CSE214 Data Structures
Balanced Trees

YoungMin Kwon

Balanced Search Trees

- Running time of standard binary search tree
  - For a random series of insertion and removal, *expected running time* is O(log n)
  - However, *worst-case running time* is O(n) when the tree is unbalanced

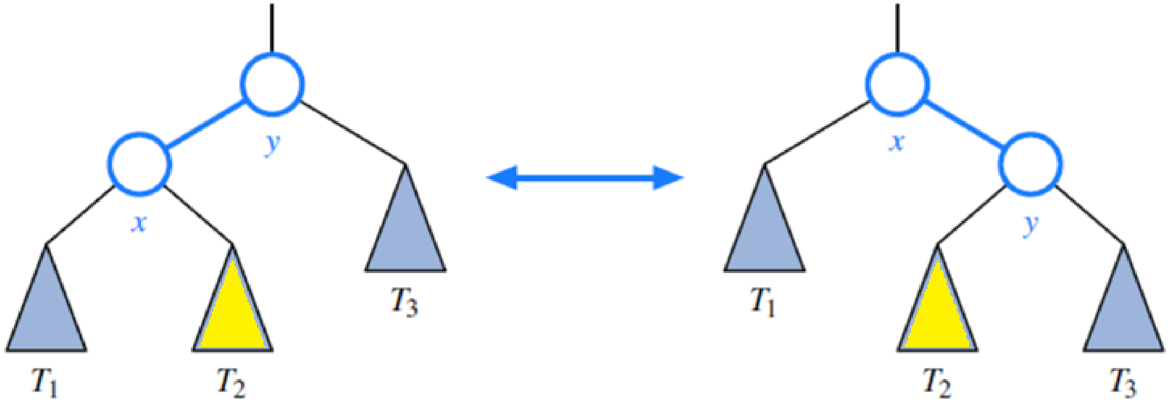
Balanced Search Trees

- Given a binary search tree T,
  - A *position* is *balanced* if the difference between the height of its children is at most 1; otherwise, *unbalanced*

- Balanced search trees reshape the tree structure to be balanced

Balanced Search Trees

- Rotation operation: *rotate*(*x*)



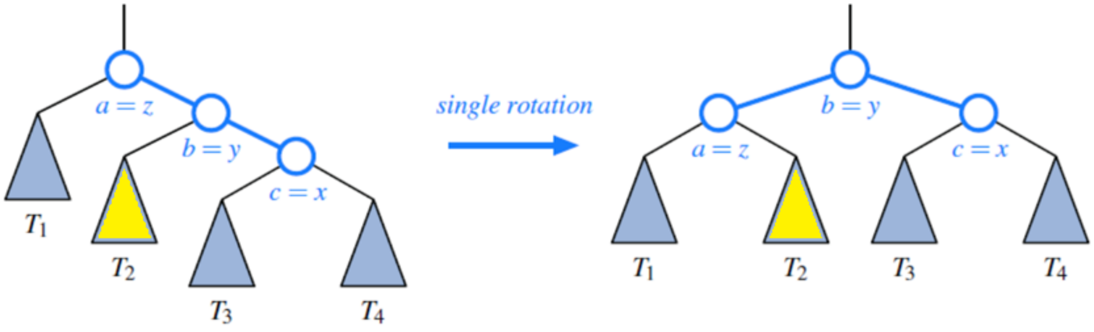
  - Keys in T1 are less than x
  - Keys in T2 are in between x and y
  - Keys in T3 are greater than y

Balanced Search Trees

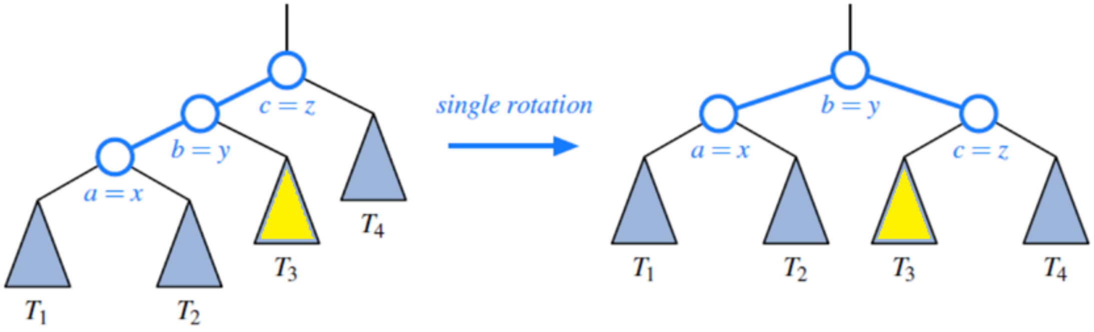
- Trinode restructuring
  - Combine one or more rotations to provide a broader rebalancing within a tree
  - Given a *position x*; its *parent y*; and its *grand parent z*
  - Let *a*, *b*, and *c* be their renamed positions in the order of their keys ⇒ make b the parent of a and c

Balanced Search Trees

- Trinode restructuring: *restructure*(*x*)
    - Case 1: y and x are on the same side
    - *restructure*(*x*) = *rotate*(*y*)



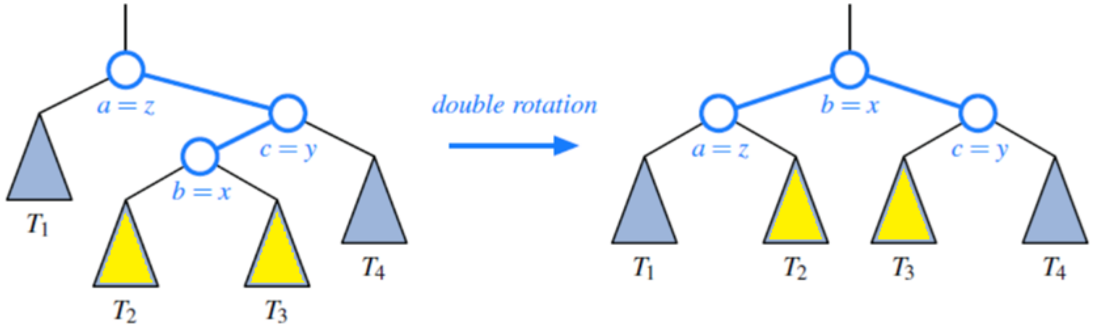
(a)



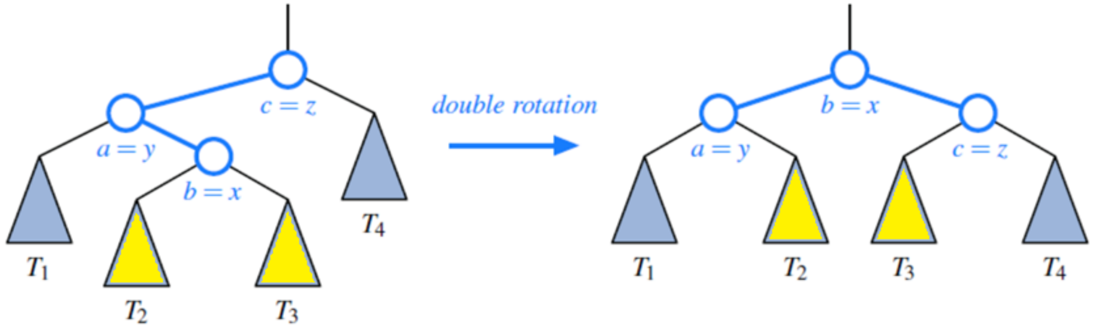
(b)

Balanced Search Trees

- Trinode restructuring: : *restructure*(*x*)
  - Case 2: y and x are on different sides
  - *restructure*(*x*) = *rotate*(*x*) and *rotate*(*x*)



(c)



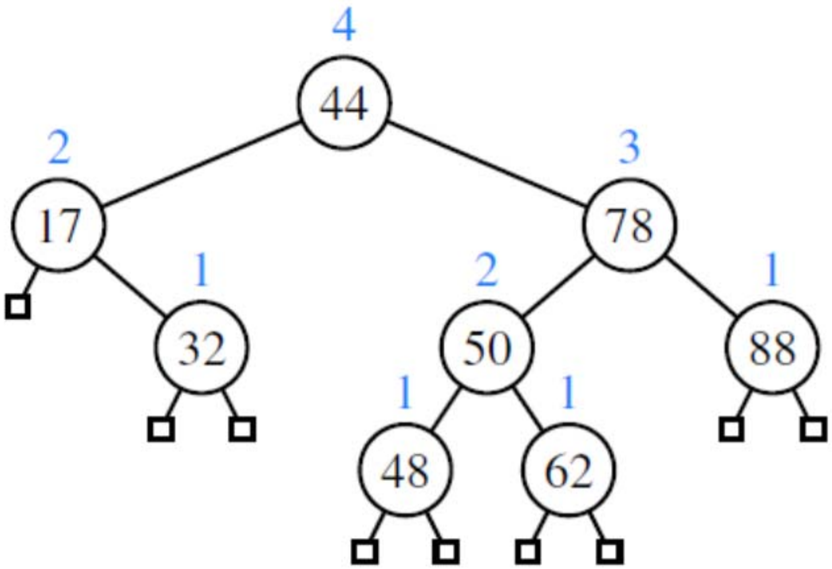
(d)

AVL Trees

- *Height-balanced property*
  - For every internal position *p* of T, the heights of the children of *p* differ by at most 1
- AVL tree
  - Any binary tree that satisfies the height-balanced property is an AVL tree
  - AVL is named after the initials of its inventors: Adel'son, Vel'skill, and Landis

