CSCI 1102 Computer Science 2

Meeting 21: Tuesday 4/13/2021

Immutable BSTs

2-3 Trees & Red Black Trees

Anthony Hoare

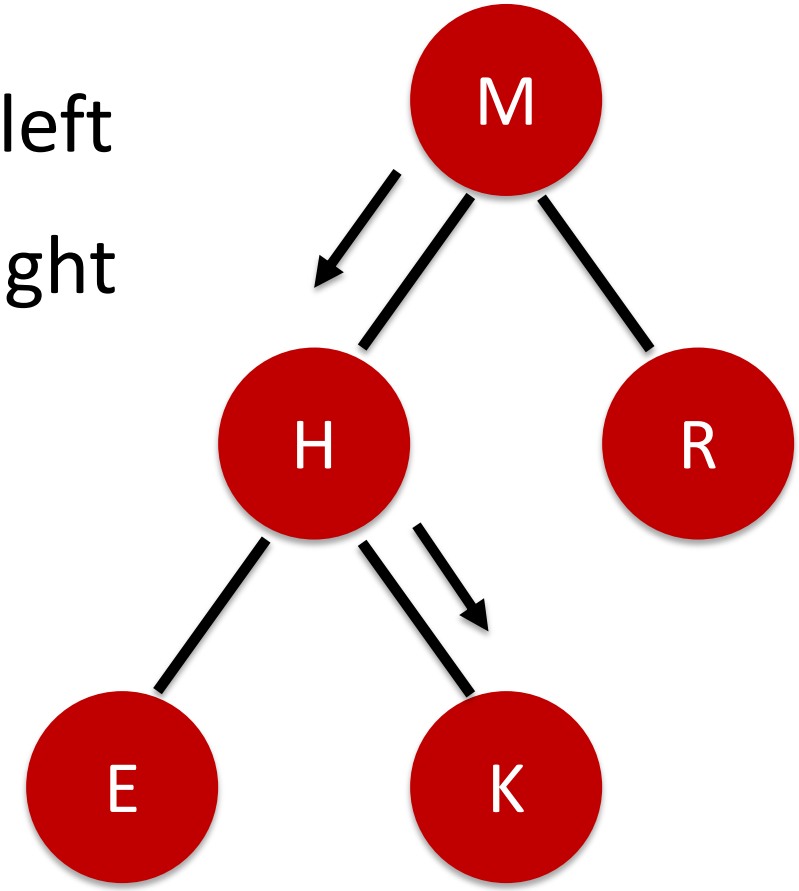
Turing Award 1980

+ Record types
+ Logical systems for reasoning about programs (Hoare Logic)
+ CSP
+ Quicksort
etc

Binary Search Trees

- Smaller keys to the left
- Larger keys to the right

bst.get(K)



```java
public interface OrderedMap<Key extends Comparable<Key>, Value> {
  int size();

  boolean isEmpty();

  boolean contains(Key key);

  Key floor(Key key);

  Key ceiling(Key key);

  Key max();

  Key min();

  void put(Key key, Value value);

  void deleteMin();

  void deleteMax();

  void delete(Key key);
}
```

An API for the BST ADT in the style of Sedgewick & Wayne

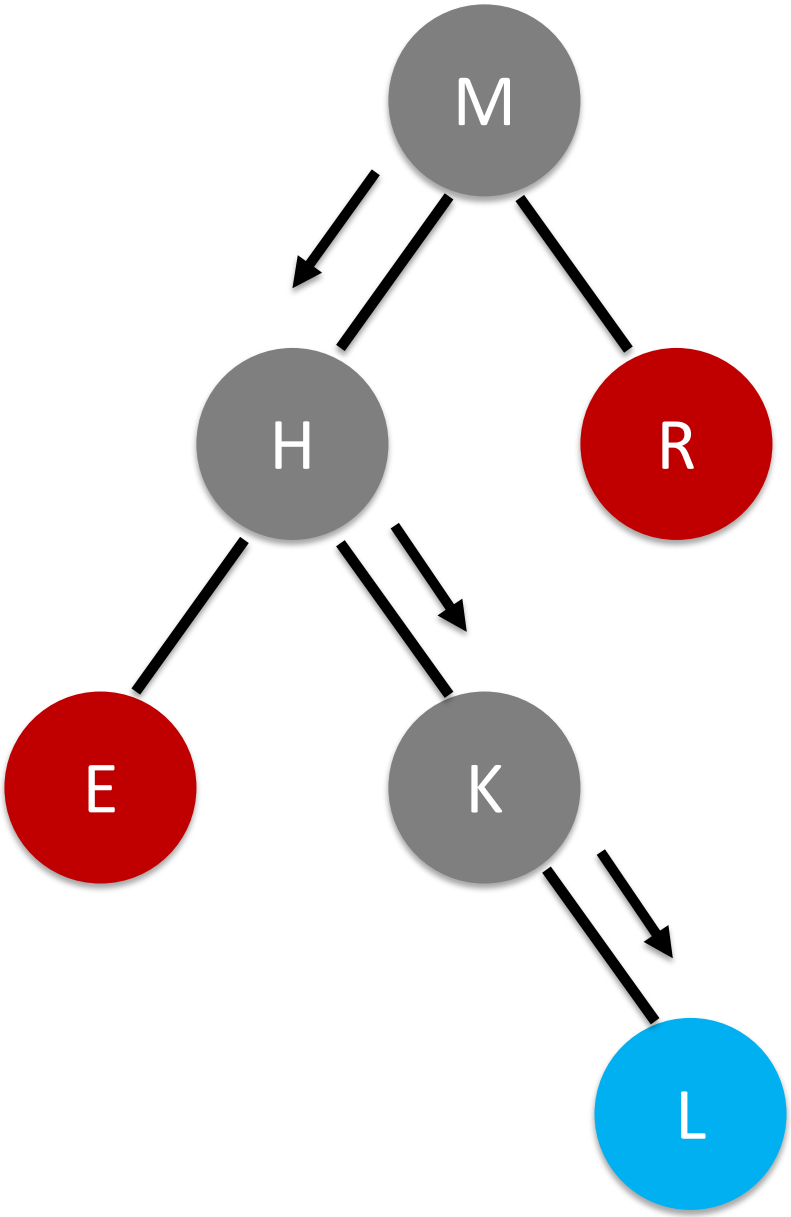
```java
public class BST<K extends Comparable<K>, V> implements OrderedMap<K, V> {

  private Node root;

  private class Node {
    private K key;
    private V value;
    private Node left;
    private Node right;
    private int size;
```

A private inner recursive class Node, empty BST represented by null.

```java
public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("calls put(
    if (val == null) {
        delete(key);
        return;
    }
    root = put(root, key, val);
    assert check();
}

private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = put(x.left,  key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else              x.val   = val;
    x.size = 1 + size(x.left) + size(x.right);
    return x;
}
```

bst.put(L, 5)

Mutates all of the nodes on the path from the root to the insertion point.



```
public interface OrderedMap<Key extends Comparable<Key>, Value> {
  boolean isEmpty();

  int size();

  Value get(Key key);

  boolean contains(Key key);

  Key ceiling(Key key);

  Key floor(Key key);

  Key max();

  Key min();

  OrderedMap<Key, Value> put(Key key, Value value);

  OrderedMap<Key, Value> deleteMin();

  OrderedMap<Key, Value> deleteMax();

  OrderedMap<Key, Value> delete(Key key);
```

