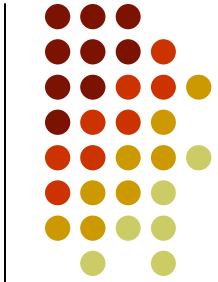
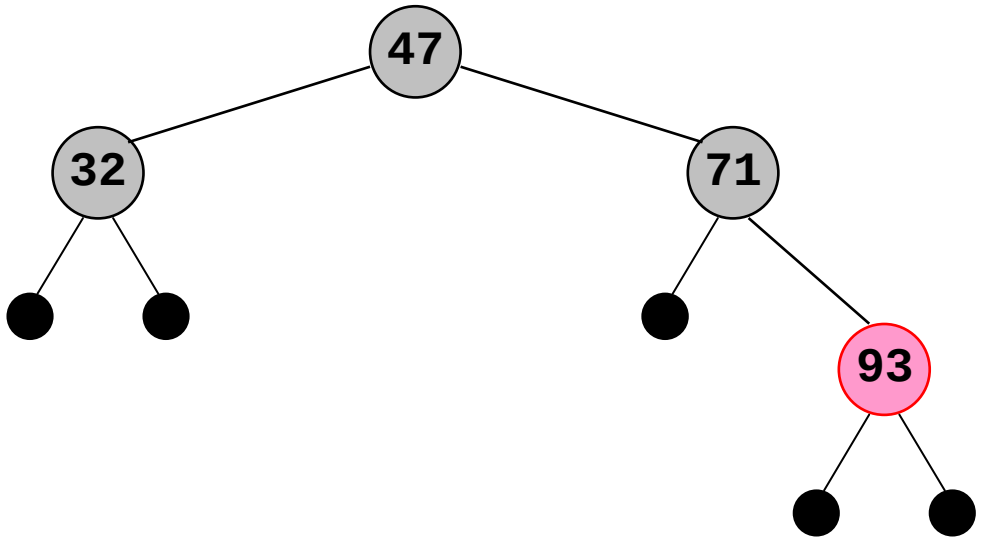
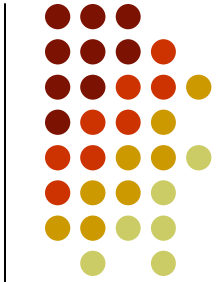
Insertion Example



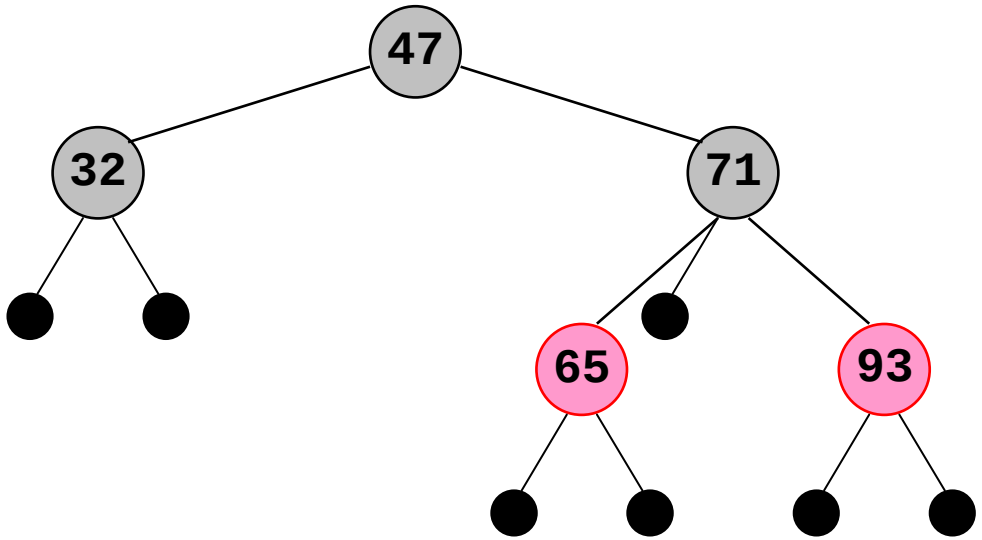
```
Insert 65
```



Insertion Example

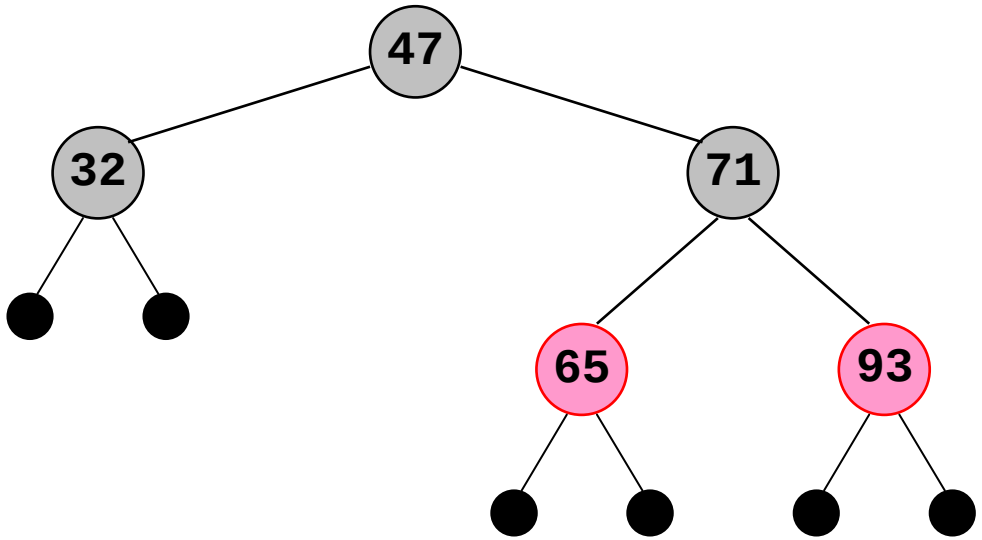


```
Insert 65
```



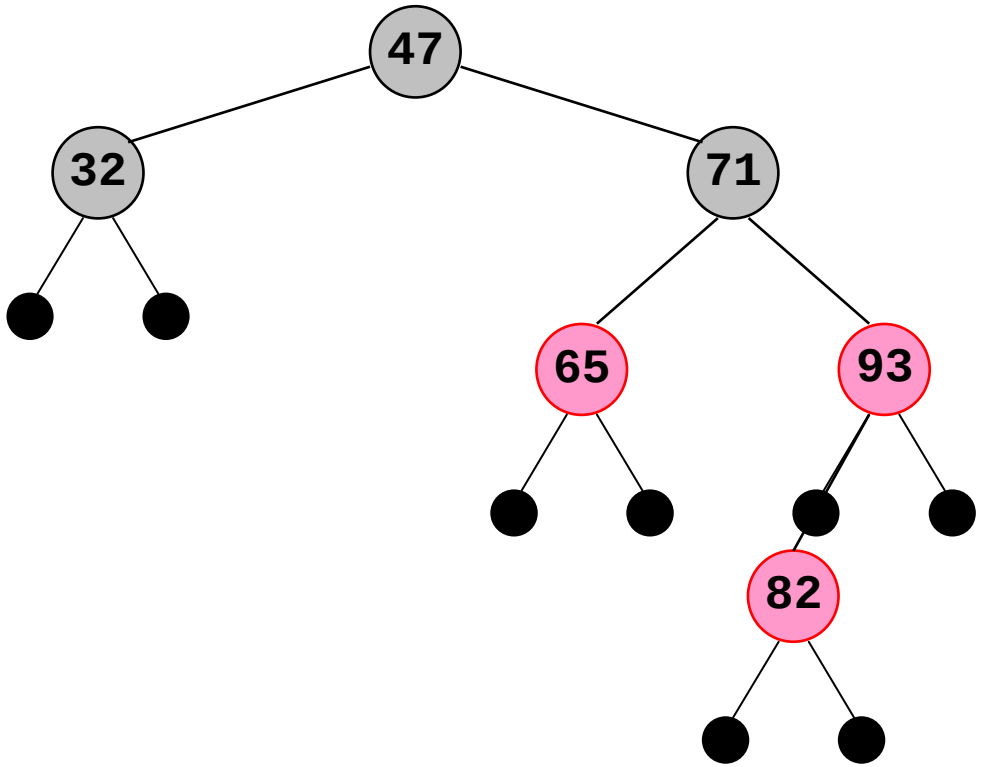
Insertion Example

```
Insert 65
Insert 82
```



