INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 7

Instructor: James Pope

Email: jpope8@gmu.edu

Today

- Last Class
  - Heaps, Tree Traversals, Unit Testing
- Today
  - Midterm Feedback
  - Assignment Reviews
  - Binary Search Trees
  - Balancing & AVLs

Grading & Contesting

- Grade explanations
  - you must come to office hours or make an appointment
- Grades may be contested
  - you must justify your change request
  - any request should be made this week

Feedback

- Textbook
- Weekly Assignments
- Lecture Format
    - Reviews?
- Lecture Content
    - More from the book?
    - Less from the book?
    - Pace?
- Instruction

Questions?

Last Class: Heaps

- Relationship maintained between...
  - parent and child
- Removing items
  - removes the root (“top” item)
- Common uses
  - priority queue
  - sorting

Priority Queue Summary

- Binary heap supports insertion and deletion of the max (min) item in logarithmic worst-case time. Uses an array, easy to implement, and elegant. Often best choice.

Operation Implementation	insert	delMax	findMax
Unordered Array	1	N	N
Ordered Array	N	1	1
Binary Heap	lg(N)	lg(N)	1
???*	1	1	1

Impossible: Lower bounds (omega) for compare sorting is N lg N. If O(1), then heap sort O(N).

Sorting Summary

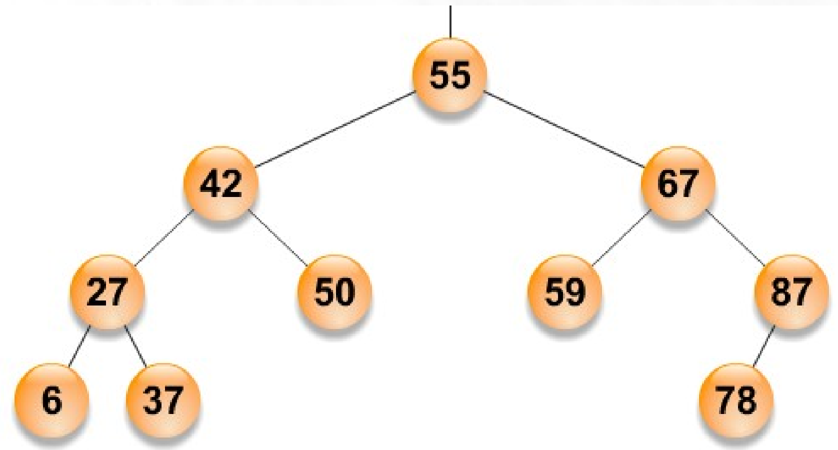
- Heap sort is in place and guarantees N lg N performance, so why not used more?
- Heap sort poor cache relative to merge/quick (compares with values far apart).

Operation Implementation	worst	average	best	in place O(1)	stable	remarks
Selection Sort	N^2	N^2	N^2	yes	no	never use
Insertion Sort	N^2	N^2	N	yes	yes	small n
Merge Sort	N lg N	N lg N	N lg N	no	yes	extra memory
Quick Sort	N^2	N lg N	N lg N	yes*	no	fast practice
Heap Sort	N lg N	N lg N	N lg N	yes	no	poor cache
???	N lg N	N lg N	N lg N	yes	yes	Unknown

* Depending on variant, will assume O(lg(N)) ~ O(1)

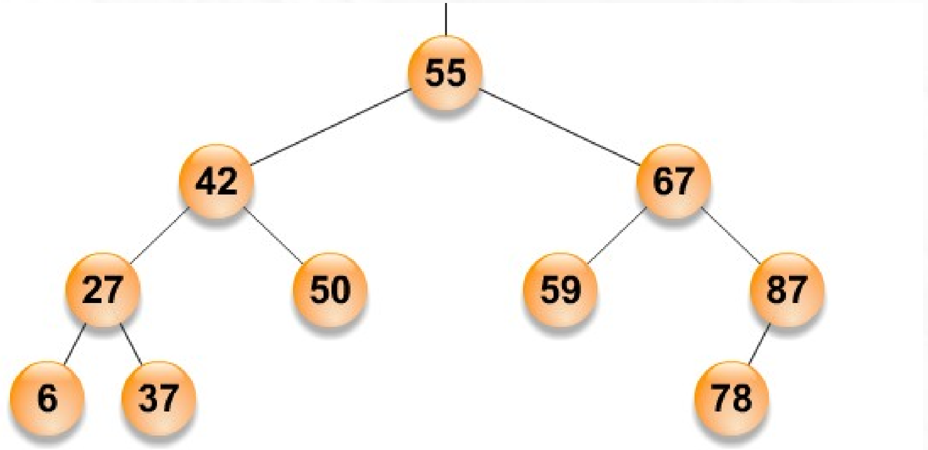
Tree Traversals: Pre Order

- process order 55, 42, 27, 6, 37, 50, 67, 59, 87, 78
  - self
  - left children
  - right children



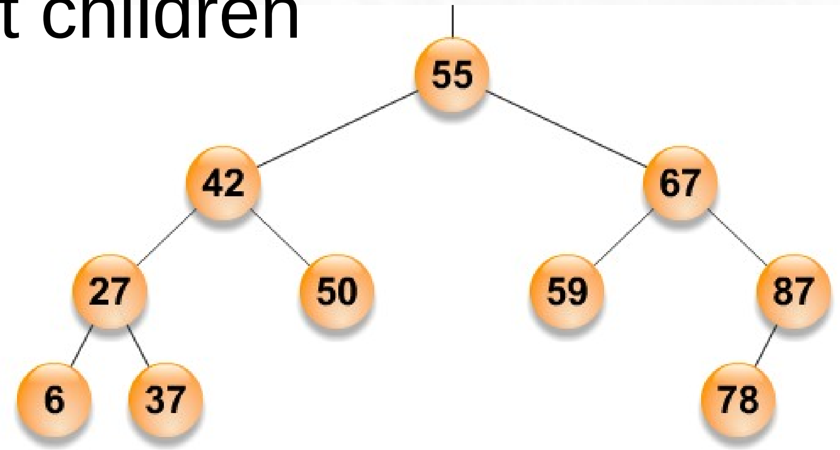
Tree Traversals: Post Order

- process order 6, 37, 27, 50, 42, 59, 78, 87, 67, 55
  - left children
  - right children
  - self



Tree Traversals: In Order

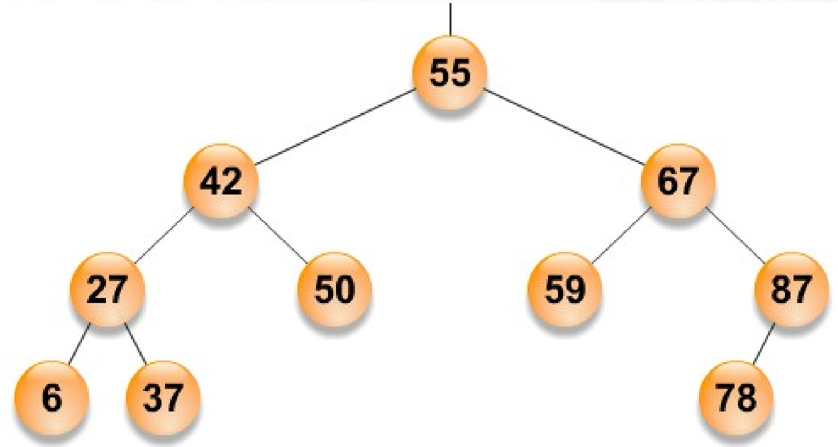
- process order 6, 27, 37, 42, 50, 55, 59, 67, 78, 87
  - left children
  - self
  - right children



If tree satisfies search order property,
Let gravity generate sorted items

Immediate Predecessor/Ancestor

- Immediate Predecessor
  - Left once, right until null
- Immediate Ancestor
  - Right once, left until null
- Max/Min?



Questions?

Symbol Tables

Sedgewick/Wayne 3.1

- Primary purpose is to associate a value with a key (a.k.a. “associative array”, “dictionaries”, “maps”). Key and value are separate objects.
- Can insert key/value and later search for the value by the key
- If the key is ordered (i.e. Comparable) then other convenient operations are possible
- Historically, inserting into the symbol table is called put(key,value) and searching is called get(key)

Key'ed Data Structures

- So far the object stored in the data structure is the key.
  - Requiring the object to adhere to key operations (e.g. compareTo, equals, hashCode) is not always desirable or possible
- Need way to store analogous to arrays with the key as the index and value as object in that position.
- The key is typically an attribute (or can be derived from the attributes) of the value object.

Basic Symbol Table Operations

- The get operation would be similar to accessing an array at an index position.
  - Object value = items[key];
  - Object value = symbolTable.get(key);
- The put operation would be similar to setting the value for an index position.
  - items[key] = value;
  - symbolTable.put(key, value);
- This is why this data structure is commonly called an "associative array".

