CSE 326 Data Structures
Midterm Review

Hal Perkins

Winter 2008

Dates

- Midterm Friday!
- Project 2b due next Wednesday
- Homework 4
  - Out soon, due a week from Friday

Logistics

- Closed Notes
- Closed Book except for one 5x8 or smaller notecard with hand-written (only) notes
- Open Mind
- You may bring a calculator, though don't even think about loading it with notes or programs. And you probably won't find it of much use anyway.

Material Covered

- Everything we've talked/read in class up to AVL trees
  - And for AVL trees, up to inserting and rotations, but not implementations in Java

Material Not Covered

- We generally won't make you write syntactically correct Java code (pseudocode okay unless requested otherwise)
- We won't make you do a super hard proof
- We won't test you on the gory details of generics, interfaces, etc. in Java
  - But you should know the basic ideas since we spent lecture time on them and had to deal with them in project 2A

Order Notation: Definition

O(f(*n*)) : a set or class of functions

g(n) $\in$ O(f(n))   iff there exist consts c and n_0 such that:

g(n) $\leq$ c f(n) for all $n \geq n_0$

Example: g(n) =1000n vs. f(n) = n^2

Is g(n) $\in$ O(f(n)) ?

Pick: n0 = 1000, c = 1

Log?

$\log_k n \in O(\log_2 n)$?

$\log_2 n^2 \in O(\log_2 n)$?

Definition of Order Notation

- Upper bound: $T(n) = O(f(n))$ Big-O

  Exist constants c and n' such that

  $T(n) \leq c\, f(n)$ for all $n \geq n'$
- Lower bound: $T(n) = \Omega(g(n))$ Omega

  Exist constants c and n' such that

  $T(n) \geq c\, g(n)$ for all $n \geq n'$
- Tight bound: $T(n) = \theta(f(n))$ Theta

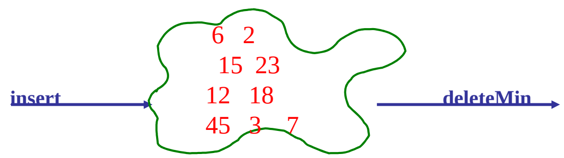
  When both hold:

  $T(n) = O(f(n))$

  $T(n) = \Omega(f(n))$

Priority Queue ADT

- Checkout line at the supermarket ???
- Printer queues ???
- operations: insert, deleteMin



Implementations of Priority Queue ADT

	insert	deleteMin
Unsorted list (Array)		
Unsorted list (Linked-List)		
Sorted list (Array)		
Sorted list (Linked-List)		
Binary Search Tree (BST)		

Binary Heap

Tree Review

root(**T**):

leaves(**T**):

children(B):

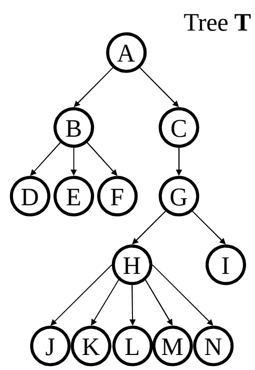
parent(H):

siblings(E):

ancestors(F):

descendents(G):

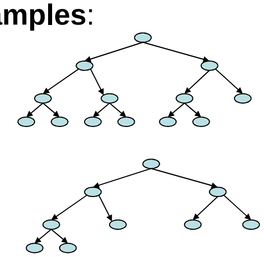
subtree(C):



Heap **Structure** Property

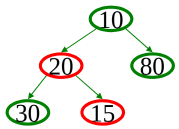
- A binary heap is a ***complete*** binary tree.
- **Complete binary tree** – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:

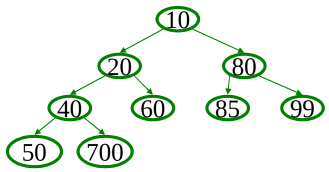


Heap **Order** Property

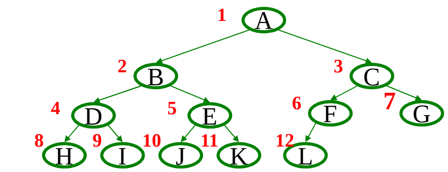
Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.



