

# Algorithms and Analysis

## Lesson 8: *Keep Trees Balanced*



*AVL trees, red-black trees, TreeSet, TreeMap*

# Outline

1. **Deletion**
2. Balancing Trees
  - Rotations
3. AVL
4. Red-Black Trees
  - TreeSet
  - TreeMap



# Recap

- Binary search trees are commonly used to store data because we need to only look down one branch to find any element
- We saw how to implement many methods of the binary search tree
  - ★ `find`
  - ★ `insert`
  - ★ `successor` (in outline)
- One method we missed was `remove`

# Recap

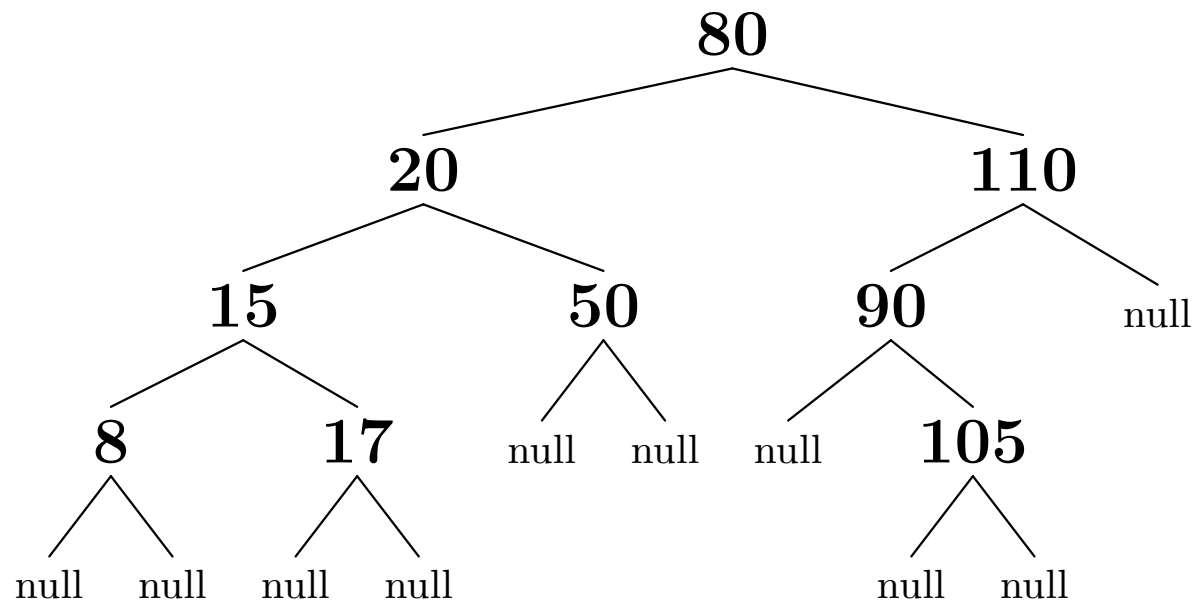
- Binary search trees are commonly used to store data because we need to only look down one branch to find any element
- We saw how to implement many methods of the binary search tree
  - ★ `find`
  - ★ `insert`
  - ★ `successor` (in outline)
- One method we missed was `remove`

# Recap

- Binary search trees are commonly used to store data because we need to only look down one branch to find any element
- We saw how to implement many methods of the binary search tree
  - ★ `find`
  - ★ `insert`
  - ★ `successor` (in outline)
- One method we missed was `remove`

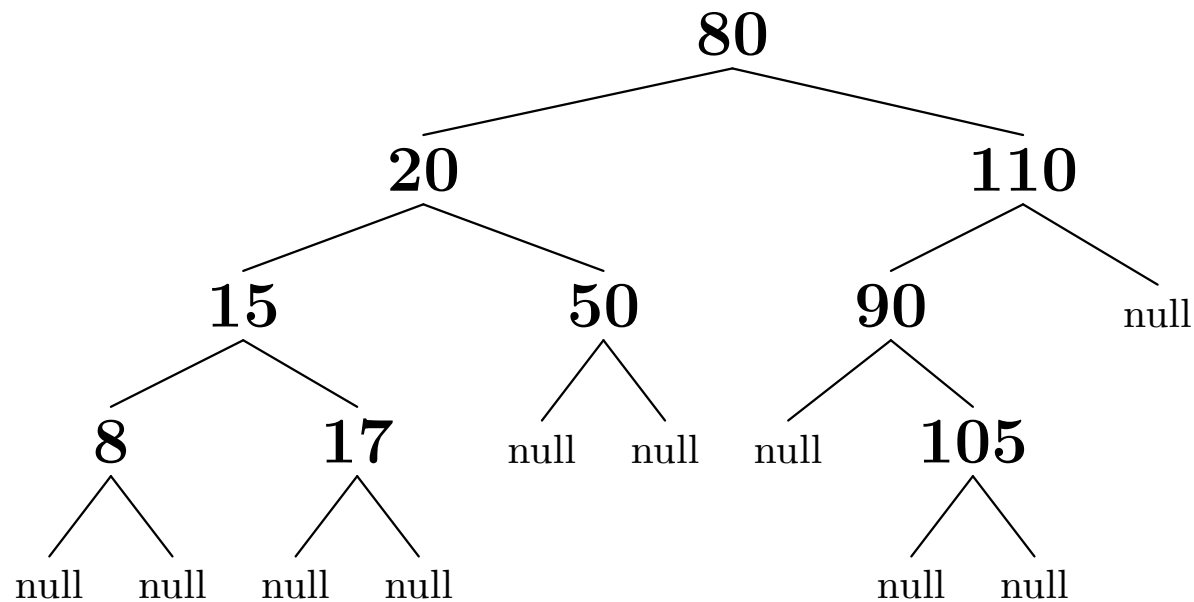
# Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)



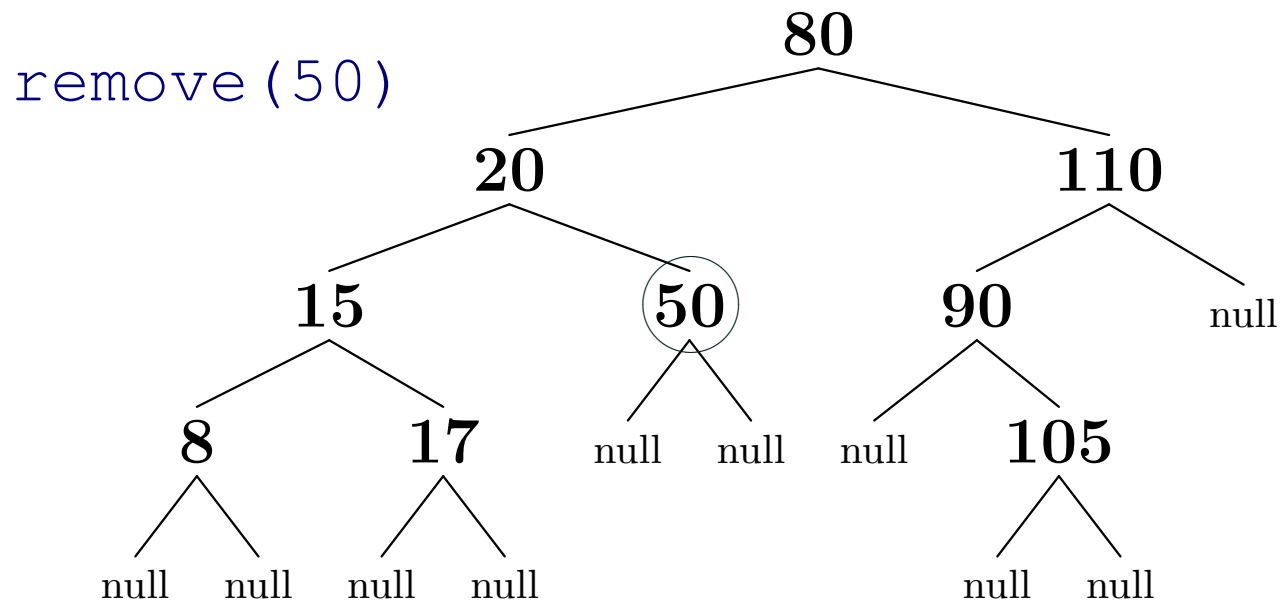
# Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)



# Deletion

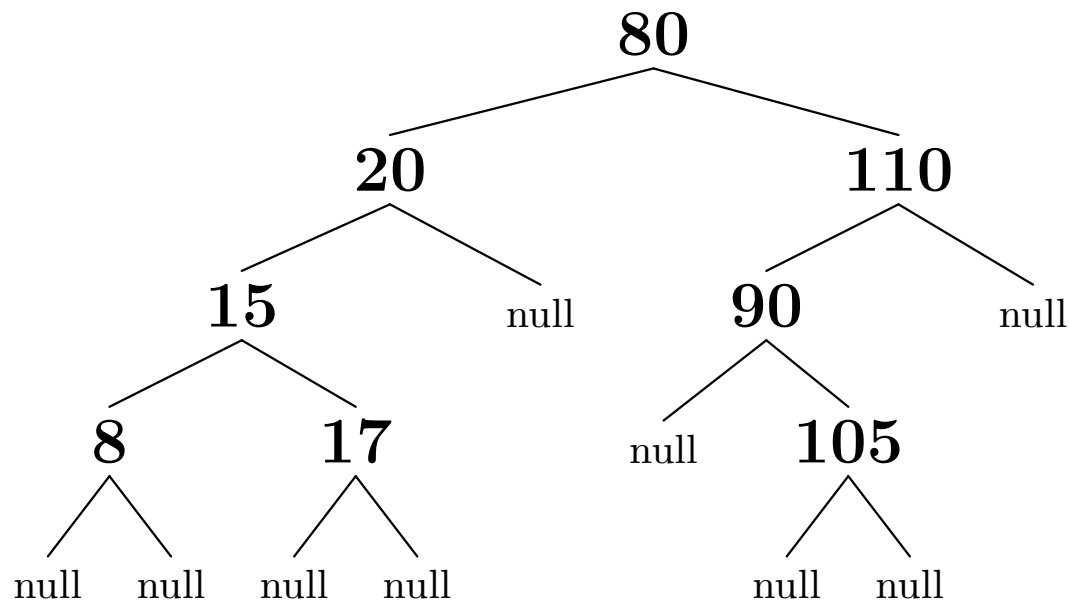
- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)





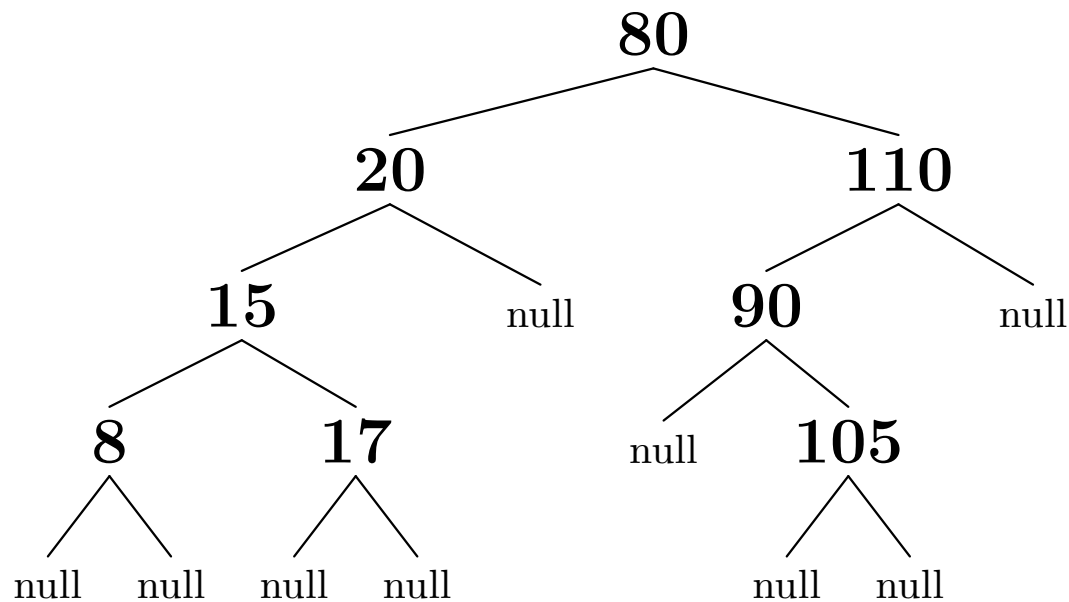
# Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)



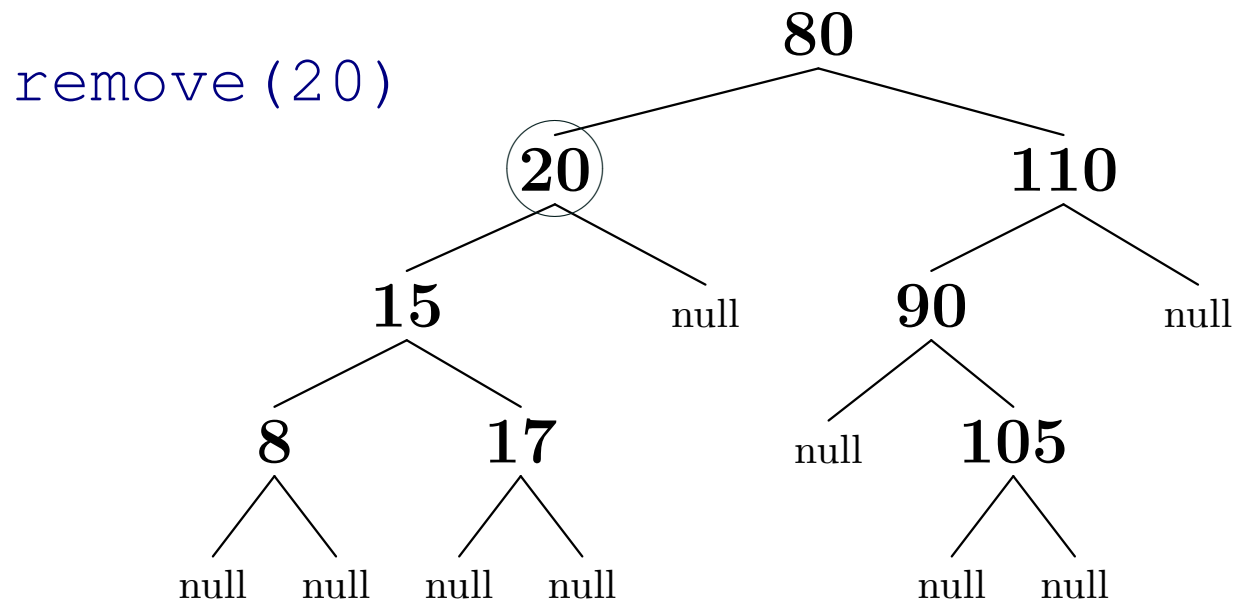
# Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)



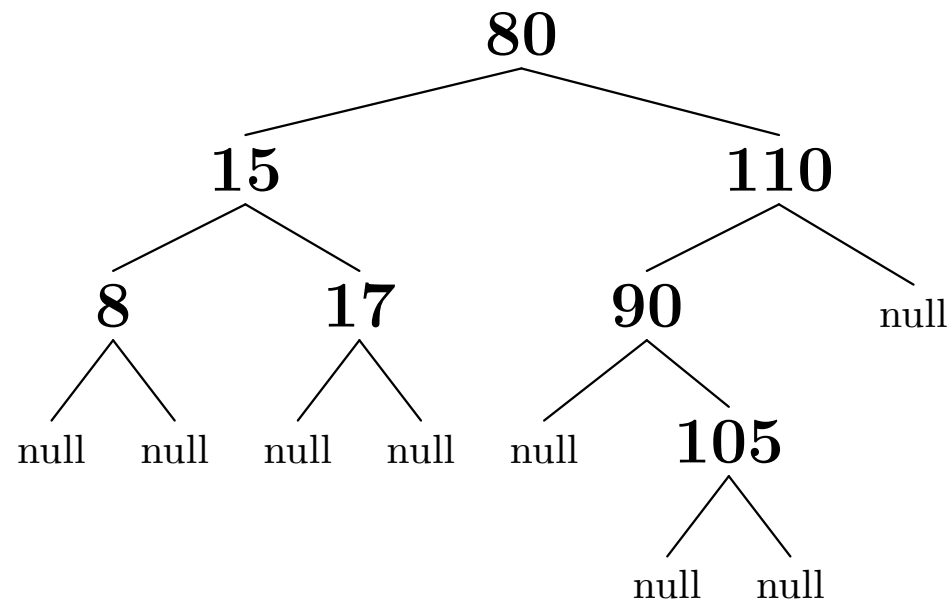
# Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)



# Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)

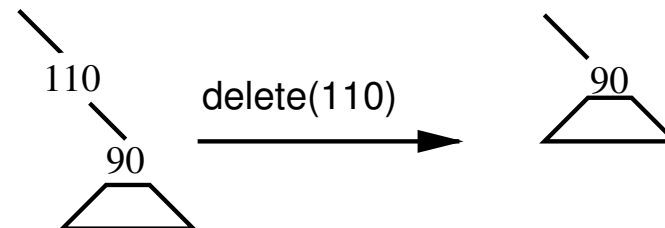
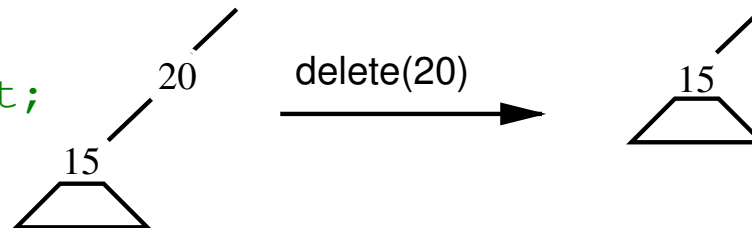
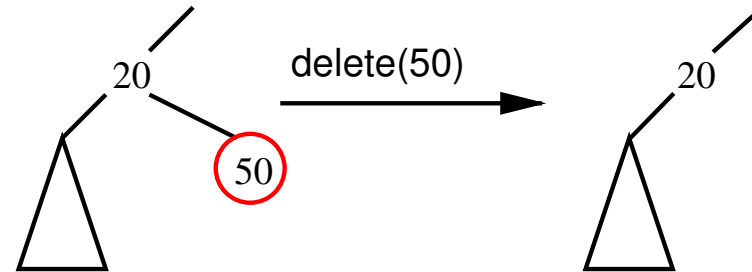


# Code to remove Node $n$

```

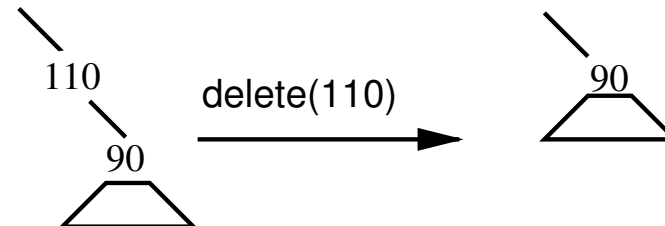
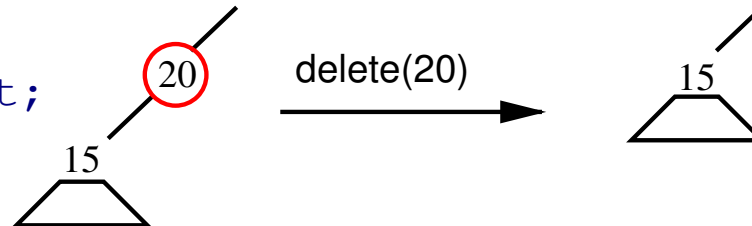
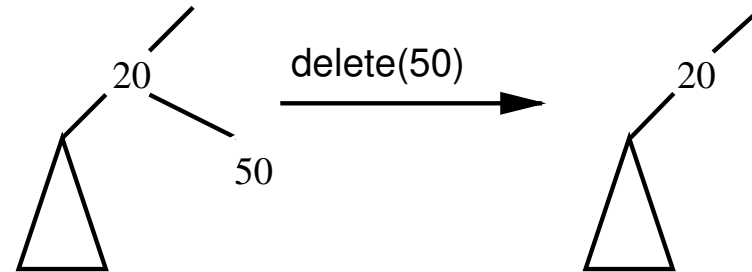
if (n->left==0 && n->right==0) {
    if (n == n->parent->left)
        n->parent->left = 0;
    else
        n->parent->right = 0;
} else if (n->right==0) {
    if (n == n->parent->left)
        n->parent->left = n->left;
    else
        n->parent->right = n->left;
    n->left->parent = n->parent;
} else if (n->left==0) {
    if (n == n->parent->left)
        n->parent->left = n->right;
    else
        n->parent->right = n->right;
    n->right->parent = n->parent;
}
delete n;

```



# Code to remove Node $n$

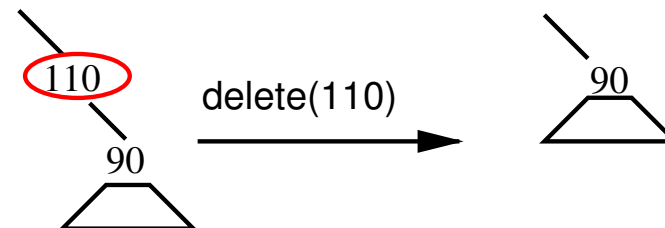
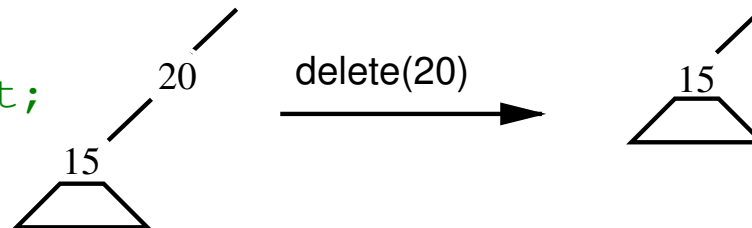
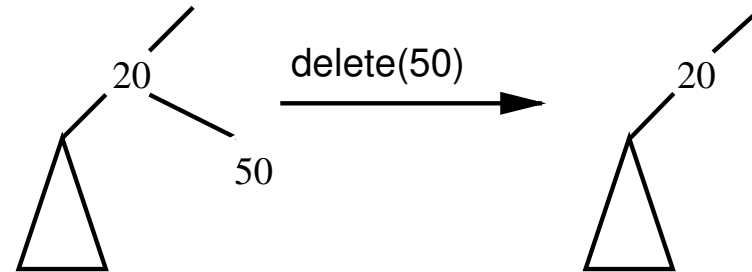
```
if (n->left==0 && n->right==0) {  
    if (n == n->parent->left)  
        n->parent->left = 0;  
    else  
        n->parent->right = 0;  
}  
else if (n->right==0) {  
    if (n == n->parent->left)  
        n->parent->left = n->left;  
    else  
        n->parent->right = n->left;  
    n->left->parent = n->parent;  
}  
else if (n->left==0) {  
    if (n == n->parent->left)  
        n->parent->left = n->right;  
    else  
        n->parent->right = n->right;  
    n->right->parent = n->parent;  
}  
delete n;
```



# Code to remove Node $n$

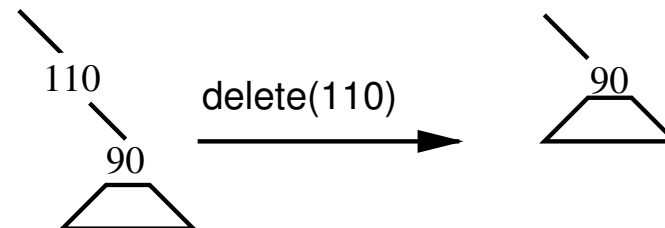
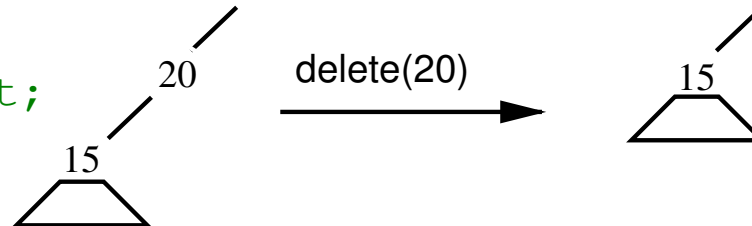
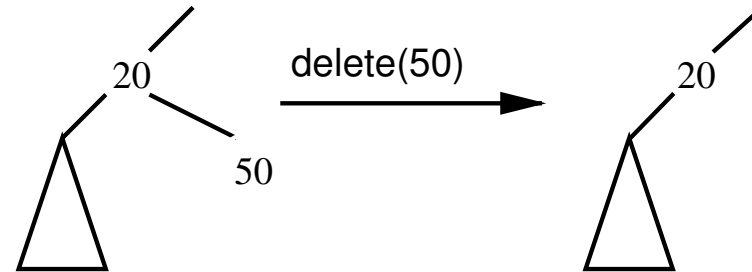
```

if (n->left==0 && n->right==0) {
    if (n == n->parent->left)
        n->parent->left = 0;
    else
        n->parent->right = 0;
} else if (n->right==0) {
    if (n == n->parent->left)
        n->parent->left = n->left;
    else
        n->parent->right = n->left;
    n->left->parent = n->parent;
} else if (n->left==0) {
    if (n == n->parent->left)
        n->parent->left = n->right;
    else
        n->parent->right = n->right;
    n->right->parent = n->parent;
}
delete n;
    
```



# Code to remove Node $n$

```
if (n->left==0 && n->right==0) {  
    if (n == n->parent->left)  
        n->parent->left = 0;  
    else  
        n->parent->right = 0;  
}  
else if (n->right==0) {  
    if (n == n->parent->left)  
        n->parent->left = n->left;  
    else  
        n->parent->right = n->left;  
    n->left->parent = n->parent;  
}  
else if (n->left==0) {  
    if (n == n->parent->left)  
        n->parent->left = n->right;  
    else  
        n->parent->right = n->right;  
    n->right->parent = n->parent;  
}  
delete n;
```





# Removing Element with Two Children

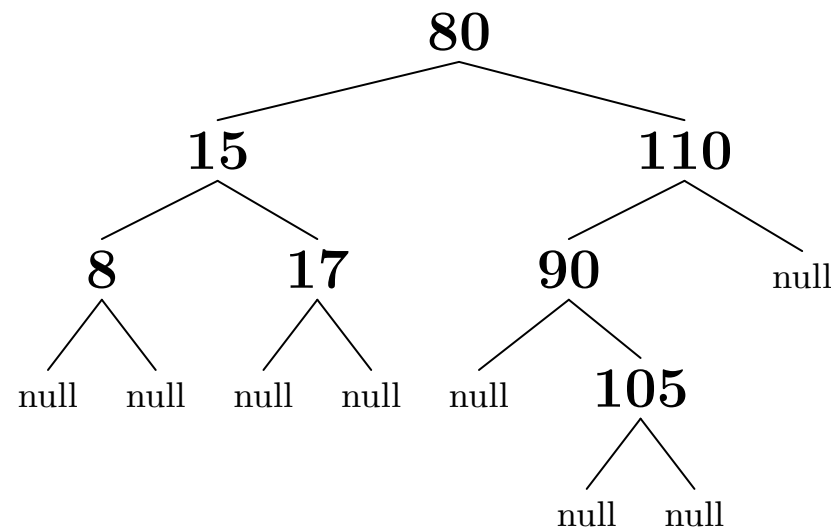
- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure

# Removing Element with Two Children

- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure

# Removing Element with Two Children

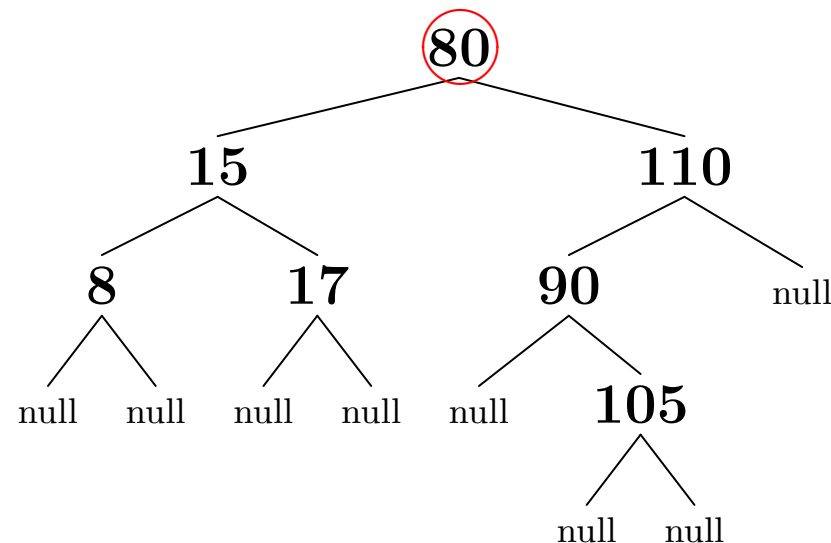
- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure



# Removing Element with Two Children

- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure

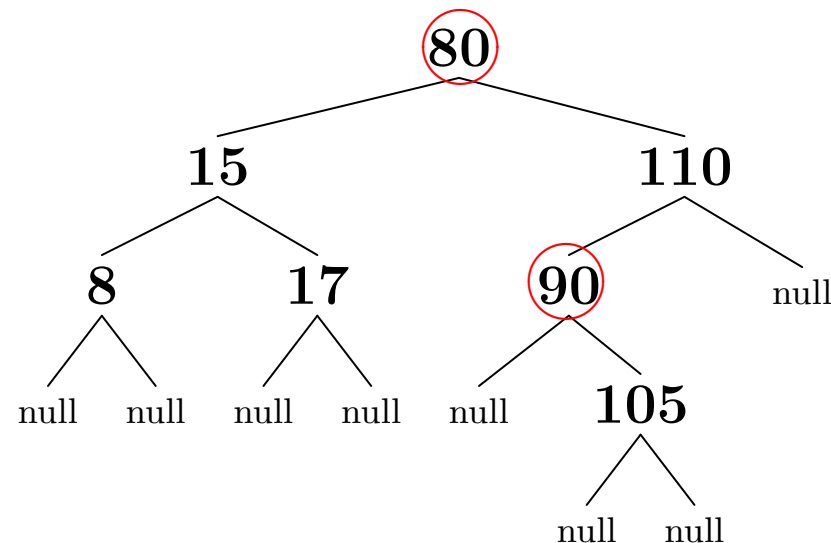
`remove(80)`



# Removing Element with Two Children

- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure

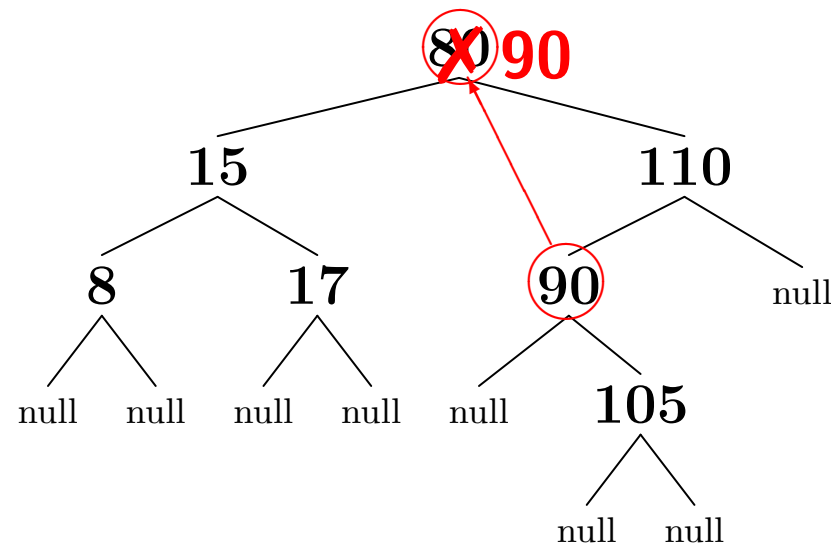
`remove(80)`



# Removing Element with Two Children

- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure

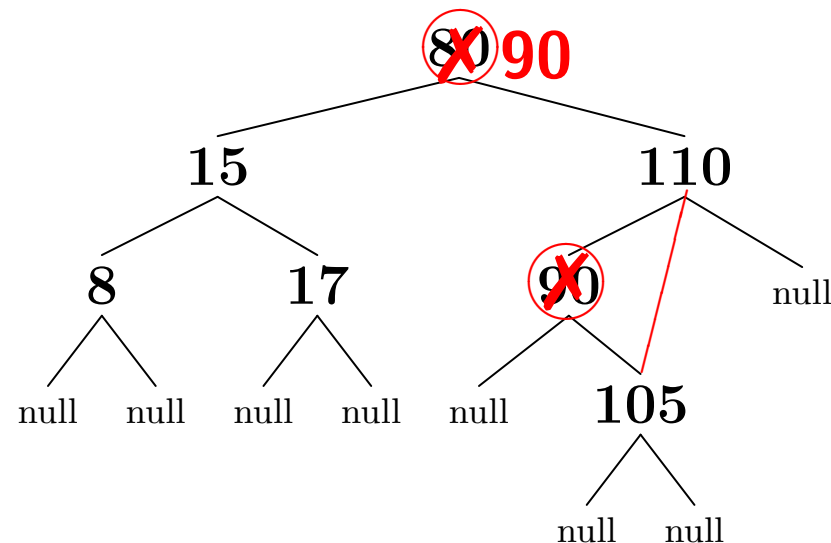
`remove(80)`



# Removing Element with Two Children

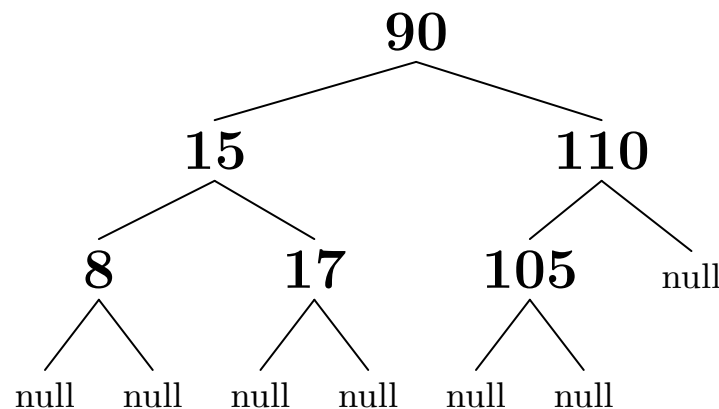
- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure

`remove(80)`



# Removing Element with Two Children

- If an element has two children then
  - ★ replace that element by its successor
  - ★ and then remove the successor using the above procedure





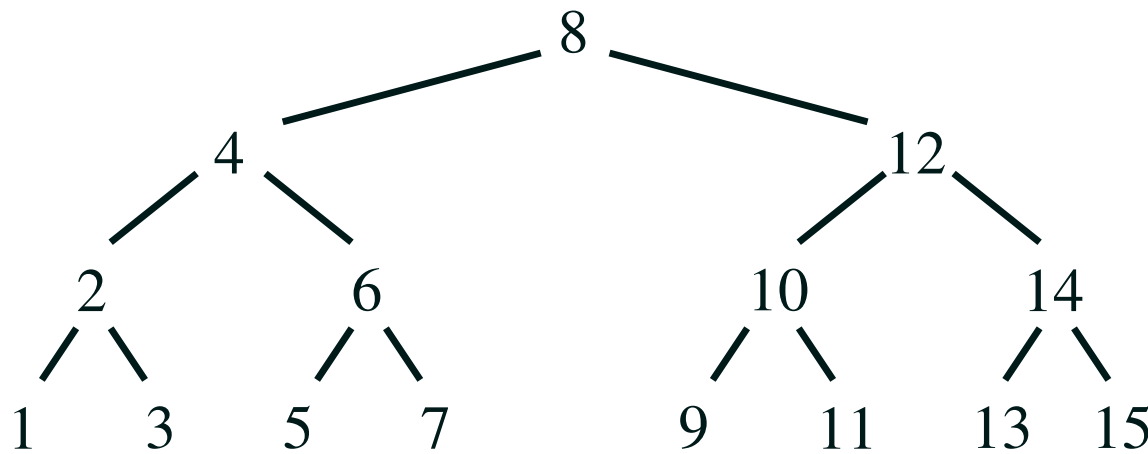
# Outline

1. Deletion
2. **Balancing Trees**
  - Rotations
3. AVL
4. Red-Black Trees
  - TreeSet
  - TreeMap

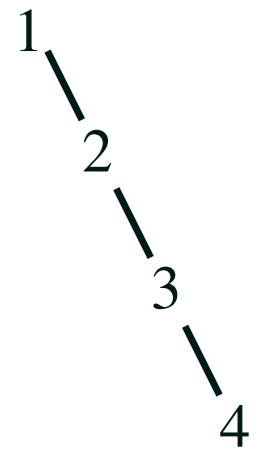


# Why Balance Trees

- The number of comparisons to access an element depends on the depth of the node
- The average depth of the node depends on the shape of the tree



full tree

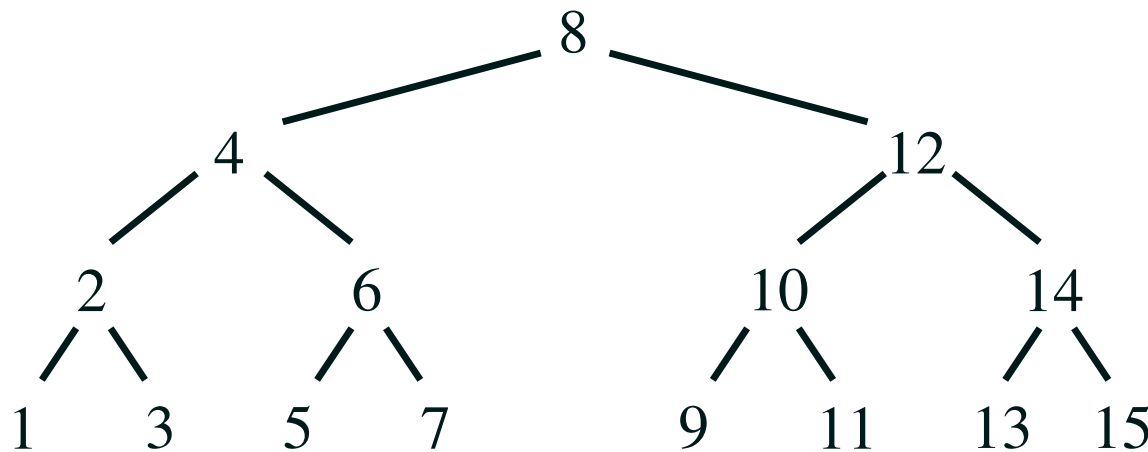


sparse tree

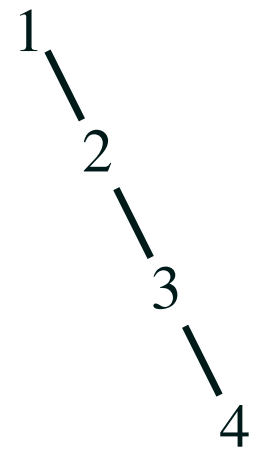
- The shape of the tree depends on the order the elements are added

# Why Balance Trees

- The number of comparisons to access an element depends on the depth of the node
- The average depth of the node depends on the shape of the tree



full tree

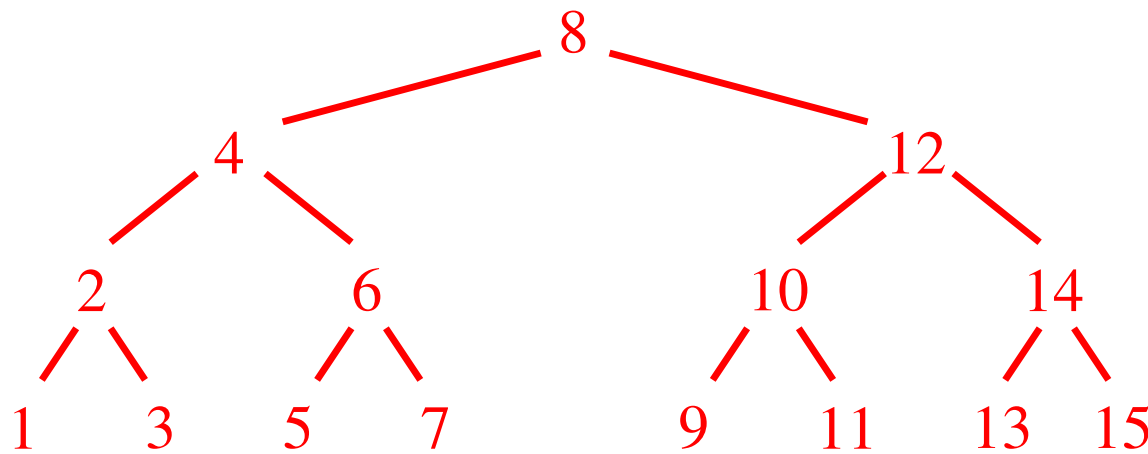


sparse tree

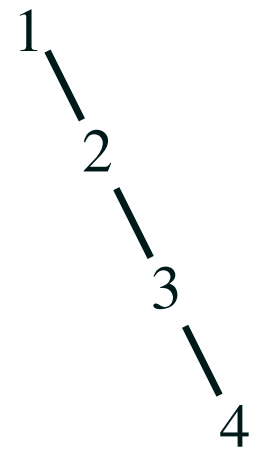
- The shape of the tree depends on the order the elements are added

# Why Balance Trees

- The number of comparisons to access an element depends on the depth of the node
- The average depth of the node depends on the shape of the tree



full tree

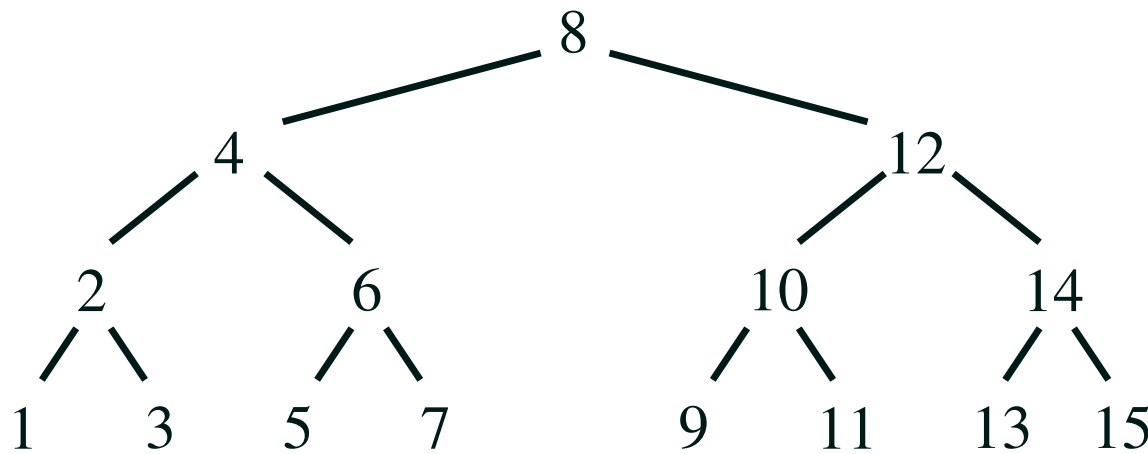


sparse tree

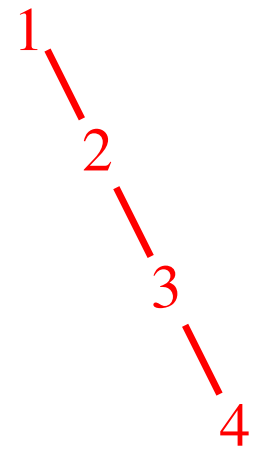
- The shape of the tree depends on the order the elements are added

# Why Balance Trees

- The number of comparisons to access an element depends on the depth of the node
- The average depth of the node depends on the shape of the tree



full tree

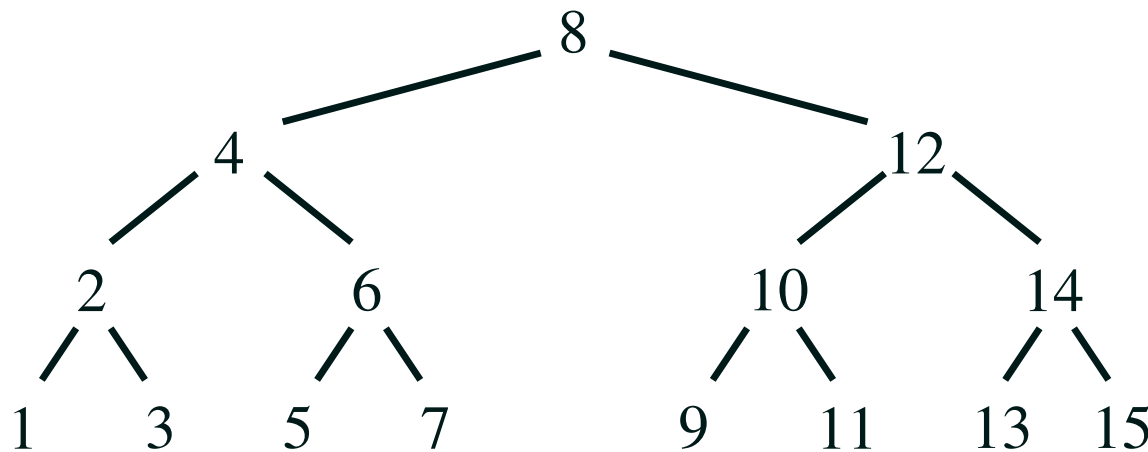


sparse tree

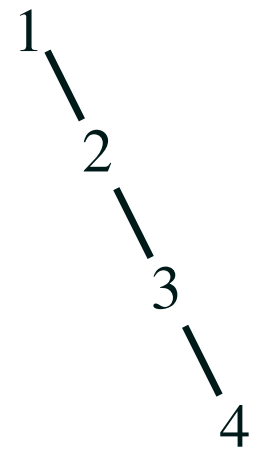
- The shape of the tree depends on the order the elements are added

# Why Balance Trees

- The number of comparisons to access an element depends on the depth of the node
- The average depth of the node depends on the shape of the tree



full tree



sparse tree

- The shape of the tree depends on the order the elements are added

# Time Complexity

- In the best situation (a full tree) the number of elements in a tree is  $n = \Theta(2^l)$  the depth is  $l$  so that the maximum depth is  $\log_2(n)$
- It turns out for random sequences the average depth is  $\Theta(\log(n))$
- In the worst case (when the tree is effectively a linked list), the average depth is  $\Theta(n)$
- Unfortunately, the worst case happens when the elements are added *in order* (not a rare event)

# Time Complexity

- In the best situation (a full tree) the number of elements in a tree is  $n = \Theta(2^l)$  the depth is  $l$  so that the maximum depth is  $\log_2(n)$
- It turns out for random sequences the average depth is  $\Theta(\log(n))$
- In the worst case (when the tree is effectively a linked list), the average depth is  $\Theta(n)$
- Unfortunately, the worst case happens when the elements are added *in order* (not a rare event)



# Time Complexity

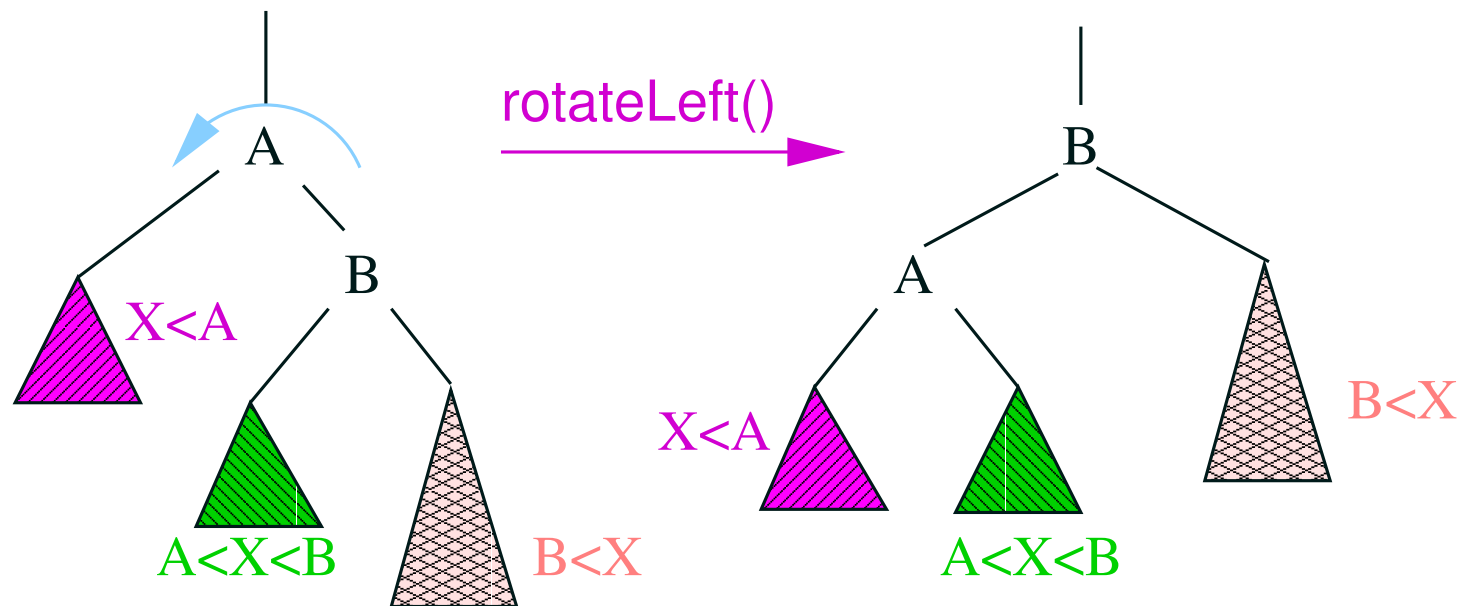
- In the best situation (a full tree) the number of elements in a tree is  $n = \Theta(2^l)$  the depth is  $l$  so that the maximum depth is  $\log_2(n)$
- It turns out for random sequences the average depth is  $\Theta(\log(n))$
- In the worst case (when the tree is effectively a linked list), the average depth is  $\Theta(n)$
- Unfortunately, the worst case happens when the elements are added *in order* (not a rare event)

# Time Complexity

- In the best situation (a full tree) the number of elements in a tree is  $n = \Theta(2^l)$  the depth is  $l$  so that the maximum depth is  $\log_2(n)$
- It turns out for random sequences the average depth is  $\Theta(\log(n))$
- In the worst case (when the tree is effectively a linked list), the average depth is  $\Theta(n)$
- Unfortunately, the worst case happens when the elements are added *in order* (not a rare event)

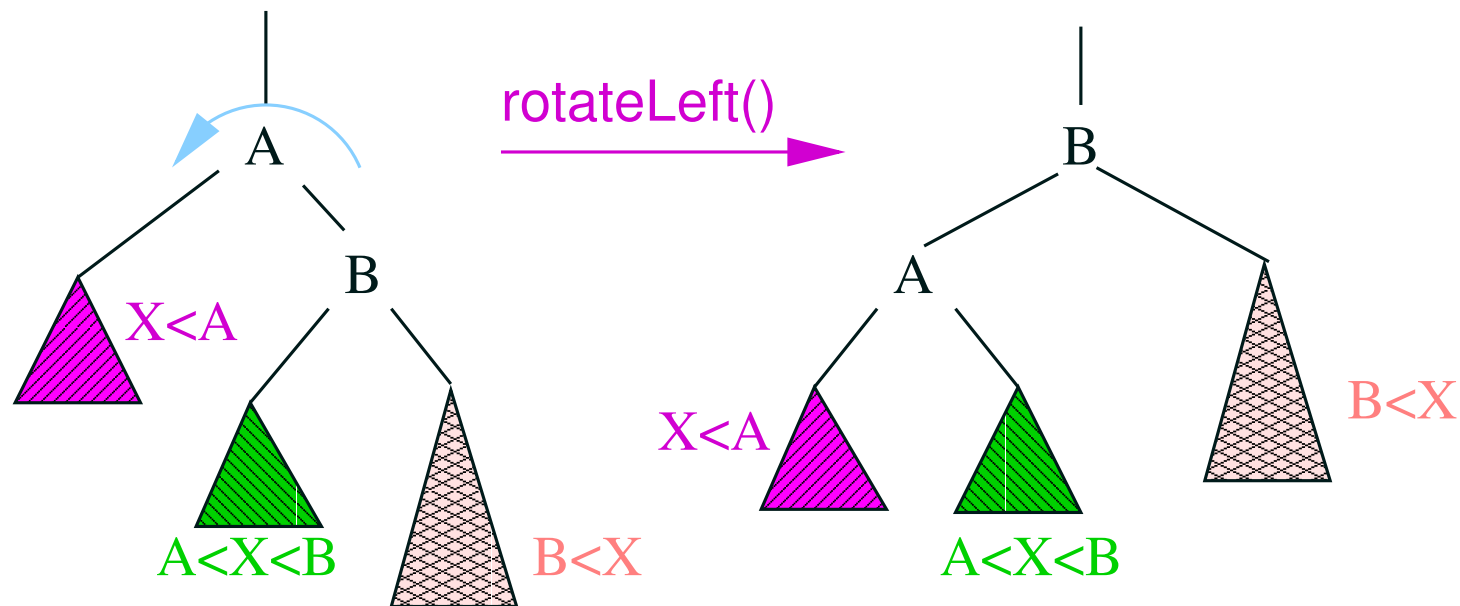
# Rotations

- To avoid unbalanced trees we would like to modify the shape
- This is possible as the shape of the tree is not uniquely defined (e.g. we could make any node the root)
- We can change the shape of a tree using **rotations**
- E.g. left rotation