AVL Tree

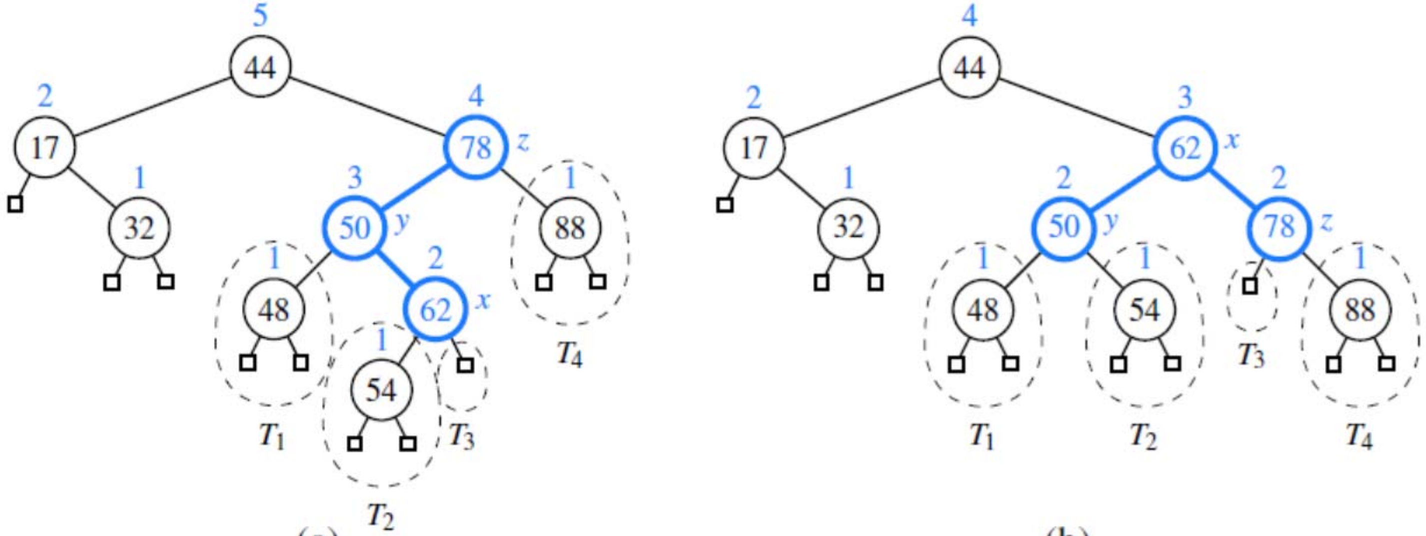
- Example



  - The numbers on nodes are the height of the subtree rooted at the node

AVL Trees

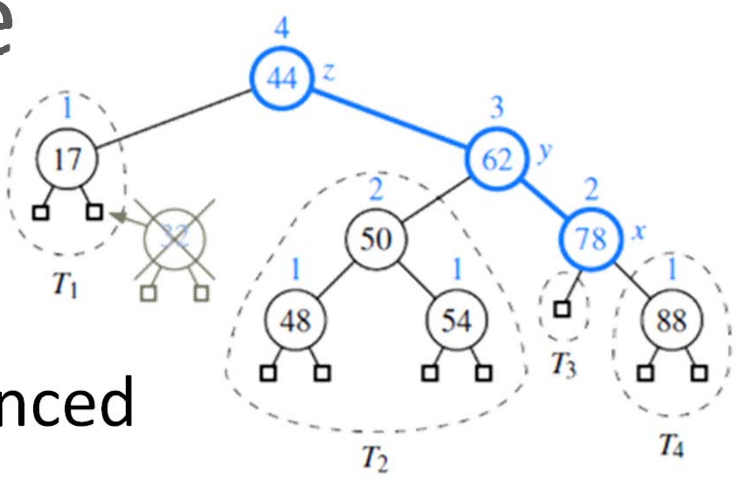
- Insertion
  - A tree T was height balanced before adding a new entry
  - Adding an entry to a position *p* may violate the height-balanced property
  - The only positions that may be unbalanced are the ancestors of *p*



(a) (b)

- Insertion example
  - (a) after adding 54 to an AVL tree
    - z is the first position that become unbalanced
    - y and x are its child and grand child with greater heights than their siblings
  - (b) *Trinode-restructuring on x* restores the balance

AVL Tree



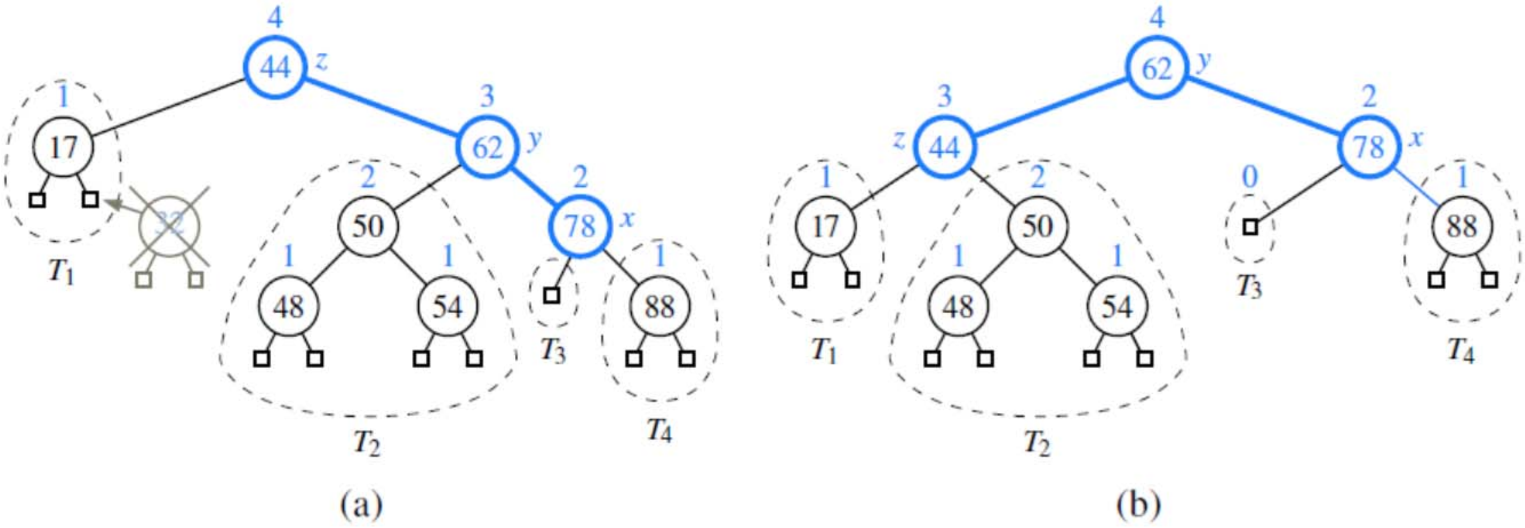
- Deletion
  - Removing an entry from a balanced tree may unbalance the tree
    - *z* is the first unbalanced position
    - *y* is *z*’s taller child
    - *x* is *y*’s taller child or if they have the same height the same sided one as *y*
  - *Trinode-restructuring on x* restores the balance locally

AVL Tree

- Deletion
  - The restructuring may reduce the height of the subtree and may unbalance ancestors of the tree
    - Repeat until the height is not changed or the root is reached

AVL Trees

- Deletion example
  - (a) 32 is deleted from an AVL tree
  - (b) after trinode-restructuring on 78 (*x*)



AVL Trees

- Proposition
  - The height *h* of an AVL tree storing *n* entries is O(log *n*)

- Justification
  - Let n(*h*) be the minimum number of *internal* nodes of an AVL tree with height *h*
  - We will show that n(*h*) grows at least exponentially

AVL Trees

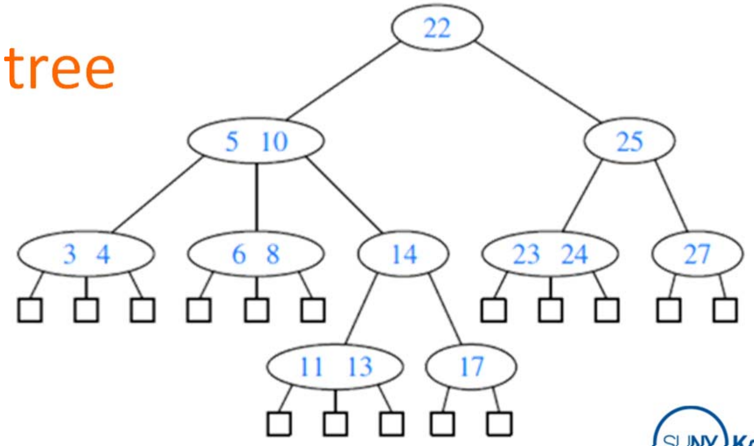
- Justification (continued)
  - $n(1) = 1, n(2) = 2,$
  - $n(h) = 1 + n(h - 1) + n(h - 2)$
    - root, subtree with height $h-1$, subtree with height $h-2$
  - Because $n(h - 1) > n(h - 2)$
    - $$\begin{aligned} n(h) &> 2 \cdot n(h - 2) \\ &> 4 \cdot n(h - 4) \\ &> 8 \cdot n(h - 6) \\ &\dots \\ &> 2^i \cdot n(h - 2 \cdot i) \end{aligned}$$

AVL Trees

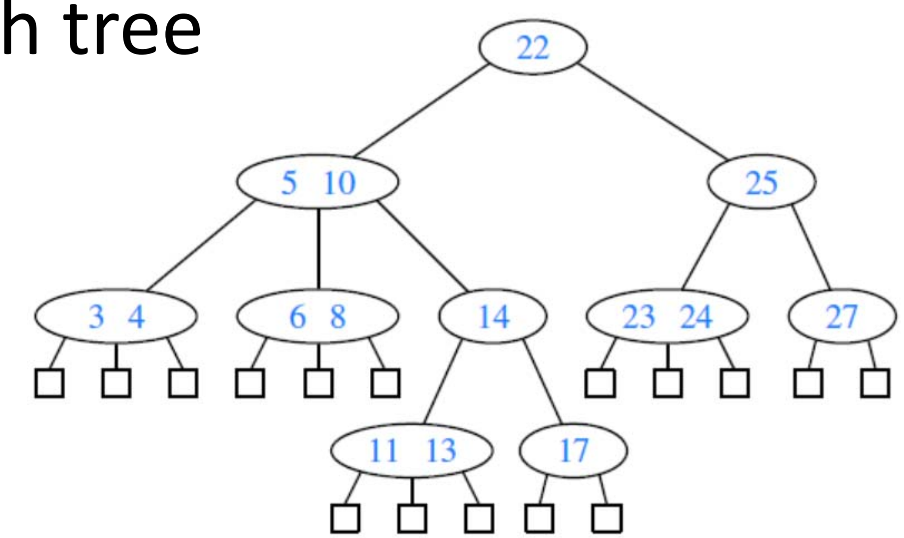
- Justification (continued)
  - Pick i such that $h - 2 \cdot i$ is either 1 or 2 $(i = \lceil h/2 \rceil - 1)$
    - $$\begin{aligned} n(h) &> 2^{\lceil h/2 \rceil - 1} \cdot n(h - 2\lceil h/2 \rceil + 2) \\ &\geq 2^{\lceil h/2 \rceil - 1} \cdot n(1) \\ &\geq 2^{h/2 - 1} \end{aligned}$$
  - Taking logs on both sides
    - $\log(n(h)) > h/2 - 1$
    - Hence, $h < 2 \log(n(h)) + 2$
  - AVL trees storing n entries have height at most $2 \log n + 2$

