Design and Analysis of Algorithms

Unit-1

Dharmendra Kumar(Associate Professor(UCER))

Definition of Algorithm

❖An **algorithm** is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

❖An **algorithm** is thus a sequence of computational steps that transform the input into the output.

❖An **algorithm** is a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it.

Characteristics of an algorithm

- **Input:** Algorithm must contain 0 or more input values.
- **Output:** Algorithm must contain 1 or more input values.
- **Finiteness:** Algorithm must complete after finite number of steps.
- **Definiteness:** Each instruction of the algorithm must be defined precisely or clearly.
- **Effectiveness:** Every instruction must be basic i.e. simple instruction.

Design Strategy of Algorithm

- Divide and Conquer
- Dynamic Programming
- Branch and Bound
- Greedy Approach
- Backtracking
- Randomized Algorithm

Analysis of algorithms

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Time complexity: The amount of time required by an algorithm is called time complexity.

Space complexity: The amount of space/memory required by an algorithm is called space complexity.

Analysis of algorithms

The main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis –

Worst case – The maximum number of steps taken on any instance of size **n**.
Best case – The minimum number of steps taken on any instance of size **n**.
Average case – An average number of steps taken on any instance of size **n**.

Running time of an algorithm

- **Running time of an algorithm is** the time taken by the algorithm to execute it successfully.
- The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed.
- It is convenient to define the notion of step so that it is as machine independent as possible.
- It is denoted by **T(n)**, where n is the input size.

Pseudo code conventions

We use the following conventions in our pseudo code.

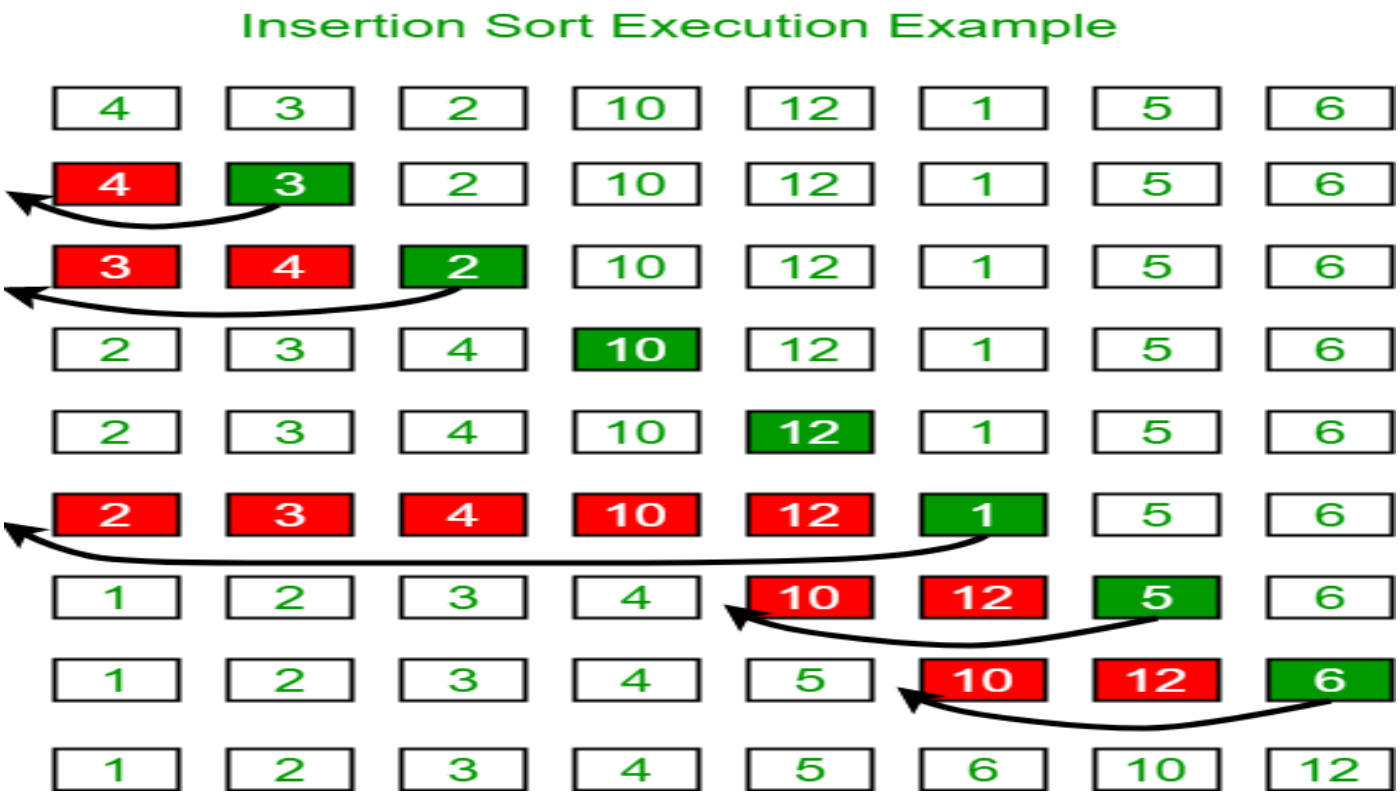
- Indentation is used to indicate a block.
- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C.
- The symbol "//" indicates that the remainder of the line is a comment.
- A multiple assignment of the form $i \leftarrow j \leftarrow e$ assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment $j \leftarrow e$ followed by the assignment $i \leftarrow j$.

Pseudo code conventions

- We access array elements by specifying the array name followed by the index in square brackets. For example, A[i] indicates the ith element of the array A.
- The notation "‥" is used to indicate a range of values within an array. Thus, A[1..j] indicates the sub-array of A consisting of the j elements A[1], A[2],……,A[j].
- The Boolean operators "and" and "or" are used.
- A **return** statement immediately transfers control back to the point of call in the calling procedure.

Insertion Sort

Ex. Sort the following elements by insertion sort:- 4, 3, 2, 10, 12, 1, 5, 6.



Insertion Sort Algorithm

Insertion_Sort(A)

1. n← length[A]
2. for j←2 to n
    1. a←A[j]
    2. //Insert A[j] into the sorted sequence A[1 .. j-1].
    3. i← j-1
    4. While i > 0 and a < A[i]
        1. A[i+1]←A[i]
        2. i← i-1
    5. A[i+1] ← a

Insertion Sort Algorithm Analysis

Insertion_Sort(A)

Instruction	cost	times
n← length[A]	c_1	1
for j←2 to n	c_2	n
a←A[j]	c_3	n-1
//Insert A[j] into the sorted sequence A[1 .. j-1].	0	n-1
i← j-1	c_4	n-1
While i > 0 and a < A[i]	c_5	$\sum_{j=2}^{n} t_j$
A[i+1]←A[i]	c_6	$\sum_{j=2}^{n}(tj\text{-}1)$
i← i-1	c_7	$\sum_{j=2}^{n}(tj\text{-}1)$
A[i+1] ← a	c_8	n-1

Time complexity of Insertion sort

$$T(n) = c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\sum_{j=2}^{n} t_j + c_6.\sum_{j=2}^{n}(tj-1) + c_7.\sum_{j=2}^{n}(tj-1) + c_8.(n-1) \quad \ldots\ldots\ldots\ldots(1)$$

Here, there will be three case.

(1) Best case

(2) Worst case

(3) Average case

Time complexity of Insertion sort

Best case: This case will be occurred when data is already sorted.

In this case, value of t_j will be 1. Put the value of t_j in equation (1) and find T(n). Therefore,

$$T(n) = c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\sum_{j=2}^{n} 1$$
$$+ c_6.\sum_{j=2}^{n}(1-1) + c_7.\sum_{j=2}^{n}(1-1) + c_8.(n-1)$$
$$= (c_2+c_3+c_4+c_5+c_8).n + (c_1 - c_3 - c_4 - c_5 - c_8)$$
$$= an + b$$

Clearly T(n) is in linear form, therefore,

$$\mathbf{T(n) = \theta(n)}$$

Time complexity of Insertion sort

Worst case: This case will be occurred when data is in reverse sorted order.

In this case, value of t_j will be j. Put the value of t_j in equation (1) and find T(n). Therefore,

$$
\begin{aligned}
T(n) &= c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\sum_{j=2}^{n} j \\
&\quad + c_6.\sum_{j=2}^{n}(j-1) + c_7.\sum_{j=2}^{n}(j-1) + c_8.(n-1) \\
&= c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.(n+2).(n-1)/2 \\
&\quad + c_6.n(n-1)/2 + c_7.n(n-1)/2 + c_8.(n-1) \\
&= (c_5+c_6+c_7).n^2/2 + (c_2+c_3+c_4+(c_5-c_6-c_7)/2+c_8).n + \\
&\quad (c_1-c_3-c_4-c_5-c_8) \\
&= an^2 + bn + c
\end{aligned}
$$

Clearly T(n) is in quadratic form, therefore, $\mathbf{T(n) = \theta(n^2)}$

Time complexity of Insertion sort

Average case: This case will be occurred when data is in any order except best and worst case.

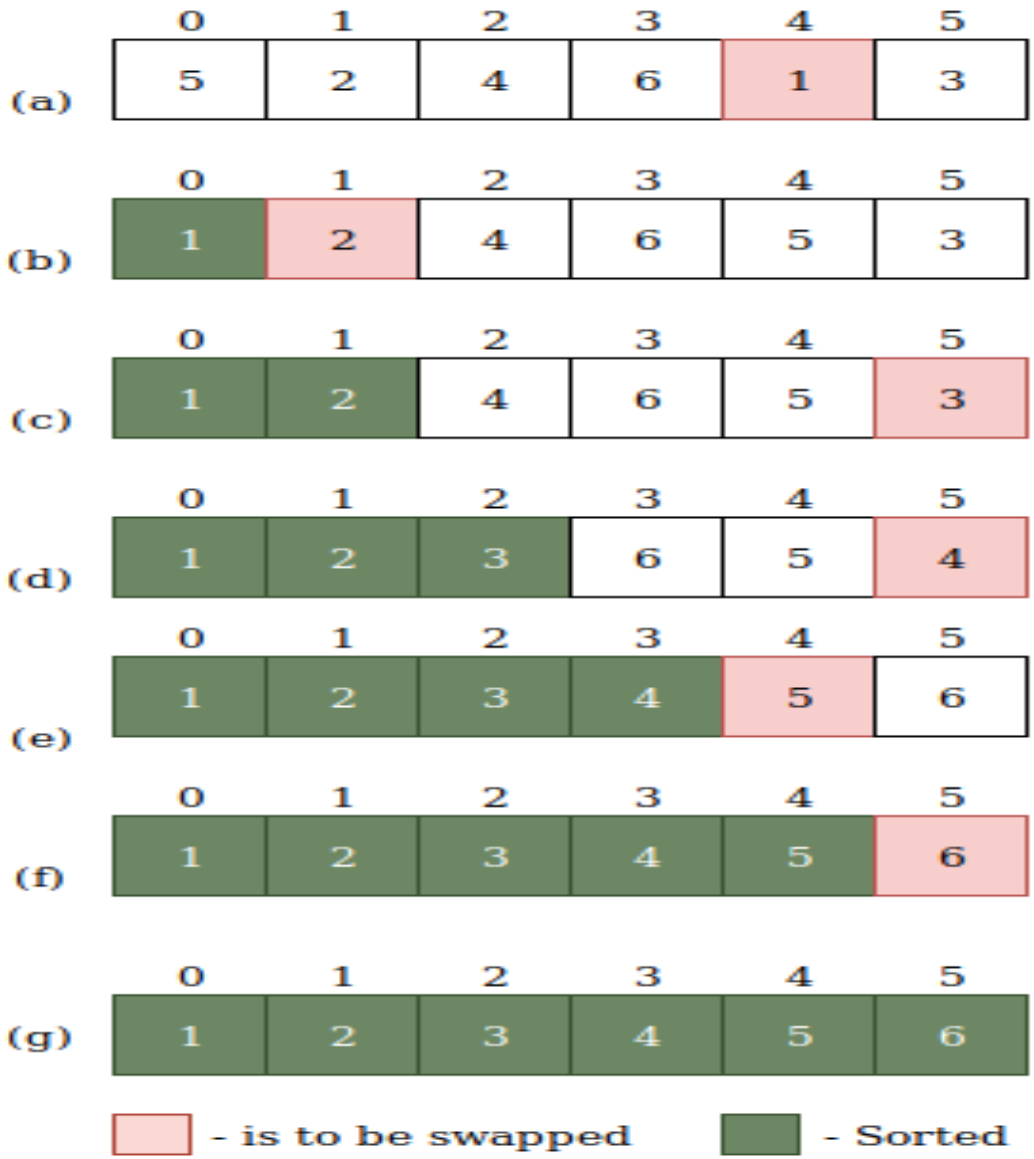
In this case, value of t_j will be j/2. Put the value of t_j in equation (1) and find T(n). Therefore,

$$
\begin{aligned}
T(n) &= c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.\sum_{j=2}^{n} (j/2) \\
&\quad + c_6.\sum_{j=2}^{n}(j/2-1) + c_7.\sum_{j=2}^{n}(j/2-1) + c_8.(n-1) \\
&= c_1.1 + c_2.n + c_3.(n-1) + c_4.(n-1) + c_5.(n+2).(n-1)/4 \\
&\quad + c_6.(n-2)(n-1)/4 + c_7.(n-2)(n-1)/4 + c_8.(n-1) \\
&= (c_5+c_6+c_7).n^2/4 + (c_2+c_3+c_4+(c_5 - 3c_6 - 3c_7)/4+c_8).n + \\
&\quad (c_1 - c_3 - c_4 - c_5/2 + c_6/2 + c_7/2 - c_8) \\
&= an^2 + bn + c
\end{aligned}
$$

Clearly T(n) is in quadratic form, therefore, $\mathbf{T(n) = \theta(n^2)}$

Some other sorting algorithms

- **Selection Sort**



Selection Sort

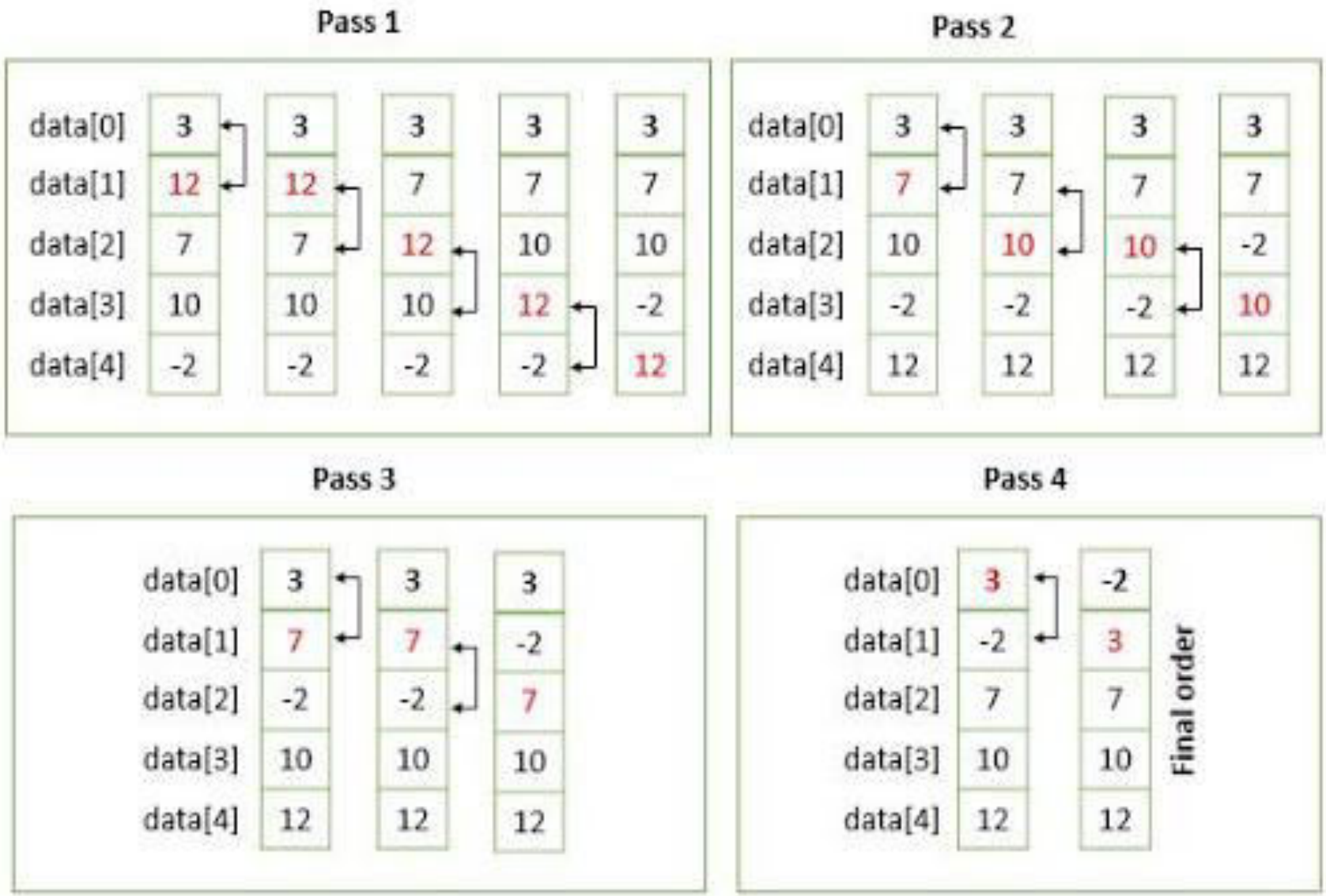
Some other sorting algorithms

```
Selection_sort(A)
n ←length[A]
for i←1 to n-1
    min ← i
    for j←i+1 to n
        if(A[j] < A[min])
            min ←j
    Interchange  A[i]  ↔ A[min]
```

Time complexity, $T(n) = \theta(n^2)$

Some other sorting algorithms

Bubble Sort



Some other sorting algorithms

Bubble _Sort(A)

```
Procedure bubblesort (List array, number length_of_array)
        for i=1 to length_of_array - 1;
                for j=1 to length_of_array – I;
                        if array [ j ] > array [ j+1] then
                                temporary = array [ j+1]
                                array[ j+1] = array [j]
                                array[ j] = temporary
                        end if
                end of j loop
        end of i loop
return array
End of procedure
```

Divide and Conquer approach

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of sub-problems that are smaller instances of the same problem.

Conquer the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub problems in a straightforward manner.

Combine the solutions to the sub-problems into the solution for the original problem.

Analysis of Divide and Conquer based algorithm

When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

Analysis of Divide and Conquer based algorithm

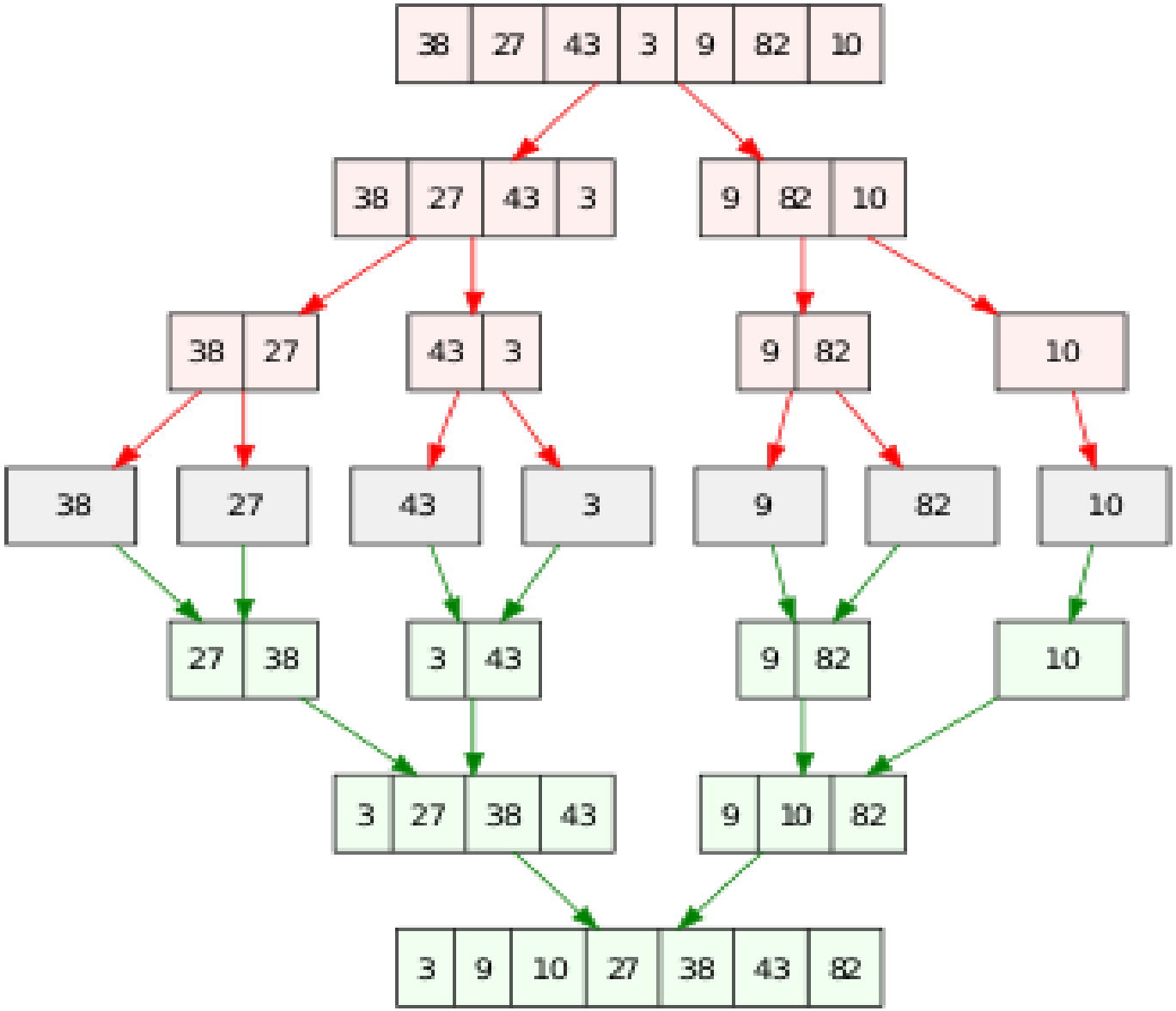
A recurrence equation for the running time of a divide-and-conquer algorithm uses the three steps of the basic paradigm.

The recurrence equation for the running time of a divide-and-conquer algorithm is the following:-

$$
\begin{aligned}
T(n) &= aT(n/b) + D(n) + C(n), && \text{if } n > c \\
&= \theta(1) && , \quad \text{otherwise}
\end{aligned}
$$

Where, n is the size of original problem. a is the number of sub-problems in which the original problem divided at an instant. Each sub-problems has size n/b. c is a small integer.

Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort Algorithm

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Merge Sort Algorithm

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Merge Sort Algorithm Analysis

The recurrence equation for the running time of merge sort algorithm will be

$$T(n) = 2T(n/2) + \theta(1) + \theta(n), \quad \text{if } n > 1$$
$$= \theta(1) \quad , \quad \text{otherwise}$$

It can be modified as:-

$$T(n) = 2T(n/2) + \theta(n) \quad , \quad \text{if } n > 1$$
$$= \theta(1) \quad , \quad \text{otherwise}$$

Merge Sort Algorithm Analysis

When we solve recurrence equation, modify it as:-

$$T(n) = 2T(n/2) + cn, \quad \text{if } n > 1$$
$$= d, \quad \text{otherwise}$$

Here, c and d are some constants.

Merge Sort Algorithm Analysis

Iterative method:

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(2T(n/4) + cn/2) + cn \\
&= 2^2T(n/4) + 2cn \\
&= 2^2(2T(n/8)+cn/4) + 2cn \\
&= 2^3T(n/8)+ 3cn \\
&= \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&= 2^kT(n/2^k) + kcn \\
&= nT(1) + c\,n\log(n) \quad \textbf{(Let } n = 2^k\textbf{)} \\
&= dn + cn\log(n) \quad \textbf{(since } T(1) = d\textbf{)} \\
&= \theta(n\log(n))
\end{aligned}
$$

Therefore, $\mathbf{T(n) = \theta(n\log(n))}$

Asymptotic Notations

- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N={ 0, 1, 2, ……}.
- We will use asymptotic notations to describe the running times of algorithms.
- Following notations are used to define the running time of algorithms.
    1. Θ–notation
    2. O-notation
    3. Ω-notation
    4. o-notation
    5. ω-notation

Asymptotic Notations

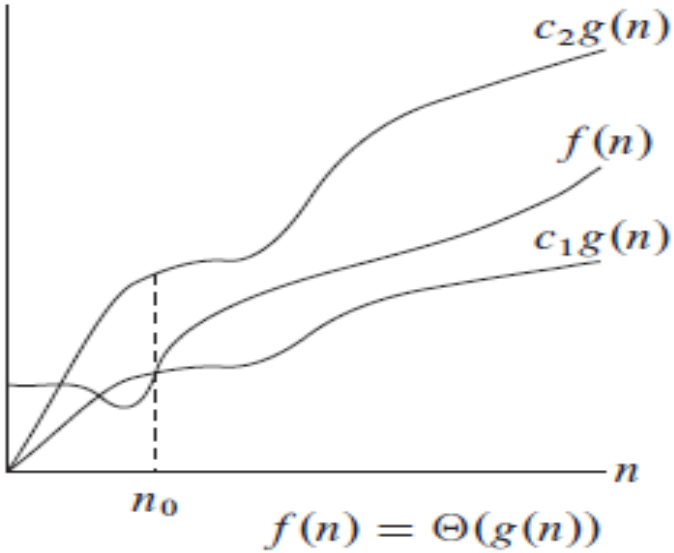
Θ–notation (Theta notation)

- For a given function g(n), it is denoted by Θ(g(n)).
- It is defined as following:-

$\Theta(g(n)) = \{ f(n) \mid \exists$ positive constants c_1, c_2 and n_0 such that

$0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n \ge n_0 \}$

- This notation is said to be tight bound.
- If $f(n) \in \Theta(g(n))$ then $f(n) = \Theta(g(n))$



Asymptotic Notations

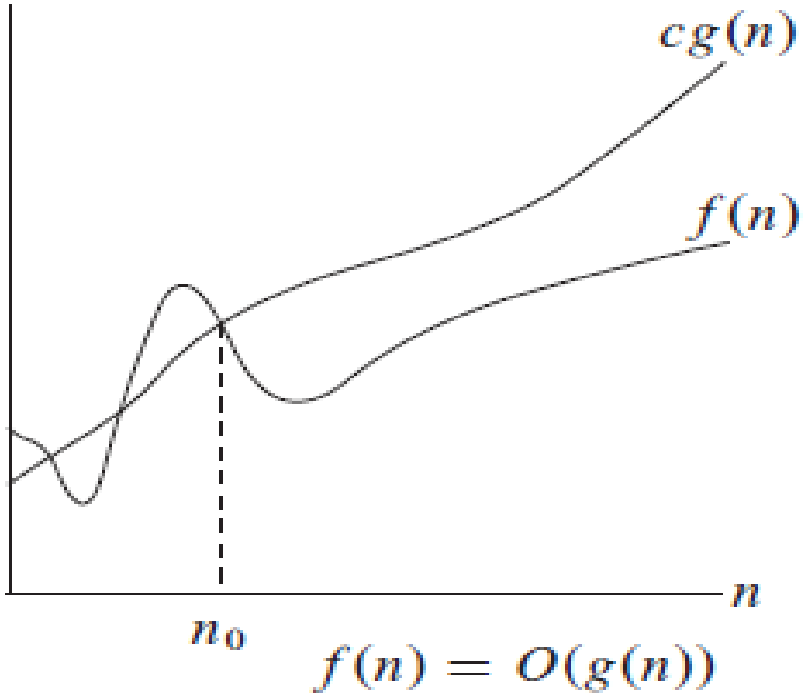
O–notation (Big-oh notation)

- For a given function g(n), it is denoted by O(g(n)).
- It is defined as following:-

$O(g(n)) = \{ f(n) \mid \exists$ positive constants c and n_0 such that

$0 \le f(n) \le cg(n), \forall n \ge n_0 \}$

- This notation is said to be upper bound.
- If $f(n) \in O(g(n))$ then $f(n) = O(g(n))$
- If $f(n) = \Theta(g(n))$ then $f(n) = O(g(n))$



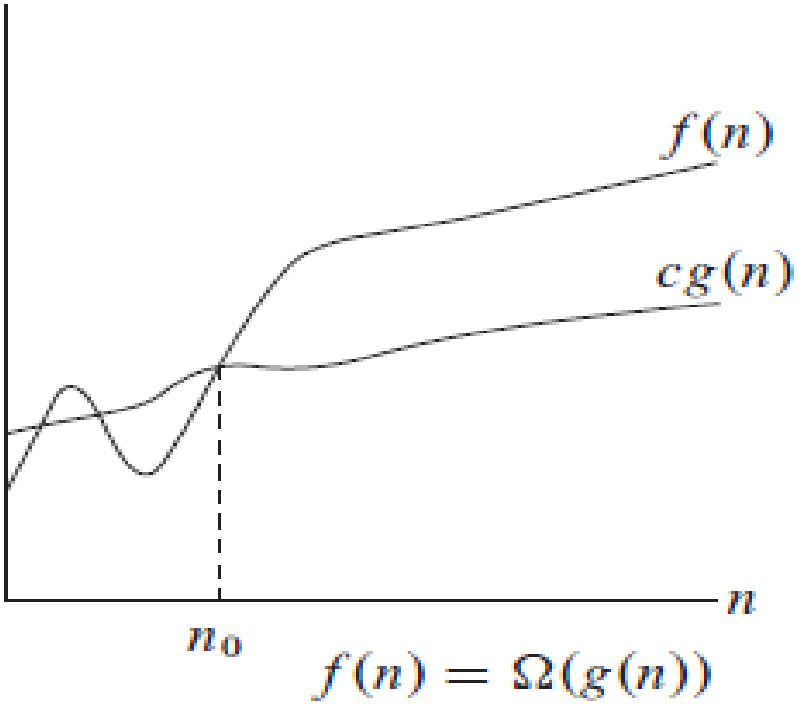
Asymptotic Notations

Ω–notation (Big-omega notation)

- For a given function g(n), it is denoted by $\Omega(g(n))$.
- It is defined as following:-

$\Omega(g(n)) = \{ f(n) \mid \exists$ positive constants c and n_0 such that

$0 \le cg(n) \le f(n), \forall n \ge n_0 \}$



- This notation is said to be lower bound.
- If $f(n) \in \Omega(g(n))$ then $f(n) = \Omega(g(n))$
- If $f(n) = \Theta(g(n))$ then $f(n) = \Omega(g(n))$
- $f(n) = \Theta(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

Asymptotic Notations

o–notation (little-oh notation)

- The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.
- o-notation denotes an upper bound that is not asymptotically tight.
- For a given function g(n), it is denoted by o(g(n)).
- It is defined as following:-

$o(g(n)) = \{ f(n)$! for any positive constants c, there exists a constant n_0 such that

$0 \leq f(n) < cg(n), \forall n \geq n_0 \}$

Asymptotic Notations

ω–notation (little-omega notation)

- The asymptotic lower bound provided by **Ω**-notation may or may not be asymptotically tight.
- **ω**-notation denotes an upper bound that is not asymptotically tight.
- For a given function g(n), it is denoted by **ω**(g(n)).
- It is defined as following:-

ω(g(n)) = { f(n) ! for any positive constants c, there exists a constant n_0 such that

$0 \leq cg(n) < f(n), \forall\ n \geq n_0$ }

Asymptotic Notations

Example: Show that $(1/2)n^2 - 3n = \theta(n^2)$.

Solution: Using definition of θ-notation,

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall\, n \geq n_0$$

In this question, $f(n) = (1/2)n^2 - 3n$ and $g(n) = n^2$, therefore

$$c_1\, n^2 \leq (1/2)n^2 - 3n \leq c_2\, n^2, \quad \forall\, n \geq n_0$$

We divide above by n^2, we get

$$c_1 \leq (1/2) - (3/n) \leq c_2\,, \quad \forall\, n \geq n_0 \quad \ldots\ldots\ldots\ldots\ldots\ldots(1)$$

Now, we have to find c_1, c_2 and n_0, such that equation (1) is satisfied.

Consider, left part of (1), $c_1 \leq (1/2) - (3/n)$ …… (2)

The value of c_1 will be positive value less than or equal to the minimum value of $(1/2) - (3/n)$. Minimum value of $(1/2) - (3/n) = 1/14$. Therefore, $c_1 = 1/14$. This value of c_1 will satisfy equation (2) for $n \geq 7$.

Here, $c_1 = 1/14$ and $n \geq 7$ which satisfy (2).

Asymptotic Notations

Consider, right part of (1), $(1/2) - (3/n) \leq c_2$, ... (3)

The value of c_2 will be positive value greater than or equal to the maximum value of $(1/2) - (3/n)$. Maximum value of $(1/2) - (3/n) = 1/2$. Therefore, $c_2 = 1/2$. This value of c_2 will satisfy equation (3) for $n \geq 1$.

Here, $c_2 = 1/2$ and $n \geq 1$ which satisfy (3).

Therefore, for $c_1 = 1/14$, $c_2 = 1/2$ and $n_0 = 7$, equation (1) is satisfied.

Hence by using definition of θ-notation ,

$$(1/2)n^2 - 3n = \theta(n^2).$$

It is proved.

Asymptotic Notations

Example: Show that $2n+5 = O(n^2)$.

Solution: Using definition of O-notation,

$$f(n) \leq cg(n), \forall n \geq n_0$$

In this question, $f(n) = 2n+5$ and $g(n) = n^2$, therefore

$$2n+5 \leq c\, n^2 \quad \forall n \geq n_0$$

We divide above by n^2, we get

$$(2/n)+(5/n^2) \leq c, \quad \forall n \geq n_0 \quad \ldots\ldots\ldots(1)$$

Now, we have to find c and n_0, such that equation (1) is satisfied.

Asymptotic Notations

The value of c will be positive value greater than or equal to the maximum value of $(2/n)+(5/n^2)$.

Maximum value of $(2/n)+(5/n^2) = 7$.

Therefore, $c = 7$.

Clearly equation (1) is satisfied for $c = 7$ and $n \geq 1$.

Hence by using definition of O-notation ,

$$2n+5 = O(n^2).$$

It is proved.

Asymptotic Notations

Example: Show that $2n^2+5n+6 = \mathbf{\Omega}(n)$.

Solution: Using definition of $\mathbf{\Omega}$ -notation,

$$cg(n) \leq f(n)\ ,\ \forall\ n \geq n_0$$

In this question, $f(n) = 2n^2+5n+6$ and $g(n) = n$, therefore

$$cn \leq 2n^2+5n+6\ ,\quad \forall\ n \geq n_0$$

We divide above by n, we get

$$c \leq 2n + 5 + (6/n)\ ,\ \forall\ n \geq n_0 \quad \ldots\ldots\ldots(1)$$

Now, we have to find c and n_0, such that equation (1) is always satisfied.

Asymptotic Notations

The value of c will be positive value less than or equal to the minimum value of $2n + 5 + (6/n)$.

Minimum value of $2n + 5 + (6/n) = 12$.

Therefore, $c = 12$.

Clearly equation (1) is satisfied for $c = 12$ and $n \geq 2$.

Hence by using definition of $\mathbf{\Omega}$ -notation ,

$$2n^2+5n+6 = \mathbf{\Omega}(n).$$

It is proved.

Asymptotic Notations

Example: Show that $2n^2 = o(n^3)$.

Solution: Using definition of o-notation,

$$f(n) < cg(n) \quad , \forall\ n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n^3$. Therefore,

$$2n^2 < cn^3 , \qquad \forall\ n \geq n_0$$

We divide above by n^3, we get

$$(2/n) < c , \qquad \forall\ n \geq n_0 \ \ldots\ldots\ldots\ldots(1)$$

for $c = 1$, there will be $n_0 = 3$, which satisfy (1).

for $c = 0.5$, there will be $n_0 = 7$, which satisfy (1).

Therefore, for every c, there exists n_0 which satisfy (1).

Hence $2n^2 = o(n^3)$.

Asymptotic Notations

Example: Show that $2n^2 \neq o(n^2)$.

Solution: Using definition of o-notation,

$$f(n) < cg(n) \quad , \forall\ n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n^2$. Therefore,

$$2n^2 < cn^2 , \qquad \forall\ n \geq n_0$$

We divide above by n^2, we get

$$2 < c , \qquad \forall\ n \geq n_0 \ \ldots\ldots\ldots\ldots(1)$$

Clearly for $c = 1$, inequality (1) does not satisfy.

Therefore, for every c, there does not exist n_0 which satisfy (1). Hence $2n^2 \neq o(n^2)$.

Asymptotic Notations

Example: Show that $2n^2 = \boldsymbol{\omega}(n)$.

Solution: Using definition of $\boldsymbol{\omega}$ -notation,

$$cg(n) < f(n)\ ,\ \forall\ n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n$. Therefore,

$$cn < 2n^2\ , \qquad \forall\ n \geq n_0$$

We divide above by n, we get

$$c\ < 2n\ , \qquad \forall\ n \geq n_0\ \ldots\ldots\ldots\ldots(1)$$

for $c = 1$, there will be $n_0 = 1$, which satisfy (1).

for $c = 10$, there will be $n_0 = 6$, which satisfy (1).

Therefore, for every c, there exists n_0 which satisfy (1).

Hence $2n^2 = \boldsymbol{\omega}(n)$.

Asymptotic Notations

Example: Show that $2n^2 \neq \boldsymbol{\omega}(n^2)$.

Solution: Using definition of $\boldsymbol{\omega}$ -notation,

$$cg(n) < f(n)\ ,\ \forall\ n \geq n_0$$

Here, $f(n) = 2n^2$, and $g(n) = n^2$. Therefore,

$$cn^2 < 2n^2\ , \qquad \forall\ n \geq n_0$$

We divide above by n^2, we get

$$c\ < 2\ , \qquad \forall\ n \geq n_0\ \ldots\ldots\ldots\ldots(1)$$

Clearly for $c = 3$, there does not exists n_0, which satisfy (1).

Therefore, for every c, there does not exist n_0 which satisfy (1). Hence $2n^2 \neq \boldsymbol{\omega}(n^2)$.

Asymptotic Notations

Example: Show that using definition of notations

(a) $3n^3-10n+50 = \theta(n^3)$

(b) $5n^2-100n \neq \theta(n^3)$

(c) $3n^3-10n+50 = O(n^3)$

(d) $5n^2-100n \neq O(n)$

(e) $3n^3-10n+50 = \boldsymbol{\Omega}(n^3)$

(f) $5n^2-100n \neq \boldsymbol{\Omega}(n^3)$

Asymptotic Notations

Limit based method to compute notations for a function

First compute $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c.$

(1) If c is a constant such that $0 < c < \infty$, then $f(n) = \theta(g(n))$.

(2) If c is a constant such that $0 \leq c < \infty$, then $f(n) = O(g(n))$.

(3) If c is a constant such that $0 < c \leq \infty$, then $f(n) = \boldsymbol{\Omega}(g(n))$.

(4) If c is a constant such that $c = 0$, then $f(n) = o(g(n))$.

(5) If c is a constant such that $c = \infty$, then $f(n) = \boldsymbol{\omega}(g(n))$.

Asymptotic Notations

Exercises

1. Let f(n) and g(n) be asymptotically non-negative functions. Using the basic definition of ,θ-notation, prove that **$\max(f(n), g(n)) = \theta(f(n)+g(n))$.**

2. Show that for any real constants a and b, where $b > 0$,

$$(n+a)^b = \theta(n^b)$$

3. Solve the followings:-

(a) Is $2^{n+1} = O(2^n)$?

(b) Is $2^{2n} = O(2^n)$?

4. Prove that $o(g(n)) \cap \boldsymbol{\omega}(g(n))$ is the empty set.

5. Prove that $n! = \boldsymbol{\omega}(2^n)$ and $n! = o(n^n)$.

6. Which is asymptotically larger: lg(lg*n) or lg*(lg n)?

Asymptotic Notations

Exercise(cont.)

7. Arrange the following in ascending order of growth or rank the following functions by order of growth.

n^3, $(3/2)^n$, 2^n, n^2, $\log(n)$, 2^{2n}, $\log\log(n)$, $n!$, e^n.

8. Let f(n) and g(n) be two asymptotically positive functions. Prove or disprove the following:-

(a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

(b) $f(n) + g(n) = \theta(\min(f(n),g(n)))$.

(c) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

(d) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.

(e) $f(n) = O(((f(n))^2)$

Exercise(cont.)

Solution-(4):

Assume $f(n) \in o(g(n)) \cap \boldsymbol{\omega}(g(n))$.

$\Rightarrow f(n) \in o(g(n))$ and $f(n) \in \boldsymbol{\omega}(g(n))$

$\Rightarrow f(n) < cg(n)$ and $cg(n) < f(n)$, for any c

$\Rightarrow$ Both inequality can not be true for any c.

Therefore, our assumption is incorrect.

Hence, $f(n) \notin o(g(n)) \cap \boldsymbol{\omega}(g(n))$. Therefore,

$o(g(n)) \cap \boldsymbol{\omega}(g(n))$ is the empty set.

Exercise(cont.)

Solution-(5): $n! = \boldsymbol{\omega}(2^n)$ and $n! = o(n^n)$

(i) $c2^n < n!$

for $c = 1$, $n_0 = 4$

for $c = 10$, $n_0 = 6$

(ii) $n! < cn^n$

for $c = 1$, $n_0 = 2$

for $c = 0.1$, $n_0 = 3$

Exercise(cont.)

Solution-(6): lg(lg*n) or lg*(lg n)

lg(lg*n) = lg(lg n)*

lg*(lg n) = (lg lg n)*

Solution-(7):

n^3, $(3/2)^n$, 2^n, n^2, log(n), 2^{2n}, loglog(n), n!, e^n

Ascending order is

loglog(n), log(n), n^2, n^3, $(3/2)^n$, 2^n , e^n, n!, 2^{2n}

Asymptotic Notations

AKTU questions

1. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g(n)). $f_1(n) = n^{2.5}$, $f_2(n) = \sqrt{2^n}$, $f_3(n) = n + 10$, $f_4(n) = 10^n$, $f_5(n) = 100^n$, and $f_6(n) = n^2 \log n$

1. Rank the following by growth rate: n, $2\lg \sqrt{n}$, log n, log (logn), $\log^2 n$, (lgn)lgn, 4, $(3/2)^n$, n!

Recurrence relation

Recurrence equations will be of the following form:-

(1) $T(n) = aT(n/b) + f(n)$
(2) $T(n) = T(n-1) + n$
(3) $T(n) = T(n/3) + T(2n/3) + n$
(4) $T(n) = T(n-1) + T(n-2)$

Some approaches to sole recurrence relations

(1) Iterative method
(2) Substitution method
(3) Recurrence Tree
(4) Master theorem method

Substitution method

The ***substitution method*** for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

Example: Find the upper bound of following recurrence relation $T(n) = 2T(\lfloor n/2 \rfloor) + n$. .....(1)

Solution: We will solve this using substitution method.

Guess the upper bound of this equation is $T(n) = O(n\log n)$.

Now, we have to prove that this guessing solution is correct.

By definition of upper bound,

$$T(n) \leq c\, n\log n, \quad \forall\, n \geq n_0. \quad \ldots\ldots\ldots\ldots\ldots(2)$$

Substitution method

We will prove inequality (2) using induction method.

Assume $T(1) = 1$.

Using equation (1),

$$T(2) = 2T(\lfloor 2/2 \rfloor) + 2 = 2T(1) + 2 = 4$$

$$T(3) = 2T(\lfloor 3/2 \rfloor) + 3 = 2T(1) + 3 = 5$$

For n=1.

$$T(1) \leq c\ 1\log 1 \Rightarrow 1 \leq c\ .\ 0 \Rightarrow 1 \leq 0 \text{ (False)}$$

Therefore, equation (2) is false for $n = 1$.

For n=2.

$$T(2) \leq c\ 2\log 2 \Rightarrow 4 \leq c\ .\ 2 \Rightarrow 4 \leq 2c \text{ (True for } c \geq 2)$$

Therefore, equation (2) is true for $n = 2$.

For n=3.

$$T(3) \leq c\ 3\log 3 \Rightarrow 5 \leq 3c.\log 3 \text{ (True for } c \geq 2)$$

Therefore, equation (2) is true for $n = 3$.

Substitution method

Assume equation (2) is true for n= n/2. We will prove for n.

Since equation (2) is true for n = n/2, therefore

$T(n/2) \leq c\,(n/2)\log(n/2), \quad \forall\, n \geq n_0$ .....(3)

Now for n,

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\leq 2c\,(\lfloor n/2 \rfloor)\log(\lfloor n/2 \rfloor) + n \\
&\leq 2c\,(n/2)\log(n/2) + n \\
&= cn\,(\log n - \log 2) + n \\
&= cn\,(\log n - 1) + n \\
&= cn\log n - cn + n \\
&\leq cn\log n \quad \text{if } c \geq 1
\end{aligned}
$$

Therefore, $T(n) \leq cn\log n$ if $c \geq 1$.

Hence, equation (2) is also proved for n. Therefore, guessing solution is correct.

Therefore the upper bound of given recurrence relation is

$T(n) = O(n\log n)$.

Substitution method

Example: Find the upper bound of following recurrence relation $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$

Solution: When n is large, the difference between $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 17$ is not that large. Therefore, given equation is equivalent to the following equation

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Similarly, since $\lfloor n/2 \rfloor$ and $n/2$ are approximately same, therefore above equation is equivalent to the following equation

$$T(n) = 2T(n/2) + n$$

Since this equation is equivalent to the previous question therefore upper bound will be $T(n) = O(n\log n)$.

Substitution method

Example: Solve the following recurrence relation

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n$$

Solution: Here we change the variable n to m.

Let $n = 2^m$

Put $n = 2^m$ in given equation, we get

$$T(2^m) = 2T(\lfloor\sqrt{2^m}\rfloor) + m$$

$$= 2T(\lfloor 2^{m/2}\rfloor) + m$$

Let $T(2m) = S(m)$, therefore

$$S(m) = 2\,S(\lfloor m/2\rfloor) + m$$

Since this equation is equialent to the first question, therefore its solution will be $S(m) = m\log(m)$.

Hence $T(n) = T(2m) = S(m) = m\log(m) = \log(n)\,\log\log(n)$

i.e. $T(n) = \log(n)\,\log\log(n)$

Substitution method

Example: Solve the following recurrence relation

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

Solution: Since $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are approximately equal to $n/2$, therefore given equation is equivalent to following equation

$$T(n) = T(n/2) + T(n/2) + 1$$

$$T(n) = 2T(n/2) + 1$$

Now, we guess the solution is $T(n) = O(n)$. Therefore, we have to prove $T(n) \leq cn\text{-}d, \forall\ n \geq n_0$. .....(2)

Assume this is true for $n = n/2$, therefore

$$T(n/2) \leq cn/2 - d \quad .....(3)$$

Now for n,

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &\leq 2\,(cn/2 - d) + 1 \text{ (using equation (3))} \\ &= cn - 2d + 1 \\ &\leq cn\text{-}d \text{ if } d \geq 1 \end{aligned}$$

Hence $T(n) \leq cn\text{-}d$ for $d \geq 1$. Therefore **$T(n) = O(n)$.**

Recurrence tree method

- In a ***recursion tree***, each node represents the cost of a single sub-problem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

- A recursion tree is best used to generate a good guess, which you can then verify by the substitution method.

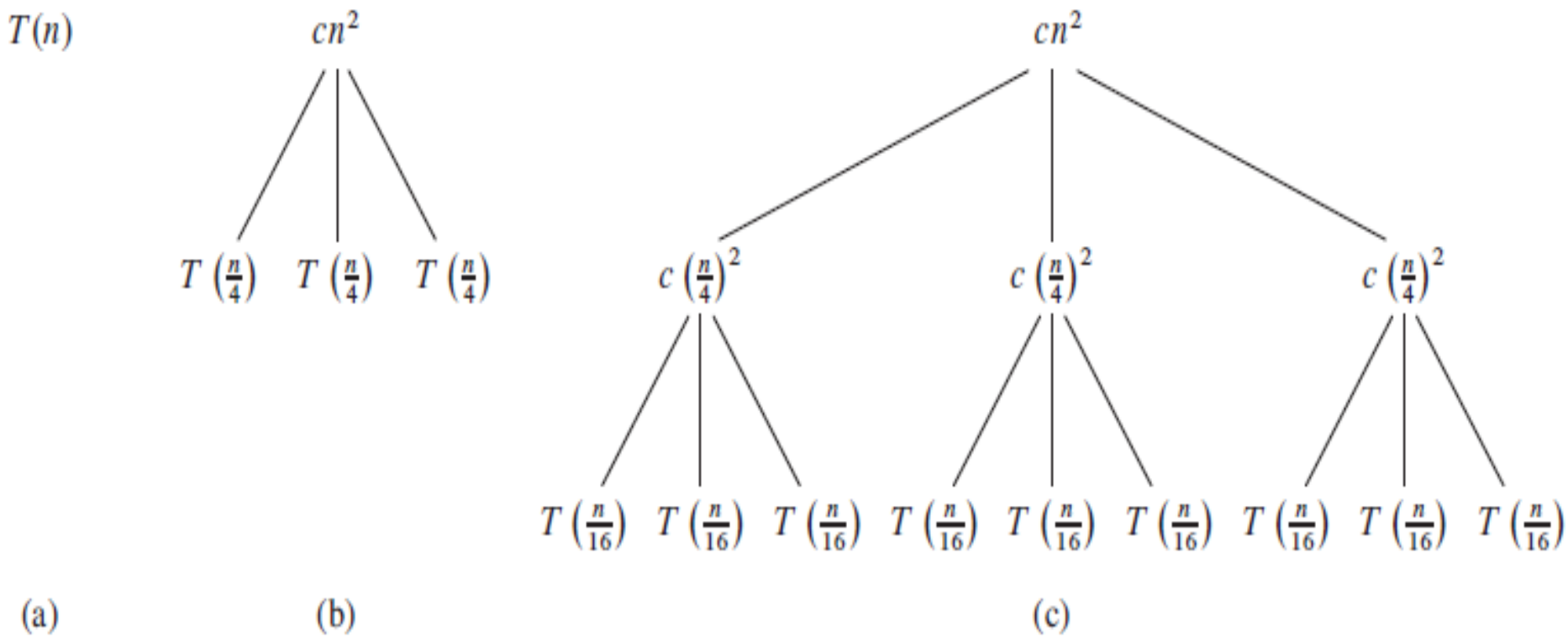
Recurrence tree method

Example: Solve the following recurrence equation

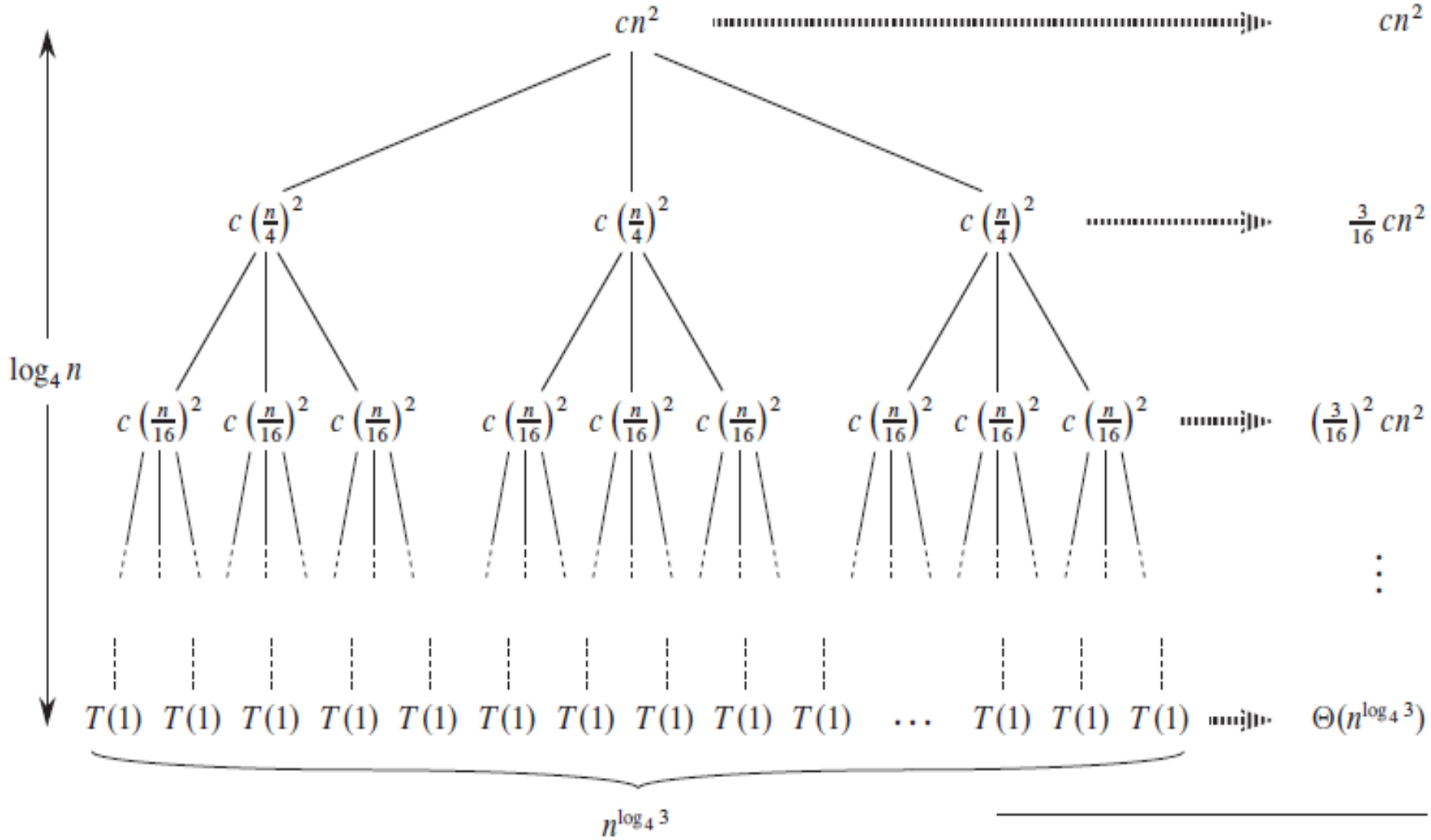
$$T(n) = 3T(\lfloor n/4 \rfloor) + \theta(n^2)$$

Solution: This equation is equivalent to the following equation $T(n) = 3T(n/4) + cn^2$

Here, we create recurrence tree for above equation.



Recurrence tree method



(d)

Total: $O(n^2)$

Recurrence tree method

Let h is the height of the tree. Then

$$n/4^h = 1 \Rightarrow h = \log_4 n$$

Now we add up the costs over all levels to determine the cost for the entire tree:

$$T(n) = cn^2 + (3/16)cn^2 + (3/16)^2 cn^2 + \ldots\ldots\ldots\ldots\ldots + (3/16)^{\log_4 n - 1} cn^2 + \theta(n^{\log_4 3})$$

$$= cn^2 \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i + \theta(n^{\log_4 3})$$

$$< cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \theta(n^{\log_4 3})$$

$$= cn^2 (1/(1-(3/16))) + \theta(n^{\log_4 3})$$

$$= (16/13)\, cn^2 + \theta(n^{\log_4 3})$$

$$= O(n^2)$$

Therefore, solution will be **$T(n) = O(n^2)$.**

Recurrence tree method

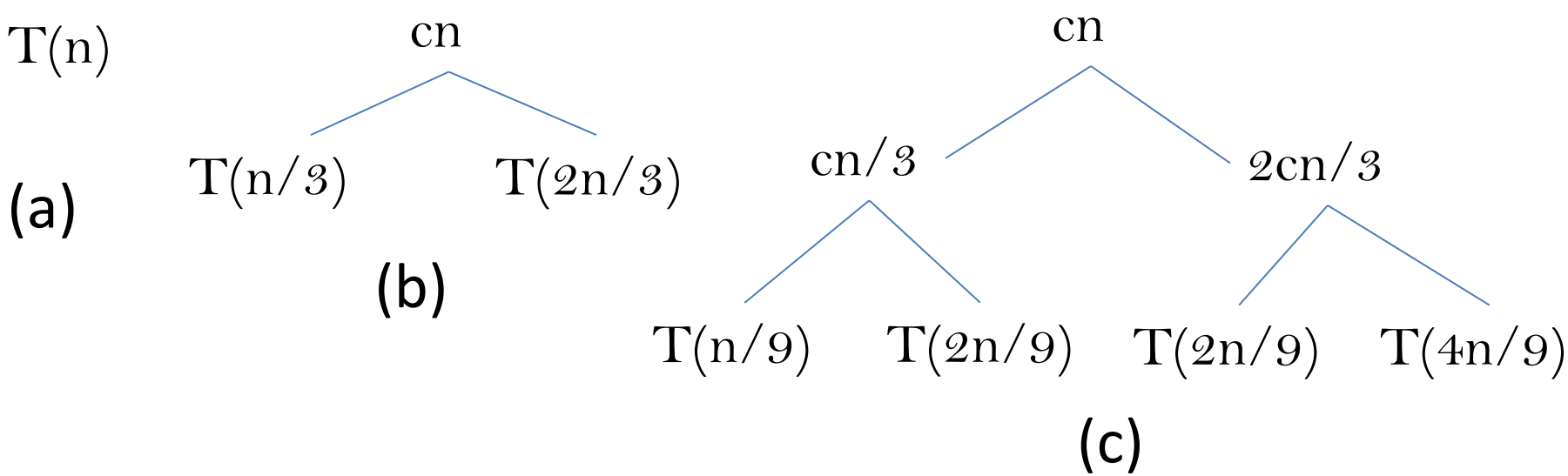
Example: Solve the following recurrence relation using recurrence tree method

$$T(n) = T(n/3) + T(2n/3) + \theta(n)$$

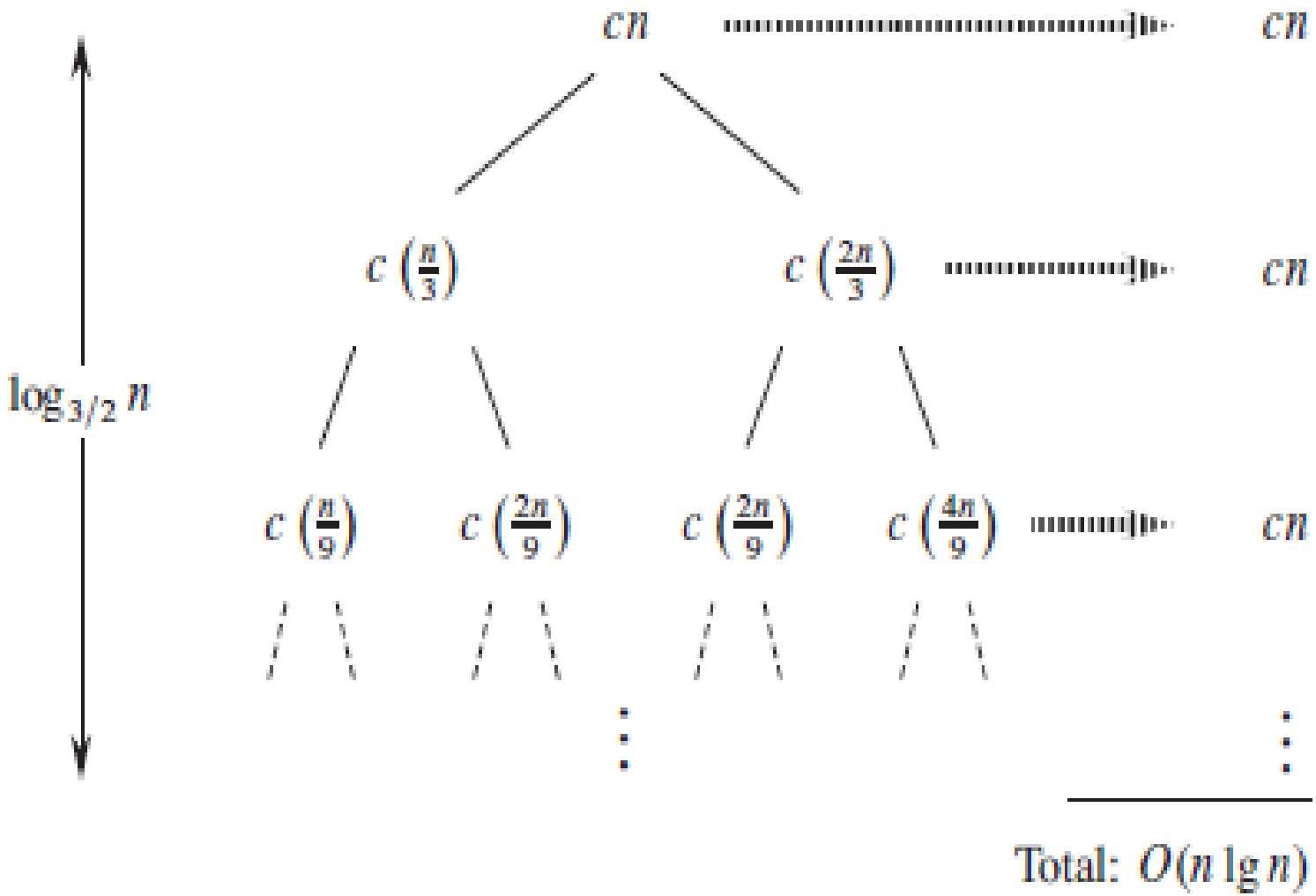
Solution: This recurrence relation is equivalent to the following relation

$$T(n) = T(n/3) + T(2n/3) + cn$$

Recurrence tree for this equation will be the following:-



Recurrence tree method



(d)

Recurrence tree method

Let the height of the tree is h. Therefore

$$\frac{n}{\left(\frac{3}{2}\right)^h} = 1 \Rightarrow n = \lg_{3/2} n$$

Therefore, the total cost of the tree is

$$T(n) < cn + cn + \ldots\ldots\ldots\ldots\ldots\ldots + cn$$

$$= (h+1)\, cn$$

$$= (\lg_{3/2} n + 1) cn$$

$$= cn \lg_{3/2} n + cn$$

$$= O(n \lg_{3/2} n)$$

$$= O(n \lg n)$$

Therefore **$T(n) = O(n \lg n)$**

Recurrence tree method

Exercise

(1) Draw the recursion tree for $T(n) = 4\,T(\lfloor n/2 \rfloor) + cn$, where c is a constant and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.

(2) Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

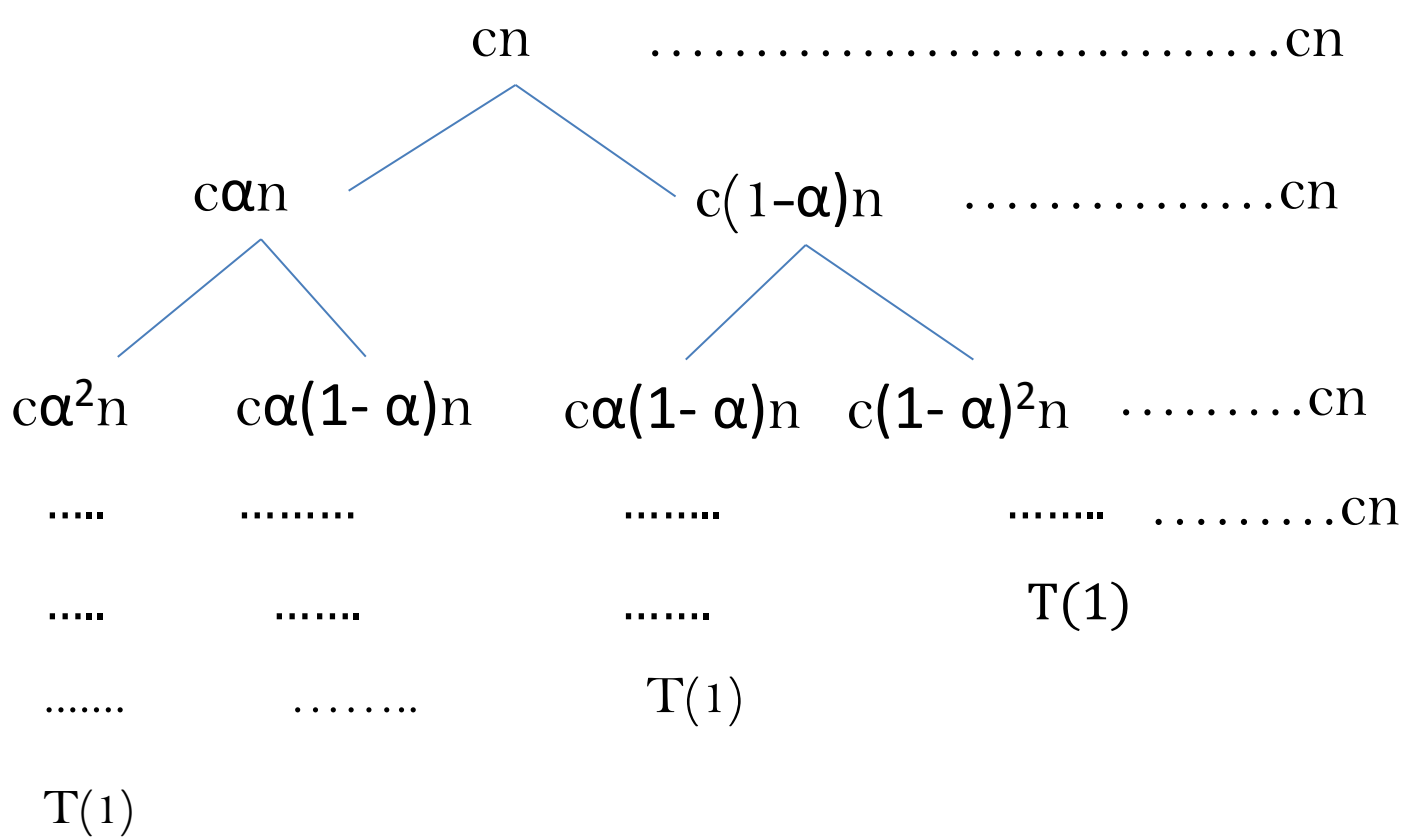
(3) Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

Recurrence tree method

Exercise(Solution)

(3.) Assume $\frac{1}{2} \leq \alpha < 1$. Then $0 < (1-\alpha) \leq \frac{1}{2}$.

Recurrence tree for this recurrence relation will be the following:



Let h is the height of the tree. Therefore,

$$\alpha^h n = 1 \Rightarrow h = \log_{1/\alpha}(n)$$

Therefore, total cost of the tree

$$T(n) = cn + cn + cn + \ldots\ldots\ldots\ldots\ldots\ldots + cn$$

$$= (h+1)\, cn$$

$$= (\log_{1/\alpha}(n) + 1)\, cn$$

$$= cn \log_{1/\alpha}(n) + cn$$

$$= O(n \log_{1/\alpha}(n))$$

$$= O(n\log n)$$

Therefore, $T(n) = O(n\log n)$.

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n);$$

Where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then
   $T(n) = \theta(n^{\log_b a})$.
2. If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \theta(f(n))$.

Master Theorem Method

Example: Solve the following recurrence relations using master theorem method

(a) $T(n) = 9T(n/3) + n$

(b) $T(n) = T(2n/3) + 1$

(c) $T(n) = 3T(n/4) + n \log n$

Solution:

(a) Consider $T(n) = 9T(n/3) + n$

In this recurrence relation, $a = 9$, $b = 3$ and $f(n) = n$.

Therefore, $n^{\log_b a} = n^{\log_3 9} = n^2$

Clearly, $n^{\log_b a} > f(n)$, therefore case1 can be applied.

Now determine ϵ such that $f(n) = O(n^{2-\epsilon})$. Here $\epsilon = 1$.

Therefore case 1 will be applied.

Hence solution will be $T(n) = \theta(n^2)$.

Master Theorem Method

Solution:

(b) Consider $T(n) = T(2n/3) + 1$

In this recurrence relation, $a = 1$, $b = 3/2$ and $f(n) = 1$.

Therefore, $n^{\log_b a} = n^{\log_{3/2} 1} = 0$

Clearly, $f(n) = \theta(n^{\log_b a})$, therefore case 2 will be applied.

Hence solution will be $T(n) = \theta(\log n)$.

Master Theorem Method

Solution:

(c) Consider $T(n) = 3T(n/4) + n \log n$

In this recurrence relation, $a = 3$, $b = 4$ and $f(n) = n \log n$.

Therefore, $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

Clearly, $n^{\log_b a} < f(n)$, therefore case 3 can be applied.

Now determine ϵ such that $f(n) = \Omega(n^{0.793+\epsilon})$. Here $\epsilon = 0.207$.

Now, $af(n/b) \leq cf(n)$ imply that $3f(n/4) \leq cf(n)$

$\Rightarrow 3(n/4)\log(n/4) \leq c\, n \log n$

$\Rightarrow (3/4)\log(n/4) \leq c \log n$

Clearly above inequality is satisfied for $c = 3/4$. Therefore case 3 will be applied.

Hence solution will be $T(n) = \theta(n \log n)$.

Master theorem method

Example: Solve the following recurrence relation

$$T(n) = 2T(n/2) + n \log n$$

Solution: Here, $a = 2$, $b=2$ and $f(n) = n \log n$.

$$n^{\log_b a} = n^{\log_2 2} = n$$

If we compare $n^{\log_b a}$ and $f(n)$, we get $f(n)$ is greater than $n^{\log_b a}$. Therefore, case 3 may be applied.

Now we have to determine $\epsilon > 0$ which satisfy $f(n) = \Omega(n^{\log_b a+\epsilon})$, i.e. $n \log n = \Omega(n^{1+\epsilon})$. Clearly there does not exist any ϵ which satisfy this condition. Therefore case 3 can not be applied. Other two cases are also not satisfied. Therefore Master theorem can not be applied in this recurrence relation.

Generalized Master theorem

Theorem: If $f(n) = \theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$, then the solution of recurrence will be $T(n) = \theta(n^{\log_b a} \lg^{k+1} n)$.

Now, consider the previous example:-

$$T(n) = 2T(n/2) + n \log n$$

Solve it using aboe theorem,

Here a =2, b= 2, and k = 1. Therefore, the solution of this recurrence will be

$$T(n) = \theta(n^{\log_2 2} \lg^{1+1} n)$$

$$= \theta(n \lg^2 n)$$

Hence, $T(n) = \theta(n \lg^2 n)$

Recurrence relation

Exercise

1. Use the master method to give tight asymptotic bounds for the following recurrences:-

(a) $T(n) = 8T(n/2) + \theta(n^2)$

(b) $T(n) = 7T(n/2) + \theta(n^2)$

(c) $T(n) = 2T(n/4) + 1$

(d) $T(n) = 2T(n/4) + \sqrt{n}$

2. Can the master method be applied to the recurrence $4T(n/2) + n^2 \log n$? Why or why not? Give an asymptotic upper bound for this recurrence.

Recurrence relation

3. Give asymptotic upper and lower bounds for T(n) in each of the following recurrences. Assume that T(n) is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

(a) $T(n) = 2T(n/2) + n^4$

(b) $T(n) = T(7n/10) + n$

(c) $T(n) = 16T(n/4) + n^2$

(d) $T(n) = 2T(n/4) + \sqrt{n}$

(e) $T(n) = T(n-2) + n^2$

(f) $T(n) = 7T(n/3) + n^2$

(g) $T(n) = 3T(n/3 - 2) + n/2$

Recurrence relation

4. Give asymptotic upper and lower bounds for T(n) in each of the following recurrences. Assume that T(n) is constant for sufficiently small n. Make your bounds as tight as possible, and justify your answers.

(a) $T(n) = 4T(n/3) + n \lg n$

(b) $T(n) = 3T(n/3) + n/\lg n$

(c) $T(n) = 2T(n/2) + n/\lg n$

(d) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

(e) $T(n) = T(n-1) + 1/n$

(f) $T(n) = T(n-1) + \lg n$

(g) $T(n) = T(n-2) + 1/\lg n$

(h) $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

Recurrence relation

Some exercise solution

(3-e) Recurrence relation is $T(n) = T(n-2) + n^2$.

We will solve it using iteration method.

$$T(n) = T(n-2) + n^2$$

$$= T(n-4) + (n-2)^2 + n^2$$

$$= T(n-6) + (n-4)^2 + (n-2)^2 + n^2$$

.....

.....

$$= T(0) + 2^2 + 4^2 + 6^2 + \ldots\ldots\ldots\ldots\ldots + (n-2)^2 + n^2$$

$$= d + 2^2 + 4^2 + 6^2 + \ldots\ldots\ldots + (n-2)^2 + n^2 \quad (\text{Let } T(0) = d)$$

$$= d + \frac{n(n+1)(n+2)}{6} = \theta(n^3)$$

Recurrence relation

Some exercise solution

(3-g) Recurrence relation is $T(n) = 3T(n/3 - 2) + n/2$.

This recurrence relation is equivalent to the following equation $T(n) = 3T(n/3) + n/2$

Apply master theorem, here $a = 3$, $b = 3$ and $f(n) = n/2$.

$$n^{\log_b a} = n^{\log_3 3} = n$$

Clearly, $f(n) = n/2 = \theta(n) = \theta(n^{\log_b a})$

Therefore case 2 will be applied. Hence the solution is

$$T(n) = \theta(n \lg n)$$

Recurrence relation

Some exercise solution

(4-a) Recurrence relation is $T(n) = 4T(n/3) + n \lg n$.

Apply master theorem, here $a = 4$, $b = 3$ and $f(n) = n \lg n$.

$$n^{\log_b a} = n^{\log_3 4} = n^{1.26}$$

Clearly, $f(n) = n \lg n = O(n^{1.26-\epsilon}) = O(n^{\log_b a - \epsilon})$

Therefore case 1 will be applied. Hence the solution is

$$T(n) = \theta(n^{1.26})$$

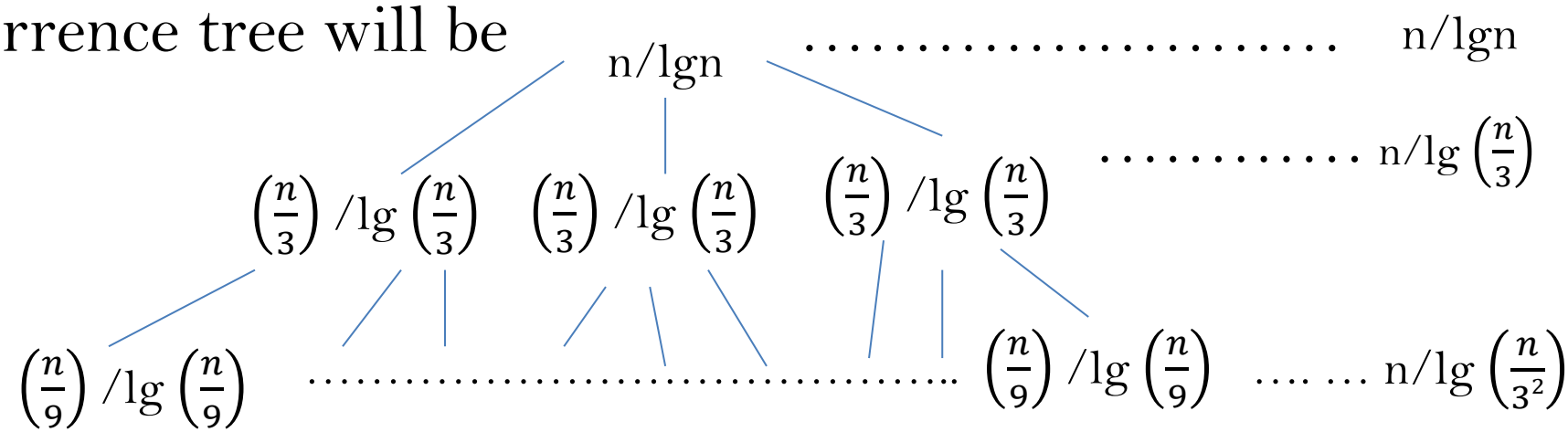
Recurrence relation

Some exercise solution

(4-b) Recurrence relation is $T(n) = 3T(n/3) + n/\lg n$.

We will use recurrence tree method to solve it.

Recurrence tree will be



$T(1)=d$ $T(1) = d$ ..... $T(1) = d$ ..... $3^h d$

Here d -> constant , h-> height of tree

Recurrence relation

Now, we calculate height of tree.

Clearly, $\frac{n}{3^h} = 1$, $\Rightarrow$ h = $log_3 n$.

