Unit #7: AVL Trees

CPSC 221: Algorithms and Data Structures

Lars Kotthoff[1]
larsko@cs.ubc.ca

[1] With material from Will Evans, Steve Wolfman, Alan Hu, Ed Knorr, and Kim Voll.

Unit Outline

- Binary search trees
- Balance implies shallow (shallow is good)
- How to achieve balance
- Single and double rotations
- AVL tree implementation

Learning Goals

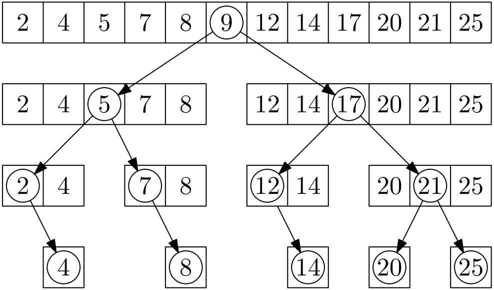
- Compare and contrast balanced/unbalanced trees.
- Describe and apply rotation to a BST to achieve a balanced tree.
- Recognize balanced binary search trees (among other tree types you recognize, e.g., heaps, general binary trees, general BSTs).

Dictionary ADT Implementations

Worst Case Runtime:

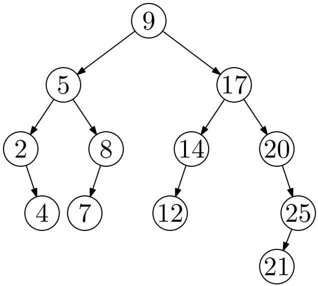
	insert	find	delete (after find)
Linked list			
Unsorted array			
Sorted array			
Hash table			

Binary Search in a Sorted List




```
int bSearch(int A[], int key, int i, int j) {
  if(j < i) return -1;
  int m = (i + j) / 2;
  if(key < A[m])
    return bSearch(A, key, i, m-1);
  else if(key > A[m])
    return bSearch(A, key, m+1, j);
  else return m;
}
```

Binary Search Tree as Dictionary Data Structure



Binary tree property

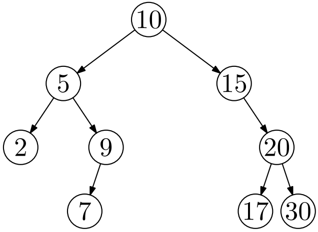
- each node has ≤ 2 children

Search tree property

- all keys in left subtree smaller than node's key
- all keys in right subtree larger than node's key

Result: easy to find any given key

In-, Pre-, Post-Order Traversal



In-order: 2, 5, 7, 9, 10, 15, 17, 20, 30

Pre-order:

Post-order:

Beauty is Only $O(\log n)$ Deep

Binary Search Trees are fast if they're shallow.
Know any shallow trees?

- perfectly complete
- almost complete (except the last level, like a heap)
- anything else?

Beauty is Only $O(\log n)$ Deep

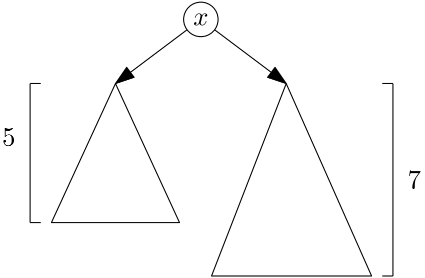
Binary Search Trees are fast if they're shallow.
Know any shallow trees?

- perfectly complete
- almost complete (except the last level, like a heap)
- anything else?

What matters here?

Siblings should have about the same height.

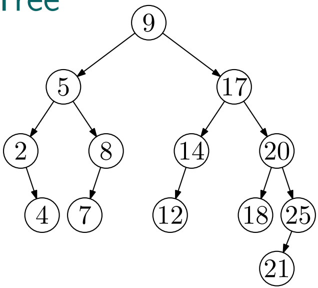
Balance



balance(x) = height(x.left) − height(x.right)
height(NULL) = −1.
If for all nodes x,

- balance(x) = 0 then perfectly balanced.
- |balance(x)| is small then balanced enough.
- $-1 \leq$ balance(x) ≤ 1 then tree height $\leq c \lg n$ where $c < 2$.

