AVL Tree Splitting and Merging

261217 Data Structures for Computer Engineers

Patiwet Wuttisarnwattana, Ph.D.

patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

Learning Objectives

- Implement merging and splitting of AVL trees.
- Analyze the runtime of these operations

New Operations

- Another useful feature of binary search trees is the ability to recombine them in interesting ways.
- We discuss two new operations:
  - Merge: Combines two binary search trees into a single one.
  - Split: Breaks one binary search tree into two

Merge

- In general, to merge two sorted AVL trees takes O(n log n) times
- However, when they are separated it is faster

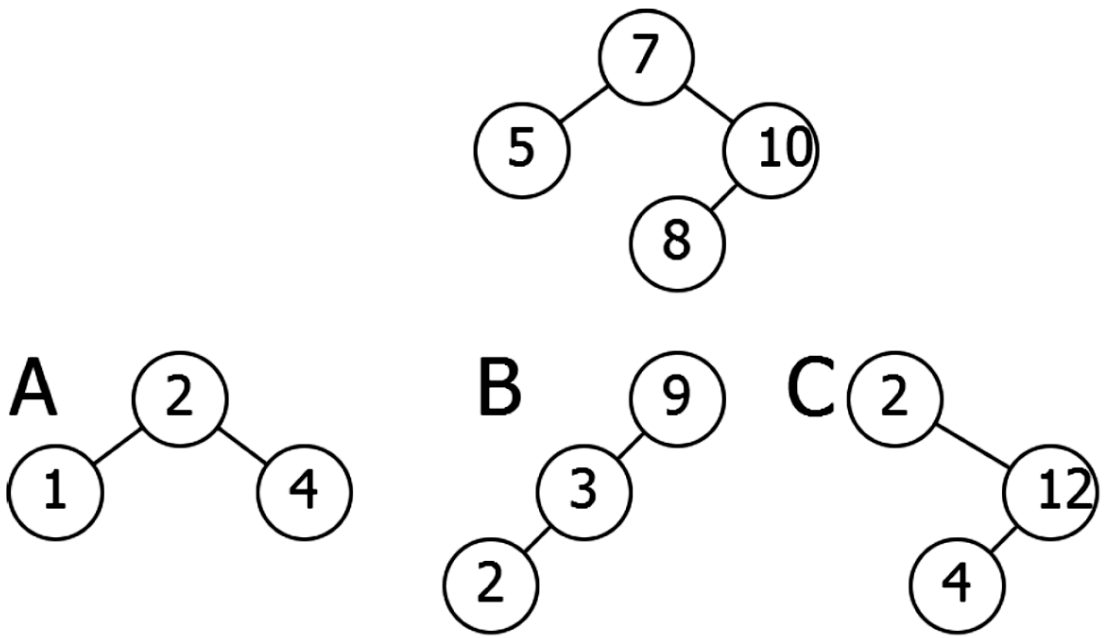
Merge

Input: Roots $\mathbf{R_1}$ and $\mathbf{R_2}$ of trees with all keys in $\mathbf{R_1}$'s tree smaller than those in $\mathbf{R_2}$'s

