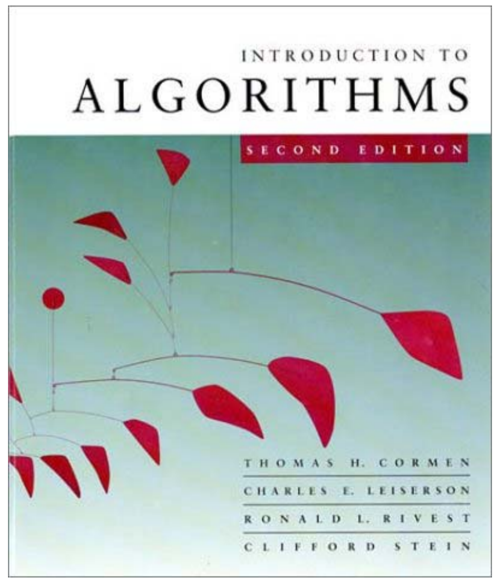
Introduction to Algorithms

6.046J/18.401J

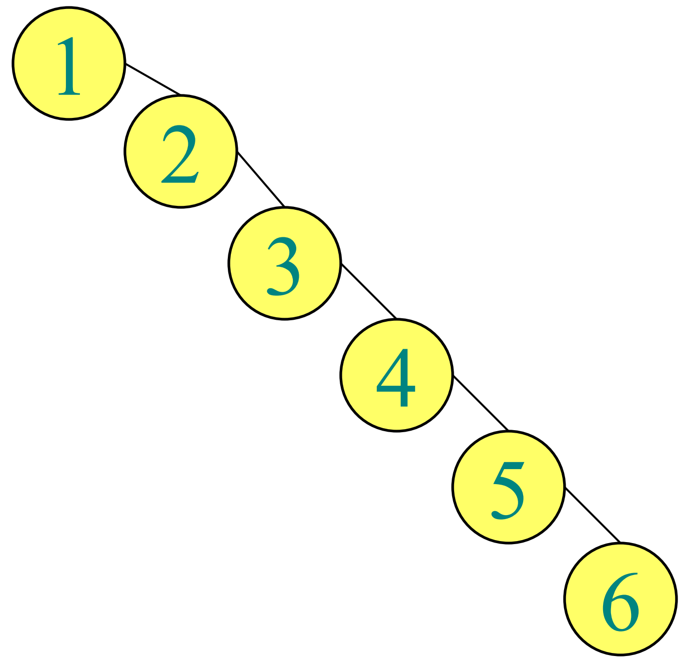


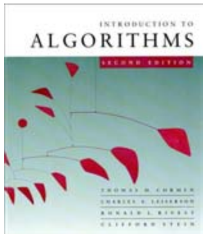
Lecture 9

Prof. Piotr Indyk

Today

- Balanced search trees, or how to avoid this even in the worst case



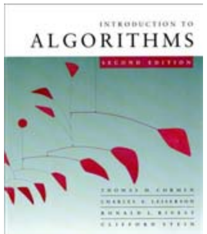


Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of n items.

Examples:

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees



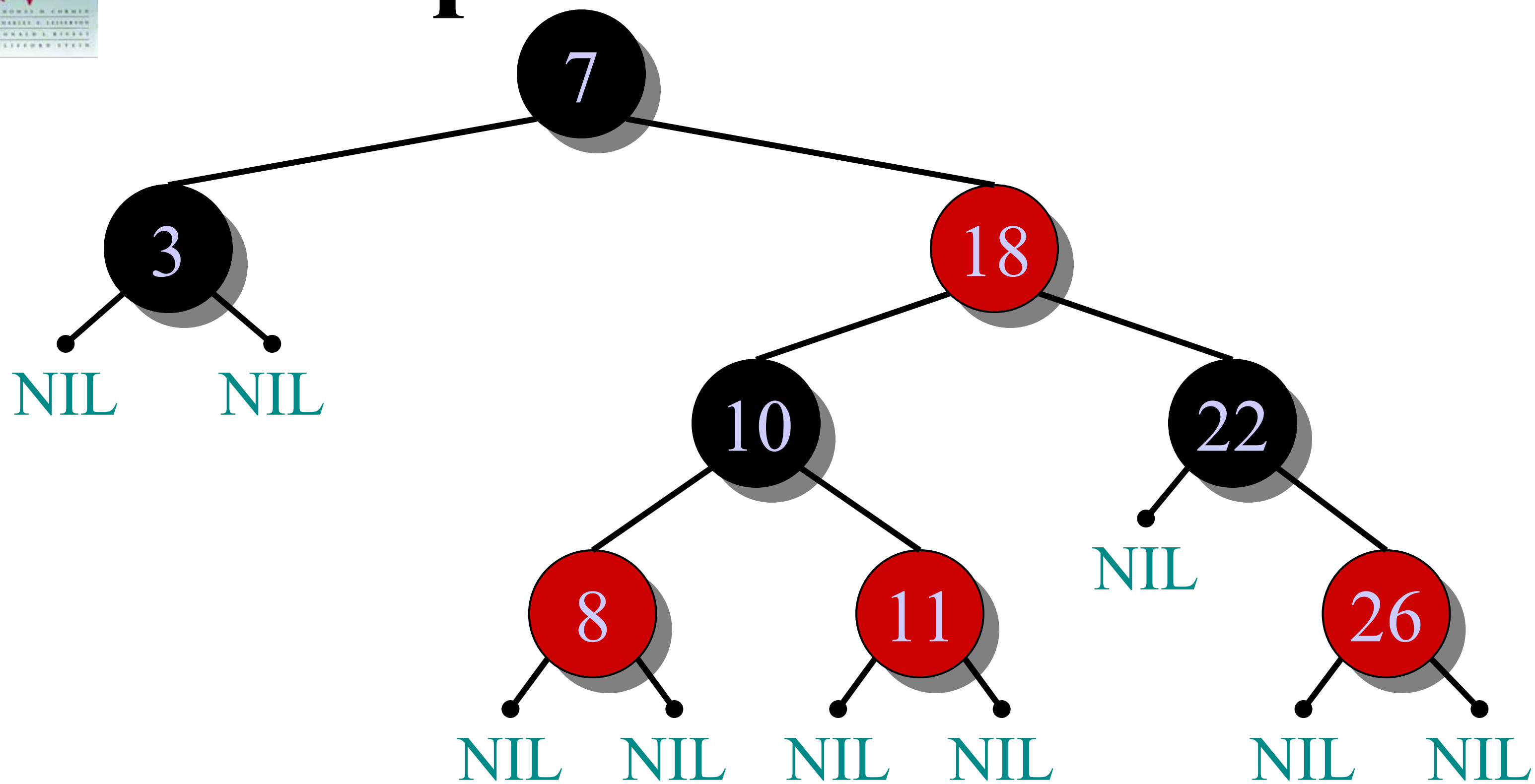
Red-black trees

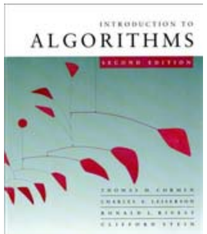
BSTs with an extra one-bit color field in each node.

Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes.

