Computer Science 331
Red Black Trees: Deletions

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #18

Outline

Red-Black Properties

Recall that the following properties must be maintained (along with the binary-search properties) when a deletion from a red-black tree is performed:

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.

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

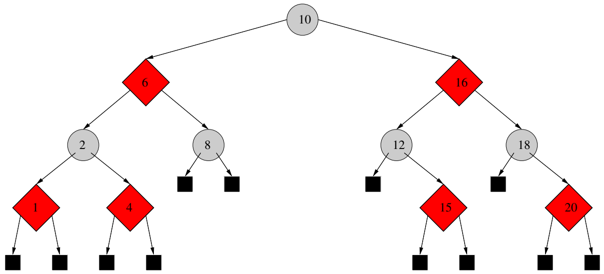
1. Paths from nodes to leaves that included y are now missing one black node $\Longrightarrow$ black-height is not well-defined!
2. It is possible that either
   - x and its parent might both be red, or
   - x might be red *and* be the root

   (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:

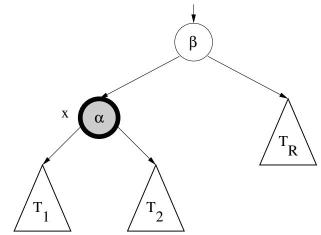
Change the type of x as described above.

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)

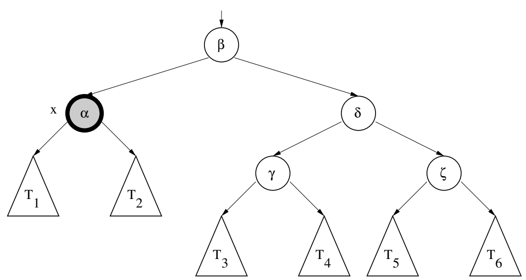


Another Major Subcase: x is the right child of its parent.

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)

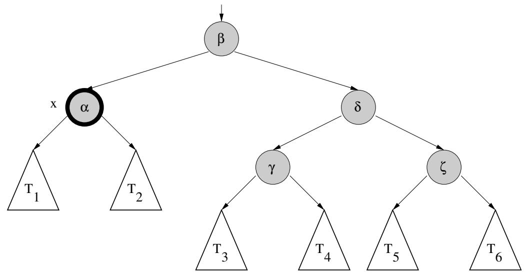
Further Breakdown of Subcases

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

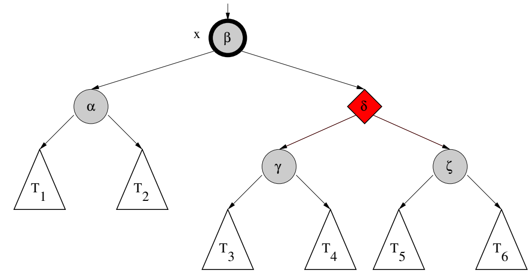
Case 3a: β, γ, δ, ζ all black. Goal: move x closer to root.



Adjustment:

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

Case 3a: After Adjustment

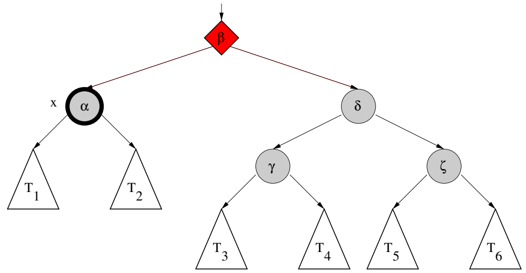


After the adjustment:

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

Case 3b: Before Adjustment

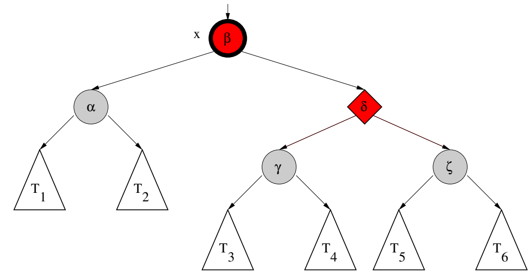
Case 3b: β red; γ, δ, ζ black. Goal: finish after this case.



