Course – Analysis and Design of Algorithms

UNIT 3:Part 2
Transform-and-Conquer

Unit 3: Transform-and-Conquer

- Presorting
- Heapsort
- Horner’s rule for Polynomial Evaluation

Transform and Conquer Technique: Introduction

- ***Transform-and-conquer*** methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.
- There are three major variations of this idea that differ by what we transform a given instance to:
  - Transformation to a simpler or more convenient instance of the same problem—we call it ***instance simplification***.
  - Transformation to a different representation of the same instance—we call it ***representation change***.
  - Transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***.

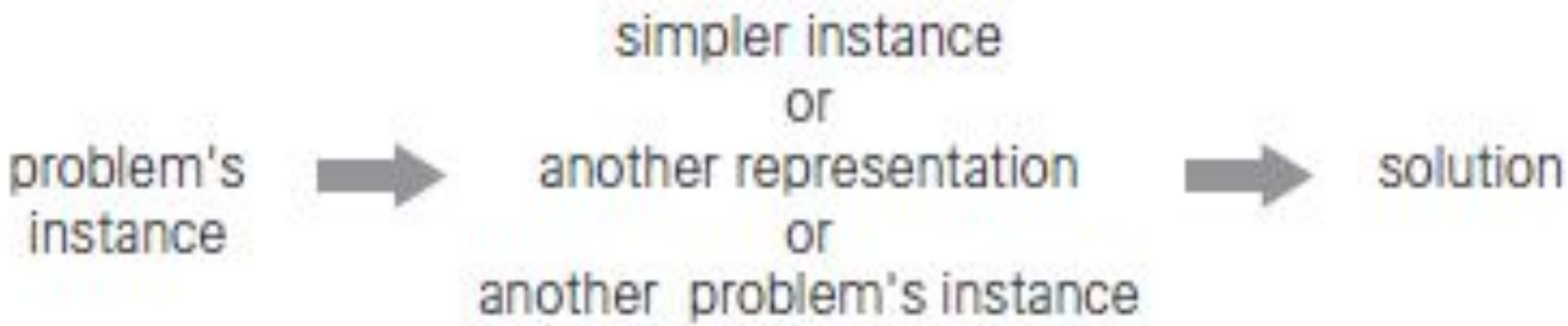


FIGURE Transform-and-conquer strategy.

Presorting

- Presorting is an old idea in computer science.
- Obviously, **the time efficiency of algorithms that involve sorting may depend on the efficiency of the sorting algorithm being used.**
- So far, we have discussed three elementary sorting algorithms—
  i. selection sort,
  ii. bubble sort, and
  iii. insertion sort

  which are quadratic in the worst and average cases,

  And two advanced algorithms
  i. merge sort, which is always in (n log n), and
  ii. quicksort, whose efficiency is also (n log n) in the average case but is quadratic the worst case.

Presorting

- What is Presorting? Why is it necessary?
- Arranging the numbers in ascending order or descending order before solving the actual instance of a problem is called Presorting.
- Following are three examples that illustrate the idea of presorting:
  1. Checking element uniqueness in an array.
  2. Computing a mode.
  3. Searching problem

EXAMPLE 1 *Checking element uniqueness in an array*

- Consider the element uniqueness problem: to check whether all the elements in a given array of n elements are distinct. This problem can be solved by the following straightforward algorithm.

```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] == A[j]return false
return true
```

- This brute-force algorithm compares pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was in $\mathbf{(n^2)}$.

EXAMPLE 1 *Checking element uniqueness in an array*

- Alternatively, we can **sort the array first and then check only its consecutive elements**: if the array has equal elements, a pair of them must be next to each other, and vice versa.
- 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
- The running time of this algorithm is the **sum of the time spent on sorting and the time spent on checking consecutive elements**.

EXAMPLE 1 *Checking element uniqueness in an array*

- The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements.
- The time spent on **sorting requires at least n log n** comparisons and the time spent on **checking consecutive elements needs no more than n − 1 comparisons.**
- Therefore, it is the sorting part that will determine the overall efficiency of the algorithm.
- So, if we use a quadratic sorting algorithm here, the entire algorithm will not be more efficient than the brute-force one.
- But if we use a good sorting algorithm, such as mergesort, with worst-case efficiency in $\theta(n \log n)$, the worst-case efficiency of the entire presorting-based algorithm will be also in $\theta(n \log n)$:

  $T(n) = T_{sort}(n) + T_{scan}(n) \in \theta(n \log n) + \theta(n) = \theta(n \log n)$.

EXAMPLE 2 *Computing a mode*

- A ***mode*** is a value that occurs most often in a given list of numbers. For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5. (If several different values occur most often, any of them can be considered a mode.)
- The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.
- In order to implement this idea, we can store the values already encountered, along with their frequencies, in a separate list.
- On each iteration, the *i*th element of the original list is compared with the values already encountered by traversing this auxiliary list.
- If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with a frequency of 1.

EXAMPLE 2 *Computing a mode*

- It is not difficult to see that the worst-case input for this algorithm is a list with no equal elements.
- For such a list, its *i*th element is compared with $i - 1$ elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of 1.
- As a result, the worst-case number of comparisons made by this algorithm in creating the frequency list is

$$C(n) = \sum_{i=1}^{n} (i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

The additional $n - 1$ comparisons needed to find the largest frequency in the auxiliary list do not change the quadratic worst-case efficiency class of the algorithm.

EXAMPLE 2 *Computing a mode (Contd…)*

- As an alternative, let us first sort the input. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.

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 ← 0                                  //current run begins at position i
modefrequency ← 0                      //highest frequency seen so far
while i ≤ n − 1 do
    runlength ← 1;   runvalue ← A[i]
    while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength;   modevalue ← runvalue
    i ← i + runlength
return modevalue
```

EXAMPLE 2 *Computing a mode (Contd…)*

