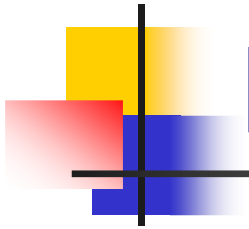
Ch 16. Balanced Search Trees

© copyright 2006 SNU IDB Lab.

SNU
IDB Lab.

Bird's-Eye View (0)

- Chapter 15: Binary Search Tree
  - BST and Indexed BST
- Chapter 16: Balanced Search Tree
  - AVL tree: BST + Balance
  - B-tree: generalized AVL tree
- Chapter 17: Graph



Bird's-Eye View

- Balanced tree structures
  - Height is O(log n)
- AVL
  - Binary Search Tree with Balance
- Red-black trees
- Splay trees
  - Individual dictionary operation ➔ 0(n)
  - Take less time to perform a sequence of u operations ➔ 0(u log u)
- B-trees (Balanced Tree)
  - Suitable for external memory

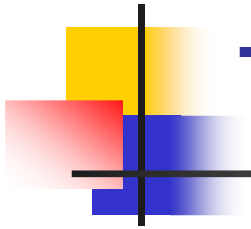
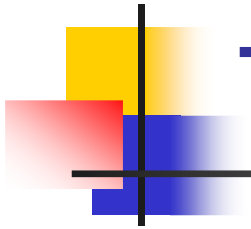


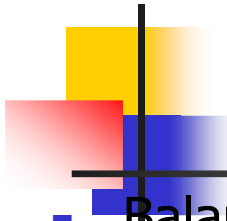
Table of Contents

- *AVL TREES*
  - *Definition*
  - Searching an AVL Search Tree
  - Inserting into an AVL Search Tree
  - Deletion from an AVL Search Tree
- RED-BLACK TREES
- SPLAY TREES
- B-TREES



The History of Balanced Trees

- Adel'son-Vel'skiĭ and Landis introduced AVL tree in 1962
  - Ensures balance by restricting every node's depth to differ at most by 1
- Bayer and McCreight introduced B-tree in 1972
  - Kept balanced by requiring that all leaf nodes are at the same depth
  - Join or split is needed instead of re-balancing
- Bayer, Guibas and Sedgewick introduced Red-black tree in 1978
  - Ensures balance by restricting the occurrence of red nodes in the tree
- Sleator and Tarjan introduced Splay tree in 1983
  - Maintains balance without any explicit balance condition such as color
  - Splay operations are performed within the tree every time an access is made



AVL TREES

- Balanced tree
  - Trees with a worst-case height of O(log n)
- AVL search tree
  - Balanced binary search trees
  - Can be generalized to a B-tree
- A height-balanced *k* tree (HB(*k*) tree)
  - Allowable height difference of any two sub-trees is *k*
- AVL Tree : HB(1) Tree
  - G.M. **A**del'son, **V**el'skii, E.M. **L**andis
  - Performance
    - Given *N* keys, worst-case search ➔ $1.44 \log_2(N+2)$

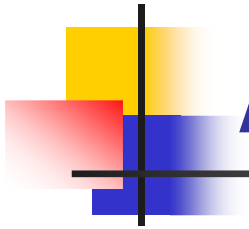
    cf. Completely balanced AVL tree : worst-case search ➔ $\log_2(N+1)$

Height of an AVL Tree

- n : nodes in AVL tree
- N_h : min number of nodes in an AVL tree of height h
- $N_h = N_{h-1} + N_{h-2} + 1$, $N_0 = 0$, and $N_1 = 1$
  - Similar in definition to Fibonacci numbers

    $F_h = F_{n-1} + F_{n-2.}$, $F_0 = 0$ and $F_1 = 1$

    It can be shown that $N_h = F_{h+2} - 1$ for $h > 0$
  - Fibonacci theory: $F_h \fallingdotseq \text{Ø}^h/\sqrt{5}$ where $\text{Ø} = (1 + \sqrt{5})/2$
    therefore $N_h \fallingdotseq \text{Ø}^{h+2}/\sqrt{5}-1$
  - If there are n nodes then its height $h = \log_{\text{Ø}}(\sqrt{5}(n+1)) - 2 \fallingdotseq 1.44\log_2(n+2)$
    $h = O(\log n)$



AVL Tree Definition

- An empty binary tree is an AVL Tree

- If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff

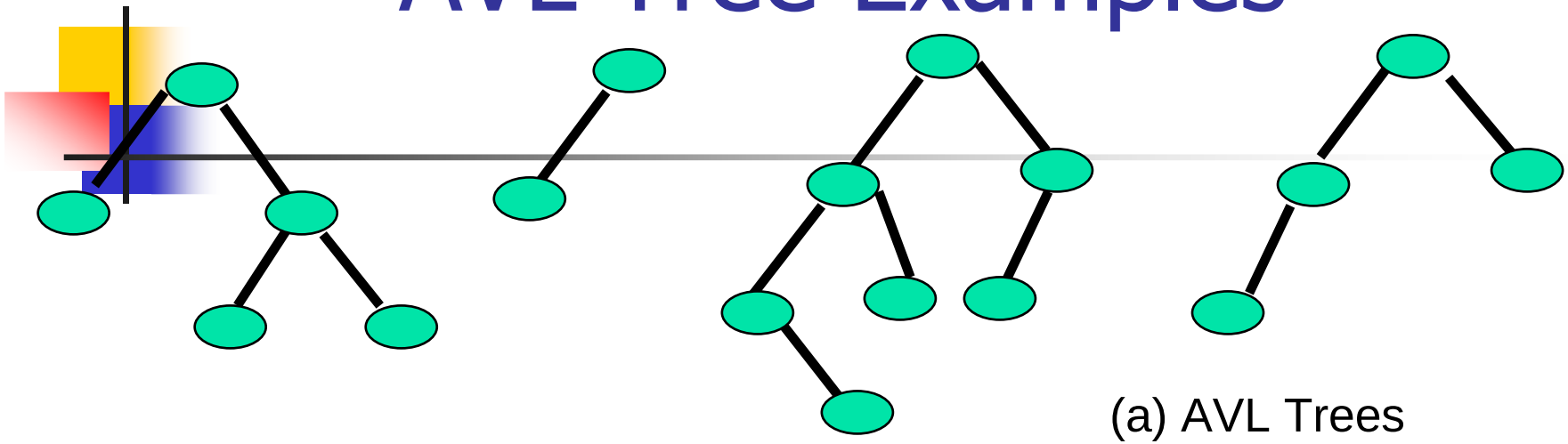
  (1) T_L and T_R are AVL Trees and

  (2) $| h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively

- For any node in tree T in AVL tree, BF(T) should be one of " -1, 0, 1"
- If BF(T) is -2 or 2, then proper rotation is performed in order to get balance

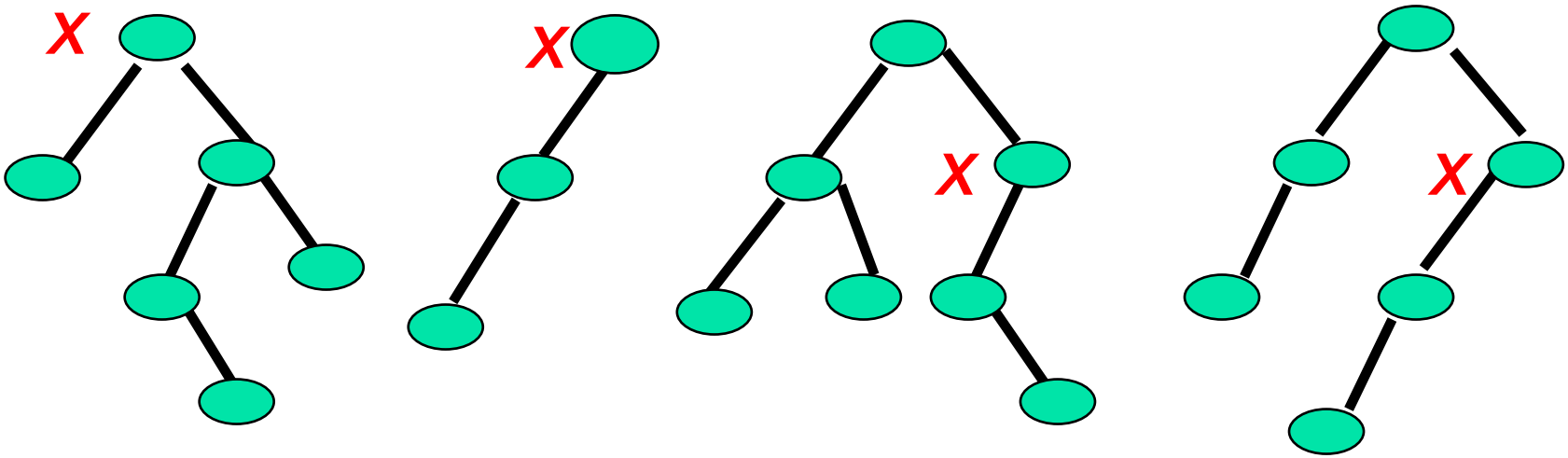
- Conceptually AVL search tree = AVL tree + Binary Search Tree

AVL Tree Examples



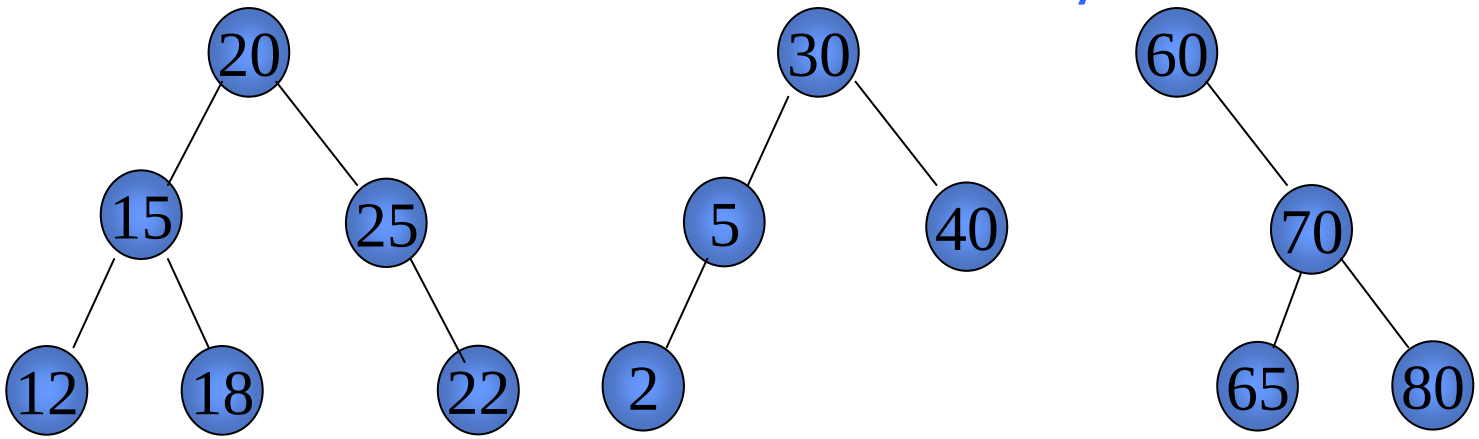
(a) AVL Trees

(b) Non - AVL Trees



Intuition: AVL Search Tree

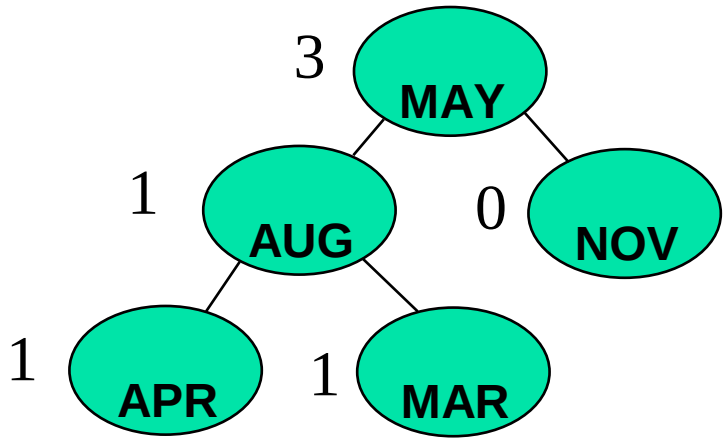
- AVL Search Tree = Binary Search Tree + AVL Tree
  = Balanced Binary Search Tree



	(a)	(b)	(c)
BST	X	O	O
AVL	O	O	X
AVL ST	X	O	X

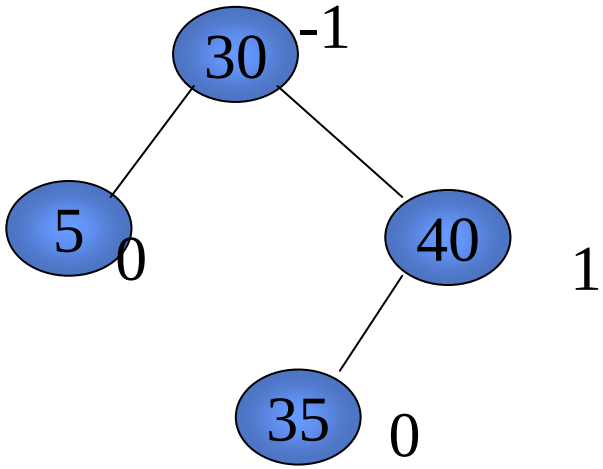
Indexed AVL Search Tree

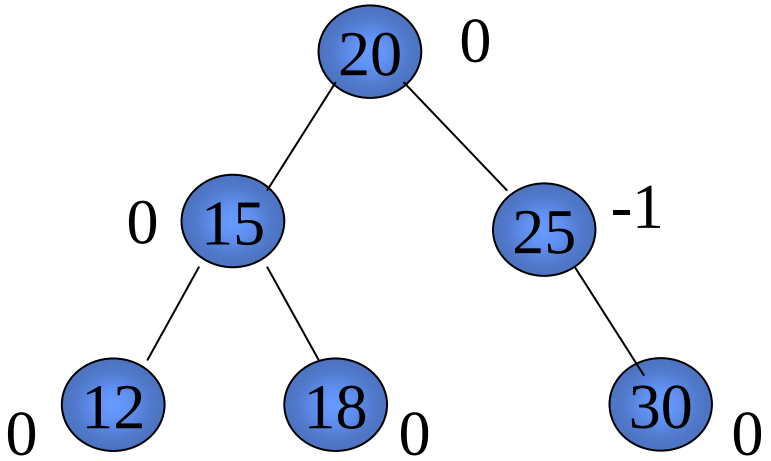
- Indexed AVL search Tree
  = AVL Tree + LeftSize variable
  = (Balanced + Binary Search Tree) + LeftSize variable



Representation of an AVL Tree

- Balance factor bf(x) of a node x = height of left subtree – height of right subtree
- Permissible balance factors: (-1, 0, 1)





AVL Search Tree Example (1)

New Identifier

MARCH

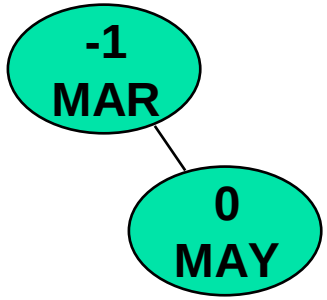
After Insertion

No Rebalancing needed

New Identifier

MAY

After Insertion

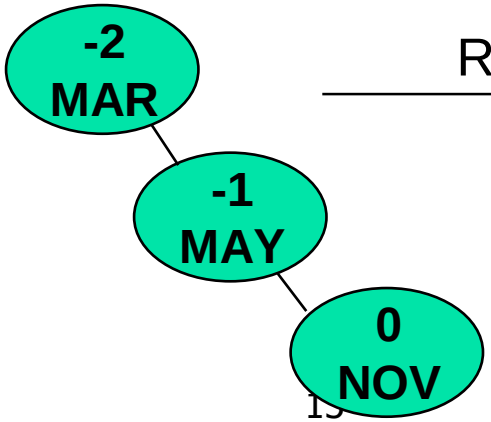


No Rebalancing needed

New Identifier

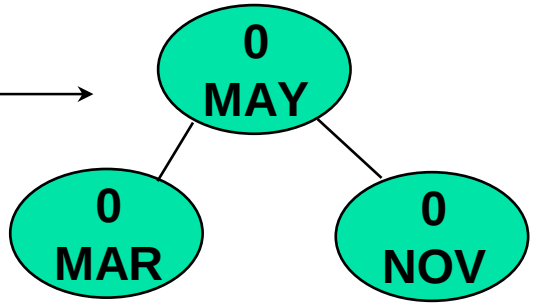
NOVEMBER

After Insertion



RR

After Rebalancing

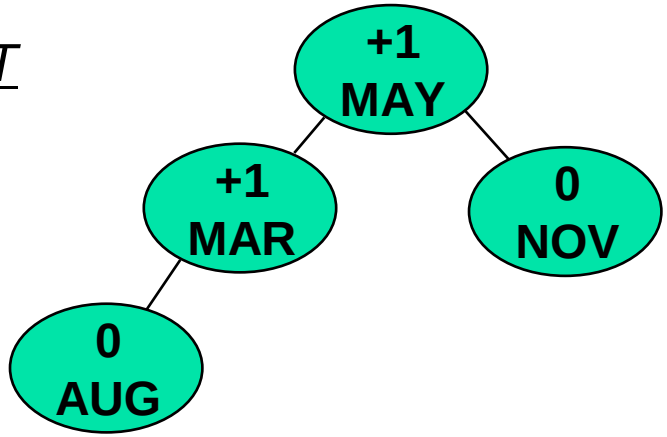


AVL Search Tree Example (2)

New Identifier

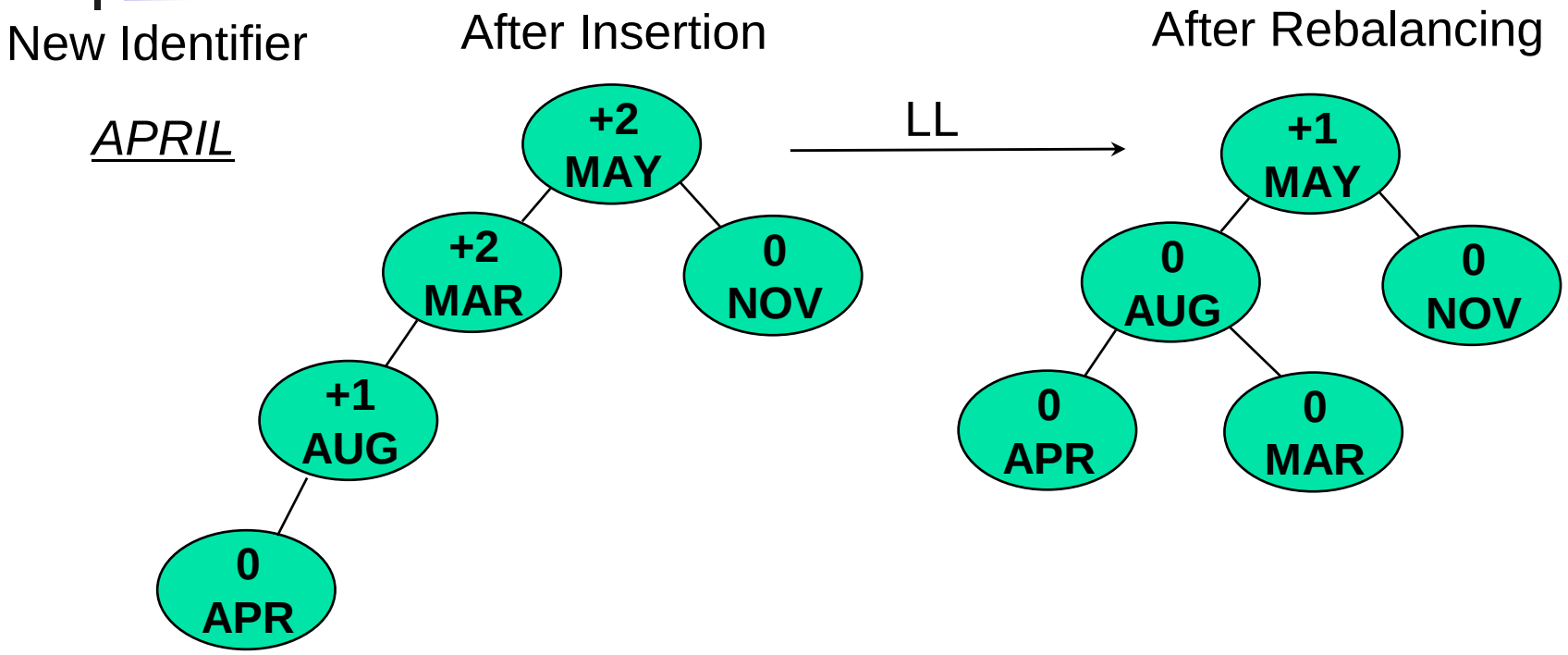
AUGUST

After Insertion

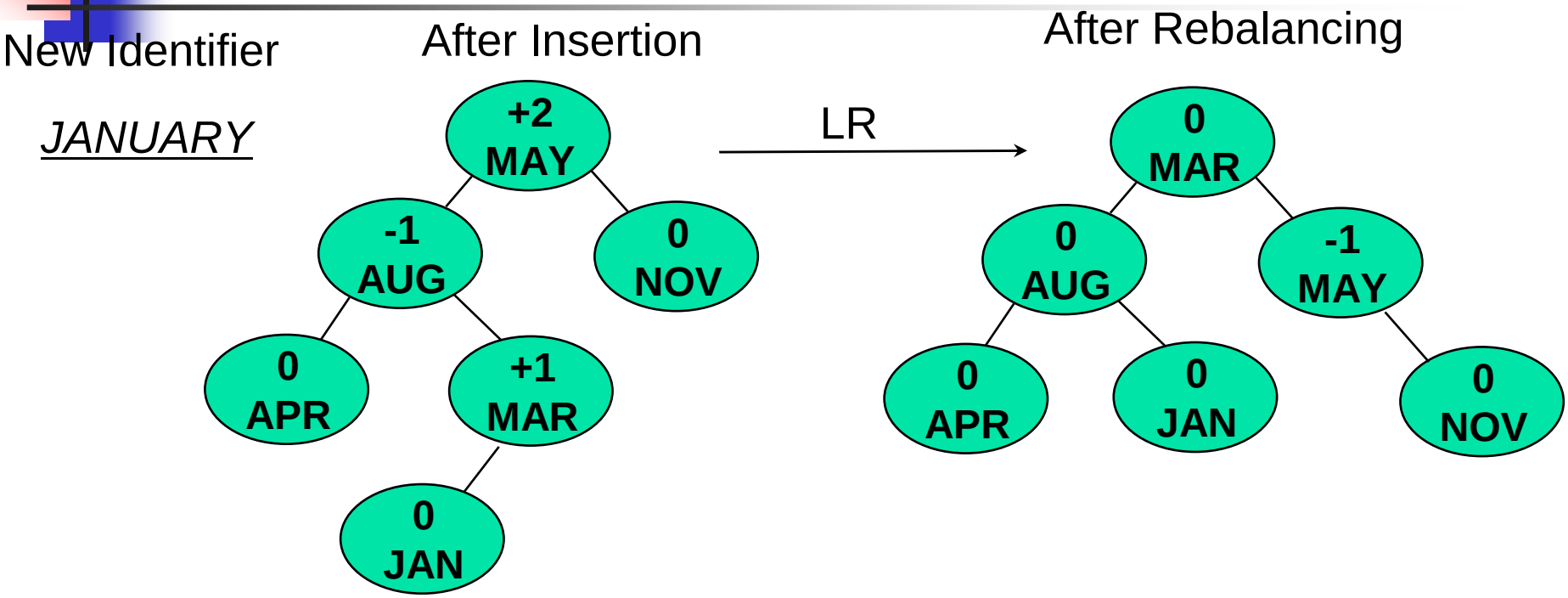


No Rebalancing needed

AVL Search Tree Example (3)



AVL Search Tree Example (4)

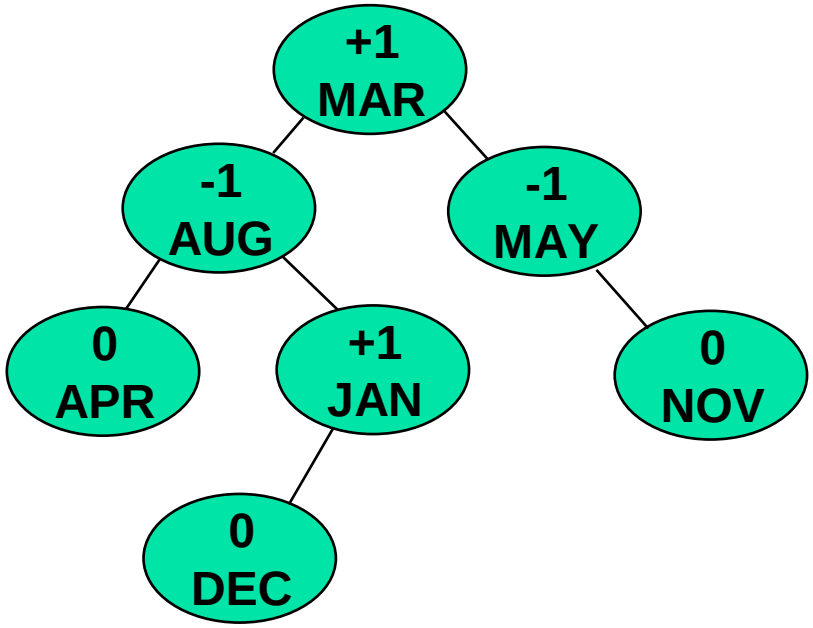


AVL Search Tree Example (5)

New Identifier

DECEMBER

After Insertion



No Rebalancing needed

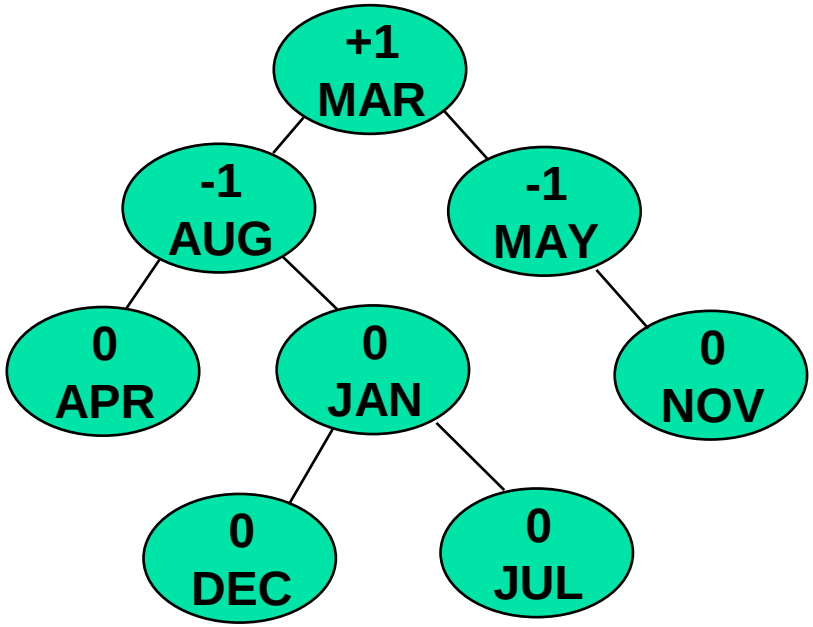
AVL Search Tree Example (6)

New Identifier

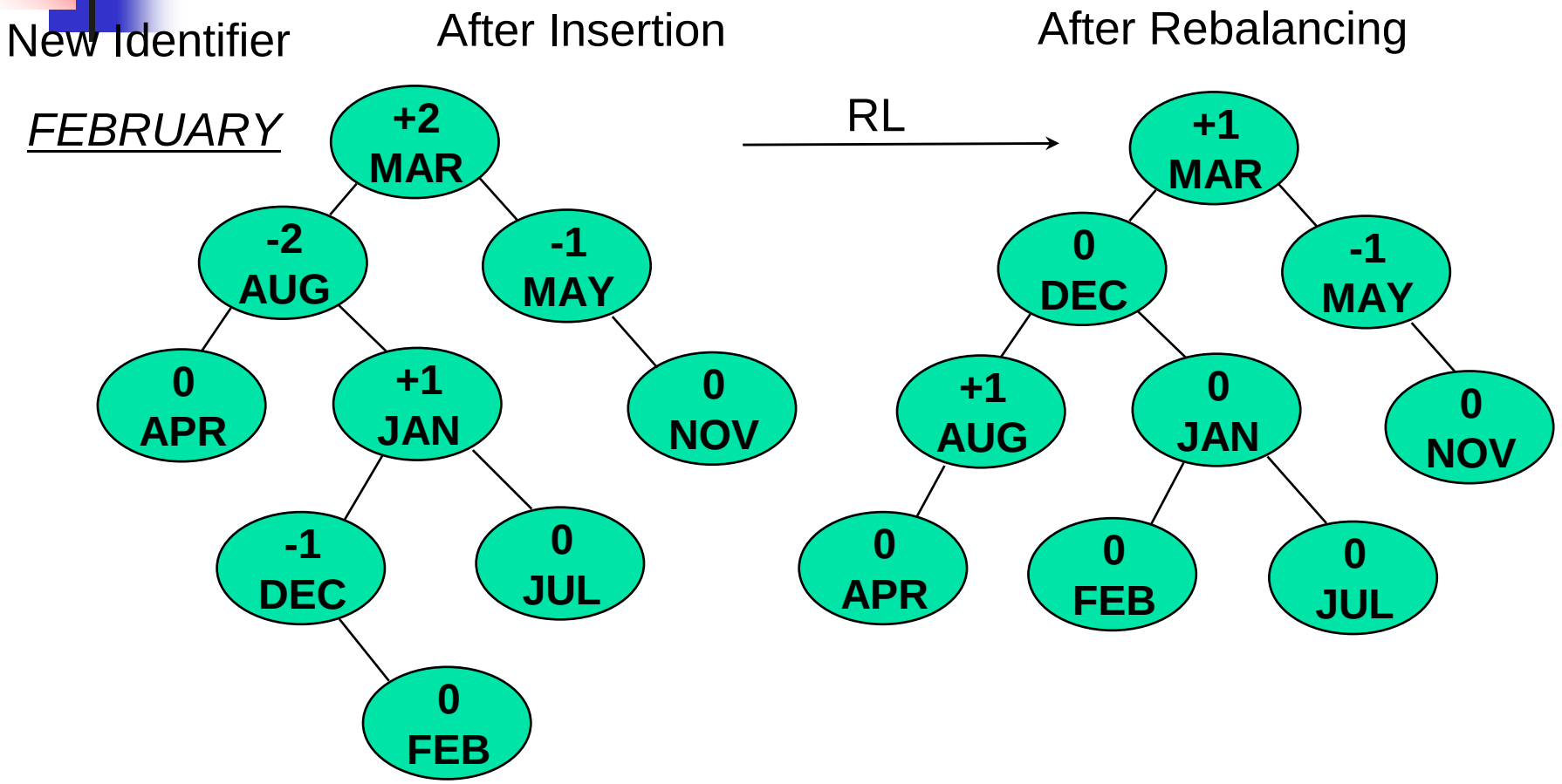
JULY

After Insertion

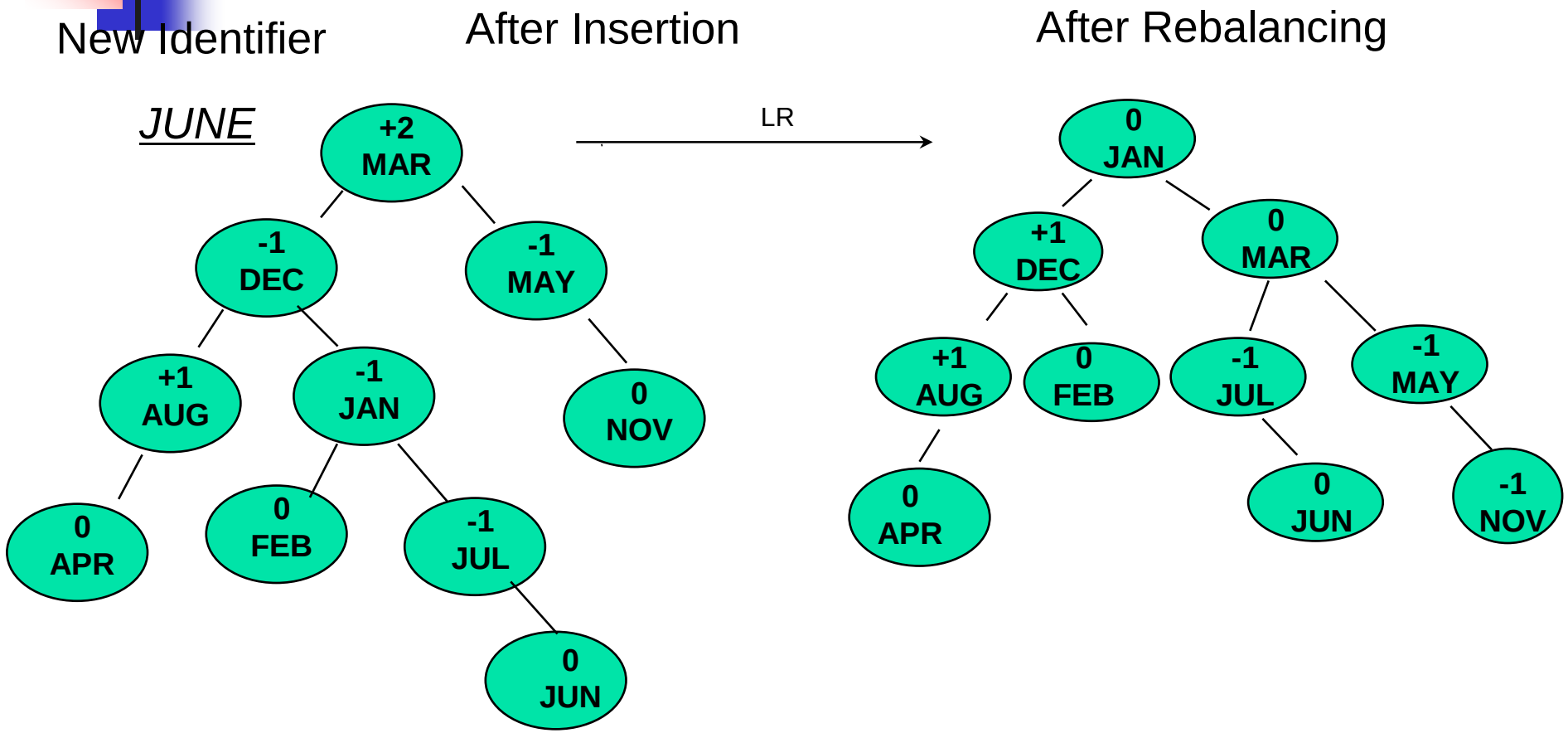
No Rebalancing needed



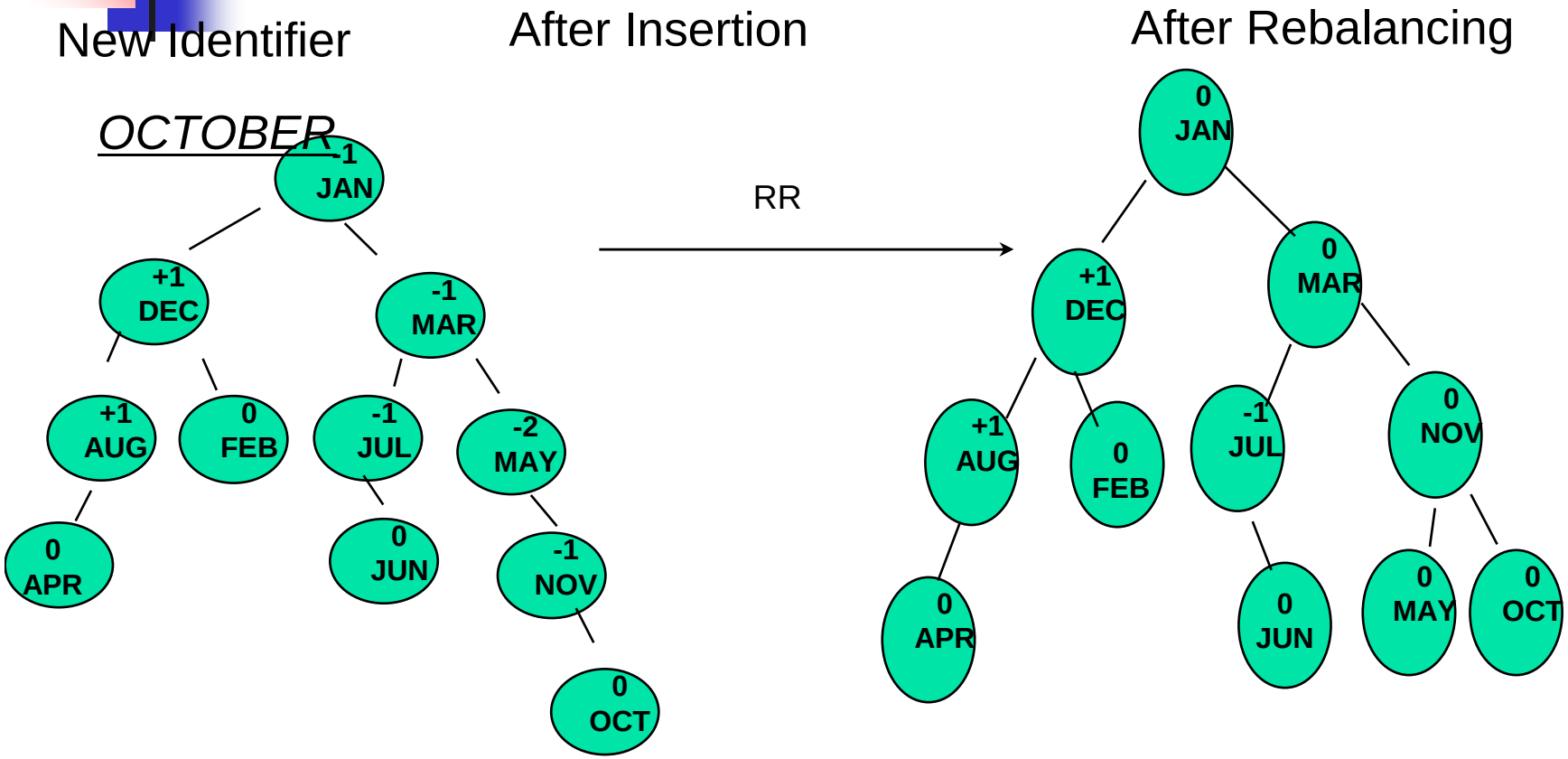
AVL Search Tree Example (7)



