CS151 Intro to Data Structures

Balanced Search Trees, AVL Trees

Announcements

HW 7 and Lab9 (Hash Maps) due Sunday

Outline

Sorting review

Balanced BSTs

Merge sort

Example

6	8	4	1	7	2	5	3

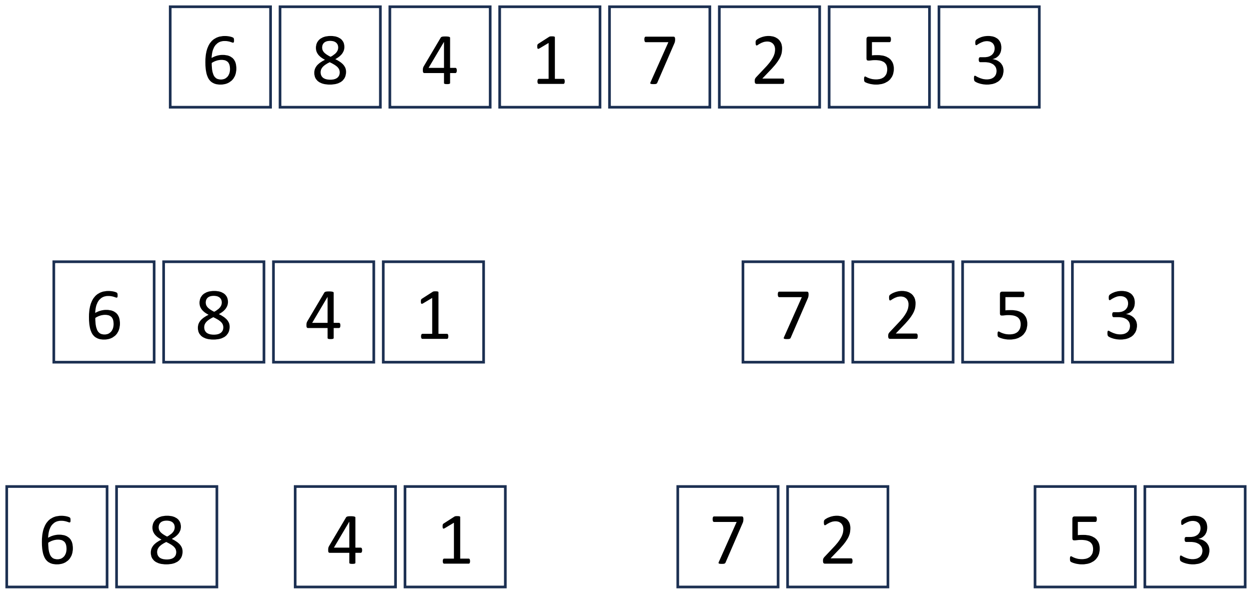
Example

6	8	4	1	7	2	5	3

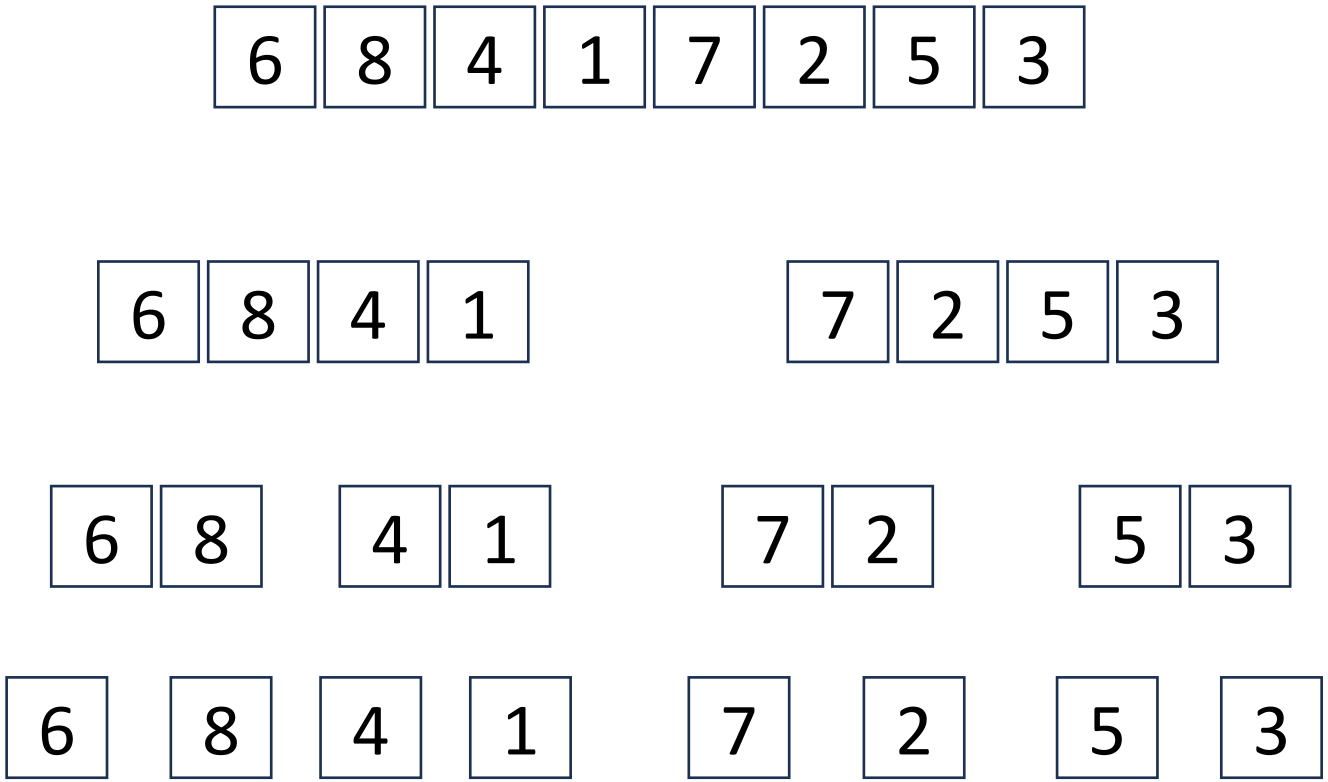
6	8	4	1

7	2	5	3

Example



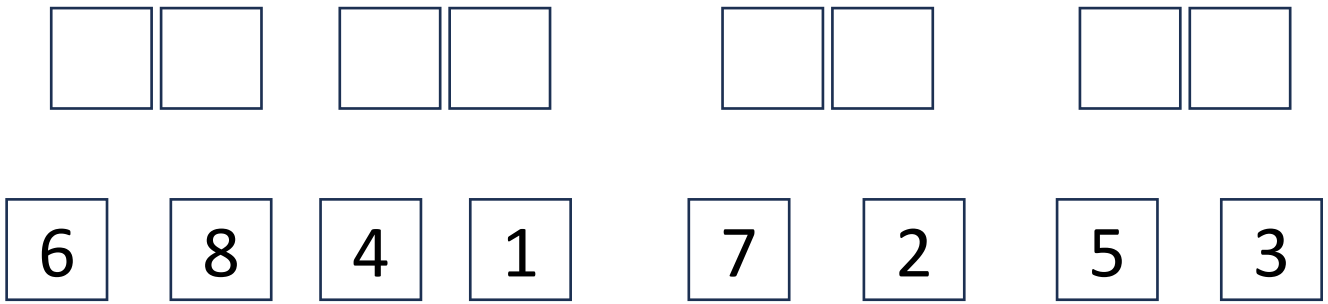
Example



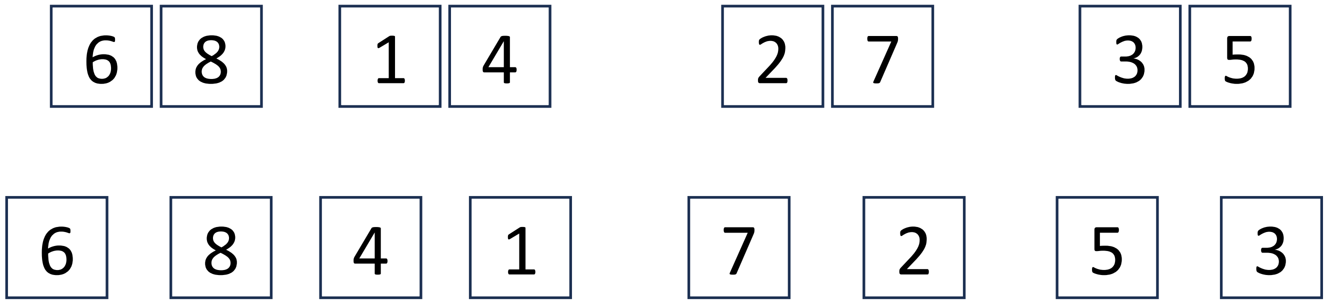
Example

6 8 4 1 7 2 5 3

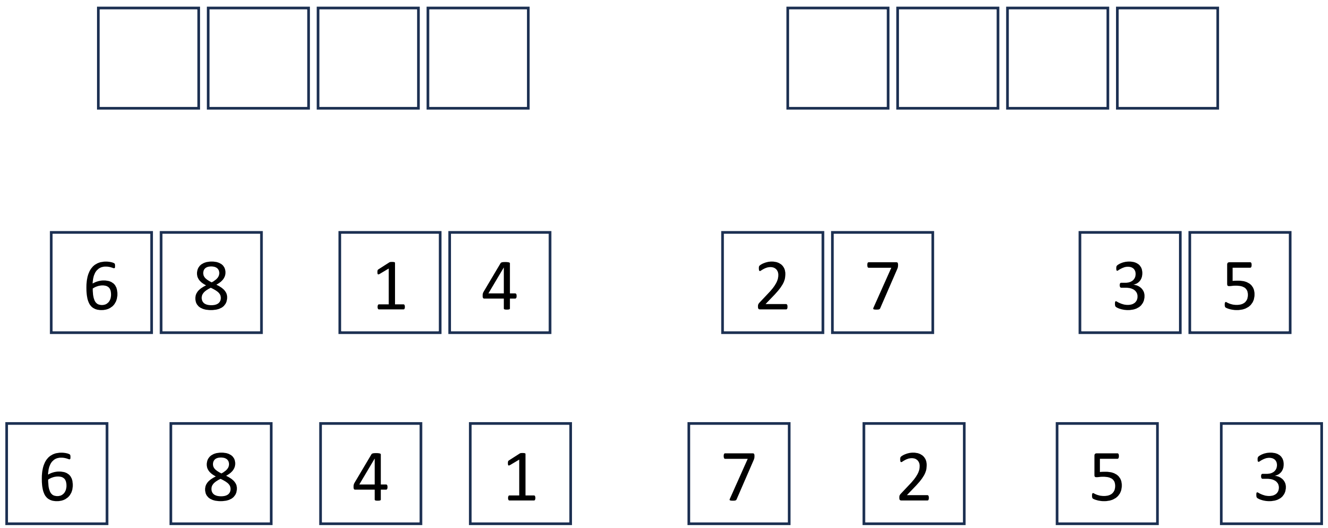
Example



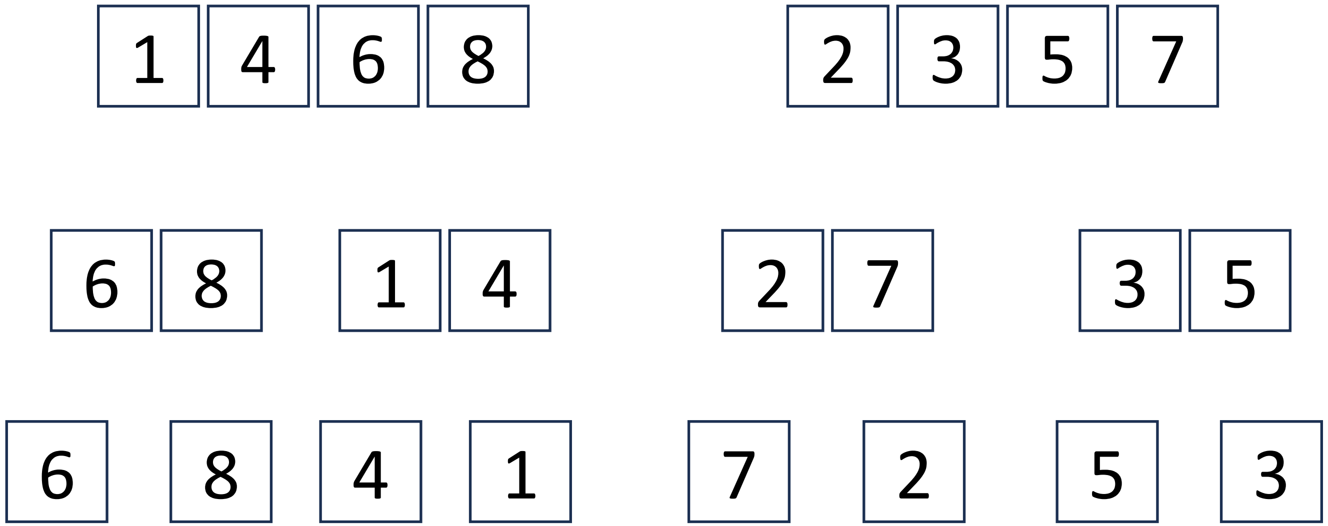
Example



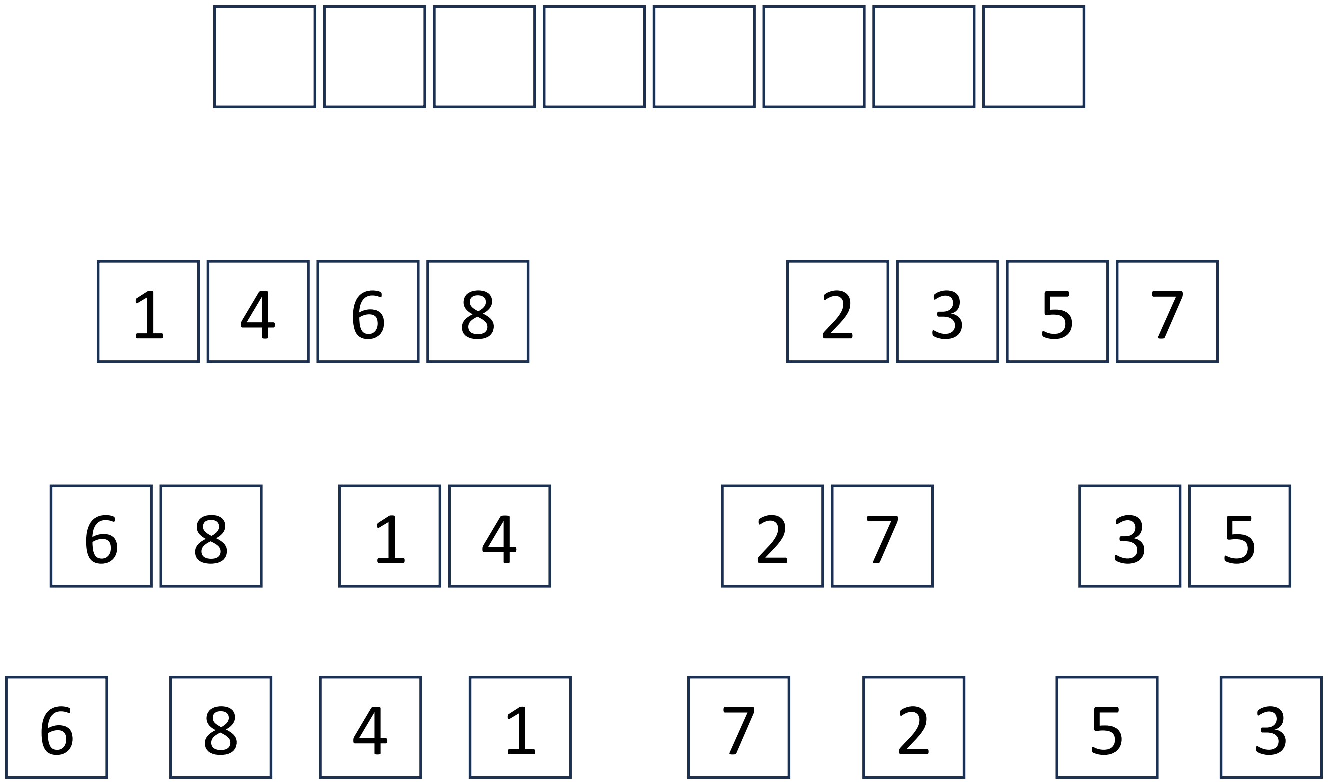
Example



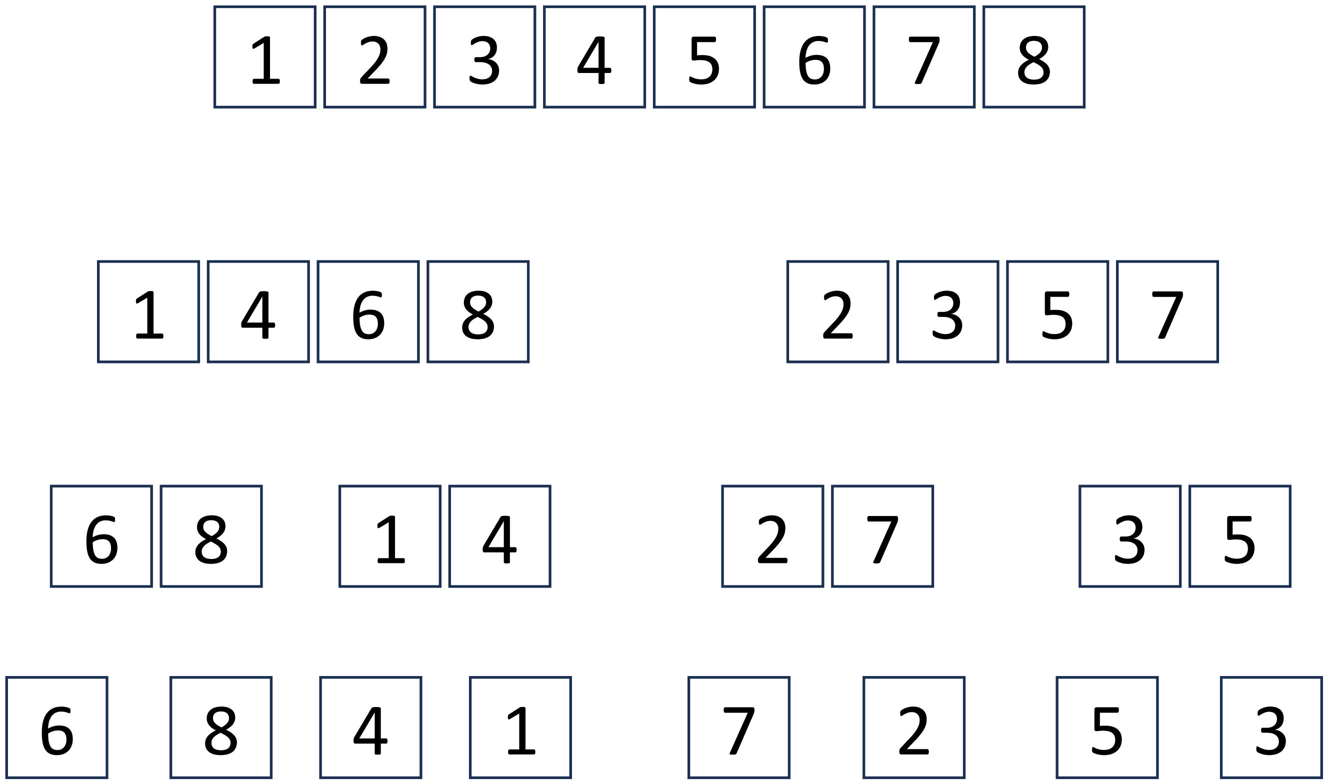
Example



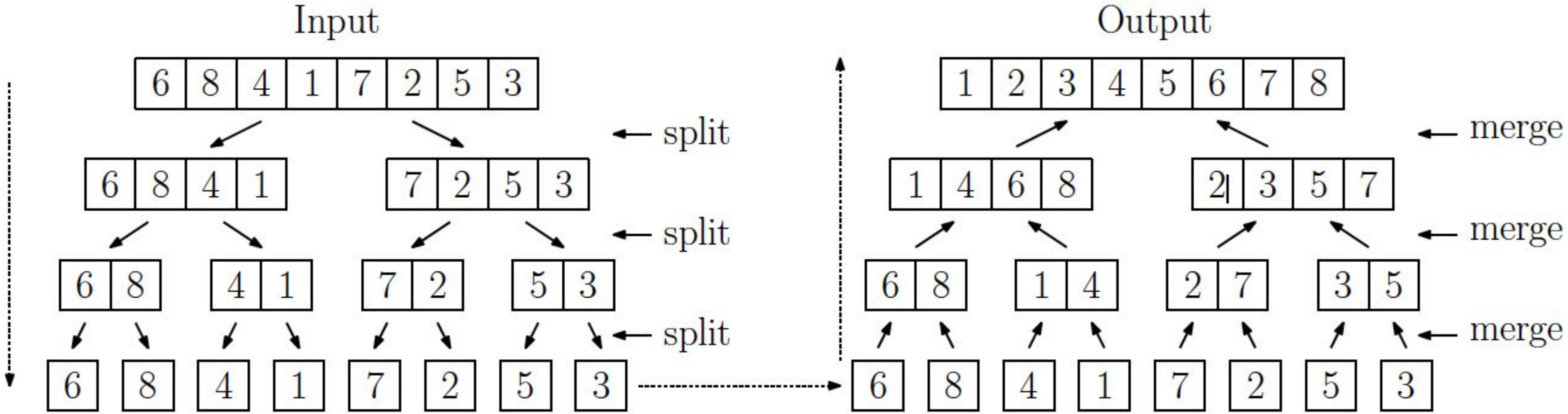
Example



Example



Example - summary



Merge - how do we sort two sorted lists?

```
Algorithm merge(A, B)
  S = []

  while(!A.isEmpty() and !B.isEmpty())
   if A[0] < B[0]
      S.add(A.removeFirst())
   else
      S.add(B.removeFirst())

   while (!A.isEmpty())
       S.add(A.removeFirst())
   while (!B.isEmpty())
      S.add(B.removeFirst())
   return S
```

runtime complexity?

