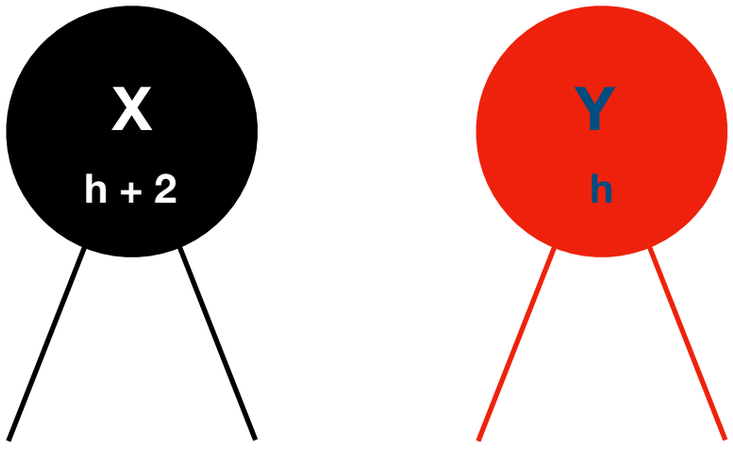


Figure 1.1: <b> and <r> nodes showing name and height

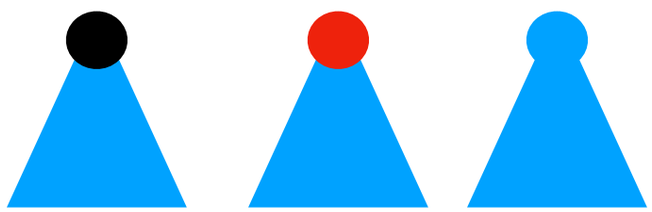


Figure 1.2: sub-trees with the colour of their root-node, if known.

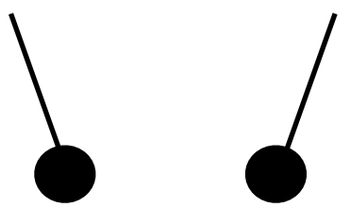


Figure 1.3: <e> nodes

Red-Black Trees

Red-Black Trees are binary search trees with particular characteristics that assist them in remaining balanced (in the sense that all paths to leaves are of roughly equal length).

The following are ***invariant*** properties designed to achieve this:

- ***Height-balance*** means that every vertical path has the same number of black (<b>) nodes giving all paths an equal basic length.
- ***Red-balance*** means that no vertical path has two adjacent red (<r>) nodes so that path lengths may differ but never exceed twice the <b> height. Maximum height is therefore $2log_2(n-1)$ for a tree containing *n* entries
- The root node of the entire tree is always black <b>.
- Empty nodes (<e>) are always counted as black.

A tree is a *recursive* structure in that every tree is composed of sub-trees and every sub-tree is a tree which obeys all the invariant rules (except that a sub-tree may have a <r> root). Key tree-maintenance operations are **insert** and **delete**. During these operations the invariants may be disturbed but they must be restored before the operation is completed.

Insertion & Deletion

The two key processes in maintaining a tree are ***insertion*** and ***deletion***. Both involve a recursive trip down a vertical path of the tree and a return along the same path to the root. The path downwards is determined by the *order-rule** which dictates the order in which elements are stored so that the appropriate location to be operated on can be found. The path upwards is '*remembered*' by returning along the downwards function calls.

On the downward path, any of four conditions may encountered:

- ***the sub-tree is empty***: we have reached a leaf so, if we're inserting, this where to do it and, if we're deleting, the element sought must not be in the tree.
- ***the value at the root of the current sub-tree is …***
  - ***equal to** the value we seek***: if we're inserting, this value is already in the tree so we cannot add it*; if we're deleting, this is the element to delete
  - ***greater than** the value we seek***: we need to descend further following the path on the left
  - ***less than** the value we seek***: we need to descend further following the path on the right.

On the upward path, we can rely on the well-formedness of surrounding sub-trees when we're repairing any imbalance.

Note: We'll assume the rule is ascending order but much more is possible. Operations analagous to =,< and > are defined.
Red-Black trees do not usually support duplicate element and this is assumed here (though my implementation actually does support them)

	Balance
Height	✓
Colour	✗

Insertion

Insertion is a relatively straight-forward operation:

- find the insertion point by following the order-rule recursively
- replace the selected leaf (always an <e>) with a new sub-tree consisting of the value to be inserted and two empty sub-trees and colour it <r> so that height-balance is undisturbed
- ascend, correcting the red-balance
- on arrival at the root, paint it black since red-balance correction may have propagated <r> up to the root

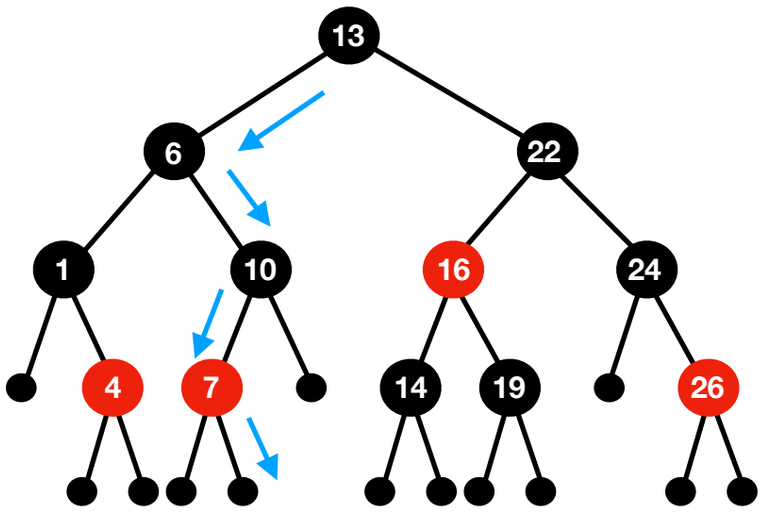


Figure 2.1: locate insertion point for a <9> …

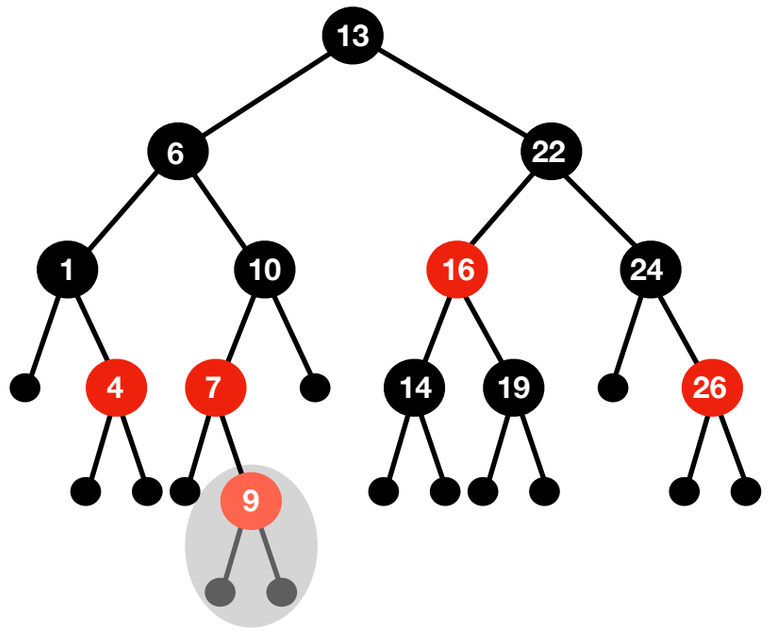


Figure 2.2: … insert new sub-tree <9> …

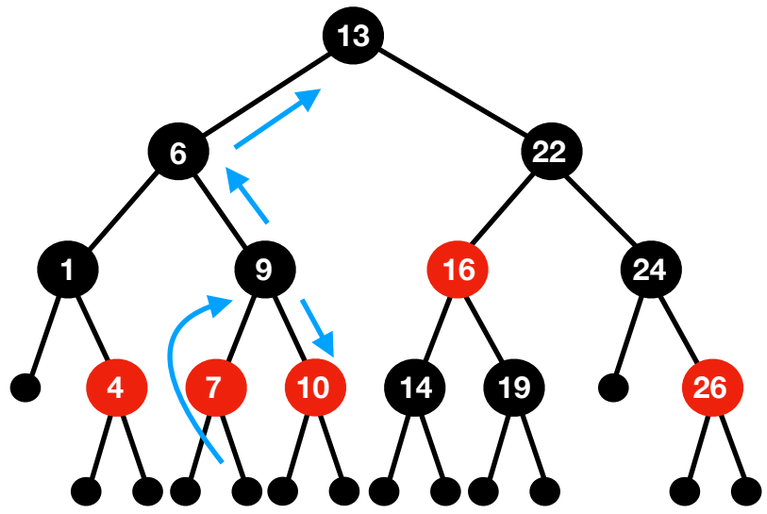


Figure 2.3: … and rebalance (to be described next)