AVL Search Tree Example (8)



AVL Search Tree Example (9)



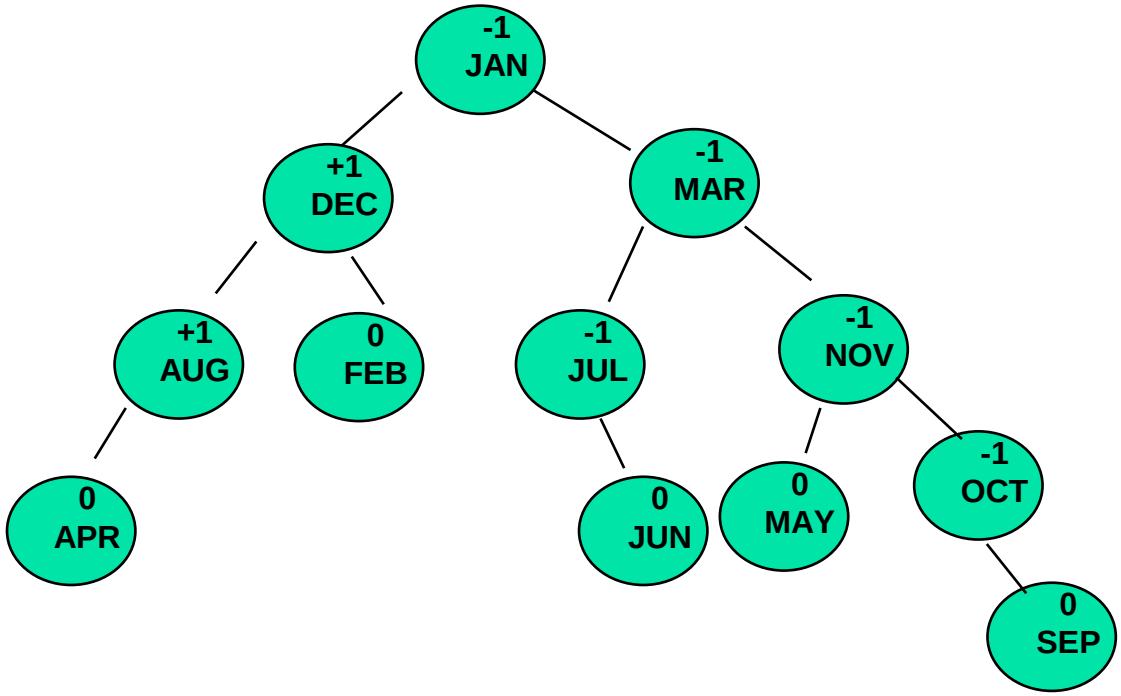
AVL Search Tree Example (11)

New Identifier

SEPTEMBER

After Insertion

No Rebalancing needed



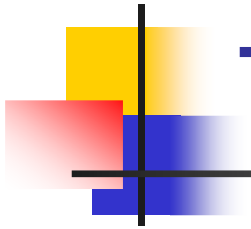


Table of Contents

- AVL TREES
  - Definition
  - *Searching an AVL Search Tree*
  - Inserting into an AVL Search Tree
  - Deletion from an AVL Search Tree
- RED-BLACK TREES
- SPLAY TREES
- B-TREES

Searching in an AVL Search Tree

- search in binary search tree : Wish to Search for thekey from root to leaf

```
If (root == null)  search is unsuccessful;
else
if        (thekey < key in root) only left subtree is to be searched;
else  if  (thekey > key in root) only right subtree is to be searched;
        else  (thekey == key in root)  search terminates successfully;
```

  - Subtrees may be searched similarly in a recursive manner

- TimeComplexity = O(height)

- Height of an AVL tree with n element
  - O(log n): search time is O(log n)

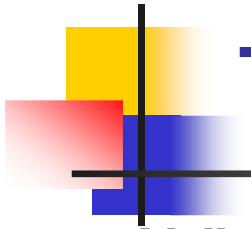
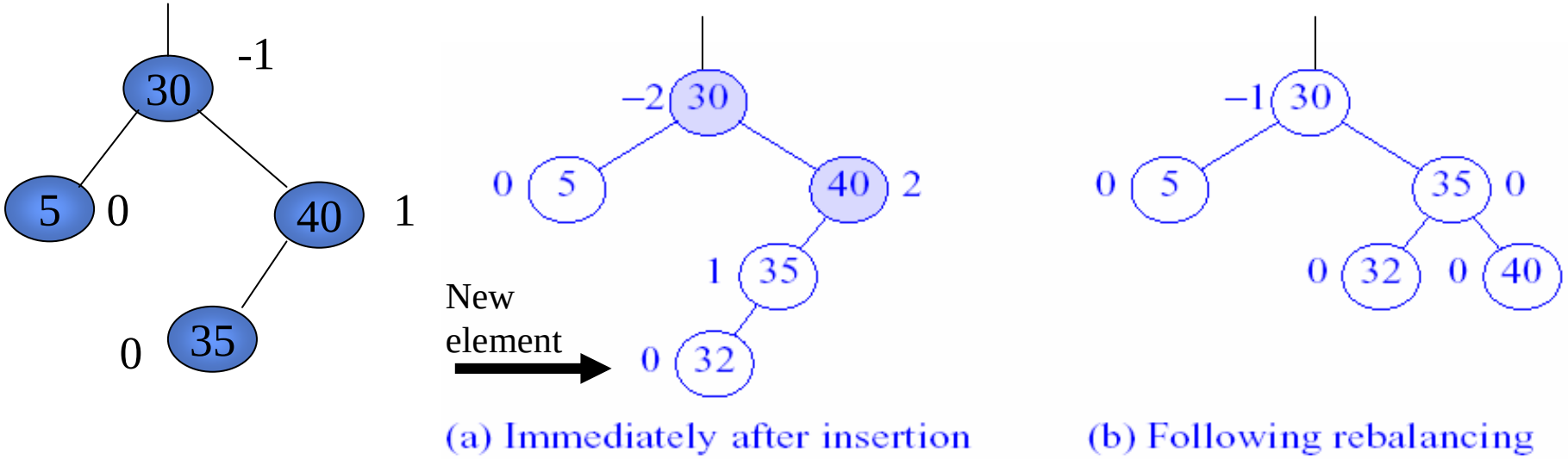


Table of Contents

- AVL TREES
  - Definition
  - Searching an AVL Search Tree
  - Inserting into an AVL Search Tree
  - Deletion from an AVL Search Tree
- RED-BLACK TREES
- SPLAY TREES
- B-TREES

Unbalance due to Inserting

- When an insertion into an AVL Tree using the strategy of Program15.5 (insert in BST), the resulting tree is unbalanced



Observations on Imbalance due to Insertion

O1: In the unbalanced tree the BFs are limited to –2, -1, 0, 1, 2

O2: A node with BF "2" had a BF "1" before the insertion

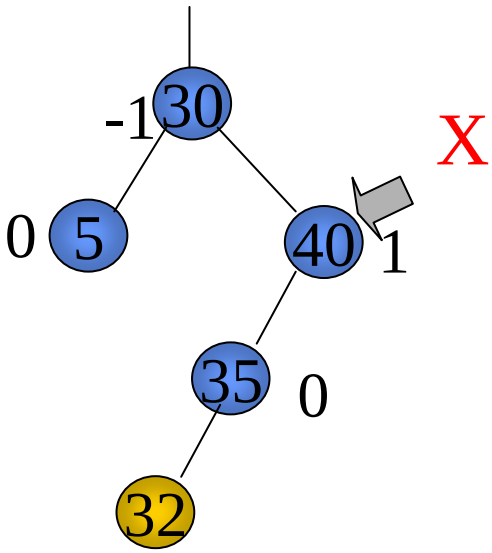
O3: The BF of only those nodes on the path from the root to the newly inserted node can change as a result of the insertion

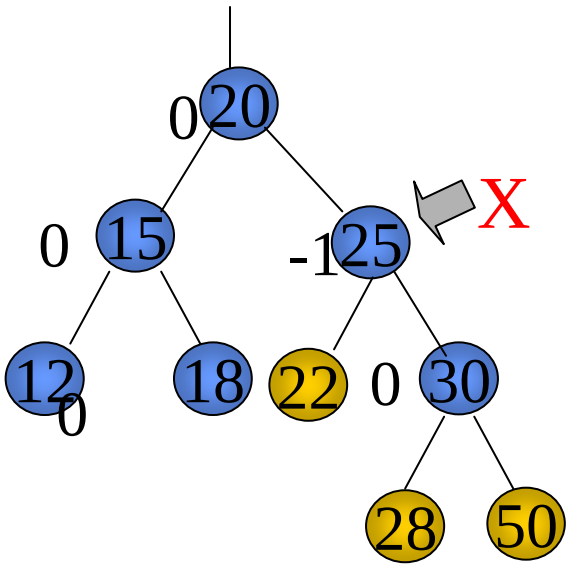
O4: Let **A** denote the nearest ancestor of the newly inserted node whose BF is either –2 or 2. The BF of all nodes on the path from A to the newly inserted node was 0 prior to the insertion

O5: Imbalance can happen in the last node encountered that has a balance factor 1 or –1 prior to the insertion

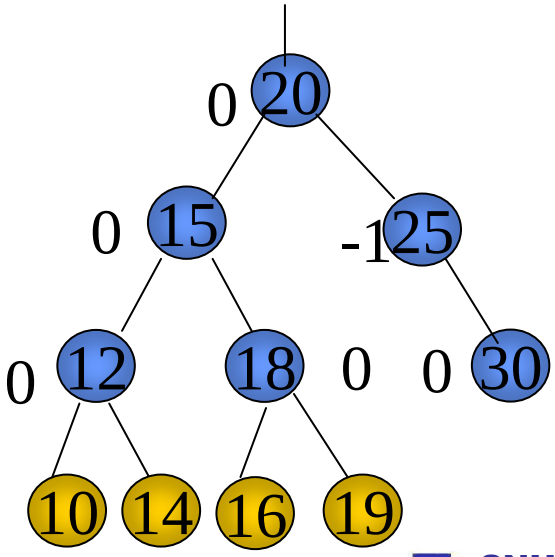
Node X with Potential Imbalance (1)

- Let **X** denote the last node encountered that has a balance factor 1 or –1 prior to the insertion
- If the tree is unbalanced following the insertion, X exists
  - If bf(x) = 0 after the insertion, then the height of the subtree with root X is the same before and after the insertion

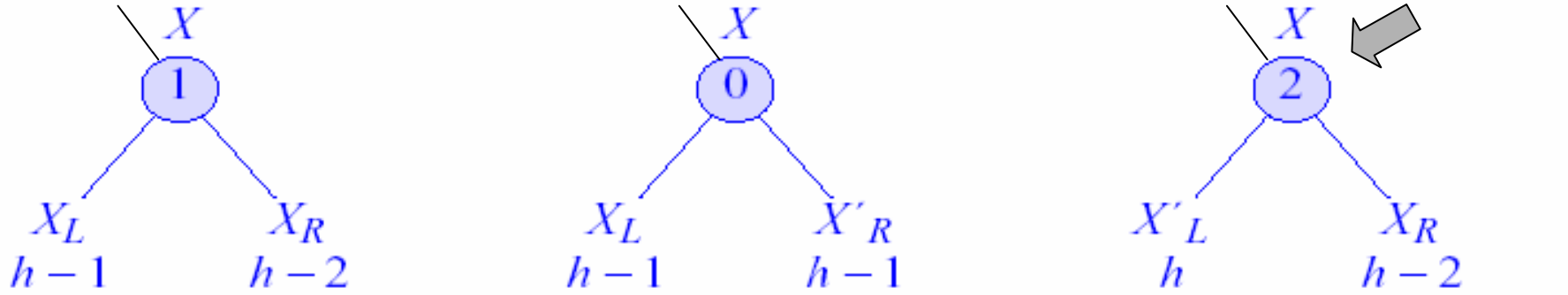




No node X



Node X with Potential Imbalance (2)



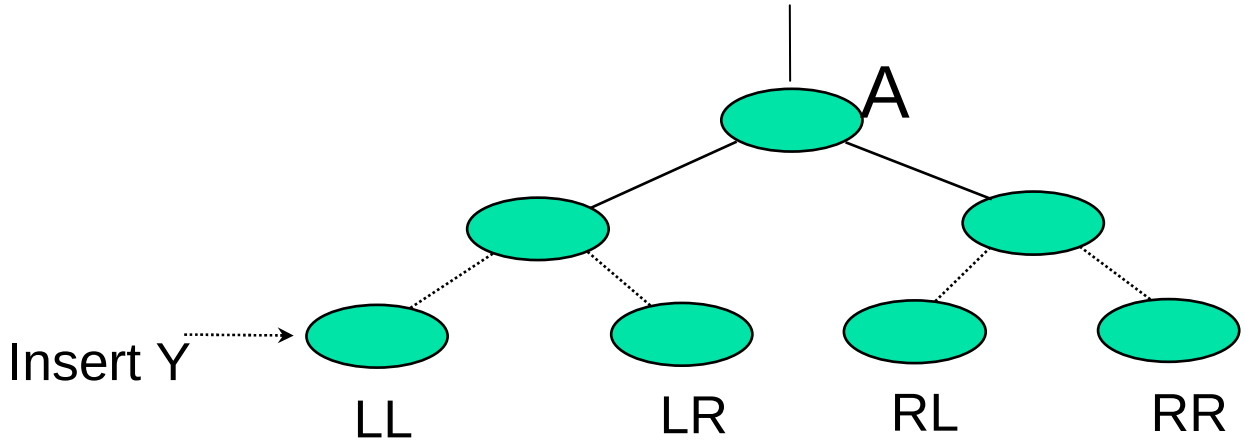
(a) Before insertion (b) After inserting into X_R (c) After inserting into X_L

	(a)	(b)	(c)
height	h	h	h + 1
bf(x)	1	0	2
balanced	balanced	balanced	imbalanced

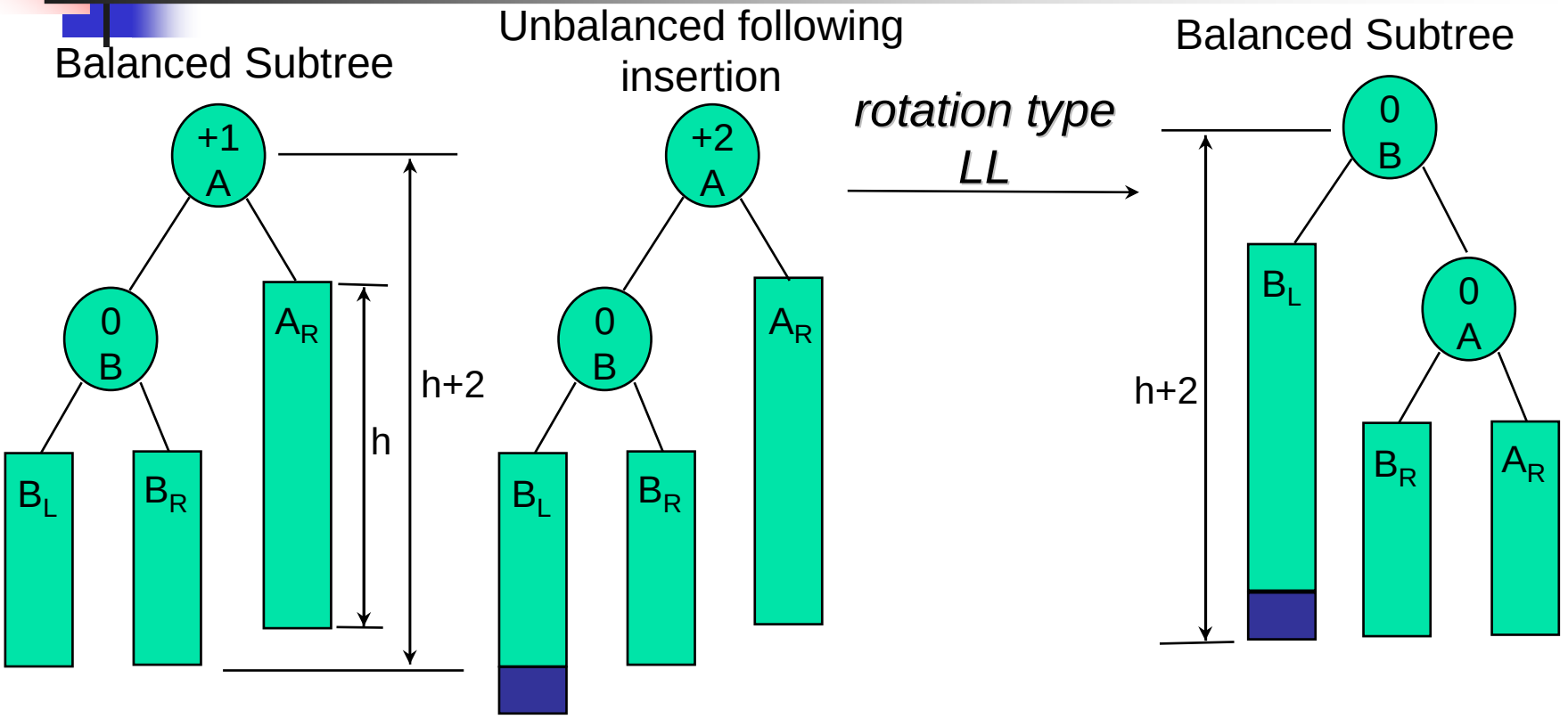
- The only way the tree can become unbalanced is when the insertion causes bf(x) to change from –1 to –2 or from 1 to 2.

Imbalance Patterns due to Insertion

- The imbalance at A is one of the types
  - LL (when new node is in the left subtree of the left subtree of A)
  - LR (when new node is in the right subtree of the left subtree of A)
  - RR (when new node is in the right subtree of the right subtree of A)
  - RL (when new node is in the left subtree of the right subtree of A)
- LL and RR imbalances require single rotation
- LR and RL imbalances require double rotations

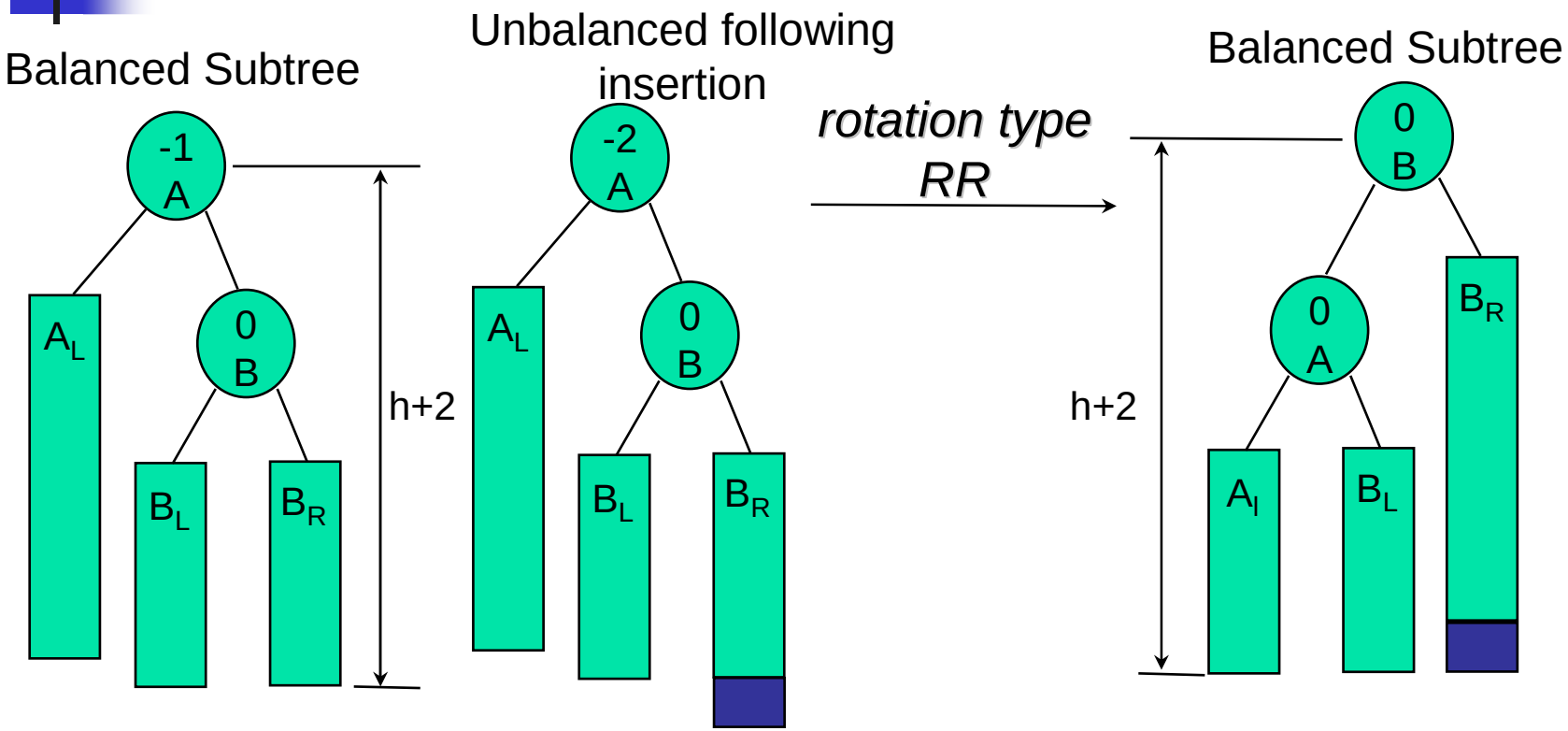


LL Rebalancing after Insertion



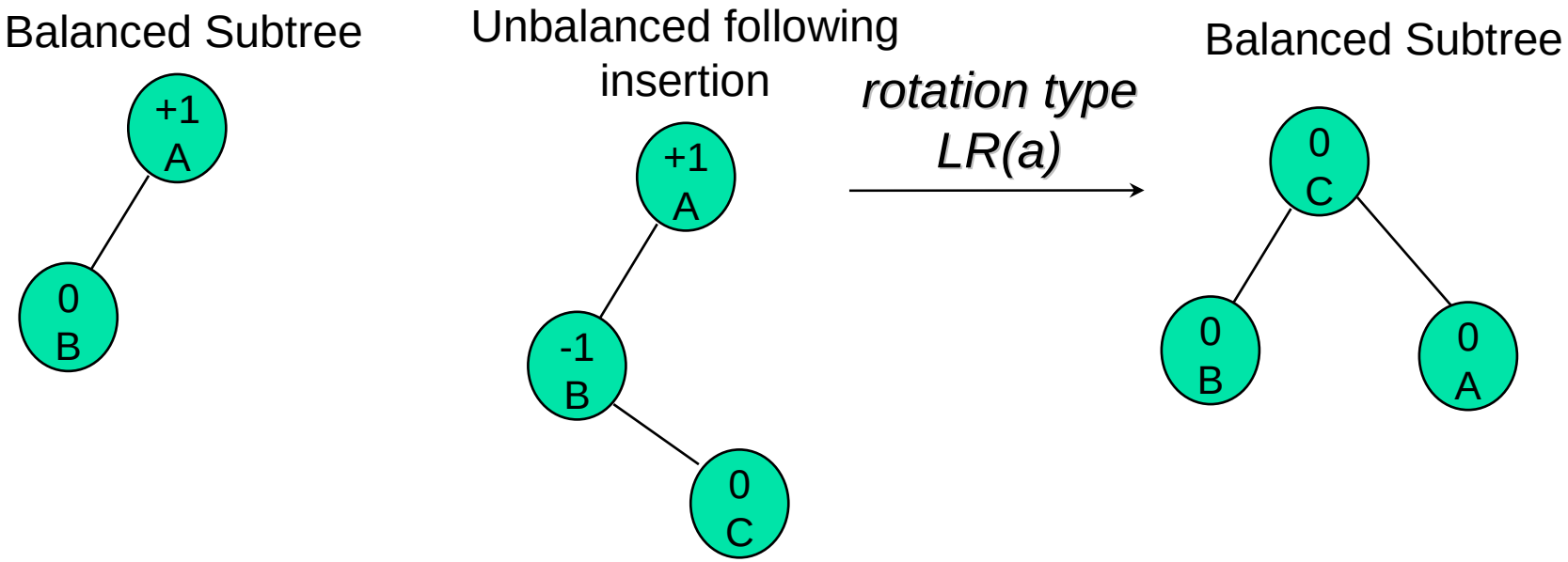
Height of B_L increase to h+1
(BL < B < BR < A < AR)

RR Rebalancing after Insertion



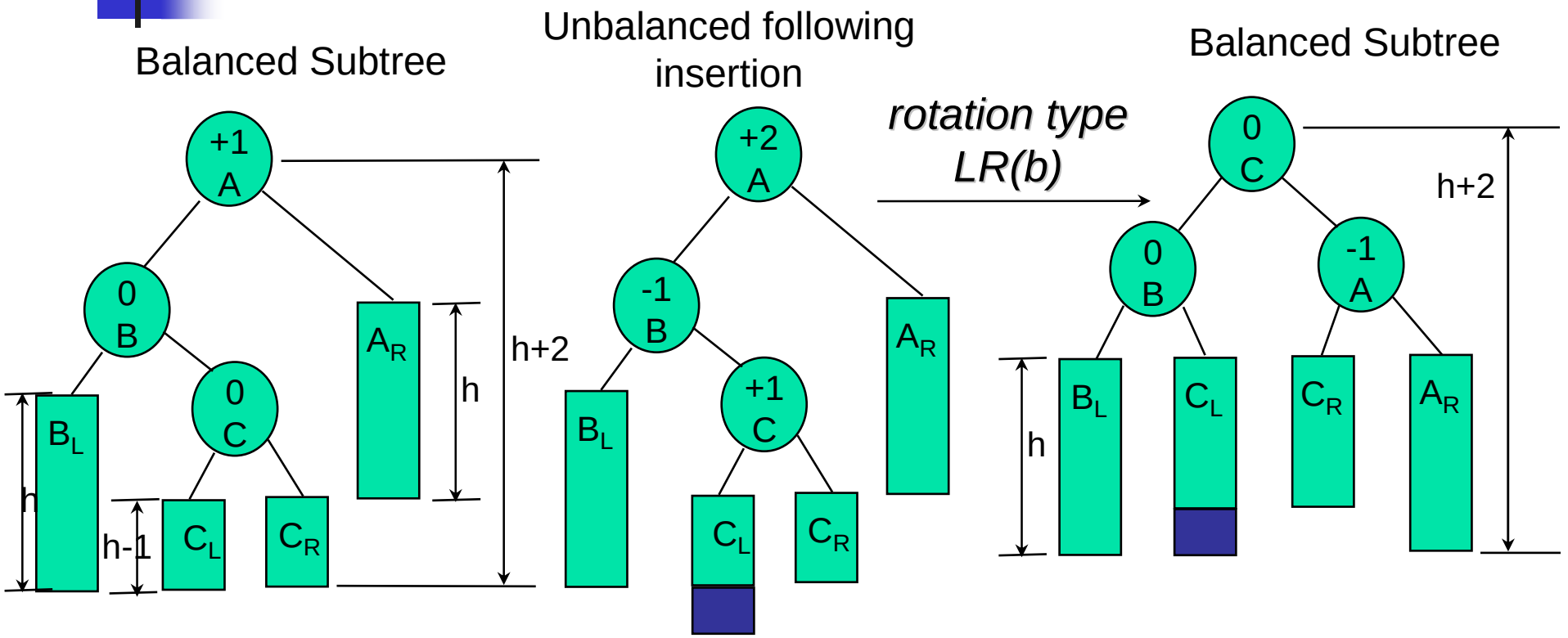
Height of B_R increase to h+1
(AL < A < BL < B < BR)

LR-a Rebalancing after Insertion



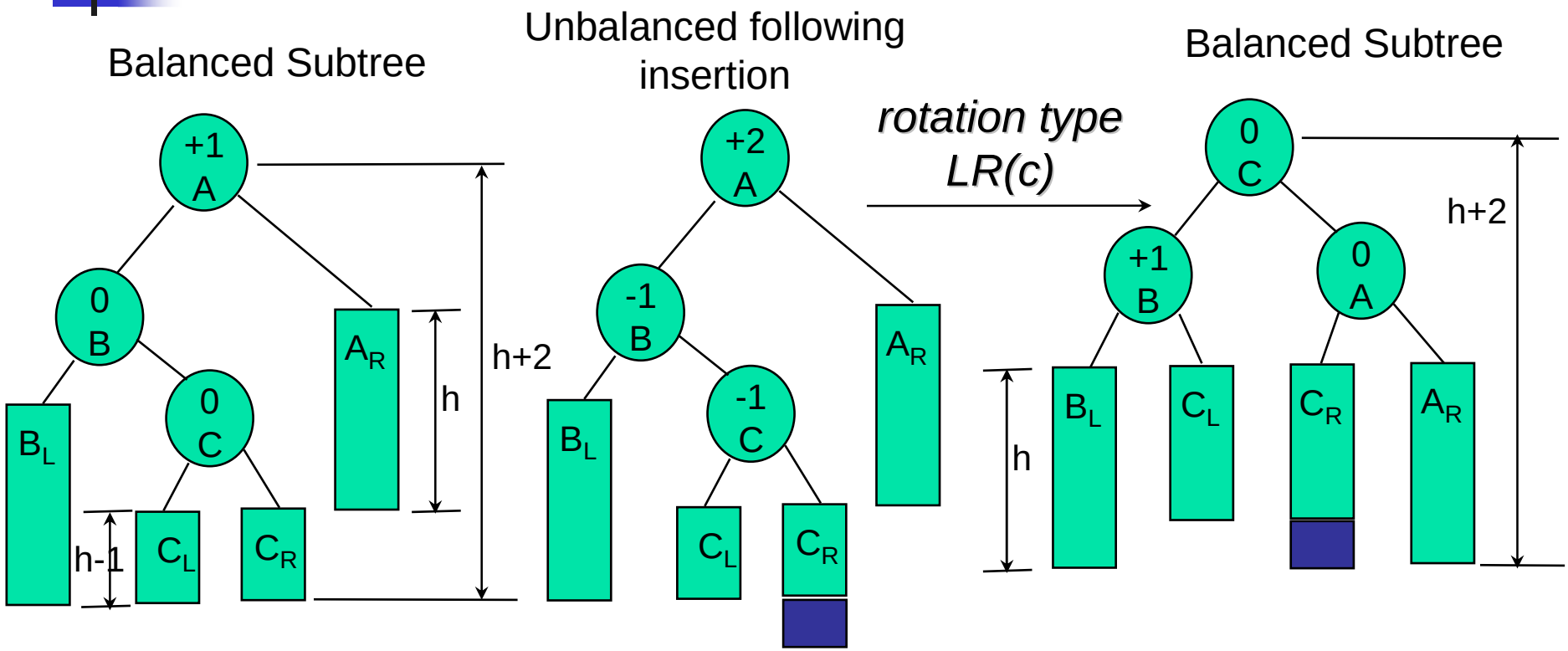
(B < C < A)

LR-b Rebalancing after Insertion



(BL < B < CL < C < CR < A < AR)

LR-c Rebalancing after Insertion



RL a, b and c are symmetric to LR a, b and c

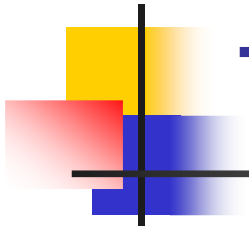
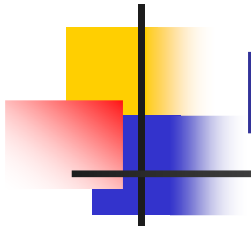


Table of Contents

- AVL TREES
  - Definition
  - Searching an AVL Search Tree
  - Inserting into an AVL Search Tree
  - Deletion from an AVL Search Tree
- RED-BLACK TREES
- SPLAY TREES
- B-TREES



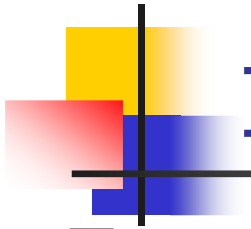
Deletion from an AVL Tree

- Let q be the parent of the node that was physically deleted
- If the deletion took place from
  - the left subtree of q ➔ bf(q) decreases by 1
  - the right subtree of q ➔ bf(q) increases by 1
- Observations

D1 : If the new BF of q is 0, its height has decreased by 1.
we need to change the BF of its parent (if any) and possibly those of its other ancestors

D2 : If the new BF of q is either –1 or 1, its height is the same as before the deletion and the BFs of tis ancestors and unchanged

D3 : If the new BF of q is either –2 or 2, the tree is unbalanced at q

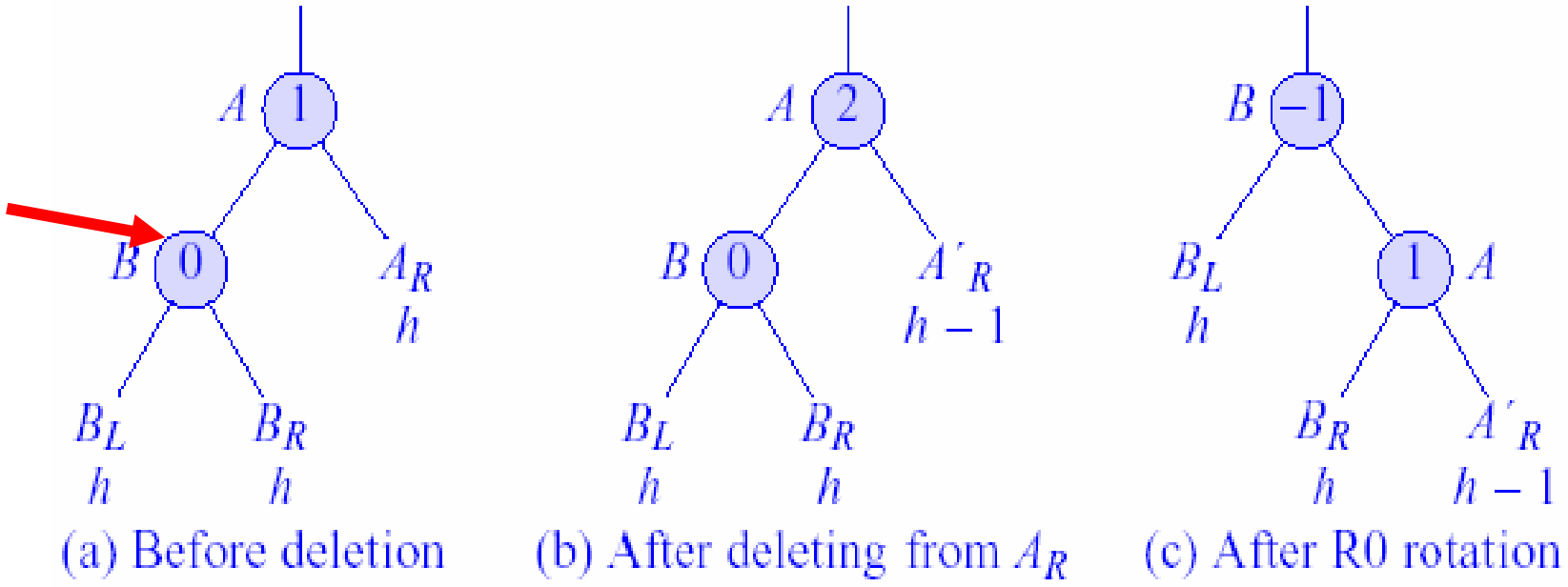


Imbalance Patterns due to Deletion

- Type L
  - If the deletion took place from A's left subtree with root B
  - Subclassified : L-1, L0 and L1 depending on bf(B)
- Type R
  - If the deletion took place from A's right subtree with root B
  - Subclassified : R-1, R0 and R1 depending on bf(B)

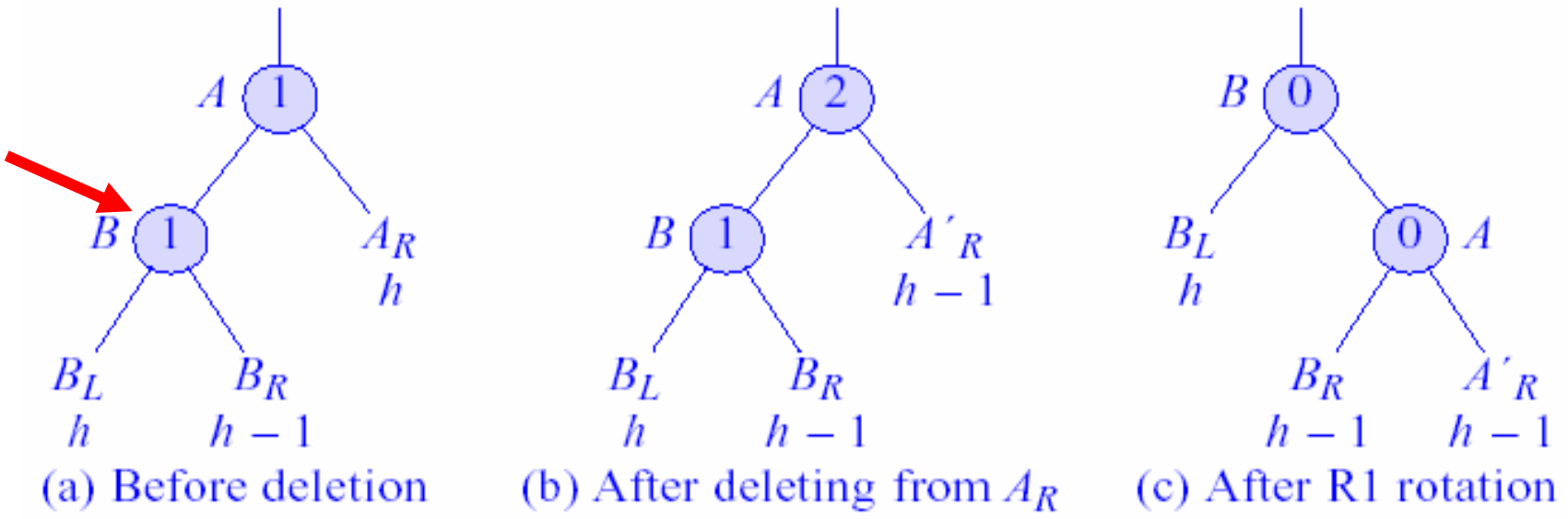
R0 rotation after Deletion

- Height of tree is h+2 (h+2) before (after) deletion
- Single rotation is sufficient
- BL < B < BR < A < AR