not a heap



Representing Complete Binary Trees in an Array



From node **i**:

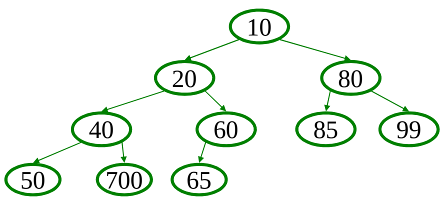
left child:
right child:
parent:

implicit (array) implementation:

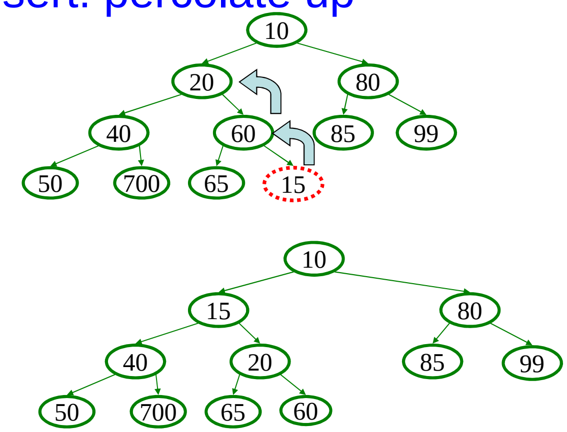
	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap Operations

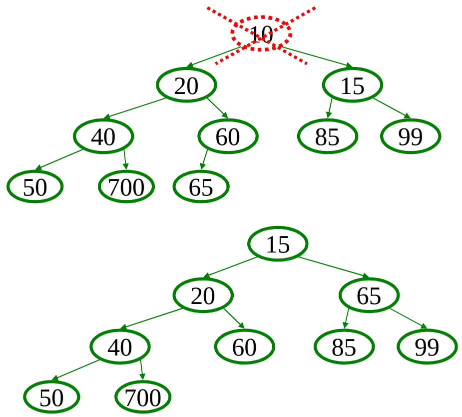
- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.



Insert: percolate up



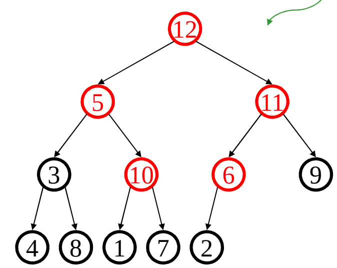
DeleteMin: percolate down



BuildHeap: Floyd's Method

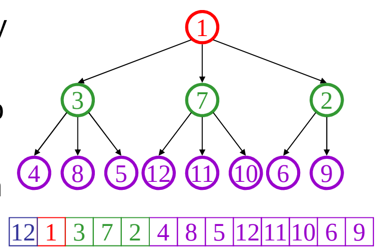
12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!



d-Heaps

- Each node has *d* children
- Still representible by array
- Good choices for *d*:
  - (choose a power of two for efficiency)
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block



Operations on *d*-Heap

- Insert : runtime =
- deleteMin: runtime =

Does this help insert or deleteMin more?

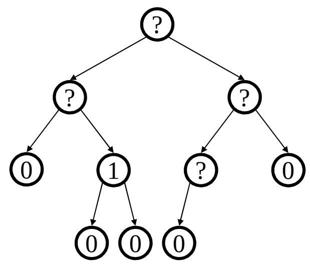
Definition: Null Path Length

null path length (npl) of a node *x* = the number of nodes between *x* and a null in its subtree

OR

npl(x) = min distance to a descendant with 0 or 1 children

- *npl*(null) = -1
- *npl*(leaf) = 0
- *npl*(single-child node) = 0



Equivalent definitions:

1. *npl(x)* is the height of largest complete subtree rooted at *x*
2. *npl(x)* = 1 + min{*npl*(left(x)), *npl*(right(x))}

