Design and Analysis of Algorithms

Unit-2

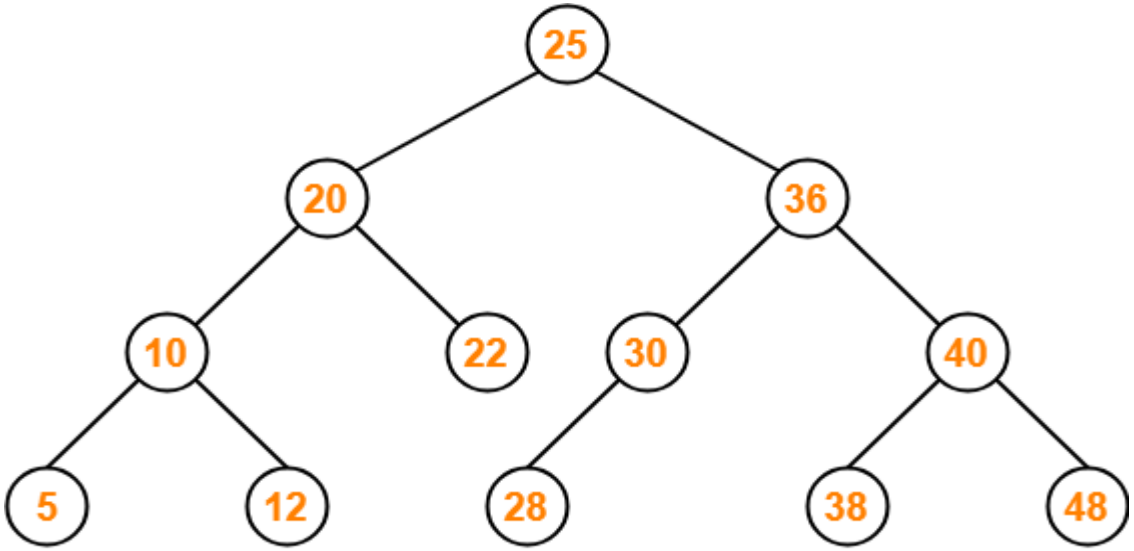
Red-Black Tree

Red-Black Tree

Binary Search Tree(BST)

A binary tree is said to be binary search tree if the value at the left child is less than value at the parent node and value at the right child is greater or equal than value at the parent node.

Example:



Binary Search Tree

Red-Black Tree

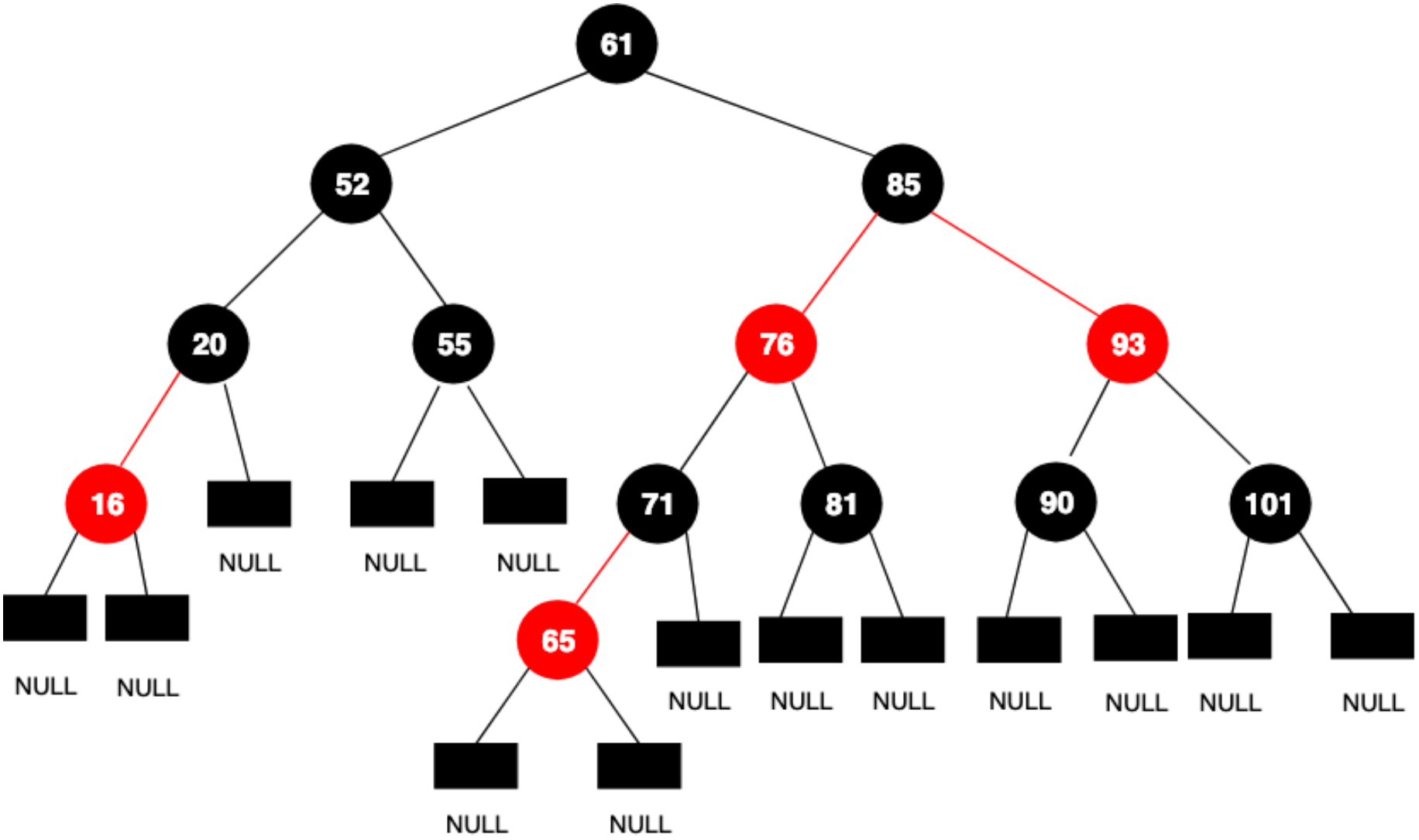
Definition

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

1. Every node is either red or black.
2. The root is black.
3. The color of every leaf node is black and its value is always NIL.
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.

Red-Black Tree

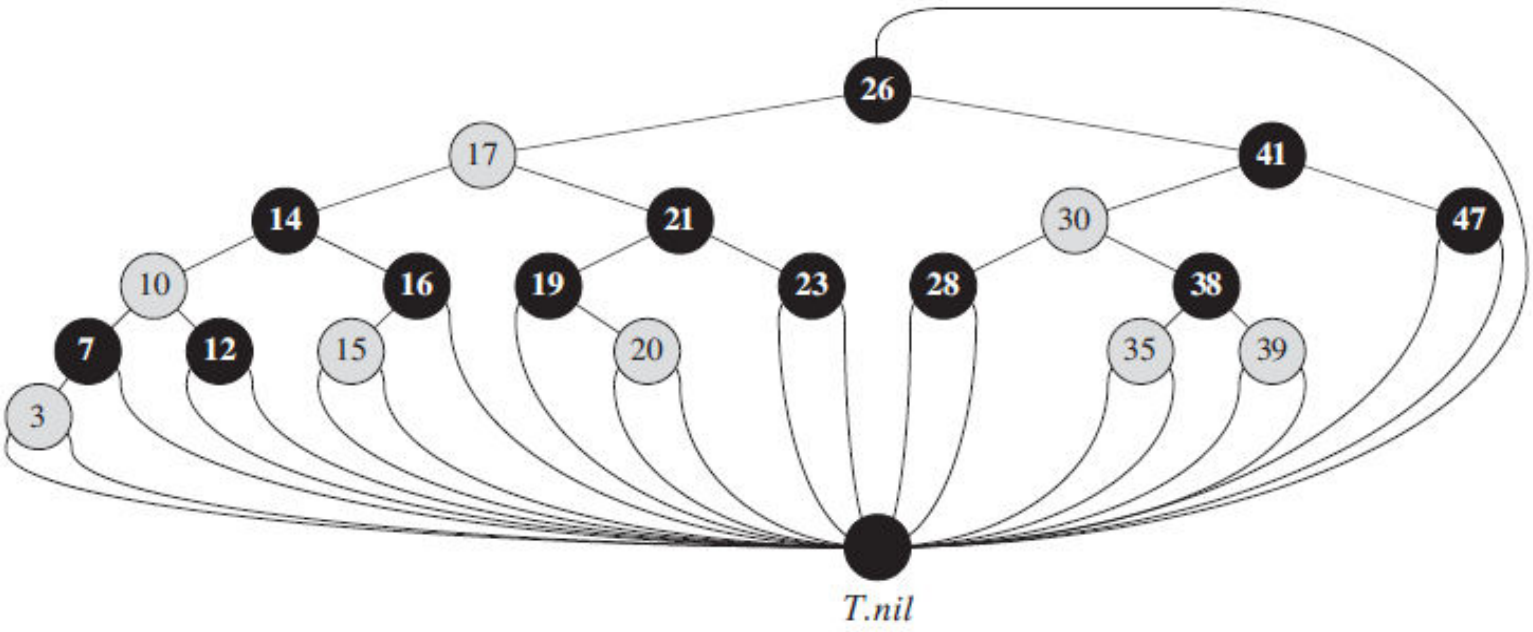
Example:



Red-Black Tree

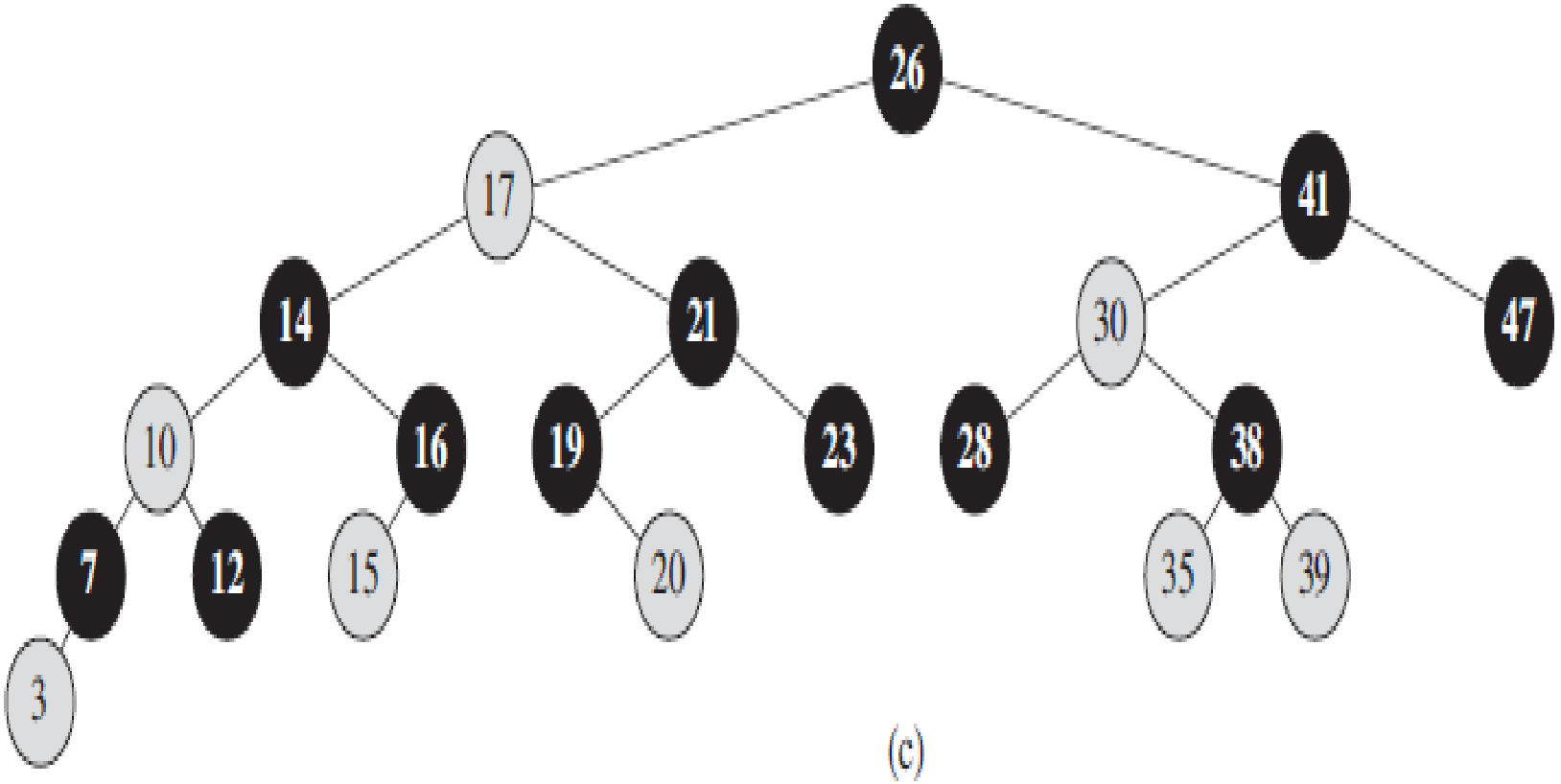
Sentinel

For a red-black tree T, the sentinel is an object with the same attributes as an ordinary node in the tree. Its color is black and its value is nil. It is represented by T.nil or nil[T].



Red-Black Tree

Red-Black tree without leaf node



(c)

Red-Black Tree

Black height of a node

Black height of a node x in the red-black tree is the number of black nodes on any downward path from node x to a leaf node but not including node x. It is denoted by bh(x).

Black height of a Red-Black tree

Black height of a red-black tree is equal to the black height of the root node.

Red-Black Tree

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

Proof: Before proving the theorem, first we will prove the following statement.

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

We will prove the statement (1) using induction method. We will use induction parameter as the height of the red-black tree.

For height, h = 0.

Red-black tree of height 0 is only single node i.e. it will be following :-

Clearly, minimum number of internal nodes in this tree = 0

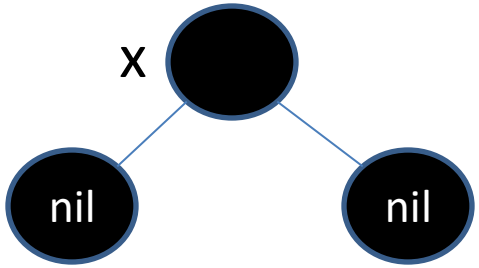
and $2^{bh(x)}-1 = 2^{0}-1 = 1-1 = 0$

Therefore, statement (1) is true for height h=0.

Red-Black Tree

For height, h = 1.

Red-black tree of height h = 1 will be the following:-



Clearly, minimum number of internal nodes in this tree = 1

And $2^{bh(x)}-1 = 2^{1}-1 = 2-1 = 1$

Therefore, statement (1) is also true for height h=1.

Now, we assume statement (1) is true for children of node x.
We will prove the statement for node x.

Red-Black Tree

Now,

The minimum number of internal nodes in the subtree rooted at node x

= Minimum number of internal nodes in the subtree rooted at left child of node x + Minimum number of internal nodes in the subtree rooted at right child of node x + 1

Since black height of each child of node x has either bh(x) or bh(x)-1, depending on the color of child. If the color of the child is red then black height of child is bh(x) but if the color of the child is black then black height of child is bh(x) -1. Therefore,

The minimum number of internal nodes in the subtree rooted at node x

$$= 2^{(bh(x)-1)} - 1 + 2^{(bh(x)-1)} - 1 + 1$$

$$= 2 \cdot 2^{(bh(x)-1)} - 1$$

$$= 2^{bh(x)} - 1$$

Therefore, statement (1) is also proved for node x of any height.

Red-Black Tree

Now, we will prove the given theorem using statement (1).

Let h is the height of the red-black tree.

Using property (4) of the red-black tree, minimum number of black nodes on any path from root node to the leaf node will be h/2. Therefore, minimum black height of red-black tree of height h will be h/2.

Now, the minimum number of internal nodes in red-black tree of height h $= 2^{h/2} - 1$

Since the gien number of internal nodes in red-black tree is n, therefore

$$2^{h/2} - 1 \leq n \Rightarrow 2^{h/2} \leq n + 1$$

$$\Rightarrow h/2 \leq \lg(n+1) \Rightarrow h \leq 2\lg(n+1)$$

Therefore, the maximum height of red-black tree will be $2\lg(n+1)$.

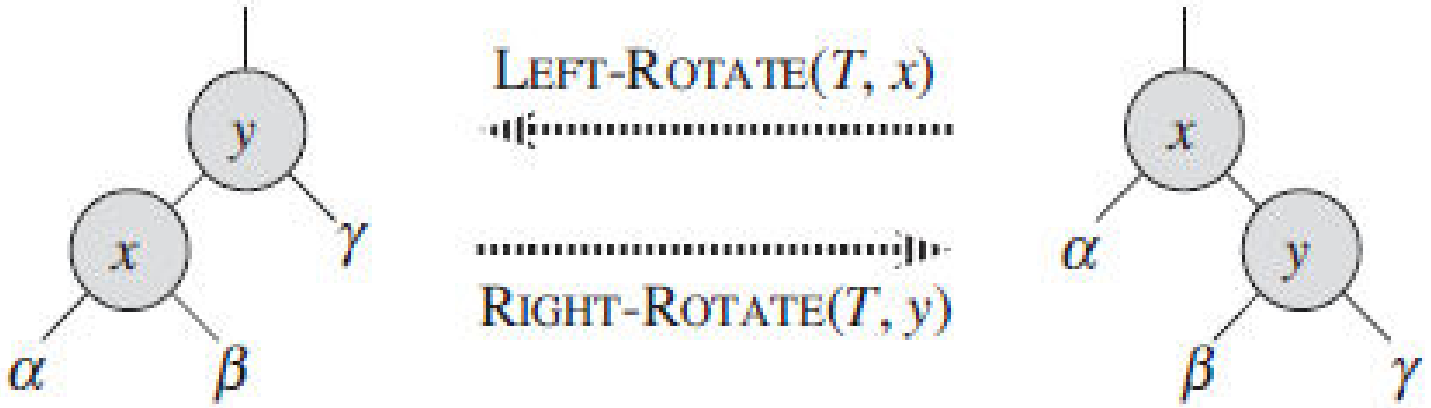
Now, it is proved.

Rotation operation

We use two types of rotations in the insertion and deletion of a node in the red-black tree.

(1) Left rotation

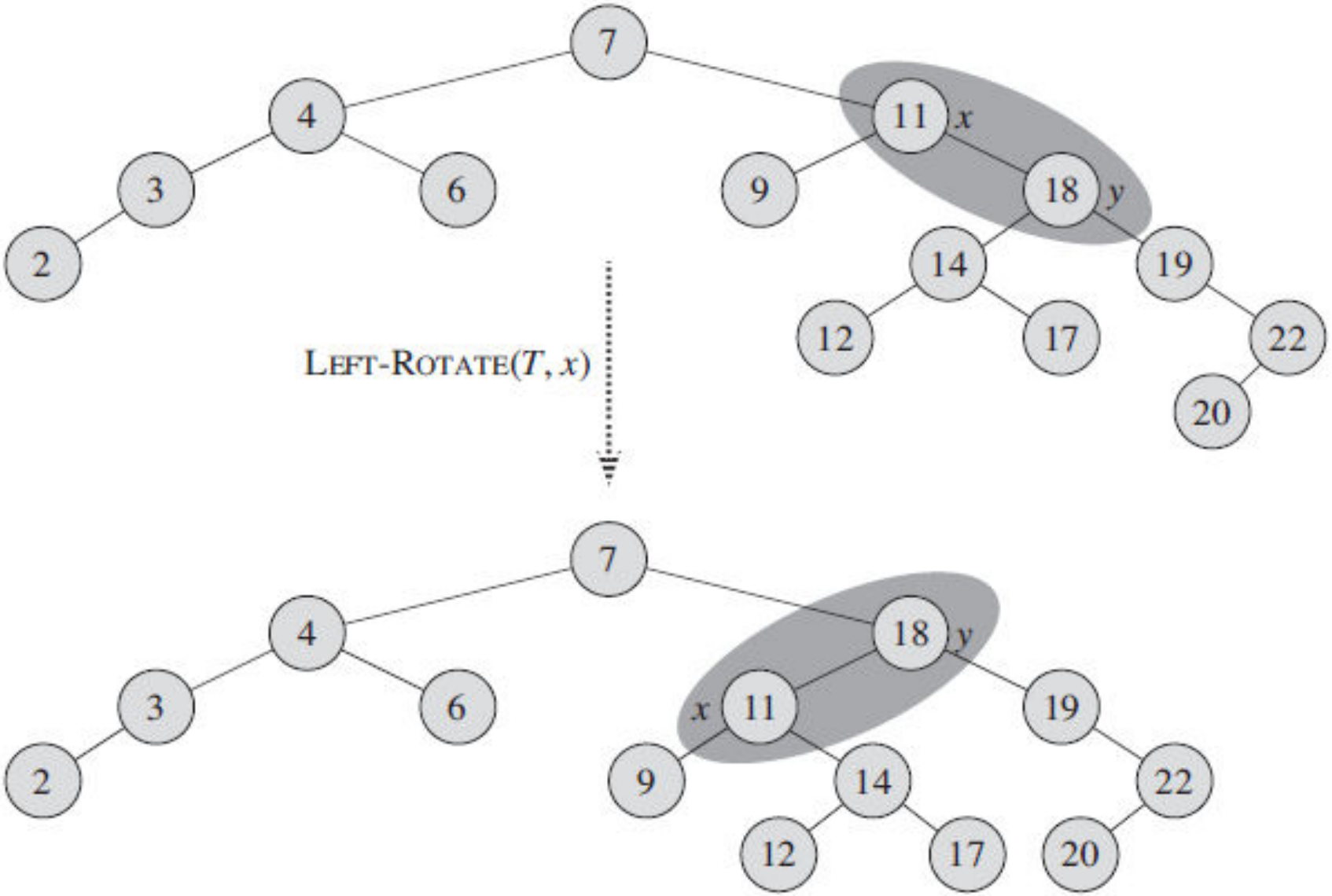
(2) Right rotation



Left rotation algorithm

```
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
```

Left rotation algorithm



Right rotation algorithm

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

Insertion operation

Suppose we want to insert a node z into red-black tree T. We use the following steps for this purpose:-

1. Insert node z into red-black tree using binary search tree insertion process.
2. Make the color of new node z to be **red**.
3. If the color of parent of node z is **black** then we make the color of root node to be **black** and stop the process.
4. Otherwise we maintain the properties of red-black tree using the following procedure.

    (4-a) We start a loop and continue until color of parent of node z turns **black**.

    (4-b) If parent of node z is the left child of its parent then we do the following actions:-

    (4b-i) Find the sibling of parent of node z i.e. uncle of z. Let it is denoted by node y.

Insertion operation

(4b-ii) Now, there will be three cases.

Case-1: If color of y is **red** then we do the following actions:-

(1) z.p.color = **black**

(2) y.color = **black**

(3) z.p.p.color = **red**

(4) z = z.p.p

Case-2: If color of y is **black** and z is right child then we do following actions:-

(1) z = z.p

(2) Perform left rotation at node z.

Case-3: If color of y is **black** and z is left child then we do following actions:-

(1) z.p.color = **black**

(2) z.p.p.color = **red**

(3) Perform right rotation at node z.p.p i.e. at grandparent of z

Insertion operation

(4-c) If parent of node z is the right child of its parent then we do the following actions:-

(4c-i) Find the sibling of parent of node z i.e. uncle of z. Let it is denoted by node y.

(4c-ii) Now, there will be three cases.

Case-1: If color of y is **red** then we do the following actions:-

(1) z.p.color = **black**

(2) y.color = **black**

(3) z.p.p.color = **red**

(4) z = z.p.p

Insertion operation

Case-2: If color of y is **black** and z is left child then we do following actions:-

(1) z = z.p

(2) Perform right rotation at node z.

Case-3: If color of y is **black** and z is right child then we do following actions:-

(1) z.p.color = **black**

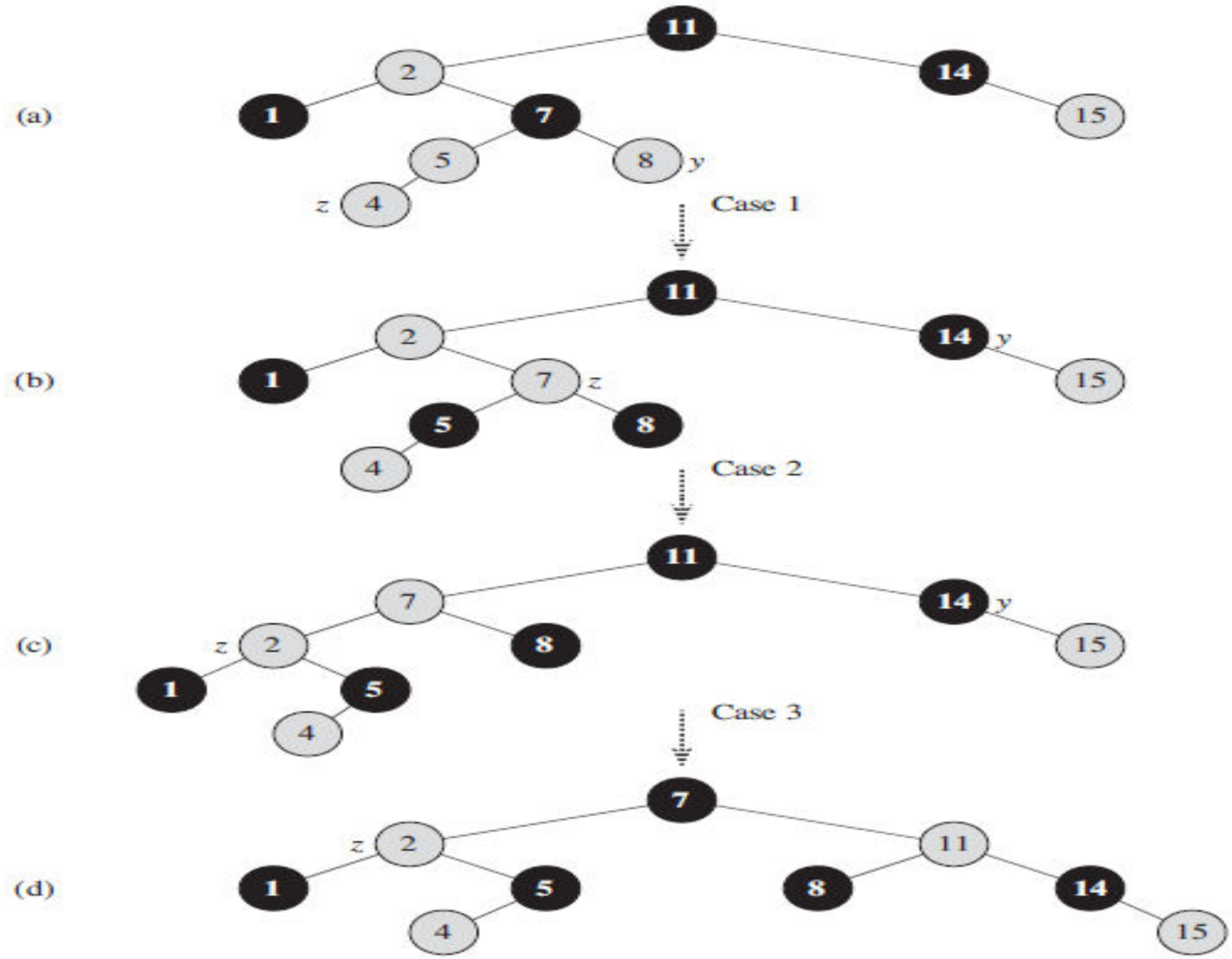
(2) z.p.p.color = **red**

(3) Perform left rotation at node z.p.p i.e. at grandparent of z

(4-d) After exit from the loop, we make the color of root node to be **black** i.e.

T.root.color = **black**

Insertion operation

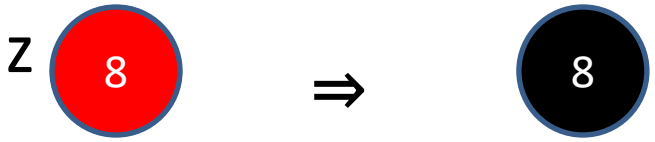


Insertion operation

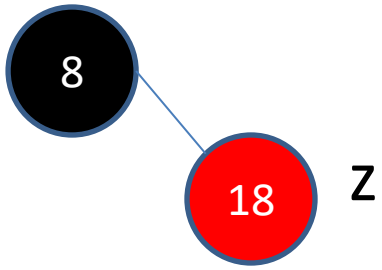
Example: Create the red-black tree by inserting following sequence of numbers:- 8, 18, 5, 15, 17, 25, 40 and 80.

Initially, red-black tree is empty.

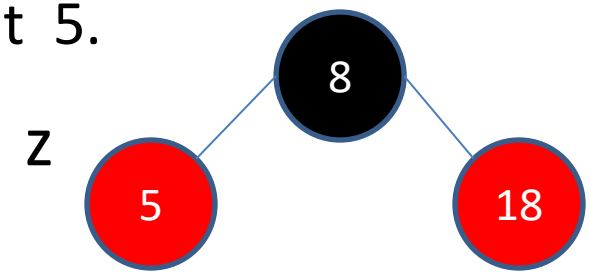
Solution: Initially tree is empty. First insert element 8.



Now, Insert next element 18.

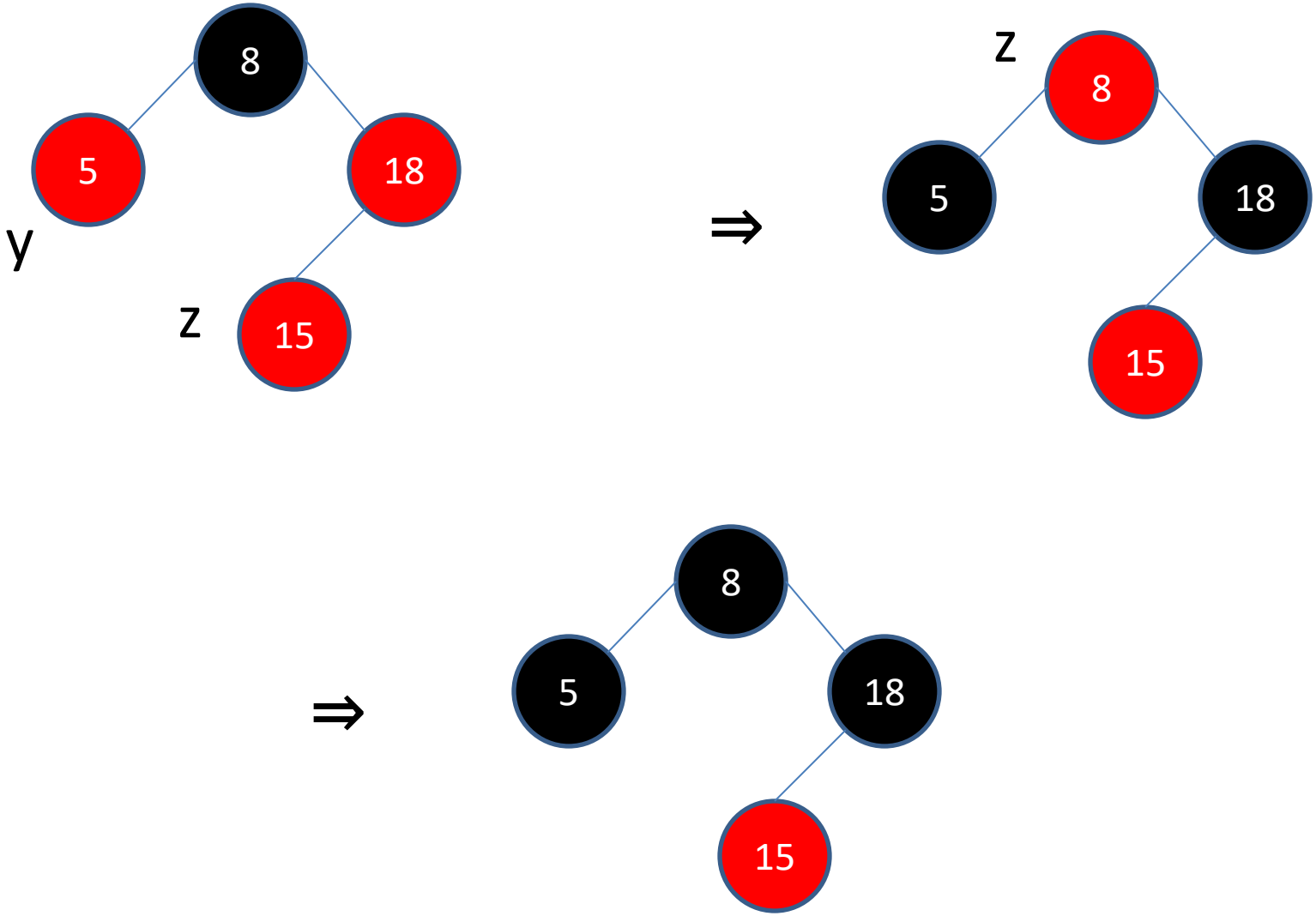


Now, Insert next element 5.



