Red–**black** trees

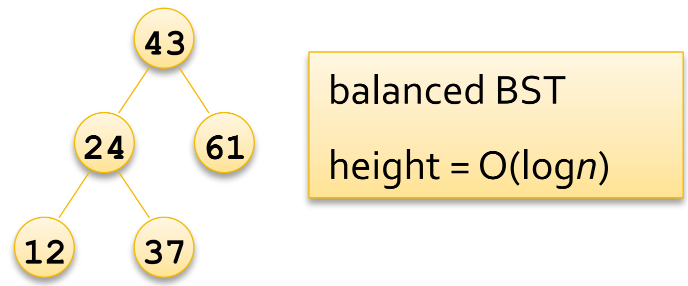
CMPT 225

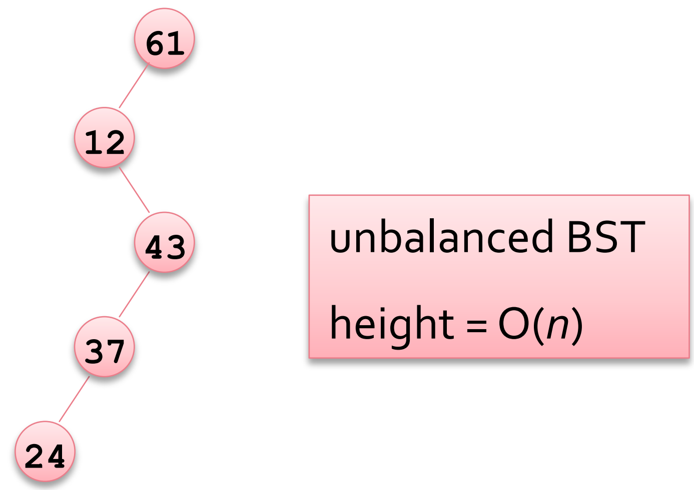
Objectives

- Define the red-black tree properties
- Describe and implement rotations
- Implement red-black tree insertion
  - We will skip red-black tree deletion

Binary Search Trees – Performance

- Items can be inserted in and removed from BSTs in *O*(*height*) time
- So what is the height of a BST?
  - If the tree is balanced: *O*(log*n*)
  - If the tree is very unbalanced: *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)

*definition of leaf on next slide

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 extra pointer to their parent
- Imagine that empty nodes are added so that every real node has two children
  - They are *imaginary* nodes so are not allocated space
  - The imaginary nodes are always coloured black

Red-black Tree Properties

1. Every node is either red or **black**
2. Every leaf is **black**
   - This refers to the *imaginary* leaves
     - 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 (mainly for convenience)

Red-black Tree Intuition

- Perfect trees are perfectly balanced
  - But they are inflexible, can only store 1, 3, 7, … nodes
- "Black" nodes form a perfect tree
- "Red" nodes allow flexibility

- Draw some pictures

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 leaf
  - Without counting v itself
  - Because of 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
  - From 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
  - A red-black tree has $\geq 2^{bh} - 1$ internal (real) nodes
    - Can be proven by induction (e.g. Algorithms, Cormen et al.)

Red-black Tree Height

- Claim: a red-black tree has height, $h \leq 2*\log(n+1)$
  - $n \geq 2^{bh} - 1$ (*see above*)
  - $bh \geq h / 2$ (*red nodes must have black children*)
  - $n \geq 2^{h/2} - 1$ (*replace bh with h*)
  - $\log(n + 1) \geq h / 2$ (*add 1, $\log_2$ of both sides*)
  - $h \leq 2*\log(n + 1)$ (*multiply both sides by 2*)

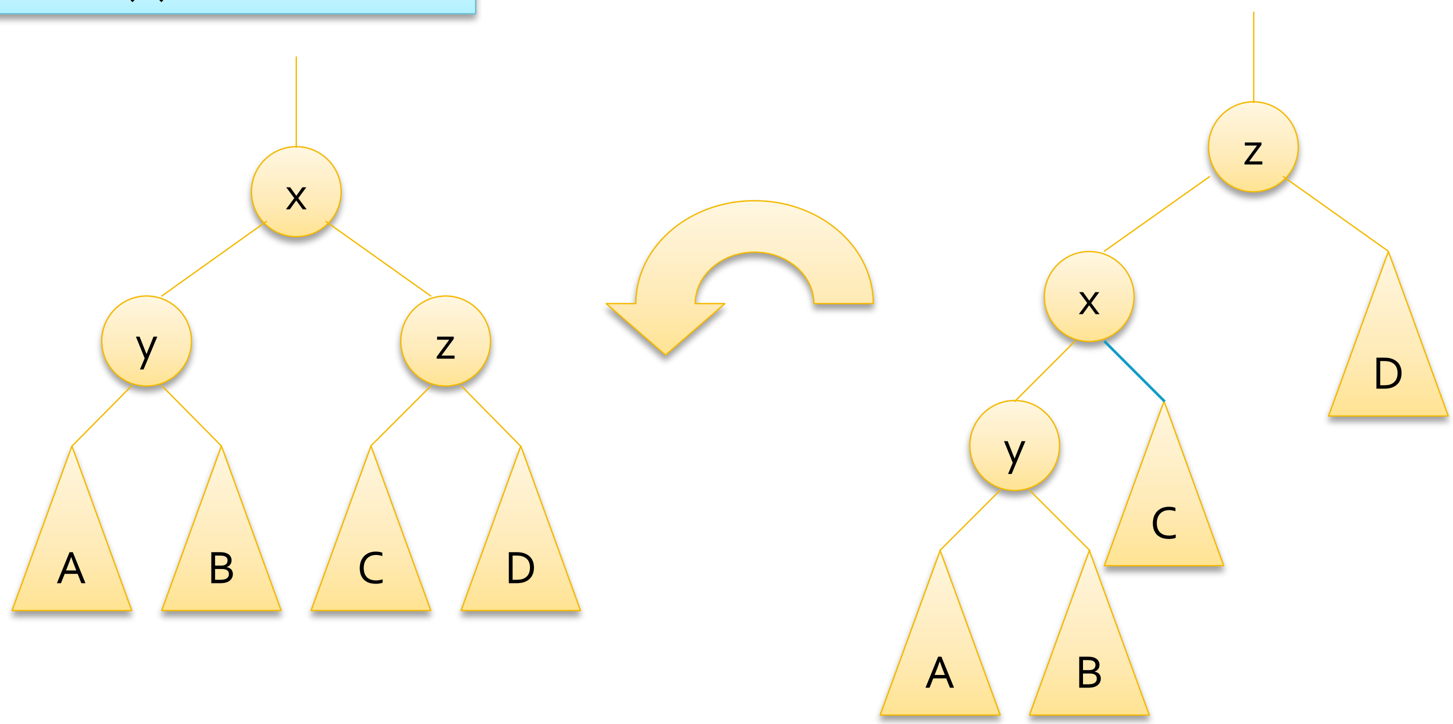
Tree Rotations

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
  - Note: only applies to *bst* with no duplicate keys!

