Algorithm Analysis and Data Structures

CSCI 7432 - Fall 2022

Tree-Based Data Structures

Dr. Yao XU

Assistant Professor

Department of Computer Science

Georgia Southern University

Email: yxu@georgiasouthern.edu

Table of Contents

Heaps

Heap Data Structure (1/2)

A ***(binary) heap*** is an array $A[1..n]$ that we can view as a **binary tree** with keys stored at its nodes (one key per node) as follow:

- Root of the tree is $A[1]$
- Parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$
- Left child of $A[i]$ is $A[2i]$
- Right child of $A[i]$ is $A[2i+1]$

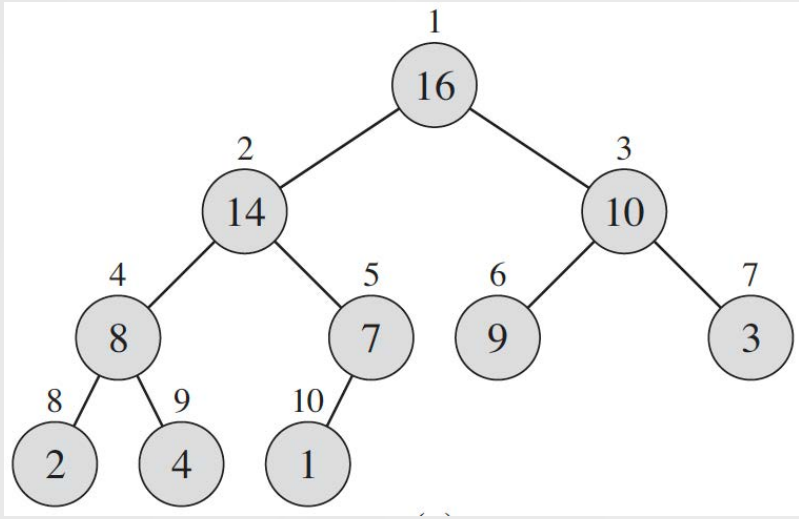
Two types of heaps: ***Max-heap***, ***Min-heap***

Example (of a max-heap):

0	1	2	3	4	5	6	7	8	9	10
	16	14	10	8	7	9	3	2	4	1

Three procedures:

- PARENT(i) returns $\lfloor i/2 \rfloor$
- LEFT(i) returns $2i$
- RIGHT(i) returns $2i+1$



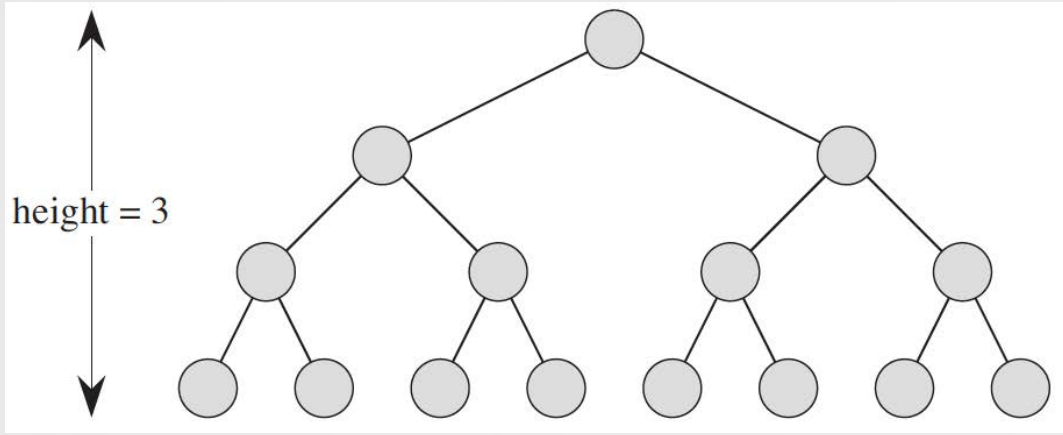
Heap Data Structure (2/2)

The binary tree representation of a ***heap*** should satisfy:

1. The **shape property**: It is a nearly complete binary tree. – All non-leaf nodes have two children except for possibly some rightmost nodes.
2. The **heap property**:
   - For ***max-heap***: The key at each node $\geq$ the key at its child
   - For ***min-heap***: The key at each node $\leq$ the key at its child

Height of a Heap

- The **height of a node v** is the number of edges on the longest downward path from v to a leaf.
- The ***height of the heap*** (viewing as a tree) is the height of its root.
- **Q:** What is the height of a heap with n elements?
  - #nodes in a complete binary tree with height h is
    $1 + 2 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$
  - $\Rightarrow$ The height of a heap with n elements is:
    $\log_2(n+1) - 1 \leq h \leq \log_2 n$



- **Ans:** $h = \lfloor \log_2 n \rfloor \in \Theta(\log n)$

Heaps

Maintaining the Heap Property

Max-Heap Property

In the binary tree representation of a **max-heap** $A[1..n]$,

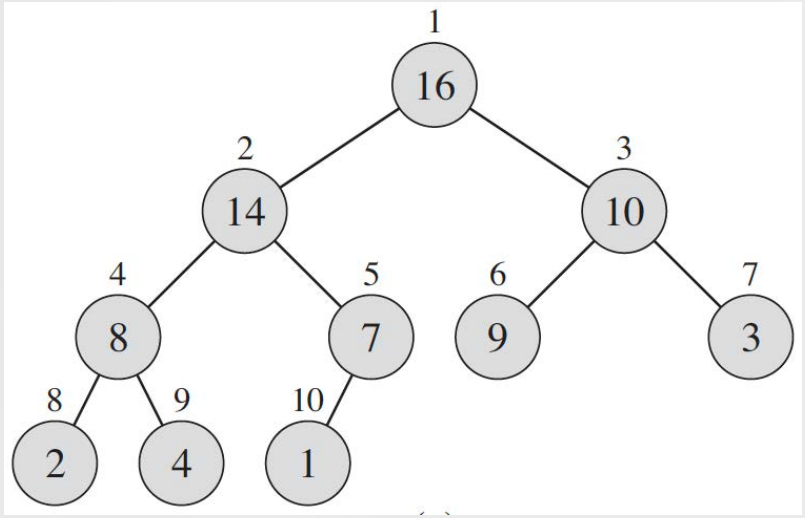
- For all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.

➔ The largest element is at root $A[1]$

➔ The nodes in any subtree also form a max-heap

Example:

0	1	2	3	4	5	6	7	8	9	10
	16	14	10	8	7	9	3	2	4	1



Q: How to convert an arbitrary array to a max-heap?

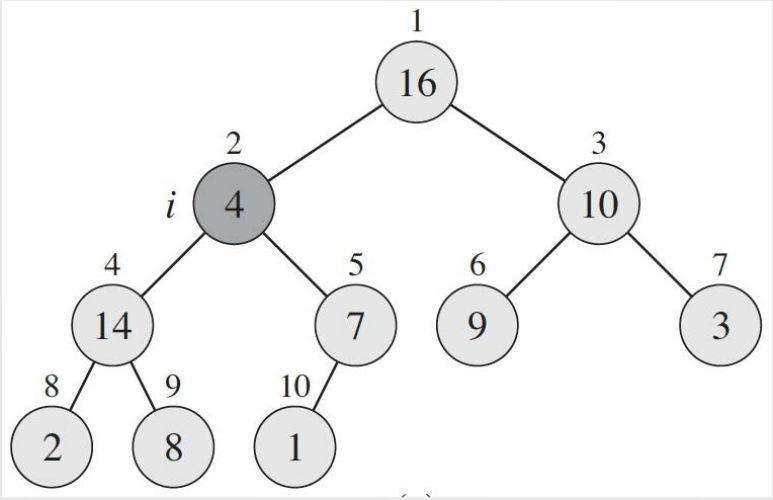
Maintaining the Heap Property

MAX-HEAPIFY converts an ***almost-max-heap*** into a **max-heap**.

- ***Almost-max-heap***: only the root might violate the max-heap property.
- Before MAX-HEAPIFY, the left and right subtrees of $A[i]$ are max-heaps.
- After MAX-HEAPIFY, the subtree rooted at $A[i]$ is a max-heap.

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

Example: $i = 2, A[i] = 4$

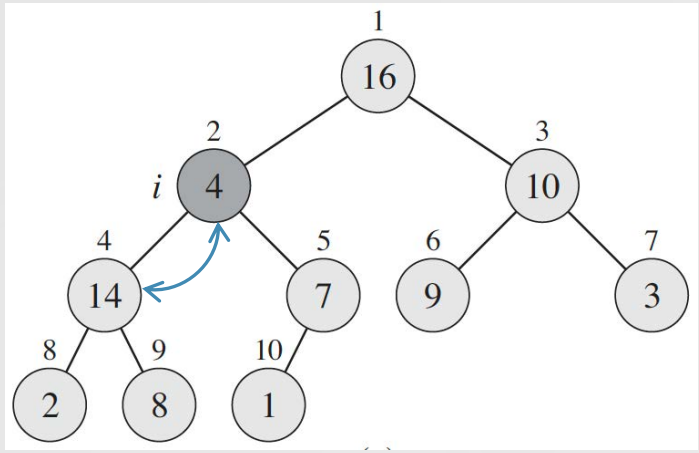


MAX-HEAPIFY Example

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

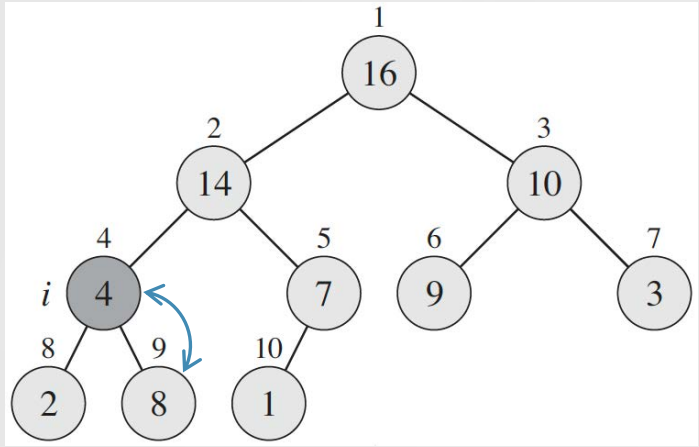
LEFT(i) returns $2i$
RIGHT(i) returns $2i + 1$

(1) MAX-HEAPIFY(A, 2, 10)

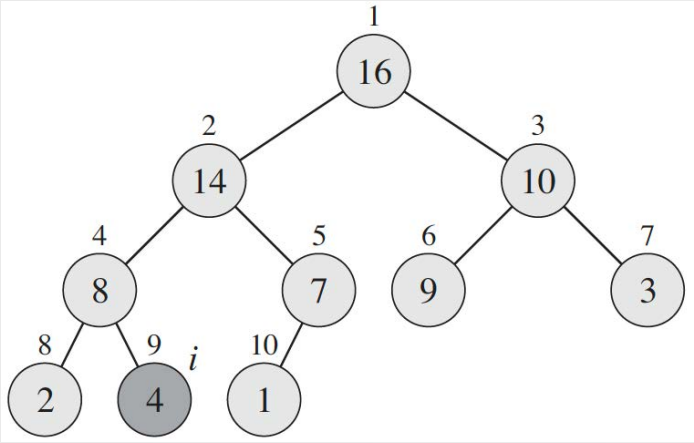


(1) $A[2] = 4$,
$A[largest] = 14$
(2) $A[4] = 4$,
$A[largest] = 8$
(3) No further changes.

(2) MAX-HEAPIFY(A, 4, 10)



(3) MAX-HEAPIFY(A, 9, 10)

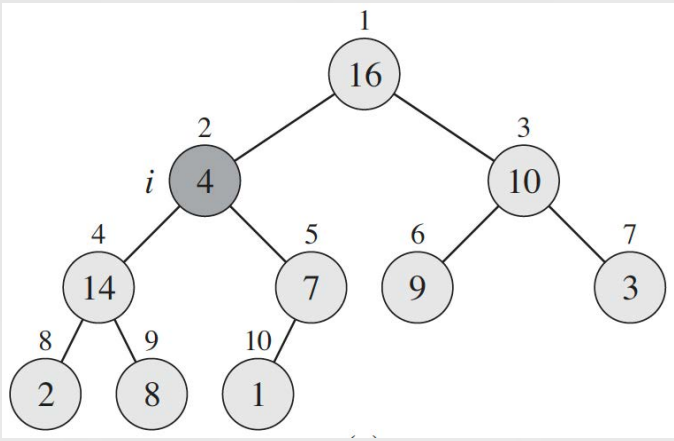


MAX-HEAPIFY Time Complexity

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

LEFT(i) returns $2i$
RIGHT(i) returns $2i + 1$

Example:



- **Running time** (in terms of heap size n):
  - Best case: $T(n) \in \Theta(1)$
  - Worst case: Let h be the height of the heap

$$T(h) = \Theta(1) + T(h-1)$$
$$\Rightarrow T(h) \in \Theta(h) \Rightarrow T(n) \in \Theta(\log n)$$

Together, $T(n) \in O(\log n)$.

Heaps

Building A Heap

Building a Max-Heap

Q: How to convert an arbitrary array to a max-heap?

Idea of a ***bottom-up heap construction*** algorithm:

- Look at its binary tree representation
- Consider the leaves (the bottom-level of nodes)

  ➔ Each leaf is a single key ⇒ already a max-heap
- Consider the nodes on the second-to-last level

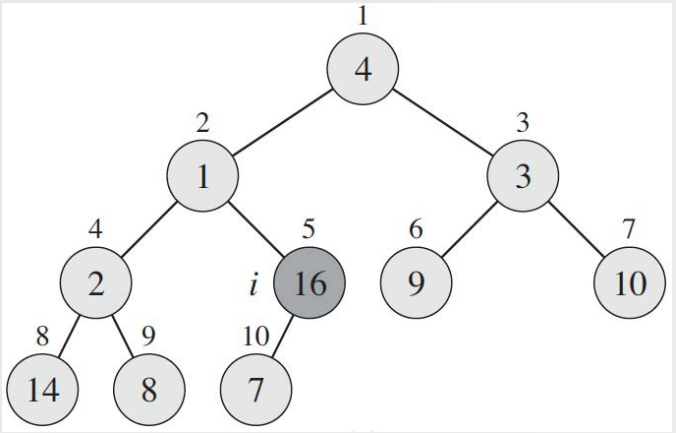
  ➔ The subtrees rooted at these nodes are almost-max-heap ⇒ MAX-HEAPIFY!
- Consider the nodes on the third-to-last level

  ➔ Now the subtrees rooted at these nodes are also almost-max-heap

  ⋮

  ⇒ MAX-HEAPIFY!
- The whole tree becomes an almost-max-heap ⇒ MAX-HEAPIFY the tree's root!