Basic Symbol Table Operations

Sedgewick/Wayne 3.1

```
public interface BasicSymbolTable <Key, Value>
{
    //Gets the number of elements currently in the queue
    public int size();

    //Determines if there are not elements in the queue.
    public boolean isEmpty();

    //Inserts the value into the table using specified key.
    public void put( Key key, Value value );

    //Finds Value for the given Key.
    public Value get( Key key );

    //Removes the Value for the given Key from the table.
    public Value delete( Key key );

    //Iterable that enumerates each key in the table.
    public Iterable<Key> keys();
}
```

Ordered Symbol Table Operations

- Floor and ceiling.
  - Floor(Key key) largest key <= key
  - Ceiling(Key key) smallest key >= key
- Rank of a key
  - Rank(Key key) number of keys less than key
- Select the k'th key
  - Select(int k) returns key that is the k'th element
- Iterate
  - Iterable<Key> keys(Key lo, Key hi)

Ordered Symbol Table Operations

Sedgewick/Wayne 3.1, Weiss 19.2

```
public interface OrderedSymbolTable <Key extends Comparable, Value>
    extends BasicSymbolTable<Key, Value>
{
    //... previous BasicSymbolTable operations

    public Value min();      //finds and returns minimum value
    public Value max();      //finds and returns maximum value

    public Key floor(Key key);  //largest key <= key
    public Key ceiling(Key key);//smallest key >= key

    //Returns number of keys less than key.
    public int rank( Key key );

    //Finds and returns the k'th Key in the symbol table.
    public Key select( int k );

    //Iterable keys sorted in [lo..hi].
    public Iterable<Key> keys(Key lo, Key hi);
}
```

Key Operations

- For basic symbol table, the key has to have the following operations.
  - Key.equals(Object o)
  - Optionally: Key.hashCode()
- For ordered symbol tables, the key must have an additional ordering operation.
  - Key.compareTo(Object o)

Symbol Tables Conventions

Sedgewick/Wayne 3.1

- Do not allow keys to be null
- No key can be associated with null value
  - If get(key) returns null, know not in table
- Do not allow duplicated values for a key
  - put(key1, val1) followed by put(key1, val2) overwrites previous val1
- Iteration allowed on the keys only
  - Can then use key to get associated value

Questions?

Symbol Table Implementations

- Can we efficiently handle large number of get operations after large number of put/get operations?
- Naive
  - Unordered linked list
  - Ordered array
- Trees
  - Binary Search Trees
  - Balanced Variants (AVL, Red-Black, AA)
- Hash Tables

Binary Search Tree

- A type of binary tree!
- Relationship maintained between...
  - parent and both children
- Relationship
  - parent > elements in left sub tree
  - parent < elements in right sub tree
  - both children are binary search trees
  - no duplicates (how do we handle this?)

Binary Search Tree: Example

- White board time...

- insert random numbers

- insert a sorted list
  - what's the problem?

Binary Search Tree: Big-O

- Degenerate binary search trees
  - What is the height?
  - What is big-O of:
    - finding an element
    - inserting an element
    - deleting an element

Binary Search Tree: Big-O

- Balanced binary search trees
  - What is the height?
  - What is big-O of:
    - finding an element
    - inserting an element
    - deleting an element

Binary Search Tree: Big-O

- So... binary search trees
  - What is the height?
  - What is worst/best of:
    - finding an element
    - inserting an element
    - deleting an element

Binary Search Tree: Delete

- Cases to consider.
  - No children, easy
  - 1 child, easy
  - 2 children, hard
- Can select predecessor or successor. Safe because order is maintained in both cases.
  - Hibbard deletion always selects successor
  - May consider random predecessor/successor
- Typically helper min and deleteMin methods

Binary Search Tree: Min/Max

- Can find min by continuously going left
- Can find max by continuously going right
- Easiest to do iteratively

Binary Search Tree: deleteMin/Max

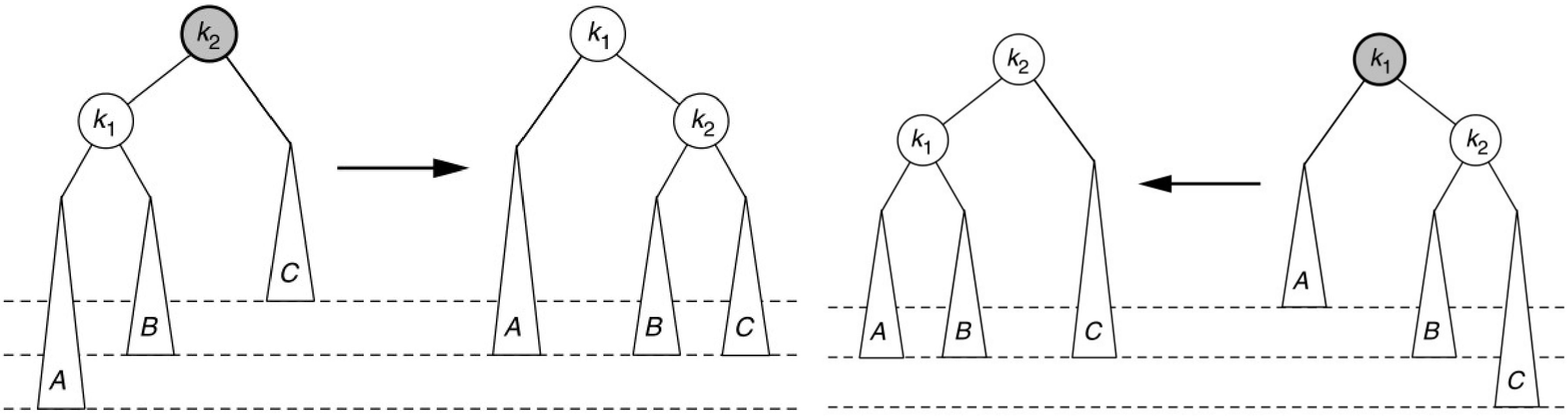
Weiss Figure 19.11

- Usually recursive, keep parent in stack instead of iterative loop.
  - Min: Keep going left, if a right subtree, attach
  - Max: Keep going right, if left subtree, attach
- Seems to disconnect subtree but correctly resets as it unwinds.

Questions?

How do we improve performance?

- Balance!
  - preferably self-balancing! (balance as you add/remove/search the tree)
- How? Rotate!



(a) Before rotation (b) After rotation (a) After rotation (b) Before rotation

Weiss 19.4.2 Figures 19.23 and 19.26

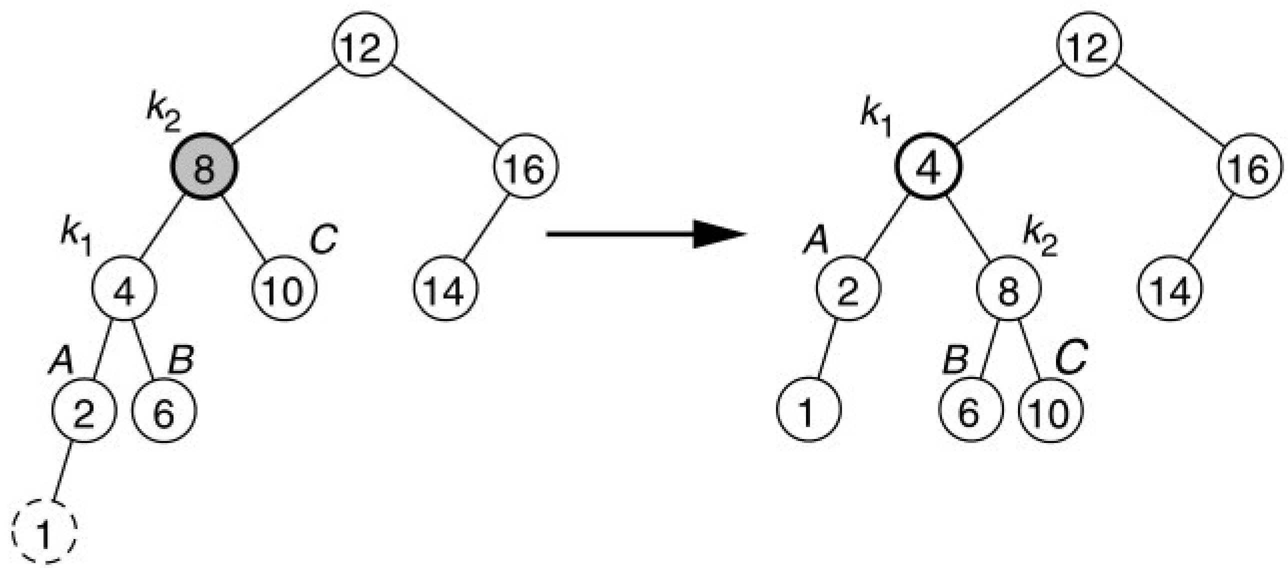
How do we do this?

