بسم الله الرحمن الرحیم

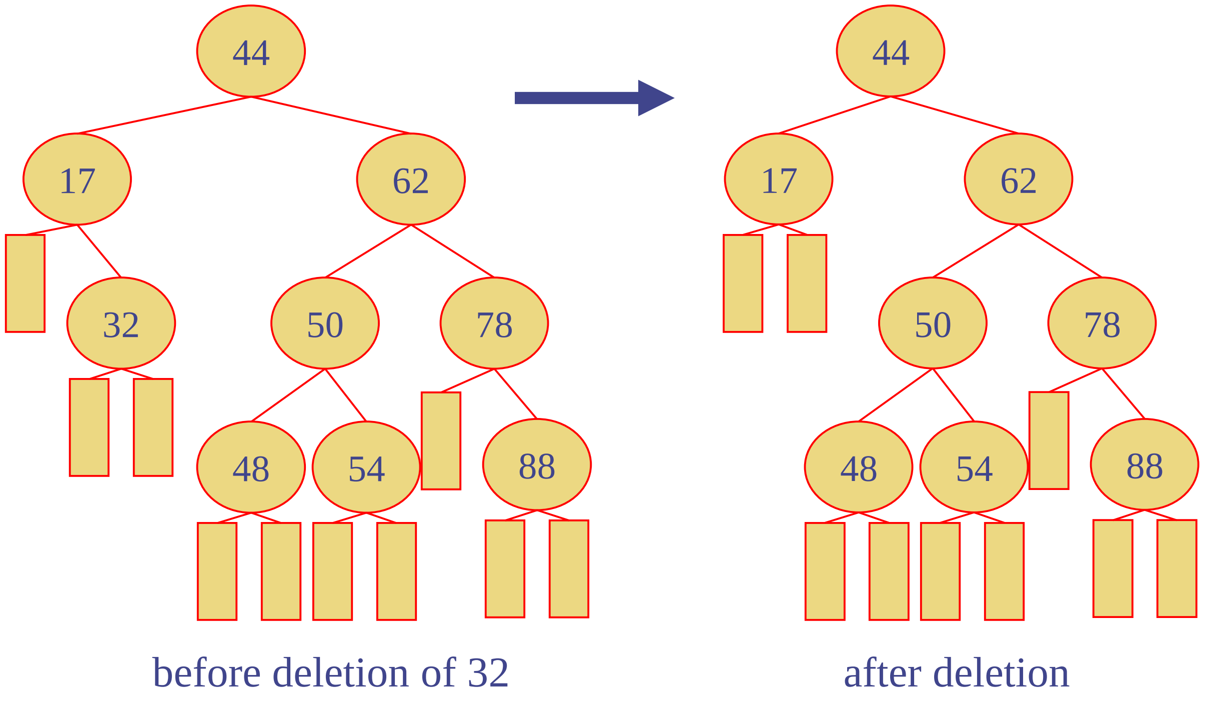
ساختمان‌های داده

جلسه ۱۹

مجتبی خلیلی
دانشکده برق و کامپیوتر
دانشگاه صنعتی اصفهان

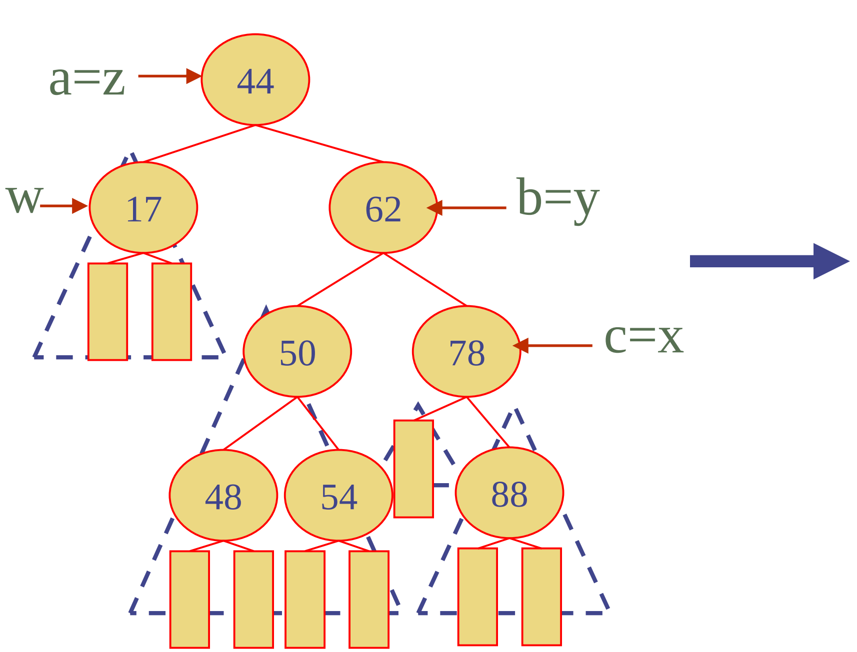
Removal

- Removal begins as in a binary search tree, which means the node removed (after copying the in-order successor) will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



Rebalancing after a Removal

- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height
- We perform restructure(x) to restore balance at z



- What happens if z is an internal node, not the root?
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

AVL Tree Performance

- a single restructure takes O(1) time
  - using a linked-structure binary tree
- find takes O(log n) time
  - height of tree is O(log n), no restructures needed
- put takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- erase takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Recall: Rebalancing Needed

IUT-ECE

How should we do this?

- (1) Take some examples
- (2) Find difference cases
- (3) Make each sub-algorithm for each case
- (4) Make an entire algorithm
- (5) Run it with some inputs
- (6) Find out it is not working perfectly, and say "What the hell is this?" "How should I do?"