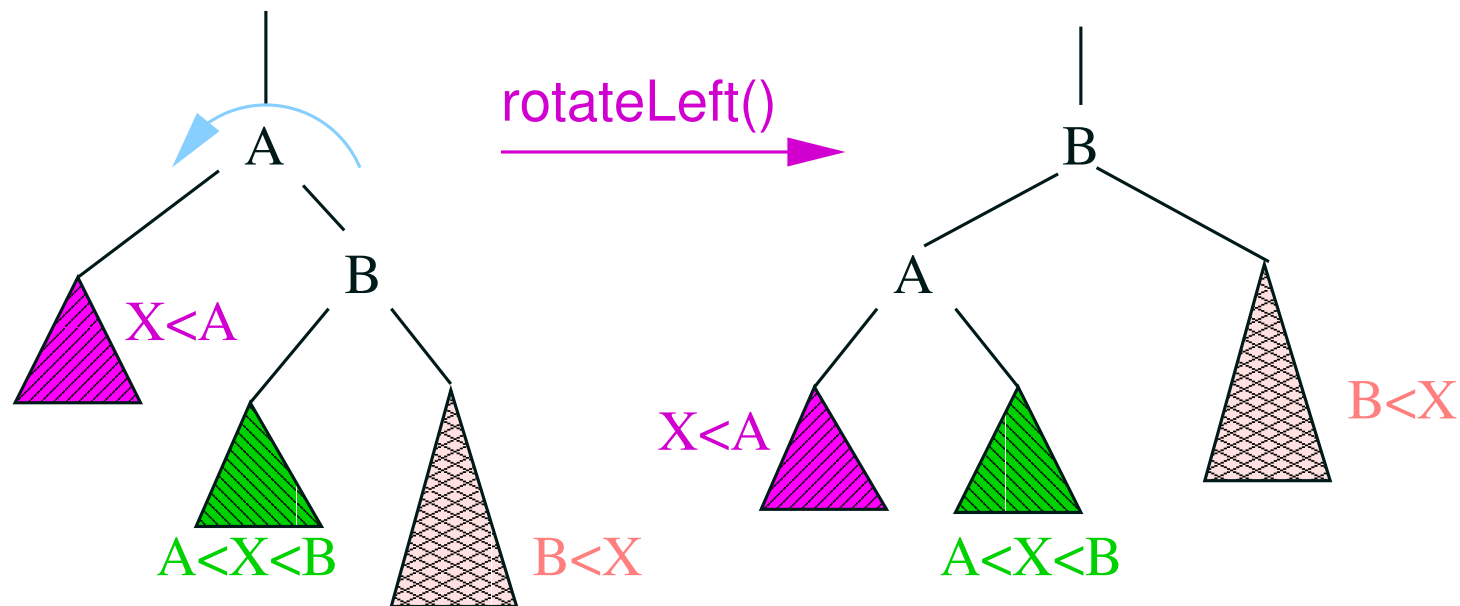
# Rotations

- To avoid unbalanced trees we would like to modify the shape
- This is possible as the shape of the tree is not uniquely defined (e.g. we could make any node the root)
- We can change the shape of a tree using **rotations**
- E.g. left rotation



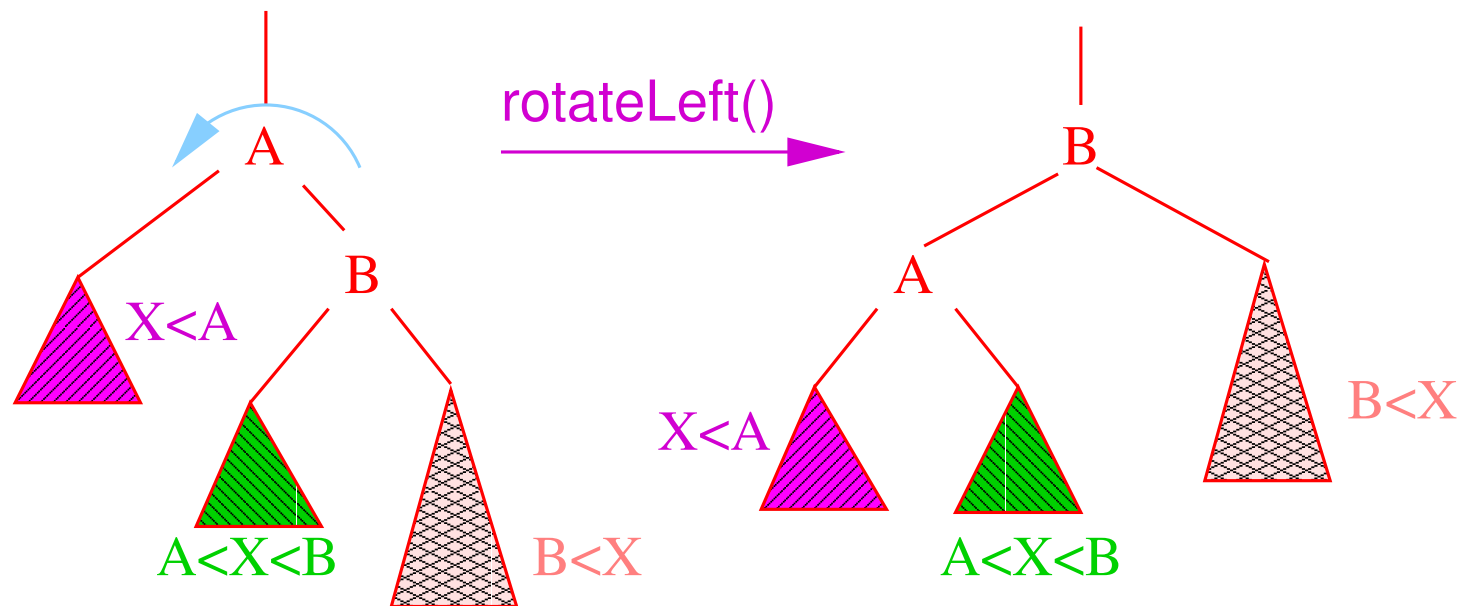
# Rotations

- To avoid unbalanced trees we would like to modify the shape
- This is possible as the shape of the tree is not uniquely defined (e.g. we could make any node the root)
- We can change the shape of a tree using **rotations**
- E.g. left rotation



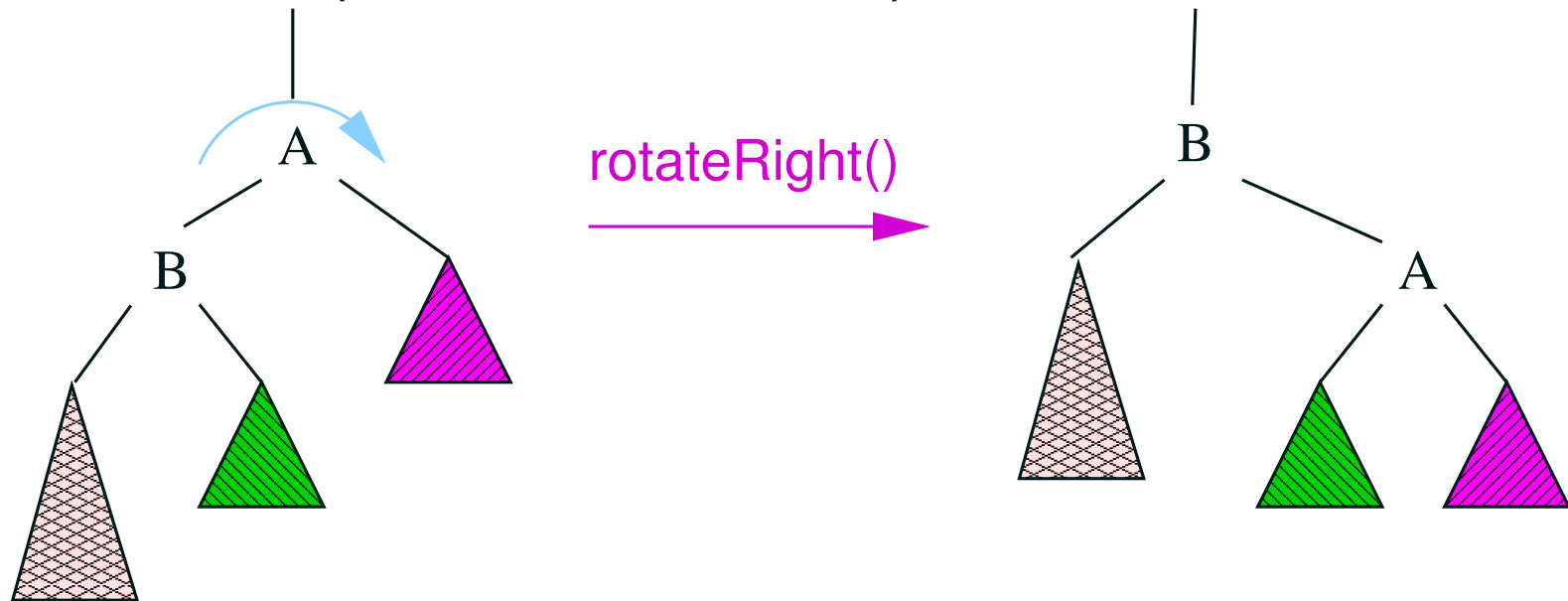
# Rotations

- To avoid unbalanced trees we would like to modify the shape
- This is possible as the shape of the tree is not uniquely defined (e.g. we could make any node the root)
- We can change the shape of a tree using **rotations**
- E.g. left rotation



# Types of Rotations

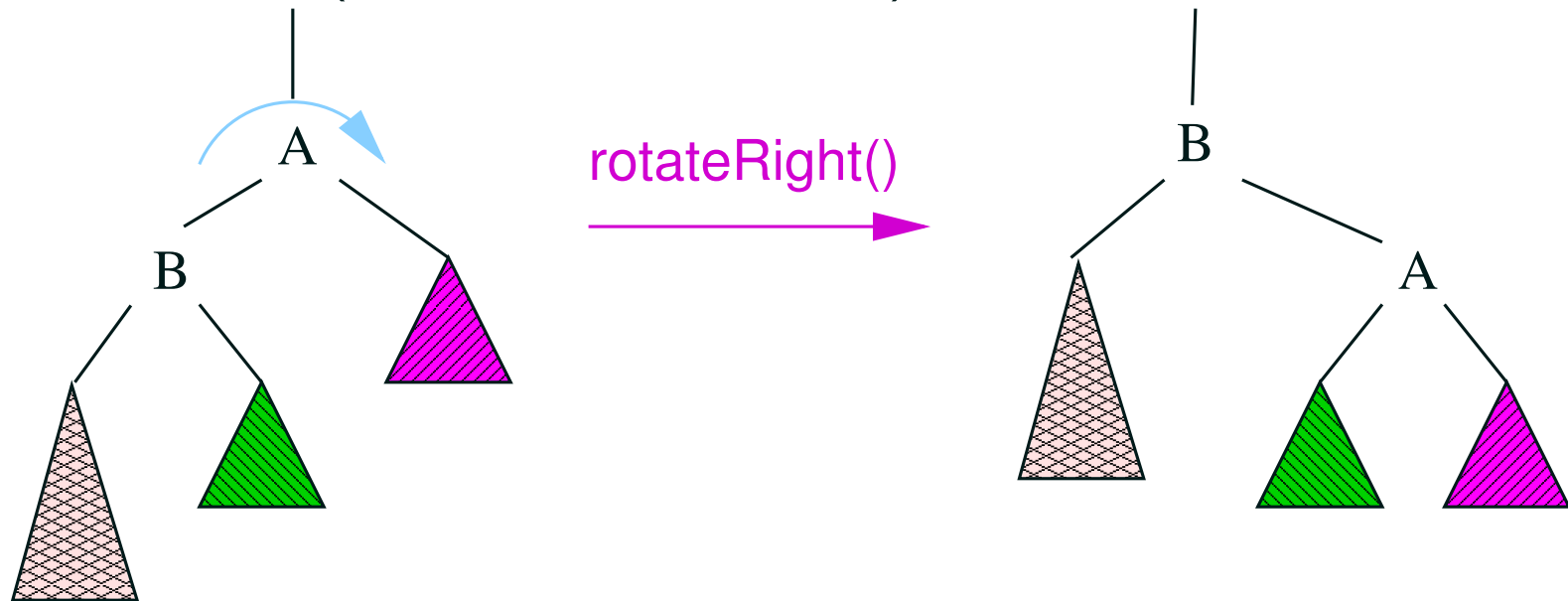
- We can get by with 4 types of rotations
  - ★ Left rotation (as above)
  - ★ Right rotation (symmetric to above)



- ★ Left-right double rotation
- ★ Right-left double rotation

# Types of Rotations

- We can get by with 4 types of rotations
  - ★ Left rotation (as above)
  - ★ Right rotation (symmetric to above)

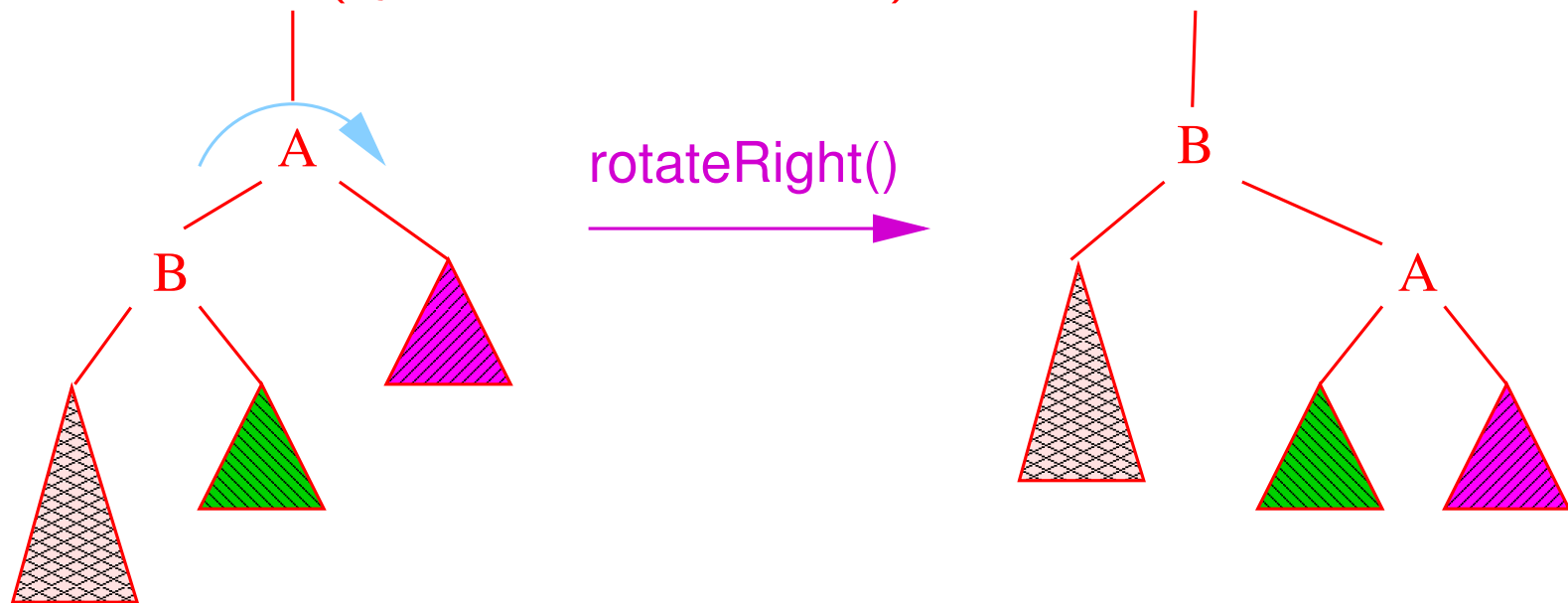


- ★ Left-right double rotation
- ★ Right-left double rotation



# Types of Rotations

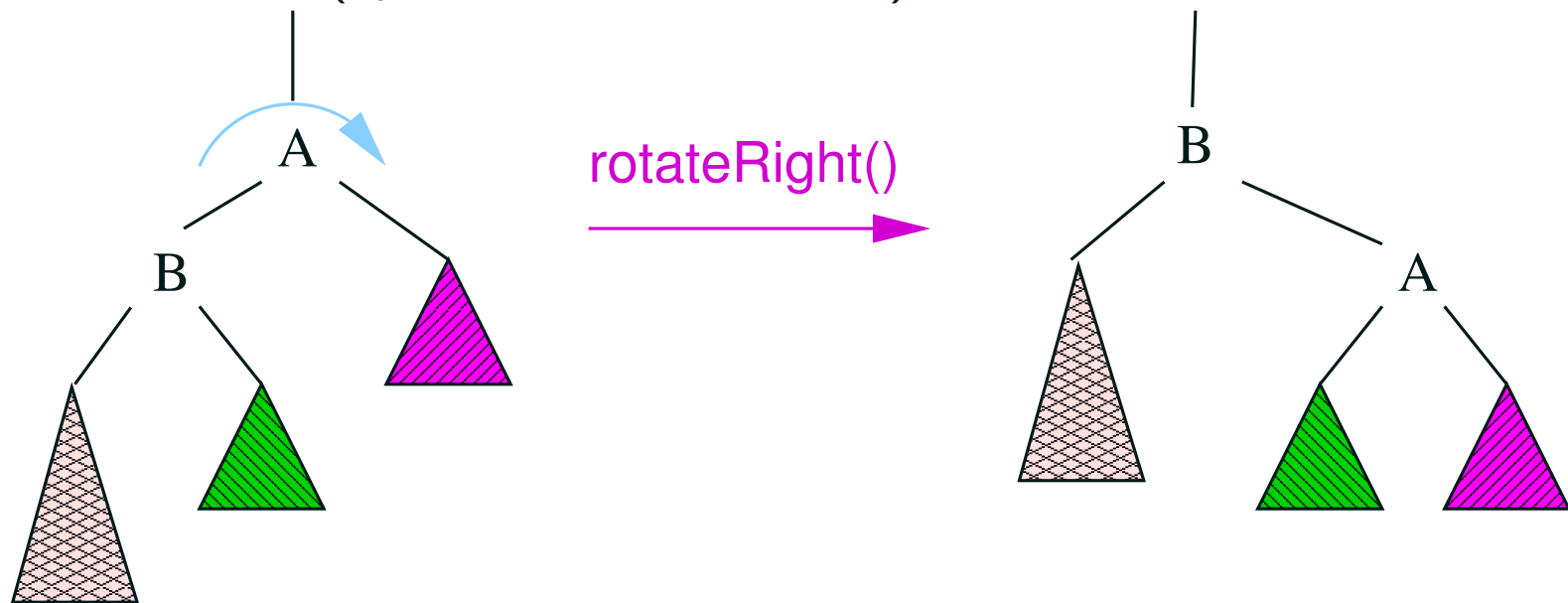
- We can get by with 4 types of rotations
  - ★ Left rotation (as above)
  - ★ Right rotation (symmetric to above)



- ★ Left-right double rotation
- ★ Right-left double rotation

# Types of Rotations

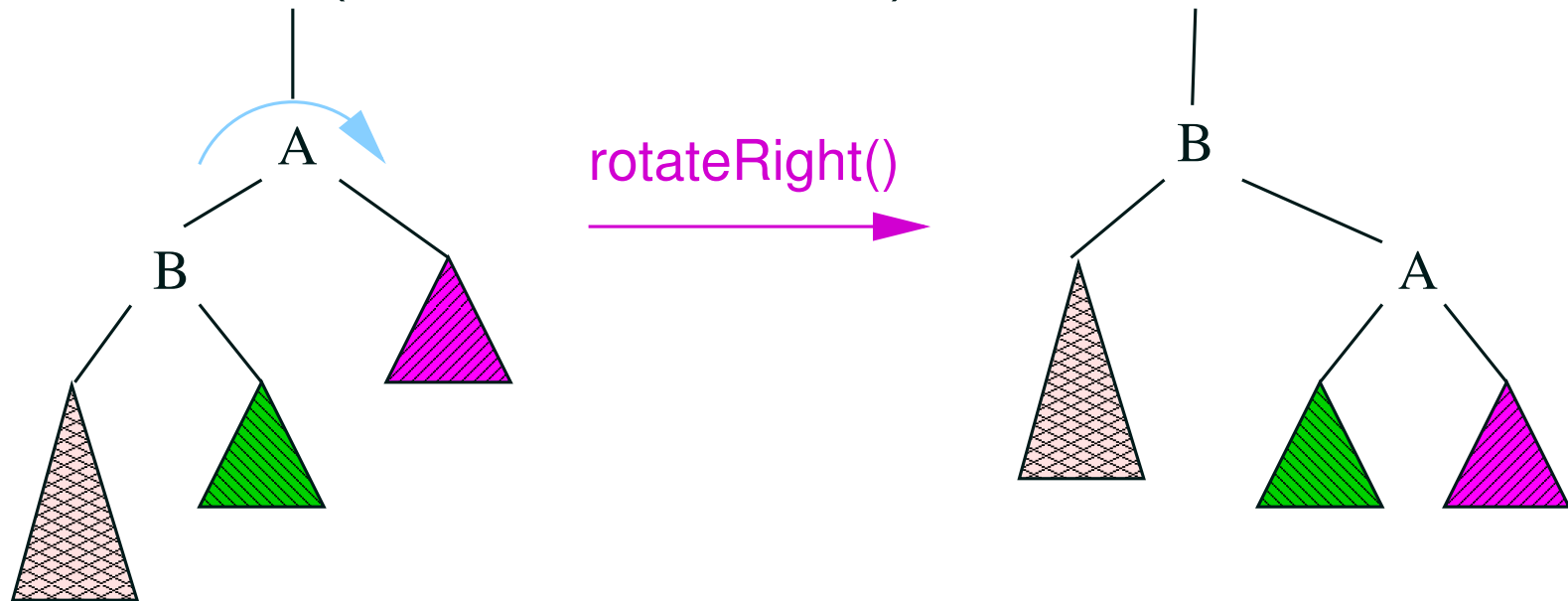
- We can get by with 4 types of rotations
  - ★ Left rotation (as above)
  - ★ Right rotation (symmetric to above)



- ★ Left-right double rotation
- ★ Right-left double rotation

# Types of Rotations

- We can get by with 4 types of rotations
  - ★ Left rotation (as above)
  - ★ Right rotation (symmetric to above)



- ★ Left-right double rotation
- ★ Right-left double rotation

# Coding Rotations

```
void rotateLeft(Node* e)
```

```
{
```

```
    Node* r = e->right;
```

```
    e->right = r->left;
```

```
    if (r->left != 0)
```

```
        r->left->parent = e;
```

```
    r->parent = e->parent;
```

```
    if (e->parent == 0)
```

```
        root = r;
```

```
    else if (e->parent->left == e)
```

```
        e->parent->left = r;
```

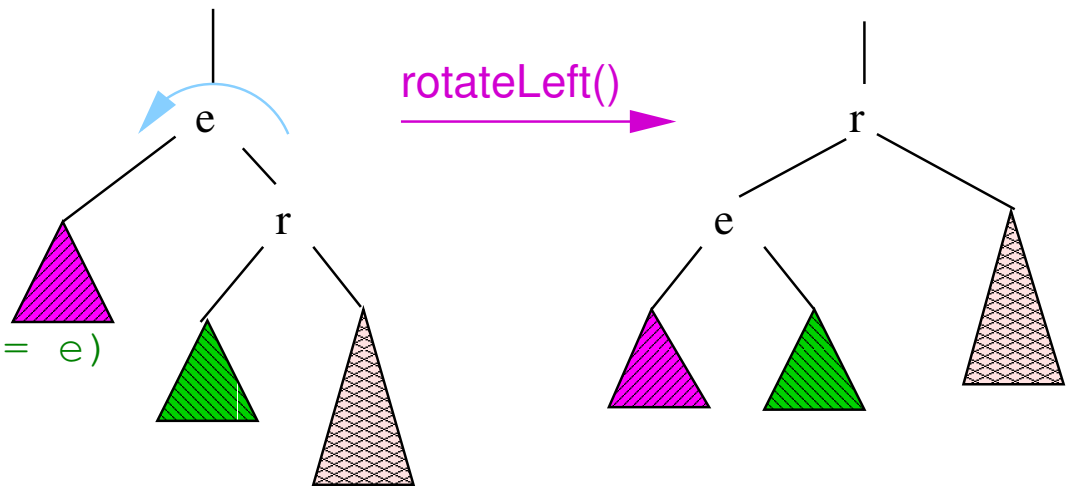
```
    else
```

```
        e->parent->right = r;
```

```
    r->left = e;
```

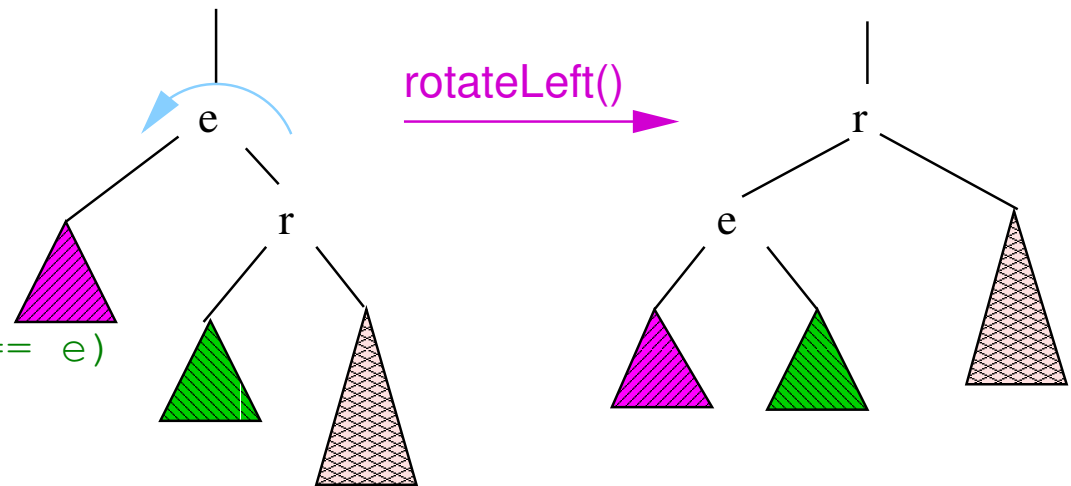
```
    e->parent = r;
```

```
}
```



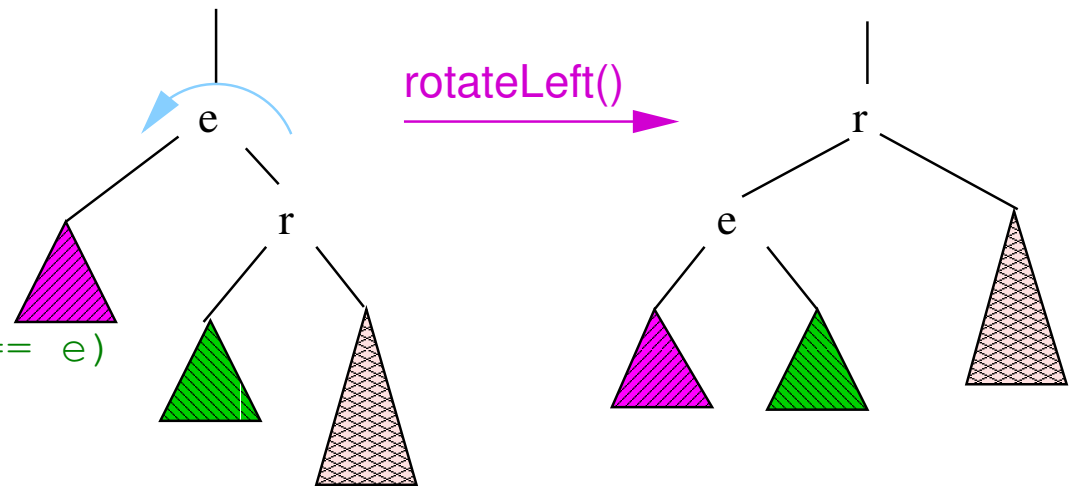
# Coding Rotations

```
void rotateLeft(Node* e)
{
    Node* r = e->right;
    e->right = r->left;
    if (r->left != 0)
        r->left->parent = e;
    r->parent = e->parent;
    if (e->parent == 0)
        root = r;
    else if (e->parent->left == e)
        e->parent->left = r;
    else
        e->parent->right = r;
    r->left = e;
    e->parent = r;
}
```