Output: The root of a new tree with all the elements of both trees

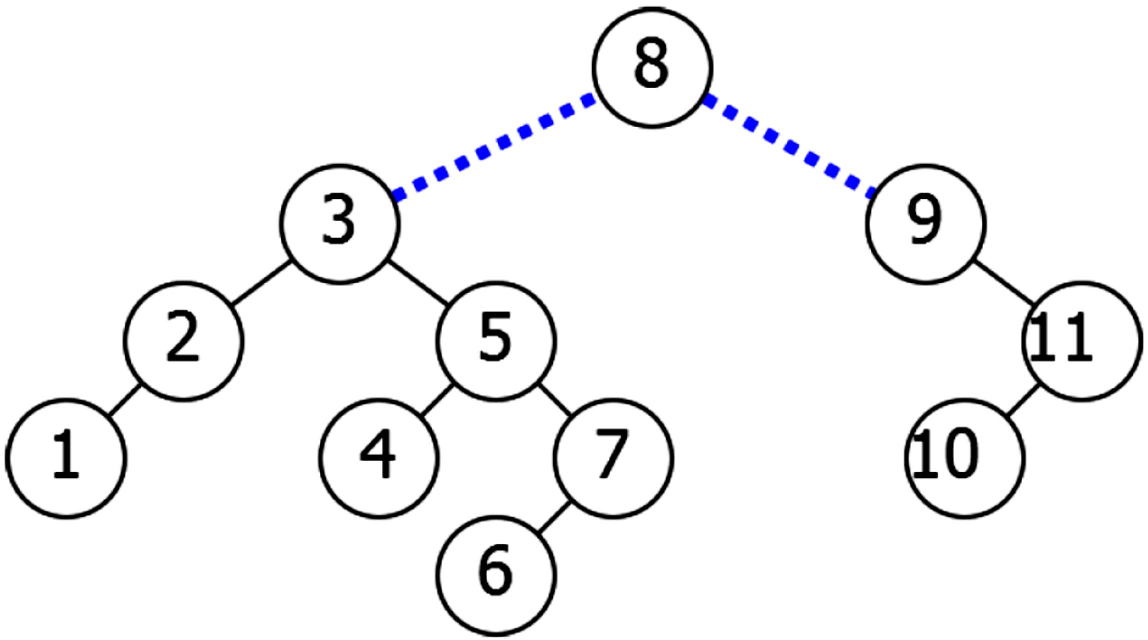
Question

- Which tree can be merged with the given one?



Extra Root

- Easy if you have an extra node to add as root

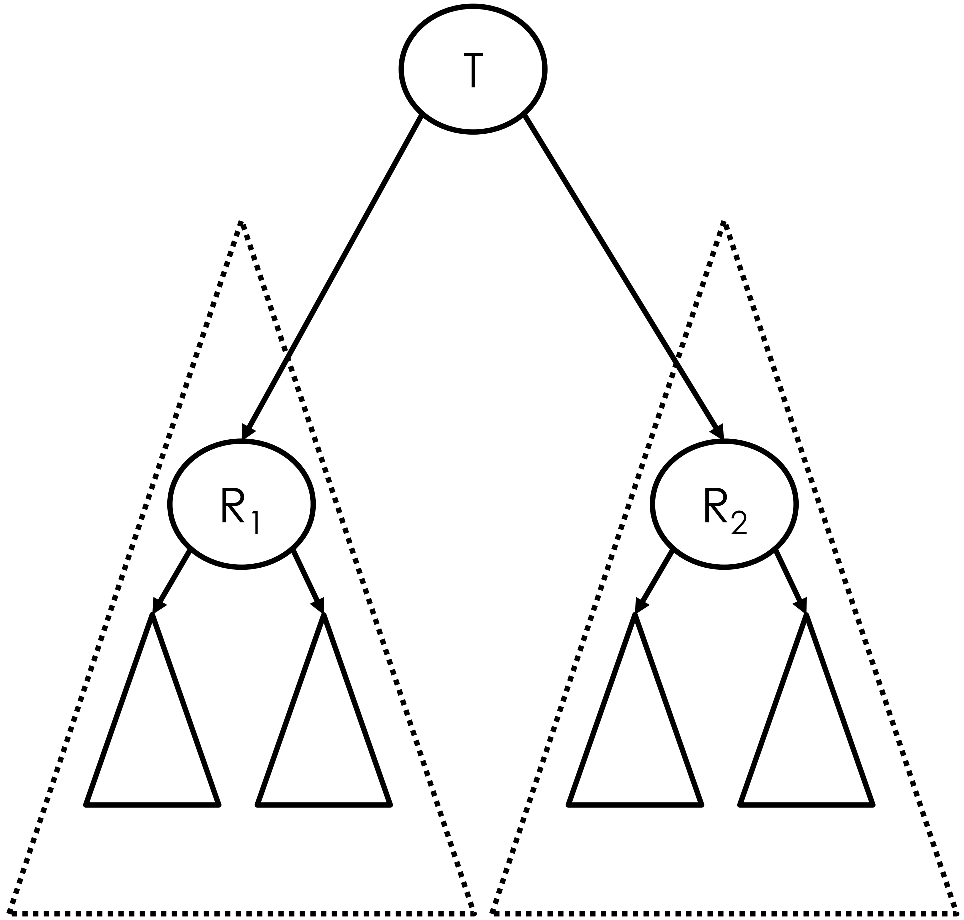


Implementation

MergeWithRoot(R_1, R_2, T)

