CS151 Intro to Data Structures

Sorted Maps, Tree Maps,
Balanced Search Trees, AVL Trees

Announcements

HW06 due Wednesday 11/29

  Lab08 due Wednesday too

No lab this week

Last lab will be next week

HW07 due Wednesday 12/06

  Hashmaps & Sorting

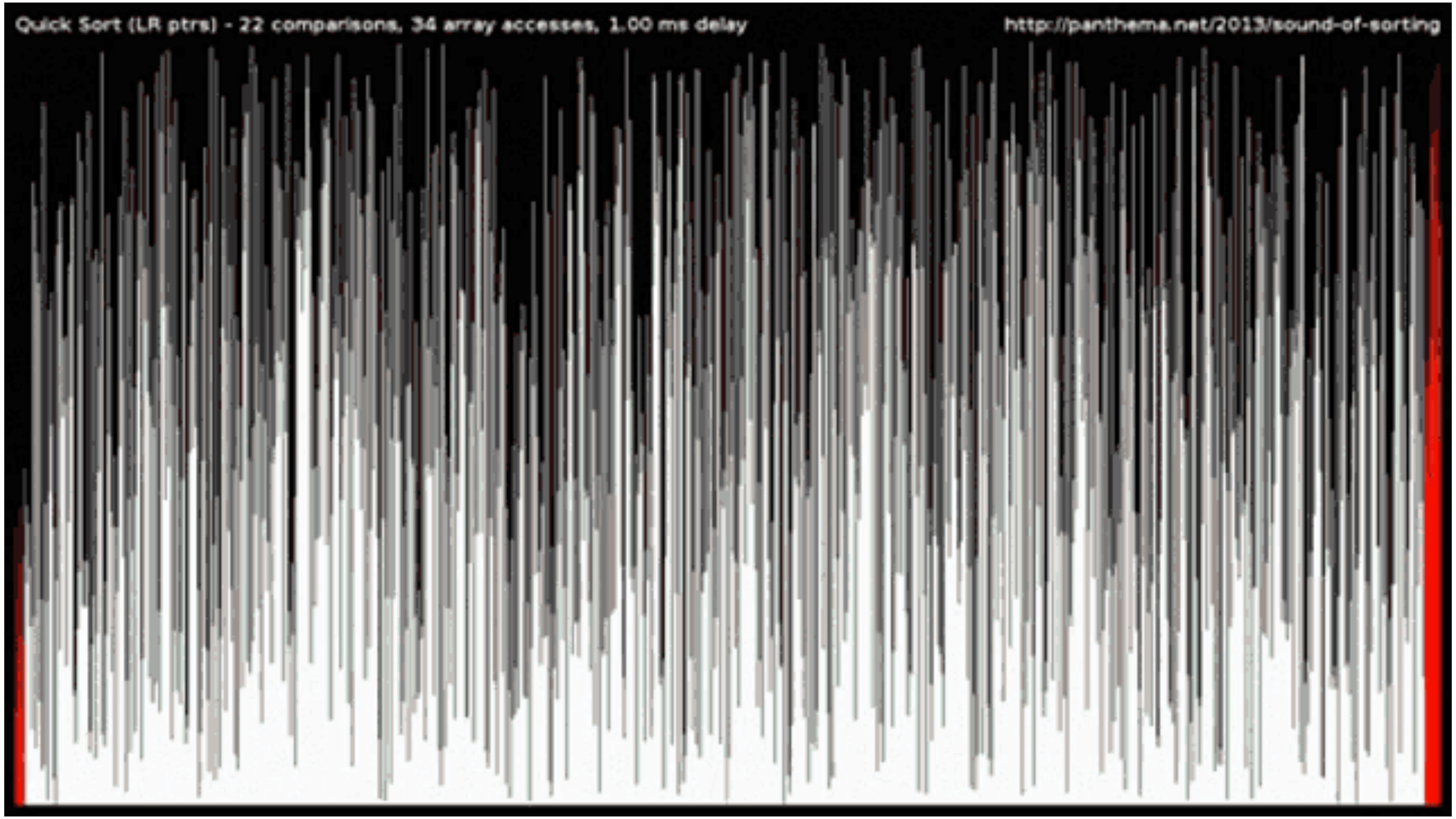
HW08 due 12/14

Outline

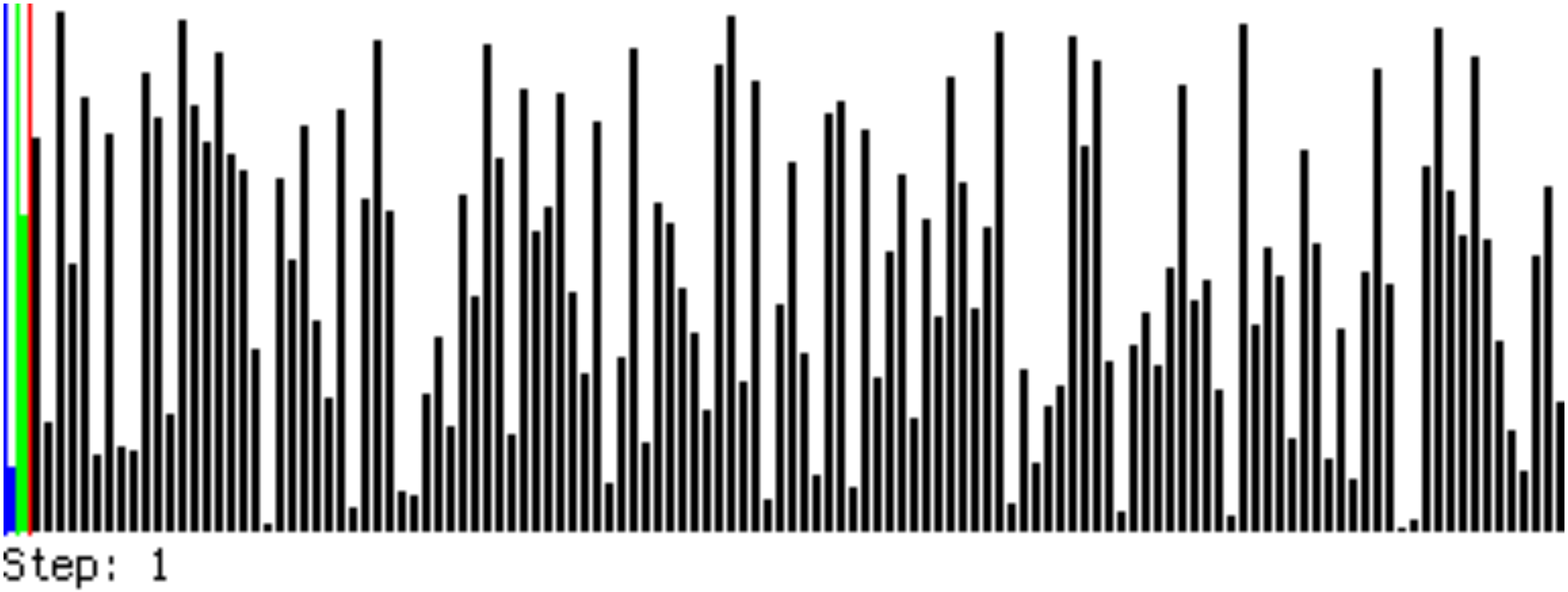
Sorting Review

Sorted Maps

Sorting


Quick Sort (LR ptrs) - 22 comparisons, 34 array accesses, 1.00 ms delay
http://panthema.net/2013/sound-of-sorting

Sorting



In-place Partitioning (Hoare's)

- Use two indices to split into L and $E \cup G$

`l` `r`

6 2 5 1 0 7 3 5 9 2 7 9 8 9 7 9 3

- Repeat until `l` and `r` cross:
  - Move `l` to the right to find $\geq x$
  - Move `r` to the left to find $< x$
  - swap elements at `l` and `r`

