CS151 Intro to Data Structures

Balanced Search Trees, AVL Trees

Announcements

Last lab will be this week

HW07 due Friday 12/08
    Hashmaps & Sorting

HW08 due 12/14

Faculty Interview/Mock Lecture

- Friday 12/08 – 11-11am
- Binary Search Tree
- Location: TBD
- Tea & Snacks

Outline

Double Hashing Review (Homework 07)

Balanced Binary Trees

AVL Trees

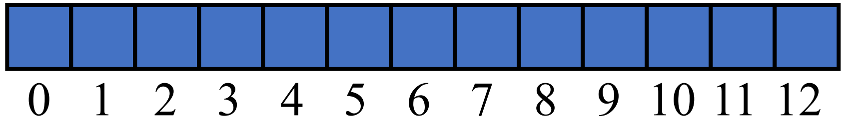
Splay Trees

Red-Black Trees

HW07

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$

k	$h(k)$	$d(k)$	Probes



HW07

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18

k	$h(k)$	$d(k)$	Probes

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44

k	$h(k)$	$d(k)$	Probes
18	5	3	5
41	2	1	2
22	9	6	9

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	
32	6	3	6	

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32, 31

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	
32	6	3	6	

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32, 31

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32, 31, 73

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0

A8

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32, 31, 73

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Outline

Double Hashing Review (Homework 07)

Balanced Binary Trees

AVL Trees

Splay Trees

Red-Black Trees

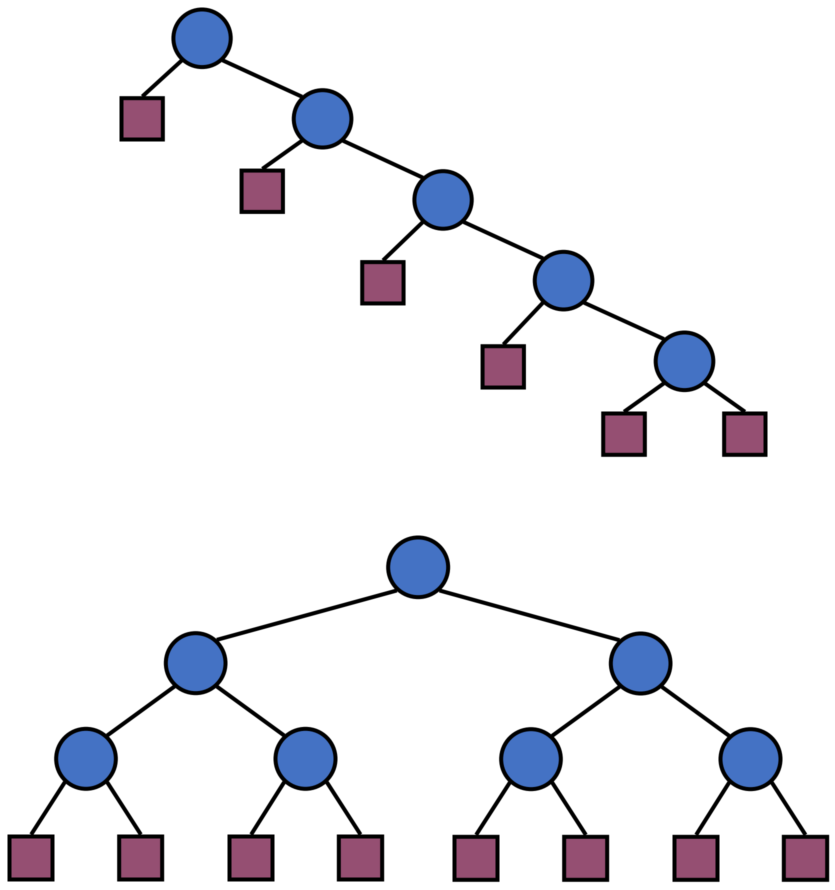
Binary Search Trees

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 Trees

- The difference between the height of the left and right subtree for any node is at most 1

- Left subtree of a node is balanced

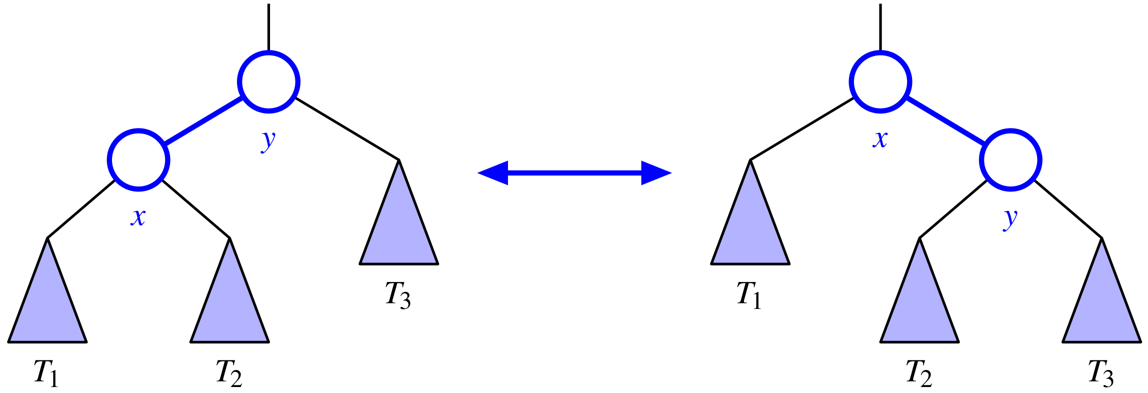
- Right subtree of a node is balanced

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

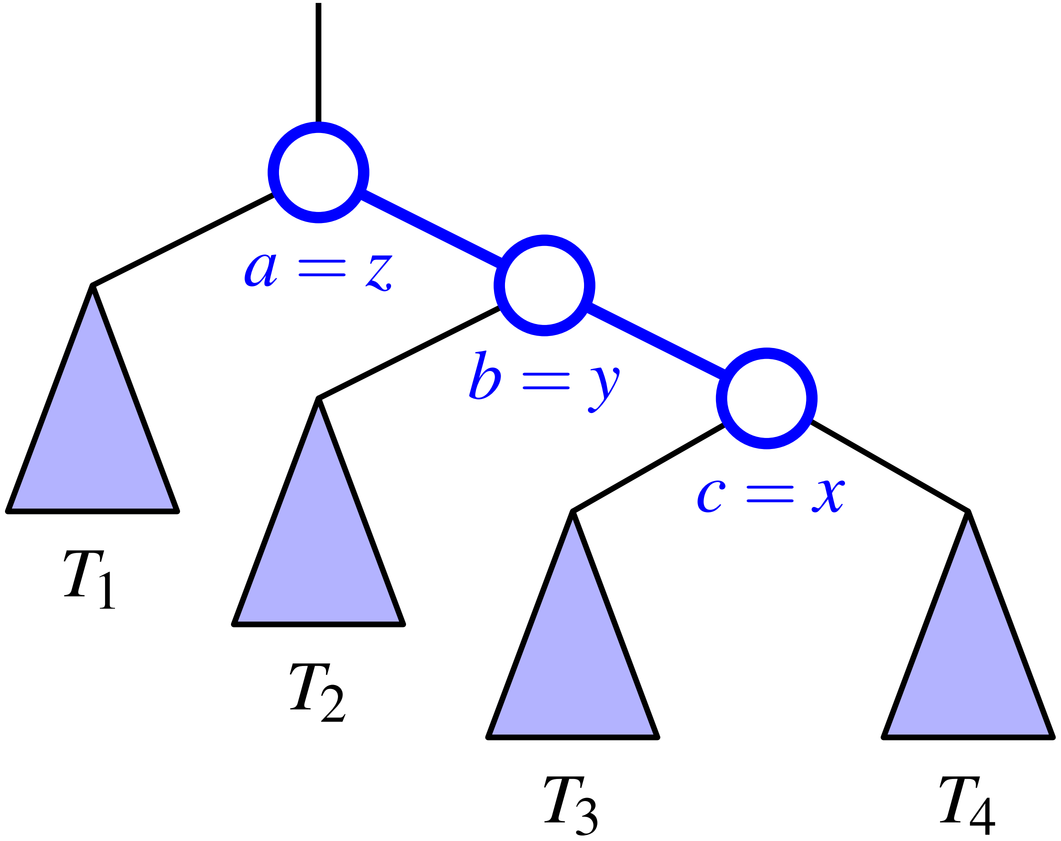
Rotate reduces/increases the depth of nodes in subtrees T_1 and T_3 by 1

Rotation maintains BST order

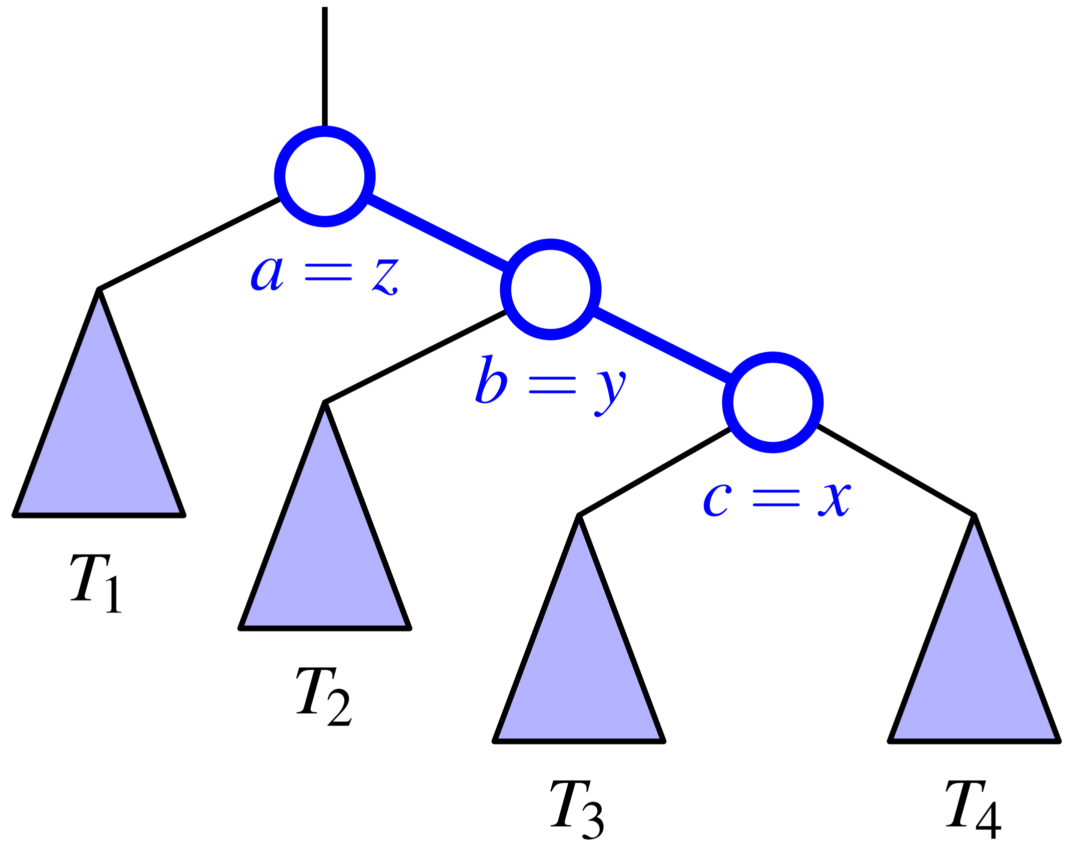
Rotate is $O(1)$

One or more rotations can be combined to provide broader rebalancing

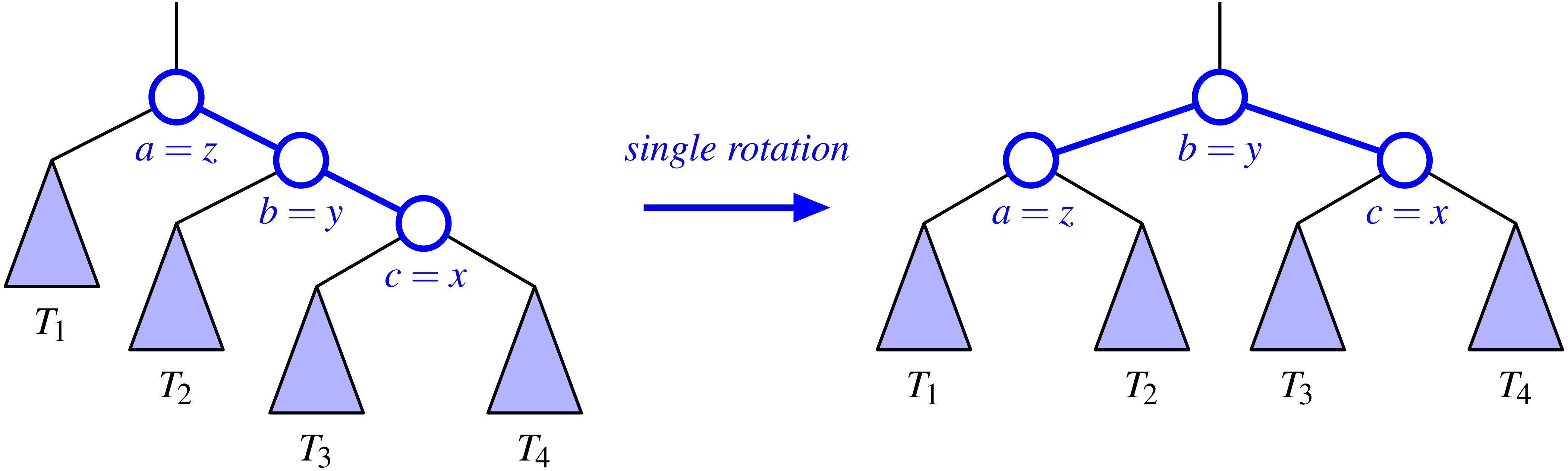
Tri-node restructuring: a node x, its parent y and its grandparent z


