Design and Analysis of Algorithms

Greedy Method

- It's a straight forward design technique that can be applied to a wide variety of problems
- Coin change problem
  - Let $A_n = \{a_1, a_2, a_3, \ldots, a_n\}$ be a finite set of distinct coin types (for example, $a_1 = 5p$, $a_2 = 10p$, $a_3 = 20p$, $a_4 = 25p$, $a_5 = 50p$ and so on.)
  - Assume each a_i is an integer and $a_1 > a_2 > a_3 \ldots\ldots a_n$.
  - Each type is available in unlimited quantity.
  - The coin-changing problem is to make up an exact amount C using a minimum total number of coins.
  - C is an integer >0.

Greedy Method

- Lets take an example
  - X goes to a shop to buy an item worth 30p and hands over 1Rs.
  - What is the change expected by X? 70p
  - Probable set of coins given by shop keeper
    - A={10, 20, 20, 20}
    - B={10, 10, 10, 10 ,10 ,10, 10}
    - C={5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
    - D={25 ,25 ,20}
    - E={50, 20}
    - F= {50, 10, 10}
- What was the constraint?
  - The sum of the coins given by the shopkeeper =70p
- The subset that satisfies this constraint is called feasible solution
- The feasible solution that maximizes/minimizes the objective function is called an optimal solution

Greedy Method

- Greedy algorithms work in stages
  - At each stage, a decision is made regarding whether a particular input is in an optimal solution.
  - This is done by considering the inputs in an order determined by some selection procedure.
  - If the inclusion of the next input into the partially constructed optimal
  - Solution will result in an infeasible solution, then this input is not added to the partial solution, otherwise, it is added

Greedy Method

Control Abstraction

```
Algorithm Greedy(a, n)
// a[1:n] contains the n inputs.
{
        solution:=0;// Initialize the solution.
        for i :=1 to n do
         {
            x :=Select(a);
            if Feasible(solution, x) then
                solution:=Union (solution, x);
         }
        return solution;
}
```

Greedy Method

Knapsack problem

- Given n objects, with each object i having a positive weight w_i and the profit p_i
- A knapsack / bag with a capacity m
- If a fraction x_i, $0 < xi < 1$, of object i is placed into the knapsack, then a profit of $p_i x_\mathrm{i}$ is earned
- The objective is to find the set of objects which fills the knapsack and maximizes the total profit
- The total weight of the selected objects can be at most m

Greedy Method

Formal definition of Knapsack problem

$$\text{maximize} \quad \sum_{1 \le i \le n} p_i \, x_i$$

$$\text{subject to} \quad \sum_{1 \le i \le n} w_i \, x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

xi = 1 when item i is selected and let xi = 0 when item i is not selected.

Greedy Method

Example

Consider the following instance of the knapsack problem

n = 3, m= 20, (p_1,p_2,p_3)=(25,24,15) and (w_1,w_2,w_3)= (18,15,10).

Find all the feasible solutions and hence an optimal solution

First feasible solution

- Consider the objects according to their profit (descending order)
- Object with largest profit is considered first and so on
- Object 1 (highest profit) with weight 18 can be put into the knapsack in its entirety . So x1=1 and RemCap =2
- Object 2 (second highest profit) with weight 15 is considered but cant be put into the bag in its entirety (RemCap<w2)
- Hence a fraction 2/15 is put into the knapsack, x2=2/15, RemCap =0

Greedy Method

– Remaining objects cant be put into the knapsack

$$\sum_{i=1 \text{ to } 3} w_i x_i = 18 \times 1 + 15 \times {}^{2}/_{15} + 0 = 20$$

$$\sum_{i=1 \text{ to } 3} p_i x_i = 25 \times 1 + 24 \times {}^{2}/_{15} + 0 = 28.2$$

The solution vector $(x_1, x_2, x_3) = (1, {}^{2}/_{15}, 0)$

Greedy Method

Second feasible solution

- Consider the objects according to their weights (increasing order)
- Object with smallest weight is considered first and so on
- Object 3 (smallest weight) with weight 10 can be put into the knapsack in its entirety . So x3=1 and RemCap =10
- Object 2 (second smallest weight) with weight 15 is considered but cant be put into the bag in its entirety (RemCap<w2)
- Hence a fraction 10/15 , i.e. 2/3 is put into the knapsack, x2=2/3, RemCap =0
- Remaining objects cant be put into the knapsack

$$\sum_{i=1 \text{ to } 3} w_i x_i = 0 + 15 \times {}^{2}/_{3} + 10 \times 1 = 20$$

$$\sum_{i=1 \text{ to } 3} p_i x_i = 0 + 24 \times {}^{2}/_{3} + 15 \times 1 = 31$$

The solution vector $(x_1, x_2, x_3) = (0,\ {}^{2}/_{3}\ , 1)$

Greedy Method

Third feasible solution

- Consider the objects according to the ratio ${p_i}/{w_i}$ (decreasing order)
- ${p_1}/{w_1} = 1.38$ ${p_2}/{w_2} = 1.6$ ${p_3}/{w_3}\ 1.5$
- Object with highest ratio is considered first and so on
- Object 2 (highest ratio) with weight 15 can be put into the knapsack in its entirety . So x2=1 and RemCap =5
- Object 3 (second highest ratio) with weight 10 is considered but cant be put into the bag in its entirety (RemCap<w3)
- Hence a fraction 5/10, i.e. 1/2 is put into the knapsack, x3=1/2, RemCap =0
- Remaining objects cant be put into the knapsack

$$\sum_{i=1 \text{ to } 3} w_i x_i = 0 + 15 \times 1 + 10 \times {1}/{2} = 20$$

$$\sum_{i=1 \text{ to } 3} p_i x_i = 0 + 24 \times 1 + 15 \times {1}/{2} = 31.5$$

The solution vector $(x_1, x_2, x_3) = (0, 1, {1}/{2})$

Greedy Method

- First feasible solution =28.2
- Second feasible solution =31
- Third feasible solution =31.5
- Since the objective function is to maximize the profit, the optimal solution is
  - Profit =31.5
  - The solution vector $(x_1, x_2, x_3) = (0, 1, 1/2)$

Greedy Method

Homework

Consider the following instance of the knapsack problems.