Lessons

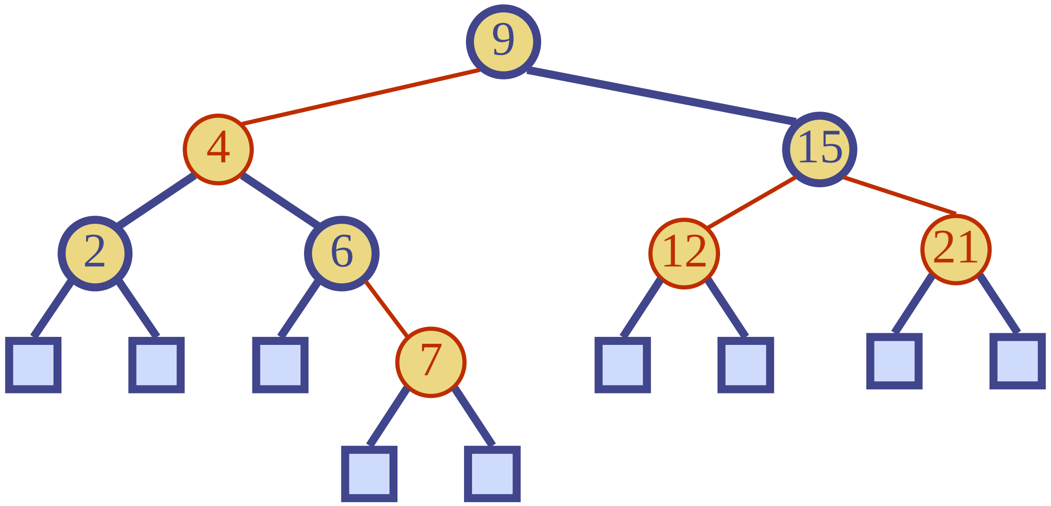
- Sometimes, we need to do case-by-case handling to complete the algorithm
- People often rely on "Half-assed algorithm design first " and "Complete it using example inputs". Not recommended.
    - Same as "Roughly make the code, and debug it later". Bad coding behavior

Red-Black Trees

Red-Black Trees

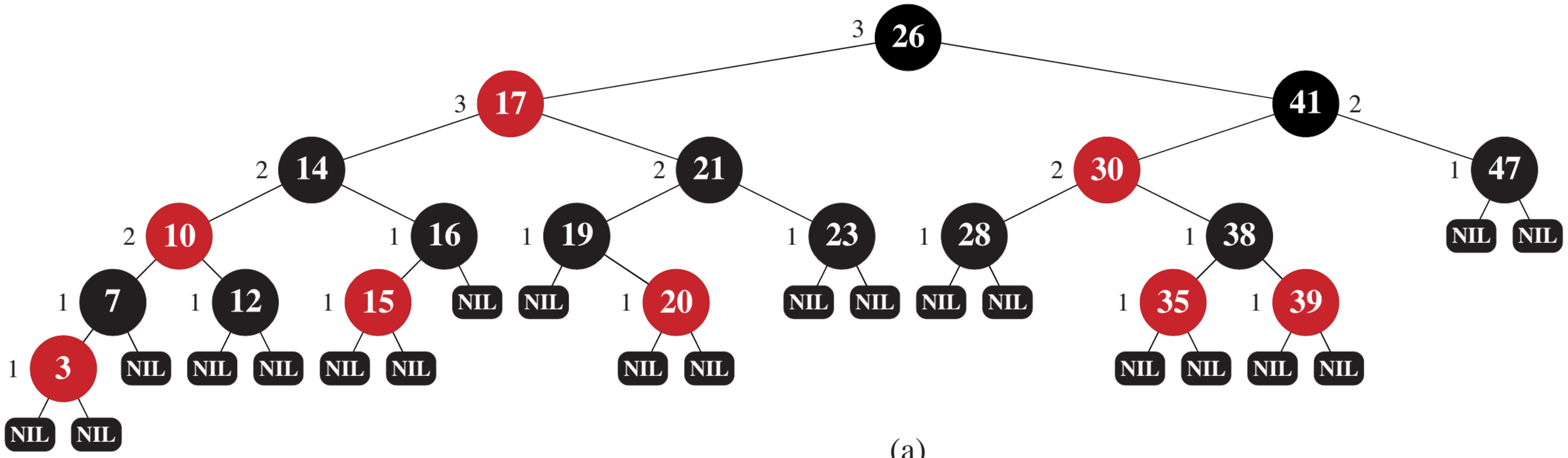
IUT-ECE

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black (red rule)
  - Depth Property: all the leaves have the same black depth (path rule)
  - (Question) How is balancing enforced here?



