Left Rotation Algorithm

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.
- RBTreeSetChild - utility function sets a node's left child
- RBTreeReplaceChild - utility function replaces a node's left or right child pointer with a new value

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
}
```

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
}
```

RBTreeRotateLeft function

- Performs a left rotation at the specified node by updating the right child's left child to point to the node
- Also, it updates 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 the node to the node's right child.
- If the node's parent is null, then the tree's root pointer is updated to point to the node's right child

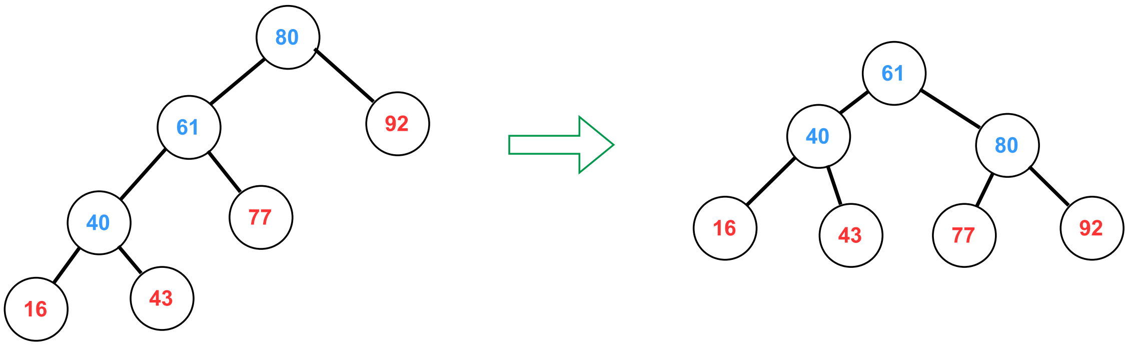
RBTreeRotateLeft function

```
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)
}
```

Red Black Balanced Tree Rules

- 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 node
- All paths from a node to any null lead descendant node must have the same number of black nodes

Right Rotation at 80



80 77

```
RBTreeRotateRight(tree, node) {
   leftRightChild = node -->left -->right
   if (node -->parent != null)
      RBTreeReplaceChild(node -->parent, node, node -->left)
   else { // node is root
      tree -->root = node -->left
      tree -->root -->parent = null
   }
   RBTreeSetChild(node -->left, "right", node)
   RBTreeSetChild(node, "left", leftRightChild)
}
```

61 61 null null

61 "right" 80

80 "left" 77

61 "right" 80

```
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
}
```

Makes 80 the right child of 61

80 "left" 77

```
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
}
```

Makes 77 the left child of 80

Red / Black Removal Operation

- Red-black remove key operation removes the key from the tree
- Tree is restructured to preserve red-black requirements
- BSTSearch() is called to find the node containing the key
- If node is found, RBTreeRemoveNode() is called to remove the node

RBTreeRemoveNode()

- 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.
- If the node is black, call RBTreePrepareForRemoval() to restructure the tree in preparation for the node's removal.
- Remove the node using the standard BST removal algorithm

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

```
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)
}
```

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

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 8.8.4: RBTreeGetSibling algorithm.

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

Feedback?

Figure 8.8.5: RBTreeIsNonNullAndRed algorithm.

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

Feedback?

Figure 8.8.6: RBTreeIsNullOrBlack algorithm.

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

Feedback?

Figure 8.8.7: RBTreeAreBothChildrenBlack algorithm.

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

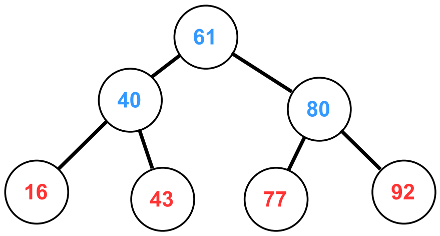
RBTreePrepareForRemoval()

- Preparation for removing a black node requires altering the number of black nodes along the path to preserve red-black properties

```
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)
   }
}
```



80

80

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

<table>
<tr><th>Case #</th><th>Code</th></tr>
<tr><td>1</td><td>RBTreeTryCase1(tree, node) {
   if (node→color == red || node→parent == null)
      return true
   else
      return false // not case 1
}</td></tr>
<tr><td>2</td><td>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
}</td></tr>
<tr><td>3</td><td>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
}</td></tr>
<tr><td>4</td><td>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
}</td></tr>
<tr><td>5</td><td>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
}</td></tr>
<tr><td>6</td><td>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</td></tr>
</table>

```
}
```