Example of a red-black tree

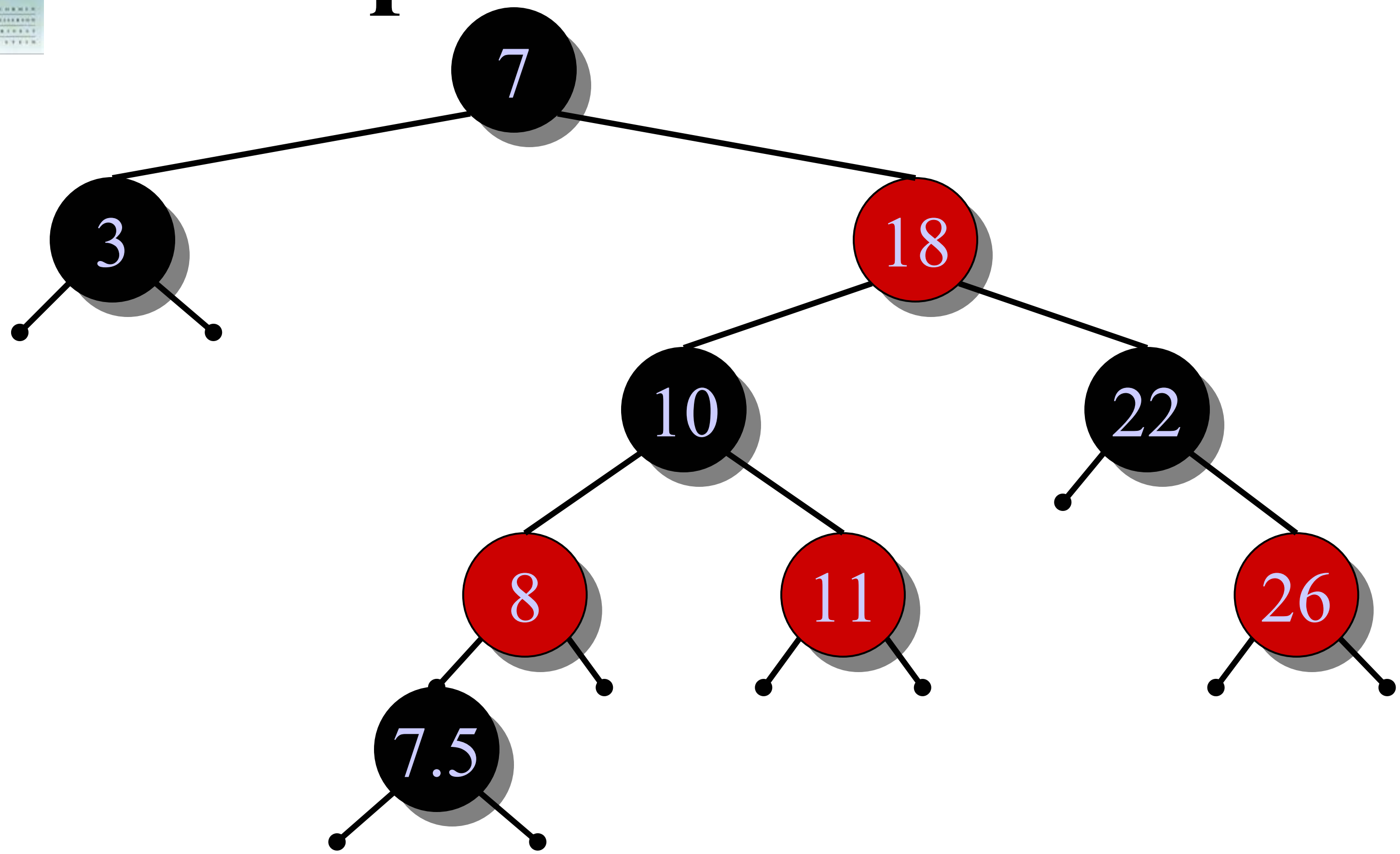




Use of red-black trees

- What properties would we like to prove about red-black trees ?
  – They **always** have O(log n) height
  – There is an O(log n)–time insertion procedure which preserves the red-black properties
- Is it true that, after we add a new element to a tree (as in the previous lecture), we can always recolor the tree to keep it red-black ?

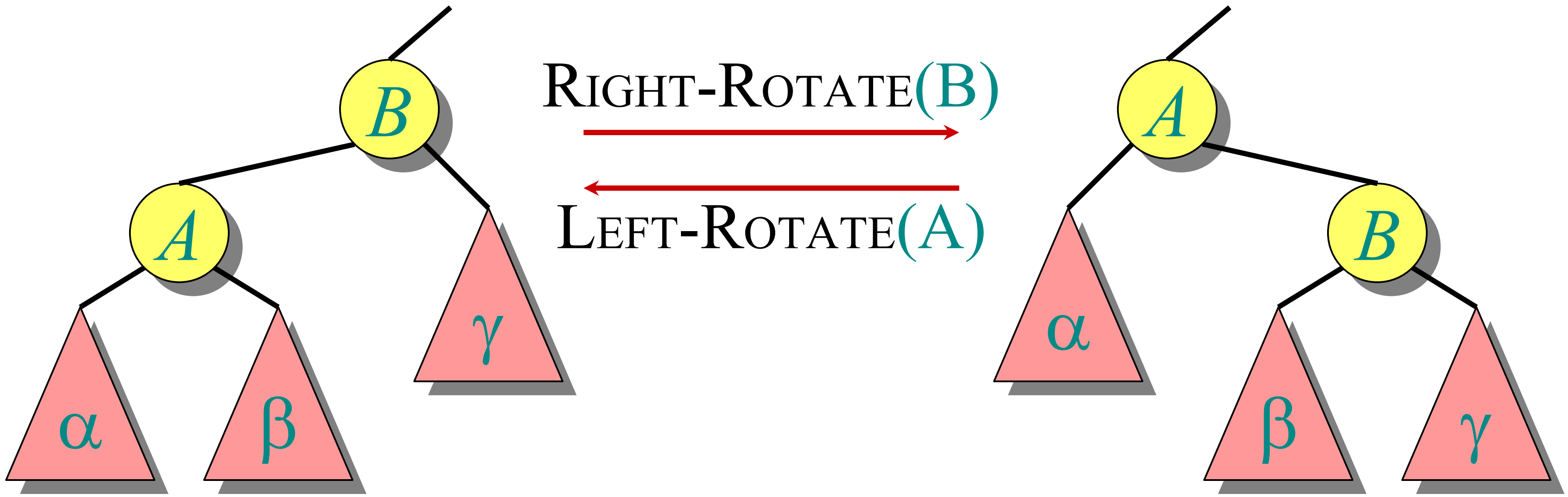
Example of a red-black tree



Use of red-black trees

- What properties would we like to prove about red-black trees ?
  - They **always** have O(log n) height
  - There is an O(log n)–time insertion procedure which preserves the red-black properties
- Is it true that, after we add a new element to a tree (as in the previous lecture), we can always recolor the tree to keep it red-black ?
- NO
- After insertions, sometimes we need to juggle nodes around

Rotations

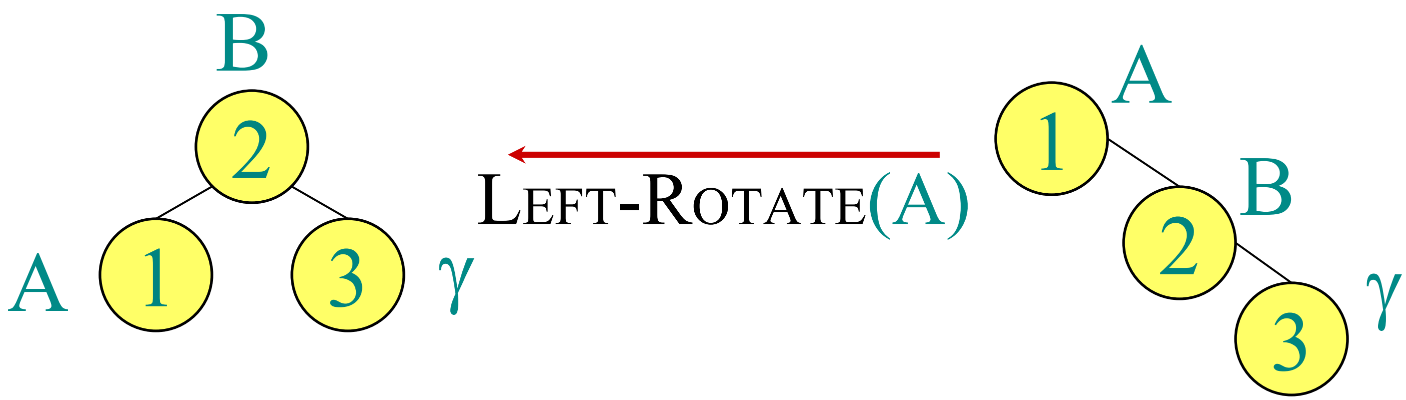


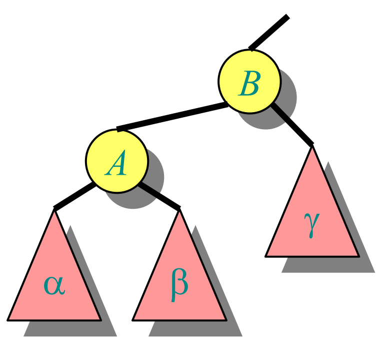
Rotations maintain the inorder ordering of keys:

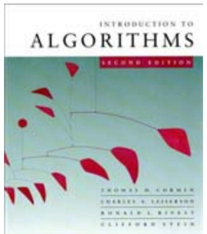
- $a \in \alpha, b \in \beta, c \in \gamma \;\Rightarrow\; a \le A \le b \le B \le c$.

A rotation can be performed in $O(1)$ time.