pivot = 6
use first
element as
pivot or
swap pivot
with last

`l` `r`

6 2 5 1 0 $\boxed{7}$ 3 5 9 2 7 9 8 9 7 9 $\boxed{3}$

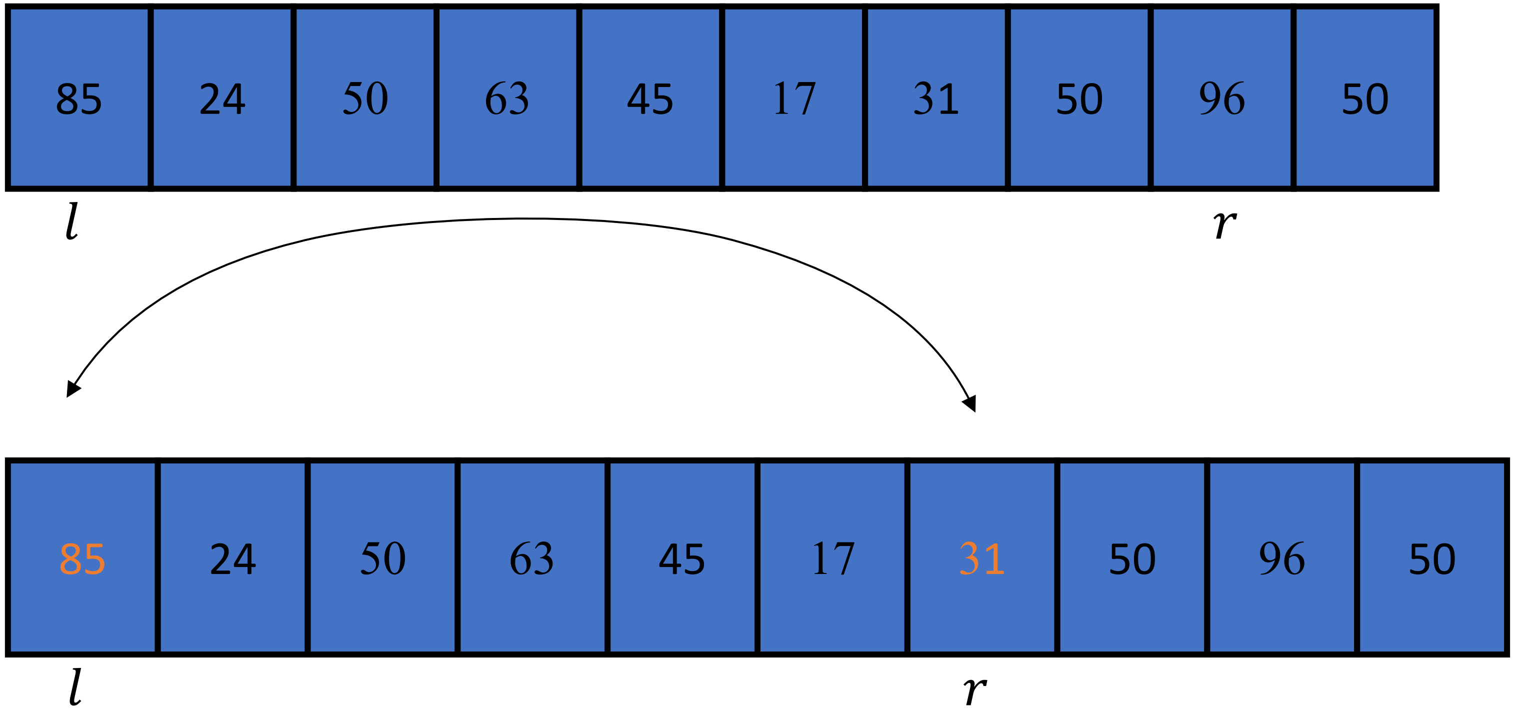
Example with Duplicates

- Use last element in array as pivot

85	24	50	63	45	17	31	50	96	50
l								r	

Example with Duplicates

- Use last element in array as pivot



Example with Duplicates

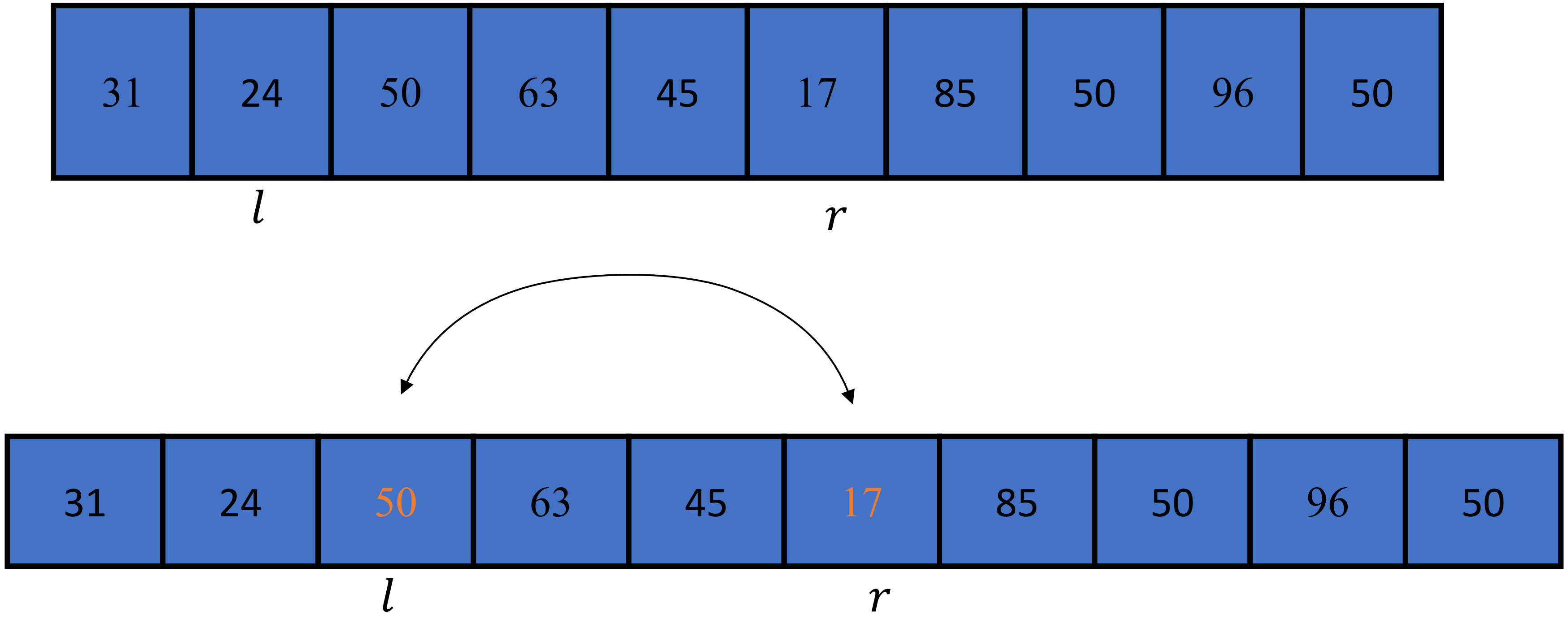
- Use last element in array as pivot

85	24	50	63	45	17	31	50	96	50
l						r			

31	24	50	63	45	17	85	50	96	50
	l				r				

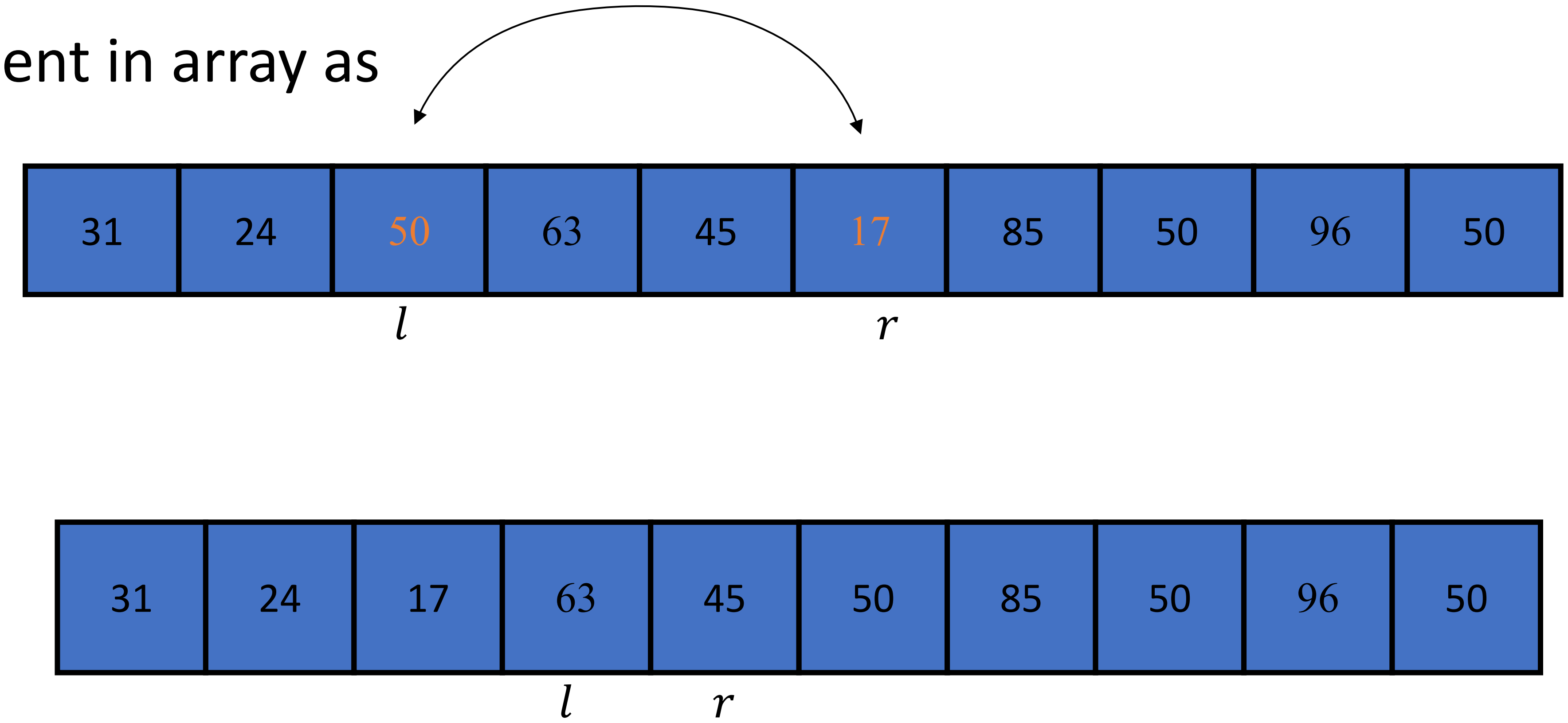
Example with Duplicates

- Use last element in array as pivot



Example with Duplicates

- Use last element in array as pivot



Example with Duplicates

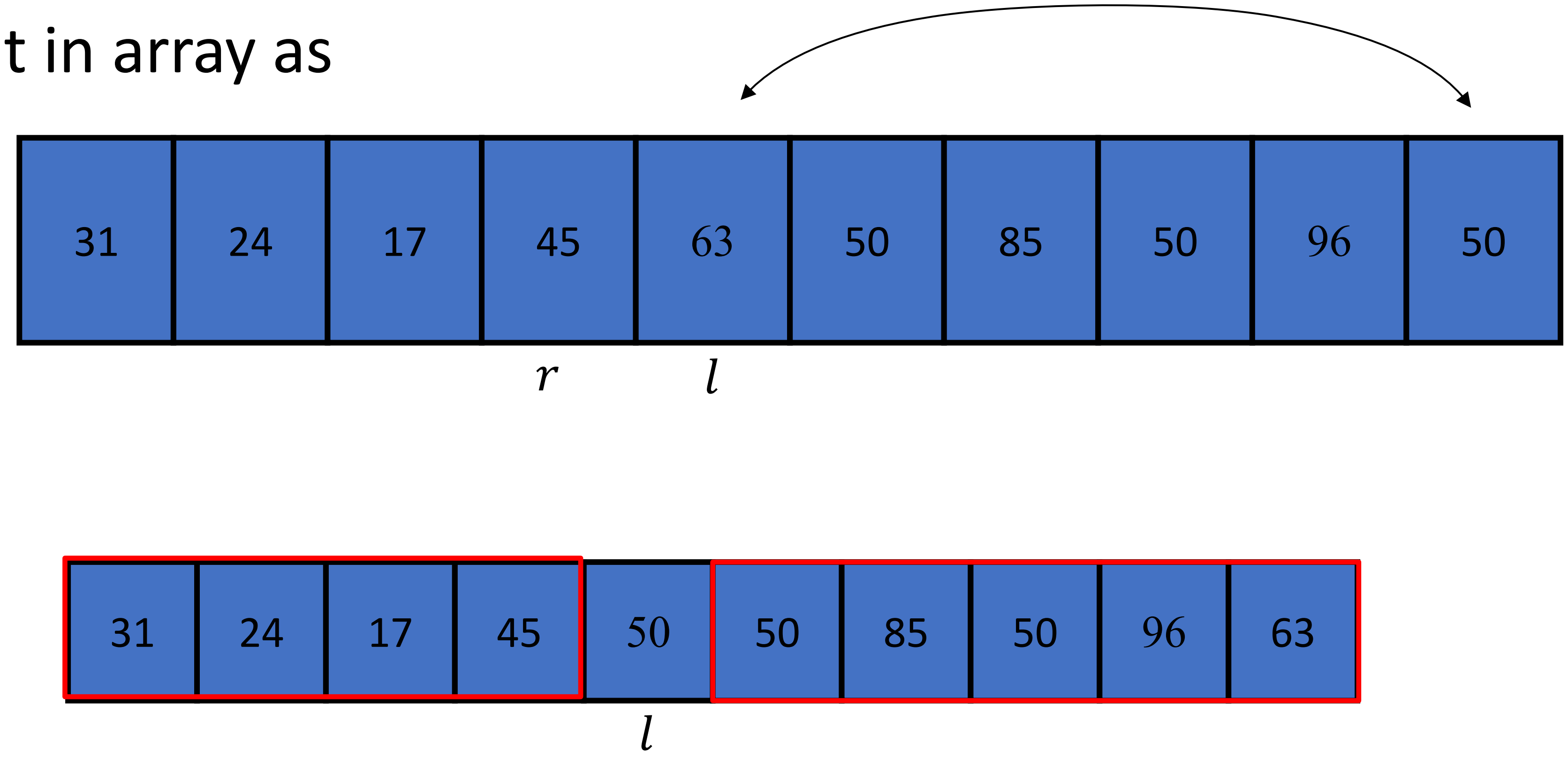
- Use last element in array as pivot

31	24	17	63	45	50	85	50	96	50
			l	r					

31	24	17	45	63	50	85	50	96	50
			r	l					

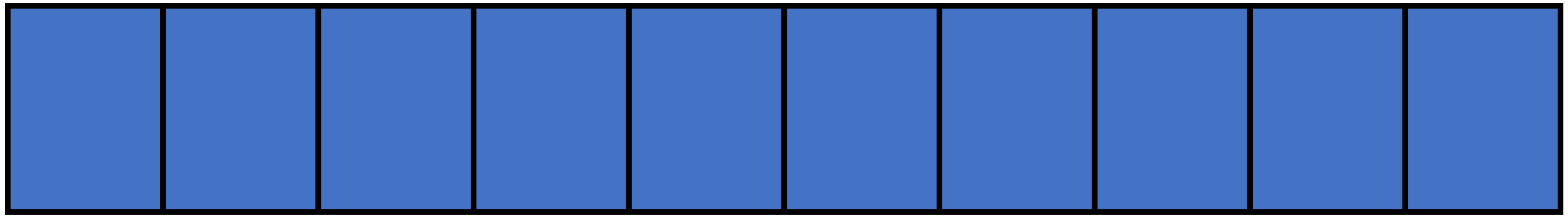
Example with Duplicates

- Use last element in array as pivot



HashMap

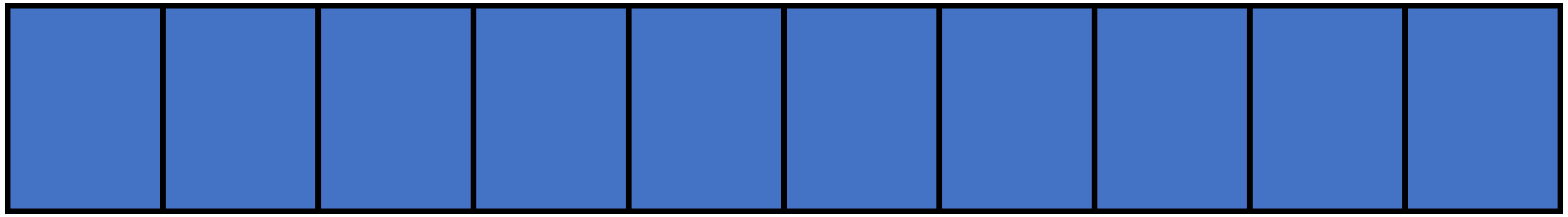
Keys are uniformly distributed in a hashtable



HashMap

Keys are uniformly distributed in a hashtable

Key: Timestamp; Value: Action

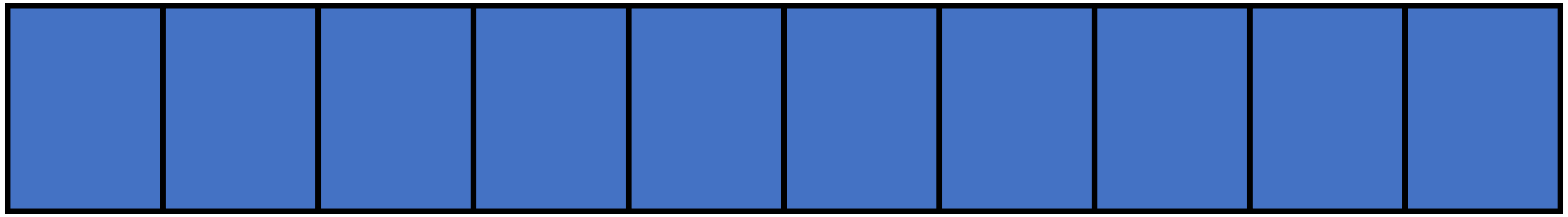


HashMap

Keys are uniformly distributed in a hashtable

Key: Timestamp; Value: Action

A: (10:14, Deposit)



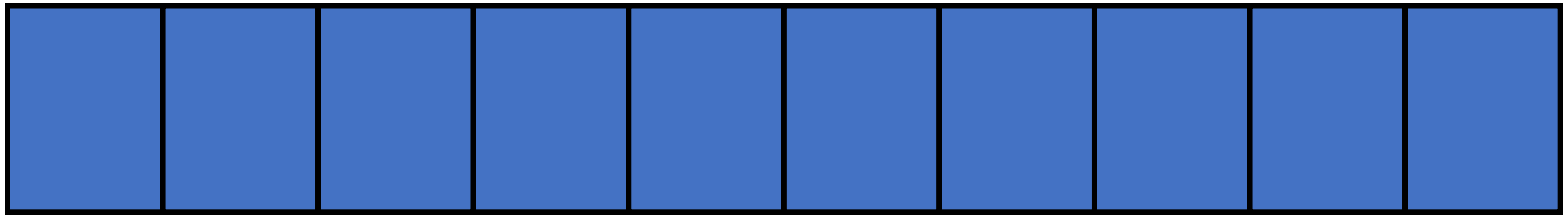
HashMap

Keys are uniformly distributed in a hashtable

Key: Timestamp; Value: Action

A = (10:14, Deposit)

$$P(h(A) = 0)$$



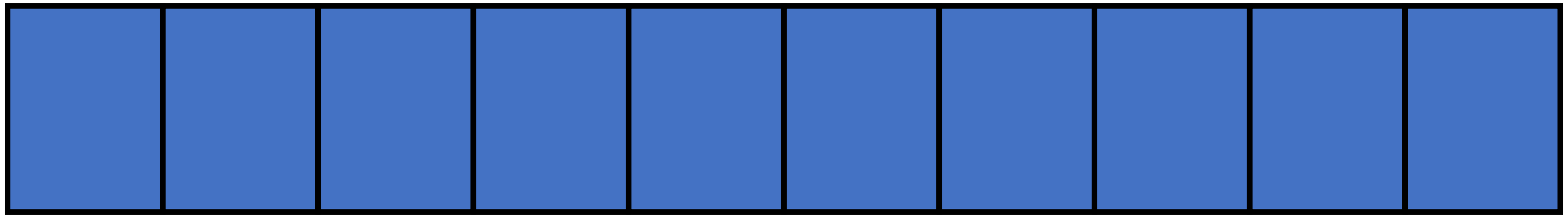
HashMap

Keys are uniformly distributed in a hashtable

Key: Timestamp; Value: Action

A = (10:14, Deposit)

$P(h(A) = 0) = 1/10$



HashMap

Find all transactions within a time range?