- Find all the feasible solutions and hence an optimal solution

  $n = 3$, $m = 20$, $(p_1,p_2,p_3)=(30, 21, 18)$ and $(w_1,w_2,w_3)= (18,15,10)$.

- Find all the feasible solutions and hence an optimal solution

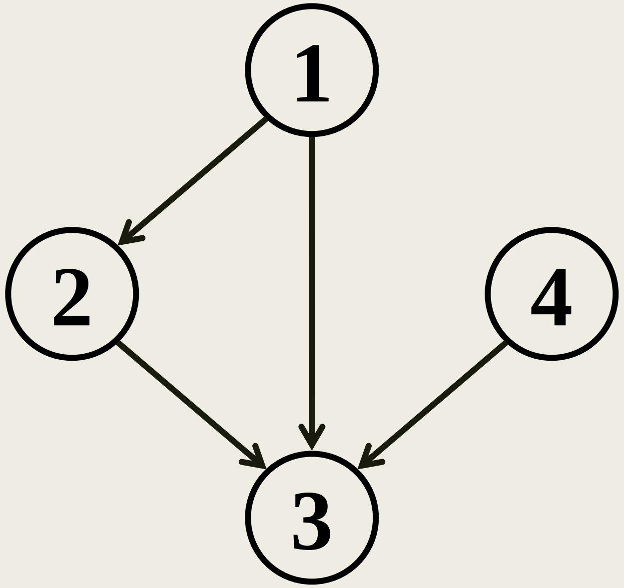
  N=7, m=15, pi=(10,5,15,7,6,18,3) and wi=(2,3,5,7,1,4,1)

Greedy Method

```
ALGORITHM Greedy_Knapsack(m,n)
{
for( i = 1 to n ) do   x[ i ] = 0.0
u=m
for( i = 1 to n ) do
  {
     if( w[ i ]>u ) then break
     x[ i ] = 1.0
     u=u-w[ i ]
  }
if( i <= n ) then
x[ i ] = u/w[ i ]
}
```

Introduction to Graphs

- **A graph G = (V, E)**
  - *V = set of vertices*
  - *E = set of edges = subset of V × V*
  - *Thus $|E| = O(|V|^2)$*

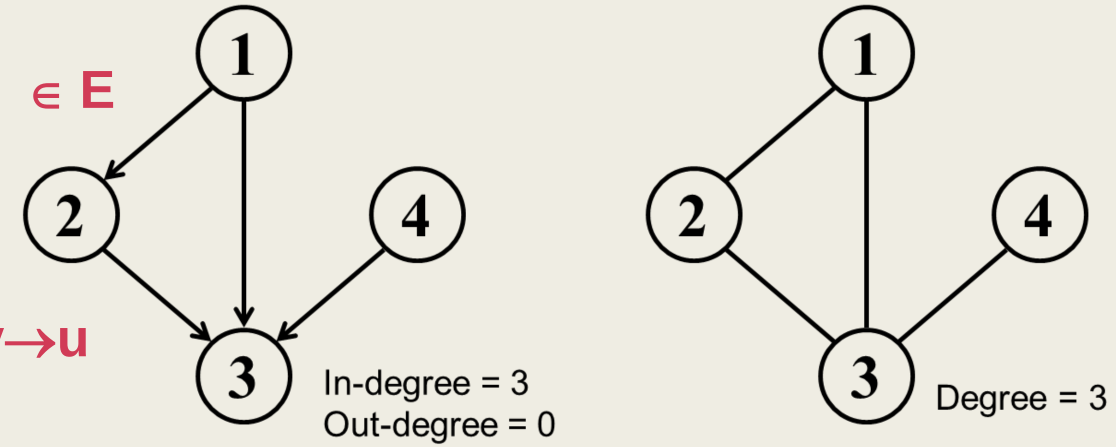


Vertices: {1, 2, 3, 4}

Edges: {(1, 2), (2, 3), (1, 3), (4, 3)}

Graph – Variations

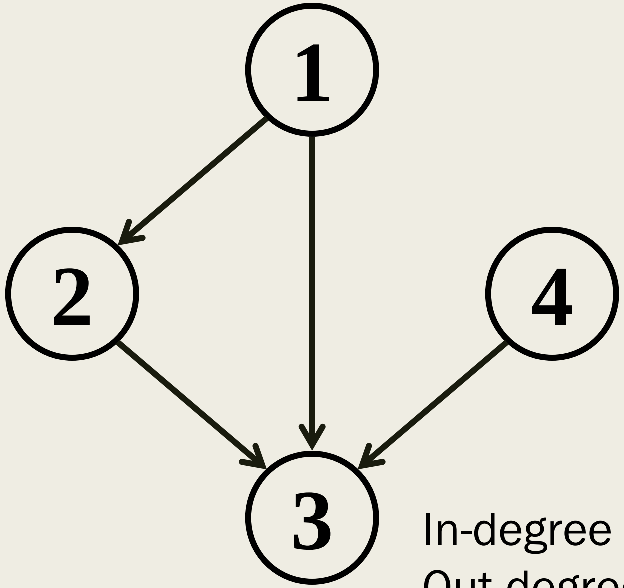
- **Directed / undirected:**
  - **In an undirected graph:**
    - Edge (u,v) $\in$ E implies edge (v,u) $\in$ E
    - Road networks between cities
  - **In a directed graph:**
    - Edge (u,v): u→v does not imply v→u
    - Street networks in downtown
  - **Degree of vertex v:**
    - The number of edges adjacency to v
    - For directed graph, there are in-degree and out-degree