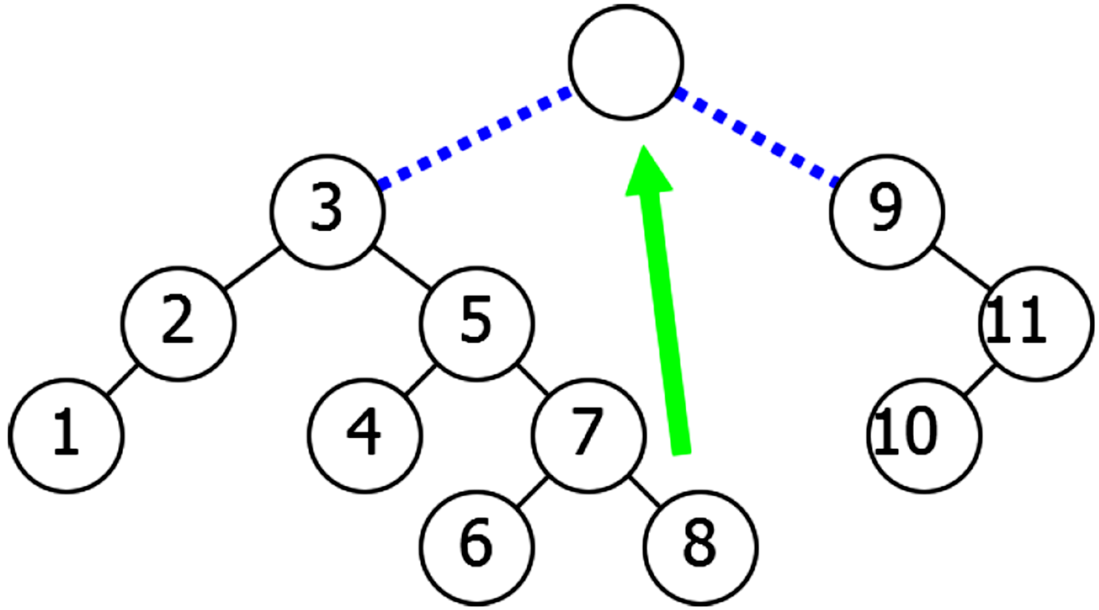
T.Left $\leftarrow R_1$
T.Right $\leftarrow R_2$
R_1.parent $\leftarrow$ T
R_2.parent $\leftarrow$ T
return T

Time O(1)



Get Root

- Get new root by removing largest element of the left subtree
  - Alternatively, you can use the smallest element of the right subtree



