Red–black trees

CMPT 225

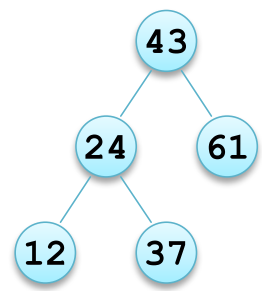
Objectives

- Define the red-black tree properties
- Describe and implement rotations
- Implement red-black tree insertion
- Implement red-black tree removal

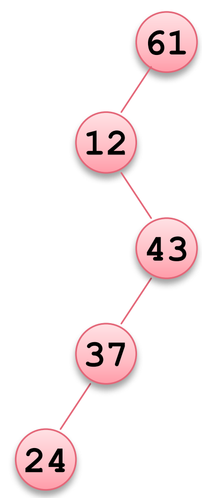
Red-black tree algorithms derived from material in *Introduction to Algorithms*, *Cormen*, *Leiserson* and *Rivest*

Binary Search Trees – Performance

- Insertion and removal from BSTs is O(*height*)
- What is the height of a BST?
  - If the tree is balanced: O(log*n*)
  - If the tree is very unbalanced: O(*n*)



balanced BST
height = O(log*n*)



unbalanced BST
height = O(*n*)

Balanced Binary Search Trees

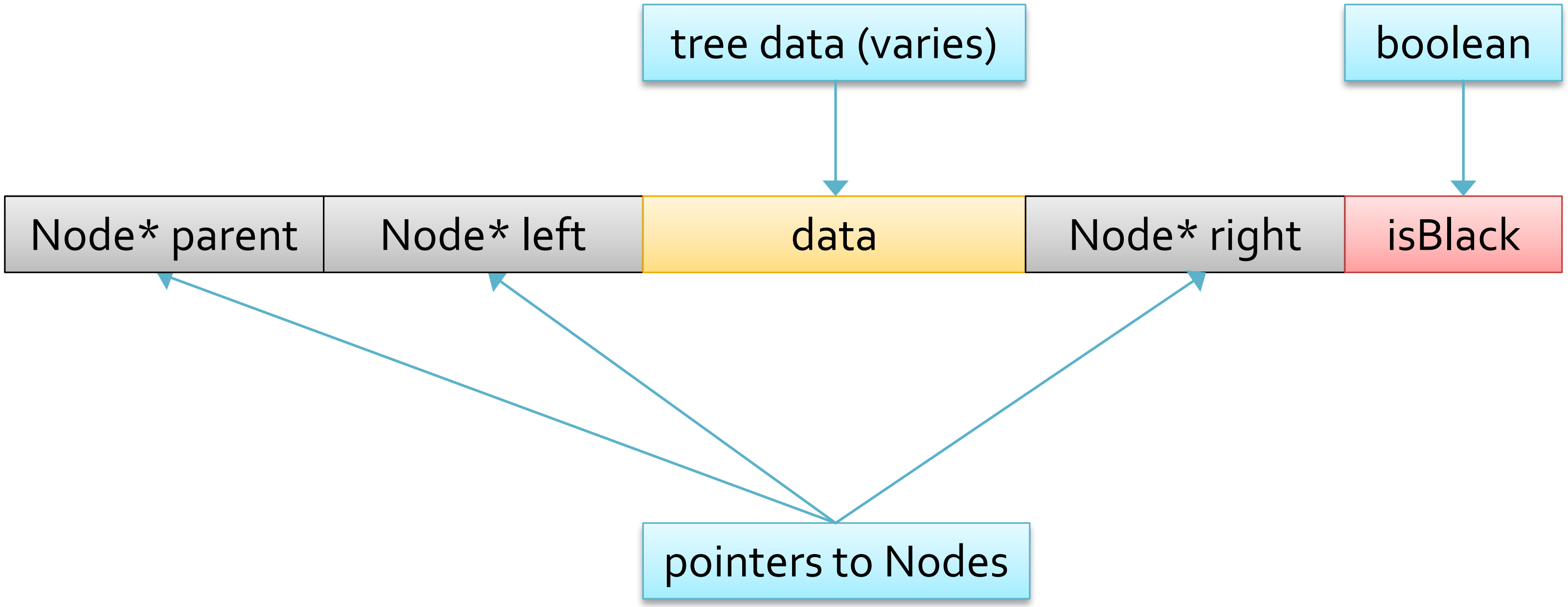
- Define a balanced binary tree as one where
  - There is no path from the root to a leaf that is more than twice as long as any other such path
  - The height of such a tree is O(log*n*)
- Guaranteeing that a BST is balanced requires either
  - A more complex structure (2-3 and 2-3-4 trees) or
  - More complex insertion and deletion algorithms (red-black trees)

Red-black Tree Structure

- A red-black tree is a balanced BST
- Each node has an extra colour field which is
  - red or **black**
    - Usually represented as a boolean – **`isBlack`**
- Nodes have an additional pointer to their parent
- A node's *null* child pointers are treated as if they were black nodes
  - These null children are *imaginary* nodes so are not allocated space
  - And are always coloured black

Red-black Tree Nodes

- Red-black trees are reference structures
- Nodes contain data, three pointers to nodes, and the node's colour



Red-black Tree Properties

1. Every node is either red or **black**
2. Every leaf is **black**
   - Leaves refers to the *imaginary* nodes
     - i.e. every *null child* of a node is considered to be a black leaf
3. If a node is red both its children *must* be **black**
4. Every path from a node to a leaf contains the same number of *black* nodes
5. The root is **black** - for convenience

Red-black Tree Height

- The black height of a node, $bh(v)$, is the number of black nodes on a path from v to a null black child
  - Without counting v itself
  - Property **4** – every path from a node to a leaf contains the same number of black nodes
- The height of a node, $h(v)$, is the number of nodes on the longest path from v to a leaf
  - Without counting v itself
  - Property **3** – a red node's children must be black
    - So $h(v) \leq 2(bh(v))$

Balanced Trees

- It can be shown that a tree with the red-black structure is balanced
  - A balanced tree has no path from the root to a leaf that is more than twice as long as any other such path
- Assume that a tree has *n* internal nodes
  - An internal node is a non-leaf node, and the leaf nodes are *imaginary* nodes so *n* is the number of actual nodes
  - A red-black tree has $\geq 2^{bh} - 1$ internal (real) nodes
    - Can be proven by induction (e.g. Algorithms, Cormen et al)
    - But consider that a *perfect* tree has 2^{h+1} leaves, *bh* must be less than or equal to *h*, and that $2^{h+1} = 2^h + 2^h$

Red-black Tree Height

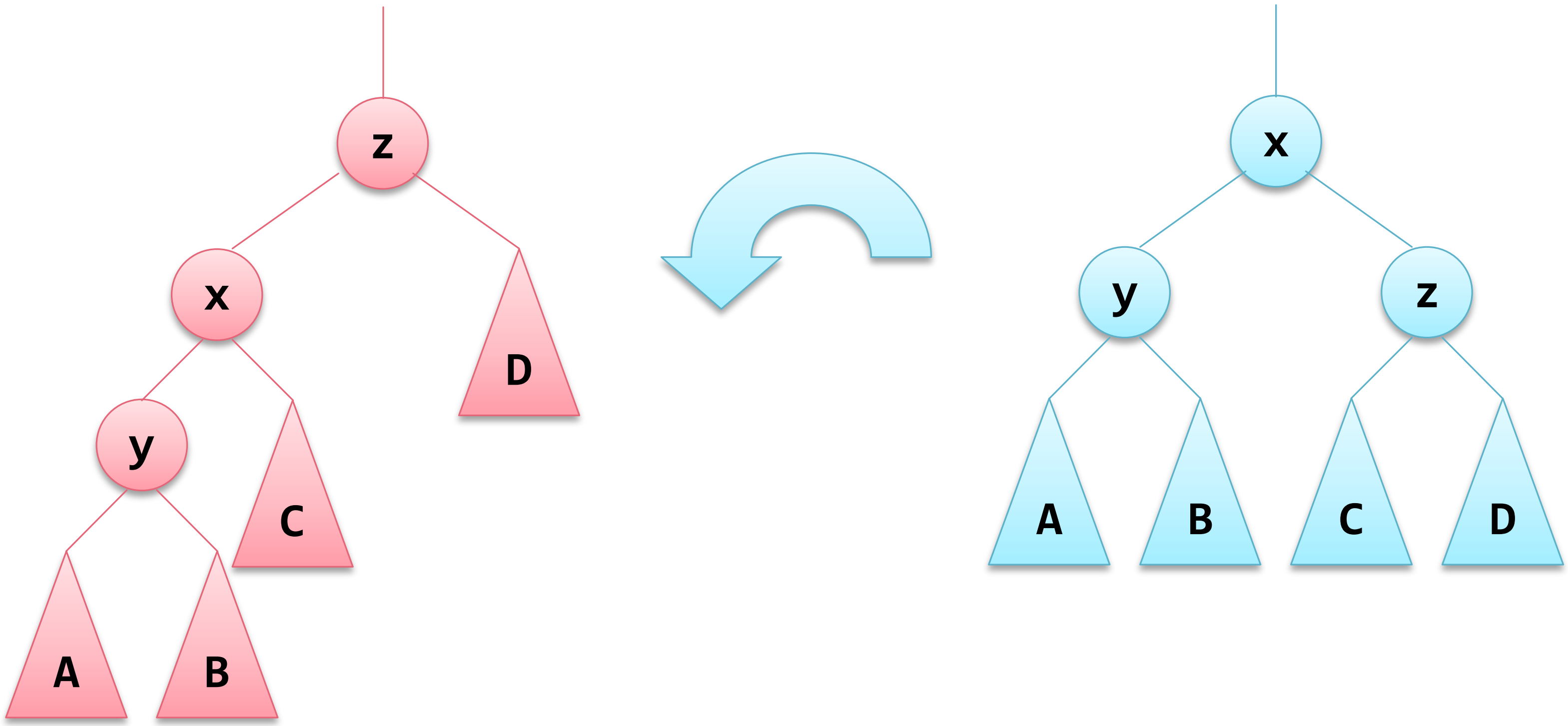
- Claim: a red-black tree has height, $h \leq 2 * \log(n+1)$
  1. $n \geq 2^{bh} - 1$ *from claim on previous slide*
  2. $bh \geq h / 2$ *red nodes must have **black** children*
  3. $n \geq 2^{h/2} - 1$ *replace bh in 1 with h*
  4. $\log(n + 1) \geq h / 2$ *$\log_2$ of both sides of 3, add 1*
  5. $2 * \log(n + 1) \geq h$ *multiply both sides of 4 by 2*
  6. $h \leq 2 * \log(n + 1)$ *reverse 5*
- Note that $2 * \log(n+1)$ is $O(\log(n))$
  - If insertion and removal are O(*height*) they are $O(\log(n))$

Rotations

- An item must be inserted into a red-**black** tree at the correct position
- The shape of a tree is determined by
  - The values of the items inserted into the tree
  - The order in which those values are inserted
- This suggests that there is more than one tree (shape) that can contain the same values
- A tree's shape can be altered by *rotation* while still preserving the *bst* property

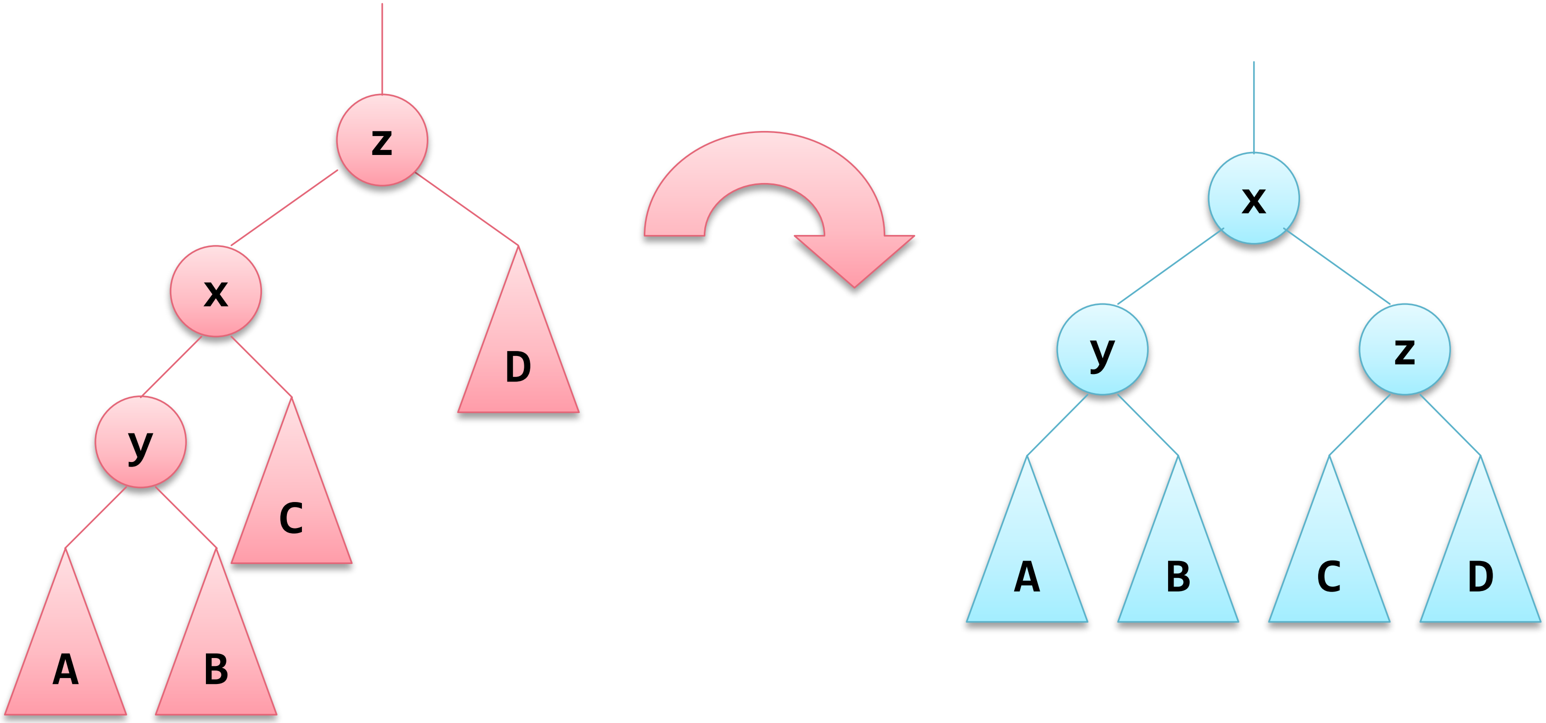
Left Rotation

Left rotate (x)