Now, total cost of this tree is

$$T(n) = \frac{n}{lgn} + \frac{n}{\lg(\frac{n}{3})} + \frac{n}{\lg(\frac{n}{3^2})} + \ldots\ldots\ldots\ldots + \frac{n}{\lg(\frac{n}{3^{h-1}})} + 3^h d$$

$$= n\left(\frac{1}{lgn} + \frac{1}{\lg(\frac{n}{3})} + \frac{1}{\lg(\frac{n}{3^2})} + \ldots\ldots\ldots\ldots + \frac{1}{\lg(\frac{n}{3^{h-1}})}\right) + 3^h d$$

Substituting $n = 3^h$, we get

$$= n\left(\frac{1}{lg3^h} + \frac{1}{\lg(3^{h-1})} + \frac{1}{\lg(3^{h-2})} + \ldots\ldots\ldots\ldots + \frac{1}{\lg(3)}\right) + dn$$

$$= n\left(\frac{1}{h\ lg3} + \frac{1}{(h-1)\lg 3} + \frac{1}{(h-2)\lg 3} + \ldots\ldots + \frac{1}{\lg 3}\right) + dn$$

$$= (n/\lg 3)\left(\frac{1}{h} + \frac{1}{(h-1)} + \frac{1}{(h-2)} + \ldots\ldots + \frac{1}{2} + \frac{1}{1}\right) + dn$$

Recurrence relation

$$T(n) = \frac{n}{lg3}\left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots\ldots\ldots\ldots+\frac{1}{h}\right)+dn$$

Suppose $h = 2^m$

$$=\frac{n}{lg3}\left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots\ldots\ldots\ldots+\frac{1}{2^m}\right)+dn$$

$$< \frac{n}{lg3}\left(1+\left(\frac{1}{2}+\frac{1}{2}\right)+\left(\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}\right)+\cdots\ldots\ldots\ldots+\left(\frac{1}{2^m}+\frac{1}{2^m}+\ldots\ldots\ldots+\frac{1}{2^m}\right)\right)+dn$$

$$= \frac{n}{lg3}\left(1+1+1+\ldots\ldots\ldots\ldots+1\right)+dn$$

$$= \frac{n}{lg3}(m+1)+dn \quad = \frac{n}{lg3}(\lg h+1)+dn$$

$$= \frac{n}{lg3}(\lg \log_3 n+1)+dn$$

$$= O(n \lg \lg n)$$

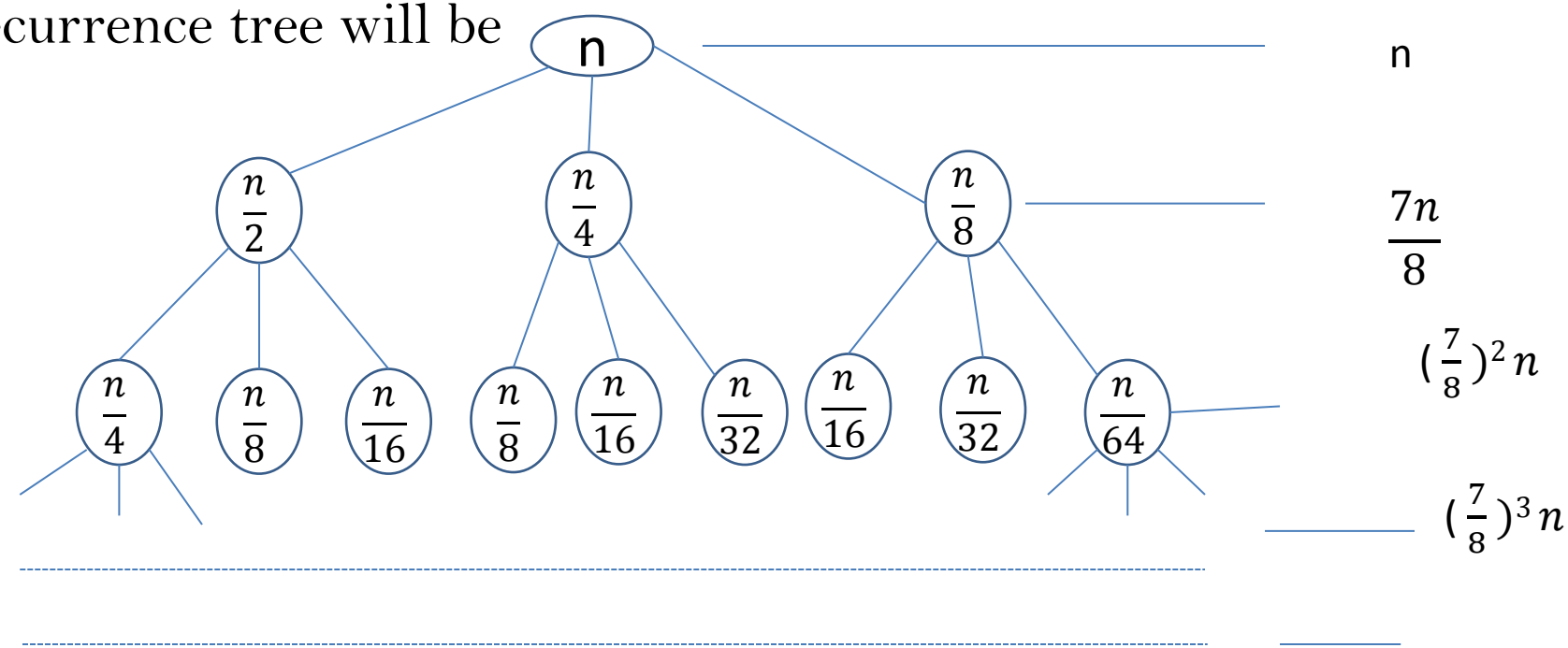
Some exercise solution

(4-d) Recurrence relation is

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

We will use recurrence tree method to solve it.

Recurrence tree will be



Let h is the height of the tree. Therefore, $\frac{n}{2^h} = 1 \Rightarrow h = \lg n$.

Some exercise solution

Therefore total cost

$$T(n) = n + \frac{7n}{8} + \left(\frac{7}{8}\right)^2 n + \left(\frac{7}{8}\right)^3 n + \ldots\ldots\ldots\ldots + \left(\frac{7}{8}\right)^h n$$

$$= n\left(1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \left(\frac{7}{8}\right)^3 + \ldots\ldots\ldots\ldots + \left(\frac{7}{8}\right)^h\right)$$

$$< n\left(1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \left(\frac{7}{8}\right)^3 + \ldots\ldots\ldots\ldots\ldots\ldots\right)$$

Here, we consider up to infinite term because common ratio of this series is $\frac{7}{8}$ that is less than 1.

$$= n\left(\frac{1}{(1-\frac{7}{8})}\right)$$

$$= 8n = O(n)$$

Therefore, $T(n) = O(n)$.

AKTU Examination Questions

1. Solve the recurrence $T(n) = 2T(n/2) + n^2 + 2n + 1$

2. Solve the recurrence using recursion tree method:

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$

3. Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

4. The recurrence $T(n) = 7T(n/3) + n^2$ describes the running time of an algorithm A. Another competing algorithm B has a running time of $S(n) = a\,S(n/9) + n^2$. What is the smallest value of 'a' such that A is asymptotically faster than B?

5. Solve the recurrence relation by substitution method

$$T(n) = 2T(n/2) + n$$

AKTU Examination Questions

6. Show that the solution to $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$.

7. Solve the recurrence: $T(n) = 50\,T(n/49) + \log n!$

8. Solve the following recurrence using Master method:

$T(n) = 4T(n/3) + n^2$

9. Find the time complexity of the recurrence relation

$T(n) = n + T(n/10) + T(7n/5)$

10. Solve the following By Recursion Tree Method

$T(n) = n + T(n/5) + T(4n/5)$

11. The recurrence $T(n) = 7T(n/2) + n^2$ describe the running time of an algorithm A. A competing algorithm A has a running time of $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for a A' is asymptotically faster than A?

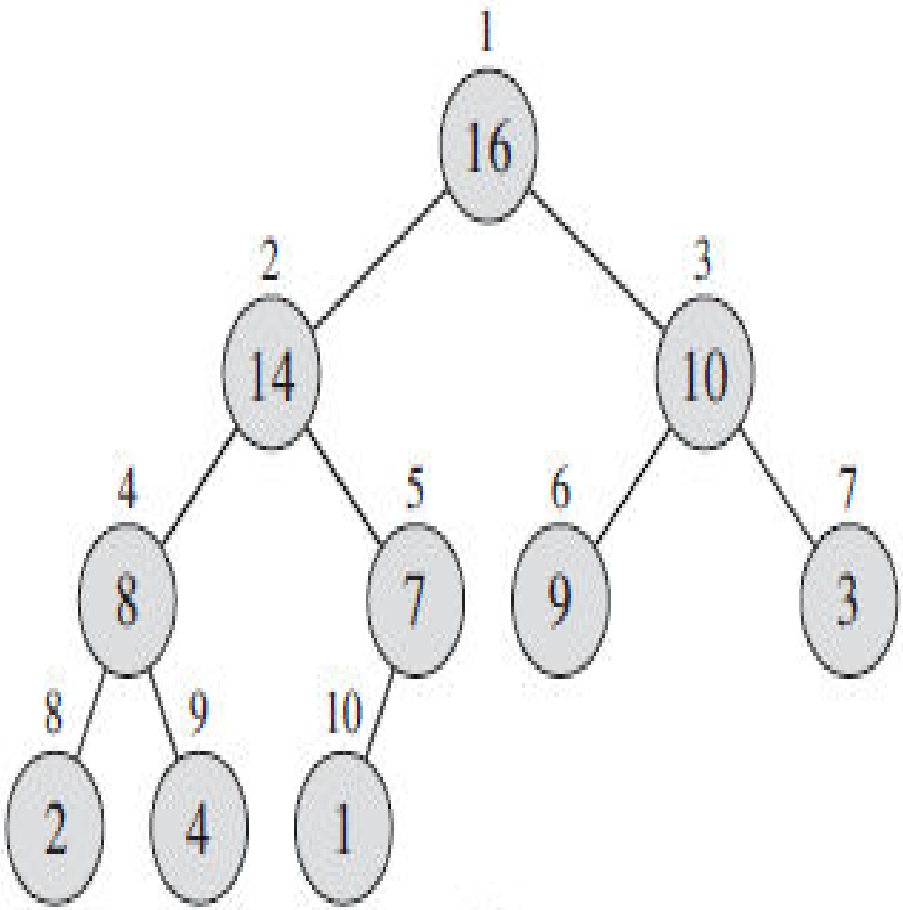
Heapsort

Heap

- The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree.

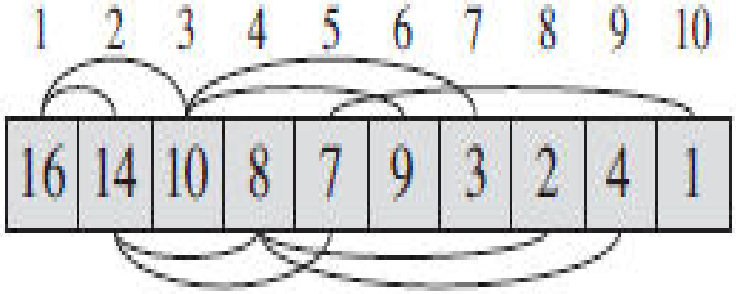
- Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

Heapsort



(a)

Max-heap



(b)

Array

Heapsort

Index: If i the index of a node, then the index of parent and its child are the following:-

Parent(i) = $\lfloor i/2 \rfloor$

Left(i) = 2i

Right(i) = 2i+1

Note: Root node has always index 1 i.e. A[1] is root element.

Heap-size: Heap-size is equal to the number of elements in the heap.

Height of a node: The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf.

Height of heap: The height of the heap is equal to the height of its root.

Heapsort

Types of heap

There are two kinds of binary heaps:

(1) max-heaps (2) min-heaps

Max-heap: The heap is said to be max-heap if it satisfy the **max-heap property**.

The **max-heap property** is that the value at the parent node is always greater than or equal to value at its children.

Min-heap: The heap is said to be min-heap if it satisfy the **min-heap property**.

The **min-heap property** is that the value at the parent node is always less than or equal to value at its children.

Heapsort

Heap sort algorithm consists of the following two sub-algorithms.

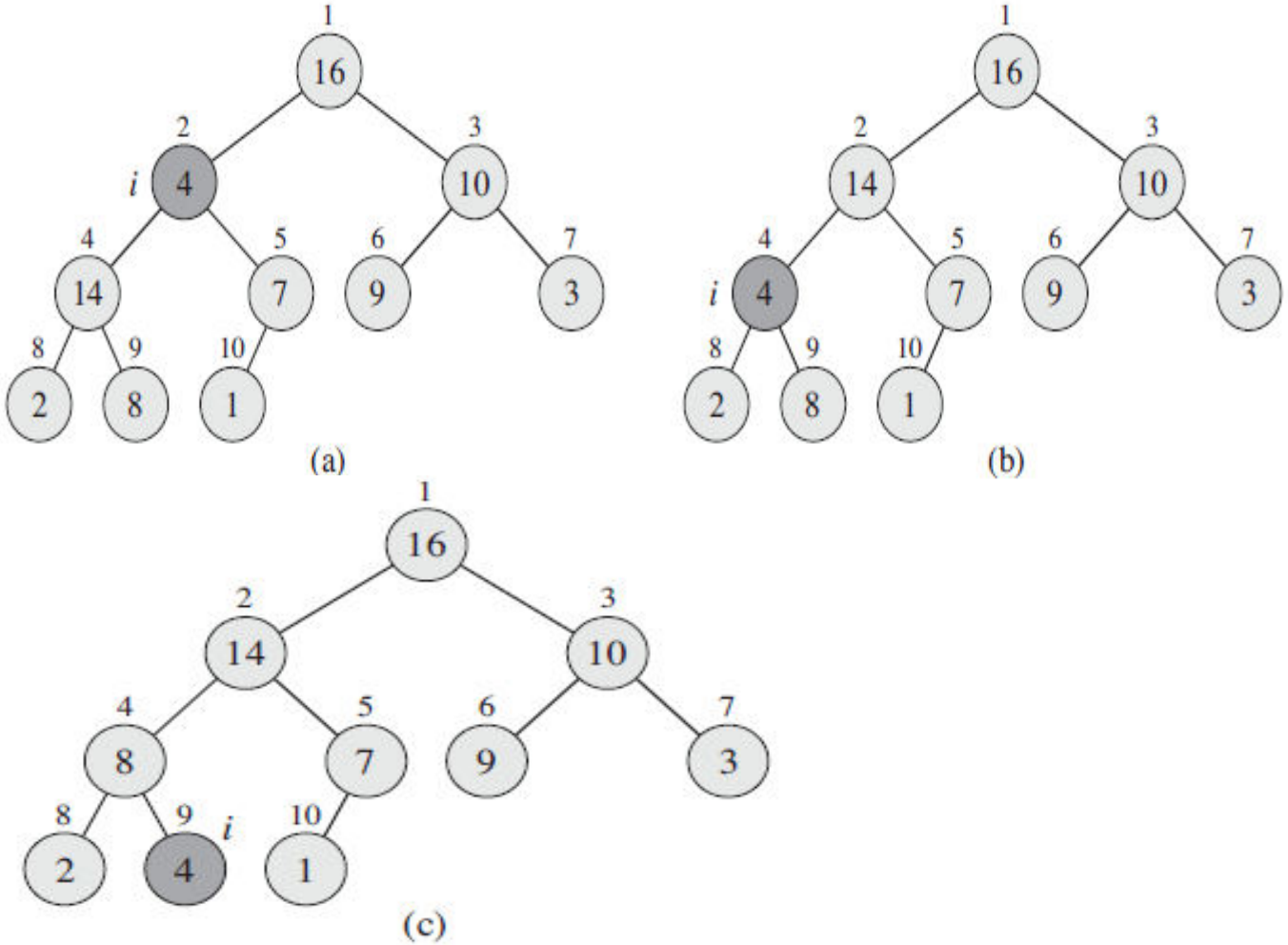
(1) Max-Heapify: It is used to maintain the max-heap property.

(2) Build-Max-Heap: It is used to construct a max-heap for the given set of elements.

Heapsort

Max-Heapify Algorithm

Action done by max-heapify algorithm is shown in the following figures:-



Heapsort

Max-Heapify Algorithm

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

Heapsort

Time complexity of Max-Heapify Algorithm

The running time of max-heapify is determined by the following recurrence relation:-

$$T(n) \leq T(2n/3) + \theta(1)$$

Here n is the size of the sub-tree rooted at node i.

Using master theorem, the solution of this recurrence relation is

$$\mathbf{T(n) = \theta(\lg n)}$$

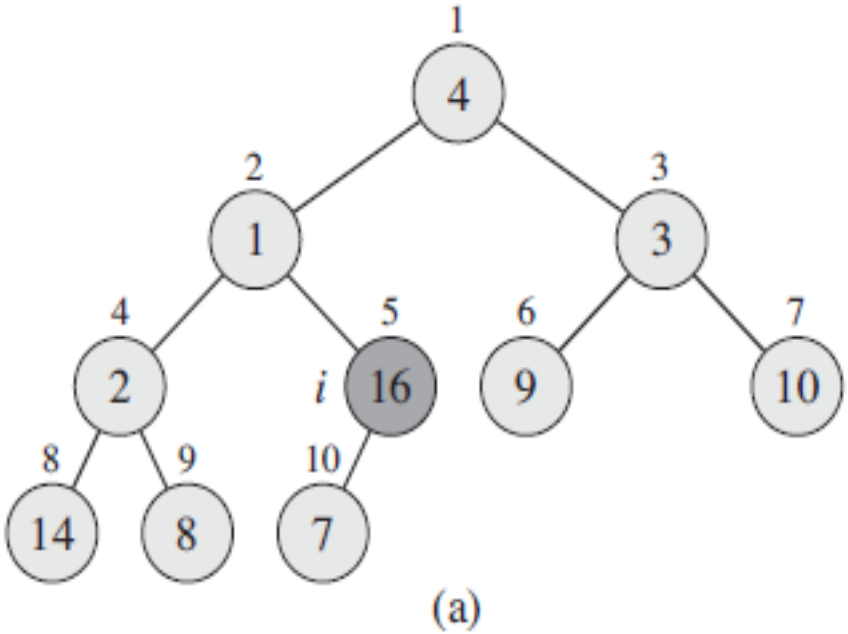
Heapsort

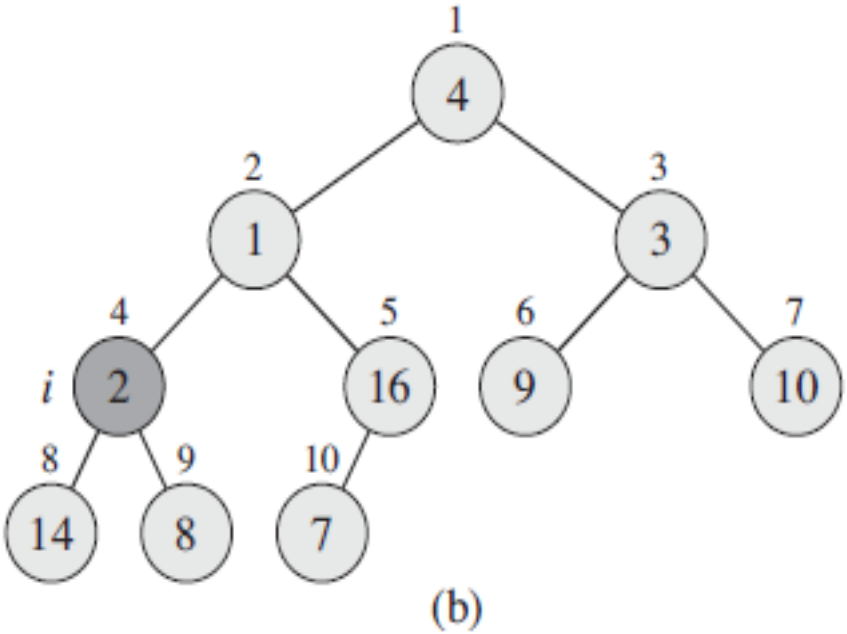
Build-Max-Heap Algorithm

Example: Construct max-heap corresponding to the following elements

4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

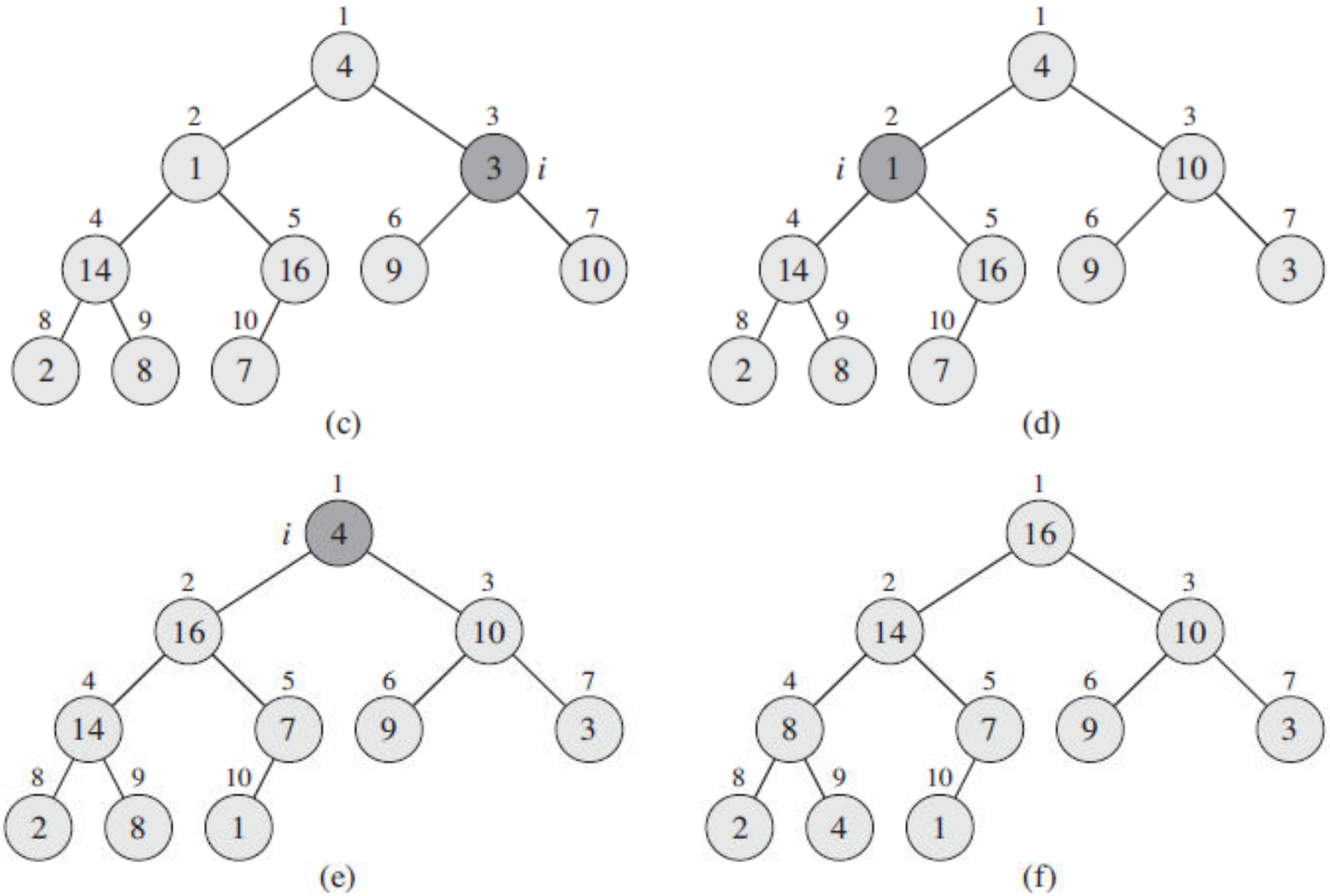
Solution:





Heapsort

Build-Max-Heap Algorithm (cont.)



Heapsort

Build-Max-Heap Algorithm (cont.)

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

The running time of this algorithm is

$$T(n) = O(n \lg n)$$

But this upper bound is not asymptotically tight.

Now, we shall calculate tight upper bound.

Heapsort

Time complexity of Build-Max-Heap Algorithm

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

If h is the height of the sub-tree then running time of max-heapify is O(h).

Therefore, total cost of build-max-heap is

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right).$$

Heapsort

Time complexity of Build-Max-Heap Algorithm

Now, $T(n) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

Since $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$

$$= 2 .$$

Therefore,

$$T(n) = O(2n)$$

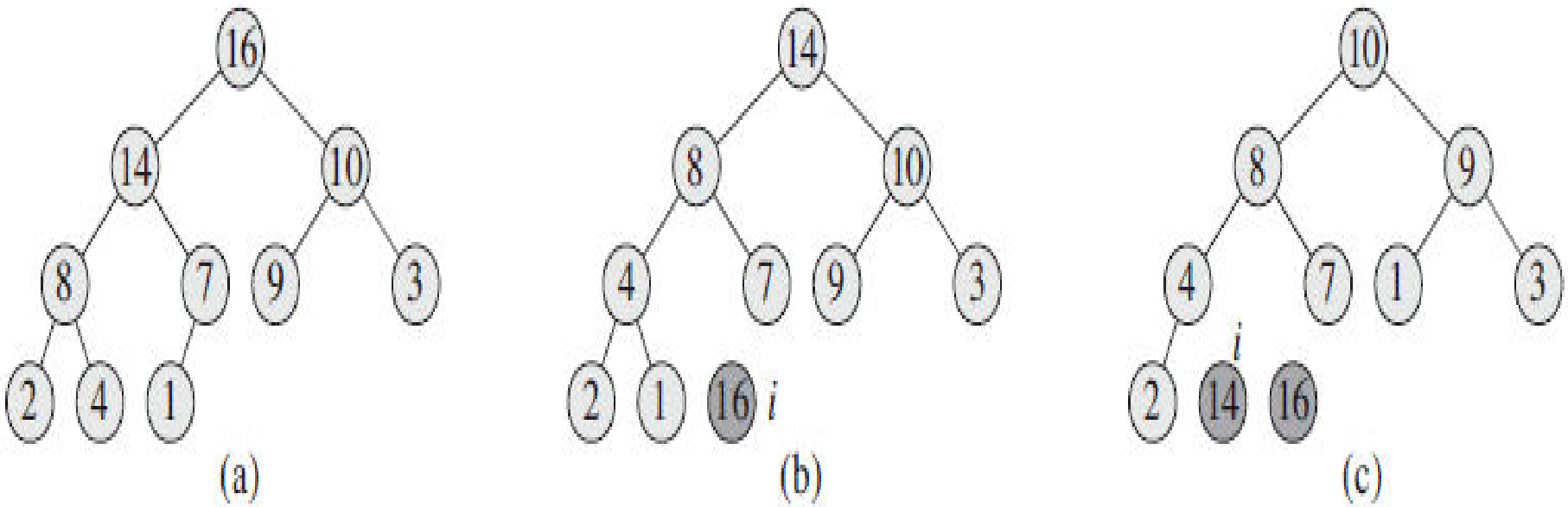
$$= \mathbf{O(n)}$$

Heapsort Algorithm

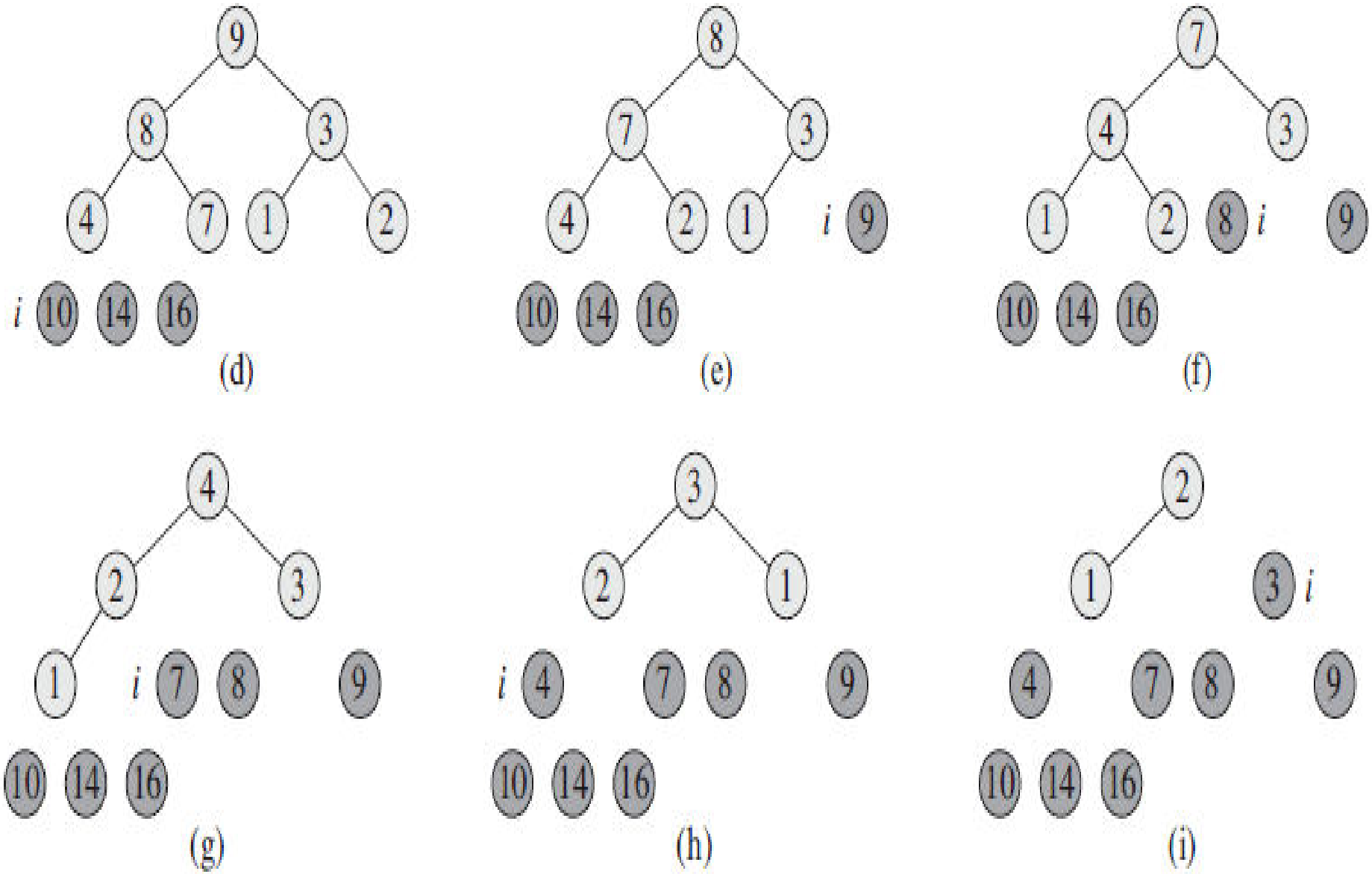
Example: Sort the following elements using heapsort

5, 13, 2, 25, 7, 17, 20, 8, 4.

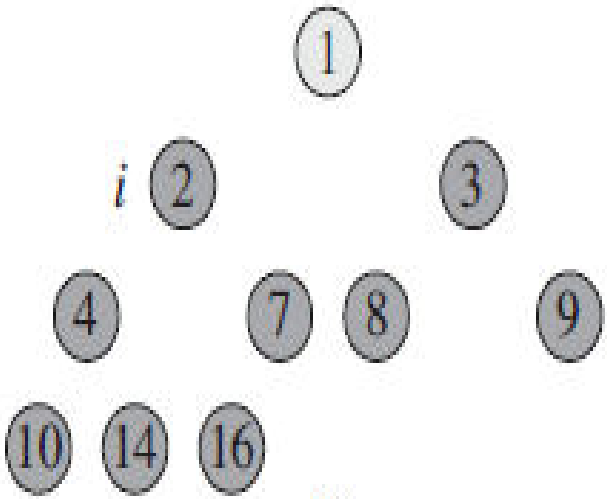
Solution: The operation of HEAPSORT is shown as following:-



Heapsort Algorithm



Heapsort Algorithm



(j)

A	1	2	3	4	7	8	9	10	14	16

(k)

Heapsort Algorithm

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

Time complexity: The HEAPSORT procedure takes time O(nlgn), since the call to BUILD-MAX-HEAP takes time O(n) and each of the n-1 calls to MAX-HEAPIFY takes time O(lgn).

Heapsort Algorithm

Exercise

1. What are the minimum and maximum numbers of elements in a heap of height h?
2. Show that an n-element heap has height $\lfloor \lg n \rfloor$.
3. Is the array with values 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 a max-heap?
4. Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.
5. What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

Heapsort Algorithm

AKTU Examination Questions

1. Sort the following array using Heap-Sort techniques—

   5,8,3,9,2,10,1,35,22

2. How will you sort following array A of elements using heap sort: A = (23, 9, 18, 45, 5, 9, 1, 17, 6).

Quicksort

Quicksort, like merge sort, applies the divide-and-conquer paradigm. The three-step divide-and-conquer process for sorting a typical subarray A[p .. r] is the following:-

Divide: Partition (rearrange) the array A[p .. r] into two (possibly empty) subarrays A[p .. q-1] and A[q+1 .. R] such that each element of A[p .. q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1 .. r]. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p .. q-1] and A[q+1 .. r] by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p .. r] is now sorted.

Quicksort

Algorithm

It divides the large array into smaller sub-arrays. And then quicksort recursively sort the sub-arrays.

Pivot

1. Picks an element called the "pivot".

Partition

2. Rearrange the array elements in such a way that the all values lesser than the pivot should come before the pivot and all the values greater than the pivot should come after it.

This method is called partitioning the array. At the end of the partition function, the pivot element will be placed at its sorted position.

Recursive

3. Do the above process recursively to all the sub-arrays and sort the elements.

Quicksort

Algorithm

Base Case

If the array has zero or one element, there is no need to call the partition method.

So we need to stop the recursive call when the array size is less than or equal to 1.

Pivot

There are many ways we can choose the pivot element.

i) The first element in the array

ii) The last element in the array

iii) The middle element in the array

iv) We can also pick the element randomly.

Note: In our algorithm, we are going to pick the last element as the pivot element.

Quicksort

Example: Sort the following elements using quicksort

2, 8, 7, 1, 3, 5, 6, 4

Solution: Here, we pick the last element in the list as a pivot

1	2	3	4	5	6	7	8
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4
2	1	7	8	3	5	6	4
2	1	3	8	4	5	6	4

Quicksort

1	2	3	4	5	6	7	8
2	1	3	8	7	5	6	4
2	1	3	4	7	5	6	8

Partition completed in first pass

2	1	3	4	7	5	6	8
2	1	3	4	7	5	6	8
2	1	3	4	7	5	6	8
2	1	3	4	7	5	6	8
2	1	3	4	7	5	6	8

Partition completed in second pass

Quicksort

1	2	3	4	5	6	7	8
2	1	3	4	7	5	6	8
2	1	3	4	7	5	6	8
1	2	3	4	5	7	6	8
1	2	3	4	5	6	7	8

Partition completed in third pass

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Process completed

Quicksort Algorithm

```
Quicksort(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, *length[A])*.

Quicksort Algorithm

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quicksort Algorithm Analysis

The running time of quicksort algorithm is

$$\mathbf{T(n) = T(n_1) + T(n_2) + \theta(n)} \quad \ldots\ldots\ldots\ldots\ldots\ldots(1)$$

Here n_1 and n_2 are the size of sub-problems.

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.

Quicksort Algorithm Analysis

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one sub-problem with n-1 elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. Therefore,

$$T(n) = T(0) + T(n-1) + \theta(n)$$

$$T(n) = T(n-1) + \theta(n) \quad (\text{since } T(0) = \theta(1))$$

After solving this recurrence relation, we get

$$\mathbf{T(n) = \theta(n^2)}$$

Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\theta(n^2)$ running time occurs when the input array is already completely sorted—a common situation in which insertion sort runs in O(n) time.

Quicksort Algorithm Analysis

Best-case partitioning

If PARTITION produces two sub-problems, each of size no more than n/2, since one is of size ⌊n/2⌋ and one of size ⌈n/2⌉-1 . In this case, quicksort runs much faster. The recurrence for the running time is then

$$T(n) = 2T(n/2) + \theta(n)$$

After solving this recurrence relation, we get

$$\mathbf{T(n) = \theta(n \lg n)}$$

Note: By equally balancing the two sides of the partition at every level of the recursion, we get an asymptotically faster algorithm.

Quicksort Algorithm Analysis

<u>Balanced partitioning</u>

The average-case running time of quicksort is much closer to the best case than to the worst case.

Suppose that the partitioning algorithm always produces a 9-to-1 proportional split. In this case, running time will be

$$T(n) = T(9n/10) + T(n/10) + \theta(n)$$

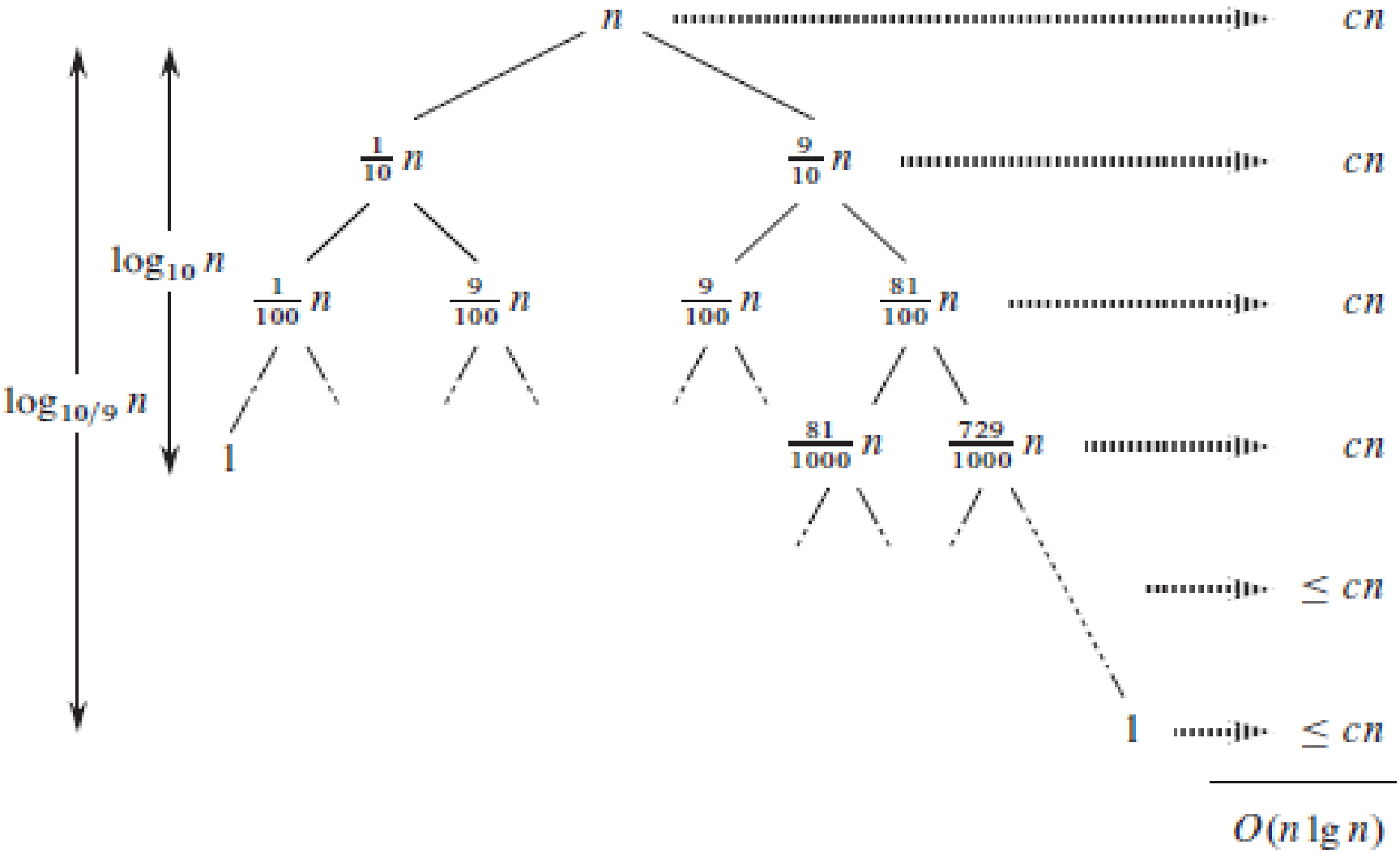
This relation is equivalent to the following

$$T(n) = T(9n/10) + T(n/10) + cn$$

We can solve this recurrence using recurrence tree method.

Quicksort Algorithm Analysis

Recurrence tree will be



Quicksort Algorithm Analysis

Therefore, the solution of recurrence relation will be

$$T(n) \leq cn + cn + cn + \ldots\ldots\ldots\ldots\ldots + cn$$
$$= cn(1 + \lg_{10/9} n)$$
$$= cn + cn\lg_{10/9} n$$

Therefore, **$T(n) = O(n\lg n)$**

Quicksort Algorithm Analysis

Exercise

1. Sort the following elements using quicksort

   13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11.

2. What is the running time of QUICKSORT when all elements of array A have the same value?

3. Show that the running time of QUICKSORT is, $\theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

4. Suppose that the splits at every level of quicksort are in the proportion $1-\alpha$ to α, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $\lg n / \lg \alpha$ and the maximum depth is approximately $\lg n / \lg(1-\alpha)$. (Don't worry about integer round-off.)

A randomized version of quicksort

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

A randomized version of quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

The running time of randomized quick sort will be

$T(n) = O(n\lg n)$

Sorting in Linear Time

Comparison Sort

In a comparison based sort, we use only comparisons between elements to gain order information about an input sequence $<a_1, a_2, \ldots\ldots\ldots\ldots\ldots, a_n>$. That is, given two elements a_i and a_j , we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order.

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees.

A ***decision tree*** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

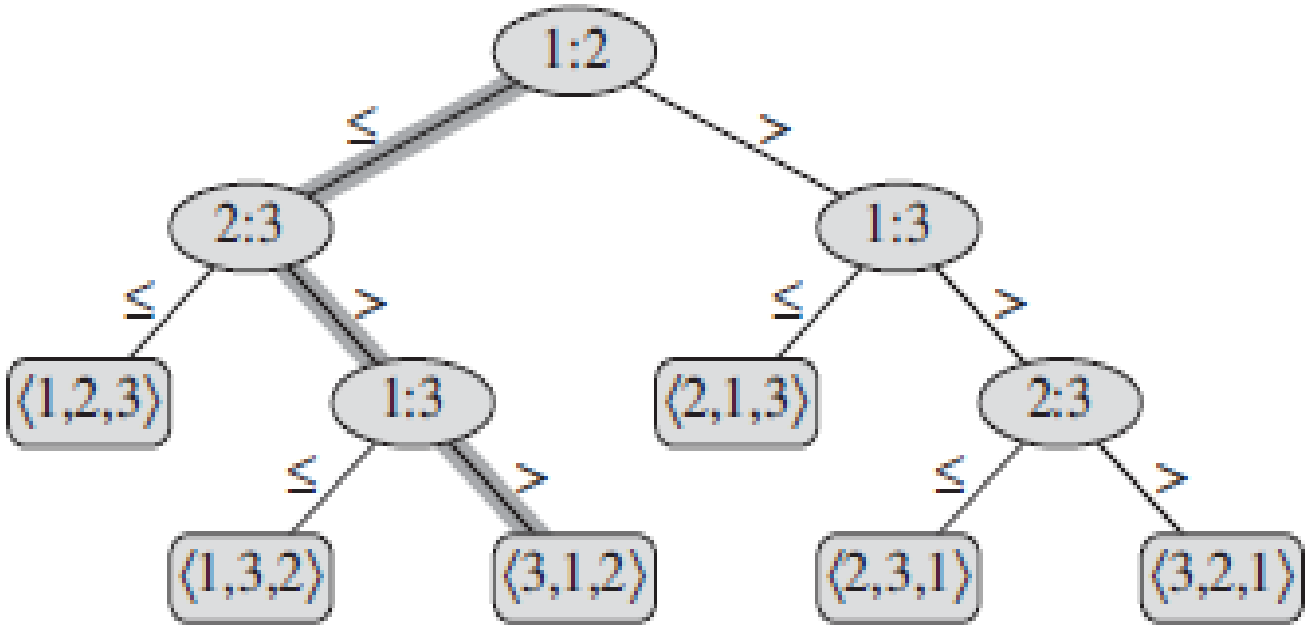
Sorting in Linear Time

In a decision tree, we annotate each internal node by **i:j** for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence. We also annotate each leaf by a permutation $< \boldsymbol{\pi}(1), \boldsymbol{\pi}(2), \boldsymbol{\pi}(3), \ldots\ldots\ldots\ldots, \boldsymbol{\pi}(n) >$.

- The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf.
- Each internal node indicates a comparison $a_i \leq a_j$. The left sub-tree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right sub-tree dictates subsequent comparisons knowing that $a_i > a_j$.
- When we come to a leaf, the sorting algorithm has established the ordering $<a_{\boldsymbol{\pi}(1)}, a_{\boldsymbol{\pi}(2)}, \ldots\ldots\ldots\ldots, a_{\boldsymbol{\pi}(n)}>$.

Sorting in Linear Time

The decision tree operating on three elements is the following:-



Sorting in Linear Time

Theorem:

Any comparison sort algorithm requires Ω(nlg n) comparisons in the worst case.

Proof: The worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.

Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the n! permutations of the input appears as some leaf, we have n! ≤ l . Since a binary tree of height h has no more than 2^h leaves, therefore

$$n! \leq l \leq 2h \;\Rightarrow\; h \geq \lg(n!)$$

Therefore $h = \Omega(n\lg n)$

Counting sort

- ***Counting sort*** assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.

- When $k = O(n)$, the sorting algo. runs in $\theta(n)$ time.

- Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array.

Counting sort

Example: Sort the following elements using counting sort

2, 5, 3, 0, 2, 3, 0, 3

Solution:

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

(c)

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

(d)

Counting sort

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

- Time complexity of counting sort is
  $T(n) = \theta(n+k)$
- If $k \leq n$, then $\mathbf{T(n) = \theta(n)}$

Stable sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Some stable sorting algorithms

1. Counting sort
2. Insertion sort
3. Merge Sort
4. Bubble Sort

Some unstable sorting algorithms

1. Selection sort
2. Quicksort
3. Heap sort

Radix sorting

- This sorting algorithm assumes all the numbers must be of equal number of digits.

- This algorithm sort all the elements on the basis of digits used in the number.

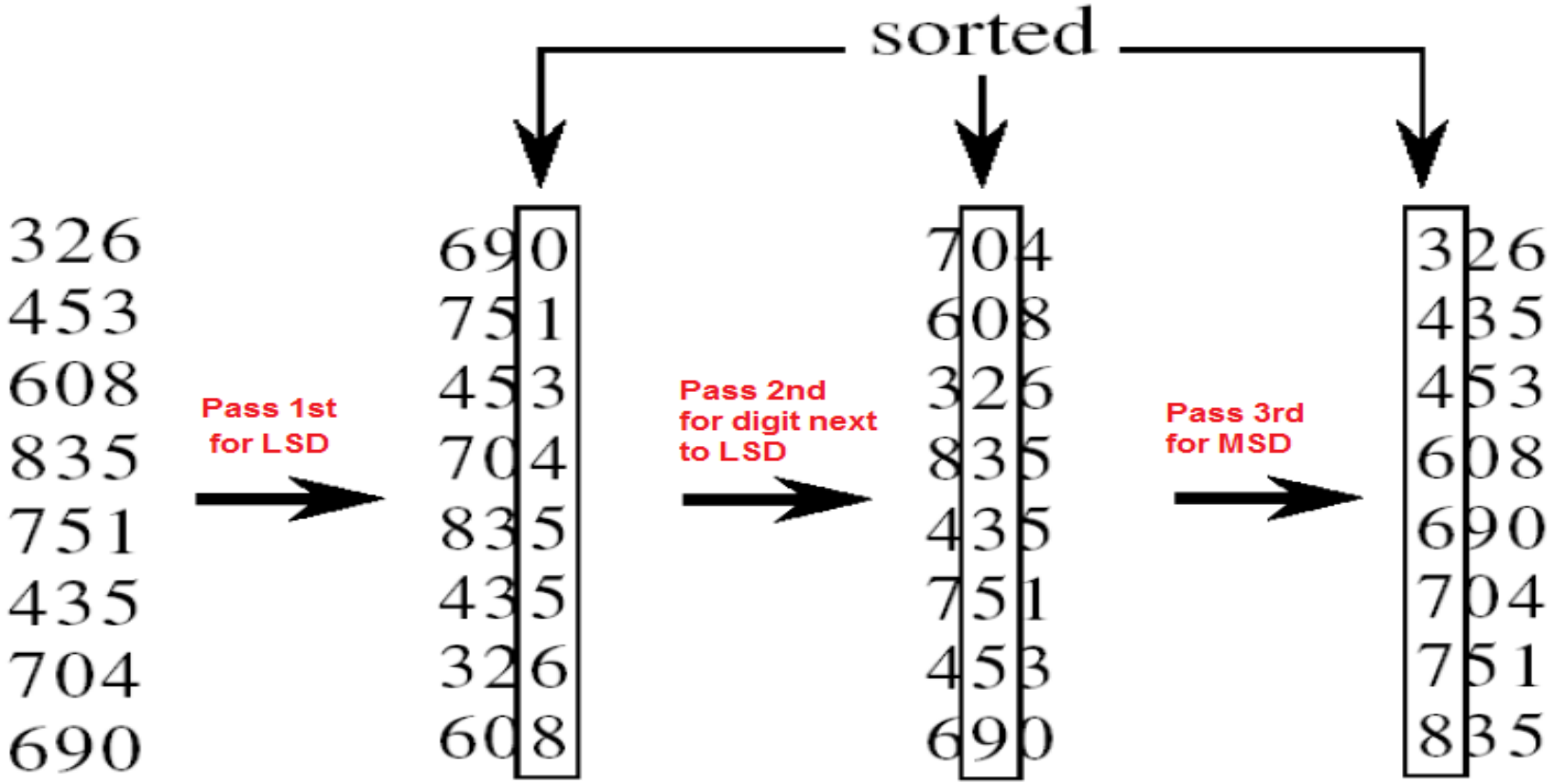
- First sort the numbers on the basis of least significant digit. Next time on the basis of 2nd least significant digit. After it on the basis of 3rd least significant digit. And so on. At the last, it sort on the basis of most significant digit.

Radix sorting

Example: Sort the following elements using radix sort

326, 453, 608, 835, 751, 435, 704, 690.

Solution: In these elements, number of digits is 3. Therefore, we have to used 3 iterations.



Radix sorting

```
radix_sort(A,d,k)
{
    for i=1 to d
        counting_sort(A,i,k)
}
```

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Radix sorting

Time complexity of radix sort is

$$T(n) = \theta(d(n+k))$$

Note: When d is constant and k=O(n), we can make radix sort run in linear time i.e $T(n) = \theta(n)$.

Bucket sorting

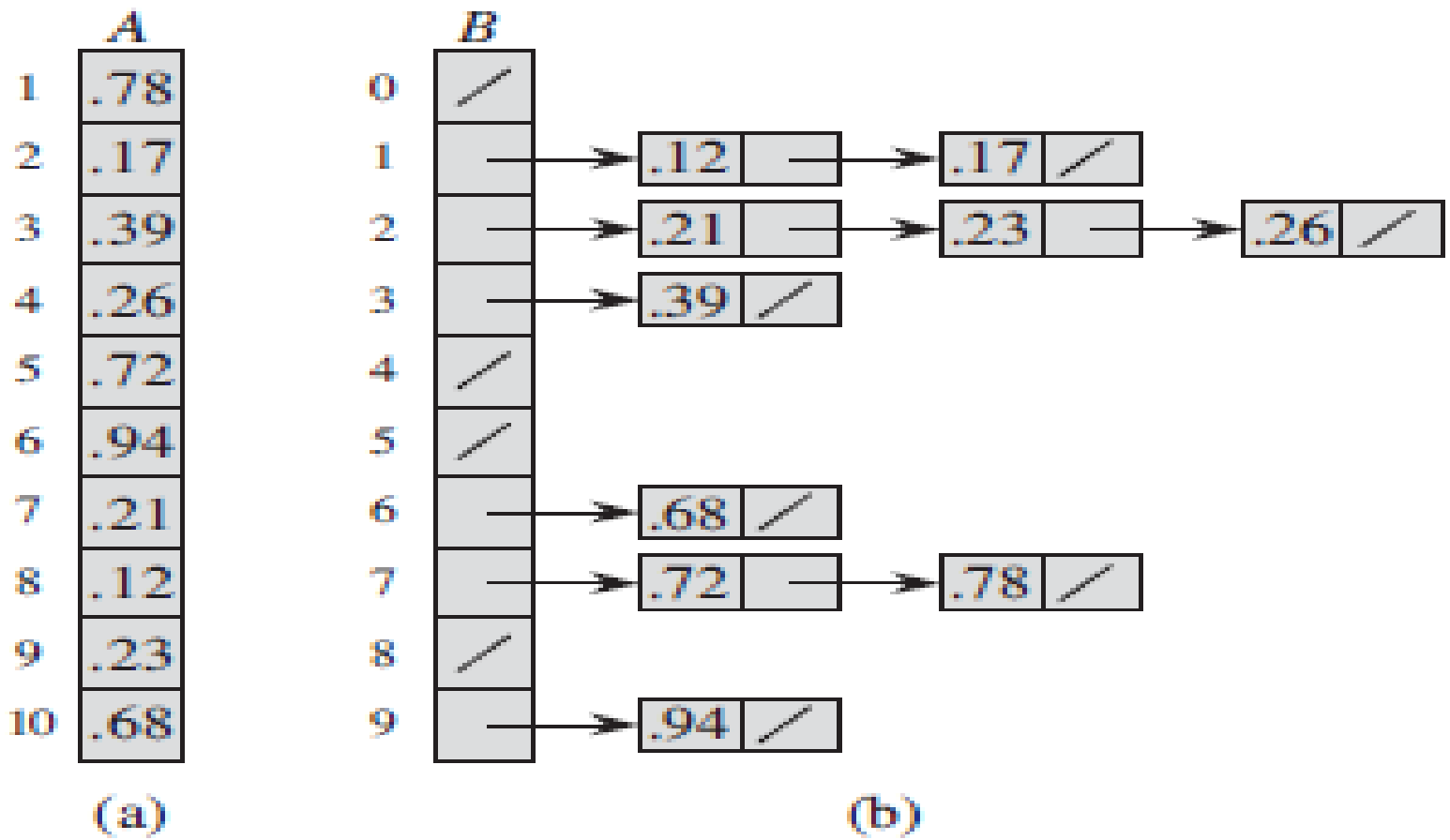
- Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1).

- Bucket sort divides the interval [0,1) into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0,1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Bucket sorting

Example: Sort the following elements using bucket sort

0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68



Bucket sorting

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Time complexity $\mathbf{T(n) = \theta(n)}$

Linear search

- It is a simple search algorithm that search a target value in an array or list.
- It sequentially scan the array, comparing each array item with the search element.
- If search element is found in an array, then the index of an element is returned.
- The time complexity of linear search is O(n).
- It can be applied to both sorted and unsorted array.

Binary search

- **Binary search** is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems.

- Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted.

- When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.

Binary search process

Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
23 > 16 take 2nd half	L=0	1	2	3	M=4	5	6	7	8	H=9
	2	5	8	12	16	23	38	56	72	91
23 > 56 take 1st half	0	1	2	3	4	L=5	6	M=7	8	H=9
	2	5	8	12	16	23	38	56	72	91
Found 23, Return 5	0	1	2	3	4	L=5, M=5	H=6	7	8	9
	2	5	8	12	16	23	38	56	72	91

Binary search

```
Binary-search(A, n, x)
l = 1
r = n
while l ≤ r
do
        m = ⌊(l + r) / 2⌋
        if A[m] < x then
                l = m + 1
        else if A[m] > x then
                r = m − 1
        else
                return m
return unsuccessful
```

Time complexity **$T(n) = O(\lg n)$**

Thank you.

Design and Analysis of Algorithms

Unit-2

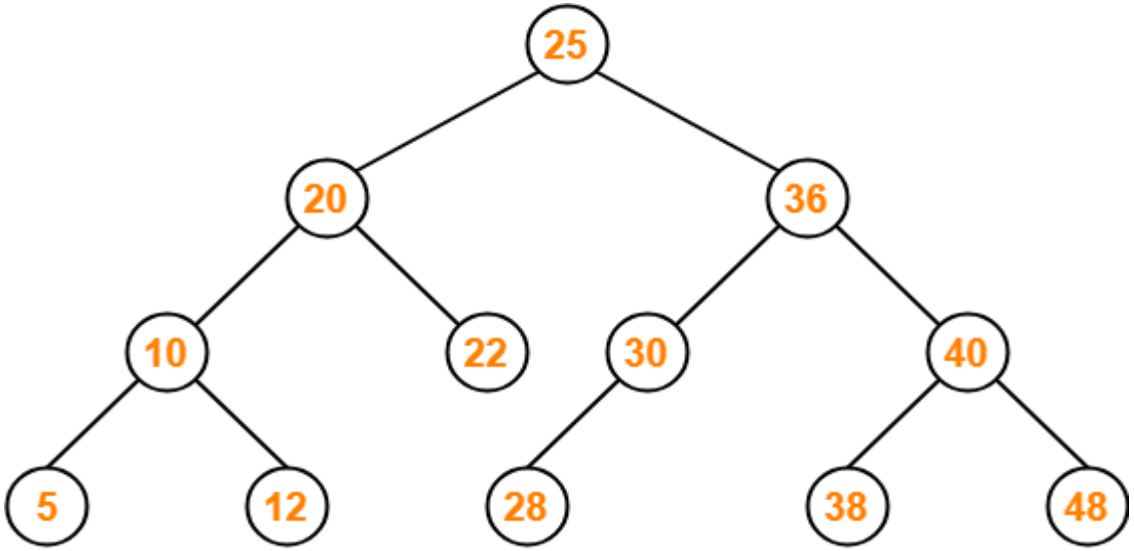
Red-Black Tree

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

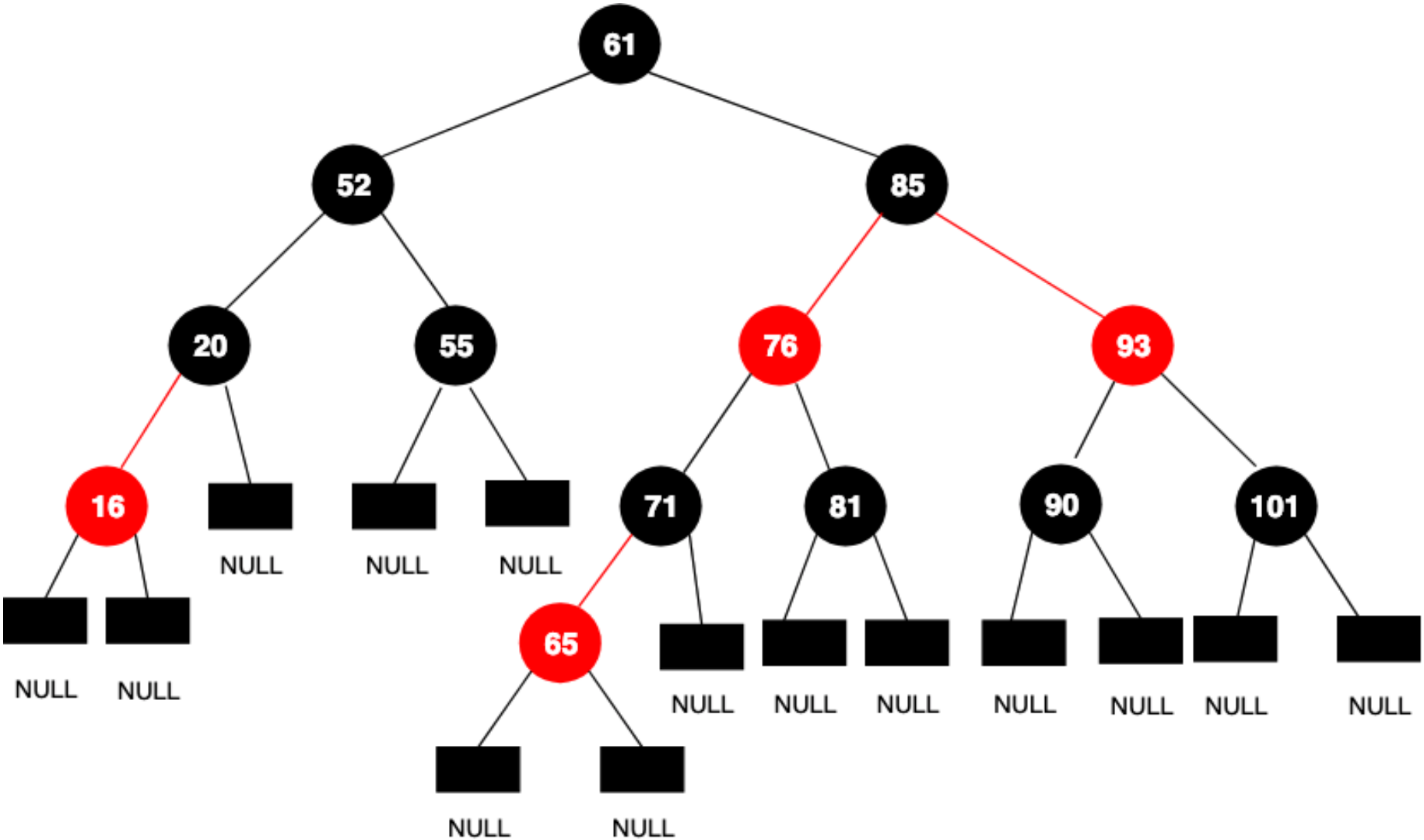
Definition

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

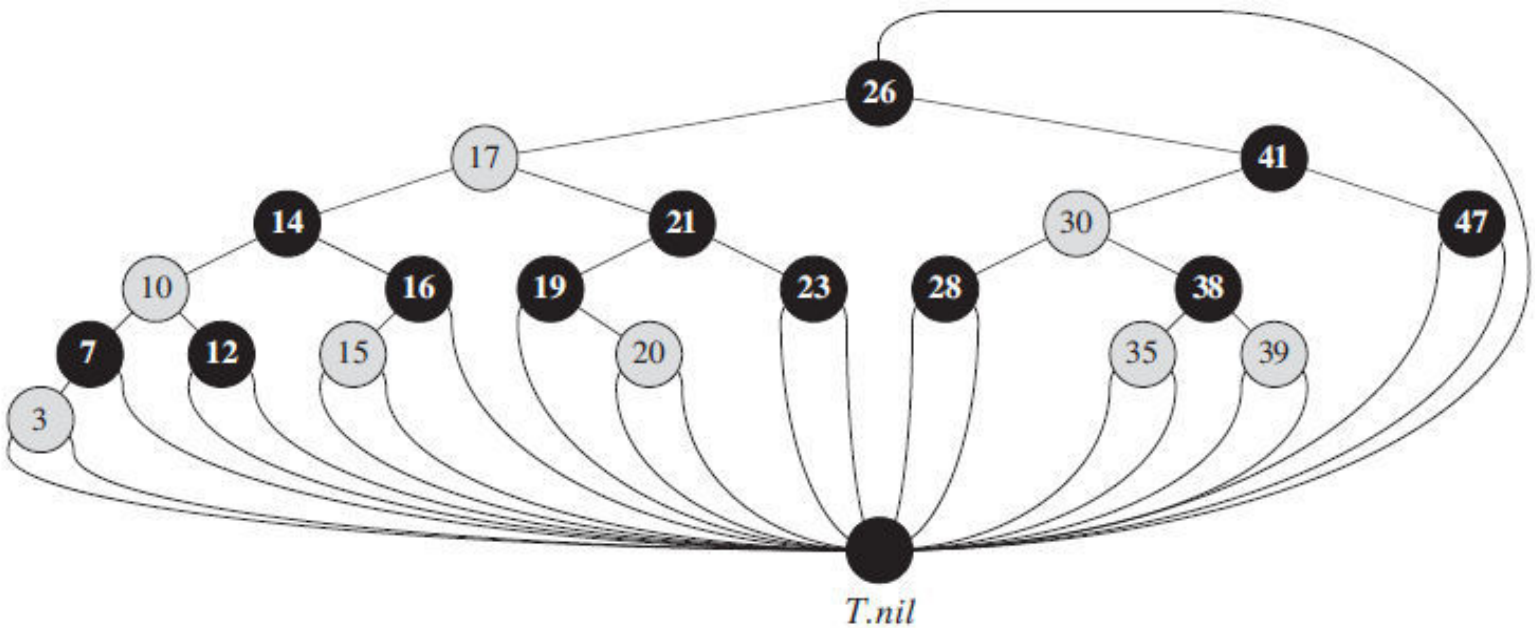
Example:



Red-Black Tree

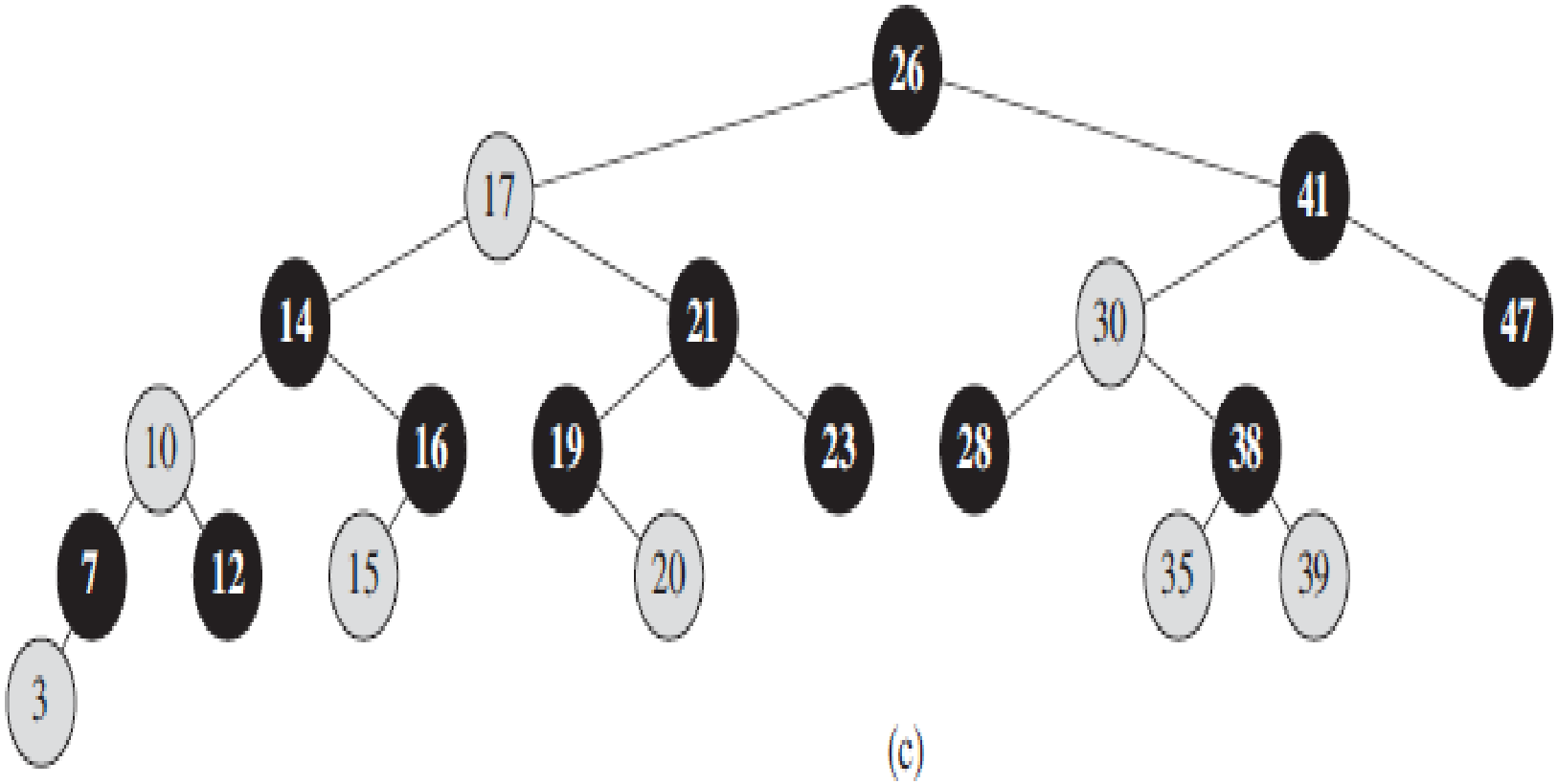
Sentinel

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

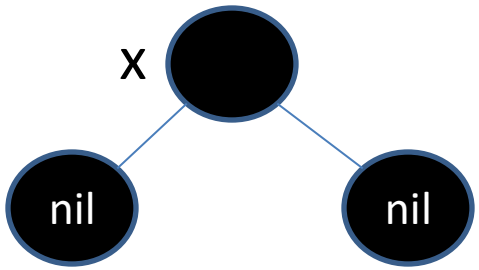
and $2^{bh(x)}-1 = 2^0-1 = 1-1 = 0$

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 $2\lg(n+1)$.

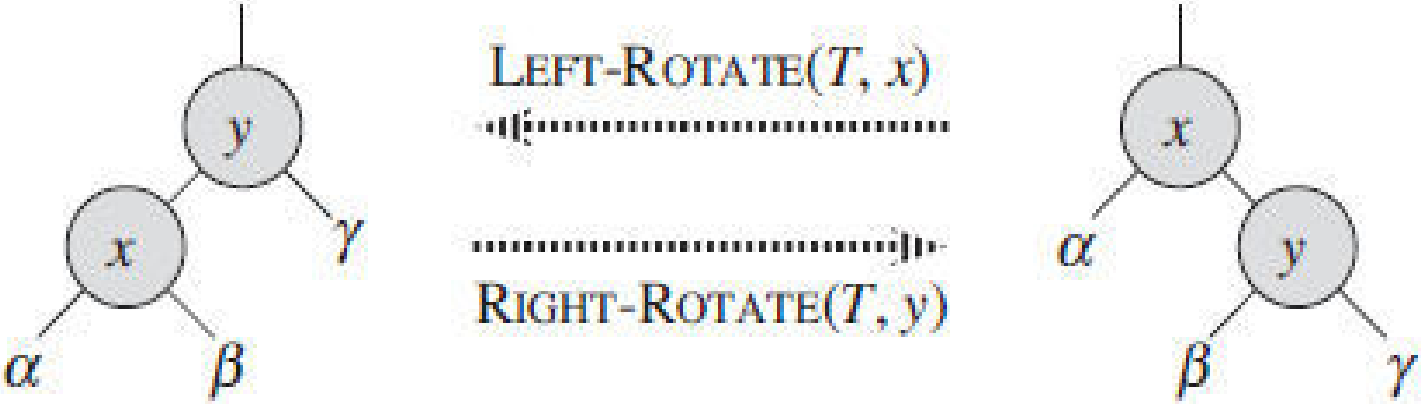
Now, it is proved.

Rotation operation

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

(1) Left rotation

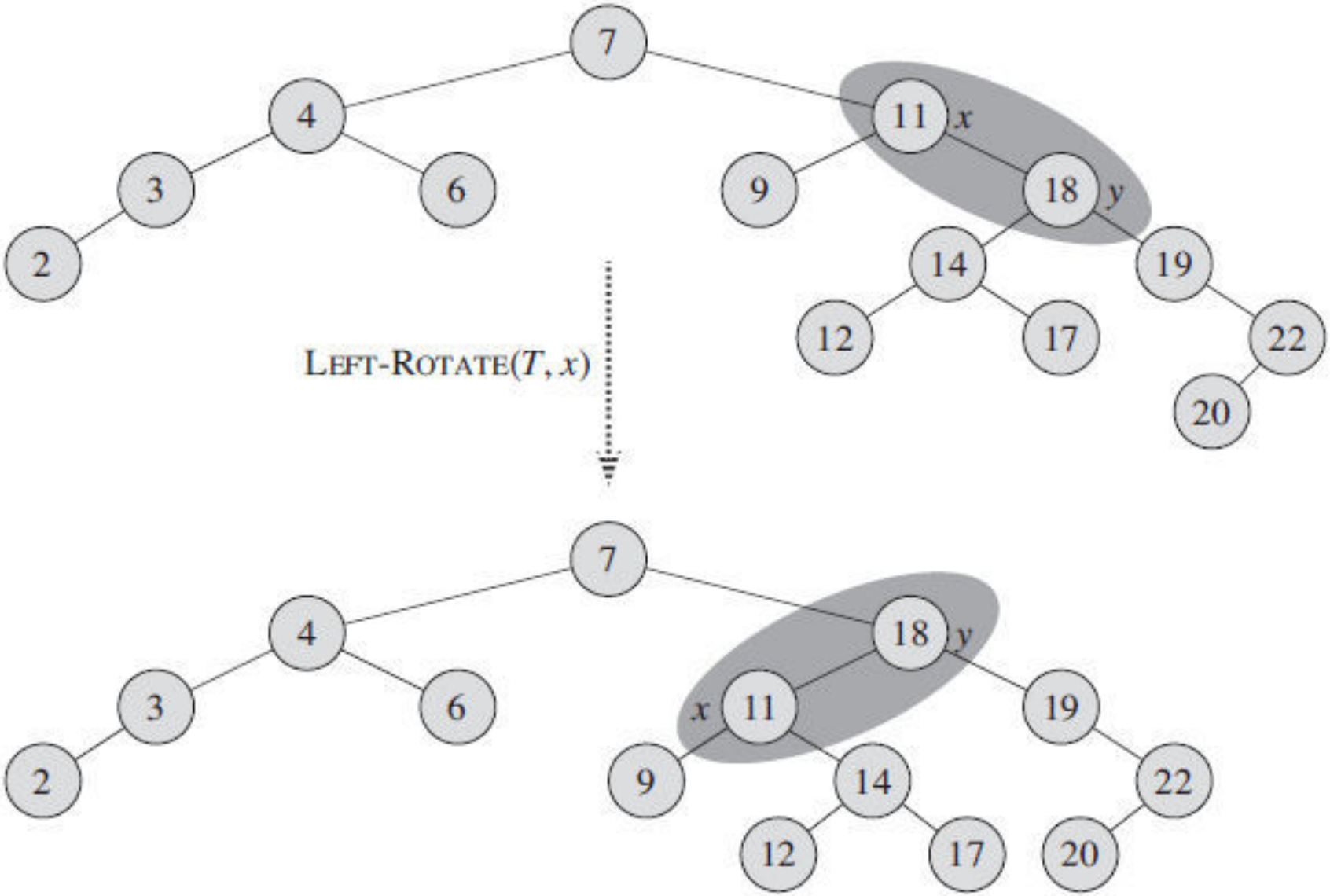
(2) Right rotation



Left rotation algorithm

```
LEFT-ROTATE(T, x)
y = x.right                    // set y
x.right = y.left               // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                      // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                     // put x on y's left
x.p = y
```

Left rotation algorithm



Right rotation algorithm

```
RIGHT-ROTATION(T, y)
    x = y.left
    y.left = x.right
    if x.right ≠ T.nil
        x.right.p = y
    x.p = y.p
    if y.p = = T.nil
        T.root = x
    else if  y = = y.p.right
        y.p.right = x
    else  y.p.left = x
    x.right = y
    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**

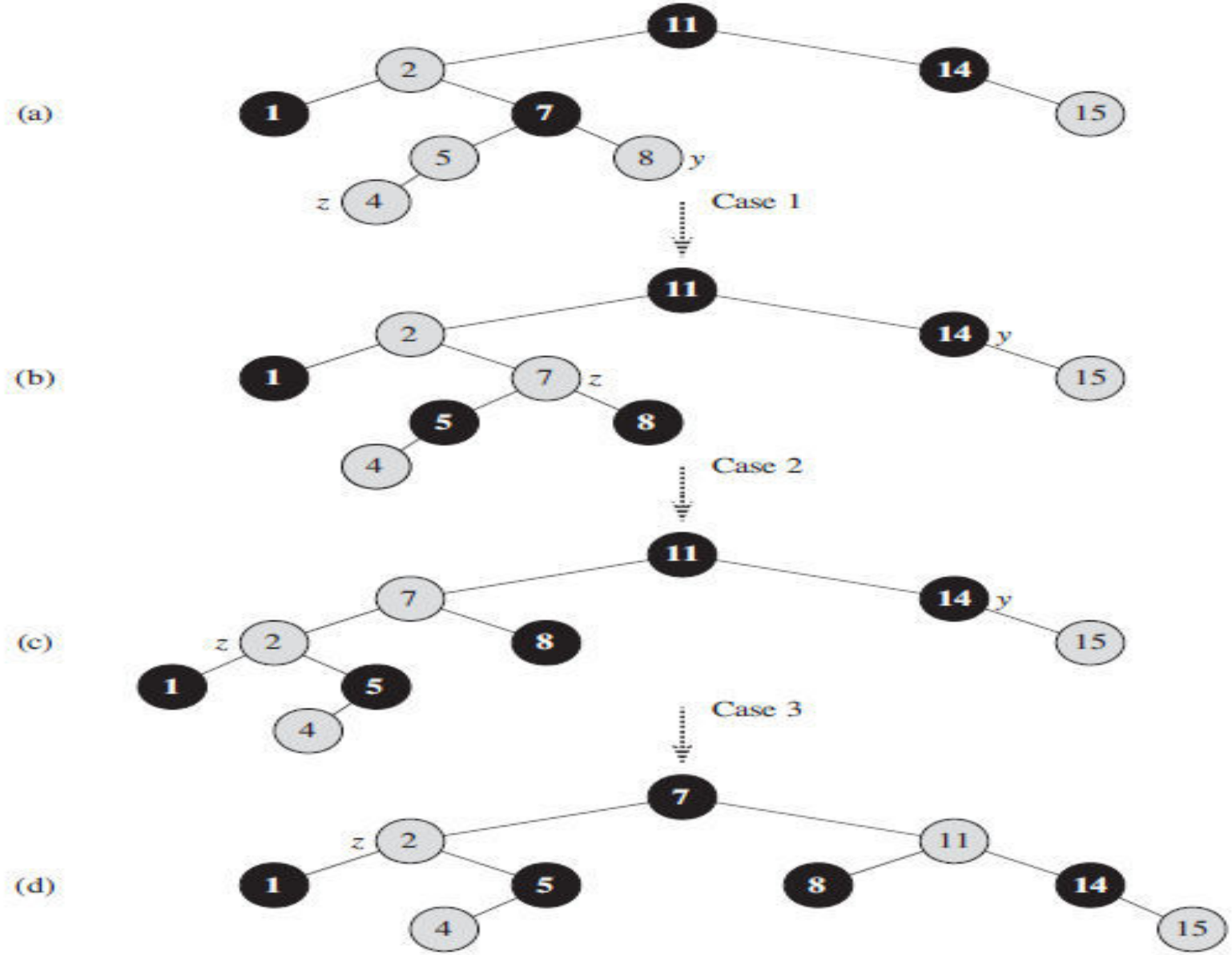
(2) z.p.p.color = **red**

(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

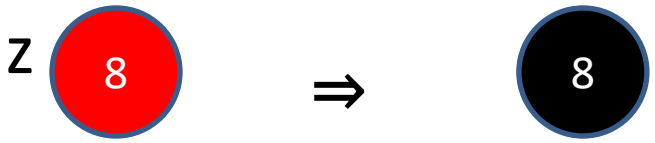


Insertion operation

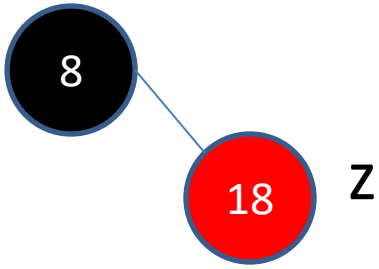
Example: Create the red-black tree by inserting following sequence of numbers:- 8, 18, 5, 15, 17, 25, 40 and 80.

