Tuesday, September 29, 2009
3:29 PM

AVL Trees

- Last time someone asked how to keep a binary tree balanced?
- Excellent question, there are several approaches (B-Trees, Splay Trees, Red-Black Trees ---)
- We look at AVL Trees as the balancing operations are pretty easy.
- AVL Tree property: basic binary search tree, except that for every node in the tree, height of left & right subtrees differ by at most 1.

- Keeping track of node height:
  - each node maintains balance info (right subtree height − left subtree height) and is considered balanced if diff is −1, 0, 1 (see wikipedia entry)

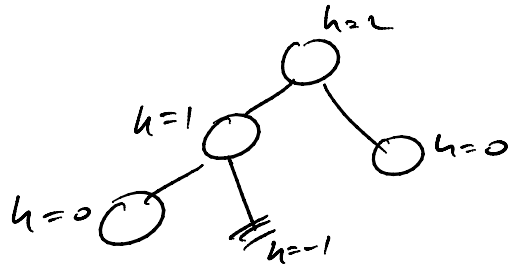
    OR

  - you can calculate node balance from subtree heights
- the textbook uses a sort of hybrid approach:
  - stores height of node
  - & calculates subtree height diff as need

- ξ calculates shear height diff as well
(I think keeping/maintaining balance w/o is easier)

Tuesday, September 29, 2009
3:40 PM

- height of a NULL node (empty subtree) is −1
- Leaf node initialized height = 0



is this an AVL tree?
Yes.

- height of an interior node = max(height(t→left), height(t→right)) + 1
- main problems are insertion & deletion
  (left as an exercise for reader)
- when inserting, need to update balancing info of each node
  an aside: in the alternate scheme (balance kept at node, not height as in the book),
  if height of right subtree increases,
  bal++, if right subtree height decreases bal--
  (this differs from text)

on the path back up to the root

(remember that basic insertion is just to add leaf in proper place)

```
insert(const T& x, Node*& t)
{
    if(!t) t = new Node(x, NULL, NULL)
    else if(x < t->data) insert(x, t->left)
```

else if (x > t->data) insert (x, t->right)

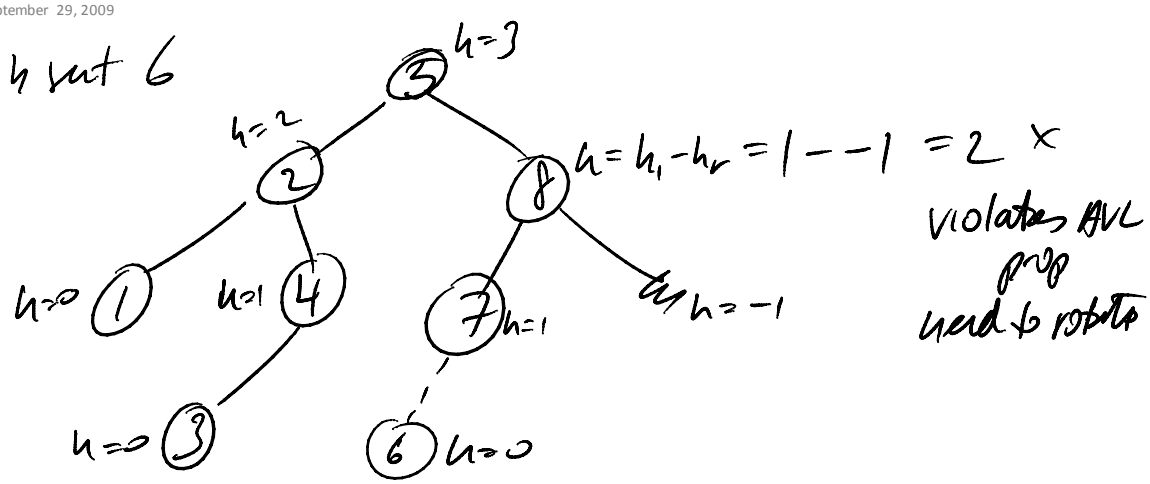
else; // no op

}

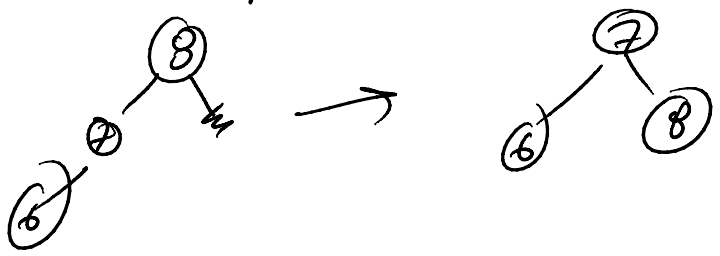
- the book assumes lazy deletion (hard deletion, no rebalancy done until another insert)