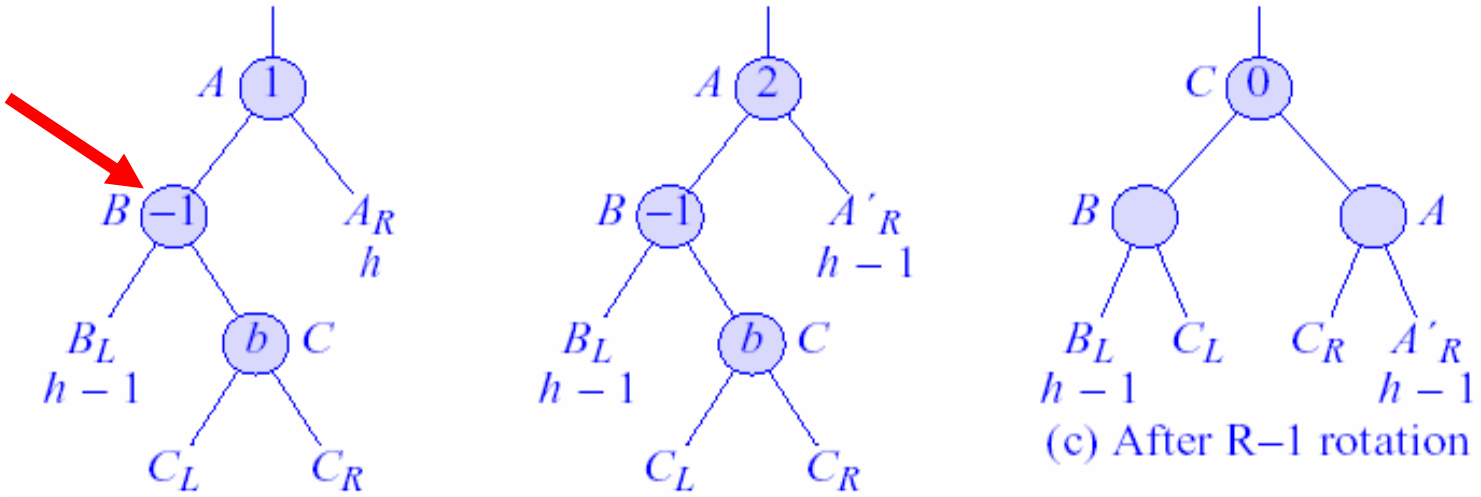
R1 rotation after Deletion

- Height of tree is h+2 (h+1) before (after) deletion
- Single rotation is sufficient
- BL < B < BR < A < AR



R-1 rotation after Deletion

- Height of tree is h+2 (h + 1) before (after) deletion
- Double rotations
- BL < B < CL < C < CR < A < AR



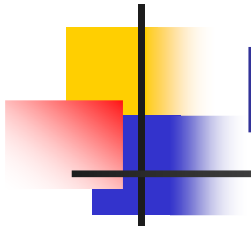
(a) Before deletion (b) After deleting from A_R

(c) After R–1 rotation

$b = 0 \Rightarrow bf(A) = bf(B) = 0$ after rotation

$b = 1 \Rightarrow bf(A) = -1$ and $bf(B) = 0$ after rotation

$b = -1 \Rightarrow bf(A) = 0$ and $bf(B) = 1$ after rotation



Rotation Taxonomy in AVL

- Rotation types due to Insertion
  - LL type
  - RR type
  - LR type: LR-a, LR-b, LR-c
  - RL type: RL-a, RL-b, LR-c
- Rotation types due to Deletion
  - R type: R-1, R0, R1
  - L type: L-1, L0, L1
- LL rotation in insertion and R1 rotation in deletion are identical
- LR rotation in insertion and R-1 rotation in deletion are identical
- LL rotation in insertion and R0 rotation in deletion differ only in the final BF of A and B

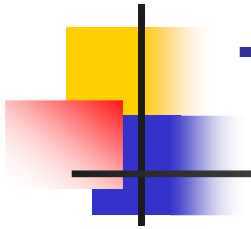
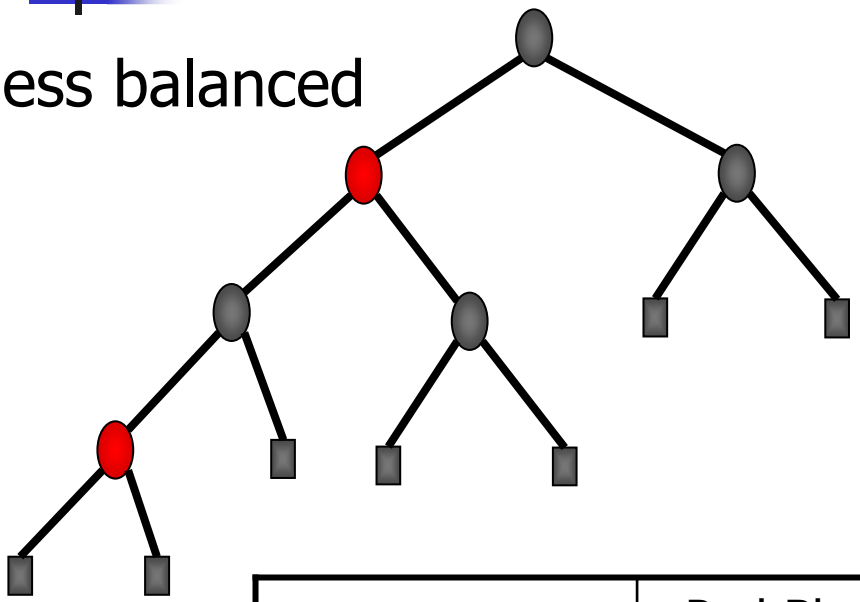


Table of Contents

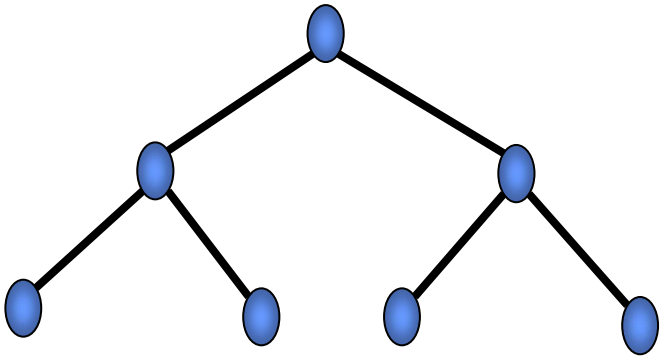
- AVL TREES
- RED-BLACK TREES
  - *Definition*
  - Searching a Red-Black Tree
  - Inserting into a Red-Black Tree
  - Deletion from a Red-Black Tree
  - Implementation Considerations and Complexity
- SPLAY TREES
- B-TREES

Red-Black Tree vs. AVL Tree (1)

less balanced

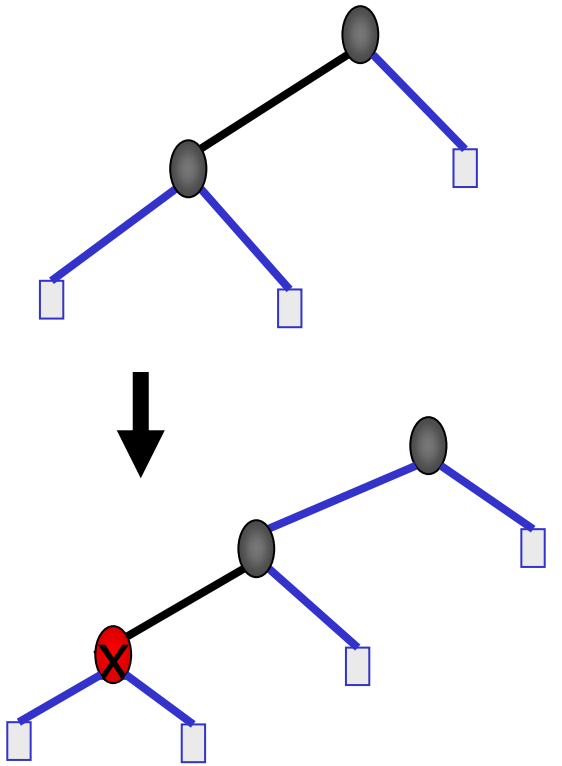


more balanced

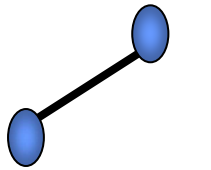


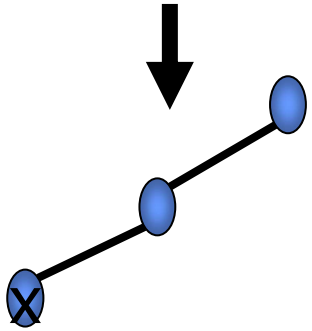
	Red-Black tree	AVL tree
Lookup	O(log*n*)	O(log*n*)
Insertion	O(log*n*)	O(log*n*)
Deletion	O(log*n*)	O(log*n*)

Red-Black Tree vs. AVL Tree (2)



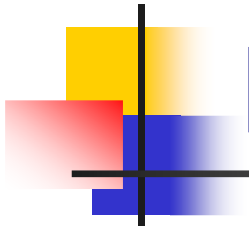
insert a node x





Red-black tree doesn't need rebalancing

AVL tree needs rebalancing



Red-Black Tree: Definition

- Red-black tree
  - Binary Search tree
  - Every node is colored red or black

RB1. Root and all external nodes are black.

RB2. No root-to-external-node path has two consecutive red nodes.

RB3. All root-to-external-node paths have the same number of black nodes

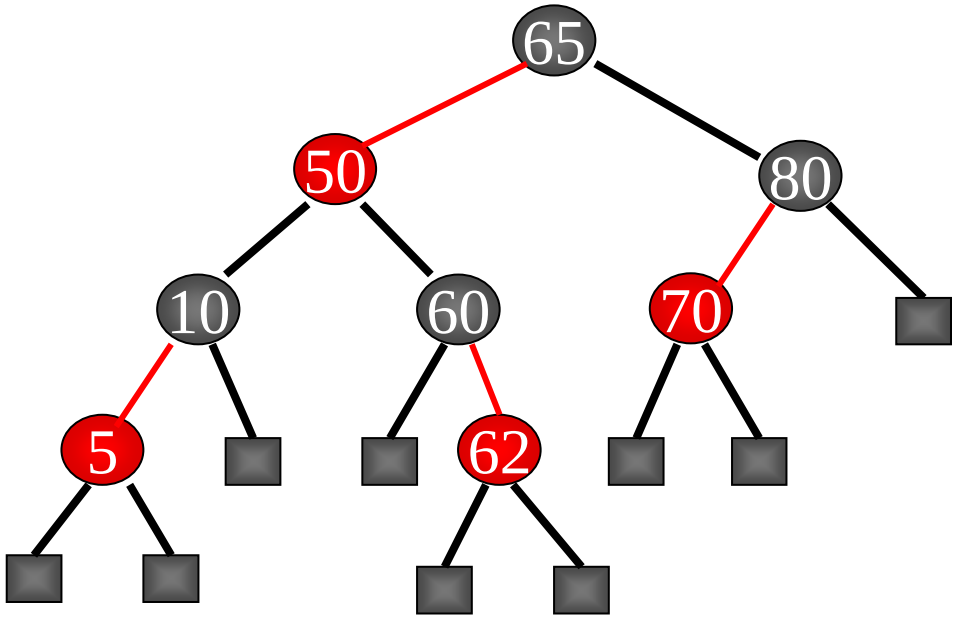
||| equivalent

RB1′. Pointers from an internal node to an external node are black

RB2′. No root-to-external-node path has two consecutive red pointers

RB3′. All root-to-external-node paths have the same number of black pointers

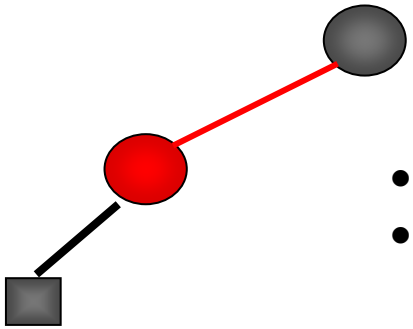
Red-Black Tree: Example



- Every path from the root to an external node has exactly 2 black pointers and 3 black nodes
- No such path has two consecutive red nodes or pointers
- Small black box nodes are for ensuring every node has two children
- The color of newly inserted node is red

RBT: Glossary

- Rank: number of black pointers on any path from the node to any external node in red-black tree
- Length (of a root-to-external-node path): number of pointers on the path.

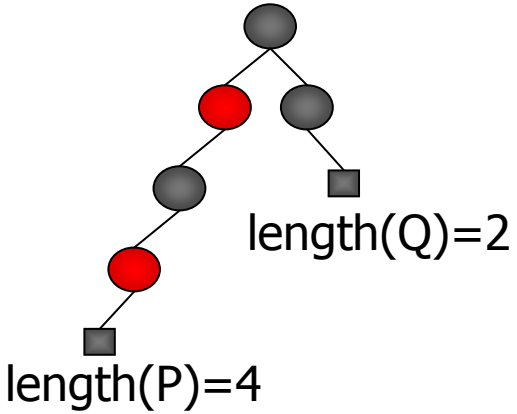


- rank = 1
- height = length = 2

RBT: Lemma 1

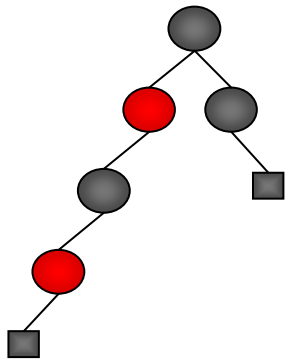
- Lemma 1
  - If P and Q are two root-to-external-node paths in a red-black tree,

    Then length(P) ≤ 2 * length(Q)
- Proof
  - Suppose that the rank of the root is r
  - From RB1' and RB2', each root-to-external-node path has between r and 2r pointers
  - So length(P) ≤ 2length(Q)



RBT: Lemma 2

- Lemma 2
  - h : height of a red-black tree
  - n : number of internal nodes
  - r : rank of the root



h=4
n=5
r=2

  - (a) $h \leq 2r$
    - From Lemma 16.1, no root-to-external-node path has length > 2r
  - (b) $n \geq 2^r - 1$
    - No external nodes at levels 1 through r so $2^r - 1$ internal nodes at these levels
  - (c) $h \leq 2\log_2(n+1)$
    - $2^r \leq n + 1$ from (b)
    - $r \leq \log_2(n+1)$
    - $f \leq 2r \leq 2\log_2(n+1)$

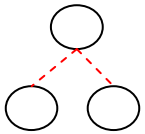
RBT: Representation

- Null pointers represent external nodes
- Pointer and node colors are closely related
- **Each node** we need to store only
  its color (one additional bit per node) or
  the color of the two pointers to its children
  (two additional bit per node)

→ null pointer

→ R / B

or



→ {R / B, R / B}

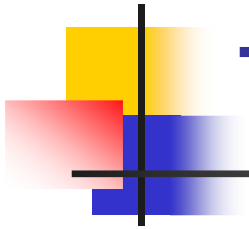


Table of Contents

- AVL Tree
- RED-BLACK TREES
  - Definition
  - *Searching a Red-Black Tree*
  - Inserting into a Red-Black Tree
  - Deletion from a Red-Black Tree
  - Implementation
- Splay Tree
- B Tree

Searching a Red-Black Tree

- Use the same code to search ordinary binary search tree (Program 15.4), AVL tree, red-black trees

```
if(root == null) {
        search is unsuccessful
} else {
        if ( thekey < key in root)
                        only left subtree is to be searched
        } else {
                        if(thekey > key in root)
                                only right subtree is to be searched
                        else (thekey == key in root)
                                search terminates successfully
                }
}
```

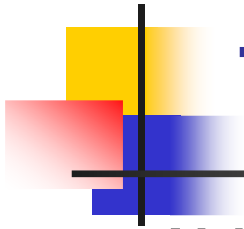
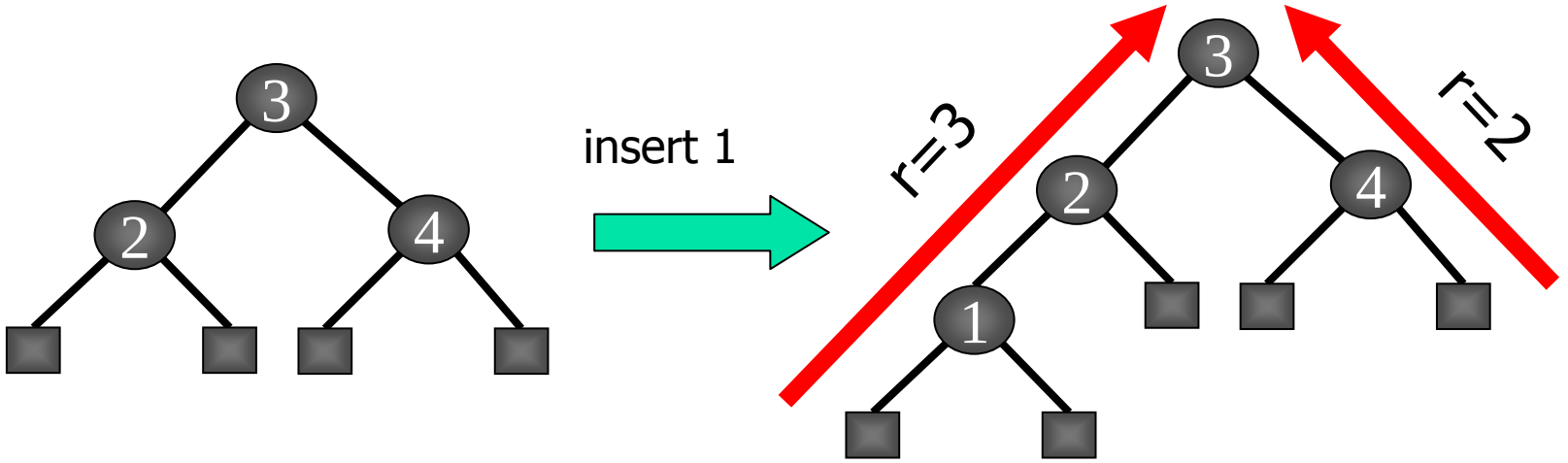


Table of Contents

- AVL Tree
- RED-BLACK TREES
  - Definition
  - Searching a Red-Black Tree
  - *Inserting into a Red-Black Tree*
  - Deletion from a Red-Black Tree
  - Implementation
- Splay Tree
- B Tree

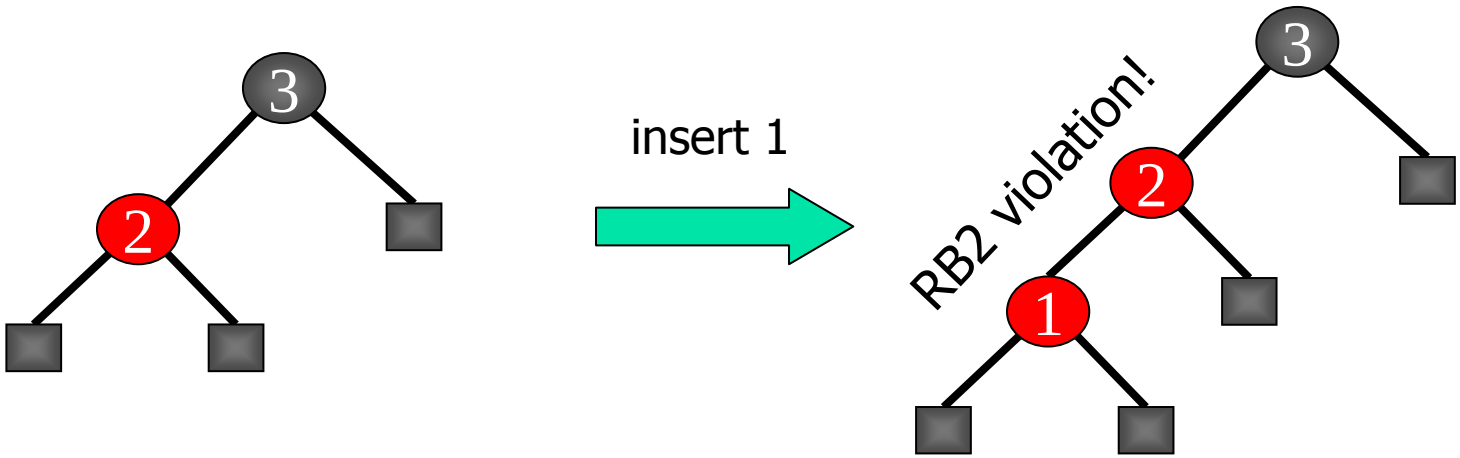
Violations due to Insertion (1)

- The RBT should have the same number of black nodes in all paths
- If new node is colored as black
  - The updated tree will always violate RB3 (same number of black nodes)



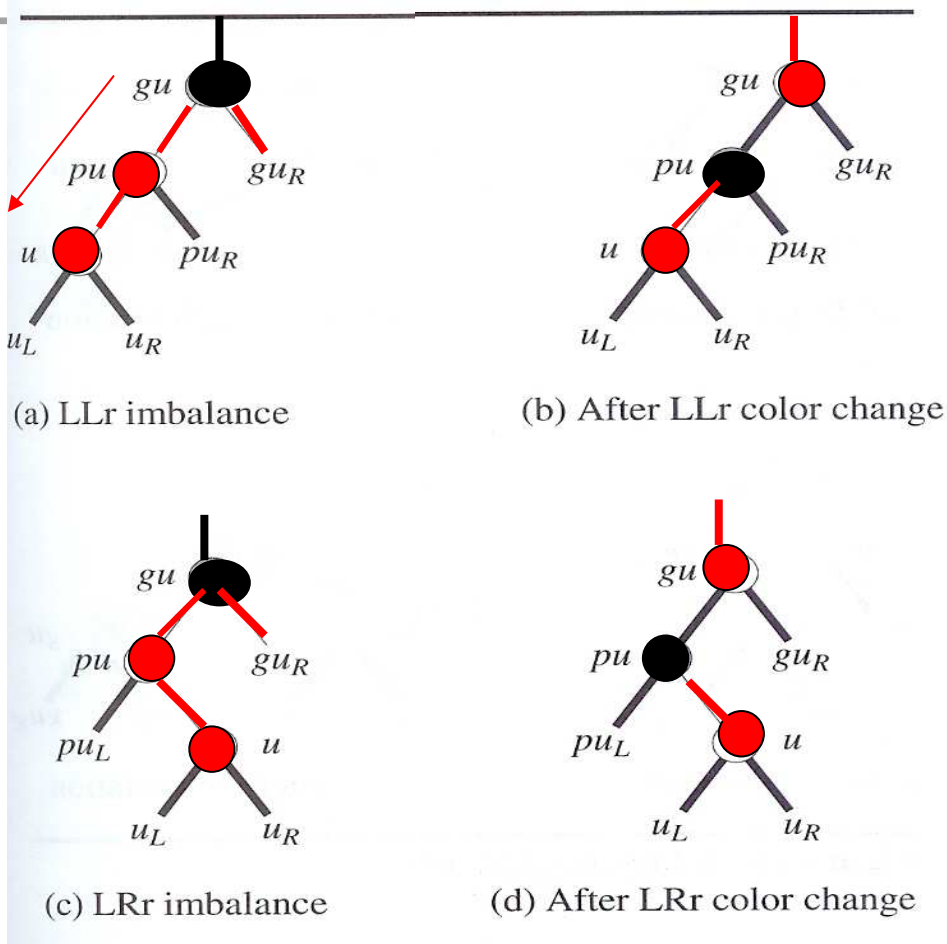
Violations due to Insertion (2)

- If new node is colored as red
  - If the parent of inserted node is black, it's OK (no violation).
  - But if the parent of inserted node is also red, violation occurs!
    - Violate RB2 (no two consecutive reds)



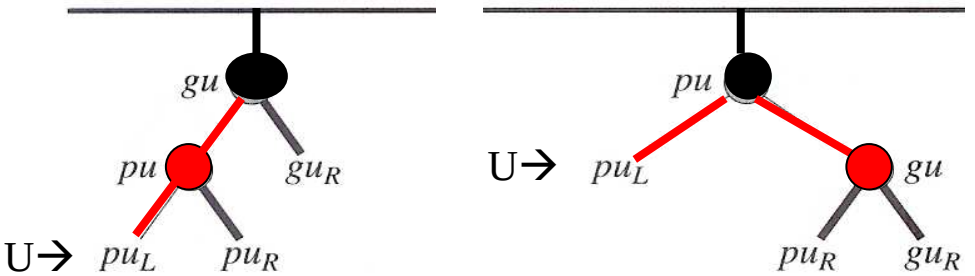
L Type Imbalances due to Insertion (1)

- *u* be the inserted node (red)
  - *uL* & *uR*
- *pu* be the parent of *u* (red)
  - *puL* & *puR*
- *gu* be the granparent of *u*
  - *guL* & *guR*
- LLr & LRr
  - The color of *guR* *is* *red*

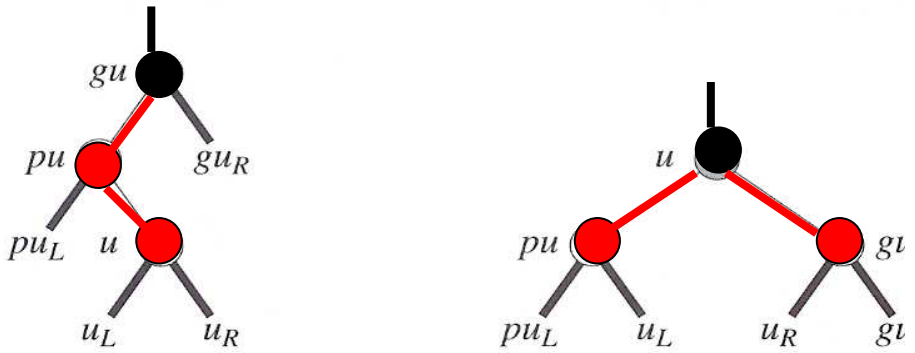


L Type Imbalances due to Insertion (2)

- *u* be the inserted node (red)
  - *uL & uR*
- *pu* be the parent of *u* (red)
  - *puL & puR*
- *gu* be the granparent of *u*
  - *guL & guR*
- LLb & LRb
  - The color of *guR is black*



(a) LLb imbalance (b) After LLb rotation



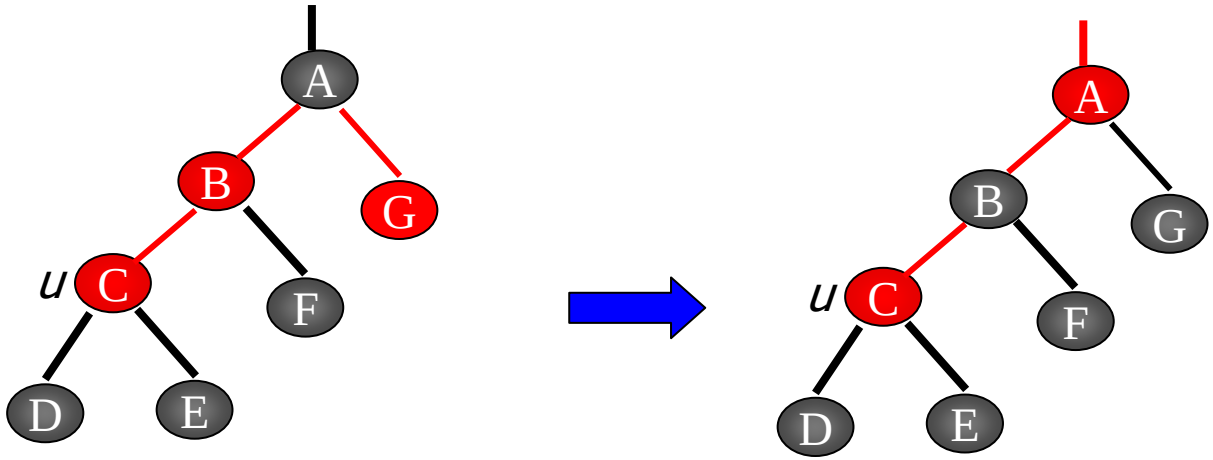
(c) LRb imbalance (d) After LRb rotation

Fixing LLr and LRr Imbalance

Begin
  change the color of pu & gu_R : red → black
  if (gu != root) { change the color of gu : black → red }
  else { the color change not done.
    the number of black nodes increases by 1.
    (on all root-to-external-node paths) }
  if (the color change of gu causes imbalance)
    gu became the new u node
  if (gu != root && the color change causes imbalance)
    continue to rebalance
End

Fixing LLr Imbalance

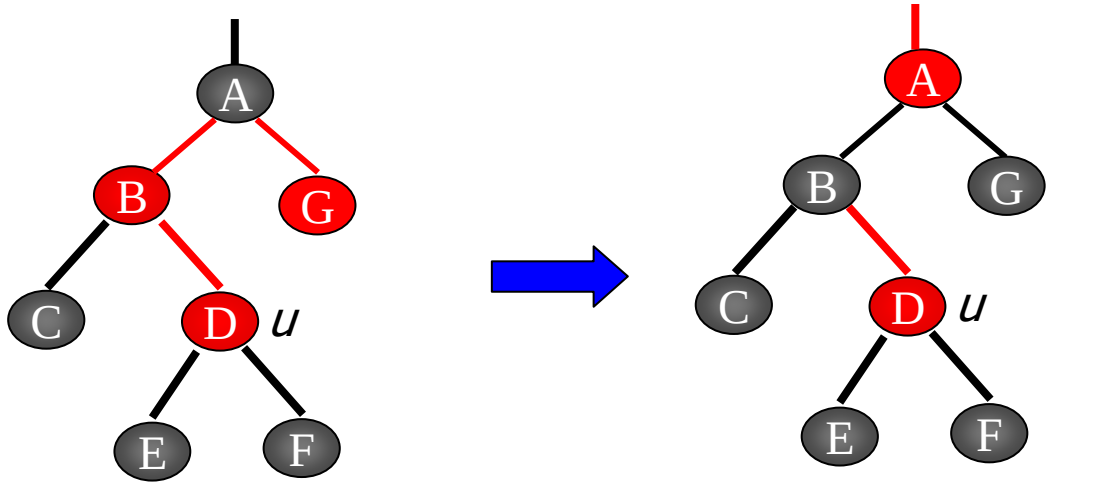
If a node (which is red) *u* is left child of its parent (also red)
and its parent is left child of its grandparent & its uncle is red,
then change its grandparent's color to red &
change its parent's and uncle's color to black



LLr imbalance **After LLr color change**

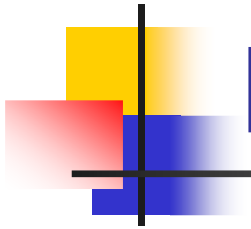
Fixing LRr Imbalance

If a node (which is red) *u* is right child of its parent (also red)
and its parent is left child of its grandparent & its uncle is red,
then change its grandparent's color to red
& change its parent's and uncle's color to black



LRr imbalance

After LRr color change



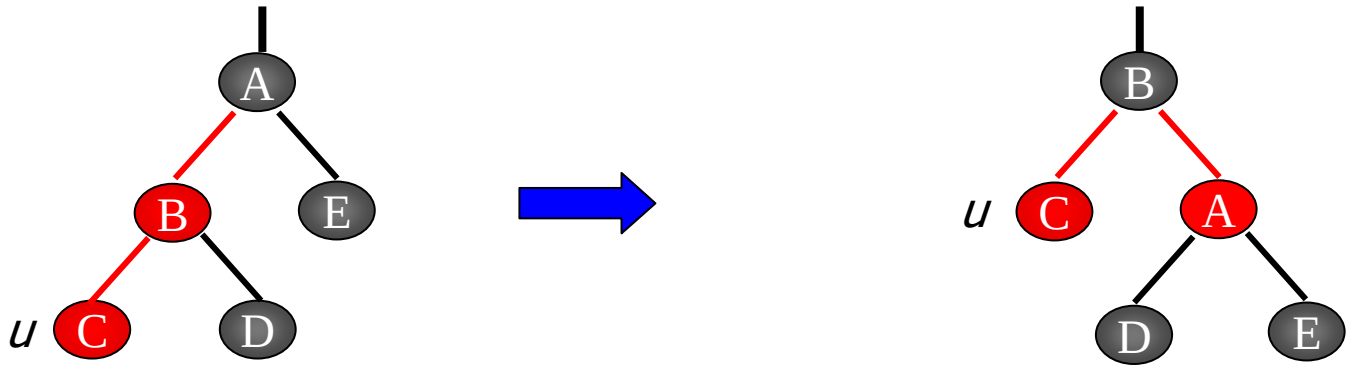
Fixing LLb and LRb Imbalance

- Rotation first & then Change the color
  - The root of the involved subtree is black following the rotation
  - Number of black nodes on all root-to-external-node paths is unchanged

- LLb rotation in RB tree is similar to LL rotation in AVL tree
- LRb rotation in RB tree is similar to LR rotation in AVL tree

Fixing LLb Imbalance

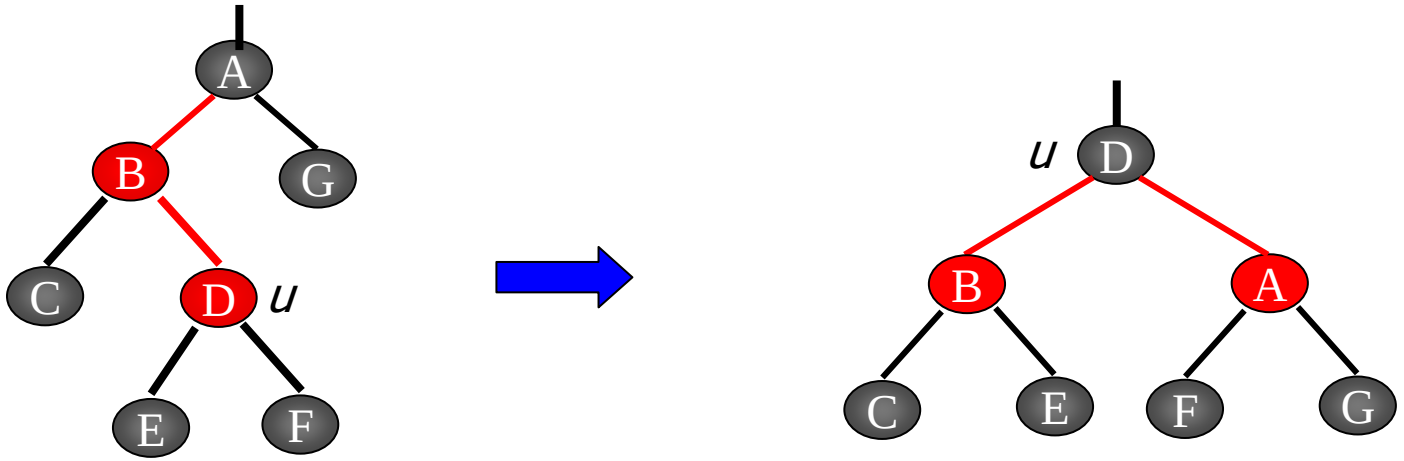
If a node (which is red) *u* is left child of its parent(also red)
and its parent is left child of its grandparent & its uncle is black,
then do rotation and color change like the following



LLb imbalance **After LLb rotation**

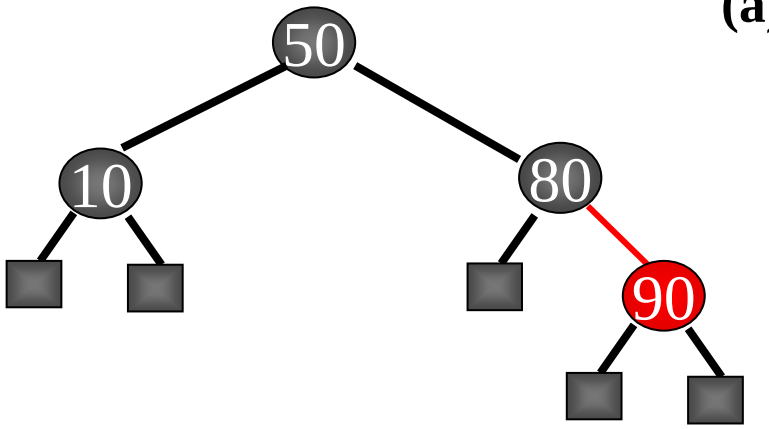
Fixing LRb Imbalance

If a node (which is red) *u* is right child of its parent (also red)
and its parent is left child of its grandparent & its uncle is black,
then do rotation and color change like the following



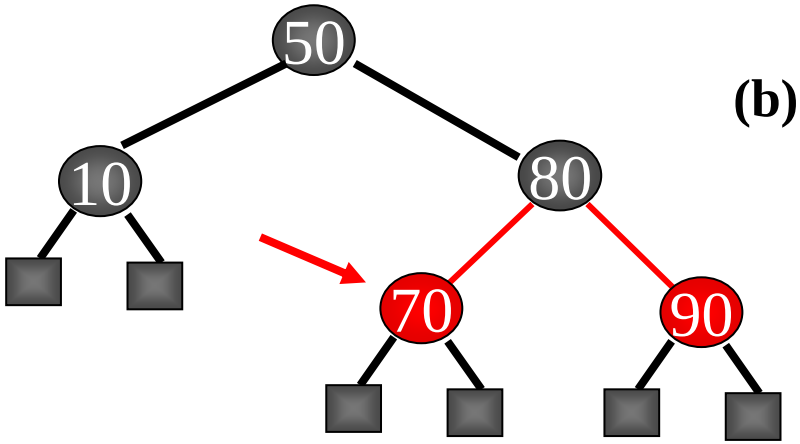
LRb imbalance **After LRb rotation**

Insertion Example in RBT (1)