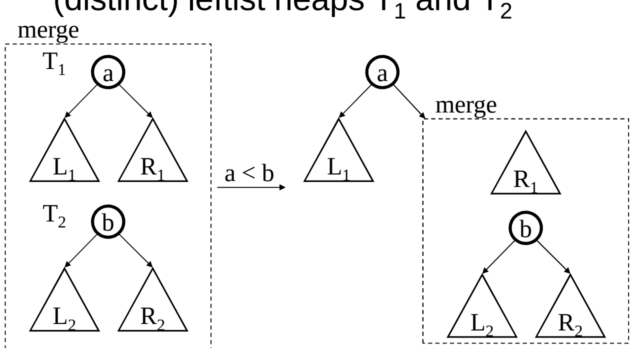
Leftist Heap Properties

- Heap-order property
  - parent's priority value is ≤ to childrens' priority values
  - result: minimum element is at the root
- Leftist property
  - For every node *x*, *npl*(left(*x*)) ≥ *npl*(right(*x*))
  - result: tree is at least as "heavy" on the left as the right

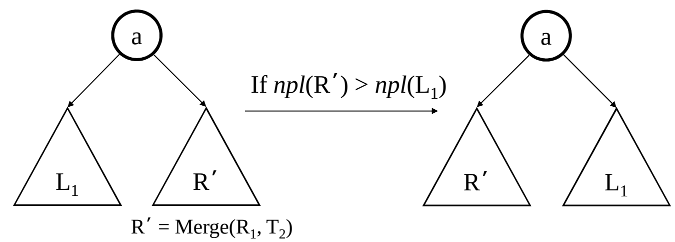
Are leftist trees…
complete?
balanced?

Merging Two Leftist Heaps

- merge(T_1,T_2) returns one leftist heap containing all elements of the two (distinct) leftist heaps T_1 and T_2

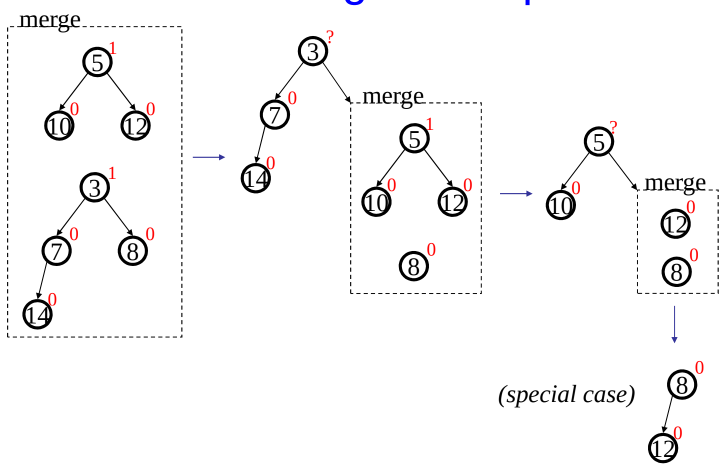


Leftist Merge Continued

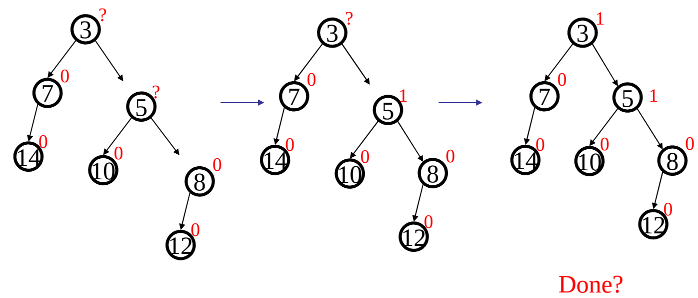


runtime:

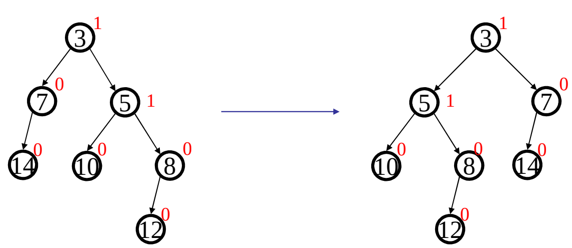
Leftist Merge Example



Sewing Up the Leftist Example



Finally…(Leftist)



Skew Heaps

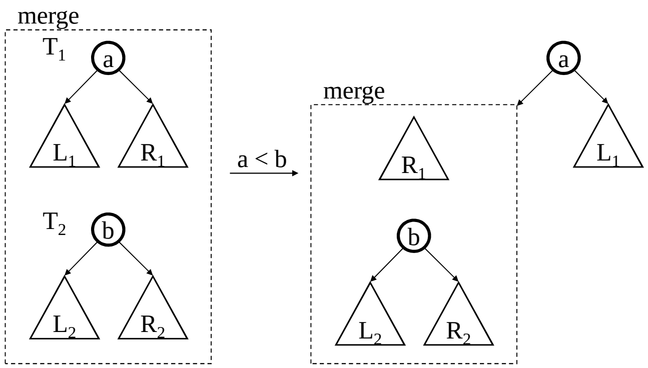
Problems with <u>leftist</u> heaps

- extra storage for npl
- extra complexity/logic to maintain and check npl
- right side is "often" heavy and requires a switch

Solution: <u>skew</u> heaps

- "blindly" adjusting version of leftist heaps
- merge *always* switches children when fixing right path
- <u>amortized time</u> for: merge, insert, deleteMin = O(log *n*)
- however, <u>worst case time</u> for all three = O(*n*)

Merging Two Skew Heaps



Only one step per iteration, with children *always* switched

Yet Another Data Structure: Binomial Queues

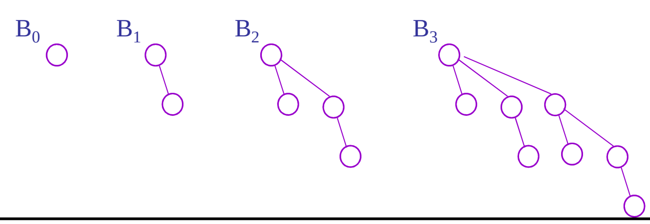
- Structural property
  - Forest of binomial trees with at most one tree of any height
- Order property
  - Each binomial tree has the heap-order property

What's a forest?

What's a binomial tree?

The Binomial Tree, B_h

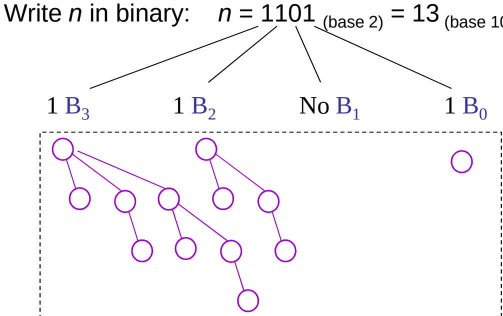
- B_h has height h and exactly 2^h nodes
- B_h is formed by making B_{h-1} a child of another B_{h-1}
- Root has exactly h children
- Number of nodes at depth d is binomial coeff. $\binom{h}{d}$
  - Hence the name; we will *not* use this last property



Binomial Queue with n elements

Binomial Q with n elements has a *unique* structural representation in terms of binomial trees!

Write n in binary: $n = 1101_{\text{(base 2)}} = 13_{\text{(base 10)}}$



Merging Two Binomial Queues

Essentially like adding two binary numbers!

