Computer Science 331
Red-Black Trees

Mike Jacobson

Department of Computer Science
University of Calgary

Lectures #16-18

Outline

Definition of a Red-Black Tree

A **red-black tree** is a binary tree that can be used to implement the "Dictionary" ADT (also "SortedSet" and "SortedMap" interfaces from the JCF)

- **Internal Nodes** are used to store elements of a dictionary.
- **Leaves** are called "NIL nodes" and do not store elements of the set.
- Every internal node has two children (either, or both, of which might be leaves).
- The smallest red-black tree has size one (single NIL node).
- If the leaves (NIL nodes) of a red-black tree are removed then the resulting tree is a binary search tree.

Red-Black Properties

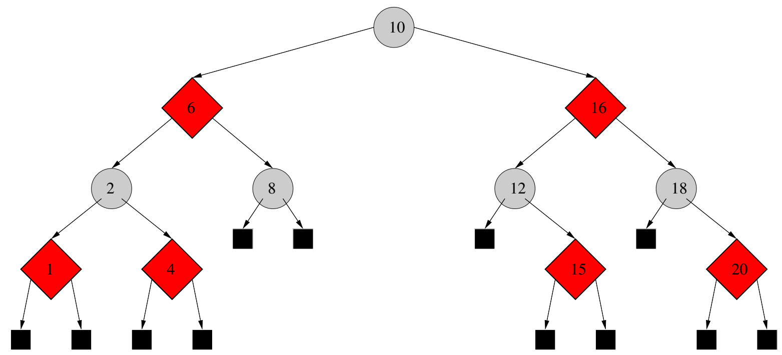
A binary search tree is a *red-black* tree if it satisfies the following:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Why these are useful:

- height is in $\Theta(\log n)$ in the worst case (tree with n internal nodes)
- worst case complexity of search, insert, delete are in $\Theta(\log n)$

Example



- "Black" internal nodes are drawn as circles
- "Red" nodes are drawn as diamonds
- NIL nodes (leaves) are drawn as black squares

Implementation Details

Example: Figure 13.1 on page 275 of the Cormen, Leiserson, Rivest, and Stein book.

- The color of a node can be represented by a Boolean value (eg, true=black, false=red), so that only one bit is needed to store the color of a node
- To save space and simplify programming, a single sentinel can replace all NIL nodes.
- The "parent" of the root node is pointed to the sentinel as well.
- An "empty" tree contains one single NIL node (the sentinel)

Black-Height of a Node

The **black-height** of a node x, denoted $\mathrm{bh}(x)$, is the number of black nodes on any path from, but not including, a node x down to a leaf.

Example: In the previous red-black tree,

- The black-height of the node with label 2 is: 1
- The black-height of the node with label 4 is: 1
- The black-height of the node with label 6 is: 2
- The black-height of the node with label 8 is: 1
- The black-height of the node with label 10 is: 2

Note: Red-Black Property #5 implies that $\mathrm{bh}(x)$ is well-defined for each node x.

The Main Theorem

Theorem 1

If T is a red-black tree with n nodes then the height of T is at most $2\log_2(n+1)$.

Outline of proof:

- prove a *lower bound* on tree size in terms of black-height
- prove an *upper bound* on height in terms of black-height of the tree
- combine to prove main theorem

Bounding Size Using Black-Height

Lemma 2

For each node x, the subtree with root x includes at least $2^{bh(x)} - 1$ nodes.

Method of Proof: mathematical induction on height of the subtree with root x (using the strong form of mathematical induction)

- Base case: prove that the claim holds for subtrees of height 0
- Inductive step: prove, for all $h \geq 0$, that if the lemma is true for all subtrees with height at most $h-1$ then it also holds for all subtrees with height h.

Base Case ($h = 0$)

Suppose that the height $h = 0$:

- x is a leaf (just the NIL node sentinel)
- black-height $\mathrm{bh}(x) = 0$
- number of nodes $n = 1$

Thus, it follows that n satisfies

$$n \geq 2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$$

as required.

