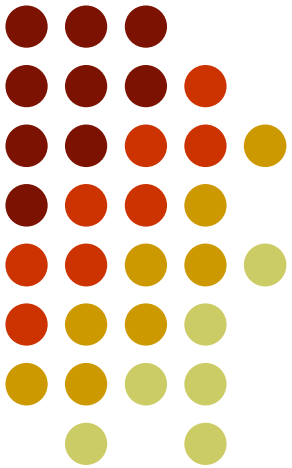
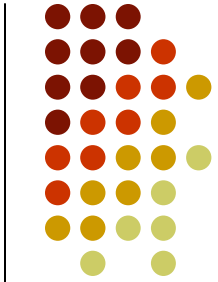
CMPT 225

Red–black trees

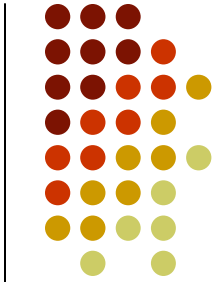


Red-black Tree Structure



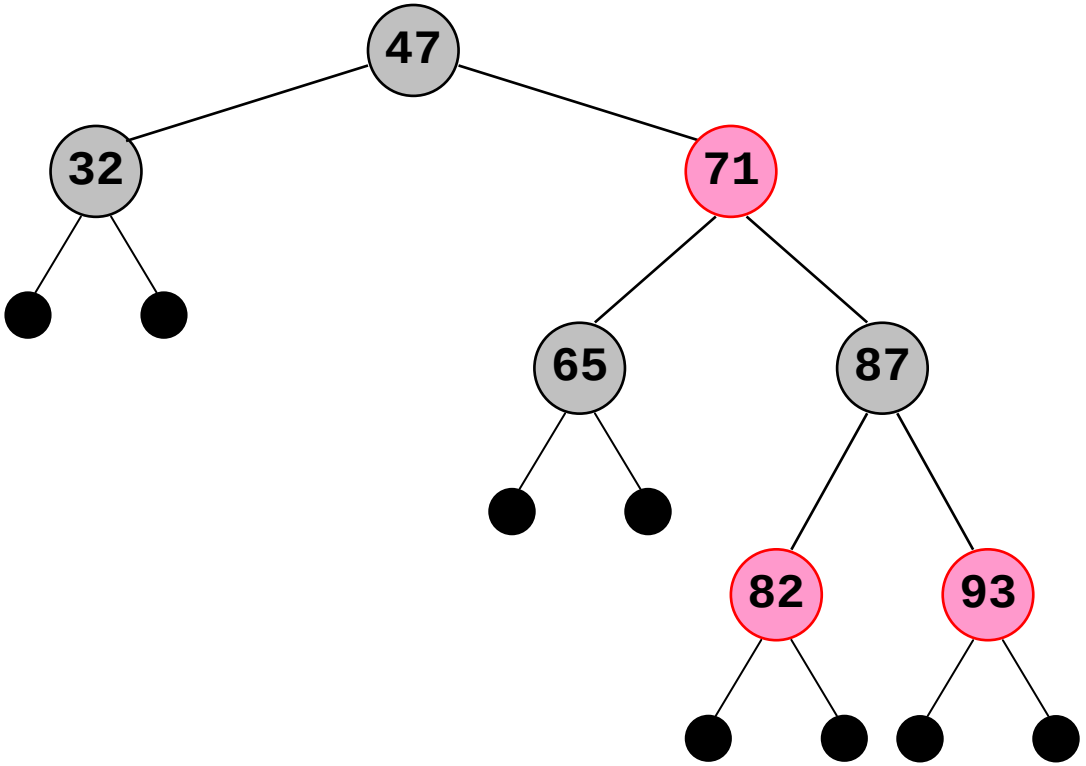
- A red-black tree is a **BST**!
- Each node in a red-black tree has an extra color field which is
  - **red** or
  - **black**
- In addition **false** nodes are added so that every (real) node has two children
  - These are **pretend** nodes, they don't have to have space allocated to them
  - These nodes are colored black
  - We do not count them when measuring a height of nodes
- Nodes have an extra reference to their parent

Red-black Tree Properties

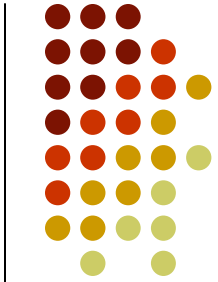


- 1 – Every node is either red or black
- 2 – Every leaf is black
  - This refers to the **pretend** leaves
  - In implementation terms, every null child of a node considered to be a black leaf
- 3 – If a node is **red** both its children must be black
- 4 – Every path from a node to its descendent leaves contains the same number of black nodes
- 5 – The root is black (mainly for convenience)

Example of RB tree

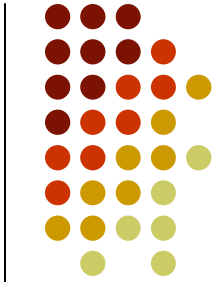


RB Trees: Data structure



```
public class TreeNode<T> {
  public enum Color {red, black}

  private T item;
  private TreeNode<T> leftChild;
  private TreeNode<T> rightChild;
  private TreeNode<T> parent;
  private Color color;

  // constructors, accessors, mutators...
}
```

Red-black Tree Height



- The black height of a node, *bh*(*v*), is the number of black nodes on any path from *v* to a leaf (not counting the pretend node).
  - Remember that 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 (not counting the pretend node).
- Let *bh* be the black height of the tree (root) and *h* the height of the tree (root)

