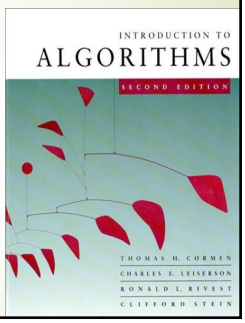
Red black trees

SUBJECT-CODE : KCS-503

Dr. Ragini Karwayun



1

RED BLACK TREES

- **Red-black trees** are a variation of binary search trees to ensure that the tree is **balanced.**
- A red-black tree has one extra bit of storage per node: its color, which can be either red or black.
- By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.
- Height is O(lg n), where n is the number of nodes.
- Operations take O(lg n) time in the worst case.

2

RED BLACK TREES

- Binary search tree + 1 bit per node: the attribute *color*, which is either red or black.
- All other attributes of BSTs are inherited:
  - *Key*, *left*, *right*, and *p*.
- Each node of the tree contains 5 fields :
  - Color
  - Key
  - Left
  - Right
  - P(parent)
- All empty trees (leaves) are colored black.
- We use a single sentinel, *nil*, for all the leaves of red-black tree *t*, with
  - *color*[*nil*] = black.
- The root's parent is also *nil*[*t*].

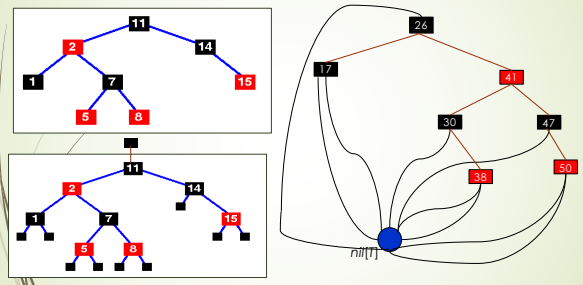
3

Properties of Red Black Tree

- **Property #1:** every node is either red or black.
- **Property #2:** the root node must be colored black.
- **Property #3:** every leaf (e.i. NULL node) must be colored BLACK.
- **Property #4:** the children of red colored node must be colored black. (There should not be two consecutive RED nodes).
- **Property #5:** in all the paths of the tree there must be same number of black colored nodes.
- **Property #6:** every new node must be inserted with red color.

4

Red Black Tree - Example



5

Properties of Red Black Tree

Remember

Every internal node has two children, even though nil leaves are not usually shown.

Nil[T] – It is a dummy object that allows us to simplify boundary conditions

Every Red Black Tree is a binary search tree but all the Binary Search Trees need not to be Red Black trees.

6

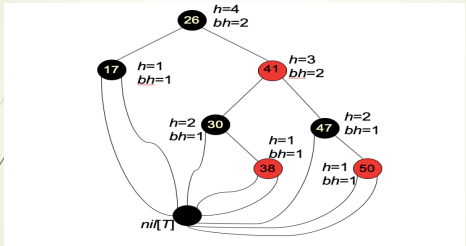
HEIGHT OF A RB-TREE

- Height of a node:
  - h(x) = number of edges in a longest path to a leaf.
- Black-height of a nodes (node x : bh(x))
  - bh(x) = number of black nodes including nil[t]) on the path from x to leaf, excluding x.
- Black-height of a red-black tree is the black-height of its root.
  - By property 5, black height is well defined.
- How are they related?

bh(x) ≤ h(x) ≤ 2 bh(x)

7

HEIGHT OF A RB-TREE



8

Bound on RB-TREE HEIGHT

- **Lemma : A red-black tree with n internal nodes has height (at most) ≤ 2 lg(n+1).**
- **Proof:**
  - *Show that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.*
  - $n \geq 2^{bh} - 1$,
  - And since $bh \geq h/2$,
  - We have $n \geq 2^{h/2} - 1$
  - $n+1 \geq 2^{h/2}$

Now by taking log on both sides

$\lg(n+1) \geq \lg 2^{h/2}$

$\Rightarrow$ Lg (n+1) ≥ (h/2) lg 2

$\Rightarrow$ 2 lg (n+1) ≥ h

$\Rightarrow$ $\boldsymbol{h \leq 2 \lg(n+1)}$.

9

Important proofs

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

- By property 5 the longest and shortest path must contain the same number of black nodes.
- By property 5 the shortest path may have every node black.
- By property 4 every other node in the longest path must be black, that is at least half of the nodes on any simple path from the root to a leaf, not including the root, must be black and therefore the length is at most twice that of the shortest path.
- So we can say that **h/2 <= bh <= h**

10

Important proofs

What is the largest possible number of internal nodes in a red-black tree with black- height k? What is the smallest possible number?

- Consider a red-black tree with black-height k. If every node is black the total number of internal nodes is $2^k - 1$. If only every other nodes is black we can construct a tree with $2^{2k} - 1$ or $4^k - 1$ nodes

(ii) **smallest possible number of internal nodes n = $2^k - 1$**

- If black height = 0 then n = $2^0 - 1 = 0$
- A node with black height k will have two black children with black height k-1.
- Let us assume that it is true for k-1 height. Then,
- n = $2^{k-1} - 1 + 2^{k-1} - 1 + 1$(for the root node itself)
- n= $2^{k-1+1} - 2 + 1$ = $\mathbf{2^k - 1}$ …… proved

smallest tree with black height = 2

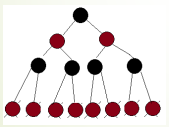
11

Important proofs

What is the largest possible number of internal nodes in a red-black tree with black- height k? What is the smallest possible number?

