UML Diagrams

Data Structures and Algorithms

Warm Up

Given the following UML diagrams of a Hash Map implementation of a Dictionary write the pseudo code to implement the set(key, value) function

HashMap<K, V>

state

Data[]

- Pair<K, V>[]
- LinkedList<E>[]

size

behavior

put() pair into array based on hash
- Resize when appropriate

get() value from array index based given key's hash

set() update value in pair for given key's hash to array index

remove() take data out of array

LinkedList<E>

state

ListNode<K, V> front

behavior

Add() add a new node that stores Key and Value to list

get() return value from node with given key

set() changes value in node with given key

remove() deletes node with given key from list

Contains() is the given key stored in list

iterator() returns an iterator to move over list

ListNode<K, V>

state

K key

V value

ListNode<K, V> next

behavior

Construct a new Node

Implementing Set

```
void set(k key, v value) {
   bucketAddress = get hash for key % table size
   bucketList = data[bucketAddress]
   loop(bucketList)
      if (this node's key is what I am trying to add)
         replace this node with new pair
         stop work
}
```

HashMap<K, V>

state
- LinkedList<E>[]
- size

behavior
- void put(key, value)
- value get(key)
- void set(key, value)
- void remove(key)

LinkedList<E>

state
- ListNode<K, V> front

behavior
- void add(key, value)
- value get(key)
- void set(key, value)
- void remove(key)
- boolean contains(key)
- iterator<E> iterator()

ListNode<K, V>

state
- K key
- V value
- ListNode<K, V> next

behavior
- Construct a new Node

Implementing a Dictionary

Dictionary ADT

state

Set of Key, Value pairs
- Keys must be unique!
- No required order

Count of data pairs

behavior

Add pair to collection

Get value for given key

Change value for given key

Remove data pair from collection

public interface Dictionary {

state

unspecified

behavior

```
void put(key, value)

value get(key)

void set(key, value)

void remove(key)
```

HashMap<K, V>

state

Data[]
- Pair<K, V>[]
- LinkedList<E>[]

size

behavior

put() pair into array based on hash
- Resize when appropriate

get() value from array index based given key's hash

set() update value in pair for given key's hash to array index

remove() take data out of array

TreeMap<K, V>

state

overallRoot<K,V>

behavior

put() add node for new pair in correct location
- Balance when appropriate

get() value based on node location in tree

set() update value in pair for given key

remove() delete given node
- replace with appropriate existing node

Implementing Tree Map

TreeMap<K, V>

state

`overallRoot<K,V>`

behavior

`put()` add node for new pair in correct location
- Balance when appropriate

`get()` value based on node location in tree

`set()` update value in pair for given key

`remove()` delete given node
- replace with appropriate existing node

ListNode<K, V>

state

`K key`

`V value`

`ListNode<K, V> left`

`ListNode<K, V> right`

`int height`

behavior

Construct a new Node

Implementing Tree Map

```
v get(k key) {
   start at top of tree
}
ListNode<K, V> getHelper(key, Node) {
   if(node is null)
      data isn't in collection
   if data at current node > what I'm looking for
      go left
   if data at current node < what I'm looking for
      go right
   else
      found it!
}
```

TreeMap<K, V>

state

```
overallRoot<K,V>
```

behavior

```
void put(key, value)
value get(key)
void set(key, value)
void remove(key)
```

ListNode<K, V>

state

```
K key
V value
ListNode<K, V> left
ListNode<K, V> right
int height
```

behavior

```
Construct a new Node
```

Implementing Tree Map

```
void put(key, value) {
    overallroot = putHelper(key, value, overallroot)
}

Node putHelper(K key, V value, Node node, int height) {
   if(node is null)
      return new node
   if data at current node > what I'm looking for
      go left
   if data at current node < what I'm looking for
      go right
   else
      return replacement node
   balance the tree if needed
}
```

TreeMap<K, V>

state

`overallRoot<K,V>`

behavior

`void put(key, value)`
`value get(key)`
`void set(key, value)`
`void remove(key)`

