Data Structures and Algorithms

Lesson 6: Keep Trees Balanced

AVL trees, red-black trees, TreeSet, TreeMap

Outline

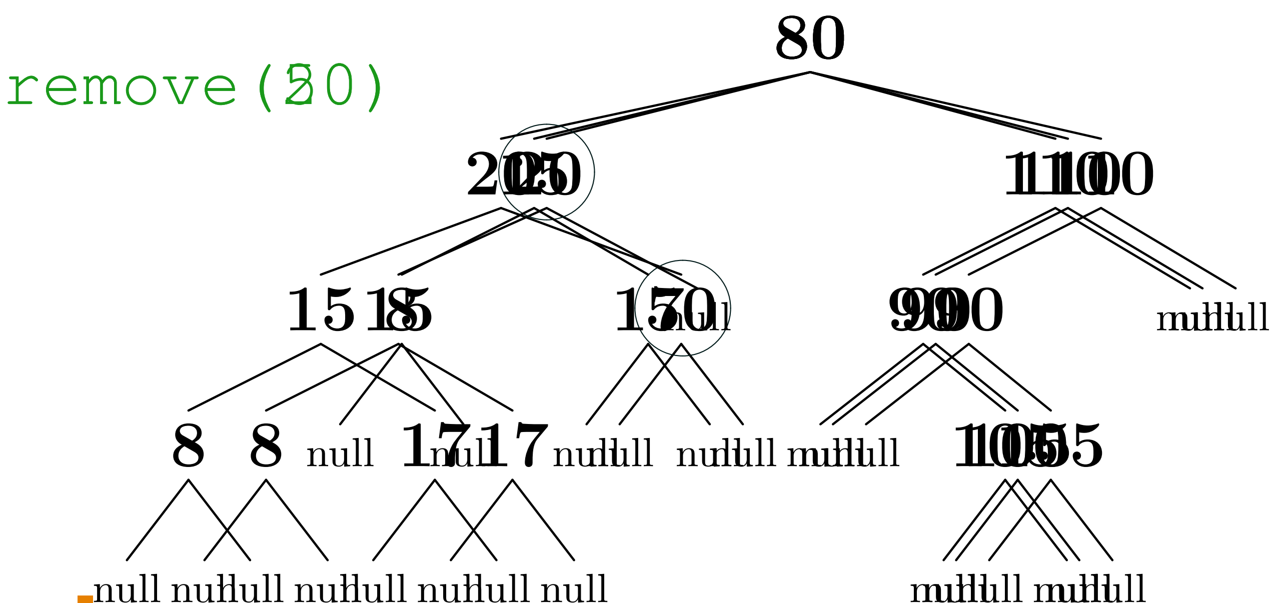
1. **Deletion**
2. Balancing Trees
   - Rotations
3. AVL Trees
4. Red-Black Trees
   - TreeSet
   - TreeMap

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
  - `contains`
  - `add`
  - `successor` (in outline)
- One method we missed was `remove`