```
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 ← 0                              //current run begins at position i
    modefrequency ← 0                  //highest frequency seen so far
    while i ≤ n − 1 do
        runlength ← 1;   runvalue ← A[i]
        while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
            runlength ← runlength + 1
        if runlength > modefrequency
            modefrequency ← runlength;   modevalue ← runvalue
        i ← i + runlength
    return modevalue
```

- The running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time.
- Consequently, with an ***(n* log *n)*** sort, this method's worst-case efficiency will be in a better asymptotic class than the worst-case efficiency of the brute-force algorithm.

EXAMPLE 3 *Searching problem*

- Consider the problem of searching for a given value v in a given array of n sortable items.
- The brute-force solution here is sequential search, which needs **n** comparisons in the worst case.
- If the array is sorted first, we can then apply binary search, which requires only $(\log_2 n) + 1$ comparisons in the worst case.
- Assuming the most efficient (n log n) sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

Heaps and Heapsort

- The data structure called the "heap" is definitely not a disordered pile of items as the word's definition in a standard dictionary might suggest.
- Rather, it is a clever, partially ordered data structure that is especially suitable for implementing priority queues.

Properties of Heaps

1. The height of a heap is floor(log n).
2. The root contains the highest priority item.
3. A node and all the descendants is a heap
4. A heap can be implemented using an array.
5. If index of the root = 1 then index of left child = $2i$ and right child = $2i+1$
6. Level i of the heap has 2^i elements
7. Heap order, the parent value is larger than the children(max heap)

Heaps and Heapsort

Notion of the Heap

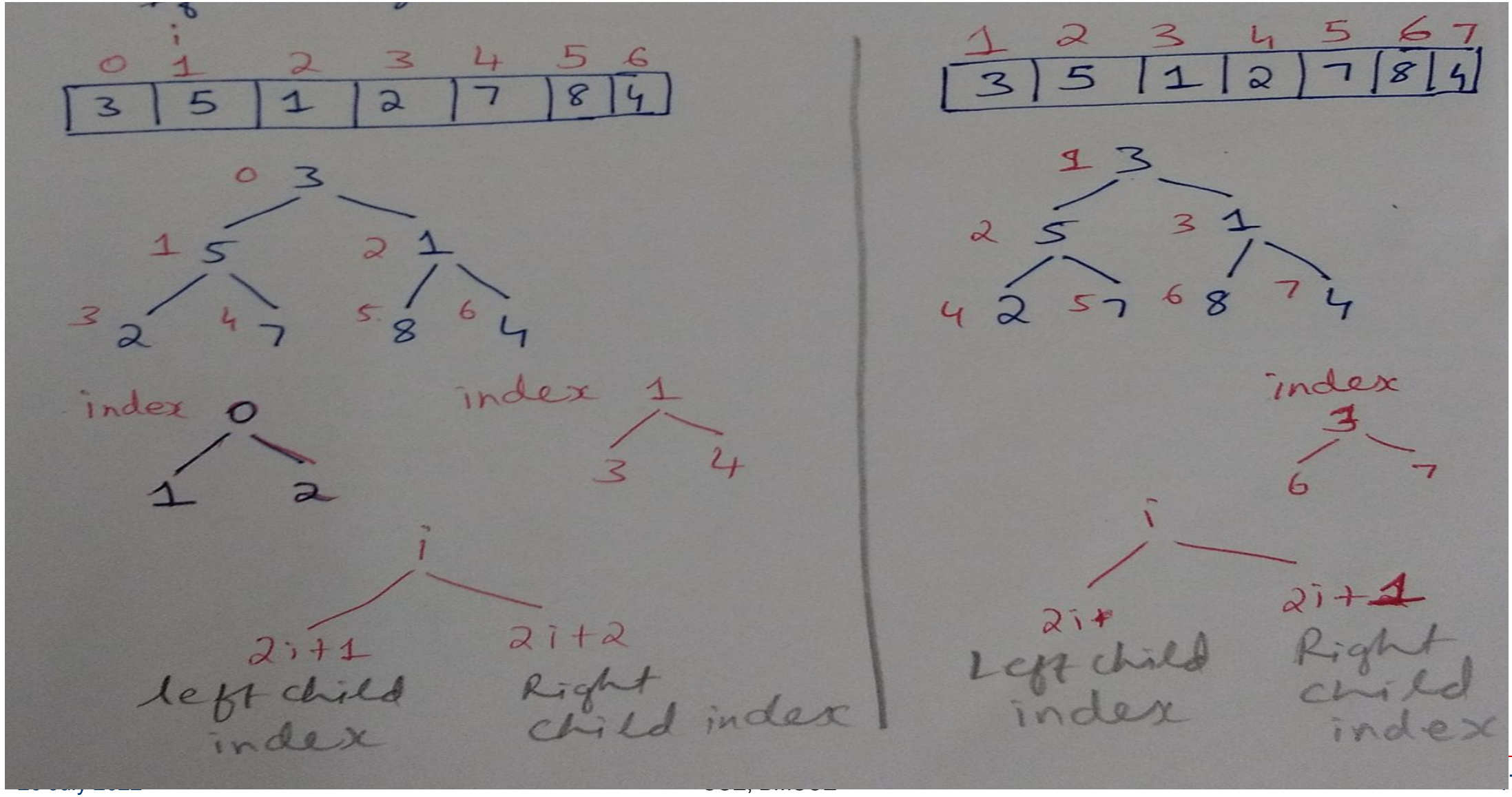
DEFINITION: A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

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.)

FIGURE Heap and its array representation.

Heap and its array representation.



Examples

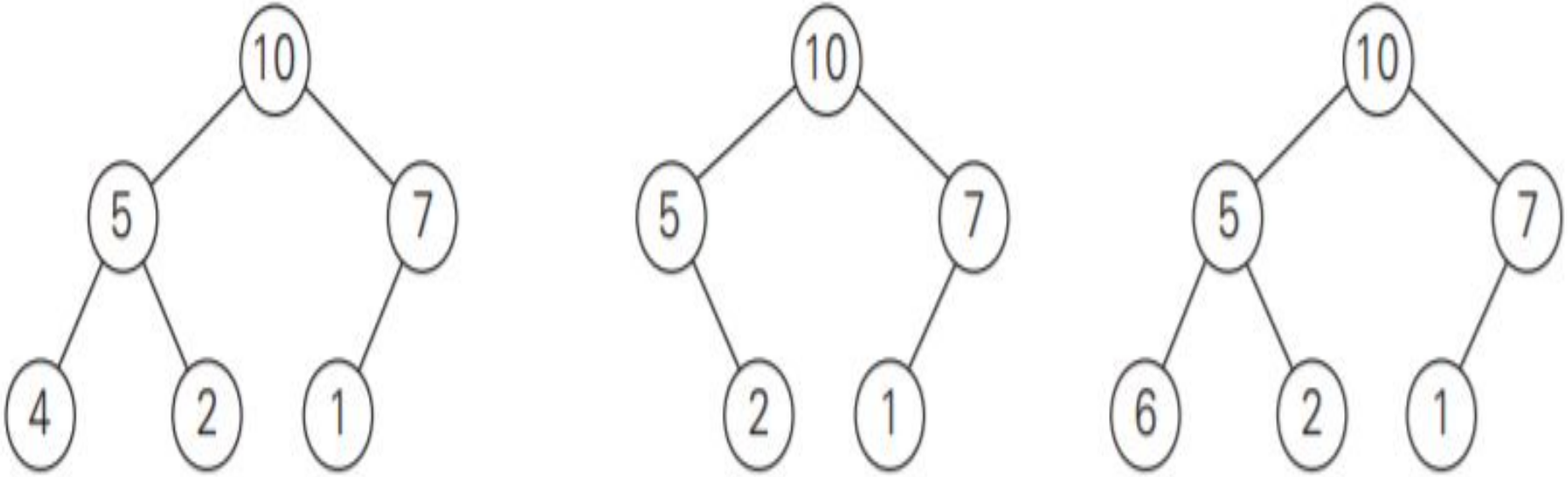
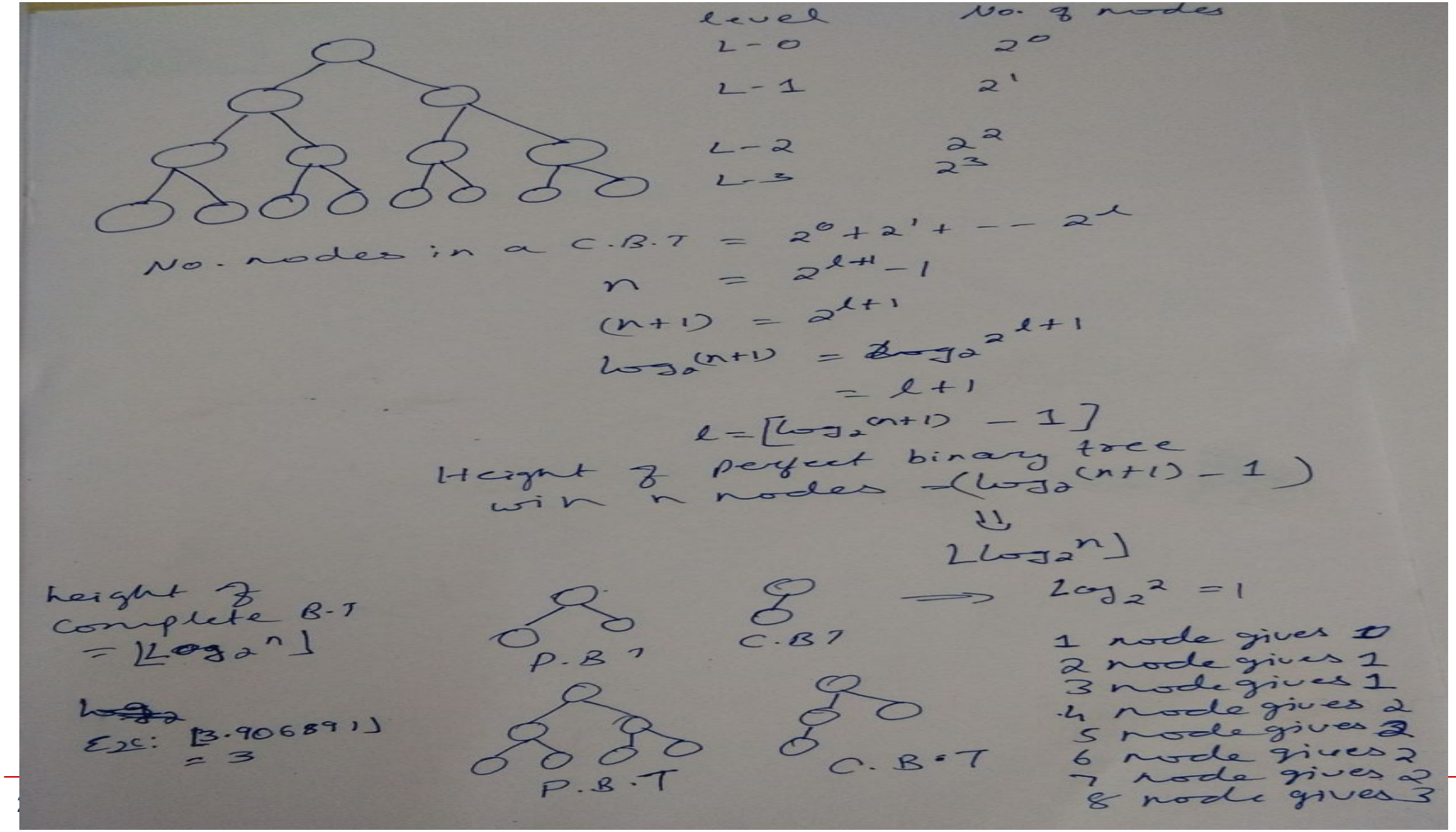


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

Properties of Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. 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;
   b. 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 i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$.

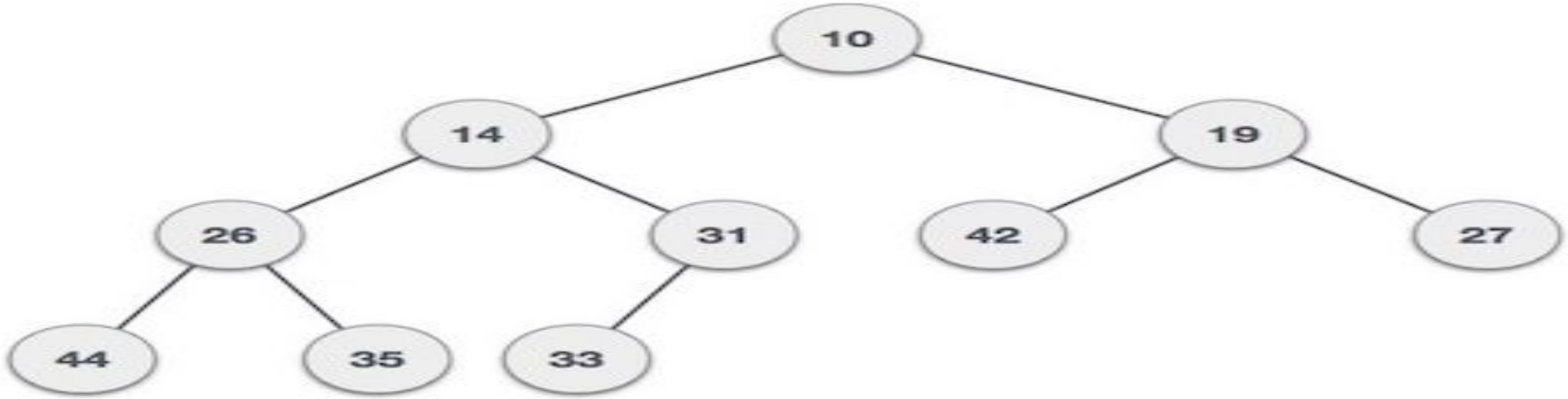
Properties of Heap



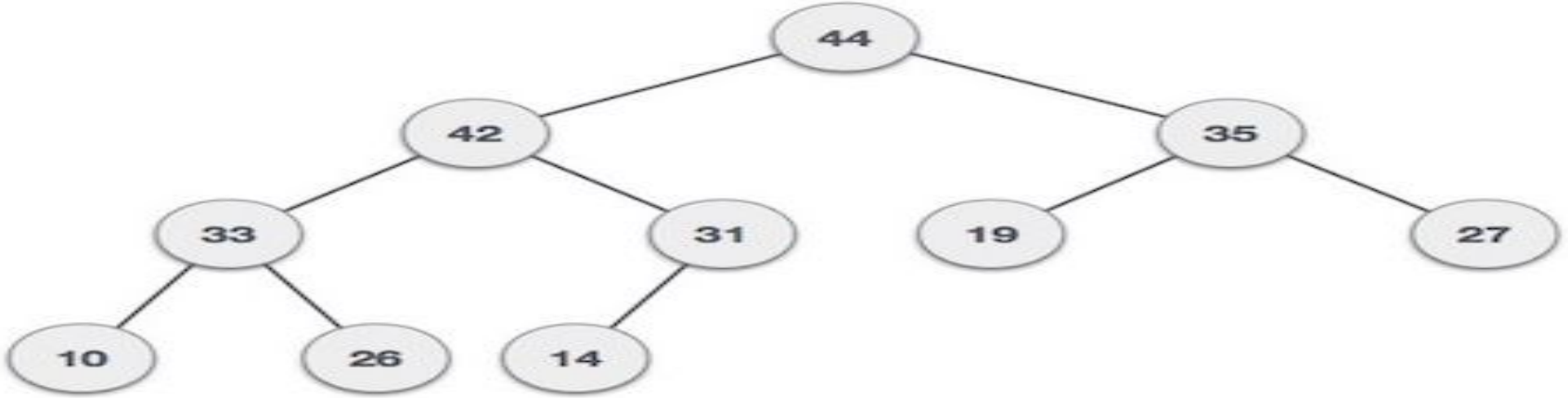
Max heap and Min heap