Right Rotation

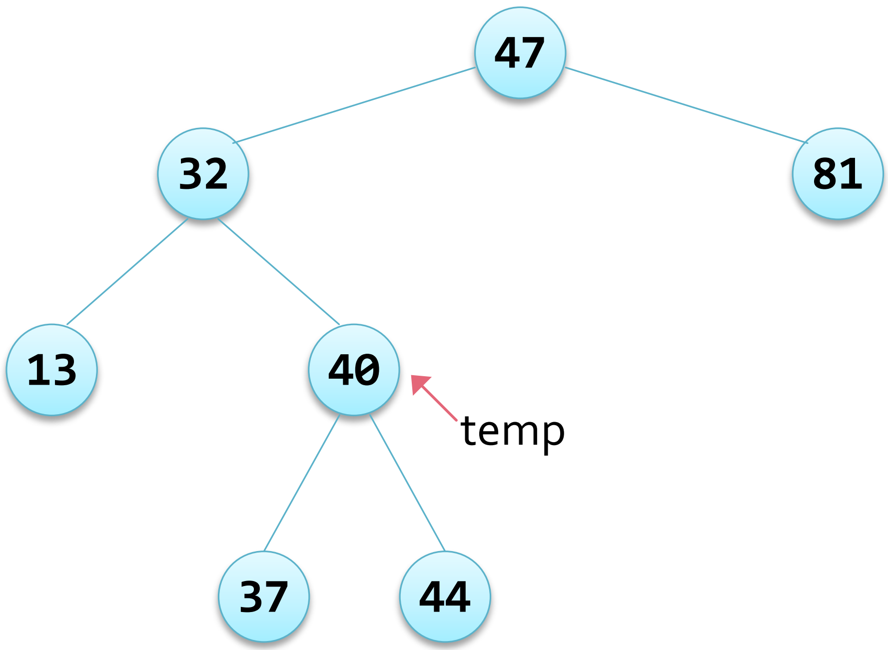
Right rotate (z)



Left Rotation Example

Left rotation of 32 (referred to as x)

Create a pointer to *x*'s right child



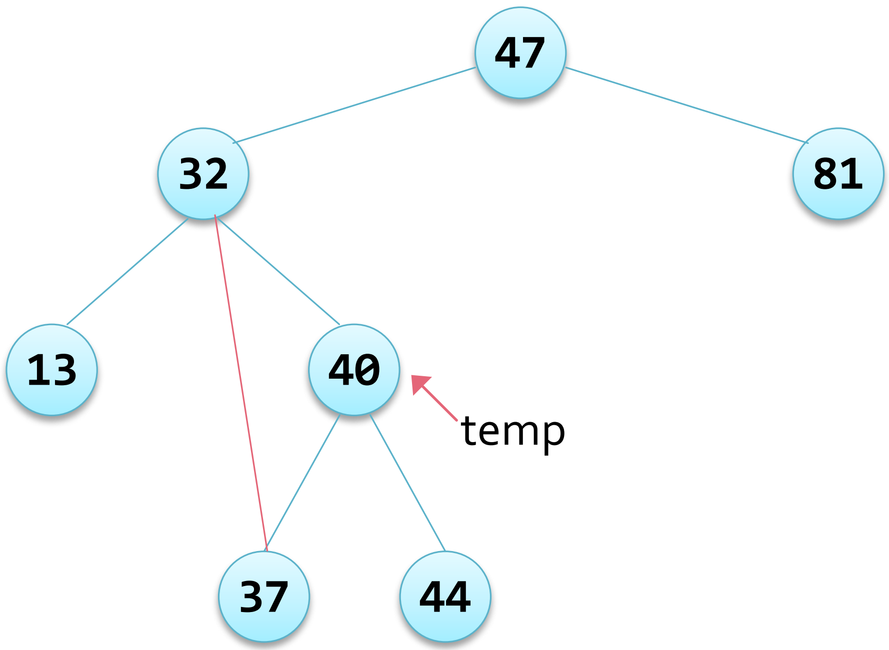
Left Rotation Example

Left rotation of 32 (referred to as x)

Create a pointer to *x*'s right child

Make *temp*'s left child, *x*'s right child

Detach *temp*'s left child



Left Rotation Example

Left rotation of 32 (referred to as x)

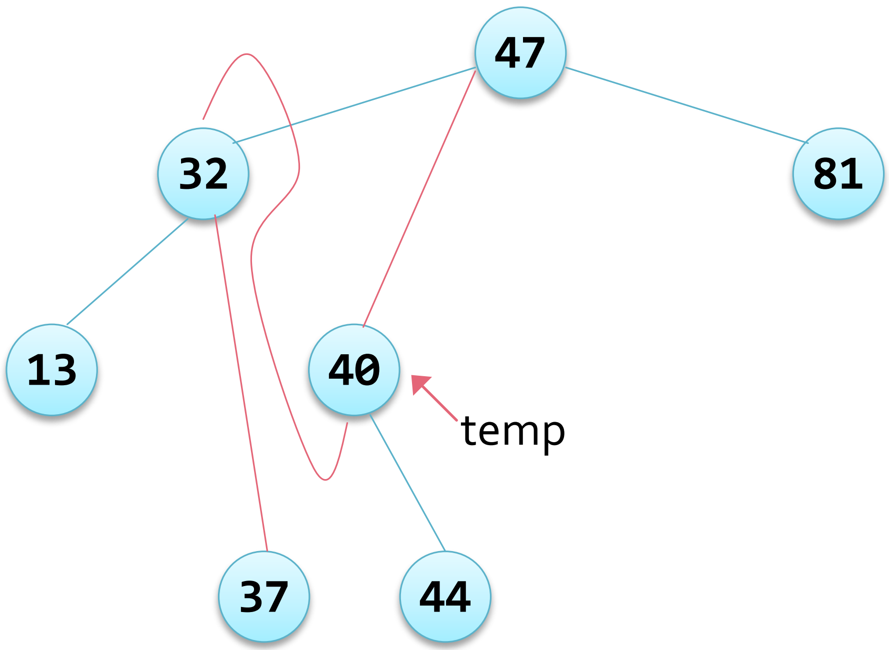
Create a pointer to *x*'s right child

Make *temp*'s left child, *x*'s right child

Detach *temp*'s left child

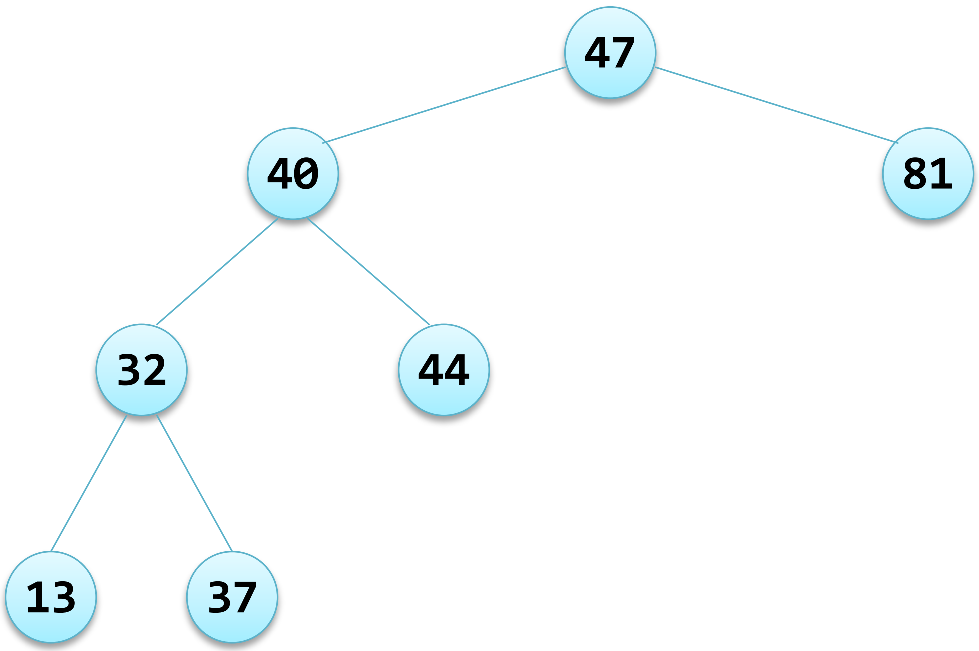
Make *x* the left child of *temp*

Make *temp* the child of *x*'s parent



Left Rotation Example

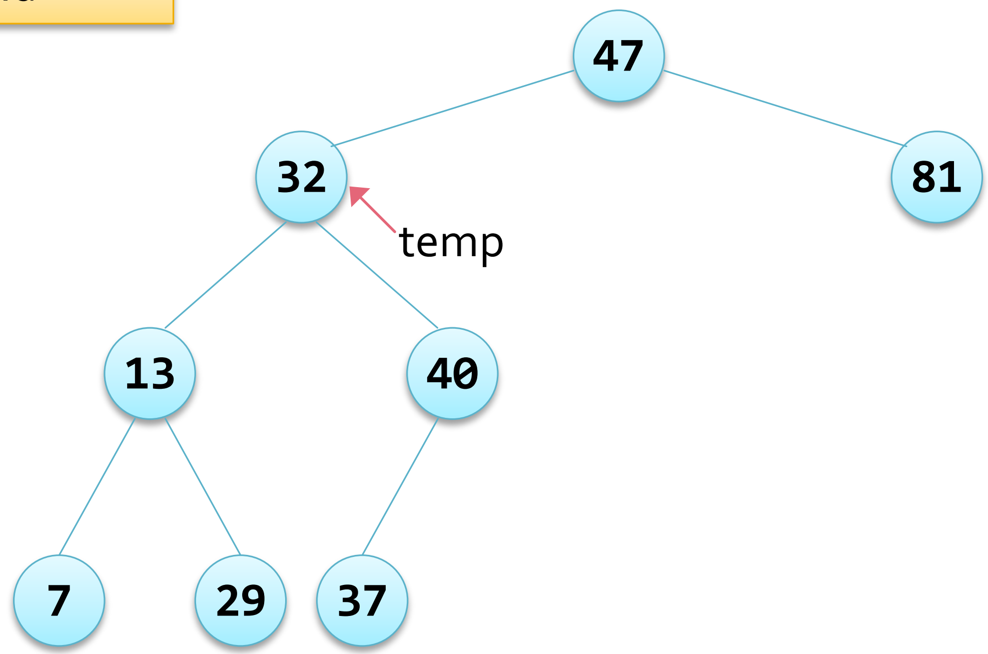
Left rotation of 32 (complete)



Right Rotation Example

Right rotation of 47 (referred to as x)

Create a pointer to *x*'s left child



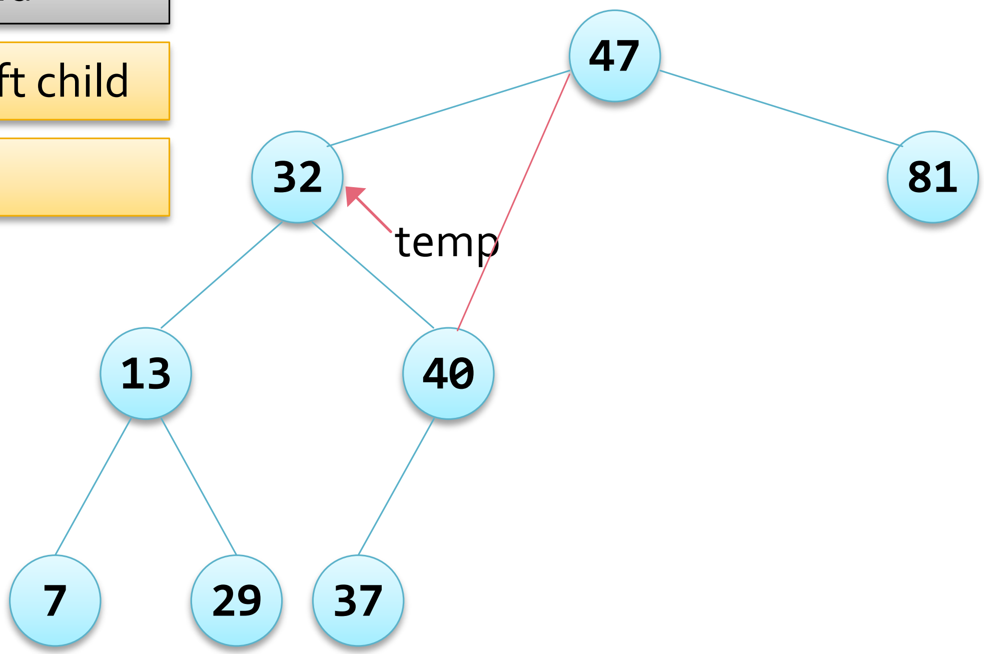
Right Rotation Example

Right rotation of 47 (referred to as x)