ListNode<K, V>

state

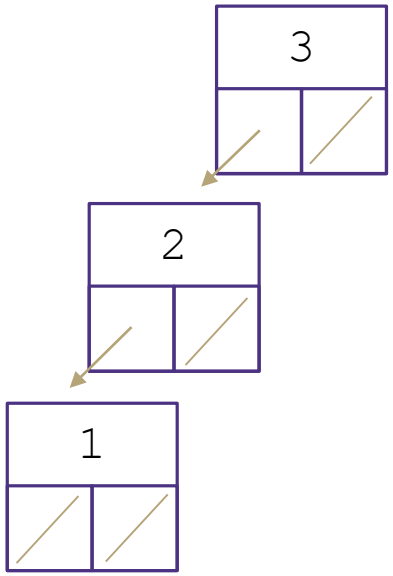
`K key`
`V value`
`ListNode<K, V> left`
`ListNode<K, V> right`
`int height`

behavior

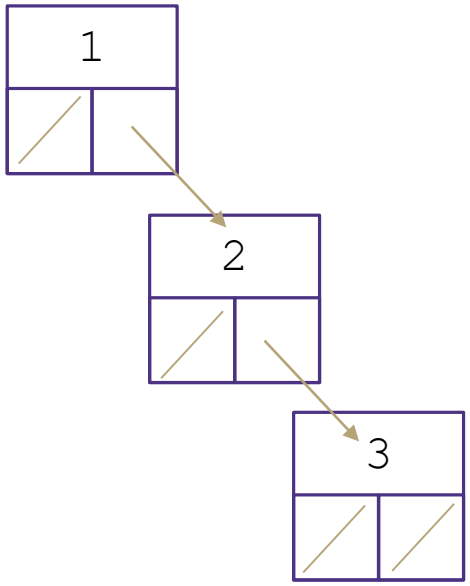
`Construct a new Node`

Two AVL Cases

Line Case
Solve with 1 rotation

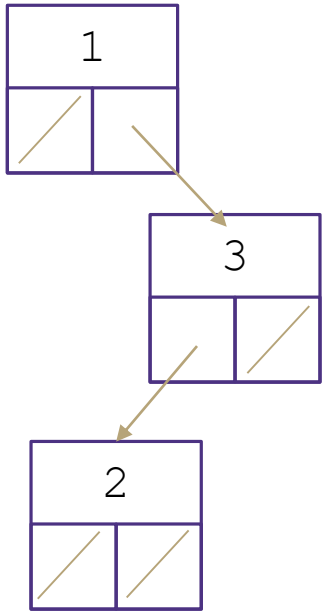


Rotate Right
Parent's left becomes child's right
Child's right becomes its parent