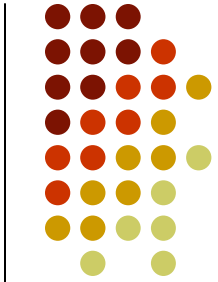
Analysis of Red-black Tree Height

Assume that a tree has *n* internal nodes

- An internal node is a non-leaf node (a node containing a value), remember that the leaf nodes are just **pretend** nodes

- **Claim:** The tree will have $h \leq 2*\log(n+1)$
  - A tree has at least $2^{bh} - 1$ internal (real) nodes, i.e., $n \geq 2^{bh} - 1$
    - Proof by MI
  - $bh \geq h / 2$ (from property 3 of a RB-tree)
  - Hence, $n \geq 2^{h/2} - 1$
  - $\log(n + 1) \geq h / 2$ (+1 and take $\log_2$ of both sides)
  - $h \leq 2*\log(n + 1)$ (multiply by 2)

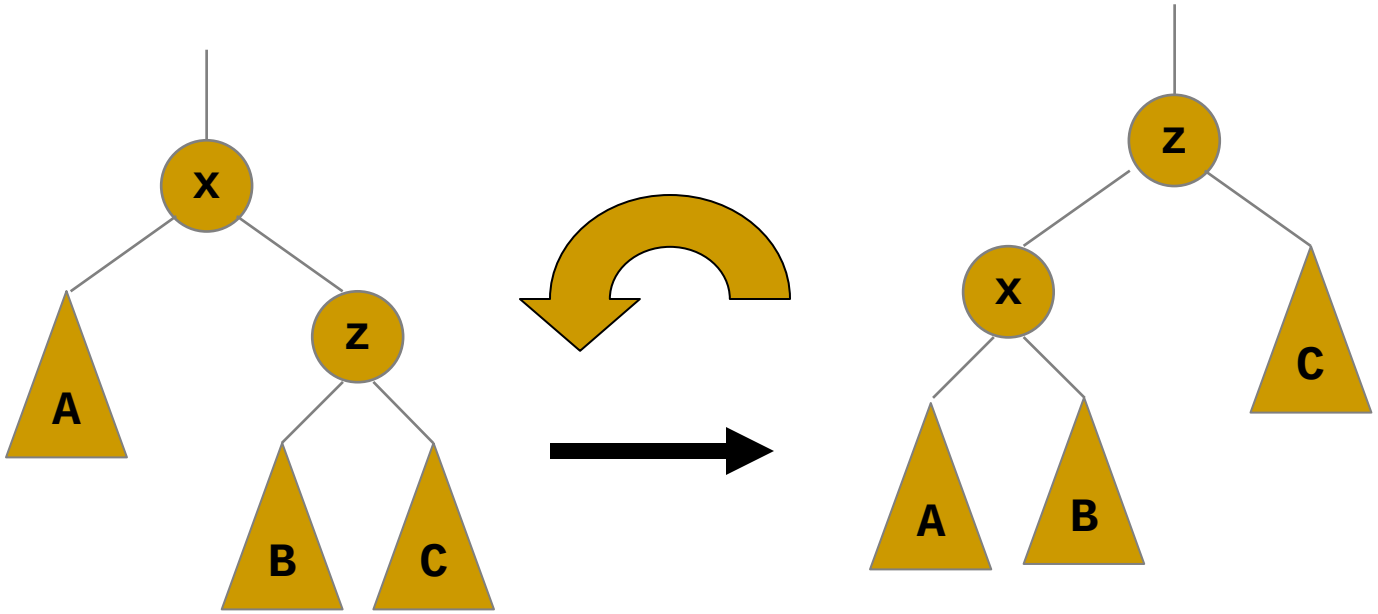
Rotations



- An item must be inserted into a **red-black** tree at the correct position
- The shape of any particular tree is determined by:
  - the values of the items inserted into the tree
  - the order in which those values are inserted
- A tree's shape can be altered by *rotation* while still preserving the **bst** property

