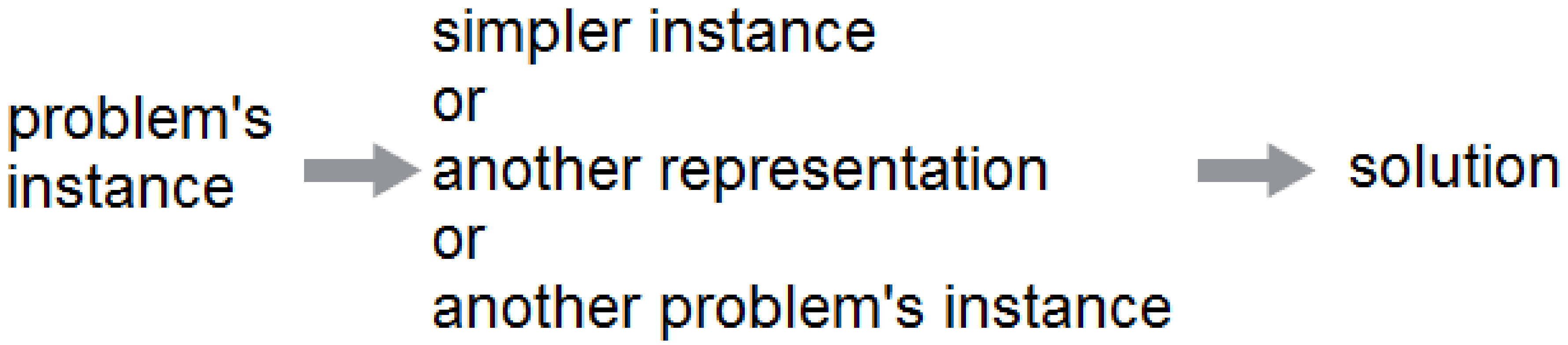
Design and Analysis of Algorithms (UE17CS251)

Unit III - Transform-and-Conquer

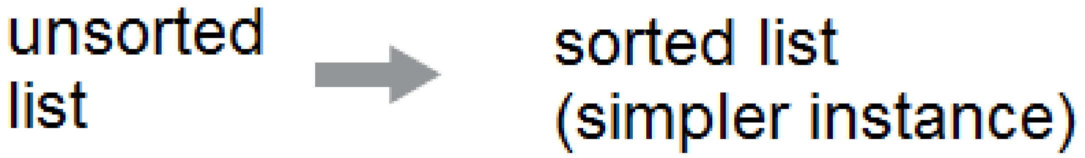
Mr. Channa Bankapur
channabankapur@pes.edu

Transform-and-Conquer:



Presorting:

Interest in sorting algorithms is due, to a significant degree, to the fact that many questions about a list are easier to answer if the list is sorted.



Finding the **largest element** in an array of n numbers using the following approaches:

1. Brute Force
2. Divide-n-Conquer
3. Decrease-n-Conquer
4. Transform-n-Conquer (Presorting-based)

Write an algorithm for:

Checking element uniqueness in an array

using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

```
ALGORITHM  PresortElementUniqueness(A[0..n − 1])
    //Solves the element uniqueness problem by sorting the array first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Returns "true" if A has no equal elements, "false" otherwise
    sort the array A
    for i ← 0 to n − 2 do
        if A[i] = A[i + 1] return false
    return true
```

$$\begin{aligned} T(n) = T_{sort}(n) + T_{scan}(n) &\in \Theta(n \log n) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Write an algorithm for:

Computing a mode in an array

using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

```
ALGORITHM  PresortMode(A[0..n − 1])
    //Computes the mode of an array by sorting it first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: The array's mode
    sort the array A
    $i \leftarrow 0$                          //current run begins at position i
    $modefrequency \leftarrow 0$              //highest frequency seen so far
    while $i \le n - 1$ do
        $runlength \leftarrow 1$;   $runvalue \leftarrow A[i]$
        while $i + runlength \le n - 1$ and $A[i + runlength] = runvalue$
            $runlength \leftarrow runlength + 1$
        if $runlength > modefrequency$
            $modefrequency \leftarrow runlength$;   $modevalue \leftarrow runvalue$
        $i \leftarrow i + runlength$
    return $modevalue$
```

Write an algorithm for:

Computing a mode in an array

using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

$$\begin{aligned} T(n) = T_{sort}(n) + T_{scan}(n) &\in \Theta(n \log n) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Write an algorithm for:

Searching an element in an array

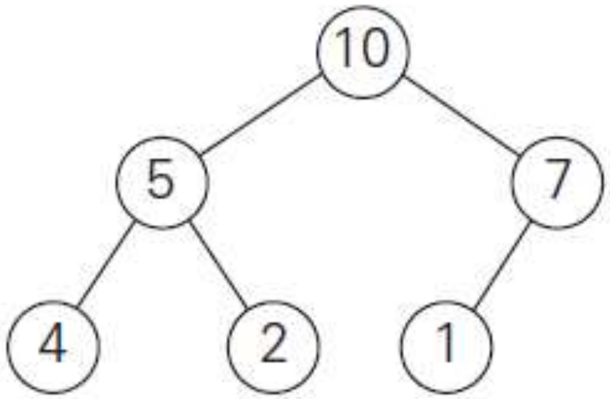
using **presorting-based** technique.

Analyze its time efficiency.

Compare with the brute force algorithm.

$$\begin{aligned} T(n) = T_{sort}(n) + T_{search}(n) &\in \Theta(n \log n) + \Theta(\log n) \\ &= \Theta(n \log n) \end{aligned}$$

Heaps:



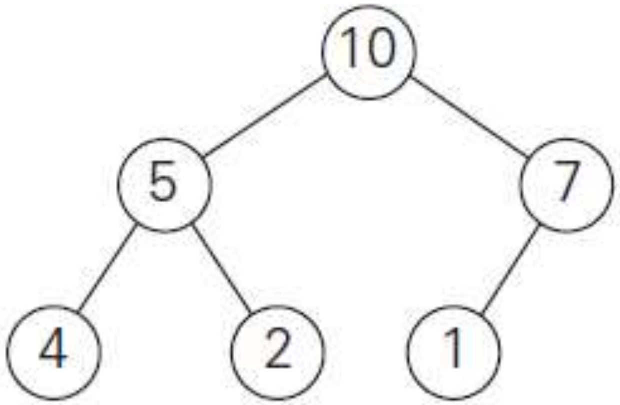
A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

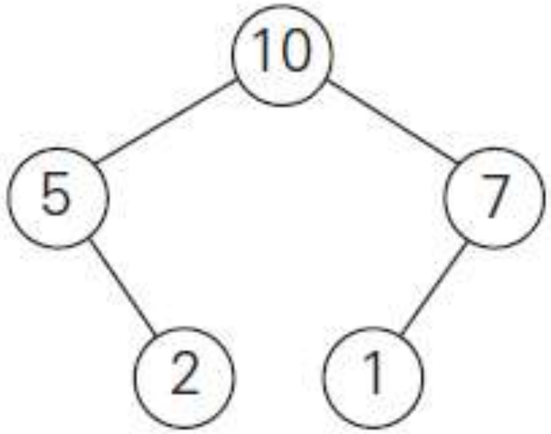
1. The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

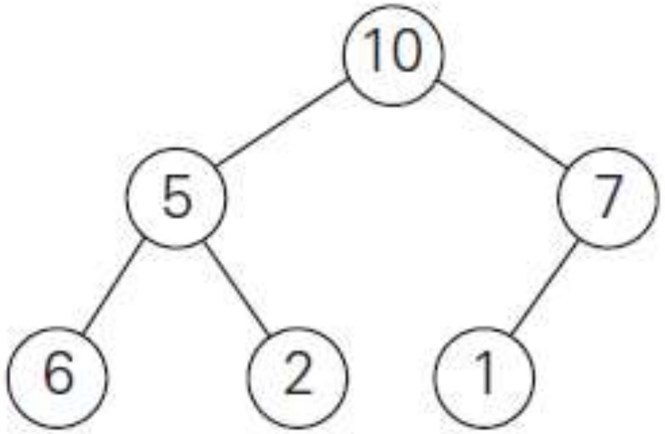
2. The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

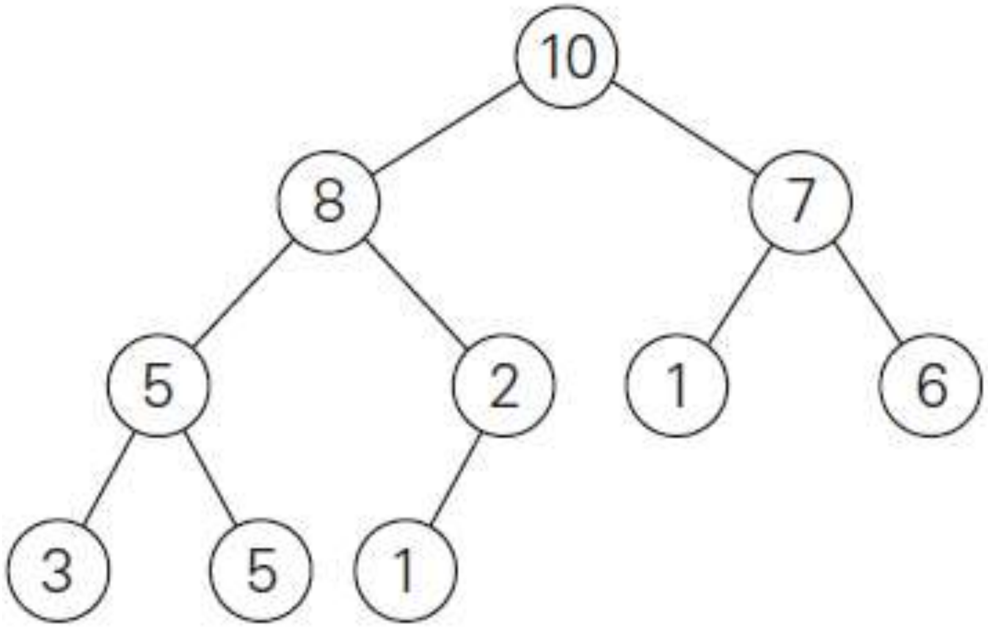
Heaps:

Which of the following are heaps?









the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

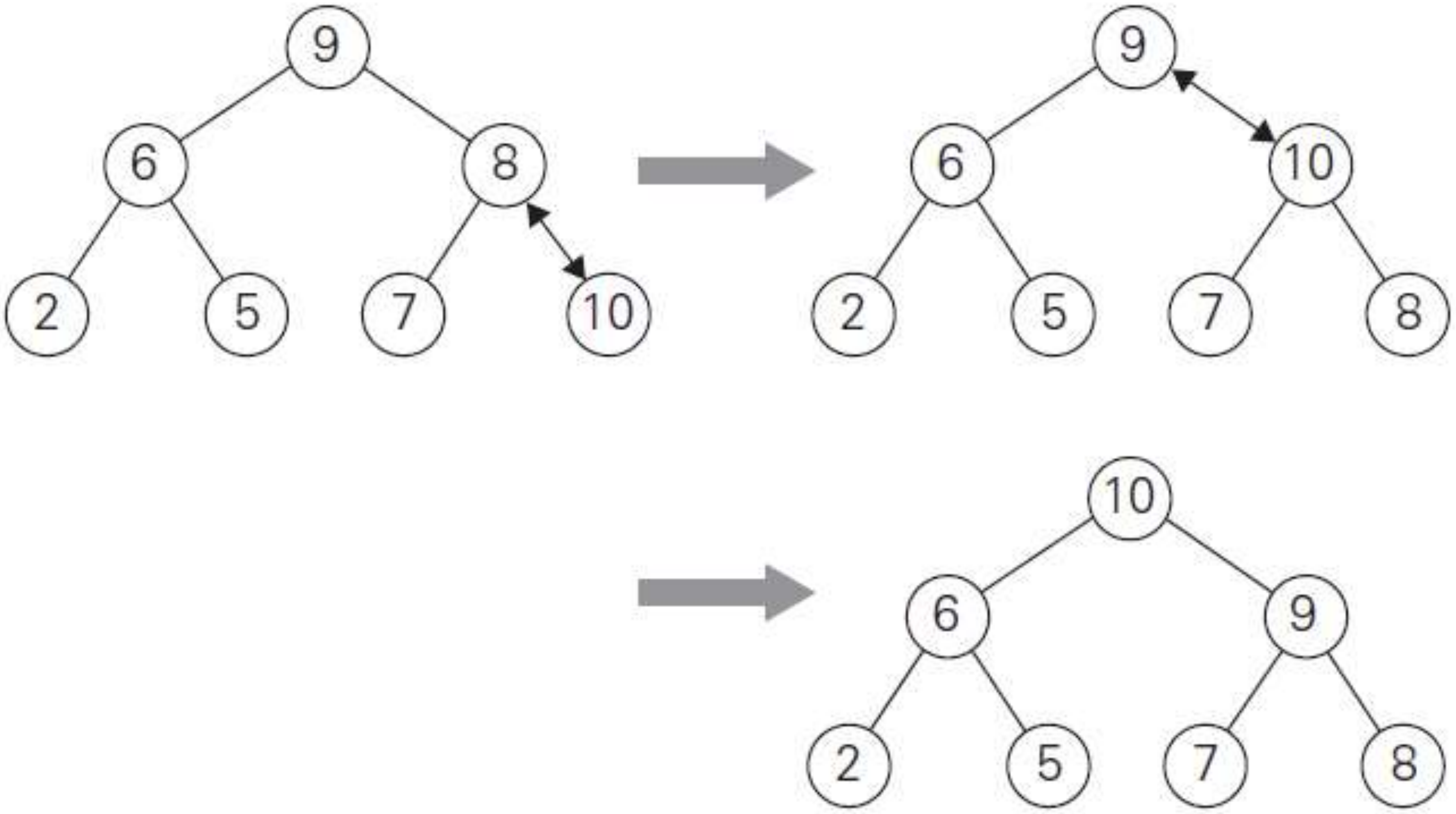
parents | leaves

A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion (BFS-way).
In such a representation (for convenience let’s store the heap’s elements in positions 1 through n of the array),

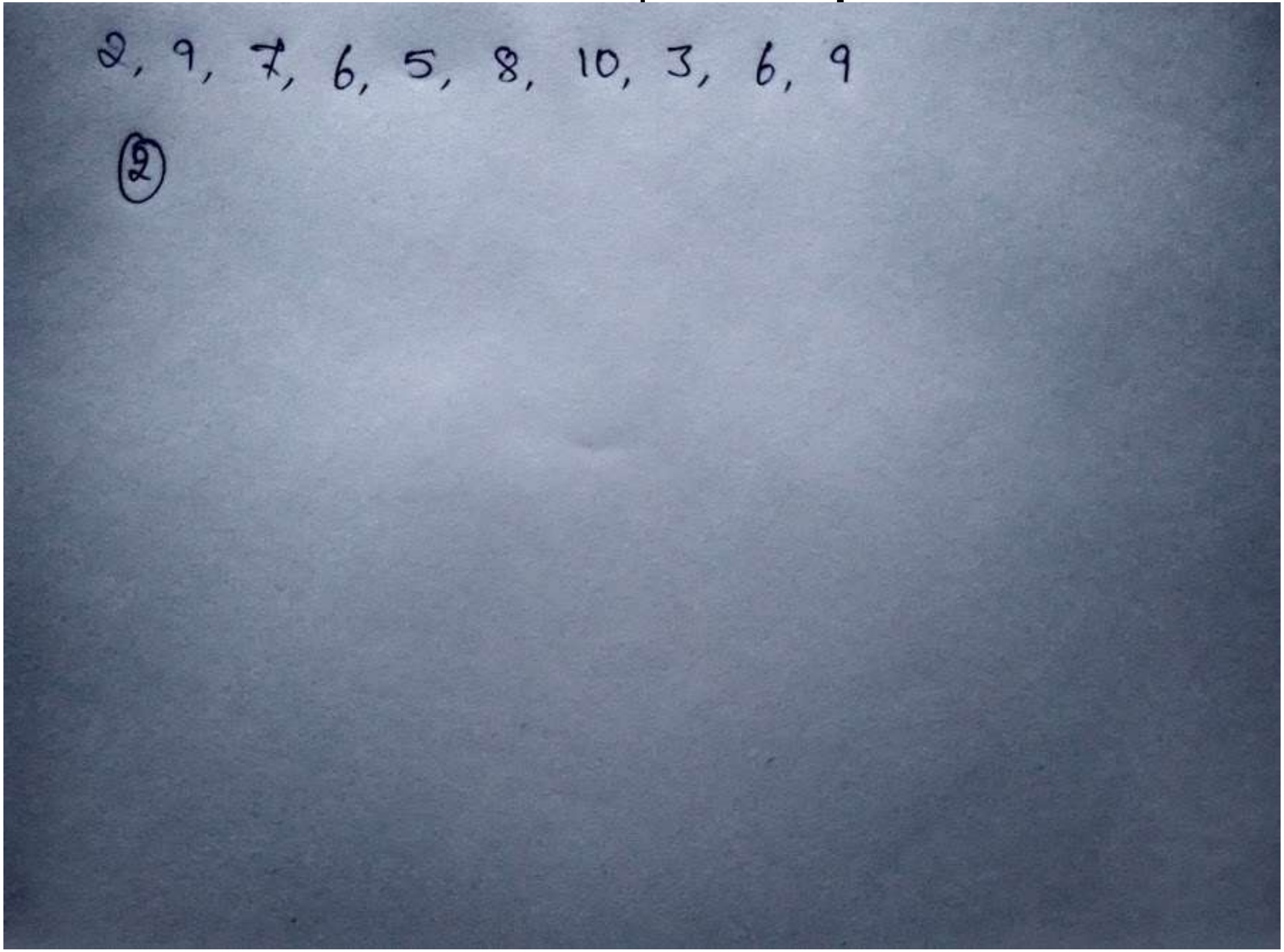
- the parental node keys will be in the first **$\lfloor n/2 \rfloor$** positions of the array, while the leaf keys will occupy the last **$\lceil n/2 \rceil$** positions.

- the children of a key in the array’s parental position **i** ($1 \le$ **i** $\le \lfloor n/2 \rfloor$) will be in positions **$2i$** and **$2i+1$**, and, correspondingly, the parent of a key in position **j** ($2 \le$ **j** $\le n$) will be in position **$\lfloor j/2 \rfloor$**.

Inserting a new element in the heap:
Add new element **‘10’** to the existing heap.



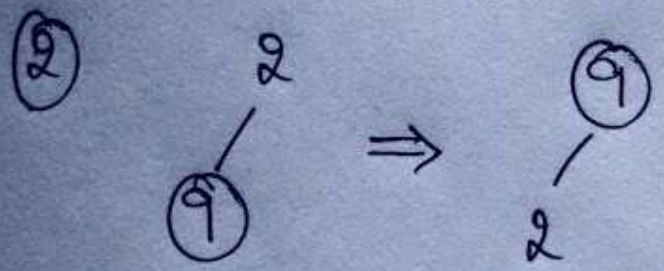
Construction of a heap from **top-down**:



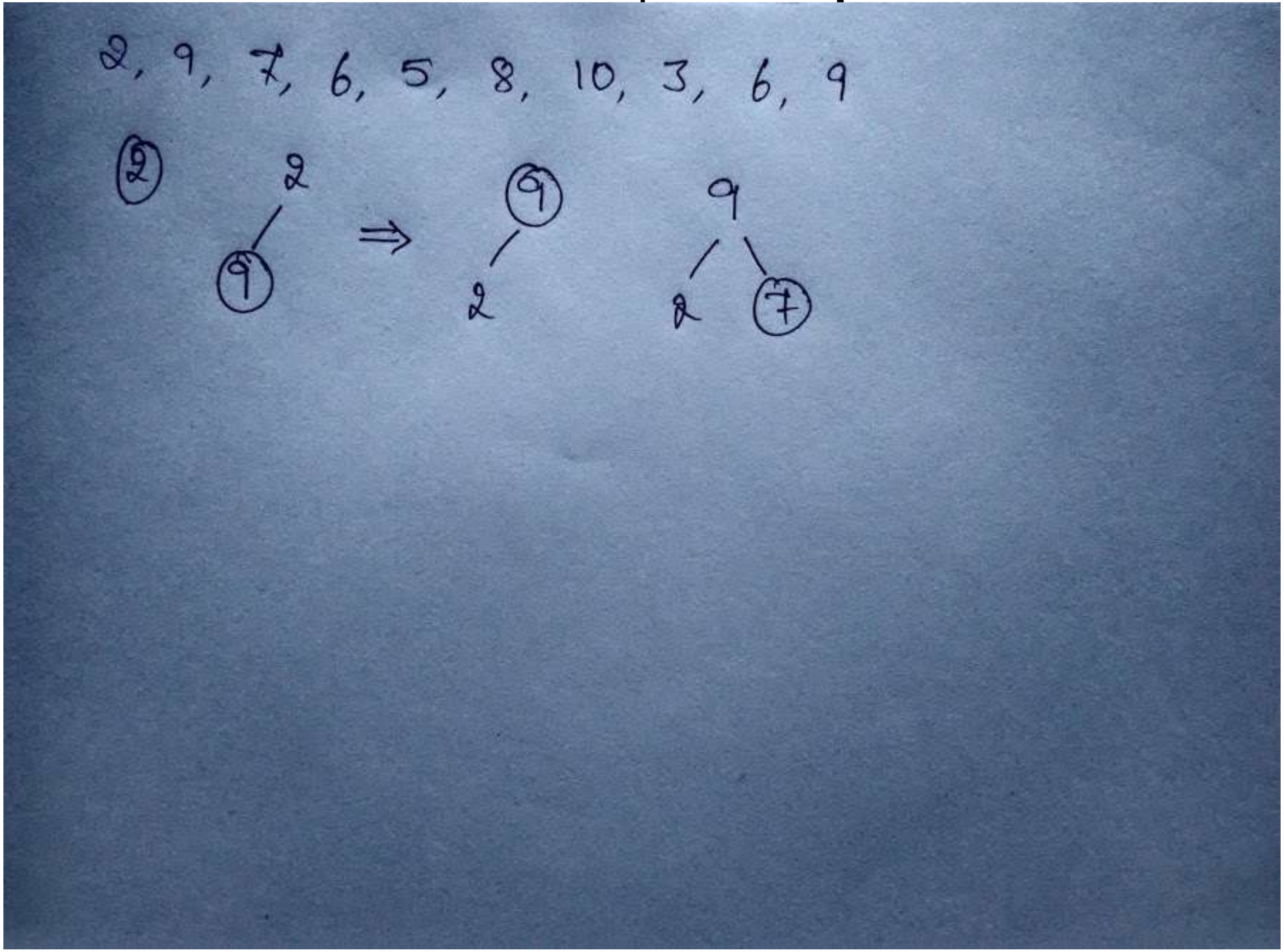
2, 9, 7, 6, 5, 8, 10, 3, 6, 9

Construction of a heap from **top-down**:

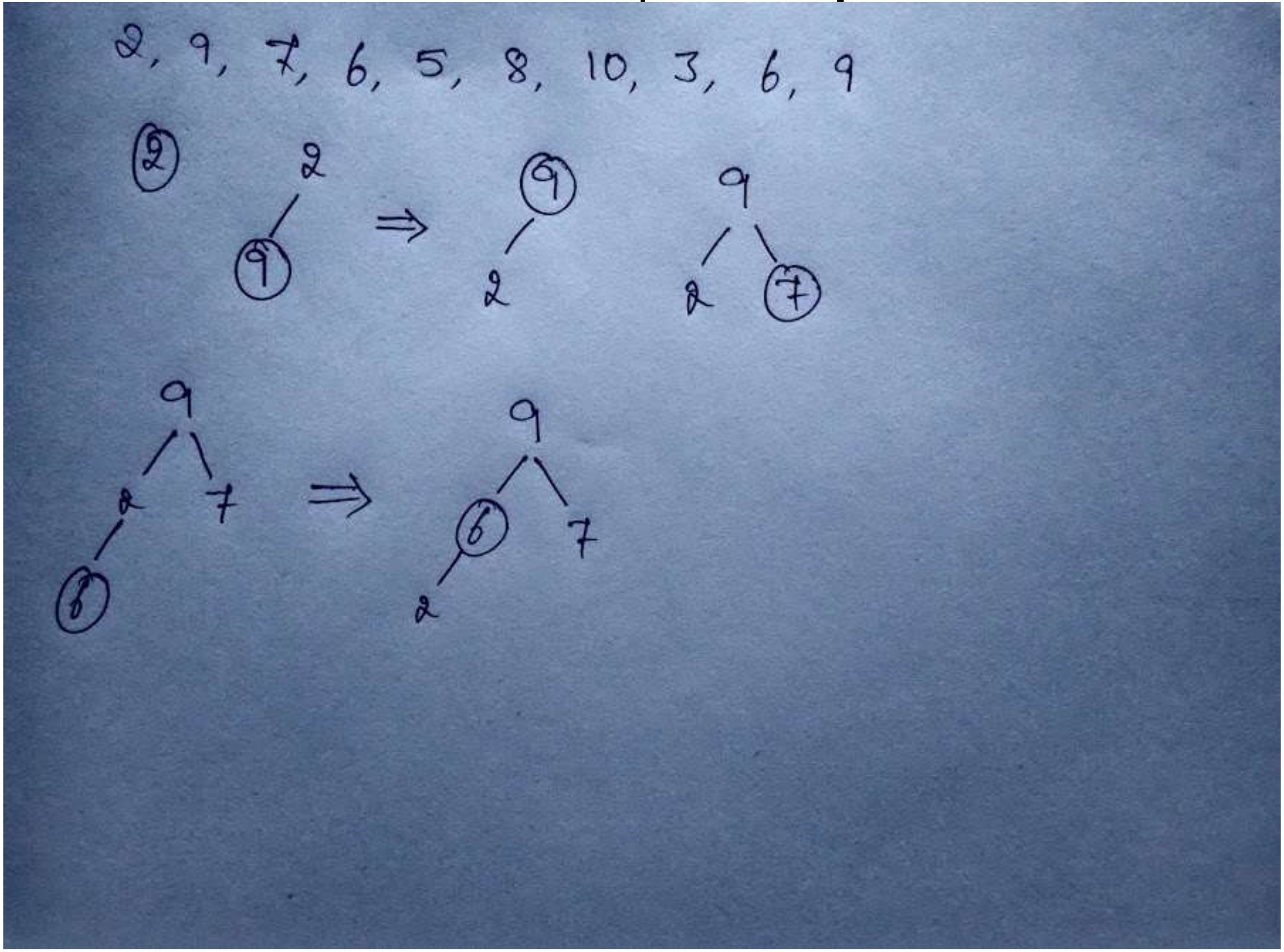
2, 9, 7, 6, 5, 8, 10, 3, 6, 9



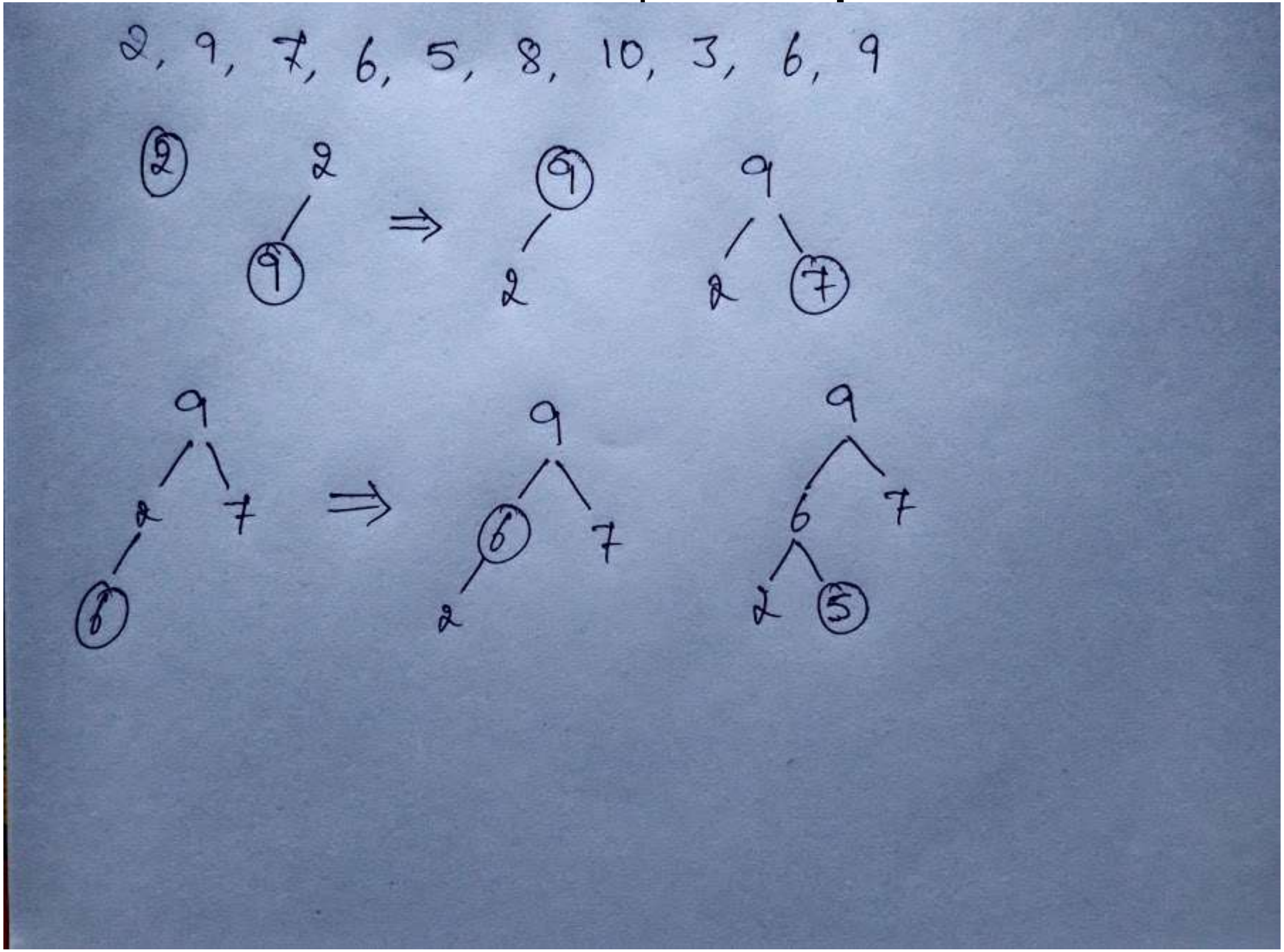
Construction of a heap from **top-down**:



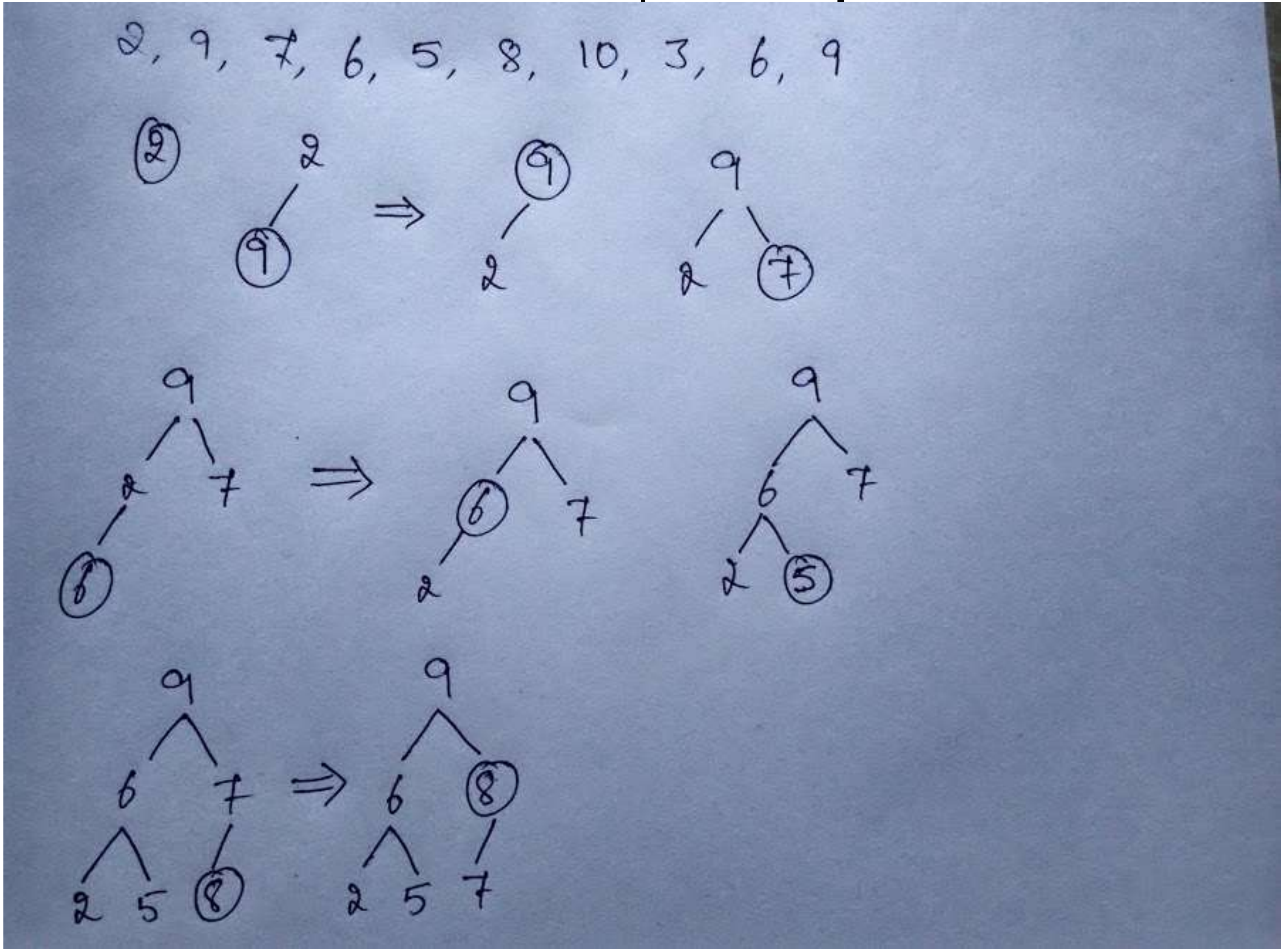
Construction of a heap from **top-down**:



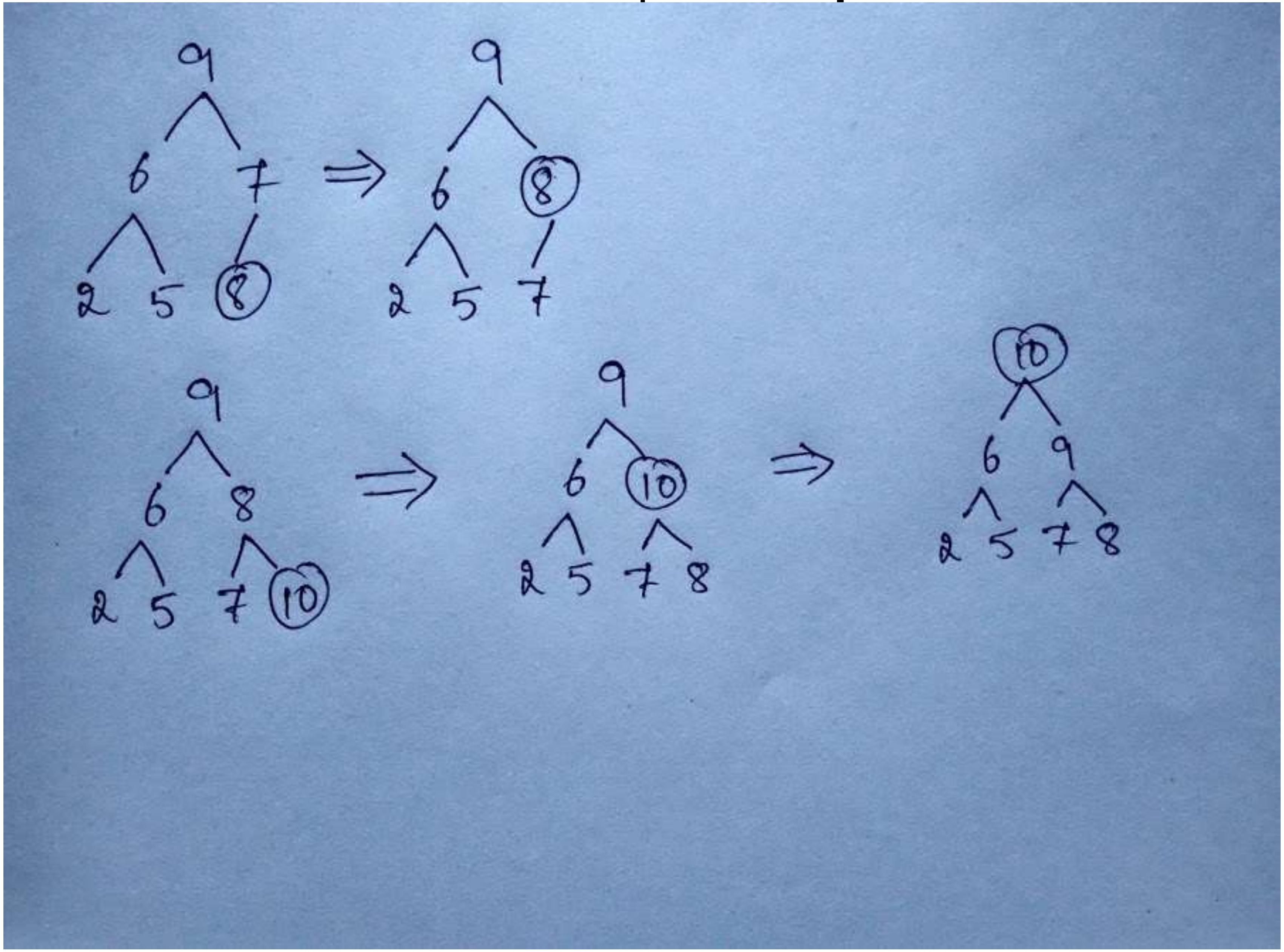
Construction of a heap from **top-down**:



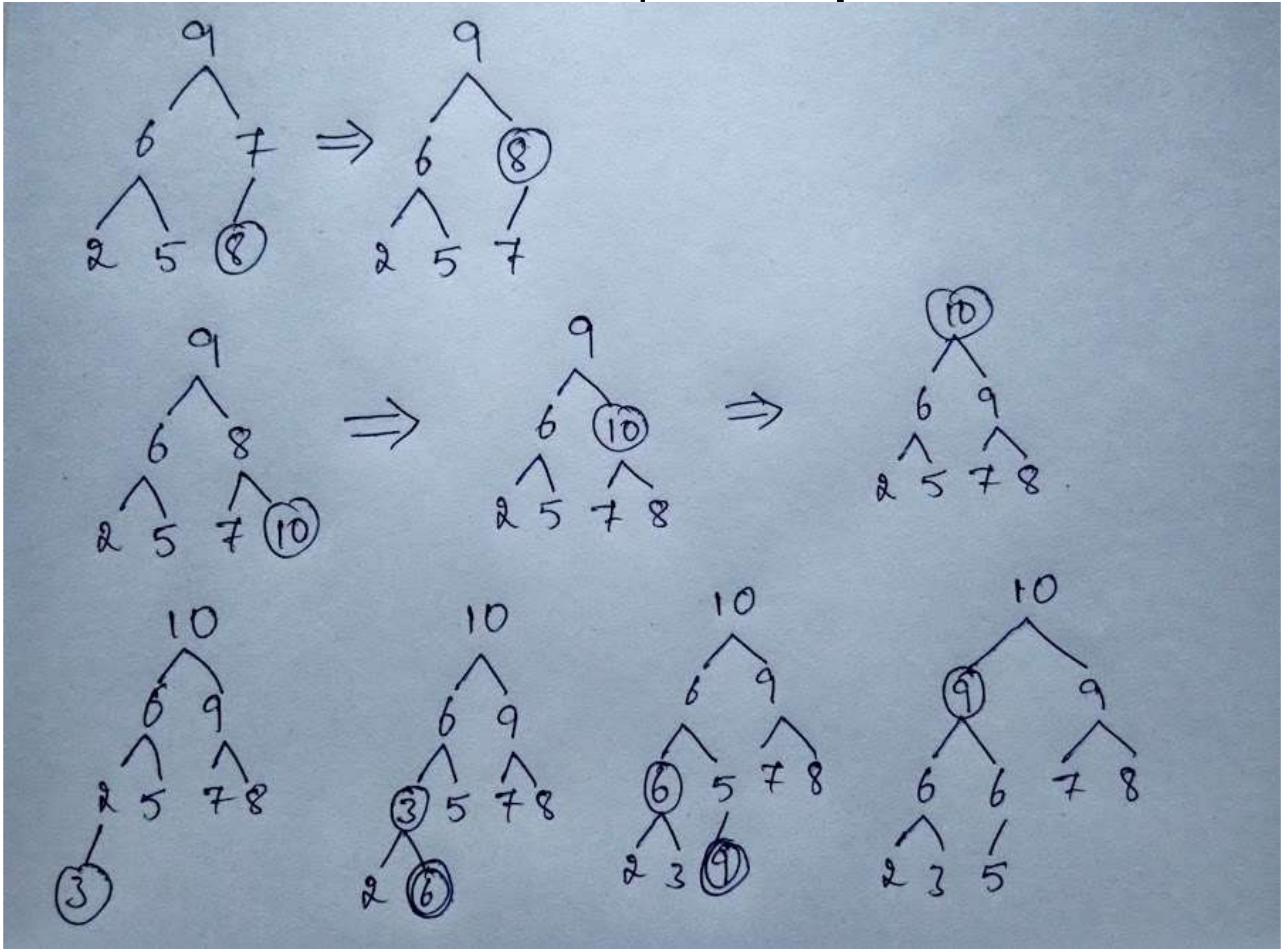
Construction of a heap from **top-down**:



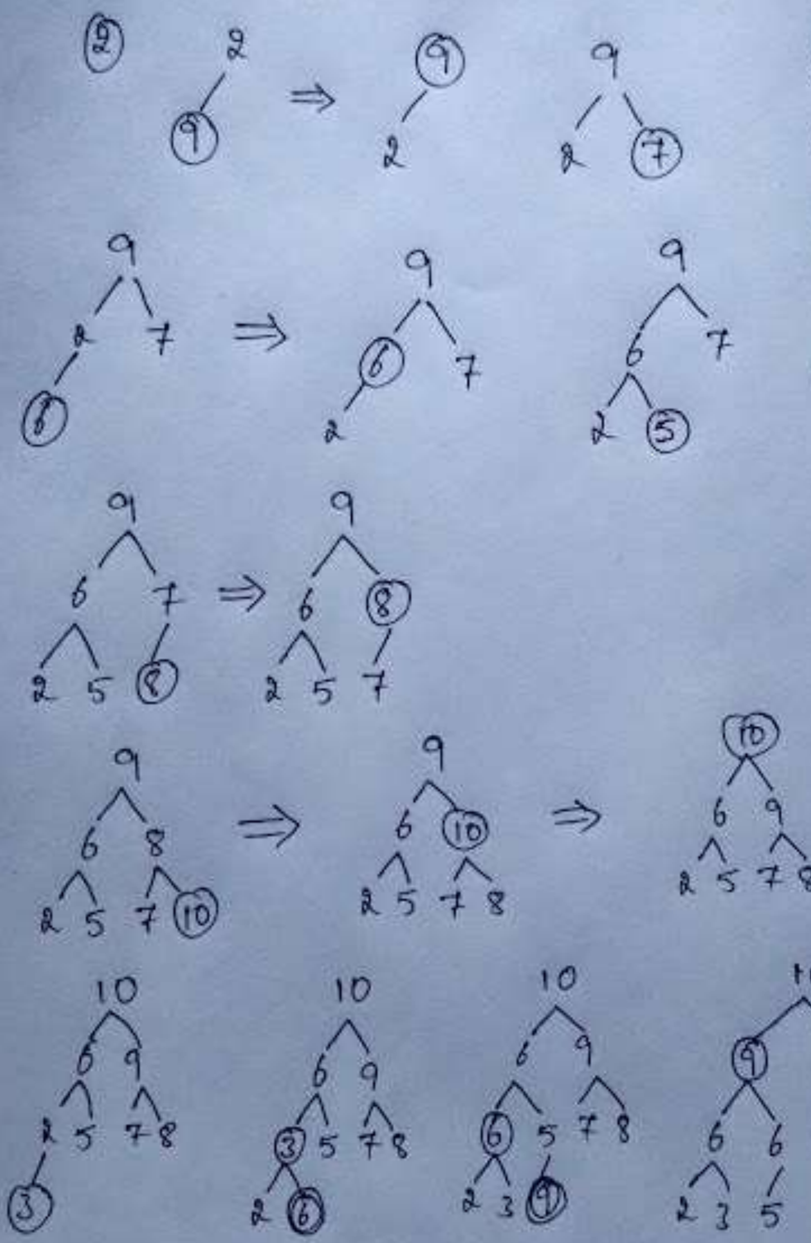
Construction of a heap from **top-down**:



Construction of a heap from **top-down**:

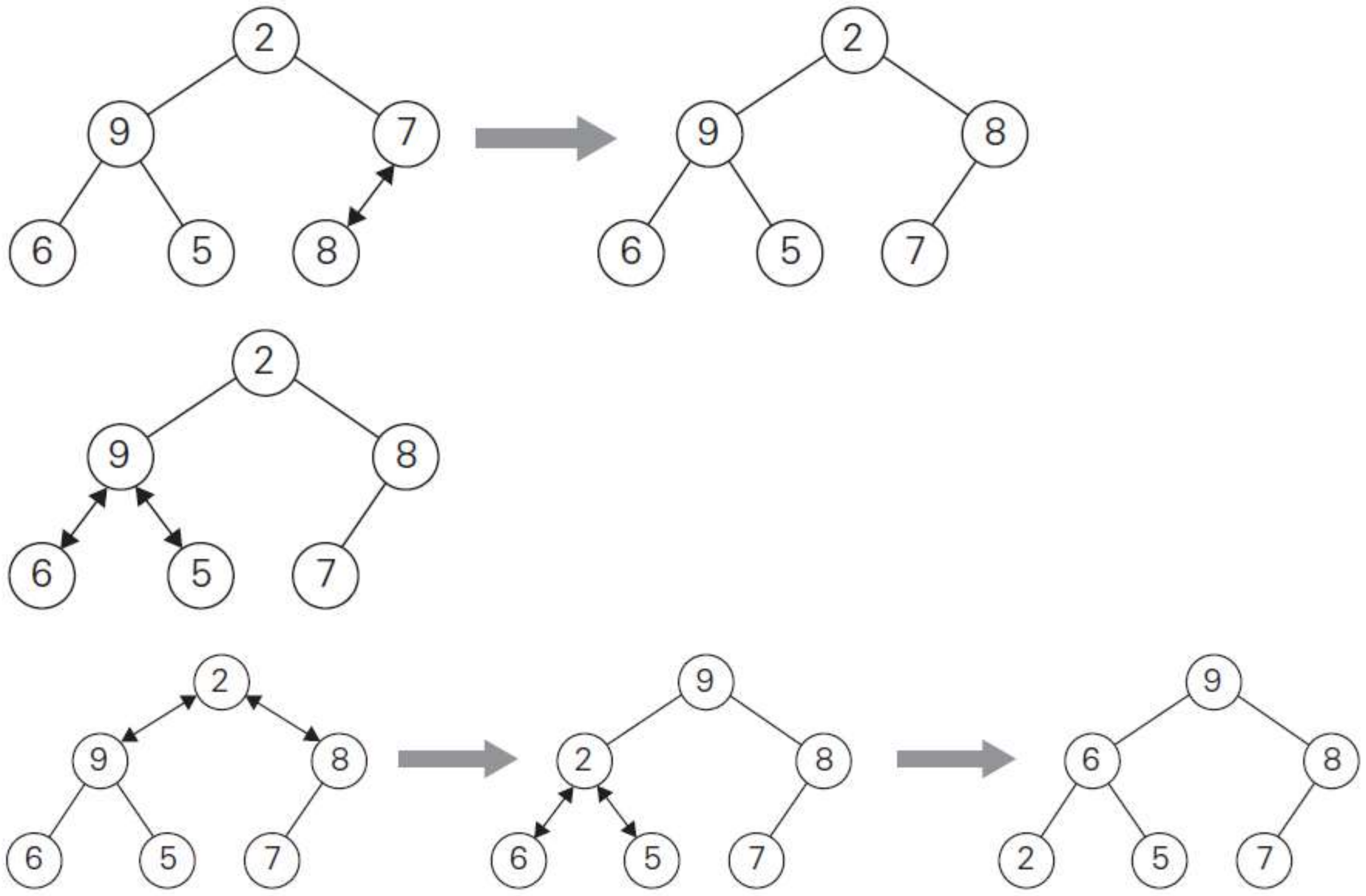


2, 9, 7, 6, 5, 8, 10, 3, 6, 9

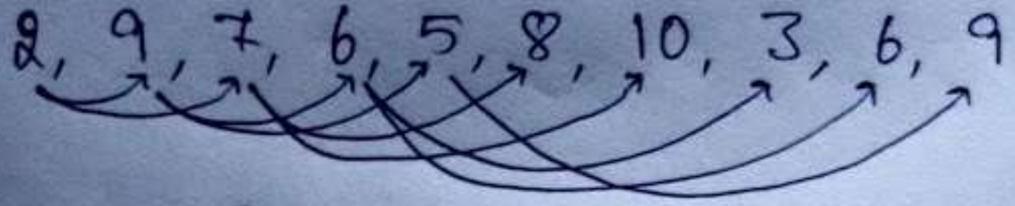


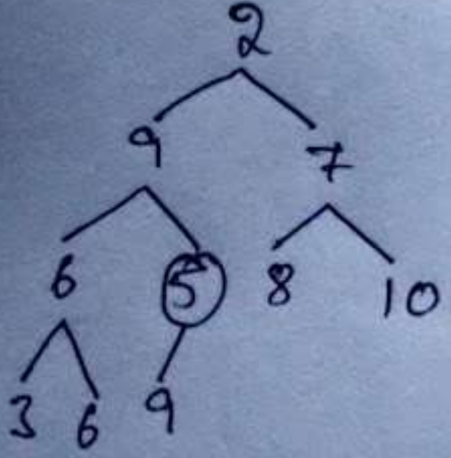
Heap Construction by top-down approach

Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8.

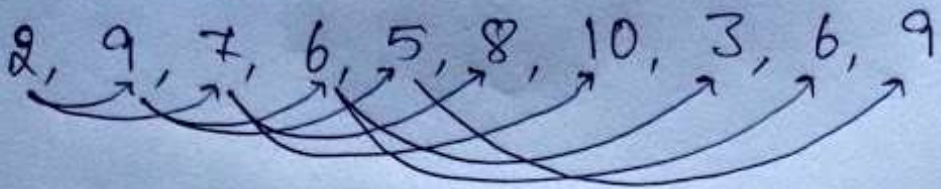


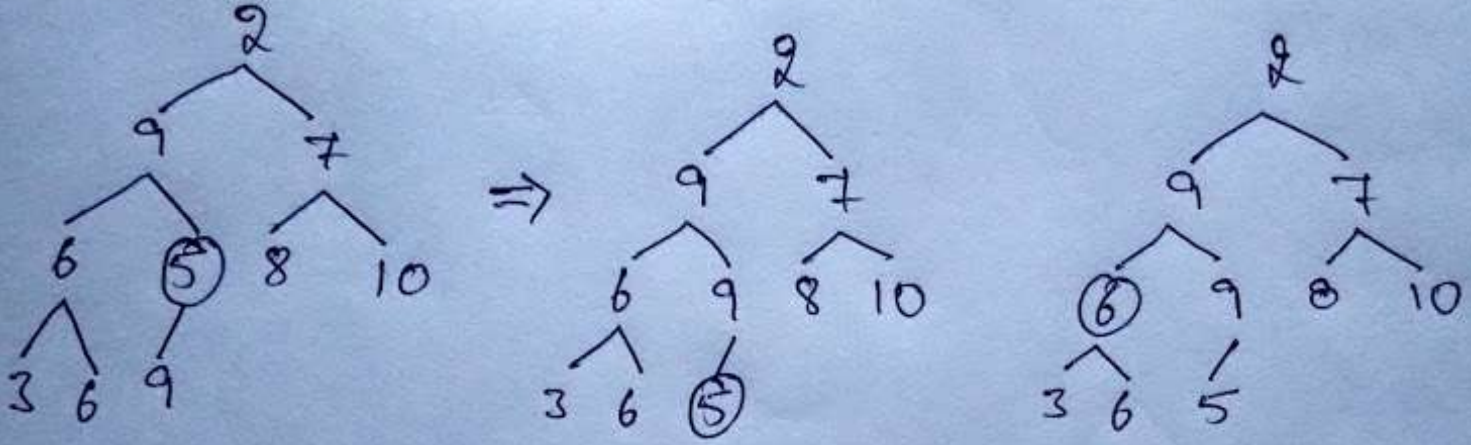
Heap Construction by bottom-up approach.

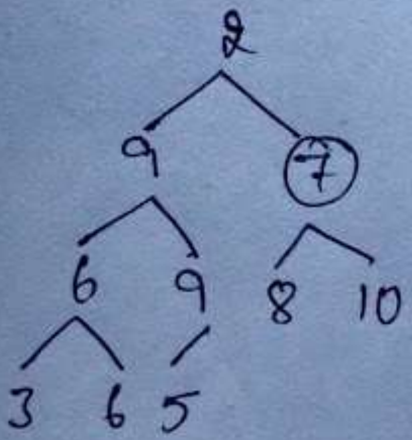




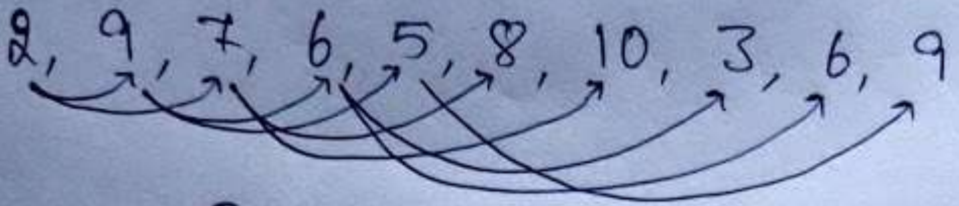
Heap Construction by bottom-up approach.