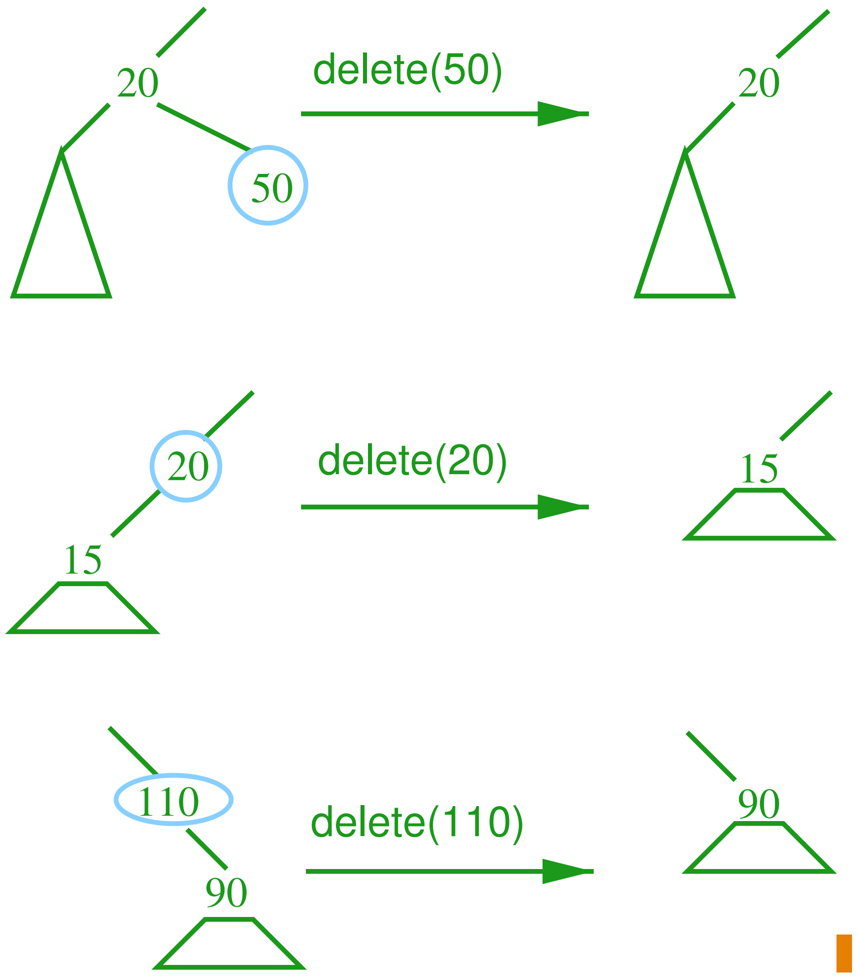
Deletion

- Suppose we want to delete an element from a binary search tree
- It is relatively easy if the element is held in a leaf node (e.g. 50)
- It is not so hard if the node holding the element has one child (e.g. 20)



Code for `remove`

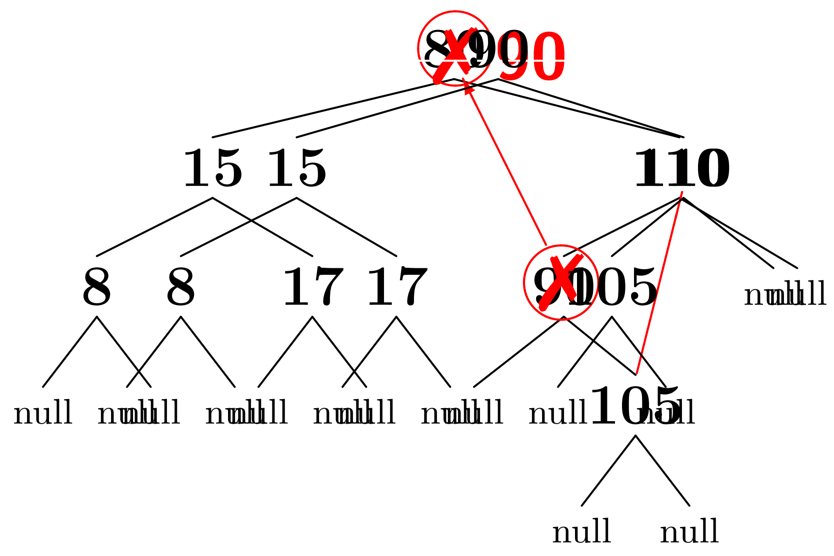
```
if (e.left==null && e.right==null) {
    // leaf node
    if (e == e.parent.left)
      e.parent.left = null;
    else
       e.parent.right = null;
} else if (e.right==null) {
    // no right child
    if (e == e.parent.left)
       e.parent.left = e.left;
    else
       e.parent.right = e.left;
    e.left.parent = e.parent;
} else if (e.left==null) {
    // no left child
    if (e == e.parent.left)
       e.parent.left = e.right;
    else
       e.parent.right = e.right;
    e.right.parent = e.parent;
}
```



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)`

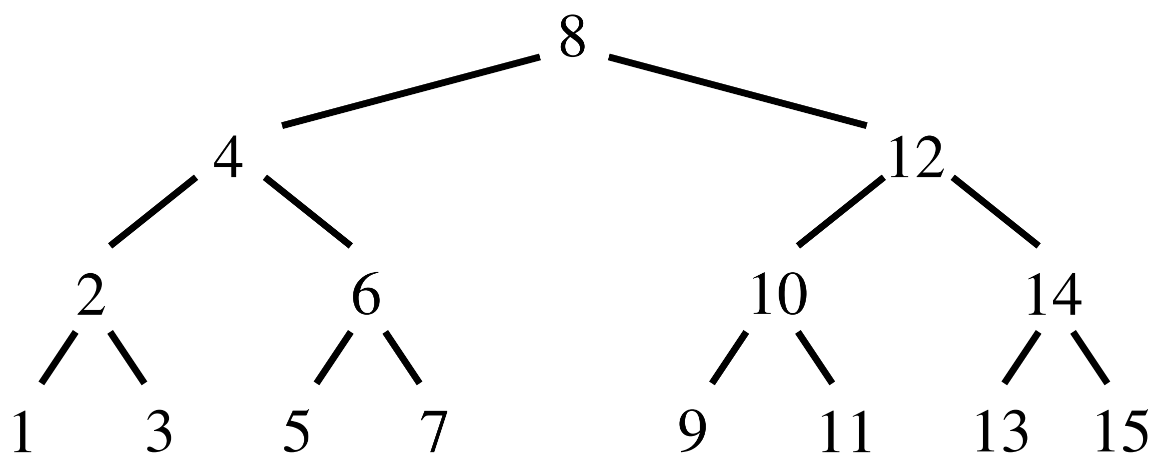


Outline

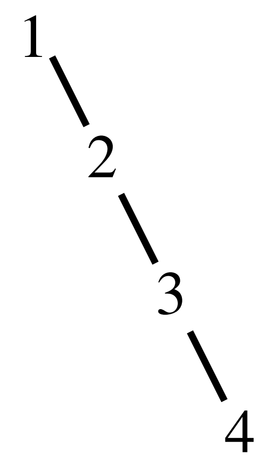
1. Deletion
2. **Balancing Trees**
   - Rotations
3. AVL Trees
4. Red-Black Trees
   - TreeSet
   - TreeMap

