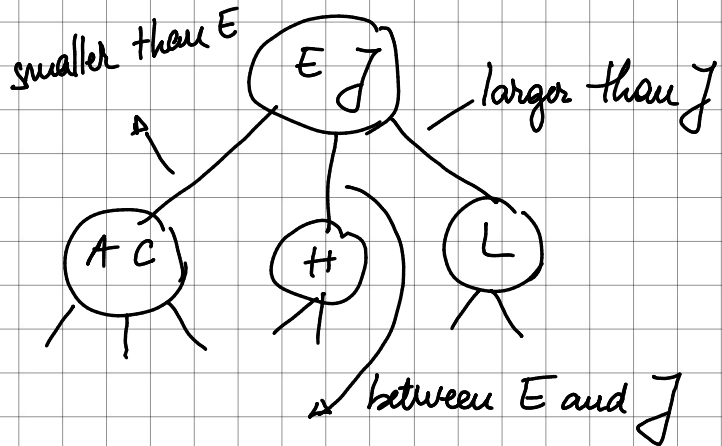
Balanced Search Trees

I) 2-3 Trees

- allow 1 or 2 keys per node
- **2-node**: 1 key, 2 children
- **3-node**: 2 keys, 3 children
  - 1 link for less
  - 1 link for between
  - 1 link for greater

- **perfect balance**: every path from root to null link has same length



- **symmetric order**: inorder traversal yields keys in ascending order

- insertion into a 3-node at the bottom
  - add new key to 3-node and create a temp. 4-node
  - move middle key into parent
  - repeat up the tree, as necessary
  - if you reach the root and it's a 4-node split it into 3 2-nodes => height of tree increases by 1

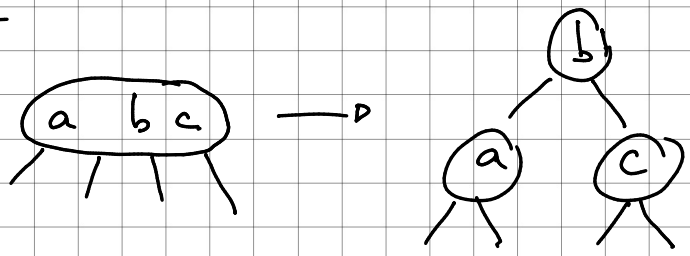
* splitting a 4-node is a local transformation: constant number of operations

Invariants

- maintains symmetric order and perfect balance

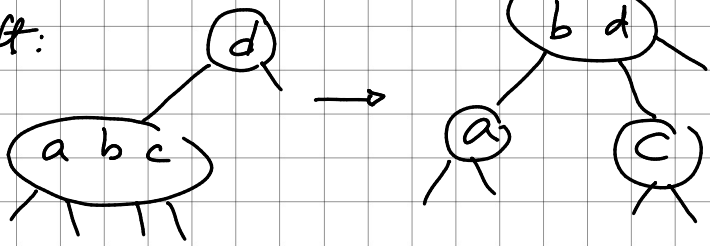
Proof - each transformation maintains symmetric order and perfect balance

root

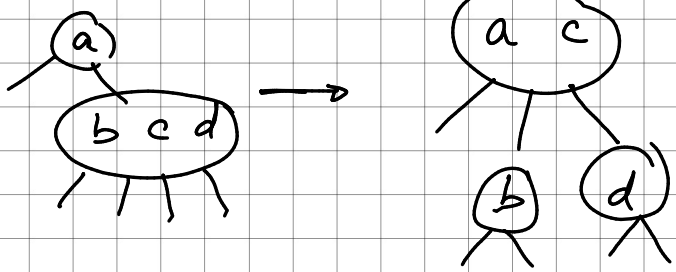


parent is a 2-node

left:

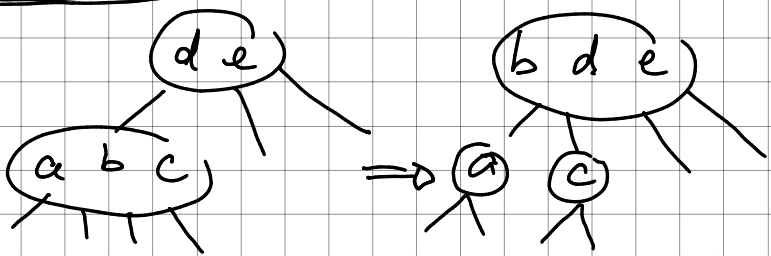


right:

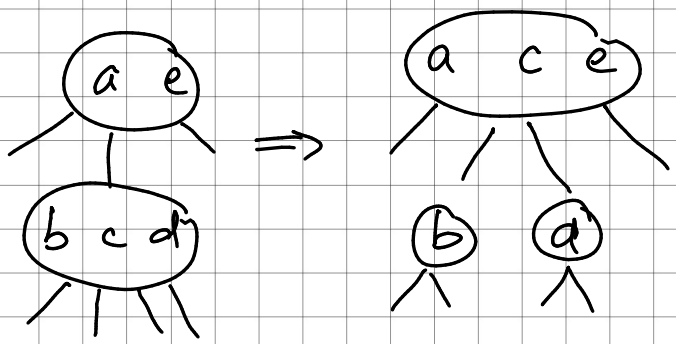


parent is a 3-node

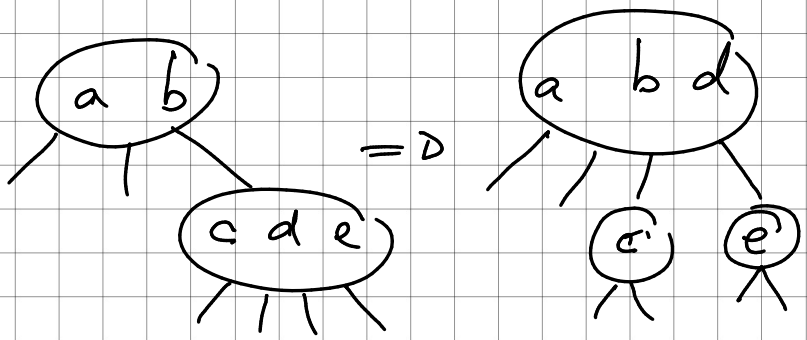
left:



middle:



right:



Performance

- perfect balance: every path from root to null link has same length

Tree height

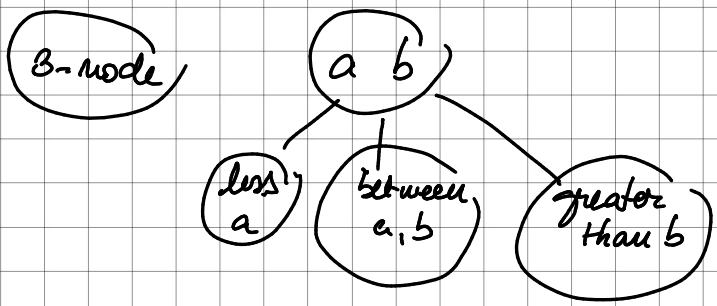
- worst case: all 2-nodes $\log_2 N$
- best case: all 3-nodes $\log_3 N = 0.631 \log_2 N$

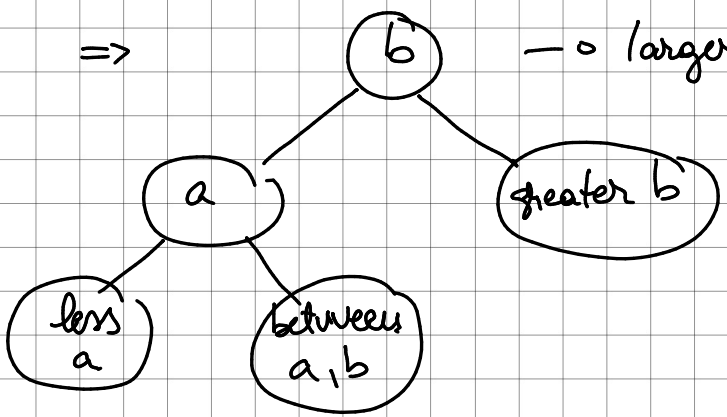
* guaranteed logarithmic performance (search, insert)