(a) Initial state:

all root-to-external-node paths have 3 black nodes & 2 black pointers

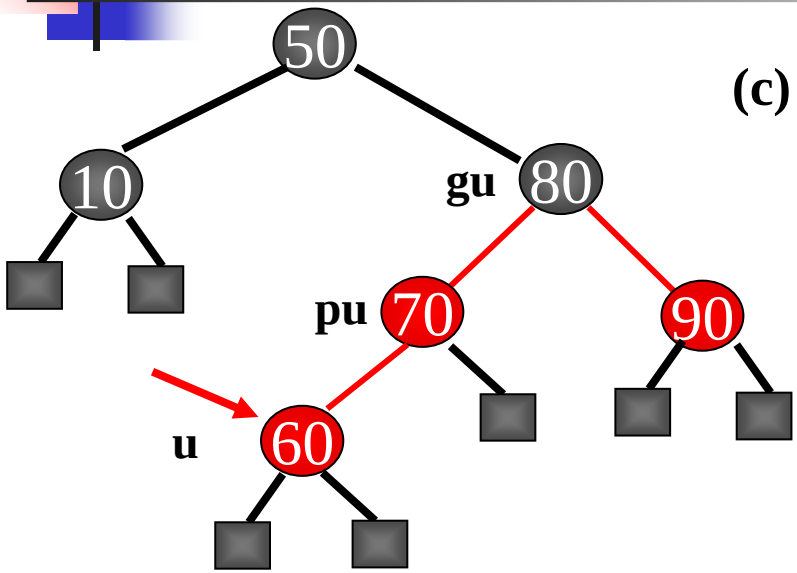


(b) insert 70 as a red node:

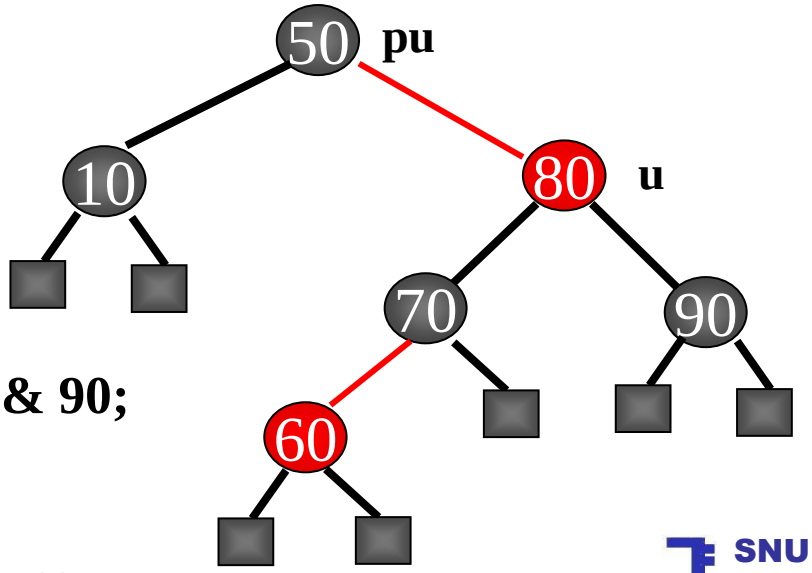
No violations of RBT ➔

No remedial action is necessary

Insertion Example in RBT (2)

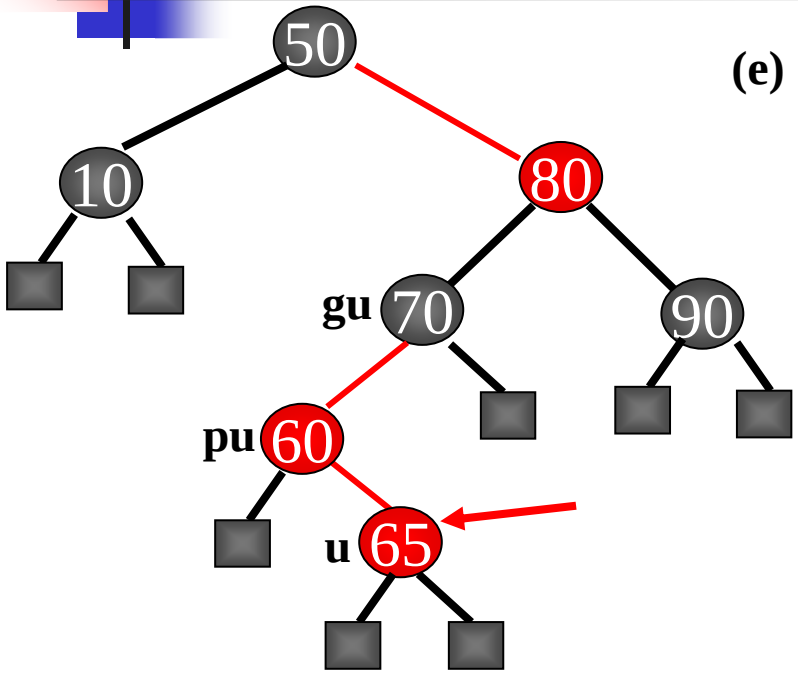


(c) insert 60 as a red node ➔ LLr imbalance

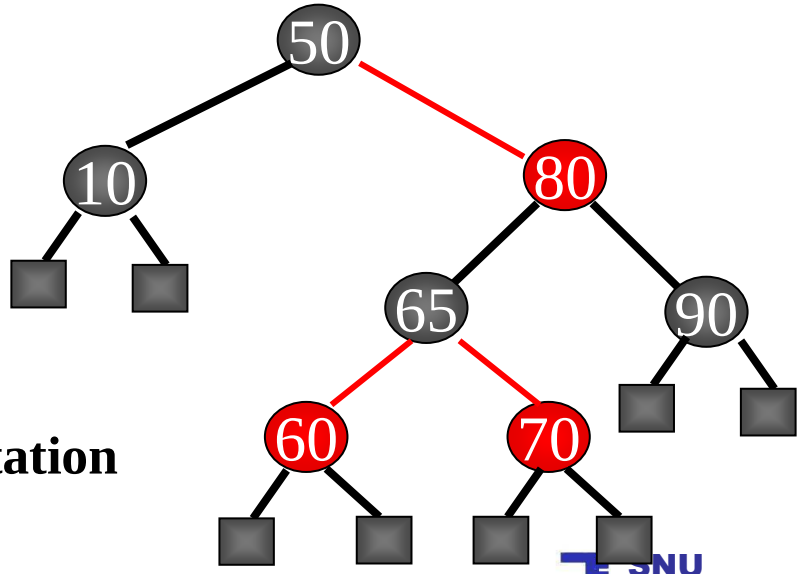


(d) LLr color change on nodes 70, 80 & 90;
gu is null, so not RB2 imbalance

Insertion Example in RBT (3)

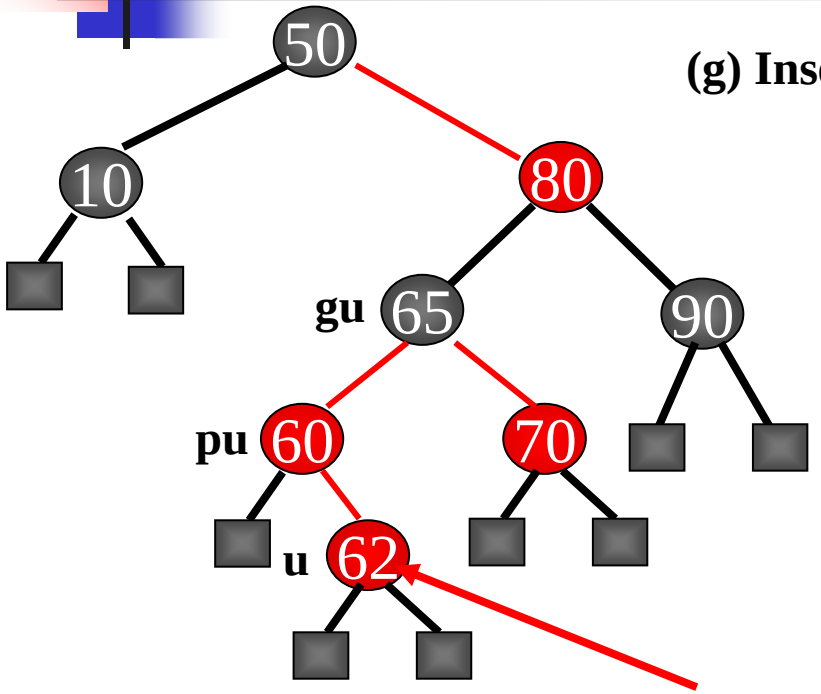


(e) Insertion 65 as a red node ➔ LRb imbalance

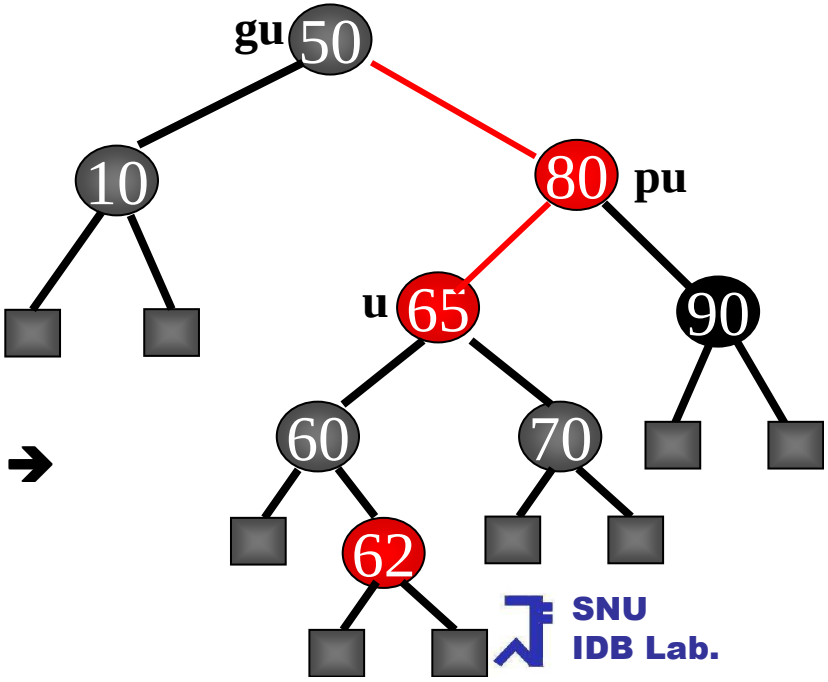


(f) Perform LRb rotation

Insertion Example in RBT (4)

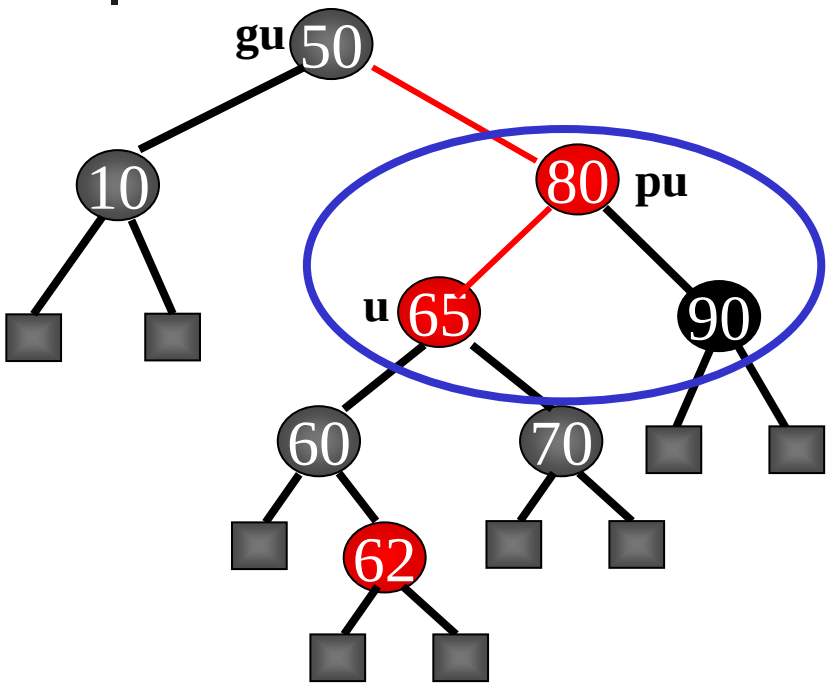


(g) Insertion 62 as a red node ➔ LRr imbalance

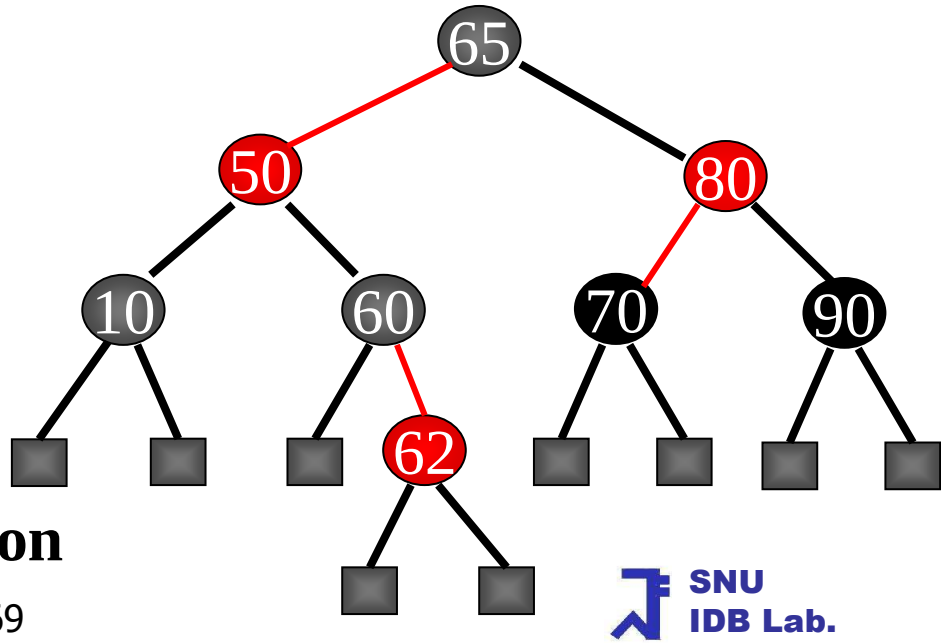


(h) LRr color change on nodes 65, 60 & 70 ➔ RLb imbalance

Insertion Example in RBT (5)



RLb imbalance



(i) Perform RLb rotation

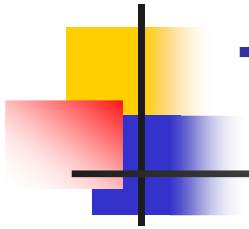
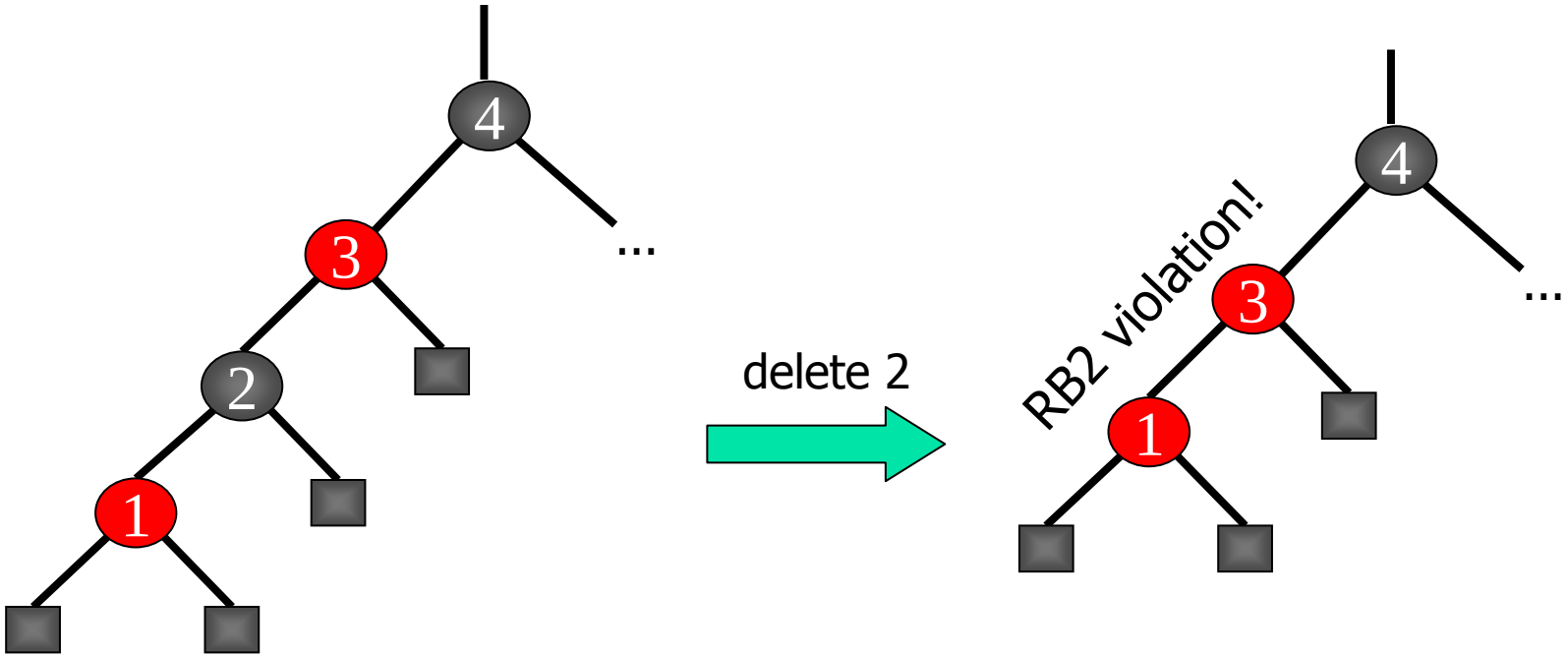


Table of Contents

- AVL Tree
- RED-BLACK TREES
  - Definition
  - Searching a Red-Black Tree
  - Inserting into a Red-Black Tree
  - *Deletion from a Red-Black Tree*
  - Implementation
- Splay Tree
- B Tree

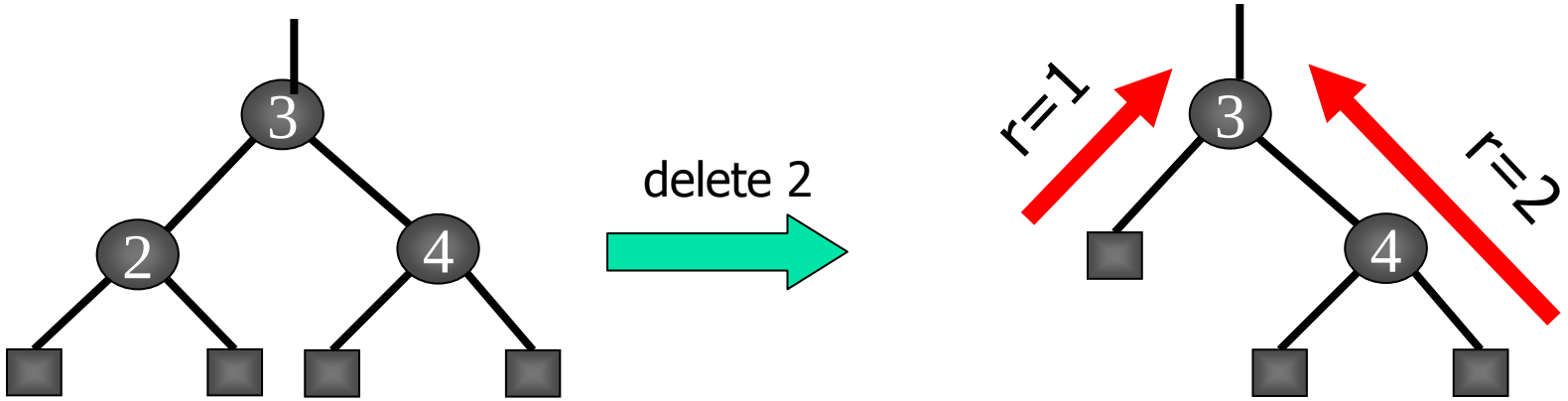
Violations due to Deletion (1)

- If the parent of deleted node is red, RB2 violation occurs!

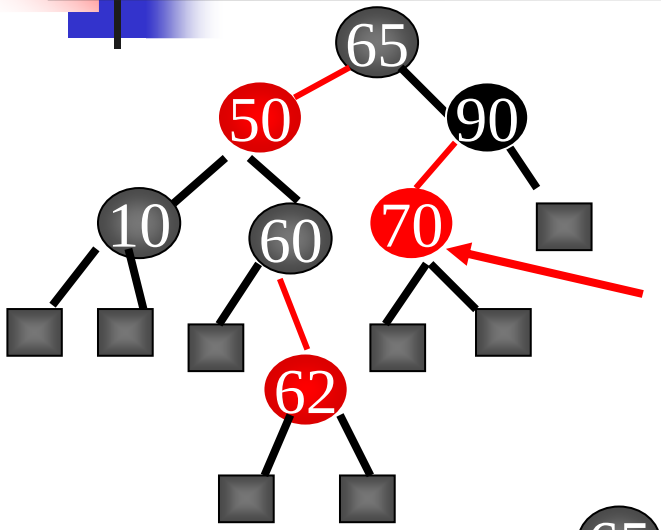


Violations due to Deletion (2)

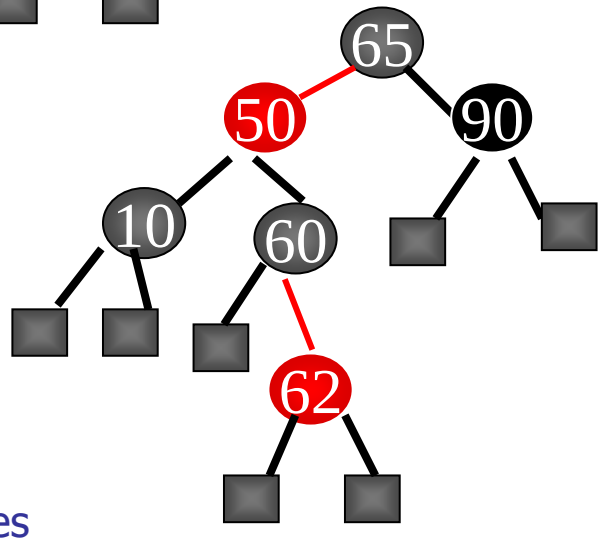
- If the deleted node is black, RB3 violation occurs!



Deletion & Imbalance in RBT (1)



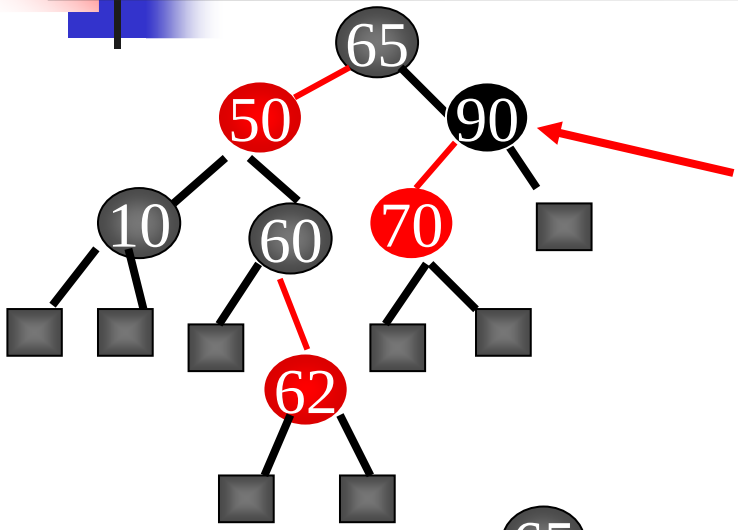
(a) A Red-Black tree



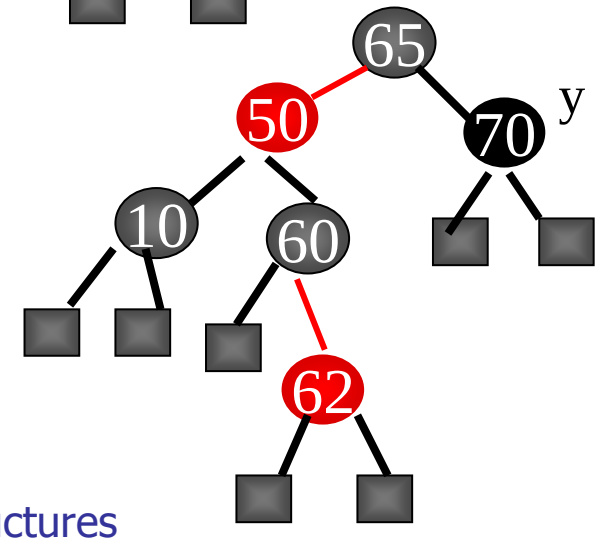
(b) Delete 70

- **Deleted node was red**
- **Same number of black nodes before and after the rotation**
- **This is OK**

Deletion & Imbalance in RBT (2)



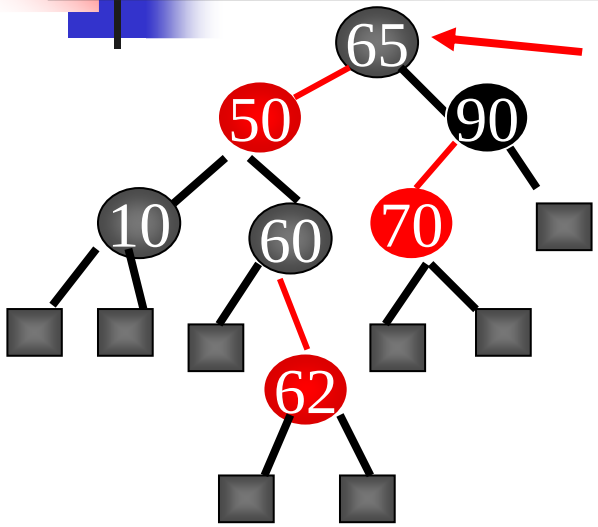
(a) A Red-Black tree



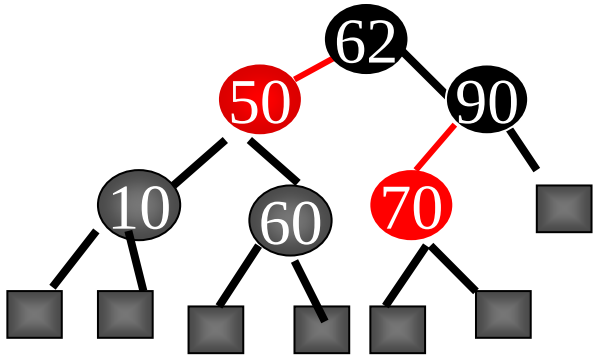
(c) Delete 90

- **The red node 70 takes the place of the deleted node which was black**
- **Then, the number of black nodes on path from root-to-external node in y is 1 less than before ➔ RB3 violation occurs = imbalance**
- **Change the color of y to Black**

Deletion & Imbalance in RBT (3)



(a) A Red-Black tree



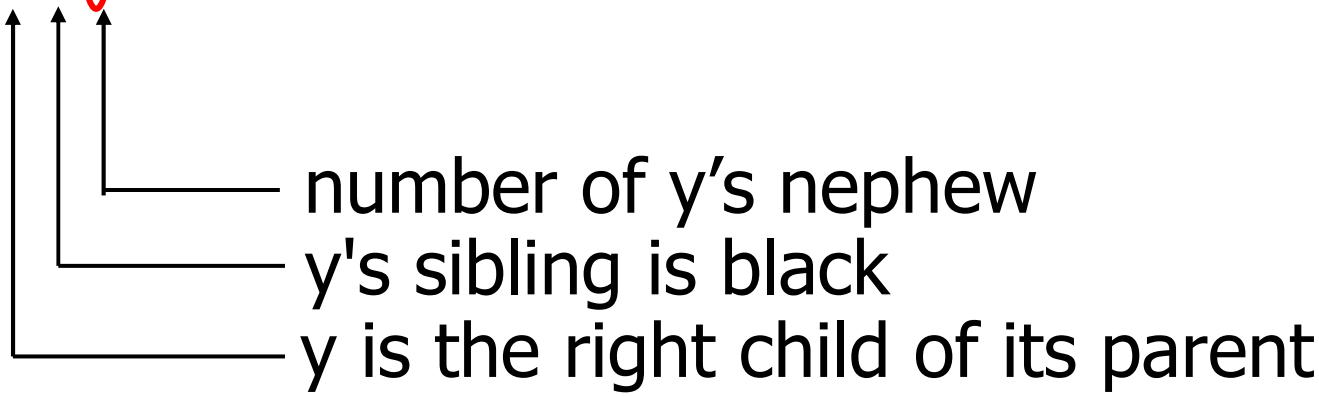
(d) Delete 65

- **Deleted node was black and the node 62 was red, so change to black**

** An RB3 violation occurs
only when the **deleted node was black**
and **y is not the root** of the resulting tree.

Rb Imbalance due to Deletion

- Rb0 => color change
- Rb1 => handled by rotation
- Rb2 => handled by rotation



(y is the node that takes the place of removed node)

Deletion Imbalances:

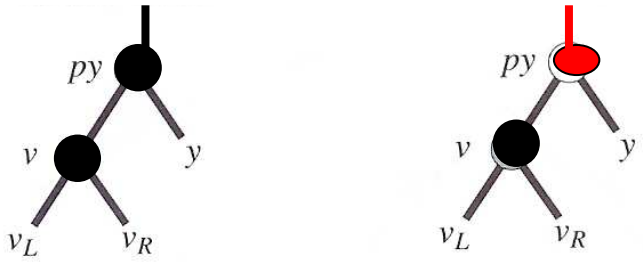
Rb family

y: the node that takes the place of removed node

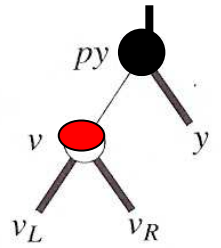
py: parent of y

v: sibling of y

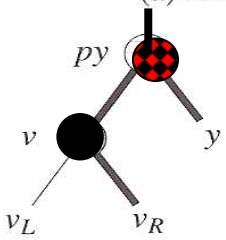
vL & vR: children of v



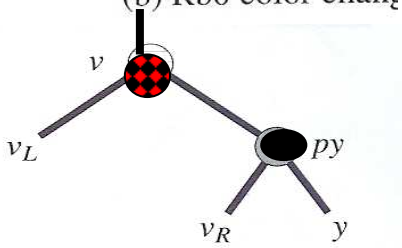
(a) Rb0 imbalance



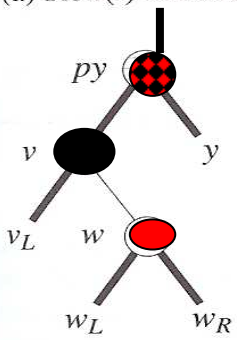
(b) Rb0 color chang



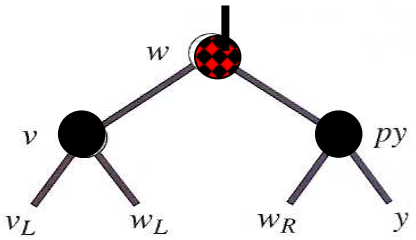
(a) Rb1(i) imbalance



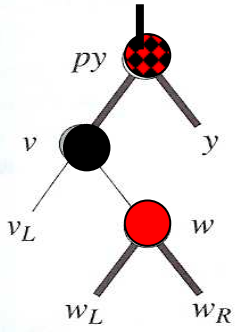
(b) After Rb1(i) rotation



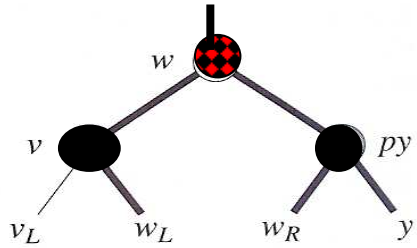
(c) Rb1(ii) imbalance



(d) After Rb1(ii) rotation



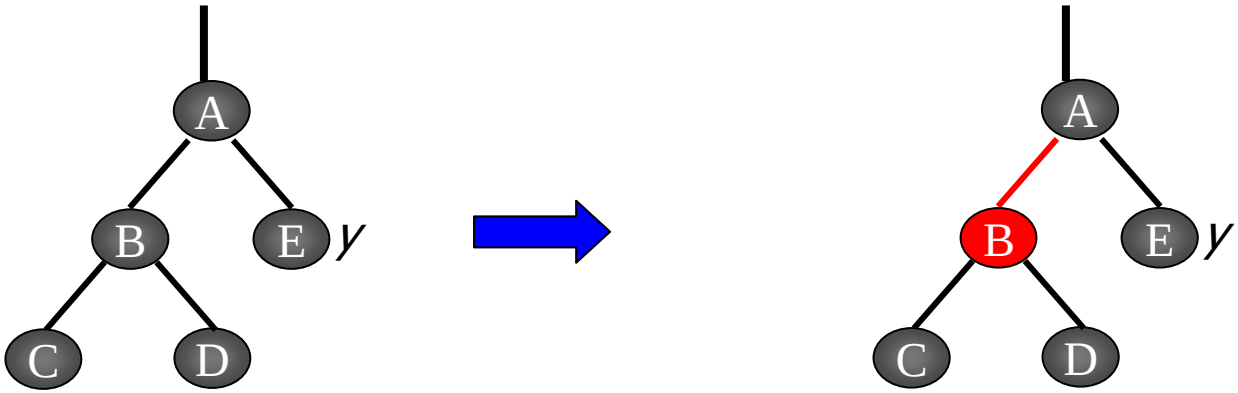
(e) Rb2 imbalance



(f) After Rb2 rotation

Fixing Rb0 Imbalance

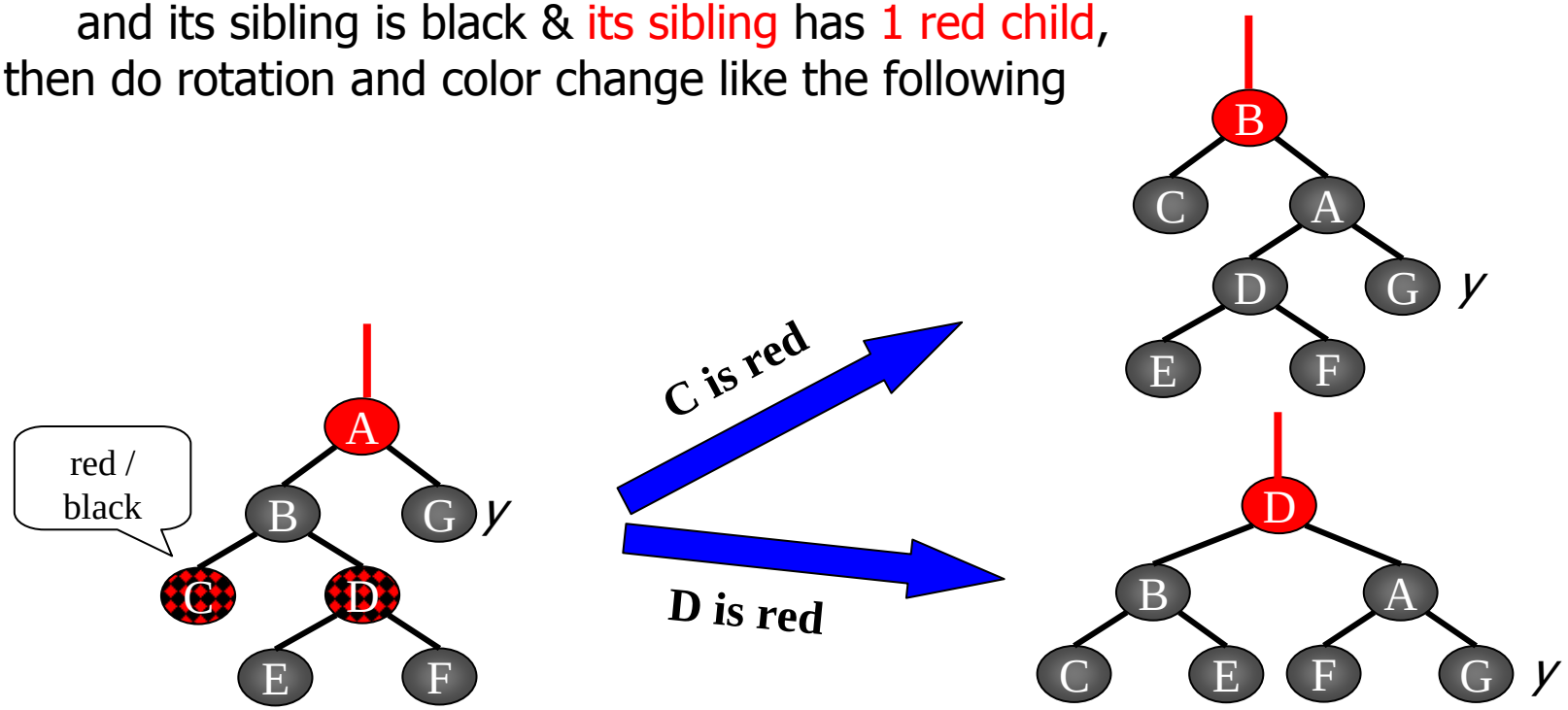
If a node (which is black) *y* is right child of its parent
and its sibling is black & its sibling has 0 red child,
then change its sibling's color to red