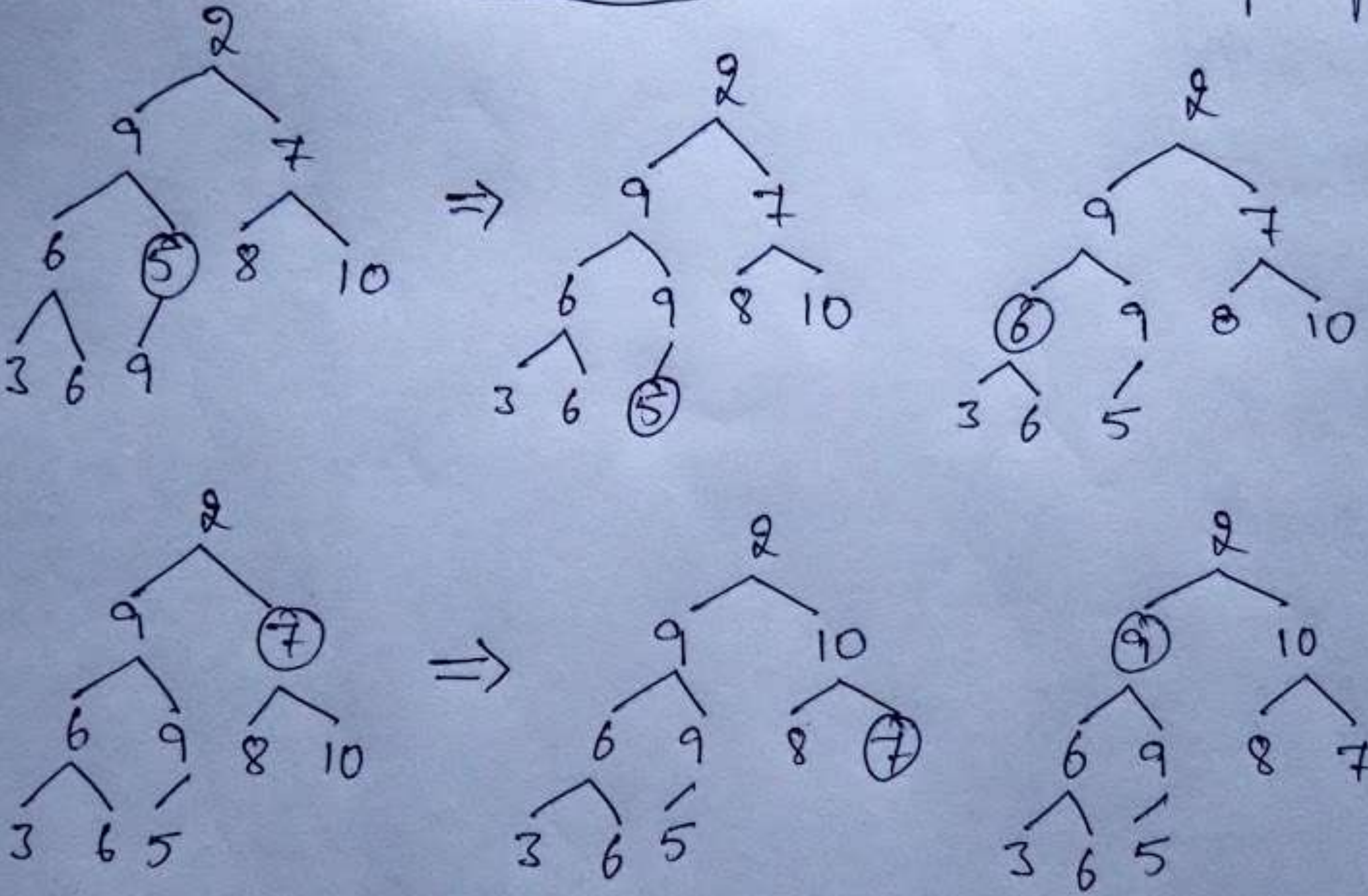




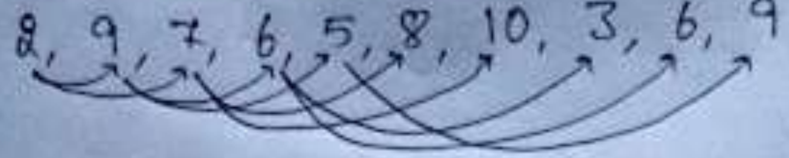


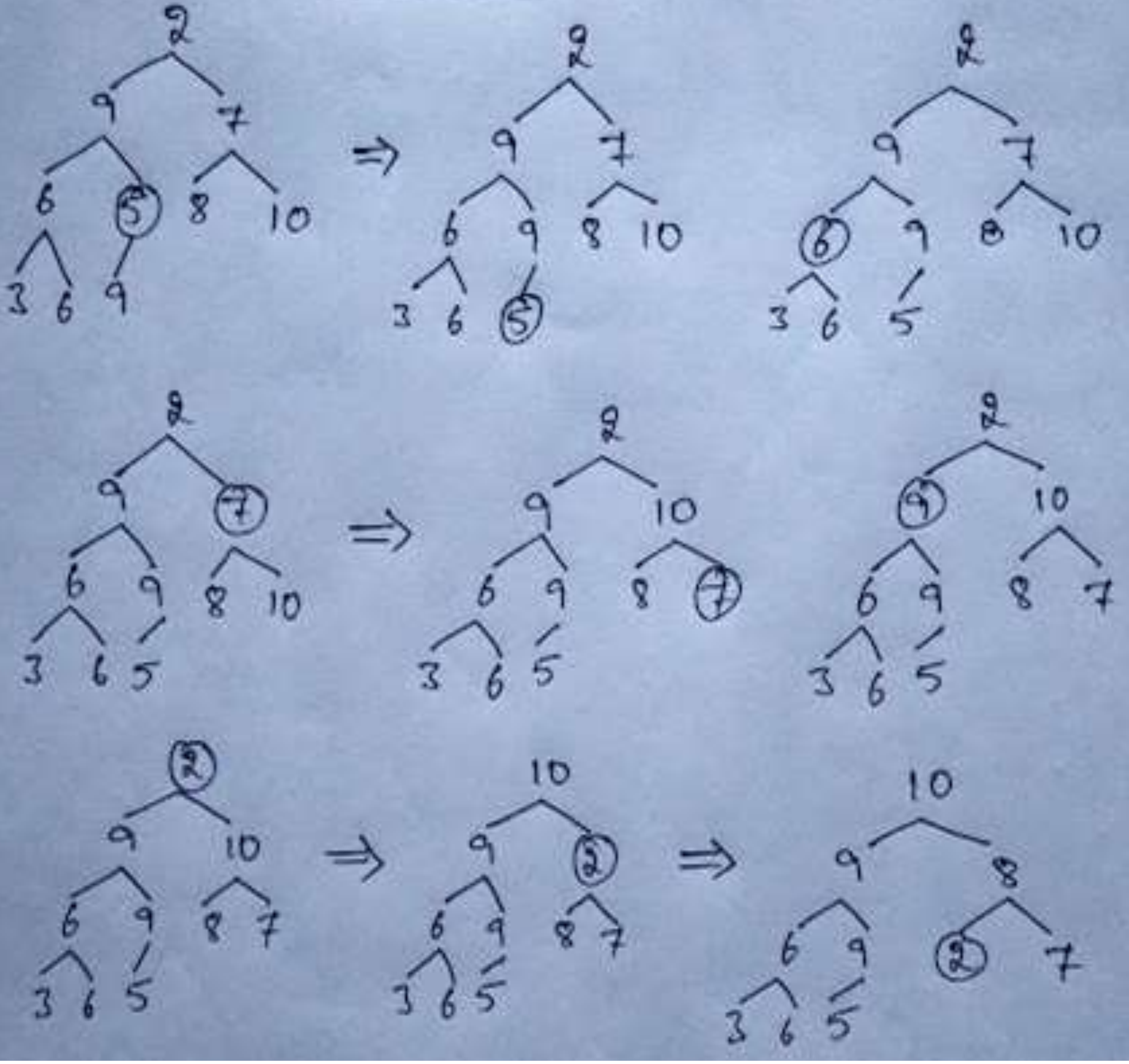
Heap Construction by bottom-up approach.





Heap Construction by bottom-up approach.



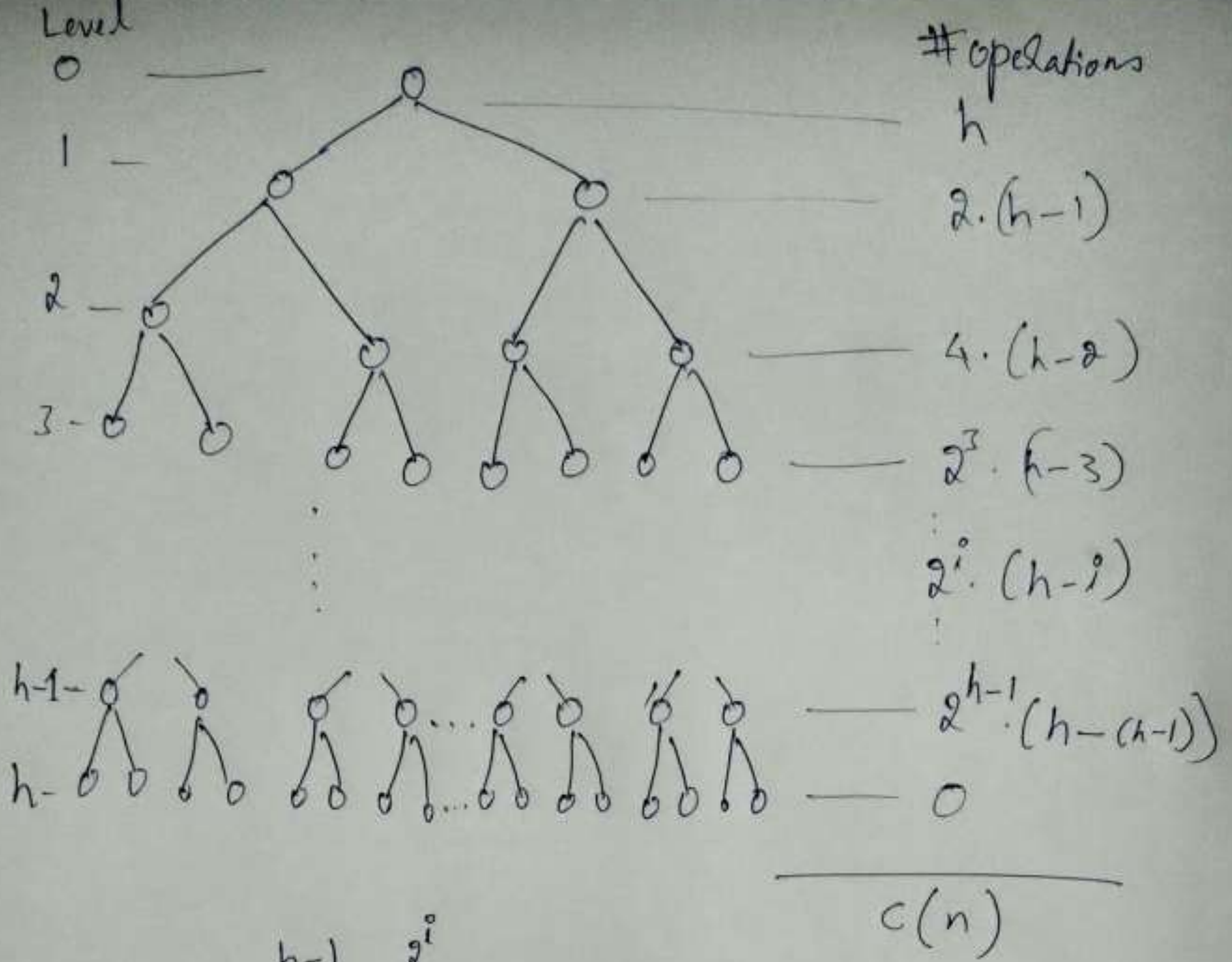


```
HeapBottomUp(H[1..n])
    if(n ≤ 1) return
    for i ← ⌊n/2⌋ downto 1 do
        Heapify(H, i)

//For the subtree rooted at k, it sifts down H[k]
//as much as possible until it becomes a heap.
Heapify(H[1..n], k)
    if(2*k > n) return  //if H[k] is a leaf
    j ← 2*k  //j points to left child of H[k]
    if(j+1 ≤ n)  //if there exists a right child of H[k]
        if(H[j+1] > H[j]) j ← j+1
    if(H[j] > H[k])  //if greater child is greater than H[k]
        H[j] ↔ H[k]
        Heapify(H, j)  //Heapify the subtree rooted at j
```

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 do
    $k \leftarrow i$;   $v \leftarrow H[k]$
    $heap \leftarrow$ false
    while not $heap$ and $2 * k \le n$ do
        $j \leftarrow 2 * k$
        if $j < n$   //there are two children
            if $H[j] < H[j+1]$  $j \leftarrow j + 1$
        if $v \ge H[j]$
            $heap \leftarrow$ true
        else $H[k] \leftarrow H[j]$;   $k \leftarrow j$
    $H[k] \leftarrow v$
```



$$c(n) = \sum_{i=0}^{h-1} \sum_{j=1}^{2^i} (h-i)$$

$$c(n) = \sum_{i=0}^{h-1} 2^i (h-i)$$

$$C(n) = \sum_{i=0}^{h-1} 2^i (h-i)$$

$$= \left(h \sum_{i=0}^{h-1} 2^i\right) - \left(\sum_{i=0}^{h-1} i \cdot 2^i\right)$$

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

$$\because \sum_{i=1}^{n} i \cdot 2^i = (n-1)2^{n+1} + 2$$

$$C(n) = \cancel{h \cdot 2^h} - h - \cancel{h \cdot 2^h} + 2^{h+1} - 2$$

$$= 2^{h+1} - h - 2$$

$$h = \lfloor \log_2 n \rfloor$$

$$C(n) = 2^{1 + \lfloor \log_2 n \rfloor} - h - 2 \in \Theta(n)$$

$$\sum_{i=1}^{n} i2^{i} = (n-1)2^{n+1}+2 \quad ?$$

$$= 1\cdot 2^1 + 2\cdot 2^2 + 3\cdot 2^3 + 4\cdot 2^4 + \dots + n\cdot 2^n$$

$$= \left.\begin{array}{rrrrrrl}
2^1 + & 2^2 + & 2^3 + & 2^4 + & \cdots & + & 2^n \\
+ & 2^2 + & 2^3 + & 2^4 + & \cdots & + & 2^n \\
 & + & 2^3 + & 2^4 + & \cdots & + & 2^n \\
 & \vdots & & \vdots & & & \\
 & & & + & 2^{n-1} & + & 2^n \\
 & & & & & + & 2^n
\end{array}\right\} n \text{ linii}$$

$$= \left(2^{n+1}-2^1\right) + \left(2^{n+1}-2^2\right) + \left(2^{n+1}-2^3\right) + \dots + \left(2^{n+1}-2^{n-1}\right) + \left(2^{n+1}-2^n\right)$$

$$= n\cdot 2^{n+1} - \left(2^1+2^2+\dots+2^n\right)$$

$$= n\cdot 2^{n+1} - \left(2^{n+1}-2\right) = \boxed{(n-1)2^{n+1}+2}$$

Efficiency of construction of heap from bottom-up:
Let h be the height of the tree.
Two key comparisons at level of trickle down of an element.

$$h = \lfloor \log_2 n \rfloor$$

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i)$$

$$= \sum_{i=0}^{h-1} 2(h-i)2^i$$

$\cong$ 2n $\in$ Θ(n)

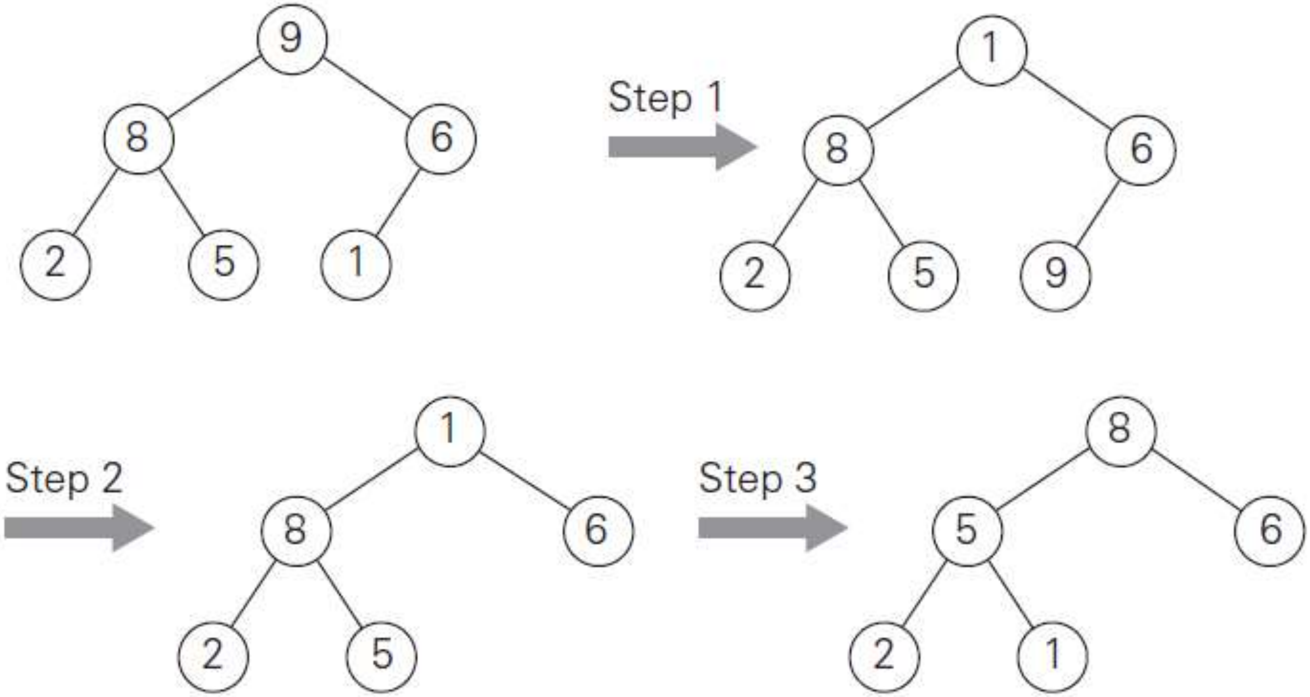
Heapsort discovered by J. W. J. Williams

This is a two-stage algorithm

Stage 1 (heap construction):
Construct a heap for a given array.

Stage 2 (maximum deletions):
Apply the root-deletion operation
$n - 1$ times to the remaining heap.

Maximum
Key Deletion
from a heap



1. Exchange the root’s key with the last key K of the heap.

1. Decrease the heap’s size by 1.

1. “Heapify” the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm.

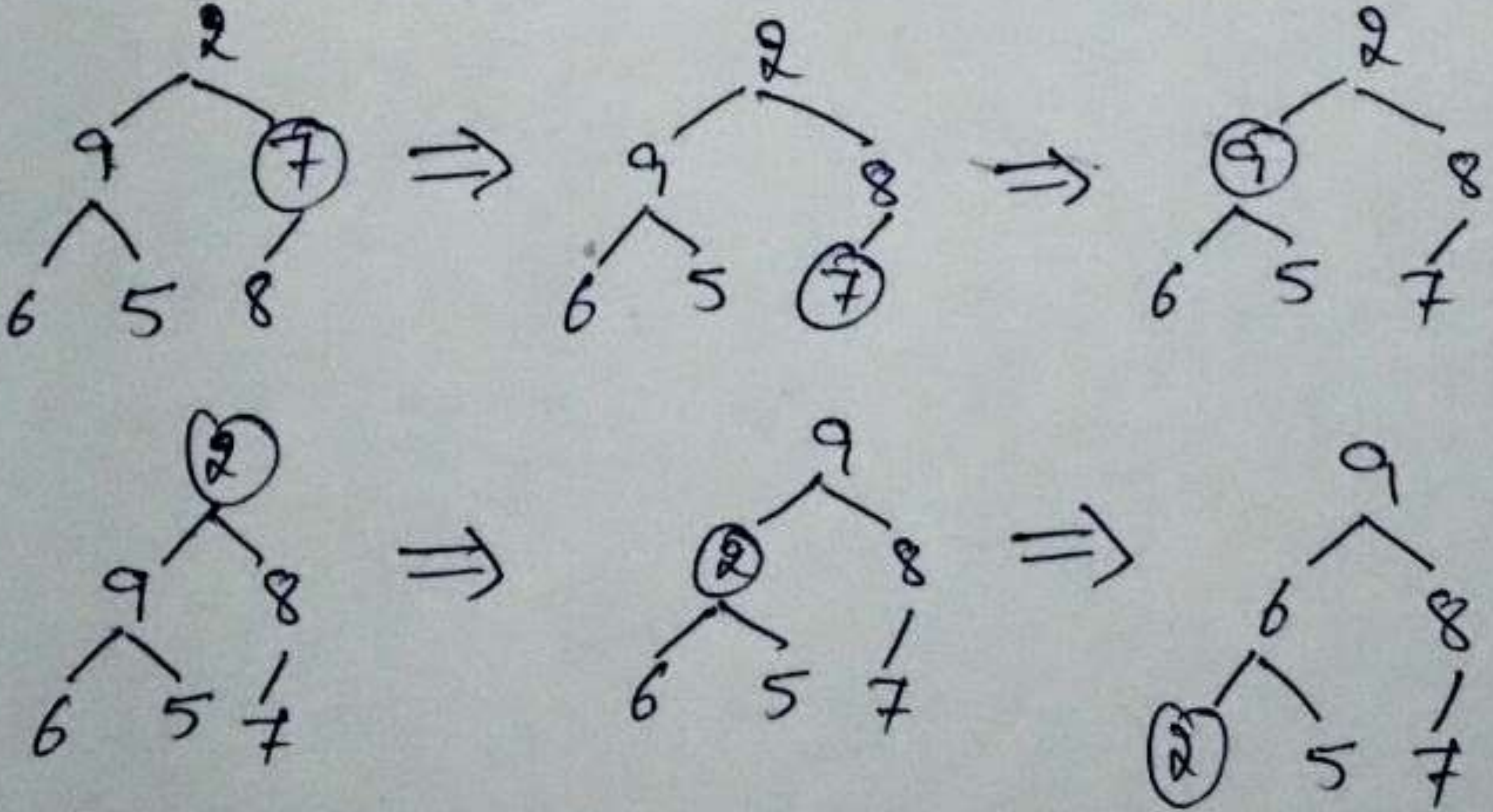
```
HeapSort(H[1..n])
    HeapBottomUp(H[1..n])  //Construct heap
    for i ← n downto 2 do
    H[1] ↔ H[i]  //H[1] has the max element.
    Heapify(H[1..i-1], 1)  //Sift down H[1]

HeapBottomUp(H[1..n])
    if(n ≤ 1) return
    for i ← ⌊n/2⌋ downto 1 do
        Heapify(H, i)