Example:



Bottom-Up Heap Construction (1/3)

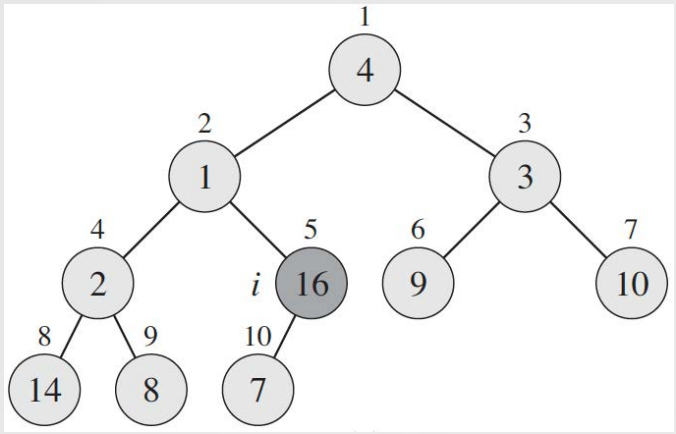
```
BUILD-MAX-HEAP(A, n)
1  for i = ⌊n/2⌋ downto 1
2      MAX-HEAPIFY(A, i, n)
```

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

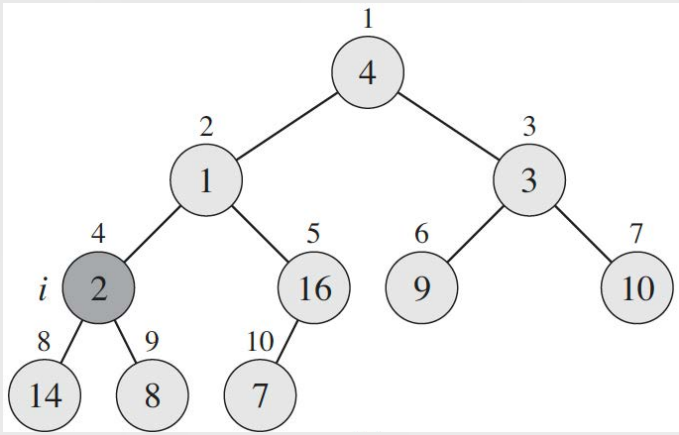
- **Example:** BUILD-MAX-HEAP(A, 10)

0	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7

(1) MAX-HEAPIFY(A, 5, 10)



(2) MAX-HEAPIFY(A, 4, 10)



Bottom-Up Heap Construction (2/3)

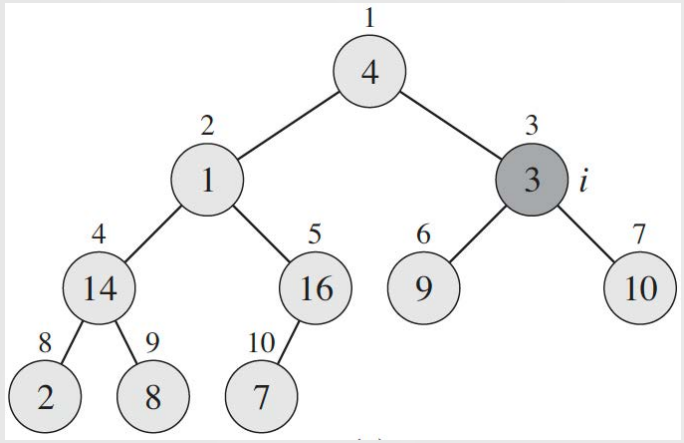
```
BUILD-MAX-HEAP(A, n)
1  for i = ⌊n/2⌋ downto 1
2      MAX-HEAPIFY(A, i, n)
```

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

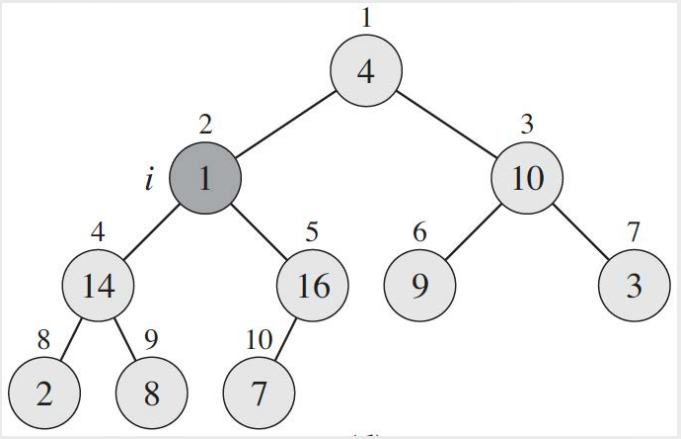
- **Example:**

0	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7

(3) MAX-HEAPIFY(A, 3, 10)



(4) MAX-HEAPIFY(A, 2, 10)



Bottom-Up Heap Construction (3/3)

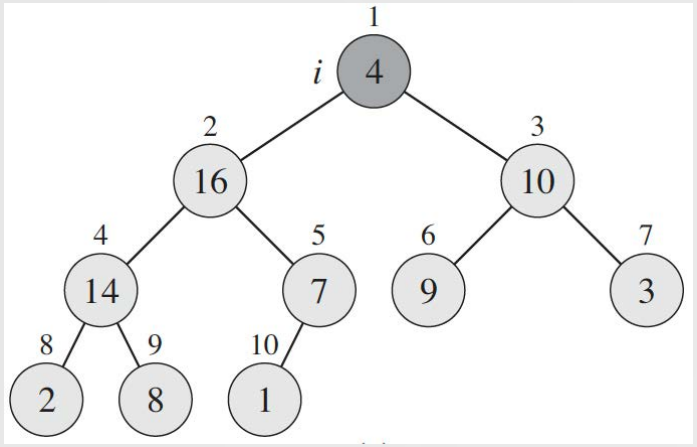
```
BUILD-MAX-HEAP(A, n)
1  for i = ⌊n/2⌋ downto 1
2      MAX-HEAPIFY(A, i, n)
```

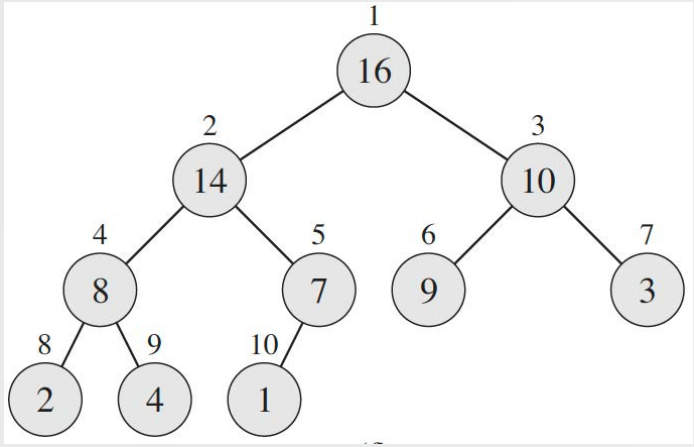
```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

- **Example:**

0	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7

(5) MAX-HEAPIFY(A, 1, 10)



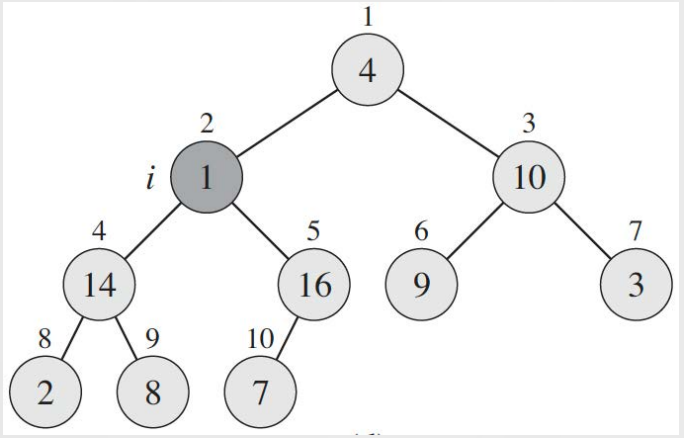


Time Complexity (1/3)

```
BUILD-MAX-HEAP(A, n)
1  for i = ⌊n/2⌋ downto 1
2      MAX-HEAPIFY(A, i, n)
```

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

- **Simple upper bound**: $O(n \log n)$
  - $O(n)$ calls to MAX-HEAPIFY
  - MAX-HEAPIFY takes $O(\log n)$ time
- **Tighter upper bound**:
  - In the worst case, for a node $A[i]$ at height h, MAX-HEAPIFY(A, i, n) takes $\Theta(h)$ time.

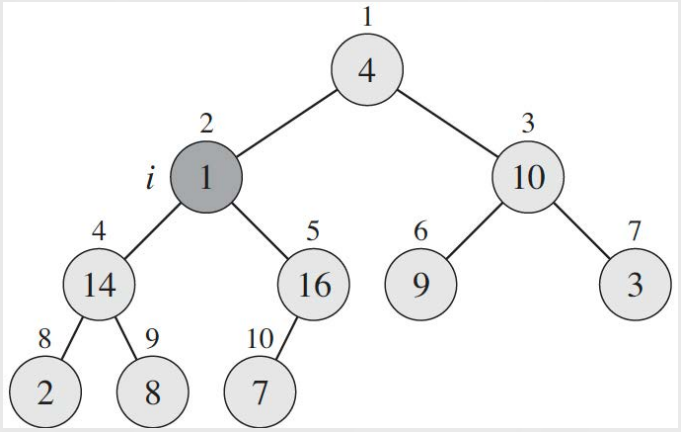


Time Complexity (2/3)

```
BUILD-MAX-HEAP(A, n)
1  for i = ⌊n/2⌋ downto 1
2      MAX-HEAPIFY(A, i, n)
```

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

- **Tighter upper bound**:
  - There are $\leq \lceil n/2 \rceil$ leaves
  - At height 1, we have $\leq \lceil n/4 \rceil$ nodes
  - At height 2, we have $\leq \lceil n/8 \rceil$ nodes
  - ⋮
  - At height $h = \lfloor \log_2 n \rfloor$, we have 1 node



- Thus, $T(n) \leq \sum_{h=1}^{\lfloor \log_2 n \rfloor} \left(\left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) \right)$

Time Complexity (2/3)

```
BUILD-MAX-HEAP(A, n)
1  for i = ⌊n/2⌋ downto 1
2      MAX-HEAPIFY(A, i, n)
```

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

- **Tighter upper bound:**

$$T(n) \le \sum_{h=1}^{\lfloor \log_2 n \rfloor} \left(\left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) \right)$$

$$T(n) \in O\left(\sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{nh}{2^{h+1}} \right)$$

Due to $\sum_{h=1}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = 2$,*

$$T(n) \in O\left(n \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right)$$

$$= O(n)$$

Therefore, $T(n) \in O(n)$.

** According to equation A.8 on p.1148 of the textbook.*

Priority Queues

Priority Queues

- A ***priority queue*** is a data structure for maintaining a dynamic set S of elements, each with an associated value called a ***key***, representing the priority of the element.
- Two types of **priority queues**:
  - **max-priority queue** (using max-heap)
  - **min-priority queue** (using min-heap)
- **Example:** A set of jobs to be scheduled on a shared computer
  - The jobs arrive and should be placed in the queue, each with a priority.
  - To perform a job, we "extract" the one in the queue with highest priority.

Max-Priority Queue

- **Max-priority queue** supports the following operations:
  1. HEAP-MAXIMUM(A): returns element of A with largest key.
  2. HEAP-EXTRACT-MAX(A): removes and returns element of A with the largest key.
  3. HEAP-INCREASE-KEY(A, i, k): increases $A[i]$ to k. Assume $k \geq A[i]$.
  4. MAX-HEAP-INSERT(A, k): inserts element with value k into A.
- **Max-priority queues** are implemented with max-heaps.

Find and Remove Maximum

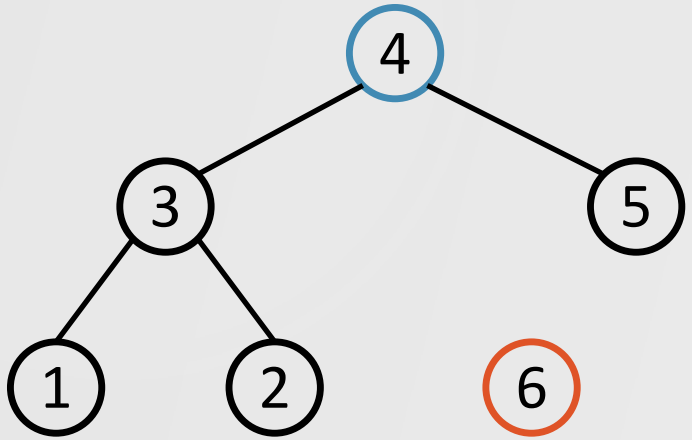
```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if A.heapsize < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heapsize]
5  A.heapsize = A.heapsize – 1
6  MAX-HEAPIFY(A, 1, A.heapsize)
7  return max
```

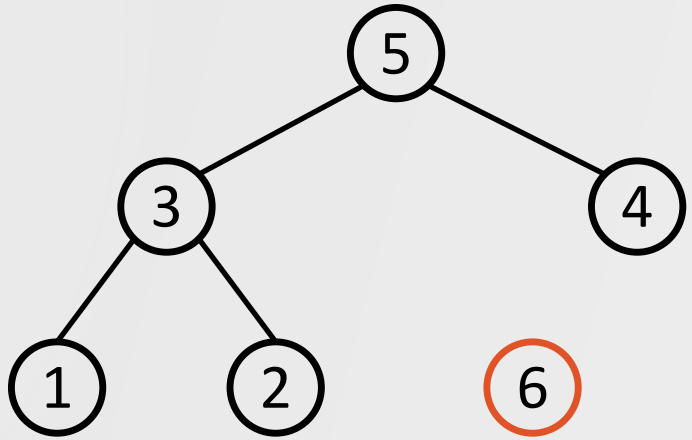
- **Running time** of HEAP-MAXIMUM: $\Theta(1)$
- **Running time** of HEAP-EXTRACT-MAX: $O(\log n)$
- **Example:**

1	2	3	4	5	6
6	3	5	1	2	4

(1) Swap $A[1]$ with $A[6]$



(2) MAX-HEAPIFY(A, 1, 5)



Increasing Key Value

Given set S, array index i, and a new value k, Update $A[i]$ to k. (Assume $k \geq A[i]$.)

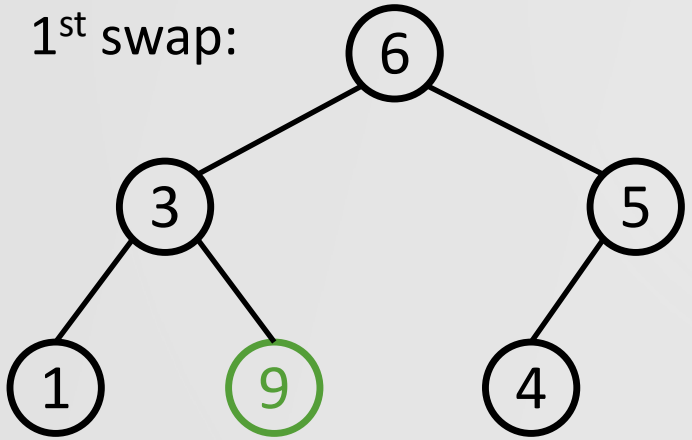
```
HEAP-INCREASE-KEY(A, i, k)
1  if k < A[i]
2       error "new key is smaller than current key"
3  A[i] = k
4  while i > 1 and A[PARENT(i)] < A[i]
5       swap A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

- **Example:** HEAP-INCREASE-KEY$(A, 5, 9)$

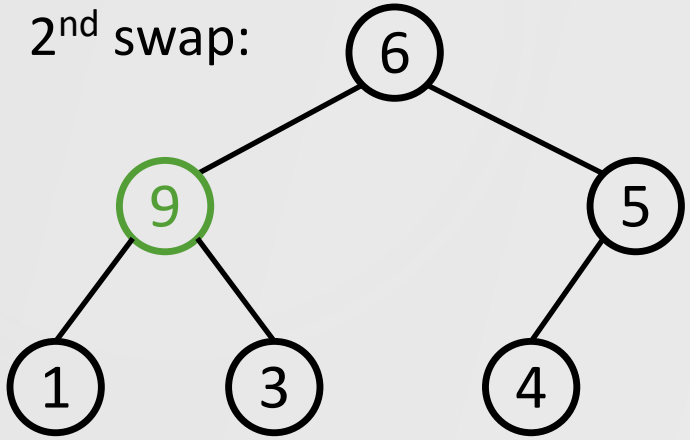
1	2	3	4	5	6
6	3	5	1	2	4

- **Running time**: $T(n) \in O(h) = O(\log n)$

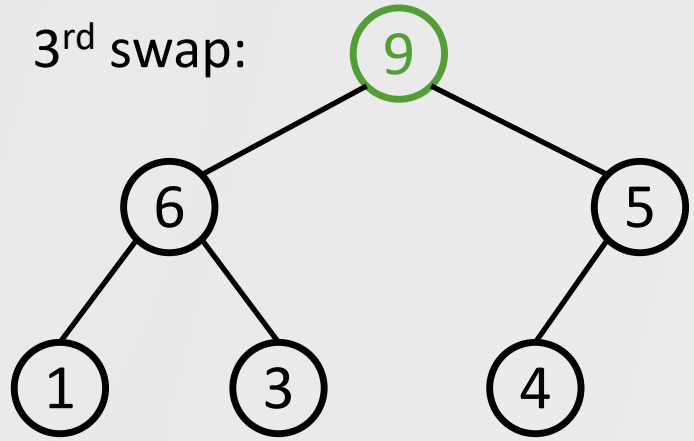
1st swap:



2nd swap:



3rd swap:



Inserting into the Heap

MAX-HEAP-INSERT(A, k)

1 $A.heapsize = A.heapsize + 1$

2 $A[A.heapsize] = -\infty$

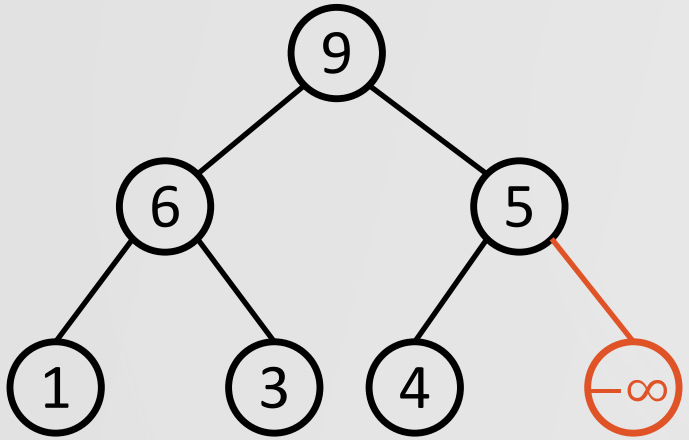
3 HEAP-INCREASE-KEY(A, $A.heapsize$, k)

- **Example:** MAX-HEAP-INSERT(A, 8)

