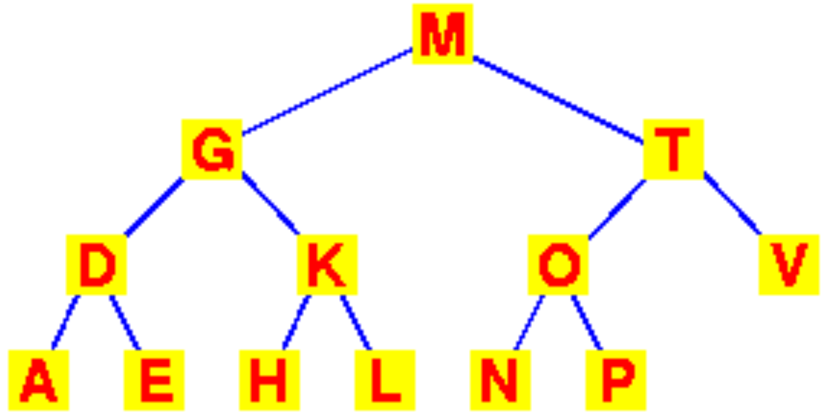
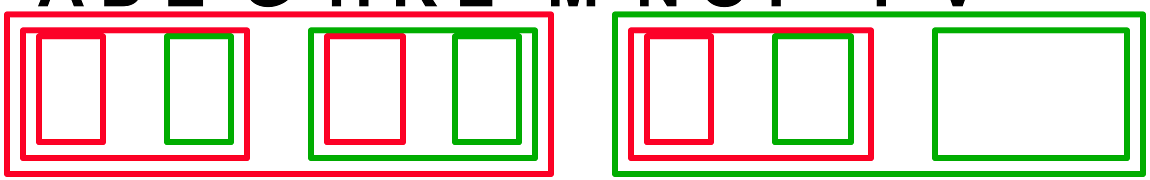
Data Structures and Algorithms (11)
CS F211

Trees - Searching

- **Binary search tree**
  - **Produces a sorted list by in-order traversal**

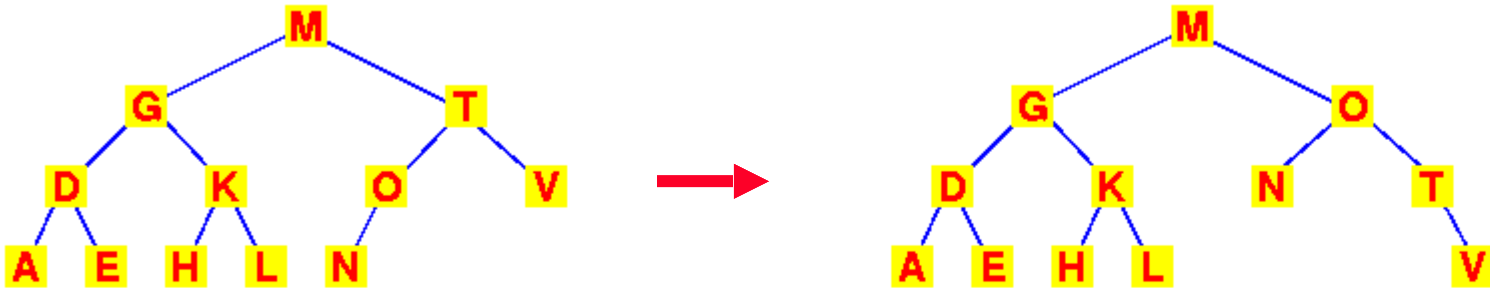


- **In order: A D E G H K L M N O P T V**



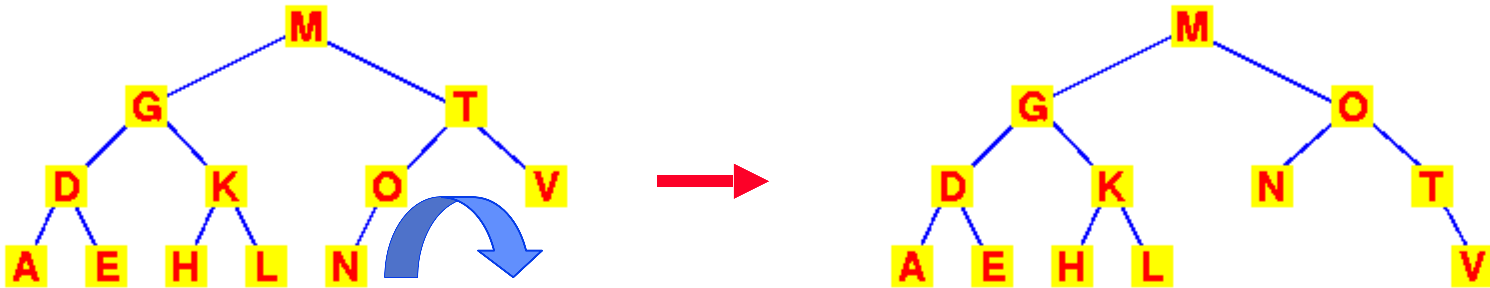
Trees - Searching

- **Binary search tree**
  - **Preserving the order**
  - **Observe that this transformation preserves the search tree**



Trees - Searching

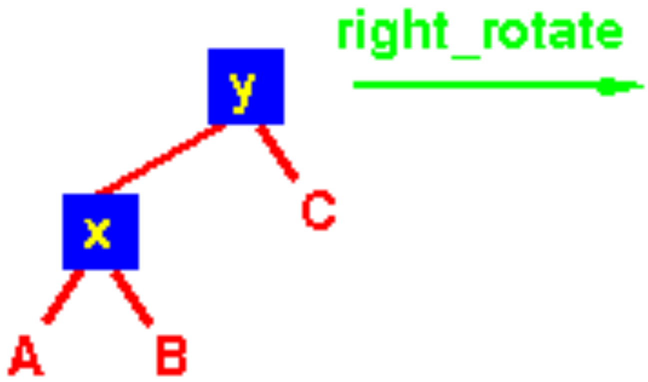
- **Binary search tree**
  - **Preserving the order**
  - **Observe that this transformation preserves the search tree**



- **We've performed a rotation of the sub-tree about the T and O nodes**

Trees - Rotations

- **Binary search tree**
  - **Rotations can be either left- or right-rotations**