Find all deposits?

(10:14, Deposit)	(1:59, Withdraw)			(10:15, Withdraw)			(2:04, Deposit)	(10:16, Transfer)	

Sorted Map ADT

Similar items are stored near each other in clusters

firstEntry( ): Returns the entry with smallest key value (or null, if the map is empty).

lastEntry( ): Returns the entry with largest key value (or null, if the map is empty).

ceilingEntry(k): Returns the entry with the least key value greater than or equal to k (or null, if no such entry exists).

floorEntry(k): Returns the entry with the greatest key value less than or equal to k (or null, if no such entry exists).

lowerEntry(k): Returns the entry with the greatest key value strictly less than k (or null, if no such entry exists).

higherEntry(k): Returns the entry with the least key value strictly greater than k (or null if no such entry exists).

subMap(k_1, k_2): Returns an iteration of all entries with key greater than or equal to k_1, but strictly less than k_2.

Sorted Maps

Used on data that has a natural ordering – comparable elements, timed events

Implement with a sorted array - relies on binary search

Fast search for min/max key, key and nearby keys - $O(logn)$

Slow updates - $O(n)$

`java.util.NavigableMap`

`java.util.SortedMap`

Sorted Map Analysis

Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$; $O(\log n)$ if map has entry with given key
remove	$O(n)$
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$ where s items are reported
entrySet, keySet, values	$O(n)$

TreeMap

A sorted map implemented with a binary search tree

Recall the fundamental methods of a map:

get(k): Returns the value v associated with key k, if such an entry exists; otherwise returns null.

put(k, v): Associates value v with key k, replacing and returning any existing value if the map already contains an entry with key equal to k.

remove(k): Removes the entry with key equal to k, if one exists, and returns its value; otherwise returns null.

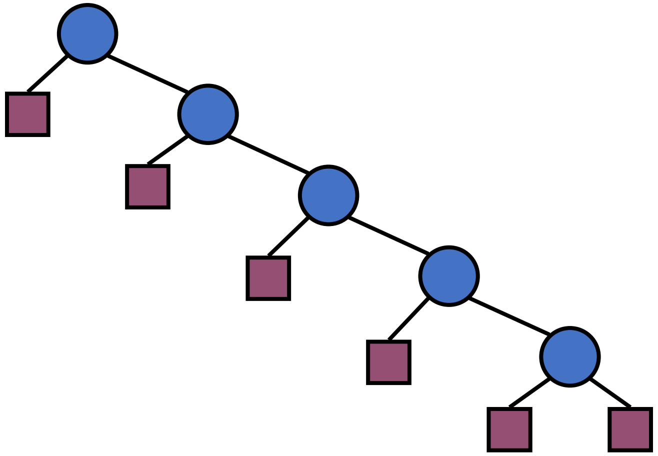
Binary Search Trees

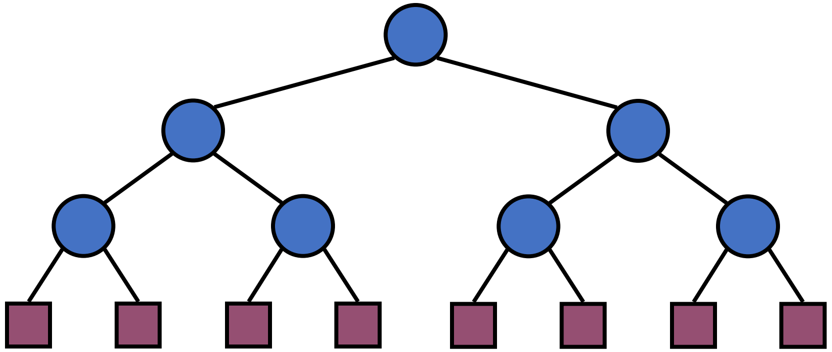
Performance is directly affected by the height of tree

All operations are $O(h)$

- $h = O(n)$ worst case
- $h = O(logn)$ best case

Expected $O(logn)$ if tree is balanced



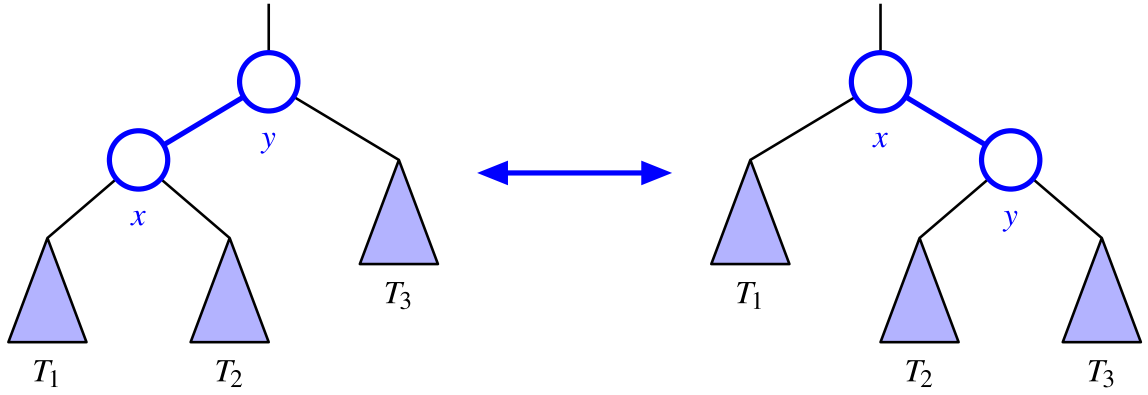


Balanced Search Trees

A variety of algorithms that augments a standard BST with occasional operations to reshape and reduce height

Rotation:

- move a child to be above its parent and relink subtrees to maintain BST order
- $O(1)$



Tree Rotation

Rotation can be to the right or left

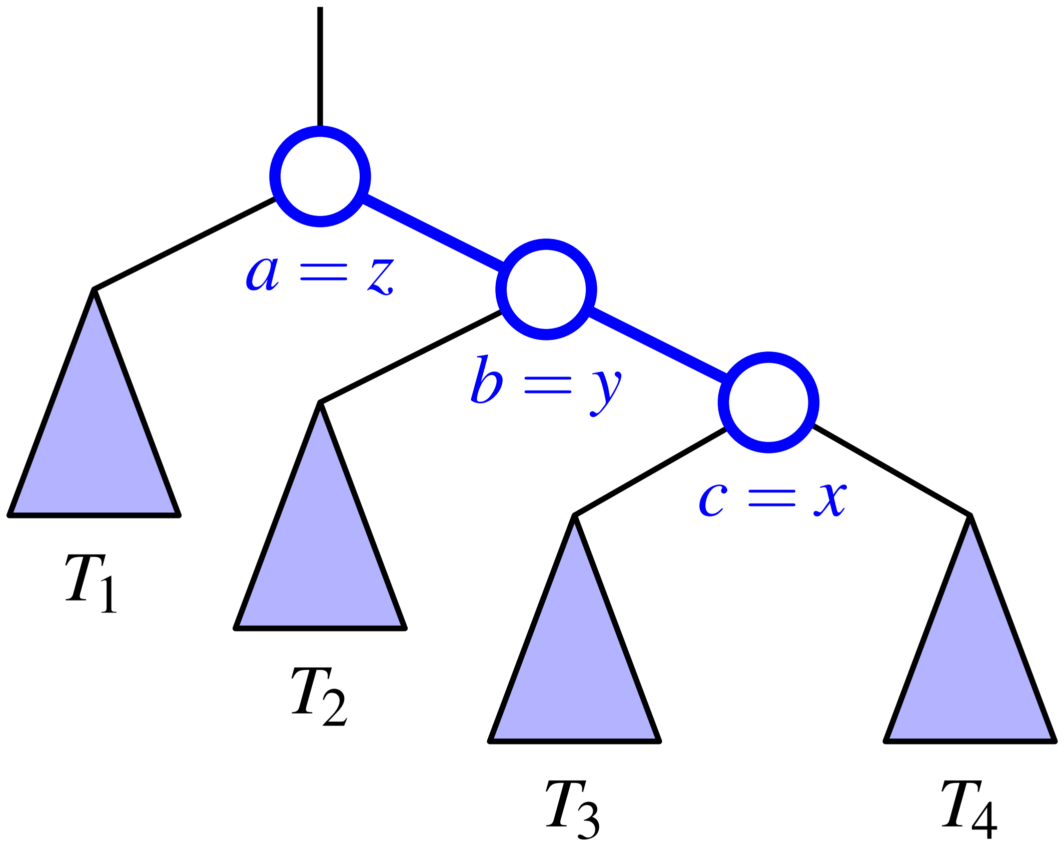
Rotate reduces/increases the depth of nodes in subtrees T_1 and T_3 by 1

Rotation maintains BST order

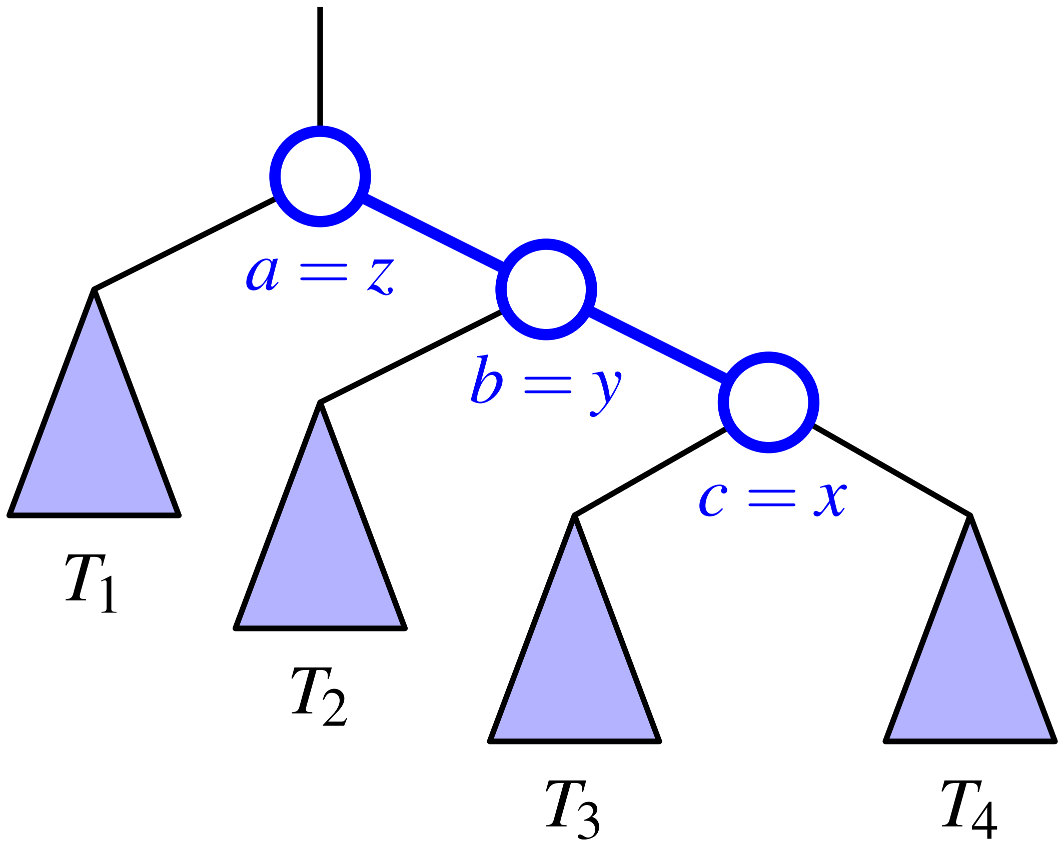
Rotate is $O(1)$

One or more rotations can be combined to provide broader rebalancing

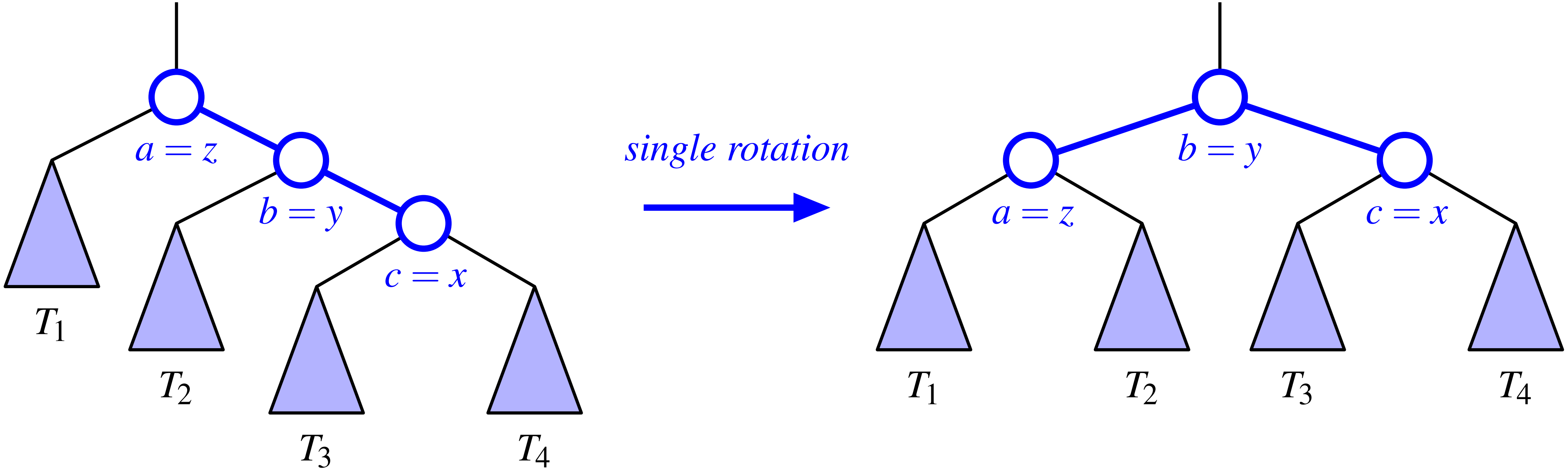
Tri-node restructuring: a node x, its parent y and its grandparent z