Rotations can reduce height







Red-black tree wrap-up

- Can show how
  - O(log n) re-colorings
  - 1 rotation

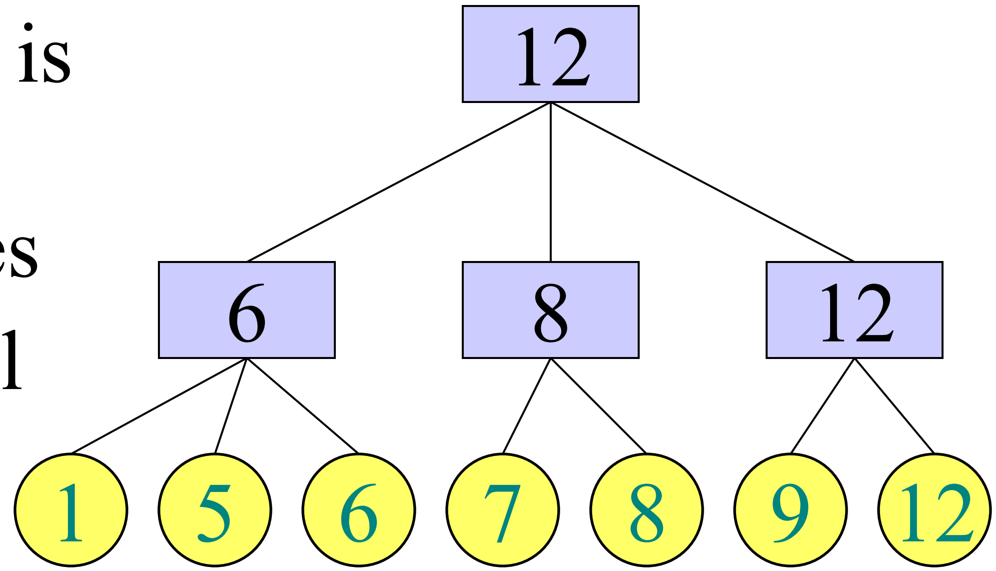
  can restore red-black properties after an insertion
- Instead, we will see 2-3 trees (but will come back to red-black trees at the end)

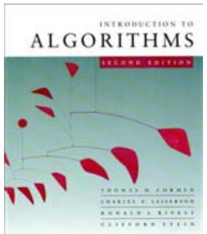
2-3 Trees

- The simplest balanced trees on the planet!
- Although a little bit more wasteful

2-3 Trees

- Degree of each node is either 2 or 3
- Keys are in the leaves
- All leaves have equal depth
- Leaves are sorted
- Each node x contains maximum key in the sub-tree, denoted x.max



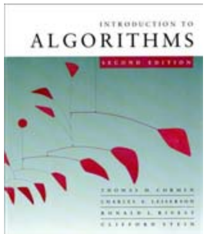


Internal nodes

- Internal nodes:
  - Values:
    - x.max: maximum key in the sub-tree
  - Pointers:
    - left[x]
    - mid[x]
    - right[x] : can be null
    - p[x] : can be null for the root
    - …
- Leaves:
  - x.max : the key

Height of 2-3 tree

- What is the maximum height h of a 2-3 tree with n nodes ?
- Alternatively, what is the minimum number of nodes in a 2-3 tree of height h ?
- It is $1+2+2^2+2^3+\ldots+2^h = 2^{h+1}-1$
- $n \geq 2^{h+1}-1 \Rightarrow h = O(\log n)$
- Full binary tree is the worst-case example!

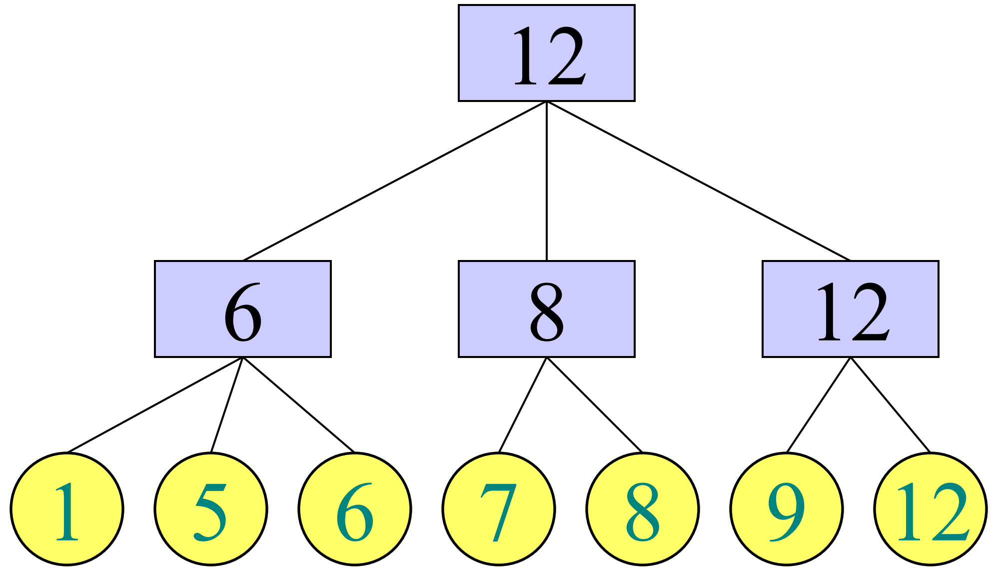


Searching

- How can we search for a key k ?

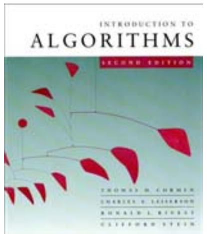
Search(x,k):

- If x=NIL then return NIL
- Else if x is a leaf then
  - If x.max=k then return x
  - Else return NIL
- Else
  - If k ≤ left[x].max then Search(left[x],k)
  - Else if k ≤ mid[x].max then Search(mid[x],k)
  - Else Search(right[x],k)



Search(8)

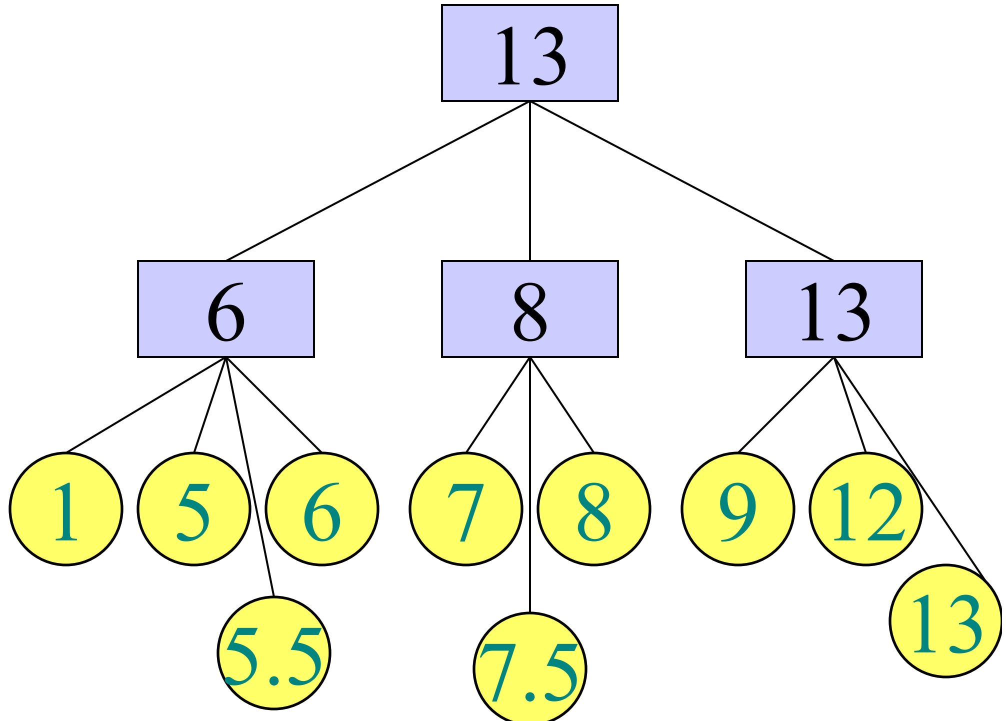
Search(13)



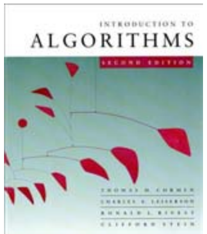
Insertion

- How to insert x ?
- Perform Search for the key of x
- Let y be the last internal node
- Insert x into y in a sorted order
- At the end, update the max values on the path to root

(continued on the next slide)



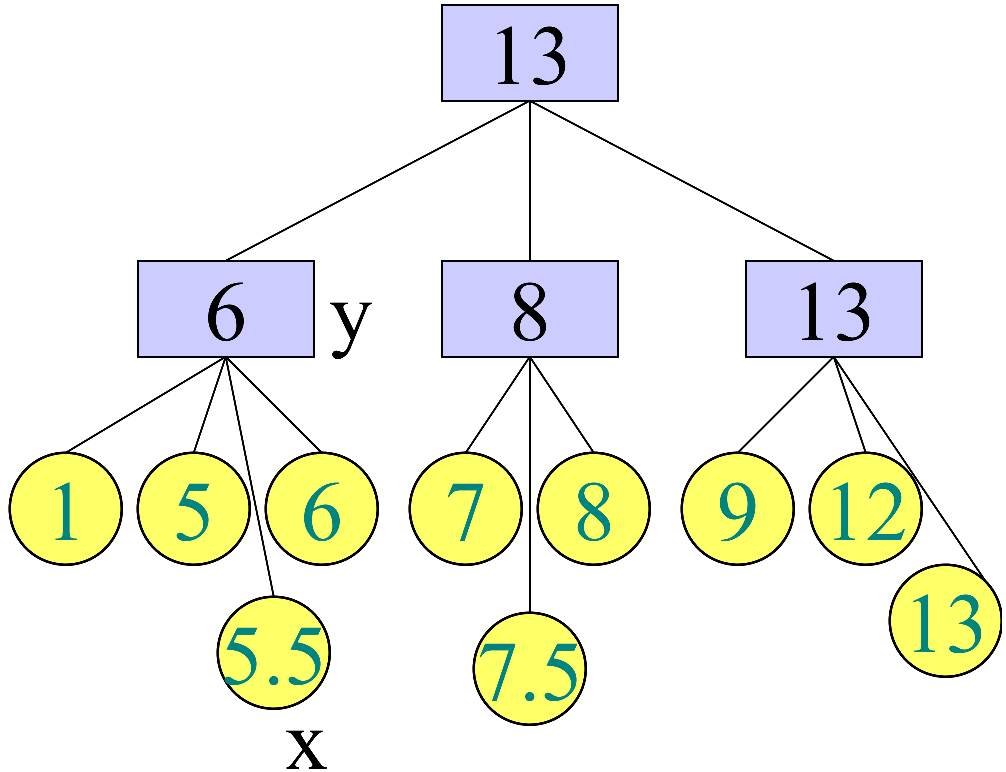
Insert(7.5)
Insert(13)
Insert(5.5)

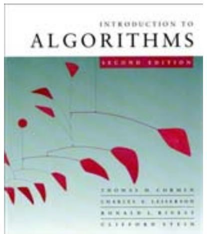


Insertion, ctd.

