Lecture 1

Course administration Search trees: AVL-trees

TDDC32

Lecture notes in *Design and implementation of a software module in Java* 14 January 2013

Tommy Färnqvist, IDA, Linköping University

1.1

Lecture Topics

Contents

1.2

1 Course administration

Teachers and Staff

- Course leader, examiner, lectures: Tommy Färnqvist `tommy.farnqvist@liu.se`
- Course assistant, labs, project: Daniel Persson `daniel.o.persson@liu.se`
- Assistant, labs, project: Rebecka Geijer Michaeli `rebge882@student.liu.se`
- Course administrator: Annelie Almquist `annelie.almquist@liu.se`

1.3

Examination

- Written exam (U, 3, 4, 5), 1.5 ECTS credits
- Lab assignments (Fail, Pass), 1.5 ECTS credits
- Project (U, 3, 4, 5), 3 ECTS credits
  - Documentation (25-33%)
  - Programming (66-75%)
    - ∗ Amount and quality of code, coding conventions, comments, exception handling

Final grade: 33% written exam + 66% project work

1.4

Course Aim According to LiTH Study Guide

After completing the course the students shall have good skills in coding software modules in Java using a development environment. The modules shall at least be 300 lines of well-structured code. The students shall have acquaintance with object-oriented analysis and design plus unit testing of their own code. Deepened studies in data structures and algorithms shall prepare students for specialisation courses.

1.5

Lectures

- Course administration, aims. AVL-trees.
- Multi way search trees, Splay trees
- ADT Map, Hash Tables, ADT Dictionary. Skip Lists.
- Threads. Networking.
- Java GUI programming. JDBC connections.
- Three lectures on OOAD, UML, and introduction to project management.

1.6

Lab assignments

- Lab 1: AVL Tree, Hash Tables
- Lab 2: Advanced Java programming
- (Optional) Lab 3: UML

- Work in pairs
- Please register in WebReg Friday 18 January at the latest!
- Group A: SU02, SU03
- Group B: SU04, SU06
- Note: Only 2/3 of the labs are staffed

Deadline for labs is Wednesday 13 February.

1.7

Project

- Work in groups of three
- Please hand in project idea and register in WebReg Thursday 14 February at the latest!
- Note: Each project member is expected to contribute 80-100 hours to the project
  - Weekly report to assistant (outside exam period)

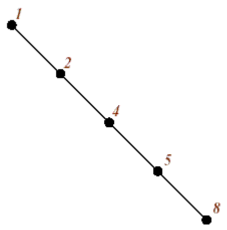
- Deadline for Requirements Specification is Friday 22 February
- Deadline for Analysis and Design Document is Tuesday 5 March
- Deadline for implementation and demonstration is Tuesday 7 May

1.8

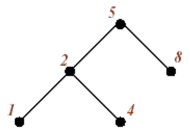
2 Binary Search Trees

Binary Search Trees are not Unique

The same data can yield different binary search trees



insert: 1,2,4,5,8



insert: 5,2,1,4,8

1.9

Successful look-up

BST in worst case

- BST degenerated into linear sequence
- expected number of comparisons is $(n+1)/2$

Balanced BST

- height is $O(\log_2 n)$
- $O(\log_2 n)$ comparisons

1.10

3 AVL Trees

AVL Tree

- Self balancing BST/height balanced BST
- AVL = Adelson-Velskii och Landis, 1962
- The idea: Keep updated balance information in each node
- AVL propertyFor each internal node v in T, the heights of the children of v differ by at most 1 ... or alternatively... For each internal node v in T, the following holds:

$$b(v) \in \{-1, 0, 1\}, \text{ where}$$

$$b(v) = \text{height}(\text{leftChild}(v)) - \text{height}(\text{rightChild}(v))$$

1.11

Maximum Height of AVL Tree

Proposition 1. The height of an AVL tree storing n elements is $O(\log n)$.

