CSCA48 WINTER 2015

WEEK 11 - BALANCED TREES

Anna Bretscher

March 23, 25/26, 2015

Why Balanced Trees?

- What is the worst case complexity of `insert, delete, search` in a binary search tree?
- O(n)
- We need to do something better...
  - AVL trees
  - B-trees
  - Splay trees

AVL Trees

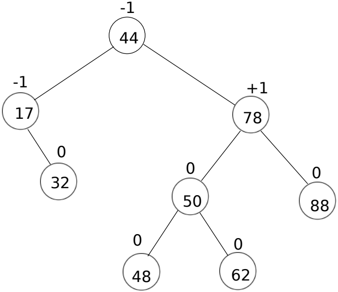
AVL Trees were invented by Adelson-Velskii and Landis in 1962.
An AVL tree is similar to a *BST* in that it

- stores values in the *internal nodes* and
- has a property *relating* the values stored in a *subtree* to the values in the *parent node*.

  but different from a *BST* because

- The height of an *AVL* tree is $O(\log n)$.
- Each *internal node* has a *balance property* equal to -1, 0, 1.
- Balance value = *height* of the *left* subtree - *height* of the *right* subtree.

AVL TREES

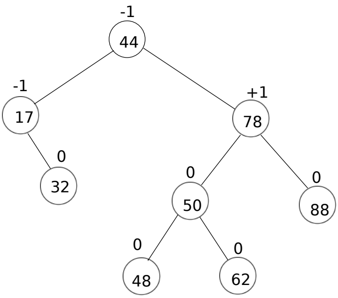


Q. What is the purpose of the *balance* property?

A. Ensures that the height is always a function of $\log n$.

Q. What information will we need to store in order to update the *balance factors* easily?

A. The *height* of the tree rooted at each node.



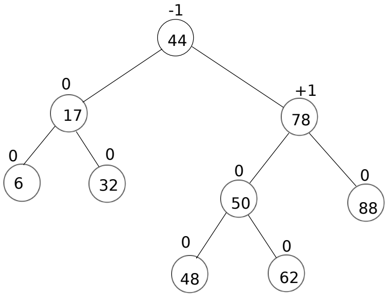
Searching in an AVL tree is the *same* as a BST.

Consider *inserting* 6 into the tree above.

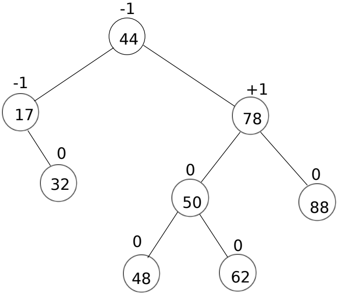
Q: What are the new *balance factors* in the tree after inserting 6?

A: 17 has value 0

AVL Insertion

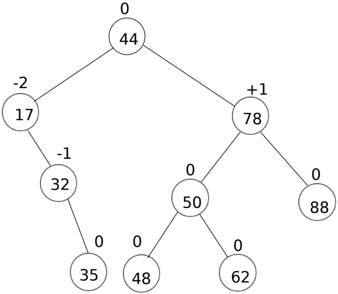


Q. Let's insert 35. What are the *balance factors* now?



AVL INSERTION

Q. Let's insert 35. What are the *balance factors* now?



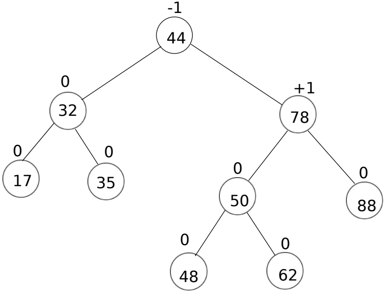
A. 32 has -1, 17 has -2. 44 has 0.

Q. How do we solve this problem?

- We resolve the problem by doing a **single rotation**. How should we *rotate*?
- Counter clock-wise. 17 comes down, 32 moves up.

AVL Insertion

- We rotate counter clock-wise. 17 comes down, 32 moves up.

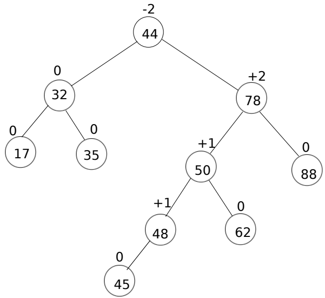


Q. How do we update the *balance factors*?

A. Update the heights first and then update balance factors.

AVL Insertion

Now let's insert *45*.



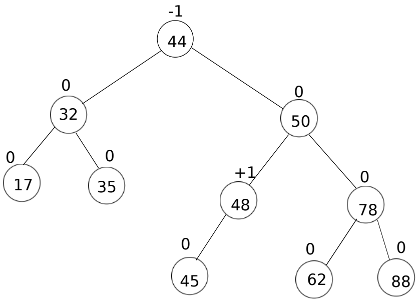
- Notice the *balance factors* now. How should we resolve the problem?