Making empty BSTs First-class

Public classes Node & Empty both implement OrderedMap.

```java
public class Node<K extends Comparable<K>, V> implements OrderedMap<K, V> {

  private K key;
  private V value;
  private int size;
  private OrderedMap<K, V> left, right;

  public Node(K key, V value) {
    this.key = key;
    this.value = value;
    this.left = new Empty<K, V>();
    this.right = this.left;
    this.size = 1;
  }
```

```java
public OrderedMap<K, V> put(K key, V value) {
  if (key == null)
    throw new IllegalArgumentException("put: null key");
  int cmp = key.compareTo(this.key);
  if (cmp == 0) this.value = value;
  else if (cmp < 0) this.left = this.left.put(key, value);
  else
    this.right = this.right.put(key, value);
  this.size = 1 + this.left.size() + this.right.size();
  return this;
}
```

Immutable BSTs

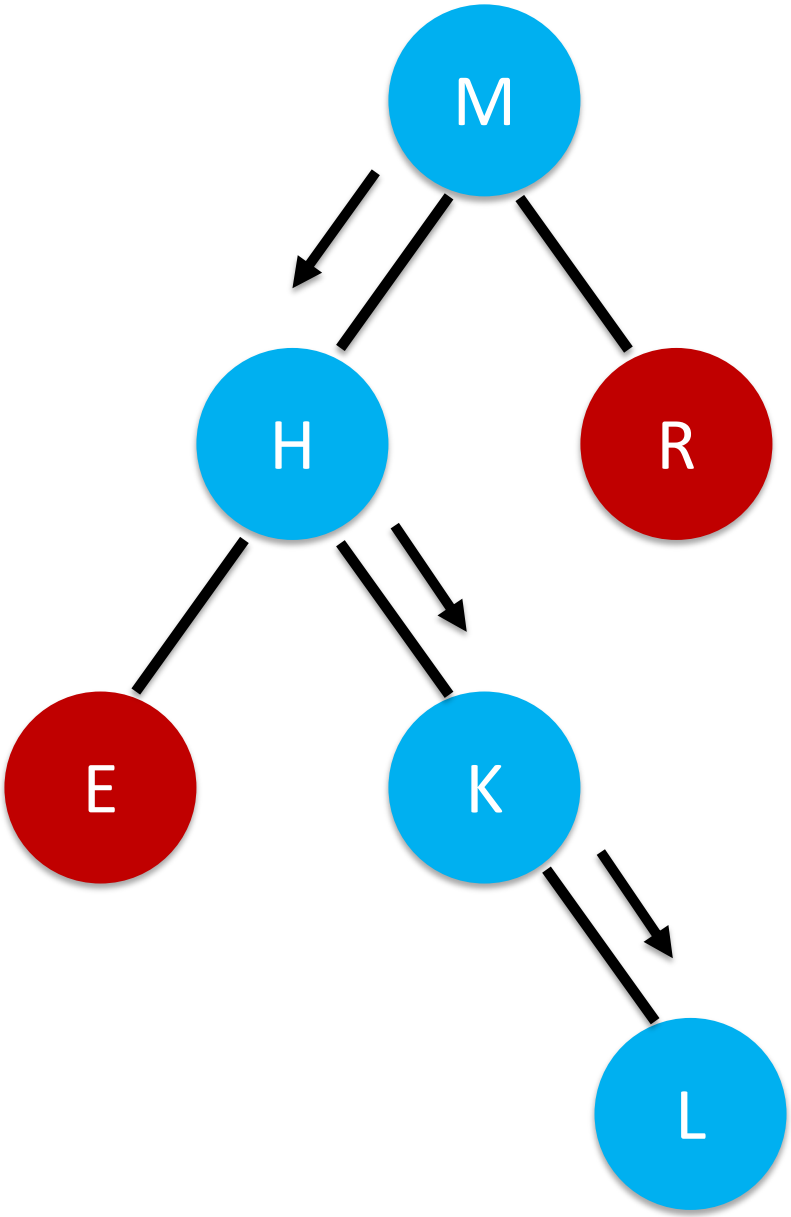
Operations that might change a BST instead return a new one.

Structure Sharing

```java
public OrderedMap<K, V> put(K key, V value) {
  if (key == null)
    throw new IllegalArgumentException("put: null key");
  int cmp = key.compareTo(this.key);
  if (cmp == 0)
    return new Node<K, V>(key, value, size, left, right);
  else if (cmp < 0) {
    OrderedMap<K, V> newLeft = left.put(key, value);
    int newSize = 1 + newLeft.size() + right.size();
    return new Node<K, V>(this.key, this.value, newSize, newLeft, right);
  }
  else {
    OrderedMap<K, V> newRight = right.put(key, value);
    int newSize = 1 + left.size() + newRight.size();
    return new Node<K, V>(this.key, this.value, newSize, left, newRight);
  }
}
```

bst.put(L, 5)

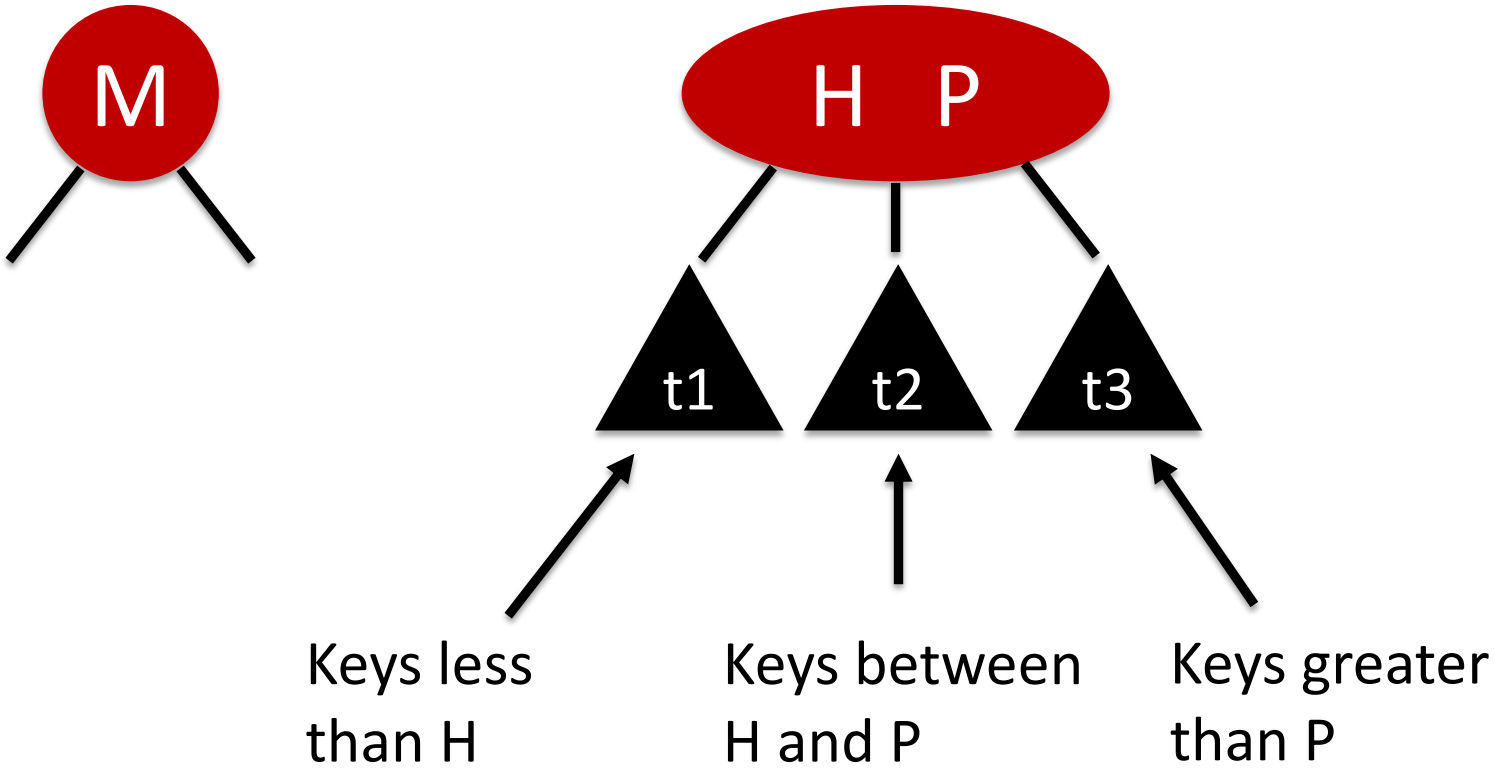
Rebuilds all of the nodes on the path from the root to the insertion point.