$a = z$
$b = y$
$c = x$
T_1
T_2
T_3
T_4

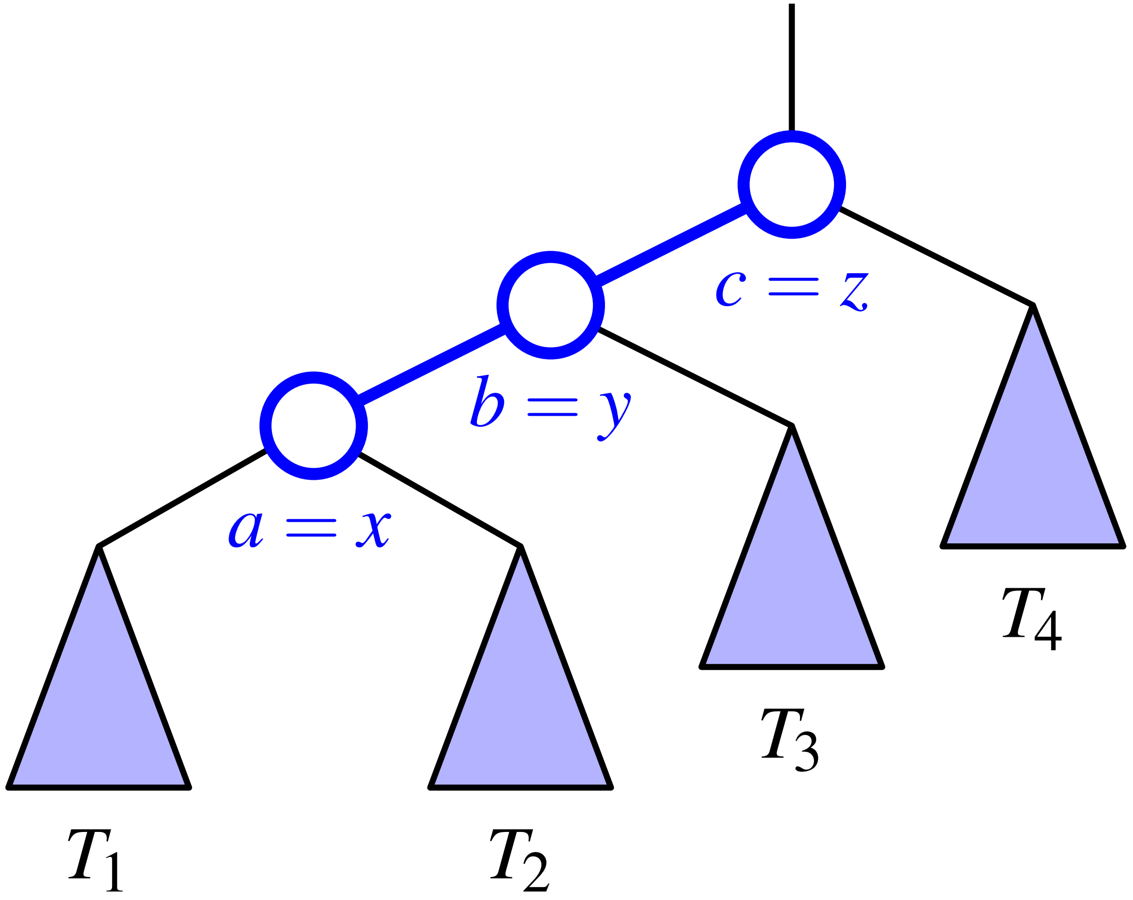
Single Rotation (around z)



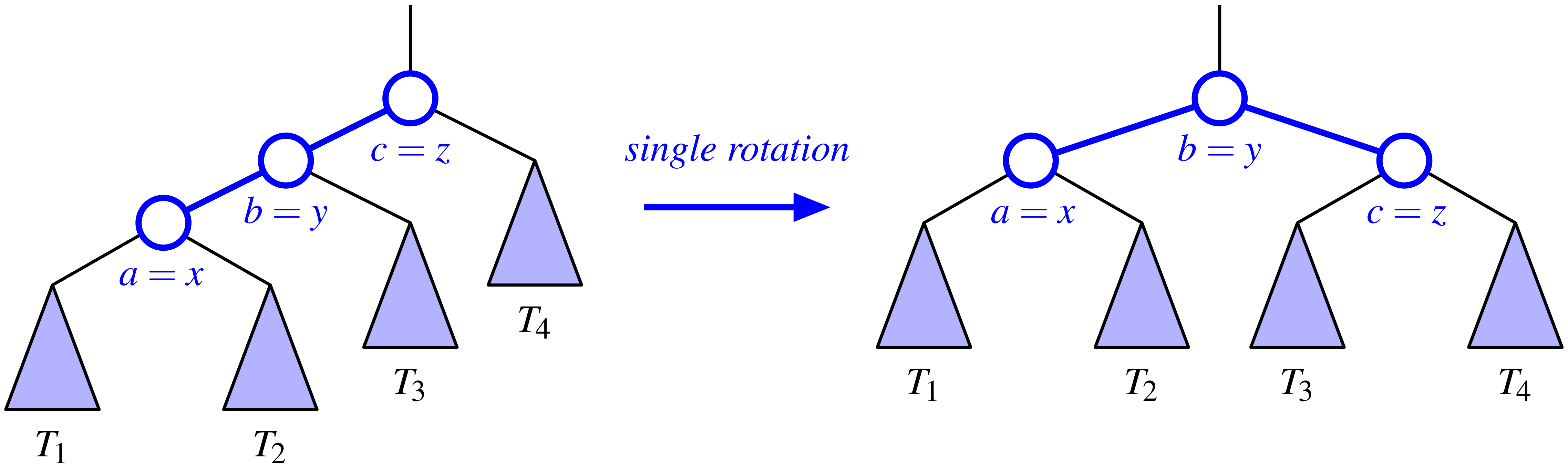
Single Rotation (around z)



Single Rotation (around z)



Single Rotation (around z)



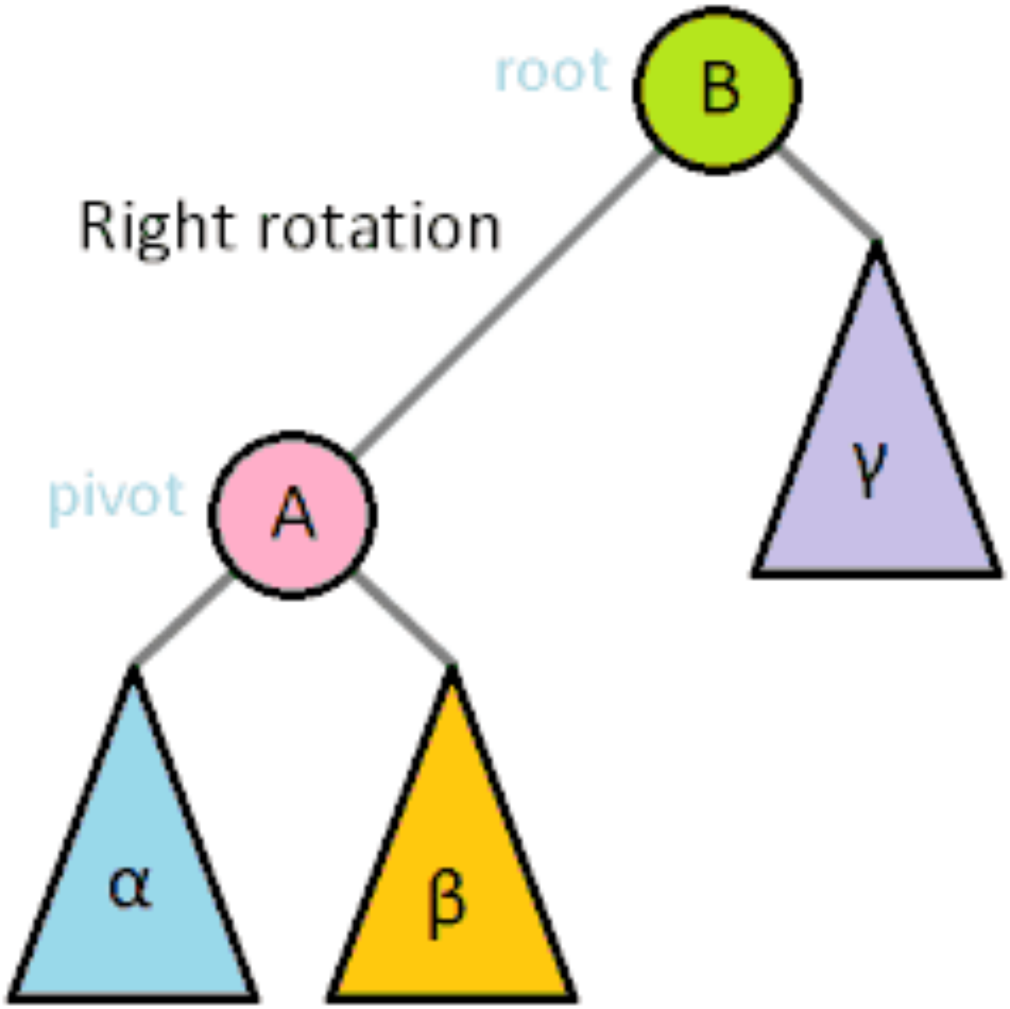
Rotations

Right rotation:

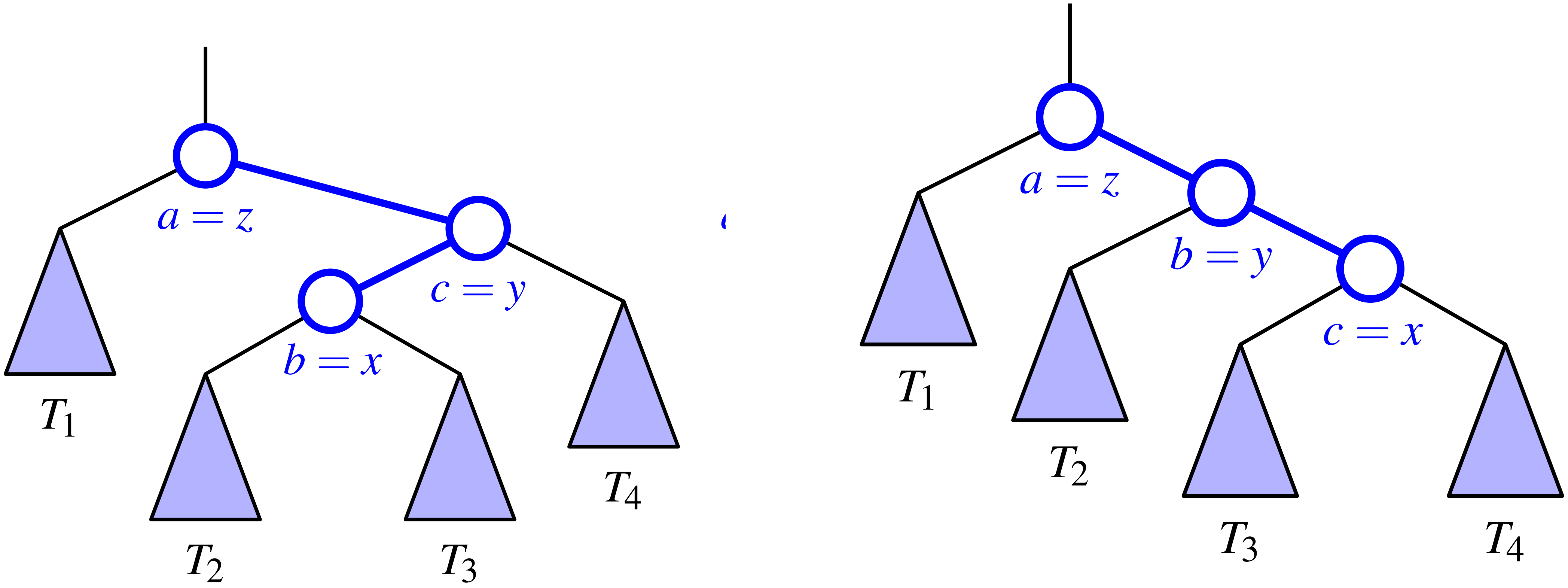
- Root node's left child becomes the new root
- Root node becomes the left child's right child

Left rotation:

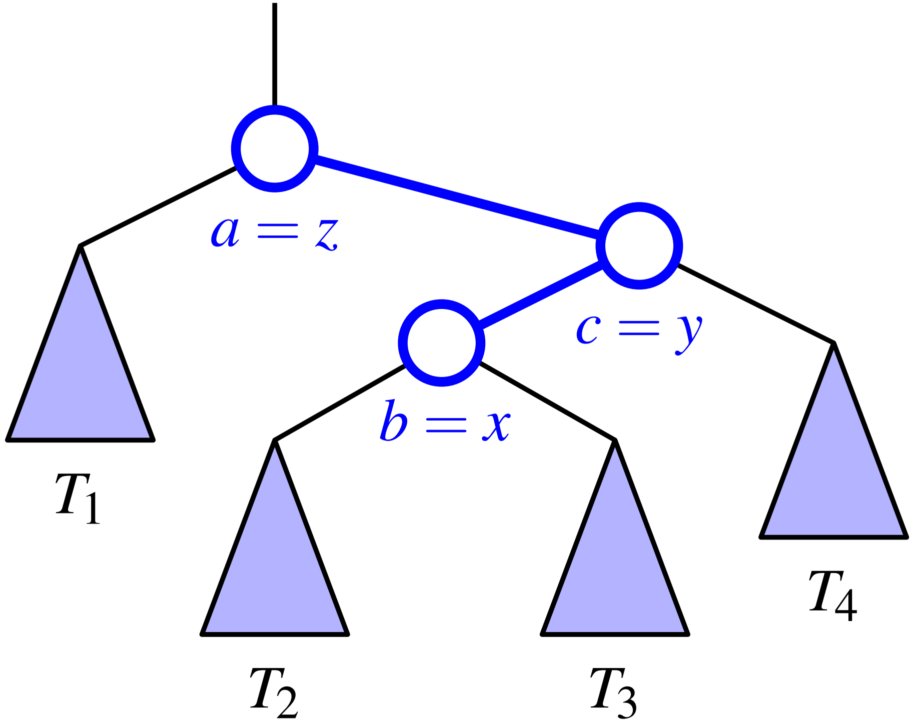
- Root node's right child becomes the new root
- Root node becomes the right child's left child