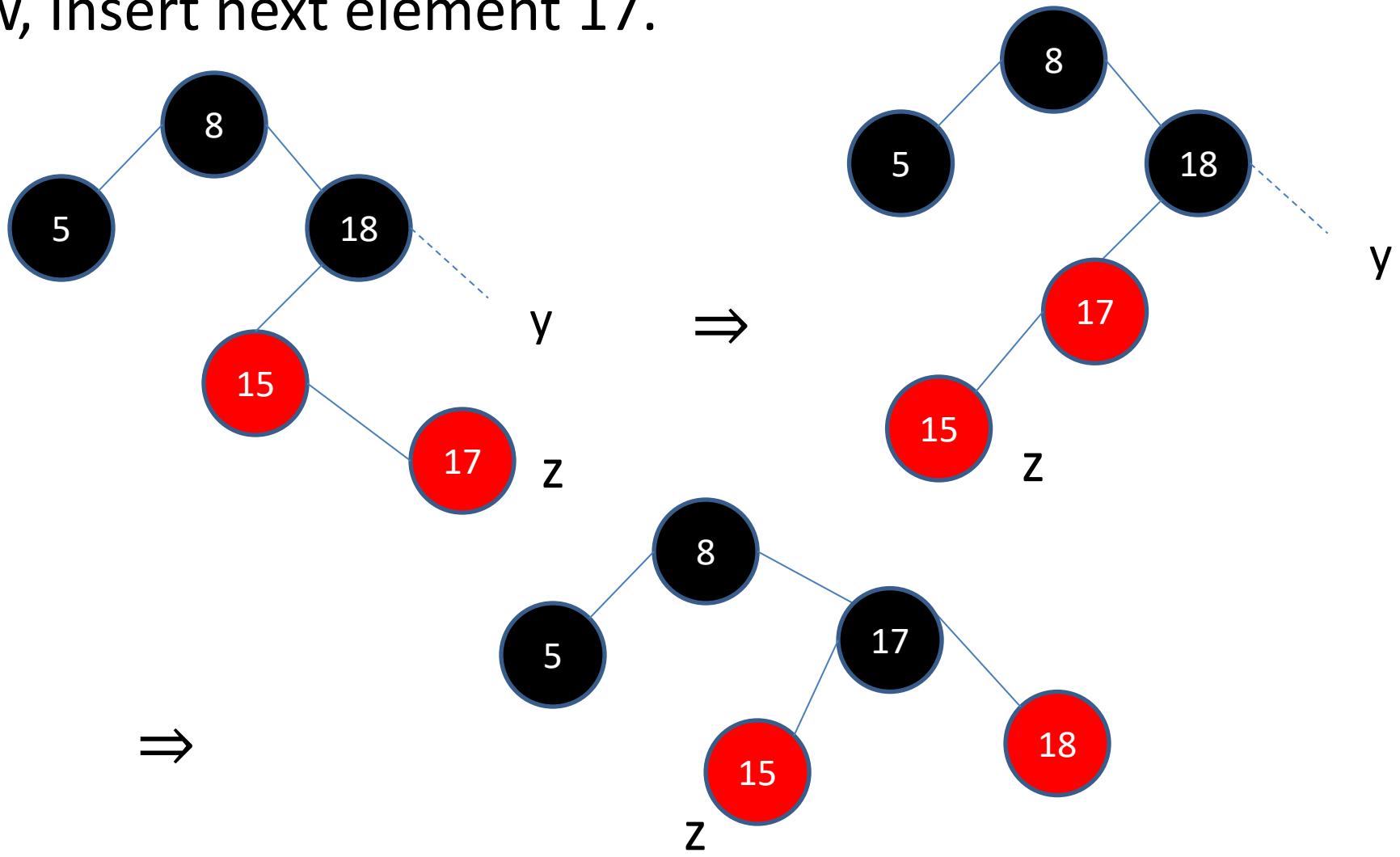
Insertion operation

Now, Insert next element 15.



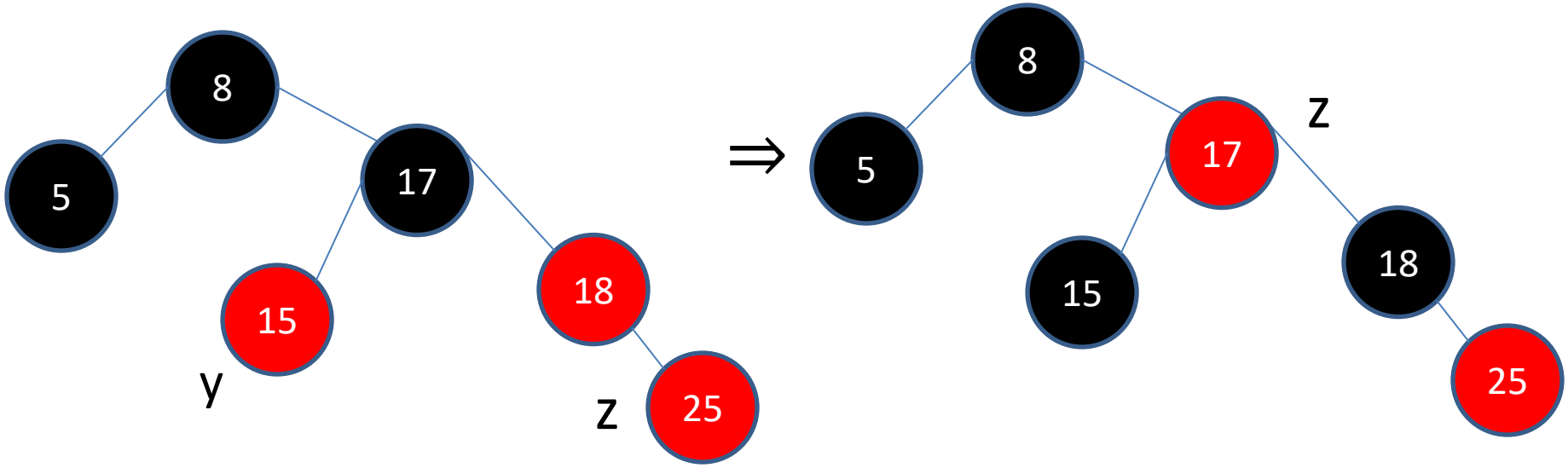
Insertion operation

Now, Insert next element 17.



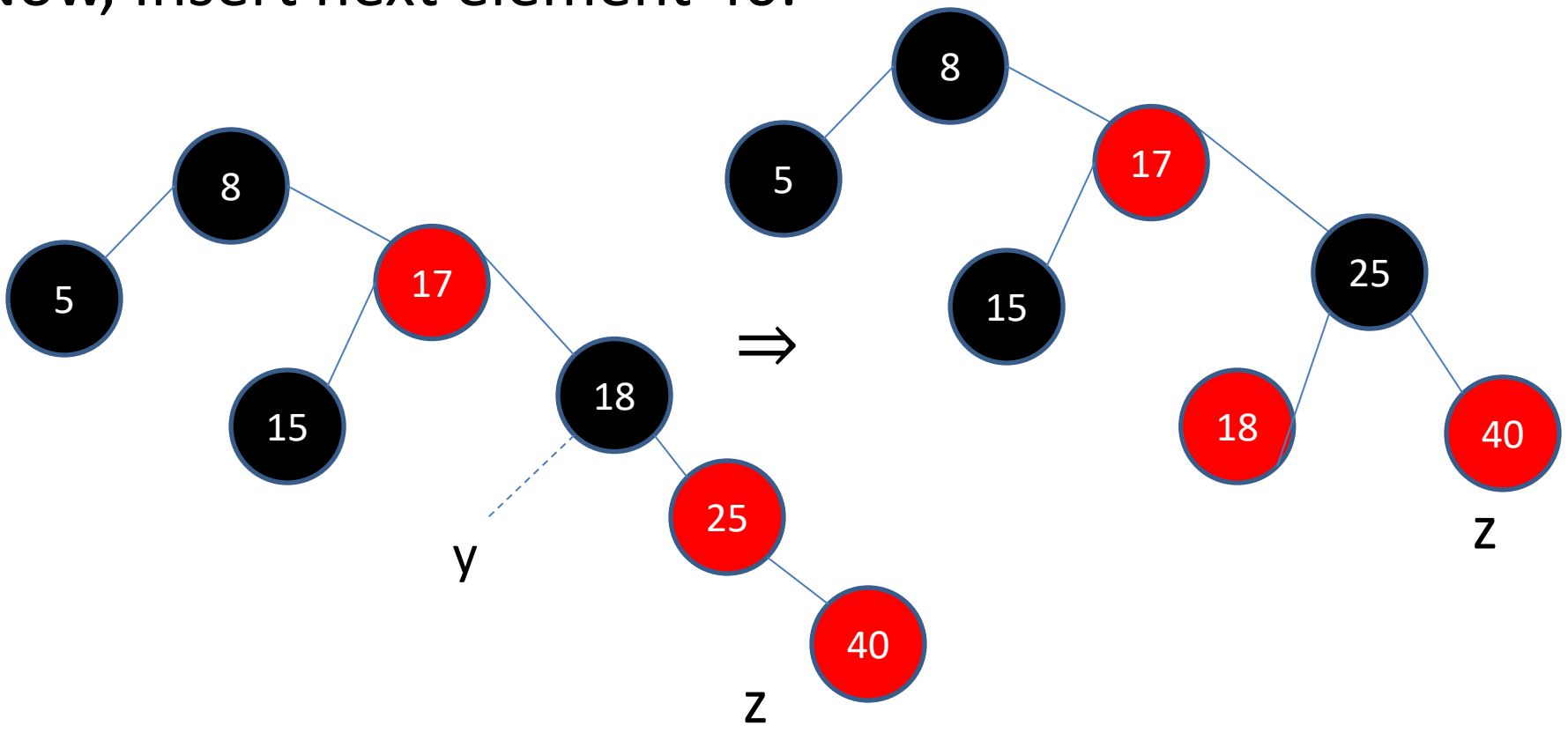
Insertion operation

Now, Insert next element 25.



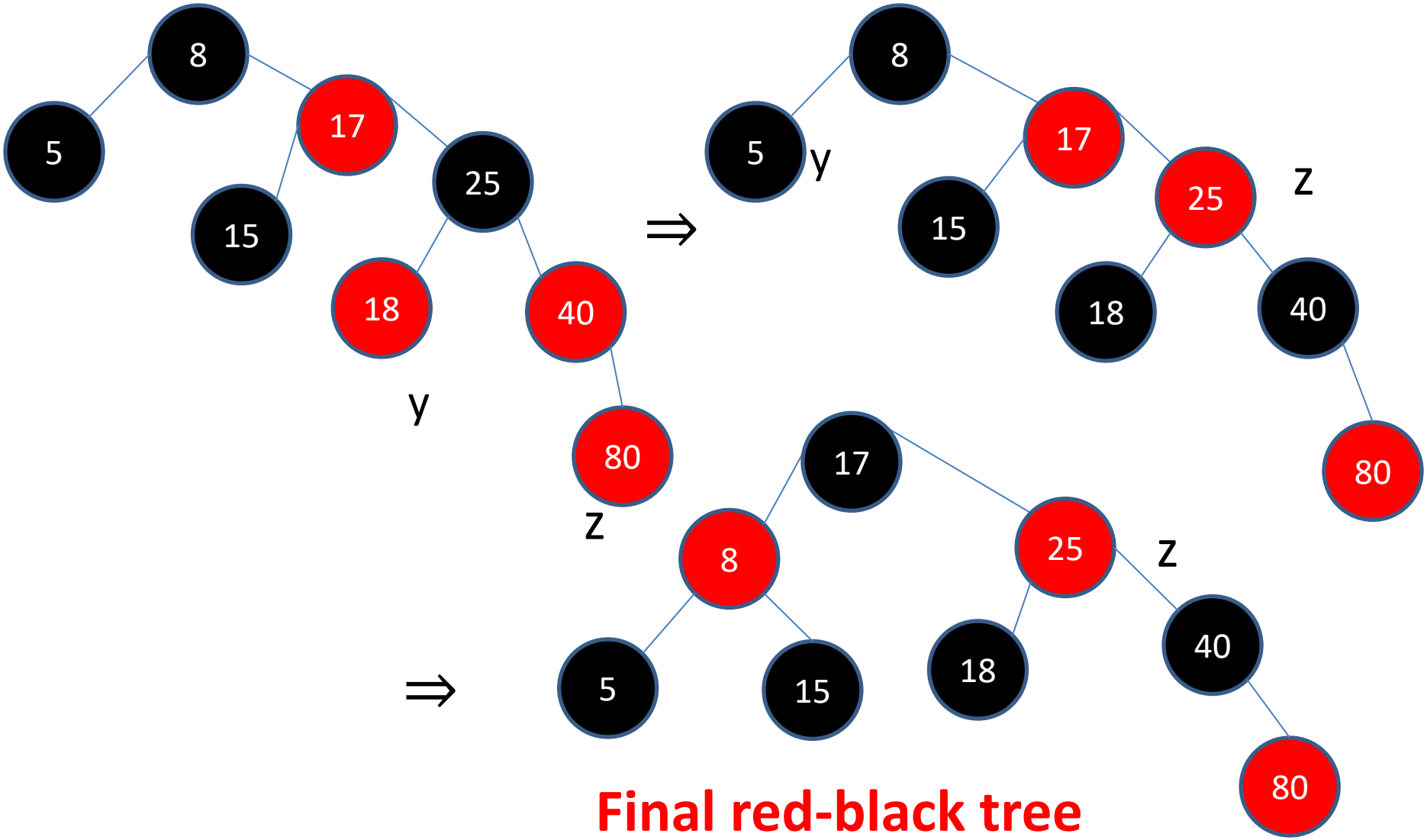
Insertion operation

Now, Insert next element 40.



Insertion operation

Now, Insert next element 80.



Final red-black tree

Insertion Algorithm

```
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)
```

Insertion Algorithm

```
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
```

Time complexity of insertion algorithm is $\theta(\log n)$.

Deletion operation

Suppose we want to delete a node z from a red-black tree T.
We use the following steps for this purpose:-

1. First we delete node z using binary search tree deletion process.
2. Find node y in the following way:-
   - ❖ If node z has two children then y will be successor of z otherwise y will be z.
3. After finding y, we find x in the following way:-
   - ❖ If node y has left child then x will be left child of y otherwise x will be right child of y.
4. If color of y is **red**, then we terminate the process.
5. If color of y is **black**, then we maintain the properties of red-black tree in the following way:-

Deletion operation

(5-a) We start and continue a loop if x is not root node and color of x is **black**.

(5-b) if x is the left child then we do the following actions:-

(i) Find sibling of x. Let it is denoted by w.

(ii) There will be four cases:-

Case-1: If color of w is **red**, then we do the following actions:-

(1) w.color = **black**

(2) x.p.color = **red**

(3) Apply left rotation at parent of node x.

Case-2: If color of w is **black** and color of its both children is also **black**, then we do the following actions:-

(1) w.color = **red**

(2) x = x.p

Deletion operation

Case-3: If color of w is **black** and color of its left child is **red** and color of its right child is **black**, then we do the following actions:-

(1) w.left.color = **black**

(2) w.color = **red**

(3) Apply right rotation at node w.

Case-4: If color of w is **black** and color of its right child is **red**, then we do the following actions:-

(1) w.right.color = **black**

(2) w.color = x.p.color

(3) x.p.color = **black**

(4) Apply left rotation at parent of node x.

(5) x = T.root

Deletion operation

(5-c) if x is the right child then we do the following actions:-

(i) Find sibling of x. Let it is denoted by w.

(ii) There will be four cases:-

Case-1: If color of w is **red**, then we do the following actions:-

(1) w.color = **black**

(2) x.p.color = **red**

(3) Apply right rotation at parent of node x.

Case-2: If color of w is **black** and color of its both children is also **black**, then we do the following actions:-

(1) w.color = **red**

(2) x = x.p

Deletion operation

Case-3: If color of w is **black** and color of its right child is **red** and color of its left child is **black**, then we do the following actions:-

(1) w.right.color = **black**

(2) w.color = **red**

(3) Apply left rotation at node w.

Case-4: If color of w is **black** and color of its left child is **red,** then we do the following actions:-

(1) w.left.color = **black**

(2) w.color = x.p.color

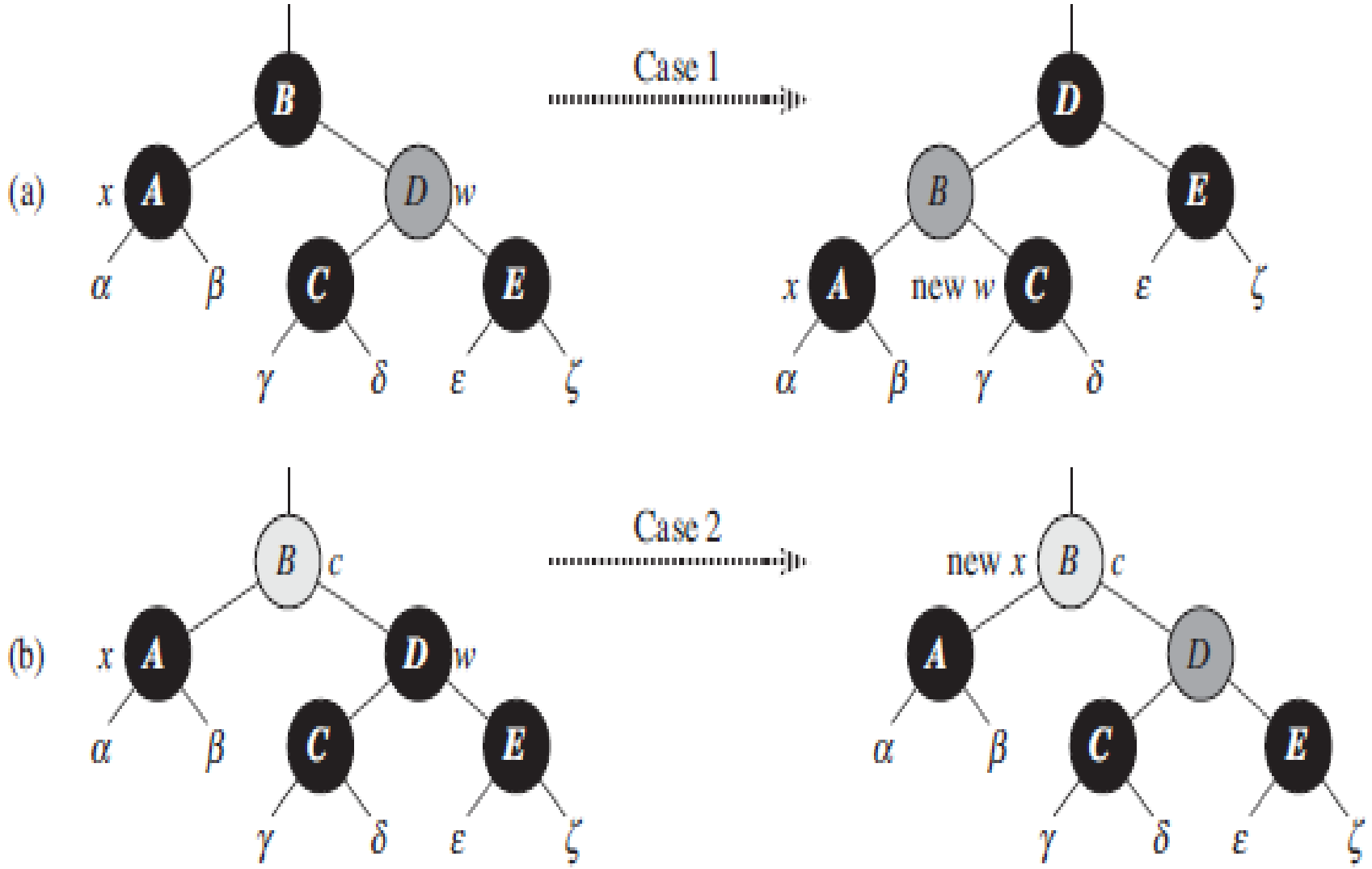
(3) x.p.color = **black**

(4) Apply right rotation at parent of node x.

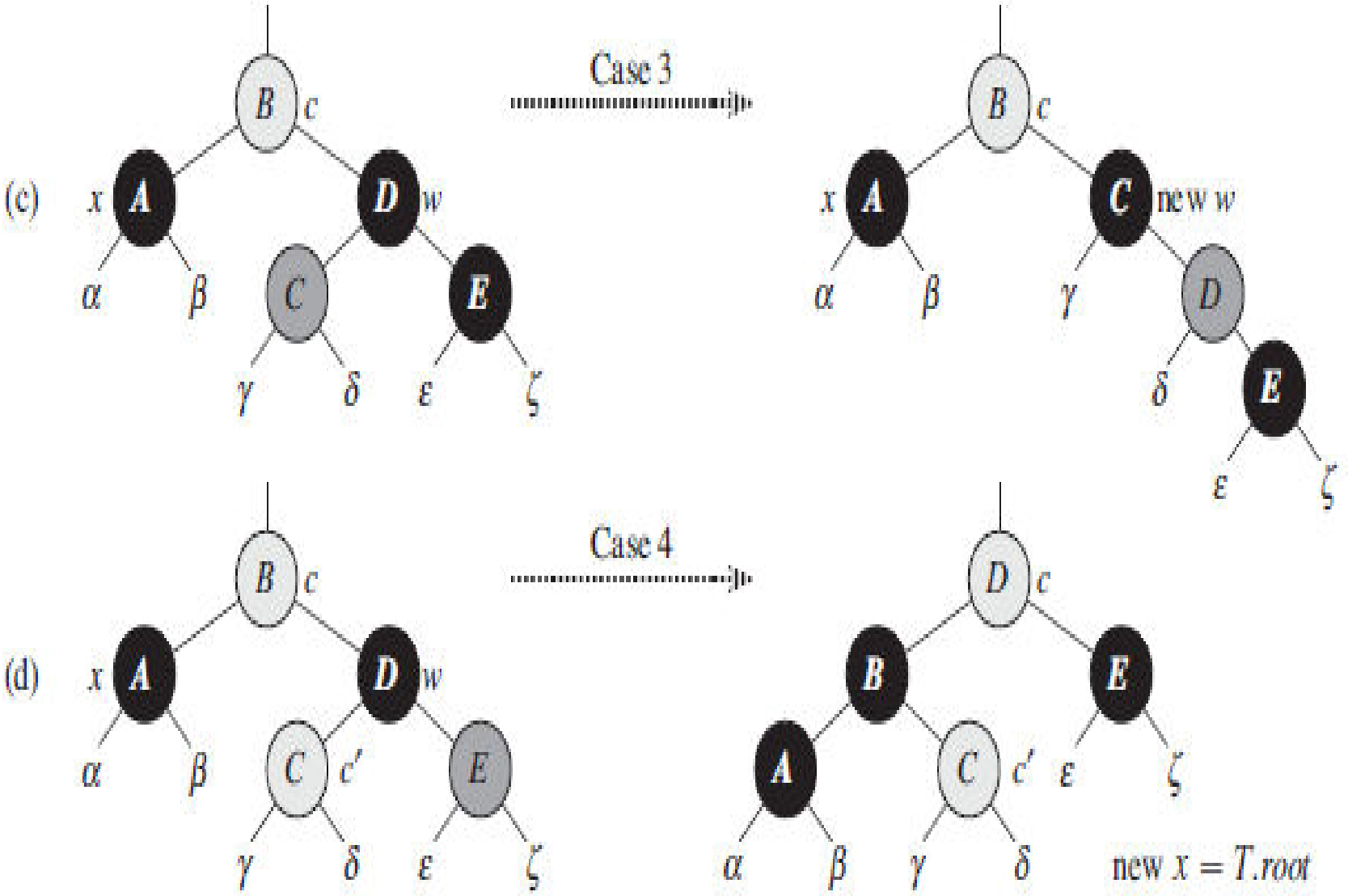
(5) x = T.root

(5-d) After exit from the loop, we make the color of node x to the **black** i.e. x.color = **black**

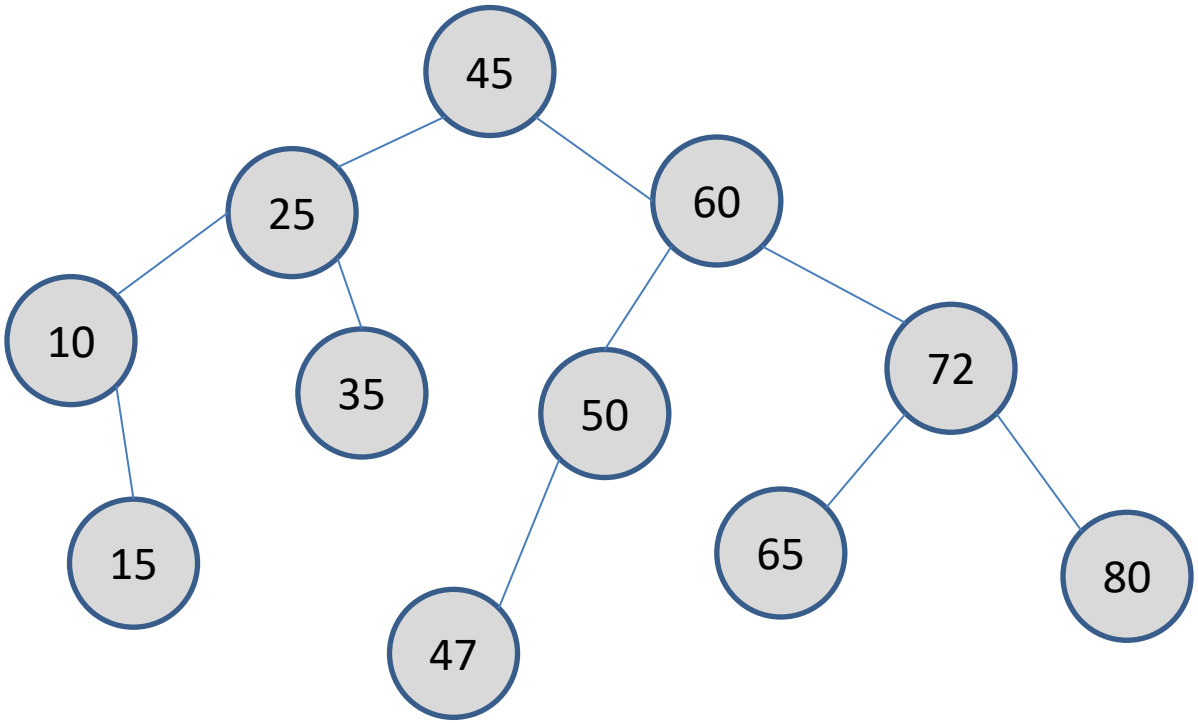
Deletion operation



Deletion operation

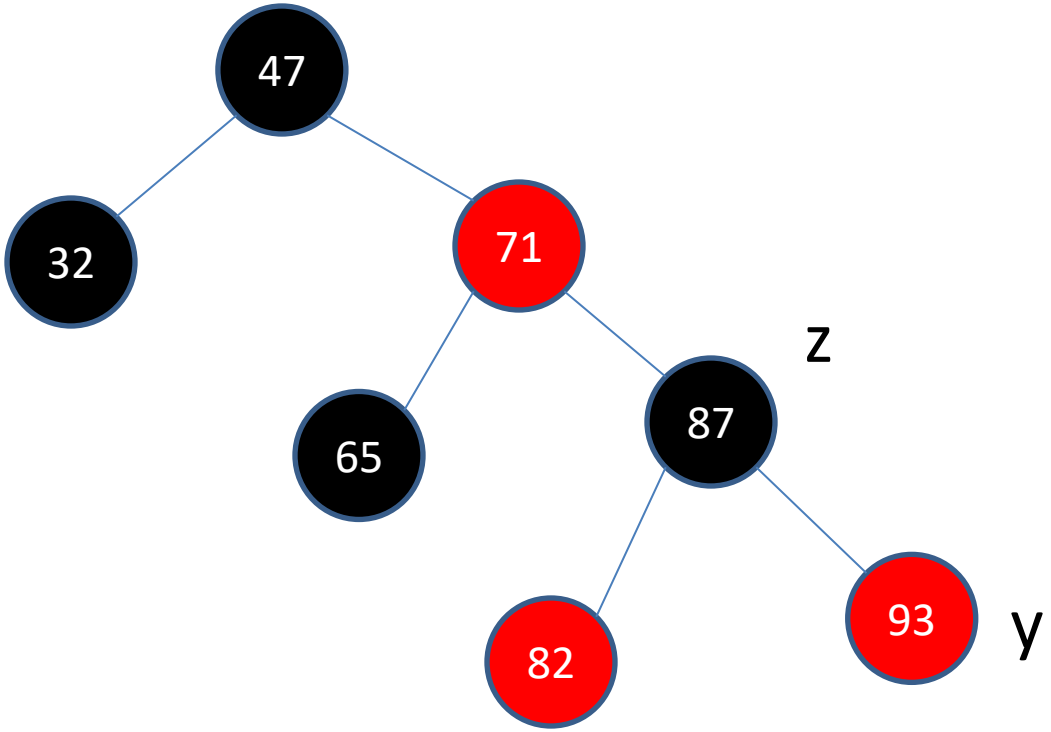


Binary search tree deletion



Deletion operation

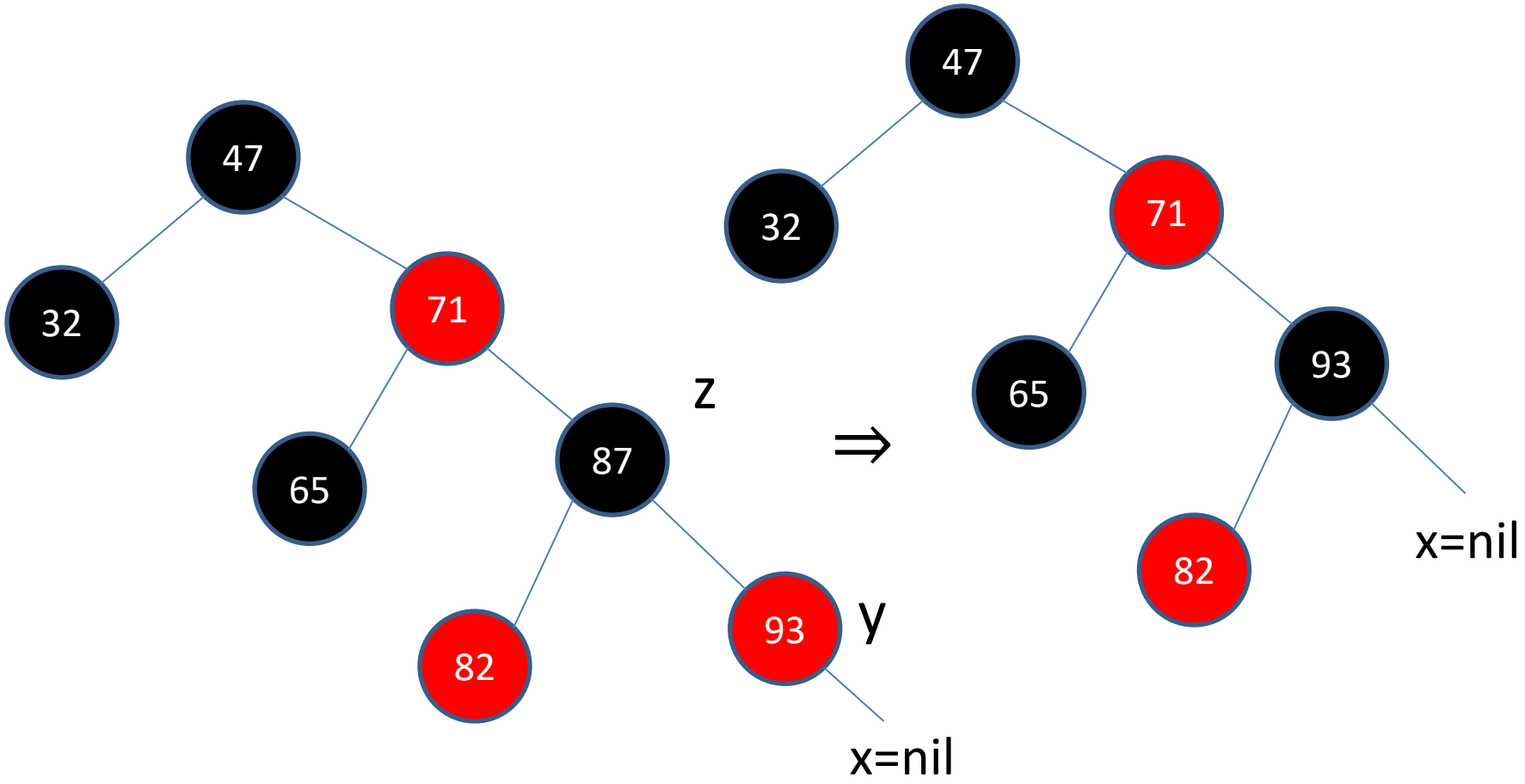
Example: Consider the following red-black tree



Delete the element 87, 32 and 71 in order.