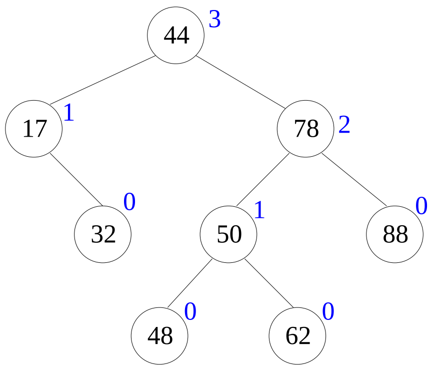
Which has the consequence that...

Proposition 2. We can do find, insert, and remove in an AVL tree in time $O(\log n)$ while preserving the AVL property.

1.12

Example: an AVL tree

1.13

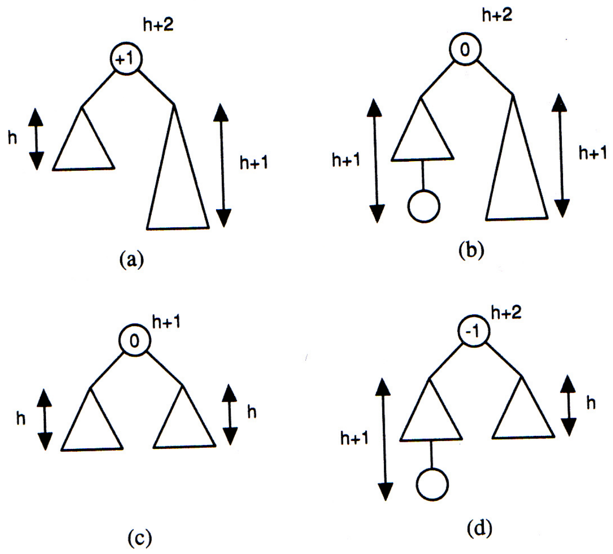


Insertion in AVL trees

- The new node changes the tree height and the tree has to be re-balanced.
  - Information about the height of sub trees can be represented in different ways:
    - Store the height explicitly in each node
    - Store the balance factor in each node
- The change is usually described as a left or right rotation of a sub tree.
- One rotation is sufficient to re-balance the tree.

1.14

Insertion in AVL trees (simple cases)



1.15

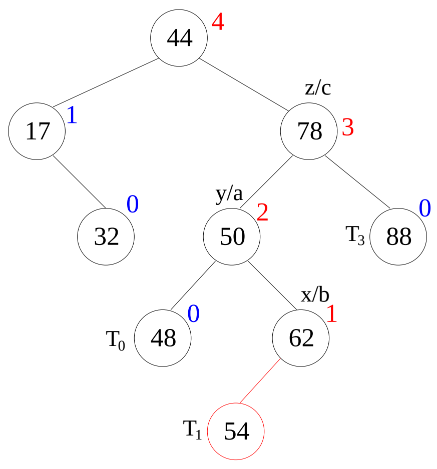
Insertion algorithm

- Start from the new node and search upwards until a node x is found, such that its grandparent z is unbalanced. Mark the parent of x with y. Reconstruct the tree as follows:
  - Rename x, y, z to a, b, c based on their inorder order.
  - Let T_0, T_1, T_2, T_3 be an enumeration in inorder of the sub trees of x, y och z. (None of the sub trees can have x, y, or z as root.)
  - Exchange z for b, its children are now a and c.
  - T_0 and T_1 are children of a, and T_2 and T_3 are children of c.