(ii). largest possible number of nodes with black height k , n = $4^k - 1$

- A root node with black height k will have two red children with black height k. Each of these two nodes will have two black children with black height k-1.
- Therefore, n = $4^{k-1} - 1 + 4^{k-1} - 1 + 4^{k-1} - 1 + 3$ (two red nodes + 1 root node)
- n = $4^* 4^{k-1} - 1 - 4 + 3 = 4k\text{-}1+1 - 1 = \mathbf{4^k - 1}$ …… Proved



largest tree with black height = 2

12

Important proofs

Argue that in any n-node binary search tree there are exactly n-1 rotations.

- Any BST starts out with at least one node, the root in the right side. Thus we will require n-1 rotations to shift all the nodes on one side.

Can the black height of nodes in RB tree maintained as fields in the nodes of the tree without affecting the asymptotic performance of any of the RBT operation.

- Yes. Because the black height of a node can be computed from the information at the node and its two children. Actually, the black height of a node can be computed from just one child's info. The black height of a red child , or black height of a black child + 1.

13

Insertion in RB Trees

- Insertion must preserve all red-black properties.
- Newly inserted node is always colored RED
- Basic steps:
  - Use tree-insert from BST (slightly modified) to insert a node z into T.
    - Procedure rb-insert(T,z).
  - Color the node z red.
  - Fix the modified tree by re-coloring nodes and performing rotation to preserve rb tree property.
    - Procedure **rb-insert-fixup(T,z).**

14

Insertion in RB Trees

Rb-insert(T, z)

```
1.  y ← nil[T]
2.  x ← root[T]
3.  While x ≠ nil[T]
4.     do y ← x
5.        if key[z] < key[x]
6.           Then x ← left[x]
7.           Else x ← right[x]
8.  p[z] ← y
9.  If y = nil[T]
10.    then root[T] ← z
11.    else if key[z] < key[y]
12.       Then  left[y] ← z
13.       Else right[y] ← z
```

RB-Insert(T, z) Contd.

```
14.  left[z] ← nil[T]
15.  right[z] ← nil[T]
16.  color[z] ← red
17.  RB-Insert-Fixup (T, z)
```

How does it differ from the Tree-Insert procedure of BSTs?

Which of the RB properties might be violated?

Fix the violations by calling RB-Insert-Fixup.

15

Insertion Fixup

Rb-insert-fixup (T, z)

```
1.  While color[p[z]] = RED
2.     do if p[z] = left[p[p[z]]]
3.        Then y ← right[p[p[z]]]
4.             If color[y] = RED
5.                Then color[p[z]] ← BLACK      // case 1
6.                     Color[y] ← BLACK         // case 1
7.                     Color[p[p[z]]] ← RED     // case 1
8.                     z ← p[p[z]]              // case 1
```


16

Insertion Fixup

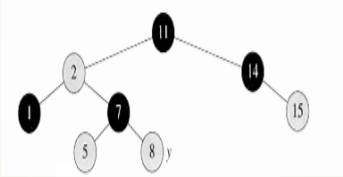
Rb-insert-fixup(*T, z*) (contd.)

```
9.      Else if z = right[p[z]]      // if color[y] ≠ red
10.         Z ← p[z]                 // case 2
11.         LEFT-ROTATE(T, z)        // case 2
12.     Color[p[z]] ← BLACK          // case 3
13.     Color[p[p[z]]] ← RED         // case 3
14.     RIGHT-ROTATE(T, p[p[z]])     // case 3
15.  Else (if p[z] = right[p[p[z]]])
16.     (same as 10-14. With "right" and "left"
          exchanged)
17. Color[root[T ]] ← BLACK
```


17

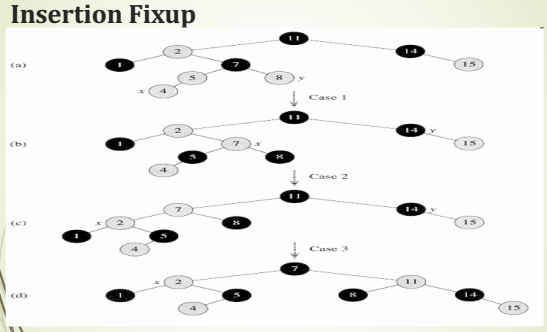
Insertion Fixup

Insert z=4 in the Initial tree and restore the red black properties.



