# Introduction to Algorithms

Treaps

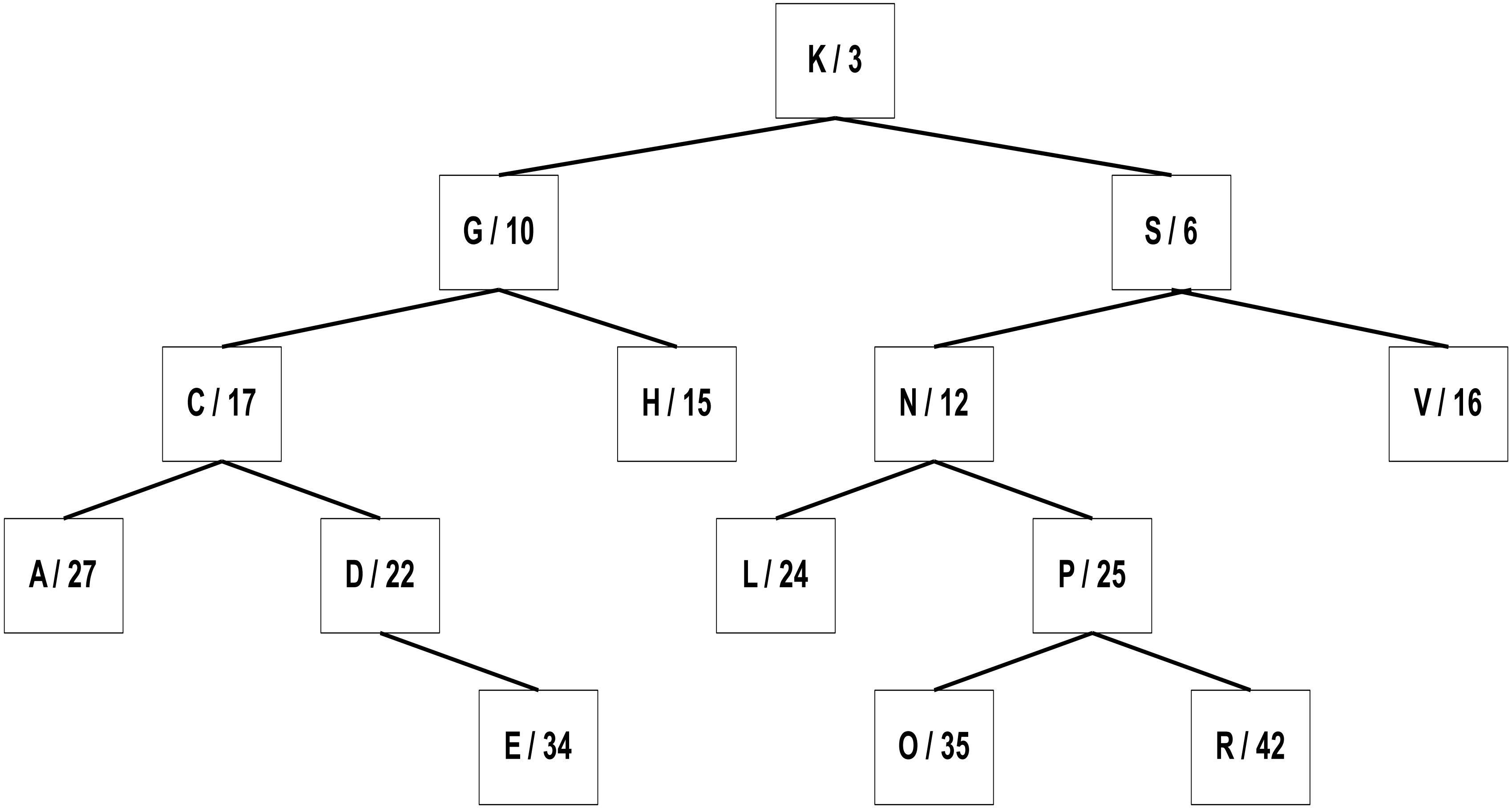
# Treaps

- First introduced in 1989 by Aragon and Seidel
  - http://people.ischool.berkeley.edu/~aragon/pubs/rst89.pdf
- Randomized Binary Search Tree
- A combination of a binary search tree and a min binary heap (a "tree - heap")
- Each node contains
  - Data / Key (comparable)
  - Priority (random integer)
  - Left, right child reference
- Nodes are in BST order by data/key **and** in min binary heap order by priority

# Why Treaps?

- High probability of O( lg N ) performance for any set of input
  - Priorities are chosen randomly when node is inserted
- Code is less complex than Red-Black trees
  - Perhaps the simplest of BSTs that try to improve performance
  - Priorities don’t have to be updated to keep tree balanced (unlike colors in RB Tree)
  - Non-recursive implementation possible