Notation for Inductive Step

b	Black-height of x
b_L	Black-height of left child of x
b_R	Black-height of right child of x
T_x	Subtree with root x
h	Height of T_x
h_L	Height of left subtree of T_x
h_R	Height of right subtree of T_x
n	Size of T_x
n_L	Size of left subtree of T_x
n_R	Size of right subtree of T_x

Useful Properties Involving Size and Height

$n = n_L + n_R + 1$. The n nodes of T_x are:

- the n_L nodes of the left subtree of T_x
- the n_R nodes of the right subtree of T_x
- one more node — the root x of T_x

$h = 1 + \max(h_L, h_R)$, so $h_L \leq h - 1$ and $h_R \leq h - 1$

- height of any tree (including T_x) is the maximum length of any path from the root to any leaf
- it follows by this definition that $h = 1 + \max(h_L, h_R)$
- the remaining inequalities are now easily established

Useful Property Involving Black-Height

$b_L \geq b - 1$ and $b_R \geq b - 1$.

Case 1: x has color red

- both children of x have color black (Red-Black Property #4)
- Red-Black Property #5 implies that $b_L = b_R = b - 1$.

Case 2: x has color black.

- children of x could each be either red or black
- $b_L \geq b - 1$, because by the definition of "black-height"

$$b_L = \begin{cases} b & \text{if the left child of } x \text{ is red} \\ b-1 & \text{if the left child of } x \text{ is black.} \end{cases}$$

- an analogous argument shows that $b_R \geq b - 1$

Inductive Step

Let h be an integer such that $h \geq 0$.

Inductive Hypothesis: Suppose the claimed result holds for every node y such that the height of the tree with root y is *less than* h.

Let x be a node such that the height of the tree T_x is h.

Let n be the number of nodes of T_x.

Required to Show: $n \geq 2^{\mathrm{bh}(x)} - 1$ holds for T_x, assuming the inductive hypothesis.

Proof of Inductive Step

The inductive hypothesis implies that

$$n_L \geq 2^{b_L} - 1 \quad \text{and} \quad n_R \geq 2^{b_R} - 1 \ .$$

because h_L and h_R are both $\leq h - 1$.

Thus, the number of internal nodes n satisfies

$$\begin{aligned} n = n_L + n_R + 1 &\geq (2^{b_L} - 1) + (2^{b_R} - 1) + 1 \\ &\geq (2^{b-1} - 1) + (2^{b-1} - 1) + 1 \\ &= 2(2^{b-1}) - 1 \\ &= 2^b - 1 \end{aligned}$$

as required. □

Bounding Height Using Black-Height

Lemma 3

If T is a red-black tree then $bh(r) \geq h/2$ where r is the root of T and h is the height of T.

Proof.

Assume that T has height h :

- by red-black Property 4, at least half the nodes on any simple path from the root to a leaf (not including the root) are black,
- height of T is the length of the longest path from root to a leaf.

Hence, $bh(r) \geq h/2$. □

Proof of the Main Theorem

Theorem 4

If T is a red-black tree with n nodes then the height of T is at most $2\log_2(n+1)$.

Proof.

Let r be the root of T. The two Lemmas state that:

$$n \geq 2^{\mathrm{bh}(r)} - 1 \quad \text{and} \quad bh(r) \geq h/2$$

Putting these together yields:

$$n \geq 2^{h/2} - 1 \quad \Rightarrow \quad \log_2(n+1) \geq h/2 \quad \Rightarrow \quad h \leq 2\log_2(n+1)$$

as required. □

Searching in a Red-Black Tree

Searching in a red-black tree is *almost* the same as searching in a binary search tree.

Difference Between These Operations:

- leaves are NIL nodes that do not store values
- thus, unsuccessful searches end when a leaf is reached instead of when a `null` reference is encountered

Worst-Case Time to Search in a Red-Black Tree:

- $\Theta(\log n)$

Insertion and Deletion?

Unfortunately, *insertions* and *deletions* are more complicated because we need to preserve the "Red-Black Properties."

Main idea: use *rotations* to

- change subtree heights
- preserve binary search tree property

Combination of rotations and other methods can be used to re-establish red-black tree properties after insertions and deletions

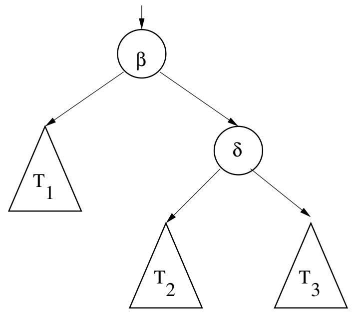
What is a Rotation?

