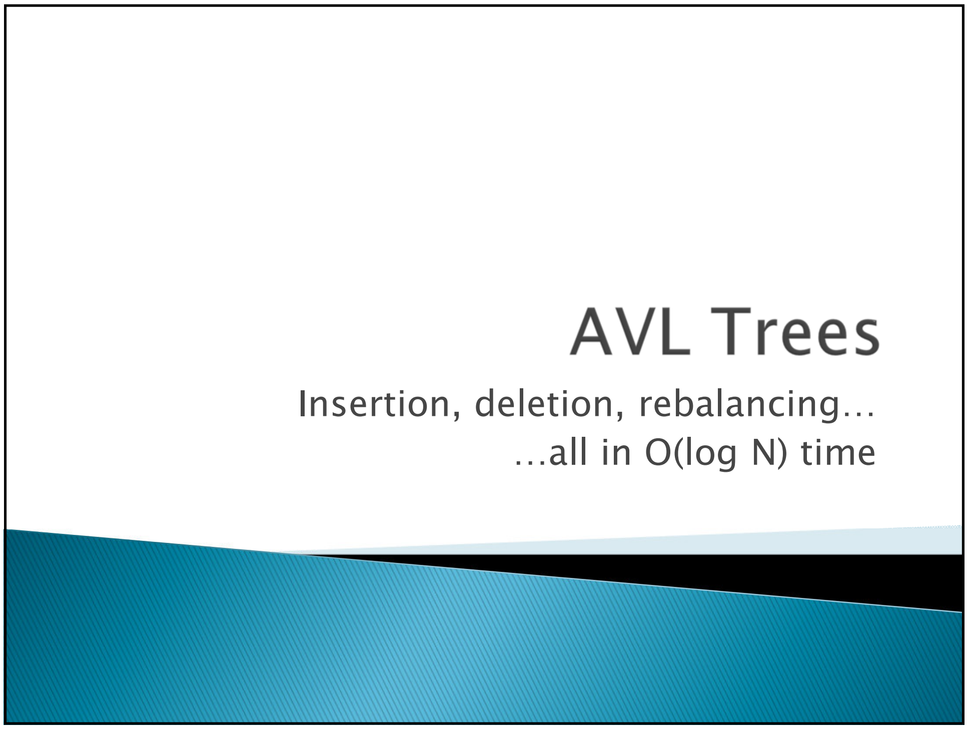
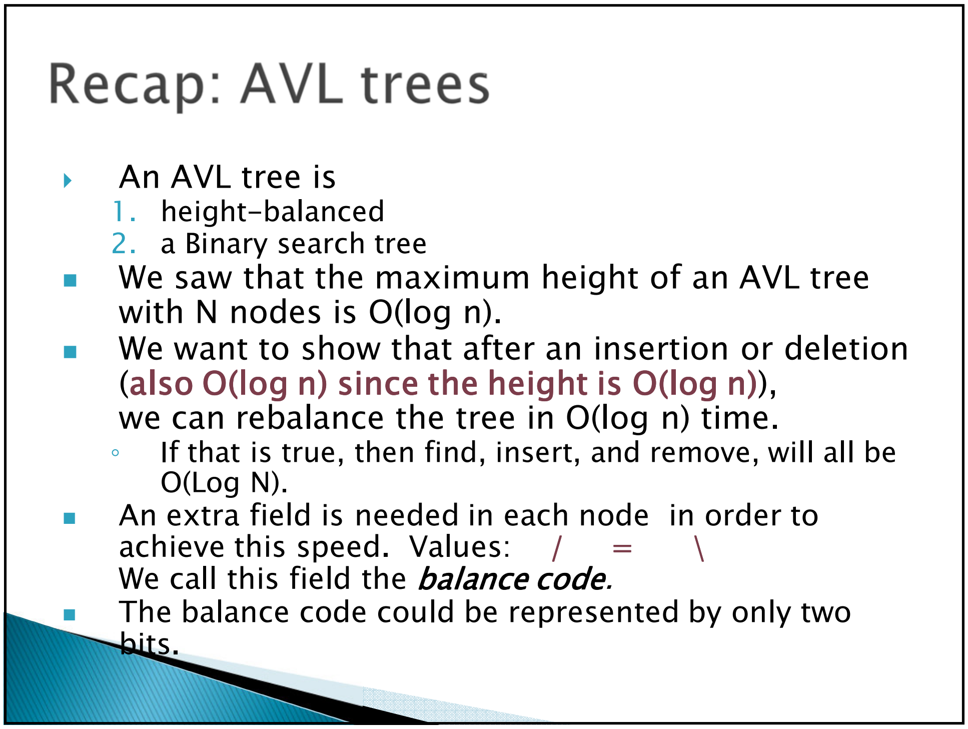


