent()
        if grandparent is None:
            return None
```

```
        if grandparent.left is self.parent:
            return grandparent.right
        return grandparent.left

    # Returns True if this node is black, False otherwise
    def is_black(self):
        return self.color == "black"

    # Returns True if this node is red, False otherwise
    def is_red(self):
        return self.color == "red"

    # Replaces one of this node's children with a new child
    def replace_child(self, current_child, new_child):
        if self.left is current_child:
            return self.set_child("left", new_child)
        elif self.right is current_child:
            return self.set_child("right", new_child)
        return False

    # Sets either the left or right child of this node
    def set_child(self, which_child, child):
        if which_child != "left" and which_child != "right":
            return False

        if which_child == "left":
            self.left = child
        else:
            self.right = child

        if child != None:
            child.parent = self

        return True
```


PARTICIPATION ACTIVITY 7.10.1: RBTNode class.

1) If both of a node's children are set to None, are_both_children_black() returns _____.

   - True
   - False

2) count() returns 1 for a node that has no children.

   - True
   - False

3) get_grandparent() never returns None for any node except the root.

- True
- False

4) get_predecessor() returns None for a node with a left child set to None.

- True
- False

5) get_sibling() returns None only for the root node.

- True
- False

6) get_uncle() returns None if the node has no grandparent.

- True
- False

Rotations and insertion

Tree rotations are required to help correct red-black tree requirement violations during an insertion or removal. The rotate_left() and rotate_right() RedBlackTree class methods perform left and right rotations, respectively.

The RedBlackTree class contains 3 methods for insertion:

- insert() - Takes a key as an argument, creates a new RBTNode containing the key, and then adds the node to the tree using insert_node().
- insert_node() - Takes a node as an argument, adds the node to the tree using standard BST insertion rules, and then calls insertion_balance() to balance the tree.
- insertion_balance() - Alters the tree as necessary to ensure red-black tree requirements are met after inserting a new node.

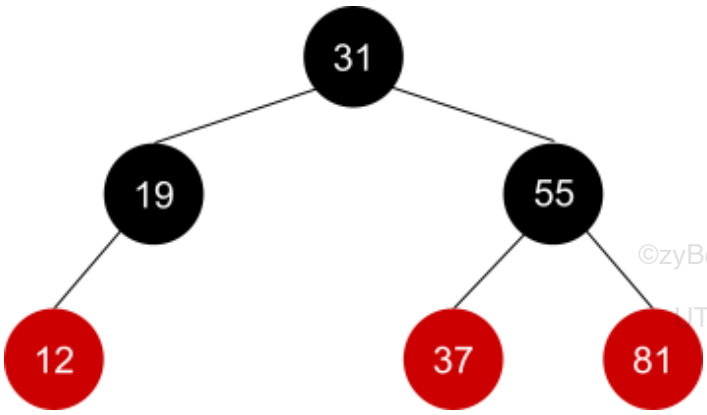
Figure 7.10.2: RedBlackTree class methods for rotations and insertion.

```
class RedBlackTree:
    def __init__(self):
        self.root = None

    def __len__(self):
        if self.root is None:
            return 0
        return self.root.count()

    def insert(self, key):
        new_node = RBTNode(key, None, True, None, None)
        self.insert_node(new_node)

    def insert_node(self, node):
        # Begin with normal BST insertion
        if self.root is None:
            # Special case for root
            self.root = node
        else:
            current_node = self.root
            while current_node is not None:
                if node.key < current_node.key:
                    if current_node.left is None:
                        current_node.set_child("left", node)
                        break
                    else:
                        current_node = current_node.left
                else:
                    if current_node.right is None:
                        current_node.set_child("right", node)
                        break
                    else:
                        current_node = current_node.right

        # Color the node red
        node.color = "red"

        # Balance
        self.insertion_balance(node)

    def insertion_balance(self, node):
        # If node is the tree's root, then color node black and return
        if node.parent is None:
            node.color = "black"
            return

        # If parent is black, then return without any alterations
        if node.parent.is_black():
            return

        # References to parent, grandparent, and uncle are needed for
remaining operations
        parent = node.parent
        grandparent = node.get_grandparent()
        uncle = node.get_uncle()

        # If parent and uncle are both red, then color parent and uncle
black, color grandparent
        # red, recursively balance  grandparent, then return
        if uncle is not None and uncle.is_red():
```

```
            parent.color = uncle.color = "black"
            grandparent.color = "red"
            self.insertion_balance(grandparent)
            return