Heapify(H[1..n], k)
    if(2*k > n) return  //if H[k] is a leaf
    j ← 2*k  //j points to left child of H[k]
    if(j+1 ≤ n)  //if there exists a right child of H[k]
        if(H[j+1] > H[j]) j ← j+1
    if(H[j] > H[k])  //if greater child is greater than H[k]
        H[j] ↔ H[k]
        Heapify(H, j)  //Heapify the subtree rooted at j
```

2	9	7	6	5	8

Unsorted array

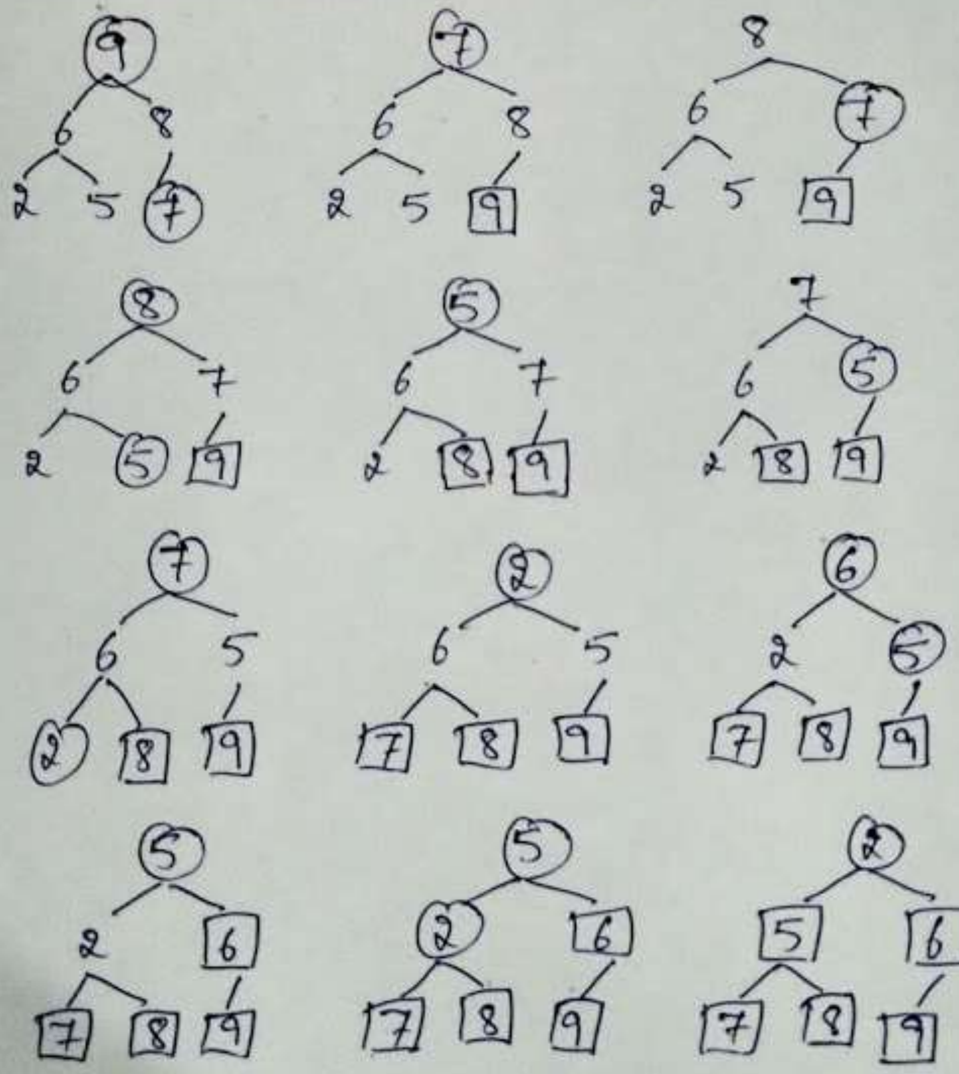


9	6	8	2	5	7

Heap

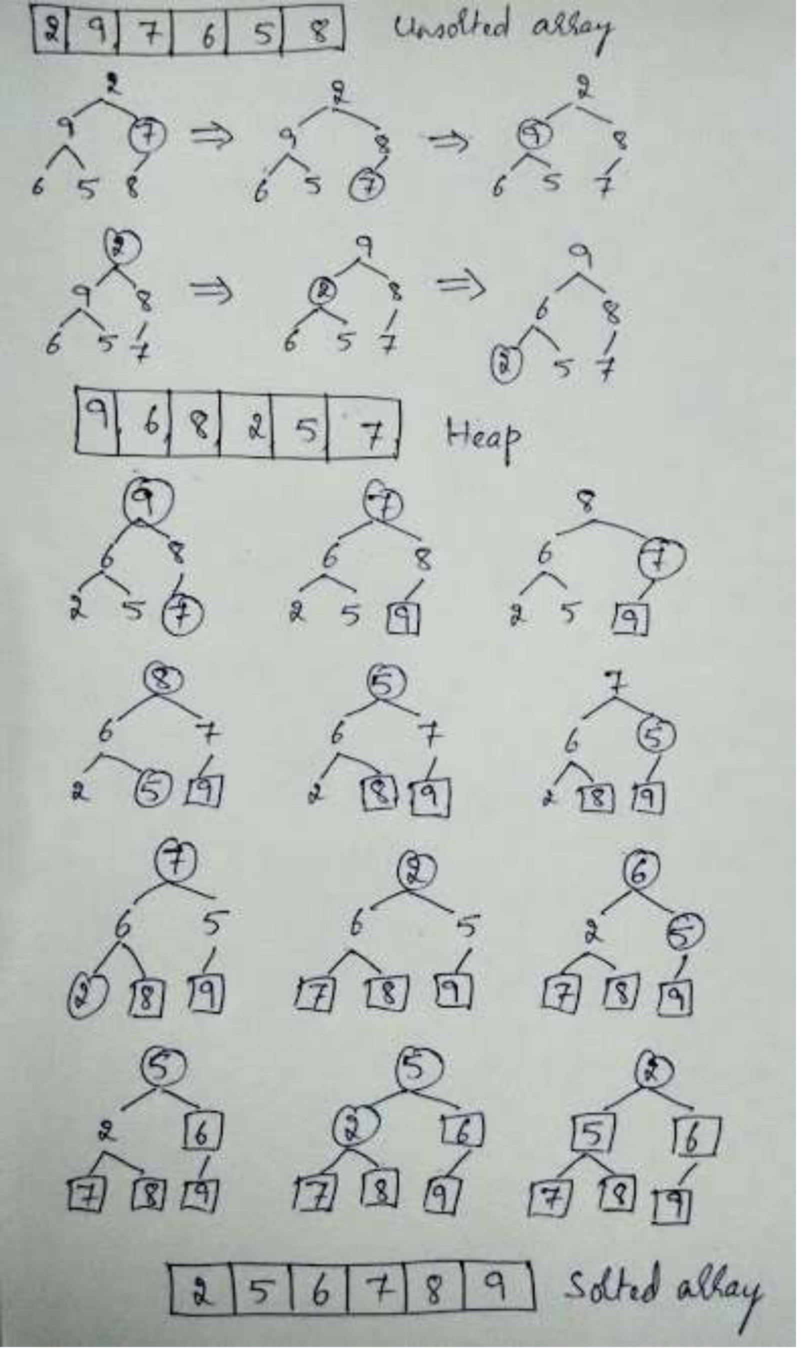
9	6	8	2	5	7

Heap



2	5	6	7	8	9

Sorted array



Analysis of Heapsort:

$T_{Heapsort}(n) = T_{Heap}(n) + T_{Sort}(n)$

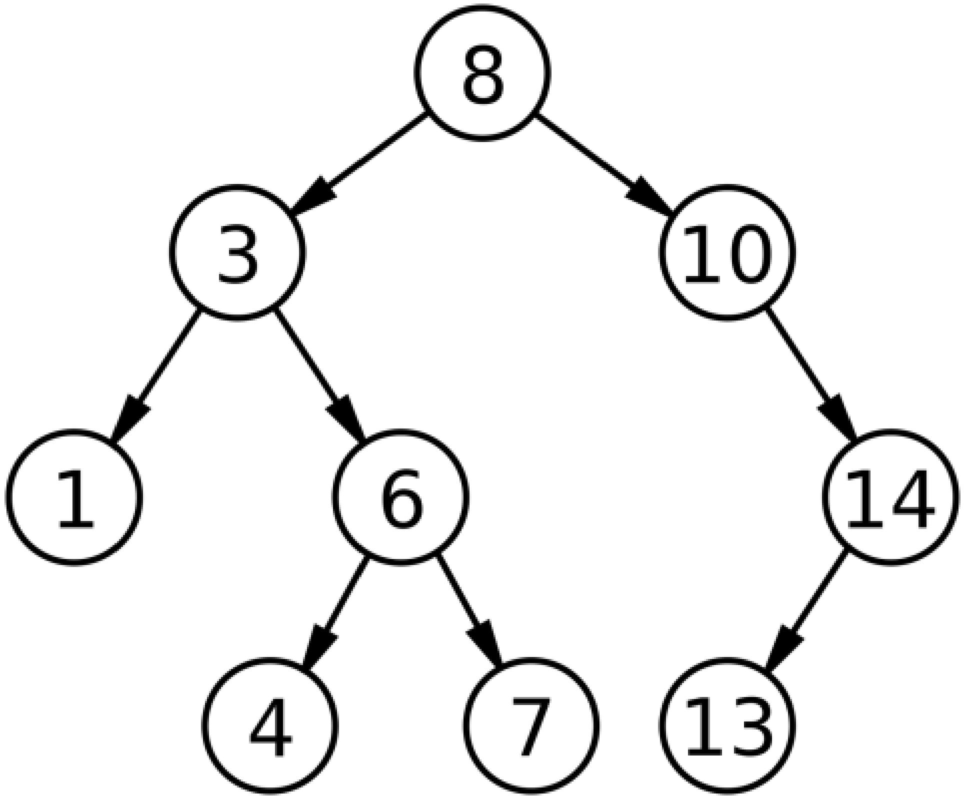
$T_{Heapsort}(n) \in \max\{\theta(n), \theta(n \log n)\}$

$T_{Heapsort}(n) \in \theta(n \log n)$

Binary Trees

Binary Search Trees (BST)

What do we “conquer” by transforming a Binary Tree into a BST?



[Optional]

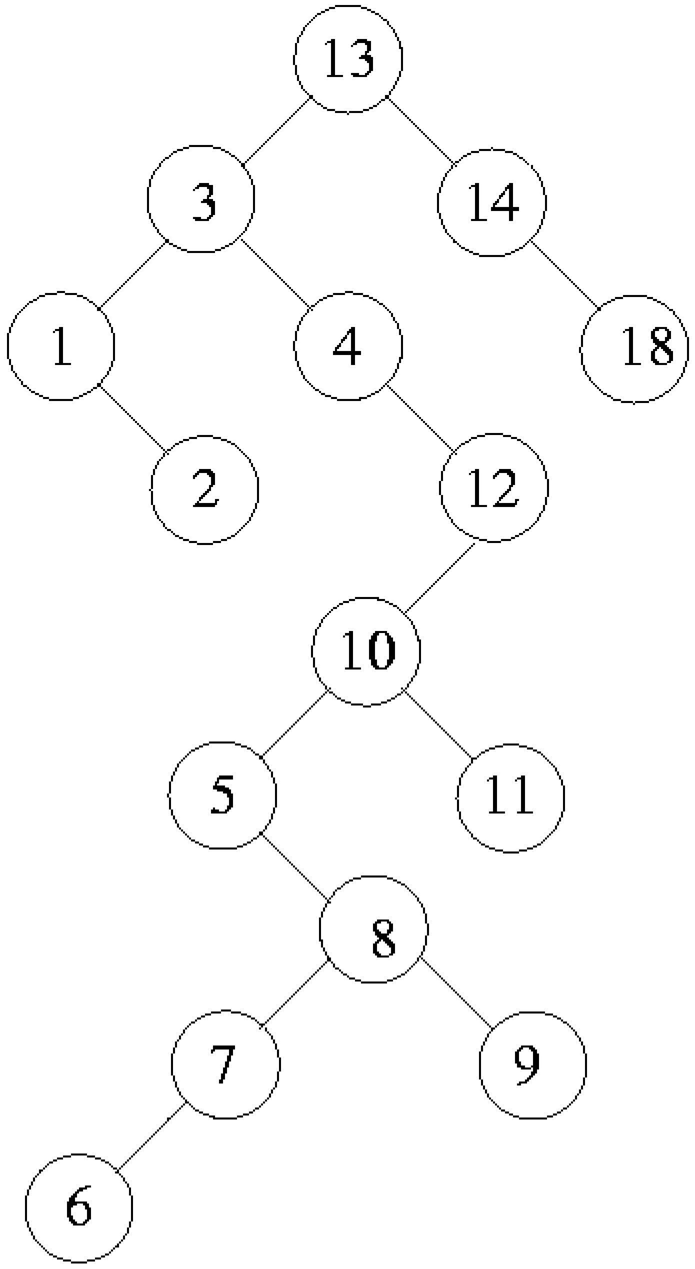
- Boolean isBST(BinaryTree t);
- BST BT2BST(BinaryTree t); //do it in-place

Binary Trees

Binary Search Trees (BST)

What do we “conquer” by transforming a Binary Tree into a BST?
Search!

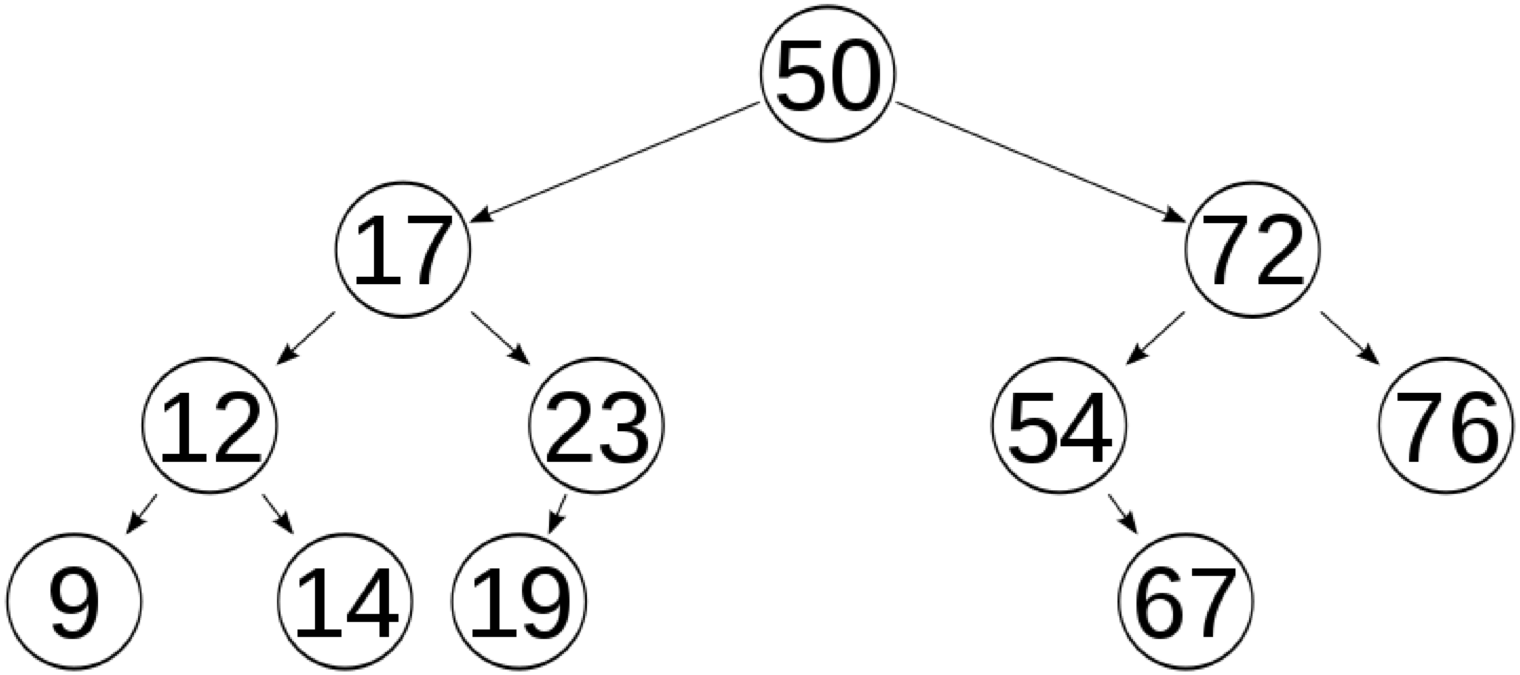
Time complexity of
- Inserting an element into a BST:
- Searching for an element in a BST:
    - Average case:
    - Worst case:



Balanced Binary Search Trees:

Time complexity of worst-case of search in a BST is **O(n)**

How can we keep the BST balanced so that the worst-case is just **O(log n)** because the height of the tree is limited to **O(log n)**?



Balanced Binary Search Trees:

Time complexity of worst-case of search in a BST is **O(n)**

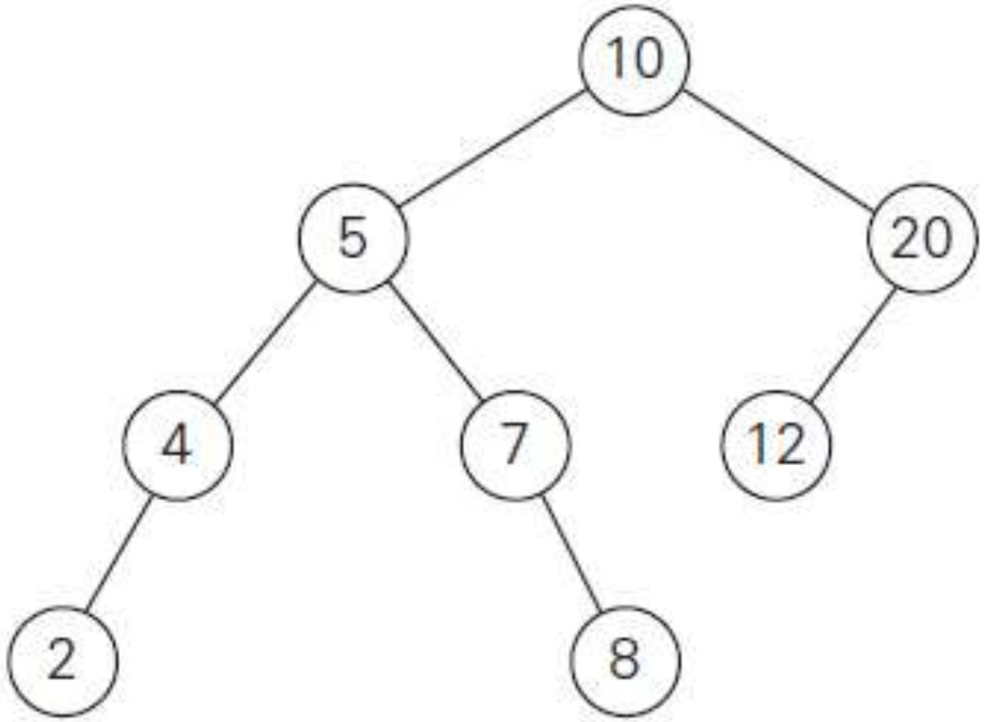
How can we keep the BST balanced so that the worst-case is just **O(log n)** because the height of the tree is limited to **O(log n)**?

1. AVL Trees
2. Red-Black Trees
3. Splay Trees
4. 2-3 Trees
    a. Not exactly a BST.
       It’s not even a Binary Tree.
       It’s a **Balanced Search Tree**.

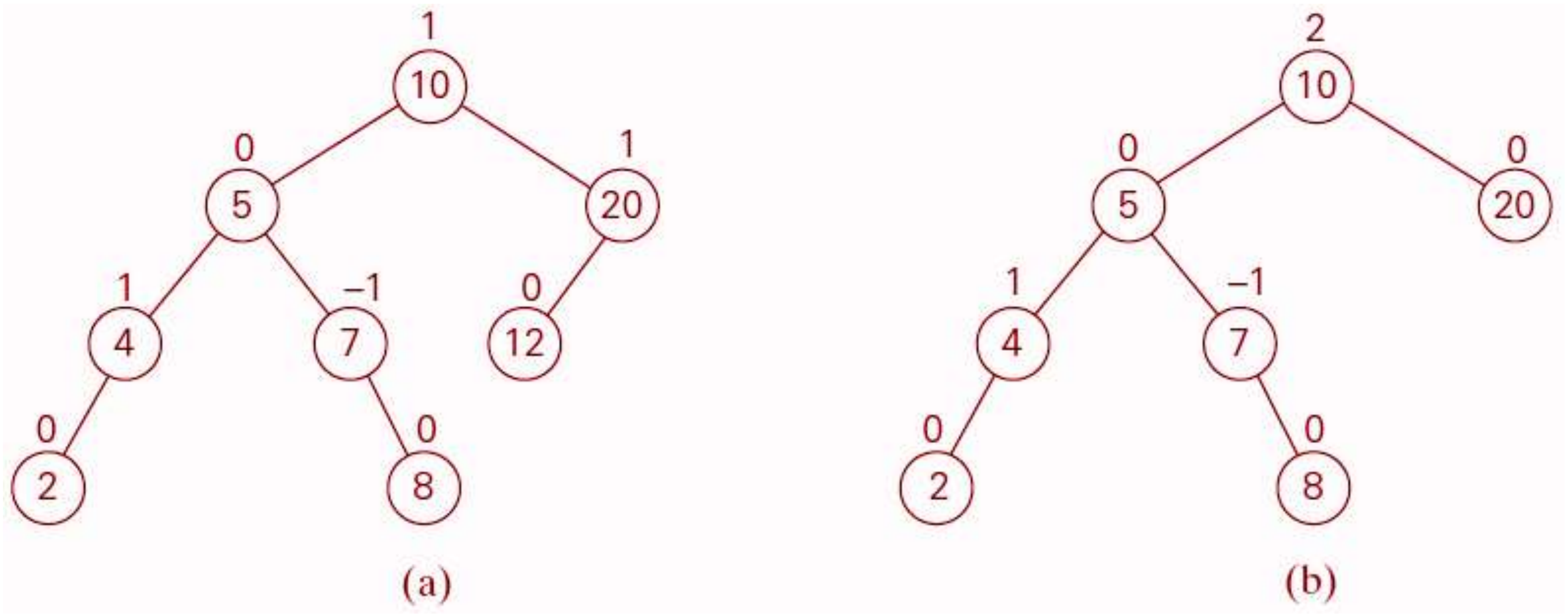
AVL Trees: An AVL tree is a BST in which the **balance factor** of every node is either 0, +1 or -1.

Balance factor: The difference between the heights of the node's left and right subtrees.

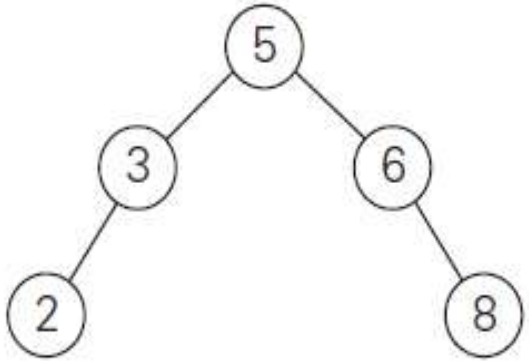
Height of a tree: The longest path from the root to a leaf. The height of a null tree is defined as -1.



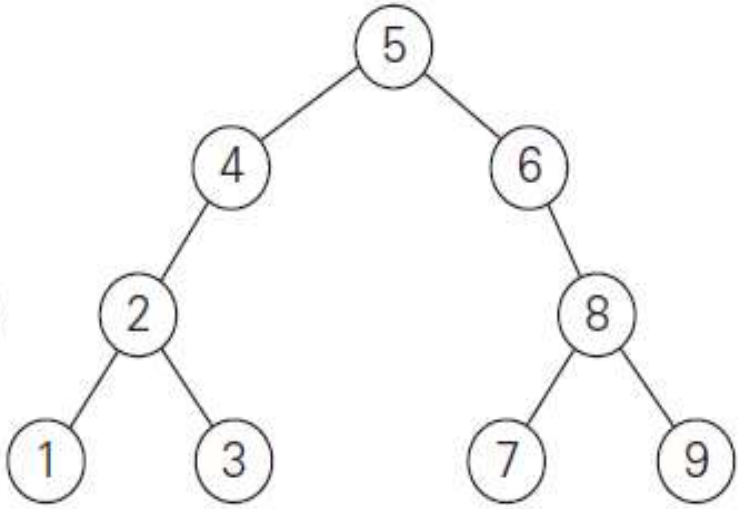
AVL Trees: An AVL tree is a BST in which the **balance factor** of every node is either 0, +1 or -1.



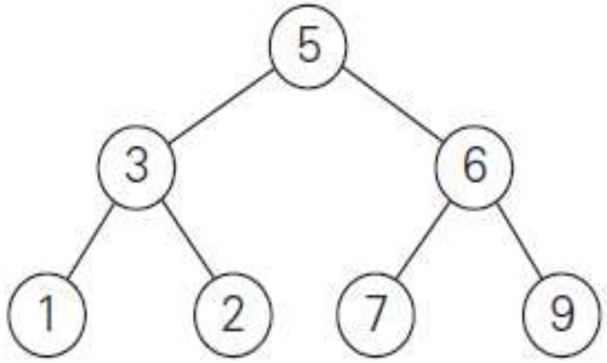
Which of the following binary trees are AVL trees?



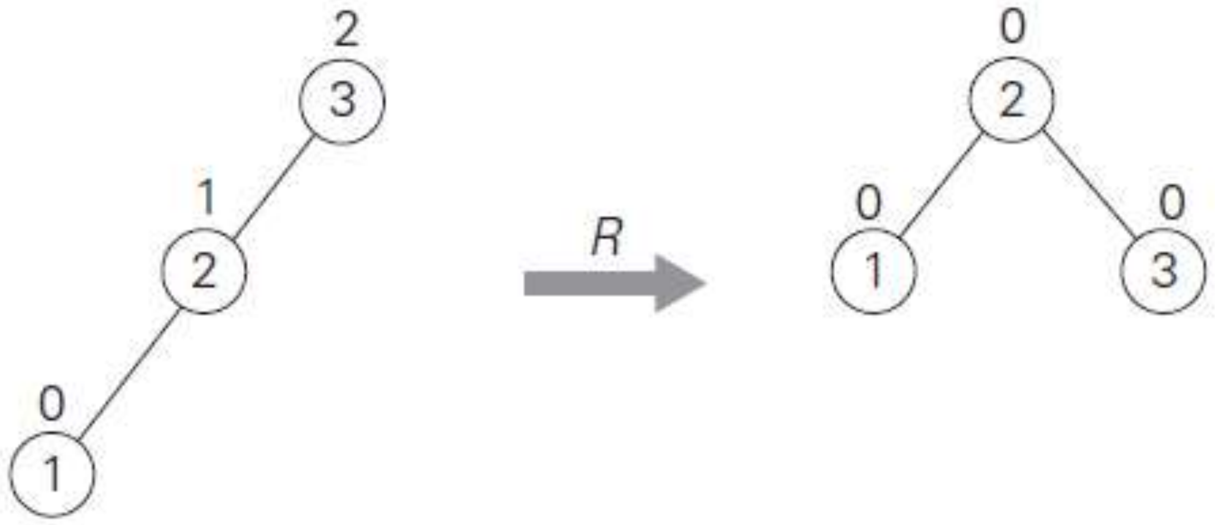
(a)



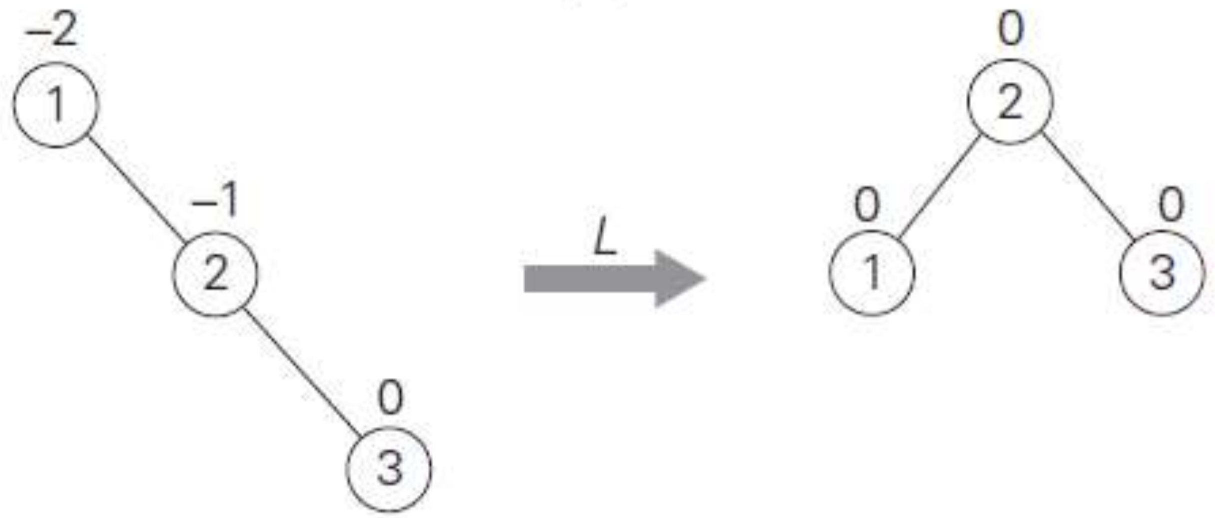
(b)



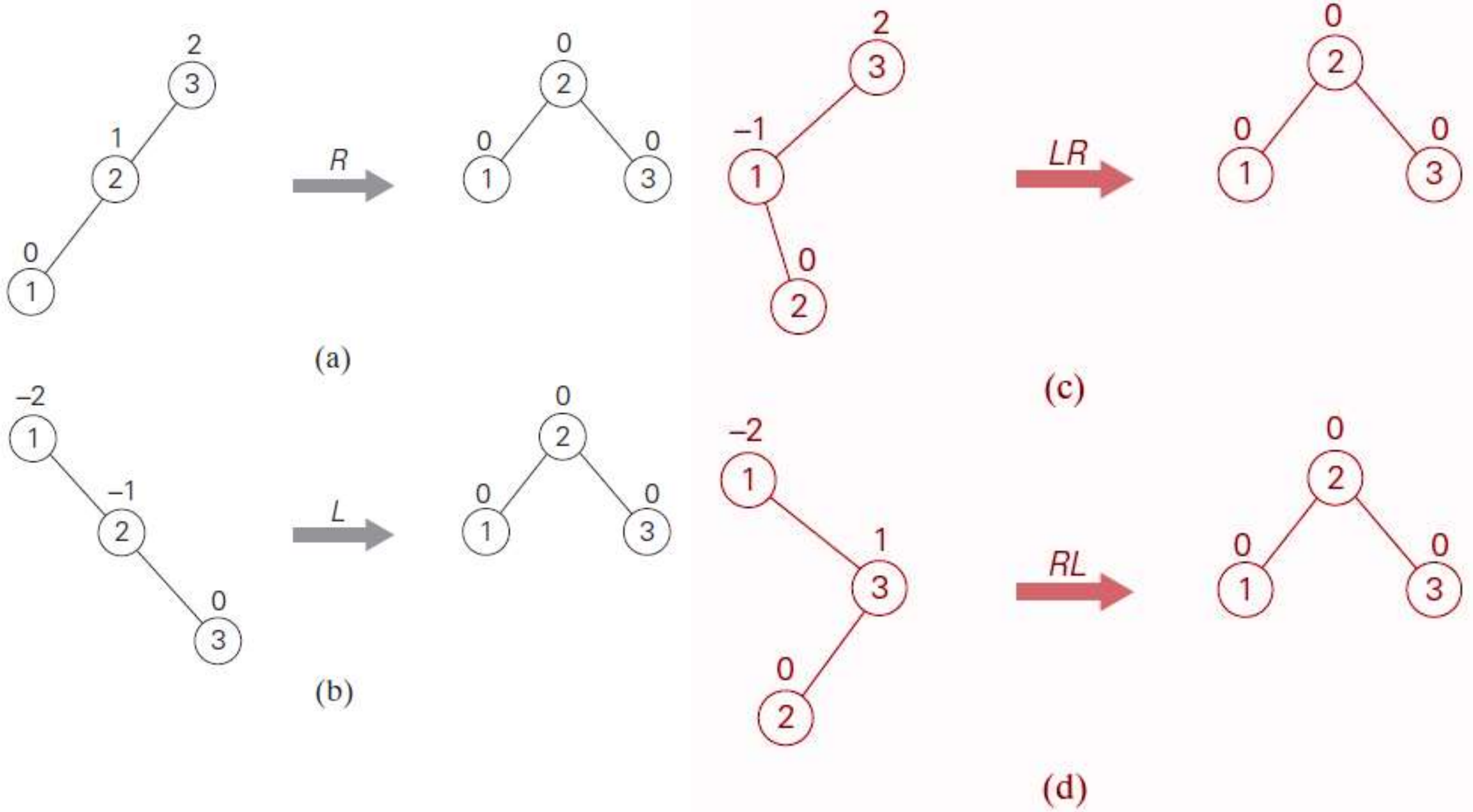
(c)



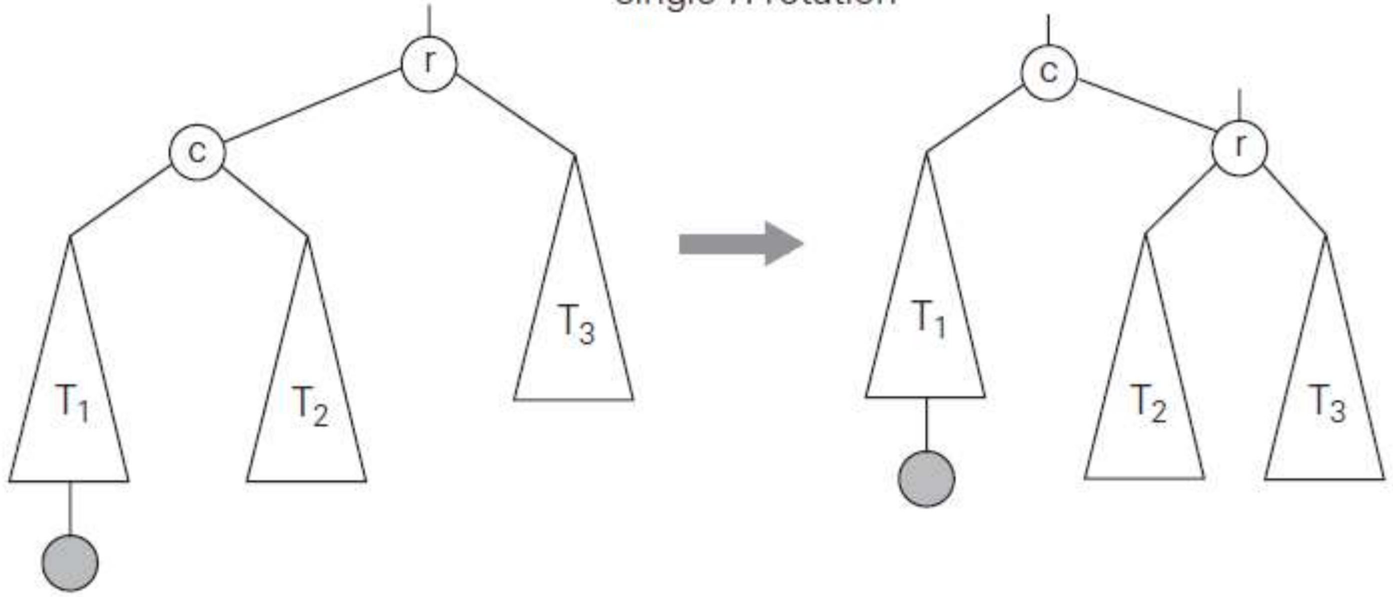
(a)



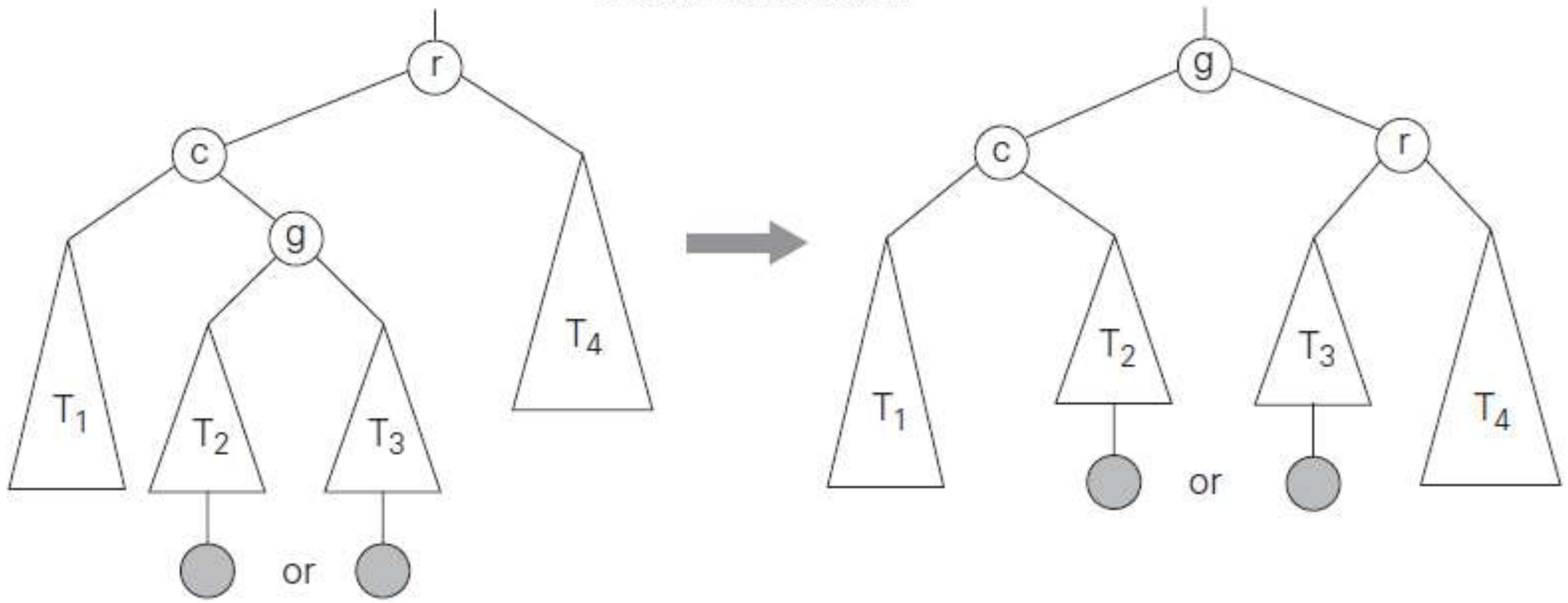
(b)


R
L
LR
RL
(a)
(b)
(c)
(d)

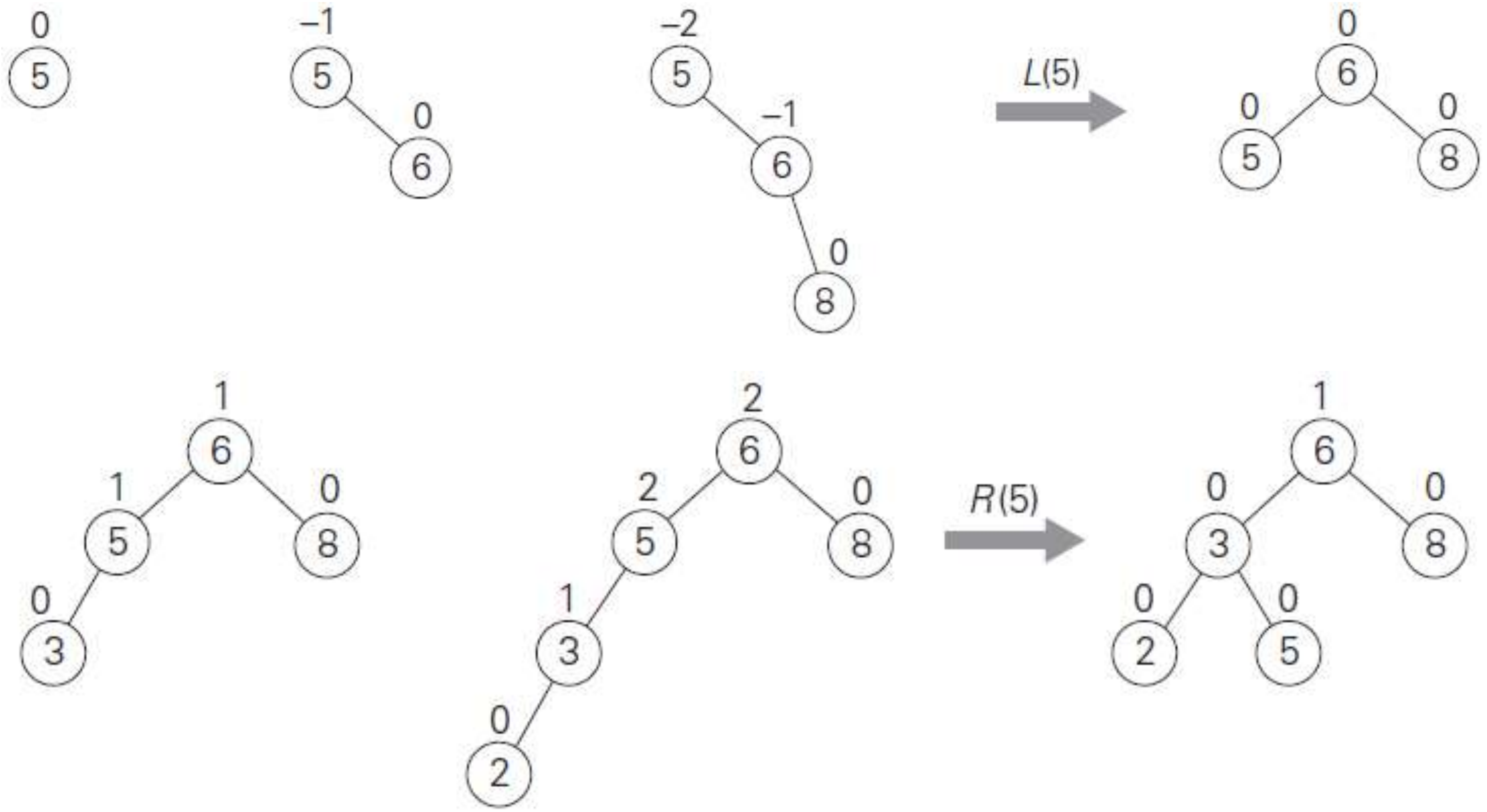
single *R*-rotation



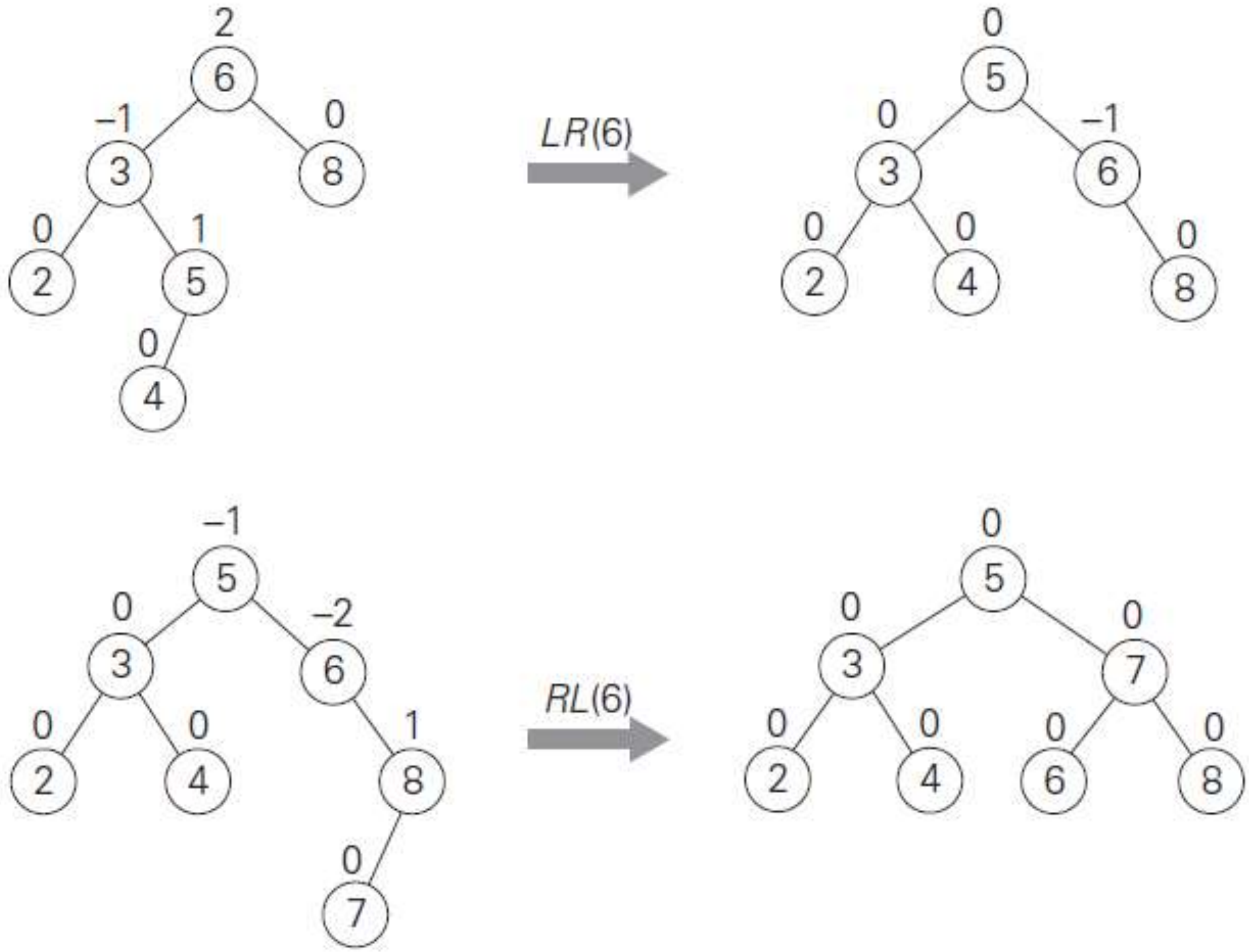
double *LR*-rotation



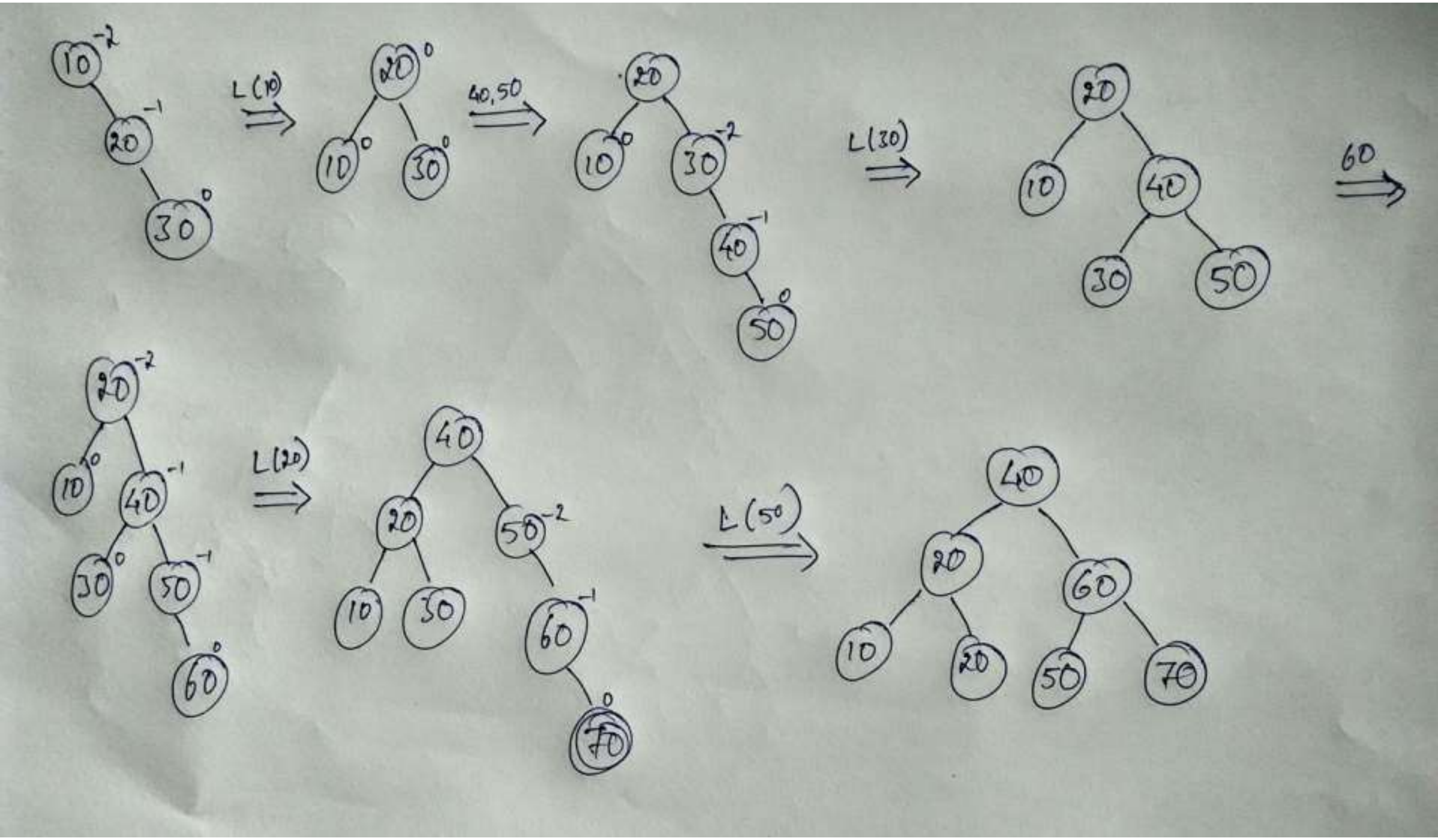
Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



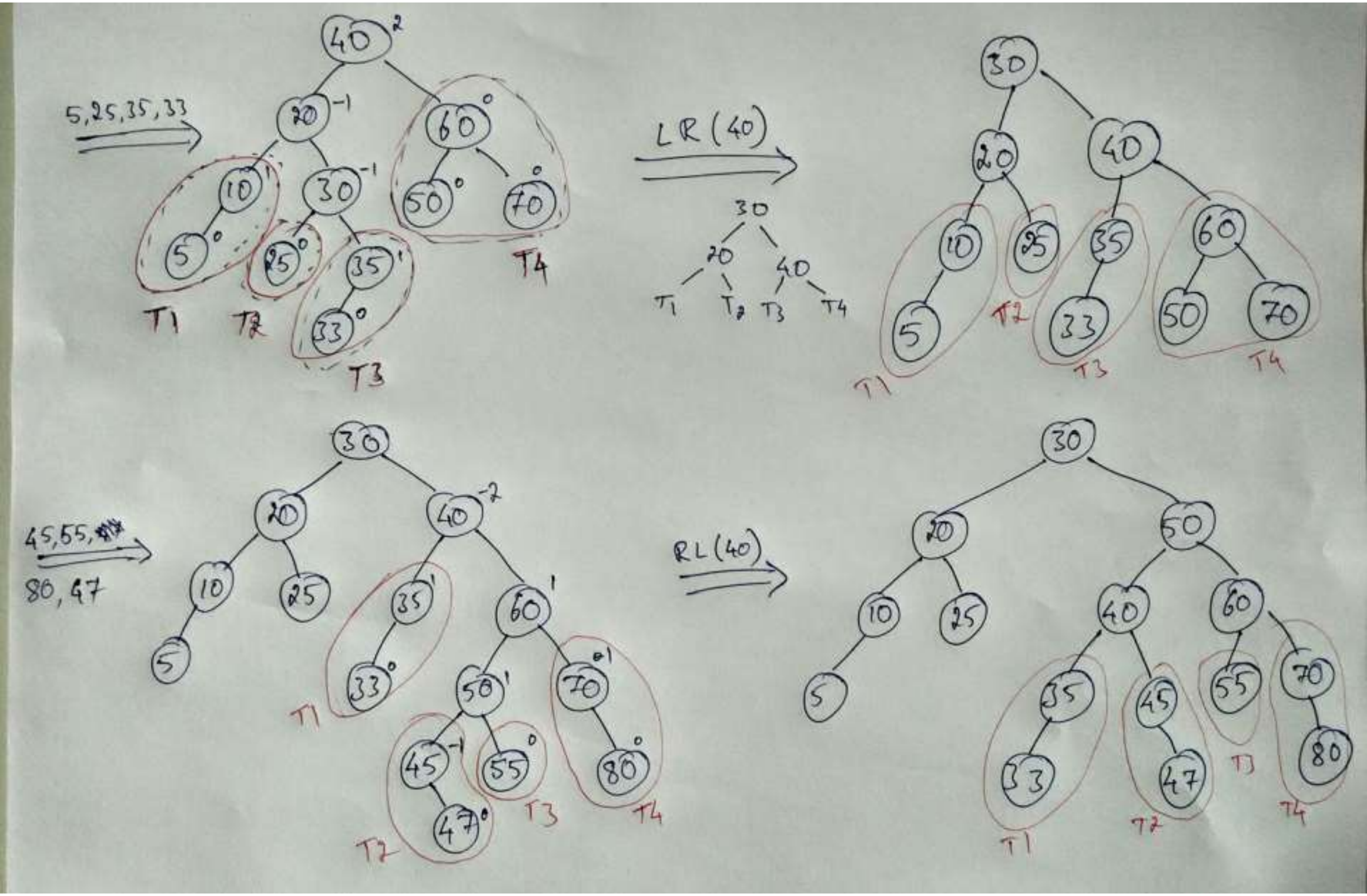
Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



Insert into an empty AVL Tree: 10, 20, 30, 40, 50, 60, 70.

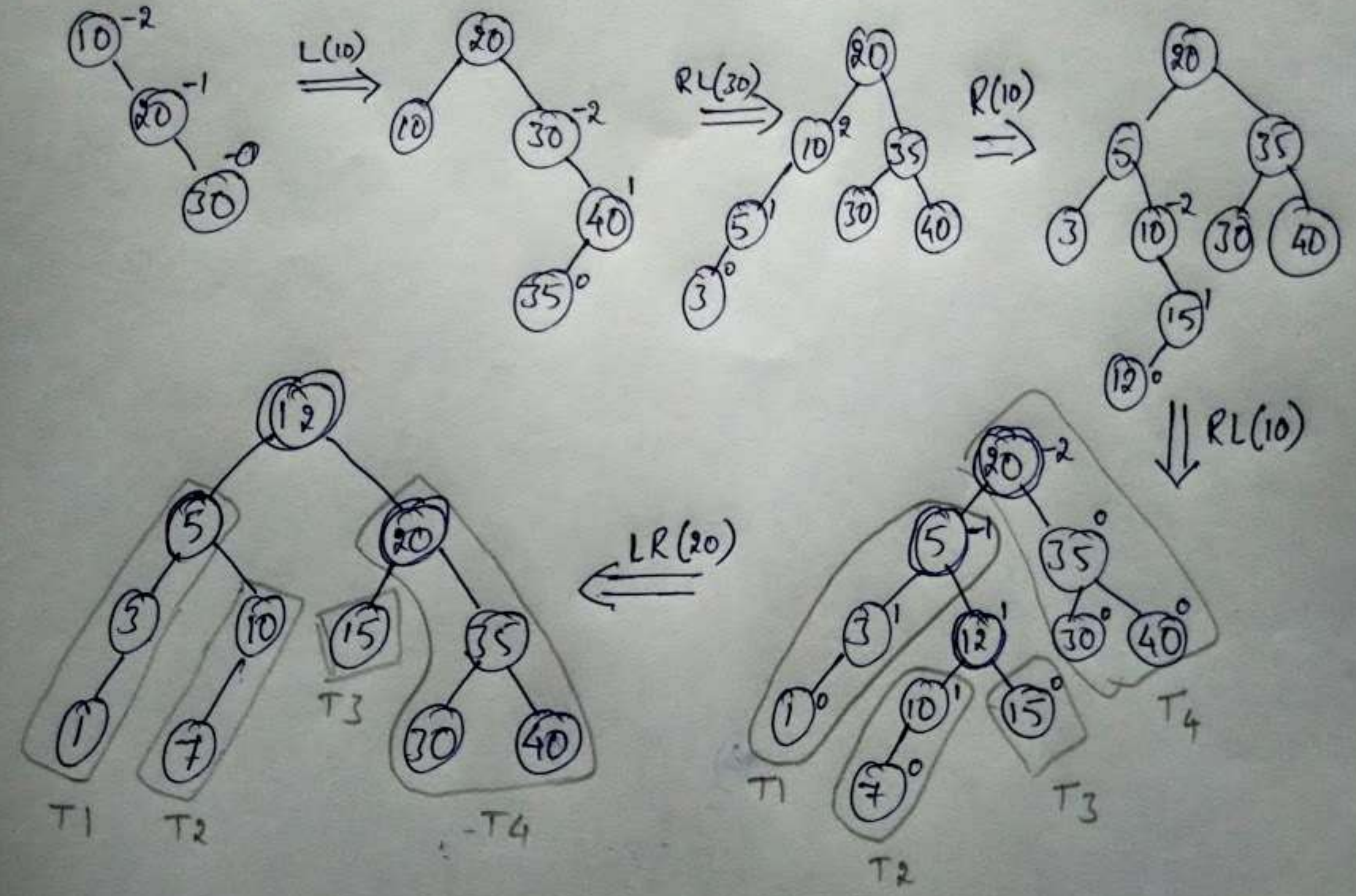


10, 20, 30, 40, 50, 60, 70, **5, 25, 35, 33, 45, 55, 80, 47.**



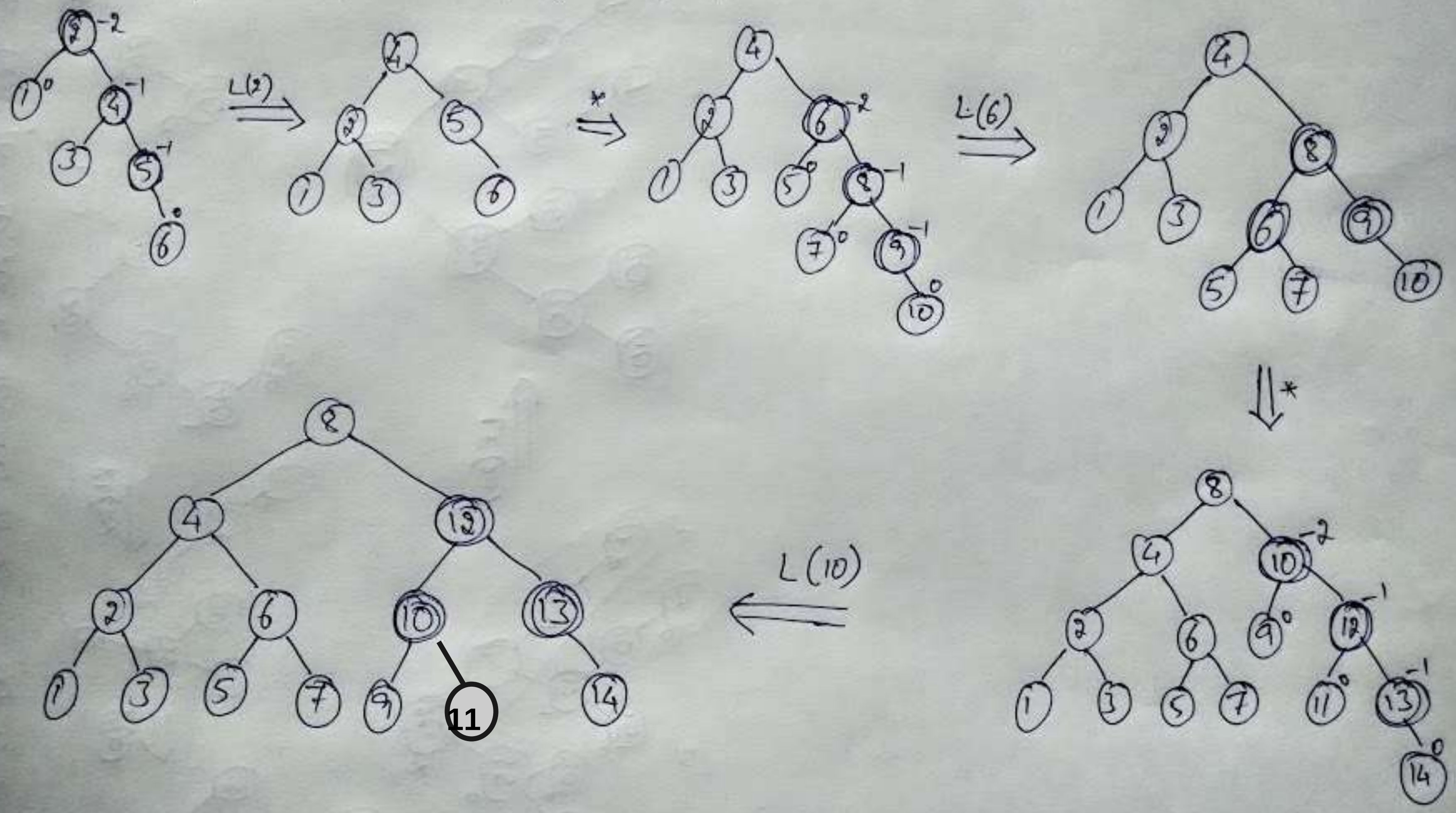
Construct AVL tree for the following sequence
10, 20, 30, 40, 35, 5, 3, 15, 12, 1, 7

AVL Tree for : 10, 20, 30, 40, 35, 5, 3, 15, 12, 1, 7

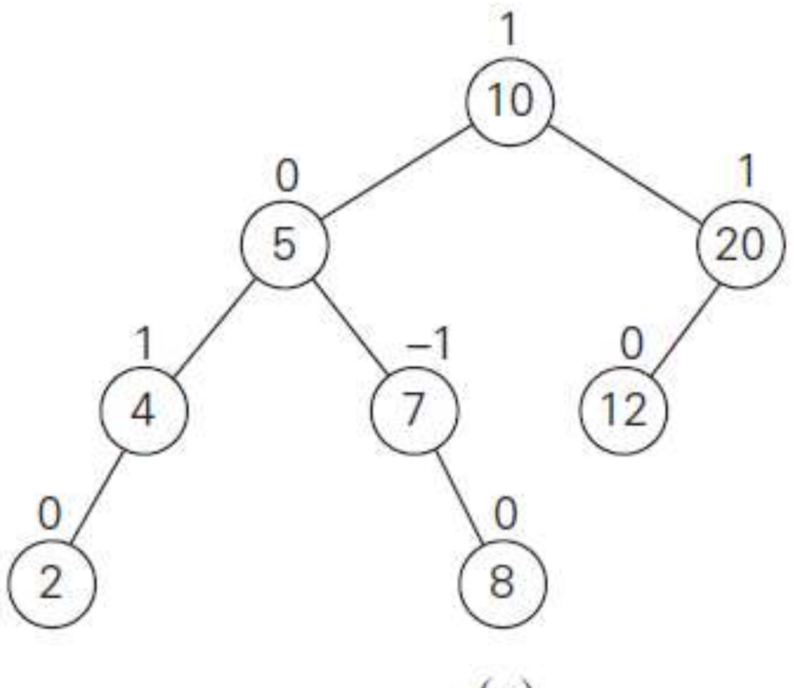


Construct AVL tree for the following sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14.

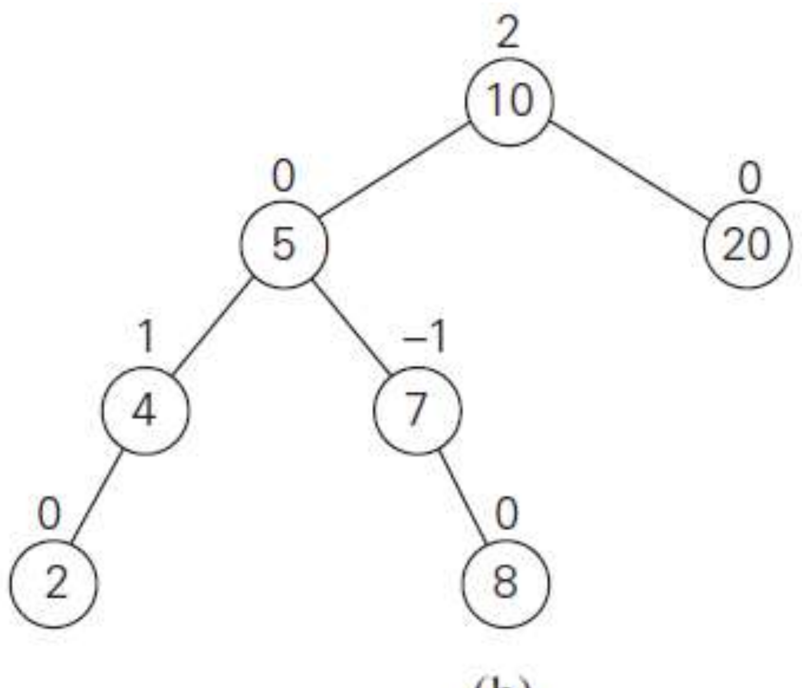
AVL Tree for: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~~15~~



- Boolean isAVL(BST t);
- How to delete a node from an AVL Tree? **[Optional]**
- Transform a BST into an AVL tree in-place. **[Optional]**



(a)

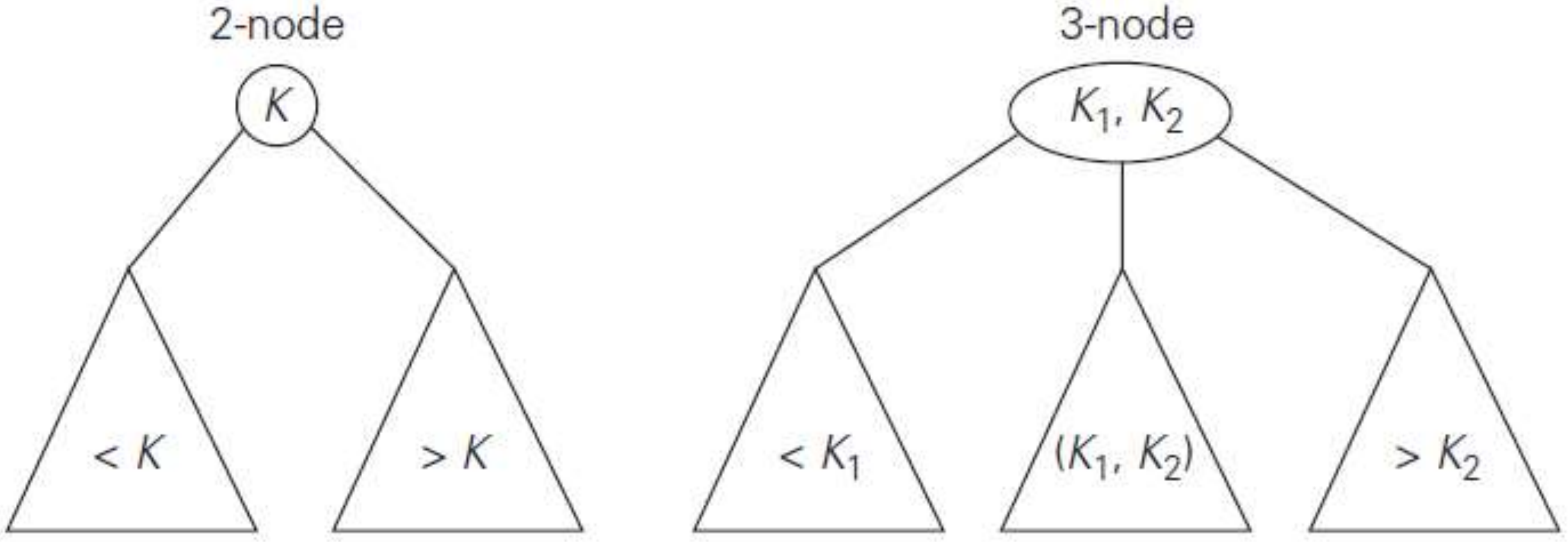


(b)

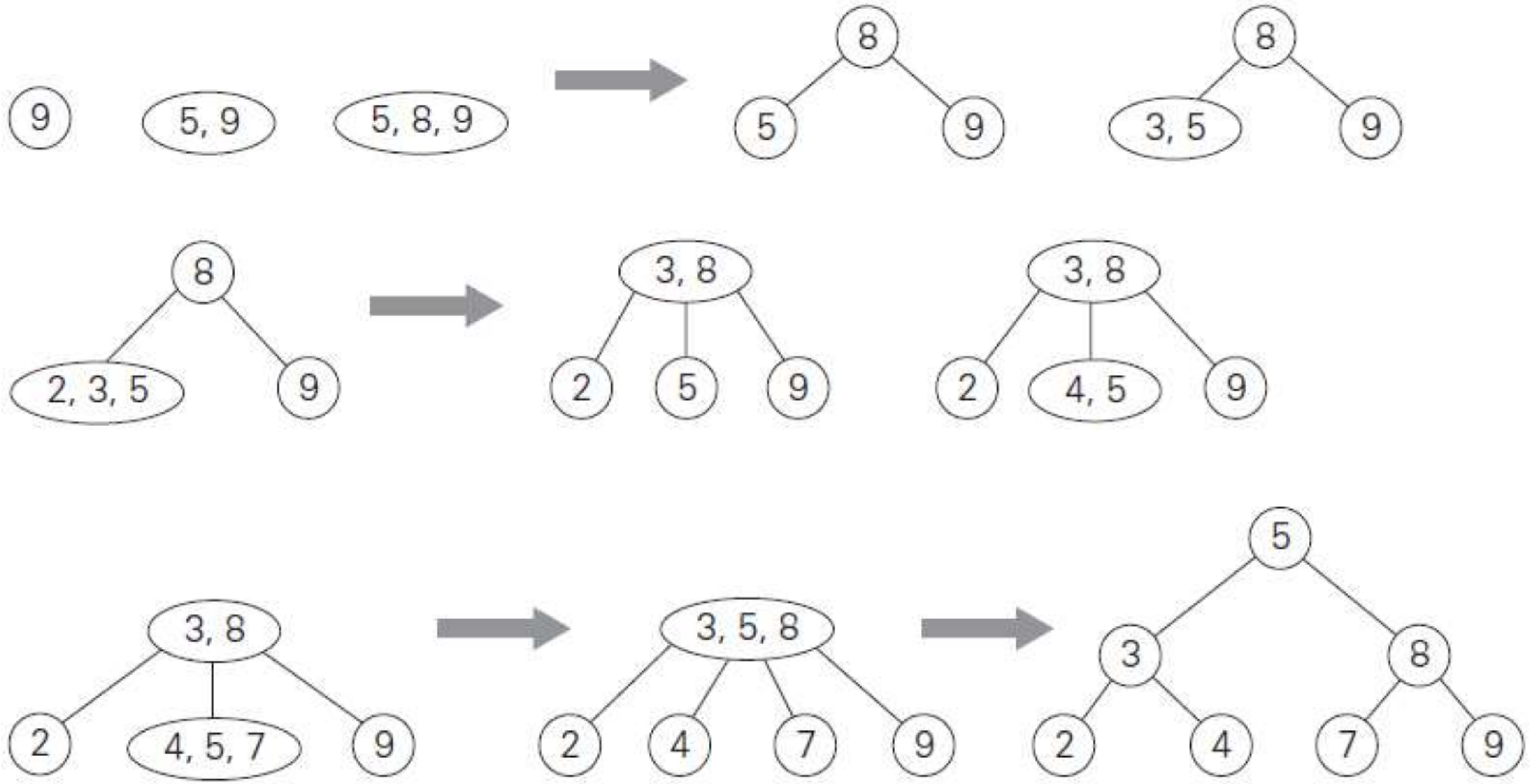
Search time efficiency of AVL trees:
Search is bounded by the height h of the tree.