Multiway Search Tree

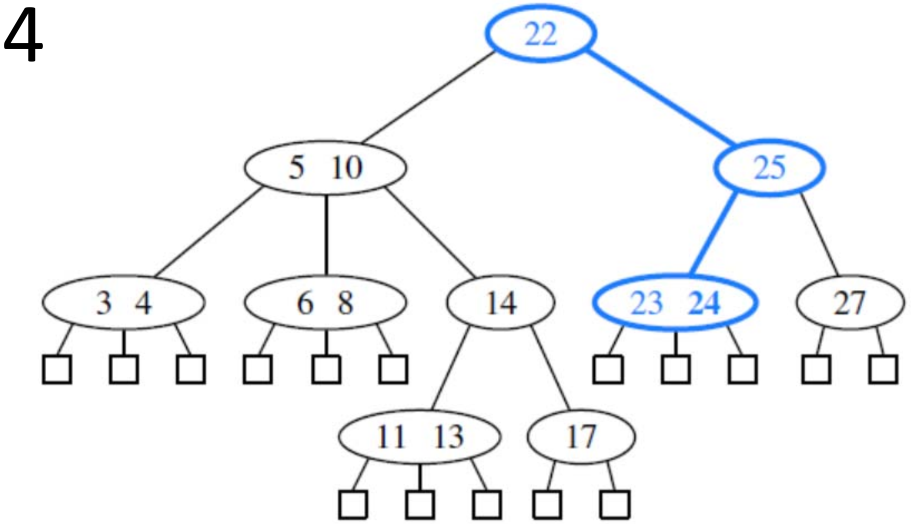
- Multiway search tree T
  - Let w be a node of an ordered tree; w is *d-node* if w has d children
  - Each internal node of T has at least two children
  - Each internal *d-node* of T has an ordered set of *d-1* keys: $k_1 \leq \ldots \leq k_{d-1}$
  - Each entry stored at a subtree rooted at c_i has keys k such that $k_{i-1} \leq k \leq k_i$



- A multiway search tree

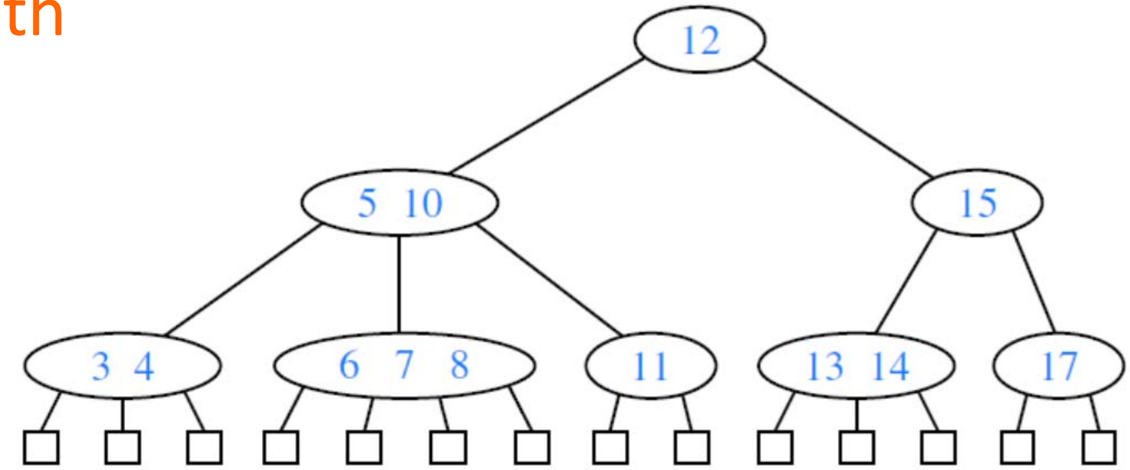


- Search path for 24



(2,4)-Trees

- (2,4) tree AKA 2-4 tree or 2-3-4 tree
  - A multiway search tree (2-4 tree is a B-tree)
  - *Size property*: every internal node has at most 4 children
  - *Depth property*: all external nodes have the same depth



(2,4)-Trees

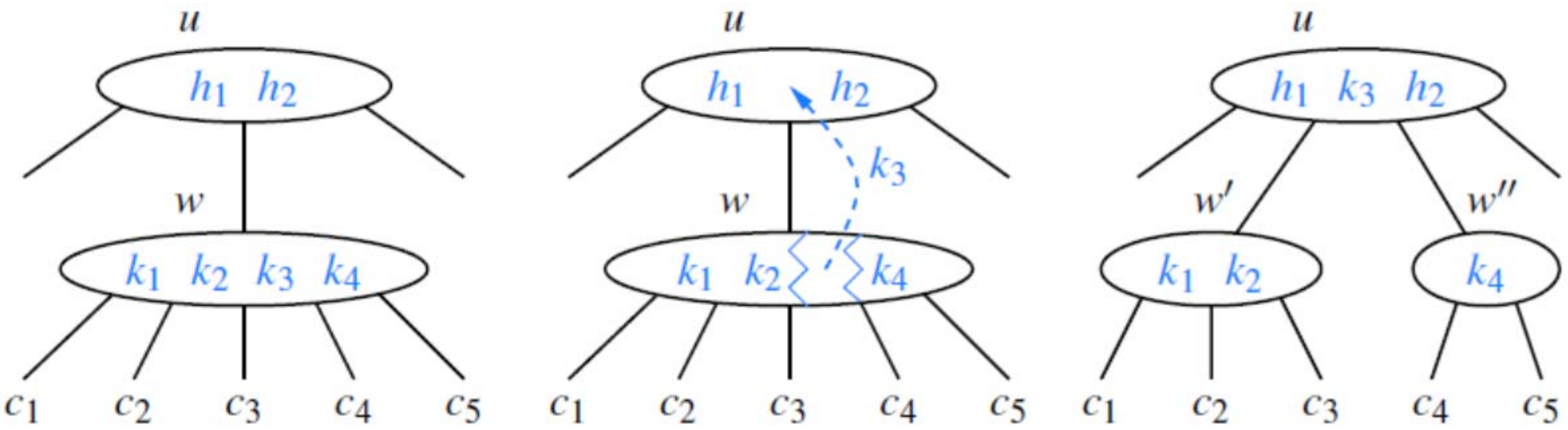
- Insertion
  - Add the new entry to an external node
    - The depth property is preserved, but the size property may be violated (overflow)
  - Split operation to fix the overflow

(2-4)-Trees

- Split operation on w
  - Replace w with two nodes w’ and w’’
    - w' is a 3-node with children c_1, c_2, c_3 and keys k_1 and k_2
    - w'' is a 2-node with children c_4, c_5 and keys k_4
  - If w is the root of T, create a new node u; otherwise, u is the parent of w
  - Insert k_3 into u and make w' and w'' children of u
- As a consequence of a split on w, an overflow may occur on u
  - Split u until no more overflow occurs

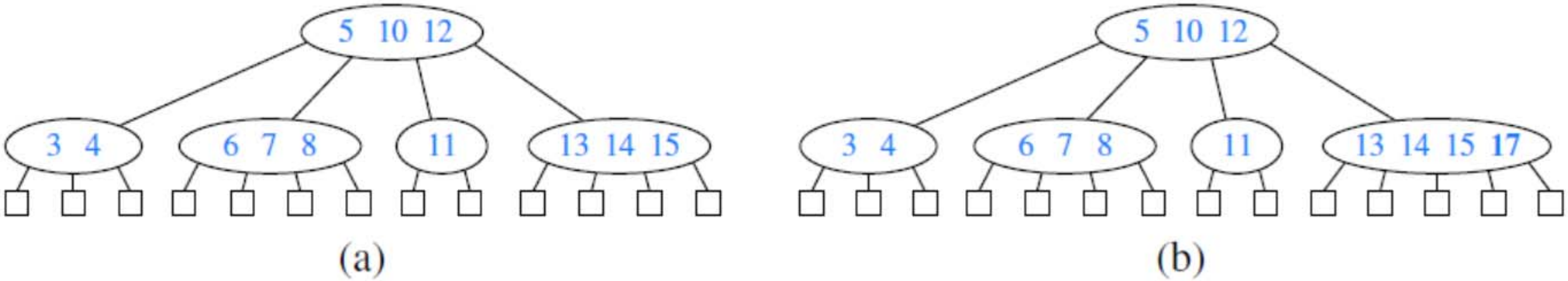
(2,4)-Trees

- Split operation



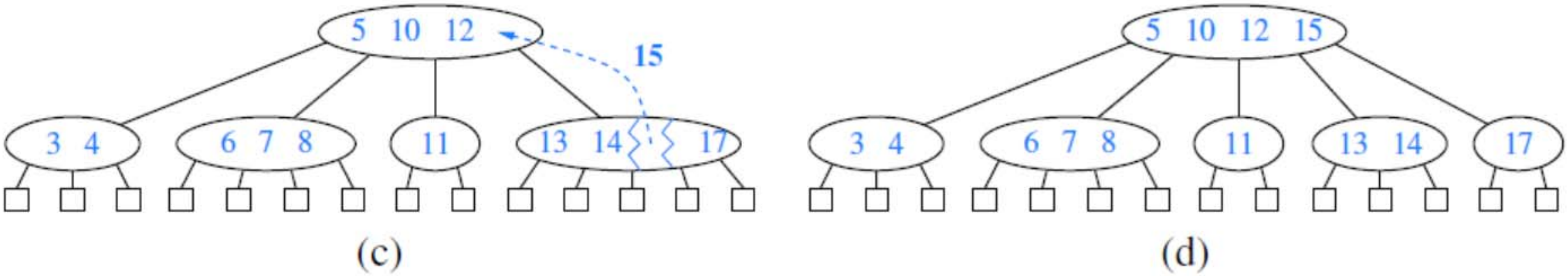
(2-4)-Trees

- Insertion



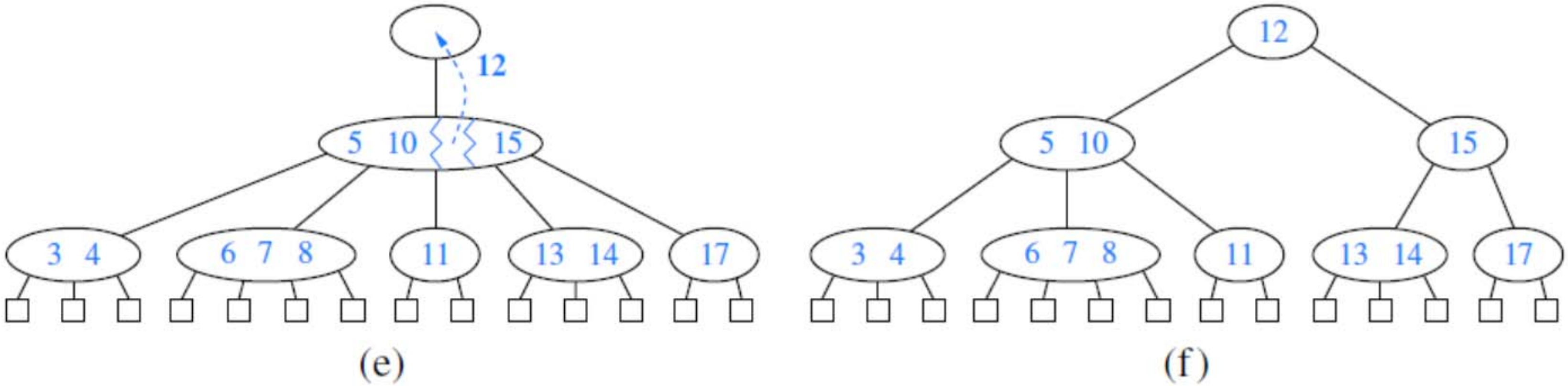
  - (a) before the insertion
  - (b) adding 17 causes an overflow