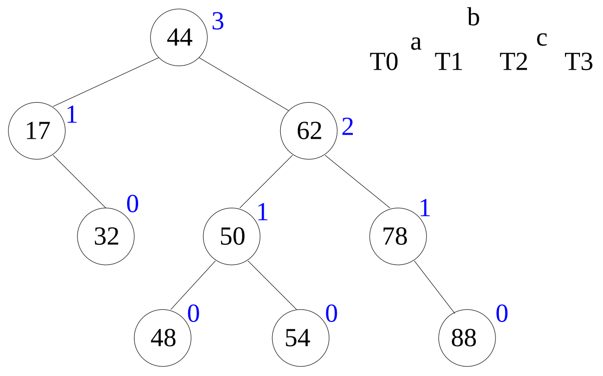
1.16

Example: insertion in an AVL tree

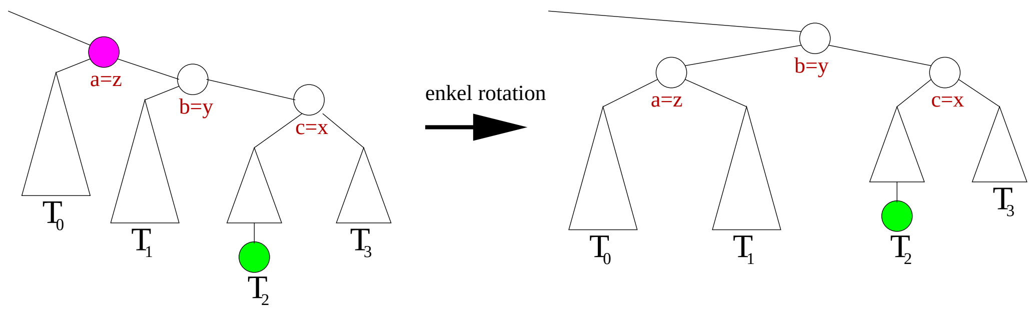
1.17



Example: insertion in an AVL tree

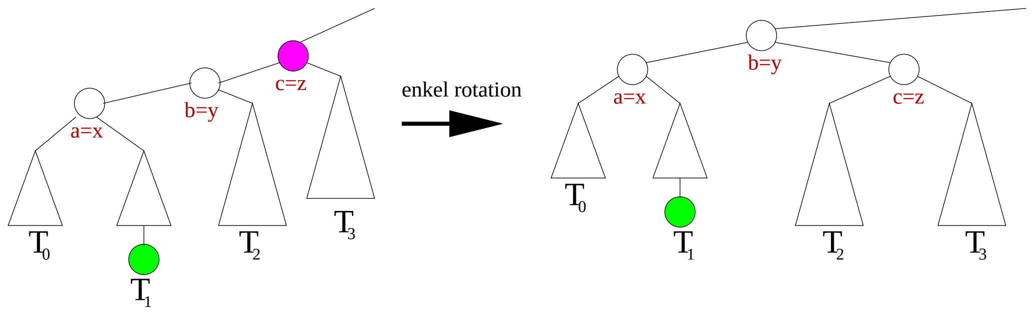


Four different rotations



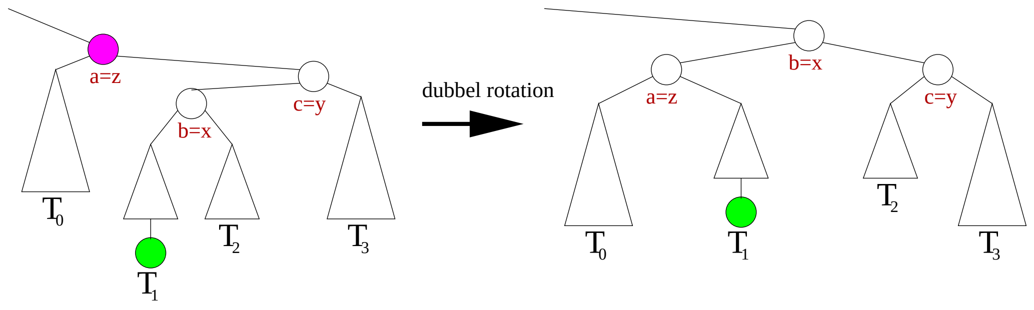
If $b = y$ we call it a single rotation."Rotate y up above z"

Four different rotations



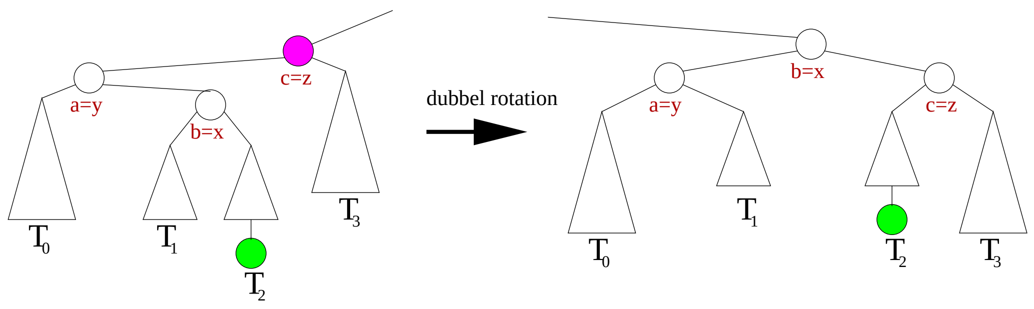
If $b = y$ we call it a single rotation."Rotate y up above z"