Insertion

descend to a leaf …. insert …. ascend to the root, rebalancing

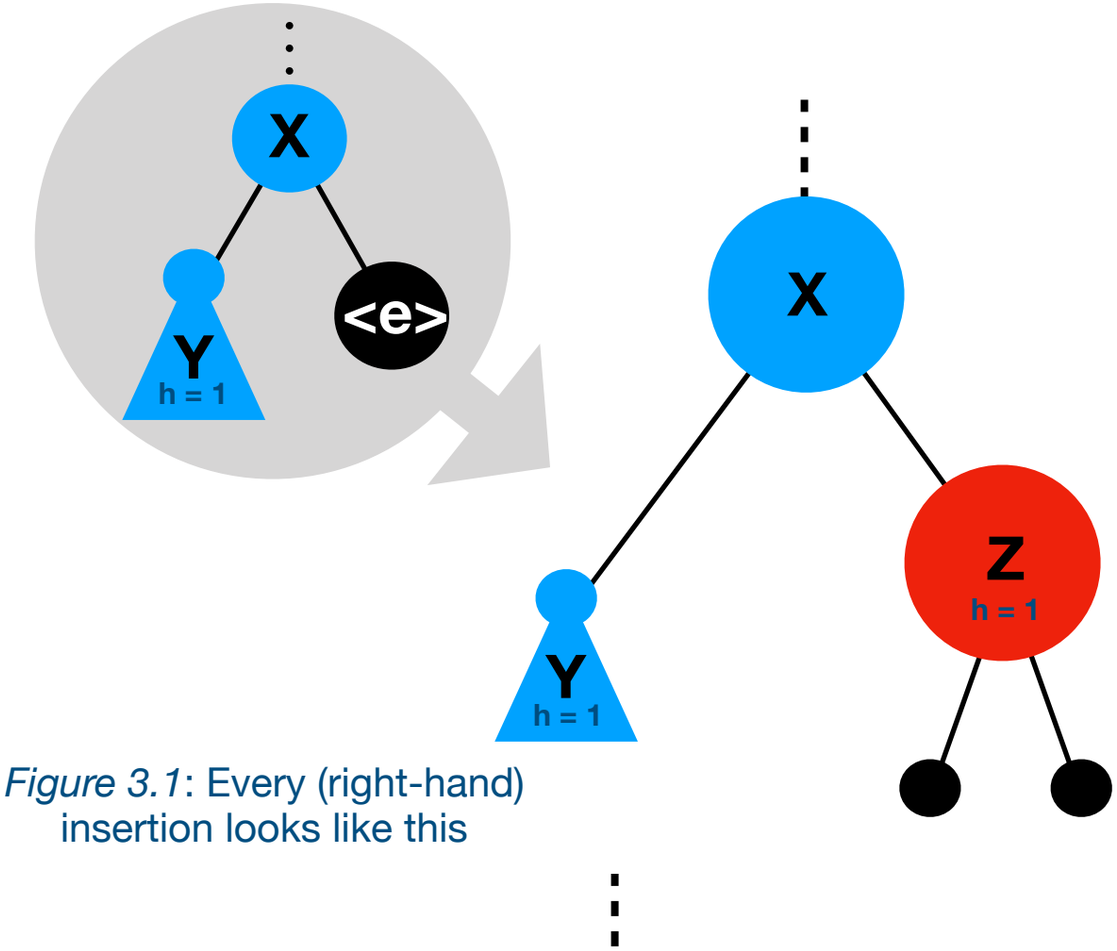


Figure 3.1: Every (right-hand) insertion looks like this

How Insertion Disrupts Red-Balance*

We're inserting **Z**. We arrived at this <e> child of **X** because **Z** > **X,** (**Z** < **X**'s parent) and $\mathbf{X}_R$ is <e>.

Insertion preserves *height-balance* in the sub-tree since, regardless of the colours of **X** and **Y** (which latter may even be an <e> and anyhow must have had a height of 1 to match the <e> which **Z** replaces) we've just added <r> to the path.

However, red-balance *may* have been disturbed since **X** may be <r> too.

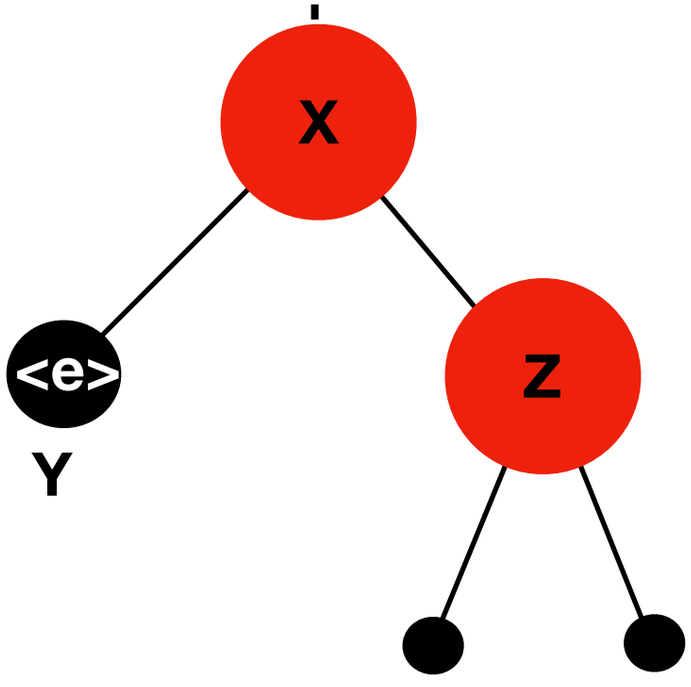


Figure 3.2: red-unbalanced case

Figure 3.2 shows the unbalanced version. When **X** is <r>, (and so **Y** must have been <e> since its height was no more than 1) this sub-tree needs *red-balancing.* Either this unbalance will be noticed as we ascend the tree and **X**'s parent sees it has a child and grandchild both <r> or (if **X** has no parent) when we repaint the root (**X**) the problem will be removed. We will look at red-balancing next.

Note: The diagrams show insertion in a right-hand sub-tree but, naturally, a symmetrical left-hand alternative exists too. It is described simply by exchanging the **Y** and **Z** sub-trees.

Red-rebalancing a sub-tree

When we encounter a red-unbalance, it is because we notice that a node has a <r> child which in turn has a <r> grandchild. There are only 4 cases:

		Child <r>	
		Left	**Right**
Grand-child <r>	**Left**	fig. 4.a	fig. 4.b
	Right	fig. 4.d	fig. 4.c

Each of the four cases has been labeled such that:

A < **X** < **B** < **Y** < **C** < **Z** < **D**

We can rely on the fact that the child and grandchild sub-trees are well-formed (we're working upwards) and this is reflected in the colouring shown. This implies that each of **A**, **B**, **C** and **D** must have the same height.

All four cases rebalance to the same sub-tree (fig. 4), leaving height unaffected. In each case, we replace the old sub-tree by this sub-tree recoloured as shown.

	Balance
Height	✓
Colour	✓

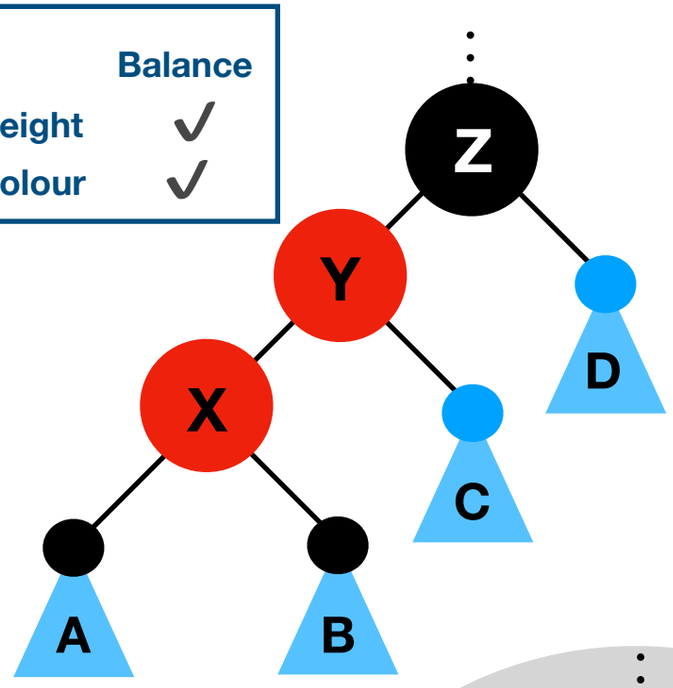


Figure 4.a

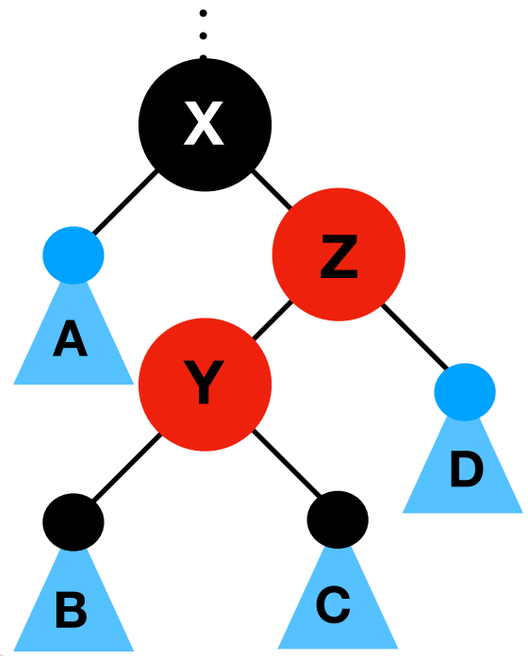


Figure 4.b

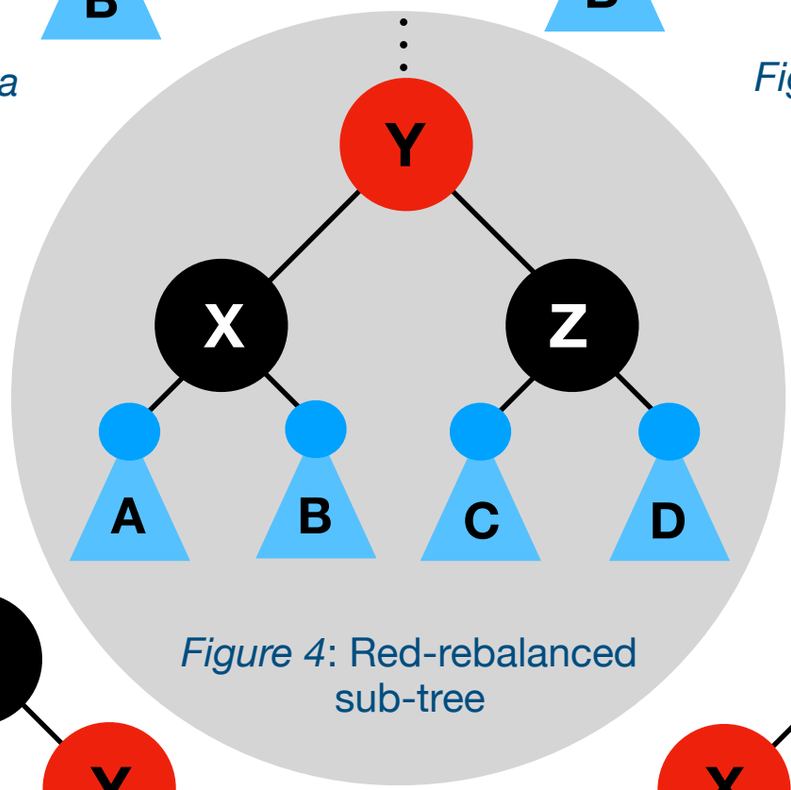


Figure 4: Red-rebalanced sub-tree

Note:
Later (figs.9), we will see that we never make a sequence of three <r> nodes so the top node is necessarily <b>.

