Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where $h$ is the height of the tree.

If the tree is balanced, $h = O(\log n)$ where $n$ is the number of elements.
Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where $h$ is the height of the tree.

If the tree is balanced, $h = O(\log n)$ where $n$ is the number of elements.

However, there is no guarantee that the tree is balanced.

The operations insert and delete can change the balance of the tree.
Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where $h$ is the height of the tree.

If the tree is balanced, $h = O(\log n)$ where $n$ is the number of elements.

However, there is no guarantee that the tree is balanced. The operations insert and delete can change the balance of the tree.

For an efficient representation we must implement insert and delete so that they preserve the balance. Not easy.

There are several ways to do it.
Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where $h$ is the height of the tree.

If the tree is balanced, $h = O(\log n)$ where $n$ is the number of elements.

However, there is no guarantee that the tree is balanced.

The operations insert and delete can change the balance of the tree.

For an efficient representation we must implement insert and delete so that they preserve the balance.

Not easy.

There are several ways to do it.

Red-Black Trees:

- Not perfect balance.
- Some paths may be twice as long as others.
- Still guarantees that the height is $O(\log n)$. 
Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, *only count black nodes*
- Keep the number of red nodes low: *no consecutive red nodes*
Red-Black Trees Definition

Idea:

- Color the nodes of a BST either Red or Black
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satisfying these properties:
Red-Black Trees Definition

Idea:

- Color the nodes of a BST either Red or Black
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satisfying these properties:

1. Every node contains an extra color value: Red or Black
   (basically a Boolean value)
Red-Black Trees Definition

Idea:

- Color the nodes of a BST either Red or Black
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satisfying these properties:

1. Every node contains an extra color value: Red or Black (basically a Boolean value)
2. The root (and leaves) are black
Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, **only count black nodes**
- Keep the number of red nodes low: **no consecutive red nodes**

A **Red-Black Tree** is a BST satisfying these properties:

1. Every node contains an extra **color value**: **Red** or **Black**
   (basically a Boolean value)
2. The root (and leaves) are black
3. The children of a red node are black
   (no consecutive red nodes)
Red-Black Trees Definition

Idea:

- Color the nodes of a BST either Red or Black
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satisfying these properties:

1. Every node contains an extra color value: Red or Black (basically a Boolean value)
2. The root (and leaves) are black
3. The children of a red node are black (no consecutive red nodes)
4. For each node, every path from it to a leaf has the same number of black nodes
Red-Black Trees Definition

Idea:

- Color the nodes of a BST either Red or Black
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satisfying these properties:

1. Every node contains an extra color value: Red or Black (basically a Boolean value)
2. The root (and leaves) are black
3. The children of a red node are black (no consecutive red nodes)
4. For each node, every path from it to a leaf has the same number of black nodes

Black-height of a node:
The number of black nodes in any path from the node to any leaf
Example of Red-Black Tree

Example (red nodes have double circles)
Example of Red-Black Tree

Example (red nodes have double circles)

All paths from root to a leaf contain two black nodes: black-height = 2

- Shortest paths: only black nodes, eg: 4, 3, ·
- Longest paths: alternating black and red, eg: 4, 11, 7, 9, ·
Example of Red-Black Tree

Example (red nodes have double circles)

All paths from root to a leaf contain two black nodes: \textit{black-height} = 2

- Shortest paths: only black nodes, eg: 4, 3, ·
- Longest paths: alternating black and red, eg: 4, 11, 7, 9, ·

Longest paths at most twice as long as shortest
Definition of the type of Red-Black trees in Haskell
Similar to Binary Search Trees, with extra field for color

```haskell
data Color = Red | Black
data RBTree = Leaf | Node Color RBTree Key RBTree
```
Definition of Data Type

Definition of the type of Red-Black trees in Haskell
Similar to Binary Search Trees, with extra field for color

```haskell
data Color = Red | Black

data RBTree = Leaf | Node Color RBTree Key RBTree
```

The term \((\text{Node} \ Red \ t_1 \ x \ t_2)\) represents the tree

![Diagram of a Red-Black tree](attachment:tree_diagram.png)
Definition of Data Type

Definition of the type of Red-Black trees in Haskell
Similar to Binary Search Trees, with extra field for color

```haskell
data Color = Red | Black

data RBTree = Leaf | Node Color RBTree Key RBTree
```