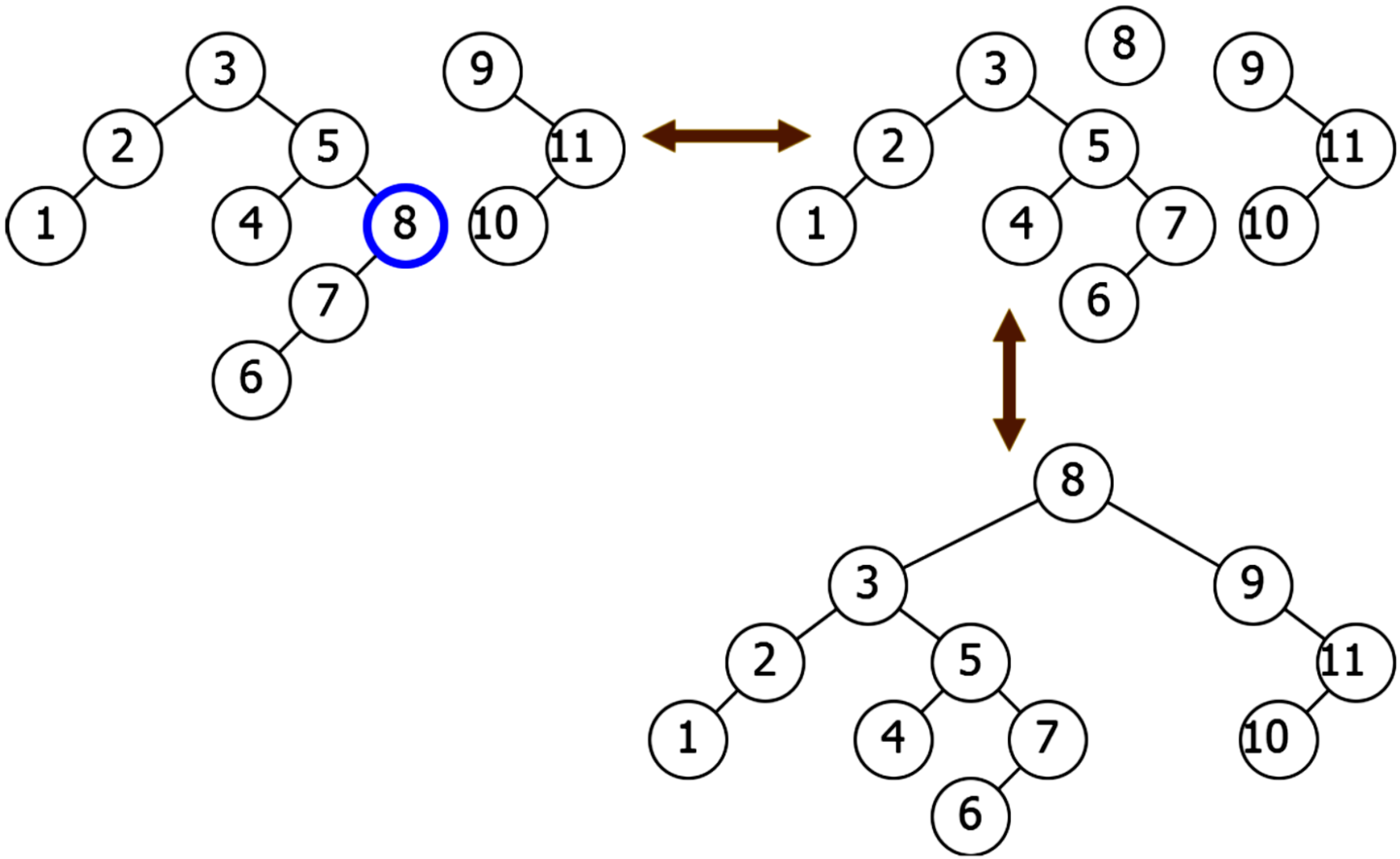
Merge

Merge(R_1, R_2)

$T \leftarrow$ FindMax(R_1)
R_1.delete(T.key)
MergeWithRoot(R_1, R_2, T)
return T

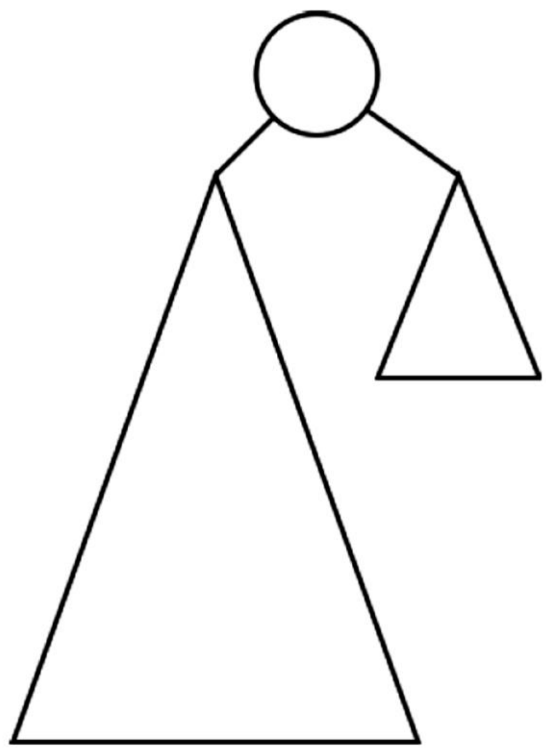
Time O(h)