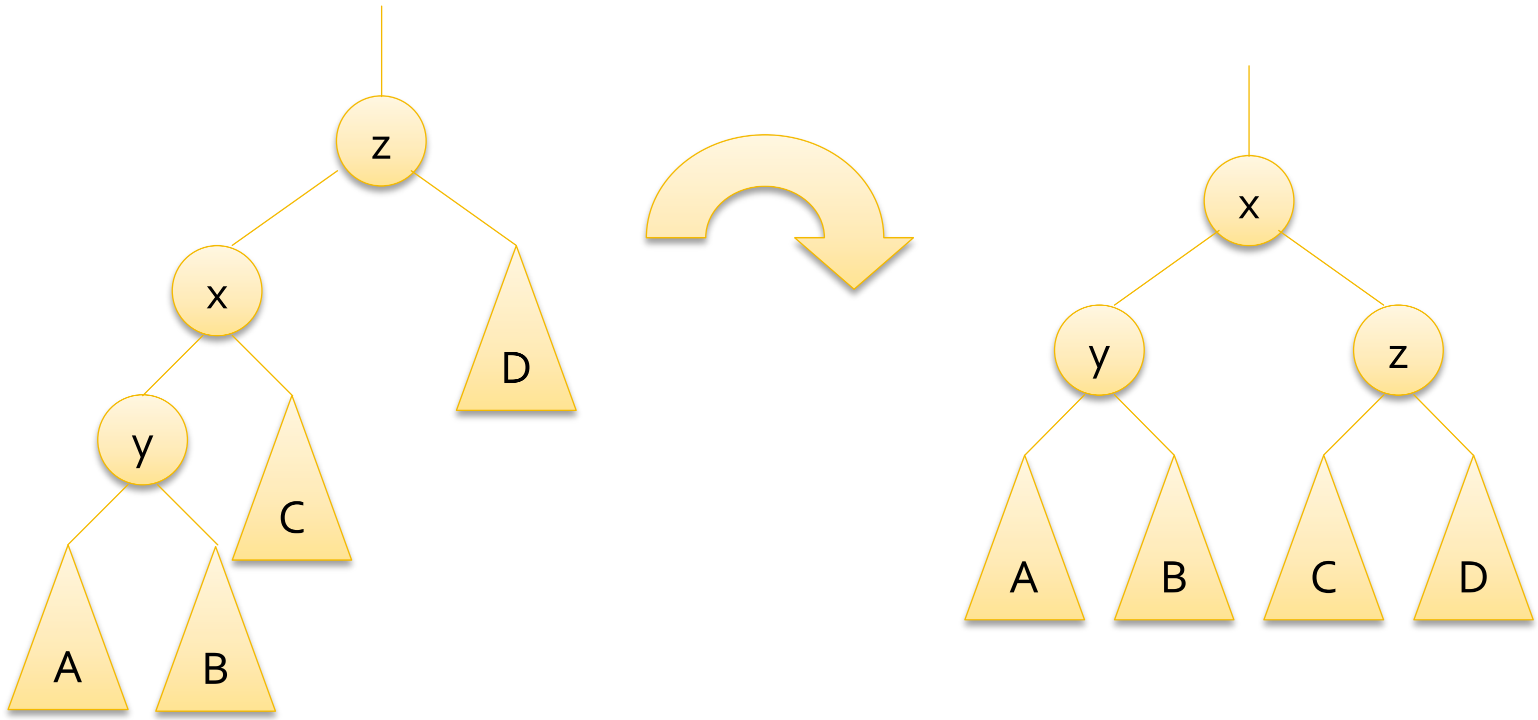
Left Rotation

Left rotate(x)



Right Rotation

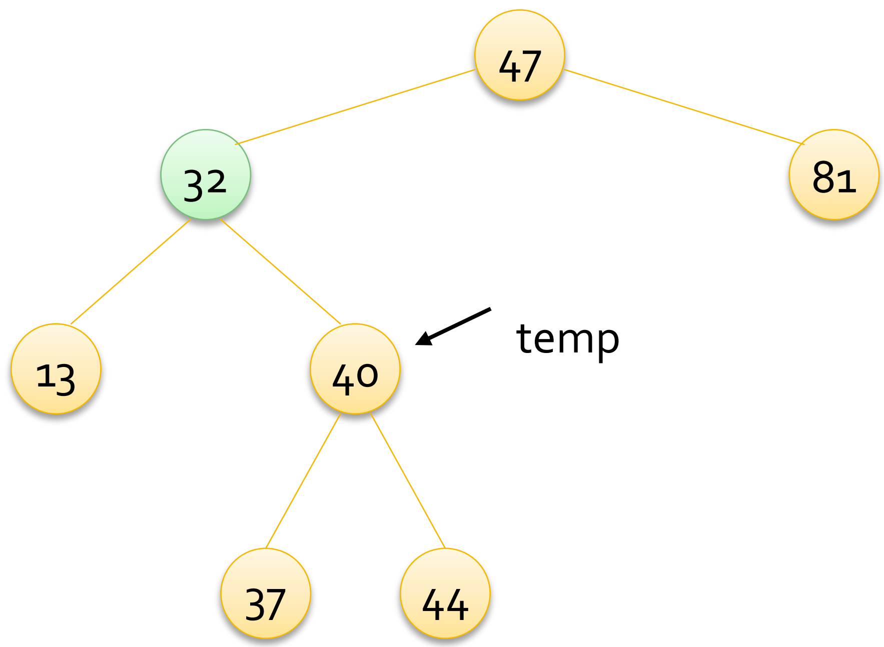
Right rotate(z)