The term \((\text{Node Red } t_1 \ x \ t_2)\) represents the tree

![Diagram of a Red-Black tree](image)

Type definition does not guarantee elements are correct Red-Black trees
Definition of Data Type

Definition of the type of Red-Black trees in Haskell
Similar to Binary Search Trees, with extra field for color

```hs
data Color = Red | Black

data RBTree = Leaf | Node Color RBTree Key RBTree
```

The term `(Node Red t1 x t2)` represents the tree

![Tree Diagram]

Type definition does not guarantee elements are correct Red-Black trees

Ensure that the properties are satisfied when you create and modify trees:
The element must be a correct Binary Search Tree and
It must satisfy the extra Red-Black properties
If a R-B tree contains $n$ elements, then the maximum length of a path is $2 \log (n + 1)$.

Even if the tree is not perfectly balanced, its height is $O(\log n)$.

Therefore the running time for searching is $O(\log n)$. 
If a R-B tree contains $n$ elements, then the maximum length of a path is $2 \log(n + 1)$

Even if the tree is not perfectly balanced, its height is $O(\log n)$

Therefore the running time for searching is $O(\log n)$

But we must modify the insertion and deletion algorithms to preserve the R-B properties
If a R-B tree contains $n$ elements, then the maximum length of a path is $2 \log(n + 1)$.

Even if the tree is not perfectly balanced, its height is $O(\log n)$.

Therefore the running time for searching is $O(\log n)$.

But we must modify the insertion and deletion algorithms to preserve the R-B properties.

Idea:

- Insert and delete as for regular binary search trees.
- Always **color new nodes red** when you insert.
- **Rotate** and recolor to restore the properties.
If a R-B tree contains $n$ elements, then the maximum length of a path is $2 \log(n + 1)$.

Even if the tree is not perfectly balanced, its height is $O(\log n)$.

Therefore the running time for searching is $O(\log n)$.

But we must modify the insertion and deletion algorithms to preserve the R-B properties.

Idea:

- Insert and delete as for regular binary search trees.
- Always color new nodes red when you insert.
- Rotate and recolor to restore the properties.

We define an auxiliary function `balance` that rotates a tree when there are two consecutive red nodes in one of its children.
Assume that the top node is **black**, but there are two consecutive red nodes under it. There are four cases, according to the position of the red nodes.

**First Case:**

BST property: \( t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4 \)

**Rotate and Change Colors**
Assume that the top node is **black**, but there are two consecutive red nodes under it. There are four cases, according to the position of the red nodes.

**First Case:**

BST property: \( t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4 \)

The black-height of every node remains the same.

No consecutive red nodes any more.

(but there may be above if the parent is red)
Second Case:

BST property: $t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4$

Rotate and Change Colors
Second Case:

BST property: \( t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4 \)

If the consecutive red nodes are in the right child, rotate symmetrically in the other direction.
Haskell program that fixes one double occurrence of red nodes:
It receives the input tree already divided into
color, left-child, key, right-child

```
balance :: Color → RBTree → Key → RBTree → RBTree
balance Black (Node Red (Node Red t1 x1 t2) x2 t3) x3 t4
  = NodeRB Red (Node Black t1 x1 t2) x2 (Node Black t3 x3 t4)

... 

balance Black t1 x1 (Node Red t2 x2 (Node Red t3 x3 t4))
  = Node Red (Node Black t1 x1 t2) x2 (Node Black t3 x3 t4)
```
Insertion

Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply balance to fix consecutive reds
- At the end, if the root is red, make it black
Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply balance to fix consecutive reds
- At the end, if the root is red, make it black

\[
\begin{align*}
\text{ins} & : \text{Key} \rightarrow \text{RBTree} \rightarrow \text{RBTree} \\
\text{ins} \ a \ \text{Leaf} & = \ \text{Node Red Leaf a Leaf} \\
\text{ins} \ a \ \text{t@(Node color t1 x t2)} & \\
& \quad | \ a < x = \text{balance color (ins a t1) x t2} \\
& \quad | \ a > x = \text{balance color t1 x (ins a t2)} \\
& \quad | \ \text{otherwise} = \ t
\end{align*}
\]
Insertion

Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply balance to fix consecutive reds
- At the end, if the root is red, make it black

\[
\text{ins} :: \text{Key} \to \text{RBTree} \to \text{RBTree}
\]

\[
\text{ins} \ a \ \text{Leaf} = \text{Node Red Leaf a Leaf}
\]

\[
\text{ins} \ a \ t@(\text{Node color t1 x t2})
\]

\[
\quad | \ a < x = \text{balance color (ins a t1) x t2}
\quad | \ a > x = \text{balance color t1 x (ins a t2)}
\quad | \ \text{otherwise} = t
\]

The result satisfies most the R-B properties, **Except:**
Its root could be red (and might have a red child)
Insertion

Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply balance to fix consecutive reds
- At the end, if the root is red, make it black

\[
\text{ins} :: \text{Key} \rightarrow \text{RBTree} \rightarrow \text{RBTree}
\]

\[
\text{ins} a \ \text{Leaf} = \text{Node Red Leaf a Leaf}
\]

\[
\text{ins} a \ (\text{Node color t1 x t2})
\]

\[
| \ a < x = \text{balance color} \ (\text{ins} a \ t1) x t2 \\
| \ a > x = \text{balance color} \ t1 x \ (\text{ins} a \ t2) \\
| \ \text{otherwise} = t
\]

The result satisfies most the R-B properties, Except:
Its root could be red (and might have a red child) - just paint it black:

\[
\text{insert} :: \text{Key} \rightarrow \text{RBTree} \rightarrow \text{RBTree}
\]

\[
\text{insert} a \ \text{tree} = \text{blackRoot} \ (\text{ins} a \ \text{tree})
\]
Let’s say that a tree is weakly R-B if it satisfies all the R-B properties except that its root may be red and one of its children may also be red (so there could be two consecutive red nodes at the top).
Let’s say that a tree is weakly R-B if it satisfies all the R-B properties except that its root may be red and one of its children may also be red (so there could be two consecutive red nodes at the top).

Observation:

- If $t$ is a weakly R-B tree, then also $(\text{ins } a \ t)$ is a weakly R-B tree.
Let’s say that a tree is **weakly R-B** if it satisfies all the R-B properties except that its root may be red and one of its children may also be red (so there could be two consecutive red nodes at the top).

Observation:

- If $t$ is a weakly R-B tree, then also $(\text{ins } a \; t)$ is a weakly R-B tree
- If $t$ is a weakly R-B tree, then we can turn it into a fully R-B tree by painting its root black

This will increase the black-height by one, but since we do it at the root, all paths will increase their black-lengths equally.
The function $\texttt{balance}$ only rearranges the first two levels of the tree, so it runs in constant time.
The function `balance` only rearranges the first two levels of the tree, so it runs in constant time. So `insert` still runs in $O(h)$ time, where $h$ is the height of the tree.
The function `balance` only rearranges the first two levels of the tree, so it runs in constant time. So `insert` still runs in $O(h)$ time, where $h$ is the height of the tree.

The height of a R-B tree is $h = O(\log n)$.
The function `balance` only rearranges the first two levels of the tree, so it runs in constant time.

So the `insert` still runs in $O(h)$ time, where $h$ is the height of the tree.

The height of a R-B tree is $h = O(\log n)$.

So `insert` runs in $O(\log n)$ time.
Deletion

Deleting an element is a bit more complicated than inserting it. Deletion may cause a subtree to decrease its black-height. Then we must apply some rotations to rebalance it.
Deleting an element is a bit more complicated than inserting it. Deletion may cause a subtree to decrease its black-height. Then we must apply some rotations to rebalance it. For example, if we delete $x$ from the tree

```
          y
         /
        /  
       t_1  t_2
```

in the case that $x < y$, we delete it from $t_1$. This may cause the black-height of $t_1$ to decrease while the black-height of $t_2$ is unchanged.
Deleting an element is a bit more complicated than inserting it. Deletion may cause a subtree to decrease its black-height. Then we must apply some rotations to rebalance it.

For example, if we delete $x$ from the tree:

```
        y
       / \
      t1   t2
```

in the case that $x < y$, we delete it from $t_1$. This may cause the black-height of $t_1$ to decrease while the black-height of $t_2$ is unchanged.