1. Combine the two forests
2. For k from 1 to maxheight {
   a. $m \leftarrow$ total number of B_k's in the two BQs
   b. if m=0: continue;
   c. if m=1: continue;
   d. if m=2: combine the two B_k's to form a B_{k+1}
   e. if m=3: retain one B_k and combine the other two to form a B_{k+1}

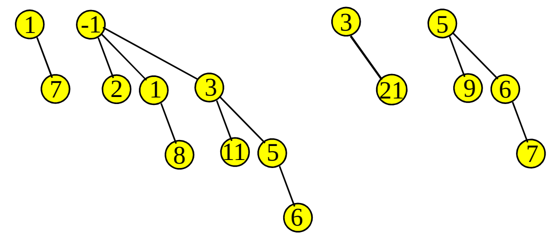
}

# of 1's
0+0 = 0
1+0 = 1
1+1 = 0+c
1+1+c = 1+c

Claim: When this process ends, the forest has at most one tree of any height

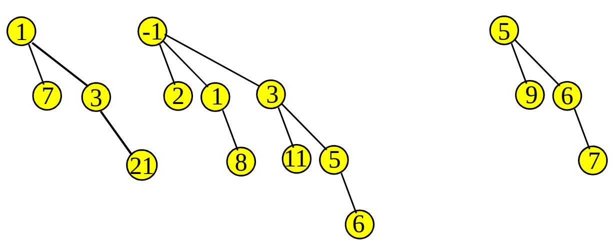
Example: Binomial Queue Merge

H1: H2:



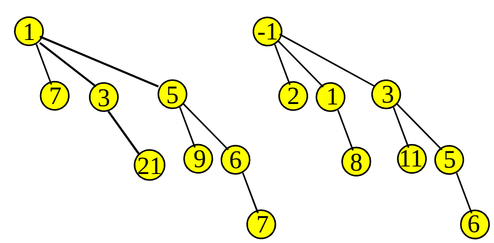
Example: Binomial Queue Merge

H1: H2:



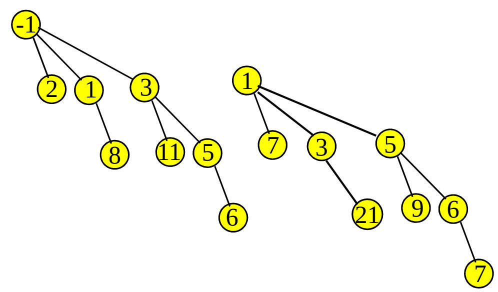
Example: Binomial Queue Merge

H1: H2:



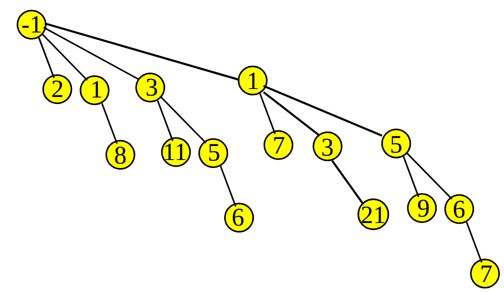
Example: Binomial Queue Merge

H1: H2:



Example: Binomial Queue Merge

H1: H2:

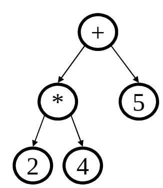


More Recursive Tree Calculations: Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

Three types:

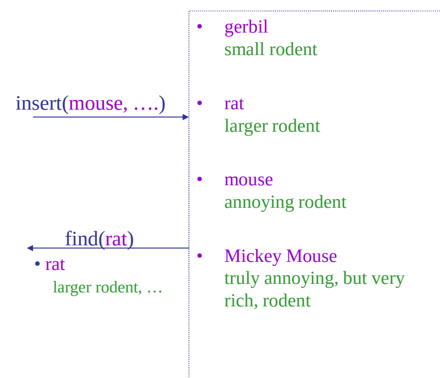
- Pre-order: Root, left subtree, right subtree
- In-order: Left subtree, root, right subtree
- Post-order: Left subtree, right subtree, root



(an expression tree)

