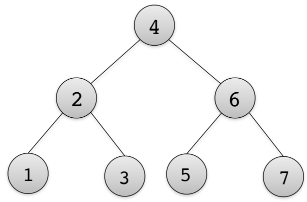
Estruturas de Dados / Programação 2
Árvore Balanceada AVL

Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

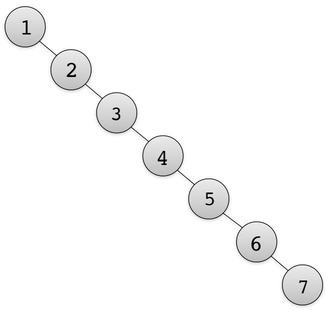
Introduction

- Create a Binary Search Tree with the items 4, 2, 6, 1, 5, 3, and 7 in this order!



Introduction

- Do it again, but now in this order: 1, 2, 3, 4, 5, 6, and 7

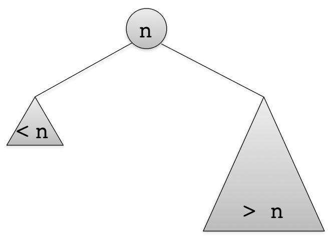


Different orders to…

- Insert nodes
- Remove nodes
- In practice, we cannot predict this order!

This may lead to Unbalanced Trees!

Intuition



We need to fix this problem

- Otherwise, we can face efficiency problems!
- Which concept we can use to help on this problem?

~~Path~~ ~~Root~~ ~~Descendant~~

Height

~~Leaf~~ ~~Sibling~~

~~Internal node~~

~~Edge~~

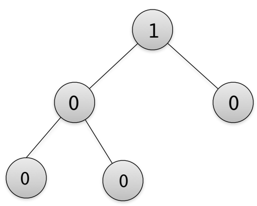
AVL Trees

AVL Trees

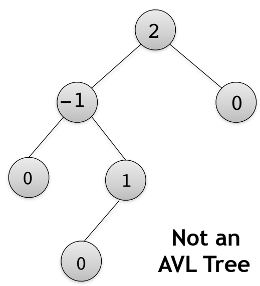
- Algorithms published in 1962 by G. M. **A**del'son-**V**elskii and Y. M. **L**andis
- In their honor, the elements of this data structure are called AVL trees
- AVL is a Self-balancing Binary Search Tree in which the maximum difference in the height of any node's right and left subtrees is 1

Balance Factor

`bf = h(leftSubtree) – h(rightSubtree)`

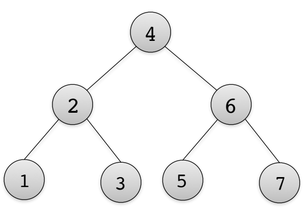


AVL Tree

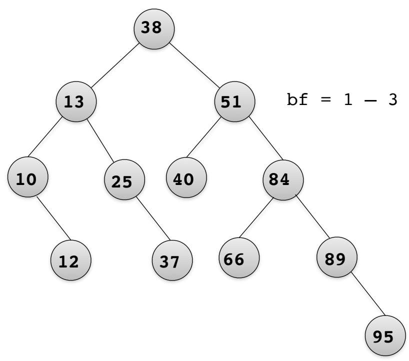


Not an AVL Tree

Is the following tree an AVL one?

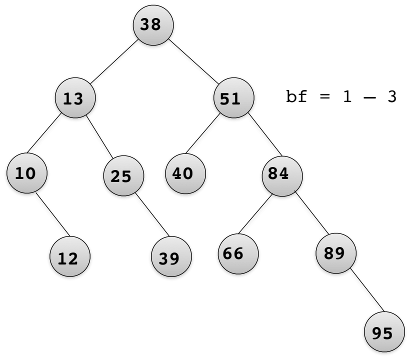


What about this one?



`bf = 1 – 3 = -2`

And this one?

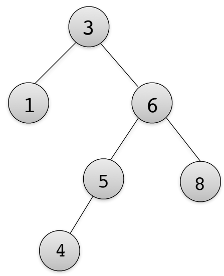


bf = 1 – 3 = -2

Balance factor function

```
int is_balanced(binary_tree *bt)
{
  int bf = h(bt->left) – h(bt->right);
  return ((-1 <= bf) && (bf <= 1));
}
```

What about the h function?




```
int max(int a, int b)
{
  return (a > b) ? a : b;
}

int h(binary_tree *bt)
{
  if (bt == NULL) {
    return -1;
  } else {
    return 1 + max(h(bt->left), h(bt->right));
  }
}
```