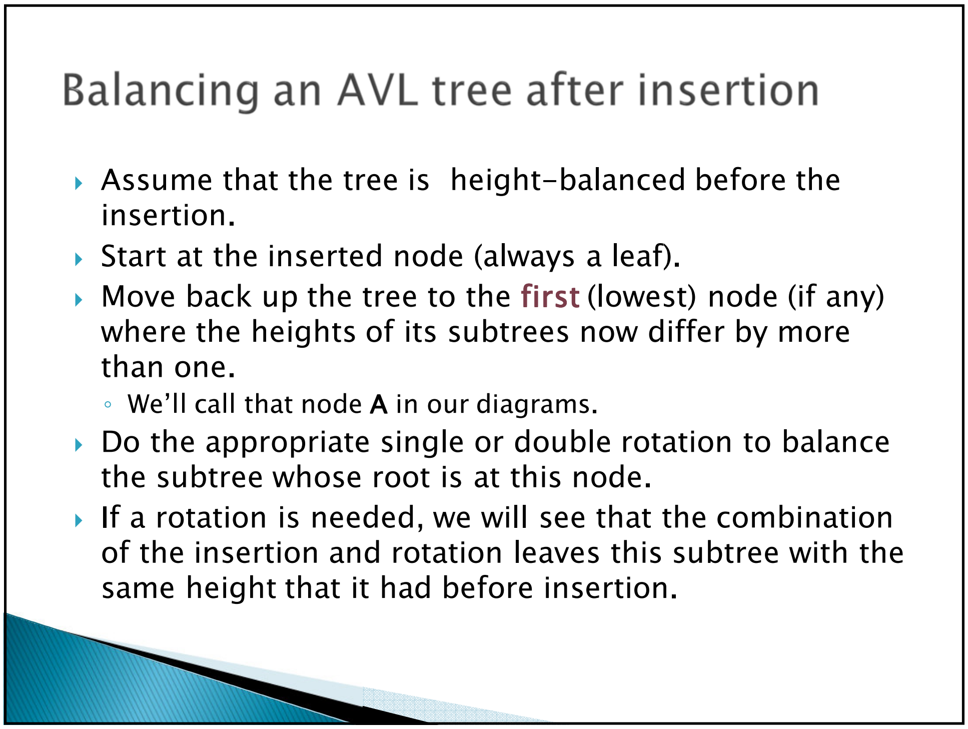
AVL Trees

Insertion, deletion, rebalancing…
…all in O(log N) time



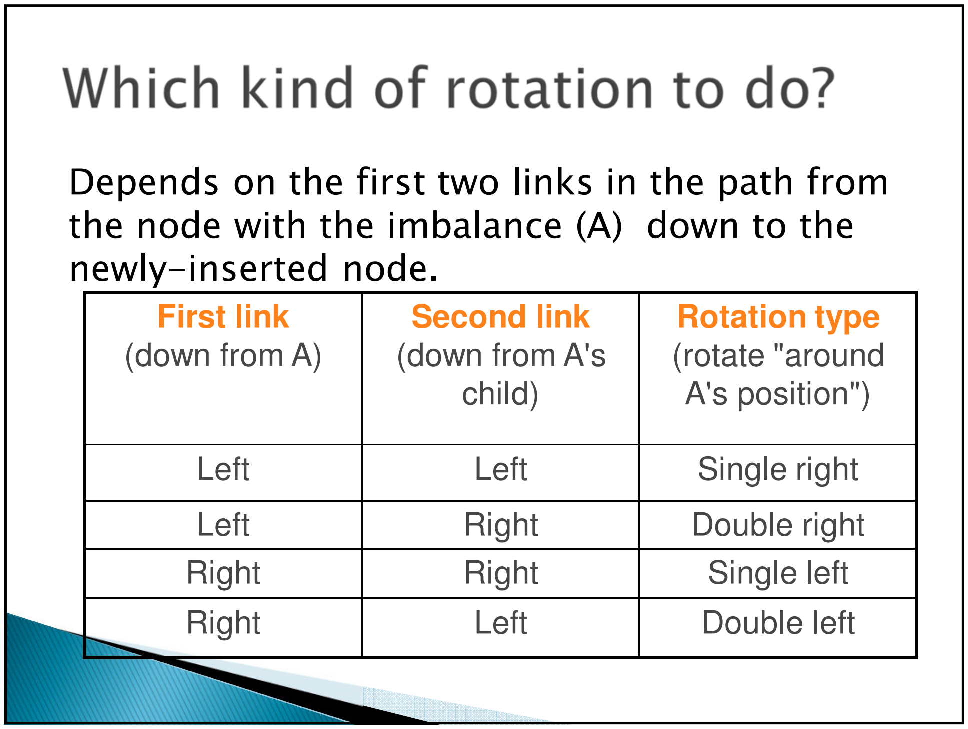
Recap: AVL trees

- An AVL tree is
  1. height-balanced
  2. a Binary search tree
- We saw that the maximum height of an AVL tree with N nodes is O(log n).
- We want to show that after an insertion or deletion (also O(log n) since the height is O(log n)), we can rebalance the tree in O(log n) time.
  - If that is true, then find, insert, and remove, will all be O(Log N).
- An extra field is needed in each node in order to achieve this speed. Values: / = \
  We call this field the ***balance code.***
- The balance code could be represented by only two bits.



Balancing an AVL tree after insertion

- Assume that the tree is height-balanced before the insertion.
- Start at the inserted node (always a leaf).
- Move back up the tree to the **first** (lowest) node (if any) where the heights of its subtrees now differ by more than one.
  - We'll call that node **A** in our diagrams.
- Do the appropriate single or double rotation to balance the subtree whose root is at this node.