- Let me count the ways...
  - AVL Trees
  - Red-Black (2-3) Trees
  - AA Trees (will mention but not cover)
  - B-Trees
  - Splay Trees (will mention but not cover)
  - ...
- What's the difference?
  - generally how and when to rotate

AVL Trees

- Not used much, but often taught
- Basic idea
  - Left and right subtrees shouldn't differ by a height of more than 1
- When to fix balance?
  - inserting/deleting
- Observation: Only nodes along the path from insertion point to root may need to potentially be balanced
  - Applies to many other balanced trees

Height Wrong? Fix it!



(a) Before rotation

(b) After rotation

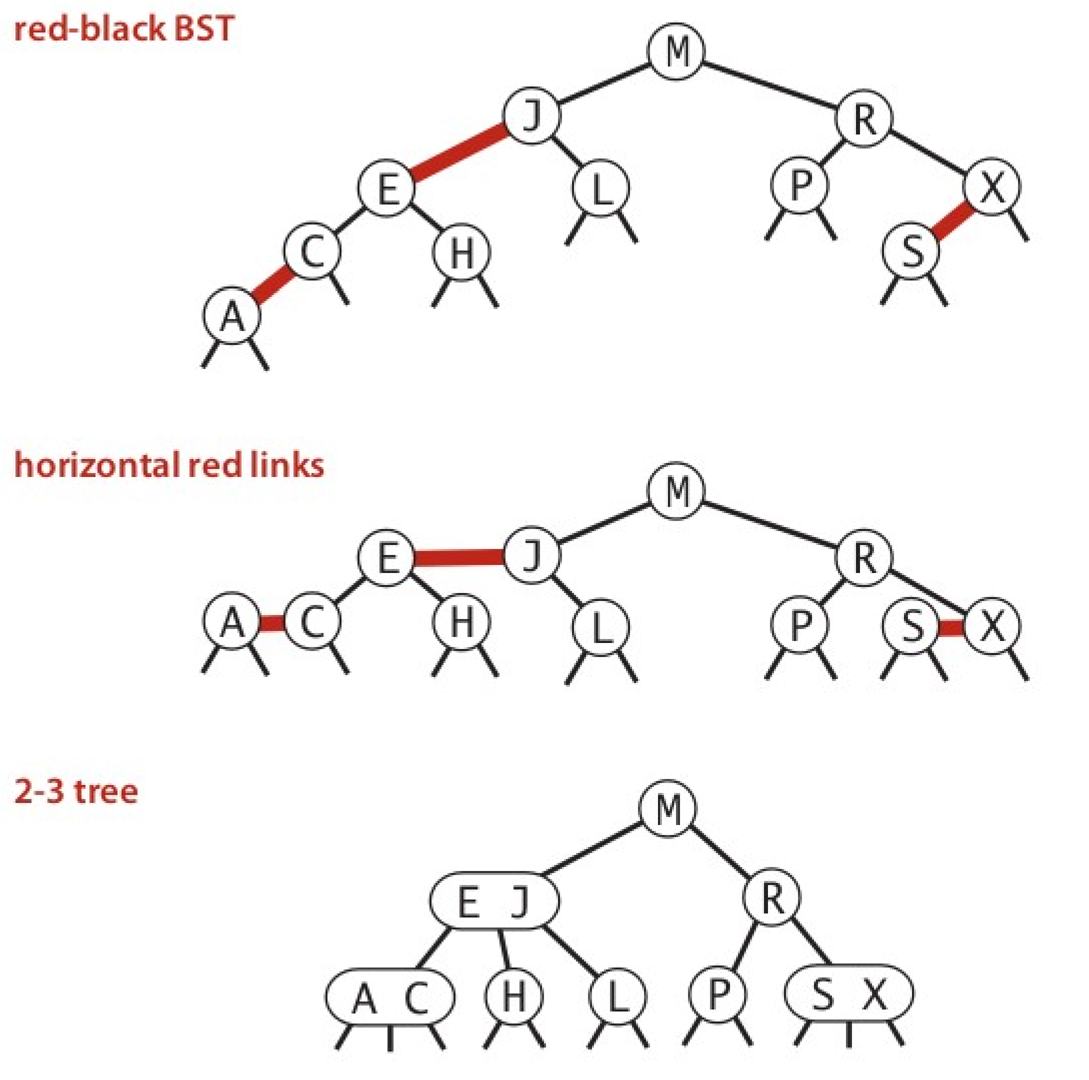
Red-Black Trees

- Often taught, often used, more complicated
- Basic idea (two interpretations):
  - Nodes can be red or black, keep the "black height" even
  - Keep 1-to-1 correspondence with a perfectly balanced 2-3 tree, red node indicates a 3 node
- When to fix balance?
  - inserting/deleting

Red-Black is a 2-3 Tree as a BST

Sedgewick/Wayne 3.3

- 2-3 Trees are balanced but difficult to implement
- Binary Search Trees are easy to implement but not balanced
- Rules needed to keep 1-1

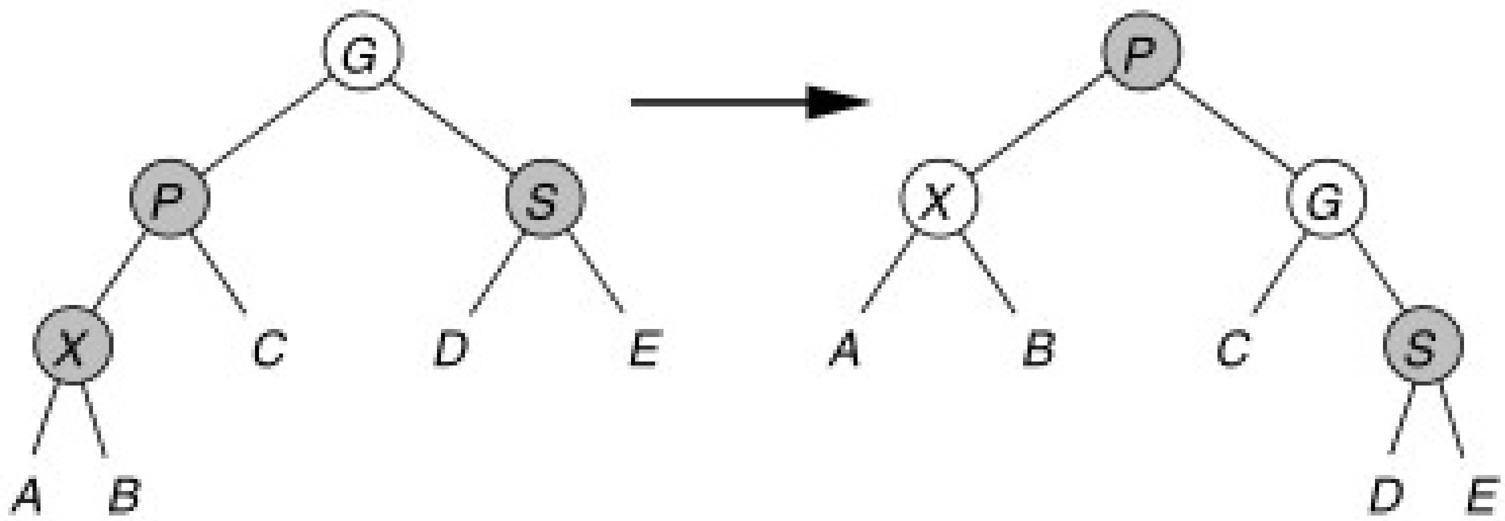


1-1 correspondence between red-black BSTs and 2-3 trees

Red-Black Tree Rules

- Nodes can be red or black
  - the root is (usually) black
  - null links are always black
- Red nodes have black node children
- All paths from a given node to its descendent leaves contains the same number of black nodes
  - if not, 6 different situations defined with specific solutions

Black height wrong? Fix it!



(a) Before rotation (b) After rotation

AA Trees

- Seldom taught, simpler variation of a red-black tree
- Basic idea:
  - Variation of a red-black tree
  - Red nodes only added to right sub tree
- When to fix balance?
  - inserting/deleting
- http://user.it.uu.se/~arnea/ps/simp.pdf

Red-Black hard? This is easier!