O(n)

where n is A.length + B.length

Merge Sort Implementation

Runtime of MergeSort

Runtime of merging two sorted two lists A, B where |A| + |B| = n :

O(n)

How many times do we merge two sorted lists?

log n times

So total runtime is:

O(n * log(n))

Quicksort

Quicksort

- Divide and conquer
- **Divide:** select a *pivot* and create three sequences:
  a. L: stores elements less than the pivot
  b. E: stores elements equal to the pivot
  c. G: stores elements greater than the pivot
- **Conquer:** recursively sort L and G
- **Combine:** L + E + G is a sorted list

Quick Sort

Sort [2, 6, 5, 3, 8, 7, 1, 0]

1. choose a pivot
2. swap pivot to the end of the array
3. Find two items:
   a. left which is larger than our pivot
   b. right which is smaller than our pivot

4. swap left and right
5. repeat 3 and 4 until right < left
6. swap left and pivot
7. Sort L E and R recursively

Quick Sort - Choosing a pivot

What if we chose our pivot to be 1?

We want a pivot that divides our list as evenly as possible.

Median-of-three: look at the first, middle, and last elems in the array, and pick the middle element.

Quicksort runtime complexity

Bad pivot:

$O(n^2)$

Good pivot:

$O(n \log n)$

Summary of Sorting Algorithms

Algorithm	Time
selection-sort	
heap-sort	
merge-sort	
quick-sort	

Binary Search Tree Review

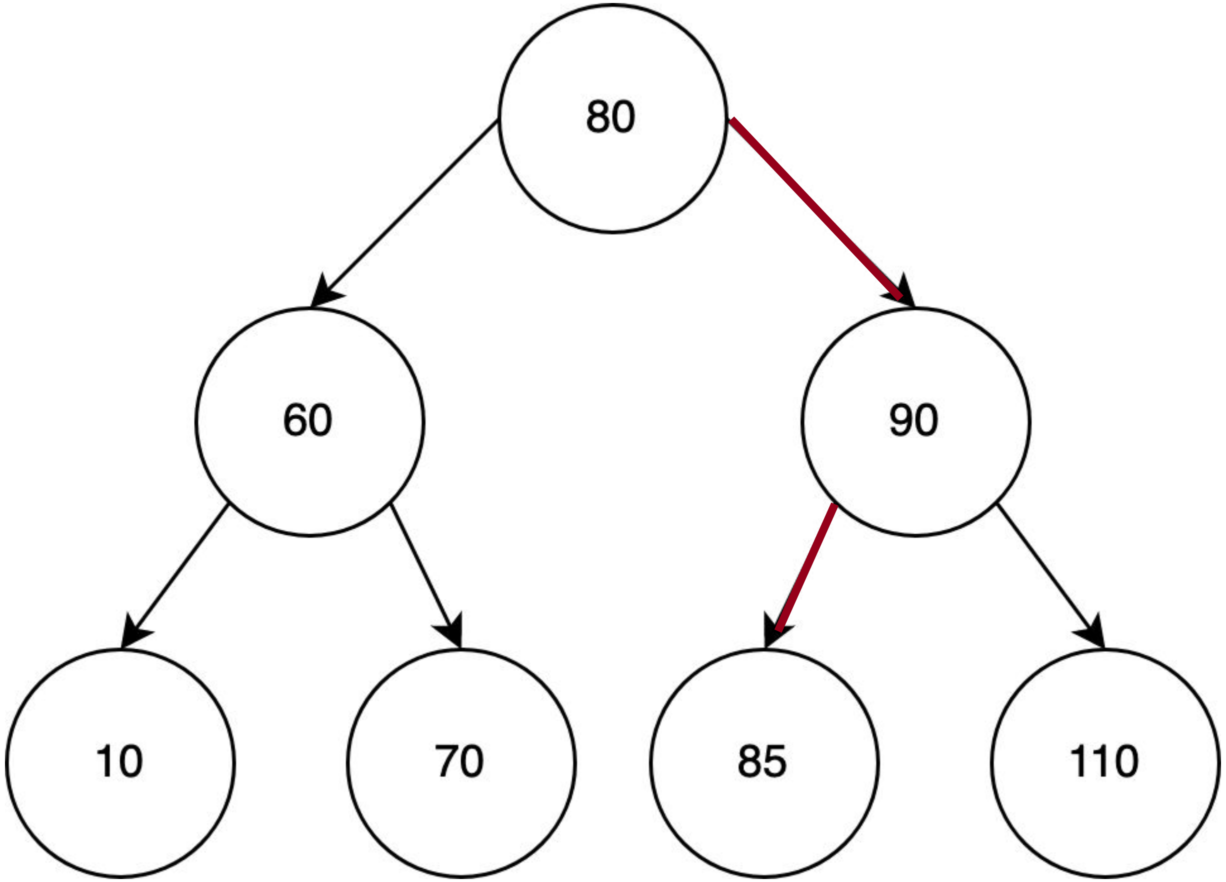
Binary Trees: Height

Height of a tree:

Maximum number of edges from a leaf node to the root

Height? 2

$\log_2(7) \approx 2$

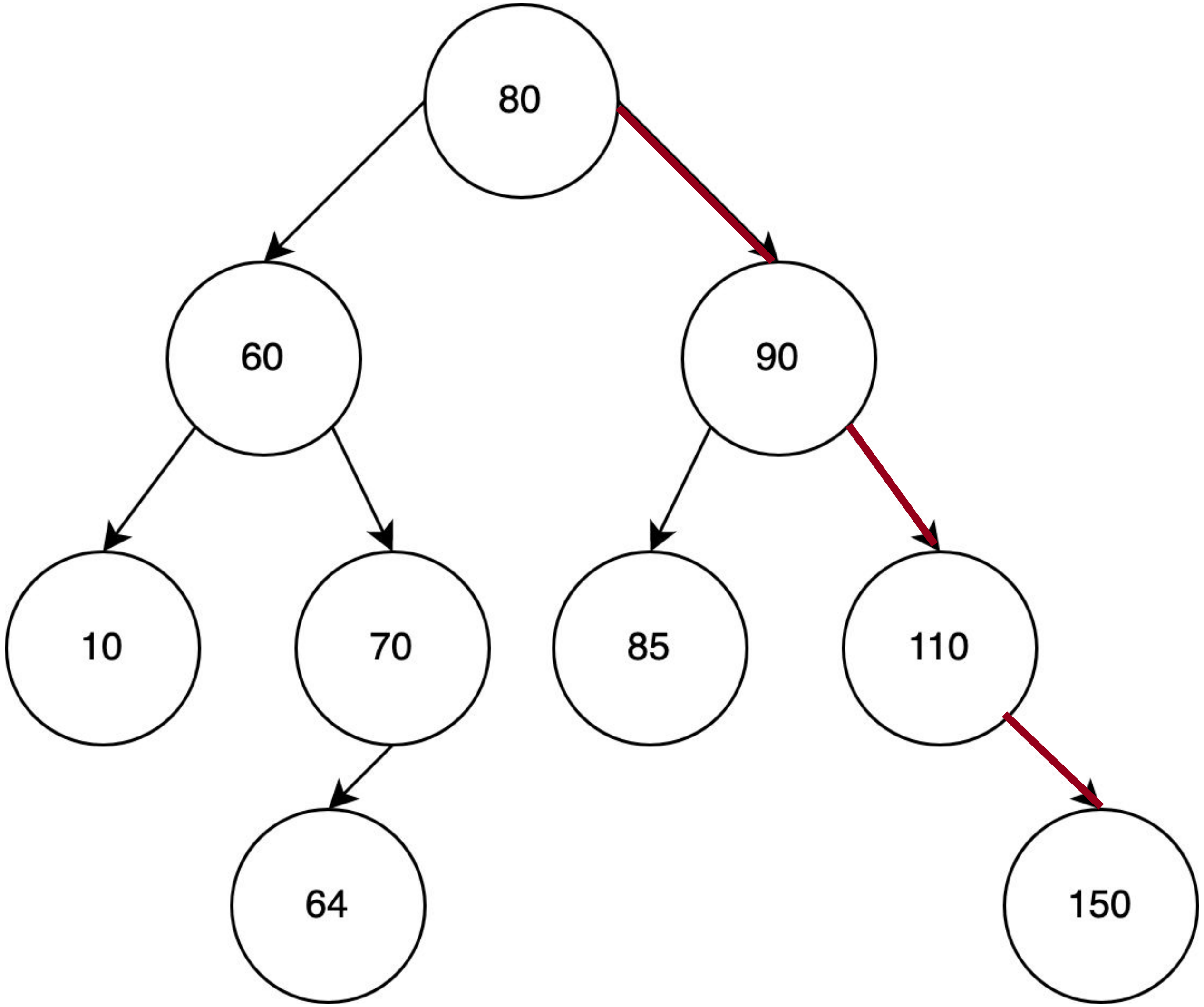


Tree Review

Height? 3

$\log_2(9) \approx 3$

Height of a binary tree is roughly log(n) where n is number of nodes



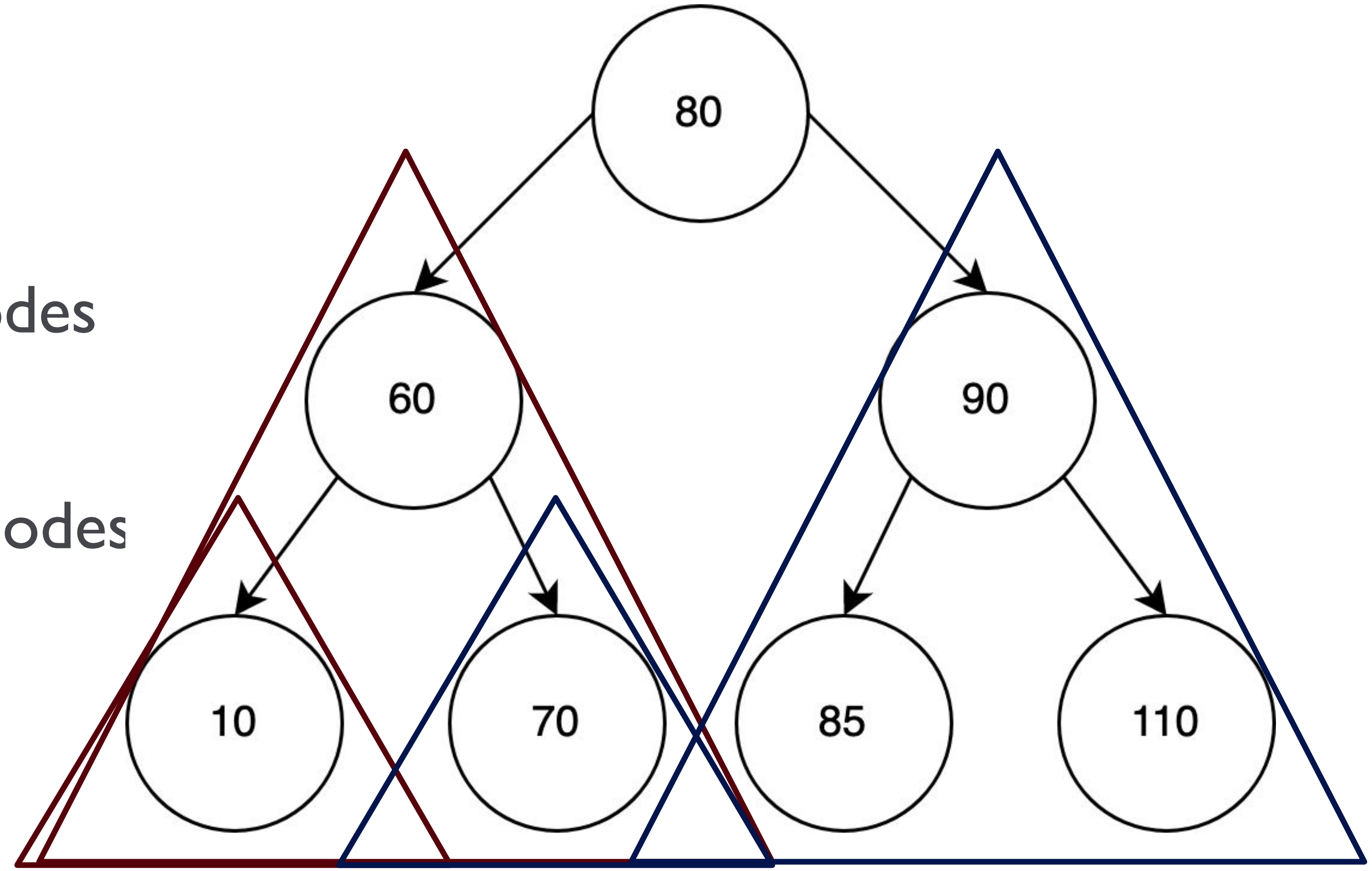
Binary Search Trees

Binary Search Trees

Definition:

At **each node** with value **k**

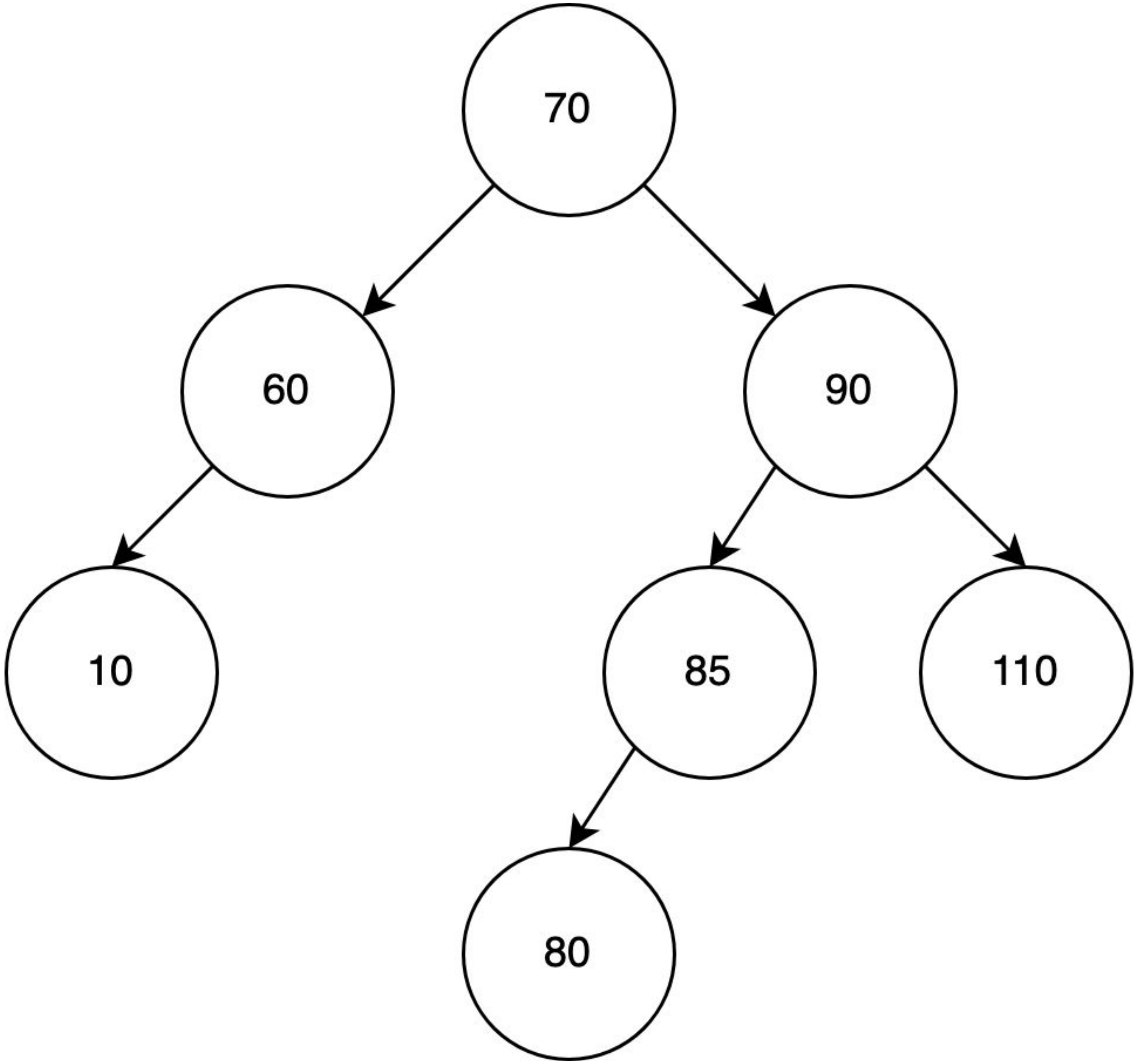
- Left subtree contains only nodes with value **lesser** than **k**
- Right subtree contains only nodes with value **greater** than **k**
- Both subtrees are a **binary search tree**



Exercise One: Binary Search Trees

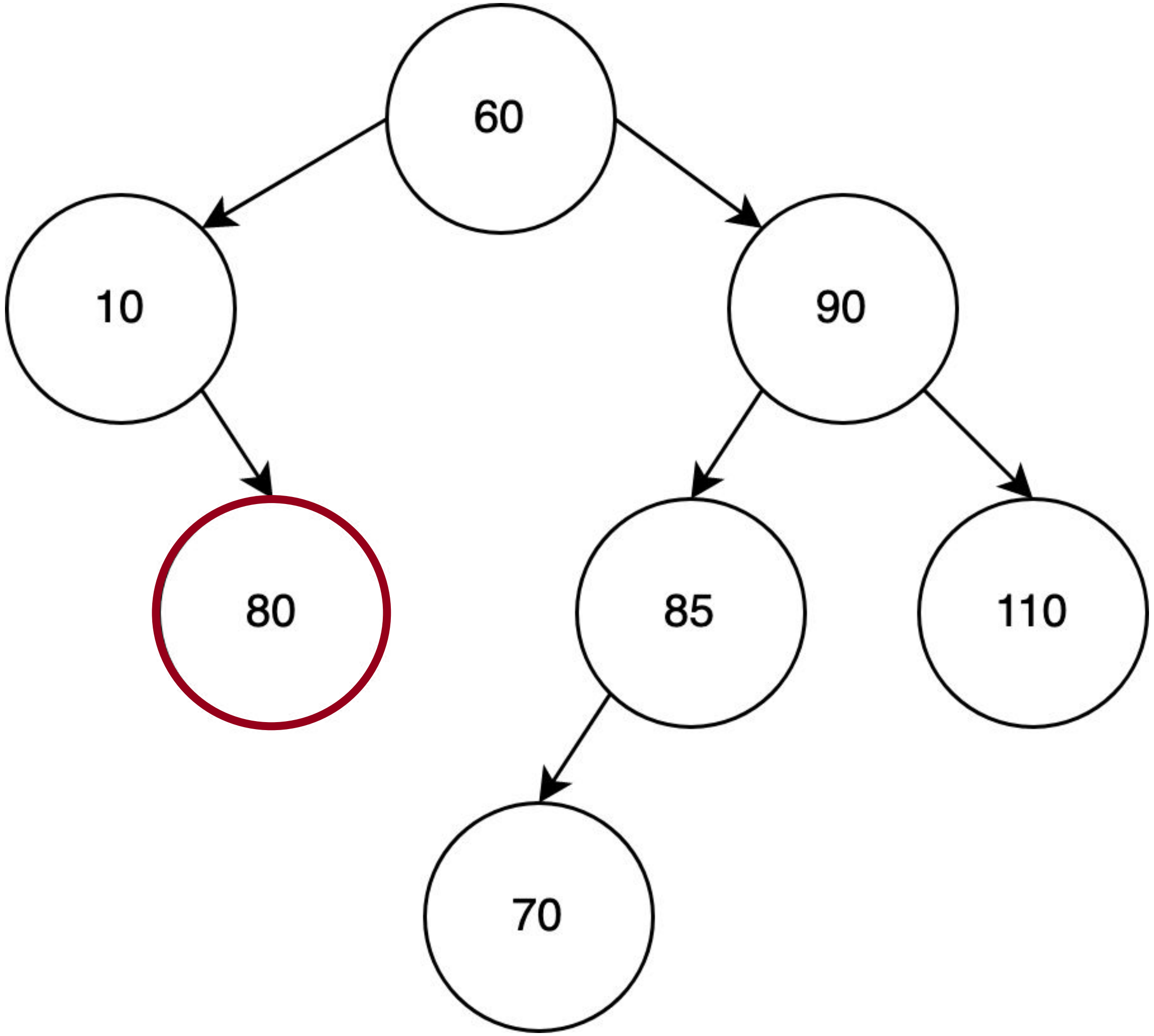
Is this a binary search tree?