Why Balance Trees

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



full tree



sparse tree

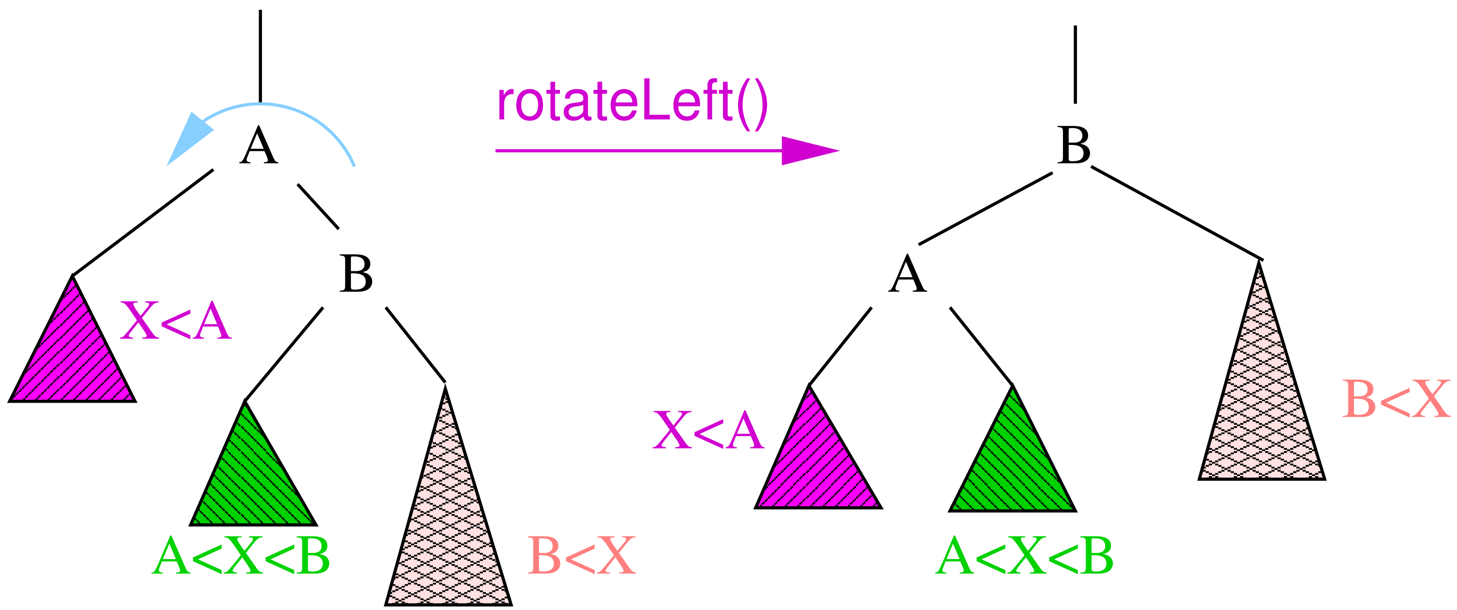
- The shape of the tree depends on the order the elements are added

Tree Depth

- In the best case (a full tree), if the number of elements in the tree is $n = 2^l - 1$ then the maximum depth of a node is $l = \log_2(n+1)$, which is $\Theta(\log(n))$
- In the worst case (when the tree is effectively a linked list), the maximum depth of a node is n, which is $\Theta(n)$
- It turns out for random sequences the average depth of a node is $O(\log(n))$
- Unfortunately, the worst case happens when the elements are added $in\ order$ (not a rare event)