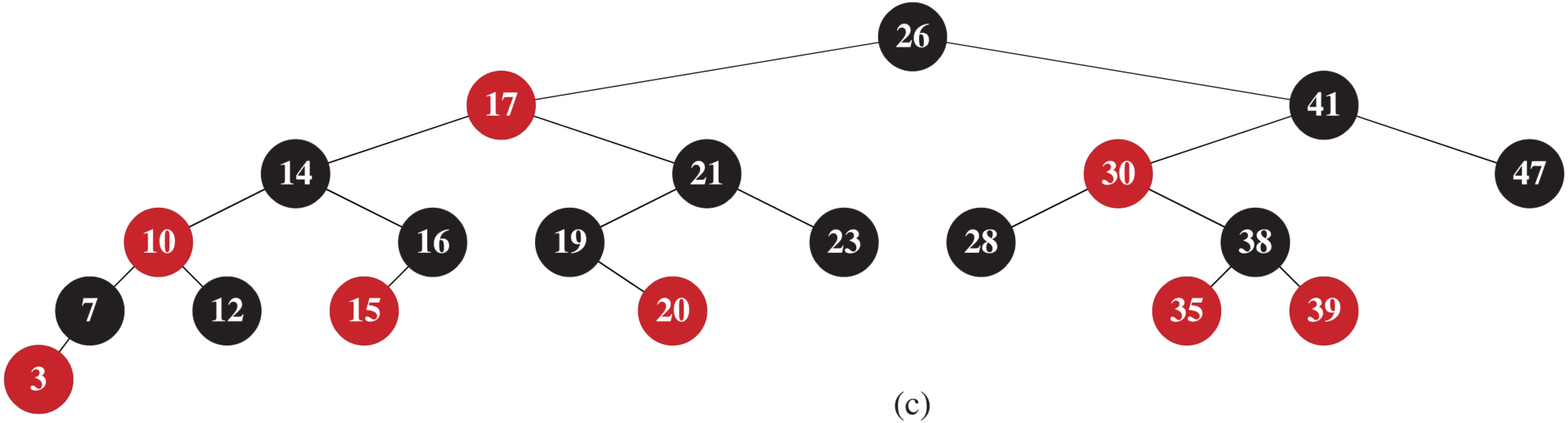
Red Black Tree

IUT-ECE



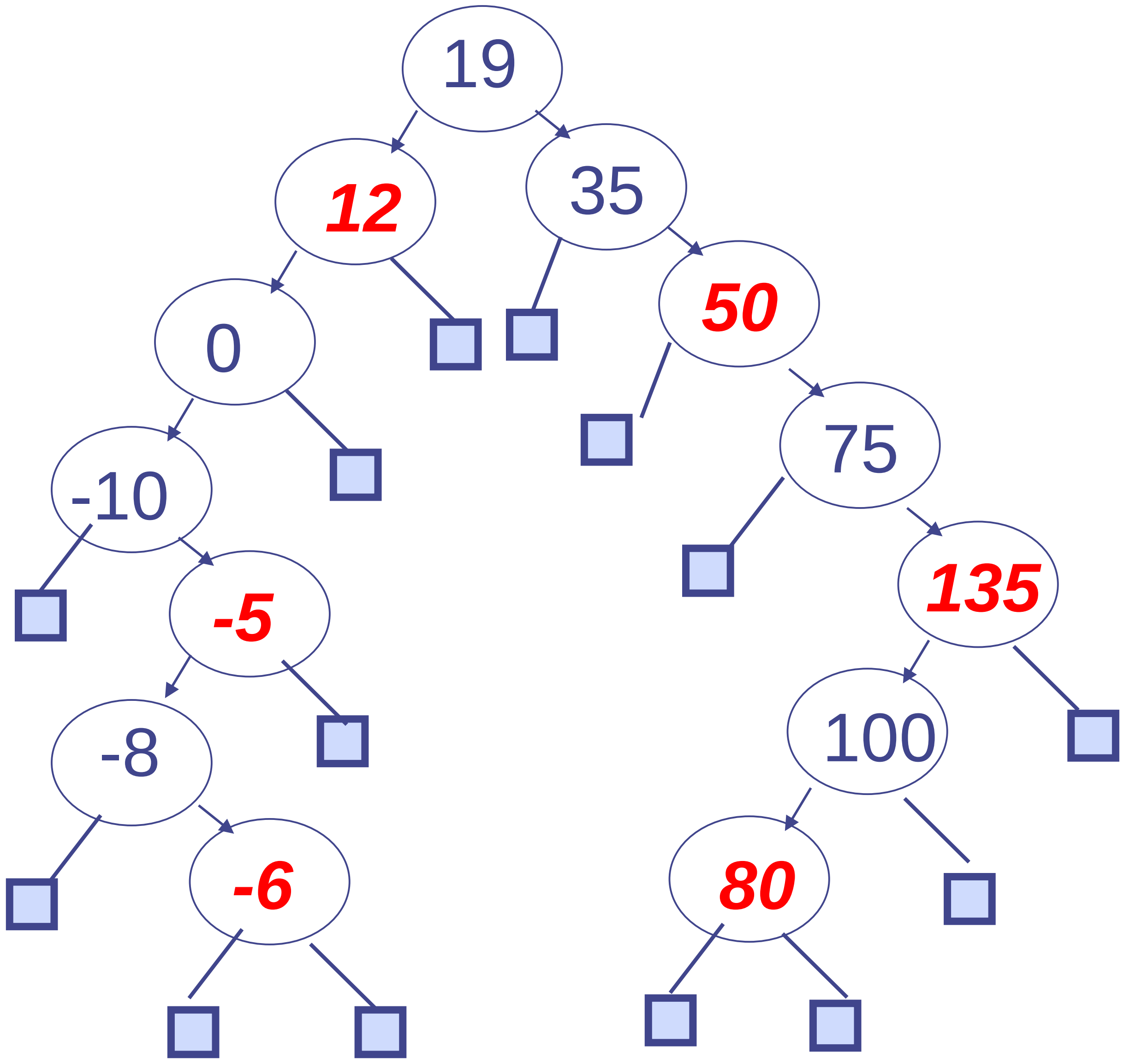
(a)