A. Do a *single rotation* clock-wise about the 78. 50 goes up, 78 down.

Q. What happens to the subtree rooted at 62?

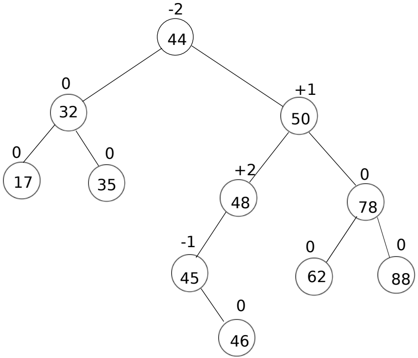
A. It becomes the left subtree of 78.

AVL INSERTION



- Notice the updated *balance factors*.
- Let's insert *46* this time.

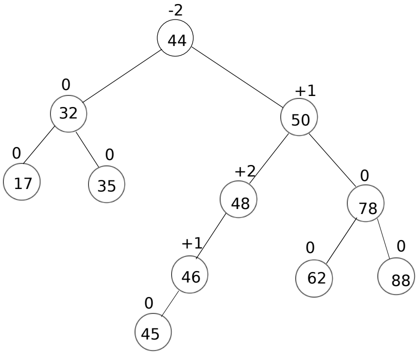
AVL Insertion



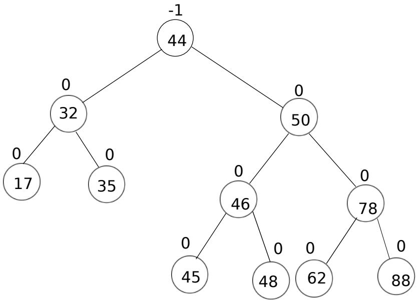
Q. Can we do a rotation about 48?

A. NO. Need a **double rotation**.

Double Rotation



Double Rotation



Delete

- If the key is a leaf node, delete and rebalance
- If the key is an internal node, replace with predecessor/successor and rebalance.

Complexity

- Since the tree is balanced the height of an *AVL* tree is O(log n).
- This means *insert*, *delete* and *search* are all O(log n).
- Searching for the location to *insert*/*delete*, takes O(log n).
- *Rebalancing* takes at most O(log n).

B-Trees

A *B-tree* of order *m* has the following properties:

- Internal nodes have at most *m* children (at most *m-1* keys)
- Internal nodes (except the root) have at most $\lceil \frac{m}{2} \rceil$ children
- A non-leaf node with *k* children has *k-1* keys
- All leaves are at the *same distance* from the root

B-TREES

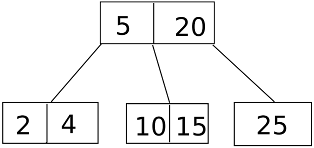
- If a node has keys $k_1, k_2, \ldots, k_{i-1}$ and children $c_1, c_2, \ldots, c_i$ where $\lceil \frac{m}{2} \rceil \leq i \leq m$ then $c_j < k_j$ and $c_i > k_{i-1}$.
- All operations are $O(h)$ where h is the height of the tree.
- $h \leq \lfloor log_{min}(\frac{n+1}{2}) + 1) \rfloor$ where min is the minimum number of elements in a node.

★ B-Trees are used in large file systems including those used by Mac OS/X, some Linux and Microsoft operating systems.

2-3 TREES

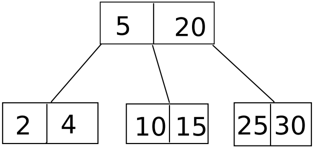
A *2-3 tree* is a *B-Tree* of order $m = 3$.

- Each node has at most 3 children and at least 2 children.
- Each node then has 1 or 2 keys.

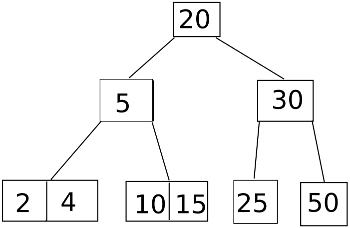


- Let's insert 30.

2-3 Tree Insert

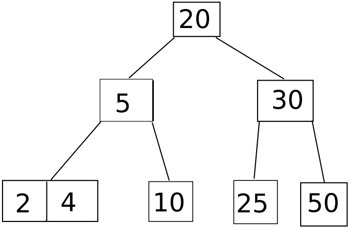


- Let's now insert 50.

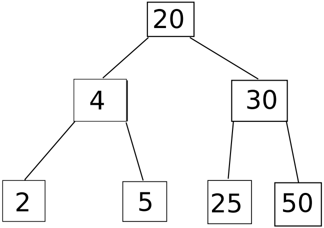


- Now lets delete 15.

Q. How can we delete 10?



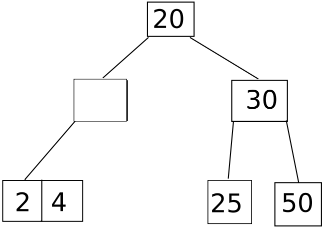
A. We *borrow* from a sibling.