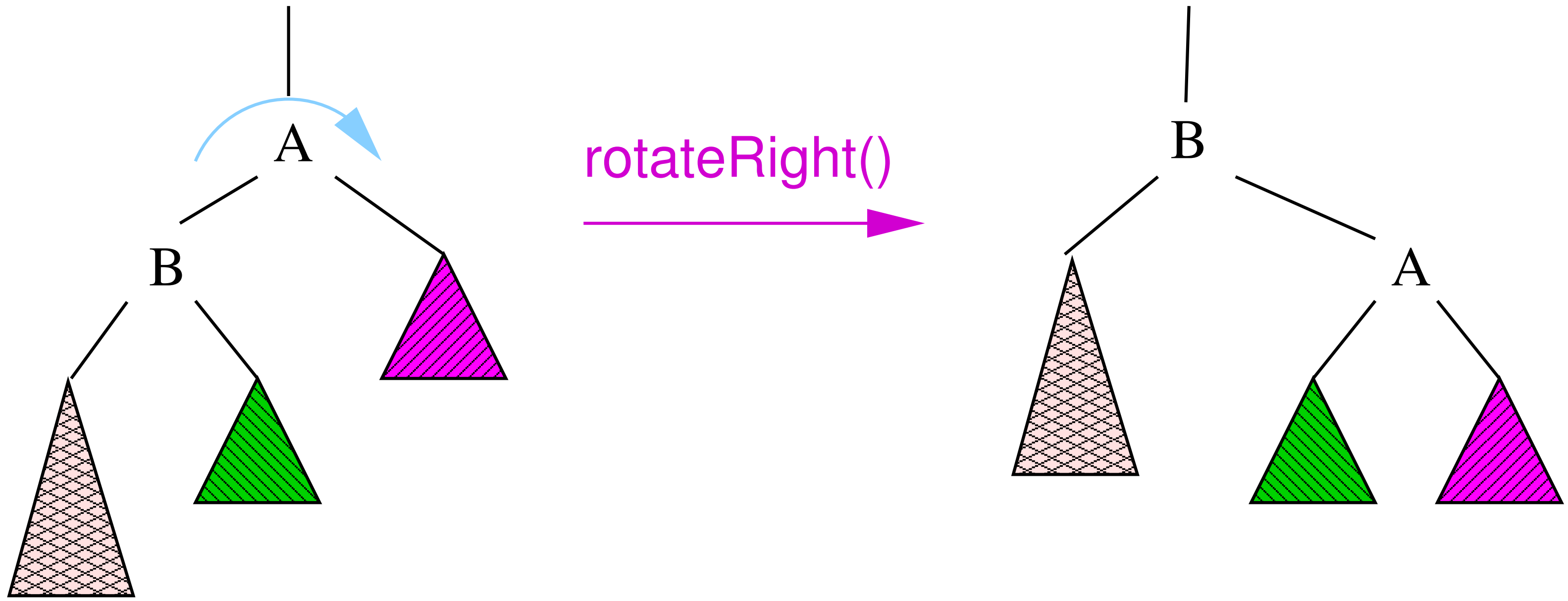
Rotations

- 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



Types of Rotations

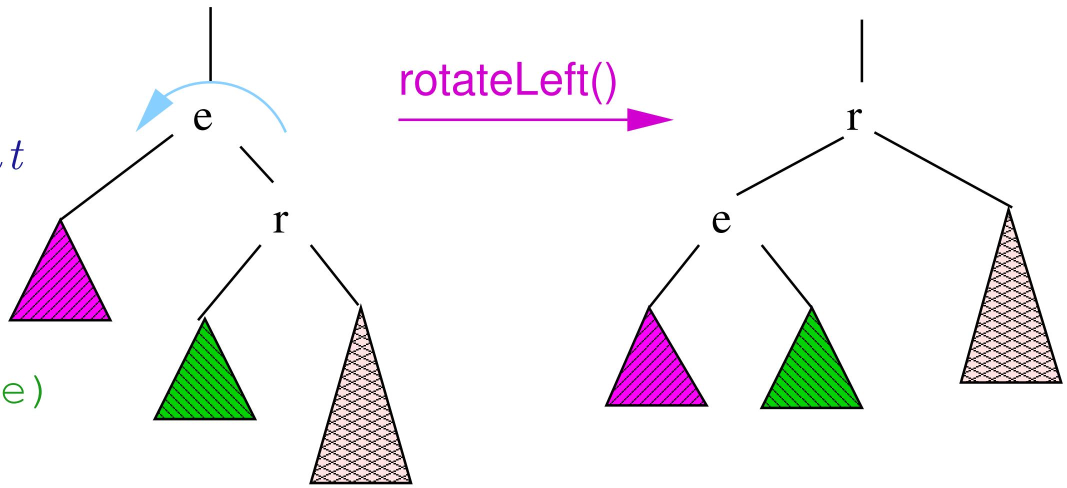
- We can get by with 4 types of rotations
  - Left rotation (as above)
  - Right rotation (symmetric to above)



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

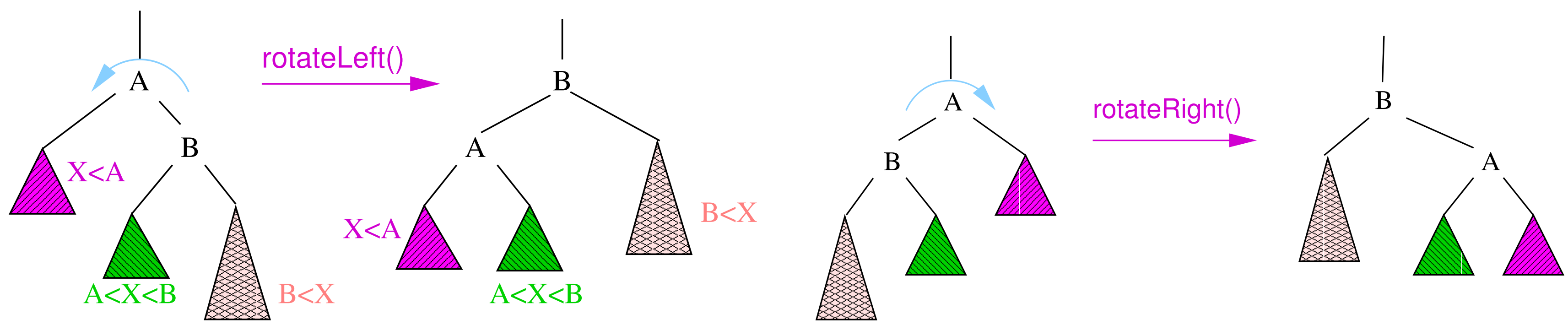
Coding Rotations

```
void rotateLeft(Node<T> e){
  // r = right node
  Node<T> r = e.right;

  // link e with its new right node
  e.right = r.left;
  if (r.left != null)
     r.left.parent = e;

  // link r to its new parent
  r.parent = e.parent;
  if (e.parent == null)
     root = r;
  else if (e.parent.left == e)
     e.parent.left = r;
  else
     e.parent.right = r;

  // fix links between r and e
  r.left = e;
  e.parent = r;
}
```



