8.1 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.

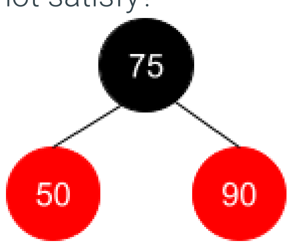
PARTICIPATION ACTIVITY | 8.1.1: Red-black tree rules.

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.

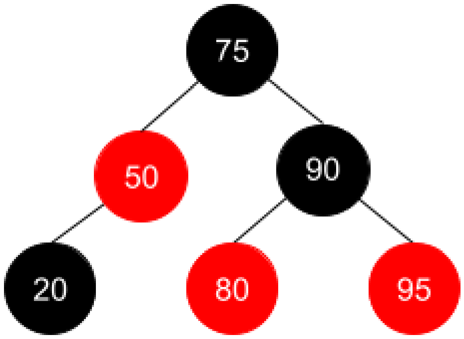
PARTICIPATION ACTIVITY | 8.1.2: Red-black tree rules.

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



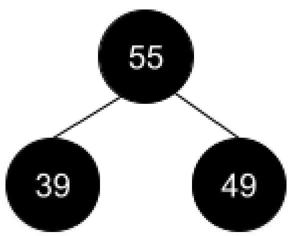
- Root node must be black.
- A red node's children cannot be red.
- All paths from a node to null leaf nodes must have the same number of black nodes.
- None.

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



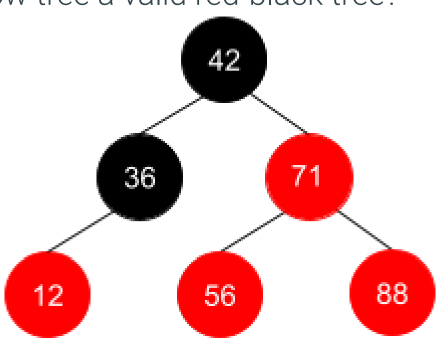
- Root node must be black.
- A red node's children cannot be red.
- Not all levels are full.
- All paths from a node to null leaf nodes must have the same number of black nodes.

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

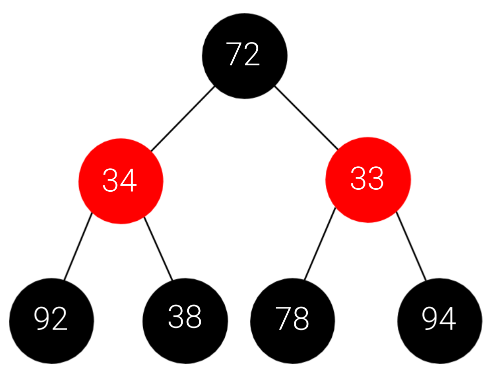
7) Any BST can be made a red-black tree by coloring all nodes black.

- True
- False

CHALLENGE ACTIVITY 8.1.1: Red-black tree: A balanced tree.

489394.3384924.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

8.2 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.

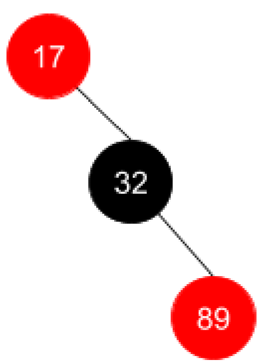
PARTICIPATION ACTIVITY | 8.2.1: A simple left rotation in a red-black tree.

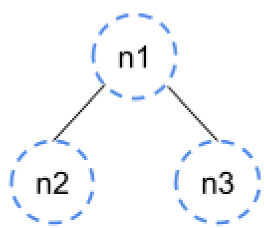
Animation captions:

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 | 8.2.2: 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.

17 89 32

n2

	n1
	n3

Reset

Left rotation algorithm

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

Figure 8.2.1: 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 8.2.2: 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 8.2.3: 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 8.2.3: RBTreeRotateLeft algorithm.

If unable to drag and drop, refresh the page.

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

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

Reset

Right rotation algorithm

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

PARTICIPATION ACTIVITY | 8.2.4: 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 | 8.2.5: 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.