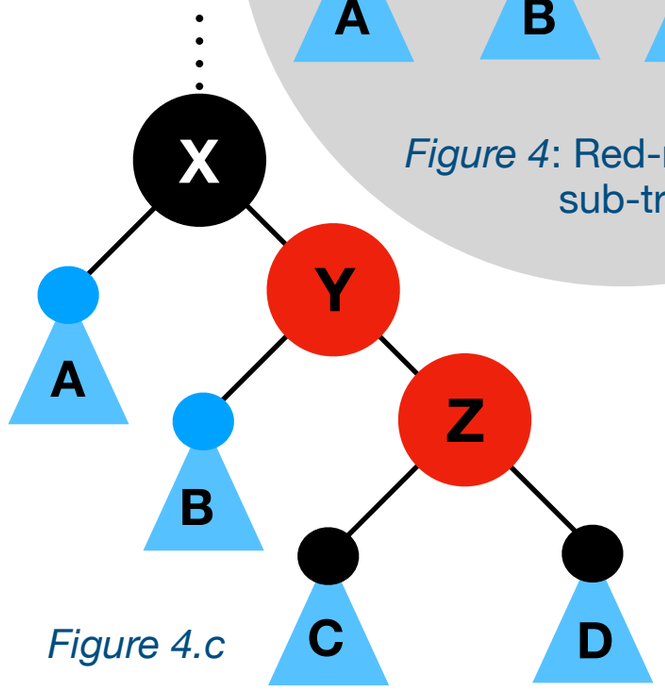


Figure 4.c

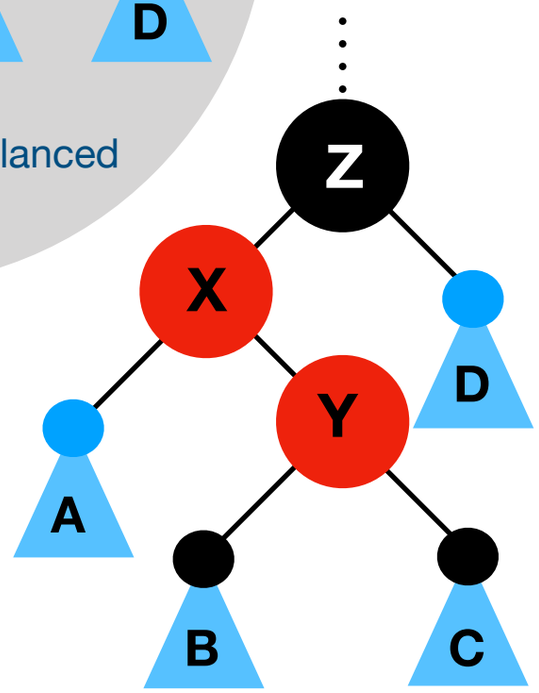


Figure 4.d

Deletion (and Fusion)

	Balance
Height	✘
Colour	✘

Deletion is considerably more complex than insertion:

- find the deletion point by following the order-rule recursively
- ***replace*** the sub-tree headed by the value to be deleted with a new sub-tree formed by ***fusing*** its two sub-trees into a single sub-tree. Unless one or both of these are <e>, this process involves further fusing at each subsequent level until a leaf is reached
- fusion will provide a sub-tree with the same height as its components, but if the deleted item was <b>, replacement will mean that this sub-tree is shorter than its sibling (height-imbalance). Fusion may also disturb red-balance
- ascend, correcting height- and red-balance
- on arrival at the root, paint it black since red-balance correction may have propagated <r> up to the root

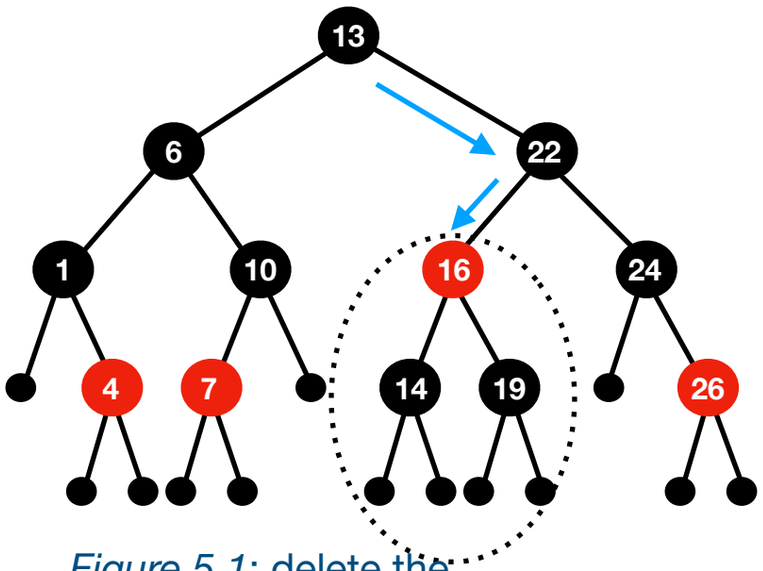


Figure 5.1: delete the <16> sub-tree …

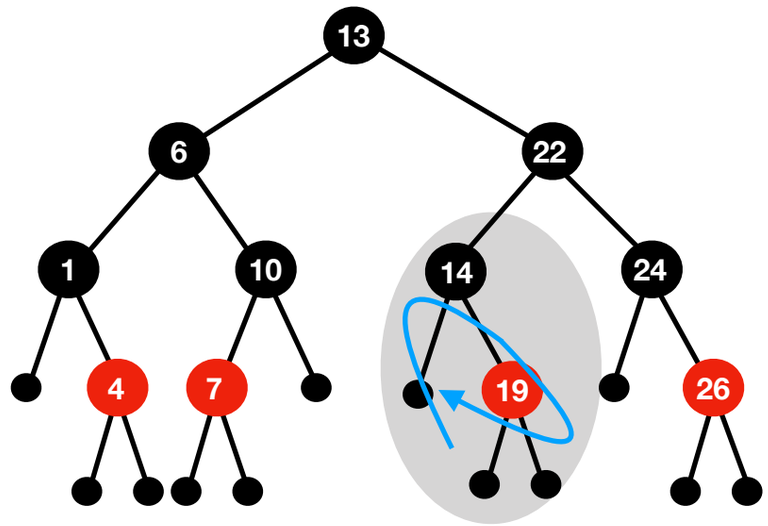


Figure 5.2: … replace with fused sub-trees …

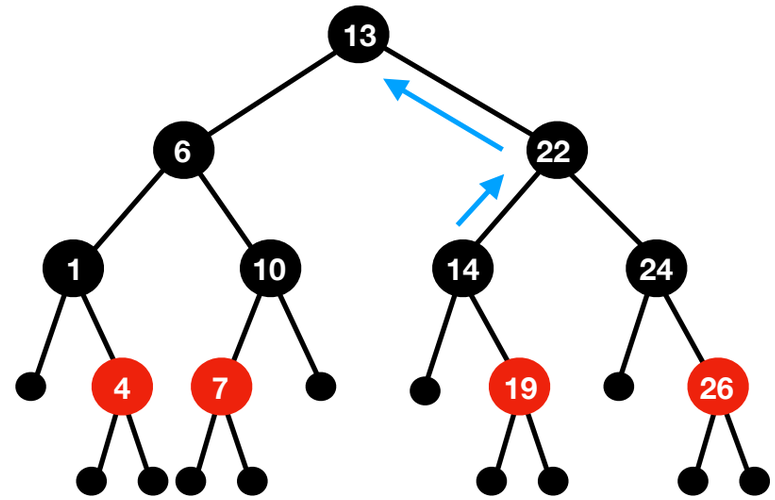


Figure 5.3: … and rebalance (to be described next)

Deletion

descend to item …. recursive fusion …. ascend to the root, rebalancing

Fusion

recursive descent to leaf …. ascend to deletion point

Fusion

Fusion is a process which is called to replace the deleted item with a sub-tree composed of its descendants. Since the descendants each have up to two components, the fusion needs to be invoked recursively down through the tree only ending at a leaf when an attempt is made to fuse a <e>. Only the first fusion in the chain *replaces* a node, the others *rearrange* nodes.

We will see that each individual fusion is height-balanced locally but the top fusion *may* propagate height-unbalance up the tree. During the return up the tree to the point of the deletion, we will apply red-balancing.

Fusion has four cases which depend on the colours of the roots of **Y** and **Z**:

- one or both <e>
- one <r>, one <b>
- both <r>
- both <b>

We will look at each in turn.