Height is bounded both above and below by logarithmic functions.

$$\lfloor \log_2 n \rfloor \le h < 1.4405 \log_2(n+2) - 1.3277$$

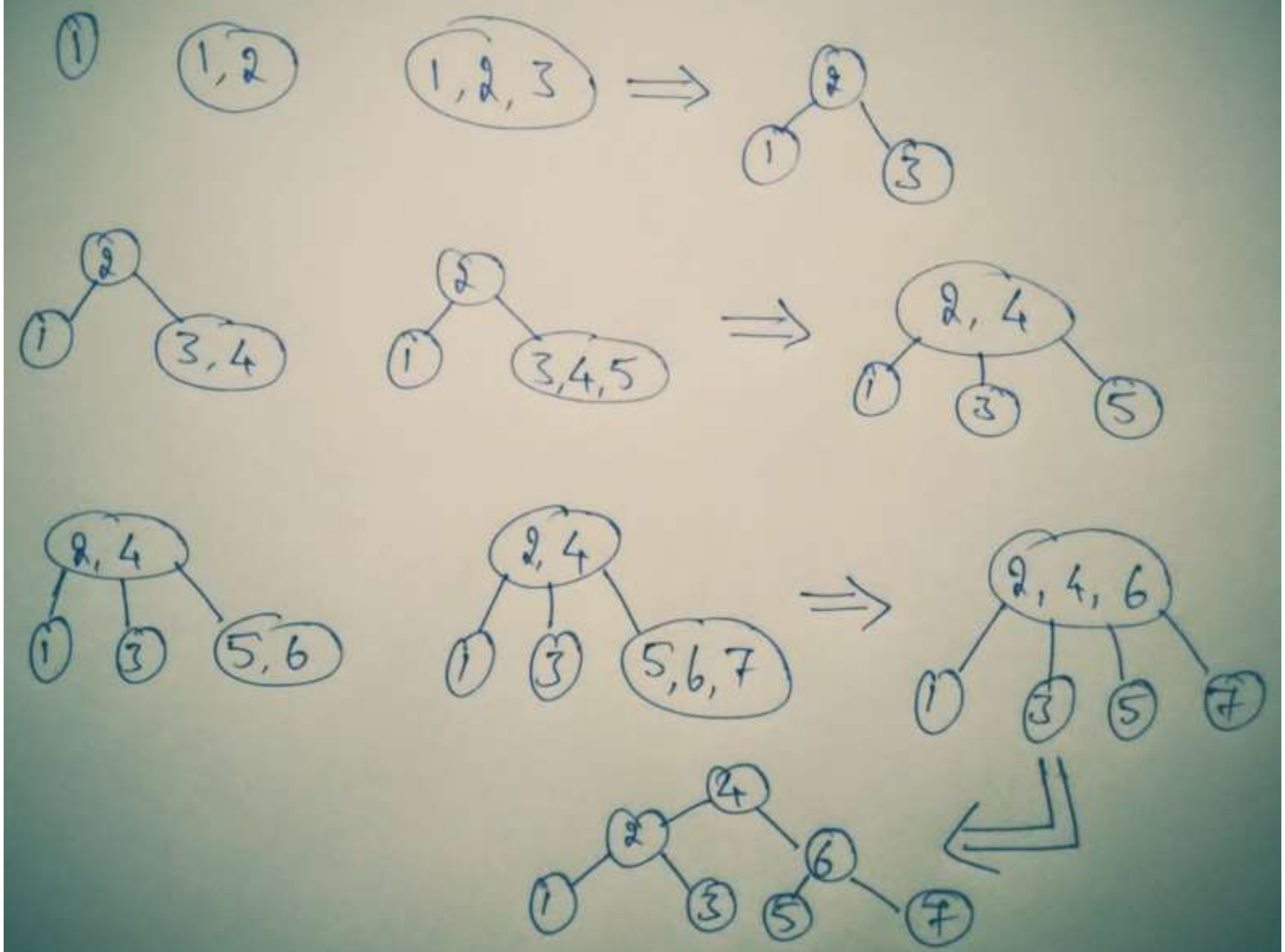
2-3 Trees: A 2-3 tree is a tree that can have nodes of two kinds; 2-nodes and 3-nodes. All its leaves must be on the same level so that a 2-3 tree is always height-balanced.



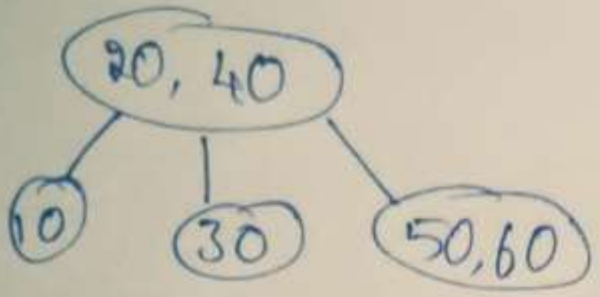
Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

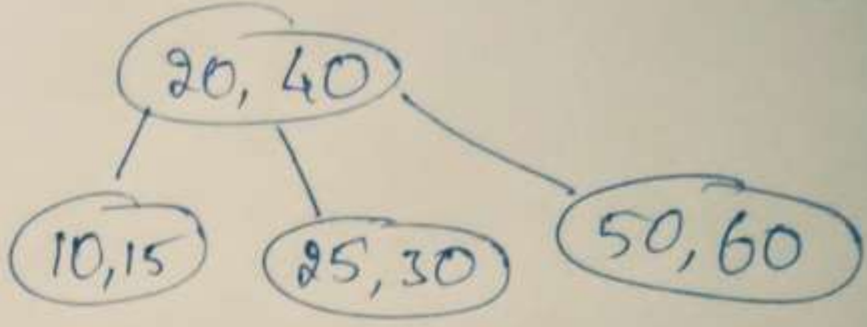


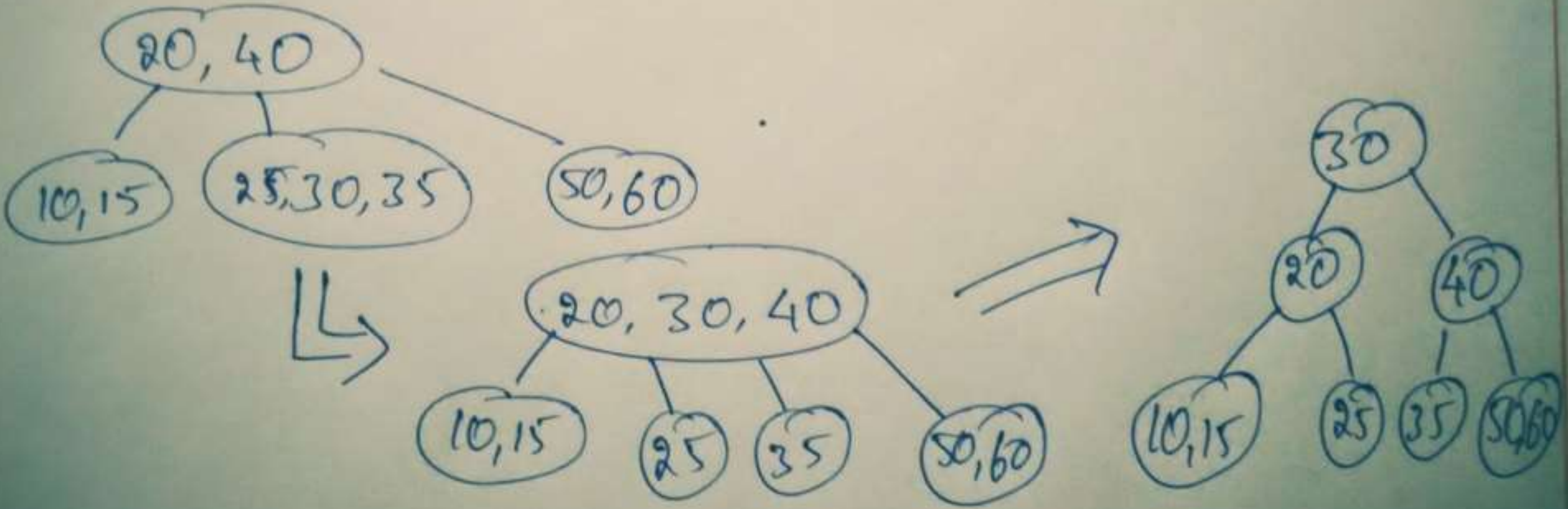
1, 2, 3, 4, 5, 6, 7,



10, 20, 30, 40, 50, 60, 15, 25, 35

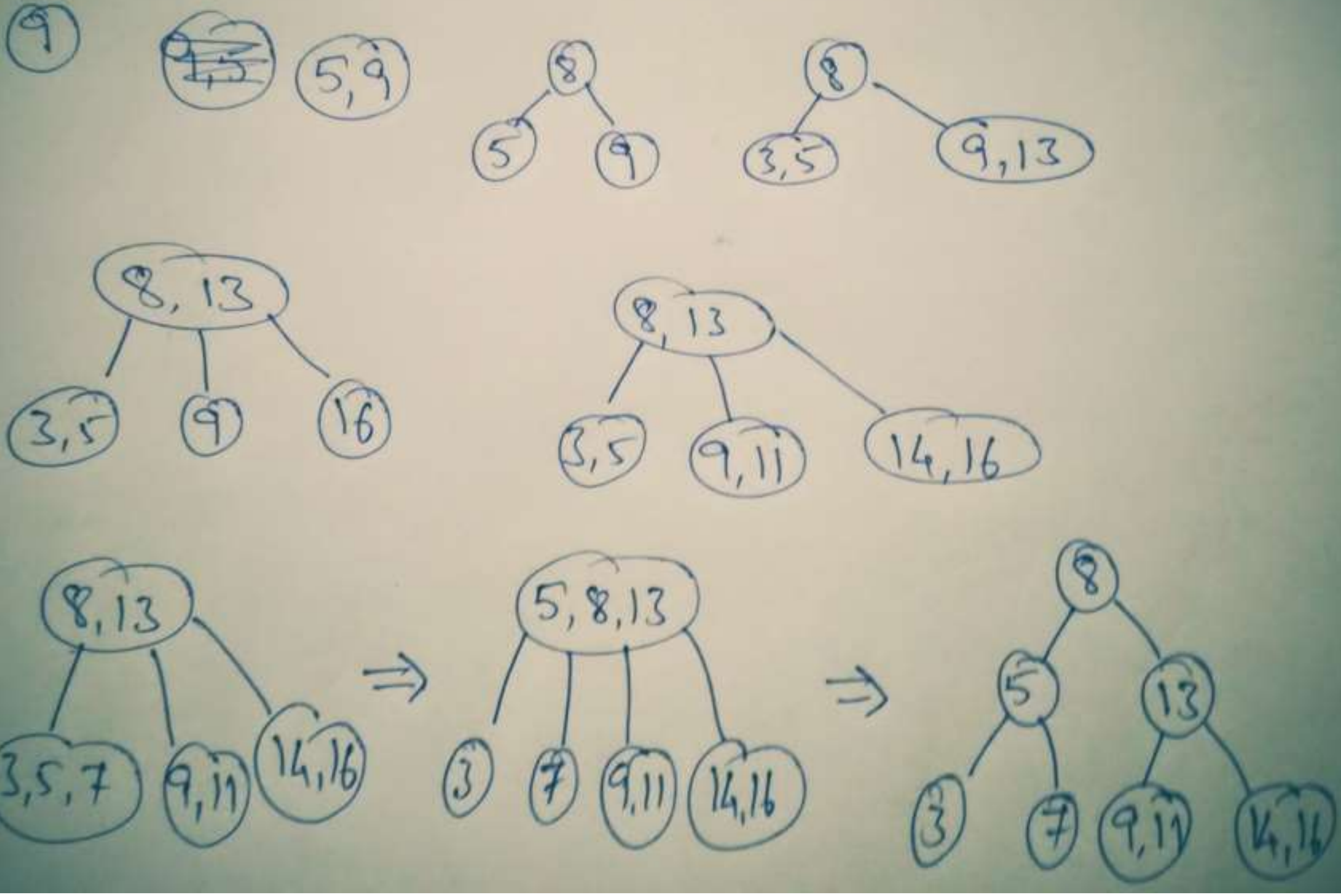






Construct a 2-3 Tree for the list
9, 5, 8, 3, 13, 16, 11, 14, 7

9, 5, 8, 3, 13, 16, 11, 14, 7



Analysis of 2-3 Tree:

Lower Bound of number of keys for height h:

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

$$h \leq \log_2(n+1) - 1$$

Analysis of 2-3 Tree:

Lower Bound of number of keys for height h:

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

$$h \leq \log_2(n+1) - 1$$

Upper Bound of number of keys for height h:

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1$$

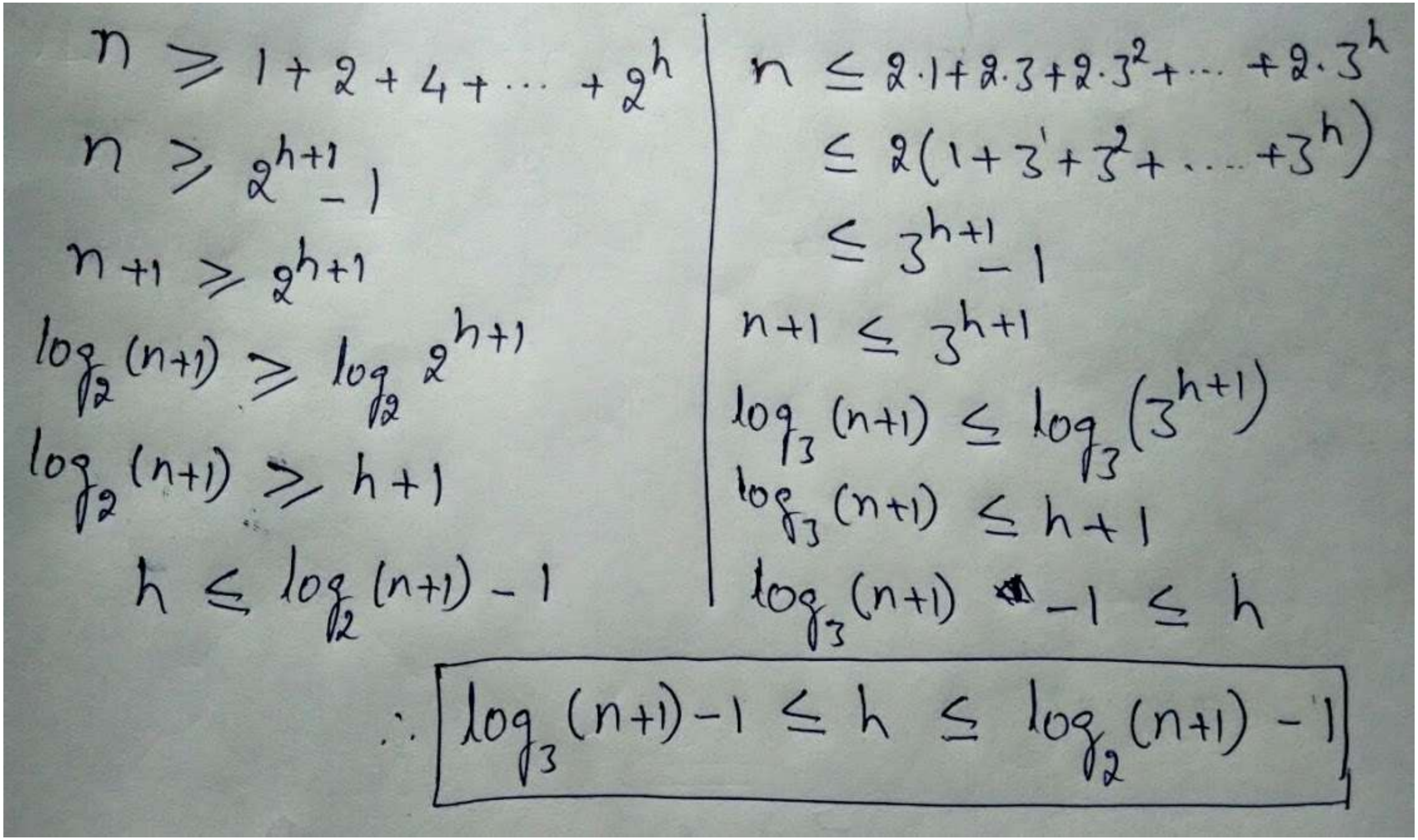
$$h \geq \log_3(n+1) - 1$$

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$$

Analysis of 2-3 Tree:

Lower Bound of n for a given h

Upper Bound of n for a



$$n \geq 1 + 2 + 4 + \cdots + 2^h$$

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

$$n + 1 \geq 2^{h+1}$$

$$\log_2(n+1) \geq \log_2 2^{h+1}$$

$$\log_2(n+1) \geq h + 1$$

$$h \leq \log_2(n+1) - 1$$

$$n \leq 2\cdot 1 + 2\cdot 3 + 2\cdot 3^2 + \cdots + 2\cdot 3^h$$

$$\leq 2(1 + 3^1 + 3^2 + \cdots + 3^h)$$

$$\leq 3^{h+1} - 1$$

$$n + 1 \leq 3^{h+1}$$

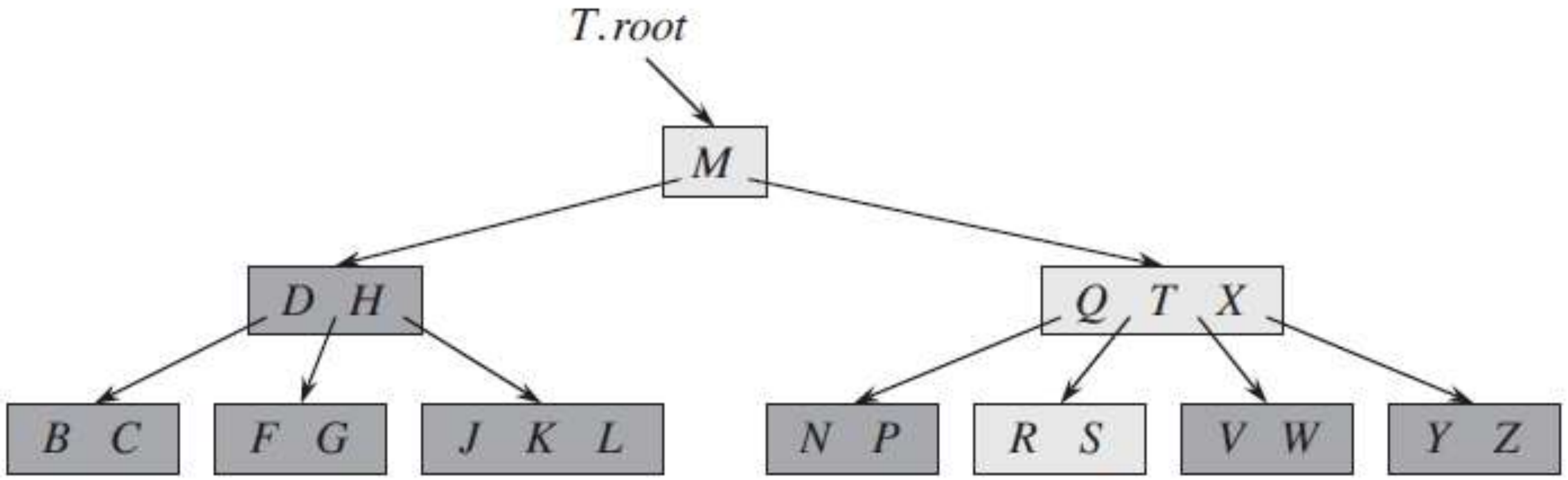
$$\log_3(n+1) \leq \log_3(3^{h+1})$$

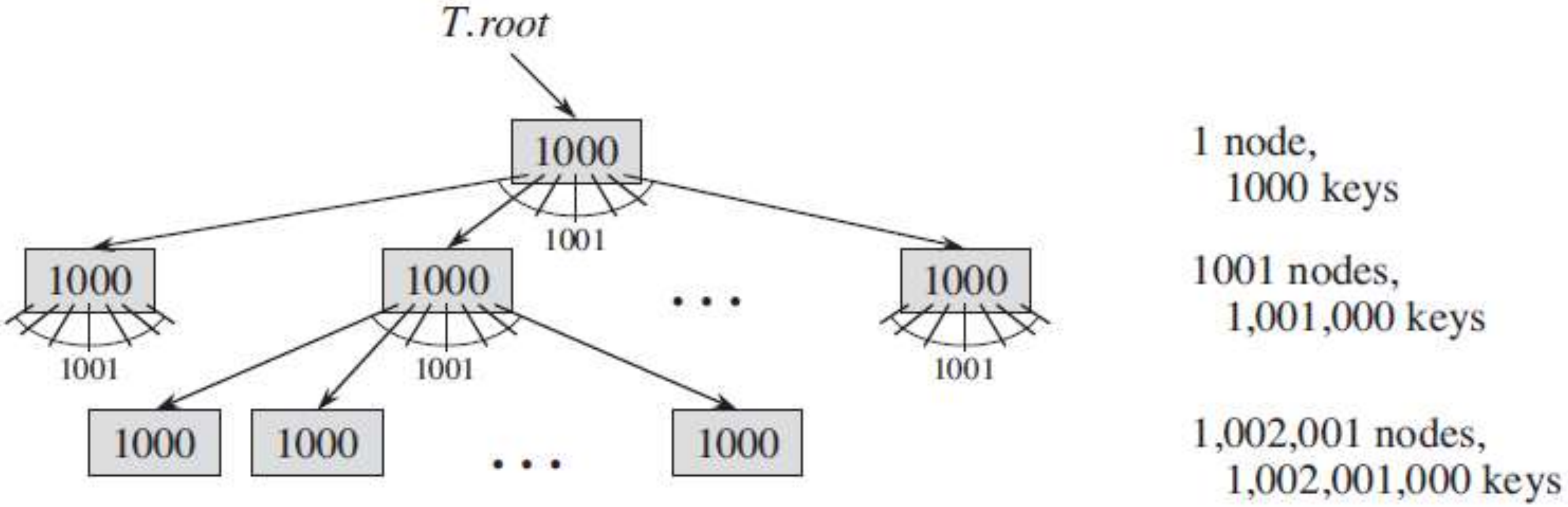
$$\log_3(n+1) \leq h + 1$$

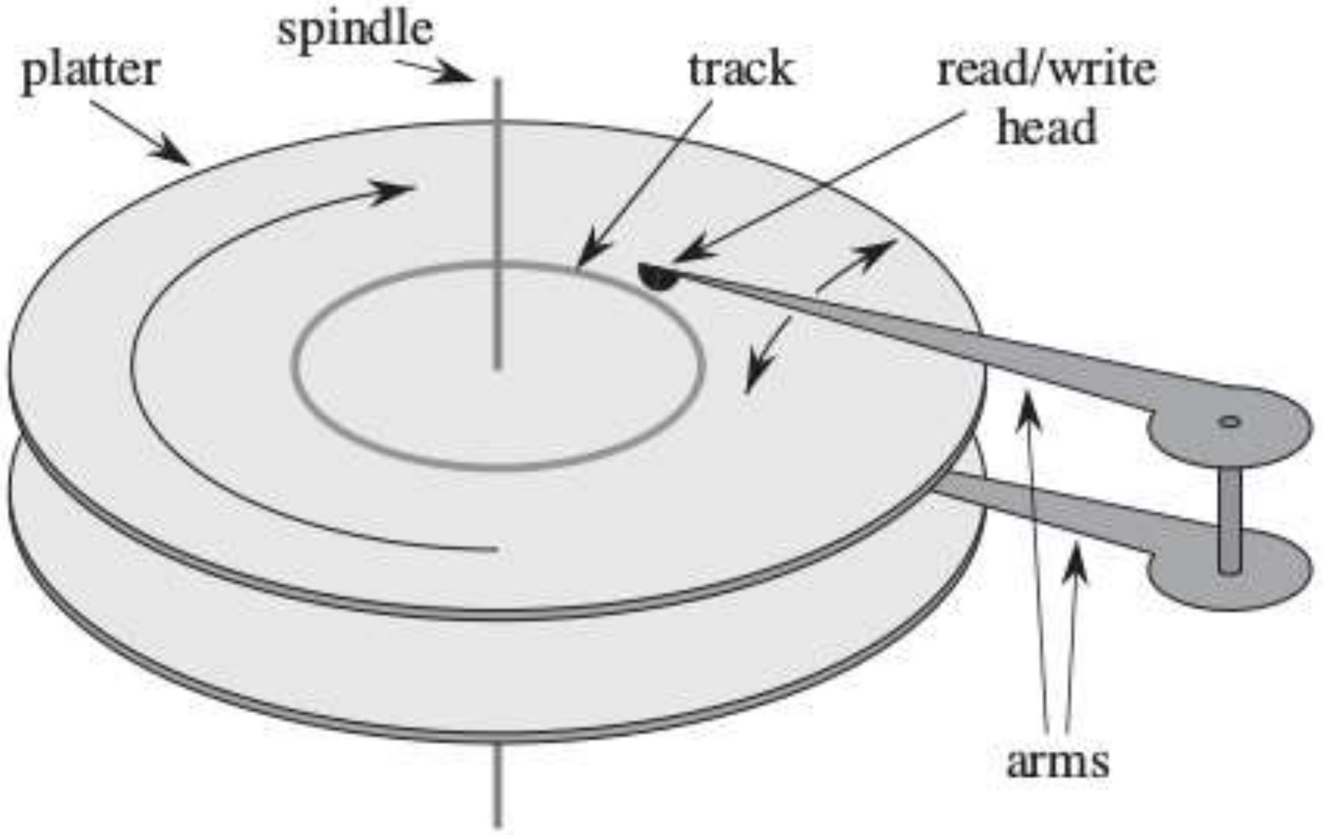
$$\log_3(n+1) - 1 \leq h$$

$$\therefore \boxed{\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1}$$

B-Trees


T.root
M
D H
Q T X
B C
F G
J K L
N P
R S
V W
Y Z


T.root
1000
1001
1 node,
1000 keys
1001 nodes,
1,001,000 keys
1,002,001 nodes,
1,002,001,000 keys



the two principal components of the running time:

- the number of disk accesses, and
- the CPU (computing) time.

B-Trees

A ***B*-*tree*** T is a rooted tree (whose root is $T.root$) 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, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,

   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 $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

…

B-Trees

…

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

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1} .$$

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:

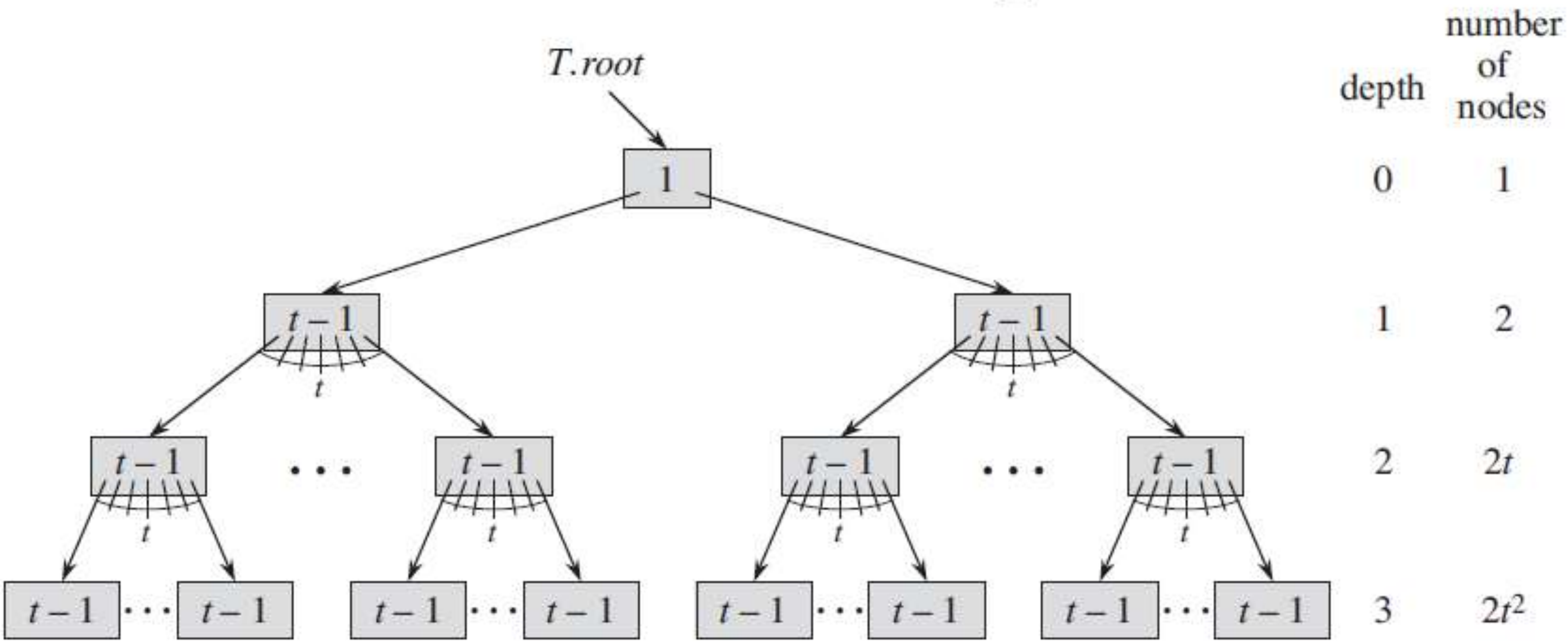
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   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 exactly $2t - 1$ keys.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree.

Theorem If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$.



Theorem If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$.

Proof The root of a B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys. Thus, T, whose height is h, has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h it has at least $2t^{h-1}$ nodes.
Thus, the number n of keys satisfies the inequality

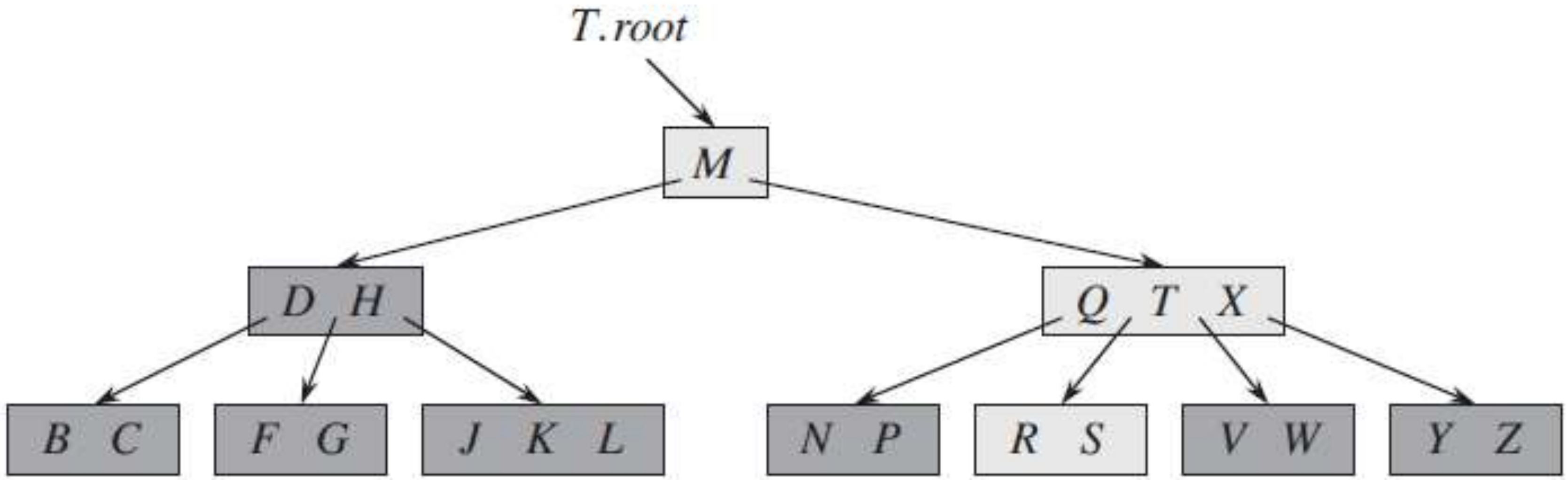
$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 . \end{aligned}$$

By simple algebra, we get $t^h \leq (n+1)/2$.
Taking base-t logarithms of both sides proves the theorem.

B-Trees

Why don't we allow a minimum degree of $t = 1$?

For what values of t is the following tree a legal B-tree?



Basic operations on B-trees

B-TREE-SEARCH,

B-TREE-CREATE,

B-TREE-INSERT.

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root; we do have to perform a DISK-WRITE of the root, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.
- The procedures are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