Define rebalancing functions for when one child has a black-height larger by one than the other.
Delete: Simultaneously Defined Functions
Delete: Simultaneously Defined Functions

- **delete :: Key -> RBTree -> RBTree**

  (delete x t) looks for key x in the tree t; if it finds it, it deletes it and rebalances the tree so the R-B properties hold
Delete: Simultaneously Defined Functions

- **delete :: Key -> RBTree -> RBTree**
  (delete x t) looks for key x in the tree t; if it finds it, it deletes it and rebalances the tree so the R-B properties hold

- **del :: Key -> RBTree -> RBTree**
  Preliminary version of delete: the result satisfies the R-B properties except it may have two consecutive red nodes at the top
Delete: Simultaneously Defined Functions

- **delete :: Key -> RBTree -> RBTree**
  (delete \(x\) \(t\)) looks for key \(x\) in the tree \(t\); if it finds it, it deletes it and rebalances the tree so the R-B properties hold

- **del :: Key -> RBTree -> RBTree**
  Preliminary version of delete: the result satisfies the R-B properties except it may have two consecutive red nodes at the top

- **delL :: Key -> RBTree -> RBTree**
  Deletes an element from the left child
Delete: Simultaneously Defined Functions

- **delete**: Key -> RBTree -> RBTree
  
  (delete x t) looks for key x in the tree t; if it finds it, it deletes it and rebalances the tree so the R-B properties hold.

- **del**: Key -> RBTree -> RBTree
  
  Preliminary version of delete: the result satisfies the R-B properties except it may have two consecutive red nodes at the top.

- **delL**: Key -> RBTree -> RBTree
  
  Deletes an element from the left child.

- **balL**: Key -> RBTree -> RBTree
  
  Rebalances a tree when the black-height of the left child is one shorter than the right.
Delete: Simultaneously Defined Functions

- **delete :: Key -> RBTree -> RBTree**
  (delete x t) looks for key x in the tree t; if it finds it, it deletes it and rebalances the tree so the R-B properties hold

- **del :: Key -> RBTree -> RBTree**
  Preliminary version of delete: the result satisfies the R-B properties except it may have two consecutive red nodes at the top

- **delL :: Key -> RBTree -> RBTree**
  Deletes an element from the left child

- **balL :: Key -> RBTree -> RBTree**
  Rebalances a tree when the black-height of the left child is one shorter than the right

- **delR, balR :: Key -> RBTree -> RBTree**
  Like delL and balL, but on the right
Delete: Simultaneously Defined Functions

- **delete :: Key -> RBTree -> RBTree**
  (delete x t) looks for key x in the tree t; if it finds it, it deletes it and rebalances the tree so the R-B properties hold

- **del :: Key -> RBTree -> RBTree**
  Preliminary version of delete: the result satisfies the R-B properties except it may have two consecutive red nodes at the top

- **delL :: Key -> RBTree -> RBTree**
  Deletes an element from the left child

- **balL :: Key -> RBTree -> RBTree**
  Rebalances a tree when the black-height of the left child is one shorter than the right

- **delR, balR :: Key -> RBTree -> RBTree**
  Like delL and balL, but on the right

- **fuse :: RBTree -> RBTree -> RBTree**
  merges two trees \( t_1 \) and \( t_2 \) when all elements of \( t_1 \) are smaller than all elements of \( t_2 \)
Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child.

There can be three cases (color of the root irrelevant or obvious).
Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child.

There can be three cases (color of the root irrelevant or obvious):

First Case (left child has a red node):

\[ y \]

\[ x \]

\[ t_1 \]

\[ t_2 \]

\[ t_3 \]

\[ y \text{ must be black} \]
Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child.

There can be three cases (color of the root irrelevant or obvious).

First Case (left child has a red node):

\[ x \]
\[ t_1 \]
\[ t_2 \]
\[ t_3 \]

\[ y \]

\( y \) must be black. We swap the colors of \( x \) and \( y \).
Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child.

There can be three cases (color of the root irrelevant or obvious).

First Case (left child has a red node):

\[ y \]
\[ x \]
\[ t_1 \]
\[ t_2 \]
\[ t_3 \]

\( y \) must be black. We swap the colors of \( x \) and \( y \).

The black-height of the left child increases by 1, the black-height of the right child is unchanged.