Rotation:

- a local operation on a binary search tree
- preserves the binary search tree property
- used to implement operations on red-black trees (and other height-balanced trees)
- two types:
  - *Left Rotations*
  - *Right Rotations*

Left Rotation: Tree Before Rotation

Tree Before Performing Left Rotation at β:



Assumption: β has a right child, δ

Useful Consequences of Binary Search Tree Property

Lemma 5

For all $\alpha \in T_1$, $\gamma \in T_2$, *and* $\zeta \in T_3$,

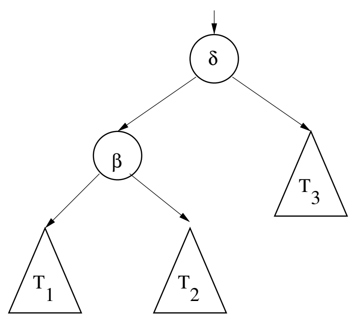
$$\alpha < \beta < \gamma < \delta < \zeta$$

Proof.

T is a BST, so:

- T_1: is the left subtree of β (so $\alpha < \beta$)
- T_2: is contained in the right subtree of β (so $\beta < \gamma$)
  is the left subtree of δ (so $\gamma < \delta$)
- T_3: is the right subtree of δ (so $\delta < \zeta$) $\square$

Left Rotation: Tree After Rotation

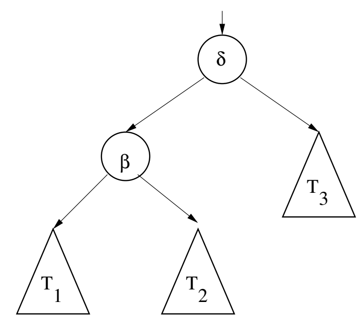


Notice that this is still a BST (inequalities on previous slide still hold)

Pseudocode: *Introduction to Algorithms*, page 278

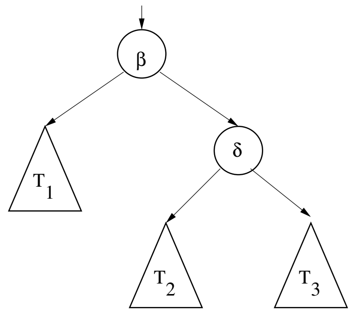
Right Rotation: Tree Before Rotation

Tree Before Performing Right Rotation at δ:



Assumption: δ has a left child, β

Right Rotation: Tree After Rotation



Note: This is both the mirror-image, and the *reversal*, of a left-rotation.

Effects of a Rotation

Exercises:

1. Confirm that a tree is a BST after a rotation if it was one before.
2. Confirm that a (single left or right) rotation can be performed using
$$\Theta(1) \text{ operations}$$
including comparisons and assignments of pointers or references

Beginning an Insertion

Suppose we wish to insert an object x into a red black tree T.

if T includes an object with the same key as x **then**

- throw `FoundException` (and terminate)

else

- Insert a new node storing the object x in the usual way.
  Both of the children of this node should be (black) leaves.
- Color the new node *red*.
- Let z be a pointer to this new node.
- Proceed as described next...

How To Continue

Strategy for Finishing the Operation:

- At this point, T is not necessarily a red-black tree, but there is only a problem at one *problem area* in the tree.
  - newly-inserted node (color red) may violate red-black tree properties #2 or #4
- Rotations and recoloring of nodes will be used to move the “problem area” closer to the root.
- Once the “problem area” has been moved to the root, at most one correction turns T back into a red-black tree.

Structure of Rest of Insertion Algorithm

Recall our assumption from the last lecture: parent of root is a dummy node with color black

Note:

- During the execution of this algorithm, z *always* points to a red node; this is the only place where there might be a problem
- z initially points to the newly-inserted node (color red)

while the parent of z is red **do**
  Make an adjustment (to be described shortly)
end while
if z is the root **then**
  Change the color of z to black
end if

Loop Invariant

z is red and **exactly one** of the following is true:

(A) The parent of z is also red.
All other red-black properties are satisfied.

