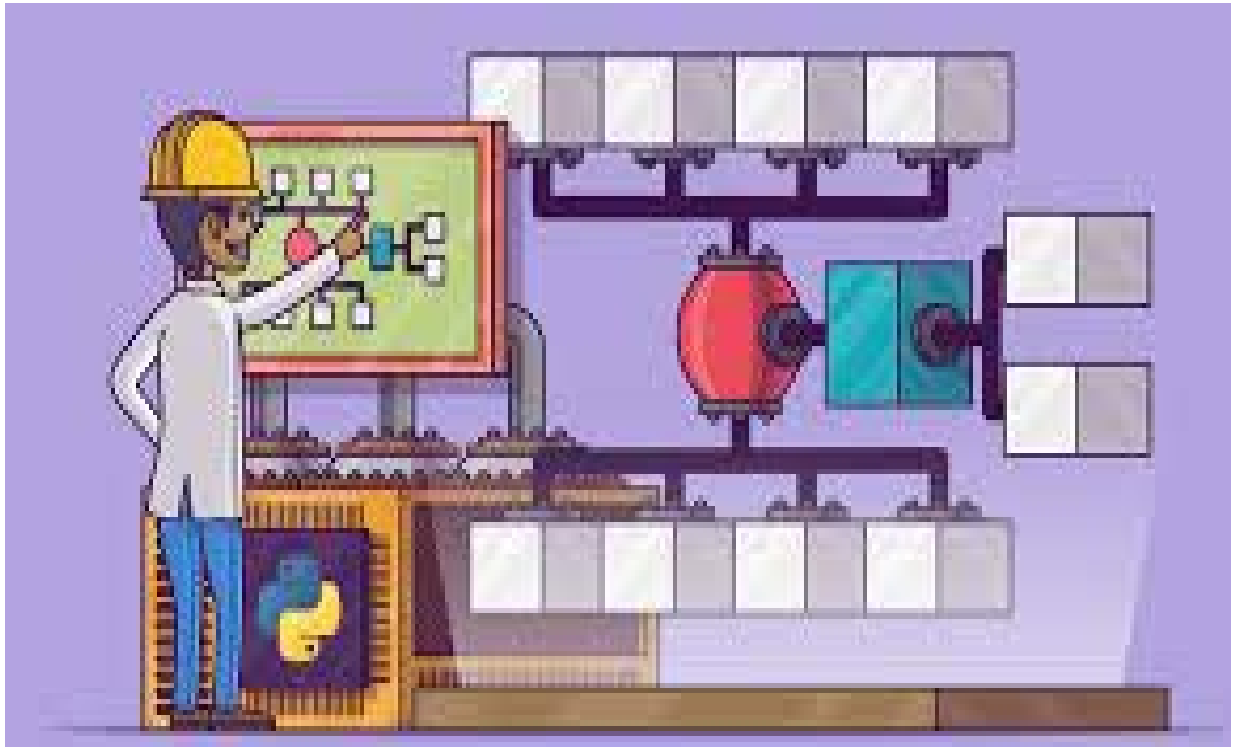
ساختمان داده ها



مدرس:
سمانه حسینی سمنانی

دانشگاه صنعتی اصفهان- دانشکده برق و کامپیوتر

درخت ها

- مفاهیم اولیه
- پیمایش درخت
- درخت دودویی معادل
- پیاده سازی درخت
- درخت جستجوی دودویی
- درخت عبارت
- Heap tree (هرم بیشینه)

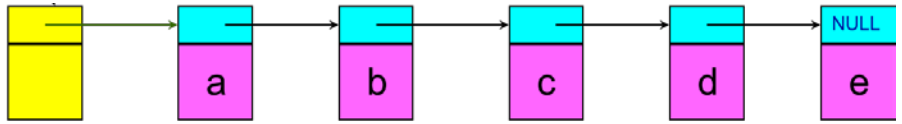
درخت ها

Heap tree (هرم بیشینه)

Priority Queue

- صف اولویت

$p_1, p_2, \ldots\ldots p_n$



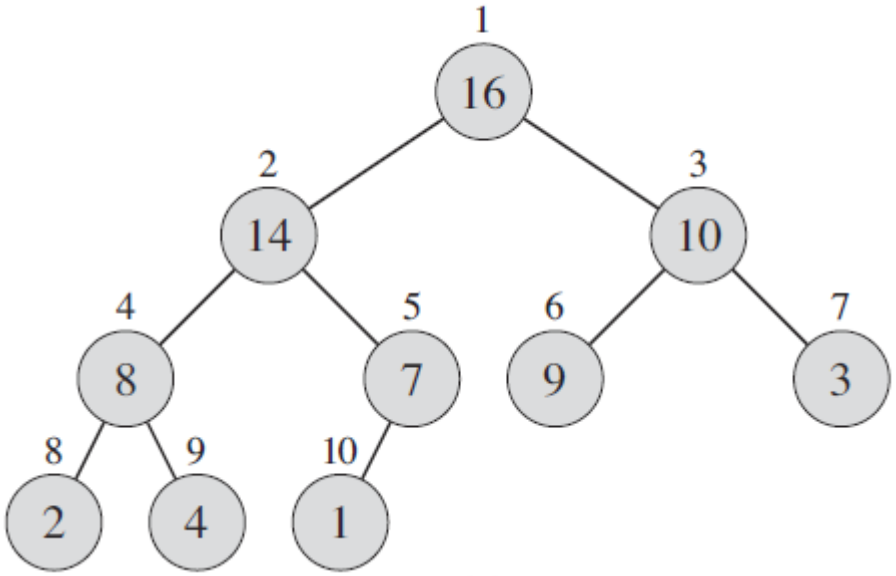
$O(n)$

max-priority queues and min-priority queues

- Application: schedule jobs on a shared computer.

- دنبال روشی هستیم که عمل حذف و درج در $O(logn)$ انجام شود.

Max-Heap



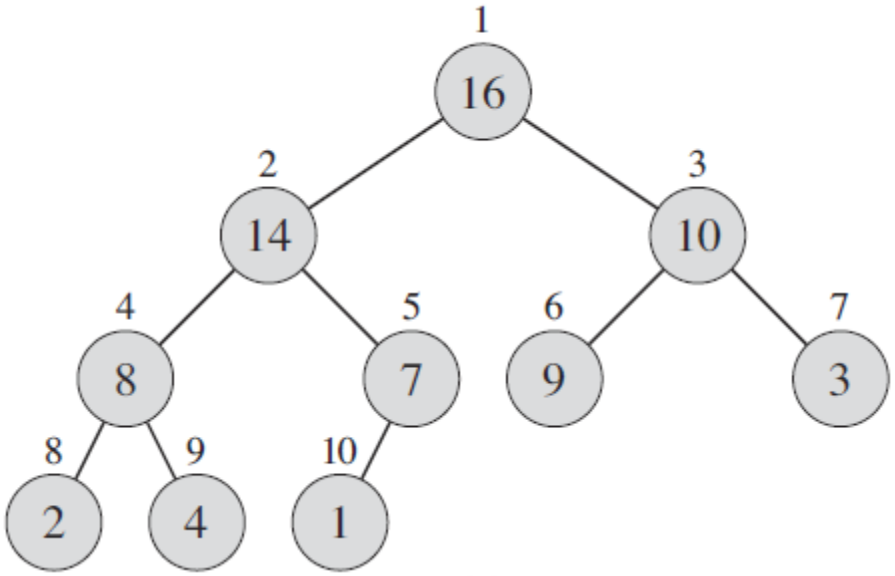
درخت متوازن

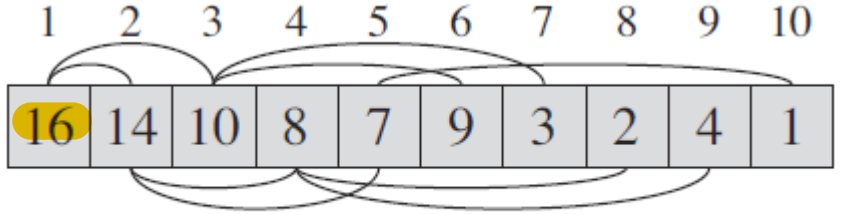
ارتفاع درخت: $logn$

عملیات درج و حذف $O(logn)$

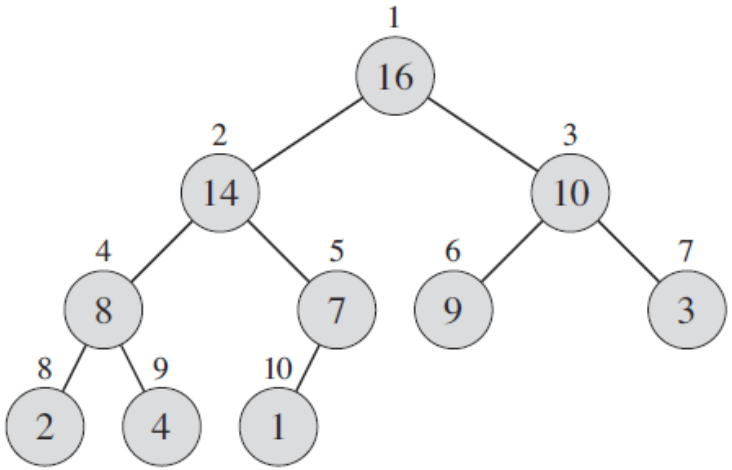
سمانه حسینی سمنانی
هیات علمی دانشکده برق و کامپیوتر- دانشگاه صنعتی اصفهان