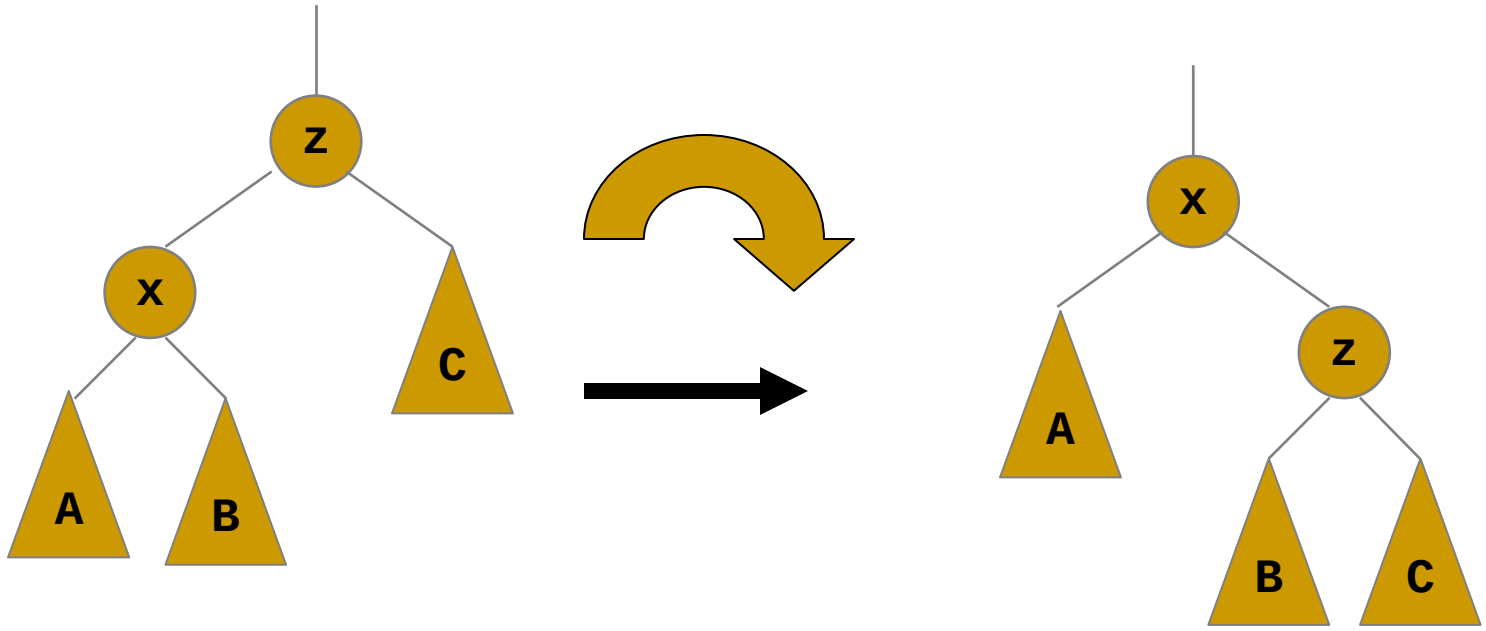
Left Rotation

```
Left rotate(x)
```



Right Rotation

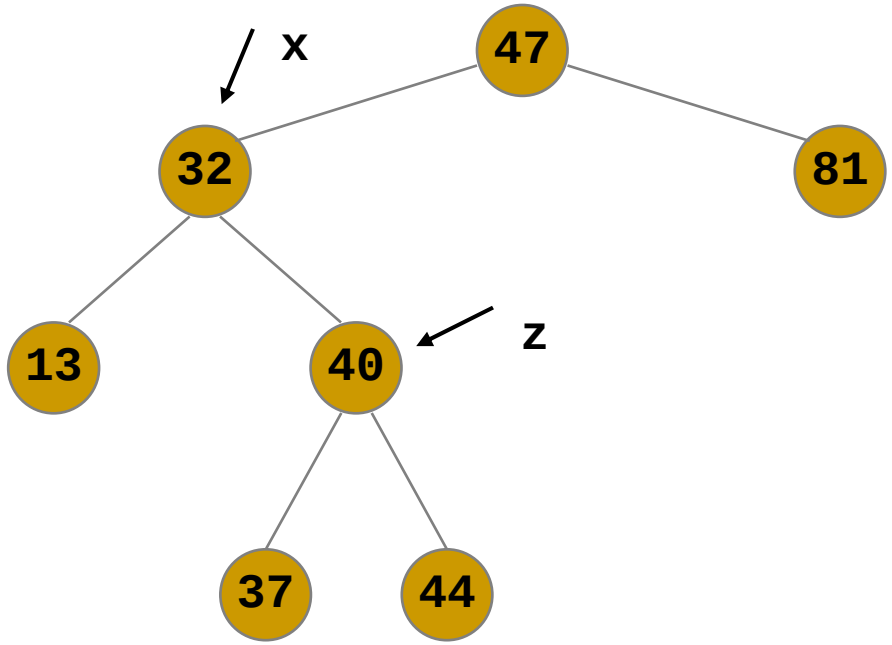
```
Right rotate(z)
```

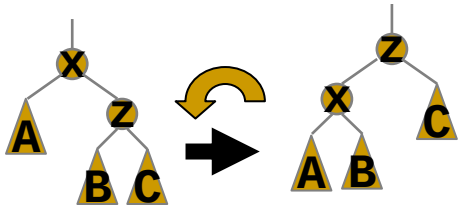


Left Rotation Example

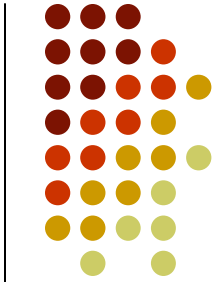
Perform a left rotation of the node with the value 32, which we will call x