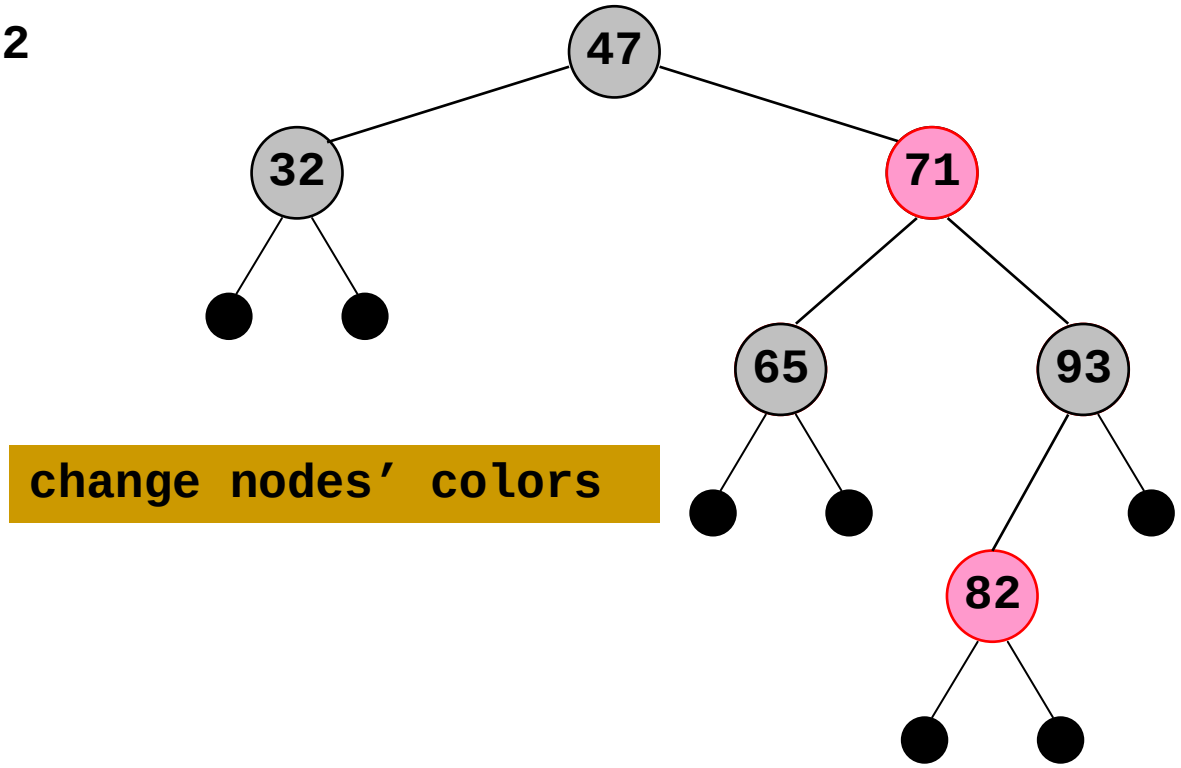
Insertion Example

```
Insert 65
Insert 82
```



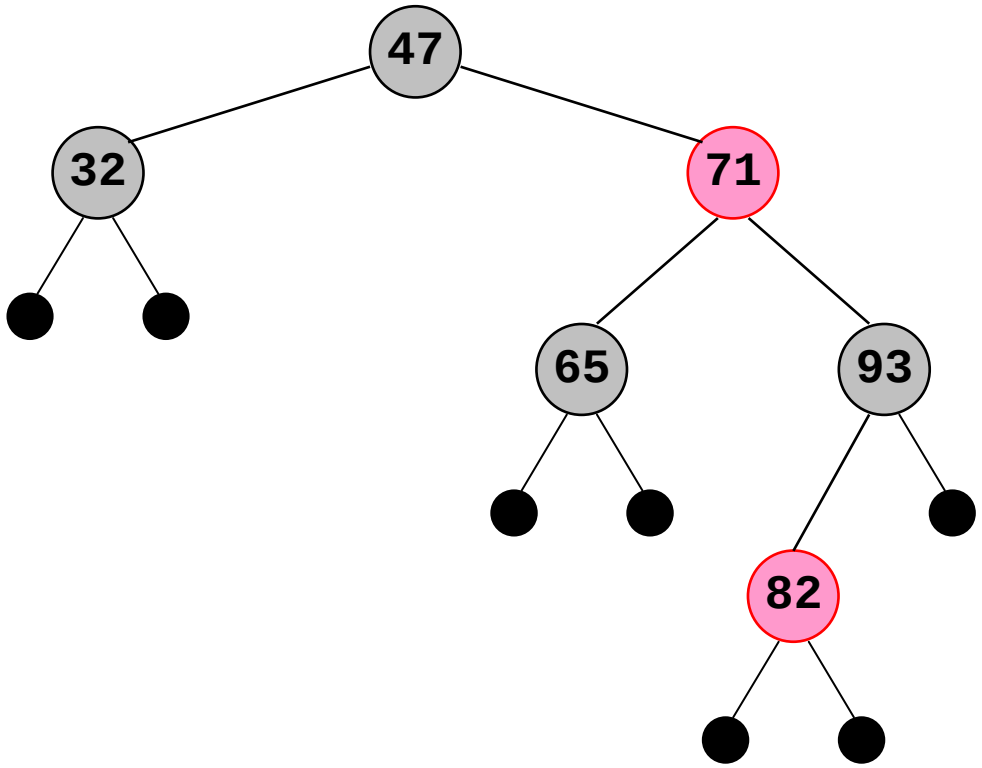
Insertion Example

Insert 65
Insert 82



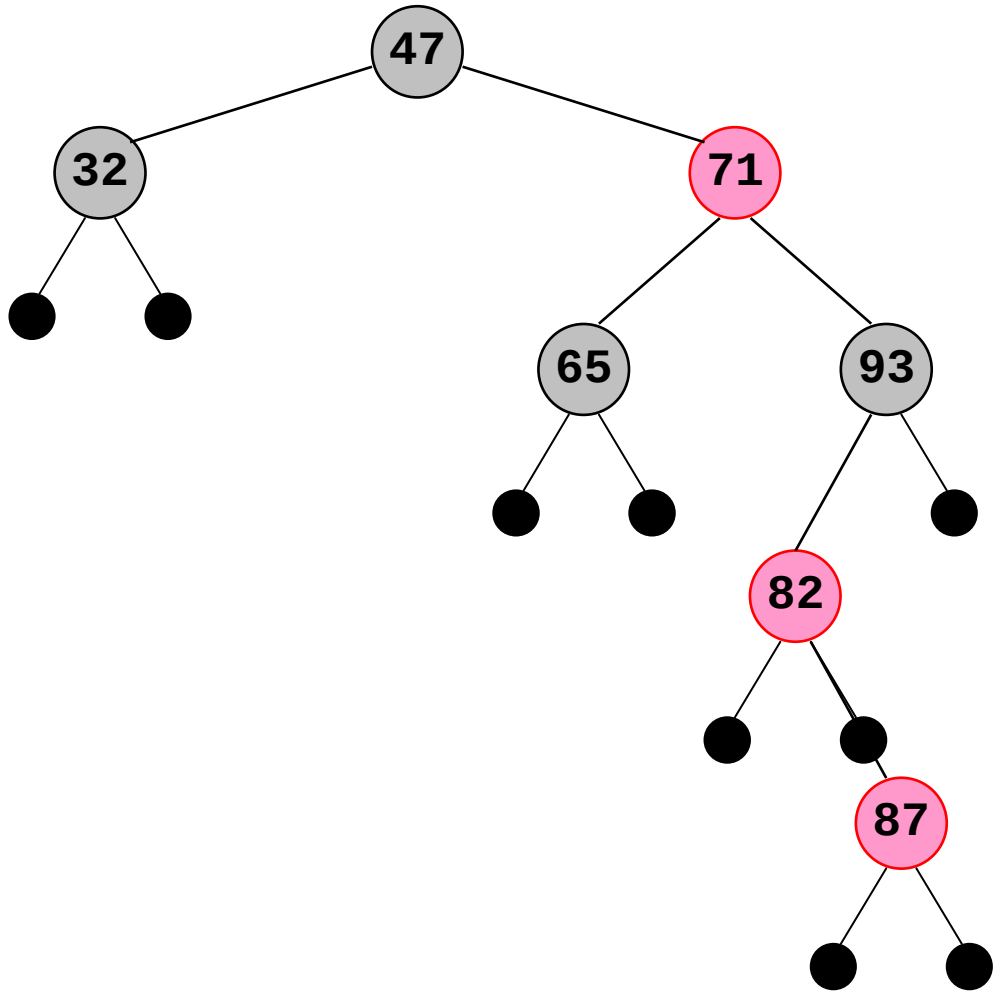
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



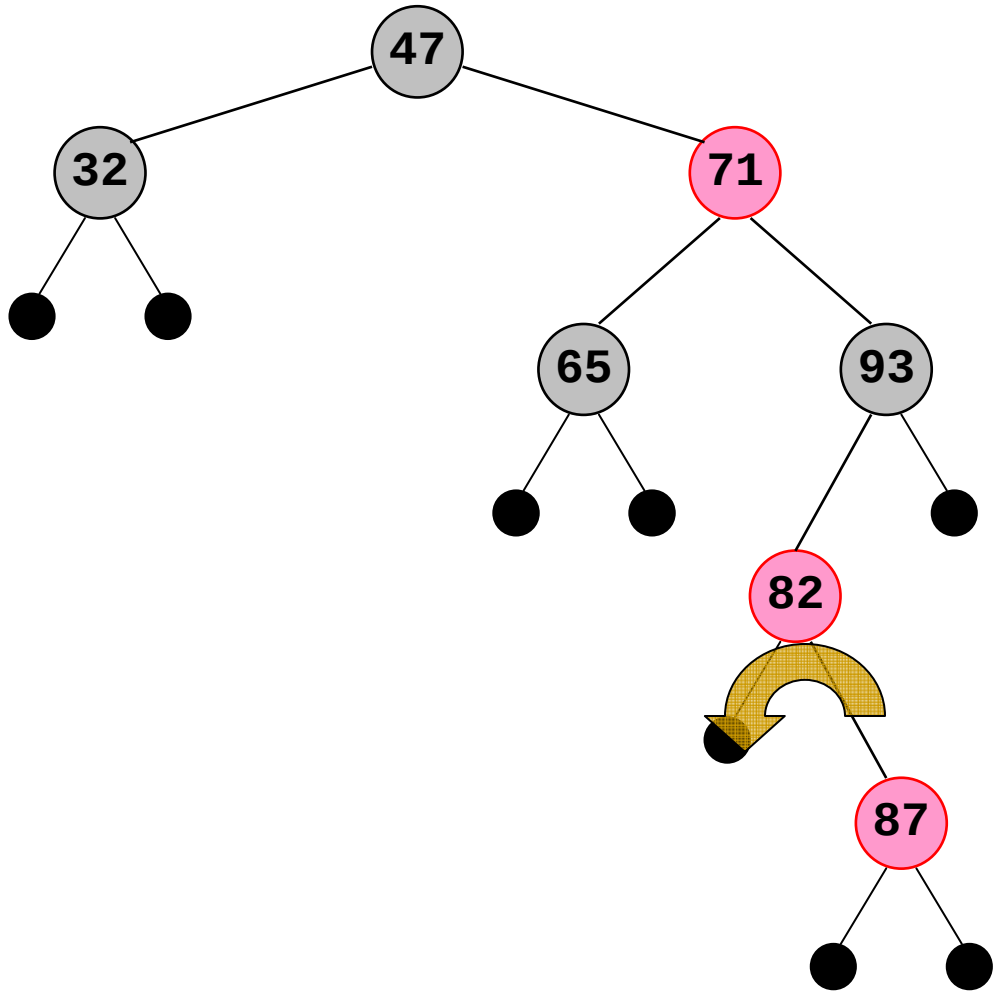
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