Red Black Tree

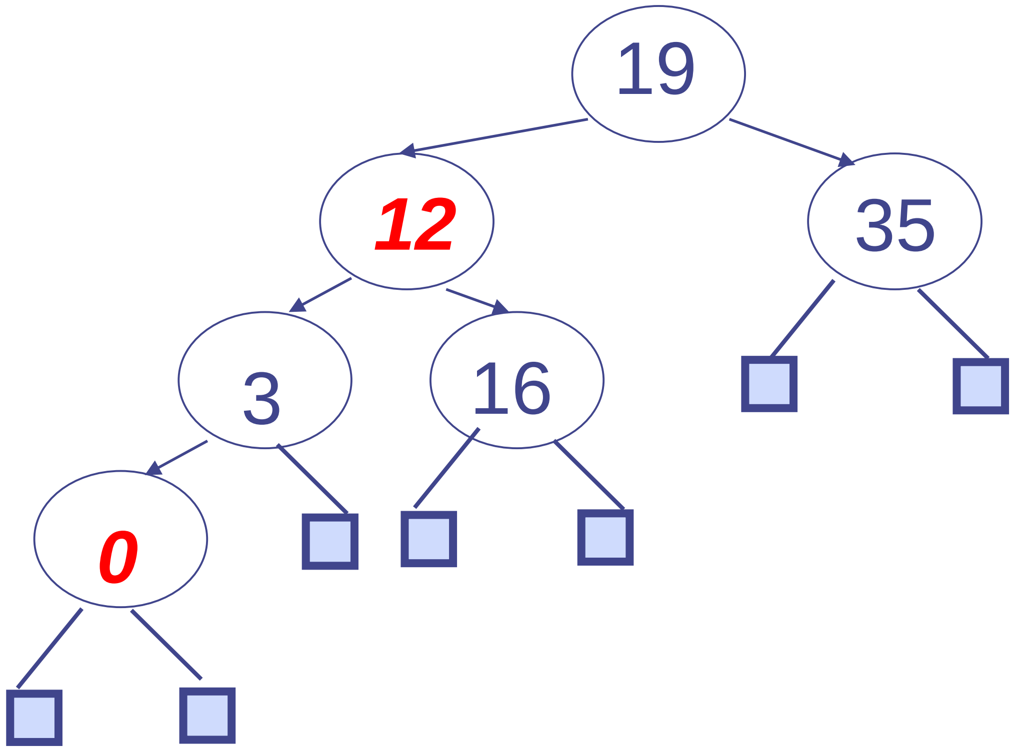


(c)

Red Black Tree?



Red Back Tree?



What if we attach a child to node 0?

Implications

- Root Property: the root is black
- External Property: every leaf is black
- Internal Property: the children of a red node are black (red rule)
- Depth Property: all the leaves have the same black depth (path rule)

- 1. If a red node has any children, it must have two children and they must be black
  - Why? Depth property

- 2. If a black node has only one “real” child then it must be a ”last” red node
  - If the child is black?
  - If the child is not the last red?

- (Question) How is balancing enforced in R-B tree?

Intuition about “rough balancing”

- The longest path <= 2 * the shortest path
  - Rough balancing → guarantees *log(n)* height

- Why?
  - From “red rule” and “path rule”
    shortest path = only black nodes
    longest path = inserting a red node between two black nodes

Root Property: the root is black
External Property: every leaf is black
Internal Property: the children of a red node are black (red rule)
Depth Property: all the leaves have the same black depth (path rule)

Height of a Red-Black Tree

IUT-ECE

- Theorem: A red-black tree storing n entries has height $O(\log n)$

  Proof:

  - Omitted

- The search algorithm for a binary search tree is the same as that for a binary search tree

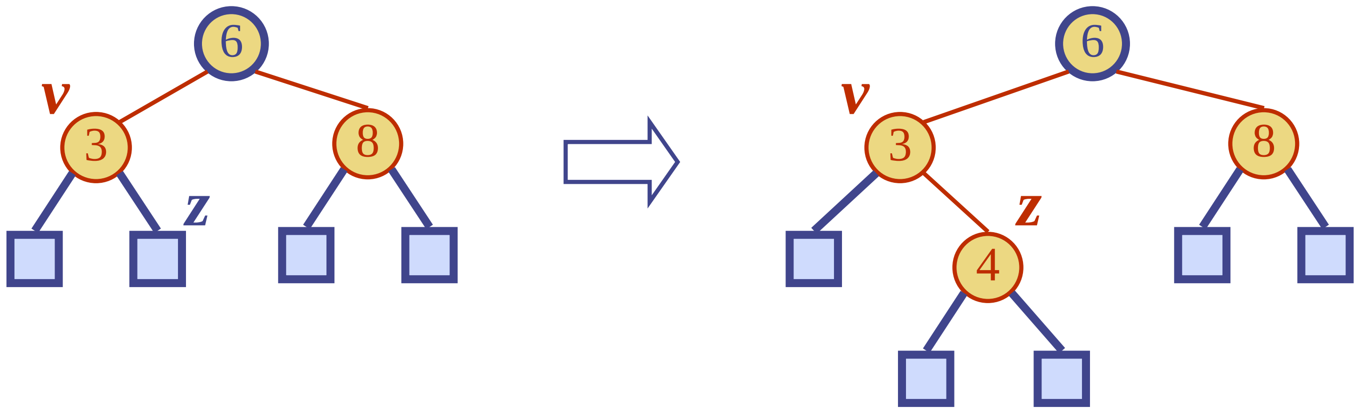
- By the above theorem, searching in a red-black tree takes $O(\log n)$ time

Insertion

Insertion

IUT-ECE

- To perform operation put(k, o), we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root
  - We preserve the root, external, and depth properties
  - If the parent v of z is black, we also preserve the internal property and we are done
  - Else (v is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
  - Goal: Removing double read without breaking the depth property
- Example where the insertion of 4 causes a double red:

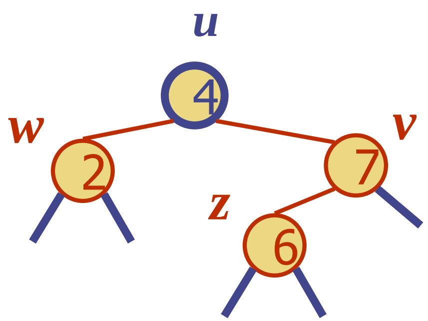


Remedying a Double Red

- Consider a double red with child z and parent v, and let w be the sibling of v

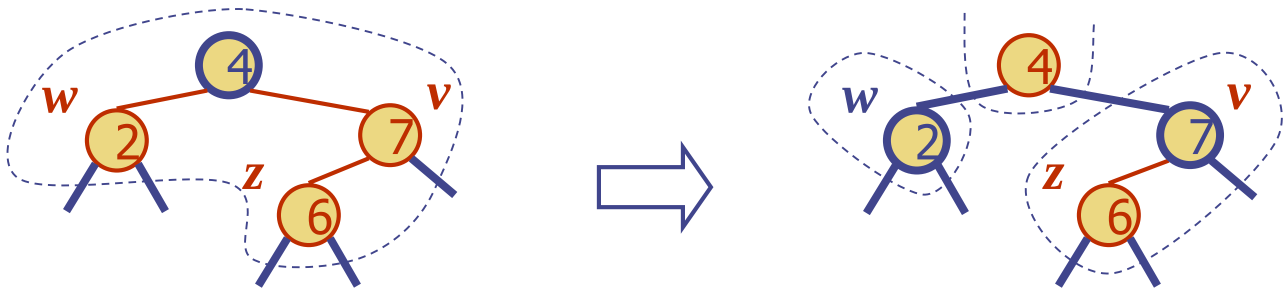
Case 1: w is red

- Recoloring: need recoloring



Recoloring

- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent ***v*** and its sibling ***w*** become black and the grandparent ***u*** becomes red, unless it is the root
- The double red violation may propagate to the grandparent ***u***