Create a pointer to *x*'s left child

Make *temp*'s right child, *x*'s left child

Detach *temp*'s right child



Right Rotation Example

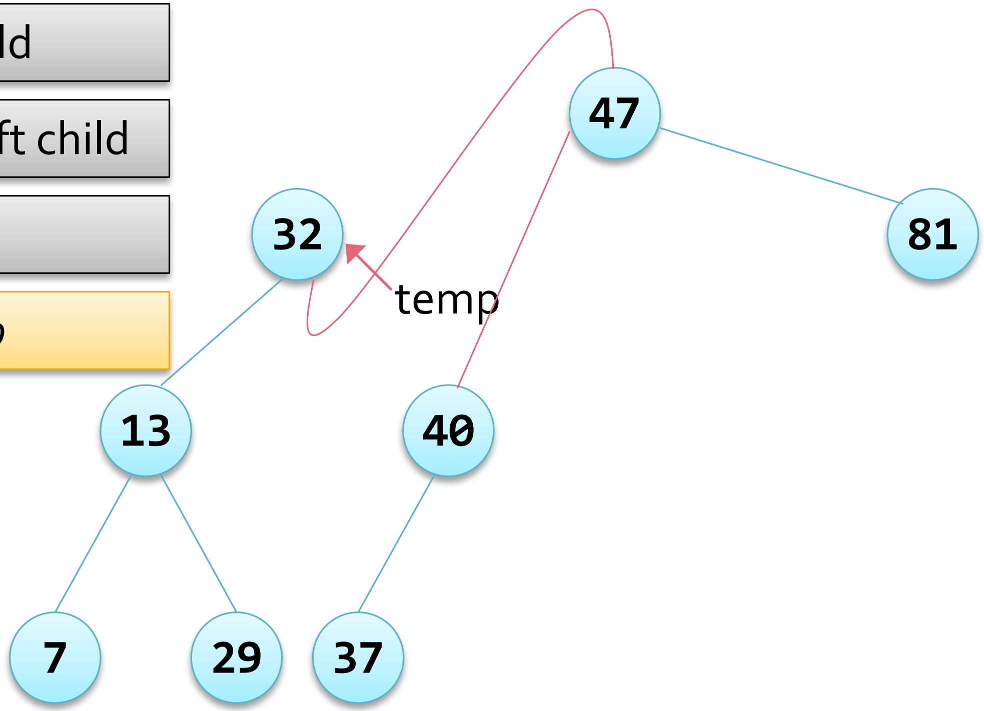
Right rotation of 47 (referred to as x)

Create a pointer to *x*'s left child

Make *temp*'s right child, *x*'s left child

Detach *temp*'s right child

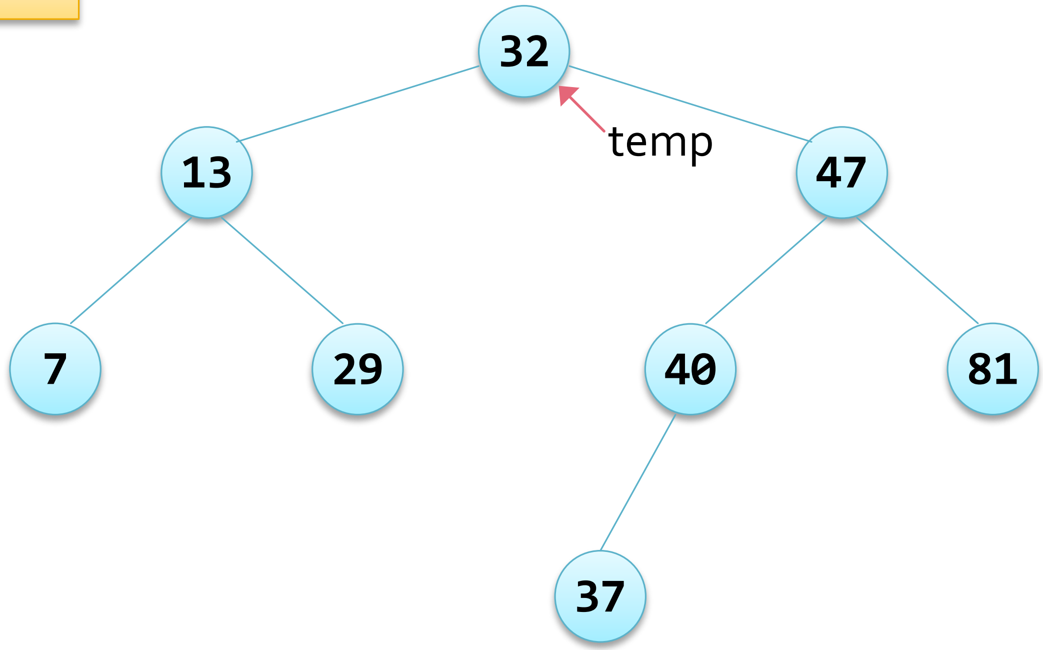
Make *x* the right child of *temp*



Right Rotation Example

Right rotation of 47

Make temp the new root



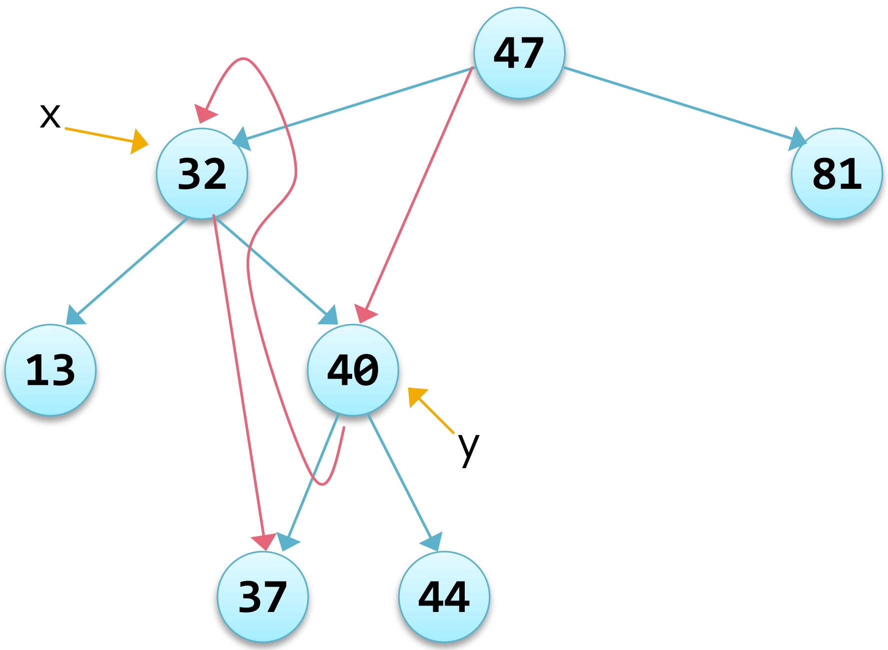
Left Rotation Code

Notation

.left is left child, *.right* is right child, *.p* is parent

```
leftRotate(x) // x is the node to be rotated
    y = x.right
    x.right = y.left
    // Set nodes' parent references
    // y's left child
    if (y.left != null)
            y.left.p = x
    // y
    y.p = x.p

    // Set child reference of x's parent
    if (x.p == null) //x was root
        root = y
    else if (x == x.p.left) //left child
        x.p.left = y
    else
        x.p.right = y
    // Make x y's left child
    y.left = x
    x.p = y
```



Red-black Tree Insertion

- Insert as for a *bst* and make the new node red
  - The only property that can be violated is that both a red node's children are black (it's parent may be red)
- If this is the case try to fix it by colouring the new node red and making it's parent and uncle black
  - Which only works if *both* were red
    - As otherwise the equal *bh* property will be violated
- If changing the colours doesn't work the tree must be rotated
  - Which also entails changing some colours

BST Insertion Algorithm

where x is the new node

```
rbInsert(x)
    bstInsert(x)
    x.colour = red
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //"uncle" of x
            if (y.colour == red) //same as x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```

calls the normal bst insert method

iterates until the root or a black parent is reached

x's parent is a left child

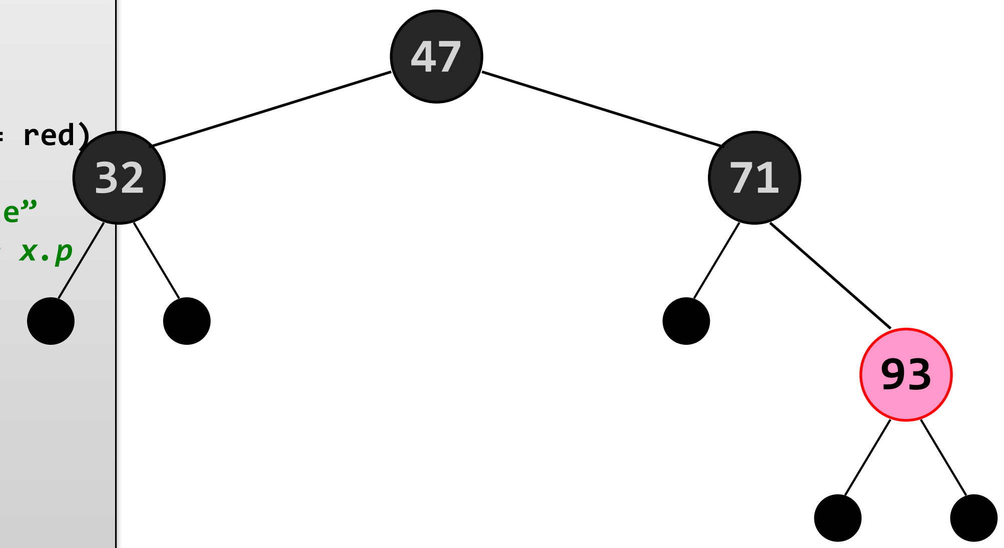
y and x's parent are both red so they can be made black, and x's grandparent can be made red, then make x the grandparent and repeat

x's grandparent must be black, so arrange x and parent in a straight line, then rotate x's grandparent to re-balance the tree, and fix the colours

one important note: in this presentation null children are just treated as black nodes, in an implementation they would have to be explicitly tested for since, being null, they do not have an *isBlack* attribute (or any other attribute)

Insertion Example – 65

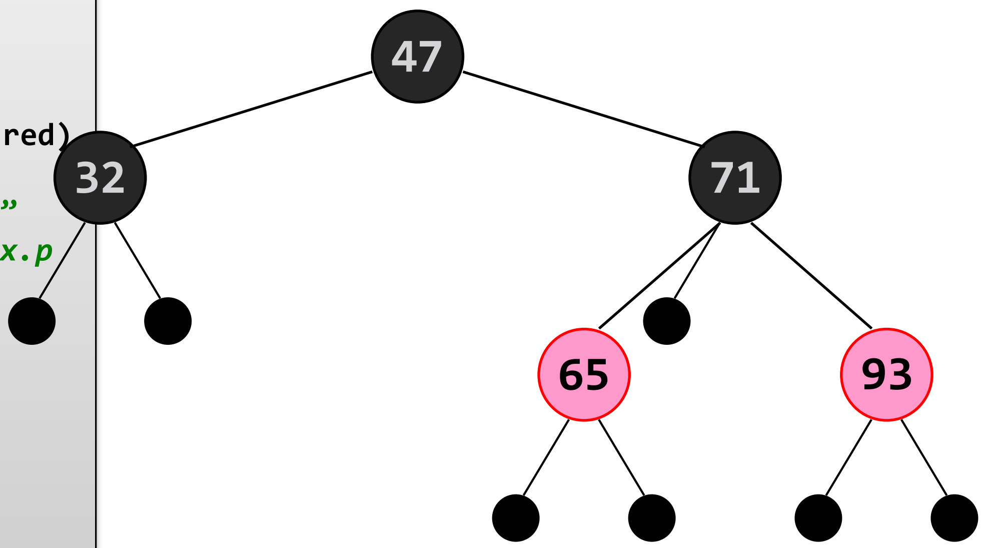
```
rbInsert(x)
    bstInsert(x)
    x.colour = red      //false
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //x's "uncle"
            if (y.colour == red) //like x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```