Directed

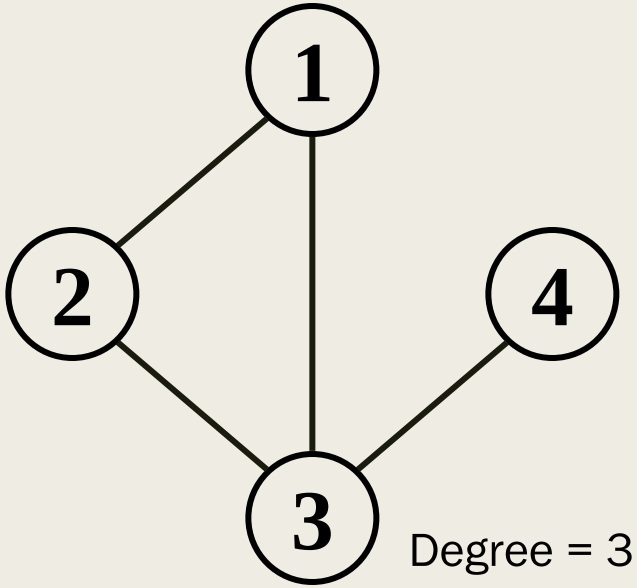
Undirected

Graph – Variations



In-degree = 3
Out-degree = 0

Directed

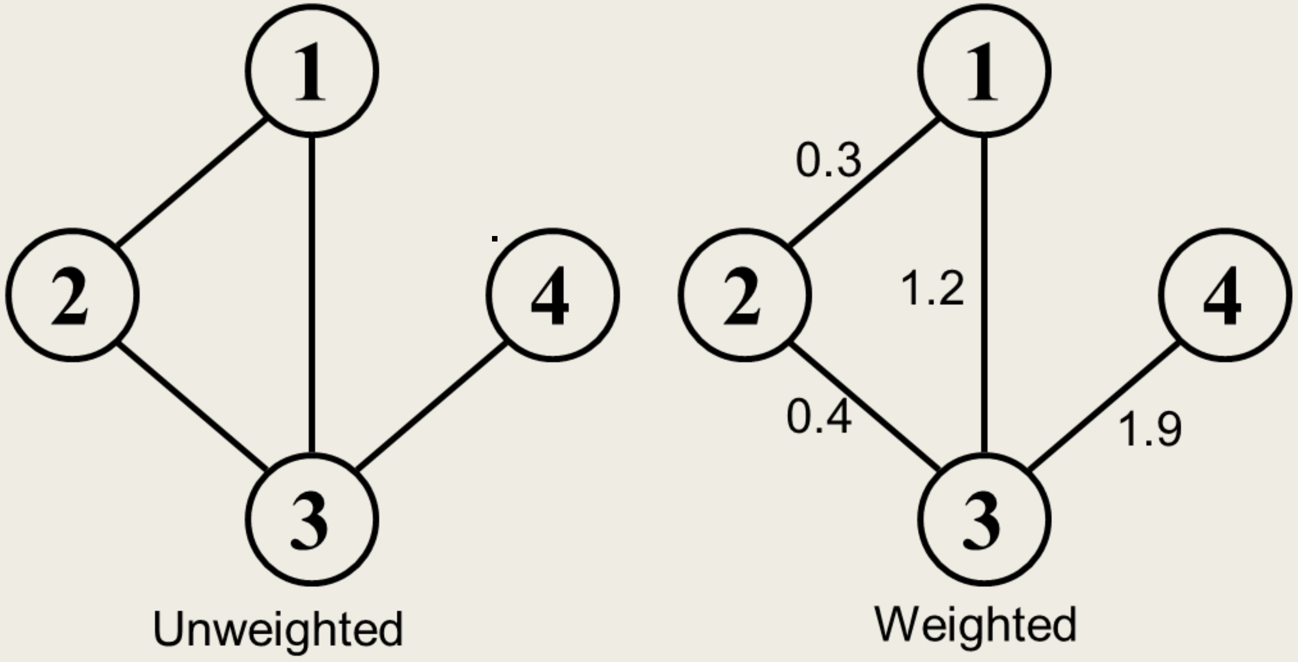


Degree = 3

Undirected

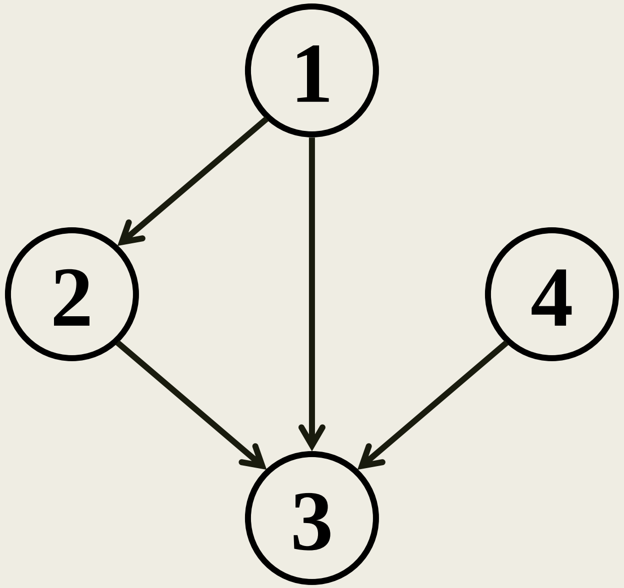
Graph – Variations

- **Weighted / unweighted:**
  - ***In a weighted graph, each edge or vertex has an associated weight (numerical value)***
    - E.g., a road map: edges might be weighted w/ distance

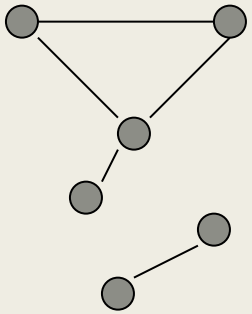


Graph – Variations

- **Connected / disconnected:**
  - *A connected graph has a path from every vertex to every other*
  - *A directed graph is strongly connected if there is a directed path between any two vertices*