Initially, red-black tree is empty.

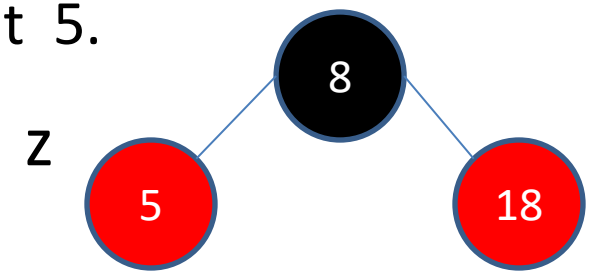
Solution: Initially tree is empty. First insert element 8.



Now, Insert next element 18.

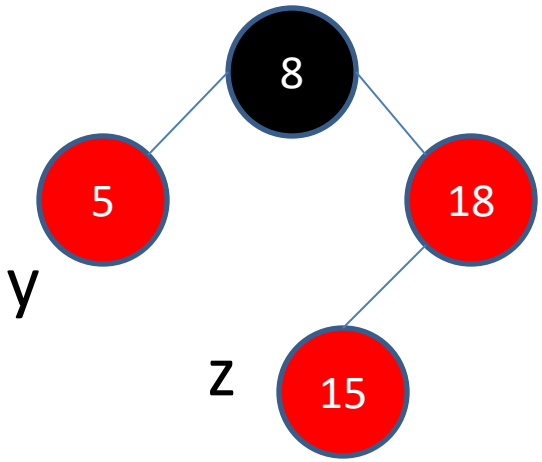


Now, Insert next element 5.

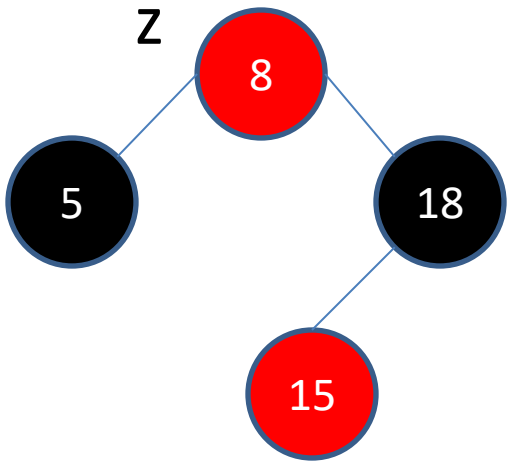


Insertion operation

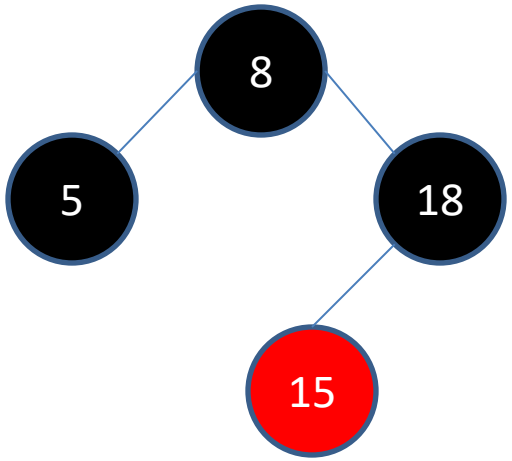
Now, Insert next element 15.



⇒

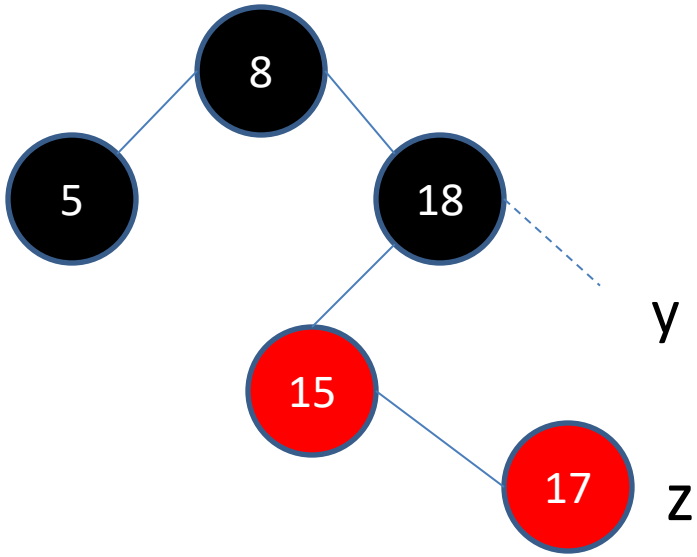


⇒

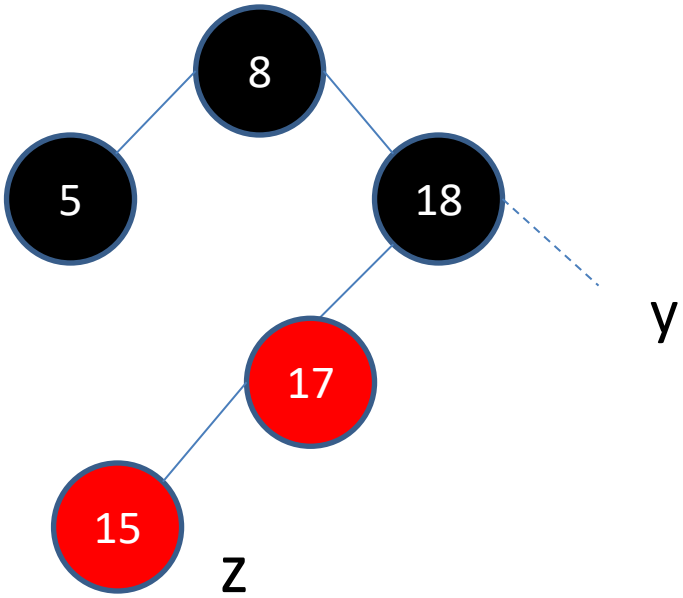


Insertion operation

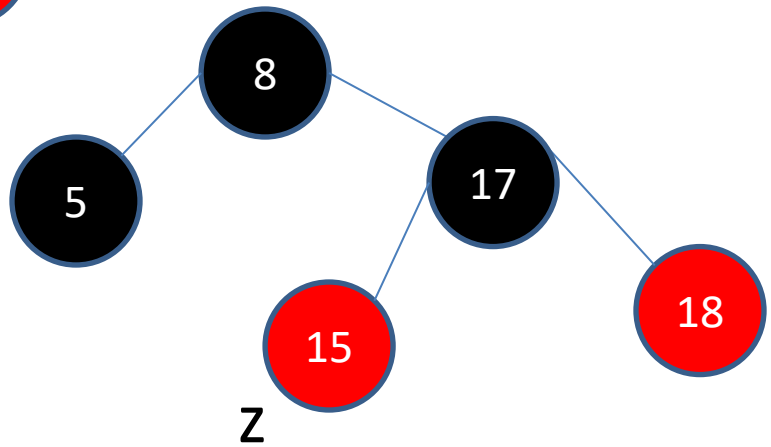
Now, Insert next element 17.



⇒

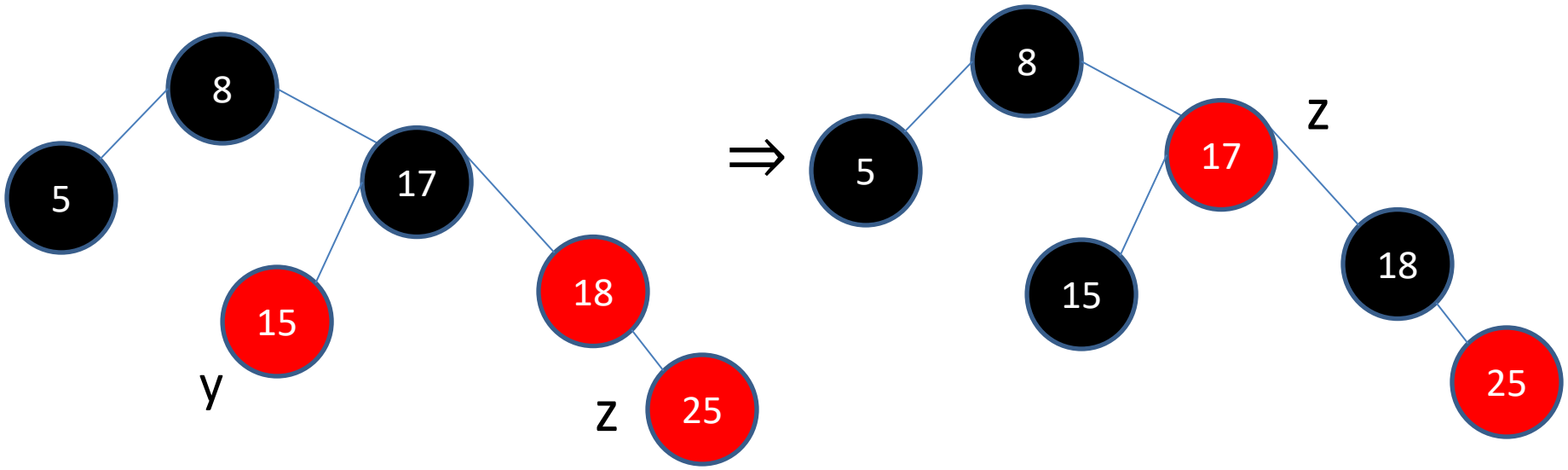


⇒



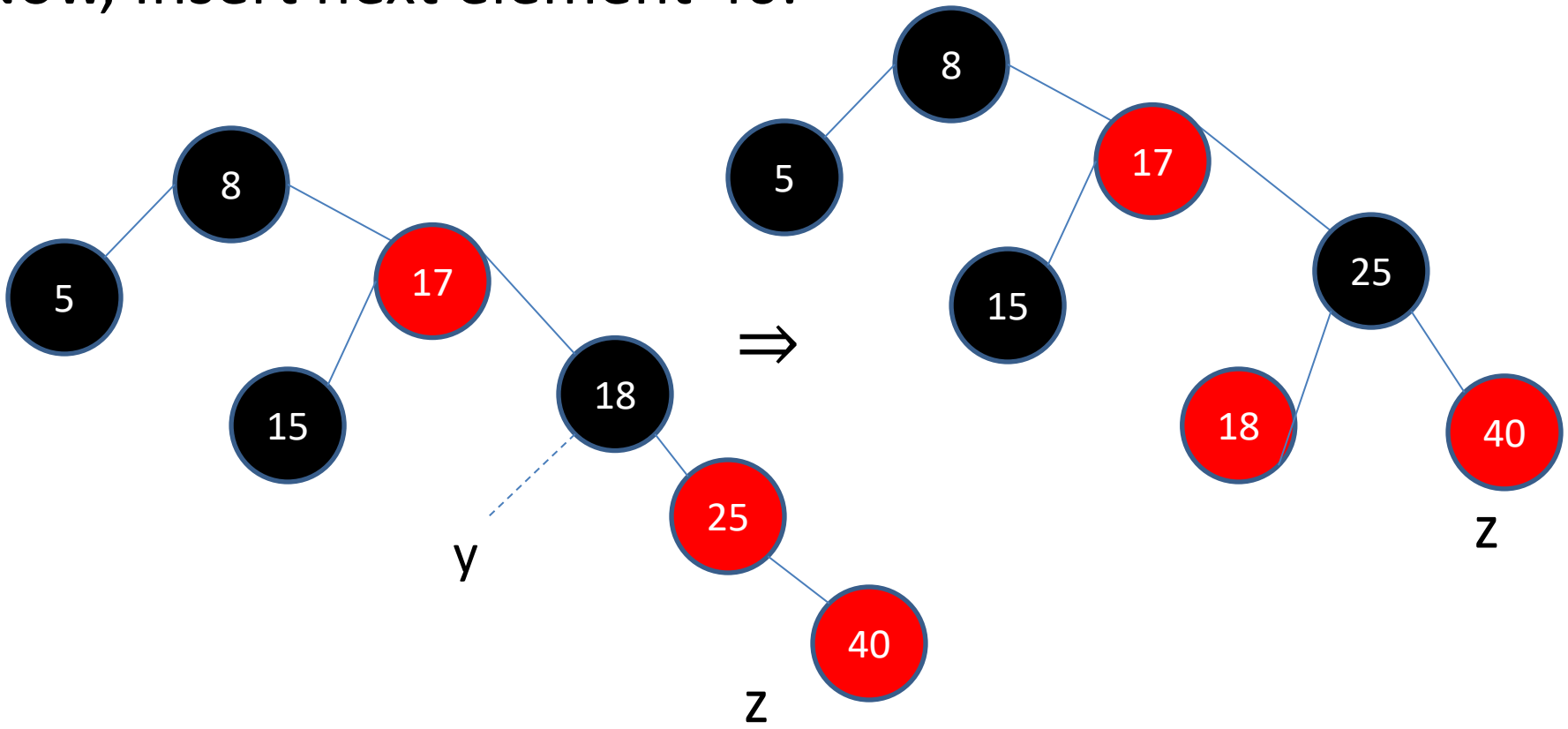
Insertion operation

Now, Insert next element 25.



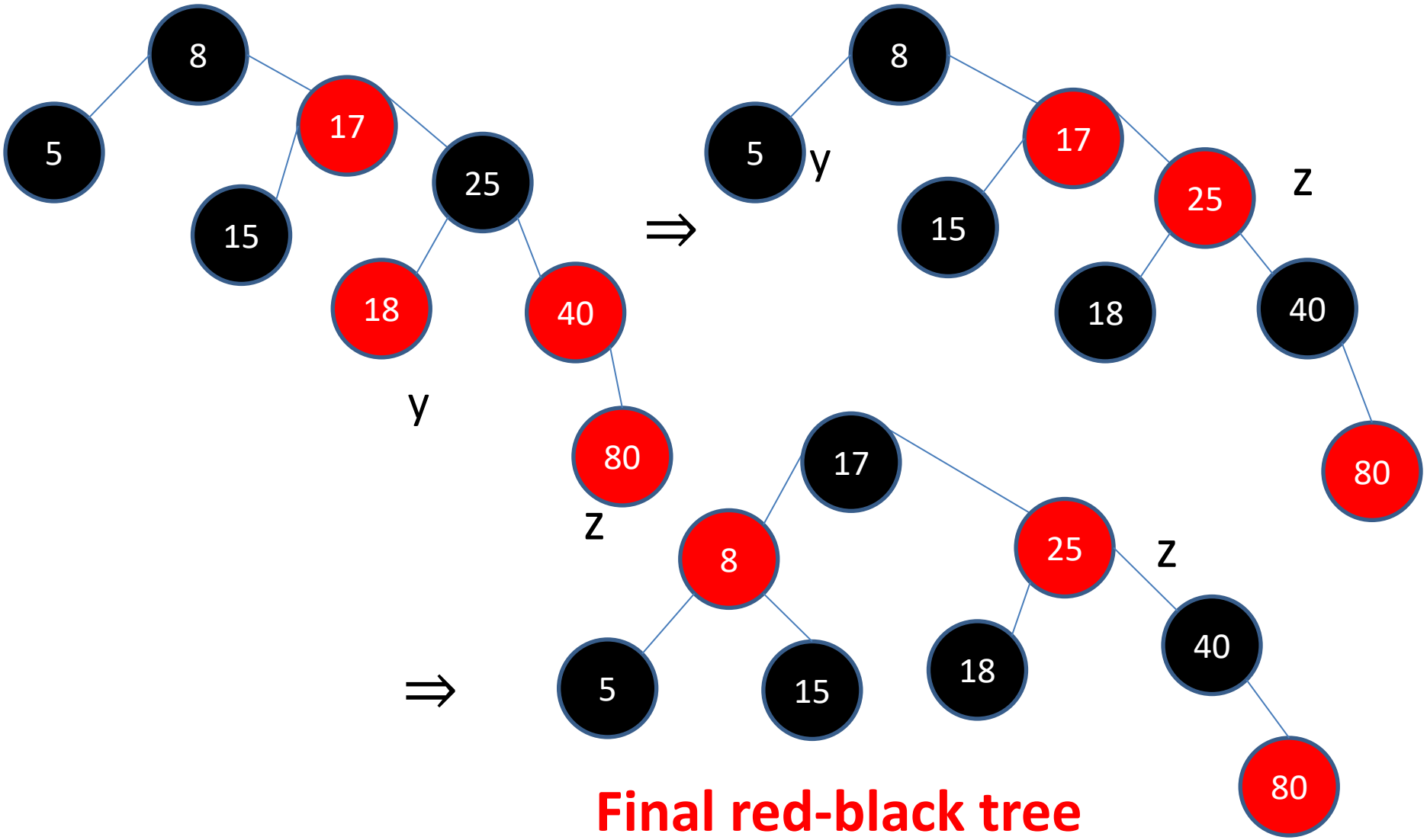
Insertion operation

Now, Insert next element 40.



Insertion operation

Now, Insert next element 80.



Final red-black tree

Insertion Algorithm

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

Insertion Algorithm

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
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

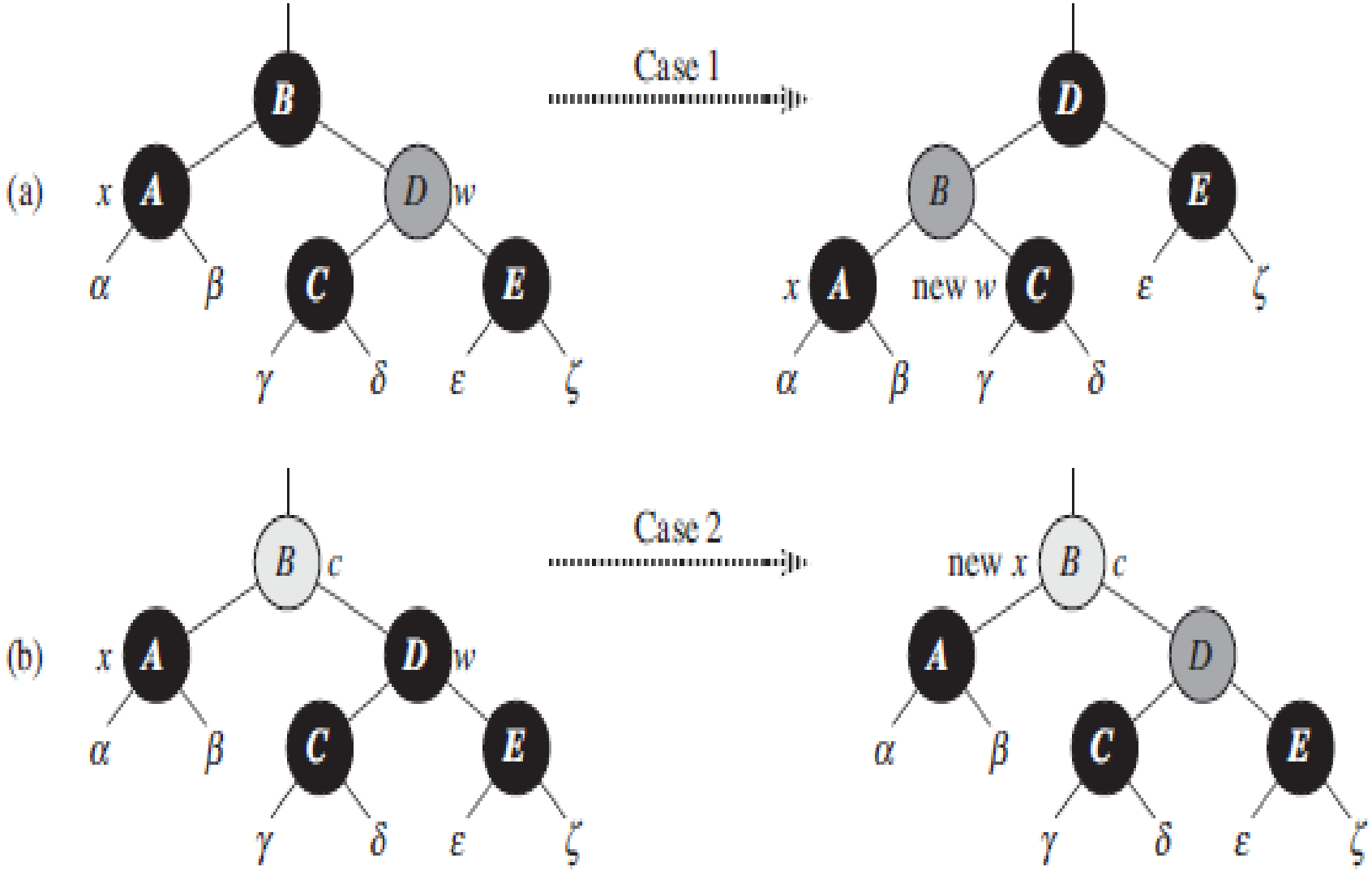
(3) x.p.color = **black**

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

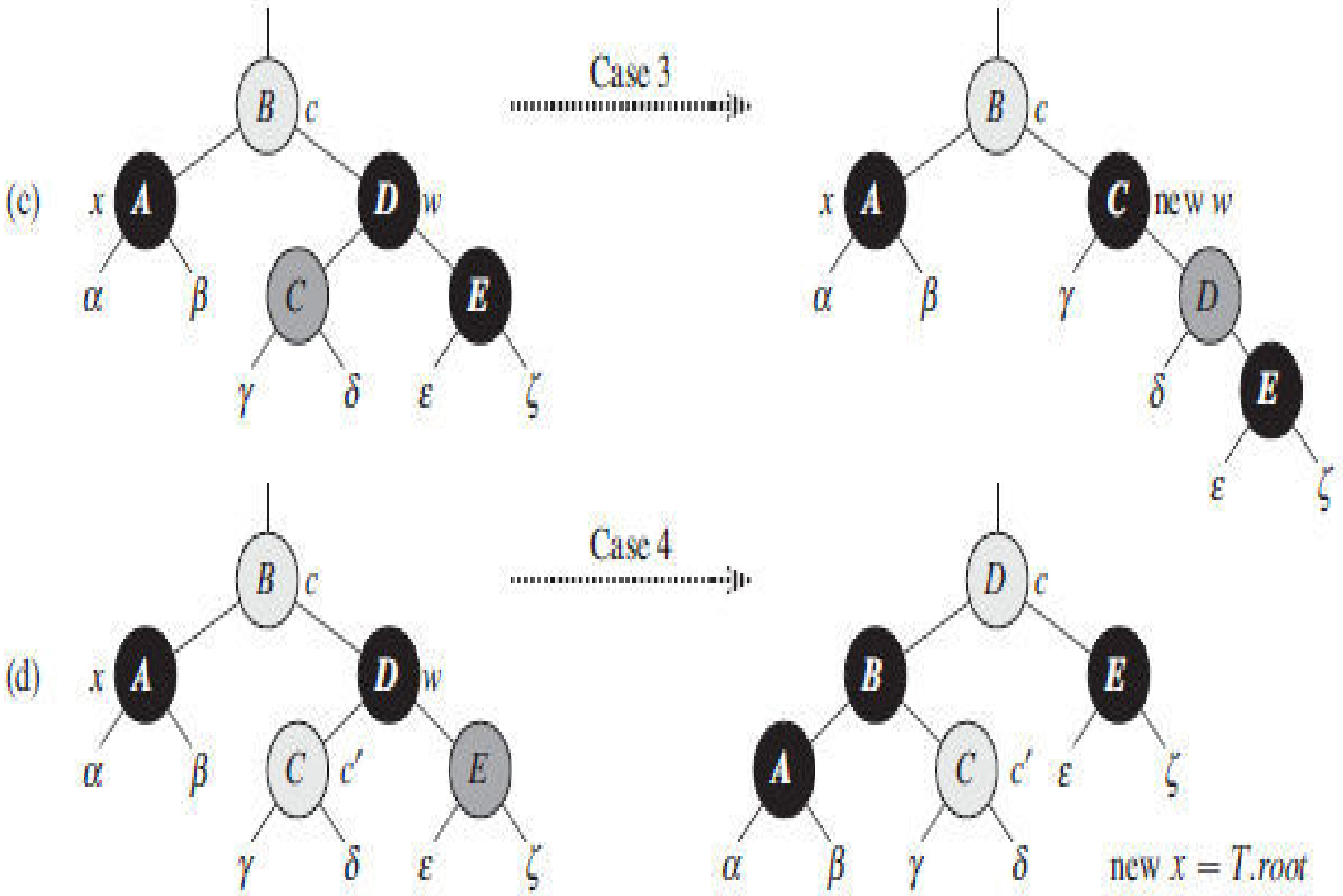
(5) x = T.root

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

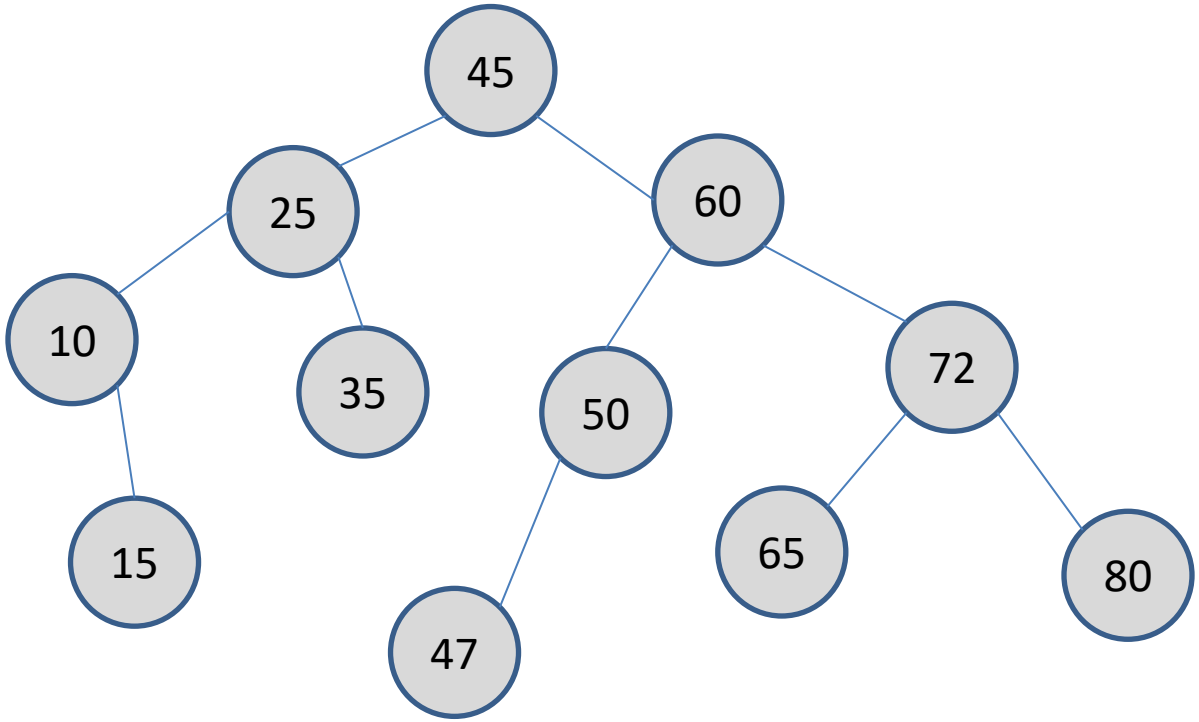
Deletion operation



Deletion operation

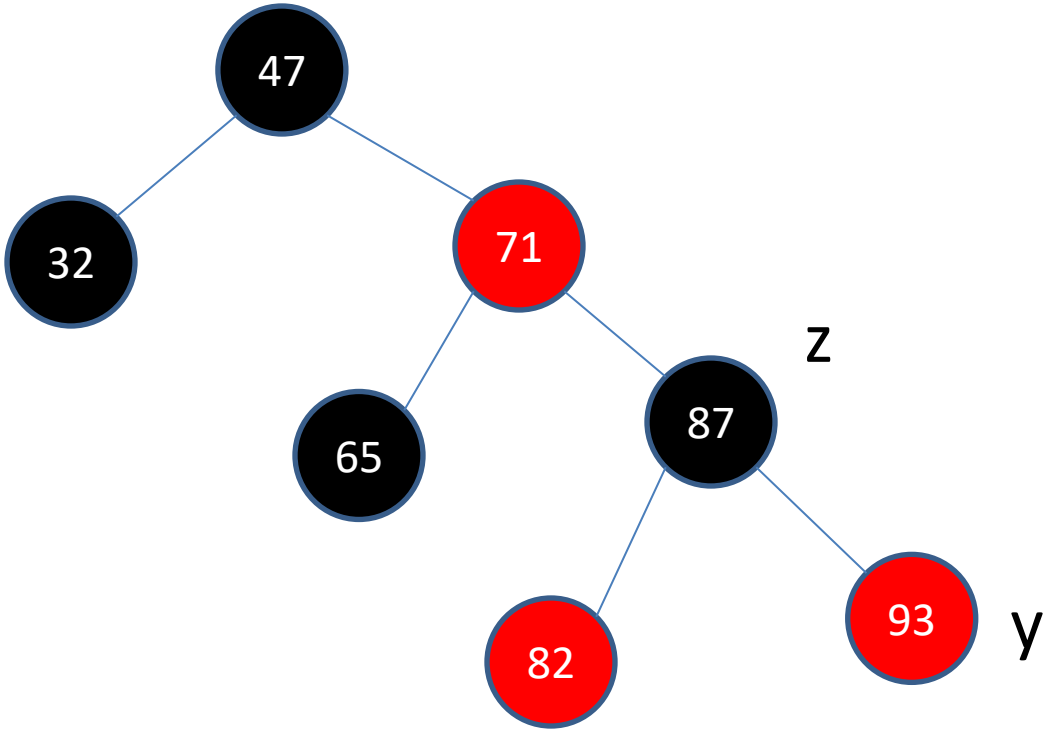


Binary search tree deletion



Deletion operation

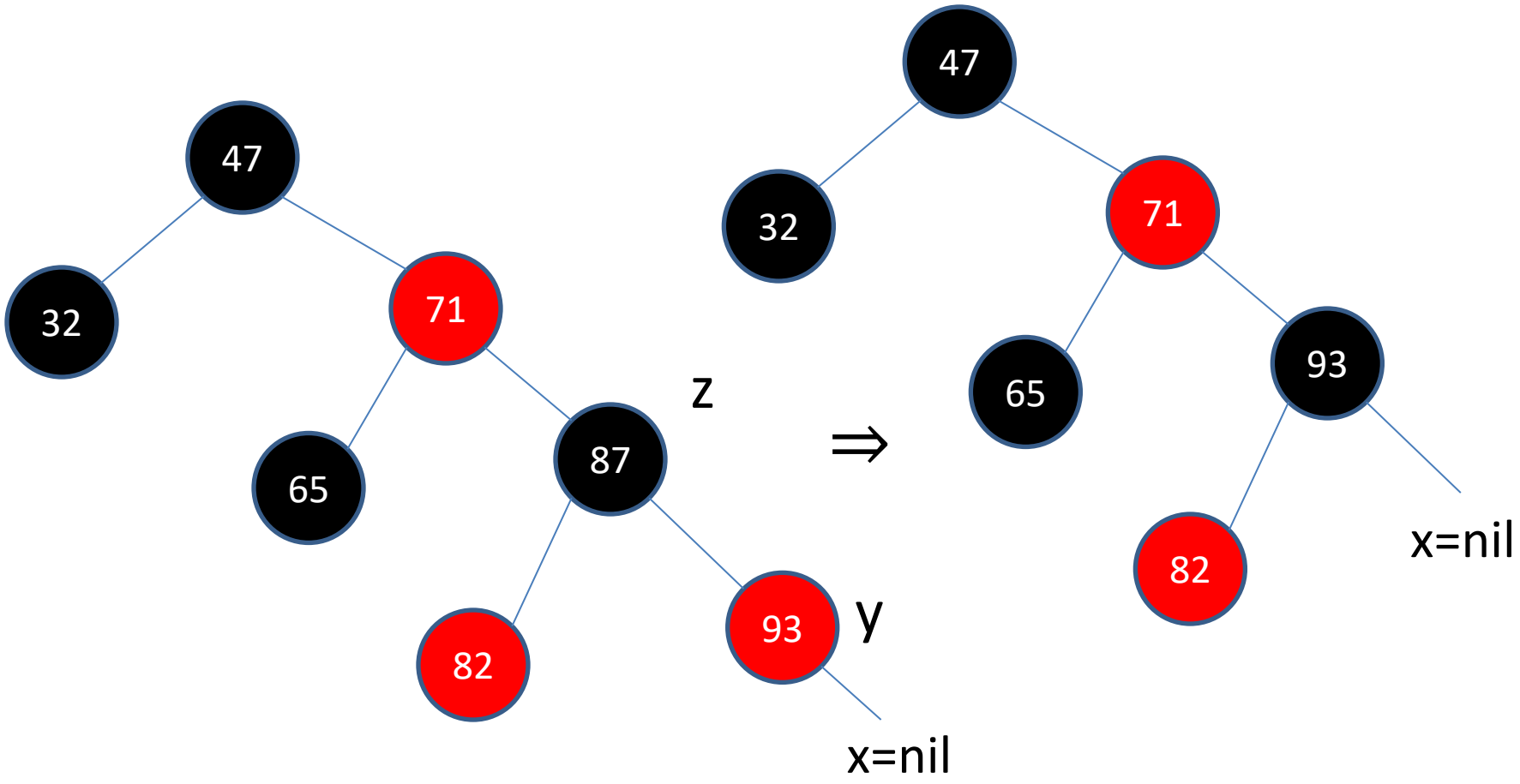
Example: Consider the following red-black tree



Delete the element 87, 32 and 71 in order.

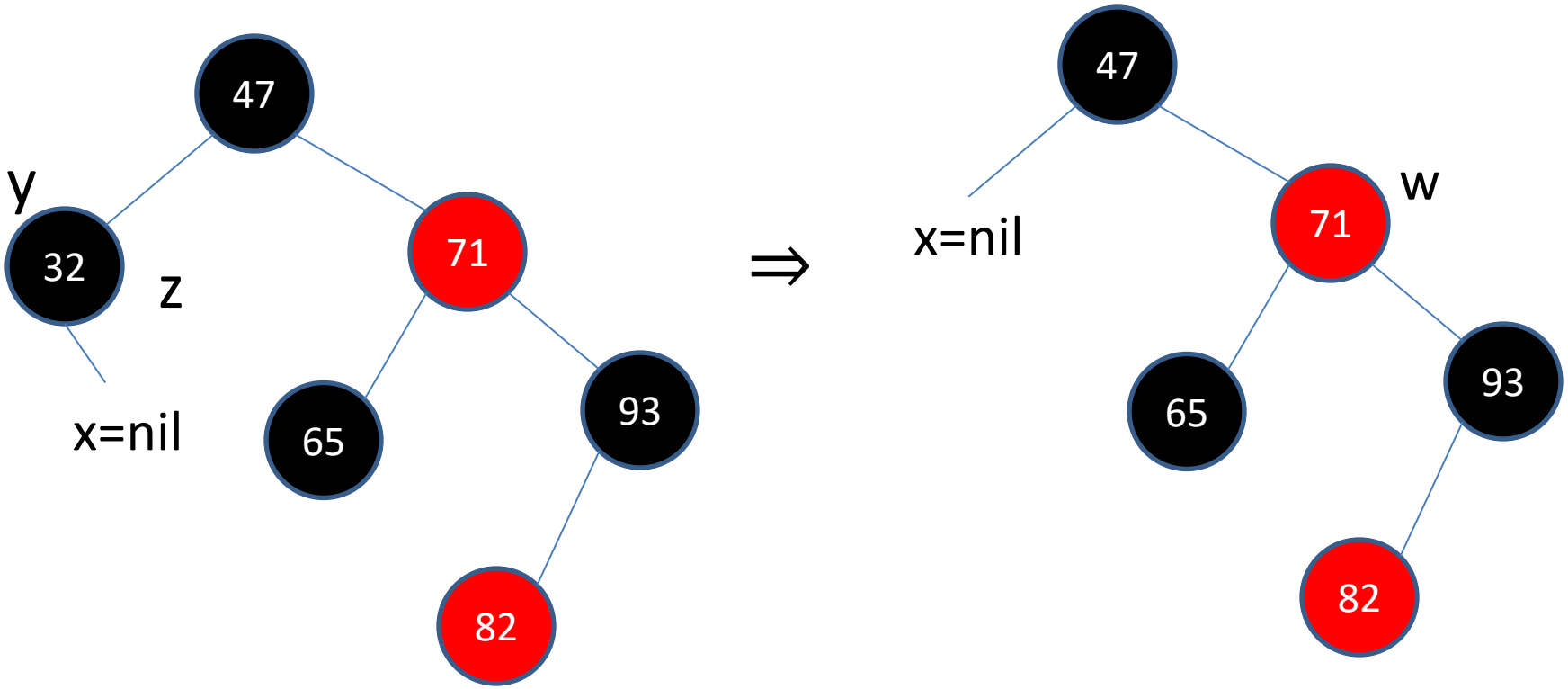
Deletion operation

Deletion of 87



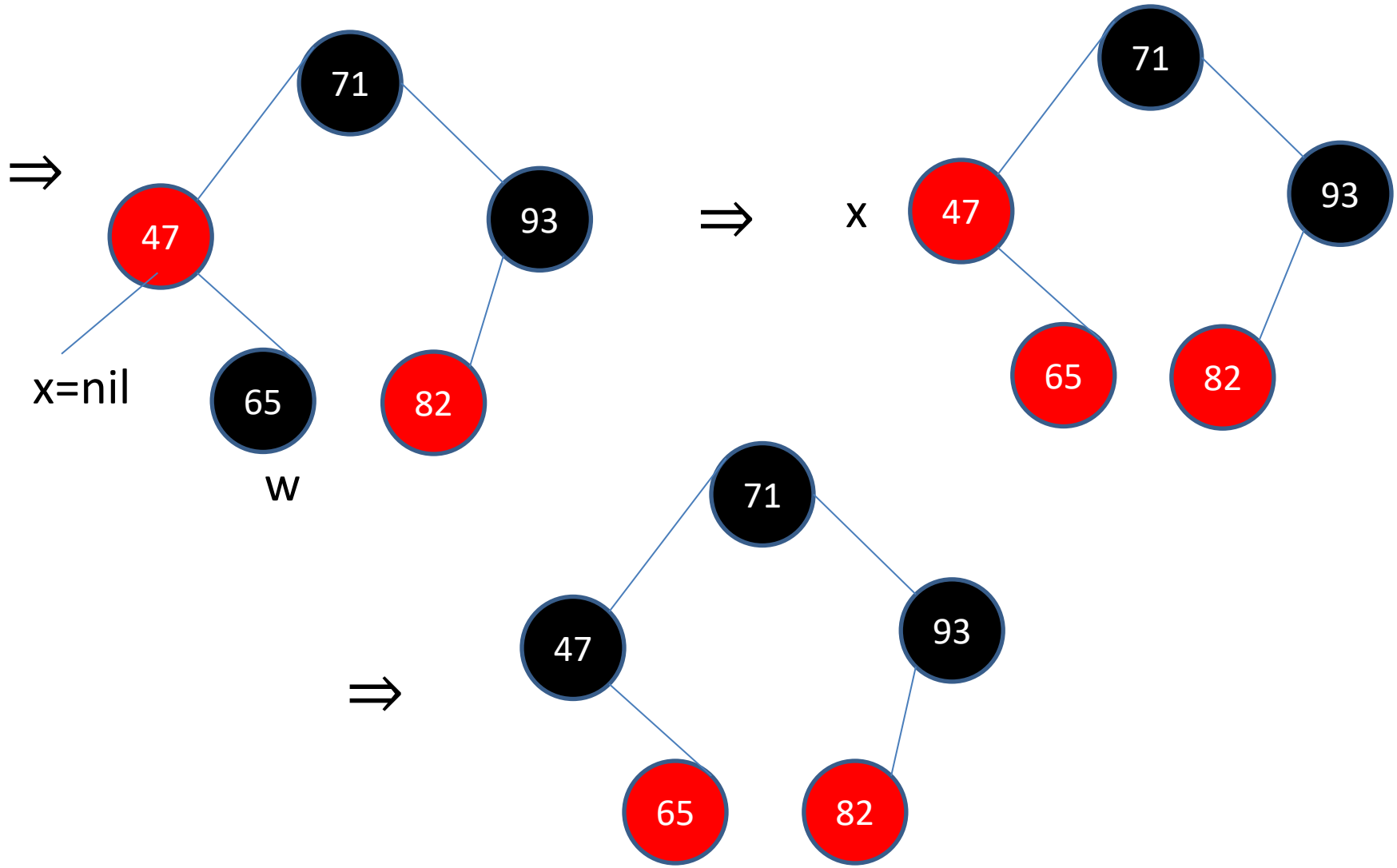
Deletion operation

Deletion of 32



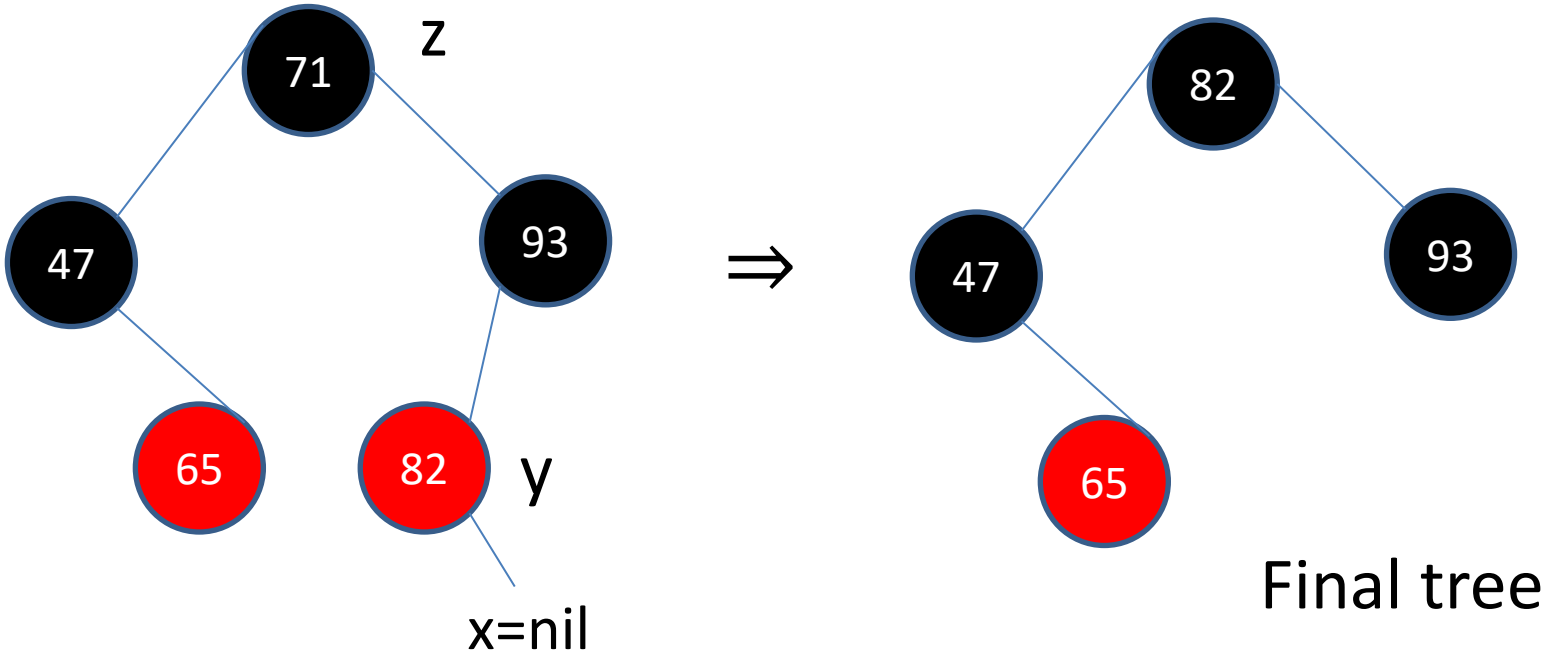
Deletion operation

Deletion of 32(continue)



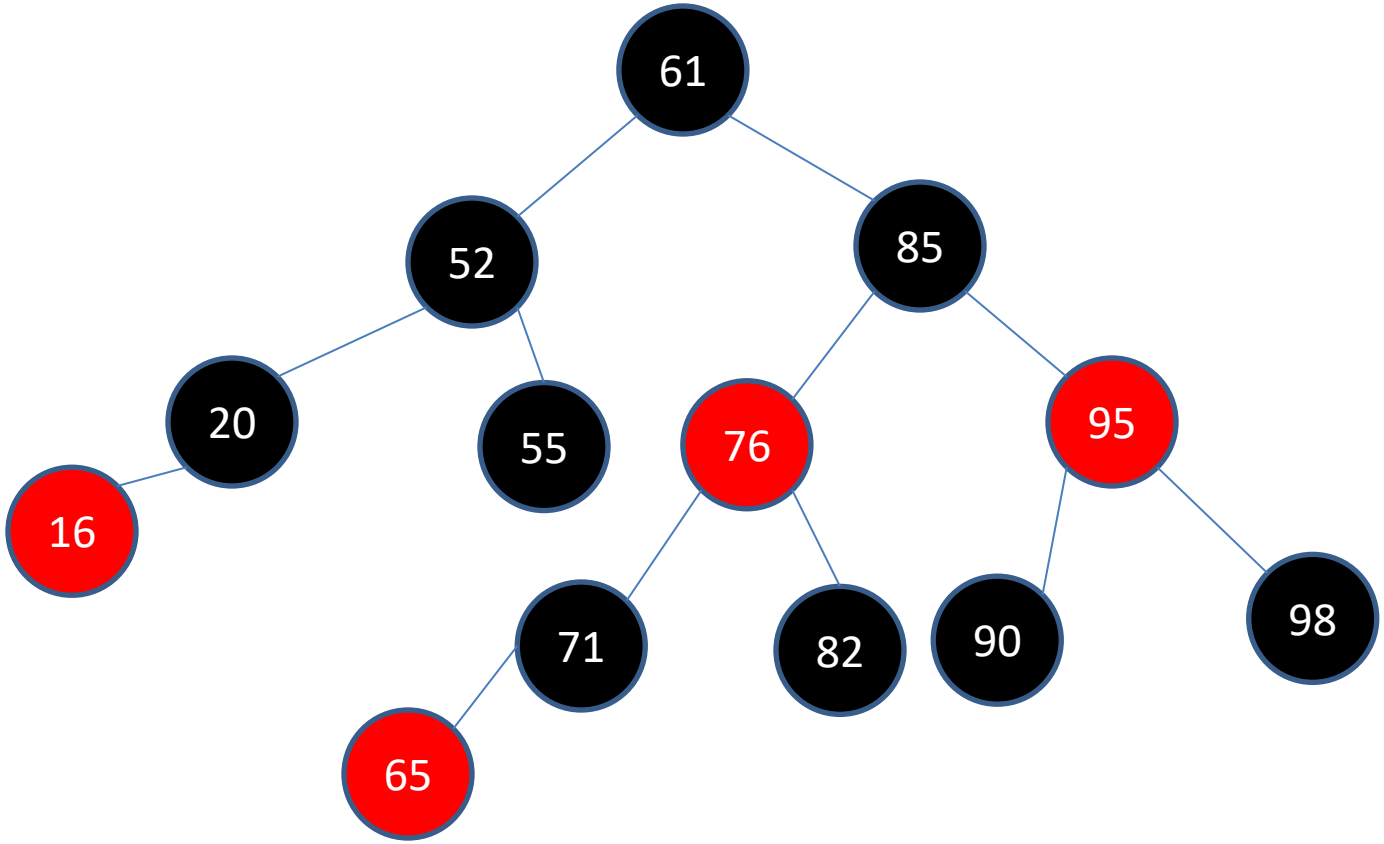
Deletion operation

Deletion of 71



Deletion operation

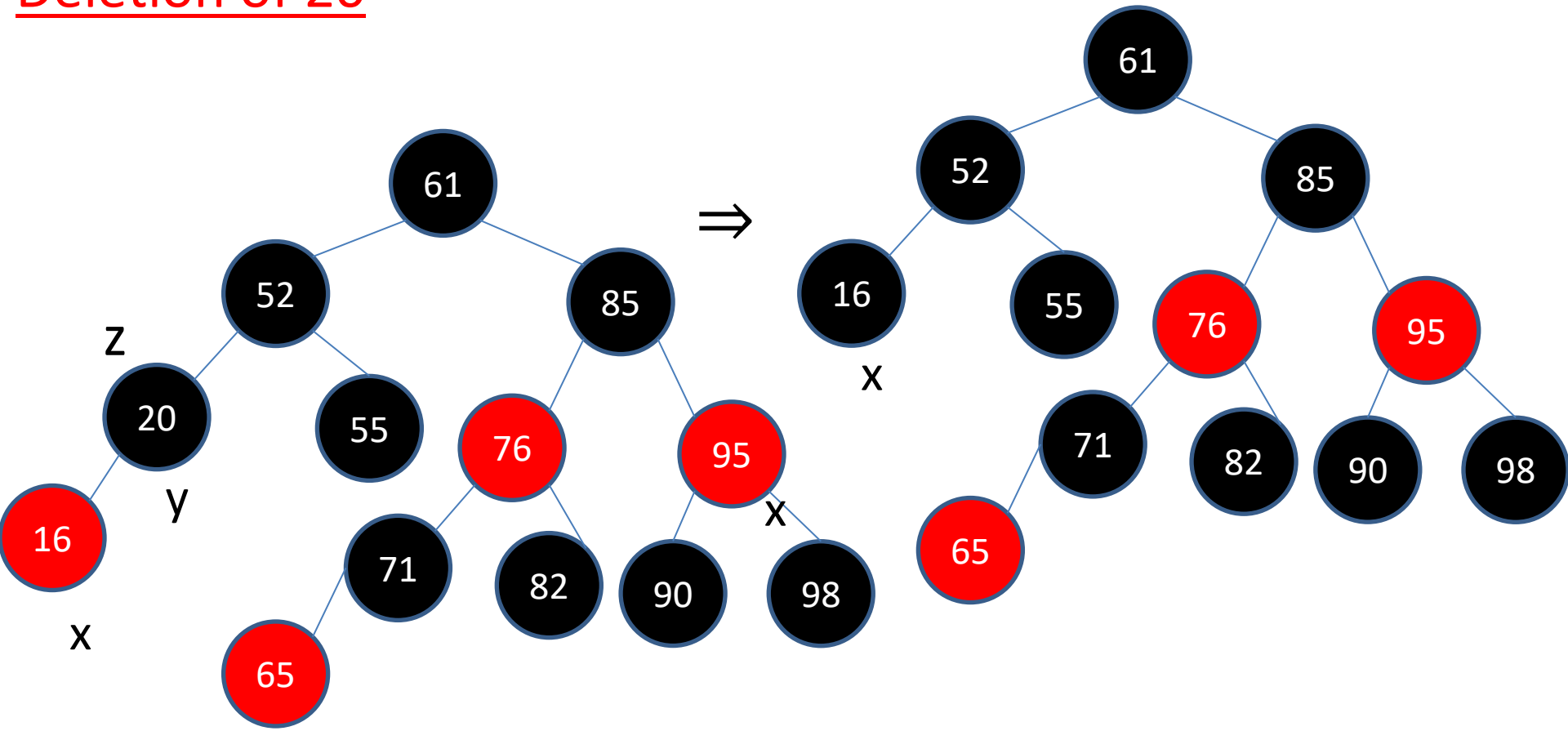
Example: Consider the following red-black tree



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

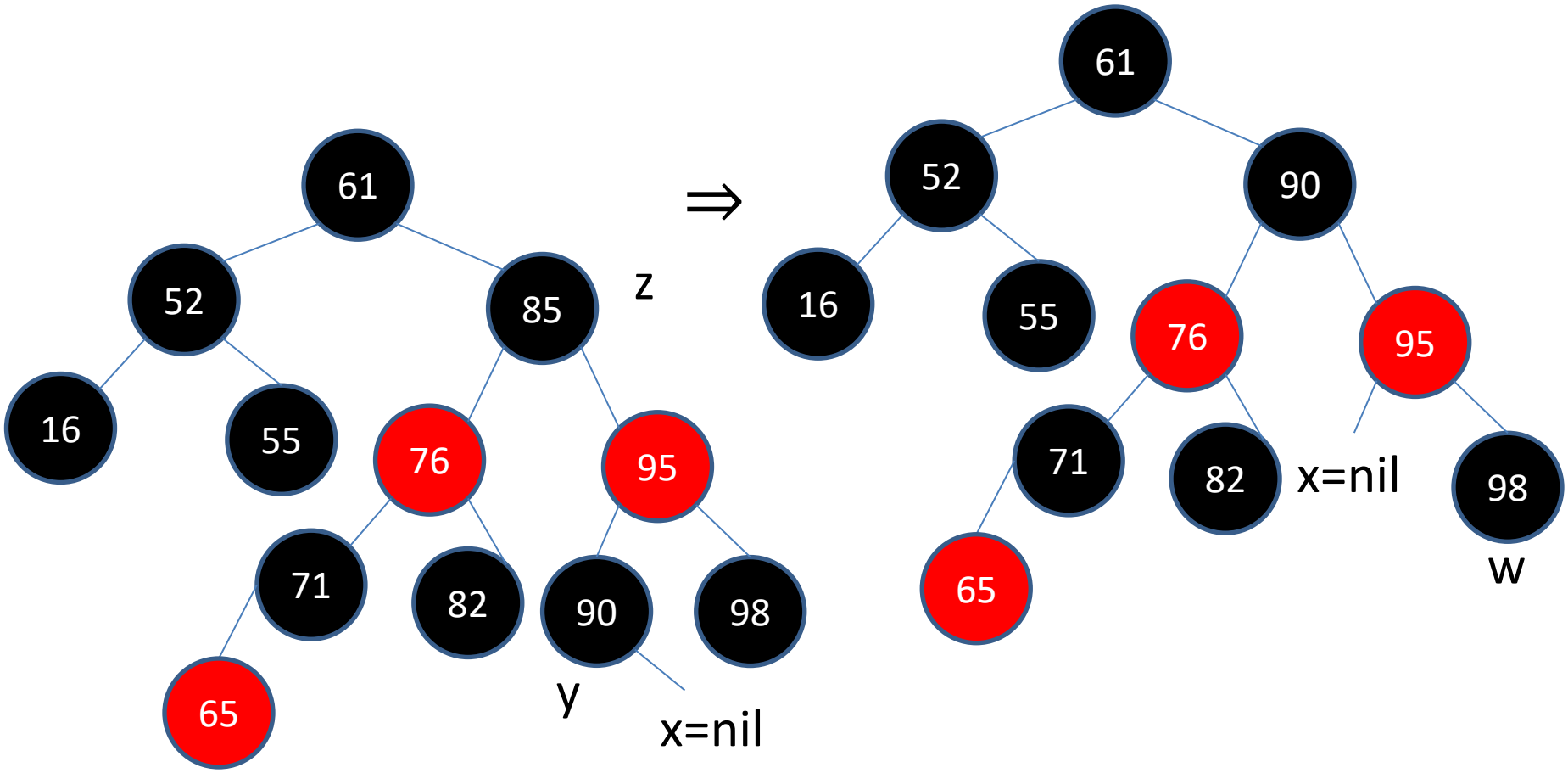
Deletion operation

Deletion of 20



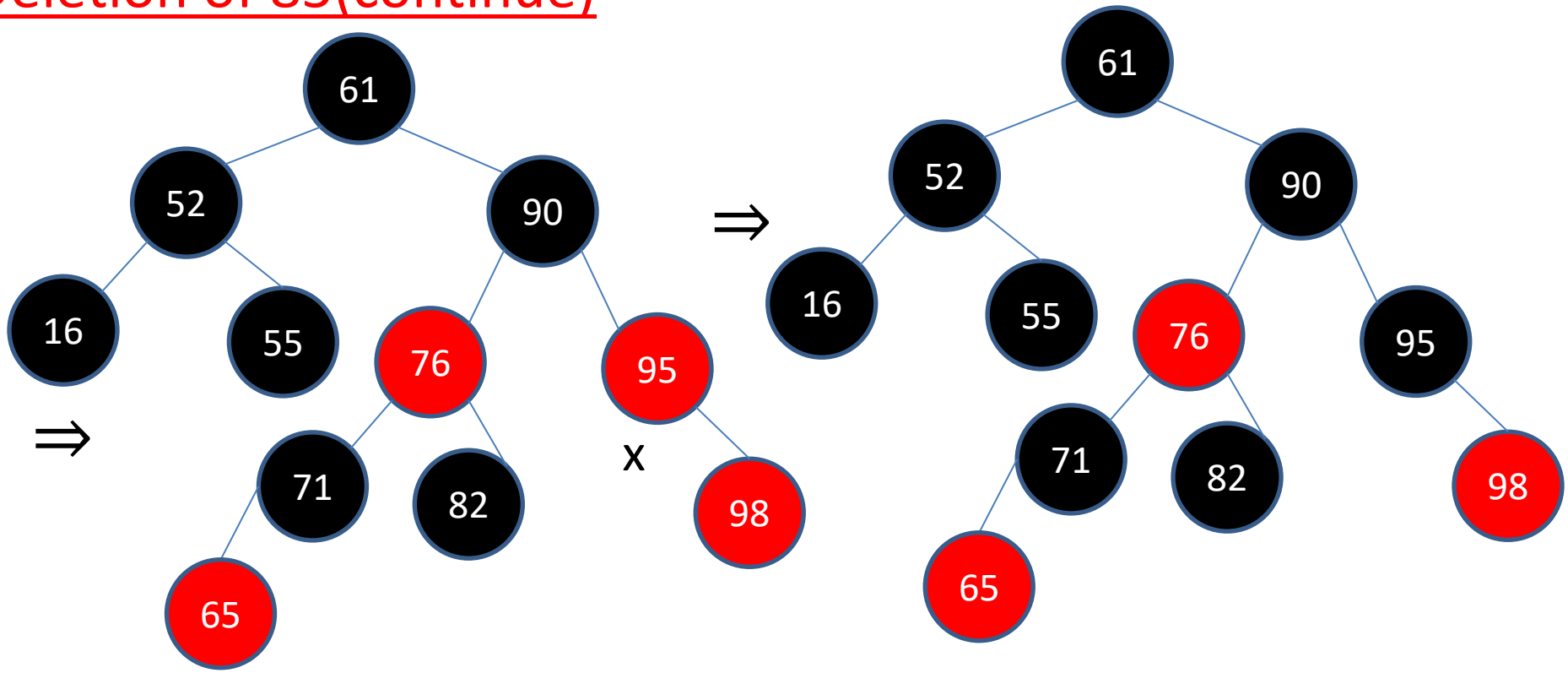
Deletion operation

Deletion of 85



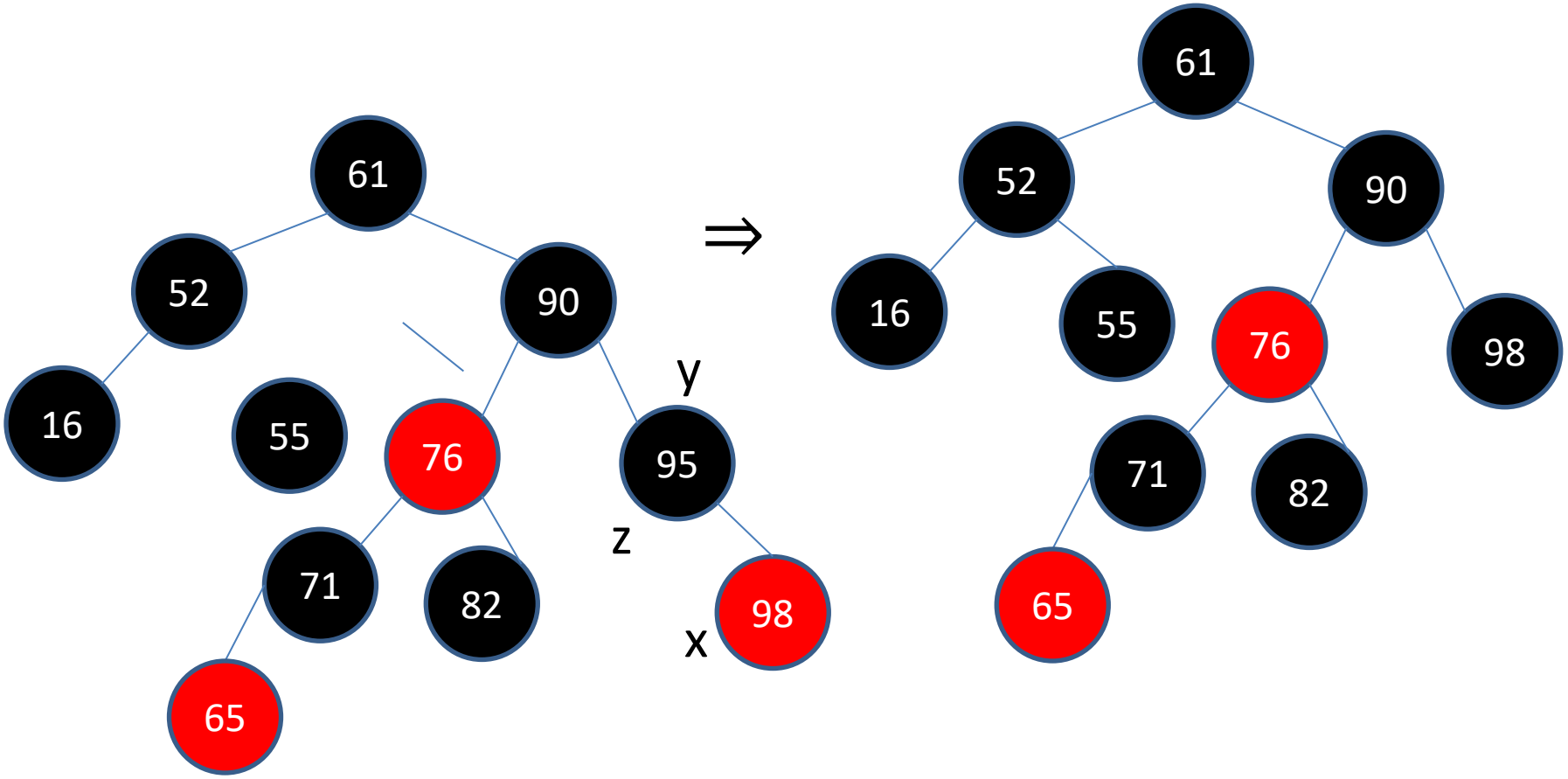
Deletion operation

Deletion of 85(continue)



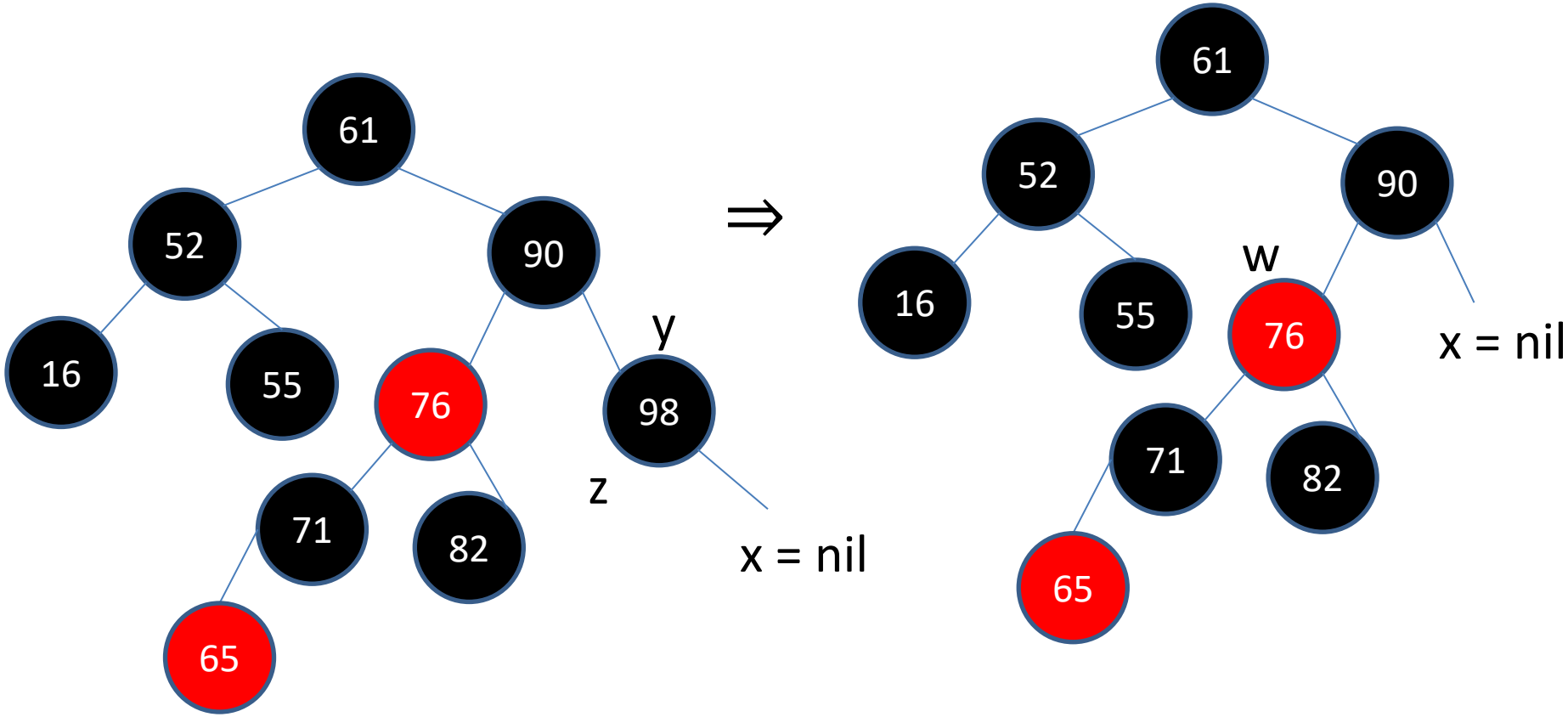
Deletion operation

Deletion of 95



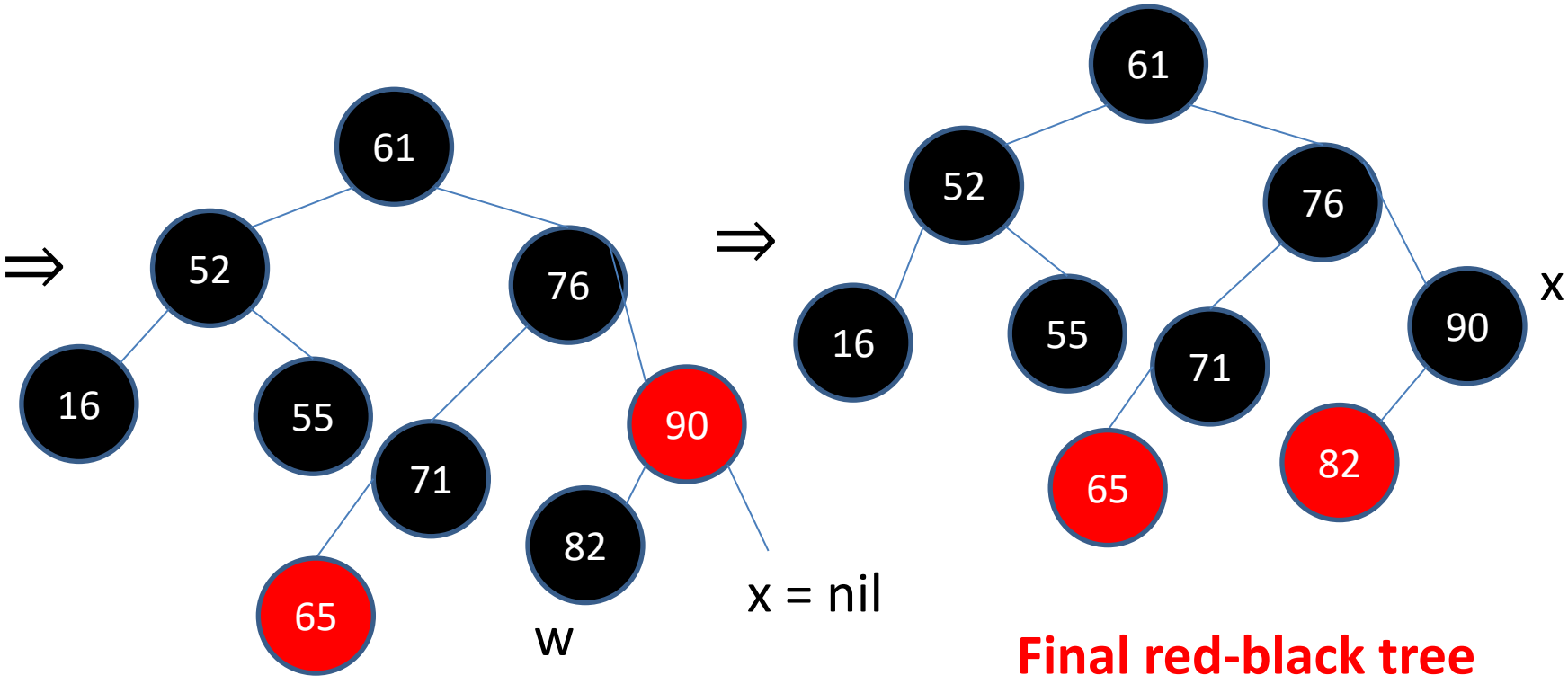
Deletion operation

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 $x.key_1 \leq x.key_2 \leq \ldots\ldots \leq x.key_{x.n}$,

   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 $x.c_1$; $x.c_2$, ……. ,$x.c_{x.n+1}$ 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

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

B-Tree

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

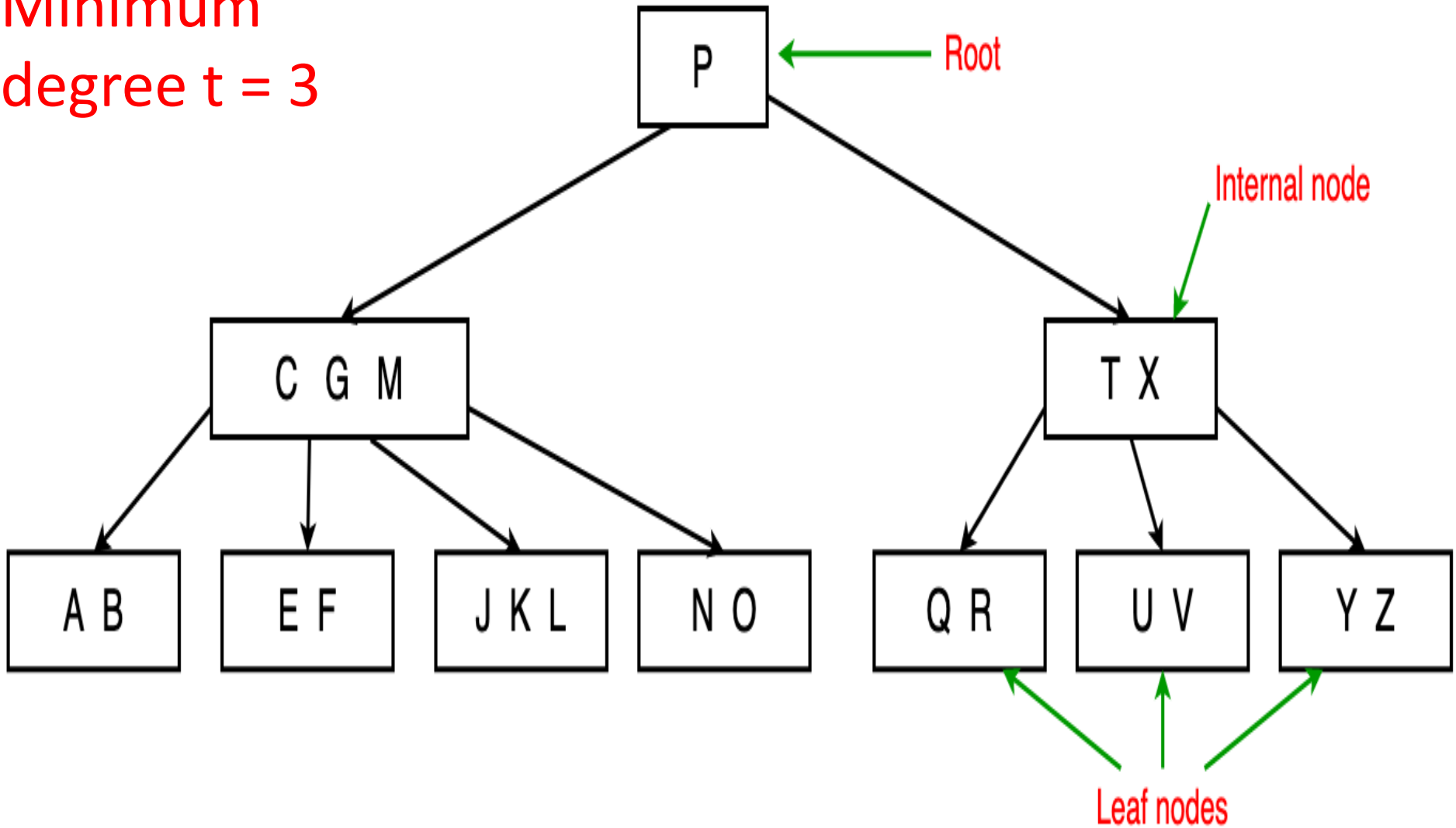
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:

   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



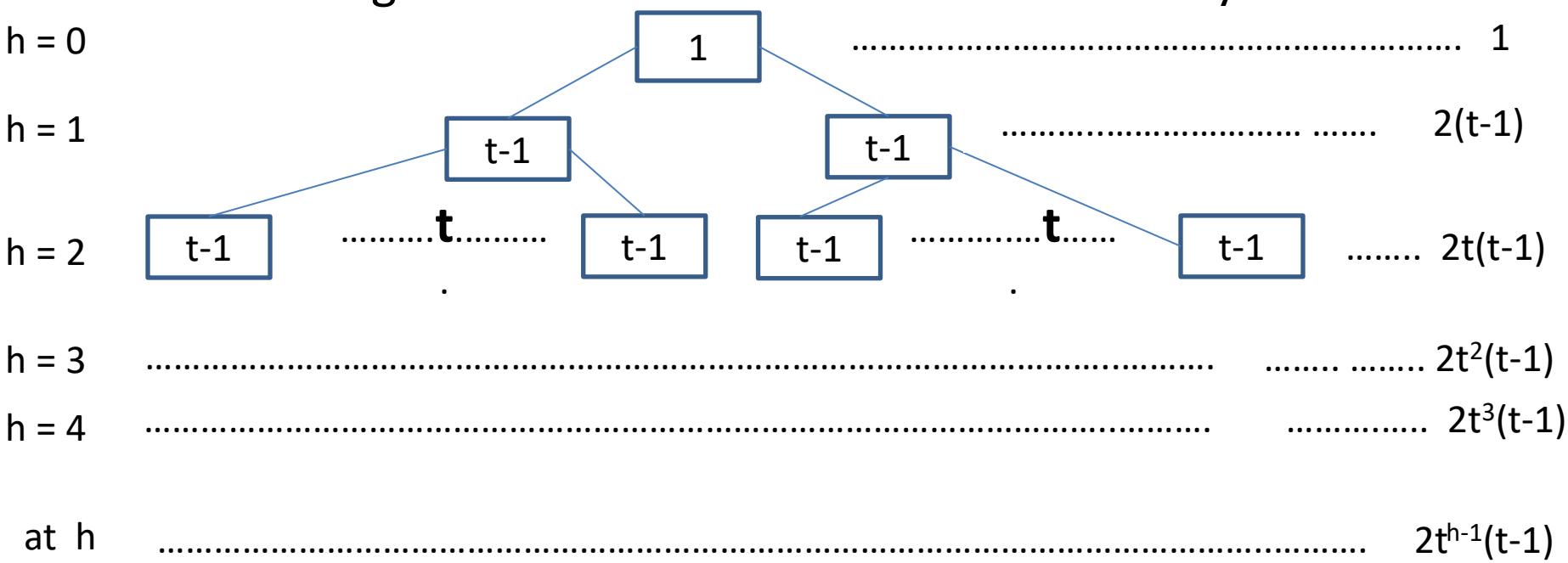
B-Tree

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

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

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

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

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

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

$$= 2t^h-1$$

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

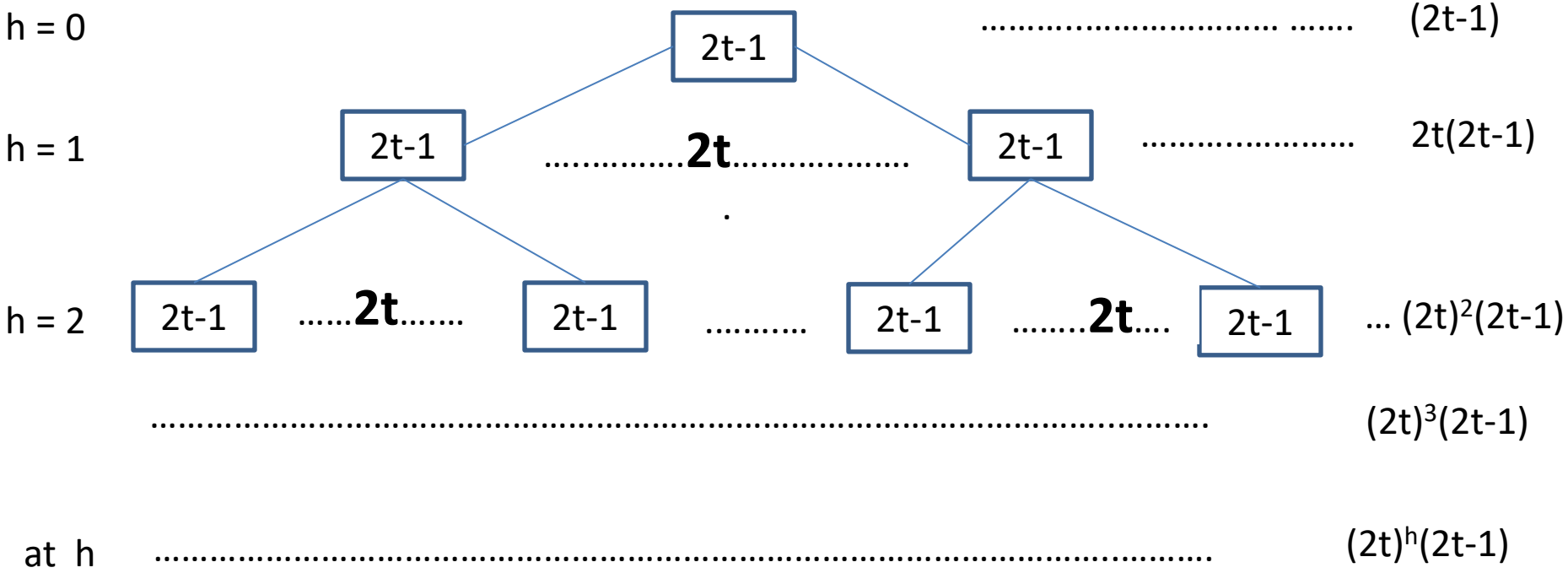
$$2t^h-1 \le n \;\Rightarrow\; t^h \le \frac{(n+1)}{2} \Rightarrow h \le \log_t \frac{(n+1)}{2}$$

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

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

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

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

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

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)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ 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)
```

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

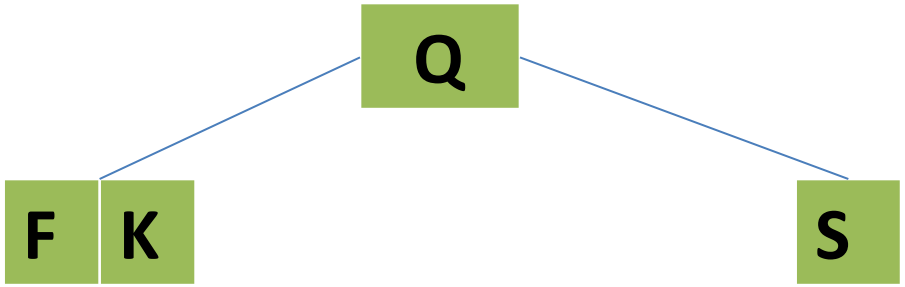
Solution:

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

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

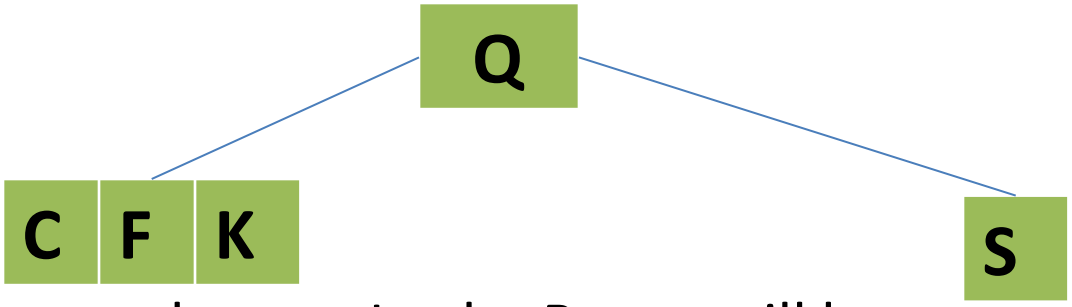
Initial consider 2t-1 elements in the sequence i.e. 3 elements. These are F, S and Q. B-tree for this will be

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

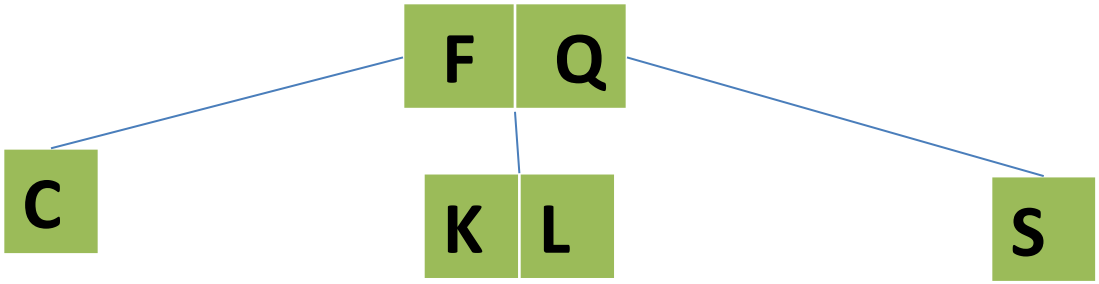


B-Tree

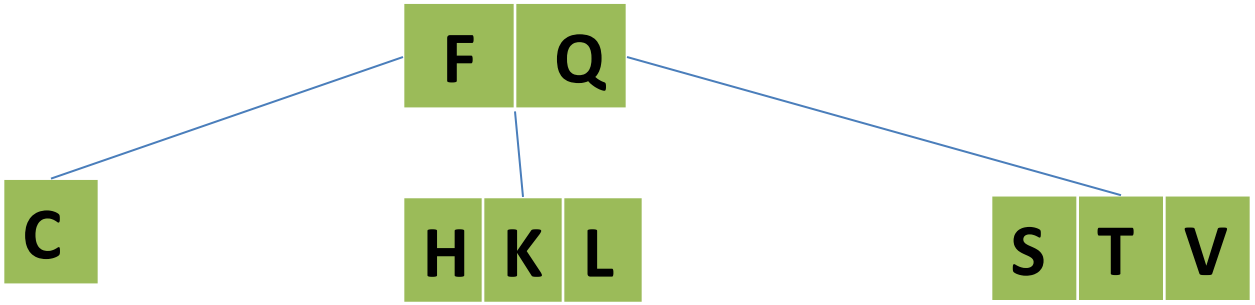
After inserting next element C, the B-tree will be



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

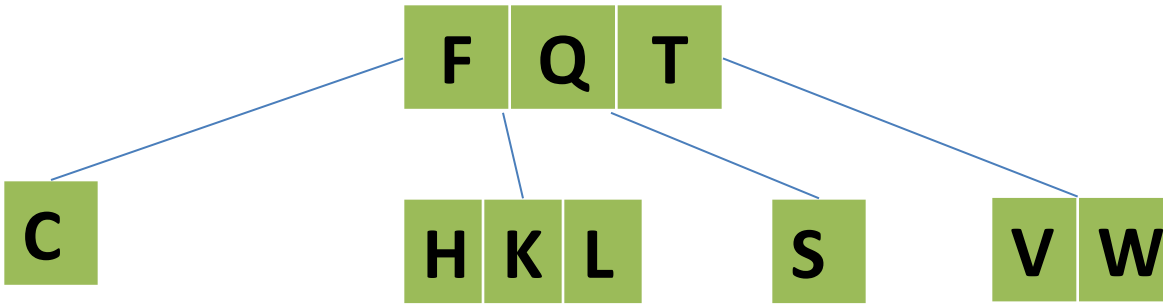


After inserting next element H,T, and V, the B-tree will be

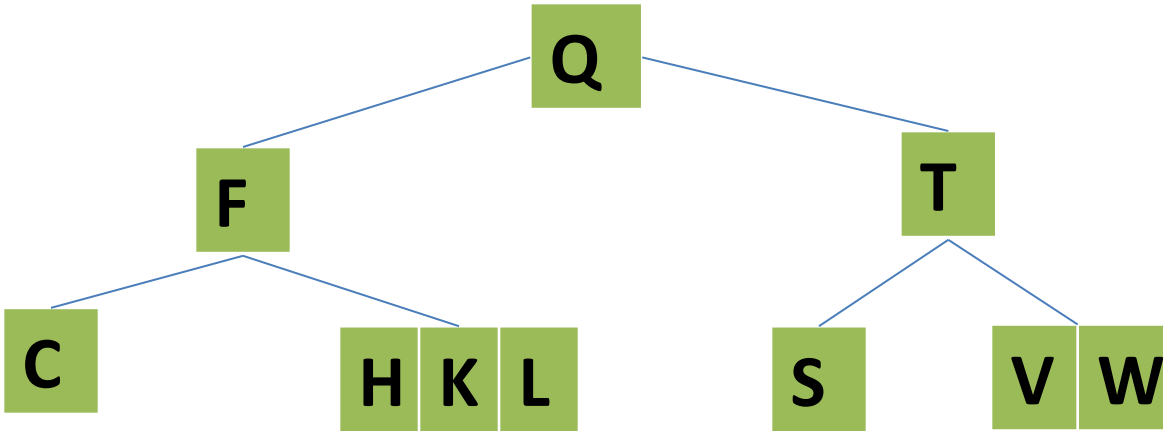


B-Tree

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

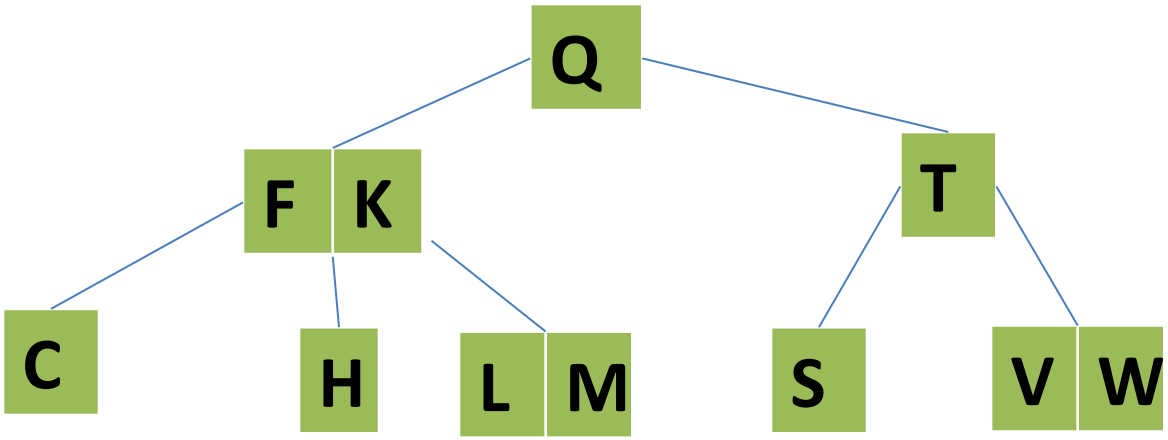


Now, insert element M. Since root node is full, therefore first, we split it.

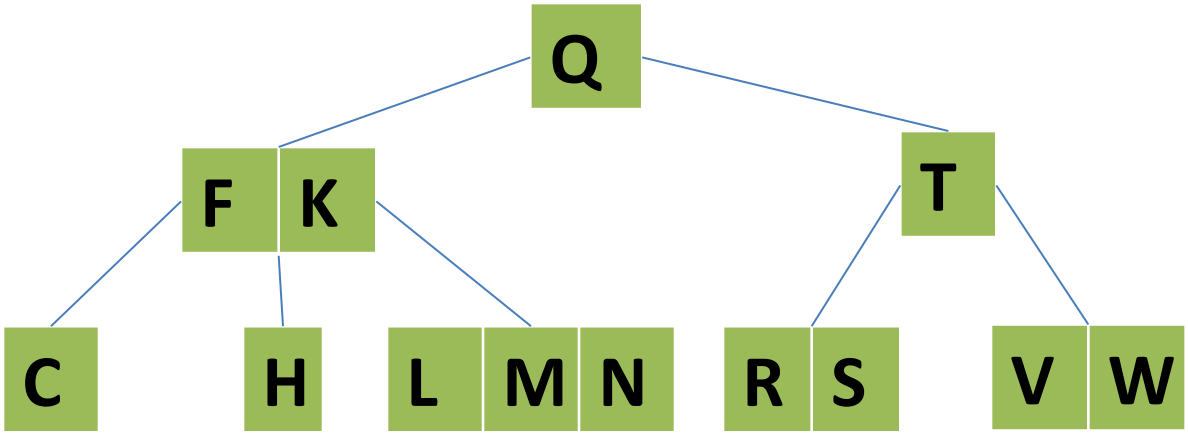


B-Tree

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

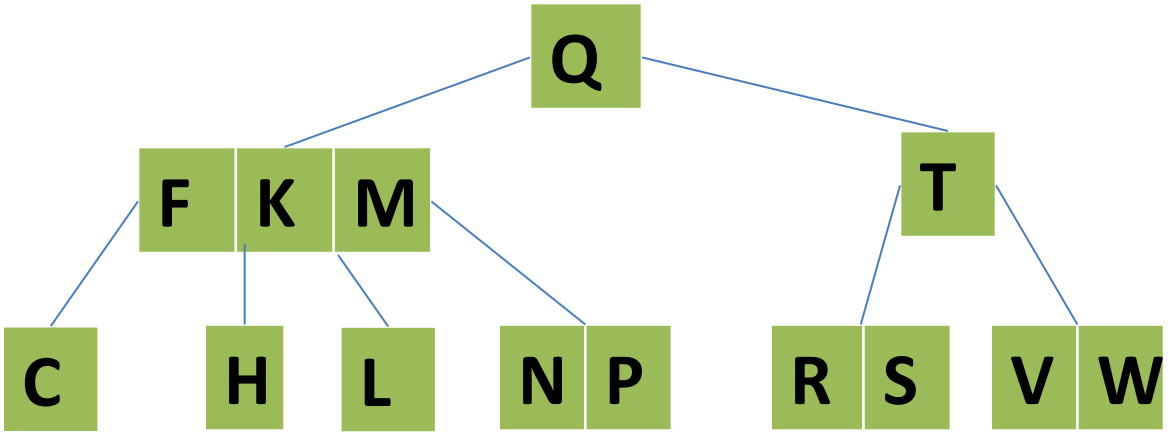


After inserting next element R and N, the B-tree will be

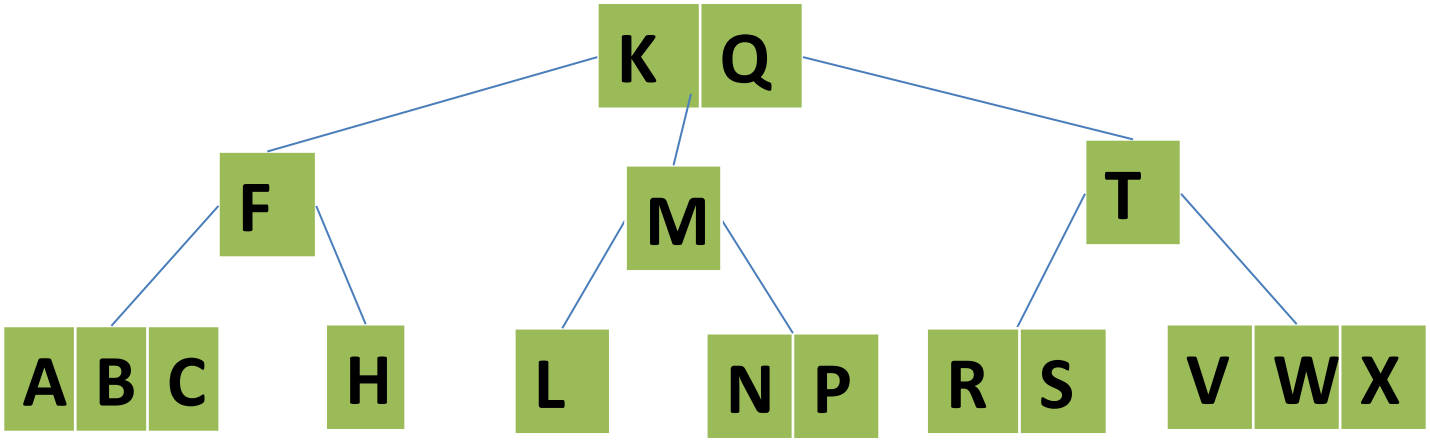


B-Tree

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

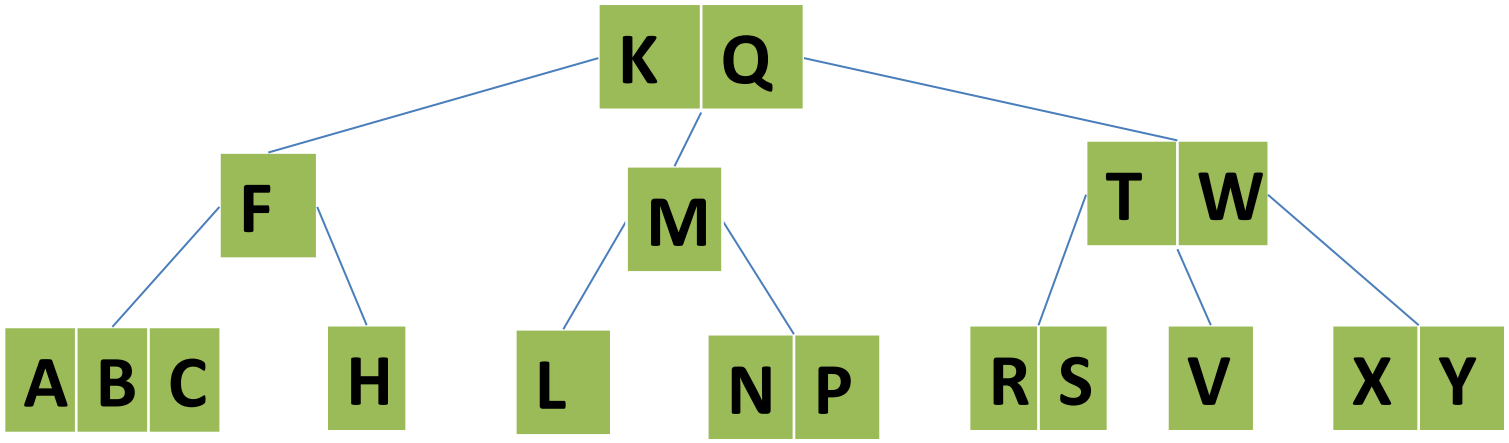


After splitting and inserting next element A, B and X, the B-tree will be

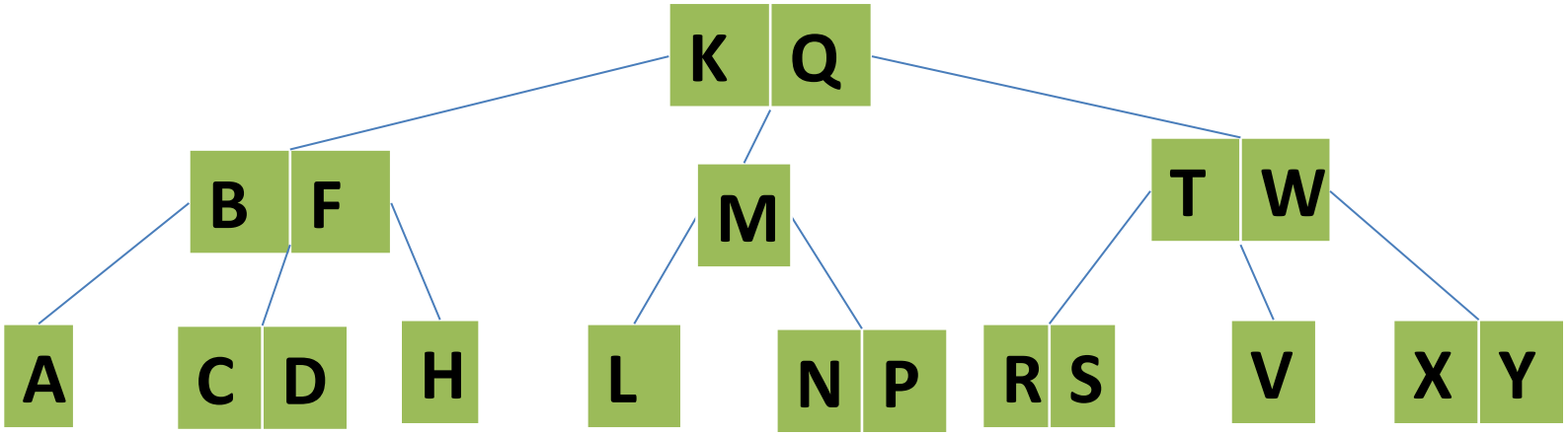


B-Tree

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

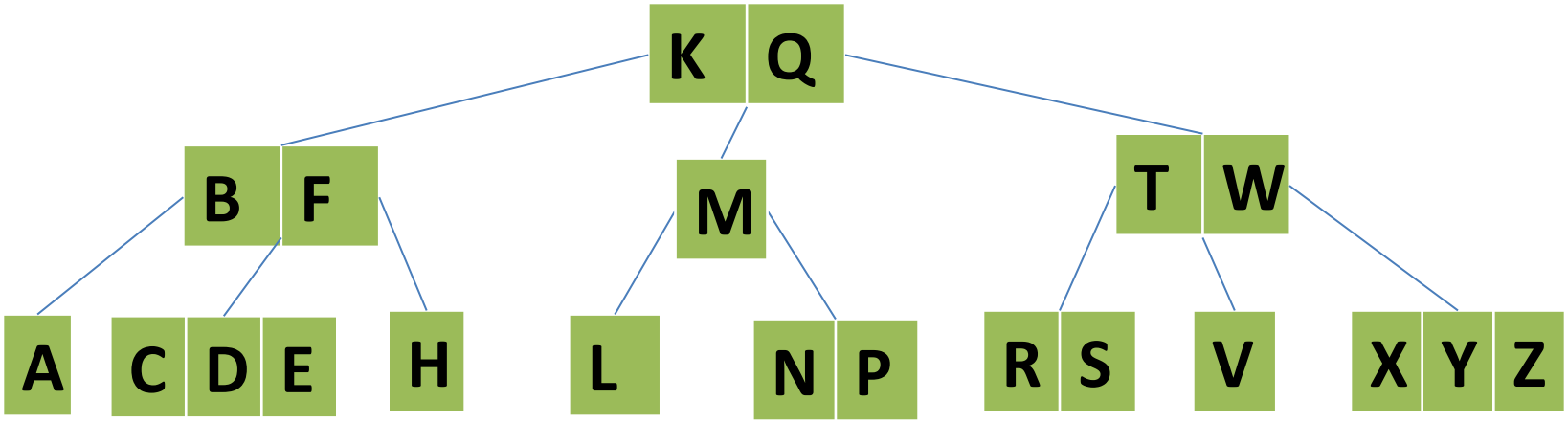


After splitting and inserting next element D, the B-tree will be



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

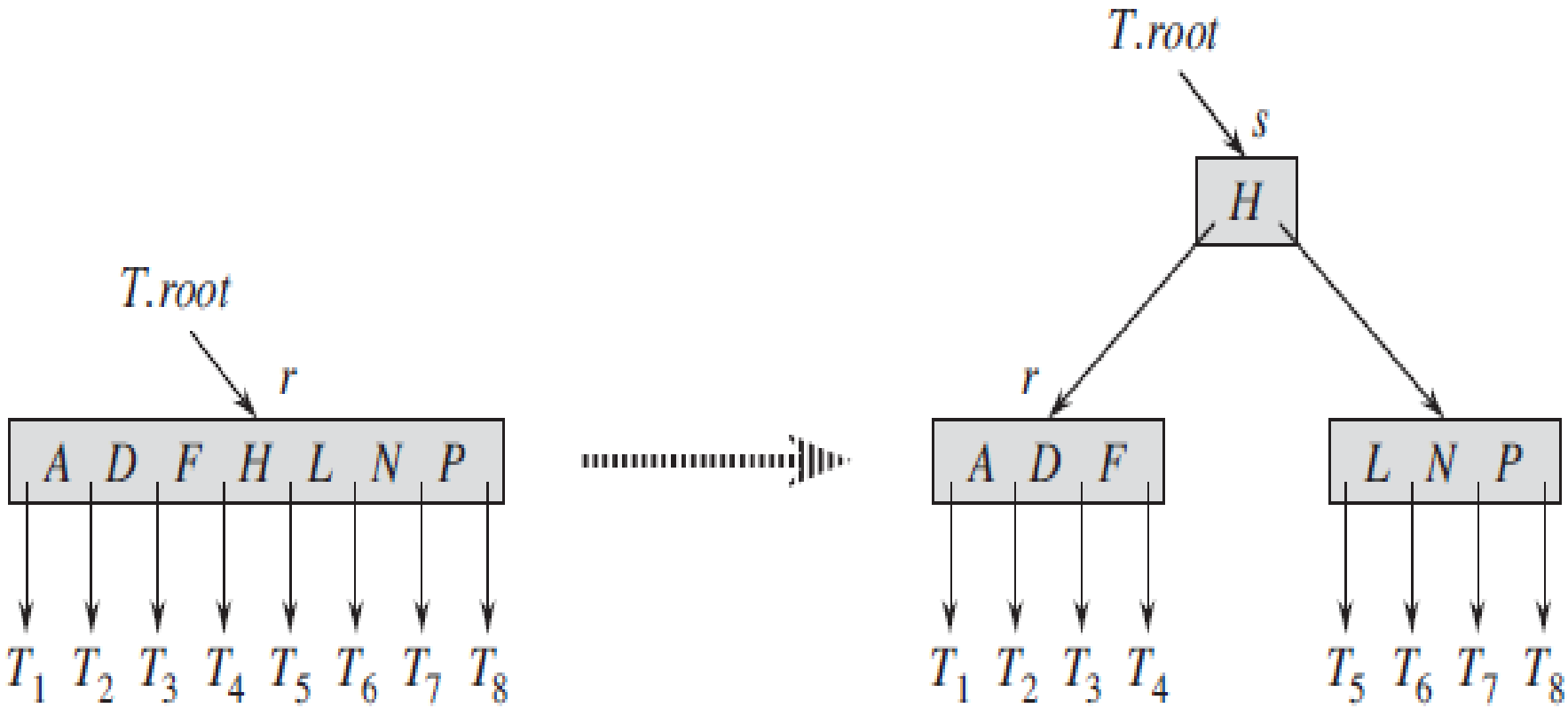
in an initially empty B-Tree. Give the number of nodes splitting operations that take place.

B-Tree Insertion Algorithm

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1
    s = ALLOCATE-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NONFULL(s, k)
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.

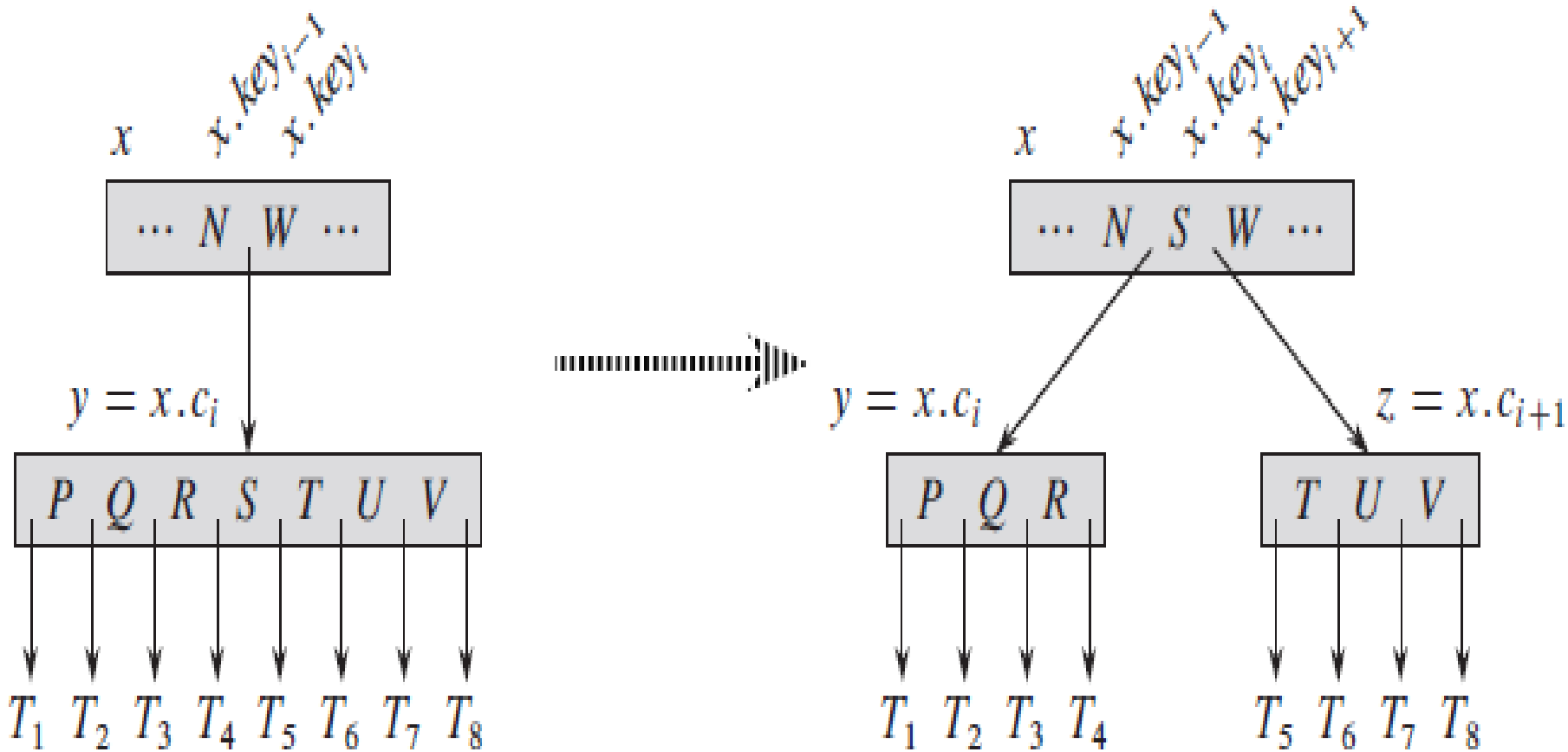


B-Tree Insertion Algorithm

```
B-TREE-SPLIT-CHILD(x, i)
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)
```

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)
i = x.n
if x.leaf
    while i ≥ 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 ≥ 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)
```

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

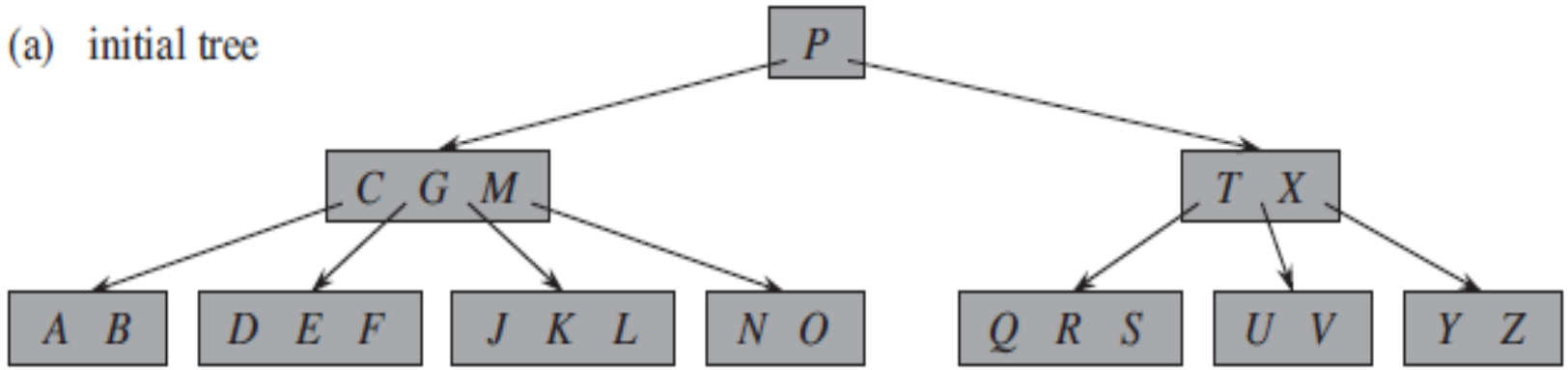
3. If the key k is not present in internal node x, then determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish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



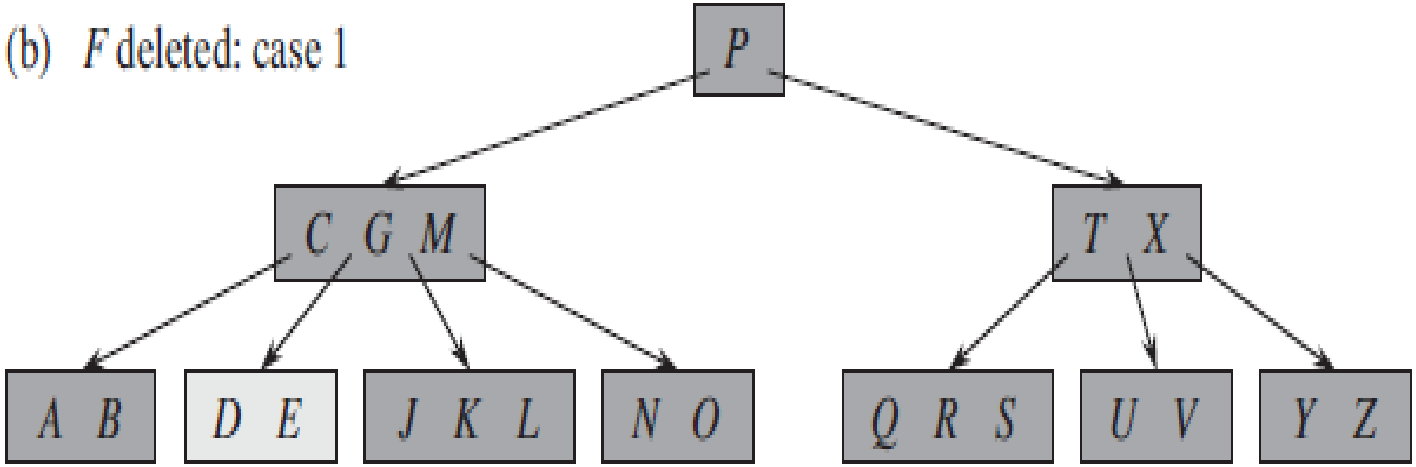
Delete the following elements from this B-tree in-order.

F, M, G, D and B.

Deletion operation

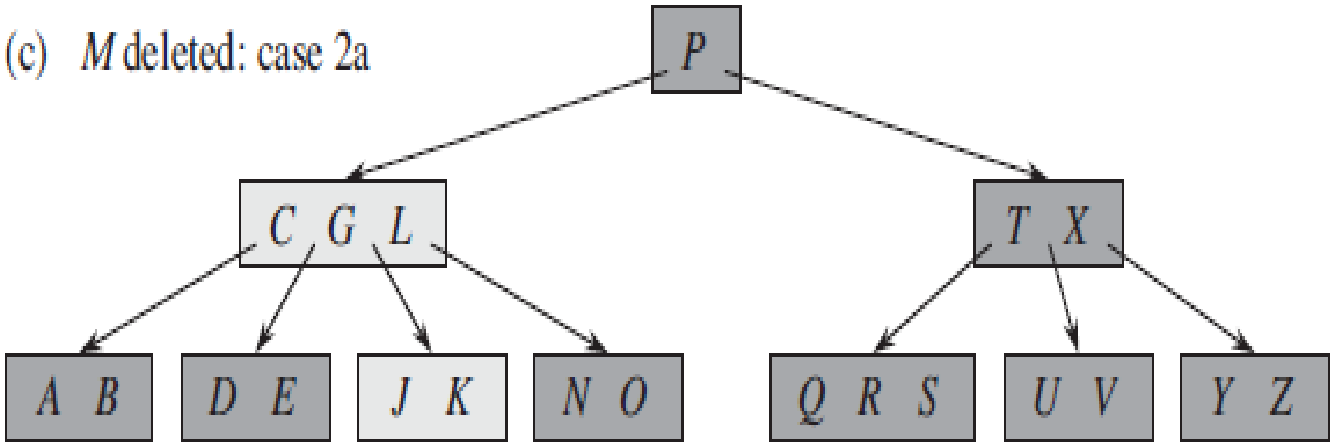
Solution:

Deletion of F



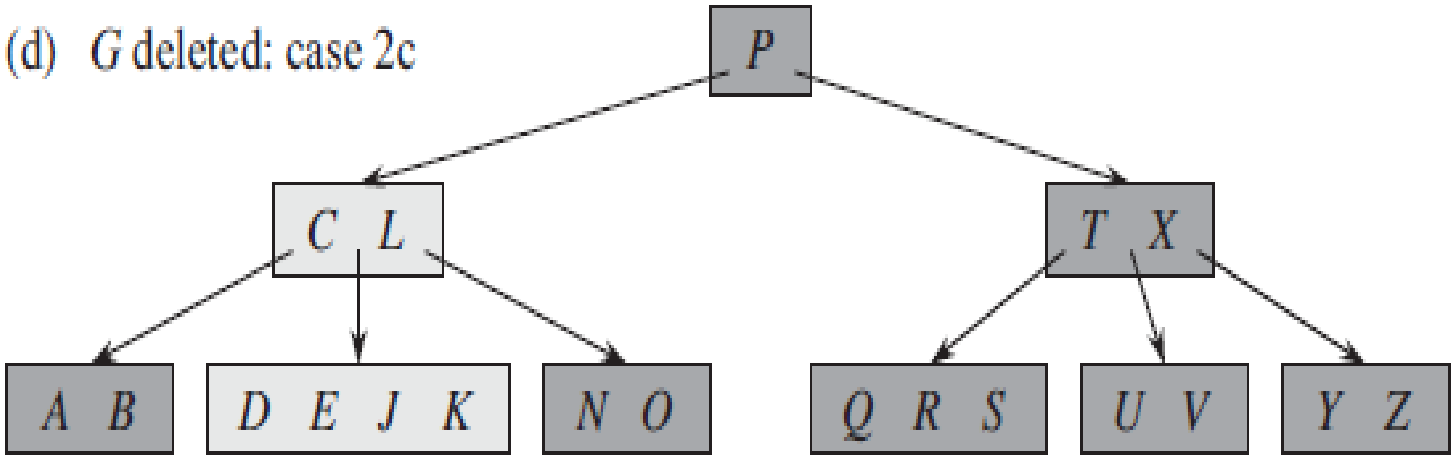
Deletion operation

Deletion of M



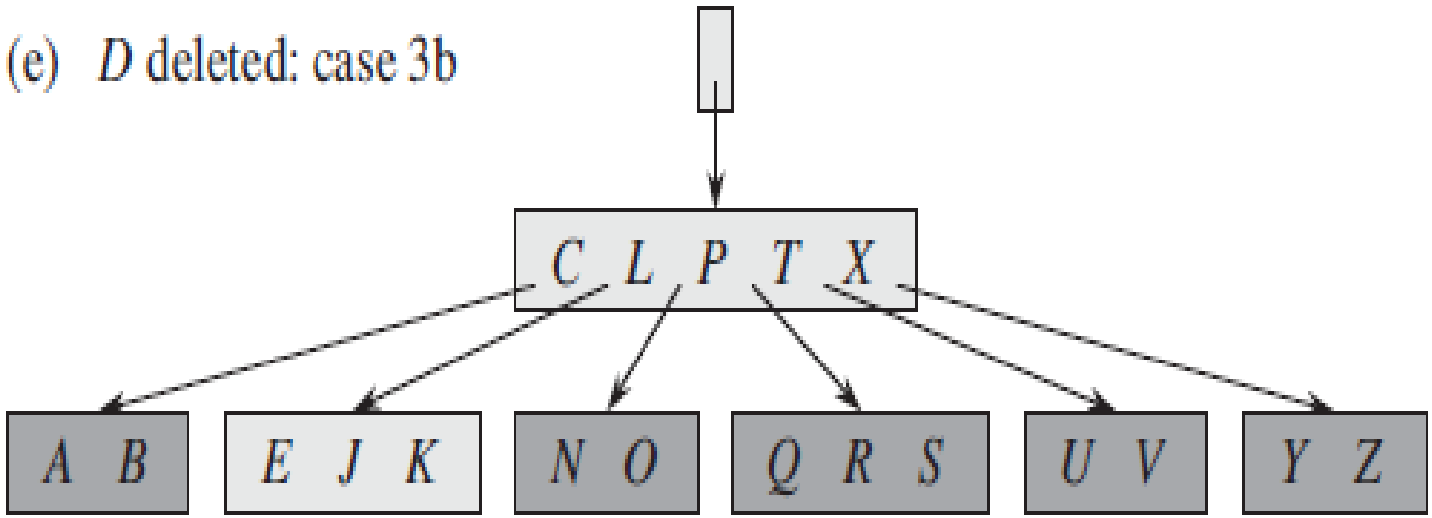
Deletion operation

Deletion of G



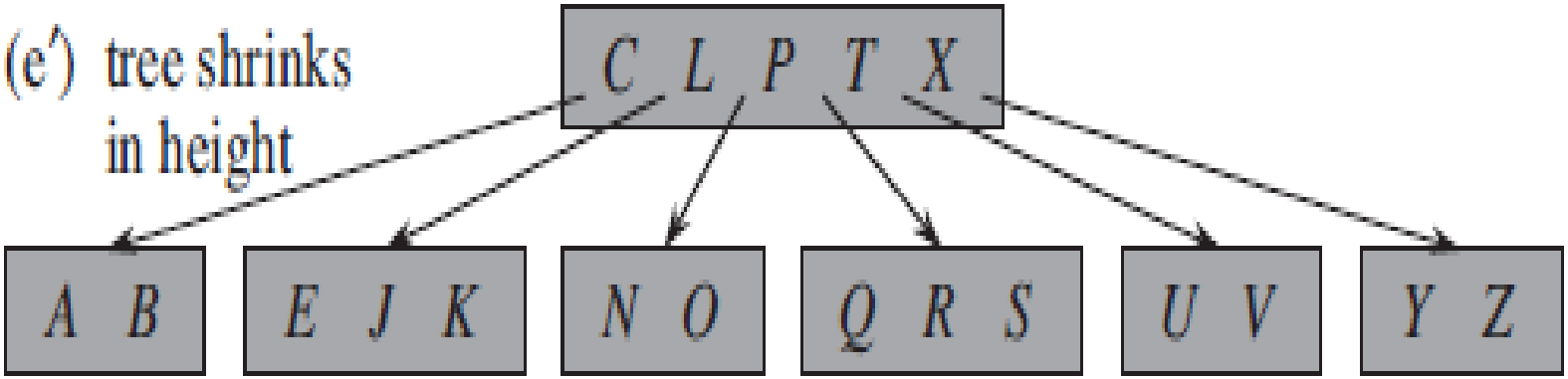
Deletion operation

Deletion of D



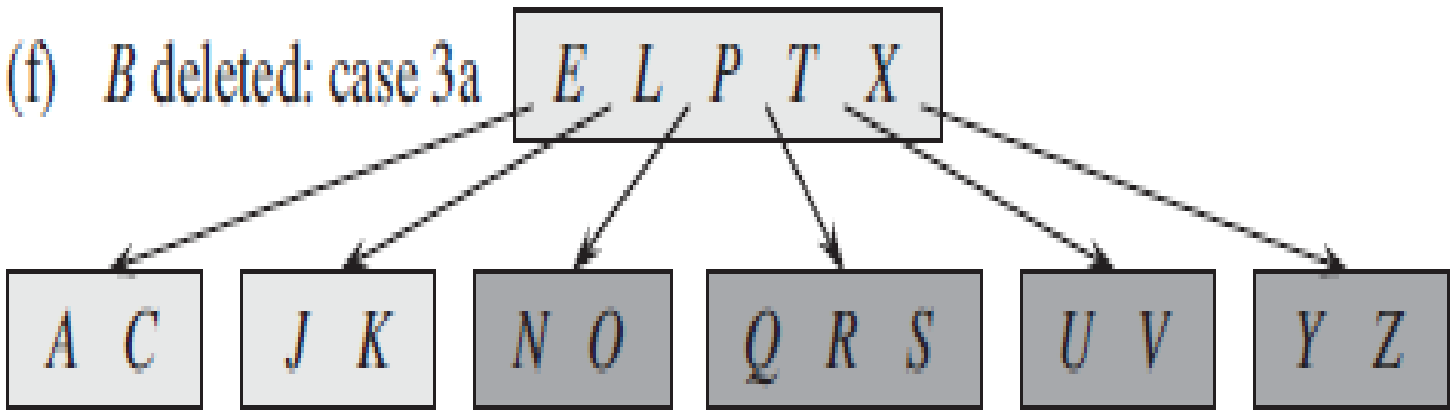
Deletion operation

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:

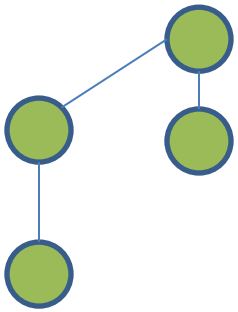
Binomial Tree

Example: Some binomial trees are the following:-

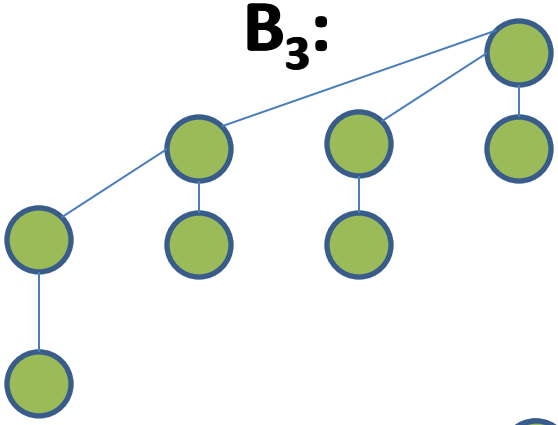
B_0:

B_1:

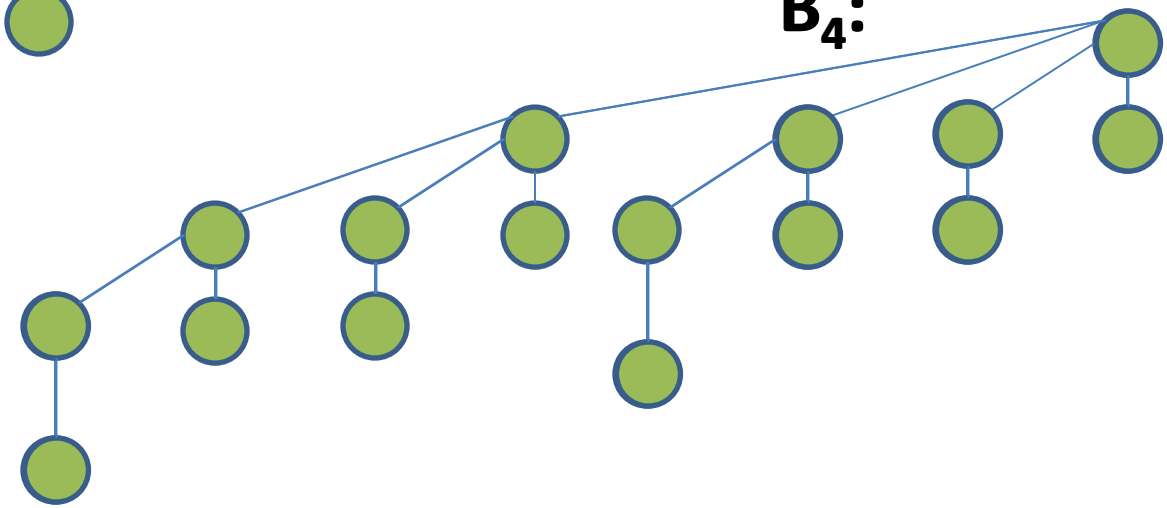
B_2:



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,

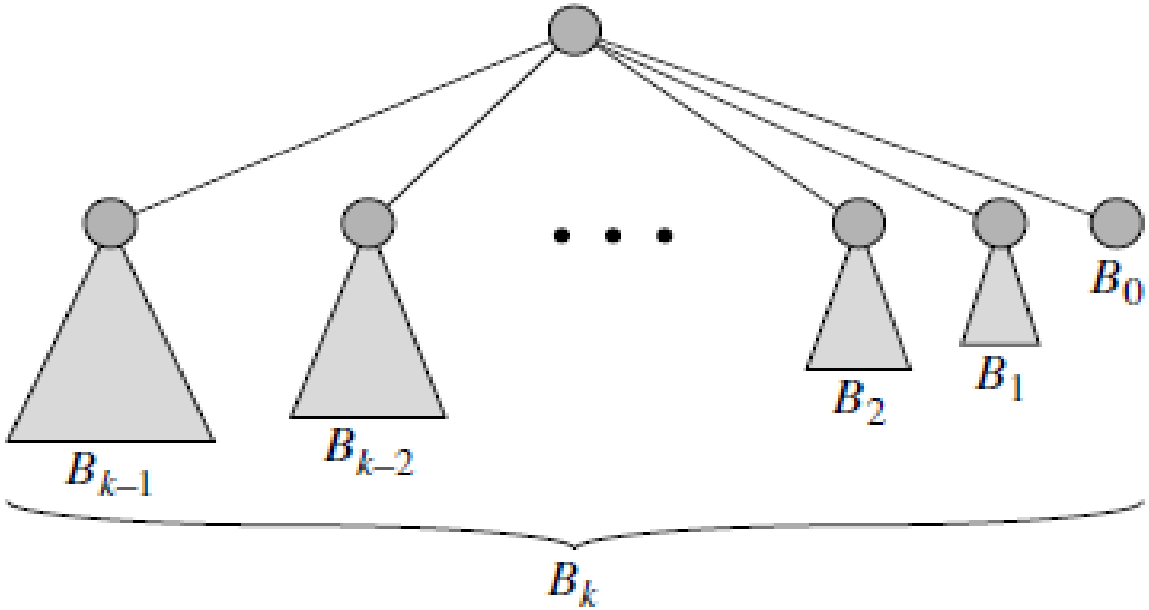
D(k,i) = D(k-1,i) + D(k-1, i-1)

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

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

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.

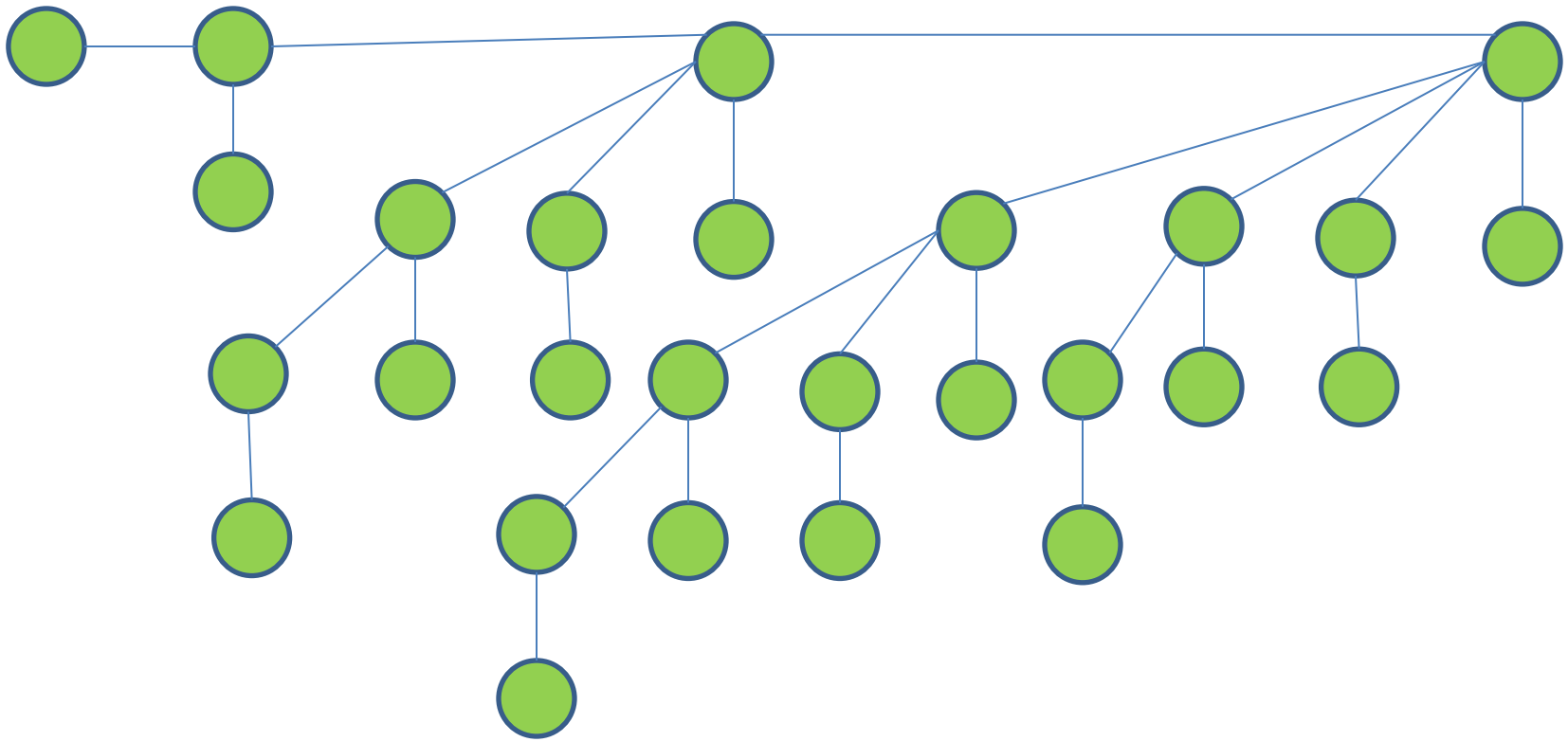
Solution: First we find binary number of 27. After it, we compare this number with $B_4B_3B_2B_1B_0$. If the corresponding binary number is 1, then we use the corresponding binomial tree in the Binomial heap.

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*

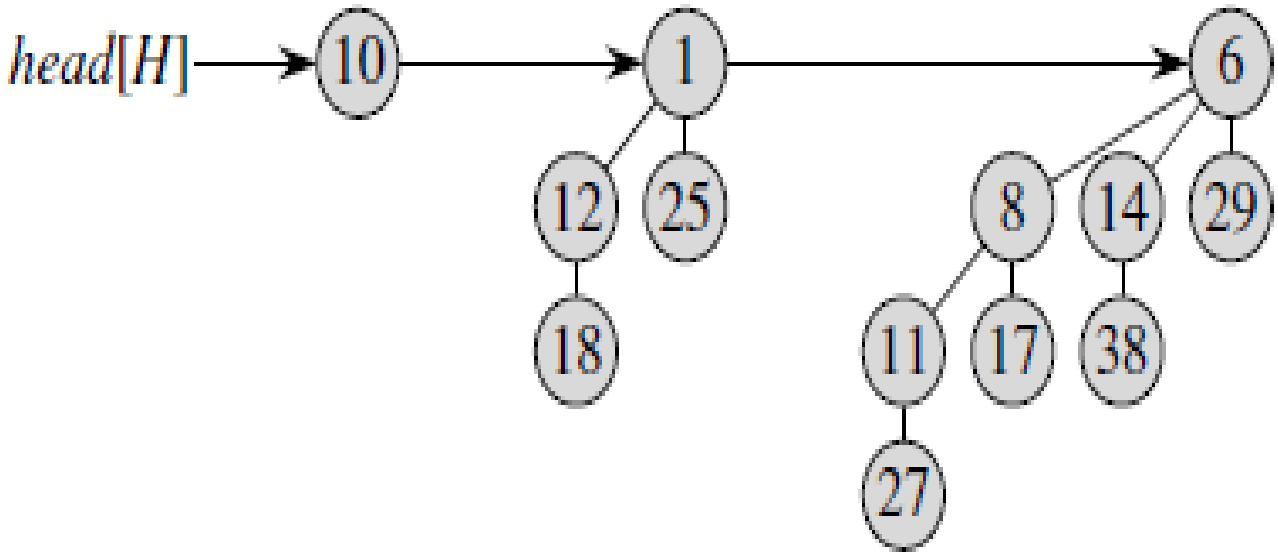
<table>
<tr><td colspan="2">p[x]</td></tr>
<tr><td colspan="2">Key[x]</td></tr>
<tr><td colspan="2">Degree[x]</td></tr>
<tr><td>Child[x]</td><td>Sibling[x]</td></tr>
</table>

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

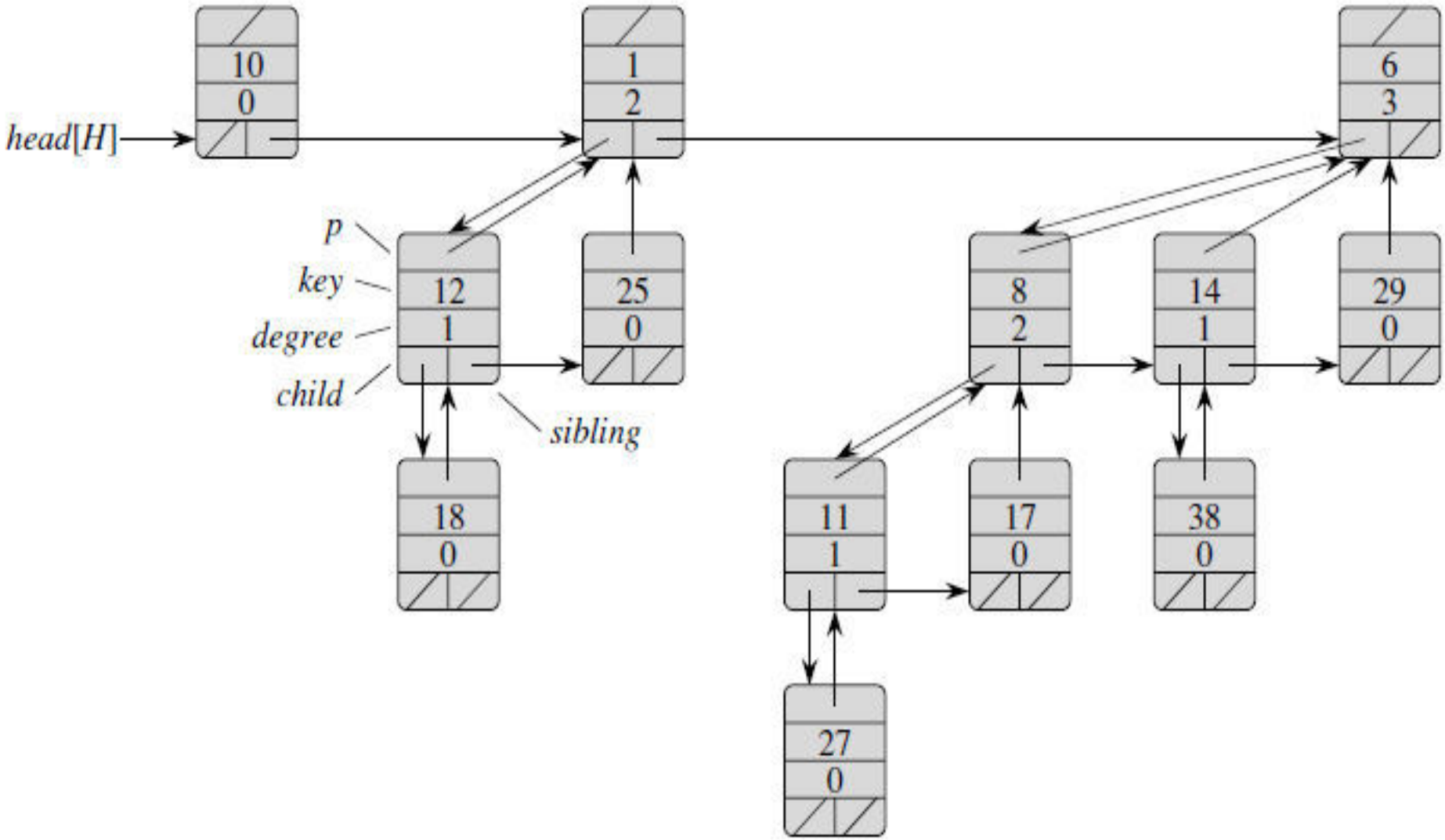
Example: Consider the following binomial heap:-



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

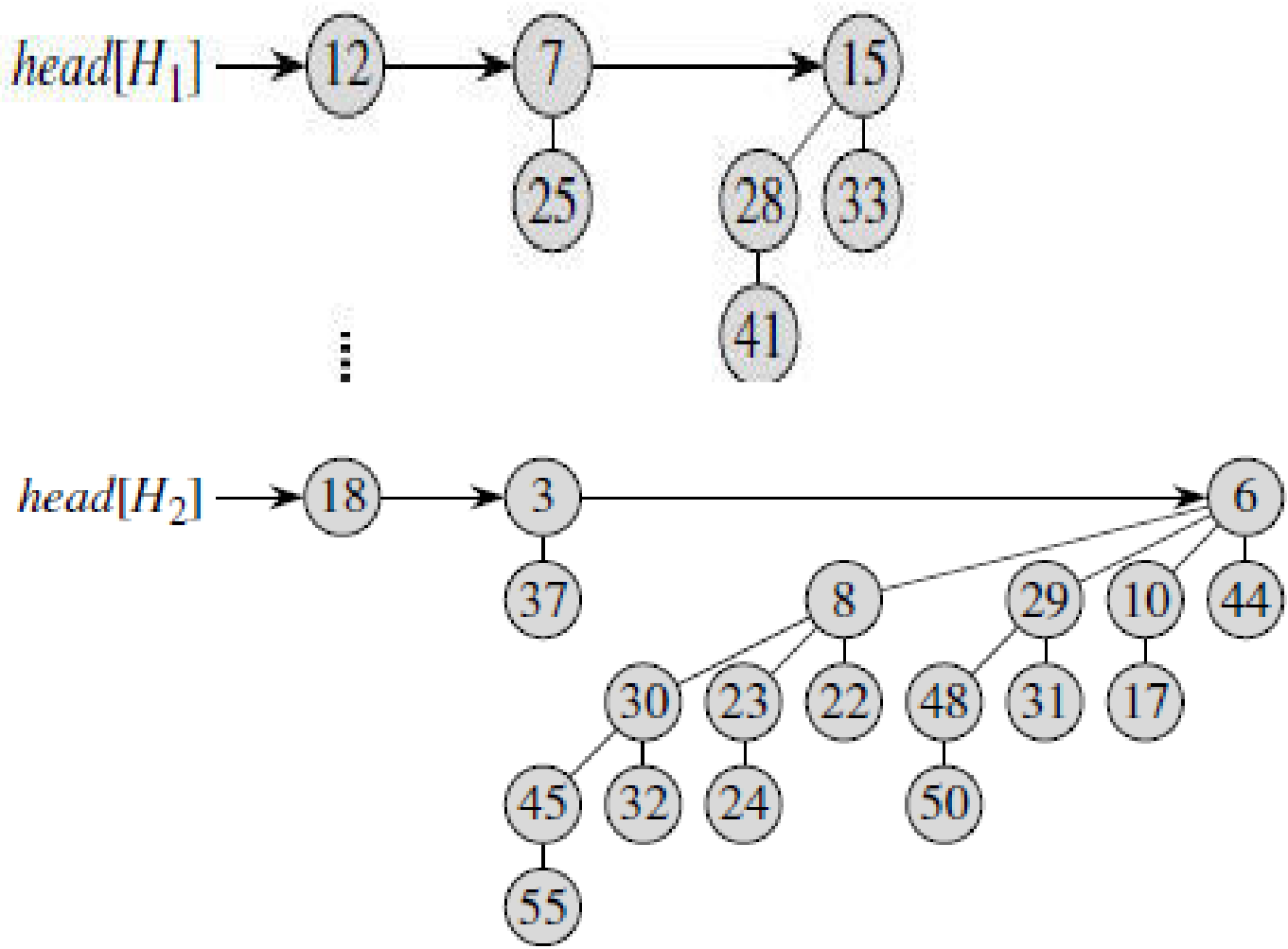
The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H.

```
BINOMIAL-HEAP-MINIMUM(H)
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
      do if key[x] < min
            then min ← key[x]
                 y ← x
         x ← sibling[x]
return y
```

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

Union of two binomial heaps

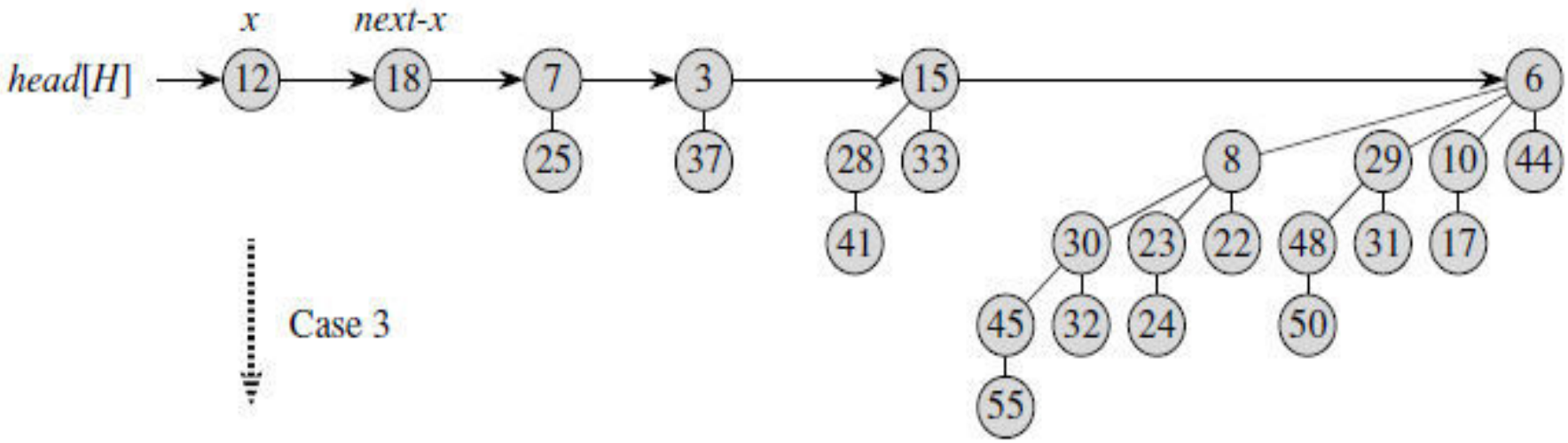
Example: Consider following two binomial heaps H_1 and H_2. Find the union of these binomial heaps.



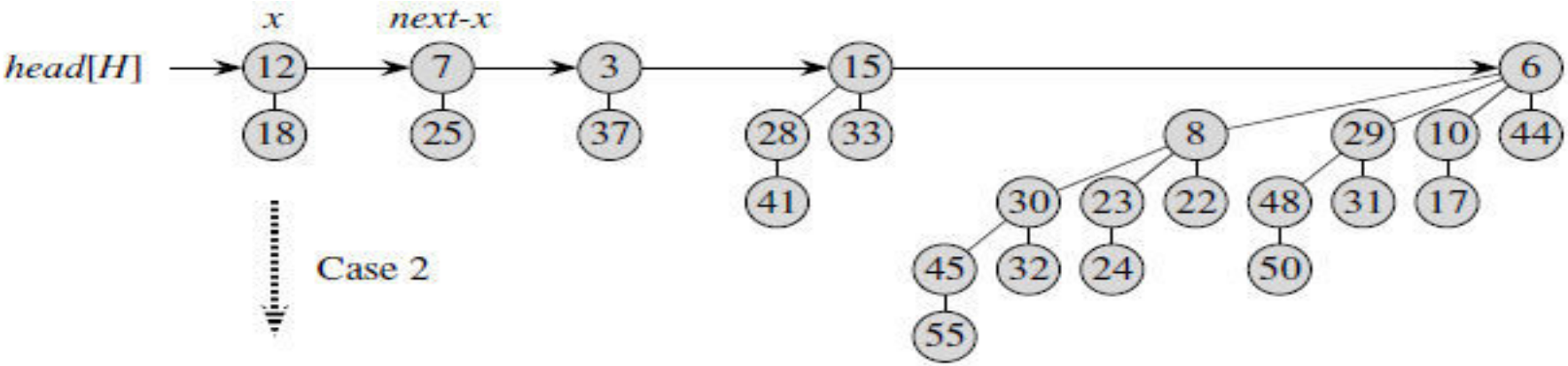
Union of two binomial heaps

Solution:

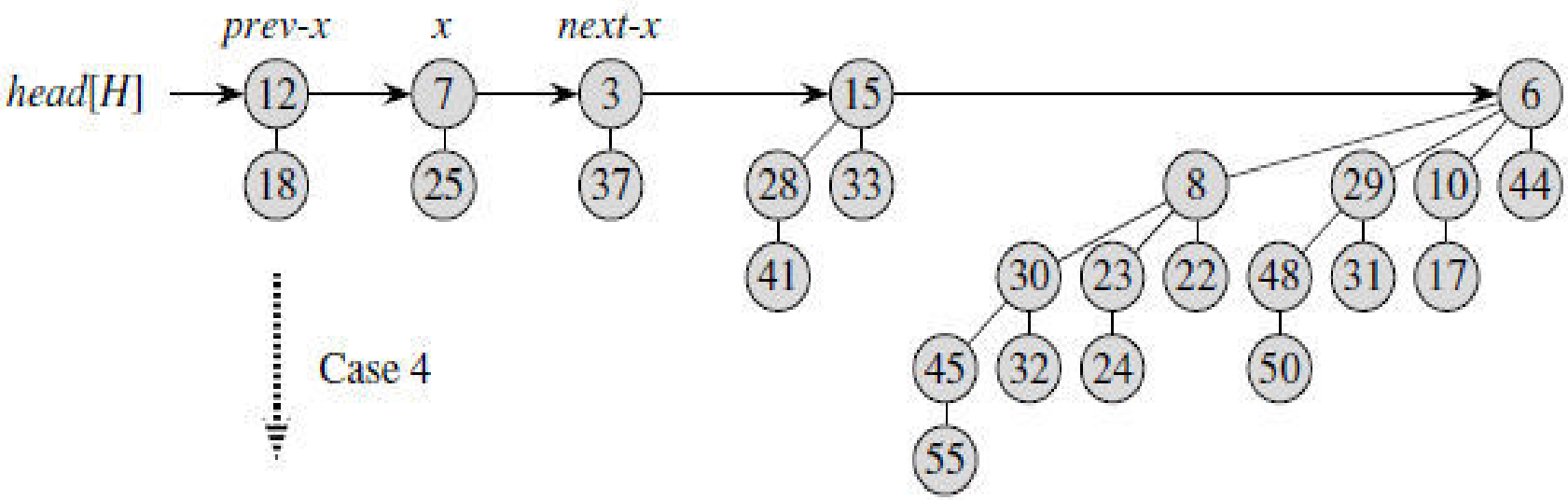
Step-1: Merge both binomial heaps.

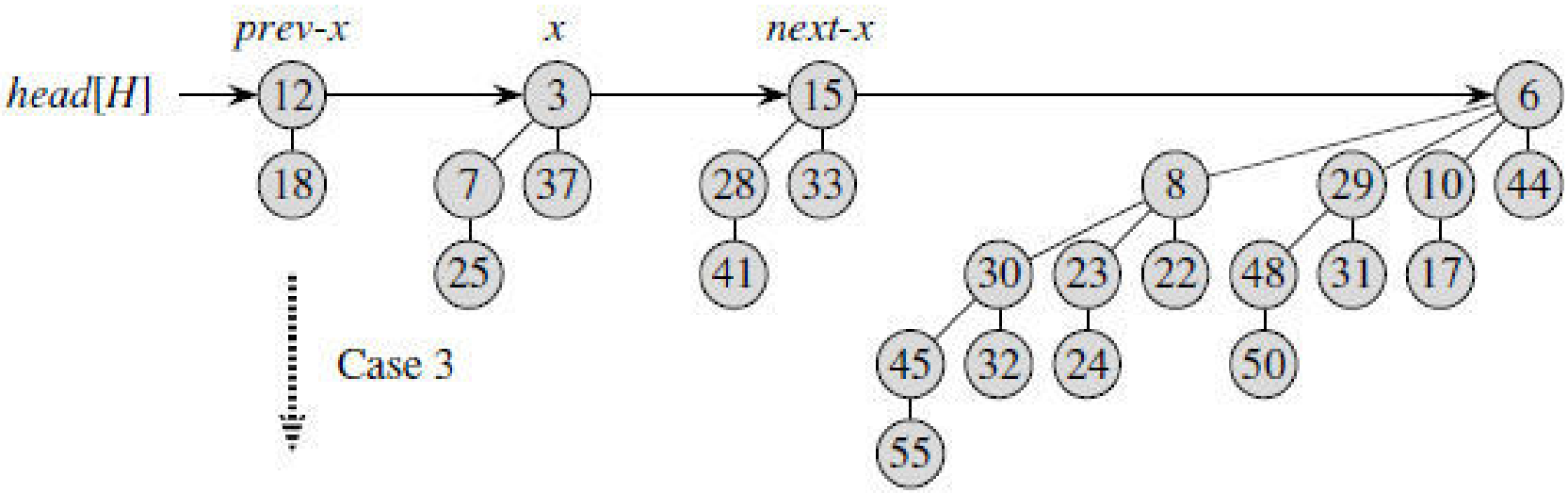


Step-2: Apply the linking process of equal degree root nodes.