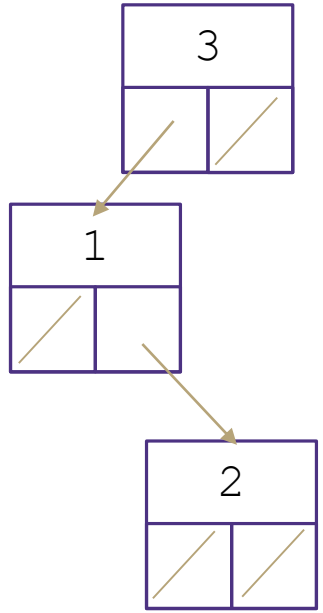


Rotate Left
Parent's right becomes child's left
Child's left becomes its parent

Kink Case
Solve with 2 rotations

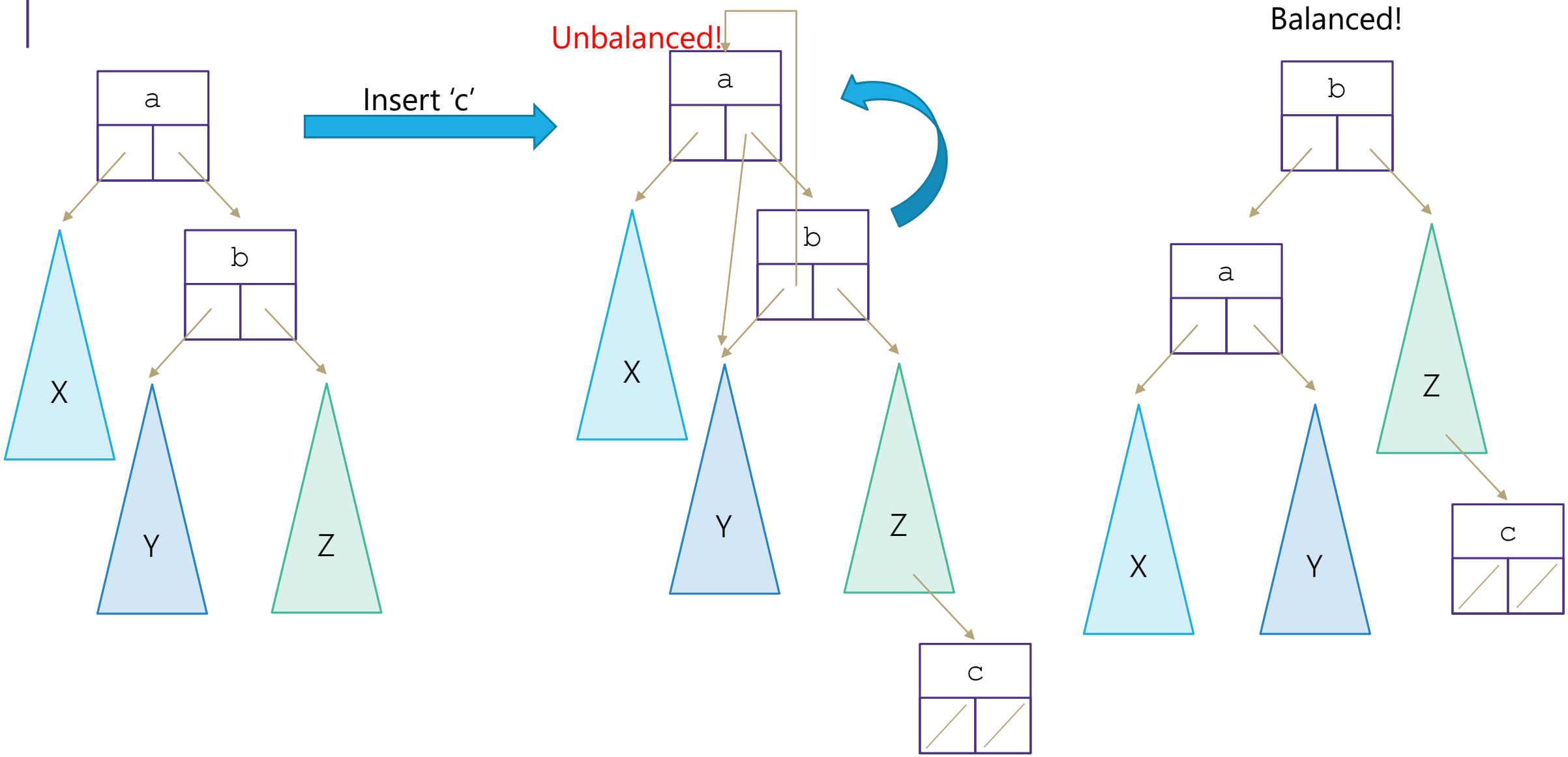