Connected but not strongly connected



Graph – Variations

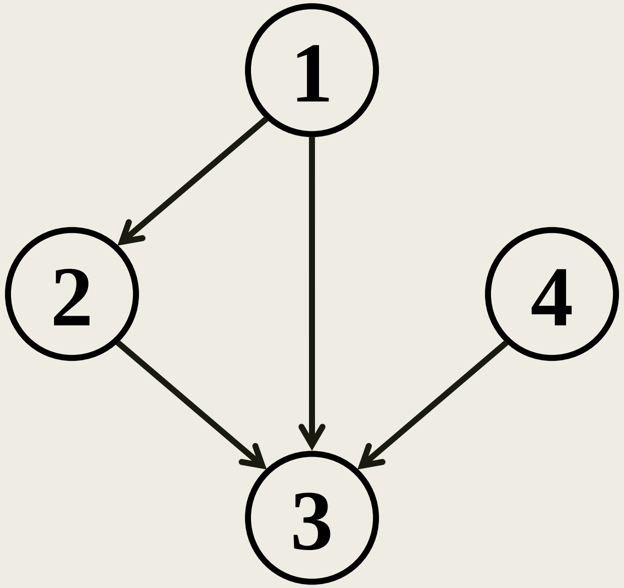
- **Dense / sparse:**
  - ***Graphs are sparse when the number of edges is linear to the number of vertices***
    - $|E| \in O(|V|)$
  - ***Graphs are dense when the number of edges is quadratic to the number of vertices***
    - $|E| \in O(|V|^2)$
  - ***Most graphs of interest are sparse***
  - ***If you know you are dealing with dense or sparse graphs, different data structures may make sense***

Representing Graphs

- Assume V = {1, 2, …, n}
- An **adjacency matrix** represents the graph as a n x n matrix A:
  - $A[i, j]$ = 1 if edge $(i, j) \in E$
    = 0 if edge $(i, j) \notin E$
- For weighted graph
  - $A[i, j]$ = w_{ij} if edge $(i, j) \in E$
    = 0 if edge $(i, j) \notin E$
- For undirected graph
  - Matrix is symmetric: $A[i, j] = A[j, i]$