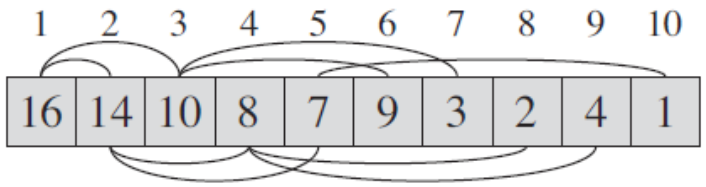
پیاده سازی Max-Heap با آرایه





پیاده سازی Max-Heap با آرایه



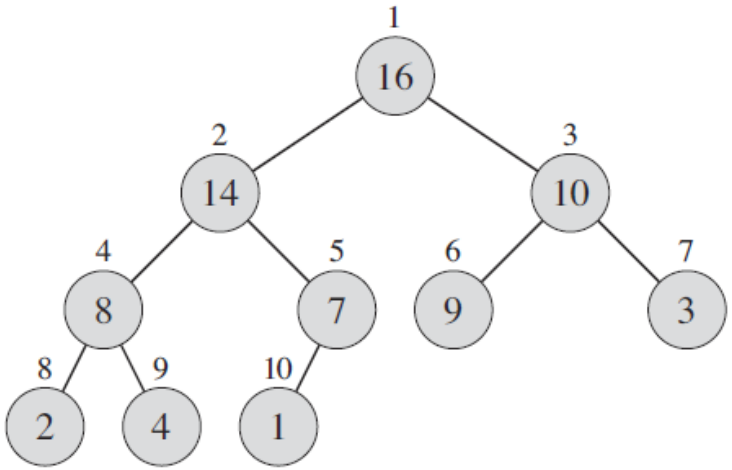



```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

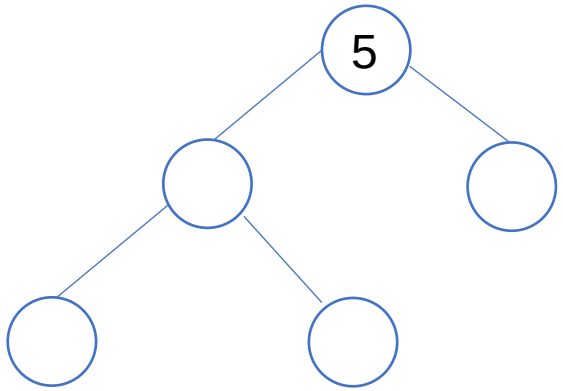
for every node i other than the root:

$A[\text{PARENT}(i)] \geq A[i]$

جستجو در Max-Heap

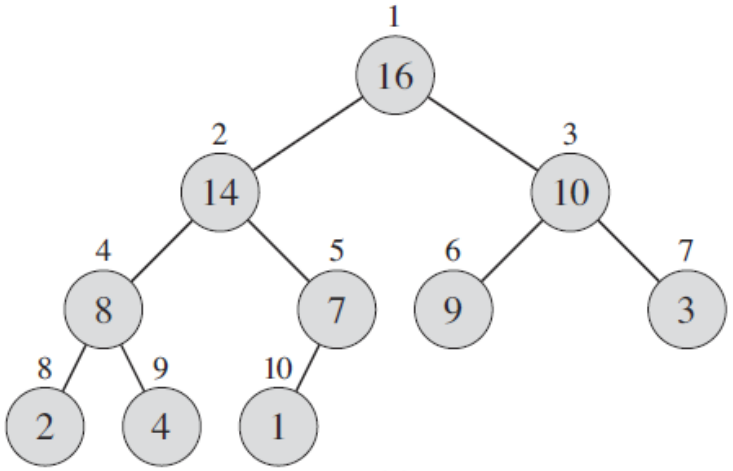


- هزینه سرچ $O(n)$
- برای search ساخته نشده و مناسب نیست
- آیا به ازای اعداد یکسان یک درخت heap واحد می توان ساخت؟
  1, 2, 3, 4, 5
- مثال:



- ۸ حالت مختلف

Operations on Max-Heap



MAX-HEAPIFY

BUILD-MAX-HEAP

HEAPSORT

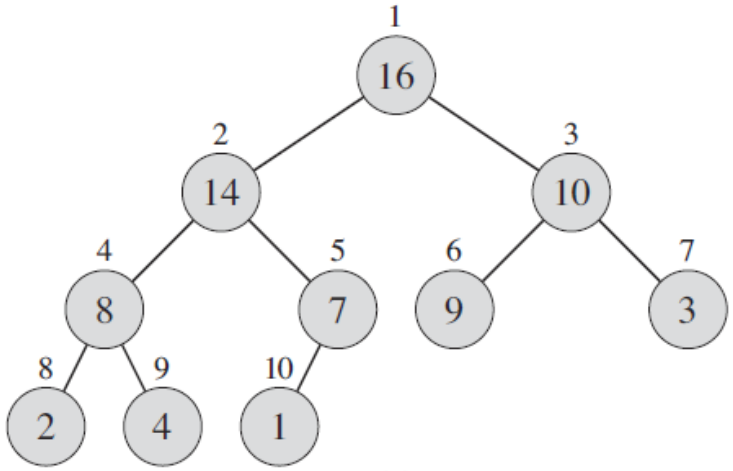
MAX-HEAP-INSERT,

HEAP-EXTRACT-MAX

HEAP-INCREASE-KEY,

HEAP-MAXIMUM

MAX-HEAPIFY



to maintain the max-heap property

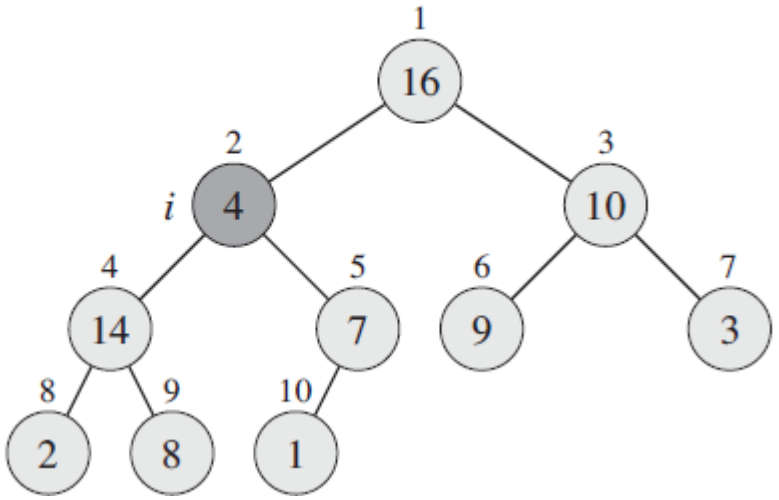
Its inputs are an array A and an index i into the array

assumes that the binary trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps,

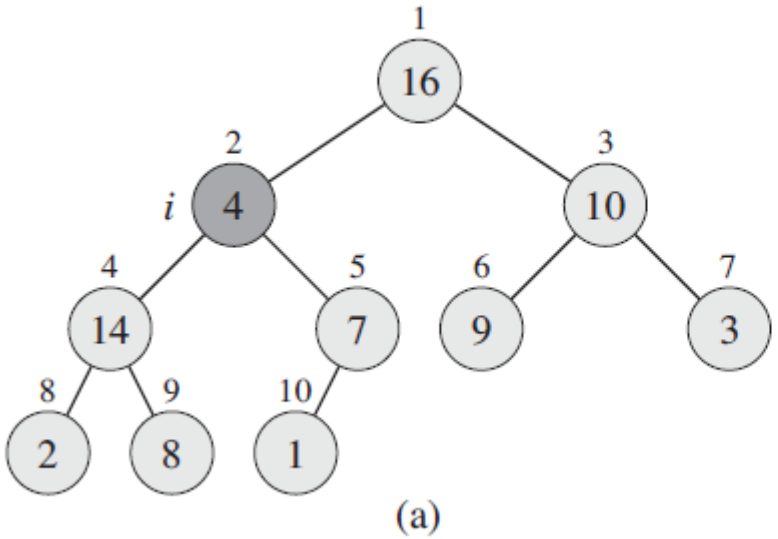
but that $A[i]$ might be smaller than its children,

lets the value at $A[i]$ "float down" in the max-heap

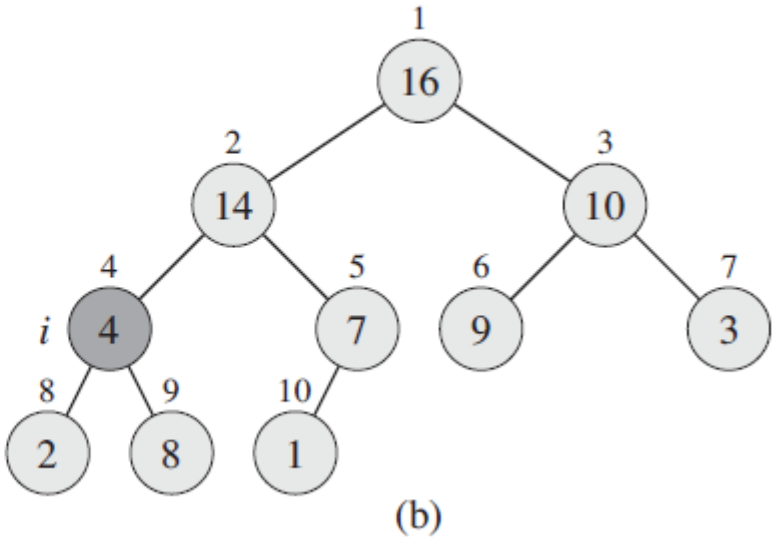
MAX-HEAPIFY




```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```



(a)



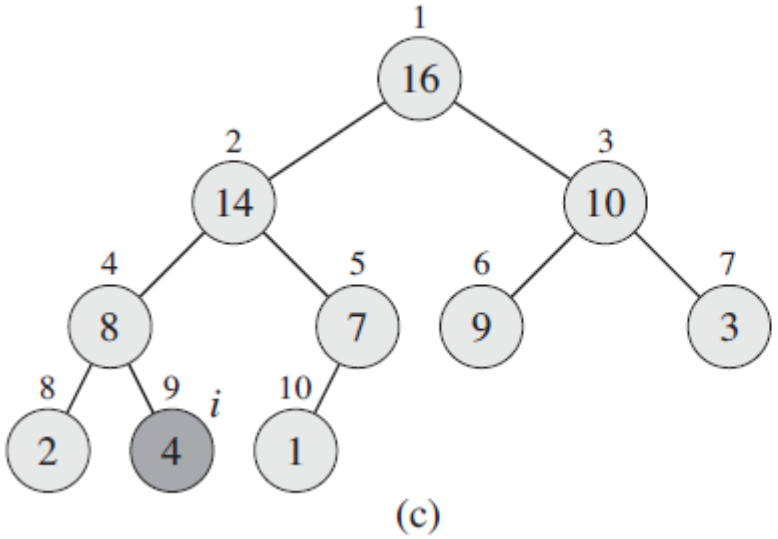
(b)

$A.heap\text{-}size = 10.$

MAX-HEAPIFY$(A, 2)$

MAX-HEAPIFY$(A, 4)$

MAX-HEAPIFY$(A, 9)$



(c)

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

BUILD-MAX-HEAP

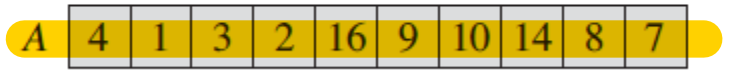
convert an array $A[1..n]$, where $n = A.length$, into a max-heap.

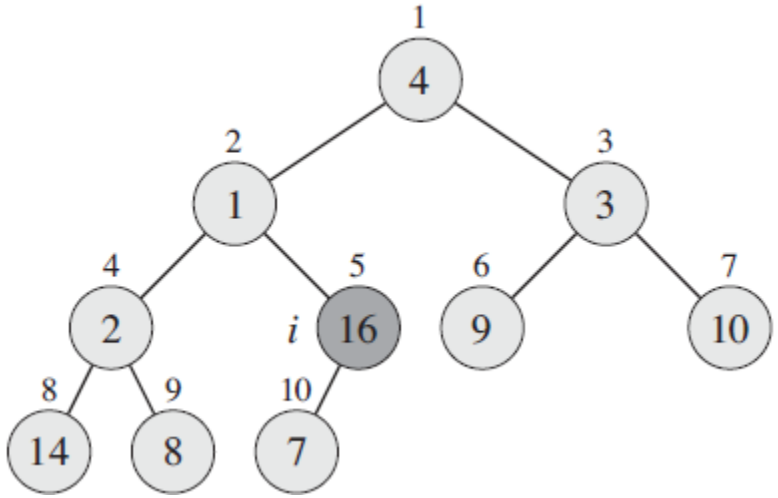
- ساده ترین راه sort(A) و بعد ساخت درخت : O(nlogn)

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



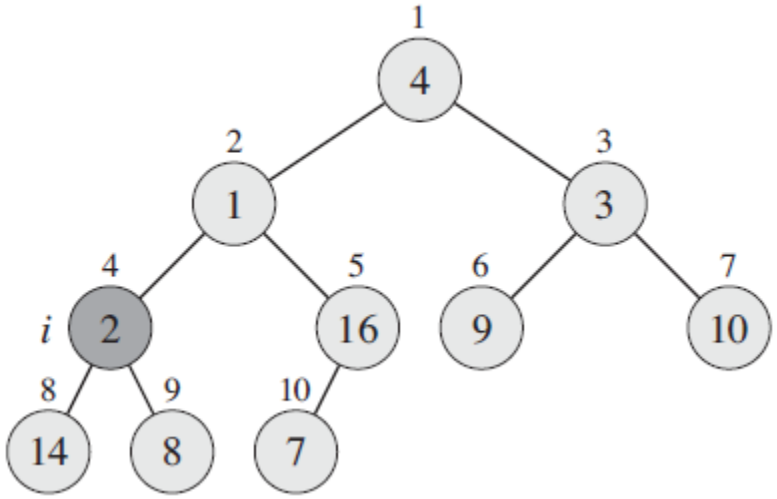


Binary tree

BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

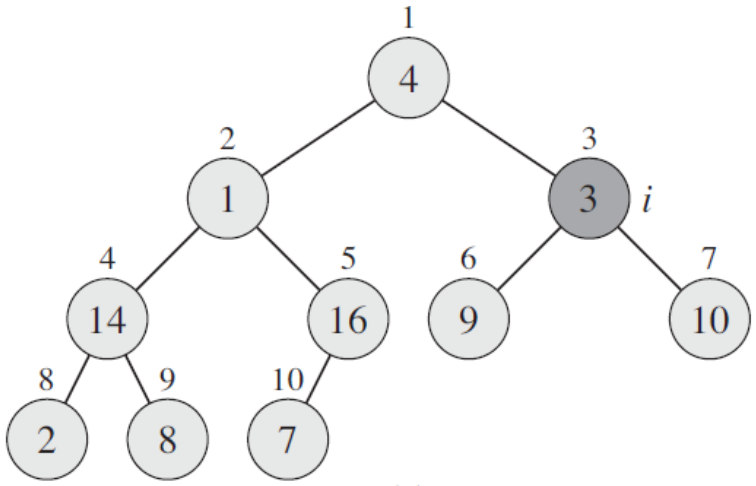
A	4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

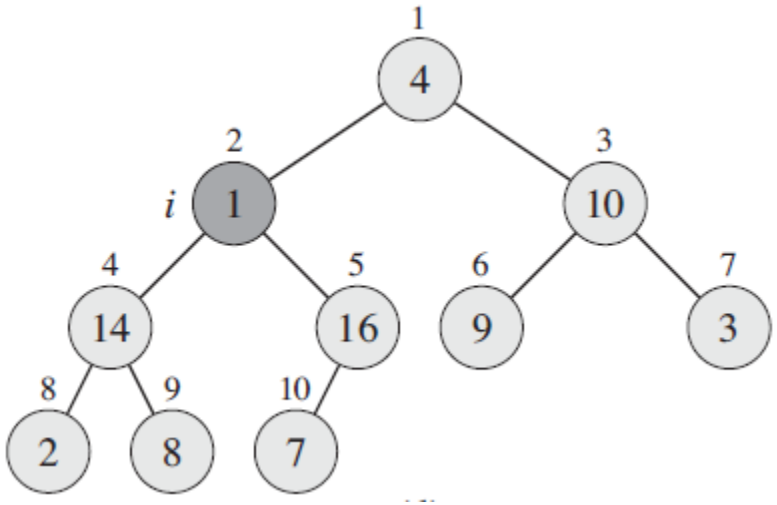
A	4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

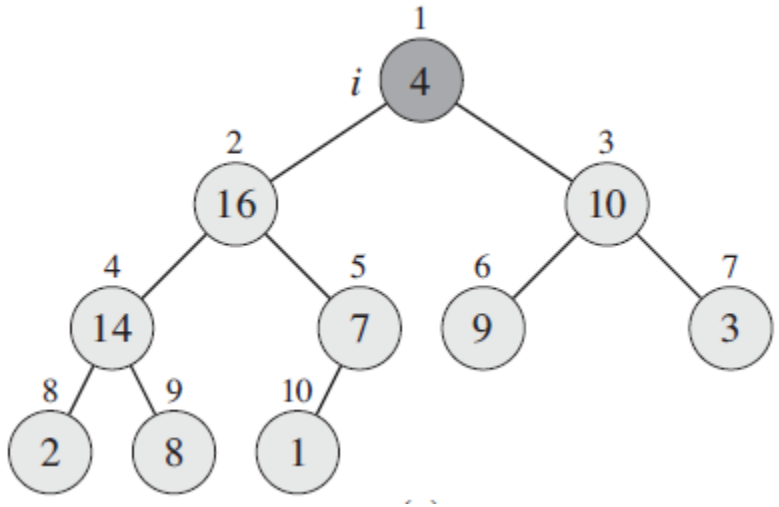
A	4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

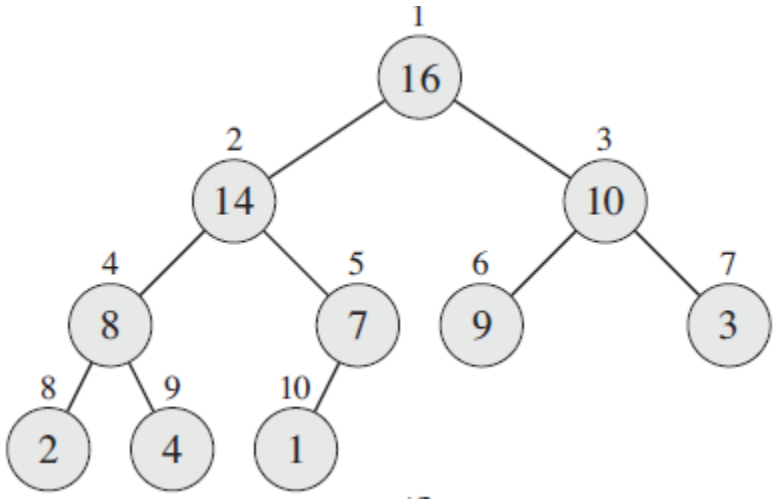
A	4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP

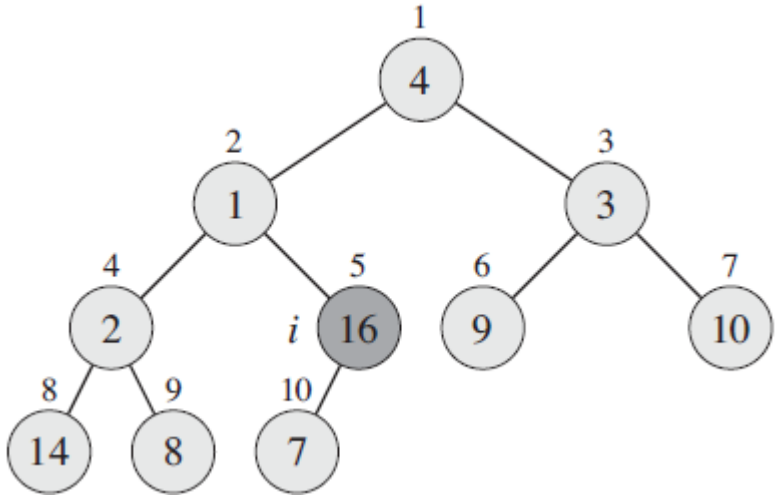
```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

A	4	1	3	2	16	9	10	14	8	7



BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



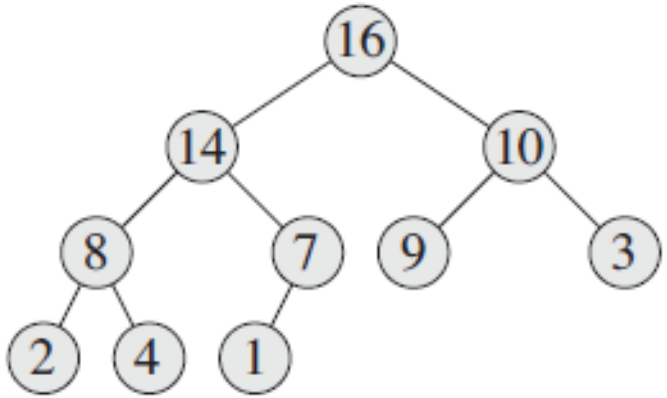
$$\sum_{h=0}^{logn} h * 2^{logn-h} = \sum_{h=0}^{logn} h * \frac{n}{2^h} = n \sum_{h=0}^{logn} \frac{h}{2^h} = O(n)$$

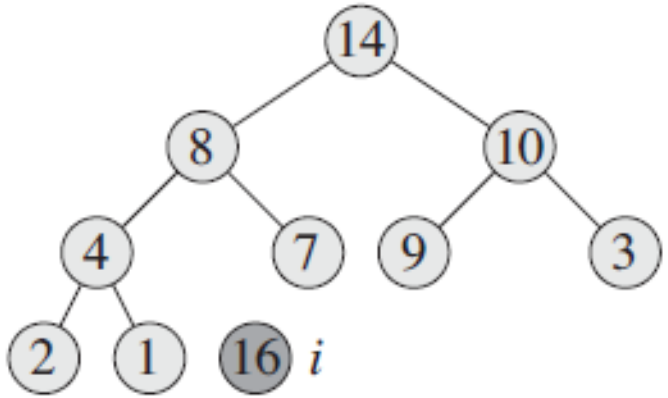
HEAPSORT

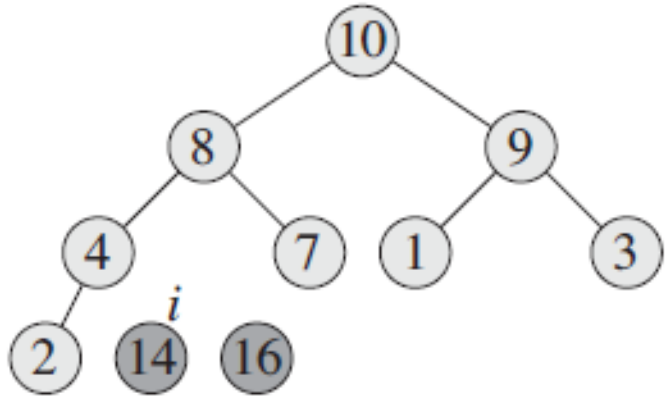
starts by using BUILD-MAX-HEAP to build a max-heap

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT

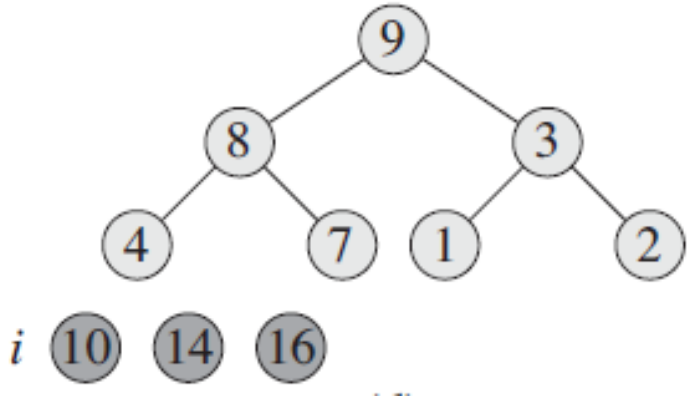


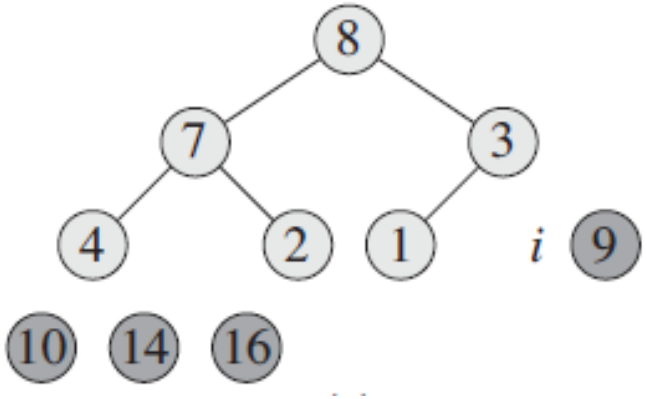


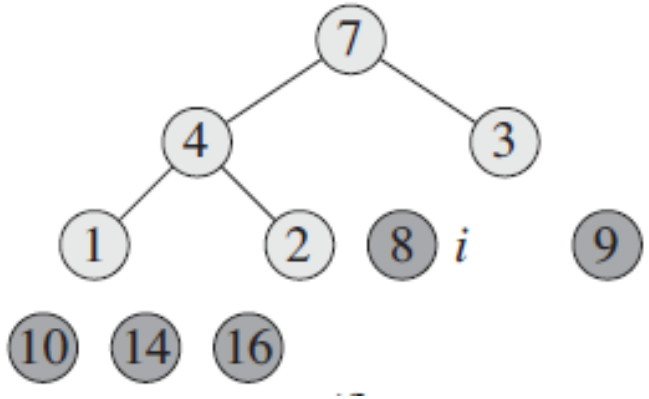



```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT

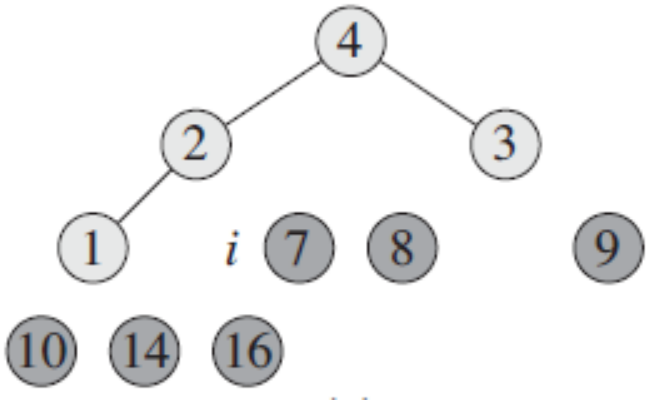


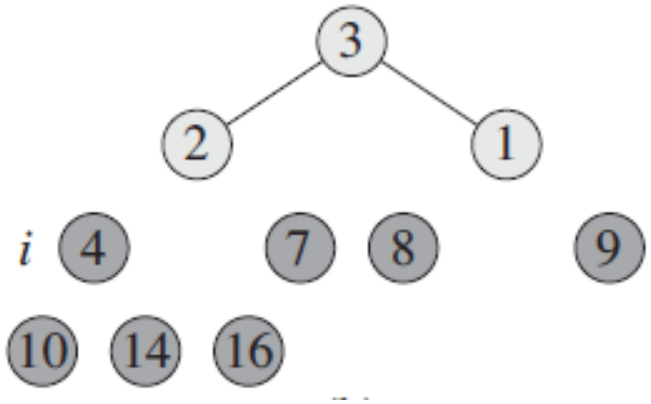


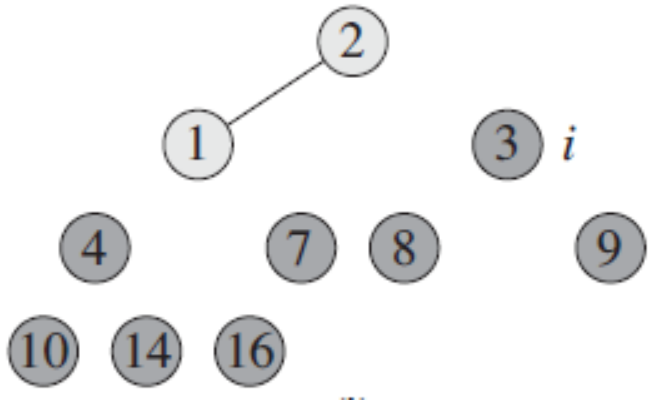



```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT

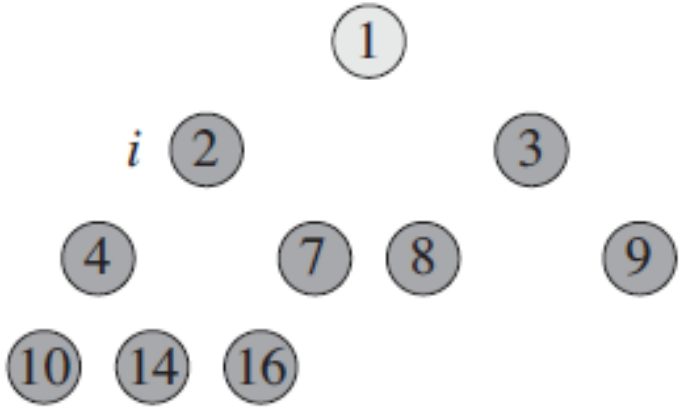







```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT



A	1	2	3	4	7	8	9	10	14	16

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n - 1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

Max-priority queue

- A ***max-priority queue*** supports the following operations:
- INSERT(S, x): inserts the element x into the set S, which is equivalent to the $S = S \cup \{x\}$
- MAXIMUM(S): returns the element of S with the largest key.
- EXTRACT-MAX(S): removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k): increases the value of element xzs key to the new value k, which is assumed to be at least as large as xzs current key value.

Max-priority queue

- max-priority queues to schedule jobs on a shared computer.
- The max-priority queue keeps track of the jobs to be performed and their relative priorities.
- When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX.
- The scheduler can add a new job to the queue at any time by calling INSERT.

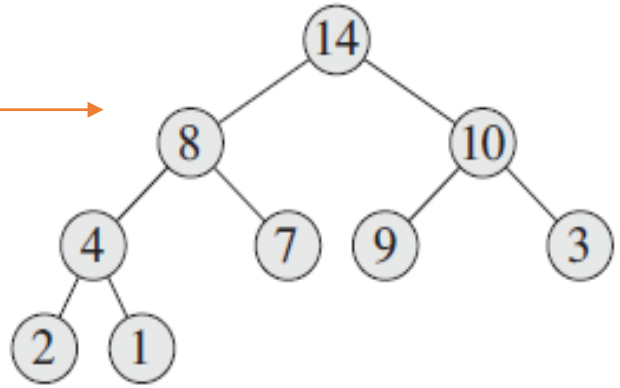
HEAP-MAXIMUM

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

$\Theta(1)$

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

$O(\lg n)$,

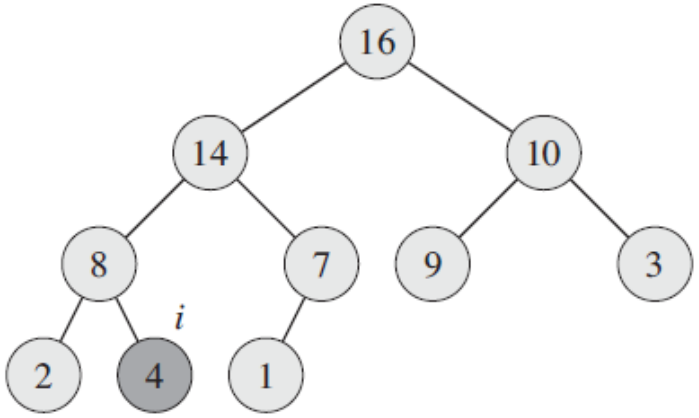


HEAP-INCREASE-KEY

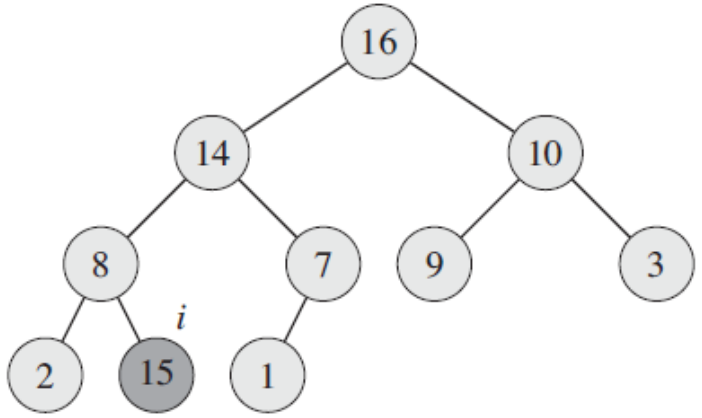
An index i into the array identifies the priority-queue element whose key we wish to increase.