- Make a reference to the right child of x



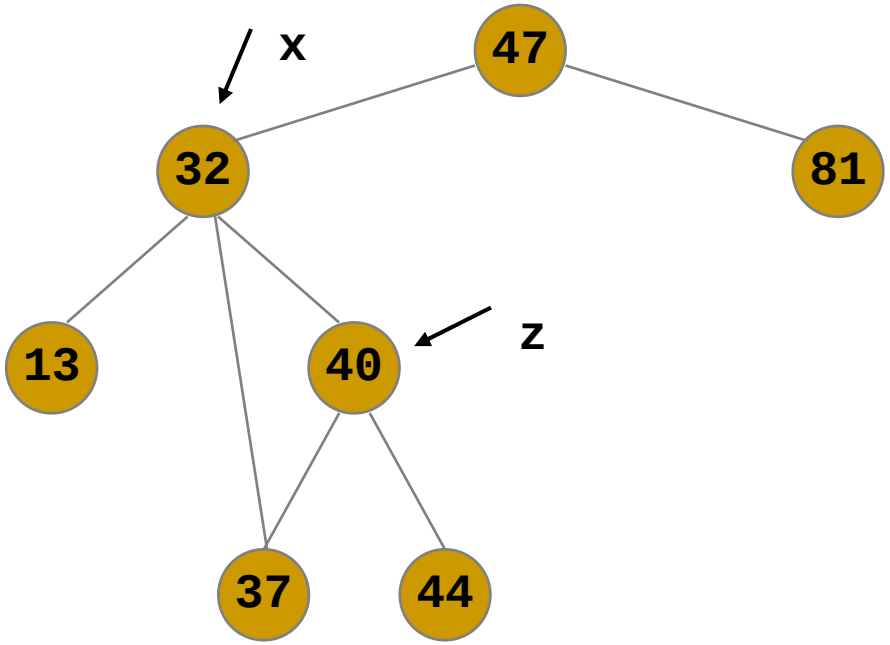


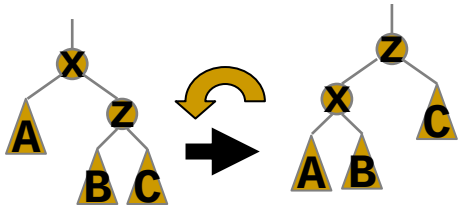
Left Rotation Example



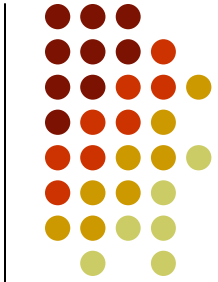
Perform a left rotation of the node with the value 32, which we will call x

- Make a reference to the right child of x
- Make z's left child x's right child
- Detach z's left child



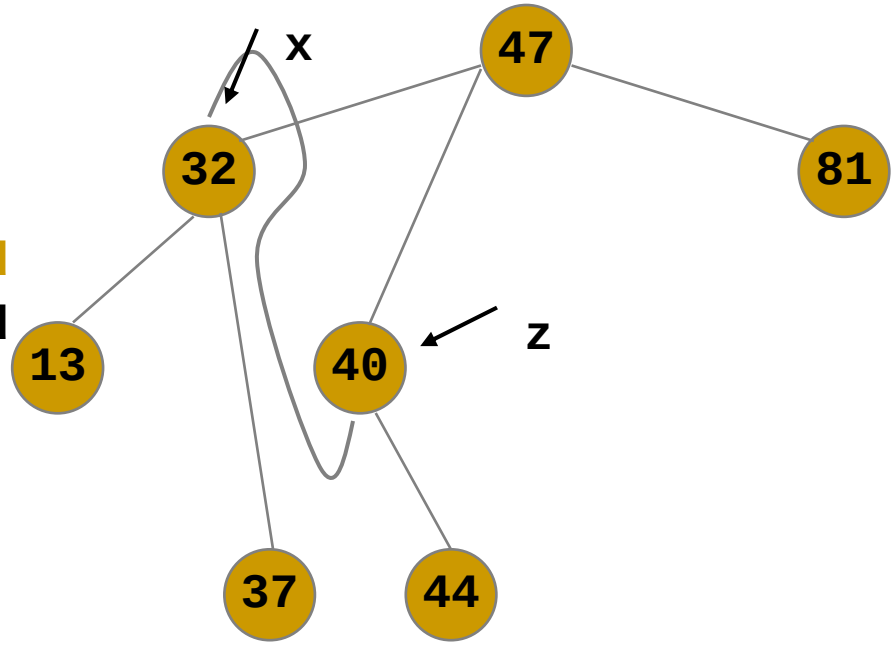


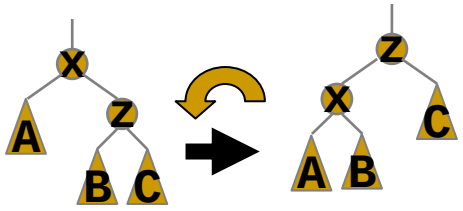
Left Rotation Example



Perform a left rotation of the node with the value 32, which we will call x

- Make a reference to the right child of x
- Make z's left child x's right child
- Detach z's left child
- Make x the left child of z
- Make z the child of x's parent

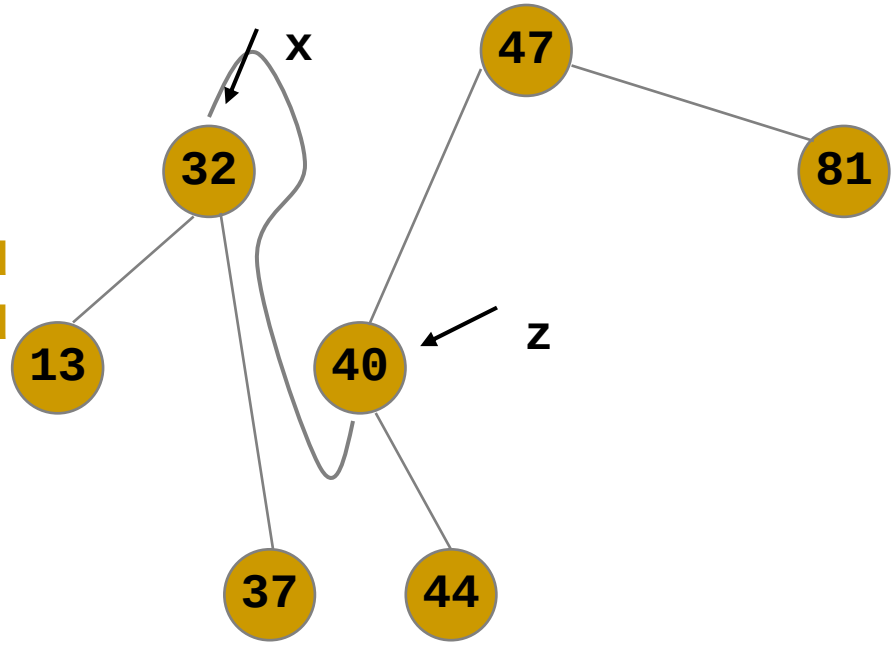


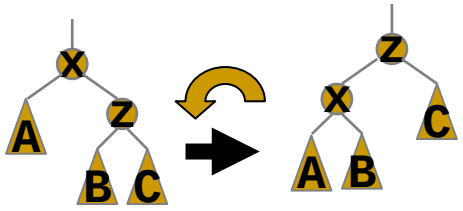


Left Rotation Example

Perform a left rotation of the node with the value 32, which we will call x

- Make a reference to the right child of x
- Make z's left child x's right child
- Detach z's left child
- Make x the left child of z
- Make z the child of x's parent