Adjustment:

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

Case 3b: After Adjustment

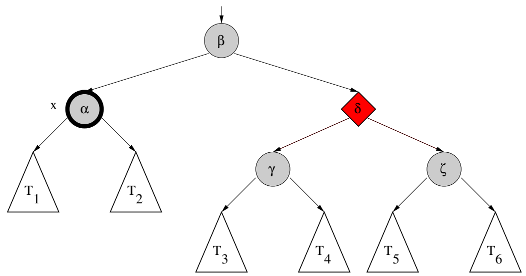


After the adjustment:

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

Case 3c: Before Adjustment

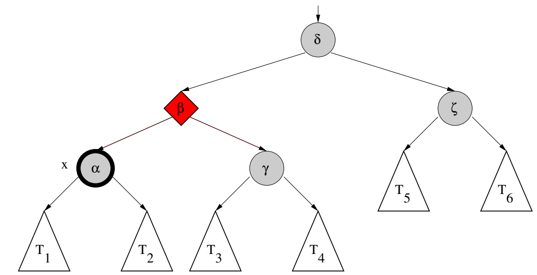
Case 3c: δ red; β, γ, ζ black. Goal: transform parent of x to red.



Adjustment:

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

Case 3c: After Adjustment

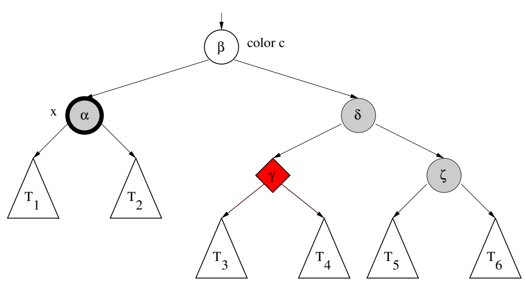


After the adjustment:

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

Case 3d: Before Adjustment

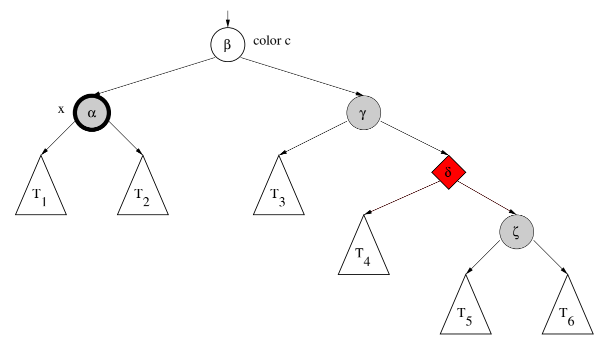
Case 3d: γ red; δ and ζ black. Goal: transform to Case 3e.



Adjustment:

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

Case 3d: After Adjustment

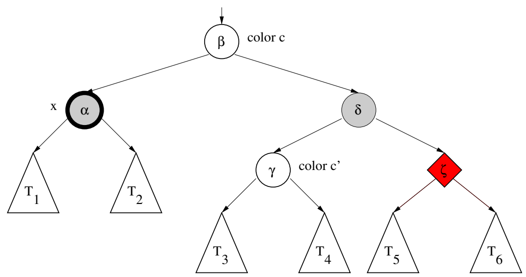


After the adjustment:

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

Case 3e: Before Adjustment

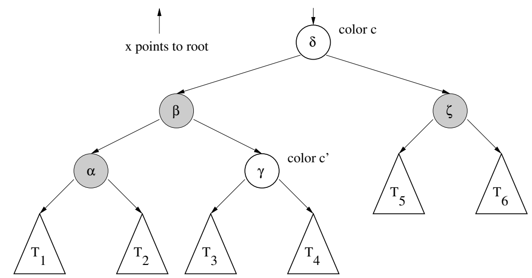
Case 3e: δ is black; ζ is red. Goal: finish after this case.



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

Please consult

Introduction to Algorithms, Chapter 13

for additional information about red-black trees.

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