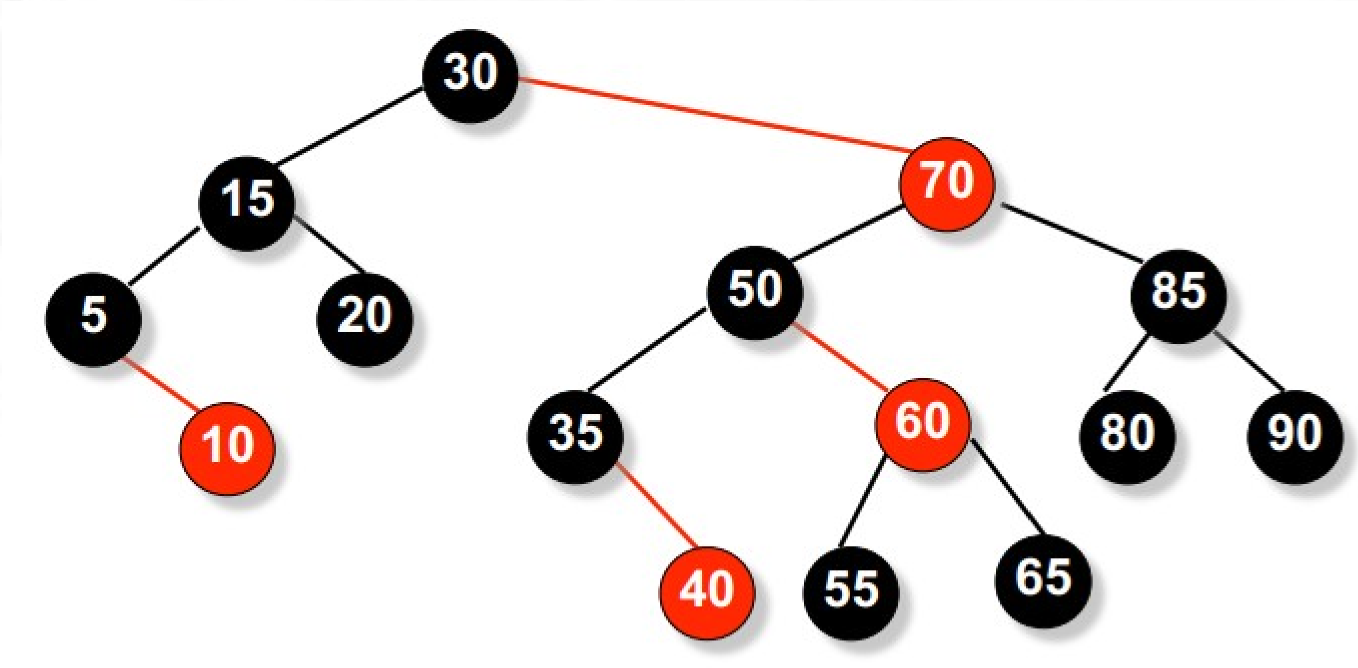
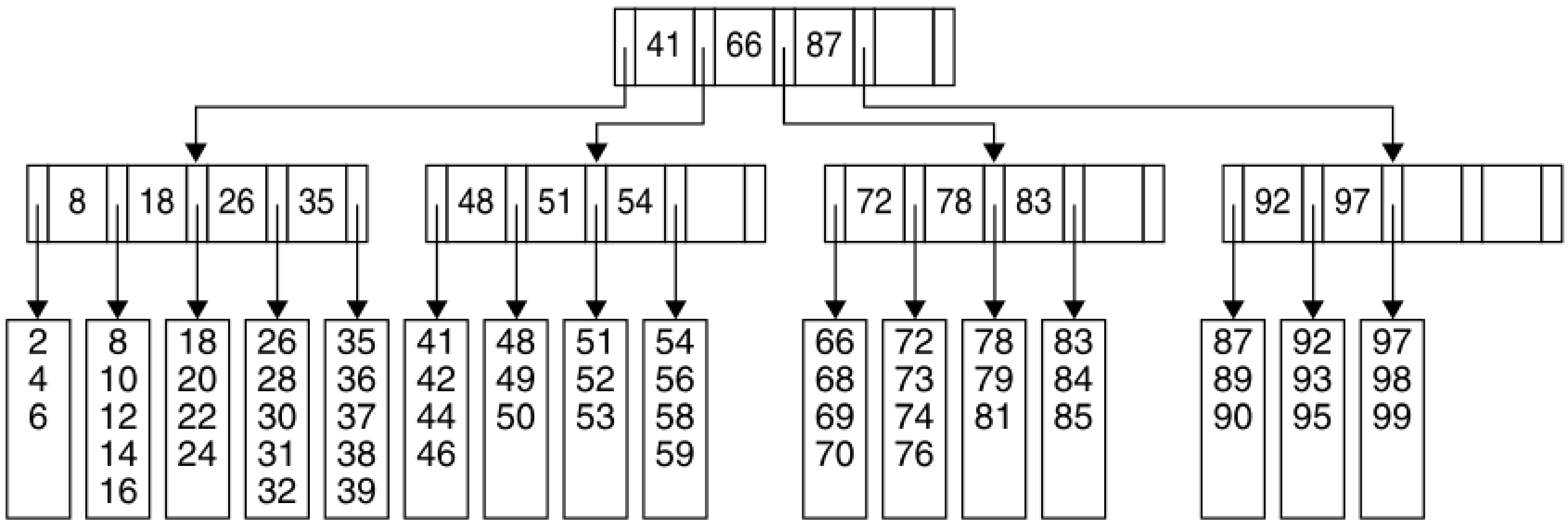


Image Source: http://web.eecs.umich.edu/~sugih/courses/eecs281/f11/lectures/12-AAtrees+Treaps.pdf (page 1)

B Trees

- Commonly taught, commonly used, easy to implement
- Basic idea:
  - Tree + List = B-Tree
  - We want a really big list...
    - but if it's too big it won't fit into memory...
    - ... so use a tree to break it up
- When to fix balance?
  - inserting/deleting

2 for 1: It's a list! In a tree!



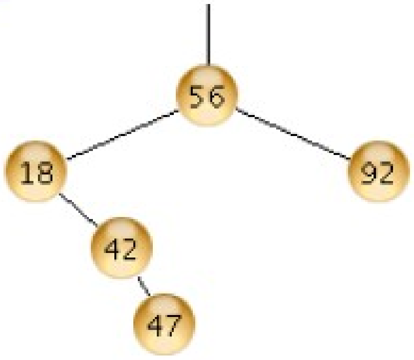
Splay Trees

- Not really balancing, but is optimizing in a way
- Basic idea:
  - Balance so the most recently accessed item is at the root
- When to fix balance?
  - inserting/deleting/searching
- Look like binary search trees but they keep moving around
- http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf

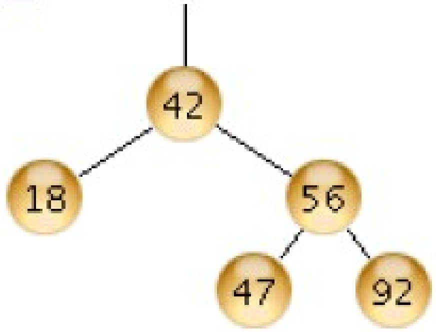
Questions?

Are These AVL Trees?