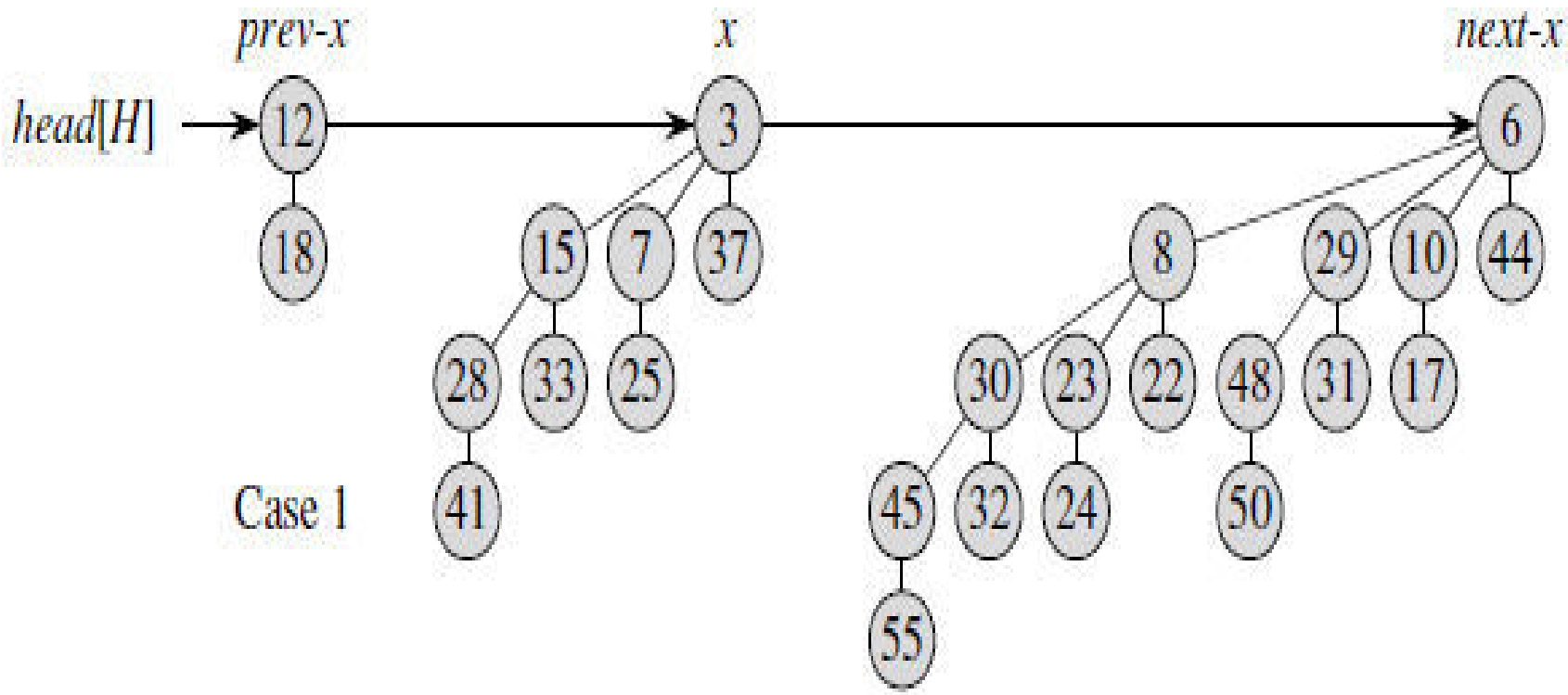


Union of two binomial heaps





Union of two binomial heaps



Final binomial heap

Union of two binomial heaps

```
BINOMIAL-HEAP-UNION(H1, H2)
H ← MAKE-BINOMIAL-HEAP()
head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
   then return H
prev-x ← NIL
x ← head[H]
next-x ← sibling[x]
while next-x ≠ NIL
      do if (degree[x] ≠ degree[next-x]) or
                (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
            then prev-x ← x                                   ▷ Cases 1 and 2
                 x ← next-x                                   ▷ Cases 1 and 2
            else  if key[x] ≤ key[next-x]
                    then sibling[x] ← sibling[next-x]                ▷ Case 3
                         BINOMIAL-LINK(next-x, x)                    ▷ Case 3
                    else  if prev-x = NIL                            ▷ Case 4
                            then head[H] ← next-x                    ▷ Case 4
                            else  sibling[prev-x] ← next-x           ▷ Case 4
                         BINOMIAL-LINK(x, next-x)                    ▷ Case 4
                         x ← next-x                                  ▷ Case 4
         next-x ← sibling[x]
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)
    p[y] ← z
    sibling[y] ← child[z]
    child[z] ← y
    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.

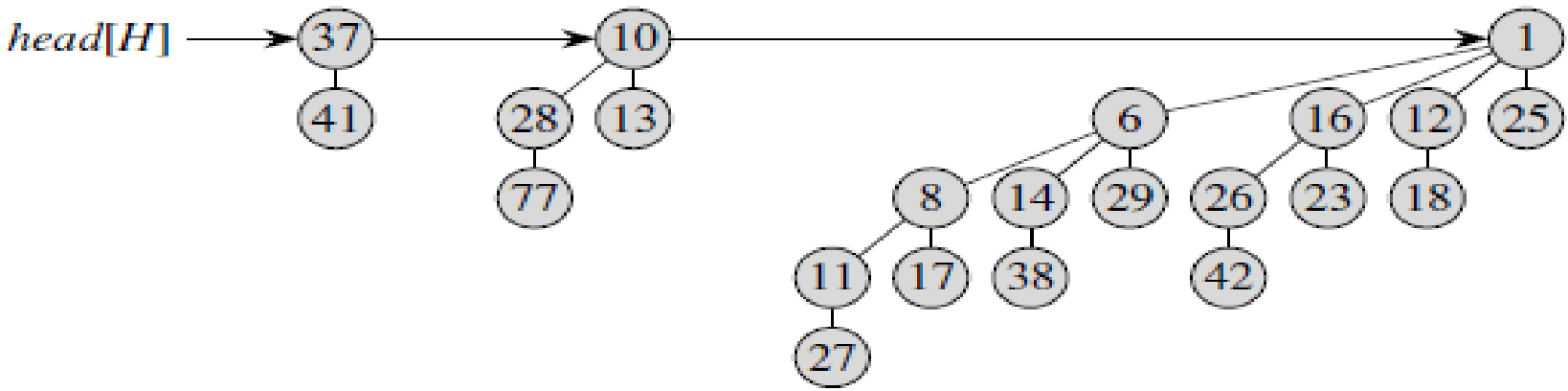
```
BINOMIAL-HEAP-INSERT(H, x)
H' ← MAKE-BINOMIAL-HEAP()
p[x] ← NIL
child[x] ← NIL
sibling[x] ← NIL
degree[x] ← 0
head[H'] ← x
H ← BINOMIAL-HEAP-UNION(H, H')
```

<u>Time Complexity:</u>

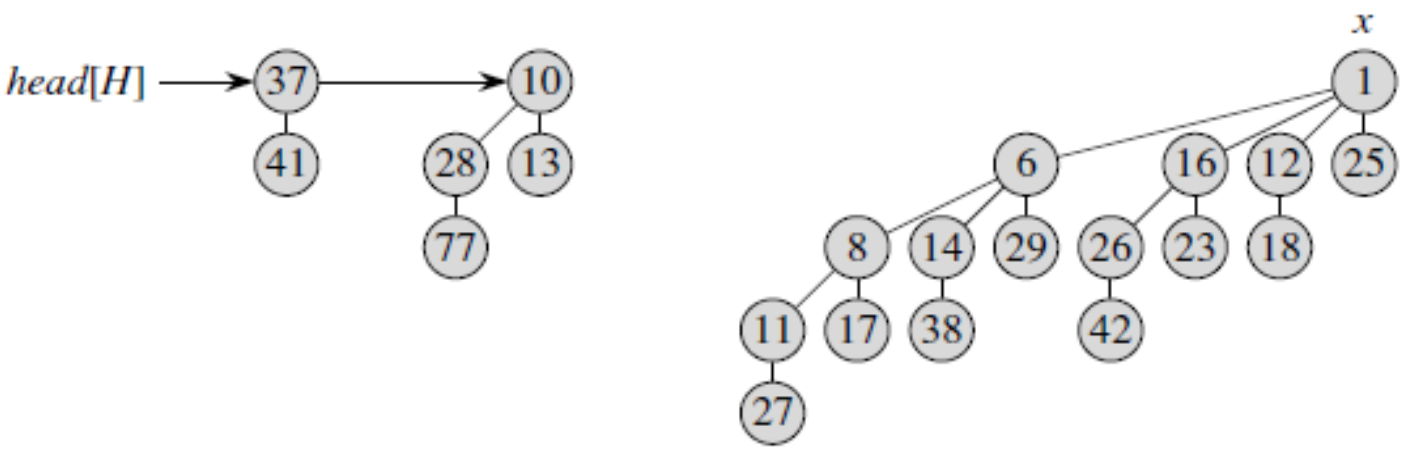
Time complexity of this algorithm is O(lg n).

Extracting the node with minimum key

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

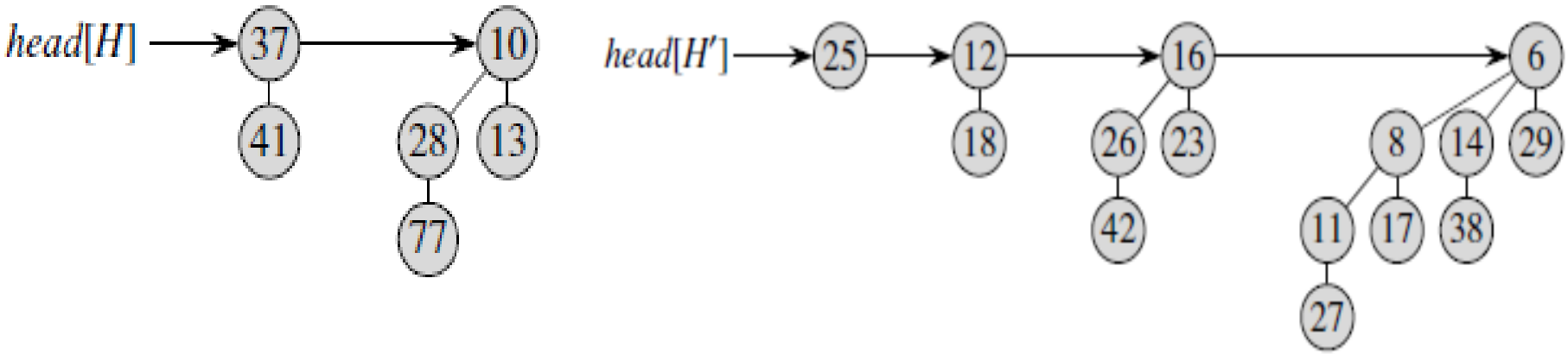


Solution: Step-1: Find the node with minimum key and remove that node.

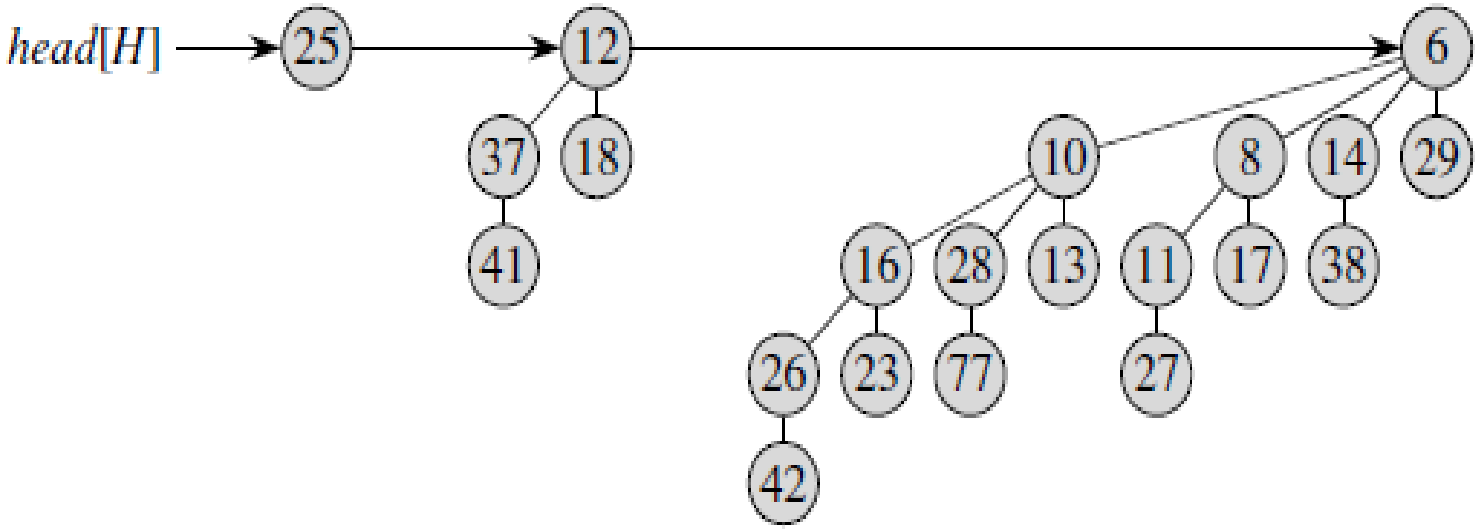


Extracting the node with minimum key

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.

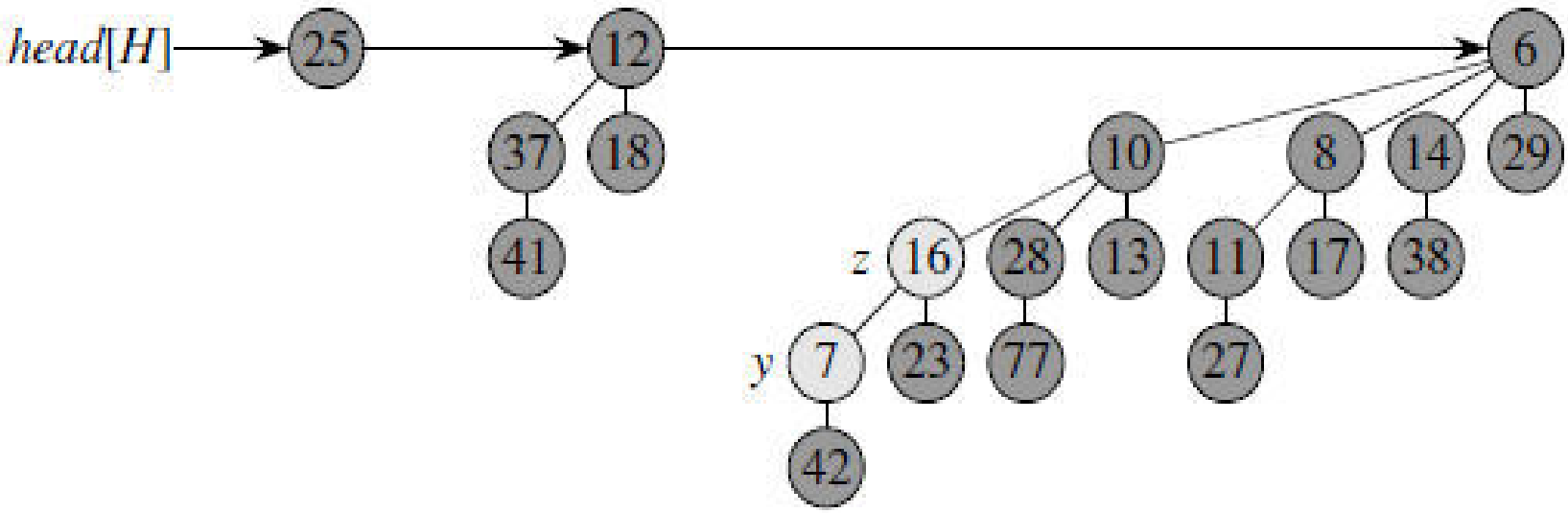
```
BINOMIAL-HEAP-EXTRACT-MIN(H)
find the root x with the minimum key in the root list of H,
        and remove x from the root list of H
H' ← MAKE-BINOMIAL-HEAP()
reverse the order of the linked list of x's children,
        and set head[H'] to point to the head of the resulting list
H ← BINOMIAL-HEAP-UNION(H, H')
return x
```

<u>Time Complexity:</u>

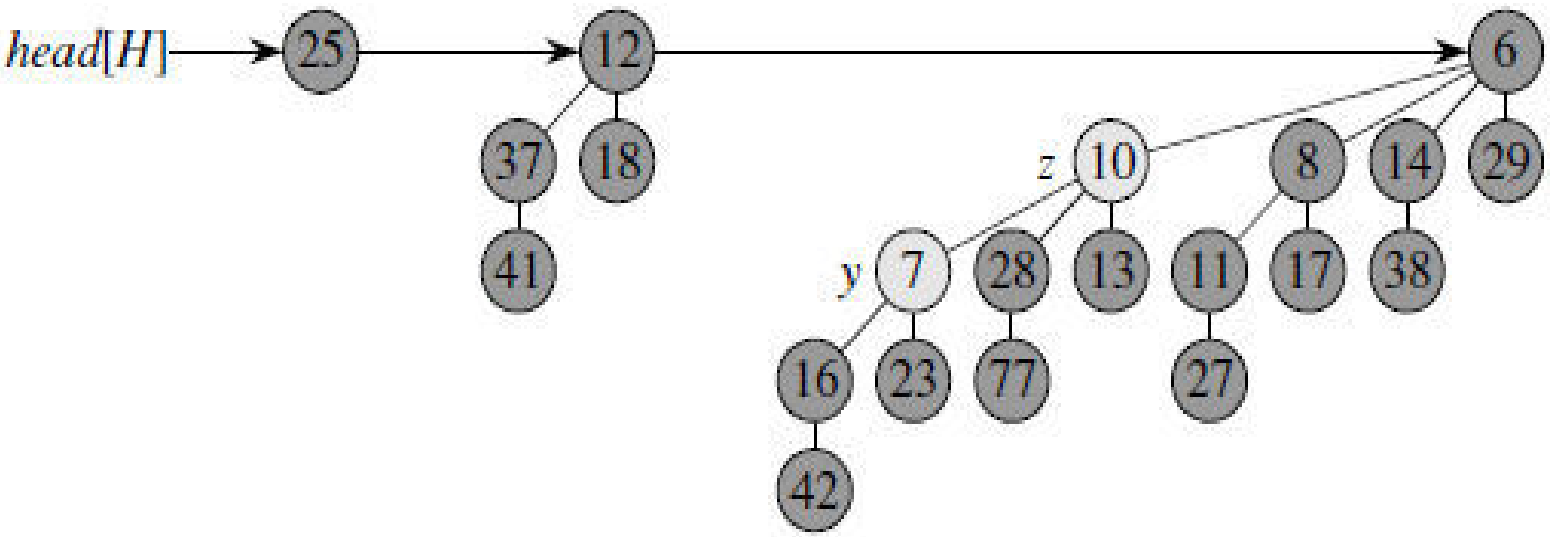
Time complexity of this algorithm is O(lg n).

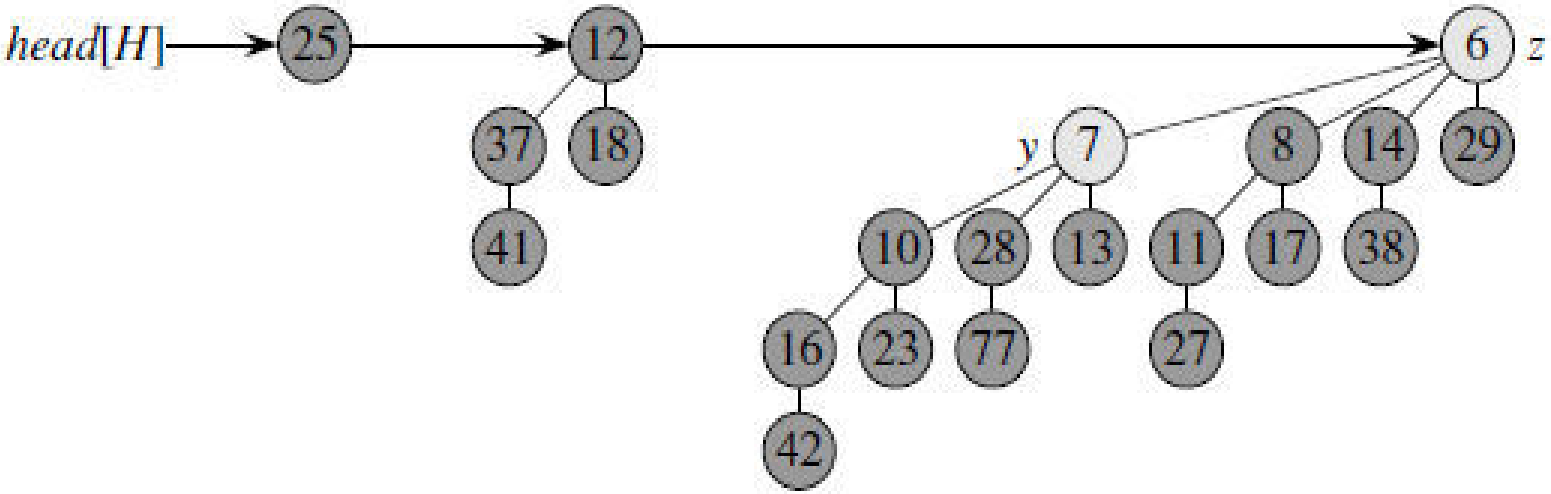
Decreasing a key

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)
if k > key[x]
    then error "new key is greater than current key"
key[x] ← k
y ← x
z ← p[y]
while z ≠ NIL and key[y] < key[z]
      do exchange key[y] ↔ key[z]
          ▷ If y and z have satellite fields, exchange them, too.
          y ← z
          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)
BINOMIAL-HEAP-DECREASE-KEY(H, x, -∞)
BINOMIAL-HEAP-EXTRACT-MIN(H)
```

<u>Time Complexity:</u>

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?

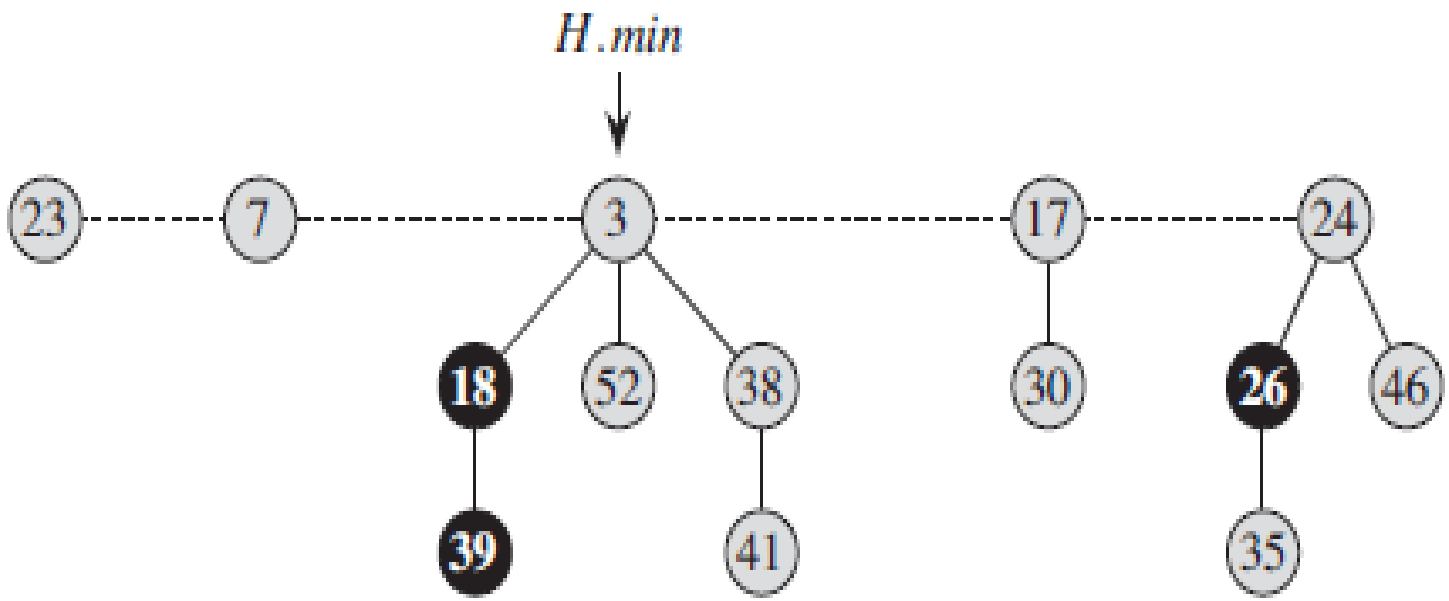
Fibonacci Heap

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

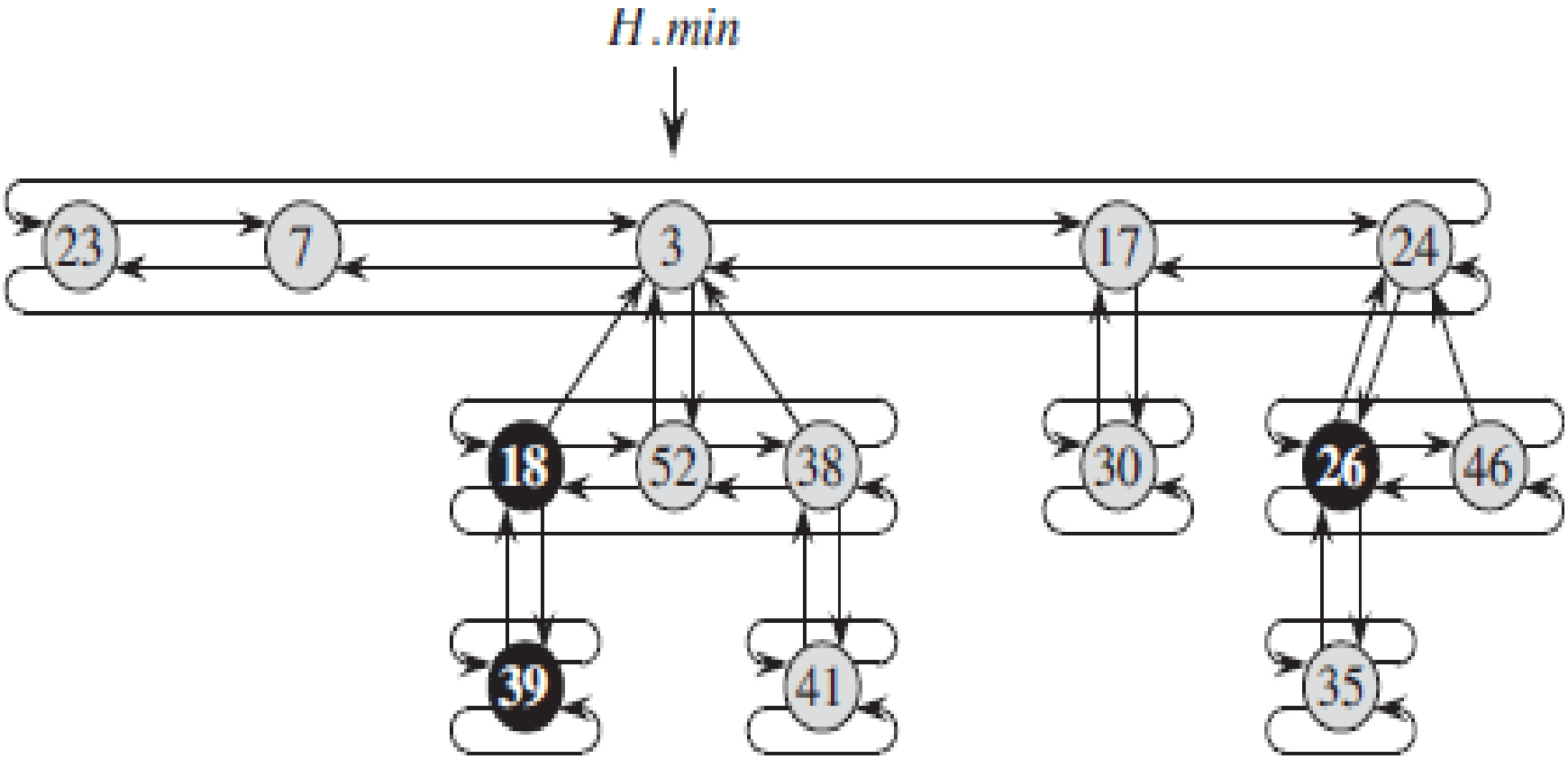
  x.degree→ number of children of node x

  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

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

<u>Maximum degree</u>

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:-

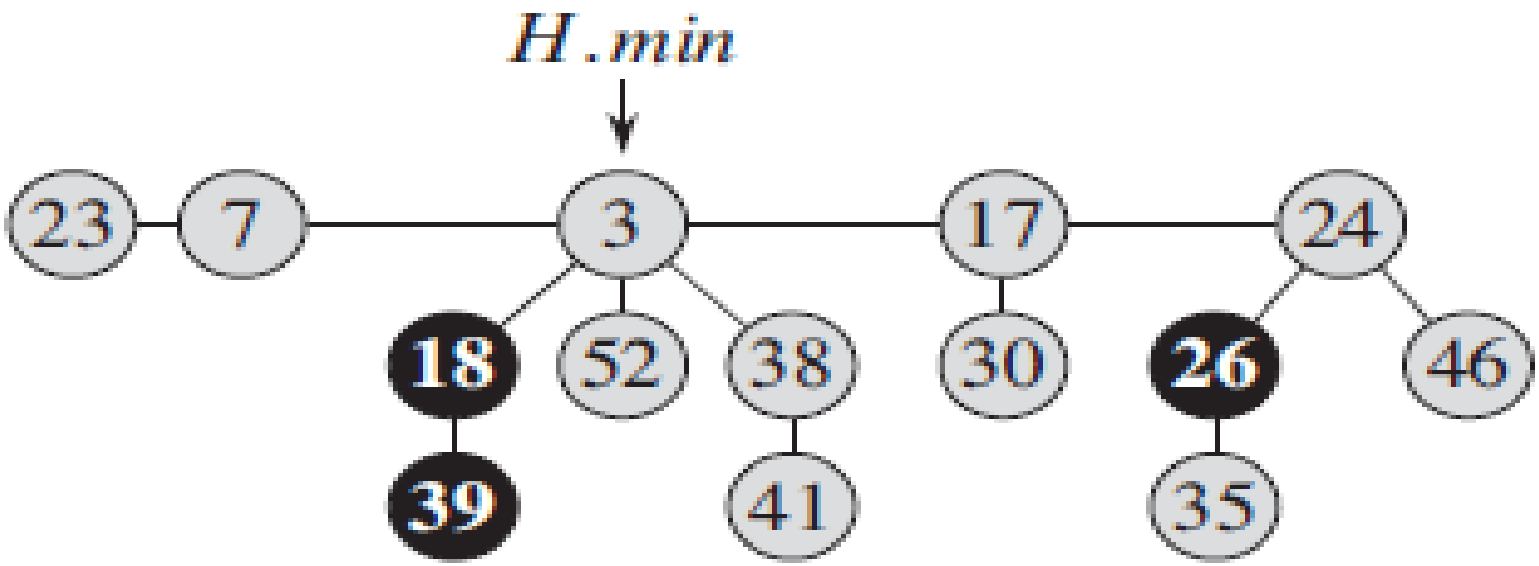
Amortized cost = Actual cost + change in potential function due to operation

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

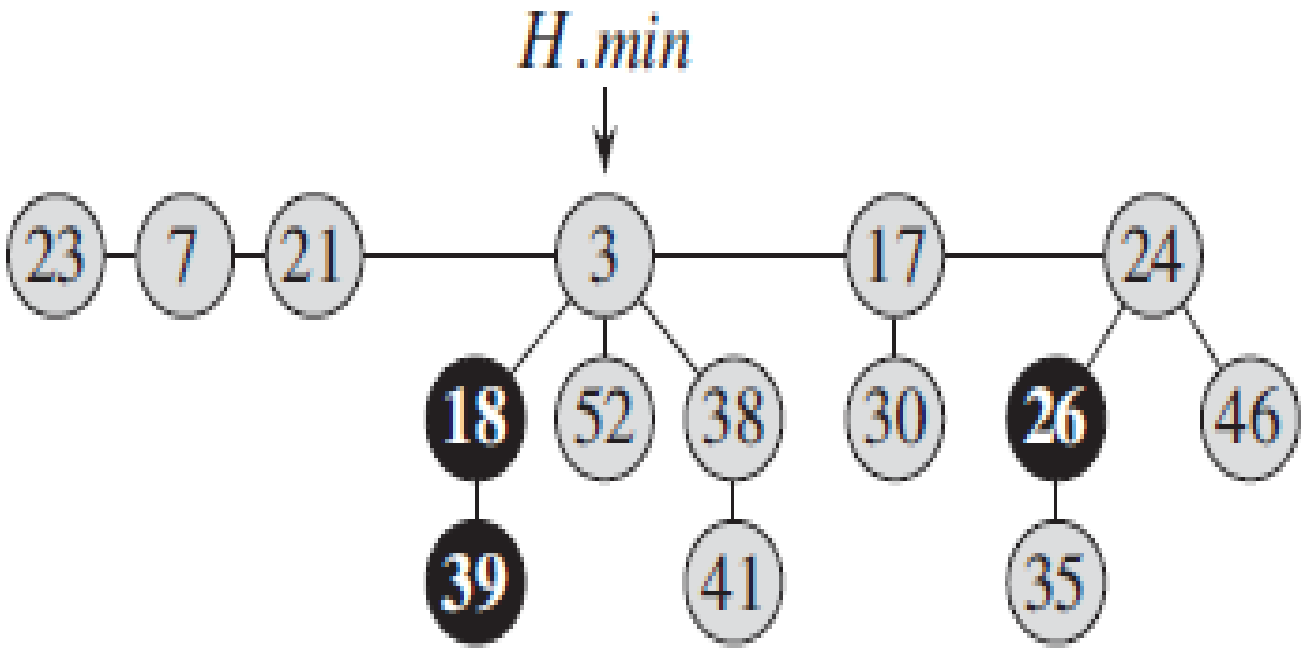
Inserting a node

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)
x.degree = 0
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == NIL
    create a root list for H containing just x
    H.min = x
else insert x into H's root list
    if x.key < H.min.key
        H.min = x
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 = $\Phi(H') - \Phi(H)$

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

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

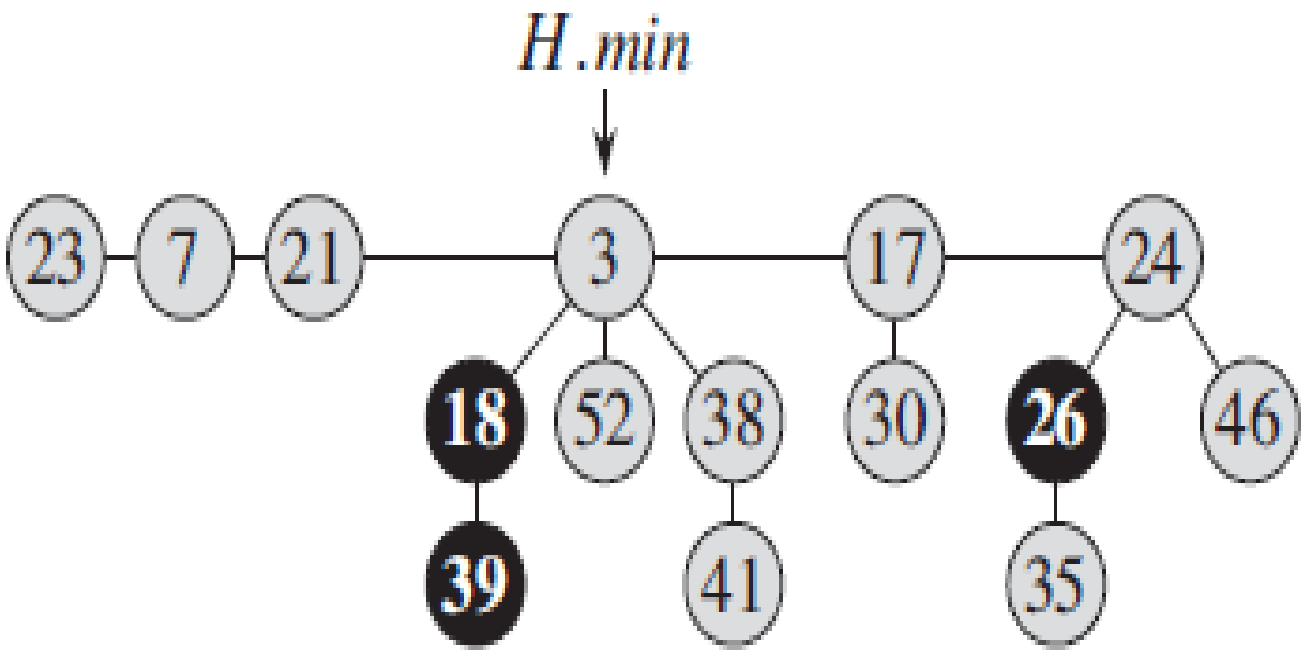
$$= 1$$

The actual cost = O(1), therefore

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

$$= O(1) + 1 = \mathbf{O(1)}$$

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)
H = MAKE-FIB-HEAP()
H.min = H1.min
concatenate the root list of H2 with the root list of H
if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
    H.min = H2.min
H.n = H1.n + H2.n
return H
```

Uniting two Fibonacci heaps

Amortied cost:

The change in potential function

$\Phi(\mathrm{H}) - (\Phi(\mathrm{H}_1) + \Phi(\mathrm{H}_2))$

$= \mathrm{t}(\mathrm{H}) + 2\mathrm{m}(\mathrm{H}) - (\mathrm{t}(\mathrm{H}_1) + 2\mathrm{m}(\mathrm{H}_1) + \mathrm{t}(\mathrm{H}_2) + 2\mathrm{m}(\mathrm{H}_2))$

$= 0$

Because t(H) = $\mathrm{t}(\mathrm{H}_1) + \mathrm{t}(\mathrm{H}_2)$ and m(H) = $\mathrm{m}(\mathrm{H}_1) + \mathrm{m}(\mathrm{H}_2)$

Therefore the amortized cost

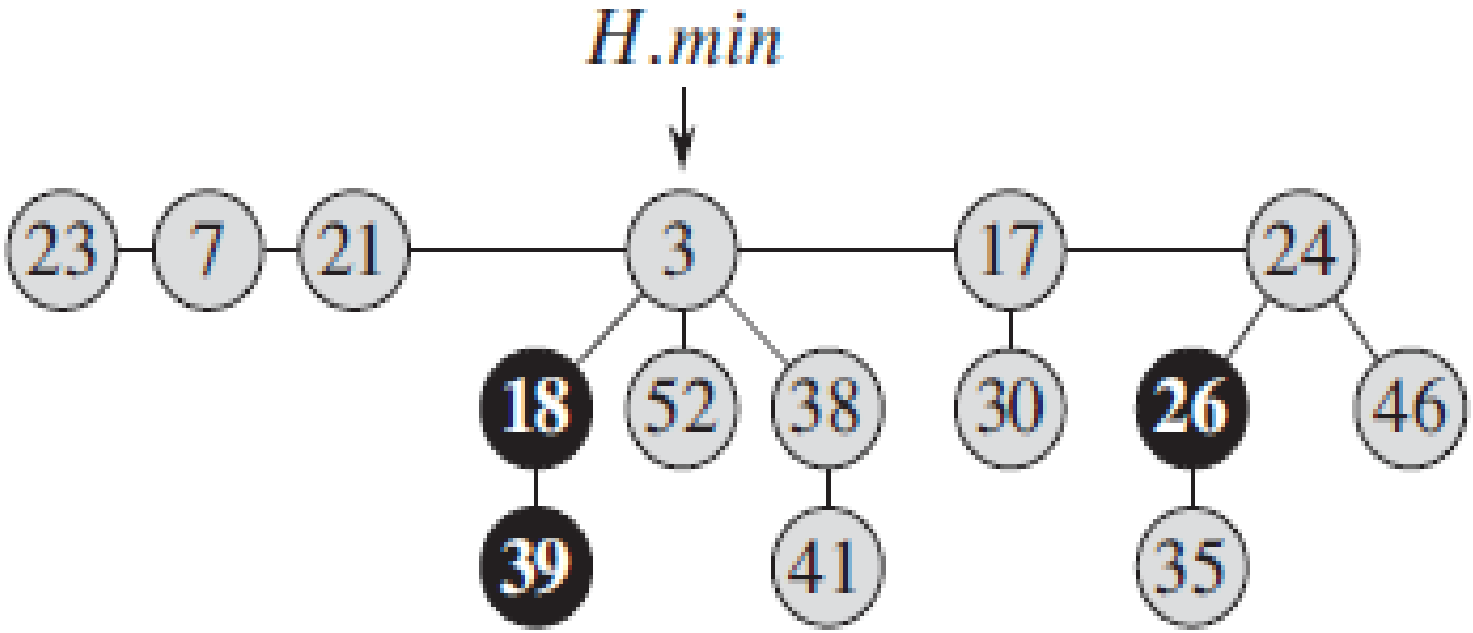
= actual cost + change in potential

= O(1) + 0

= O(1)

Extracting the minimum node

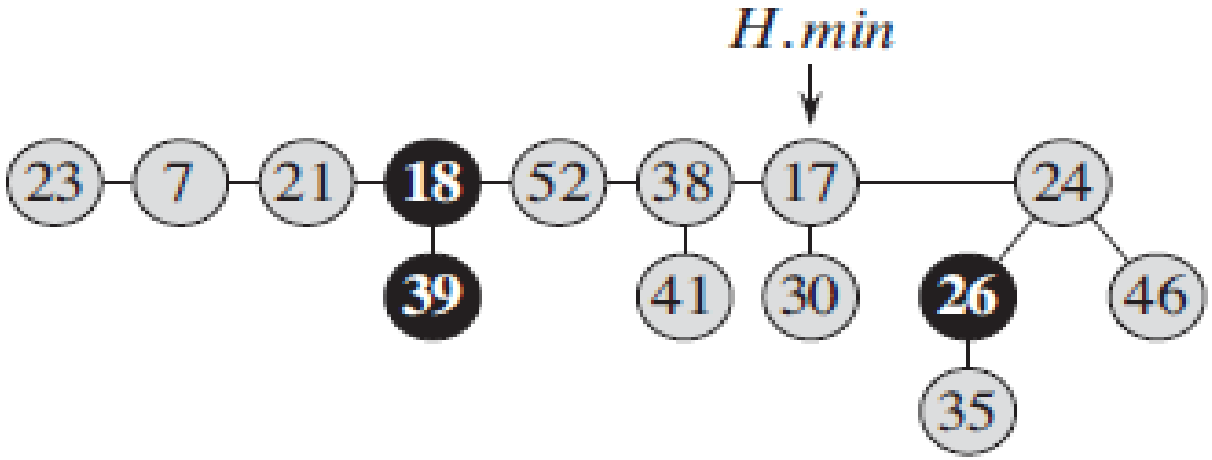
Example: Extract the minimum node from the following Fibonacci heap.



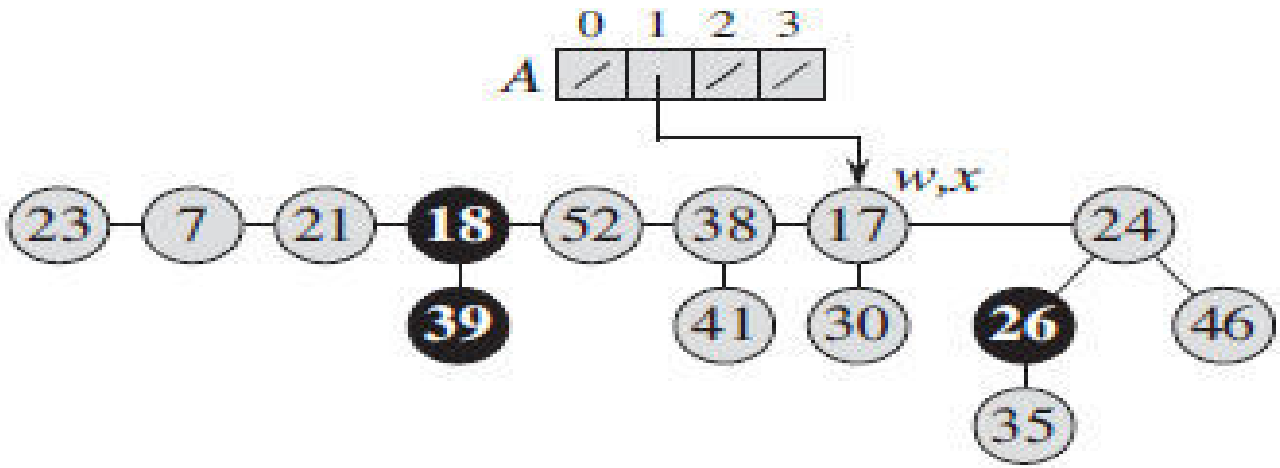
Extracting the minimum node

Solution:

Step-1:

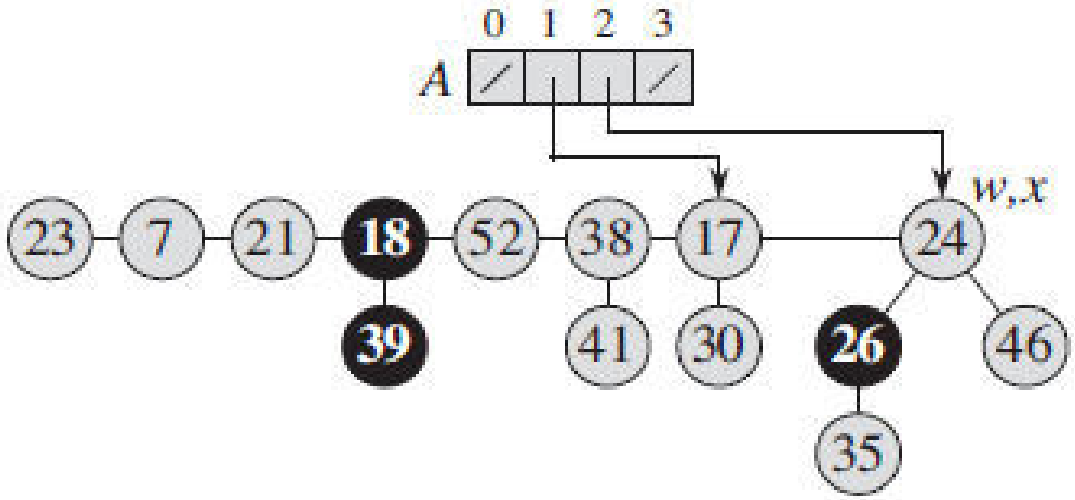


Step-2:

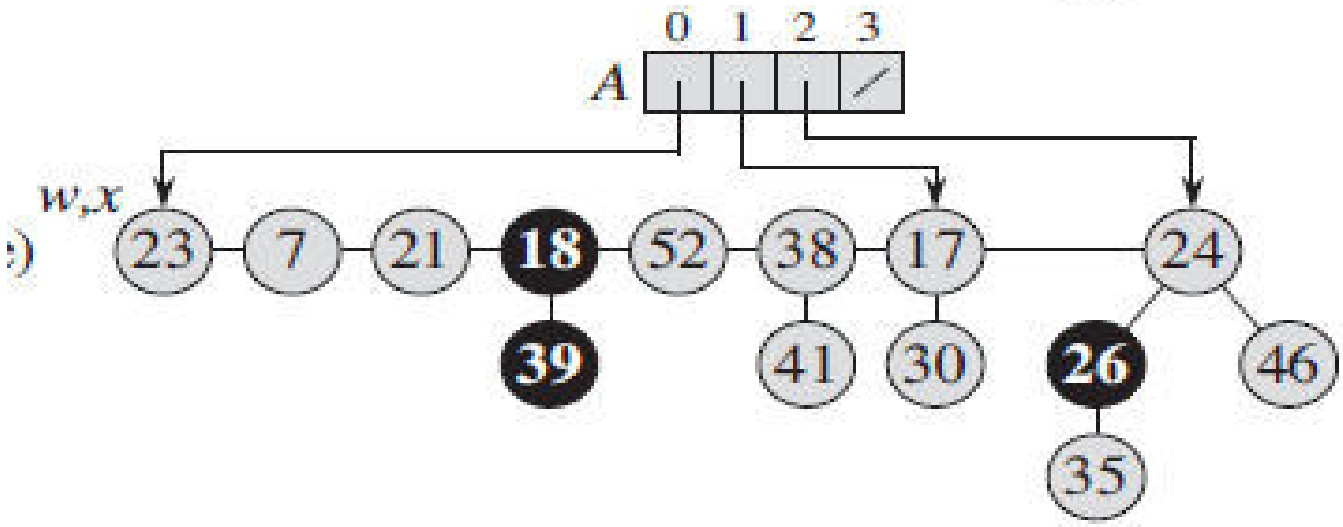


Extracting the minimum node

Step-3:

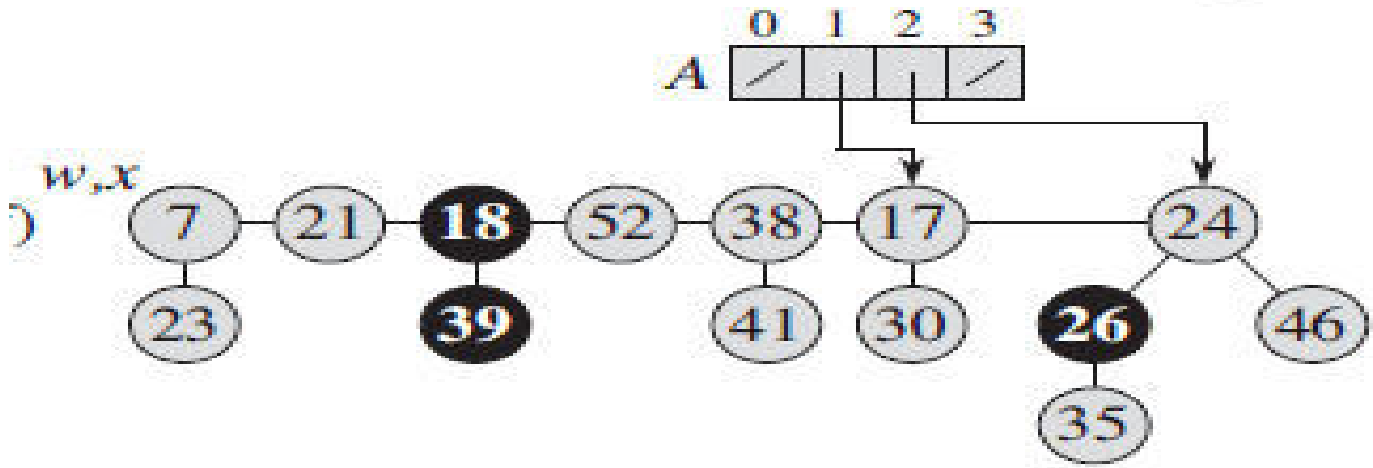


Step-4·

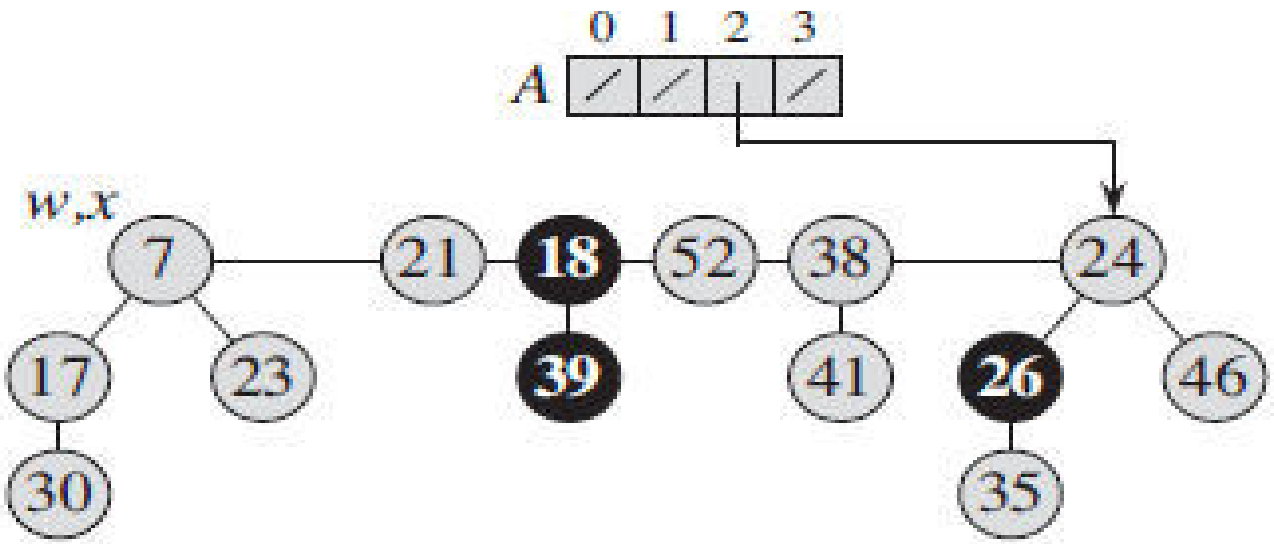


Extracting the minimum node

Step-5:

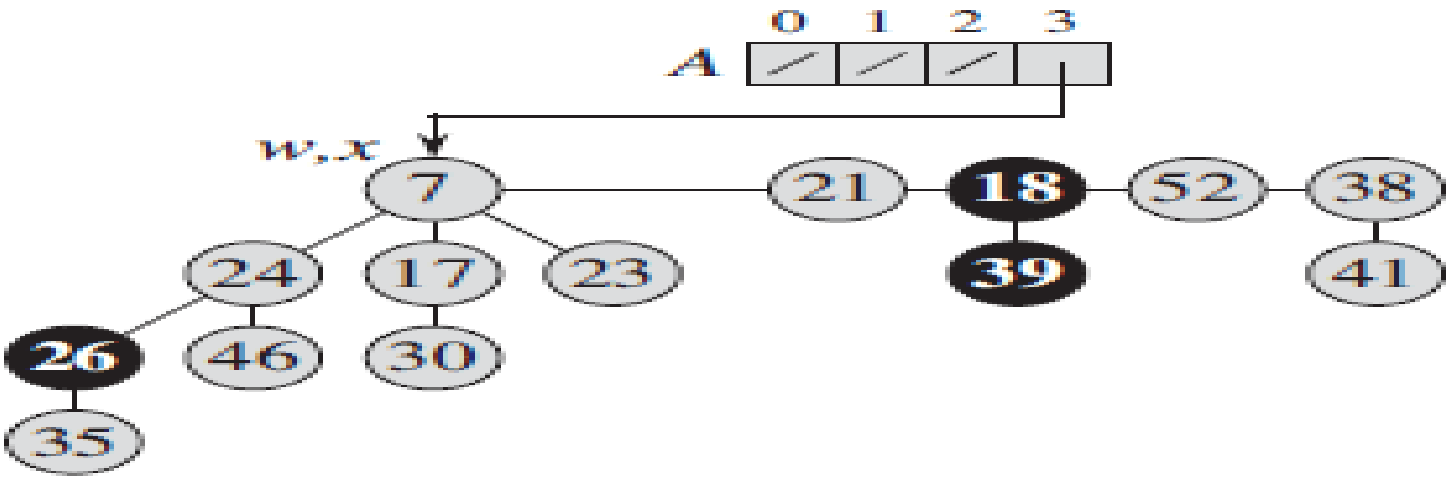


Step-6:

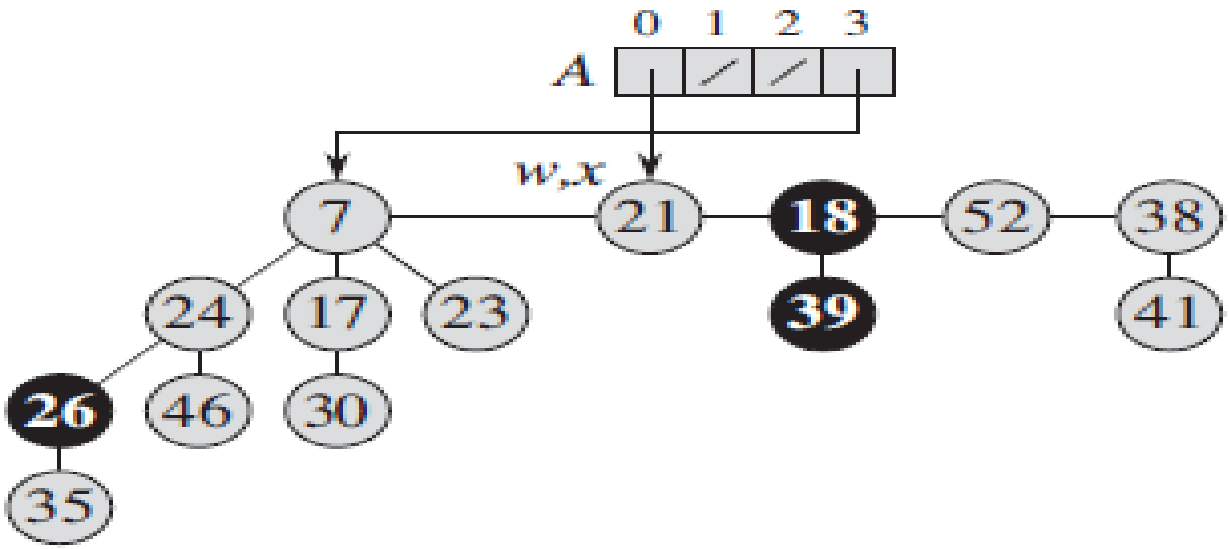


Extracting the minimum node

Step-7:

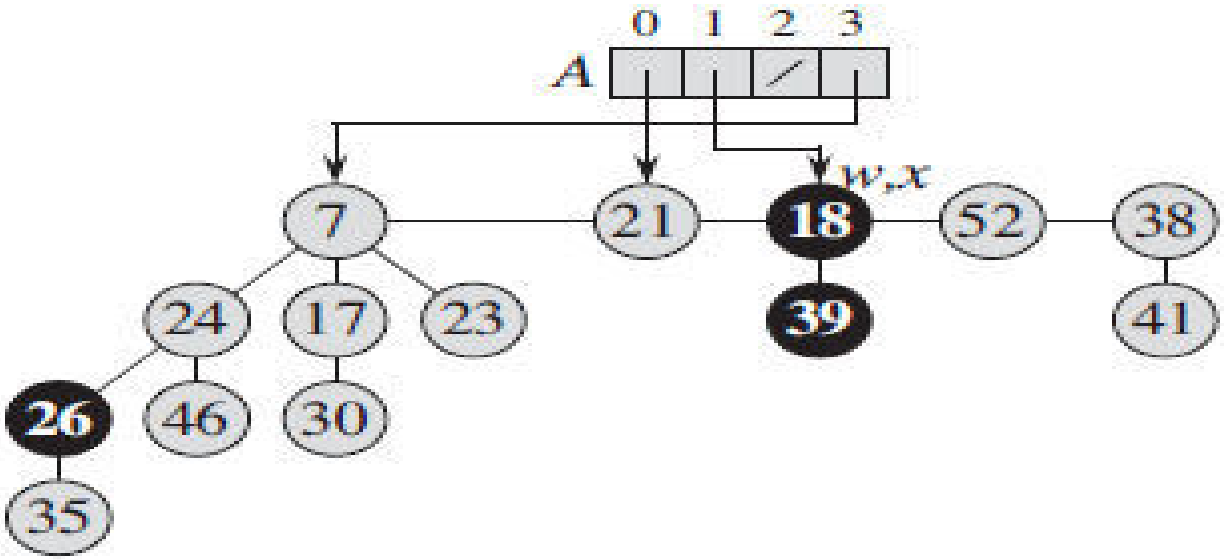


Step-8:

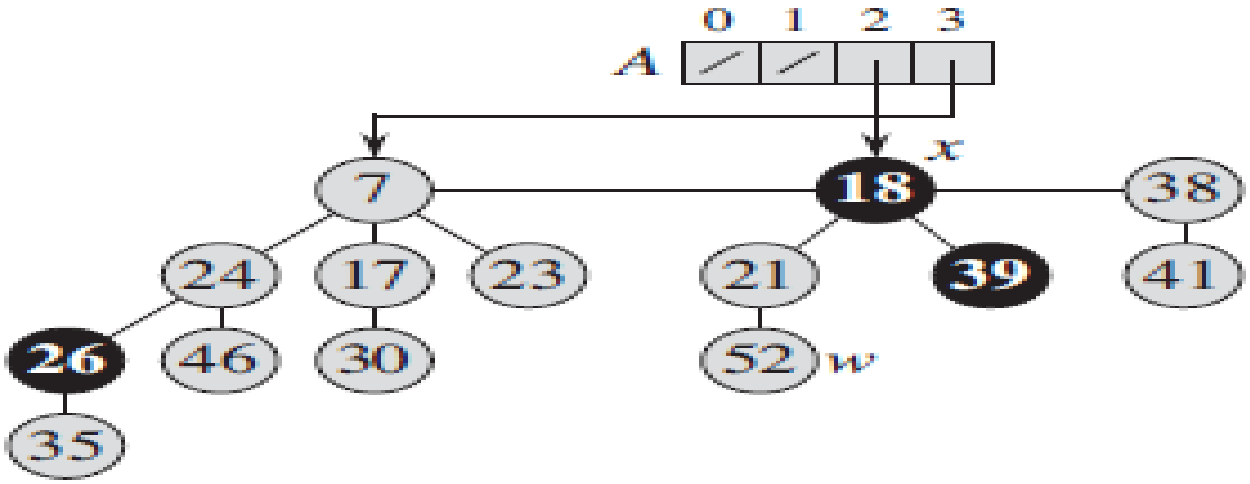


Extracting the minimum node

Step-9:

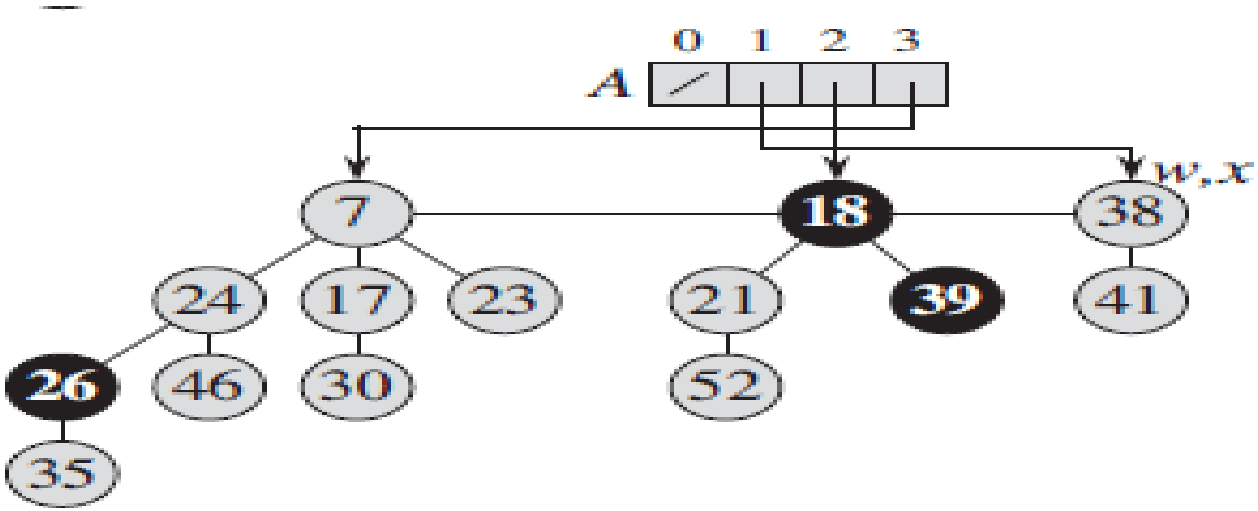


Step-10:

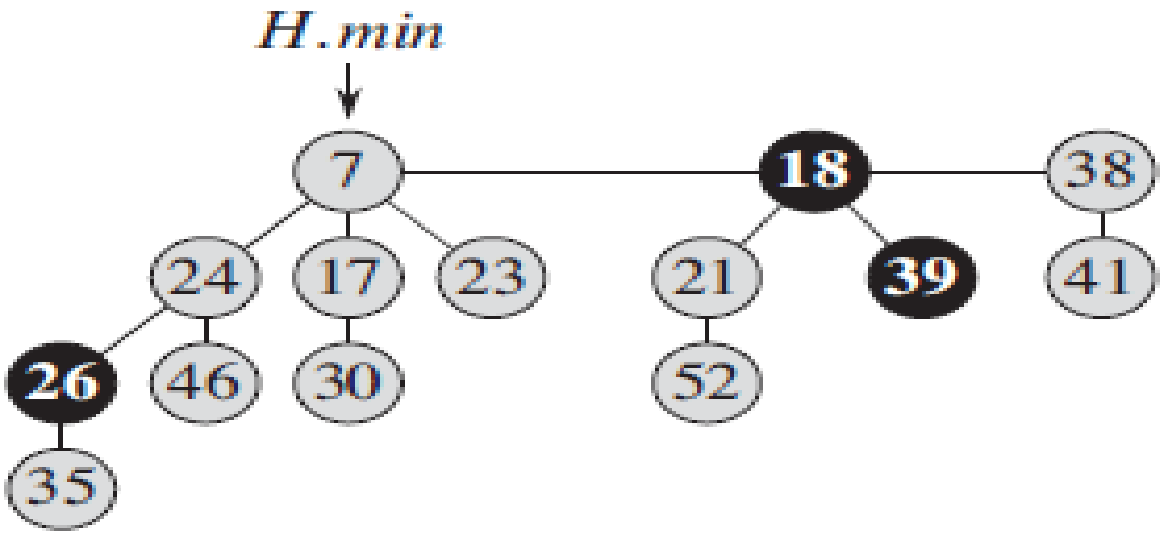


Extracting the minimum node

Step-11:



Step-12:



Final Fibonacci Heap

Extracting the minimum node

```
FIB-HEAP-EXTRACT-MIN(H)
z = H.min
if z ≠ NIL
    for each child x of z
        add x to the root list of H
        x.p = NIL
    remove z from the root list of H
    if z == z.right
        H.min = NIL
    else H.min = z.right
        CONSOLIDATE(H)
    H.n = H.n − 1
return z
```

Extracting the minimum node

```
CONSOLIDATE(H)
let A[0..D(H.n)] be a new array
for i = 0 to D(H.n)
    A[i] = NIL
for each node w in the root list of H
    x = w
    d = x.degree
    while A[d] ≠ NIL
        y = A[d]          // another node with the same degree as x
        if x.key > y.key
            exchange x with y
        FIB-HEAP-LINK(H, y, x)
        A[d] = NIL
        d = d + 1
    A[d] = x
H.min = NIL
for i = 0 to D(H.n)
    if A[i] ≠ NIL
        if H.min == NIL
            create a root list for H containing just A[i]
            H.min = A[i]
        else insert A[i] into H's root list
            if A[i].key < H.min.key
                H.min = A[i]
```

Extracting the minimum node

```
FIB-HEAP-LINK(H, y, x)
remove y from the root list of H
make y a child of x, incrementing x.degree
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))

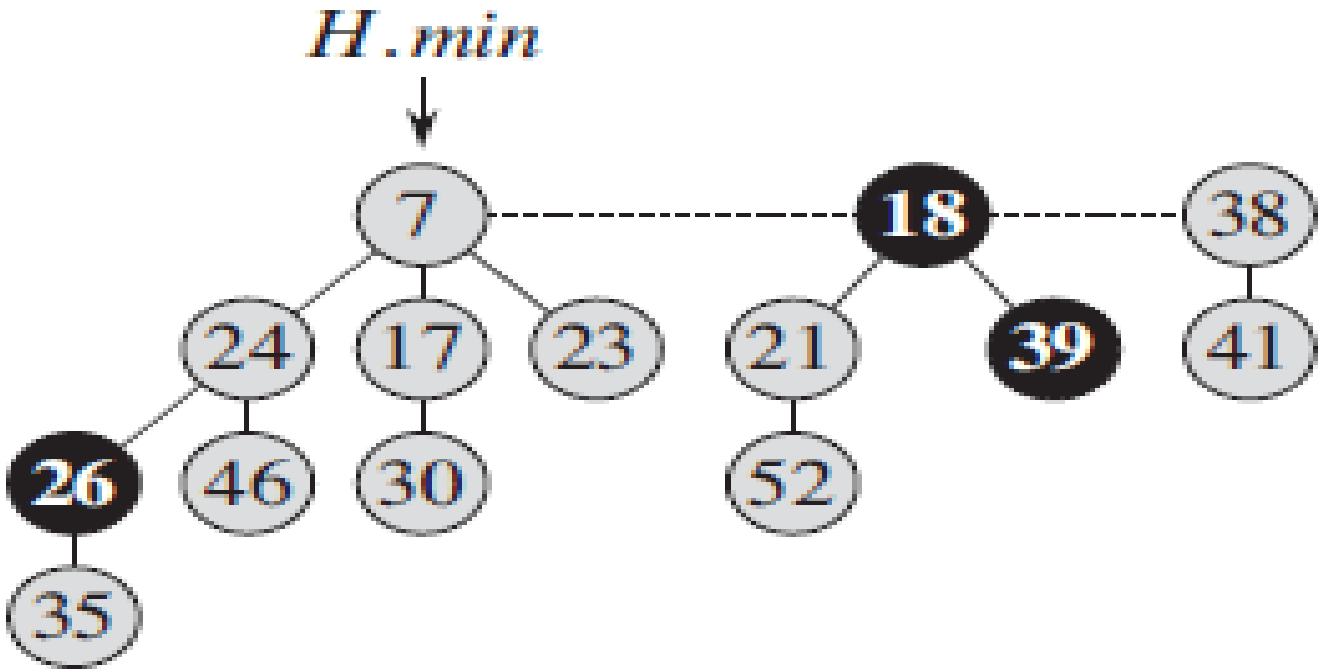
= O(D(n)) = O(log n))

Decreasing a key

Example: Consider following Fibonacci heap.

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

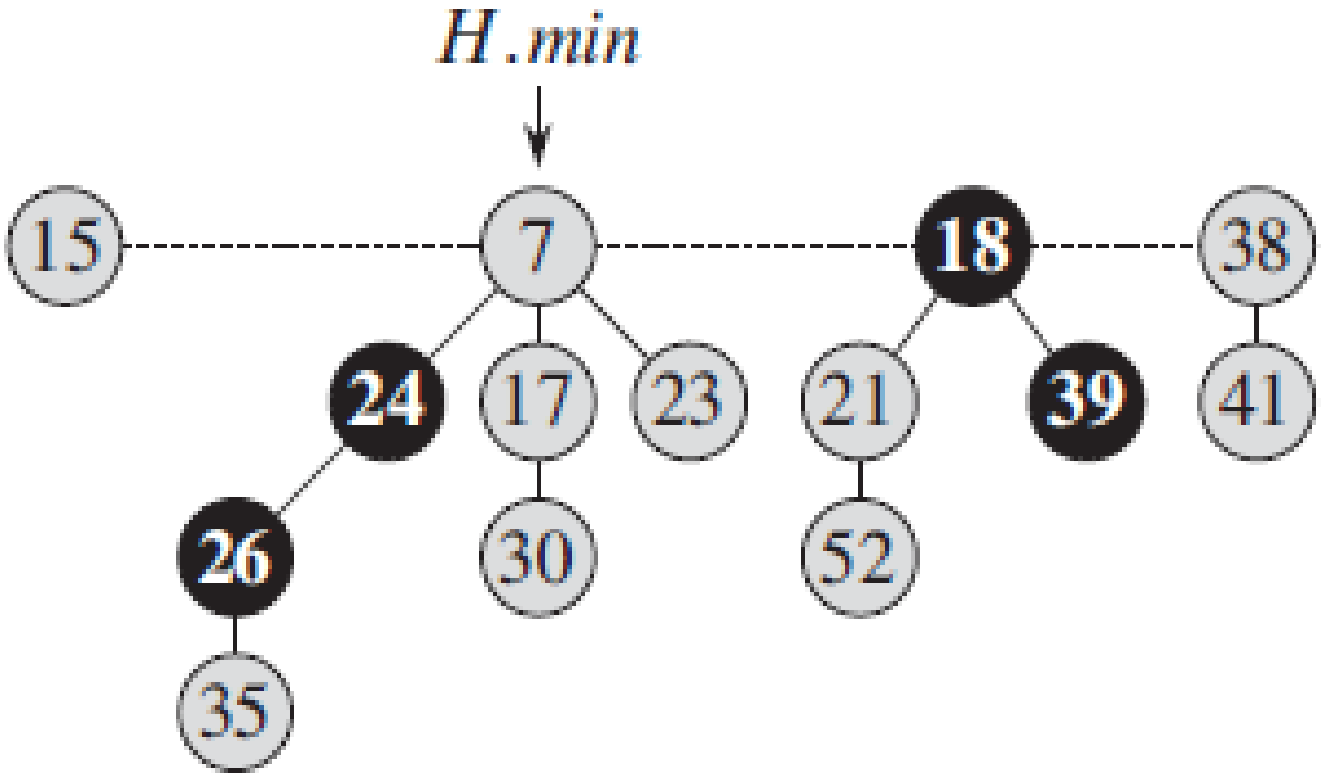
(2) After this, decrease node with key 35 to key value 5.



Decreasing a key

Solution:

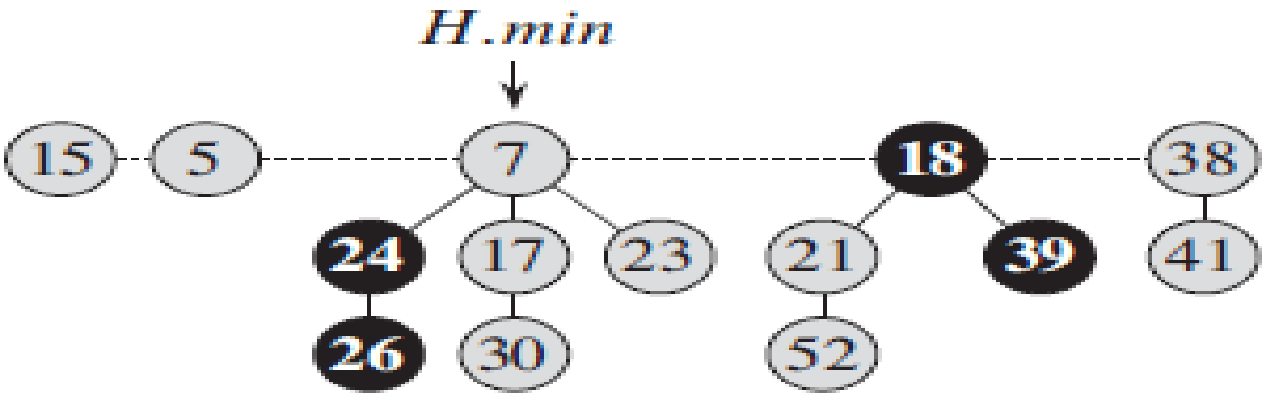
(1) Decrease the node with key 46 to key value 15.