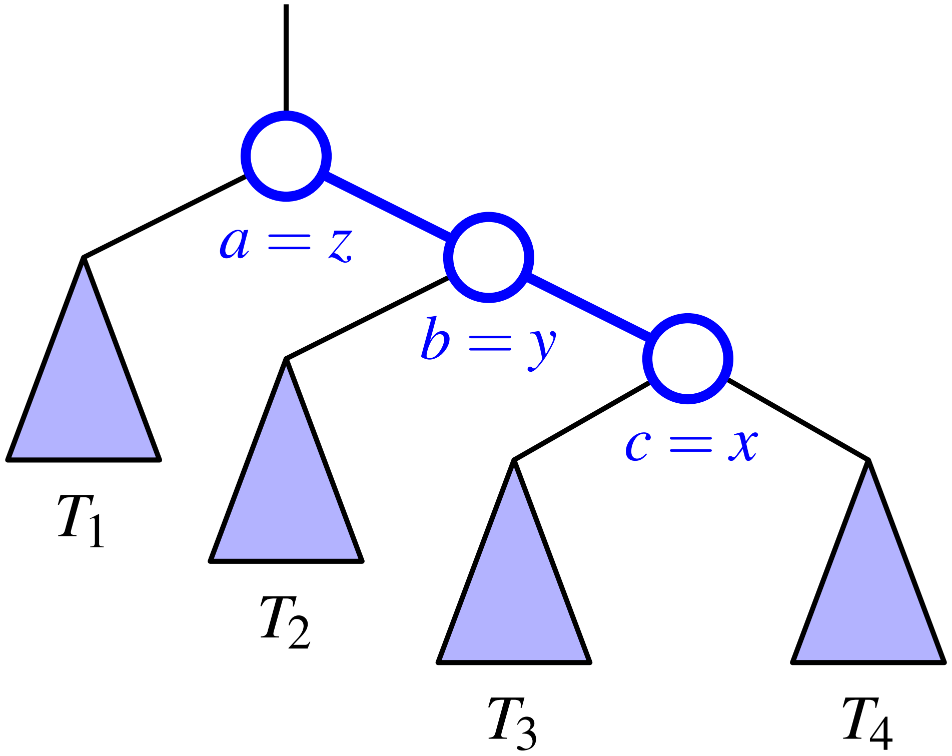
Rotation



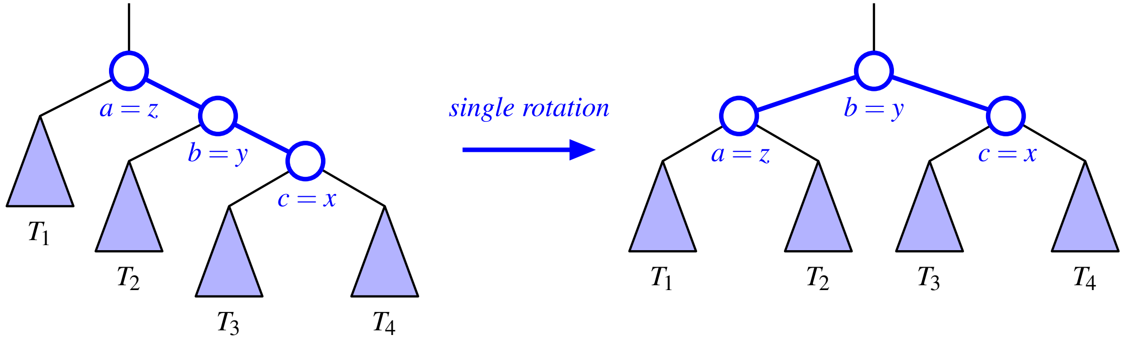
Rotation



$a = z$
$c = y$
$b = x$
T_1
T_2
T_3
T_4

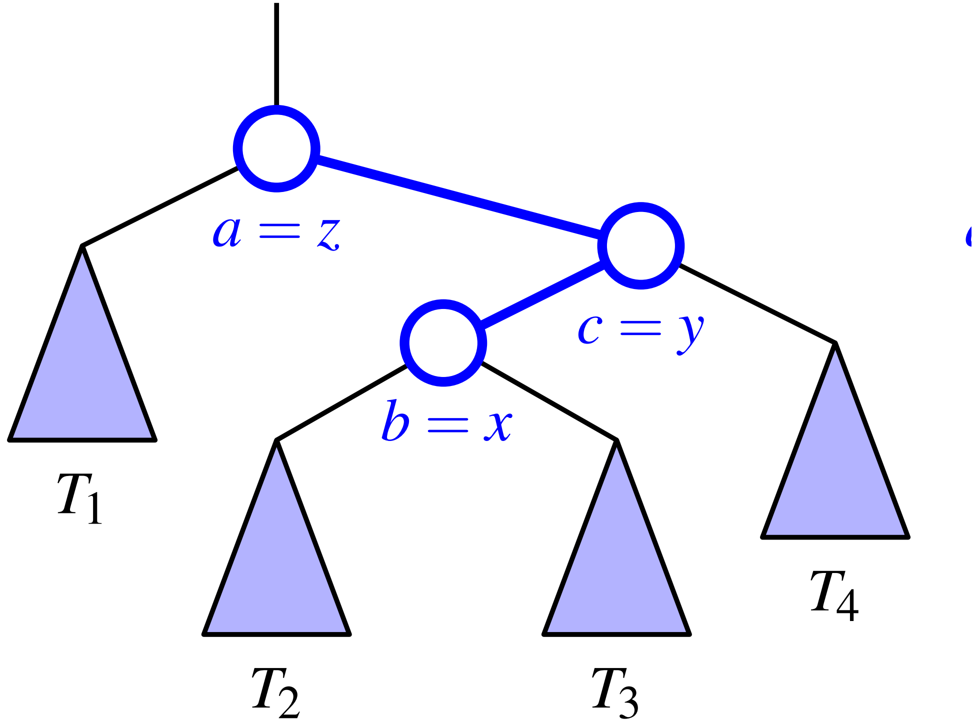


$a = z$
$b = y$
$c = x$
T_1
T_2
T_3
T_4

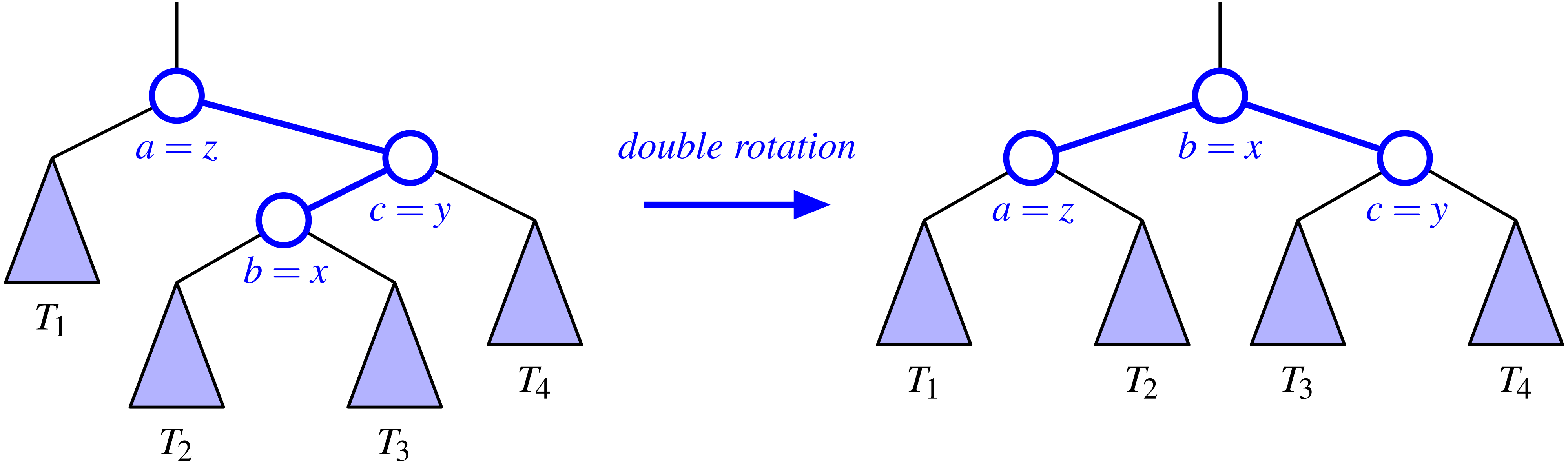


$a = z$
$b = y$
$c = x$
T_1
T_2
T_3
T_4
single rotation
$b = y$
$a = z$
$c = x$
T_1
T_2
T_3
T_4