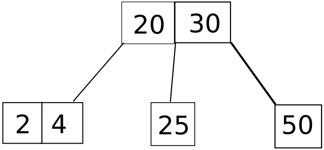
Delete With Merge

Q. What if we want to delete 5? Can we borrow?

A. No. Need to *merge*.



- Notice that this leaves a vacancy in the parent node...need to fix this by *merging* or *borrowing* again.



- In this case, *merge*.

SUMMARY

For a *2-3 Tree* or a *B-Tree*:

insert

If a node *overflows* we *split* the node and push up the middle value. If this causes an *overflow* repeatedly correct.

delete

If a node *underflows* we

- Try to *borrow* a key from a *sibling* (if the sibling has more than $\lceil \frac{m}{2} \rceil$ keys).
- Or *merge* the remaining keys with the *parent* node. If this causes an *underflow* repeatedly correct.

- Binary trees
- Not always balanced
- And any one operation can be $O(n)$

Q. So why do we like them?

A. When we do a series of $k \geq n$ operations, the series of operations is $O(klog(n))$.

- This means each operation's *amortized* cost is $O(\log n)$.
- Another nice feature, nodes *regularly accessed* will move towards the *root*.

Splaying

Basic idea: When we *insert/search* for a node *x*, move it to the root, balancing as we go.

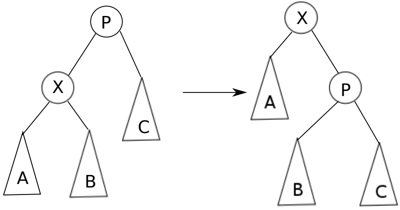
This is called **splaying**.

- Keep moving *x* up the tree 2 nodes at a time until it becomes the root.
- Three varieties: *zig-zag*, *zig-zig* and *zig*.
- Depends on relationship to parent node *p* & grandparent node *g*.

Zig

Splay on *x*.

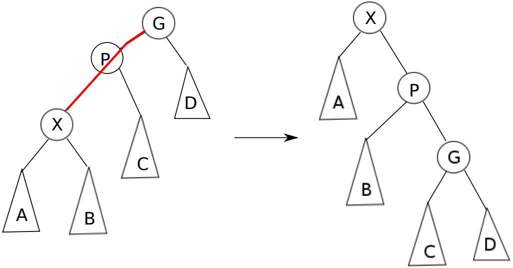
- When the parent node *p* of *x* is the *root*.
- We *zig*.



ZIG-ZIG

Splay on *x* with parent *p*, grand-parent *g*:

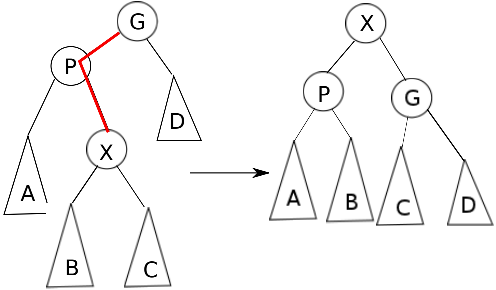
- When the *p* and *x* are both *left* children or both *right* children.
- In other words, *g*, *p* and *x* make a straight line.
- We *zig-zig*.
- First rotate about *p-g* and then rotate about *x-p*.



ZIG-ZAG

Splay on *x* with parent *p*, grand-parent *g*:

- When one of *p* and *x* is a *left* child and the other is a *right* child.
- In other words, *g*, *p* and *x* make a bend.
- We *zig-zag*.
- First rotate about *p-x* and then rotate about *x-g*.



DELETE

- We always *splay* the node being *inserted* or *searched* for.

Q. What about *delete*?

A. We replace the deleted node with the *predecessor* or *successor*.

And we *splay* the *parent* of the node being *deleted*.

Multiple Splay Example

Splay on node *c*:

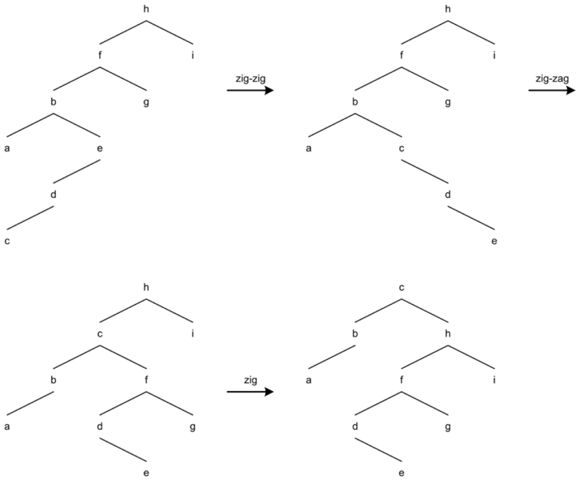


**image credit: http://digital.cs.usu.edu/ allan/DS/Notes/Ch22.pdf*