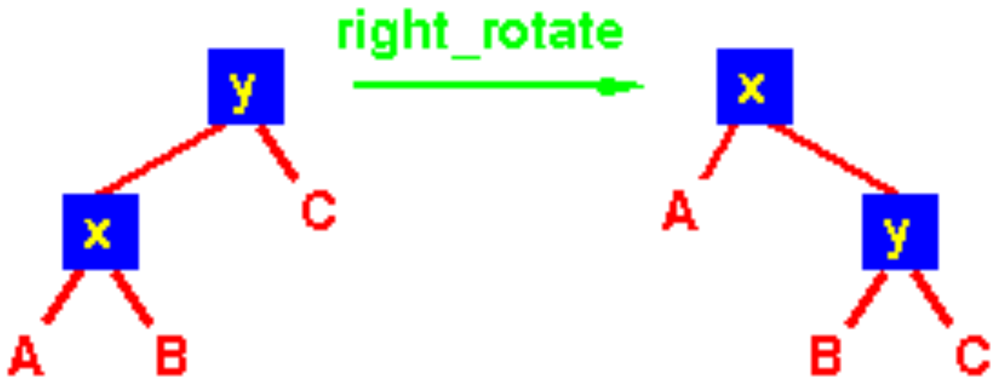


  - **For both trees: the inorder traversal is**

A x B y C

Trees - Rotations

- **Binary search tree**
  - **Rotations can be either left- or right-rotations**

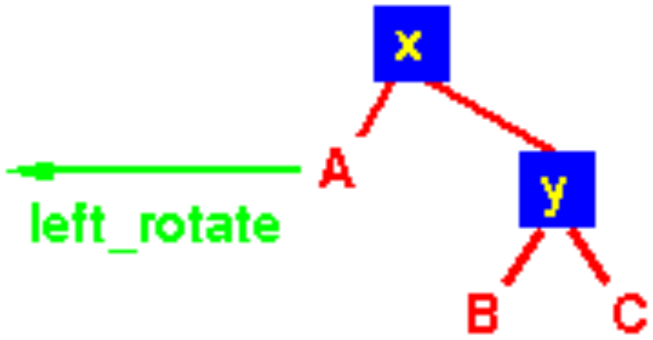


  - **For both trees: the inorder traversal is**

    A x B y C

Trees - Rotations

- **Binary search tree**
  - **Rotations can be either left- or right-rotations**

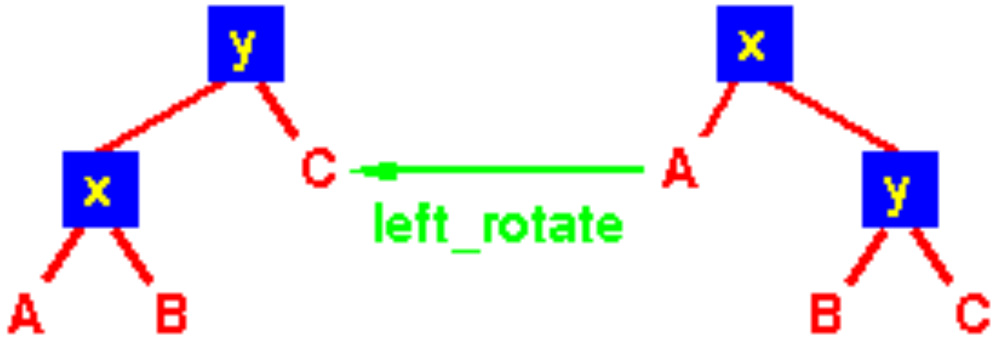


  - **For both trees: the inorder traversal is**

A x B y C

Trees - Rotations

- **Binary search tree**
  - **Rotations can be either left- or right-rotations**

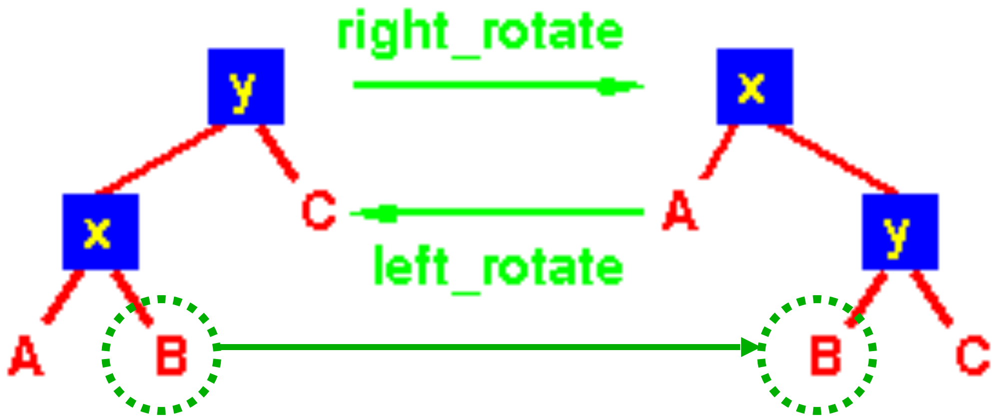


  - **For both trees: the inorder traversal is**

A x B y C

Trees - Rotations

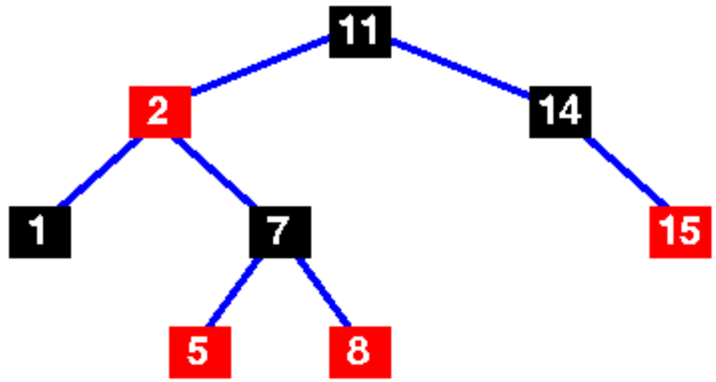
- **Binary search tree**
  - **Rotations can be either left- or right-rotations**



  - **Note that in this rotation, it was necessary to move B from the right child of x to the left child of y**

Trees - Red-Black Trees

- A Red-Black Tree
  - Binary search tree
  - Each node is “coloured” red or black