Tuesday, September 29, 2009
3:52 PM

insert 6

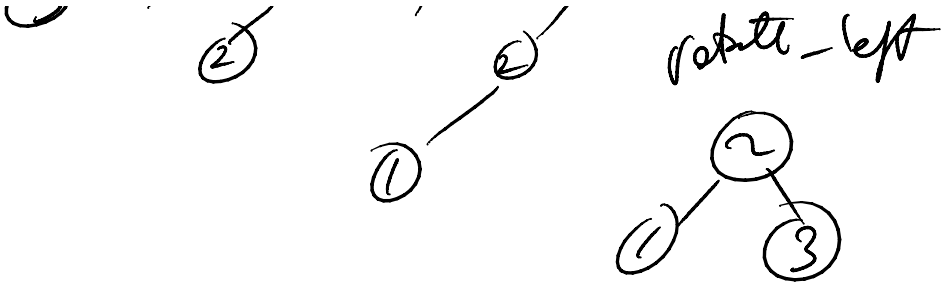


- here, need to "rotate_left" — rotate to the right with left child

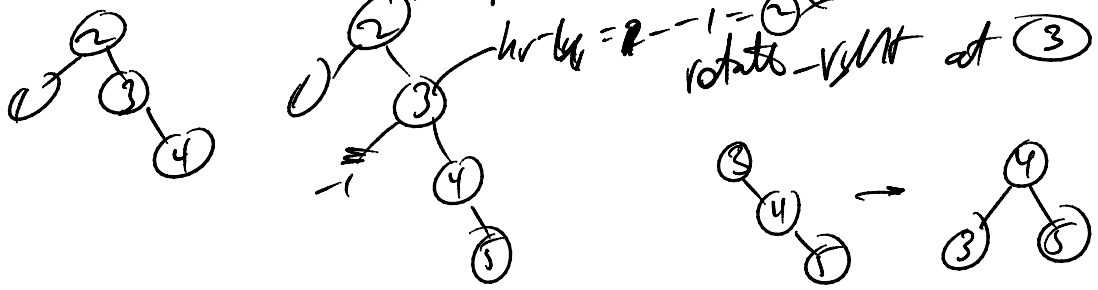


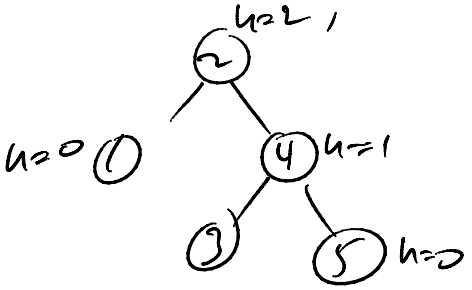
- as you recurse back up the tree need to check AVL property for violations at each node
- only 4 possible cases for imbalance to occur:
  - case 1 & 4: insertion into left subtree of left child
    insertion into right " " right "
    (insertion occurs "outside")
- these imbalances are fixed by single rotation at the node where violation occurs

insert 3, 2, 1:

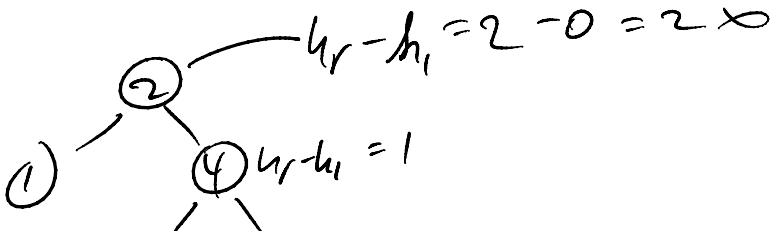


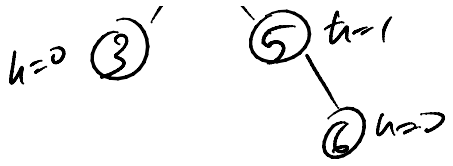
insert 4,5:

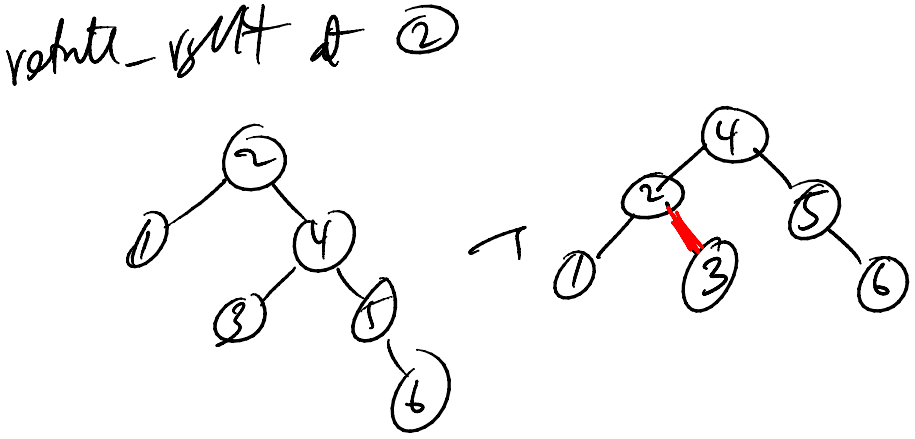


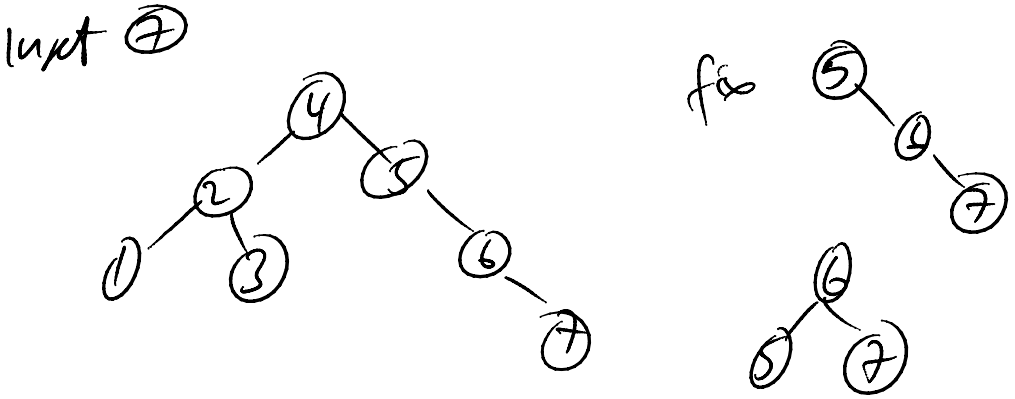


insert 6:




h=0
3
5
h=1
2
h=0

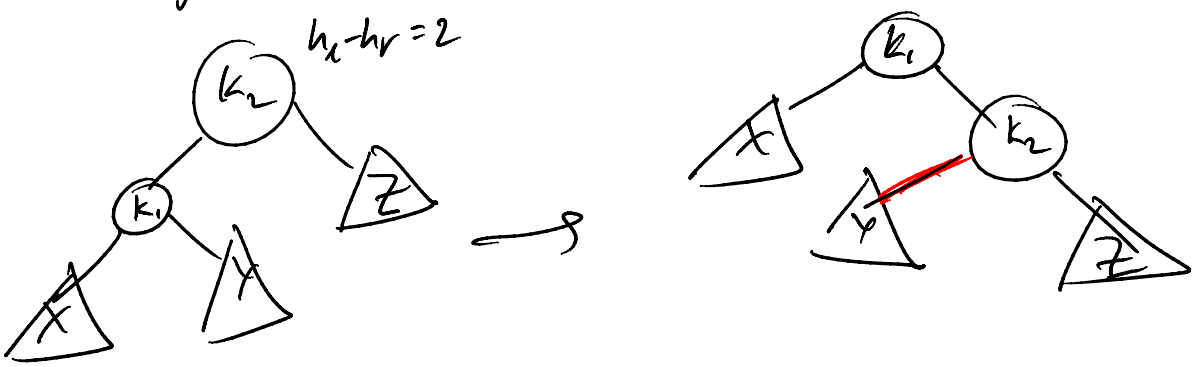

root_left
2
1
4
3
5
6
4
2
5
1
3
6


insert 7
4
2
5
1
3
6
7

5
0
7
6
5
7

Tuesday, September 29, 2009
4:06 PM

In general, we have single rotation (rotate_left)