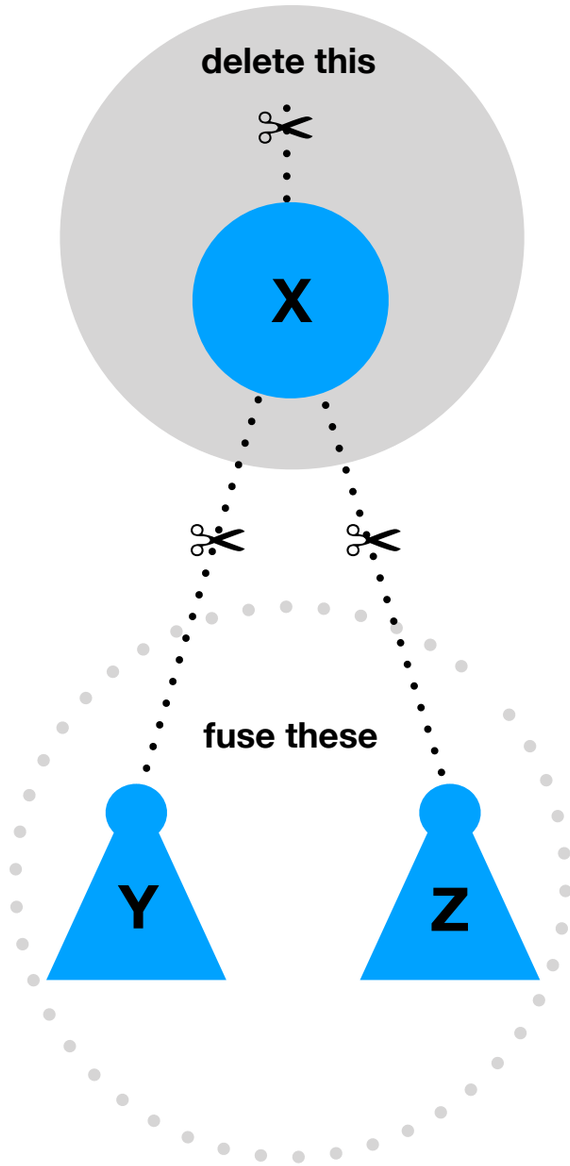


Figure 6: replace **X**, with fusion of **Y** and **Z**

	Balance
Height	✓
Colour	✓

Fusion: One or more <e>

A fusion involving one empty node (fig. 7.1) always returns the longer sub-tree, which is necessarily <r> and necessarily of height 1. (Note that red-balance is unaffected too since **X**'s parent is necessarily <b>).

A fusion of two empty nodes (fig. 7.2) produces an empty node. Again, height-balance and red-balance are preserved.

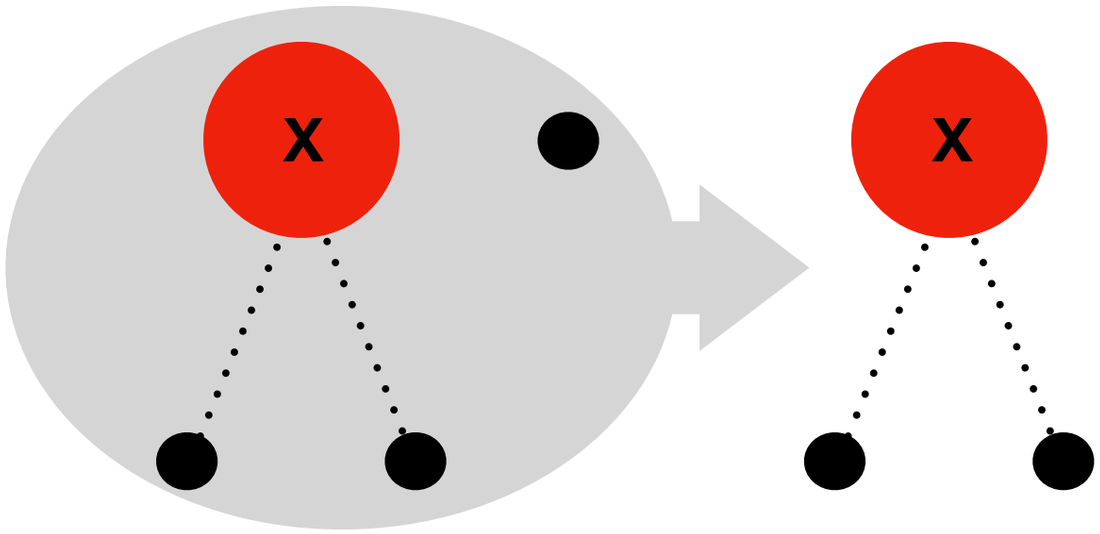


Figure 7.1: fuse **X** (must be <r>) with <e>, height preserved

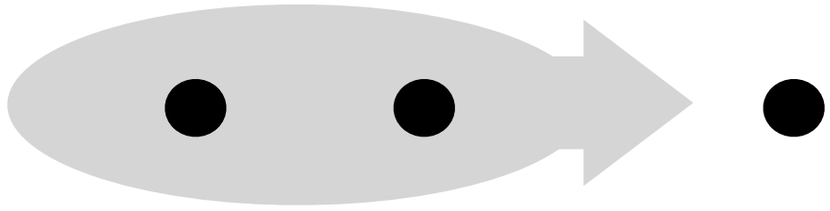


Figure 7.2: fuse <e> with <e>, height preserved

	Balance
Height	✓
Colour	✗

Fusion: One <r>, one <b>

Y, $\mathbf{Z_L}$ and $\mathbf{Z_R}$ must all have the same height and so we can fuse **Y** with $\mathbf{Z_L}$ and arrange as shown, leaving height undisturbed.

We have, in effect, passed the fusion effort down the tree to a new <b>-<b> fusion in a shorter sub-tree. So, fusion is recursive, in this case a <r>-<b> fusion giving rise to a shorter <b>-<b> fusion and so on down. Eventually we will descend to a point where <e> leaves are being fused and the job is done.

Provided that <b>-<b> fusions preserve height (and we will see that they do), this fusion also preserves height. Note, that the **Y**-$\mathbf{Z_L}$ fusion *may* produce <r> in which case we will need to correct the colour balance.

Note: The diagram shows a red-rooted sub-tree on the right and a black-rooted sub-tree on the left. A symmetrical inversed arrangement exists too.

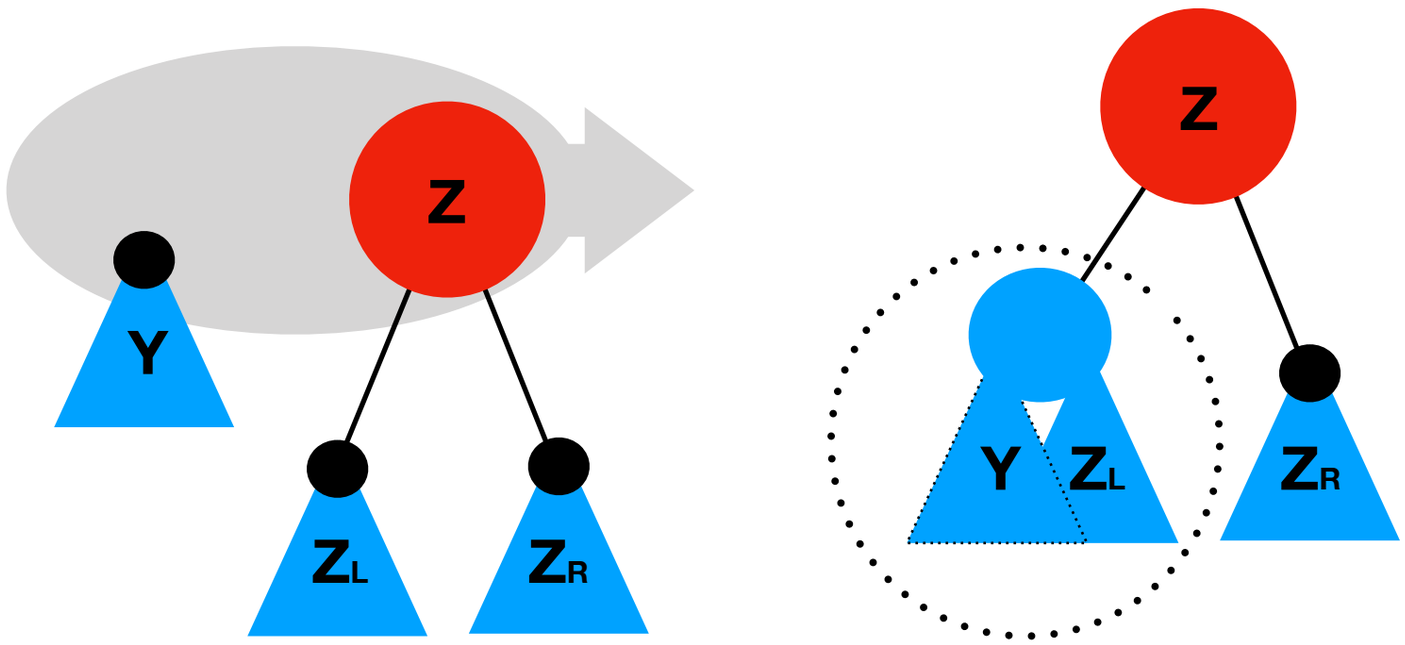


Figure 8: fuse <r> with <b>, height preserved, new <b>-<b> fusion required, colour balance not assured

	Balance
Height	✓
Colour	✗

Fusion: Both <r>

If we fuse $\mathbf{Y}_R$ and $\mathbf{Z}_L$ (always a <b>-<b> fusion) we obtain a sub-tree **S**. Two cases arise:

- the root of **S** is <b>: rearrange as shown in figure 9.a.
- the root of **S** is <r>: rearrange as shown in figure 9.b.

Provided that <b>-<b> fusions preserve height (and we will see that they do), we can see that this fusion also preserves height. In both case we can see that they are not red-balanced.

In each case we have replaced a <r>-<r> fusion by a new <b>-<b> fusion in a shallower sub-tree and again we will descend to a point where empty leaves are being fused and the job is simple.