# A treap example



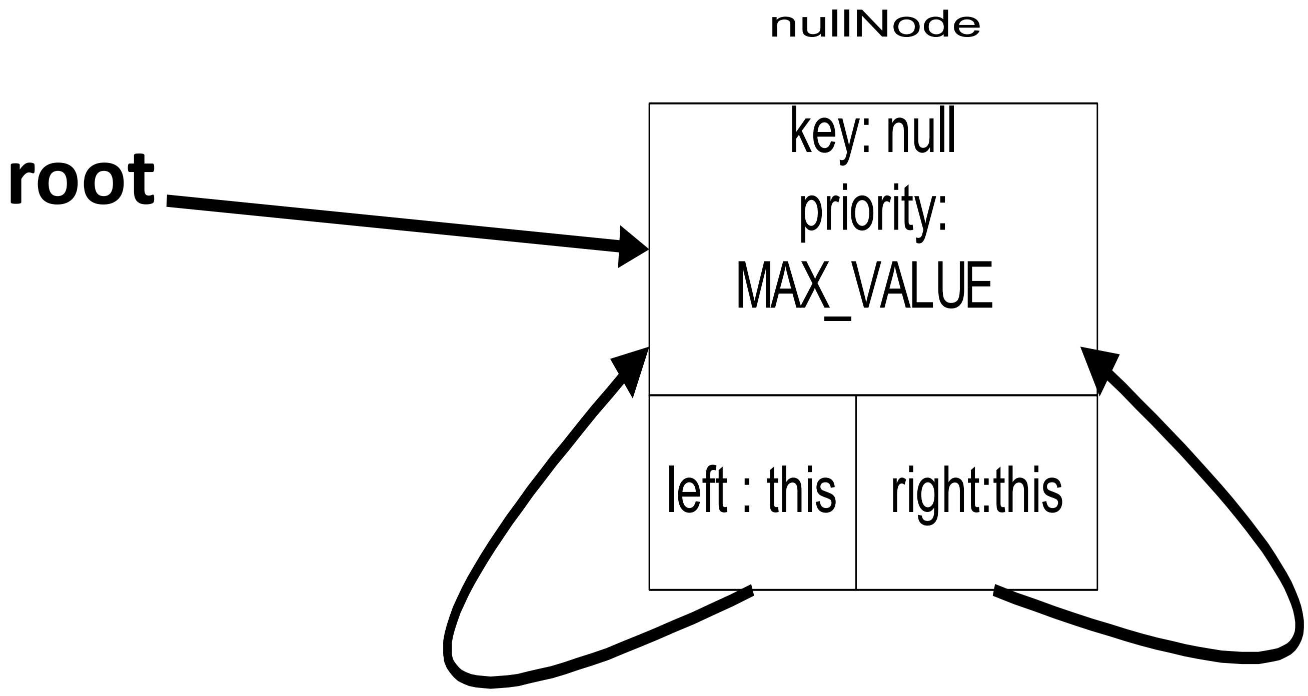
# Treap Node

```
import java.util.Random;

public class Treap<AnyType extends Comparable<? super AnyType>> {
 private static class TreapNode< AnyType >
 {
    AnyType                  element;    // the data in the node
    TreapNode<AnyType>       right;      // right child reference
    TreapNode<AnyType>       left;       // left child reference
    int                      priority;   // for balancing
    private static Random randomObj = new Random( );

    // constructors
    TreapNode( AnyType x )
    {
         this( x, null, null );  // used only by Treap constructor
    }
    TreapNode( AnyType x, TreapNode<AnyType> lt, TreapNode<AnyType> rt)
    {
         element = x;
         left = lt;
         right = rt;
         priority = randomObj.nextInt( );
    }
} ;
```

# Treap Constructor

```
// Treap class data members
private TreapNode< AnyType > root;          // usual root
private TreapNode< AnyType > nullNode;      // replaces null ref

// Default constructor
Treap( )
{
    nullNode = new TreapNode< AnyType >( null );
    nullNode.left = nullNode.right = nullNode;
    nullNode.priority = Integer.MAX_VALUE;

    root = nullNode;
}
```

