Balanced Search Trees

- Balanced Search Trees are used to efficiently implement the "Dictionary" operation:
  - INSERT
  - DELETE
  - FIND

(Unordered) dictionary:

No relationship between distinct keys

Ordered dictionary:

Relationship between distinct keys

Allows, access in order of in/de-creasing keys

Additionally operations

closestElemBefore(Key k): find and return element with largest key less than or equal to k

closestElemAfter(Key k)

Balanced Search trees can efficiently implement an ordered dictionary.

Balanced Search Trees

- The key to the efficiency of all balanced search trees is that the height of a tree holding N data elements is O(log N).
- The main difficulty in implementing balanced search trees is maintaining the O(log N) height.

Balanced Multi-Way Search Trees

- 2-3 tree and 2-4 tree.
- Balance is accomplished with SPLIT, FUSE, and TRANSFER operations.
- Easiest balanced trees to understand and implement.
- Drawback is that the space requirements exceeds those of binary balanced search trees.

Binary Balanced Search Trees

- AVL trees, first published account of a balanced search tree.
- Balancing is accomplished by SINGLE and DOUBLE ROTATIONS
- Somewhat complicated to understand and implement.

Binary Balanced Search Trees

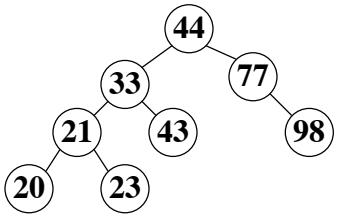
- Red-Black trees, simulates a 2-4 tree.
- Balancing is accomplished by SINGLE and DOUBLE ROTATIONS, as well as a clever "re-colouring" strategy
- Somewhat complicated to understand and implement.
- A careful implementation leads to the most efficient structure for an ordered dictionary.

Binary Balanced Search Trees

- AA trees, simulates a 2-3 tree.
- Balancing is accomplished by SINGLE ROTATIONS, coded as SPLIT and SKEW
- "Cleanest: of the balanced binary search trees. Easier to understand and code.
- A careful implementation leads to a very efficient structure for an ordered dictionary.

AVL Trees: Definition

- An **AVL Tree** T
  - is a **BST**
  - with **height-balance property**: for every internal node v of T, the heights of the children of v can differ by at most 1.
- **Example:**



- AVL trees are named after the initials of their inventors: Adel'son-Vel'skii and Landis.

Balance Factor

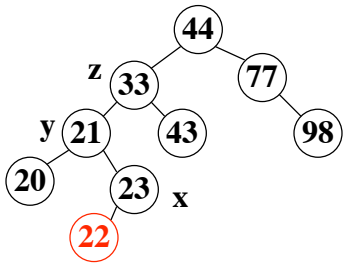
- Let v be an internal node of a binary tree
- The **balance factor** of v is defined as:
  (height of the left subtree of v) – (the height of the right subtree of v)
- **Alternative definition of AVL trees:**
  The balance factor of every internal node in an AVL tree is –1, 0 or 1.

Height of an AVL Tree

• **Proposition**: The height of an AVL tree *T* storing *n* keys is ***O***(log *n*).
• **Justification**: The easiest way to approach this problem is to try to find the minimum number of internal nodes of an AVL tree of height *h*: *n*(*h*).
• We see that *n*(1) = 1 and *n*(2) = 2
• for *n (*3), an AVL tree of height *h* with *n*(*h*) minimal contains the root node, one AVL subtree of height *n*-1 and the other AVL subtree of height *n*-2.
• *i*.e. *n*(*h*) = 1 + *n*(*h*-1) + *n*(*h*-2) {This is similar to the Fibonacci recurrence, and we know that it has an exponential lower bound.}
• We can solve an easier recurrence *n(h) > 2n(h-2)* to get an asymptotic bound.

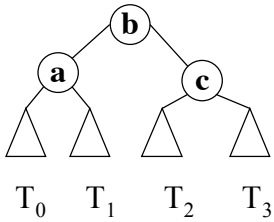
Inserting into an AVL tree

If we insert a new node *w* into an AVL tree we may ruin the balance. A rebalancing operation requires that we identify three node in the tree. Let **z** be the first *unbalanced* node encountered in the path from *w* to the root. Let **y** be the child of **z** and **x** the child of **y** in the path from z to *w*. Note that **x** may or may not be equal to w.