(B) z is the root.
All other red-black properties are satisfied.

(C) All red-black properties are satisfied.
Thus T is a red-black tree.

Note: Loop invariant + failure of loop test $\Rightarrow$ B or C.

Loop Variant

Loop Variant: depth of z

Consequence:

- number of executions of loop body is linear in the height of T.

Note:

- We will need to check that this is a loop variant!
- This is the case if z is moved closer to the root after every iteration.

Subcases of Case A

Note: Since the parent of z is red it is not the root; the *grandparent* of z must be black.

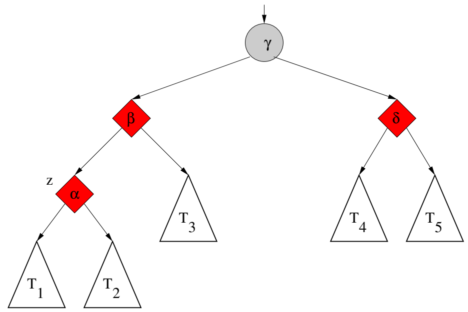
1. Parent of z is a left child; sibling y of parent of z is red.
   (a) z is a left child.
   (b) z is a right child.
2. Parent of z is a left child; sibling y of parent of z is black.
   z is a right child.
3. Parent of z is a left child; sibling y of parent of z is black.
   z is a left child.

Subcases 4–6: Mirror images of subcases 1–3:

- Exchange "left" and "right;" parent(z) is now a right child

Subcase 1a: Tree Before Adjustment

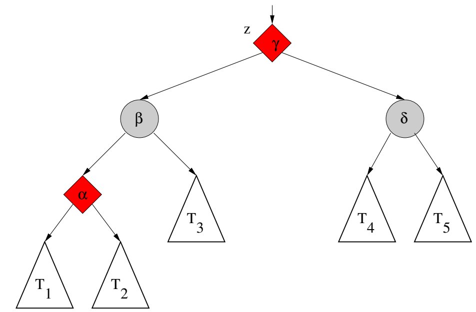
z is left child, parent of z is a left child; sibling y of parent of z is red



Adjustment:

- Recolor β, γ, δ; point z to its grandparent.

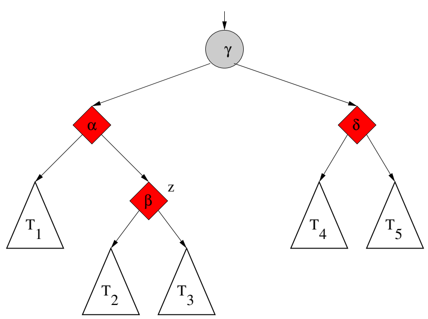
Subcase 1a: Tree After Adjustment



Node z may still cause violations of red-black tree properties #2 or #4, but z has moved closer to the root.

Subcase 1b: Tree Before Adjustment

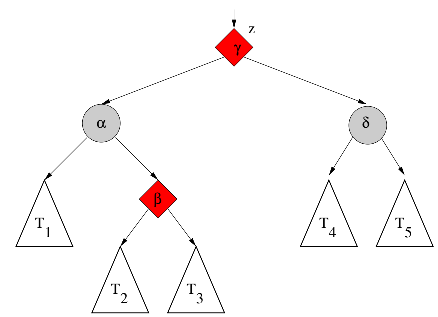
z is right child; parent of z is a left child; sibling y of parent of z is red;



Adjustment:

- Recolor α, γ, δ; point z to its grandparent.

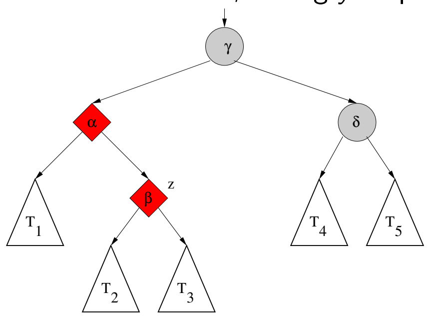
Subcase 1b: Tree After Adjustment



Node z may still cause violations of red-black tree properties #2 or #4, but z has moved closer to the root.