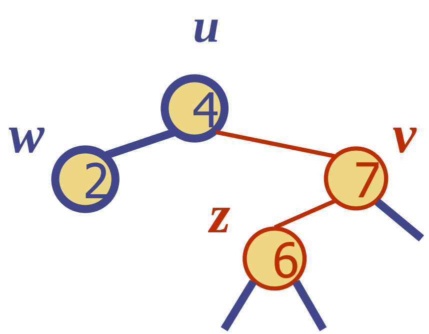


Remedying a Double Red

- Consider a double red with child z and parent v, and let w be the sibling of v

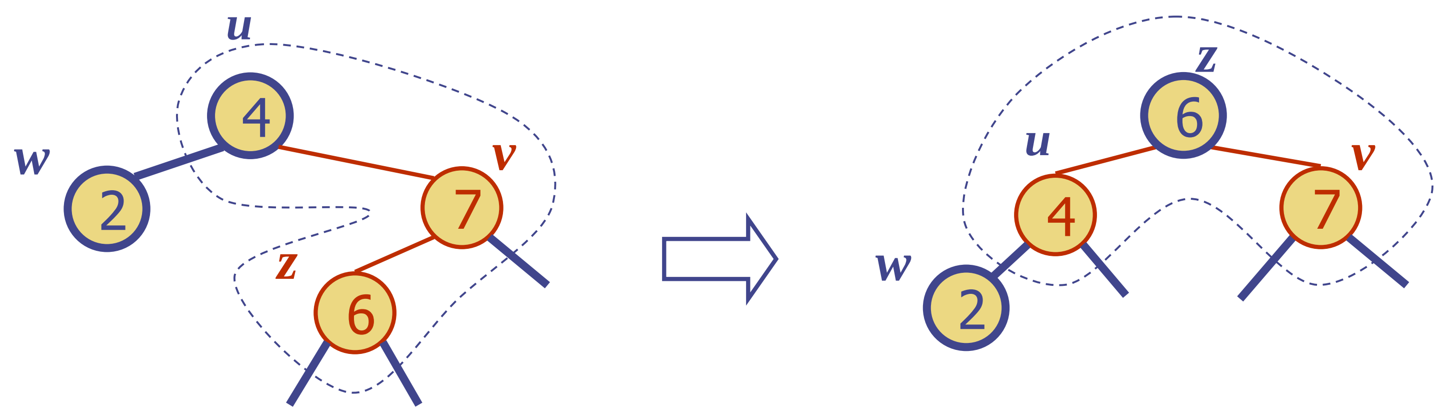
Case 2: w is black

- Restructuring: need rotation and recoloring



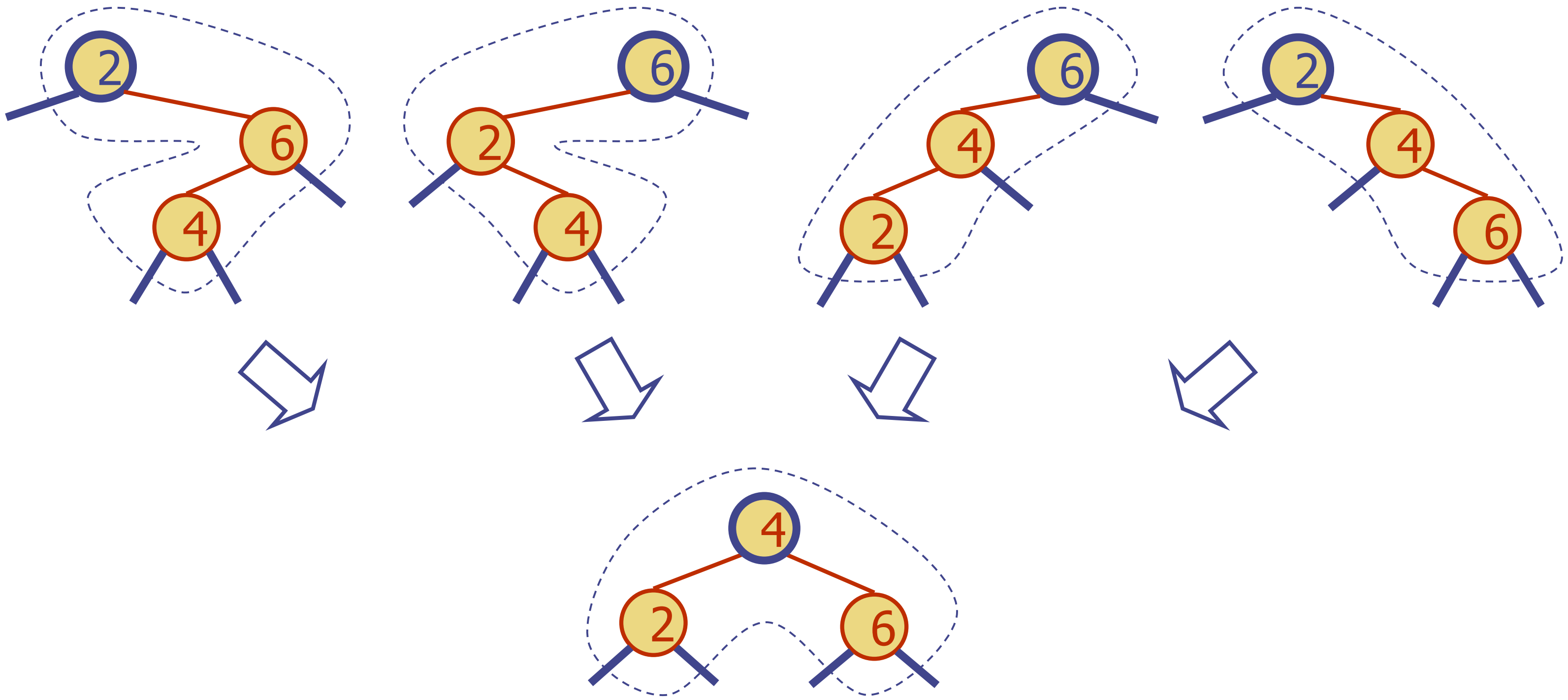
Restructuring

- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- The internal property is restored and the other properties are preserved

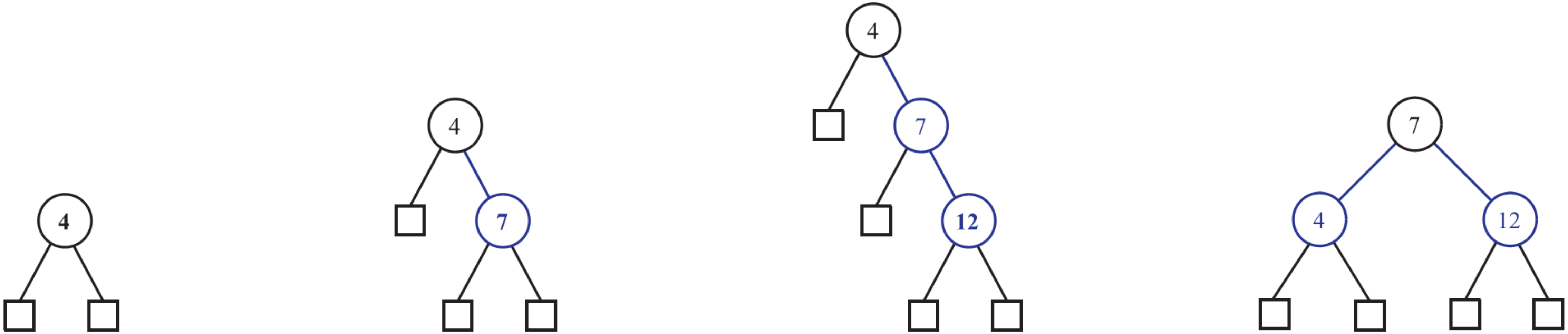


Restructuring (cont.)

- There are four restructuring configurations depending on whether the double red nodes are left or right children