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

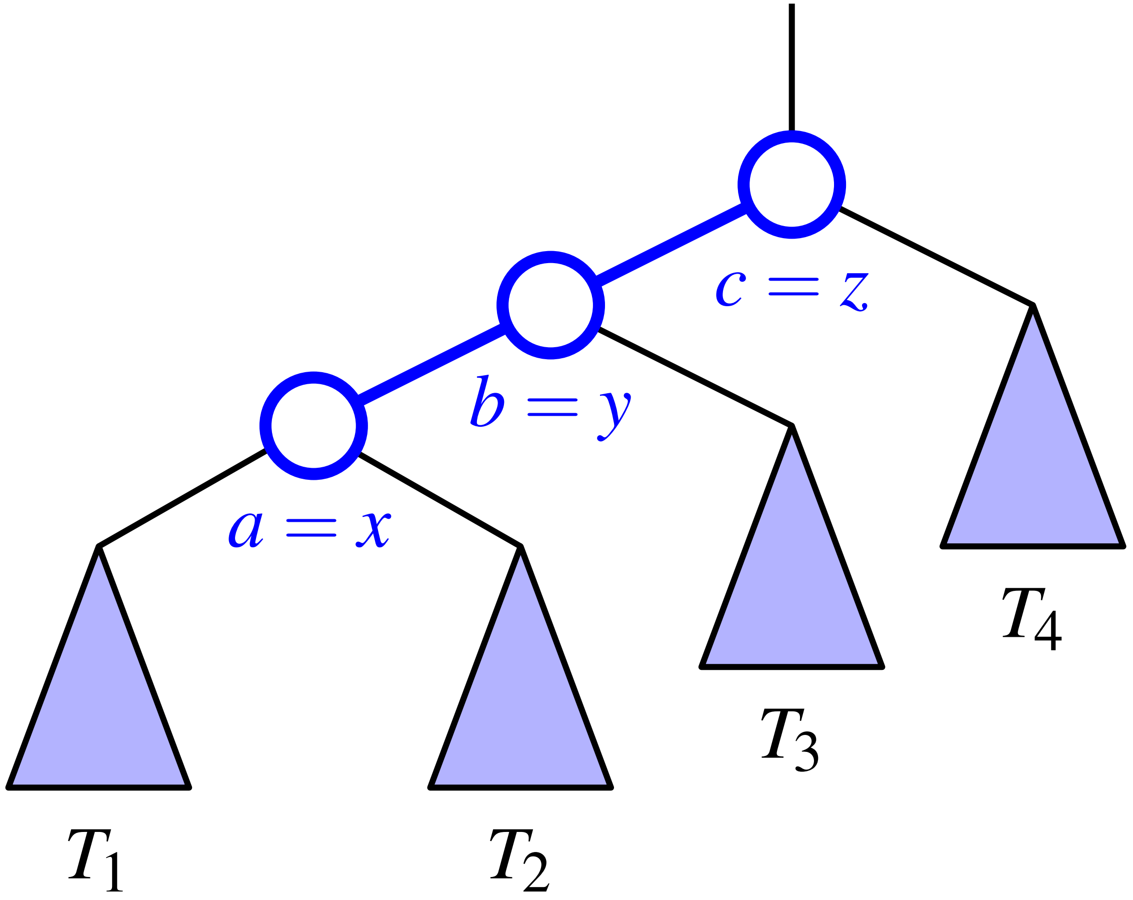
Single Rotation (around z)



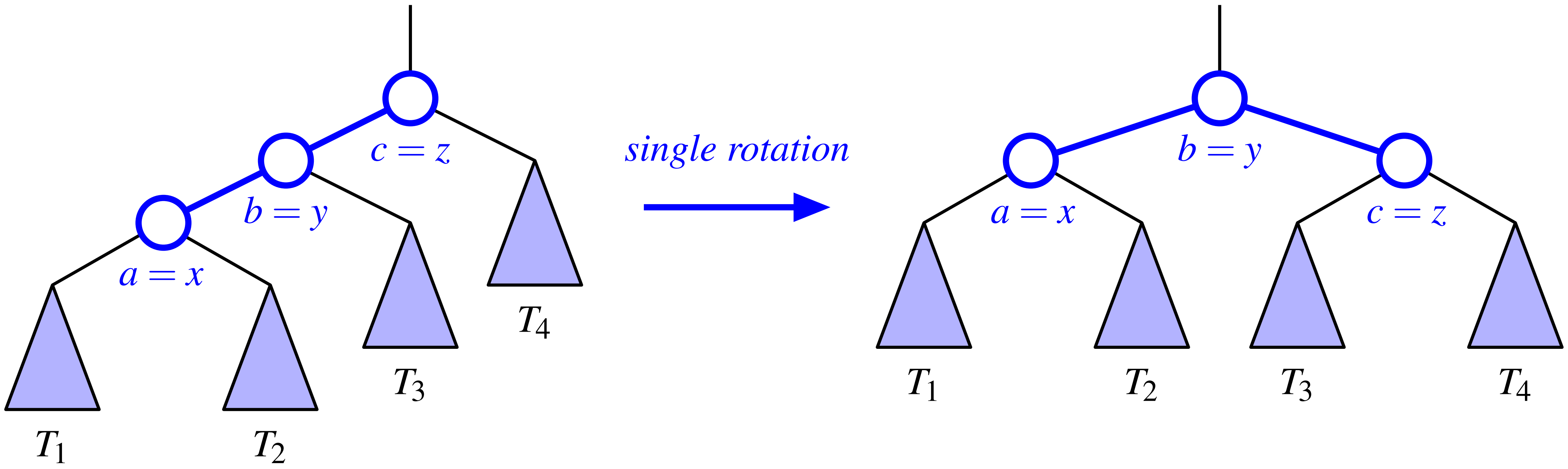
Single Rotation (around z)



Single Rotation (around z)



Single Rotation (around z)



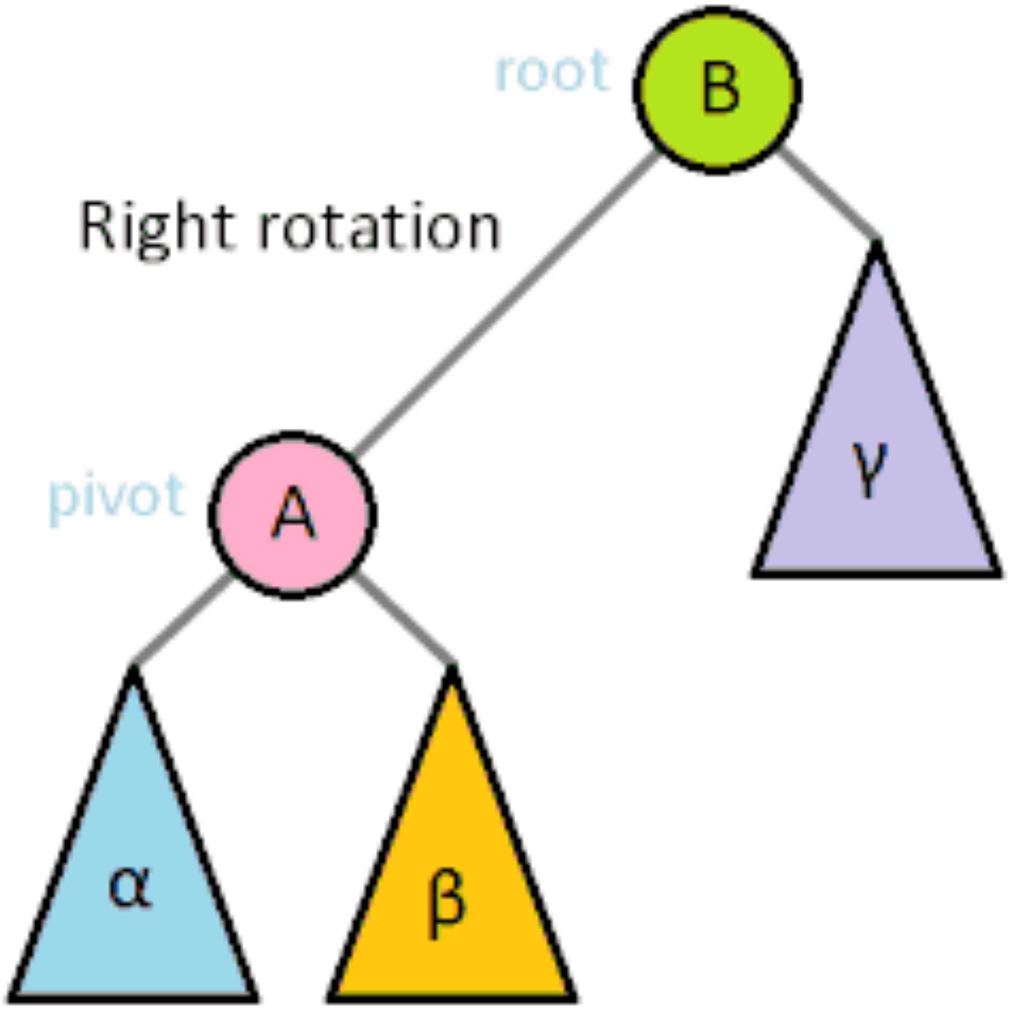
Rotations

Right rotation:

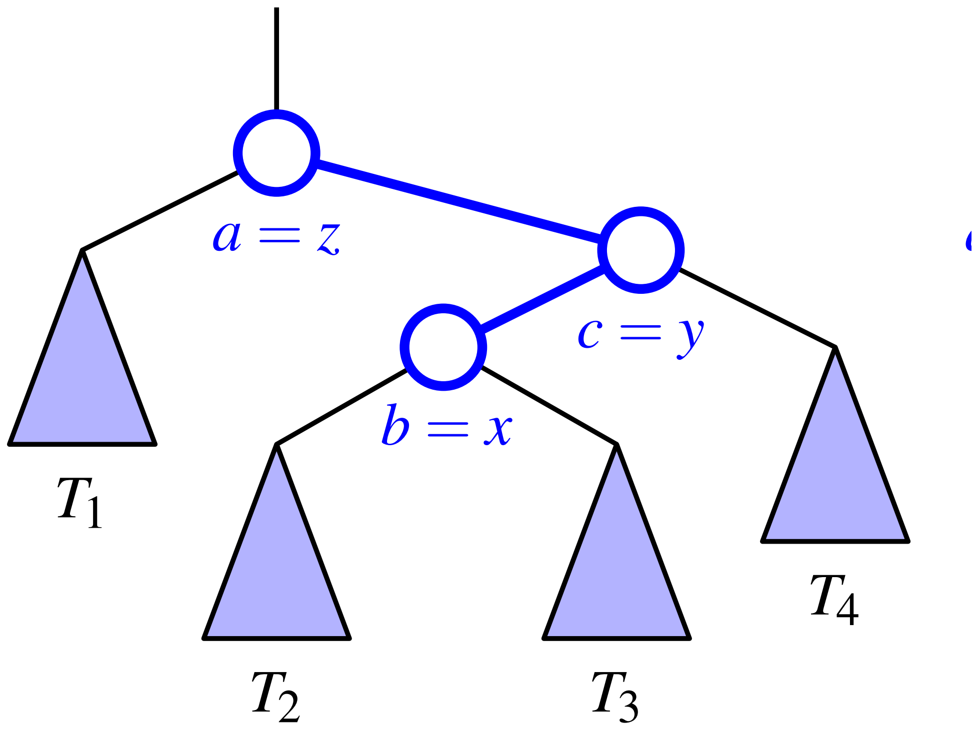
- Root node's left child becomes the new root
- Root node becomes the left child's right child