The Dictionary ADT

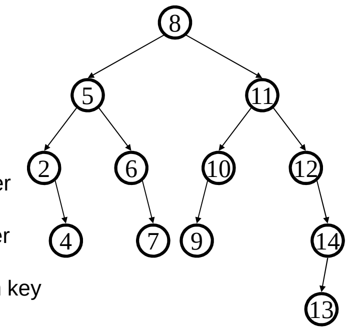
- Data:
  - a set of (key, value) pairs
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)



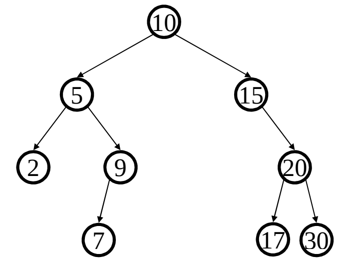
*The Dictionary ADT is sometimes called the "**Map ADT**"*

Binary Search Tree Data Structure

- Structural property
  - each node has ≤ 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small
- Order property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result: easy to find any given key
- What must I know about what I store?



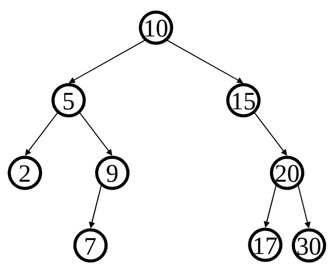
Find in BST, Recursive




```
Node Find(Object key,
                Node root) {
  if (root == NULL)
    return NULL;

  if (key < root.key)
    return Find(key,

   root.left);
  else if (key > root.key)
    return Find(key,

   root.right);
  else
    return root;
}
```

Runtime:

Insert in BST

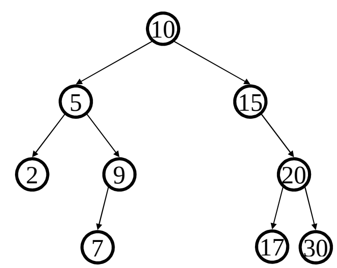


Insert(13)
Insert(8)
Insert(31)

Insertions happen only at the leaves – easy!

Runtime:

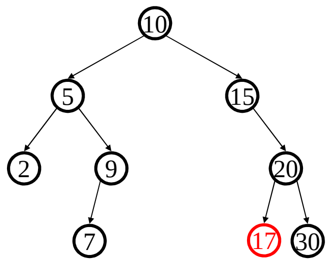
Deletion in BST



Why might deletion be harder than insertion?

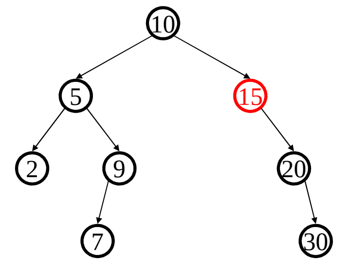
Non-lazy Deletion – The Leaf Case

Delete(17)



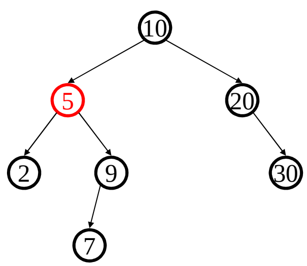
Deletion – The One Child Case

Delete(15)



Deletion – The Two Child Case

Delete(5)



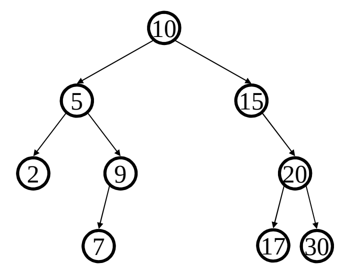
What can we replace 5 with?

Lazy Deletion

Instead of physically deleting nodes, just mark them as deleted

+ simpler
+ physical deletions done in batches
+ some adds just flip deleted flag

– extra memory for deleted flag
– many lazy deletions slow finds
– some operations may have to be modified (e.g., min and max)



Balanced BST

<u>Observation</u>

- BST: the shallower the better!
- For a BST with *n* nodes
  - Average height is O(log *n*)
  - Worst case height is O(*n*)
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

<u>Solution</u>: Require a **Balance Condition** that

1. ensures depth is O(log *n*) – strong enough!
2. is easy to maintain – not too strong!