(2,4)-Trees



- (c) split
- (d) after split, a new overflow occurs

(2,4)-Trees



- (e) another split, creating a new root node
- (f) final tree

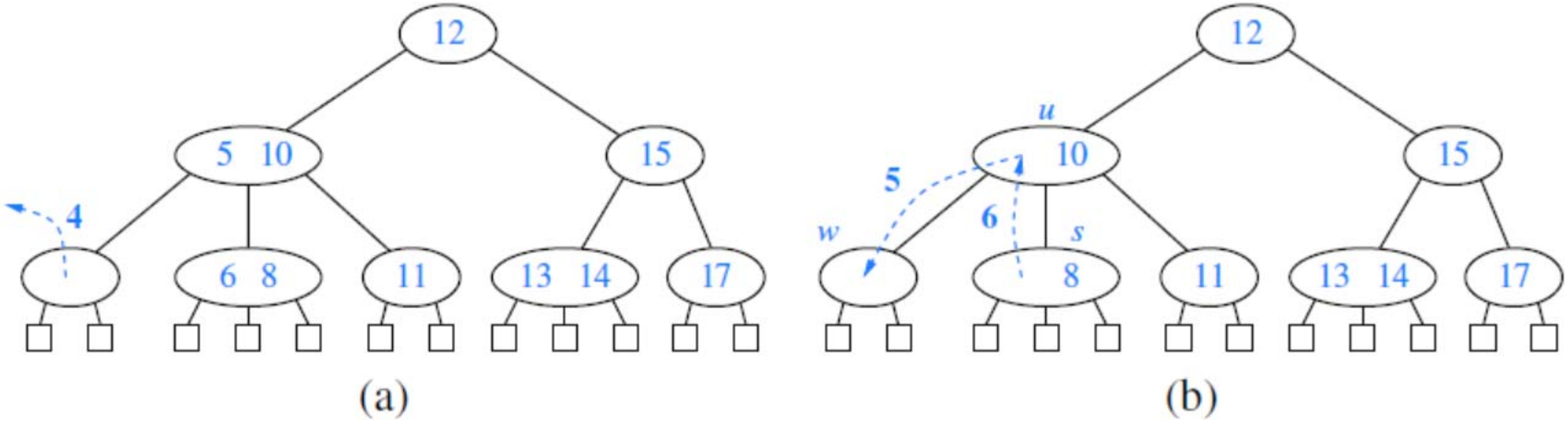
(2,4)-Trees

- Deletion
  - Removing an internal node → removing an external node
    - When removing (k_i, v_i) from z, find the rightmost external node rooted at the i^{th} child
    - Swap (k_i, v_i) at z with the last entry of w
  - Removing an entry preserves the depth property, but the size property may be violated (underflow)

(2,4)-Trees

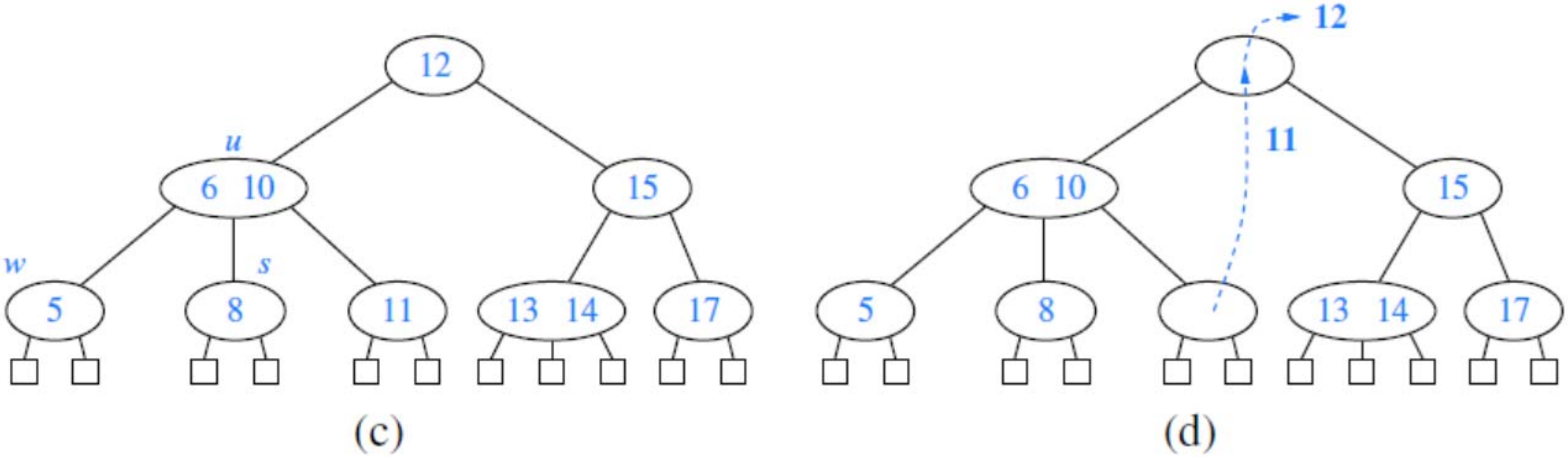
- To fix underflow at w
  - Transfer operation: if an immediate sibling is a 3-node or 4-node, move a key of the sibling to w
  - Fusion operation: otherwise,
    - Merge w with a sibling
    - Create a new node w’
    - Move a key from the parent u of w to w’

(2,4)-Trees



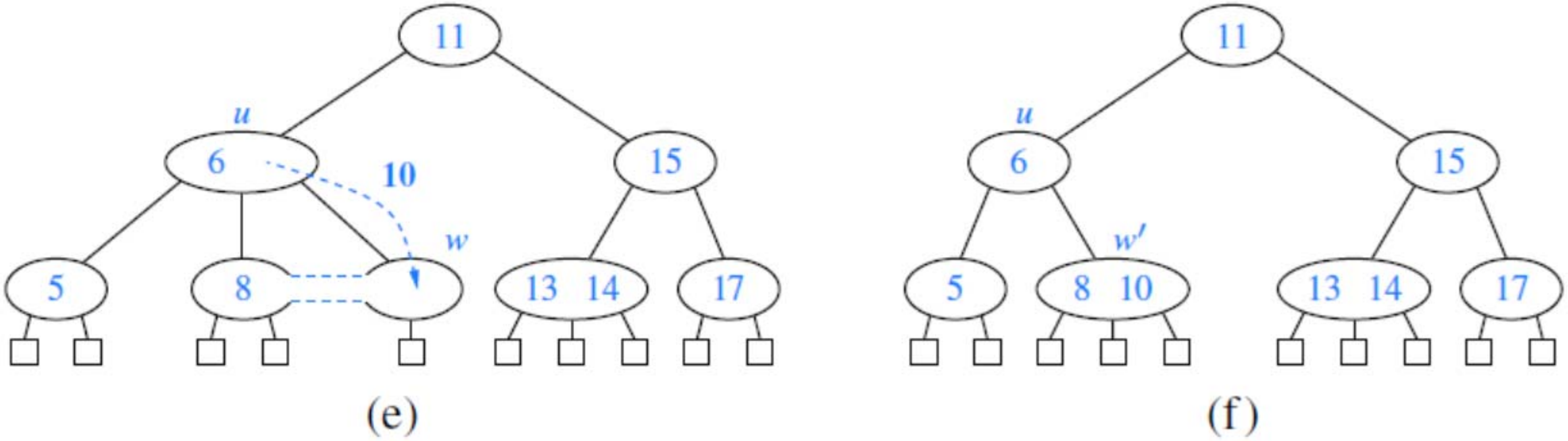
- (a) removal of 4 caused underflow
- (b) a transfer operation

(2,4)-Trees



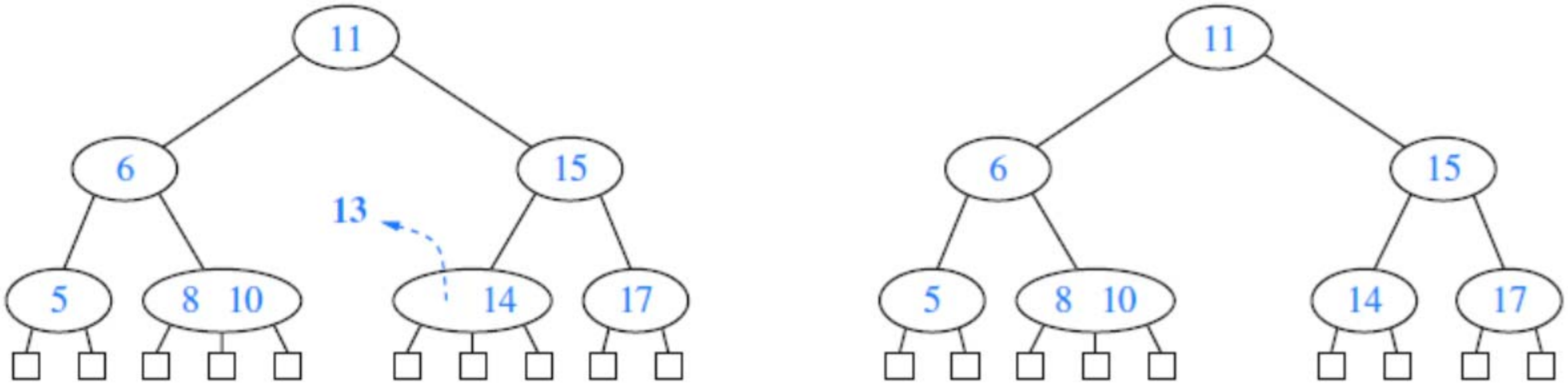
- (c) after the transfer operation
- (d) removal of 12, causes an underflow at an external node

(2,4)-Trees



- (e) a fusion operation
- (f) after the fusion operation

(2,4)-Trees

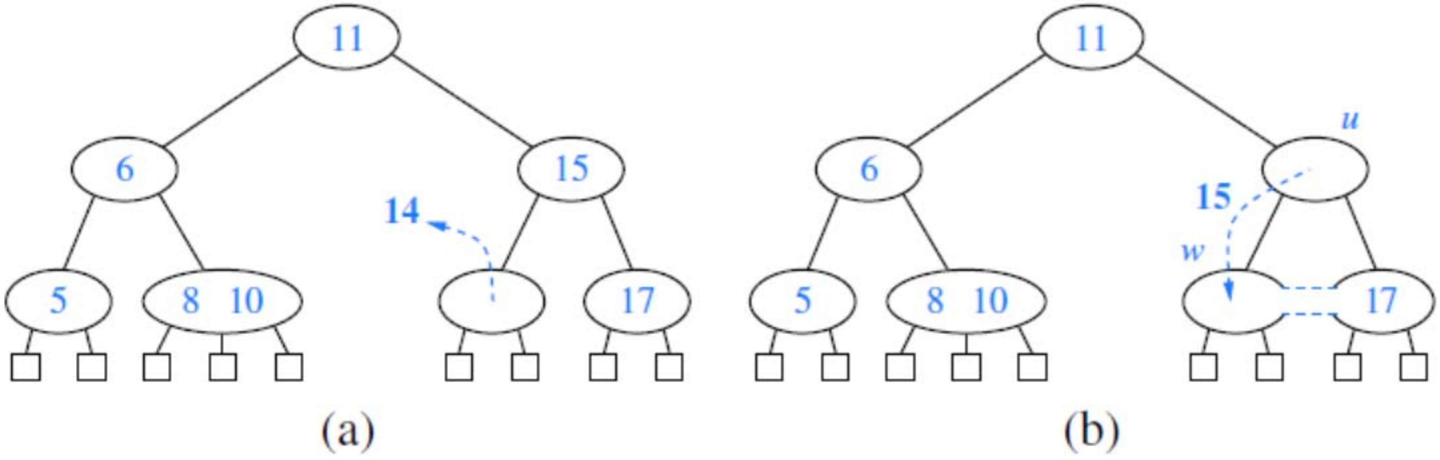


(g)