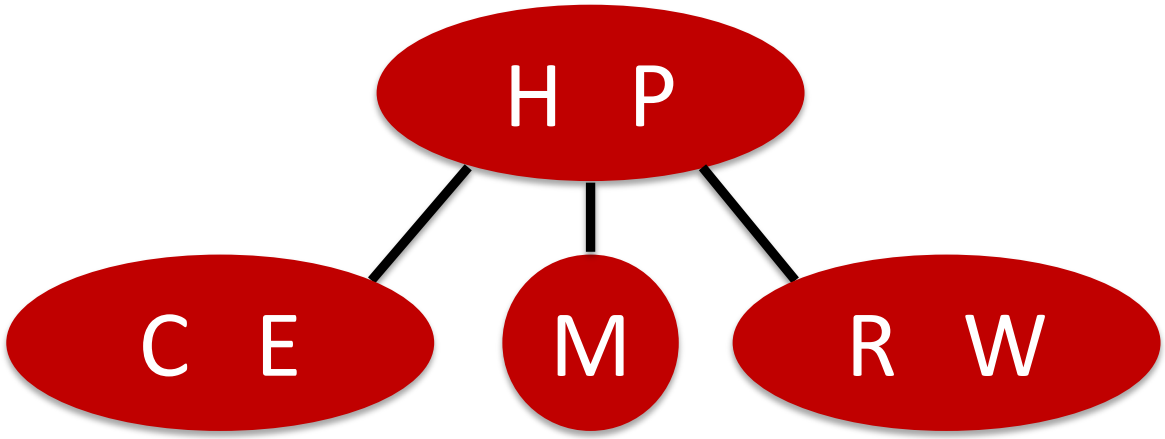
```java
public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("calls put(
    if (val == null) {
        delete(key);
        return;
    }
    root = put(root, key, val);
    assert check();
}

private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = put(x.left,  key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else              x.val   = val;
    x.size = 1 + size(x.left) + size(x.right);
    return x;
}
```

Balanced Trees

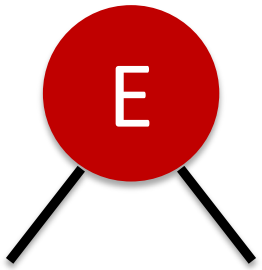
2-3 Trees are Balanced
all leaves at same depth



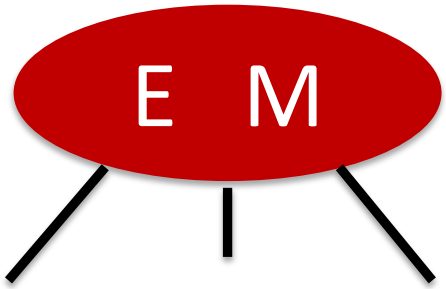
2-3 Trees are Balanced
all leaves at same depth