just inserted here, k_2 out of balance

given Node* k_2,

let $k_1 = k_2 \rightarrow left$

$k_2 \rightarrow left = k_1 \rightarrow right$

$k_1 \rightarrow right = k_2$

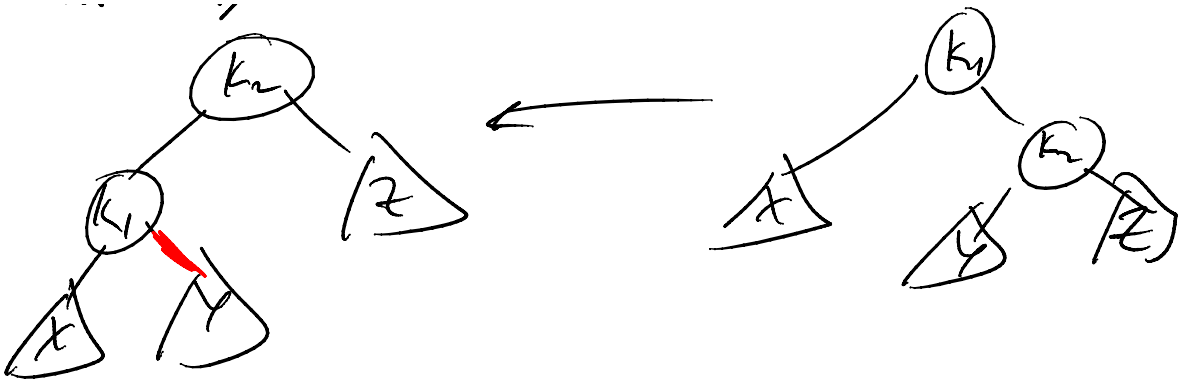
update height

$k_2 \rightarrow height = max(height(k_2 \rightarrow left), height(k_2 \rightarrow right)) + 1$

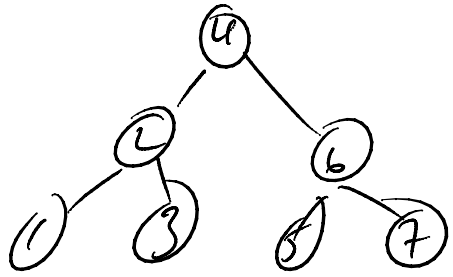
$k_1 \rightarrow height = max(height(k_1 \rightarrow left), k_2 \rightarrow height) + 1$

$k_2 = k_1$ // need this last bit to reset root pointer (k_2) coming in as argument

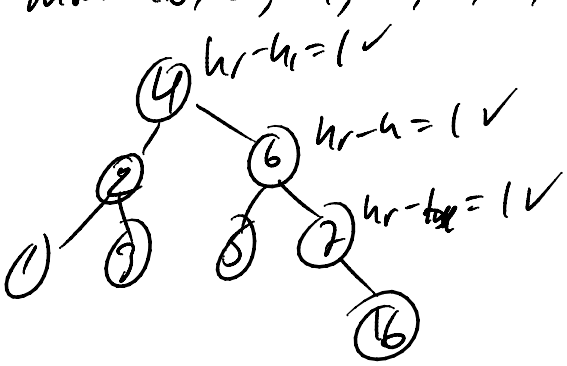
– rotate_right:

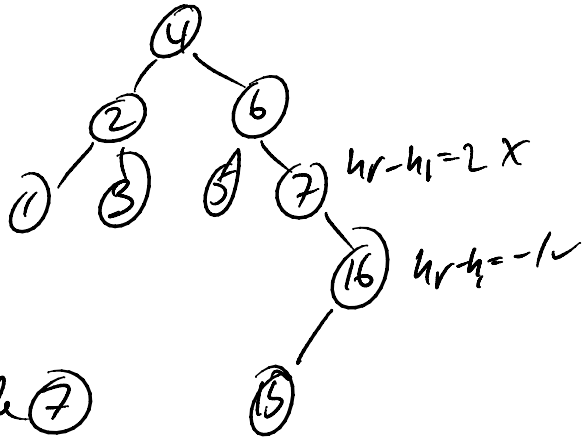

k_2
k_1
Z
X
Y
k_1
X
k_2
Y
Z

Double rotation — handles "interior" cases

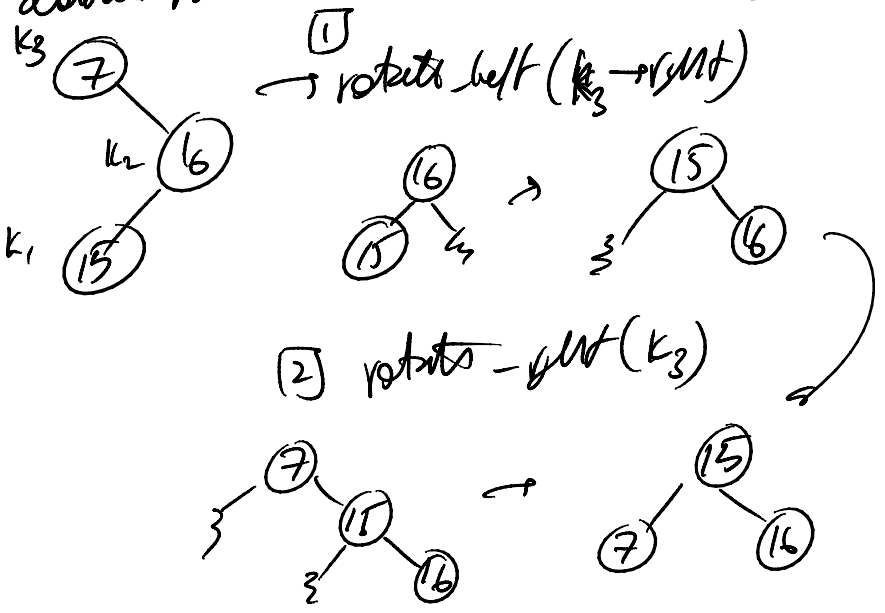


insert 16, 15, 14, 13, 12, 11, 10, 8, 9

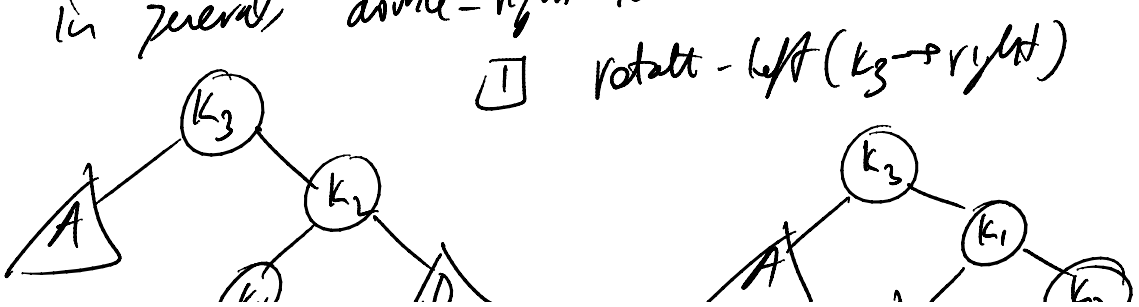




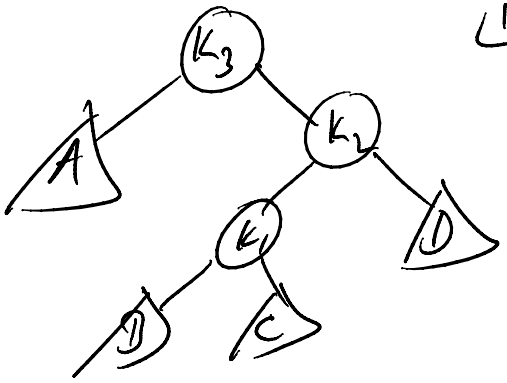
need double rotation at node 7

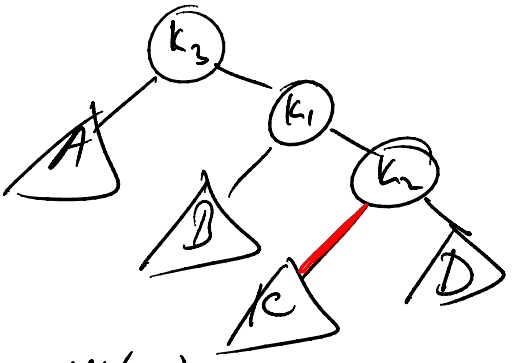


in general, double - right rotation:

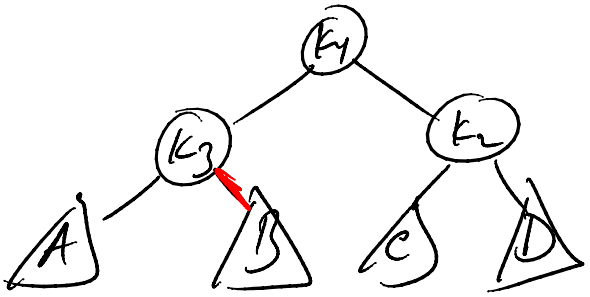


[1] rotate_right(k_3)



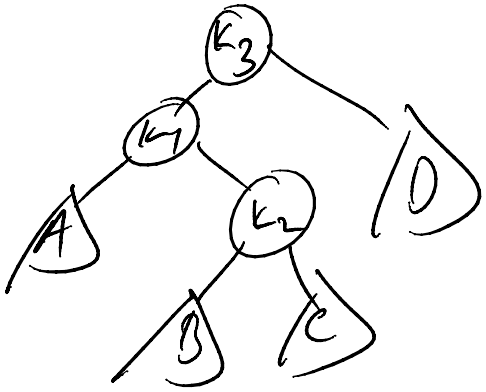


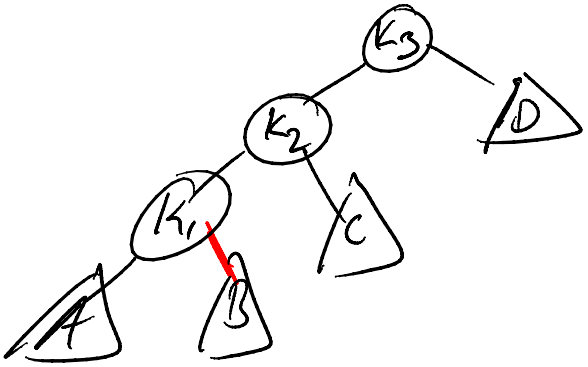
[2] rotate_right(k_3):



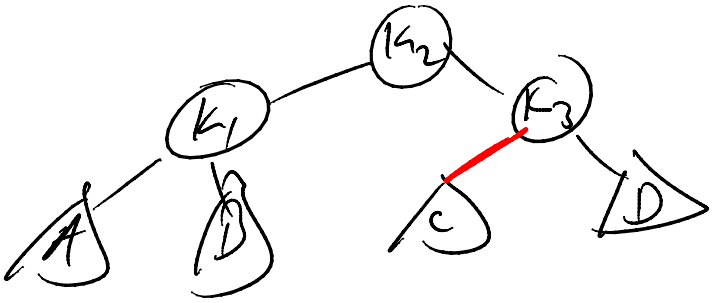
double_left rotation:

[1] rotate_right(k_3 → left)





[2] rotate_left(k_3)



Insert 14, 15, ...

Tuesday, September 29, 2009
4:25 PM

- to implement, need to redefine Node ADT with height

```
struct Node {
    T data
    Node* left
    Node* right
    int height
    Node(const T& d, Node *lt, Node *rt, int h=0)
      : data(d), left(lt), right(rt), height(h);
}
```

Tuesday, September 29, 2009
4:27 PM

```
insert(const T& x, Node*& t)
{
  if (t == NULL) t = new Node(NULL, NULL);
  else if (x < t->data) {
      insert(x, t->left);
      if (height(t->left) - height(t->right) == 2)
          if (x < t->left->data)
              rotate_left(t)   // outer
          else
              double_left(t)   // inner
  } else if (x > t->data) {
      insert(x, t->right);
      if (height(t->right) - height(t->left) == 2)
          if (x > t->right->data)
              rotate_right(t)
          else
              double_right(t)
  }
  else ; // no-op
  t->height = max(height(t->left), height(t->right)) + 1
```