Rb0 imbalance **After Rb0 color change**

Fixing Rb1 Imbalance

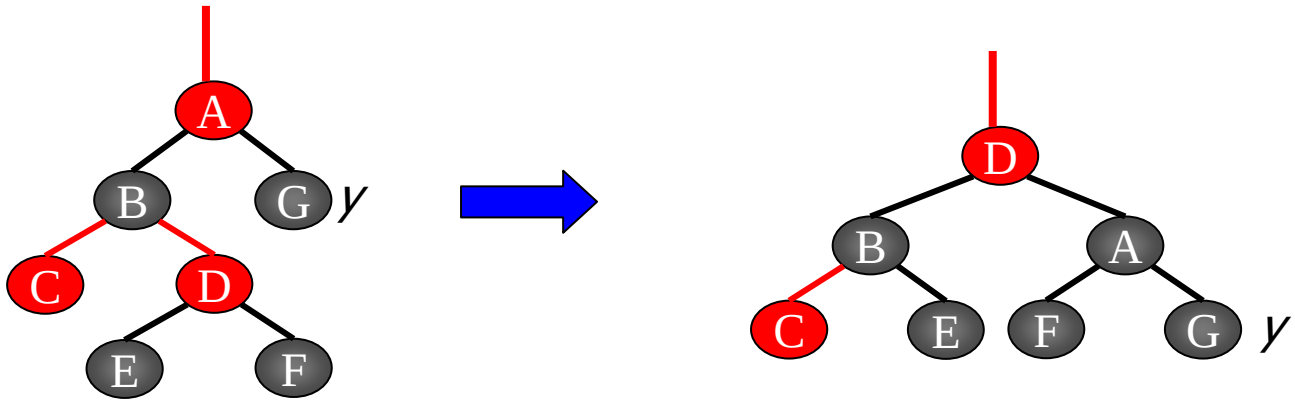
If a node(which is black) *y* is right child of its parent
and its sibling is black & its sibling has 1 red child,
then do rotation and color change like the following



Rb1 imbalance **After Rb1 rotation**

Fixing Rb2 Imbalance

If a node(which is black) *y* is right child of its parent
and its sibling is black & its sibling has 2 red children,
then do rotation and color change like the following



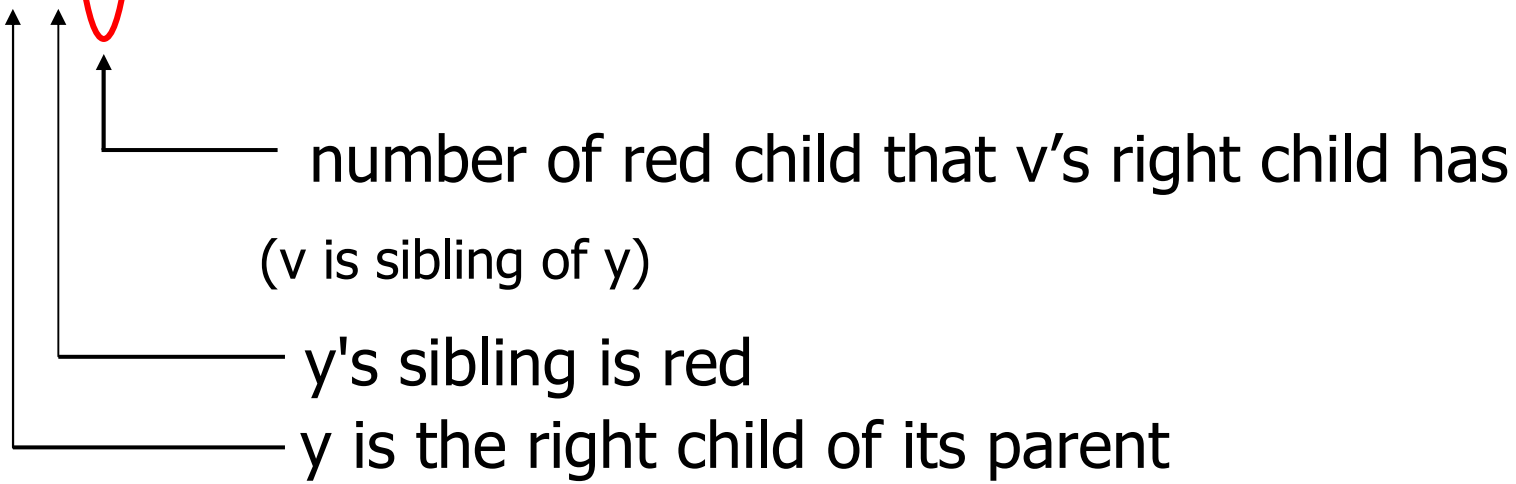
Rb2 imbalance

After Rb2 rotation

Rr Imbalance due to Deletion

- Rr0
- Rr1
- Rr2

handled by rotation



(y is the node that takes the place of removed node)

Deletion Imbalances:

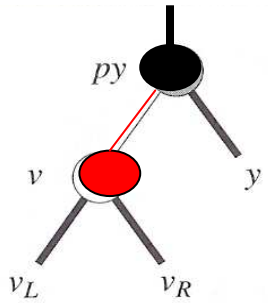
Rr family

y: the node that takes the place of removed node

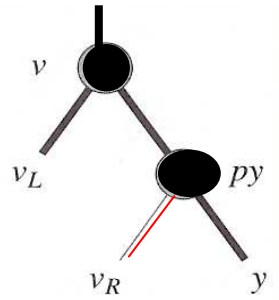
py: parent of y

v: sibling of y

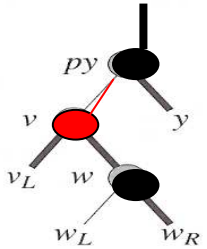
vL & vR: children of v



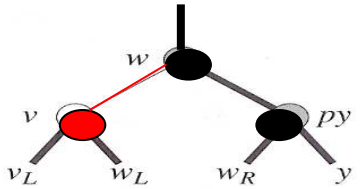
(a) Rr0 imbalance



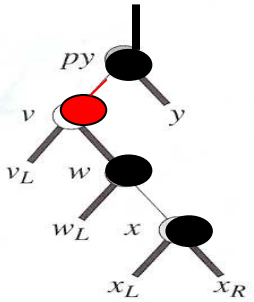
(b) Rr0 rotation



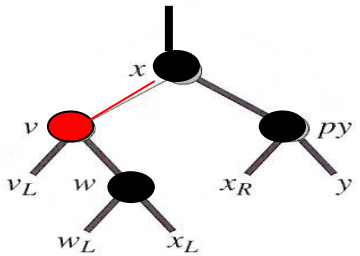
(a) Rr1(i) imbalance



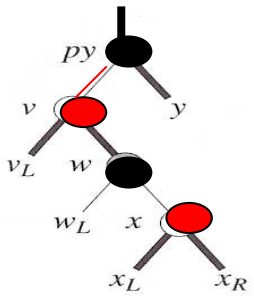
(b) After Rr1(i) rotation



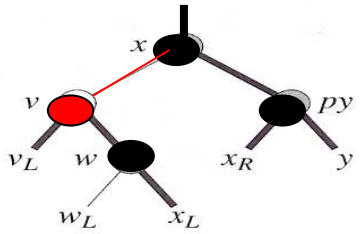
(c) Rr1(ii) imbalance



(d) After Rr1(ii) rotation



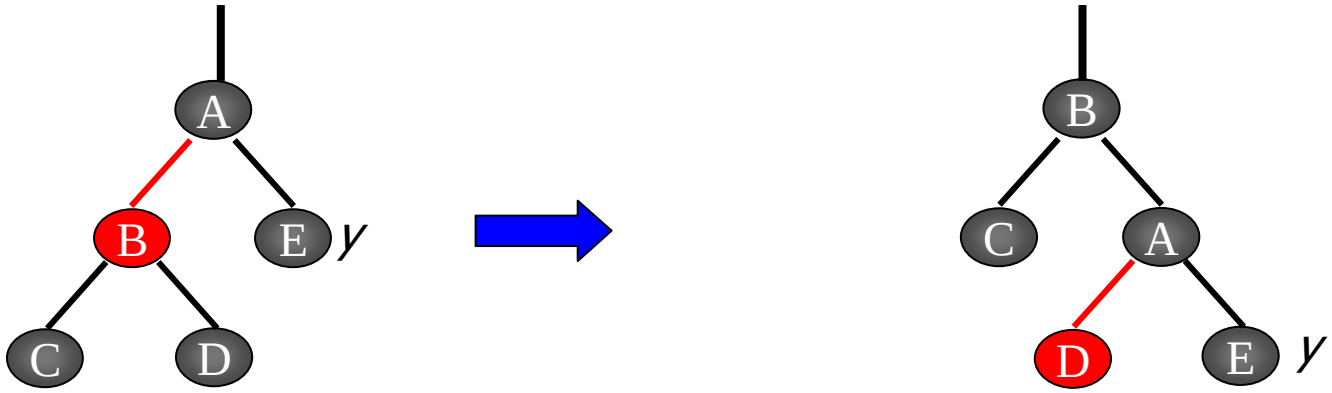
(e) Rr2 imbalance



(f) After Rr2 rotation

Fixing Rr0 Imbalance

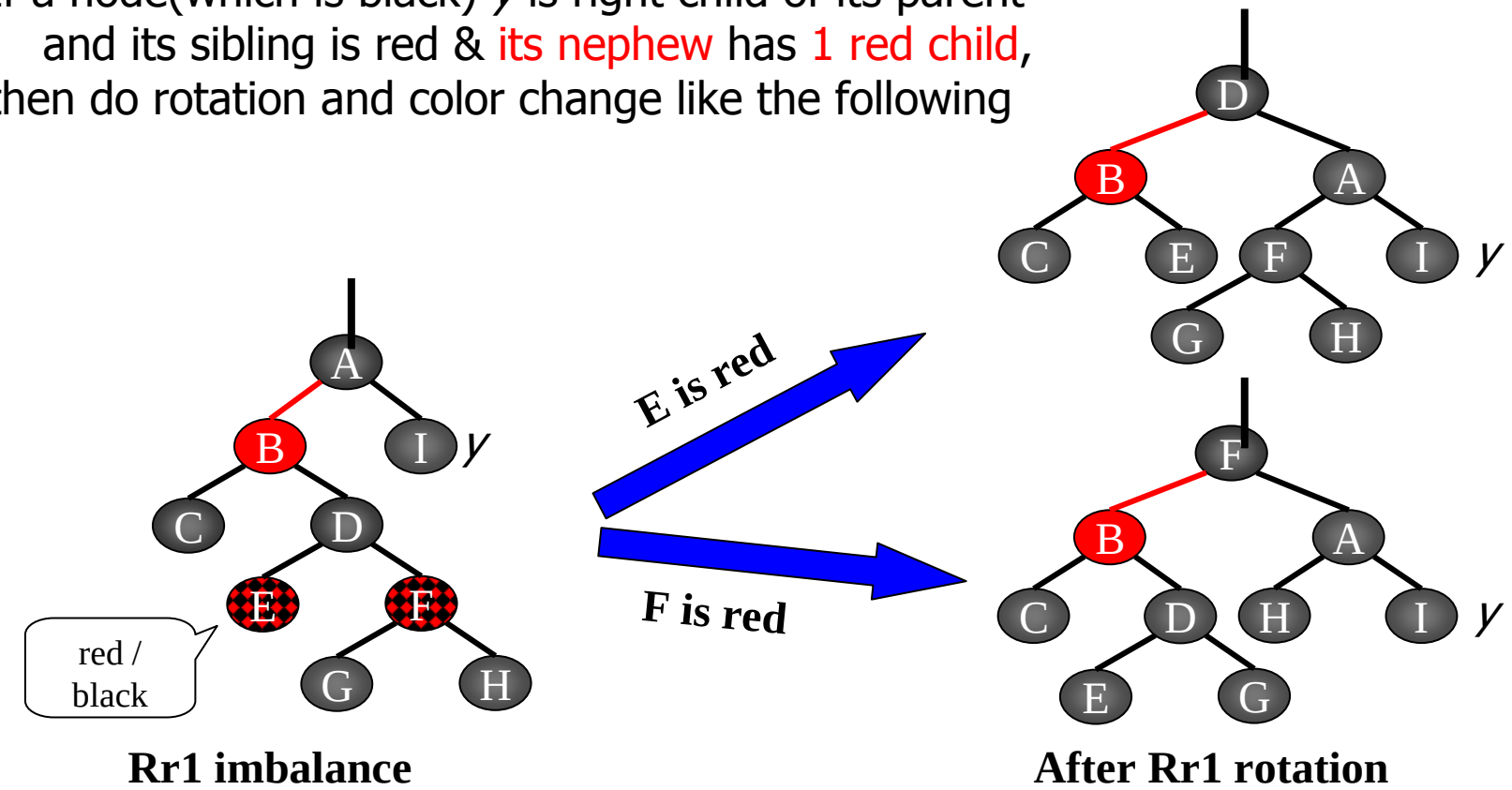
If a node(which is black) *y* is right child of its parent
and its sibling is red & its nephew has 0 red child,
then do rotation and color change like the following



Rr0 imbalance **After Rr0 rotation**

Fixing Rr1 Imbalance

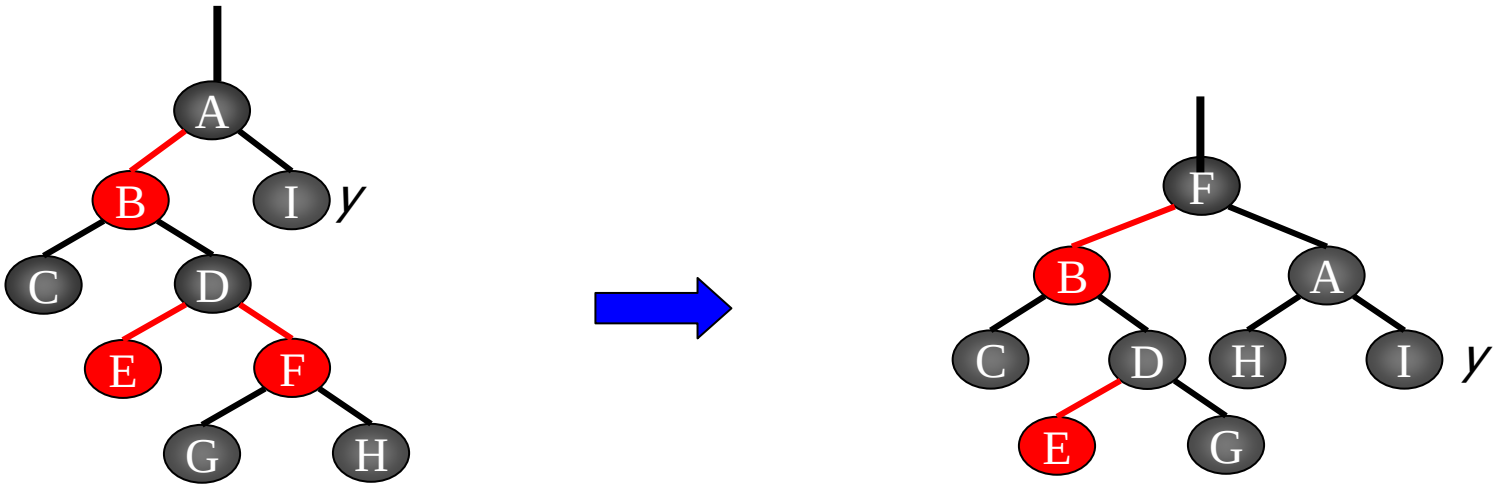
If a node(which is black) *y* is right child of its parent and its sibling is red & its nephew has 1 red child, then do rotation and color change like the following



Rr1 imbalance **After Rr1 rotation**

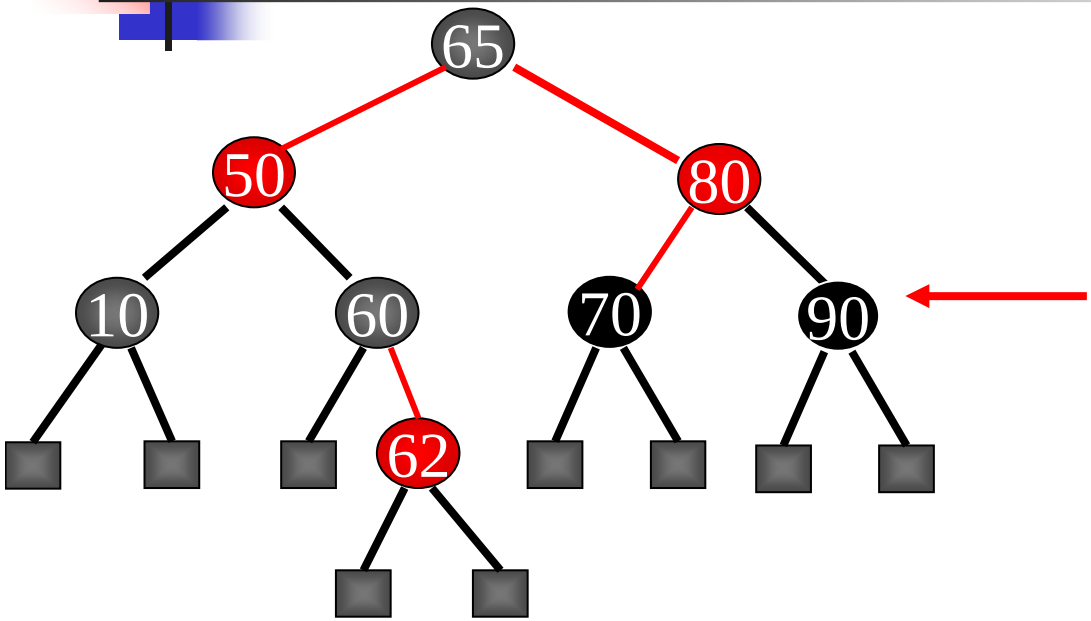
Fixing Rr2 Imbalance

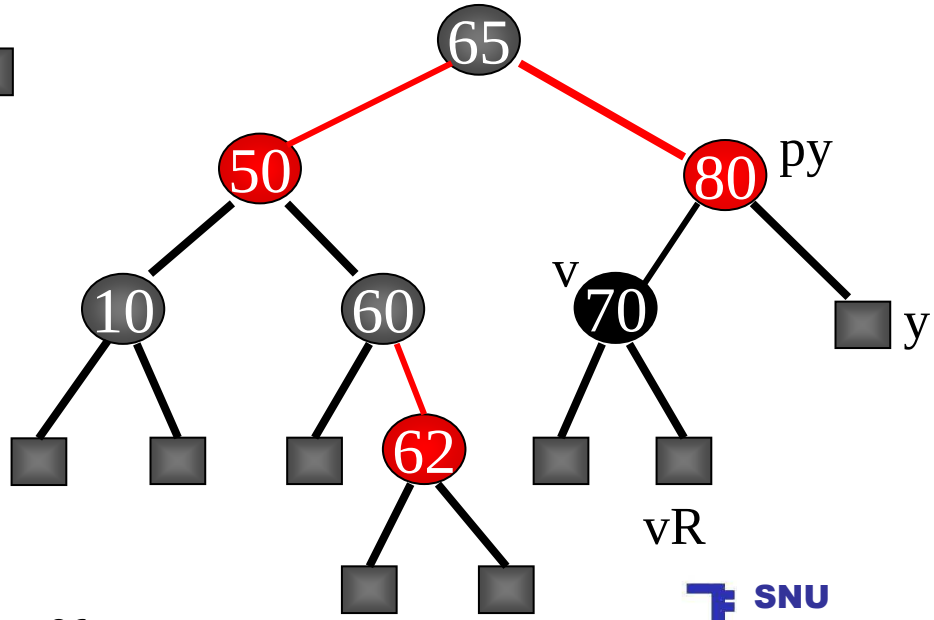
If a node(which is black) *y* is right child of its parent
and its sibling is red & its nephew has 2 red children,
then do rotation and color change like the following



Rr2 imbalance **After Rr2 rotation**

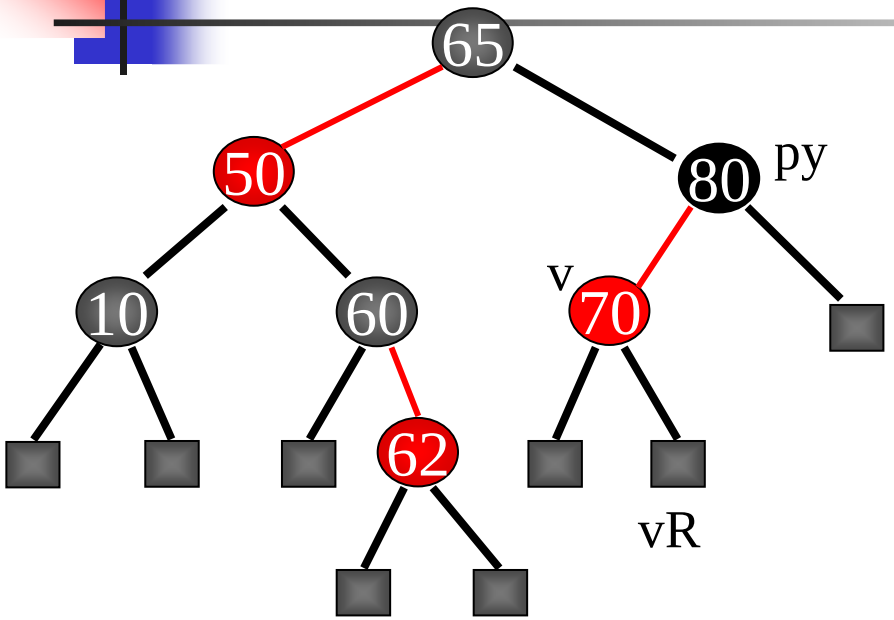
Deletion Example (1)



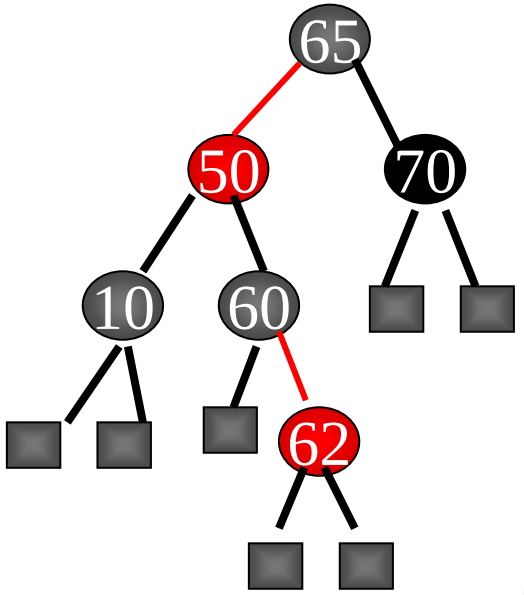


- (a) 90 deleted
- Not root & black
- Imbalance Rb0

Deletion Example (2)

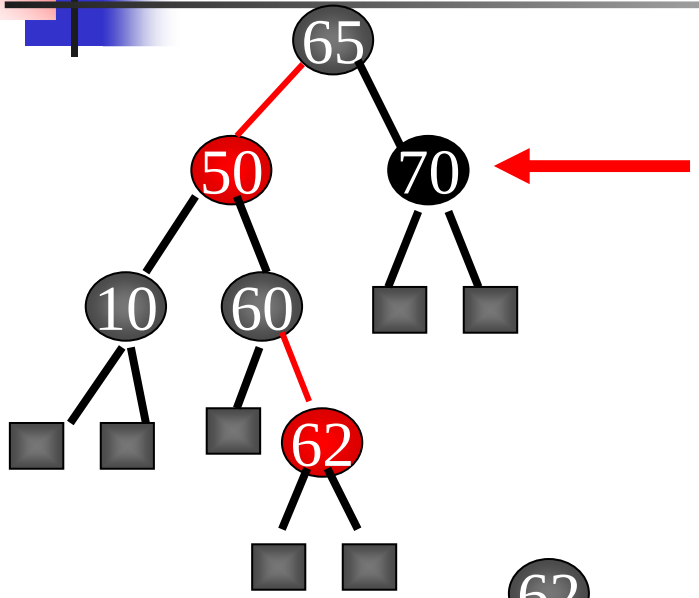


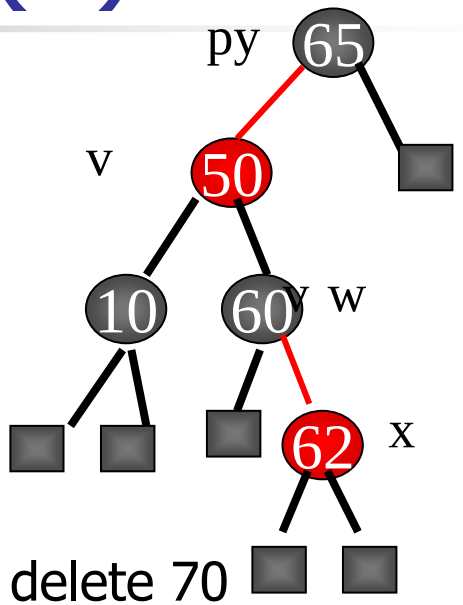
- (b) Rb0 color change
- py was red before delete
- Rb0 color change of 70 & 80
- we are done



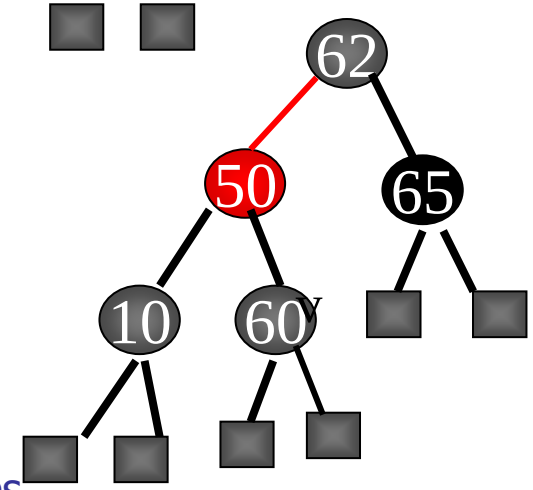
- (C) delete 80
- Black node "80" was deleted
- So tree remains balanced

Deletion Example (3)

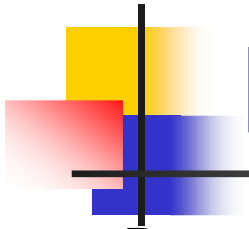




- (d) delete 70
- Nonroot black node was deleted
- Tree is imbalance Rr1(ii)



- (e) after Rr1(ii) Rotation
- This tree is now balanced!



Rotation Taxonomy in RBT

- Rotation types due to Insertion
  - L family
    - LLr type LRr type LLb type LRb type
  - R family
    - RRr type RLr type RRb type RLb type
- Rotation types due to Deletion
  - Rb family
    - Rb0 Rb1(i) Rb1(ii) Rb2
  - Rr family
    - Rr0 Rr1(i) Rr1(ii) Rr2
  - Lb family
    - Lb0 Lb1(i) Lb1(ii) Lb2
  - Lr family
    - Lr0 Lr1(i) Lr1(ii) Lr2

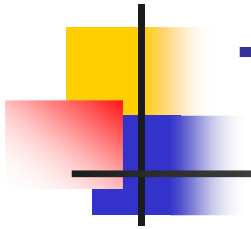
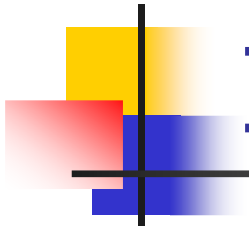


Table of Contents

- AVL Tree
- RED-BLACK TREES
  - Definition
  - Searching a Red-Black Tree
  - Inserting into a Red-Black Tree
  - Deletion from a Red-Black Tree
  - *Implementation*
- Splay Tree
- B Tree



Implementation

- Considerations
  - Insertion / Deletion require backward movement
  - If use red-black-tree nodes ➔Backward movement is easy
    else Backward movement is complex //use stack instance of color fields..etc
- Complexity
  - For an n-element red-black tree
    - parent-pointer scheme runs slightly faster than tack scheme
  - Color change : O(log n) // propagate back toward the root
  - Rotation : O(1)
  - Each color change or ratation : ⊖(1)
  - Total insert/delete O(log n)

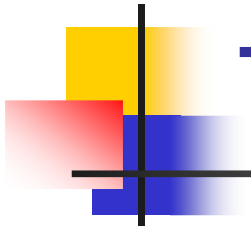
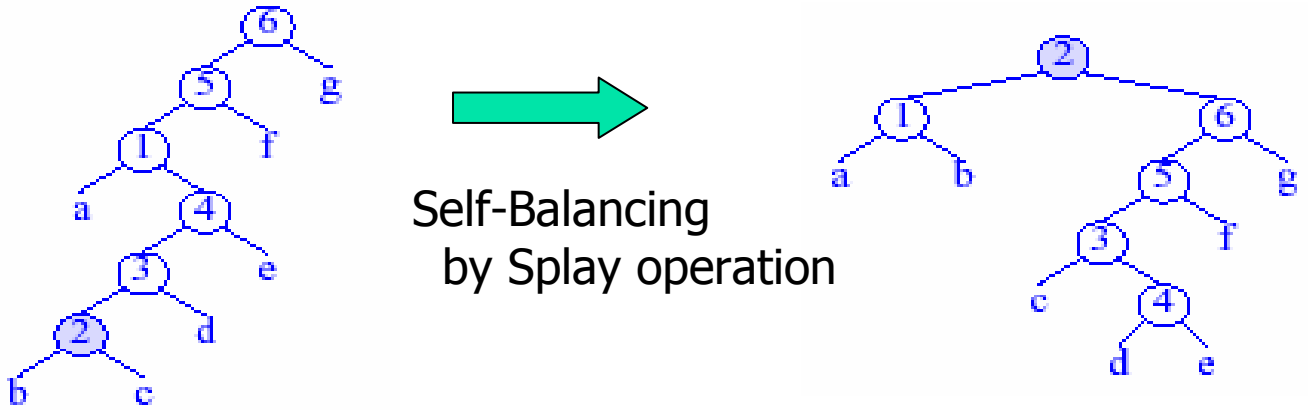


Table of Contents

- AVL TREES
- RED-BLACK TREES
- *SPLAY TREES*
- B-TREES

Splay Tree

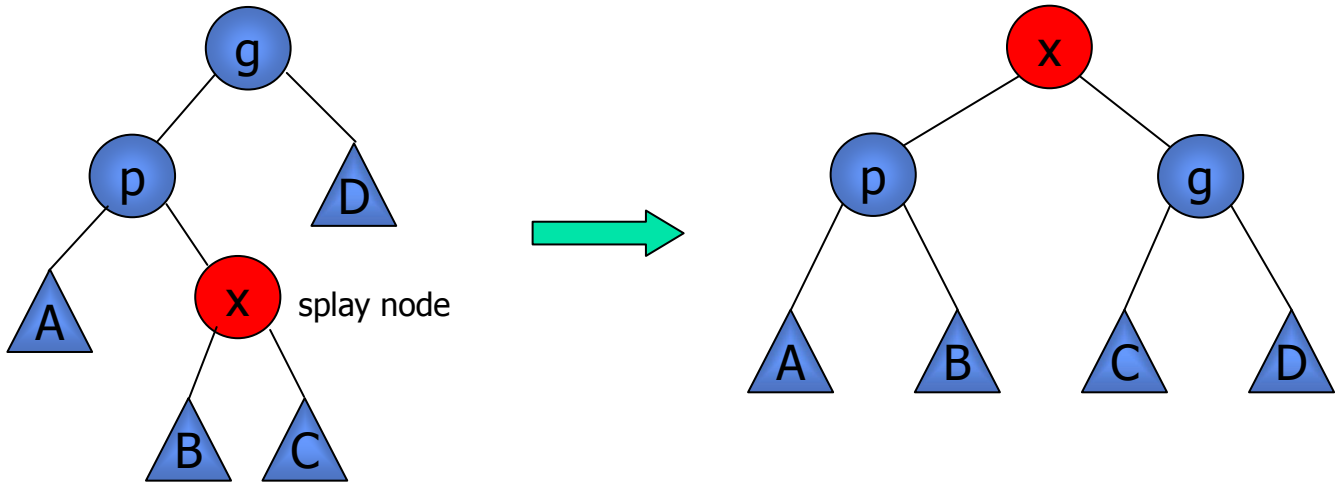
- Splay tree is a binary search tree whose nodes are rearranged by splay operation whenever search, insertion, or deletion occurs
- The recently accessed node is moved to the top



- Properties of splay tree
  - Recently accessed elements are quick to access again
    - Basic operations run in O(log *n*) amortized time
  - It is simpler to implement splay trees than red-black trees or AVL trees
  - Splay trees don't need to store any extra data in nodes

The Splay Operation

- We call recently accessed(searched, inserted, or deleted) node as *splay node*
  - Splay operation is performed on splay node to **move it to the root**
  - We can perform successive accesses faster because recently accessed node is moved to the top of the tree



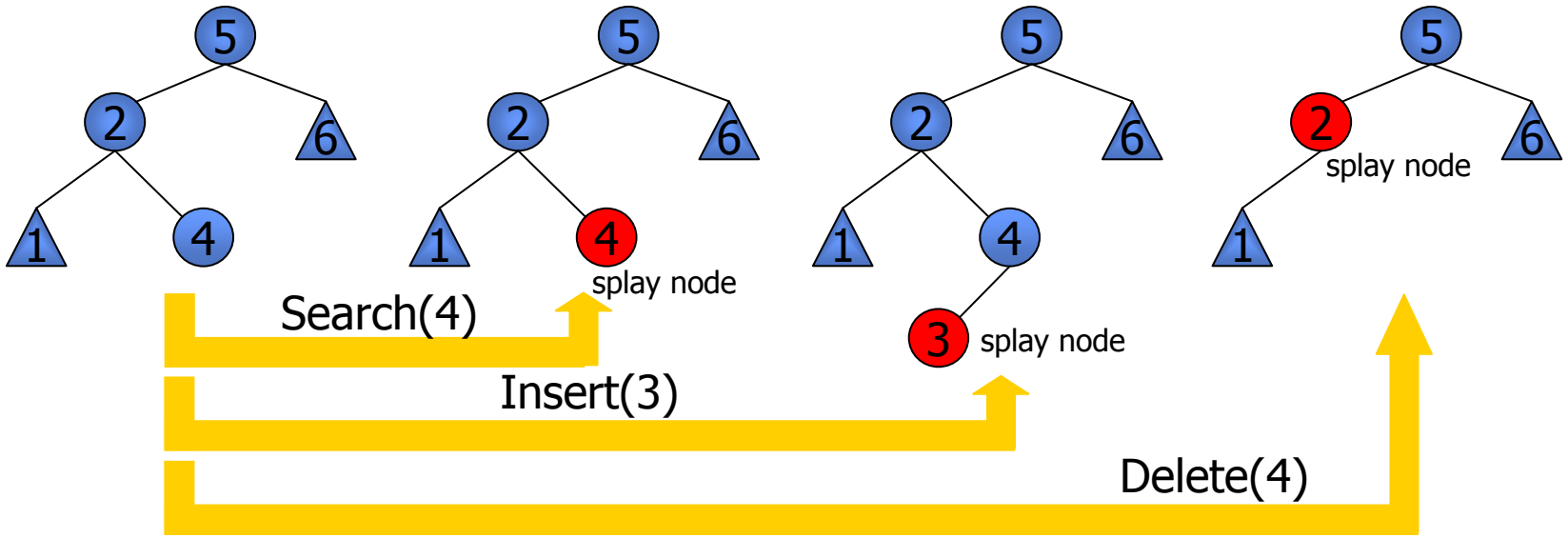
- Splay operation comprises sequence of the following splay steps.