```
B-TREE-SEARCH($x, k$)
$i = 1$
while $i \le x.n$ and $k > x.key_i$
    $i = i + 1$
if $i \le x.n$ and $k == x.key_i$
    return $(x, i)$
elseif $x.leaf$
    return NIL
else DISK-READ($x.c_i$)
    return B-TREE-SEARCH($x.c_i, k$)
```

- accesses $O(h) = O(\log_t n)$ disk pages,
  where h is the height of the B-tree and n is the number of keys in the B-tree.
- total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

```
B-TREE-CREATE(T)
1  x = ALLOCATE-NODE()
2  x.leaf = TRUE
3  x.n = 0
4  DISK-WRITE(x)
5  T.root = x
```

B-TREE-CREATE requires

$O(1)$ disk operations and

$O(1)$ CPU time.

Inserting a key into a B-tree

Insert the new key into an existing leaf node.
Since we cannot insert a key into a leaf node that is full,
split a full node y (having $2t-1$ keys) around its ***median key*** $y.key_t$
into two nodes having only $t-1$ keys each.
The median key moves up into y's parent.
But if y's parent is also full, we must split it before we can insert the new key,
and thus we could end up splitting full nodes all the way up the tree.

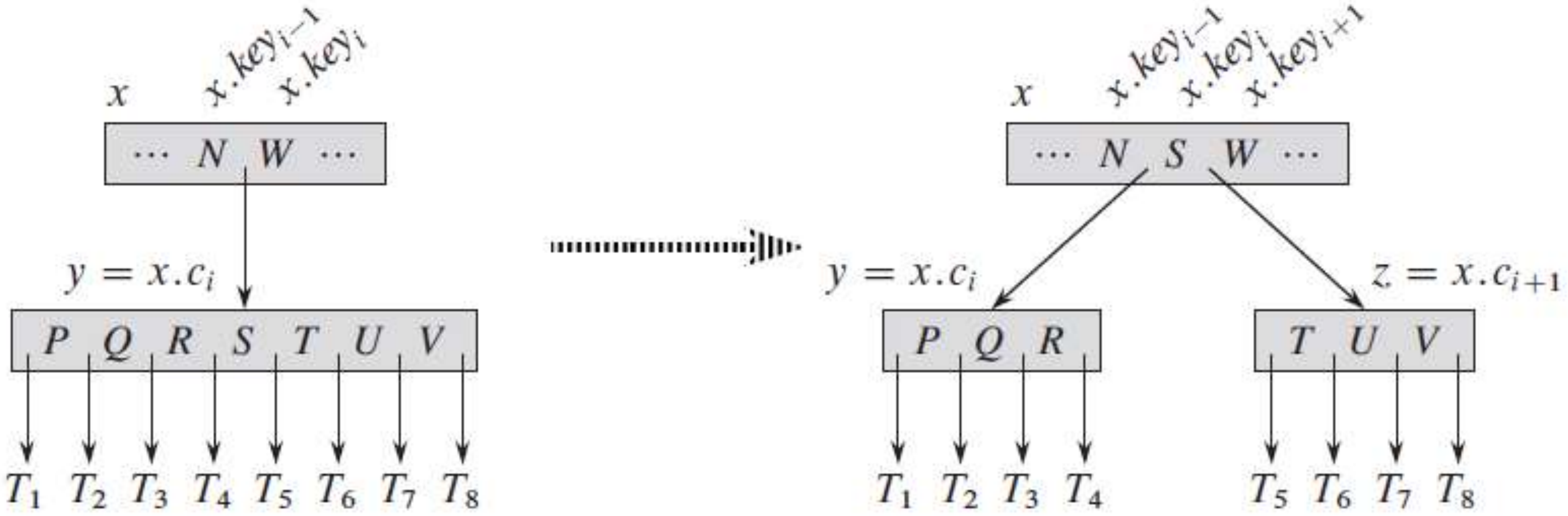
Splitting a node in a B-tree

The procedures we present are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.
To do so, we do not wait to find out whether we will actually need to split a full node in order to do the insertion. Instead, as we travel down the tree, we split each full node we come to along the way (including the leaf itself).

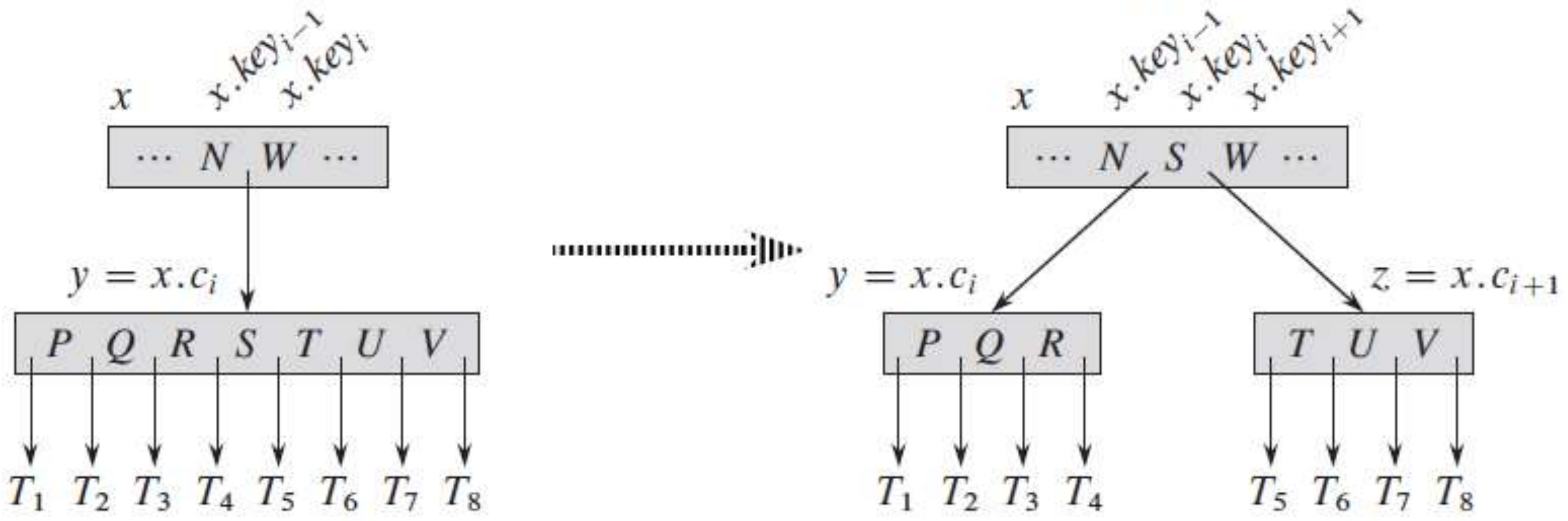
The procedure B-TREE-SPLIT-CHILD takes as input a *nonfull* internal node x (assumed to be in main memory) and an index i such that $x.c_i$ (also assumed to be in main memory) is a *full* child of x. The procedure then splits this child in two and adjusts x so that it has an additional child.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes as input a *nonfull* internal node x (assumed to be in main memory) and an index i such that $x.c_i$ (also assumed to be in main memory) is a *full* child of x. The procedure then splits this child in two and adjusts x so that it has an additional child.

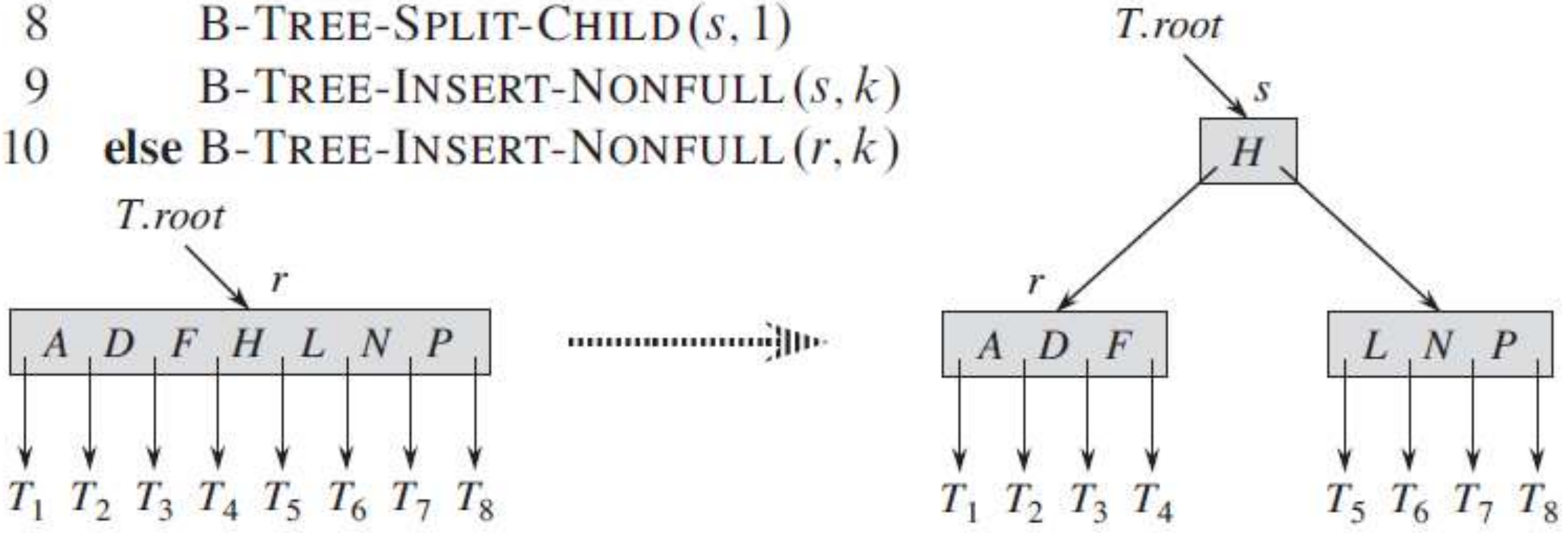