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

HEAP-INCREASE-KEY(A, 4, 15)

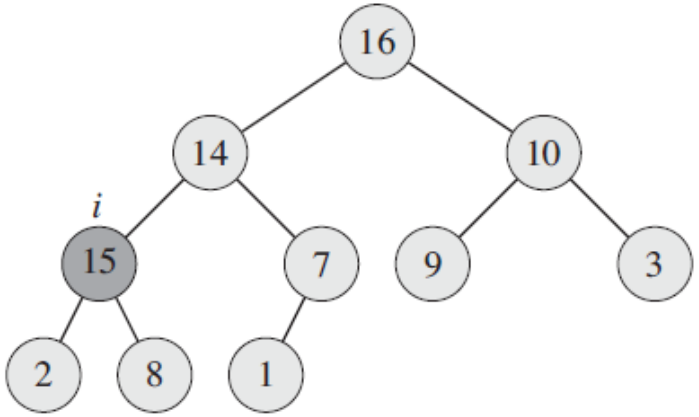


(a)

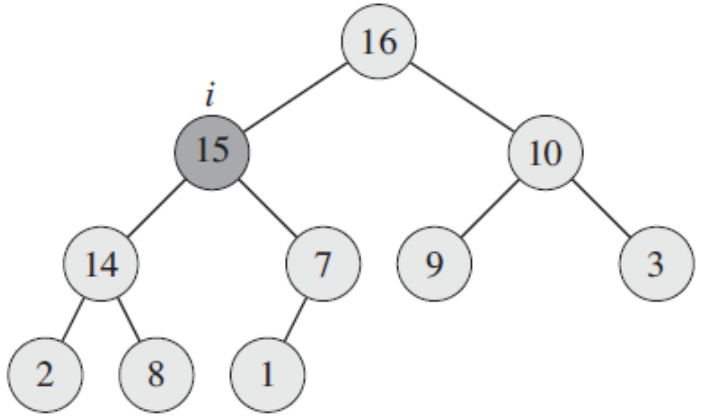


(b)

$O(\lg n)$

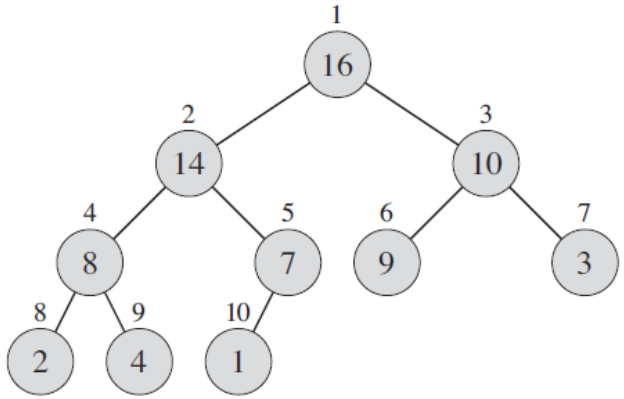


(c)



(d)