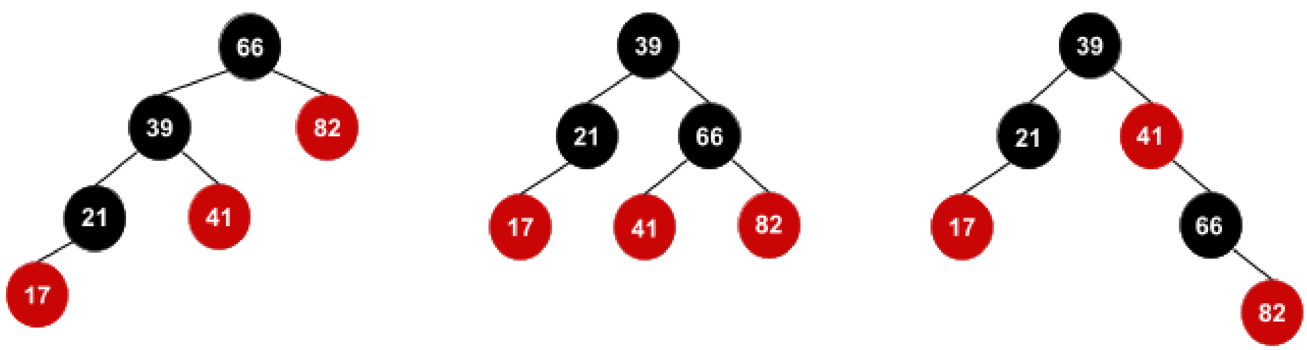
- True
- False

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

- True
- False

PARTICIPATION ACTIVITY 8.2.6: 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.

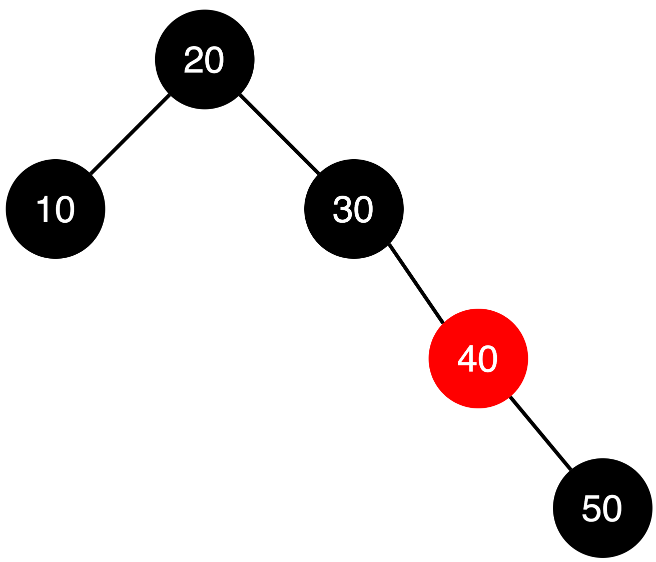
- True
- False

CHALLENGE ACTIVITY 8.2.1: Red-black tree: Rotations.

489394.3384924.qx3zqy7

Start

An invalid red-black tree is shown below.



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

1 2 3

Check Next

8.3 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 8.3.1: 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 8.3.2: 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 | 8.3.1: RBTreeBalance algorithm.

Animation content:

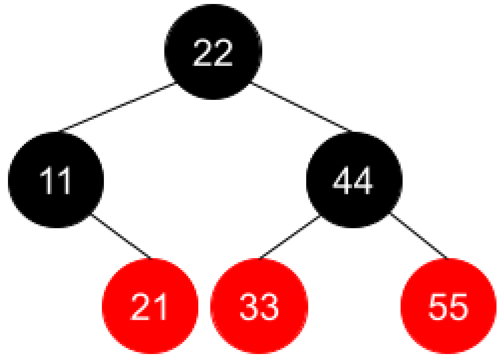
undefined

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 | 8.3.2: 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.