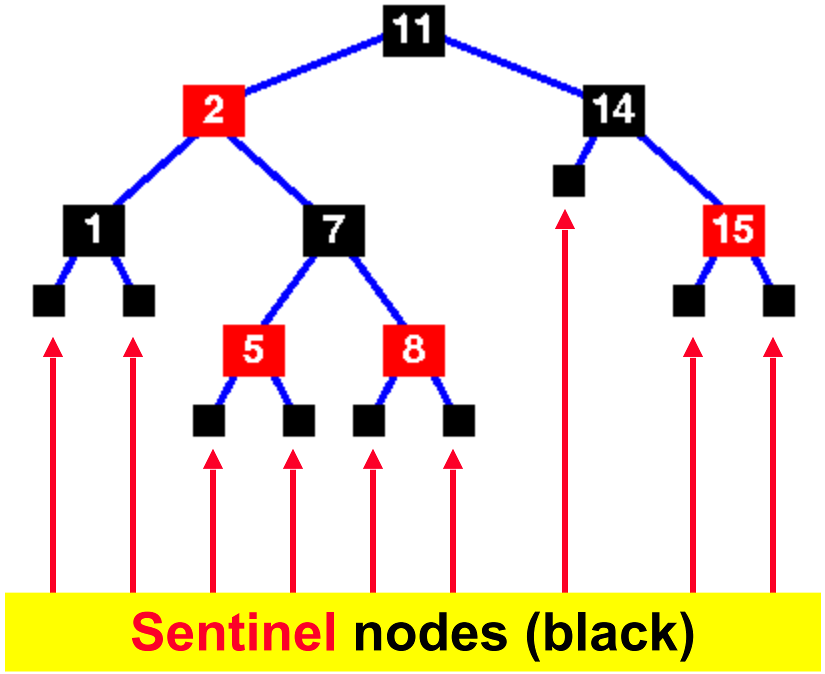


  - An ordi ode colourings to make a red-black tree

Trees - Red-Black Trees

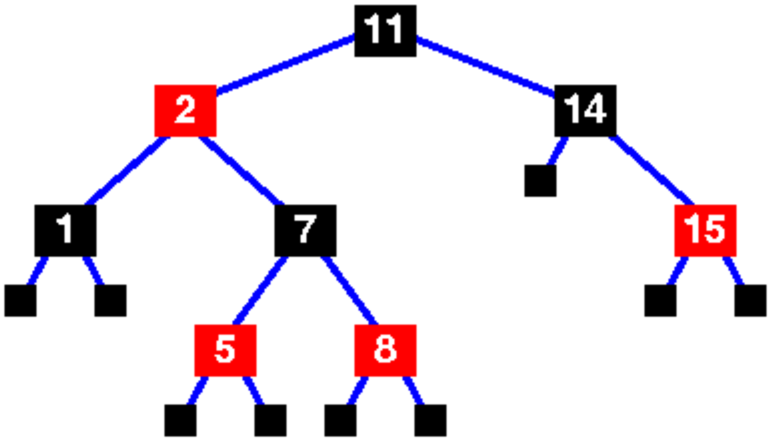
- A Red-Black Tree
  - Every node is RED or BLACK
  - Every leaf is BLACK

When you examine rb-tree code, you will see sentinel nodes (black) added as the leaves. They contain no data.



Trees - Red-Black Trees

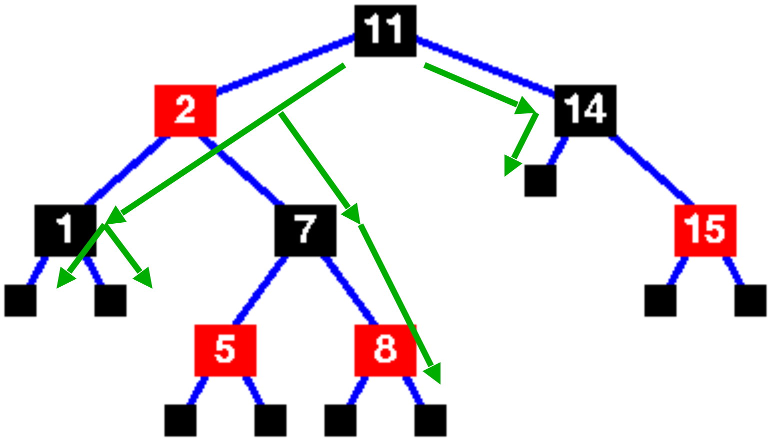
- A Red-Black Tree
  - Every node is RED or BLACK
  - Every leaf is BLACK
  - If a node is RED, then both children are BLACK



This implies that no path may have two adjacent RED nodes.
(But any number of BLACK nodes may be adjacent.)

Trees - Red-Black Trees

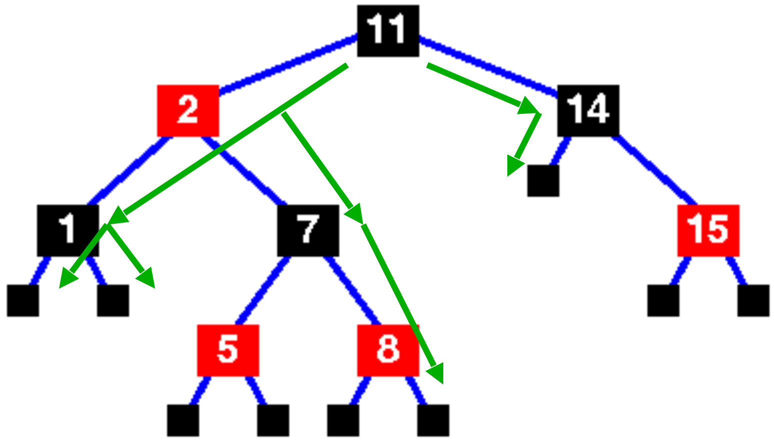
- A Red-Black Tree
  - Every node is RED or BLACK
  - Every leaf is BLACK
  - If a node is RED, then both children are BLACK
  - Every path from a node to a leaf contains the same number of BLACK nodes



From the root,
there are 3 BLACK nodes
on every path

Trees - Red-Black Trees

- A Red-Black Tree
  - Every node is RED or BLACK
  - Every leaf is BLACK
  - If a node is RED, then both children are BLACK
  - Every path from a node to a leaf contains the same number of BLACK nodes



The length of this path is the black height of the tree

Height of a Red-black Tree

Example:

- **Height of a node:**
  $h(x)$ = # of edges in a longest path to a leaf.
- **Black-height of a node**

  $bh(x)$ = # of black nodes on path from x to leaf, not counting x.

- **How are they related?**
  - $bh(x) \leq h(x) \leq 2\ bh(x)$

Bound on RB Tree Height

Lemma: The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.

Proof: By induction on height of x.