Insert 65

Insertion Example – 65

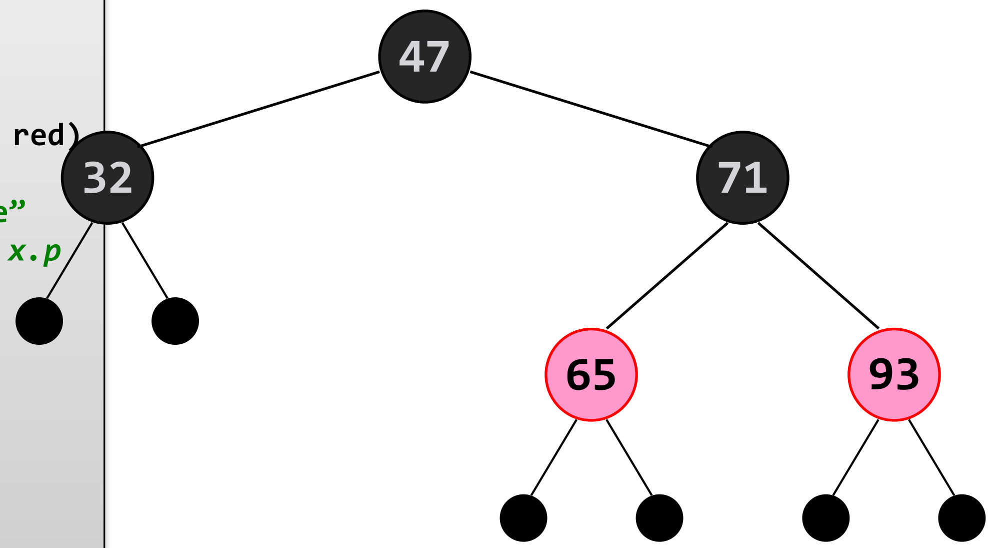
```
rbInsert(x)
    bstInsert(x)
    x.colour = red        //false
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //x’s “uncle”
            if (y.colour == red) //like x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```



Insert 65

Insertion Example – 82

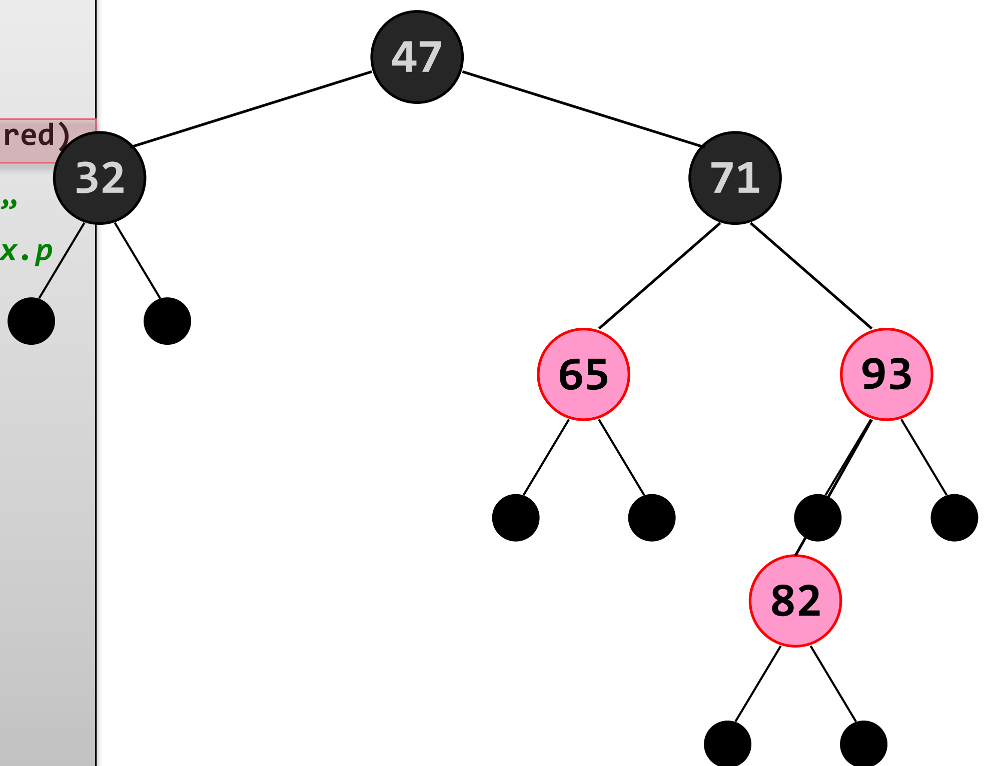
```
rbInsert(x)
    bstInsert(x)
    x.colour = red
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //x's "uncle"
            if (y.colour == red) //like x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```



Insert 82

Insertion Example – 82

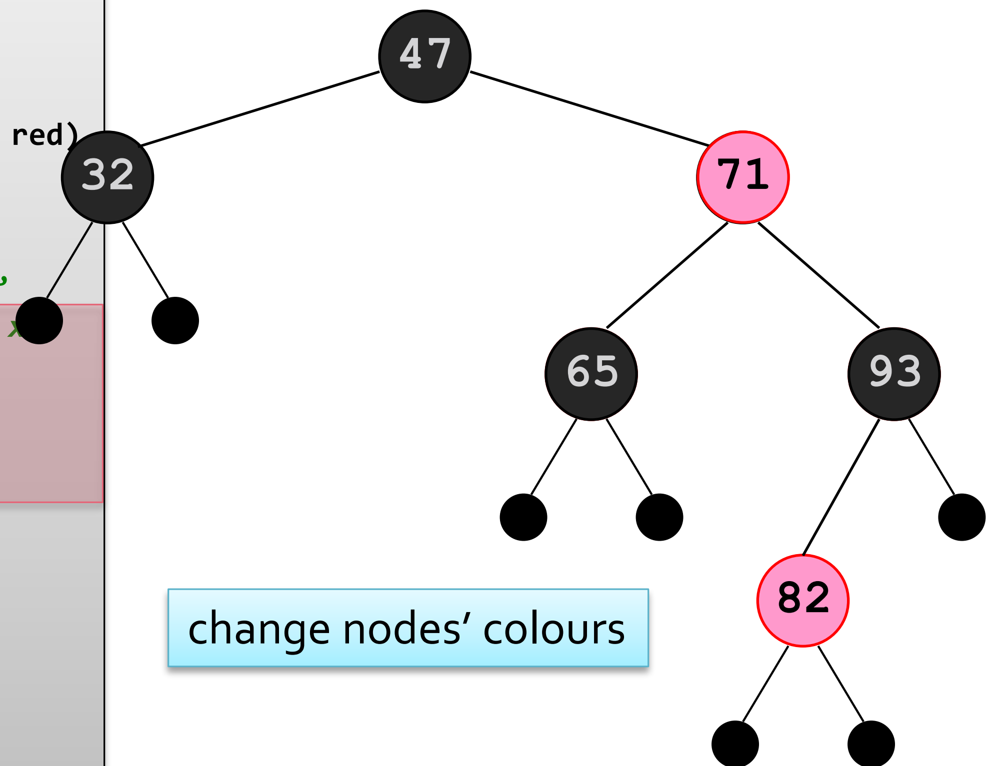
```
rbInsert(x)
    bstInsert(x)
    x.colour = red
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //x's "uncle"
            if (y.colour == red) //like x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```



Insert 82

Insertion Example – 82

```
rbInsert(x)
    bstInsert(x)
    x.colour = red
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            … //symmetric to else
        else
            y = x.p.p.left //x's "uncle"
            if (y.colour == red) //like x
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.left)
                    x = x.p
                    right_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                left_rotate(x.p.p)
    end while
    root.colour = black
```

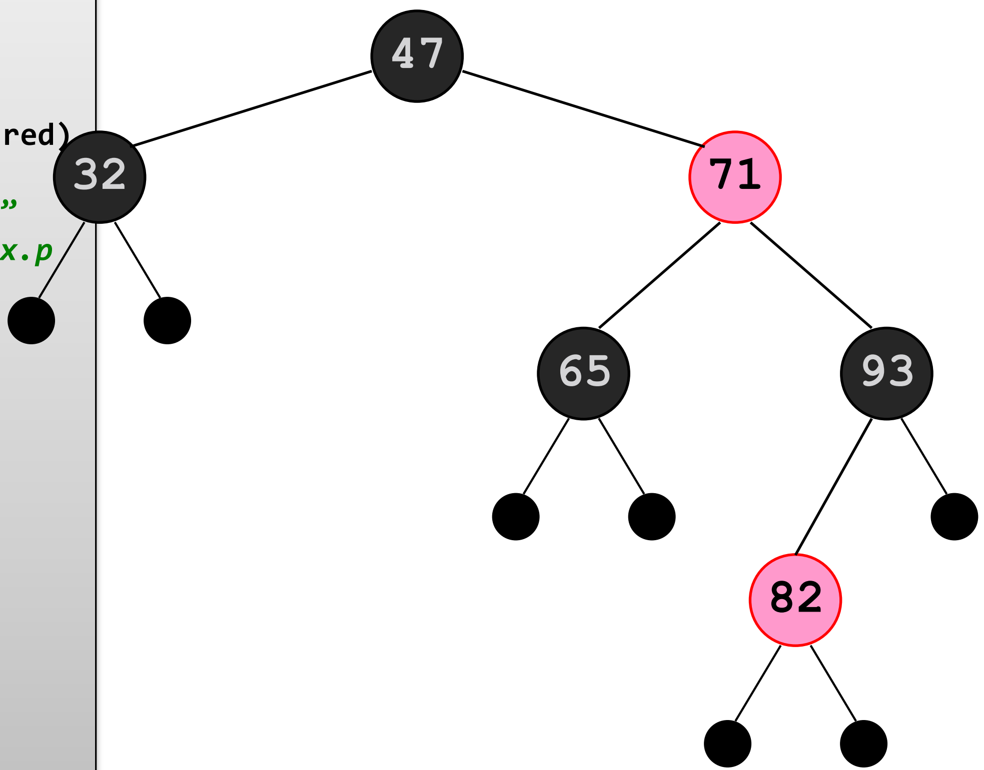


change nodes' colours

Insert 82

Insertion Example – 87