        # If node is parent's right child and parent is grandparent's left
child, then rotate left
        # at parent, update node and parent to point to parent and
grandparent, respectively
        if node is parent.right and parent is grandparent.left:
            self.rotate_left(parent)
            node = parent
            parent = node.parent
        # Else if node is parent's left child and parent is grandparent's
right child, then rotate
        # right at parent, update node and parent to point to parent and
grandparent, respectively
        elif node is parent.left and parent is grandparent.right:
            self.rotate_right(parent)
            node = parent
            parent = node.parent

        # Color parent black and grandparent red
        parent.color = "black"
        grandparent.color = "red"

        # If node is parent's left child, then rotate right at grandparent,
otherwise rotate left
        # at grandparent
        if node is parent.left:
            self.rotate_right(grandparent)
        else:
            self.rotate_left(grandparent)

    def rotate_left(self, node):
        right_left_child = node.right.left
        if node.parent != None:
            node.parent.replace_child(node, node.right)
        else: # node is root
            self.root = node.right
            self.root.parent = None
        node.right.set_child("left", node)
        node.set_child("right", right_left_child)

    def rotate_right(self, node):
        left_right_child = node.left.right
        if node.parent != None:
            node.parent.replace_child(node, node.left)
        else: # node is root
            self.root = node.left
            self.root.parent = None
        node.left.set_child("right", node)
        node.set_child("left", left_right_child)
```


PARTICIPATION ACTIVITY 7.10.2: Rotations and insertion.

Assume `tree` represents the tree shown below.



1) After calling `tree.rotate_right(node19)`, the tree is _____.

- still a valid red-black tree
- invalid, but can be fixed by recoloring nodes 19 and 12
- invalid and cannot be fixed by exclusively recoloring nodes

2) Inserting _____ would require at least 1 rotation.

- 95
- 35
- 11

3) `tree.insert(25)` requires _____.

- 1 left rotation
- 1 right rotation
- no rotations

Removals

Part of red-black tree removal utilizes the standard BST removal, which is implemented in the RedBlackTree class _bst_remove() and _bst_remove_node() methods. The search() method implements a standard BST search for a key. The remove() method searches for a node based on a key and if found removes the node with remove_node().

The remove_node() method implements red-black tree removal. When removing a black node, the tree is restructured as needed via the prepare_for_removal() method. Methods try_case1() through try_case6() check for the 6 possible removal cases. The is_none_or_black() and is_not_none_and_red() methods assist with checking nodes in removal cases 5 and 6.

Figure 7.10.3: RedBlackTree class methods for removal.

©zyBooks 11/20/23 11:03 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
def _bst_remove(self, key):
    node = self.search(key)
    self._bst_remove_node(node)

def _bst_remove_node(self, node):
    if node is None:
        return

    # Case 1: Internal node with 2 children
    if node.left is not None and node.right is not None:
        # Find successor
        successor_node = node.right
        while successor_node.left is not None:
            successor_node = successor_node.left

        # Copy successor's key
        successor_key = successor_node.key

        # Recursively remove successor
        self._bst_remove_node(successor_node)

        # Set node's key to copied successor key
        node.key = successor_key

    # Case 2: Root node (with 1 or 0 children)
    elif node is self.root:
        if node.left is not None:
            self.root = node.left
        else:
            self.root = node.right

        # Make sure the new root, if not None, has parent set to
None
        if self.root is not None:
            self.root.parent = None

    # Case 3: Internal with left child only
    elif node.left is not None:
        node.parent.replace_child(node, node.left)

    # Case 4: Internal with right child OR leaf
    else:
        node.parent.replace_child(node, node.right)

def is_none_or_black(self, node):
    if node is None:
        return True
    return node.is_black()

def is_not_none_and_red(self, node):
    if node is None:
        return False
    return node.is_red()

def prepare_for_removal(self, node):
    if self.try_case1(node):
        return

    sibling = node.get_sibling()
    if self.try_case2(node, sibling):
```

```
            sibling = node.get_sibling()
        if self.try_case3(node, sibling):
            return
        if self.try_case4(node, sibling):
            return
        if self.try_case5(node, sibling):
            sibling = node.get_sibling()
        if self.try_case6(node, sibling):
            sibling = node.get_sibling()

        sibling.color = node.parent.color
        node.parent.color = "black"
        if node is node.parent.left:
            sibling.right.color = "black"
            self.rotate_left(node.parent)
        else:
            sibling.left.color = "black"
            self.rotate_right(node.parent)

    def remove(self, key):
        node = self.search(key)
        if node is not None:
            self.remove_node(node)
            return True
        return False

    def remove_node(self, node):
        if node.left is not None and node.right is not None:
            predecessor_node = node.get_predecessor()
            predecessor_key = predecessor_node.key
            self.remove_node(predecessor_node)
            node.key = predecessor_key
            return

        if node.is_black():
            self.prepare_for_removal(node)
        self._bst_remove(node.key)

        # One special case if the root was changed to red
        if self.root is not None and self.root.is_red():
            self.root.color = "black"

    def search(self, key):
        current_node = self.root
        while current_node is not None:
            # Return the node if the key matches.
            if current_node.key == key:
                return current_node

            # Navigate to the left if the search key is
            # less than the node's key.
            elif key < current_node.key:
                current_node = current_node.left

            # Navigate to the right if the search key is
            # greater than the node's key.
            else:
                current_node = current_node.right

        # The key was not found in the tree.
        return None
```

```
        return None

    def try_case1(self, node):
        if node.is_red() or node.parent is None:
            return True
        return False # node case 1

    def try_case2(self, node, sibling):
        if sibling.is_red():
            node.parent.color = "red"
            sibling.color = "black"
            if node is node.parent.left:
                self.rotate_left(node.parent)
            else:
                self.rotate_right(node.parent)
            return True
        return False # not case 2

    def try_case3(self, node, sibling):
        if node.parent.is_black() and
    sibling.are_both_children_black():
            sibling.color = "red"
            self.prepare_for_removal(node.parent)
            return True
        return False # not case 3

    def try_case4(self, node, sibling):
        if node.parent.is_red() and
    sibling.are_both_children_black():
            node.parent.color = "black"
            sibling.color = "red"
            return True
        return False # not case 4

    def try_case5(self, node, sibling):
        if self.is_not_none_and_red(sibling.left):
            if self.is_none_or_black(sibling.right):
                if node is node.parent.left:
                    sibling.color = "red"
                    sibling.left.color = "black"
                    self.rotate_right(sibling)
                    return True
        return False # not case 5


    def try_case6(self, node, sibling):
        if self.is_none_or_black(sibling.left):
            if self.is_not_none_and_red(sibling.right):
                if node is node.parent.right:
                    sibling.color = "red"
                    sibling.right.color = "black"
                    self.rotate_left(sibling)
                    return True
        return False # not case 6
```