(h)

- (g) removal of 13
- (h) after removing 13

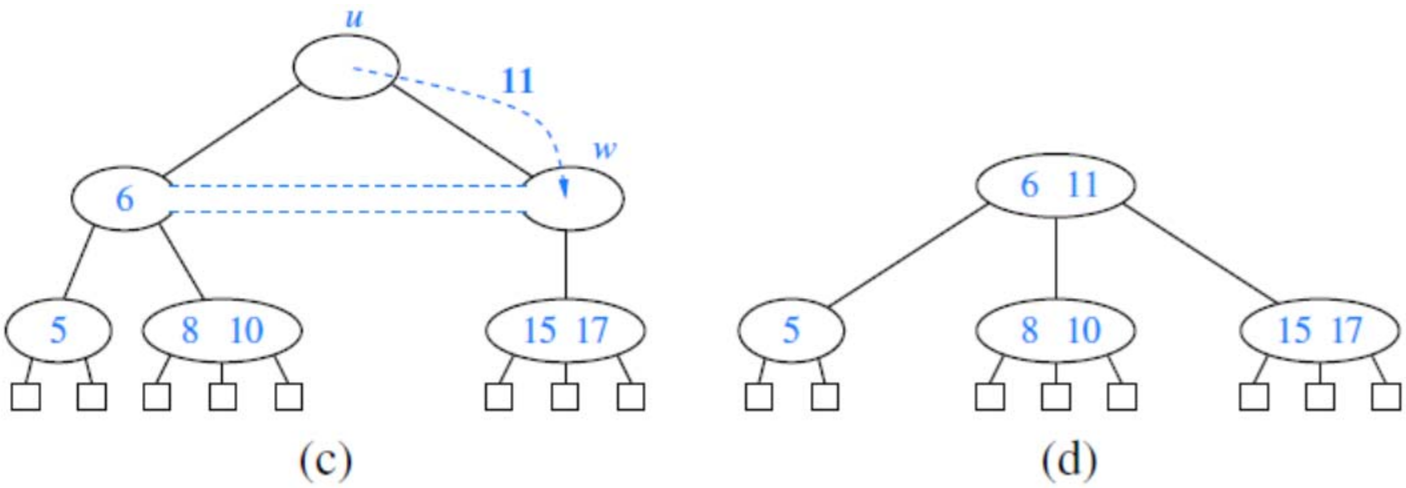
(2,4)-Trees



(a) (b)

- (a) removal of 14 causes an underflow
- (b) fusion, which causes another underflow

(2,4)-Trees



(c) (d)

- (c) second fusion, which causes the root to be removed
- (d) final tree

- Exercise:
  - Remove 17, 15, 14, 13 from the first (2,4)-tree example

(2,4)-Trees

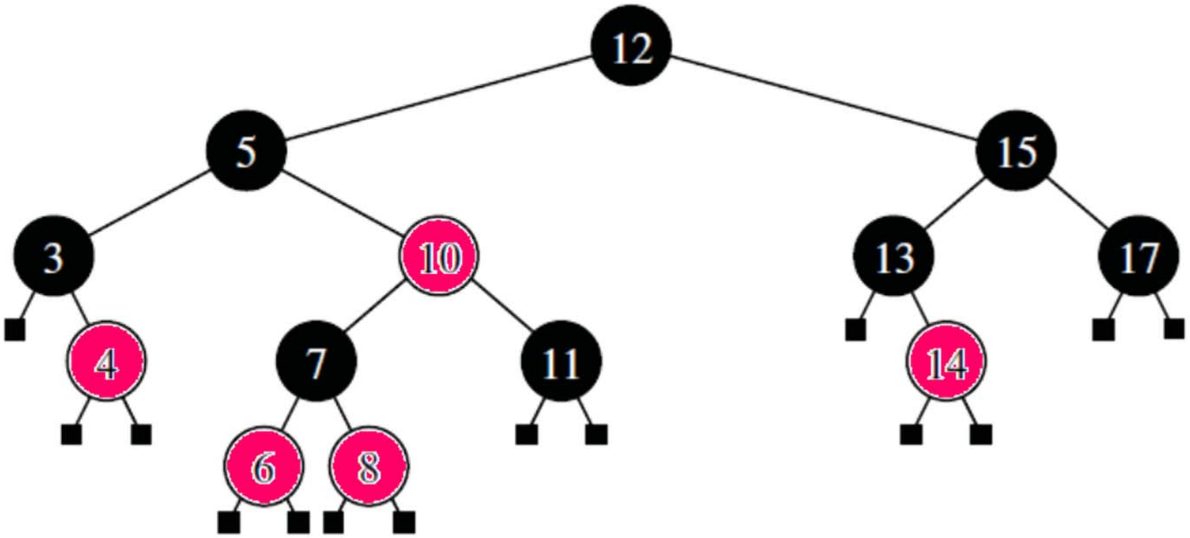
- Proposition
  - The height *h* of a (2,4)-tree storing *n* element is $O(\log n)$
- Justification
  - We will prove that $0.5 \log(n + 1) \leq h \leq \log(n + 1)$
  - The number of external nodes: at most 4^h; at least 2^h
    - At depth 1: at most 4 nodes; at least 2 nodes
    - At depth 2: at most 4^2 nodes; at least 2^2 nodes, …
  - A tree with *n* entries has $n + 1$ external nodes
    - Hence, $2^h \leq n + 1 \leq 4^h$
    - Taking logs: $h \leq \log(n + 1) \leq 2h$

Red-Black Trees

- A red-black tree is a binary tree with nodes colored in red or black such that
  - *Root property*: the root is black
  - *External property*: every external node is black
  - *Red property*: the children of a red node is black
  - *Depth property*: all external nodes have the same black depth
    - Black depth: the number of ancestors that are black

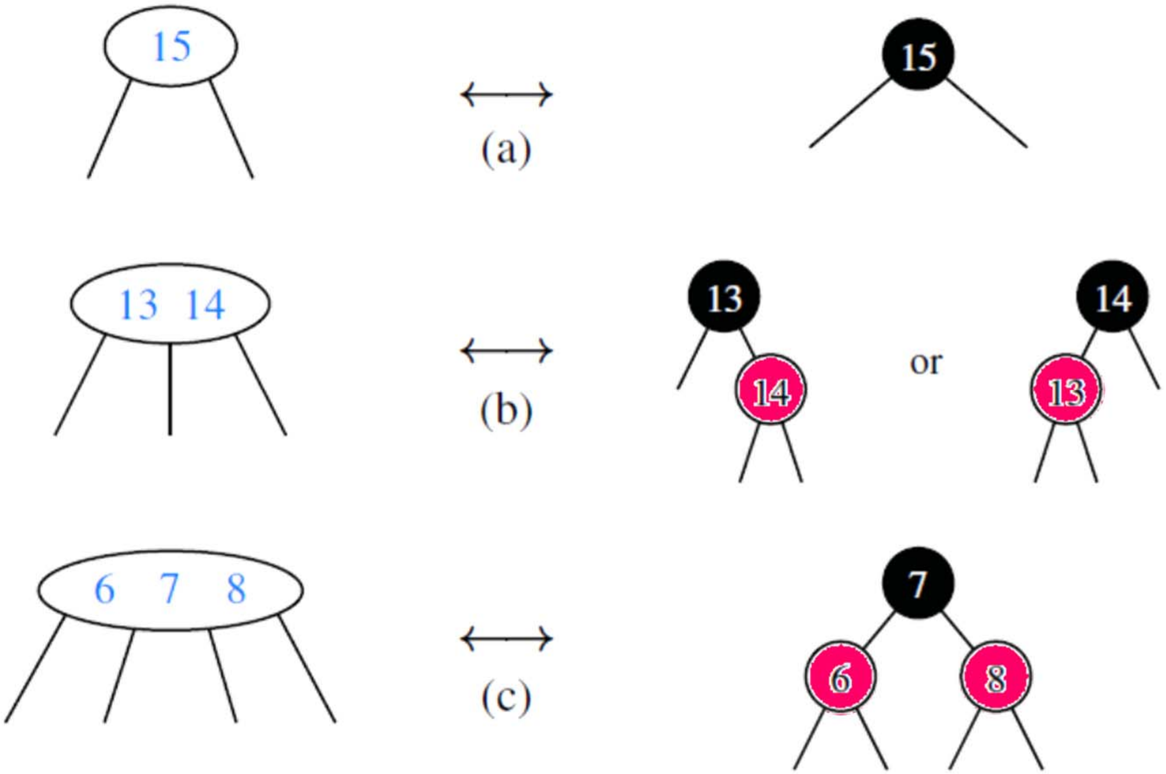
Red-Black Trees

- Example



Red-Black Trees

- Correspondence between (2,4)-trees and red-black trees



Red-Black Trees

- Searching in a red-black tree
  - The same as that for a standard binary tree

- Analogy to (2,4)-Trees
  - Split operation: recoloring
  - Fuse operation: recoloring
  - Transfer operation: trinode restructuring + recoloring
  - Orientation of 3 nodes: rotation
    - (b) in a previous slide