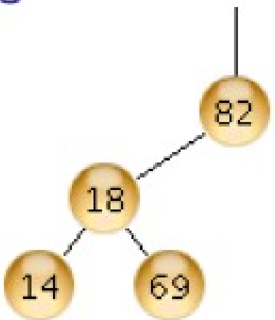
1



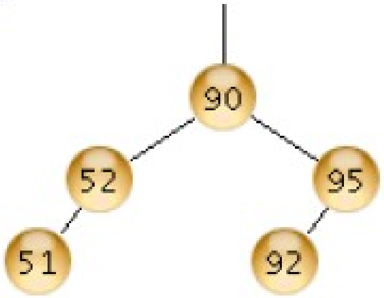
2



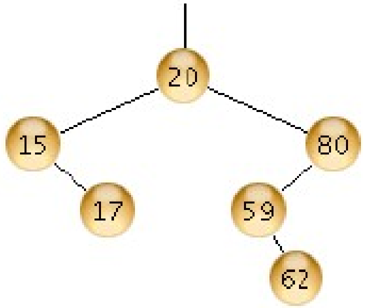
3



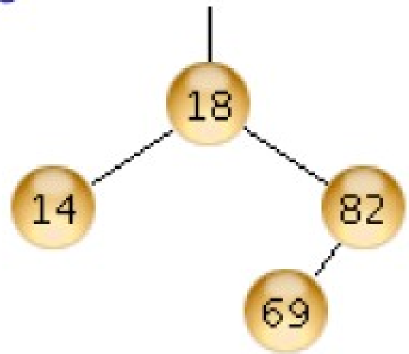
4



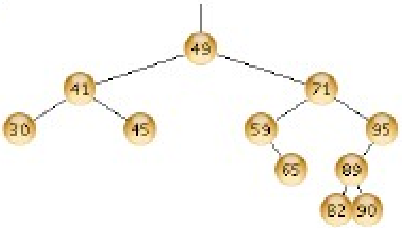
5



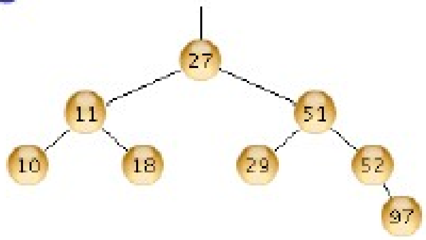
6



7

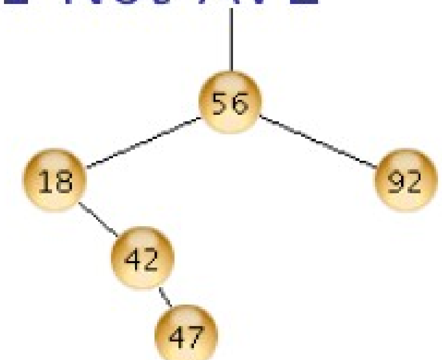


8



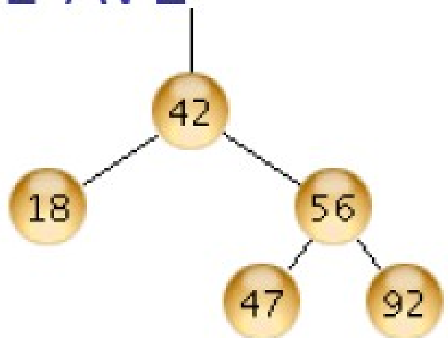
Answers

1 Not AVL

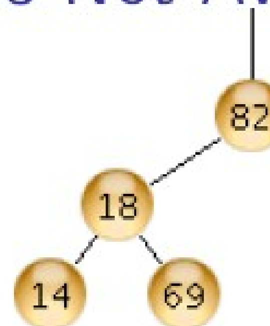


Left 0, Right 1

2 AVL

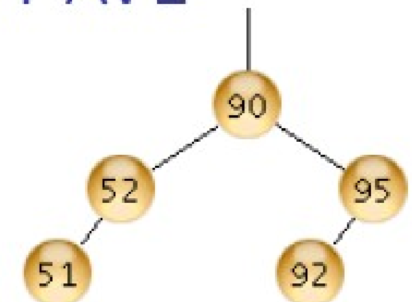


3 Not AVL

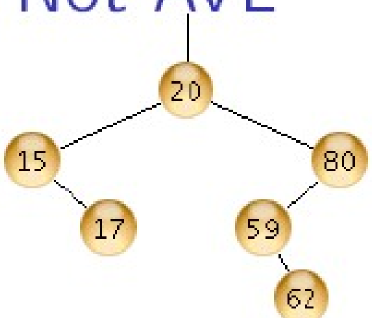


Left 2, Right 0

4 AVL

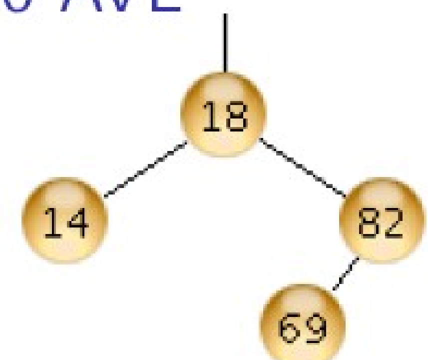


5 Not AVL

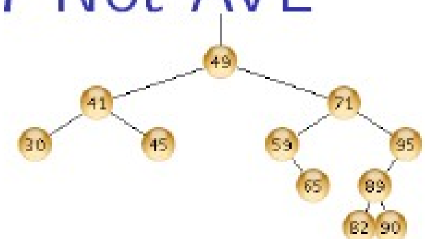


80 not AVL

6 AVL

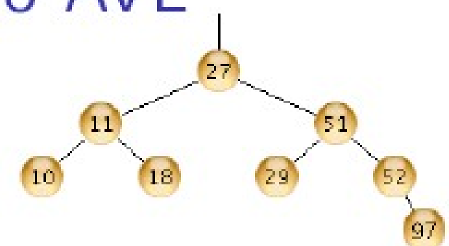


7 Not AVL



Left 2, Right 4
95 not AVL

8 AVL

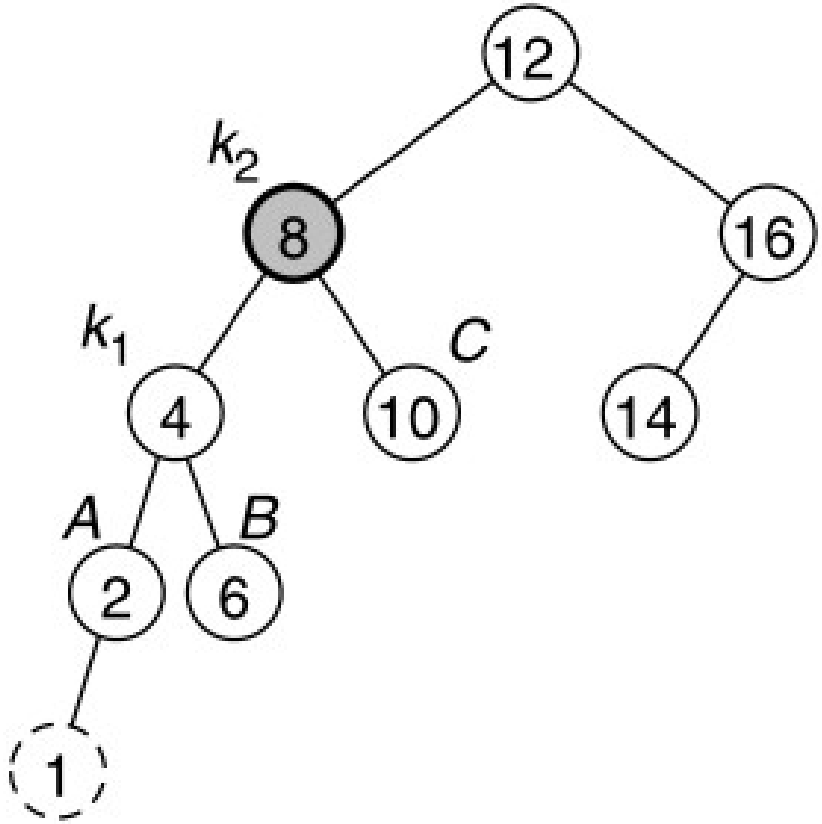


AVL Tree Balance Cases

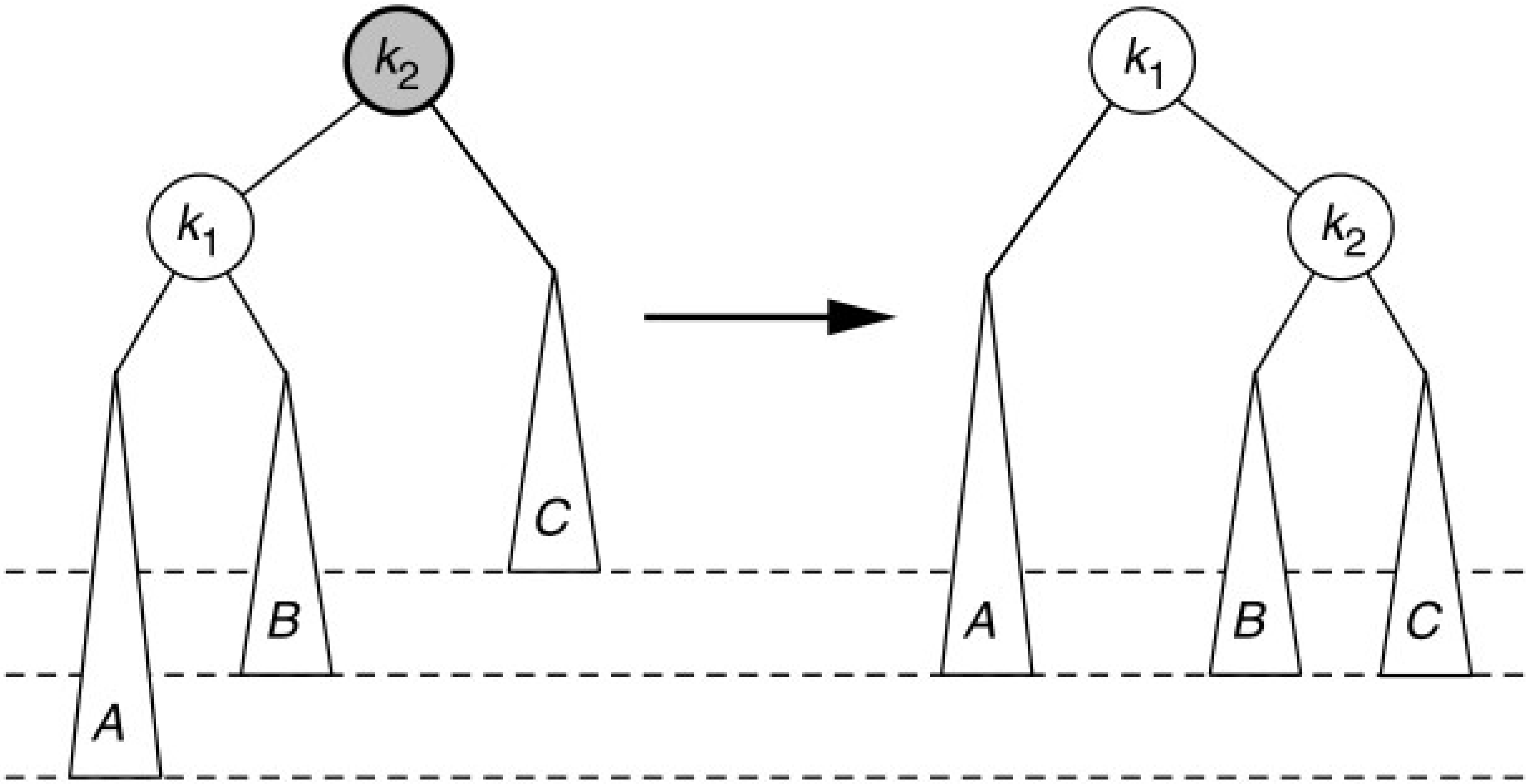
Weiss 19.4.1

- Height imbalance means some node X whose two subtrees differ by 2
    1. Insertion left subtree of the left child of X
    2. Insertion right subtree of the left child of X
    3. Insertion left subtree of the right child of X
    4. Insertion right subtree of the right child of X
- Symmetry between 1 and 4 and 2 and 3
- Similar cases when a deletion causes an imbalance