- **Base Case:** Height $h(x) = 0 \Rightarrow x$ is a leaf $\Rightarrow \text{bh}(x) = 0$. Subtree has $2^0-1 = 0$ nodes. √
- **Induction Step:** Height $h(x) = h > 0$ and $bh(x) = b$.
  - Each child of x has height h - 1 and black-height either b (child is red) or b - 1 (child is black).
  - By ind. hyp., each child has $\geq 2^{\text{bh}(x)-1}-1$ internal nodes.
  - Subtree rooted at x has $\geq 2\,(2^{\text{bh}(x)-1} - 1)+1$ $= 2^{\text{bh}(x)} - 1$ internal nodes. (The +1 is for x itself.)

Bound on RB Tree Height

Lemma: The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.

Lemma: A red-black tree with *n* internal nodes has height at most 2 lg(*n*+1).

Proof:

By the above lemma, $n \geq 2^{bh} - 1$,

and since $bh \geq h/2$, we have $n \geq 2^{h/2} - 1$.

$\Longrightarrow$ $h \leq 2 \lg(n + 1)$.

Trees - Red-Black Trees

- *Data structure*
  - As we'll see, nodes in red-black trees need to know their parents,
  - so we need this data structure

```
struct t_red_black_node {
    enum { red, black } colour;
    void *item;
    struct t_red_black_node *left,
                            *right,
                            *parent;
    }
```

Same as a binary tree with these two attributes added

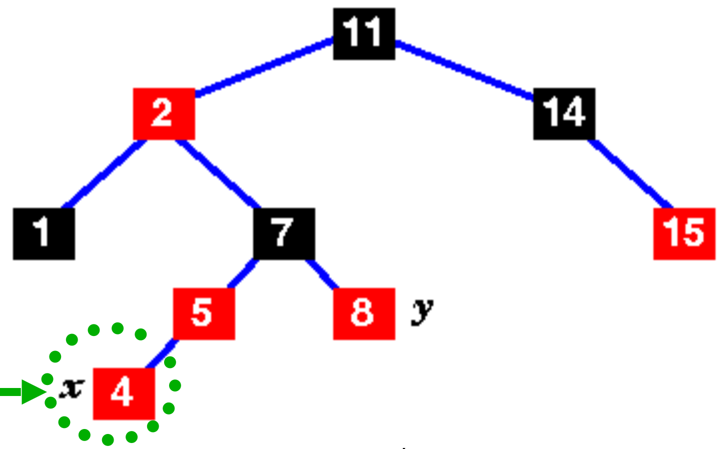
Trees - Insertion

- **Insertion of a new node**
  - **Requires a re-balance of the tree**

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
    /* Now restore the red-black property */
    x->colour = red;
              T->root) &&
              ent == x->pa
              's parent is                                         ' */
              parent->pare
              >colour == r
              case 1 - cha
              ola
              are
              the tree */
           x = x->parent->parent;
```

Insert node
4
Mark it red

Label the current node
x

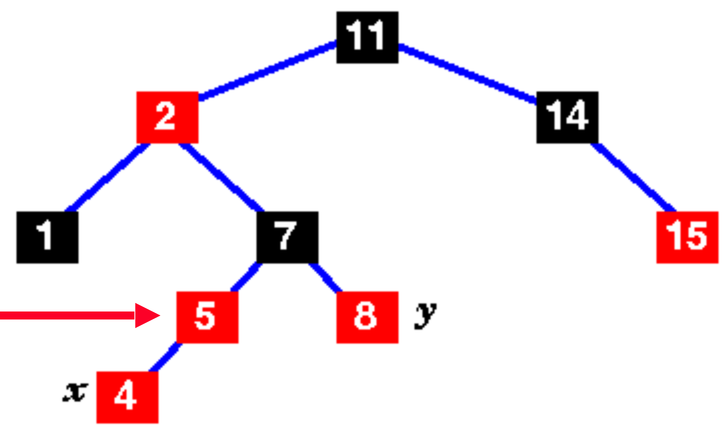


Trees - Insertion

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
     /* Now restore the red-black property */
     x->colour = red;
    while ( (x != T->root) && (x->parent->colour == red) )
{
        if ( x->parent == x->parent
                                              l
                                              >
                                              )
            /* case 1 - change
            x->parent->colour =
            y                   lack;
            x->parent->parent->
            /* Move x up the tr
            x = x->parent->parent;
```

While we haven’t reached the root and x’s parent is red

`x->parent`



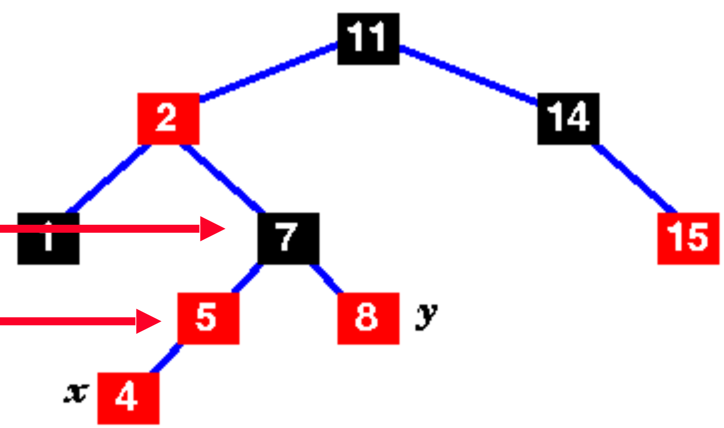
Trees - Insertion

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
    /* Now restore the red-black property */
    x->colour = red;
    while ( (x != T->root) && (x->parent->colour == red) )
{
      if ( x->parent == x->parent->parent->left ) {
                                      lef
                                ->ri
           if ( y->colour == red ) {
               /* case 1 - change th
                                  = b
               y->colour = black;
               x->            ent->co
               /* Move x up the tree
               x = x->parent->parent
```

If x is to the left of it’s granparent

x->parent->parent

x->parent



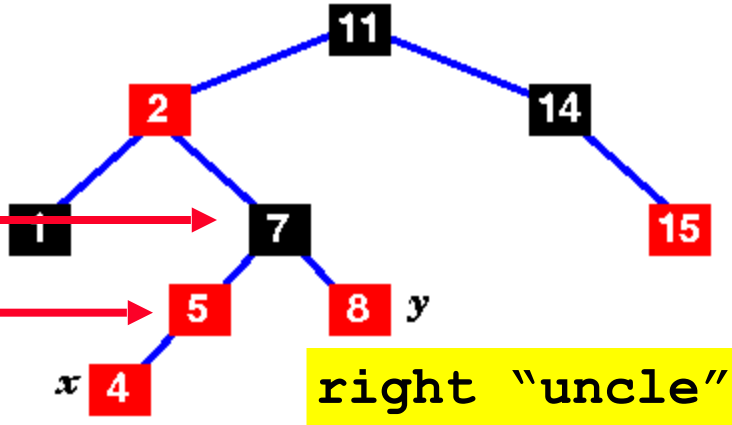
Trees - Insertion