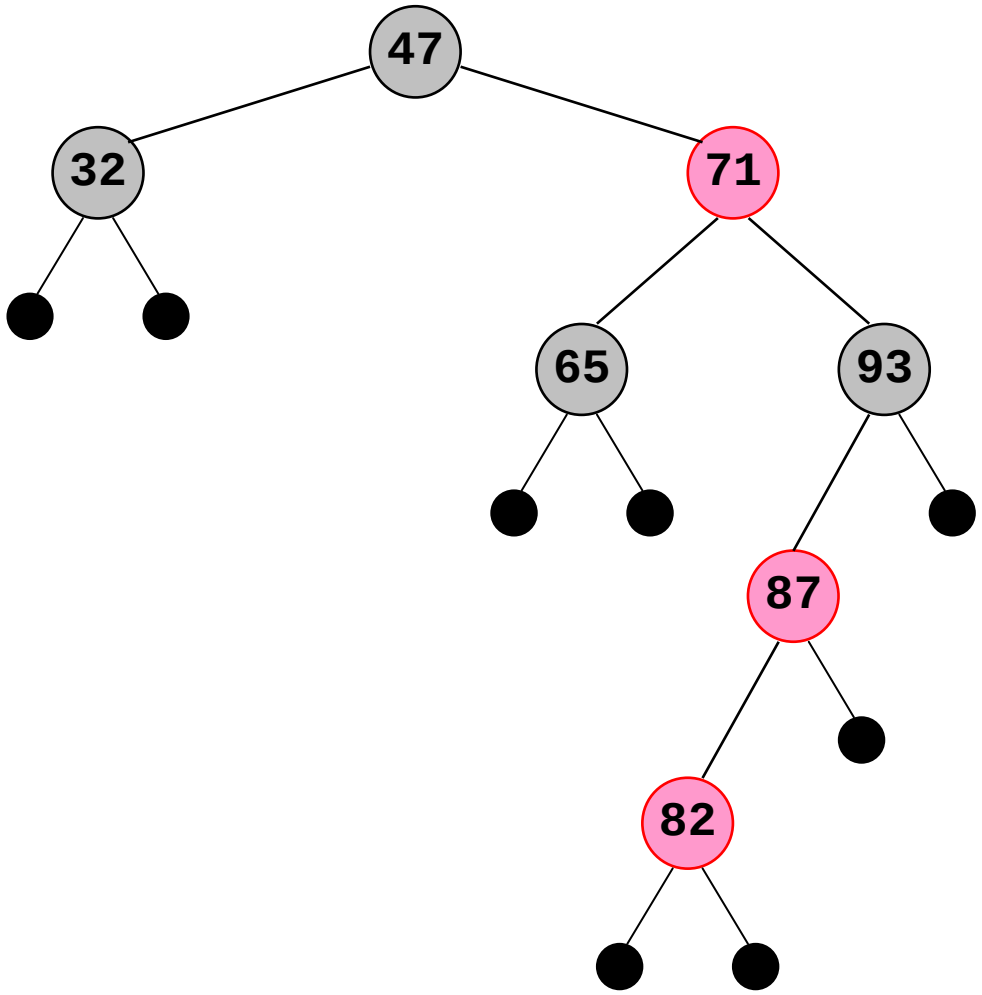
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



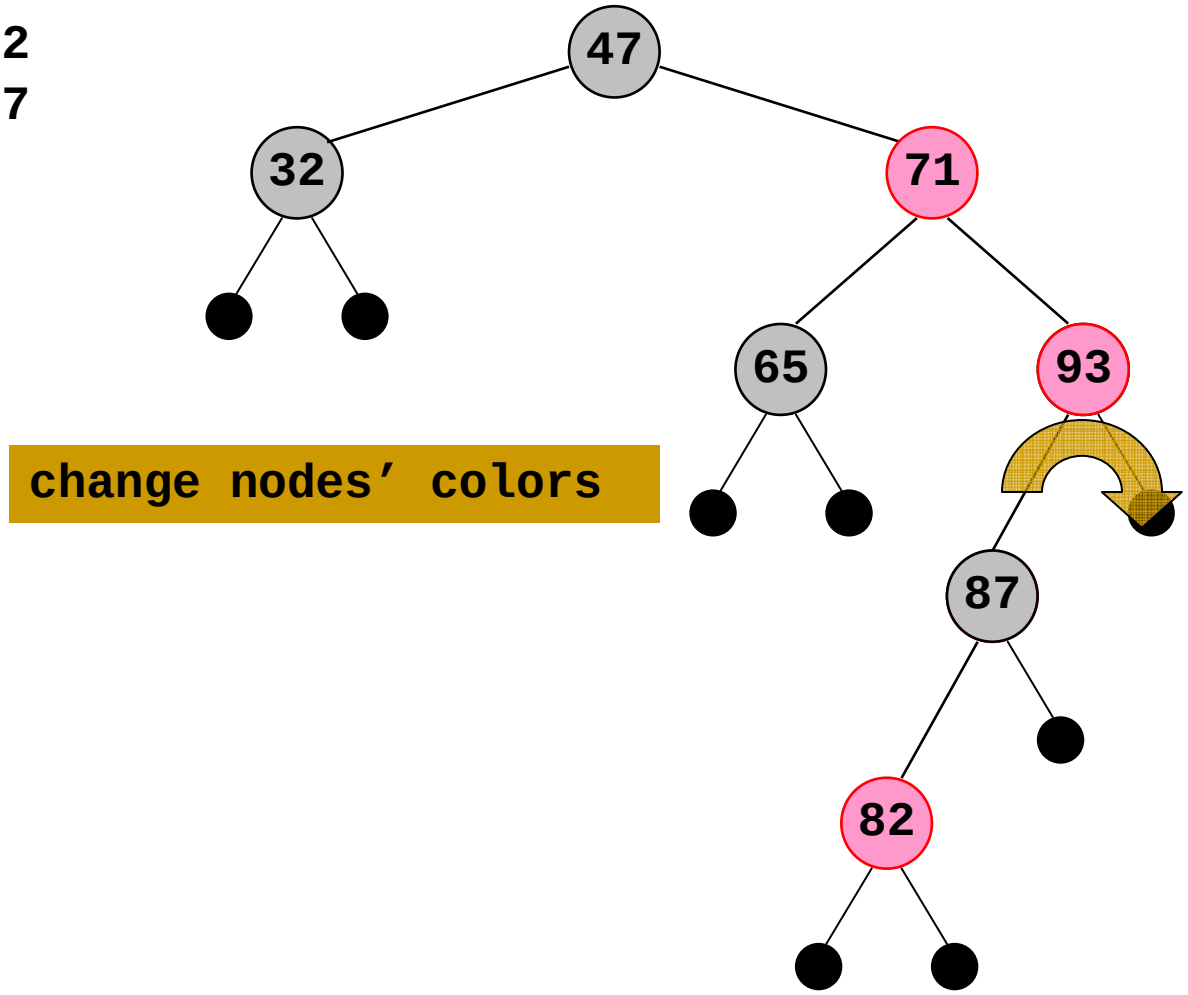
Insertion Example

```
Insert 65
Insert 82
Insert 87
```



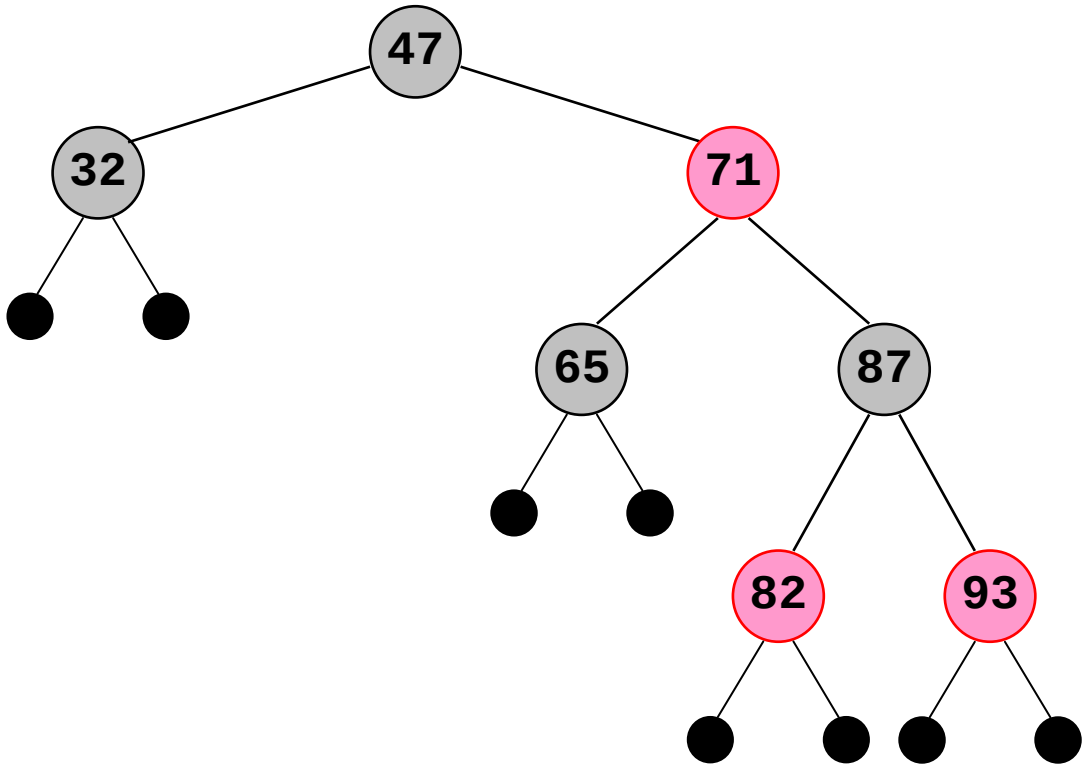
Insertion Example

```
Insert 65
Insert 82
Insert 87
```