Case 1



Single Rotation



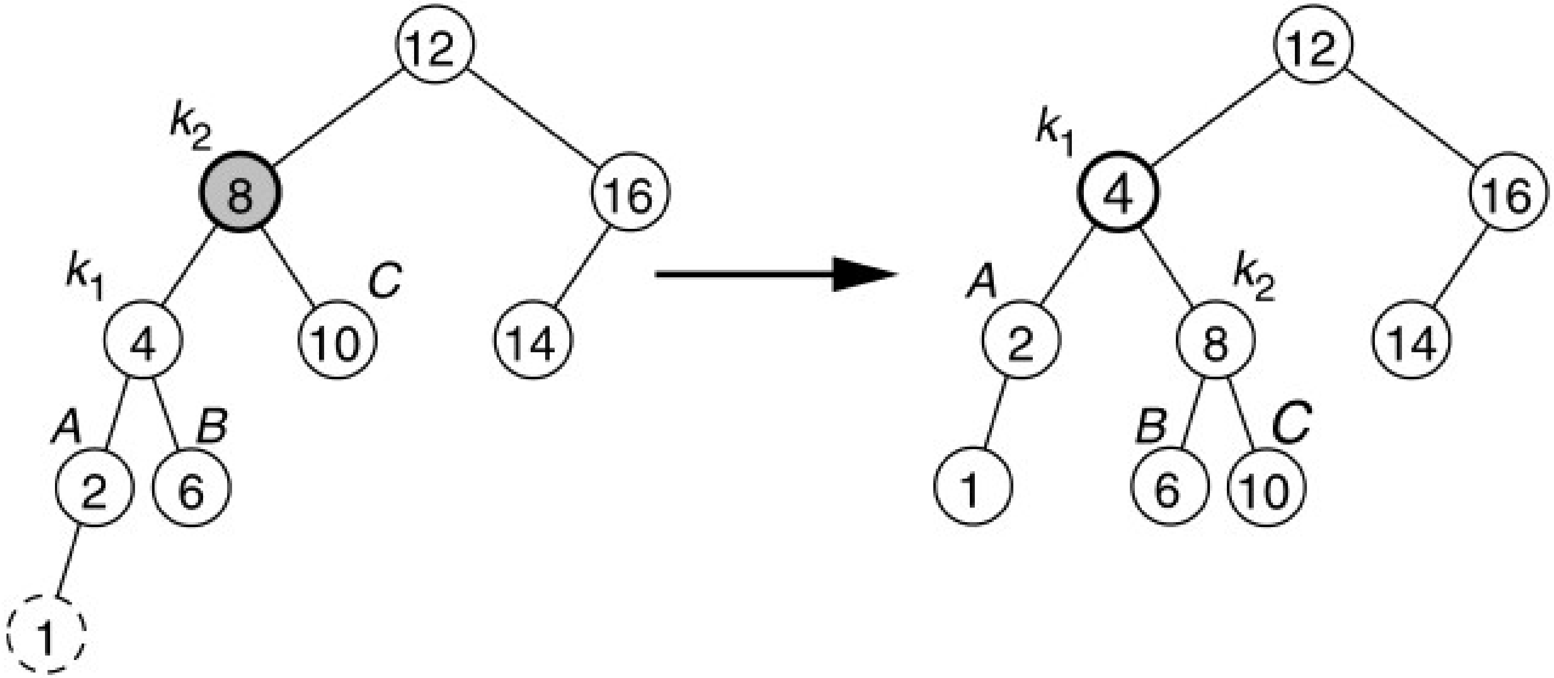
(a) Before rotation (b) After rotation

Single Rotation Idea

Weiss 19.4.1

- Any BST can be “collapsed” to bottom to make the items in sorted order
- Pick up k1 above k2 and let gravity take effect. Thus k1 becomes subtree root and k2 drops to right of k1
- Have to move subtree B to k2 left child
- Previously subtree B held items between k1 and k2
- After rotation subtree B remains between k1 and k2 maintaining order property