Four different rotations



If $b = x$ we call it a double rotation."Rotate x up above y and then above z"

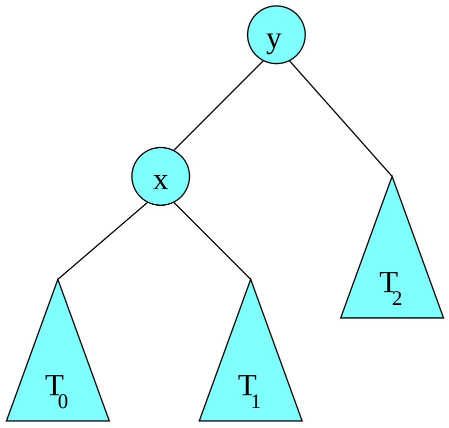
Four different rotations



If $b = x$ we call it a double.”Rotate x up above y and then above z”

1.22

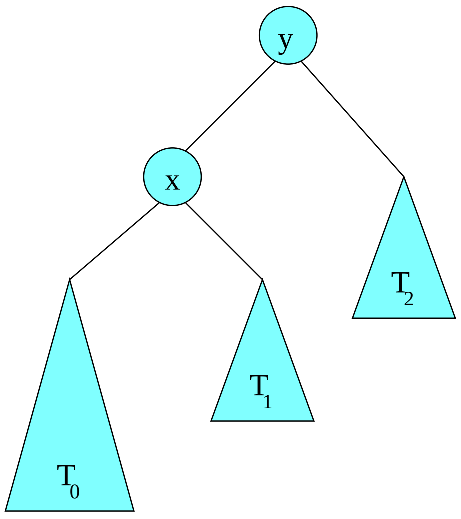
Another way to describe it



Assume that we have balance. . .