```
/* Now restore the red-black property */
x->colour = red;
while ( (x != T->root) && (x->parent->colour == red) )
{
    if ( x->parent == x->parent->parent->left ) {
        /* If x's parent is a left, y is x's right 'uncle'
*/
                          ->parer
        if ( y->colour == red ) {
            /* case 1 - change th
                                   = k
            y->colour = black;
            x->            ent->cc
            /*              the tree
            x = x->parent->parent
```

y is x's right uncle

x->parent->parent

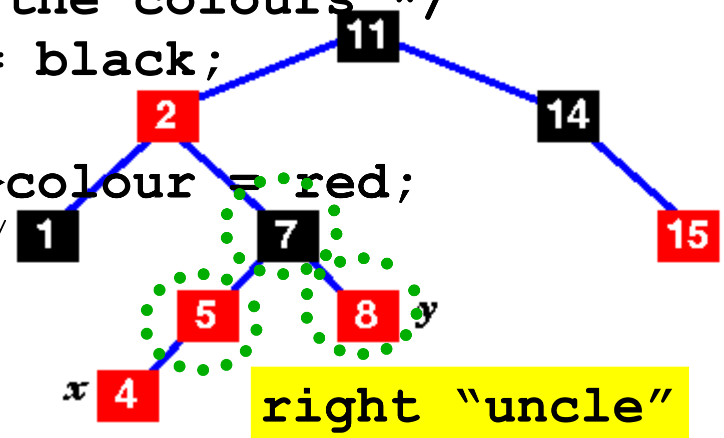
x->parent



Trees - Insertion

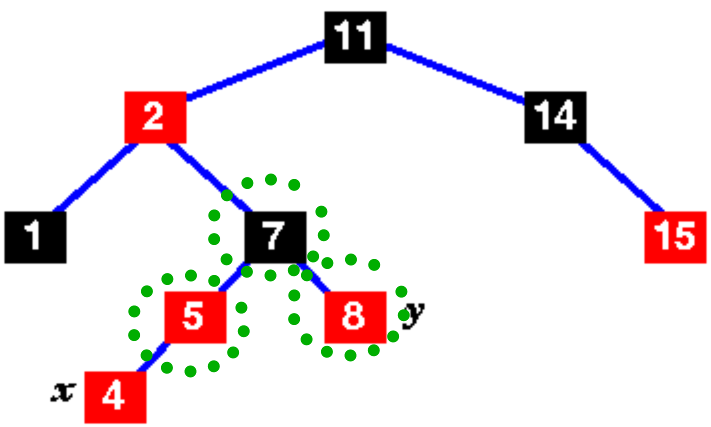
```
    while ( (x != T->root) && (x->parent->colour == red) )
{
        if ( x->parent == x->parent->parent->left ) {
            /* If x's parent is a left, y is x's right 'uncle'
*/
            y = x->parent->parent->right;
            if ( y->colour == red ) {
                /* case 1 - change the colours */
                x->parent->colour = black;
                y->colour = black;
                ent->colour = red;
                ree */
                ent;
```

If the uncle is red, change the colours of y, the grand-parent and the parent

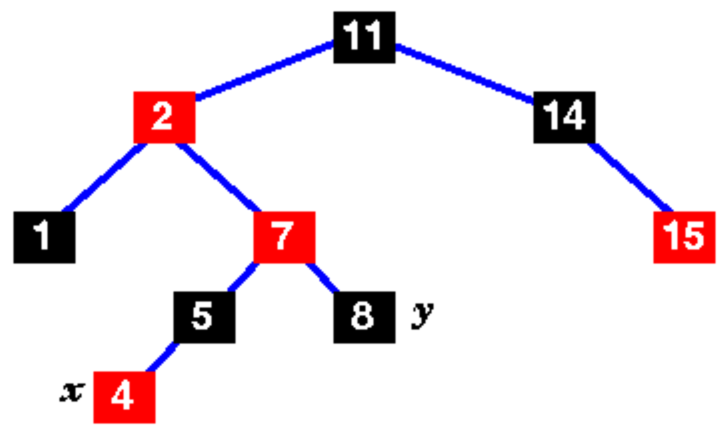


Trees - Insertion

```
    while ( (x != T->root) && (x->parent->colour == red) )
{
        if ( x->parent == x->parent->parent->left ) {
            /* If x's parent is a left, y is x's right 'uncle'
*/
            y = x->parent->parent->right;
            if ( y->colour == red ) {
```



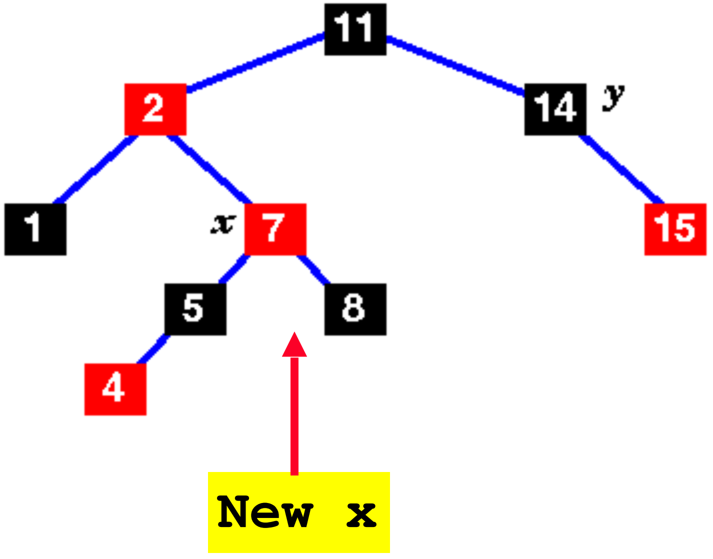