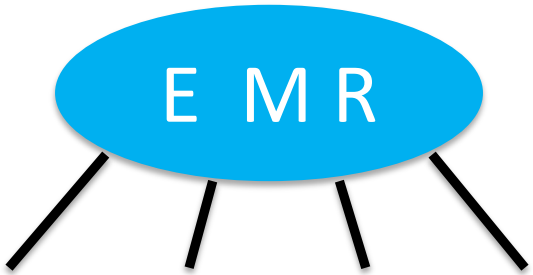
Insert E



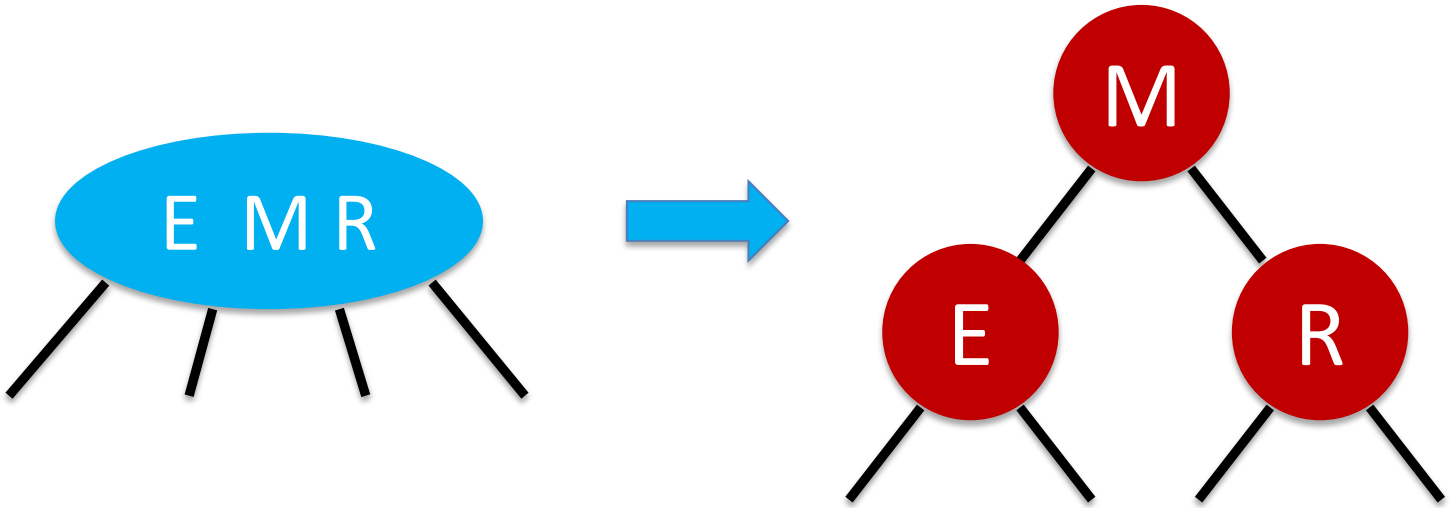
Insert M



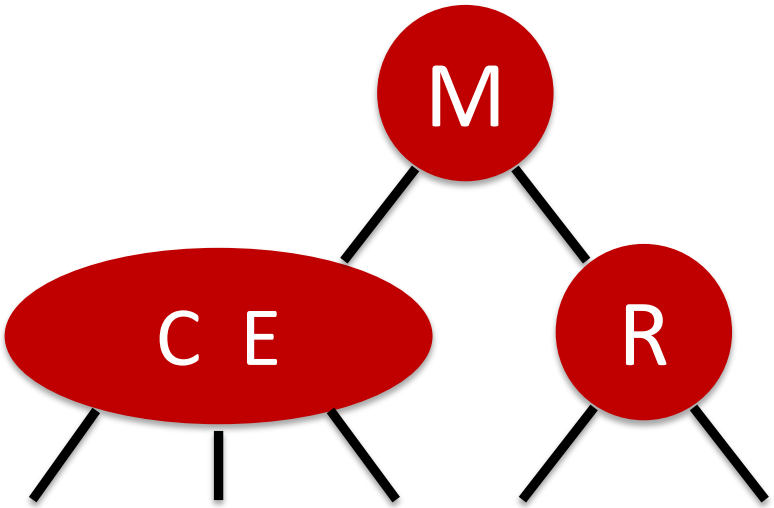
Insert R -- Split



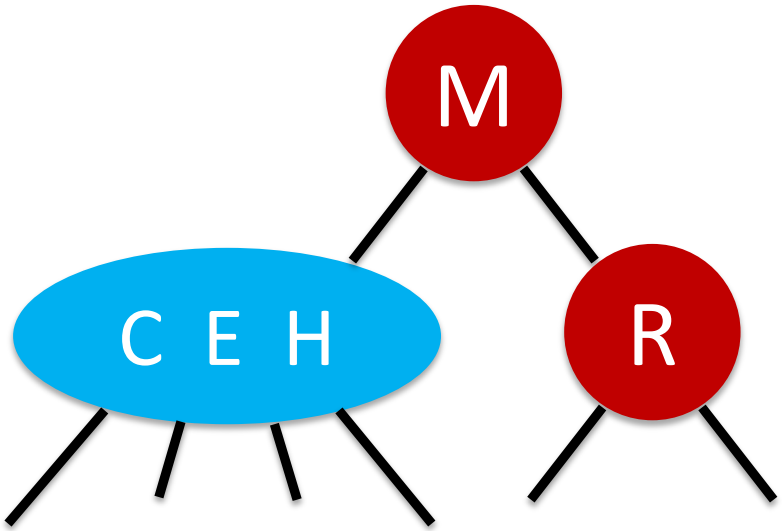
Insert R -- Split



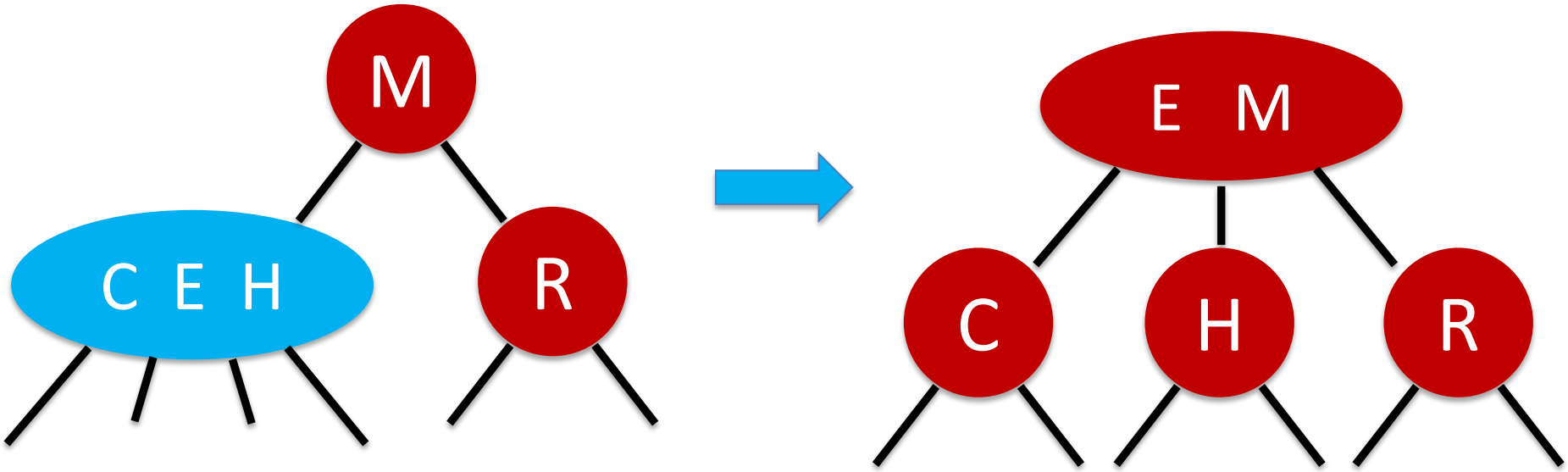
Insert C



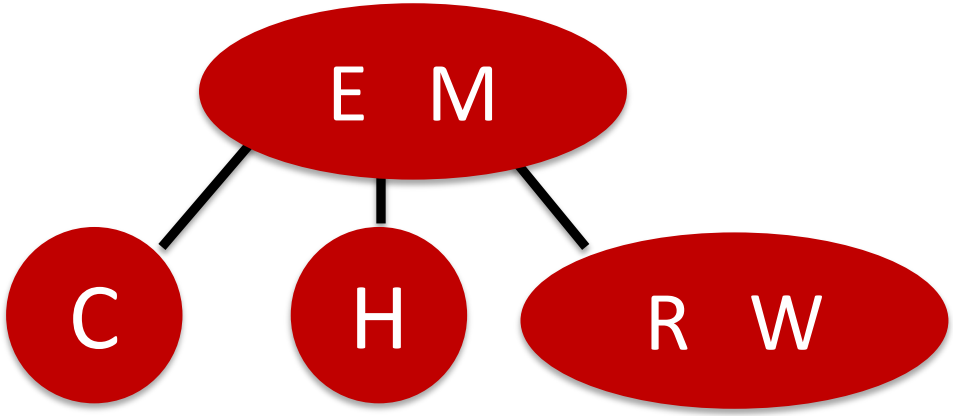