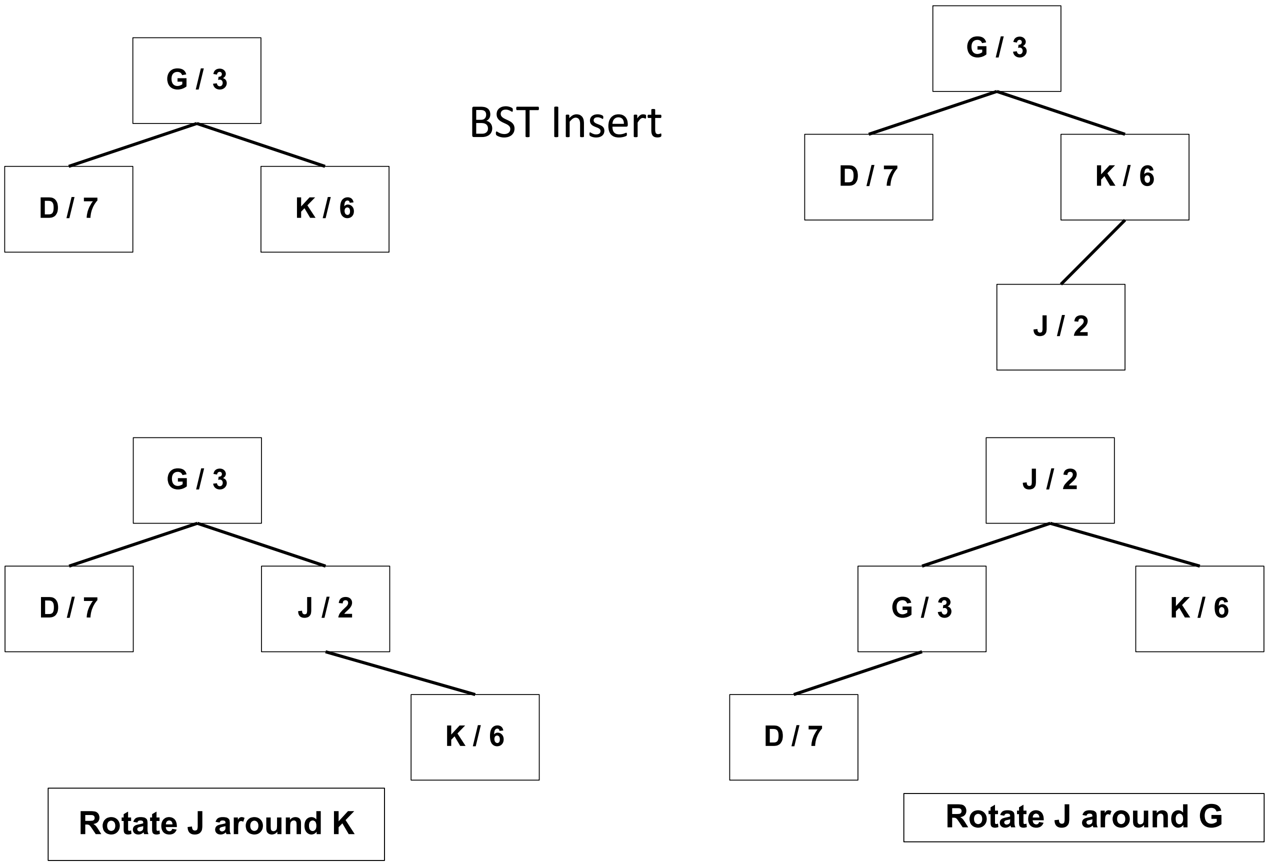
# An Empty Treap



# insert( )

```
public void insert( AnyType x )
   { root = insert( x, root ); }

// recurse down the treap to find where to insert x according to BST order
// rotate with parent on the way back if necessary according to min heap order
private TreapNode< AnyType > insert( AnyType x, TreapNode< AnyType > t )
{
    if ( t == nullNode )
         return new TreapNode<AnyType>( x, nullNode, nullNode );
    int compare = x.compareTo( t.element );
    if ( compare < 0 ) {
         t.left = insert( x, t.left );         // proceed down the treap
         if ( t.left.priority < t.priority )   // rotate coming back up the
treap
             t = rotateWithLeftChild( t );

    } else if ( compare > 0 ) {
         t.right = insert(x, t.right );
         if ( t.right.priority < t.priority )
             t = rotateWithRightChild ( t );

    } // else duplicate, do nothing

    return t;
}
```