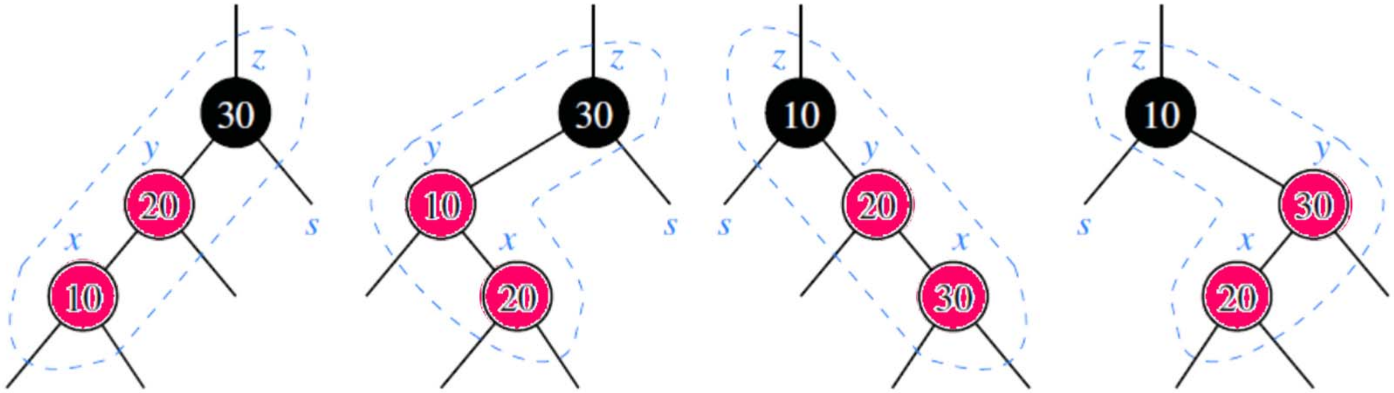
Red-Black Trees

- Insertion of x
  - If this is the first entry, x is the root and is black
  - Otherwise, we color x red

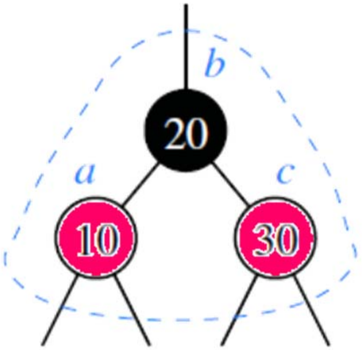
  - The root and the depth properties are preserved
  - The red property (*double-red* at node x) may be violated
    - In this case, its parent is red and its grand parent is black

Red-Black Trees (Insertion)

- Case 1: x’s uncle s is black

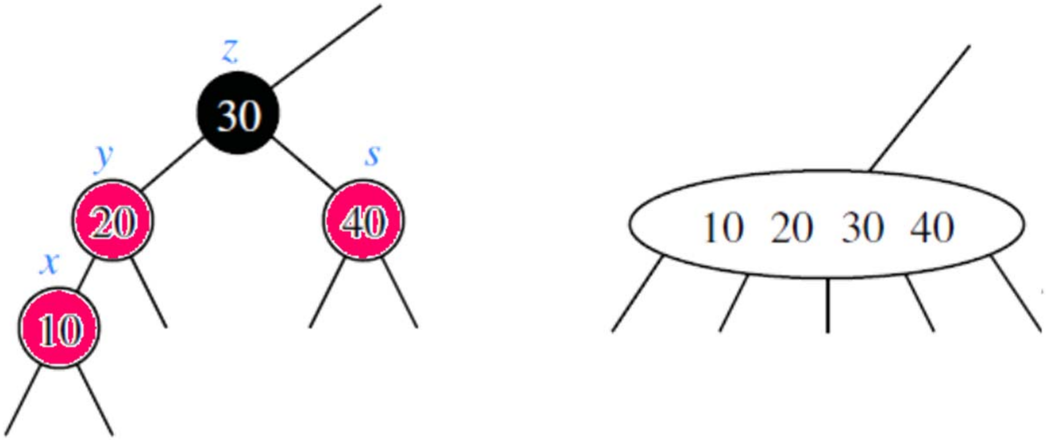


  - Malformed 4-node
  - Trinode restructuring on x and
  - Recoloring can fix the problem
    - b ⇒ black and a, c ⇒ red



Red-Black Trees (Insertion)

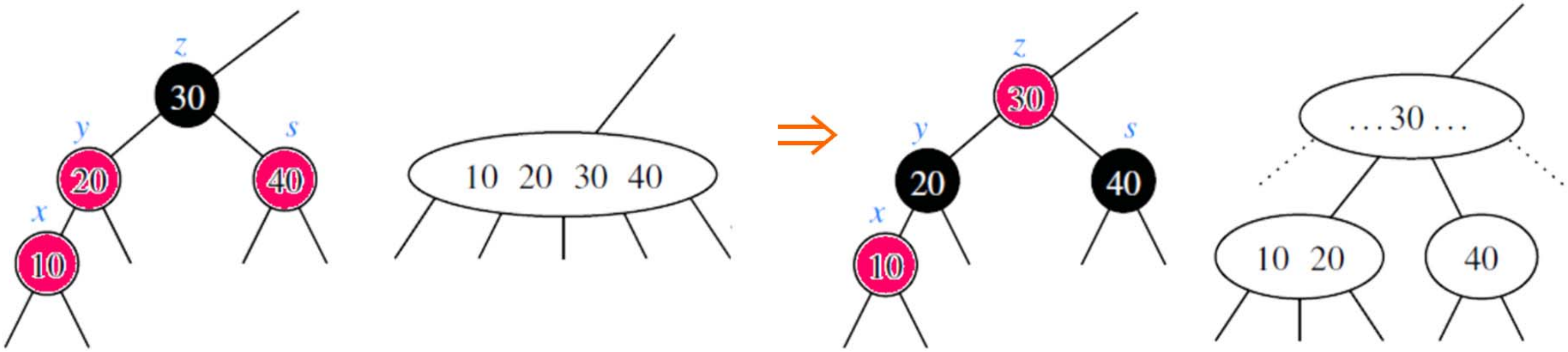
- Case 2: x’s uncle s is red



  - Overflow case

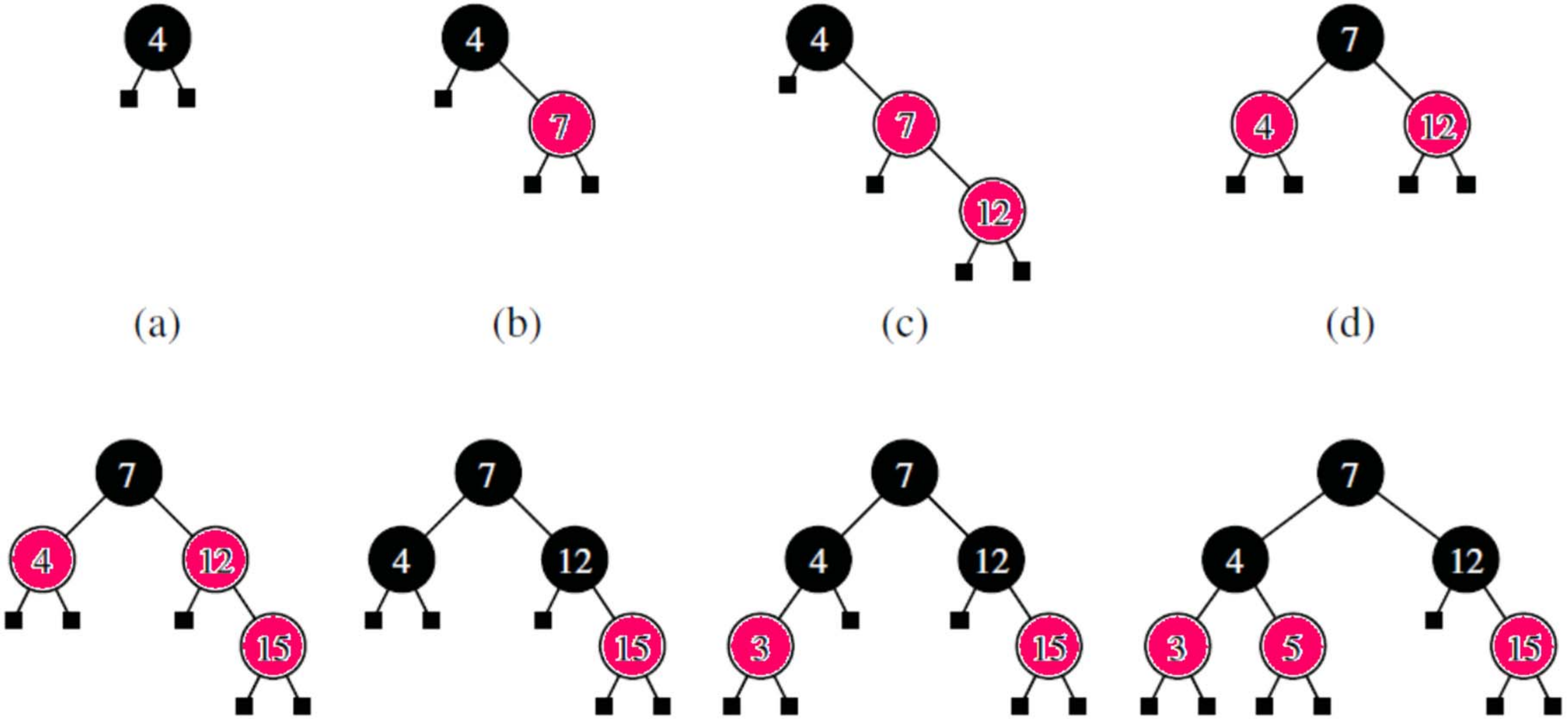
Red-Black Trees (Insertion)

- Case 2: x's uncle s is red (cont'd)



  - Recoloring: y, s ⇒ black; z ⇒ red unless it is the root
    - Black depth is unaffected
    - z may cause another double-red problem ⇒ recursively fix the double-red problem

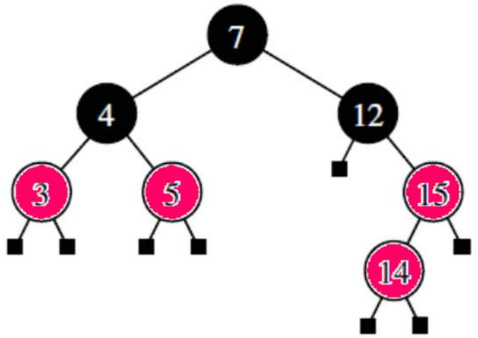
Red-Black Trees (Insertion)



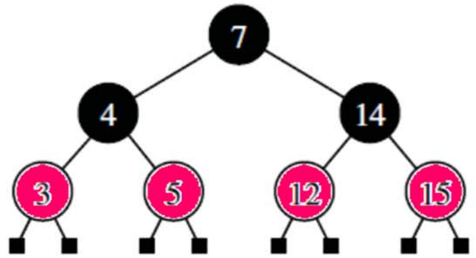
(a) (b) (c) (d)

(e) (f) (g) (h)

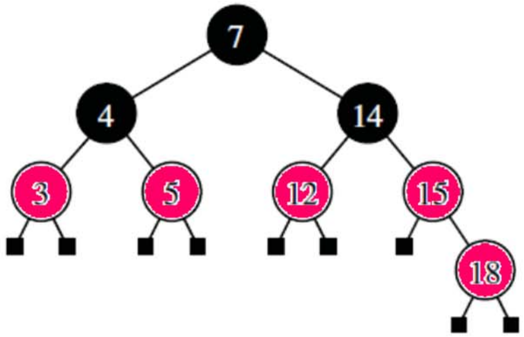
Red-Black Trees (Insertion)