```
rbInsert(x)
   bstInsert(x)
   x.colour = red
   while (x != root and x.p.colour == red)
      if (x.p == x.p.p.left)
         y = x.p.p.right //x's "uncle"
         if (y.colour == red) //like x.p
            x.p.colour = black
            y.colour = black
            x.p.p = red
            x = x.p.p
         else //y.colour == black
            if (x == x.p.right)
               x = x.p
               left_rotate(x)
            x.p.colour = black
            x.p.p.colour = red
            right_rotate(x.p.p)
      else
         … //symmetric to if
   end while
   root.colour = black
```

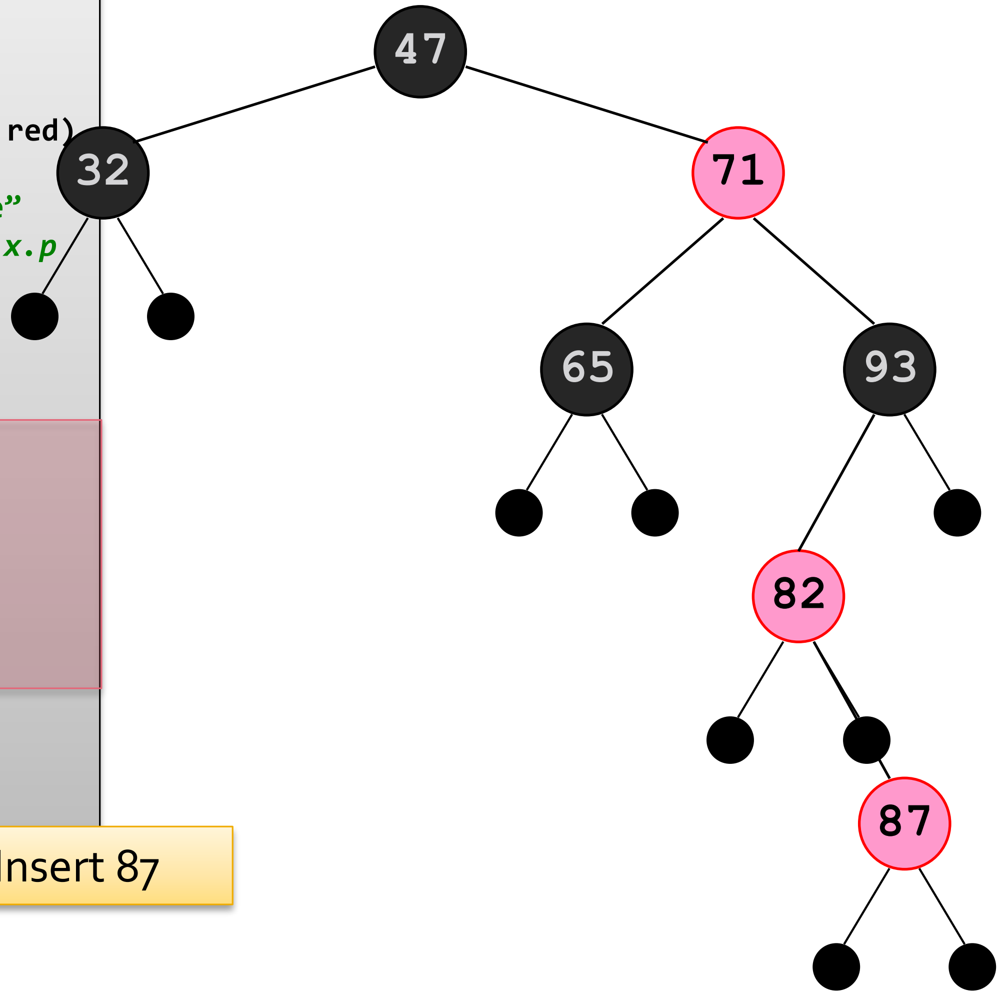


Insert 87

Insertion Example – 87

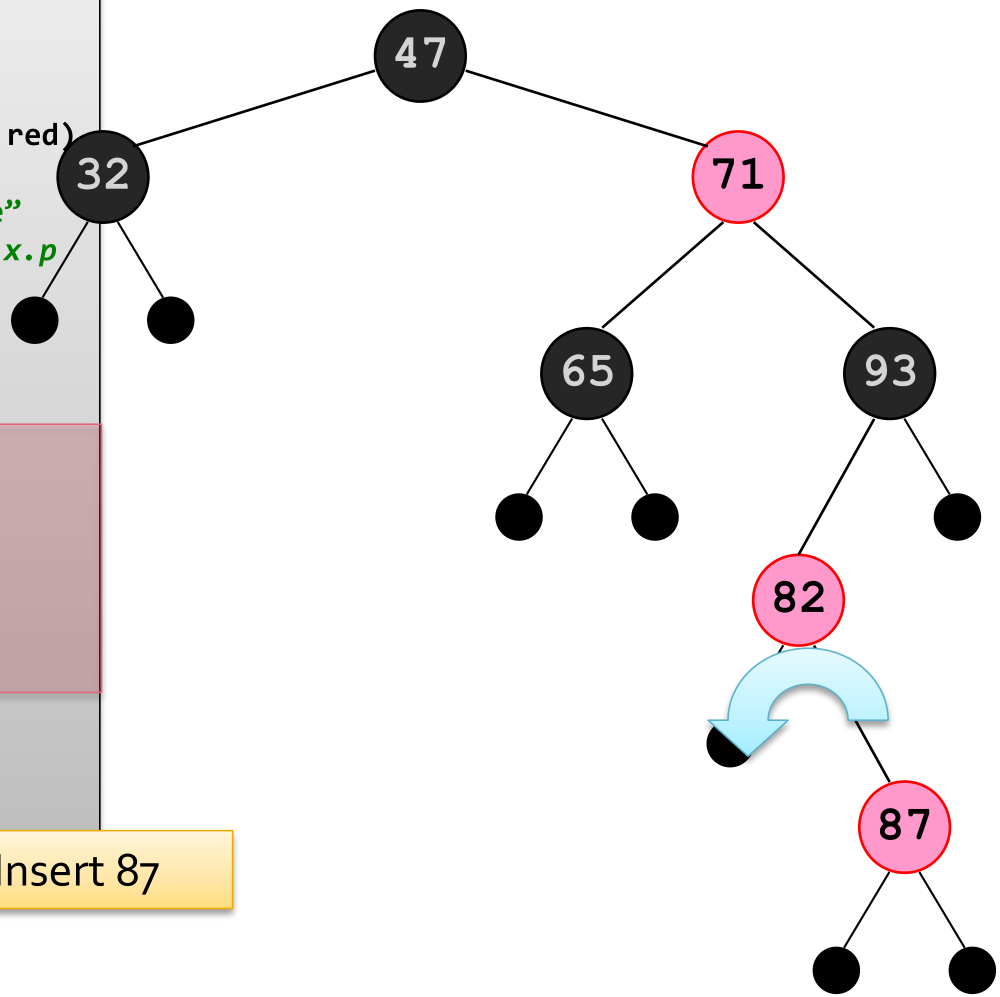
```
rbInsert(x)
   bstInsert(x)
   x.colour = red
   while (x != root and x.p.colour == red)
      if (x.p == x.p.p.left)
         y = x.p.p.right //x's "uncle"
         if (y.colour == red) //like x.p
            x.p.colour = black
            y.colour = black
            x.p.p = red
            x = x.p.p
         else //y.colour == black
            if (x == x.p.right)
               x = x.p
               left_rotate(x)
            x.p.colour = black
            x.p.p.colour = red
            right_rotate(x.p.p)
      else
         … //symmetric to if
   end while
   root.colour = black
```

Insert 87



Insertion Example – 87

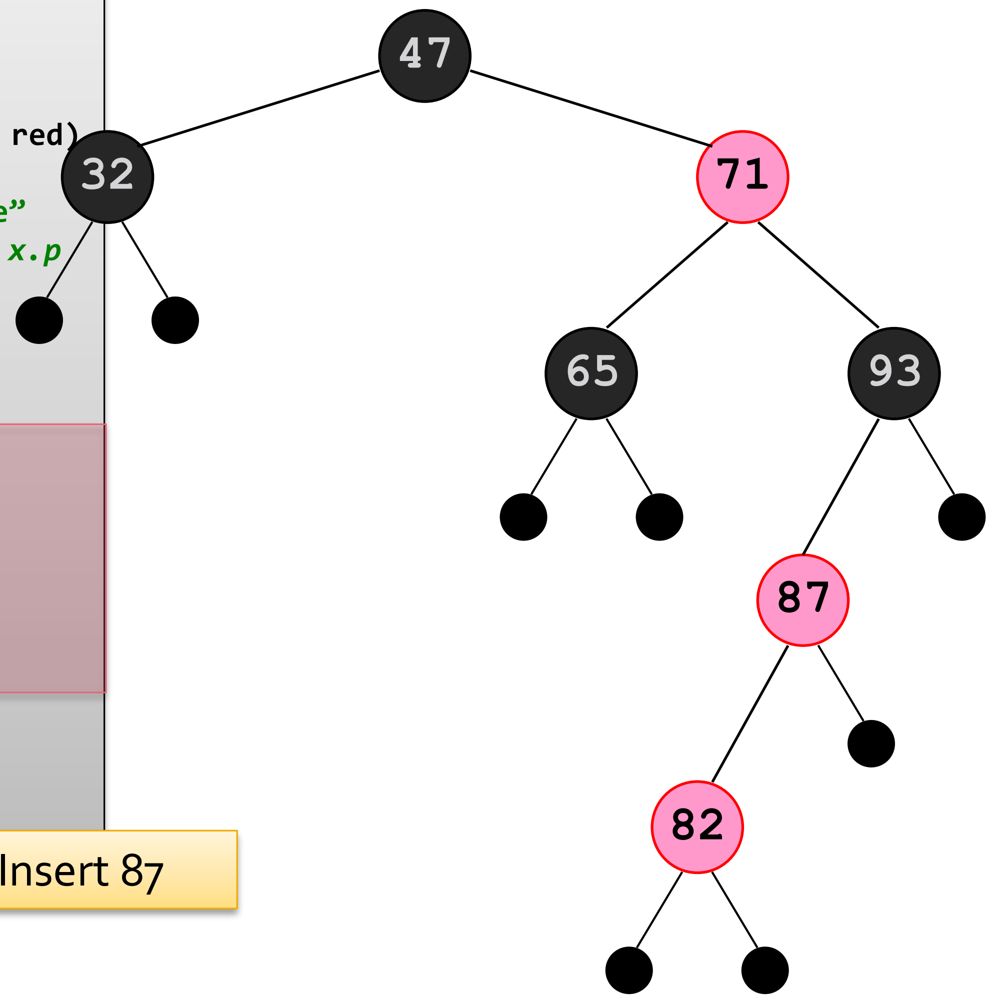
```
rbInsert(x)
   bstInsert(x)
   x.colour = red
   while (x != root and x.p.colour == red)
      if (x.p == x.p.p.left)
         y = x.p.p.right //x's "uncle"
         if (y.colour == red) //like x.p
            x.p.colour = black
            y.colour = black
            x.p.p = red
            x = x.p.p
         else //y.colour == black
            if (x == x.p.right)
               x = x.p
               left_rotate(x)
            x.p.colour = black
            x.p.p.colour = red
            right_rotate(x.p.p)
      else
         … //symmetric to if
   end while
   root.colour = black
```



Insert 87

Insertion Example – 87

```
rbInsert(x)
    bstInsert(x)
    x.colour = red
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //x’s “uncle”
            if (y.colour == red) //like x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```



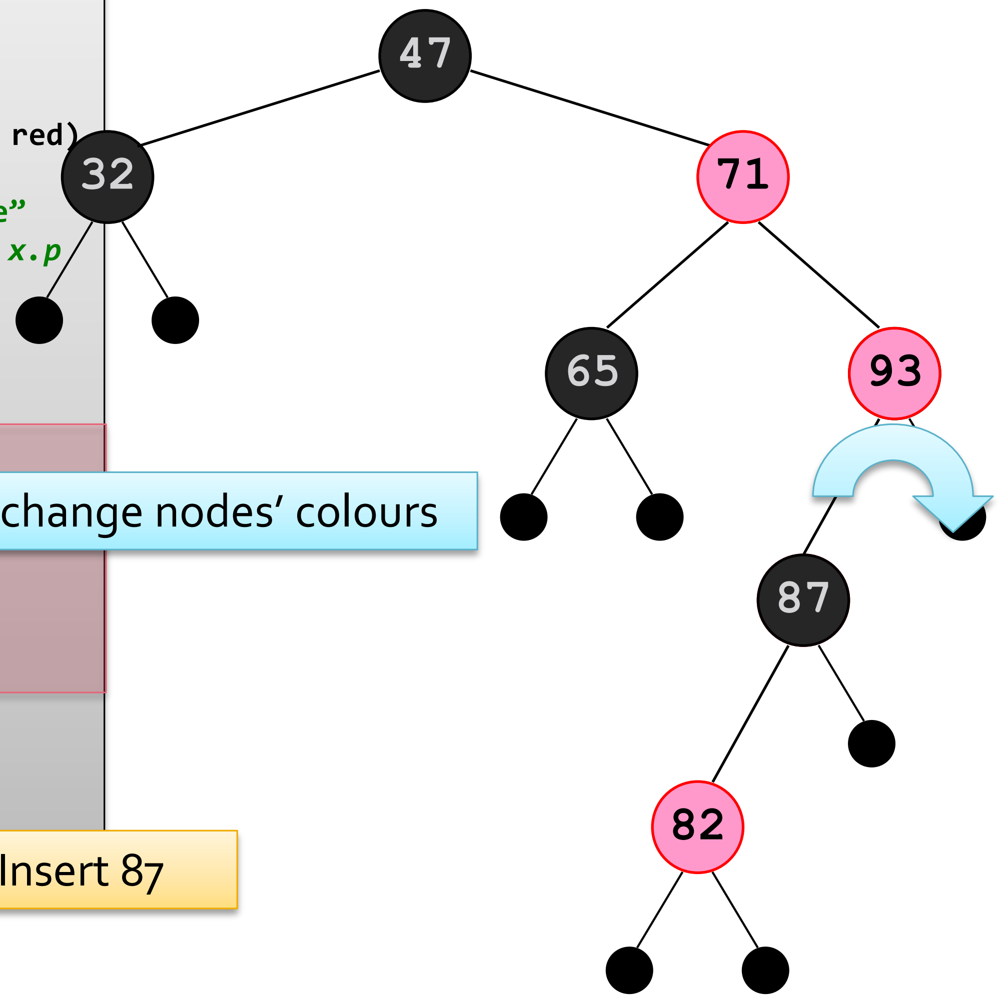
Insert 87

Insertion Example – 87

```
rbInsert(x)
    bstInsert(x)
    x.colour = red
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //x's "uncle"
            if (y.colour == red) //like x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```

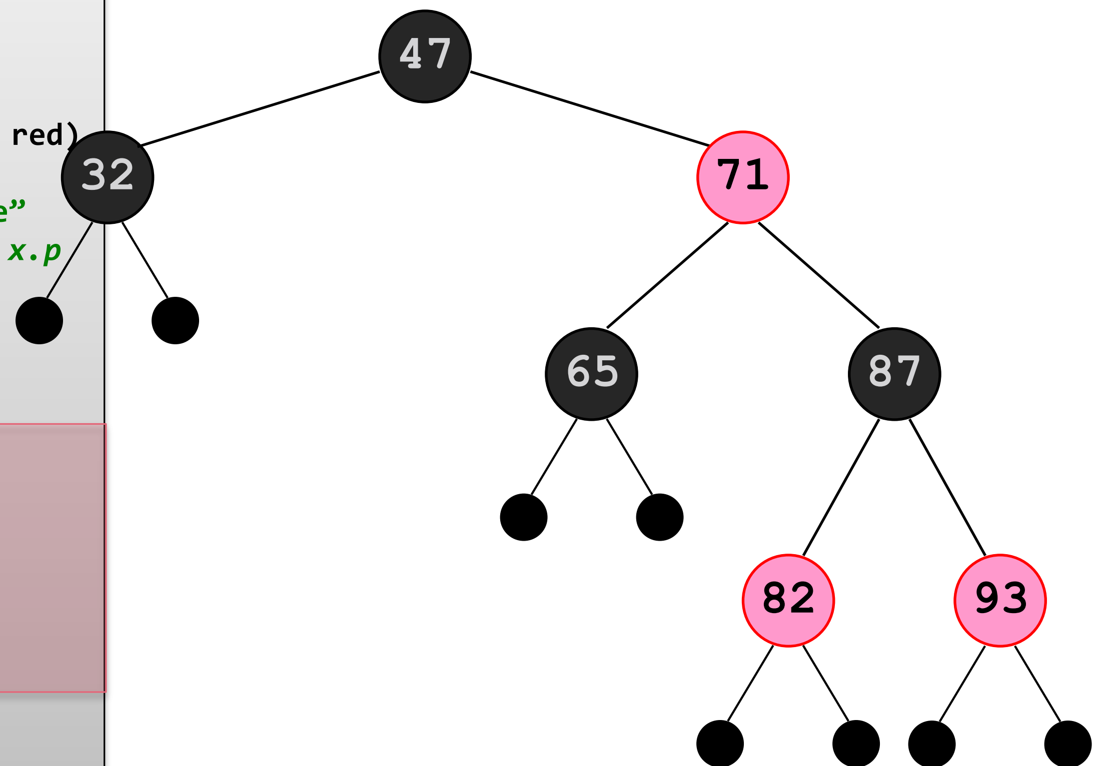
change nodes' colours

Insert 87



Insertion Example – 87

```
rbInsert(x)
    bstInsert(x)
    x.colour = red
    while (x != root and x.p.colour == red)
        if (x.p == x.p.p.left)
            y = x.p.p.right //x's "uncle"
            if (y.colour == red) //like x.p
                x.p.colour = black
                y.colour = black
                x.p.p = red
                x = x.p.p
            else //y.colour == black
                if (x == x.p.right)
                    x = x.p
                    left_rotate(x)
                x.p.colour = black
                x.p.p.colour = red
                right_rotate(x.p.p)
        else
            … //symmetric to if
    end while
    root.colour = black
```



Insert 87

red-black Tree Removal

- Modifies the standard *bst* removal algorithm slightly
  - If the removed node is be replaced by its predecessor replace its *data*, rather than the entire node
    - The node's colour remains the same
  - Then remove the predecessor

> If the target node had two children the predecessor is removed

- If the removed node was black then *fix* the tree
  - The removed node's *child* is passed to the tree fix algorithm
    - This child may be a (black) imaginary (null) child
    - In practice the removed node's child, its parent and whether the removed node was a left or a right child is required

Fixing a red-black Tree

- Tree-fix colours its node parameter, *x*, black
  - This corrects the violation to the black height property caused by removing a black node
  - If *x* used to be red it is now black and the tree is fixed
- If *x* was black then it becomes "doubly **black**"
  - Violating the property that nodes are red or black
  - The extra black colour is pushed up the tree until
    - A red node is reached, when it is made black
    - The root node is reached or
    - The tree can be rotated and re-coloured to fix the problem

Tree Fix Summary

- The algorithm to fix a red-black tree after deletion has four cases
  1. Colours a red sibling of *x* black, which converts it into one of the other three cases

     nephews?
  2. Both of *x*'s sibling's children are black
  3. One of *x*'s sibling's children is black
     - Either x is a left child and y's right sibling is black or x is a right child and y's left sibling is black
  4. One of *x*'s sibling's children is black
     - Either x is a left child and y's left sibling is black or x is a right child and y's right sibling is black

BST Removal Algorithm

z is the node that contains the data to be removed

finding it is not shown