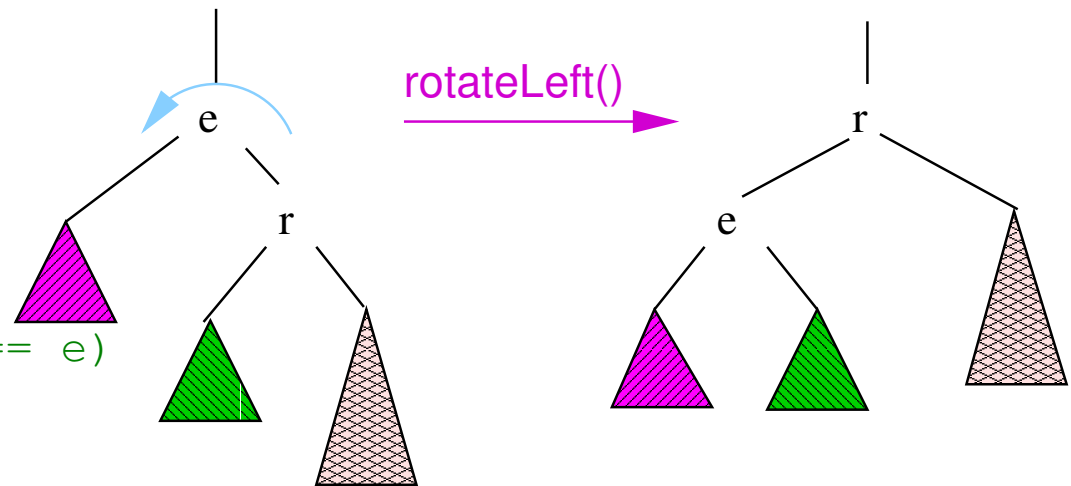
# Coding Rotations

```
void rotateLeft(Node* e)
{
    Node* r = e->right;
    e->right = r->left;
    if (r->left != 0)
        r->left->parent = e;
    r->parent = e->parent;
    if (e->parent == 0)
        root = r;
    else if (e->parent->left == e)
        e->parent->left = r;
    else
        e->parent->right = r;
    r->left = e;
    e->parent = r;
}
```



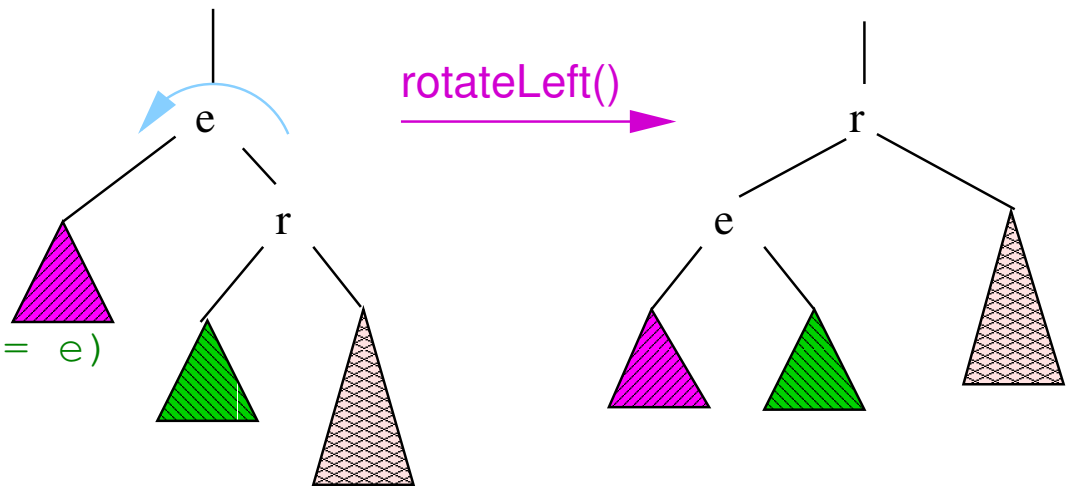
# Coding Rotations

```
void rotateLeft(Node* e)
{
    Node* r = e->right;
    e->right = r->left;
    if (r->left != 0)
        r->left->parent = e;
    r->parent = e->parent;
    if (e->parent == 0)
        root = r;
    else if (e->parent->left == e)
        e->parent->left = r;
    else
        e->parent->right = r;
    r->left = e;
    e->parent = r;
}
```



# Coding Rotations

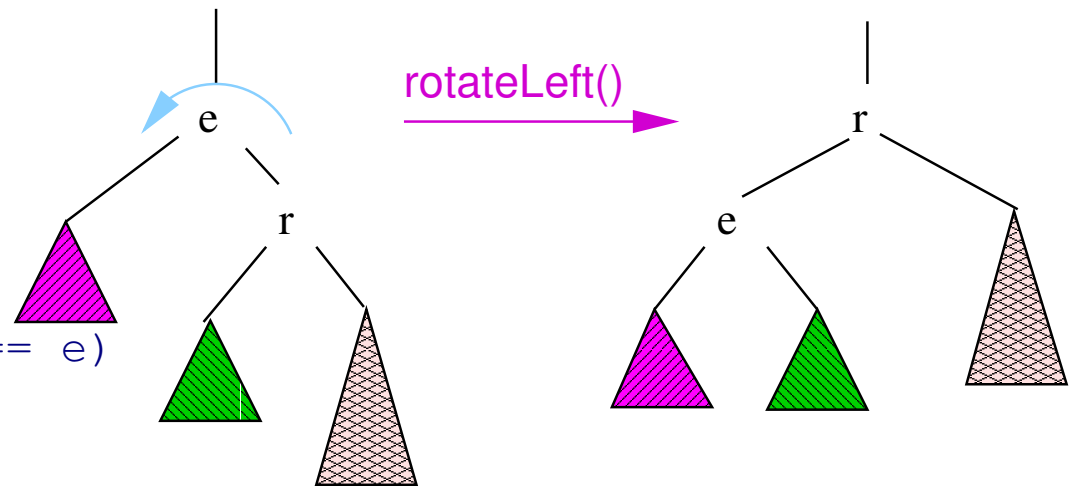
```
void rotateLeft(Node* e)
{
    Node* r = e->right;
    e->right = r->left;
    if (r->left != 0)
        r->left->parent = e;
    r->parent = e->parent;
    if (e->parent == 0)
        root = r;
    else if (e->parent->left == e)
        e->parent->left = r;
    else
        e->parent->right = r;
    r->left = e;
    e->parent = r;
}
```





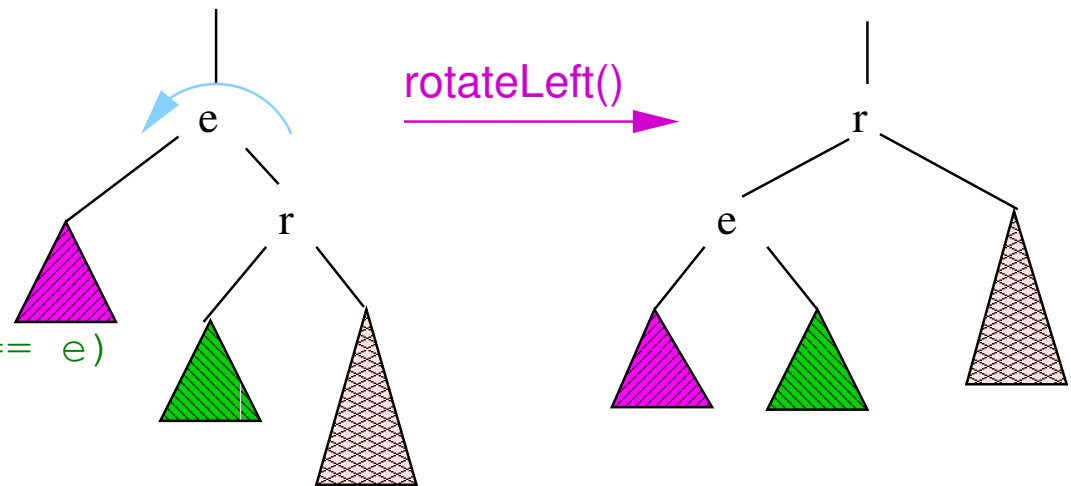
# Coding Rotations

```
void rotateLeft(Node* e)
{
    Node* r = e->right;
    e->right = r->left;
    if (r->left != 0)
        r->left->parent = e;
    r->parent = e->parent;
    if (e->parent == 0)
        root = r;
    else if (e->parent->left == e)
        e->parent->left = r;
    else
        e->parent->right = r;
    r->left = e;
    e->parent = r;
}
```



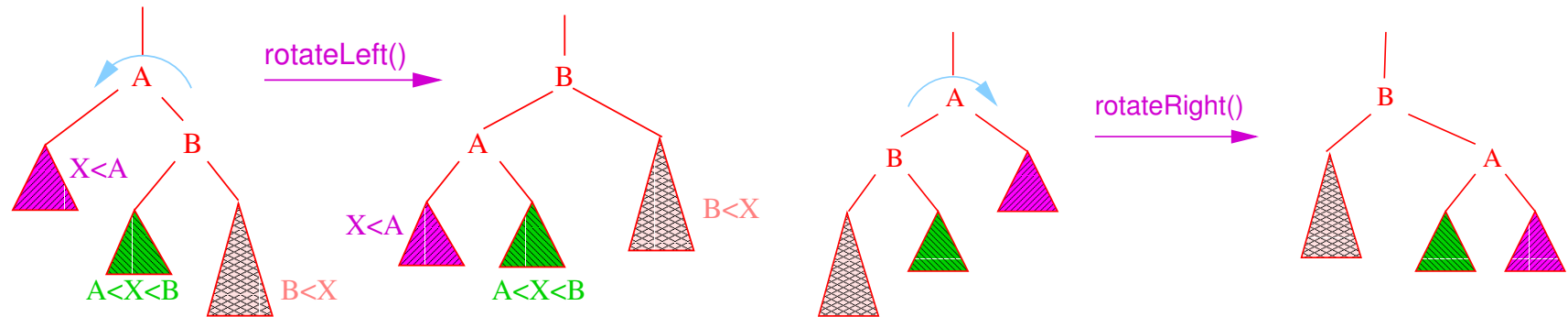
# Coding Rotations

```
void rotateLeft(Node* e)
{
    Node* r = e->right;
    e->right = r->left;
    if (r->left != 0)
        r->left->parent = e;
    r->parent = e->parent;
    if (e->parent == 0)
        root = r;
    else if (e->parent->left == e)
        e->parent->left = r;
    else
        e->parent->right = r;
    r->left = e;
    e->parent = r;
}
```



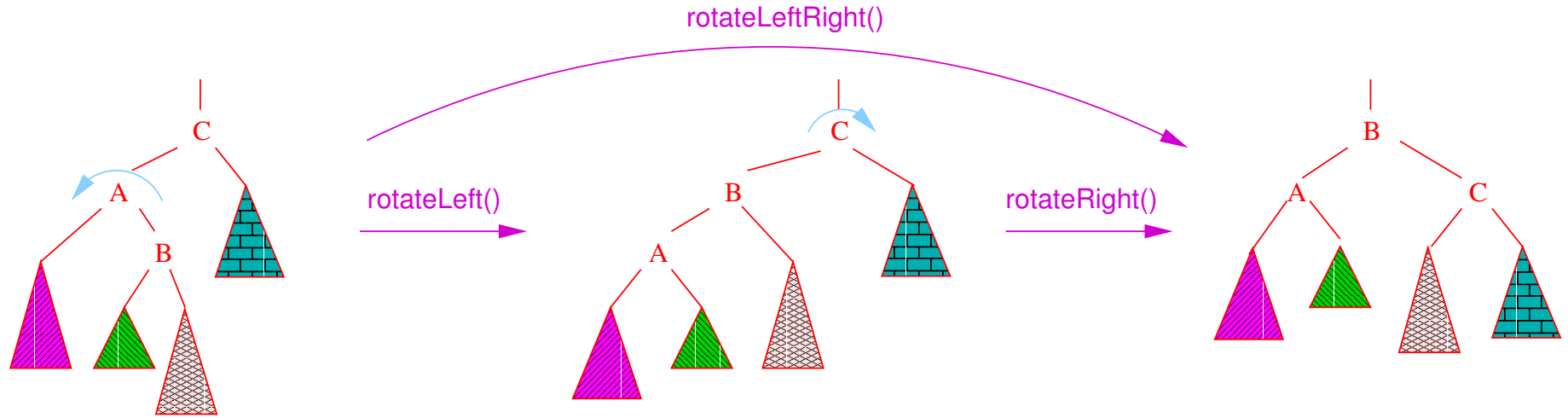
# When Single Rotations Work

- Single rotations balance the tree when the unbalanced subtree is on the outside



# Double Rotations

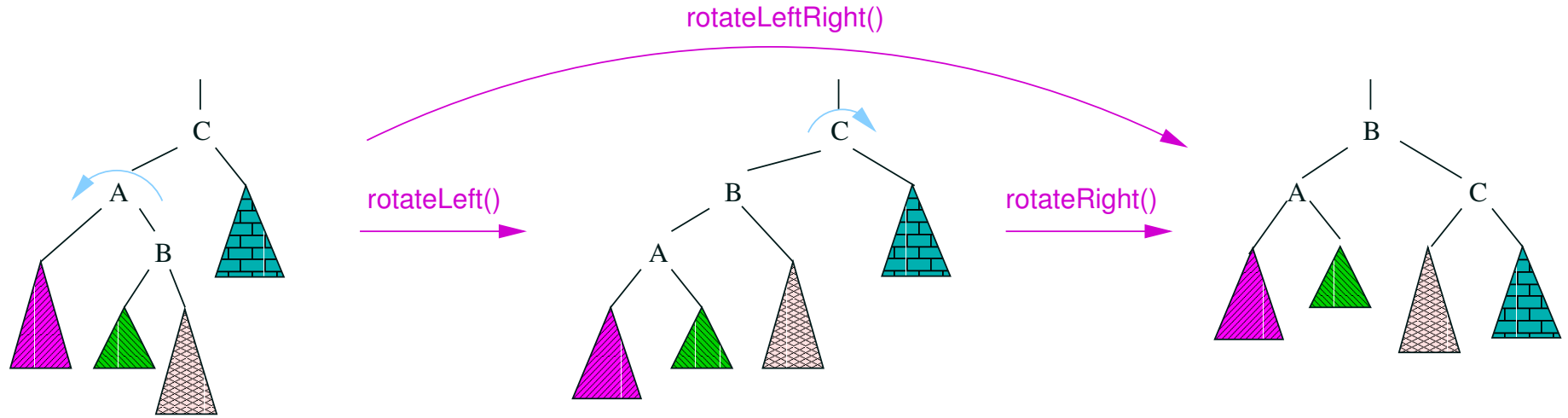
- If the unbalanced subtree is on the inside we need a double rotation



```
leftRotation(C.left);  
rightRotation(C);
```

# Double Rotations

- If the unbalanced subtree is on the inside we need a double rotation



```
leftRotation(C.left);  
rightRotation(C);
```

# Outline

1. Deletion
2. Balancing Trees
  - Rotations
3. **AVL**
4. Red-Black Trees
  - TreeSet
  - TreeMap



# Balancing Trees

- There are different strategies for using rotations for balancing trees
- The three most popular are
  - ★ AVL-trees
  - ★ Red-black trees
  - ★ Splay trees
- They differ in the criteria they use for doing rotations

# Balancing Trees

- There are different strategies for using rotations for balancing trees
- The three most popular are
  - ★ AVL-trees
  - ★ Red-black trees
  - ★ Splay trees
- They differ in the criteria they use for doing rotations



# Balancing Trees

- There are different strategies for using rotations for balancing trees
- The three most popular are
  - ★ AVL-trees
  - ★ Red-black trees
  - ★ Splay trees
- They differ in the criteria they use for doing rotations

# AVL Trees

- AVL-trees were invented in 1962 by two Russian mathematicians Adelson-Velski and Landis
- In AVL trees
  1. The heights of the left and right subtree differ by at most 1
  2. The left and right subtrees are AVL trees
- This guarantees that the worst case AVL tree has logarithmic depth

# AVL Trees

- AVL-trees were invented in 1962 by two Russian mathematicians Adelson-Velski and Landis
- In AVL trees
  1. The heights of the left and right subtree differ by at most 1
  2. The left and right subtrees are AVL trees
- This guarantees that the worst case AVL tree has logarithmic depth

# AVL Trees

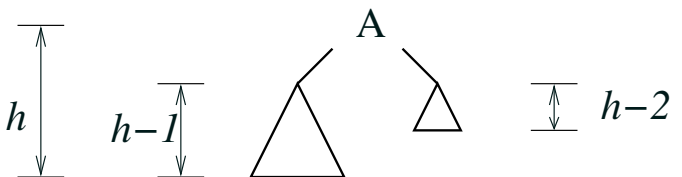
- AVL-trees were invented in 1962 by two Russian mathematicians Adelson-Velski and Landis
- In AVL trees
  1. The heights of the left and right subtree differ by at most 1
  2. The left and right subtrees are AVL trees
- This guarantees that the worst case AVL tree has logarithmic depth

# AVL Trees

- AVL-trees were invented in 1962 by two Russian mathematicians Adelson-Velski and Landis
- In AVL trees
  1. The heights of the left and right subtree differ by at most 1
  2. The left and right subtrees are AVL trees
- This guarantees that the worst case AVL tree has logarithmic depth

# Minimum Number of Nodes

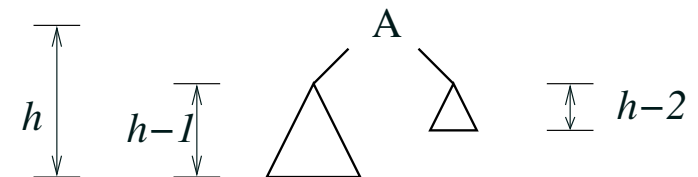
- Let  $m(h)$  be the minimum number of nodes in a tree of height  $h$
- This has to be made up of two subtrees: one of height  $h - 1$ ; and, in the worst case, one of height  $h - 2$
- Thus, the least number of nodes in a tree of height  $h$  is

$$m(h) = m(h - 1) + m(h - 2) + 1$$


- with  $m(1) = 1$ ,  $m(2) = 2$

# Minimum Number of Nodes

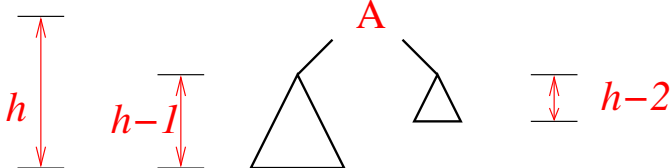
- Let  $m(h)$  be the minimum number of nodes in a tree of height  $h$
- This has to be made up of two subtrees: one of height  $h - 1$ ; and, in the worst case, one of height  $h - 2$
- Thus, the least number of nodes in a tree of height  $h$  is

$$m(h) = m(h - 1) + m(h - 2) + 1$$


- with  $m(1) = 1$ ,  $m(2) = 2$

# Minimum Number of Nodes

- Let  $m(h)$  be the minimum number of nodes in a tree of height  $h$
- This has to be made up of two subtrees: one of height  $h - 1$ ; and, in the worst case, one of height  $h - 2$
- Thus, the least number of nodes in a tree of height  $h$  is

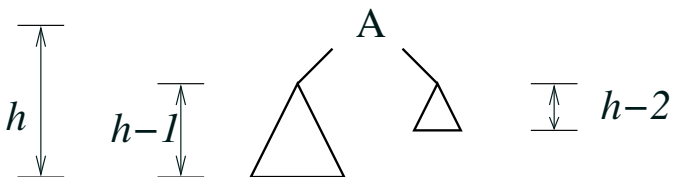
$$m(h) = m(h - 1) + m(h - 2) + 1$$


- with  $m(1) = 1$ ,  $m(2) = 2$



# Minimum Number of Nodes

- Let  $m(h)$  be the minimum number of nodes in a tree of height  $h$
- This has to be made up of two subtrees: one of height  $h - 1$ ; and, in the worst case, one of height  $h - 2$
- Thus, the least number of nodes in a tree of height  $h$  is

$$m(h) = m(h - 1) + m(h - 2) + 1$$


- with  $m(1) = 1$ ,  $m(2) = 2$

# Proof of Exponential Number of Nodes

- We have  $m(h) = m(h - 1) + m(h - 2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- This gives us a sequence  $1, 2, 4, 7, 12, \dots$
- Compare this with Fibonacci  $f(h) = f(h - 1) + f(h - 2)$ , with  $f(1) = f(2) = 1$
- This gives us a sequence  $1, 1, 2, 3, 5, 8, 13, \dots$
- It looks like  $m(h) = f(h + 2) - 1$
- Proof by substitution

# Proof of Exponential Number of Nodes

- We have  $m(h) = m(h - 1) + m(h - 2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- This gives us a sequence 1, 2, 4, 7, 12,  $\dots$
- Compare this with Fibonacci  $f(h) = f(h - 1) + f(h - 2)$ , with  $f(1) = f(2) = 1$
- This gives us a sequence 1, 1, 2, 3, 5, 8, 13,  $\dots$
- It looks like  $m(h) = f(h + 2) - 1$
- Proof by substitution

# Proof of Exponential Number of Nodes

- We have  $m(h) = m(h - 1) + m(h - 2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- This gives us a sequence 1, 2, 4, 7, 12, ...
- Compare this with Fibonacci  $f(h) = f(h - 1) + f(h - 2)$ , with  $f(1) = f(2) = 1$
- This gives us a sequence 1, 1, 2, 3, 5, 8, 13, ...
- It looks like  $m(h) = f(h + 2) - 1$
- Proof by substitution

# Proof of Exponential Number of Nodes

- We have  $m(h) = m(h - 1) + m(h - 2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- This gives us a sequence  $1, 2, 4, 7, 12, \dots$
- Compare this with Fibonacci  $f(h) = f(h - 1) + f(h - 2)$ , with  $f(1) = f(2) = 1$
- This gives us a sequence  $1, 1, 2, 3, 5, 8, 13, \dots$
- It looks like  $m(h) = f(h + 2) - 1$
- Proof by substitution

# Proof of Exponential Number of Nodes

- We have  $m(h) = m(h - 1) + m(h - 2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- This gives us a sequence  $1, 2, 4, 7, 12, \dots$
- Compare this with Fibonacci  $f(h) = f(h - 1) + f(h - 2)$ , with  $f(1) = f(2) = 1$
- This gives us a sequence  $1, 1, 2, 3, 5, 8, 13, \dots$
- It looks like  $m(h) = f(h + 2) - 1$
- Proof by substitution

# Proof of Exponential Number of Nodes

- We have  $m(h) = m(h - 1) + m(h - 2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- This gives us a sequence  $1, 2, 4, 7, 12, \dots$
- Compare this with Fibonacci  $f(h) = f(h - 1) + f(h - 2)$ , with  $f(1) = f(2) = 1$
- This gives us a sequence  $1, 1, 2, 3, 5, 8, 13, \dots$
- It looks like  $m(h) = f(h + 2) - 1$
- Proof by substitution

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1, m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1, m(2) = 2 \geq (3/2)^1 = 3/2$

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$



# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1}$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$



# Proof of Logarithmic Depth

- $m(h) = m(h - 1) + m(h - 2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1} \quad \checkmark$$

- Taking logs:  $\log(m(h)) \geq (h - 1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1} \quad \checkmark$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$  with  $m(1) = 1$ ,  $m(2) = 2$
- We can prove by inductions,  $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$ ,  $m(2) = 2 \geq (3/2)^1 = 3/2$  ✓

$$m(h) \geq \left(\frac{3}{2}\right)^{h-3} \left(\frac{3}{2} + 1 + \left(\frac{3}{2}\right)^{3-h}\right) \geq \left(\frac{3}{2}\right)^{h-3} \frac{5}{2} = \left(\frac{3}{2}\right)^{h-3} \frac{10}{4} \geq \left(\frac{3}{2}\right)^{h-3} \frac{9}{4} = \left(\frac{3}{2}\right)^{h-1} \quad \checkmark$$

- Taking logs:  $\log(m(h)) \geq (h-1) \log(3/2)$  or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O(\log(m(h)))$$

- The number of elements,  $n$ , we can store in an AVL tree is  $n \geq m(h)$  thus

$$h \leq O(\log(n))$$

# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

add(16)

# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

16 0

# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

add(51)

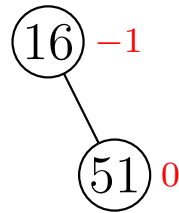
16 0



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

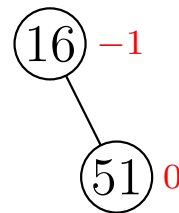


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

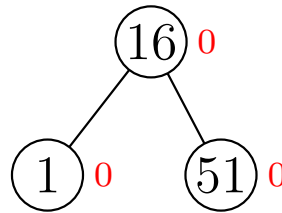
add(1)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

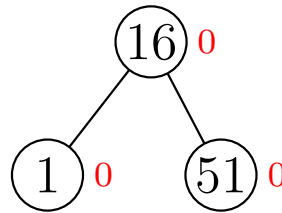


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

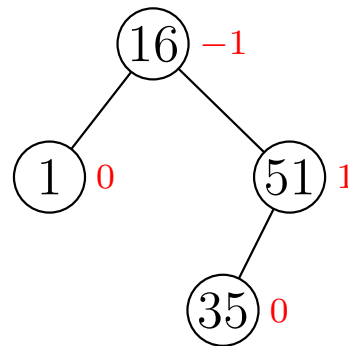
add(35)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

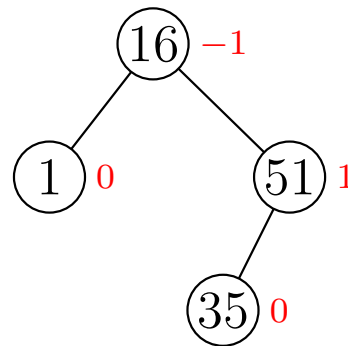


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

add(23)