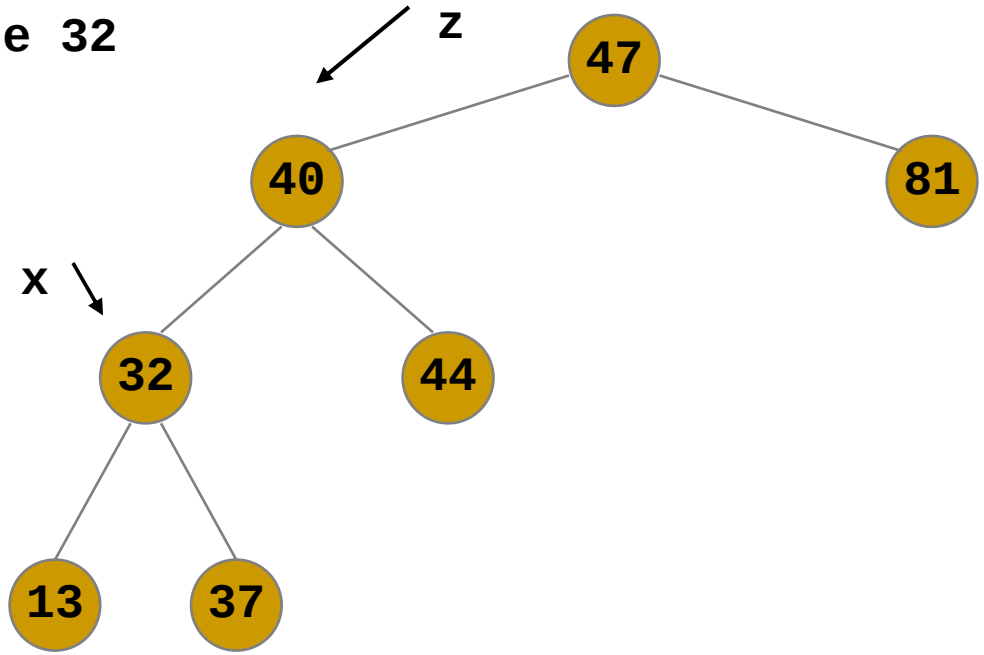


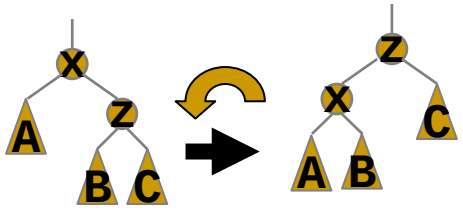


Left Rotation Example

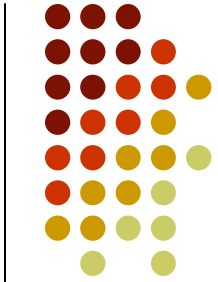
After

Performing a left rotation of the node with the value 32





Left Rotation: Algorithm

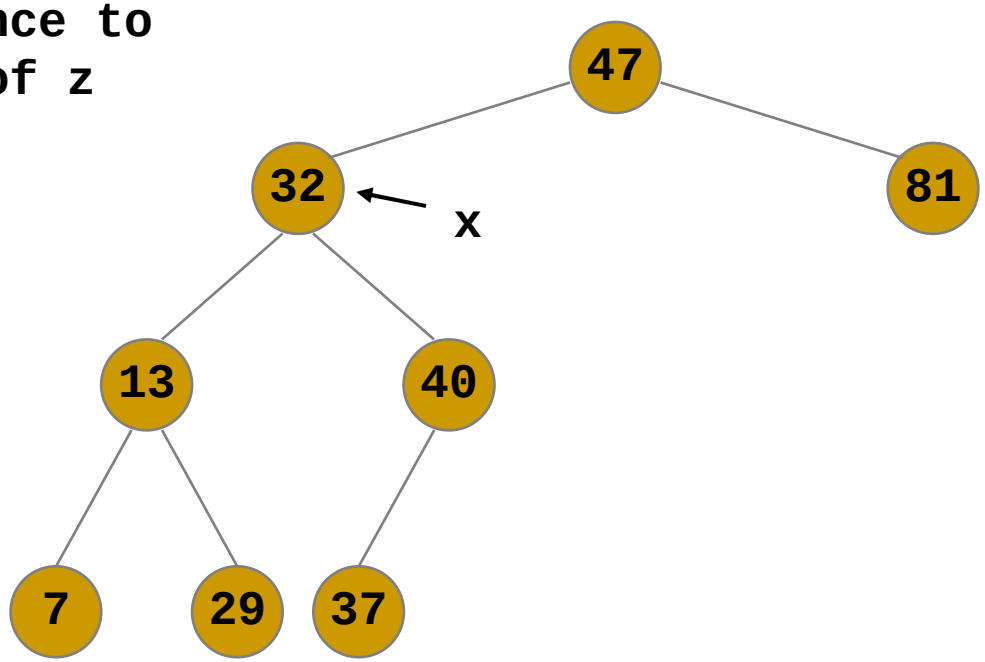


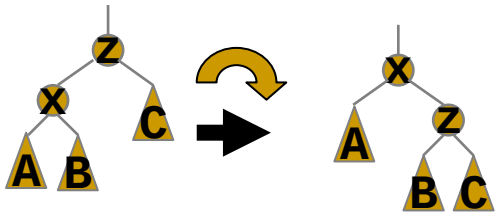
```
TreeNode<T> leftRotate(TreeNode<T> x)
// returns a new root of the subtree originally rooted in x
// (that is: returns a reference to z)
// Pre: rigth child of x is a proper node (with value)
{
  TreeNode<T> z = x.getRight();

  x.setRight(z.getLeft());
  // Set parent reference
  if (z.getLeft() != null)
    z.getLeft().setParent(x);

  z.setLeft(x); //move x down

  z.setParent(x.getParent());