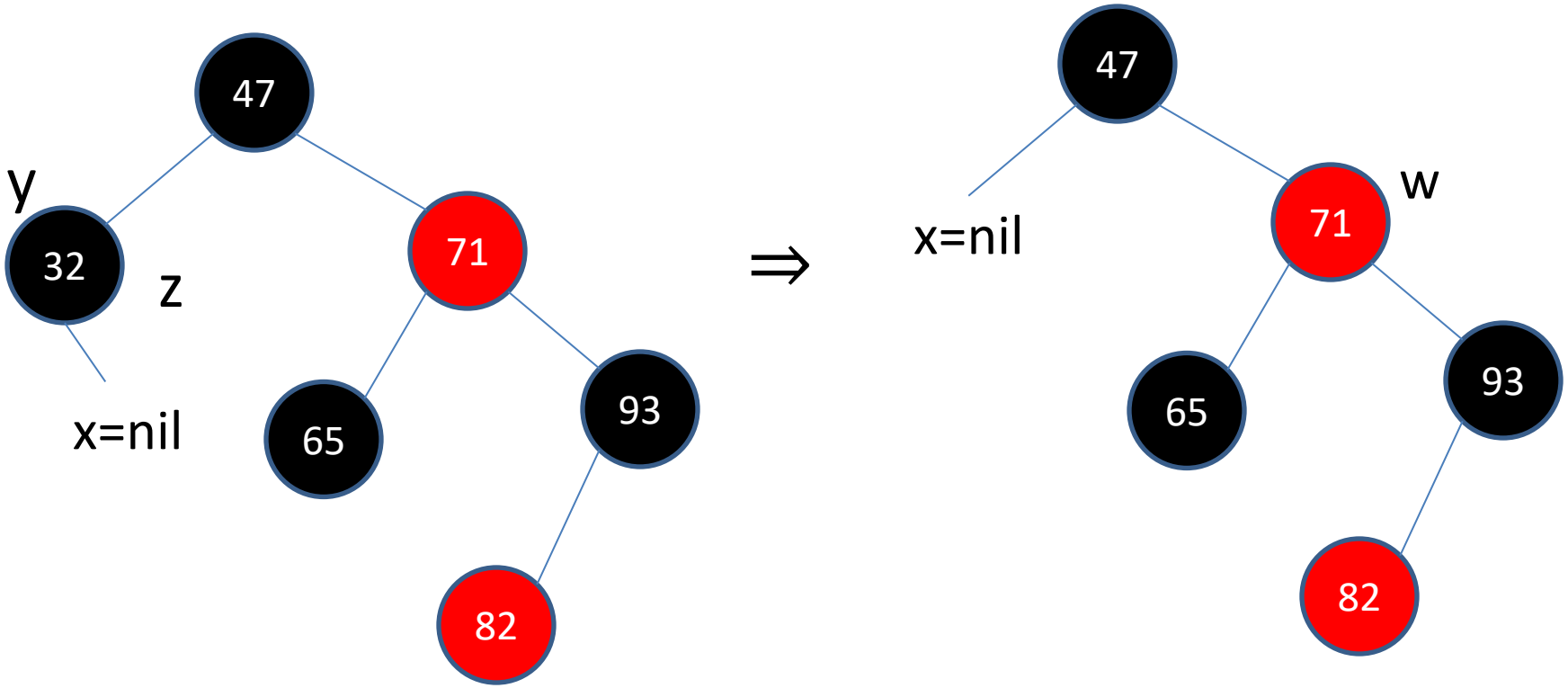
Deletion operation

Deletion of 87



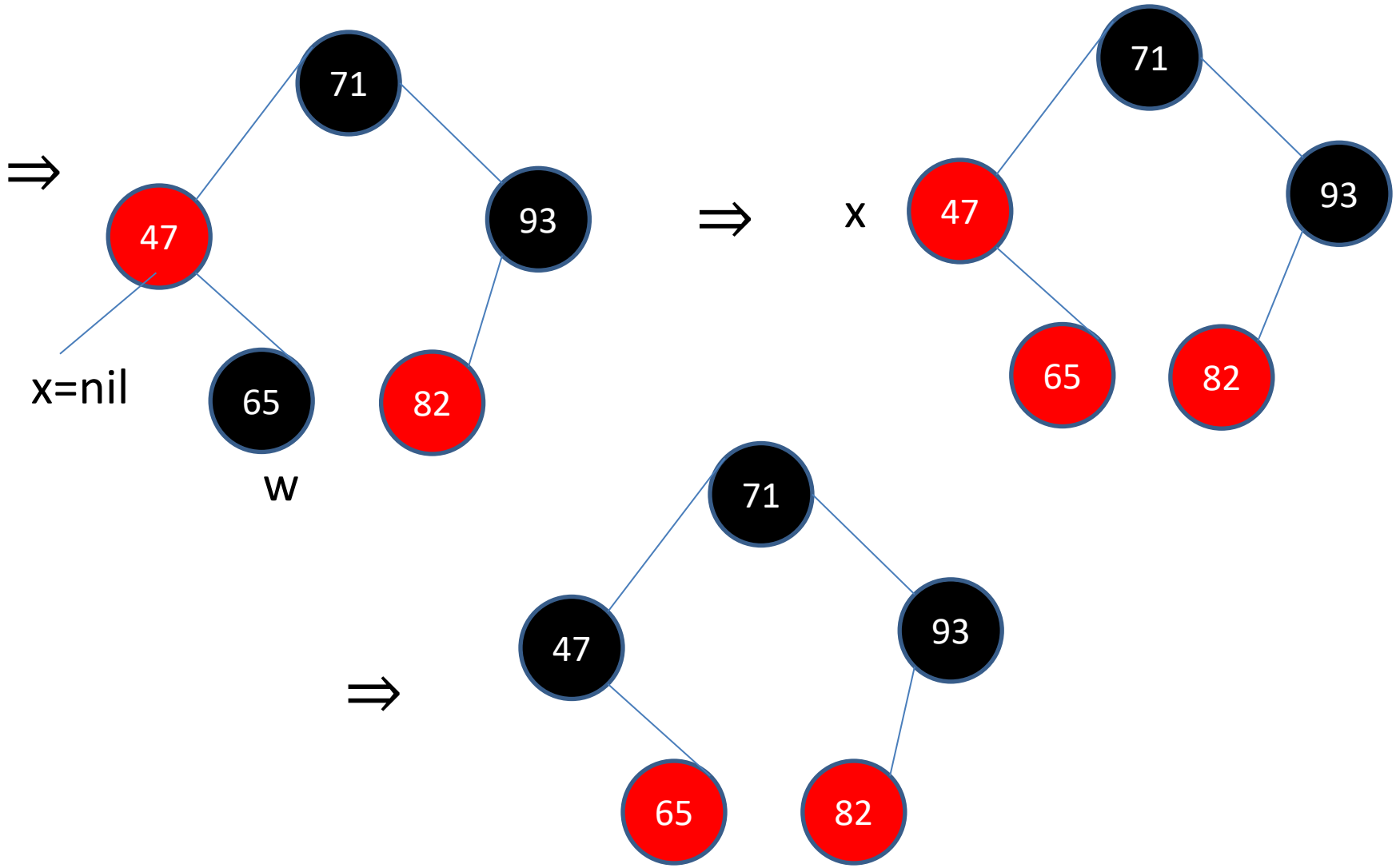
Deletion operation

Deletion of 32



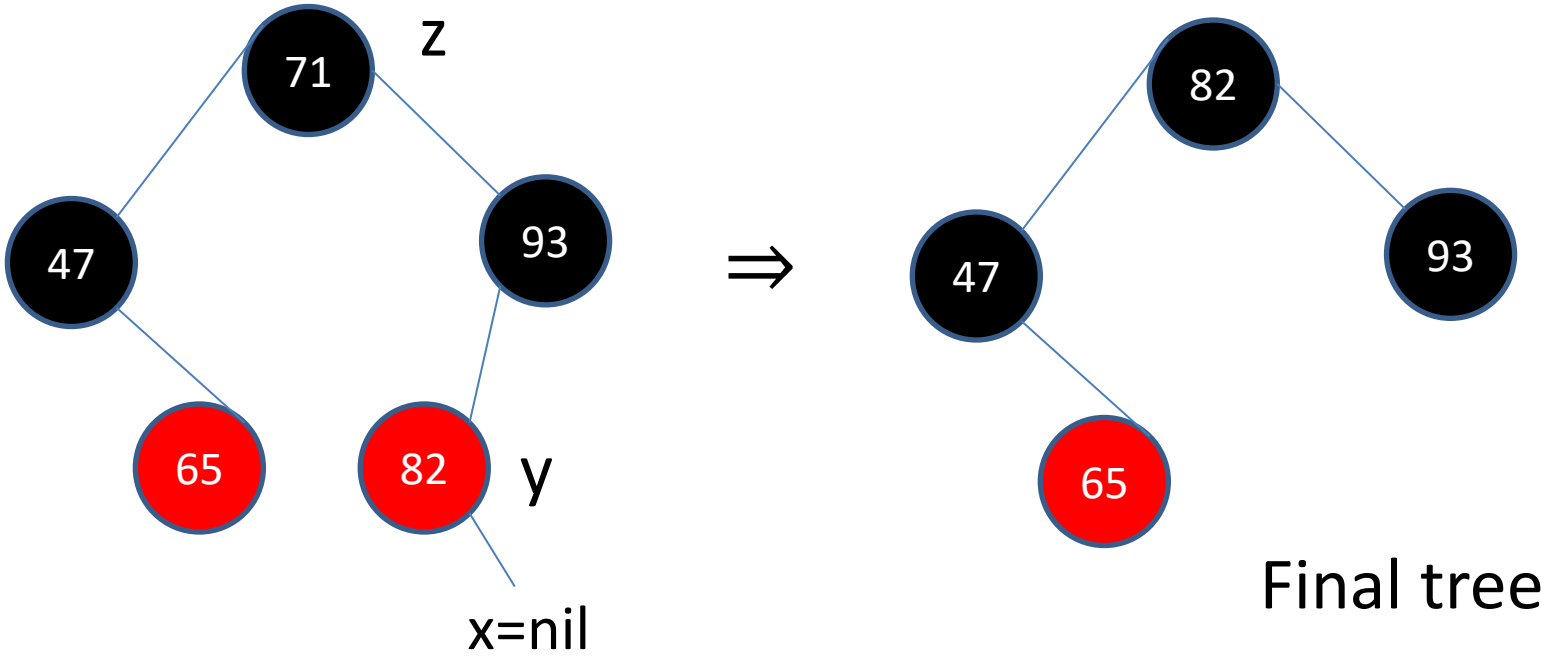
Deletion operation

Deletion of 32(continue)



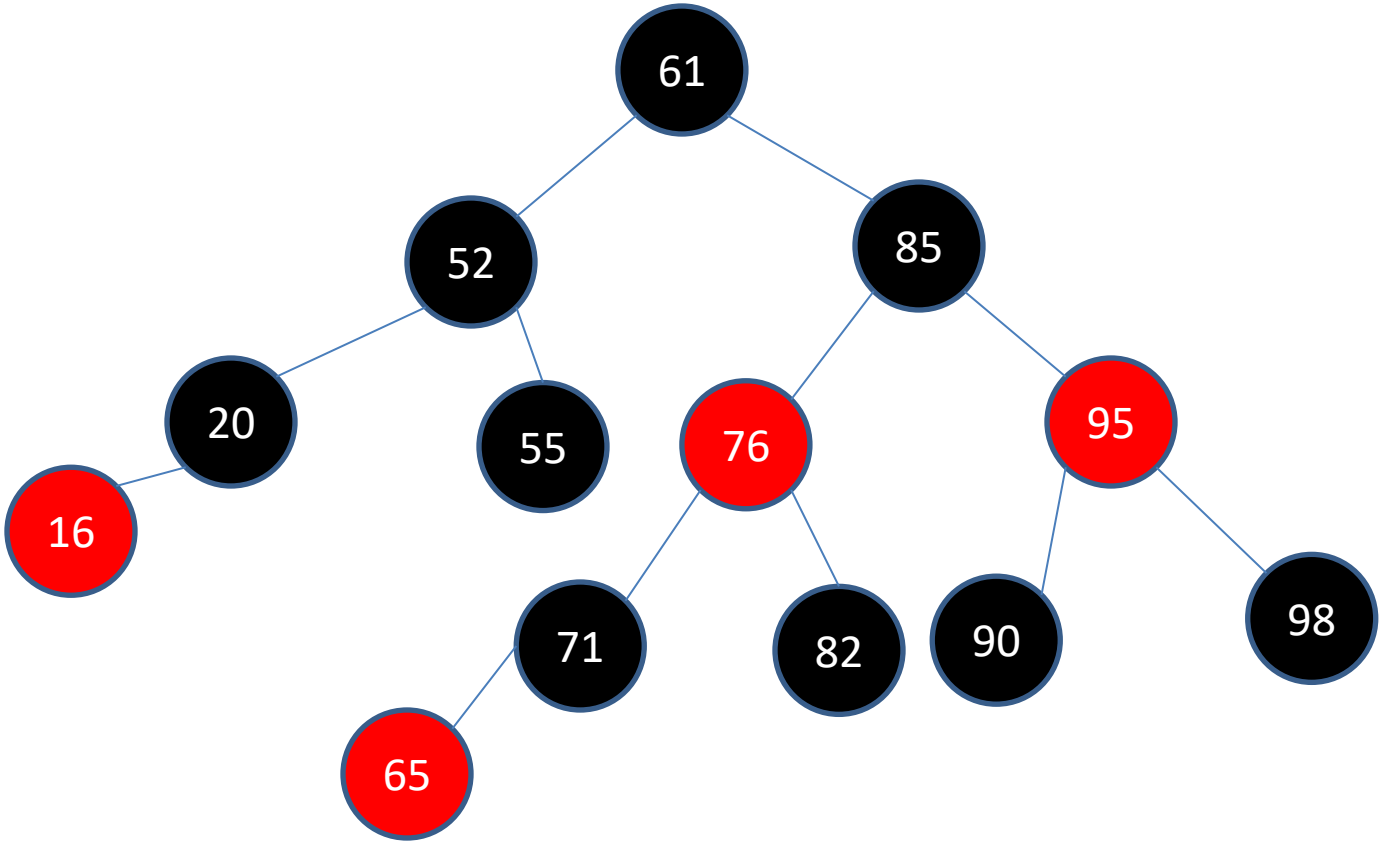
Deletion operation

Deletion of 71



Deletion operation

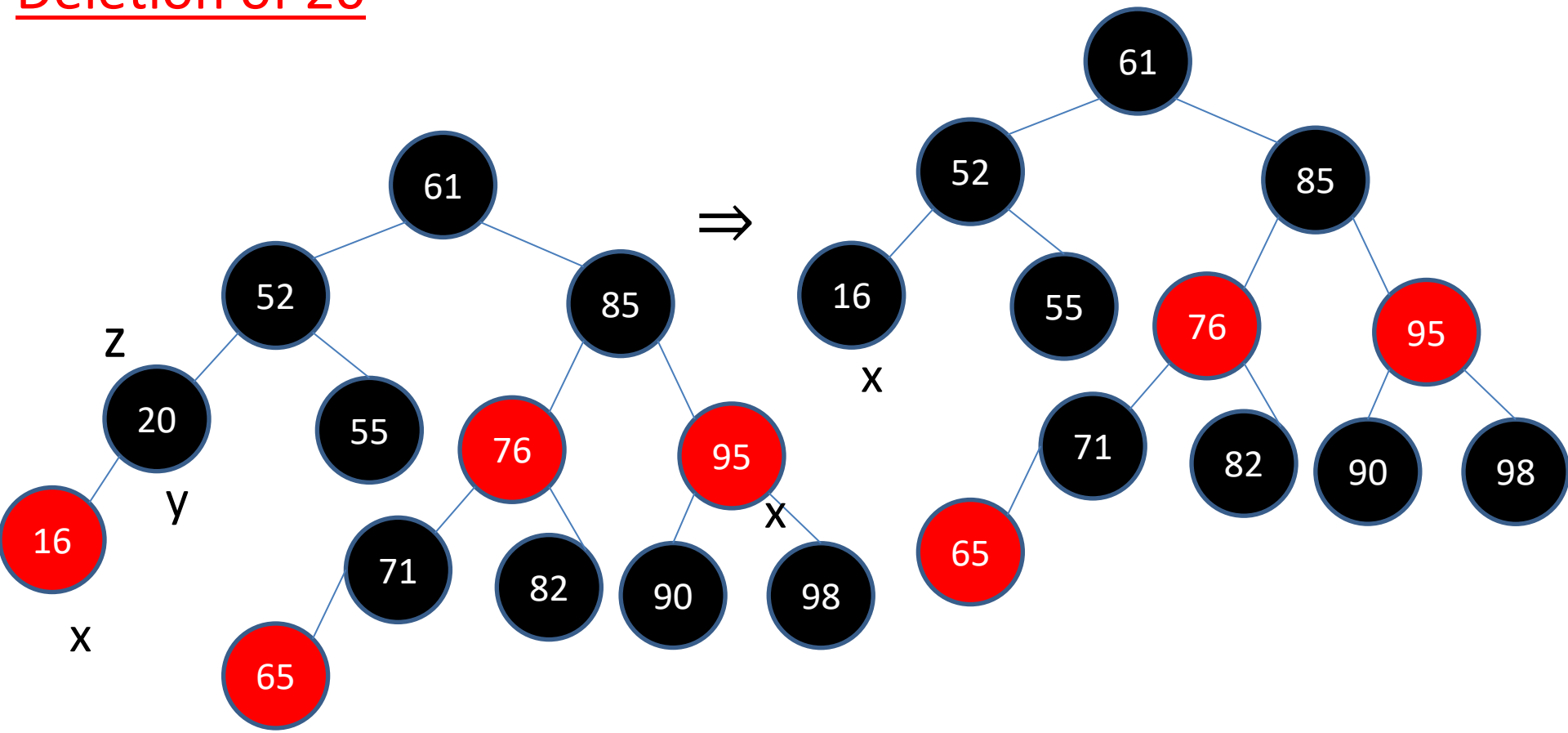
Example: Consider the following red-black tree



Delete the element 20, 85, 95 and 98 in order.

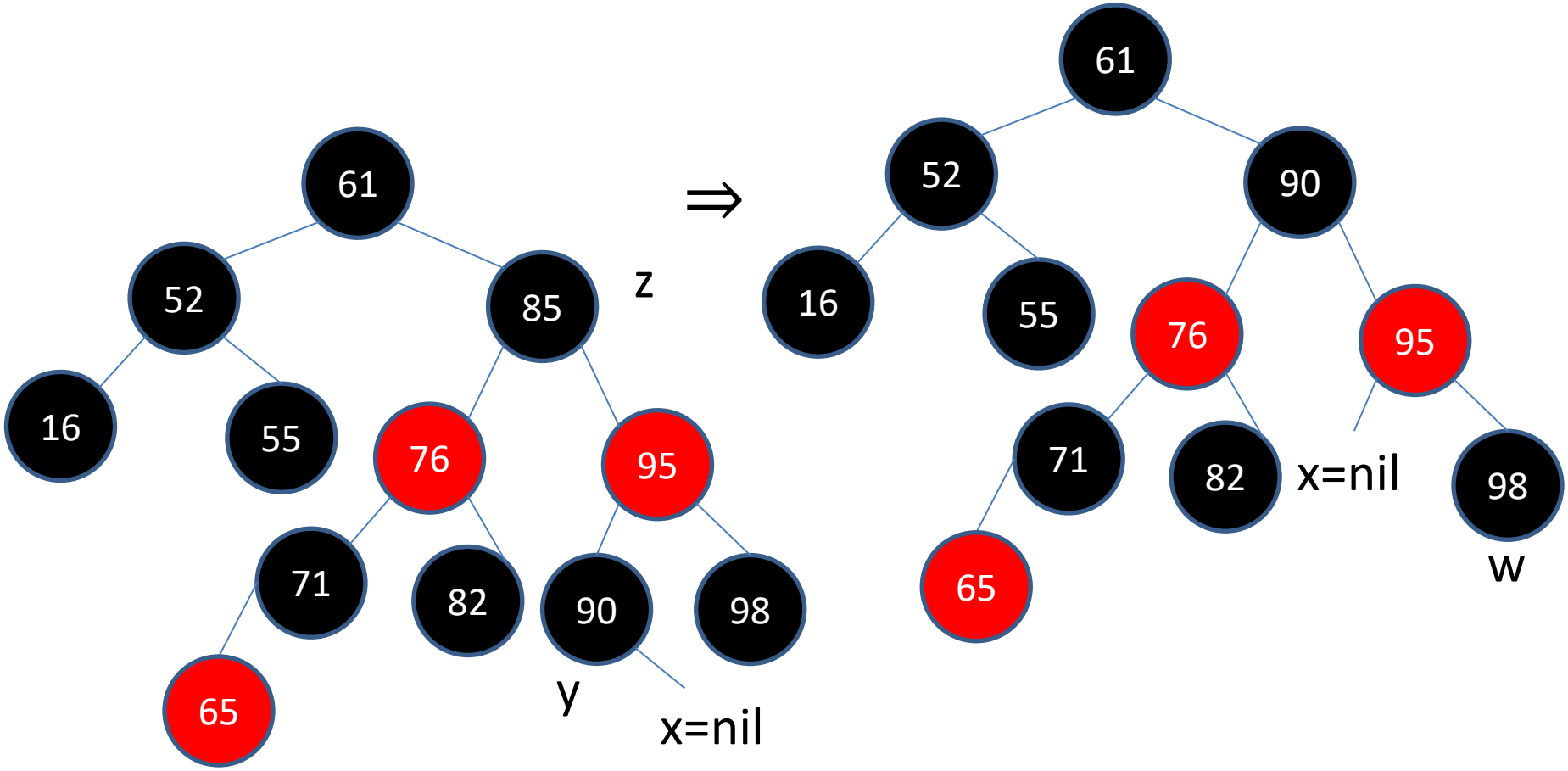
Deletion operation

Deletion of 20



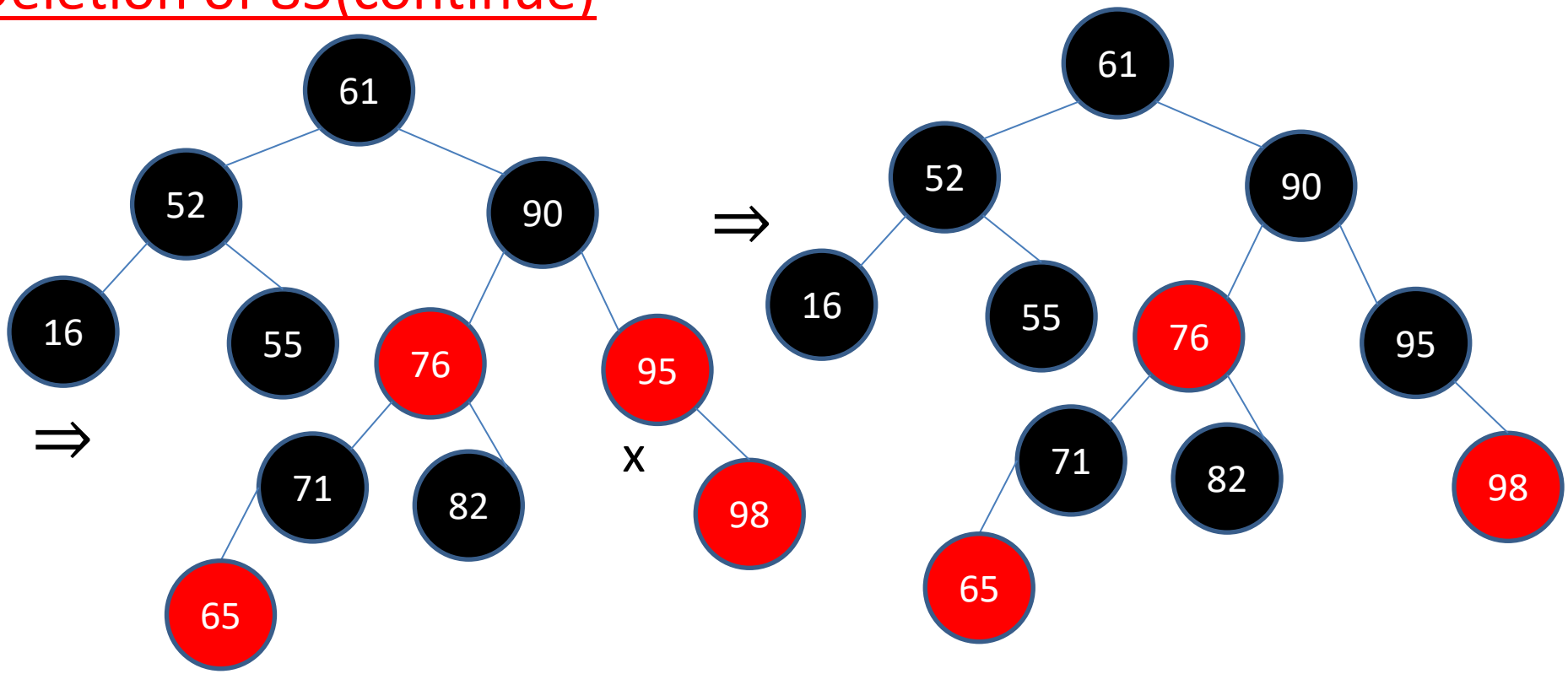
Deletion operation

Deletion of 85



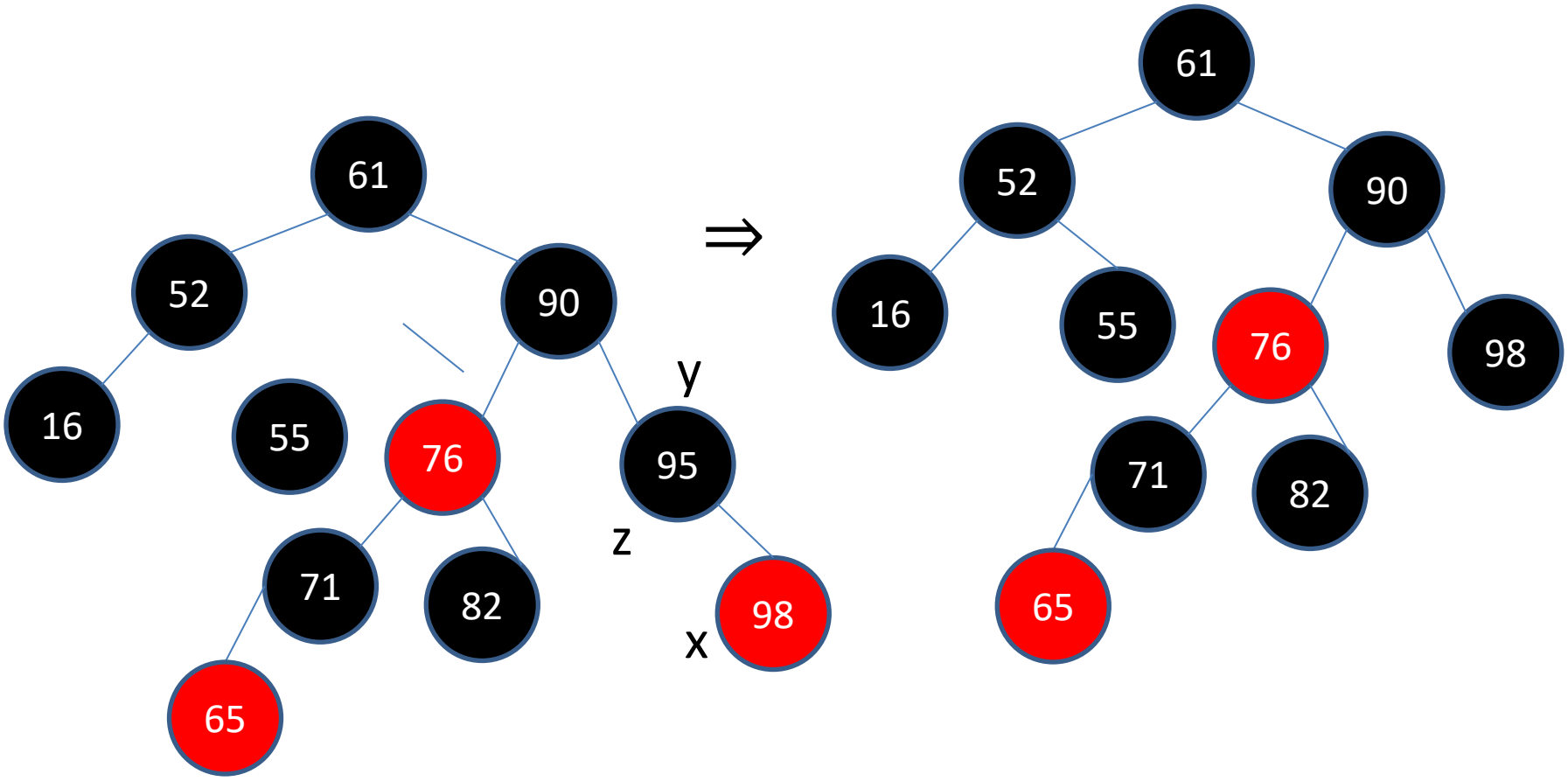
Deletion operation

Deletion of 85(continue)



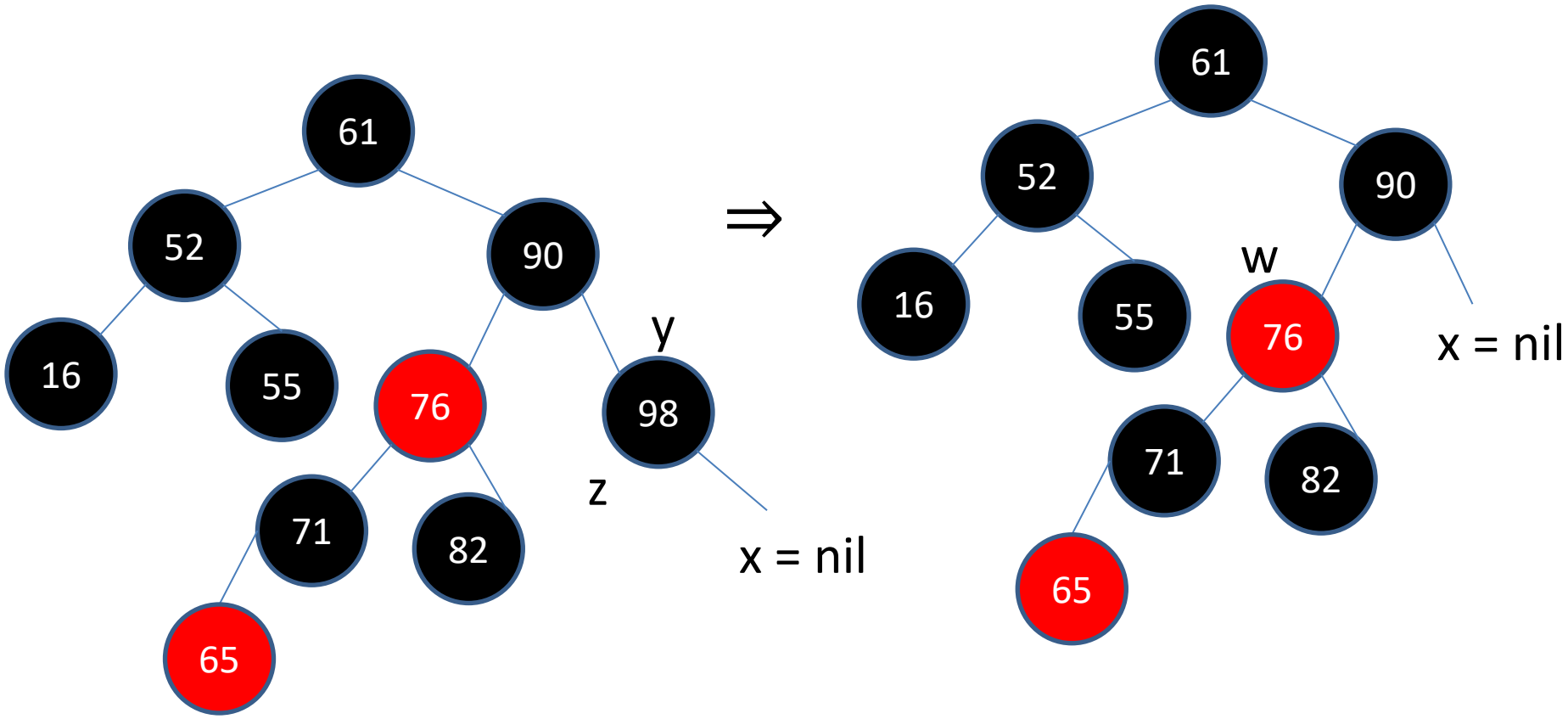
Deletion operation

Deletion of 95



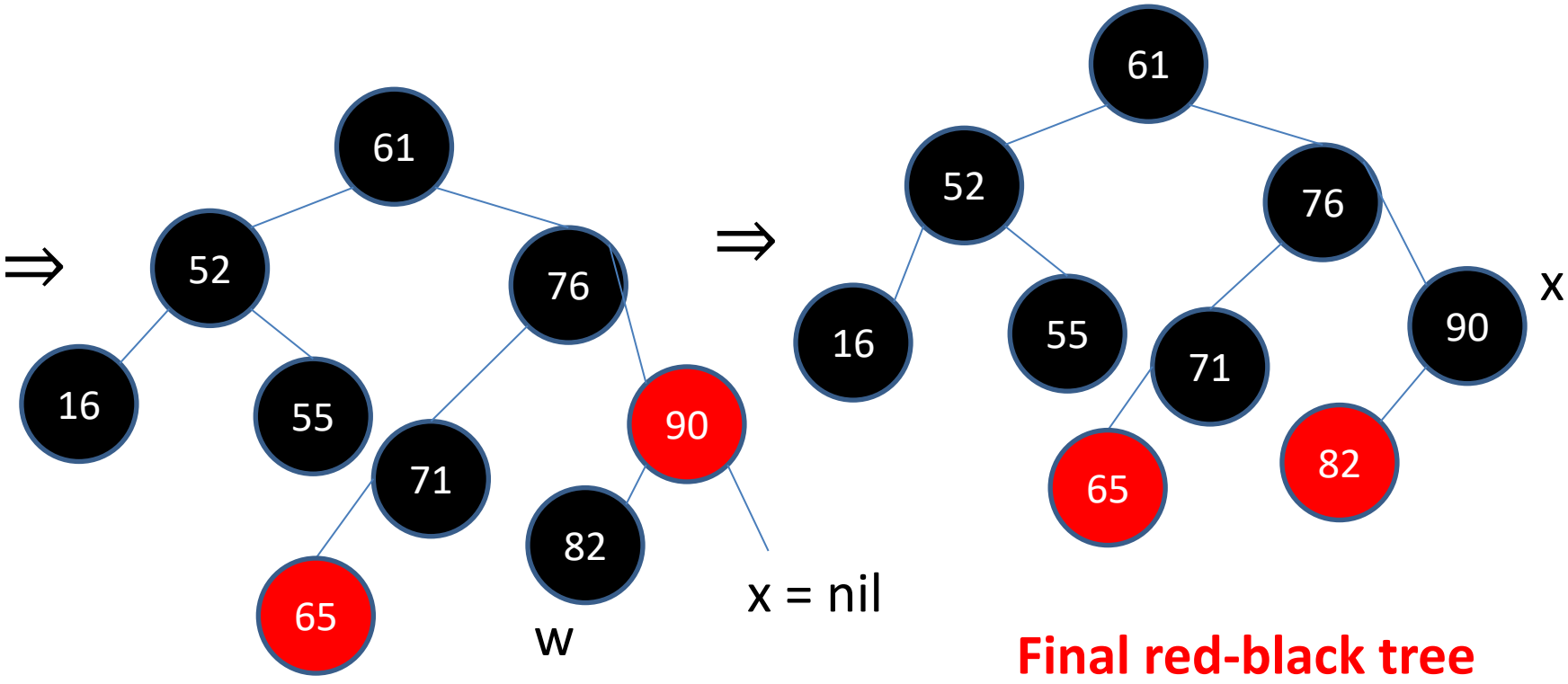
Deletion operation

Deletion of 98



Deletion operation

Deletion of 98(continue)



Final red-black tree

AKTU exam. questions

1. Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black Tree and delete 12, 4, 9, 14 respectively.
2. What is Red-Black tree? Write an algorithm to insert a node in an empty red-black tree explain with suitable example.
3. Insert the following element in an initially empty RB-Tree. 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now Delete 23 and 81.
4. Write the properties of Red-Black Tree. Illustrate with an example, how the keys are inserted in an empty red-black tree.

B-Tree

B-Tree

Definition

A B-tree T is a rooted tree having the following properties:

1. Every node x has the following attributes:

   a. x.n, the number of keys currently stored in node x,

   b. the x.n keys themselves, $x.key_1$, $x.key_2$, ……,$x.key_{x.n}$, stored in non-decreasing order, so that $x.key_1 \le x.key_2 \le \ldots\ldots\ldots \le x.key_{x.\,n}$,

   c. x.leaf , a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains x.n+1 pointers $x.c_1$; $x.c_2$, ……. ,$x.c_{x.\,n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \ldots\ldots\ldots\ldots\ldots\ldots \le X.key_{x.\,n} \le k_{x.\,n+1}$.