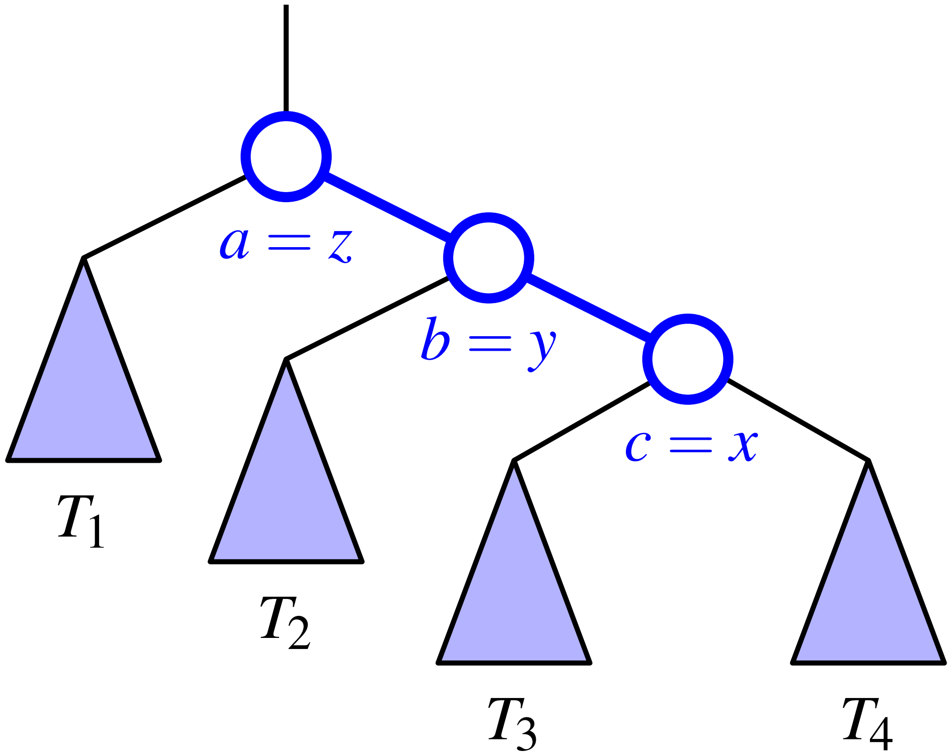
Left rotation:

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

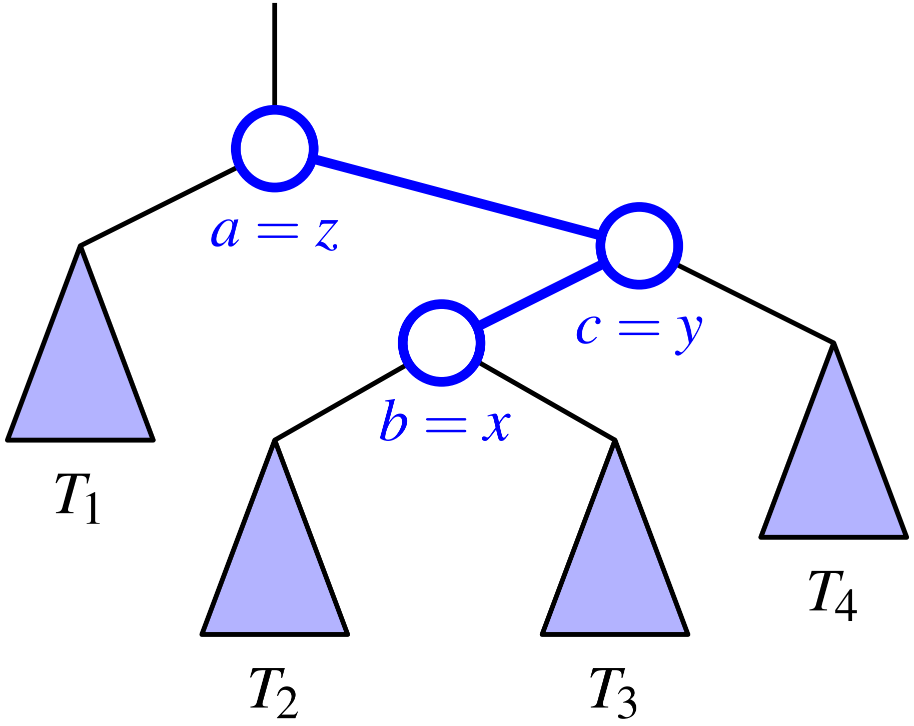


Rotation

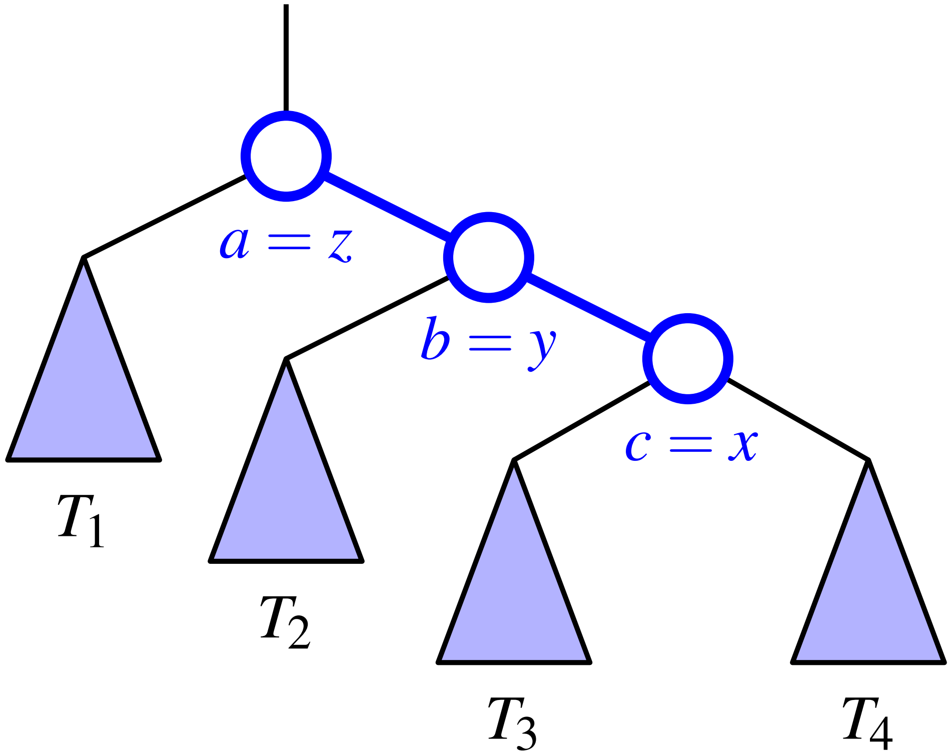




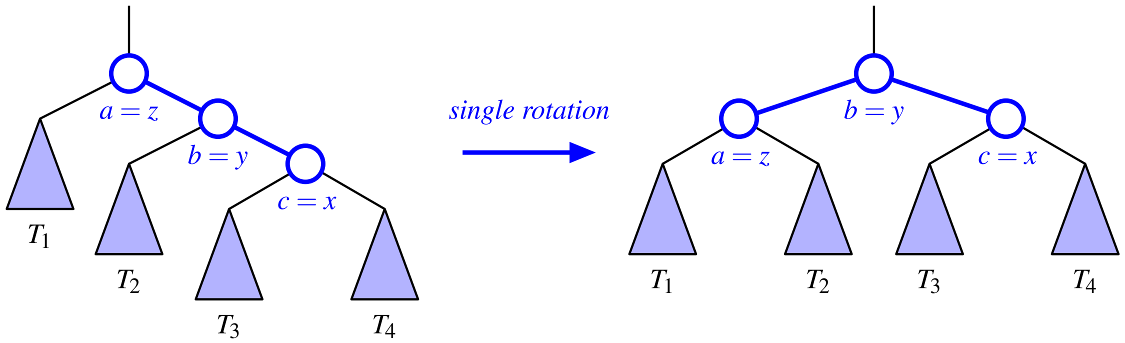
Rotation



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

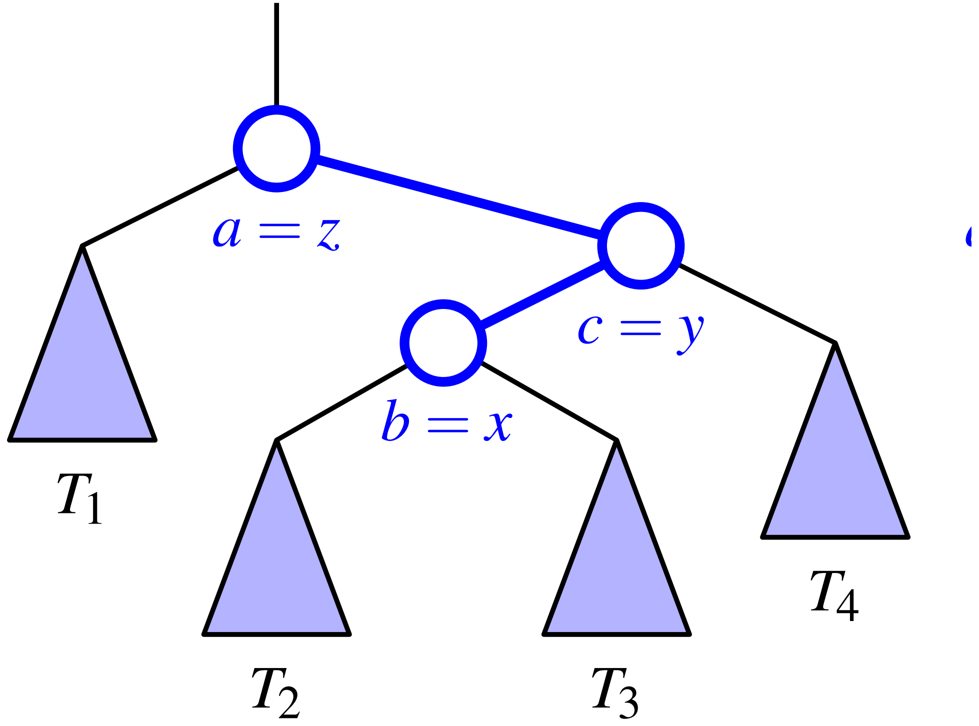


$a = z$
$b = y$
$c = x$
T_1
T_2
T_3
T_4

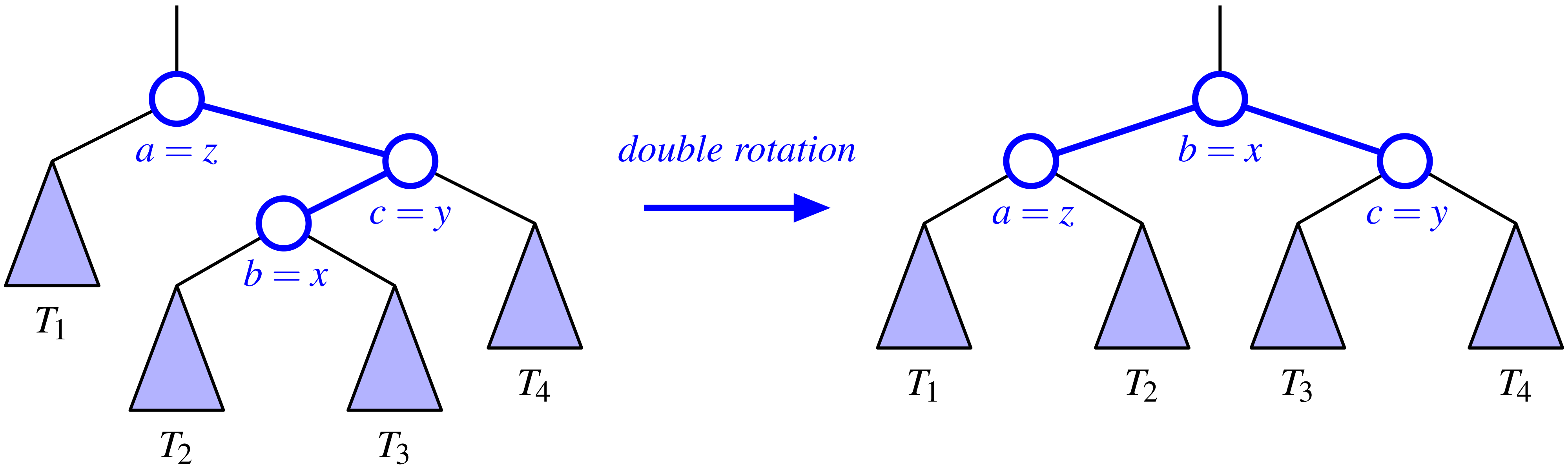


$a = z$
$b = y$
$c = x$
T_1
T_2
T_3
T_4
single rotation
$b = y$
$a = z$
$c = x$
T_1
T_2
T_3
T_4