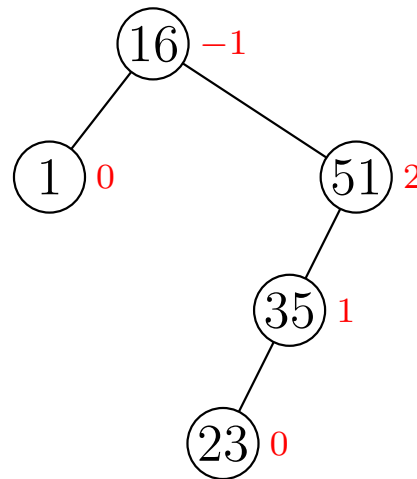


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

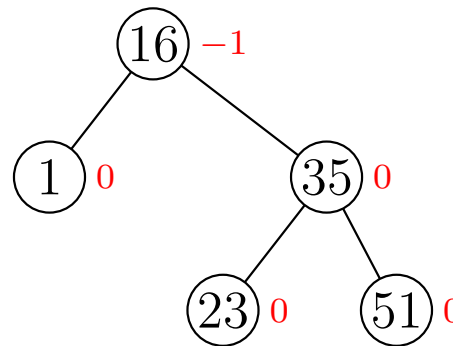
RotateRight



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



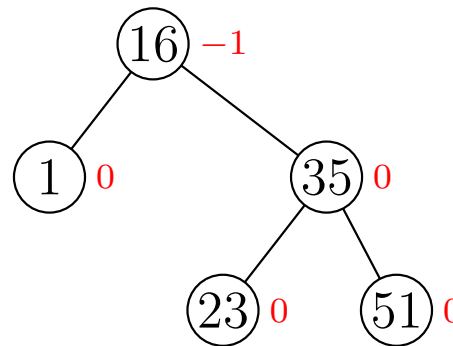


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

add(45)

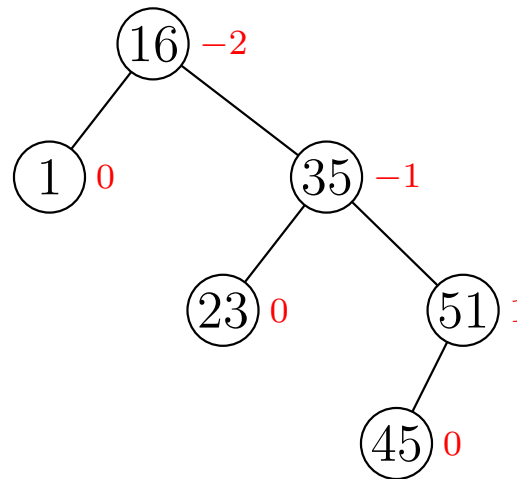


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

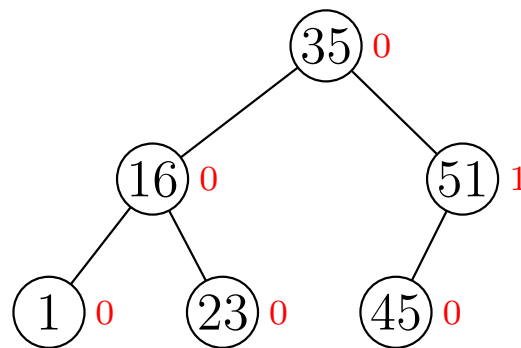
RotateLeft



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

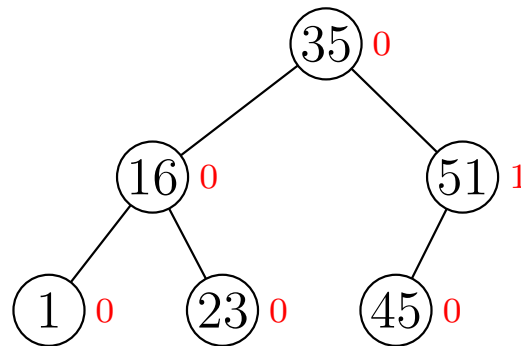


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

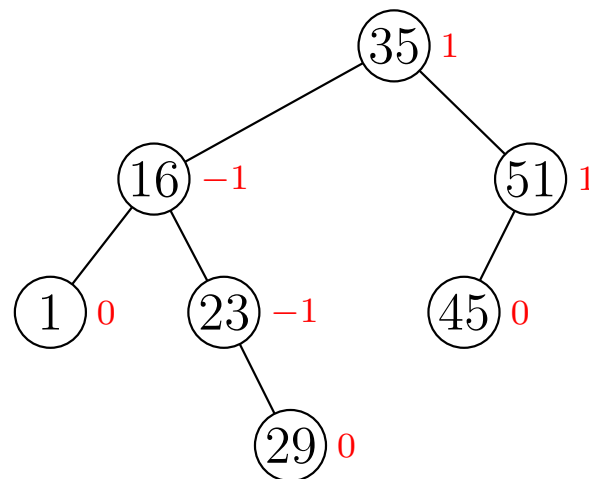
add(29)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

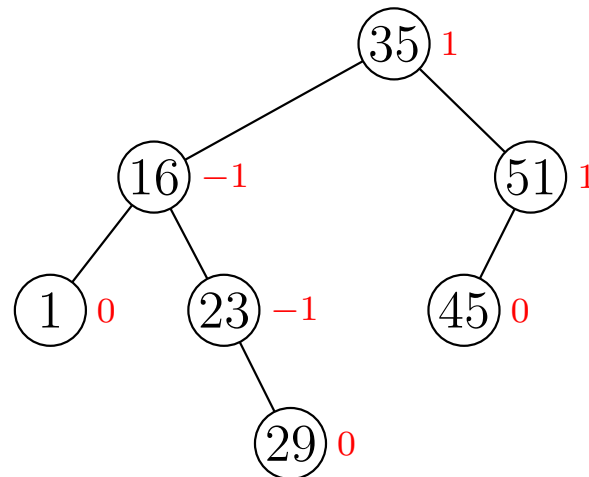


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

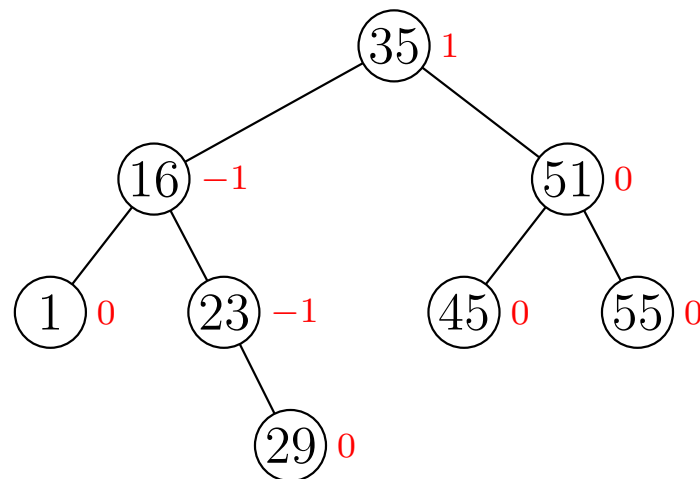
add(55)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

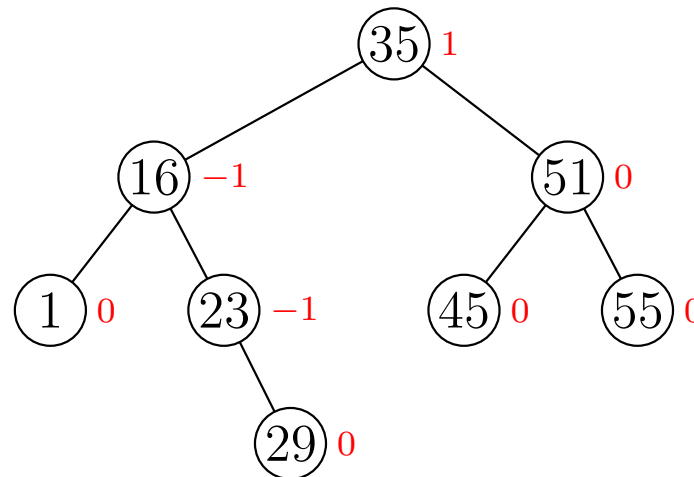


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

add(88)

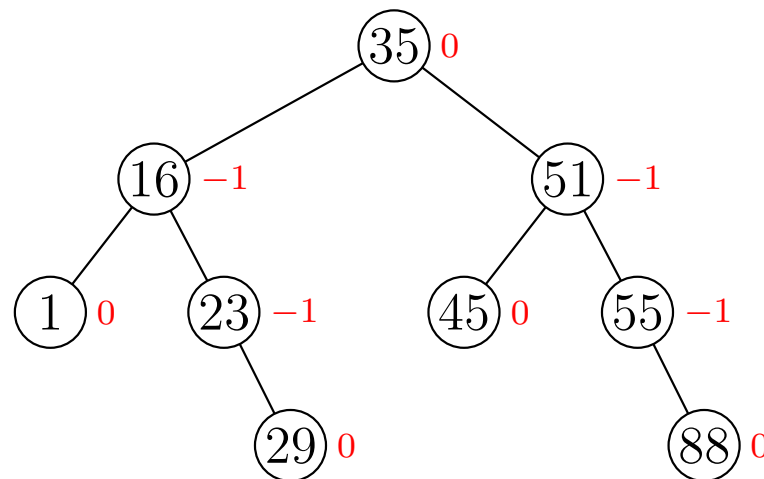




# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

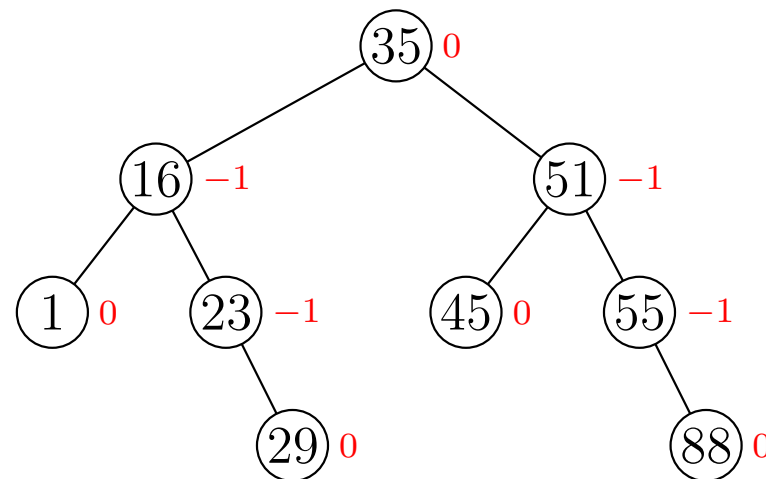


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

add(91)

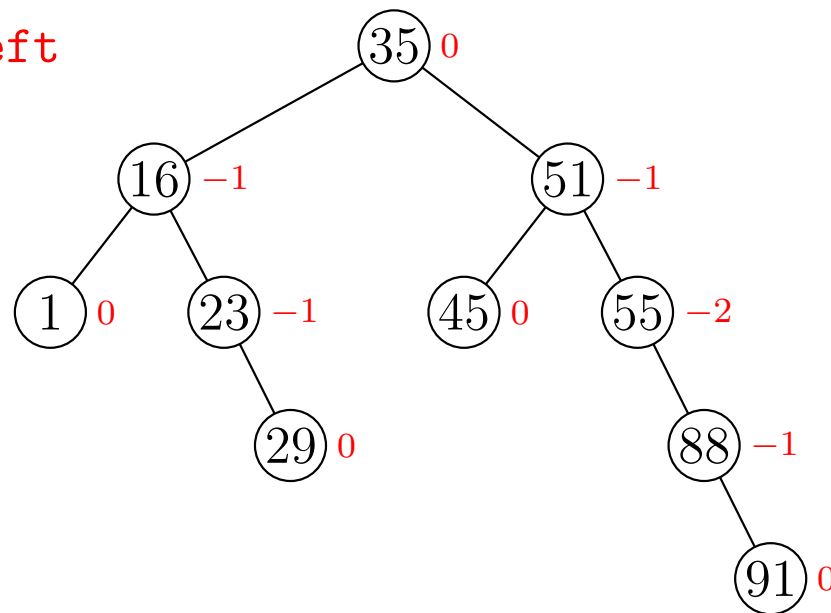


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

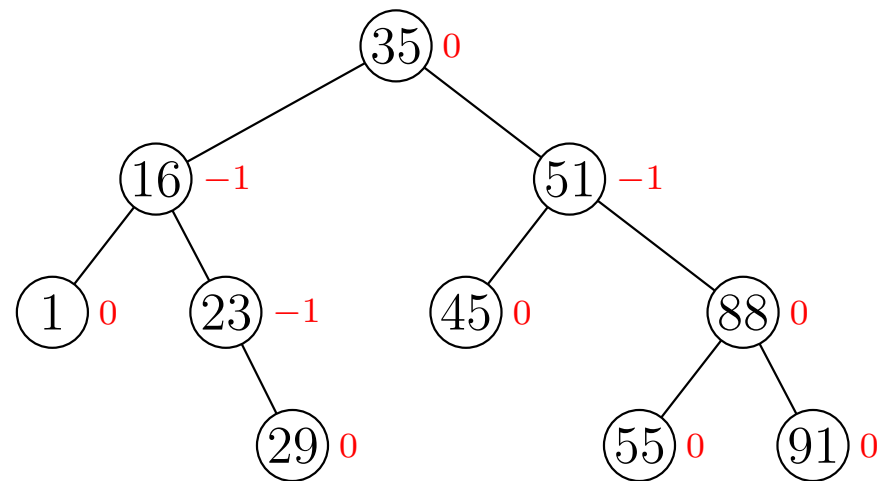
RotateLeft



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

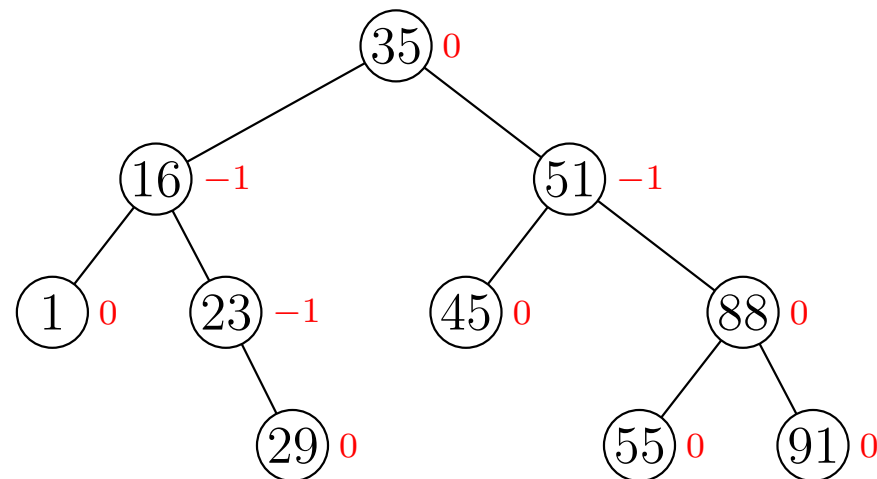


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

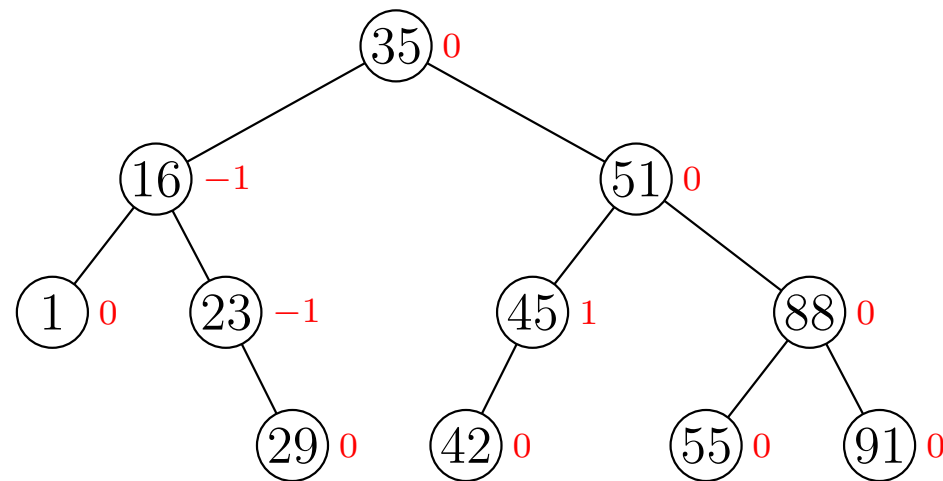
add(42)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

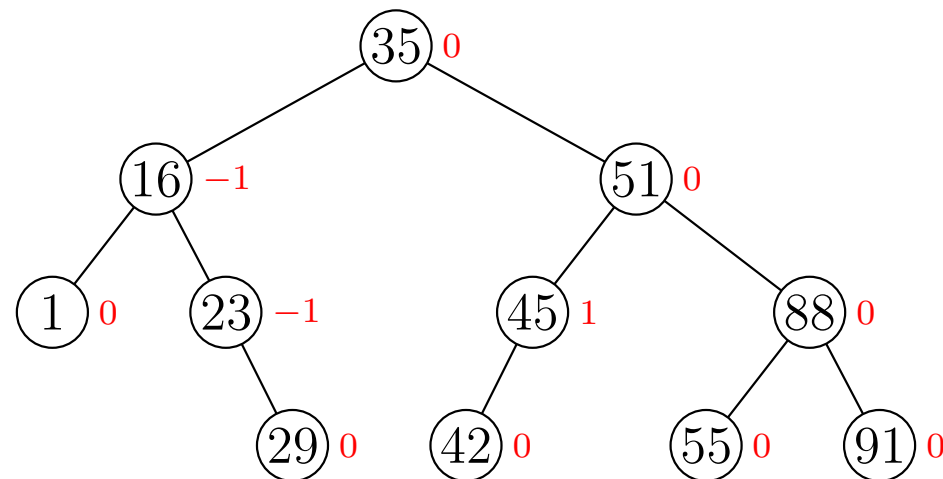


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

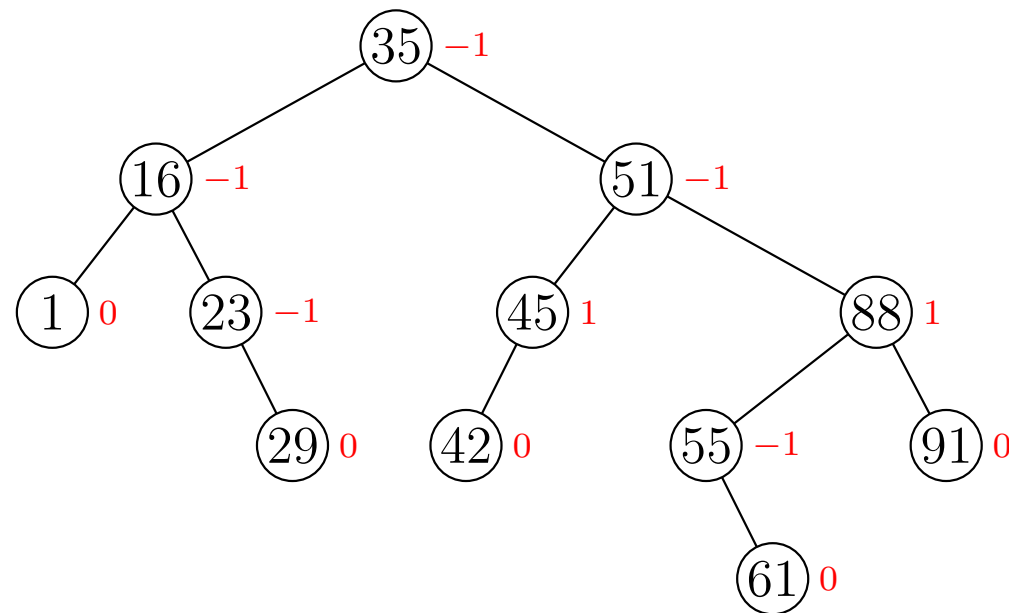
add(61)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



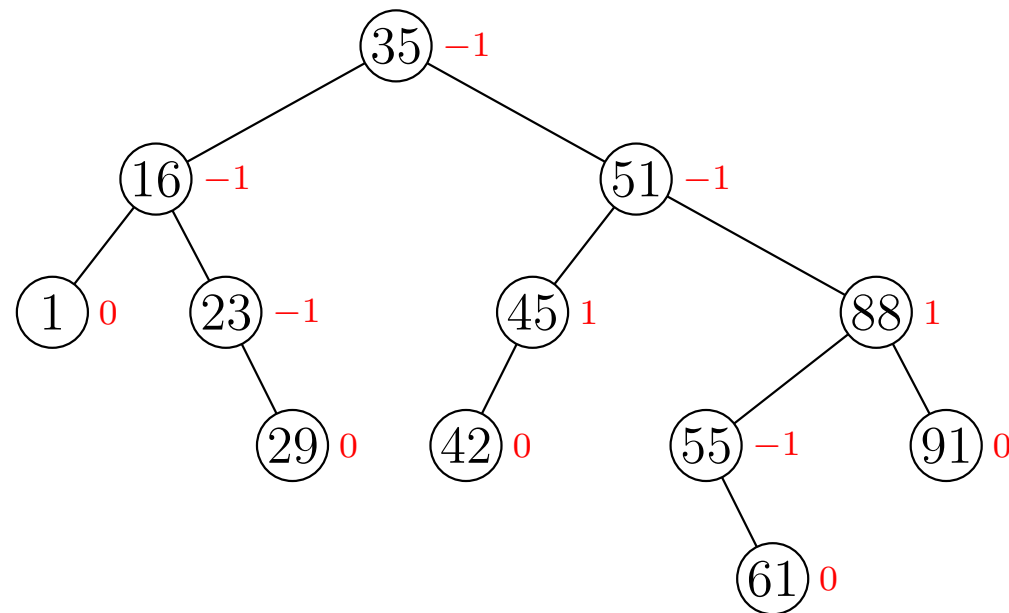


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

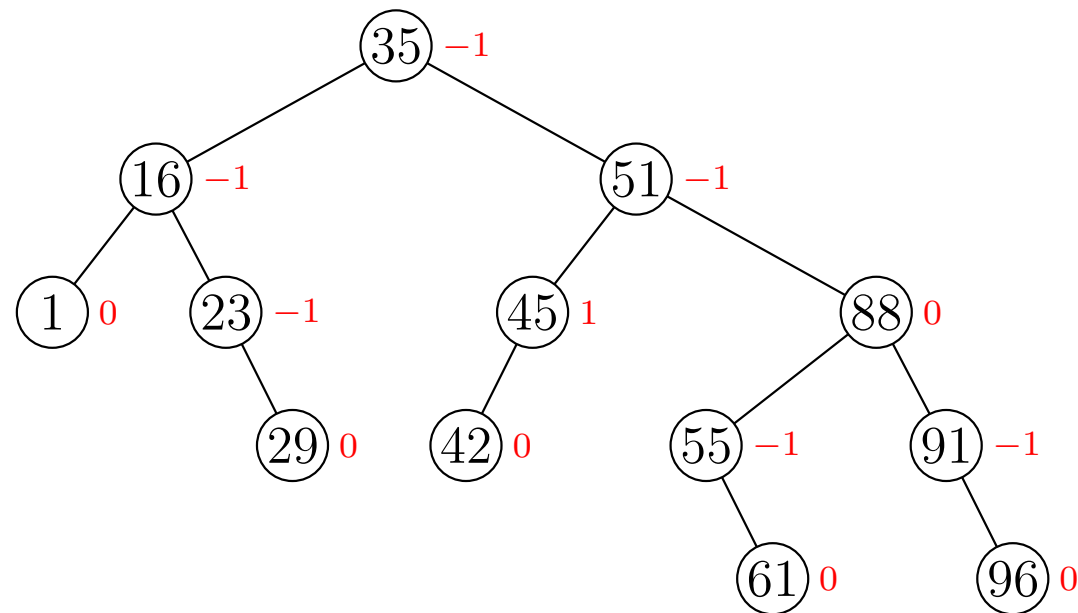
add(96)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

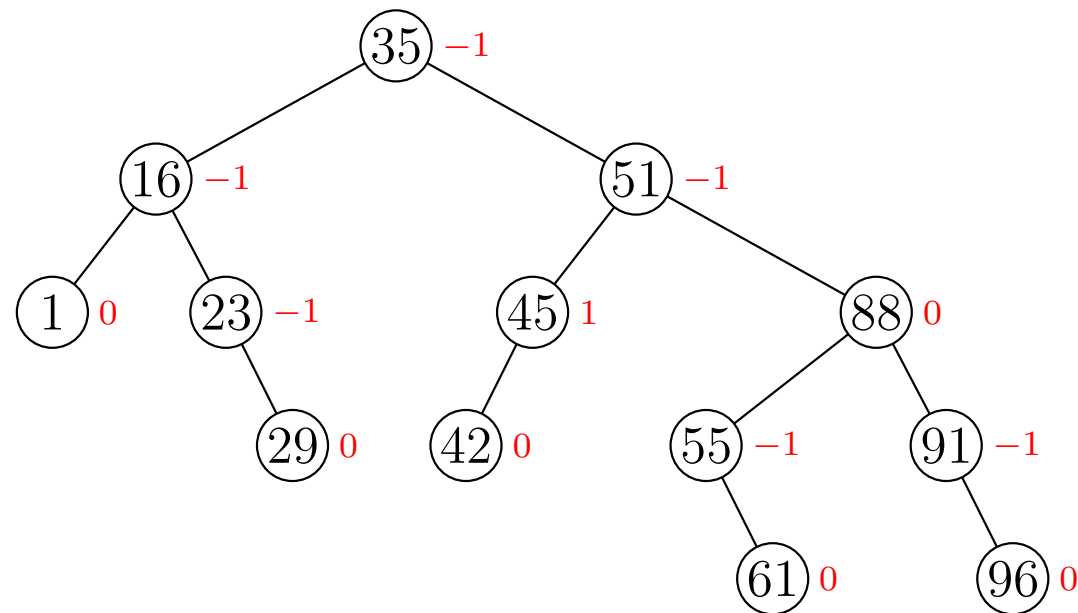


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

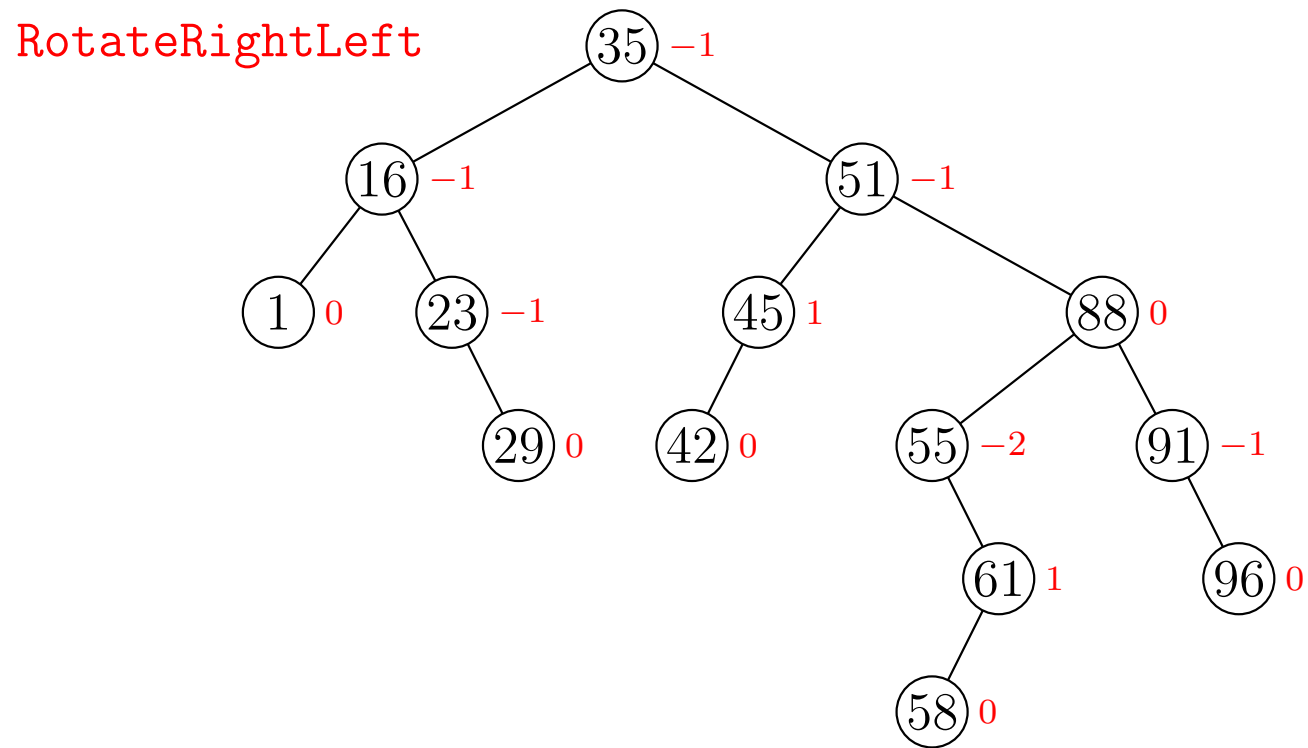
add(58)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