```
For Input → 35 33 42 10 14 19 27 44 26 31
```

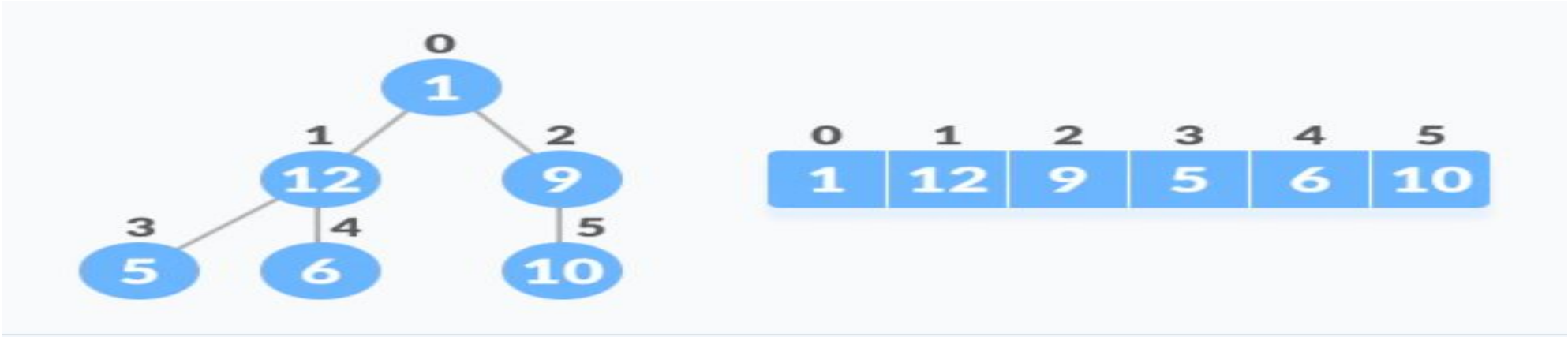
Min-Heap – Where the value of the root node is less than or equal to either of its children.

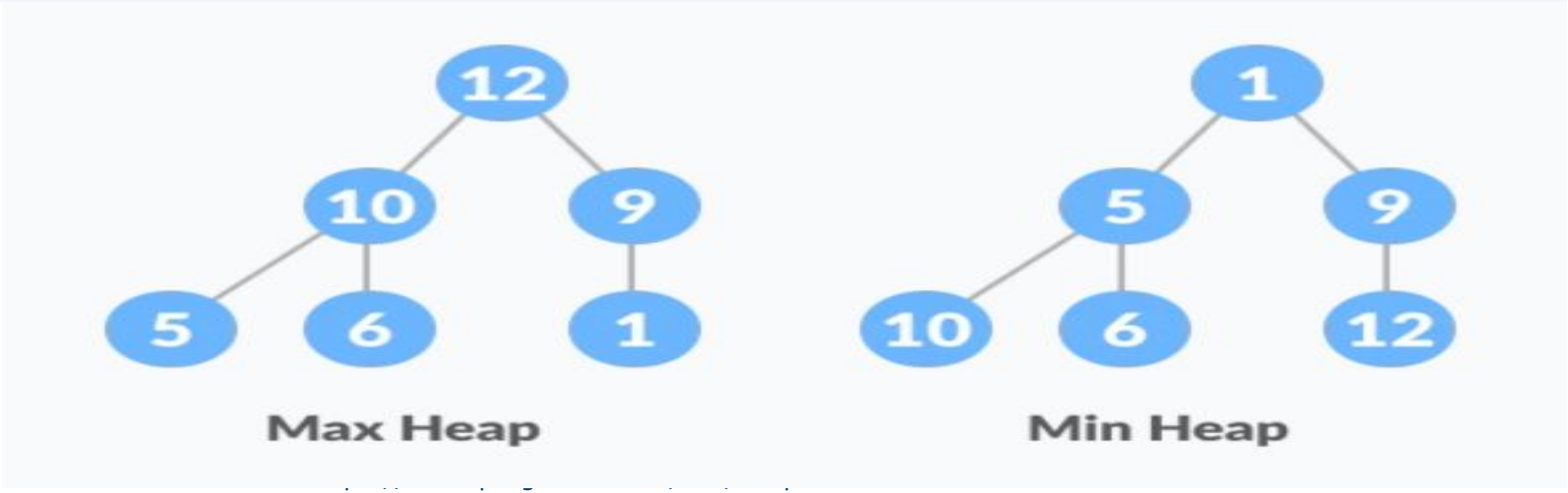


Max-Heap – Where the value of the root node is greater than or equal to either of its children.



Max heap and Min heap





Construction of Heap

- There are two approaches for constructing heap for a given list of keys:
  - Bottom-up heap construction
  - Top-down heap construction
- Bottom-up Approach
  - It initializes essentially the complete binary tree with n nodes by placing the keys in the order given and then "hepifies".

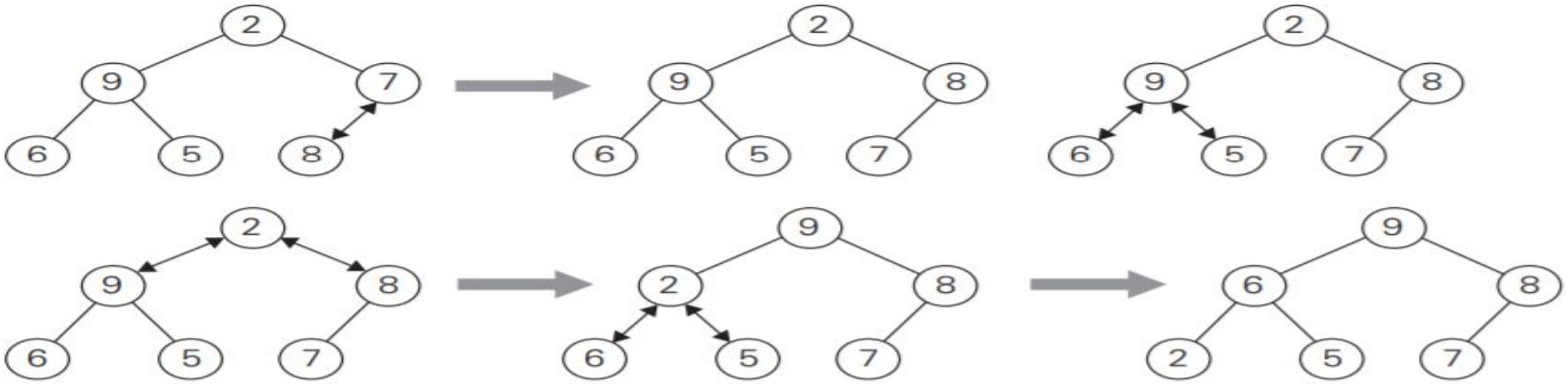


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

Construction of Heap

Top-Down Approach:

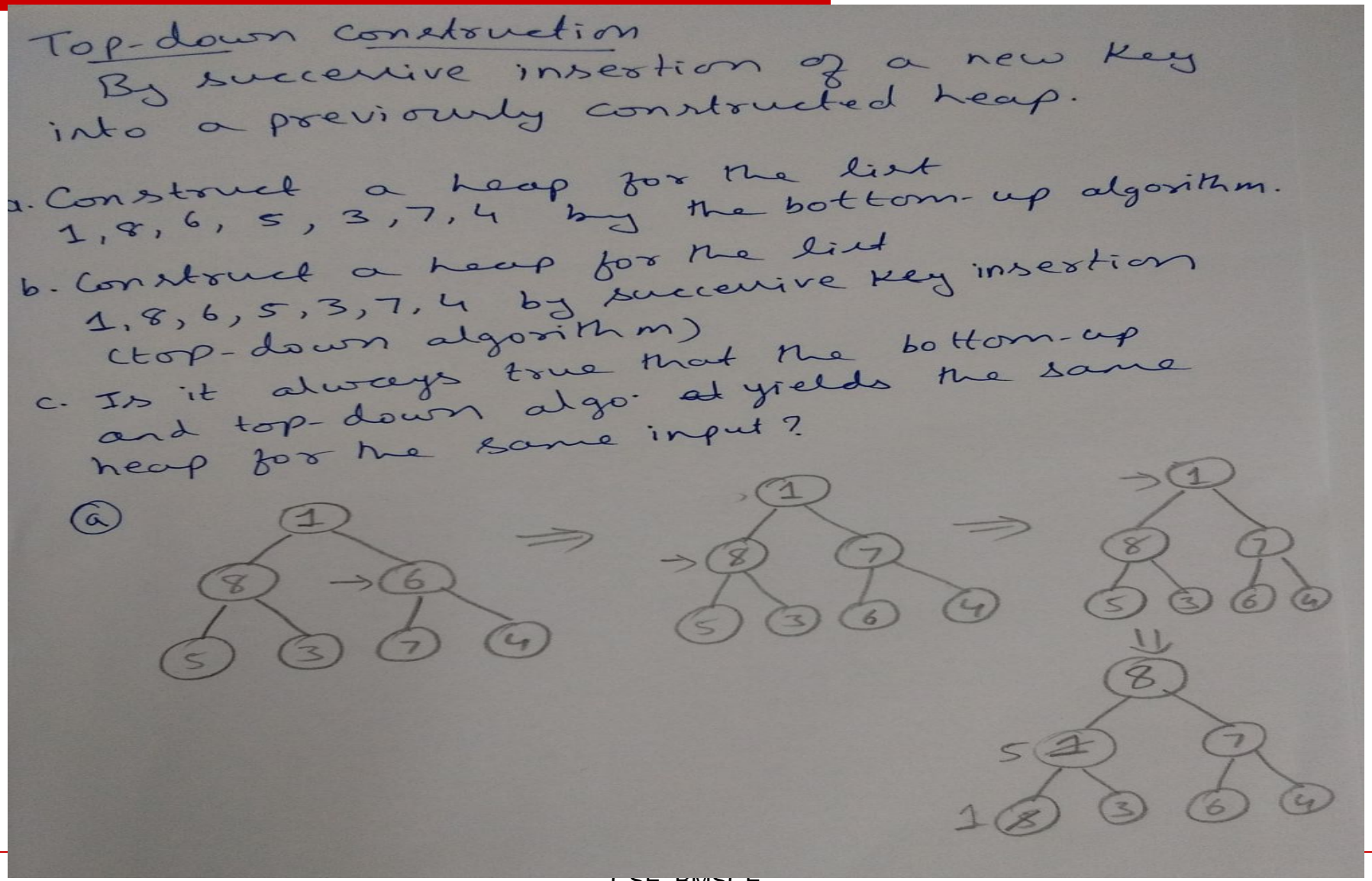
Successive insertion of a new key into a previously constructed heap.

Construct a heap for the list 1,8,6,5,3,7,4 using Top-Down approach.

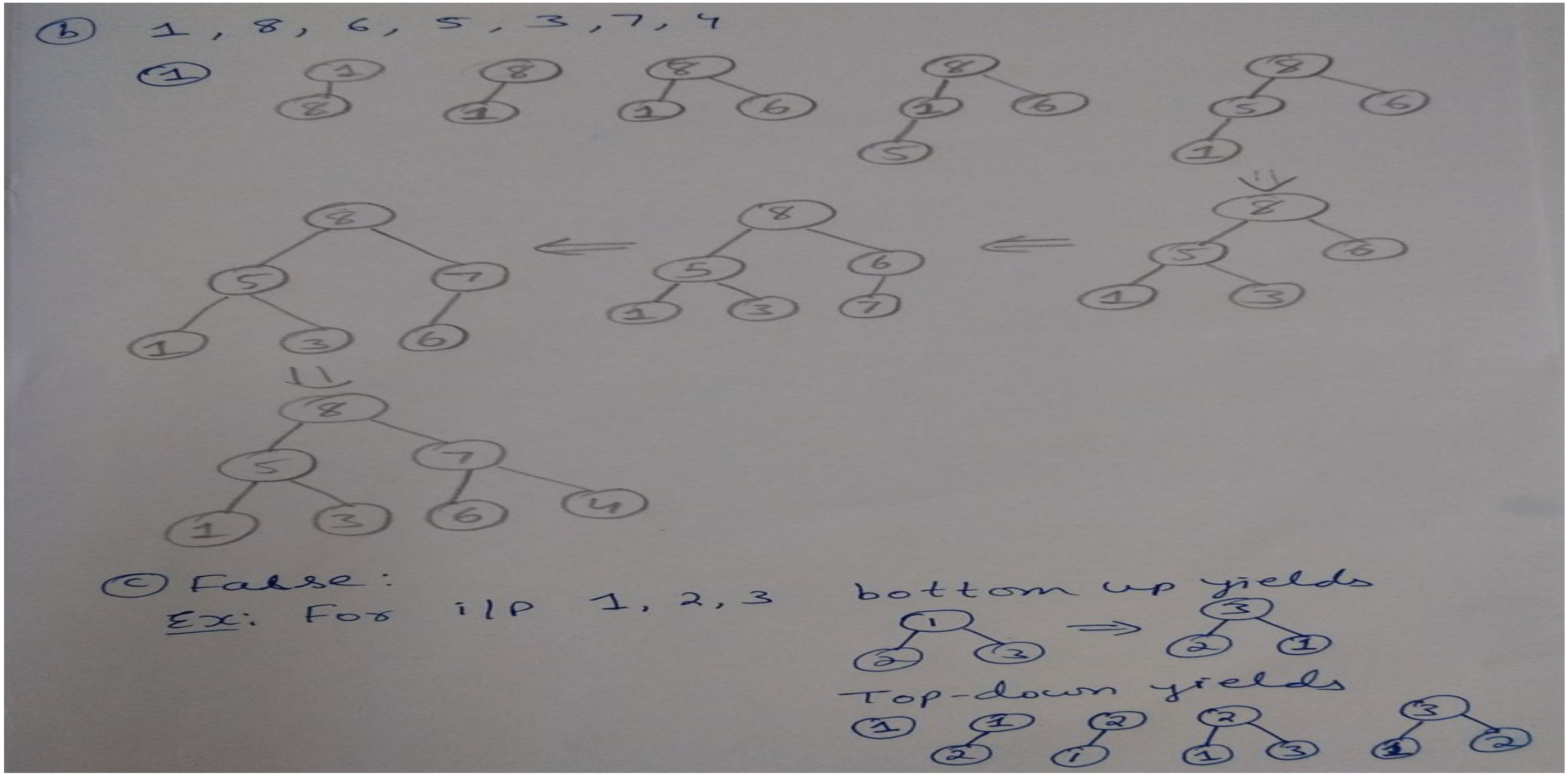
Quiz

- Is it always true that bottom-up and top-down algorithms yield the same heap for the same input?

Construction of Heap



Construction of Heap



Bottom up Heap construction algorithm