- If a rotation is needed, we will see that the combination of the insertion and rotation leaves this subtree with the same height that it had before insertion.

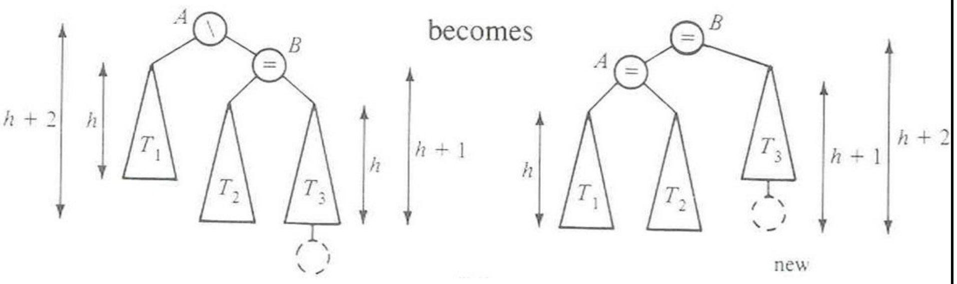


Which kind of rotation to do?

Depends on the first two links in the path from the node with the imbalance (A) down to the newly-inserted node.

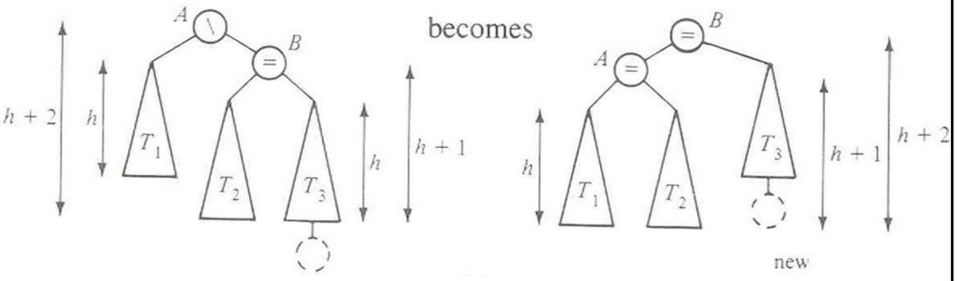
First link (down from A)	**Second link** (down from A's child)	**Rotation type** (rotate "around A's position")
Left	Left	Single right
Left	Right	Double right
Right	Right	Single left
Right	Left	Double left

Single left rotation (right is the mirror image of this picture)



Diagrams are from *Data Structures* by E.M. Reingold and W.J. Hansen.

Your turn — work with a partner



- Write the method:

-
  ```
  BalancedBinaryNode singleRotateLeft (
      BalancedBinaryNode parent,    /* A */
      BalancedBinaryNode child      /* B */   ) {

  }
  ```
- Returns a reference to the new root of this subtree.
- Don’t forget to set the balanceCode fields of the nodes.