Graphs: Adjacency Matrix

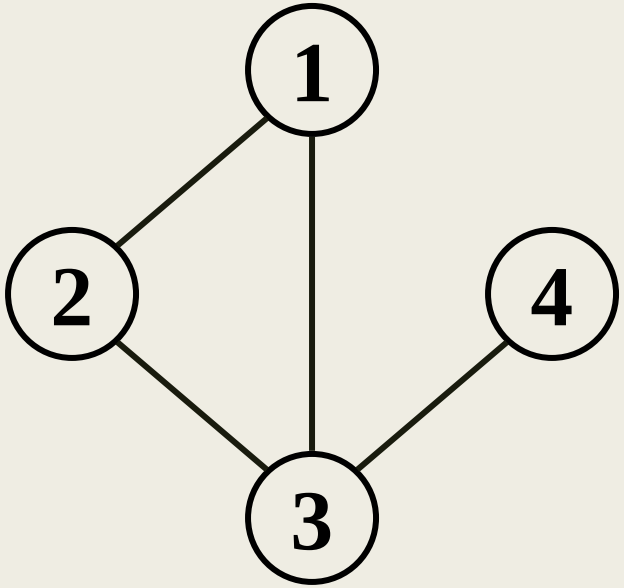
- **Example:**



A	1	2	3	4
1				
2				
3			??	
4				

Graphs: Adjacency Matrix

- **Example**

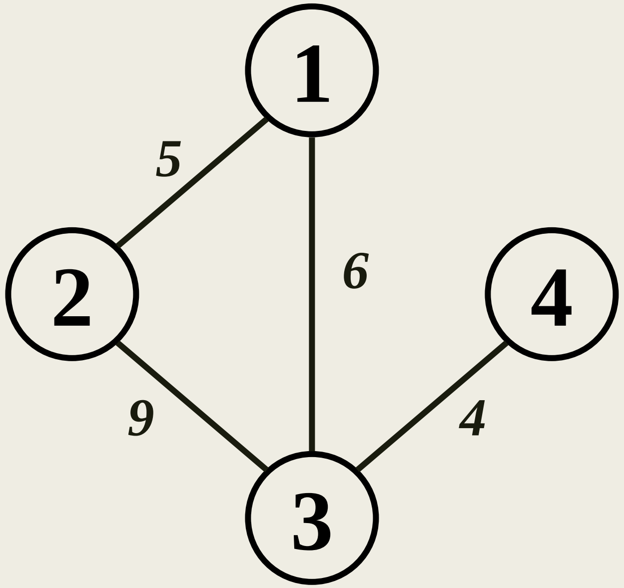


Undirected graph

A	1	2	3	4
1	0	1	1	999
2	1	0	1	0
3	1	1	0	1
4	999	0	1	0

Graphs: Adjacency Matrix

- **Example**

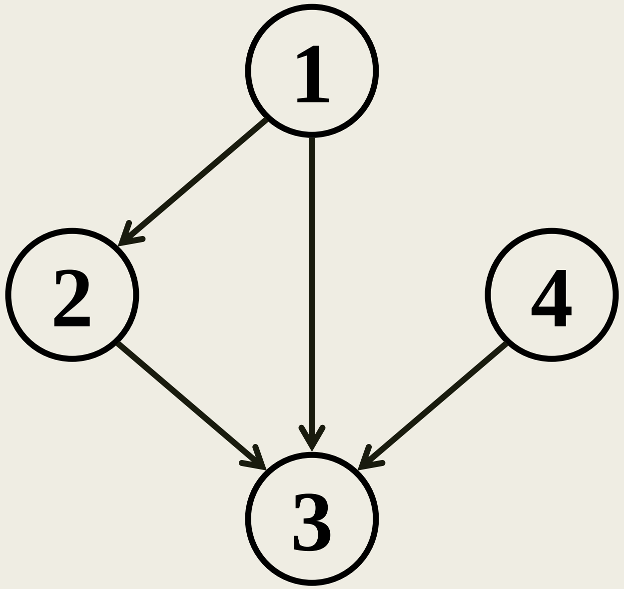


Weighted graph

A	1	2	3	4
1	0	5	6	999
2	5	0	9	0
3	6	9	0	4
4	999	0	4	0

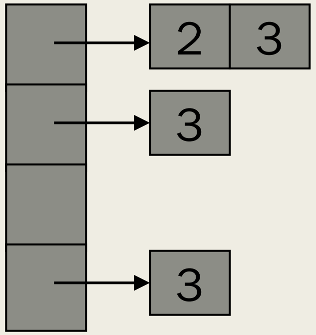
Graphs: Adjacency List

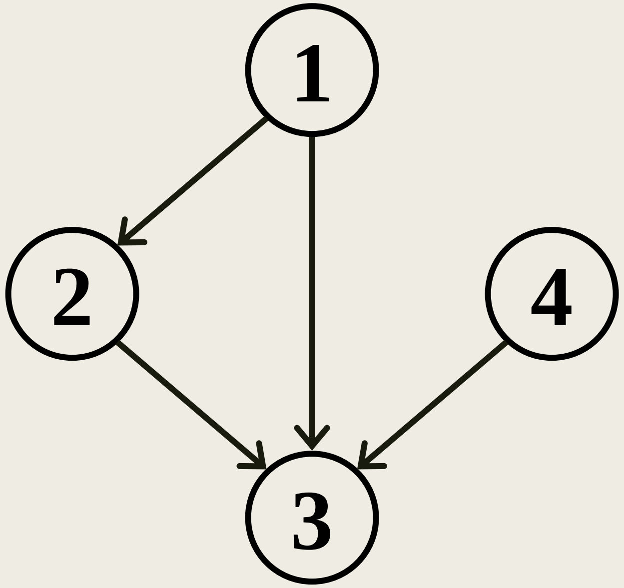
- Adjacency list: for each vertex v ∈ V, store a list of vertices adjacent to v
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming into vertex



Graphs: Adjacency List

- **Adjacency list**



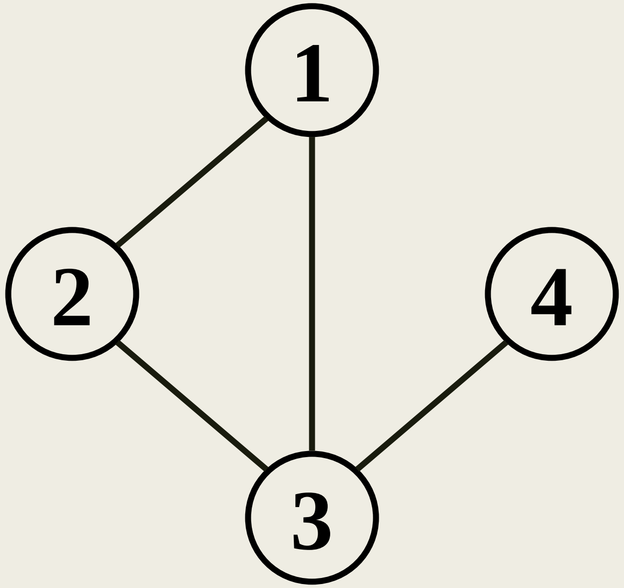


How much storage does the adjacency list require?

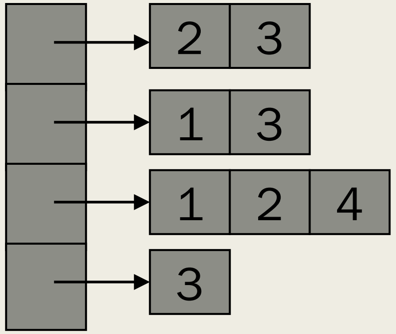
A: O(V+E)