18

Insertion Fixup



19

Insertion & fixup in RB Trees

1. Initialization

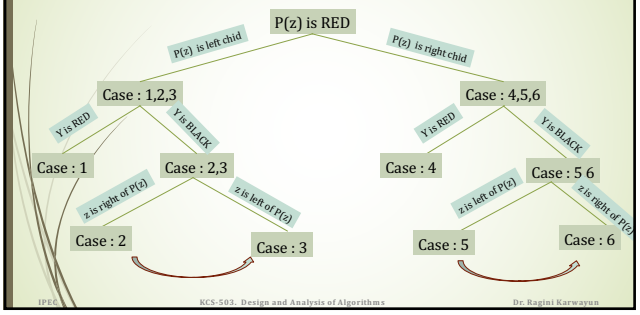
   z ← New node T ← Tree P(z) ← Parent of z y ← Uncle of z

   P(P(z)) ← Grandparent of z RED ← color of z

2. If tree T is empty, insert z as the root node, color it black and exit. [Property 2 violation]
3. If tree T is non empty, insert the new node z as a leaf node according to the properties of BST. [there is at most 1 RED-BLACK violation]. [Property 4]
4. If color of P(z) is BLACK → Exit
5. If color of P(z) is RED then proceed ---

20

Insertion & fixup in RB Trees



21

Case 1 – uncle *y* is red

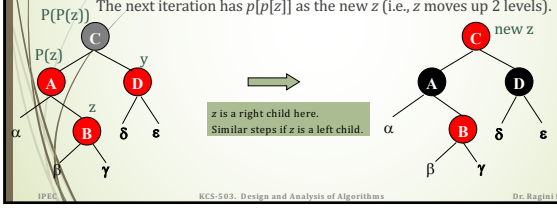
1. P(z) is left of P(P(z)) and z's uncle y is RED

Actions:

Color $p[z]$ and y = black (prop 5 violated)

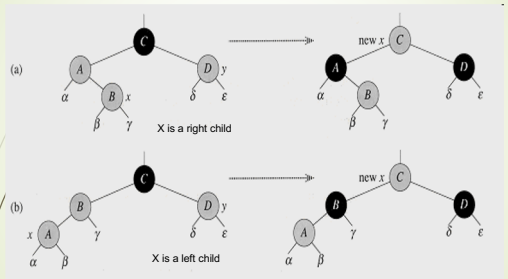
Color $p[p[z]]$ = red (restores property 5)

The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).



22

Case 1 – uncle *y* is red



23

Case 2 – *y* is black, *z* is a right child

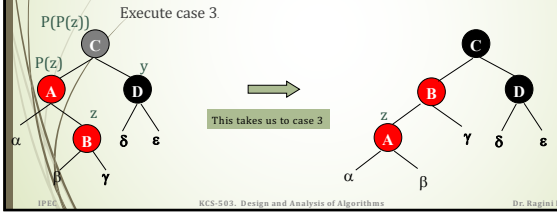
2. P(z) is left of P(P(z)) and z's uncle y is BLACK and z is right child of P(z)

Actions:

Move z to P [z]

Left rotate around $p[z]$ (or new z)

Execute case 3.

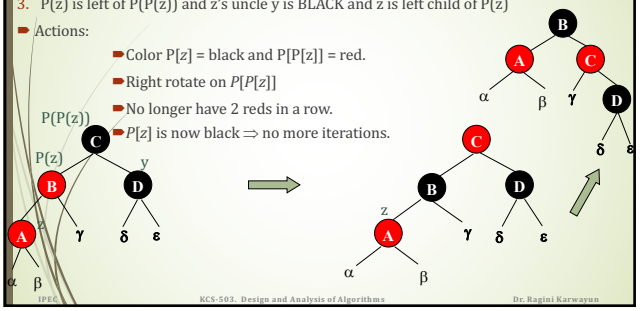


24

Case 3 – *y* is black, *z* is a left child

3. P(z) is left of P(P(z)) and z's uncle y is BLACK and z is left child of P(z)

- Actions:
  - Color P[z] = black and P[P[z]] = red.
  - Right rotate on *P*[*P*[*z*]]
  - No longer have 2 reds in a row.
  - *P*[*z*] is now black ⇒ no more iterations.

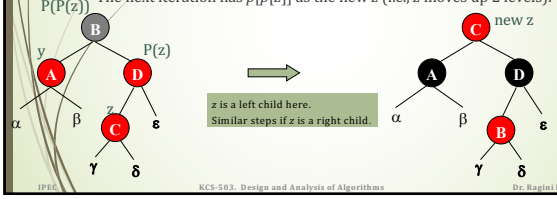


25

Case 4 – uncle *y* is red

4. P(z) is right of P(P(z)) and z's uncle y is RED

- Actions:
  - Color *p*[*z*] and *y* = black (prop 5 violated)
  - Color *p*[*p*[*z*]] = red (restores property 5)
  - The next iteration has *p*[*p*[*z*]] as the new *z* (i.e., *z* moves up 2 levels).



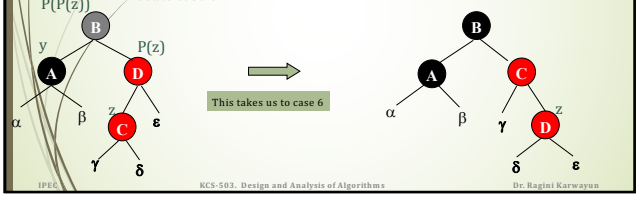
z is a left child here.
Similar steps if z is a right child.