(continued from the previous slide)

- If y has 4 children, then Split(y)

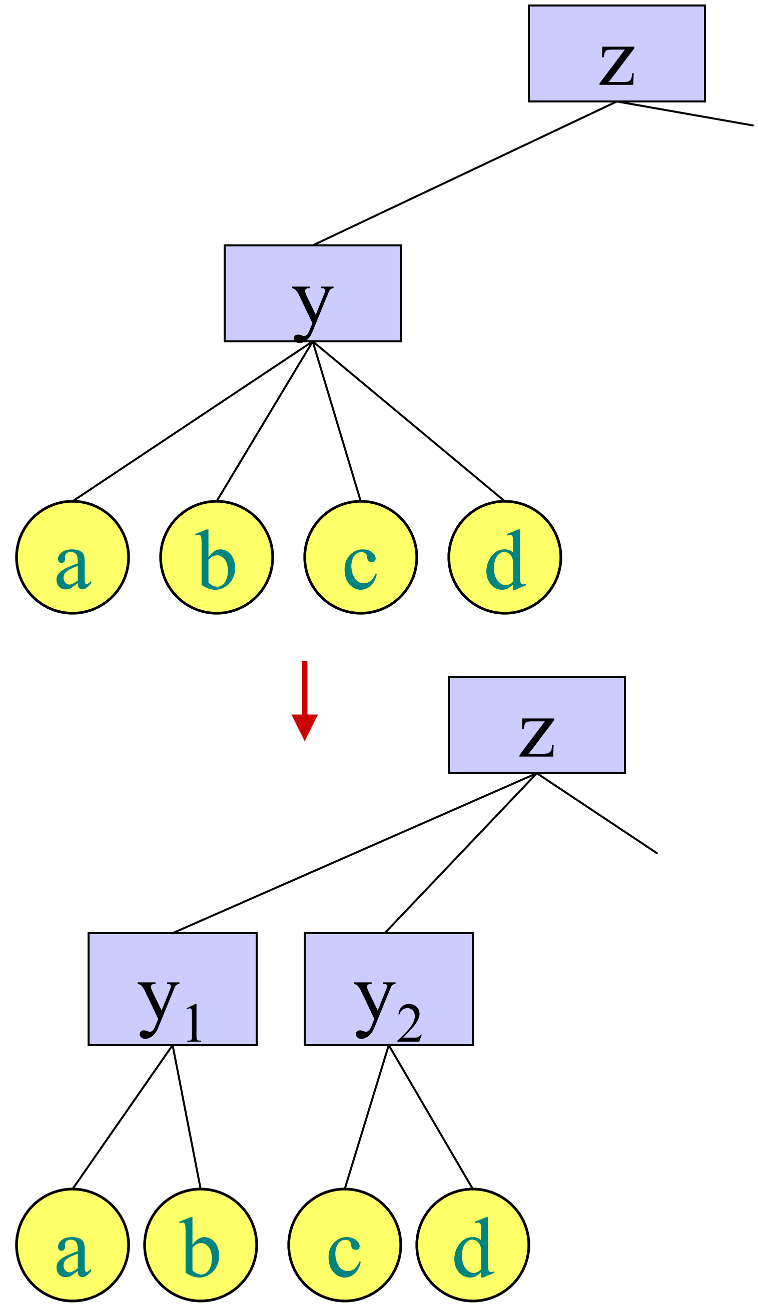




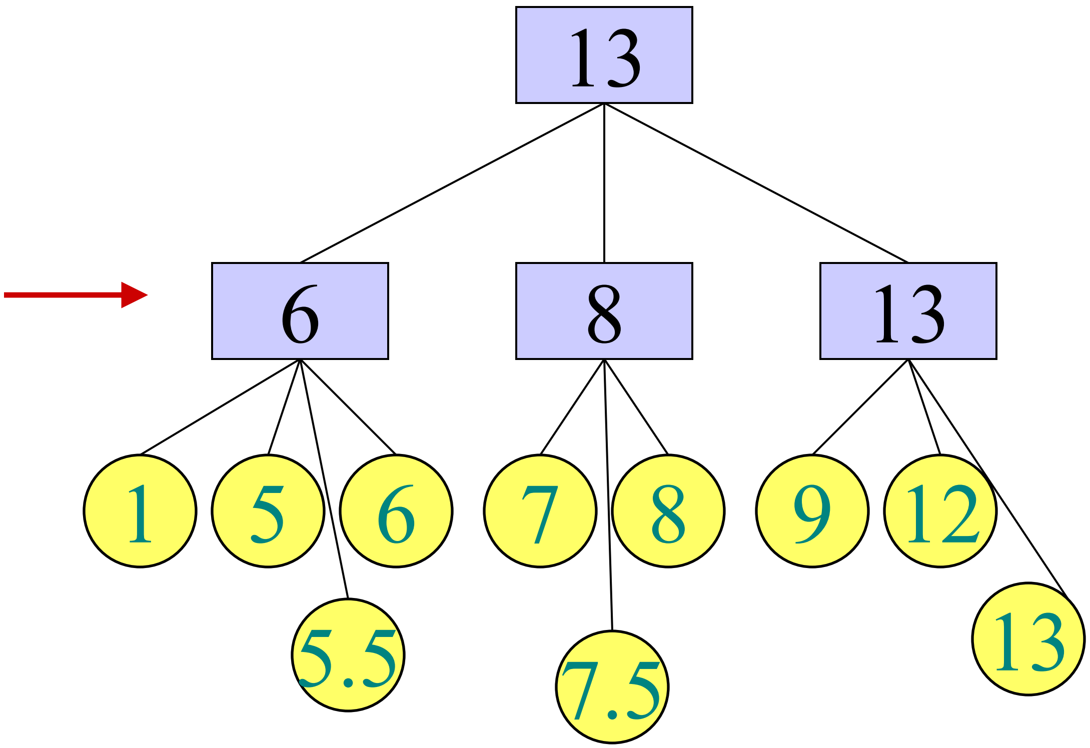
Split

- Split y into two nodes y_1, y_2
- Both are linked to z=parent(y)*
- If z has 4 children, split z

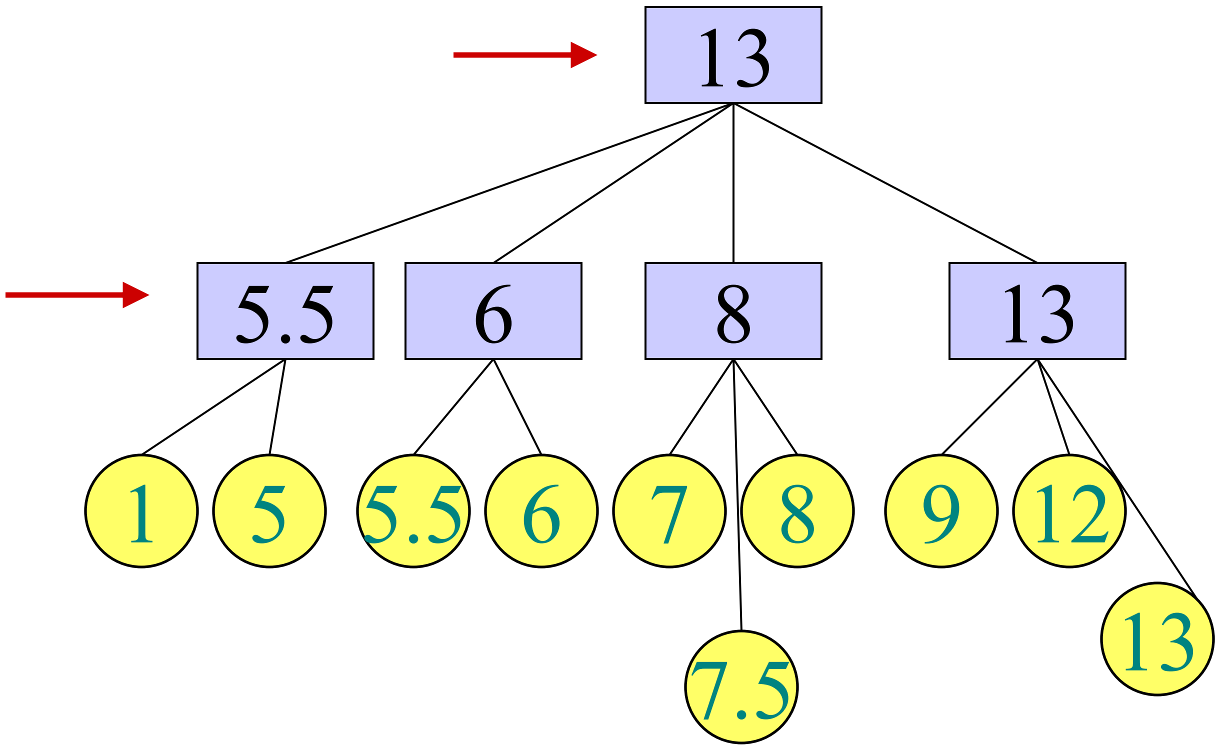
*If y is a root, then create new parent(y)=new root