```
rbRemove(z)
    if (z.left == null or z.right == null)
        y = z //node to be removed
    else
        y = predecessor(z) //or successor
    if (y.left != null)
        x = y.left
    else
        x = y.right
    x.p = y.p //detach x from y (if x is not null)
    if (y.p == null) //y is the root
        root = x
    else
        // Attach x to y's parent
        if (y == y.p.left) //left child
            y.p.left = x
        else
            y.p.right = x
    if (y != z) //i.e. y has, conceptually, been moved up
        z.data = y.data //replace z with y
    if (y.colour == black)
        rbFix(x) //note that x could be null
```

if *z* has one or no children

z has two children

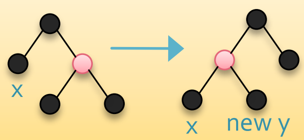
identify if *y*'s only child is right or left

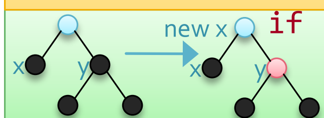
y is the predecessor

so, in practice, requires more information

Tree Fix Algorithm

```
rbFix(x)see note
    while (x != root and x.colour = black)
        if (x == x.p.left) //x is left child
            y = x.p.right //x's sibling
            if (y.colour == red)
                y.colour = black
                x.p.colour = red //x's parent must have been black since y is red
                left_rotate(x.p)
                y = x.p.right
            if (y.left.colour == black and y.right.colour == black)
                y.colour = red
                x = x.p //and into while again …
            else
                …
        else
            … //symmetric to if
    x.colour = black
```





the algorithm is trying to correct the black height of the tree since a black node has been removed

the black height of all nodes is unchanged but x's sibling is now black

by making *y* red this makes the sibling's subtree the same black height, so then push the fix up the tree

since we've found a node that is red fix black height by making it black

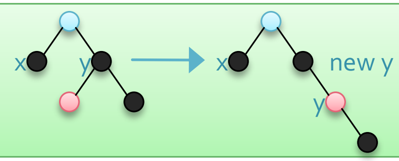
Implementation note: *x* may be null so 3 parameters are required: *x*, *x*'s parent and whether the removed node was a left or right child

```
rbFix(x)
    while (x != root and x.colour = black)
        if (x == x.p.left) //x is left child
            y = x.p.right //x's sibling
            if (y.colour == red)
                y.colour = black
                x.p.colour = red //p was black
                left_rotate(x.p)
                y = x.p.right
            if (y.left.colour == black and y.right.colour == black)
                y.colour = red
                x = x.p //and into while again …
            else
                if (y.right.colour == black)
                    y.left.colour = black
                    y.colour = red
                    right_rotate(y)
                    y = x.p.right
                y.colour = x.p.colour
                x.p.colour = black
                y.right.colour = black
                left_rotate(x.p)
                x = root
        else
            … //symmetric
```

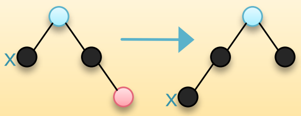
Tree Fix Algorithm

```
rbFix(x)
    while (x != root and x.colour = black)
        if (x == x.p.left) //x is left child
            y = x.p.right //x's sibling
            if (y.colour == red)
                …
            else
                if (y.right.colour == black)
                    y.left.colour = black
                    y.colour = red
                    right_rotate(y)
                    y = x.p.right
                y.colour = x.p.colour
                x.p.colour = black
                y.right.colour = black
                left_rotate(x.p)
                x = root
        else
            … //symmetric to if
    x.colour = black
```

makes *x*'s sibling black or pushes problem up tree



makes *x*'s sibling's right child red

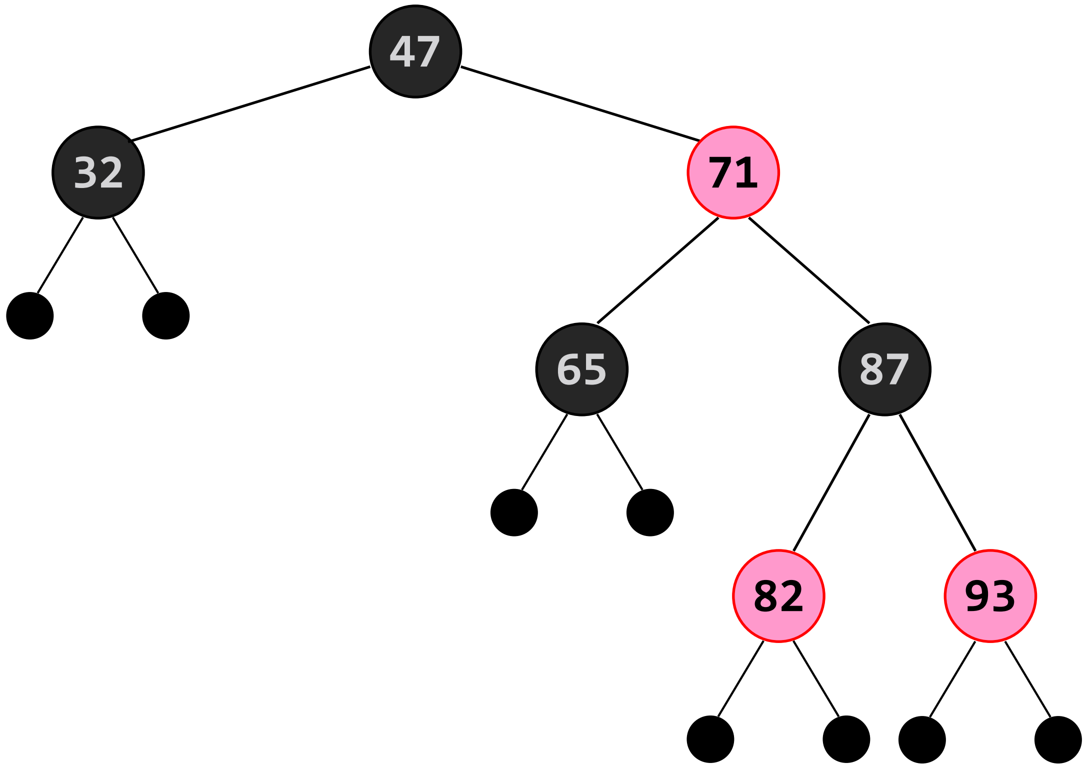


fixed!

```
rbFix(x)
    while (x != root and x.colour = black)
        if (x == x.p.left) //x is left child
            y = x.p.right //x's sibling
            if (y.colour == red)
                y.colour = black
                x.p.colour = red //p was black
                left_rotate(x.p)
                y = x.p.right
            if (y.left.colour == black and y.right.colour == black)
                y.colour = red
                x = x.p //and into while again …
            else
                if (y.right.colour == black)
                    y.left.colour = black
                    y.colour = red
                    right_rotate(y)
                    y = x.p.right
                y.colour = x.p.colour
                x.p.colour = black
                y.right.colour = black
                left_rotate(x.p)
                x = root
        else
            … //symmetric
```

Removal Example 1

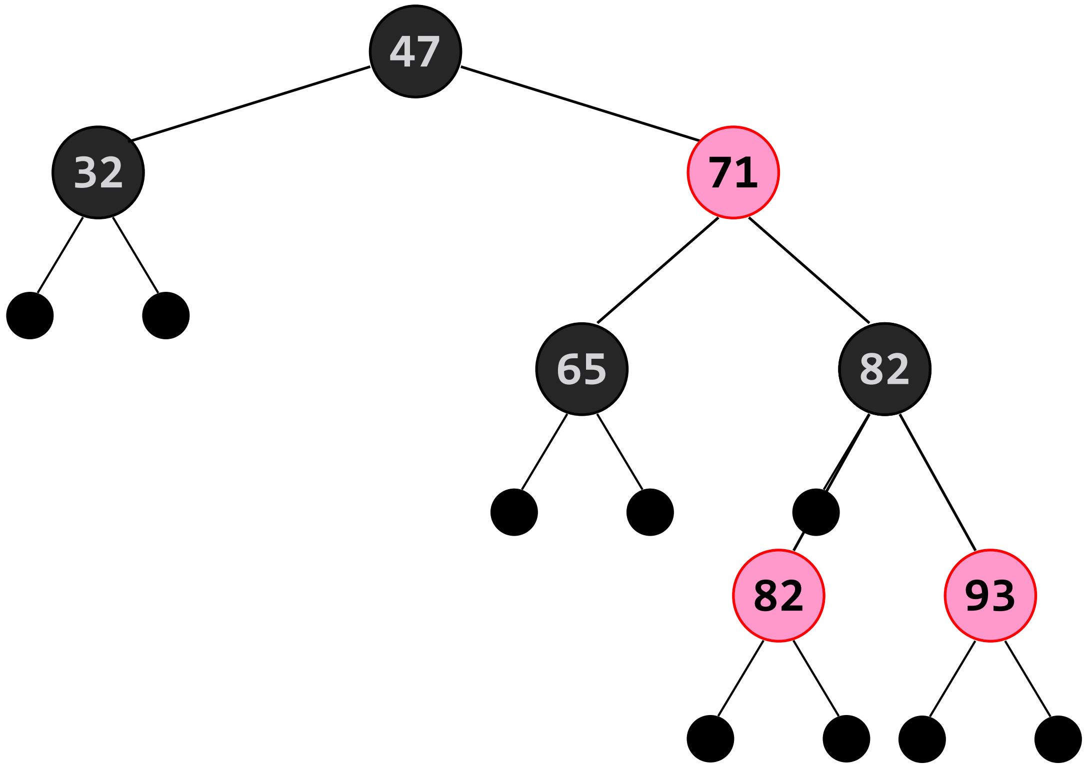
Remove 87




```
rbRemove(z)
   if (z.left == null or z.right == null)
      y = z //node to be removed
   else
      y = predecessor(z) //or successor
   if (y.left != null)
      x = y.left
   else
      x = y.right
   x.p = y.p //detach x from y; if not null
   if (y.p == null) //y is the root
      root = x
   else
      // Attach x to y's parent
      if (y == y.p.left) //left child
         y.p.left = x
      else
         y.p.right = x
   if (y != z) //i.e. y moved up
      z.data = y.data //replace z with y
   if (y.colour == black)
      rbFix(x) //note that x could be null
```

Removal Example 1

Remove 87



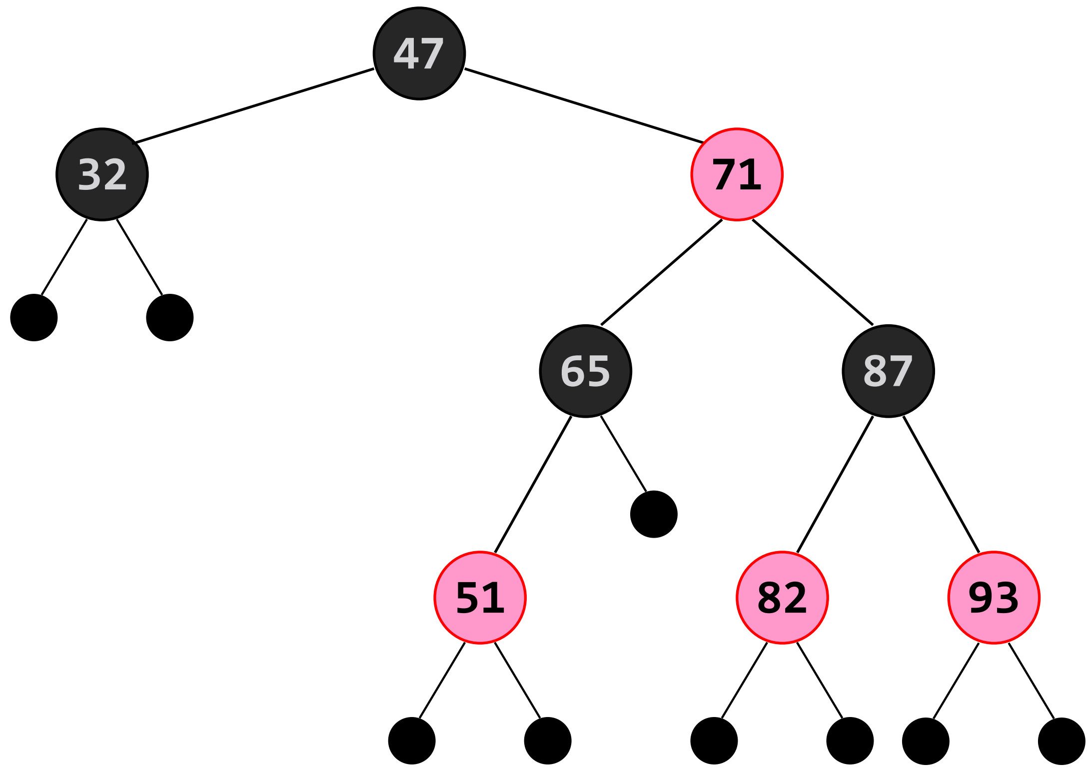