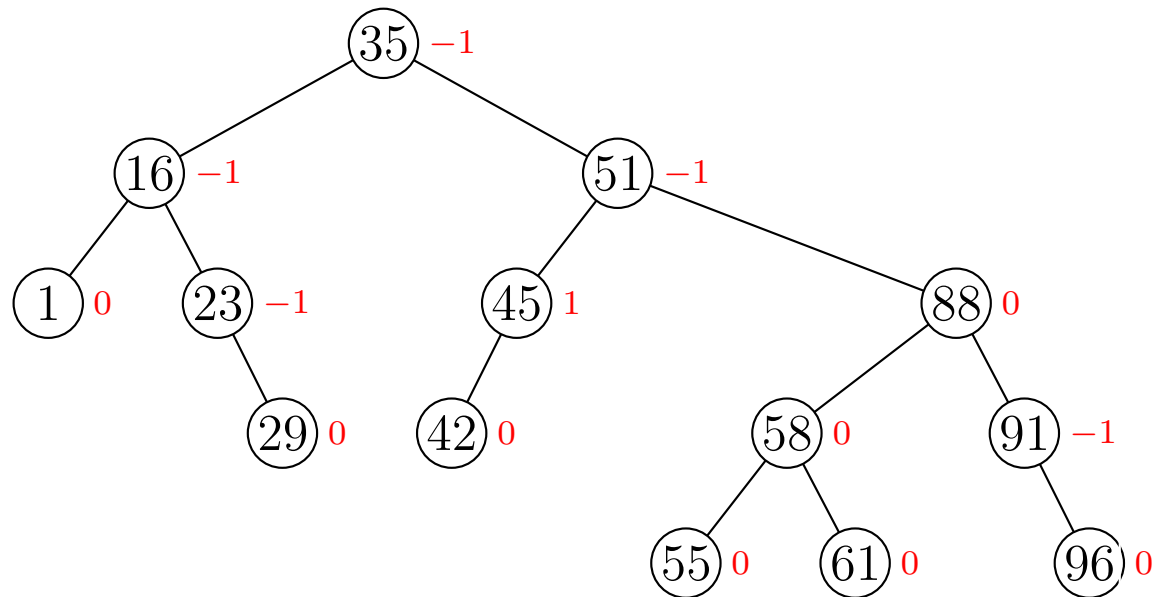
$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

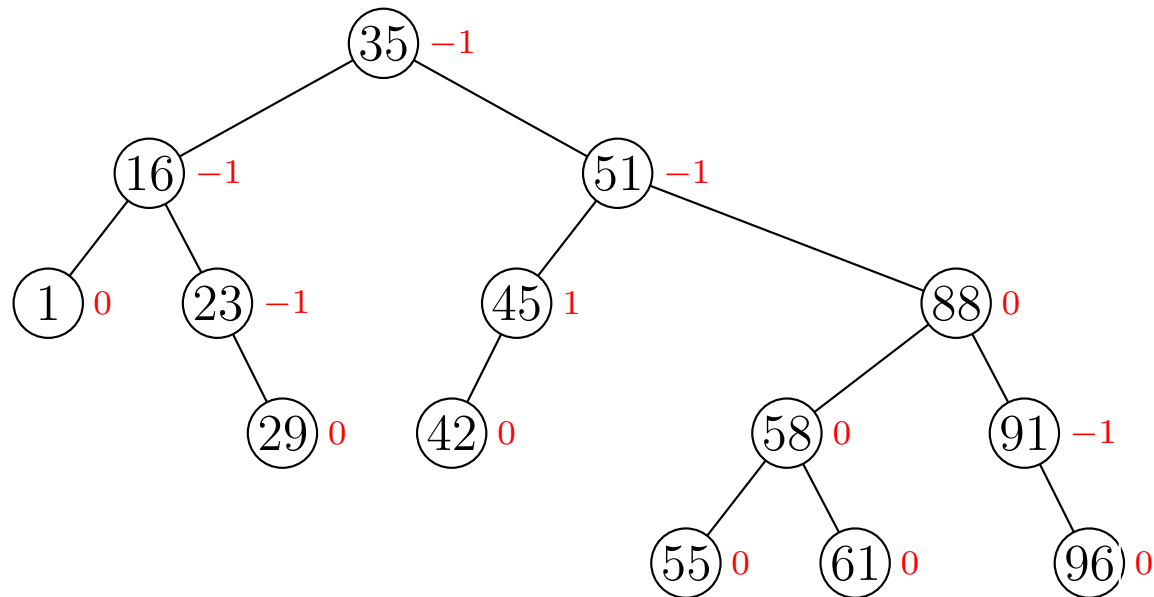


# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

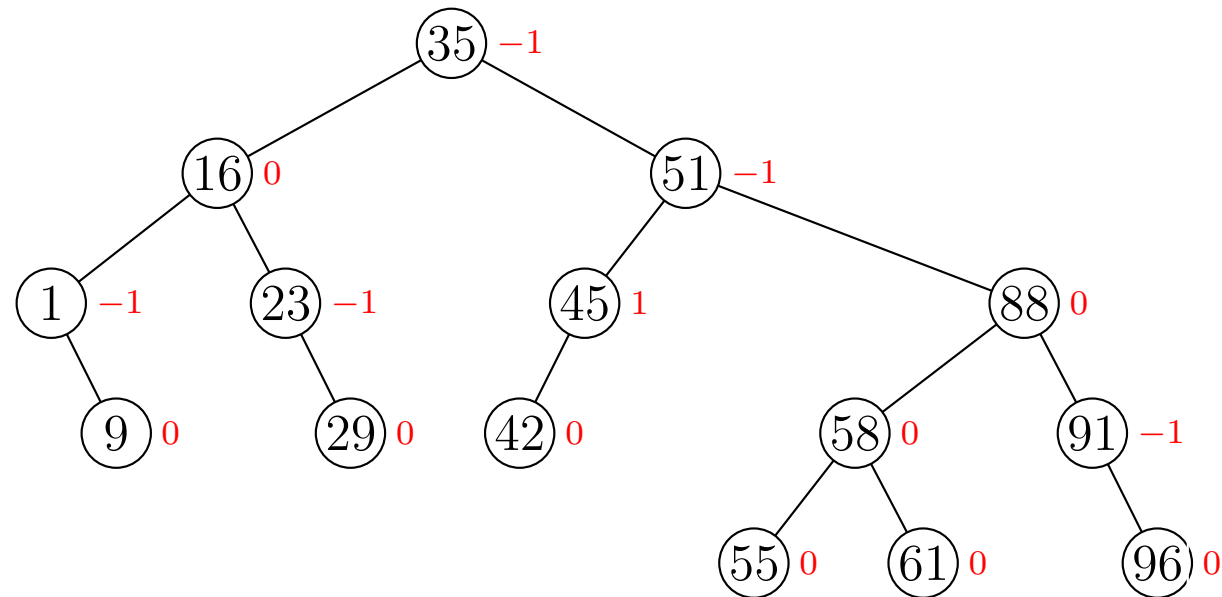
add(9)



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

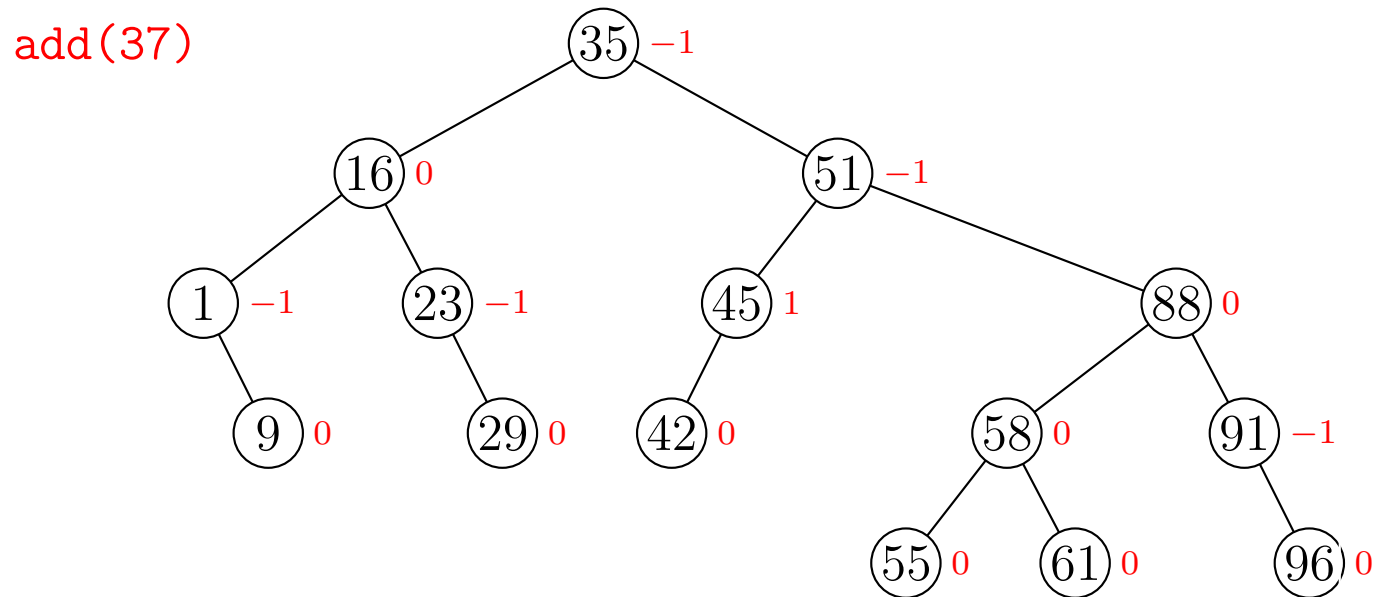
$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$

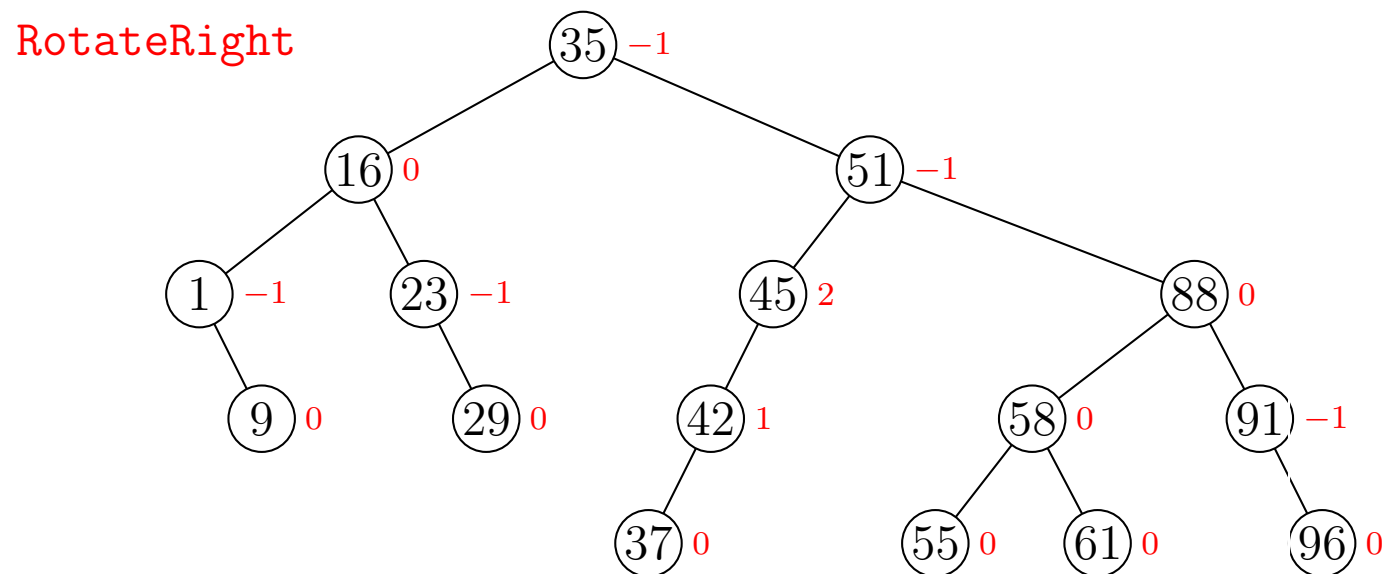




# Implementing AVL Trees

- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

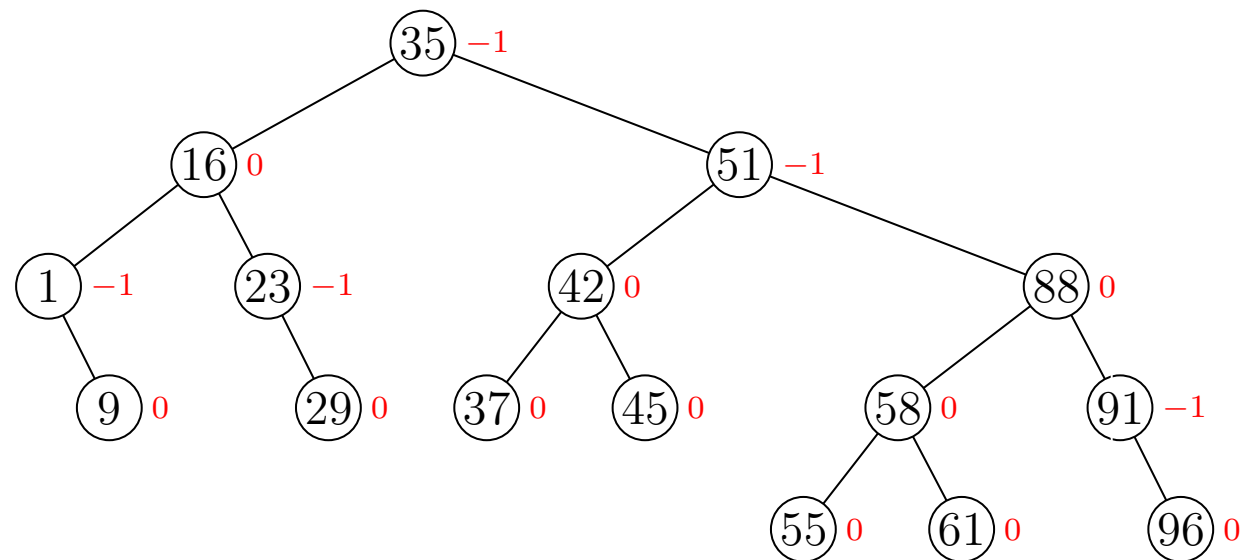
$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



# Implementing AVL Trees

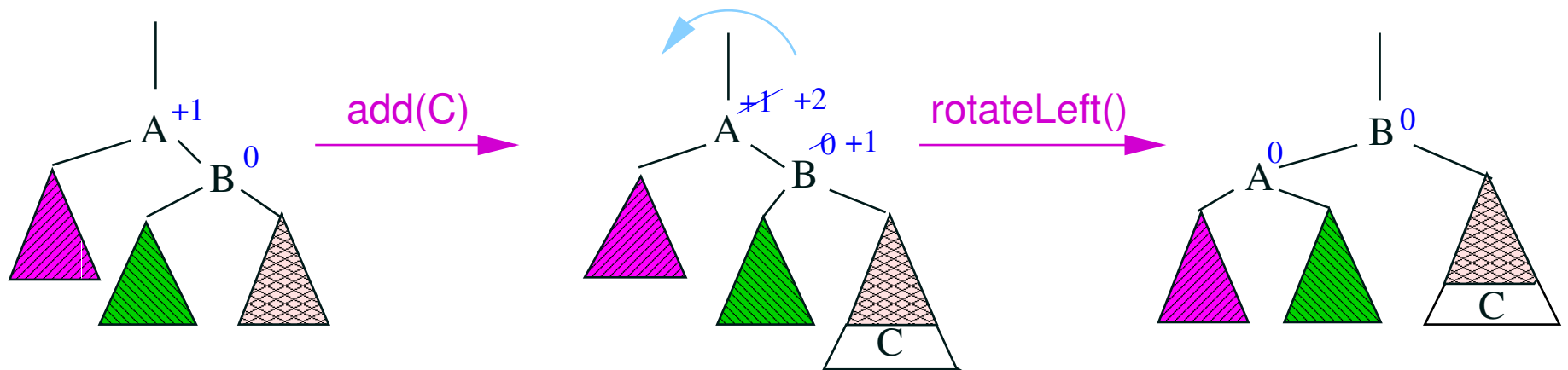
- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\text{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



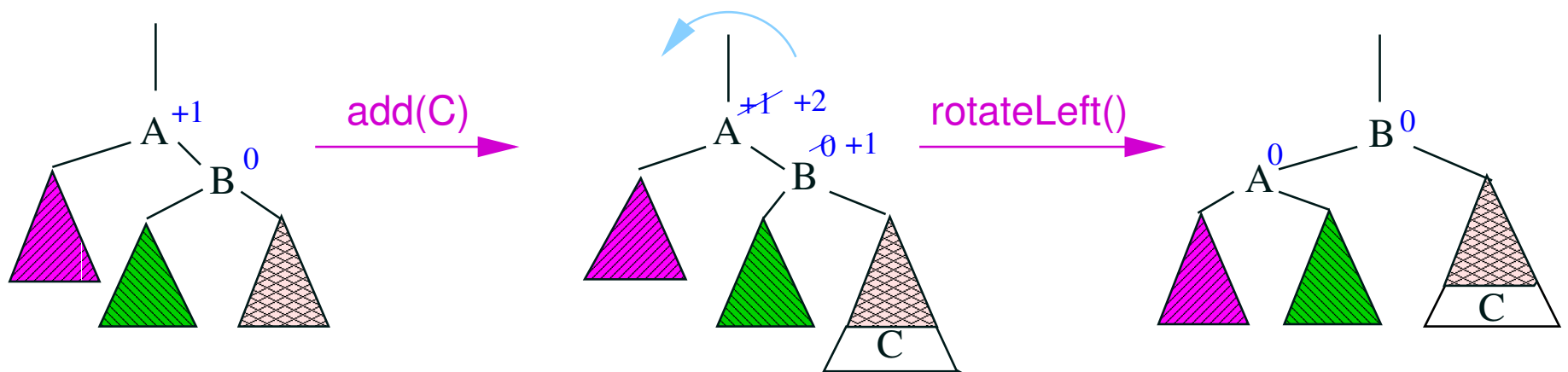
# Balancing AVL Trees

- When adding an element to an AVL tree
  - ★ Find the location where it is to be inserted
  - ★ Iterate up through the parents re-adjusting the balanceFactor
  - ★ **If** the balance factor exceeds  $\pm 1$  then re-balance the tree and stop
  - ★ **else** if the balance factor goes to zero **then** stop



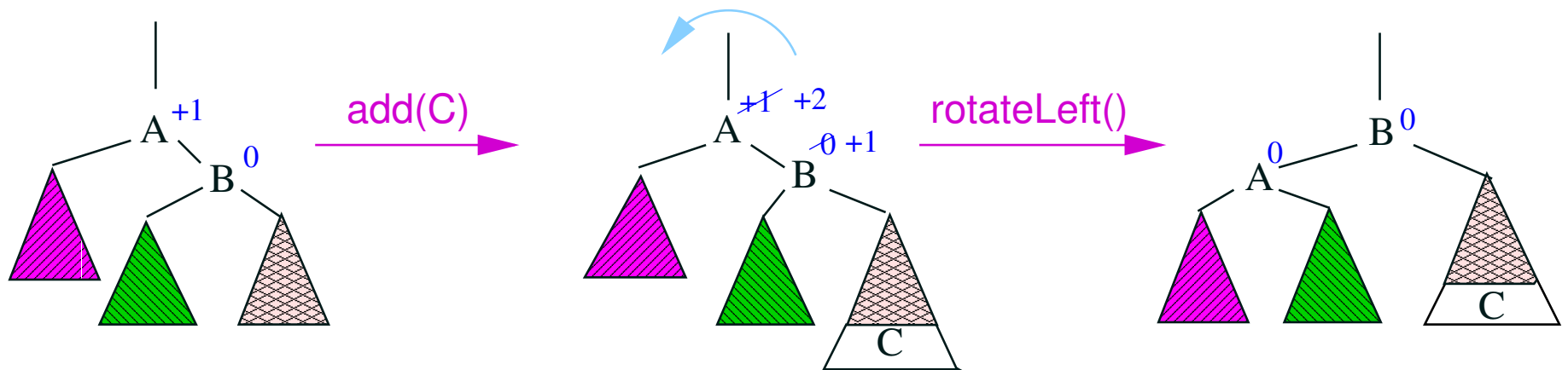
# Balancing AVL Trees

- When adding an element to an AVL tree
  - ★ Find the location where it is to be inserted
  - ★ Iterate up through the parents re-adjusting the balanceFactor
  - ★ If the balance factor exceeds  $\pm 1$  then re-balance the tree and stop
  - ★ else if the balance factor goes to zero **then** stop



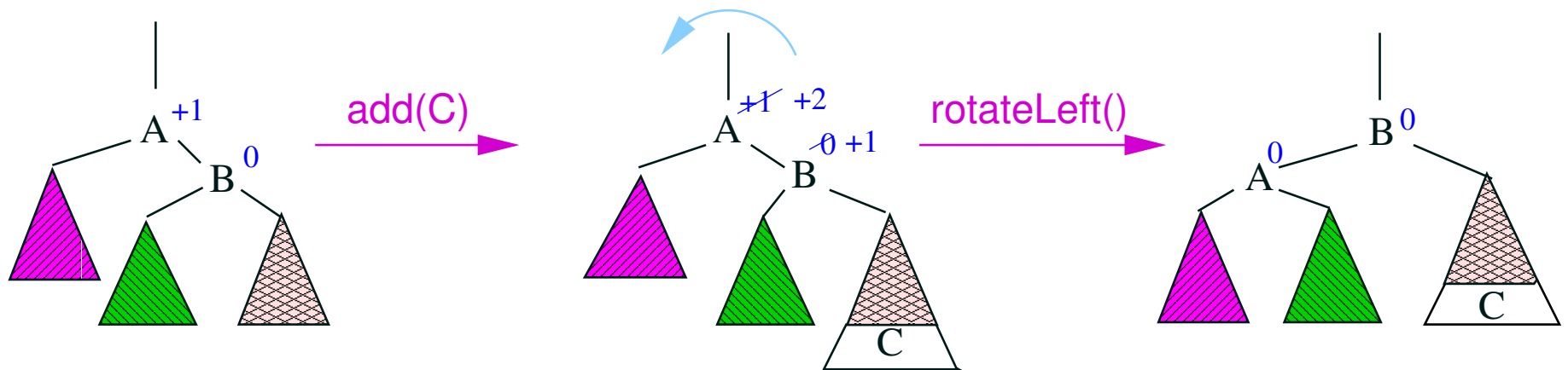
# Balancing AVL Trees

- When adding an element to an AVL tree
  - ★ Find the location where it is to be inserted
  - ★ Iterate up through the parents re-adjusting the `balanceFactor`
  - ★ If the balance factor exceeds  $\pm 1$  then re-balance the tree and stop
  - ★ else if the balance factor goes to zero then stop



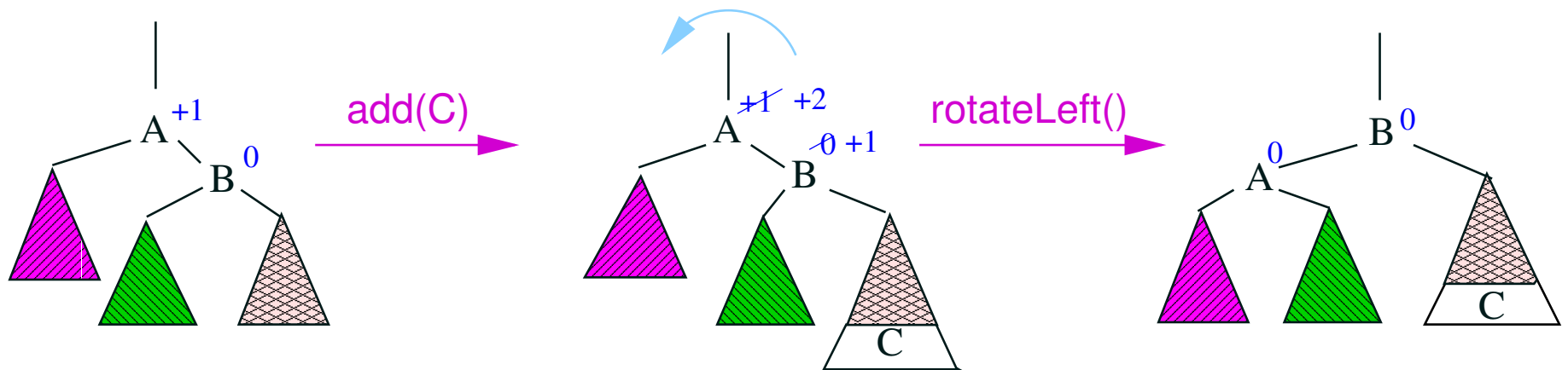
# Balancing AVL Trees

- When adding an element to an AVL tree
  - ★ Find the location where it is to be inserted
  - ★ Iterate up through the parents re-adjusting the balanceFactor
  - ★ **If the balance factor exceeds  $\pm 1$  then re-balance the tree and stop**
  - ★ else if the balance factor goes to zero **then stop**