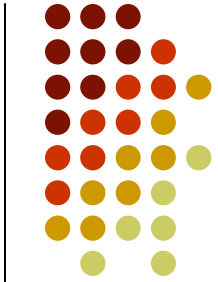


Insertion Example

```
Insert 65
Insert 82
Insert 87
```



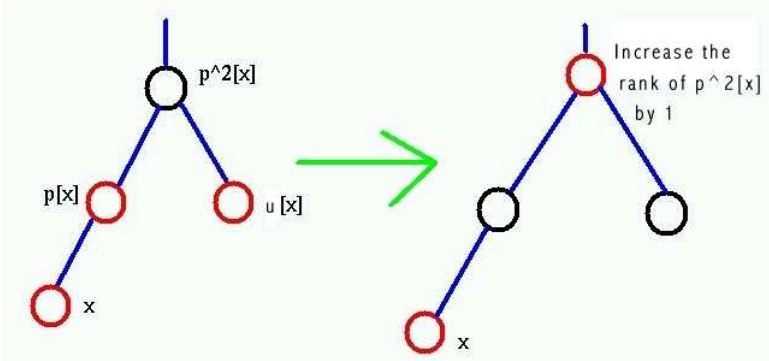
Left Rotation: Modified algorithm



```
TreeNode<T> leftRotate(TreeNode<T> root,TreeNode<T> x)
//returns a new root; Pre: right 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);
  else
    root=z;

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

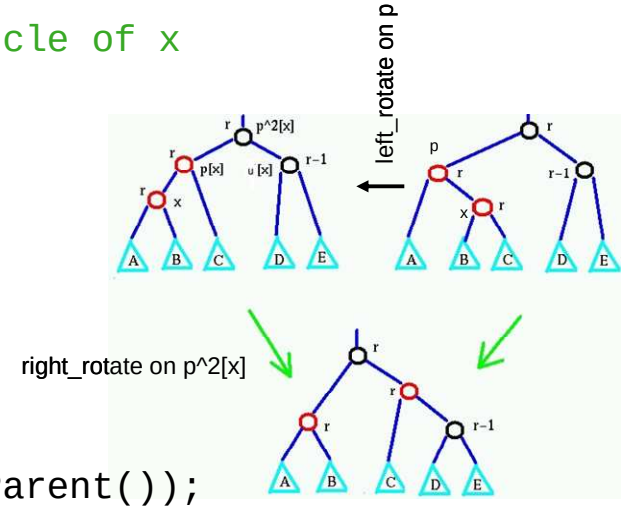
RB Tree: Insertion Algorithm

```
TreeNode<T> rbInsert(TreeNode<T> root,TreeNode<T> x)
// returns a new root
{
  root=bstInsert(root,x); // a modification of BST insertItem
  x.setColor(red);
  while (x != root and x.getParent().getColor() == red) {
    if (x.getParent() == x.getParent().getParent().getLeft()) {
      //parent is left child
      y = x.getParent().getParent().getRight() //uncle of x
      if (y.getColor() == red) {// uncle is red
        x.getParent().setColor(black);
        y.setColor(black);
        x.getParent().getParent().setColor(red);
        x = x.getParent().getParent();
      } else { // uncle is black
        // ................
      }
    } else
      // ... symmetric to if
  } // end while
  root.setColor(black);
  return root;
}
```

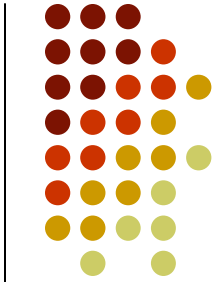


RB Tree: Insertion Algorithm