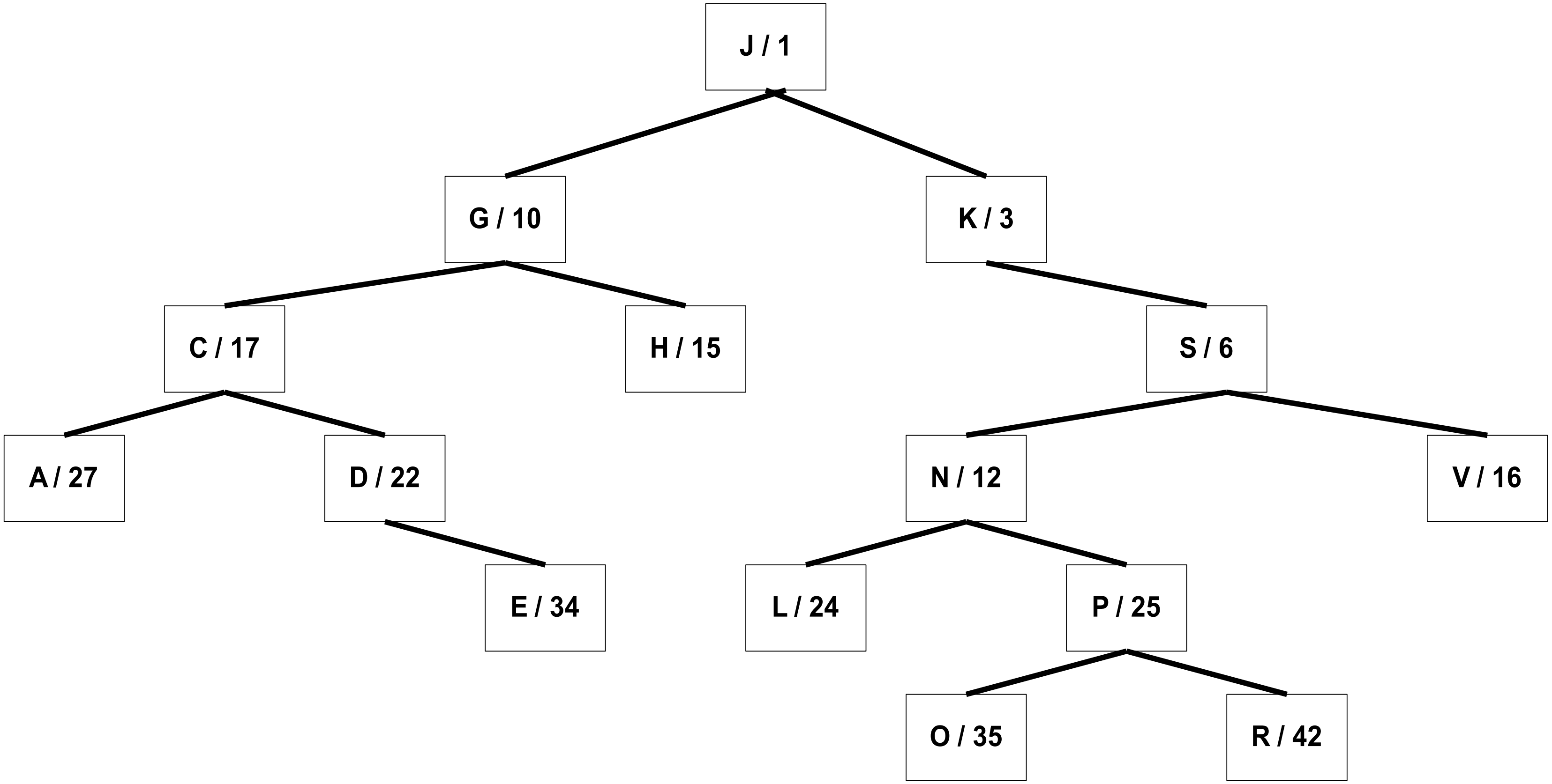
# Insert J with priority 2



# Remove Strategy

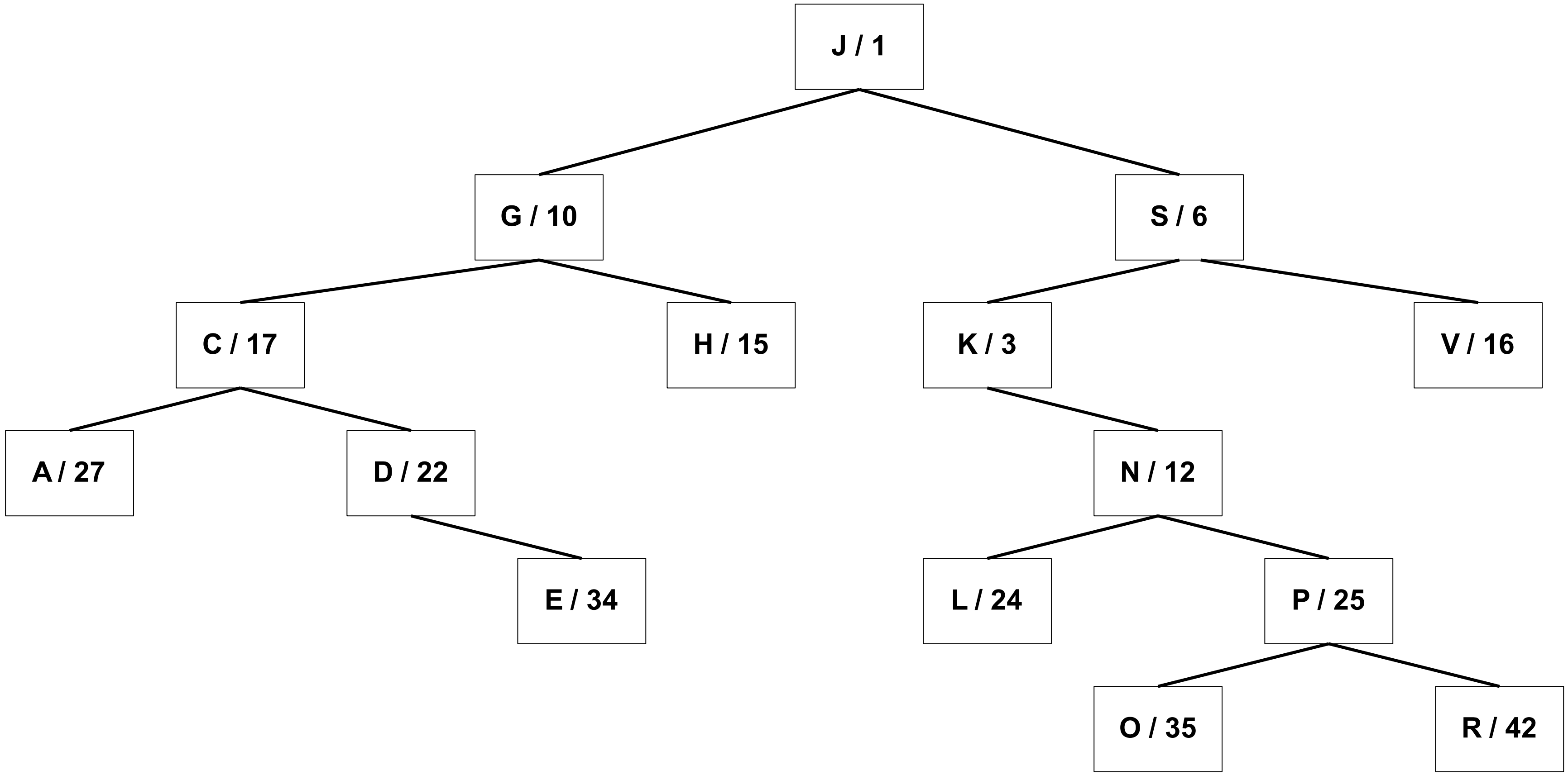
- Find X via recursive BST search
- When X is found, rotate with child that has the smaller priority
- If X is a leaf, just delete it
- If X is not a leaf, recursively remove X from its new subtree