Graphs: Adjacency List

- **Undirected graph**

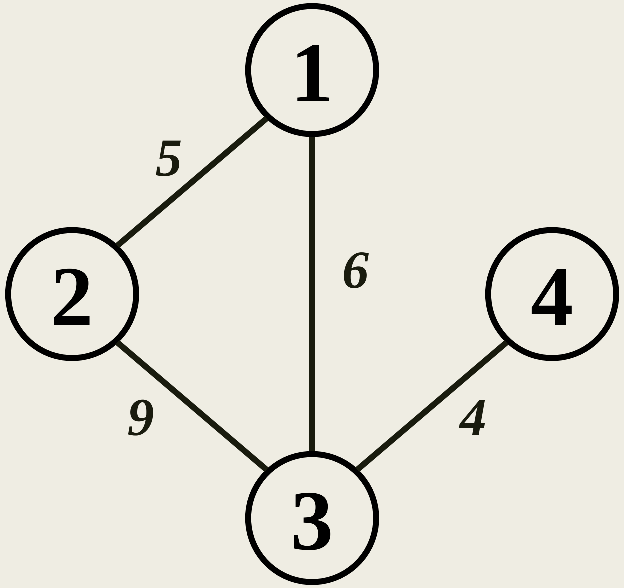


A	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

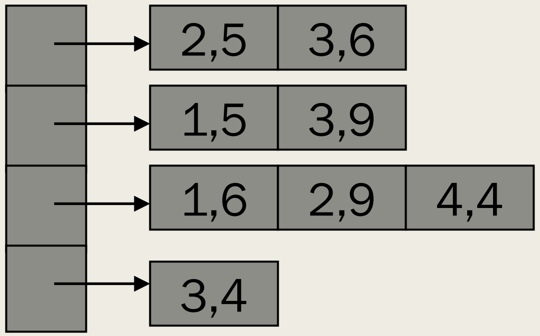


Graphs: Adjacency List

- **Weighted graph**



A	1	2	3	4
1	0	5	6	0
2	5	0	9	0
3	6	9	0	4
4	0	0	4	0



Trade of between two representations

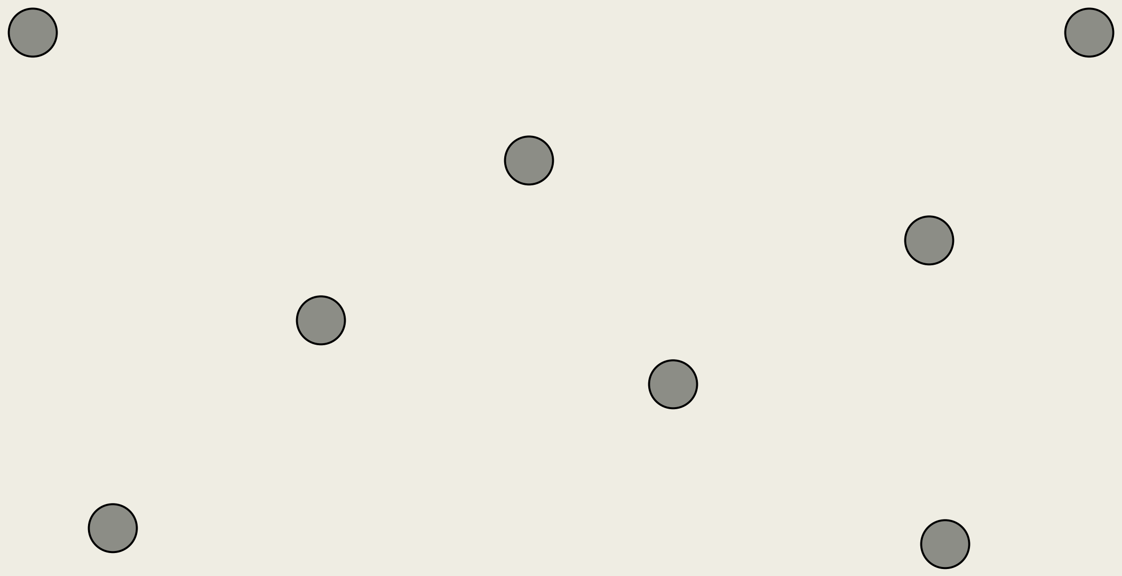
- |V| = n, |E| = m

	Adj Matrix	Adj List
test $(u, v) \in E$	$\Theta(1)$	$O(n)$
Degree(u)	$\Theta(n)$	$O(n)$
Memory	$\Theta(n^2)$	$\Theta(n+m)$
Edge insertion	$\Theta(1)$	$\Theta(1)$
Edge deletion	$\Theta(1)$	$O(n)$
Graph traversal	$\Theta(n^2)$	$\Theta(n+m)$

Both representations are very useful and have different properties, although adjacency lists are probably better for more problems

Problems

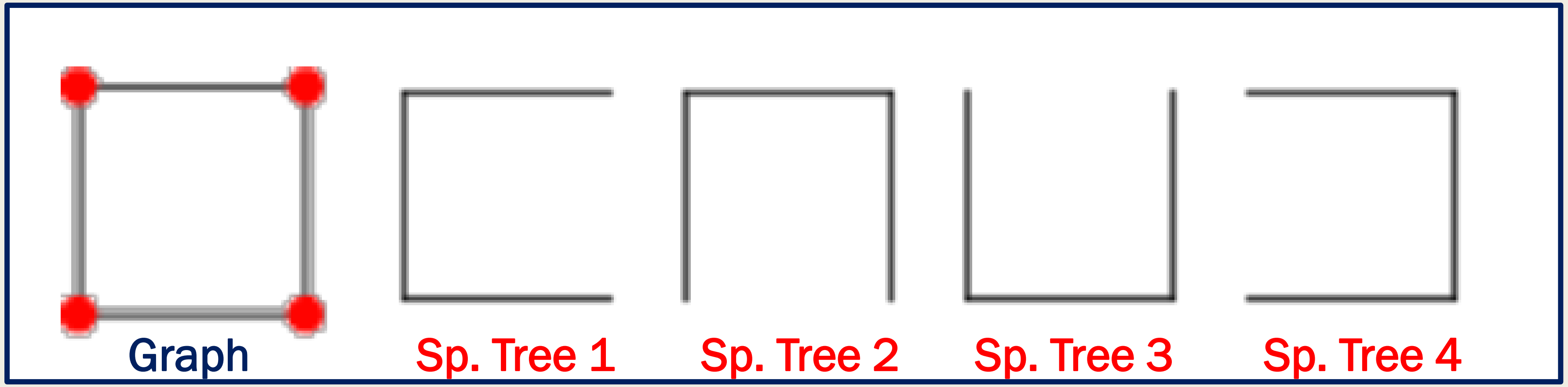
- **Given a set of cities, how to construct minimum length of highways to connect the cities so that there are paths between every two cities?**