1.23

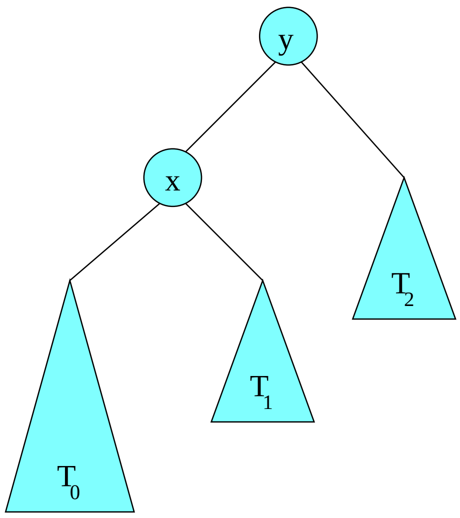
Another way to describe it



. . . and then insert something which destroys it

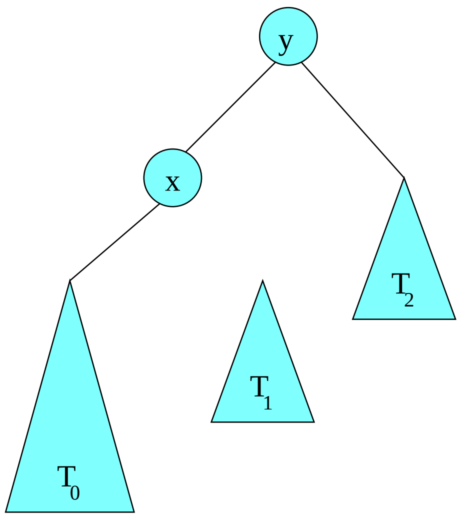
1.24

Another way to describe it



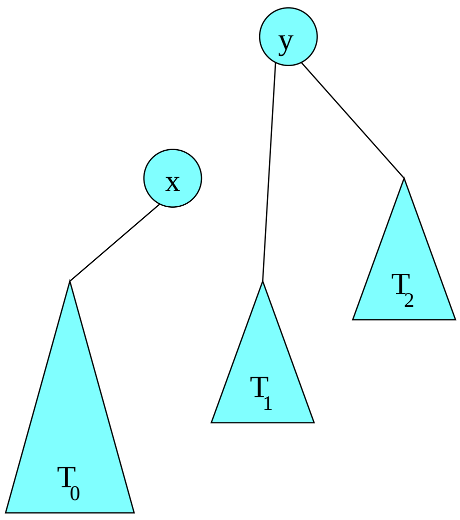
Do a single rotation

Another way to describe it



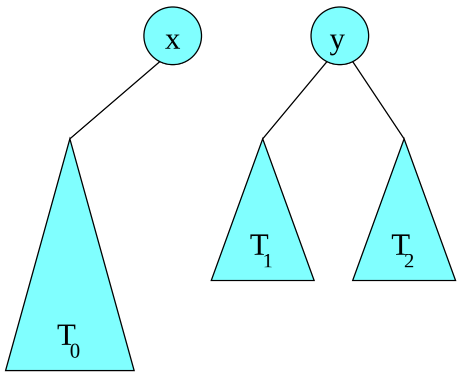
Do a single rotation

Another way to describe it



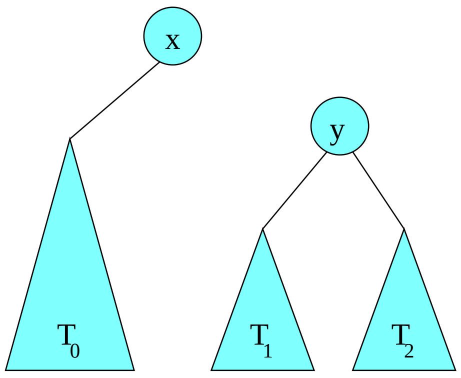
Do a single rotation

Another way to describe it



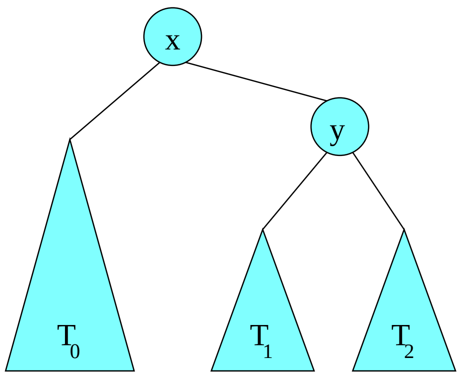
Do a single rotation

Another way to describe it



Do a single rotation

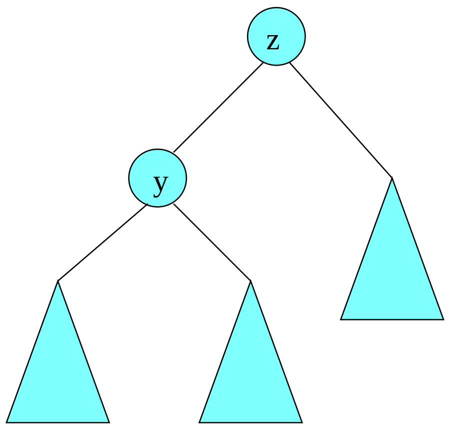
Another way to describe it



Done!

1.30

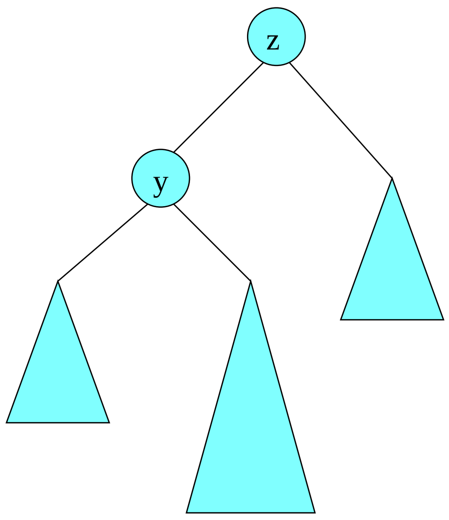
Another way to describe it



Another example...