Insert H -- Split



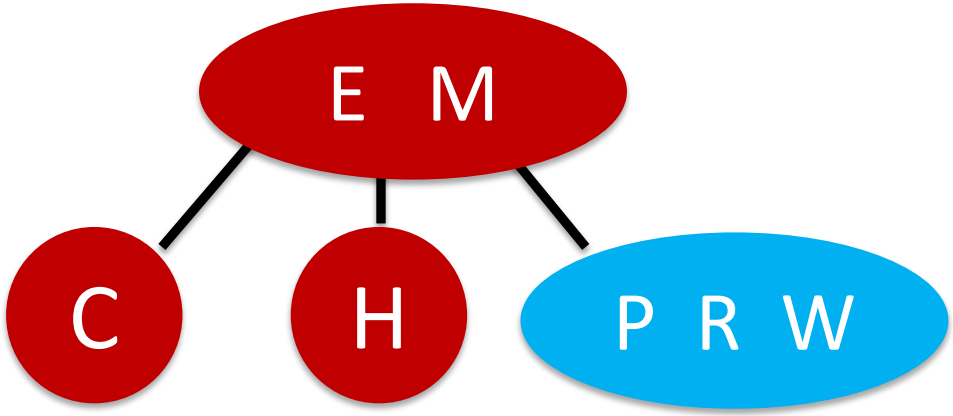
Insert H -- Split



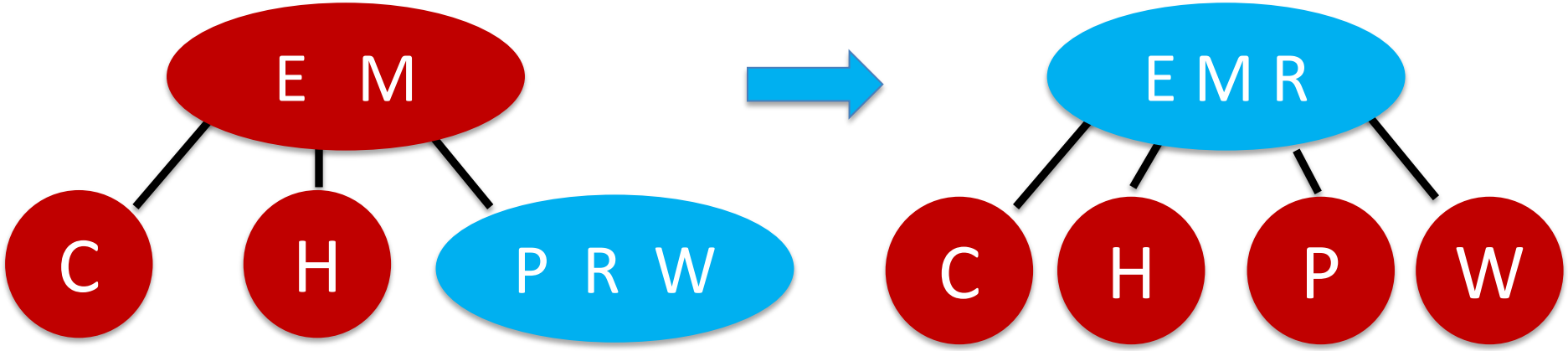
Insert W



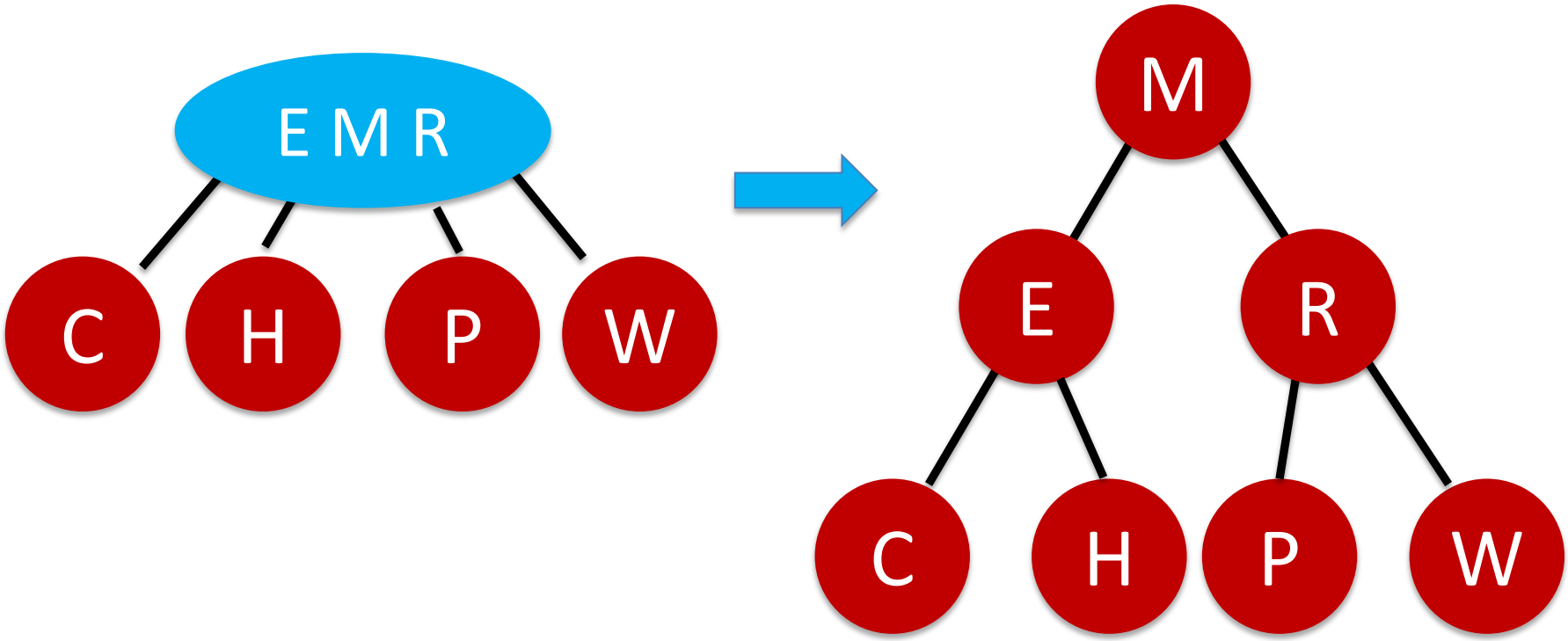
Insert P -- split



Insert P -- split

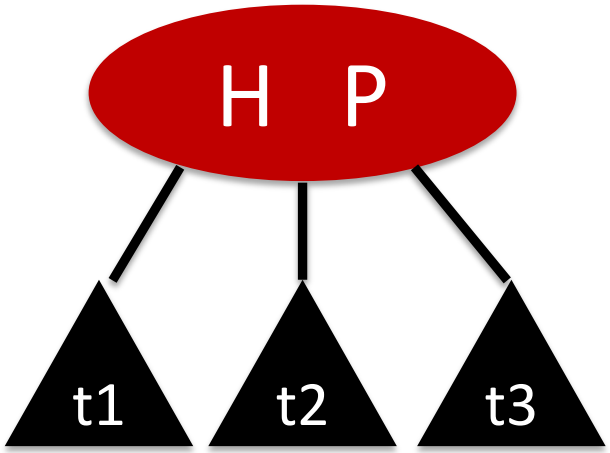


Split the root

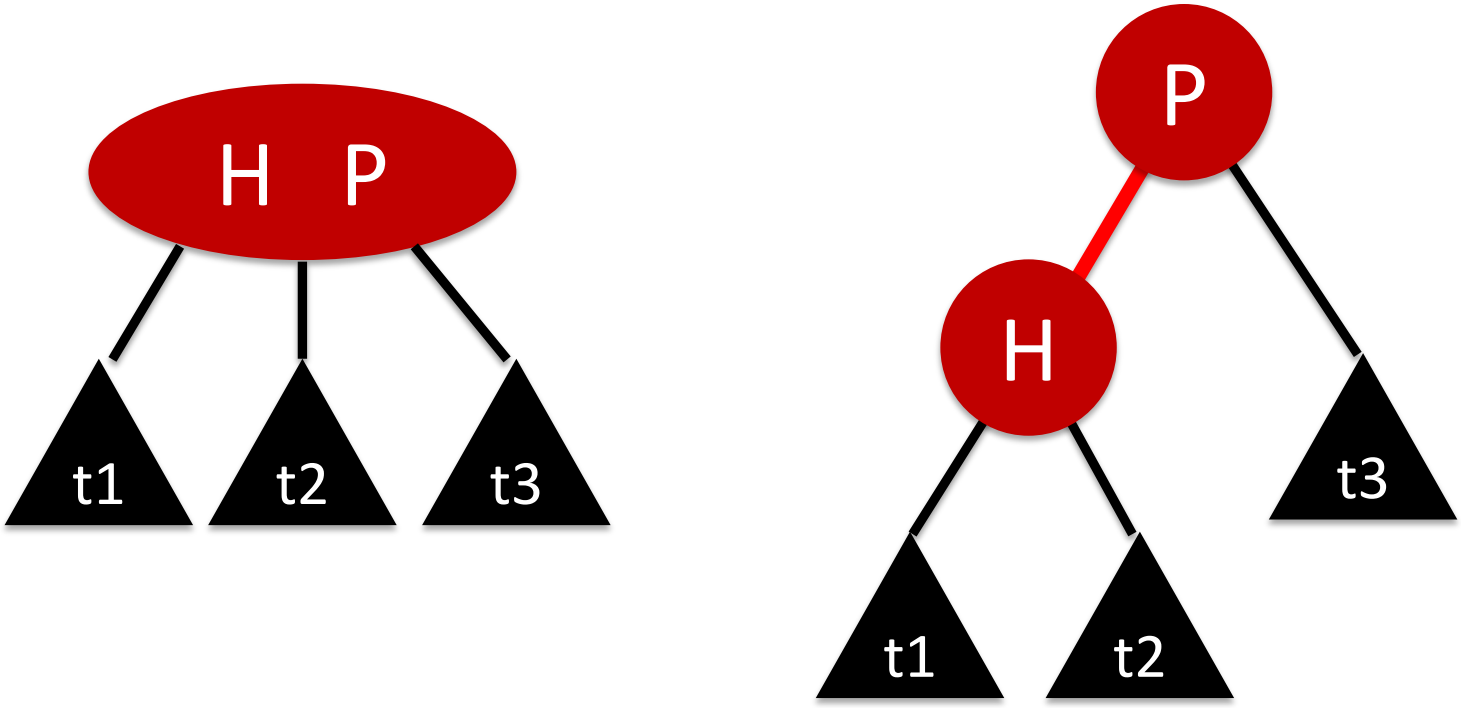