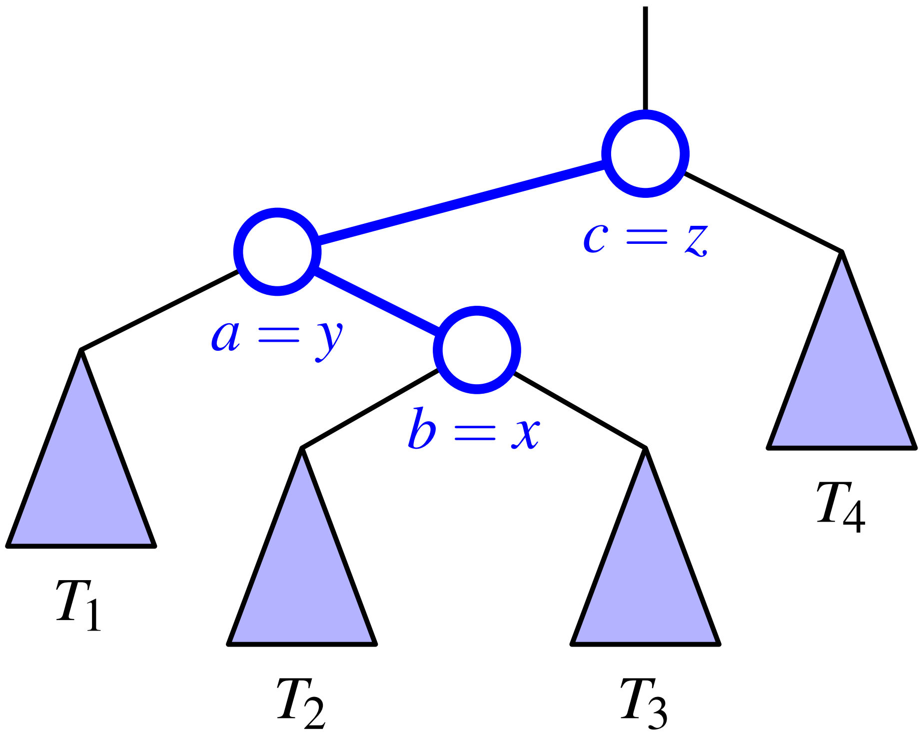
Double Rotation



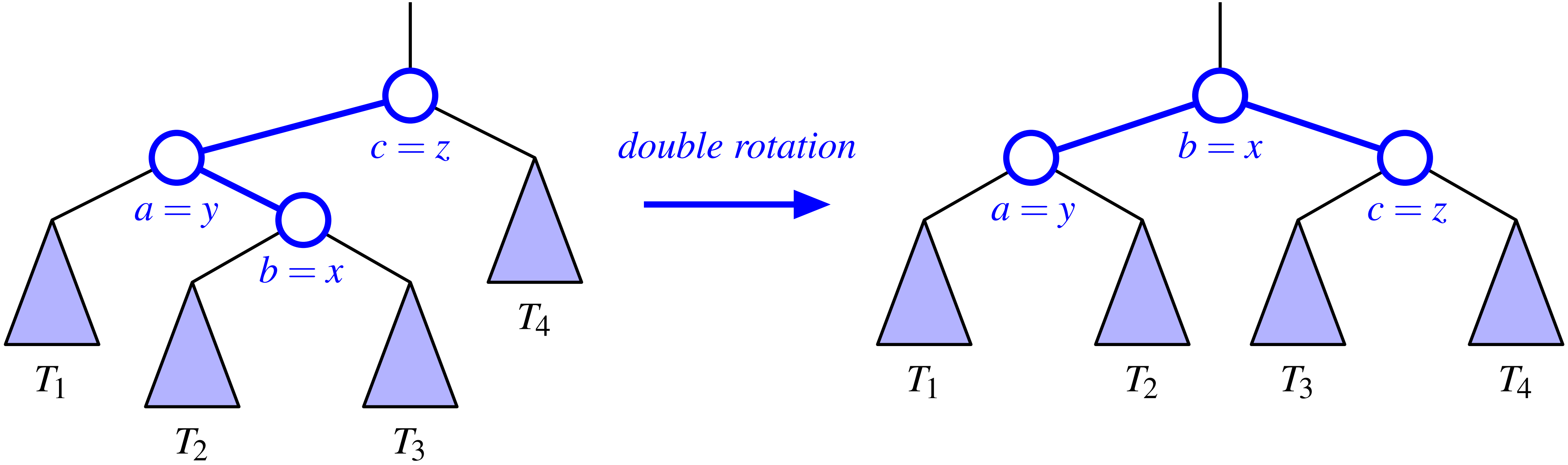
Double Rotation



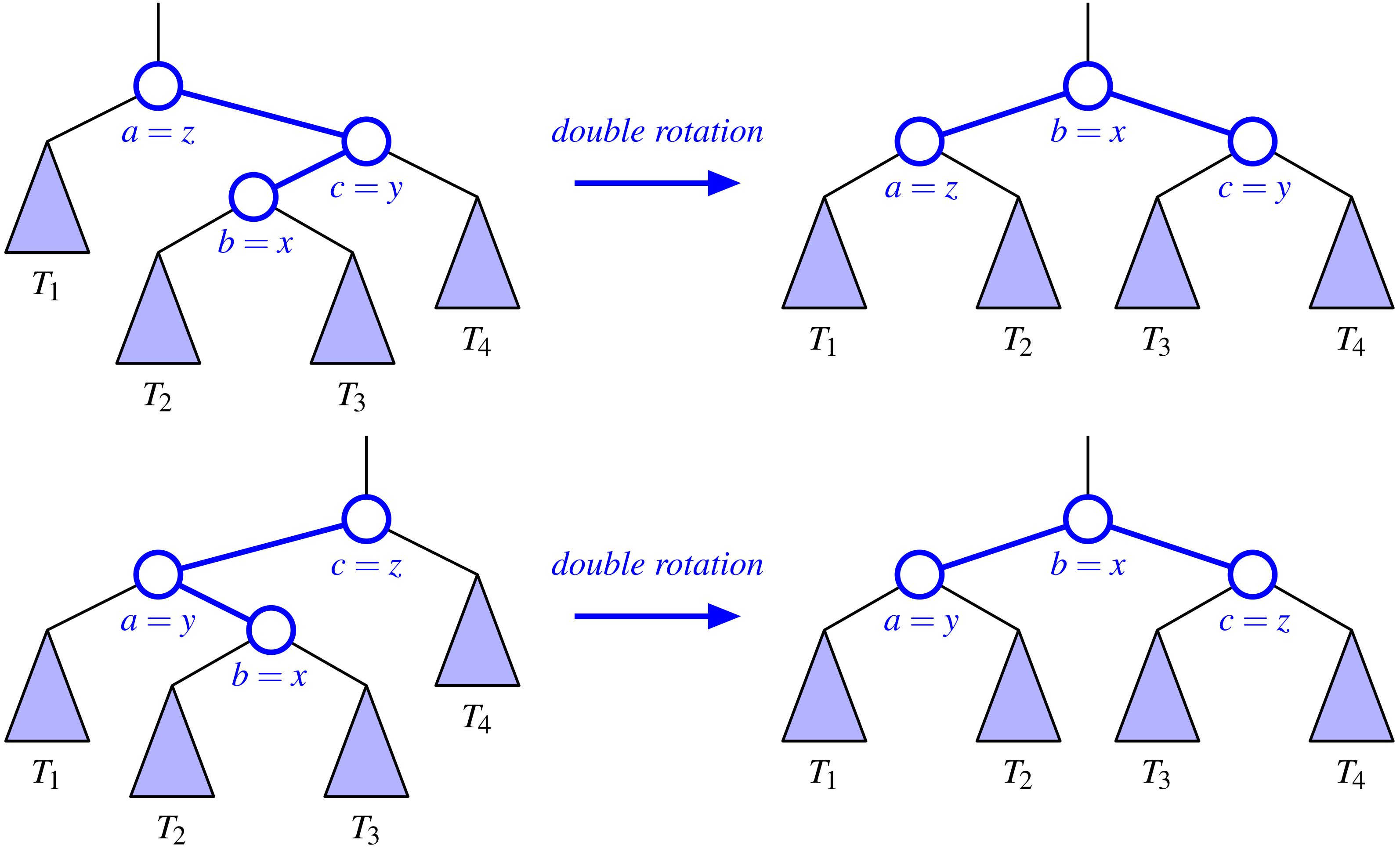
Double Rotation (around z)



Double Rotation (around z)

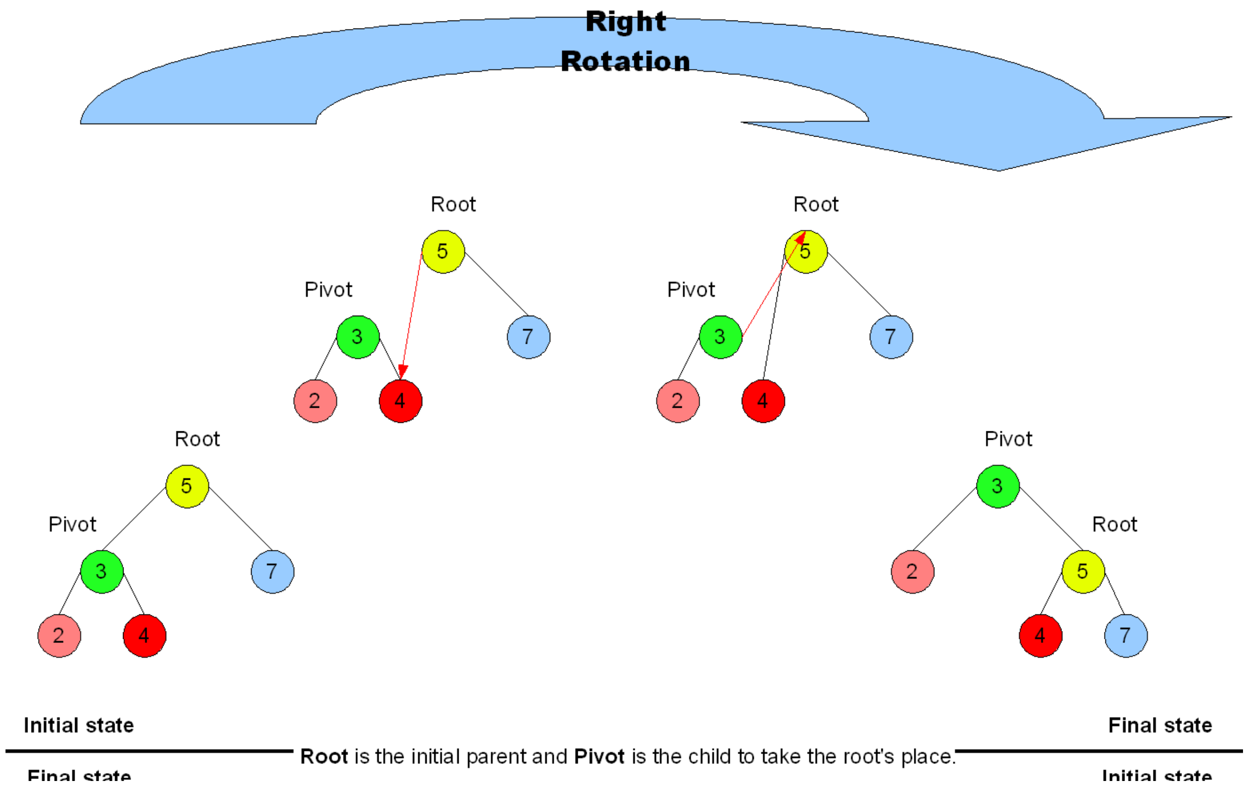


Double Rotation (around z)



Tree Rotations

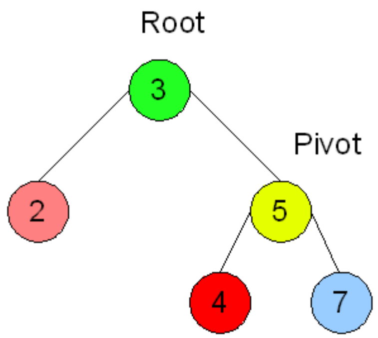
```
rotateRight(r):
  if (r.left==null) return
  p = r.left
  r.left = p.right
  p.right = r

  // 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
```