Red-Black Binary Search Tree

Left-leaning red-black BSTs

- represent a 2-3 tree as a BST
- use internal left-leaning links as "glue" for 3-nodes





- red links glue nodes within a (3-node)
- black links connect (2-nodes) and (3-nodes)

Red-black BST - a BST such that

- no node has 2 red links connected to it
- every path from root to null link has the same no. of black links ("perfect black balance")
- red links lean left

Operations

1) search - same as BST, ignore color but runs faster because of better balance

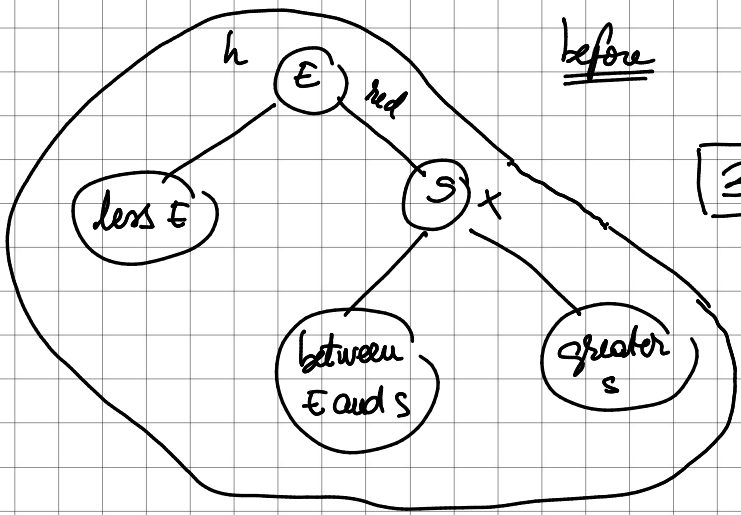
* Representation red-black BST

- each node is pointed to by precisely one link (from its parent) ⇒ can encode color of link in nodes