©zyBooks 11/20/23 11:03 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

PARTICIPATION ACTIVITY 7.10.3: Red-black tree removal.

1) If prepare_for_removal() calls any try_case method that returns True, no additional try_case methods are ever called.

- True
- False

2) remove_node() starts by calling _bst_remove() to remove the node with the standard BST removal algorithm, then balances the tree.

- True
- False

3) Each removal try_case method except for try_case1() requires at least 1 rotation if the case is encountered.

- True
- False

4) remove_node() has worst case time complexity.

- True
- False

zyDE 7.10.1: Exploring the RedBlackTree class.

The following program inserts some nodes into a red-black tree and then removes some nodes. Try the following activities to see if you can predict how the tree will change:

- Sort the list of keys before inserting into the tree. Does the height of the tree change significantly?
- What key could be added to the end of the keys list that would cause at least one rotation when the node is inserted?
- Remove the root node.
- Remove a red leaf node.
- Remove a black leaf node.
- What removal cases are triggered when removing various nodes in the tree?

Current file: main.py Load default templat

```
1 from RedBlackTree import RedBlackTree
```

```
2
3 user_values = input('Enter insert values with spaces between: ')
4 print()
5
6 # Create an RedBlackTree object and add the values
7 tree = RedBlackTree()
8 for value in user_values.split():
9     tree.insert(int(value))
10
11 # Display the height after all inserts are complete.
12 print("Red-black tree with " + str(len(tree)) + " nodes has height "
13
14 # Read user input to get a list of values to remove
15 user_values = input('Enter remove values with spaces between: ')
```



```
15 42 77 18 90 21 36 19 89 47
15 47 40 19
```