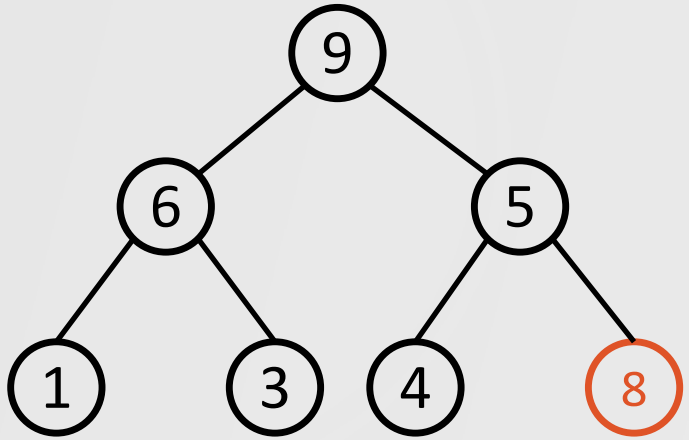
1	2	3	4	5	6
9	6	5	1	3	4

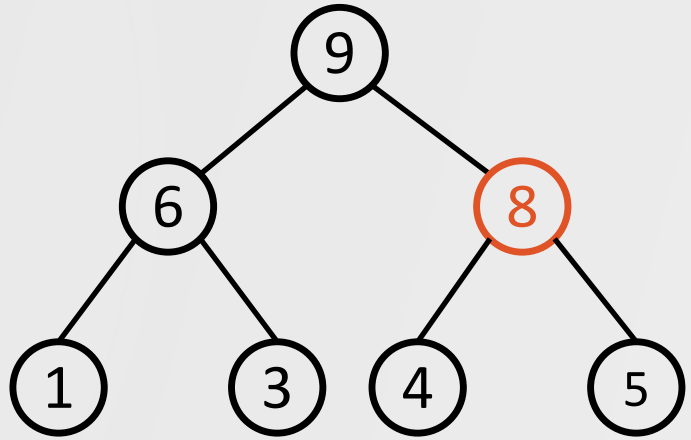
- **Running time**: $O(\log n)$

(1) $A[7] = -\infty$



(2) HEAP-INCREASE-KEY(A, 7, 8)





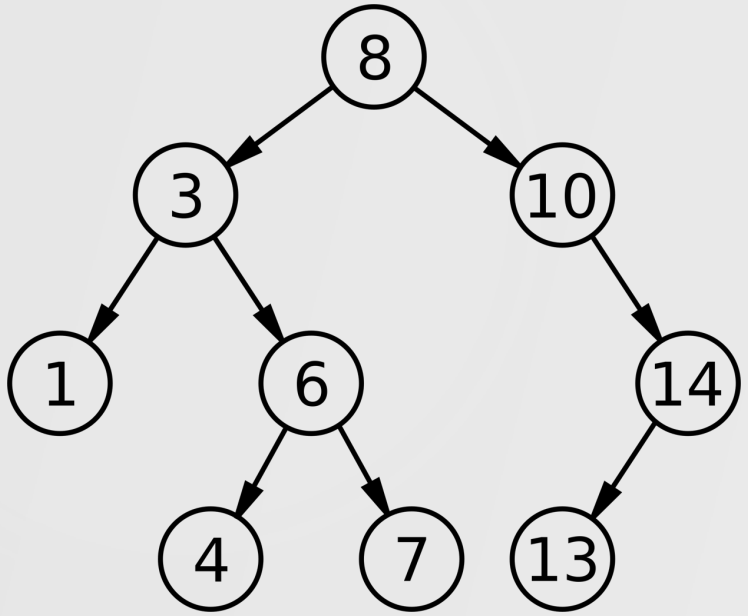
Max-Priority Queue Operations

- **Max-priority queue** supports the following operations:
  1. HEAP-MAXIMUM(A): returns element of A with largest key.
  2. HEAP-EXTRACT-MAX(A): removes and returns element of A with the largest key.
  3. HEAP-INCREASE-KEY(A, i, k): increases $A[i]$ to k. Assume $k \geq A[i]$.
  4. MAX-HEAP-INSERT(A, k): inserts element with value k into A.
- **Time complexity:**
  - HEAP-MAXIMUM(A) takes $\Theta(1)$ time
  - The other three take $O(\log n)$ time

Review: Binary Search Trees

Binary Search Trees

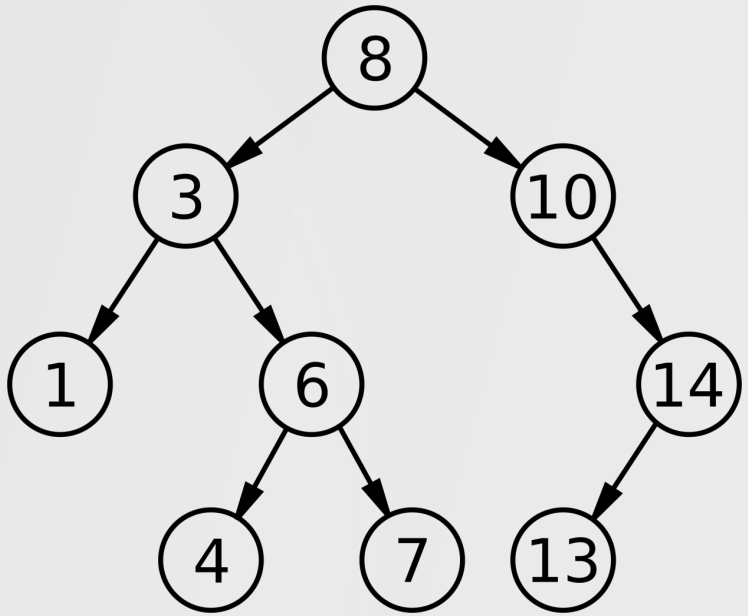
- ***Binary search trees (BST)*** are an important data structure for dynamic sets.
- Stored keys in a BST must satisfy the ***BST property***:
  - If y is in x's left subtree, then $y.key \leq x.key$.
  - If y is in x's right subtree, then $y.key \geq x.key$.
- **Example:**



Implementing A Binary Search Tree

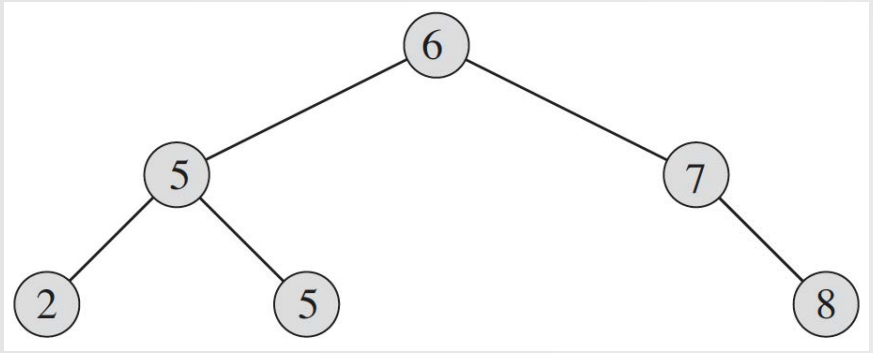
- A BST T is represented by a linked data structure, with each node being an object.
  - $T.root$ points to the root of tree T.
  - Each node contains attributes key, $left$, $right$, and $parent$ (or just p).
  - For an empty child or parent, the attribute contains the value NIL.
- Empty children are called ***external nodes***, and the original nodes are called ***internal nodes***.

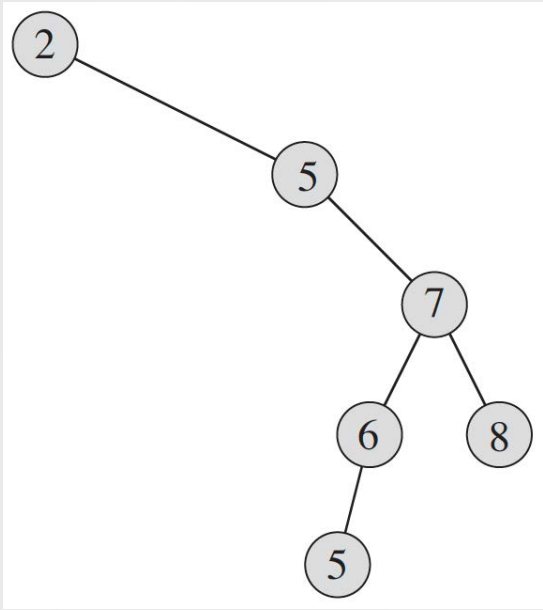
Example:



Height of a Binary Search Tree

- The height of a **tree** is the length of the longest root-to-leaf path.
- Consider a size-n **binary search tree** of height h.
- The height h is
  - $\Theta(n)$ in the worst case and
  - $\Theta(\log n)$ in the best case (***balanced***)
- **Note:** on the same set of nodes/keys, there could be many binary trees of different heights.
- **Example:**



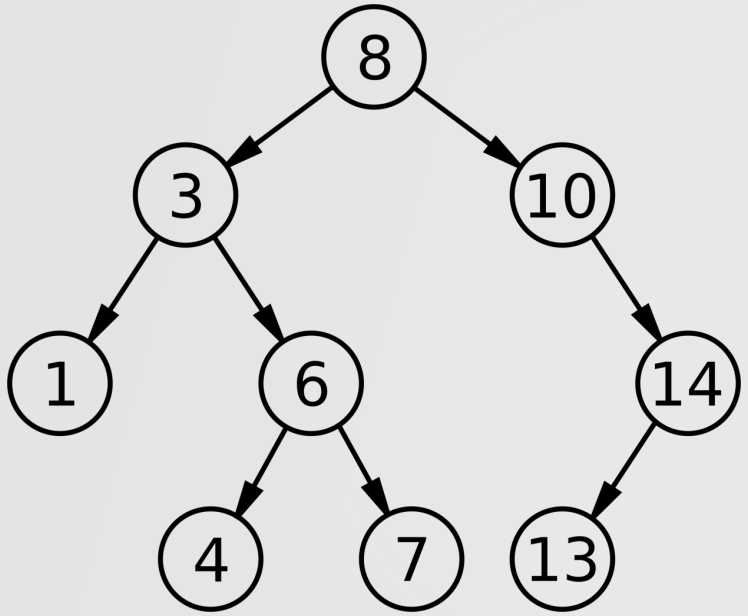


Review: Binary Search Trees

Binary Tree Traversals

Binary Tree Traversals (1/3)

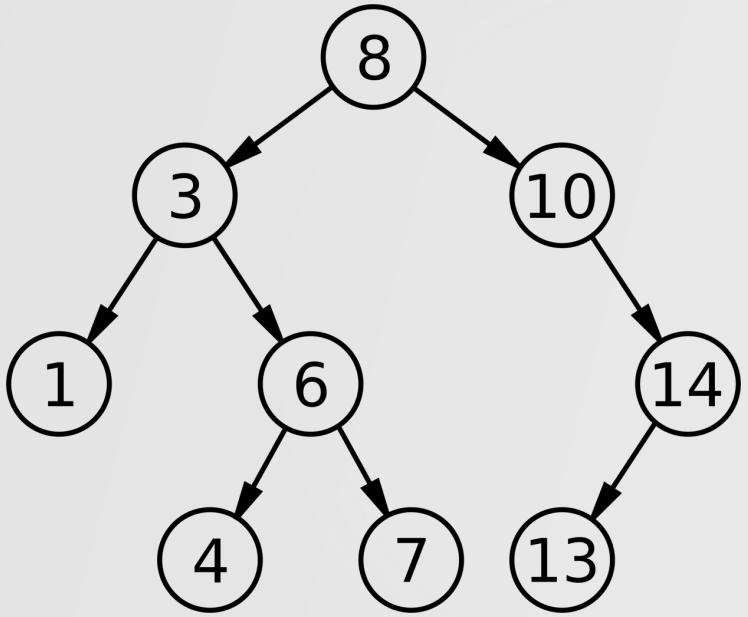
- ***Inorder tree walk***: print out the left subtree first, then the root, and finally the right subtree (print all the keys in sorted order)
- **Example:** INORDER-TREE-WALK($T.root$)




```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

Binary Tree Traversals (2/3)

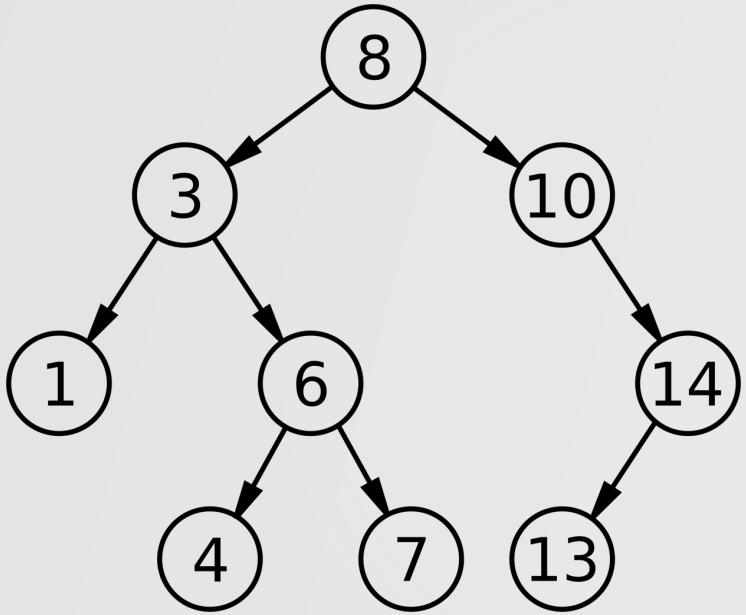
- ***Preorder tree walk***: print out the root first, then the left subtree, and finally the right subtree
- **Example:** PREORDER-TREE-WALK($T.root$)




```
PREORDER-TREE-WALK(x)
1  if x ≠ NIL
2       print x.key
3       PREORDER-TREE-WALK(x.left)
4       PREORDER-TREE-WALK(x.right)
```

Binary Tree Traversals (3/3)

- ***Postorder tree walk***: print out the left subtree first, then the right subtree, and finally the root
- **Example:** POSTORDER-TREE-WALK(*T.root*)




```
POSTORDER-TREE-WALK(x)
1  if x ≠ NIL
2      POSTORDER-TREE-WALK(x.left)
3      POSTORDER-TREE-WALK(x.right)
4      print x.key
```

Analysis of Binary Tree Traversals

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

```
PREORDER-TREE-WALK(x)
1  if x ≠ NIL
2      print x.key
3      PREORDER-TREE-WALK(x.left)
4      PREORDER-TREE-WALK(x.right)
```

```
POSTORDER-TREE-WALK(x)
1  if x ≠ NIL
2      POSTORDER-TREE-WALK(x.left)
3      POSTORDER-TREE-WALK(x.right)
4      print x.key
```

- **Running time:**

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n_L) + T(n_R) + 1, & n > 0 \end{cases}$$

- Solves to $T(n) \in \Theta(n)$.*

** See Theorem 12.1 on p.288 of the textbook for a formal proof.*

Review: Binary Search Trees

Querying a Binary Search Tree

Querying a Binary Search Tree

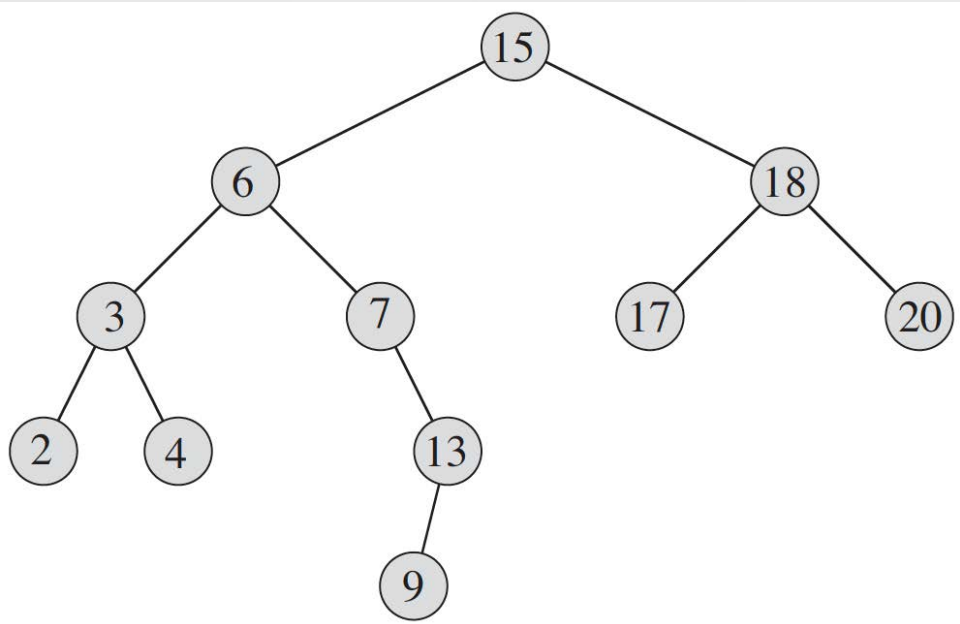
- Search for a particular key stored in a binary search tree (BST):
  1) **Searching:** TREE-SEARCH(x, k)
  2) **Find minimum:** TREE-MINIMUM(x)
  3) **Find maximum:** TREE-MAXIMUM(x)
  4) **Find successor:** TREE-SUCCESSOR(x)
  5) **Find predecessor:** TREE-PREDECESSOR(x)
     - TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.
- The time complexity of every operation is $O(h)$ on any BST of height h.

Searching

- TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else
5              x = x.right
6  return x
```

Example: TREE-SEARCH($T.root$, 4)



- **Time complexity:** $T(n) \in O(h)$

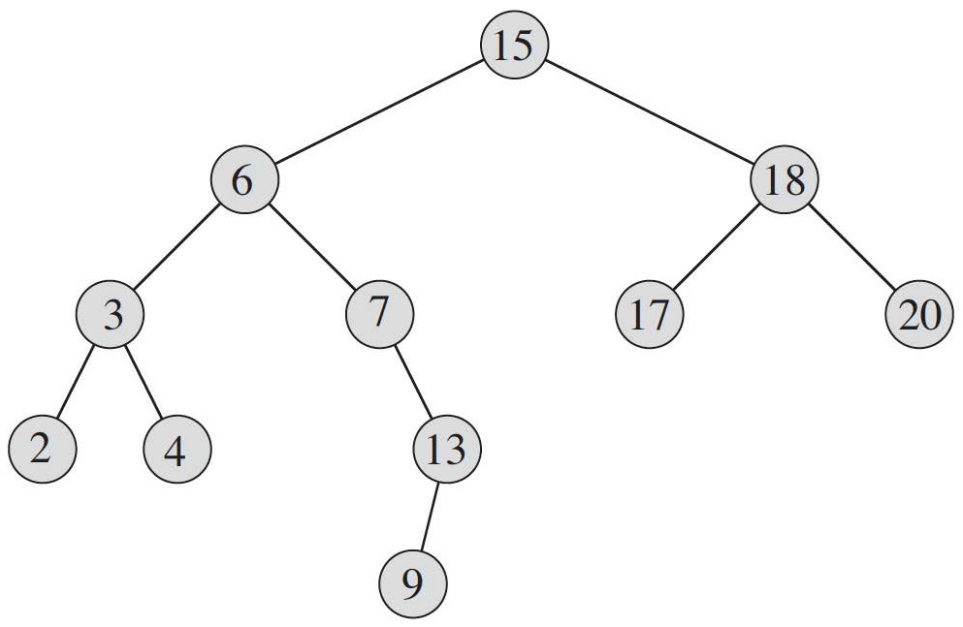
Find Minimum and Maximum

- The **BST property** guarantees that
  - the minimum key of a BST is located at the leftmost node, and
  - the maximum key of a BST is located at the rightmost node.
- Assume input $x \neq \text{NIL}$.