```
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 ← ⌊n/2⌋ downto 1 do
        k ← i;   v ← H[k]
        heap ← false
        while not heap and 2 * k ≤ n do
            j ← 2 * k
            if j < n    //there are two children
                if H[j] < H[j + 1]  j ← j + 1
            if v ≥ H[j]
                heap ← true
            else H[k] ← H[j];    k ← j
        H[k] ← v
```

Bottom up Heap construction algorithm

10	20	15	30	40
	40			20

i=5/2=2 k=2 v=20
Heap=false
j=4 4<5 ->two children
30<40 j=5
20>=40
h[2]=40 k=5
H[5]=20

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 ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

10	20	15	30	40
	40			20

Worst case analysis of

How efficient is this algorithm in the worst case? Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of nodes occurs on each level. Let h be the height of the tree. According to the first property of heaps in the list at the beginning of the section, $h = \lfloor \log_2 n \rfloor$ or just $\lceil \log_2 (n+1) \rceil - 1 = k - 1$ for the specific values of n we are considering. Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key on level i will be $2(h - i)$. Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^{h} i2^i$ or by mathematical induction on h. Thus, with this bottom-up algorithm, a heap of size n can be constructed with fewer than $2n$ comparisons.

Insert into a Heap

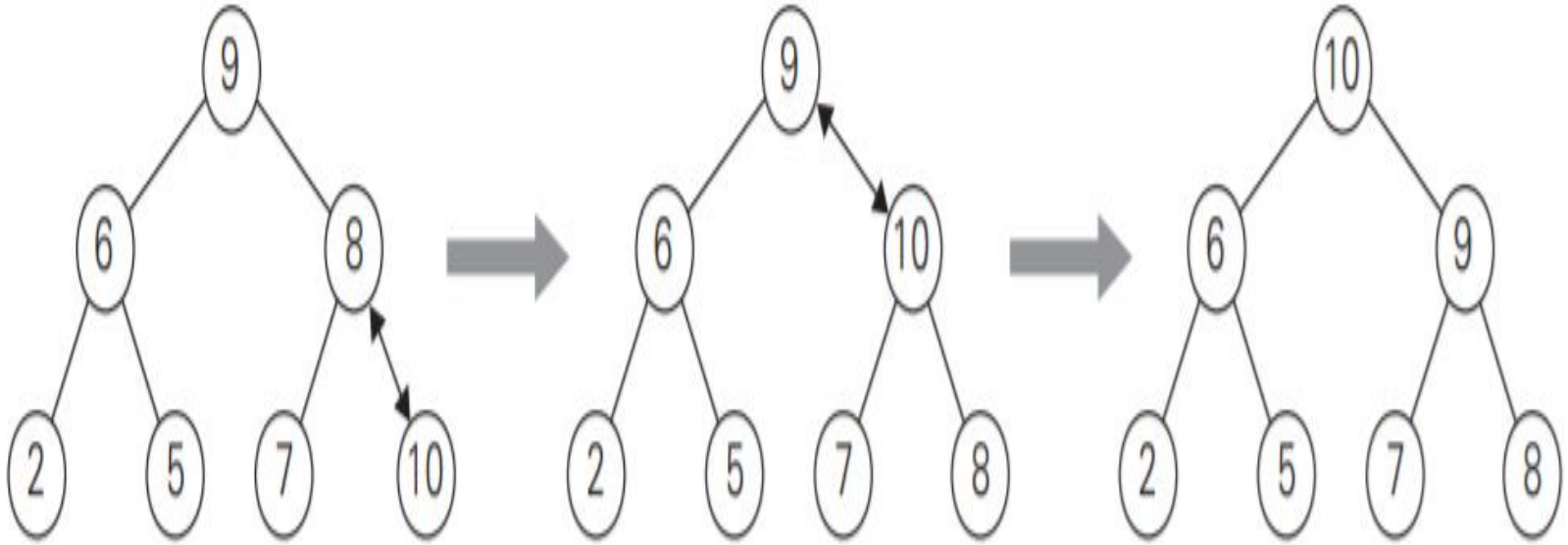


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Delete an element from a Heap

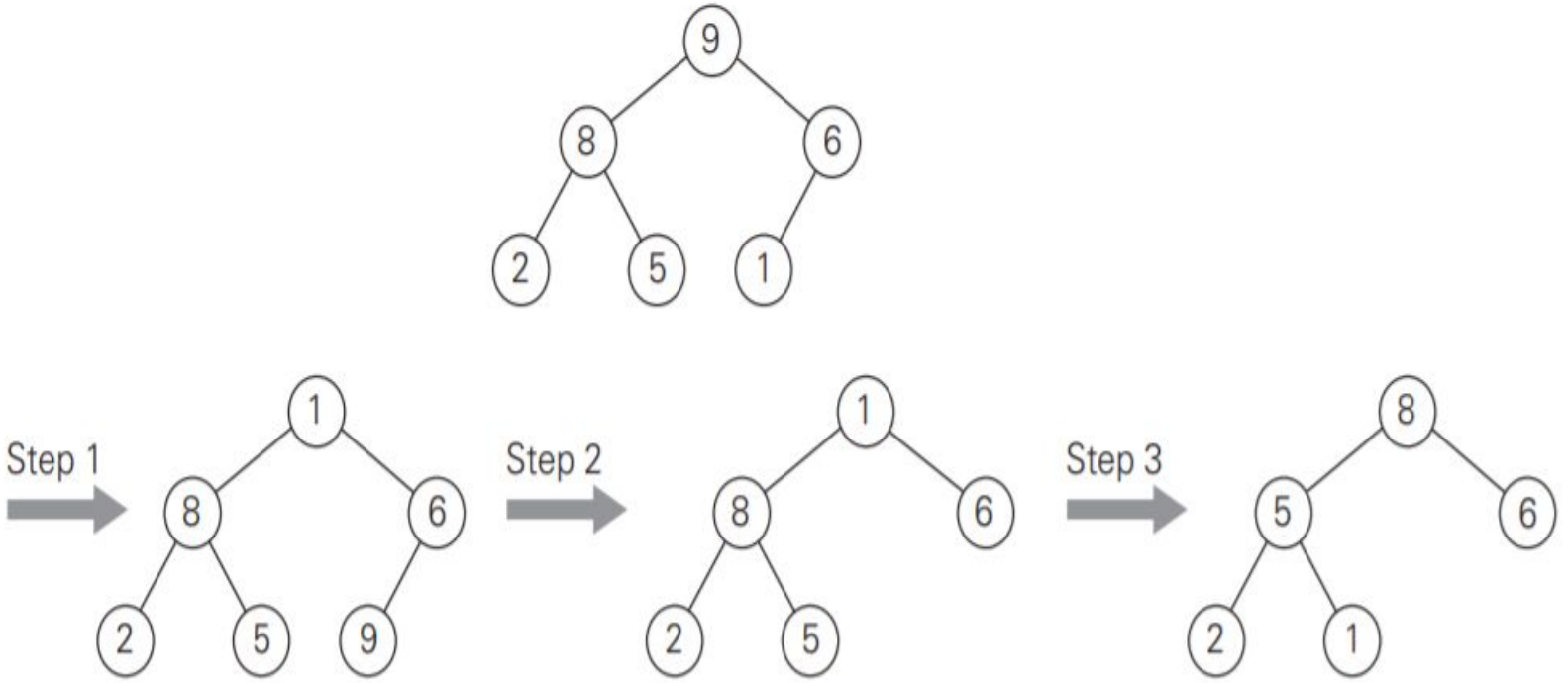
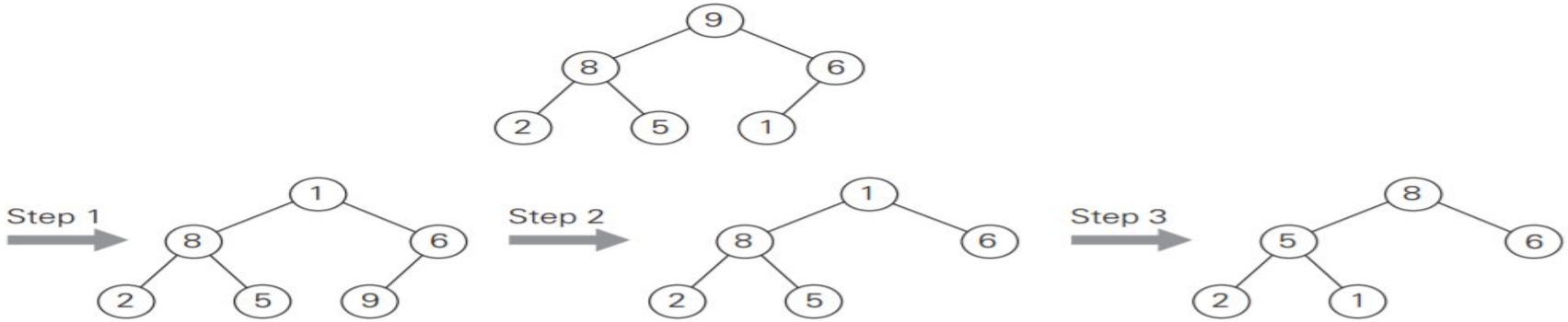


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Delete an element from a Heap



Maximum Key Deletion from a heap

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

Step 2 Decrease the heap's size by 1.

Step 3 "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. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Heap Sort

- Algorithm

1. Heapify
2. Take the first element of the sorted elements
3. Replace the first position with the last element
4. Check if the tree is a Heap?

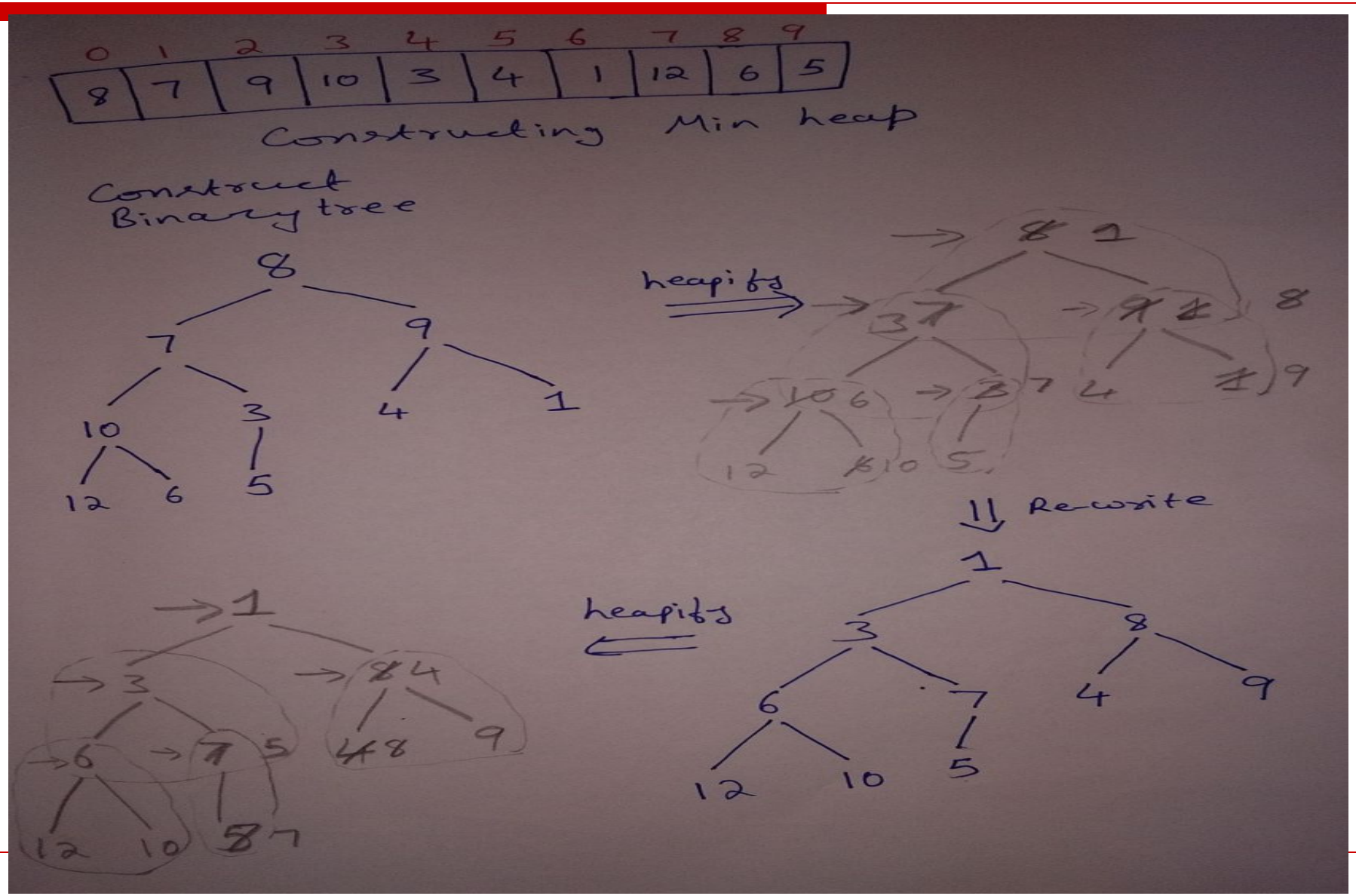
   If YES, go to Step 2

   If NO, go to Step 1

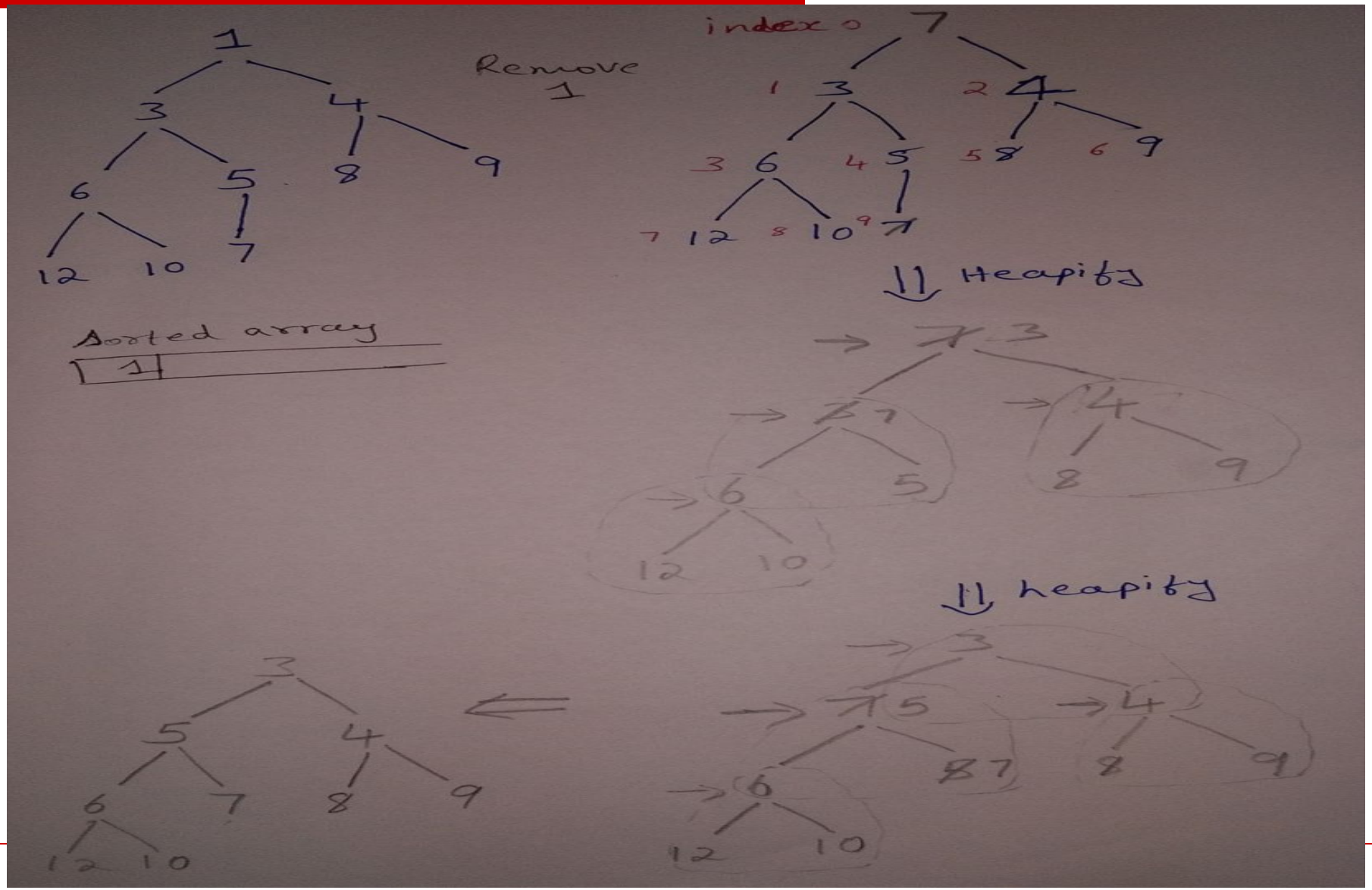