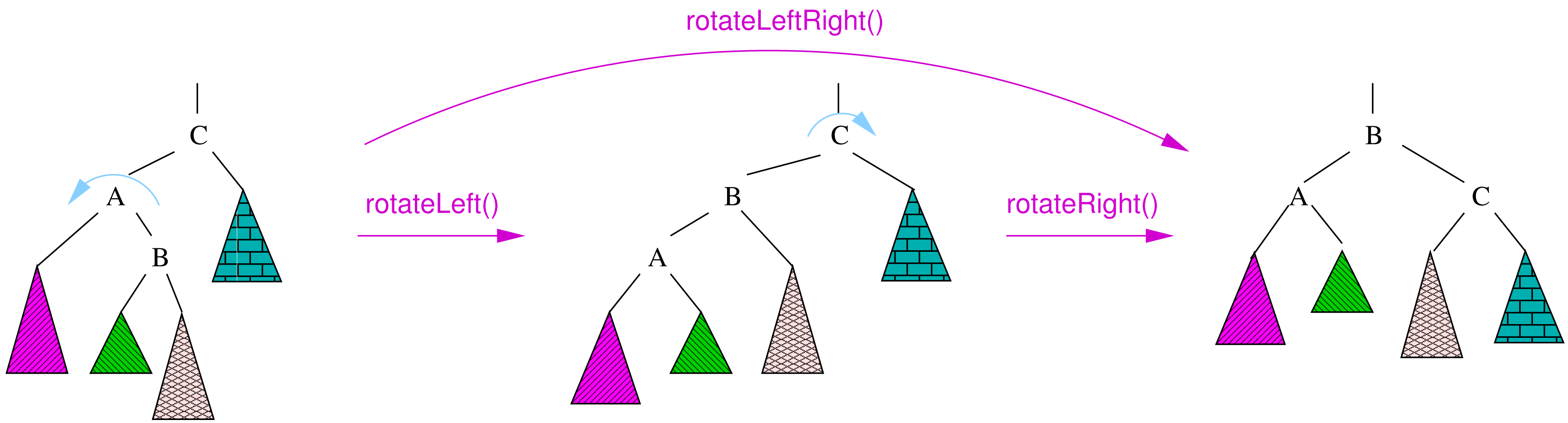
When Single Rotations Work

- 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);
```

Outline

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

Balancing Trees

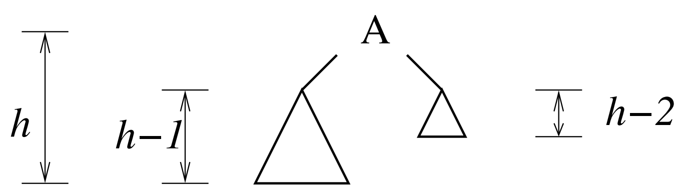
- There are different strategies for using rotations for balancing trees
- The three most popular are
  - AVL trees
  - Red-black trees
  - Splay trees
- They differ in the criteria they use for doing rotations

AVL Trees

- AVL trees were invented in 1962 by two Russian mathematicians Adelson-Velski and Landis
- An AVL tree is a binary search tree in which
  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

- We want to see how full an AVL tree has to be, at the minimum
- Let $m(h)$ be the minimum number of nodes in an AVL tree of height h
- This has to be made up of a root and two subtrees: one of height $h-1$; and, in the worst case, one of height $h-2$



- Thus, the least number of nodes in an AVL 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 $1, 2, 4, 7, 12, \cdots$
- Compare this with Fibonacci $f(h) = f(h-1) + f(h-2)$, with $f(1) = f(2) = 1$
- This gives us a sequence $1, 1, 2, 3, 5, 8, 13, \cdots$
- It looks like $m(h) = f(h+2) - 1$
  - this can be proved by induction
- This shows that $m(h)$ grows exponentially with h

Proof of Logarithmic Depth

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

$$m(1) = 1 \geq (3/2)^0 = 1, \quad m(2) = 2 \geq (3/2)^1 = 3/2 \ \checkmark$$