The Restructure Algorithm

- Find x,y,z and relabel them a, b, c according to their inorder traversal.
- Then relink as follows:

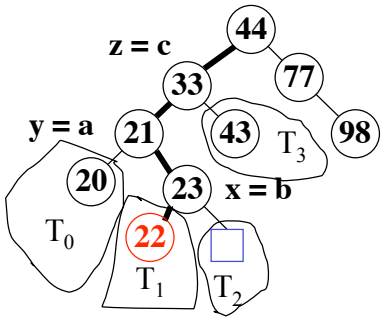


The Restructure Algorithm

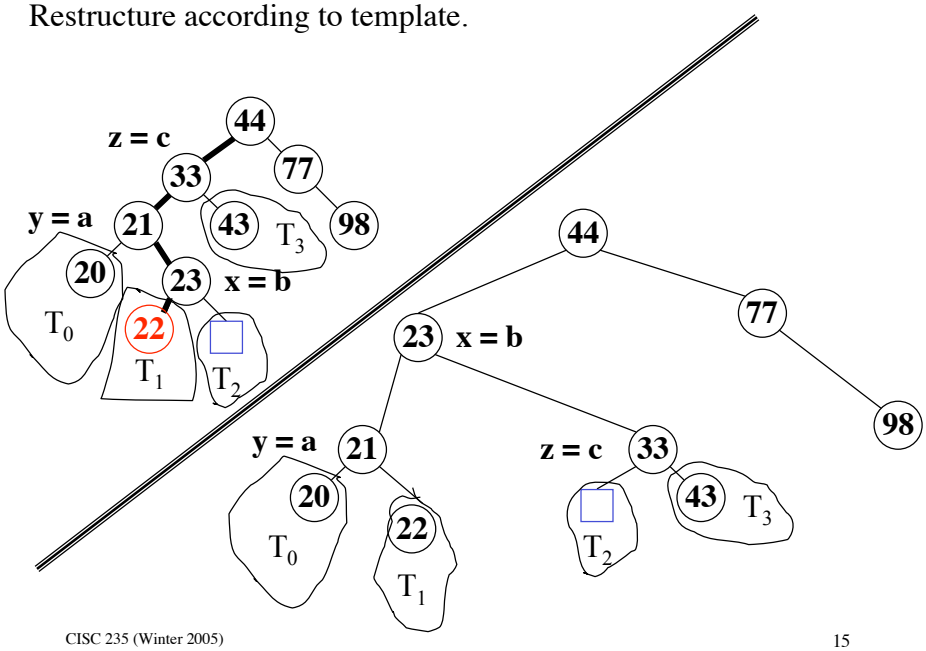
1. Let (a, b, c) be an **inorder** listing of the nodes x, y, and z
2. Let (T_0, T_1, T_2, T_3) be an inorder listing of the four subtrees of x, y, and z not rooted at x, y, or z
3. Replace subtree rooted at z with a new subtree rooted at b

- Make a the left child of b and make T_0, T_1 the left and right subtrees of a, respectively.
- Make c the right child of b and make T_2, T_3 be the left and right subtrees of c, respectively

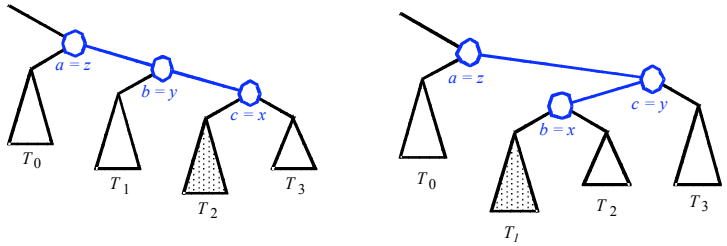
Relabel x, y, z and identify T_0, T_1, T_2, T_3.



Restructure according to template.



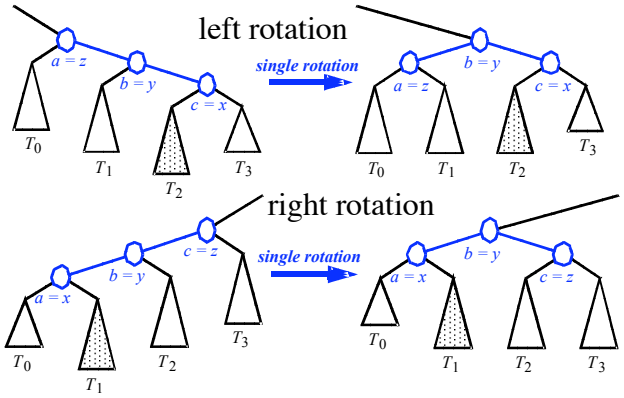
- Going from left to right, label subtrees of x, y, and z with T_0, T_1, T_2 , and T_3
- Label x, y, z with a, b, c based on order of nodes in **in-order** traversal
- Examples:



Case 1: Single Rotations

Case 1: y in the middle, *i.e.* b=y

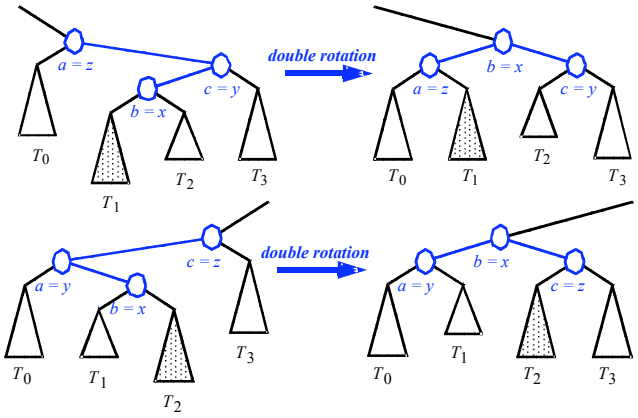
- perform **"single rotation"**
- y "rotated over" z



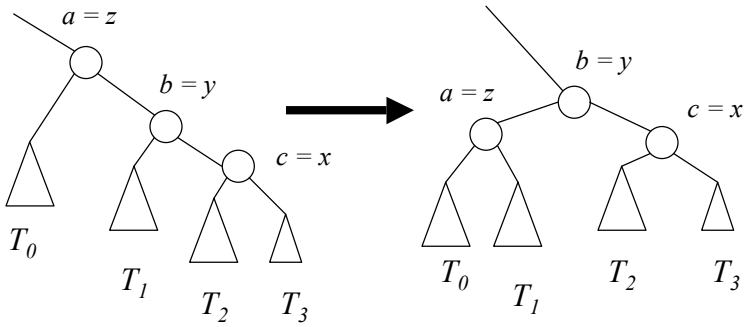
Case 2: Double Rotations

Case 2: x in the middle, *i.e.* b=x

- perform **"double rotation"**
- x "rotated" over y and then over z



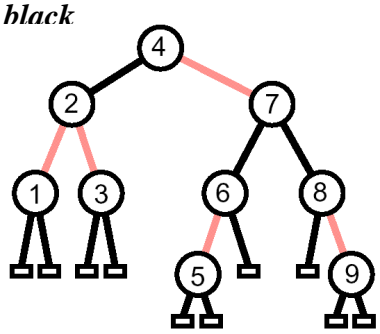
After performing a restructuring after an insertion an AVL is guaranteed to retain BST ordering, and will become balanced.



Red-Black Tree

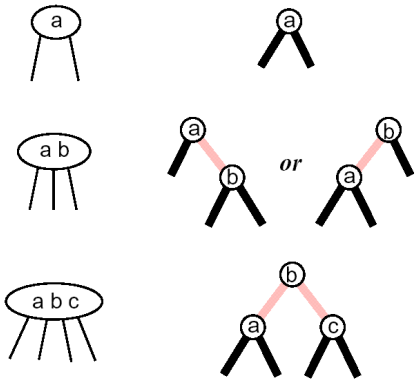
A **red-black tree** is a binary search tree with the following properties:

- edges are colored ***red*** or ***black***
- ***no two consecutive red edges on any root-leaf path***
- ***same number of black edges on any root-leaf path*** (***black height***)
- ***edges connecting leaves are black***



Note how **(2,4) trees** relate to **red-black trees**

(2,4) Red-Black



Now we see red-black trees are just a way of representing 2-4 trees!

Red-Black Tree Properties

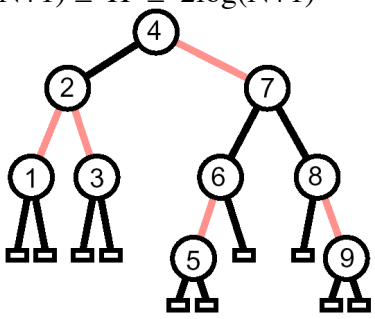
N := # of internal nodes; L:= # leaves (= N + 1);

H:= height B:= black height

Property 1: $2^B \leq N + 1 \leq 4^B$

Property 2: $(1/2) \log (N+1) \leq B \leq \log(N+1)$

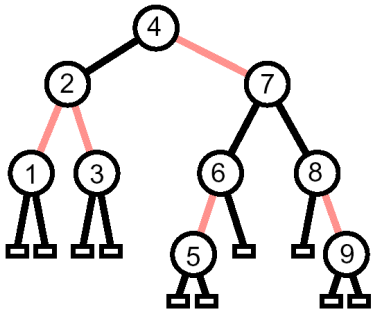
Property 3: $\log (N+1) \leq H \leq 2\log(N+1)$



This implies that searches take time O(log N)!