Left Rotation Example

Left rotation of 32, call the node x

Assign a pointer to x's R child



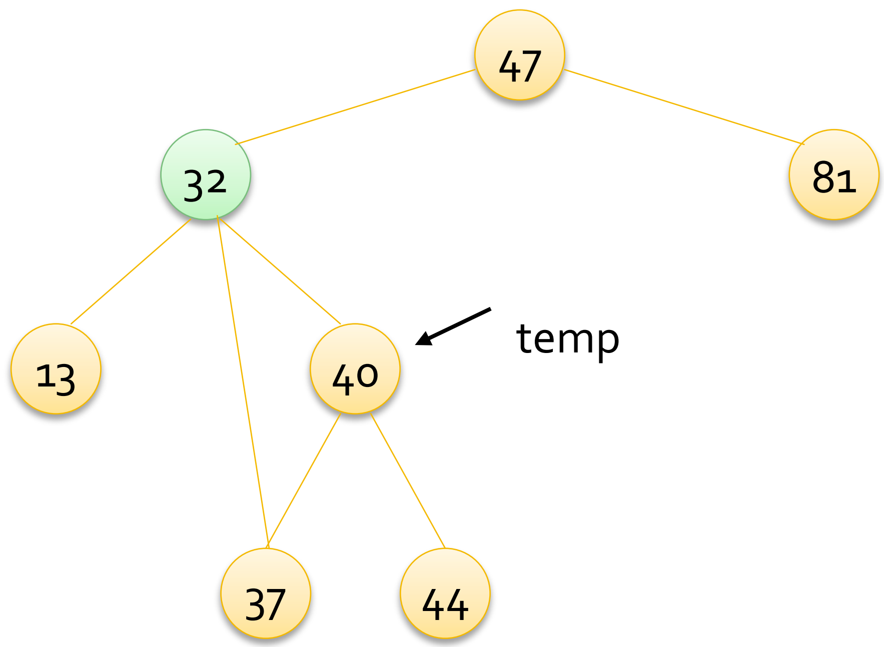
Left Rotation Example

Left rotation of 32, call the node x

Assign a pointer to x's R child

Make temp's L child x's R child

Detach temp's L child



Left Rotation Example

Left rotation of 32, call the node x

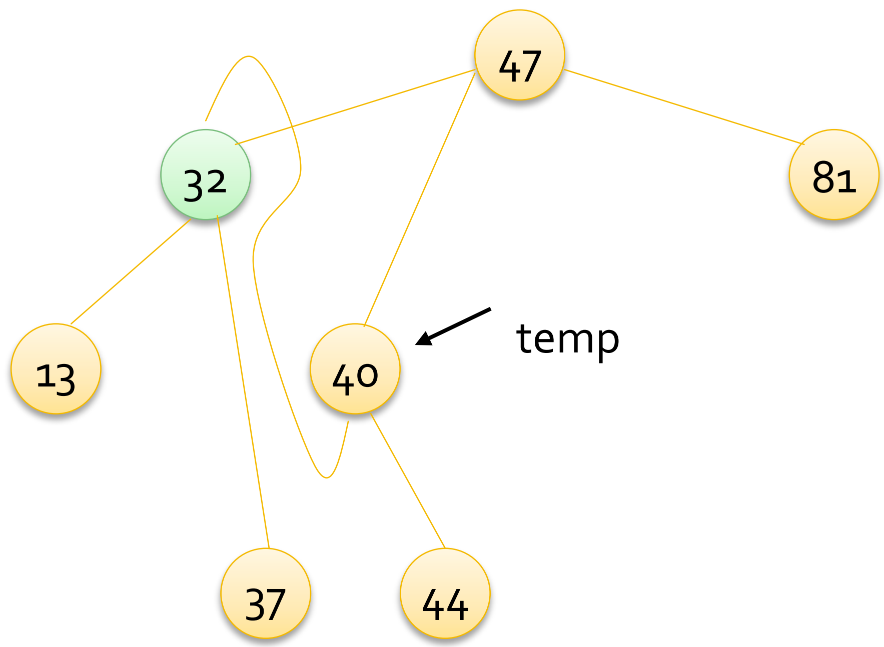
Assign a pointer to x's R child

Make temp's L child x's R child

Detach temp's L child

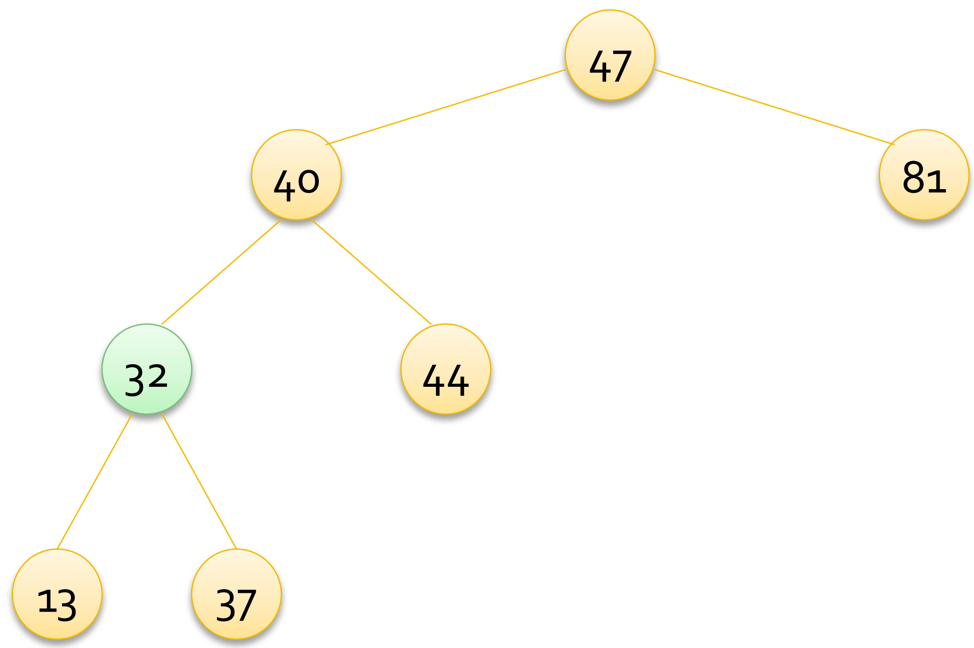
Make x temp's L child

Make temp x's parent's child



Left Rotation Example

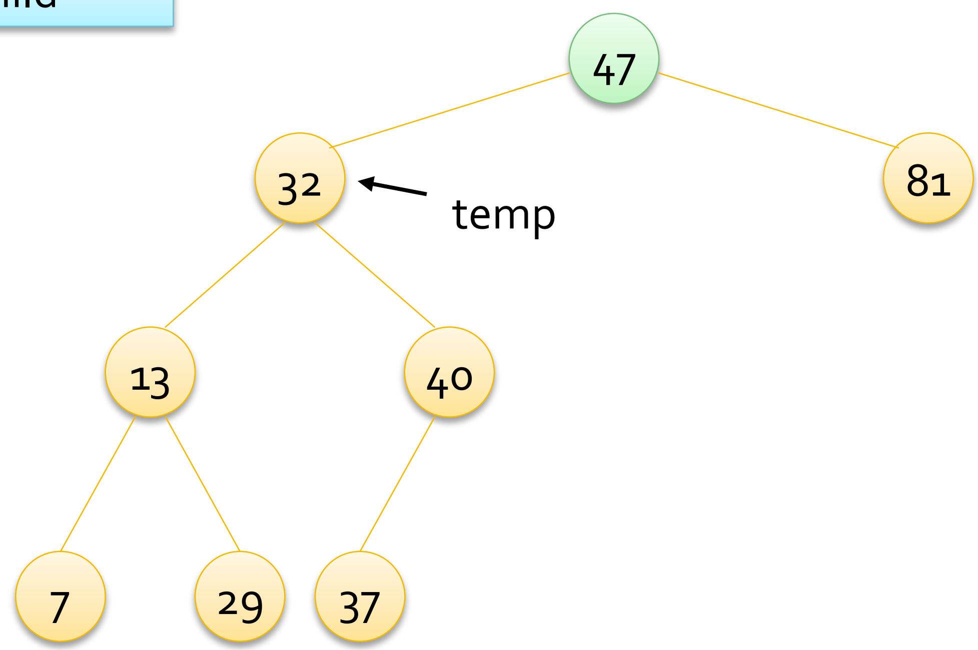
Left rotation of 32, call the node x



Right Rotation Example

Right rotation of 47, call the node x

Assign a pointer to x's L child



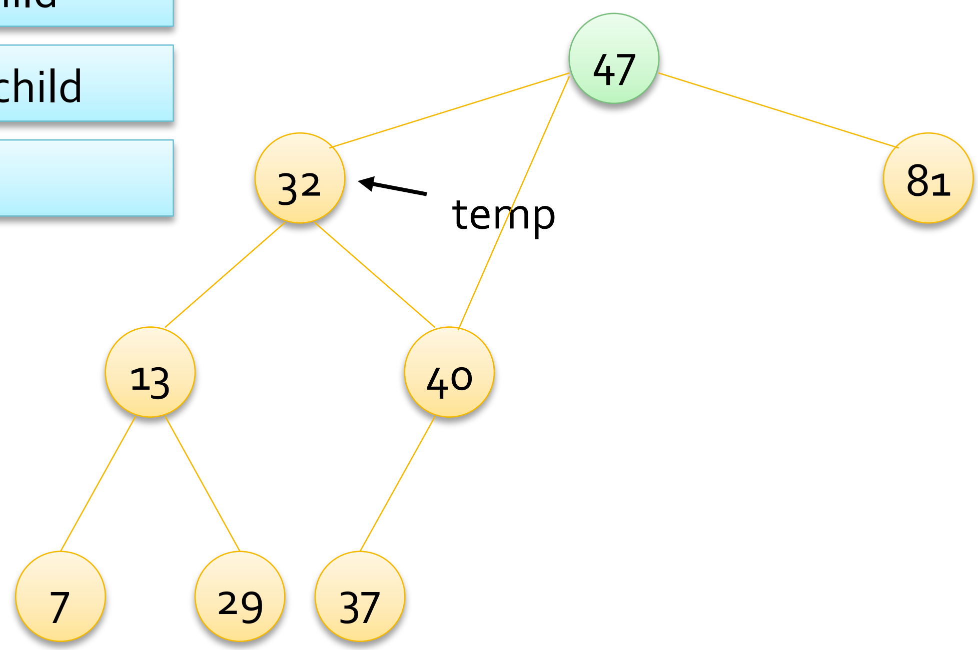
Right Rotation Example

Right rotation of 47, call the node x

Assign a pointer to x's L child