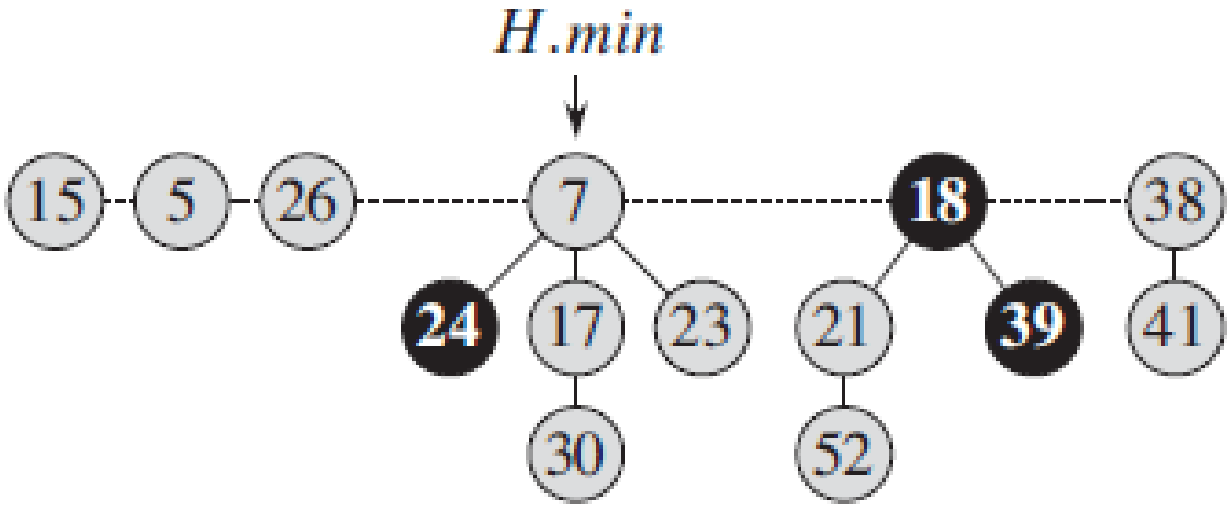
Decreasing a key

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

Step-1:

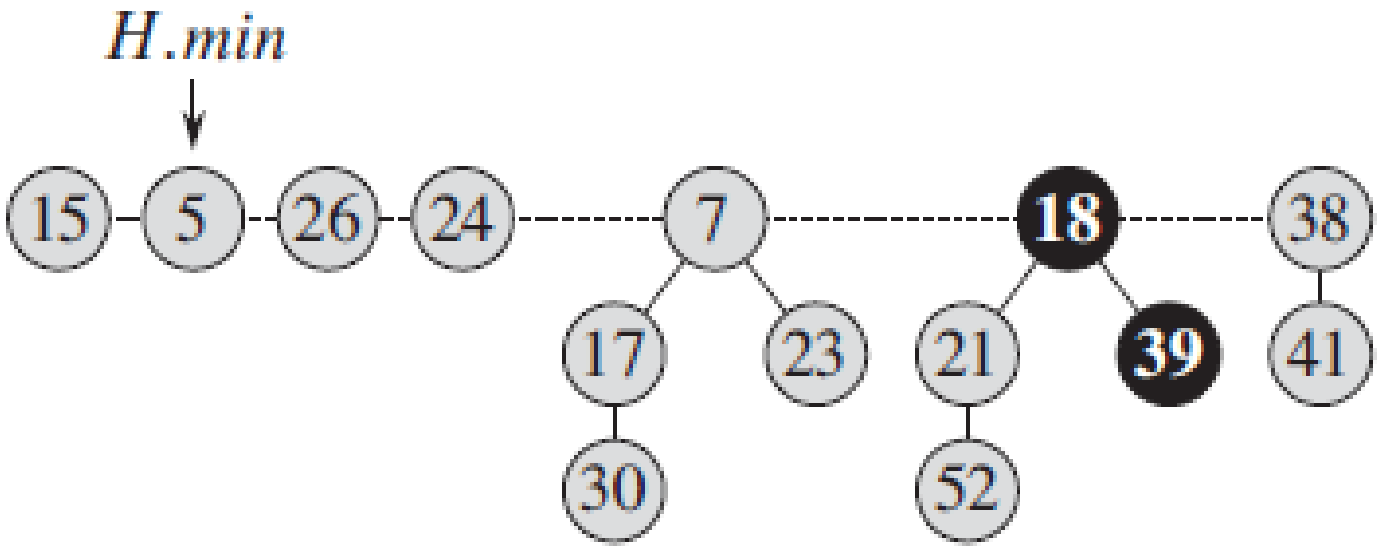


Step-2:



Decreasing a key

Step-3:



Final Fibonacci heap

Decreasing a key

```
FIB-HEAP-DECREASE-KEY(H, x, k)
if k > x.key
    error "new key is greater than current key"
x.key = k
y = x.p
if y ≠ NIL and x.key < y.key
    CUT(H, x, y)
    CASCADING-CUT(H, y)
if x.key < H.min.key
    H.min = x
```

Decreasing a key

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

```
CASCADING-CUT(H, y)
z = y.p
if z ≠ NIL
    if y.mark == FALSE
        y.mark = TRUE
    else CUT(H, y, z)
        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 $= O(c) + ((t(H') + 2m(H')) - (t(H) + 2m(H)))$

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

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

Design and Analysis of Algorithms

Unit-3

Greedy Algorithm

Greedy Algorithm

- A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms are used to solve optimization problems.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.
- We will solve the following problems using greedy algorithm:-
  - Activity selection problem
  - Knapsack problem
  - Minimum spanning tree problem
  - Shortest path problem

Activity selection problem

- Suppose we have a set $S = \{a_1, a_2, a_3, \ldots\ldots, a_n\}$ of n proposed ***activities*** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.
- Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \leq s_i < f_i < \infty$.
- Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.
- In the ***activity-selection problem***, we wish to select a maximum-size subset of mutually compatible activities.

Activity selection problem

Example: For example, consider the following set S of activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Solve this activity selection problem.'

Solution: The subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.

It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities. Another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

Activity selection problem

A recursive greedy algorithm

We assume that the n input activities are already ordered by monotonically increasing finish time. If not, we can sort them into this order in O(nlgn) time, breaking ties arbitrarily. Following algorithm solves the activity selection problem recursively.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n).

Time complexity: the running time of this algorithm is $\theta(n)$.

Activity selection problem

An iterative greedy algorithm

We assume that the n input activities are already ordered by monotonically increasing finish time. If not, we can sort them into this order in O(nlgn) time, breaking ties arbitrarily. Following algorithm solves the activity selection problem iteratively

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

Time complexity: The running time of this algorithm is $\theta(n)$.

Greedy algorithm property

Greedy choice property

we can assemble a globally optimal solution by making locally optimal (greedy) choices.

In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Optimal substructure property

A problem exhibits optimal substructure if an optimal solution to the problem

contains within it optimal solutions to subproblems.

Knapsack problem

- A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are n items available in the store and weight of ith item is w_i and its profit is p_i. What items should the thief take?
- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.
- Based on the nature of the items, Knapsack problems are categorized as
  1. Fractional Knapsack
  2. 0-1 Knapsack

Knapsack problem

Fractional Knapsack

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $\boldsymbol{x_i}$ of $\mathbf{i^{th}}$ item.

0-1 Knapsack

In this version of Knapsack problem, we cannot break an item, either pick the complete item or don't pick it (0-1 property).

Knapsack problem

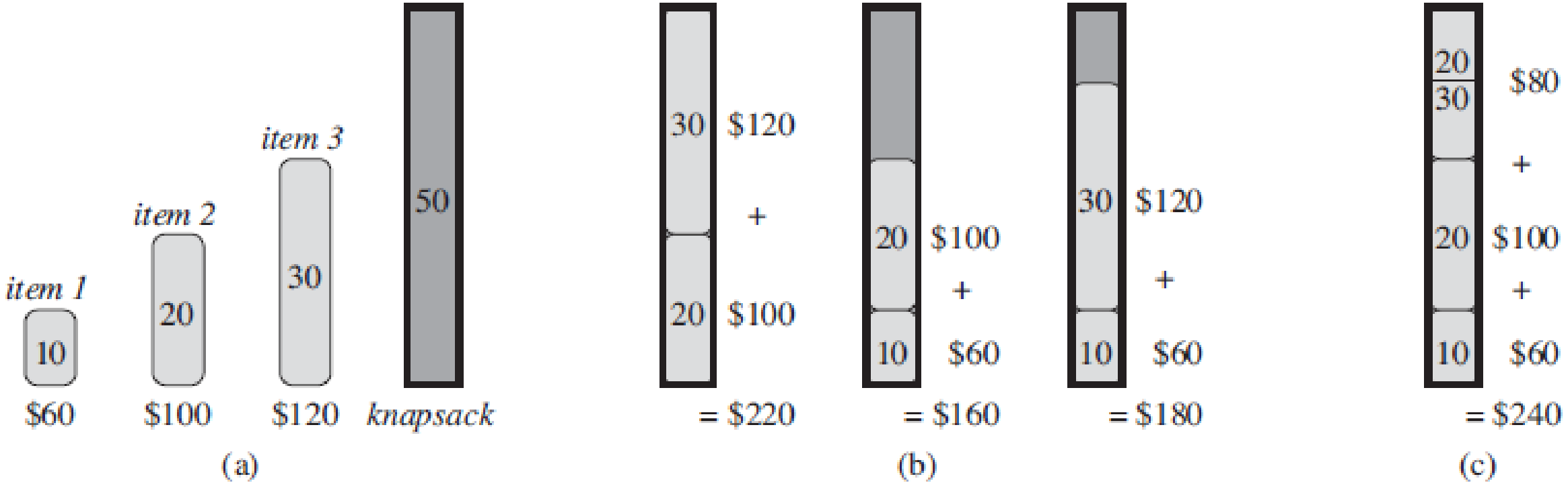
Greedy approach to solve the knapsack problem

- First, we compute value v_i/w_i for each item i.
- Arrange all the items in descending order on the basis of v_i/w_i .
- Put first item in the knapsack fully if weight of first item is less than or equal to W. Put first item in the knapsack partially if weight of first item is greater than W.
- Similarly, put the second item in the knapsack fully if there is a sufficient space in the knapsack. Otherwise, we put second item partially.
- This process continue till Knapsack is filled.

Knapsack problem

Note: Greedy algorithm solves the fractional knapsack but 0-1 Knapsack can not be solved by greedy algorithm.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in following figure:-



Knapsack problem

Example: Consider the following instance for knapsack problem. Find the solution using Greedy method:

N= 8, W=130

W = {21, 31, 43, 53, 41, 63, 65, 75}

V = {11, 21, 31, 33, 43, 53, 65, 65}

Solution: Compute

$$\frac{V}{W} = \left\{ \frac{11}{21}, \frac{21}{31}, \frac{31}{43}, \frac{33}{53}, \frac{43}{41}, \frac{53}{63}, \frac{65}{65}, \frac{65}{75} \right\}$$

= { 0.52, 0.68, 0.72, 0.62, 1.05, 0.84, 1, 0.87 }

First arrange all items in descending order of their value per weight i.e. (v_i/w_i).

W' = {41,65,75,63,43,31,53,21}

V' = {43,65,65,53,31,21,33,11}

Descending order of items = item5, item7, item8, item6, item3, item2, item4, item1

Therefore, optimal solution = (0, 0, 0, 0, 1, 0, 1, 24/75)

Optimal value = 43 + 65 + (24/75)*65 = 108 + 20.8 = **128.8**

Knapsack problem

Example: Consider the following instance for knapsack problem. Find the solution using Greedy method:

w = (5, 10, 20, 30, 40), v = (30, 20, 100, 90,160)

The capacity of knapsack W = 60

Example: Given the six items in the table below and a Knapsack with Weight 100, what is the solution to the Knapsack problem in all concepts. i.e. explain greedy all approaches and find the optimal solution.

ITEM ID	WEIGHT	VALUE	VALUE/WEIGHT
A	100	40	.4
B	50	35	.7
C	40	20	.5
D	20	4	.2
E	10	10	1
F	10	6	.6

Knapsack problem

Algorithm: Assume all the items are arranged in descending order of their value per weight i.e. (v_i/w_i).

```
Fractional_Knapsack(w, v, W)
        n = length[w]
        for i=1 to n
         do      x[i] = 0
        i=1
        weight = 0
        while( I <= n and weight < W)
        do      if(weight+w[i] < = W)
                then    x[i] = 1
                        weight = weight + w[i]
                else
                        x[i] =  (W – weight)/w_i
                        weight = W
        return x
```

- Time complexity of this algorithm will be O(n) if all the items are arranged in descending order of their value per weight .
- Time complexity of this algorithm will be O(nlogn) if all the items are not arranged in descending order of their value per weight .

Minimum Spanning Tree

Spanning tree:

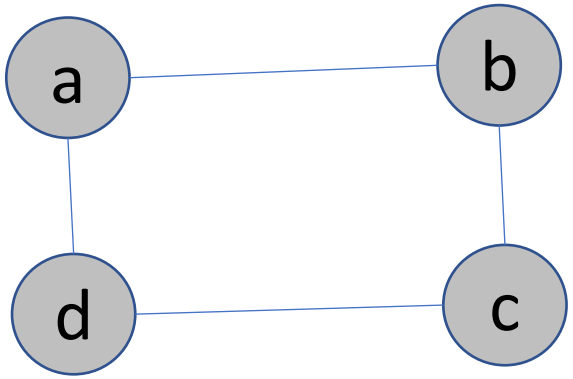
Spanning tree is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together.

General Properties of Spanning Tree

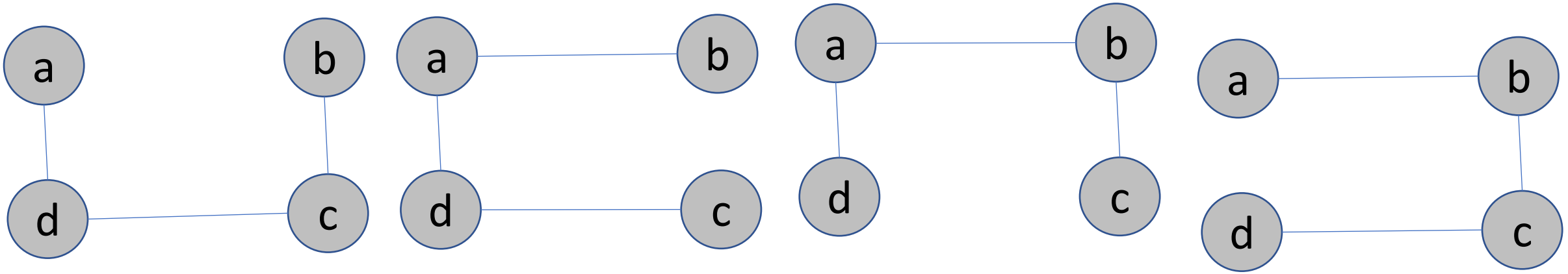
- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop.
- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Minimum Spanning Tree

Example: Consider the following graph:-



All the spanning tree of this graph are the following:-

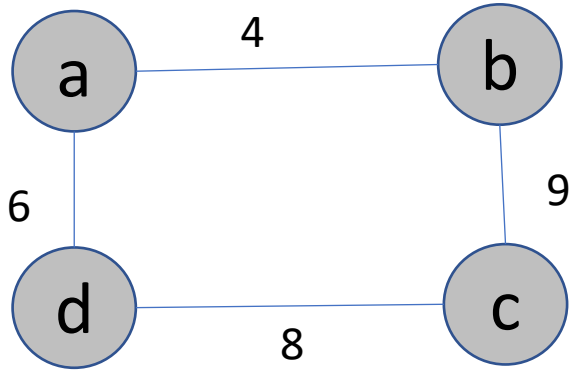


Minimum Spanning Tree

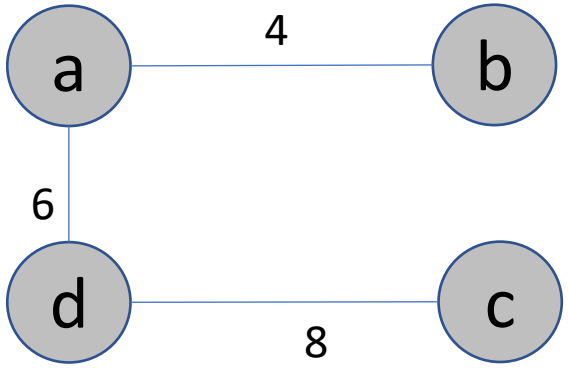
If the given graph is weighted graph, then we define the minimum spanning tree.

Definition: A spanning tree is said to be minimum spanning tree if sum of weights of all the edges in the tree is smallest.

Example: Consider the following graph:-



Graph



Total cost = 18

Minimum spanning tree

Minimum Spanning Tree

- In this chapter, we will study two algorithms to find the minimum spanning tree.

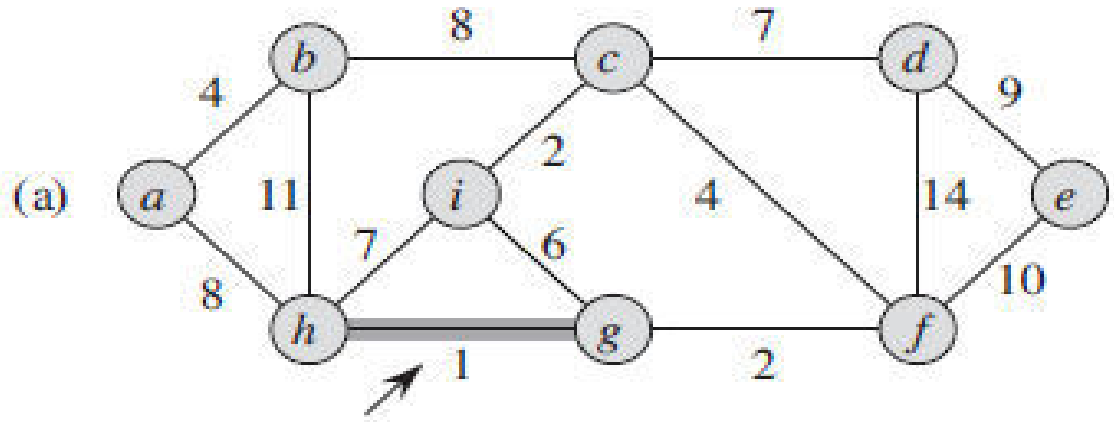
(1) Kruskal's Algorithm

(2) Prim's Algorithm

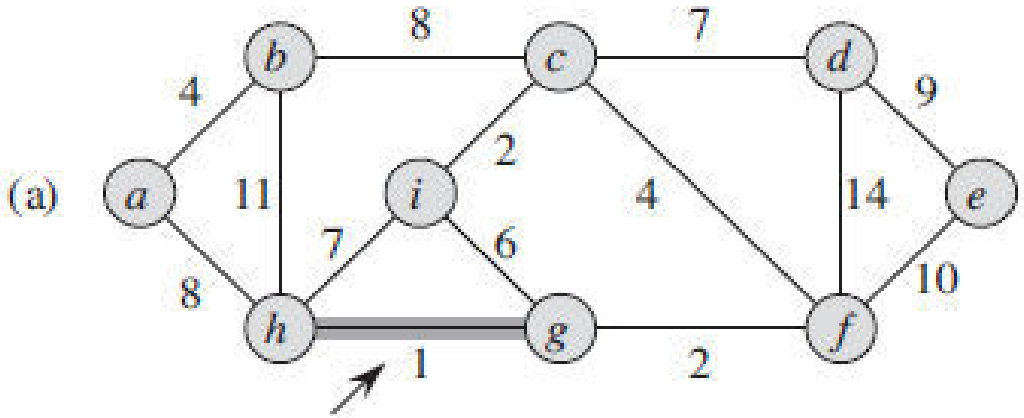
Both algorithms are based on Greedy approach.

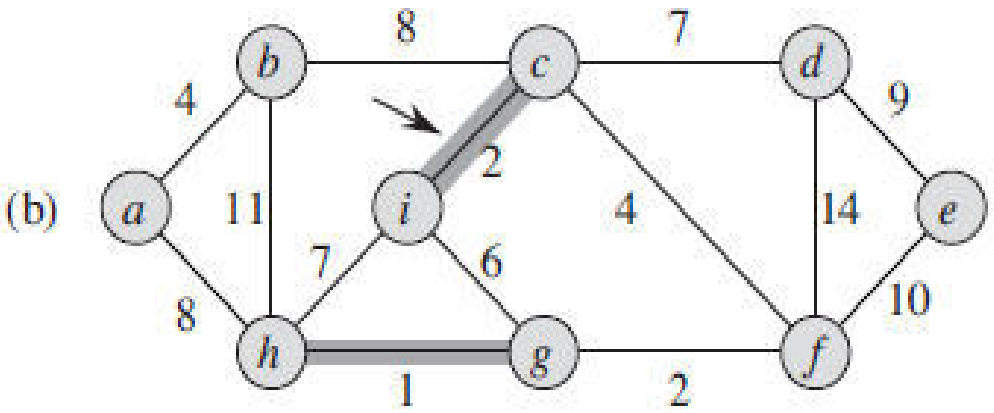
Kruskal's Algorithm

Example: Find the minimum spanning of the following graph using Kruskal algorithm.

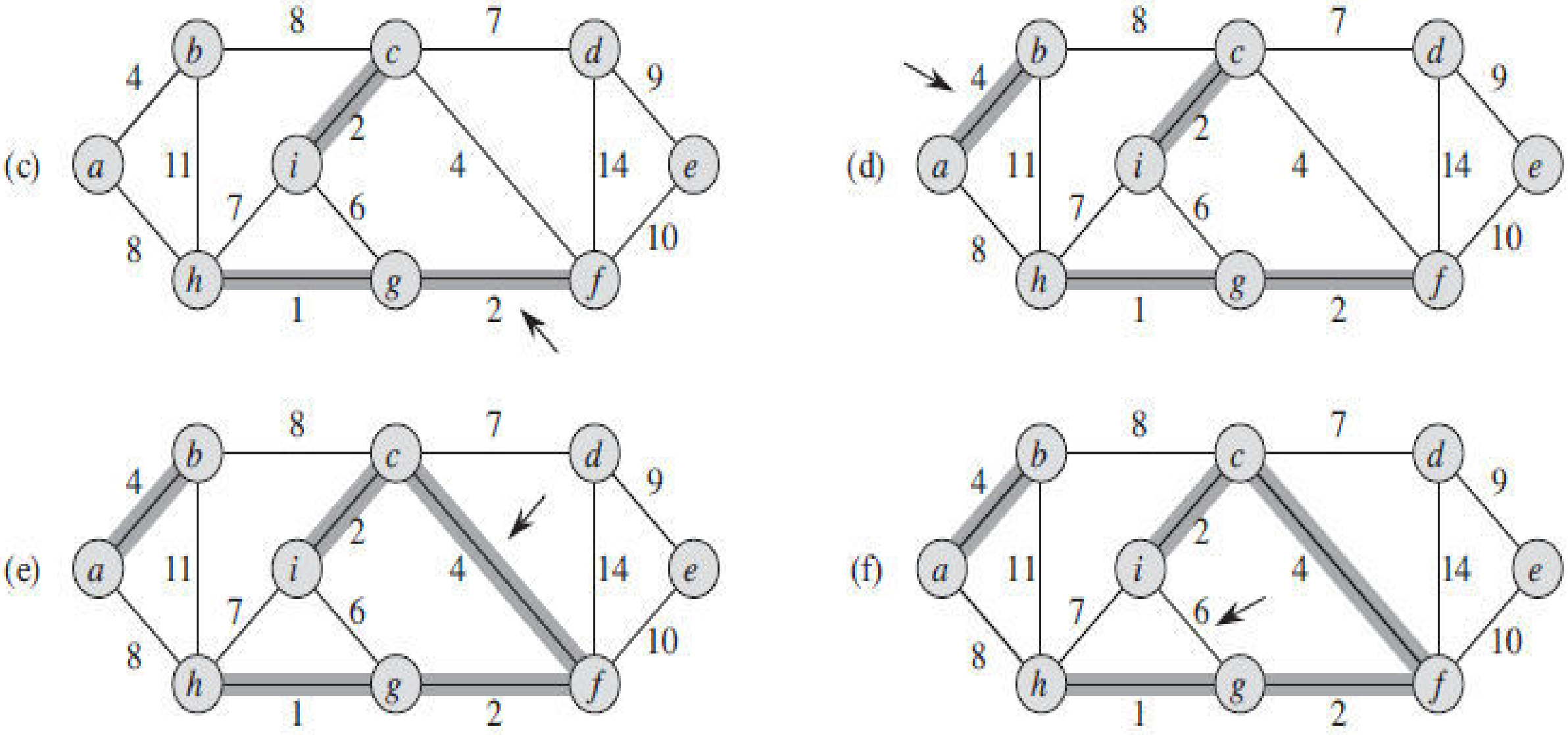


Solution:

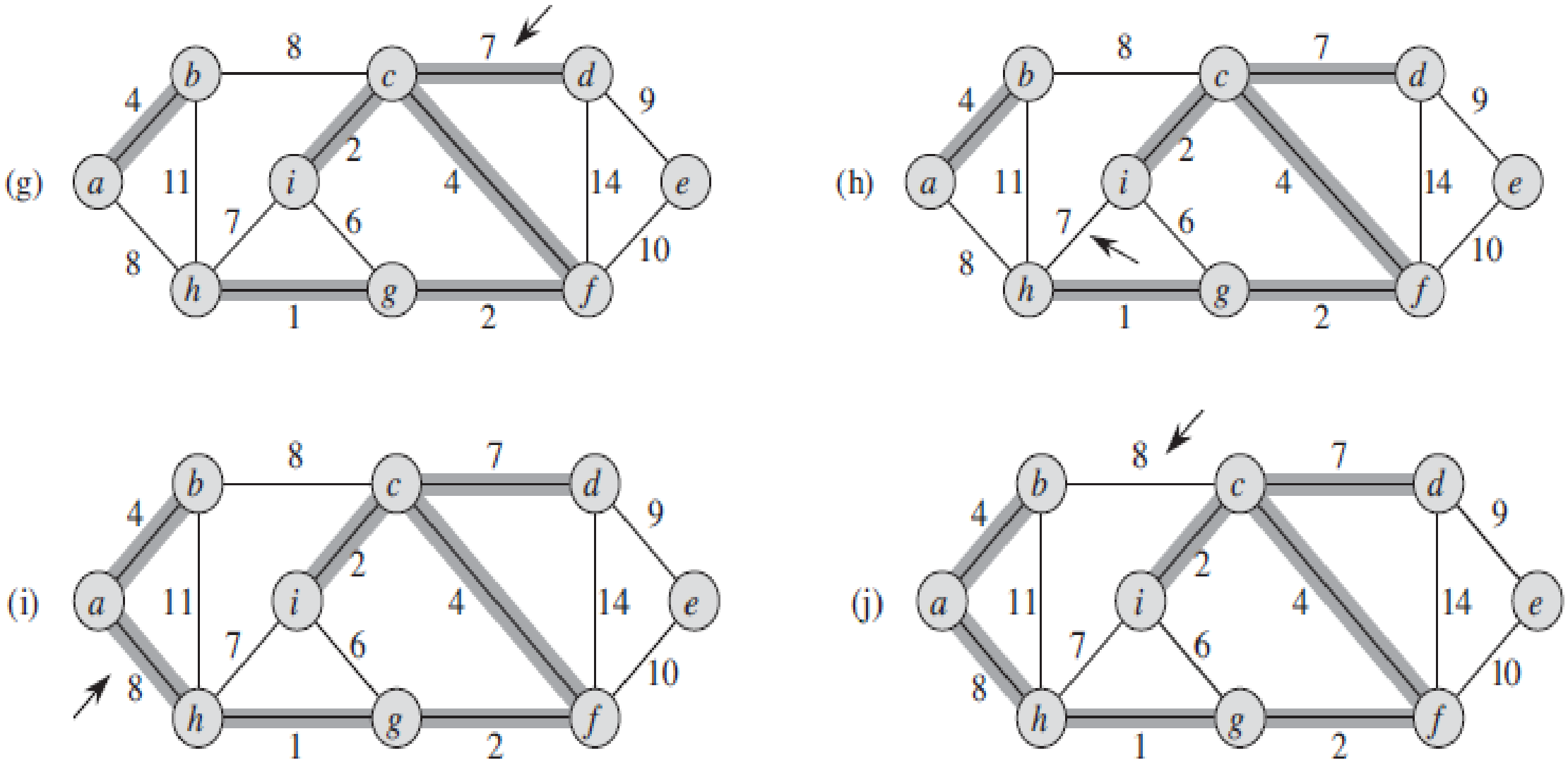




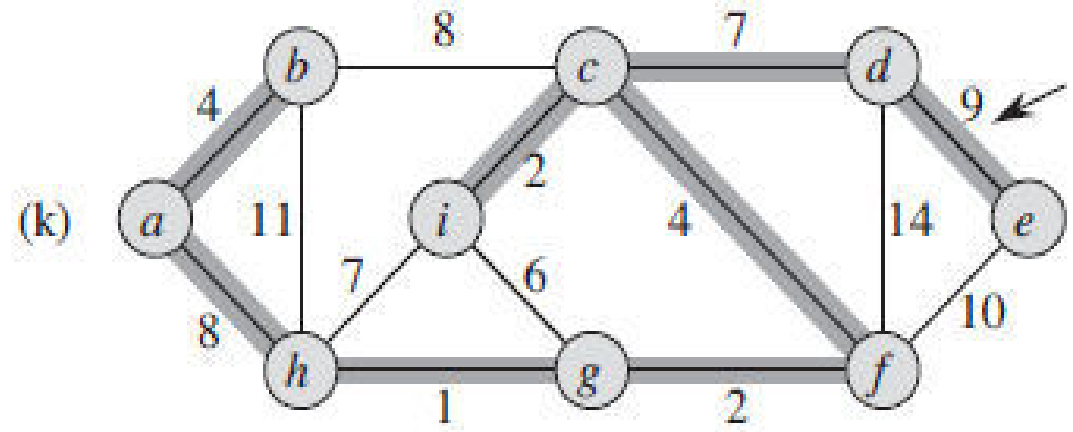
Kruskal's Algorithm

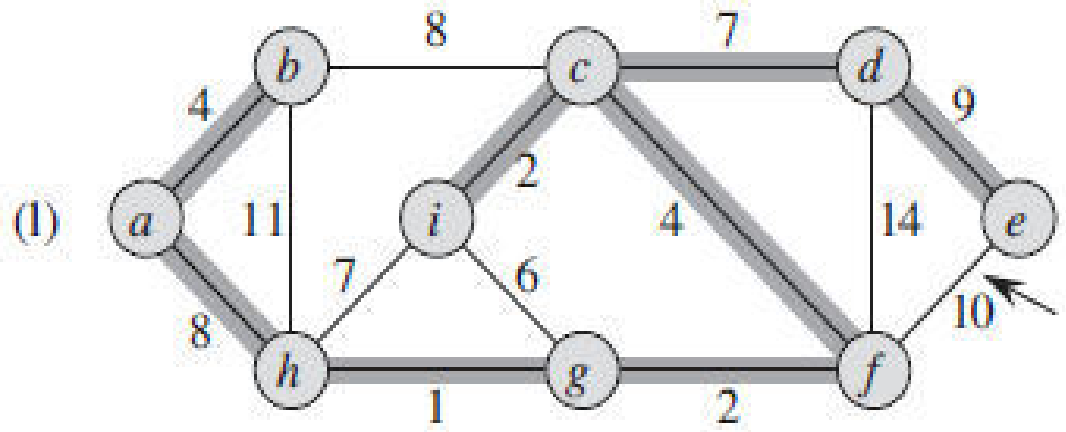


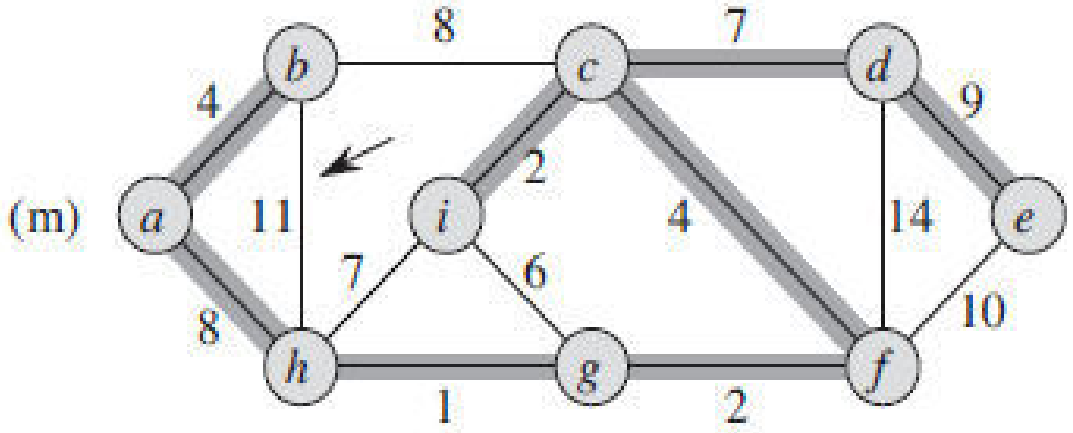
Kruskal's Algorithm

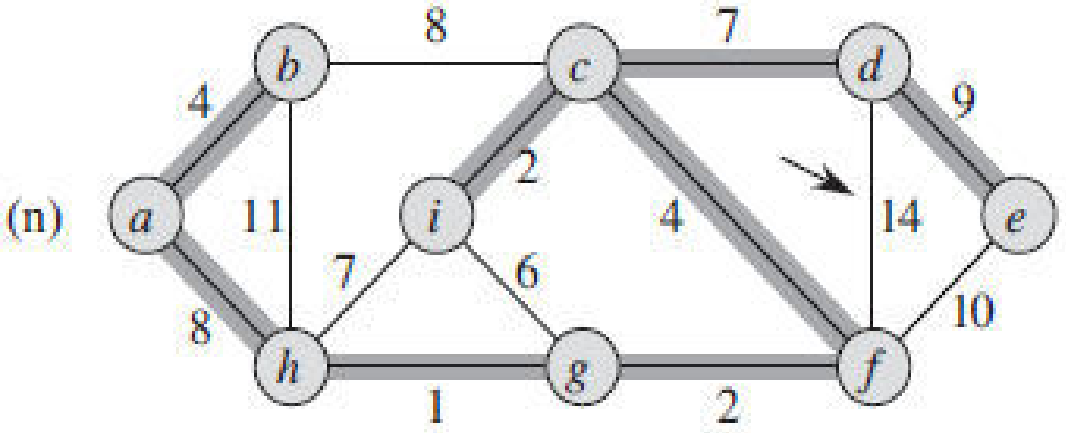


Kruskal's Algorithm









Total cost = 37

Final minimum spanning tree

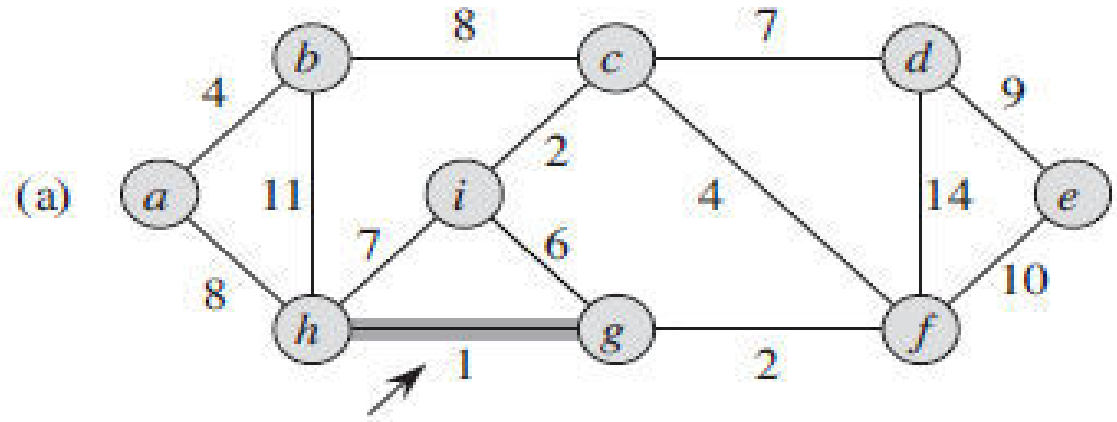
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

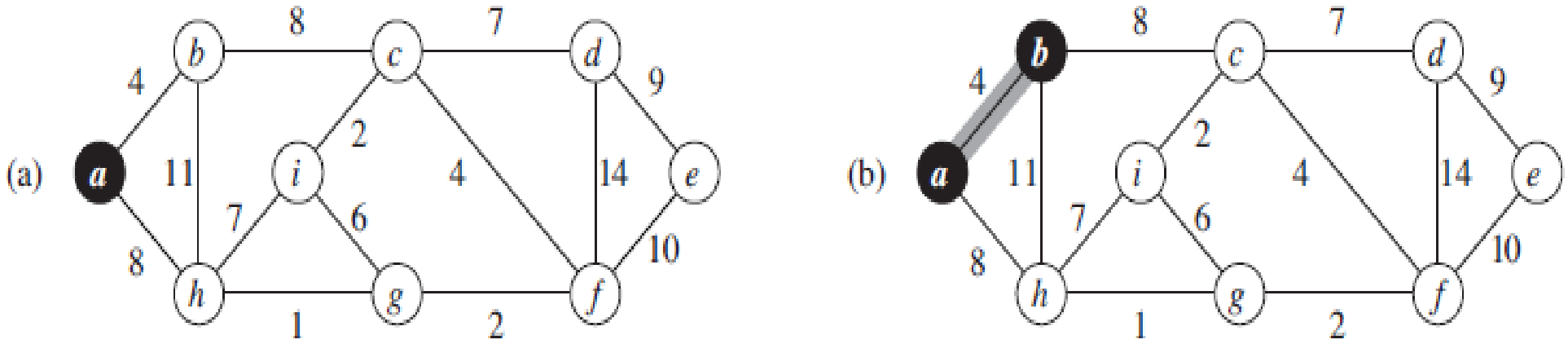
Time complexity of this algorithm is O(E lg E) or O(E lgV).

Prim's Algorithm

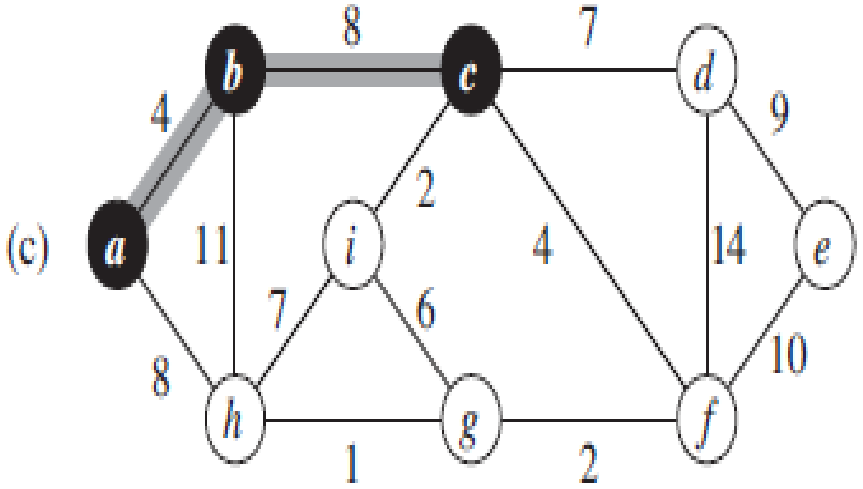
Example: Find the minimum spanning of the following graph using Prim's algorithm.

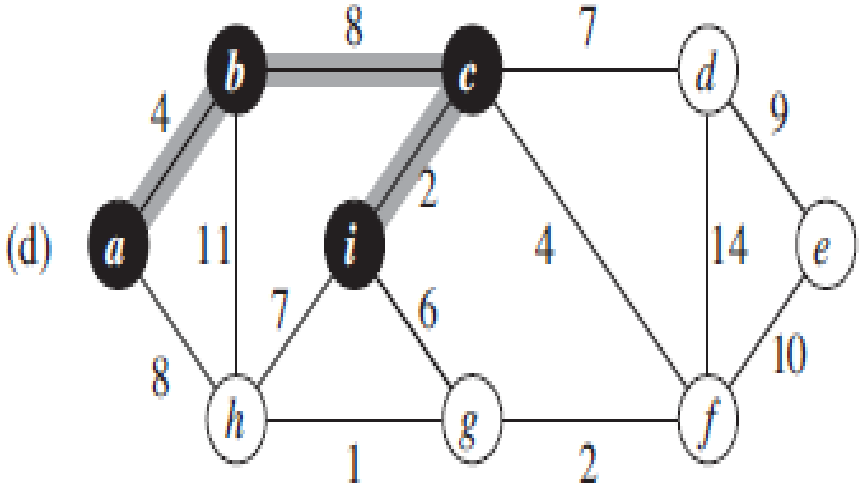


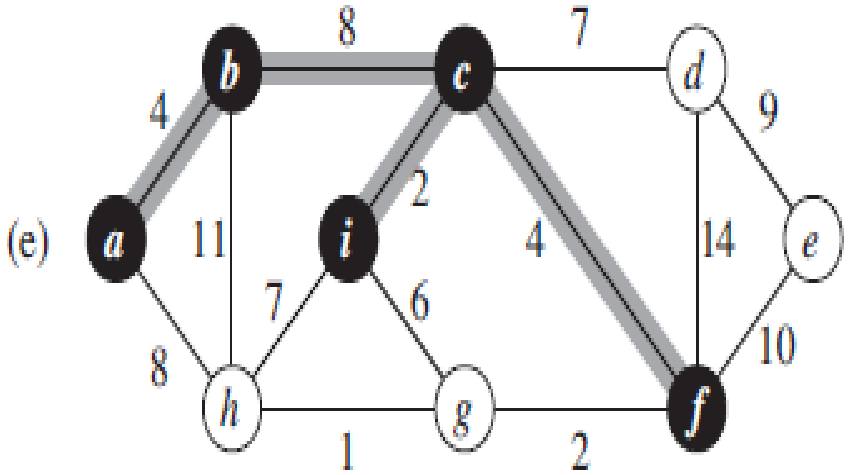
Solution:

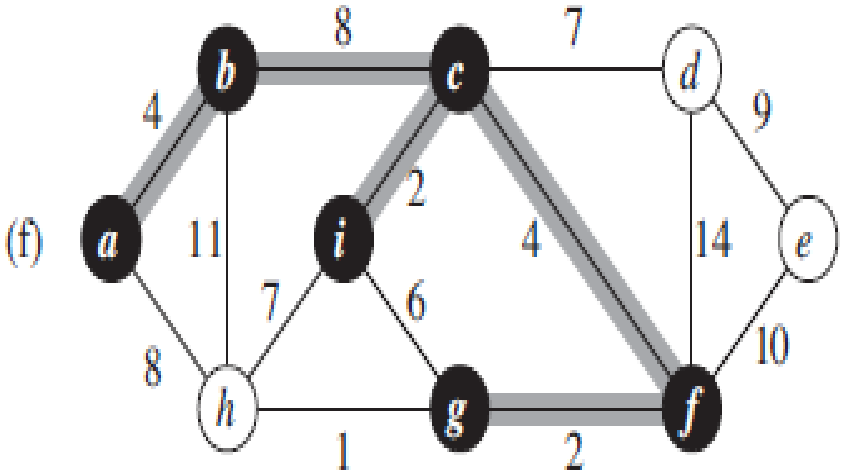


Prim's Algorithm

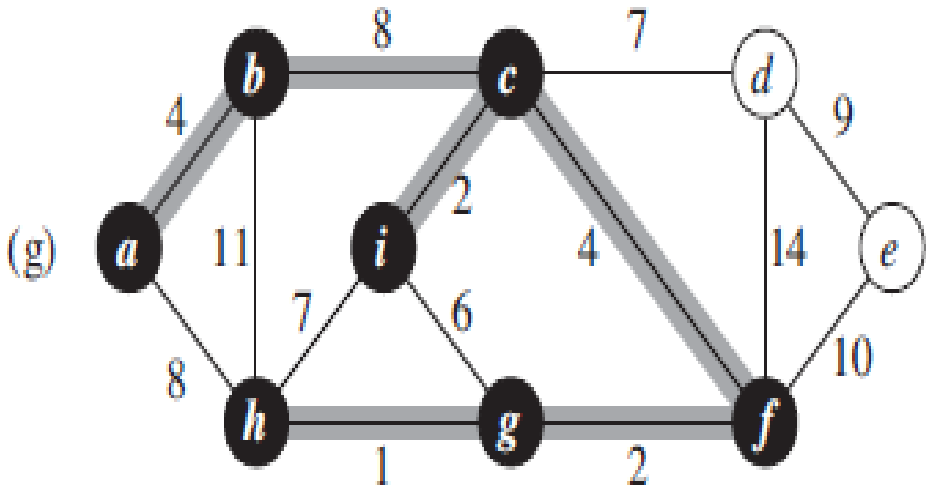


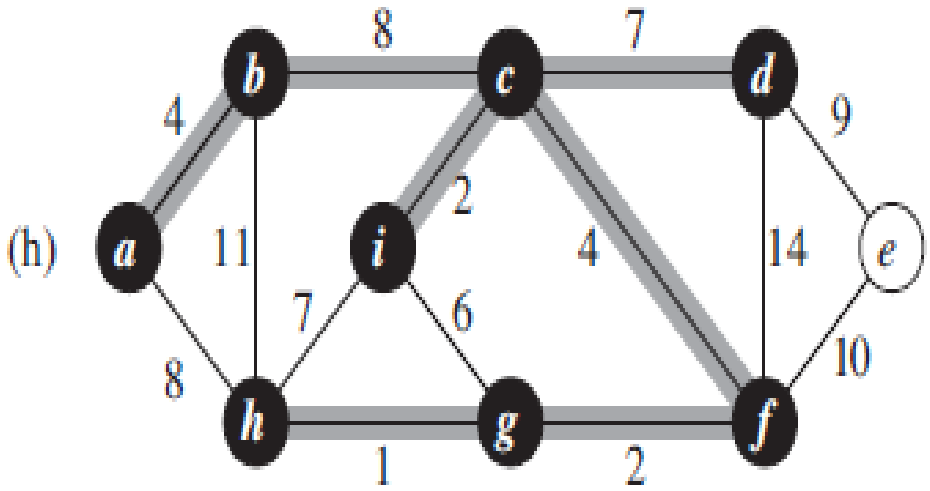


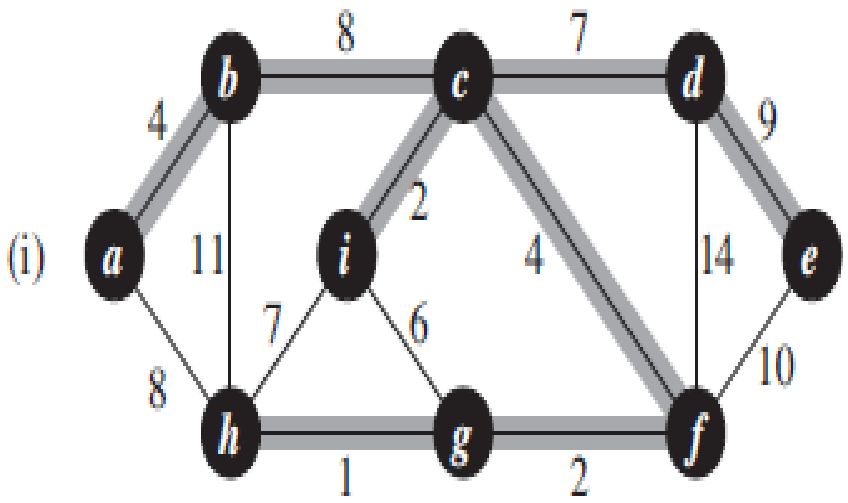




Prim's Algorithm







Total cost = 37

Final minimum spanning tree

Prim's Algorithm

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Time complexity of this algorithm is O(E lgV + V lgV) = O(E lgV).

Shortest path algorithm

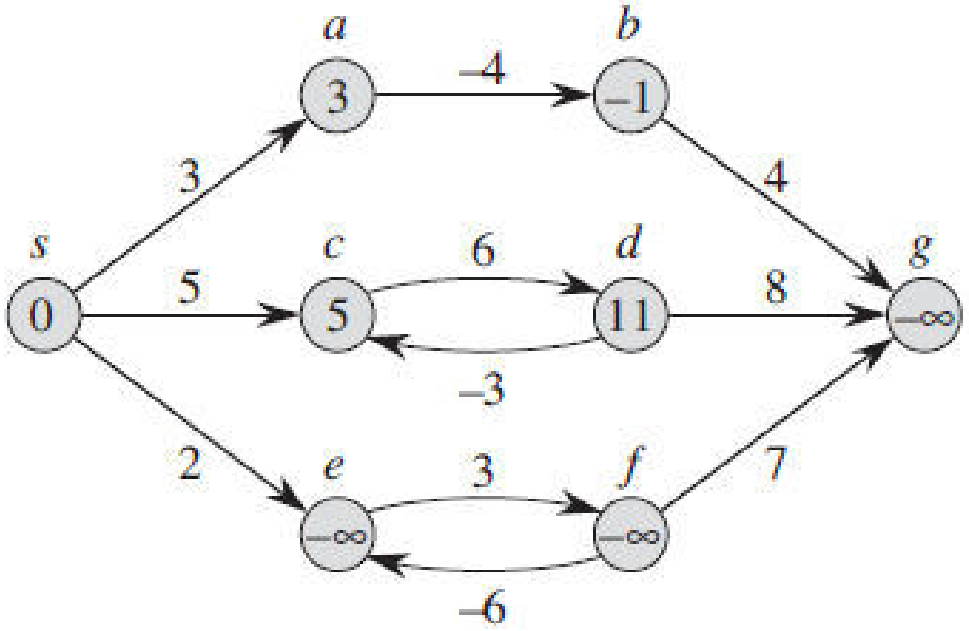
Single-source shortest-paths problem

Given a graph G = (V, E), we want to find a shortest path from a given source vertex s ϵ V to each vertex v ϵ V .

Negative weight cycles

A cycle in the graph is said to be negative weight cycle if the sum of weights of all the edges in the cycle is negative value.

Example: Consider the following graph.



Shortest path algorithm

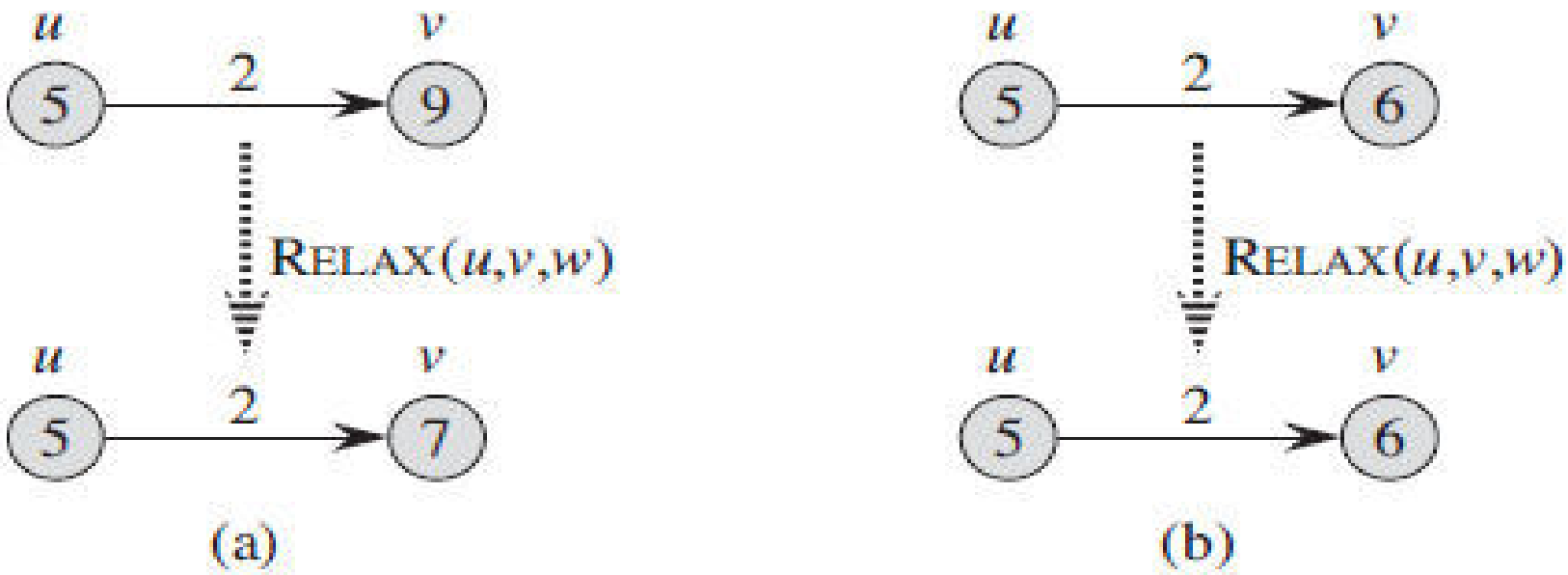
Initialize the single source function

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

s is the source vertex.
v.d is the distance of vertex v from s.
v.π is the predecessor vertex of v.

Shortest path algorithm

Relaxation function




```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Shortest path algorithm

Here, we shall study two single source shortest path algorithms. These algorithms are the following:-

1. Bellman-Ford algorithm
2. Dijkastra algrithm

Both algorithms are based on greedy approach.

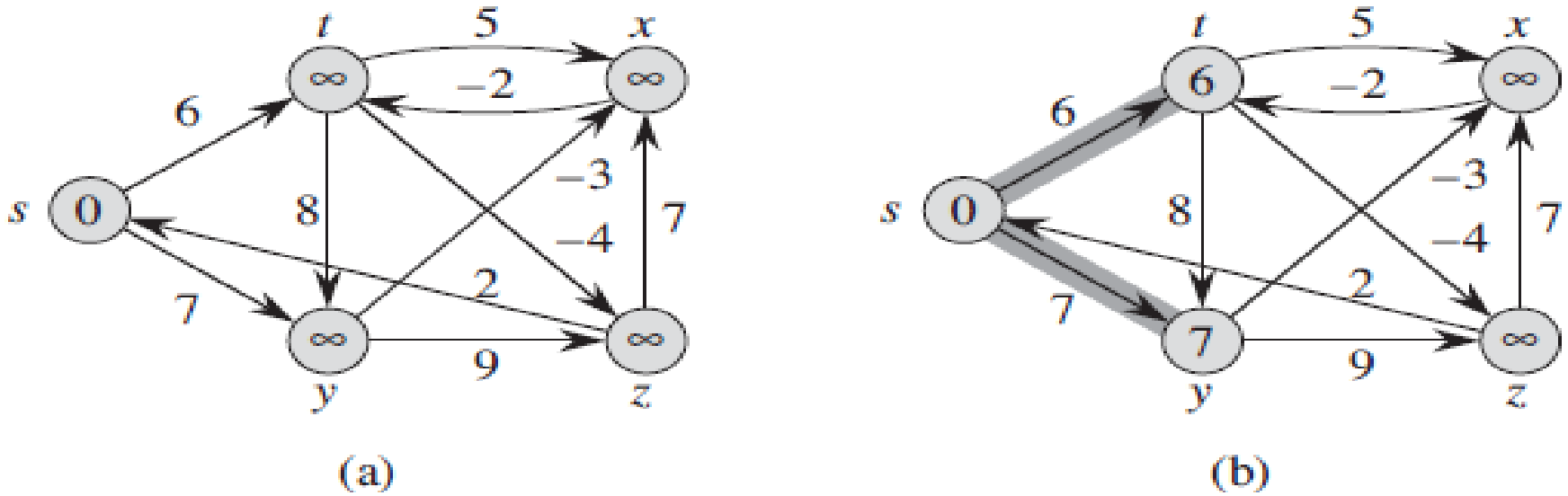
Bellman-Ford algorithm

The **Bellman-Ford algorithm** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

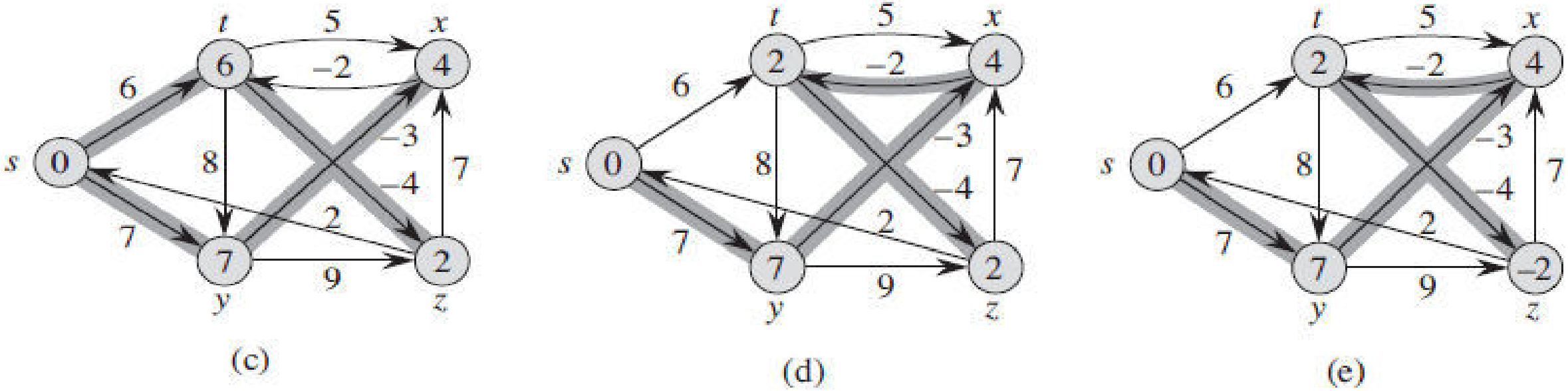
Bellman-Ford algorithm

Example:



The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then: v.**π** = u. In this particular example, each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).

Bellman-Ford algorithm



Order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y).

Bellman-Ford algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Time complexity of this algorithm is O(VE).

Dijkstra's algorithm

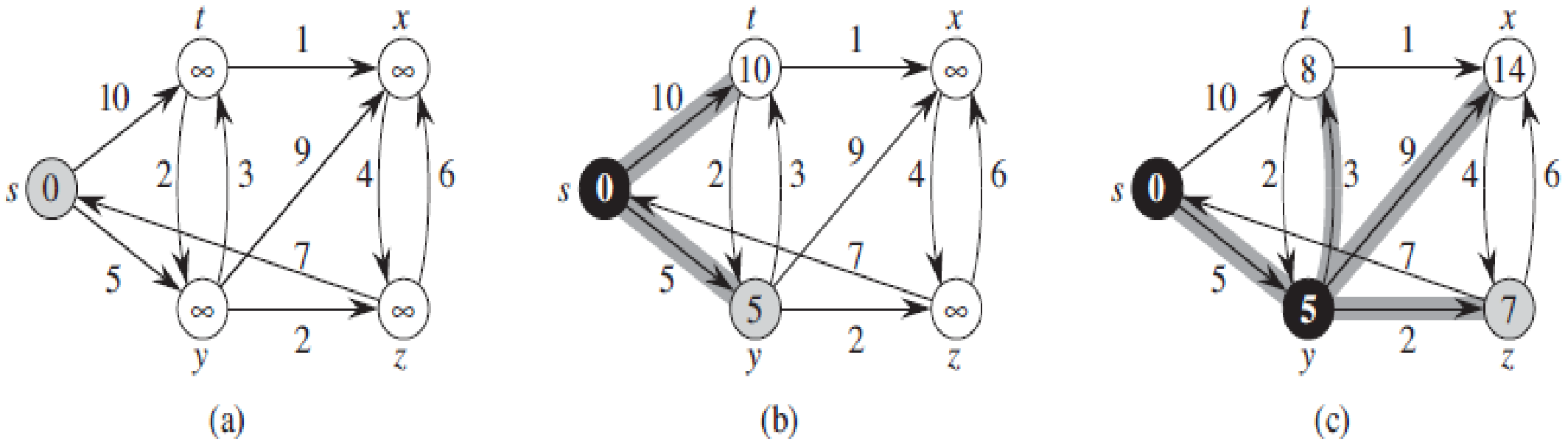
- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.

- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V\text{-}S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

- In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

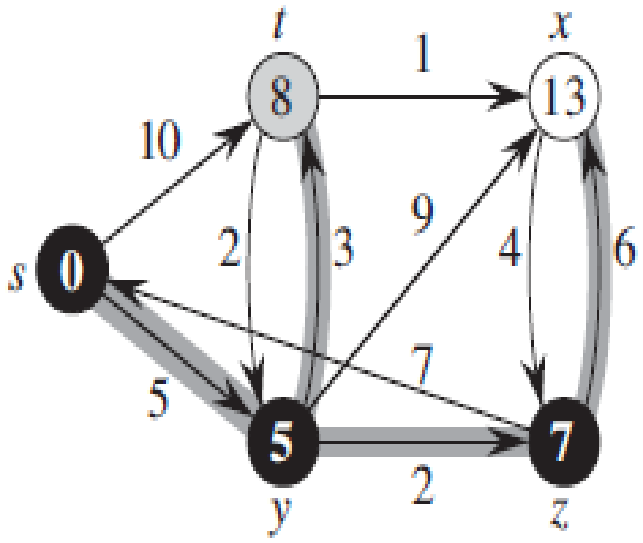
Dijkstra's algorithm

Example:

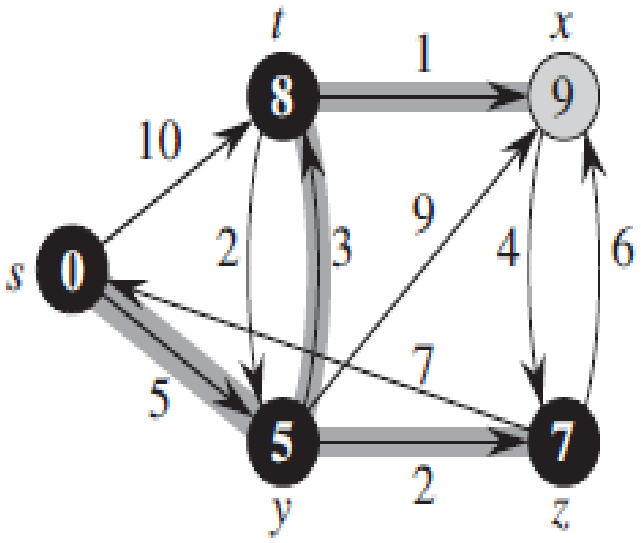


The execution of Dijkstra's algorithm. s is the source vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue Q = V - S.

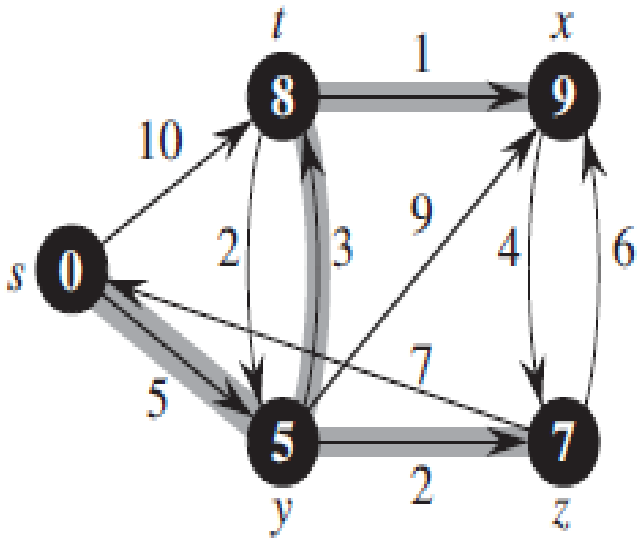
Dijkstra's algorithm



(d)



(e)



(f)

Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Time complexity of this algorithm is O(E logV).

AKTU Examination Questions

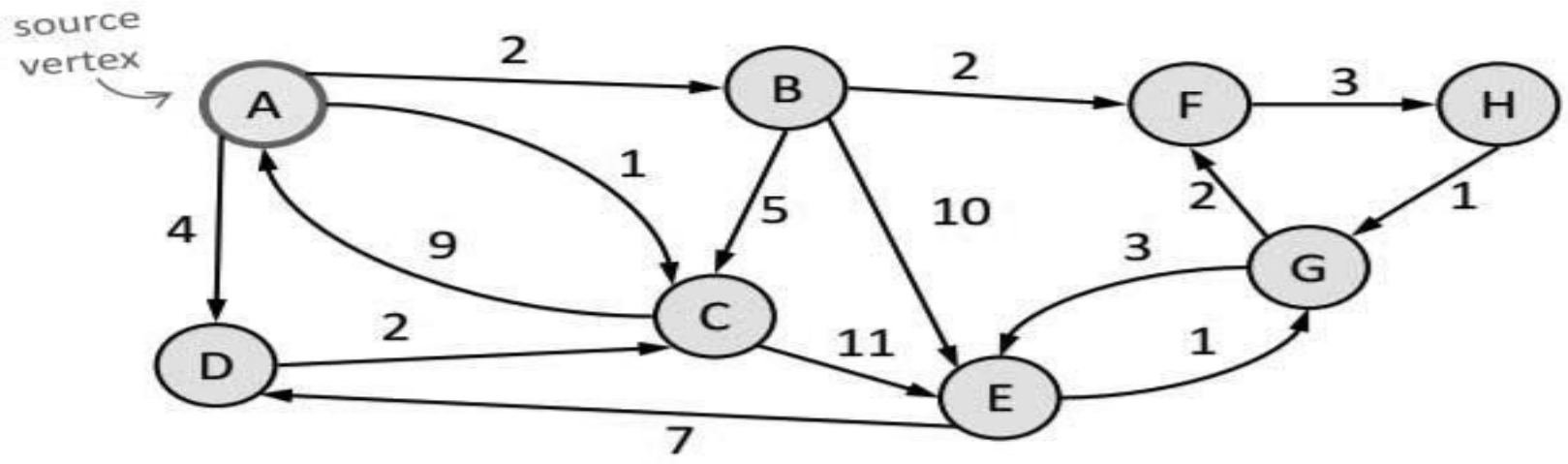
1. Consider the following instance for knapsack problem. Find the solution using Greedy method:

N= 10, W=130

P [] = {21, 31, 43, 53, 41, 63, 65, 75}

V [] = {11, 21, 31, 33, 43, 53, 65, 65}

2. Apply the greedy single source shortest path algorithm on the following graph:



AKTU Examination Questions

3. Describe Activity selection problem.

4. Write an algorithm for minimum spanning tree with example.

5. What are greedy algorithms? Explain their characteristics?

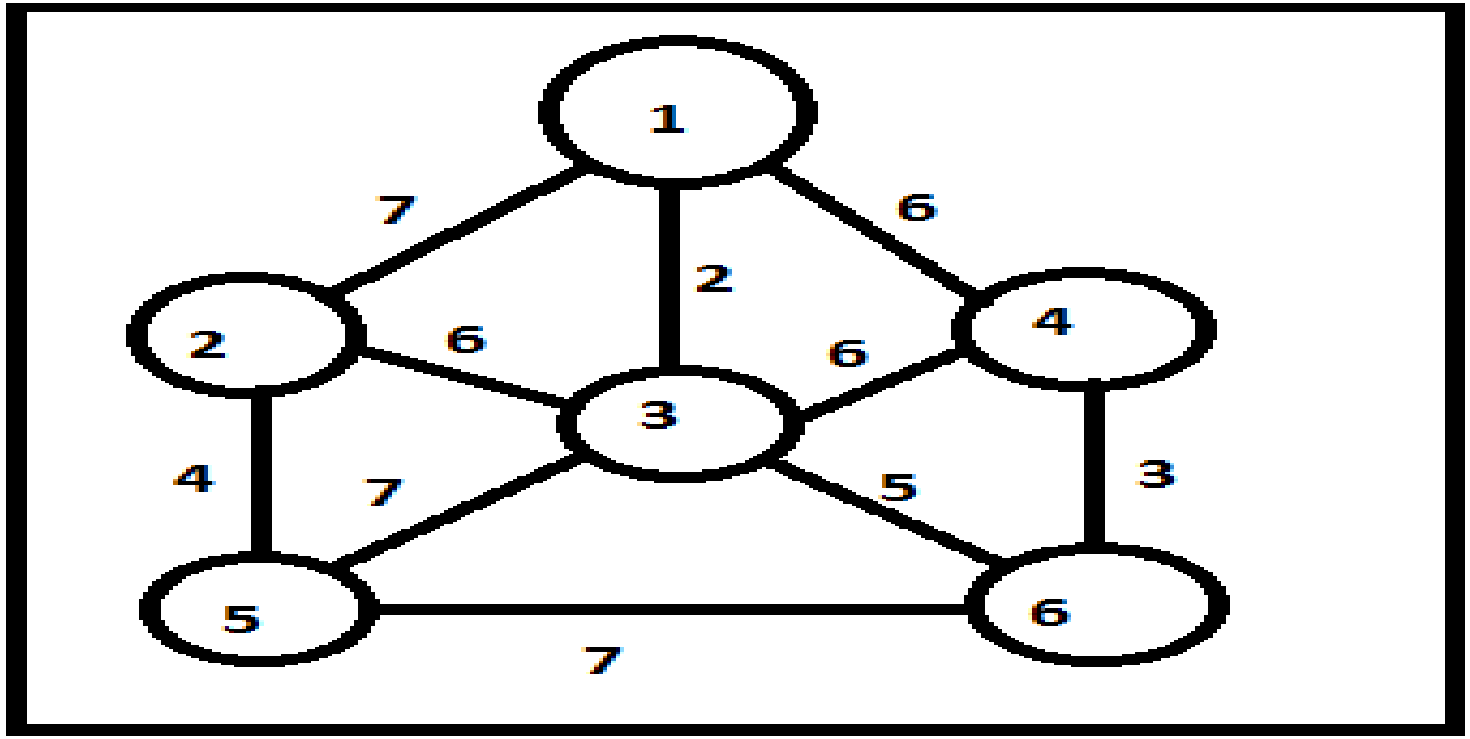
6. Define feasible and optimal solution.

7. Define spanning tree. Write Kruskal's algorithm for finding minimum cost spanning tree. Describe how Kruskal's algorithm is different from Prim's algorithm for finding minimum cost spanning tree.

8. Explain Single source shortest path.

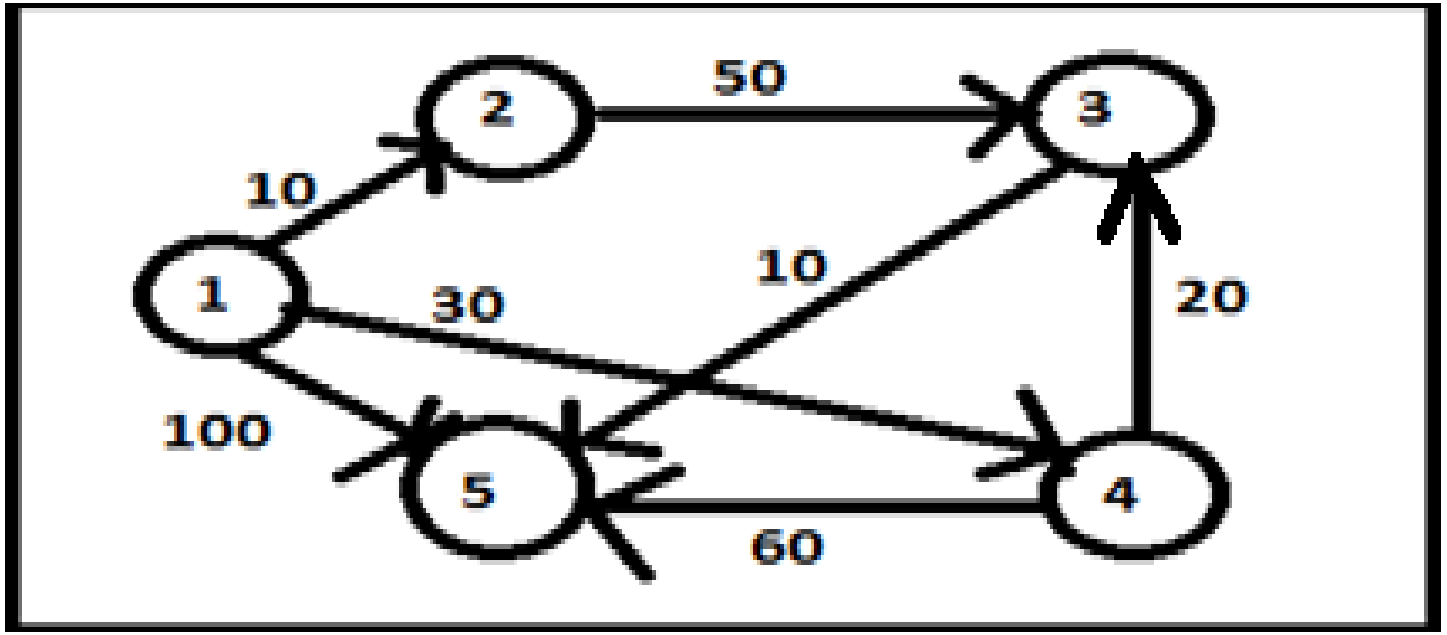
AKTU Examination Questions

9. What is Minimum Cost Spanning Tree? Explain Kruskal's Algorithm and Find MST of the Graph. Also write its Time-Complexity.



AKTU Examination Questions

10. Find the shortest path in the below graph from the source vertex 1 to all other vertices by using Dijkstra's algorithm.



AKTU Examination Questions

11. Given the six items in the table below and a Knapsack with Weight 100, what is the solution to the Knapsack problem in all concepts. I.e. explain greedy all approaches and find the optimal solution.

ITEM ID	WEIGHT	VALUE	VALUE/WEIGHT
A	100	40	.4
B	50	35	.7
C	40	20	.5
D	20	4	.2
E	10	10	1
F	10	6	.6

12. Prove that if the weights on the edge of the connected undirected graph are distinct then there is a unique Minimum Spanning Tree. Give an example in this regard. Also discuss Prim's Minimum Spanning Tree Algorithm in detail.

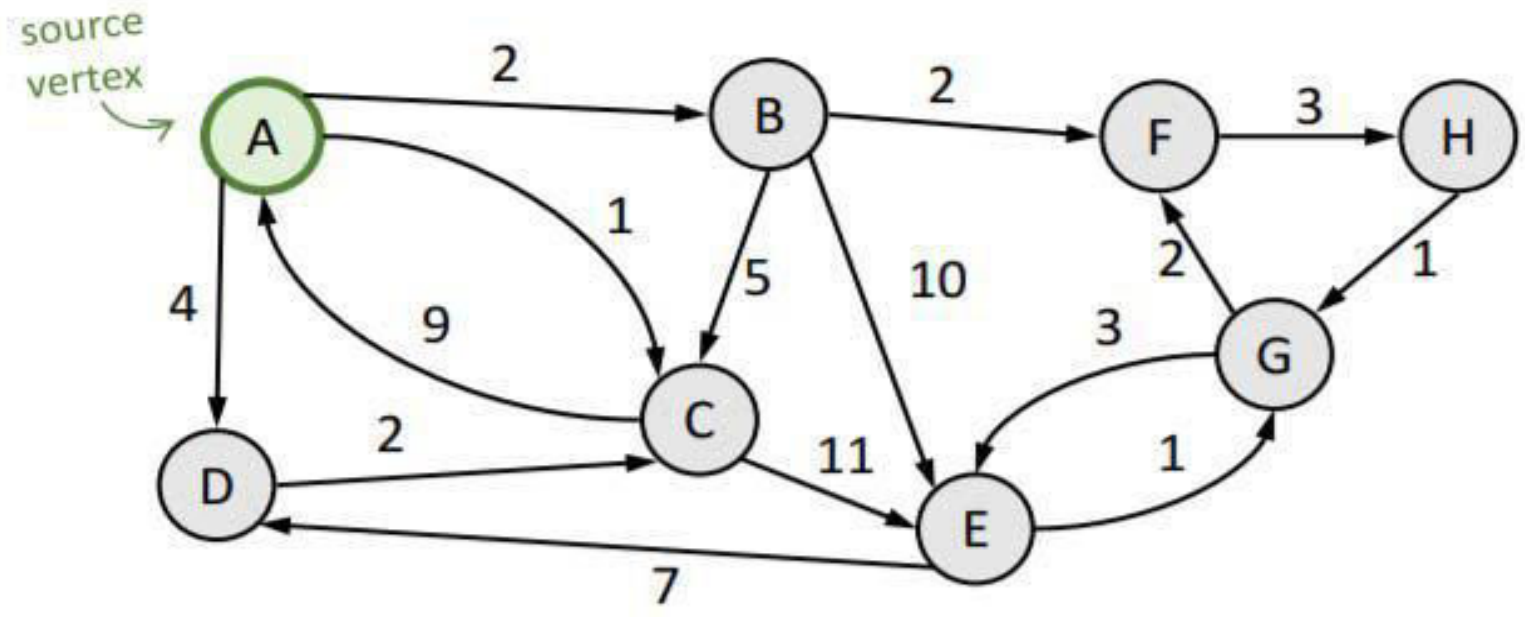
AKTU Examination Questions

13. Consider the weights and values of items listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 Kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt}. A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy}. Find the value of $V_{opt} - V_{greedy}$.

Item	I_1	I_2	I_3	I_4
W	10	7	4	2
V	60	28	20	24

AKTU Examination Questions

14. When do Dijkstra and the Bellman-Ford algorithm both fail to find a shortest path? Can Bellman ford detect all negative weight cycles in a graph? Apply Bellman Ford Algorithm on the following graph:



Divide and Conquer

Divide and Conquer Approach

- The divide-and-conquer paradigm involves three steps at each level of the recursion:
- **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
- **Combine** the solutions to the sub-problems into the solution for the original problem.