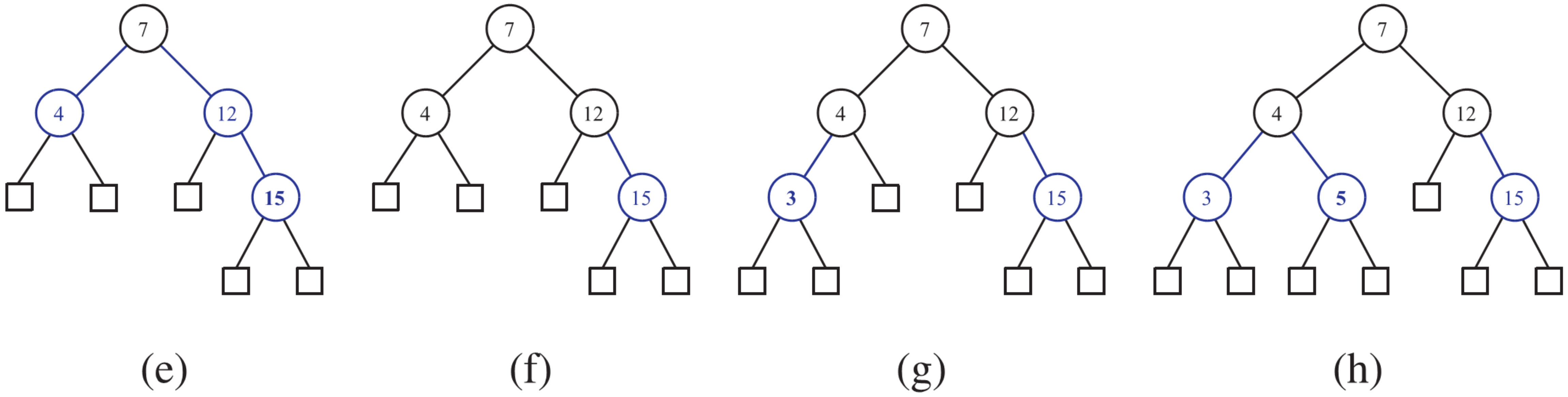


Example

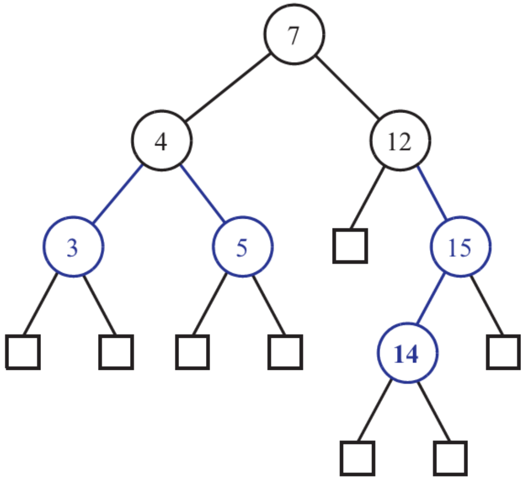


(a) (b) (c) (d)

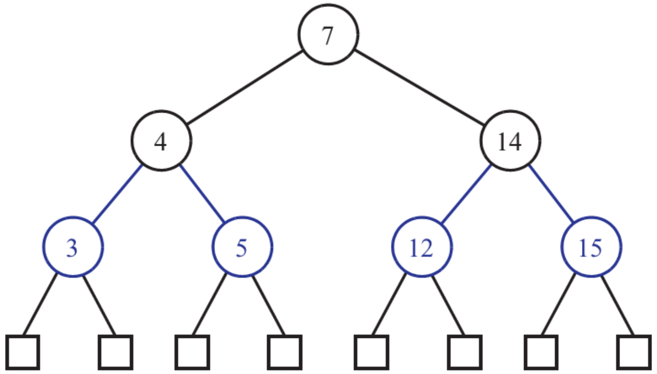
Example



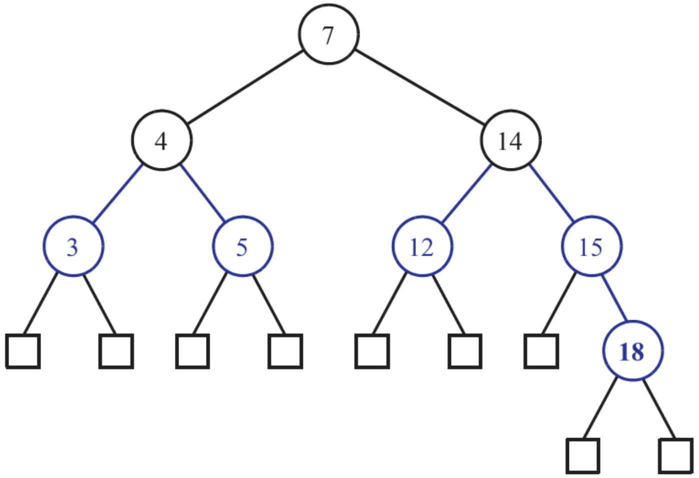
Example



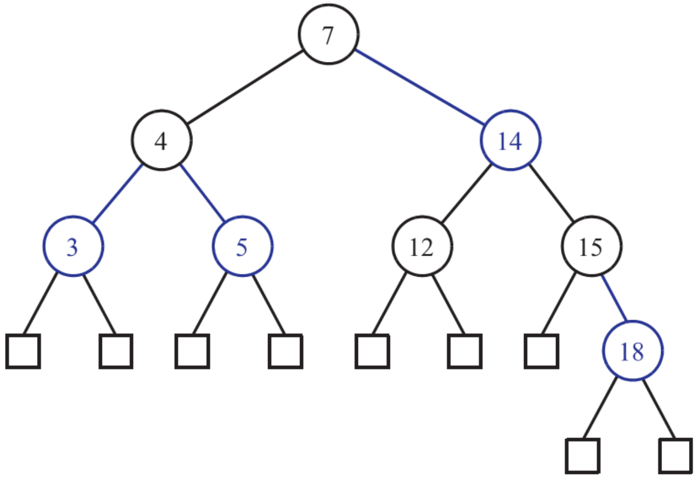
(i)



(j)

