Binary Search Trees: Split and Merge

Daniel Kane

Department of Computer Science and Engineering
University of California, San Diego

Data Structures
Data Structures and Algorithms

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.

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.

Outline

1. Merge

2. Split

Merge

In general, to merge two sorted lists takes $O(n)$ time. However, when they are separated it is faster.

Merge

In general, to merge two sorted lists takes $O(n)$ time. However, when they are separated it is faster.

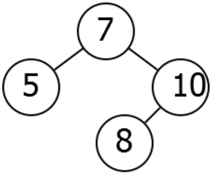
Merge

Input: Roots R_1 and R_2 of trees with all keys in R_1's tree smaller than those in R_2's

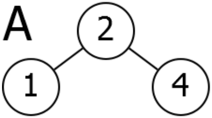
Output: The root of a new tree with all the elements of both trees

Problem

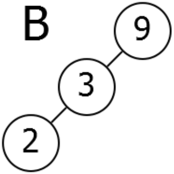
Which tree can be merged with the given one?



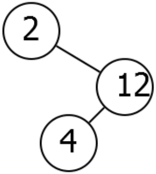
A



B

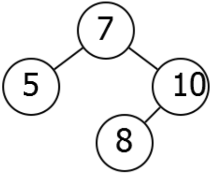


C

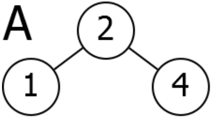


Problem

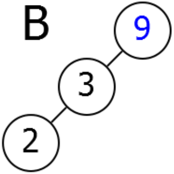
Which tree can be merged with the given one?



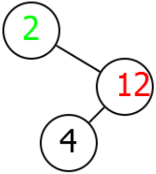
A



B

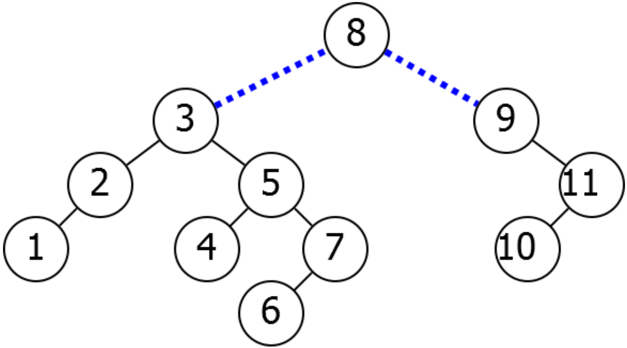


C



Extra Root

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



Implementation

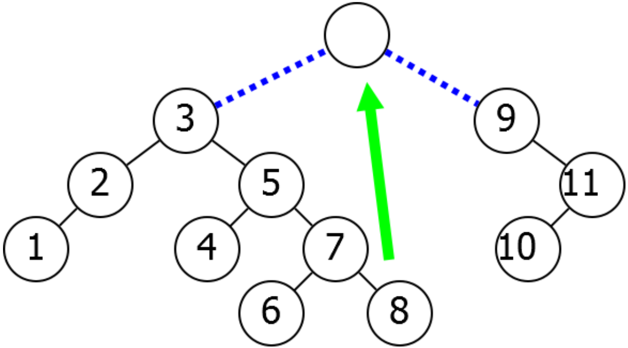
`MergeWithRoot`(R_1, R_2, T)

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 left subtree.



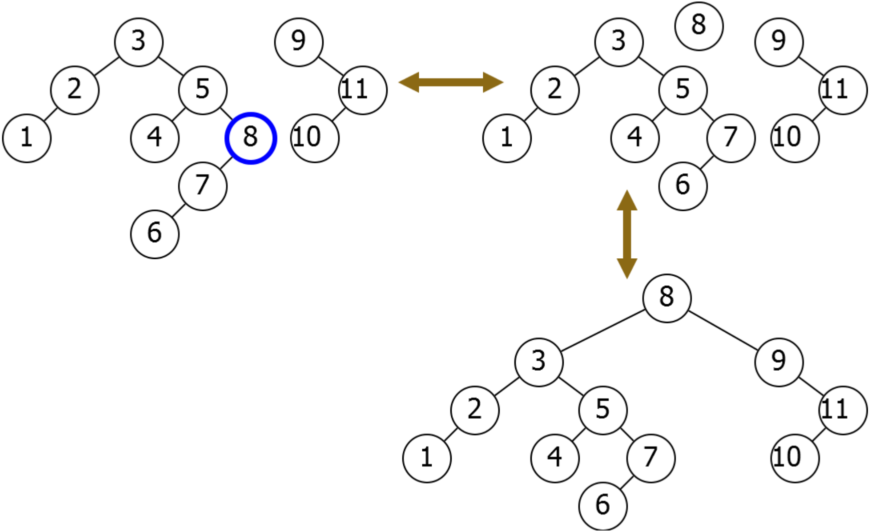
Merge

Merge(R_1, R_2)

$T \leftarrow$ Find(∞, R_1)
Delete(T)
MergeWithRoot(R_1, R_2, T)
return T

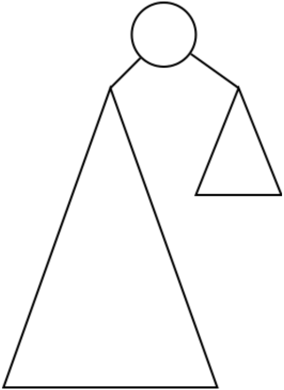
Time $O(h)$.