```
hange t
lour =
lack;
rent->c
 tree */
arent;
```



Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
    /* If x's parent is a left, y is x's right 'uncle' */
    y = x->parent->parent->right;
    if ( y->colour == red ) {
      /* case 1 - change the colours */
      x->parent->colour = black;
      y->colour = black;
                        r = red;
```

x’s parent is a left again,
mark x’s uncle
but the uncle is black this time

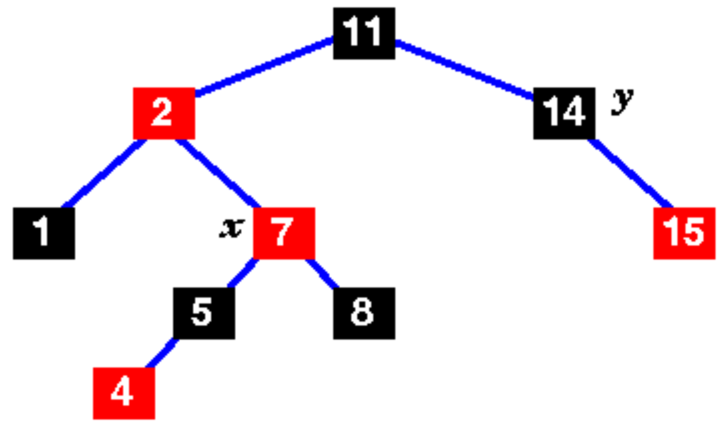


Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
    /* If x's parent is a left, y is x's right 'uncle' */
    y = x->parent->parent->right;
     if ( y->colour == red ) {
      /* case 1 - change the colours */
```

.. but the uncle is black this time and x is to the right of it's parent

```
     else {
          /* y is a black node */
          if ( x == x->parent->right ) {
               /* and x is to the right */
               /* case 2 - move x up and rotate */
               x = x->parent;
               left_rotate( T, x );
```

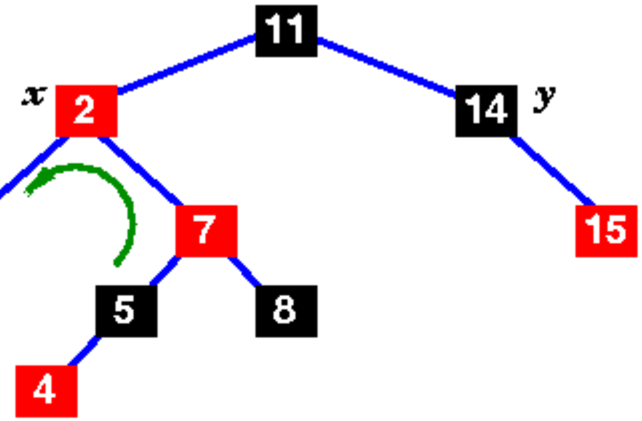


Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
    /* If x's parent is a left, y is x
    y = x->parent->parent->right;
     if ( y->colour == red ) {
      /* case 1 - change the colours */
```

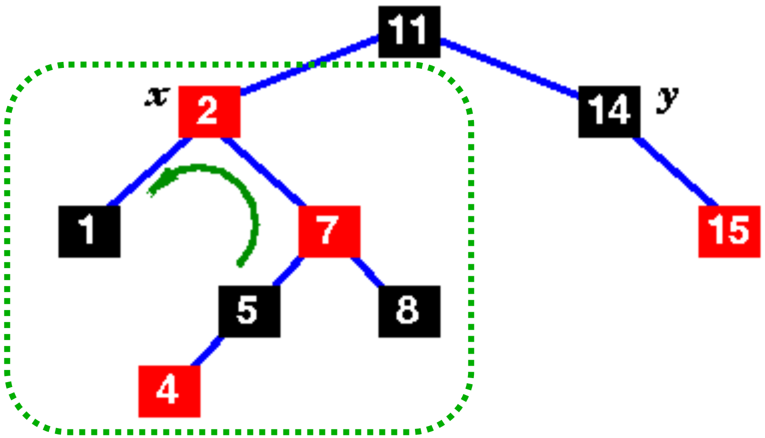
.. So move x up and rotate about x as root ...

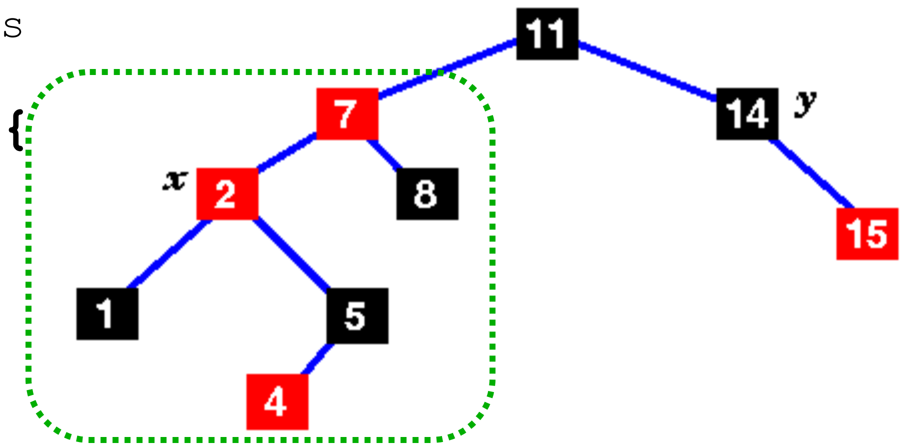
```
     else {
          /* y is a black node */
          if ( x == x->parent->right ) {
               /* and x is to the right */
               /* case 2 - move x up and rotate */
               x = x->parent;
               left_rotate( T, x );
```



Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
                                 parent->left ) {
                                  y is
                                 ;
                                 ) {
```

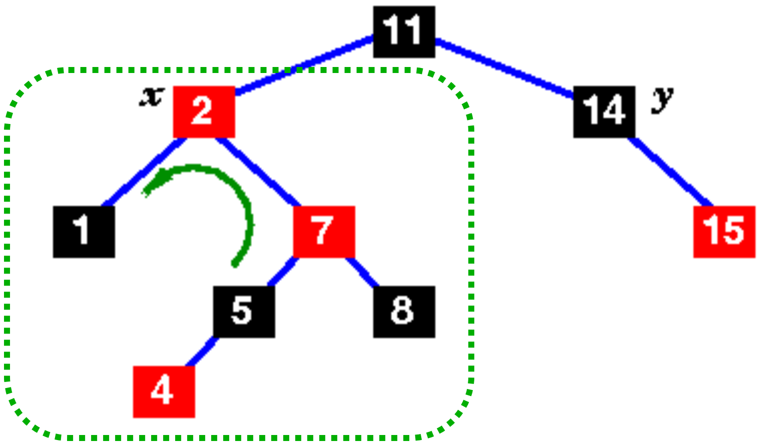


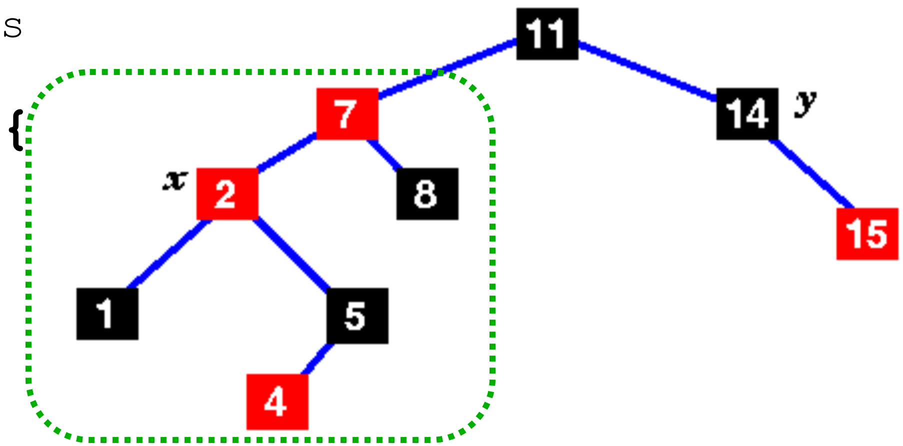



```
    else {
        /* y is a black node */
        if ( x == x->parent->right ) {
            /* and x is to the right */
            /* case 2 - move x up and rotate */
            x = x->parent;
            left_rotate( T, x );
```

Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
                              arent->left ) {
                               y is
                              ;
                              ) {
    else {
        /* y is a black node */
        if ( x == x->parent->right ) {
            /* and x is to the right */
            /* case 2 - mo
            x = x->parent;
            left_rotate( T, x );
```





.. but x's parent is still red ...

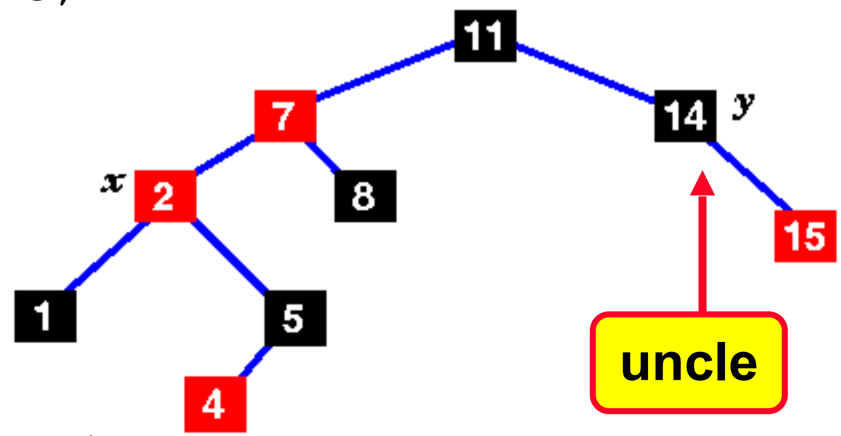
Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
   /* If x's parent is a left, y is x's right 'uncle' */
  y = x->parent->parent->right;
  if ( y->colour == red ) {
      /* case 1 - change the colours */
                        k;

                        r = red;
      /* Move x up the tree */
      x = x->parent->parent;
  else {
    /* y is a black node */
    if ( x == x->parent->right ) {
        /* and x is to the right */
```

.. The uncle is black ..

.. and x is to the *left* of its parent

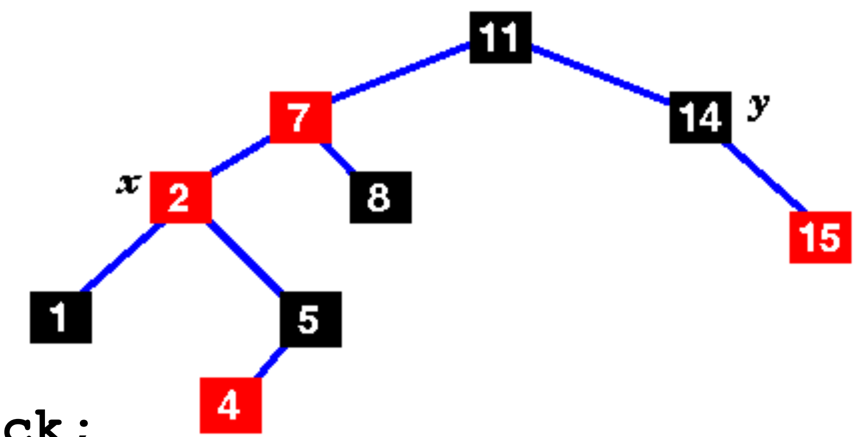


Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
  /* If x's parent is a left, y is x's right 'uncle' */
  y = x->parent->parent->right;
  if ( y->colour == red ) {
        /* case 1 - change the colours */
        x->parent->colour = black;
        y->colour = black;
        x->parent->parent->colour = red;
        /* Move x up the tree */
        x = x->parent->parent;
```

.. So we have the final case ..

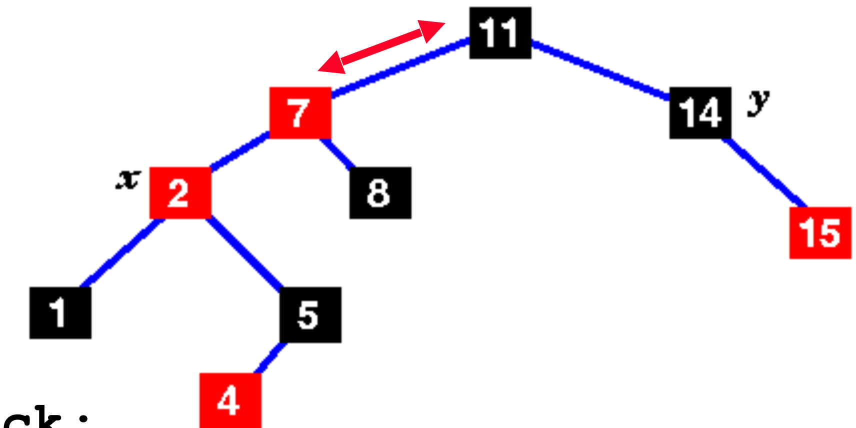
```
           /* and x is to the right */
           /* case 2 - move x up and rotate */
           x = x->parent;
           left_rotate( T, x );
   else { /* case 3 */
      x->parent->colour = black;
      x->parent->parent->colour = red;
      right_rotate( T, x->parent->parent );
      }
```