Divide and Conquer Approach

We will solve the following problems using divide and conquer approach :-

- Sorting
  - Merge sort
  - Quick sort
- Searching
  - Binary search
- Matrix Multiplication
- Convex Hull

Binary search

- **Binary search** is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems.

- Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted.

- When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.

Binary search process

Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2nd half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 > 56 take 1st half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91

Binary search algorithm

```
Binary-search(A, n, x)
l = 1
r = n
while l ≤ r
do
        m = ⌊(l + r) / 2⌋
        if A[m] < x then
                l = m + 1
        else if A[m] > x then
                r = m − 1
        else
                return m
return unsuccessful
```

Time complexity $\mathbf{T(n) = O(lgn)}$

Matrix Multiplication (Divide and Conquer Method)

To multiply two matrices A and B of order nxn using **Divide and Conquer approach**, we use to multiply two matrices of order 2x2.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Where, A_{ij} and B_{ij} are $\frac{n}{2} \text{x} \frac{n}{2}$ matrices for i,j = 1,2.

Resultant matrix C will be

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Where,

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$ $\qquad$ $C_{12} = A_{11}B_{21} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$ $\qquad$ $C_{22} = A_{21}B_{21} + A_{22}B_{22}$

Matrix Multiplication

Clearly, computation of c_{ij} consists of two multiplications of two $\frac{n}{2} \times \frac{n}{2}$ matrices and one addition of two $\frac{n}{2} \times \frac{n}{2}$ matrices. Therefore, the algorithms for this is the following:-

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

Matrix Multiplication

Time complexity of this algorithm is computed as following:-

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 8T(n/2) + \theta(n^2) & \text{if } n > 1 \end{cases}$$

After solving this recurrence relation, we get

$$\mathbf{T(n) = \theta(n^3)}$$

Strassen's Matrix Multiplication Algorithm

Strassen's algorithm has three steps:

1) Divide the input matrices A and B into $\frac{n}{2} \text{x} \frac{n}{2}$ sub-matrices.
2) Using the sub-matrices created from the step above, recursively compute seven matrix products P_1, P_2, … P_7. Each matrix P_i is of size $\frac{n}{2} \text{x} \frac{n}{2}$.

   $P1 = A_{11}(B_{12}-B_{22})$ $\quad$ $P2 = (A_{11}+A_{12})B_{22}$

   $P3 = (A_{21}+A_{22})B_{11}$ $\quad$ $P4 = A_{22}(B_{21}-B_{11})$

   $P5 = (A_{11}+A_{22})(B_{11}+B_{22})$ $\quad$ $P6 = (A_{12}-A_{22})(B_{21}+B_{22})$

   $P7 = (A_{11}-A_{21})(B_{11}+B_{12})$
3) Get the desired sub-matrices C_{11}, C_{12}, C_{21}, and C_{22} of the result matrix C by adding and subtracting various combinations of the P_i sub-matrices.

$C_{11} = P_5+P_4-P_2+P_6$ $\quad$ $C_{12}= P_1+P_2$

$C_{21}= P_3+P_4$ $\quad$ $C_{22}= P_1+P_5-P_3-P_7$

Strassen's Matrix Multiplication Algorithm

Strassen_Matrix_Multiplication(A, B, n)

If n=1 then return AxB.

Else

1. Compute A_{11}, B_{11}, ……, A_{22}, B_{22}.
2. $P_1 \leftarrow$ Strassen_Matrix_Multiplication(A_{11}, B_{12}-B_{22}, n/2)
3. $P_2 \leftarrow$ Strassen_Matrix_Multiplication(A_{11}+A_{12}, B_{22}, n/2)
4. $P_3 \leftarrow$ Strassen_Matrix_Multiplication(A_{21}+A_{22}, B_{11}, n/2)
5. $P_4 \leftarrow$ Strassen_Matrix_Multiplication(A_{22}, B_{21}-B_{11}, n/2)
6. $P_5 \leftarrow$ Strassen_Matrix_Multiplication(A_{11}+A_{22}, B_{11}+B_{22}, n/2)
7. $P_6 \leftarrow$ Strassen_Matrix_Multiplication(A_{12}-A_{22}, B_{21}+B_{22}, n/2)
8. $P_7 \leftarrow$ Strassen_Matrix_Multiplication(A_{11}-A_{21}, B_{11}+B_{12}, n/2)
9. $C_{11} = P_5+P_4-P_2+P_6$
10. $C_{12} = P_1+P_2$
11. $C_{21} = P_3+P_4$
12. $C_{22} = P_1+P_5-P_3-P_7$
13. return C

End if

Strassen's Matrix Multiplication Algorithm

Time complexity of this algorithm is computed as following:-

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 7T(n/2) + \theta(n^2) & \text{if } n > 1 \end{cases}$$

After solving this recurrence relation, we get

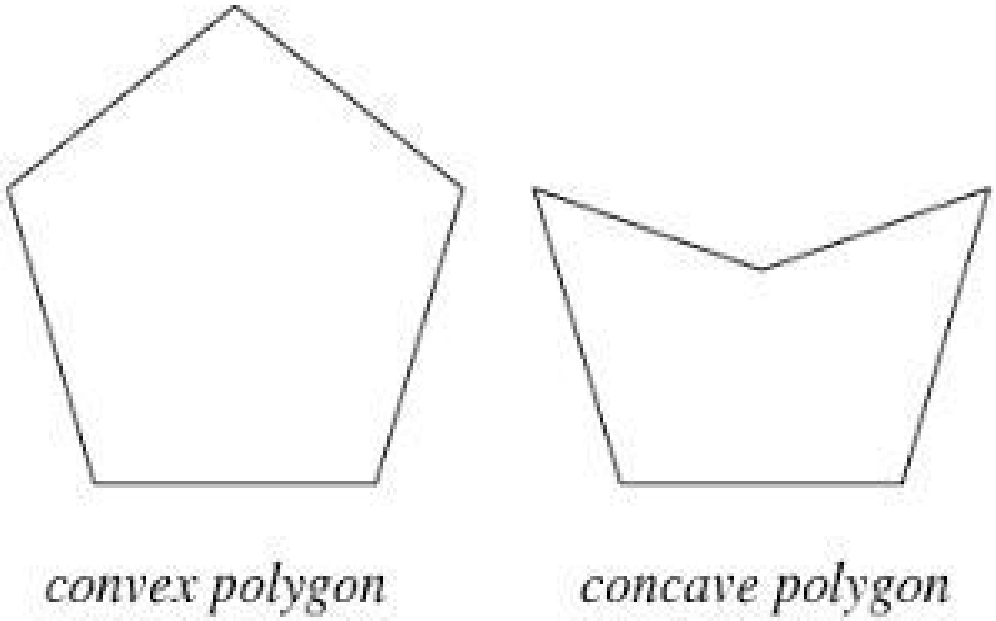
$$\mathbf{T(n) = \theta(n^{2.8})}$$

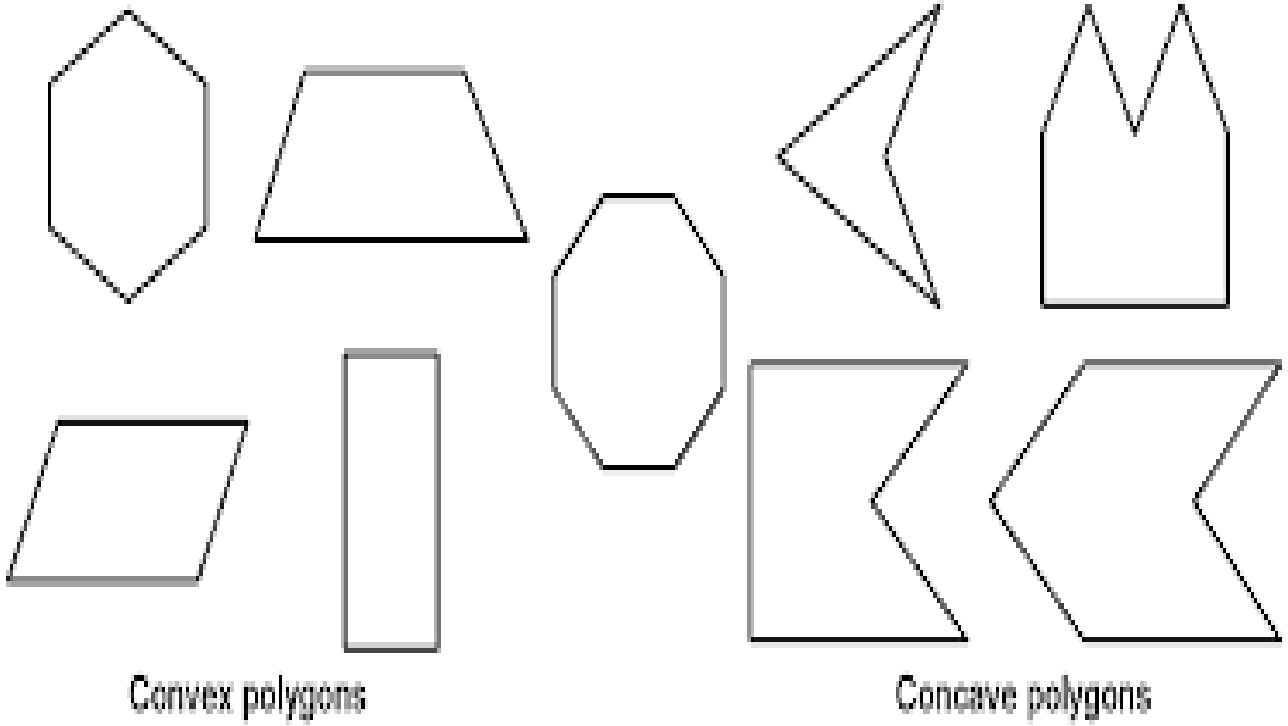
Convex Hull

Convex Polygon:

A polygon P is said to be convex polygon if for any two points p and q on the boundary of P, segment pq lies entirely inside P.

Example:

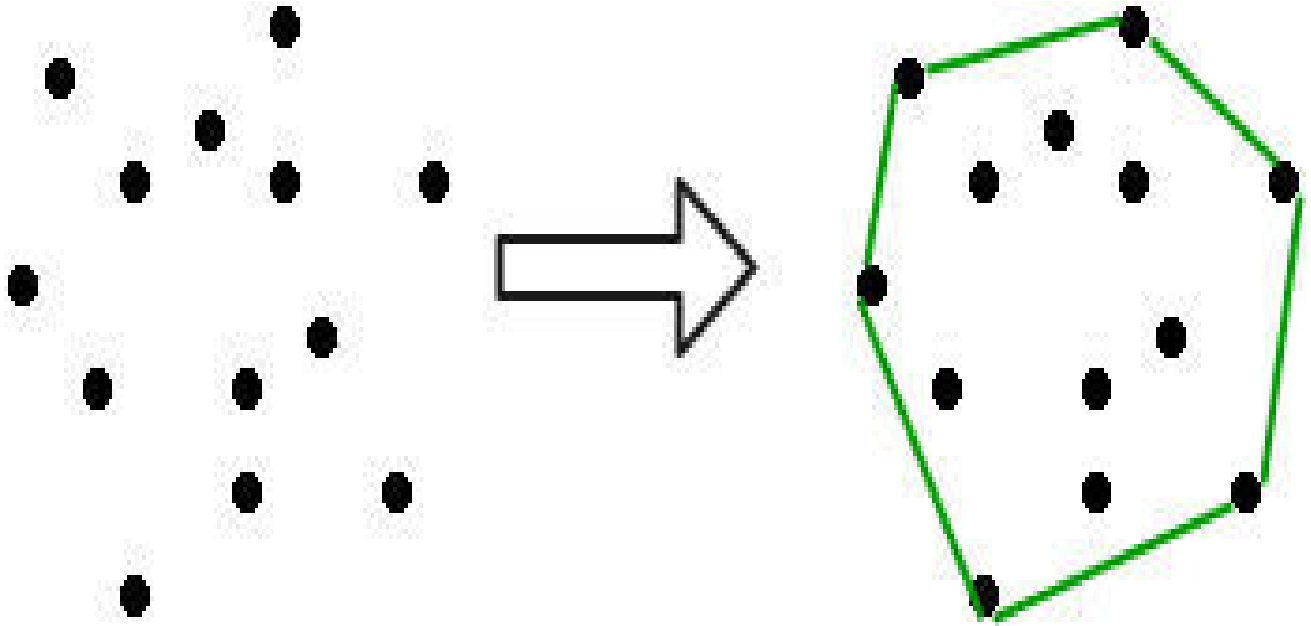




Convex Hull

Convex hull: Convex hull of a set Q of points is the smallest convex polygon P for which each points in Q is either on the boundary of P or in its interior.

Example:



Convex hull

Convex Hull

There are many algorithms for computing convex hull. But here, we shall study only two algorithms.

1. QuickHull
2. Divide and Conquer

QuickHull Algorithm

- To compute the convex hull of a set X of n points in the plane, an algorithm is designed. This algorithm, called QuickHull, first identifies the two points (call them p_1 and p_2) of X with the smallest and largest x-coordinate values. Assume now that there are no ties. Both p_1 and p_2 are extreme points and part of the convex hull.

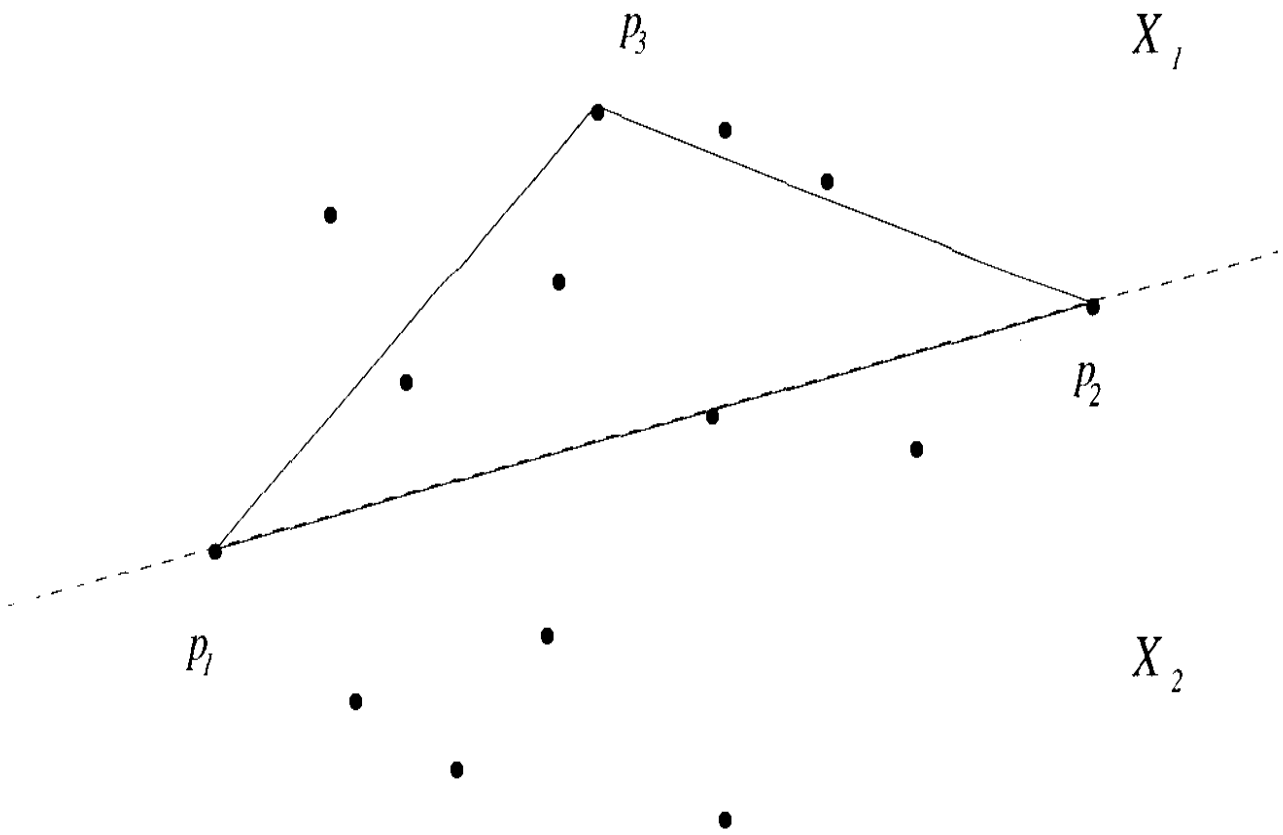
- The set X is divided into X_1 and X_2 so that X_1 has all the points to the left of the line segment (p_1, p_2) and X_2 has all the points to the right of (p_1,p_2). Both X_1 and X_2 include the two points p_1 and p_2. Then, the convex hulls of X_1 and X_2 (called the upper hull and lower hull, respectively) are computed using a divide-and-conquer algorithm called QuickHull. The union of these two convex hulls is the overall convex hull.

QuickHull Algorithm

Computation of the convex hull of X_1

- We determine a point of X_1 that belongs to the convex hull of X_1 and use it to partition the problem into two independent subproblems.
- Such a point is obtained by computing the area formed by p_1 , p , and p_2 for each p in X_1 and picking the one with the largest (absolute) area.
- Ties are broken by picking the point p for which the angle pp_1p_2 is maximum. Let p_3 be that point.
- Now X_1 is divided into two parts; the first part contains all the points of X_1 that are to the left of (p_1,p_3) (including p_1 and p_3), and the second part contains all the points of X_1 that are to the left of (p_3, p_2) (including p_3 and p_2).
- All the other points are interior points and can be dropped from future consideration.
- The convex hull of each part is computed recursively, and the two convex hulls are merged easily by placing one next to the other in the right order.

QuickHull Algorithm



Running time of algorithm

$T(n) = O(n\log n)$

Where, n is the number of points in the set.

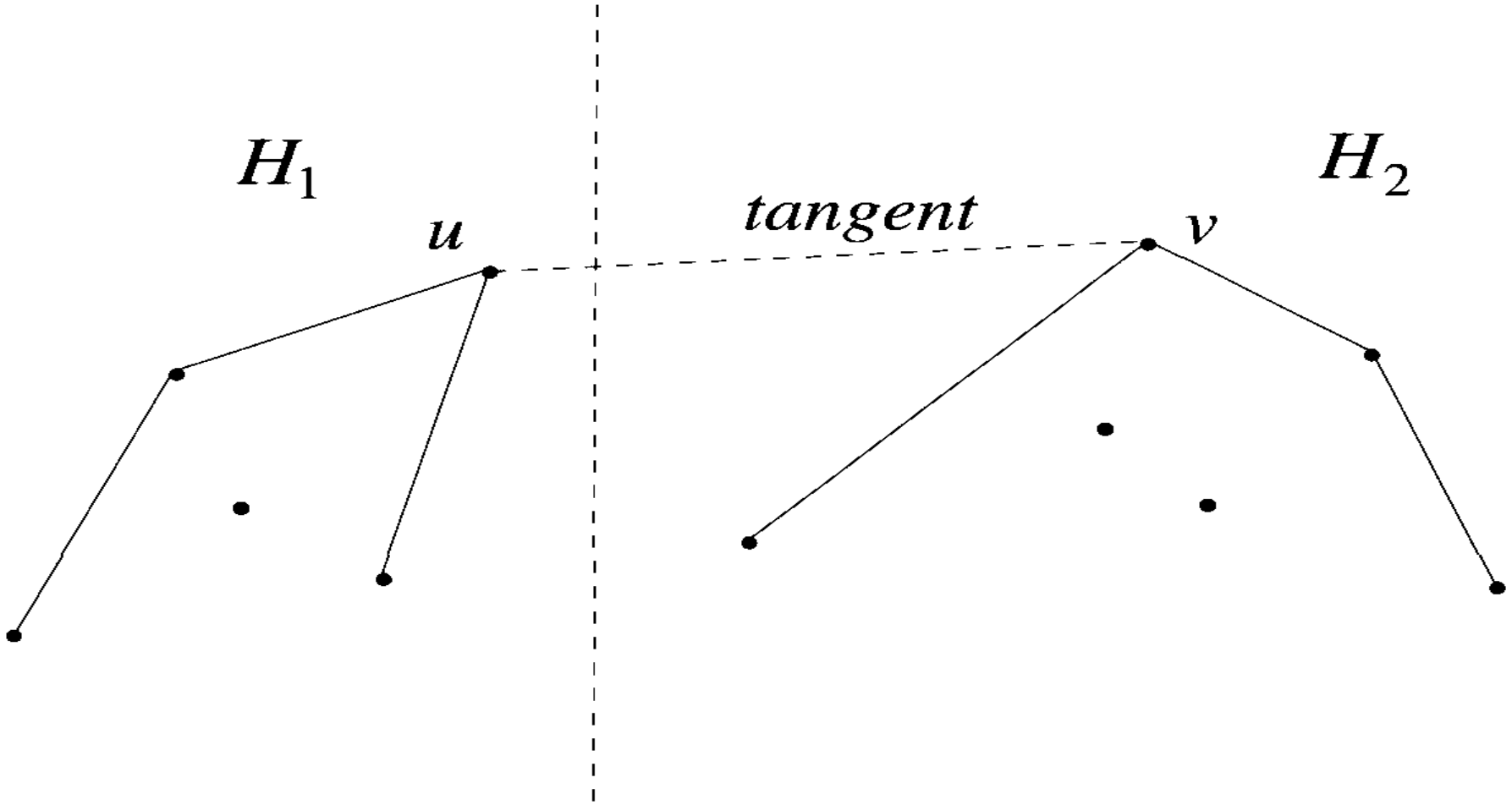
Identifying a point on the convex hull of X_1

Divide and Conquer Algorithm

This algorithm is called DCHull. This algorithm also takes O(nlogn) time. It computes the convex hull in clockwise order.

- Given a set of n points, the problem is reduced to finding the upper hull and the lower hull separately and then putting them together. Since the computations of the upper and lower hulls are very similar, we restrict our discussion to computing the upper hull.
- The divide-and-conquer algorithm for computing the upper hull partitions X into two nearly equal halves. Partitioning is done according to the x-coordinate values of points using the median x-coordinate as the splitter.
- Upper hulls are recursively computed for the two halves. These two hulls are then merged by finding the line of tangent (i.e., a straight line connecting a point each from the two halves, such that all the points of X are on one side of the line).

Divide and Conquer Algorithm



Divide and Conquer Algorithm

- To begin with, the points p_1 and p_2 are identified [where p_1 (p_2) is the point with the least (largest) x-coordinate value].
- All the points that are to the left of the line segment (p_1, p_2) are separated from those that are to the right.
- Sort the input points according to their x-coordinate values.
- Let $q_1, q_2, .. \ldots, q_N$ be the sorted order of these points. Now partition the input into two equal halves with $q_1, q_2, .. \ldots, q_{N/2}$ in the first half and $q_{N/2+1}, q_{N/2+2}, .. \ldots, q_N$ in the second half.
- The upper hull of each half is computed recursively. Let H_1 and H_2 be the upper hulls. Upper hulls are maintained as linked lists in clockwise order.

Divide and Conquer Algorithm

- The line of tangent is then found in $0(\log^2 N)$ time. If (u, v) is the line of tangent, then all the points of H_1 that are to the right of u are dropped.
- Similarly, all the points that are to the left of v in H_2 are dropped. The
- Remaining part of H_1, the line of tangent, and the remaining part of H_2 form the upper hull of the given input set.

- If T(N) is the run time of the above recursive algorithm for the upper hull on an input of N points, then we have

$$T(N) = 2T(N/2) + \log^2 N$$

The solution of this recurrence relation is $T(N) = O(N)$.

- But, the sorting of points takes O(N logN) time, therefore the running time of whole algorithm is $O(N \log N)$.

AKTU Examination Questions

1. Describe convex hull problem.
2. What do you mean by convex hull? Describe an algorithm that solves the convex hull problem. Find the time complexity of the algorithm.
3. Given an integer x and a positive number n, use divide & conquer approach to write a function that computes x^n with time complexity O (log n).

Design and Analysis of Algorithm
Unit-4

Dynamic Programming

Dynamic programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

- In contrast to the divide-and-conquer method, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.

- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table.

Optimization problems

- We typically apply dynamic programming to ***optimization problems***.

- Such problems can have many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- We call such a solution *an* optimal solution to the problem.

Dynamic programming

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1) Characterize the structure of an optimal solution.
2) Recursively define the value of an optimal solution.
3) Compute the value of an optimal solution, typically in a bottom-up fashion.
4) Construct an optimal solution from computed information.

Dynamic programming

Using dynamic programming, we shall solve the following problems:-

- Matrix-chain multiplication
- Longest common subsequence problem
- 0-1 Knapsack problem
- All pairs shortest path problem

Matrix-chain multiplication problem

This problem is stated as following:-

Given a sequence (chain) < A_1, A_2, ..., A_n > of n matrices where for i = 1, 2, 3, ..., n, matrix A_i has dimension p_{i-1} x p_i, fully parenthesize the product A_1A_2 ..... A_n in a way that minimizes the number of scalar multiplications.

Example: Find all parenthesization of matrix for n=4.

Solution:

1) ((A1A2)(A3A4))
2) (((A1A2)A3)A4)
3) (A1(A2(A3A4)))
4) ((A1(A2A3))A4)
5) (A1((A2A3)A4))

Matrix-chain multiplication problem

Example: Consider the following chain of matrices < A1, A2, A3 > . The dimension of matrices are the following:-

A1 = 10x100, A2 = 100x5, A3= 5x50

Find the optimal parenthesization of these matrices.

Solution: All the parenthesization of these matrices are:-

1) ((A1A2)A3)
2) (A1(A2A3))

Number of scalar multiplications in (1)

= 10*100*5 + 10*5*50

= 5000 + 2500 = 7500

Number of scalar multiplications in (2)

= 100*5*50 + 10*100*50

= 25000 + 50000 = 75000

Therefore, the solution ((A1A2)A3) is optimal solution.

Number of parenthesizations

Let P(n) denote the number of parenthesizations for n matrices. It is computed as following:-

P(n) = 1 if n=1

$= \sum_{k=1}^{n-1} P(k)P(n-k)$ if n >=2

Dynamic programming approach for Matrix-chain multiplication problem

Step 1: In this step, we find the optimal substructure and then use it to construct an optimal solution to the problem.

Let $A_{i..j}$ → Matrix that results from evaluating the product $A_iA_{i+1}.....A_j$. Where $i \leq j$.

- If $i < j$, then to parenthesize the product $A_iA_{i+1}.....A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$.
- The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

Matrix-chain multiplication problem

Step 2:

- Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$.
- m[i,j] is recursively defined as following:-

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\,. \end{cases}$$

- We define s[i, j] to be the value of k at which m[i, k] + m[k+1, j] + $p_{i-1}p_kp_j$ is minimum.
- We compute the optimal solution using matrix s.
- And matrix m is used to compute the value of optimal solution.

Matrix-chain multiplication problem

Step 3: In this step, we compute the matrices m and s.

Following algorithm is used to compute the matrices m and s. This algorithm assumes that matrix A_i has dimensions $p_{i-1} x p_i$ for i = 1, 2, ..., n. Its input is a sequence p = < p_0, p_1, ..., p_n >, where p.*length* = n+1.

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Matrix-chain multiplication problem

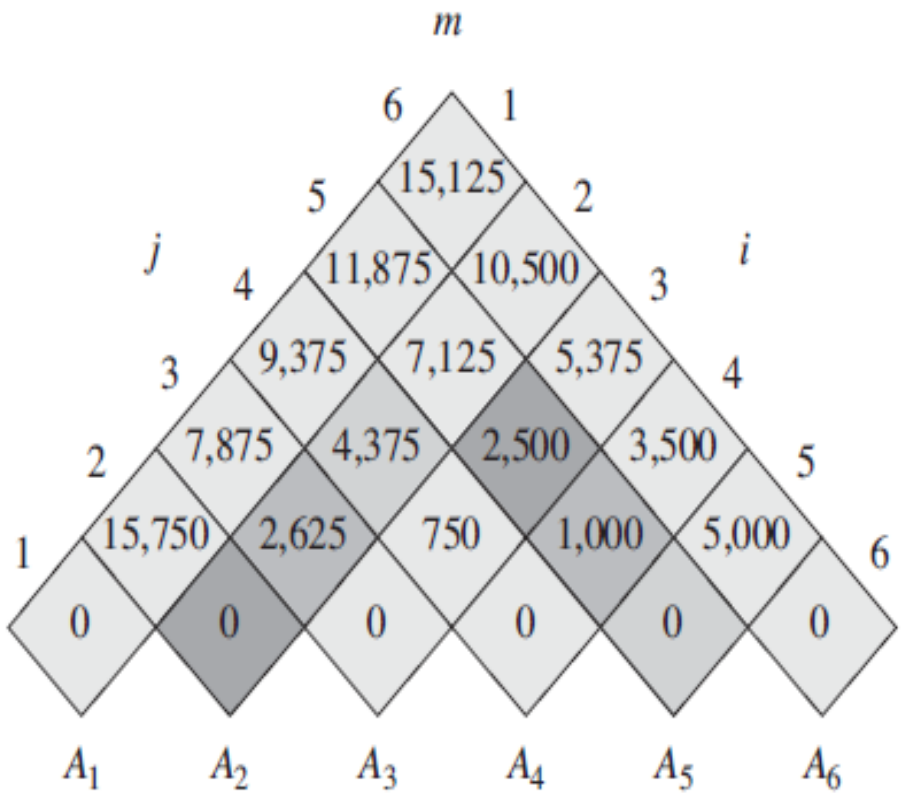
Example: Consider the following matrix chain multiplication problem.

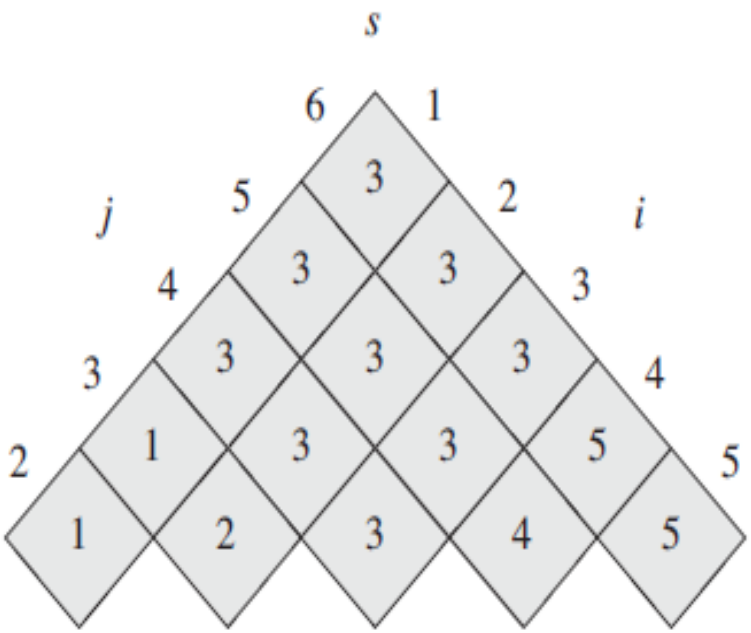
$A_1 \rightarrow$ 30x35, $A_2 \rightarrow$ 35x15, $A_3 \rightarrow$ 15x5,

$A_4 \rightarrow$ 5x10, $A_5 \rightarrow$ 10x20, $A_6 \rightarrow$ 20x25.

Find the optimal solution and its value.

Solution:





Matrix-chain multiplication problem

Step 4: Constructing an optimal solution

Following algorithm is used to create optimal solution i.e. it finds optimal parenthesization.

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

- The initial call PRINT-OPTIMAL-PARENS(s, 1, n) prints an optimal parenthesization of $< A_1, A_2, \ldots\ldots, A_n >$.

Time complexity of this algorithm is O(n).

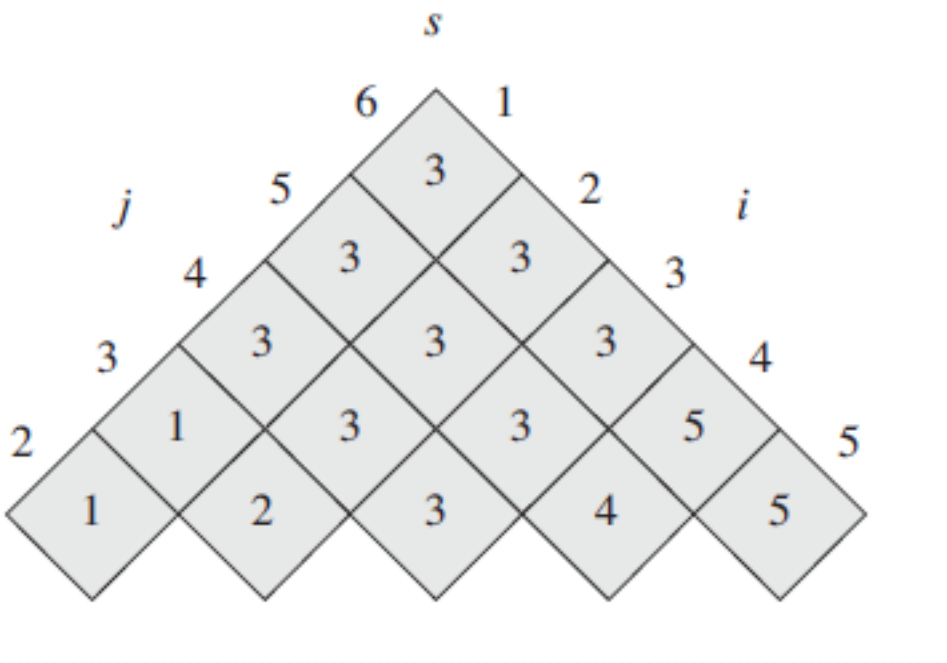
Matrix-chain multiplication problem

Step 4: Constructing an optimal solution for previous example

The initial call PRINT-OPTIMAL-PARENS(s, 1, 6) prints an optimal parenthesization .

Optimal solution will be

((A1(A2A3))((A4A5)A6))



Longest Common Subsequence Problem (LCS problem)

In the ***longest-common-subsequence problem***, we are given two sequences $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$ and $Y = < y_1, y_2, \ldots\ldots\ldots, y_n >$ and wish to find a maximum length common subsequence of X and Y.

Longest Common Subsequence Problem (LCS problem)

Example:

- If X = < A, B, C, B, D, A, B > and Y= < B, D, C, A, B, A >, the sequence < B, C, A > is a common subsequence of both X and Y .
- The sequence < B, C, A > is not a *longest* common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence < B, C, B, A >, which is also common to both X and Y , has length 4.
- The sequence < B, C, B, A > is an LCS of X and Y , as is the sequence < B, D, A, B >, since X and Y have no common subsequence of length 5 or greater.

Solving the LCS problem using dynamic programming

Step-1: Characterizing a longest common subsequence

***Prefix of* a sequence:**

Given a sequence $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$, we define the ith ***prefix*** of X, for i = 0, 1, 2, ..., m, as $X_i = < x_1, x_2, \ldots\ldots\ldots, x_i >$.

For example, if X = < A, B, C, B, D, A, B >, then X4= <A, B, C, B > and X_0 is the empty sequence.

Solving the LCS problem using dynamic programming

Optimal substructure of an LCS

Let $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$ and $Y = < y_1, y_2, \ldots\ldots\ldots, y_n >$ be sequences, and let $Z = < z_1, z_2, \ldots\ldots\ldots, z_k >$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Solving the LCS problem using dynamic programming

Step 2: A recursive solution

Let c[i, j] be the length of an LCS of the sequences X_i and Y_j. If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula:-

$$
\begin{aligned}
c[i,j] &= 0 && \text{if } i = 0 \text{ or } j = 0 \\
&= c[i-1,j-1]+1, && \text{if } i, j > 0 \text{ and } x_i = y_j \\
&= \max\{ c[i-1, j], c[i,j-1] \}, && \text{if } i, j > 0 \text{ and } x_i \neq y_j
\end{aligned}
$$

Let b is the matrix which stores one of the following three arrows, ←, ↑ and ↖.

$b[i,j] = \leftarrow$, if $c[i,j] = c[i,j-1]$

$b[i,j] = \uparrow$, if $c[i,j] = c[i-1,j]$

$b[i,j] = \leftarrow$, if $c[i,j] = c[i-1,j-1]$

LCS problem using dynamic programming

Step-3: Computing the length of an LCS

Following procedure is used to compute the table b and c.

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

LCS problem using dynamic programming

Step-3: Computing the length of an LCS.

Example: Find LCS of the following sequences:-

X = < A, B, C, B, D, A, B > and Y = < B, D, C, A, B, A >.

Solution: We compute the table corresponding to b and c as following:-

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS problem using dynamic programming

Step-4: Constructing an LCS

- The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS(b, X, m, n).

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

LCS problem using dynamic programming

Step-4:

Time complexity of this algorithm = O(m+n).

The LCS of sequences taken in previous example = BCBA.

0-1 Knapsack using dynamic programming

- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.

- Based on the nature of the items, Knapsack problems are categorized as
  1. Fractional Knapsack
  2. 0-1 Knapsack

0-1 Knapsack using dynamic programming

Fractional Knapsack

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of ith item.

0-1 Knapsack

In this version of Knapsack problem, we cannot break an item, either pick the complete item or don't pick it (0-1 property).

0-1 Knapsack using dynamic programming

Dynamic programming approach for solving 0-1 Knapsack problem is explained as following:-

Let c[i,w] denotes the value of the solution for items 1,2,3,…,i and maximum weight w.

It is defined as

$$
\begin{aligned}
c[i,w] &= 0 && \text{, if } i = 0 \text{ or } w = 0 \\
&= c[i-1, w] && \text{, if } i > 0 \text{ and } w_i > w \\
&= \max\{ v_i + c[i-1, w-w_i], c[i-1,w] \} && \text{, if } i > 0 \text{ and } w_i \leq w
\end{aligned}
$$

0-1 Knapsack using dynamic programming

Dynamic programming algorithm for solving 0-1 Knapsack problem is the following:-

Dynamic-0-1-Knapsack(v, w, n, W)

```
for l ← 0 to W
        c[0,l] ← 0
for i ← 1 to n
        c[i,0] ← 0
        for l ← 1 to W
                if w_i ≤ l then
                        if v_i + c[i-1,l-w_i] > c[i-1,l] then
                                c[i,l] ← v_i + c[i-1,l-w_i]
                        else
                                c[i,l] ← c[i-1,l]
                else
                        c[i,l] ← c[i-1,l]
return c
```

Time complexity of this algorithm = O(nW)

0-1 Knapsack using dynamic programming

Example: Solve the following 0-1 knapsack.

$n = 4, \quad W = 5$

$(v_1, v_2, v_3, v_4) = (3, 4, 5, 6)$

$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$

Solution:

Here, we have to calculate $c[4, 5]$.

$c[4,5] = \max\{ v_4 + c[3,0], c[3,5] \} = \max\{ 6 + 0, c[3,5] \}$

$= \max\{ 6, c[3,5] \} = \max\{ 6, 7\} = 7$

$c[3,5] = \max\{ v_3 + c[2, 1], c[2, 5] \} = \max\{ 5 + 0, c[2, 5] \}$

$= \max\{ 5, c[2, 5] \} = \max\{ 5, 7\} = 7$

$c[2,5] = \max\{ v_2 + c[1, 2], c[1, 5] \} = \max\{ 4 + 3, 3\} = 7$

Therefore, the value of the optimal solution is 7. And the optimal solution is (1, 2).

0-1 Knapsack using dynamic programming

Solution by Tabular Method

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

This table is used to find the value of the optimal solution.

Therefore, value of optimal solution = 7

0-1 Knapsack using dynamic programming

Solution by Tabular Method

	1	2	3	4	5
1	↑	←	←	←	←
2	↑	↑	←	←	←
3	↑	↑	↑	←	↑
4	↑	↑	↑	↑	↑

This table is used to find the optimal solution.

Therefore, optimal solution = (1, 2)

0-1 Knapsack using dynamic programming

Example: Solve the following 0-1 knapsack.

$n = 6, \quad W = 100$

$(v_1, v_2, v_3, v_4, v_5, v_6) = (40, 35, 20, 4, 10, 6)$

$(w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 40, 20, 10, 10)$

Solution:

	0	10	20	30	40	50	60	70	80	90	100
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	40
2	0	0	0	0	0	35	35	35	35	35	40
3	0	0	0	0	20	35	35	35	35	55	55
4	0	0	4	4	20	35	35	39	39	55	55
5	0	10	10	14	20	35	45	45	49	55	65
6	0	10	16	16	20	35	45	51	51	55	65

0-1 Knapsack using dynamic programming

From table in the previous slide, the value of optimal solution will be 65.

Following is the table to compute the optimal solution.

The optimal solution will be (2, 3, 5).

	10	20	30	40	50	60	70	80	90	100
1	↑	↑	↑	↑	↑	↑	↑	↑	↑	←
2	↑	↑	↑	↑	←	←	←	←	←	↑
3	↑	↑	↑	←	↑	↑	↑	↑	←	←
4	↑	←	←	↑	↑	↑	←	←	↑	↑
5	←	←	←	↑	↑	←	←	←	↑	←
6	↑	←	←	↑	↑	↑	←	←	↑	↑

Floyd's Warshall Algorithm for shortest path

- This algorithm is used to solve the all-pairs shortest-paths problem on a directed graph G =(V,E).
- This algorithm is based on the dynamic programming approach.
- Let $\mathbf{d}^{(k)}_{ij}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, ..., k}.
- When k = 0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $\mathbf{d}^{(0)}_{ij} = \mathbf{w}_{ij}$.

Floyd's Warshall Algorithm for shortest path

Recursive formula to compute $d^{(k)}_{ij}$

We define $d^{(k)}_{ij}$ as the following:-

$$d^{(k)}_{ij} = w_{ij} \quad \text{if } k = 0$$

$$= \min\{ d^{(k-1)}_{ij} , d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \} \quad \text{if } k \geq 1.$$

- Because for any path, all intermediate vertices are in the set {1, 2, ..., n}, therefore the matrix $D^{(n)} = (d^{(n)}_{ij})$ gives the final answer.

Floyd's Warshall Algorithm for shortest path

Constructing a shortest path

- We define $\pi^{(k)}_{ij}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set {1, 2, ..., k}.
- When **k = 0**, a shortest path from i to j has no intermediate vertices at all. Therefore,

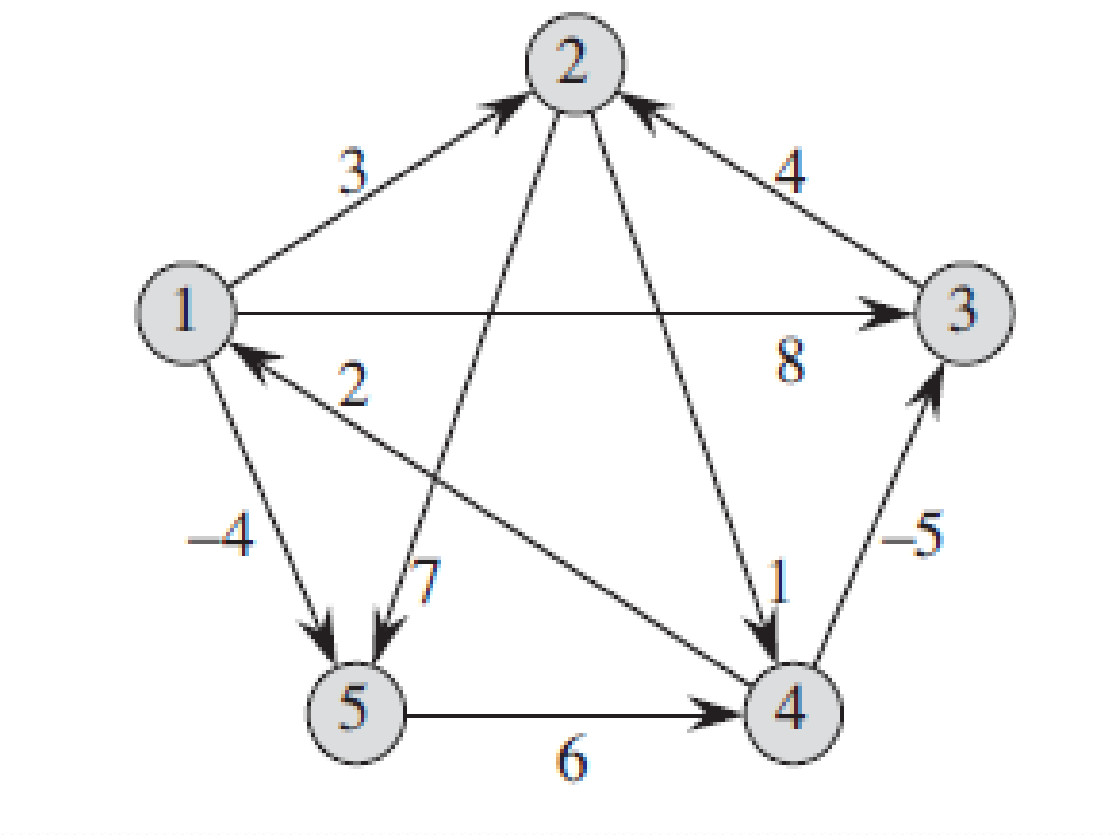
$$\pi^{(0)}_{ij} = \text{NIL} \quad \text{if } i=j \text{ or } w_{ij} = \infty$$
$$= i \quad \text{if } i \neq j \text{ and } w_{ij} < \infty.$$

- **For k ≥ 1.**

$$\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij}, \quad \text{if } d^{(k-1)}_{ij} \leq d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$$
$$= \pi^{(k-1)}_{kj}, \quad \text{if } d^{(k-1)}_{ij} > d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$$

Floyd's Warshall Algorithm for shortest path

Example: Apply the Floyd's Warshall algorithm in the following graph :-



Floyd's Warshall Algorithm for shortest path

Solution: Weighted matrix of this graph is the following:-

$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Therefore,

$$D^{(o)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(o)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

Here, we have to calculate $D^{(5)}$ and $\Pi^{(5)}$. It is calculated as following:-

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

- Matrix $D^{(5)}$ stores the shortest distance between each pair of vertices.
- Matrix $\Pi^{(5)}$ is used to find shortest path between each pair of vertices.

Floyd's Warshall Algorithm for shortest path

```
Floyd-Warshall(W, n)
    D^(0) = W
    for i←1 to n
        for j← 1 to n
            if (i=j or w_ij=∞)
                π^(0)_ij = Nil
            if (i≠j and w_ij< ∞)
                π^(0)_ij = i
    for k←1 to n
        for i←1 to n
            for j← 1 to n
                if (d^(k-1)_ij ≤ d^(k-1)_ik + d^(k-1)_kj )
                    d^(k)_ij = d^(k-1)_ij
                    π^(k)_ij = π^(k-1)_ij
                else
                    d^(k)_ij = d^(k-1)_ik + d^(k-1)_kj
                    π^(k)_ij = π^(k-1)_kj
```

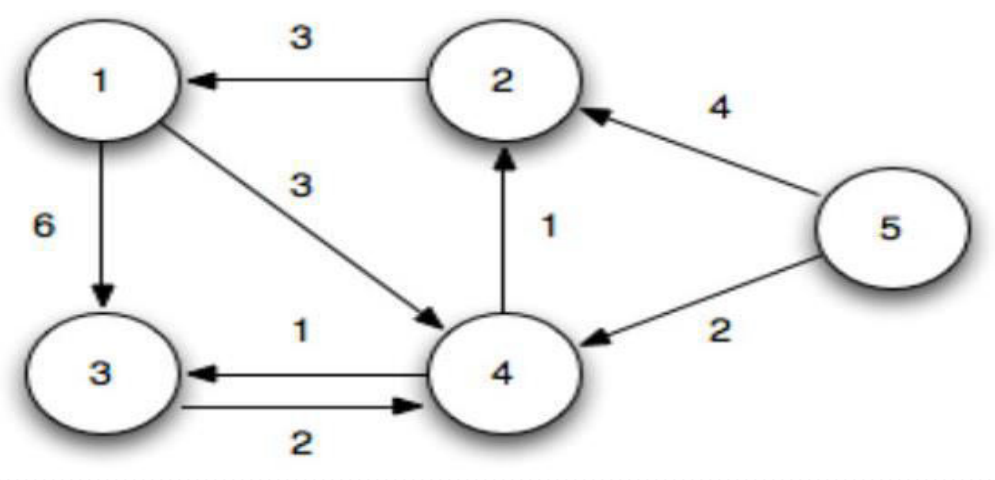
Time complexity of this algorithm is $\theta(n^3)$.

AKTU Examination Questions

1. Define principal of optimality. When and how dynamic programming is applicable.
2. Give Floyd Warshall algorithm to find the shortest path for all pairs of vertices in a graph. Give the complexity of the algorithm. Explain with example.
3. Difference between Greedy Technique and Dynamic programming.
4. Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyze its time complexity.
5. What is dynamic programming? How is this approach different from recursion? Explain with example.
6. Solve the following 0/1 knapsack problem using dynamic programming. P=[11,21,31,33] w=[2,11,22,15] c=40, n=4.

AKTU Examination Questions

7. Define Floyd Warshall Algorithm for all pair shortest path and apply the same on following graph:



7. Define the terms—LCS, Matrix Chain multiplication & Bellman-Ford algorithm.
8. Explain the Floyd Warshall algorithm with an example.
9. Find an optimal parenthesization of a matrix chain product whose sequence of dimensions is {10, 5, 3, 12, 6}.

Backtracking

Backtracking

- **Backtracking** is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

- Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation.

Backtracking

- In the backtrack method, the desired solution is expressible as an n-tuple (x_1, x_2, … , x_n), where the x_i are chosen from some finite set S_i.
- Using backtracking, we find one vector that maximizes(or minimizes or satisfies) a criterion function P (x_1, x_2, … , x_n).
- Sometimes, this method finds all the vectors that satisfies P.
- If m_i is the size of set S_i, then $m = m_1 m_2 \ldots\ldots m_n$ tuples are possible that satisfy the criterion function P.
- The brute force approach would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum.
- The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials.

Backtracking

- Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions P $(x_1, x_2, \ldots, x_i)$. (sometimes called bounding functions) to test whether the vector being formed has any chance of success.

- If the partial vector $(x_1, x_2, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \ldots\ldots m_n$ possible test vectors can be ignored entirely.

Backtracking

Explicit constraints

Explicit constraints are rules that restrict each x_i to take on values only from a given set.

Common example of explicit constraints are

(1) $x_i \geq 0$ or S_i = The set of all positive real numbers

(2) $x_i = 0$ or 1 or $S_i = \{0, 1\}$

All tuples that satisfy the explicit constraints define a possible solution space.

Implicit constraints

The implicit constraints are rules that determine which of the tuples in the solution space satisfy the criterion function.

Backtracking

State space tree

When we search the solutions of the problem using backtracking, a tree is formed by the result of search. This tree is said to be sate space tree. Each node in the tree represents a state.

Problem state

Each node in the tree is called problem state.

Solution state

Solution states are those problem states s for which the path from root to s defines a tuple in the solution space.

Answer state

Answer states are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions of the problem.

Backtracking

State space

All the paths from the root to other nodes define the state space of the problem.

Live node

A node which has been generated and all of whose children have not yet been generated is called a live node.

E-node

The live node whose children are currently being generated is called the E-node (node being expanded).

Dead node

A dead node is a generated node which is not to be expanded further or all of whose children have been generated.

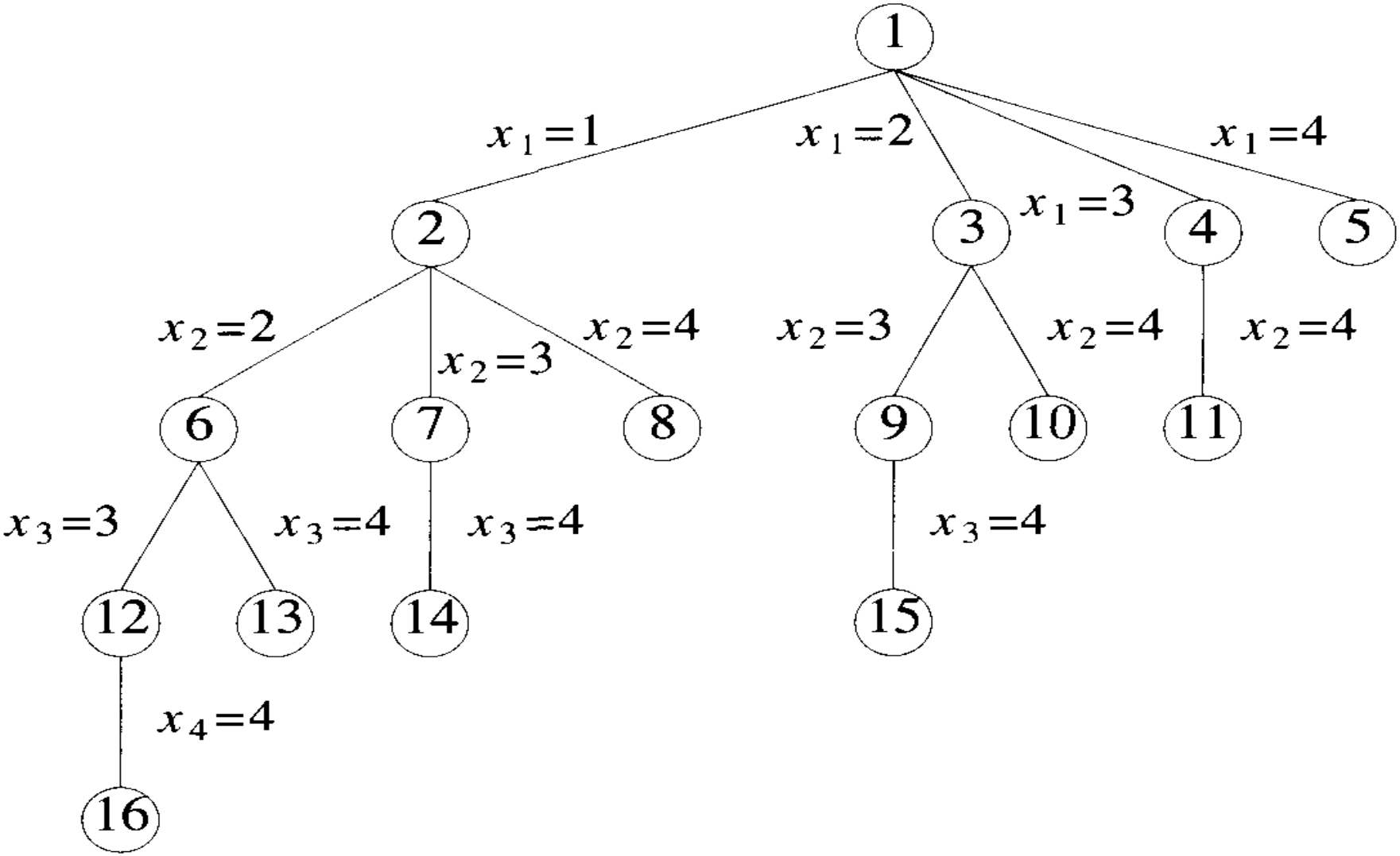
Backtracking

Note: Bounding functions are used to kill live nodes without generating all their children.

Note: Depth first node generation with bounding functions is called backtracking.

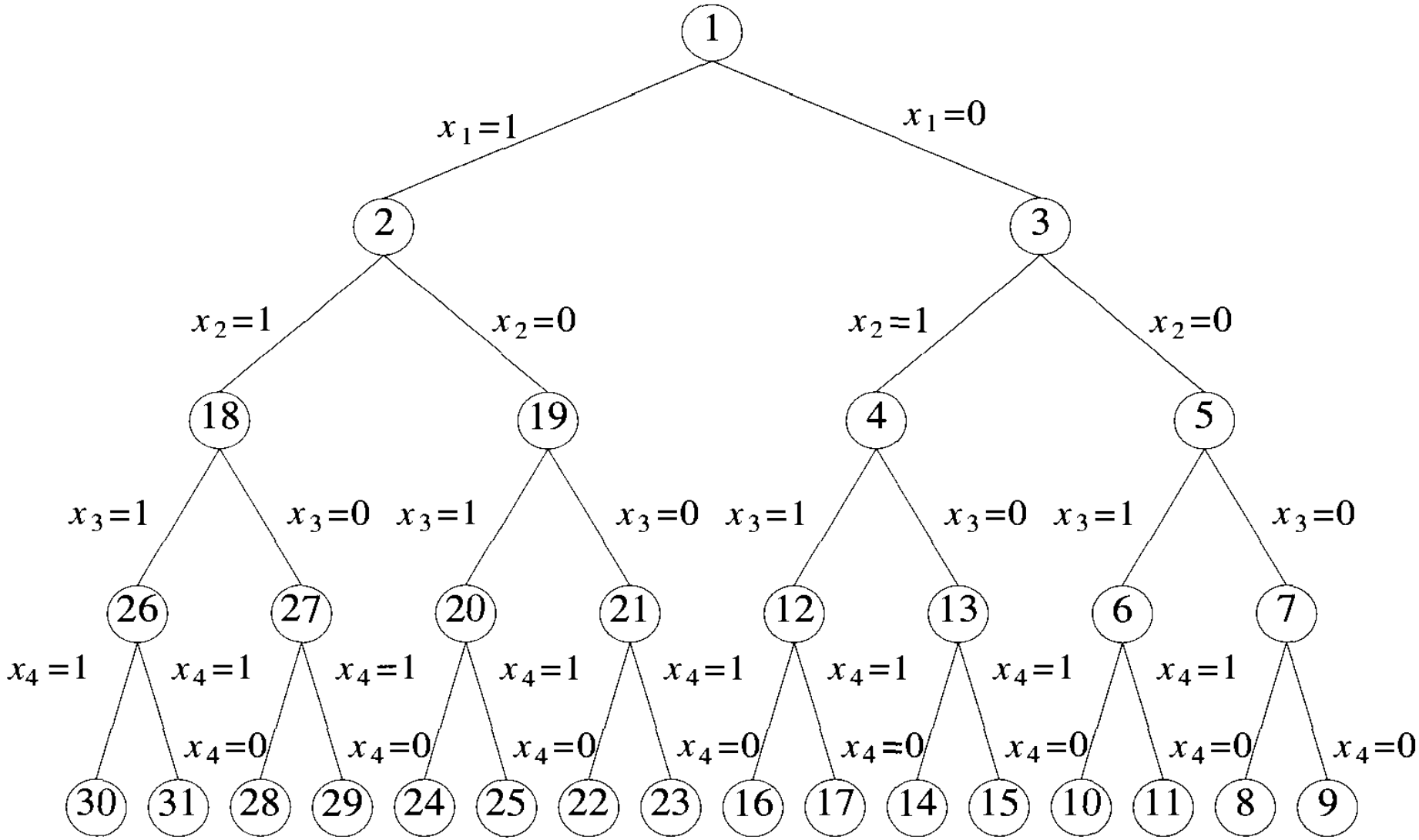
Backtracking

Example:



Backtracking

Example:



N-Queen Problem

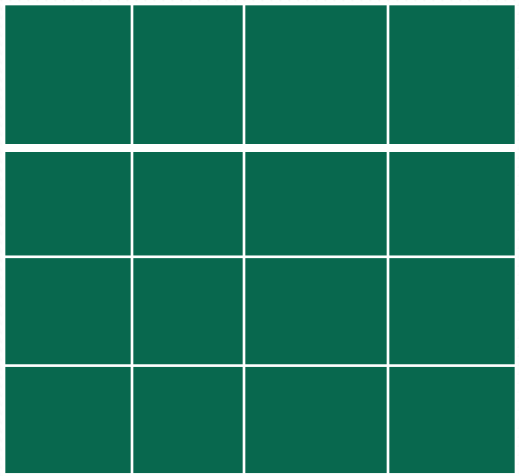
Statement:

In this problem, we have given N queens and a NxN chessboard. The objective of the problem is to place all the N queens on this chessboard in such way no two queens lie on the same row, column or diagonal.

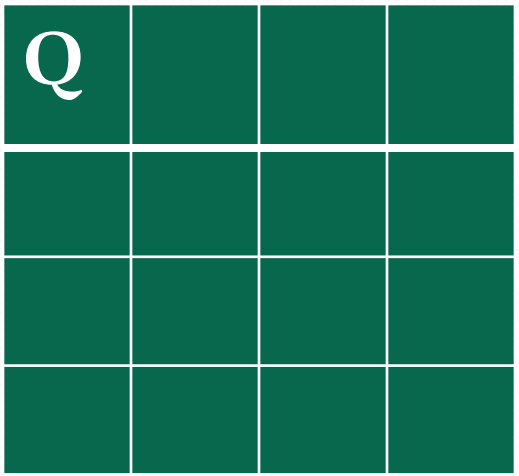
N-Queen Problem

Example: Solve the 4-Queen Problem.

Solution:



$\Rightarrow$



Initial position of chessboard

N-Queen Problem

Q			
		Q	

⇒

Q			
			Q
	Q		

⇒

	Q		
			Q
Q			
		Q	

Final solution

Backtracking algorithm for N-Queen

- According to this approach, we put 1st queen in 1st row, 2nd queen in 2nd row and so on. We have to only find column number of the queen.
- Therefore, the solution vector will be $(x_1, x_2, \ldots\ldots, x_n)$. Here x_i is the column number of ith queen.
- The explicit constrains for this problem are that no two x_i's can be same and no two queens can be on the same diagonal.
- Using these two constraints, the size of solution space are n!.
- Suppose two queens are placed at positions (i,j) and (k,l). They are on the same diagonal only if

$$i-j = k-l \quad \text{or} \quad i+j = k+l$$

$$\Rightarrow \quad i-k = j-l \quad \text{or} \quad i-k = -(j-l)$$

Therefore, two queens lie on the same diagonal iff

$$|i-k| = |j-l|$$

Backtracking algorithm for N-Queen

An algorithm f or determining the solution of n-queens problem is:-

```
N-Queen(k, n)
for i ← 1 to n
        do if  Place(k,i) = True
                then  x[k] ← I
                        if k=n
                            then for  j←1 to n
                                    do print x[j]
                        else
                                N-Queen(k+1, n)
```

Backtracking algorithm for N-Queen

An algorithm f or determining the solution of n-queens problem is:-

```
Place(k, i)
for j ← 1 to k-1
        do if  (x[j]=i) or (abs(x[j]-i) = abs(j-k))
                then  return (False)
return(True)
```

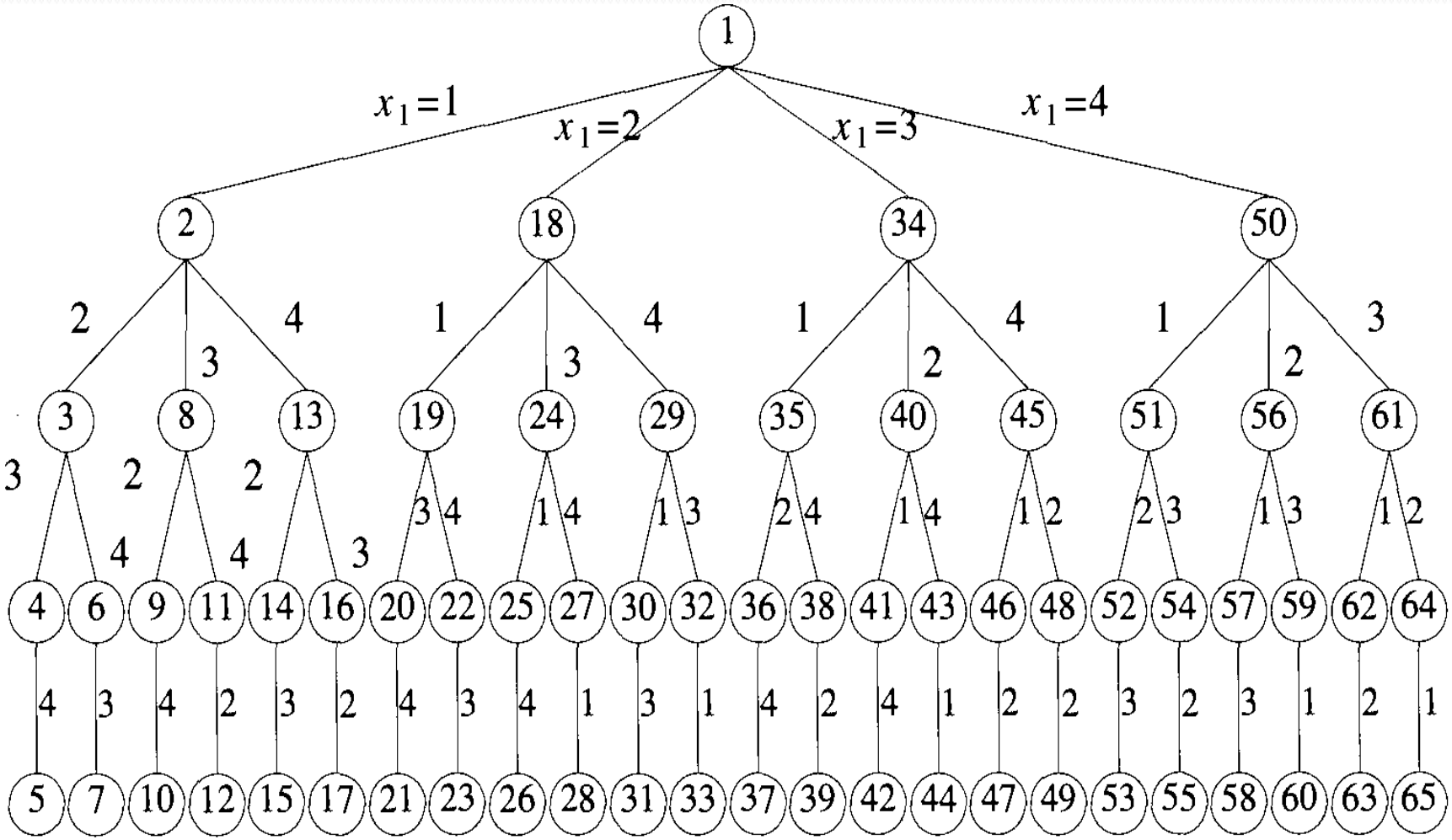
The initial call is N-Queen(1, n).

Running time of this algorithm is O(n!).

Backtracking algorithm for N-Queen

Example: Draw the state space tree for 4-queen problem.

Solution: State space tree for 4-queen problem is the following:-



Backtracking algorithm for N-Queen

Example: Solve the 8-queens problem.

Solution:

		Q					
					Q		
	Q						
						Q	
Q							
			Q				
							Q
				Q			

Sum of Subsets problem

Statement: In this problem, we are given n distinct positive numbers (called weights) and one another number m. We have to find all combinations of these numbers whose sums are m.

This problem can be formulated using either fixed or variable sized tuples.

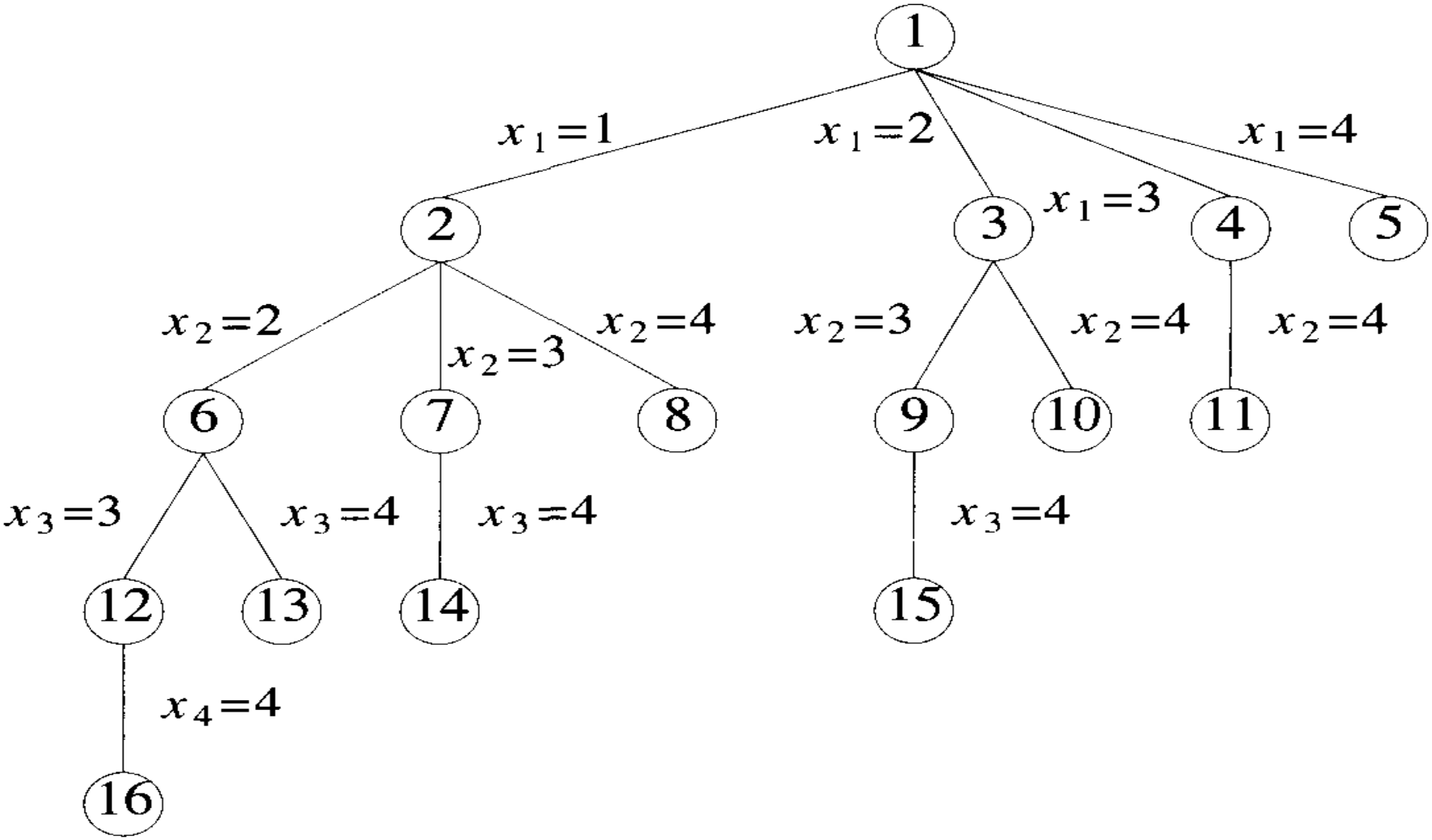
Example: Consider n=4 , (w1, w2, w3, w4) = (11, 13, 24, 7)

and m=31.

- The desired subsets are (11, 13, 7) and (24, 7).
- The solution in variable-size tuple will be (1, 2, 4) and (3,4).
- The solution in fixed-size tuple will be (1, 1, 0, 1) and (0, 0, 1, 1).

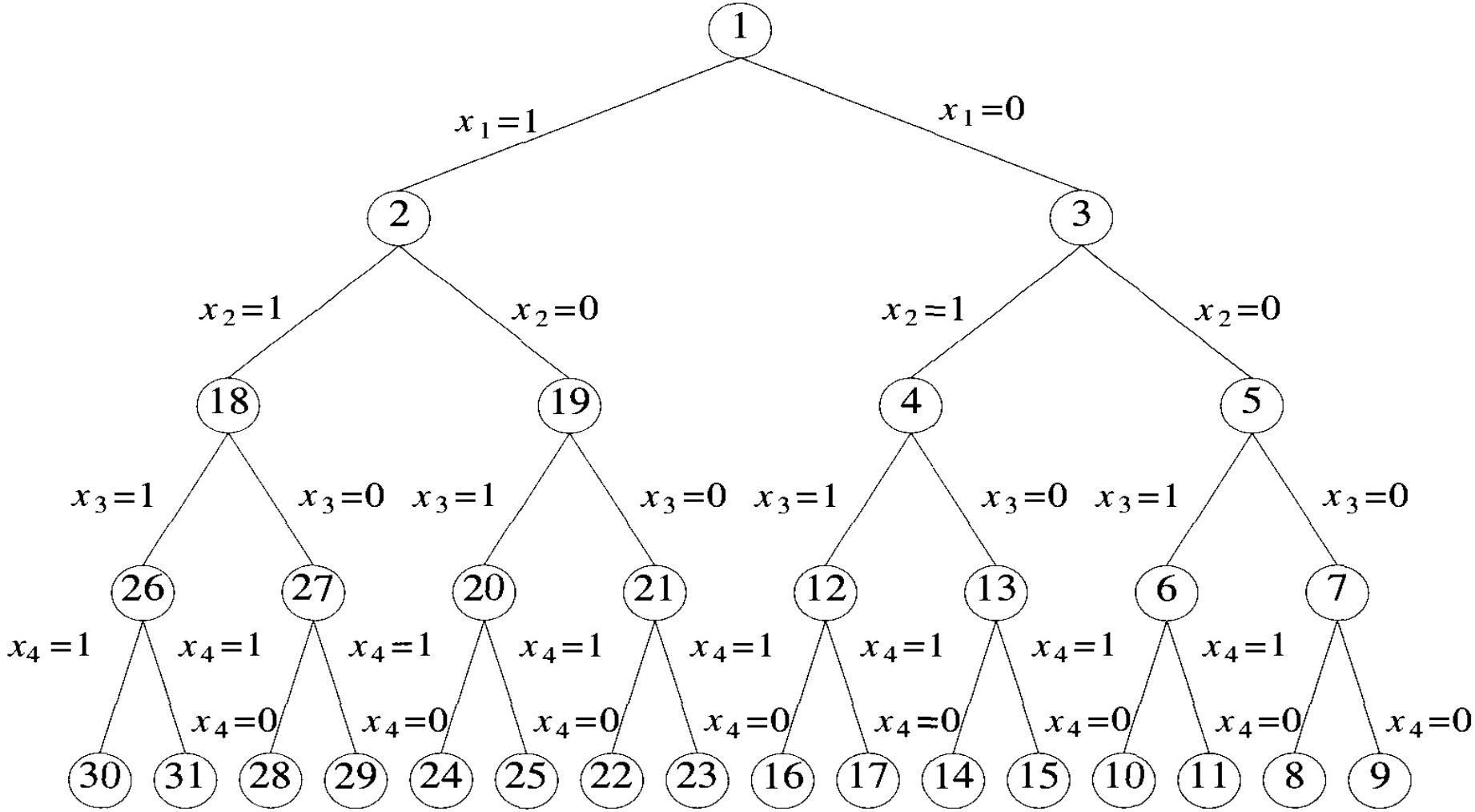
Sum of Subsets problem

The state space tree for this question in case of variable-size tuple is the following:-



Sum of Subsets problem

The state space tree for this question in case of fixed-size tuple is the following:-



Backtracking approach for Sum of Subsets problem

Backtracking solution for this problem uses fixed-sized tuple strategy.

In this case, the value of x_i in the solution vector will be either 1 or 0 depending on whether weight w_i is included or not.

The bounding function is $B_k(x_1, x_2, \ldots\ldots, x_k)$ = true iff

$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$

Clearly $(x_1, x_2, \ldots\ldots, x_k)$ can not lead to an answer node if this condition is not satisfied.

If all the weights are initially in ascending order then $(x_1, x_2, \ldots\ldots, x_k)$ can not lead to an answer node if

$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$

Backtracking approach for Sum of Subsets problem

Therefore, we use the following bounding function

The bounding function is $B_k(x_1, x_2, \ldots\ldots, x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

and

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

Note: Here, we have assume all the weights are in ascending order.

Backtracking approach for Sum of Subsets problem

SumOfSub(s,k,r)

```
x[k] ← 1
if (s+w[k] = m)
    for j←1 to k
        print x[j]
else if (s+w[k]+w[k+1] ≤ m)
        SumOfSub(s+w[k], k+1, r-w[k])
if (( s+r-w[k] ≥ m) and (s+w[k+1] ≤ m))
x[k] ← 0
        SumOfSub(s, k+1, r-w[k])
```

The initial call is SumOfSub(0, 1, $\sum_{i=1}^{n} w_i$)

Backtracking approach for Sum of Subsets problem

Time complexity of this algorithm is $O(2^n)$.