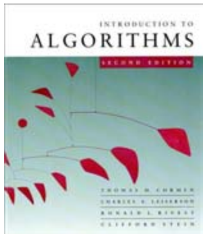


Split

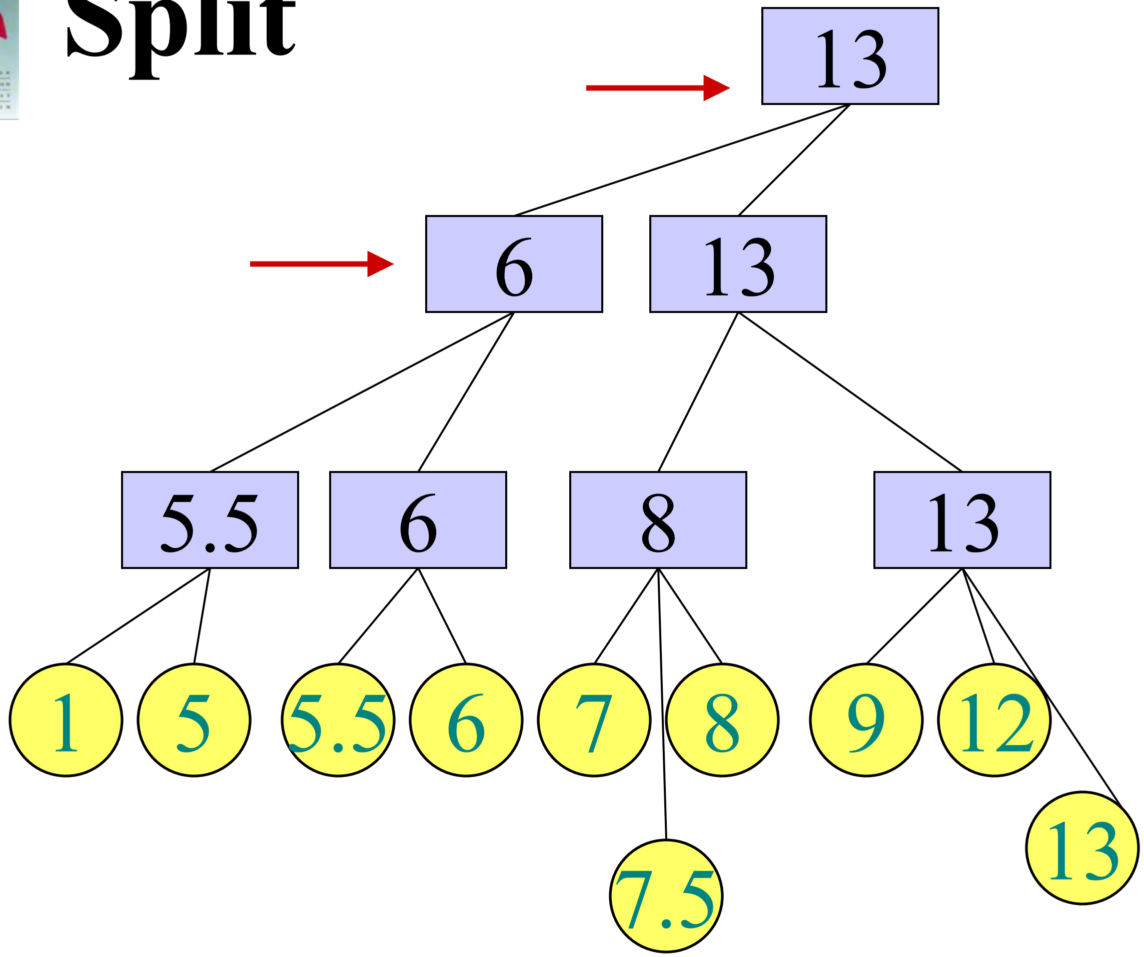


Split

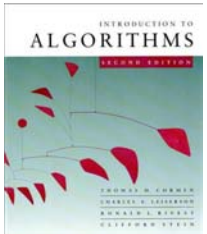




Split

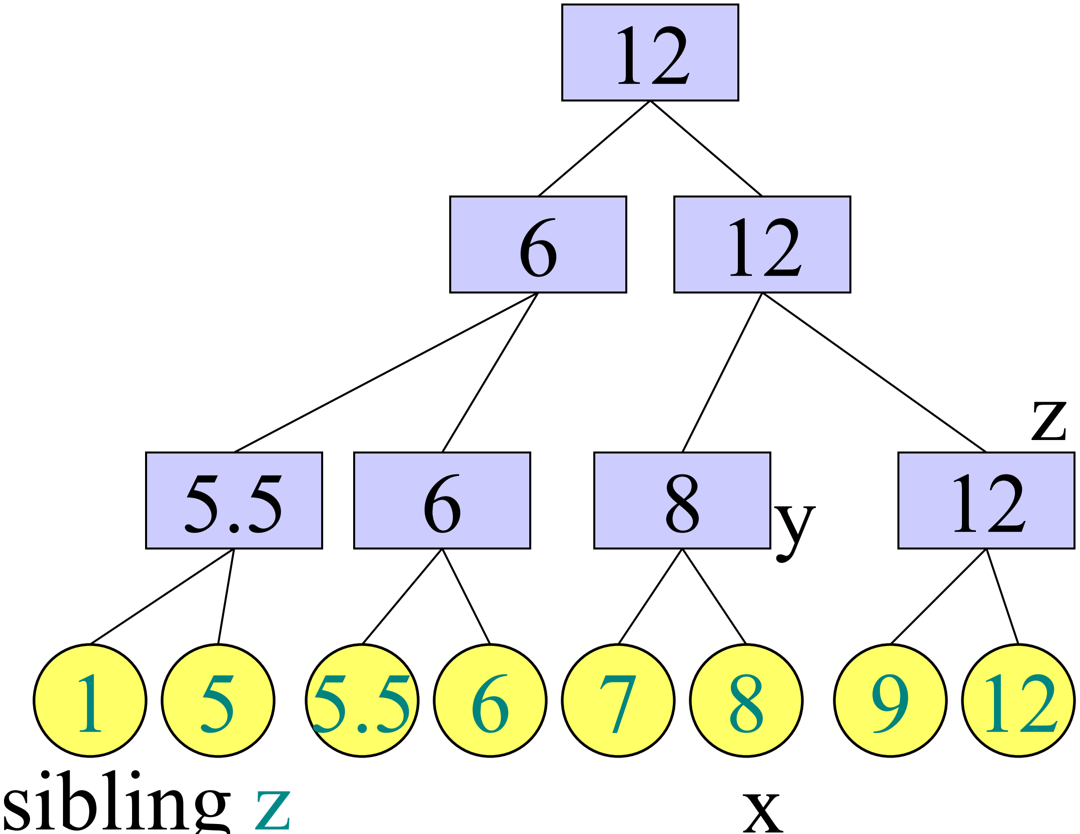


- Insert and Split preserve heights, unless new root is created, in which case all heights are increased by 1
- After Split, all nodes have 2 or 3 children
- Everything takes O(log n) time

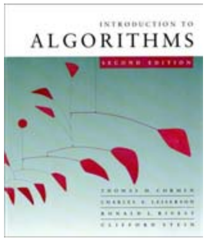


Delete

- How to delete x ?
- Let y=p(x)
- Remove x from y
- If y has 1 child:
  - Remove y
  - Attach x to y's sibling z