Merge



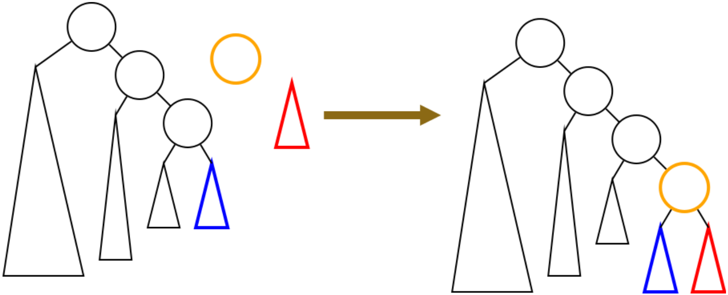
Balance

Unfortunately, this merge does not preserve balance properties.



Idea

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



Implementation

$\texttt{AVLTreeMergeWithRoot}(R_1, R_2, T)$

if $|R_1.\texttt{Height} - R_2.\texttt{Height}| \leq 1$:
  $\texttt{MergeWithRoot}(R_1, R_2, T)$
  $T.\texttt{Ht} \leftarrow \max(R_1.\texttt{Height}, R_2.\texttt{Height}) + 1$
  return T

Implementation (continued)

$\texttt{AVLTreeMergeWithRoot}(R_1, R_2, T)$

```
else if R₁.Height > R₂.Height:
    R' ← AVLTreeMWR(R₁.Right, R₂, T)
    R₁.Right ← R'
    R'.Parent ← R₁
    Rebalance(R₁)
    return root
else if R₁.Height < R₂.Height:
    ...
```

Analysis

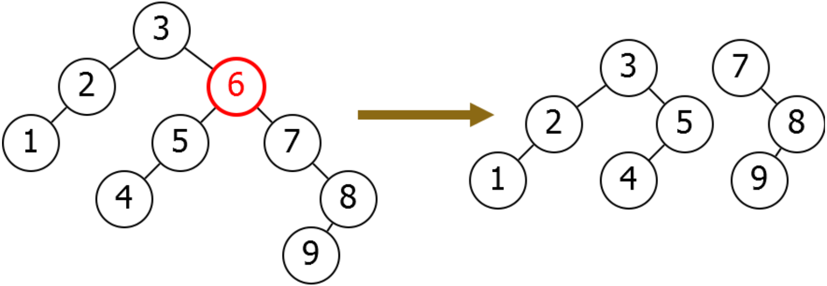
- Each step changes height difference by 1 or 2.
- Eventually within 1.
- Time $O(|R_1.\mathtt{Height} - R_2.\mathtt{Height}| + 1)$.

Outline

1. Merge
2. Split

Split

Break tree into two trees:



Formal Definition

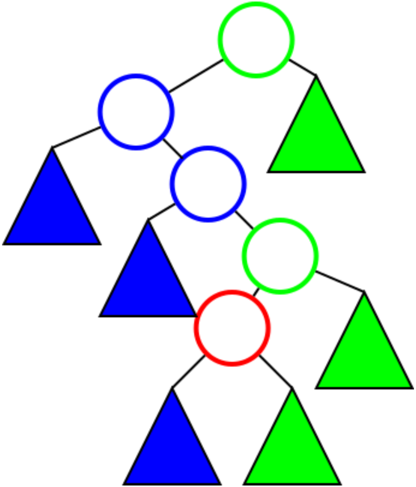
Split

Input: Root R of a tree, key x

Output: Two trees, one with elements $\leq x$, one with elements $> x$.

Idea

Search for x, merge subtrees.



Implementation

$\texttt{Split}(R, x)$

```
if R = null:
    return (null, null)
if x ≤ R.Key:
    (R1, R2) ← Split(R.Left, x)
    R3 ← MergeWithRoot(R2, R.Right, R)
    return (R1, R3)
if x > R.Key:
    ...
```

AVL Trees

- Using `AVLMergeWithRoot` maintains balance.
- Time $= \sum O(|h_i - h_{i+1}| + 1) = O(h_{max}) = O(\log(n))$.

Conclusion

Summary

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