# Remove K



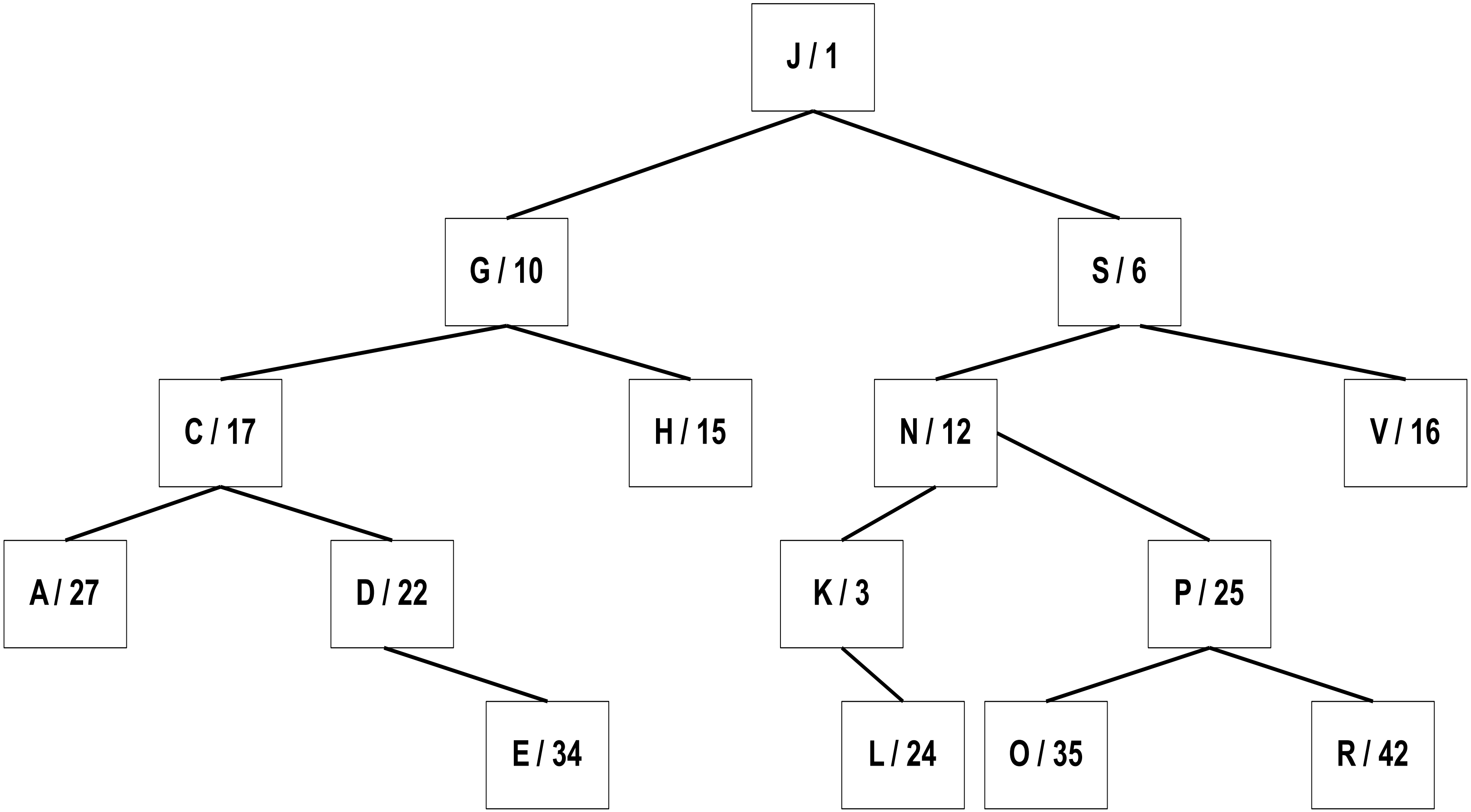

**Step 1 - Rotate K with Right Child ( S )**

# After Rotating K with S



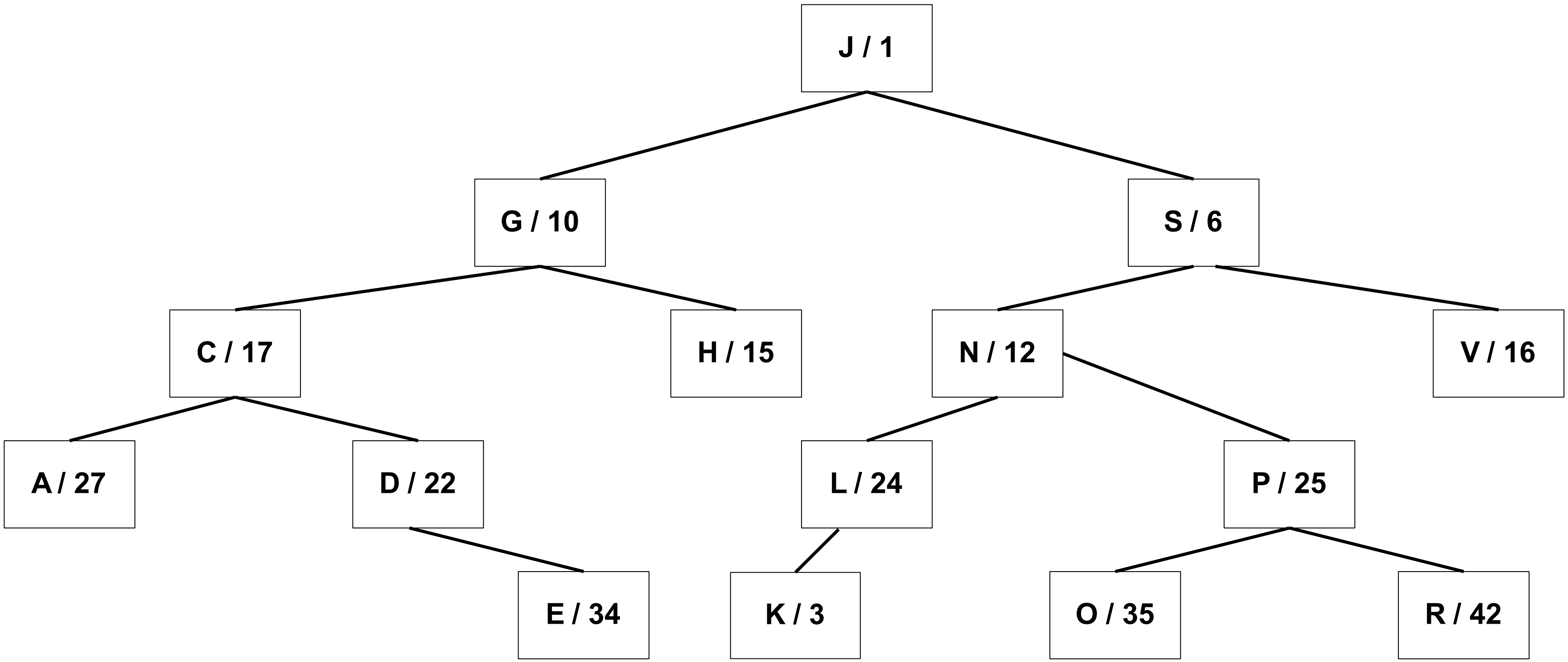

**Step 2 - Rotate K with Right Child ( N )**

# After Rotating K with N




**Step 3 - Rotate K with Right Child ( L )**

# After Rotating K with L




**Step 4 - K is a leaf delete K**

# remove( )

```
public void remove( AnyType x )  { root = remove( x, root ); }

private TreapNode< AnyType > remove( AnyType x, TreapNode< AnyType > t) {
  if( t != nullNode ) {
         int compare = x.compareTo( t.element );
         if ( compare < 0 )
             t.left = remove( x, t.left );
         else if ( compare > 0 )
             t.right = remove( x, t.right );

         // found x, swap x with child with smaller priority until x is a leaf
         else  {
             if ( t.left.priority < t.right.priority )
                  t = rotateWithLeftChild( t );
             else
                  t = rotateWithRightChild( t );
             if( t != nullNode )  // not at the bottom, keep going
                  t = remove( x, t );
             else                // at the bottom; restore nullNode's left child
                  t.left = nullNode;
         }
  }
  return t;
}
```