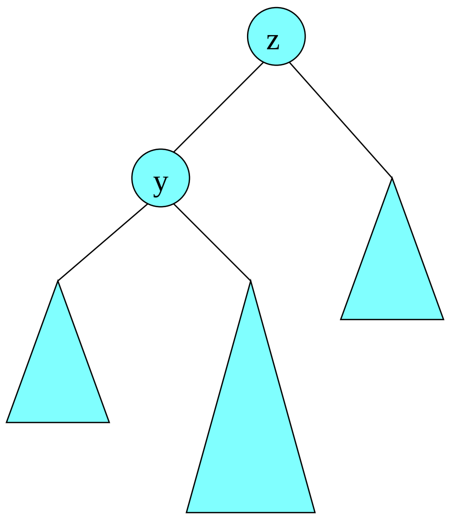
1.31

Another way to describe it



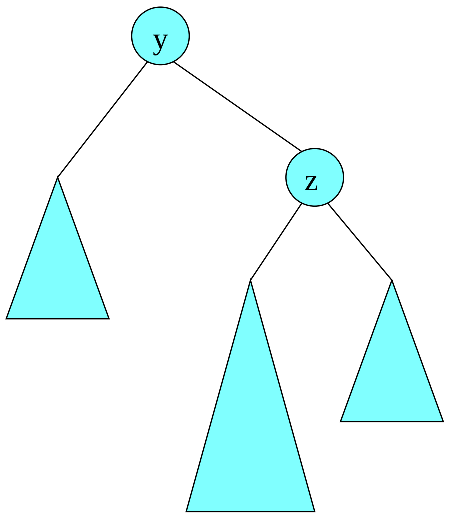
. . . this time the insertion is in another position 1.32