Heapsort: Example

8	7	9	10	3	4	1	12	6	5

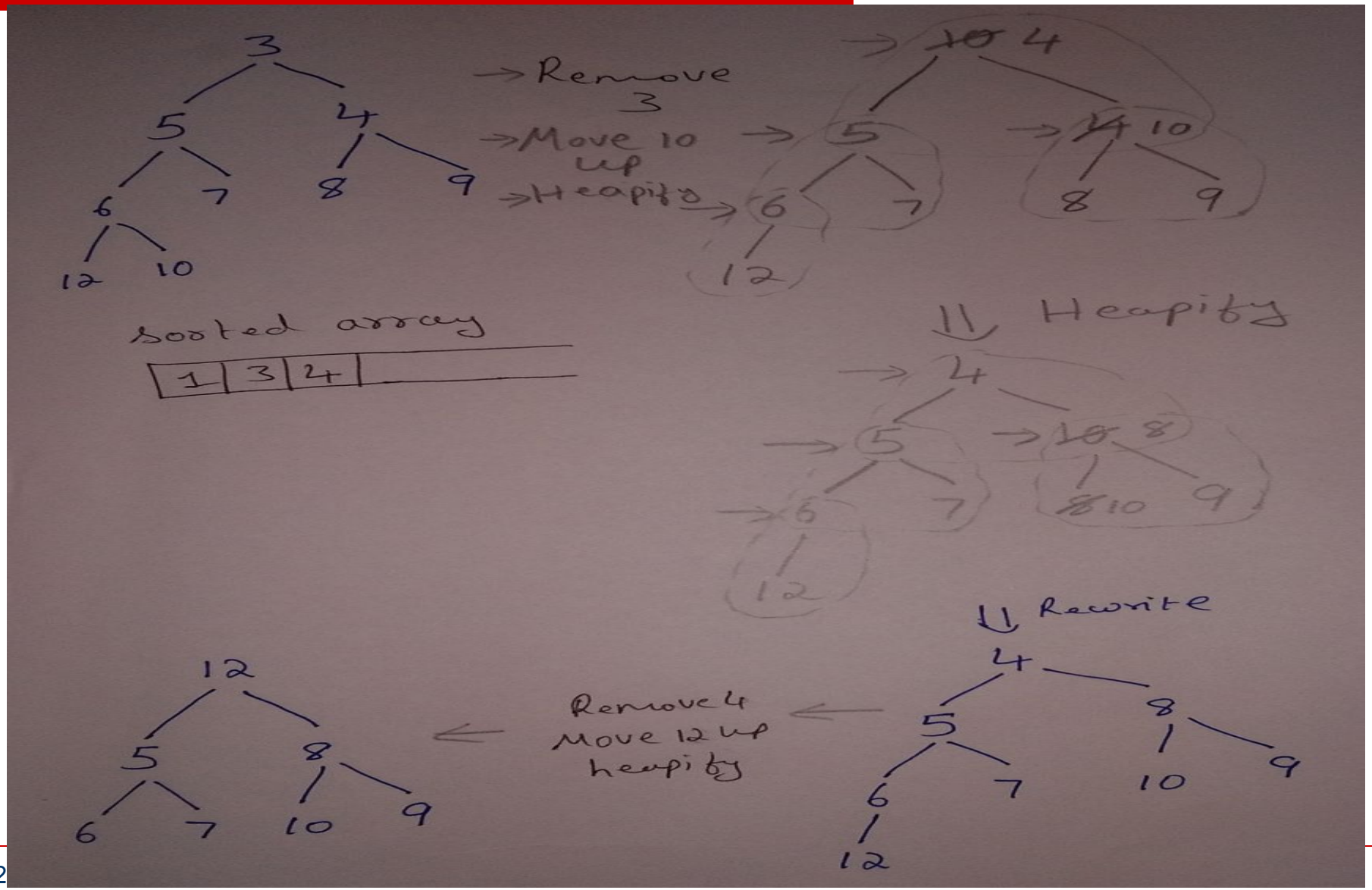
Heapsort: Example



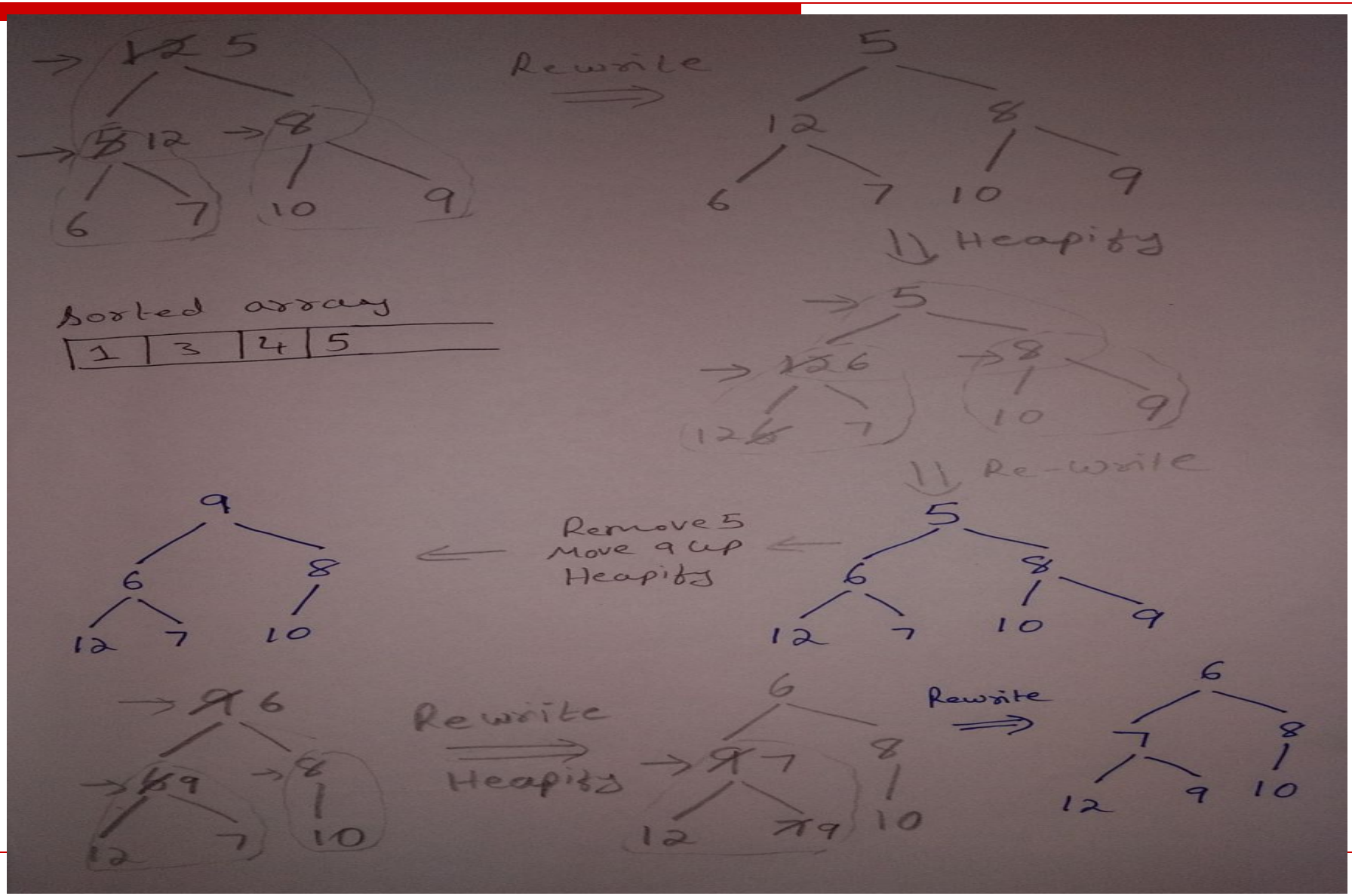
Heapsort: Example



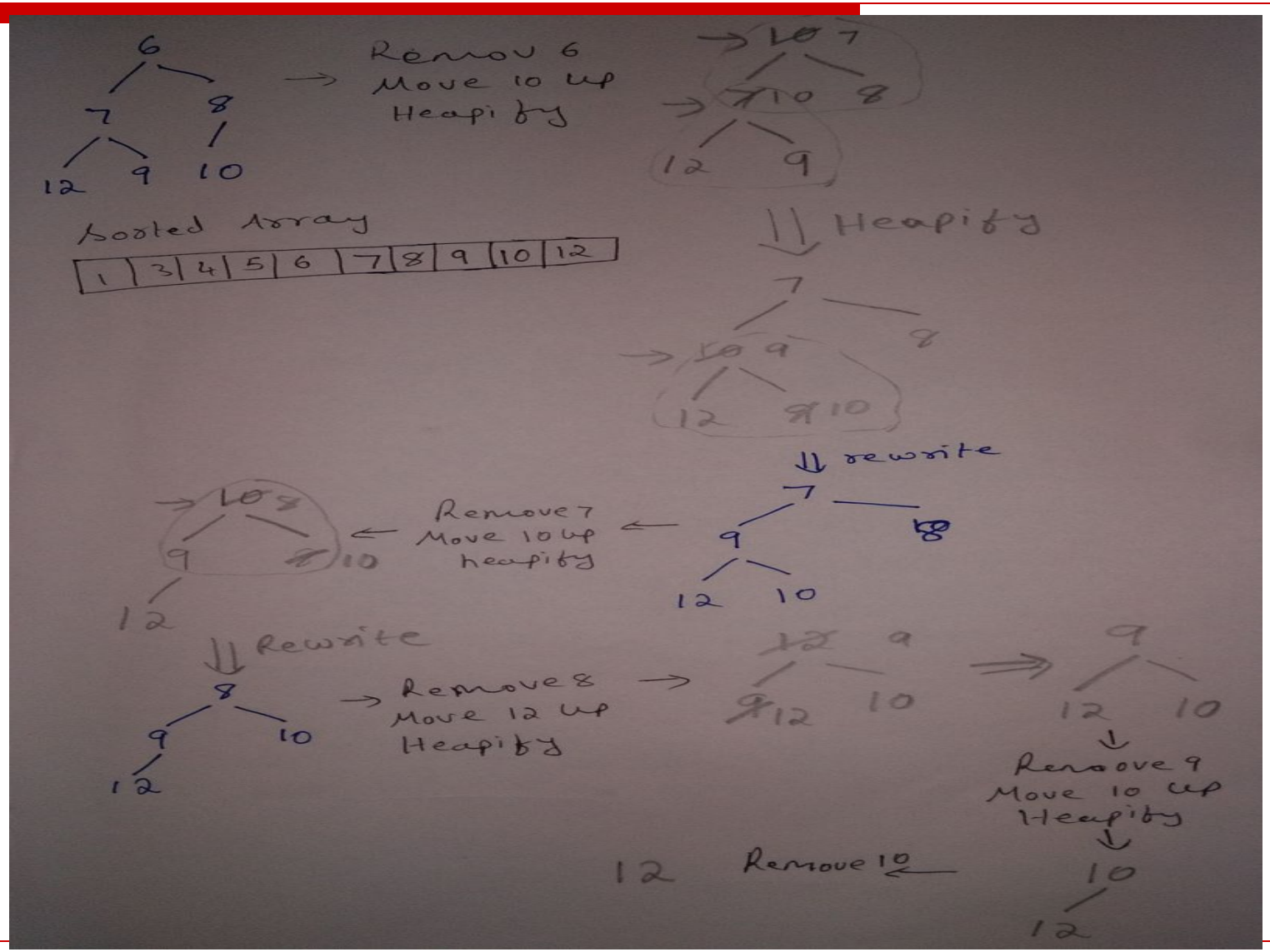
Heapsort: Example



Heapsort: Example



Heapsort: Example



Horner’s rule for Polynomial Evaluation

Horner's rule

- We consider the problem of computing the value of a polynomial at a given point *x*

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \qquad \textbf{(6.1)}$$

- Horner's rule is a good example of the representation-change technique since it is based on representing *p(x)* by a formula different from (6.1).This new formula is obtained from (6.1) by successively taking *x* as a common factor in the remaining polynomials of diminishing degrees:

$$p(x) = (\cdots (a_n x + a_{n-1})x + \cdots)x + a_0. \qquad \textbf{(6.2)}$$

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned}$$

Example

- The first row contains the polynomial's coefficients (including all the coefficients equal to zero, if any) listed from the highest an to the lowest a0.
- Except for its first entry, which is an, the second row is filled left to right as follows: the next entry is computed as the x's value times the last entry in the second row plus the next coefficient from the first row.
- The final entry computed in this fashion is the value being sought.

Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	−1	3	1	−5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

Thus, $p(3) = 160$.

Example

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned}$$

EXAMPLE 1 Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	−1	3	1	−5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

Thus, $p(3) = 160$. (On comparing the table's entries with formula (6.3), you will see that $3 \cdot 2 + (-1) = 5$ is the value of $2x - 1$ at $x = 3$, $3 \cdot 5 + 3 = 18$ is the value of $x(2x - 1) + 3$ at $x = 3$, $3 \cdot 18 + 1 = 55$ is the value of $x(x(2x - 1) + 3) + 1$ at $x = 3$, and, finally, $3 \cdot 55 + (-5) = 160$ is the value of $x(x(x(2x - 1) + 3) + 1) - 5 = p(x)$ at $x = 3$.) ■

Synthetic Division

Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

Thus, $p(3) = 160$.