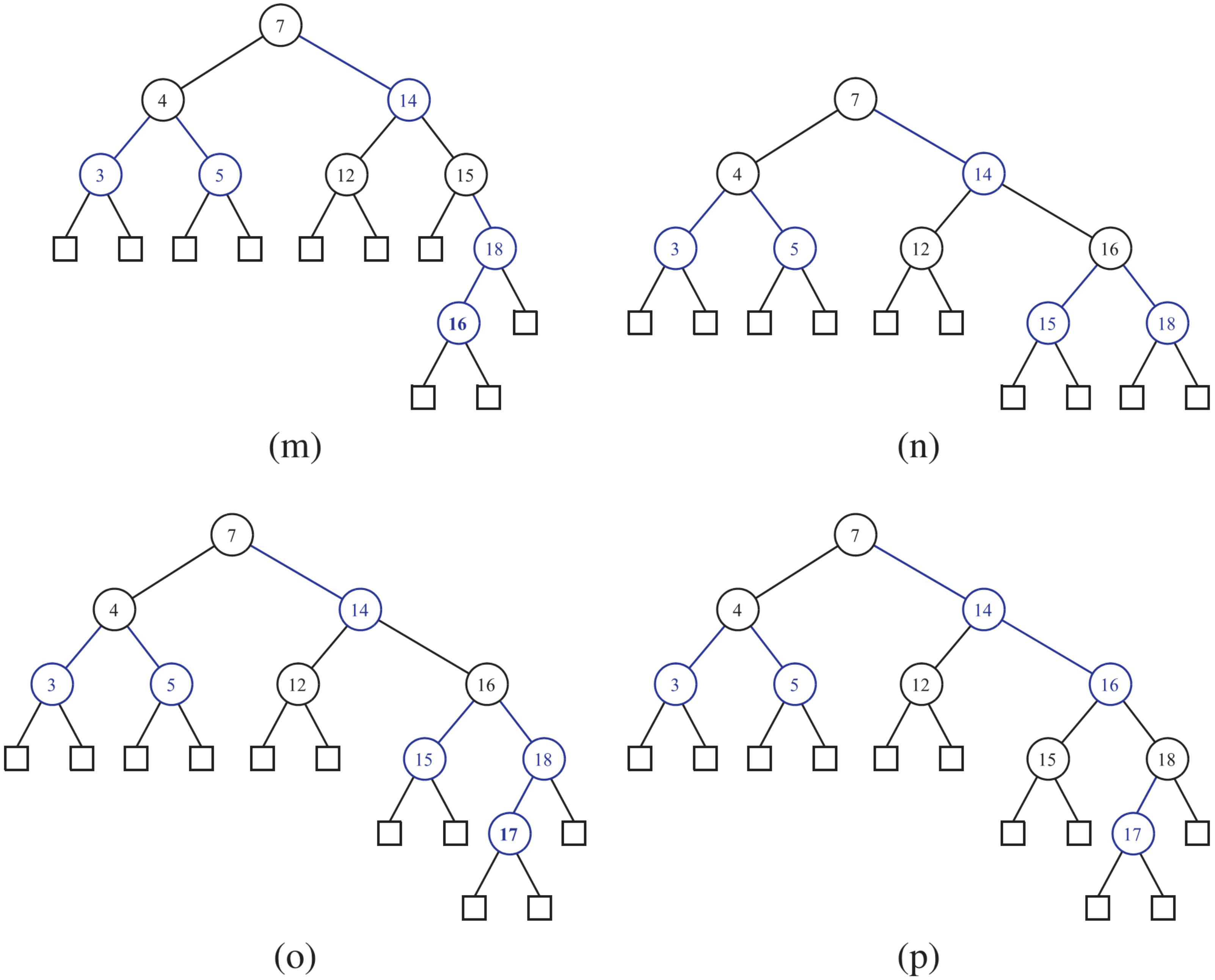


(k)



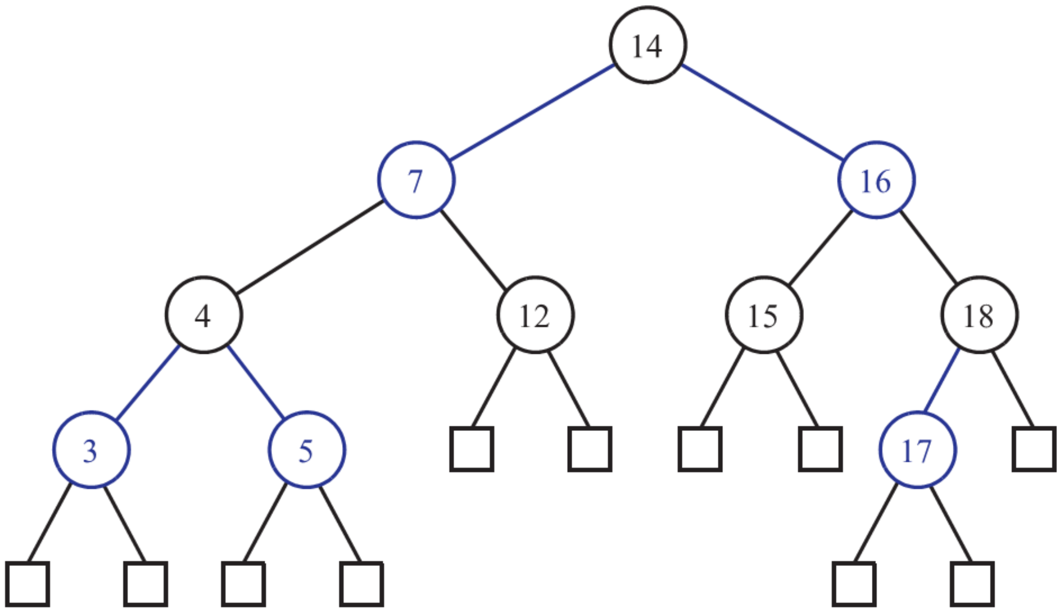
(l)

Example



7
4
14
3
5
12
15
18
16
(m)
7
4
14
3
5
12
16
15
18
(n)
7
4
14
3
5
12
16
15
18
17
(o)
7
4
14
3
5
12
16
15
18
17
(p)

Example



(q)

Analysis of Insertion

Algorithm ***put***(k, o)

1. We search for key k to locate the insertion node z

2. We add the new entry (k, o) at node z and color z red

3. **while** ***doubleRed***(z)

    if ***isBlack***(***sibling***(***parent***(z))))

        $z \leftarrow$ ***restructure***(z)

        return

    else { ***sibling***(***parent***(z)) is red }

        $z \leftarrow$ ***recolor***(z)

- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because we perform
  - $O(\log n)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$ time
- Thus, an insertion in a red-black tree takes $O(\log n)$ time