(There could now be two red nodes at the top)
Second Case (left child black or leaf, right black):

![Diagram of a tree structure](image)
Balancing Left II

Second Case (left child black or leaf, right black):

Just repaint $z$ red
Second Case (left child black or leaf, right black):

Just repaint $z$ red

We decreased the black-height of the right child
Balancing Left II

Second Case (left child black or leaf, right black):

![Tree Diagram]

- Just repaint $z$ red
- We decreased the black-height of the right child
- But we may have created consecutive red nodes on the right
Second Case (left child black or leaf, right black):

Just repaint $z$ red

We decreased the black-height of the right child

But we may have created consecutive red nodes on the right

Apply `balance` to fix this problem
Third Case (left child black or leaf, right red):

There must be at least a black node under $z$ for the right child to have higher black-height

$t_4$ must have a black top node
Balancing Left III

Third Case (left child black or leaf, right red):

There must be at least a black node under $z$ for the right child to have higher black-height
$t_4$ must have a black top node

We might have created consecutive red nodes in the right child
Apply balance to the right child
If we put the three cases together we obtain the function to rebalance when the left child has black-height smaller by 1.
If we put the three cases together we obtain the function to rebalance when the left child has black-height smaller by 1.

The input tree is given in its three components.
No need to specify the color of the root.
If we put the three cases together we obtain the function
to rebalance when the left child has black-height smaller by 1

The input tree is given in its three components
No need to specify the color of the root

\[
\begin{align*}
\text{balL} & : \text{RBTree} \to \text{Key} \to \text{RBTree} \to \text{RBTree} \\
\text{balL} \ (\text{Node Red } t1 \ x \ t2) \ y \ t3 \\
& = \text{Node Red } (\text{Node Black } t1 \ x \ t2) \ y \ t3 \\
\text{balL} \ t1 \ y \ (\text{Node Black } t2 \ z \ t3) \\
& = \text{balance Black } t1 \ y \ (\text{Node Red } t2 \ z \ t3) \\
\text{balL} \ t1 \ y \ (\text{Node Red } (\text{Node Black } t2 \ u \ t3) \ z \ t4) \\
& = \text{Node Red } (\text{Node Black } t1 \ y \ t2) \\
& \quad \quad \text{u} \\
& \quad \quad \quad (\text{balance Black } t3 \ z \ (\text{redRoot } t4))
\end{align*}
\]
Delete Left

Deleting a key from the left child (when $x < y$)
Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black
the deletion will decrease the black height
so we must rebalance with balL.
Delete Left

Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black
the deletion will decrease the black height
so we must rebalance with balL.

Otherwise (top node red)
the black-height stays the same and we don’t need to rebalance.
Delete Left

Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black,
the deletion will decrease the black height
so we must **rebalance with balL**.

Otherwise (top node red)
the black-height stays the same and we don’t need to rebalance

\[
\text{delL} :: \text{Key} \rightarrow \text{RBTree} \rightarrow \text{Key} \rightarrow \text{RBTree} \rightarrow \text{RBTree}
\]

\[
\text{delL} \ x \ t1 \ y \ t2 =
\begin{cases}
\text{if (color t1) } = \text{ Black}
\text{ then balL (del x t1) y t2}
\text{else NodeRB Red (del x t1) y t2}
\end{cases}
\]
Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black, the deletion will decrease the black height so we must **rebalance with balL**.

Otherwise (top node red), the black-height stays the same and we don’t need to rebalance.

```haskell
delL :: Key -> RBTree -> Key -> RBTree -> RBTree
delL x t1 y t2 =
  if (color t1) == Black
  then balL (del x t1) y t2
  else NodeRB Red (del x t1) y t2
```

Define similar functions **balR** and **delR** to rebalance and delete on the right.
In the case when $x = y$, we must delete the root of the tree.

If we delete $x$ from

We’re left with the orphan trees $t_1$ and $t_2$.
We must put them back together into a single tree.
In the case when $x = y$, we must delete the root of the tree

If we delete $x$ from

We're left with the orphan trees $t_1$ and $t_2$

We must put them back together into a single tree

The strategy that we used with Binary Search Trees of replacing the deleted node with the minimum of the right child doesn't work any more, because it may disrupt the R-B properties
In the case when \( x = y \), we must delete the root of the tree.

If we delete \( x \) from

We’re left with the orphan trees \( t_1 \) and \( t_2 \)
We must put them back together into a single tree.

The strategy that we used with Binary Search Trees
of replacing the deleted node with the minimum of the right child
doesn’t work any more, because it may disrupt the R-B properties.