Merge



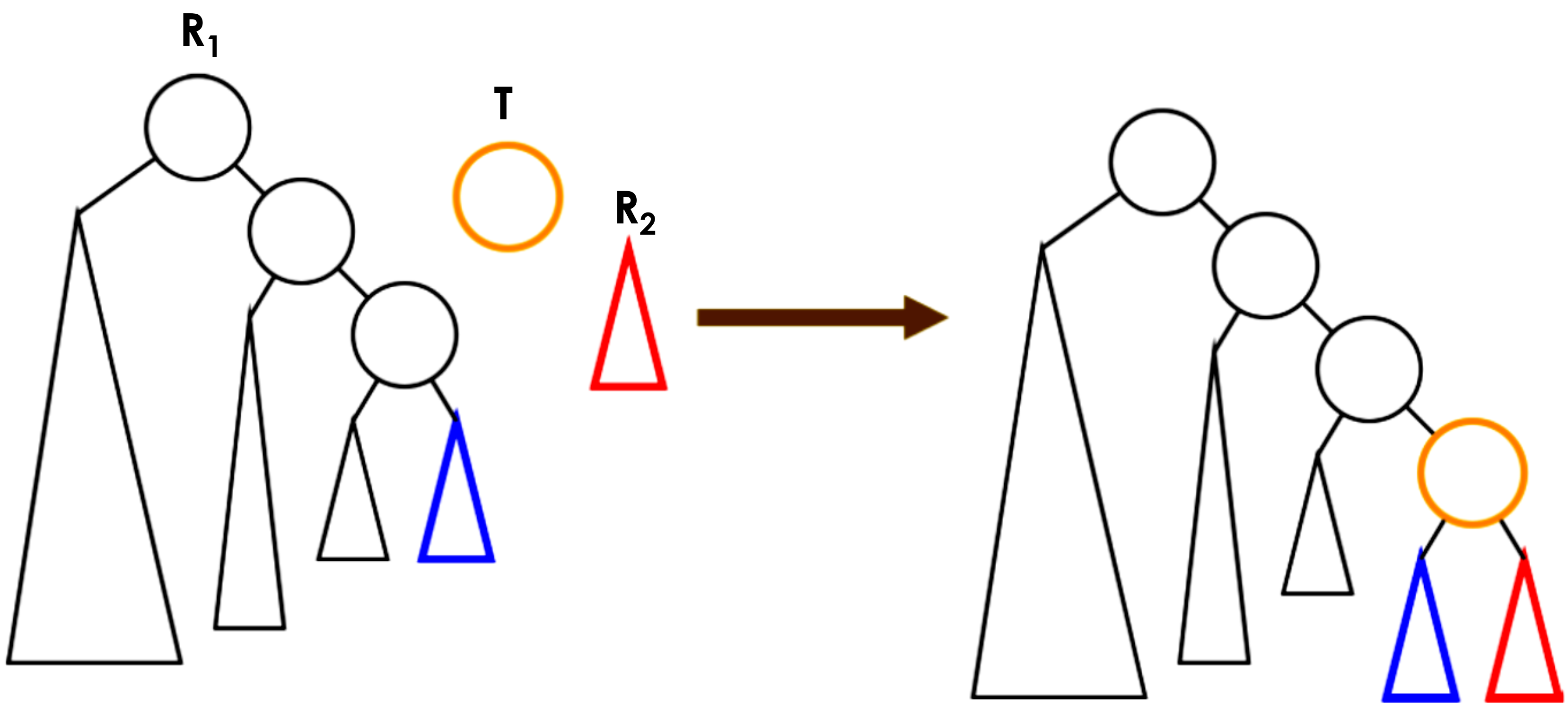
Balance

- Unfortunately, this merge does not preserve balance properties



Idea

- Go down side of tree until merge with subtree of same height



Implementation

AVLTreeMergeWithRoot(R_1, R_2, T)

```
if |R₁.height − R₂.height| ≤ 1:
    MergeWithRoot(R₁, R₂, T)
    T.height ← max(R₁.height, R₂.height) + 1
    return T
...
```