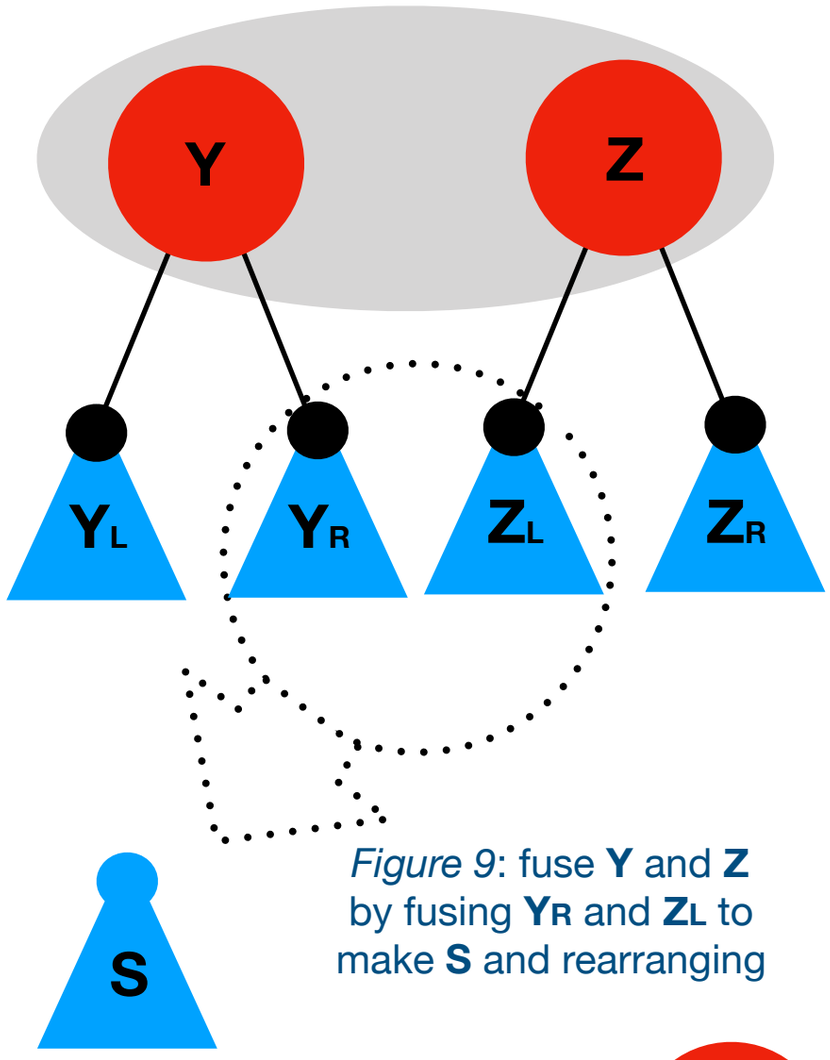


Figure 9: fuse **Y** and **Z** by fusing $\mathbf{Y}_R$ and $\mathbf{Z}_L$ to make **S** and rearranging

Note
If any of the sub-trees in fig 9 are <e> then they all must be <e> and we end in fig 9.a with all the blue trees empty.

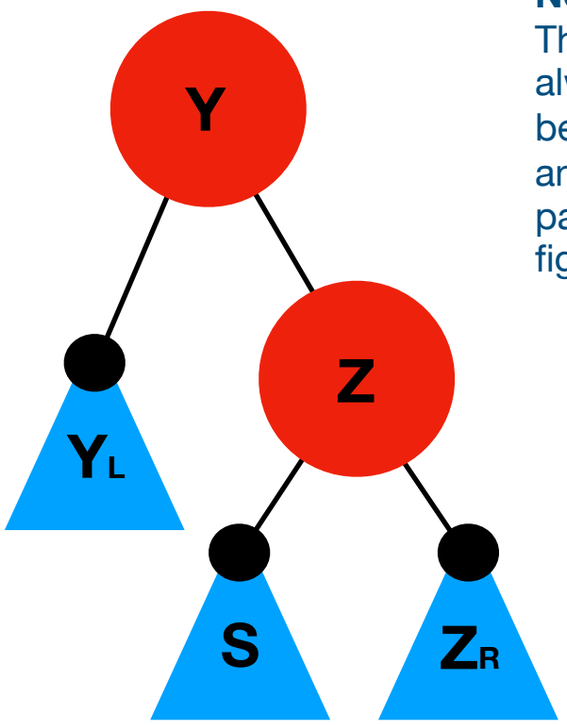


Figure 9.a: arrange thus when **S** is <b>

Note
This fusion of **Y** and **Z** is always for attachment below a <b> because **Y** and **Z** cannot have <r> parents. (See note in figs. 4 for pertinence)

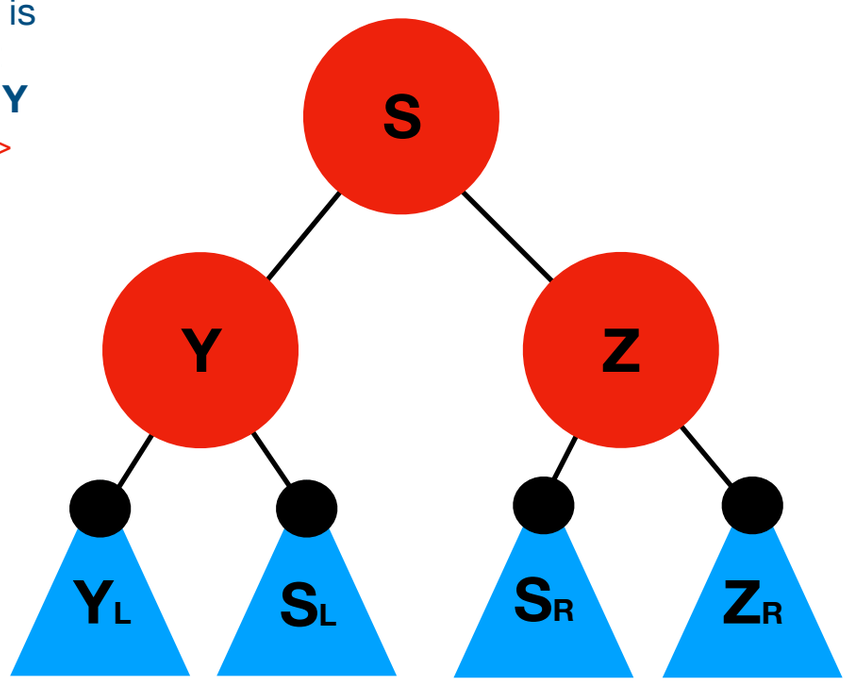


Figure 9.b: arrange thus when **S** is <r>

Fusion: Both <b>

If we fuse $\mathbf{Y}_R$ and $\mathbf{Z}_L$ we obtain a sub-tree **S**. As with <r>-<r> fusion, two cases arise which need to be handled similarly (figures 10.a and 10.b).

Our other fusions all rely on this <b>-<b> fusion preserving height-balance and clearly it does. Red balance may not be preserved if the fused tree is joined to a <r>.

In each case we have replaced a fusion by a new fusion in a shorter sub-tree and again we will descend to a point where empty leaves are being fused and the job is done.

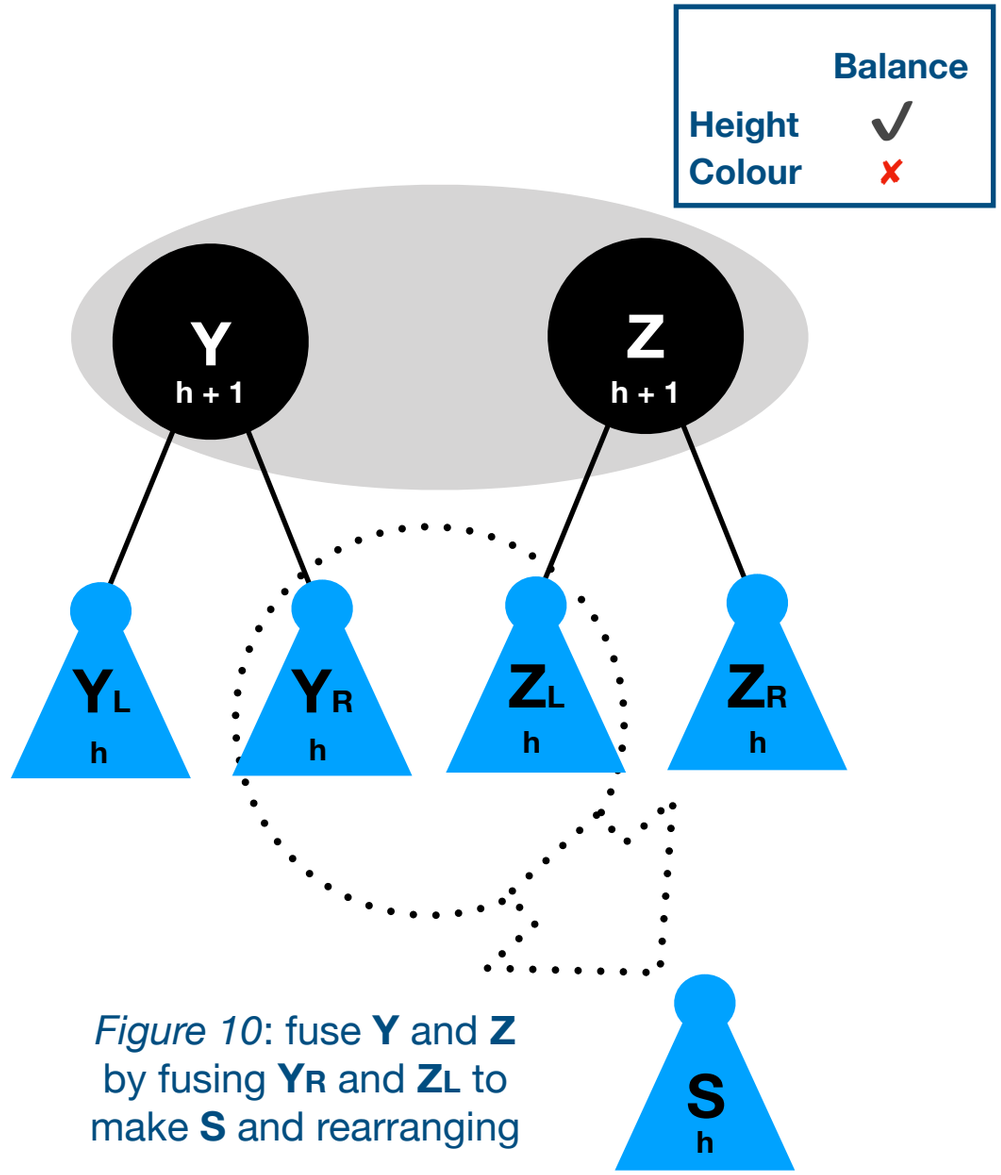


Figure 10: fuse **Y** and **Z** by fusing $\mathbf{Y}_R$ and $\mathbf{Z}_L$ to make **S** and rearranging