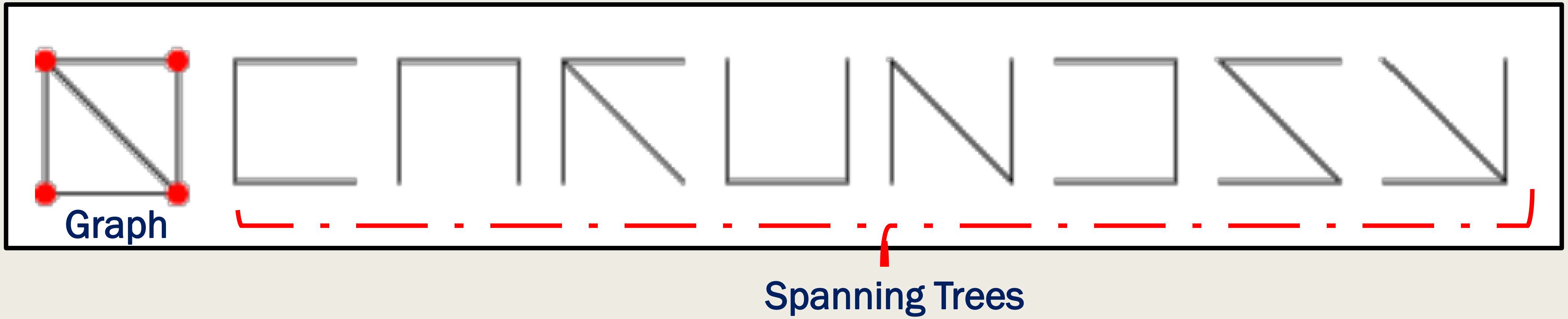
Spanning Tree

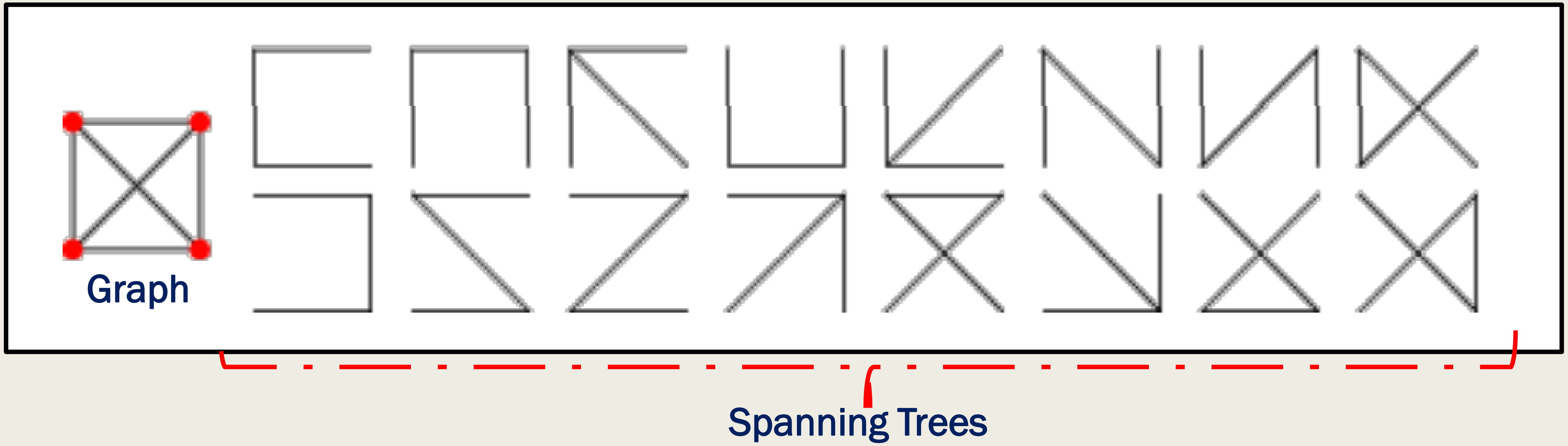
Definition

- Let G = (V, E) be an undirected connected graph.
- A subgraph t = (V, E’) of G is a spanning tree of G iff t is a tree