# Other Methods

```
public boolean isEmpty( ) { return root == nullNode; }

public void makeEmpty( ) { root = nullNode; }

public AnyType findMin( ) {
  if (isEmpty( ) ) throw new UnderFlowException( );
  TreapNode<AnyType>  ptr = root;
  while ( ptr.left != nullNode )
        ptr = ptr.left;
  return ptr.element;
}

public AnyType findMax( ) {
  if (isEmpty( ) ) throw new UnderFlowException( );
  TreapNode<AnyType>  ptr = root;
  while ( ptr.right != nullNode )
        ptr = ptr.right;
  return ptr.element;
}
```

# Treap Performance

- Determined by the height
- In theory, the random priority will result in a relatively balanced tree giving O( lg N ) performance
- To test the theory we inserted N integers in sorted order 10 times

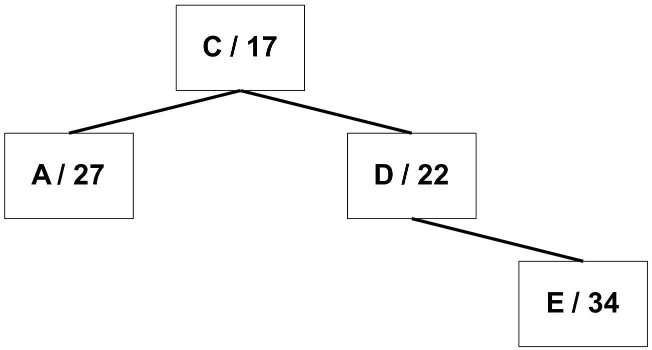
| N Consecutive Ints | lg( N ) | Height |
|---|---|---|
| 1024 | 10 | 19 - 23 |
| 32768 | 15 | 33 - 38 |
| 1048571 | 20 | 48 - 53 |
| 2097152 | 21 | 50 - 57 |
| 4194304 | 22 | 52 - 58 |
| 8388608 | 23 | 55 - 63 |
| 16777216 | 24 | 58 - 64 |

# Treaps and Sets

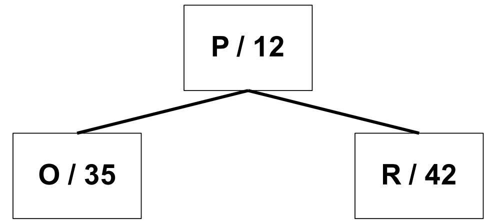
- Given two treaps, T1 and T2, such that all keys in T1 are “less than” all keys in T2
- To merge the treaps together
  - Create a “dummy” root node
  - Attach T1 as the root’s left subtree
  - Attach T2 as the root’s right subtree
  - Remove the root