Trees - Insertion

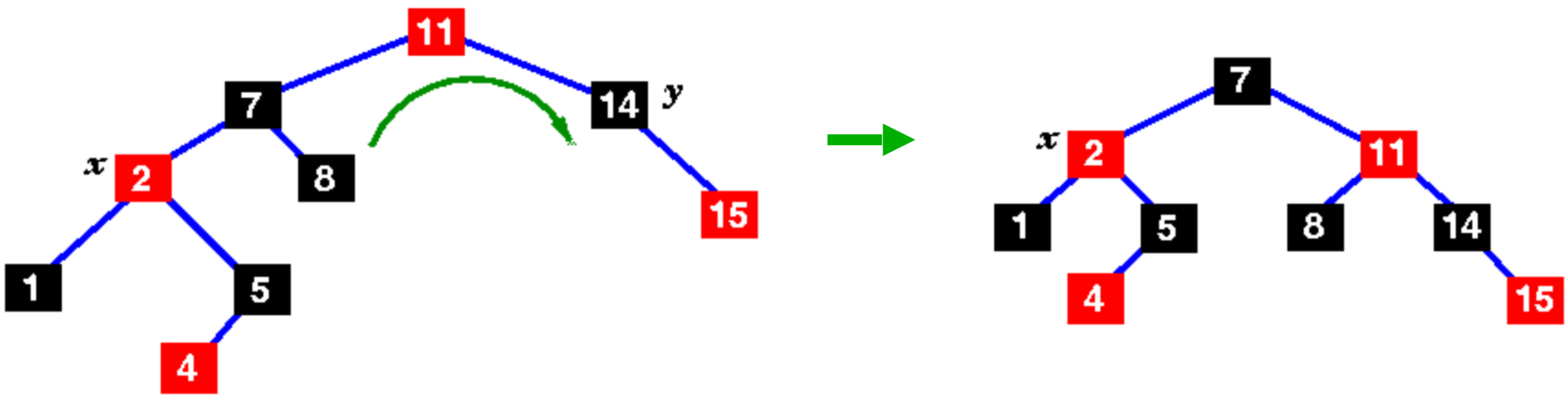
```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
  /* If x's parent is a left, y is x's right 'uncle' */
  y = x->parent->parent->right;
  if ( y->colour == red ) {
        /* case 1 - change the colours */
        x->parent->colour = black;
        y->colour = black;
        x->parent->parent->colour = red;
        /* Move x up the tree */
        x = x->parent->parent;
     else
     /* y
     if
                                    ate */
          x = x->parent;
          left_rotate( T, x );
   else { /* case 3 */
     x->parent->colour = black;
     x->parent->parent->colour = red;
     right_rotate( T, x->parent->parent );
     }
```

.. Change colours and rotate ..



Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
  /* If x's parent is a left, y is x's right 'uncle' */
```



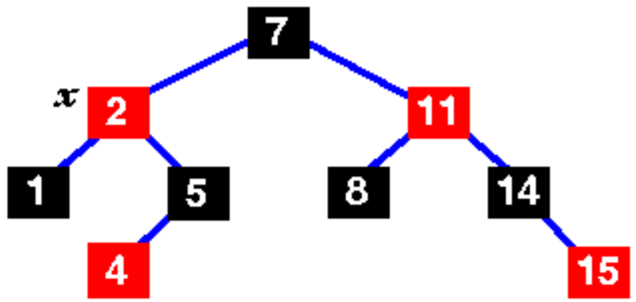

```
      x->parent->colour = black;
      x->parent->parent->colour = red;
      right_rotate( T, x->parent->parent );
      }
```

Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
  if ( x->parent == x->parent->parent->left ) {
  /* If x's parent is a left, y is x's right 'uncle' */
  y = x->parent->parent->right;
  if ( y->colour == red ) {
        /* case 1 - change the colours */
        x->parent->colour = black;
```

This is now a red-black tree ..
So we're finished!

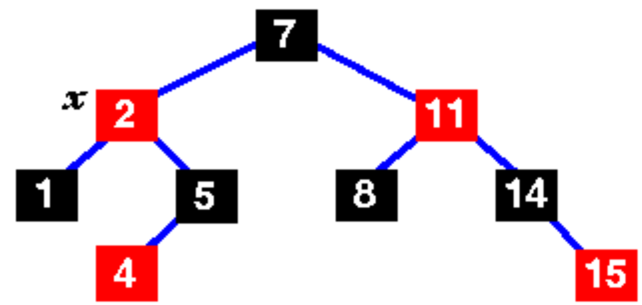
```
             /* and x is to the right */
             /* case 2 - move x up and rotate */
             x = x->parent;
             left_rotate( T, x );
   else { /* case 3 */
      x->parent->colour = black;
      x->parent->parent->colour = red;
      right_rotate( T, x->parent->parent );
      }
```



Trees - Insertion

```
while ( (x != T->root) && (x->parent->colour == red) ) {
 if ( x->parent == x->parent->parent->left ) {
  /* If x's parent is a left, y is x's right 'uncle' */
  y = x->parent->parent->right;
  if ( y->colour == red ) {
        /* case 1 - change the colours */
        x->
        y
        x
        /
        x
     else
     /* y
     if
           /* case 2 - move x up and rotate */
           x = x->parent;
           left_rotate( T, x );
    else { /* case 3 */
      x->parent->colour = black;
      x->parent->parent->colour = red;
      right_rotate( T, x->parent->parent );
      }
   }
 else ....
```

There's an equivalent set of cases when the parent is to the right of the grandparent!



Red-black trees - Analysis

- Addition
  - Insertion Comparisons $O(\log n)$
  - Fix-up
    - At every stage,
      x moves up the tree
      at least one level $O(\log n)$
  - Overall $O(\log n)$
- Deletion
  - Also $O(\log n)$
- More complex
- ... *but* gives $O(\log n)$ behaviour in dynamic cases

Red Black Trees - What you need to know?

- **You need to know**
  - **The algorithm exists**
  - **What it's called**
  - **When to use it**
    - *ie* **what problem does it solve?**
  - **Its complexity**
  - **Basically how it works**
  - ***Where to find an implementation***
    - **How to transform it to your application**

Thank You!!