Double left rotation (right is the mirror image) Weiss calls this "right-left double rotation"

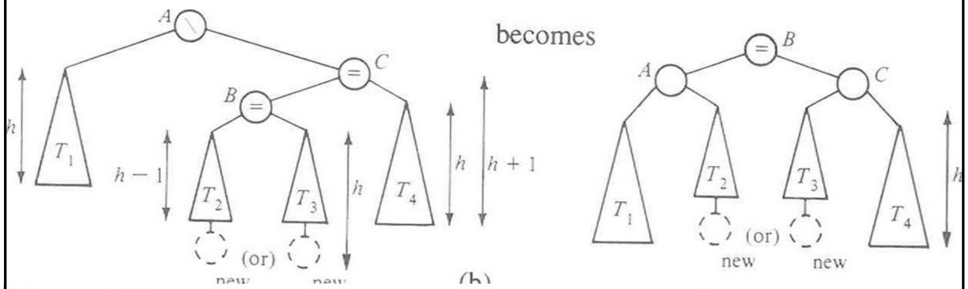


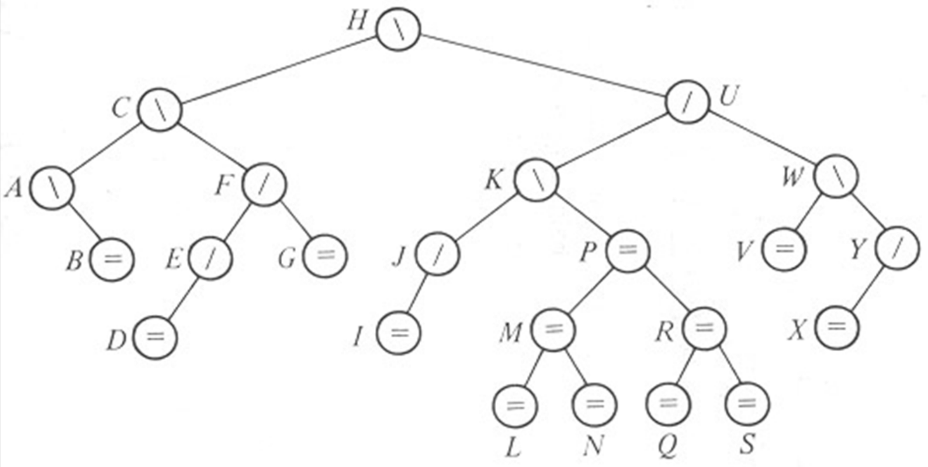
Figure 7.15
The transformations used to rebalance a height-balanced tree after the insertion of a new element: (a) rotation around A, (b) double rotation around A. The height condition codes in A and C in the right-hand drawing of (b) depend on whether the new element is at the bottom of T_2 or T_3. Both T_2 and T_3 are empty when B is the new element (see also Exercise 9). Notice that in each transformation the inorder of the tree is unchanged and the height of the tree *after* the transformation is the same as the height of the tree *before* the insertion. In each case, there are corresponding mirror-image transformations.

Your turn — (after class?)

- Write the method:
-
```
BalancedBinaryNode doubleRotateLeft (
  BalancedBinaryNode parent,      /* A */
  BalancedBinaryNode child,       /* C */
  BalancedBinaryNode grandChild   /* B */ ) {


}
```
- Returns a reference to the new root of this subtree.

A sample AVL tree



Insert **HA** into the tree, then **DA**, then **O**.
Delete **G** from the original tree, then **I**, **J**, **V**.

Your turn again (after class)

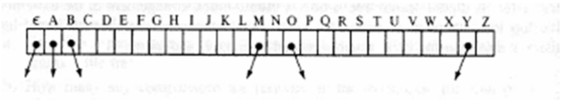
- Start with an empty AVL tree.
- Add elements in the following order; do the appropriate rotations when needed.
  - 1 2 3 4 5 6 11 13 12 10 9 8 7
- How should we rebalance if each of the following sequences is deleted from the above tree?
  - (10 9 7 8) (13) (1 5)
  - For each of the three sequences, start with the original 13-element tree. E.g. when deleting 13, assume 10 9 8 7 are still in the tree.