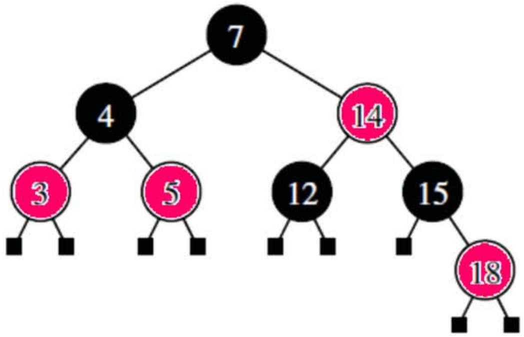
(i)



(j)

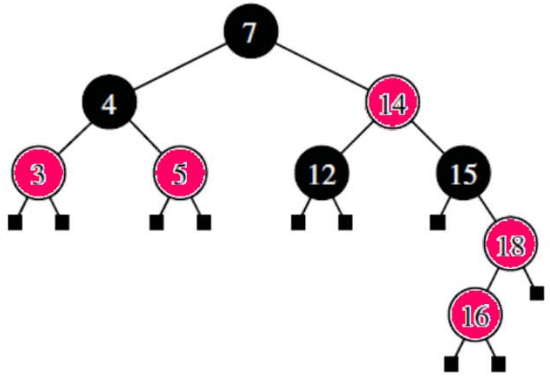


(k)

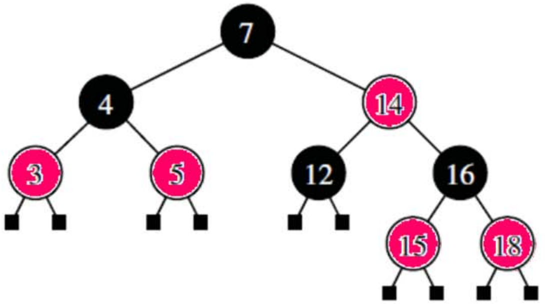


(l)

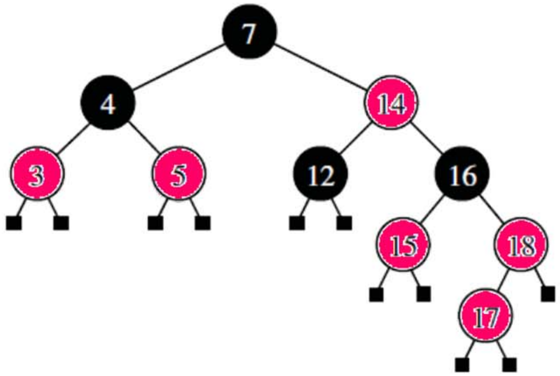
Red-Black Trees (Insertion)



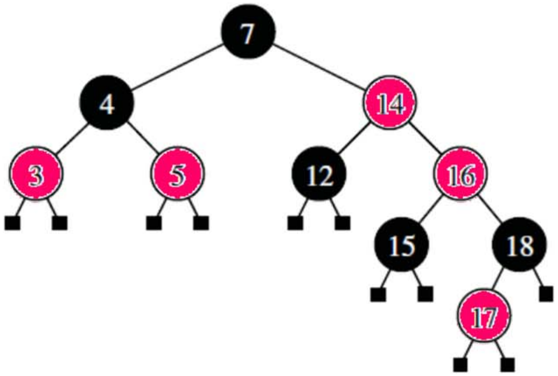
(m)



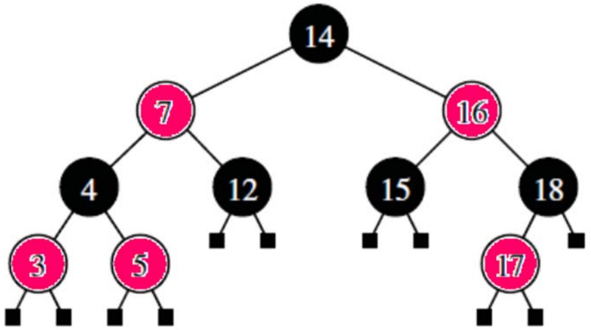
(n)



(o)



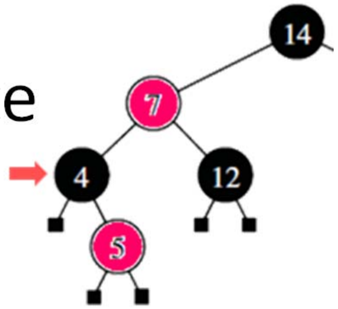
(p)



(q)

Red-Black Trees (Deletion)

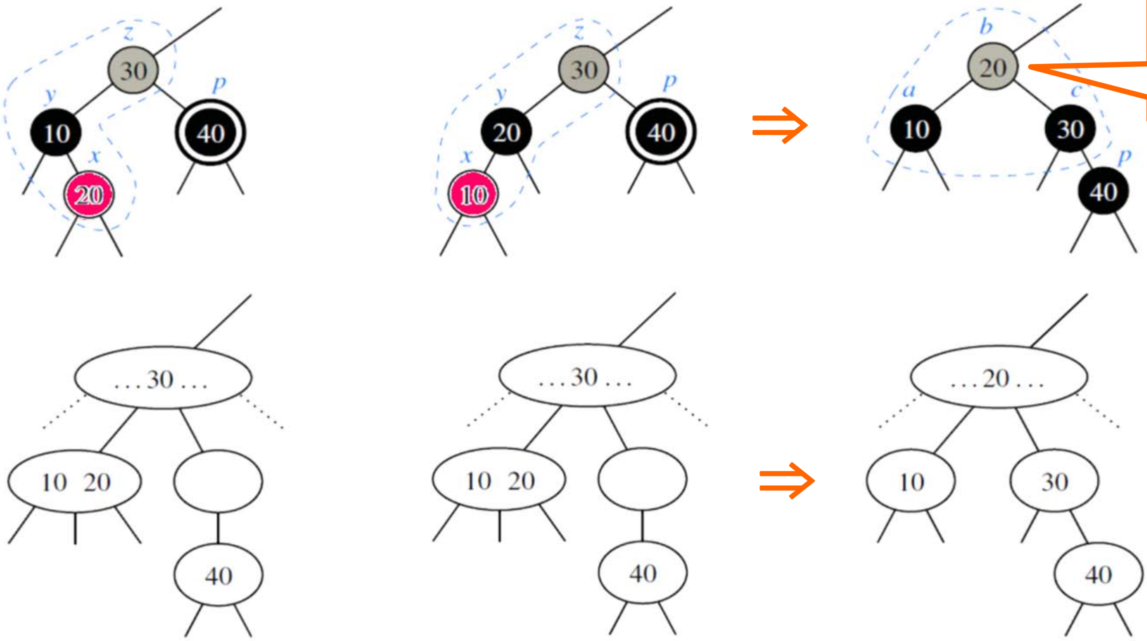
- Deletion of n
  - Delete a node like a binary search tree
    - If n has two internal children: swap it with its predecessor
  - If *n is red* ⇒ the depth and the red properties are maintained (shrinking of 4-node or 3-node)
  - If *n is black* and has one *red child* ⇒ promote the child and recolor it to black (3-node)



  - If *n is black* and both of its *children are black* (removal from a 2-node: underflow) ⇒ next slides

Red-Black Trees (Deletion)

- Case 1: p's sibling is black and has a red child x
  - *p* is the promoted child of n marked in double-black

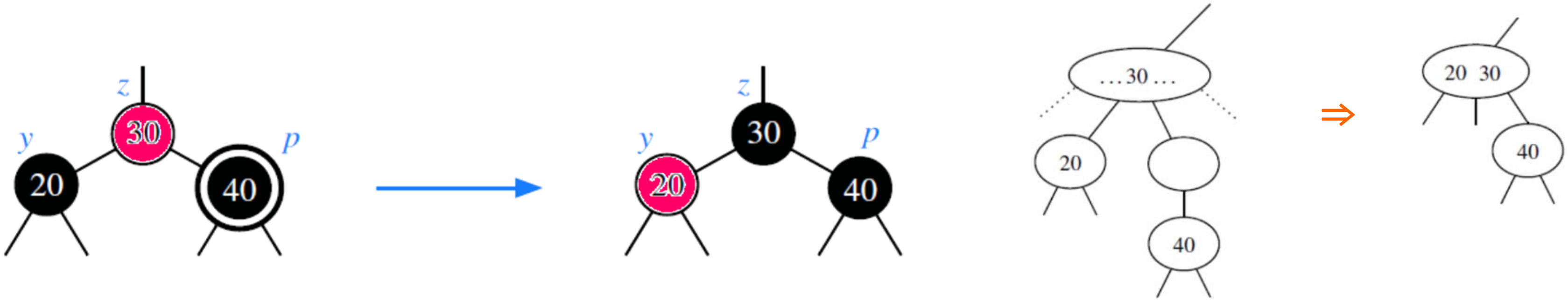


Keep the original color

- Trinode restructure on x (transfer of 2-4 tree)
  - b ⇒ z's previous color and a, c ⇒ black

Red-Black Trees (Deletion)

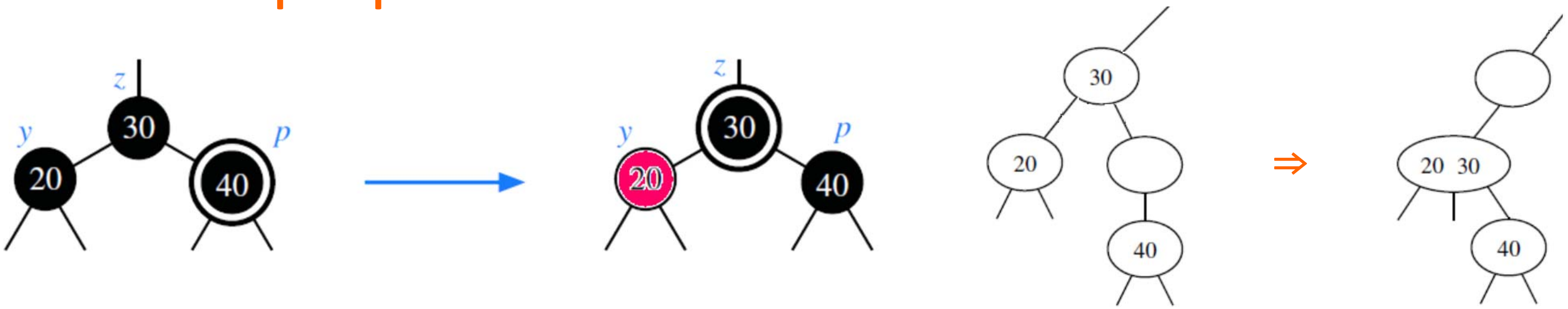
- Case 2: p’s sibling is black with two black children;
  p’s parent is red



  - Recoloring (fusion operation of (2,4)-trees)
    - y ⇒ red, z ⇒ black
  - The process ends here
    - y, z path: same black depth
    - p, z path: increased black depth (deficiency is fixed)

Red-Black Trees (Deletion)