26

Case 5 – *y* is black, *z* is a left child

2. P(z) is right of P(P(z)) and z's uncle y is BLACK and z is left child of P(z)

- Actions:
  - Move z to P [z]
  - right rotate around *p*[*z*] (or new z)
  - Execute case 6



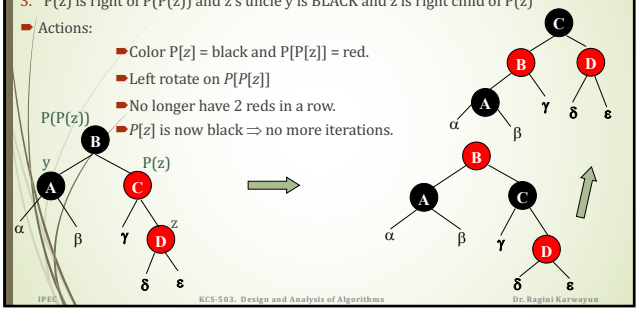
This takes us to case 6

27

Case 6 – *y* is black, *z* is a right child

3. P(z) is right of P(P(z)) and z's uncle y is BLACK and z is right child of P(z)

- Actions:
  - Color P[z] = black and P[P[z]] = red.
  - Left rotate on *P*[*P*[*z*]]
  - No longer have 2 reds in a row.
  - *P*[*z*] is now black ⇒ no more iterations.



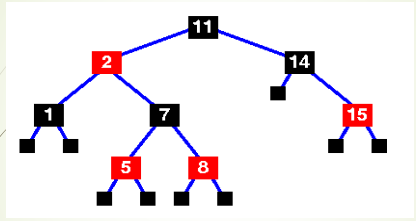
28

Algorithm Analysis

- $O(\lg n)$ time to get through rb-insert up to the call of rb-insert-fixup.
- Within rb-insert-fixup:
  - Each iteration takes $O(1)$ time.
  - Each iteration but the last moves z up 2 levels.
  - $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
  - Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  - Note: there are at most 2 rotations overall.

29

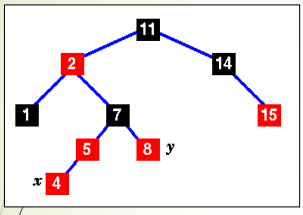
Example of RBT insertion



The original tree ..
Note that in the following diagrams, the black sentinel nodes have been omitted to keep the diagrams simple.

30

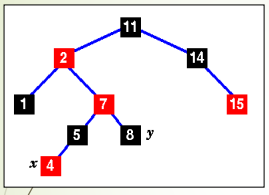
Example of RBT insertion



- **The tree insert routine has just been called to insert node "4" into the tree.**
- **This is no longer a red-black tree – there are two successive red nodes on the path 11 - 2 - 7 - 5 - 4**
- **Mark the new node, x, and it's uncle, y.**
- **y is red, so we have case 1 ...**

31

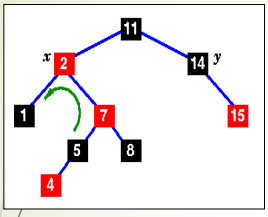
Example of RBT insertion



Action : case 1
- **Color $p[x]$ and y = black (prop 5 violated)**
- **Make $p[p[x]]$ = red (restores property 5)**
- **The next iteration has $p[p[x]]$ as the new x (i.e., x moves up 2 levels).**

32

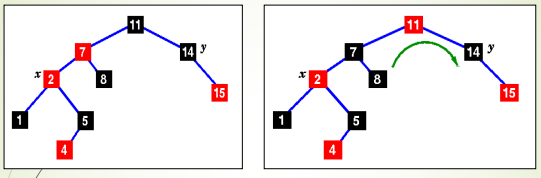
Example of RBT insertion



- Action : case 2
- Make x = p[x]
- Left rotate around new x
- Execute case 3.

33

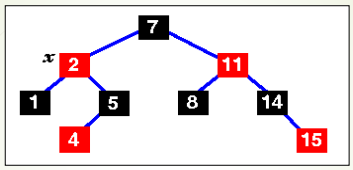
Example of RBT insertion



- Make $p[x]$ = black and $p[p[x]]$ = red.
- Right rotate on $p[p[x]]$
- No longer have 2 reds in a row.
- $p[x]$ is now black ⇒ no more iterations.

34

Example of RBT insertion

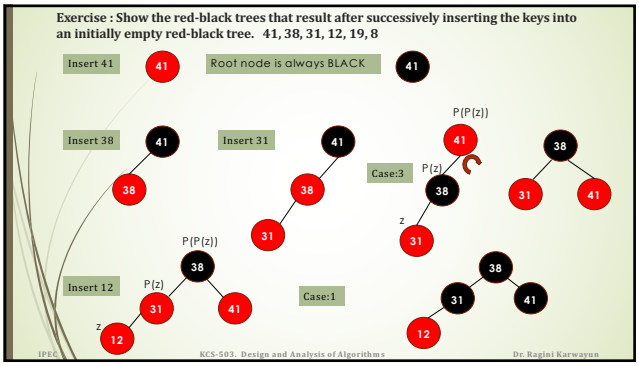


- Make $p[x]$ = black and $p[p[x]]$ = red.
- Right rotate on $p[p[x]]$
- No longer have 2 reds in a row.
- $p[x]$ is now black ⇒ no more iterations.