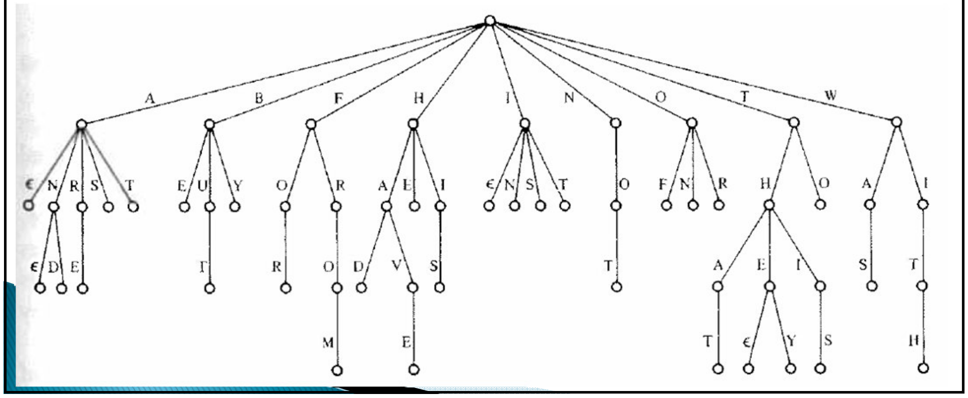
Other approaches to Tree Balancing

- Red-black trees
- AA trees
  - Red-Black and AA-trees are simpler to implement than AVL trees, but harder to understand why they work.
- balanced multiway trees (B+ trees)
  - Used for disk-based searches, and for database index storage.
  - Algorithms similar to red-black trees.
- Splay trees
  - Reasonably simple algorithms, amortized log N time.
- Skip Lists
  - An alternative to trees
- We will talk about one or more of these alternatives near the end of the course.

Another approach to search trees

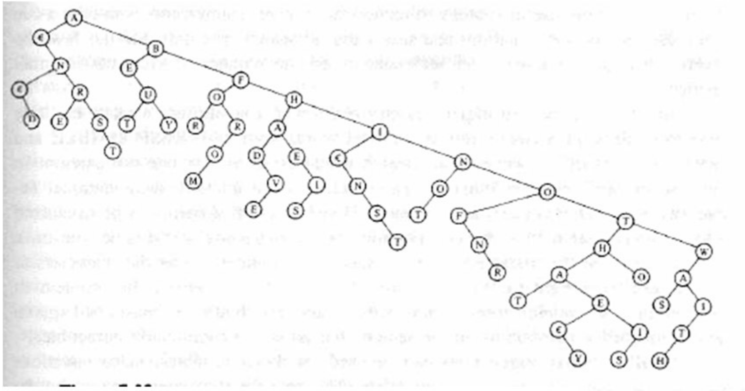
- Digital search tree (trie).
- We store the data digit-by-digit (or letter by letter).
- How to actually represent nodes?



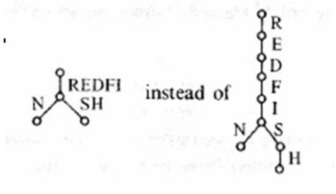


Improving Trie Space-efficiency

- Represent it as a binary tree



- Collapse "single branch" paths.
- Have a single "ϵ-node"