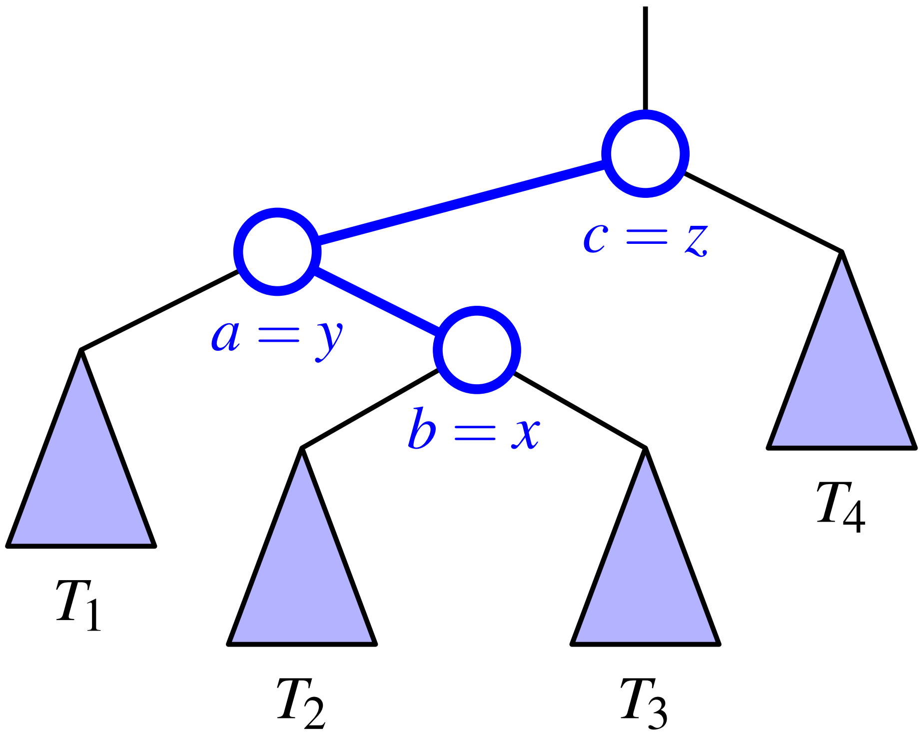
Double Rotation



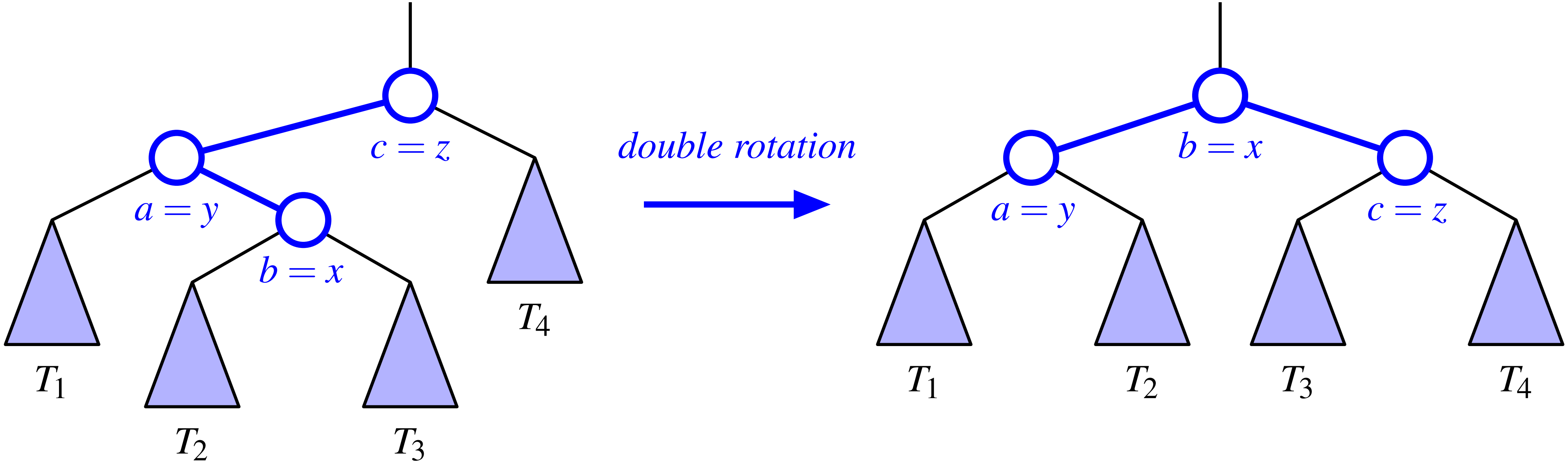
Double Rotation



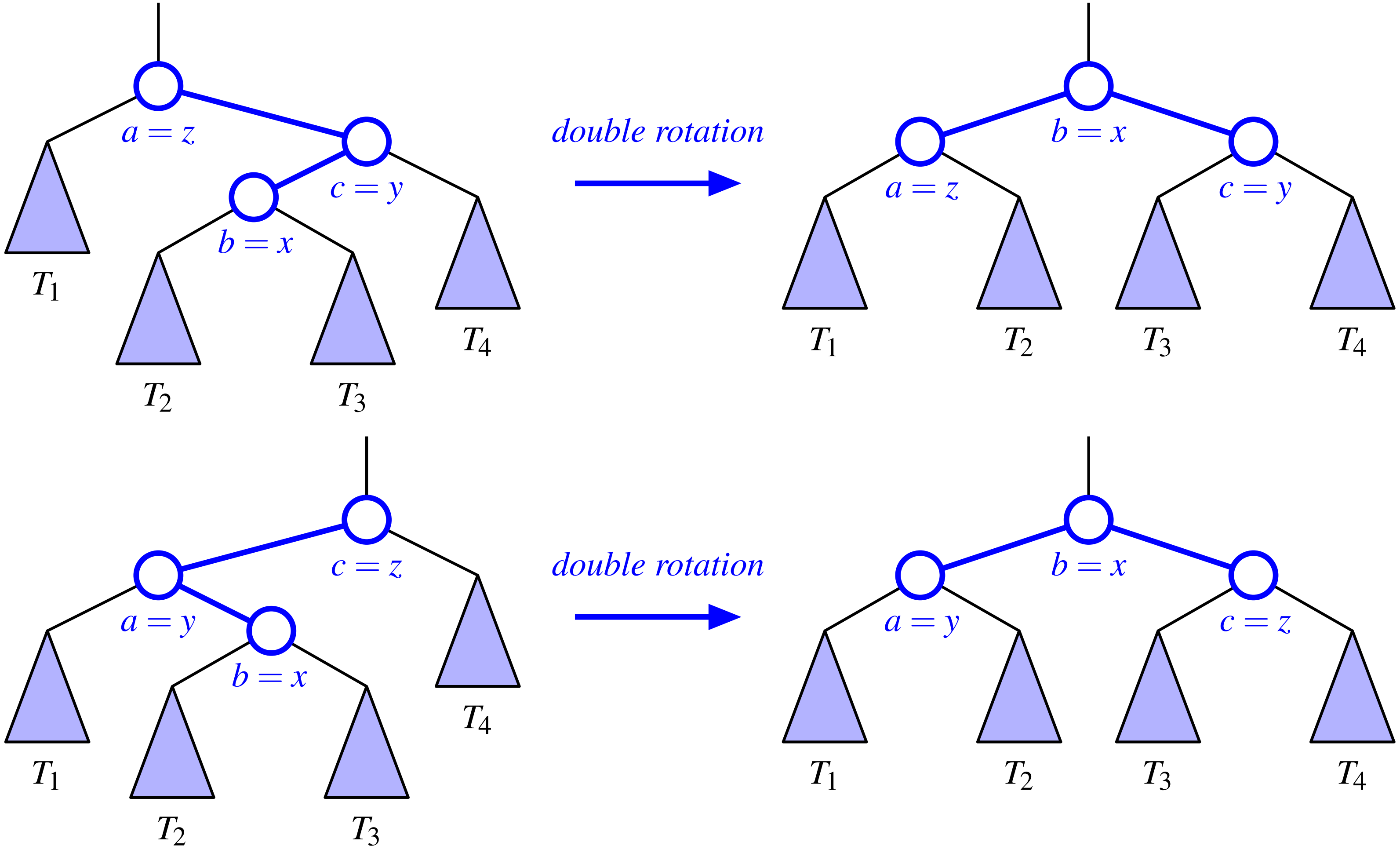
Double Rotation (around z)



Double Rotation (around z)

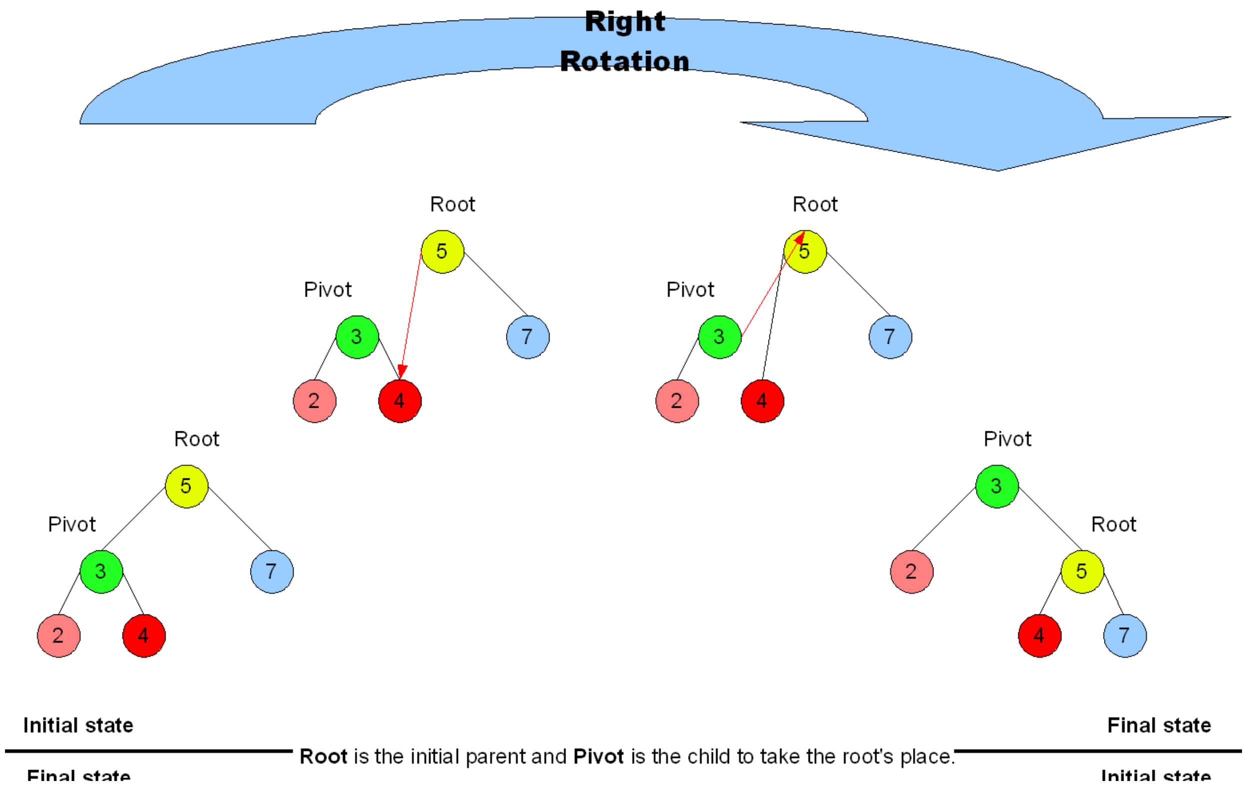


Double Rotation (around z)



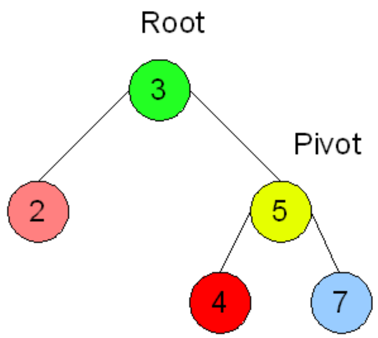
Tree Rotations

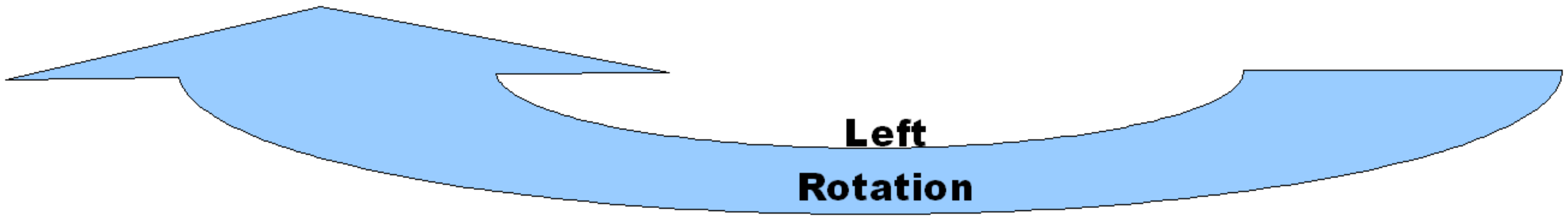
```
rotateRight(r):
  if (r.left==null) return
  p = r.left
  r.left = p.right
  p.right = r

  // set parent
  if r.parent == null
    root = p
    p.parent = null
  else
    if(r.parent.left == r)
      r.parent.left=p
    else
      r.parent.right=p
```



Initial state	**Final state**
Final state	**Initial state**

Root is the initial parent and **Pivot** is the child to take the root's place.