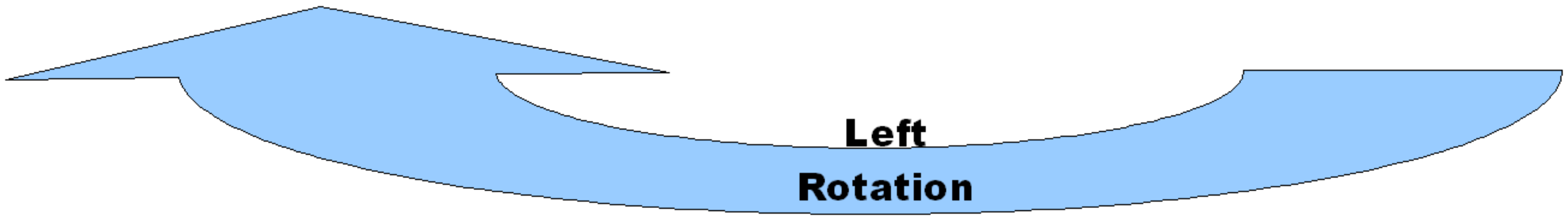


Initial state **Final state**

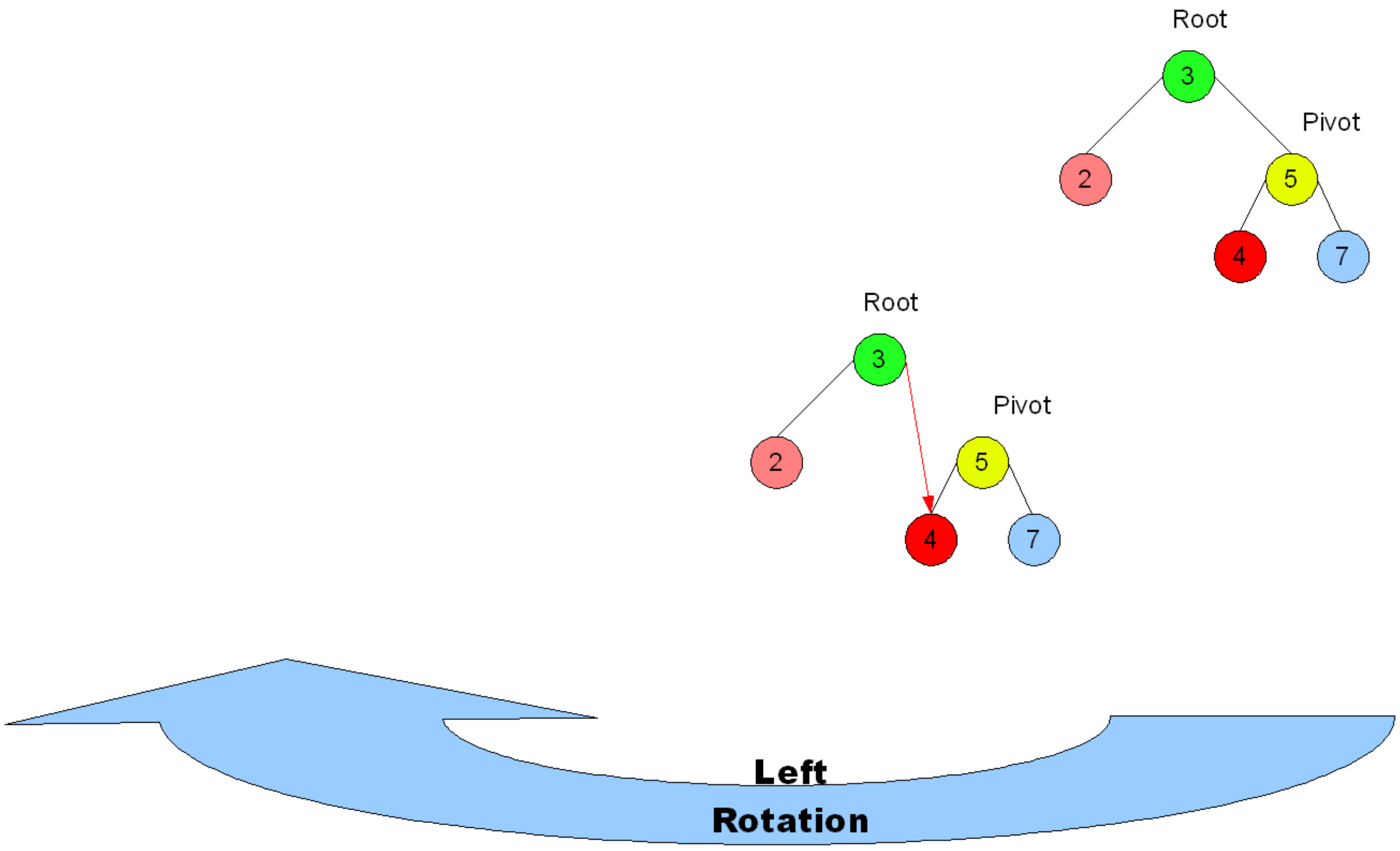
Root is the initial parent and **Pivot** is the child to take the root's place.

Final state **Initial state**





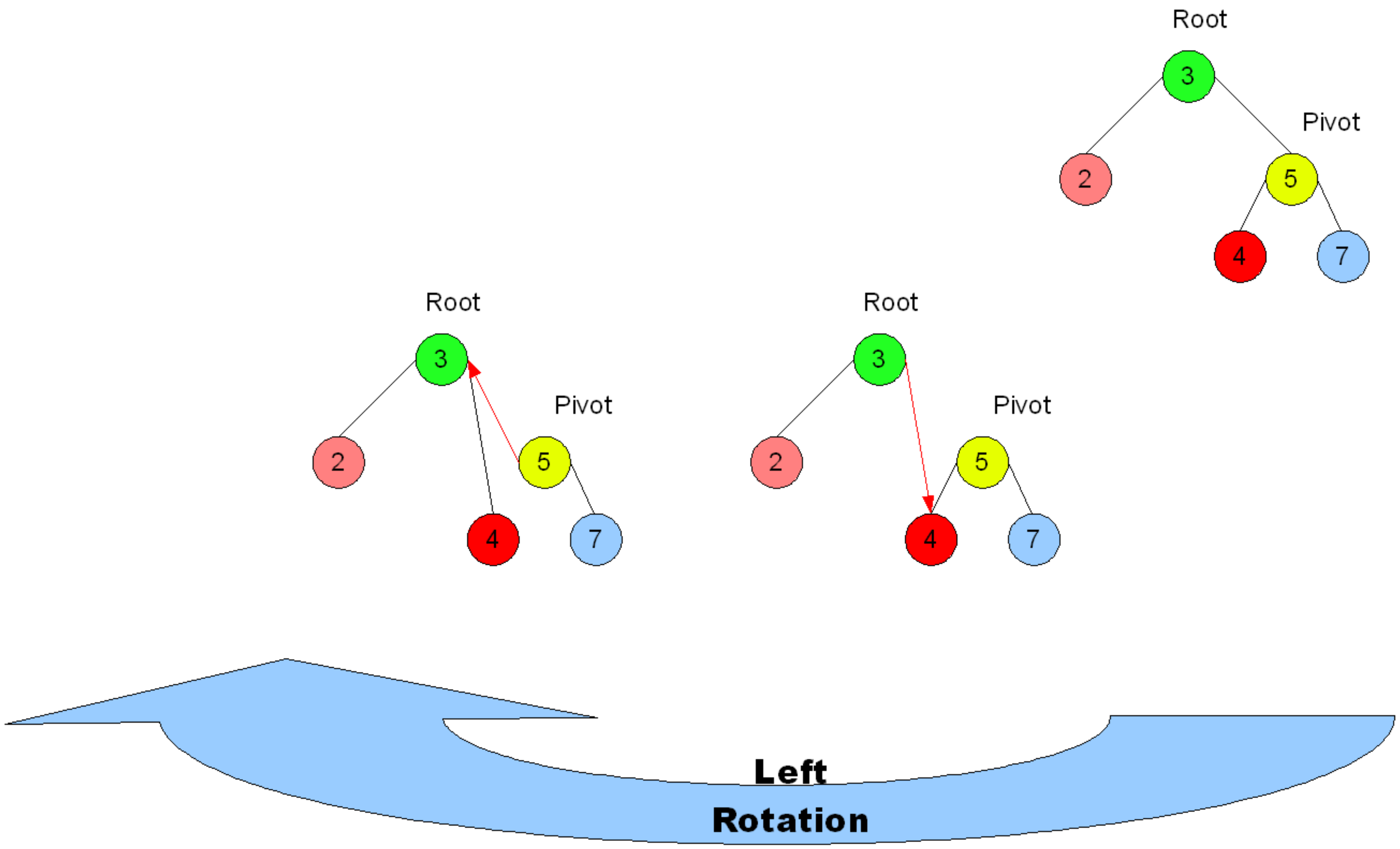
Initial state	**Root** is the initial parent and **Pivot** is the child to take the root's place.	Final state
Final state		Initial state

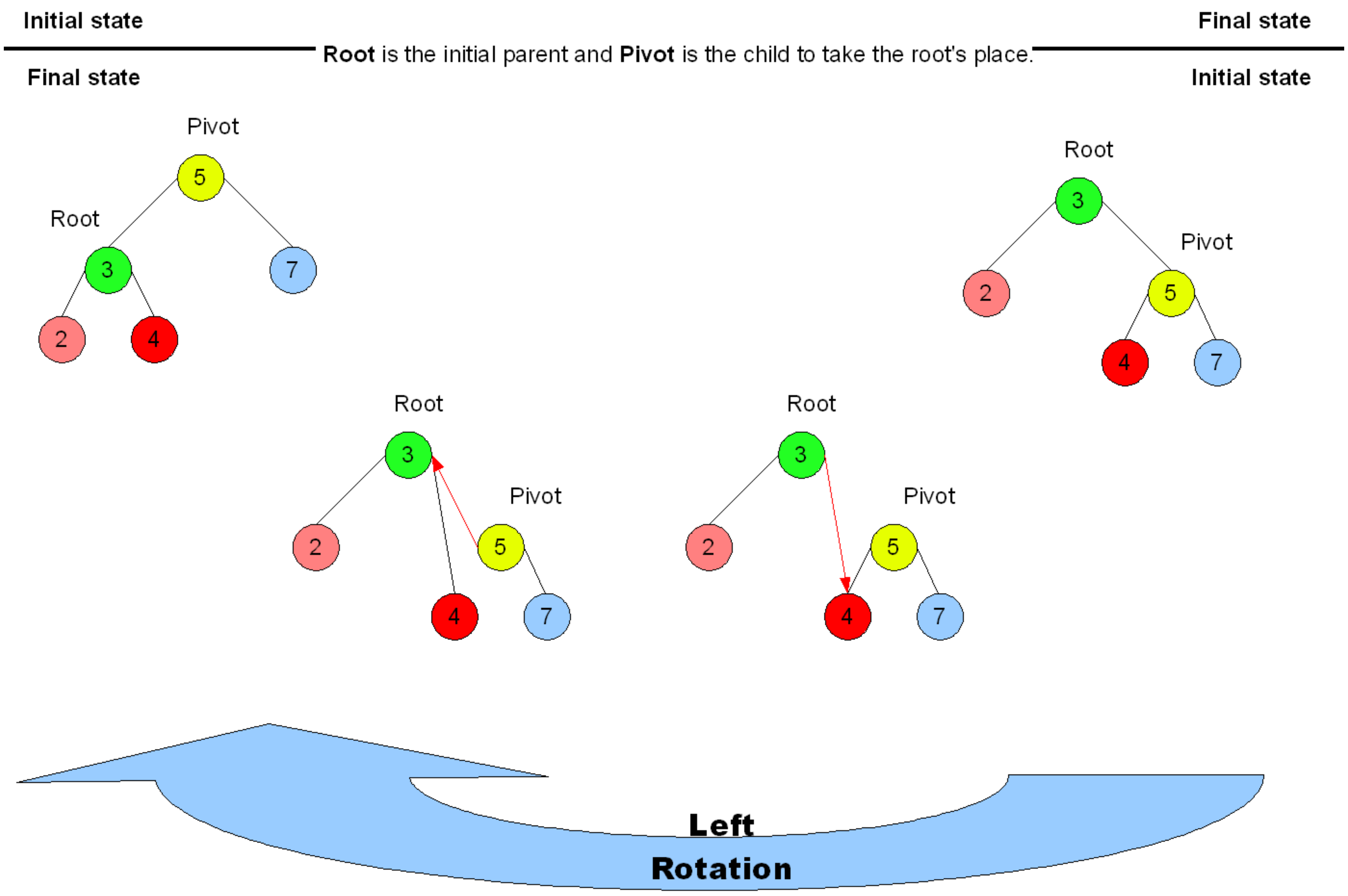


Initial state | Final state

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
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

Outline

Double Hashing Review (Homework 07)