Searching in a **Red-Black Tree**.

No different than searching in a binary search tree.



Insertion into Red-Black Tree

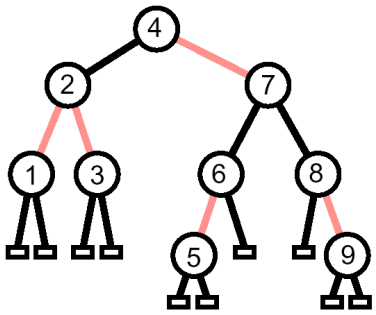
1.Perform a standard search to find the leaf where the key should be added.

2.Replace the leaf with an internal node with the new key

3.Color the incoming edge of the new node red.

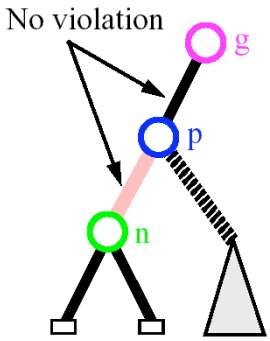
4.Add two new leaves, and color their incoming edges black.

5. If the parent had an incoming red edge, we now have two consecutive red edges! We must reorganize tree to remove that violation. What must be done depends on the sibling of the parent.



Let:
n be the new node
p be its parent
g be its grandparent

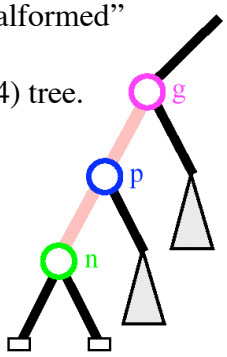
Case 1: Incoming edge of p is black.



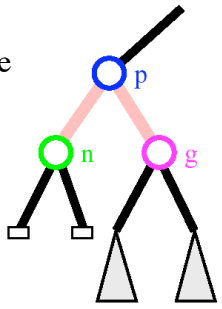
No restructuring required.

Case 2: Incoming edge of p is red, and its sibling is black.
Restructure.

Observe that:
- Inorder remains unchanged
- Black depth is preserved for all leaves
- No more consecutive red edges!
- Corrects "malformed" 4-node in the associated (2,4) tree.



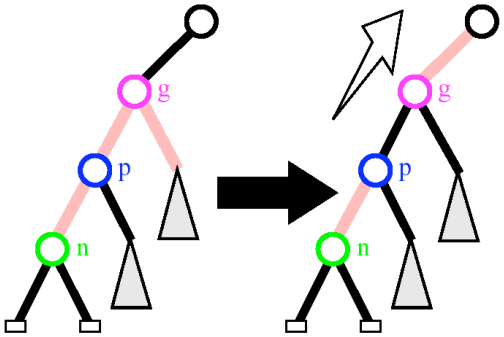
restructure



Case 3: Incoming edge of p is red and its sibling is also red.
Recolour.

Observe that:

- The black depth remains unchanged for all descendants of **g.**
- This process will continue upward beyond **g** if necessary: rename **g** as **n** and repeat.
- Splits 5-node of the associated (2,4) tree



Summary of Insertion

- If two red edges are present, we do either
- a *restructuring* (with a single or double rotation) and **stop**, or
- a recouloring and possibly continue
- A restructuring takes constant time and is performed at most once. It reorganizes an off-balanced section of the tree.
- Recolourings may continue up the tree and are executed O(log N) times.
- The **time complexity** of an insertion is **O(log N).**

Summary of Red-Black Trees

• An insertion or deletion may cause a local *perturbation* (two consecutive red edges, or a "**double-black"** edge)
• The perturbation is either
• *resolved locally* (restructuring), or *propagated* to a higher level in the tree by recolouring.
• O(1) time for a restructuring or recolouring.
• At most one restructuring per insertion, and at most two restructurings per deletion.
• O(log N) recolourings.
• Total time for insert or delete : O(log N)