Case 2: Tree Before Adjustment

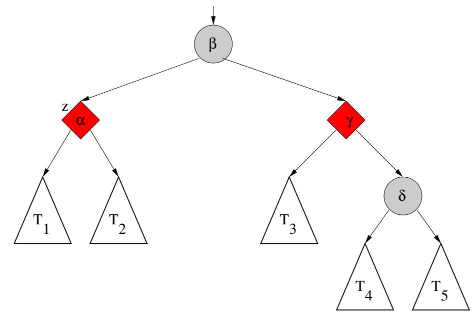
z is right child; parent of z is left child; sibling y of parent of z is black;



Adjustment:

- Point z to its parent. Rotate left at α
- Rotate right at γ, recolor β and γ.

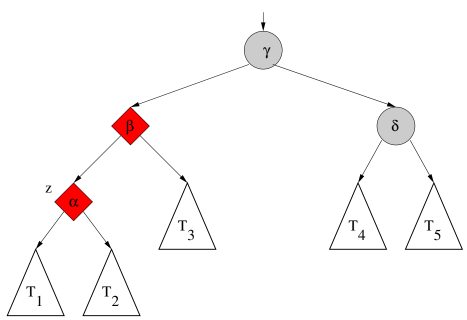
Case 2: Tree After Adjustment



Parent of z is now black, so the while loop terminates and we are finished.

Case 3: Tree Before Adjustment

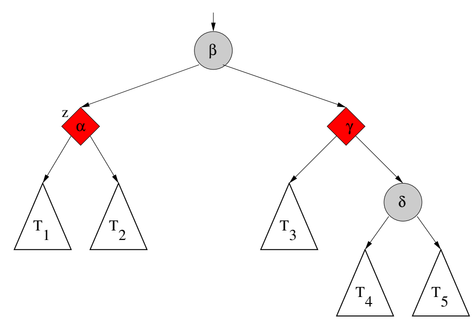
z is left child; parent of z is left child; sibling y of parent of z is black;



Adjustment:

- Rotate right at γ; recolor β and γ

Case 3: Tree After Adjustment



Parent of z is now black, so the while loop terminates and we are finished.

Exercises

3. Describe cases 4–6 and draw the corresponding trees.
4. Confirm that the "loop invariant" holds after each adjustment.
5. Confirm that the distance of z from the root decreases after each adjustment — so the claimed "loop variant" satisfies the properties it should.

Handling Cases B and C

Case B: z is the root (so, the root is red)

- *All other red-black properties are satisfied.*
- *Adjustment:* change the color of the root to black.

Case C: T is a red-black tree.

- *Adjustment:* We're finished!

Pseudocode for adjustments: *Introduction to Algorithms*, page 281

Exercises:

6. Show that the "insertion" algorithm as a whole is correct.
7. Confirm that the total number of steps used by the insertion algorithm is at most linear in the depth of the given tree.

Beginning of a Deletion

Suppose we wish to delete an object with key k from a red black tree T.

if T does not include an object with key k **then**

- T is not modified; throw `KeyNotFoundExcepction` and terminate

else

- Ignore the NIL nodes (for now)
- Consider what would happen if the "standard" algorithm was applied
- Let y point the the *node* **that would be deleted**

Clarification: What is y?

Specifically . . .

- If at least one child of the object storing k is a leaf (that is, a NIL node) then y is the node storing k
- Otherwise y is the node storing *the smallest key in the right subtree with the node storing k as root*

Please review the description of deletion of a node from a regular binary search tree if this is not clear!

Case 1: Deleted Node y was Red

Situation:

- At least one child of y is a NIL node (because of the choice of y)
- y and a NIL child can be discarded, with the other child of y promoted to replace y in T
- Then T is still a red black tree. $\Longrightarrow$ We are finished!

Exercise: Confirm that T really is still a red-black tree after a red node has been removed (in the usual way).

The rest of the lecture concerns the case that the deleted node y was black.