  // Set parent reference of x
  if (x.getParent() != null) //x is not the root
    if (x == x.getParent().getLeft()) //left child
      x.getParent().setLeft(z);
    else
      x.getParent().setRight(z);

  x.setParent(z);
  return z;
}
```

Right Rotation Example

Perform a right rotation of the node with the value 47, which we will call z

- Make a reference to the left child of z

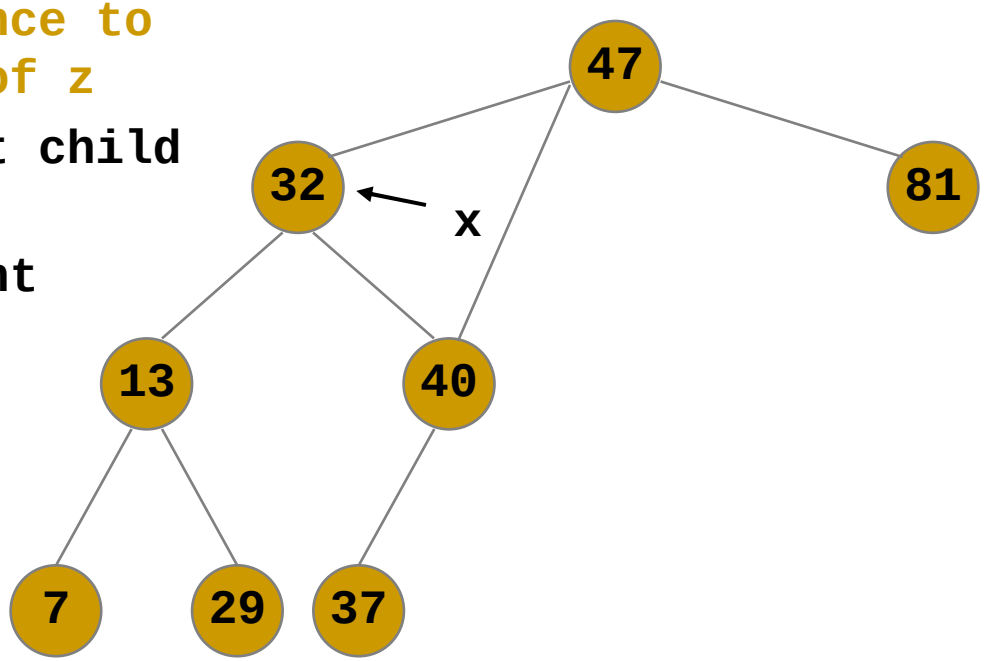


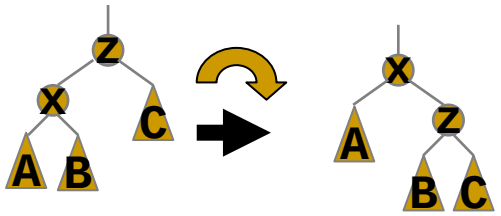


Right Rotation Example

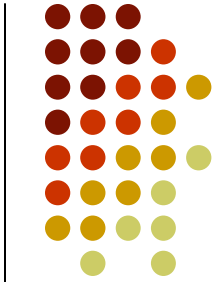
Perform a right rotation of the node with the value 47, which we will call z

- Make a reference to the left child of z
- Make x's right child z's left child
- Detach x's right child