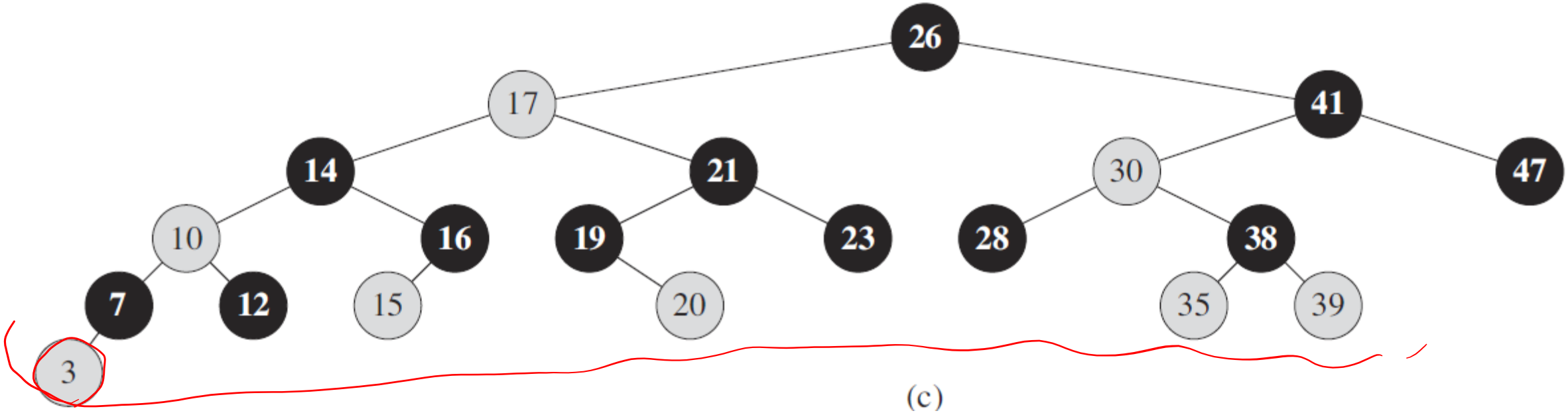
MAX-HEAP-INSERT




```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

$O(\lg n)$

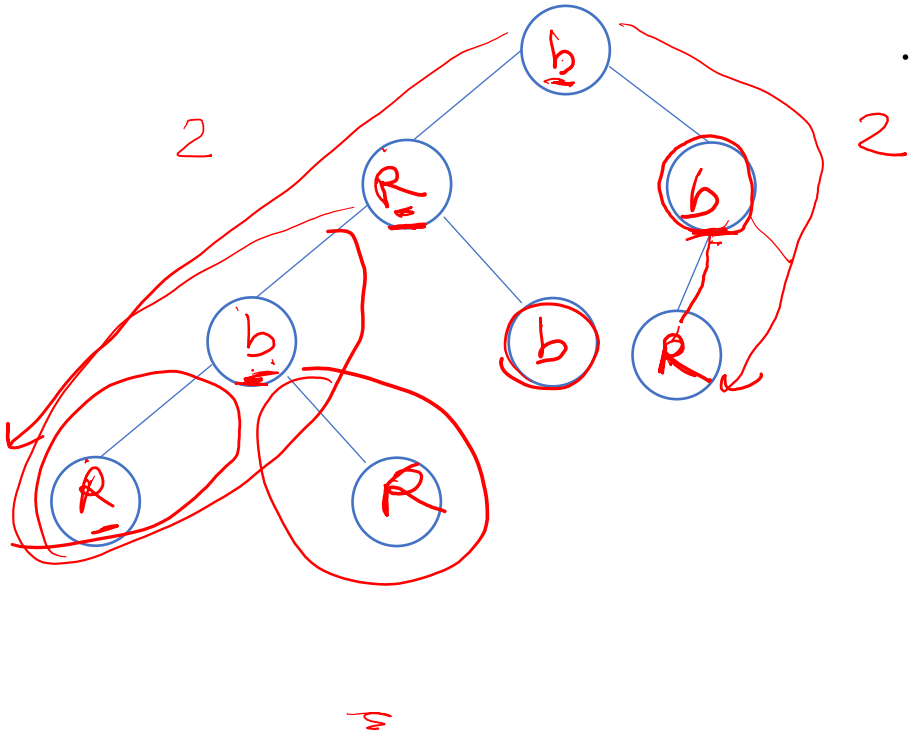
Red-black tree

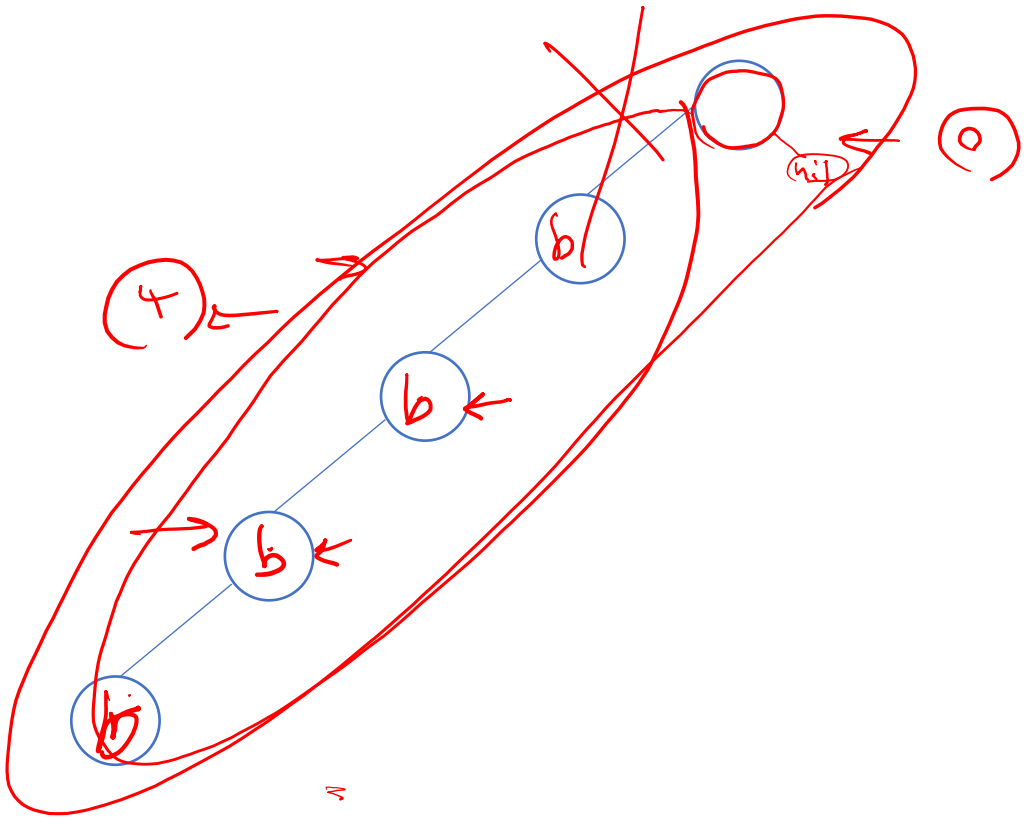


(c)

Red-black tree

- درخت های روبرو را طوری رنگ آمیزی کنید که red-black باشد.





Red-black tree

- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately **balanced**.

Lemma 13.1

A red-black tree with n internal nodes has height at most $2 \lg(n + 1)$.

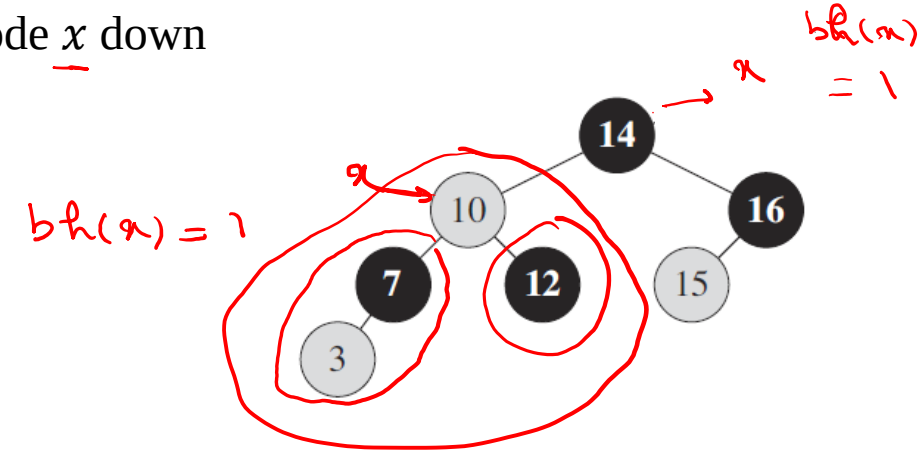
Red-black tree

Lemma 13.1

A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

- $bh(x)$: تعداد نودهای سیاه از نود x تا برگ به غیر از خود نود

the number of black nodes on any simple path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted bh(x)



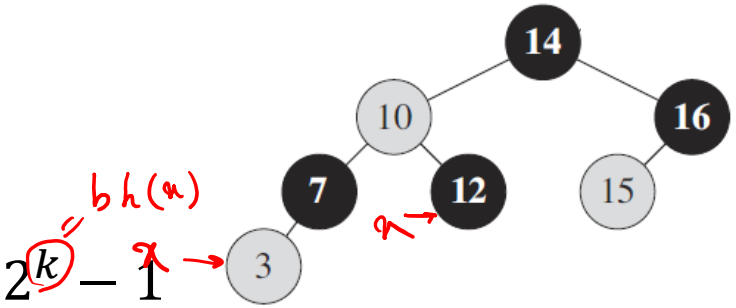
subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

- اثبات استقرایی روی ارتفاع x

Red-black tree

subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

- if the height of x is 0, then x must be a leaf (T.nil),
- the subtree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes

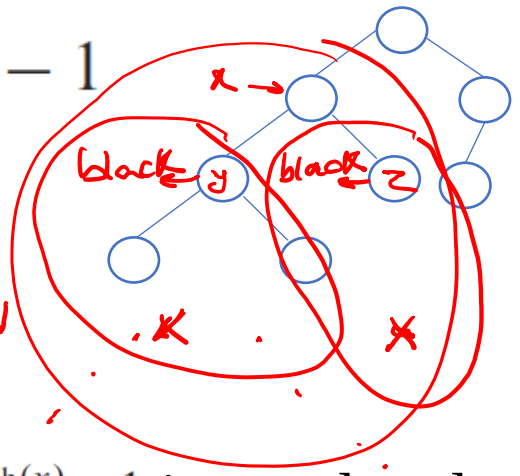


- Suppose: $bh(x) = k \rightarrow number\ of\ nodes\ in\ the\ subtree \geq 2^k - 1$
- prove: $bh(x) = k + 1 \rightarrow number\ of\ nodes\ in\ the\ subtree \geq 2^{k+1} - 1$

Red-black tree

subtree rooted at any node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes.

- Suppose: $bh(x) = k \rightarrow number\ of\ nodes\ in\ the\ subtree \geq 2^k - 1$
- prove: $bh(x) = k + 1 \rightarrow number\ of\ nodes\ in\ the\ subtree \geq 2^{k+1} - 1$
- Each child y and z has a black-height of either $\mathrm{bh}(x)$ or $\mathrm{bh}(x) - 1$
- depending on whether its color is red or black, respectively.
- y and z has at least $2^{\mathrm{bh}(x)-1} - 1$ internal nodes.
- The subtree rooted at x has at least $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes.



Red-black tree

Lemma 13.1

A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

- $bh(root) \geq \frac{h}{2}$

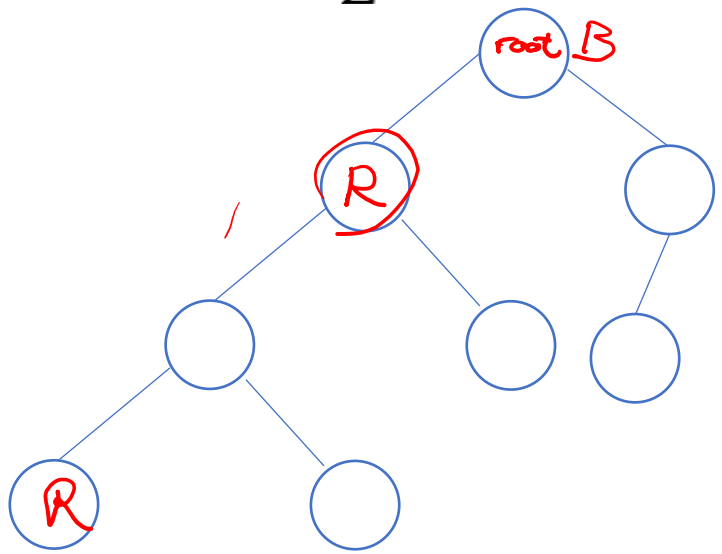
subtree rooted at any node x contains at least

- $bh(root) \geq \frac{h}{2}$

- تعداد گره ها $= n \geq 2^{bh(root)} - 1 \geq 2^{\frac{h}{2}} - 1$

- $n+1 \geq 2^{\frac{h}{2}} \quad \rightarrow h \leq 2\log(n+1)$

$\log(n+1) \geq \frac{h}{2}\log_2^2 \rightarrow h \leq 2\log(n+1)$

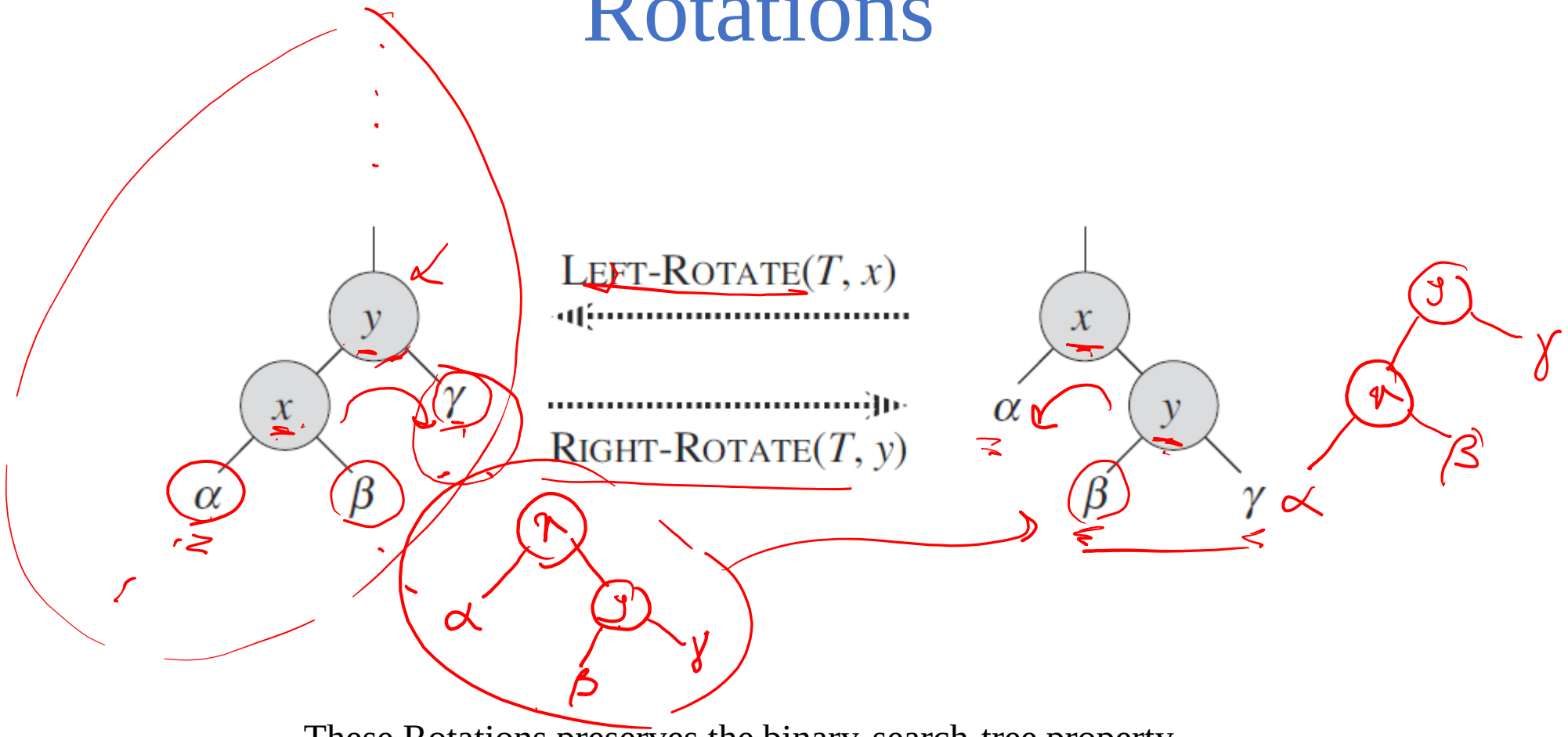


Red-black tree

A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

- SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR are O(log n)
  - دنبال حفظ توازن درخت هستیم ← باید خاصیت قرمز- سیاه را در درج و حذف حفظ کنیم.
- INSERT and DELETE

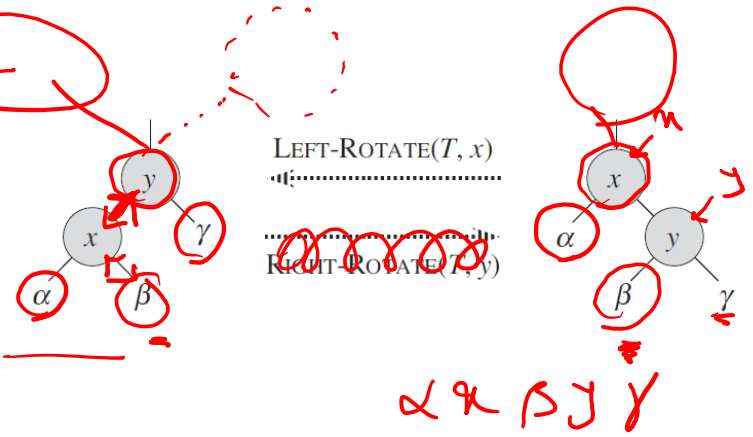
Rotations



These Rotations preserves the binary-search-tree property.

Rotations

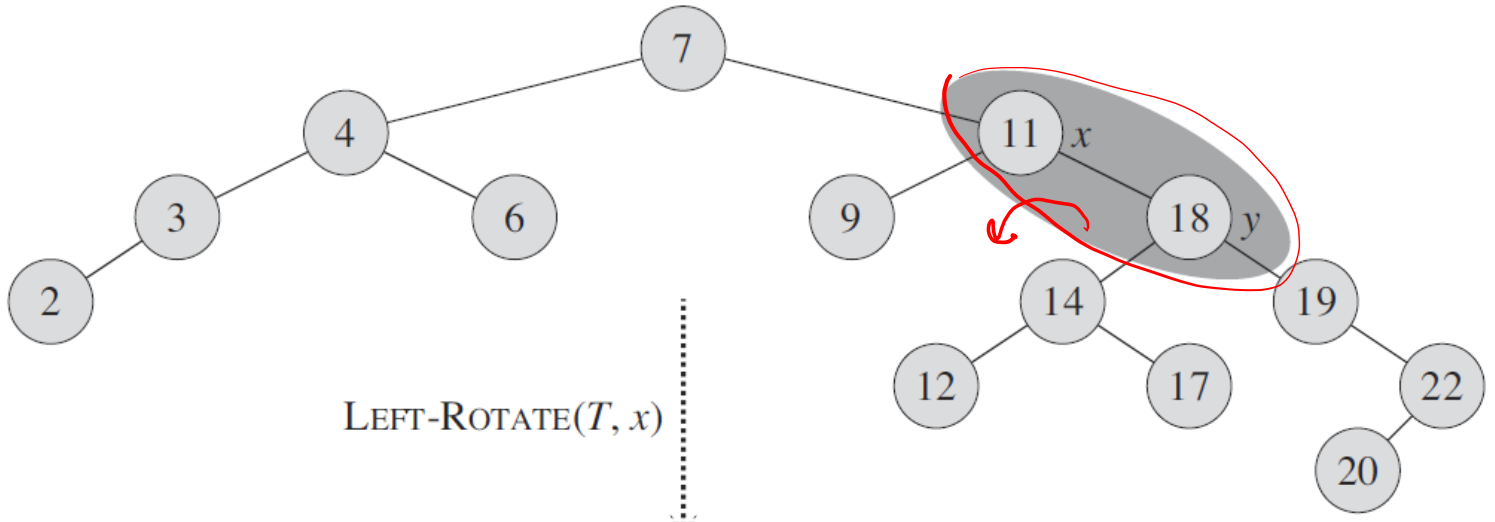
```
LEFT-ROTATE(T, x)
 1  y = x.right                  // set y
 2  x.right = y.left             // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                    // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x                   // put x on y's left
12  x.p = y
```



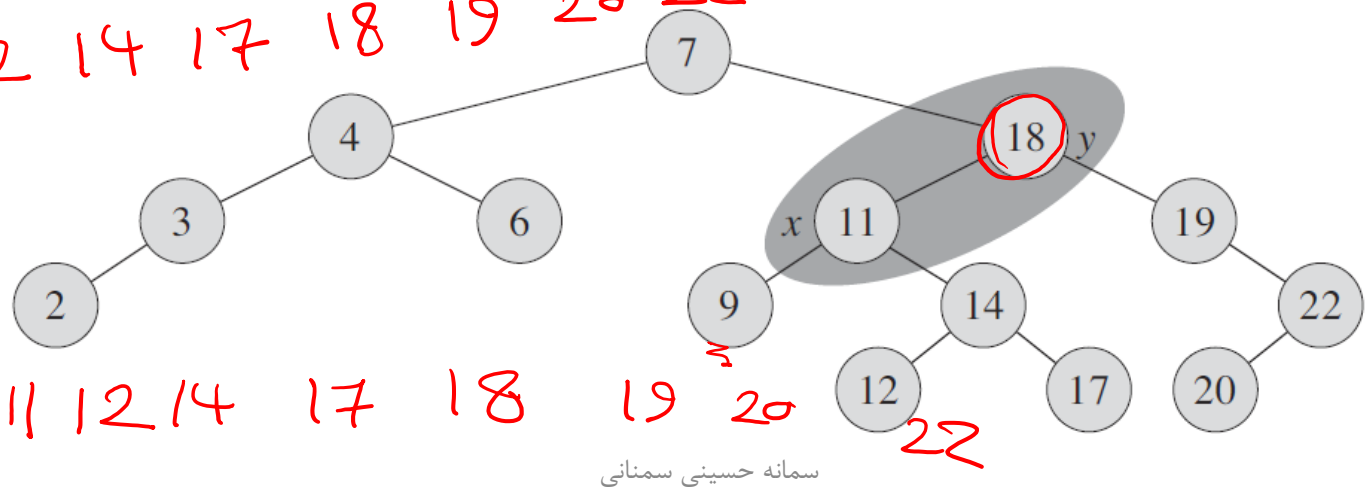
$O(1)$

Rotations

In order tree walks of the input tree and the modified tree produce the same listing of key values.



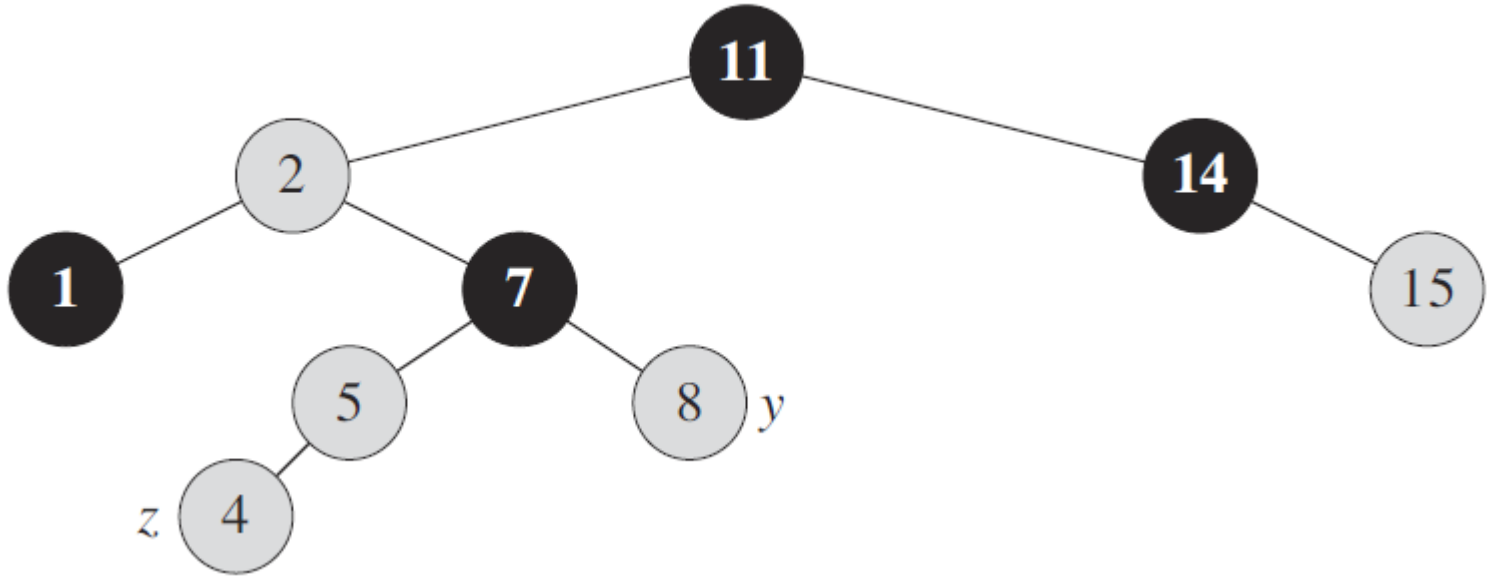
LEFT-ROTATE(T, x)



2 3 4 6 7 9 11 12 14 17 18 19 20 22

2 3 4 6 7 9 11 12 14 17 18 19 20 22

Insertion in Red-black tree



Insertion of node Z in binary search tree and then color Z red. Why red?

To guarantee that the red-black properties are preserved,
we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations.

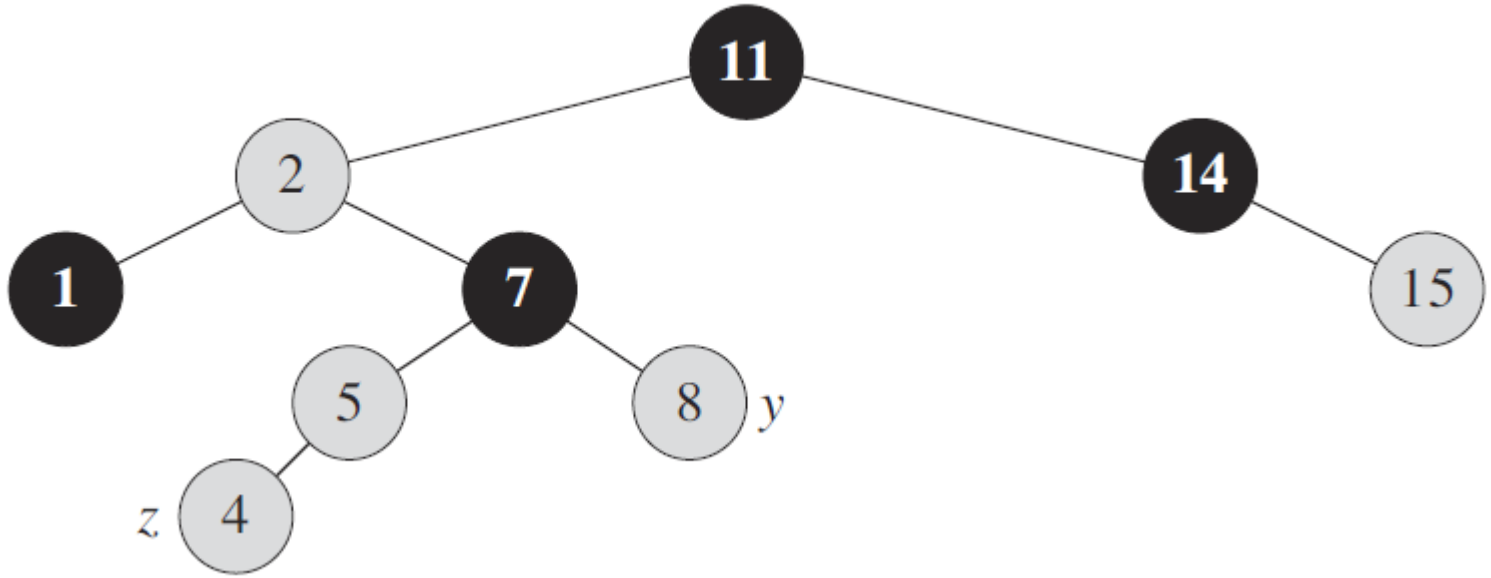
Insertion in Red-black tree