Rotate subtree left
Rotate root tree right

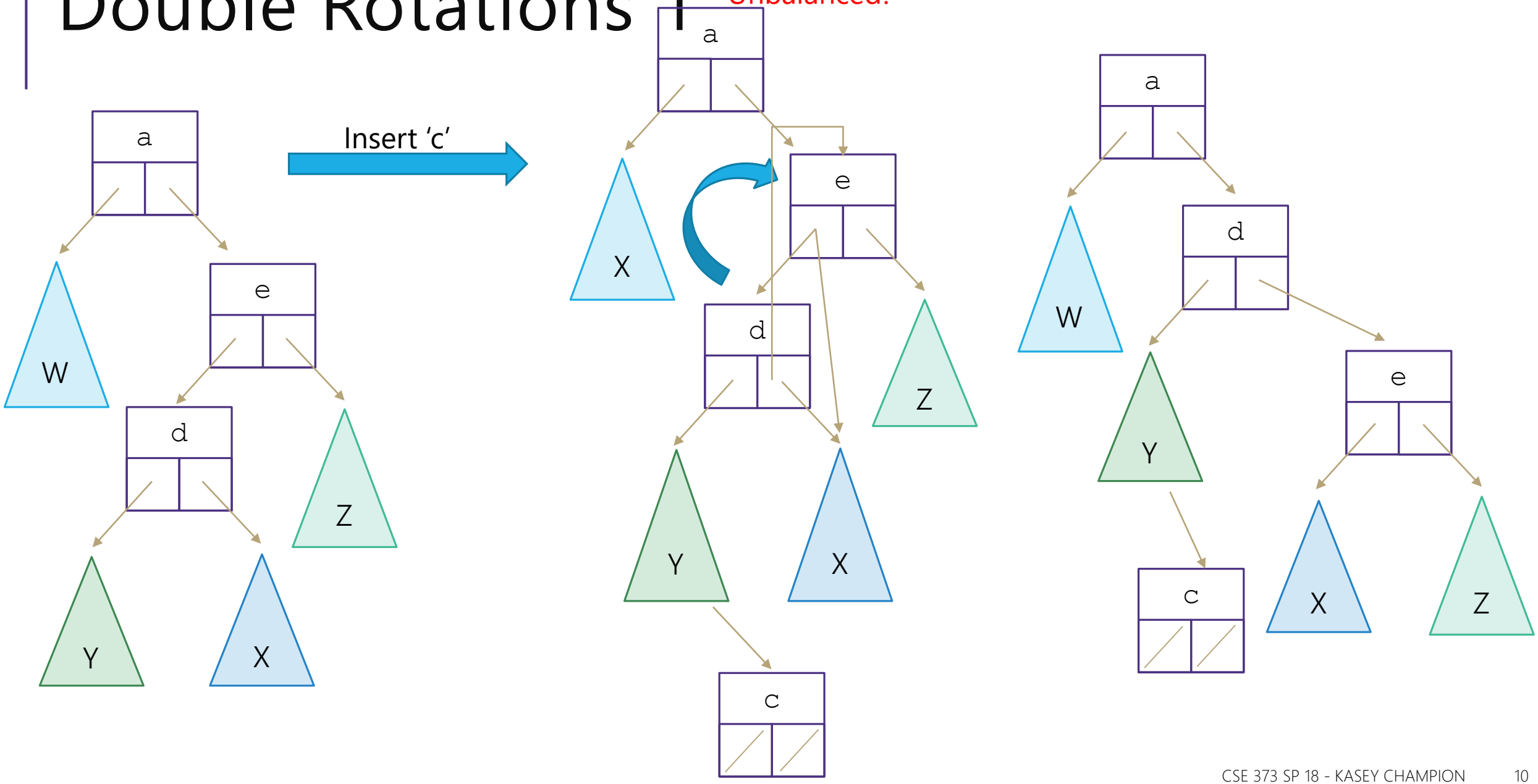


Rotate subtree right
Rotate root tree left

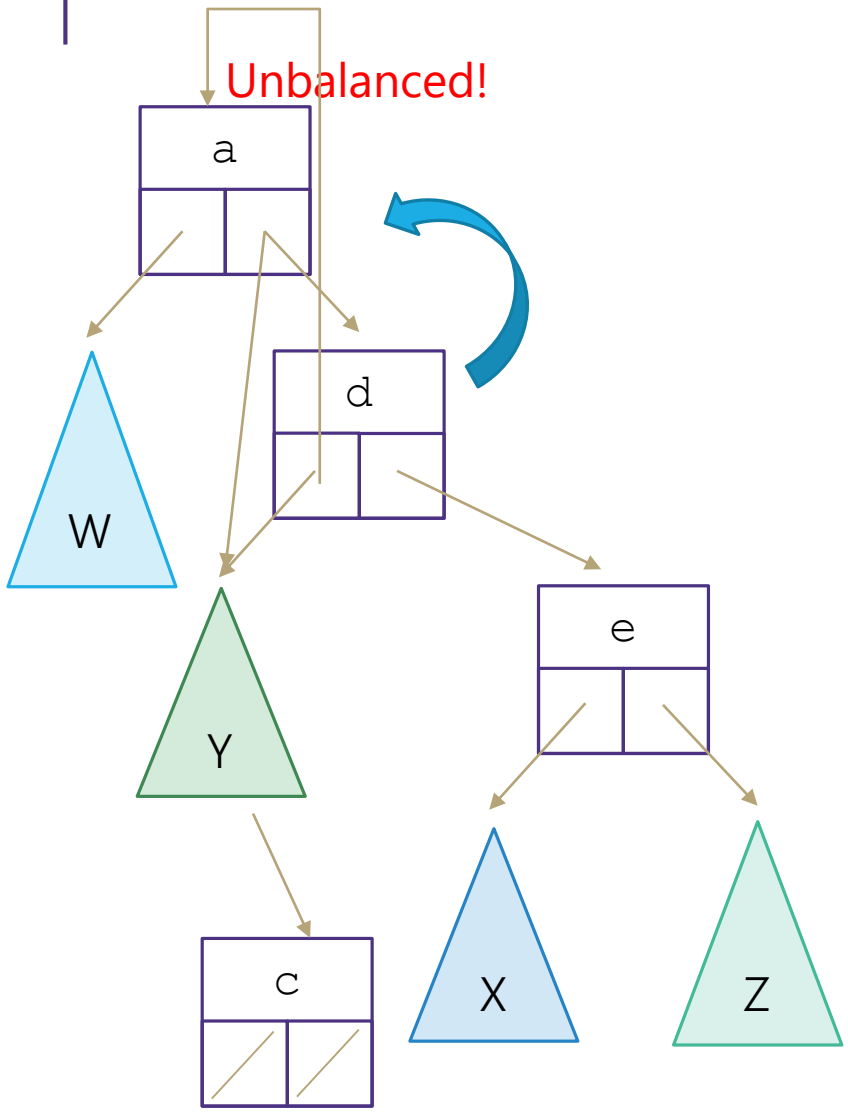
Rotations!