AVL (Adelson-Velsky and Landis) Tree



Binary tree property

- each node has $\leq$ 2 children

Search tree property

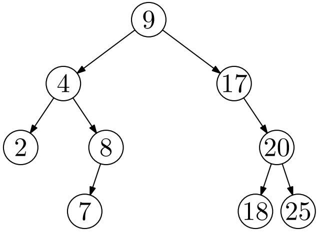
- all keys in left subtree smaller than node's key
- all keys in right subtree larger than node's key

Balance property

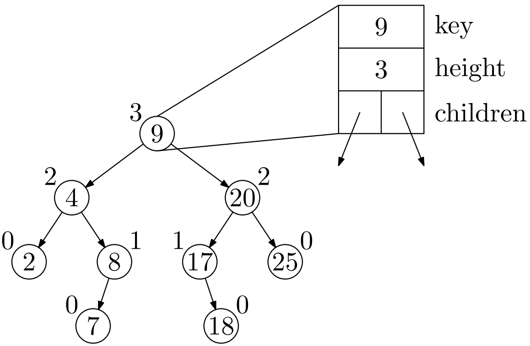
- For all nodes x, $-1 \leq$ balance(x) ≤ 1

Result: height is $\Theta(\log n)$.

Is this an AVL tree?

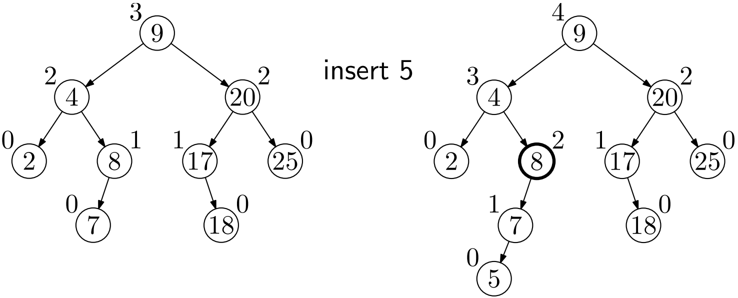


An AVL Tree



How Do We Stay Balanced?

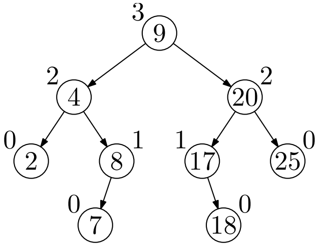
Suppose we start with a balanced search tree (an AVL tree).



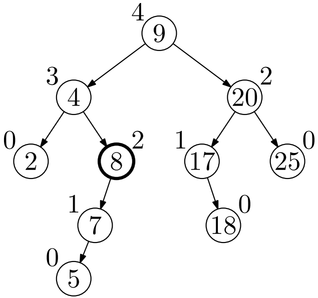
It's no longer an AVL tree. What can we do?

How Do We Stay Balanced?

Suppose we start with a balanced search tree (an AVL tree).



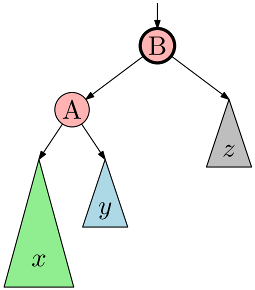
insert 5



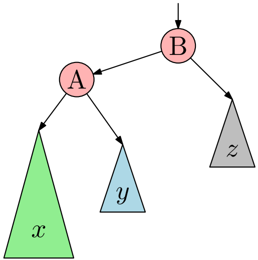
It's no longer an AVL tree. What can we do?

ROTATE!