Make temp's R child x's L child

Detach temp's R child



Right Rotation Example

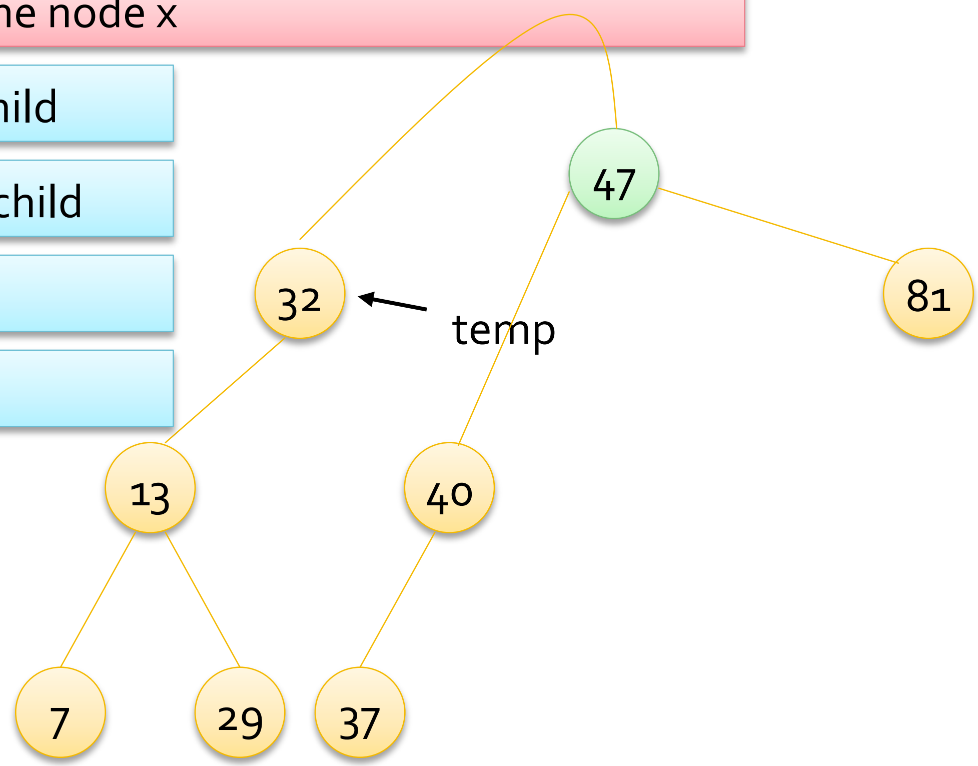
Right rotation of 47, call the node x

Assign a pointer to x's L child

Make temp's R child x's L child

Detach temp's R child

Make x temp's L child



Right Rotation Example

Right rotation of 47, call the node x

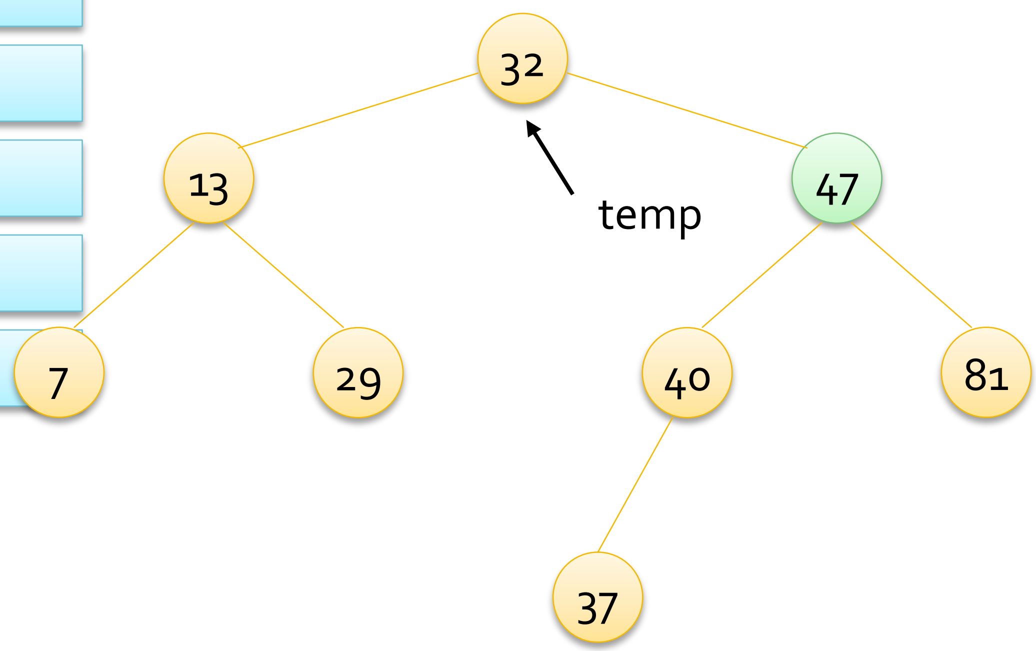
Assign a pointer to x's L child

Make temp's R child x's L child

Detach temp's R child

Make x temp's L child

Make temp the new root



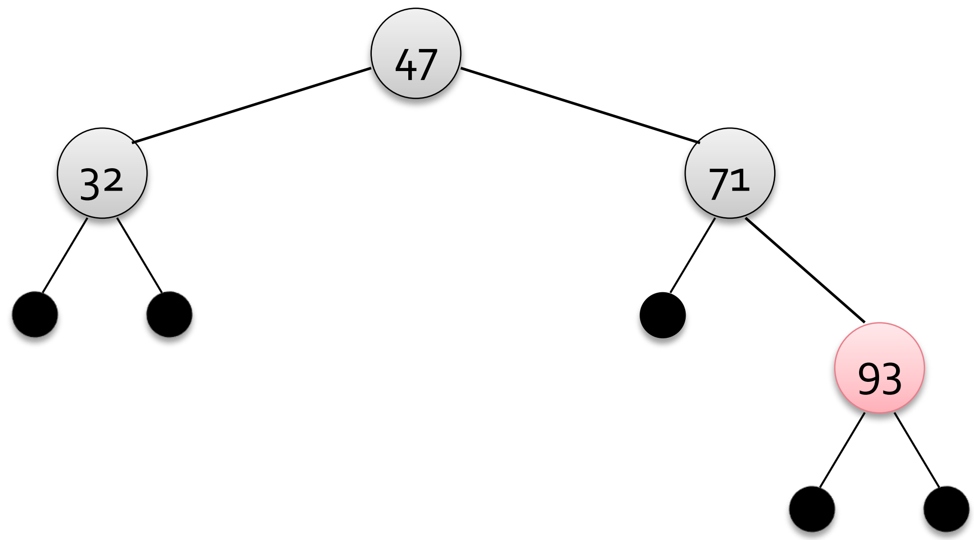
Red-black Tree Insertion

Red-black Tree Insertion

- Insert as for a binary search tree
  - Make the new node red

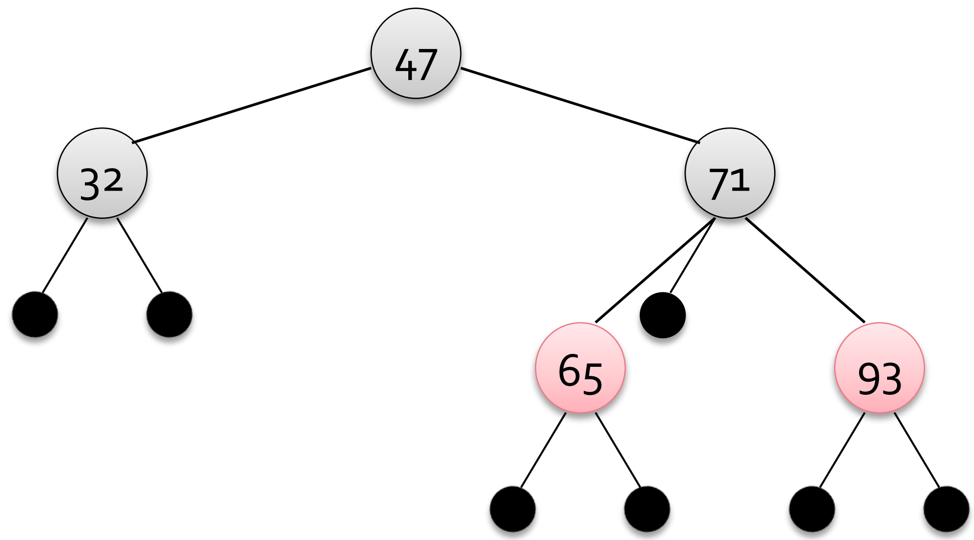
Insertion Example

Insert 65



Insertion Example

Insert 65



Red-black Tree Insertion

- Insert as for a binary search tree
  - Make the new node red
- What can go wrong? (see slide 6)