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Example: TREE-MINIMUM($T.root$)

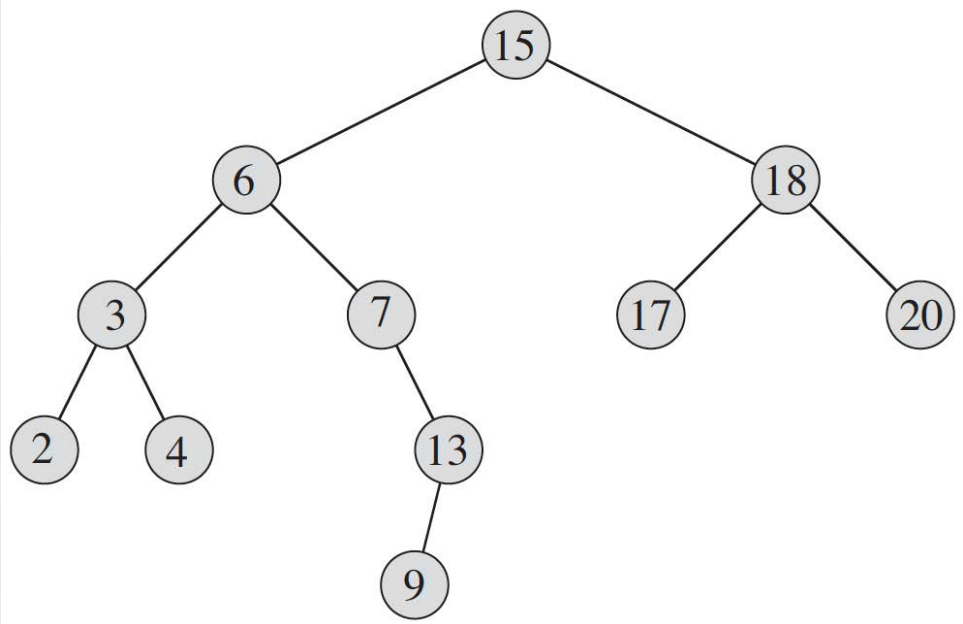


- **Time complexity:** $T(n) \in O(h)$.

Successor and Predecessor

- Given a node in a binary search tree, find its ***successor***/***predecessor*** in the sorted order determined by an inorder tree walk.
- If all keys are distinct,
  - The ***successor*** of a node x is the node with the smallest key $> x.key$.
  - The ***predecessor*** of a node x is the node with the largest key $< x.key$.
- If x has the largest key in the BST, then x's **successor** is NIL.
- If x has the smallest key in the BST, then x's **predecessor** is NIL.

Example:



Inorder tree walk: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

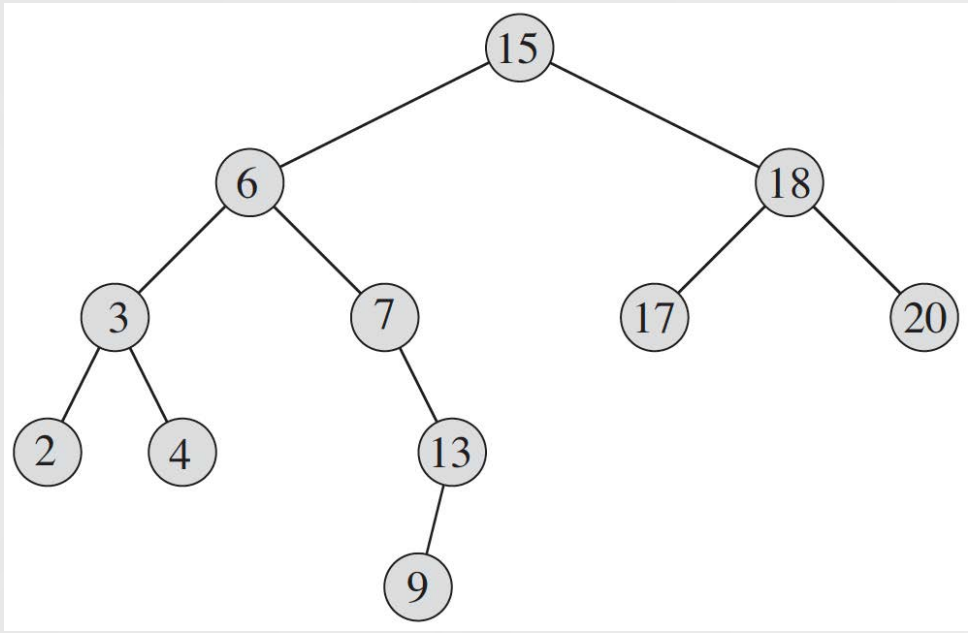
Find Successor (1/3)

Case 1: x has a non-empty right subtree.

- The successor of x is the node with the minimum key in its right subtree.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

Example:



Inorder tree walk: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

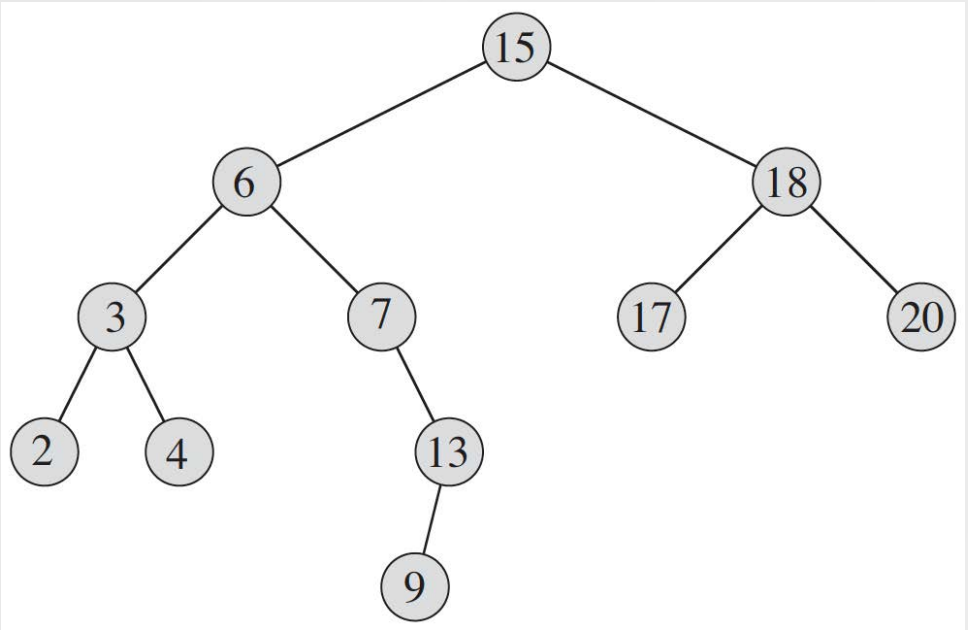
Find Successor (2/3)

Case 2: The right sub-tree of x is empty

- The successor of x, say y, is the first ancestor for which x is in its left subtree. (x is the maximum in y's left subtree)

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

Example:



Inorder tree walk: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Find Successor (3/3)

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

- **Time complexity:**
  - Either visit nodes on a path down the tree:
    - TREE-MINIMUM: $O(h)$
  - Or visit nodes on a path up the tree:
    - **while** loop: $O(h)$

  Thus, $T(n) \in O(h)$.

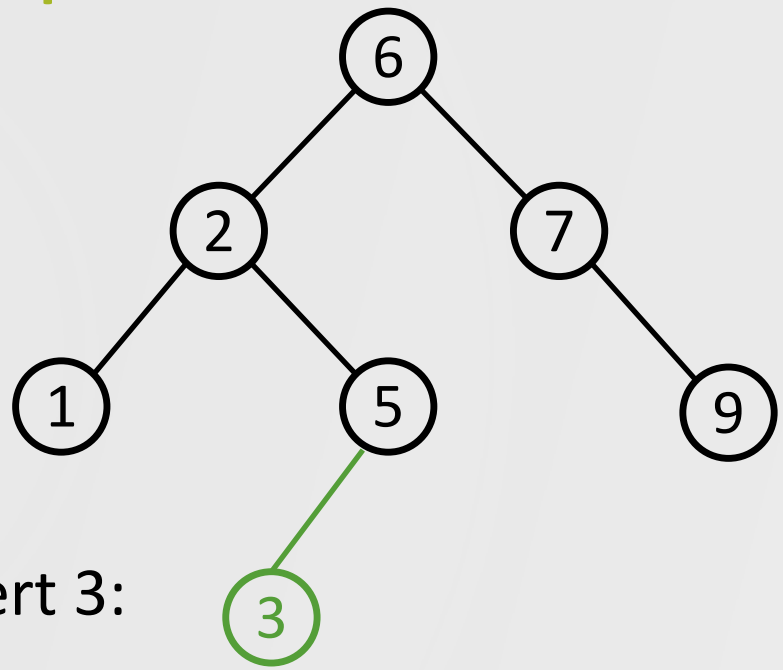
Review: Binary Search Trees

Insertion

Insertion (1/2)

- ***Insertion*** and ***deletion*** allows the dynamic set represented by a BST to change. The **BST property** must hold after the change.
- To insert a new value v into the BST, procedure TREE-INSERT takes a node z with
  - $z.key = v$
  - $z.left = \text{NIL}$
  - $z.right = \text{NIL}$

Example:

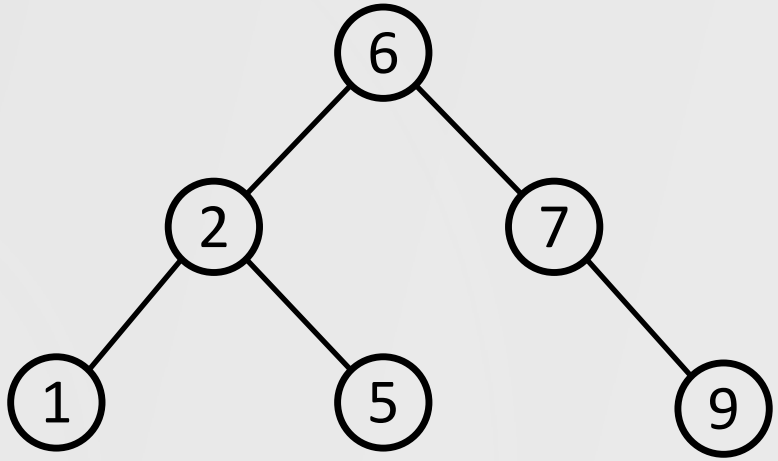


Insert 3:

Insertion (2/2)

```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z   // Tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

Example: Insert 3 to the following BST



- **Time complexity:** $T(n) \in O(h)$

Sorting Using BST

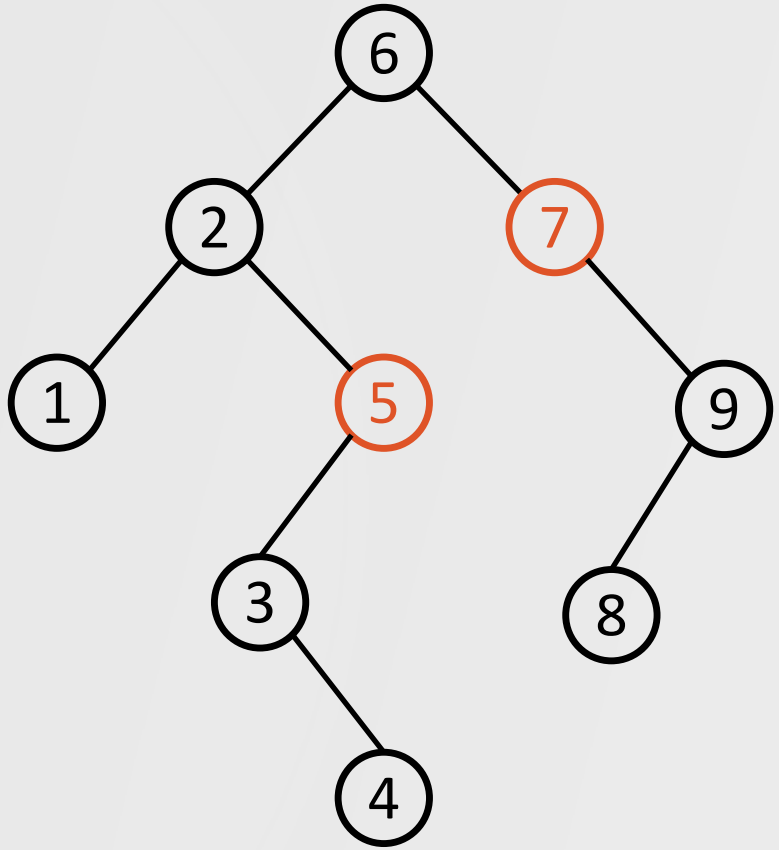
- To sort a given list of n keys, we can
  1. Make n TREE-INSERT calls. - $O(nh)$
  2. Call INORDER-TREE-WALK. - $O(n)$
- **Example:** Sort a sequence of 7 numbers $\langle 16, 4, 3, 9, 1, 35, 23 \rangle$.
  1) Initialize $T = \text{NIL}$
  2) TREE-INSERT(T, 16)
  3) TREE-INSERT(T, 4)
  4) TREE-INSERT(T, 3)
  5) TREE-INSERT(T, 9)
  6) TREE-INSERT(T, 1)
  7) TREE-INSERT(T, 35)
  8) TREE-INSERT(T, 23)
  9) INORDER-TREE-WALK($T.root$)

Review: Binary Search Trees

Deletion

Three Cases of Deletion

- The **BST property** must hold after **deletion**. **Example:**
- To delete a node z from BST T:
  - **Case 1:** z has no left child
    - **E.g.:** Delete 7
  - **Case 2:** z has no right child
    - **E.g.:** Delete 5
  - **Case 3:** z has two children
    - Will discuss shortly

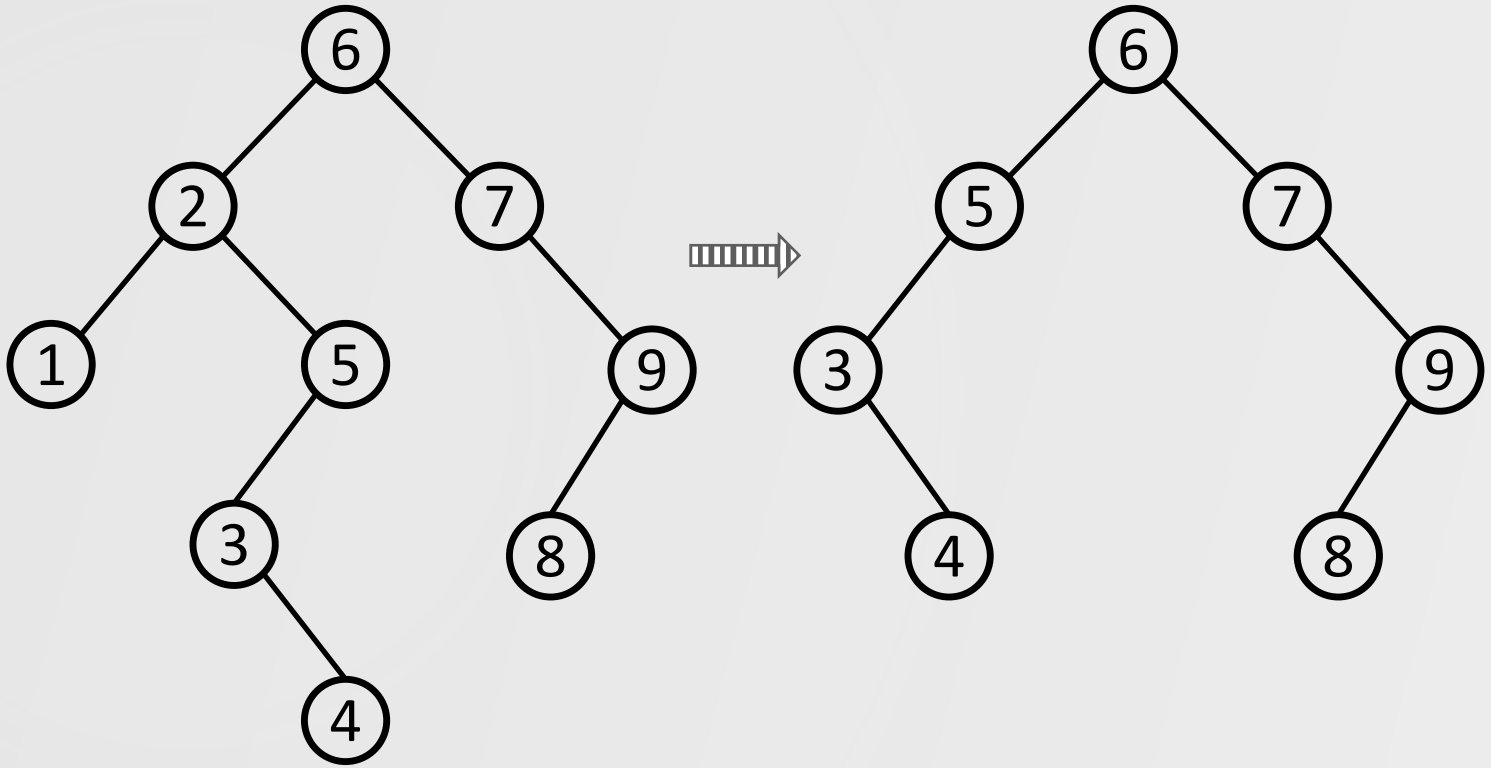


Deletion Cases 1&2

- TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

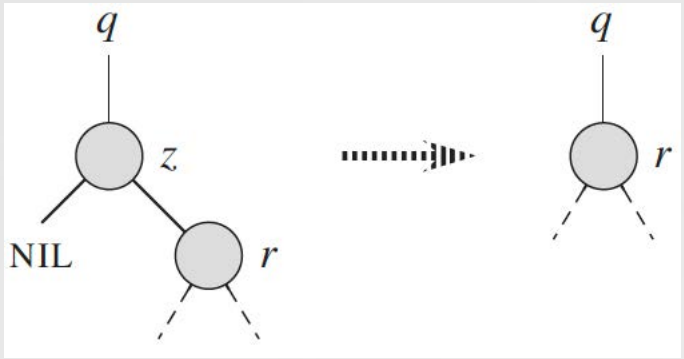
Example: TRANSPLANT(T, u, v) where $u.key = 2$ and $v.key = 5$



Deletion Cases 1

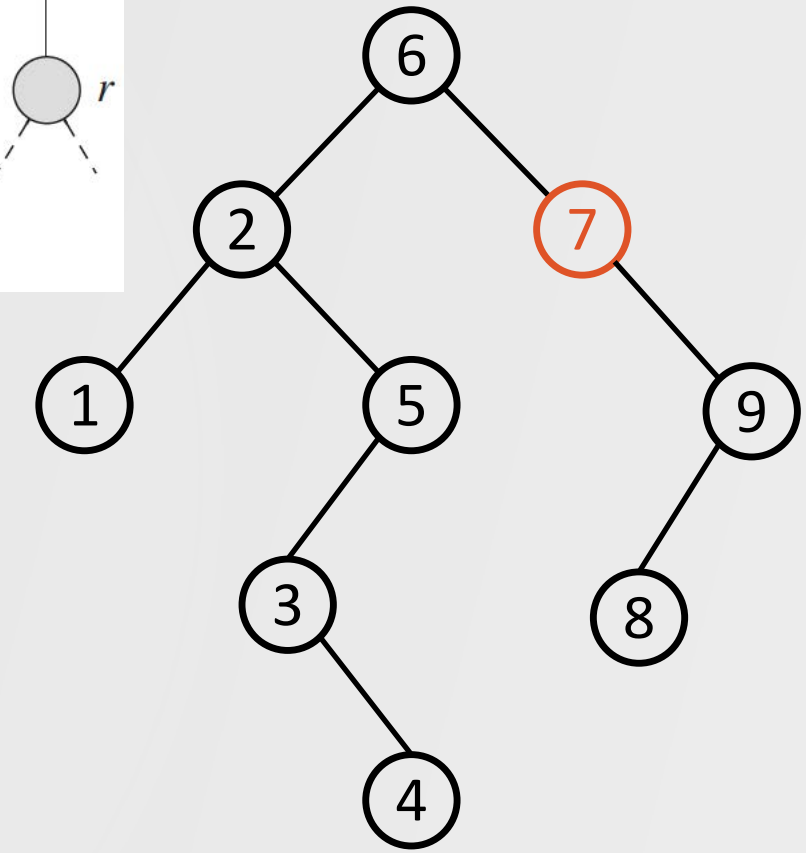
```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