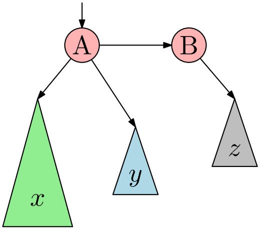
Rotation



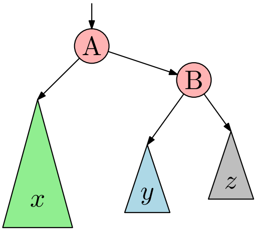
Rotation



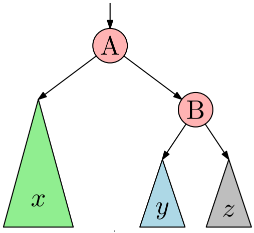
Rotation



Rotation



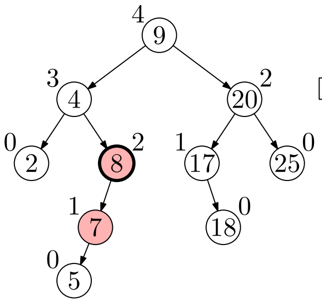
Rotation



Time Complexity of Rotation

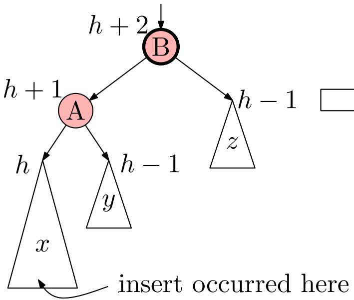
- $\mathsf{O}(1)$
- $\mathsf{O}(\lg n)$
- $\mathsf{O}(n)$
- $\mathsf{O}(n \lg n)$
- $\mathsf{O}(n^2)$
- none of the above

Single Rotation



Single Rotation

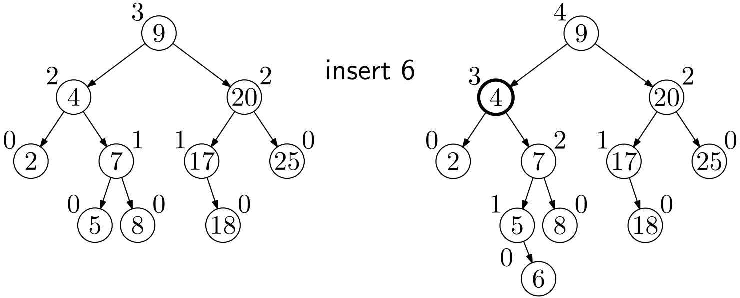
`rotateRight` is shown. There's also a symmetric `rotateLeft`.



After rotation, subtree's height is the same as before insert.
So heights of ancestors don't change.

Double Rotation

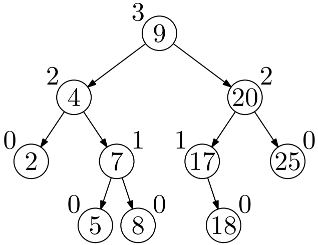
Start with



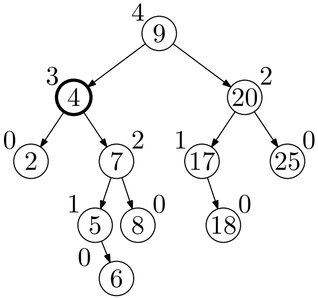
A single rotation won't fix this.

Double Rotation

Start with



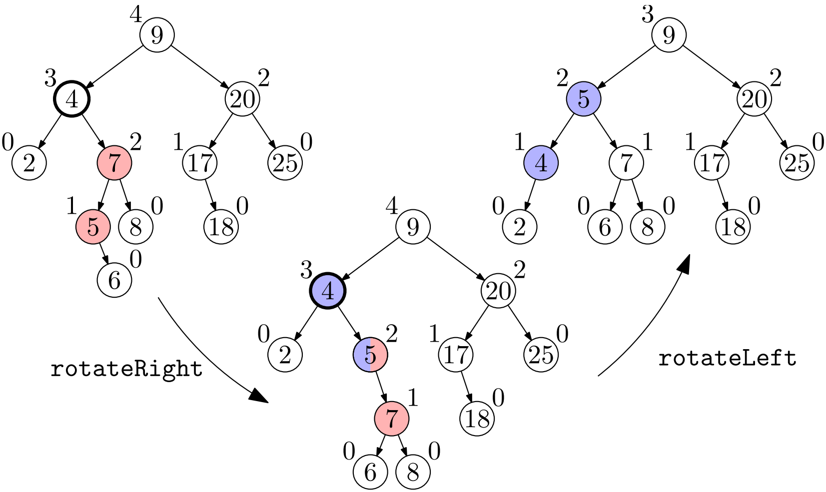
insert 6



A single rotation won't fix this.