Implementation (continued)

AVLTreeMergeWithRoot(R_1, R_2, T)

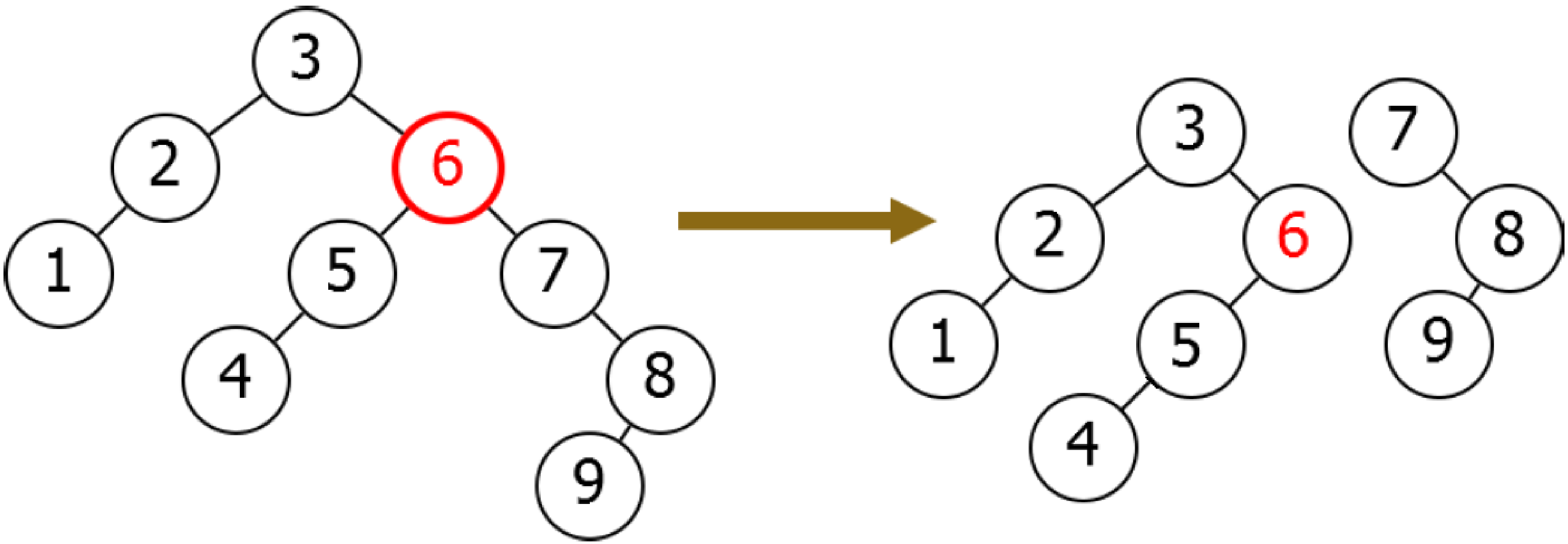
```
else if R₁.height > R₂.height:
    R′ ← AVLTreeMergeWithRoot(R₁.right, R₂, T)
    R₁.right ← R′
    R′.parent ← R₁
    Rebalance(R₁)
    return root

else if R₁.height < R₂.height:
    … (homework) …
```

Analysis

- Each step changes height difference by 1
- Eventually within 1
- Time $O(|R_1.\text{height} - R_2.\text{height}| + 1)$