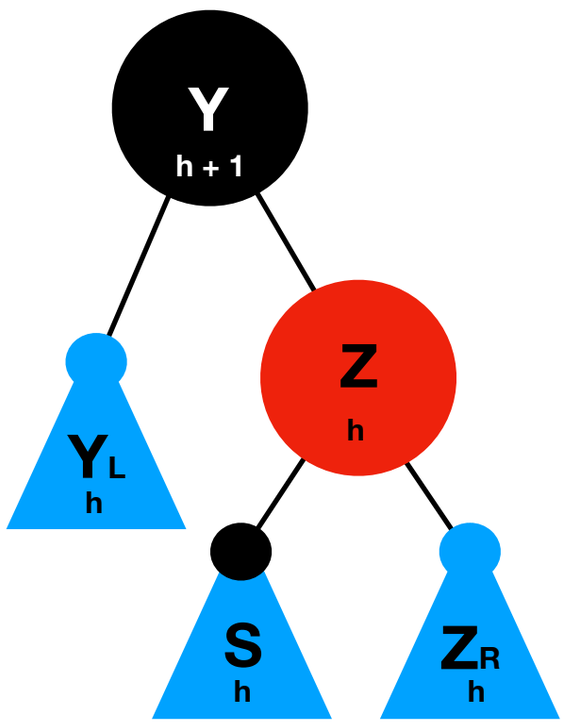


Figure 10.a: arrange thus when **S** is <b> and recolour **Z**

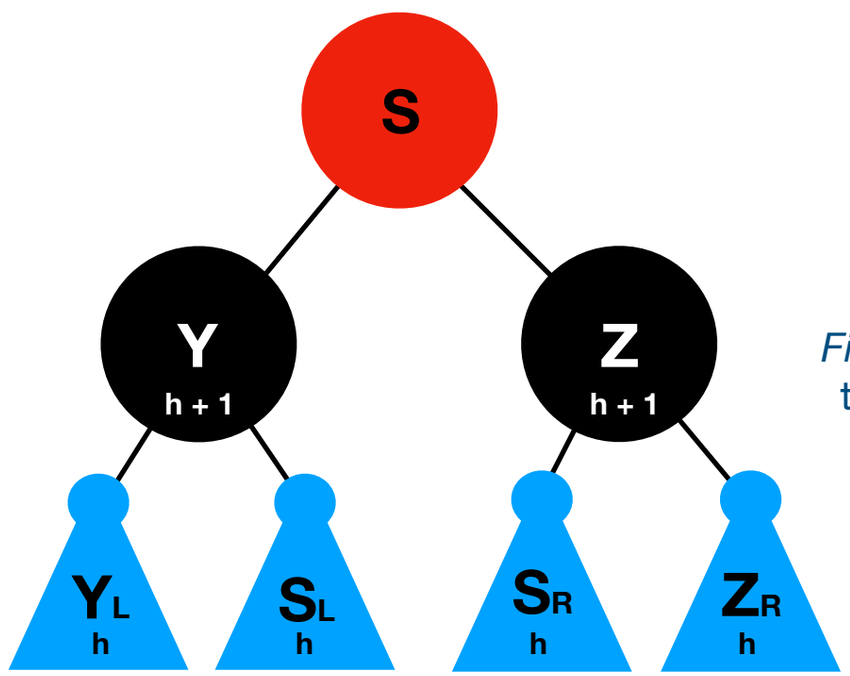


Figure 10.b: arrange thus when **S** is <r>

Deletion: The Story So Far

We're in the process of deleting. We've descended the tree until we found the item to delete (**Z**). We've fused its descendants and now we want to replace the deleted item with this height-balanced but not necessarily red-balanced sub-tree (**S**). Our plan is now to head back up the tree, fixing red-balance, and one further hurdle remains.

If our replacement operation disturbs height-balance (and sometimes it will, as in fig. 11), we will need to fix this too during our ascent.

Specifically, if **Z** is <b>, it's removal from the tree will leave **X** with a sub-tree which is one level shorter than its sibling.

It turns out that the best strategy to handle this will be to shorten its sibling too and pass the unbalance up the tree until either a <r> is reached and repainted or until the root (which has no sibling) is reached. We pass this information up the tree using a *needsBalancing* flag.

Note: The diagrams show deletion in the right-hand sub-tree. Naturally, a symmetrical left-hand alternative exists too. It is obtained simply by exchanging **Y** and **Z**.

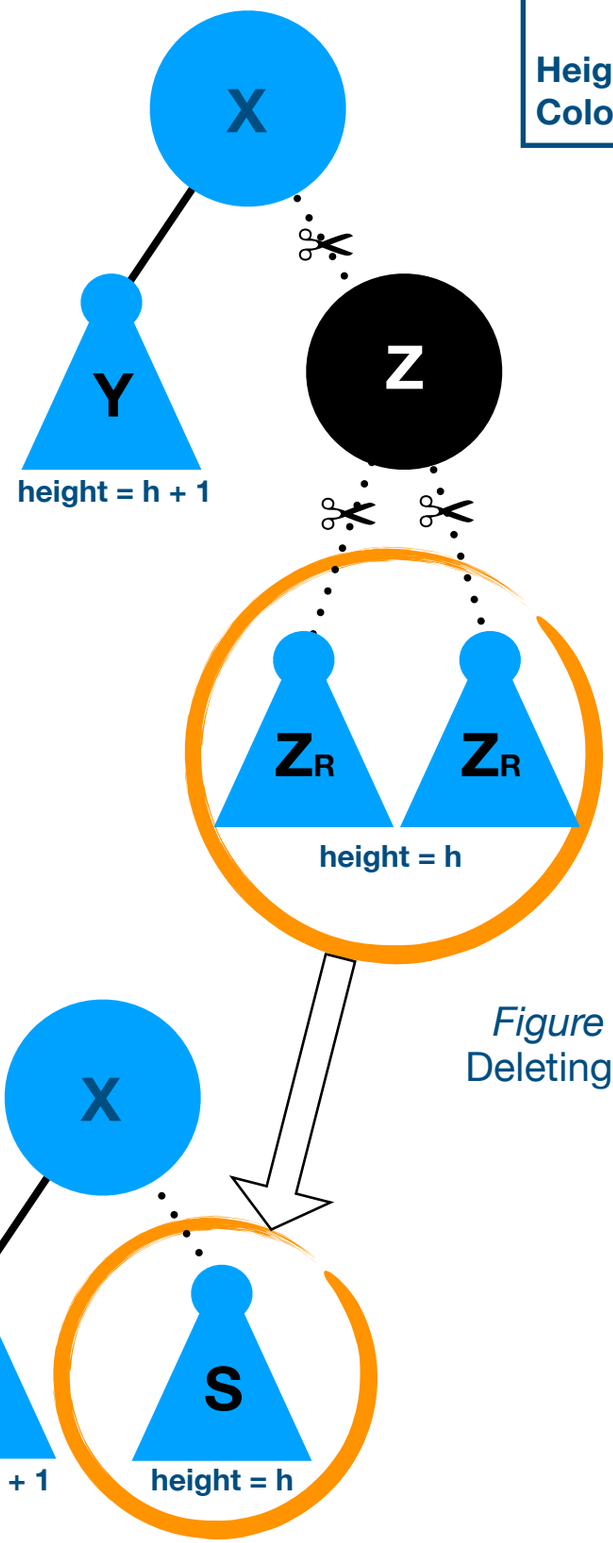


Figure 11.a: Deleting a <b>

Figure 11.b: Resulting in height-imbalance

Height Balancing

We only need to height-balance when we delete a <b> node. Our rebalancing process will return a sub-tree which is not height-balanced and possibly not red-balanced and we can be sure that it will always be one level shorter than its sibling.

Now, if the parent is <r>, we simply paint it <b> and we can rest assured that there will be no further height-imbalance. Note that we do this *after* we height-rebalance so as to first shorten the sibling sub-tree.

If the parent is <b>, we pass a message up the return chain to the next node and it in turn height-balances. Eventually we will meet a <r> node or the root where the imbalance is resolved.

In order to height-rebalance, it turns out that there are four cases, all illustrated on the following slide where **X** is the parent of the deleted node:

- **X** is <r>: Simply paint the sibling root <r> (since as X's child it must be <b> (see fig. 12.1).
- **X** is <b>, **S** is <r>: Simply swap their colours (see fig. 12.2).
- **X** is <b>, **S** is <b>, **Y** is <b>: Change Y to <r> (see fig. 12.3) and correct red-balancing as **Y**'s sub-trees may have red-roots.
- **X** is <b>, **S** is <b>, **Y** is <r>: Complex rearrangement required (see fig. 12.4) and $\mathbf{Y_L}$ will need red-rebalancing.

Note: The diagrams show deletion somewhere down the right-hand sub-tree but, naturally, a symmetrical left-hand alternative exists too.