Balanced Binary Trees

AVL Trees

Splay Trees

Red-Black Trees

AVL Tree

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

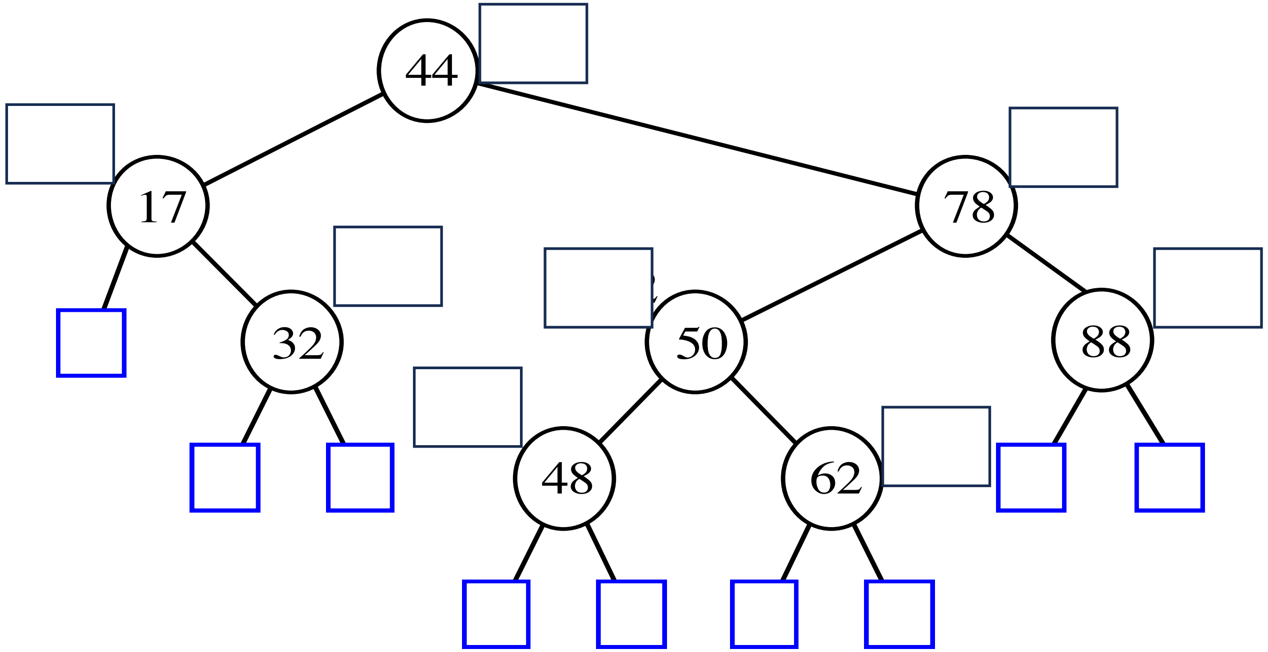
Height-balance property

- 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

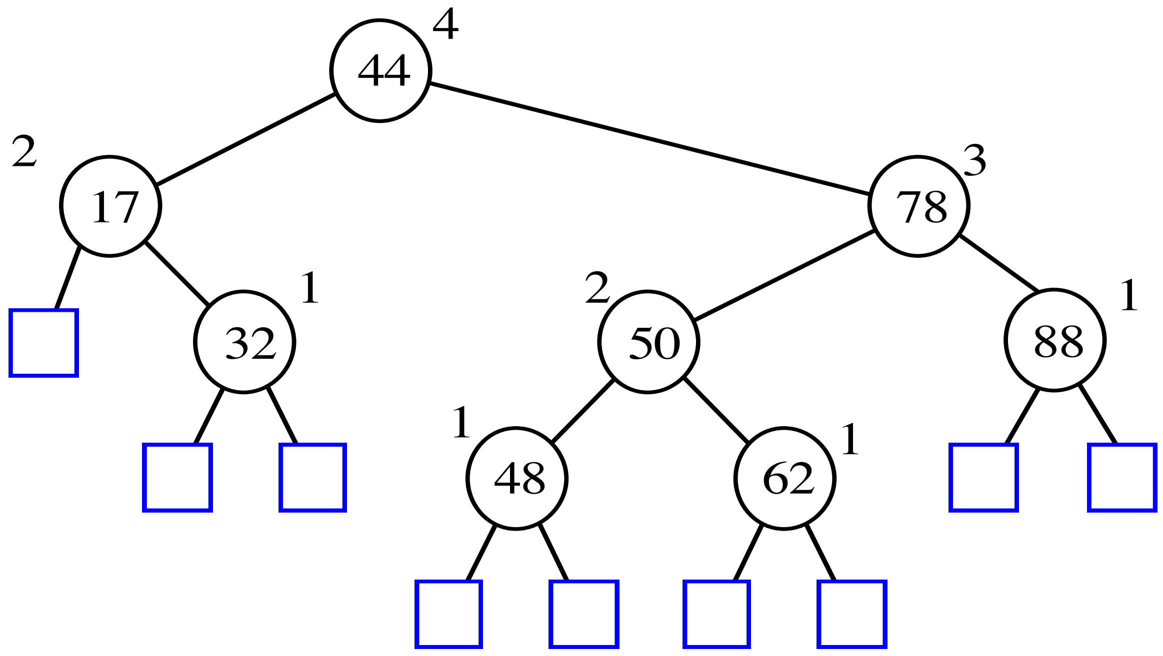
AVL Tree Example

- leaves are sentinels and have height 0



AVL Tree Example

- leaves are sentinels and have height 0



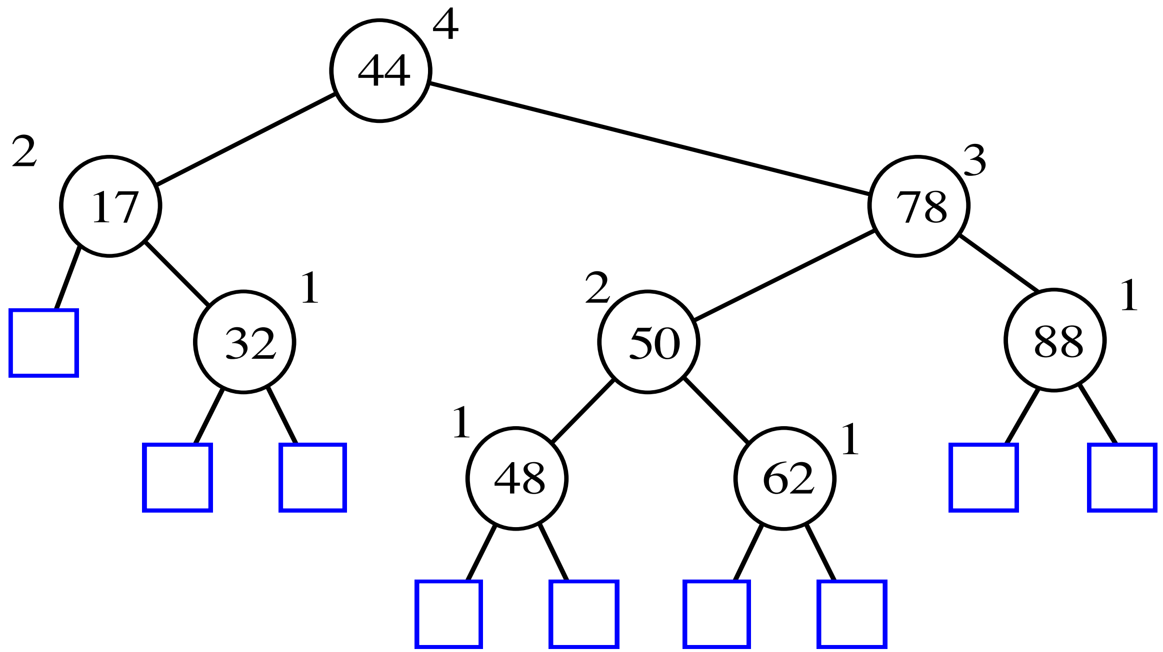
AVL height

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



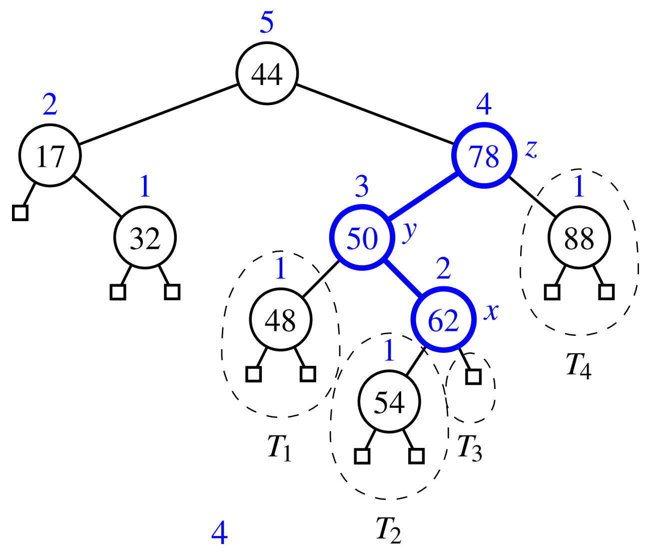
Insertion (54)

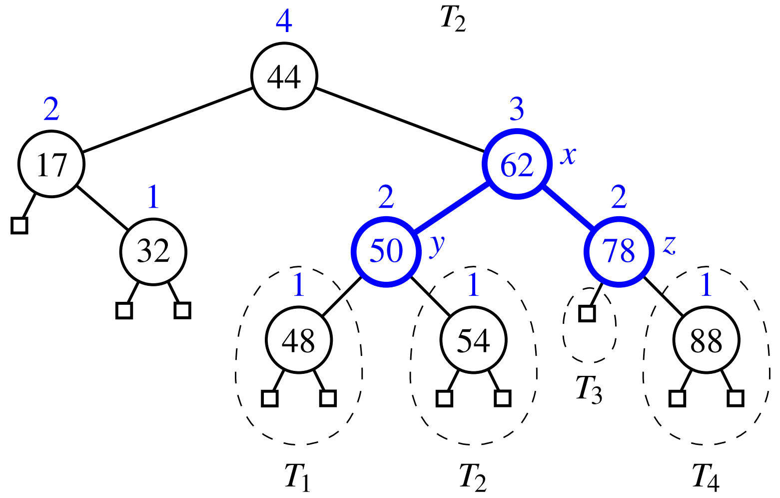
New node always has height 1

Parent may change height

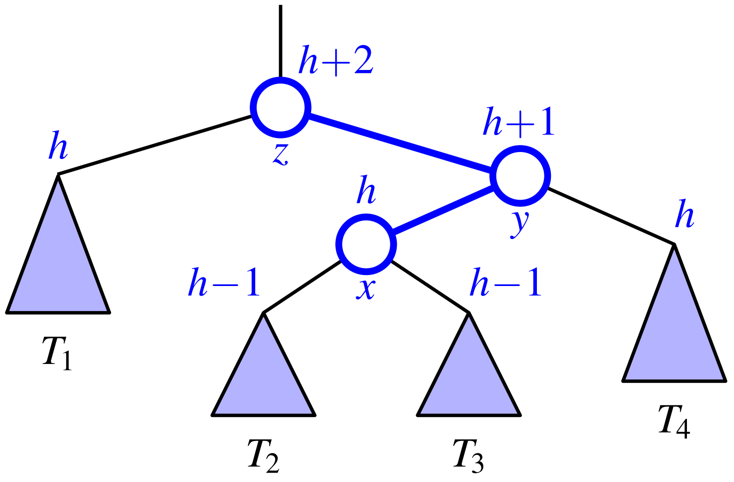
All ancestors may become unbalanced

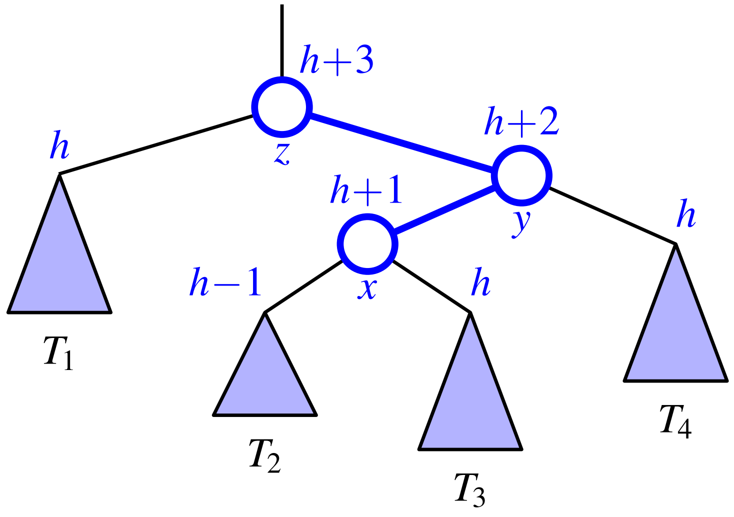
Perform rotations for unbalanced ancestors





$O(1)$ Rotation Restores Global Balance

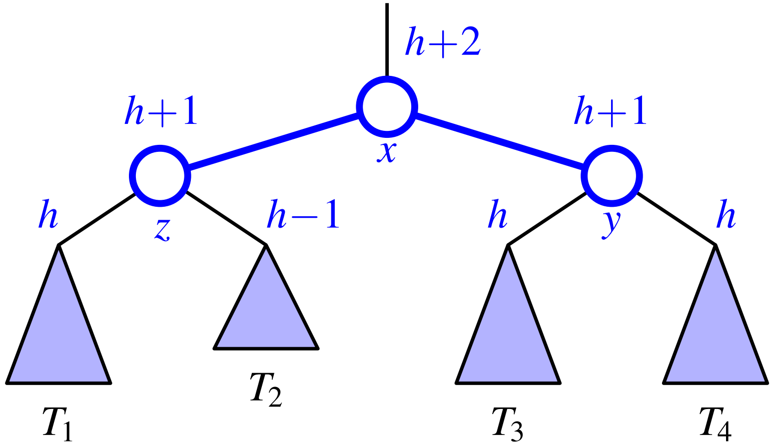




Insert into T_3

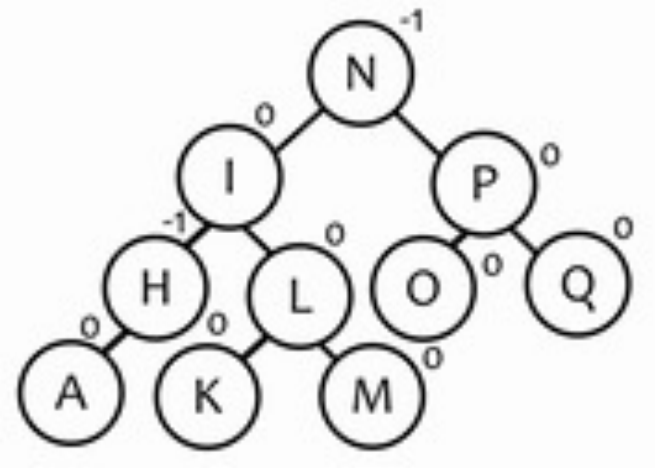
After rebalance:

- 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

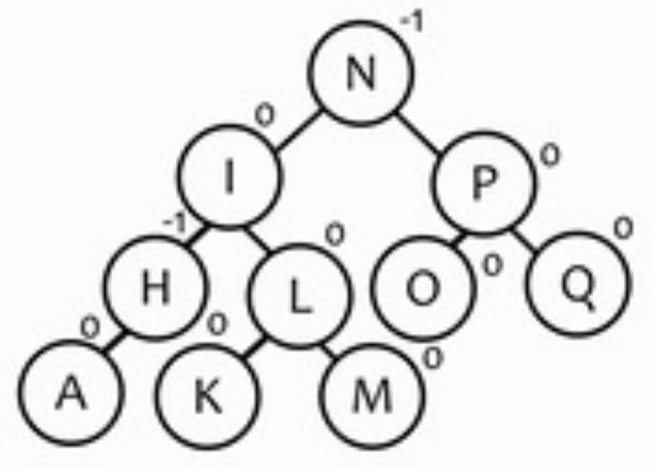


- AVL balance marked on nodes
- balance(n) = height of right subtree – height of left subtree
- AVL balance property: |balance(n)| ≤ 1