Left-Leaning Red-Black Trees

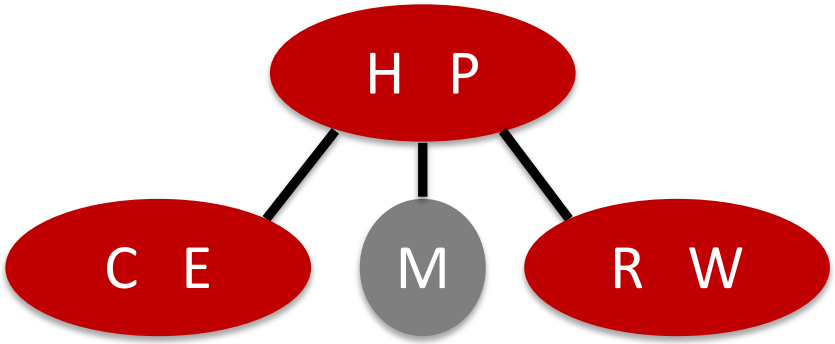
Representing 3-nodes with colored 2-nodes

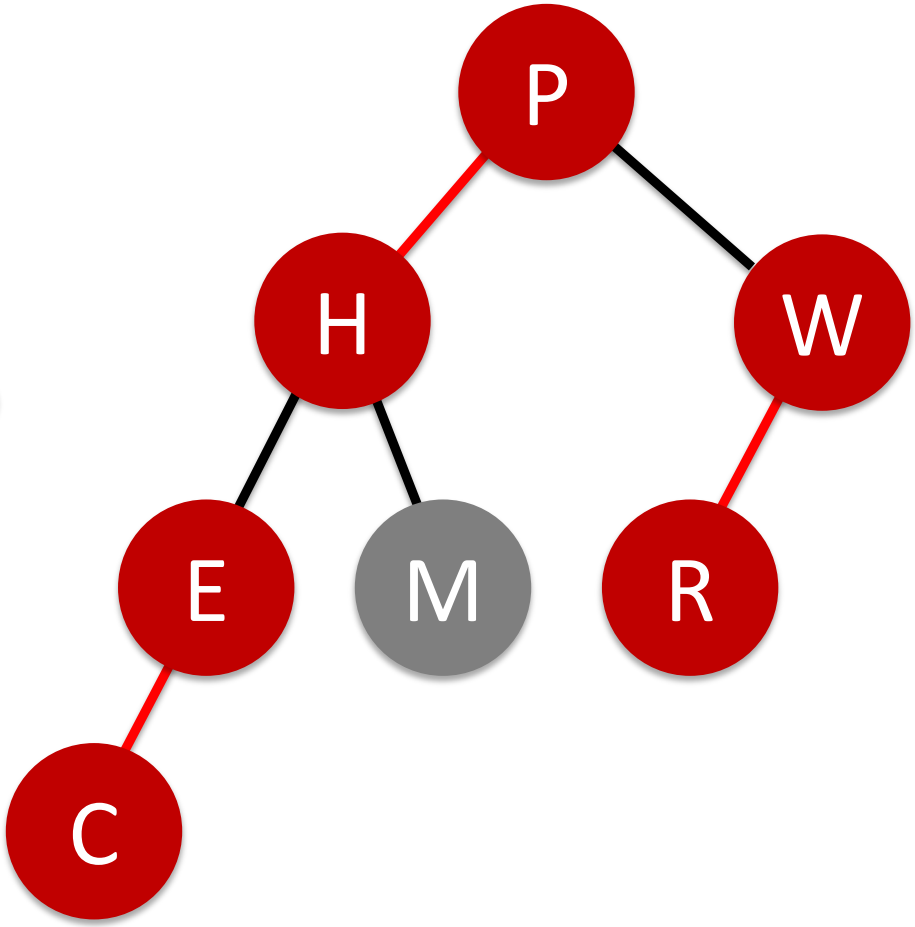


Representing 3-nodes with colored 2-nodes



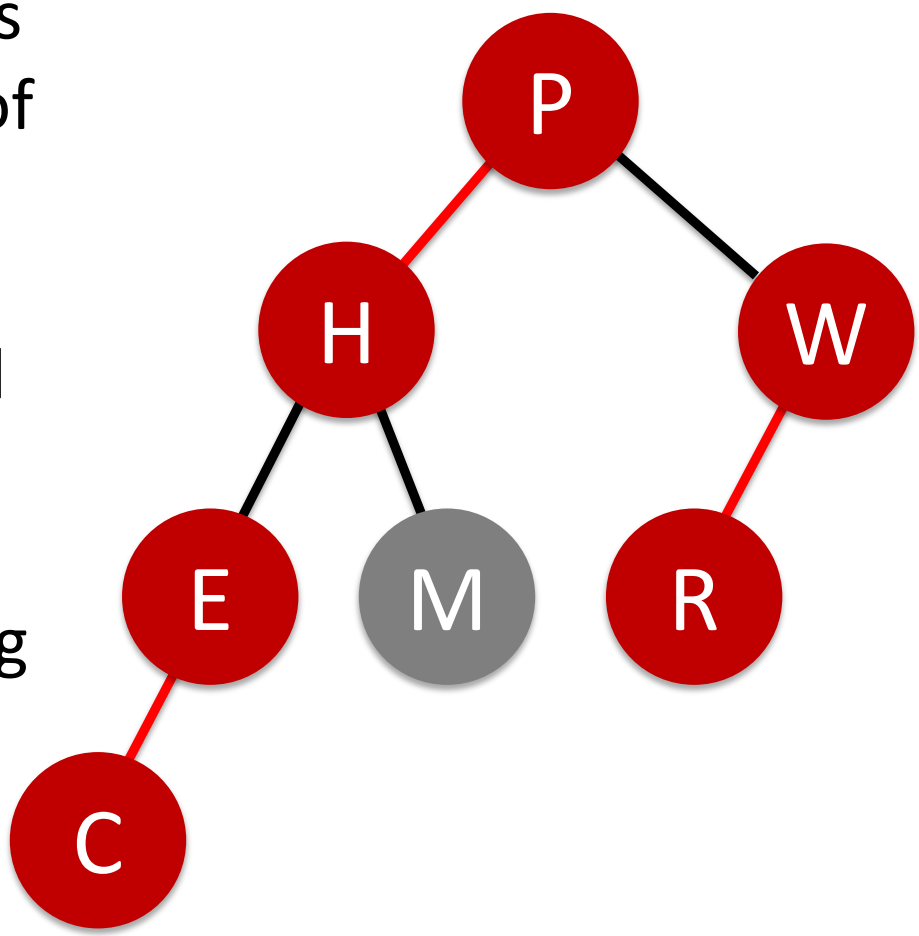
Representing 3-nodes with colored 2-nodes