Balancing Steps

- if deleted node <b> set *needsBalancing* = true
- replace node with fused sub-trees
- if *needsBalancing* = true shorten sibling using one of the patterns in figs. 12
- if siblings' parent node is <r> paint it <b> and set *needsBalancing* = false
- red balancing should always follow
- return *needsBalancing* up the tree

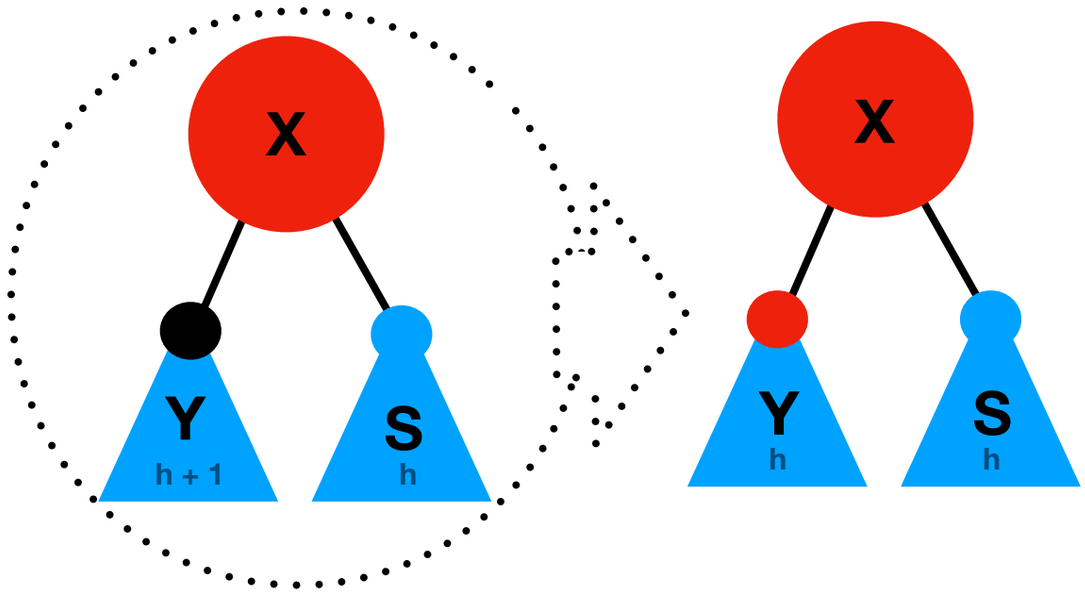


Figure 12.1: Repaint **Y** to shorten the sub-tree.

Note: In every case, the sub-tree returned is one short. Later, rebalancing will notice if it is <r> and repaint it or rebalance at the higher level.

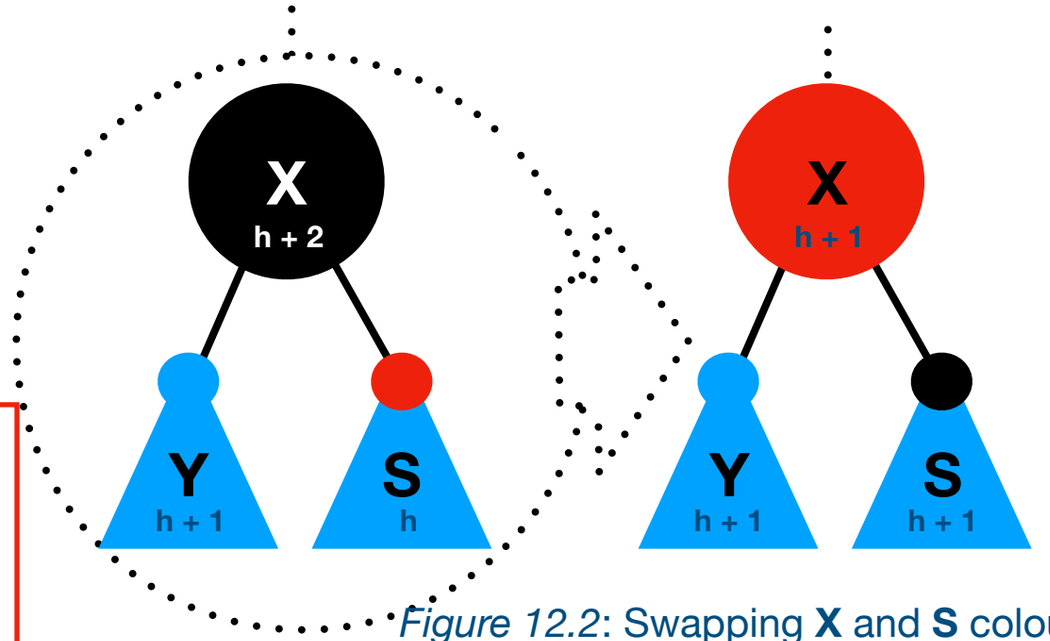


Figure 12.2: Swapping **X** and **S** colours rebalances and reduces **X** height, sending imbalance up a level.

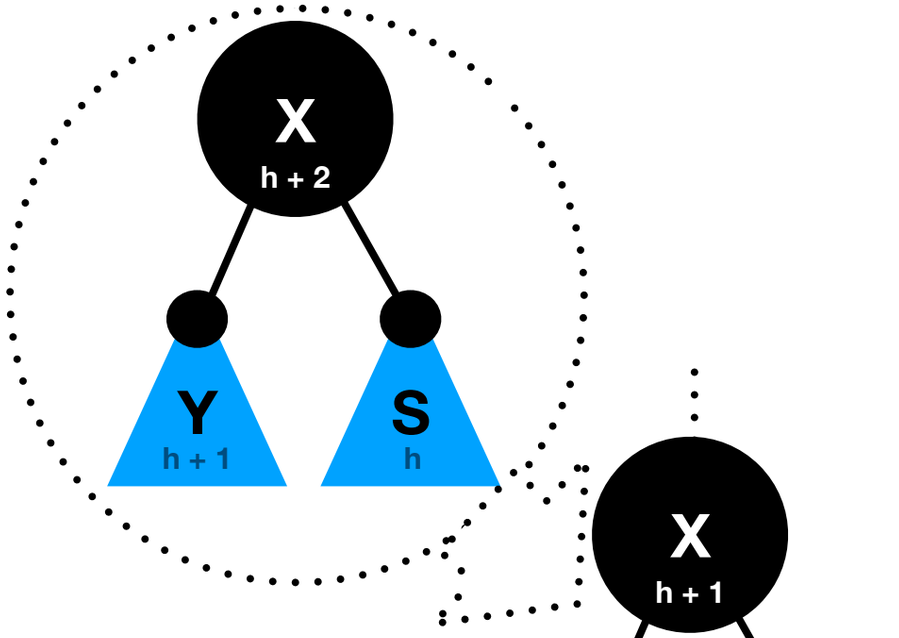
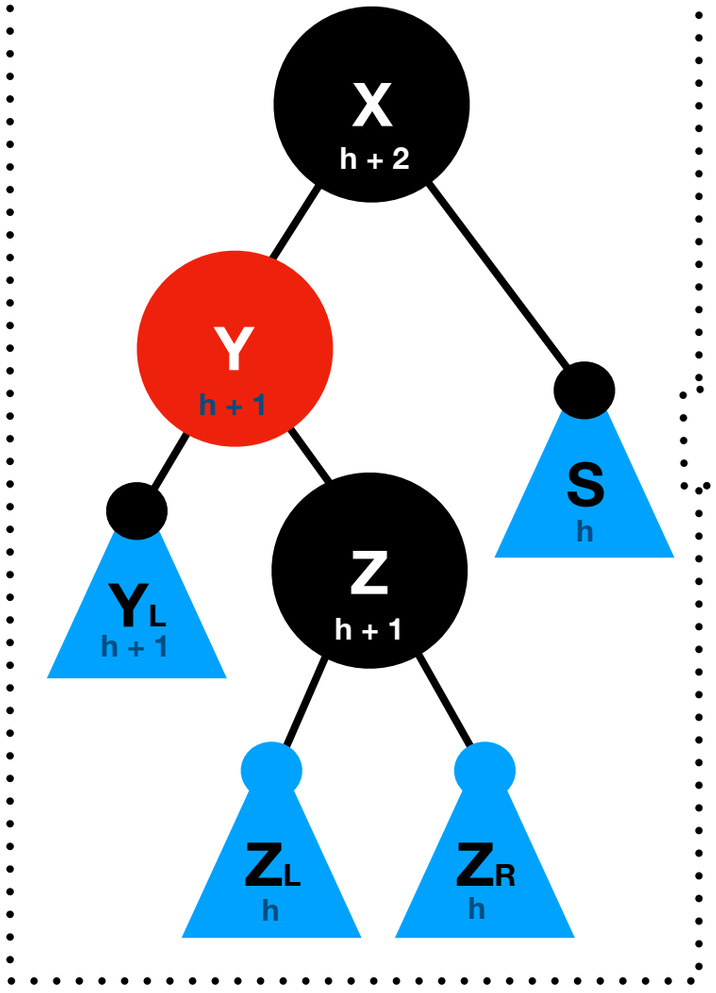


Figure 12.3: Recolouring **Y** reduces its height but may required red-balancing as **Y** may have <r> children



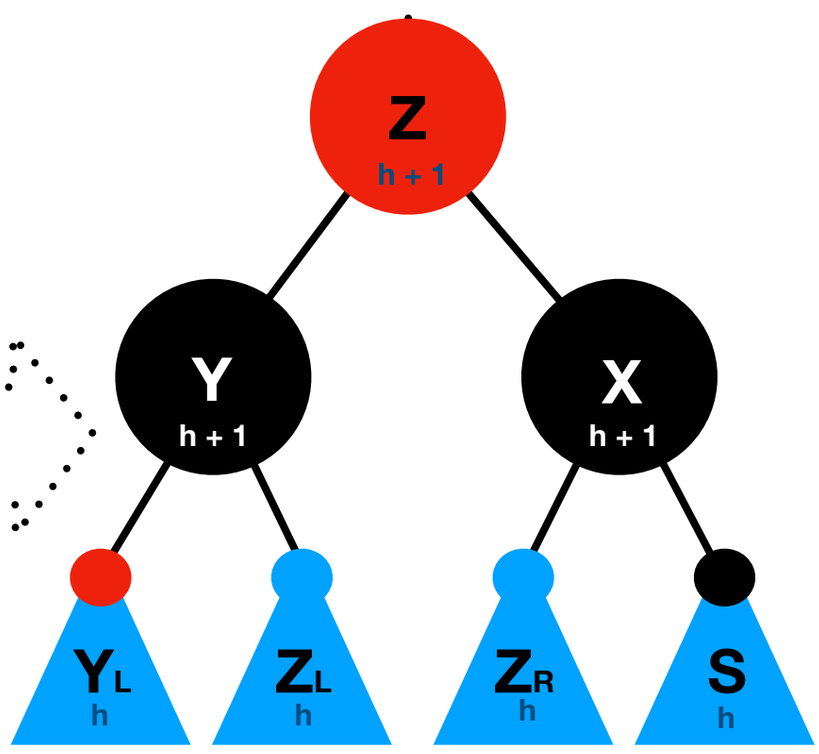


Figure 12.4: Recolouring **YL** reduces its height but may required red-balancing