Another way to describe it



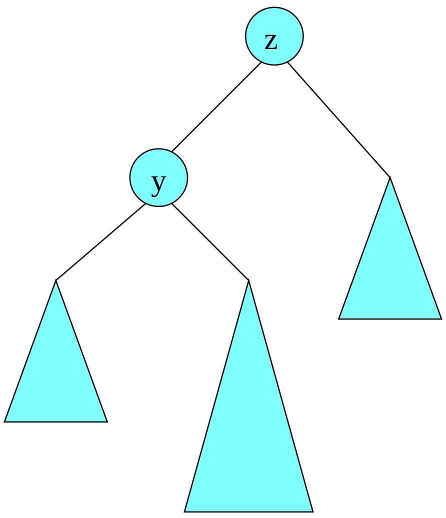
Try a single rotation again. . . 1.33

Another way to describe it



. . . hmm, we did not get balance 1.34

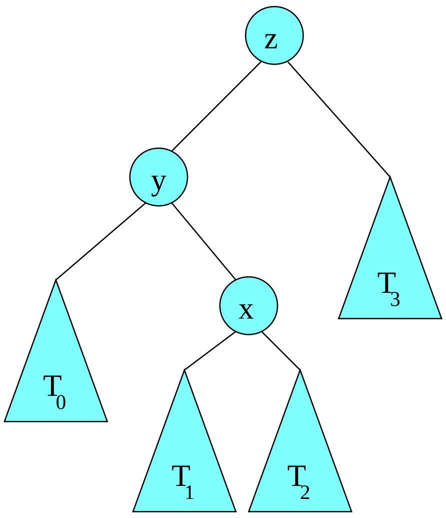
Another way to describe it



Start over. . . and look at the structure of *y*

1.35

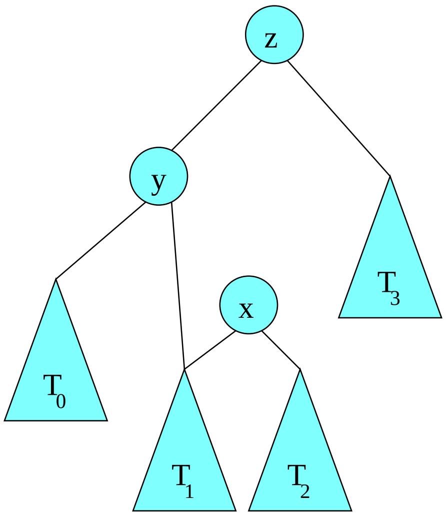
Another way to describe it



We have to perform a double rotation

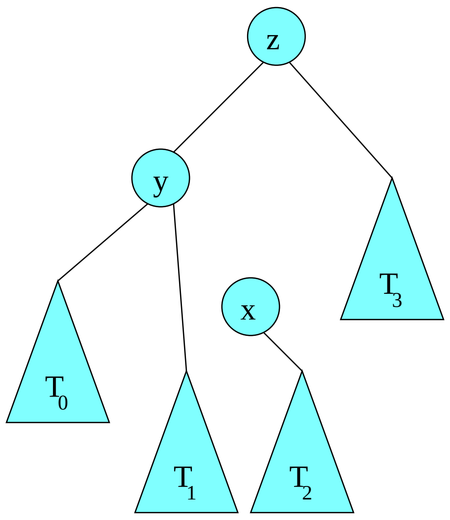
1.36

Another way to describe it



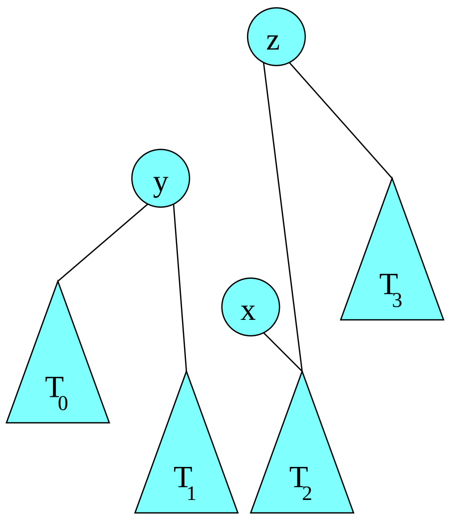
We have to perform a double rotation

Another way to describe it



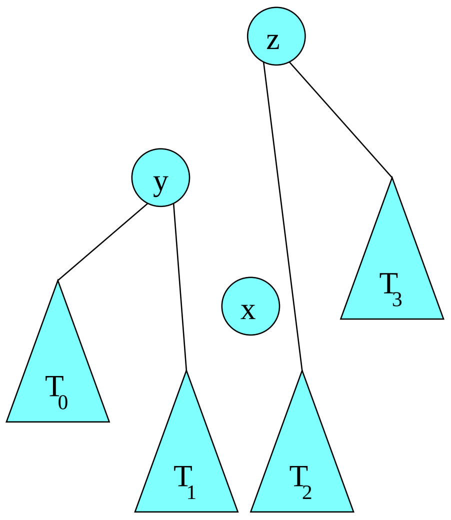
We have to perform a double rotation

Another way to describe it



We have to perform a double rotation

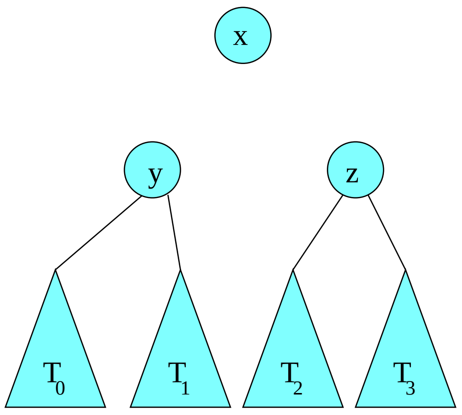
Another way to describe it



We have to perform a double rotation

1.40

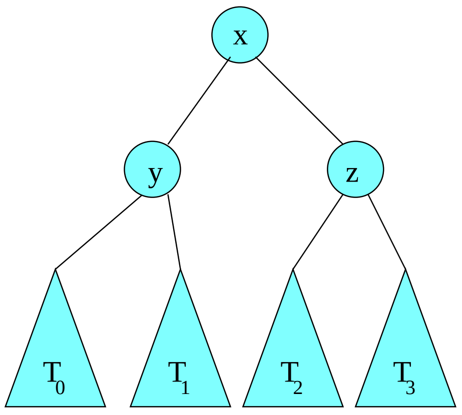
Another way to describe it



We have to perform a double rotation

1.41

Another way to describe it



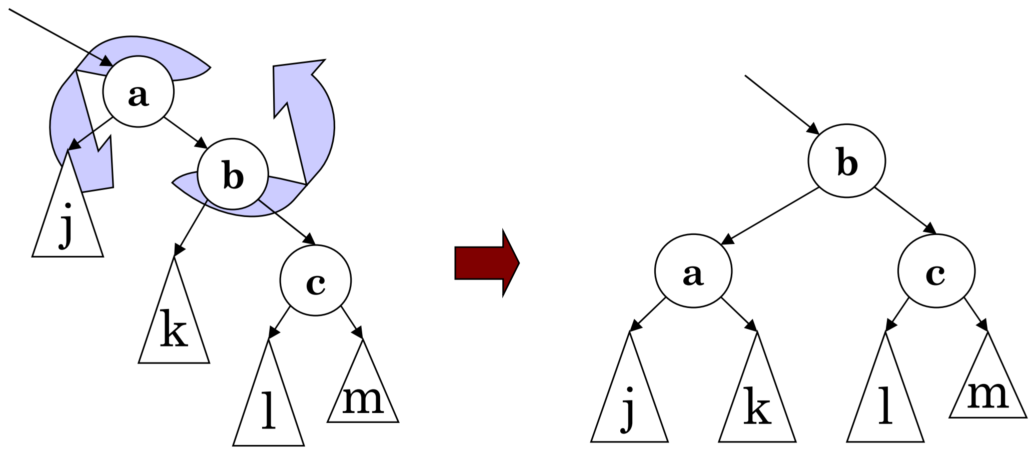
Done!

1.42

Trinode restructuring = rotations. . .

Some authors use left and right rotations: Single left rotation:

- left part of subtree (a and j) is lowered
- we have "rotated (up) b above a"