B-Tree

4. All leaves have the same depth, which is the tree's height h.

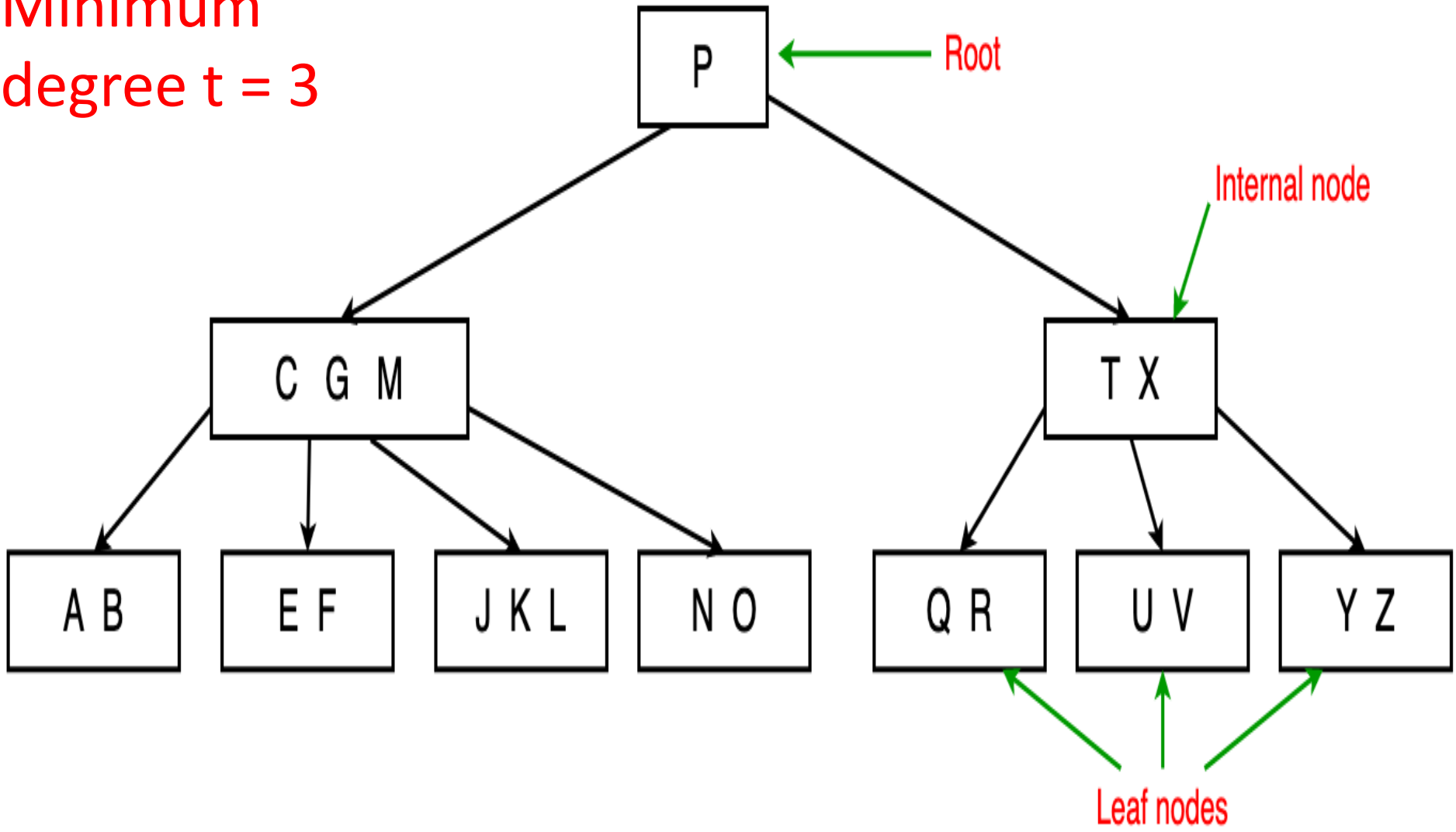
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:

    a. Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

    b. Every node may contain at most 2t -1 keys. Therefore, an internal node may have at most 2t children. We say that a node is full if it contains exactly2t -1 keys.

B-Tree

Minimum degree t = 3



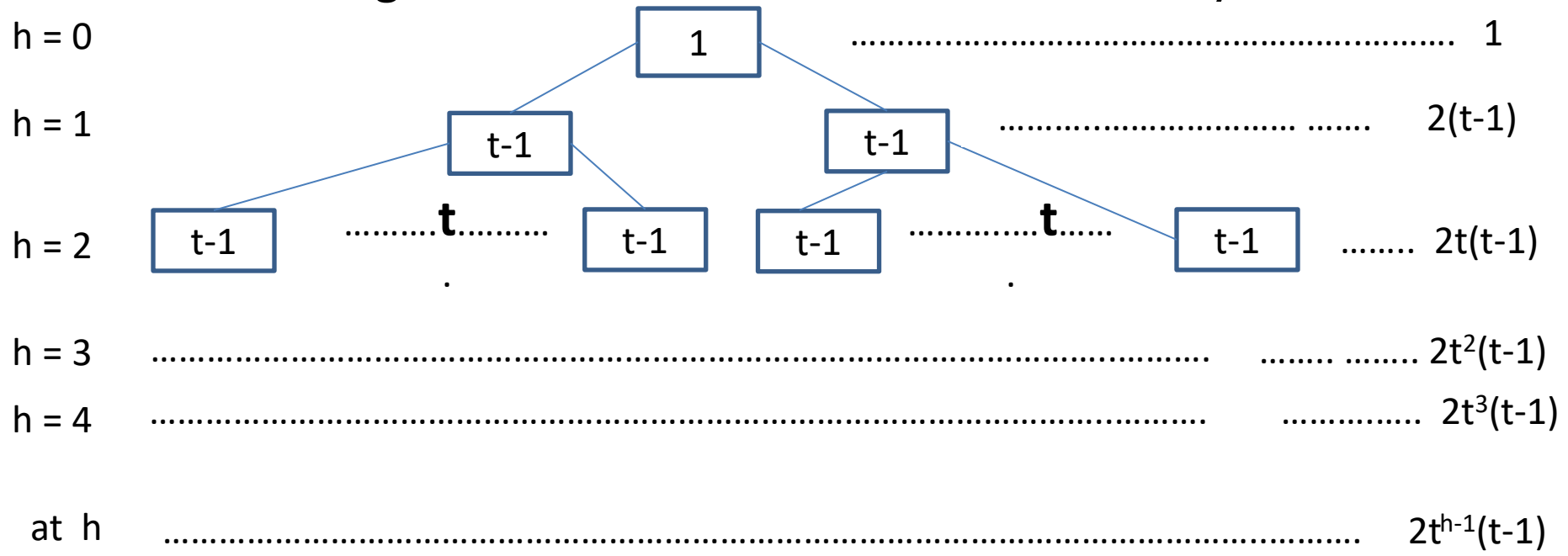
B-Tree

Theorem: If n ≥ 1, then for any n-key B-tree T of height h and minimum degree t ≥ 2,

$$h \leq \log_t \frac{(n+1)}{2}$$

Proof: To prove this, first we will find the minimum number of keys in the B-tree with minimum degree t and height h.

Consider following structure of B-tree with minimum keys:-



B-Tree

Therefore, the minimum number of the keys in B-tree with minimum degree t and height h

$$= 1+ 2(t-1) + 2t(t-1) + 2t^2(t-1) + 2t^3(t-1) + \ldots\ldots + 2t^{h-1}(t-1)$$

$$= 1 + 2(t-1)\left(1 + t + t^2 + t^3 + \ldots\ldots\ldots + t^{h-1} \right)$$

$$= 1 + 2(t-1)\frac{(th-1)}{(t-1)}$$

$$= 1 + 2(t^h-1)$$

$$= 1 + 2t^h-2$$

$$= 2t^h-1$$

Now, since the number of keys in given B-tree is n, therefore

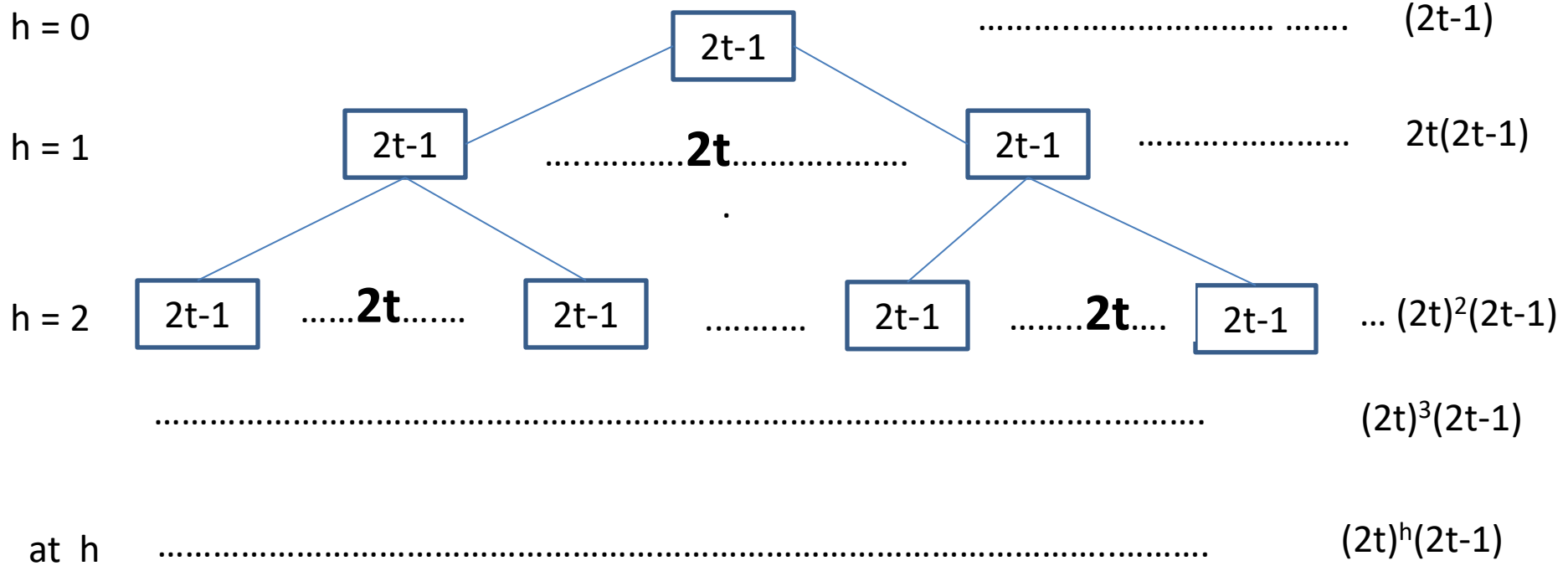
$$2t^h-1 \leq n \Rightarrow t^h \leq \frac{(n+1)}{2} \Rightarrow h \leq \log_t \frac{(n+1)}{2}$$

It is proved.

B-Tree

Question: As a function of the minimum degree t , what is the maximum number of keys that can be stored in a B-tree of height h?

Solution: Consider following structure of B-tree with maximum keys:-



B-Tree

Therefore maximum number of keys in B-tree with minimum degree t and height h

$= (2t-1) + 2t(2t-1) + (2t)^2(2t-1) + (2t)^3(2t-1) + \ldots\ldots\ldots\ldots + (2t)^h(2t-1)$

$= (2t-1)(1+2t + (2t)^2 + (2t)^3 + \ldots\ldots\ldots\ldots\ldots\ldots + (2t)^h)$

$= (2t-1)\frac{((2t)^{h+1}-1)}{(2t-1)}$

$= (2t)^{h+1}-1$

B-Tree

Note:

The simplest B-tree occurs when t = 2. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of t yield B-trees with smaller height.

B-Tree of order m

B-tree of order m is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least ⌈m/2⌉ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains k − 1 keys.
5. All leaves appear in the same level and carry no information.

Each internal node's keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a1 and a2. All values in the leftmost subtree will be less than a1, all values in the middle subtree will be between a1 and a2, and all values in the rightmost subtree will be greater than a2.

Searching operation in B-Tree

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, we make a multiway branching decision according to the number of the node's children.

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3      i = i + 1
4  if i ≤ x.n and k == x.key_i
5      return (x, i)
6  elseif x.leaf
7      return NIL
8  else DISK-READ(x.c_i)
9      return B-TREE-SEARCH(x.c_i, k)
```

Searching operation in B-Tree

- B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree.
- The initial call of this algorithm will be

  B-TREE-SEARCH(T.root, k)
- If k is in the B-tree, B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that $y.key_i = k$. Otherwise, the procedure returns NIL.

B-Tree creation

Example: Create B-tree for the following elements with minimum degree t = 2.

F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E

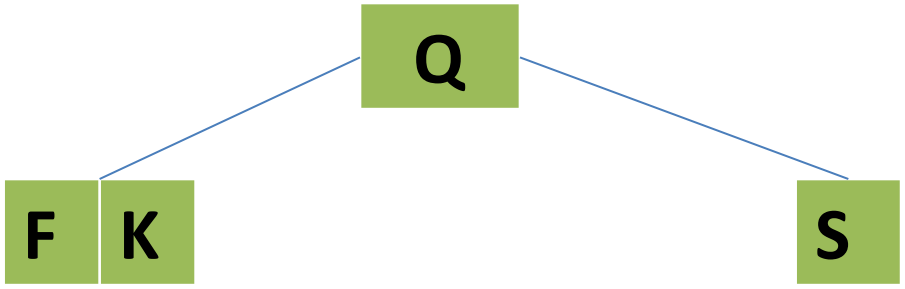
Solution:

Minimum number of keys in a node = t-1 = 1

Maximum number of keys in a node = 2t-1 = 3

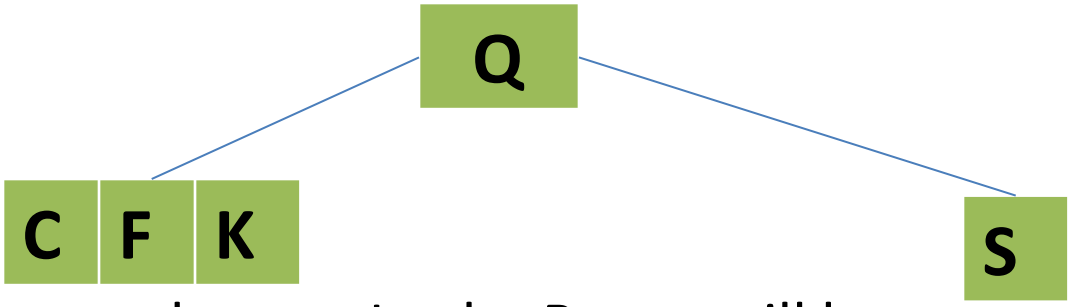
Initial consider 2t-1 elements in the sequence i.e. 3 elements. These are F, S and Q. B-tree for this will be

After inserting next element K, the B-tree will be

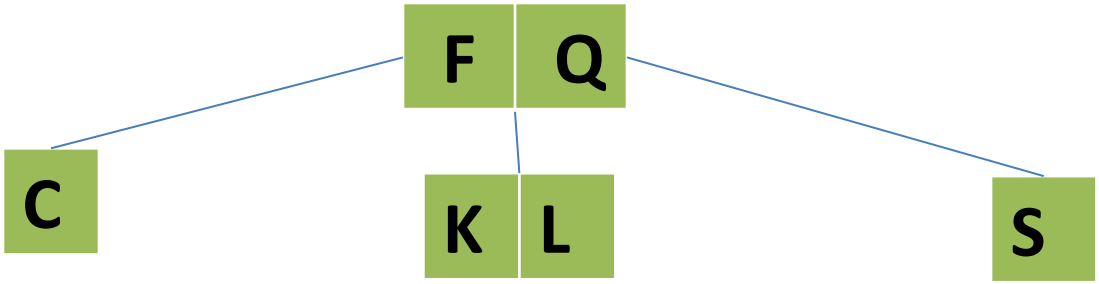


B-Tree

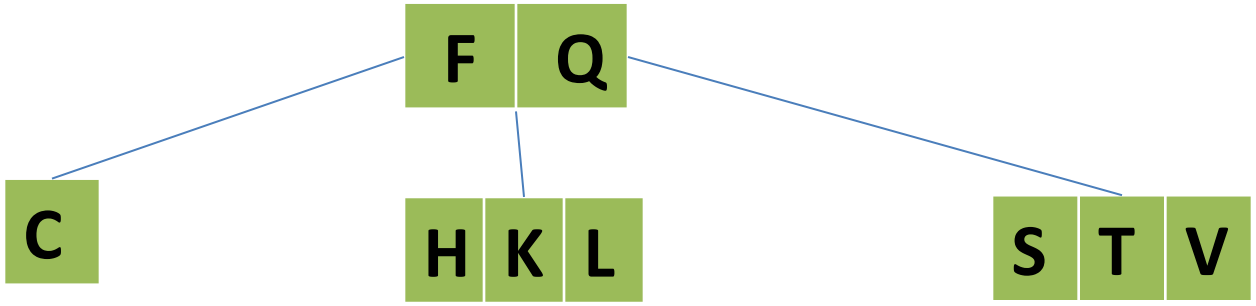
After inserting next element C, the B-tree will be



After inserting next element L , the B-tree will be

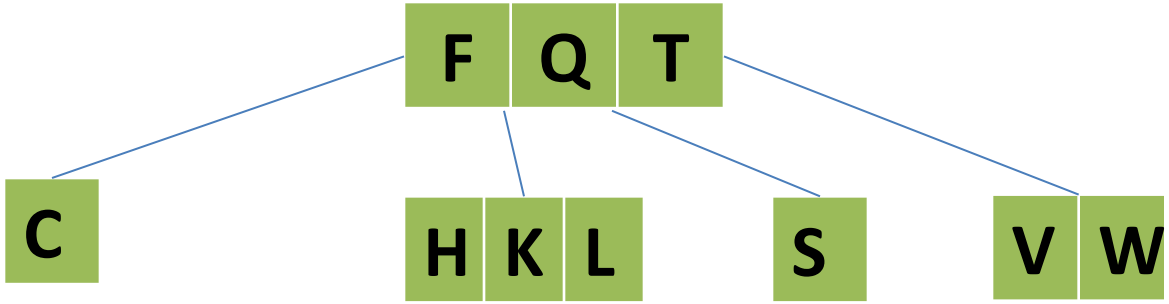


After inserting next element H,T, and V, the B-tree will be

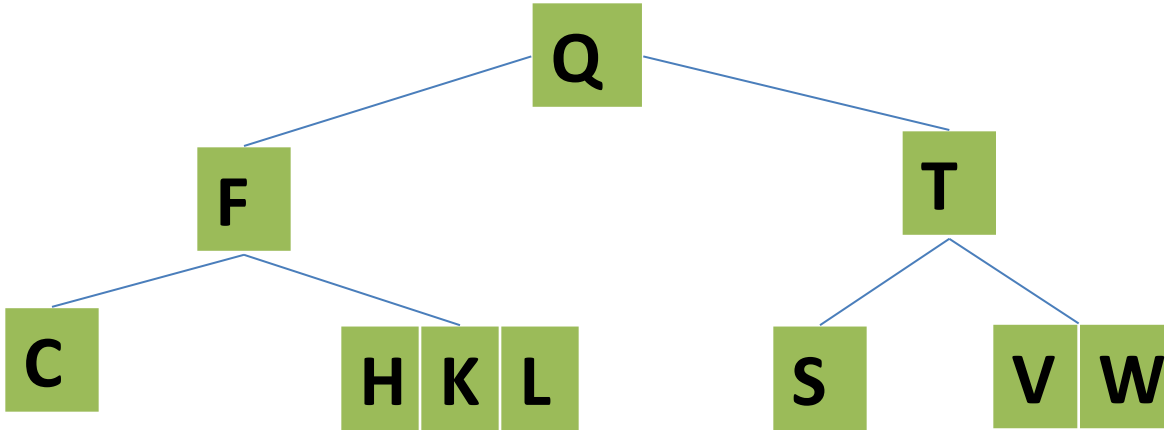


B-Tree

After inserting next element W, the B-tree will be

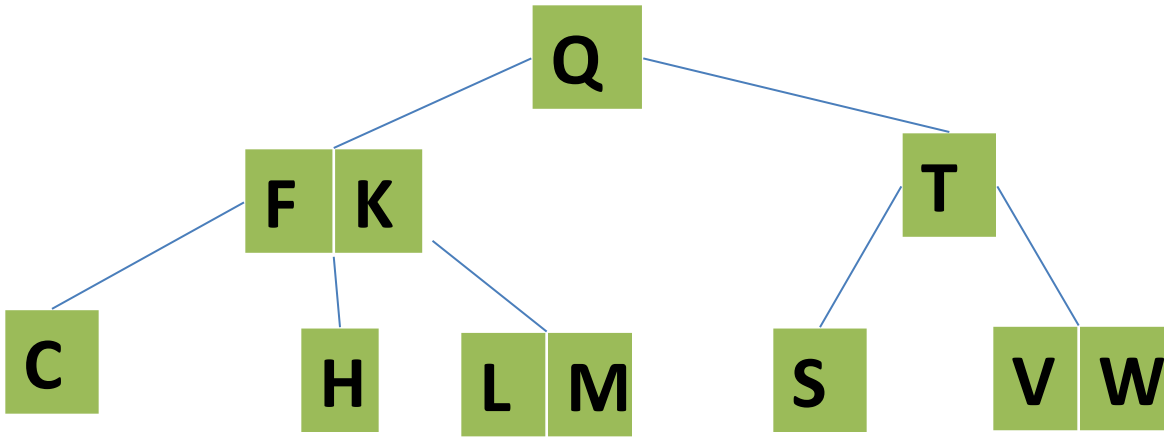


Now, insert element M. Since root node is full, therefore first, we split it.

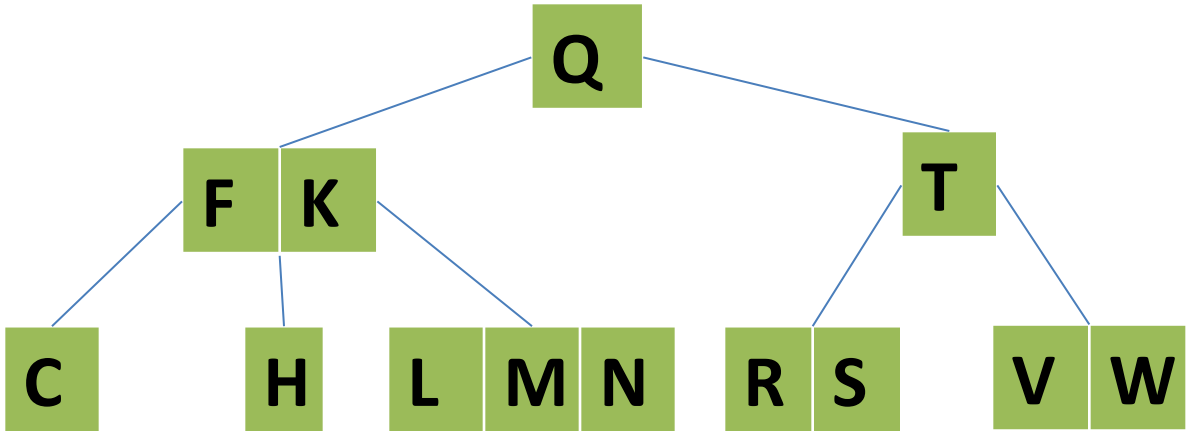


B-Tree

. After splitting and inserting M, B-tree will be

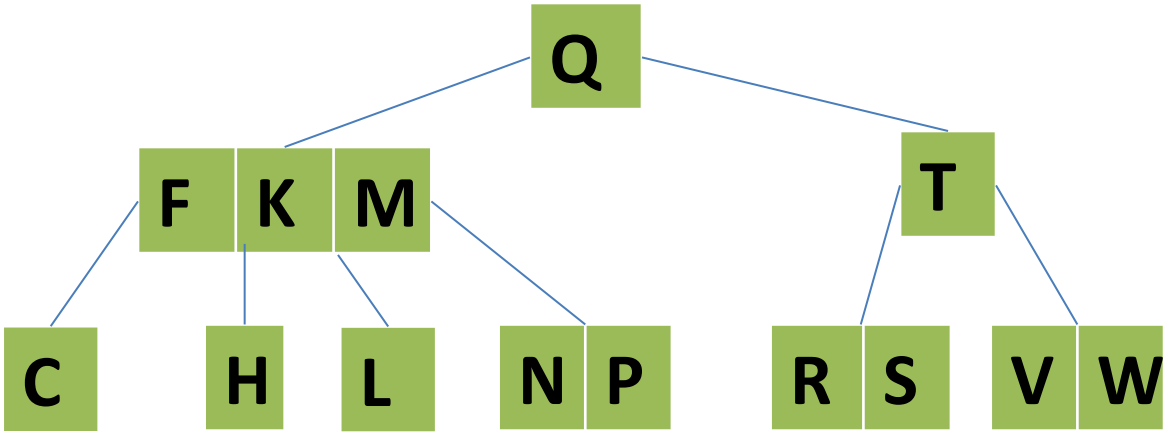


After inserting next element R and N, the B-tree will be

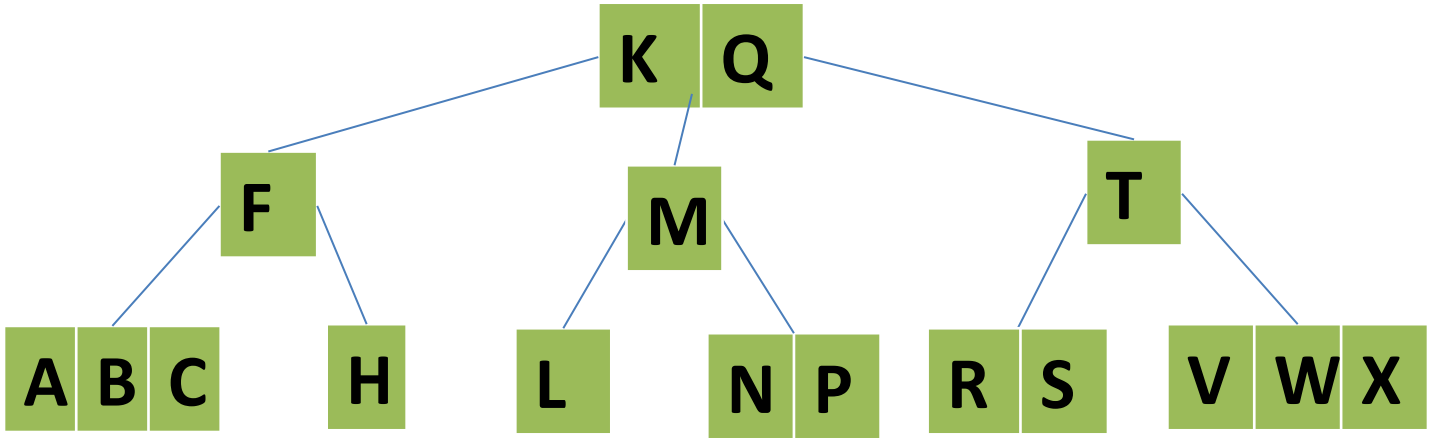


B-Tree

After inserting next element P, the B-tree will be

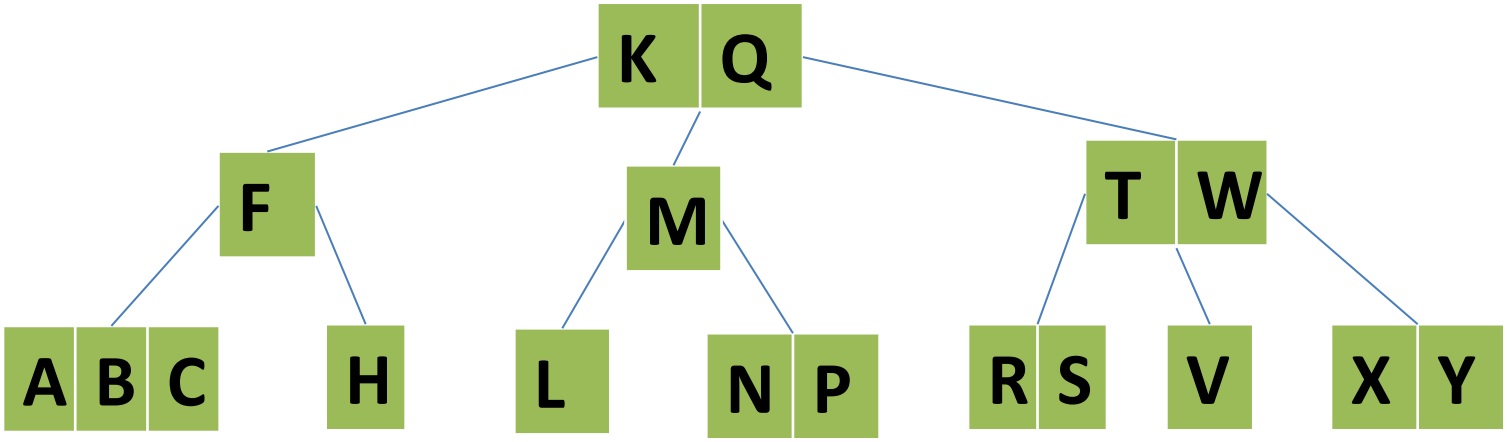


After splitting and inserting next element A, B and X, the B-tree will be

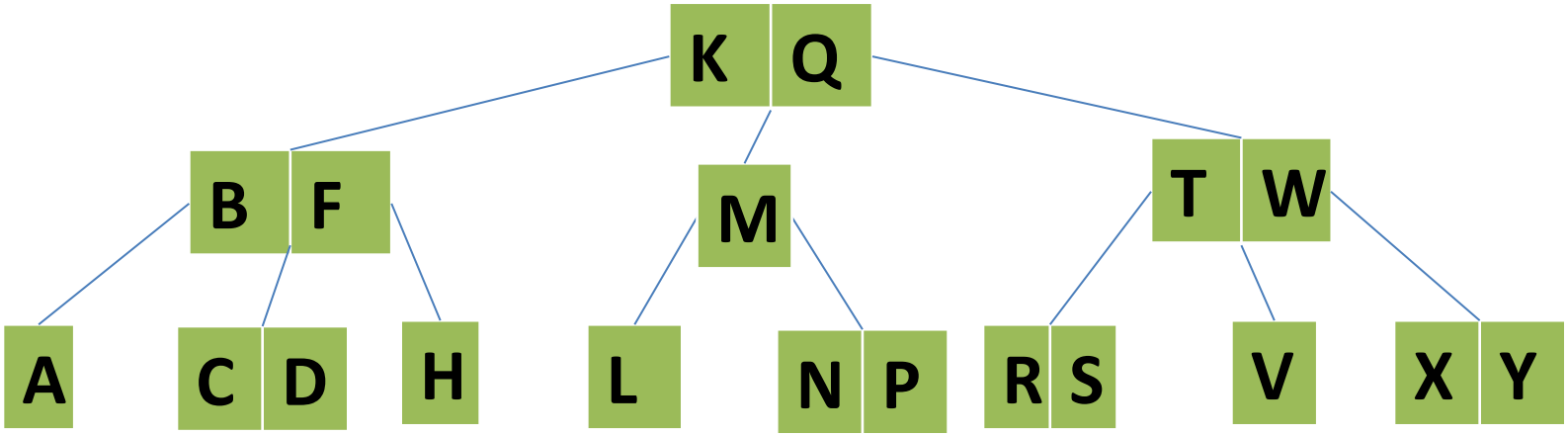


B-Tree

After splitting and inserting next element Y, the B-tree will be

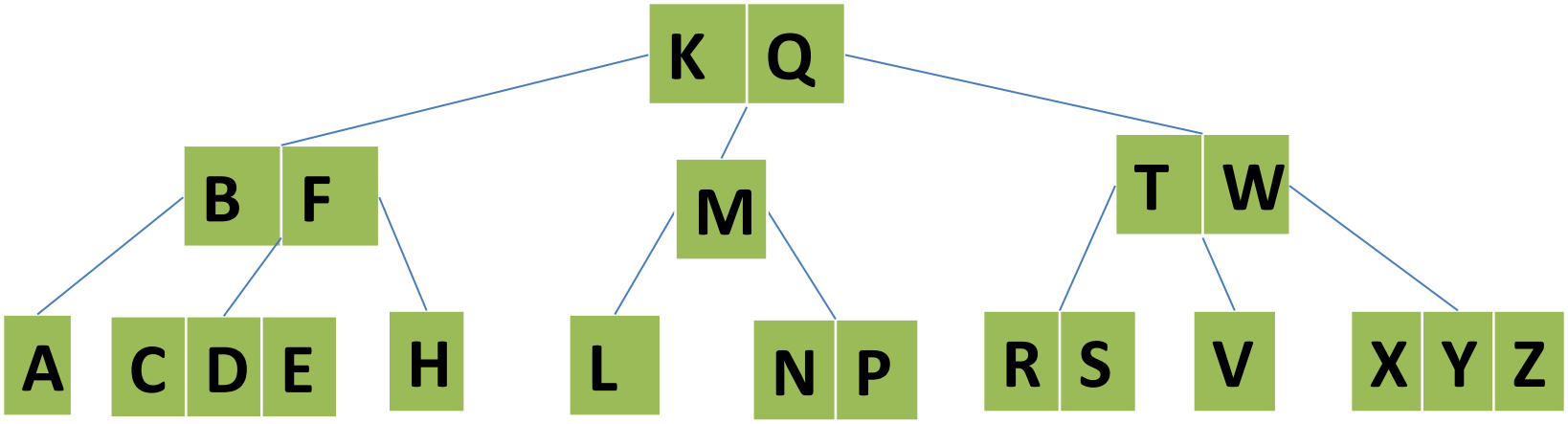


After splitting and inserting next element D, the B-tree will be



B-Tree

After inserting next element Z and E, the B-tree will be



Final B-Tree

B-Tree

Example: Create B-tree for the following elements with minimum degree t = 3.

F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I.

Example: Insert the following keys in a *2-3-4 B Tree*:

40, 35, 22, 90, 12, 45, 58, 78, 67, 60

Example: Using minimum degree ‘t’ as 3, insert following sequence of integers

10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90

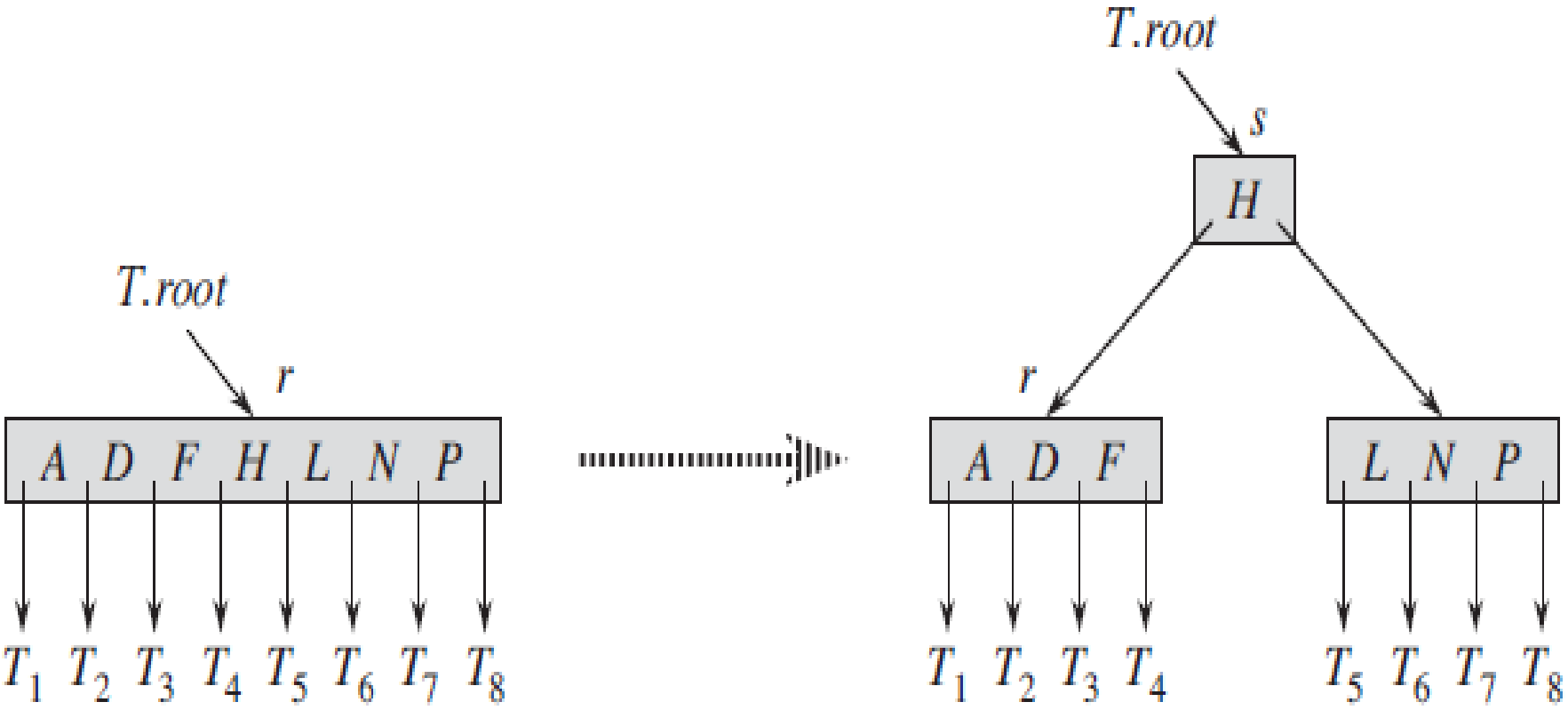
in an initially empty B-Tree. Give the number of nodes splitting operations that take place.

B-Tree Insertion Algorithm

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t − 1
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c_1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

B-Tree Insertion Algorithm

Splitting the root with t = 4. Root node r splits in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

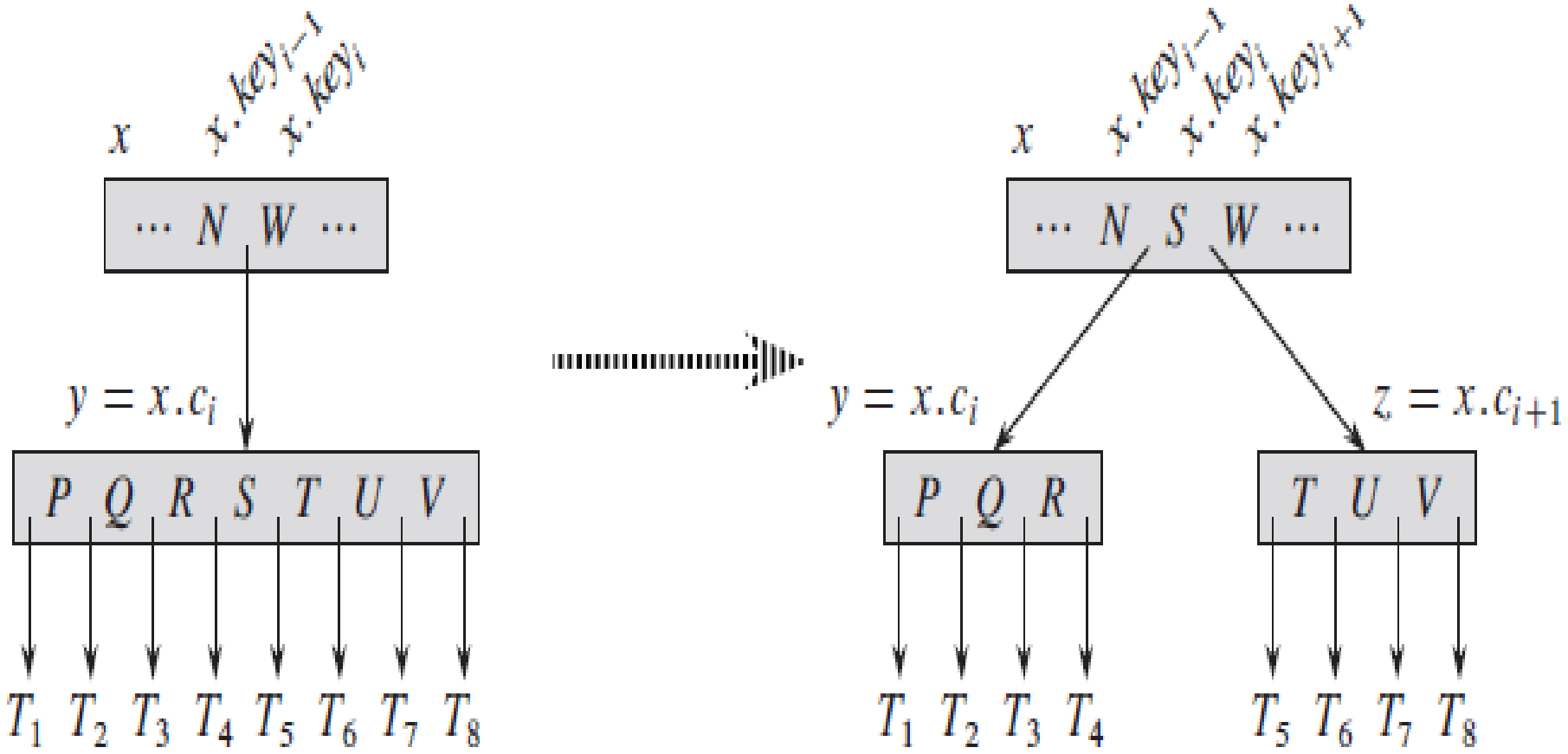


B-Tree Insertion Algorithm