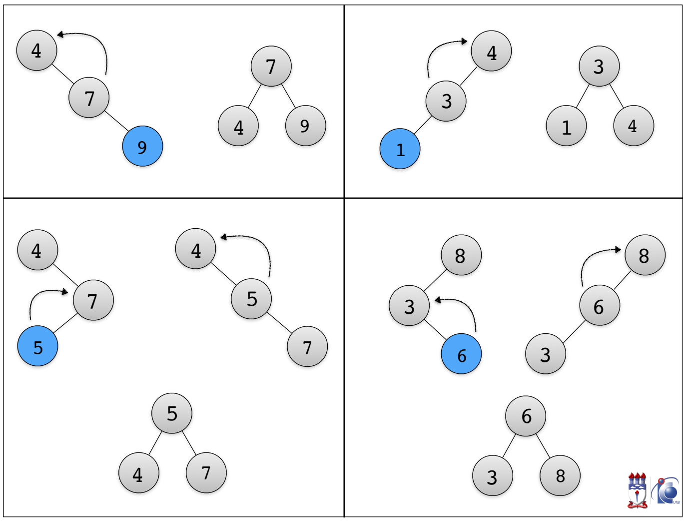
When not balanced…

Rotations!

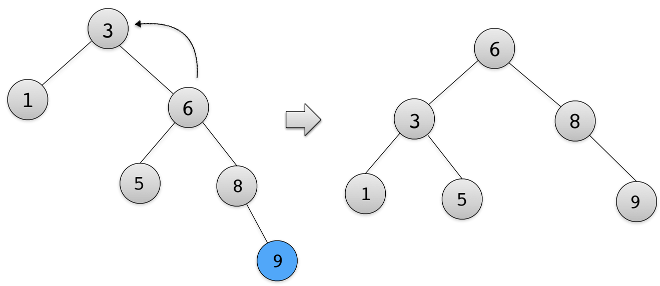
Four Cases

Adding nodes...

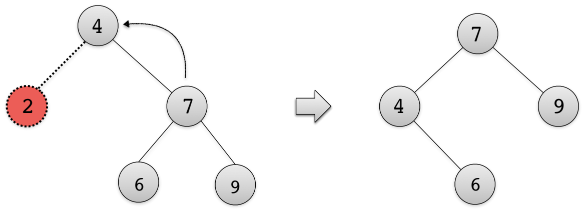
... and keeping balanced



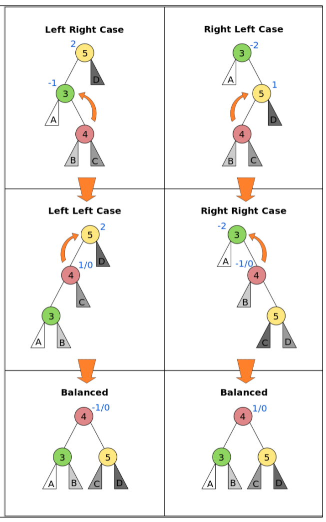
Before adding: `bf = -1`
After adding: `bf = -2`



Before removing: `bf = -1`
After removing: `bf = -2`



```
if (balance_factor(bt) == 2) {

  child = bt->left;

  if (balance_factor(child) == -1) {
    bt->left = rotate_left(child);
  }

  bt = rotate_right(bt);

} else if (balance_factor(bt) == -2) {

  child = bt->right;

  if (balance_factor(child) == 1) {
    bt->right = rotate_right(child);
  }

  bt = rotate_left(bt);
}
```



From *http://en.wikipedia.org/wiki/AVL_tree*

Abstract Data Type: AVL

AVL ADT

```c
binary_tree* create_empty_binary_tree();

binary_tree* create_binary_tree(
    int item, binary_tree *left, binary_tree *right);

binary_tree* add(binary_tree *bt, int item);

int is_empty(binary_tree *bt);

int h(binary_tree *bt);

int balance_factor(binary_tree *bt);

binary_tree* rotate_left(binary_tree *bt);

binary_tree* rotate_right(binary_tree *bt);
```

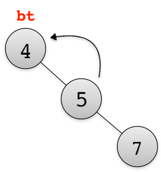
Struct

```c
struct binary_tree {
    int item;
    int h;
    binary_tree* left;
    binary_tree* right;
};
```