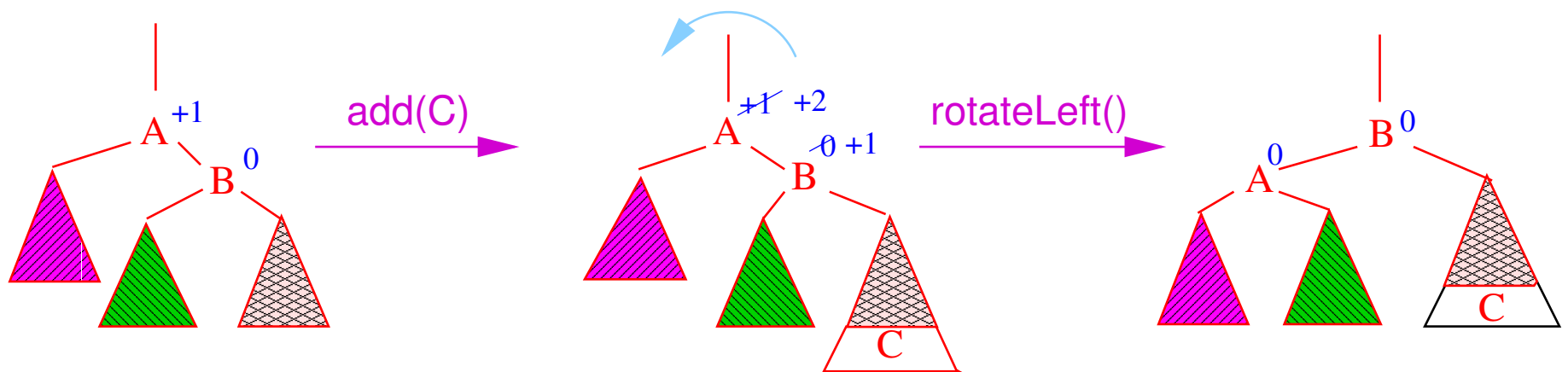
# Balancing AVL Trees

- When adding an element to an AVL tree
  - ★ Find the location where it is to be inserted
  - ★ Iterate up through the parents re-adjusting the balanceFactor
  - ★ **If** the balance factor exceeds  $\pm 1$  then re-balance the tree and stop
  - ★ **else if** the balance factor goes to zero **then** stop



# Balancing AVL Trees

- When adding an element to an AVL tree
  - ★ Find the location where it is to be inserted
  - ★ Iterate up through the parents re-adjusting the balanceFactor
  - ★ **If** the balance factor exceeds  $\pm 1$  then re-balance the tree and stop
  - ★ **else** if the balance factor goes to zero **then** stop





# AVL Deletions

- AVL deletions are similar to AVL insertions
- One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined
- In the worst case  $\Theta(\log(n))$  rotations may be necessary
- This may be relatively slow—but in many applications deletions are rare

# AVL Deletions

- AVL deletions are similar to AVL insertions
- One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined
- In the worst case  $\Theta(\log(n))$  rotations may be necessary
- This may be relatively slow—but in many applications deletions are rare

# AVL Deletions

- AVL deletions are similar to AVL insertions
- One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined
- In the worst case  $\Theta(\log(n))$  rotations may be necessary
- This may be relatively slow—but in many applications deletions are rare

# AVL Deletions

- AVL deletions are similar to AVL insertions
- One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined
- In the worst case  $\Theta(\log(n))$  rotations may be necessary
- This may be relatively slow—but in many applications deletions are rare

# AVL Deletions

- AVL deletions are similar to AVL insertions
- One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined
- In the worst case  $\Theta(\log(n))$  rotations may be necessary
- This may be relatively slow—but in many applications deletions are rare

# AVL Tree Performance

- Insertion, deletion and search in AVL trees are, at worst,  $\Theta(\log(n))$
- The height of an average AVL tree is  $1.44 \log_2(n)$
- The height of an average binary search tree is  $2.1 \log_2(n)$
- Despite being more compact insertion is slightly slower in AVL trees than binary search trees without balancing (for random input sequences)
- Search is, of course, quicker

# AVL Tree Performance

- Insertion, deletion and search in AVL trees are, at worst,  $\Theta(\log(n))$
- The height of an average AVL tree is  $1.44 \log_2(n)$
- The height of an average binary search tree is  $2.1 \log_2(n)$
- Despite being more compact insertion is slightly slower in AVL trees than binary search trees without balancing (for random input sequences)
- Search is, of course, quicker

# AVL Tree Performance

- Insertion, deletion and search in AVL trees are, at worst,  $\Theta(\log(n))$
- The height of an average AVL tree is  $1.44 \log_2(n)$
- The height of an average binary search tree is  $2.1 \log_2(n)$
- Despite being more compact insertion is slightly slower in AVL trees than binary search trees without balancing (for random input sequences)
- Search is, of course, quicker



# AVL Tree Performance

- Insertion, deletion and search in AVL trees are, at worst,  $\Theta(\log(n))$
- The height of an average AVL tree is  $1.44 \log_2(n)$
- The height of an average binary search tree is  $2.1 \log_2(n)$
- Despite being more compact insertion is slightly slower in AVL trees than binary search trees without balancing (for random input sequences)
- Search is, of course, quicker

# AVL Tree Performance

- Insertion, deletion and search in AVL trees are, at worst,  $\Theta(\log(n))$
- The height of an average AVL tree is  $1.44 \log_2(n)$
- The height of an average binary search tree is  $2.1 \log_2(n)$
- Despite being more compact insertion is slightly slower in AVL trees than binary search trees without balancing (for random input sequences)
- Search is, of course, quicker

# Outline

1. Deletion
2. Balancing Trees
  - Rotations
3. AVL
4. **Red-Black Trees**
  - TreeSet
  - TreeMap

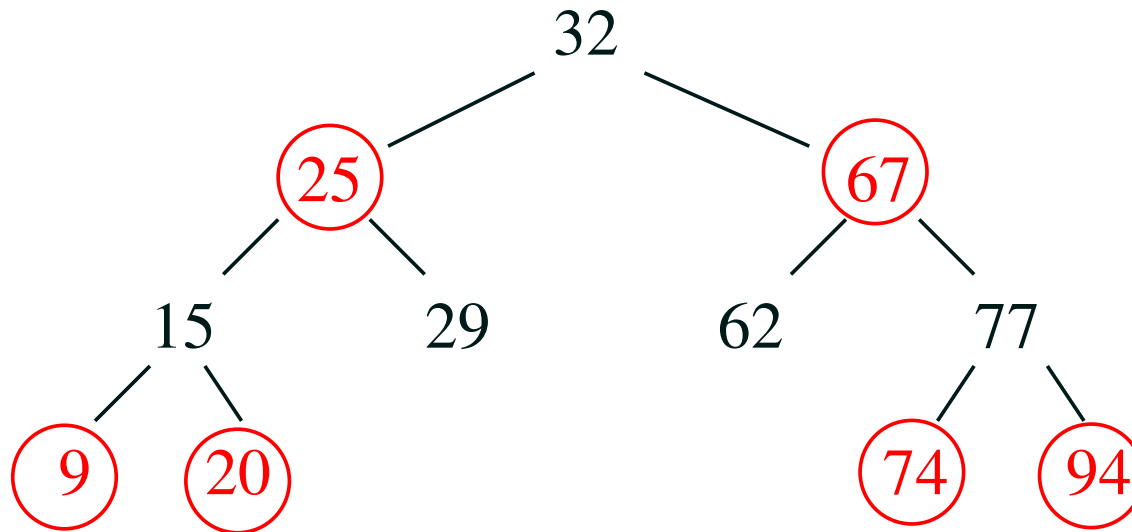


# Red-Black Trees

- Red-black trees are another strategy for balancing trees
- Nodes are either *red* or *black*
- Two rules are imposed

**Red Rule:** the children of a red node must be black

**Black Rule:** the number of black elements must be the same in all paths from the root to elements with no children or with one child

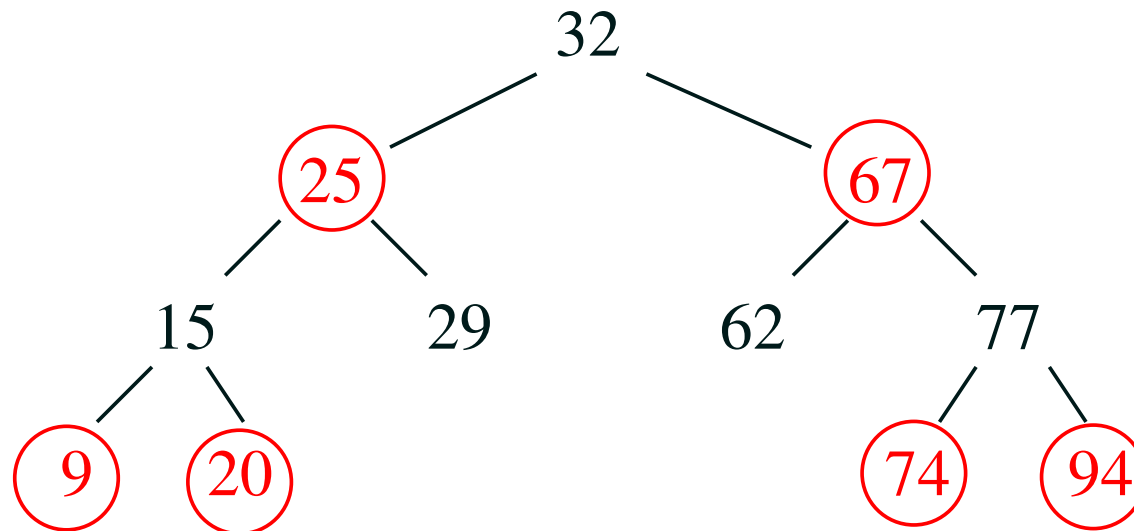


# Red-Black Trees

- Red-black trees are another strategy for balancing trees
- Nodes are either *red* or *black*
- Two rules are imposed

**Red Rule:** the children of a red node must be black

**Black Rule:** the number of black elements must be the same in all paths from the root to elements with no children or with one child

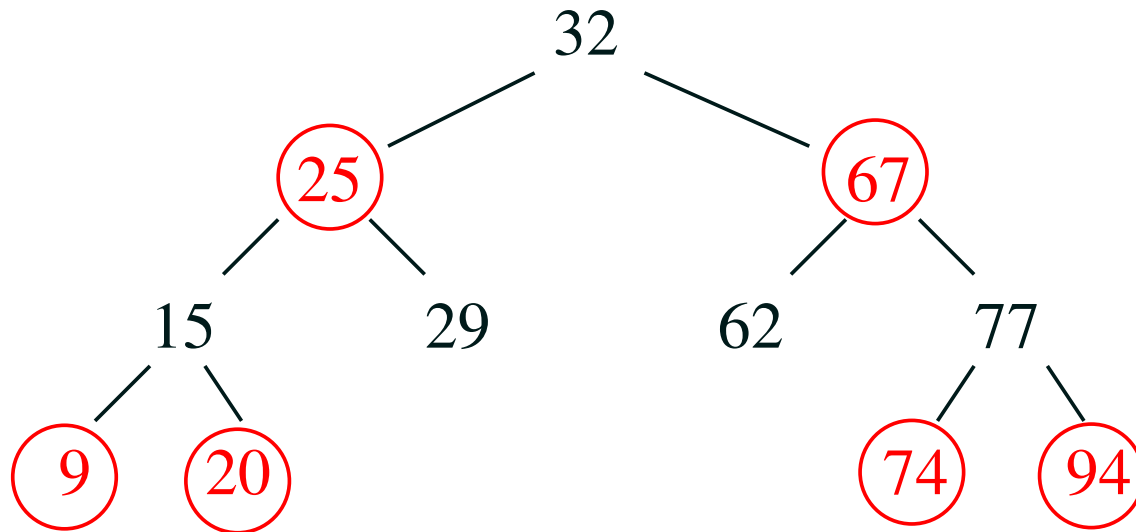


# Red-Black Trees

- Red-black trees are another strategy for balancing trees
- Nodes are either *red* or *black*
- Two rules are imposed

**Red Rule:** the children of a red node must be black

**Black Rule:** the number of black elements must be the same in all paths from the root to elements with no children or with one child

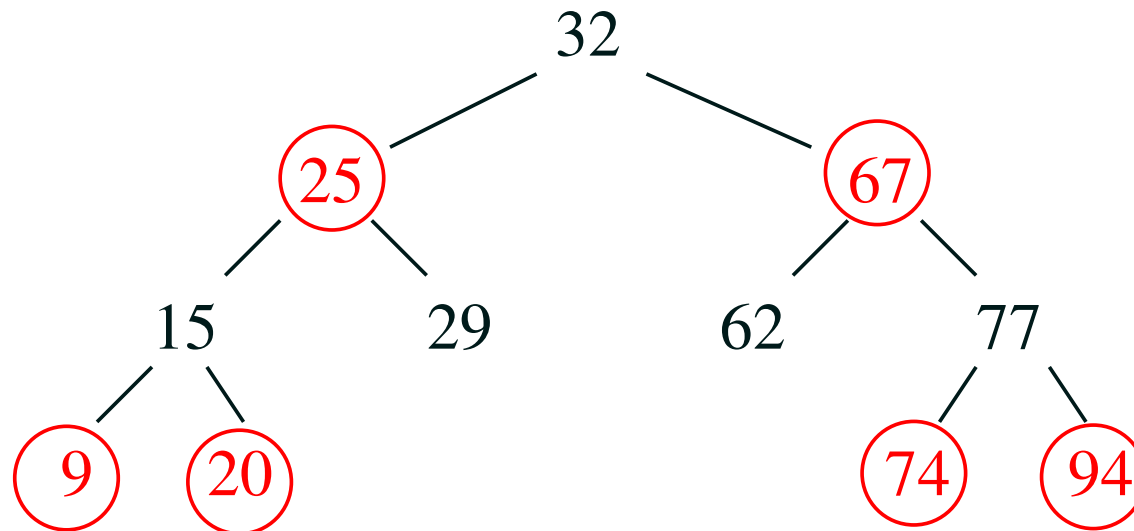


# Red-Black Trees

- Red-black trees are another strategy for balancing trees
- Nodes are either *red* or *black*
- Two rules are imposed

**Red Rule:** the children of a red node must be black

**Black Rule:** the number of black elements must be the same in all paths from the root to elements with no children or with one child

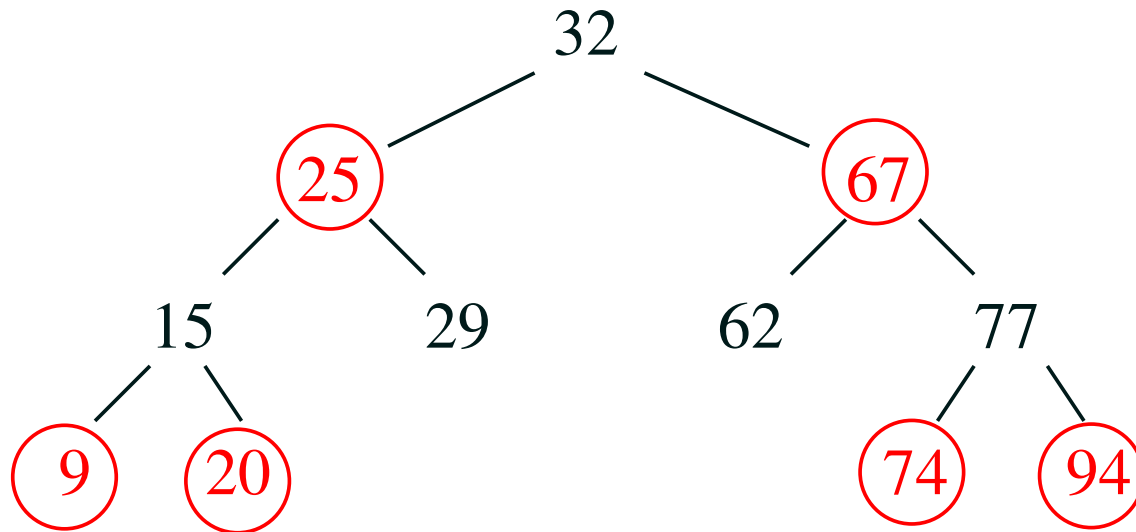


# Red-Black Trees

- Red-black trees are another strategy for balancing trees
- Nodes are either *red* or *black*
- Two rules are imposed

**Red Rule:** the children of a red node must be black

**Black Rule:** the number of black elements must be the same in all paths from the root to elements with no children or with one child





# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree

# Restructuring

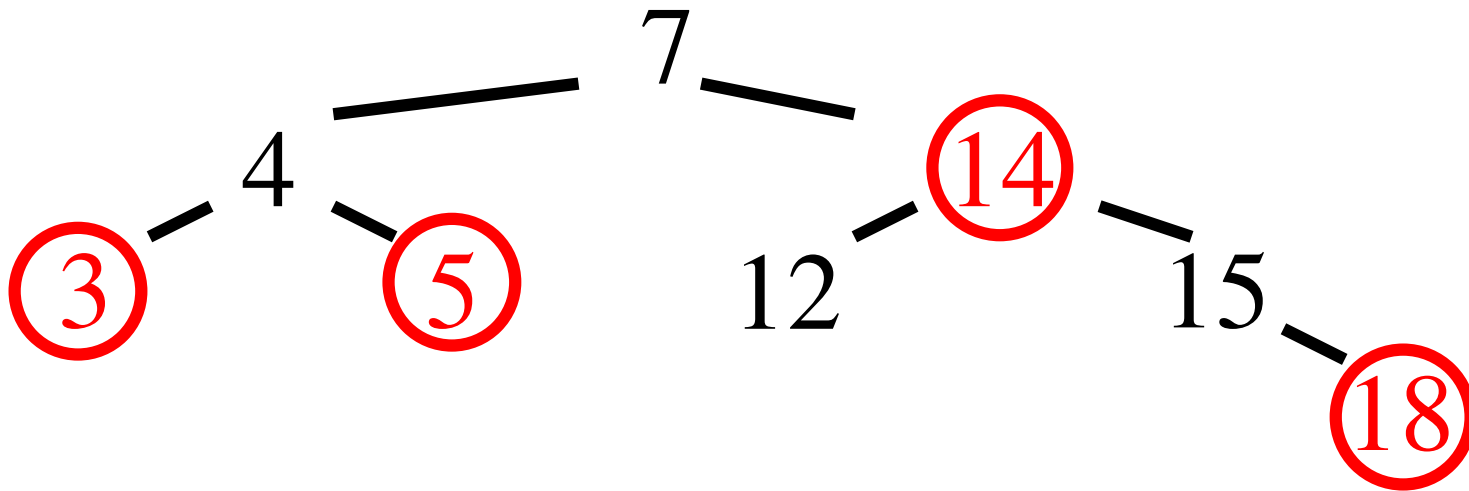
- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree

# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree

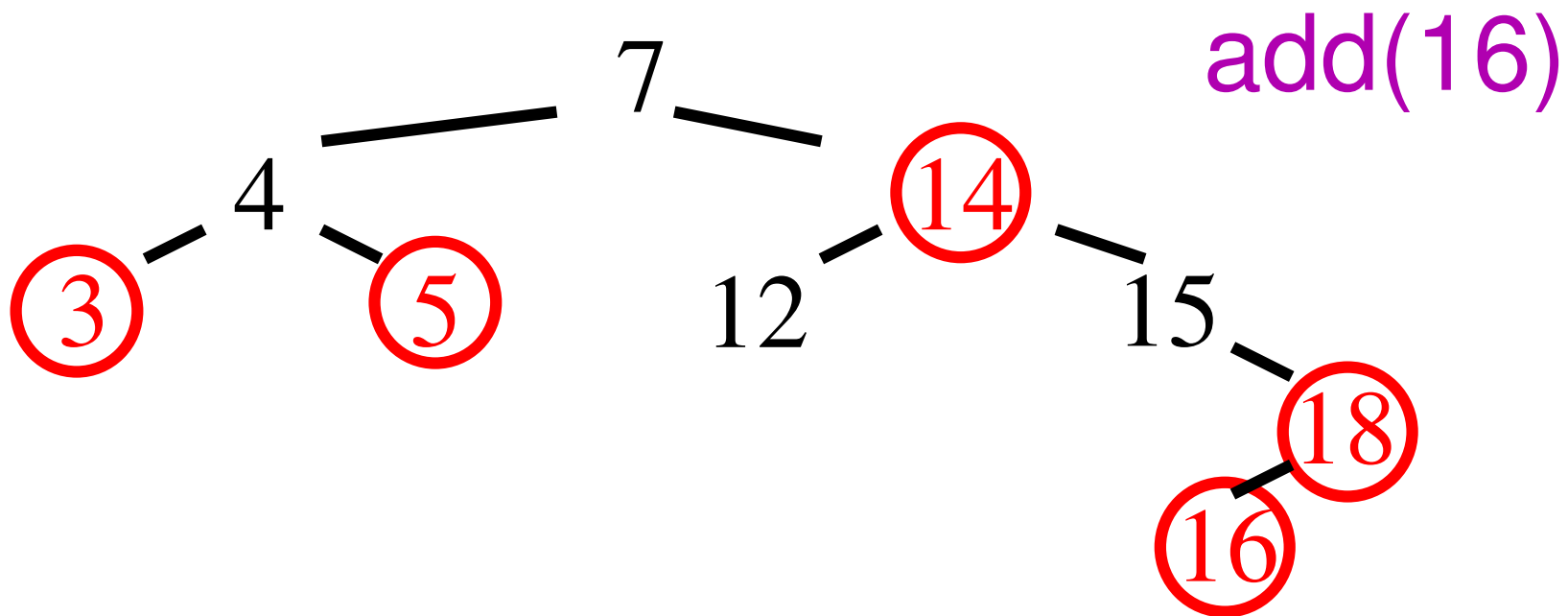
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



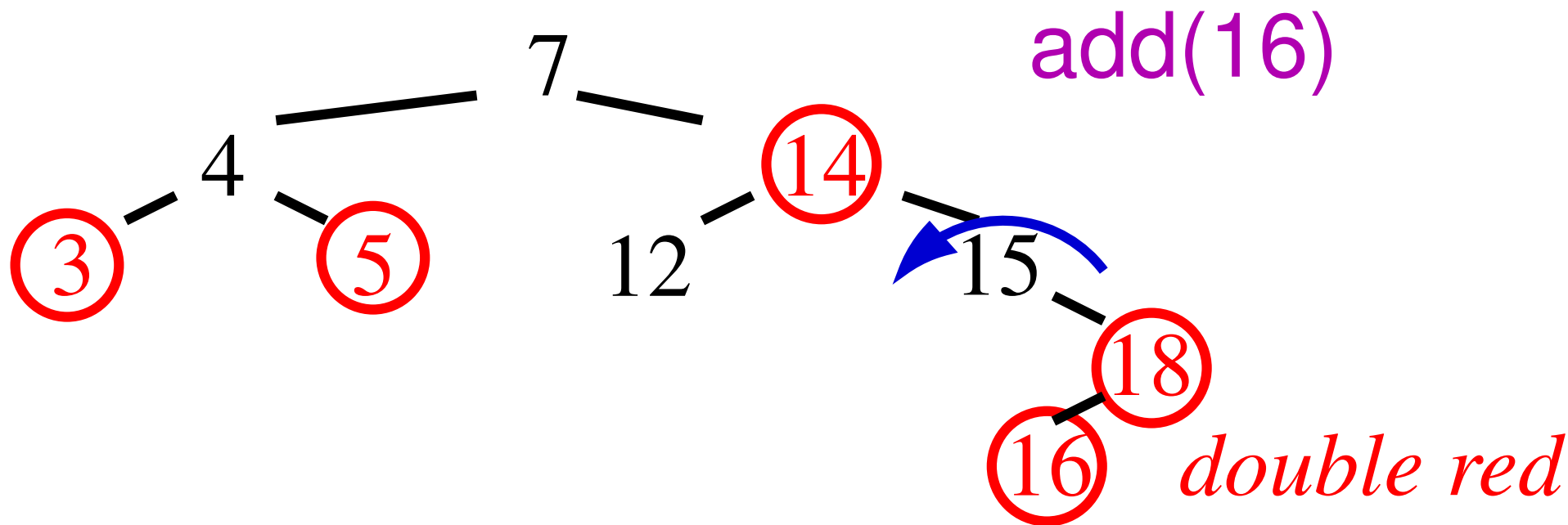
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



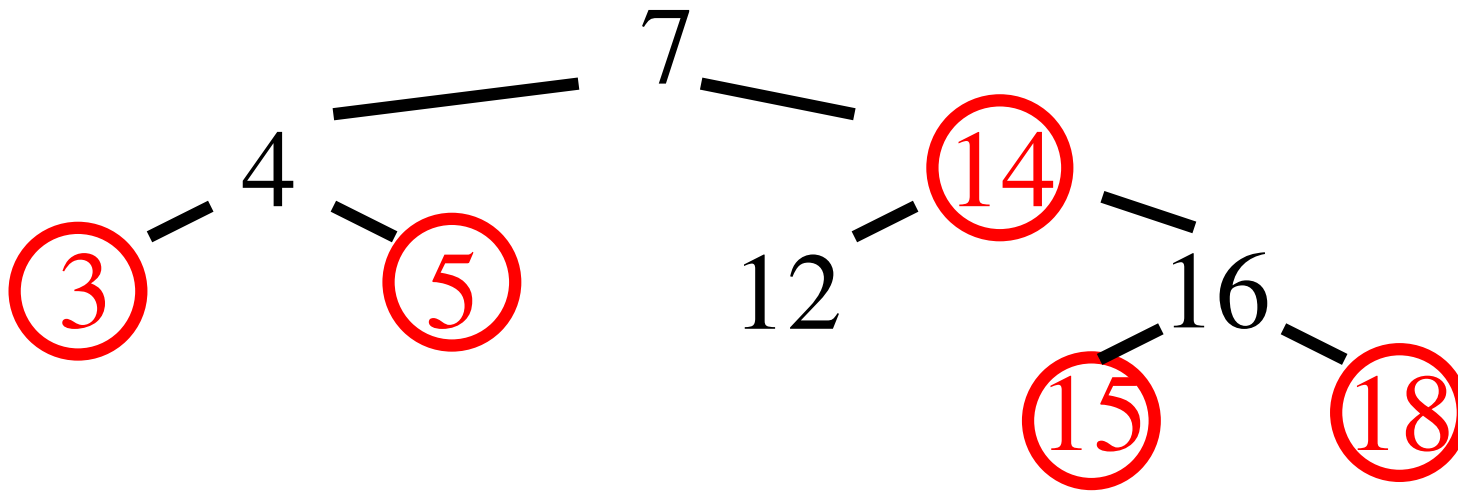
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