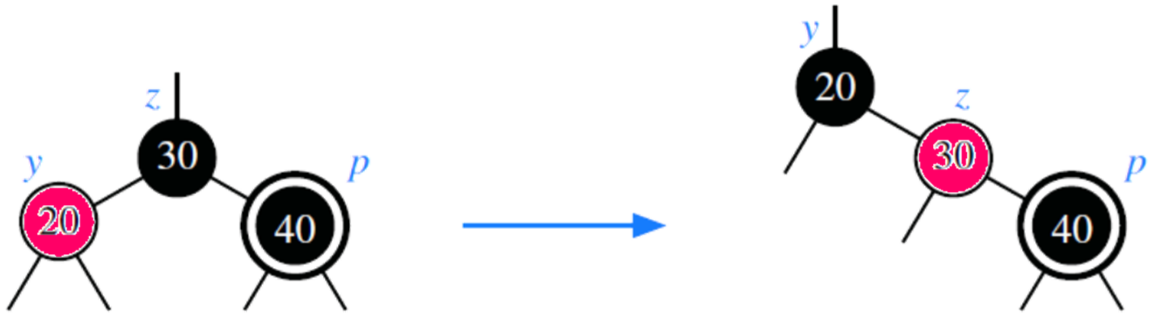
- Case 2’: p’s sibling is black with two black children;
  p’s parent is black



  - Recoloring (fusion operation of (2,4)-trees)
    - y ⇒ red, z ⇒ double-black
  - Cascading the problem upward
    - y, z path: same black depth
    - p, z path: same black depth (deficiency is NOT fixed)

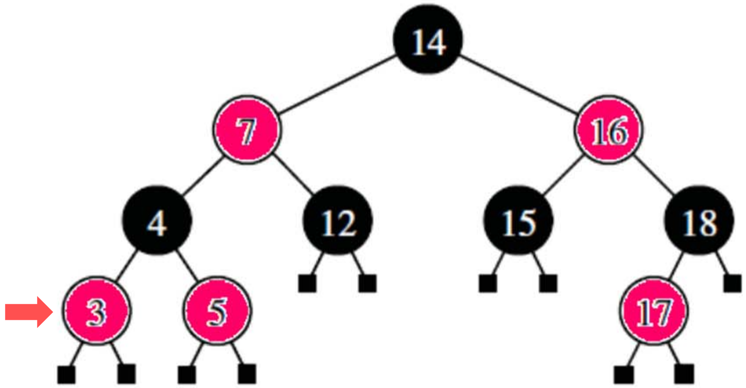
Red-Black Trees (Deletion)

- Case 3: p’s sibling is red

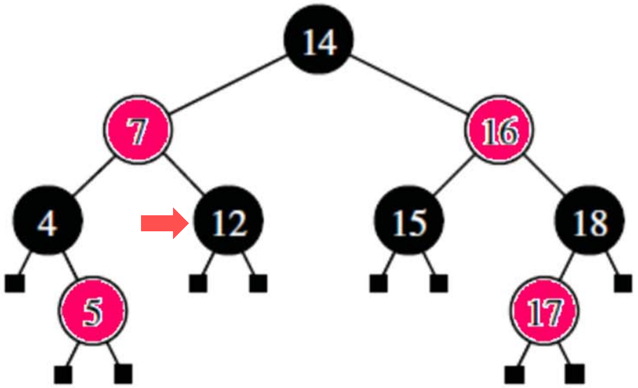


  - Rotate y and z (reorientation of a 3-node)
    - Recolor: y ⇒ black, z ⇒ red
  - After the rotation, p’s sibling is black

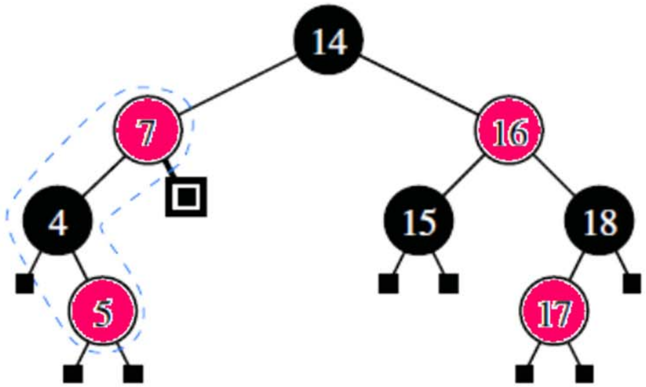
Red-Black Trees (Deletion)



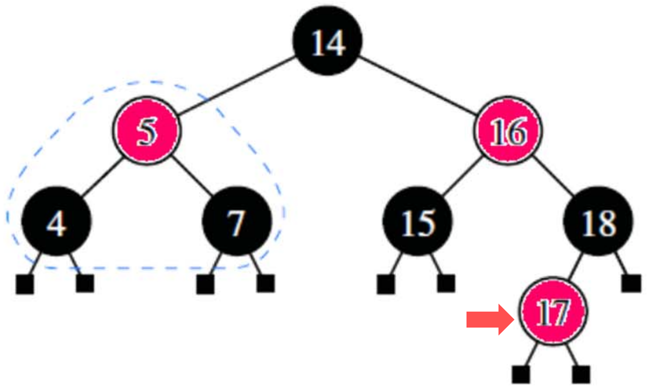
(a)



(b)

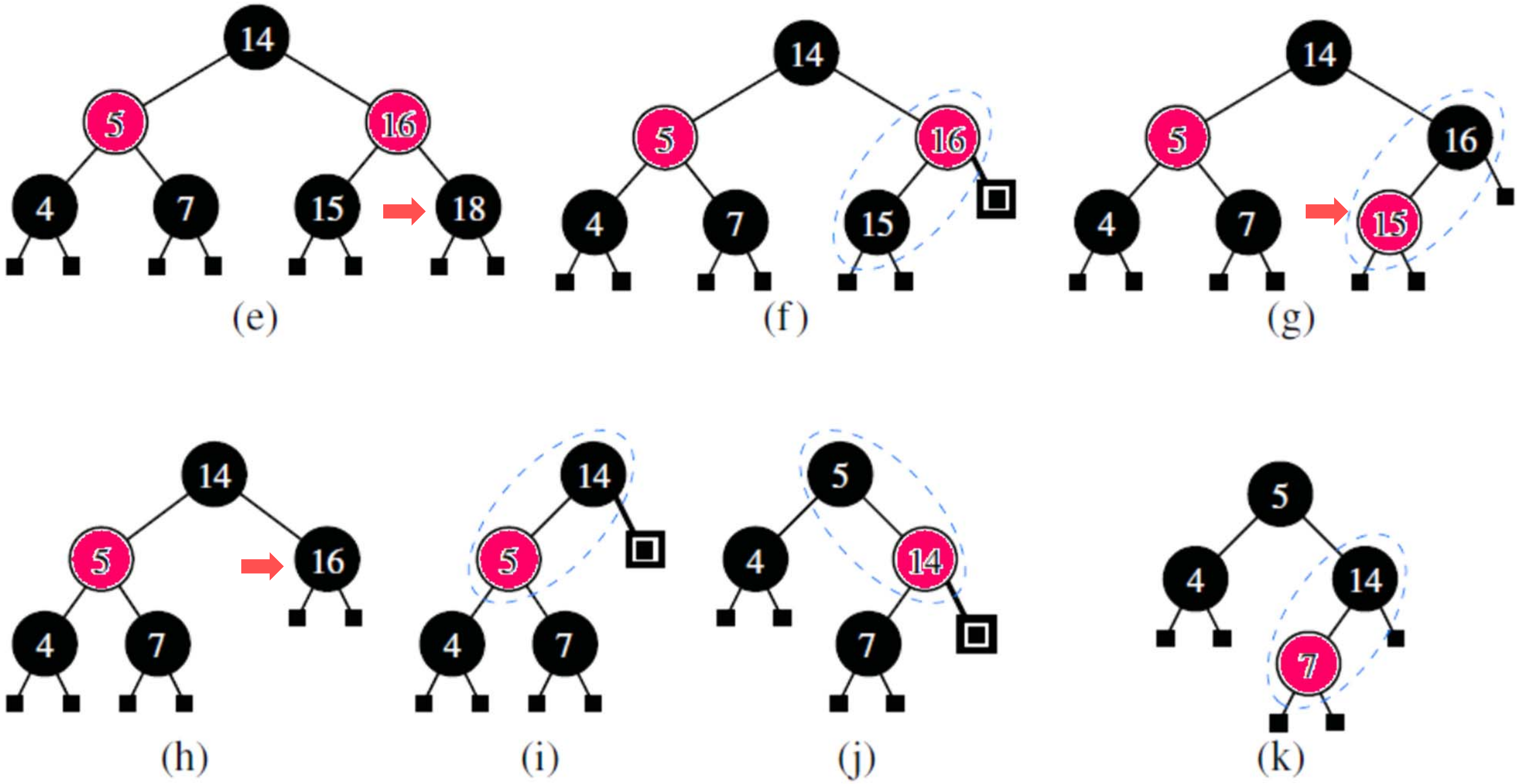


(c)



(d)

Red-Black Trees (Deletion)



(e) (f) (g)

(h) (i) (j) (k)

- Exercise:
  - Remove 15, 16, 18, 17, 12, 14, 7, 5, 3, 4 from (a)

Red-Black Trees

- Proposition
  - The height h of a red-black tree storing n entries is O(log n)

- Justification
  - Will prove that $\log(n + 1) \leq h \leq 2 \log(n + 1)$
  - Let d be the black depth of all external nodes; let h' be the height of the corresponding (2,4)-tree
    - $d = h' \leq \log(n + 1)$
    - By the red property $h \leq 2d$.
    - Hence, $h \leq 2 \log(n + 1)$
  - Skipping the other half: it comes from the properties of binary trees (when (2,4)-tree is a binary tree)

Assignment 9

- In this assignment, we will
    - Implement key methods of a heap
    - Implement key methods of a red-black tree
    - Find a shortest path using priority queue
    - Play the Pac-Man game

- Download hw9.zip
    - Implement all TODO lines
    - Zip the java files you modified and submit it

- Due date: 5/26/2022

Assignment 9

- Java files to update
  - RBTree.java: a Red-Black tree
  - RBTreeQueue.java: a priority queue using an RBTree
  - Path.java: a shorted path algorithm

Assignment 9

- Expected result from RBTree.java

```
java RBTree
0 ,
0 , 1*,
1 , 0*, 2*,
1 , 0 , 2 , 3*,
1 , 0 , 3 , 2*, 4*,
1 , 0 , 3*, 2 , 4 , 5*,
1 , 0 , 3*, 2 , 5 , 4*, 6*,
3 , 1*, 0 , 2 , 5*, 4 , 6 , 7*,
3 , 1*, 0 , 2 , 5*, 4 , 7 , 6*, 8*,
3 , 1 , 0 , 2 , 5 , 4 , 7*, 6 , 8 , 9*,
Success: increasing order
Success: decreasing order
Success: random order
```

Assignment 9

- When you are done, enjoy the Pac-Man game