

Algorithm

is Exclusive (A, B, C)

(1, 2) A
(3, 2) B

C contain all element in A or B not A and B

~~(2, A) = 2~~

$Dx := \text{new Dict}(HT)$

1, (a)

Load Dictionary (A, A.first(), a, Dx)

3, (b)

Compute XOR (B, p, Dx)

A

if !c.is Empty() then

C = {1, 2}

$Dc := \text{new Dict}(HT)$

B = {1, 2, 3}

Load Dictionary (C, C.first(), c, Dc)

if $Dx.size() \neq Dc.size$ then

< 1, 2, 3

return false

else iter x := Dx.keys()

return

check XOR (iter x, Dx)

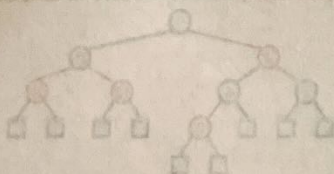
Algorithm compute XOR (B, p, Dx)

be := p.element()

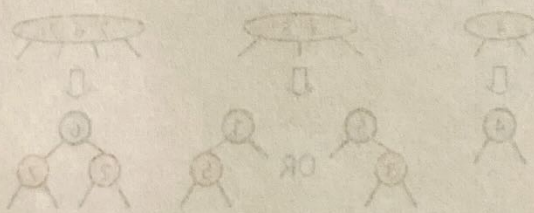
$V := Dx.findValue(be)$

if $V = a$ then

$Dx.remove(p)$



From (2, 4) to Red-Black Trees



Algorithm CheckXOR(iterX, DC).

$e := \text{iterX.next}()$

$v := DC.\text{findValue}(e)$

if $v = \text{NO_SUCH_KEY}$ then.

return false.

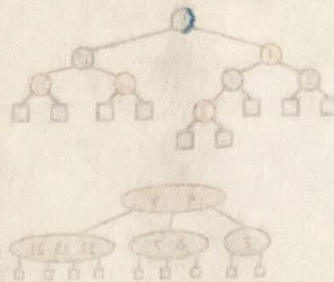
if iterX.hasNext() then.

return checkXOR(iterX, DC).

else.

return true.

Red-Black Tree to (2,4) Tree



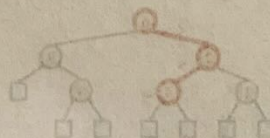
Height of a Red-Black Tree

- Theorem: A red-black tree storing n items has height $O(\log n)$.
- Proof:
 - The height of a red-black tree is at most twice the height of its associated (2,4) tree. (Which is $O(\log n)$).
- The search algorithm for a red-black tree is the same as that for a binary search tree.
- By the above theorem, searching in a red-black tree takes $O(\log n)$ time.

Red-Black Tree Search

Search

- To search for a key k :
 - We trace a downward path starting at the root.
 - At each node, we compare k with the node's key.
 - If we reach a leaf, the key is not found, so we return NO_SUCH_KEY.
 - Otherwise we return the element associated with the leaf.
- Example: findElement()



Helper

findPosition

- To search for a key k , we place a downward path starting at the root.
- The first node visited is based on the comparison of k with the key of the current node.
- If we reach a leaf, the key is not found, so we return the parent of the current node.
- If we find the key, then we return the node.
- Example: findPosition of key 3. It would return node 3.