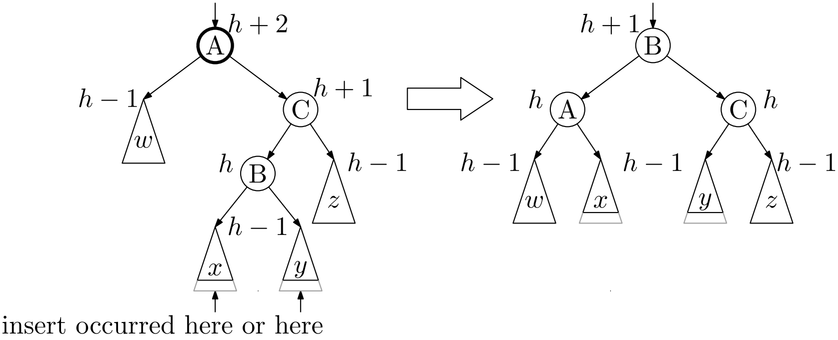
DOUBLE ROTATE!

Double Rotation



Double Rotation

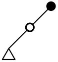
`doubleRotateLeft` is shown. There's also a symmetric `doubleRotateRight`.

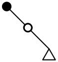


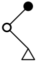
Either x or y increased to height $h-1$ after insert.
After rotation, subtree's height is the same as before insert.
So height of ancestors doesn't change.

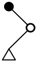
Insert Algorithm

1. Find location for new key.
2. Add new leaf node with new key.
3. Go up tree from new leaf searching for imbalance.
4. At lowest unbalanced ancestor:

   Case LL:  `rotateRight`

   Case RR:  `rotateLeft`

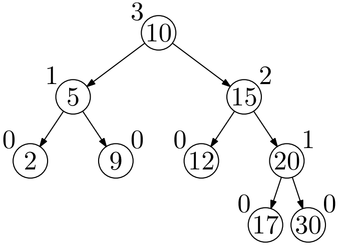
   Case LR:  `doubleRotateRight`

   Case RL:  `doubleRotateLeft`

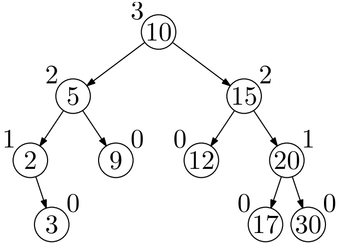
   The case names are the first two steps on the path from the unbalanced ancestor to the new leaf.

Insert: No Imbalance

Insert(3)

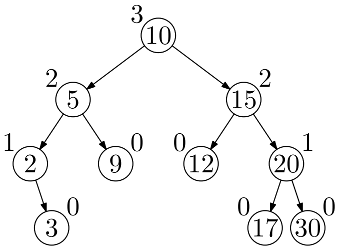


Insert: No Imbalance

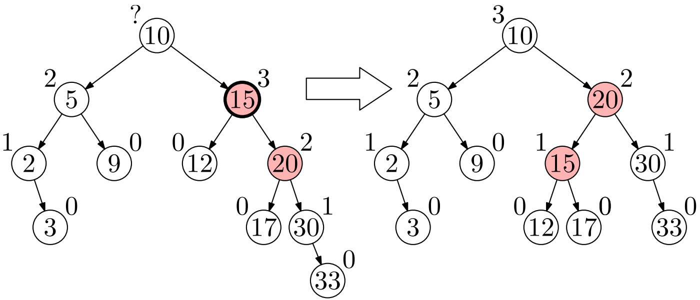


Insert: Imbalance Case RR

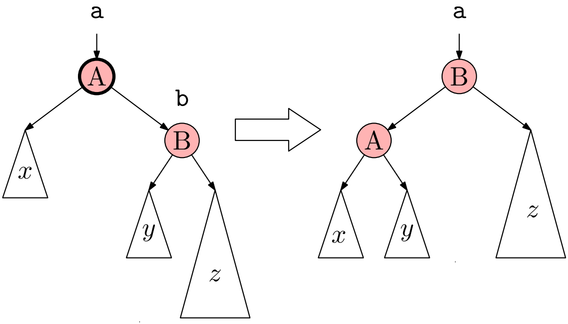
Insert(33)