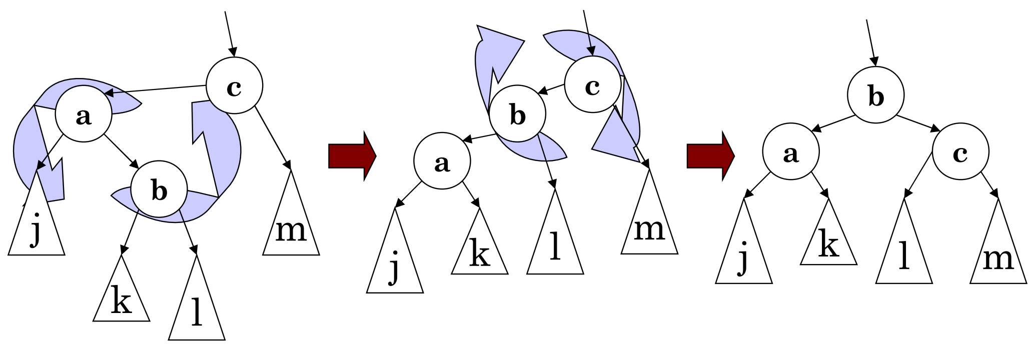


1.43

Double rotations. . .

Two rotations are needed when the nodes to be re-balanced are placed in a zig-zag pattern.

- Rotate b up above a
- Rotate b up above c



1.44

Deletion in an AVL tree

- find and remove as in an ordinary binary search tree
- Update balance information on the way back up to the root
- If to unbalanced: Restructure . . . but . . .
  - When we restore balance in one position it might incur unbalance in another position
  - Have to repeat the re-balancing procedure (or balance control) until the root is reached
  - At most $O(\log n)$ re-balancing operations

1.45

Voluntary Homework Problem

A set $K = \{k_1, \ldots, k_n\}$ of $n \geq 1$ keys are to be stored in an AVL tree. Which of the following statements are always true?

(A) If k_i is the smallest key in K, then the left child of the node storing k_i in every AVL tree storing K is a leaf node.
(B) Every AVL tree storing K has exactly the same height, no matter what order was used to insert the keys into the tree.
(C) A preorder traversal of an AVL tree storing K visits the nodes in increasing key order.
(D) In every AVL tree storing K the key stored in the root node is the same, no matter what order was used to insert the keys into the tree.

1.46