Case 2: Deleted Node was Black

Suppose we deleted (as described above) a black node y

Let x be the node that will be “promoted” to replace y. We have the following possibilities:

- Both children of y are NIL nodes
  $\Longrightarrow$ x is a single NIL node that replaces both of these.
- One child of y is a NIL node
  $\Longrightarrow$ x is the other child (ie, the child of y that is *not* NIL)
- Neither child of y is a NIL node
  $\Longrightarrow$ This case is impossible (because of the choice of y)

Possible Problems

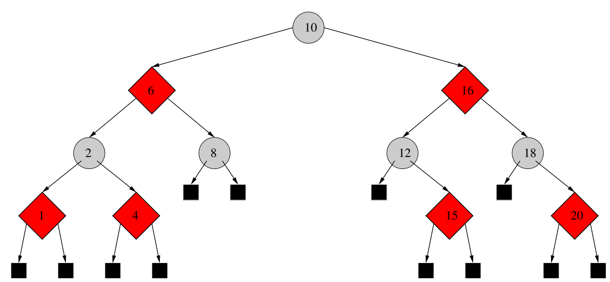
1. Paths from nodes to leaves that included y are now missing one black node $\Longrightarrow$ black-height is not well-defined!
2. It is possible that either
   - x and its parent might both be red, or
   - x might be red *and* be the root

   (Note that both cannot be true at the same time.)

There can be no other problems with the tree (yet!)

The rest of the notes are about how to correct these problems.

Example



Possible cases for x :

- delete 1 : $x = NIL$ (no problems!)
- delete 8 : $x = NIL$ (black height problem)
- delete 18 : $x = 20$ (black height problem)
- delete 6 : $x = NIL$ (black height problem)
- delete 10 : $x = 15$ (black height problem)

Initialization: Fixing "Black-Height"

Fixing Black-Height: Add two more kinds of nodes, to define black-height once again

Red-Black Node
Count as *one* black node on a path when computing black-height.

Double-Black Node
Count as *two* black nodes on a path when computing black-height.

In practice, can use a flag called, for example, "fixupRequired" to denote the "extra" black colour.

Initialization: Fixing "Black-Height" (cont.)

Set the new type of x to be

- Red-Black (if x was a red child of the deleted black node)
- Double-Black (if x was a black child of a deleted black node)

Note: "Black-height" of nodes are well-defined again after this change!

Possible Cases, At This Point:

1. x is a red-black node.
2. x is a double-black node at the root.
3. x is a double-black node, not at the root.

In each case, there are no other problems in the tree.

Two of These Cases are Easy!

Case 1: x is a red-black node.

- Change x to a black node, and stop
- **Exercise:** confirm that T is a red-black tree after this change.

Case 2: x is a double-black node at the root.

- Change x to a black node, and stop
- **Exercise:** confirm that T is a red-black tree after this change.

Pseudocode to Finish Deletion of a Black Node

Pseudocode to finish deletion if a black node was deleted and x points to child being promoted:

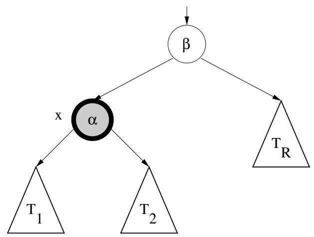
Change the type of x as described above.

while x is double-black and not at the root **do**
  Make an adjustment as described next
end while

if x is red-black or at the root **then**
  Change x to a black node
end if

Expanding the Remaining Case

One Major Subcase: x is the left child of its parent (β red or black)

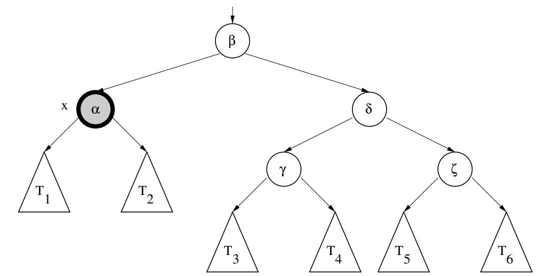


Another Major Subcase: x is the right child of its parent.

The first of these subcases will be described in detail. The algorithm for the second is almost identical.