Run

7.11 LAB: AVL tree Nth largest operation

Step 1: Inspect the BSTNode.py and BinarySearchTree.py files

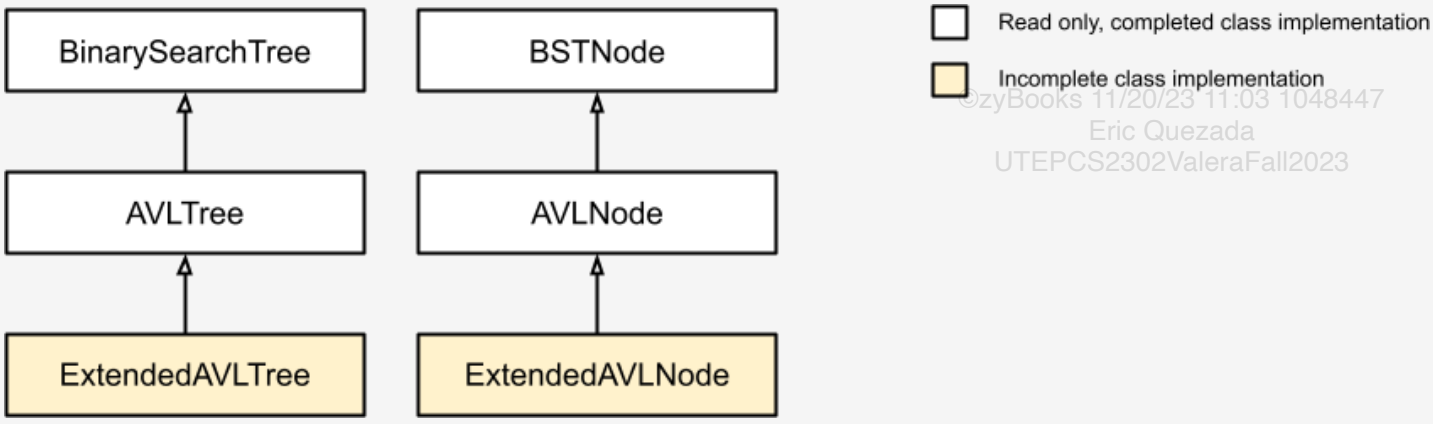
Inspect the BSTNode class declaration for a binary search tree node in BSTNode.py. Access BSTNode.py by clicking on the orange arrow next to main.py at the top of the coding window. The BSTNode class has attributes for the key, reference to the parent, reference to the left child, and reference to the right child. Accessor methods exist for each.

Inspect the BinarySearchTree class declaration for a binary search tree node in BinarySearchTree.py. The get_nth_key() method is the only abstract method that exists.

Step 2: Inspect other files related to the inheritance hierarchy

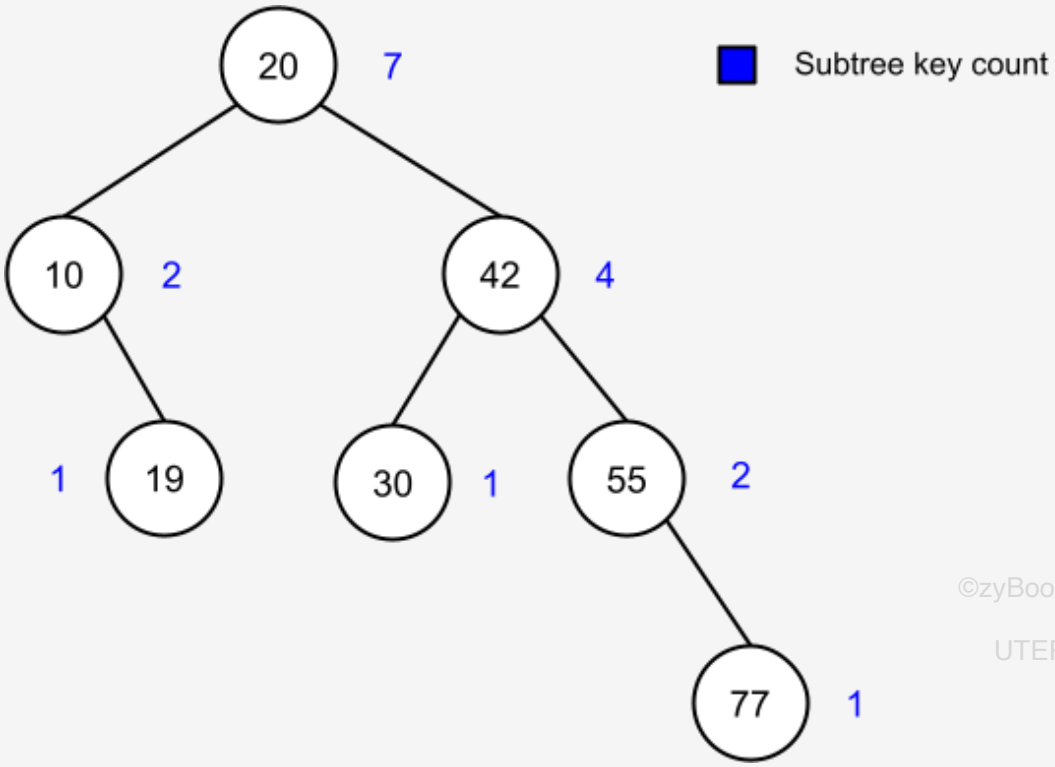
Classes AVLNode and AVLTree inherit from BSTNode and BinarySearchTree, respectively. Each class is implemented in a read only file.