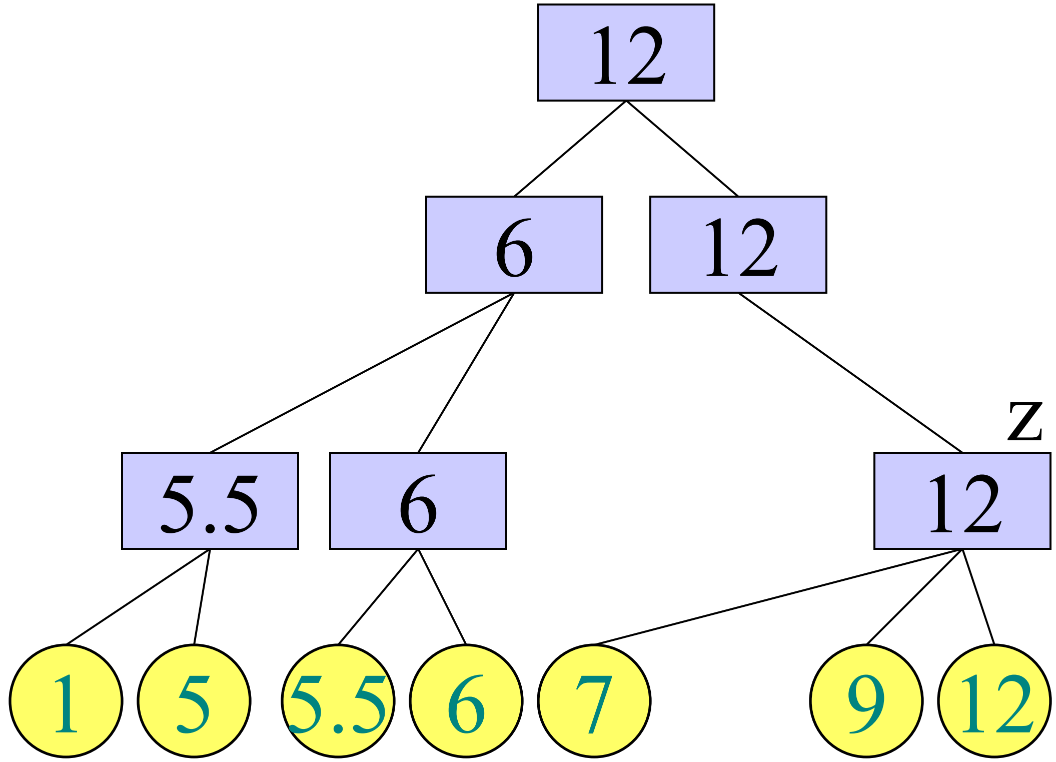


Delete(8)



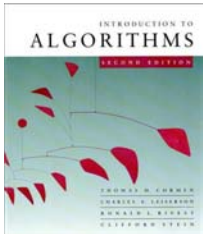
Delete

- How to delete x ?
- Let y=p(x)
- Remove x from y
- If y has 1 child:
  - Remove y
  - Attach x to y's sibling z
- If z has 4 children, then Split(z)



Delete(8)

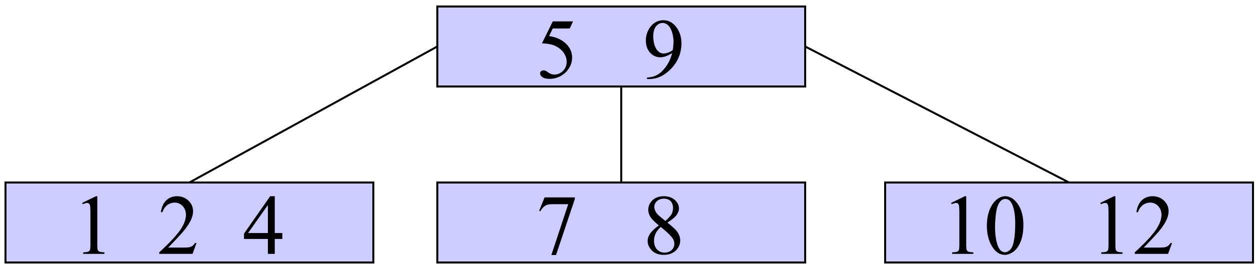
INCOMPLETE – SEE THE END FOR FULL VERSION

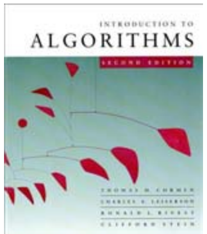


Summing up

- 2-3 Trees:
  - O(log n) depth ⇒ Search in O(log n) time
  - Insert, Delete (and Split) in O(log n) time
- We will now see 2-3-4 trees
  - Same idea, but:
    - Each parent has 2,3 or 4 children
    - Keys in the inner nodes
    - More complicated procedures

2-3-4 Trees





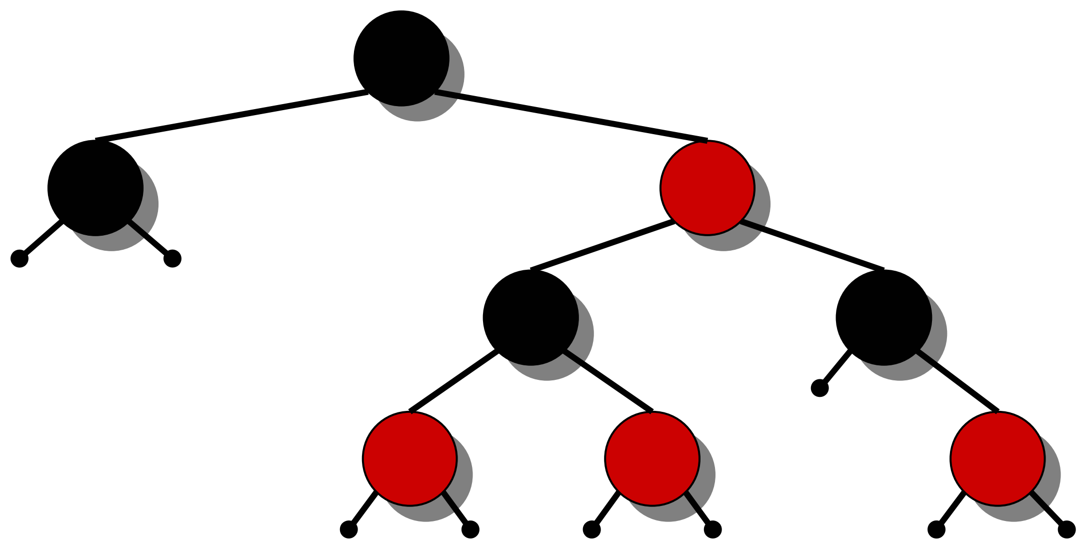
Height of a red-black tree

Theorem. A red-black tree with n keys has height

$$h \le 2 \lg(n + 1).$$

INTUITION:

- Merge red nodes into their black parents.



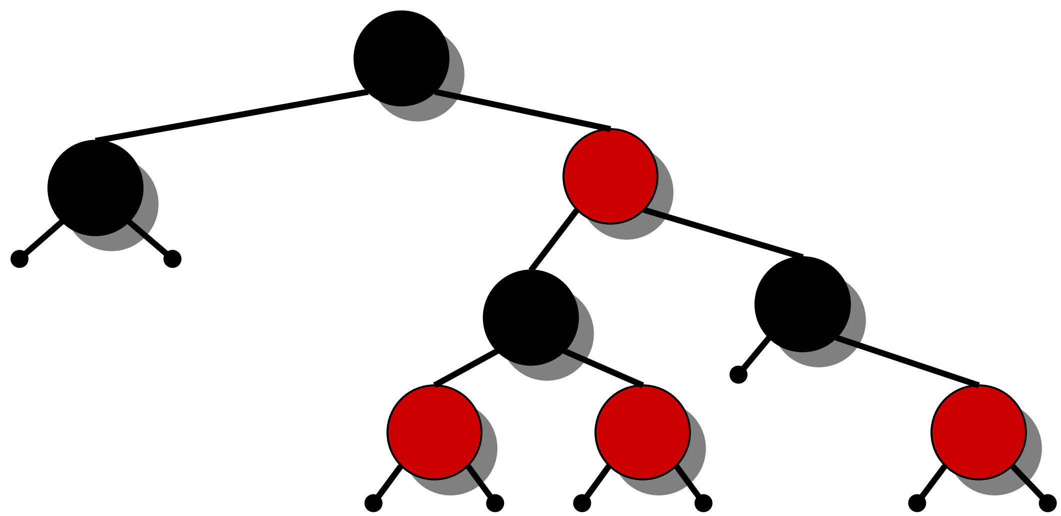
Height of a red-black tree

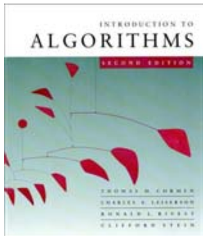
Theorem. A red-black tree with n keys has height

$$h \le 2 \lg(n + 1).$$

INTUITION:

- Merge red nodes into their black parents.

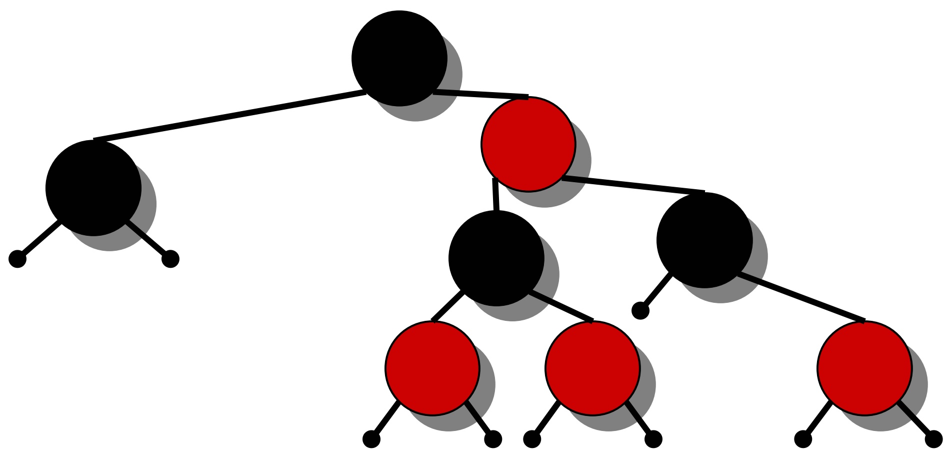


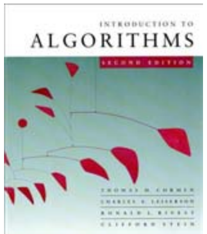


Height of a red-black tree

Theorem. A red-black tree with n keys has height
$$h \leq 2\lg(n + 1).$$

INTUITION:
- Merge red nodes into their black parents.