✓



Exercise One: Binary Search Trees

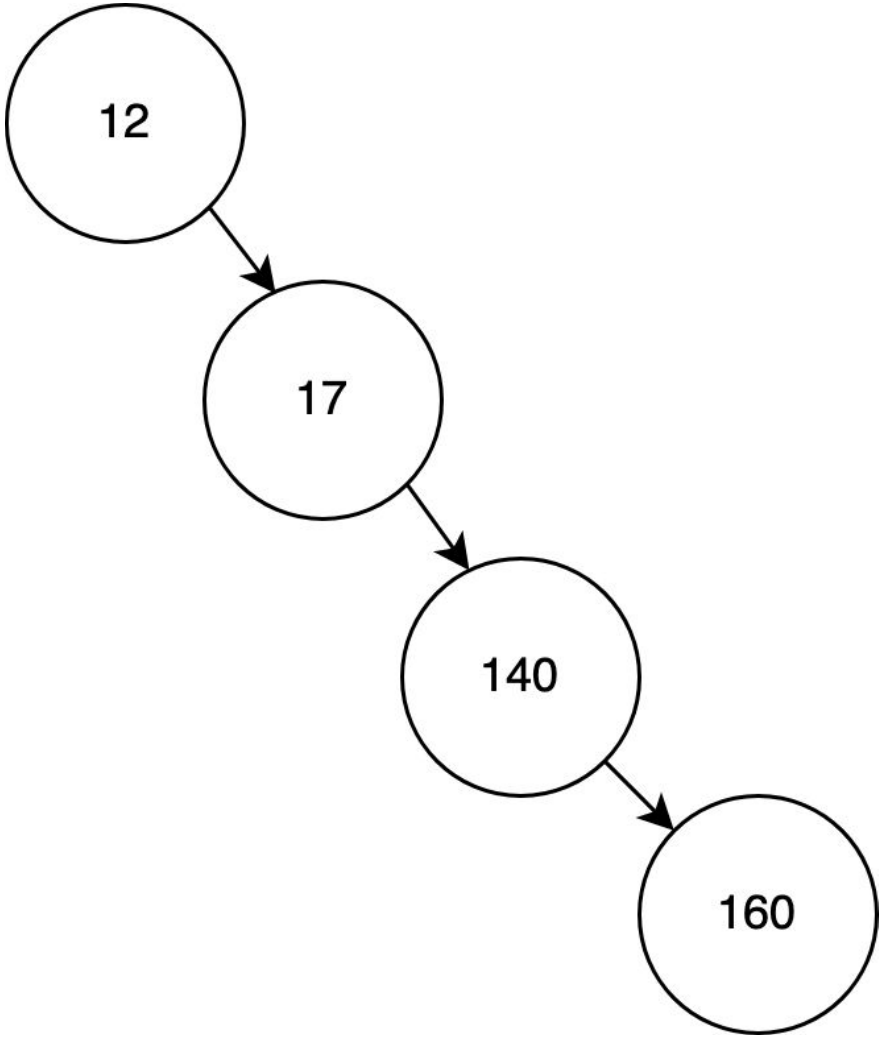
Is this a binary search tree?



Exercise One: Binary Search Trees

Is this a binary search tree?

✓



Today's Lecture

1. Binary Search Trees
2. **Search**
3. Insertion
4. Removal
5. Summary

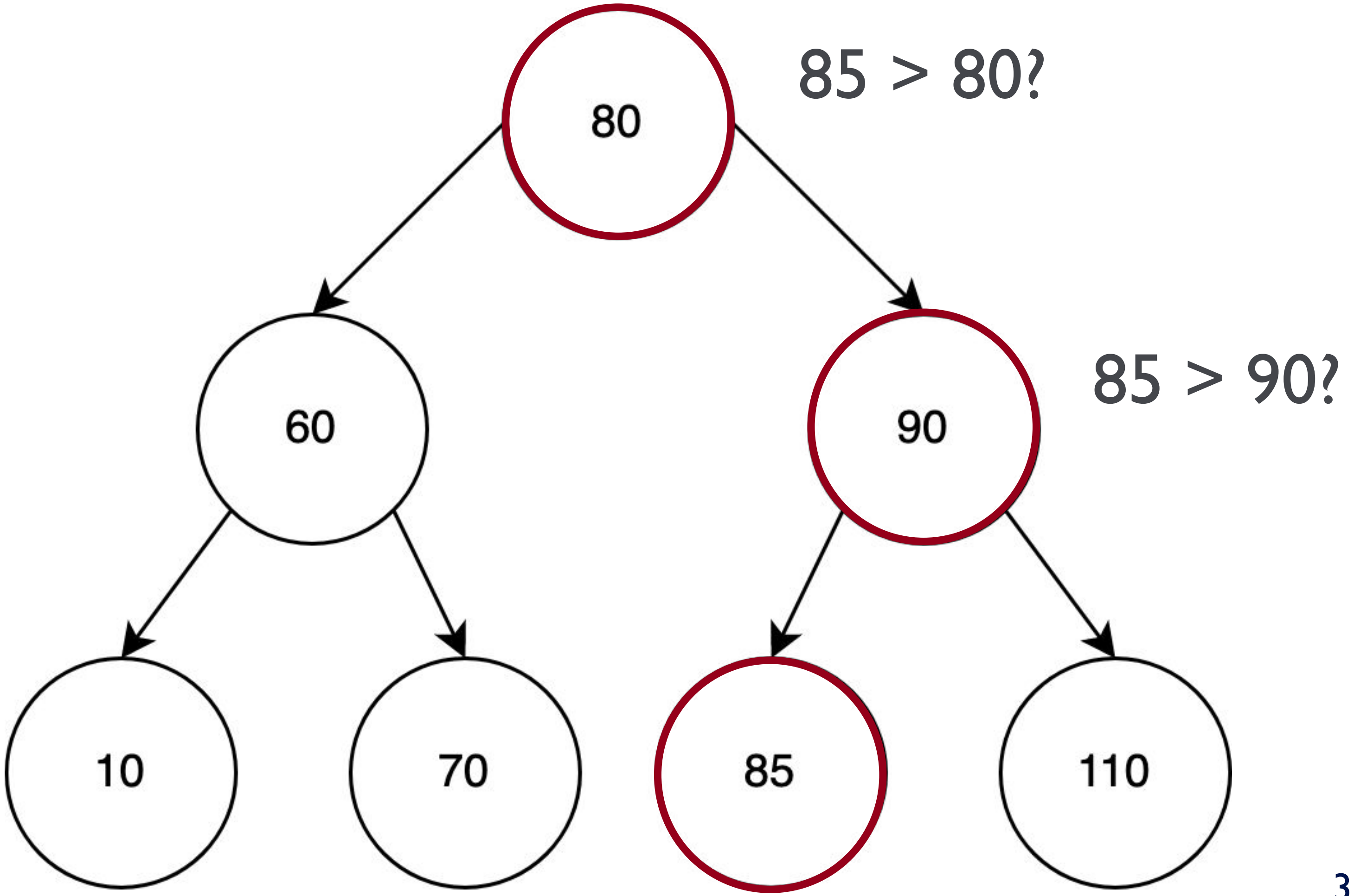
Binary Search Trees: Efficient Search

Goal: Report if a value exists in the tree

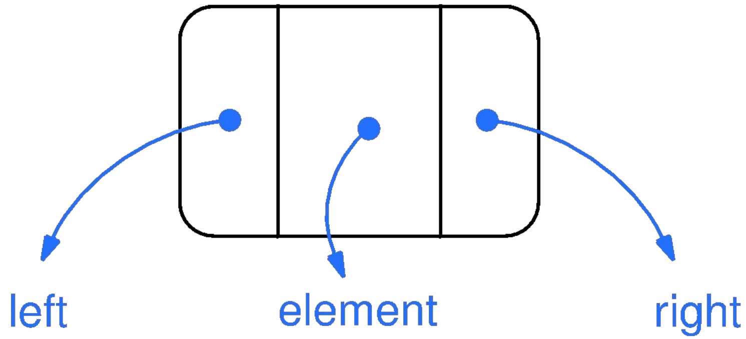
Target: 85

if **target** > **k**:
    Move right
else:
    Move Left

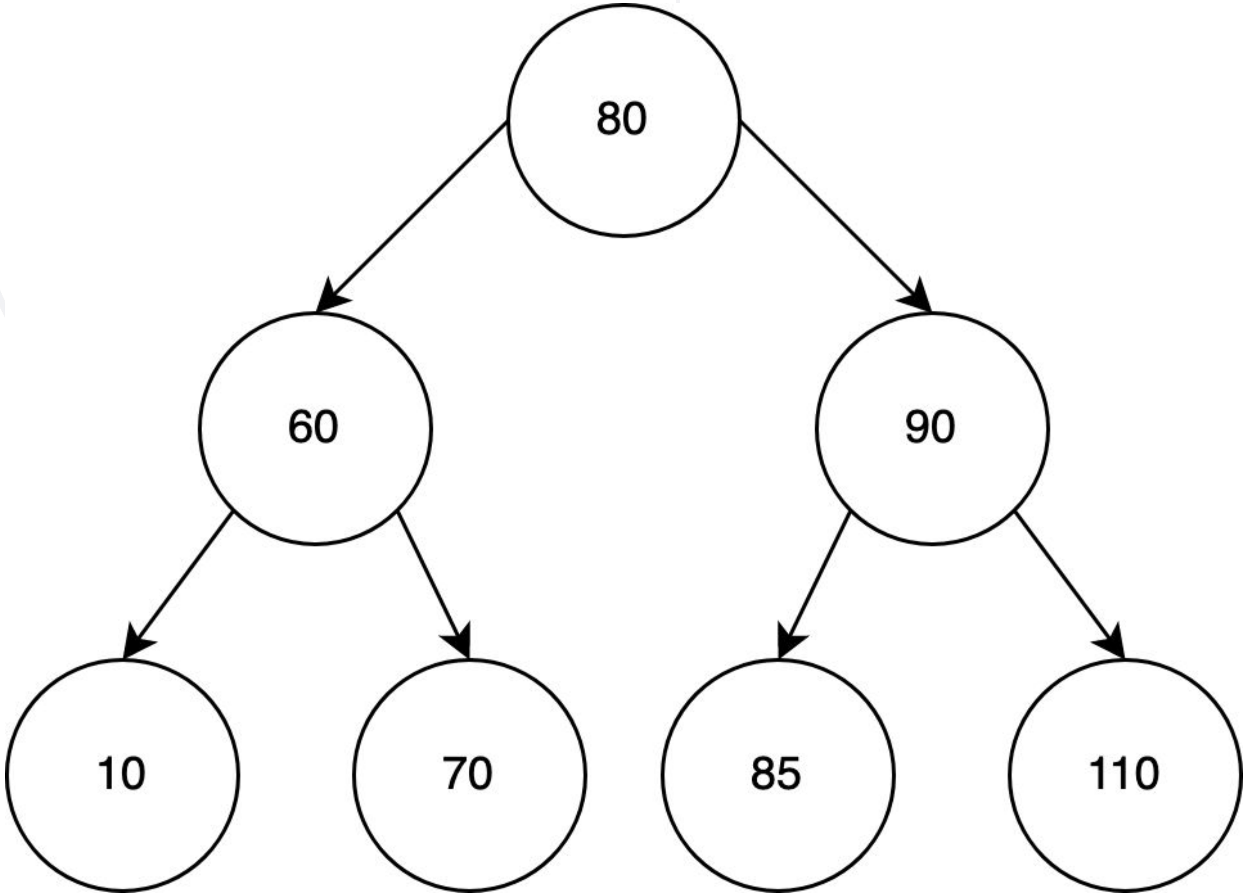
Complexity?
O(log n)



BSTs: Search Implementation



BSTs: Search Implementation



search(Node(80), 85)

    search(Node(90), 85)

        search(Node(85), 85)

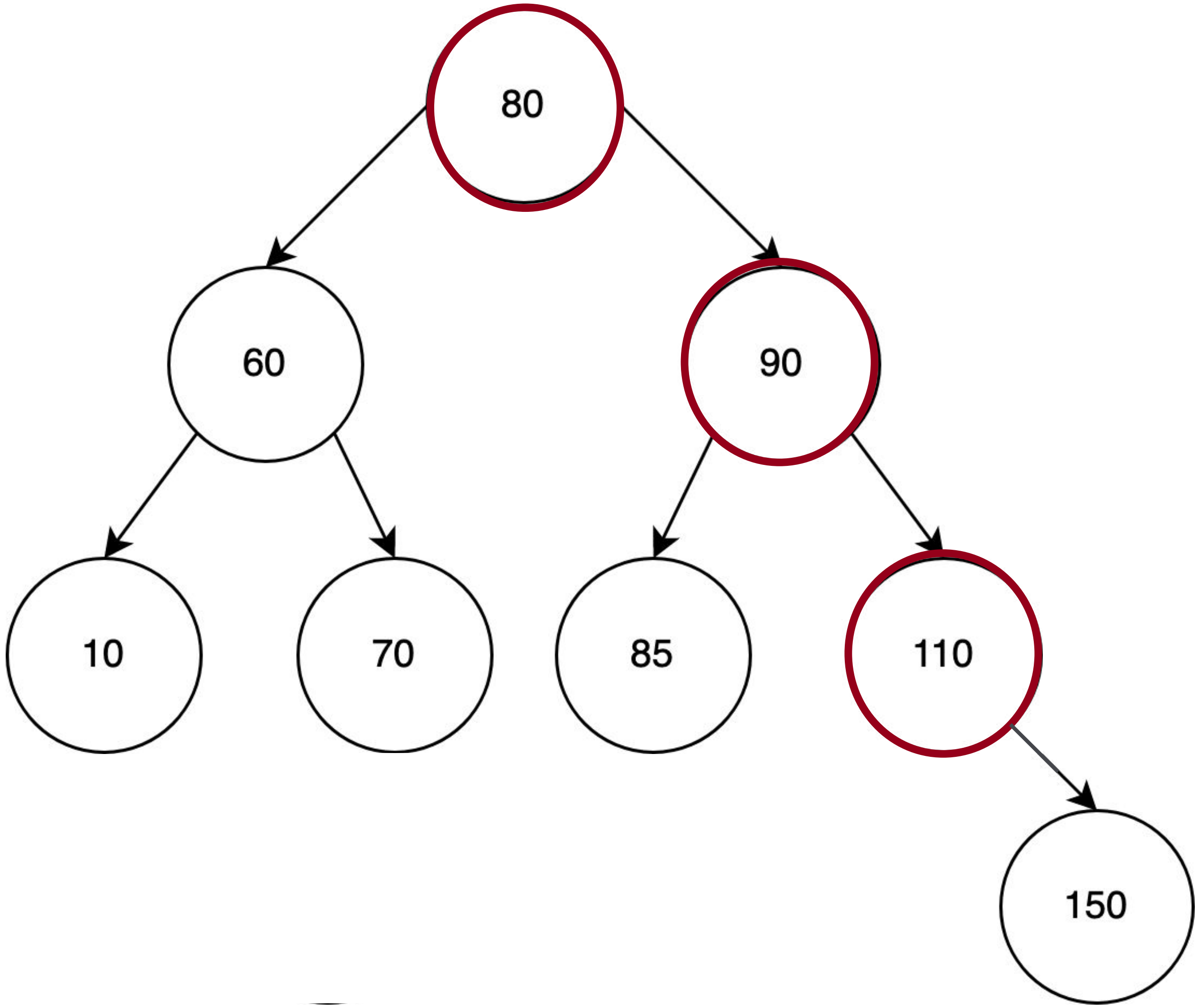
Today's Lecture

1. Binary Search Trees
2. Search
3. **Insertion**
4. Removal
5. Summary

Binary Search Trees: Insertion

Insertion must maintain the properties of a BST!

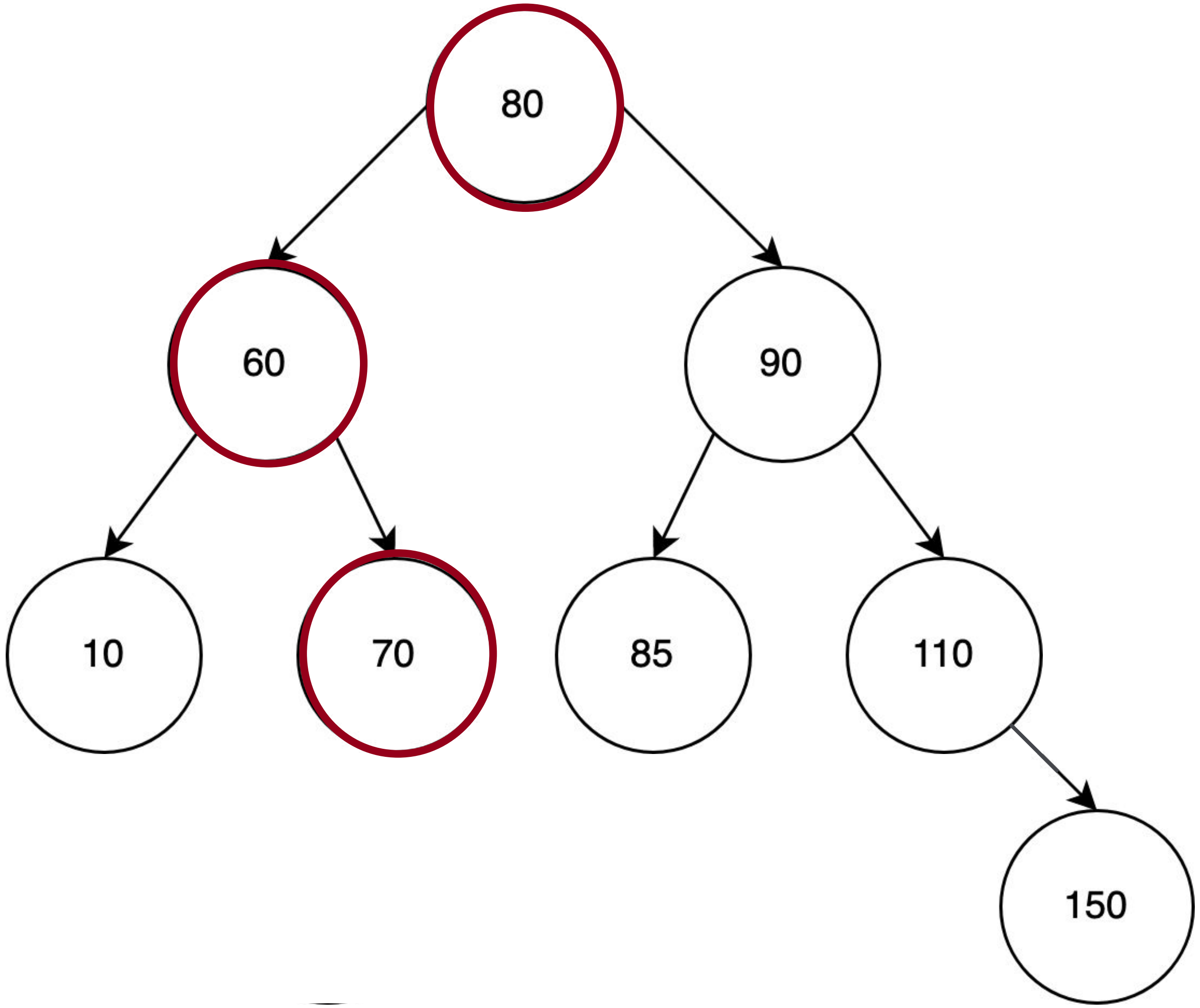
Insert: 150



Binary Search Trees: Insertion

Insertion must maintain the properties of a BST!