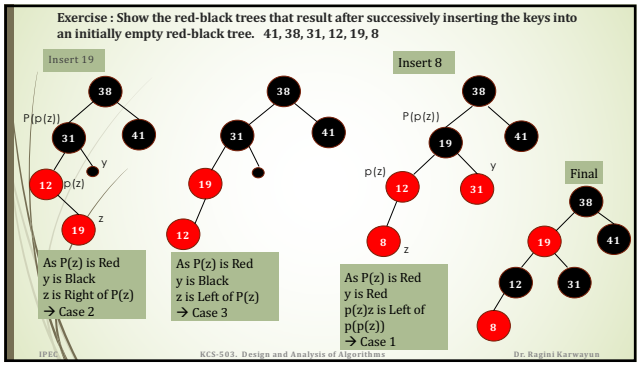
35

Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 41, 38, 31, 12, 19, 8

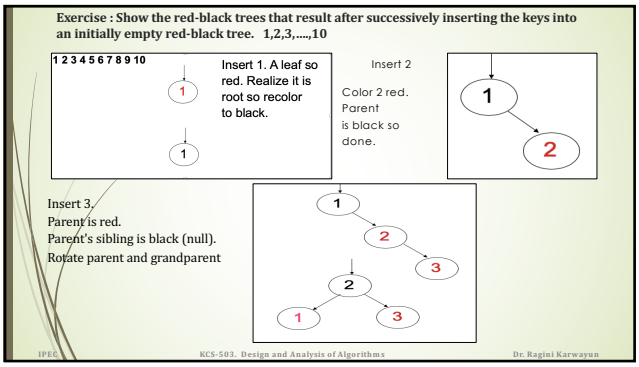
36


Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 41, 38, 31, 12, 19, 8
Insert 41
Root node is always BLACK
Insert 38
Insert 31
Case:3
Insert 12
Case:1
P(P(z))
P(z)
z
IPEC
KCS-503, Design and Analysis of Algorithms
Dr. Ragini Karwayun

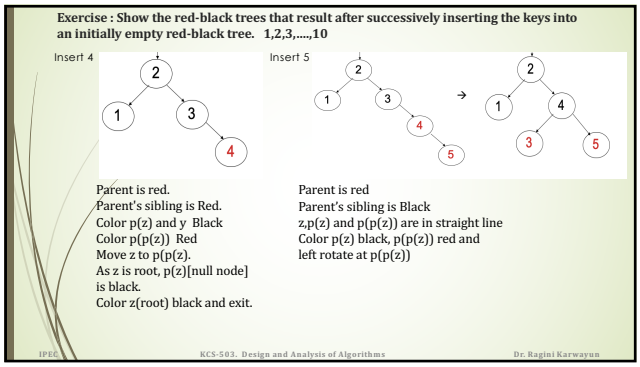
37


Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 41, 38, 31, 12, 19, 8
Insert 19
Insert 8
Final
P(p(z))
p(z)
y
z
As P(z) is Red
y is Black
z is Right of P(z)
→ Case 2
As P(z) is Red
y is Black
z is Left of P(z)
→ Case 3
As P(z) is Red
y is Red
p(z)z is Left of
p(p(z))
→ Case 1
IPEC
KCS-503, Design and Analysis of Algorithms
Dr. Ragini Karwayun

38


Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 1,2,3,….,10
1 2 3 4 5 6 7 8 9 10
Insert 1. A leaf so red. Realize it is root so recolor to black.
Insert 2
Color 2 red. Parent is black so done.
Insert 3.
Parent is red.
Parent's sibling is black (null).
Rotate parent and grandparent
IPEC
KCS-503, Design and Analysis of Algorithms
Dr. Ragini Karwayun

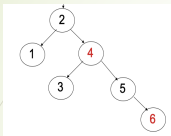
39


Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 1,2,3,….,10
Insert 4
Insert 5
Parent is red.
Parent's sibling is Red.
Color p(z) and y Black
Color p(p(z)) Red
Move z to p(p(z).
As z is root, p(z)[null node] is black.
Color z(root) black and exit.
Parent is red
Parent's sibling is Black
z,p(z) and p(p(z)) are in straight line
Color p(z) black, p(p(z)) red and left rotate at p(p(z))
IPEC
KCS-503, Design and Analysis of Algorithms
Dr. Ragini Karwayun

40

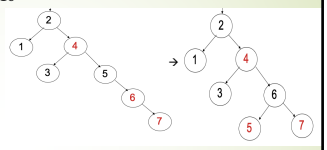
Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 1,2,3,….,10

Insert 6



Parent is red.
Parent's sibling is Red.
Color p(z) and y Black
Color p(p(z)) Red
Move z to p(p(z).
As p(z)[2]is black exit.

Insert 7

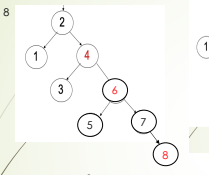


Parent is red
Parent's sibling is Black
z,p(z) and p(p(z)) are in straight line
Color p(z) black, p(p(z)) red and
left rotate at p(p(z))

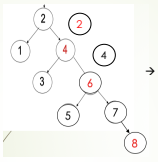
41

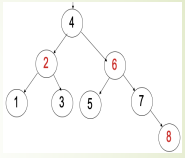
Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 1,2,3,….,10

Insert 8



Parent is red.
Parent's sibling is Red.
Color p(z) and y Black
Color p(p(z)) Red
Move z to p(p(z)).
As z[6] & p(z)[4]are Red
y[1] is black,



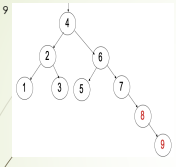


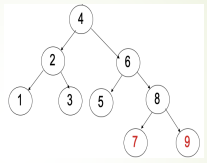
Parent is red
Parent's sibling is Black
z,p(z) and p(p(z)) are in straight line
Color p(z) black, p(p(z)) red and
left rotate at p(p(z))