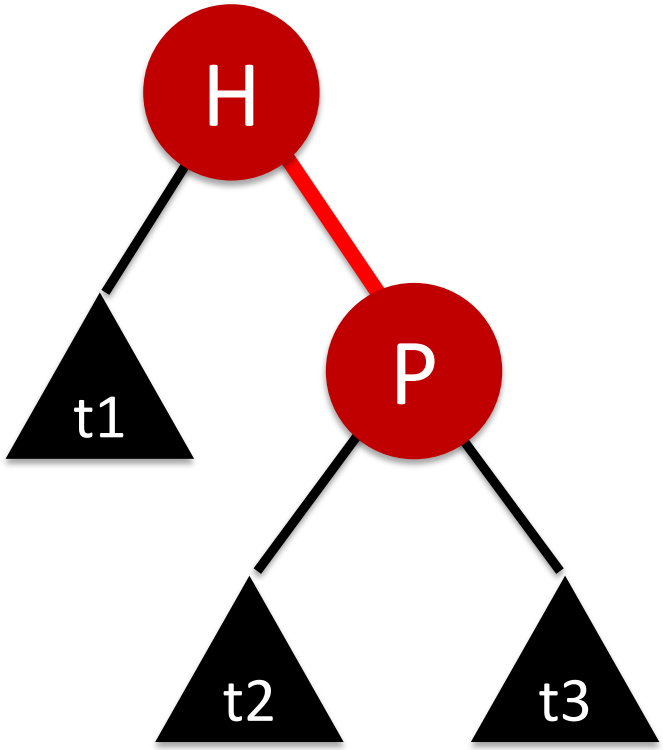
Invariants:

- Path to every leaf passes through same number of black links;
- Never 2 consecutive red links;
- No node with 2 outgoing red links;
- No right red links.