- **Case 1:** z has no left child



- **Example:**

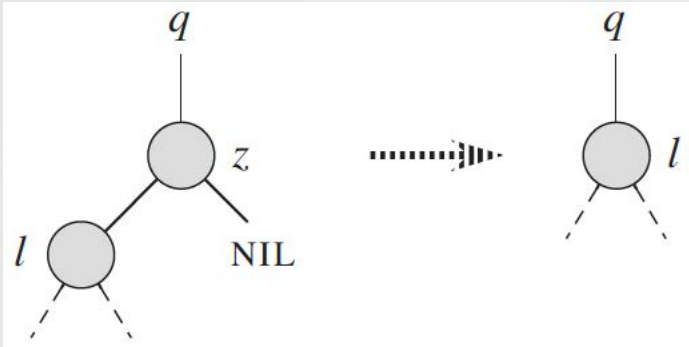
  Delete 7



Deletion Cases 2

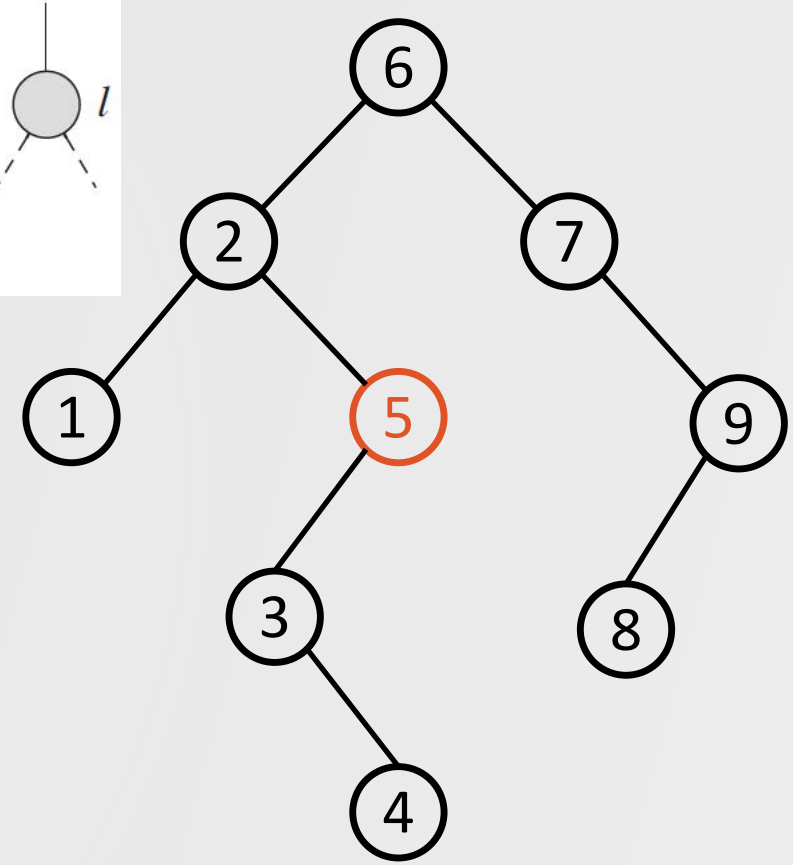
```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

- **Case 2:** z has no right child



- **Example:**

  Delete 5



Deletion Case 3 (1/3)

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Case 3: z has two children

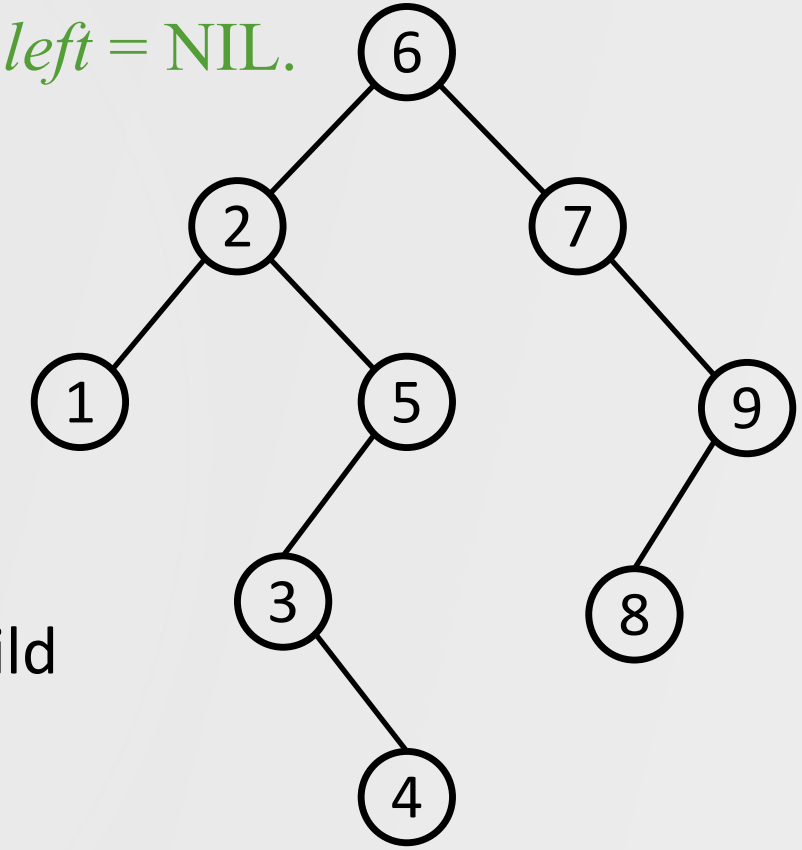
1. Find z's successor y: TREE-MINIMUM($z.right$).
   - We must have $y.left = \text{NIL}$.
2. Replace z by y.

- **Case 3.1:**

  y is z's right child
  - **E.g.:** Delete 6

- **Case 3.2:**

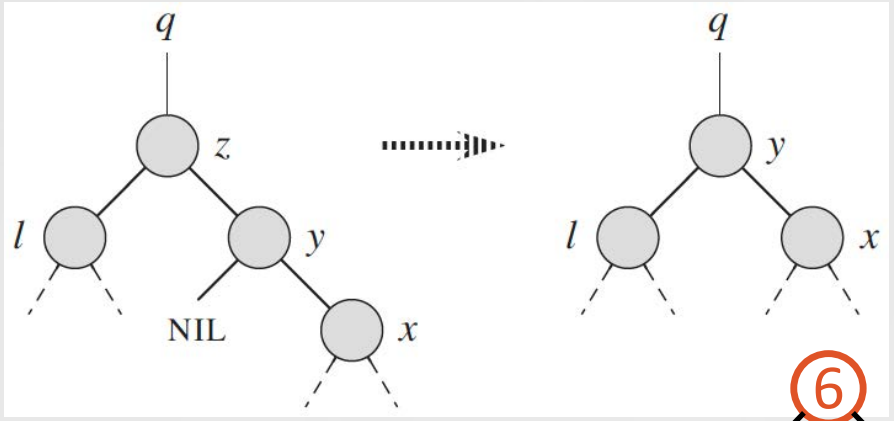
  y is not z's right child
  - **E.g.:** Delete 2



Deletion Case 3 (2/3)

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

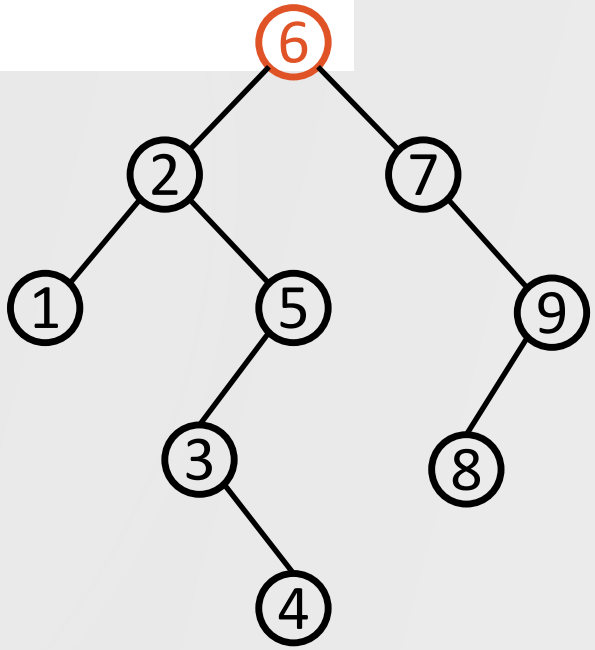
- **Case 3.1:** y is z's right child



- **Example:**

  Delete 6

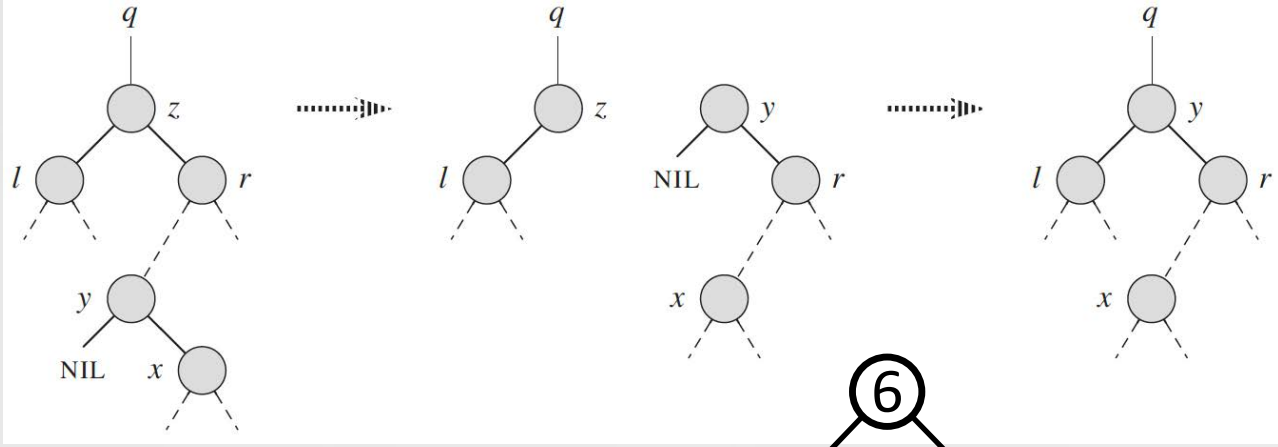
  ($y.key$ = 7)



Deletion Case 3 (3/3)

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

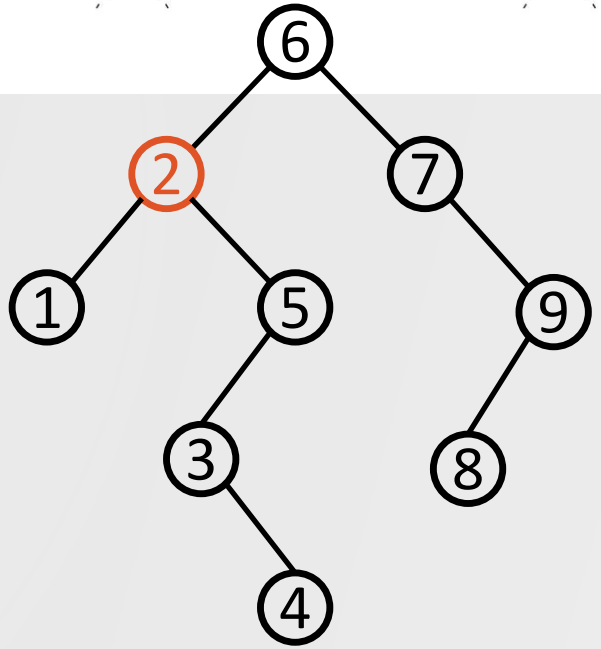
- **Case 3.2:** y is not z's right child



- **Example:**

  Delete 2

  ($y.key$ = 3)



Deletion Time Complexity

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

Time complexity:

- TREE-MINIMUM: $O(h)$
- Everything else: $O(1)$

In total, $T(n) \in O(h)$.

Balanced BST: Red-Black Trees

Balanced BST

- *Recall*: To sort a given list of n keys, we can
  1. Make n TREE-INSERT calls. - $O(nh)$
  2. Call INORDER-TREE-WALK. - $O(n)$
- In the worst case, $h \in \Theta(n)$

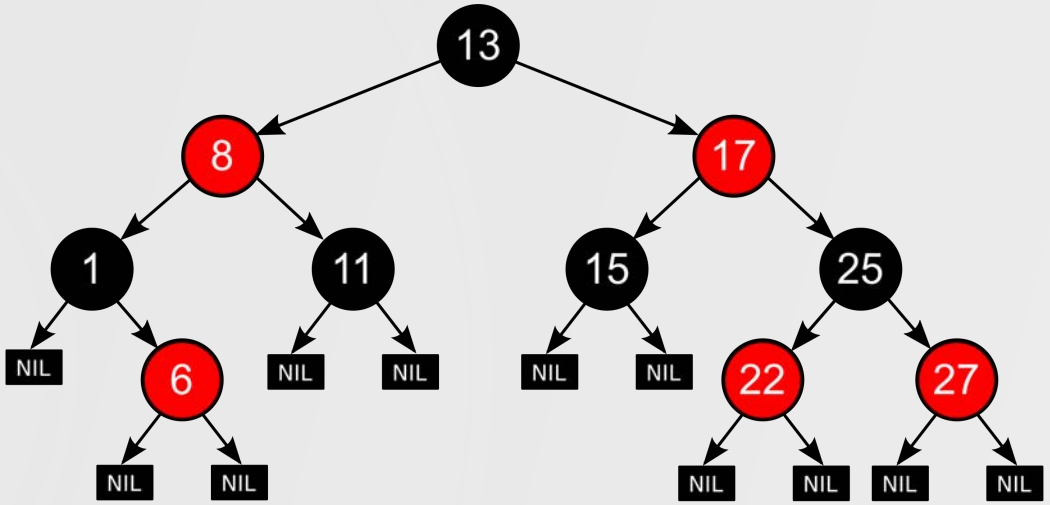
  $\Longrightarrow$ Sorting using BST takes $O(n^2)$ time.
- Whereas we can sort already in $O(n \log n)$ time.
- **Observation:** If we maintain the tree of height $O(\log n)$, then sorting via BST would take $O(n \log n)$ time.
- For that end, we need a ***balanced BST***: $h \in O(\log n)$

Balanced BST: Red-Black Trees

Properties of Red-Black Trees

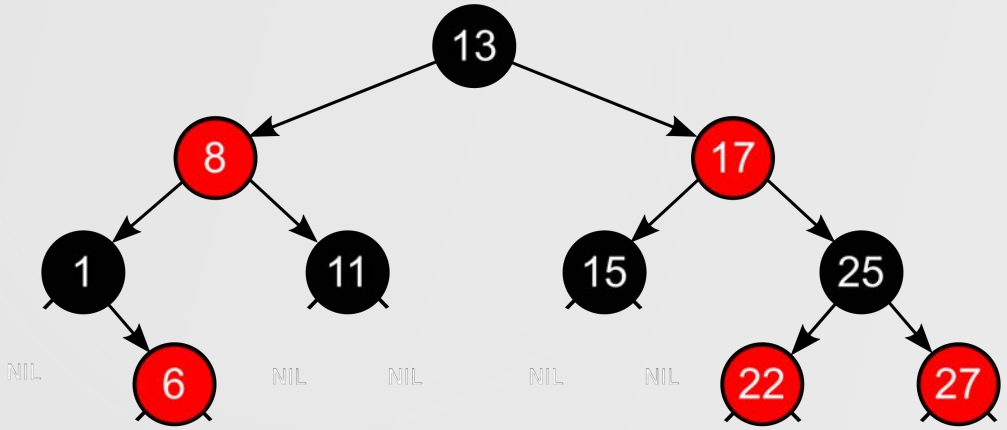
Properties of Red-Black Trees

- A ***red-black tree (RB-tree)*** is a BST with one extra bit of storage per node: its *color*, which can be either **RED** or **BLACK**.
- ***Red-black properties***:
  1) Every node is either **red** or **black**.
  2) The root is **black**.
  3) Every leaf (NIL) is **black**.
  4) If a node is **red**, then both its children are **black**.
  5) Each root-to-leaf simple path has the same number of **black** nodes. (***Black-heights*** matter!)
- **Example:**



Implementing RB-Trees

- Each node of a RB-tree contains the attributes $color$, key, $left$, $right$, and p.
- We use a single sentinel, $T.nil$, for all the leaves and the root's parent.
- $T.nil.color$ is **black,** and we don't care about the key in $T.nil$.
- ***Black-height*** of a node x: $bh(x)$ is the number of black nodes (including NIL) on the x-to-leaf path, not counting x.
- **Example:**



Height of a RB-Tree (1/2)

Lemma: A RB-tree with n internal nodes has height $h \leq 2\log(n+1)$.

Proof. Consider the longest root-to-leaf path P in the RB-tree.

- There are $h+1$ nodes on P.
- P has no two consecutive red nodes (due to property 4).

$$\Longrightarrow \#\text{ black nodes on } P \geq h/2$$

- **Claim:** The first $h/2$ layers in the tree are full.
  - Can be proved by induction.*

➔ *Continued on next slide...*

**See the complete proof of Lemma 13.1 on p.309 of the textbook.*

Height of a RB-Tree (2/2)

Lemma: A RB-tree with n internal nodes has height $h \leq 2\log(n+1)$.

Proof. Consider the longest root-to-leaf path P in the RB-tree. (cont'd)

- The first $h/2$ layers in the tree are full.
- Thus,

$$n \geq 1 + 2 + 2^2 + \cdots + 2^{\frac{h}{2}-1} = 2^{\frac{h}{2}} - 1$$

$$\Rightarrow \log(n+1) \geq \frac{h}{2}$$

$$\Rightarrow h \leq 2\log(n+1)$$

□

Operations in RB-Trees

- These five BST operations
  1) Searching: $\text{TREE-SEARCH}(x, k)$
  2) Find minimum: $\text{TREE-MINIMUM}(x)$
  3) Find maximum: $\text{TREE-MAXIMUM}(x)$
  4) Find successor: $\text{TREE-SUCCESSOR}(x)$
  5) Find predecessor: $\text{TREE-PREDECESSOR}(x)$

  all take $O(\log n)$ time on RB-trees.
- Insertion and deletion are not so easy.

Insertion and Deletion in RB-Trees

- **Insertion** and **deletion** on RB-trees are not so easy.
  - If we insert, what color to make the new node?
  - If we delete, thus removing a node, what color was the node that was removed?

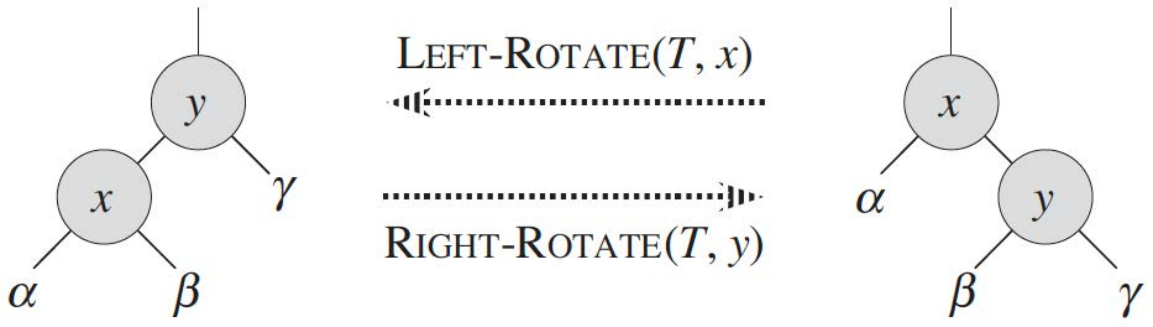
- **Red-black properties**:
  1) Every node is either **red** or **black**.
  2) The root is **black**.
  3) Every leaf (NIL) is **black**.
  4) If a node is **red**, then both its children are **black**.
  5) Each root-to-leaf simple path has the same number of **black** nodes.

Balanced BST: Red-Black Trees

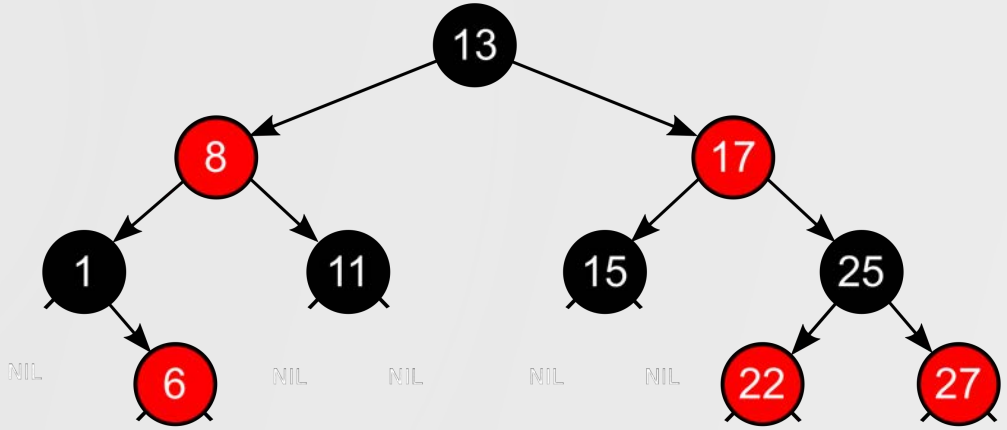
Rotations

Rotations (1/2)

- Only **insertion** and **deletion** will alter the tree structure.
- In order to maintain the RB-tree as a balanced BST and preserve the BST property, we introduce two kinds of rotations:
  - ***Left-rotation***: the old root becomes the left child of the new root.
  - ***Right-rotation***: the old root becomes the right child of the new root.



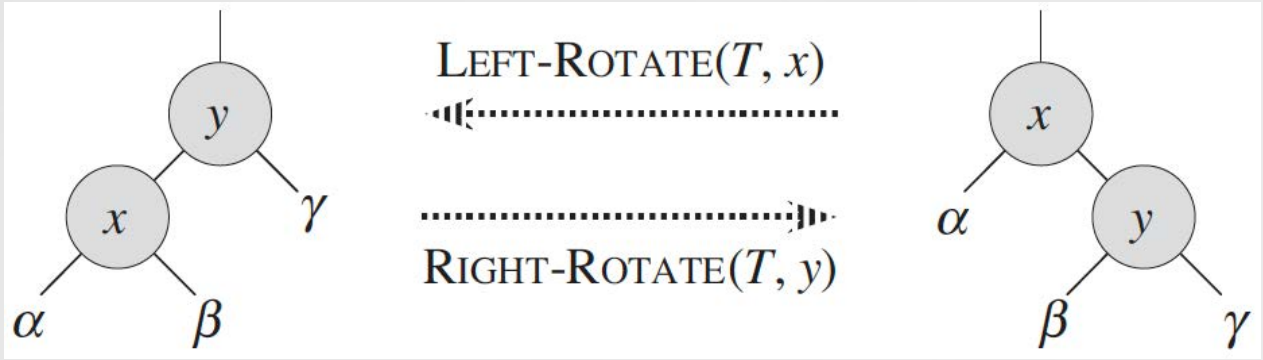
- **Examples:**
  - Do left-rotation on 17
  - Do right-rotation on 8



Rotations (2/2)