42

Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 1,2,3,….,10

Insert 9



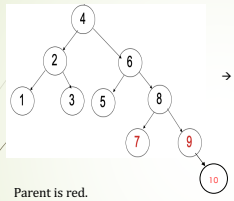


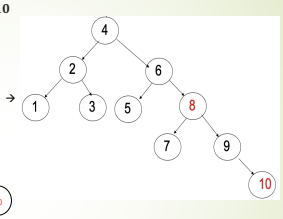
Parent is red
Parent's sibling is Black
z,p(z) and p(p(z)) are in straight line
Color p(z) black, p(p(z)) red and
left rotate at p(p(z))

43

Exercise : Show the red-black trees that result after successively inserting the keys into an initially empty red-black tree. 1,2,3,….,10

Insert 10





Parent is red.
Parent's sibling is Red.
Color p(z) and y Black
Color p(p(z)) Red
Move z to p(p(z)).
As p(z)[6]is black, exit

44

RB TREE DELETION

RB-Delete(*T*, *z*)

1. if *left*[*z*] = *nil*[*T*] or *right*[*z*] = *nil*[*T*]
   y ← *z*
2. else *y* ← TREE-SUCCESSOR(*z*)
3. if *left*[*y*] ≠ *nil*[*T*]
4. *x* ← *left*[*y*]
5. else *x* ← *right*[*y*]
6. *p*[*x*] ← *p*[*y*] // Do this, even if *x* is *nil*[*T*]

The node passed to the fixup routine is the lone child of the spliced-up node, or the sentinel.

RB-Delete (*T*, *z*) (Contd.)

7. if *p*[*y*] = *nil*[*T*]
8. *root*[*T*] ← *x*
9. else if *y* = *left*[*p*[*y*]]
10. *left*[*p*[*y*]] ← *x*
11. else *right*[*p*[*y*]] ← *x*
12. if *y* ≠ *z*
13. *key*[*z*] ← *key*[*y*]
14. copy *y*'s satellite data into *z*
15. if *color*[*y*] = BLACK
16. RB-Delete-Fixup(*T*, *x*)
17. return *y*

45

RB Properties Violation

- **If *y* is black, we could have violations of red-black properties:**
  - Prop. 1. OK.
  - Prop. 2. If *y* is the root and *x* is red, then the root has become red.
  - Prop. 3. OK.
  - Prop. 4. Violation if *p*[*y*] and *x* are both red.
  - Prop. 5. Any path containing *y* now has 1 fewer black node.
- **Remove the violations by calling RB-Delete-Fixup.**

46

RB Properties Violation

- Prop. 5. Any path containing *y* now has 1 fewer black node.
  - Correct by giving *x* an "extra black."
  - Add 1 to count of black nodes on paths containing *x*.
  - Now property 5 is OK, but property 1 is not.
  - *x* is either *doubly black* (if *color*[*x*] = BLACK) or *red & black* (if *color*[*x*] = RED).
  - The attribute *color*[*x*] is still either RED or BLACK. No new values for *color* attribute.
  - In other words, the extra blackness on a node is by virtue of *x* pointing to the node.
- Remove the violations by calling RB-Delete-Fixup.

47

RB Deletion – Fixup

RB-Delete-Fixup(*T*, *x*)

1. while *x* ≠ *root*[*T*] and *color*[*x*] = BLACK
2. do if *x* = *left*[*p*[*x*]]
3. then *w* ← *right*[*p*[*x*]]
4. if *color*[*w*] = RED
5. then *color*[*w*] ← BLACK // Case 1
6. *color*[*p*[*x*]] ← RED // Case 1
7. LEFT-ROTATE(*T*, *p*[*x*]) // Case 1
8. *w* ← *right*[*p*[*x*]] // Case 1

48

RB Deletion – Fixup

```
RB-Delete-Fixup(T, x) (Contd.)
        /* x is still left[p[x]] */
9.        if color[left[w]] = BLACK and color[right[w]] = BLACK
10.         then color[w] ← RED                          // Case 2
11.              x ← p[x]                                // Case 2
12.         else if color[right[w]] = BLACK
13.                then color[left[w]] ← BLACK           // Case 3
14.                     color[w] ← RED                   // Case 3
15.                     RIGHT-ROTATE(T,w)                // Case 3
16.                w ← right[p[x]]                       // Case 3
17.              color[w] ← color[p[x]]                  // Case 4
18.              color[p[x]] ← BLACK                     // Case 4
19.              color[right[w]] ← BLACK                 // Case 4
20.              LEFT-ROTATE(T, p[x])                    // Case 4
21.              x ← root[T ]                            // Case 4
22.       else (same as then clause with "right" and "left" exchanged)
23.  color[x] ← BLACK
```


49

RB TREE DELETION OPERATION

- To delete a leaf node, just delete it.
- If the node to be deleted has only one child, splice that node out by connecting its parent and child.
- If the node to be deleted has two children, then find the successor, and replace the key in the node to be deleted by the key from the successor of the node to be deleted.

Note : the node to be deleted is considered as y.