  If (Splay node = root) then sequence of steps is empty

  Else splay step moves the splay node either 1 level or 2 levels up the tree

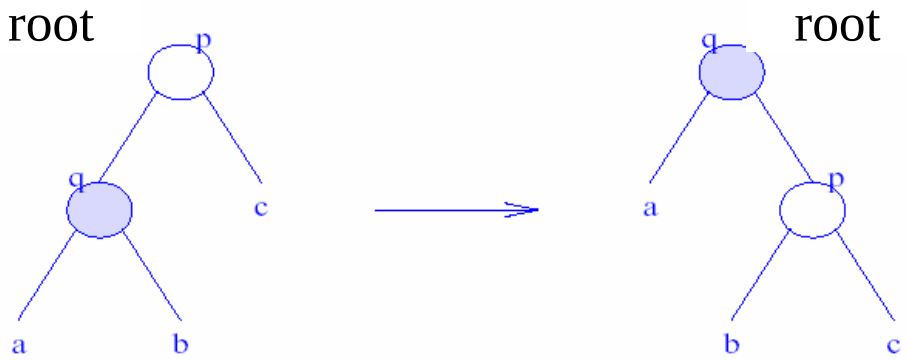
Splay Node

- Search(x) makes the node x as a splay node
- Insert(x) makes the node x as a splay node
- Delete(x) makes the parent node of x as a splay node



One Level Splay Step

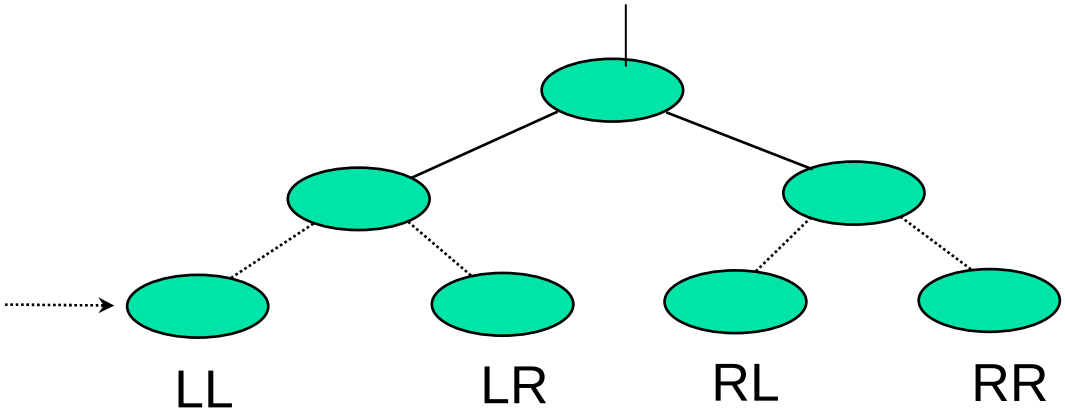
- When the level of splay node = 2 (Only)
  - L splay step : splay node is **L**eft child of its parent
  - R splay step : splay node is **R**ight child of its parent
- L splay step
  - If splay node q is the left child of its parent, then do rotation like the following
  - Notice that following the splay step the splay node becomes the root of binary search tree



a, b, and c are subtrees
shaded node is the splay node
Figure 16.20 Type L splay step

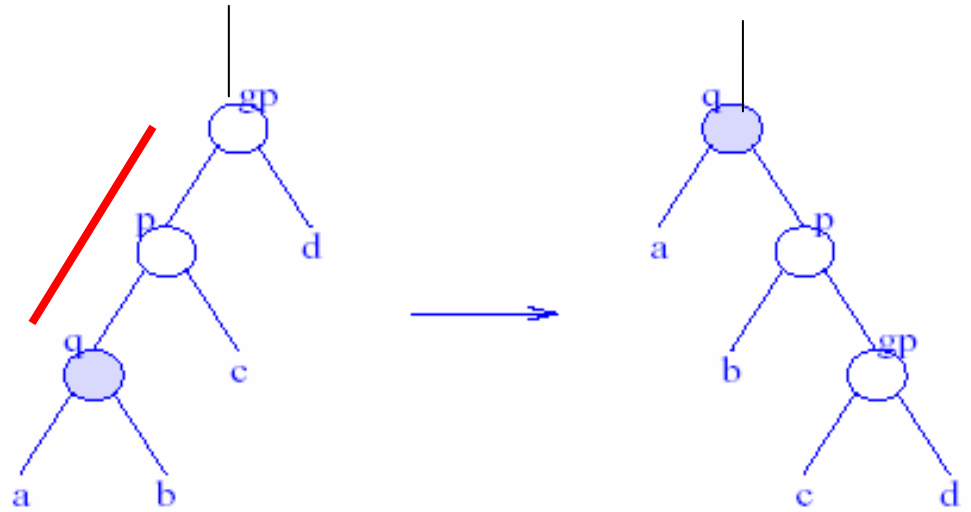
Two Level Splay Step

- When the level of splay node > 2
- Types
  - LL : p is Left child of gp, q is Left child of p
  - LR : p is Left child of gp, q is Right child of p
  - RR : p is Right child of gp, q is Right child of p
  - RL : p is Right child of gp, q is Left child of p



LL Splay Step

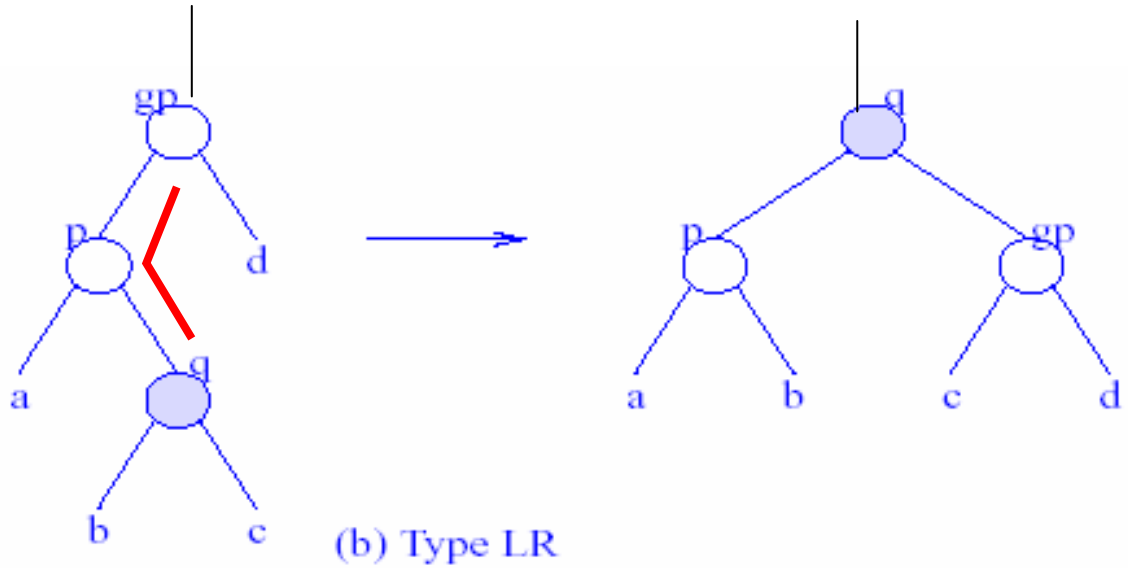
- If splay node q is the left child of its parent
  & its parent is the left child of its grandparent,
  then do rotation like the following
- The splay node is moved 2 level up



(a) Type LL

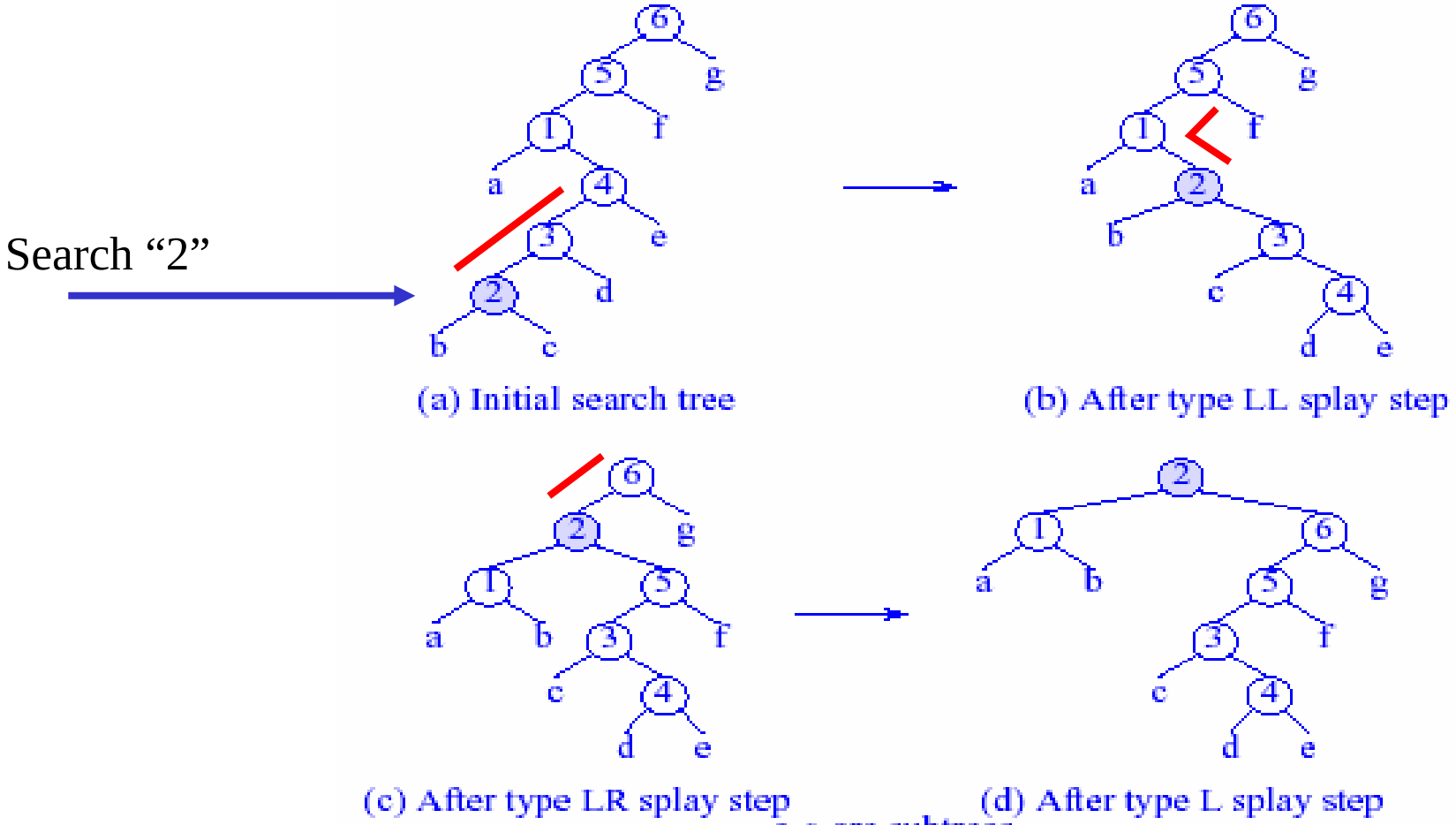
LR Splay Step

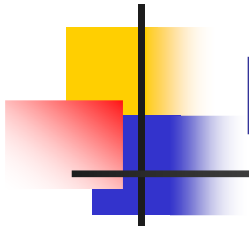
- If splay node q is the right child of its parent
  & its parent is the left child of its grandparent,
  then do rotation like the following
- The splay node is moved 2 level up



(b) Type LR

Sample Splay Operation





Rotation Taxonomy of Splay Tree

- 1 level splay step
  - L type
  - R type
- 2 level splay step
  - LL type
  - LR type
  - RL type
  - RR type

Concept of Amortized

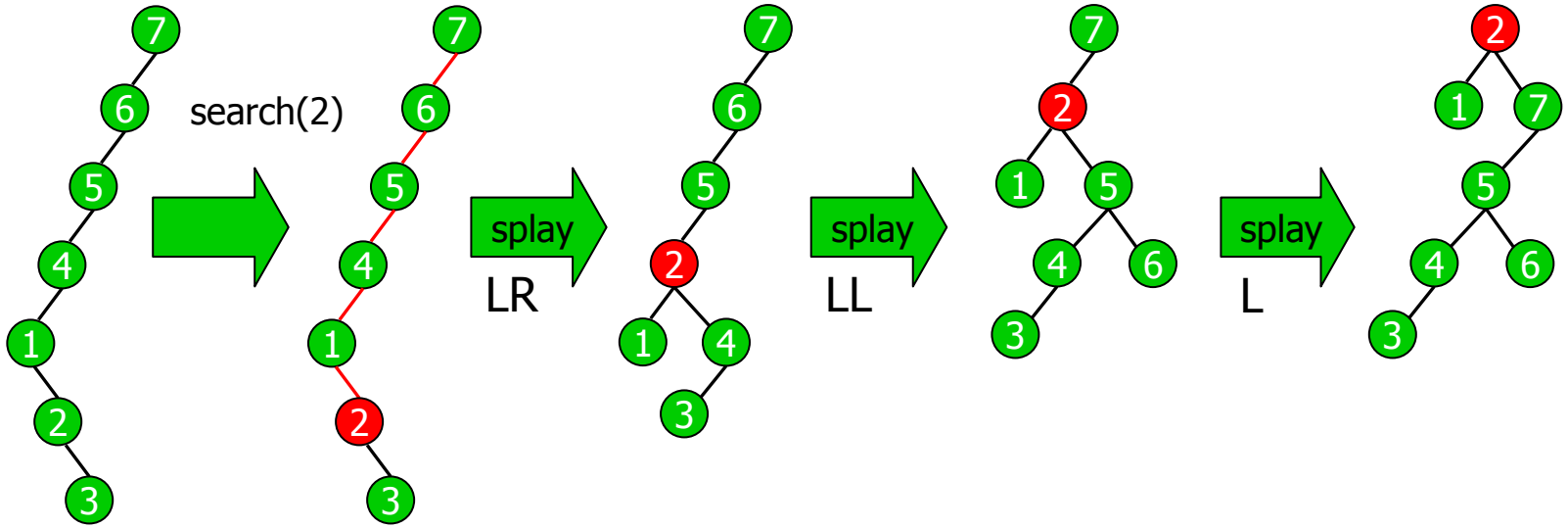
- Rule: Spend less than 100$ per month
  - Normal spending – Spend less than 100$ per month
  - Amortized spending – Spend less than (100 * 12)$ per year
- Remember array expansion
  - Regular complexity
    - Double the size (initialize) -- O(n)
    - Copy the old array to the new array – O(n)
  - Amortized complexity
    - Doubling will happen after n insertions!
    - One insertion is responsible for one slot expansion ➔ O(1)

Amortized Complexity (1)

- In an *amortized analysis*, the time required to perform a sequence of data-structure operations is averaged over all the operations performed
- Amortized analysis differs from average-case analysis
  - Amortized analysis guarantees the *average performance of each operation in the worst case*
- $$\sum_{i=1}^{n} amortized(i) \geq \sum_{i=1}^{n} actual(i)$$
- Theorem 16.1
  - The amortized complexity of a get, put or remove operation performed on a splay tree with n element is O(log n)
- Actual Complexity of any sequence of g get, p put and r remove operations
  - O((g+p+r)log n)

Amortized Complexity (2)

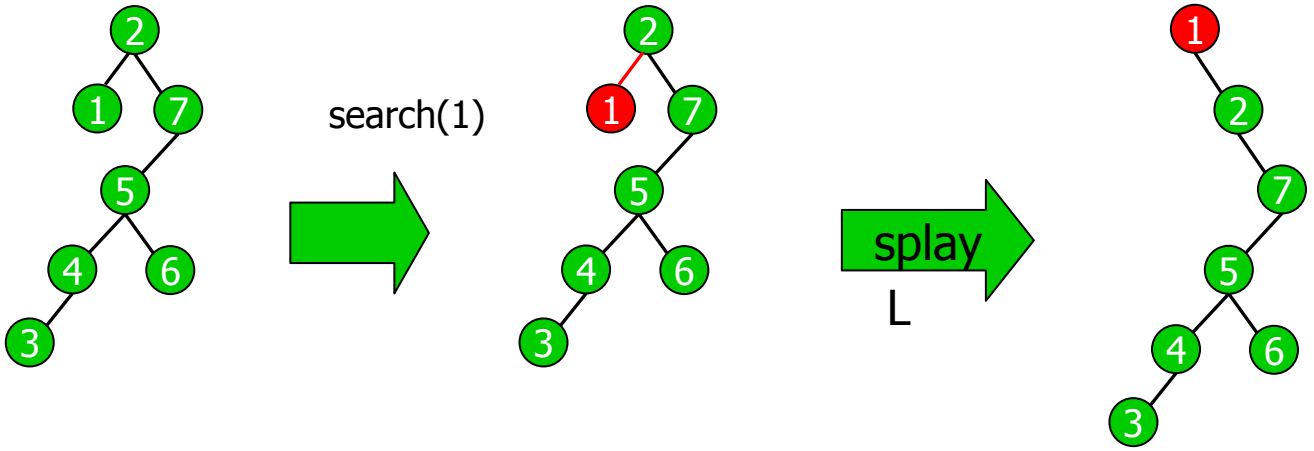
- Example (1)



T_1 = (search time)+(splay time)= 6 comparisons + 5 rotations

Amortized Complexity (3)

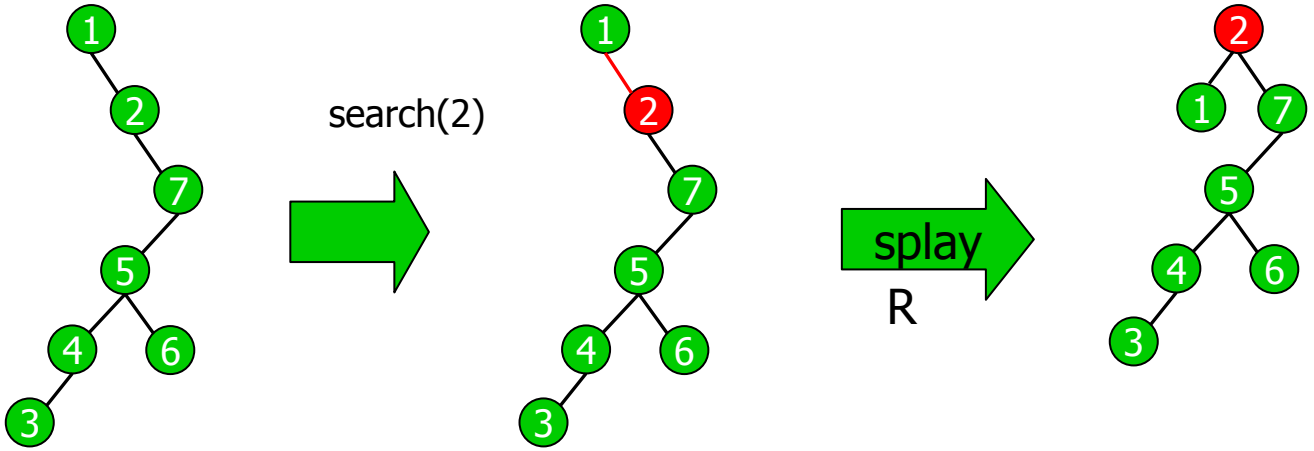
- Example (2)



T_2 = (search time) + (splay time) = 2 comparisons + 1 rotation

Amortized Complexity (4)

- Example (3)



T_3 = (search time) + (splay time) = 2 comparisons + 1 rotations

Amortized Complexity (5)

- Example (4)
  - In the previous example, total time taken is 10 comparisons + 7 rotations
  - If there were no splay operation, total time taken would be 18 comparisons
  - Generally, it is known that $(t_1+t_2+\ldots+t_k) / k \leq 3*\log_2 n$, where n is the number of nodes if k is large enough

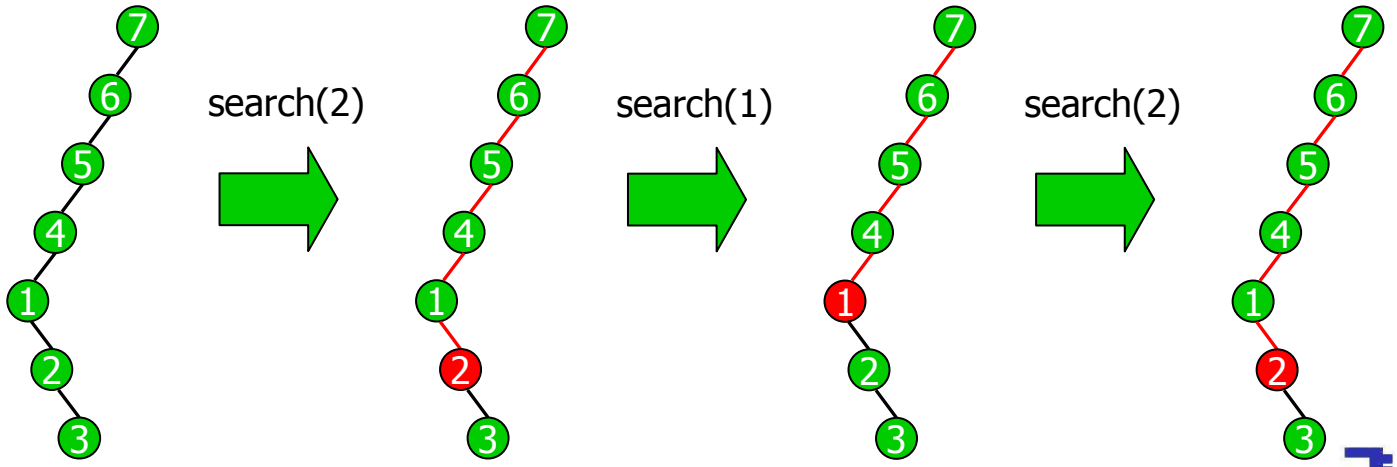
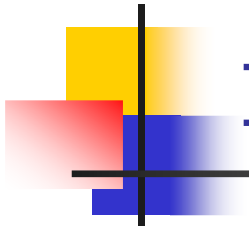


Table of Contents

- AVL TREES
- RED-BLACK TREES
- SPLAY TREES

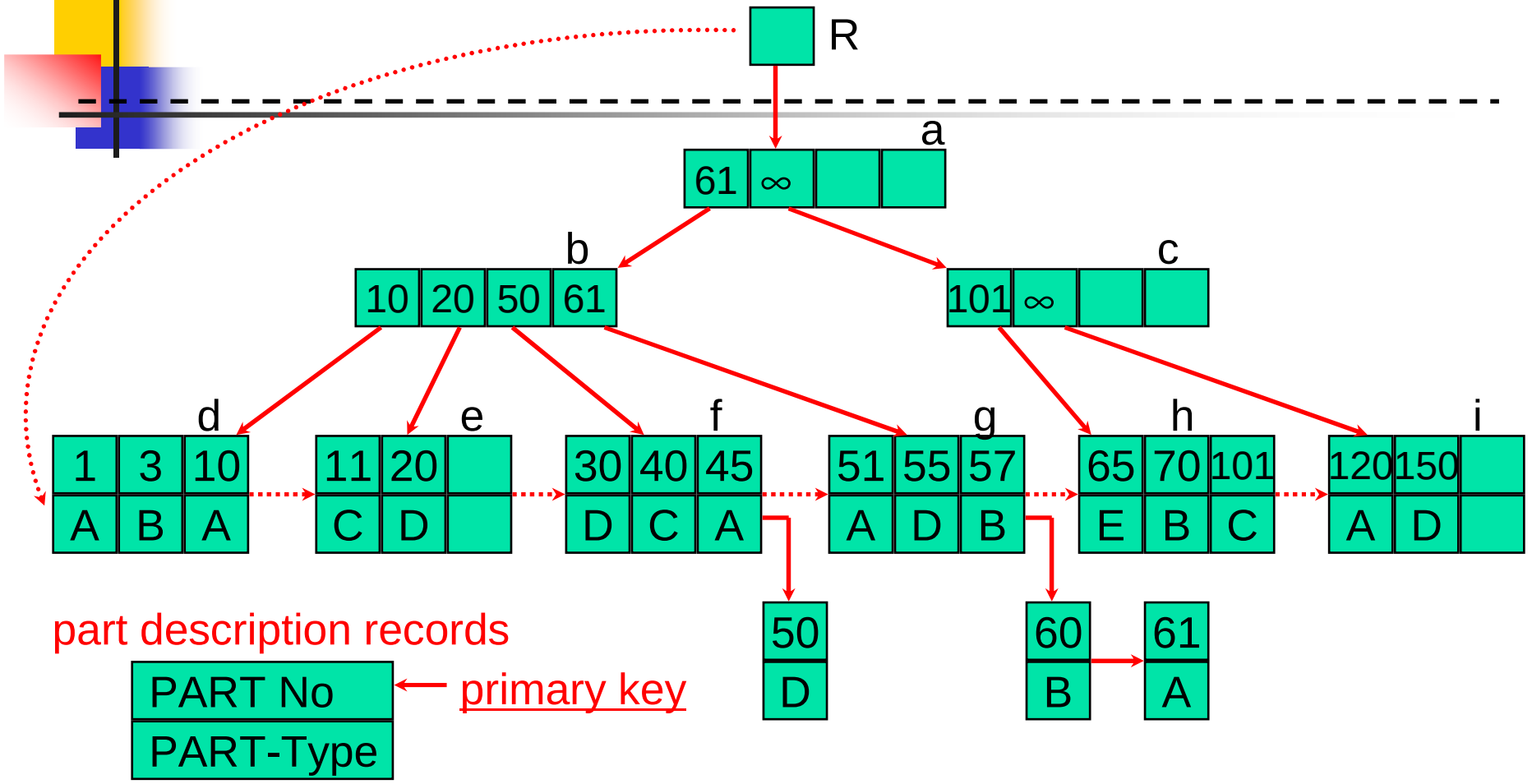
- B-TREES
  - *Indexed Sequential Files (ISAM)*
  - m-WAY Search Trees
  - B-Trees of Order *m*
  - Height of a B-Tree
  - Searching a B-Tree
  - Inserting into a B-Tree
  - Deletion from a B-Tree



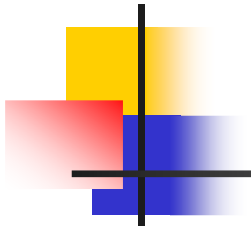
Indexed Sequential Access Method (ISAM)

- Small dictionary may reside in internal memory
- Large dictionary must reside on a disk
  - A disk consists of many blocks
  - Elements (records) are packed into a block in ascending order
- ISAM file (= Indexed Sequential file)
  - disk-based file structure for large dictionary
  - Provide good sequential and random access
- Primary Concern: reducing the number of disk IO s during search

Overview : ISAM File



Example : Indexed sequential structure (when using overflow chain)



File Structure Evolution

- Sequential file: records can be accessed sequentially
  - not good for access, insert, delete records in random order
- Indexed-sequential file = Indexed Sequential Access Method (ISAM)
  - Sequential file + Index
- B+ tree file
  - Indexed-sequential file + Balance

But here we study “B tree” data structure
--- m-Way search tree is similar to ISAM file

Table of Contents

- AVL TREES
- RED-BLACK TREES
- SPLAY TREES

- B-TREES
  - Indexed Sequential Files (ISAM)
  - m-WAY Search Trees
  - B-Trees of Order *m*
  - Height of a B-Tree
  - Searching a B-Tree
  - Inserting into a B-Tree
  - Deletion from a B-Tree

m-Way Search Tree

- Binary Search Tree can be generalized to m-Way search tree
- White box is an internal node while solid square is external node
- Each internal node can have up to six keys and seven pointers
- A certain input sequence would build the following example

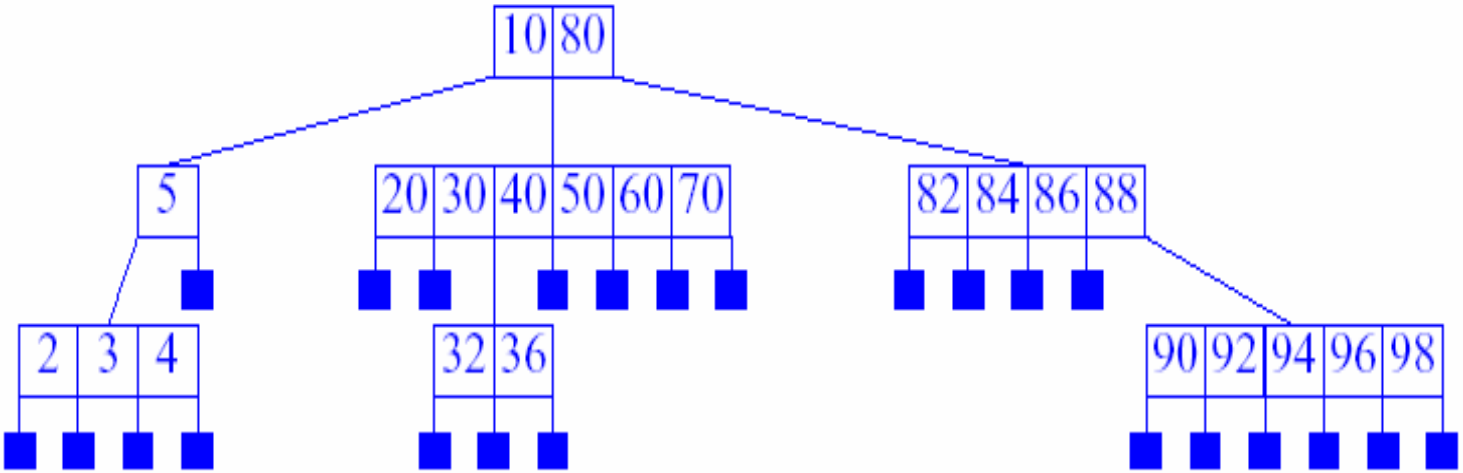
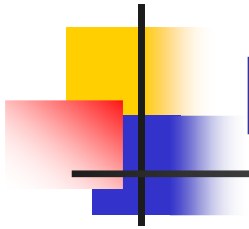


Figure 16.23 A seven-way search tree



Properties of m-WAY Search Tree

- m-Way search tree has the following properties
  - In the corresponding extended search tree, each internal node has up to p+1 children and between 1 and p elements.
  - Every node with p elements has exactly p + 1 children
    - Let k_1, ...,k_p be the keys of these ordered elements ($k_1 < k_2 < \ldots < k_p$)
    - Let c_0, $c_1 \ldots$, c_p be the p+1 children of the node.

- Key ranges
  - The elements in the subtree with root c_o have keys smaller than k_1
  - The elements in the subtree with root c_p have keys larger than k_p
  - The elements in the subtree with root c_i have keys larger than k_i but smaller than k_{i+1}, $1 \leq i \leq p$

Searching an m-Way Search Tree

- Search the element with key 31
  - 10< 31 <80 : Move to the middle subtree
  - k_2< 31 <k_3 : Move to the third subtree
  - 31< k_1 : Move to the first subtree, Fall off the tree, No element

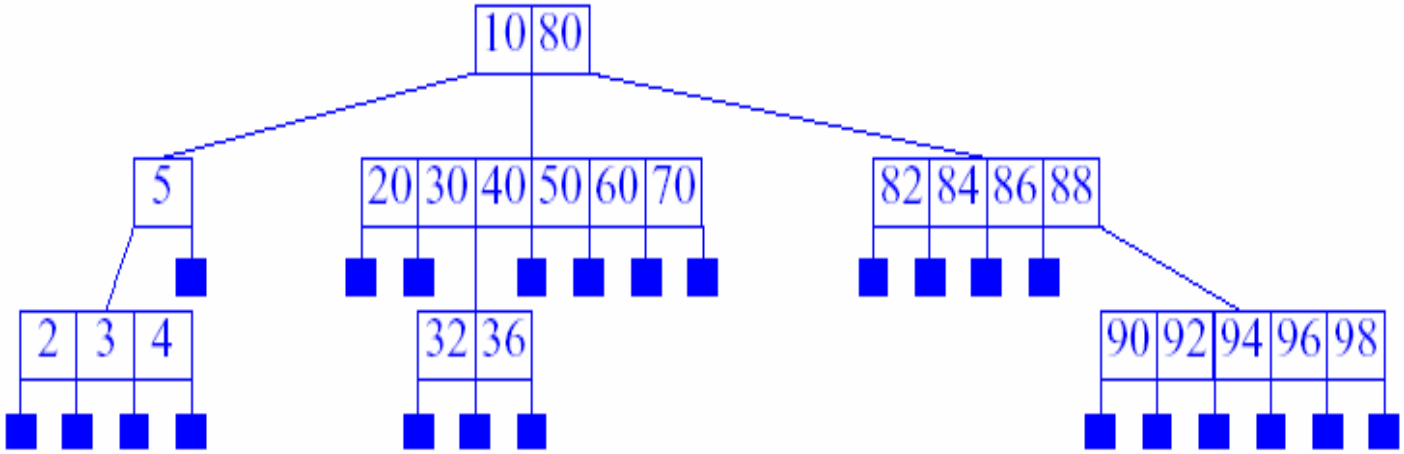


Figure 16.23 A seven-way search tree

Inserting into an m-Way Search Tree

- Insert the new key 31
  (a) Search for 31 & Fall off the tree at the node[32,26]
  (b) Insert at the first element in the node

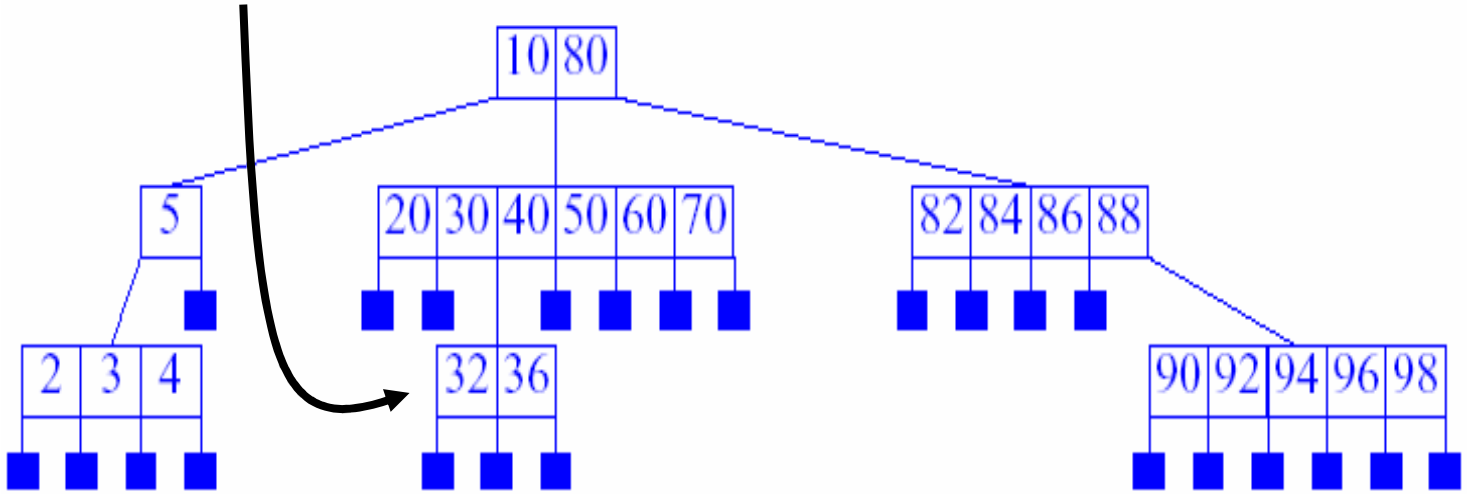


Figure 16.23 A seven-way search tree

Inserting into an m-Way Search Tree

- Insert the new key 65:
  (a) Search for 65 & Fall off the tree at six subtree of node [20,30,40,50,60,70]
  (b) New node obtained & New node becomes the sixth child of [20,30,40,50,60,70]

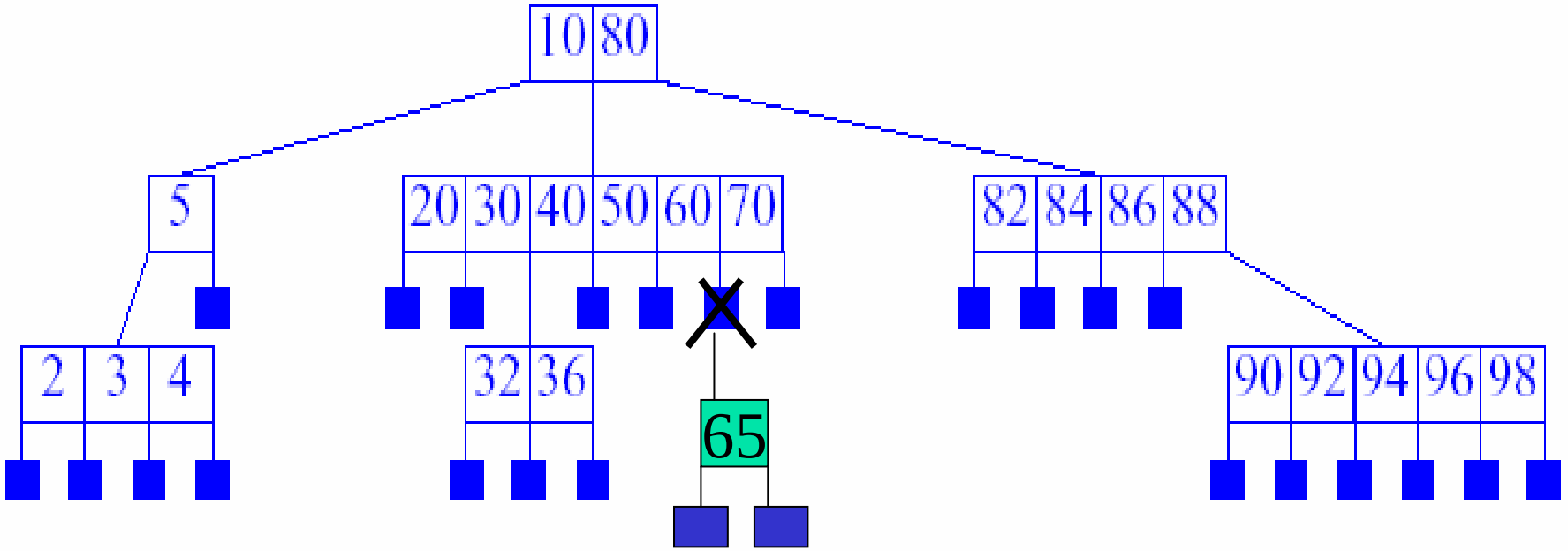


Figure 16.23 A seven-way search tree

Deleting from an m-Way Search Tree

- Delete the key 20
  - Search for 20, k_1=20 & C_0=C_1=0, and Simply Delete 20

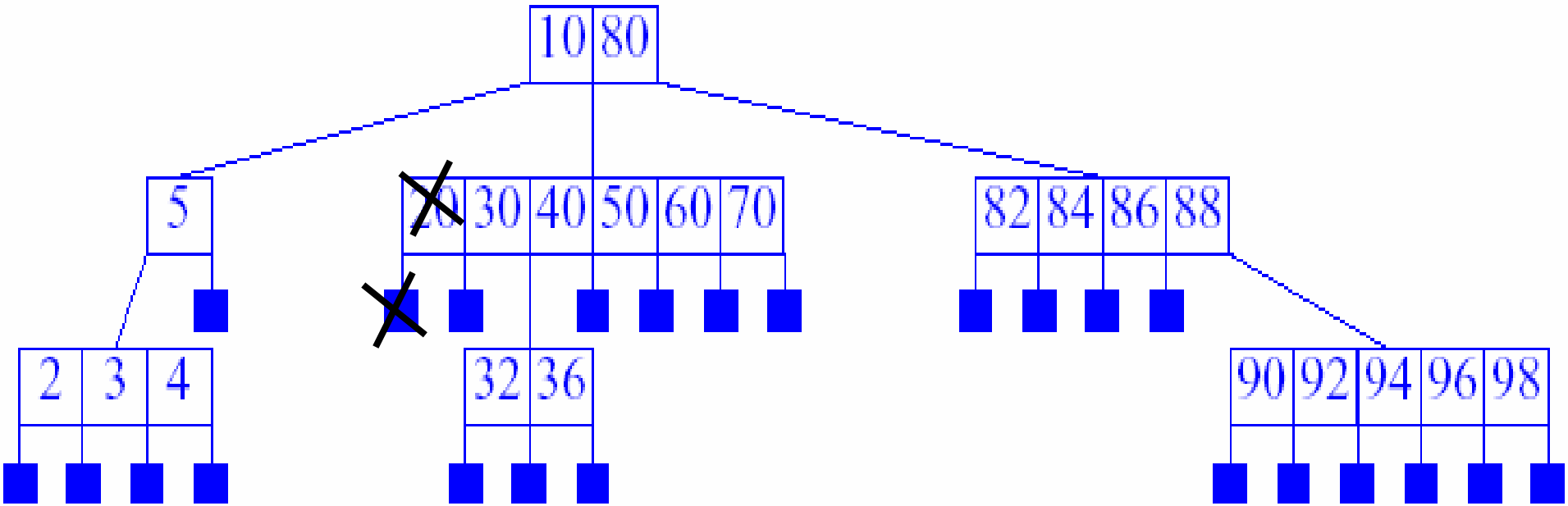


Figure 16.23 A seven-way search tree

Deleting from an m-Way Search Tree

- Delete the key 84
  - Search for 84, k_2=84 & C_1=C_2=0, and Simply Delete 84

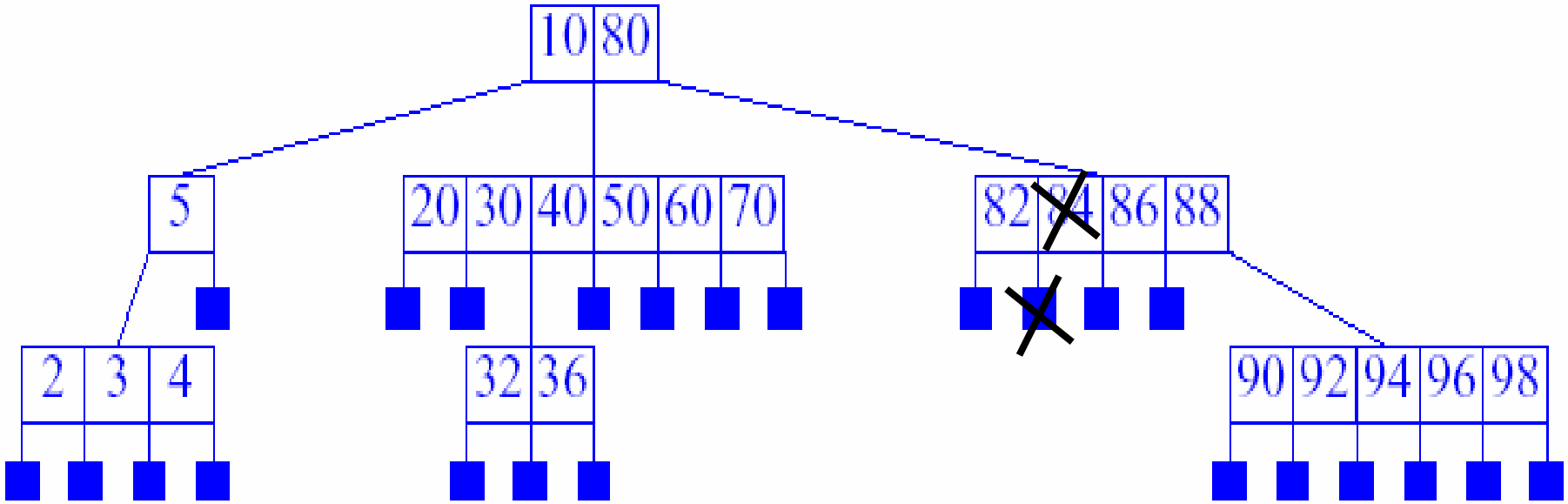


Figure 16.23 A seven-way search tree

Deleting from an m-Way Search Tree

- Delete the key 5 :
  (a) Only one key in the node ➔ Need to replace
  (b) From C_0, move up the element with largest key → move the key 4 to the key 5's position