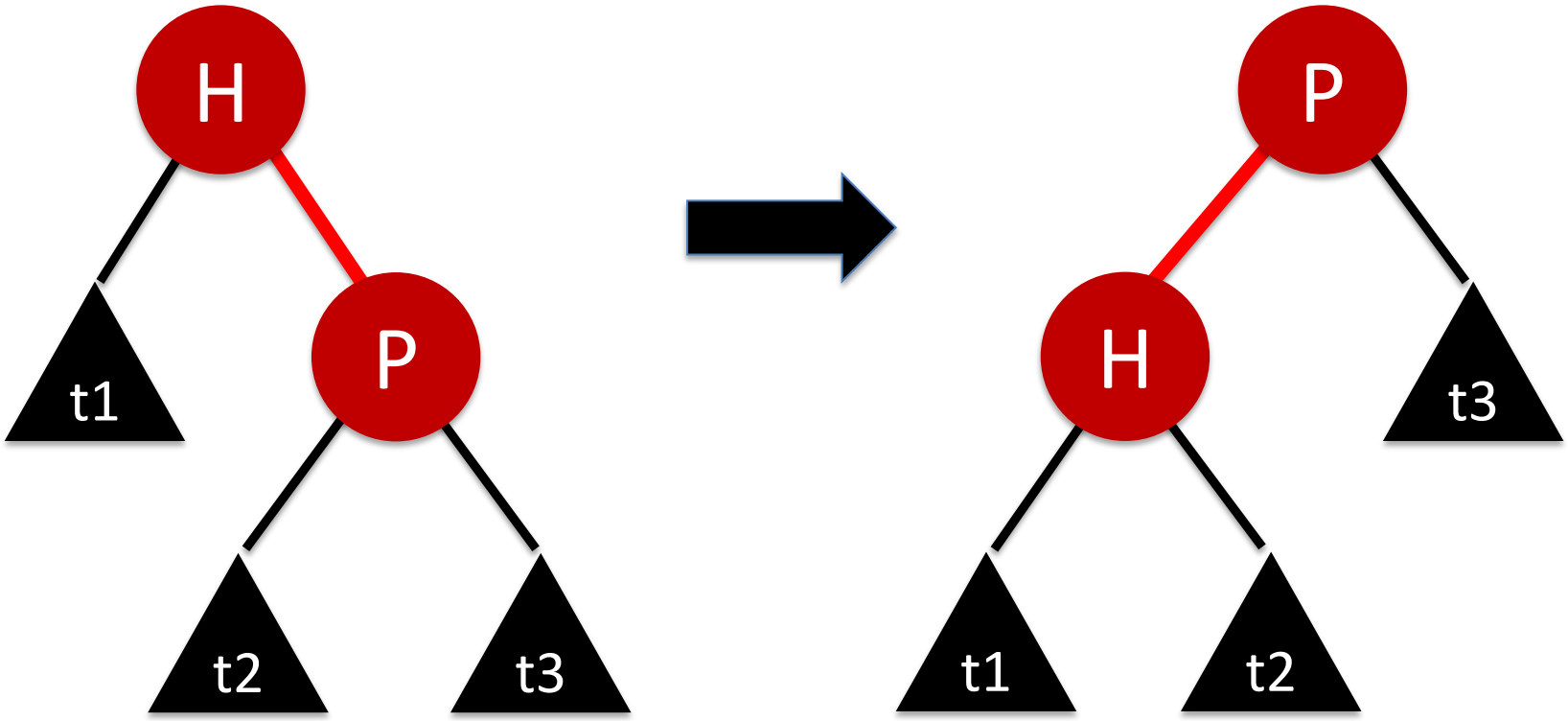


Tree Rotations

Rotate Left



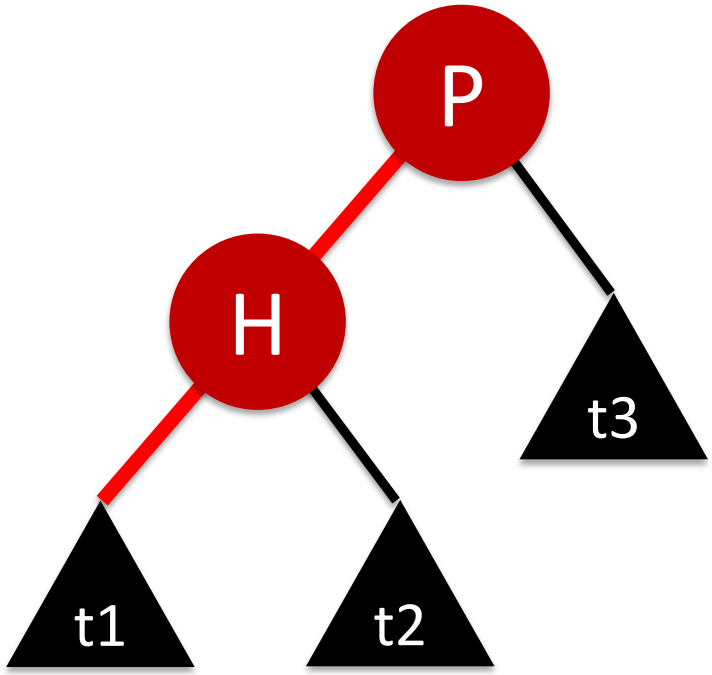
Rotate Left



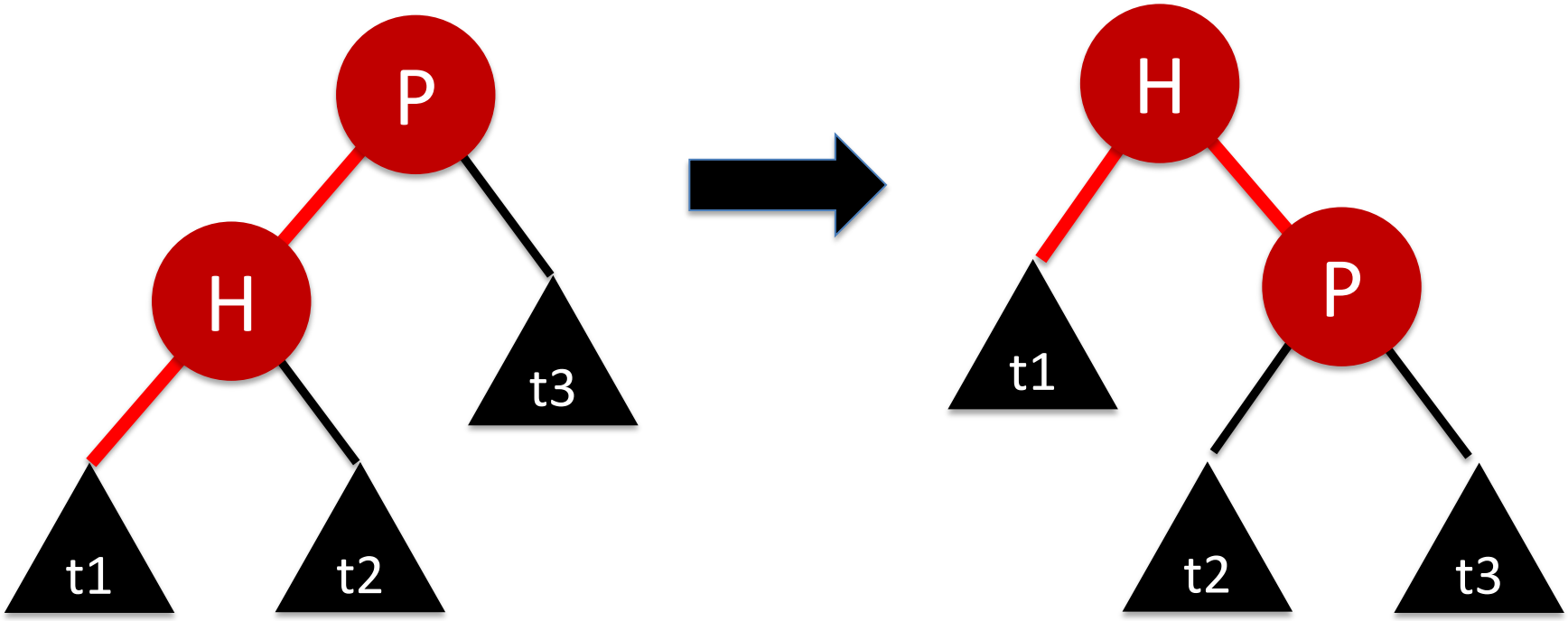
```java
// make a right-leaning link lean to the left
private Node rotateLeft(Node h) {
  assert (h != null) && isRed(h.right);
  Node x = h.right;
  h.right = x.left;
  x.left = h;
  x.color = x.left.color;
  x.left.color = RED;
  x.size = h.size;
  h.size = size(h.left) + size(h.right) + 1;
  return x;
}
```

```java
// insert the key-value pair in the subtree rooted at h
private Node put(Node h, Key key, Value val) {
  if (h == null) return new Node(key, val, RED, 1);

  int cmp = key.compareTo(h.key);
  if (cmp < 0) h.left = put(h.left, key, val);
  else if (cmp > 0) h.right = put(h.right, key, val);
  else h.val = val;

  // fix-up any right-leaning links
  if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
  if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
  if (isRed(h.left) && isRed(h.right)) flipColors(h);
  h.size = size(h.left) + size(h.right) + 1;

  return h;
}
```

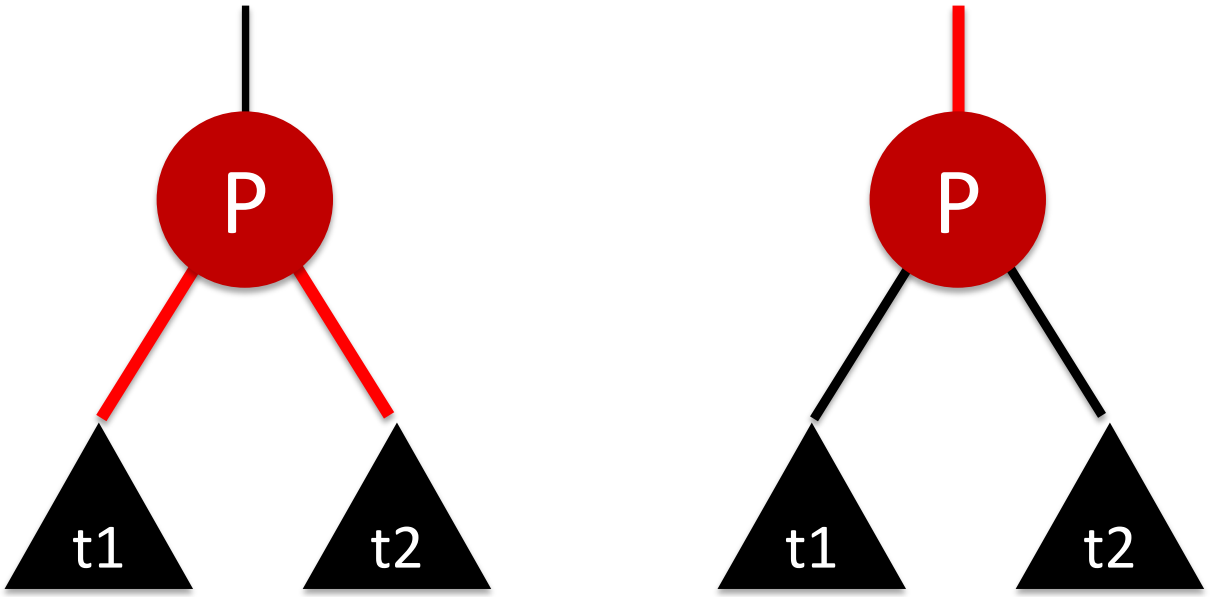
Rotate Right



Rotate Right



Flip



Work

How Much?

- How much time does an operation take?
- How much memory does it use?
- In the worst case? On average?
- How does the work grow as the input grows?

Measure the Input Size – N