50

RB TREE DELETION FIXUP

- Node y = a node which is physically deleted.
- Node x = is a child of y.
- We call this fixup algorithm when color[y]=black.
- If color[x]= red **or** x is a root then
  - Color[x]=black
  - Exit
- If x=left[p[x]] then
  - W=right[p[x]] and execute cases 1,2,3,4.
- Else
  - W=left[p[x]] and execute cases 5,6,7,8.

51

If x=left[p[x]] then W=right[p[x]] and execute cases 1,2,3,4.

- Case 1 – color[*w*] is red
  - Color[*w*]= black
  - Color[*p*[*x*]]= red .
  - left rotate on *p*[*x*].
  - W=right[p[x]].
  - Go immediately to case 2, 3, or 4.
- Case 2 – *w* is black, both *w*'s children are black
  - Color[w]=red
  - X=p[x].
- Case 3 – w is black, w's left child is red, w's right child is black
  - Color[left[w]]=black.
  - Color[w]=red
  - right rotate on w.
  - W=right[p[x]].
- Case 4 – w is black, w's right child is red
  - Color[w]=color[p[x]].
  - Color[p[x]]=black.
  - Color[right[w]]=black.
  - left rotate on p[x].
  - x=root[t].

52

If x= right[p[x]] then w= left[p[x]] and execute cases 5,6,7,8

- Case 5 – color[*w*] is red
  - Color[*w*]= black
  - Color[*p*[*x*]]= red .
  - right rotate on *p*[*x*].
  - W=left[p[x]].
  - Go immediately to case 6,7, or 8.
- Case 6 – *w* is black, both *w*'s children are black
  - Color[w]=red
  - X=p[x] .
- Case 7 – w is black, w's right child is red, w's left child is black
  - Color[right[w]]=black.
  - Color[w]=red
  - left rotate on w.
  - W=left[p[x]].
- Case 8 – w is black, w's left child is red
  - Color[w]=color[p[x]].
  - Color[p[x]]=black.
  - Color[left[w]]=black.
  - right rotate on p[x].
  - x=root[t].

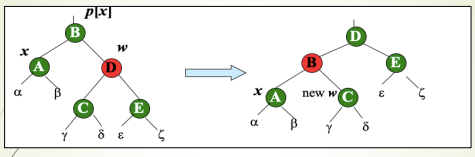
53

RB Tree Deletion – Fixup

- *Idea:* Move the extra black up the tree until *x* points to a red & black node ⇒ turn it into a black node,
- *x* points to the root ⇒ just remove the extra black, or
- We can do certain rotations and re-colorings and finish.
- Within the while loop:
  - *x* always points to a non root doubly black node.
  - *w* is *x*'s sibling.
  - *w* cannot be *nil*[*T*], since that would violate property 5 at *p*[*x*].
- 8 cases in all, 4 of which are symmetric to the other.

54

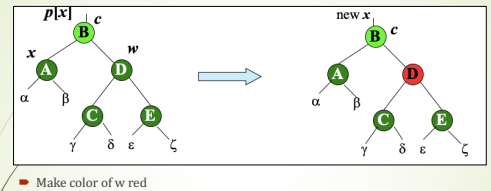
CASE 1 – *w* IS RED



- *w* must have black children.
- Make *w* black and *p*[*x*] red (because *w* is red *p*[*x*] couldn't have been red).
- Then left rotate on *p*[*x*].
- Right child of p(x) becomes w.
- Go immediately to case 2, 3, or 4.

55

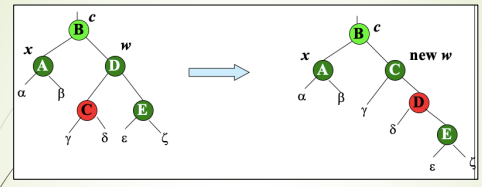
CASE 2 – *w* IS BLACK, BOTH *w*'S CHILDREN ARE BLACK



- Make color of w red
- Move x to parent of x

56

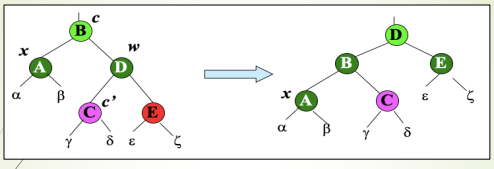
Case 3 – *w* is black, *w*'s left child is red, *w*'s right child is black



- Make *w* red.
- Make *w*'s left child black.
- Then right rotate on *w*.
- Make the right child of parent of x as new w
- New sibling *w* of *x* is black with a red right child ⇒ case 4.

57

Case 4 – *w* is black, *w*'s right child is red



- Color *w* as the color of *p*[x]
- Make *p*[x] black and *w*'s right child black.
- Then left rotate on *p*[x].
- All done. Setting x to root causes the loop to terminate.

58

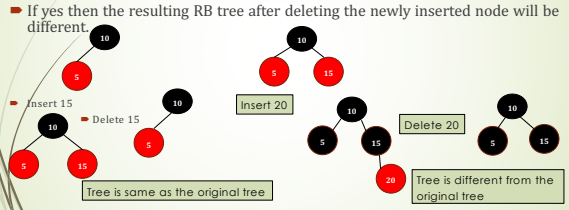
Analysis

- $O(\lg n)$ time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - Case 2 is the only case in which more iterations occur.
    - *x* moves up 1 level.
    - Hence, $O(\lg n)$ iterations.
  - Each of cases 1, 3, and 4 has 1 rotation ⇒ ≤ 3 rotations in all.
  - Hence, $O(\lg n)$ time.