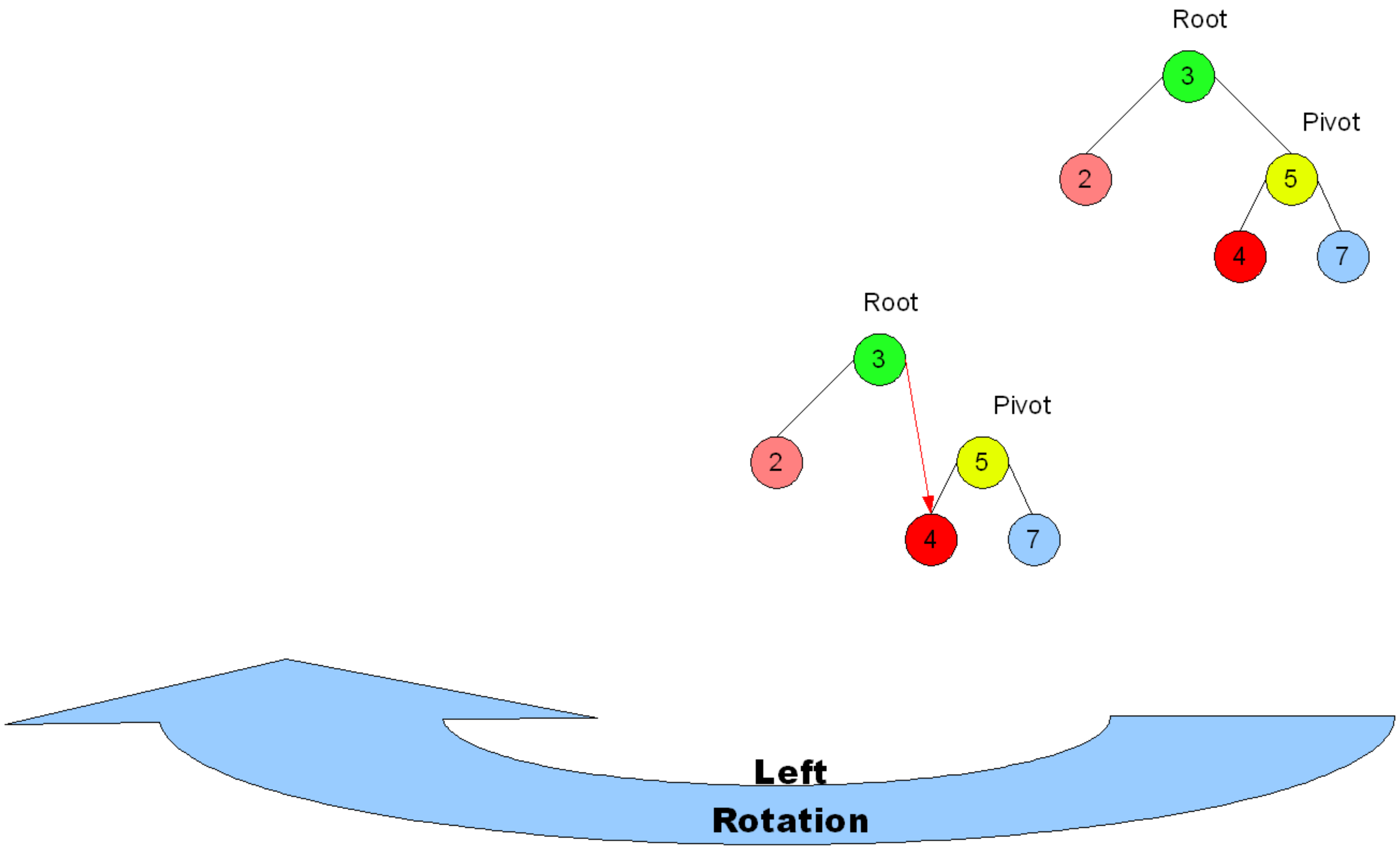


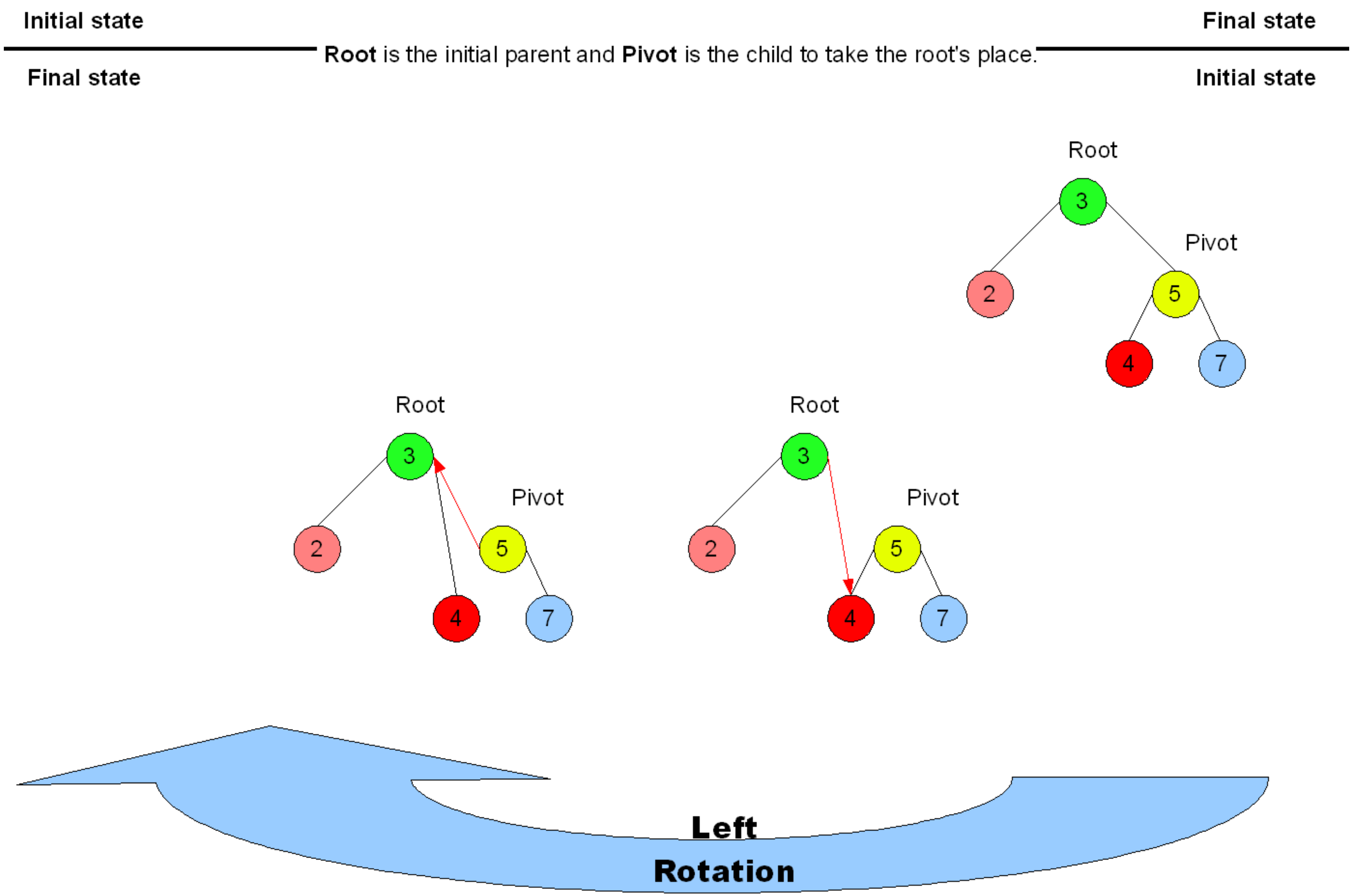


Initial state | **Final state**

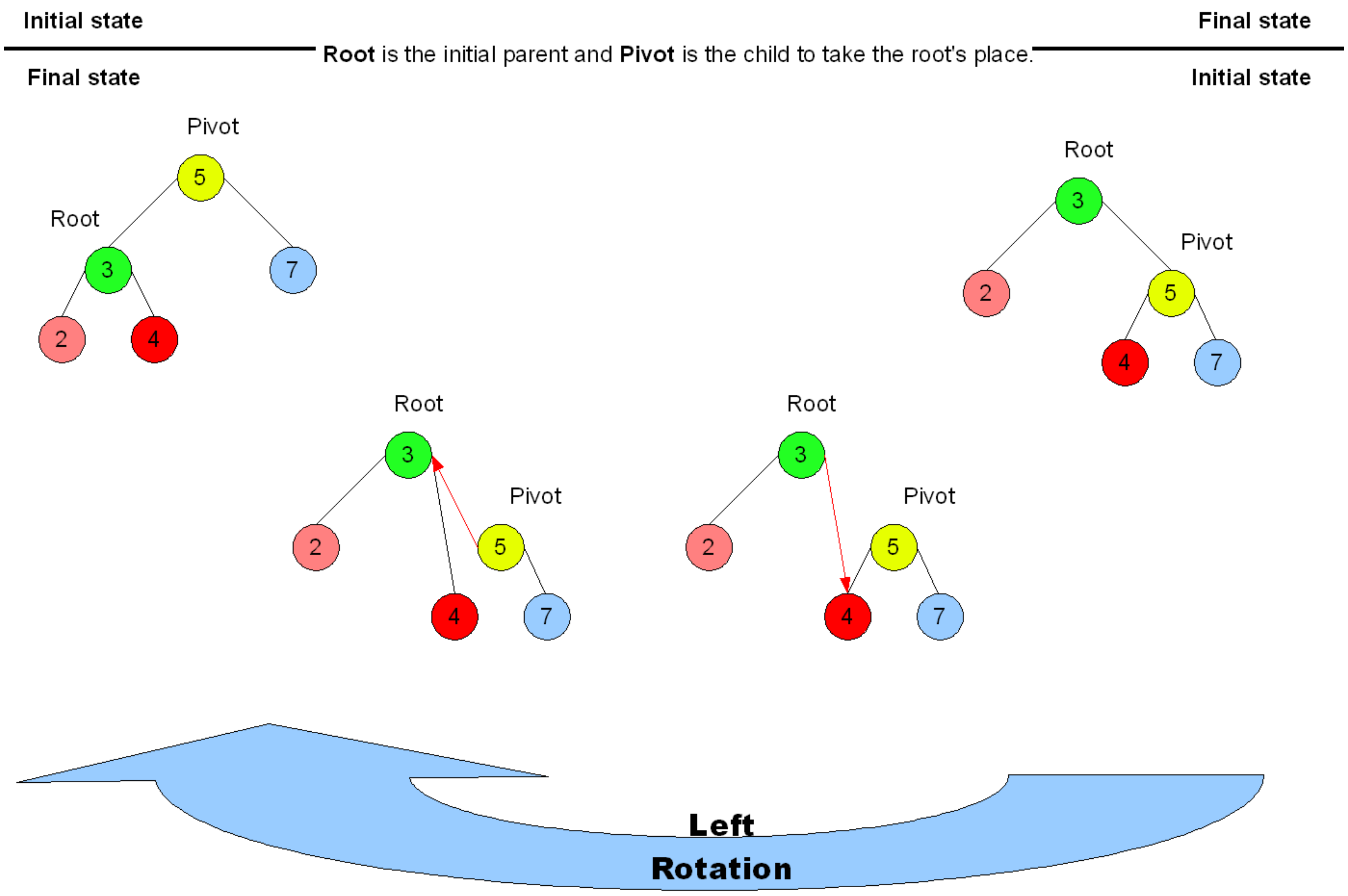
Root is the initial parent and **Pivot** is the child to take the root's place.

Final state | **Initial state**





Initial state
Final state
Root is the initial parent and Pivot is the child to take the root's place.
Final state
Initial state
Root
3
Pivot
2
5
4
7
Root
3
Pivot
2
5
4
7
Root
3
Pivot
2
5
4
7
Left
Rotation


Initial state
Final state
Root is the initial parent and Pivot is the child to take the root's place.
Final state
Initial state
Pivot
5
Root
3
7
2
4
Root
3
Pivot
2
5
4
7
Root
3
Pivot
2
5
4
7
Root
3
Pivot
2
5
4
7
Left
Rotation

AVL Tree

Height of a subtree is the number of edges on the longest path from subtree root to a leaf

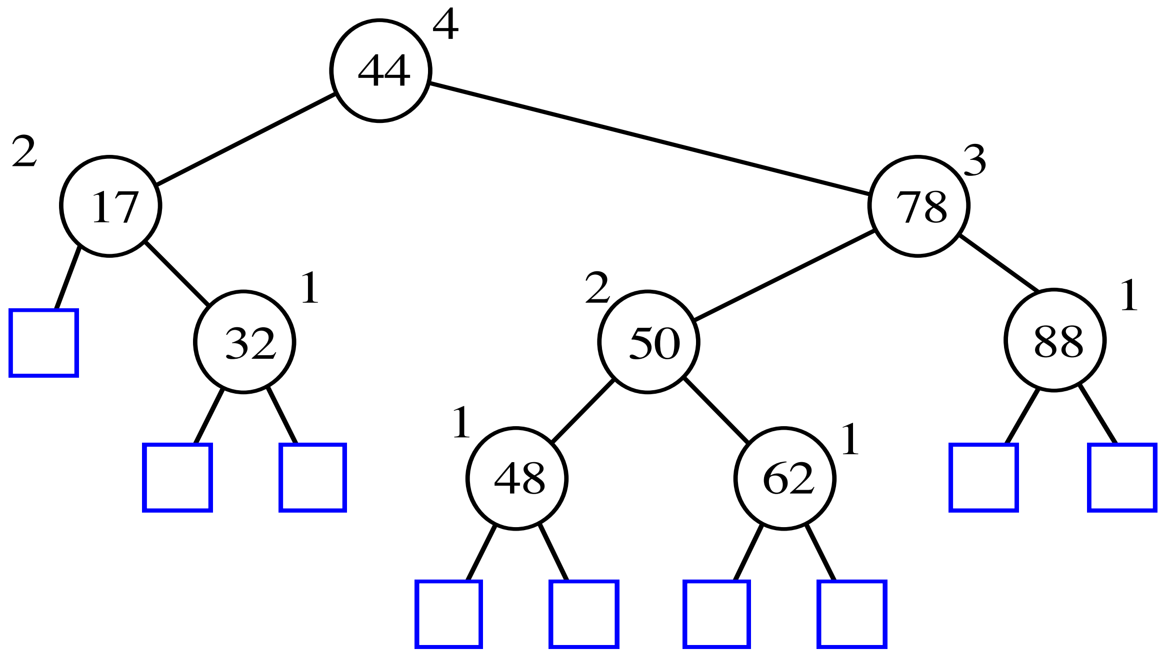
Height-balance property

- For every internal node, the heights of the two children differ by at most 1

Any binary tree satisfying the height-balance property is an AVL tree

AVL Tree Example

- leaves are sentinels and have height 0



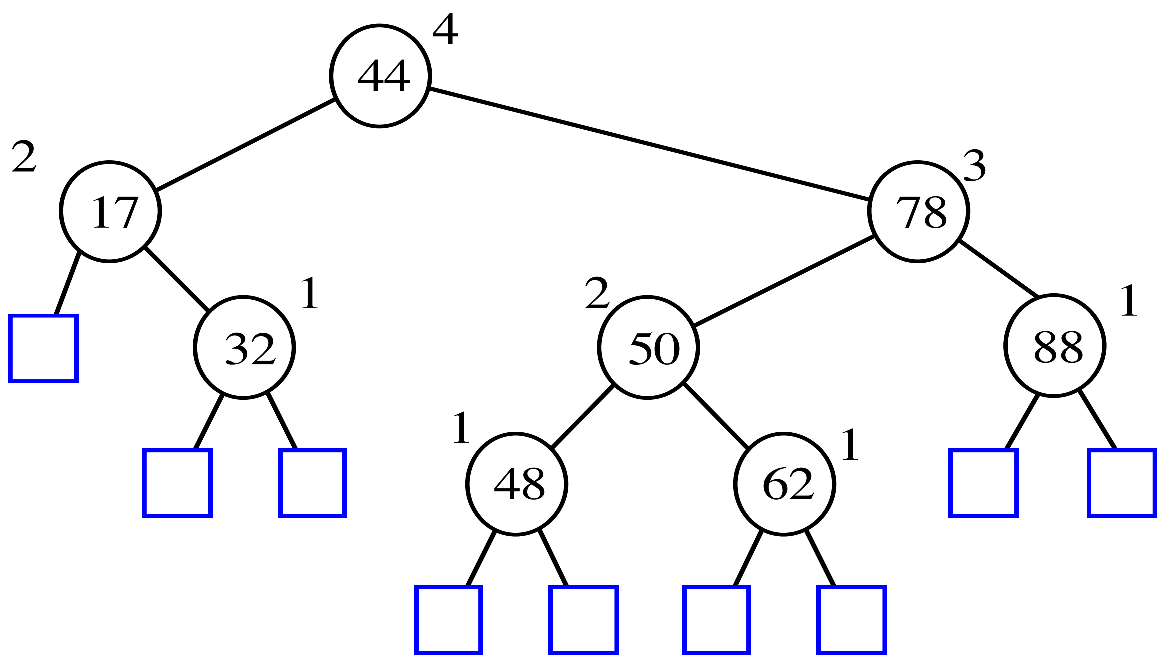
AVL height

The height of an AVL is $O(logn)$

$n(h)$ denotes the number of minimum internal nodes for an AVL with height h

- $n(1) = 1$ and $n(2) = 2$
- $n(h) = 1 + n(h-1) + n(h-2)$
- $n(h) > 2 \cdot n(h-2) > 2^i \cdot n(h-2i)$
- $h - 2i = 1 \implies i = \frac{h}{2} - 1$
- $\log\big(n(h)\big) = \frac{h}{2} - 1 \implies h < 2\log\big(n(h)\big) + 1$

Insert 54



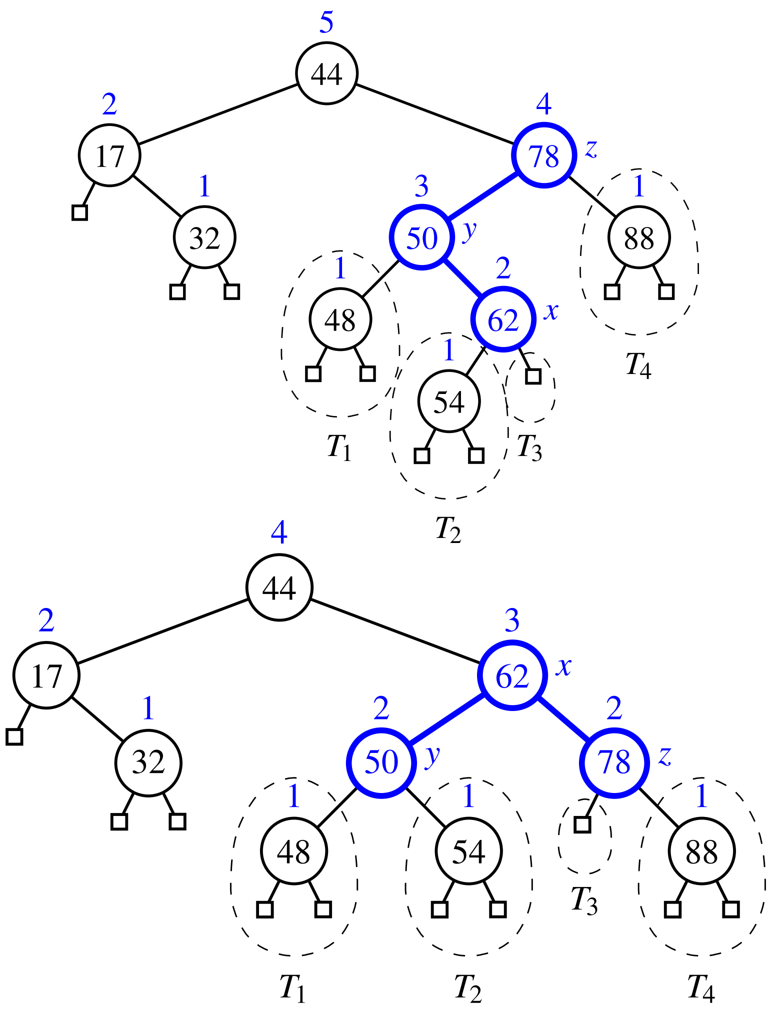
Insertion (54)