Horner's rule also has some useful byproducts. The intermediate numbers generated by the algorithm in the process of evaluating $p(x)$ at some point x_0 turn out to be the coefficients of the quotient of the division of $p(x)$ by $x - x_0$, and the final result, in addition to being $p(x_0)$, is equal to the remainder of this division. Thus, according to Example 1, the quotient and the remainder of the division of $2x^4 - x^3 + 3x^2 + x - 5$ by $x - 3$ are $2x^3 + 5x^2 + 18x + 55$ and 160, respectively. This division algorithm, known as ***synthetic division***, is more convenient than so-called long division.

Algorithm

ALGORITHM *Horner*$(P[0..n], x)$

//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n-1$ **downto** 0 **do**
  $p \leftarrow x * p + P[i]$
return p

The number of multiplications and the number of additions are given by the same sum:

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n.$$

Question

a. Apply Horner's rule to evaluate the polynomial

$$p(x) = 3x^4 - x^3 + 2x + 5 \text{ at } x = -2.$$

b. Use the results of the above application of Horner's rule to find the quotient and remainder of the division of $p(x)$ by $x + 2$.

Question

a. Apply Horner's rule to evaluate the polynomial

$$p(x) = 3x^4 - x^3 + 2x + 5 \text{ at } x = -2.$$

b. Use the results of the above application of Horner's rule to find the quotient and remainder of the division of $p(x)$ by $x + 2$.

Answer

a. Evaluate $p(x) = 3x^4 - x^3 + 2x + 5$ at $x = -2$.

coefficients	3	-1	0	2	5
$x = -2$	3	(-2)·3+(-1)= -7	(-2)·(-7)+0=14	(-2)·14+2= -26	(-2)·(-26)+5=57

b. The quotient and the remainder of the division of $3x^4 - x^3 + 2x + 5$ by $x + 2$ are $3x^3 - 7x^2 + 14x - 26$ and 57, respectively.

Question

Consider the following brute-force algorithm for evaluating a polynomial.

Algorithm *BruteForcePolynomialEvaluation*$(P[0..n], x)$
//The algorithm computes the value of polynomial P at a given point x
//by the "highest to lowest term" brute-force algorithm
//Input: An array $P[0..n]$ of the coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at the point x

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do
        power ← power * x
    p ← p + P[i] * power
return p
```