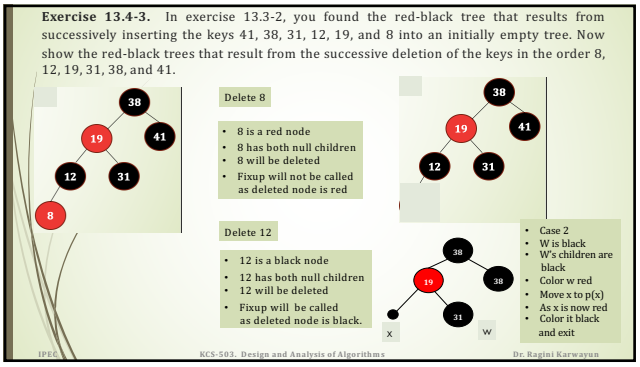
59

Exercise 13.4-7 : Suppose that a node x is inserted into a red-black tree with RB-INSERT and then immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

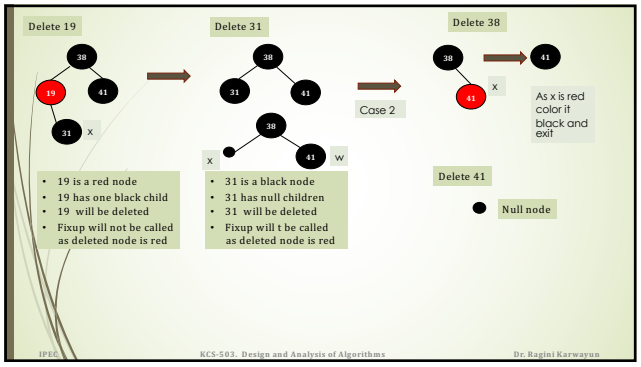
- It depends on the fact whether any operation was performed in fix-up algorithm after inserting the node in RB Tree.
- If yes then the resulting RB tree after deleting the newly inserted node will be different.



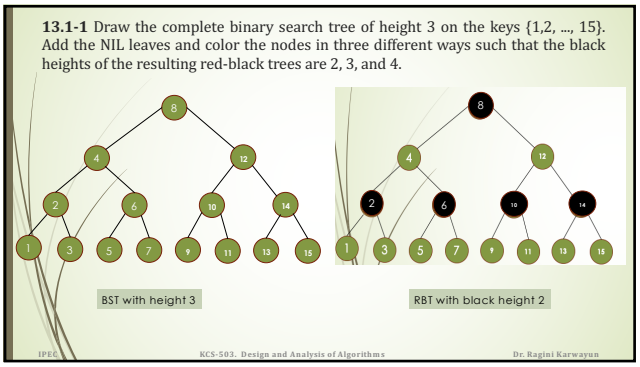
60


Exercise 13.4-3. In exercise 13.3-2, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, and 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, and 41.
Delete 8
8 is a red node
8 has both null children
8 will be deleted
Fixup will not be called as deleted node is red
Delete 12
12 is a black node
12 has both null children
12 will be deleted
Fixup will be called as deleted node is black.
Case 2
W is black
W's children are black
Color w red
Move x to p(x)
As x is now red
Color it black and exit
x
w
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

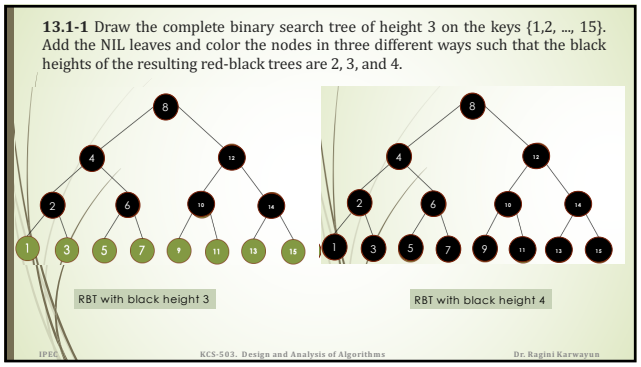
61


Delete 19
19 is a red node
19 has one black child
19 will be deleted
Fixup will not be called as deleted node is red
Delete 31
31 is a black node
31 has null children
31 will be deleted
Fixup will t be called as deleted node is red
Case 2
Delete 38
As x is red color it black and exit
Delete 41
Null node
x
w
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

62


13.1-1 Draw the complete binary search tree of height 3 on the keys {1,2, ..., 15}. Add the NIL leaves and color the nodes in three different ways such that the black heights of the resulting red-black trees are 2, 3, and 4.
BST with height 3
RBT with black height 2
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

63


13.1-1 Draw the complete binary search tree of height 3 on the keys {1,2, ..., 15}. Add the NIL leaves and color the nodes in three different ways such that the black heights of the resulting red-black trees are 2, 3, and 4.
RBT with black height 3
RBT with black height 4
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

64

Exercise 13.1-3 : Suppose that root of a Red-Black tree is red. If we make it black, does the tree remain a Red-Black tree?

If we color the root of a relaxed red-black tree black but make no other changes, the resulting tree is a red-black tree. Not even any black-heights change.

65