```
B-TREE-SPLIT-CHILD$(x, i)$
$z$ = ALLOCATE-NODE()
$y = x.c_i$
$z.leaf = y.leaf$
$z.n = t - 1$
**for** $j = 1$ **to** $t - 1$
    $z.key_j = y.key_{j+t}$
**if** not $y.leaf$
    **for** $j = 1$ **to** $t$
        $z.c_j = y.c_{j+t}$
$y.n = t - 1$
**for** $j = x.n + 1$ **downto** $i + 1$
    $x.c_{j+1} = x.c_j$
$x.c_{i+1} = z$
**for** $j = x.n$ **downto** $i$
    $x.key_{j+1} = x.key_j$
$x.key_i = y.key_t$
$x.n = x.n + 1$
DISK-WRITE$(y)$
DISK-WRITE$(z)$
DISK-WRITE$(x)$
```



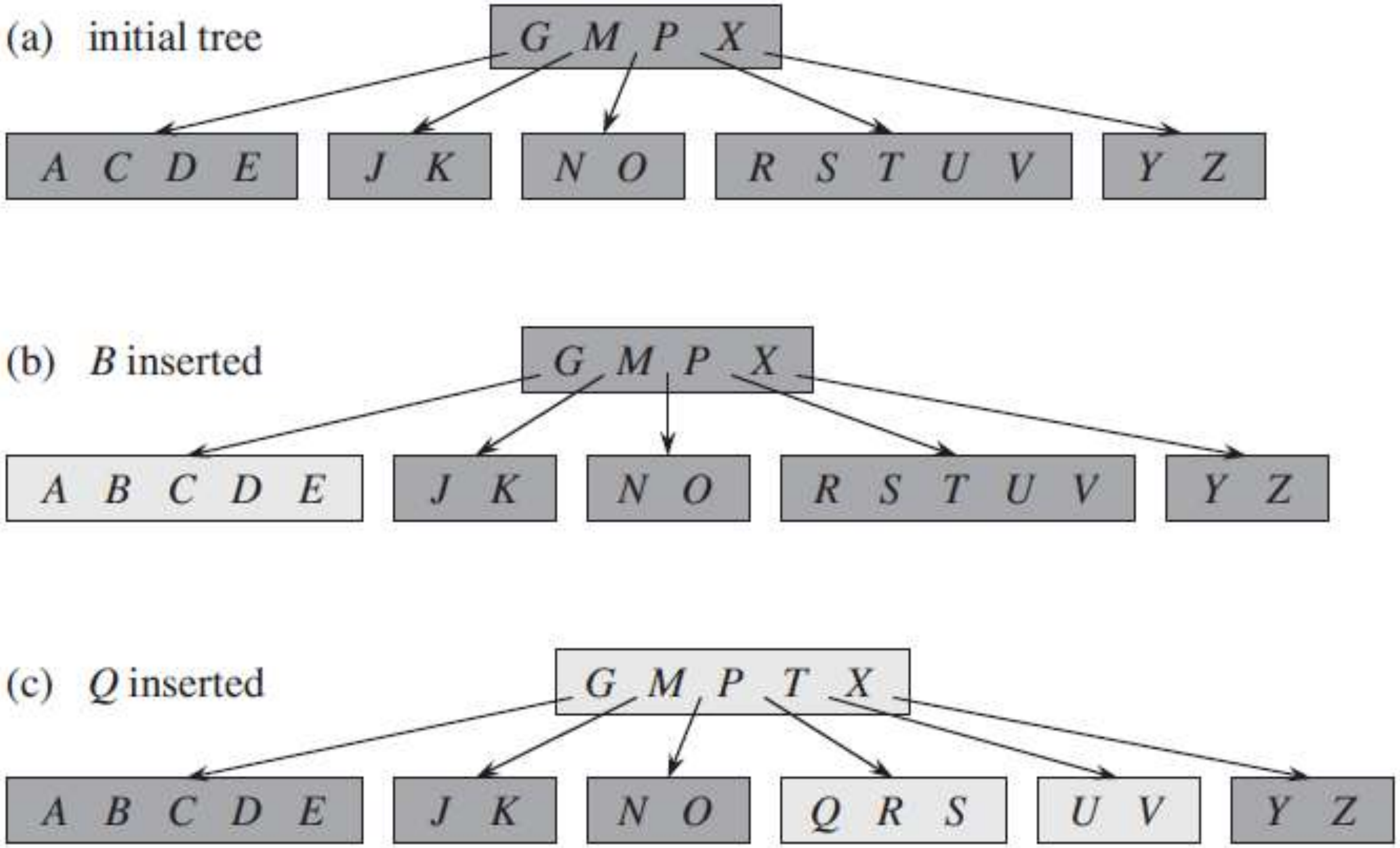
Inserting a key into a B-tree in a single pass down the tree