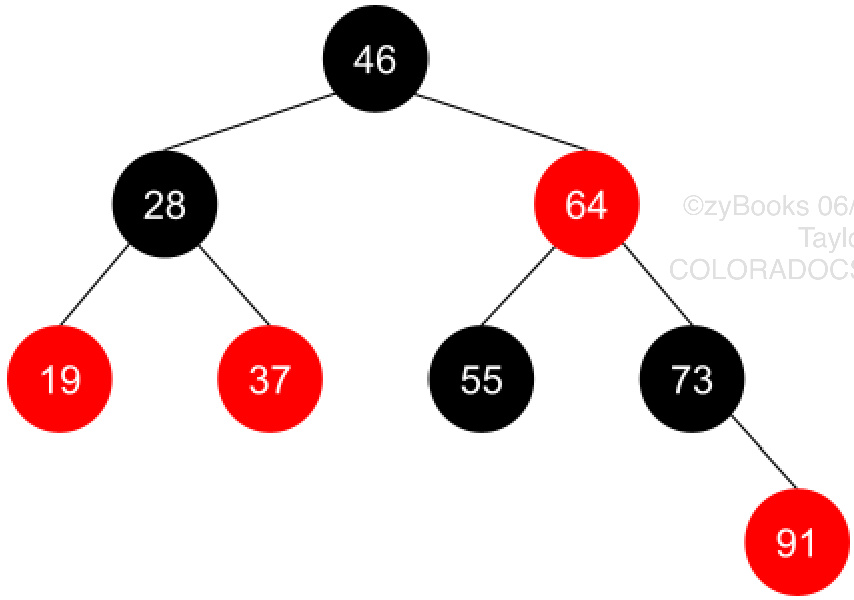
- True
- False

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 8.3.3: 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.

2 4 5 1 3 Never

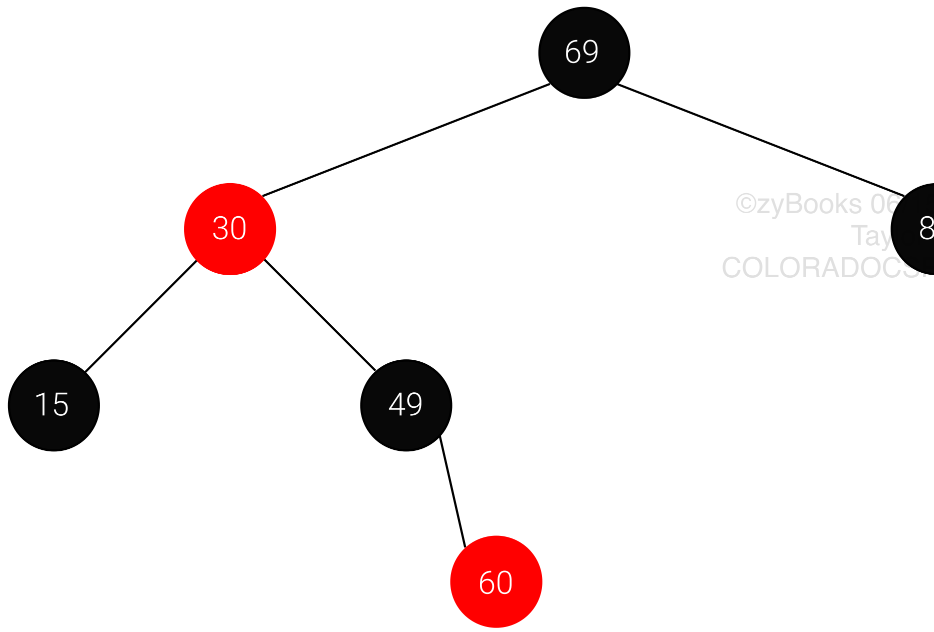
	Rotate left at node 73.
	Insert red node 82 as node 91's left child.
	Color grandparent node red.
	Color parent node black.
	Rotate right at node 91.
	Call RBTreeBalance recursively on node 73.

Reset

CHALLENGE ACTIVITY 8.3.1: Red-black tree: Insertion.

489394.3384924.qx3zqy7

Start



Node 11 is inserted. Prior to any rotations, what is node 11's _______?

Parent node: Ex: 10 or null

Grandparent node:

Uncle node:

Is a rotation required to complete the insertion?

1 2 3

Check Next

8.4 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 8.4.1: RBTreeRemove algorithm.

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


The RBTreeRemove algorithm consists of the following steps:

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

Figure 8.4.2: 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 8.4.3: RBTreeGetPredecessor utility function.

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

PARTICIPATION ACTIVITY | 8.4.1: 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 8.4.4: RBTreeGetSibling algorithm.

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


Figure 8.4.5: RBTreeIsNonNullAndRed algorithm.

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

Figure 8.4.6: RBTreeIsNullOrBlack algorithm.

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

Figure 8.4.7: 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 8.4.2: 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 8.4.8: 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 8.4.3: 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 8.4.1: 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	Color sibling red and call removal preparation function on parent.	No

	black.		
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 8.4.2: 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 | 8.4.4: 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 | 8.4.5: 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 | 8.4.6: Prepare-for-removal algorithm cases.

If unable to drag and drop, refresh the page.

RBTreeTryCase2 RBTreeTryCase4 RBTreeTryCase6 RBTreeTryCase5

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