```
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)
```

almost the same as Insert in binary tree

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

Insertion in Red-black tree



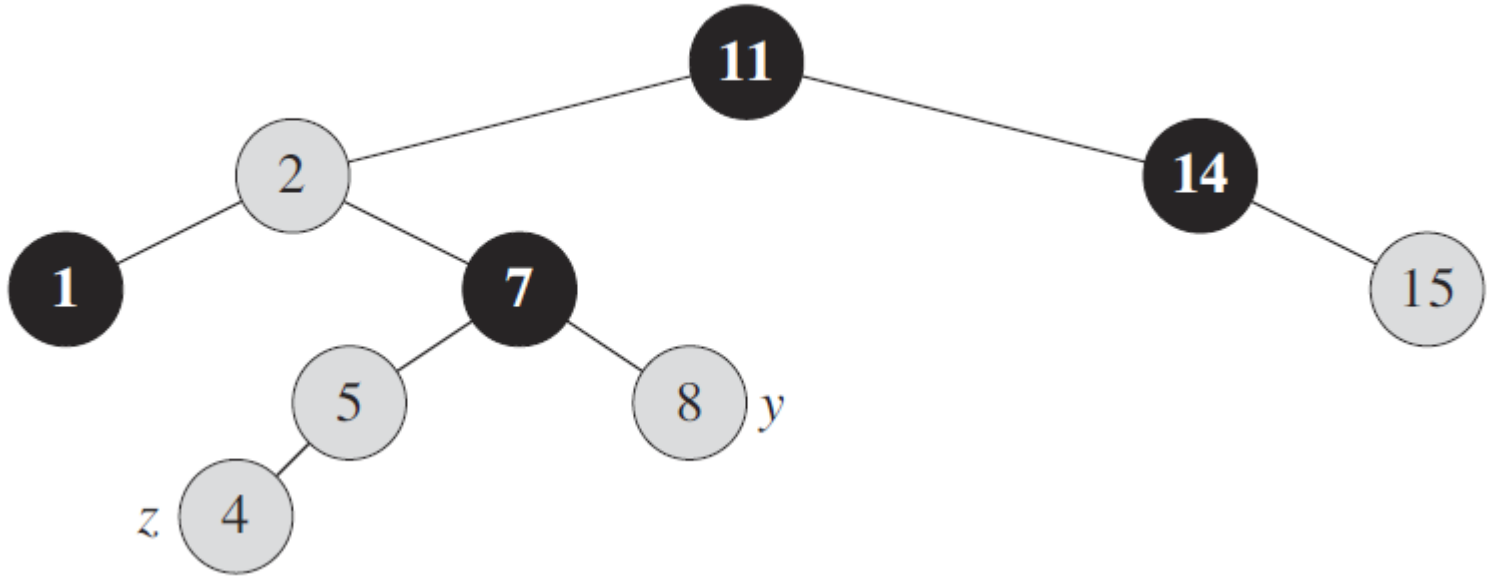
Because both z and its parent z.*p* are red, a violation of property 4 occurs

Red-black tree

- A red-black tree is a binary tree that satisfies the following 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. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

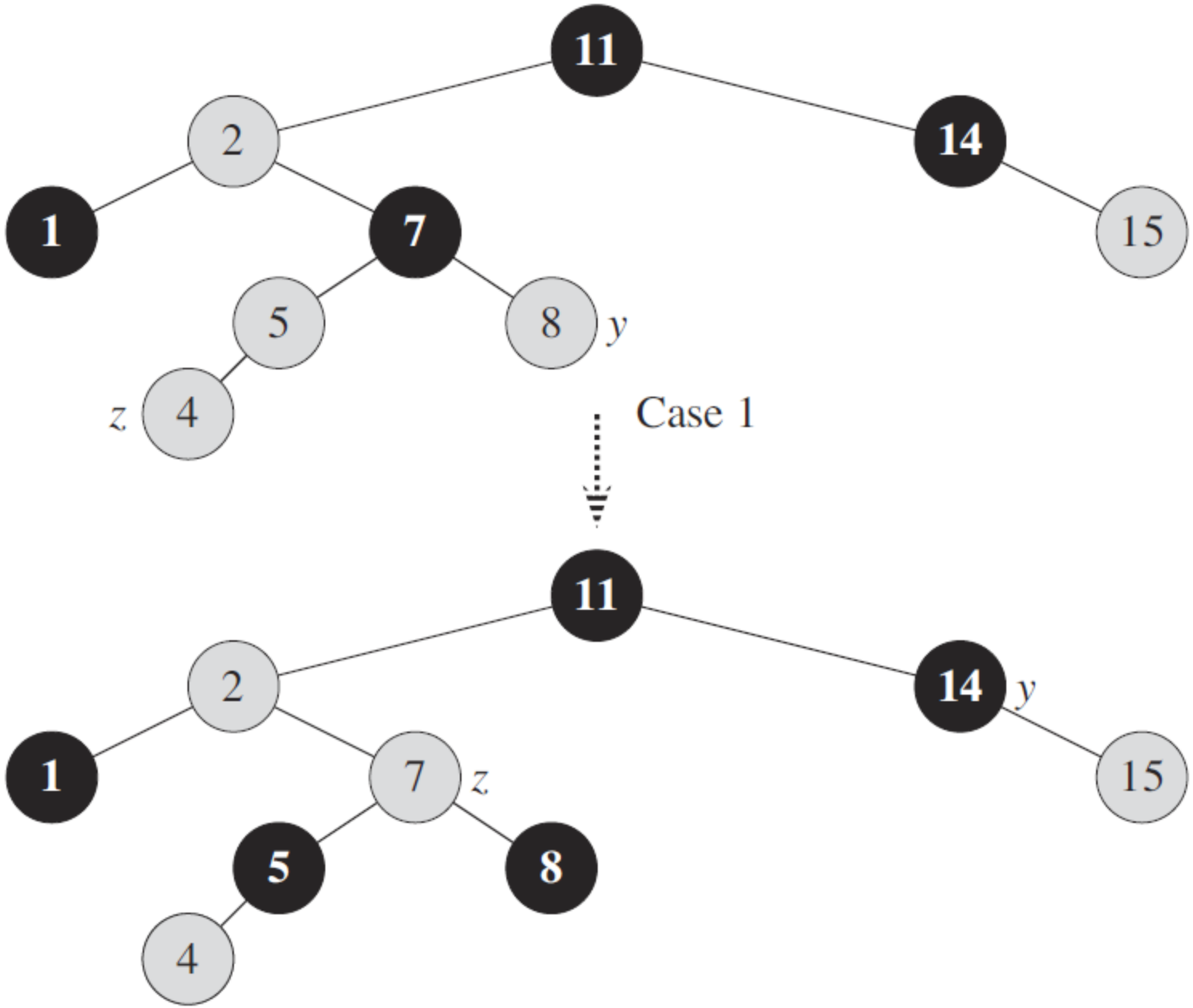
Insertion in Red-black tree



Because both z and its parent z.*p* are red, a violation of property 4 occurs

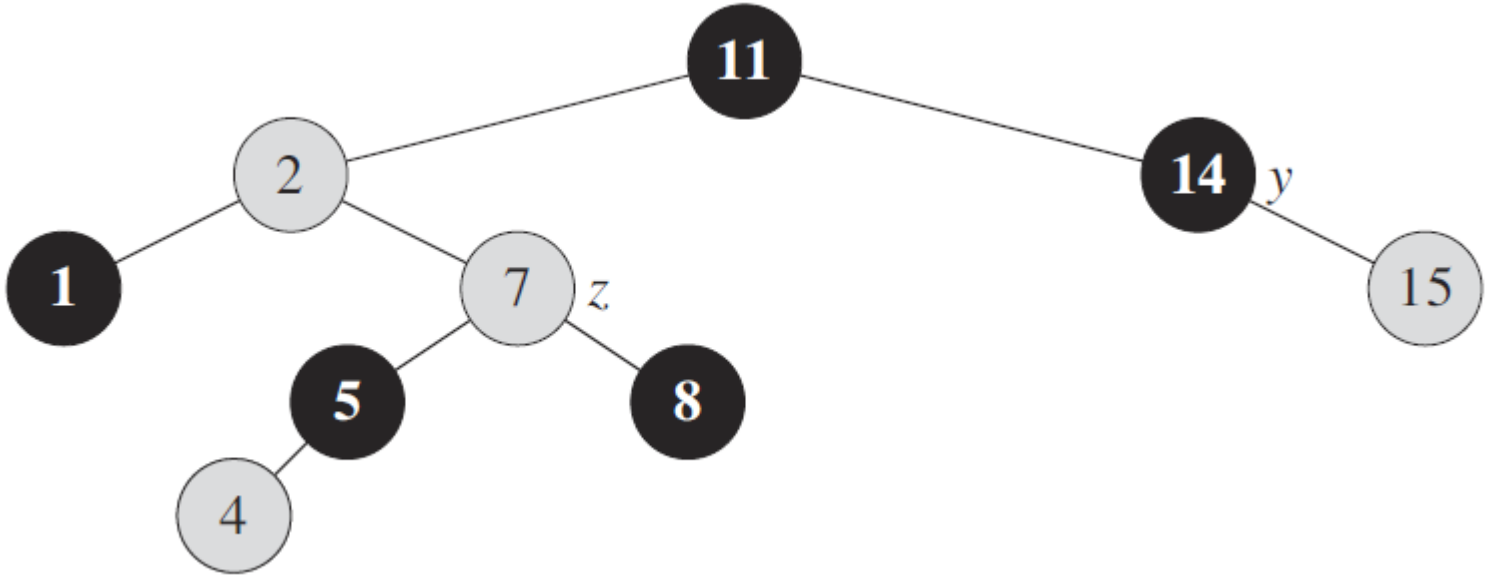
Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**.

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


11
2
14
1
7
15
5
8
y
z
4
Case 1
11
2
14
y
1
7
z
15
5
8
4

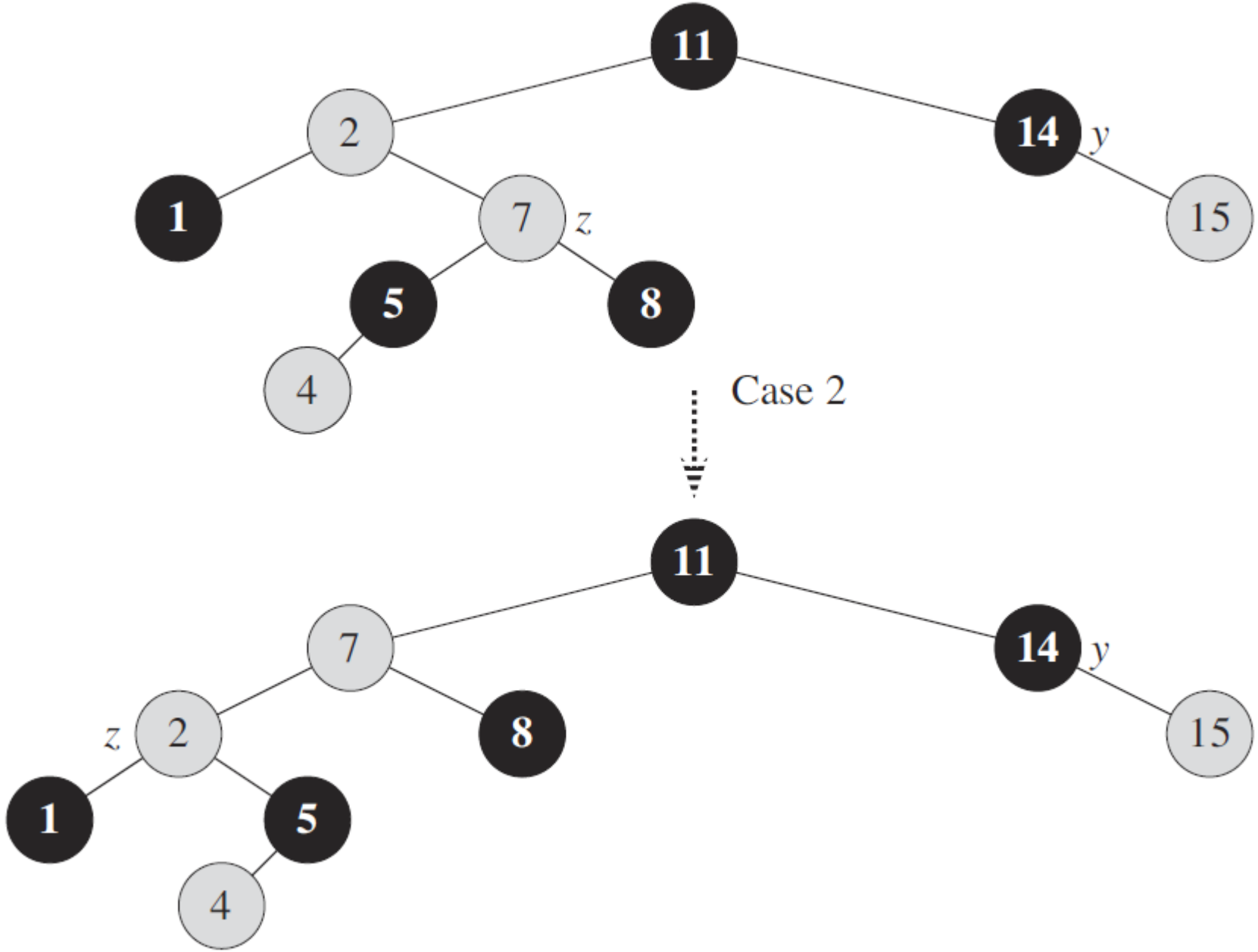
```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED 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