  - The only property that can be violated is that both a red node's children are black (its parent may be red)
- So, after inserting, fix the tree by re-colouring nodes and performing rotations

Fixing the Red-black Tree

- The fixing of the tree remedies the problem of two consecutive red nodes
  - There are a number of cases (that's what is next)
- It is iterative (or recursive) and pushes this problem one step up the tree at each step
  - I.e. if the consecutive red nodes are at level d, at the next step they are at d-1
  - This is why it turns out to be O(log n)
    - We won't go into the analysis

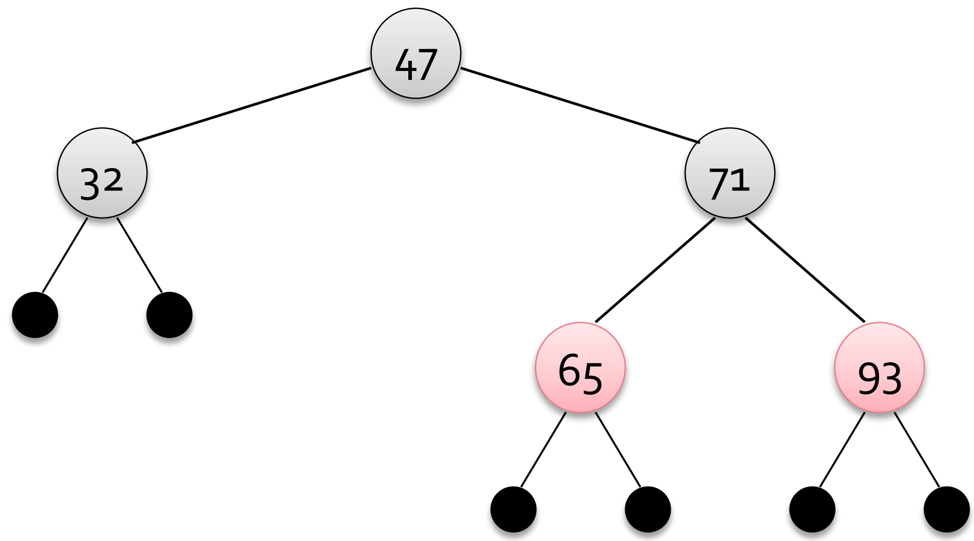
Red-black Tree Insertion I

- Need to fix tree if new node's parent is red
- Case I for fixing:
- If parent and uncle are both red
  - Then colour them black
  - And colour the grandparent red
    - It must have been black beforehand, why?