```
TreeNode<T> rbInsert(TreeNode<T> root,TreeNode<T> newNode)
// returns a new root
{
  root=bstInsert(root,newNode); // a modification of BST insertItem
  x.setColor(red);
  while (x != root and x.getParent().getColor() == red) {
    if (x.getParent() == x.getParent().getParent().getLeft()) {
      //parent is left
      y = x.getParent().getParent().getRight() //uncle of x
      if (y.getColor() == red) {// uncle is red
        // ................
      } else { // uncle is black
        if (x == x.getParent().getRight()) {
          x = x.getParent();
          root = left_rotate(root,x);
        }
        x.getParent().setColor(black);
        x.getParent().getParent().setColor(red);
        root = right_rotate(root,x.getParent().getParent());
      }
    } else
      // ... symmetric to if
  } // end while
  root.setColor(black);
  return root;
}
```



Red-black **Tree Deletion**



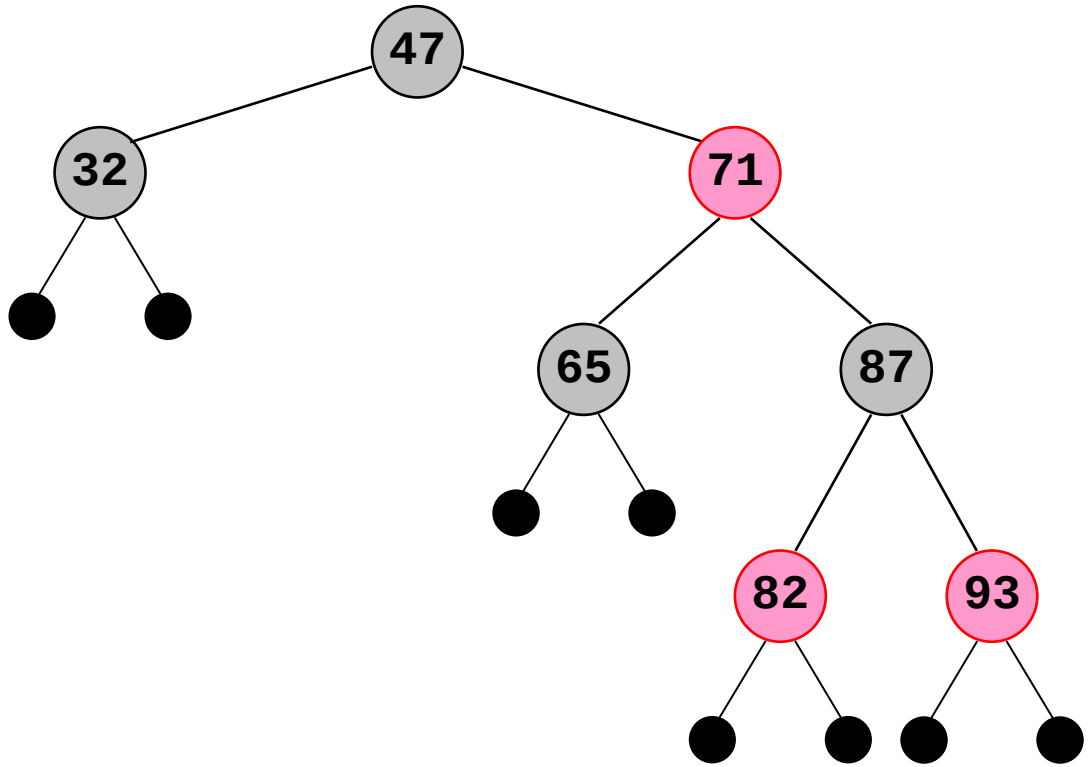
- First use the standard BST tree deletion algorithm
  - If the node to be deleted is replaced by its successor/predecessor (if it has two non-null children), consider the deleted node's data as being replaced by it's successor/predecessor's, and its color remaining the same
    - The successor/predecessor node is then removed
- Let y be the node to be removed
- If the removed node was red, no property could get violated, so just remove it.
- Otherwise, remove it and call the tree-fix algorithm on y's child x (the node which replaced the position of y)
  - Remember, the removed node can have at most one real (non-null) child
  - If it has one real child, call the tree-fix algorithm on it
  - If it has no real children (both children are null), Note that this child may be a (black) pretend (null) child

Fixing a red-black Tree

- The tree-fix algorithm considers the parameter (**x**) as having an *"extra" black token*