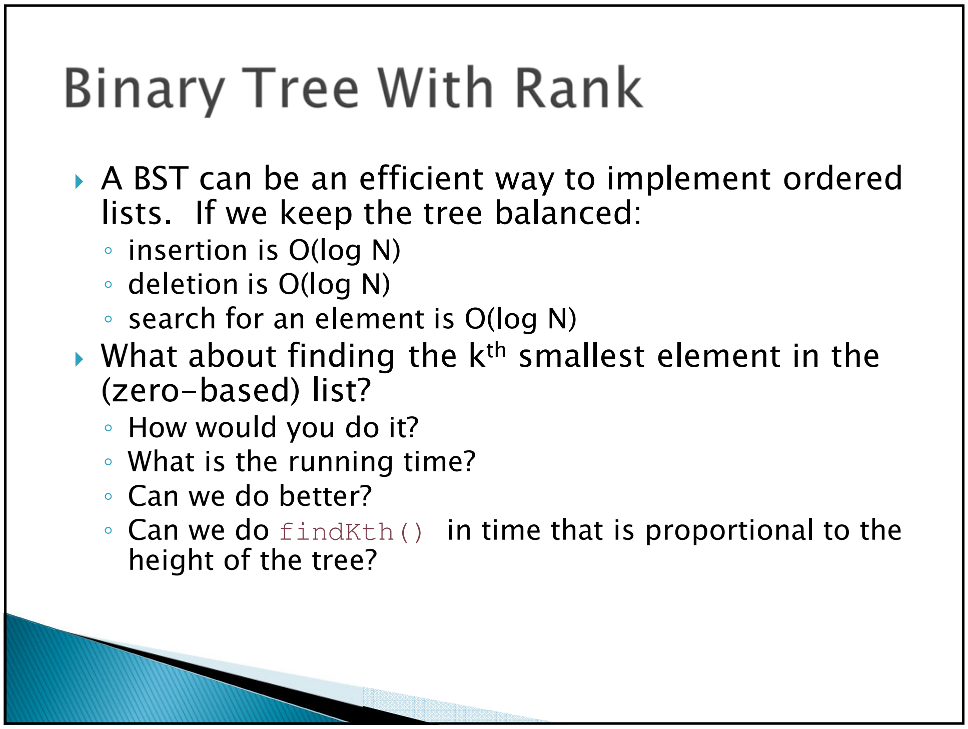


Interlude: What is Computer Science?

What is the central core of the subject? What is it that distinguishes it from the separate subjects with which it is related?

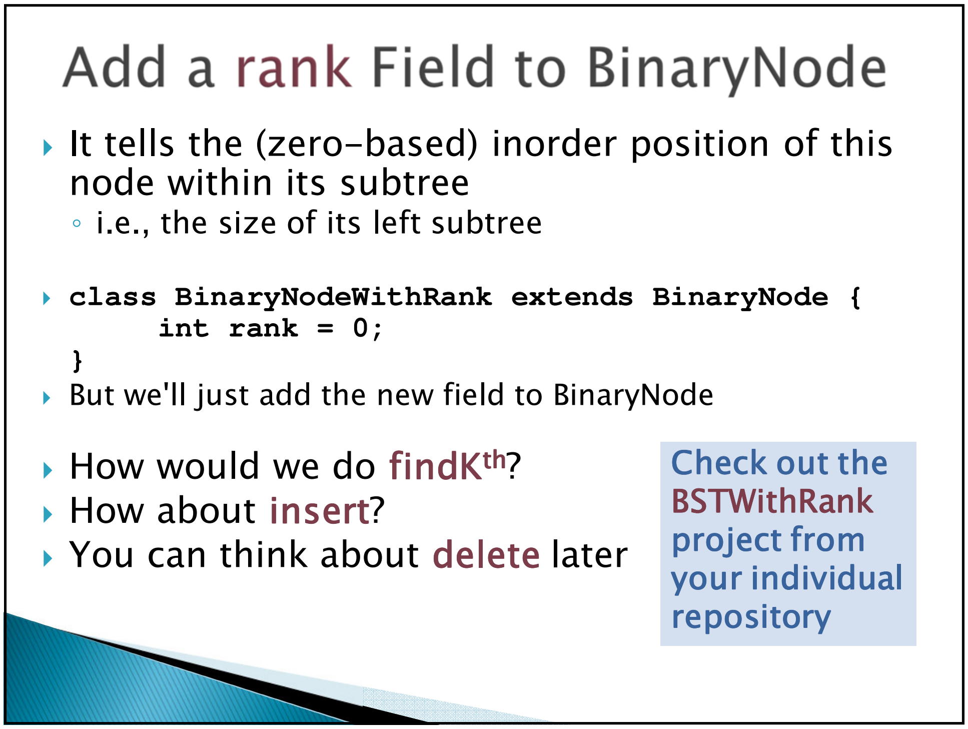
What is the linking thread which gathers these disparate branches into a single discipline? My answer to these questions is simple --- *it is the art of programming a computer*.

This slide is from a talk by Owen Astrachan, given at SIGCSE 2004.



Binary Tree With Rank

- A BST can be an efficient way to implement ordered lists. If we keep the tree balanced:
  - insertion is O(log N)
  - deletion is O(log N)
  - search for an element is O(log N)
- What about finding the k^{th} smallest element in the (zero-based) list?
  - How would you do it?
  - What is the running time?
  - Can we do better?
  - Can we do `findKth()` in time that is proportional to the height of the tree?



Add a rank Field to BinaryNode

- It tells the (zero-based) inorder position of this node within its subtree
  - i.e., the size of its left subtree

```
class BinaryNodeWithRank extends BinaryNode {
      int rank = 0;
}
```

- But we'll just add the new field to BinaryNode

- How would we do $findK^{th}$?
- How about insert?
- You can think about delete later

Check out the BSTWithRank project from your individual repository