Insertion Example

Insert 65

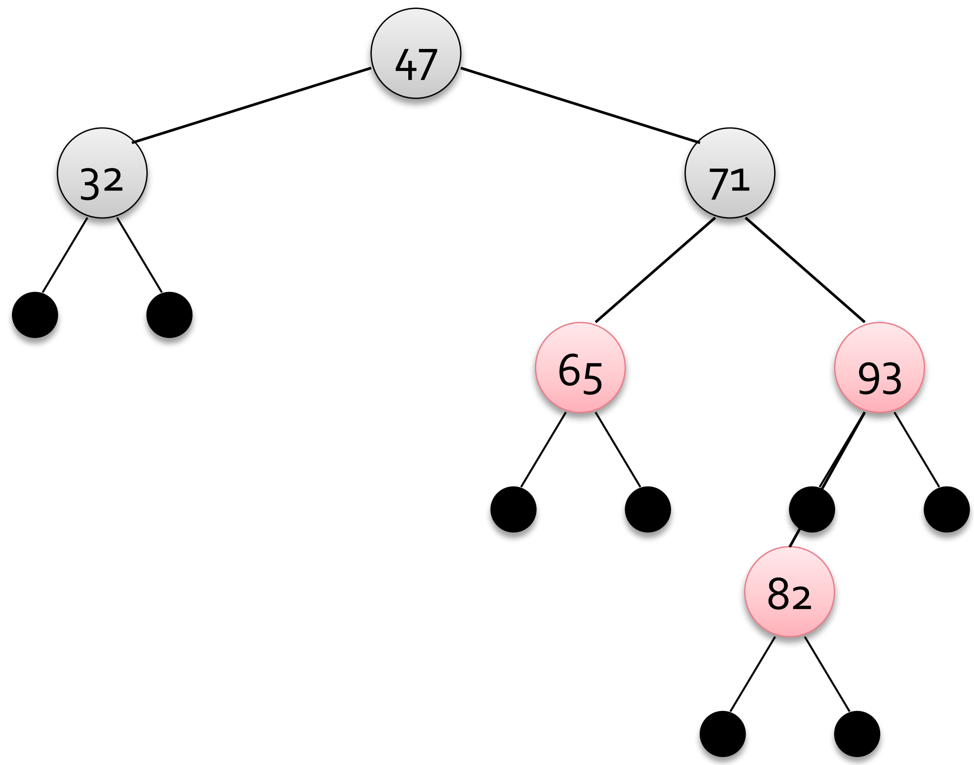
Insert 82



Insertion Example

Insert 65

Insert 82

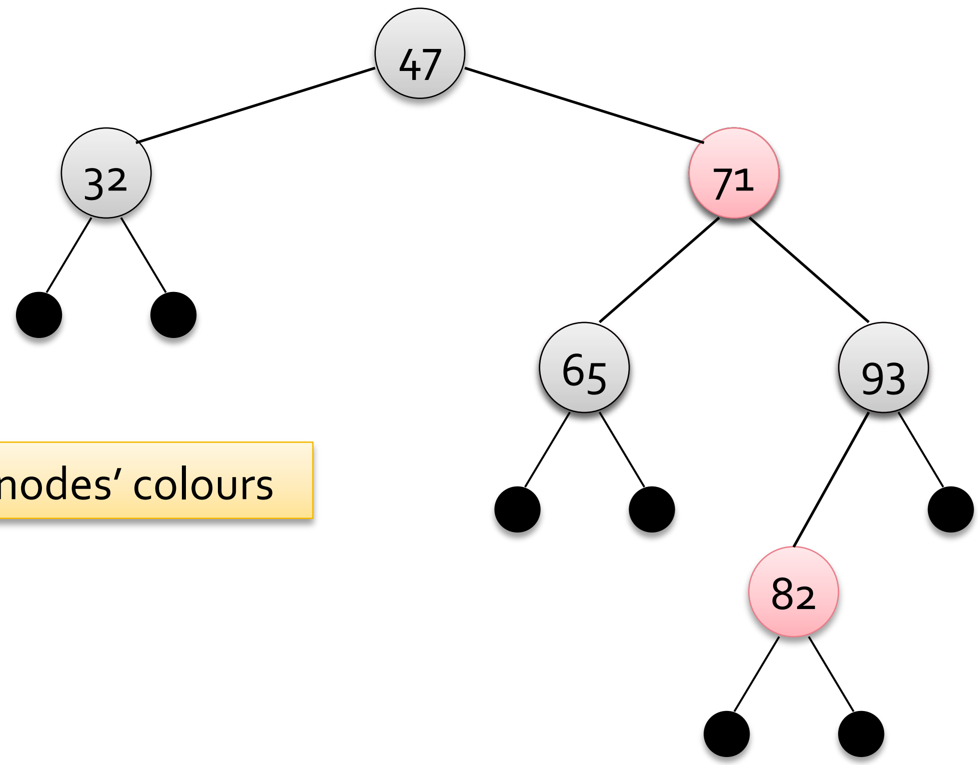


Insertion Example

Insert 65

Insert 82

change nodes' colours



Red-black Tree Insertion II

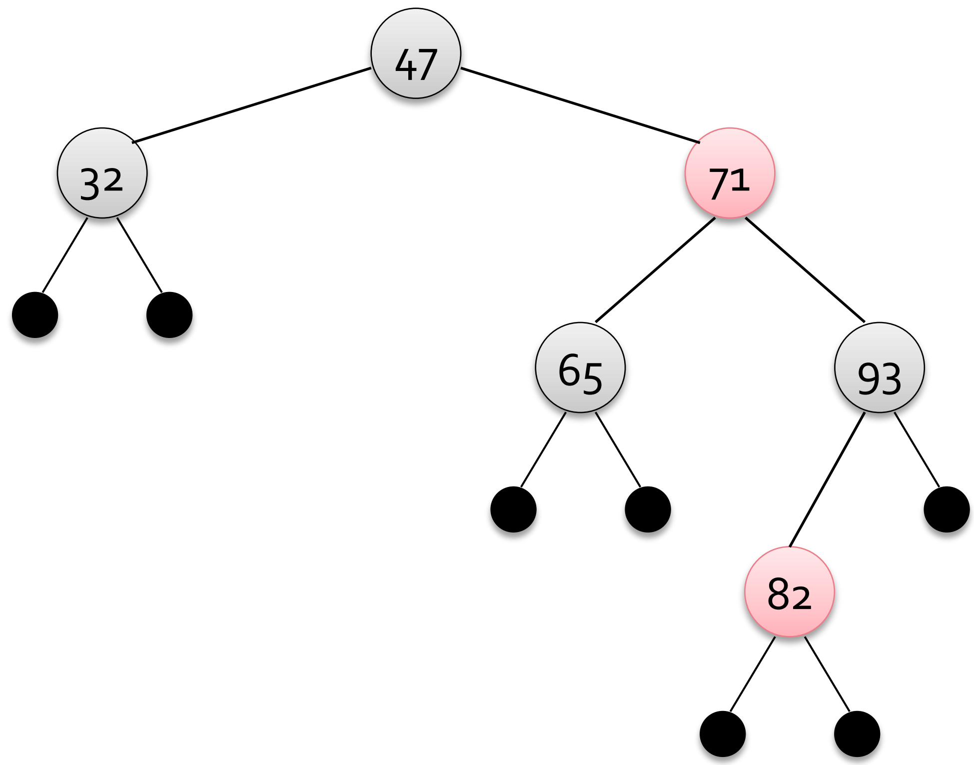
- Need to fix tree if new node's parent is red
- Case II for fixing:
- If parent is red but uncle is black
  - Need to do some tree rotations to fix it