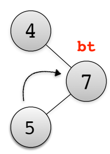
```c
int balance_factor(binary_tree *bt)
{
  if (bt == NULL) {
    return 0;
  } else if ((bt->left != NULL) && (bt->right != NULL)) {
    return (bt->left->h - bt->right->h);
  } else if ((bt->left != NULL) && (bt->right == NULL)) {
    return (1 + bt->left->h);
  } else {
    return (-bt->right->h - 1);
  }
}
```

```c
binary_tree* add(binary_tree *bt, int item)
{
  if (bt == NULL) {
    return create_binary_tree(item, NULL, NULL);
  } else if (bt->item > item) {
    bt->left = add(bt->left, item);
  } else {
    bt->right = add(bt->right, item);
  }

  bt->h = h(bt);
  binary_tree *child;

  if (balance_factor(bt) == 2 || balance_factor(bt) == -2) {
    if (balance_factor(bt) == 2) {
      child = bt->left;
      if (balance_factor(child) == -1) {
        bt->left = rotate_left(child);
      }
      bt = rotate_right(bt);
    } else if (balance_factor(bt) == -2) {
      child = bt->right;
      if (balance_factor(child) == 1) {
        bt->right = rotate_right(child);
      }
      bt = rotate_left(bt);
    }
  }
  return bt;
}
```

```c
binary_tree* rotate_left(binary_tree *bt)
{
  binary_tree *subtree_root = NULL;

  if (bt != NULL && bt->right != NULL) {
    subtree_root = bt->right;
    bt->right = subtree_root->left;
    subtree_root->left = bt;
  }

  subtree_root->h = h(subtree_root);
  bt->h = h(bt);
  return subtree_root;
}
```

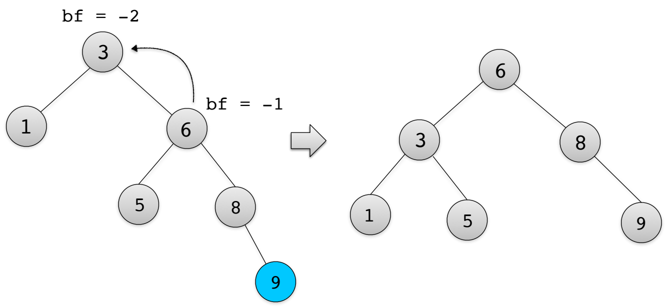



```c
binary_tree* rotate_right(binary_tree *bt)
{
  binary_tree *subtree_root = NULL;

  if (bt != NULL && bt->left != NULL) {
    subtree_root = bt->left;
    bt->left = subtree_root->right;
    subtree_root->right = bt;
  }

  subtree_root->h = h(subtree_root);
  bt->h = h(bt);
  return subtree_root;
}
```

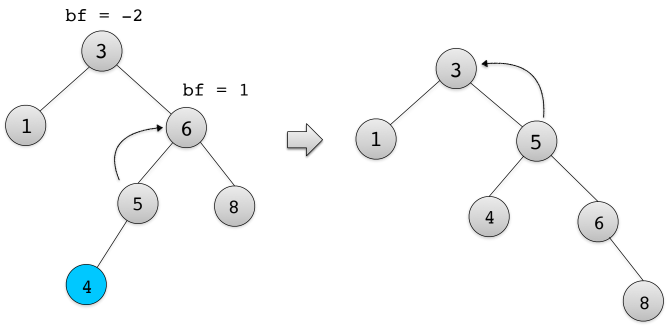


Examples

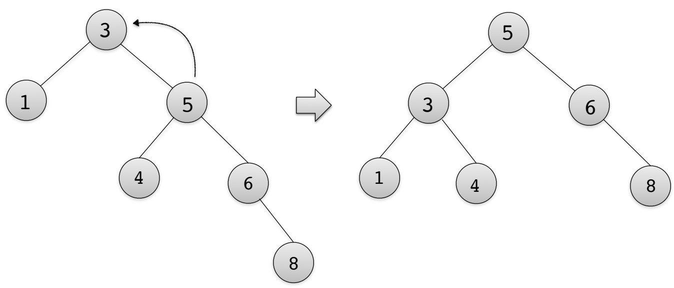
Right-Right: only one step



Right-Left: two steps. Here is the first...



and now the second...



Poscomp 2009

27 - Suponha que T seja uma árvore AVL inicialmente vazia, e considere a inserção dos elementos $10, 20, 30, 5, 15, 2$ em T, nesta ordem. Qual das seqüências abaixo corresponde a um percurso de T em pré-ordem:

(a) $10, 5, 2, 20, 15, 30$

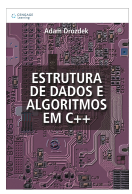
(b) $20, 10, 5, 2, 15, 30$

(c) $2, 5, 10, 15, 20, 30$

(d) $30, 20, 15, 10, 5, 2$

(e) $15, 10, 5, 2, 20, 30$

References



Chapter 6