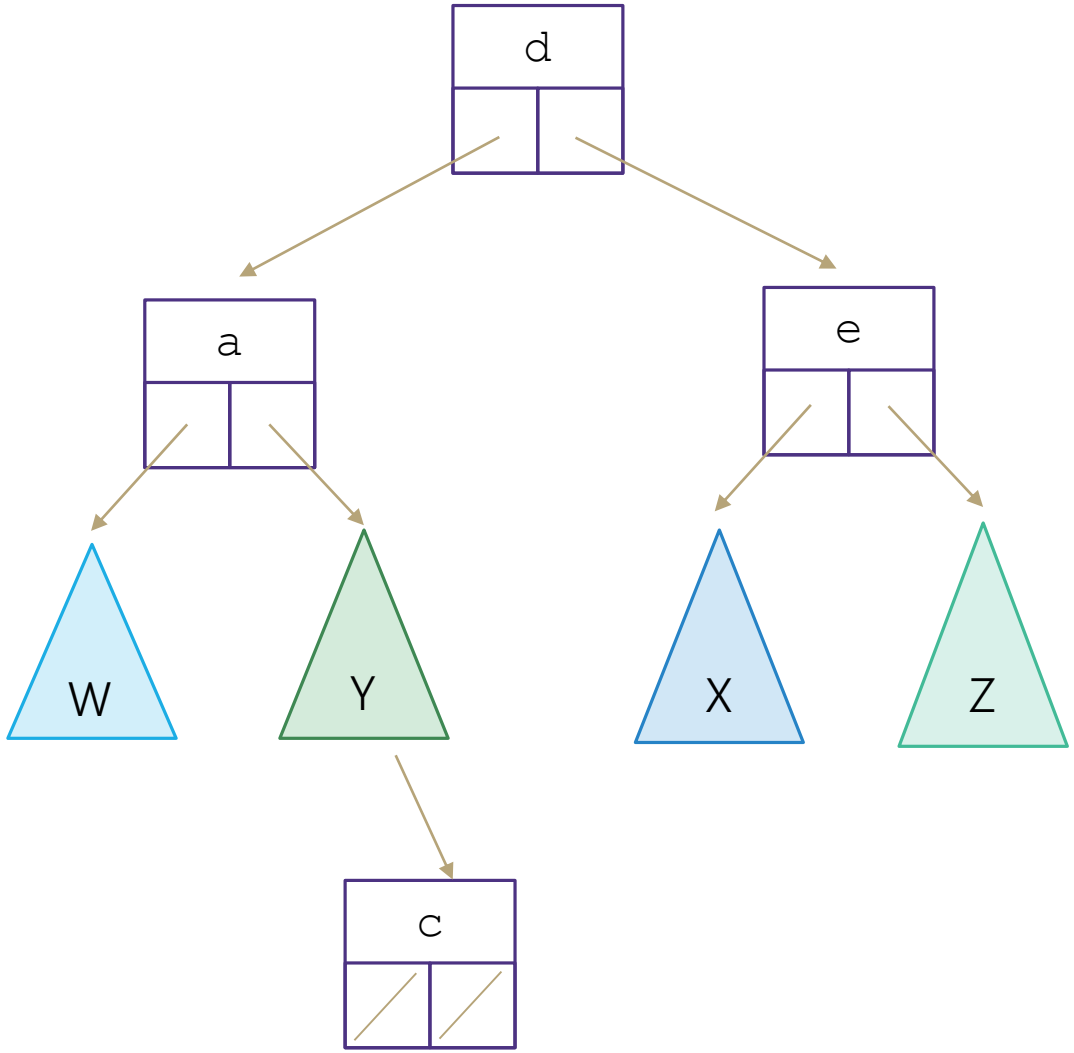


Double Rotations 1



Double Rotations 2





Implementing Tree Map

```
void put(key, value) {
    overallroot = putHelper(key, value, overallroot)
}

Node balanceTree(Node) {
   if(node is null)
      return null
   else if (left subtree is more than 1 level taller than right subtree)
      if (if left subtrees left subtree is bigger than the left's right)
         Rotate myself with my left child
      else
         Double rotate left then right
   else if (right subtree is more than 1 level taller than right subtree)
      if (right subtree's right is taller than it's left)
         Rotate myself with my right child
      else
         Double rotate right then left
}
```

TreeMap<K, V>

state

- overallRoot<K,V>

behavior

- void put(key, value)
- value get(key)
- void set(key, value)
- void remove(key)

ListNode<K, V>

state

- K key
- V value
- ListNode<K, V> left
- ListNode<K, V> right
- int height

behavior

- Construct a new Node