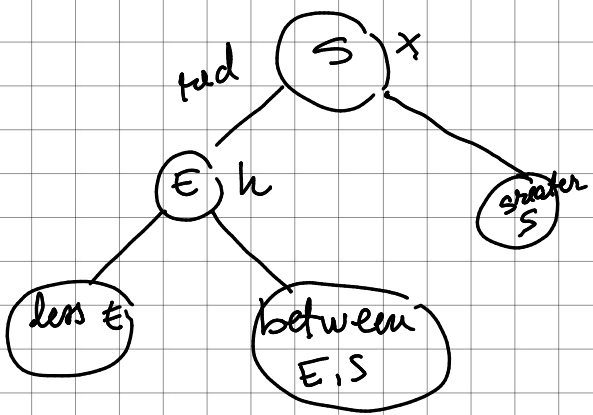
Rotations

1) Left rotations = orient a temporarily right leaning red link to lean left

before



3-node



after

h.right = x.left

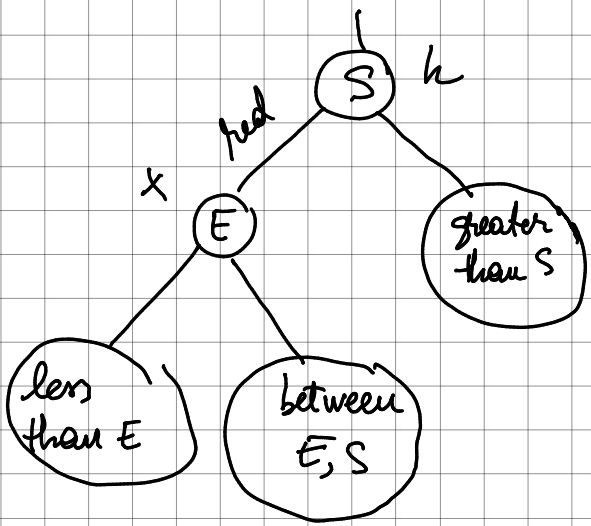
x.left = h

x.color = h.color

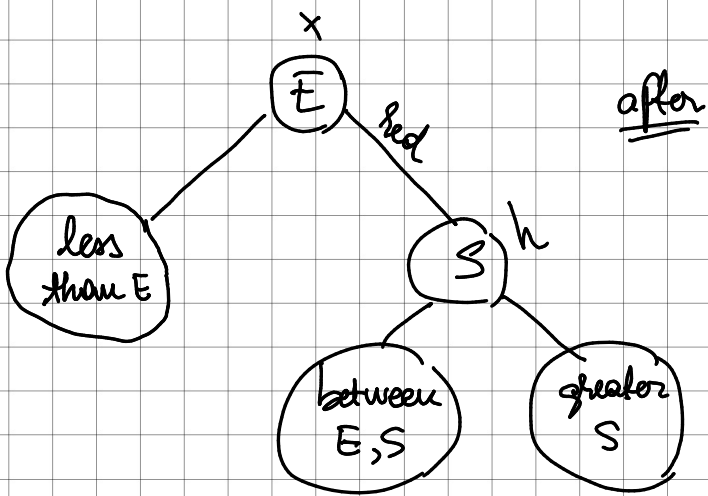
h.color = red

* maintains symmetric order and perfect balance (BLACK)

- Right rotation : orient a left-leaning red link to (temporarily) lean right

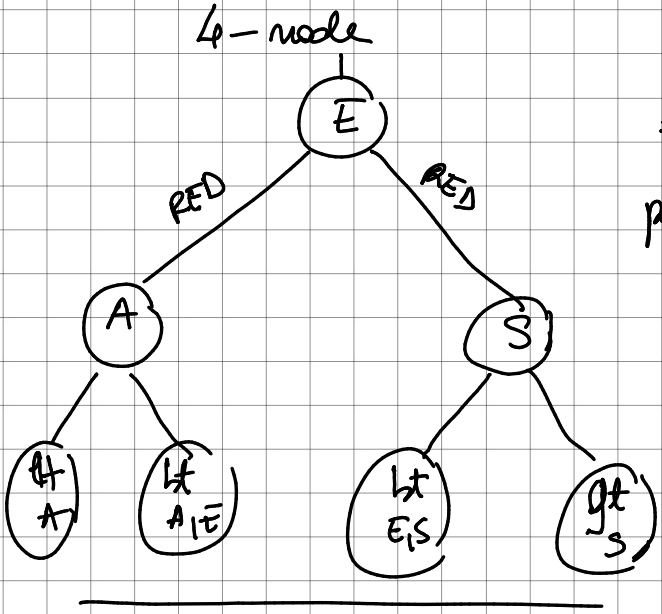


before



after

- Color flip = trick to split a temporarily

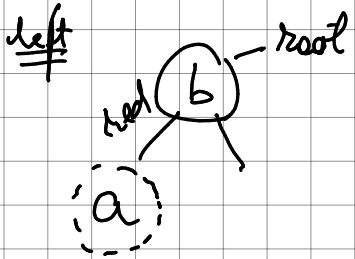


⇒ insert E into its parent by making its color RED

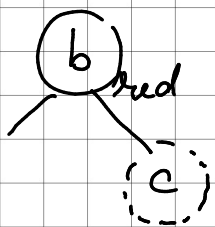
- Insert

1) insert into a tree with exactly one node

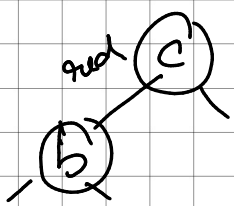
left



right



+ rotate left

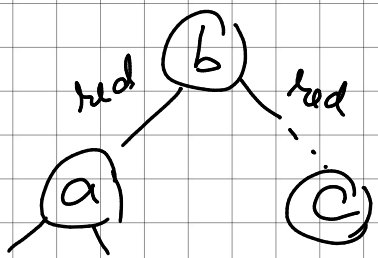


CASE 1: insert into a 2-node at the bottom

- standard BST insert, color link RED
- if new RED link is a right link, rotate left