```
B-TREE-SPLIT-CHILD(x, i)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1
 6       z.key_j = y.key_{j+t}
 7  if not y.leaf
 8       for j = 1 to t
 9            z.c_j = y.c_{j+t}
10  y.n = t − 1
11  for j = x.n + 1 downto i + 1
12       x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15       x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

B-Tree Insertion Algorithm

Splitting a node with t = 4. Node y = $x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent.



B-Tree Insertion Algorithm

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3       while i ≥ 1 and k < x.key_i
 4            x.key_{i+1} = x.key_i
 5            i = i − 1
 6       x.key_{i+1} = k
 7       x.n = x.n + 1
 8       DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10            i = i − 1
11       i = i + 1
12       DISK-READ(x.c_i)
13       if x.c_i.n == 2t − 1
14            B-TREE-SPLIT-CHILD(x, i)
15            if k > x.key_i
16                 i = i + 1
17       B-TREE-INSERT-NONFULL(x.c_i, k)
```

Time complexity of insertion algorithm

= O(th)

= $O(t \log_t n)$

Deletion operation

Procedure: Suppose we want to delete key k from the B-tree with minimum degree t. Then we use following steps for this purpose:-

1. If the key k is in node x and x is a leaf, then delete the key k from x.

2. If the key k is in node x and x is an internal node, then we do the following:

    2a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x.

Deletion operation

2b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k’ of k in the subtree rooted at z. Recursively delete k’ , and replace k by k’ in x.

2c. Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

Deletion operation

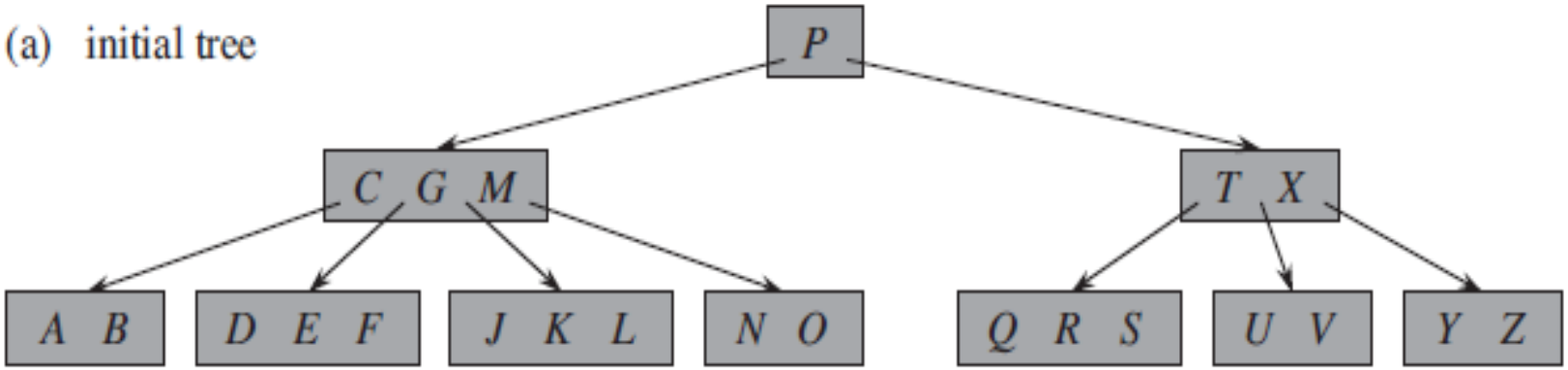
3. If the key k is not present in internal node x, then determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

3a. If $x.c_i$ has only t-1 keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.

3b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have t-1 keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Deletion operation

Example: Consider the following B-tree with minimum degree t = 3.



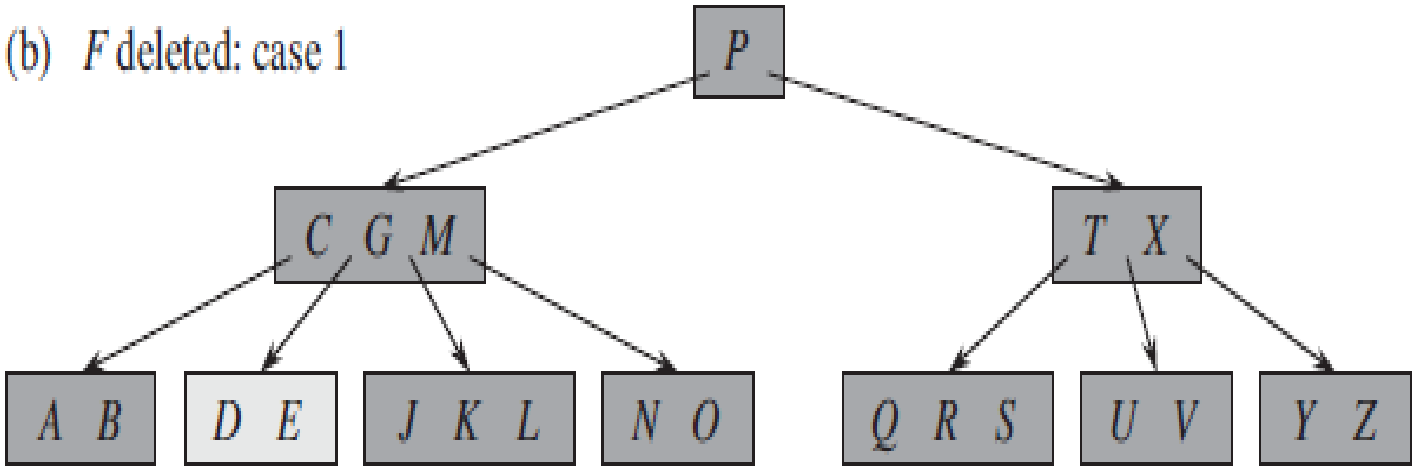
Delete the following elements from this B-tree in-order.

F, M, G, D and B.

Deletion operation

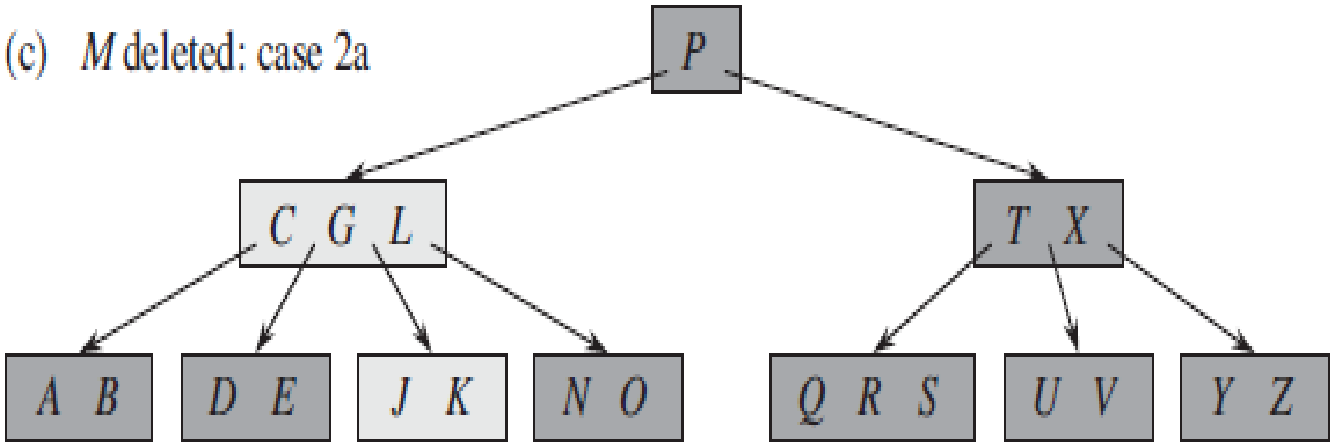
Solution:

Deletion of F



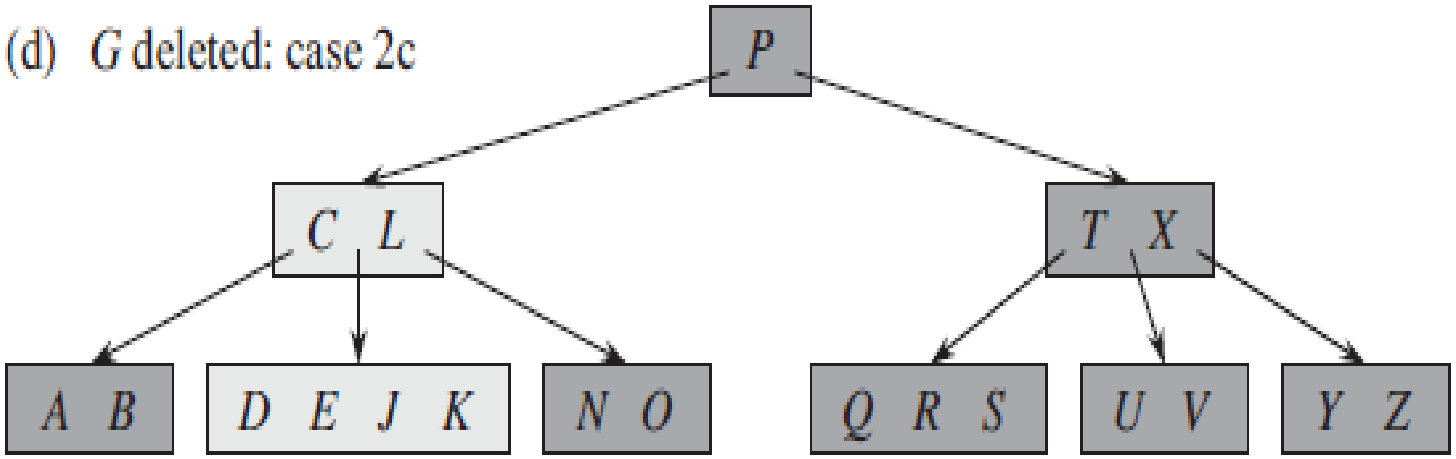
Deletion operation

Deletion of M



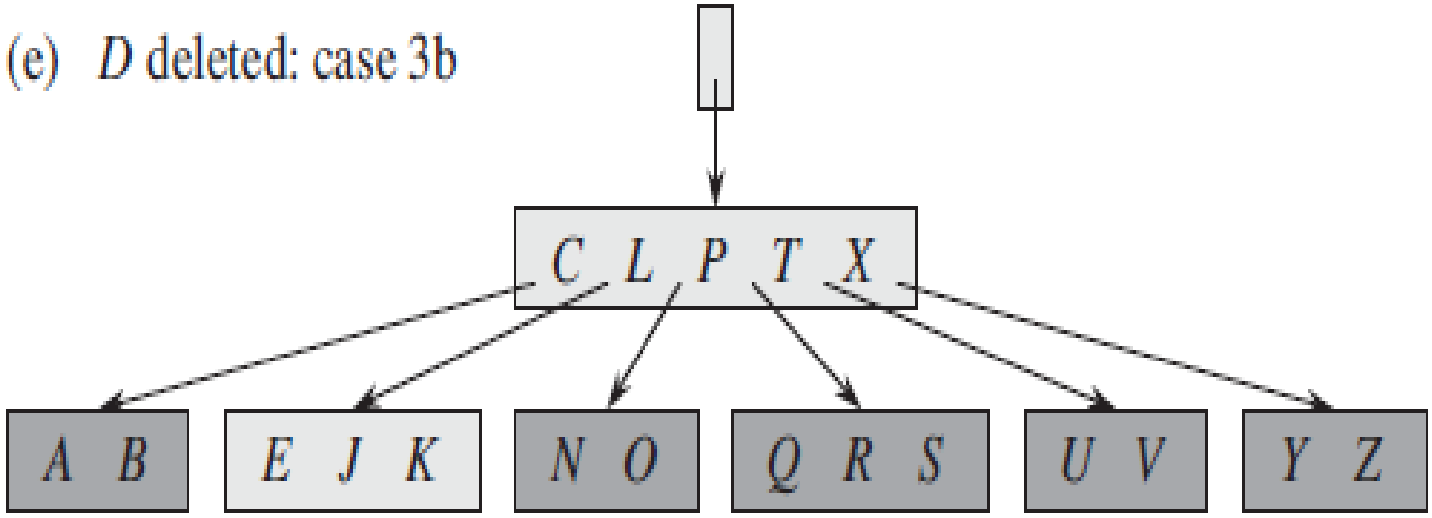
Deletion operation

Deletion of G



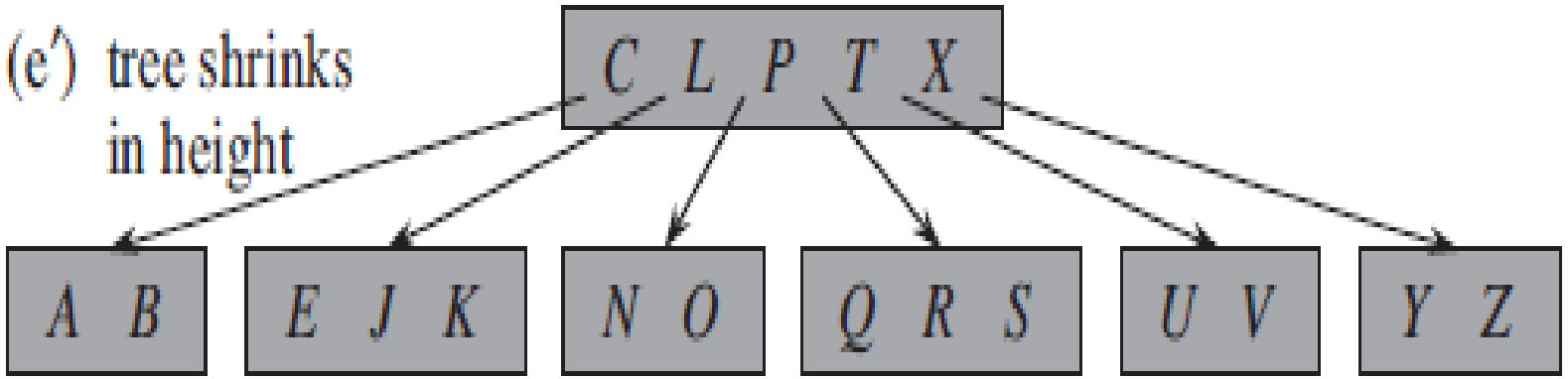
Deletion operation

Deletion of D



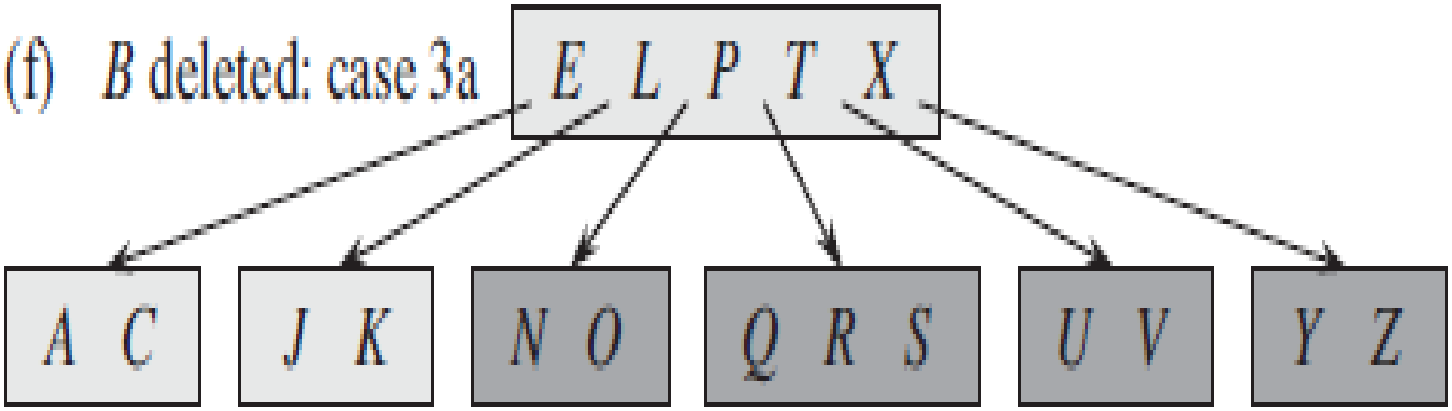
Deletion operation

Deletion of D



Deletion operation

Deletion of B



Final B-tree

Time complexity of deletion operation is O(th) i.e.

$O(th) = O(t \log_t n)$

AKTU Questions

1. Discuss the advantages of using B-Tree. Insert the following Information 86, 23, 91, 4, 67, 18, 32, 54, 46, 96, 45 into an empty B-Tree with degree t = 2 and delete 18, 23 from it.
2. Define a B-Tree of order m. Explain the searching operation in a B-Tree.
3. Using minimum degree ‘t’ as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.
4. Insert the following information F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I. Into an empty B-tree with degree t=3.
5. Prove that if n>=1, then for any n-key B-Tree of height h and minimum degree t >=2, h<=log t ((n +1)/2).
6. Insert the following keys in a 2-3-4 B Tree: 40, 35, 22, 90, 12, 45, 58, 78, 67, 60 and then delete key 35 and 22 one after other.

Binomial Heap

Binomial Tree

Definition

The binomial tree B_k is an ordered tree defined recursively.

1. The binomial tree B_0 consists of a single node.
2. The binomial tree B_k consists of two binomial trees B_{k-1} i.e.

$$B_k = B_{k-1} + B_{k-1}$$

They are linked together in the following way:

The root of one is the leftmost child of the root of the other.

Example: Some binomial trees are the following:-

B0:

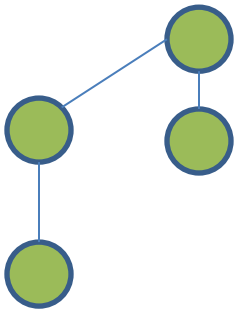
Binomial Tree

Example: Some binomial trees are the following:-

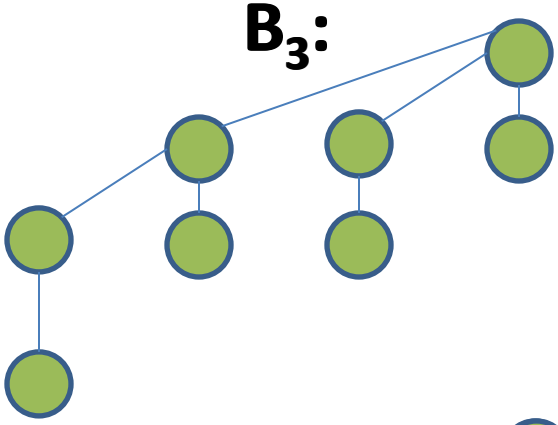
B_0:

B_1:

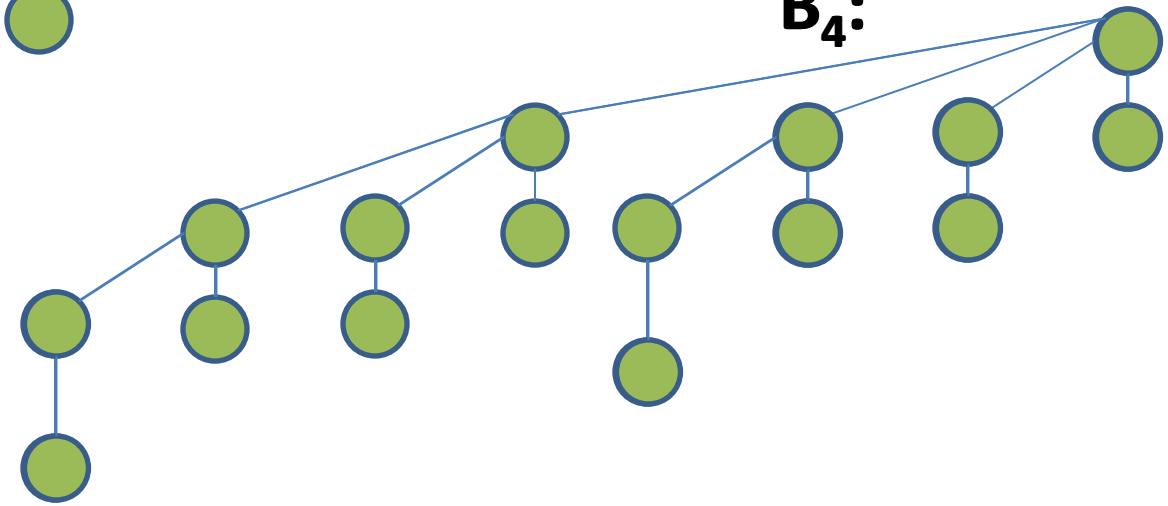
B_2:



B_3:



B_4:



Properties of binomial trees

For the binomial tree B_k,

1. There are 2^k nodes.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ nodes at depth i for i = 0, 1, ..., k.
4. The root has degree k, which is greater than that of any other node. Moreover if the children of the root are numbered from left to right by k−1, k−2, ..., 2, 1, 0, then child i is the root of a subtree B_i.

Properties of binomial trees(cont.)

Proof: The proof is by induction on k. For each property, the basis is the binomial tree B_0. Verifying that each property holds for B_0 is trivial. For the inductive step, we assume that all the properties holds for B_{k-1}.

1. Since binomial tree B_k consists of two copies of binomial tree B_{k-1}, therefore

   Number of nodes in $B_k = 2^{k-1} + 2^{k-1} = 2.2^{k-1} = 2^k$

2. Since in B_k, one B_{k-1} is child of the other B^{k-1}, therefore

   the height of B_k = the height of B_{k-1} +1

   $= (k-1) + 1$

   $= k$

Properties of binomial trees(cont.)

3.Let D(k, i) be the number of nodes at depth i of binomial tree B_k. Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in B_{k-1} appears in B_k once at depth i and i in B_k is the number of nodes at depth i in B_{k-1} plus the number of nodes at depth i−1 in B_{k-1}. Thus,

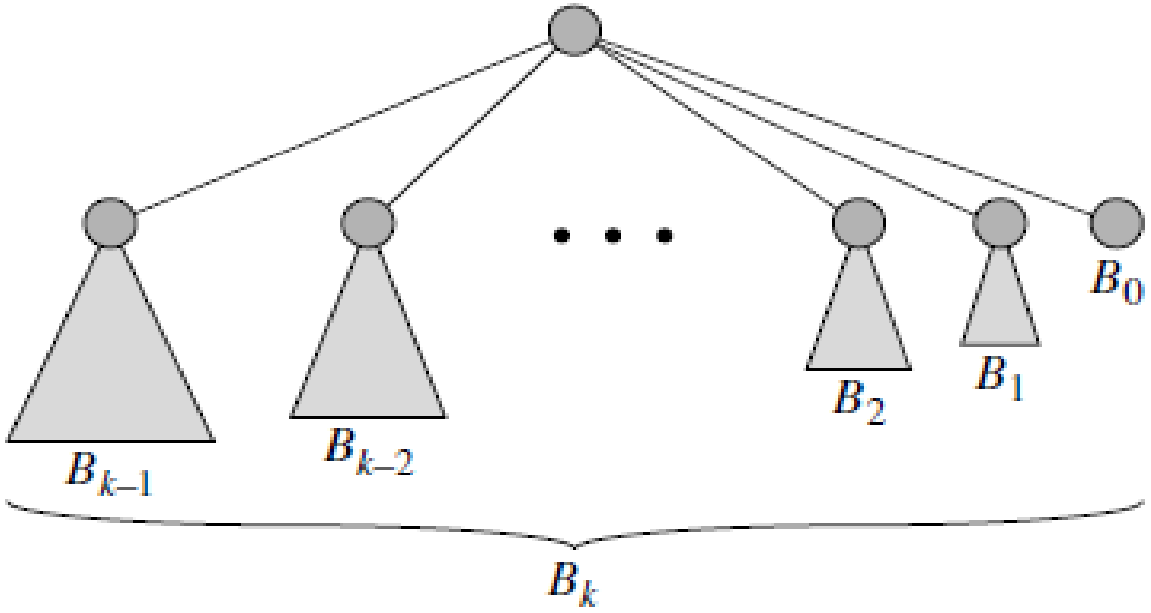
D(k,i) = D(k-1,i) + D(k-1, i-1)

$$= \binom{k-1}{i} + \binom{k-1}{i-1}$$

$$= \binom{k}{i}$$

Properties of binomial trees(cont.)

4. The only node with greater degree in B_k than in B_{k-1} is the root, which has one more child than in B_{k-1}. Since the root of B_{k-1} has degree k−1, therefore the root of B_k has degree k.



Now, by the inductive hypothesis, and as figure shows, from left to right, the children of the root of B_{k-1} are roots of B_{k-2}, B_{k-3}, ..., B_0. When B_{k-1} is linked to B_{k-1}, therefore, the children of the resulting root are roots of B_{k-1}, B_{k-2}, ..., B_0.

Binomial Tree

Lemma: The maximum degree of any node in an *n*-node binomial tree is lg *n*.

Proof: Let the maximum degree of any node is k.

According to property (4), the root node has a maximum degree . Therefore degree of root node is k. This imply that the binomial tree will be B_k.

According to property (1), the total number of nodes in binomial tree B_k is 2^k. Since the number of nodes in binomial tree is n, therefore

$$2^k = n \quad \Rightarrow \quad k = \lg n$$

It is proved.

Binomial Heap

Definition: A binomial heap H is a set of binomial trees that satisfies the following binomial heap properties.

1. Each binomial tree in H obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. We say that each such tree is min-heap-ordered.

2. For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.

Note: An n-node binomial heap H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

Example: Construct binomial heap for 27 nodes.

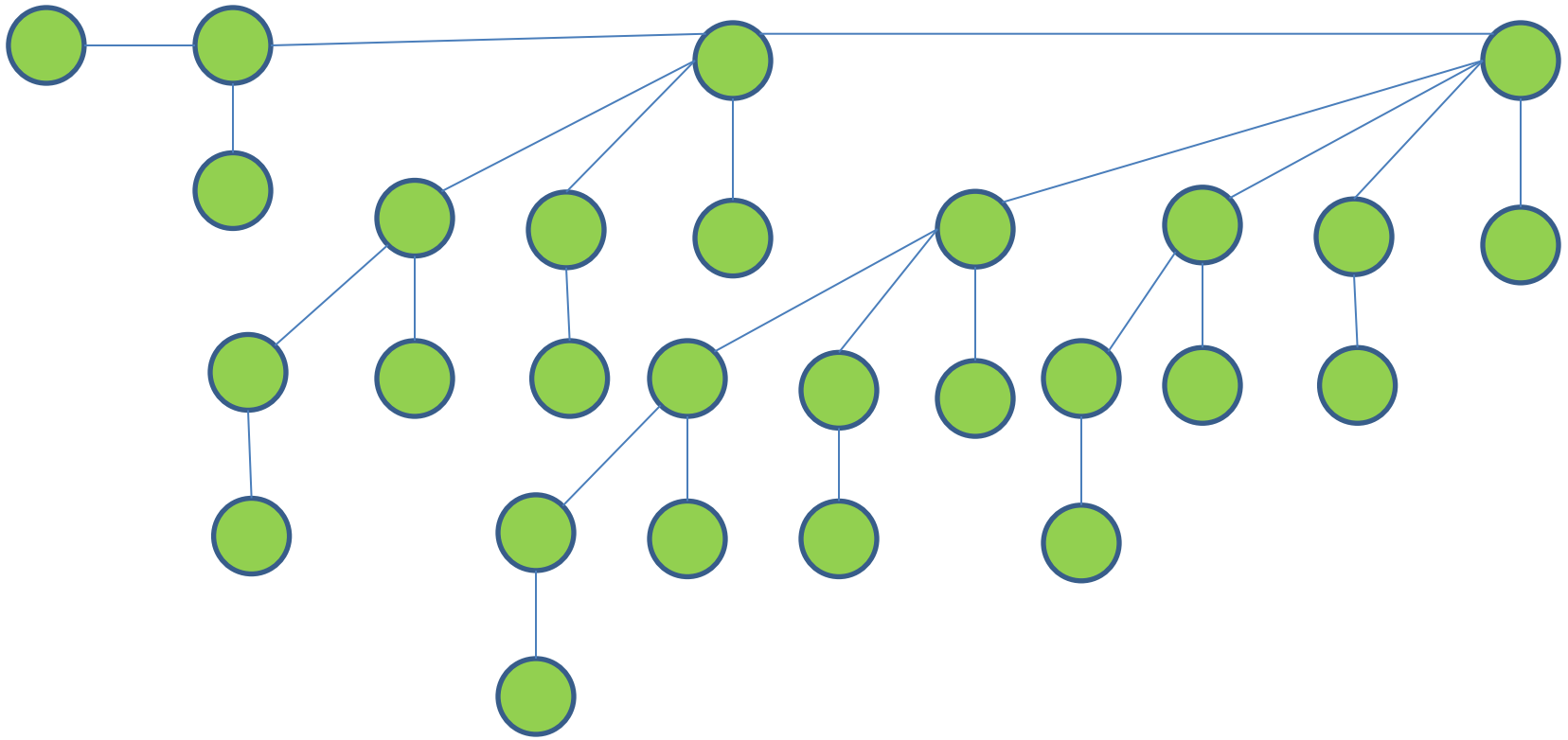
Solution: First we find binary number of 27. After it, we compare this number with $B_4B_3B_2B_1B_0$. If the corresponding binary number is 1, then we use the corresponding binomial tree in the Binomial heap.

Binary number of 27 = 11011

Therefore, binomial tree in binomial heap will be B_4, B_3, B_1, B_0.

Binomial heaps

Therefore, binomial heap for 27 nodes will be



Representation of binomial heaps

- Each binomial tree within a binomial heap is stored in the left-child, right-sibling representation.
- Each node x in binomial heap consists of following fields:-

Key[x] → value stored in the node x

p[x] → pointer representing parent of node x

child[x] → pointer representing left most child of node x

sibling[x] → pointer representing immediate right sibling of node x

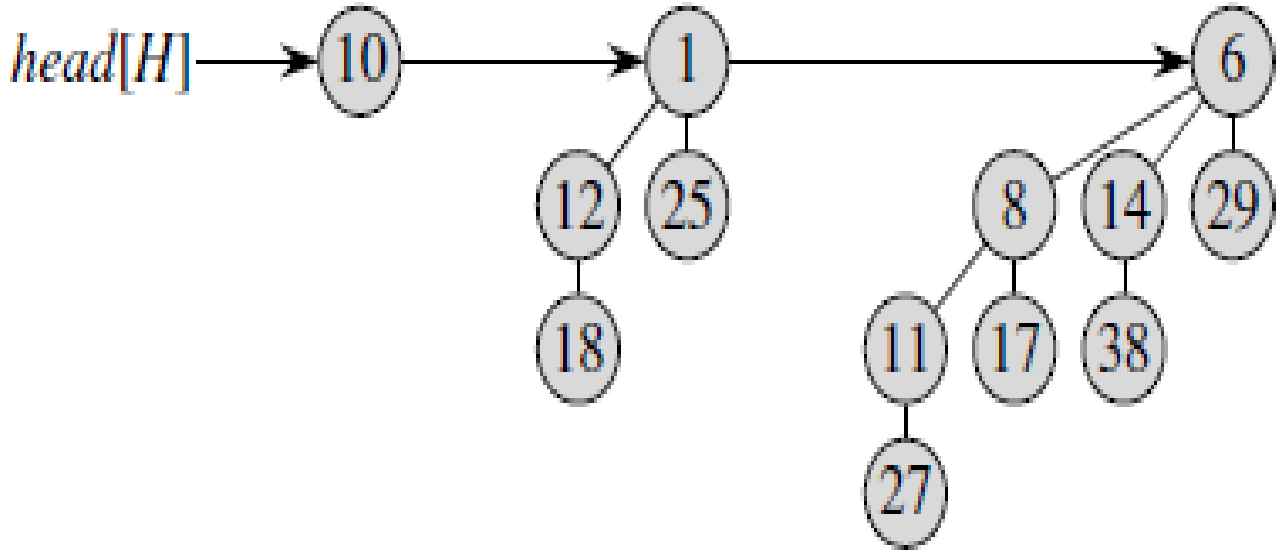
degree[x] → the number of children of node *x*

Representation of binomial heaps

- The roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the root list. The degrees of the roots strictly increase as we traverse the root list.
- The sibling field has a different meaning for roots than for non roots. If x is a root, then sibling[x] points to the next root in the root list.
- A given binomial heap H is accessed by the field head[H], which is simply a pointer to the first root in the root list of H. If binomial heap H has no elements, then head[H] = NIL.

Representation of binomial heaps

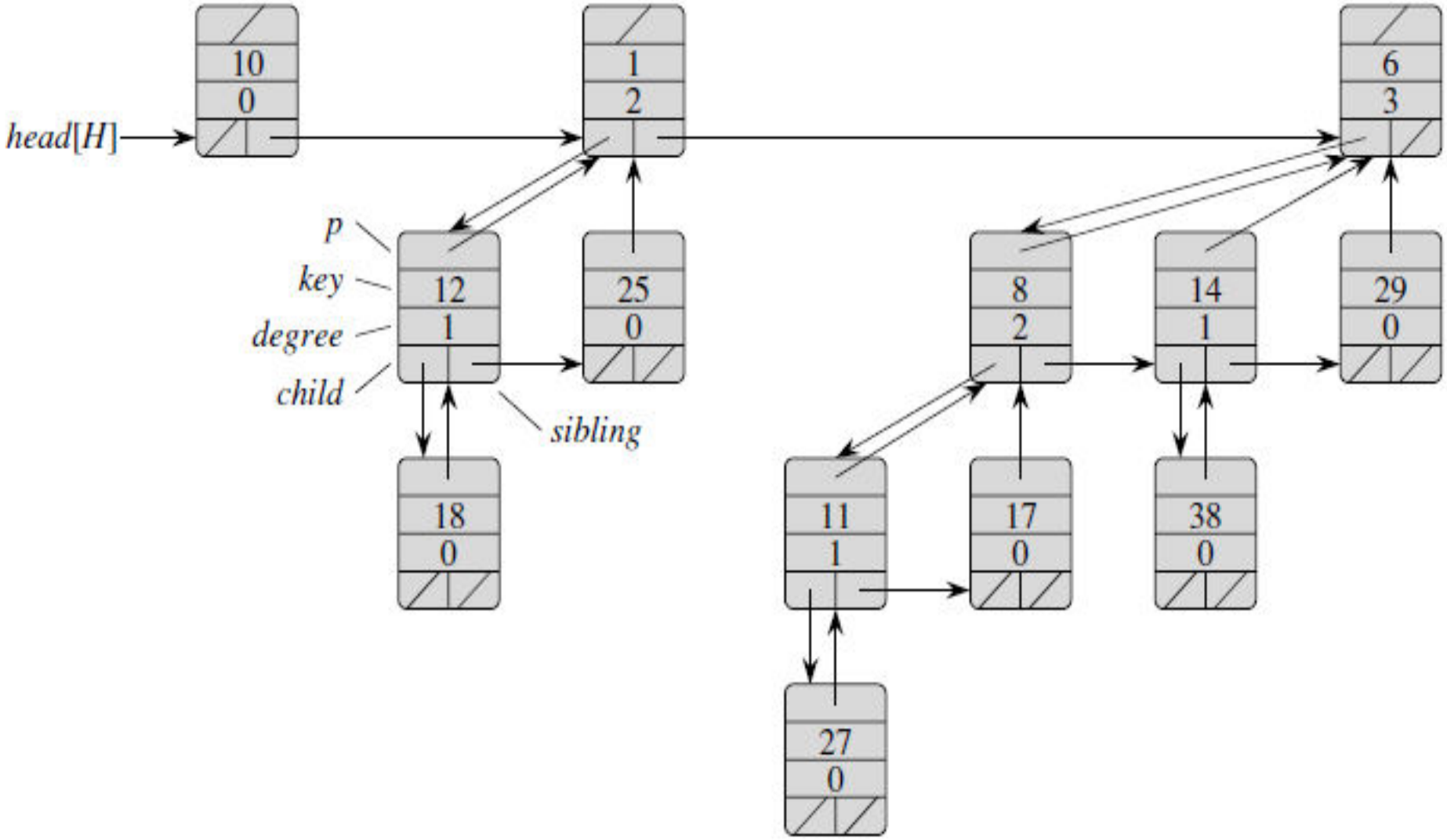
Example: Consider the following binomial heap:-



Find the representation of this binomial heap.

Representation of binomial heaps

Representation of binomial heap is



Operations defined on binomial heaps

Finding the minimum key

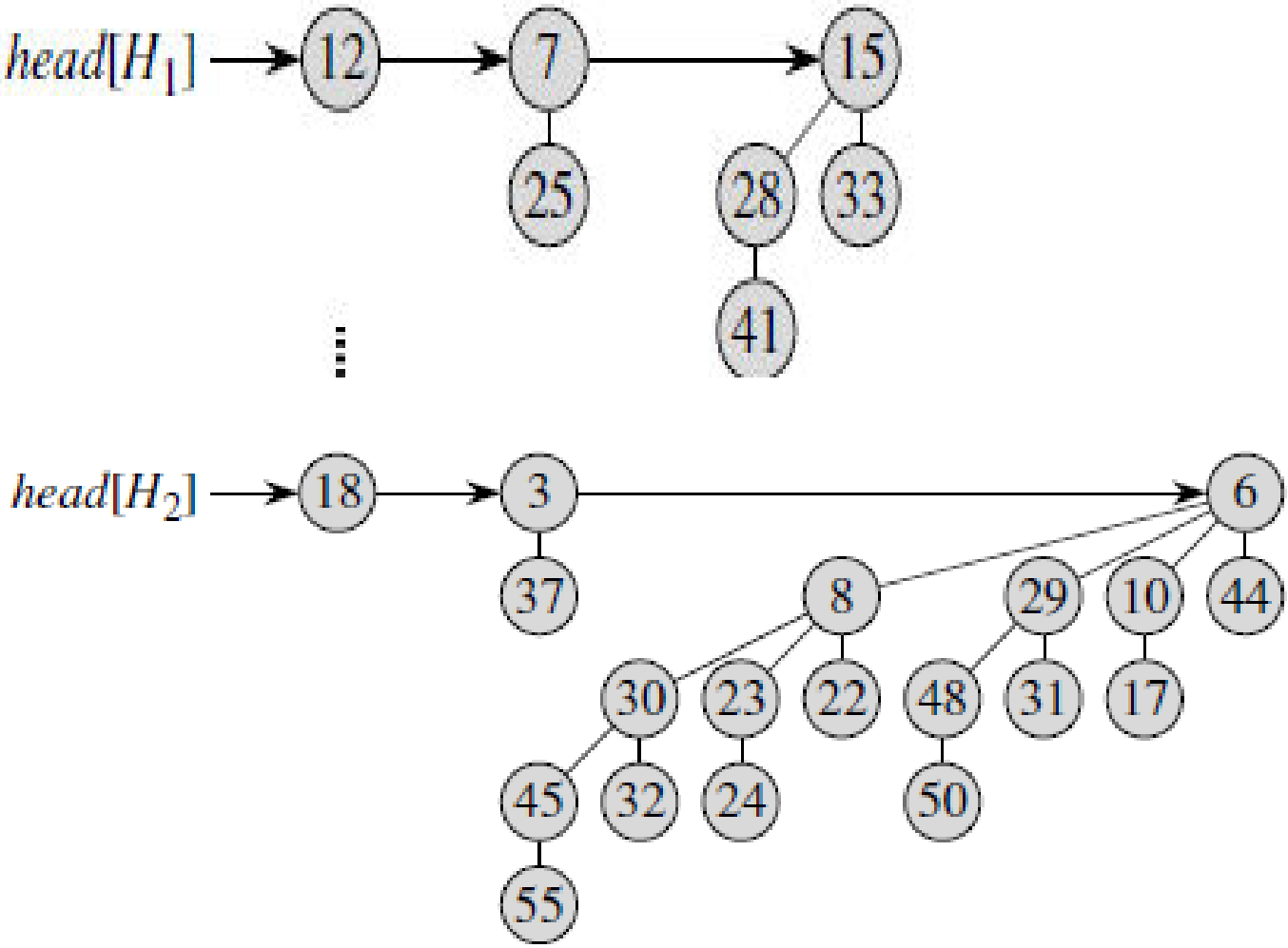
The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H.