Case RR: `rotateLeft`



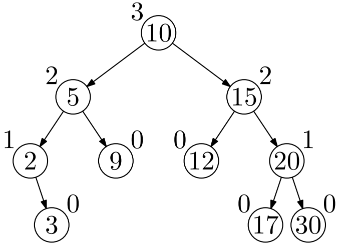
Single Rotation Code



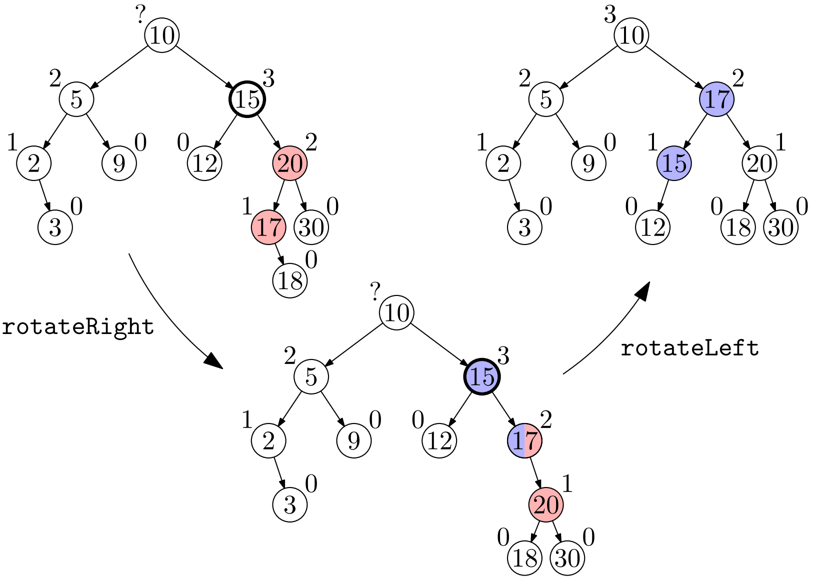

```
void rotateLeft(Node *&a) {
  Node* b = a->right;
  a->right = b->left;
  b->left = a;
  updateHeight(a);
  updateHeight(b);
  a = b;
}
```

Insert: Imbalance Case RL

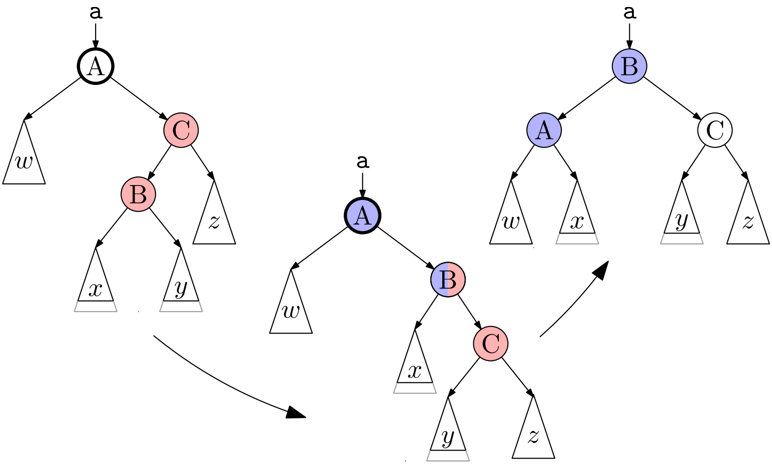
Insert(18)



Case RL: `doubleRotateLeft`



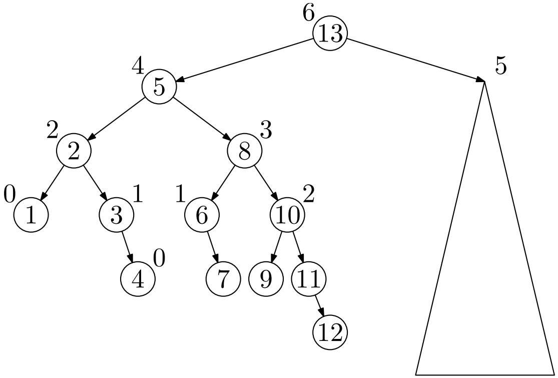
Double Rotation Code




```
void doubleRotateLeft(Node *&a) {
  rotateRight(a->right);
  rotateLeft(a);
}
```

Delete

1. Delete as for general binary search tree. (This way we reduce the problem to deleting a node with 0 or 1 child.)
2. Go up tree from deleted node searching for imbalance (and fixing heights).
3. Fix **all** unbalanced ancestors (bottom-up).



Thinking about AVL trees

Observations

- Binary search trees that allow only slight imbalance.
- Worst-case $O(\log n)$ time for find, insert, and delete.
- Elements (even siblings) may be scattered in memory.

Realities

- For large data sets, disk accesses dominate runtime.

Could we have perfect balance if we relax binary tree restriction?