Split

- Break tree into two trees

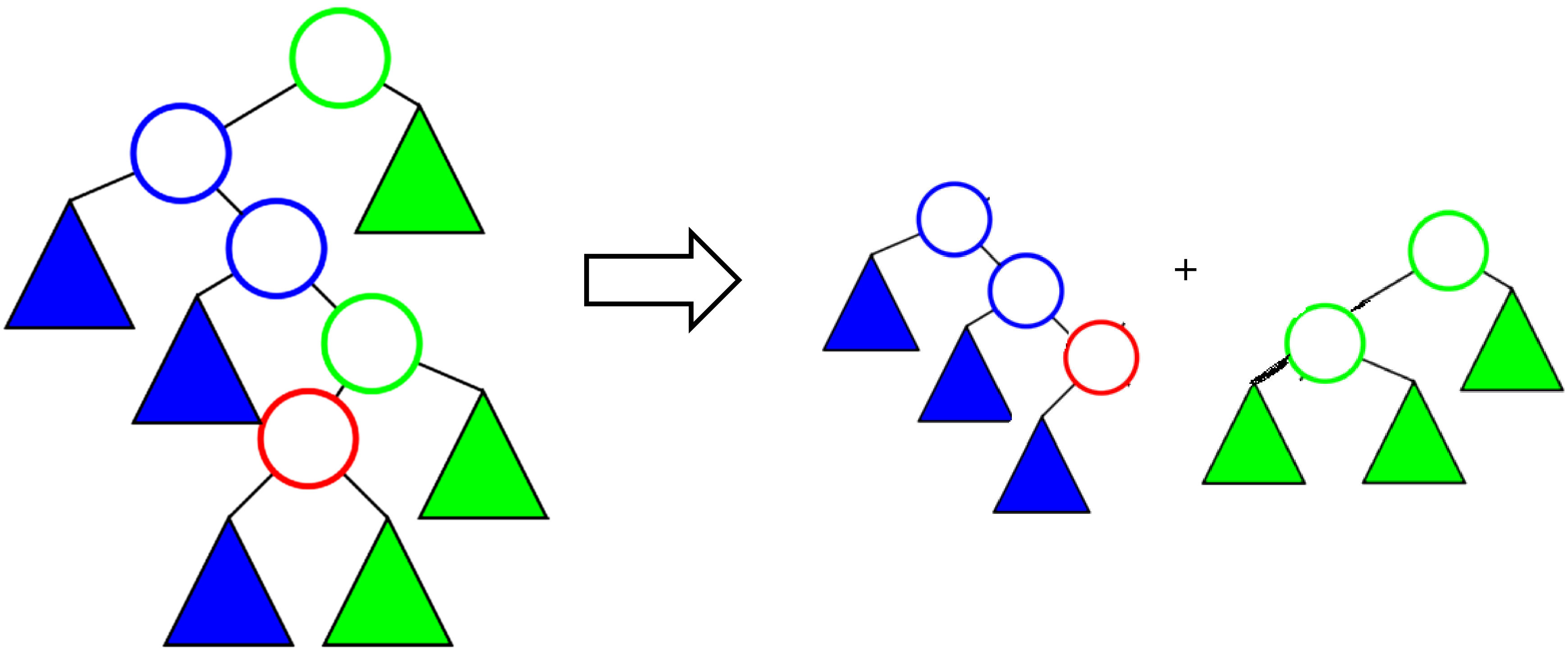


Formal Definition

Split
Input: Root **R** of a tree, key x Output: Two trees (List), one with elements ≤ x, one with elements > x

Idea

- Search for x and split the trees along the search path
- Merge left subtrees (including the node) into one tree
- Merge right subtrees into another



Implementation

Split(R, x)

```
if R is null
    return (null, null)
else if x < R.key:
    (R1, R2) ← Split(R.left, x)
    R3 ← MergeWithRoot(R2, R.right, R)
    return (R1, R3)
else if x ≥ R.key
    (R1, R2) ← Split(R.right, x)
    R4 ← MergeWithRoot(R.left, R1, R)
    return (R4, R2)
```

AVL Trees

- Using AVLMergeWithRoot maintains balance
- Time = O(log n)

Summary

- Merge combines trees.
- Split turns one tree into two.
- Both can be implemented in O(log n) time for AVL tree