```
BINOMIAL-HEAP-MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5        do if key[x] < min
6              then min ← key[x]
7                   y ← x
8           x ← sibling[x]
9  return y
```

Note: The running time of BINOMIAL-HEAP-MINIMUM is O(lg n).

Union of two binomial heaps

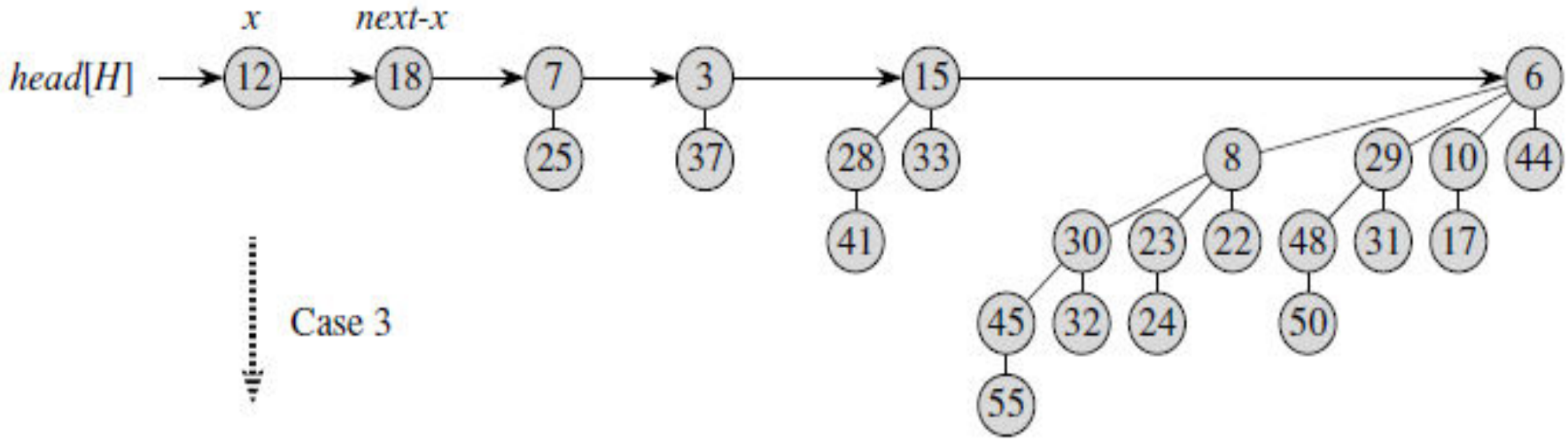
Example: Consider following two binomial heaps H_1 and H_2. Find the union of these binomial heaps.



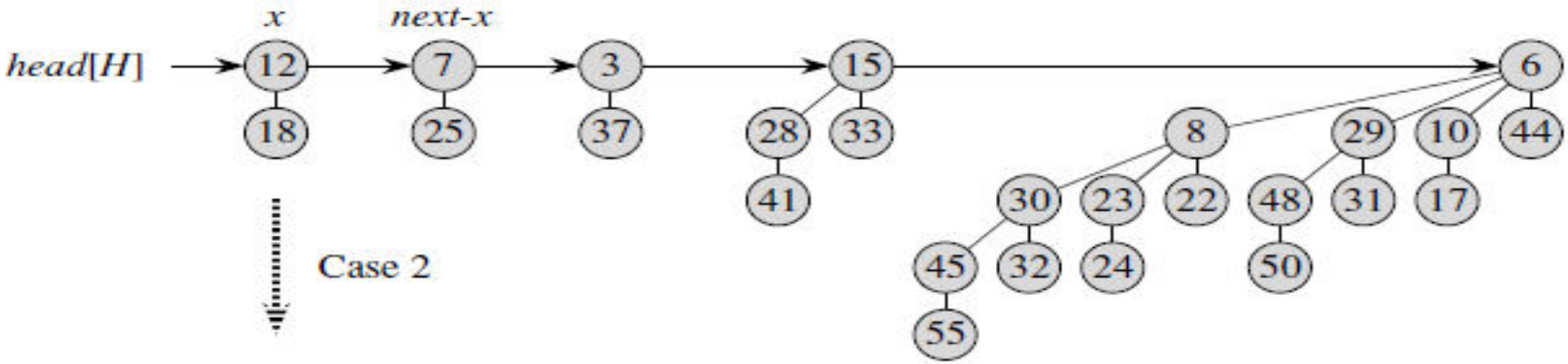
Union of two binomial heaps

Solution:

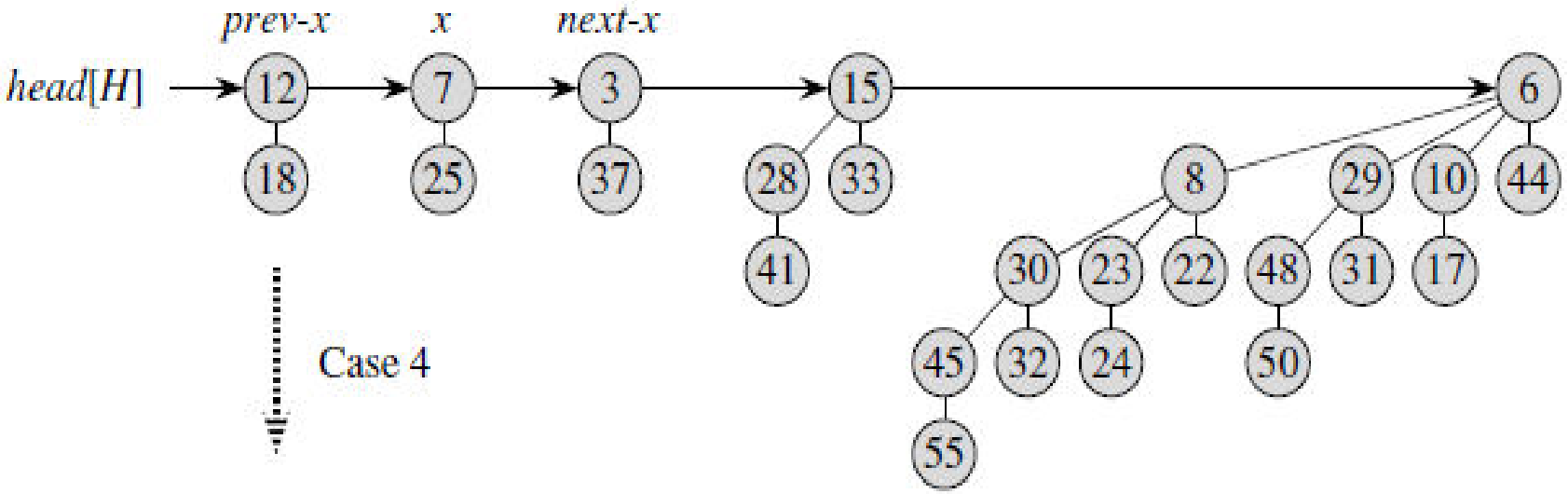
Step-1: Merge both binomial heaps.

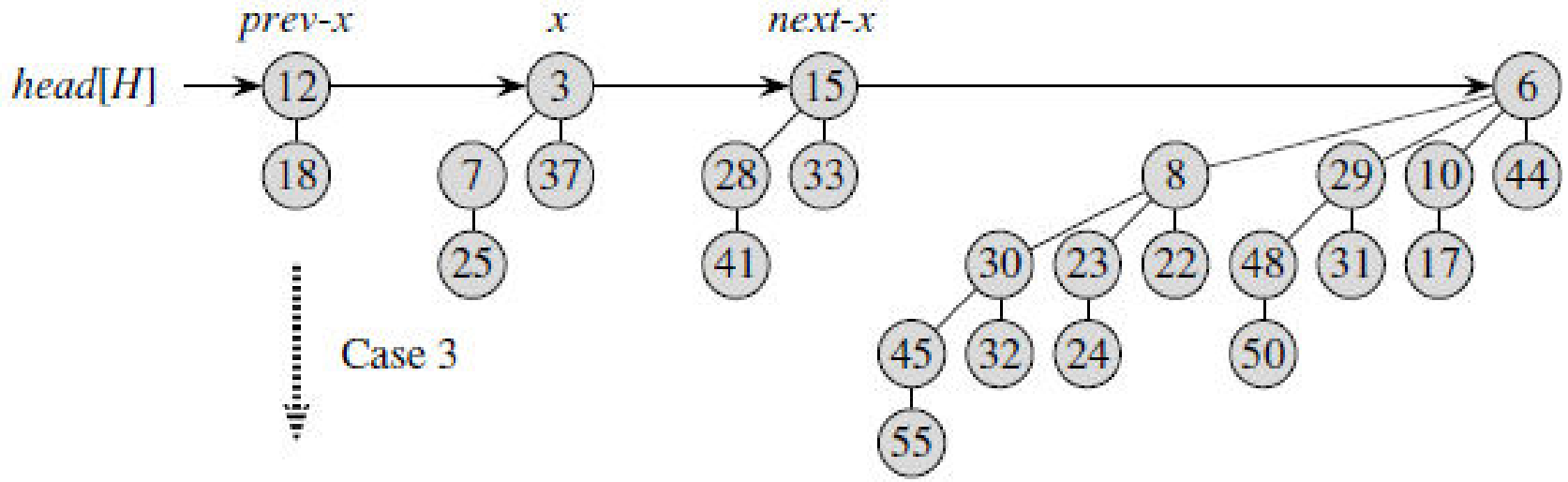


Step-2: Apply the linking process of equal degree root nodes.

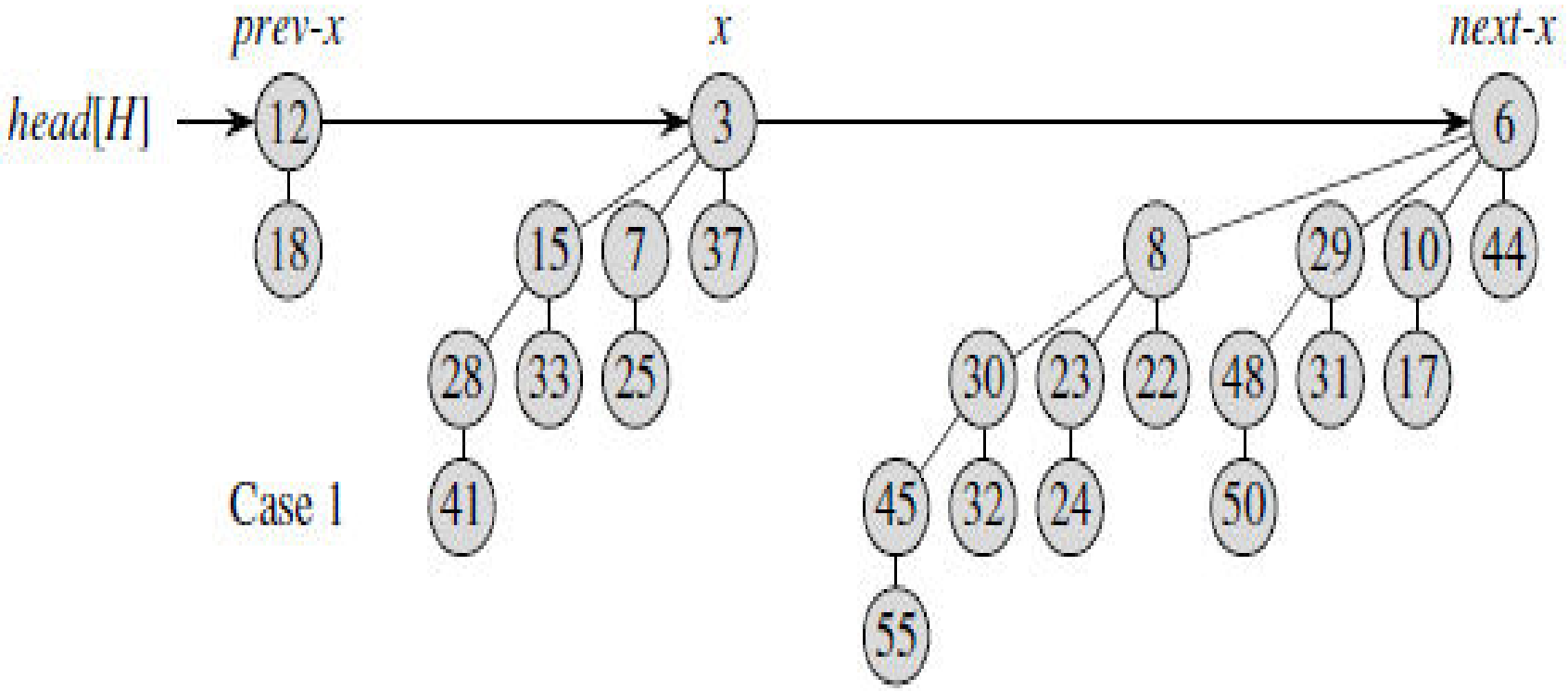


Union of two binomial heaps





Union of two binomial heaps



Final binomial heap

Union of two binomial heaps

```
BINOMIAL-HEAP-UNION(H1, H2)
 1  H ← MAKE-BINOMIAL-HEAP()
 2  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
 3  free the objects H1 and H2 but not the lists they point to
 4  if head[H] = NIL
 5     then return H
 6  prev-x ← NIL
 7  x ← head[H]
 8  next-x ← sibling[x]
 9  while next-x ≠ NIL
10        do if (degree[x] ≠ degree[next-x]) or
                 (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
11              then prev-x ← x                                  ▷ Cases 1 and 2
12                   x ← next-x                                  ▷ Cases 1 and 2
13              else  if key[x] ≤ key[next-x]
14                       then sibling[x] ← sibling[next-x]               ▷ Case 3
15                            BINOMIAL-LINK(next-x, x)                   ▷ Case 3
16                       else  if prev-x = NIL                           ▷ Case 4
17                                then head[H] ← next-x                  ▷ Case 4
18                                else  sibling[prev-x] ← next-x         ▷ Case 4
19                            BINOMIAL-LINK(x, next-x)                   ▷ Case 4
20                            x ← next-x                                 ▷ Case 4
21           next-x ← sibling[x]
22  return H
```

Union of two binomial heaps

The BINOMIAL-HEAP-UNION procedure has two phases.

- The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H1 and H2 into a single linked list H that is sorted by degree into monotonically increasing order.
- In the second phase, we link roots of equal degree until at most one root remains of each degree.

```
BINOMIAL-LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z] + 1
```

Time Complexity:

Time complexity of BINOMIAL-HEAP-UNION is O(lg n).

Inserting a node

The following procedure inserts node x into binomial heap H, assuming that x has already been allocated and key[x] has already been filled in.

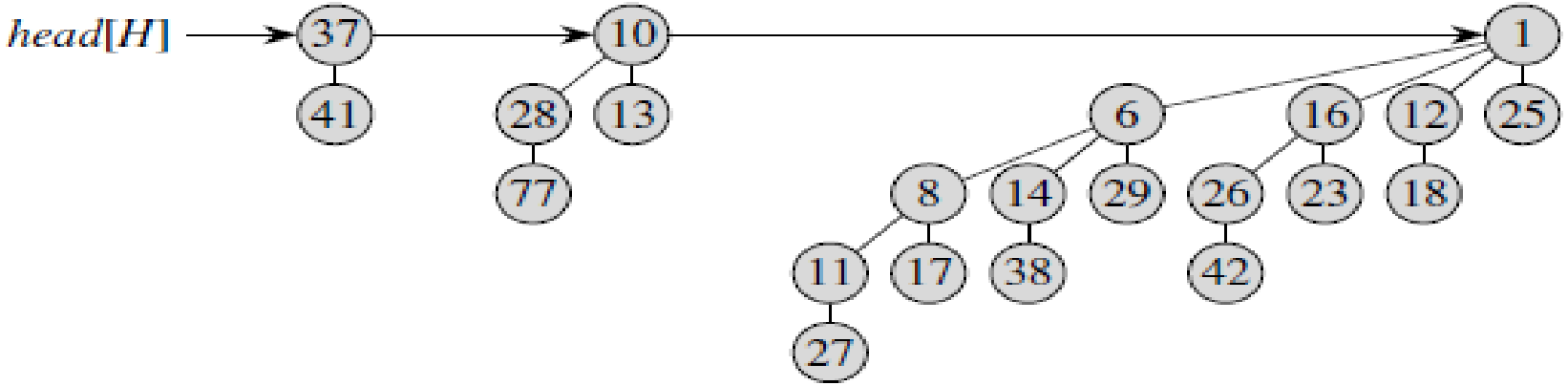
```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

Time Complexity:

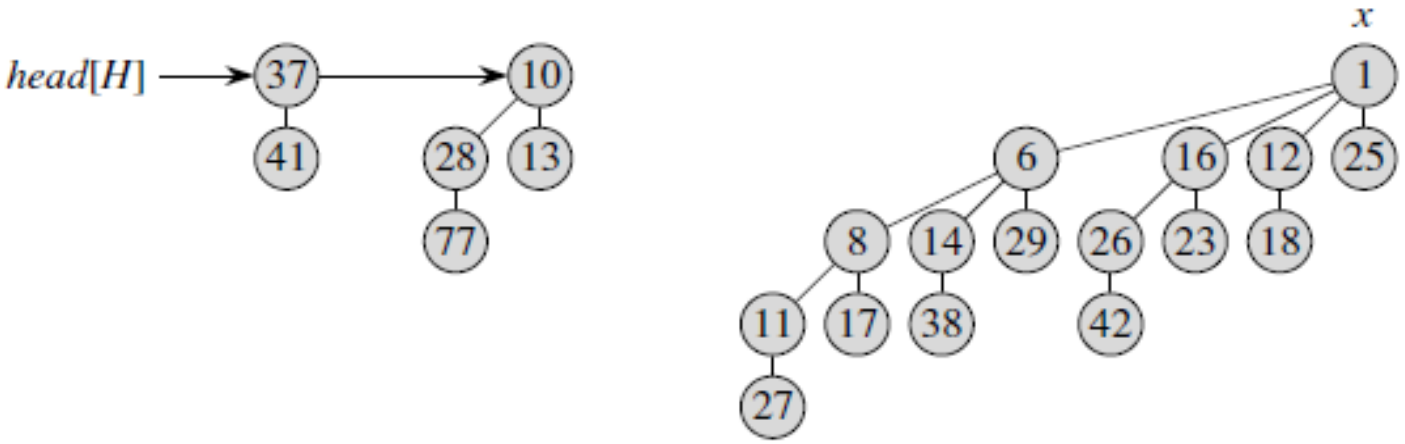
Time complexity of this algorithm is O(lg n).

Extracting the node with minimum key

Example: Extract the node with the minimum key from the following binomial heap H:-

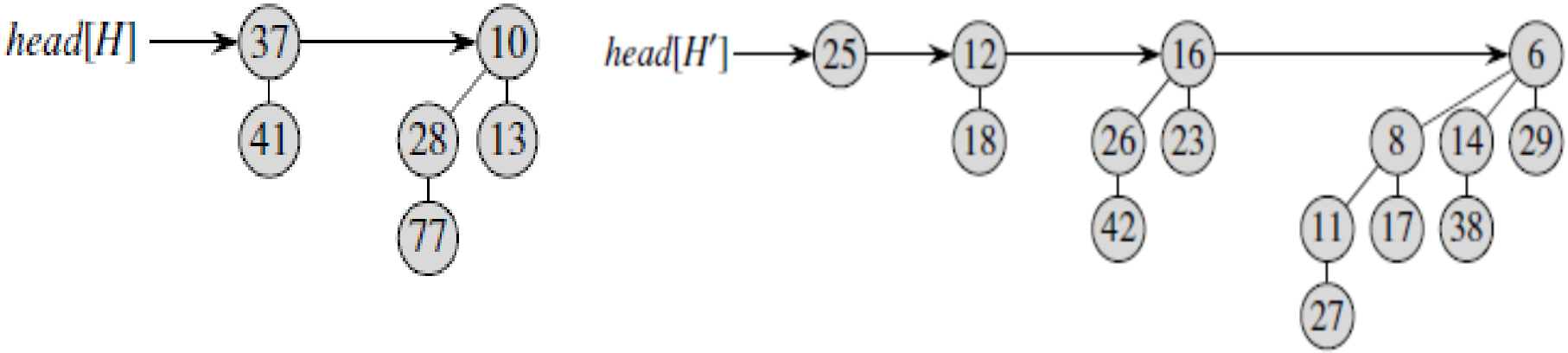


Solution: Step-1: Find the node with minimum key and remove that node.

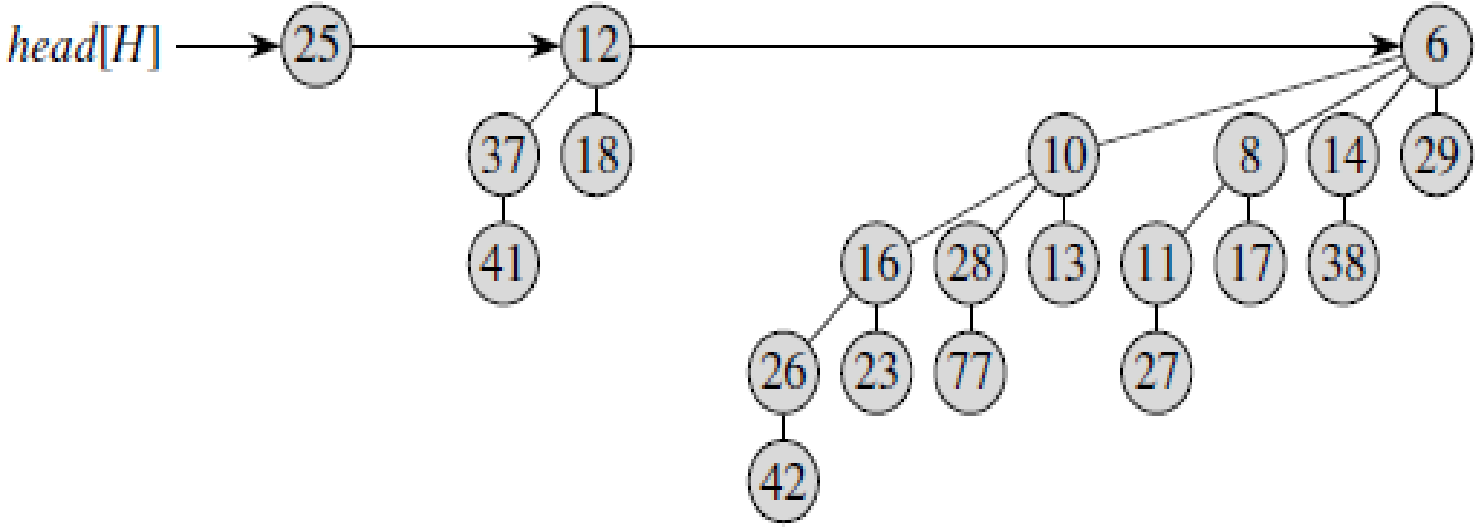


Extracting the node with minimum key

Step-2: Make the binomial heap H' from children of minimum node.



Step-3: Find the union of H and H'.



Extracting the node with minimum key

The following procedure extracts the node with the minimum key from binomial heap H and returns a pointer to the extracted node.