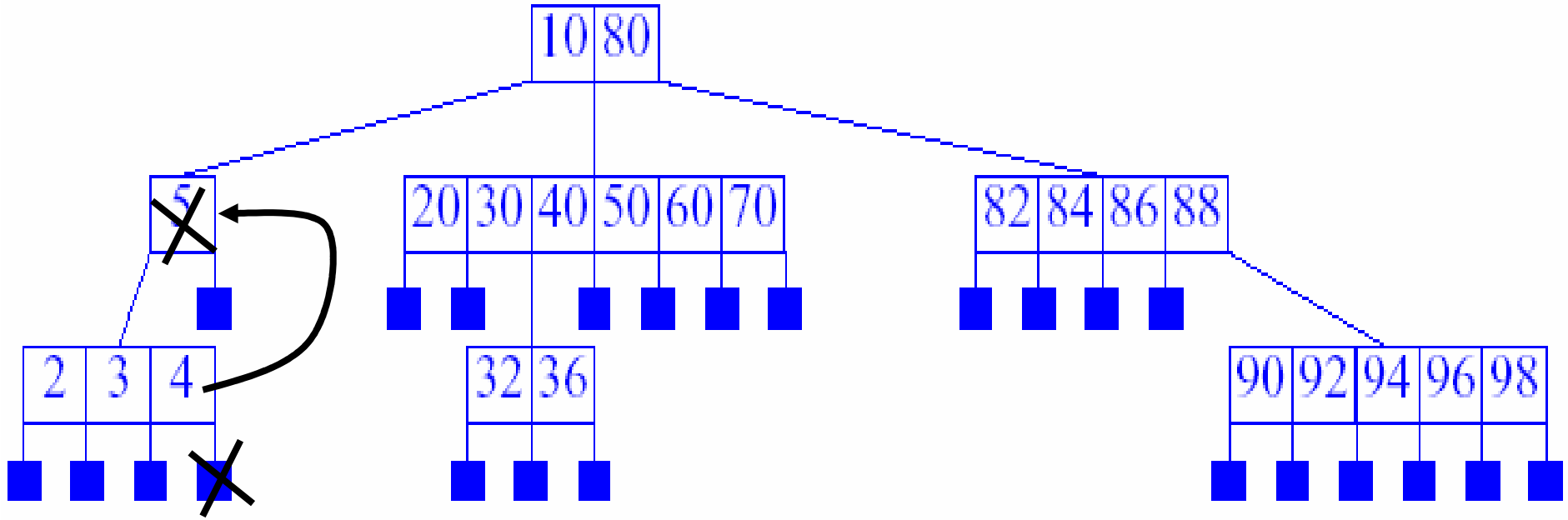


Figure 16.23 A seven-way search tree

Deleting from an m-Way Search Tree

- Delete the key 10

Replace this element with either the largest element in C_0 or smallest element in C_1

So, element with key 5 is moved to top & element with key 4 is moved up to the key 5's position

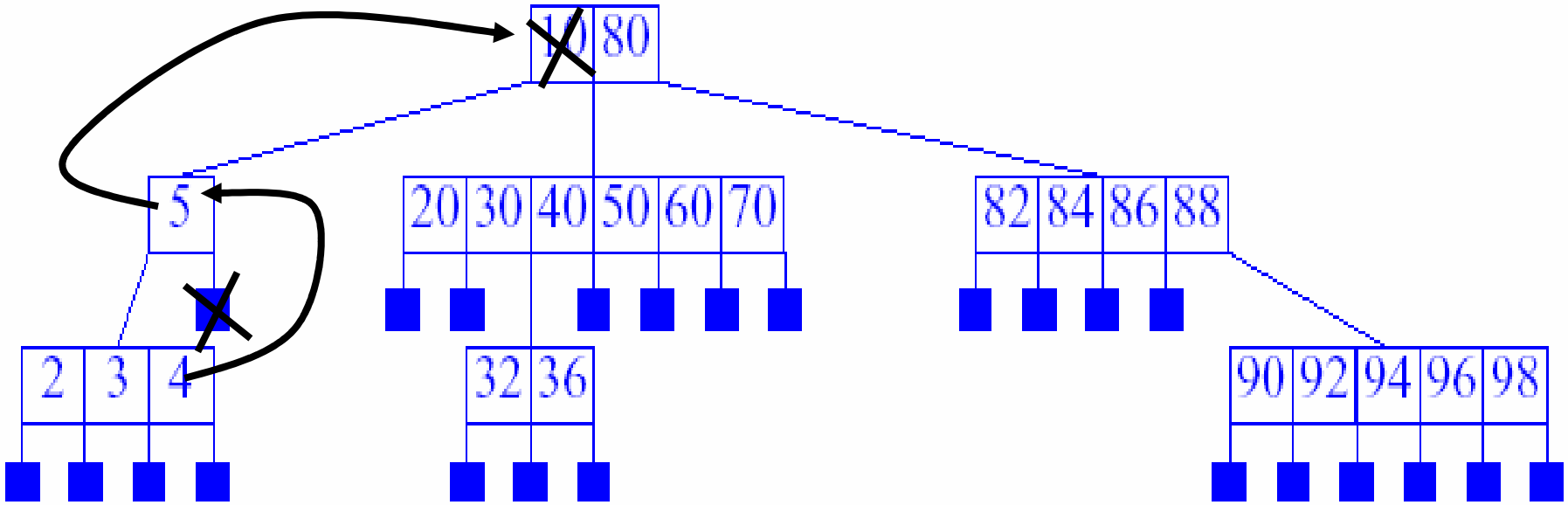


Figure 16.23 A seven-way search tree

Height of an m-Way Search Tree

- h : Height, n : number of elements, m : m-way
- The number of elements: $h \le n \le m^h - 1$
- The number of nodes : $\sum_{i=0}^{h-1} m^i = (m^h-1)/(m-1)$ nodes
- The range of height: $\log_m(n+1) \le h \le n$
- The number of disk accesses : O(h)
- We want to ensure that the height h is close to $\log_m(n+1)$ ➔ this is accomplished by B-tree!

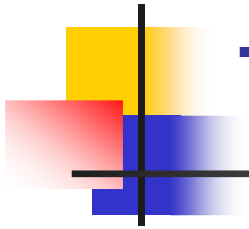


Table of Contents

- B-TREES
  - Indexed Sequential Access Method(ISAM)
  - *m*-WAY Search Trees
  - *B-Trees of Order m*
  - Height of a B-Tree
  - Searching a B-Tree
  - Inserting into a B-Tree
  - Deletion from a B-Tree

Definition: B tree of Order m

- B-tree is a m-way search tree satisfying the following properties
  1. The root has at least two children
  2. All internal nodes other than the root at least $\lceil m/2 \rceil$ children (pointers to the children nodes)
  3. All external nodes are at the same level

- Internal node has several pairs of a key and a pointer to a disk block

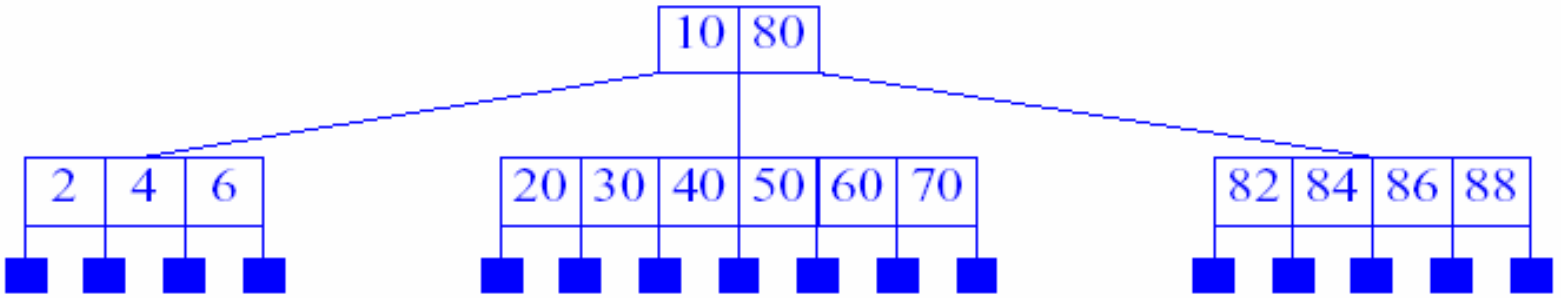


Figure 16.24 A B-tree of order 7

B-Trees of Order m

- B-tree of order 2: Fully binary tree
- B-tree of order 3 (= 2- 3 tree): 2 or 3 children
- B-tree of order 4 (= 2- 3- 4 tree): 2 or 3 or 4 children

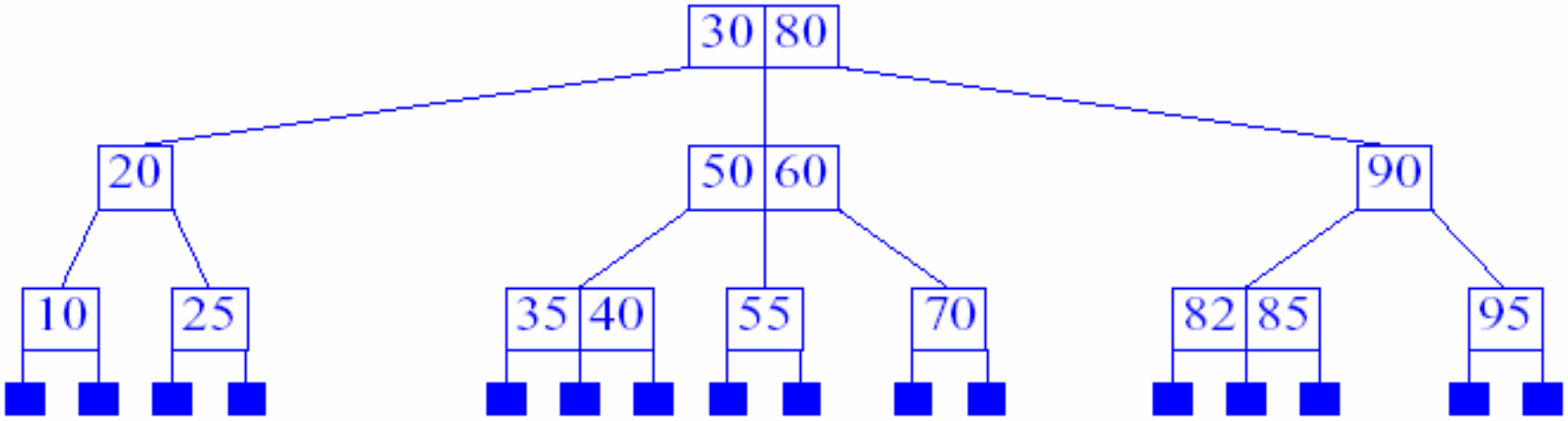


Figure 16.25 A 2-3 tree or B-tree of order 3

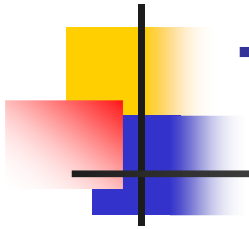


Table of Contents

- B-TREES
  - Indexed Sequential Access Method(ISAM)
  - *m*-WAY Search Trees
  - B-Trees of Order *m*
  - *Height of a B-Tree*
  - Searching a B-Tree
  - Inserting into a B-Tree
  - Deletion from a B-Tree

Height of a B-Tree of Order m

- Remember: All internal nodes other than the root at least $\lceil m/2 \rceil$ children (pointers to the children nodes)

- Lemma 16.3

  Let T be a B-tree of order m
  Let h be the height of T

  Let $d = \lceil m/2 \rceil$ be the degree of T
  Let n be the number of elements in T

  - (a) $2d^{h-1} \leq n \leq m^h - 1$
  - (b) $\log_m(n + 1) \leq h \leq \log_d((n+1)/2) + 1$

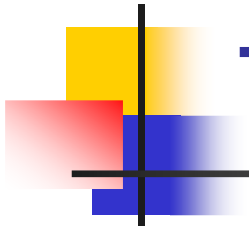


Table of Contents

- B-TREES
  - *m*-WAY Search Trees
  - B-Trees of Order *m*
  - Height of a B-Tree
  - *Searching a B-Tree*
  - Inserting into a B-Tree
  - Deletion from a B-Tree

Searching a B-Tree

- Using the same algorithm as is an m-way search tree
  - First visit the root with the given key K
  - Compare K and the keys in the root
  - Follow the corresponding pointer
  - Search the child node recursively until the leaf node
  - If arrived at the leaf node, Search the external node

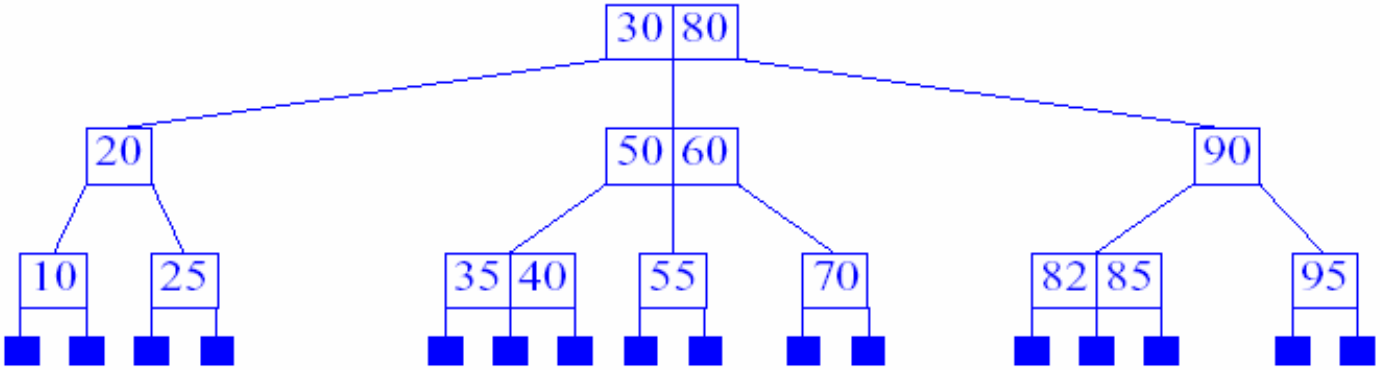


Figure 16.25 A 2-3 tree or B-tree of order 3

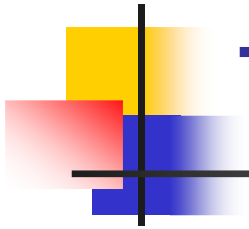
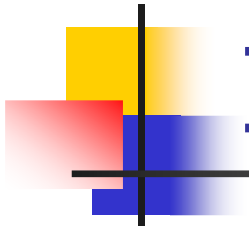


Table of Contents

- B-TREES
  - Indexed Sequential Access Method(ISAM)
  - *m*-WAY Search Trees
  - B-Trees of Order *m*
  - Height of a B-Tree
  - Searching a B-Tree
  - *Inserting into a B-Tree*
  - Deletion from a B-Tree



Inserting into a B-Tree

- First search with the key of the new element
- Found ➔ Insertion fails (if duplicates are not permitted)
- Not Found

  Insert the new element into the last encountered internal node

  If (no overflow) return ok

  Else (overflow) { split the last internal node into 2 new nodes;

  go to the 1-level up for updating the parent node (recursively)}

Notations in B-tree

- e : element c : children p : parent node
- Full node has m elements & m+1 children
  - d : degree of a node ➔ at least $\lceil m/2 \rceil$
  - e_i : element pointers c_i : children pointers

- Overfull node
  - $m, c_0, (e_1, c_1), \ldots, (e_m, c_m)$
- P : Left remainder
  - $d-1, c_0, (e_1, c_1), \ldots, (e_{d1-1}, c_{d-1})$
- Q : Right remainder
  - $m-d, c_d, (e_{d+1}, c_{d+1}), \ldots, (e_m, c_m)$
- Pair(e_d, Q) is inserted into the parent of P

Insert the key 3 in B-tree

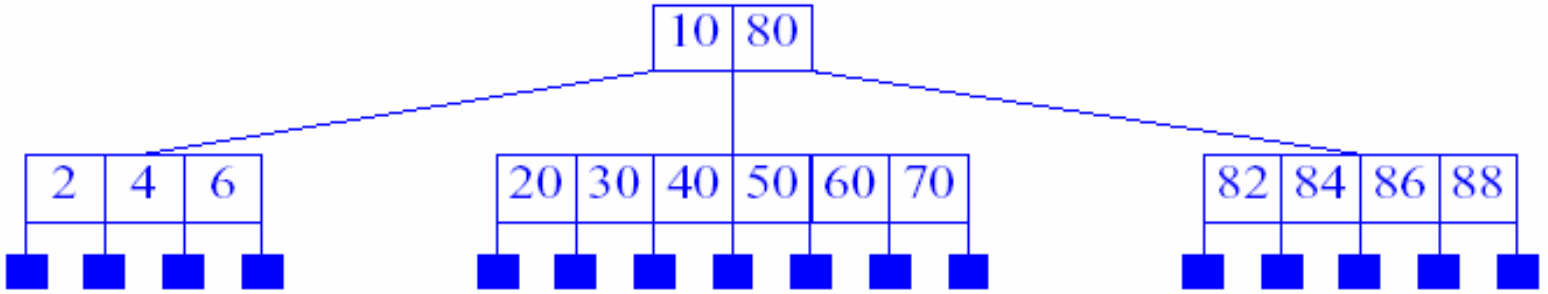


Figure 16.24 A B-tree of order 7

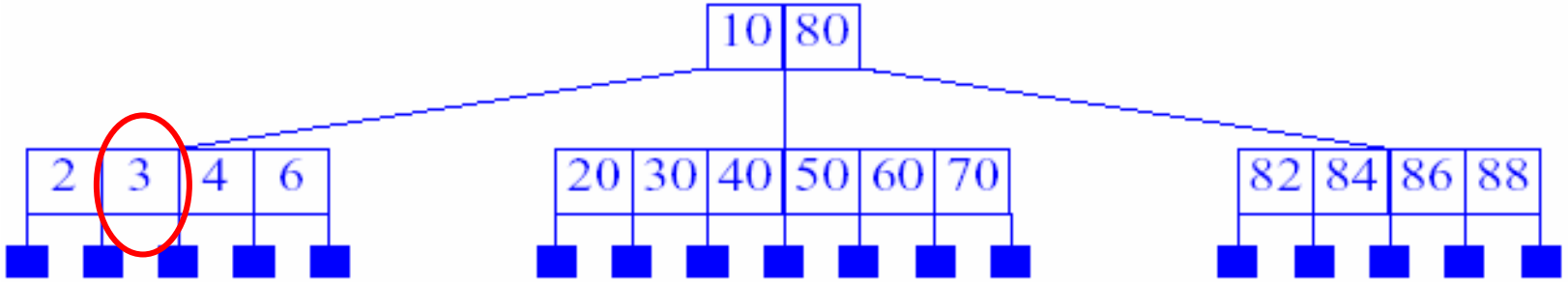


Figure 16.26 (a) Insert 3 into Figure 16.24

Insert the key 25 in B-tree

- d = 4 & the target node (“6”, 20,30,40,50,60,70)
  - P : 3, 0, (20,0), (25,0), (30,0)
  - Q : 3, 0, (50,0), (60,0), (70,0)
  - (40, Q) is inserted into parent of P

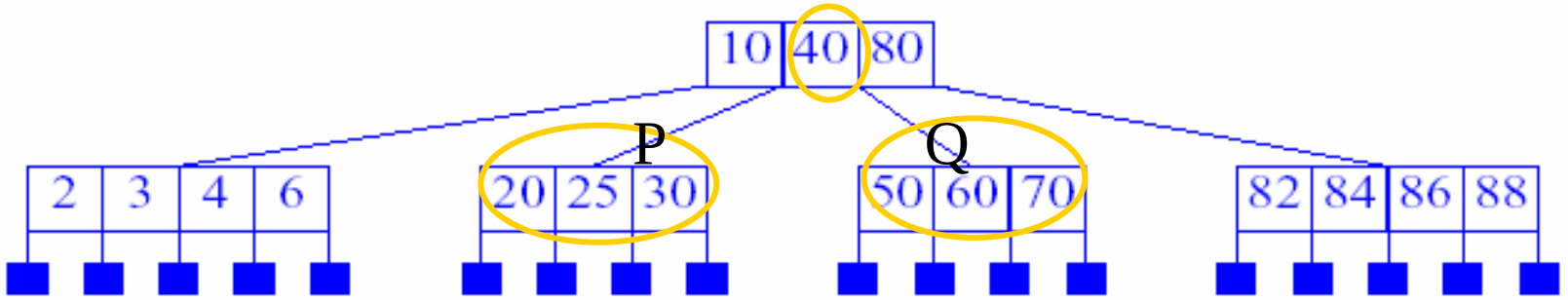


Figure 16.26 (b) Insert 25 into (a)

Growing B tree by Insertion (1)

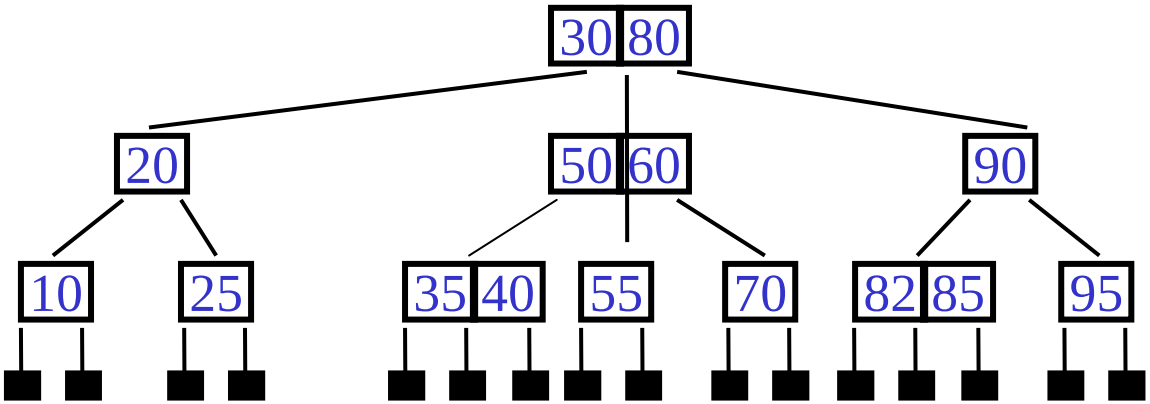
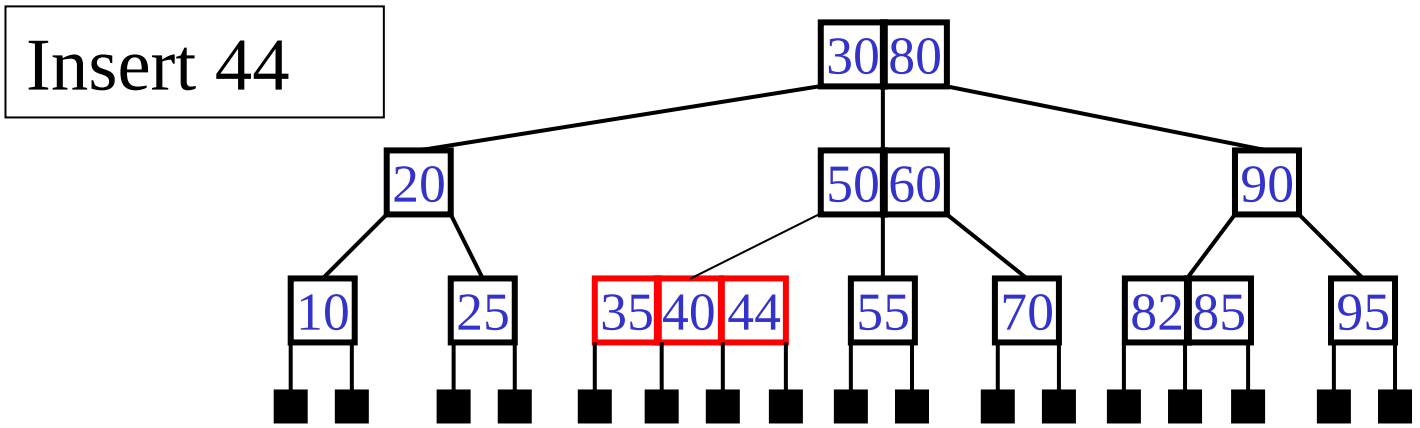


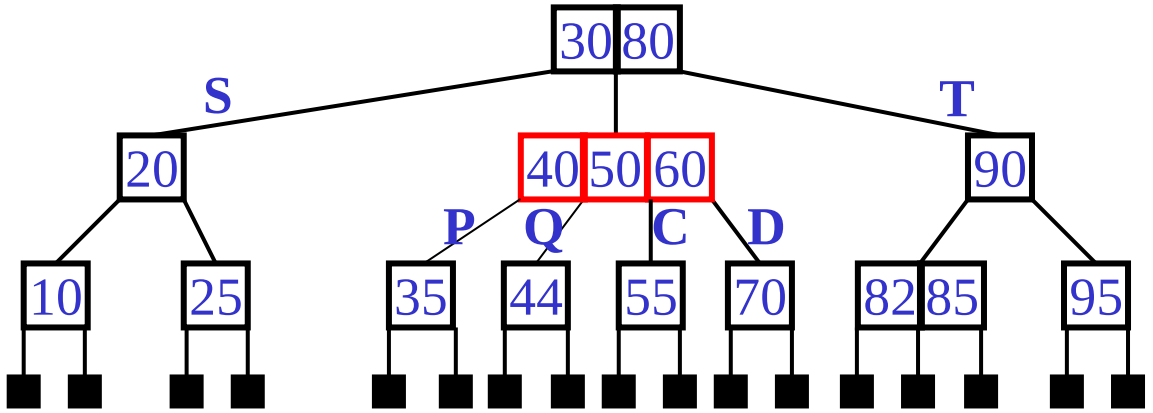
Fig 16.25 B-tree of order 3 (at least 2 pointers)
node format: M, C0, (e1, c1), (e2, c2)… (em, cm)
where m= no of elements, ei = elements, ci = children

Growing B tree by Insertion (2)



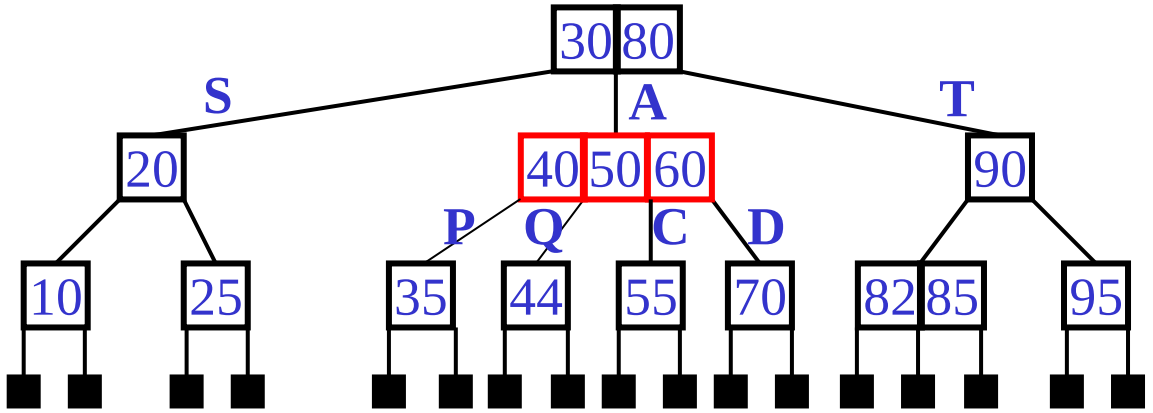
- d = 2 & the target node was (2, c5, (35,c6),(40, c7))
- Overfull node
  - 3, c5, (35,c6), (40,c7), (44,cn)

Growing B tree by Insertion (3)



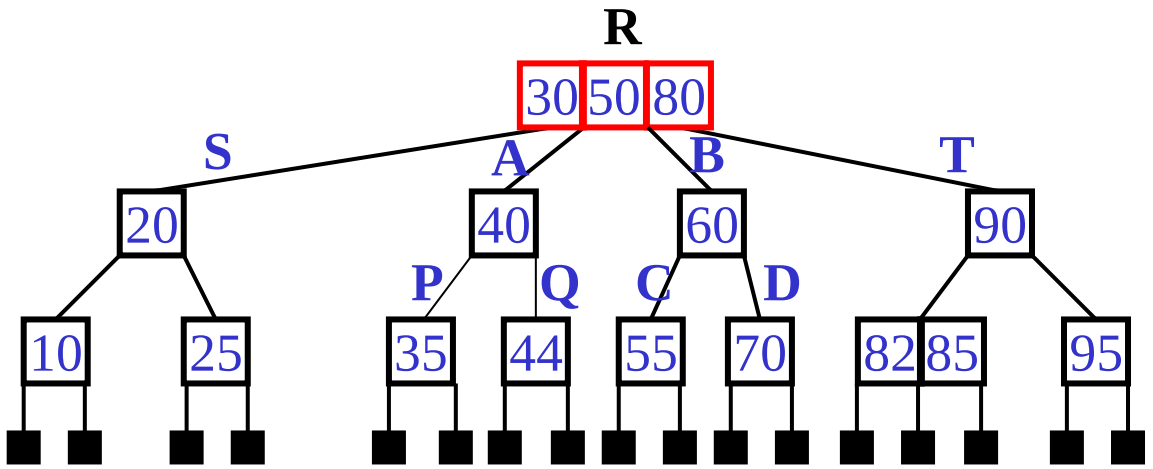
- d= 3/2 ➔ 2, split the overfull node into P & Q
  - P : 1, 0, (35,0)
  - Q : 1, 0, (44,0)
- (40,Q) into the parent A of P
  - Again the parent A is overfull node

Growing B tree by Insertion (4)



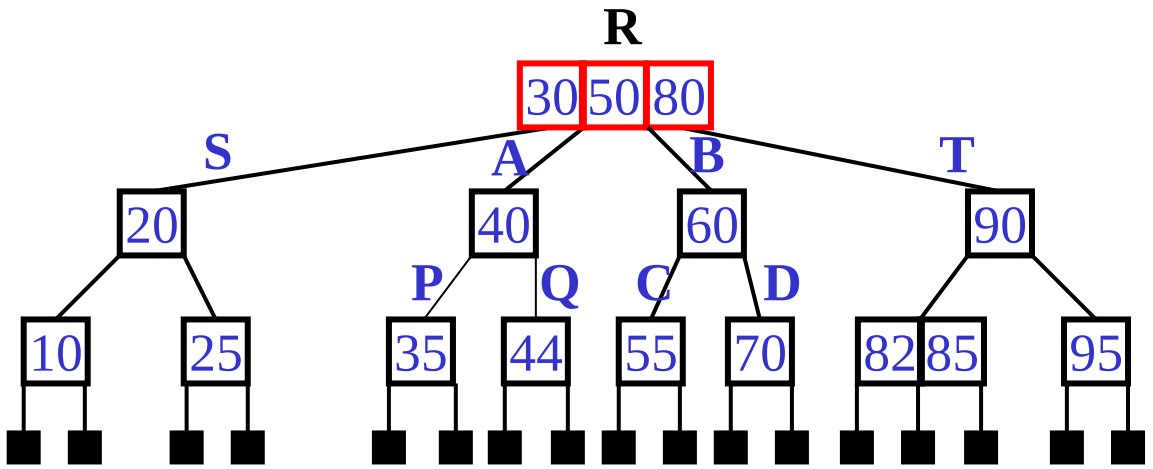
- Node A is again the overfull node
  - A : 3, P, (40,Q), (50,C), (60,D)

Growing B tree by Insertion (5)



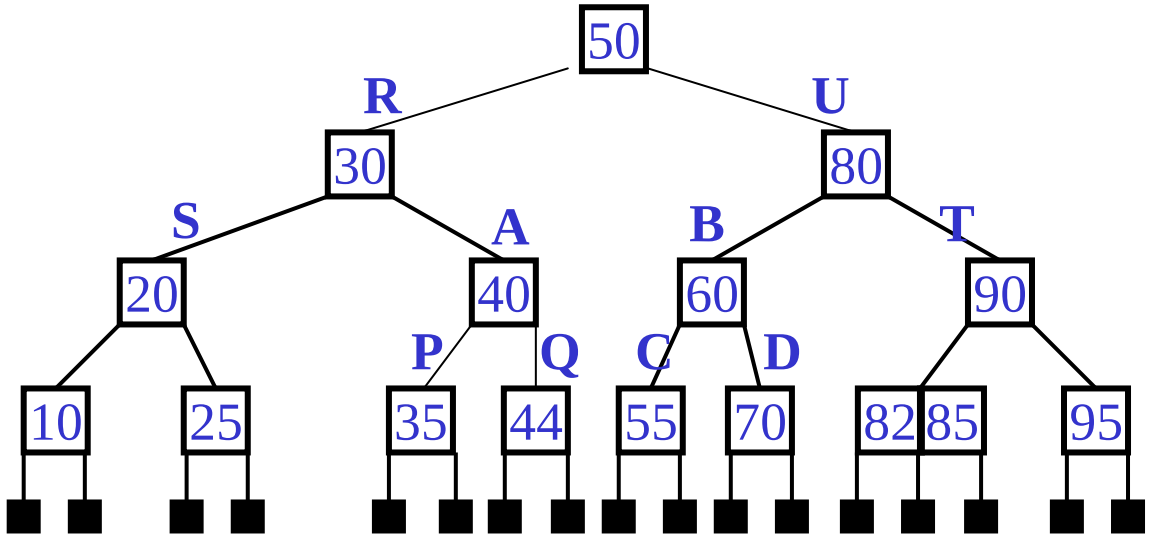
- d= 3/2 = 2, split the node A into A & B
  - A : 1, P, (40,Q)
  - B : 1, C, (60,D)
- Move (50,B) into the parent of A
- Again the parent of A is overfull node

Growing B-tree by Insertion (6)



- The root node R is now the overfull node
  - R : 3, S, (30,A), (50,B), 80,T)

Growing B tree by Insertion (7)



- d= 3/2 ➔ 2, split the root node R into R & U
  - R : 1, S, (30,A)
  - U : 1, B, (80,T)
- Move the new index (50, U) into the parent of R
- R has no parent, we create a new root for the new index

Disk accesses in B tree

- Worst case: Insertion may cause s nodes to split upto root
- Number of disk accesses in the worst case

  h (to read in the nodes on the search path)
  +
  2s (to write out the two split parts of each node)
  +
  1 (to write the new root or the node into which an insertion that does not result in a split is made)

  - ➔ h + 2s + 1
  - ➔ at most 3h + 1 because s is at most h

- The worst scenario is to have 3h+1 disk IOs by splitting

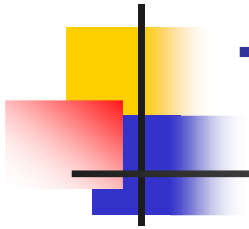
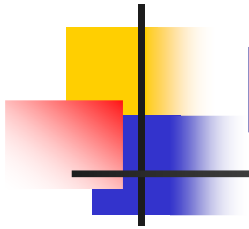