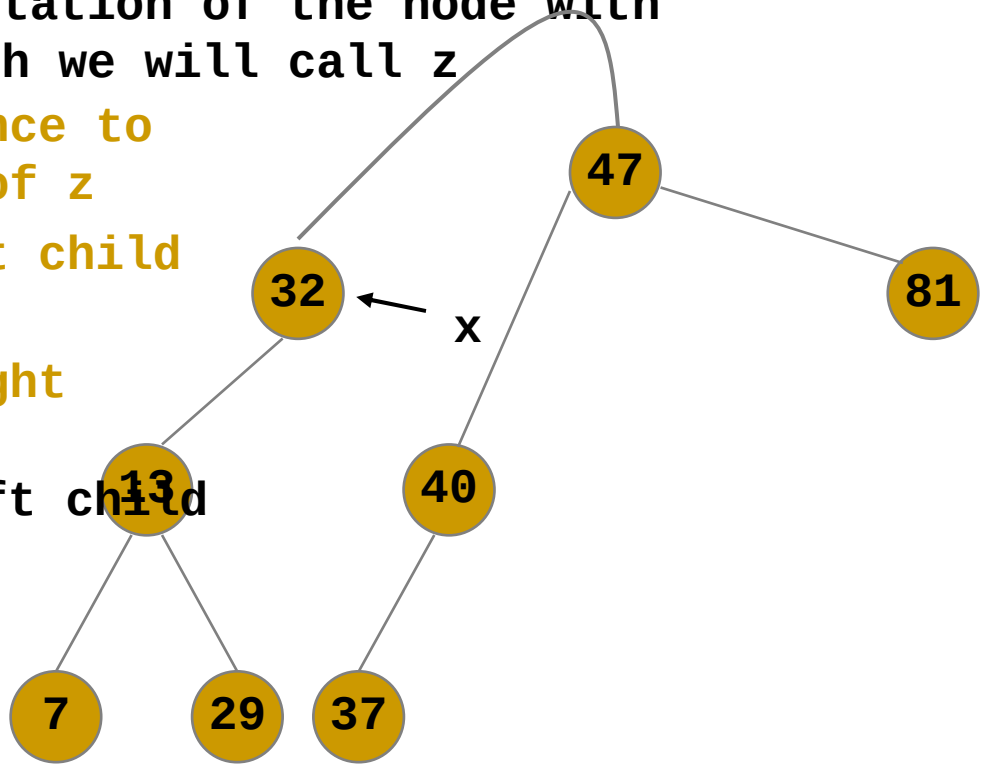


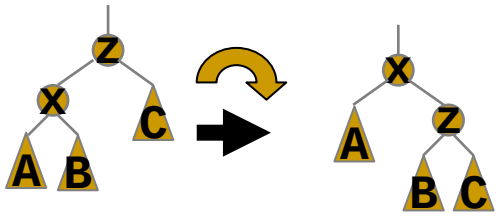
Right Rotation Example



Perform a right rotation of the node with the value 47, which we will call z

- Make a reference to the left child of z
- Make x's right child z's left child
- Detach x's right child
- Make z the left child of x

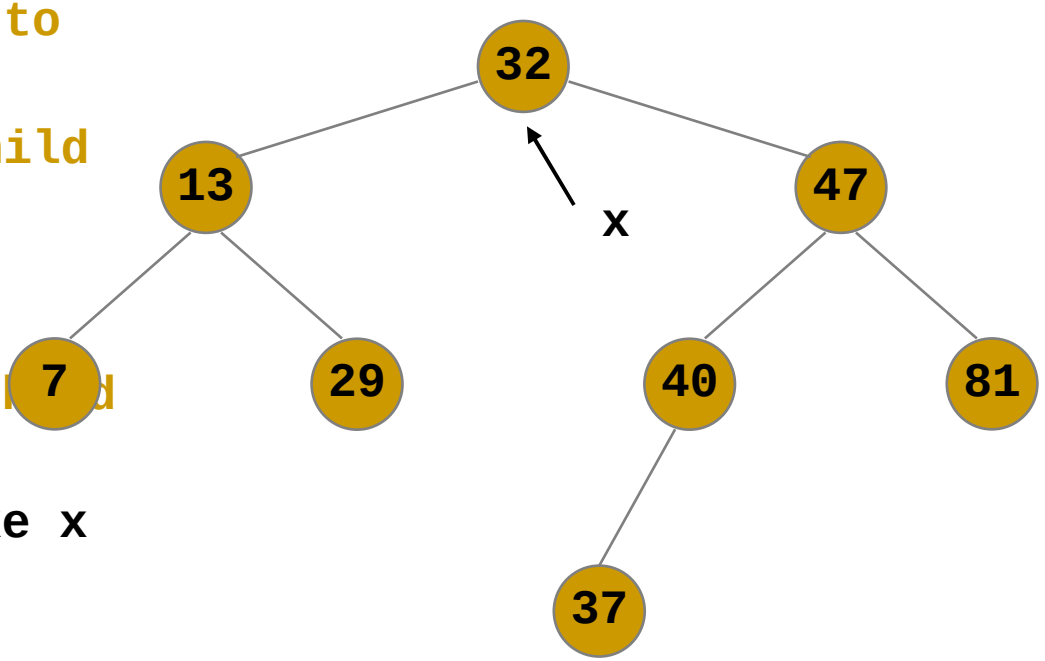


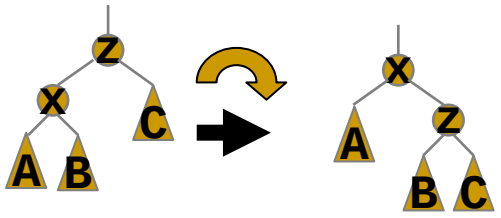


Right Rotation Example

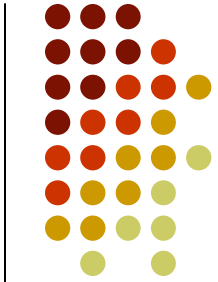
Perform a right rotation of the node with the value 47, which we will call z

- Make a reference to the left child of z
- Make x's right child z's left child
- Detach x's right child
- Make z the left child of x
- In this case, make x the new root





RB Tree: Operations



- *Search*: the same as Search for BST (cannot spoil RB properties)
- *Insertion*: first insert with BST insertion algorithm, then fix RB properties
- *Deletion*: first delete with BST deletion algorithm, then fix RB properties

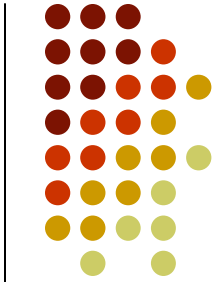
To fix use:

- rotate operations
- recoloring of nodes

-> neither of them will spoil the BST property

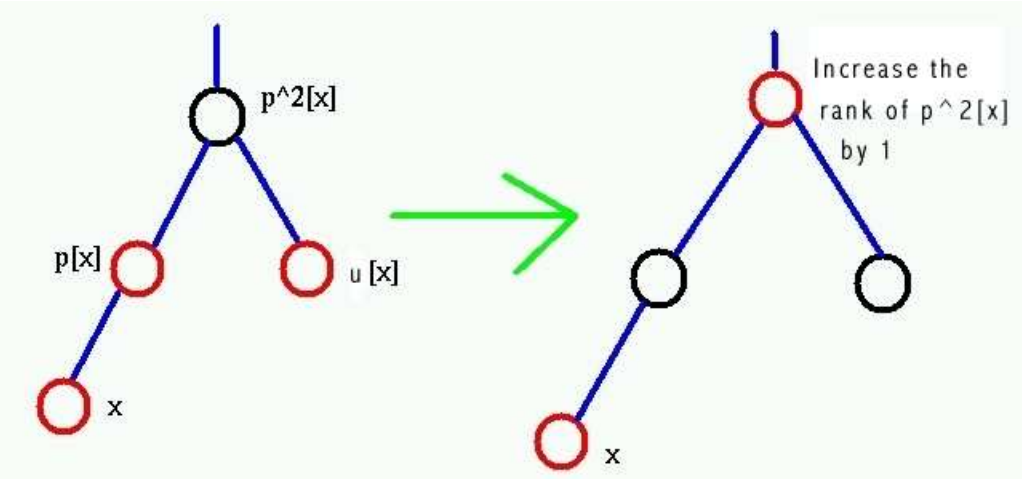
Demonstration of Red/Black Tree operations:
http://www.ece.uc.edu/~franco/C321/html/RedBlack/

Red-black Tree Insertion

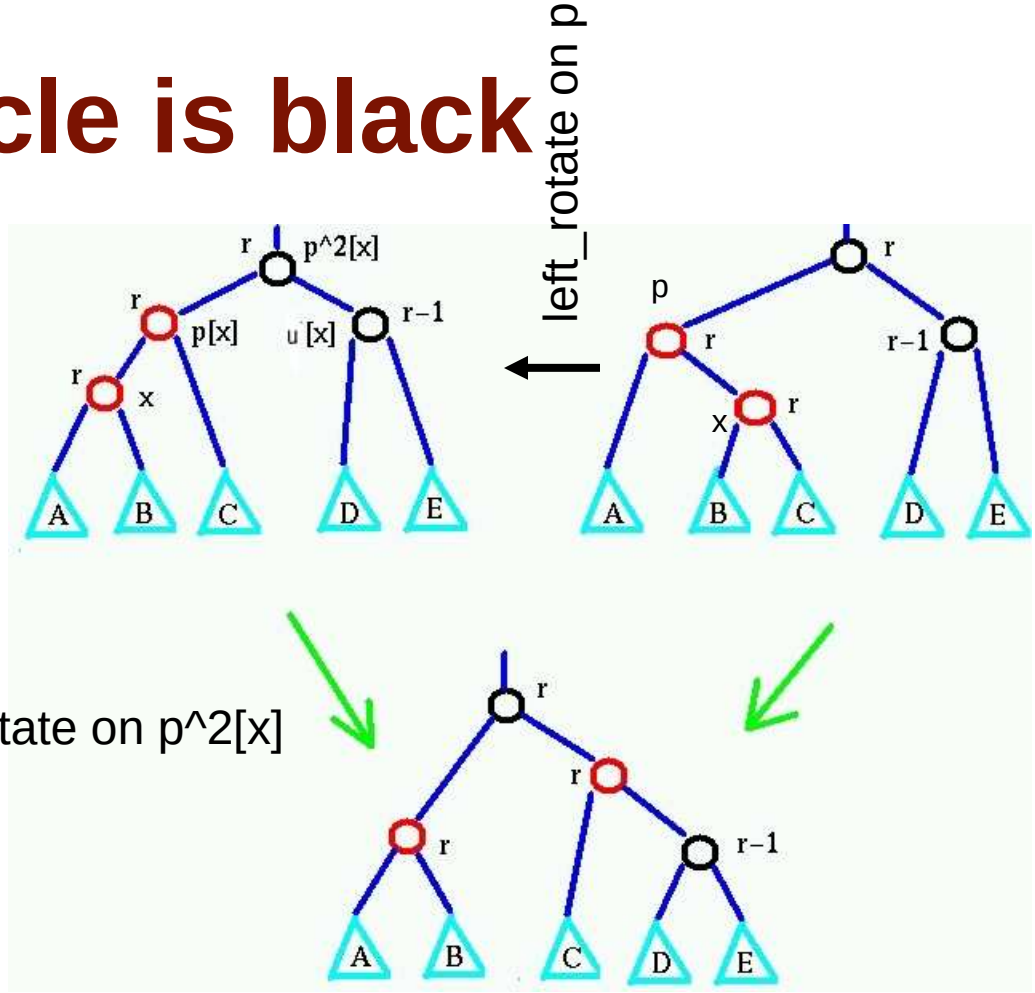


- First perform the standard BST insertion
- Color the inserted node x with color red
  - The only **r-b** property that can be violated is #3, that both a red node's children are black
- If x's parent p is black, we are done
- Assume it's red:
  - If x's uncle is red, recolor x's parent, its parent and uncle, and continue recursively
  - If x's uncle is black, first make sure x is left child (left_rotate in p if necessary); second right_rotate in parent of p

x’s uncle is red



x's uncle is black



- Assuming that p is left child of its parent, in the other case, everything is symmetric