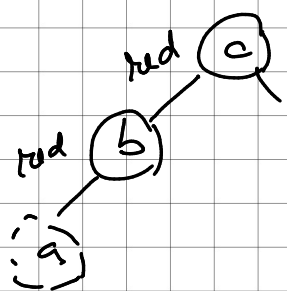
---

o insert into a tree with exactly 2 nodes



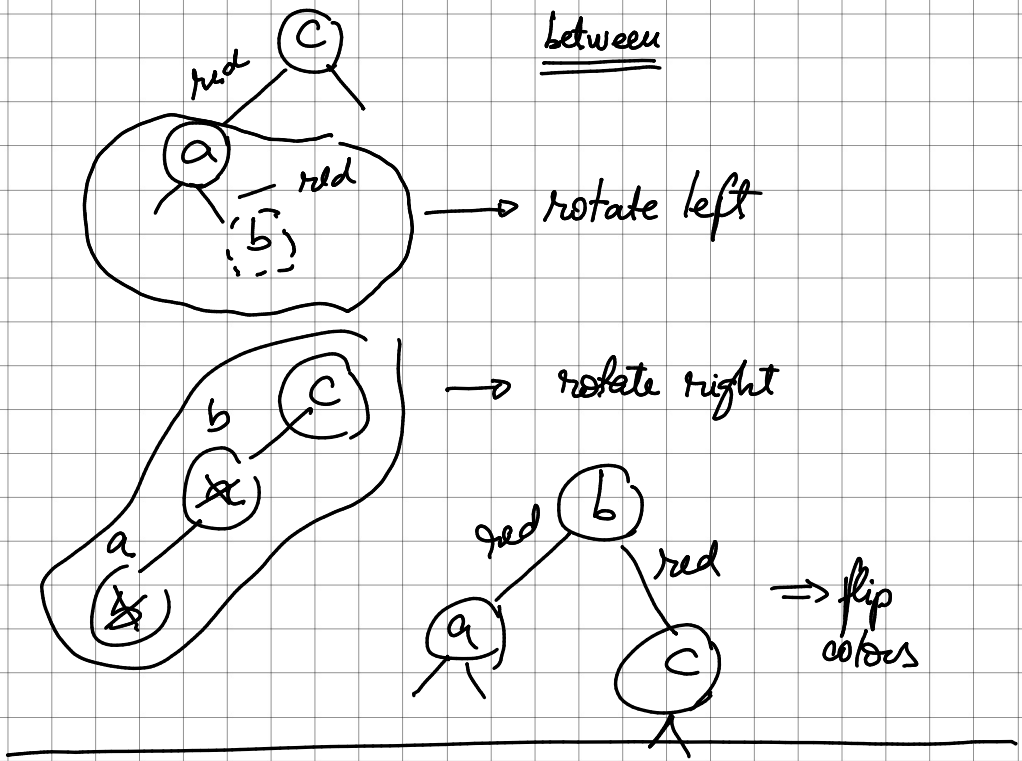
insert new node, color link to red and flip colors

larger



smaller (insert a)

- insert new node, make link red, rotate top node to right and flip colors



CASE 2 : Insert into a (3-node) at the bottom

- standard BST insert ; color new link RED
- rotate to balance the 4-node if needed
- flip colors to pass red link up one level
- rotate to make lean left (if needed)
- repeat case 1 or case 2 up the tree if needed

Java Implementation

- small code handles all cases
  - right child red, left child black
    → rotate LEFT
  - left child, left-left grandchild red:
    → rotate RIGHT
  - both children red → flip colors

Proposition

- height of tree is $\leq 2 \lg N$ in the worst case

PF
- every path from root to null link has the same number of black links
- never two red links in a row

B-trees

- file system model
- page = contiguous block of data
- probe = first access to a page
- property = time required for a probe is much longer than the time to access data within a page
- cost model = no. of probes
- goal = access data using minimum number of probes

* generalize 2-3 trees by allowing up to M-1 key-link pairs per node