```
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)
```

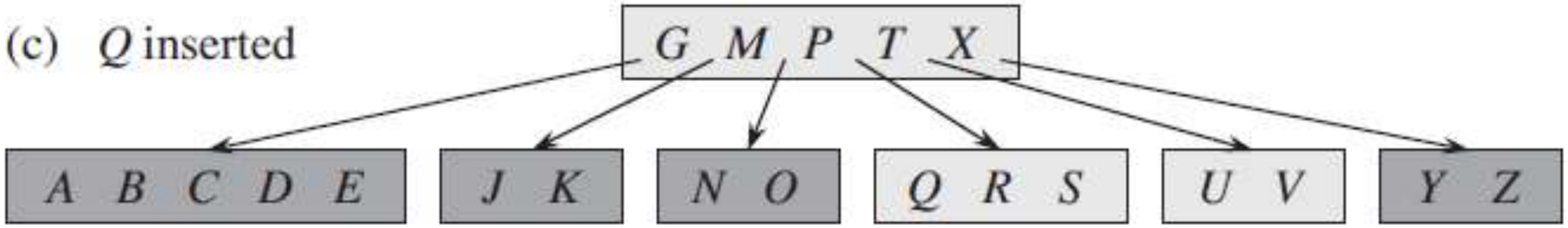


Inserting a key into a B-tree in a single pass down the tree

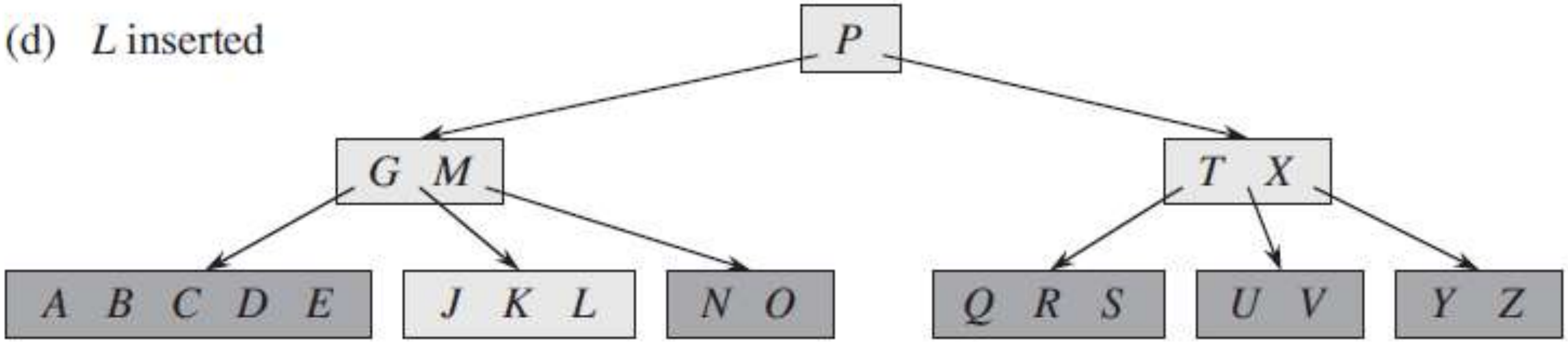
```
B-TREE-INSERT-NONFULL($x, k$)
$i = x.n$
if $x.leaf$
    while $i \geq 1$ and $k < x.key_i$
        $x.key_{i+1} = x.key_i$
        $i = i - 1$
    $x.key_{i+1} = k$
    $x.n = x.n + 1$
    DISK-WRITE($x$)
else while $i \geq 1$ and $k < x.key_i$
        $i = i - 1$
    $i = i + 1$
    DISK-READ($x.c_i$)
    if $x.c_i.n == 2t - 1$
        B-TREE-SPLIT-CHILD($x, i$)
        if $k > x.key_i$
            $i = i + 1$
    B-TREE-INSERT-NONFULL($x.c_i, k$)
```


(a) initial tree
G M P X
A C D E
J K
N O
R S T U V
Y Z
(b) B inserted
G M P X
A B C D E
J K
N O
R S T U V
Y Z
(c) Q inserted
G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z

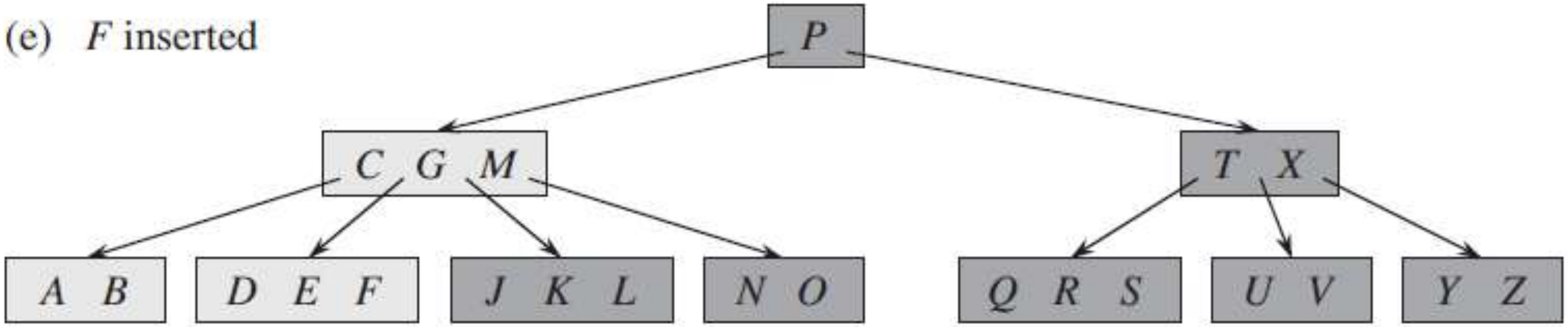
(c) *Q* inserted



(d) *L* inserted



(e) *F* inserted

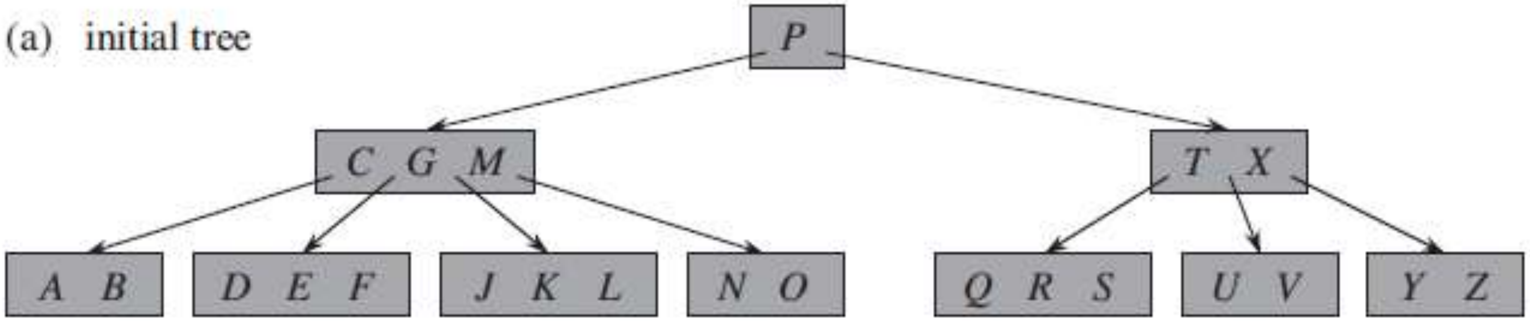


Analysis: Inserting a key into a B-tree

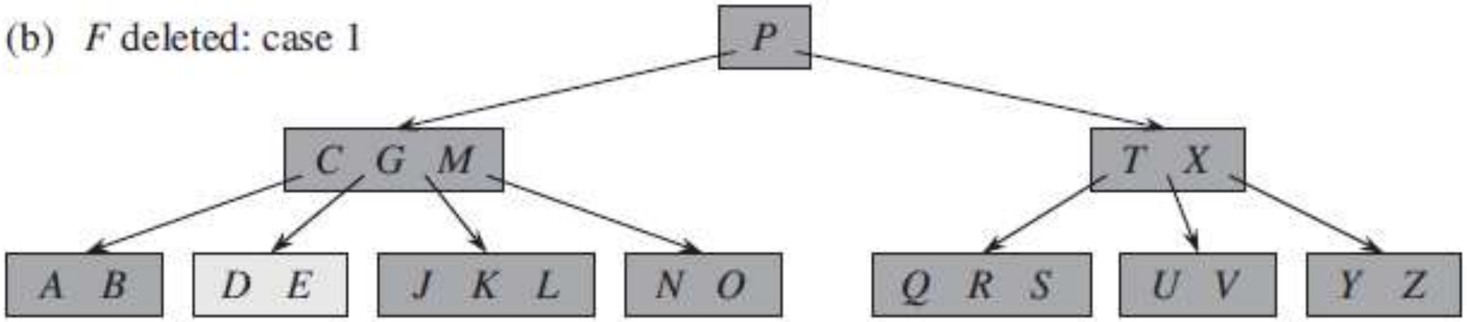
- For a B-tree of height h, B-TREE-INSERT performs O(h) disk accesses, since only O(1) DISK-READ and DISK-WRITE operations occur between calls to B-TREE-INSERT-NONFULL.
- The total CPU time used is $O(t\ h) = O(t \log_t n)$.
- Since B-TREE-INSERT-NONFULL is tail-recursive, we can alternatively implement it as a while loop, thereby demonstrating that the number of pages that need to be in main memory at any time is O(1).

Deleting a key from a B-tree

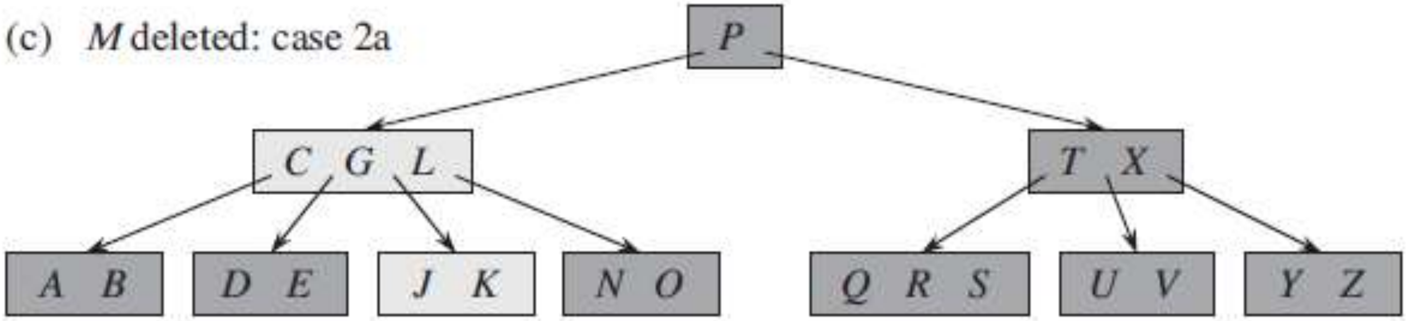
(a) initial tree



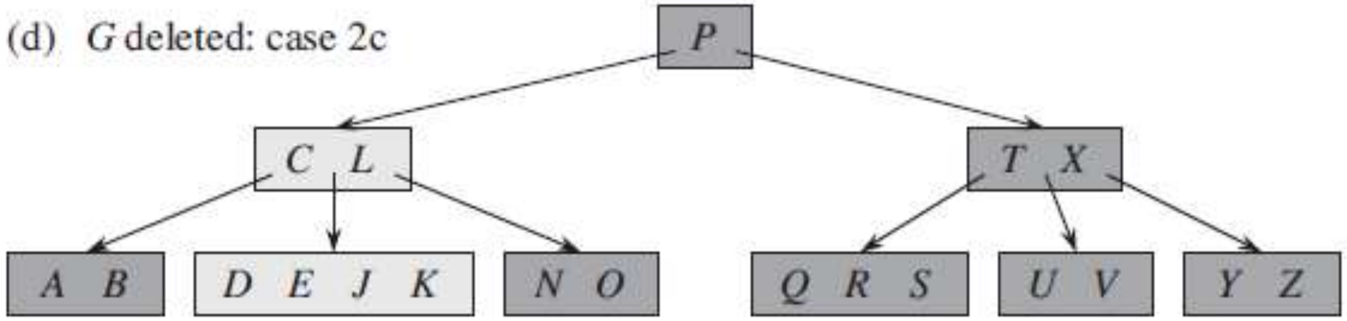
(b) *F* deleted: case 1



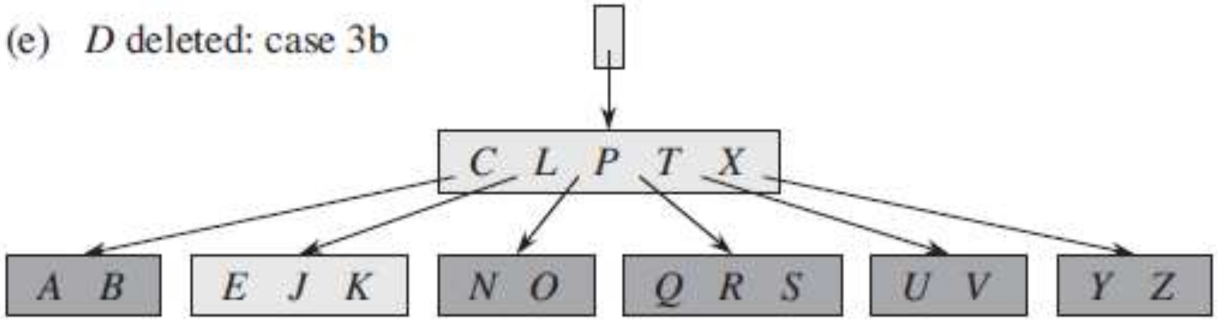
(c) *M* deleted: case 2a



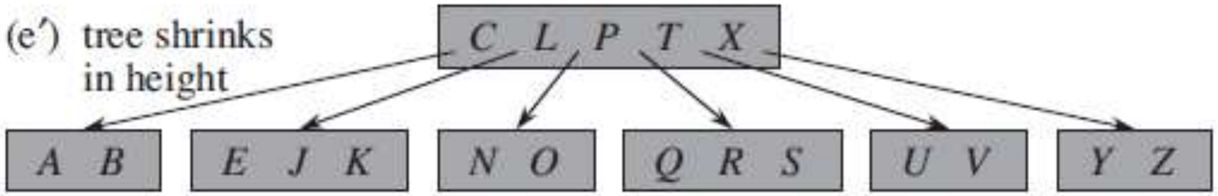
(d) *G* deleted: case 2c



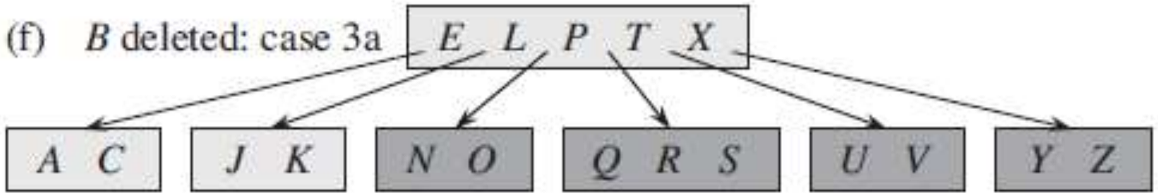
(e) *D* deleted: case 3b



(e′) tree shrinks in height



(f) *B* deleted: case 3a



System of two linear equations in two variables:

$2x + 3y = 4$ … (1)

$5x + 6y = 7$ … (2)

By (1),

$x = (4 - 3y) / 2$ … (3)

Substituting (3) in (2),

$5(4 - 3y) / 2 + 6y = 7$

Solve for y.

And then solve for x using (3) and the value of y.

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

$$\begin{aligned}
a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\
a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\
&\vdots \\
a'_{nn}x_n &= b'_n
\end{aligned}$$

Gaussian Elimination

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\
a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\
&\vdots \\
a'_{nn}x_n &= b'_n
\end{aligned}
$$

In matrix notations, we can write this as

$$Ax = b \quad \Longrightarrow \quad A'x = b'$$

Gaussian Elimination

$$Ax = b \quad \Longrightarrow \quad A'x = b'$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

$$A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix}, \; b' = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

Gaussian Elimination

$$Ax = b \implies A'x = b'$$

How do we efficiently generate *A*` and *b*` from A and b?
We can do that through a series of **elementary operations**.

- exchange two equations of the system
- replacing an equation with its nonzero multiple
- replacing an equation with a sum or difference of this equation and some multiple of another equation

Since no elementary operation can change a solution to a system, any system that is obtained through a series of elementary operations will have the same solution as the original one.

Solve the system by Gaussian elimination.

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0. \end{aligned}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{array}{l} \\ \text{row } 2 - \frac{4}{2} \text{ row } 1 \\ \text{row } 3 - \frac{1}{2} \text{ row } 1 \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{array}{l} \\ \\ \text{row } 3 - \frac{1}{2} \text{ row } 2 \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

$$x_3 = (-2)/2 = -1$$

$$x_2 = (3 - (-3)x_3)/3 = 0$$

$$x_1 = (1 - x_3 - (-1)x_2)/2 = 1$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}\begin{array}{l} \leftarrow \text{pivot} = 1 \\ Row_2 = Row_2 - \frac{4}{2} Row_1 \quad [row = 2,\ col = 1 \text{ to } 4] \\ Row_3 = Row_3 - \frac{1}{2} Row_1 \quad [row = 3,\ col = 1 \text{ to } 4] \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 3/2 & 1/2 & -1/2 \end{bmatrix}\begin{array}{l} \\ \leftarrow \text{pivot} = 2 \\ Row_3 = Row_3 - \frac{3}{2}\frac{1}{3} Row_2 \quad [row = 3,\ col = 2 \text{ to } 4] \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

$$\rightarrow 2x_3 = -2 \Rightarrow x_3 = \frac{-2}{2} \Rightarrow \boxed{x_3 = -1}$$

$$\rightarrow 3x_2 - 3x_3 = 3 \Rightarrow x_2 = \frac{3x_3 + 3}{3} \Rightarrow \boxed{x_2 = 0}$$

$$2x_1 + (-1)x_2 + x_3 = 1 \Rightarrow x_1 = \frac{x_2 - x_3 + 1}{2} \Rightarrow \boxed{x_1 = 1}$$

$$2x_1 - x_2 + 5x_3 = 10$$
$$x_1 + x_2 - 3x_3 = -2$$
$$2x_1 + 4x_2 + x_3 = 1$$

Solve for x_1, x_2, x_3 using Gaussian Elimination.

$$\begin{bmatrix} 2 & -1 & +5 & 10 \\ 1 & 1 & -3 & -2 \\ 2 & 4 & 1 & 1 \end{bmatrix}$$

$$2x_1 - x_2 + 5x_3 = 10$$
$$x_1 + x_2 - 3x_3 = -2$$
$$2x_1 + 4x_2 + x_3 = 1$$

Solve for x_1, x_2, x_3 using Gaussian Elimination.

$$\begin{bmatrix} 2 & -1 & +5 & 10 \\ 1 & 1 & -3 & -2 \\ 2 & 4 & 1 & 1 \end{bmatrix}$$

$\leftarrow$ pivot $= 1$

$R_2 = R_2 - \frac{1}{2}R_1$ $[r = 2,\ c = 1 \text{ to } 4]$

$R_3 = R_3 - \frac{2}{2}R_1$ $[r = 3,\ c = 1 \text{ to } 4]$

$$\begin{bmatrix} 2 & -1 & +5 & 10 \\ 0 & 3/2 & -11/2 & -7 \\ 0 & 5 & -4 & -9 \end{bmatrix}$$

$\leftarrow$ pivot $= 2$

$R_3 = R_3 - \frac{10}{3}R_2$ $[r = 3,\ c = 2 \text{ to } 4]$

$$\begin{bmatrix} 2 & -1 & 5 & 10 \\ 0 & 3/2 & -11/2 & -7 \\ 0 & 0 & 43/3 & 43/3 \end{bmatrix}$$

$\rightarrow \boxed{x_3 = 1}$

$\rightarrow \frac{3}{2}x_2 - \frac{11}{2}x_3 = -7 \Rightarrow \boxed{x_2 = -1}$

$\rightarrow 2x_1 - x_2 + 5x_3 = 10 \Rightarrow \boxed{x_1 = 2}$

```
ALGORITHM GaussianElimination($A[1..n, 1..n], b[1..n]$)
    //Applies Gaussian elimination to matrix $A$ of a system's coefficients,
    //augmented with vector $b$ of the system's right-hand side values
    //Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
    //Output: An equivalent upper-triangular matrix in place of $A$ with the
    //corresponding right-hand side values in the $(n + 1)$st column
    for $i \leftarrow 1$ to $n$ do $A[i, n + 1] \leftarrow b[i]$    //augments the matrix
    for $i \leftarrow 1$ to $n - 1$ do   //pivot row
        for $j \leftarrow i + 1$ to $n$ do   //row index
            for $k \leftarrow i$ to $n + 1$ do   //column index
                $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] \,/\, A[i, i]$
```

1. In the first iteration of k, k=i. So, **`A[j,k]`** and **`A[j,i]`** are same, which will be rendered to zero. In the following iterations of k, we are expecting the older value of **`A[j,i]`**, which is lost.
2. **`A[j,i]/A[i,i]`** is loop invariant in the loop of k.
   a. Instead of a division operation in every iteration of k, we can do it outside of k-loop.
3. When **`A[i,i]`** is zero.
   a. Swap with a row which has a non-zero in i^{th} column.
4. When **`A[i,i]`** is very small, the factor **`A[j,i]/A[i,i]`** could be very large and suffer rounding-off errors.
   a. Swap with a row which has max value in i^{th} column.

```
ALGORITHM  BetterGaussianElimination($A[1..n, 1..n]$, $b[1..n]$)
    //Implements Gaussian elimination with partial pivoting
    //Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
    //Output: An equivalent upper-triangular matrix in place of $A$ and the
    //corresponding right-hand side values in place of the $(n+1)$st column
    for $i \leftarrow 1$ to $n$ do $A[i, n+1] \leftarrow b[i]$ //appends $b$ to $A$ as the last column
    for $i \leftarrow 1$ to $n-1$ do
        $pivotrow \leftarrow i$  //potential pivotrow
        for $j \leftarrow i+1$ to $n$ do
            if $|A[j, i]| > |A[pivotrow, i]|$  $pivotrow \leftarrow j$
        for $k \leftarrow i$ to $n+1$ do
            $swap(A[i, k], A[pivotrow, k])$ // swap pivotrow with ith row
        for $j \leftarrow i+1$ to $n$ do
            $temp \leftarrow A[j, i] / A[i, i]$
            for $k \leftarrow i$ to $n+1$ do
                $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$
```

Time efficiency of Gaussian Elimination:

Basic Operation: $A[j,k] \leftarrow A[j,k] - A[i,k] * temp$

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+2-i) \\
&= \sum_{i=1}^{n-1} (n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\
&= (n+1)(n-1) + n(n-2) + \cdots + 3 \cdot 1
\end{aligned}
$$

Time efficiency of Gaussian Elimination:

Basic Operation: $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$

$$C(n) = (n+1)(n-1) + n(n-2) + \cdots + 3 \cdot 1$$

$$= \sum_{j=1}^{n-1} (j+2)j$$

$$= \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2}$$

$$= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3)$$

Let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ be two sets of numbers. Consider the problem of finding their intersection, i.e., the set C of all the numbers that are in both A and B.

a. Design a brute-force algorithm for solving this problem and determine its efficiency class.

b. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

Consider the problem of finding the distance between the two closest numbers in an array of n numbers. (The distance between two numbers x and y is computed as $|x - y|$.)

a. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

b. Compare the efficiency of this algorithm with that of the brute-force algorithm

You have an array of n real numbers and another integer s. Find out whether the array contains two elements whose sum is s. (For example, for the array 5, $9, 1, 3$ and $s = 6$, the answer is yes, but for the same array and $s = 7$, the answer is no.) Design an algorithm for this problem with a better than quadratic time efficiency.

Red-Black Tree

Transform for good!

</ End of Transform-n-Conquer >