Insert: 64



Complexity?
O(log n)

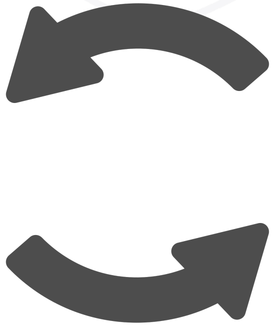
Today's Lecture

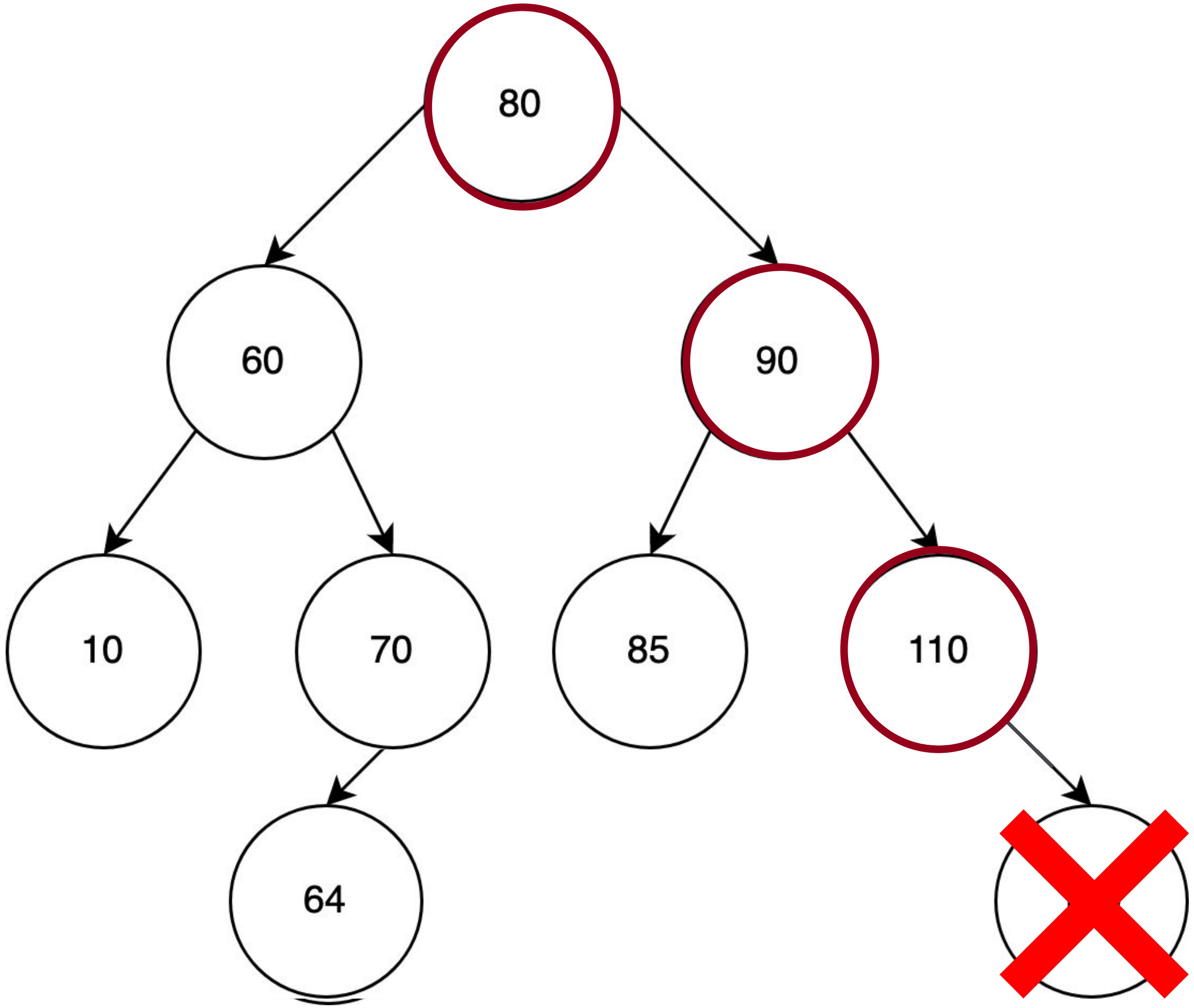
1. Binary Search Trees
2. Search
3. Insertion
4. **Removal**
5. Summary

Binary Search Trees: Deletion

Deletion must maintain the properties of a BST!

Delete: 150

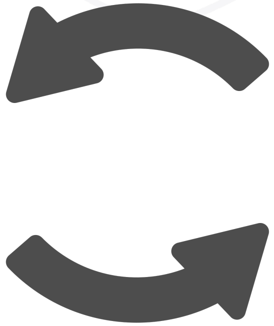


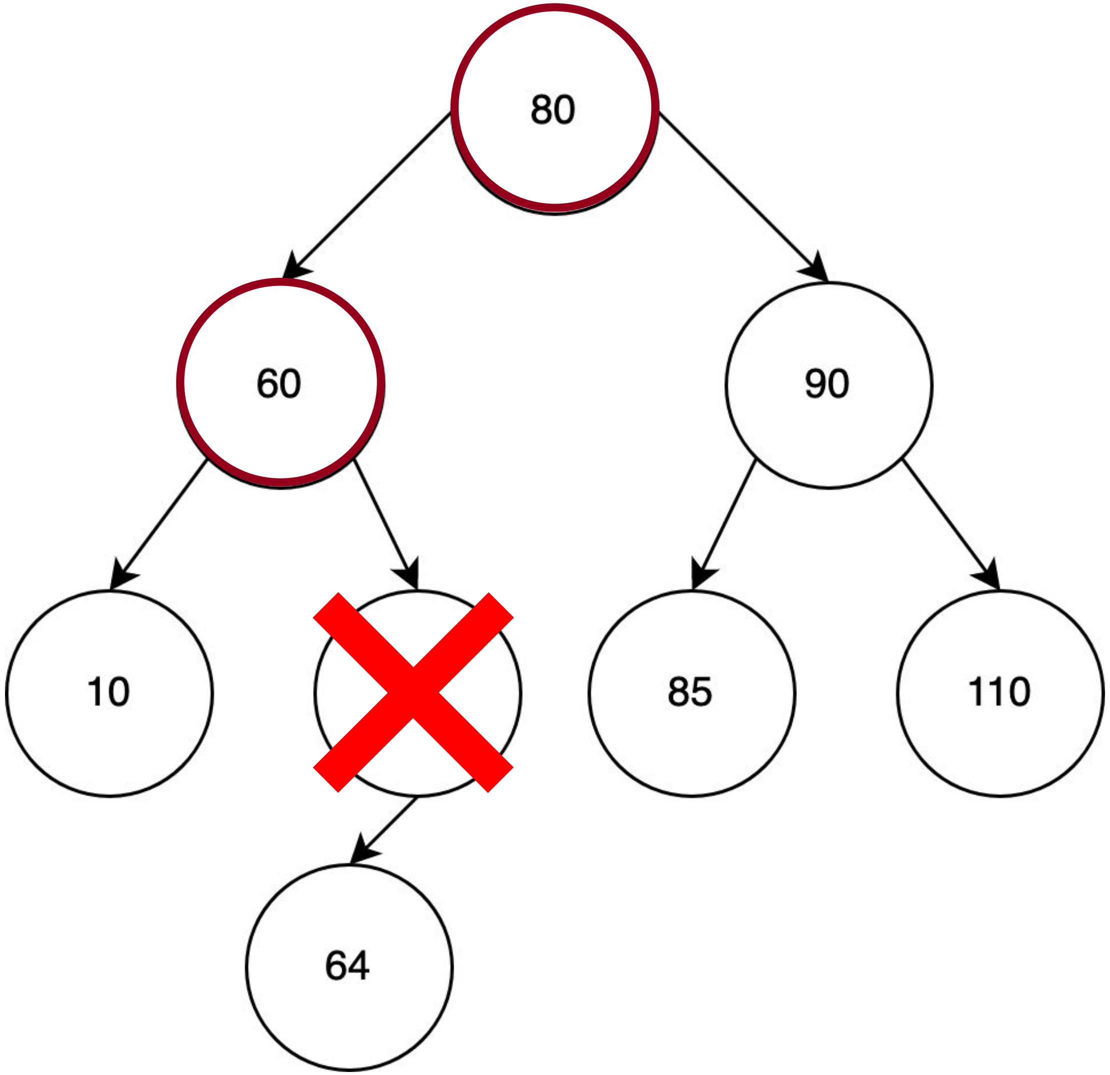


Binary Search Trees: Deletion

Deletion must maintain the properties of a BST!

Delete: <u>70</u>





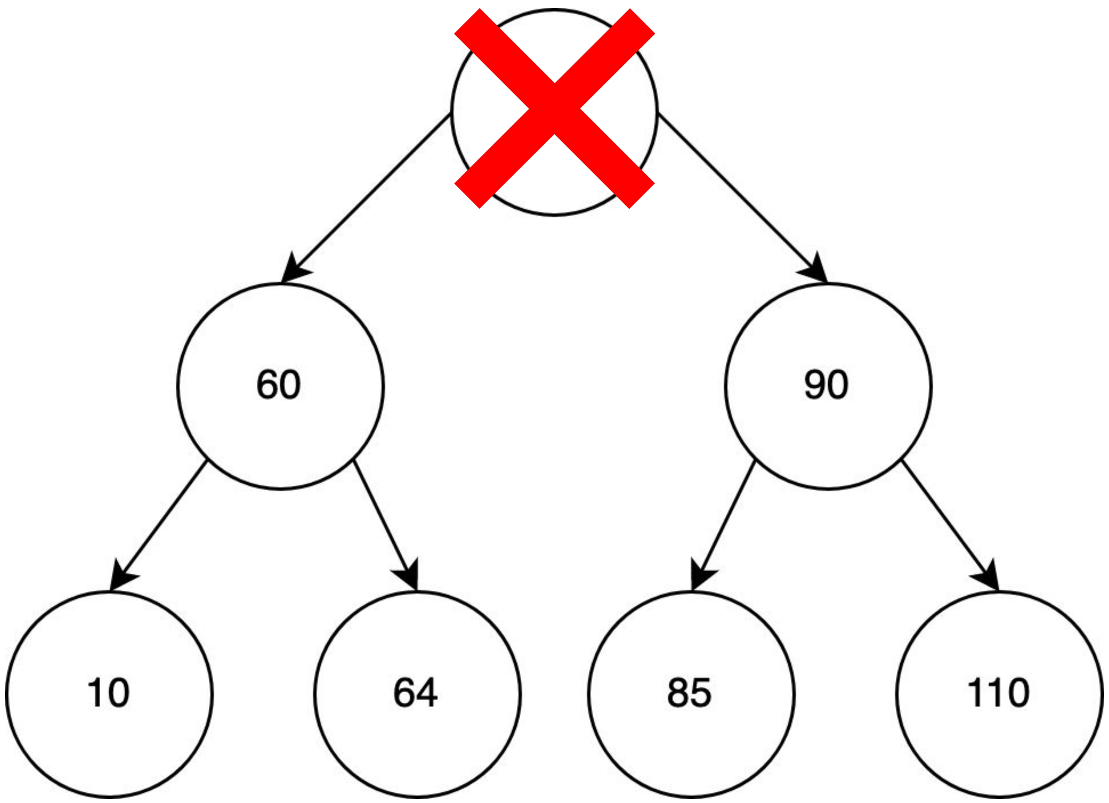
Binary Search Trees: Deletion

Deletion must maintain the properties of a BST!

Delete: 80

At **each node** with value **k**

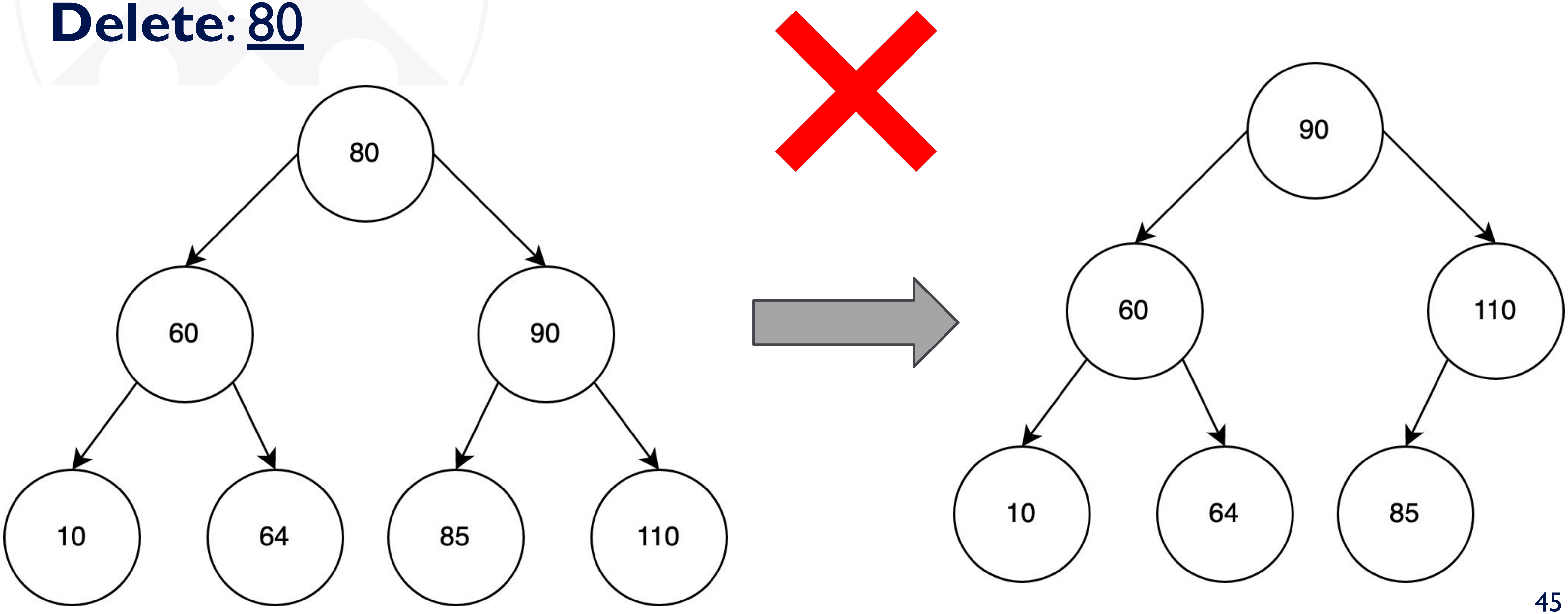
- Left subtree contains only nodes with value **lesser** than **k**
- Right subtree contains only nodes with value **greater** than **k**
- Both subtrees are a **binary search tree**



Binary Search Trees: Deletion

Replace with 90?

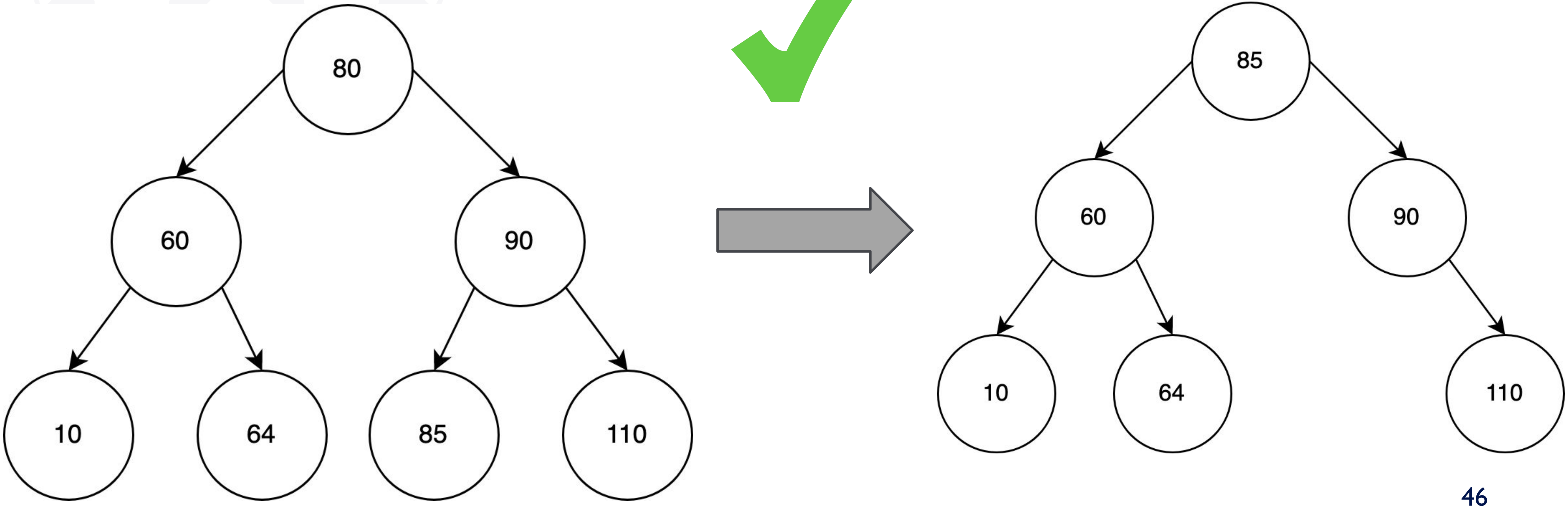
Delete: 80



Binary Search Trees: Deletion

Replace with 85?

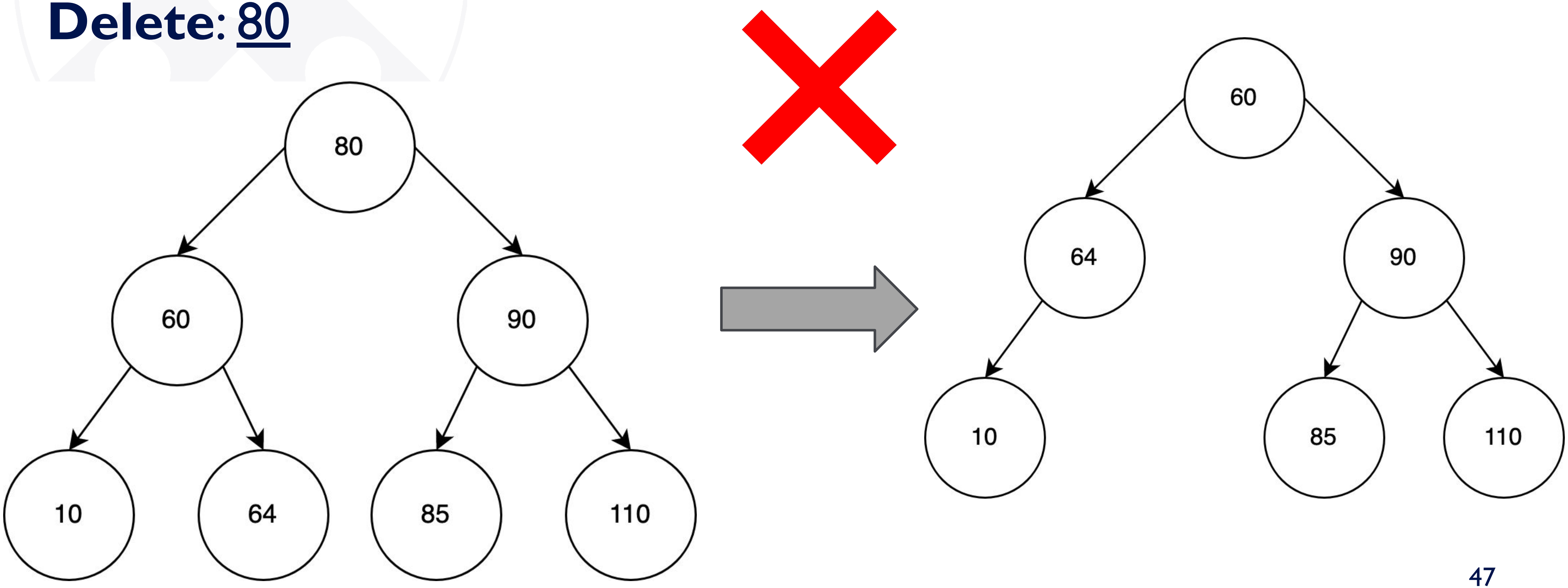
Delete: <u>80</u>



Binary Search Trees: Deletion

Replace with 60?

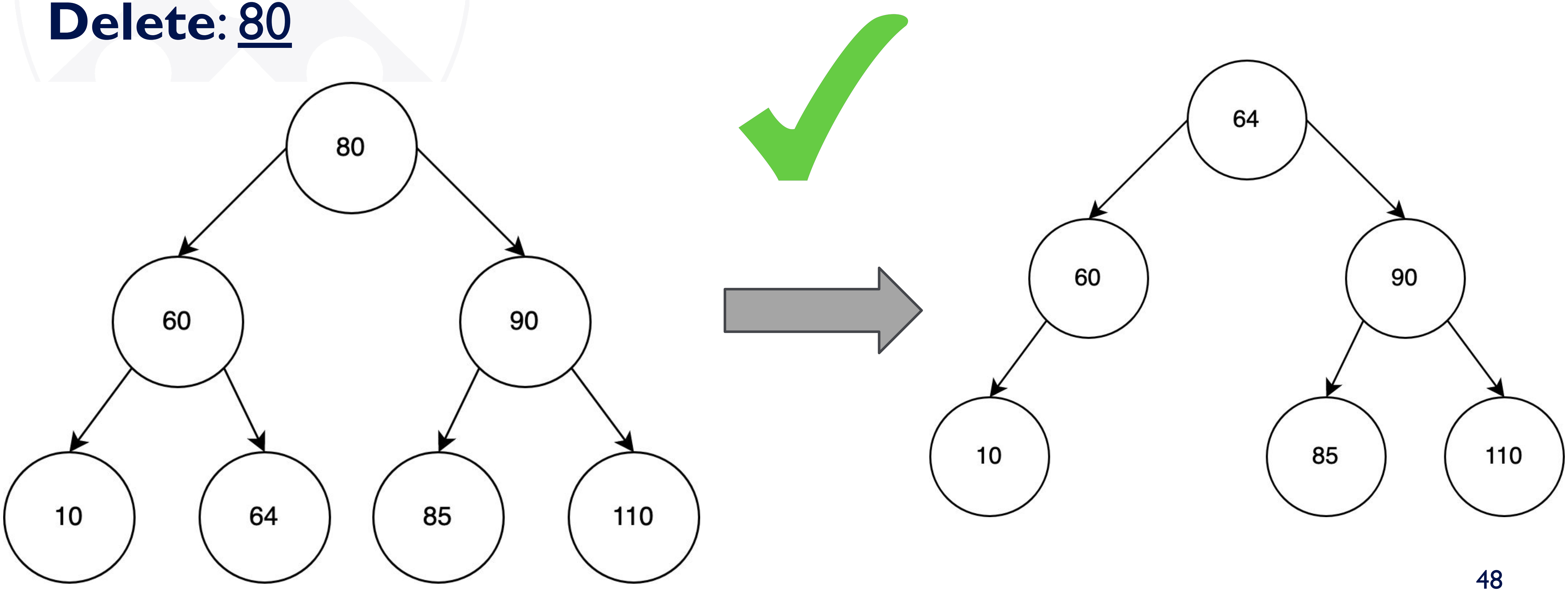
Delete: 80



Binary Search Trees: Deletion

Replace with 64?