- Used for set union
  - Each set in its own treap

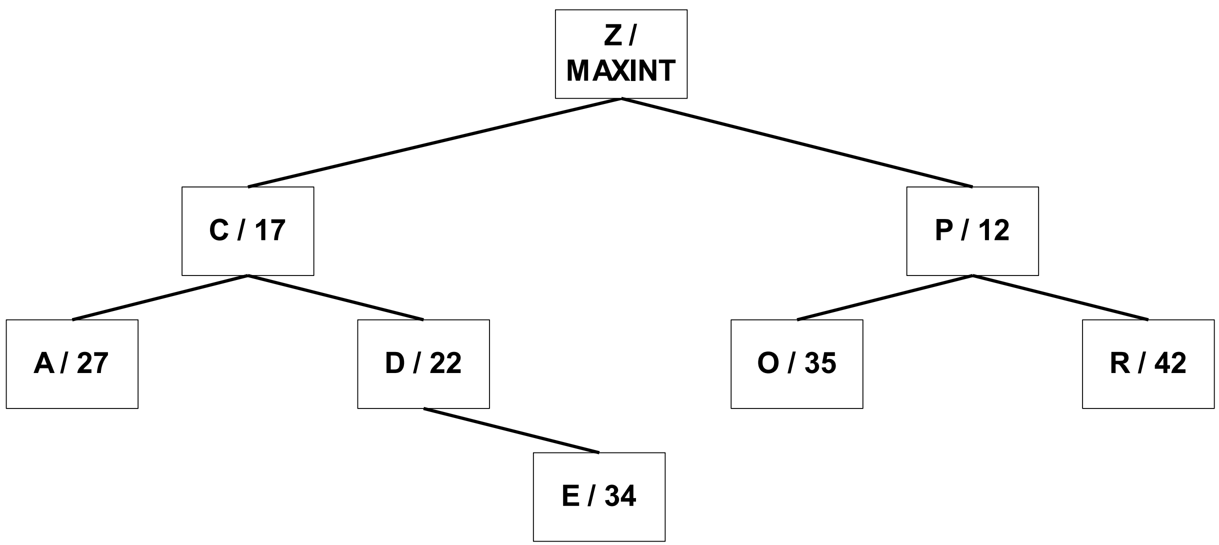
# Merge Example




Treap T1

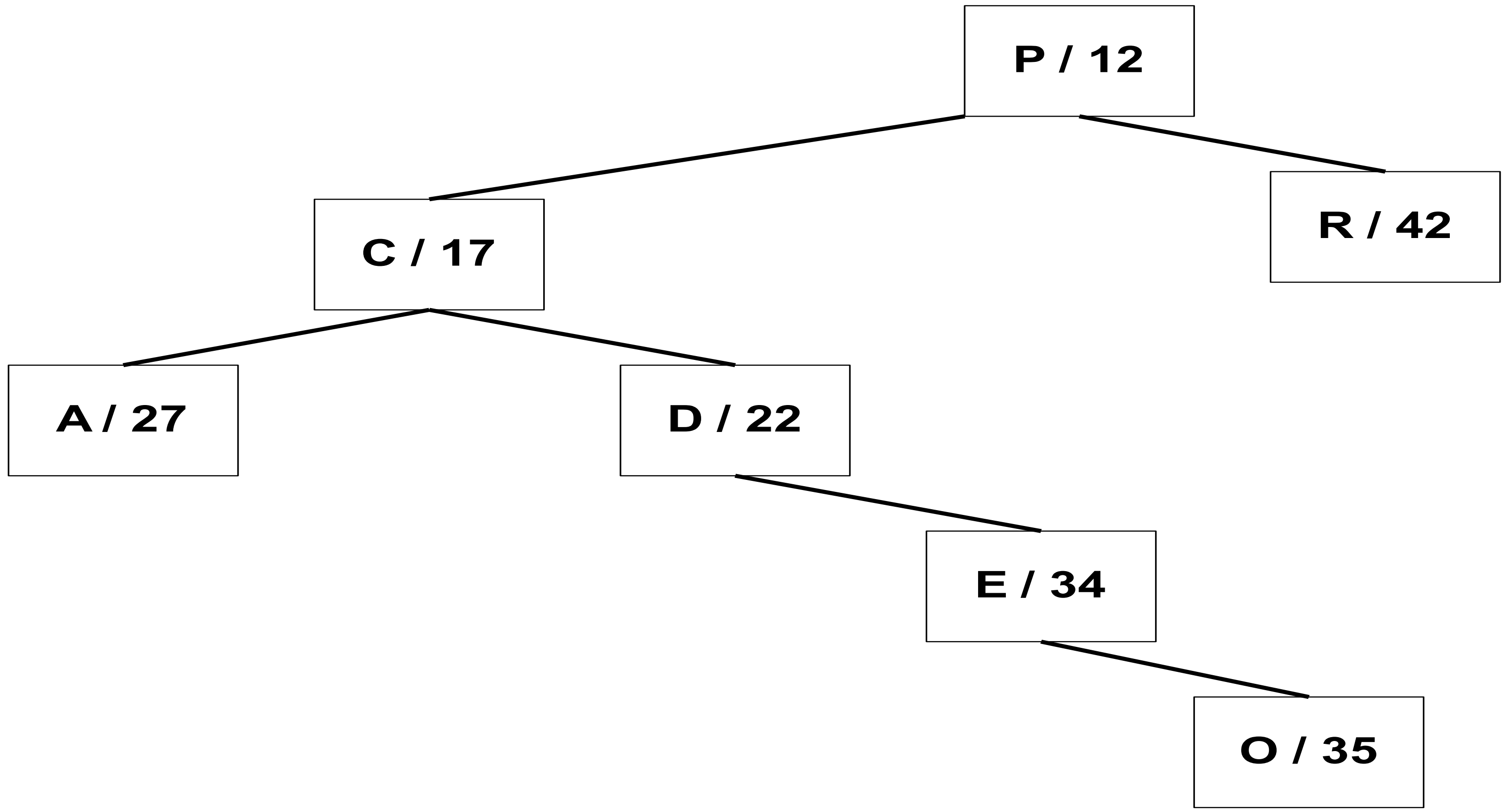



Treap 2




Dummy Root with T1 & T2 attached

# Merge Result



# Exercise

- Insert the characters

  K, F, P, M, N , L , G

  into an empty treap with priorities

  17, 22, 29, 10, 15, 26, 13 respectively.

# Exercise Solution