BINOMIAL-HEAP-EXTRACT-MIN(H)

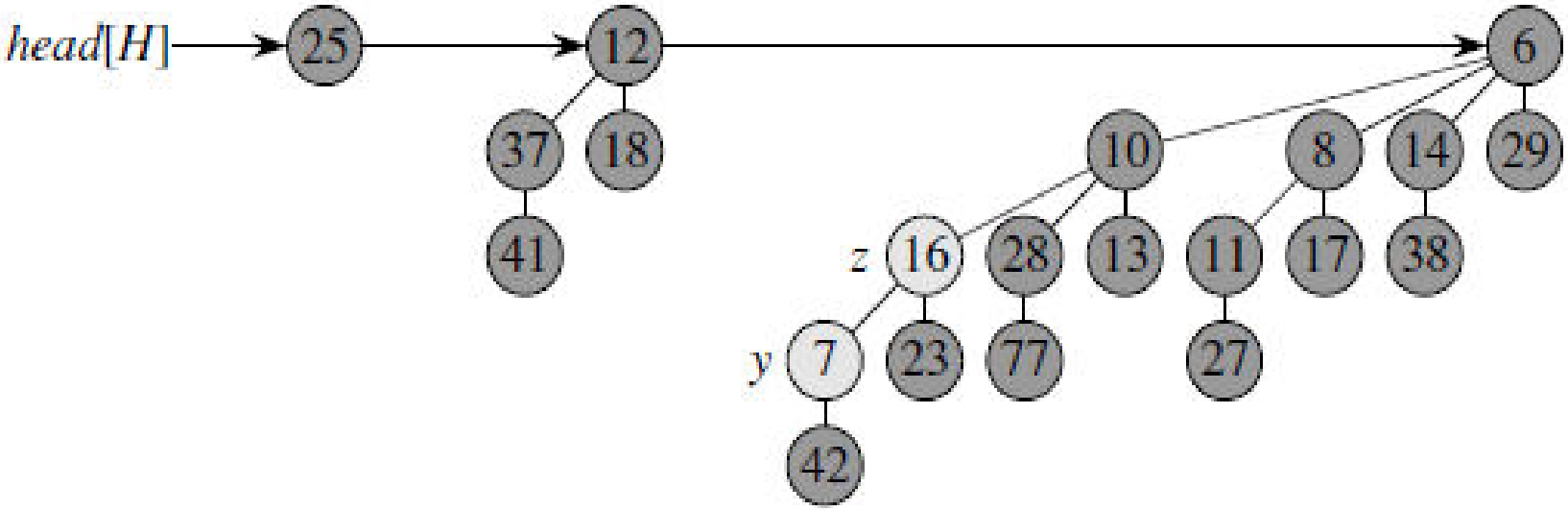
1 find the root x with the minimum key in the root list of H, and remove x from the root list of H
2 $H' \leftarrow$ MAKE-BINOMIAL-HEAP()
3 reverse the order of the linked list of x's children, and set $head[H']$ to point to the head of the resulting list
4 $H \leftarrow$ BINOMIAL-HEAP-UNION(H, H')
5 **return** x

Time Complexity:

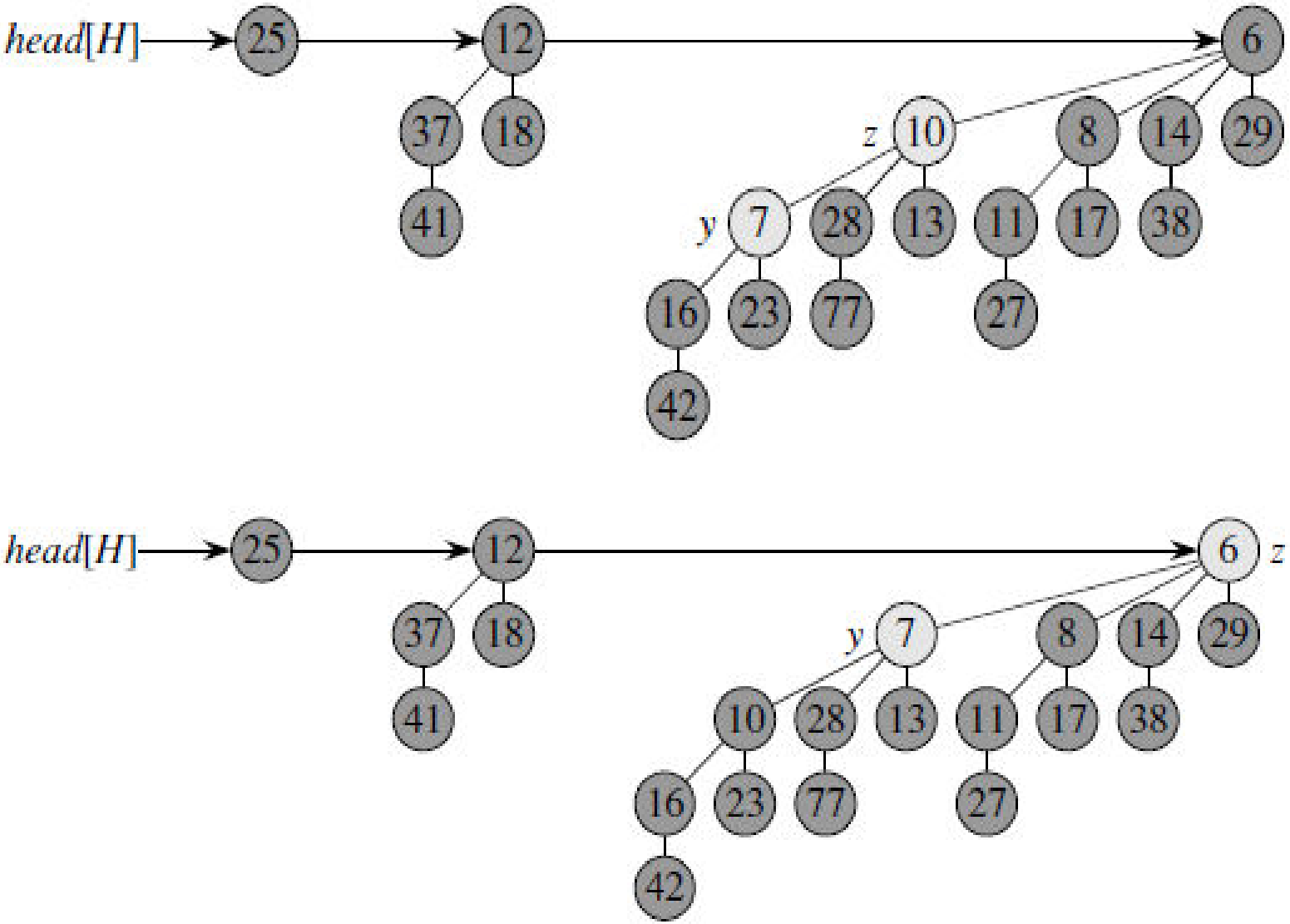
Time complexity of this algorithm is O(lg n).

Decreasing a key

Example: Decrease the value of a node y to be 7.



Decreasing a key



Decreasing a key

The following procedure decreases the key of a node x in a binomial heap H to a new value k. It signals an error if k is greater than x's current key.

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
 1  if k > key[x]
 2     then error "new key is greater than current key"
 3  key[x] ← k
 4  y ← x
 5  z ← p[y]
 6  while z ≠ NIL and key[y] < key[z]
 7        do exchange key[y] ↔ key[z]
 8           ▷ If y and z have satellite fields, exchange them, too.
 9           y ← z
10           z ← p[y]
```

Time Complexity:

Time complexity of this algorithm is O(lg n).

Deleting a key

Following procedure is used to delete the key value of a node. This implementation assumes that no node currently in the binomial heap has a key of $-\infty$.

```
BINOMIAL-HEAP-DELETE(H, x)
1  BINOMIAL-HEAP-DECREASE-KEY(H, x, -∞)
2  BINOMIAL-HEAP-EXTRACT-MIN(H)
```

Time Complexity:

Time complexity of this algorithm is O(lg n).

AKTU examination questions

1. Explain various properties of Binomial Tree.
2. Prove that maximum degree of any node in an n node binomial tree is log n.
3. Explain the algorithm to extract the minimum elements in a binomial Heap. Give an example for the same.
4. Define Binomial Heap with example.
5. Explain the algorithm to delete a given element in a binomial Heap. Give an example for the same.
6. Explain properties of Binomial Heap. Write an algorithm to perform uniting two Binomial Heaps. And also to find Minimum Key.
7. Explain the different conditions of getting union of two existing binomial Heaps. Also write algorithm for union of two Binomial Heaps. What is its complexity?

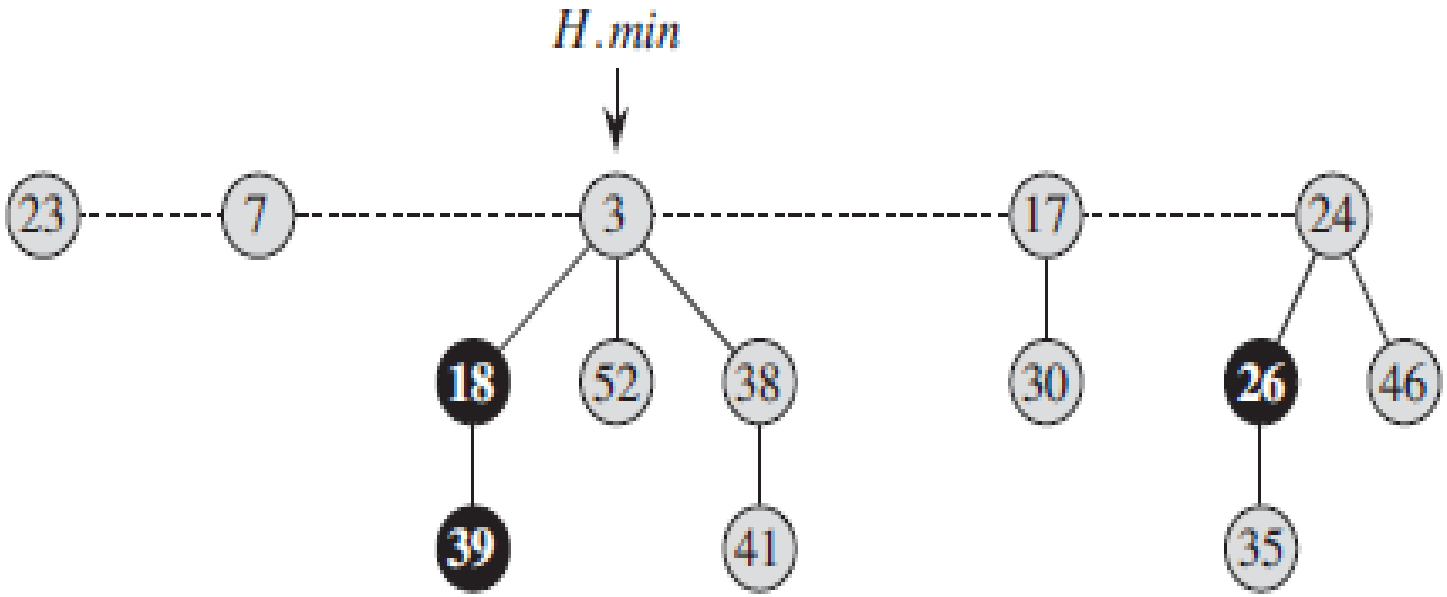
Fibonacci Heap

Fibonacci Heap

Definition

A Fibonacci heap is a collection of rooted trees that are min-heap ordered. That is, each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent.

Example:



Representation of Fibonacci Heap

- Circular doubly linked list is used to represent Fibonacci heap.
- Each node x in binomial heap consists of following fields:-

  x.p → pointer points to parent of x

  x.child → pointer points to any child of x

  x.left → pointer points to left sibling of x

  x.right → pointer points to right sibling of x

  x.key → value stored at node x

  x.degree→ number of children of node x

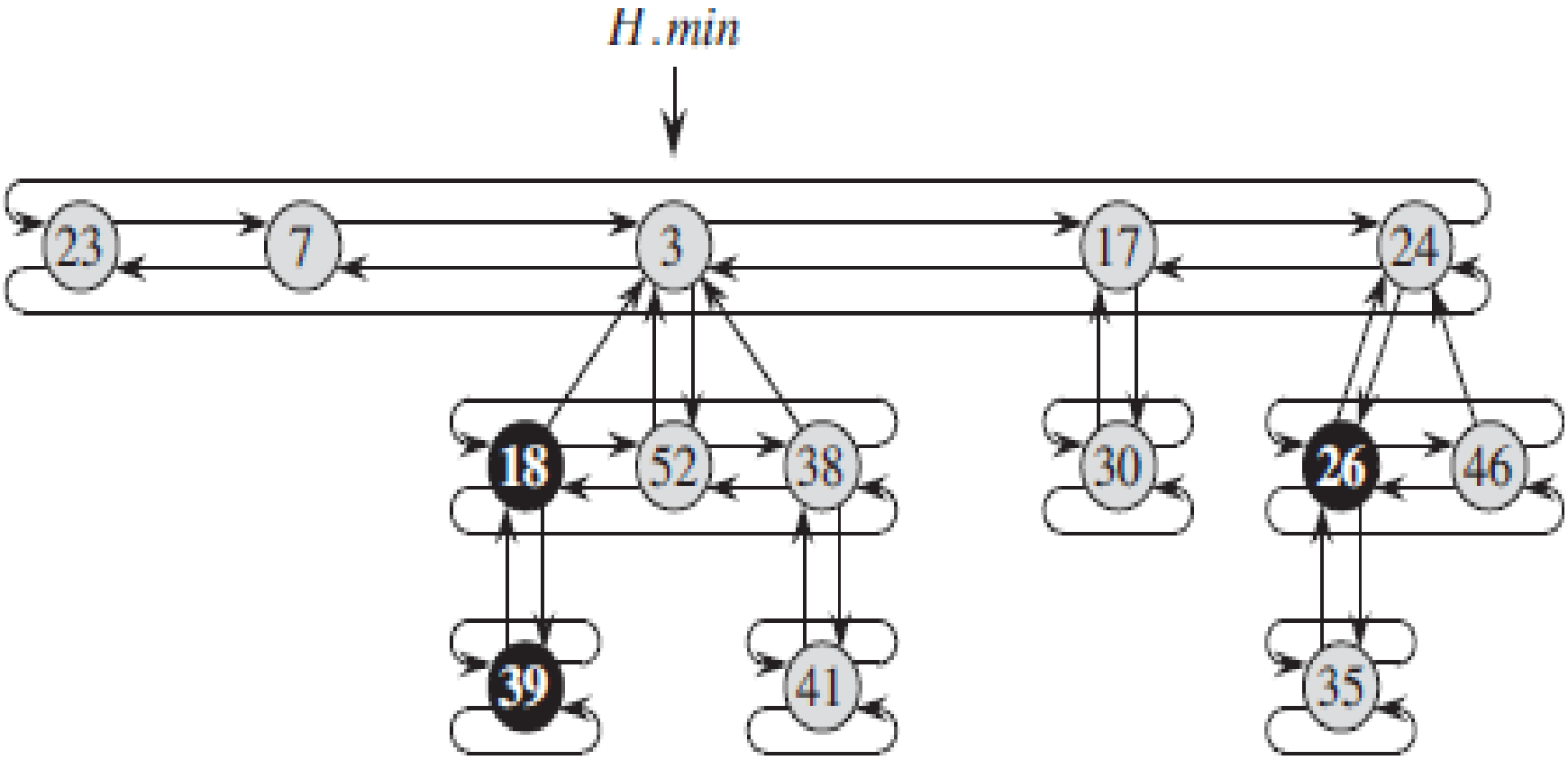
  x.mark → The boolean-valued attribute indicates whether node x has lost a child since the last time x was made the child of another node.

- Fibonacci heap H has two fields:- H.min and H.n .

H.min → pointer points to the root of a tree containing the minimum key;

H.n → the number of nodes currently in H

Representation of Fibonacci Heap



Potential Function

To analyze the performance of Fibonacci heap, we use the potential function.

We then define the potential $\Phi(H)$ of Fibonacci heap H by

$$\Phi(H) = t(H) + 2m(H)$$

Where, $t(H)$ is the number of tree in H and $m(H)$ is the number of marked nodes.

Example: Consider the Fibonacci heap of previous slide.

Here, t(H) = 5 and m(H) = 3. Therefore

$$\Phi(H) = 5 + 2*3 = 11$$

Maximum degree

Maximum degree of any n-node Fibonacci is denoted by D(n).

$D(n) \leq \lfloor \log n \rfloor$

Amortized Cost

Amortized cost is computed for an operation. It is defined as following:-

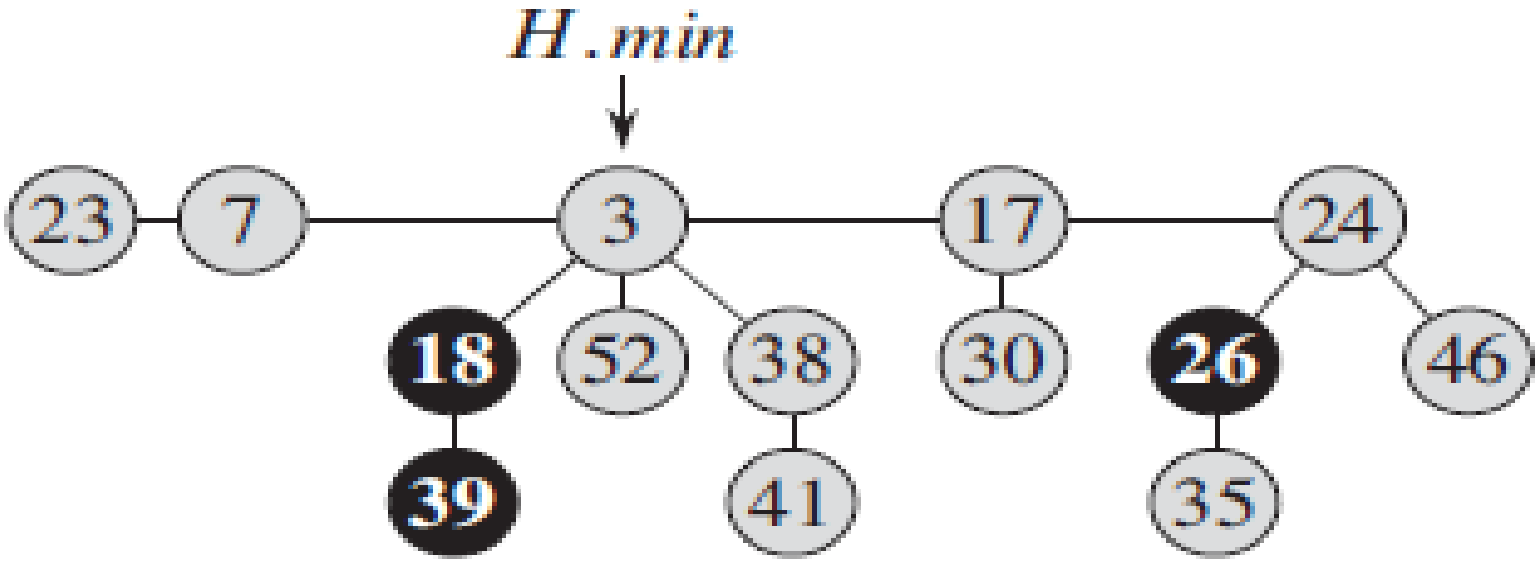
Amortized cost = Actual cost + change in potential function due to operation

Mergeable-heap operations

1. Inserting a node
2. Finding the minimum node
3. Uniting two Fibonacci heaps
4. Extracting the minimum node
5. Decreasing a key
6. Deleting a node

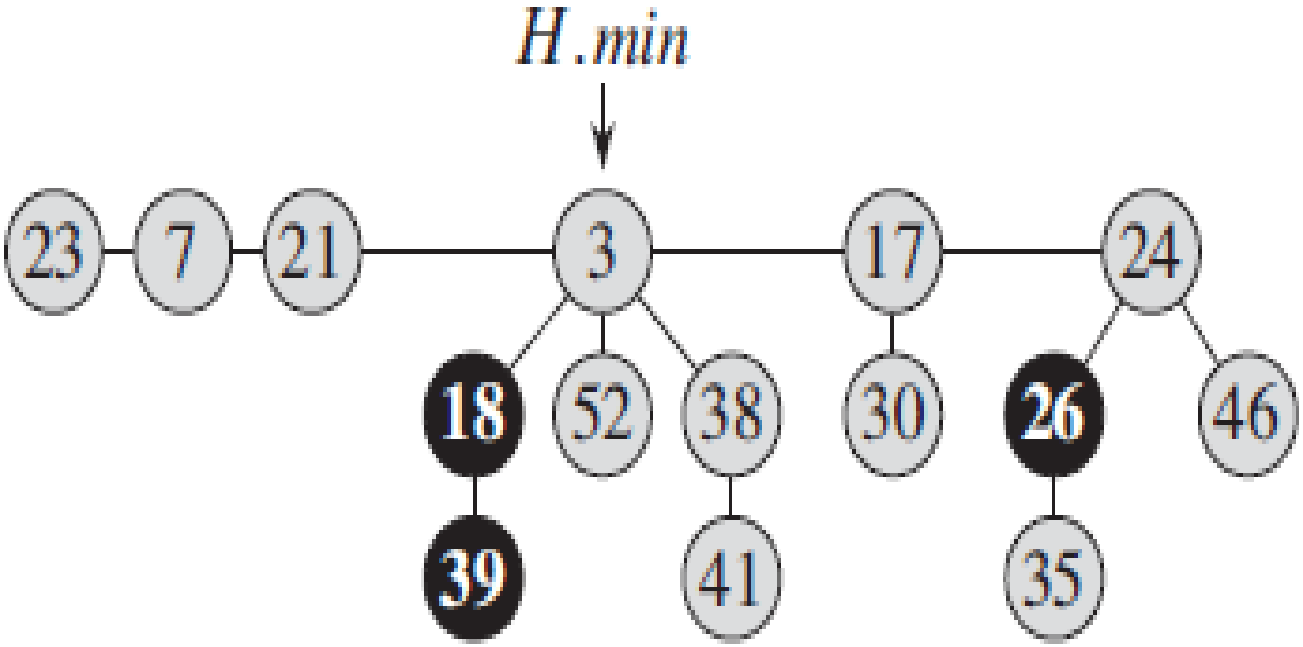
Inserting a node

Example: Insert a node with key 21 in the following Fibonacci heap.



Inserting a node

Solution: Fibonacci heap after inserting element 21 in the Fibonacci heap.



Inserting a node

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that x:key has already been filled in.

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6       create a root list for H containing just x
 7       H.min = x
 8  else insert x into H's root list
 9       if x.key < H.min.key
10            H.min = x
11  H.n = H.n + 1
```

Inserting a node

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then,

$t(H') = t(H) + 1$

and

$m(H') = m(H)$,

and the increase in potential = $\Phi(H') - \Phi(H)$

$= t(H') + 2m(H') - (t(H) + 2m(H))$

$= t(H) + 1 + 2m(H) - (t(H) + 2m(H))$

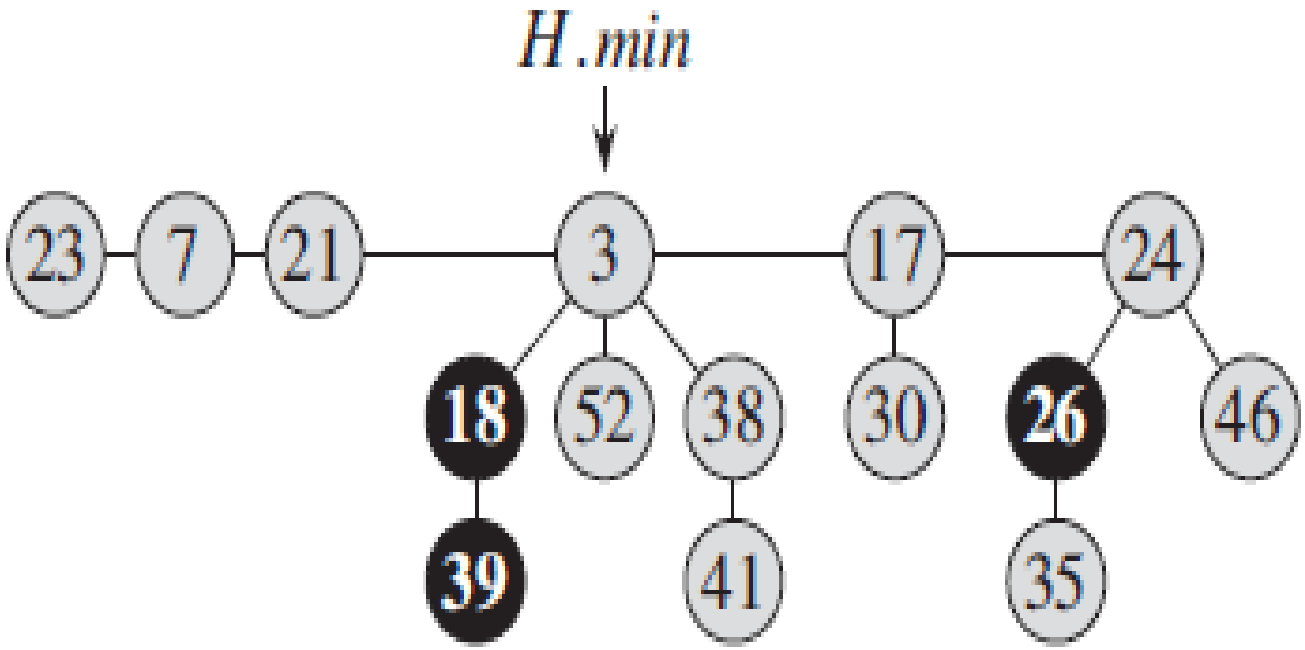
$= 1$

The actual cost = O(1), therefore

Amortized cost = Actual cost + $\Phi(H') - \Phi(H)$

$= O(1) + 1 =$ **O(1)**

Finding the minimum node



The minimum node of a Fibonacci heap H is given by the pointer H.min, so we can find the minimum node in O(1) actual time. Because the potential of H does not change, therefore the amortized cost of this operation is equal to its O(1) actual cost.

Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H1 and H2, destroying H1 andH2 in the process. It simply concatenates the root lists of H1 and H2 and then determines the new minimum node. Afterward, the objects representing H1 and H2 will never be used again.

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
5        H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

Uniting two Fibonacci heaps

Amortied cost:

The change in potential function

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2))$$

$$= t(H) + 2m(H) - (t(H_1) + 2m(H_1) + t(H_2) + 2m(H_2))$$

$$= 0$$

Because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$

Therefore the amortized cost

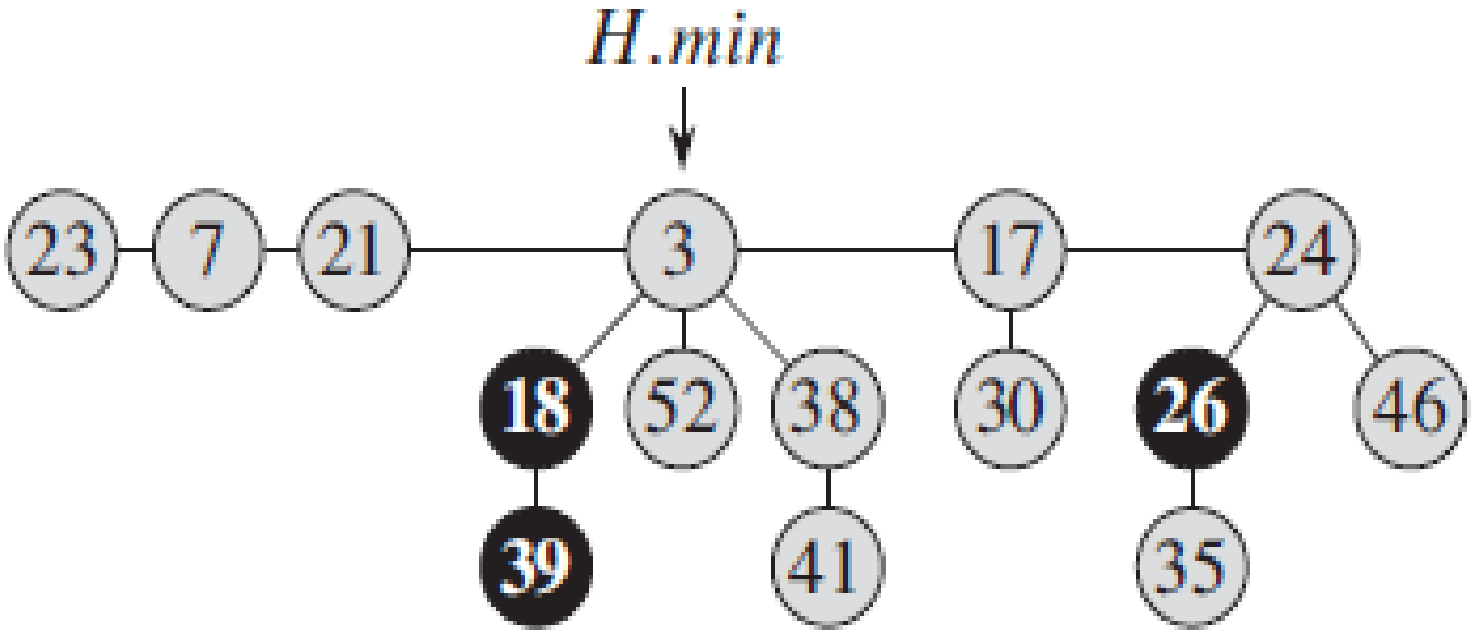
= actual cost + change in potential

= O(1) + 0

= O(1)

Extracting the minimum node

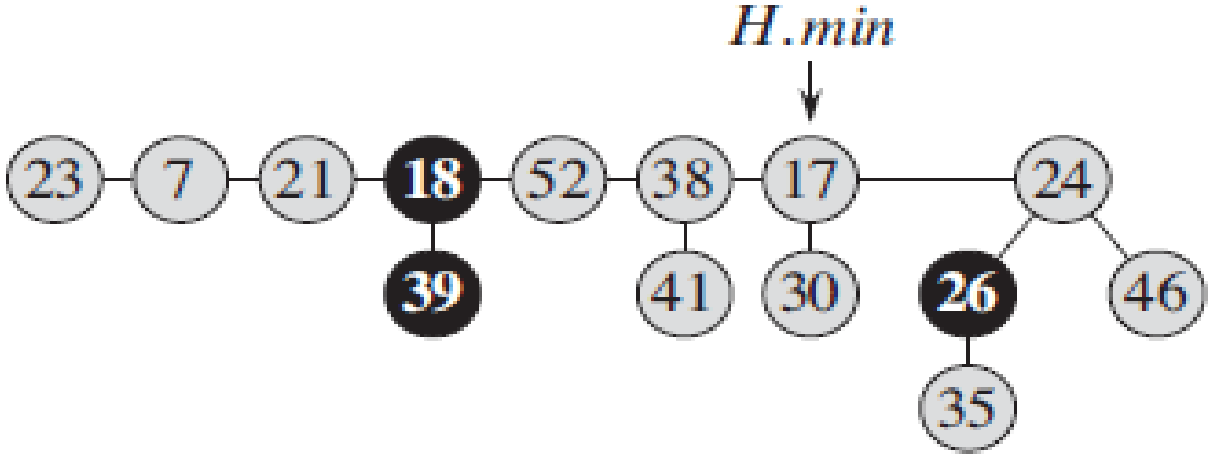
Example: Extract the minimum node from the following Fibonacci heap.