Delete: <u>80</u>



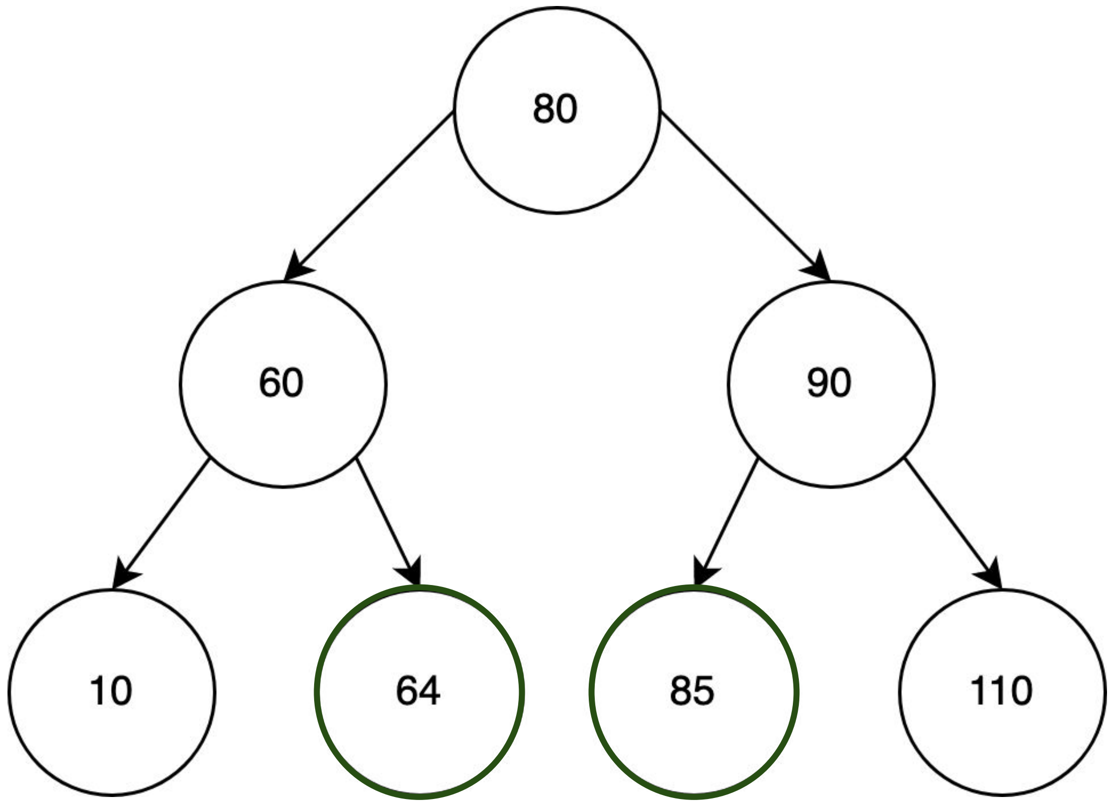
Binary Search Trees: Deletion

Deletion must maintain the properties of a BST!

Delete: 80

Replace deleted node with either:

1. Smallest value in right subtree
2. Largest value in left subtree



Binary Search Trees: Deletion

Complexity?

Case 1: Removing a **leaf node**

O(log n)

Case 2: Removing a **node with one child**

O(log n)

Case 3: Removing a **node with two children**

O(log n)

What can go wrong?

Complexity?

Search

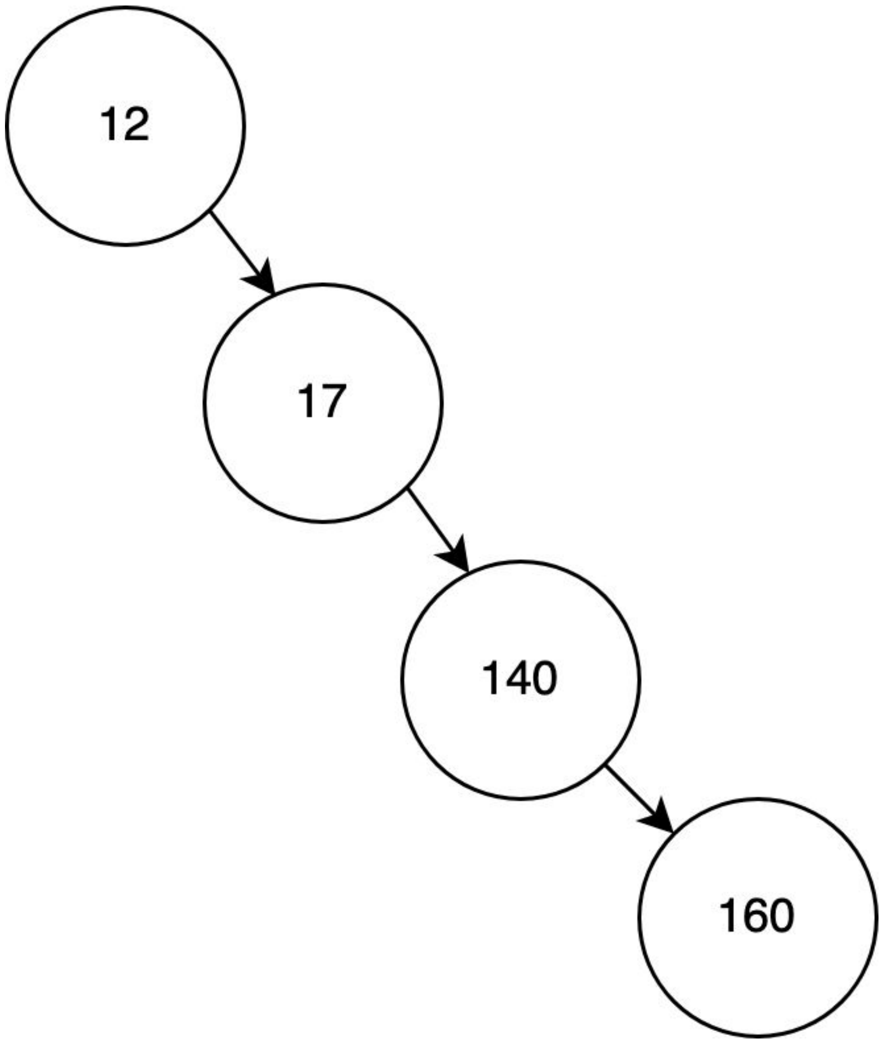
O(n)

Insertion:

O(n)

Deletion:

O(n)



Balanced Binary Trees

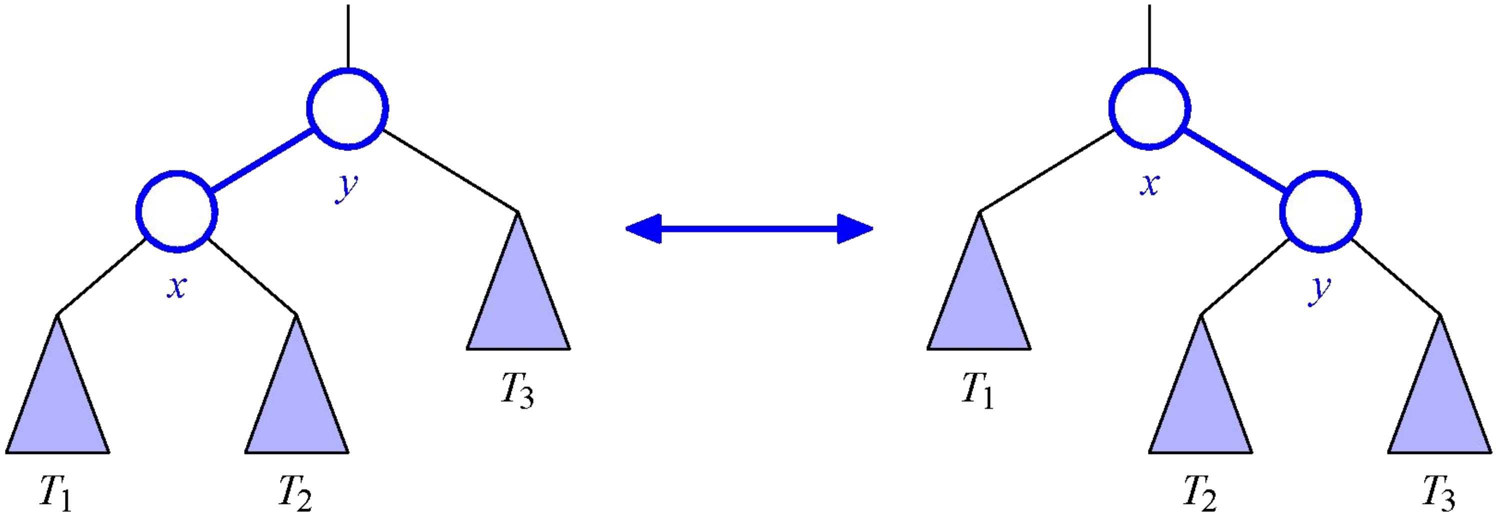
Balanced Binary Trees

- Difference of heights of left and right subtrees at any node is at most 1
- Add an operation to BSTs to maintain balance:
  - **Rotation**

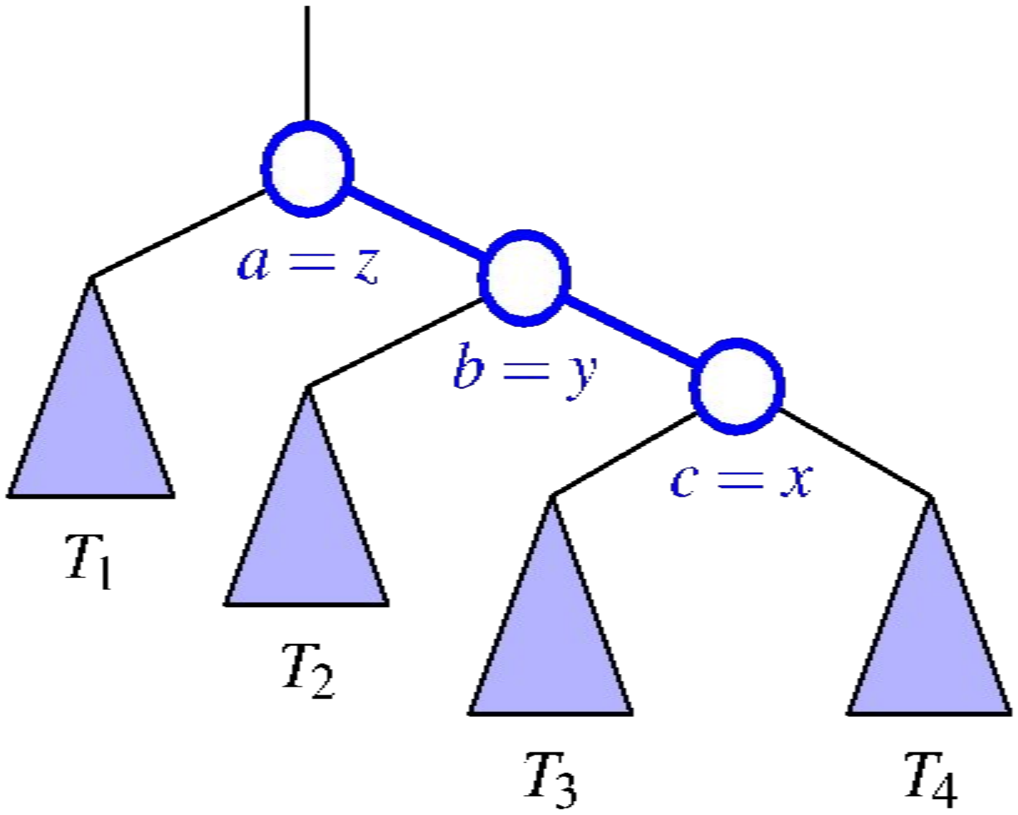
Rotation

Move a child above its parent and relink subtrees

Maintains BST order

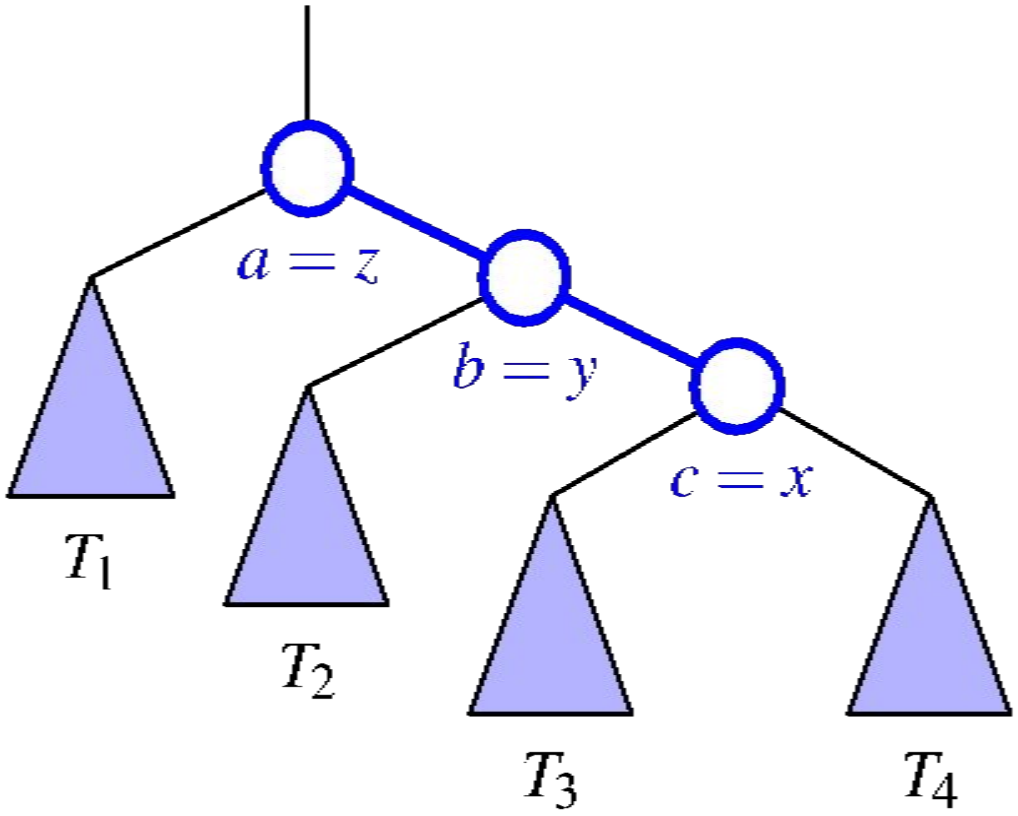


Rotations



- Assume heights of subtrees are equal
  - h(T1) = h(T2) = h(T3) = h(T4)
- What is the height of the entire tree?
  - h(T3) + 2
- What is the height of the left subtree of a?
  - h(T1)
- What is the height of the right subtree of a?
  - h(T4) + 2
- Is this tree balanced?

Rotations

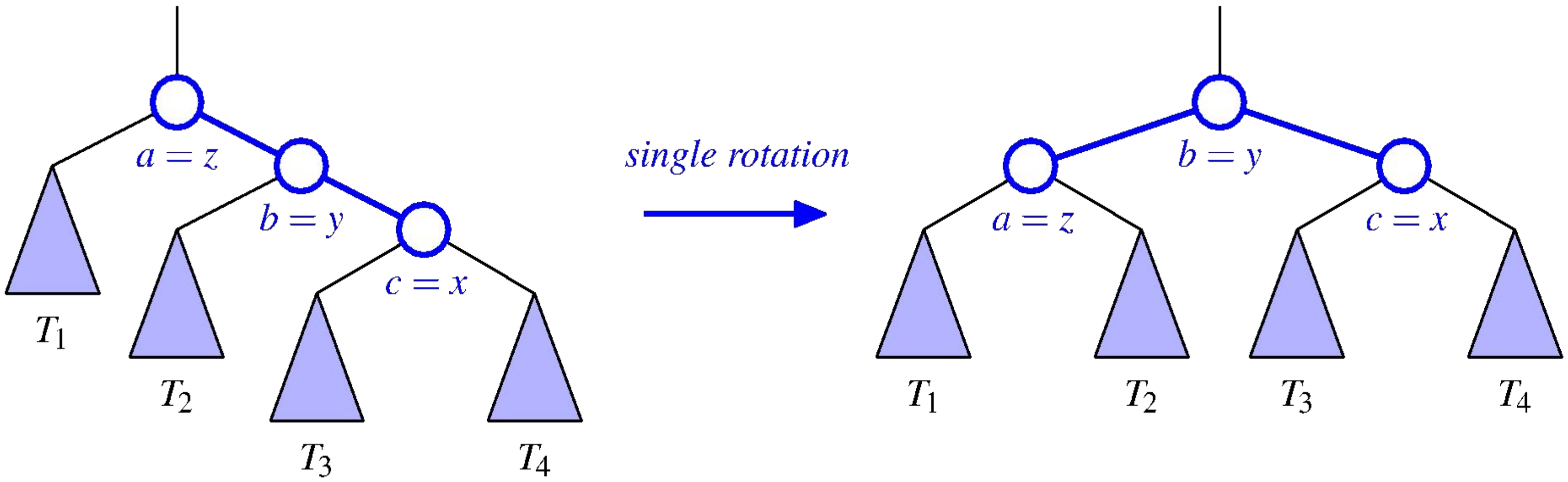


Right subtree is too large!

How can we rotate to fix this?

What should we make the root?

Single Rotation (around z)



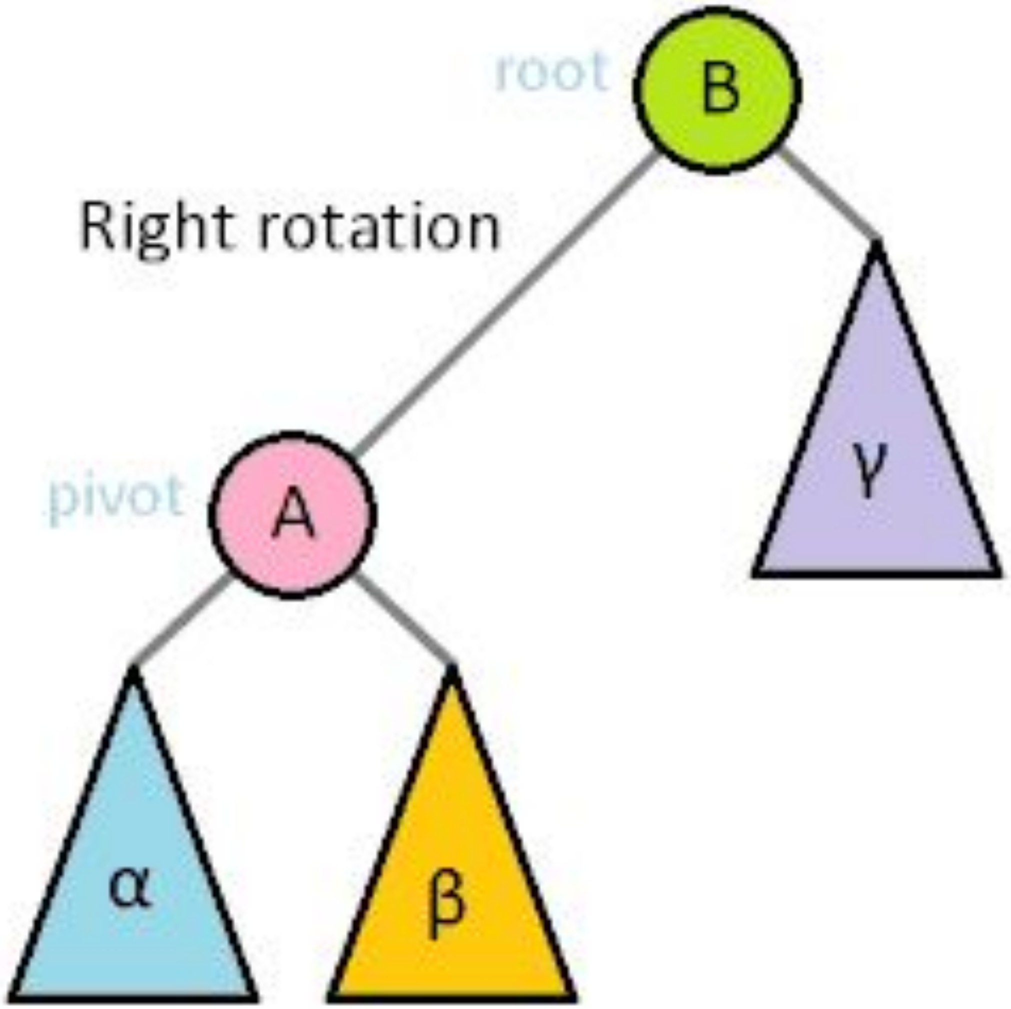
Rotations

Right rotation:

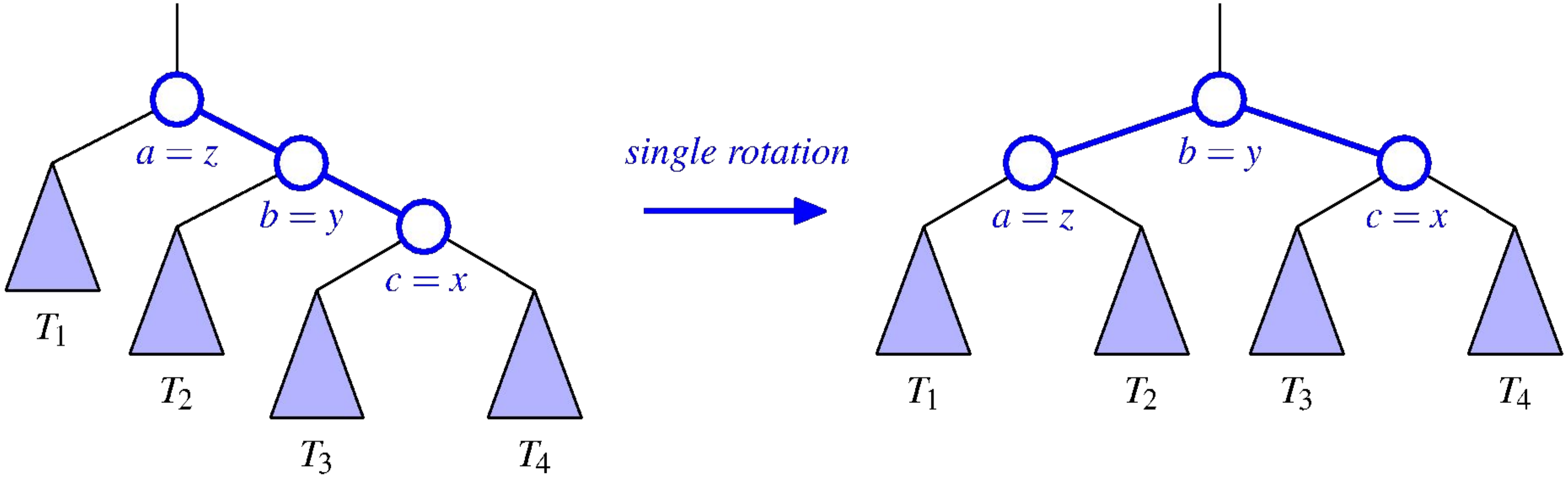
- Performed when left side is heavier
- left child becomes root

Left rotation:

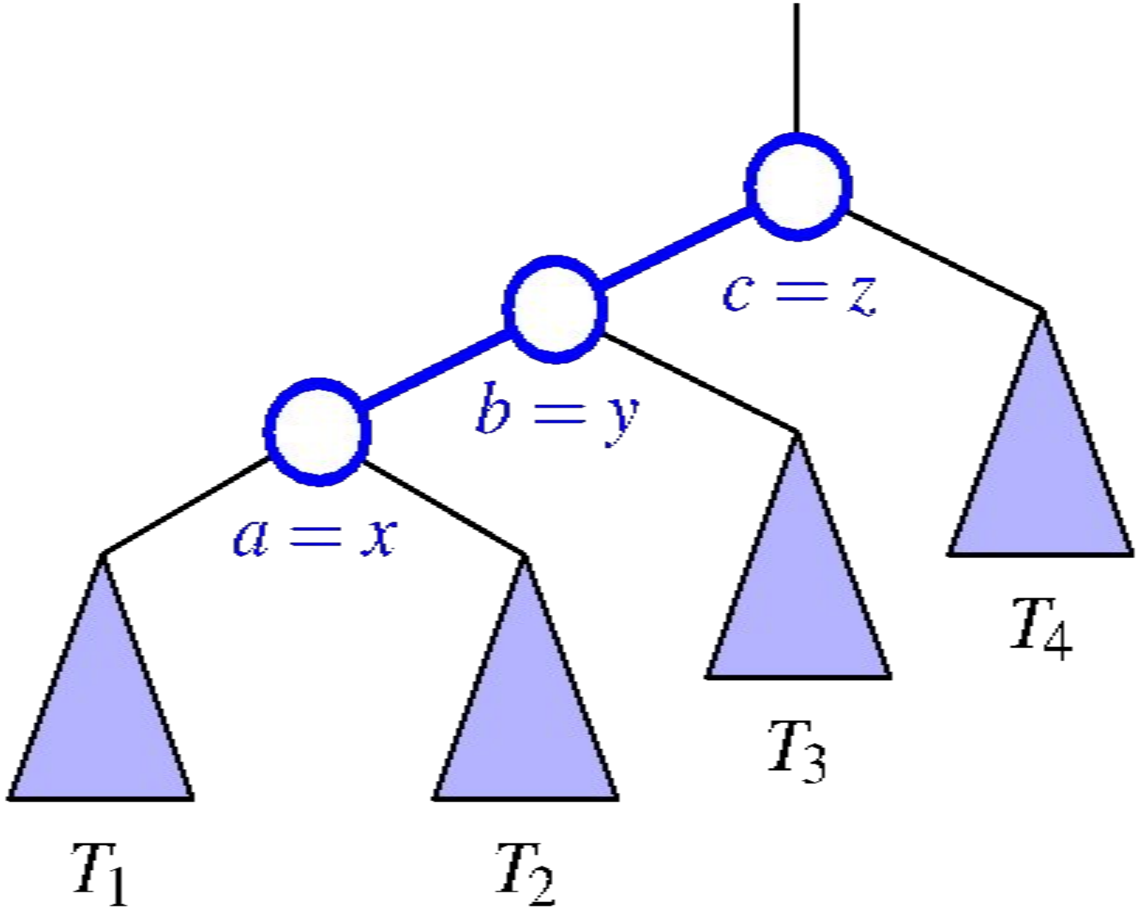
- Performed when right side is heavier
- right child becomes root



Left or Right rotation?



Example 2:

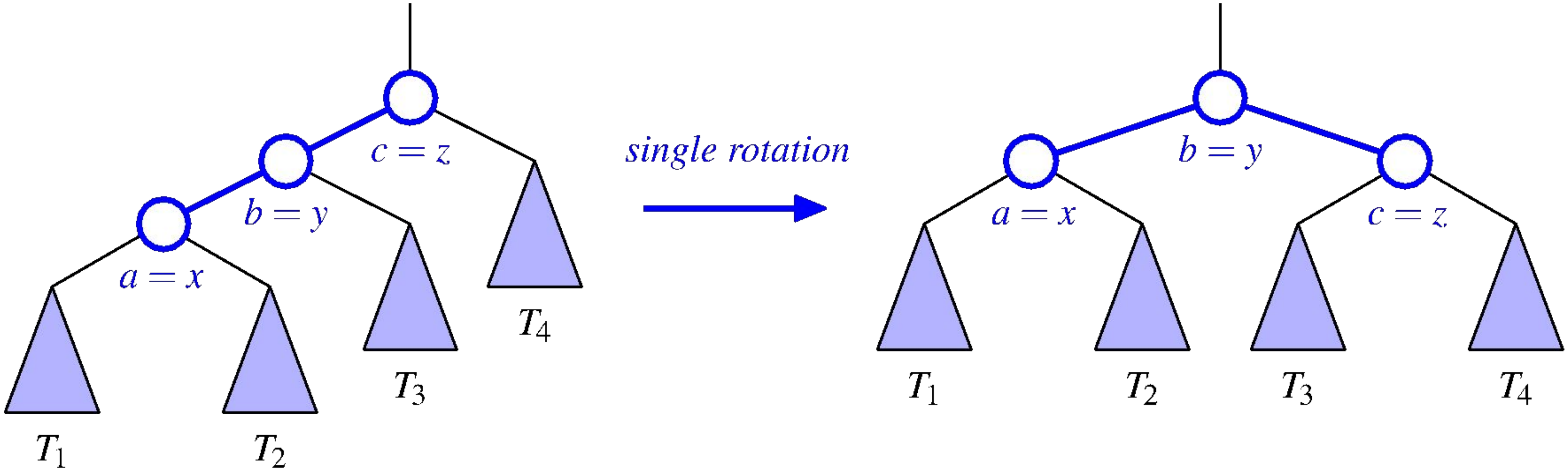


Should we do a left or right rotation?

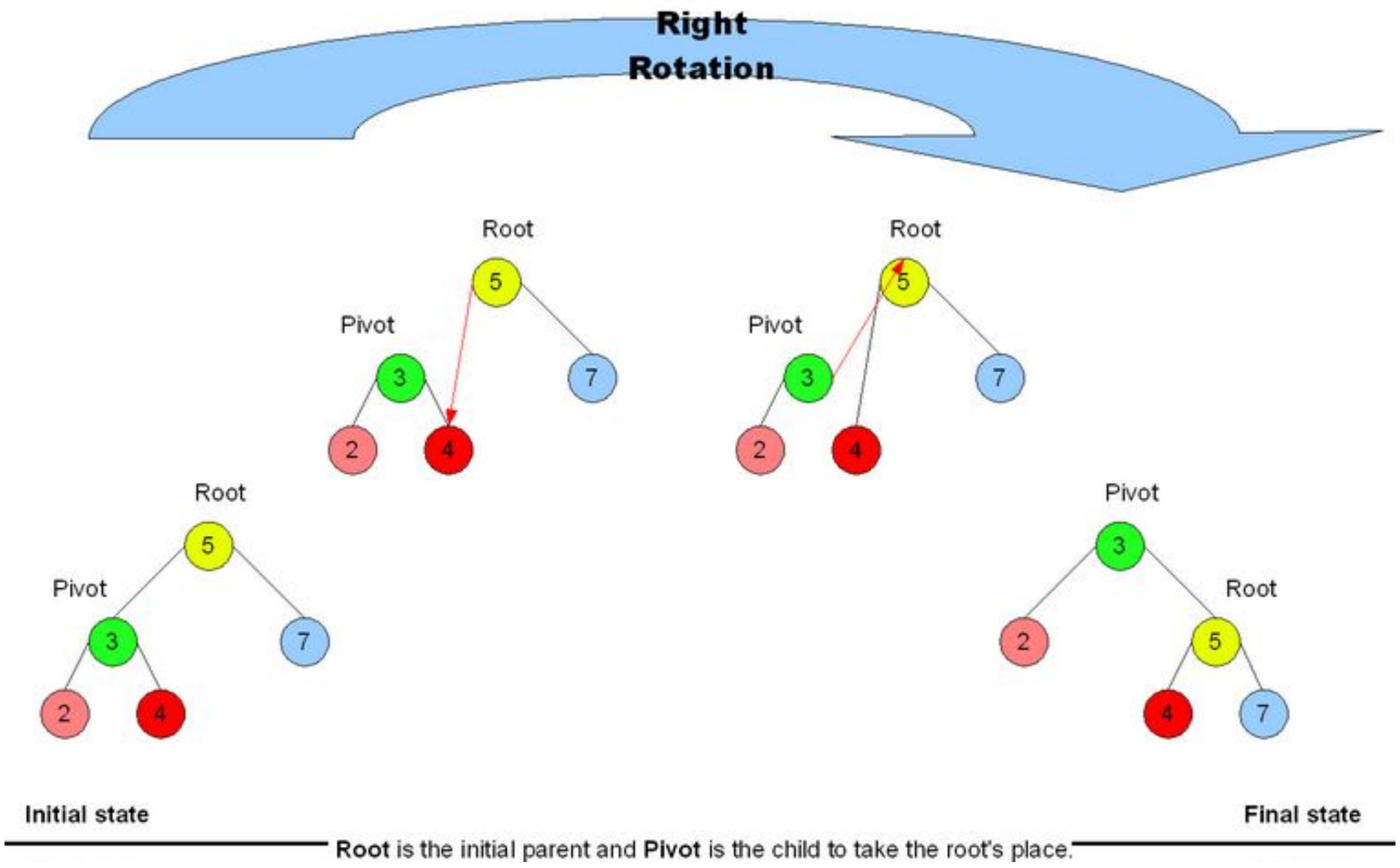
What will become the root?

Let’s draw what it will look like after rotation

Example 2: Rotate Right

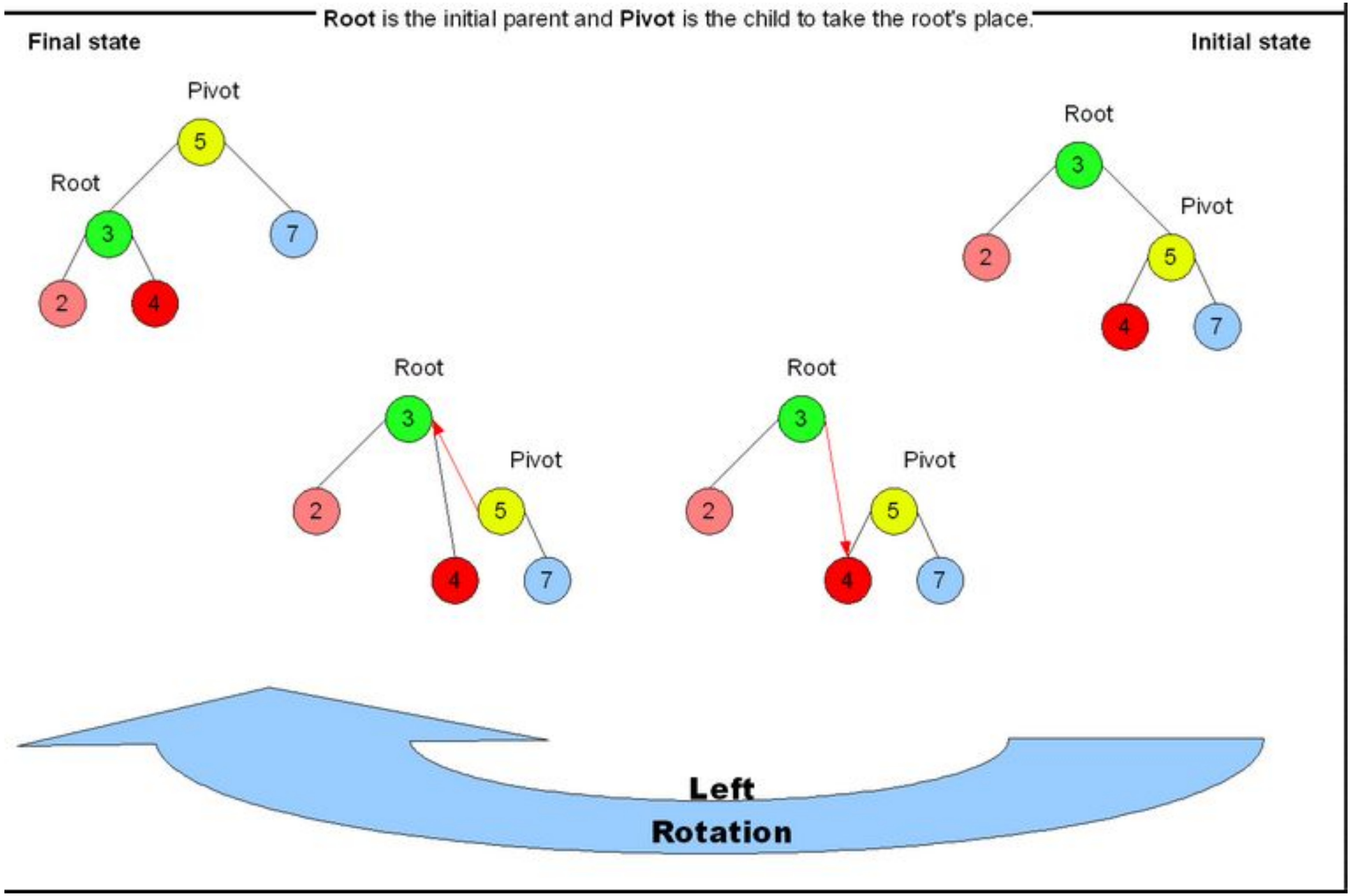


RotateRight Algorithm



1. `Root.left = Pivot.right`
2. `Pivot.right = root`

RotateLeft Algorithm



1. Root.right = Pivot.left

2. Pivot.left = root

Runtime Complexity

Runtime Complexity of rotation?

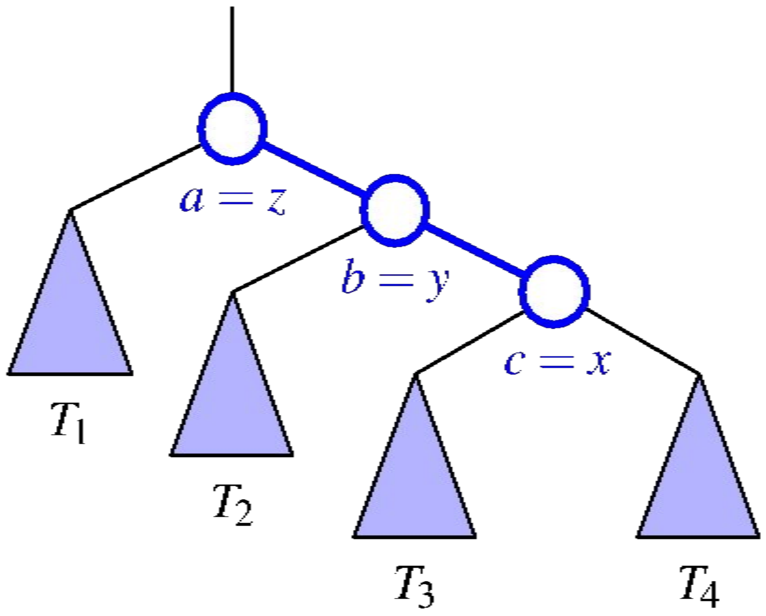
- O(1)

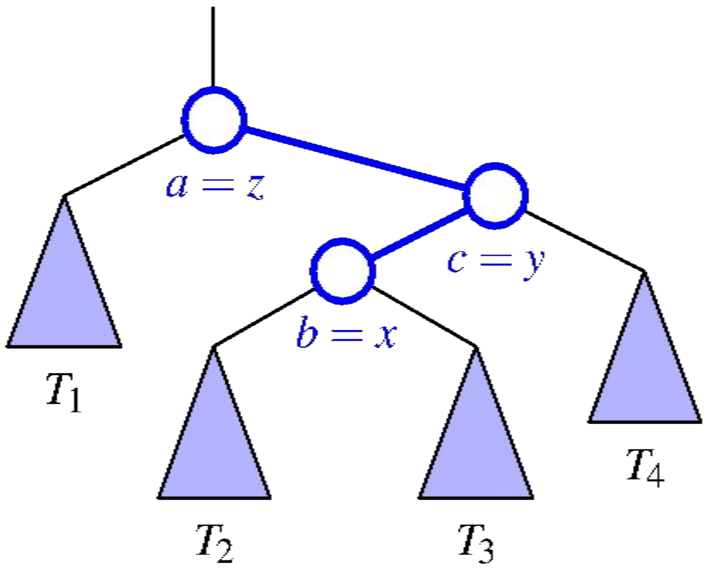
Constant time... we’re just updating links

Double Rotation

Sometimes a single rotation is not enough to restore balance

Double Rotation

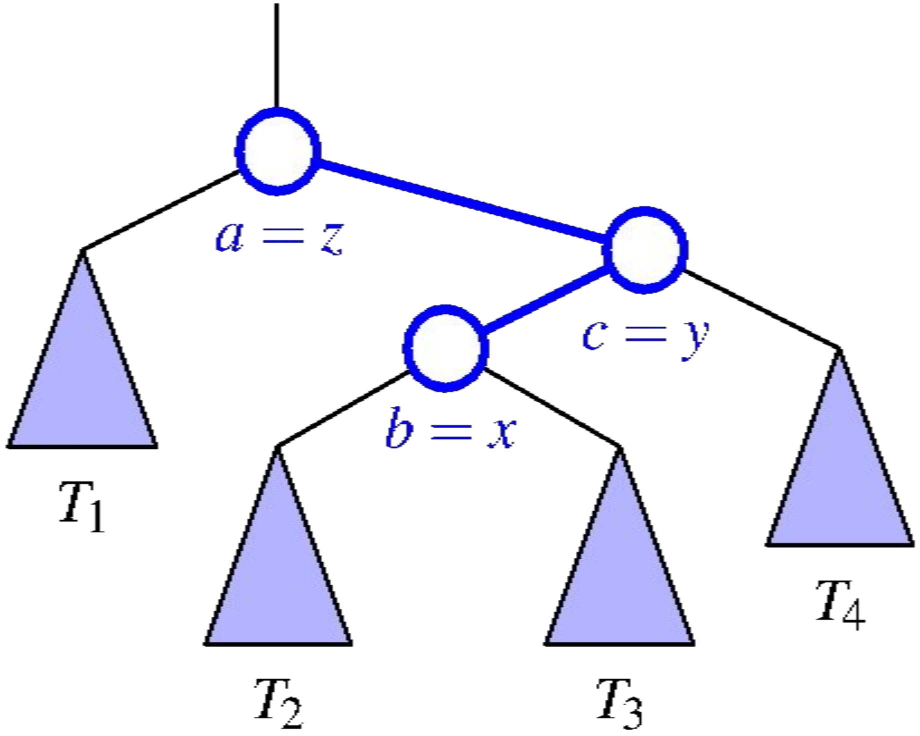




Right child of a is too heavy.. because
Right subtree of b is too heavy..
Single Left rotation on the root needed

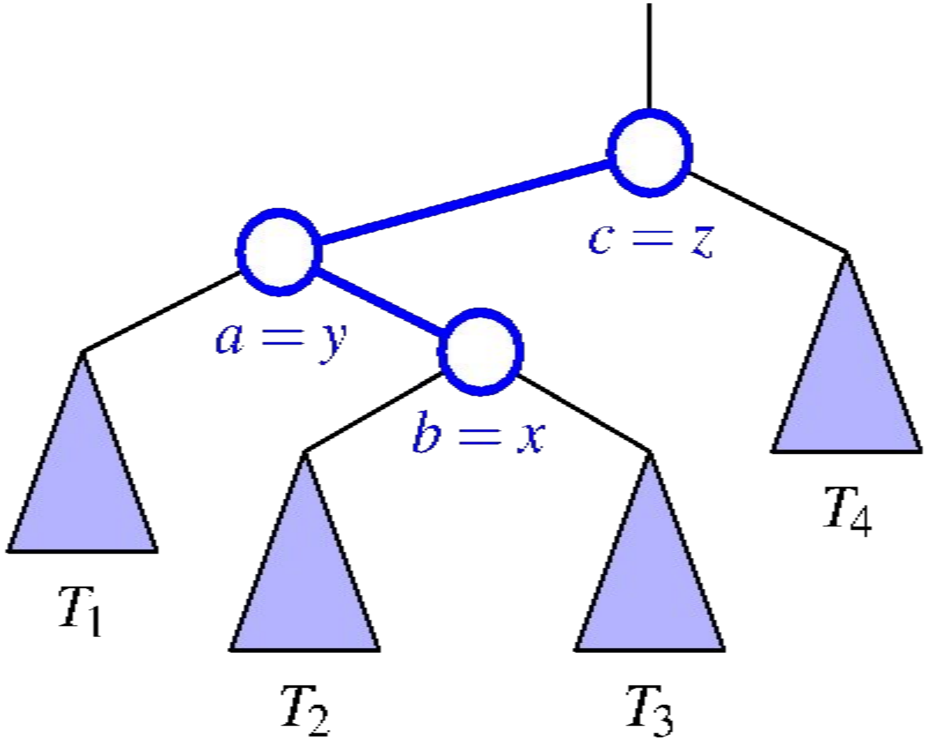
Right child of a is too heavy... because
Left subtree of c is too heavy
Is a single rotation enough?