Classes ExtendedAVLNode and ExtendedAVLTree are declared, but implementations are incomplete. Both classes must be implemented in this lab.



Step 3: Understand the purpose of the subtree_key_count variable

The ExtendedAVLNode class inherits from AVLNode and adds one integer attribute, subtree_key_count. Each node's subtree key count is the number of keys in the subtree rooted at that node. Ex:



ExtendedAVLNode's constructor and get_subtree_key_count() methods are already implemented and should not be changed. Additional methods are needed to ensure that subtree_key_count remains accurate.

Step 4: Implement ExtendedAVLTree and ExtendedAVLNode

Each node in an ExtendedAVLTree must have a correct subtree_key_count after an insertion or removal operation. Determine which methods in AVLTree and AVLNode must be overridden in ExtendedAVLTree and ExtendedAVLNode to keep each node's subtree_key_count correct. New methods can be added along with overridden methods, if desired.

Hint: Consider an update_subtree_key_count() method for the ExtendedAVLNode class. The method requires each child node's subtree_key_count to be correct, and updates the node's subtree_key_count appropriately. Overridden methods in both ExtendedAVLNode and ExtendedAVLTree can call a node's update_subtree_key_count() method as needed.

Once determinations are made, complete the implementation of both the ExtendedAVLTree and ExtendedAVLNode classes. Do not implement ExtendedAVLTree's get_nth_key() in this step. get_nth_key() requires correct subtree counts at each node.

Step 5: Run tests in develop mode and submit mode

TreeTestCommand is an abstract base class defined in TreeCommands.py. A TreeTestCommand object is an executable command that operates on a binary search tree. Classes inheriting from TreeTestCommand are also declared in TreeCommands.py:

- TreeInsertCommand inserts keys into the tree
- TreeRemoveCommand removes keys from the tree
- TreeVerifyKeysCommand verifies the tree's keys using an inorder traversal
- TreeVerifySubtreeCountsCommand verifies that each node in the tree has the expected subtree key count
- TreeGetNthCommand verifies that get_nth_key() returns an expected value

Code in main.py contains three automated test cases. Each test executes a list of TreeTestCommand objects in sequence. The third test includes TreeGetNthCommands and will not pass until the completion of Step 6. The first two tests should pass after completion of step 4.

Before proceeding to Step 6, run code in develop mode and ensure that the first two tests in main.py pass. Then submit code and ensure that the first two unit tests pass.

Step 6: Implement ExtendedAVLTree's get_nth_key() method (worst case O(log n))

get_nth_key() must return the tree's nth-largest key. The parameter **n** starts at 0 for the smallest key in the tree. Ex: Suppose a tree has keys:

```
10, 19, 20, 30, 42, 55, 77
```

Then `get_nth_key(0)` returns 10, `get_nth_key(1)` returns 19, ..., `get_nth_key(5)` returns 55, and `get_nth_key(6)` returns 77.

Determine an algorithm that uses the subtree key counts so that get_nth_key() operates in worst case O(log n) time.

LAB ACTIVITY | 7.11.1: LAB: AVL tree Nth largest operation | 0 / 10

Downloadable files

`main.py` , `BSTNode.py` , `AVLNode.py` , `ExtendedAVLNode.py` , `BinarySearchTree.py` , `AVLTree.py` , `ExtendedAVLTree.py` , `BSTNodeVisitor.py` , `BSTNodeListVisitor.py` , and `TreeCommands.py`

Download

File is marked as read only Current file: main.py

```
1 Loading latest submission...
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

7.12 LAB: Red-black tree Nth largest operation

This section's content is not available for print.