Find the total number of multiplications and the total number of additions made by this algorithm.

Answer

The total number of multiplications made by the algorithm can be computed as follows:

$$M(n) = \sum_{i=0}^{n}(\sum_{j=1}^{i} 1+1) = \sum_{i=0}^{n}(i+1) = \sum_{i=0}^{n} i + \sum_{i=0}^{n} 1$$

$$= \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2} \in \Theta(n^2).$$

The number of additions is obtained as

$$A(n) = \sum_{i=0}^{n} 1 = n+1.$$

Binary Exponentiation

- Two algorithms for computing a^n that are based on the representation-change idea.
  1. ***left-to-right binary exponentiation*** method of computing a^n.
  2. ***right-to-left binary exponentiation*** method of computing a^n.
- They both exploit the binary representation of exponent ***n,*** but one of them processes this binary string left to right, whereas the second does it right to left.

Let

$$n = b_l \ldots b_i \ldots b_0$$

be the bit string representing a positive integer n in the binary number system. This means that the value of n can be computed as the value of the polynomial

$$p(x) = b_l x^l + \cdots + b_i x^i + \cdots + b_0$$

at $x = 2$. For example, if $n = 13$, its binary representation is 1101 and

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

left-to-right binary exponentiation method of computing a^n

ALGORITHM *LeftRightBinaryExponentiation*$(a, b(n))$

//Computes a^n by the left-to-right binary exponentiation algorithm
//Input: A number a and a list $b(n)$ of binary digits $b_l, \ldots, b_0$
// in the binary expansion of a positive integer n
//Output: The value of a^n
$product \leftarrow a$
for $i \leftarrow l - 1$ **downto** 0 **do**
  $product \leftarrow product * product$
  if $b_i = 1$ $product \leftarrow product * a$
return $product$

EXAMPLE Compute a^{13} by the left-to-right binary exponentiation algorithm. Here, $n = 13 = 1101_2$. So we have

binary digits of n	1	1	0	1
product accumulator	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$

right-to-left binary exponentiation method of computing a^n

ALGORITHM *RightLeftBinaryExponentiation*$(a, b(n))$

//Computes a^n by the right-to-left binary exponentiation algorithm
//Input: A number a and a list $b(n)$ of binary digits $b_l, \ldots, b_0$
// in the binary expansion of a nonnegative integer n
//Output: The value of a^n
$term \leftarrow a$ //initializes a^{2^i}
if $b_0 = 1$ $product \leftarrow a$
else $product \leftarrow 1$
for $i \leftarrow 1$ **to** l **do**
  $term \leftarrow term * term$
  if $b_i = 1$ $product \leftarrow product * term$
return $product$

EXAMPLE 3 Compute a^{13} by the right-to-left binary exponentiation method.

Here, $n = 13 = 1101_2$. So we have the following table filled in from right to left:

1	1	0	1	binary digits of n
a^8	a^4	a^2	a	terms a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	product accumulator

Question

a. Apply the left-to-right binary exponentiation algorithm to compute a^{17}.

b. Is it possible to extend the left-to-right binary exponentiation algorithm to work for every nonnegative integer exponent?

Apply the right-to-left binary exponentiation algorithm to compute a^{17}.

Question

a. Compute a^{17} by the left-to-right binary exponentiation algorithm.
Here, $n = 17 = 10001_2$. So, we have the following table filled left-to-right:

binary digits of n	1	0	0	0	1
product accumulator	a	a^2	$(a^2)^2 = a^4$	$(a^4)^2 = a^8$	$(a^8)^2 \cdot a = a^{17}$

b. Algorithm *LeftRightBinaryExponentiation* will work correctly for $n = 0$ if the variable *product* is initialized to 1 (instead of a) and the loop starts with I (instead of $I - 1$).

Compute a^{17} by the right-to-left binary exponentiation algorithm.
Here, $n = 17 = 10001_2$. So, we have the following table filled right-to-left:

1	0	0	0	1	binary digits of n
a^{16}	a^8	a^4	a^2	a	terms a^{2^i}
$a \cdot a^{16} = a^{17}$				a	product accumulator

Thanks for your attention.