```
rbRemove(z)
    if (z.left == null or z.right == null)
        y = z //node to be removed
    else
        y = predecessor(z) //or successor
    if (y.left != null)
        x = y.left
    else
        x = y.right
    x.p = y.p //detach x from y; if not null
    if (y.p == null) //y is the root
        root = x
    else
        // Attach x to y's parent
        if (y == y.p.left) //left child
            y.p.left = x
        else
            y.p.right = x
    if (y != z) //i.e. y moved up
        z.data = y.data //replace z with y
    if (y.colour == black)
        rbFix(x) //note that x could be null
```

Replace data with predecessor

Predecessor is red so no violation

Removal Example 2

Remove 71



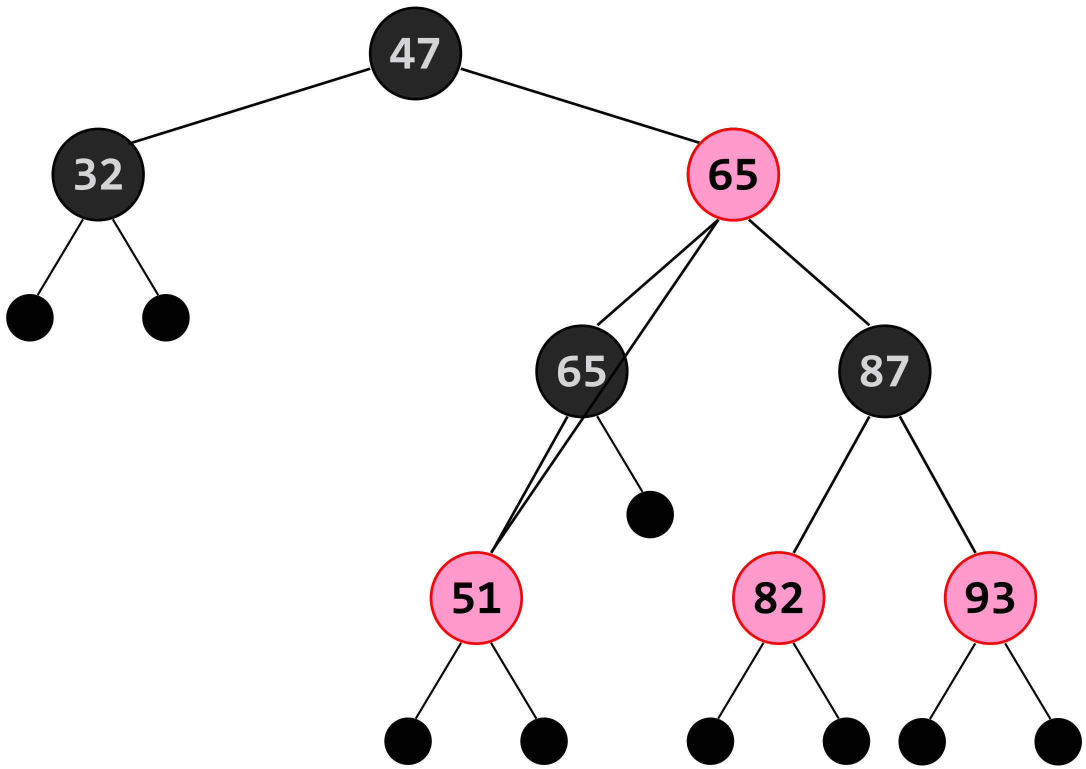

```
rbRemove(z)
    if (z.left == null or z.right == null)
        y = z //node to be removed
    else
        y = predecessor(z) //or successor
    if (y.left != null)
        x = y.left
    else
        x = y.right
    x.p = y.p //detach x from y; if not null
    if (y.p == null) //y is the root
        root = x
    else
        // Attach x to y's parent
        if (y == y.p.left) //left child
            y.p.left = x
        else
            y.p.right = x
    if (y != z) //i.e. y moved up
        z.data = y.data //replace z with y
    if (y.colour == black)
        rbFix(x) //note that x could be null
```

Removal Example 2

Remove 71

Replace with predecessor

Attach predecessor's child




```
rbRemove(z)
    if (z.left == null or z.right == null)
        y = z //node to be removed
    else
        y = predecessor(z) //or successor
    if (y.left != null)
        x = y.left
    else
        x = y.right
    x.p = y.p //detach x from y; if not null
    if (y.p == null) //y is the root
        root = x
    else
        // Attach x to y's parent
        if (y == y.p.left) //left child
            y.p.left = x
        else
            y.p.right = x
    if (y != z) //i.e. y moved up
        z.data = y.data //replace z with y
    if (y.colour == black)
        rbFix(x) //note that x could be null
```

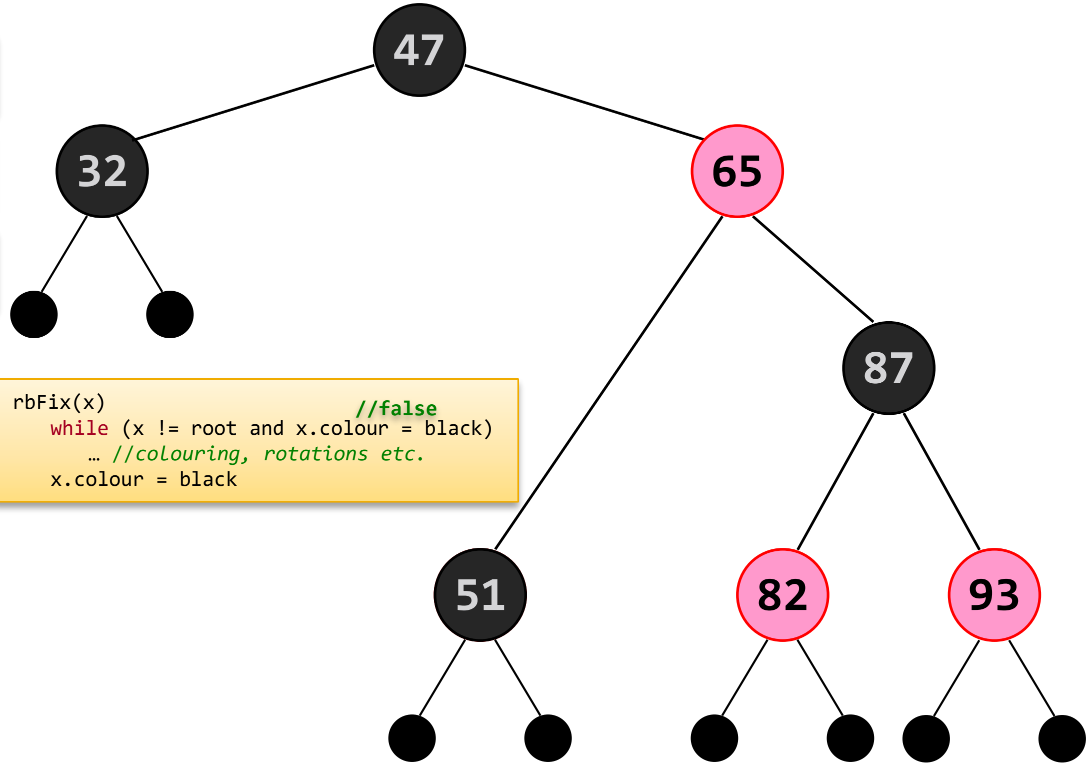
Removal Example 2

Remove 71

Replace with predecessor

Attach predecessor's child

Fix tree – make 51 black

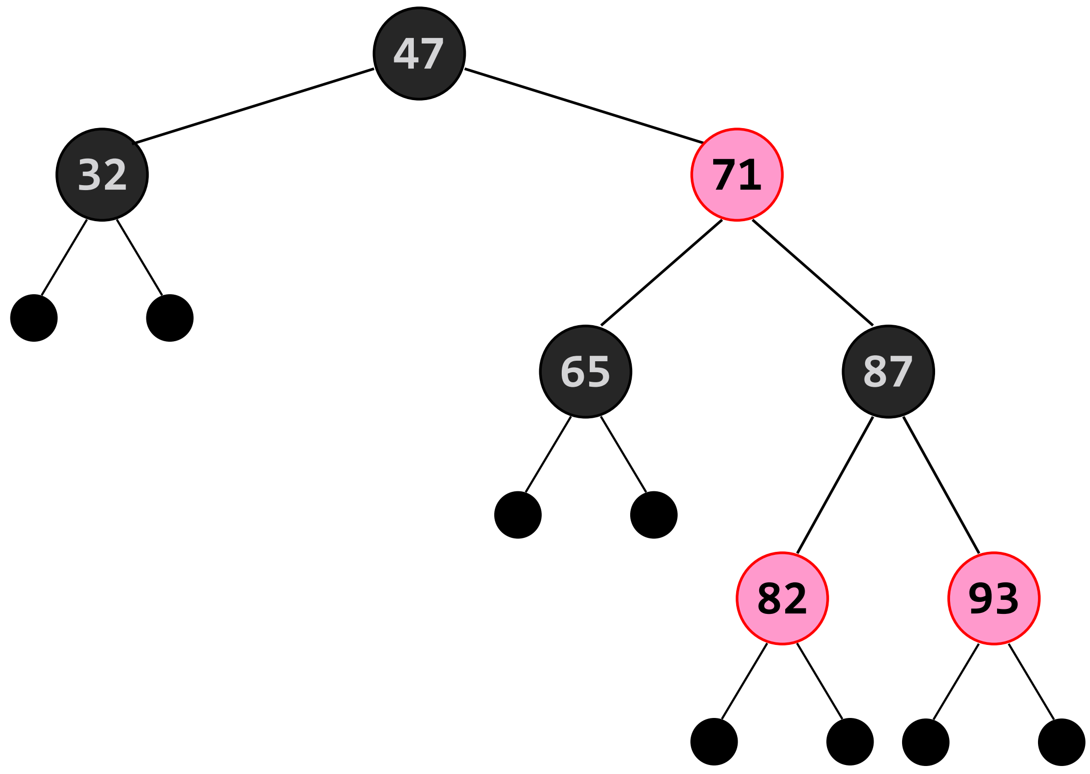



```
rbRemove(z)
    if (z.left == null or z.right == null)
        y = z //node to be removed
    else
        y = predecessor(z) //or successor
    if (y.left != null)
        x = y.left
    else
        x = y.right
    x.p = y.p //detach x from y; if not null
    if (y.p == null) //y is the root
        root = x
    else
        // Attach x to y's parent
        if (y == y.p.left) //left child
            y.p.left = x
        else
            y.p.right = x
    if (y != z) //i.e. y moved up
        z.data = y.data //replace z with y
    if (y.colour == black)
        rbFix(x) //note that x could be null
```

```
rbFix(x)                              //false
    while (x != root and x.colour = black)
        … //colouring, rotations etc.
    x.colour = black
```

Removal Example 3

Remove 32




```
rbRemove(z)
   if (z.left == null or z.right == null)
      y = z //node to be removed
   else
      y = predecessor(z) //or successor
   if (y.left != null)
      x = y.left
   else
      x = y.right
   x.p = y.p //detach x from y; if not null
   if (y.p == null) //y is the root
      root = x
   else
      // Attach x to y's parent
      if (y == y.p.left) //left child
         y.p.left = x
      else
         y.p.right = x
   if (y != z) //i.e. y moved up
      z.data = y.data //replace z with y
   if (y.colour == black)
      rbFix(x) //note that x could be null
```

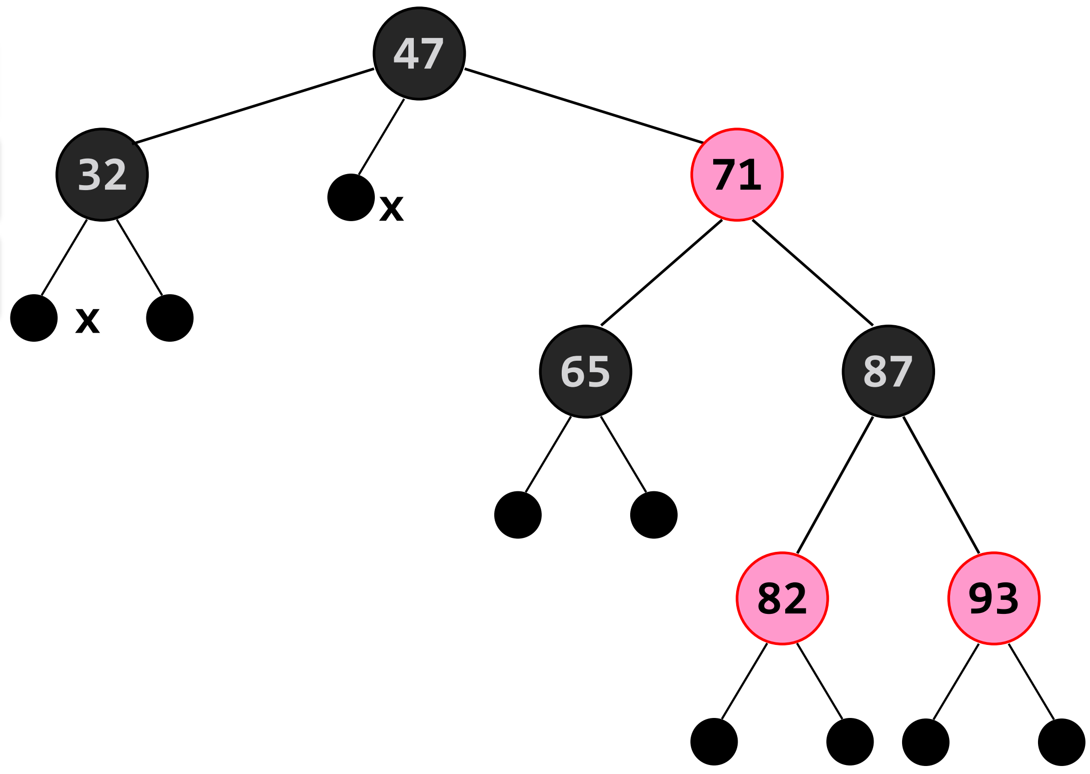
Removal Example 3

Remove 32

Identify node's left child, *x*

Remove target node

Attach *x* to target's parent