Double Rotation

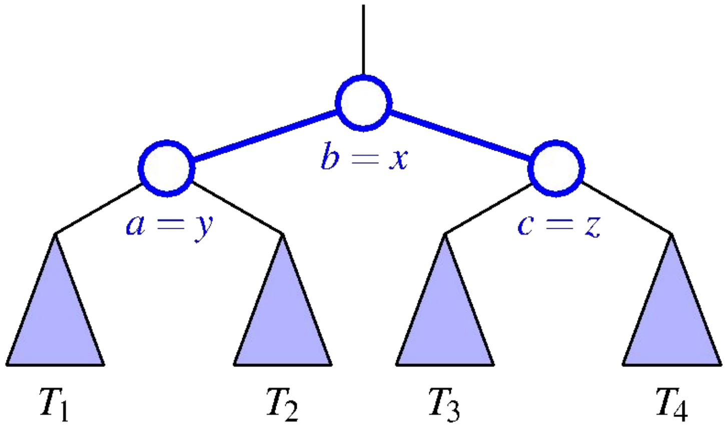


1. **Rotate Right** at c because right subtree of root is too heavy
2. **Rotate Left** at the root (a)

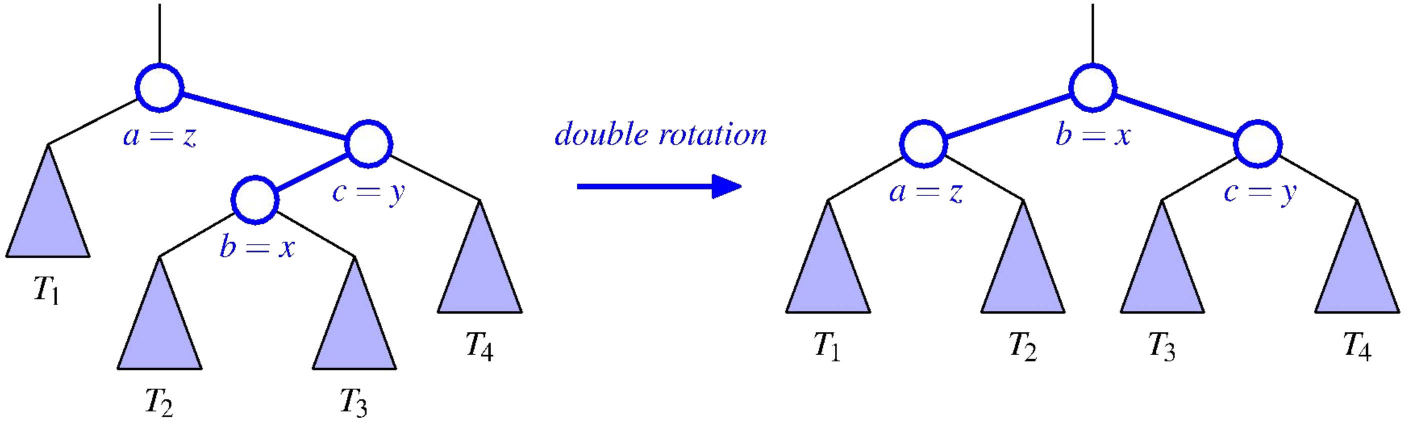
Double Rotation Example 2:



1. **Rotate Left** at `a` because right subtree of root is too heavy
2. **Rotate right** at the root (c)

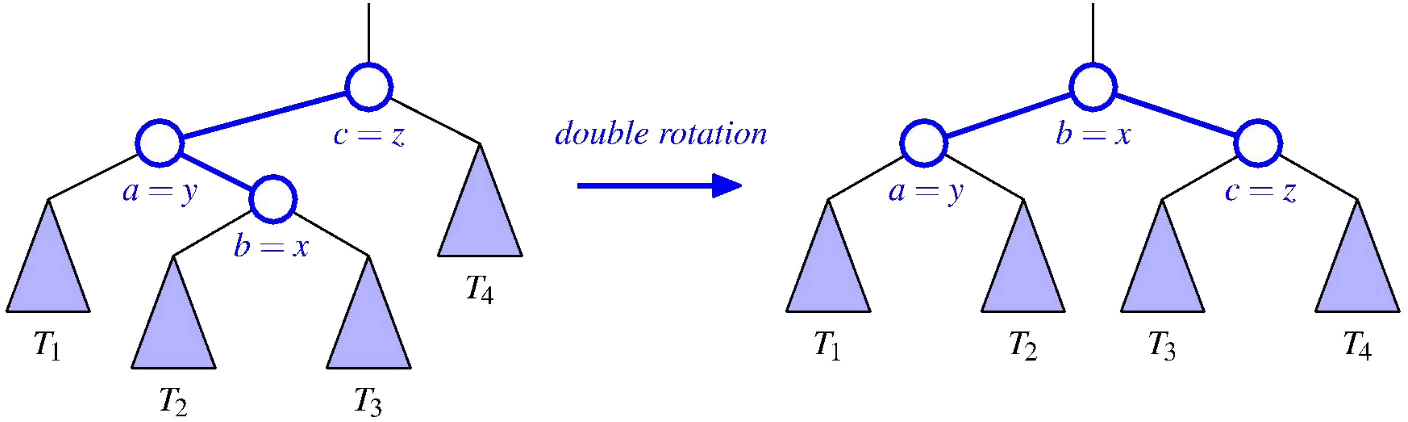


Double Rotations



Right subtree is too heavy because of **left** subtree of c

1. Rotate Right about c
2. Rotate Left about a

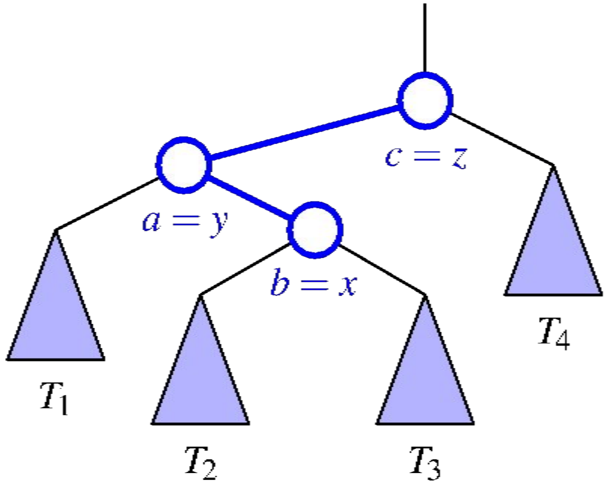


Left subtree is too heavy because of **right** subtree of a

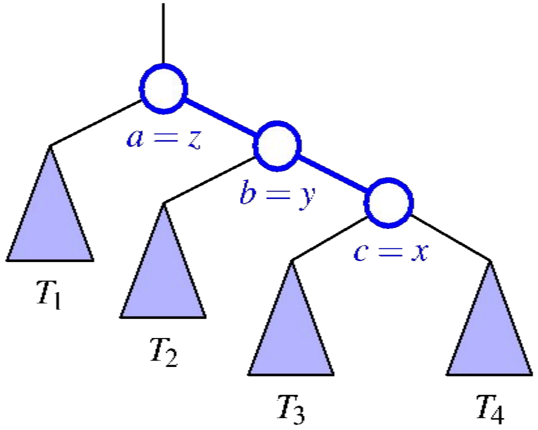
1. Rotate Left about a
2. Rotate Right about c

Double Rotation

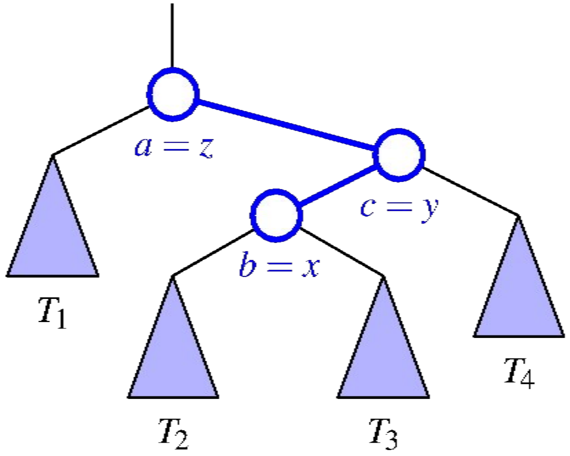
When do we need a double rotation vs a single rotation?



Double rotation



Single rotation



Double rotation

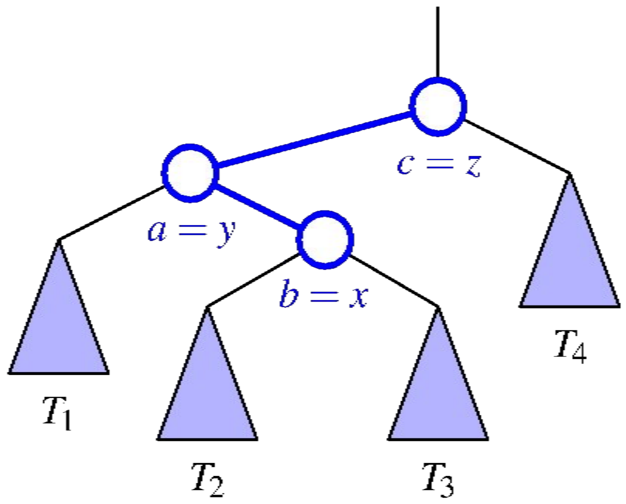
Look for zig-zag pattern!

Double rotation

When do we need a double rotation?

Left subtree is too heavy on the right side

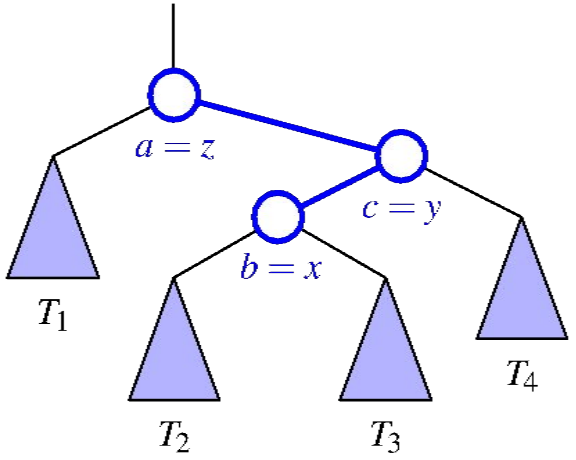
`rotateLeftRight`



OR

Right subtree is too heavy on the left side

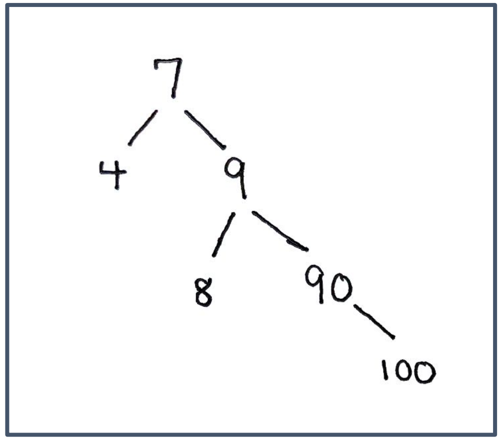
`rotateRightLeft`

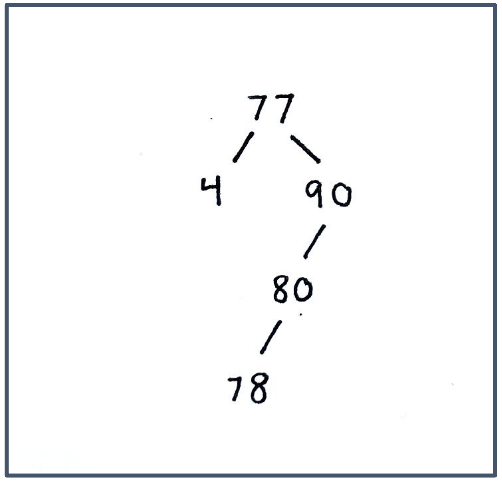


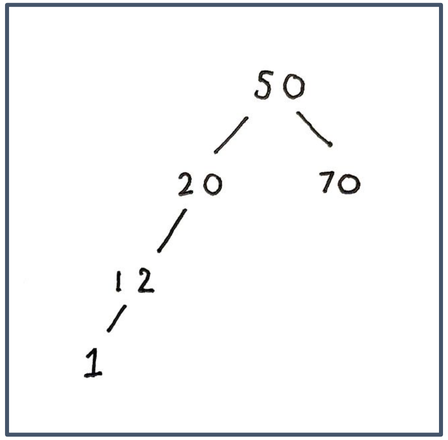
Double Rotation Code

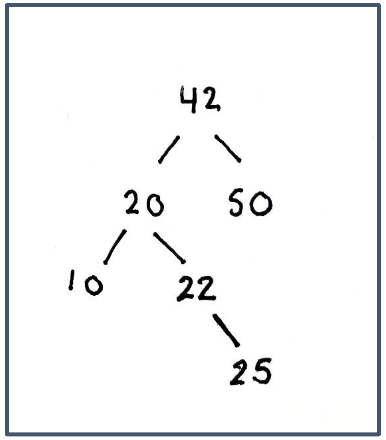
```
def rotateLeftRight(n)
    n.left = rotateLeft(n.left);
    n = rotateRight(n);

def rotateRightLeft(n)
    n.right = rotateRight(n.right);
    n = rotateLeft(n);
```

Examples - which way should I rotate?









rotateLeft

rotateRightLeft

rotateRight

rotateLeftRight

Summary: Tree rotation

- Can rotate to left or right
- Used to restore balance in height
- Rotation maintains BST order
- Runtime complexity of rotation?
  - O(1)

AVL Trees

AVL Trees

- "*self balancing* binary search tree"
- For every internal node, **the heights of the two children differ by at most 1**
- does rotations upon insert/removal if necessary

AVL Height

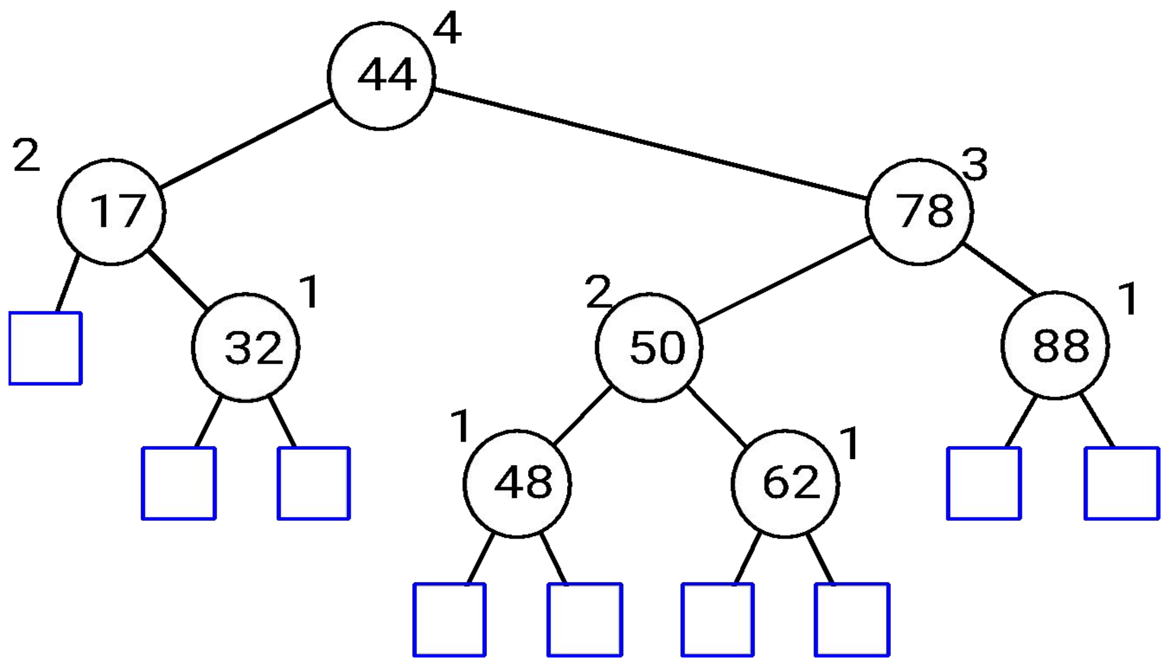
- We keep track of the height of each node as a field for quick access

- The height of an AVL tree is logn
  - Always balanced

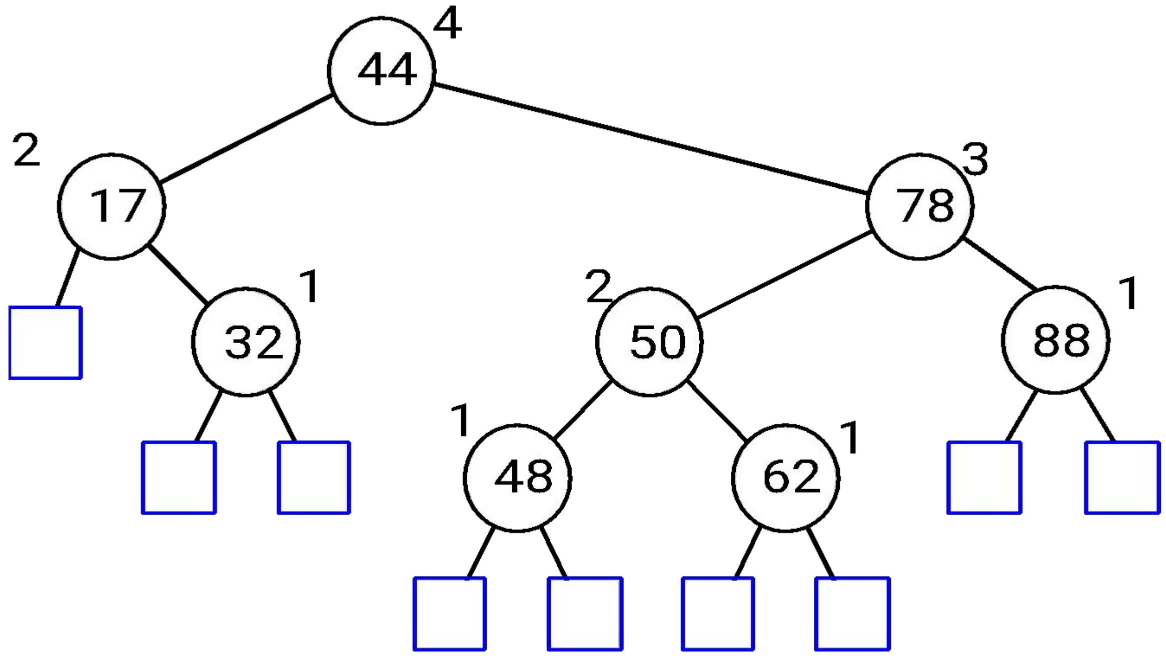
Insertion

AVL Tree Example

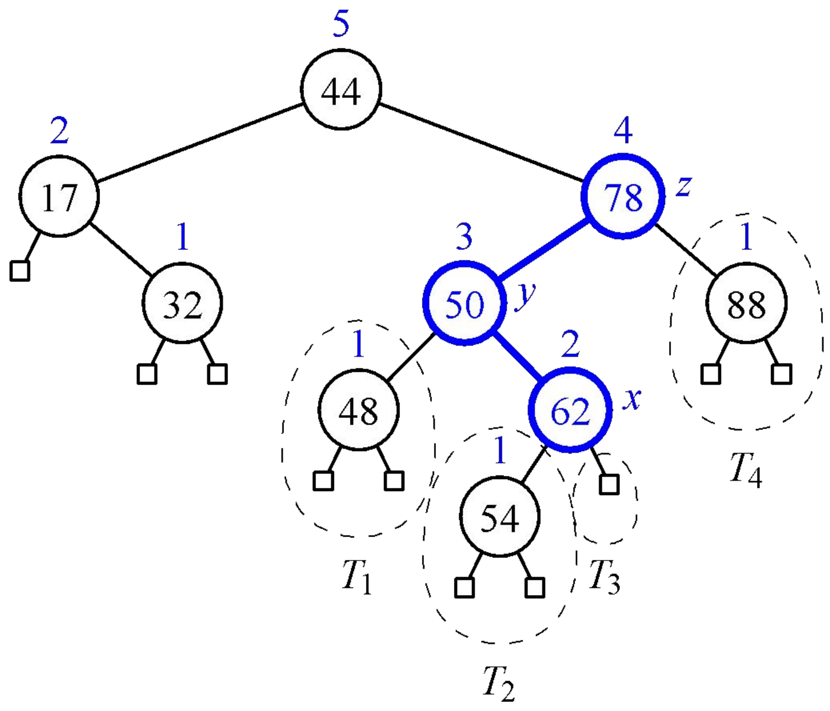
- leaves are sentinels and have height 0