Expanding the First Subcase

Note: Black-height of β is at least two (Property #5)



Various possibilities (depends on color of sibling of x)

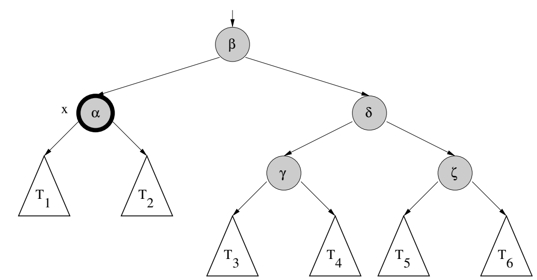
Further Breakdown of Subcases

Case	β	γ	δ	ζ
3a	black	black	black	black
3b	red	black	black	black
3c	black	black	red	black
3d	?	red	black	black
3e	?	?	black	red

Exercise: Check that these cases are pairwise exclusive and that no other cases are possible.

Case 3a: Before Adjustment

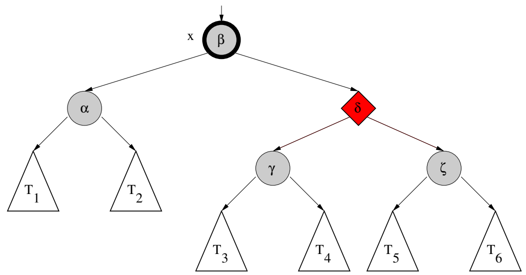
Case 3a: β, γ, δ, ζ all black. Goal: move x closer to root.



Adjustment:

- Change colors of α, β, and δ; x points to its parent

Case 3a: After Adjustment

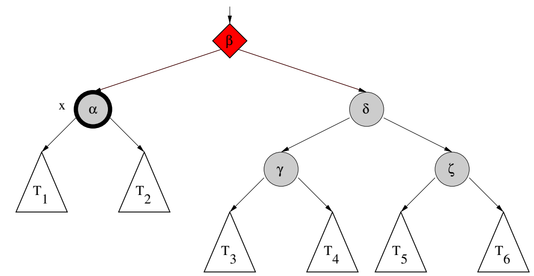


After the adjustment:

- All cases are now possible; x is closer to the root.

Case 3b: Before Adjustment

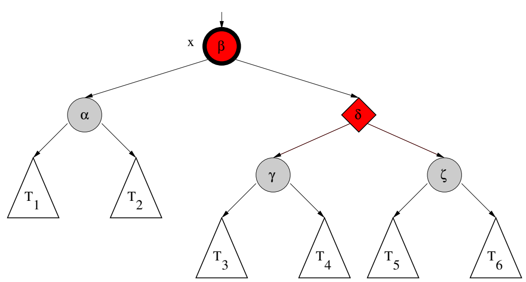
Case 3b: β red; γ, δ, ζ black. Goal: finish after this case.



Adjustment:

- Change colors of α, β, and δ; x points to parent.

Case 3b: After Adjustment

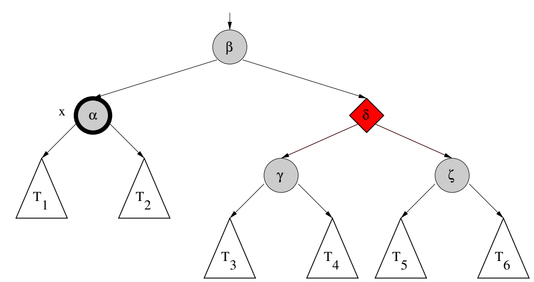


After the adjustment:

- None of the cases apply (loop terminates, x changed to black)

Case 3c: Before Adjustment

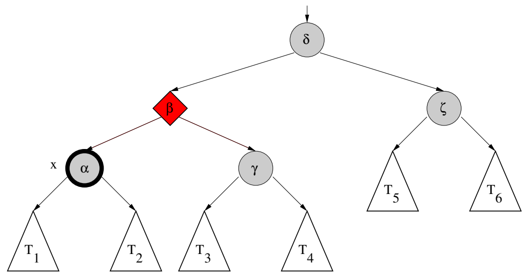
Case 3c: δ red; β, γ, ζ black. Goal: transform parent of x to red.