Insertion in Red-black tree

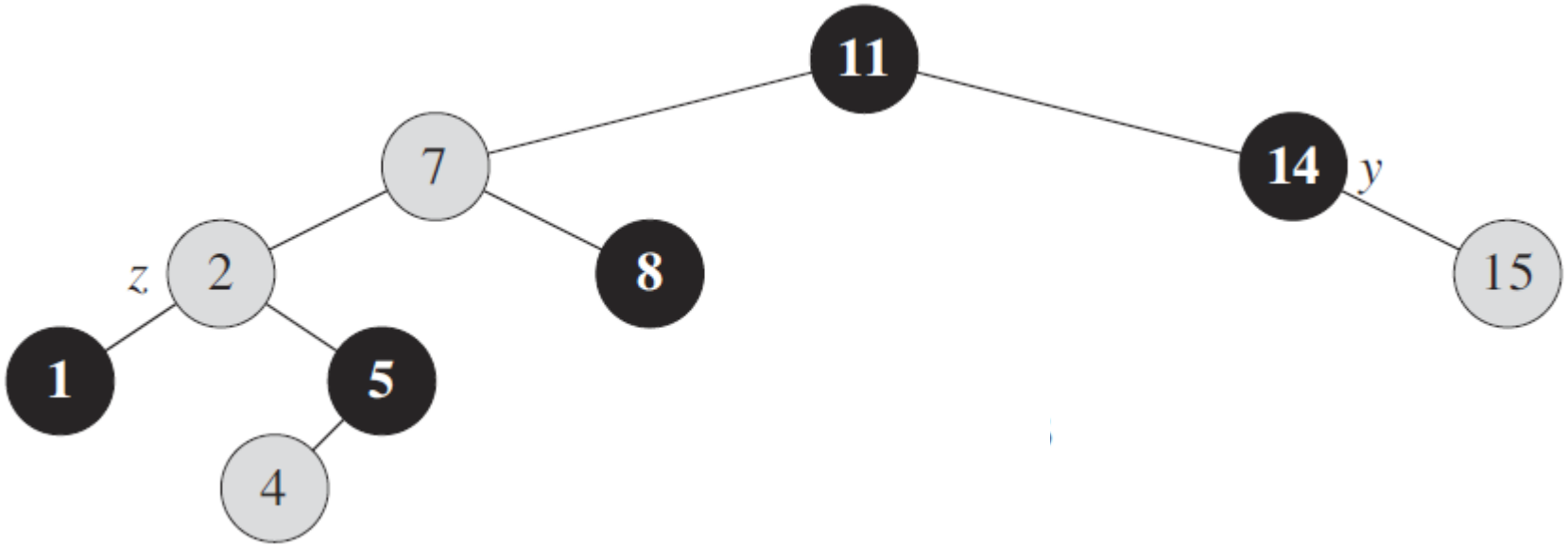


z and its parent are both red, but z's uncle y is black.
Since z is the right child of z.*p*, case 2 applies.
We perform a left rotation, and the tree that results is shown in **(c)**.

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

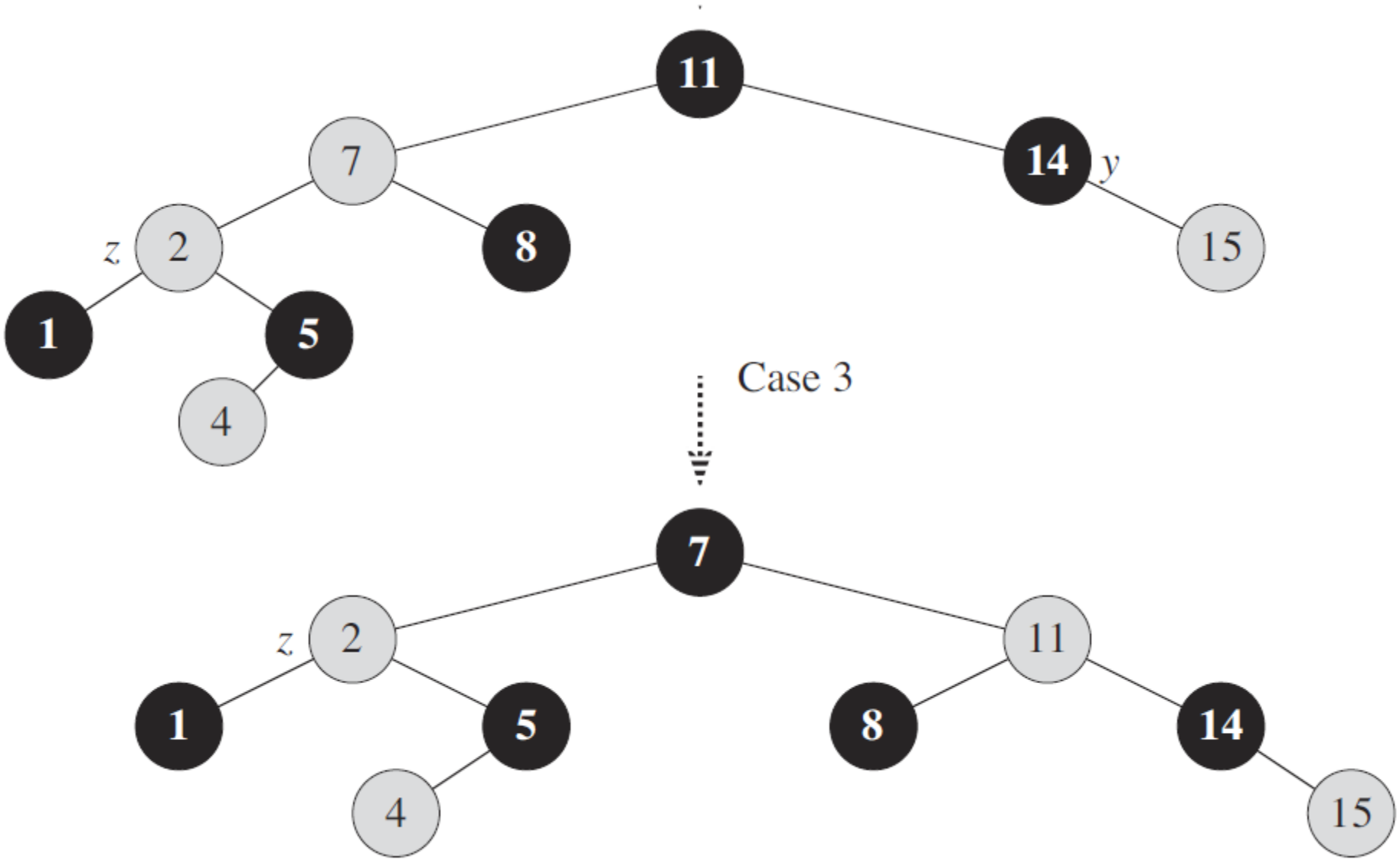

11
2
14
y
1
7
z
15
5
8
4
Case 2
11
7
14
y
z
2
8
15
1
5
4

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



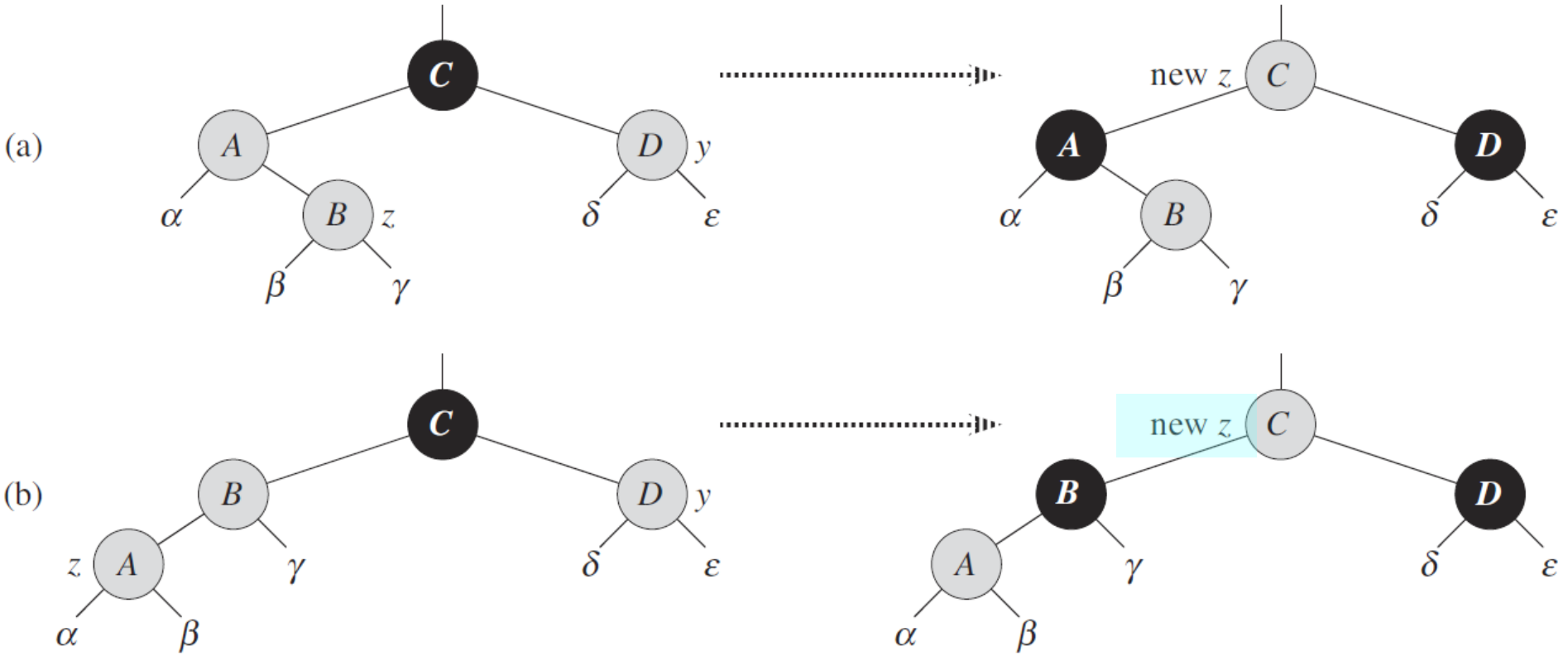
Now, z is the left child of its parent, and case 3 applies.
Recoloring and right rotation yield the tree in **(d)**,
which is a legal red-black tree.