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
```

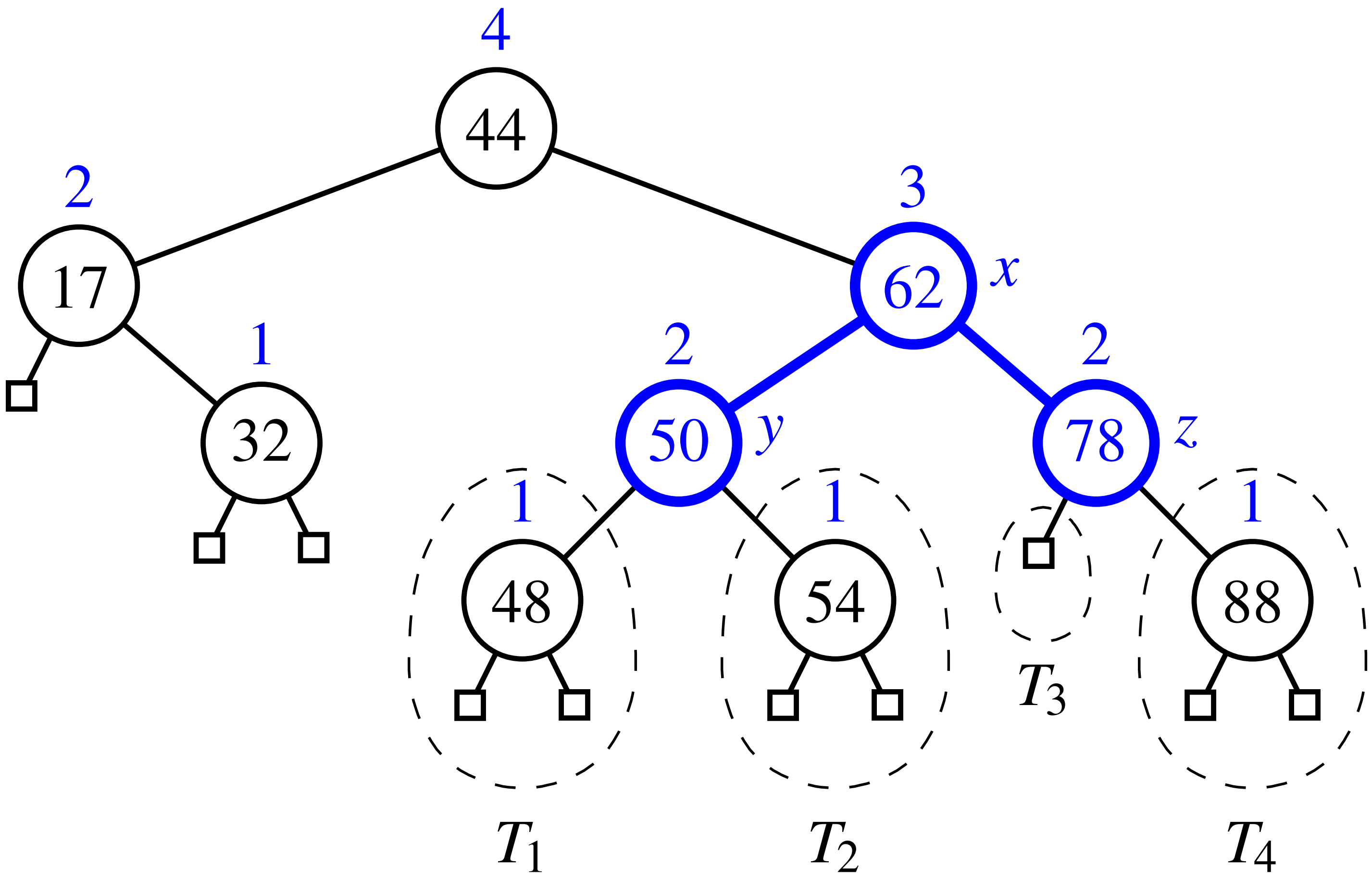
```
updateHeight(n):
  lh = n.left.height
  rh = n.right.height
  height = 1+max(lh, rh)
```

```
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

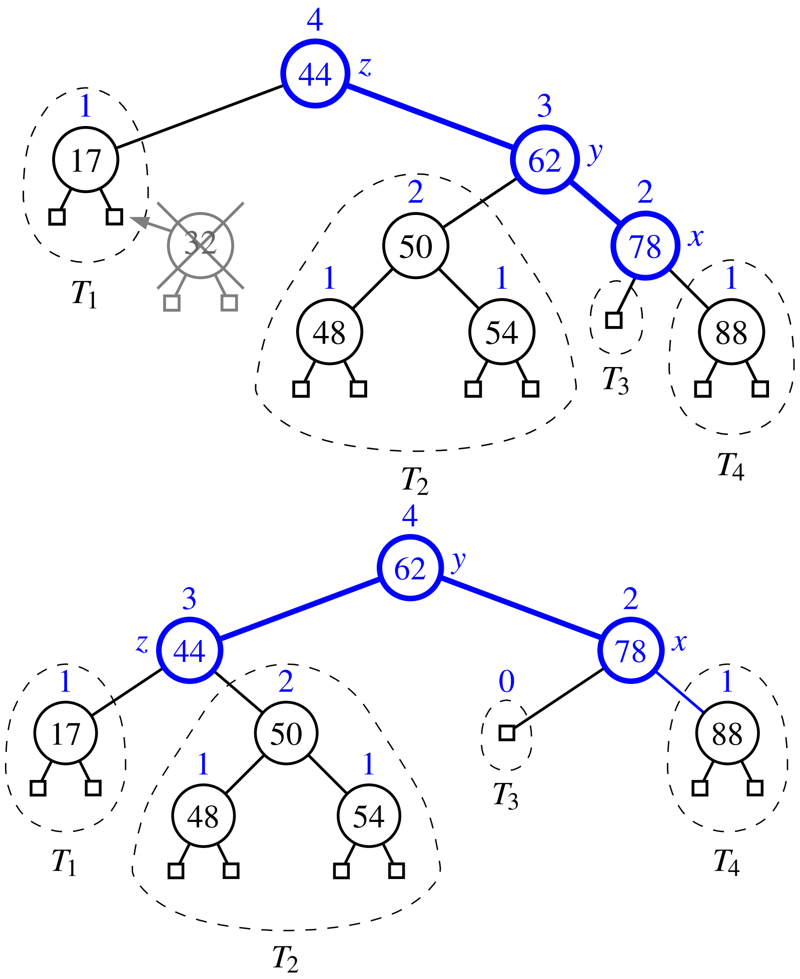
Deletion structurally removes a node with 0 or 1 child

- 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`

Outline

Double Hashing Review (Homework 07)

Balanced Binary Trees

AVL Trees

Splay Trees

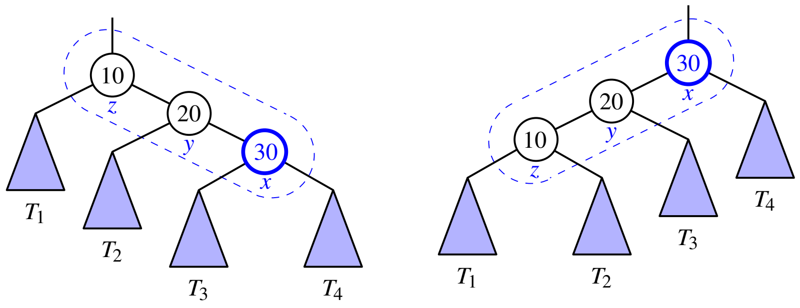
Red-Black Trees

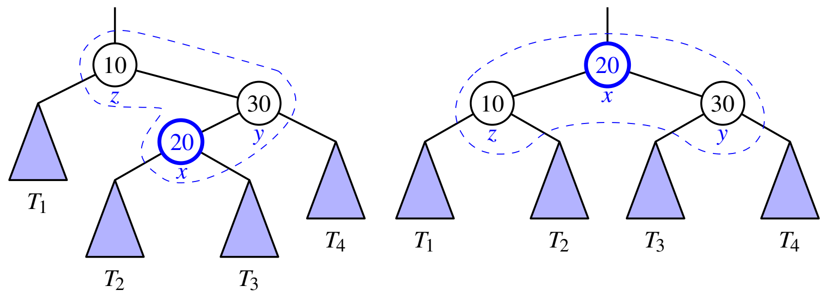
Splay Tree

- A binary search tree that doesn't enforce a $O(logn)$ bound on the height
- Efficiency is achieved due to a move-to-root operation, called splaying
- Performed at the leaf reached during every insert, delete and search
- Causes the more frequently accessed elements to be near the top

Splaying

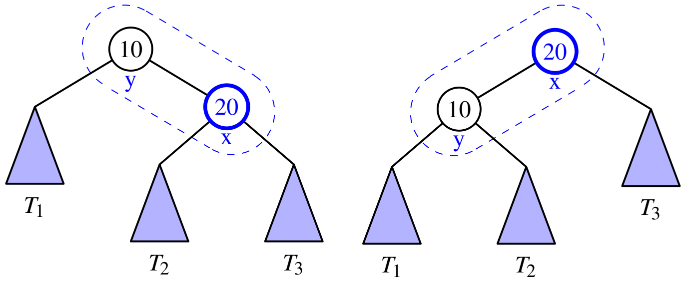
- Swapping a BST node x up depends on the relative position of x, its parent y and its grandparent z
- zig-zig (zag-zag): x and y are both right/left children
- zig-zag (zag-zig): one right one left





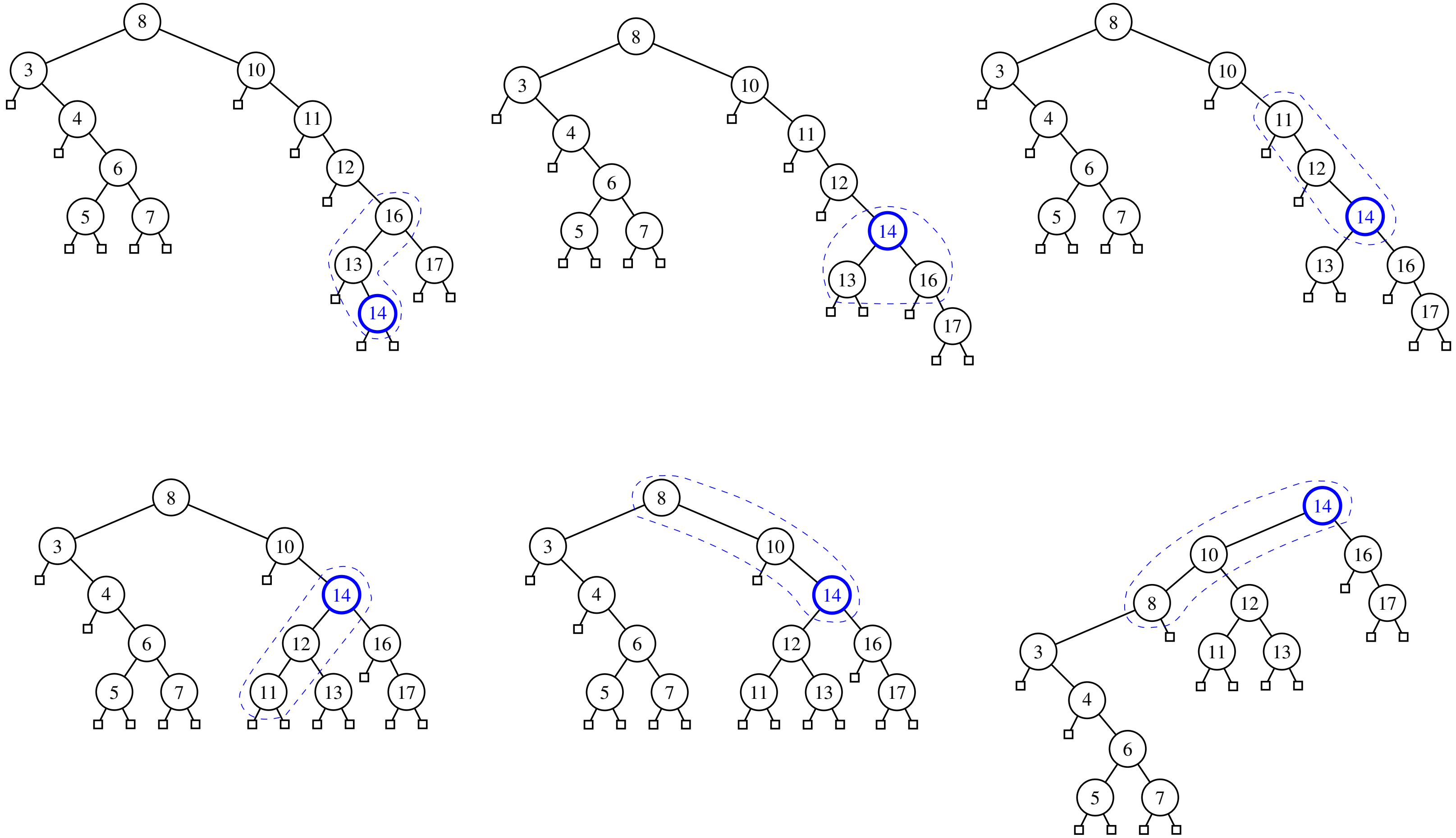
Splaying

- zig (zag): y has no parent



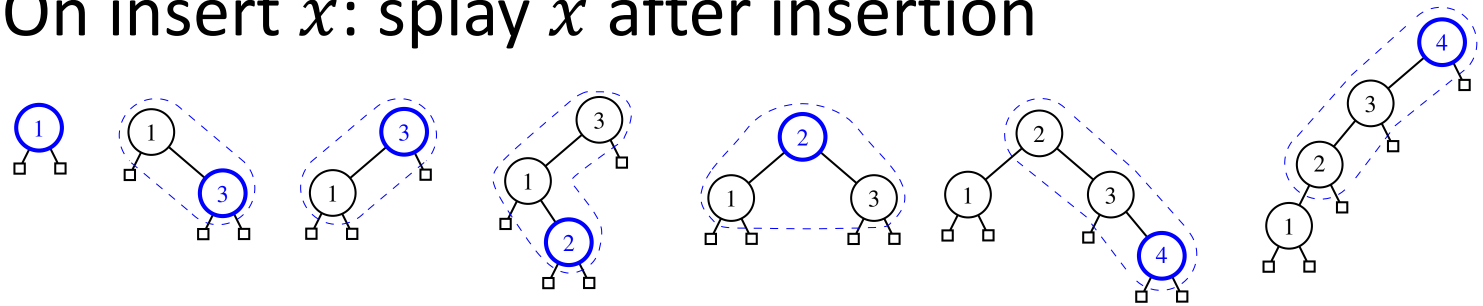
- Splaying will continue these rotations until x becomes root