```
LEFT-ROTATE(T, x)
1   y = x.right
2   x.right = y.left
3   if y.left ≠ T.nil
4       y.left.p = x
5   y.p = x.p
6   if x.p == T.nil
7       T.root = y
8   elseif x == x.p.left
9       x.p.left = y
10  else x.p.right = y
11  y.left = x
12  x.p = y
```

- For LEFT-ROTATE(T, x), assume x is a node in T and $x.right \neq T.nil$



- RIGHT-ROTATE is symmetric.
- **Time complexity** for both LEFT-ROTATE and RIGHT-ROTATE: $\Theta(1)$

Balanced BST: Red-Black Trees

Insertion

Insertion on A RB-Tree

Red-black properties:

1) Every node is either **red** or **black**.
2) The root is **black**.
3) Every leaf (NIL) is **black**.
4) If a node is **red**, then both its children are **black**.
5) Each root-to-leaf simple path has the same number of **black** nodes.

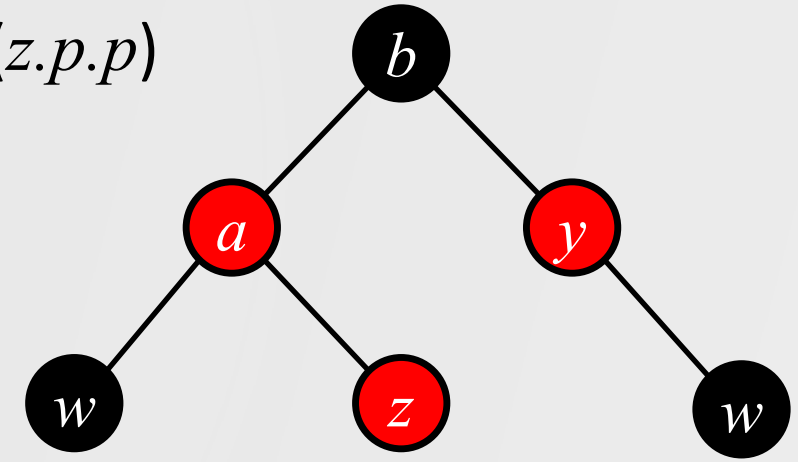
1. Insert node z by TREE-INSERT (insertion for BST) and color it **red**.

   Q: Why color it red initially?

2. Recolor and/or rotate nodes to fix violation.

Relationships of z:

- b: z's grandparent ($z.p.p$)
- a: z's parent ($z.p$)
- y: z's uncle
- w: z's sibling



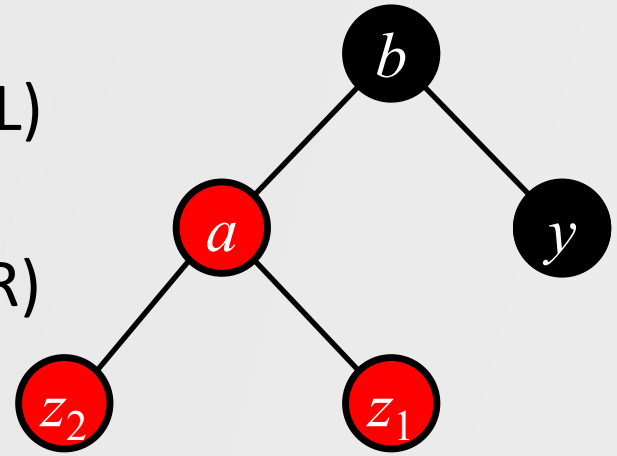
Four Cases of Insertion

Red-black properties:

1) Every node is either **red** or **black**.
2) The root is **black**.
3) Every leaf (NIL) is **black**.
4) If a node is **red**, then both its children are **black**.
5) Each root-to-leaf simple path has the same number of **black** nodes.

Recolor and/or rotate nodes to fix violation.

- **Case 0:** z is the root
  Just color it **black**. Done.
- **Cases 1-3:** $z.p$ is also **red**
  $z.p.p$ must be **black** – why?
  Consider z's uncle, y:
  - **Case 1:** y is **red**
  - **Case 2:** y is **black** (LR/RL)
    - **E.g.:** b - a - z_1
  - **Case 3:** y is **black** (LL/RR)
    - **E.g.:** b - a - z_2

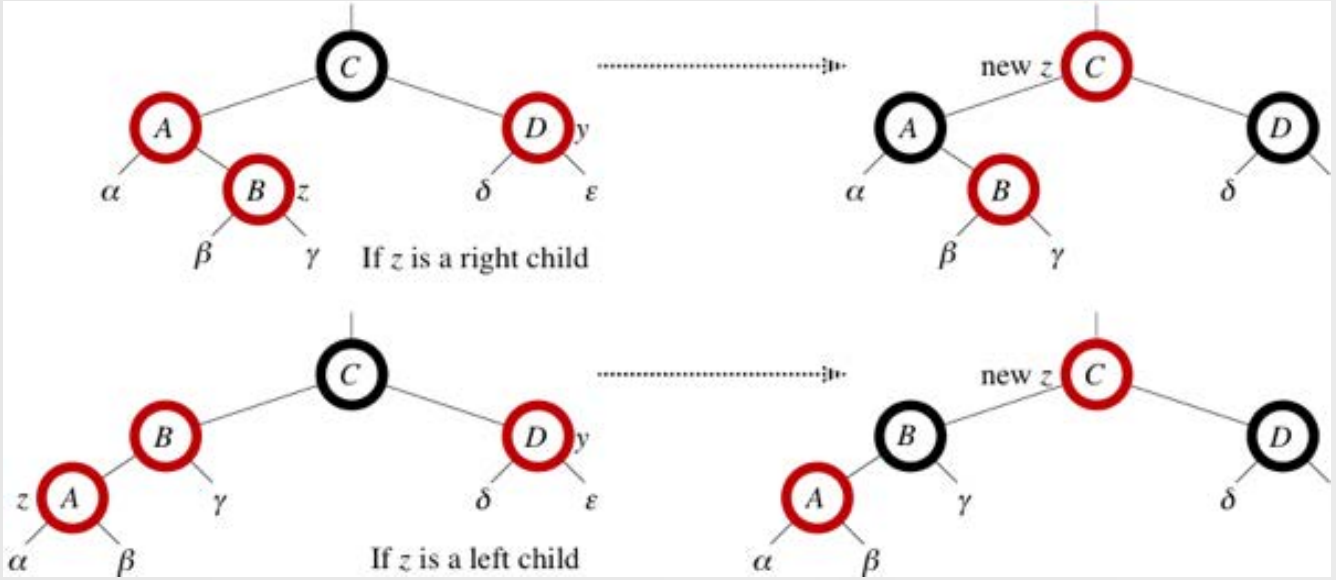


Insertion Case 1

Red-black properties:

1) Every node is either **red** or **black**.
2) The root is **black**.
3) Every leaf (NIL) is **black**.
4) If a node is **red**, then both its children are **black**.
5) Each root-to-leaf simple path has the same number of **black** nodes.

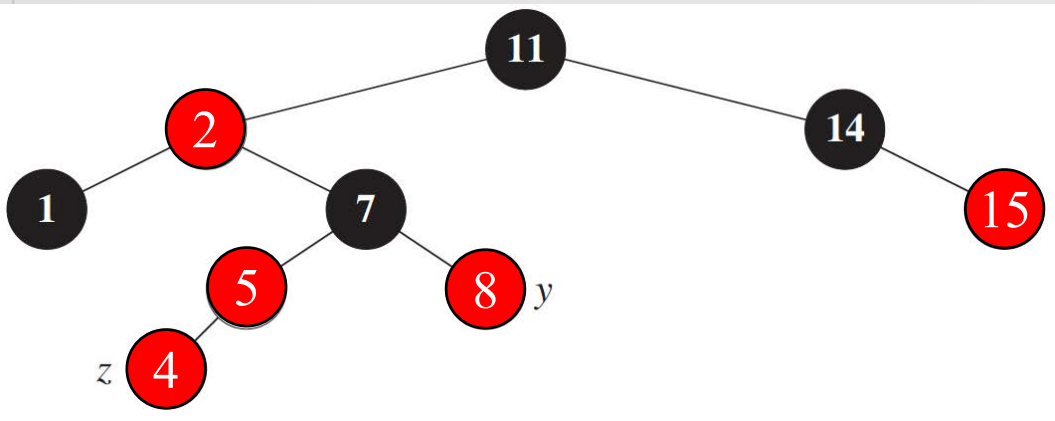
Case 1: z's uncle, y, is **red**



1) Color $z.p$ and y **black**
2) Color $z.p.p$ **red**
3) Recurse up with $z = z.p.p$

Insertion Case 1 Example

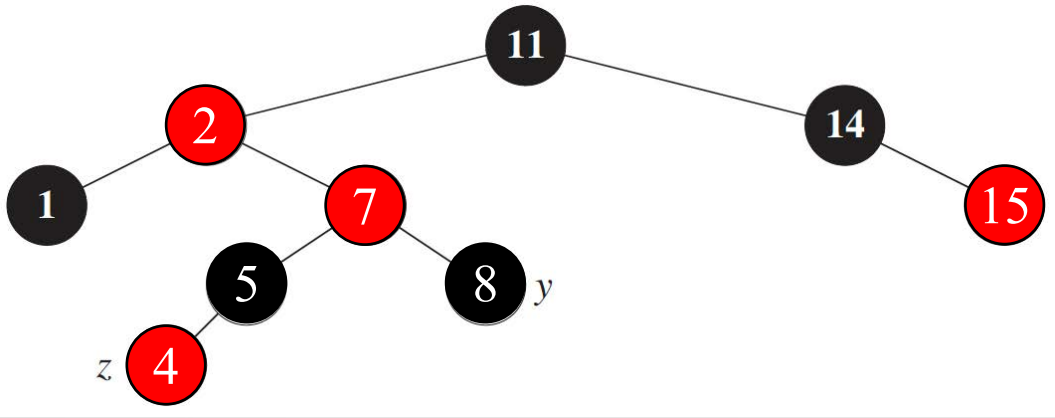
Example: Insert 4



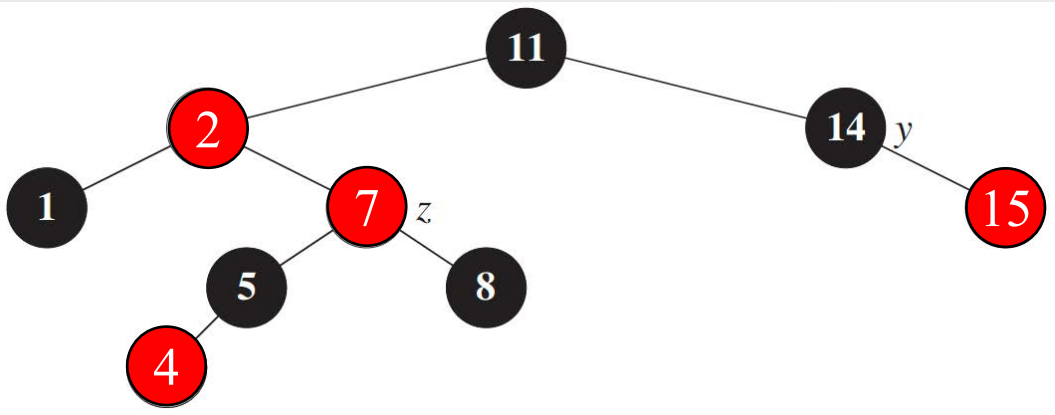
Case 1: z's uncle, y, is **red**

1) Color $z.p$ and y **black**
2) Color $z.p.p$ **red**
3) Recurse up with $z = z.p.p$

Steps 1) – 2):



Step 3):

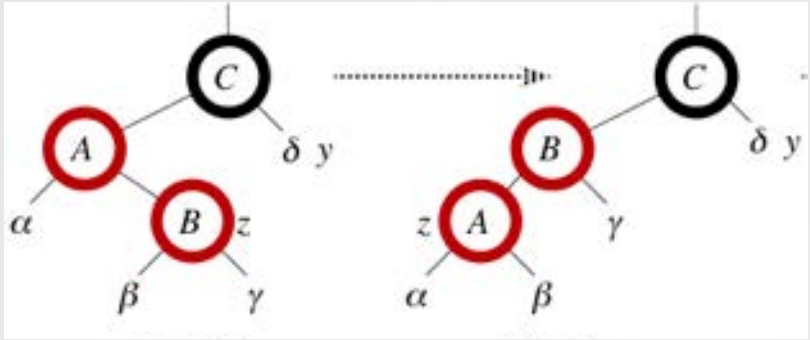


Insertion Case 2

Red-black properties:

1) Every node is either **red** or **black**.
2) The root is **black**.
3) Every leaf (NIL) is **black**.
4) If a node is **red**, then both its children are **black**.
5) Each root-to-leaf simple path has the same number of **black** nodes.

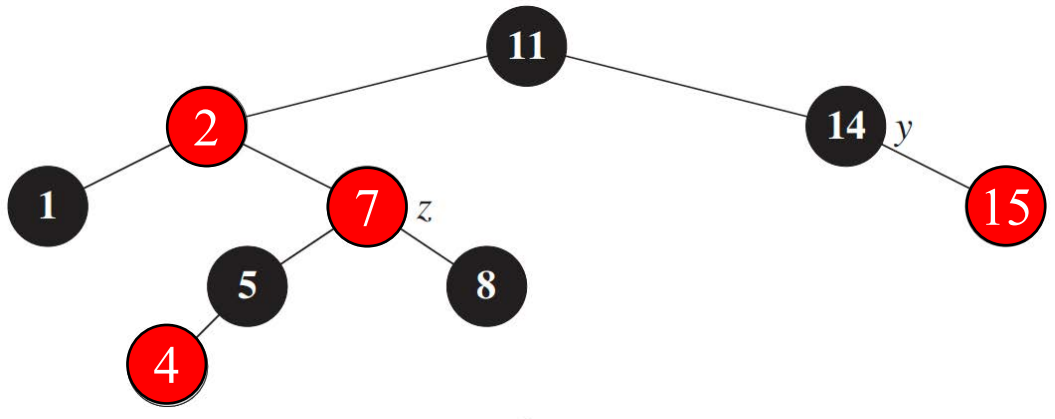
Case 2: z's uncle, y, is **black** (LR/RL)



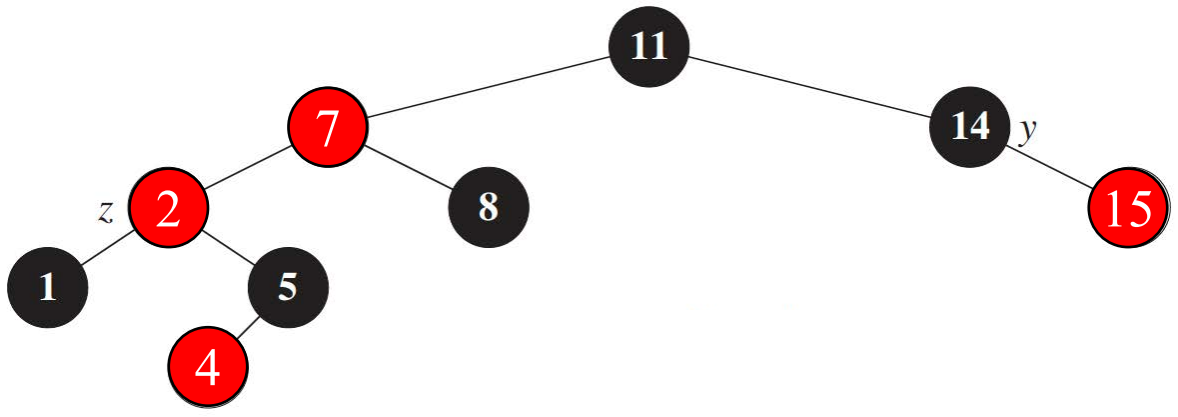
1) Let $z = z.p$
2) Make it **Case 3** by
   - Calling LEFT-ROTATE(T, z) for the LR case
   - Calling RIGHT-ROTATE(T, z) for the RL case
3) Solve **Case 3**

Insertion Case 2 Example

Example: $z.key = 7$



Steps 1) – 2):



Case 2: z's uncle, y, is **black** (LR/RL)

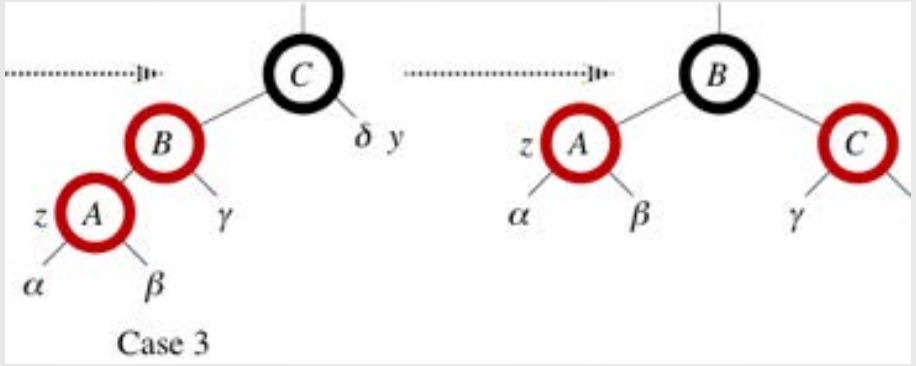
1) Let $z = z.p$
2) Make it **Case 3** by
   - Calling LEFT-ROTATE(T, z) for the LR case
   - Calling RIGHT-ROTATE(T, z) for the RL case
3) Solve **Case 3**: z's uncle, y, is **black** (LL/RR)

Insertion Case 3

Red-black properties:

1) Every node is either **red** or **black**.
2) The root is **black**.
3) Every leaf (NIL) is **black**.
4) If a node is **red**, then both its children are **black**.
5) Each root-to-leaf simple path has the same number of **black** nodes.

Case 3: z's uncle, y, is **black** (LL/RR)



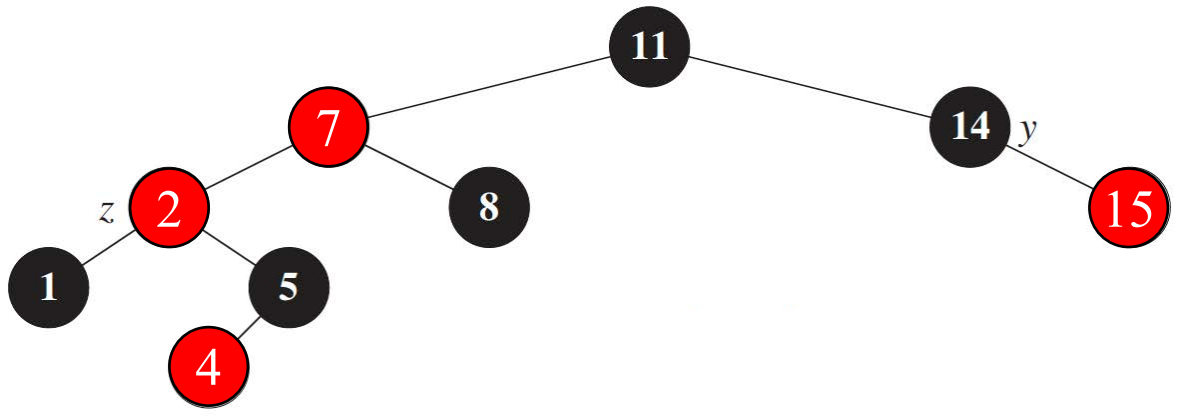
Case 3

1) Flip colors of $z.p.p$ and $z.p$
2) Call
   - RIGHT-ROTATE(T, $z.p.p$) for the LL case
   - LEFT-ROTATE(T, $z.p.p$) for the RR case

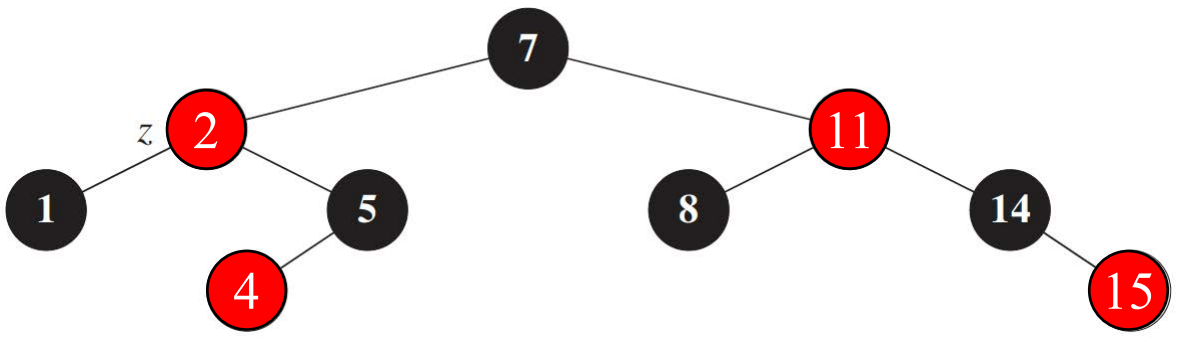
Done.

Insertion Case 3 Example

Example: $z.key = 7$



Steps 1) – 2):



Case 3: z's uncle, y, is **black** (LL/RR)

1) Flip colors of $z.p.p$ and $z.p$
2) Call
   - RIGHT-ROTATE(T, $z.p.p$) for the LL case
   - LEFT-ROTATE(T, $z.p.p$) for the RR case

Running Time of Insertion

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else if z == z.p.right
10                  z = z.p
11                  LEFT-ROTATE(T, z)
12              z.p.color = BLACK
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

In RB-INSERT,

- Lines 1-16: $O(h)$

In RB-INSERT-FIXUP,

- # iterations: $O(h)$

Thus, **running time** of RB-INSERT:

$$T(n) \in O(h)$$

$$\Longrightarrow T(n) \in O(\log n)$$

Exercise: RB-Tree Insertion

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

Balanced BST: Red-Black Trees

Deletion

Deletion on A RB-Tree

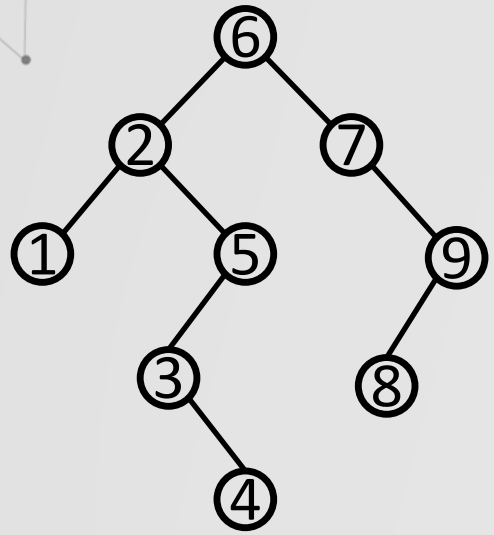
Red-black properties:

1) Every node is either **red** or **black**.
2) The root is **black**.
3) Every leaf (NIL) is **black**.