Table of Contents

- B-TREES
  - Indexed Sequential Access Method (ISAM)
  - *m*-WAY Search Trees
  - B-Trees of Order *m*
  - Height of a B-Tree
  - Searching a B-Tree
  - Inserting into a B-Tree
  - *Deletion from a B-Tree*



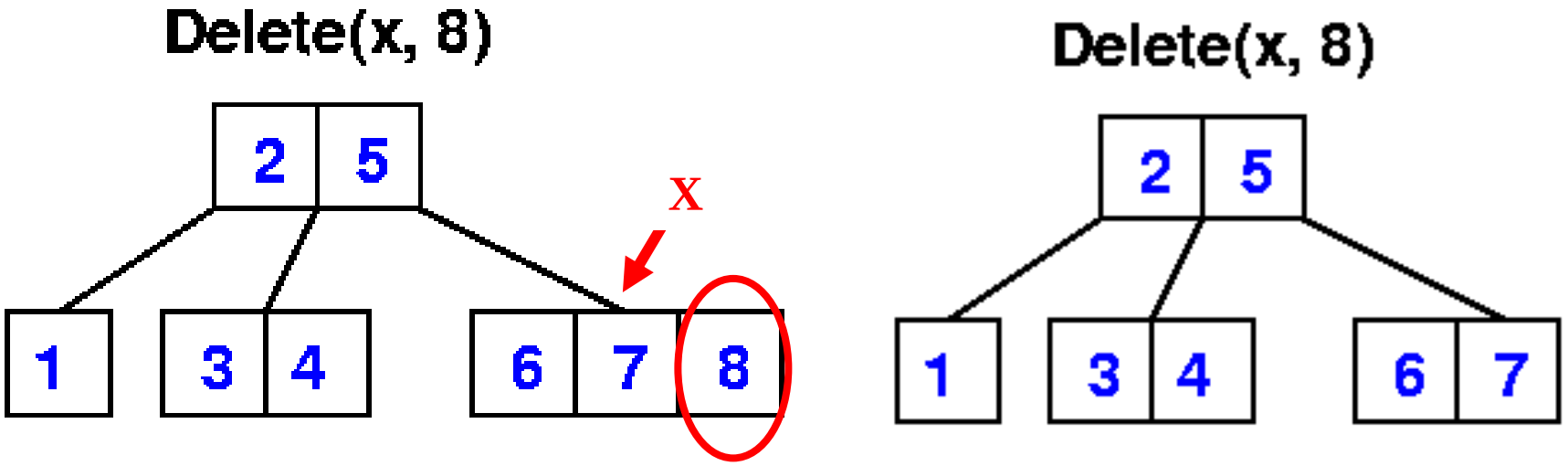
Deletion from a B-Tree

- Deletion cases
  - Case 1: Key k is in the leaf node
  - Case 2: Key k is in the internal node

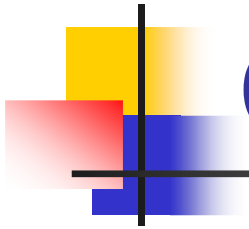
- Case 2
  - by replacing the deleted element with
    - The largest element in its left-neighboring subtree
    - The smallest element in its right-neighboring subtree
  - Replacing element is supposed to be in a leaf, so we can apply case 1

Case 1: Leaf Node Deletion

- If key k is in leaf node, then remove k from leaf node



- If underfull node happens, care must be exercised (will address shortly)



Case 2: Internal Node Deletion

- If the key k is in the internal node x

  One of 3 subcases:

  a. If the left child y preceding k in x has $\geq$ t keys

  b. If the right child z following k in x has $\geq$ t keys

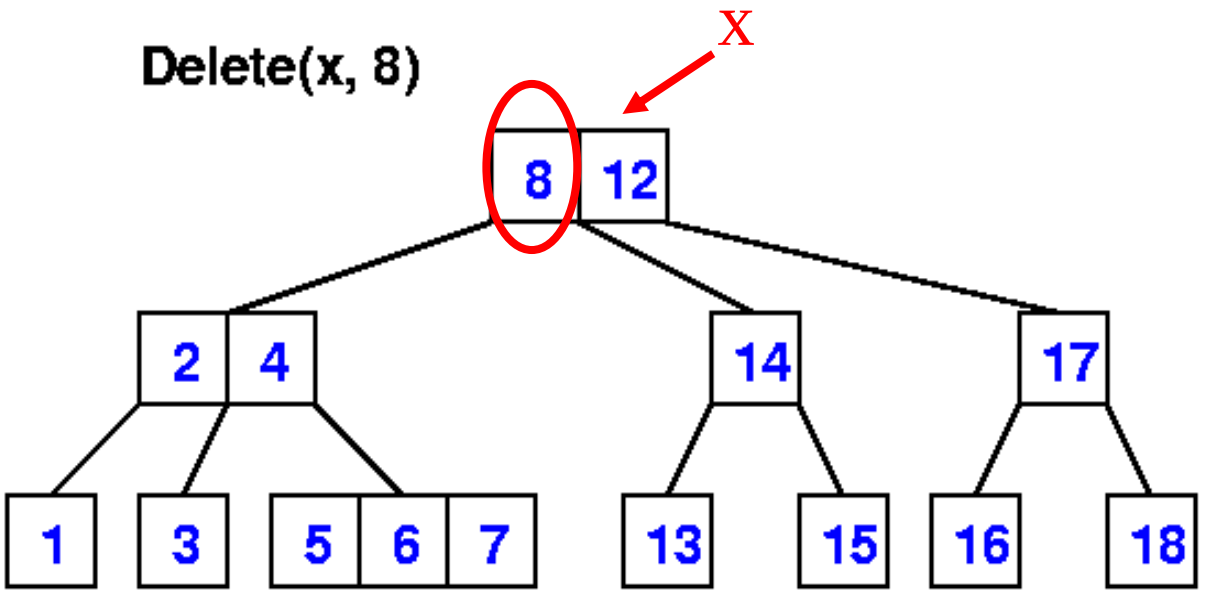
  c. If both the left and right subchild y and z have t-1 keys

- t : $\lceil m/2 \rceil - 1$ (half of the keys)

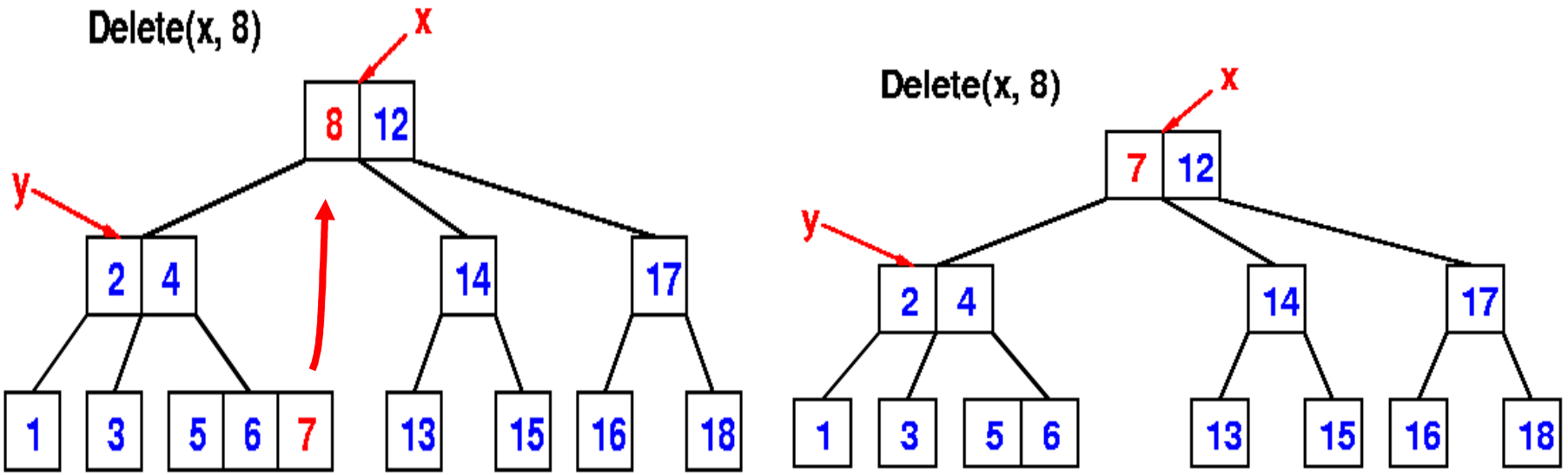
Case 2a: Internal Node Deletion (1)

If the left child y preceding k in x has ≥ t keys

- Find predecessor k' of k in subtree rooted at y
- Replace k by k' in x



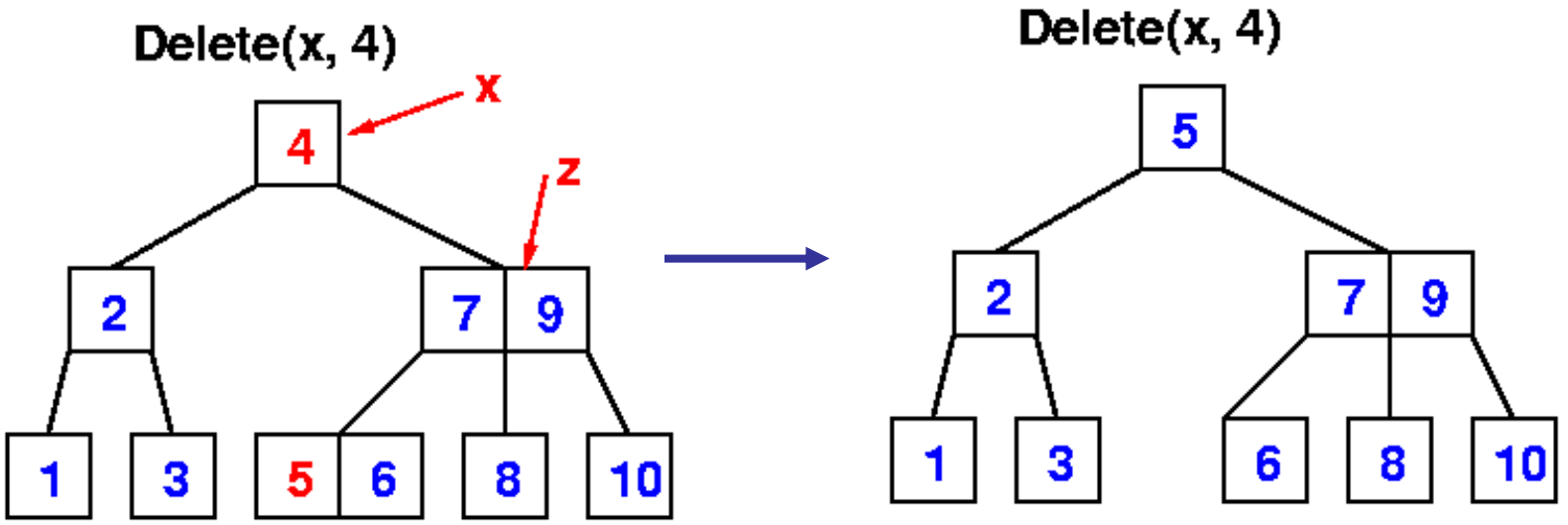
Case 2a: Internal Node Deletion (2)



- If underfull node happens, care must be exercised (will address shortly)

Case 2b: Internal Node Deletion

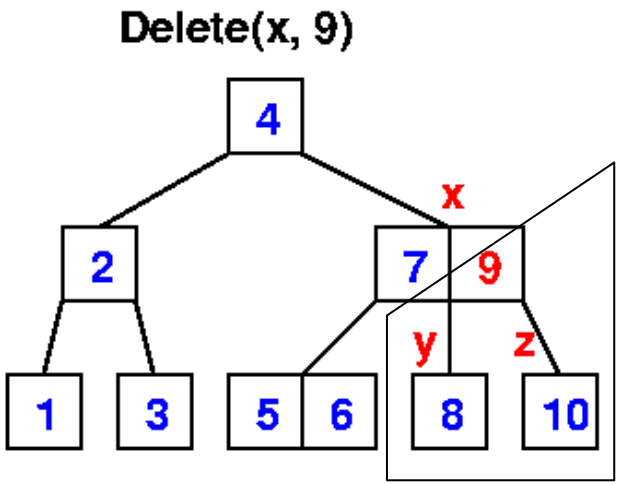
- If the right child z following k in x has ≥ t keys:
  (a) Find successor k' of k in subtree y,
  (b) Replace k by k' in x

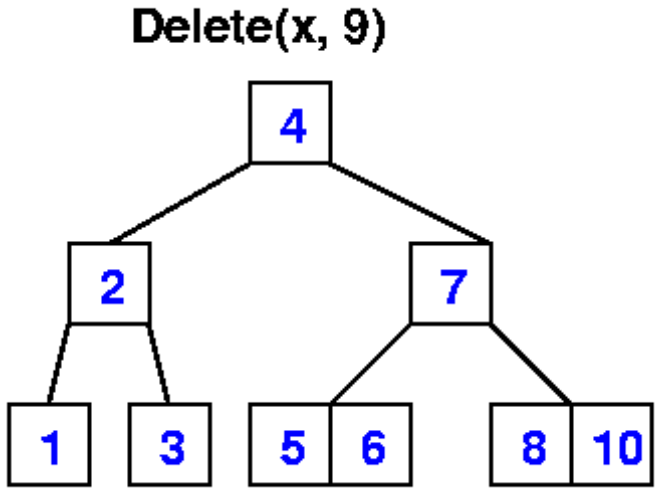


- If underfull node happens, care must be exercised (will address shortly)

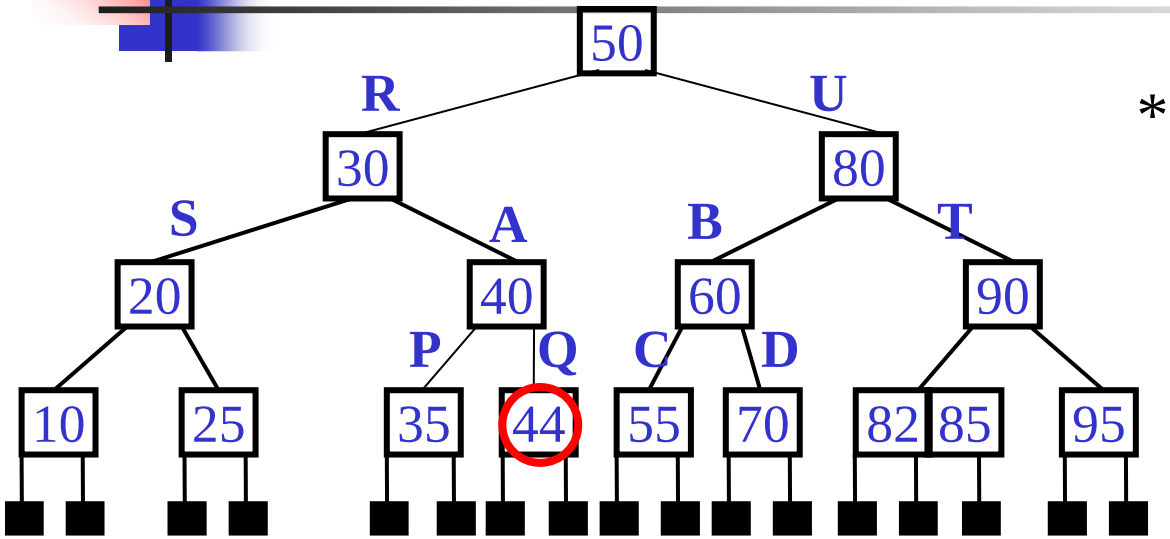
Case 2c: Internal Node Deletion

- If both the left and right subchild y and z have t-1 keys
  - Select the replacement as shown in case 2a or case 2b
  - If underfull node happens, care must be exercised as shown in the below



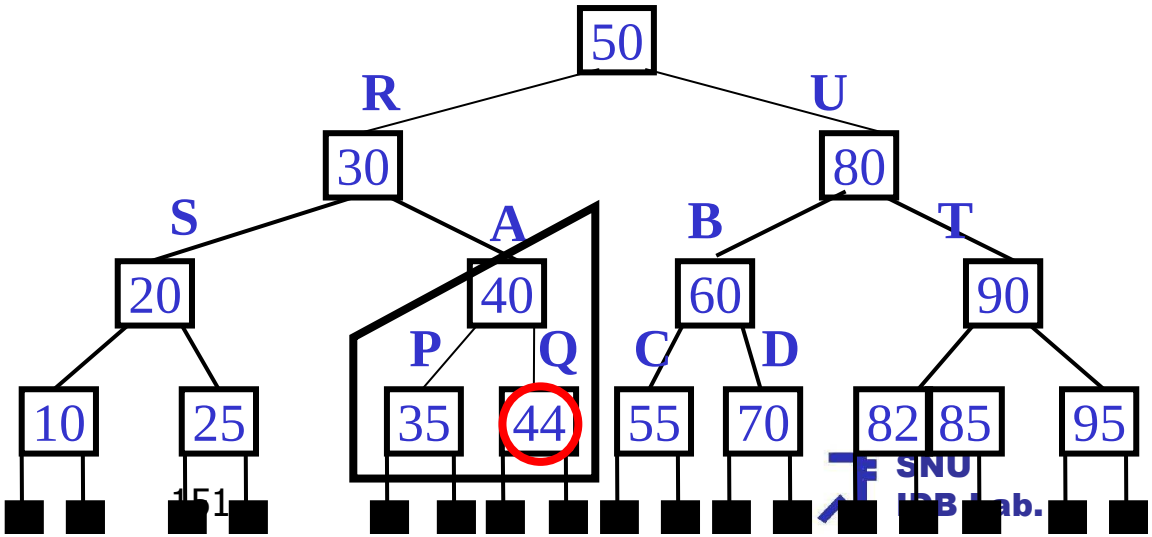


Shrinking B-Tree by Deletion (1)

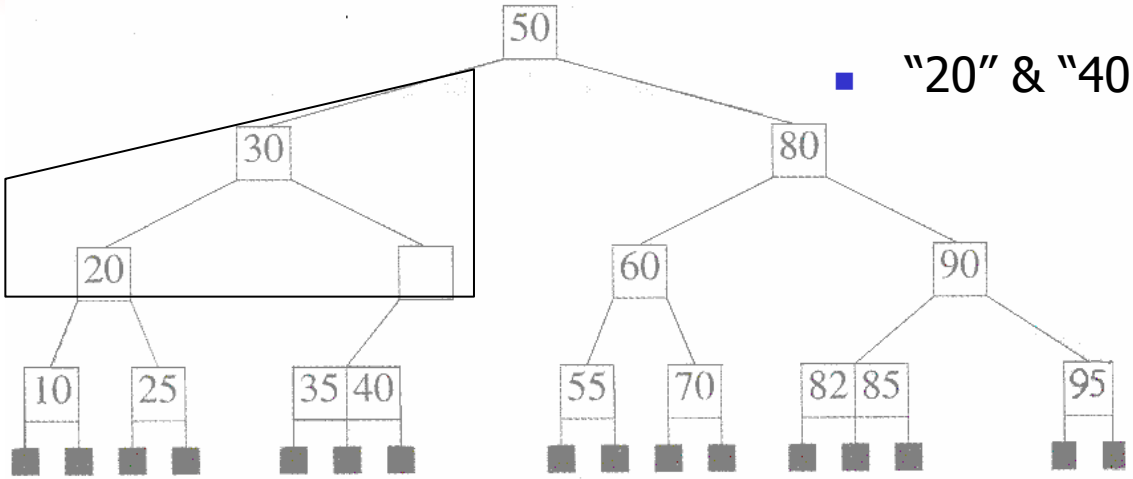


* Try to delete “44”

- After deleting “44”, “35” & “40” are merged



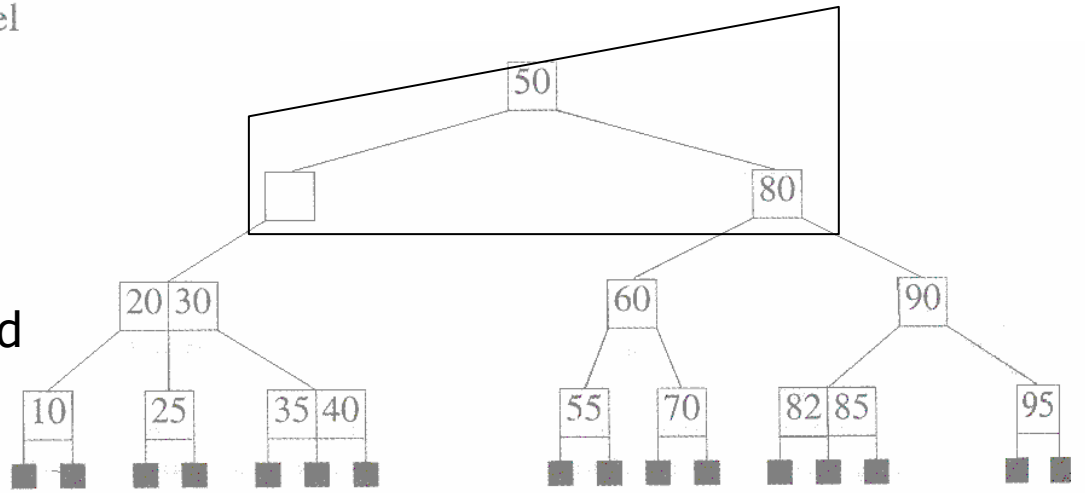
Shrinking B-Tree by Deletion (2)



(a) After merging at leaf level

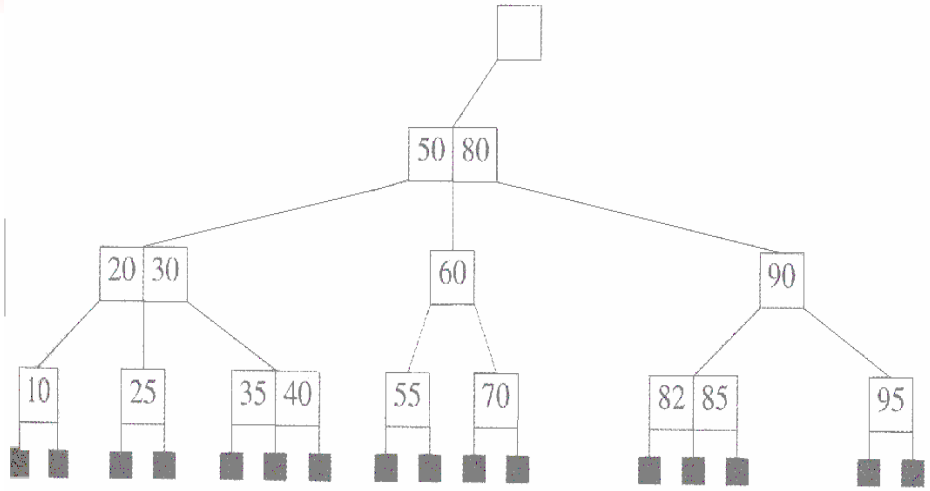
- "20" & "40" also needs to merged

- "50" and "80" also needs to merged



(b) After merging at level 3

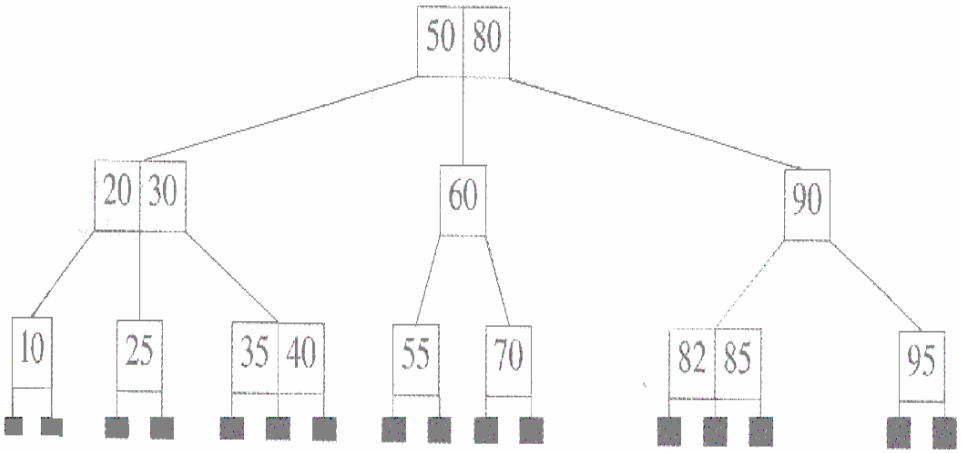
Shrinking B-Tree by Deletion (3)



(c) After merging at level 2

- "50" & "80" are merged and now the old root becomes empty

- Free the old root and make the new root



(d) After deletion of root

Technique for Reducing Node Merging : B tree Deletion with Redistribution (1)

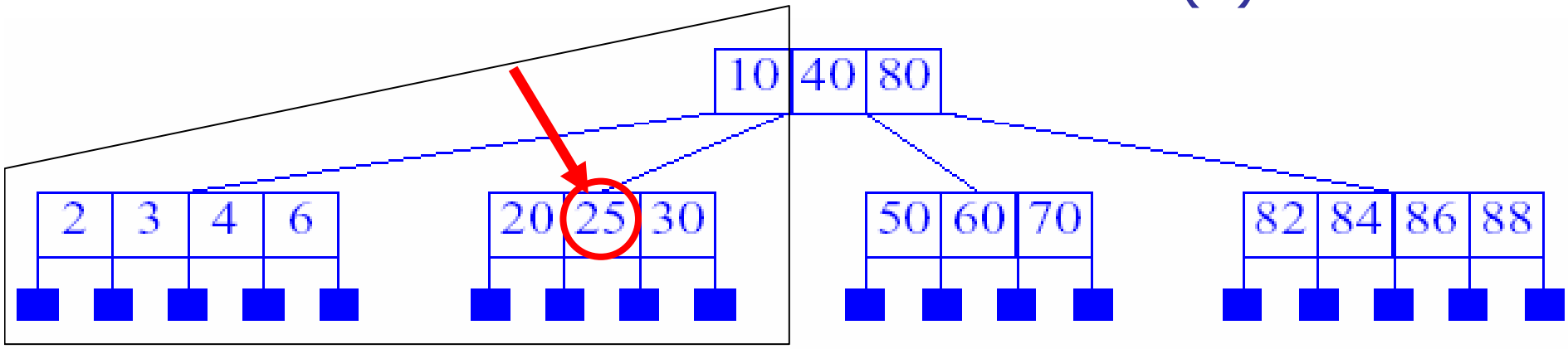
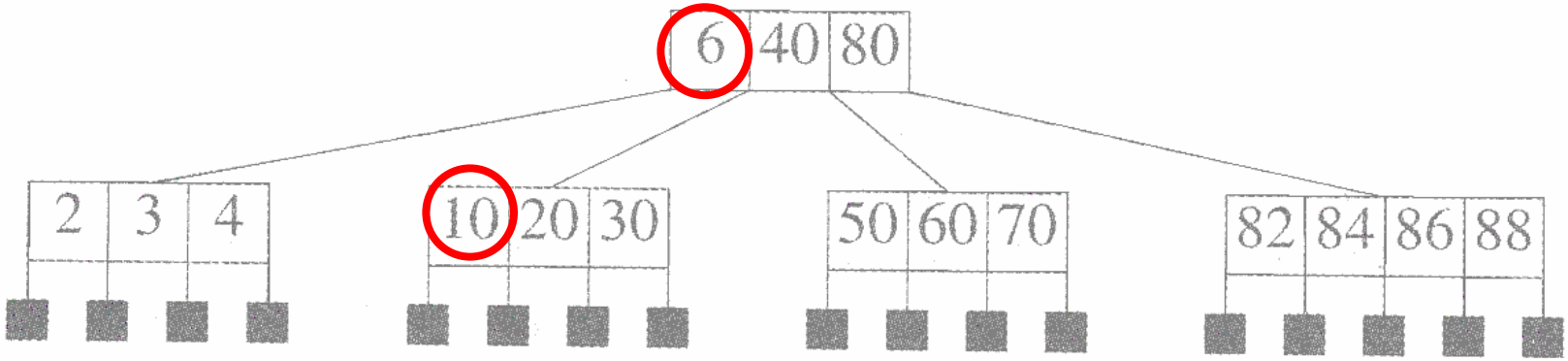


Figure 16.26 (b) Try to delete "25"

- Underflow happens & Redistribute some neighbor nodes
  - Move down 10 & move up 6



Save node merging

Technique for Reducing Node Merging : B tree Deletion with Redistribution (2)

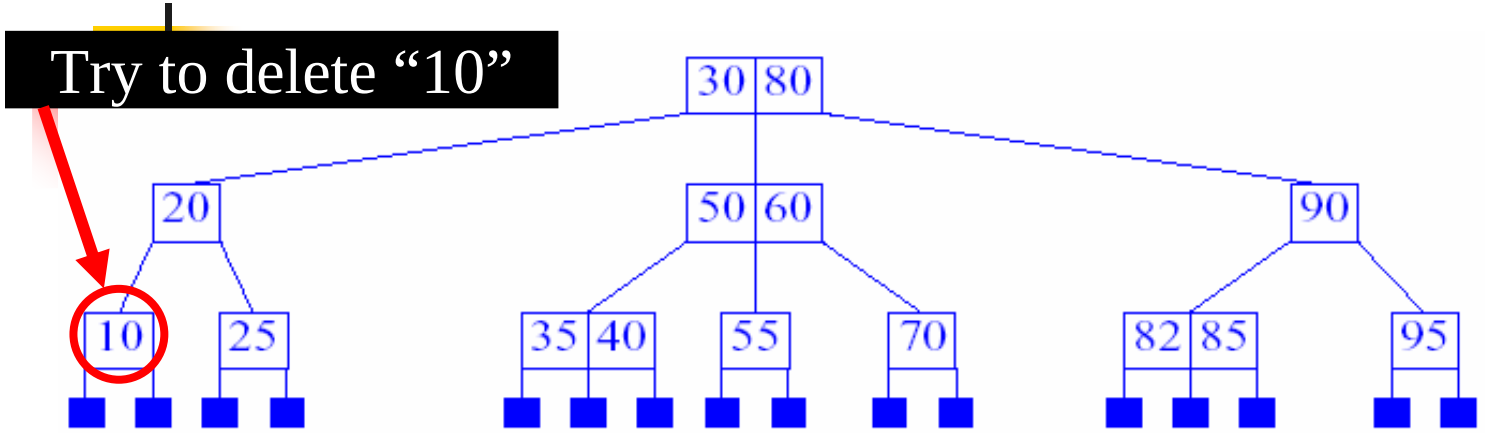
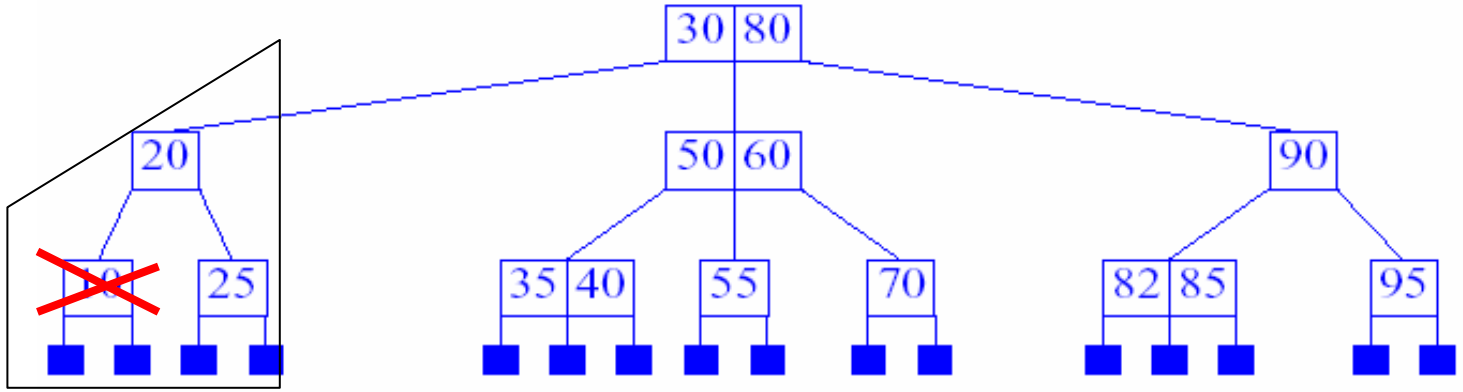
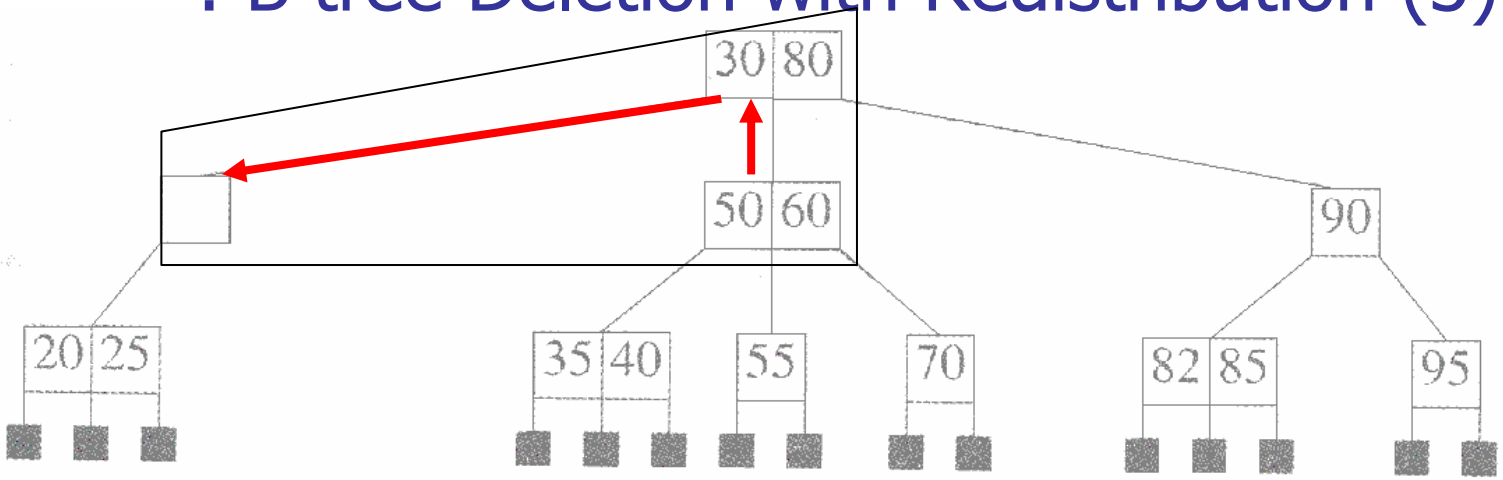


Figure 16.25 A 2-3 tree or B-tree of order 3



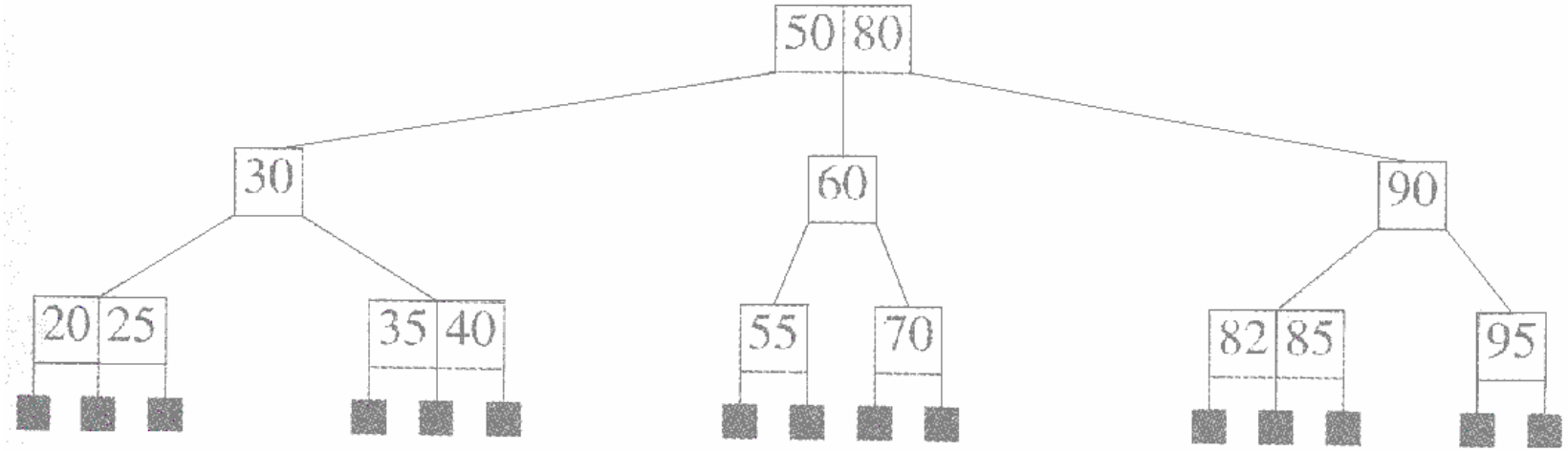
Merging is unavoidable

Technique for Reducing Node Merging : B tree Deletion with Redistribution (3)

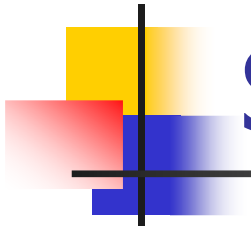


(b) After merging at leaf node

- Consider redistributing some nodes: move down "30" & move up "50"

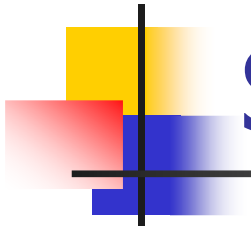


Save propagation of node merging



Summary (0)

- Chapter 15: Binary Search Tree
  - BST and Indexed BST
- Chapter 16: Balanced Search Tree
  - AVL tree: BST + Balance
  - B-tree: generalized AVL tree
- Chapter 17: Graph



Summary (1)

- Balanced tree structures
  - Height is O(log n)
- AVL and Red-black trees
  - Suitable for internal memory applications
- Splay trees
  - Individual dictionary operation ➔ 0(n)
  - Take less time to perform a sequence of u operations ➔ 0(u log u)
- B-trees
  - Suitable for external memory