The AVL Balance Condition

Left and right subtrees of *every node* have equal *heights* **differing by at most 1**

Define: **balance**(*x*) = height(*x*.left) – height(*x*.right)

AVL property: **–1 ≤ balance(*x*) ≤ 1, for every node *x***

- Ensures small depth
  - Will prove this by showing that an AVL tree of height *h* must have a lot of (i.e. O(2^h)) nodes
- Easy to maintain
  - Using single and double rotations

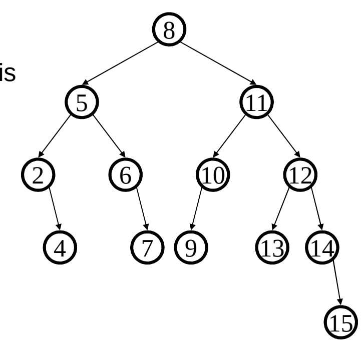
The AVL Tree Data Structure

<u>Structural properties</u>

1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

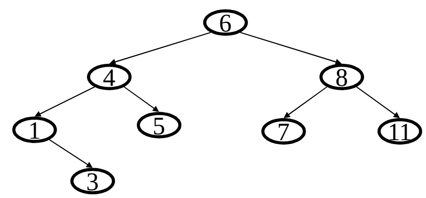
Result:

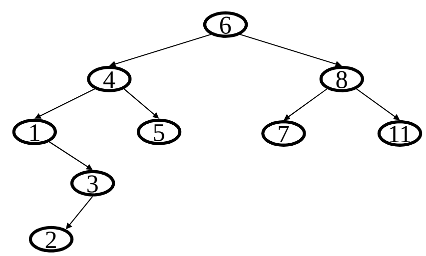
Worst case depth is O(log *n*)



<u>Ordering property</u>

– Same as for BST





AVL tree insert

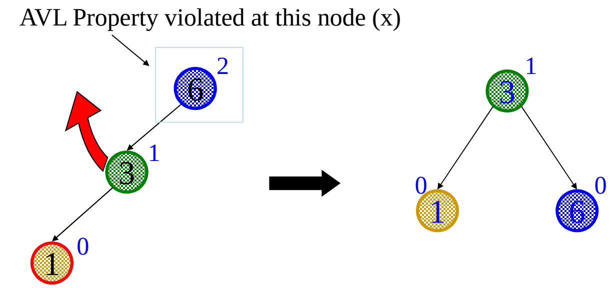
Let *x* be the node where an imbalance occurs.

Four cases to consider. The insertion is in the

1. left subtree of the left child of *x.*
2. right subtree of the left child of *x.*
3. left subtree of the right child of *x.*
4. right subtree of the right child of *x.*

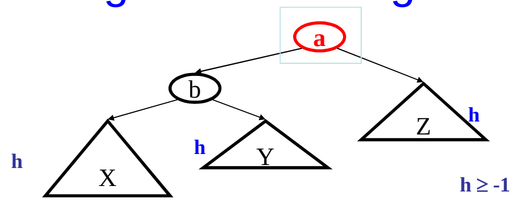
Idea: Cases 1 & 4 are solved by a single rotation.
Cases 2 & 3 are solved by a double rotation.