Adjustment:

- left rotation at β
- change colors of β and δ

Case 3c: After Adjustment

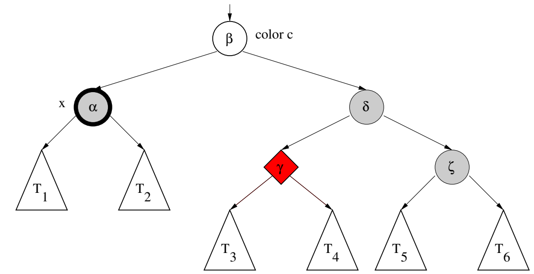


After the adjustment:

- x has not moved, but cases 3b, 3d, or 3e may now apply.

Case 3d: Before Adjustment

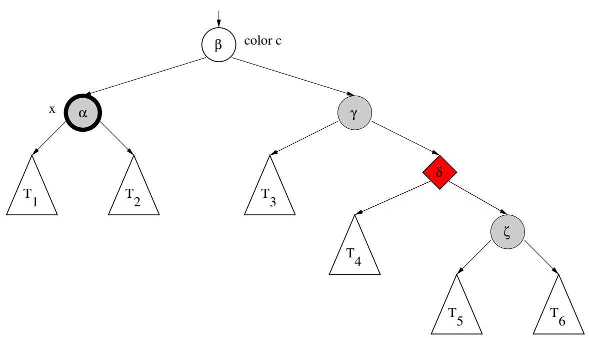
Case 3d: γ red; δ and ζ black. Goal: transform to Case 3e.



Adjustment:

- right rotation at δ
- change colors of γ and δ

Case 3d: After Adjustment

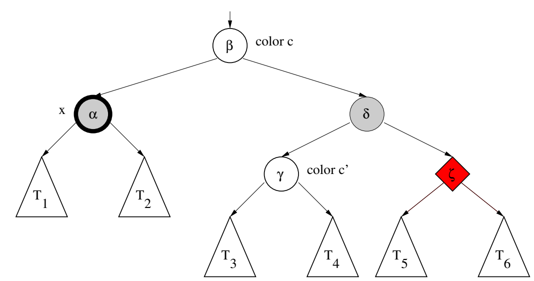


After the adjustment:

- x has not moved, but case 3e now applies.

Case 3e: Before Adjustment

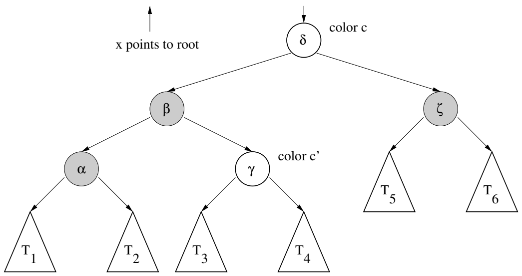
Case 3e: δ is black; ζ is red. Goal: finish after this case.



Adjustment:

- left rotation at β
- recolor α and ζ
- switch colors of β and δ; x will point to the root of the tree.

Case 3e: After Adjustment



After the adjustment:

- Result is a red-black tree!

Other Major Subcase: x is a Right Child

3f: Mirror Image of 3a
3g: Mirror Image of 3b
3h: Mirror Image of 3c
3i: Mirror Image of 3d
3j: Mirror Image of 3d

In each case, the "mirror image" is produced by exchanging the left and right children of β and of δ

Loop Invariant (Elimination of Double-Black Node)

Exactly one of the following cases applies:

- T is a red-black tree,
- x is a red-black node (no other problems),
- x is a double-black node at the root (no other problems),
- Exactly one of cases 3a–3j applies (no other problems).

Exercise: verify that this is in fact a loop invariant

Loop Variant (Elimination of Double Black Node)

Consider the function that is defined as follows.

Case	Function Value
Red-Black Tree	0
x is red-black	0
x is at root	0
Case 3a or 3f	$\text{depth}(x) + 4$
Case 3b or 3g	1
Case 3c or 3h	3
Case 3d or 3i	2
Case 3e or 3j	1

Exercise: Show that this is a loop variant

- total cost linear in height of the tree

Reference

Main reference:

Introduction to Algorithms, Chapter 13

Note: In the above reference, cases are named and grouped differently to provide more compact pseudocode — but the result may be (even more) confusing.

Additional Reference:

Data Structures & Algorithms in Java, Chapter 9