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


11
7
14
y
z
2
8
15
1
5
4
Case 3
7
z
2
11
1
5
8
14
4
15

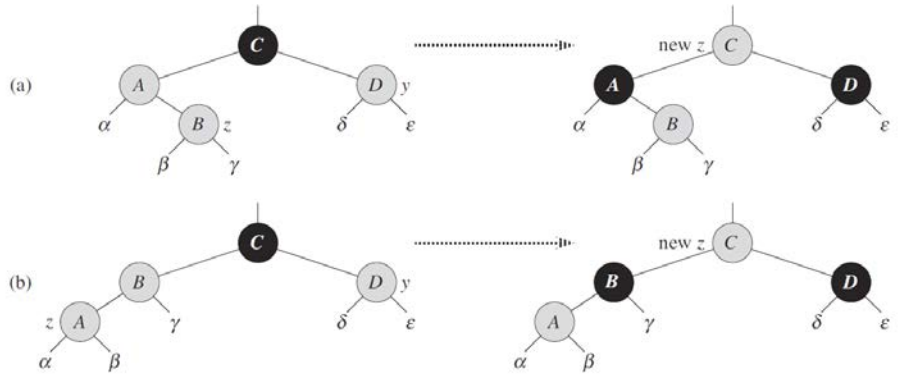
```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED 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

Case 1: z's uncle y is red



Case 1: z's uncle y is red

$$z' = z.p.p$$



a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

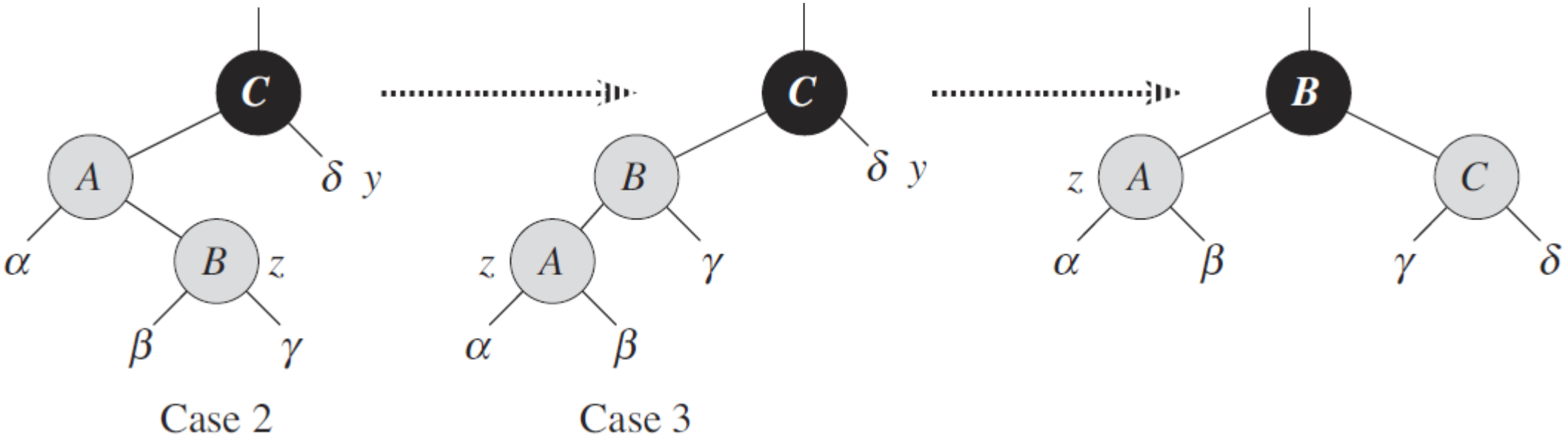
Red-black tree

- A red-black tree is a binary tree that satisfies the following red-black properties:

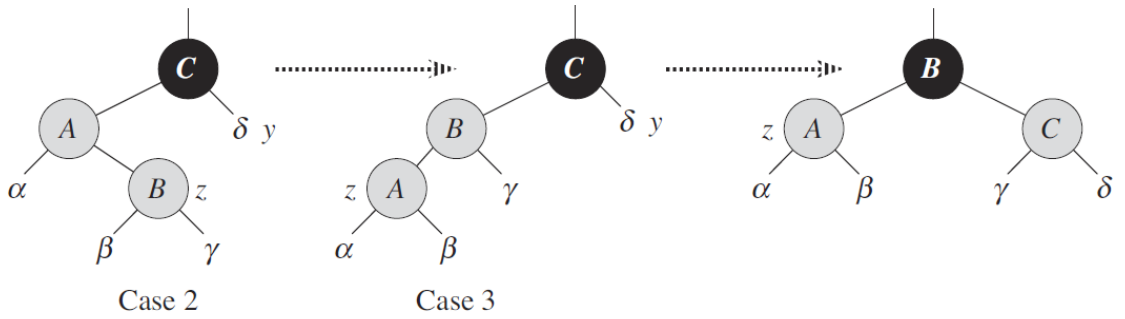
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Case 2 and Case 3

- Case 2: z's uncle y is black and z is a right child
- Case 3: z's uncle y is black and z is a left child.



Case 1: z's uncle y is red



a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

What is the running time of RB-INSERT?

```
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)
```

$O(\lg n)$

```
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
```

the **while** loop repeats only if case 1 occurs,
and then the pointer **z** moves two levels up the tree.

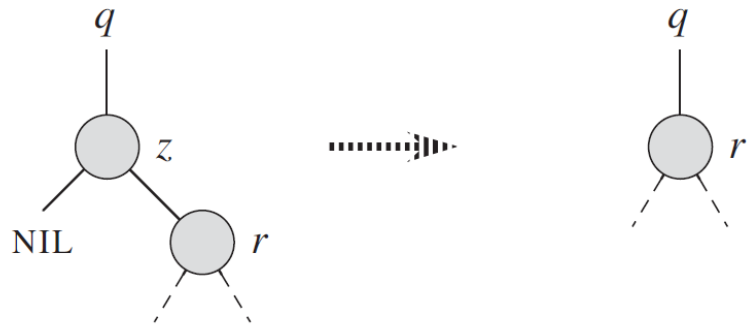
$O(\lg n)$

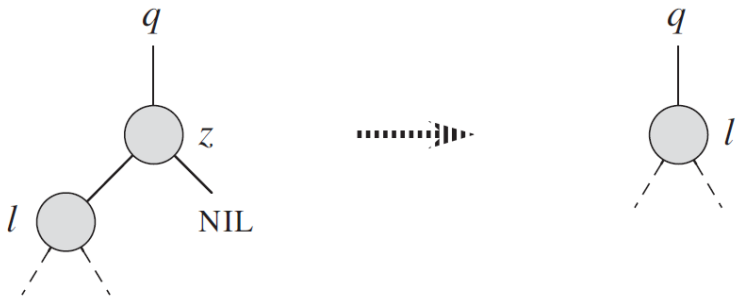
Deletion in Binary Tree

```
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
```

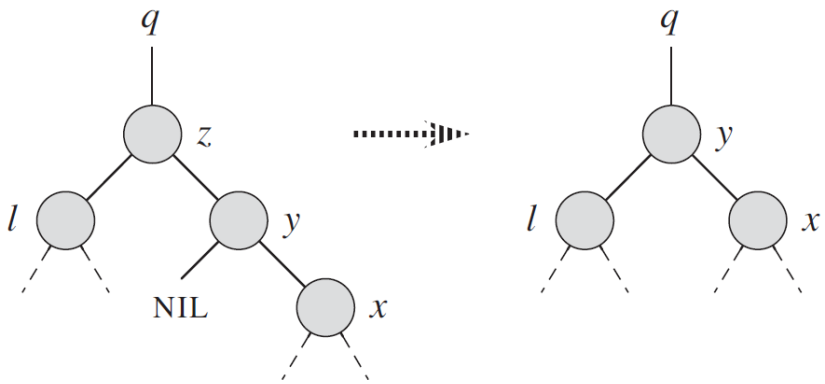
```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T,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

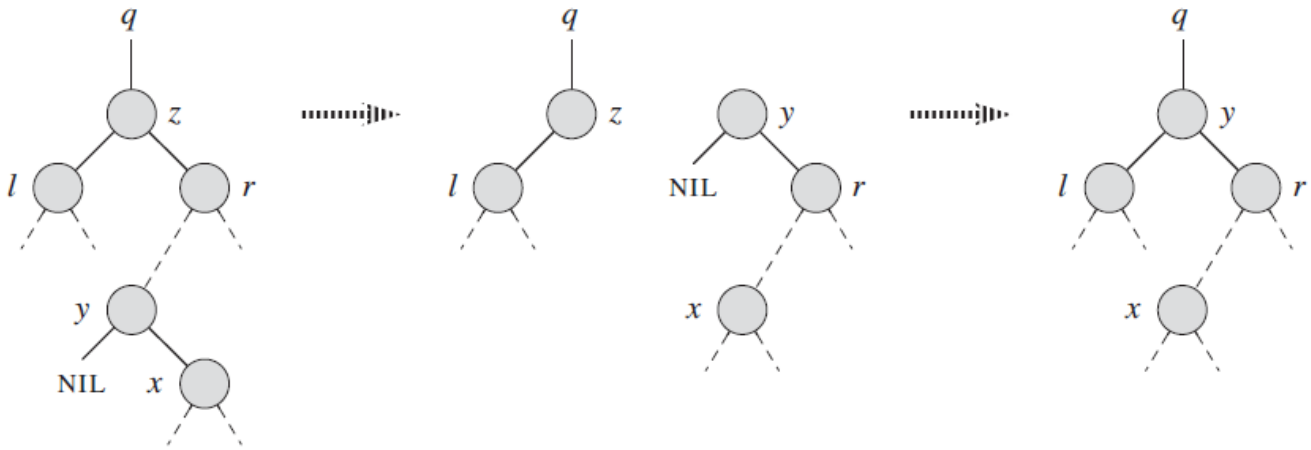
Deletion in Binary Tree





Deletion in Binary Tree





Deletion in Red-Black Tree

```
RB-TRANSPLANT(T, u, v)
1   if u.p == T.nil
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   v.p = u.p
```

```
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)
```

Deletion in Red-Black Tree

```
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
```

```
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)
```

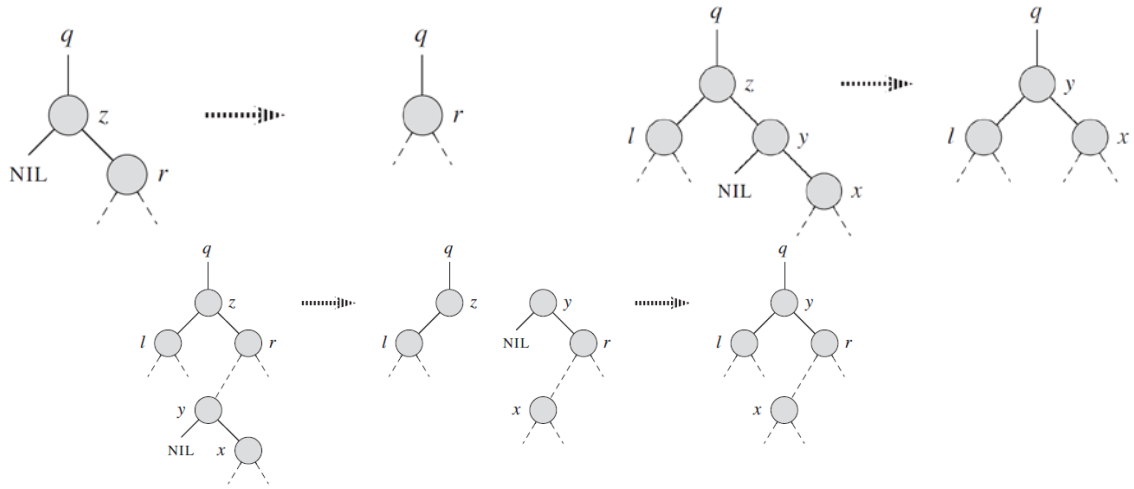
Deletion in Red-Black Tree

```
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)
```

We maintain node y as the node either removed from the tree or moved within the tree.

Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur

we keep track of the node x that moves into node y's original position.

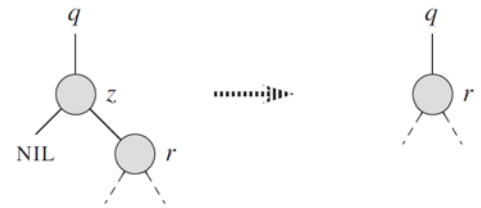


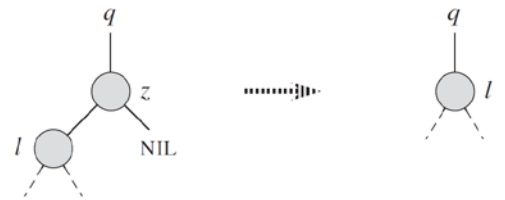
node y moves into node z's original position in the red-black tree

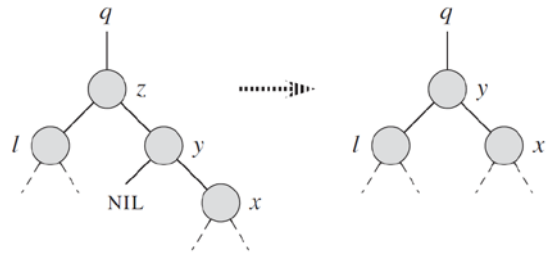
if node y was black, we might have introduced one or more violations of the red-black properties, and so we call RB-DELETE-FIXUP to restore the red-black properties.

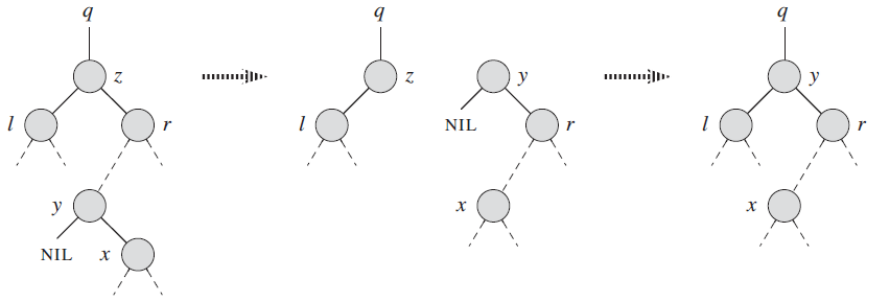
Deletion in Red-Black Tree

```
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)
```









$O(\lg n)$

Deletion in Red-Black Tree

- If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

1. No black-heights in the tree have changed.
2. No red nodes have been made adjacent. Because y takes z's place in the tree, along with z's color, we cannot have two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. If y is red, then x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
3. Since y could not have been the root if it was red, the root remains black.

Deletion in Red-Black Tree

- If node y was black, three problems may arise:

1. If y had been the root and a red child of y becomes the new root, we have violated property 2.
2. If both x and x.p are red, then we have violated property 4.
3. moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree.

RB-DELETE-FIXUP

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