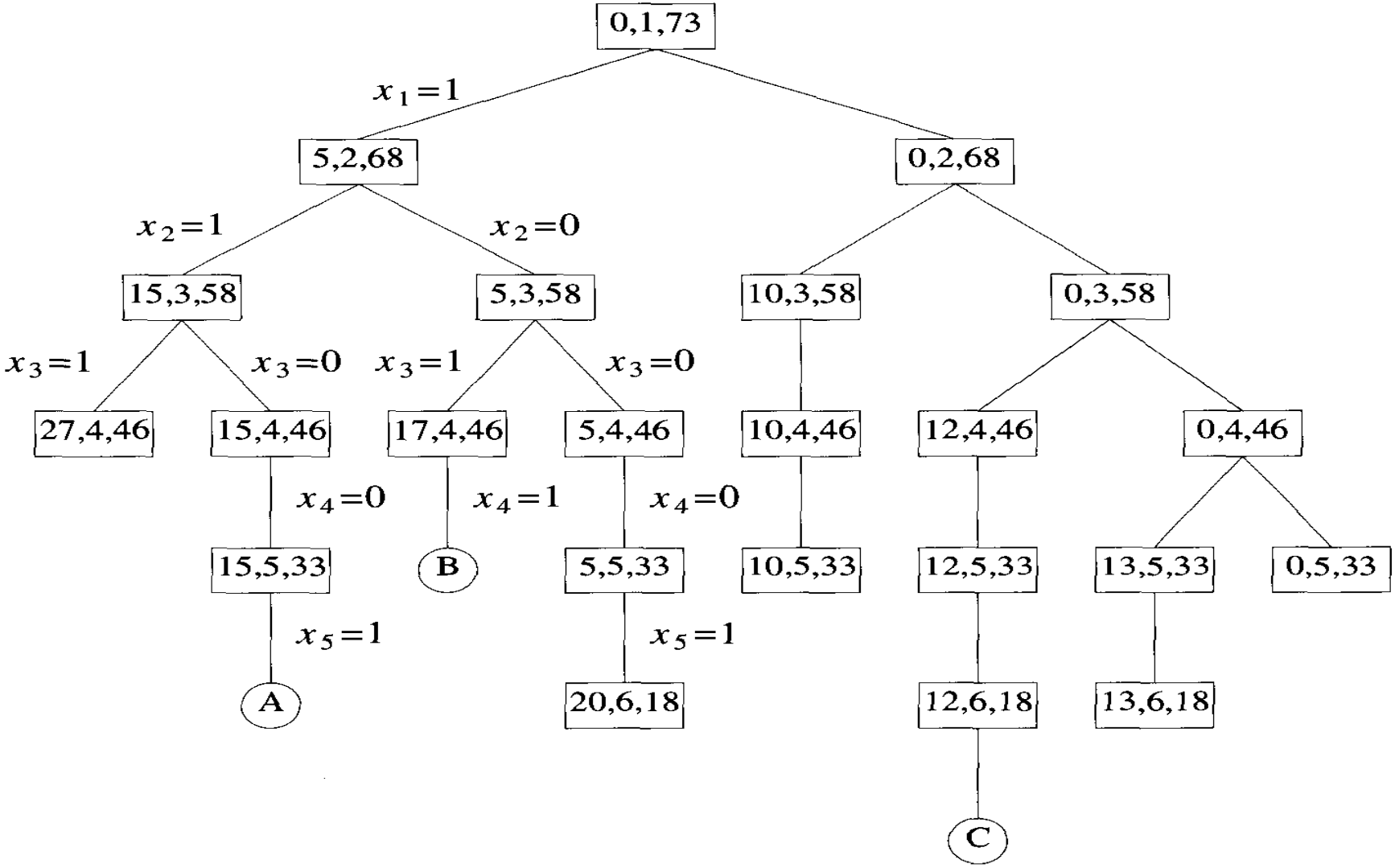
Because

Total number of nodes $= 1+2+2^2+2^3+\ldots\ldots+2^n$

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

Example: Let w = {5, 10, 12, 13, 15, 18} and m = 30. Find all possible subsets of w that sum to m. Do this using SumOfSub. Draw the portion of the state space tree that is generated.

Solution:



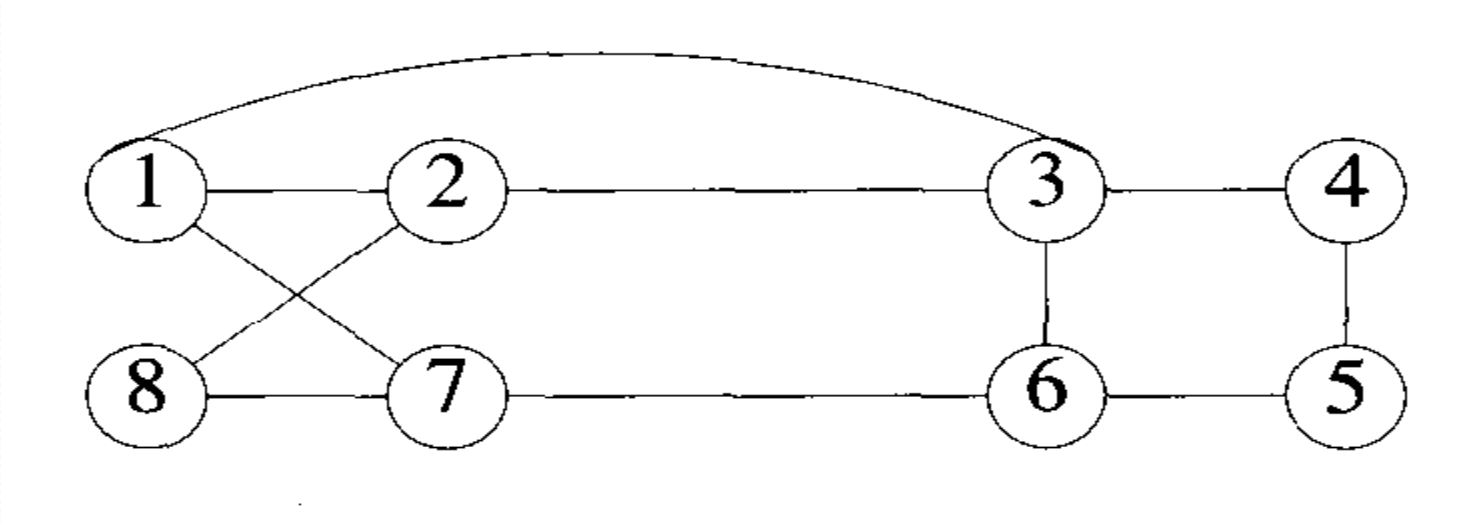
Hamiltonian Cycle Problem

Hamiltonian cycle: A cycle in a graph is said to be Hamiltonian cycle if it contains all the vertices of the graph and no vertex is repeated.

Statement: In Hamiltonian cycle problem, we have to find a Hamiltonian cycle present in the given graph.

Hamiltonian Cycle Problem

Example: Consider the following graph.



Find a Hamiltonian cycle in this graph.

Solution: Hamiltonian cycles are

(1, 3, 4, 5, 6, 7, 8, 2, 1)

(1, 2, 8, 7, 6, 5, 4, 3, 1)

Backtracking algorithm for Hamiltonian Cycle Problem

- The backtracking solution vector (x_1, x_2, ……, x_n) is defined so that x_i represents the ith visited vertex of the proposed cycle.

- First we choose, x_1, can be any vertex of the n vertices.

- We will take $x_1 = 1$. To avoid printing same cycle n-times, we require that $x_1 = 1$.

Backtracking algorithm for Hamiltonian Cycle Problem

- If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots\ldots, x_{k-1}$ and v is connected by an edge to x_{k+1}. The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1.

- This algorithm is started by first initializing the adjacency matrix G[1:n][1:n], then x[2:n] to 0 and x[1] to 1 and then executing algorithm Hamiltonian(2).

- Time complexity of this algorithm = **$O(2^n n^2)$.**

Backtracking algorithm for Hamiltonian Cycle Problem

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```

Backtracking algorithm for Hamiltonian Cycle Problem

```
Algorithm NextValue(k)
// x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
// no vertex has as yet been assigned to x[k]. After execution,
// x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1 : k − 1] and is connected by
// an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
// in addition x[k] is connected to x[1].
{
    repeat
    {
        x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
        if (x[k] = 0) then return;
        if (G[x[k − 1], x[k]] ≠ 0) then
        { // Is there an edge?
            for j := 1 to k − 1 do if (x[j] = x[k]) then break;
                                // Check for distinctness.
            if (j = k) then // If true, then the vertex is distinct.
                if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
                    then  return;
        }
    } until (false);
}
```

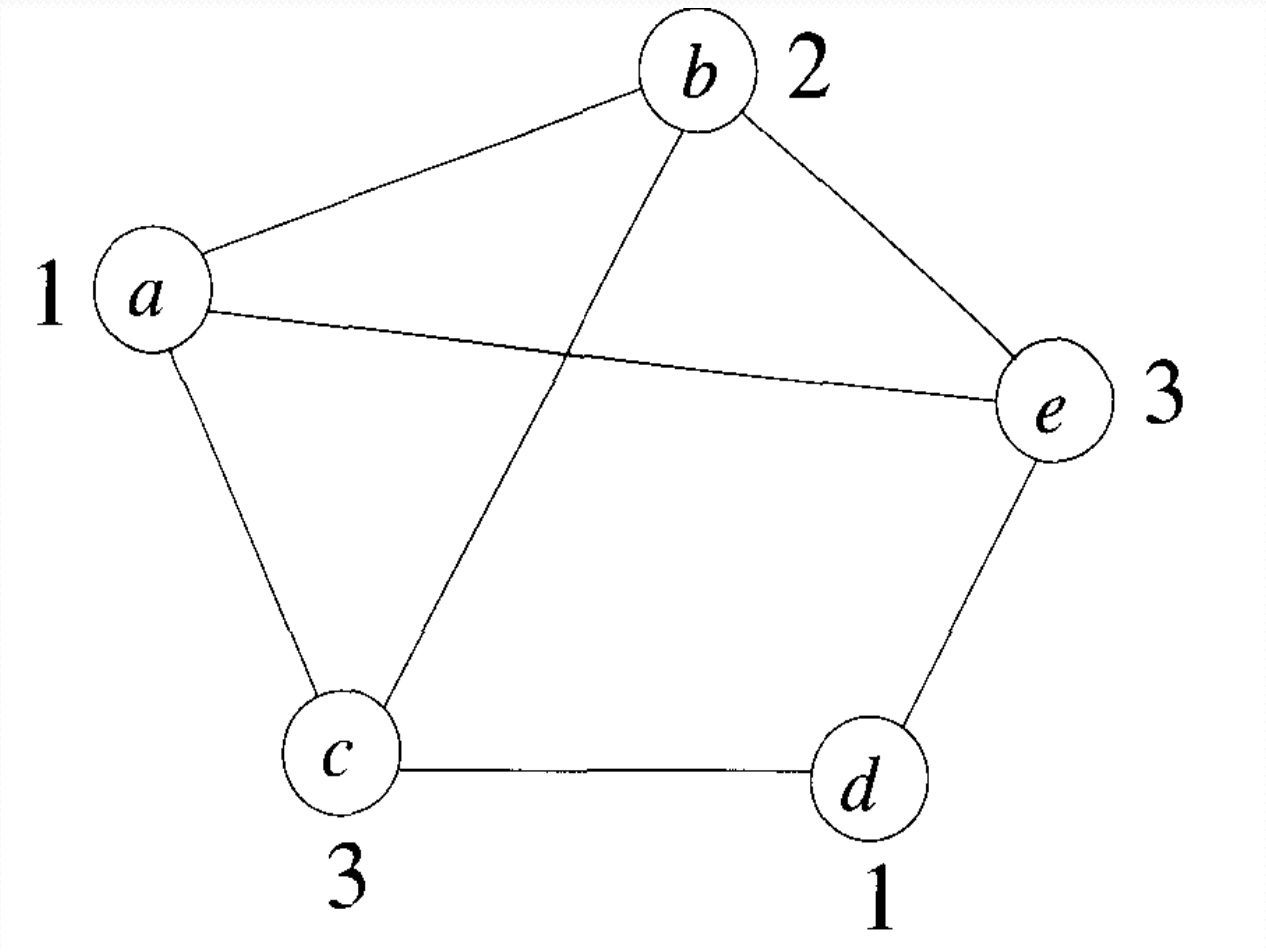
Graph Coloring Problem

- Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the **m-colorability decision problem.**

- The **m-colorability optimization problem** asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the **chromatic number** of the graph.

Graph Coloring Problem

Example: Consider the following graph



Find the chromatic number of this graph.

Backtracking algorithm for Graph Coloring Problem

- Following algorithm determines all the different ways in which a given graph can be colored using at most m colors.

- In this algorithm, a graph is represented by its adjacency matrix G[1:n, 1:n]. The colors are represented by the integers 1, 2, ..., m and the solutions are represented by the n-tuple (x_1, x_2, ..., x_n), where x_i is the color of node i.

- Function **mColoring** is begun by first assigning the graph to its adjacency matrix, setting the array x[] to 0, and then invoking the statement **mColoring(1).**

Backtracking algorithm for Graph Coloring Problem

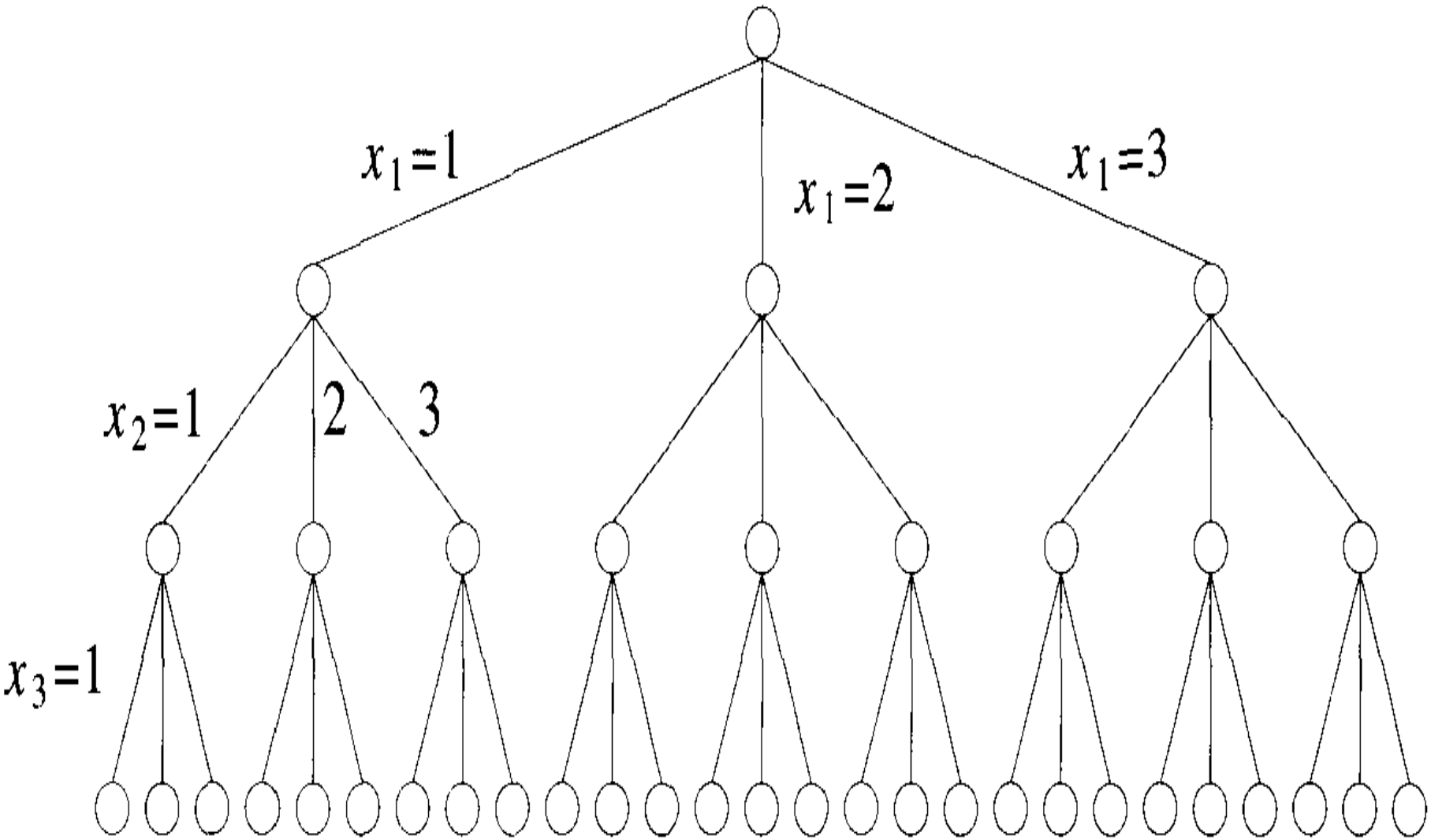
```
mColoring(k)
{
        while(1)
        {
                NextValue(k)
                if (x[k] = 0)
                        return
                if(k=n)
                        print x[1:n]
                else
                        mColoring(k+1)
        }
}
```

Backtracking algorithm for Graph Coloring Problem

```
NextValue(k)
{
        while(1)
        {
                x[k] ← (x[k] + 1) mod (m+1)
                if (x[k] = 0)
                        return
                for ( j← 1  to  n )
                        if(G[k,j] ≠ 0 and (x[k] = x[j]))
                                break
                if (j=n+1)
                        return
        }
}
```

Backtracking algorithm for Graph Coloring Problem

State space tree when n = 3 and m = 3 is



Backtracking algorithm for Graph Coloring Problem

- An upper bound on the computing time of mColoring can be arrived at by the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$.

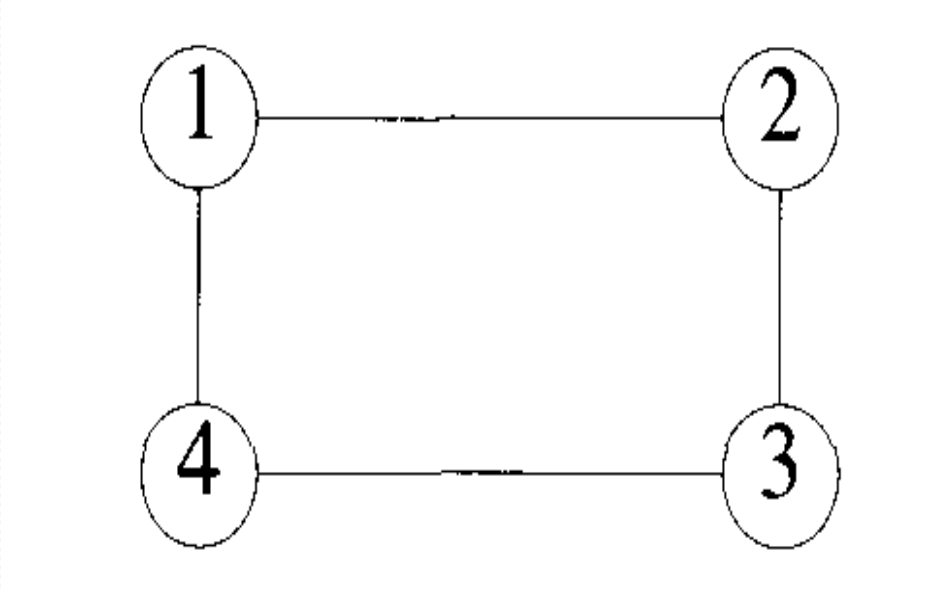
- At each internal node, O(mn) time is spent by NextValue to determine thechildren corresponding to legal colorings. Hence the total time is bounded by

$$\sum_{i=0}^{n-1} m^{i+1} \text{n} = \sum_{i=1}^{n} m^i n = \text{n}(\text{m}^{\text{n}+1} - \text{m})/(\text{m}-1)$$

$$= O(\text{nm}^{\text{n}})$$

Backtracking algorithm for Graph Coloring Problem

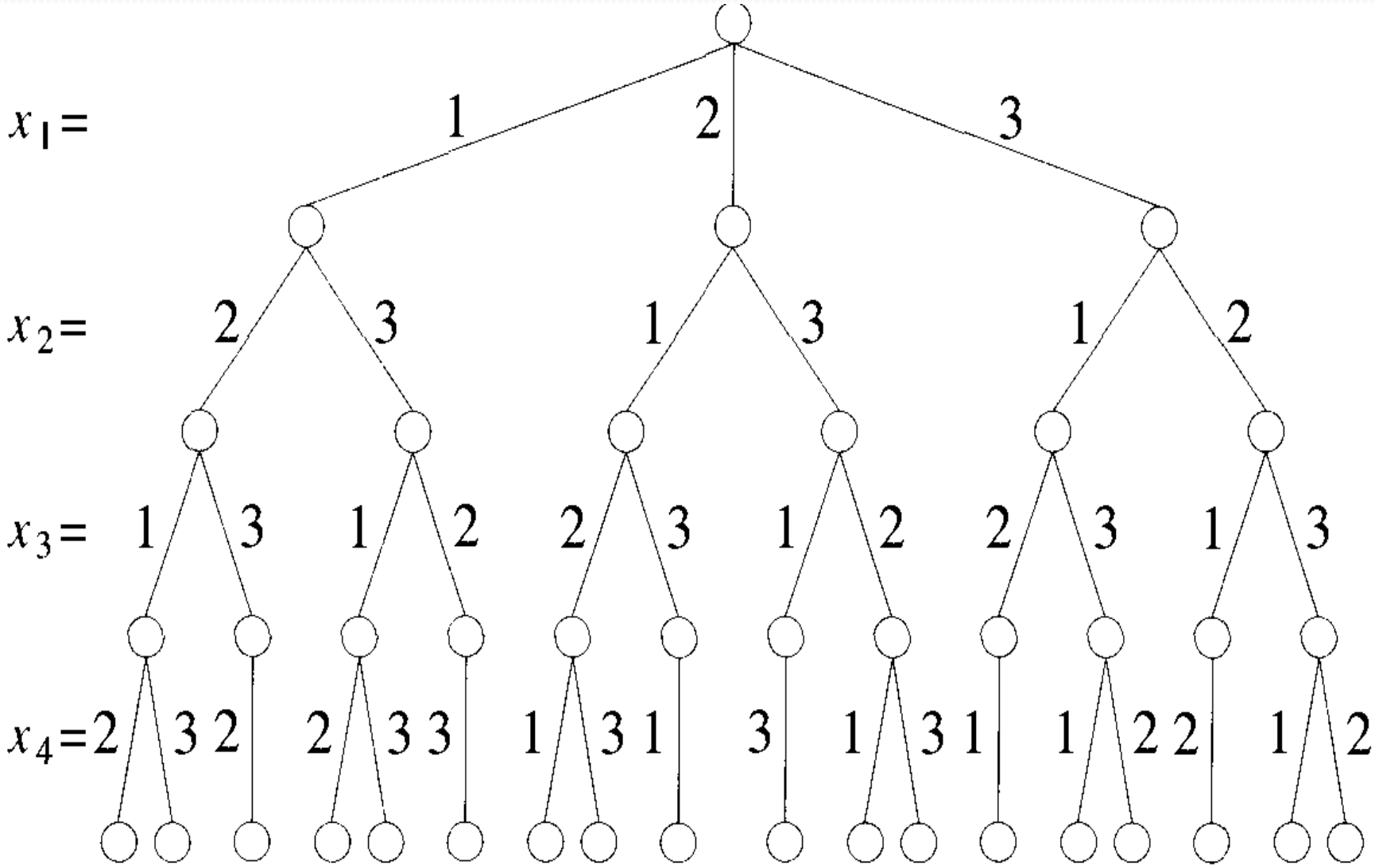
Example: Consider the following graph:-



- Here m = 3.
- Draw state space tree for this graph using mColoring.

Backtracking algorithm for Graph Coloring Problem

Solution: State space tree for the graph will be



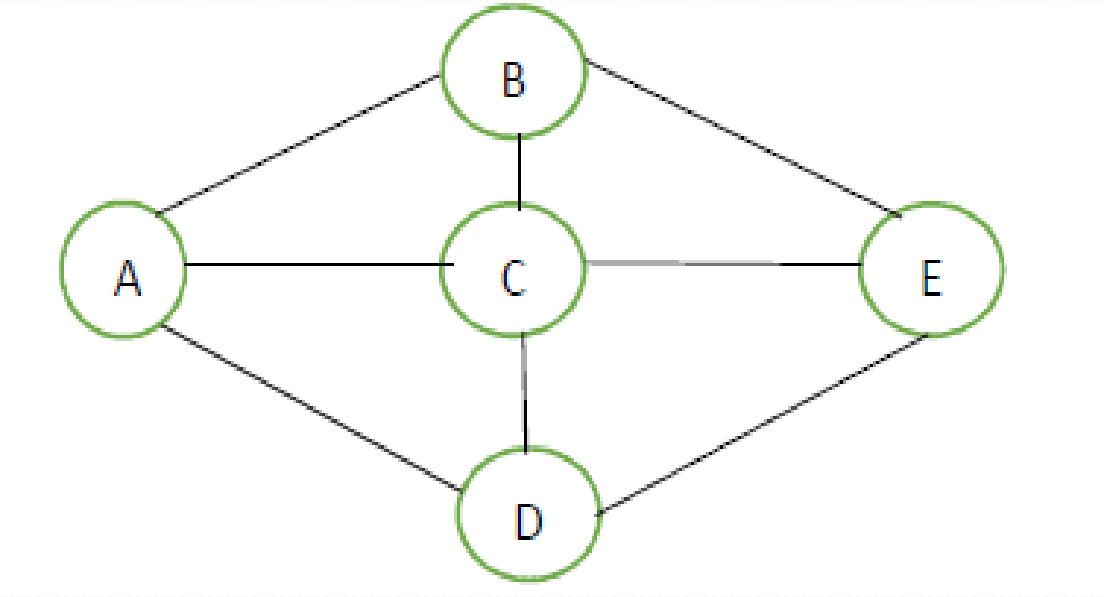
AKTU Examination Questions

1. Explain application of graph coloring problem.
2. Solve the Subset sum problem using Backtracking, where

   $n=4$, $m=18$, $w[4] = \{5, 10, 8, 13\}$
3. Define Graph Coloring.
4. What is backtracking? Discuss sum of subset problem with the help of an example.
5. Explain Implicit and Explicit constraints of N-queen Problem.

AKTU Examination Questions

6. What is the difference between Backtracking and Branch & Bound? Write Pseudo code for Subset Sum Problem using Backtracking. Give example for the same.
7. Consider a graph G=(V,E). We have to find a Hamiltonian cycle using backtracking method.



Branch and Bound

Branch and Bound

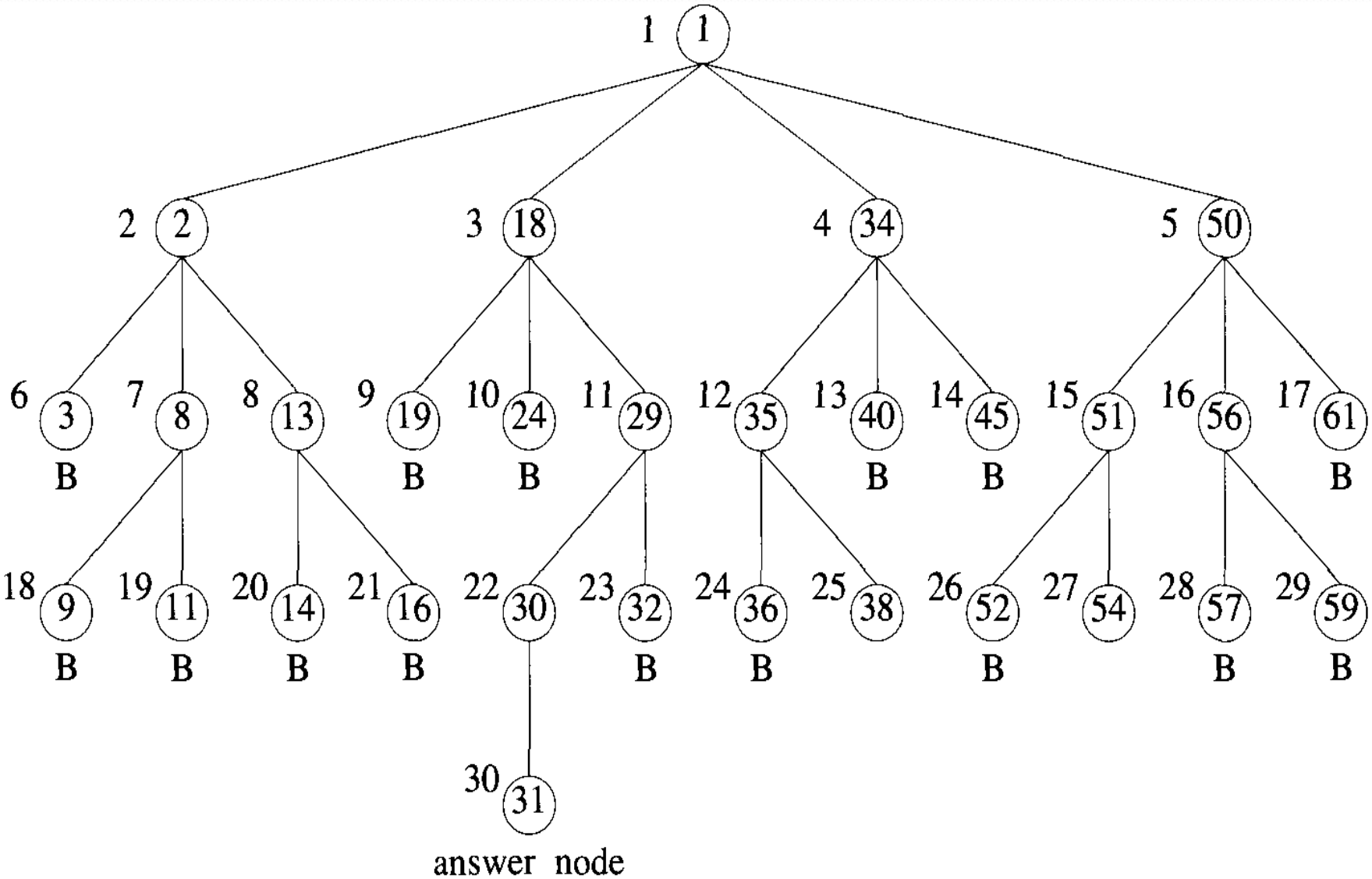
- A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution.
- Branch and bound is a systematic method for solving optimization problems. B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case.

Branch and Bound

- Branch and bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node.
- There are two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored.
- In branch and bound terminology, a BFS-like state space search will be called FIFO(First In First Out) search as the list of live nodes is a first-in-first-out list (or queue).
- A D-search-like state space search will be called LIFO(Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack).

Example: Consider the 4-queen problem.

FIFO branch-and-bound algorithm to search the state space tree for this problem.



Least Cost (LC) Search Branch and Bound(LCBB)

- In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind.
- The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.
- The search for an answer node can often be speeded by using an "intelligent" ranking function c' for live nodes.
- The next E-node is selected on the basis of this ranking function.
- A search strategy that uses a cost function c'(x) to select the next E-node would always choose for its next E-node a live node with least c'. Hence, such a search strategy is called an LC-search(Least Cost search).

Travelling Salesperson Problem

In this problem, we have given a directed graph with cost matrix.

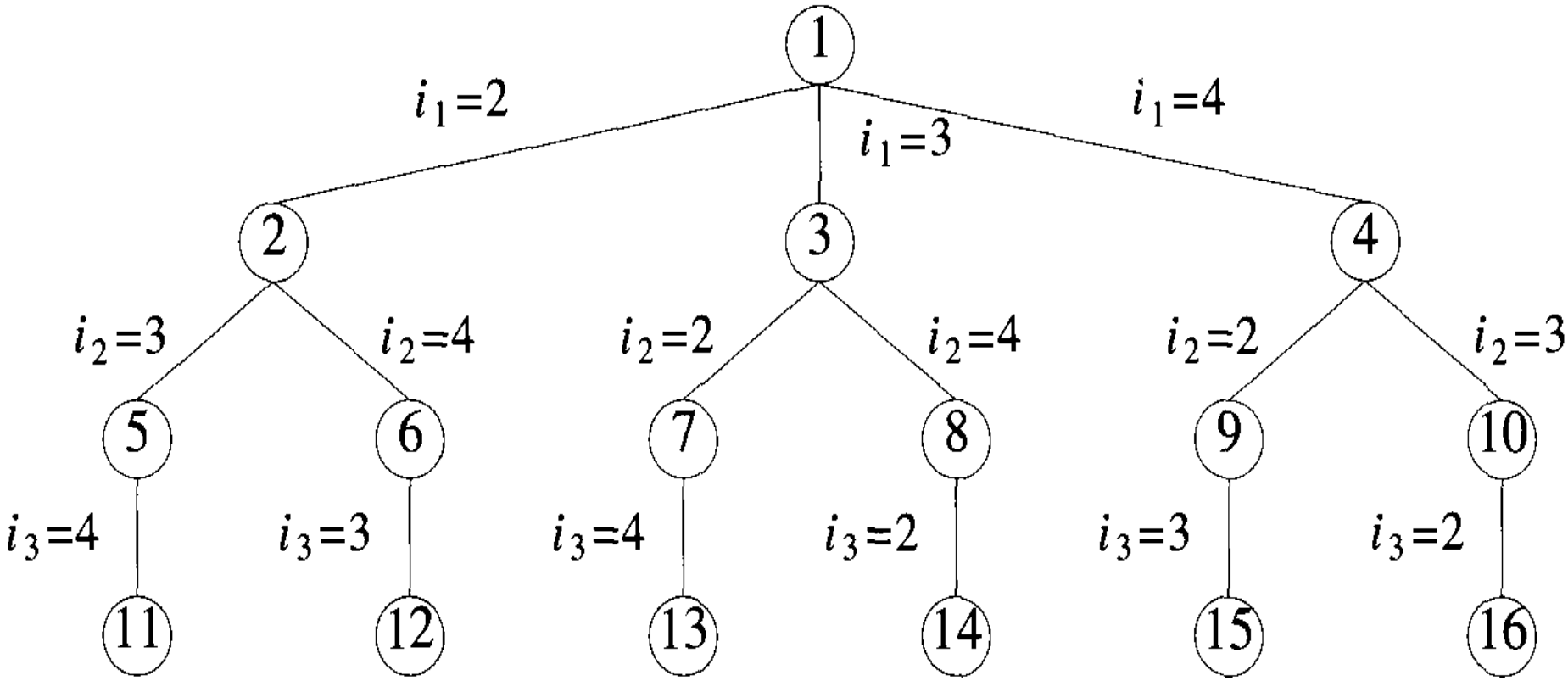
We have to find a shortest tour of the graph which contains all the vertices of the graph with no vertex repeated except first and last vertex.

Travelling Salesperson Problem

- Let G = (V, E) be a directed graph defining an instance of the traveling salesperson problem. Let c_{ij} denote the cost of edge(i, j), $c_{ij} = \infty$ if $(i,j) \notin E$, and let $|V| = n$.
- Without loss of generality, we can assume that every tour starts and ends at vertex 1. So, the solution space S is given by S= {1, **π**, 1 ! **π** is a permutation of (2,3,… , n)}.Then |S|= (n-1)! .
- The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $(i_j, i_{j-1}) \in E$, $0 \leq j \leq n-1$ and $i_0 = i_n = 1$.
- S can be organized into a state space tree.

Travelling Salesperson Problem

Following figure shows the tree organization for the case of a complete graph with |V| = 4.



State space tree for the traveling salesperson problem with n = 4 and $i_0 = i_4 = 1$

Travelling Salesperson Problem

- Each leaf node L is a solution node and represents the tour defined by the path from the root to L. Node 14 represents the tour $i_0 = 1$, $i_1 = 3$, $i_2 = 4$, $i_3 = 2$, and $i_4 = 1$.

Reduced row or column:

A row (column) is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.

Reduced cost matrix:

A matrix is said to be reduced iff every row and column is reduced.

Travelling Salesperson Problem

- To solve the travelling salesperson problem using LCBB method, we construct state space tree.
- We compute reduced cost matrix at each node of the state space tree. And also we compute a cost function c' at each node.
- Reduced cost matrix of root node is computed directly from given cost matrix of the graph.
- If C is the cost matrix of the graph, then reduced cost matrix of C is the reduced cost matrix corresponding to root node.
- If c' represent the cost function at each node to determine the optimal tour, then

c' for root node = Total amount subtracted in determining reduced cost matrix at root node

Travelling Salesperson Problem

Reduced cost matrix of the remaining node is computed in the following way:-

Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge (i,j) in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows:

(1) Change all entries in row i and column j of A to ∞.

(2) Set A(j,1) to ∞.

(3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞.

If r is the total amount subtracted in the step (3), then

$$c'(S) = c'(R) + A(i,j) + r$$

Travelling Salesperson Problem

Example: Solve the following travelling salesperson problem whose cost matrix is the following:-

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Solution: The reduced cost matrix corresponding to root node will be the following:-

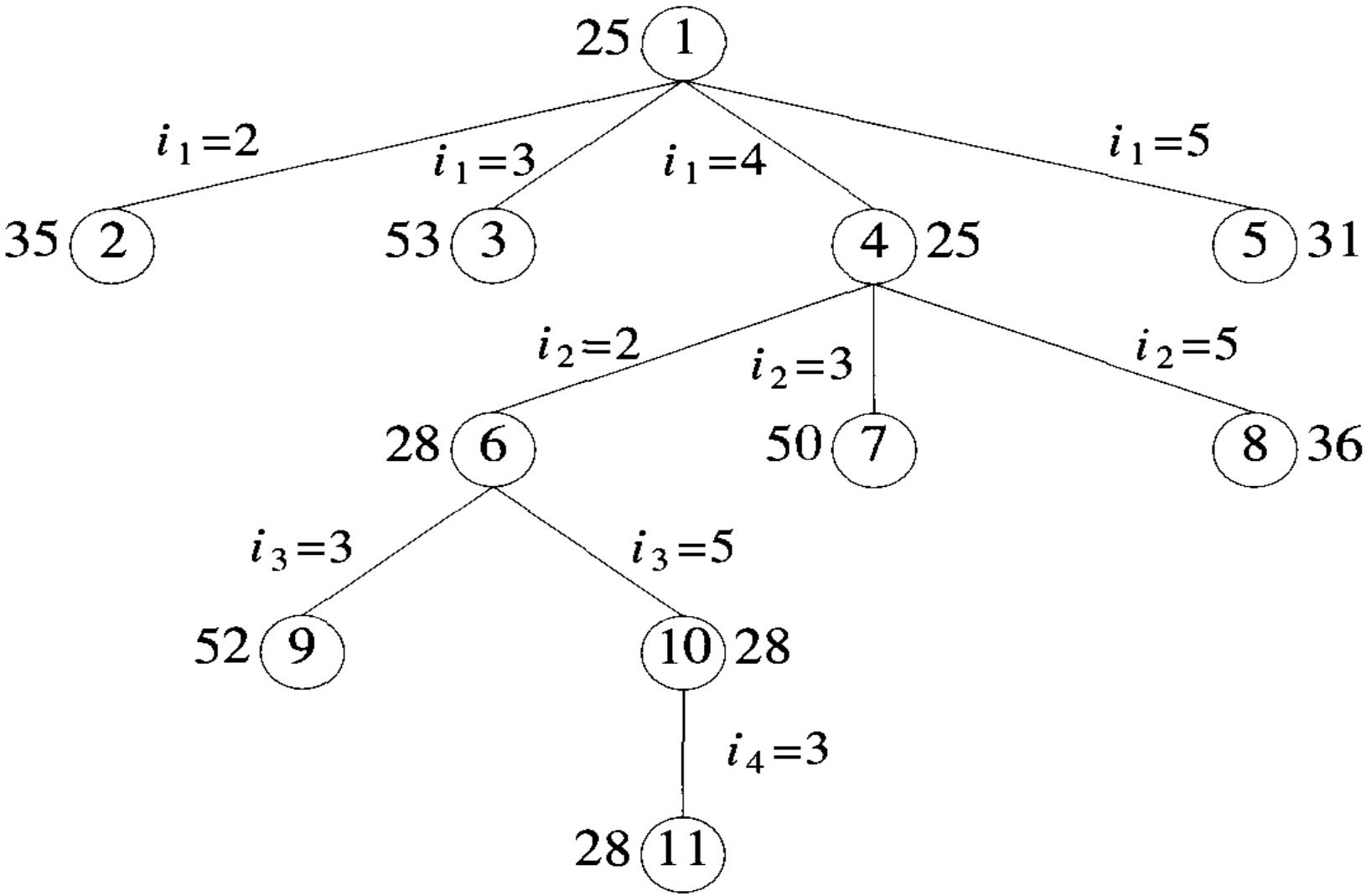
$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

And value of cost function at root node is

$c'(\text{root}) = 25$

Travelling Salesperson Problem

The portion of the state space tree that gets generated is



$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

AKTU Examination Questions

1. Differentiate between Backtracking and Branch and Bound Techniques.
2. How BFS is differ from DFS.
3. Consider I=<I1,I2,I3>; W=<5,4,3>; V=<6,5,4> and W=7, we have to pack this knapsack using the branch and bound technique.

Design and Analysis of Algorithm

Unit-5

String Matching

String Matching Problem

- We assume that the text is an array T[1..n] of length n and that the pattern is an array P[1..m] of length $m \leq n$.
- We further assume that the elements of P and T are characters drawn from a finite alphabet $\sum$.
- Pattern P ***occurs with shift*** s in text **T** if $0 \leq s \leq n-m$ and T [s+1 .. s+m] = P[1..m].
- If **P** occurs with shift s in **T** , then we call s a ***valid shift***; otherwise, we call s an ***invalid shift***.
- The ***string-matching problem*** is the problem of finding all valid shifts with which a given pattern P occurs in a given text T .

String Matching Problem

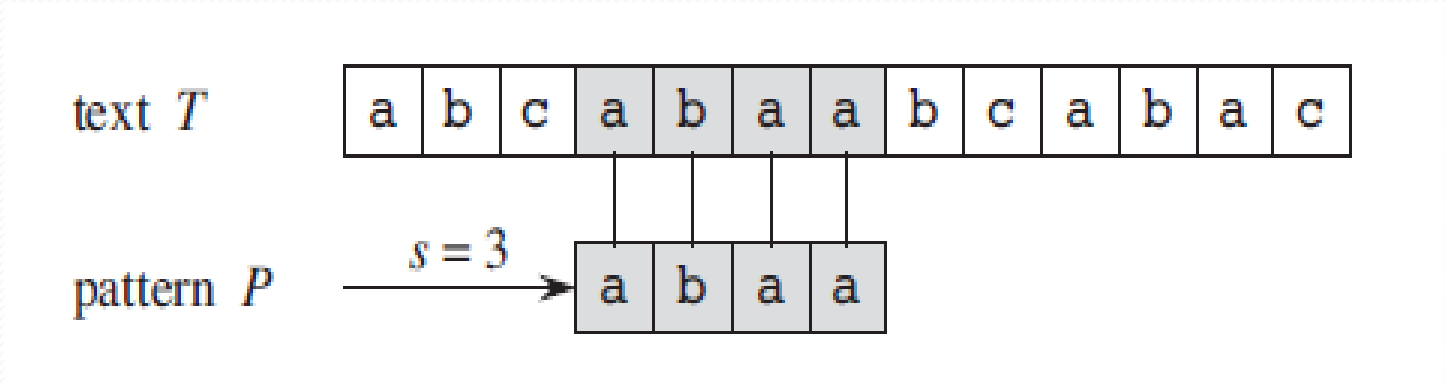
Example: Consider the text T and pattern P as following:-

T = abcabaabcabac

P= abaa

Find all valid shifts.

Solution:



Valid shift s = 3

There will be only one valid shift in this example.

Prefix and Suffix of a string

- **Prefix:** A string w is a ***prefix*** of a string x, denoted $w \sqsubset x$,
  if $x = wy$ for some string $y \in \Sigma^*$.

- **Suffix:** A string w is a ***suffix*** of a string x, denoted $w \sqsupset x$,
  if $x = yw$ for some string $y \in \Sigma^*$.

- Example: Clearly, `ab` $\sqsubset$ `abcca` and `cca` $\sqsupset$ `abcca`.

- The empty string ε is both a suffix and a prefix of every string.

The naive string-matching algorithm

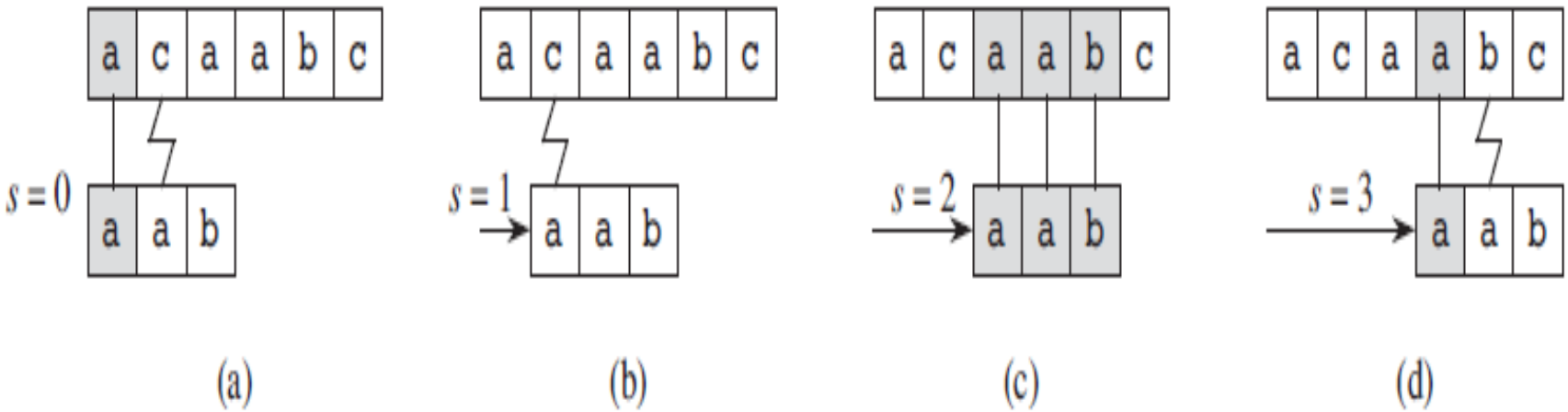
```
NAIVE-STRING-MATCHER(T, P)
n = T.length
m = P.length
for s = 0 to n − m
    if P[1..m] == T[s + 1..s + m]
        print "Pattern occurs with shift" s
```

- The worst-case running time is $\theta((n-m)m)$, which is $\theta(n^2)$ if $m = \lfloor n/2 \rfloor$.

The naive string-matching algorithm

Example: The operation of this algorithm is shown in the following:-

Here T = acaabc and P = aab



The Rabin-Karp algorithm

- Rabin and Karp proposed a string-matching algorithm that performs well.
- Given a pattern P{1..m], let p denote its corresponding decimal value. In a similar manner, given a text T[1..n], let t_s denote the decimal value of the length-m substring T [s+1 .. s+m] , for s= 0, 1,....., n-m.
- t_s = p iff T [s+1 .. s+m] = P[1..m]
- Therefore, s is a valid shift if and only if t_s = p.

The Rabin-Karp algorithm

Computation of p and t_s using Horner's rule:

- p = P[m]+10(P[m-1]+10(P[m-2]+10(P[m-3] + …. + 10(P[2]+10P[1]))))
- The value of t_0 can be computed similarly from T[1..m].
- To compute the remaining values $t_1, t_2, t_3, \ldots\ldots, t_{n-m}$, t_{s+1} can be computed from t_s in the following way:-

$$t_{s+1} = 10\,(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \quad \ldots\ldots\ldots\ldots (1)$$

- The only difficulty with this procedure is that p and t_s may be too large.

The Rabin-Karp algorithm

- To solve this problem, with d-ary alphabet {0,1,2,…., d-1}, we choose q so that dq fits with in a computer word and adjust the recurrence equation (1) to work modulo q, so that it becomes

$$t_{s+1} = (d (t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

where $h = d^{m-1} \bmod q$

- The solution of working modulo q is not perfect, because:

$t_s \equiv p \bmod q$ does not imply that $t_s = p$. On the other hand, if $ts \not\equiv p \bmod q$, then we definitely have that $t_s \neq p$, so that shift s is invalid.

- Any shift s for which $t_s \equiv p \bmod q$ must be tested further to see whether s is really valid or we just have a ***spurious hit***. This additional test explicitly checks the condition

P[1..m] = T[s+1……..s+m]

The Rabin-Karp algorithm

Example: Consider T and P as following:-

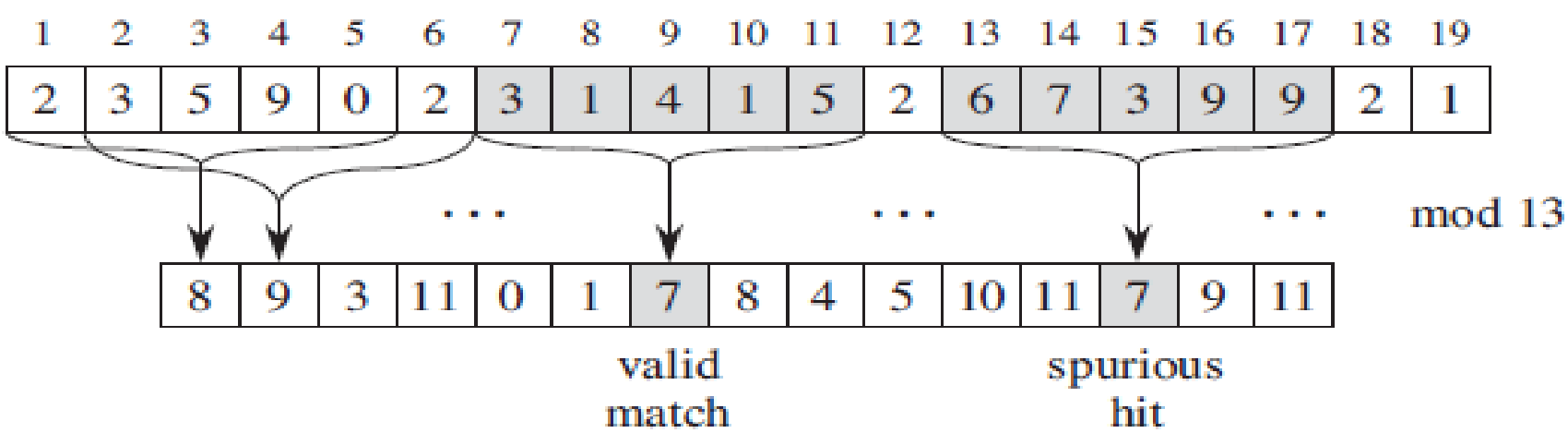
T= 2359023141526739921

P= 31415

q = 13

Find all valid shifts and spurious hit.

Solution:



The Rabin-Karp algorithm

```
RABIN-KARP-MATCHER(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m                    // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (dt_0 + T[i]) mod q
for s = 0 to n - m                // matching
    if p == t_s
        if P[1..m] == T[s + 1..s + m]
            print "Pattern occurs with shift" s
    if s < n - m
        t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

The Rabin-Karp algorithm

Time complexity

- RABIN-KARP-MATCHER takes ,$\theta(m)$ preprocessing time, and its matching time is , $\theta((n-m+1)m)$ in the worst case.

Question: For q=11, how many spurious hits does the Robin-Karp matcher encounter in the text T = 31415926535897 93 when looking for the pattern P= 26?

Solution:

Knuth-Morris-Pratt(KMP) algorithm

Prefix function for a pattern

Given a pattern P[1..m], the ***prefix function*** for the pattern P is the function $\pi : \{1,2,3,\ldots\ldots,m\} \rightarrow \{0,1,2,\ldots\ldots,m-1\}$ such that

$$\pi(q) = \max\{ k \,!\, K < q \text{ and } P_k \sqsupset P_q\}$$

$\pi(q)$ is the length of the longest prefix of P that is a proper suffix of P_q.

Knuth-Morris-Pratt(KMP) algorithm

Example: Compute the prefix function of the pattern

P = abababababca

Solution:

	1	2	3	4	5	6	7	8	9	10
i	a	b	a	b	a	b	a	b	c	a
P(i)	0	0	1	2	3	4	5	6	0	1
π(i)										

Knuth-Morris-Pratt(KMP) algorithm

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                   // number of characters matched
for i = 1 to n                          // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                        // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                       // next character matches
    if q == m                           // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                        // look for the next match
```

Knuth-Morris-Pratt(KMP) algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

Knuth-Morris-Pratt(KMP) algorithm

Time complexity

Running time of compute-prefix-function is $\theta(m)$.
The matching time of KMP-Matcher is $\theta(n)$.

Question: Consider text and pattern as following:-

T = bacbababaabcbab

P = aba

Find all valid shifts using KMP algo.

Question: Compute the prefix function for the pattern ababbabbabbababbabb.

Knuth-Morris-Pratt(KMP) algorithm

Prefix function for a pattern

Given a pattern P[1..m], the ***prefix function*** for the pattern P is the function $\pi : \{1,2,3,\ldots\ldots,m\} \rightarrow \{0,1,2,\ldots\ldots,m-1\}$ such that

$$\pi(q) = \max\{ k \,!\, K < q \text{ and } P_k \sqsupset P_q\}$$

$\pi(q)$ is the length of the longest prefix of P that is a proper suffix of P_q.

Knuth-Morris-Pratt(KMP) algorithm

Example: Compute the prefix function of the pattern

P = abababababca

Solution:

	1	2	3	4	5	6	7	8	9	10
i	a	b	a	b	a	b	a	b	c	a
P(i)	0	0	1	2	3	4	5	6	0	1
π(i)										

Knuth-Morris-Pratt(KMP) algorithm

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                   // number of characters matched
for i = 1 to n                          // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                        // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                       // next character matches
    if q == m                           // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                        // look for the next match
```

Knuth-Morris-Pratt(KMP) algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

Knuth-Morris-Pratt(KMP) algorithm

Time complexity

Running time of compute-prefix-function is $\theta(m)$.
The matching time of KMP-Matcher is $\theta(n)$.

Question: Consider text and pattern as following:-

T = bacbababaabcbab

P = aba

Find all valid shifts using KMP algo.

Question: Compute the prefix function for the pattern ababbabbabbababbabb.

AKTU Examination Questions

1. Write an algorithm for Naïve string matcher.
2. Write KMP algorithm for string matching. Perform the KMP algorithm to search the occurrences of the pattern abaab in the text string abbabaabaabab.
3. Write Rabin Karp string matching algorithm. Working modulo q=11, how many spurious hits does the Rabin karp matcher in the text T= 3141592653589793, when looking for the pattern P=26.
4. Explain and Write the Knuth-Morris-Pratt algorithm for pattern matching also write its time complexity.
5. Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function π for the pattern ababbabbabbababbabb when the alphabet is $\Sigma = \{a,b\}$.
6. Compute the prefix function π for the pattern P= a b a c a b using KNUTHMORRIS –PR

Approximation Algorithms

Approximation Algorithms

- An algorithm that returns near-optimal solutions is said to be approximation algorithm.
- This technique does not guarantee the best solution.
- The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time.
- Approximation algorithms are designed to get the solution of NP-complete problems in polynomial time.
- If we work on an optimization problem where every solution carries a cost, then an approximation algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

Approximation Ratio

- Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.
- An algorithm for a problem has an **approximation ratio** of P(n) for any input of size n, if the cost C of the solution produced by the algorithm is within a factor of P(n) of the cost C* of an optimal solution i.e.

$$\max(C/C^*, C^*/C) \leq P(n)$$

- If an algorithm achieves an approximation ratio of P(n), we call it a **P(n)-approximation algorithm**.

Approximation Ratio(cont.)

- The approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.
- Observe that $P(n)$ is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution.
- For a minimization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of the optimal solution is larger than the cost of approximate solution.
- Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of optimal solution.

Vertex Cover Problem

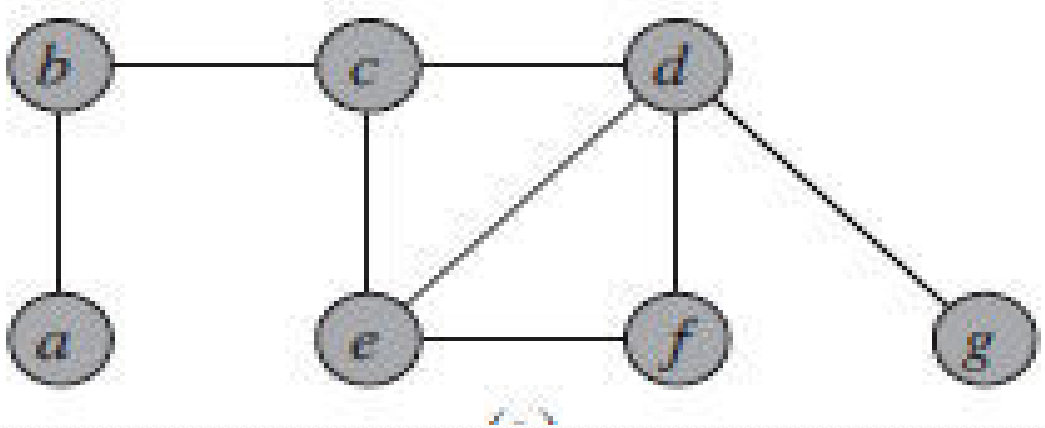
- A **vertex cover** of an undirected graph G =(V, E) is a subset $V' \subseteq V$ such that if (u,v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both).
- The size of a vertex cover is the number of vertices in it.
- The **vertex-cover problem** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an optimal vertex cover.
- This problem is the optimization version of an NP-complete decision problem.

Approximation Algorithm for Vertex Cover Problem

```
APPROX-VERTEX-COVER(G)
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```

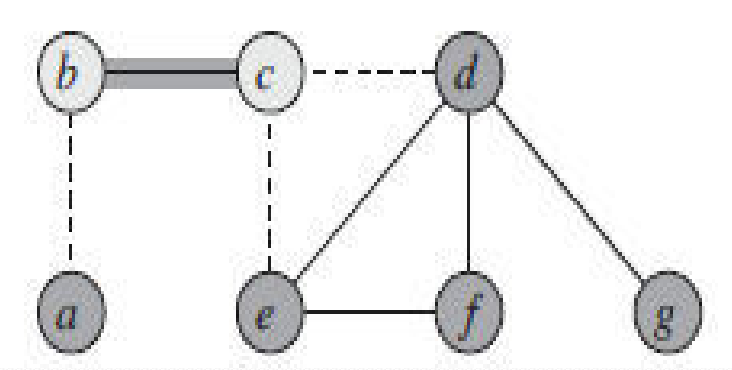
Approximation Algorithm for Vertex Cover Problem

Ex. Consider the following graph:-

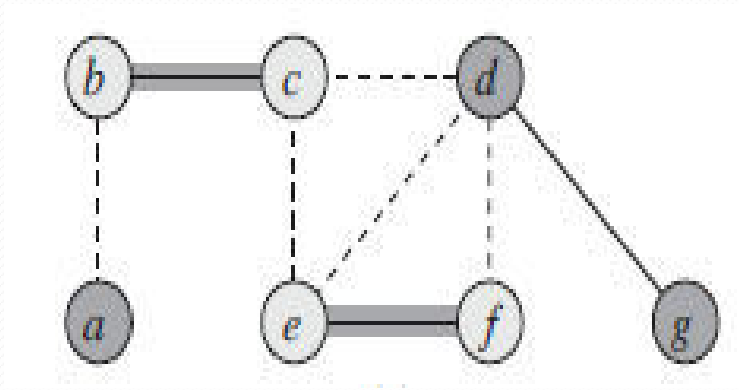


Find the optimal vertex cover of this graph.

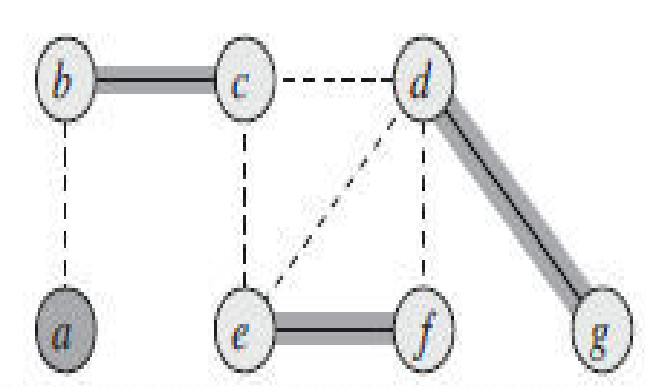
Solution



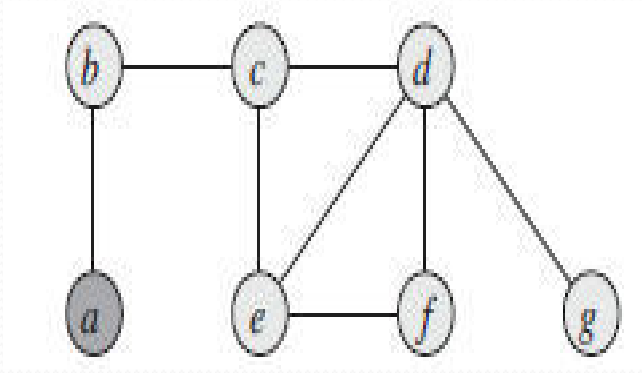
(a)



(b)

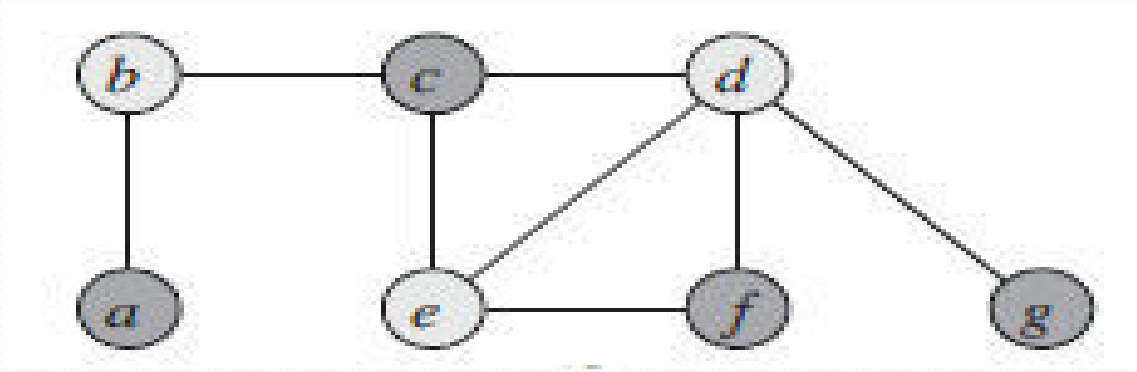


(c)



(d)

Approximation Algorithm for Vertex Cover Problem



Optimal vertex cover for this problem contains only three vertices: b, d, and e.

Note: The running time of this algorithm is O(E+V) using adjacency lists to represent E'.

Traveling-salesman problem

- In the traveling-salesman problem, we are given a complete undirected graph G(V,E) that has a nonnegative integer cost c(u,v) associated with each edge (u,v) ∈ E, and we must find a Hamiltonian cycle (a tour) of G with minimum cost.
- Let c(A) denote the total cost of the edges in the subset $A \subseteq E$.

$$c(A) = \sum_{(u,v) \in A} c(u,v)$$

- Cost function c satisfies the ***triangle inequality*** *if, for all vertices* u,v,w ∈*V,*

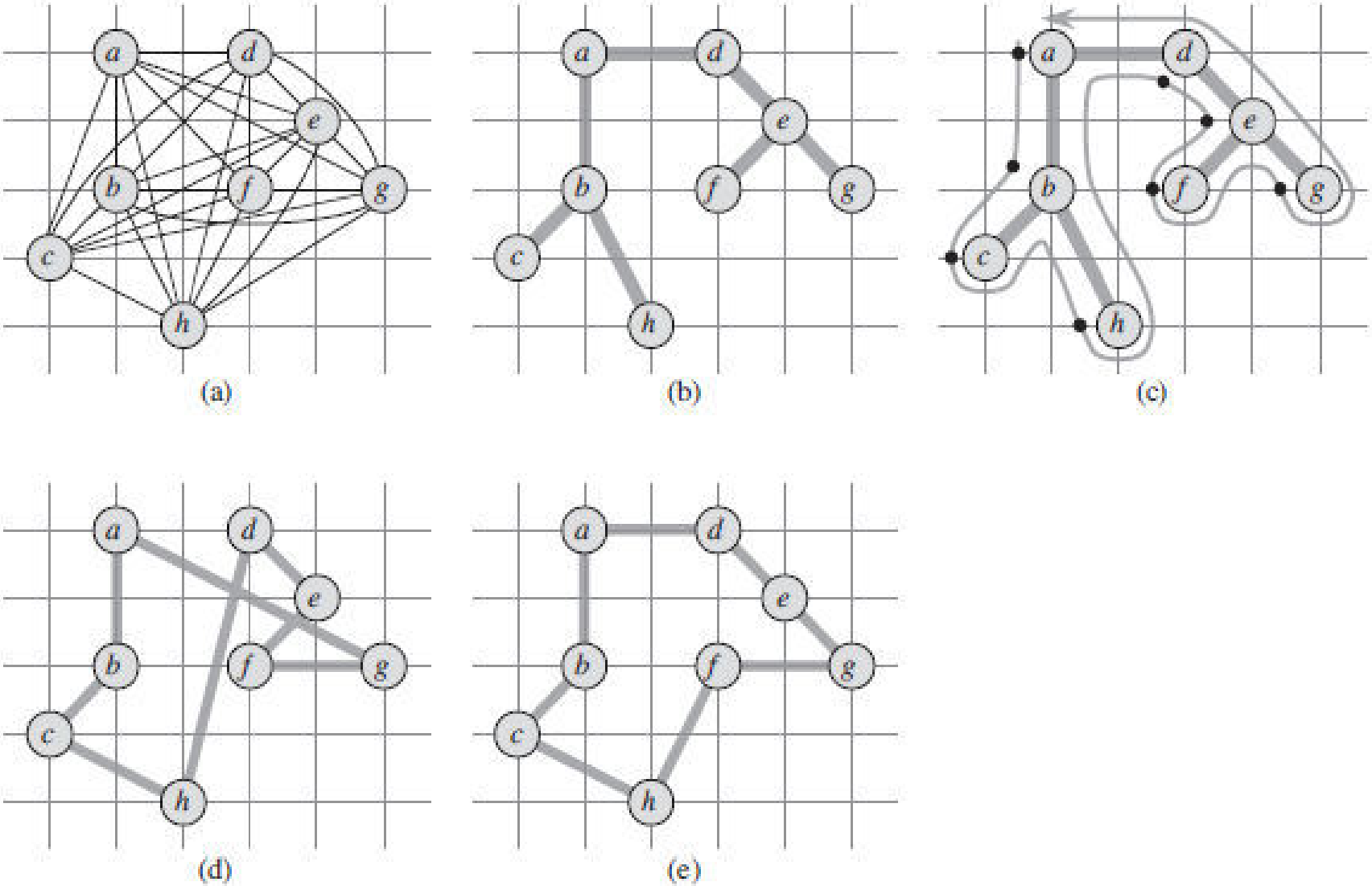
$$c(u,v) \leq c(u,v) + c(v,w)$$

Traveling-salesman problem with the triangle inequality

- The following algorithm computes a near-optimal tour of an undirected graph G, using the minimum-spanning-tree algorithm MST-PRIM.
- Here, the cost function satisfies the triangle inequality.
- The tour that this algorithm returns is no worse than twice as long as an optimal tour.

```
APPROX-TSP-TOUR(G, c)
select a vertex r ∈ G.V to be a "root" vertex
compute a minimum spanning tree T for G from root r
    using MST-PRIM(G, c, r)
let H be a list of vertices, ordered according to when they are first visited
    in a preorder tree walk of T
return the hamiltonian cycle H
```

Traveling-salesman problem with the triangle inequality



Traveling-salesman problem with the triangle inequality

(a) **A complete undirected graph. Vertices lie** on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance.

(b) **A minimum spanning** tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order.

(c) **A walk of T , starting at a. A** full walk of the tree visits the vertices in the order a, b, c, b, h, b, a, d, e, f, e, g, e, d, a. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g.

Traveling-salesman problem with the triangle inequality

(d) **A tour obtained by visiting the vertices in the order** given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074.

(e) **An optimal tour H for the original complete graph. Its total cost is** approximately 14.715.

Note: The running time of APPROX-TSP-TOUR is $O(V^2)$.

NP-Completeness

Classes P and NP

- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.
- The class NP consists of those problems that are "verifiable" in polynomial time.
- What do we mean by a problem being verifiable? If we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

Classes P and NP

- For example, in the hamiltonian cycle problem, given a directed graph G(V,E), a certificate would be a sequence < $v_1, v_2, v_3, \ldots\ldots, v_n$ > of n vertices. We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for i = 1, 2, 3,……, n-1 and that $(v_n, v_1) \in E$ as well.

- Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate.

NP-Hard and NP-complete(NPC)

NP-Hard: A problem L is said to be NP-hard if every problems belong in to NP is polynomial time reducible to L.

That is,

Problem L is NP-hard if for all problems $L' \in NP$,

$$L' \leq_p L.$$

That is, if we can solve L in polynomial time, then we can solve all NP problems in polynomial time.

NP-Complete

NP-Complete: Problem L is NP-complete if

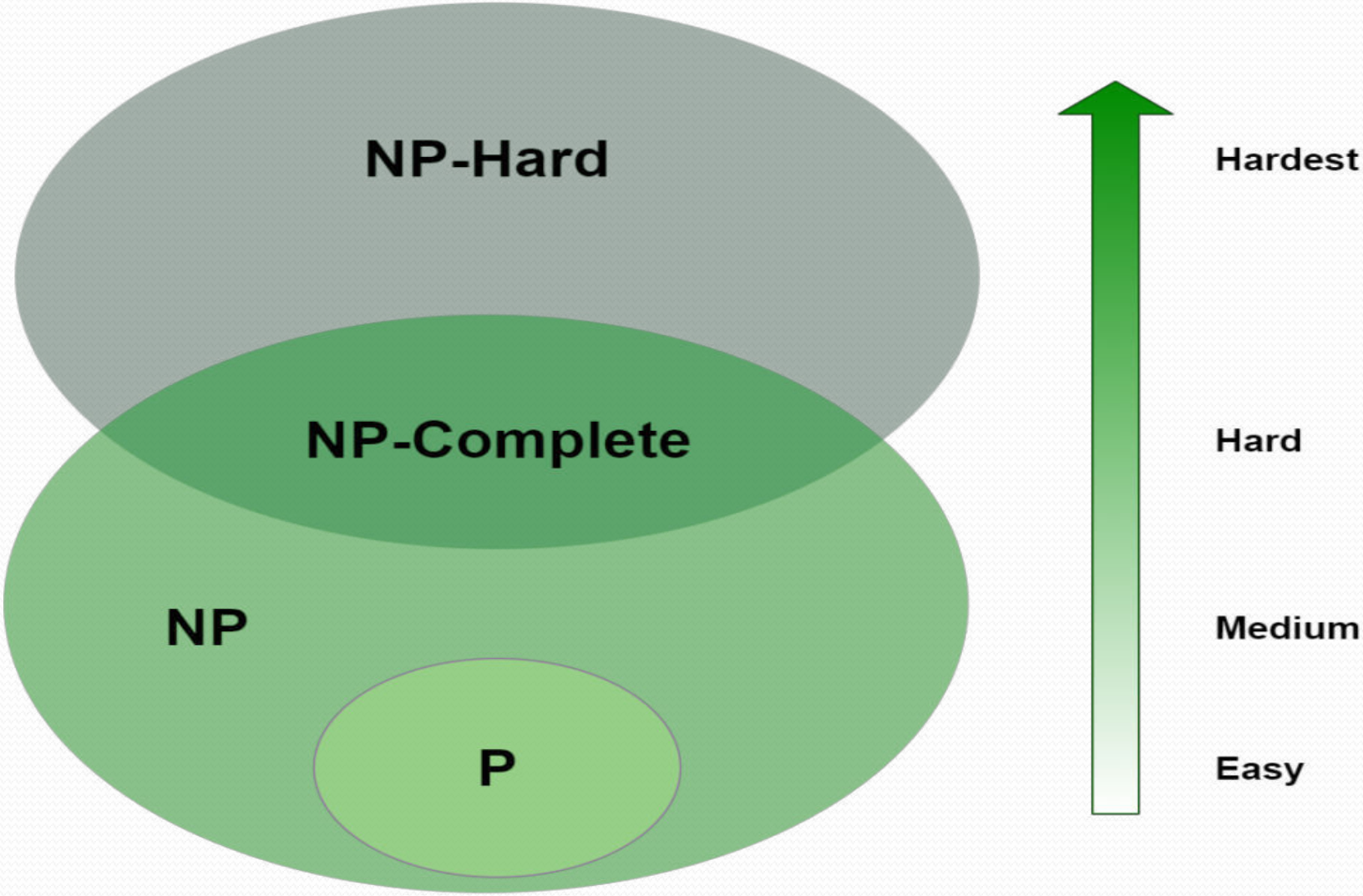
1. L ϵ NP and
2. L is NP-hard

NP-Complete problems:

- Boolean satisfiability problem (SAT)
- Hamiltonian cycle problem.
- Travelling salesman problem
- Vertex cover problem

Note: If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time.

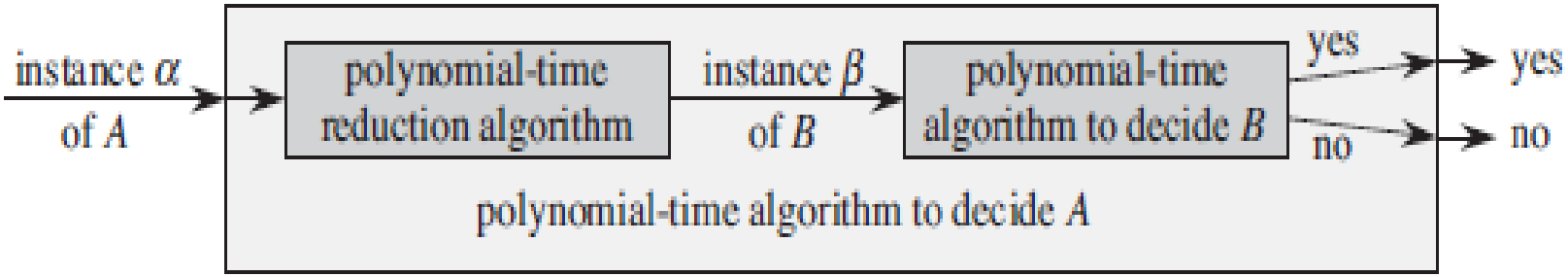
Relationship between P, NP, NP-hard and NPC



Polynomial-time ***Reduction Algorithm***

- Consider a decision problem A, which we would like to solve in polynomial time.
- We call the input to a particular problem an ***instance of that problem.***
- Suppose that we already know how to solve a different decision problem B in polynomial time.
- Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:
  - The transformation takes polynomial time.
  - The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."
- We call such a procedure a **polynomial-time *reduction algorithm.***

Polynomial-time ***Reduction Algorithm(cont.)***



Polynomial-time reduction algorithm provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

Lemma

If L is a language such that $L' \leq_P L$ for some $L' \in NPC$, then L is NP-hard. If, in addition, $L \in NP$, then $L \in NPC$.

Note: The hamiltonian cycle problem is NP-complete.

Note: Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Note: The vertex-cover problem is NP-complete.

Theorem: Show that the traveling-salesman problem is NP-complete.

Proof:

We first show that TSP belongs to NP.

Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE$\leq_P$TSP.

Let G=(V,E) be an instance of HAM-CYCLE. We construct an instance of TSP as follows.

We form the complete graph G'=(V,E'), where

$E' = \{(i,j) \mid i,j \in V \text{ and } i \neq j\}$ and

we define the cost function c by:-

$c(i,j) = 0 \quad \text{if} \quad (i,j) \in E$

$\quad\quad = 1 \quad \text{if} \quad (i,j) \notin E$

The instance of TSP is then < G',c,0 >, which we can easily create in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0.

Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We conclude that h' is a hamiltonian cycle in graph G.

AKTU Examination Questions

1. Write and explain the algorithm to solve vertex cover problem using approximation algorithm.
2. Explain NP-complete and NP-Hard.
3. Explain Randomized algorithm in brief.
4. What is an approximation algorithm? What is meant by P(n) approximation algorithms? Discuss approximation algorithm for Travelling Salesman Problem.
5. Define NP-Hard and NP- complete problems. What are the steps involved in proving a problem NP-complete? Specify the problems already proved to be NP-complete.

AKTU Examination Questions

6. What are approximation algorithms? What is meant by P(n) approximation algorithms?
7. Define NP, NP hard and NP Complete Problems. Prove that Travelling Salesman Problem is NP-Complete.
8. What is the application of Fast Fourier Transform (FFT)? Also write the recursive algorithm for FFT.
9. Discuss the problem classes P, NP and NP –complete with class relationship.
10. Explain Approximation and Randomized algorithms.
11. What do you mean by polynomial time reduction?
12. Explain applications of FFT.