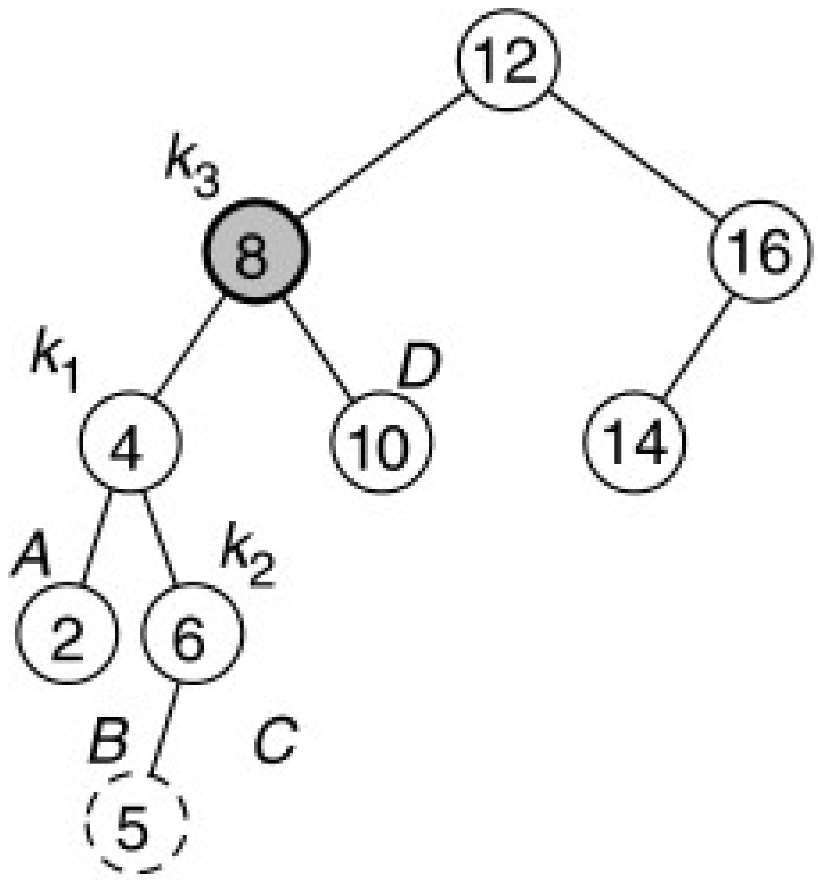
Case 1 Fixed Single Rotation



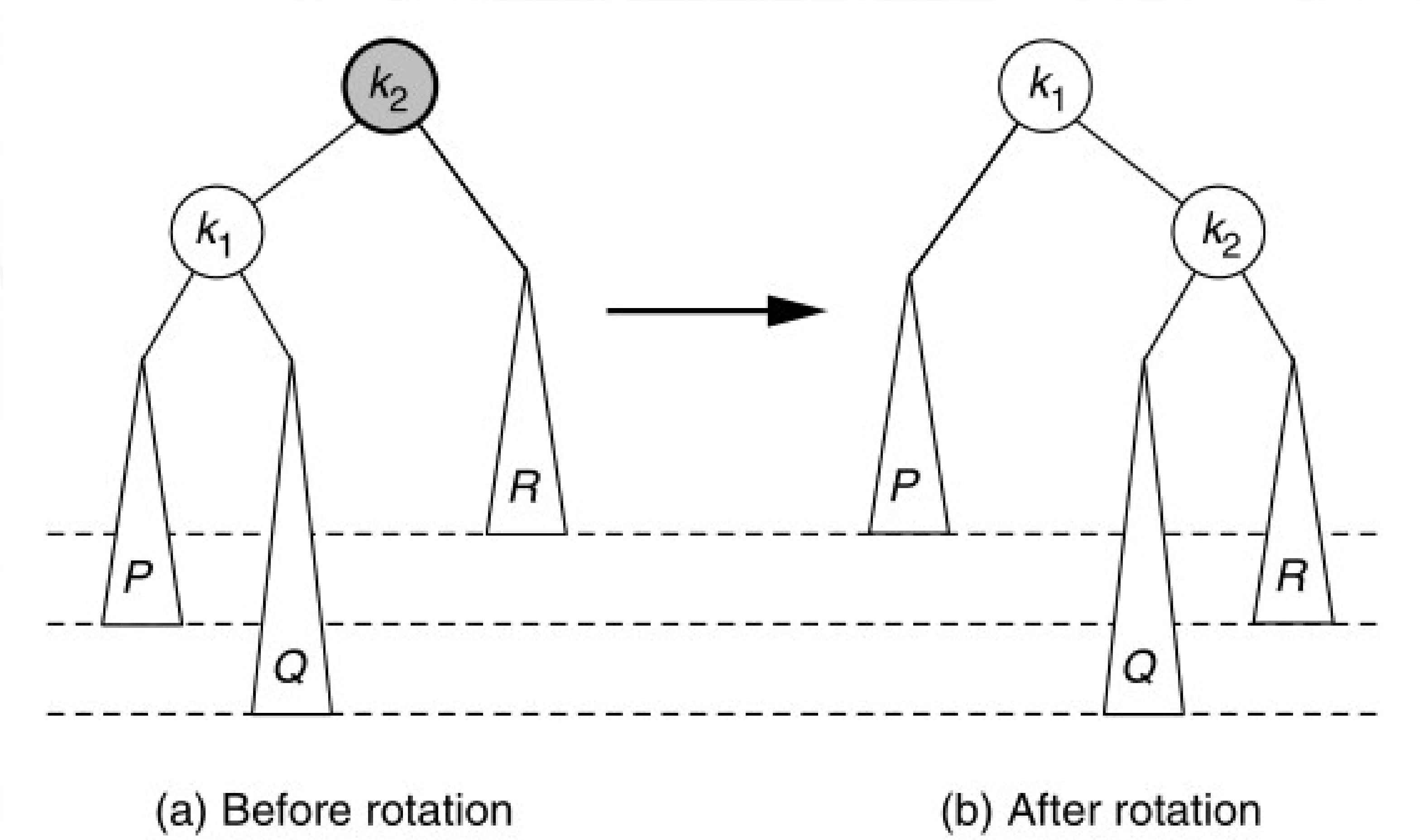
(a) Before rotation

(b) After rotation

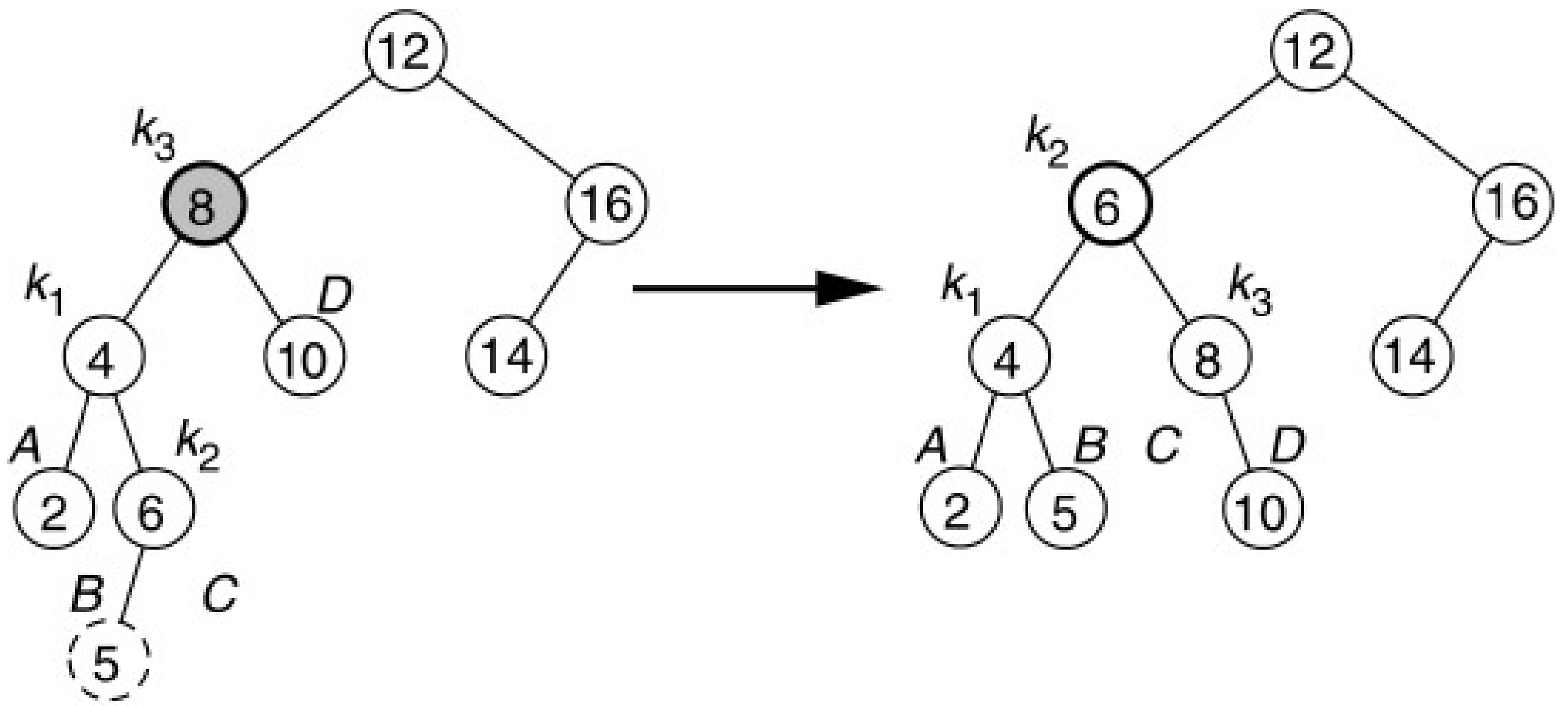
Case 2



Single Rotation Won't Fix!