Fix: Apply Single Rotation

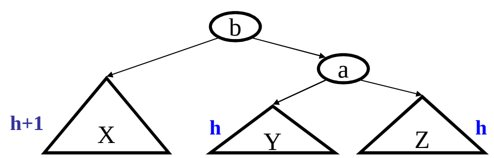


Single Rotation:
1. Rotate between x and child

Single rotation in general

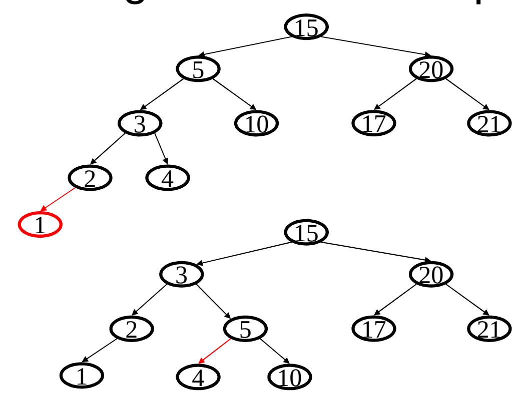


X < b < Y < a < Z



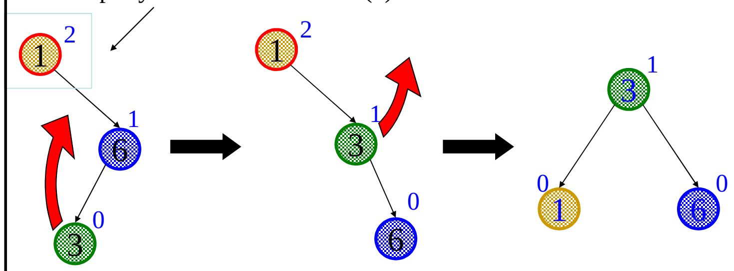
Height of tree before? Height of tree after? Effect on Ancestors?

Single rotation example



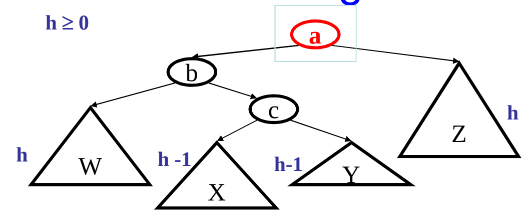
Fix: Apply Double Rotation

AVL Property violated at this node (x)

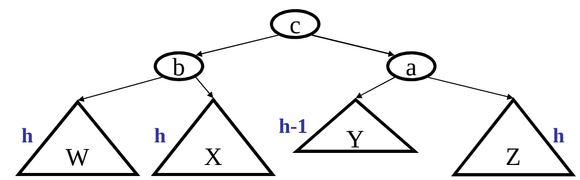


Double Rotation
1. Rotate between x's child and grandchild
2. Rotate between x and x's new child

Double rotation in general

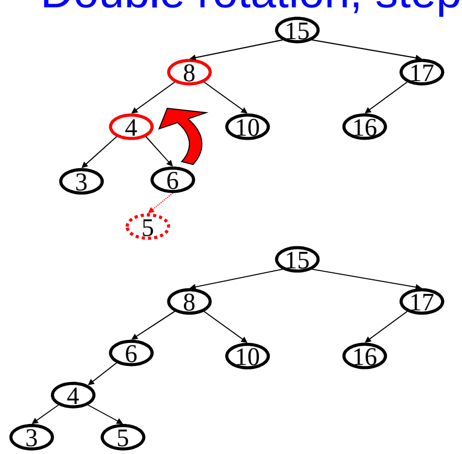


W < b <X < c < Y < a < Z

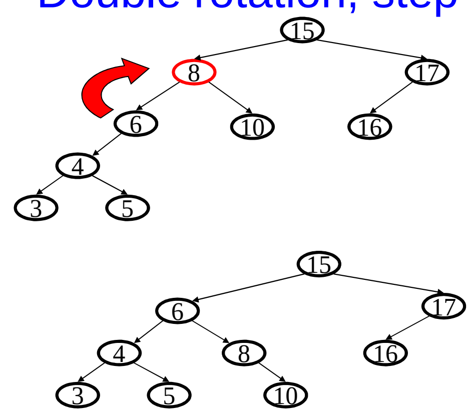


Height of tree before? Height of tree after? Effect on Ancestors?

Double rotation, step 1

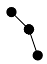


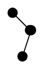
Double rotation, step 2



Insertion into AVL tree

1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:


case #1: Perform single rotation and exit


case #2: Perform double rotation and exit

Both rotations keep the subtree height unchanged.
Hence only one (sinlge or double) rotation is sufficient!