Example



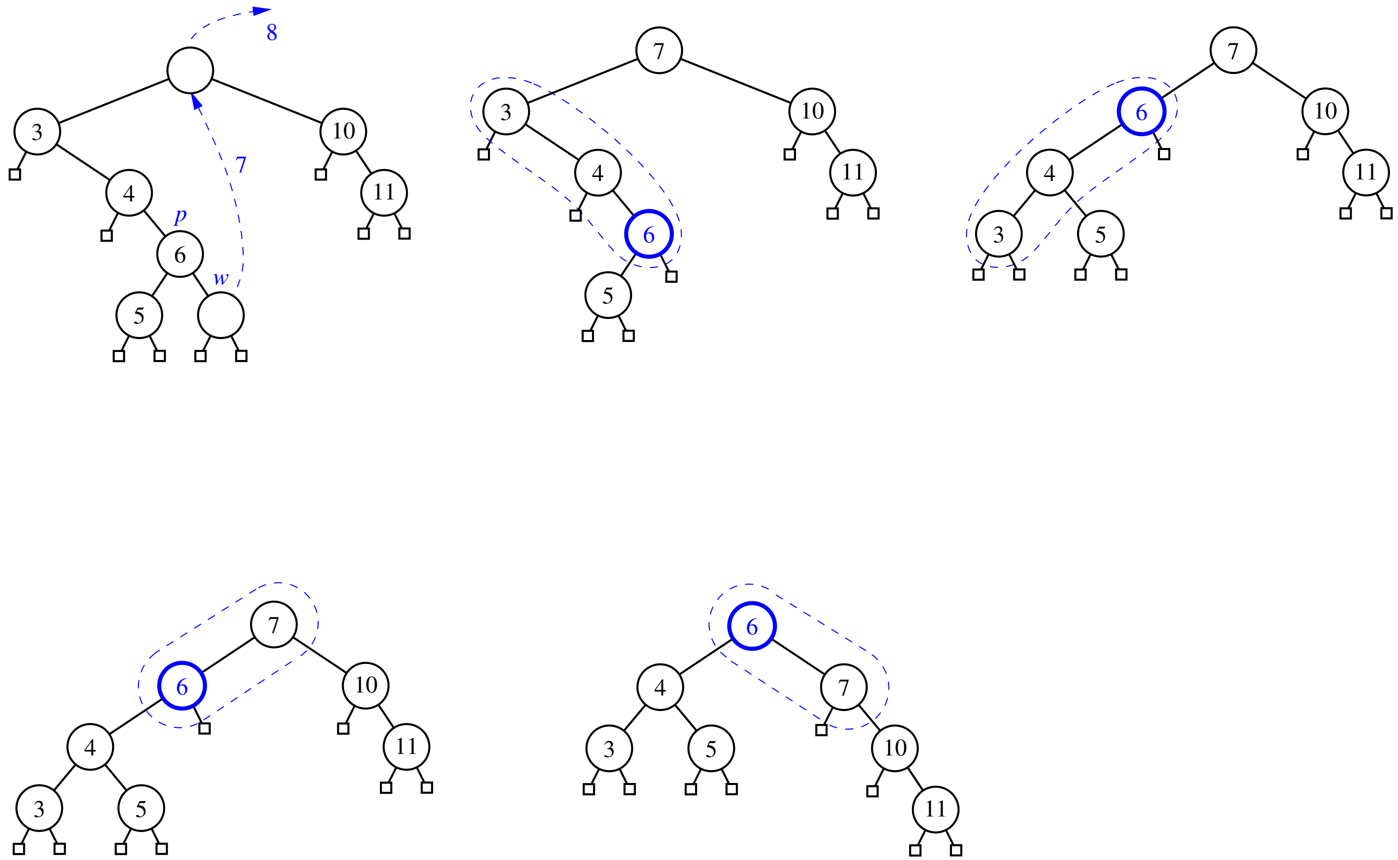
When/what to Splay

- On search for x: if x is found, splay x else splay x's parent
- On insert x: splay x after insertion

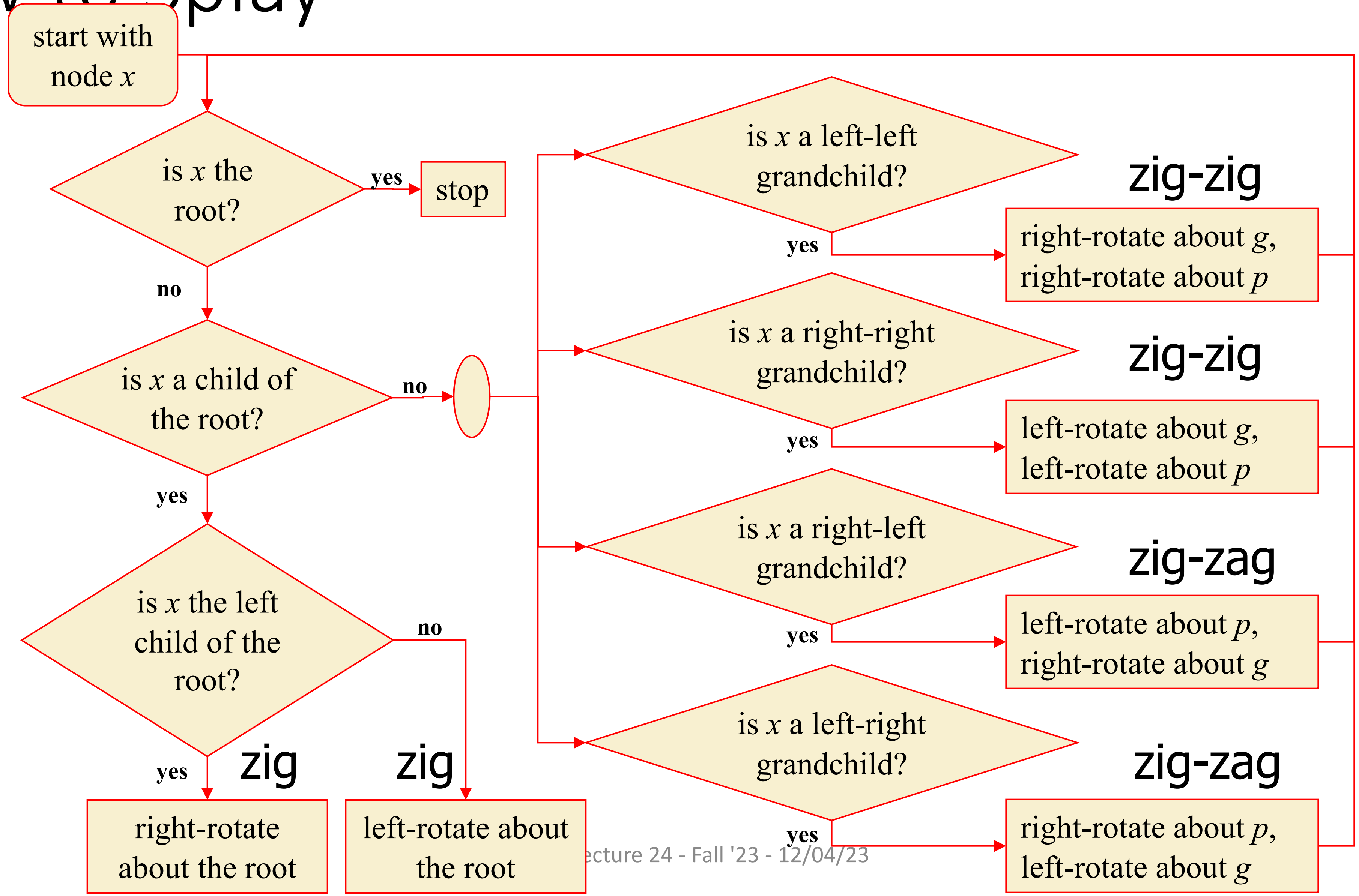


- On delete x: splay parent of removed node
    - x is removed
    - in-order successor/predecessor removed

Deletion



How to Splay


start with node x
is x the root?
yes
stop
no
is x a child of the root?
no
yes
is x the left child of the root?
yes
no
zig
right-rotate about the root
zig
left-rotate about the root
is x a left-left grandchild?
yes
zig-zig
right-rotate about g, right-rotate about p
is x a right-right grandchild?
yes
zig-zig
left-rotate about g, left-rotate about p
is x a right-left grandchild?
yes
zig-zag
left-rotate about p, right-rotate about g
is x a left-right grandchild?
yes
zig-zag
right-rotate about p, left-rotate about g

Analysis of Splaying

- Splay trees do rotations after every operation (even search)
- Runtime of each search/insert/delete is proportional to the time for splaying
- Each zig-zig, zig-zag or zig is $O(1)$
- Splaying a node at height h is $O(h)$
- Worst case height of a splay tree is $O(n)$

Amortized Performance

- A splay tree performs well in amortization – in a sequence of mixed searches, insertions and deletions
- Splay tree performs better for many sequences of non-random operations
- Amortized cost for any splay operation is $O(logn)$
- Must faster search than $O(logn)$ on frequently requested items

Comparison of Maps

	Search	Insert	Delete	Notes
Hash Table	$O(1)$ expected	$O(1)$ expected	$O(1)$ expected	○ not ordered ○ simple to implement
Skip List	$O(logn)$ high prob.	$O(logn)$ highprob.	$O(logn)$ high prob.	○ randomized insertion ○ simple to implement
AVL	$O(logn)$ worst-case	$O(logn)$ worst-case	$O(logn)$ worst-case	○ complex to implement
Splay	$O(logn)$ amortized	$O(logn)$ amortized	$O(logn)$ amortized	○ complex to implement ○ faster than $O(logn)$ on favorites

AVL Rotations

- AVL insert – $O(logn)$
  - Find the lowest out-of-balance ancestor – also known as the critical node, rotate critical node to balance. Loop ends after single rotation
  - $O(logn)$ search up the tree to find critical node + $O(1)$ rotations
- AVL delete - $O(logn)$
  - $O(logn)$ rotations on delete

Outline

Double Hashing Review (Homework 07)

Balanced Binary Trees

AVL Trees

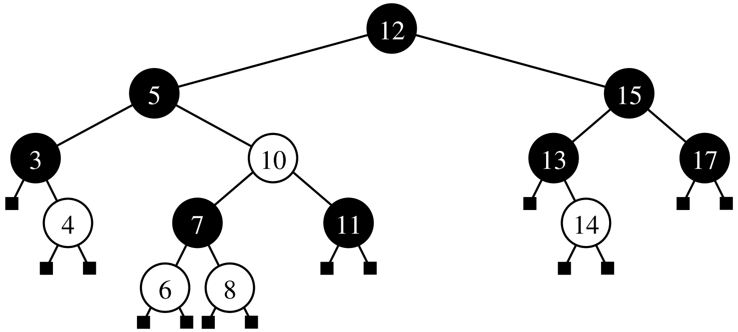
Splay Trees

Red-Black Trees

Red-Black Tree

- AVL has $O(1)$ rotations on insert and $O(logn)$ rotations on delete
- Splay has $O(logn)$ rotations (amortized) on all operations
- Red-black tree
  - insert and delete: $O(1)$ rotations + $O(logn)$ recoloring up the tree
  - $O(logn)$ search

Red-Black Properties



- All null nodes are black
- Children of red nodes are black
- All null nodes have same black depth - number of ancestors that are black
- Root is black (made black)

AVL versus RB

- AVL is a subset of RB
- AVL height is more rigidly balanced
- RB height property: longest path from the root to a leaf is no more than twice as long as shortest
- AVL is faster on searches
- RB is faster on deletion