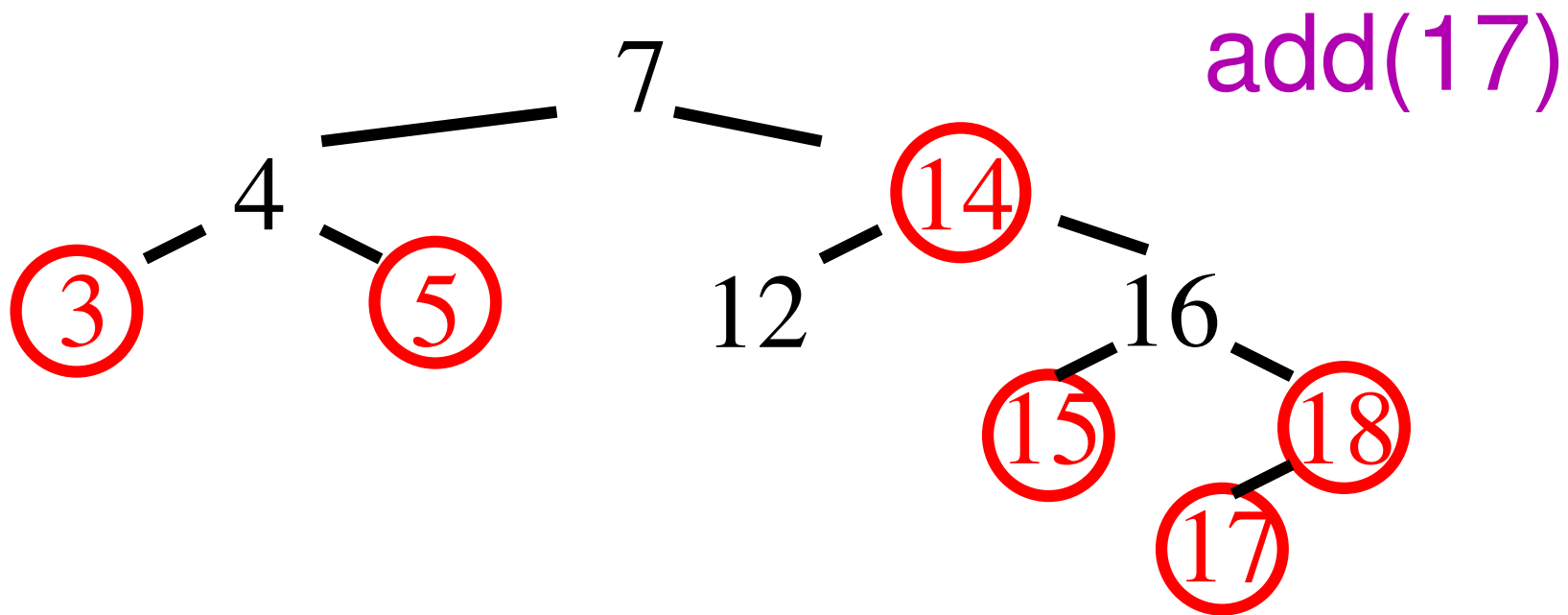
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



# Restructuring

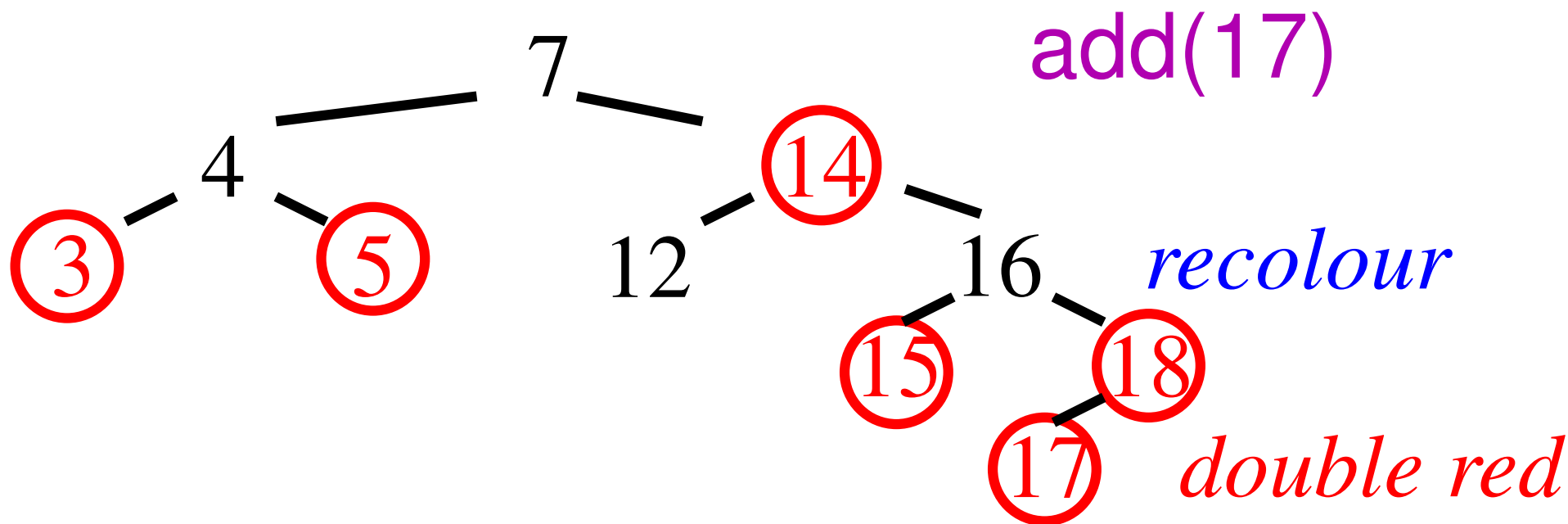
- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree





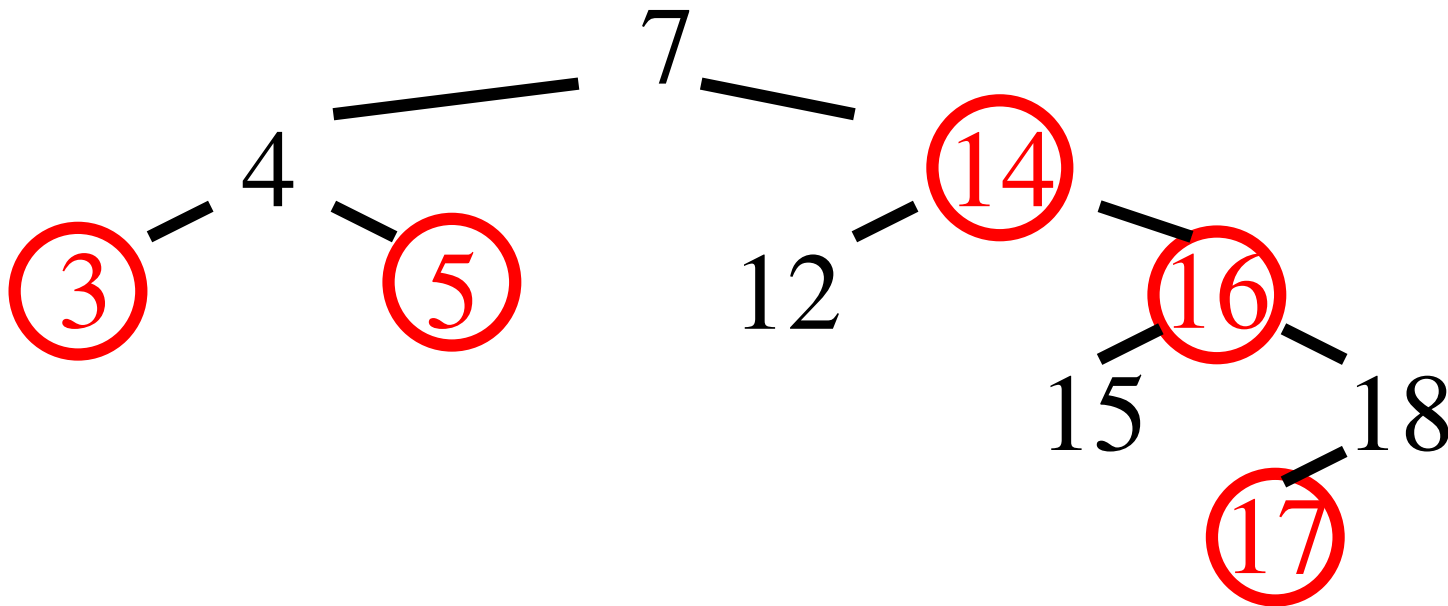
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



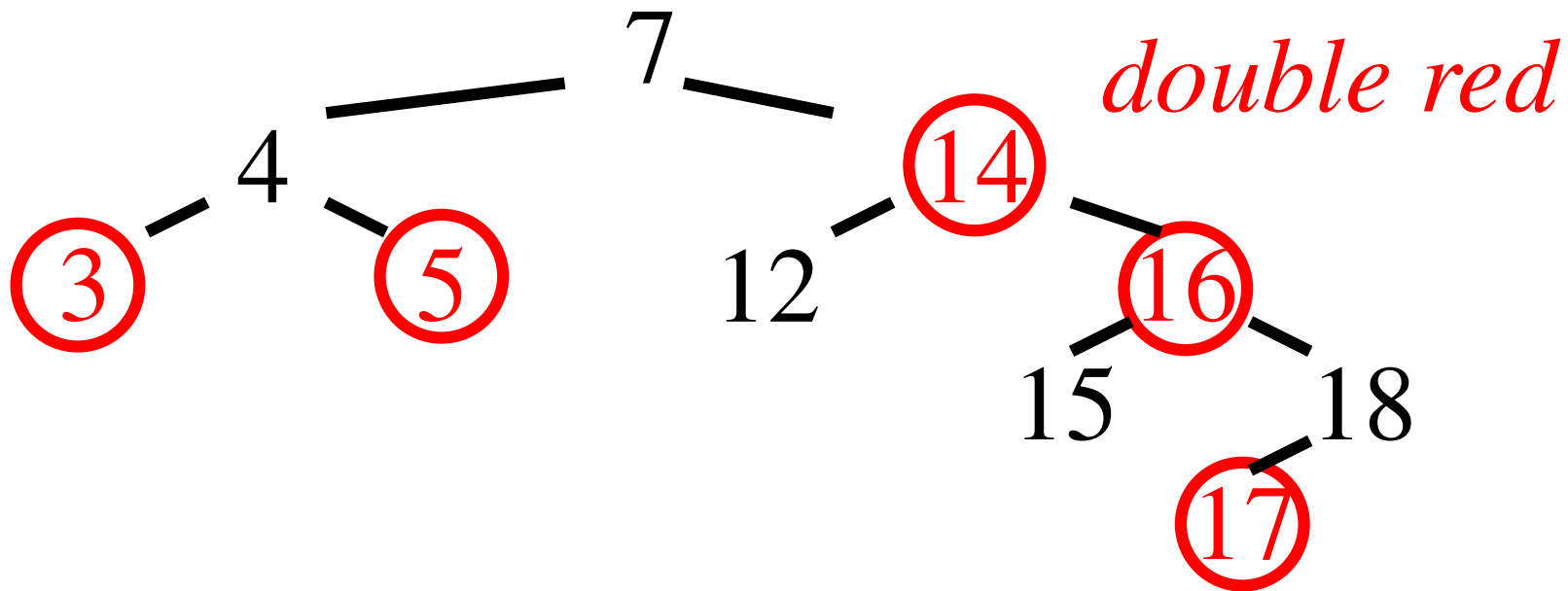
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



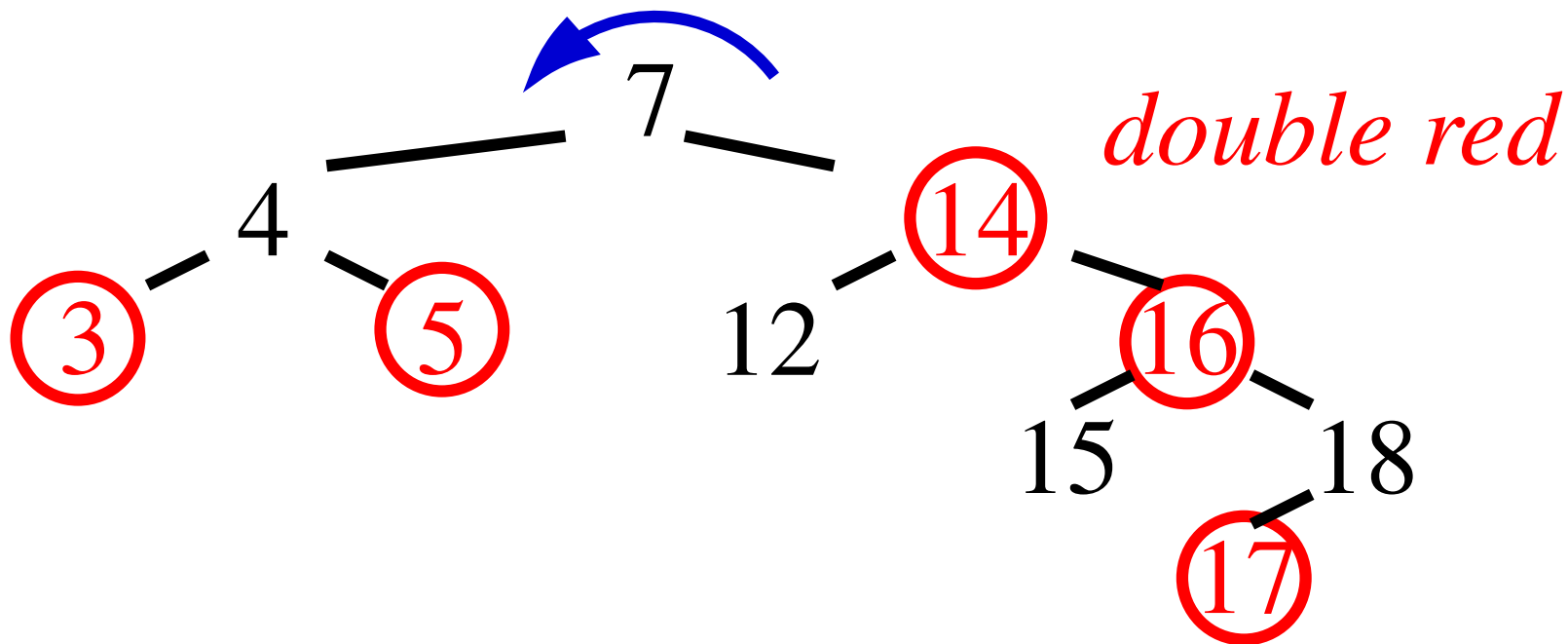
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



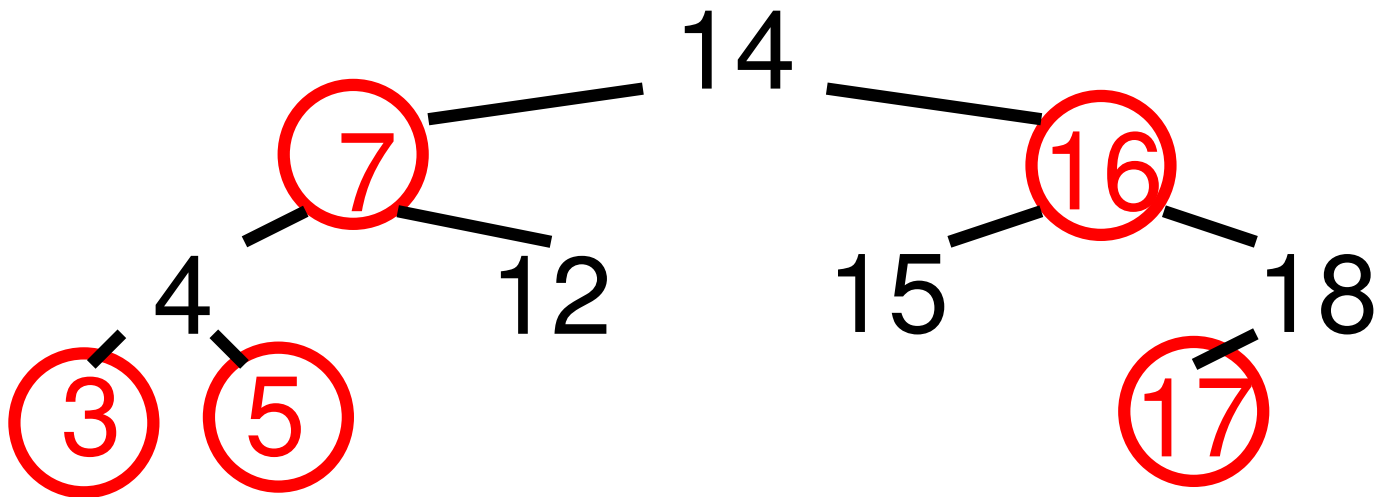
# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



# Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



# Performance of Red-Black Trees

- Red-black trees are slightly more complicated to code than AVL trees
- Red-black trees tend to be slightly less compact than AVL trees
- However, insertion and deletion run slightly quicker
- Both Java Collection classes and C++ STL use red-black trees

# Performance of Red-Black Trees

- Red-black trees are slightly more complicated to code than AVL trees
- Red-black trees tend to be slightly less compact than AVL trees
- However, insertion and deletion run slightly quicker
- Both Java Collection classes and C++ STL use red-black trees

# Performance of Red-Black Trees

- Red-black trees are slightly more complicated to code than AVL trees
- Red-black trees tend to be slightly less compact than AVL trees
- However, insertion and deletion run slightly quicker
- Both Java Collection classes and C++ STL use red-black trees



# Performance of Red-Black Trees

- Red-black trees are slightly more complicated to code than AVL trees
- Red-black trees tend to be slightly less compact than AVL trees
- However, insertion and deletion run slightly quicker
- Both Java Collection classes and C++ STL use red-black trees

# Set

- The standard template library (STL) has a class `std::set<T>`
- It also has a `std::unordered_set<T>` class (which uses a hash table covered later)
- As well as `std::multiset<T>` that implements a multiset (i.e. a set, but with repetitions)
- Using sets you can also implement **maps**

# Set

- The standard template library (STL) has a class `std::set<T>`
- It also has a `std::unordered_set<T>` class (which uses a hash table covered later)
- As well as `std::multiset<T>` that implements a multiset (i.e. a set, but with repetitions)
- Using sets you can also implement **maps**

# Set

- The standard template library (STL) has a class `std::set<T>`
- It also has a `std::unordered_set<T>` class (which uses a hash table covered later)
- As well as `std::multiset<T>` that implements a multiset (i.e. a set, but with repetitions)
- Using sets you can also implement **maps**

# Set

- The standard template library (STL) has a class `std::set<T>`
- It also has a `std::unordered_set<T>` class (which uses a hash table covered later)
- As well as `std::multiset<T>` that implements a multiset (i.e. a set, but with repetitions)
- Using sets you can also implement **maps**

# Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std::map<Key, V>` contain key-value pairs `pair<Key, V>`
  - ★ The first element of type `Key` is the **key**
  - ★ The second element of type `V` is the **value**
- Maps work as content addressable arrays

```
map<string, int> students;
student["John_Smith"] = 89;
student["Terry_Jones"] = 98;
cout << students["John_Smith"];
```

# Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std::map<Key, V>` contain key-value pairs `pair<Key, V>`
  - ★ The first element of type `Key` is the **key**
  - ★ The second element of type `V` is the **value**
- Maps work as content addressable arrays

```
map<string, int> students;  
student["John_Smith"] = 89;  
student["Terry_Jones"] = 98;  
cout << students["John_Smith"];
```

# Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std::map<Key, V>` contain key-value pairs `pair<Key, V>`
  - ★ The first element of type `Key` is the **key**
  - ★ The second element of type `V` is the **value**
- Maps work as content addressable arrays

```
map<string, int> students;
student["John_Smith"] = 89;
student["Terry_Jones"] = 98;
cout << students["John_Smith"];
```



# Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std::map<Key, V>` contain key-value pairs `pair<Key, V>`
  - ★ The first element of type `Key` is the **key**
  - ★ The second element of type `V` is the **value**
- Maps work as content addressable arrays

```
map<string, int> students;  
student["John_Smith"] = 89;  
student["Terry_Jones"] = 98;  
cout << students["John_Smith"];
```

# Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std::map<Key, V>` contain key-value pairs `pair<Key, V>`
  - ★ The first element of type `Key` is the **key**
  - ★ The second element of type `V` is the **value**
- Maps work as content addressable arrays

```
map<string, int> students;  
student["John_Smith"] = 89;  
student["Terry_Jones"] = 98;  
cout << students["John_Smith"];
```

# Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std::map<Key, V>` contain key-value pairs `pair<Key, V>`
  - ★ The first element of type `Key` is the **key**
  - ★ The second element of type `V` is the **value**
- Maps work as content addressable arrays

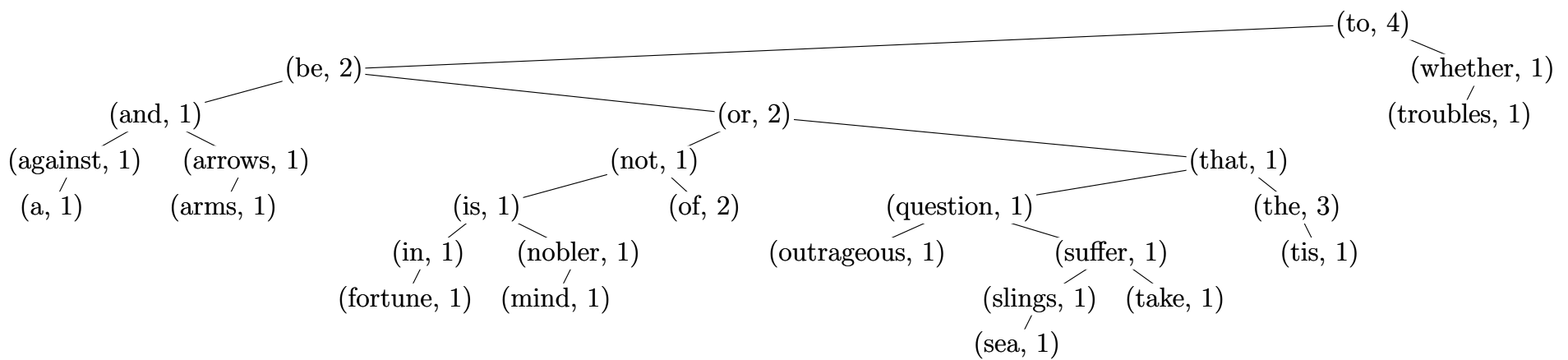
```
map<string, int> students;  
student["John_Smith"] = 89;  
student["Terry_Jones"] = 98;  
cout << students["John_Smith"];
```

# Implementing a Map

- Maps can be implemented using a set by making each node hold a `pair<K, V>` objects

```
class pair<K, V>
{
    public:
    K first;
    V second;
}
```

- We can count words using the key for words and value to count

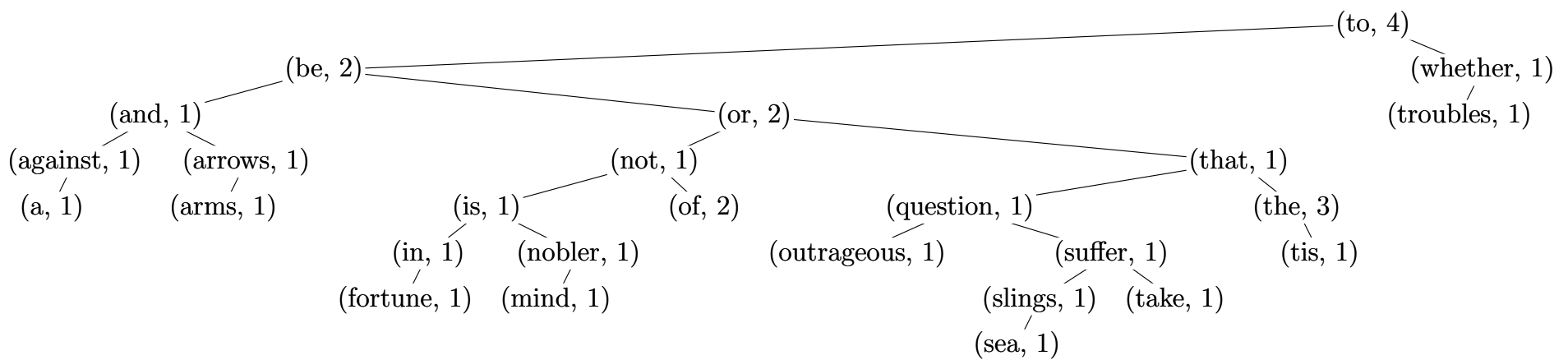


# Implementing a Map

- Maps can be implemented using a set by making each node hold a `pair<K, V>` objects

```
class pair<K, V>
{
    public:
    K first;
    V second;
}
```

- We can count words using the key for words and value to count

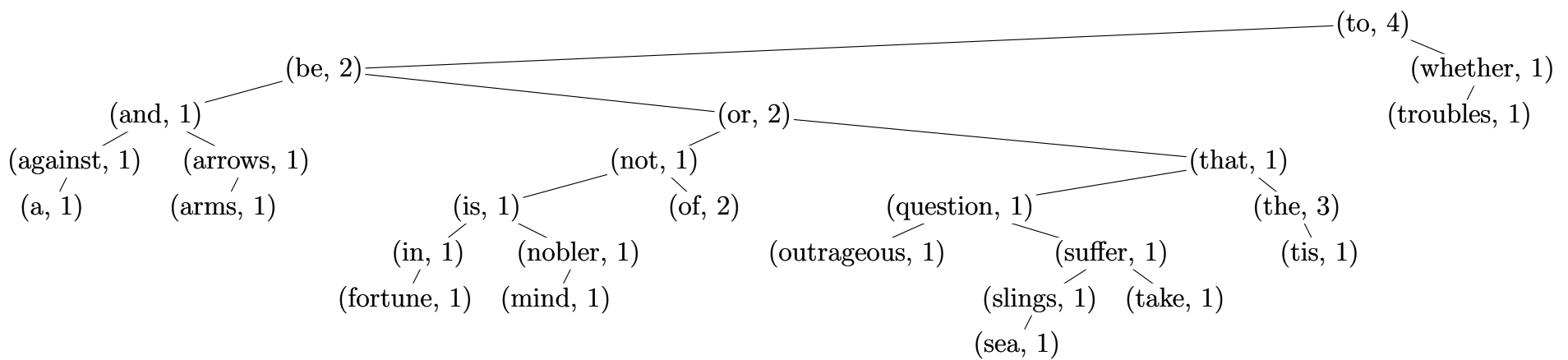


# Implementing a Map

- Maps can be implemented using a set by making each node hold a `pair<K, V>` objects

```
class pair<K, V>
{
    public:
    K first;
    V second;
}
```

- We can count words using the key for words and value to count



# Lessons

- Binary search trees are very efficient (order  $\log(n)$  insertion, deletion and search) provided they are balanced
- Balanced trees are achieved by performing rotations
- There are different strategies for deciding when to rotate including
  - ★ AVL trees
  - ★ Red-black trees
- Binary trees are used for implementing **sets** and **maps**

# Lessons

- Binary search trees are very efficient (order  $\log(n)$  insertion, deletion and search) provided they are balanced
- Balanced trees are achieved by performing rotations
- There are different strategies for deciding when to rotate including
  - ★ AVL trees
  - ★ Red-black trees
- Binary trees are used for implementing **sets** and **maps**



# Lessons

- Binary search trees are very efficient (order  $\log(n)$  insertion, deletion and search) provided they are balanced
- Balanced trees are achieved by performing rotations
- There are different strategies for deciding when to rotate including
  - ★ AVL trees
  - ★ Red-black trees
- Binary trees are used for implementing **sets** and **maps**

# Lessons

- Binary search trees are very efficient (order  $\log(n)$  insertion, deletion and search) provided they are balanced
- Balanced trees are achieved by performing rotations
- There are different strategies for deciding when to rotate including
  - ★ AVL trees
  - ★ Red-black trees
- Binary trees are used for implementing **sets** and **maps**