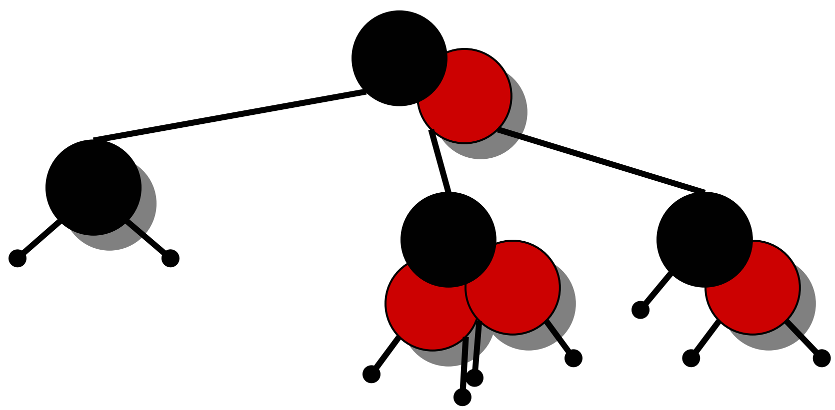
Height of a red-black tree

Theorem. A red-black tree with n keys has height

$$h \le 2 \lg(n + 1).$$

INTUITION:

- Merge red nodes into their black parents.



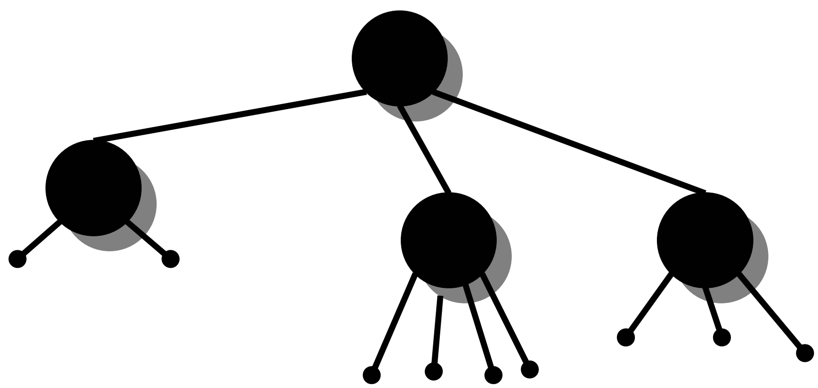
Height of a red-black tree

Theorem. A red-black tree with n keys has height

$$h \le 2 \lg(n + 1).$$

INTUITION:

- Merge red nodes into their black parents.



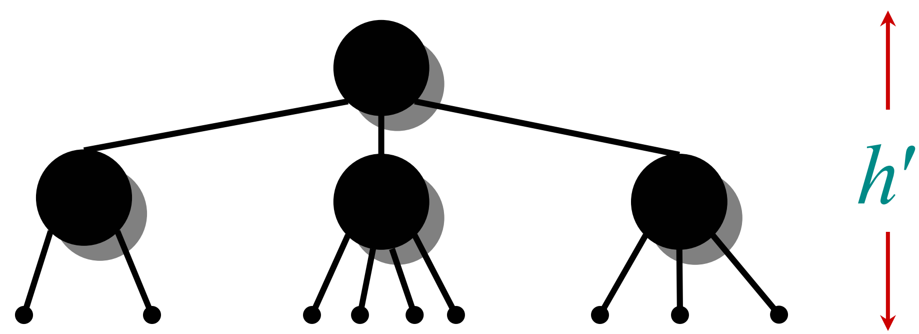
Height of a red-black tree

Theorem. A red-black tree with n keys has height

$$h \le 2\lg(n+1).$$

INTUITION:

- Merge red nodes into their black parents.



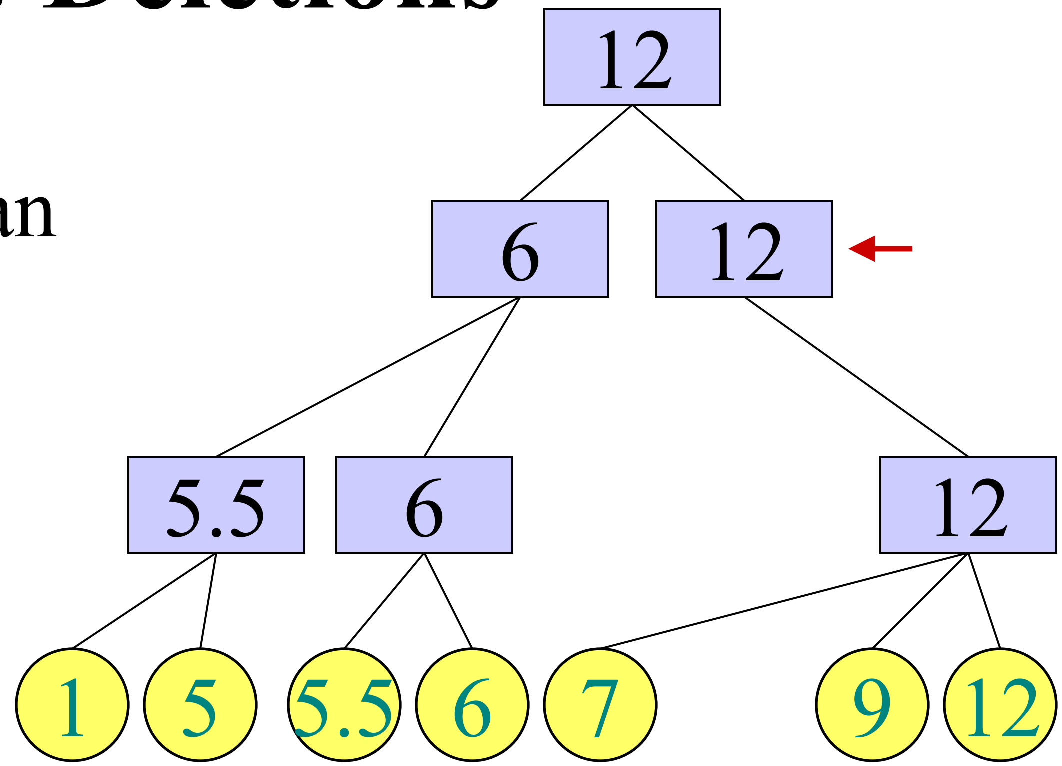
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.

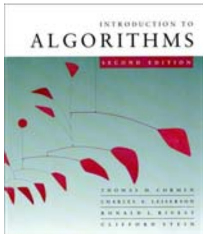
Summing up

- We have seen:
  - Red-black trees
  - 2-3 trees (in detail)
  - 2-3-4 trees
- Red-black trees are undercover 2-3-4 trees
- In most cases, does not matter what you use

2-3 Trees: Deletions

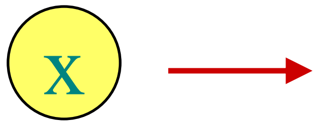
- Problem: there is an internal node that has only 1 child



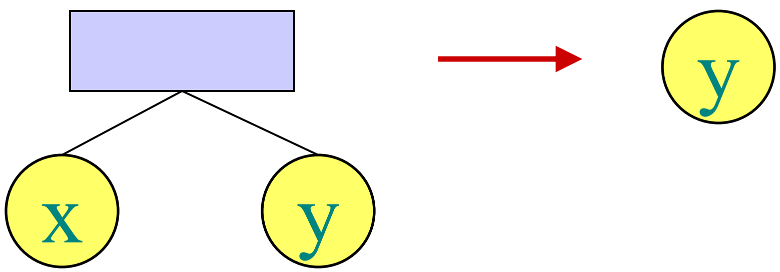


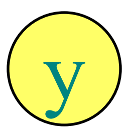
Full procedure for Delete(x)

- Special case: x is the only element in the tree: delete everything



- Not-so-special case: x is one of two elements in the tree. In this case, the procedure on the next slide will delete x



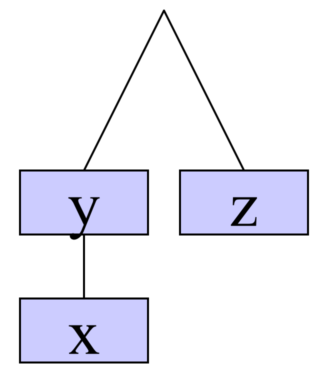
- Both NIL and  are special 2-3 trees

Procedure for Delete(x)