$$m(h) = m(h-1) + m(h-2) + 1 \geq \left(\tfrac{3}{2}\right)^{h-2} + \left(\tfrac{3}{2}\right)^{h-3} + 1 = \left(\tfrac{3}{2}\right)^{h-3}\left(\tfrac{3}{2} + 1 + \left(\tfrac{3}{2}\right)^{3-h}\right)$$

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

- The number of elements, n, we can store in an AVL tree of height h is $n \geq m(h)$, and thus $n \geq (3/2)^{h-1}$
- Taking logs: $\quad log(n) \geq (h-1)\log(3/2)$
- So $h \leq \frac{log(n)}{\log(3/2)} + 1$
- That is, h is $O(log(n))$

Implementing AVL Trees

- To implement an AVL tree, we include additional information at each node indicating the balance of the subtrees:

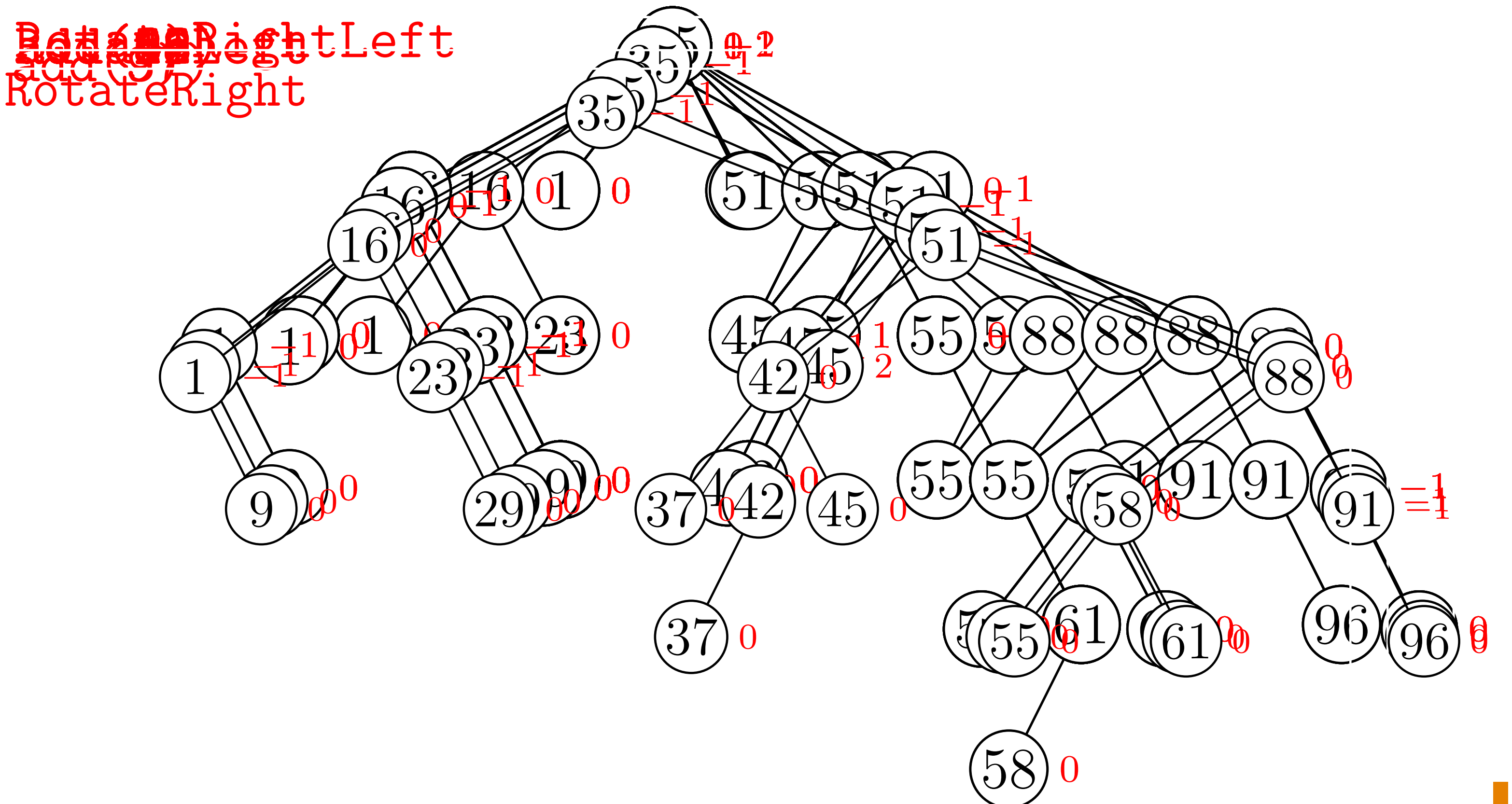
$$\texttt{balanceFactor} = \text{(height of left subtree) - (height of right subtree)}$$

- So for an AVL tree:

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

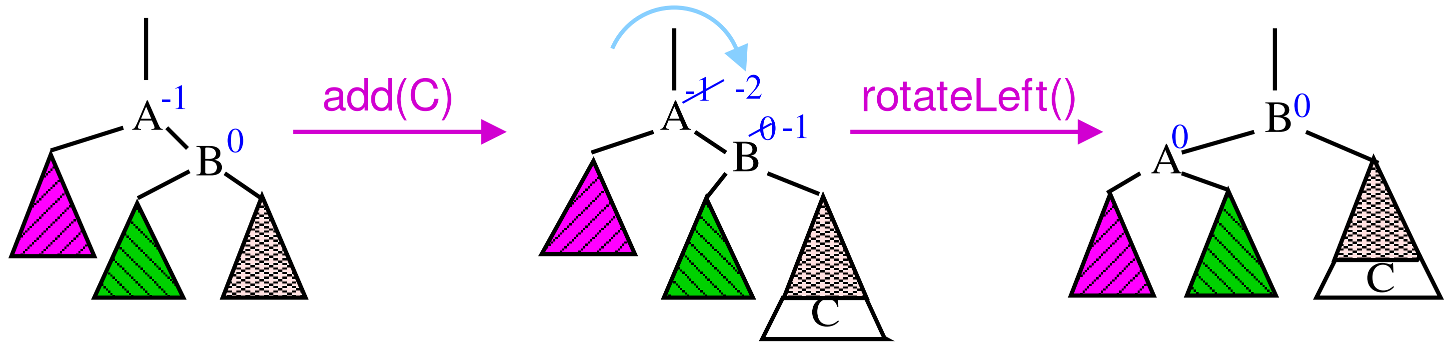
Implementing AVL Trees

- update `balanceFactor` after each `add`/`remove` operation
- rebalance tree using rotations if needed



Balancing AVL Trees

- When `adding` an element to an AVL tree
  - Find the location where it is to be inserted, and insert
  - Iterate up through the parents re-adjusting the `balanceFactor`
  - **If** the balance factor exceeds ± 1 `then` re-balance the tree and stop – why can we stop?
  - `else if` the balance factor goes to zero **then** stop – why can we stop?


A
-1
B
0
add(C)
A
-1
-2
B
0
-1
C
rotateLeft()
B
0
A
0
C

AVL Deletions