New node always has height 1

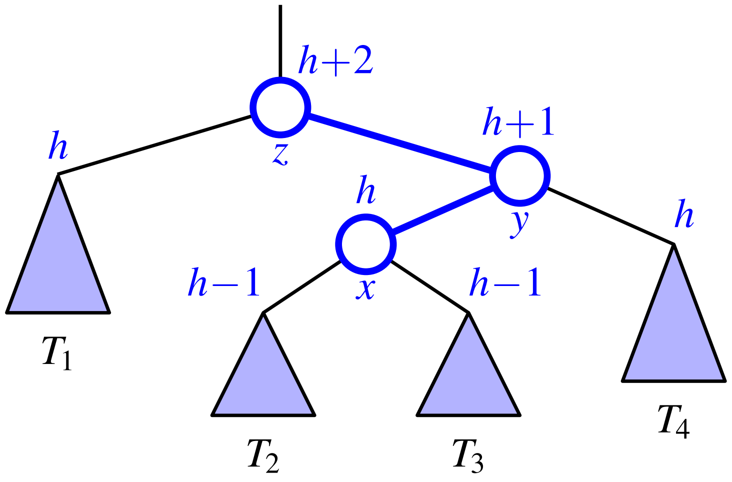
Parent may change height

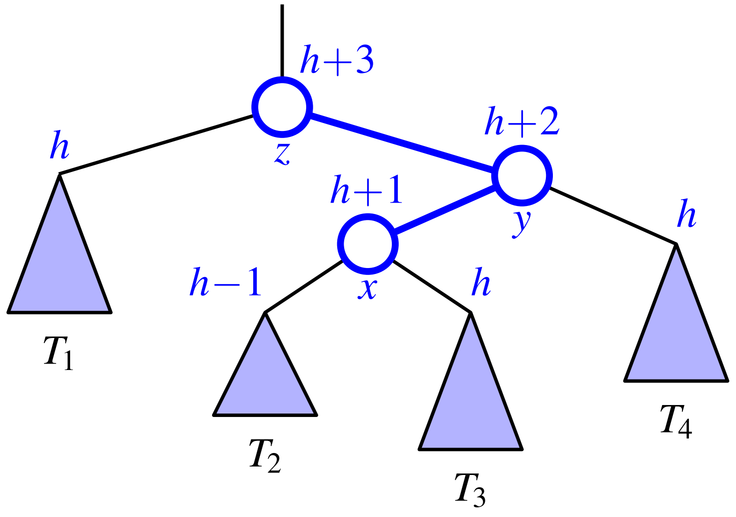
All ancestors may become unbalanced

Perform rotations for unbalanced ancestors



$O(1)$ Rotation Restores Global Balance

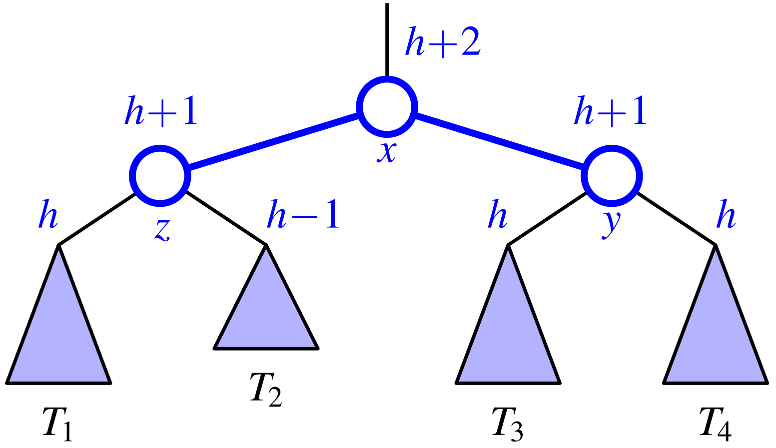




Insert into T_3

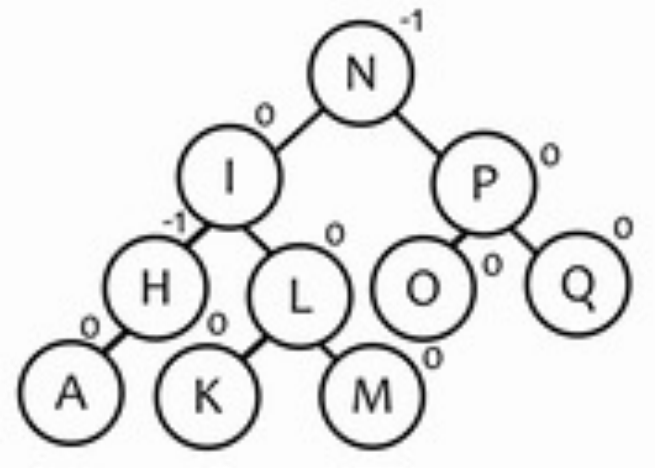
After rebalance:

- x, y and z are balanced after
- root of subtree returns to height $h + 2$, as before



Exercise

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

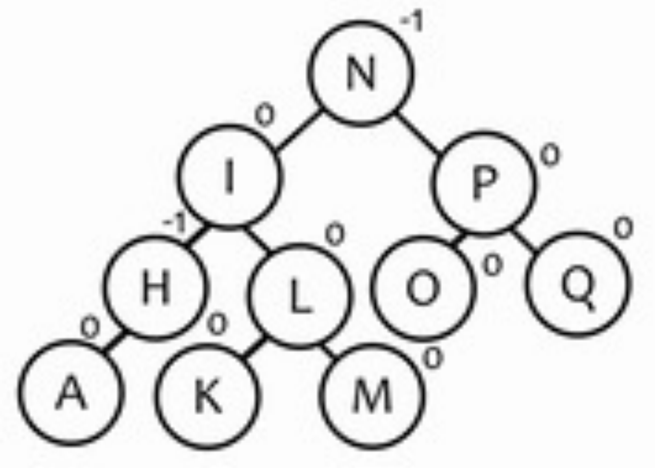


- AVL balance marked on nodes
- balance(n) = height of right subtree – height of left subtree
- AVL balance property: |balance(n)| ≤ 1

AVL Animation

Exercise

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



- AVL balance marked on nodes
- balance(n) = height of right subtree – height of left subtree
- AVL balance property: |balance(n)| ≤ 1

Rebalance: no null checks

```
rebalance(n):
  updateHeight(n) // update height from children
  lh = n.left.height rh = n.right.height
  if (lh > rh+1) // left subtree too tall
    llh = n.left.left.height lrh = n.left.right.height
    if (llh >= lrh)
      return rotateRight(n)  //left-left
    else
      return rotateLeftRight(n) //left-right
  else if (rh > lh+1) // right subtree too tall
    // … symmetric
  else return n // no rotation
```

Helpers

```
rotateRight(r):
  p = r.left
  r.left = p.right
  p.right = r
  updateHeight(r)
  updateHeight(p)
  // let caller set parent
  // return new subtree root
  return p
```

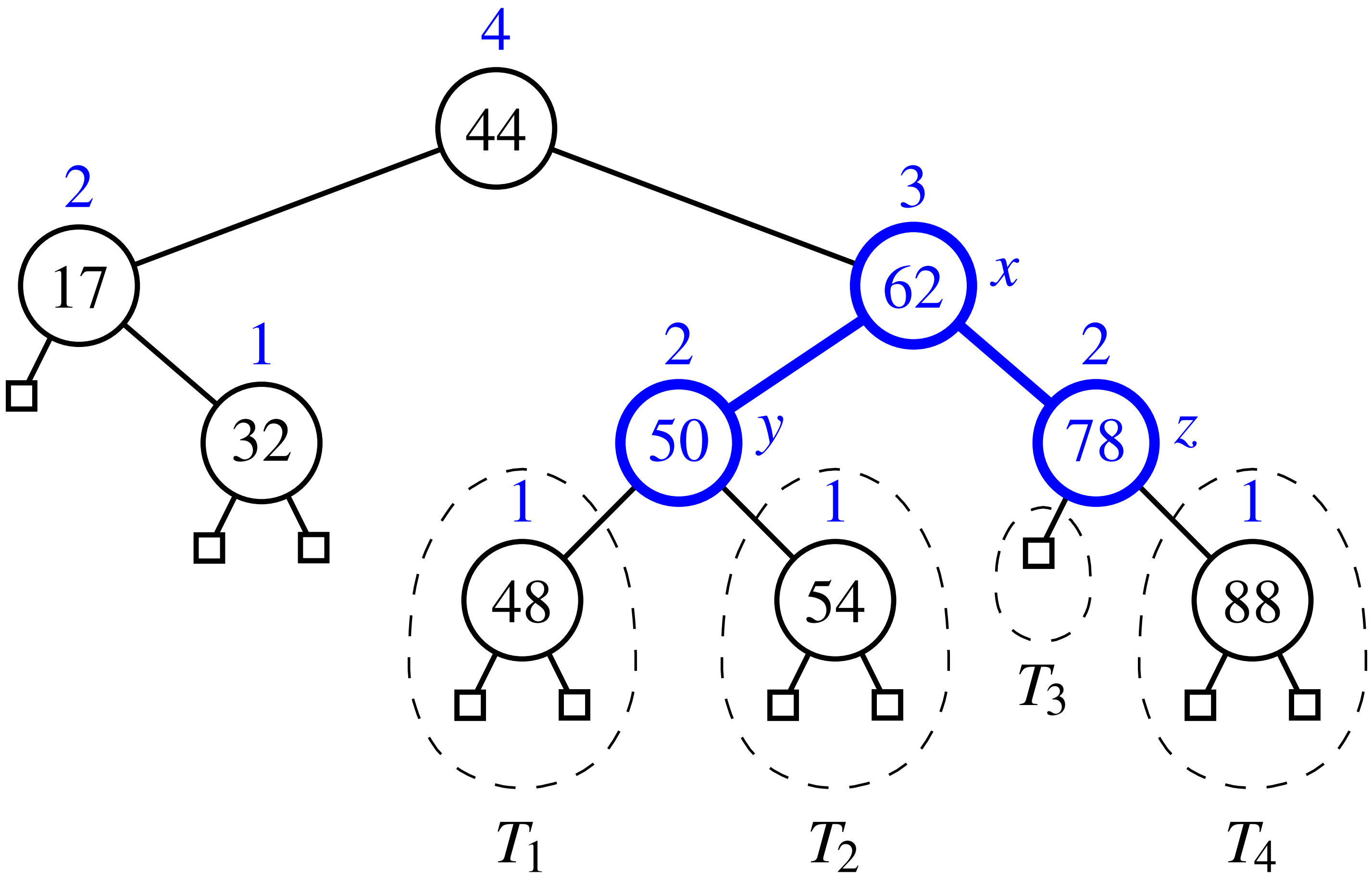
```
updateHeight(n):
  lh = n.left.height
  rh = n.right.height
  height = 1+max(lh, rh)
```

```
rotateLeftRight(r):
  r.left = rotateLeft(r.left)
  return rotateRight(r)
```

Insert with `parent`

```
insertRec(root, key):
  if root == null:
    return new Node(key)
  if root.key > key:
    root.left = insertRec(root.left, key)
    root.left.parent = root
  else
    root.right = insertRec(root.right, key)
    root.right.parent = root
  return root
```

Delete 32



Deletion

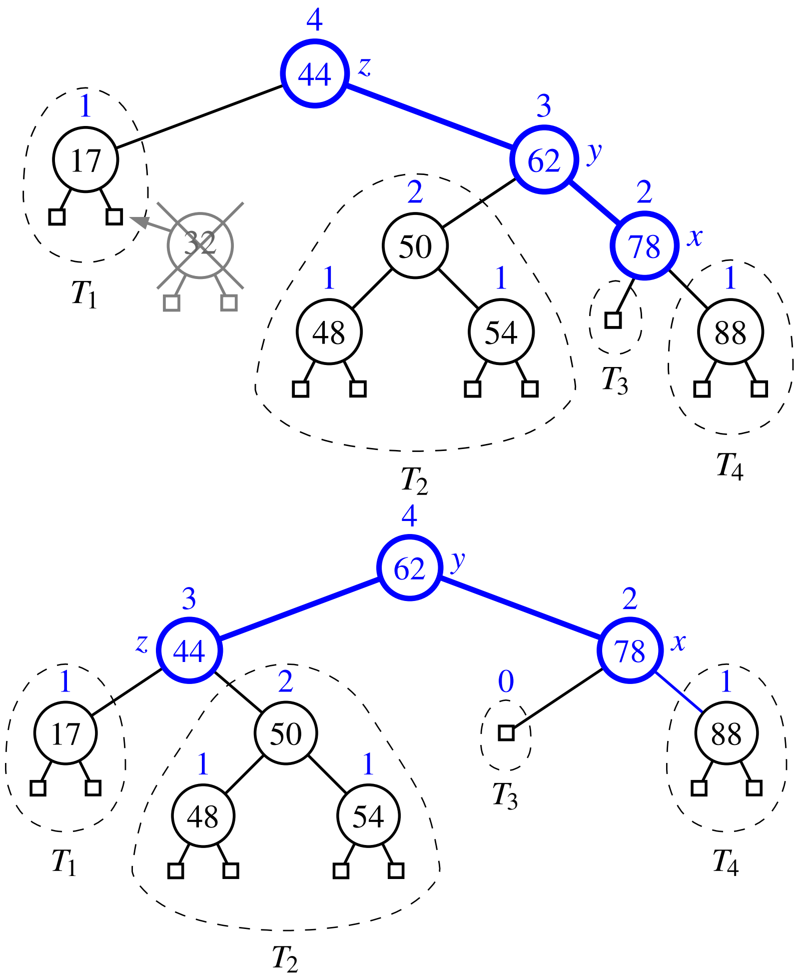
Deletion structurally removes a node with 0 or 1 child

- predecessor has 0 or 1 left child
- successor has 0 or 1 right child

Deletion may reduce the height of parent

Ancestors may become unbalanced

Rotate to rebalance just like insertion



$O(logn)$ Rotations

Unlike insertion where rotation of the nearest unbalanced ancestor restores the balance globally

On deletion, rotation of the nearest unbalanced ancestor only guarantees balance locally to the subtree

Worst-case requires $O(logn)$ rotations up the tree to restore balance globally

Performance of `AVLTreeMap`

Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(\log n)$
firstEntry, lastEntry	$O(\log n)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$
entrySet, keySet, values	$O(n)$

Book's Implementation of AVL

- 17 classes!
- Interfaces
  - `Entry`
  - `Position`
  - `Queue`
  - `Tree`
  - `BinaryTree`
  - `Map`
  - `SortedMap`
- Abstract classes:
  - `AbstractTree`
  - `AbstractBinaryTree`
  - `AbstractMap`
  - `AbstractSortedMap`
- Concrete classes
  - `SinglyLinkedList`
  - `LinkedQueue`
  - `LinkedBinaryTree`
  - `TreeMap`
  - `AVLTreeMap`