Insertion Example

Insert 65

Insert 82

Insert 87

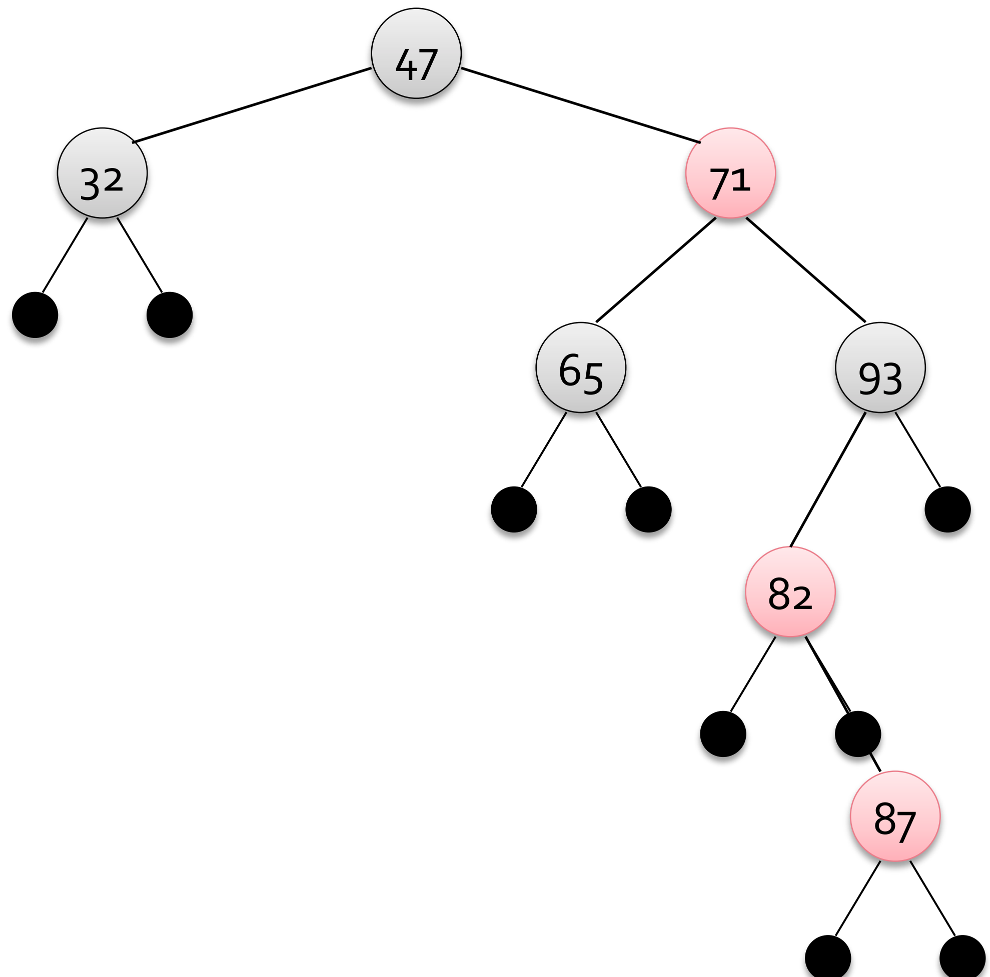


Insertion Example

Insert 65

Insert 82

Insert 87

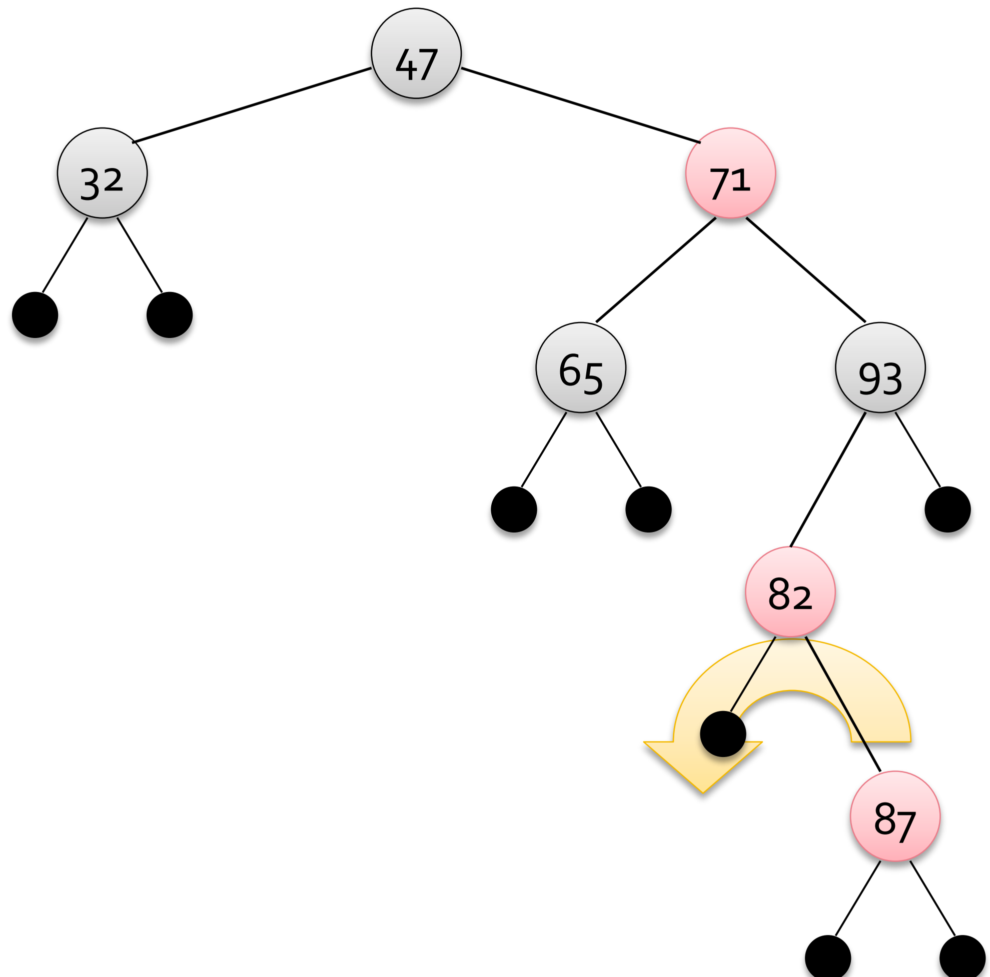


Insertion Example

Insert 65

Insert 82

Insert 87

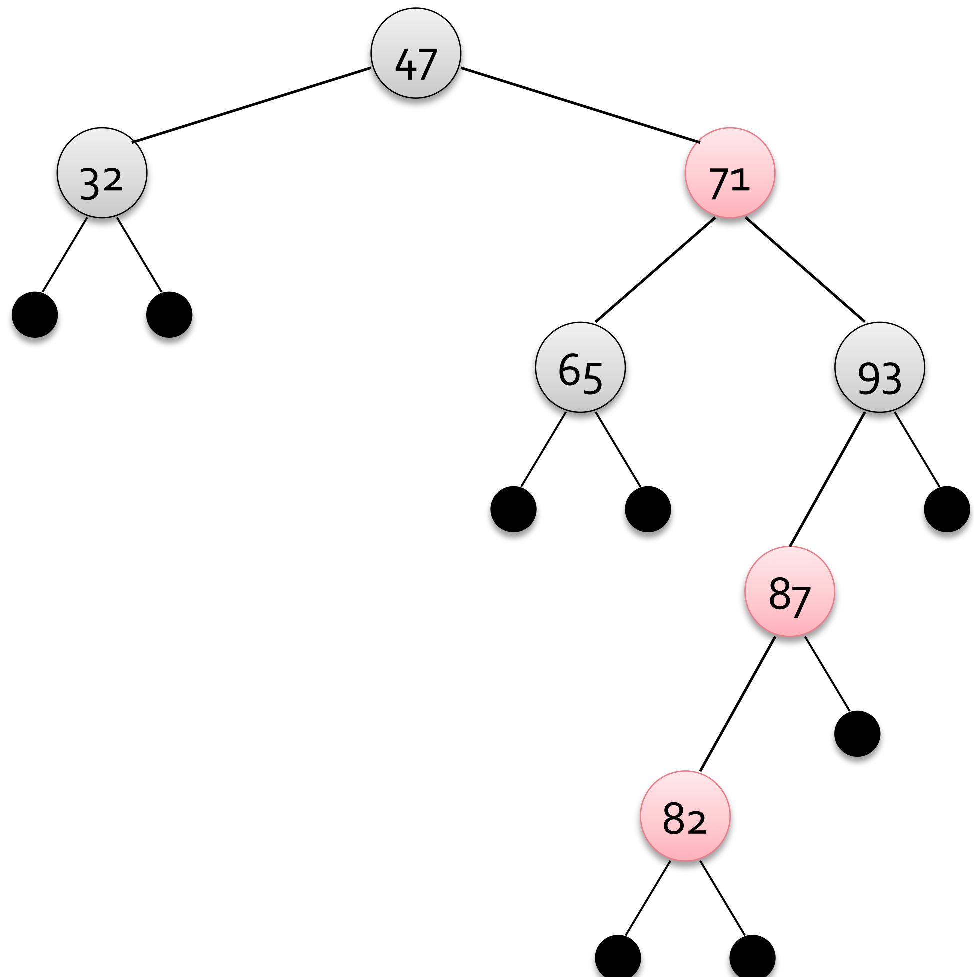


Insertion Example

Insert 65

Insert 82

Insert 87



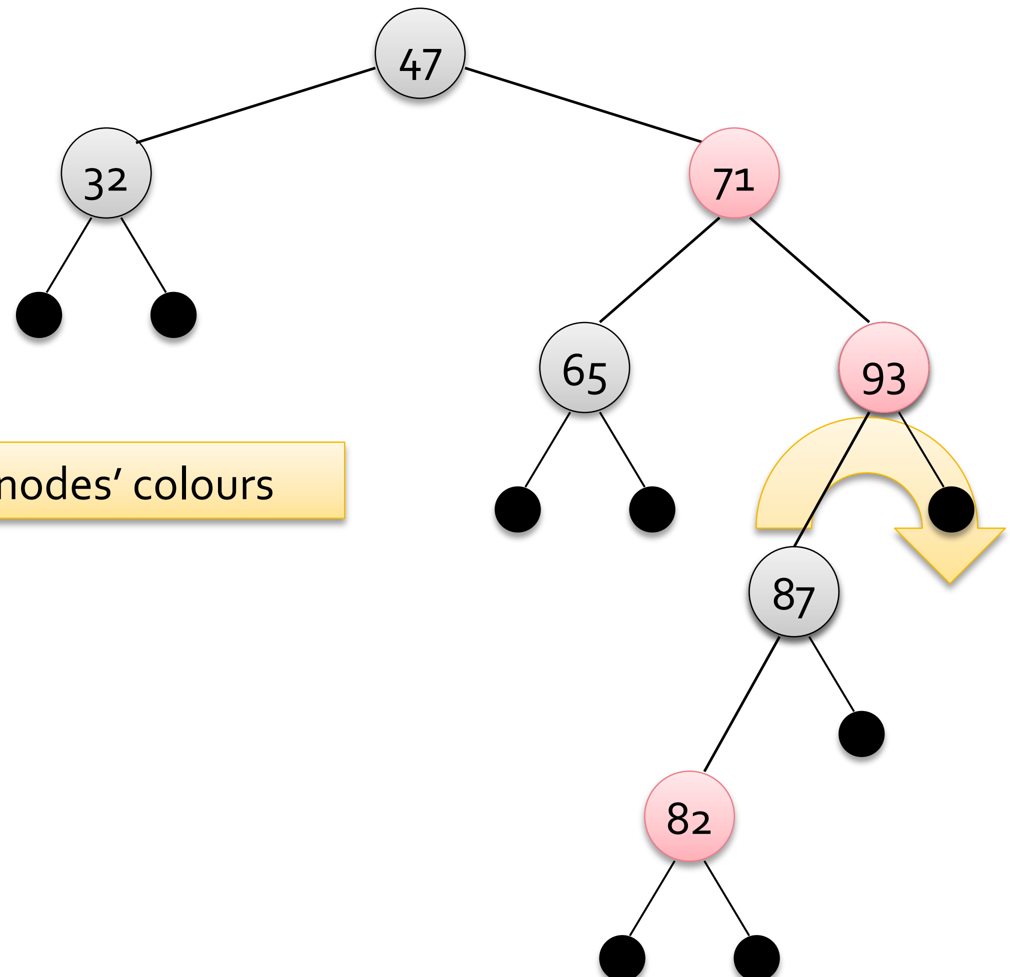
Insertion Example

Insert 65

Insert 82

Insert 87

change nodes' colours

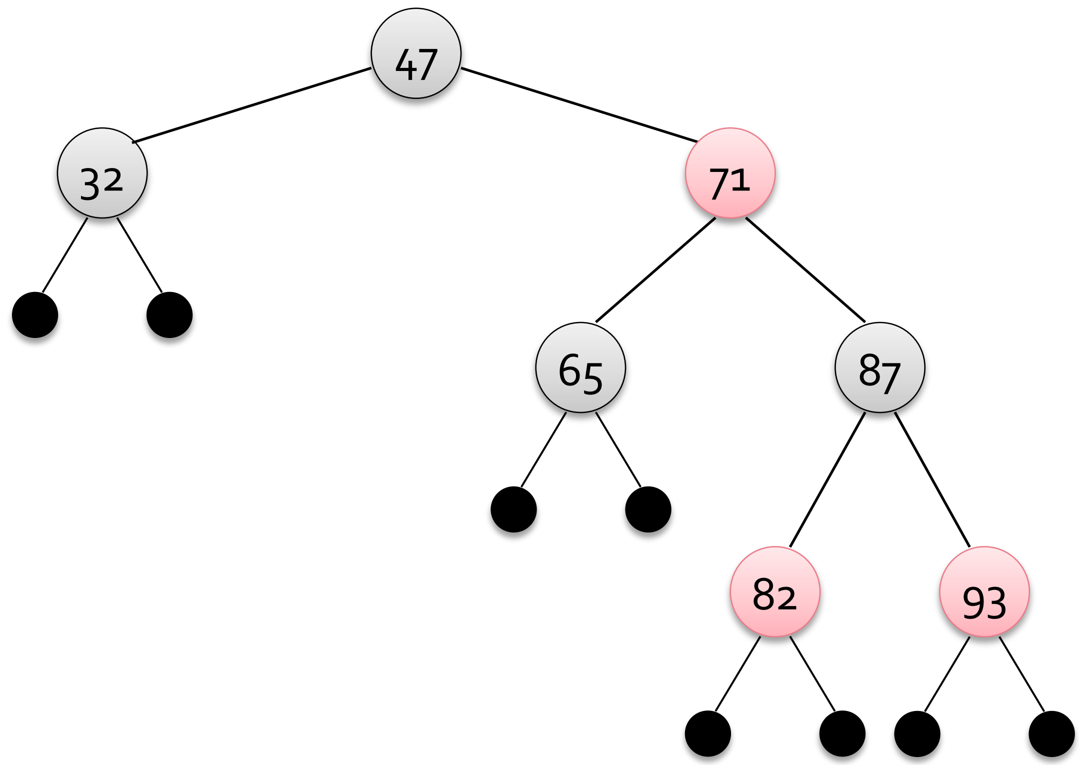


Insertion Example

Insert 65

Insert 82

Insert 87



Insertion Rotations

- Why were these rotations performed?
- First rotation made the two red nodes left children of their parents
  - This rotation isn't performed if this is already the case
  - Note that grandparent must be a black node
- Second rotation and subsequent recolouring fixes the tree

Insertion Summary

- Full details require a few cases
  - See link to example code snippets at end
  - Understand the application of tree rotations

Summary

Summary

- Red-black trees are *balanced* binary search trees
- Augment each node with a *colour*
  - Maintaining relationships between node colours maintains balance of tree
- Important operation to understand: *rotation*
  - Modify tree but keep binary search tree property (ordering of nodes)

Readings

- For implementation details, please see: http://en.wikipedia.org/wiki/Red-black_tree

(see "Operations")