What We Want

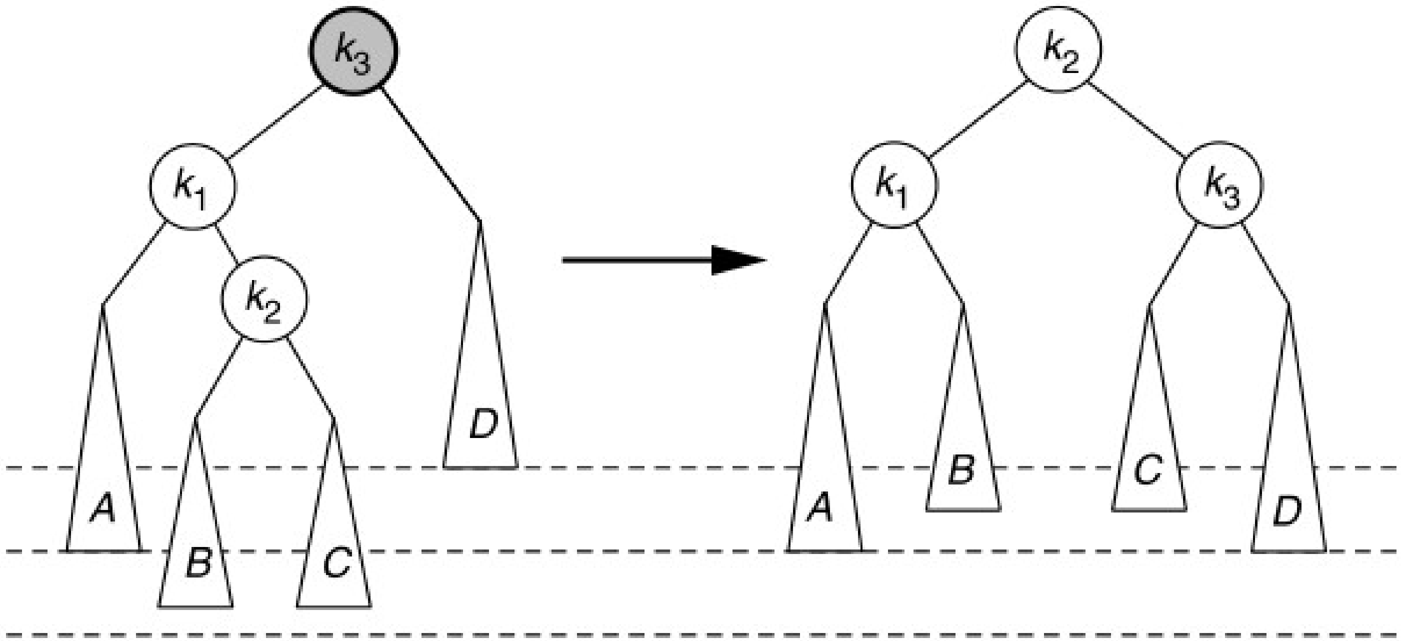


(a) Before rotation

(b) After rotation

Left-Right Double Rotation

- Left Rotate at k_1
- Right Rotate at k_3



(a) Before rotation (b) After rotation

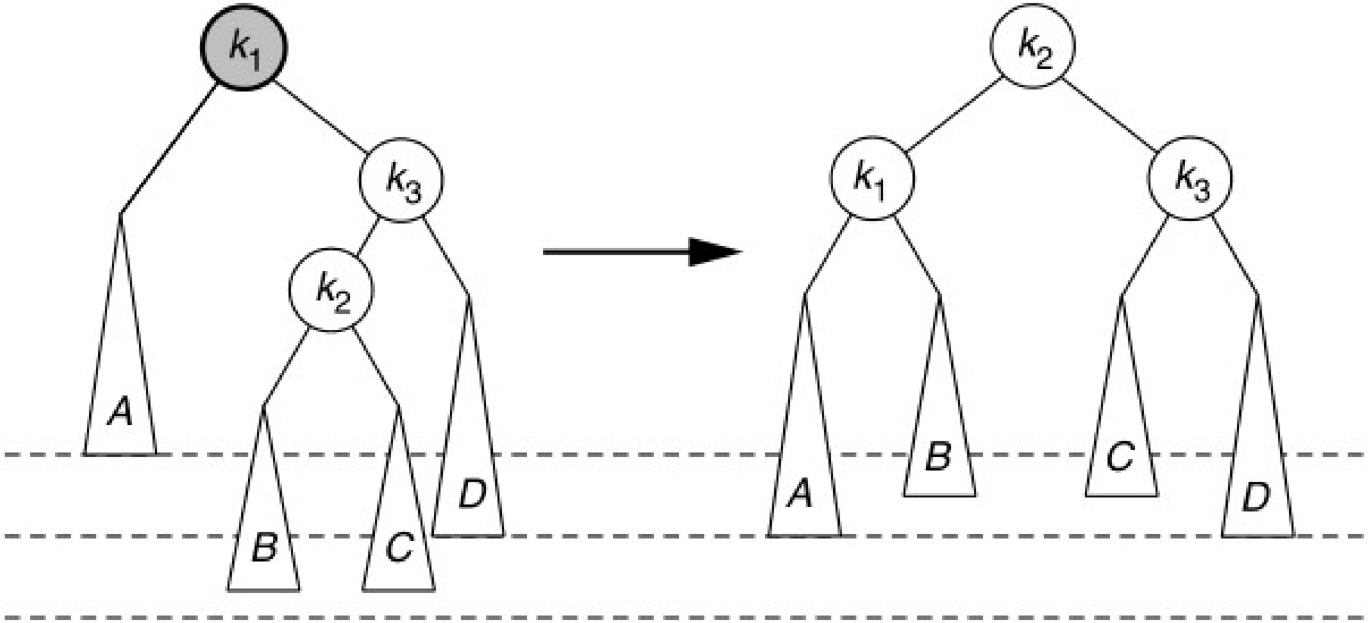
Double Rotation Idea

Weiss 19.4.3

- Know either/both subtrees B and C is two levels deeper than D
- Rotation between X's child and grandchild
- Rotation between X and its new child
- Subtree B remains between k1 and k2
- Subtree C remains between k2 and k3

Right-Left Double Rotation

- Right Rotate at k_3
- Left Rotate at k_2

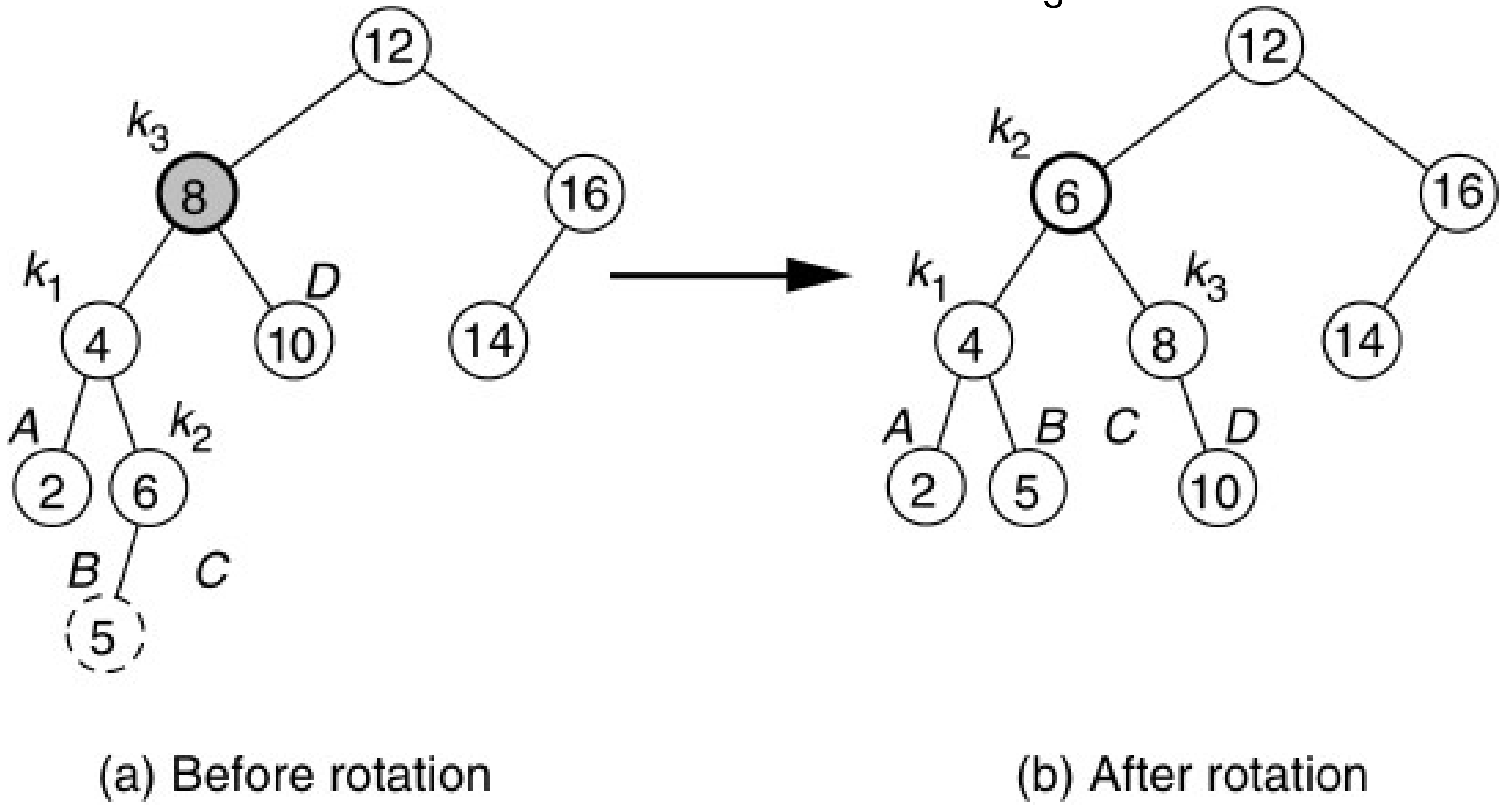


(a) Before rotation (b) After rotation

What We Did: Left-Right Rotation

Left Rotate at k_1

Right Rotate at k_3



AVL Tree Practice

- Gnarley trees, but other resources
  - http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html
  - http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html

M-ary Trees

- aka. n-ary and k-ary trees
- what is an m-ary tree?
  - m is the branching factor
- number of leaves when it's full and complete?
  - first level m^0, second m^1, third m^2... m^h
- number of nodes when it's full and complete?
  - $(m^{h+1}-1)/(m-1)$
    - for binary trees this was m=2
    - so $2^{h+1}-1/(2-1) = 2^{h+1}-1$

Symbol Table Summary

- Generally use hash table unless guaranteed performance or need ordered operations

Implementation	Worse-Case		Average-Case		Order Ops	remarks
	Search	Insert	Search	Insert		
Unordered List	N	N	N	N	No	
Ordered Array	lg N	N	lg N	N	Yes	
BST	**N**	**N**	**lg N**	**lg N**	**Yes**	**Easy**
AVL	**lg N**	**lg N**	**lg N**	**lg N**	**Yes**	**Easy**
Red-Black	lg N	lg N	lg N	lg N	Yes	Often Used*
HT Chaining	N	N	N / M	N / M	No	Often Used*
HT Probing	N	N	1	1	No	

* Good constants and relatively easy to implement, used in many libraries

Questions?

Assignments: PA6

- PA5
  - Use Comparable[] items. The methods swap, sink, swim use integers as index into items.
  - Iteration is “level order”
- PA6
  - Implement ordered symbol table using binary search tree.
  - Iteration is sorted order
  - Will use and make balanced (AVL) for next PA7

Free Question Time!