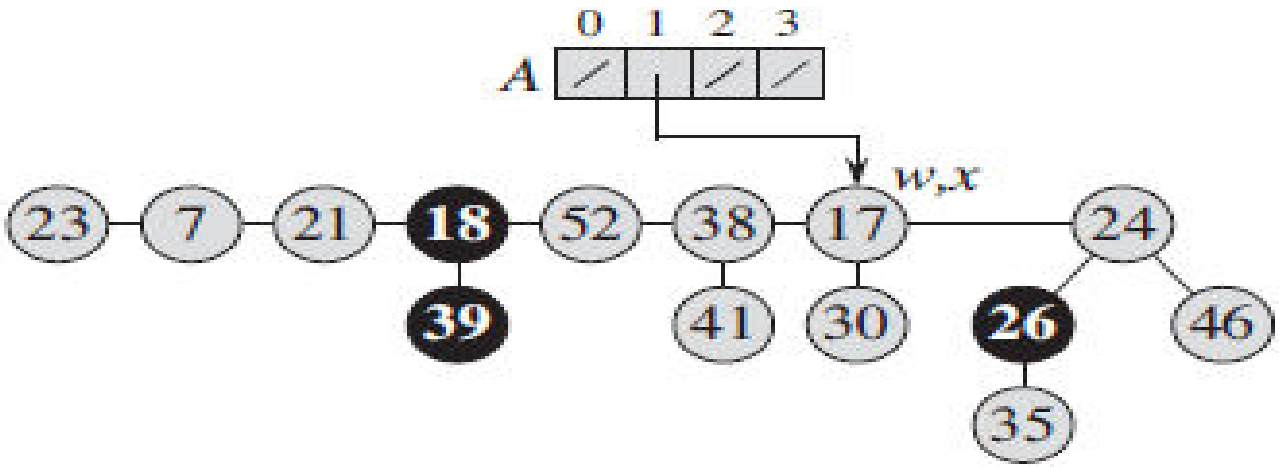
Extracting the minimum node

Solution:

Step-1:

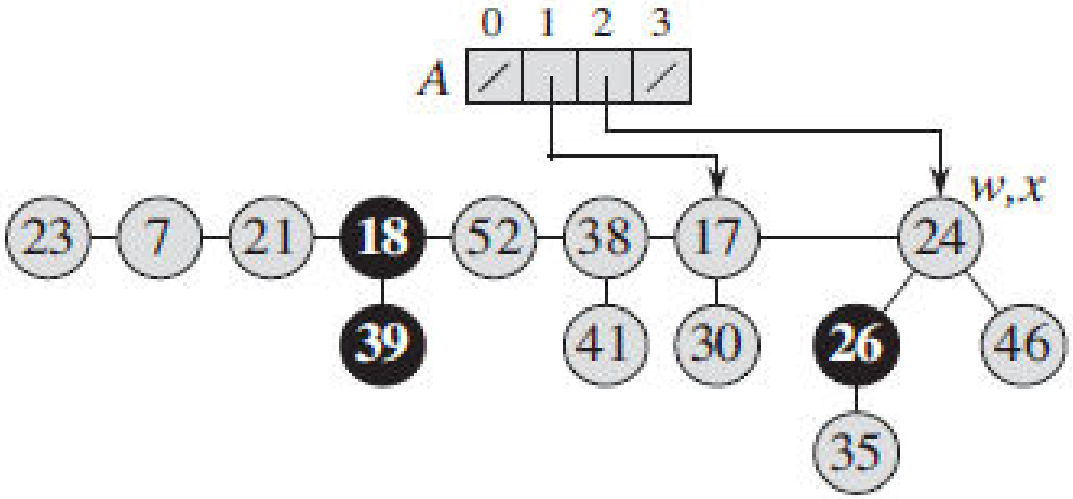


Step-2:

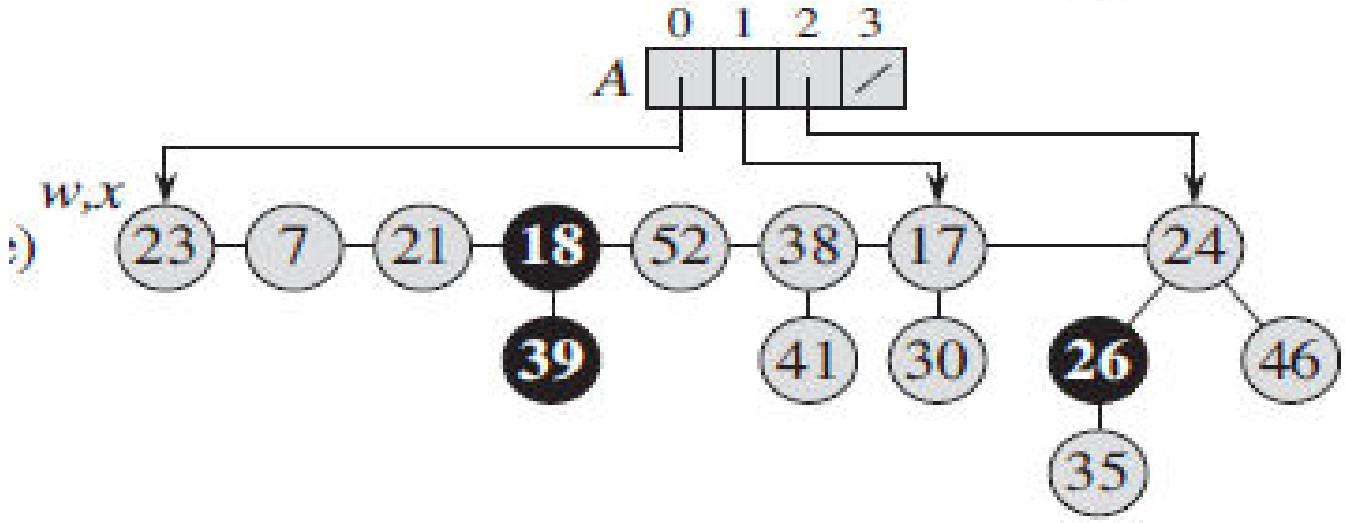


Extracting the minimum node

Step-3:

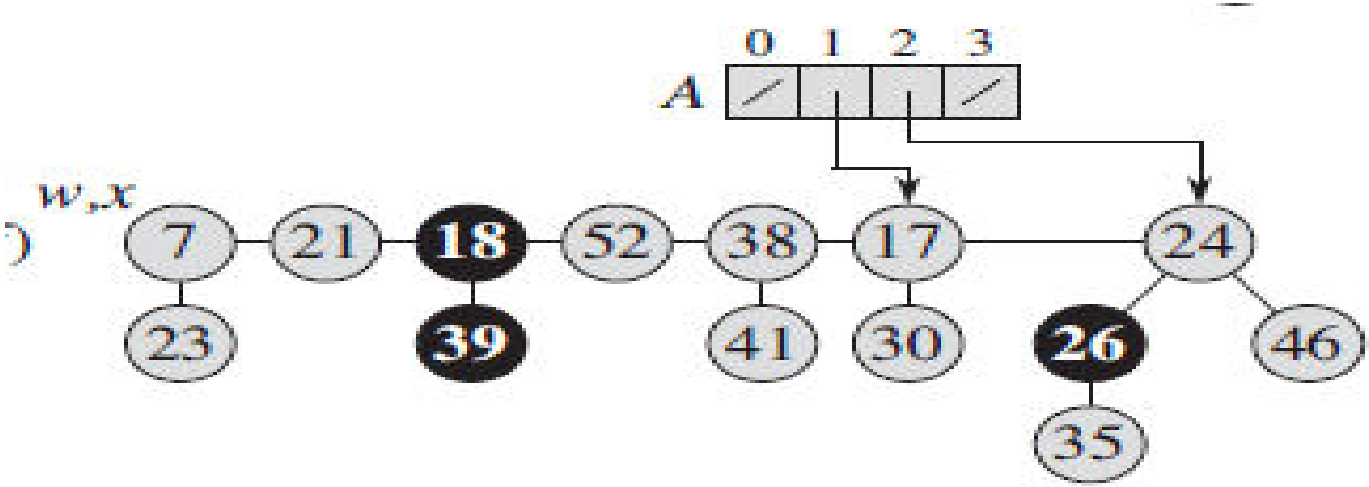


Step-4·

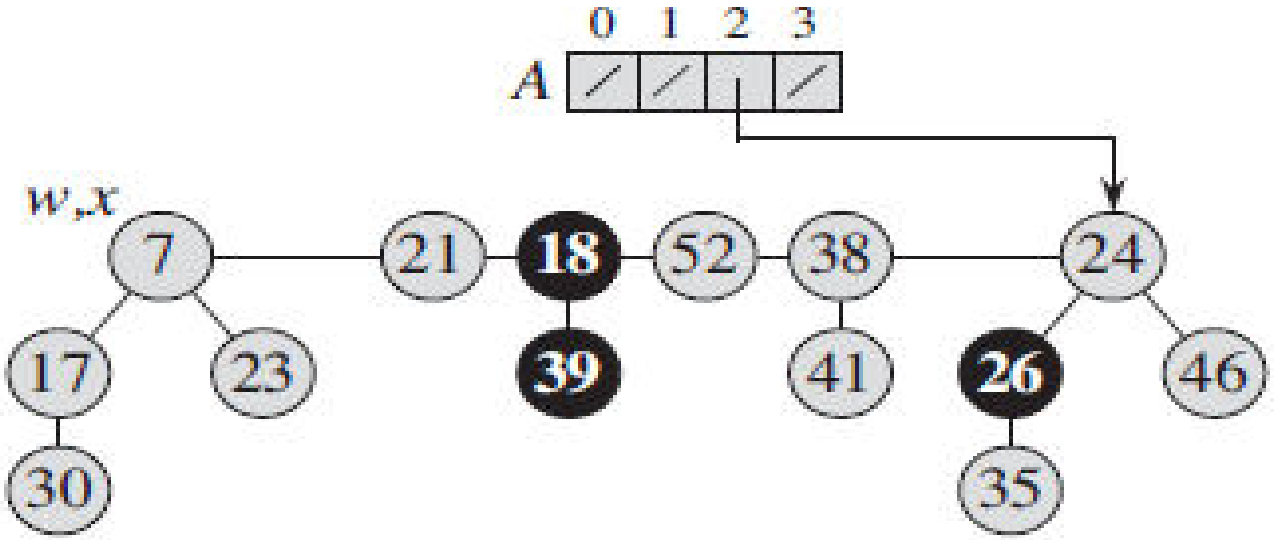


Extracting the minimum node

Step-5:

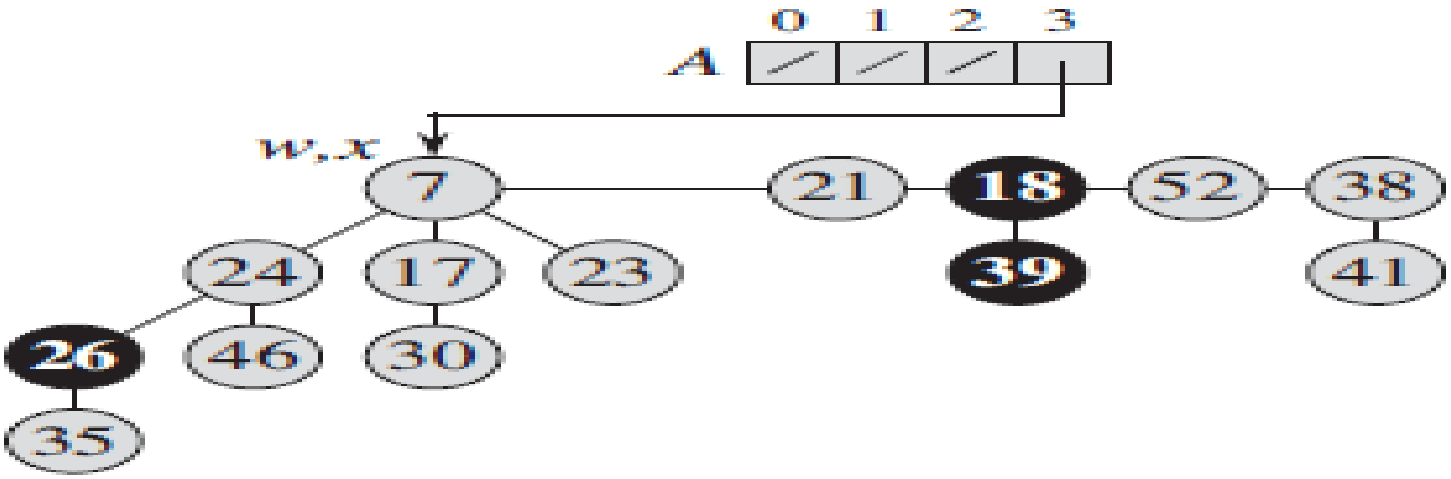


Step-6:

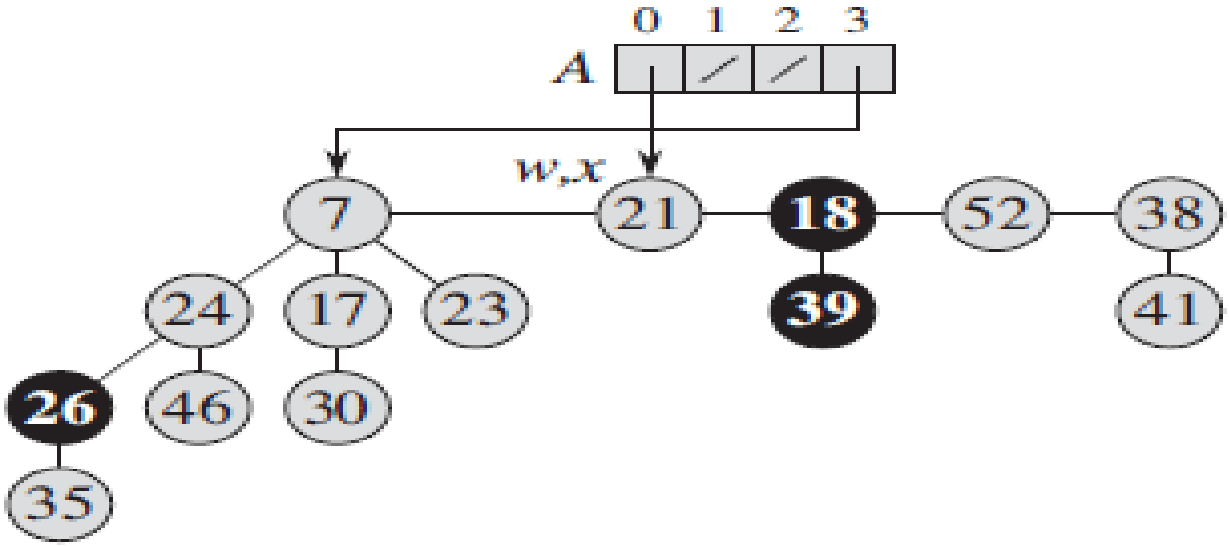


Extracting the minimum node

Step-7:

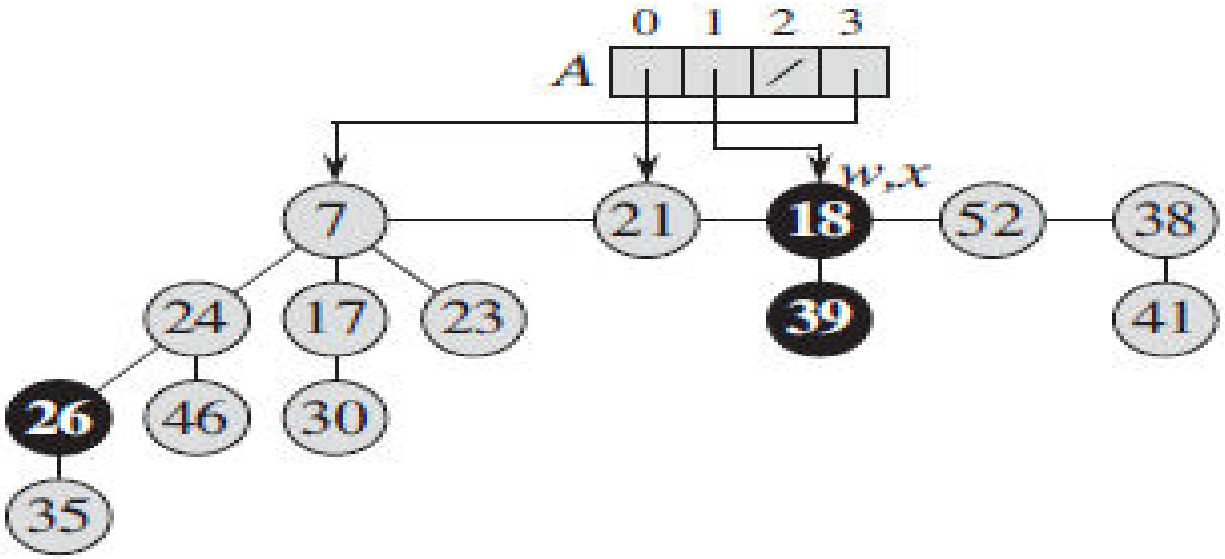


Step-8:

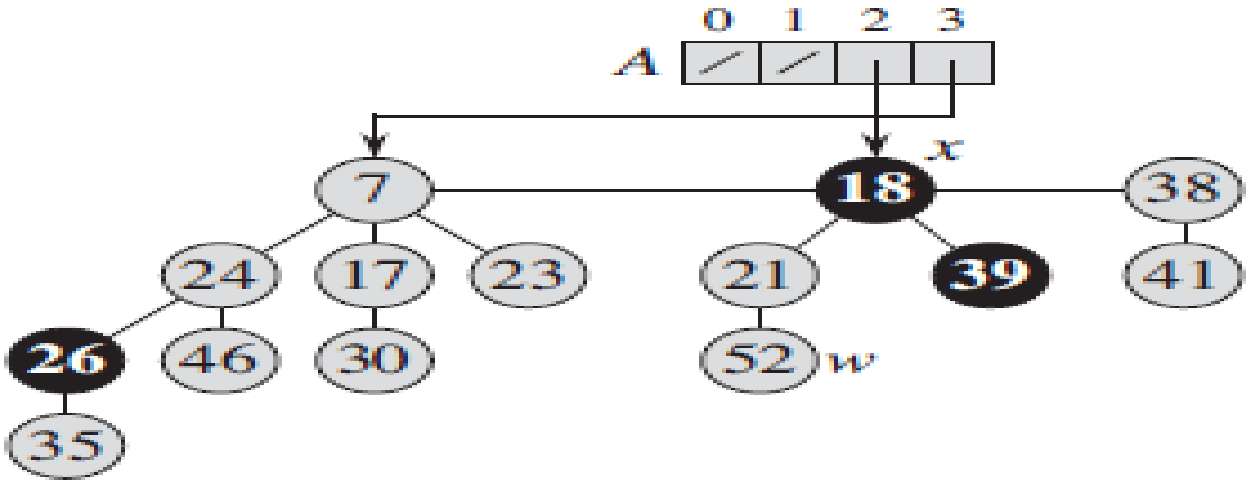


Extracting the minimum node

Step-9:

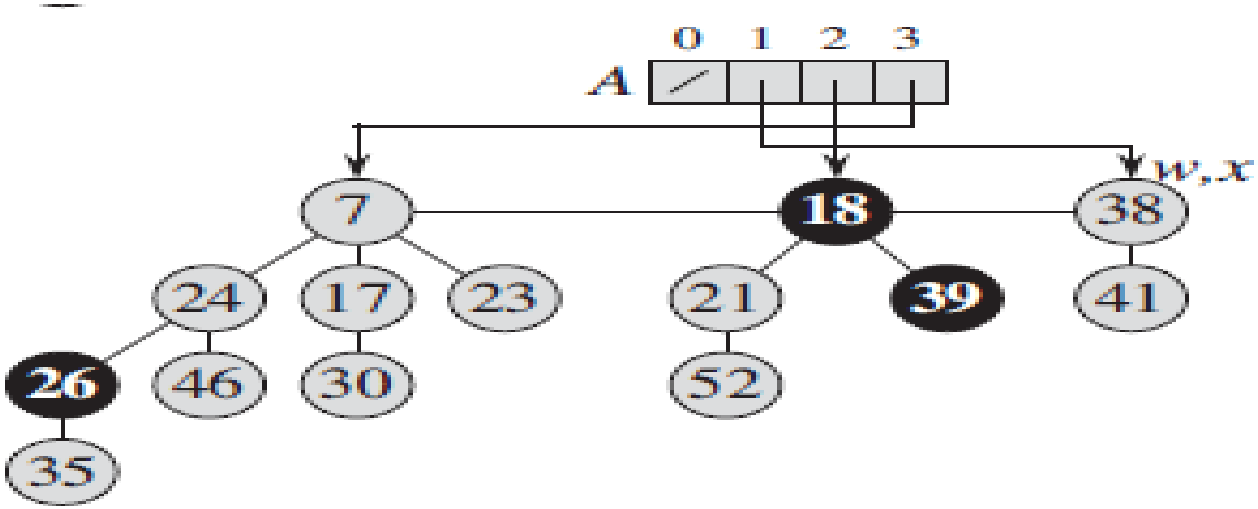


Step-10:

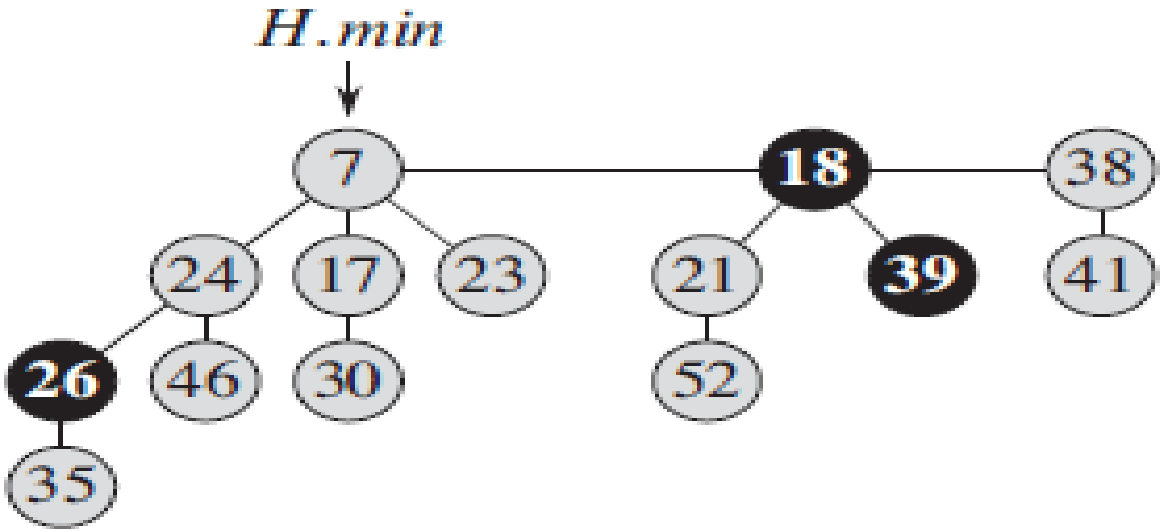


Extracting the minimum node

Step-11:



Step-12:



Final Fibonacci Heap

Extracting the minimum node

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3       for each child x of z
 4            add x to the root list of H
 5            x.p = NIL
 6       remove z from the root list of H
 7       if z == z.right
 8            H.min = NIL
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n − 1
12  return z
```

Extracting the minimum node

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]          // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

Extracting the minimum node

FIB-HEAP-LINK(H, y, x)

1 remove y from the root list of H
2 make y a child of x, incrementing $x.degree$
3 $y.mark$ = FALSE

Computation of Amortized cost:

Let H denote the Fibonacci heap just prior to the FIB-HEAP-EXTRACT-MIN operation. Let n is the number of nodes in Fibonacci heap H. Let H' is the Fibonacci heap after this operation. Therefore,

Actual cost = O(t(H)-1 + D(n)) = O(D(n) + t(H))

Now, t(H') = D(n) and m(H') = m(H)

Therefore, amortized cost = actual cost + change in potential

= O(D(n) + t(H)) + (Φ(H') - Φ(H))

= O(D(n) + t(H)) + (D(n) + 2m(H) – t(H) -2m(H))

= O(D(n) + t(H) + D(n) – t(H))

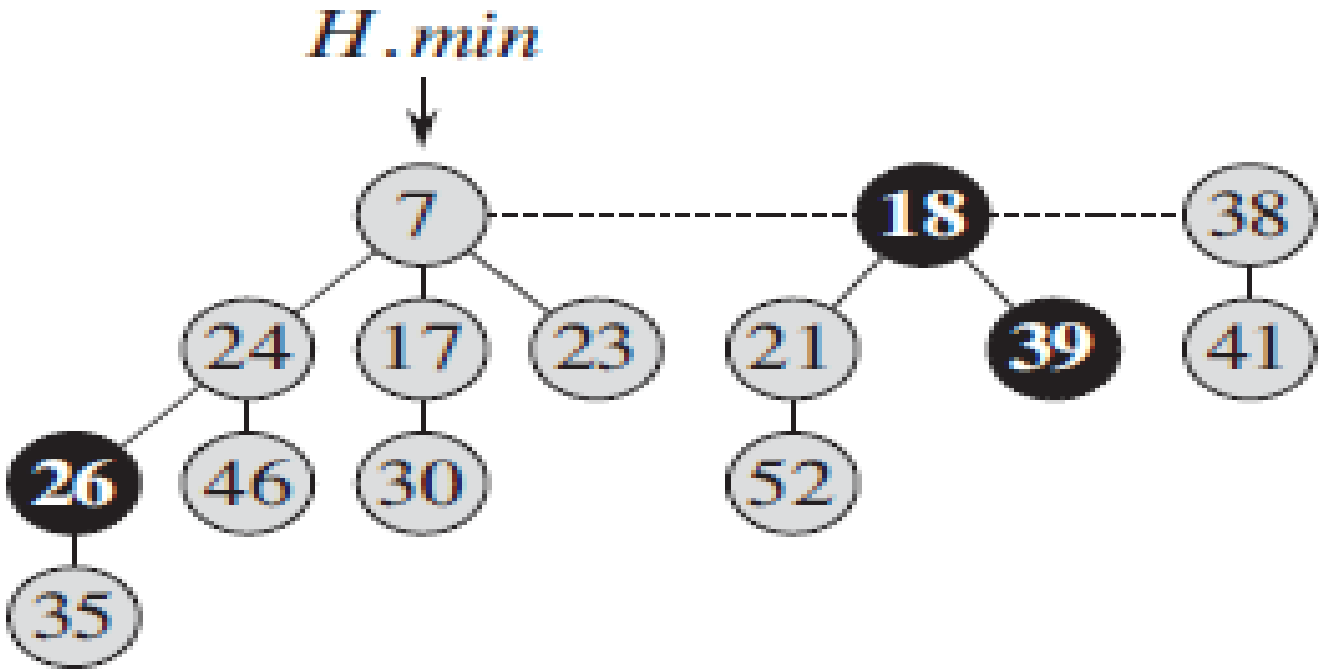
= O(D(n)) = O(log n))

Decreasing a key

Example: Consider following Fibonacci heap.

(1) Decrease the node with key 46 to key value 15.

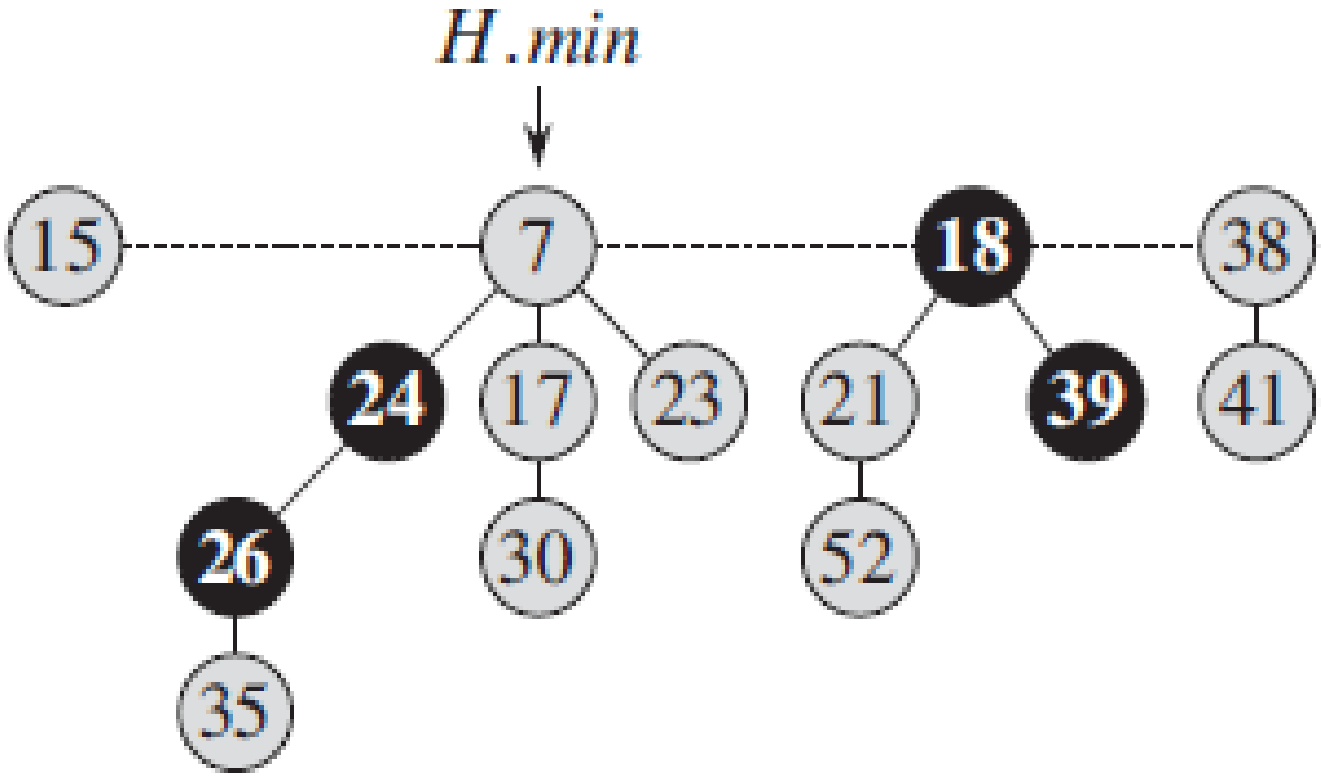
(2) After this, decrease node with key 35 to key value 5.



Decreasing a key

Solution:

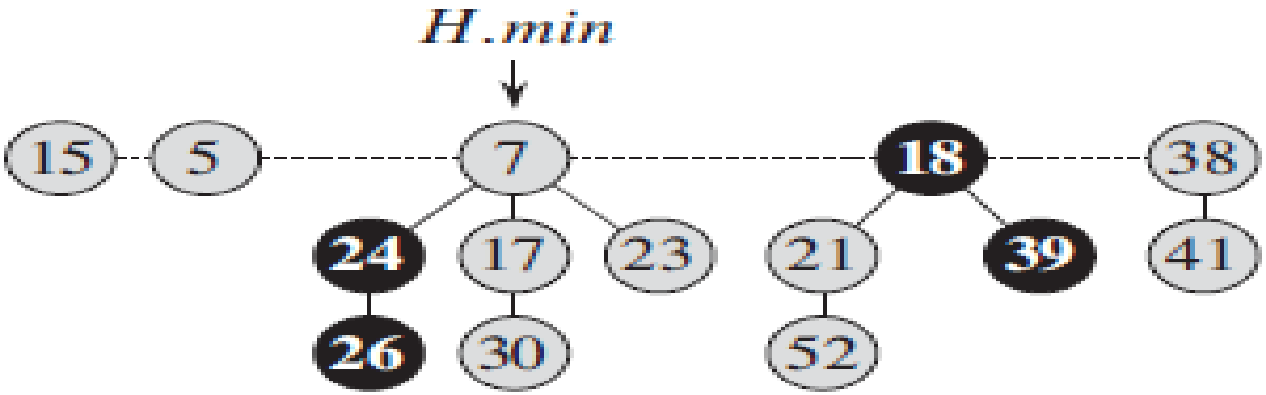
(1) Decrease the node with key 46 to key value 15.



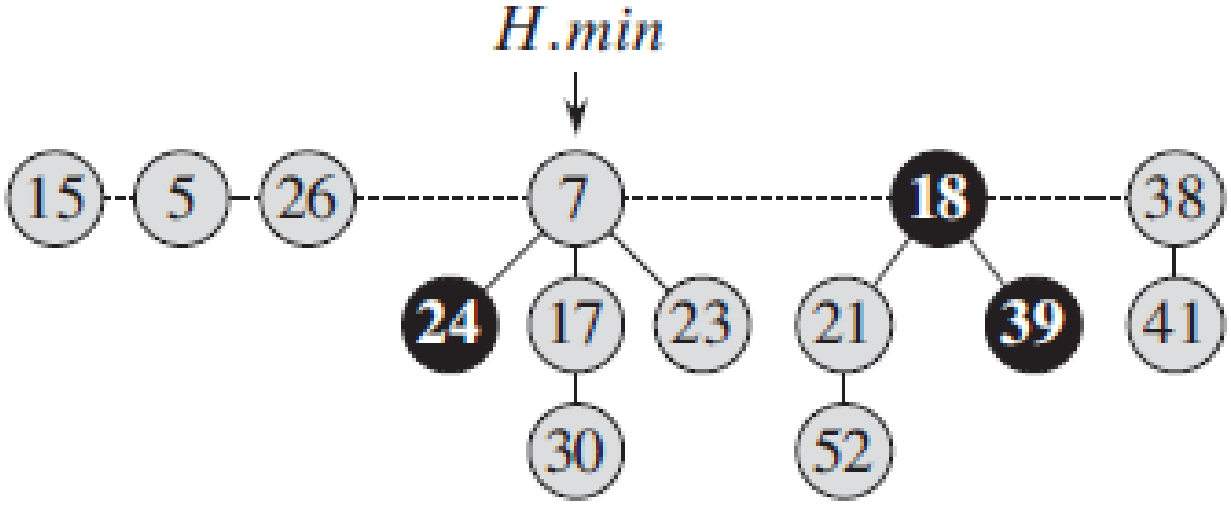
Decreasing a key

2. Decrease the node with key 35 to key value 5.

Step-1:

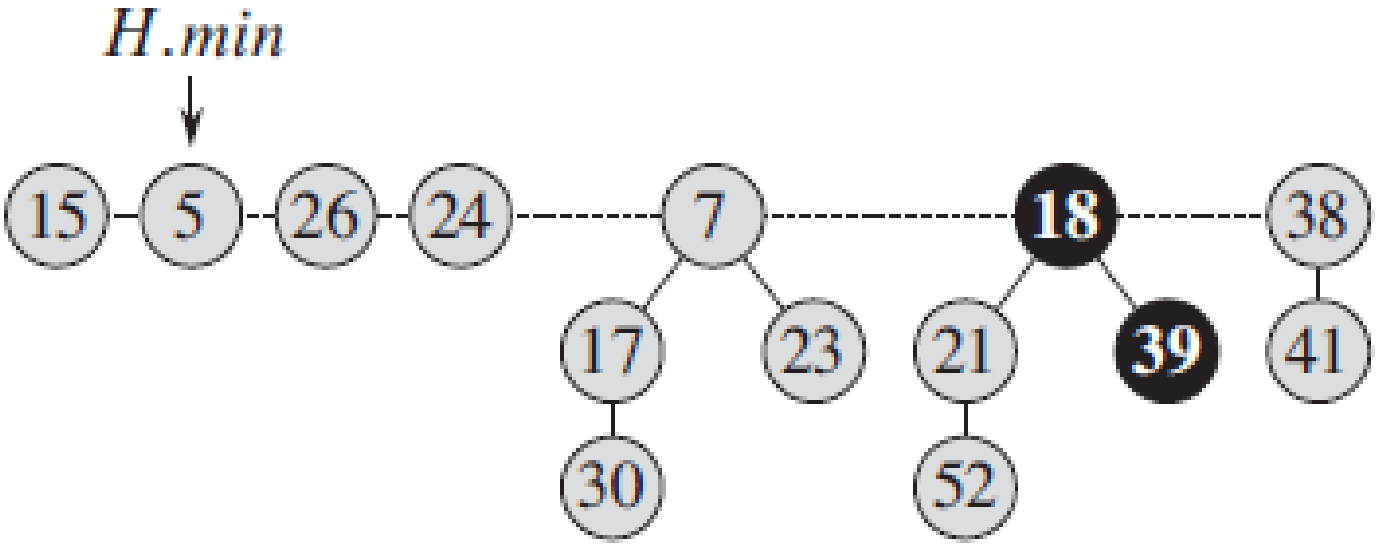


Step-2:



Decreasing a key

Step-3:



Final Fibonacci heap

Decreasing a key

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x
```

Decreasing a key

```
CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE

CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3       if y.mark == FALSE
4            y.mark = TRUE
5       else CUT(H, y, z)
6            CASCADING-CUT(H, z)
```

Decreasing a key

Amortized cost:

Suppose the cascading cut function is called c times.

Therefore, the actual cost of FIB-HEAP-DECREASE-KEY is $O(c)$.

Now, Let H is the initial Fibonacci heap and H' is the Fibonacci heap after this operation. Therefore,

$$t(H') = t(H) + c$$

(the original t(H) trees, c-1 trees produced by cascading cuts, and the tree rooted at x)

Maximum number of marked nodes,

$$m(H') = m(H) - c + 2$$

(c -1 were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node)

Therefore, amortized cost = $O(c) + ((t(H') + 2m(H')) - (t(H) + 2m(H)))$

$= O(c) + (t(H) + c + 2(m(H) - c + 2) - (t(H) + 2m(H)))$

$= O(c) - c + 4 = O(4) = O(1)$