```
rbRemove(z)
   if (z.left == null or z.right == null)
      y = z //node to be removed
   else
      y = predecessor(z) //or successor
   if (y.left != null)
      x = y.left
   else
      x = y.right
   x.p = y.p //detach x from y; if not null
   if (y.p == null) //y is the root
      root = x
   else
      // Attach x to y's parent
      if (y == y.p.left) //left child
         y.p.left = x
      else
         y.p.right = x
   if (y != z) //i.e. y moved up
      z.data = y.data //replace z with y
   if (y.colour == black)
      rbFix(x) //note that x could be null
```

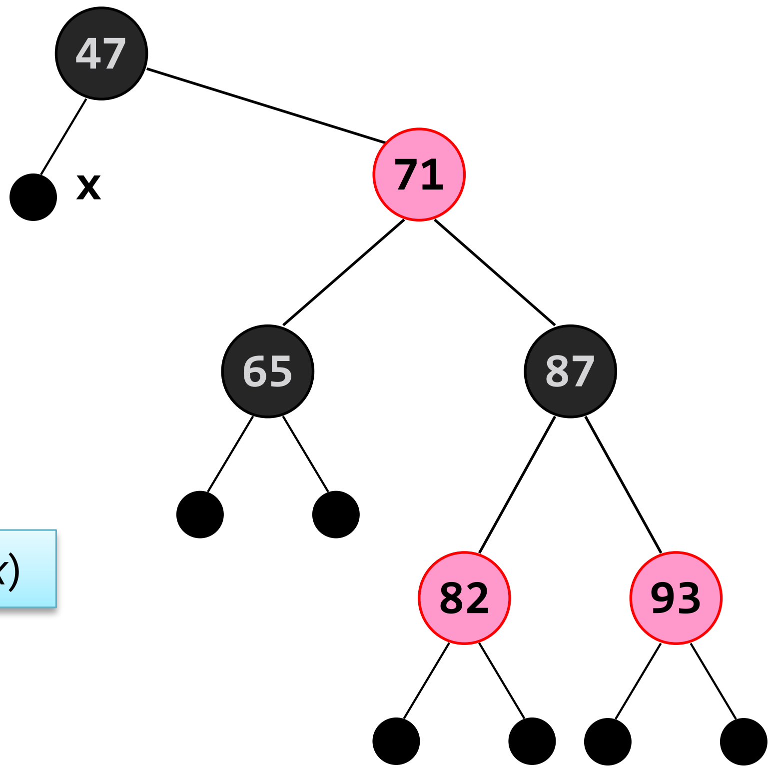
Removal Example 3

Remove 32

Identify node's left child, x

Remove target node

Attach x to target's parent




```
rbRemove(z)
   if (z.left == null or z.right == null)
      y = z //node to be removed
   else
      y = predecessor(z) //or successor
   if (y.left != null)
      x = y.left
   else
      x = y.right
   x.p = y.p //detach x from y; if not null
   if (y.p == null) //y is the root
      root = x
   else
      // Attach x to y's parent
      if (y == y.p.left) //left child
         y.p.left = x
      else
         y.p.right = x
   if (y != z) //i.e. y moved up
      z.data = y.data //replace z with y
   if (y.colour == black)
      rbFix(x) //note that x could be null
```

Fix the tree (passing *x*)

Removal Example 3

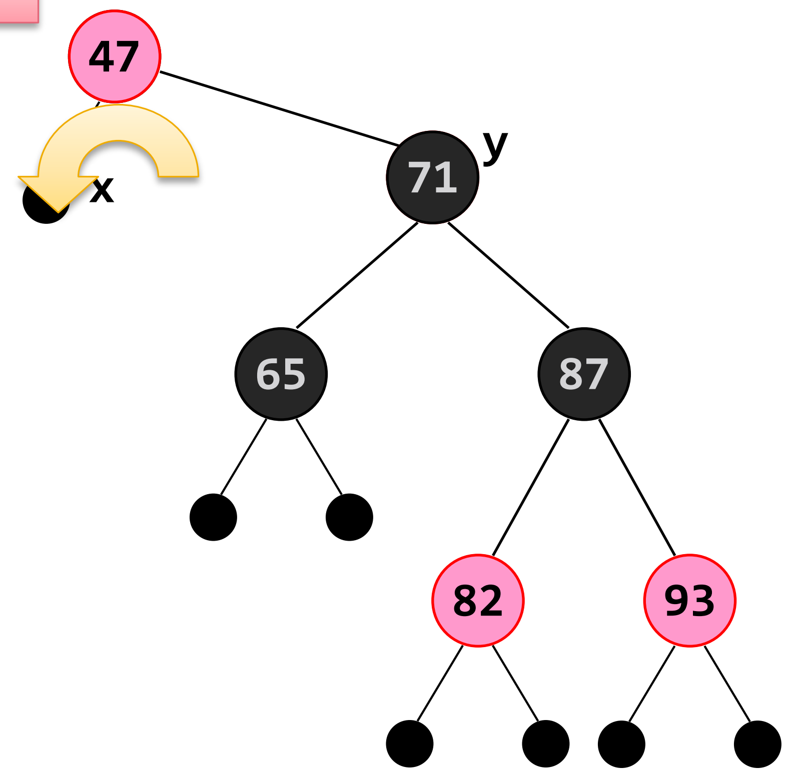
Remove 32

Calling TreeFix on *x*

Identify *y*, *x*'s sibling

Set *y* black, *y*'s parent red

Left rotate *x*'s parent




```
rbFix(x)
    while (x != root and x.colour = black)
        if (x == x.p.left) //x is left child
            y = x.p.right //x's sibling
            if (y.colour == red)
                y.colour = black
                x.p.colour = red //p was black
                left_rotate(x.p)
                y = x.p.right
            if (y.left.colour == black and y.right.colour == black)
                y.colour = red
                x = x.p //and into while again …
            else
                if (y.right.colour == black)
                    y.left.colour = black
                    y.colour = red
                    right_rotate(y)
                    y = x.p.right
                y.colour = x.p.colour
                x.p.colour = black
                y.right.colour = black
                left_rotate(x.p)
                x = root
        else
            … //symmetric
```

Removal Example 3

Remove 32

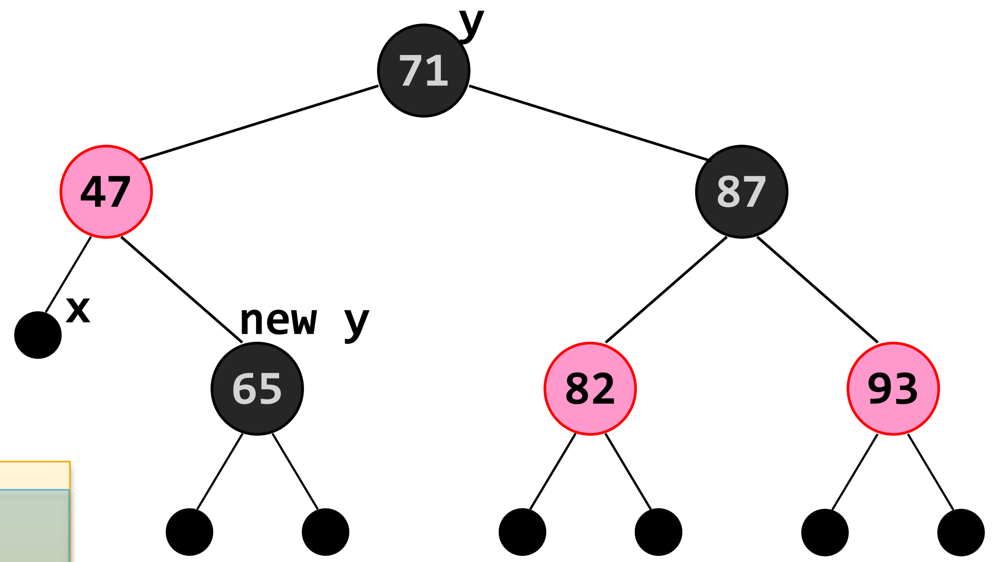
Identify y, x's sibling

Set y black, y's parent red

Left rotate x's parent

Identify *y*: *x*'s new sibling

```
rbFix(x)
    while (x != root and x.colour = black)
        if (x == x.p.left) //x is left child
            y = x.p.right //x's sibling
            if (y.colour == red)
                y.colour = black
                x.p.colour = red //p was black
                left_rotate(x.p)
                y = x.p.right
            if (y.left.colour == black and y.right.colour == black)
                y.colour = red
                x = x.p //and into while again …
            else
                if (y.right.colour == black)
                    y.left.colour = black
                    y.colour = red
                    right_rotate(y)
                    y = x.p.right
                y.colour = x.p.colour
                x.p.colour = black
                y.right.colour = black
                left_rotate(x.p)
                x = root
        else
            … //symmetric
```



Removal Example 3

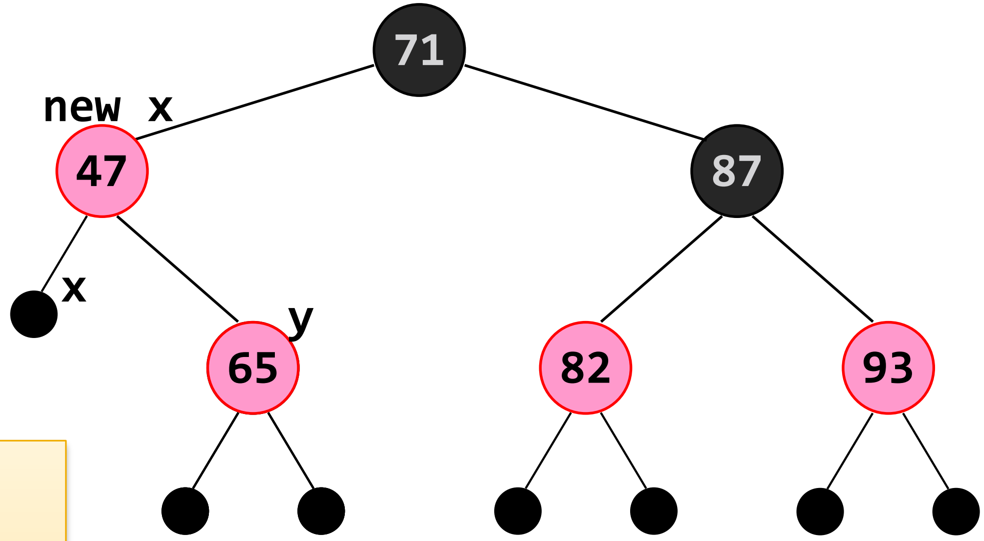
Remove 32

Identify y, x's sibling

Set y black, y's parent red

Left rotate x's parent

Identify y: x's new sibling




```
rbFix(x)
    while (x != root and x.colour = black)
        if (x == x.p.left) //x is left child
            y = x.p.right //x's sibling
            if (y.colour == red)
                y.colour = black
                x.p.colour = red //p was black
                left_rotate(x.p)
                y = x.p.right
            if (y.left.colour == black and y.right.colour == black)
                y.colour = red
                x = x.p //and into while again …
            else
                if (y.right.colour == black)
                    y.left.colour = black
                    y.colour = red
                    right_rotate(y)
                    y = x.p.right
                y.colour = x.p.colour
                x.p.colour = black
                y.right.colour = black
                left_rotate(x.p)
                x = root
        else
            … //symmetric
```

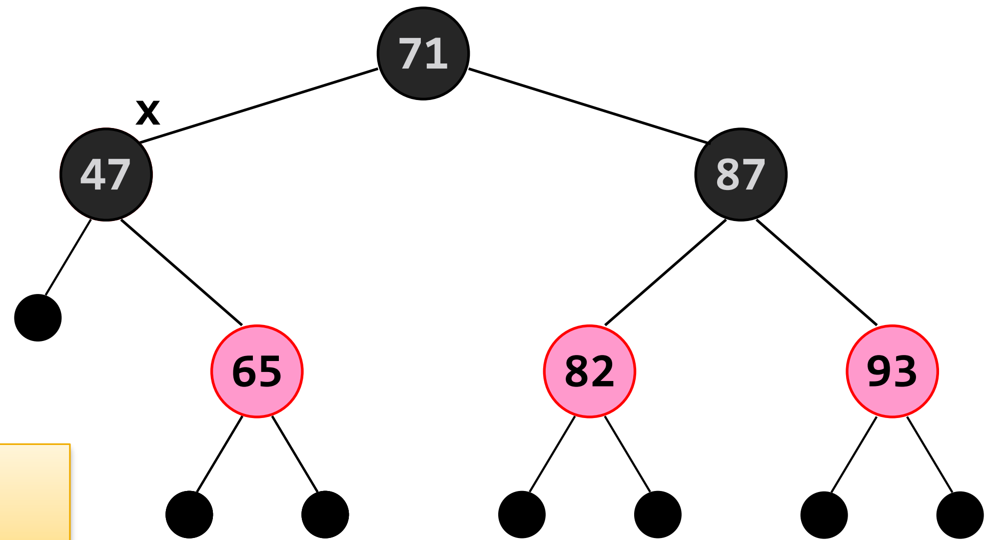
Colour y red

Assign *x* it's parent and repeat while

Removal Example 3

Remove 32

Colour x black




```
rbFix(x)                        //false
   while (x != root and x.colour = black)
      … //colouring, rotations etc.
   x.colour = black
```