  - This corrects the violation of property 4 caused by removing a black node
- If **x** is red, just color it black
- But if **x** is black then it becomes "doubly black"
  - This is a violation of property 1
  - The extra black token is pushed up the tree until
    - a red node is reached, when it is made black
    - the root node is reached or
    - it can be removed by rotating and recoloring

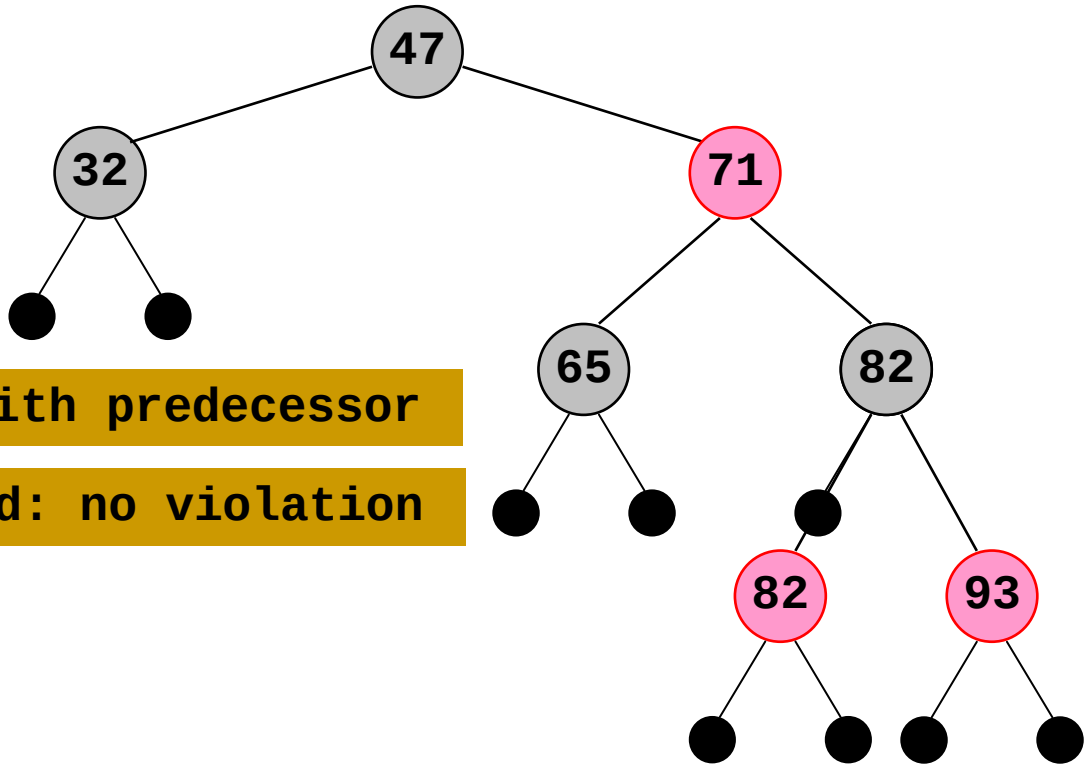
Deletion Example 1

Delete 87



Deletion Example 1

Delete 87

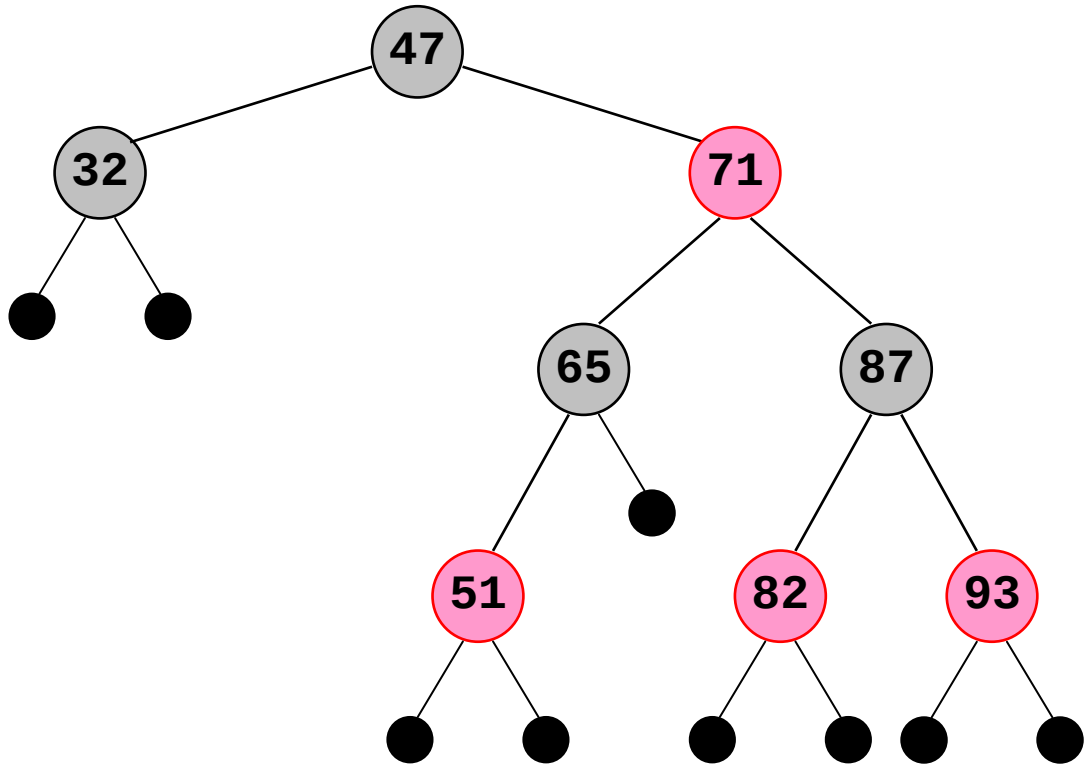


Replace data with predecessor

Predecessor red: no violation

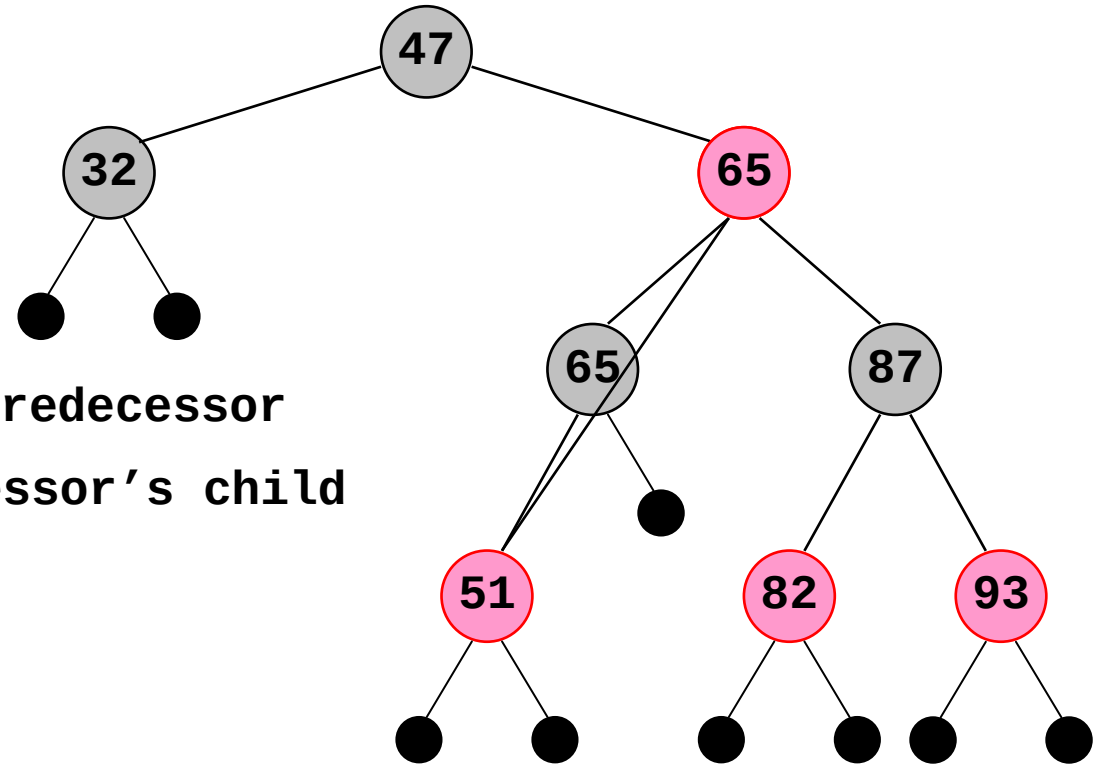
Deletion Example 2

Delete 71



Deletion Example 2

Delete 71

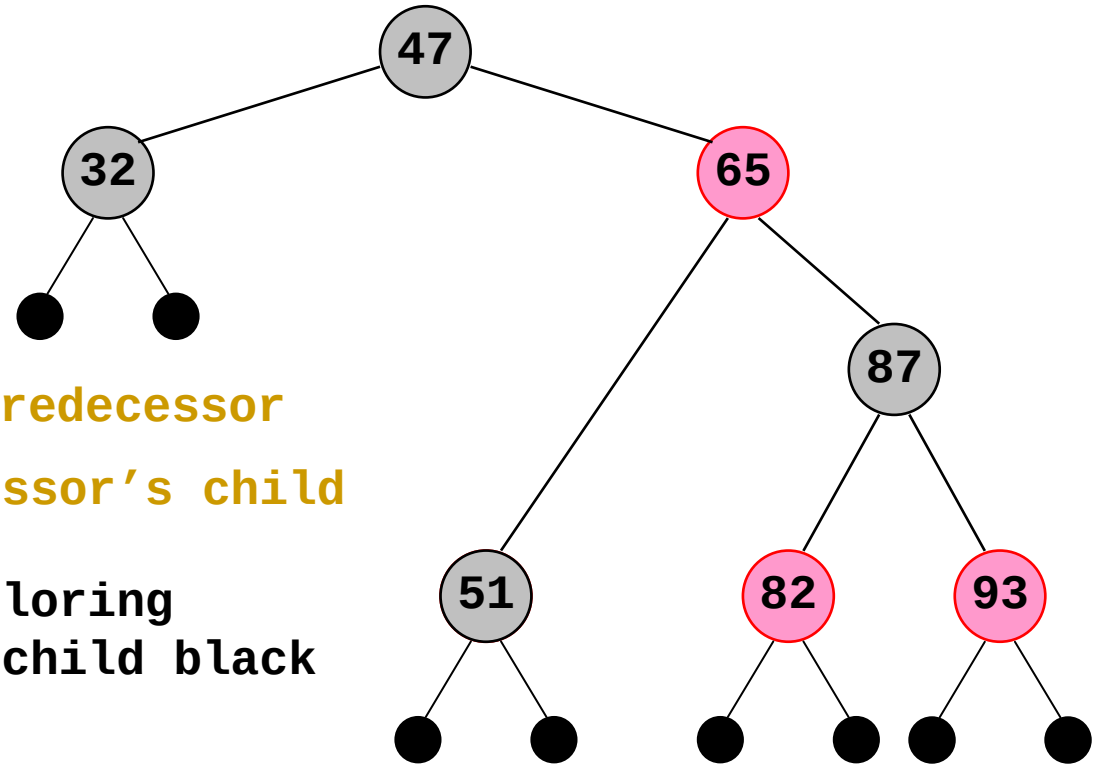


Replace with predecessor

Attach predecessor's child

Deletion Example 2

Delete 71



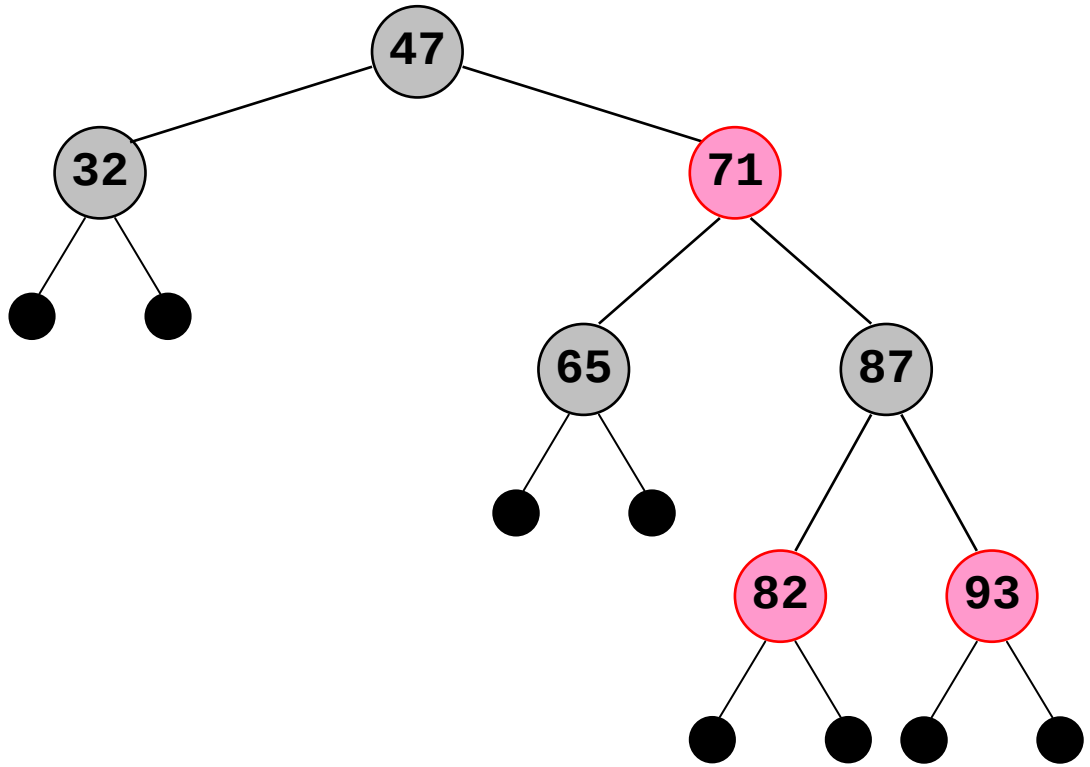
Replace with predecessor

Attach predecessor's child

Fix tree by coloring predecessor's child black

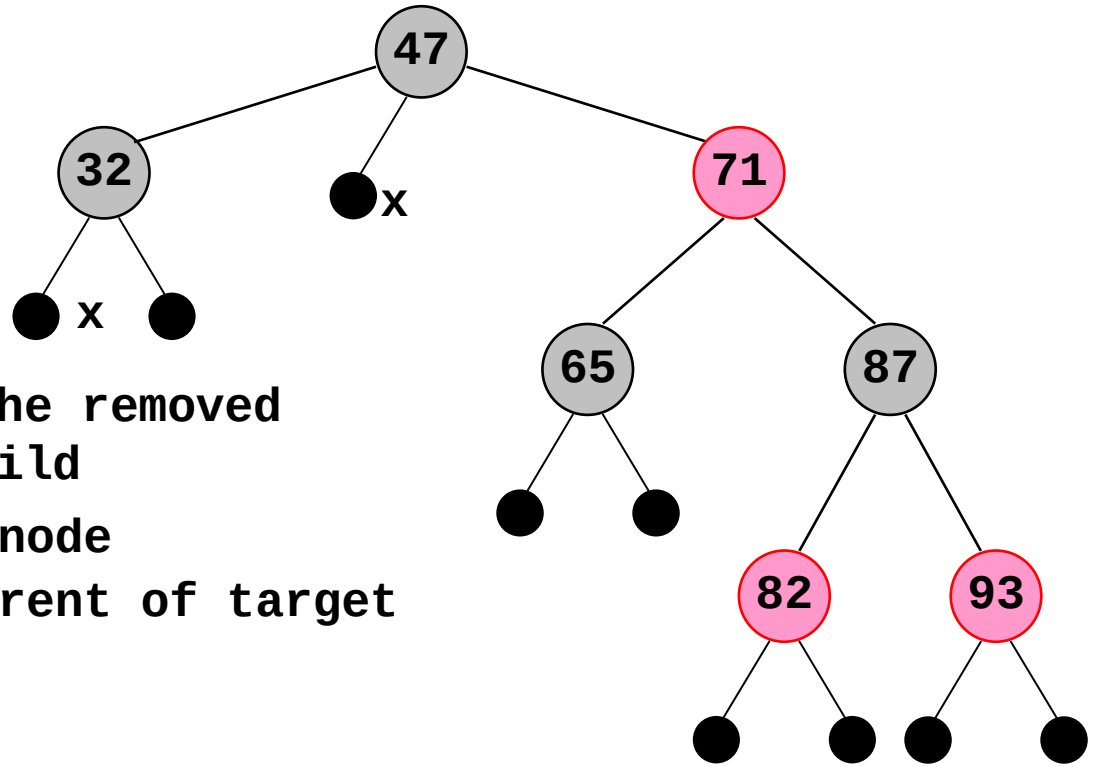
Deletion Example 3

Delete 32



Deletion Example 3

Delete 32



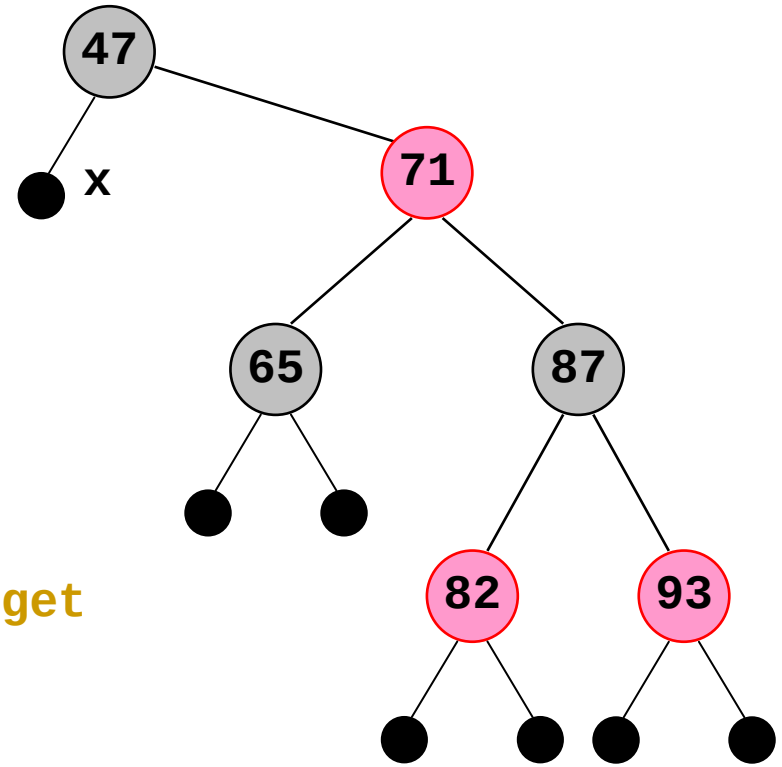
Identify x – the removed node's left child

Remove target node

Attach x to parent of target

Deletion Example 3

Delete 32



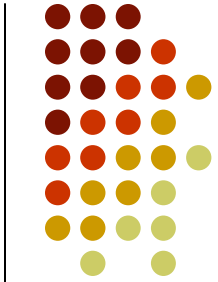
Identify x – the removed node's left child

Remove target node

Attach x to parent of target

Call rbTreeFix on x

RB Tree Deletion Algorithm



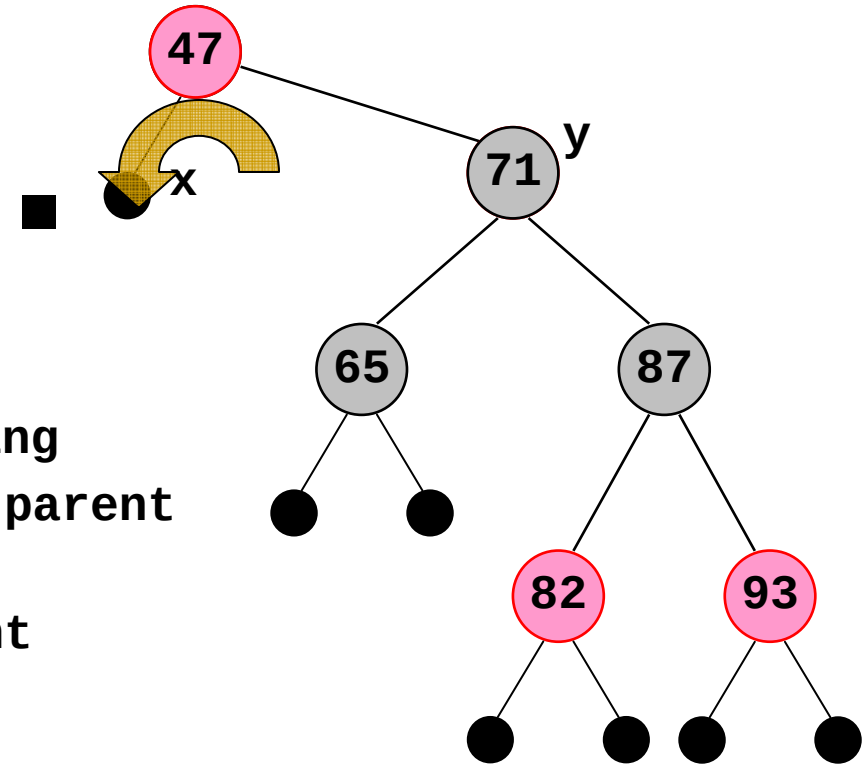
```
TreeNode<T> rbDelete(TreeNode<T> root,TreeNode<T> z)
//return new root, z contains item to be deleted
{
  TreeNode<T> x,y;
  // find node y, which is going to be removed
  if (z.getLeft() == null || z.getRight() == null)
    y = z;
  else {
    y = successor(z); // or predecessor
    z.setItem(y.getItem); // move data from y to z
  }