4) If a node is **red**, then both its children are **black**.
5) Each root-to-leaf simple path has the same number of **black** nodes.

1. Delete a node z by TREE-DELETE (deletion for BST) with a slight modification.
   - **Observation:** TREE-DELETE eventually deletes/moves a node with at least one child missing. - See next slide
   - y: the node either removed from the tree or moved within the tree.
2. Recolor and/or rotate nodes to fix violation.
   - If y was **red**, no violation occurs.
   - If y was **black**, we fix-up with y's child x
     - that moves into y's original position
     - could be **black** NIL

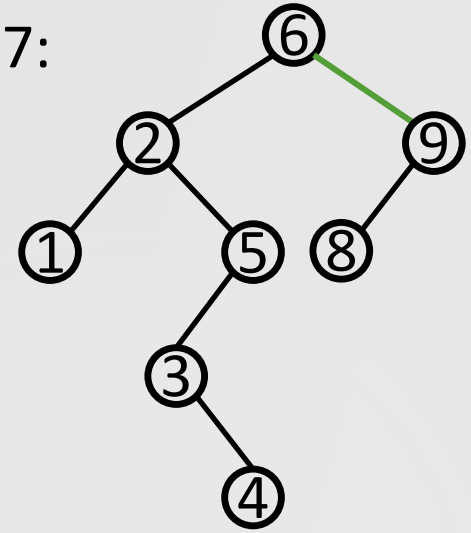
Review: TREE-DELETE

Example:



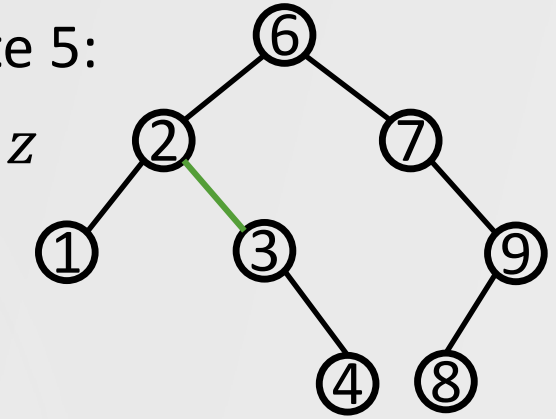
1) Delete 7:

$y = z$



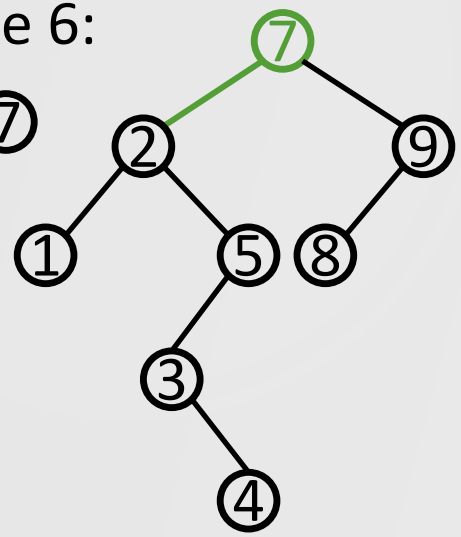
2) Delete 5:

$y = z$



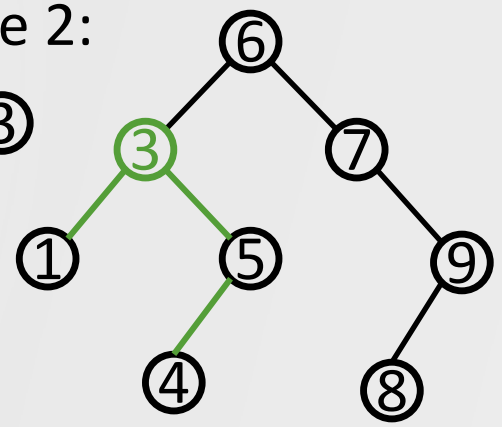
3.1) Delete 6:

$y =$ ⑦



3.2) Delete 2:

$y =$ ③



Cases of Deletion (1/2)

Red-black properties:

1) Every node is either **red** or **black**.
2) The root is **black**.
3) Every leaf (NIL) is **black**.
4) If a node is **red**, then both its children are **black**.
5) Each root-to-leaf simple path has the same number of **black** nodes.

If y was **black**, we fix-up with y's child x.

Recall: x is the node that moves into y's original position.

- **Case 0:** x is the root or x is **red**

  Just color it black. Done.

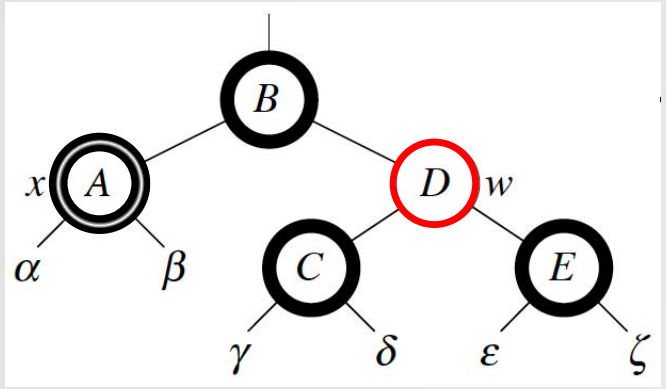
- **Cases 1-4:** x is **black** and not the root
  - x is considered "***doubly black***" (y and x)
  - **Goal:** preserve the number of **black** nodes in each sub-path.

  We look into x's sibling w.

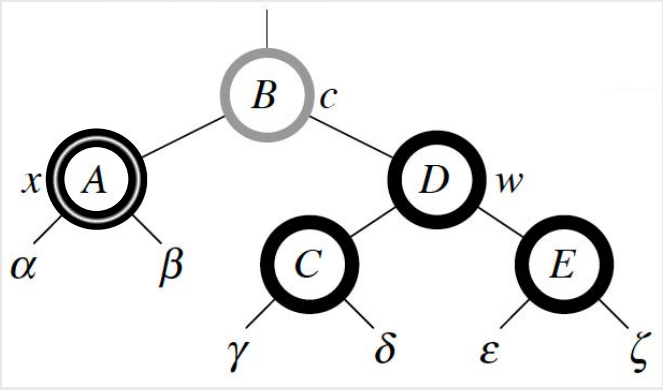
Cases of Deletion (2/2)

Cases 1-4: x is **black** and not the root – x is "***doubly black***". Look into x's sibling w.

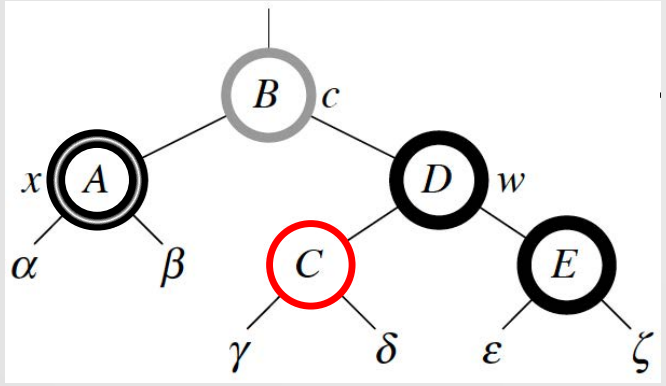
Case 1: w is **red**



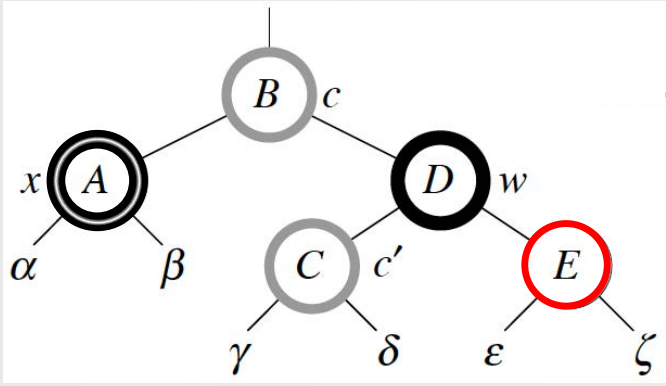
Case 2: w and both its children are all **black**



Case 3: w is **black** and it has exactly ONE **red** child on the opposite side (RL/LR).

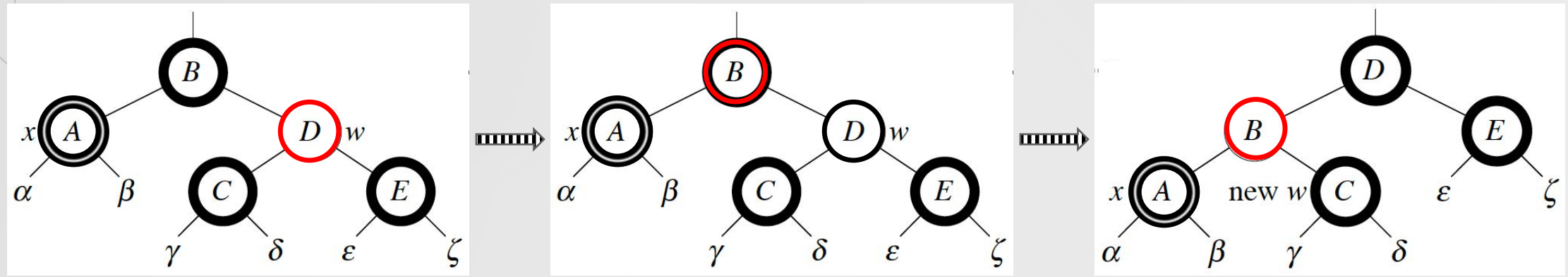


Case 4: w is **black** and its child on the same side (RR/LL) is **red**.



Deletion Case 1

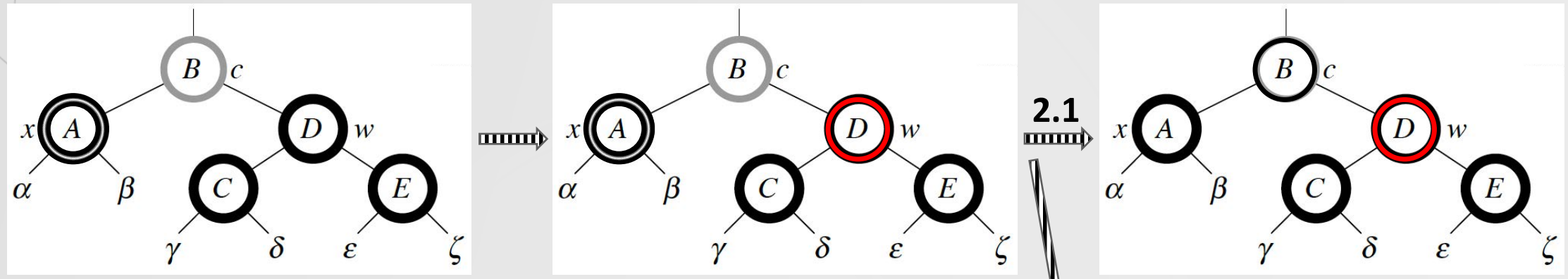
Case 1: w is **red**



1) Color w **black**, color $x.p$ **red**
2) Call LEFT-ROTATE$(T, x.p)$ if x is a left child OR
   Call RIGHT-ROTATE$(T, x.p)$ if x is a right child
3) New w is x's new sibling, which must be **black** – Why?
4) Check **Cases 2**, **3**, and **4**.

Deletion Case 2

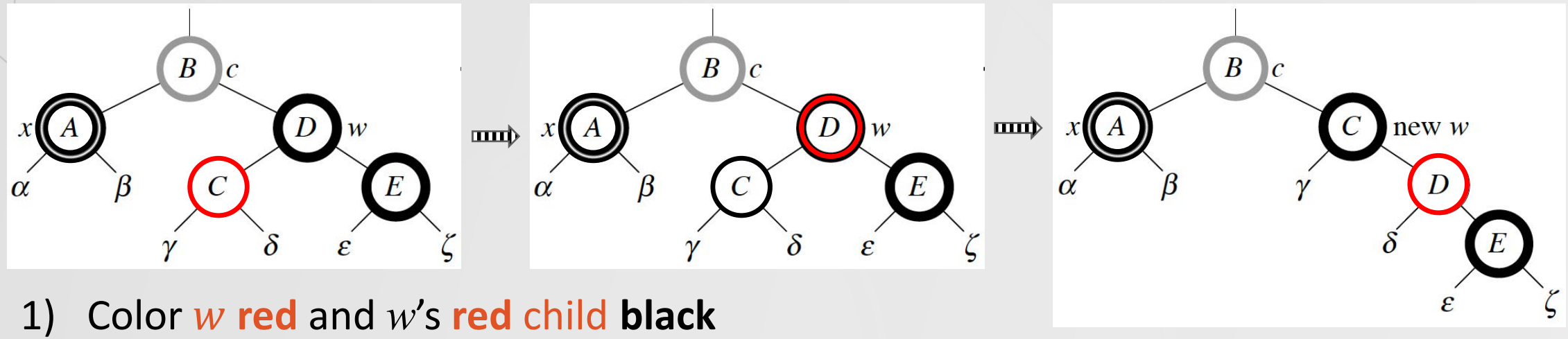
Case 2: w and both its children are all **black**



1) Color w **red**
2) The color of $w.p$ is unknown.
   - **Case 2.1:** $x.p$ is **red** → Color $x.p$ black. Done.
   - **Case 2.2:** $x.p$ is **black** → Recurse up with $x = x.p$

Deletion Case 3

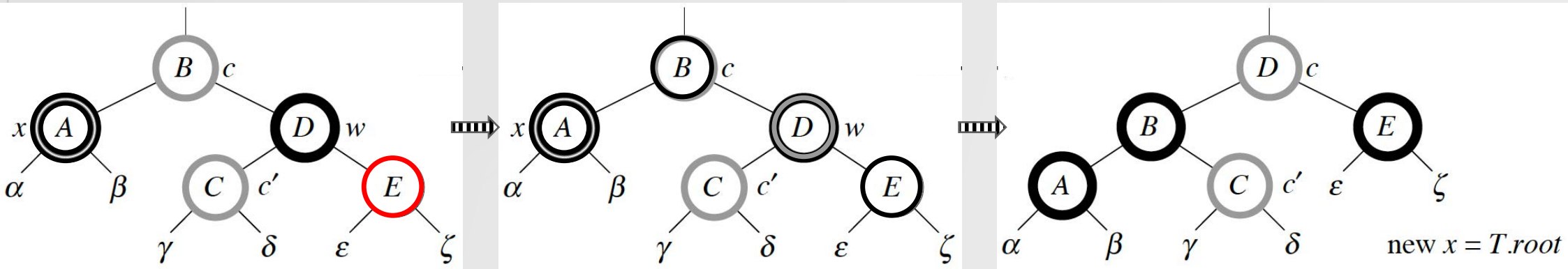
Case 3: w is **black** and it has exactly ONE **red** child on the opposite side (RL/LR).



1) Color w **red** and w's **red** child **black**
2) Make it **Case 4** by calling
   - RIGHT-ROTATE(T, w) for the RL case (w is a right child and $w.left$ is **red**)
   - LEFT-ROTATE(T, w) for the LR case (w is a left child and $w.right$ is **red**)
3) New w is x's new sibling ➔ Check **Case 4**.

Deletion Case 4

Case 4: w is **black** and its child on the same side (RR/LL) is **red**.



1) Give w the color of $x.p$;
2) Color $x.p$ and w's same side **red** child **black**.
3) Call LEFT-ROTATE(T, $x.p$) for the RR case (w is a right child and *w.right* is **red**) OR
   Call RIGHT-ROTATE(T, $x.p$) for the LL case (w is a left child and *w.left* is **red**)

Done.

Running Time of Deletion

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13            x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15            y.right = z.right
16            y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21  if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2       if x == x.p.left
 3            w = x.p.right
 4            if w.color == RED
 5                 w.color = BLACK
 6                 x.p.color = RED
 7                 LEFT-ROTATE(T, x.p)
 8                 w = x.p.right
 9            if w.left.color == BLACK and w.right.color == BLACK
10                 w.color = RED
11                 x = x.p
12            else if w.right.color == BLACK
13                      w.left.color = BLACK
14                      w.color = RED
15                      RIGHT-ROTATE(T, w)
16                      w = x.p.right
17                 w.color = x.p.color
18                 x.p.color = BLACK
19                 w.right.color = BLACK
20                 LEFT-ROTATE(T, x.p)
21                 x = T.root
22       else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

In RB-DELETE,

- Lines 1-21: $O(h)$

In RB-DELETE-FIXUP,

- # iterations only depends on Case 2:

$$O(h)$$

Thus, **running time** of RB-DELETE:

$$T(n) \in O(h)$$

$$\Longrightarrow T(n) \in O(\log n)$$

Exercise: RB-Tree Deletion

- In the Exercise for RB-tree insertion, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.

Thank you!
Questions?