- AVL deletions are similar to AVL insertions
  - need to update balance factors/rebalance tree
- 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, $\Theta(\log(n))$
  - height of an AVL tree is $\Theta(\log(n))$
  - so searching is at worst $\Theta(\log(n))$
  - insertion without balancing is $\Theta(\log(n))$, balancing takes an additional $\Theta(\log(n))$ steps in the worst case
  - deletion without balancing is $\Theta(\log(n))$ at worst (need to find the node first), balancing takes an additional $\Theta(\log(n))$ steps in the worst case
- The height of an average AVL tree is $\approx 1.44 \log_2(n)$
- The height of an average binary search tree is $\approx 2.1 \log_2(n)$
- Despite being more compact, insertion is slightly slower in AVL trees than in BSTs without balancing
- Search is, of course, quicker!

Outline

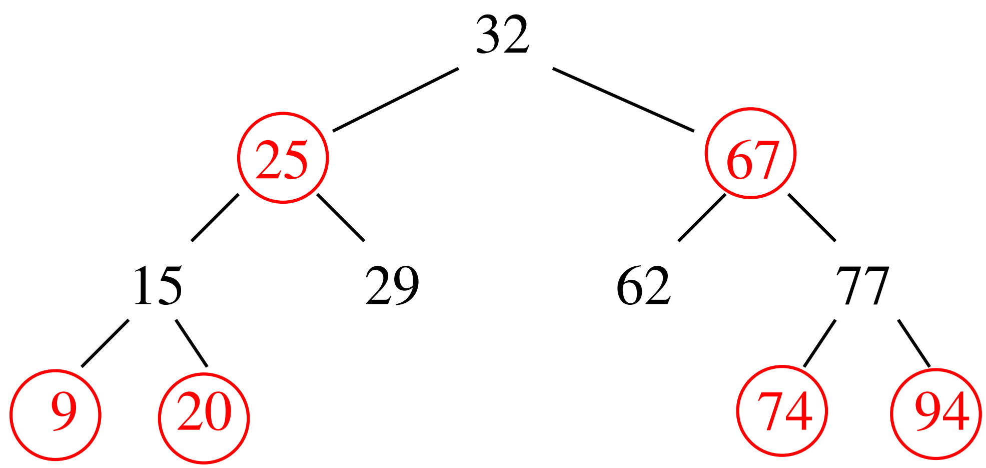
1. Deleting
2. Balancing Trees
   - Rotations
3. AVL Trees
4. **Red-Black Trees**
   - TreeSet
   - TreeMap

Red-Black Trees

- Red-black trees are another strategy for balancing trees
- Nodes are either red or black
- Two rules ensure that no path from the root to a leaf, or to a node with one child, is more than twice as long as another:

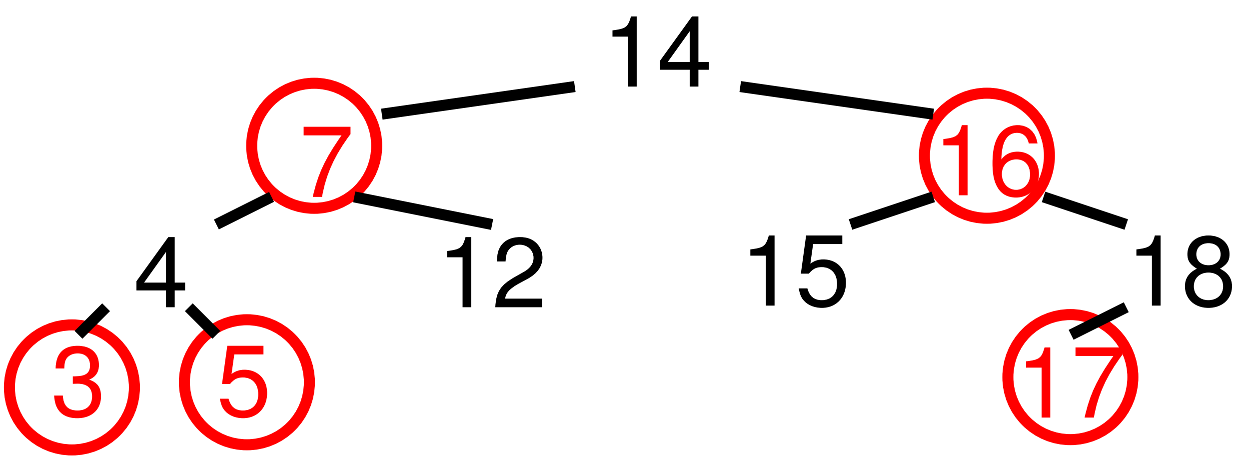
  Red Rule: the children of a red node must be black

  Black Rule: the number of black nodes must be the same in all paths from the root to nodes 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