  // find child x of y
  if (y.getRight() != null)
    x = y.getRight();
  else
    x = y.getLeft();

  // Note x might be null; create a pretend node
  if (x == null) {
    x = new TreeNode<T>(null);
    x.setColor(black);
  }
```

RB Tree Deletion Algorithm

```
  x.setParent(y.getParent()); // detach x from y
  if (y.getParent() == null)
    // if y was the root, x is a new root
    root = x;
  else
    // Atttach x to y's parent
    if (y == y.getParent().getLeft()) // left child
      y.getParent().setLeft(x);
    else
      y.getParent().setRight(x);

  if (y.getColor() == black)
    root=rbTreeFix(root,x);

 if (x.getItem() == null) // x is a pretend node
    if (x==x.getParent().getLeft())
      x.getParent().setLeft(null);
    else
      x.getParent().setRight(null);

  return root;
}
```

Deletion Example 3 (continued)

After deleting 32, x is a node with black token



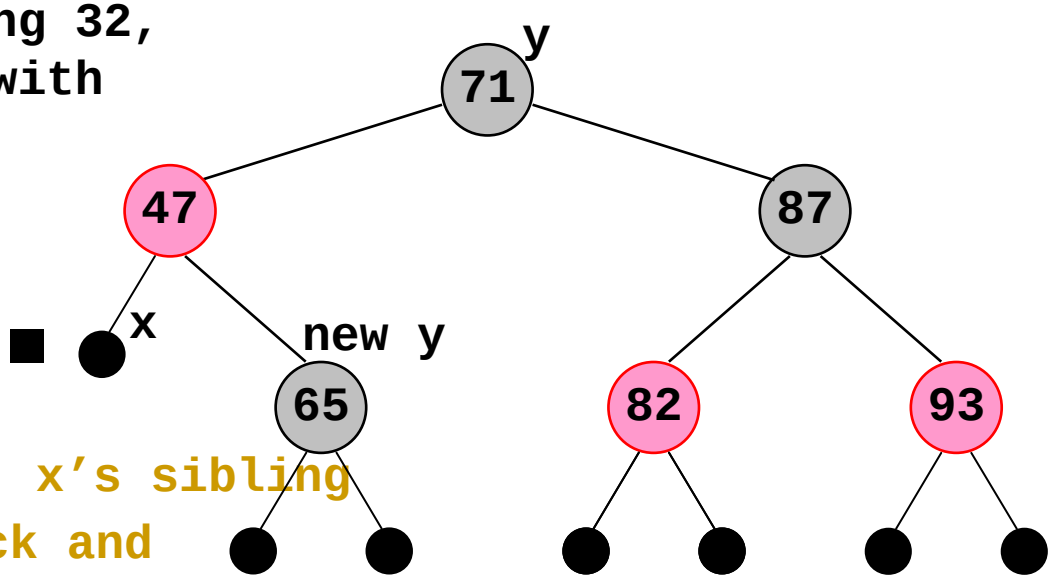
Identify y, x's sibling

Make y black and y's parent red

Left rotate x's parent

Deletion Example 3

After deleting 32,
x is a node with
black token



Identify y, x's sibling

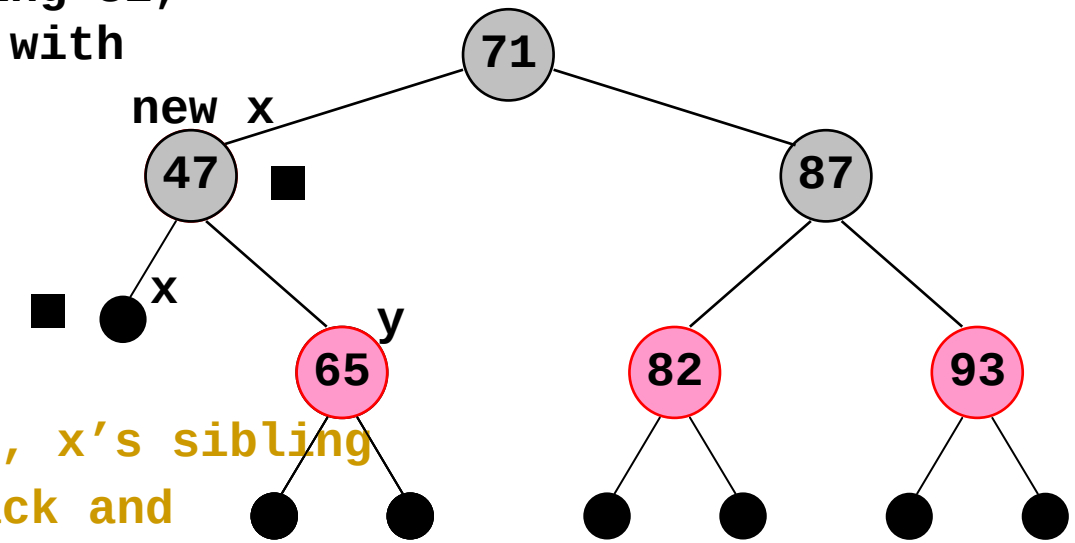
Make y black and
y's parent red

Left rotate x's parent

Identify y – x's new sibling

Deletion Example 3

After deleting 32,
x is a node with
black token



Identify y, x's sibling

Make y black and
y's parent red

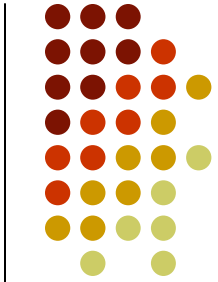
Left rotate x's parent

Identify y – x's new sibling

Color y red

Assign x it's parent, and color it black

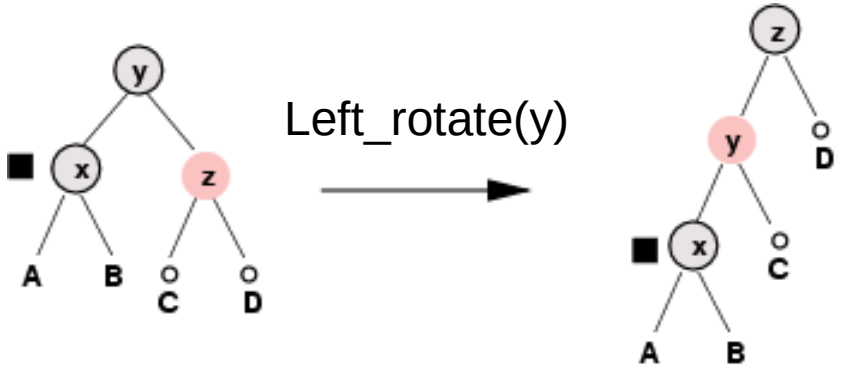
Tree Fix algorithm cases: case (1) x is red



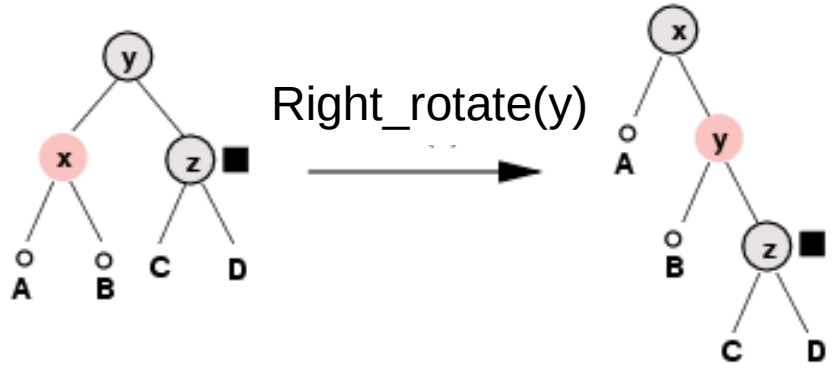
- The simplest case
- x has a black token and is colored red, so just color it black and remove token a we are done!

- In the remaining cases, assume x is black (and has the black token, i.e., it's double black)

Tree Fix algorithm cases: case (2) x's sibling is red



Colors of y and z were swapped



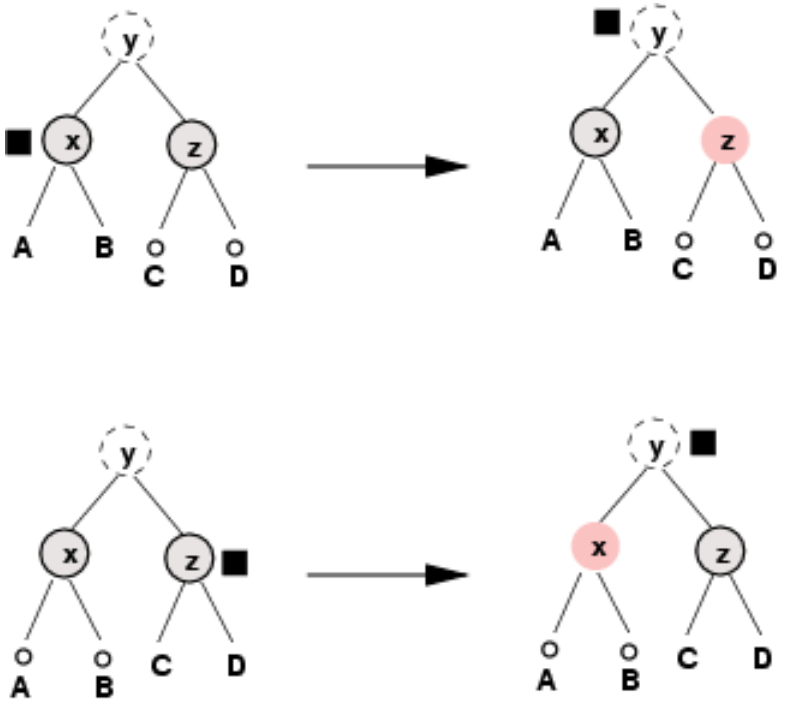
Colors of x and y were swapped

Remarks:

- the roots of subtrees C and D are black
- the second is the symmetric case, when x is the right child
- in the next step (case (3) or (4)) the algorithm will finish!

Tree Fix algorithm cases: case (3)

x's sibling is black and both nephews are black

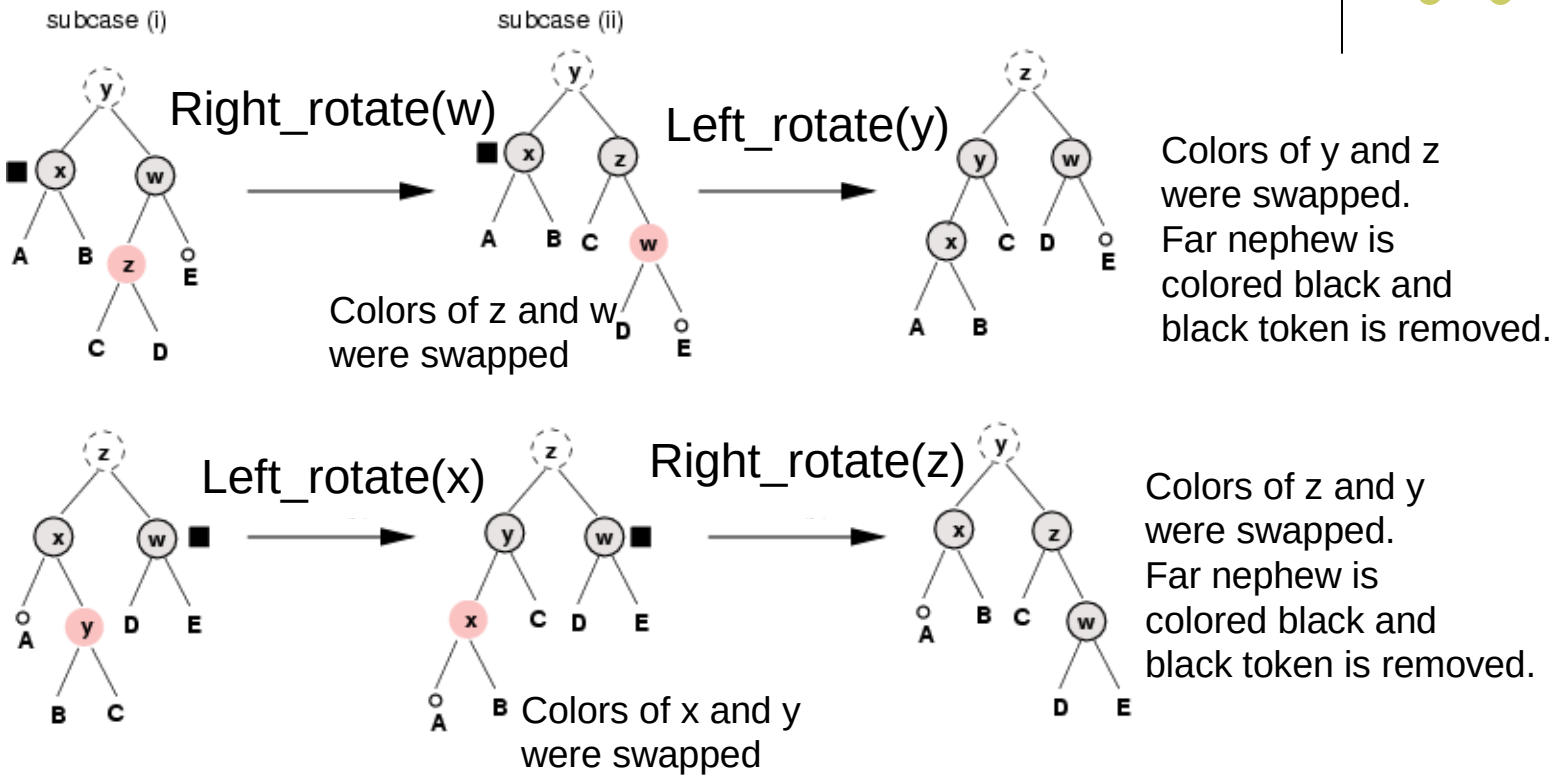