We must come up with a cleverer way of fusing \( t_1 \) and \( t_2 \)

\[
\text{fuse} :: \text{RBTree} \rightarrow \text{RBTree} \rightarrow \text{RBTree}
\]

We know that all elements of \( t_1 \) are smaller than all elements of \( t_2 \).
If the two trees have top nodes of different color

\[ t_1 = \quad t_2 = \]

We can choose the red one as new top node
Fuse: Different Color

If the two trees have top nodes of different color

![Diagram showing two trees with a red node labeled as the new top node]

We can choose the red one as new top node
If the two trees have top nodes of different color

We can choose the red one as new top node

Similarly when the first is red and the second is black
Fuse: Both Red

If both trees have a red top node

\[ t_1 = \begin{array}{c} x \\ t_3 \\ t_4 \end{array} \quad t_2 = \begin{array}{c} y \\ t_5 \\ t_6 \end{array} \]
Fuse: Both Red

If both trees have a red top node

\[ t_1 = x \]

\[ t_2 = y \]

First we recursively fuse the middle subtrees: \( s = \text{fuse} \ t_4 \ t_5 \)
If both trees have a red top node

\[ t_1 = \begin{array}{c} x \\ \downarrow \\ t_3 \end{array} \quad t_2 = \begin{array}{c} y \\ \downarrow \\ t_5 \end{array} \quad \text{fuse } t_4 t_5 = \begin{array}{c} \bullet \\ \downarrow \\ s \end{array} \]

First we recursively fuse the *middle subtrees*: \( s = \text{fuse } t_4 t_5 \)

If \( s \) has a black top node,
Fuse: Both Red

If both trees have a red top node

First we recursively fuse the middle subtrees: $s = \text{fuse } t_4 t_5$

If $s$ has a black top node, we put it under $y$
Fuse: Both Red

If both trees have a red top node

\[ t_1 = \text{x} \quad \text{middle subtrees: } s = \text{fuse } t_4 t_5 \]

First we recursively fuse the middle subtrees: \( s = \text{fuse } t_4 t_5 \)

If \( s \) has a red top node,
Fuse: Both Red

If both trees have a red top node

First we recursively fuse the *middle subtrees*: \( s = \text{fuse } t_4 t_5 \)

If \( s \) has a red top node, we use its node as new root

There are double red nodes on both sides, but
the top node will be recolored black either by balL or balR or delete,
according to where we deleted: left, right, or root
Fuse: Both Black

If both trees have a black top node

\[ t_1 = x \]
\[ t_2 = y \]

\[ t_3 \]
\[ t_4 \]

\[ t_5 \]
\[ t_6 \]
If both trees have a black top node

\[ t_1 = \begin{array}{c} x \\ \text{\quad} t_3 \\ \text{\quad} t_4 \end{array} \quad \text{and} \quad t_2 = \begin{array}{c} y \\ \text{\quad} t_5 \\ \text{\quad} t_6 \end{array} \]

Again we recursively fuse the middle subtrees: \( s = \text{fuse} t_4 t_5 \)
Fuse: Both Black

If both trees have a black top node

\[ t_1 = x \]
\[ t_2 = y \]
\[ \text{fuse } t_4 \ t_5 = s \]

Again we recursively fuse the middle subtrees: \( s = \text{fuse } t_4 \ t_5 \)

If \( s \) has a black top node,
If both trees have a black top node

Again we recursively fuse the middle subtrees: \( s = \text{fuse } t_4 \ t_5 \)

If \( s \) has a black top node, we put it under \( y \)
But this time the right subtree has increased black-height
We must apply \texttt{balL}
If both trees have a black top node

\[ t_1 = x \quad t_2 = y \quad s = z \]

Again we recursively fuse the middle subtrees: \( s = \text{fuse} \ t_4 \ t_5 \)

If \( s \) has a red top node,
Fuse: Both Black

If both trees have a black top node

Again we recursively fuse the middle subtrees: $s = \text{fuse } t_4 \ t_5$

If $s$ has a red top node, we use it as new root
Having defined all the auxiliary functions, we can now simply implement the main delete function:

```haskell
delete :: Key \rightarrow RBTree \rightarrow RBTree
delete x t = blackRoot (del x t)

del :: Key \rightarrow RBTree \rightarrow RBTree
del x LeafRB = LeafRB
del x (NodeRB _ t1 y t2)
  | x<y = delL x t1 y t2       -- delete from left child
  | x>y = delR x t1 y t2       -- delete from right child
  | otherwise = fuse t1 t2     -- delete root, fuse children
```