Algebraic notation for trees

A tree can be expressed recursively as a list of four elements:

[*colour*, *value*, *left sub-tree*, *right sub-tree*]

So, for example, the sub-tree opposite contains*:

Z = [<b>, z, <e>, <e>],
Y = [_, y, $\mathbf{Y}_L$, $\mathbf{Y}_R$],
X = [<r>, x, **Y**, **Z**] and

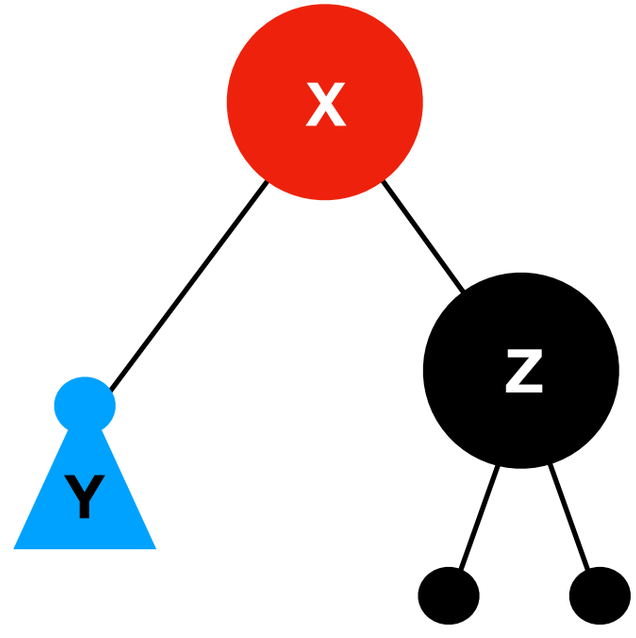


Figure 13

Trees can be expanded using the notation. For instance we can write:

X = [<r>, x, [_, y, $\mathbf{Y}_L$, $\mathbf{Y}_R$], [<b>, z, <e>, <e>]]

and continue as deep as we wish while we have sub-trees to expand, such as $\mathbf{Y}_L$ and $\mathbf{Y}_R$.

This can be easily represented as an `enum` and all the operations described implemented using *pattern matching* in `switch` statements.

Note: Lowercase letters indicate the value stored at the node. The underscore signifies that any value works, so here it means, “whatever the colour of **Y**”. Actually, in this particular tree, in order to observe the invariant rules, **Y** is either <b> or [<r>,y,<e>,<e>], so when we can see leaves, we can often deduce some of the sub-trees nearby.

Pattern Matching

Look! all those complex diagrams translate to a line of code each, in the form of: *oldPattern <- newPattern*

Red-Balancing (4 patterns, see fig 4)

```
private func redBalanced() -> RedBlackTree<T> {
   switch self {
   case let .node(_, z, .node(.red, y, .node(.red, x, a, b), c), d),
        let .node(_, z, .node(.red, x, a, .node(.red, y, b, c)), d),
        let .node(_, x, a, .node(.red, z, .node(.red, y, b, c), d)),
        let .node(_, x, a, .node(.red, y, b, .node(.red, z, c, d))):
      return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
   default: return self
   }
}
```

Height-Balancing (4 patterns, see fig 12)

```
private func rightBalanced() -> RedBlackTree<T> {
   var result = RedBlackTree<T>.empty
   switch self {
   case let .node(.red, y, .node(.black, k, t2, t3), t1):
      return RedBlackTree<T>.node(.red, y, .node(.red, k, t2, t3), t1).redBalanced()
   case let .node(.black, y, t1, .node(.red, x, t2, t3)):
      return .node(.red, y, t1, .node(.black, x, t2, t3))
   case let .node(.black, y, .node(.black, z, t1, t2), t3):
      return RedBlackTree<T>.node(.black, y, .node(.red, z, t1, t2), t3).redBalanced()
   case let .node(.black, y, .node(.red, z, .node(.black, k, l, r), .node(.black, u, t2, t3)), t4):
      return .node(.red, u, RedBlackTree<T>.node(.black, z, .node(.red, k, l, r), t2).redBalanced(),
                   .node(.black, y, t3, t4))
   default: return self
   }
}
```

Note: a symmetric leftBalanced() function exists too

Pattern Matching (continued)

Fusion (6 patterns, see figs 7-10)

```
extension RedBlackTree {
   private func recursiveFuse(_ with: RedBlackTree<T>) -> RedBlackTree<T> {
     switch (self, with) {
      case (.empty,.empty):
         return .empty
      case let (t1,.empty), let (.empty,t1):
         return t1
      case let (.node(.black, _, _, _), .node(.red, y, t3, t4)):
         return RedBlackTree<T>.node(.red, y, self.recursiveFuse(t3), t4).redBalanced()
      case let (.node(.red, x, t1, t2), .node(.black, _, _, _)):
         return RedBlackTree<T>.node(.red, x, t1, t2.recursiveFuse(with)).redBalanced()
      case let (.node(.red, x, t1, t2),.node(.red, y, t3, t4)):
         let s = t2.recursiveFuse(t3)
         switch s {
         case let .node(.red, z, s1, s2):
            return RedBlackTree<T>.node(.red, z, .node(.red, x, t1, s1), .node(.red, y, s2, t4)).redBalanced()
         default:
            return RedBlackTree<T>.node(.red, x, t1, .node(.red, y, s, t4)).redBalanced()
         }
      case let (.node(.black, x, t1, t2),.node(.black, y, t3, t4)):
         let s = t2.recursiveFuse(t3)
         switch s {
         case let .node(.red, z, s1, s2):
            return RedBlackTree<T>.node(.red, z, .node(.black, x, t1, s1), .node(.black, y, s2, t4)).redBalanced()
         default:
            return RedBlackTree<T>.node(.black, x, t1, .node(.red, y, s, t4)).redBalanced()
        }
      }
   }
}
```

The Tree

Enums

Look! No *structure* or *class* object; simply an enum…

```
public enum RedBlackTree<T: RedBlackTreeOrderingProtocol> {
   case empty
   indirect case node(_ colour: NodeColour,
      _ value:T,
      _ left: RedBlackTree<T>,
      _ right: RedBlackTree<T>)
}

public enum RedBlackTreeComparator {
   case leftTree
   case keySlot
   case rightTree
}

public enum NodeColour: CustomStringConvertible {
   case black
   case red
}
```

Protocol

… and a protocol to make ordering more flexible.

```
infix operator ≺: ComparisonPrecedence

public protocol RedBlackTreeOrderingProtocol {
   static func ≺(lhs: Self,rhs: Self) -> RedBlackTreeComparator
}
```

Insertion

```
extension RedBlackTree {
   @discardableResult
   /// Inserts an element in the `RedBlackTree` provided it
   /// has no duplicate key in the tree already.
   /// Returns true/false to show success/failure.
   public mutating func insert(_ element: T) -> Bool {
      let (tree, old) = recursiveInsert(element)
      switch tree {
      case .empty: return false
      case let .node(_, value, left, right):
         self = .node(.black, value, left, right)
      }
      return old == nil
   }

   /// Recursive helper function for insert(element) which should
   /// not be called directly.
   private func recursiveInsert(_ element: T) -> (tree: RedBlackTree, old: T?) {
      switch self {
      case .empty:
         return (.node(.red, element, .empty, .empty), nil)
      case let .node(colour, value, left, right):
         switch element ≺ value {
         case .keySlot:
            return (self, value)
         case .leftTree:
            let (l, old) = left.recursiveInsert(element)
            if let old = old { return (self, old) }
            return (RedBlackTree<T>.node(colour, value, l, right).redBalanced(), old)
         case .rightTree:
            let (r, old) = right.recursiveInsert(element)
            if let old = old { return (self, old) }
            return (RedBlackTree<T>.node(colour, value, left, r).redBalanced(), old)
         }
      }
   }
}
```

Deletion

```
extension RedBlackTree {
   public mutating func delete(_ element: T) {
      let search = recursiveDeleteSearch(element)
      switch search.tree {
      case .empty: self = .empty
      case let .node(_, value, left, right):
         self = RedBlackTree<T>.node(.black, value, left, right)
      }
   }

   private func recursiveDeleteSearch(_ element: T) -> (tree: RedBlackTree<T>, fixHeight: Bool) {
      switch self {
      case .empty:
         return (self, false)
      case let .node(_, key, _, _):
         switch element <> key {
         case .keySlot:    // found it!!
            return self.replace()
         case .leftTree:      // Still looking (left)
            let s = self.leftDelete(element)
            return (s.tree.redBalanced(), s.fixHeight)
         case .rightTree:     // Still looking (right)
            let s = self.rightDelete(element)
            return (s.tree.redBalanced(), s.fixHeight)
         }
      }
   }
}
```

Deletion Helpers

```
extension RedBlackTree {
   private func replace() -> (tree: RedBlackTree<T>, fixHeight: Bool) {
      switch self {
      case let .node(.black, _, left, right):
         let s = left.recursiveFuse(right)
         return (s, true)
      case let .node(.red, _, left, right):
         let s = left.recursiveFuse(right)
         return (s, false)
      default: return (self, false)
      }
   }

   private func rightDelete(_ element: T) -> (tree: RedBlackTree<T>, fixHeight: Bool) {
      switch self {
      case .empty:
         return (self, false)
      case let .node(.red, key, left, right):
         let s = right.recursiveDeleteSearch(element)
         if s.fixHeight {
            return (RedBlackTree<T>.node(.black, key, left, s.tree).rightBalanced(), false)
         } else {
            return (RedBlackTree<T>.node(.red, key, left, s.tree), s.fixHeight)
         }
      case let .node(.black, key, left, right):
         let s = right.recursiveDeleteSearch(element)
         if s.fixHeight {
            return (RedBlackTree<T>.node(.black, key, left, s.tree).rightBalanced(), s.fixHeight)
         } else {
            return (RedBlackTree<T>.node(.black, key, left, s.tree), s.fixHeight)
         }
      }
   }
}
```

Note: a symmetric leftDelete() function exists too