Remarks:

- nephews are roots of subtrees C and D
- the black token is passed one level up

Tree Fix algorithm cases: case (4)

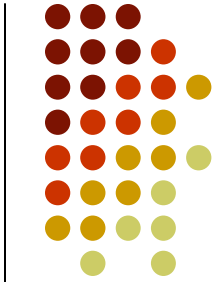
x's sibling is black and at least one nephew is red



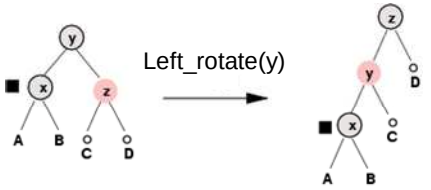
Remarks:

- in this case, the black token is removed completely
- if the "far" nephew is black (subcase (i)), rotate its parent, so that a new "far" nephew is red; otherwise start in subcase(ii)

Tree Fix Algorithm



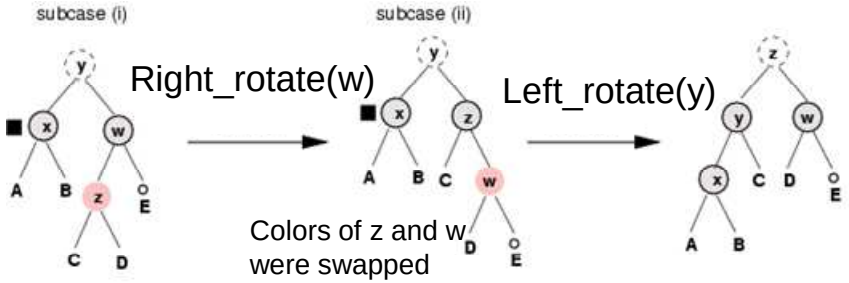
```
TreeNode<T> rbTreeFix(TreeNode<T> root,TreeNode<T> x)
//return new root; x is a node with the black token
{
  while (x != root && x.getColor() == black) // not case (1)
    if (x == x.getParent().getLeft()) { // x is left child
      y = x.getParent().getRight(); // y is x's sibling
      if (y.getColor() == red) { // case (2)
        y.setColor(black);
        x.getParent().setColor(red); // p was black
        root = left_rotate(root,x.getParent());
        y = x.getParent().getRight(); // new sibling
      }
      if (y.getLeft().getColor() == black &&
          y.getRight().getColor() == black) {
        // nephews are black - case (3)
        y.setColor(red);
        x = x.getParent();
      } else { // case (4)
        // ..........
      }
    } else {
      … // x is right child - symmetric
    }
  // end while loop
  x.setColor(black);
  return root;
}
```



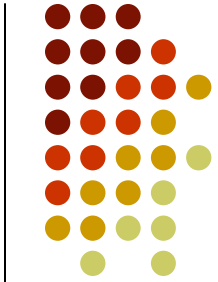
Colors of y and z were swapped

Tree Fix Algorithm (continued)

```
} else { // case (4)
  if (y.getRight().getColor() == black) {
    // subcase (i)
    y.getLeft().setColor(black);
    y.setColor(red);
    root = right_rotate(root,y);
    y = x.getParent().getRight();
  }
  // subcase (ii)
  y.setColor(x.getParent().getColor());
  x.getParent().setColor(black);
  y.getRight().setColor(black);
  root = left_rotate(root, x.getParent());
  x = root; // we can finish
}
```



RB Trees efficiency



- All operations work in time O(height)
- and we have proved that heigh is O(log n)

- hence, all operations work in time O(log n)! – much more efficient than linked list or arrays implementation of sorted list!

Sorted List	Search	Insertion	Deletion
with arrays	O(log n)	O(n)	O(n)
with linked list	O(n)	O(n)	O(n)
with RB trees	O(log n)	O(log n)	O(log n)