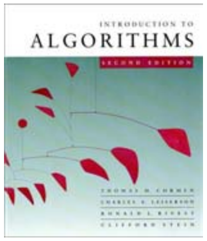
- Let y=p(x)
- Remove x
- If y≠root then
  - Let z be the sibling of y.
  - Assume z is the right sibling of y, otherwise the code is symmetric.
  - If y has only 1 child w left

    Case 1: z has 3 children
    - Attach left[z] as the rightmost child of y
    - Update y.max and z.max

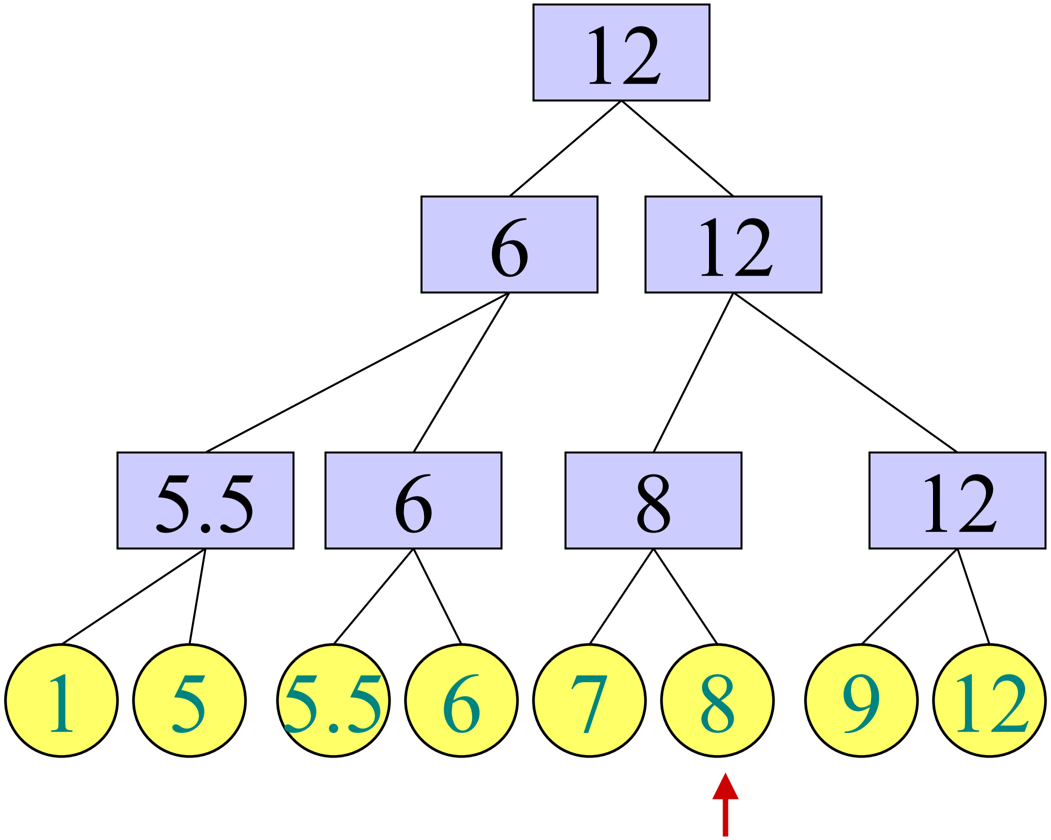
    Case 2: z has 2 children:
    - Attach the child w of y as the leftmost child of z
    - Update z.max
    - Delete(y) (recursively*)
  - Else
    - Update max of y, p(y), p(p(y)) and so on until root
- Else
  - If root has only one child u
    - Remove root
    - Make u the new root



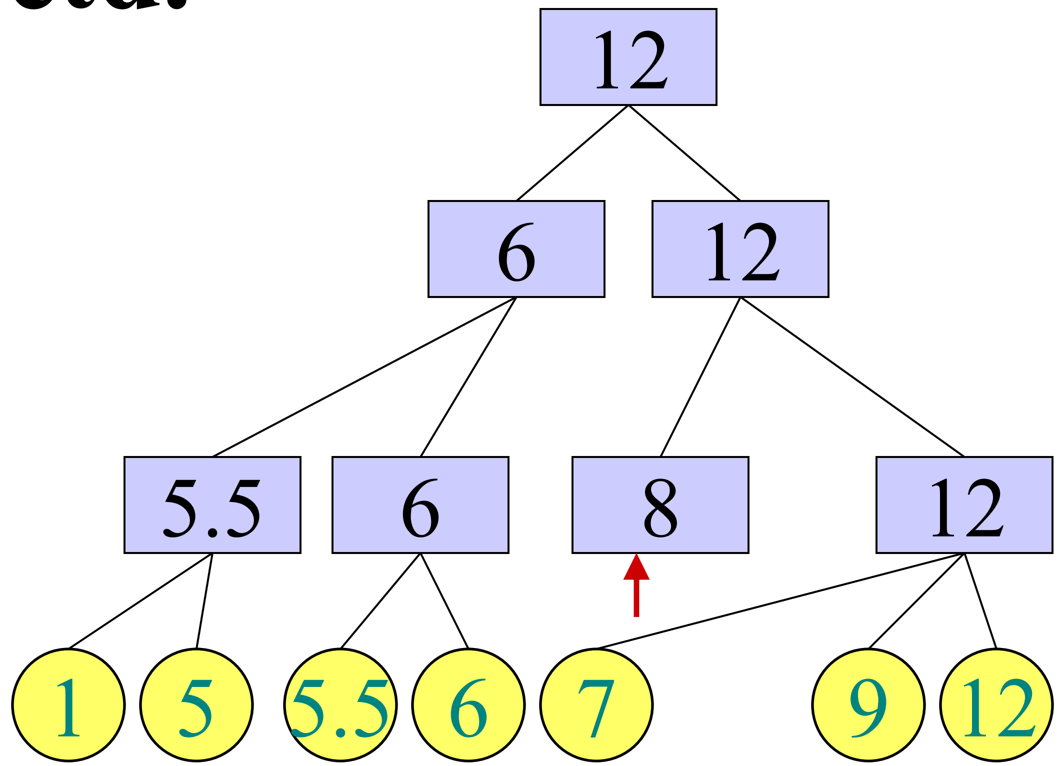
*Note that the input of Delete does not have to be a leaf


INTRODUCTION TO
ALGORITHMS
SECOND EDITION

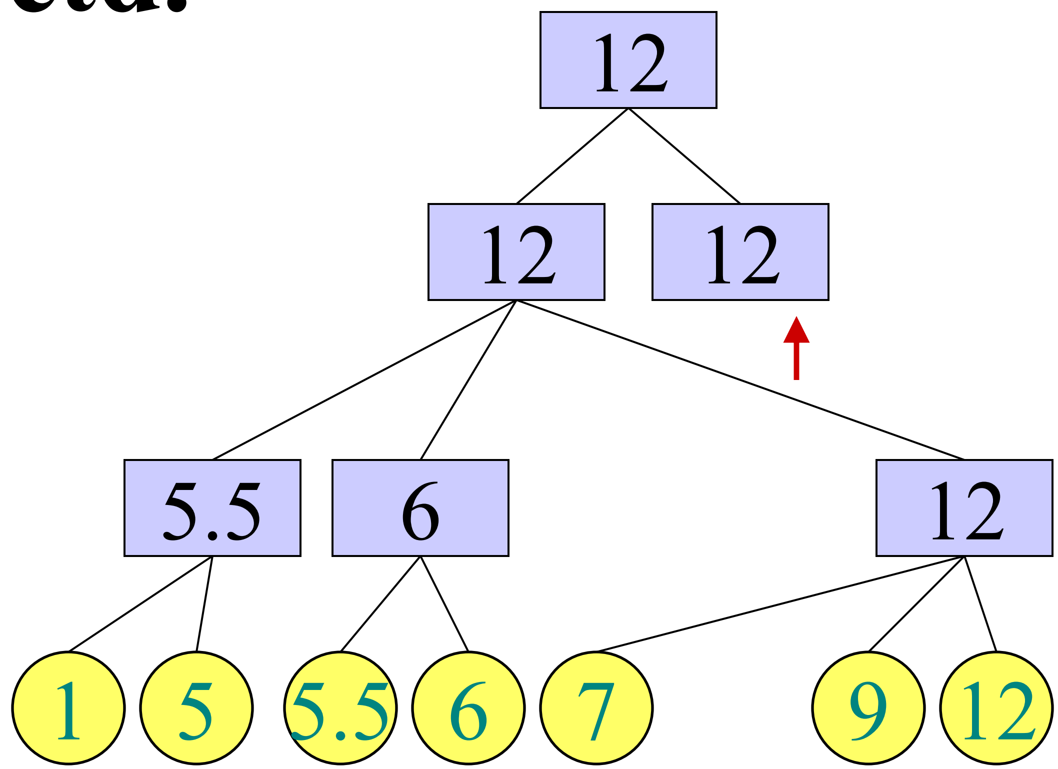
Example


12
6
12
5.5
6
8
12
1
5
5.5
6
7
8
9
12

Example, ctd.



Example, ctd.



Example, ctd.