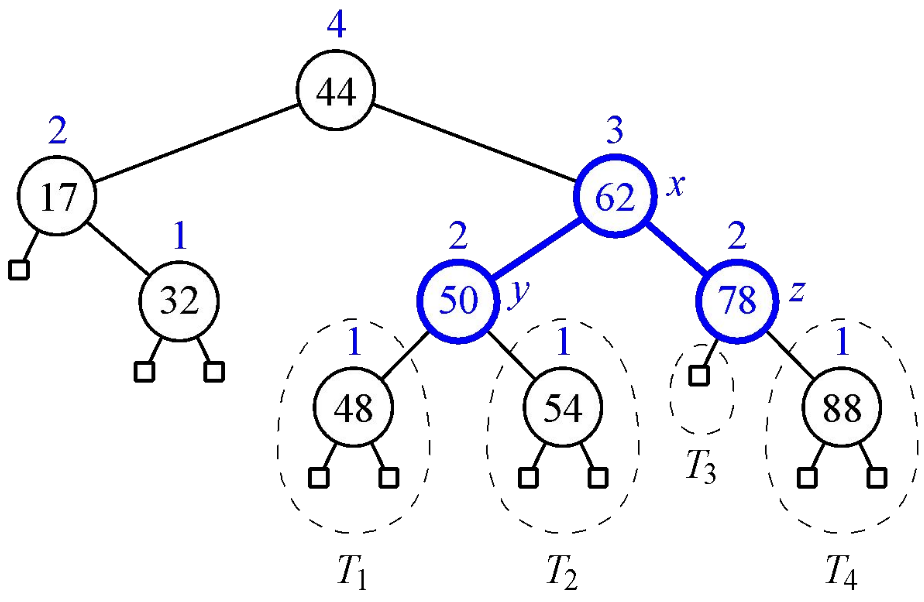


Insert 54



Insertion (54)

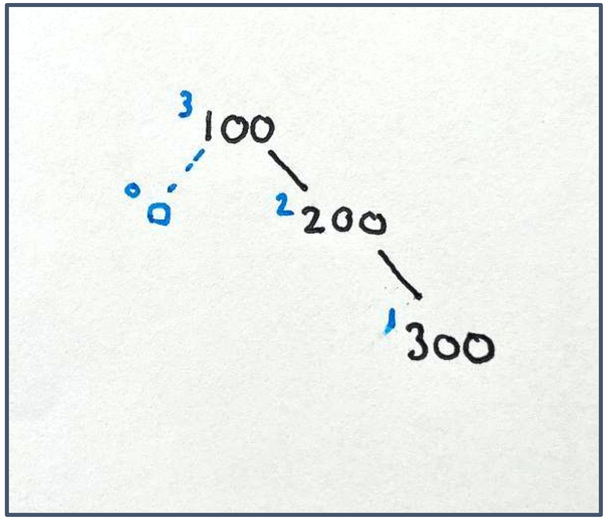


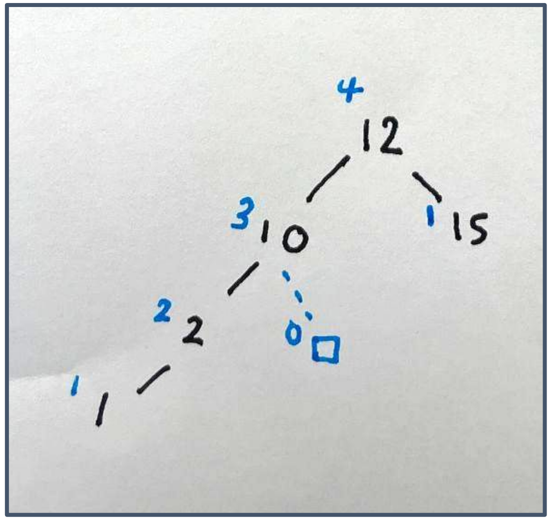


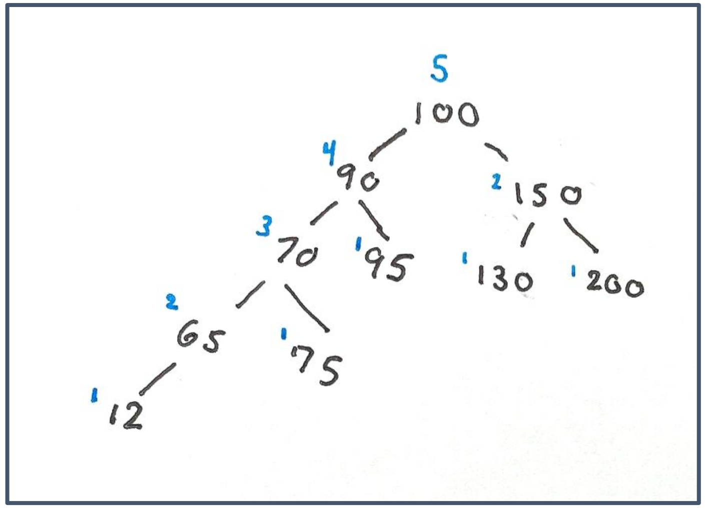
New node always has height 1

Parent may change height

Which node do we “rebalance over”?







lowest subtree with diff(heights) > 1

Exercise

- Create an AVL tree by inserting the nodes in this order:
  - M, N, O, L, K, Q, P, H, I, A

AVL Animation

Rebalance Algorithm

```
If left.height > right.height + 1:
    if (left.right.height > left.left.height)  //double rotate
        rotateLeftRight(n)
    else:
        rotateRight(n)

else if right.height > left.height + 1:
    if (right.left.height > right.right.height) //double rotate
        rotateRightLeft(n)
    else:
        rotateLeft(n)
```

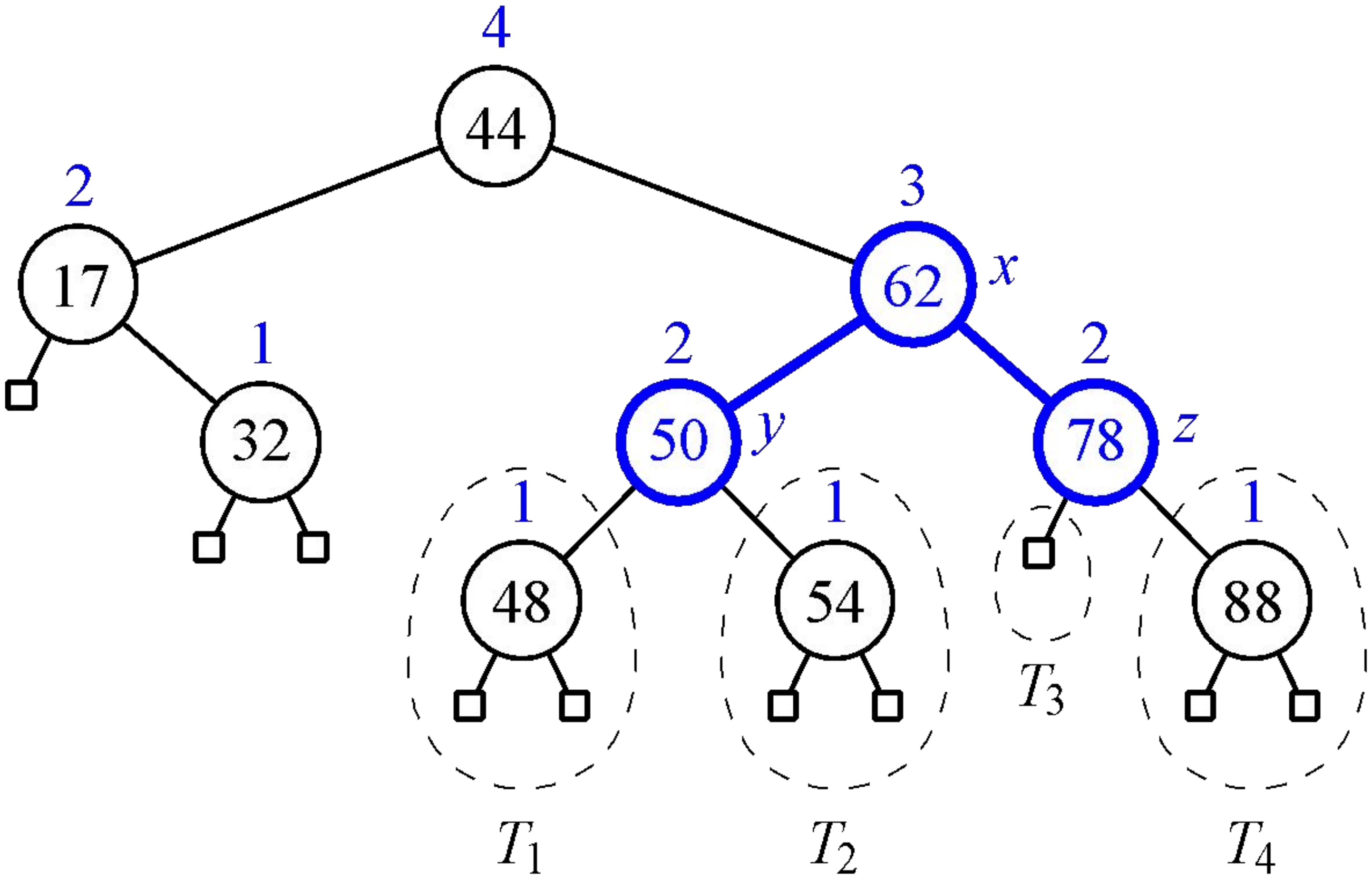
Runtime Complexity:

Insertion (plus rotation)

a. search + find node to rebalance + rotate
b. O(logn) + O(logn) + O(1) = **O(logn)**

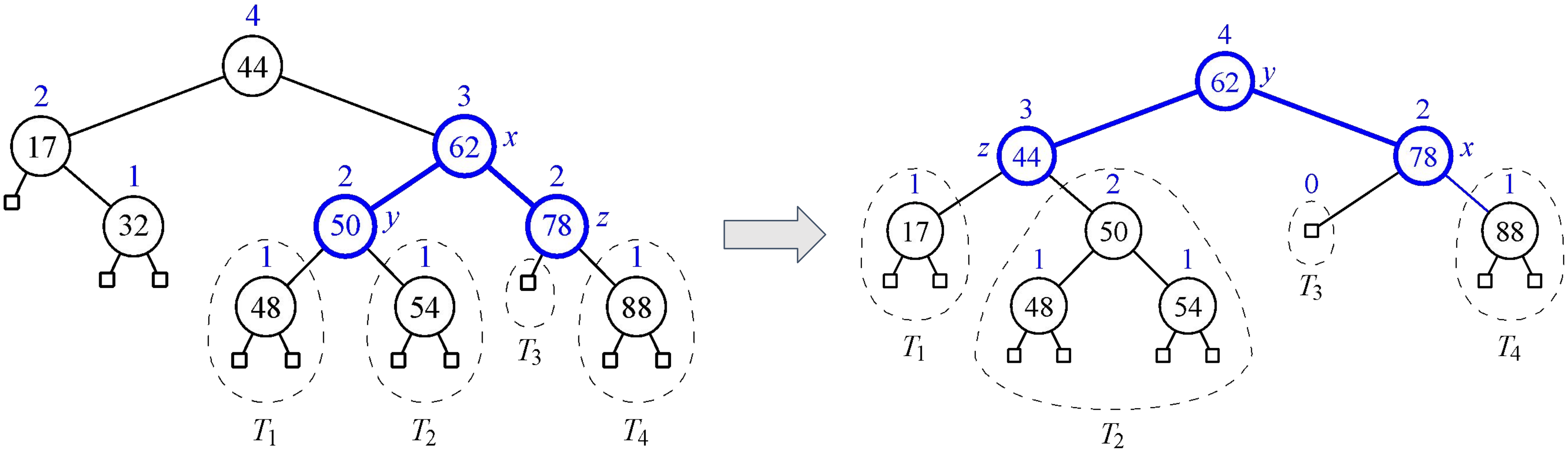
Deletion

Delete Example 1: 32

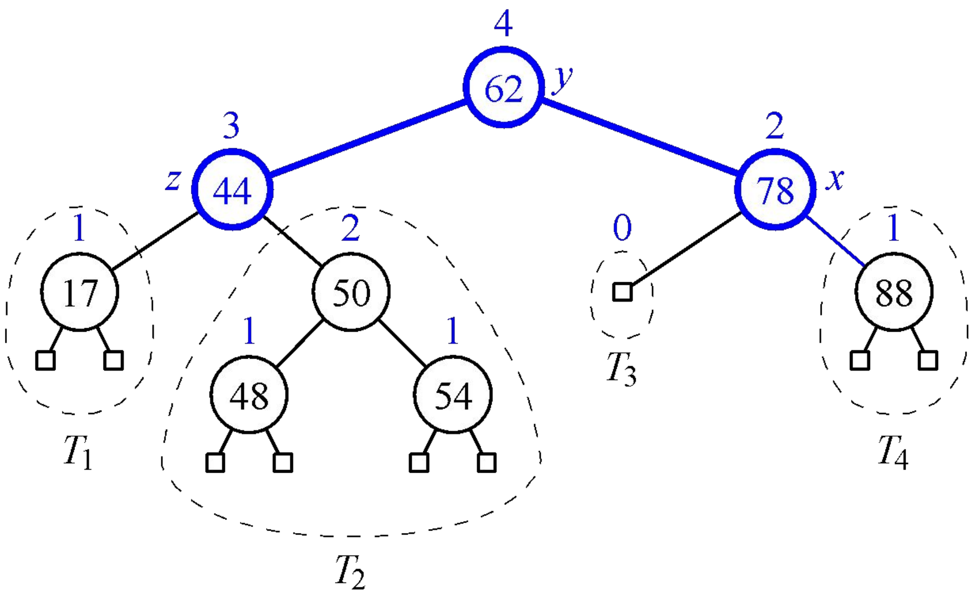


Delete Example 1: 32

rotateLeft

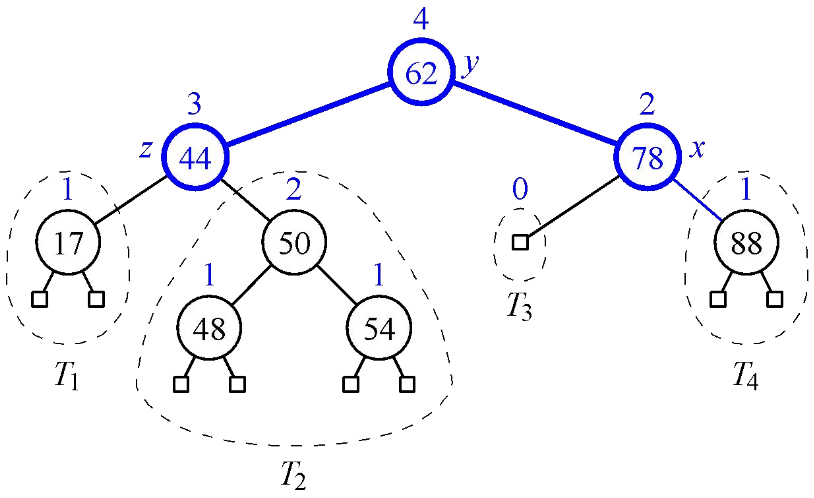


Delete Example 2: 88

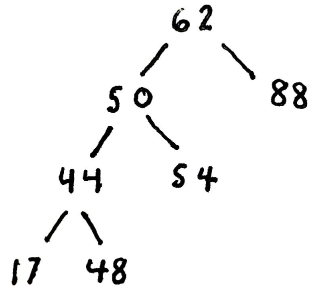


Delete Example 2: 88

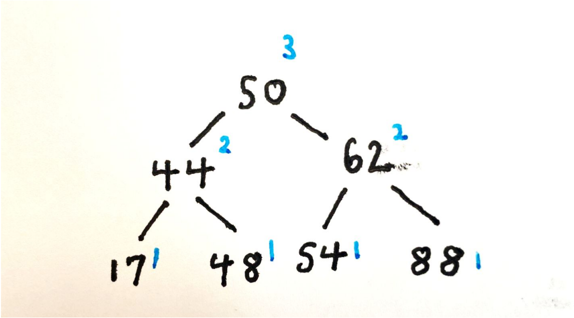
rotateLeftRight



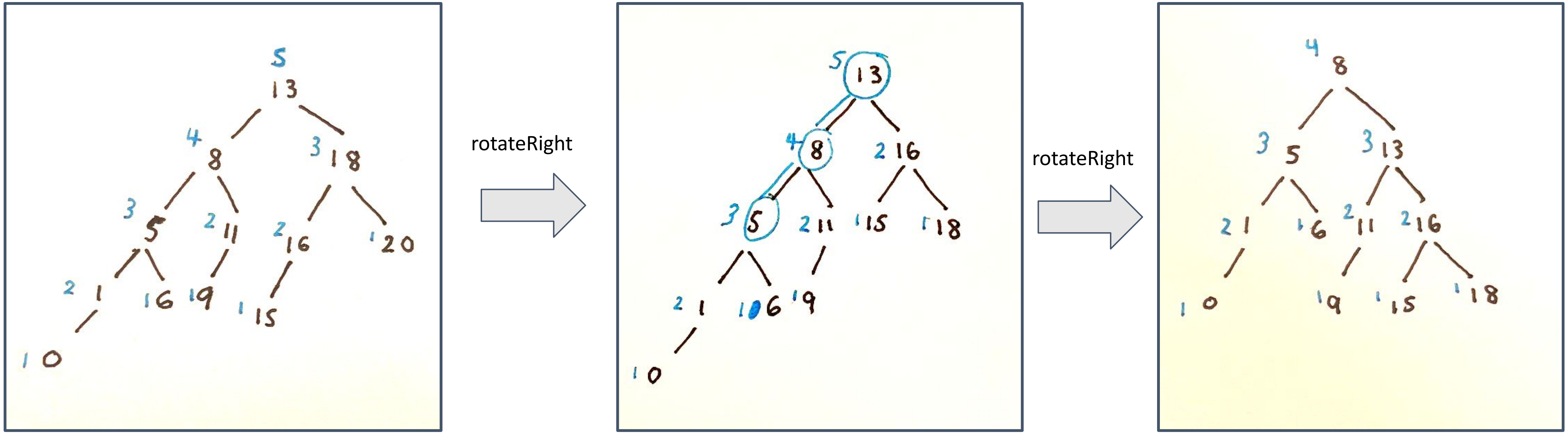
rotateLeft



rotateRight



Delete Example 3: 20



Delete Example 3: 20

- Deletion can cause more than one rotation

- Worst case requires O(logn) rotations
  - deleting from a deepest leaf node and rotating each subtree up to the root

Removal

Runtime Complexity?

a. search + find node to rebalance + rotate
b. O(logn) + O(logn) + O(1) = **O(logn)**

Still O(logn) even though we may need multiple rotations?

Why?

-> Even though we may need to find multiple nodes to rebalance we only traverse the height of the tree once

Performance of BSTs

Runtime complexity:

search?
- BST:
  - O(n)
- AVL:
  - O(logn)

Performance of BSTs

Runtime complexity:

insert?
- BST:
  - O(n)
- AVL:
  - O(logn)

Performance of BSTs

Runtime complexity:

remove?
- BST:
  - O(n)
- AVL:
  - O(logn)