  - relabel if the “uncle” exists and is also red
  - rotate if the “uncle” does not exist or it is black



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

TreeSet

- The Java Collection class has a `TreeSet` class implemented using a red-black tree
- It also has a `HashSet` class (which we cover later)
- The `TreeSet` class iterates over the elements in order (unlike the `HashSet` class)
- These are the classes to use if you want a collection of objects with no repetitions

Maps

- One major abstract data type (ADT) we have not yet seen an implementation for is maps

- The Java `Map` interface contains element pairs `Map<K,V>`
  - The first element of type `K` is the key
  - The second element of type `V` is the value

- The Java class `TreeMap<K,V>` implements this interface, using Red-Black trees

- Maps work as content addressable arrays

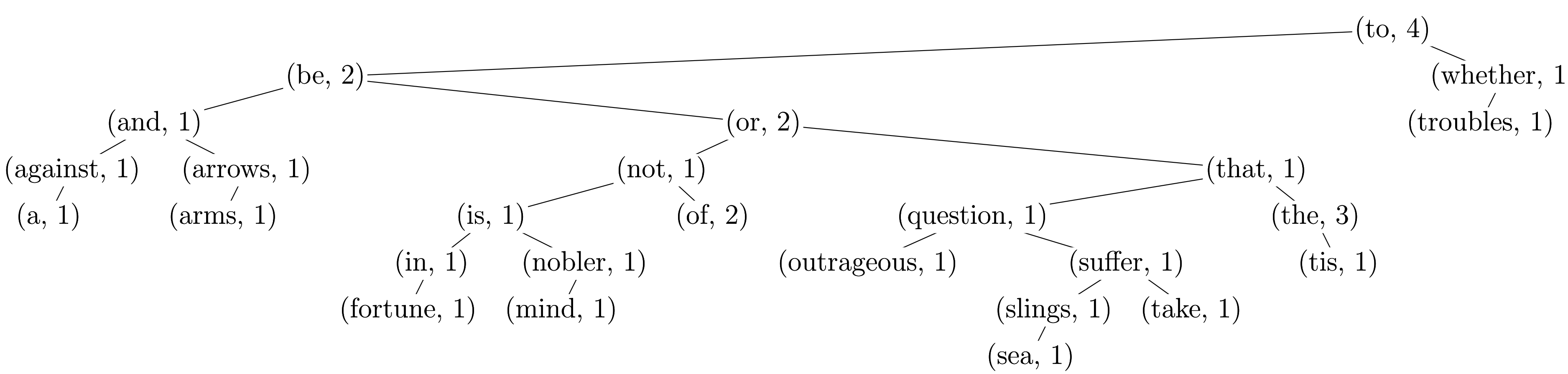
```
Map<String,Integer> students = new TreeMap<String,Integer>();
student.put("John Smith", 89);
student.put("Terry Jones", 98);
System.out.println(students.get("John Smith"));
```

Implementing a TreeMap

- We can implement Map using a TreeSet by making each node hold a `Entry<K,V>` object

```
public class Entry<K,V> implements Comparable
{
    private K key;
    private V value;
}
```

- We can count occurrences of words using the key for words and the 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 search trees are used for implementing sets and maps