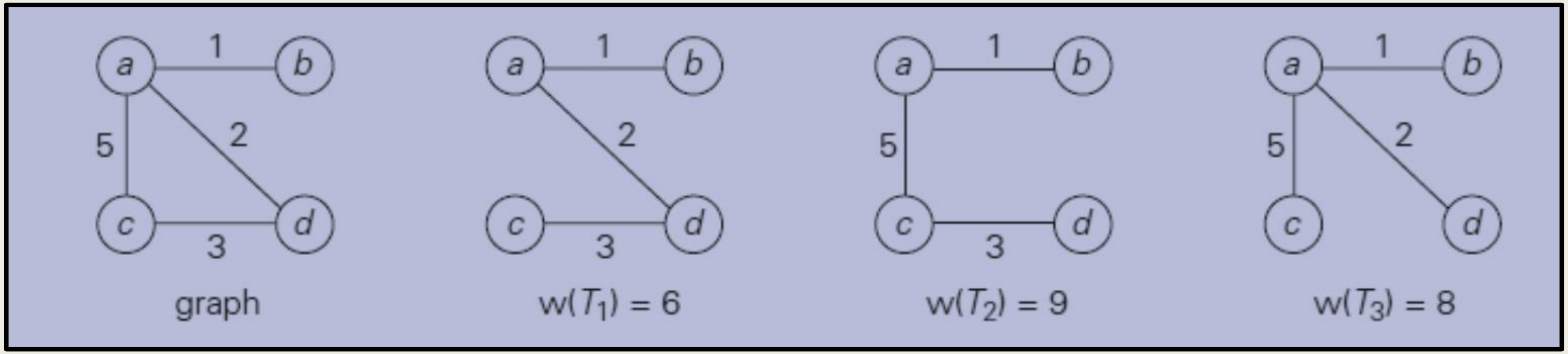
Spanning Tree



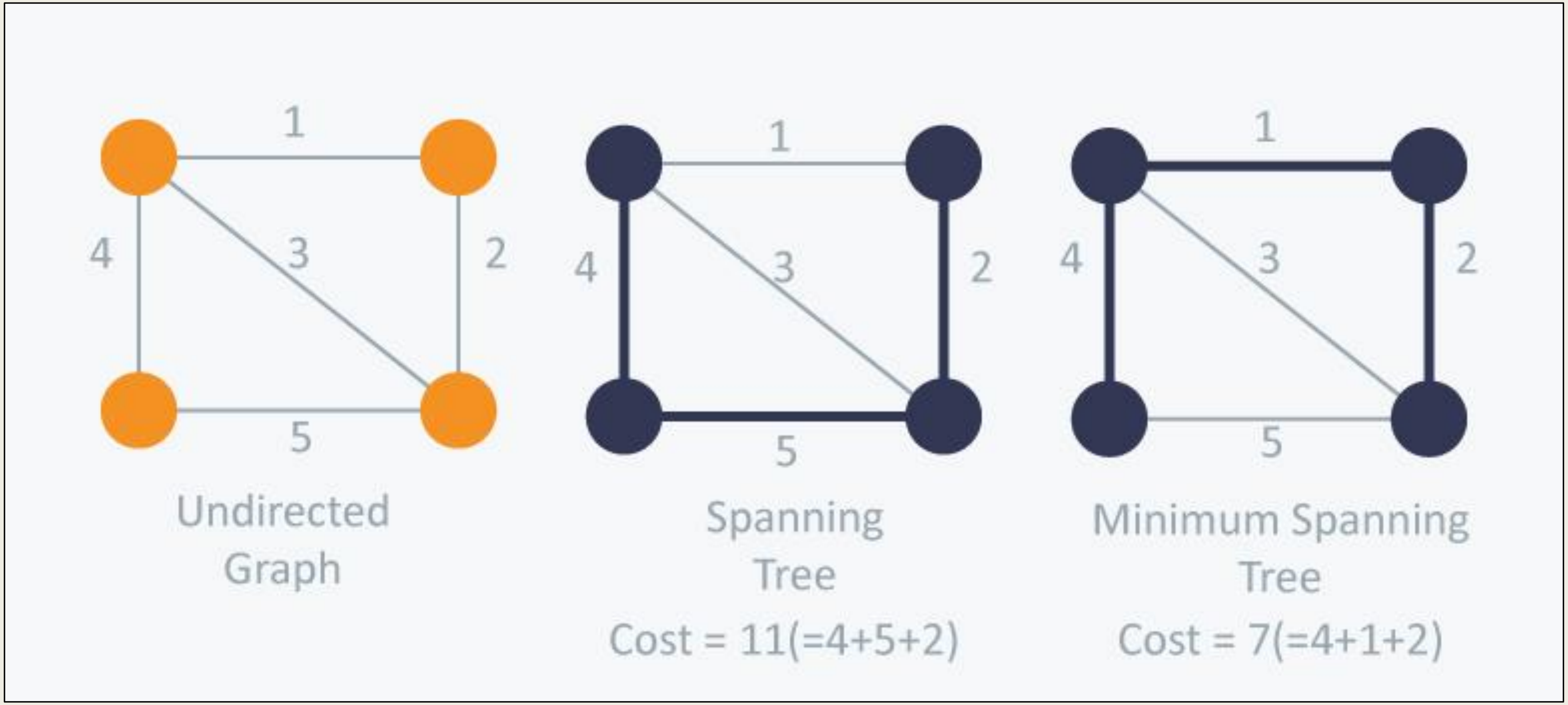


Spanning Tree

- A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.
- For a weighted graph, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges.
- The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.



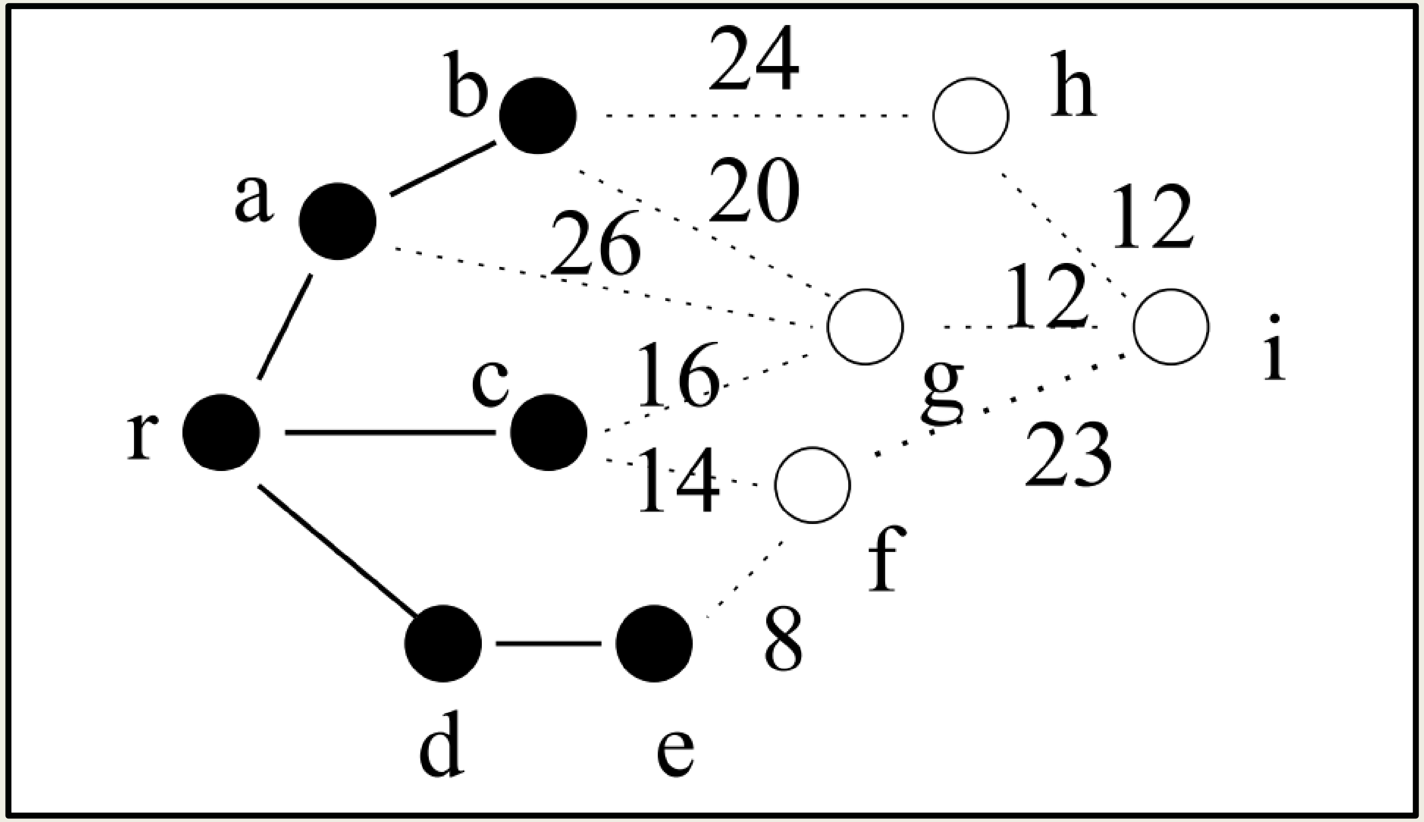
Spanning Tree



Prims Algorithm

- A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge.
- The next edge to include is chosen according to some optimization criterion.
- The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included

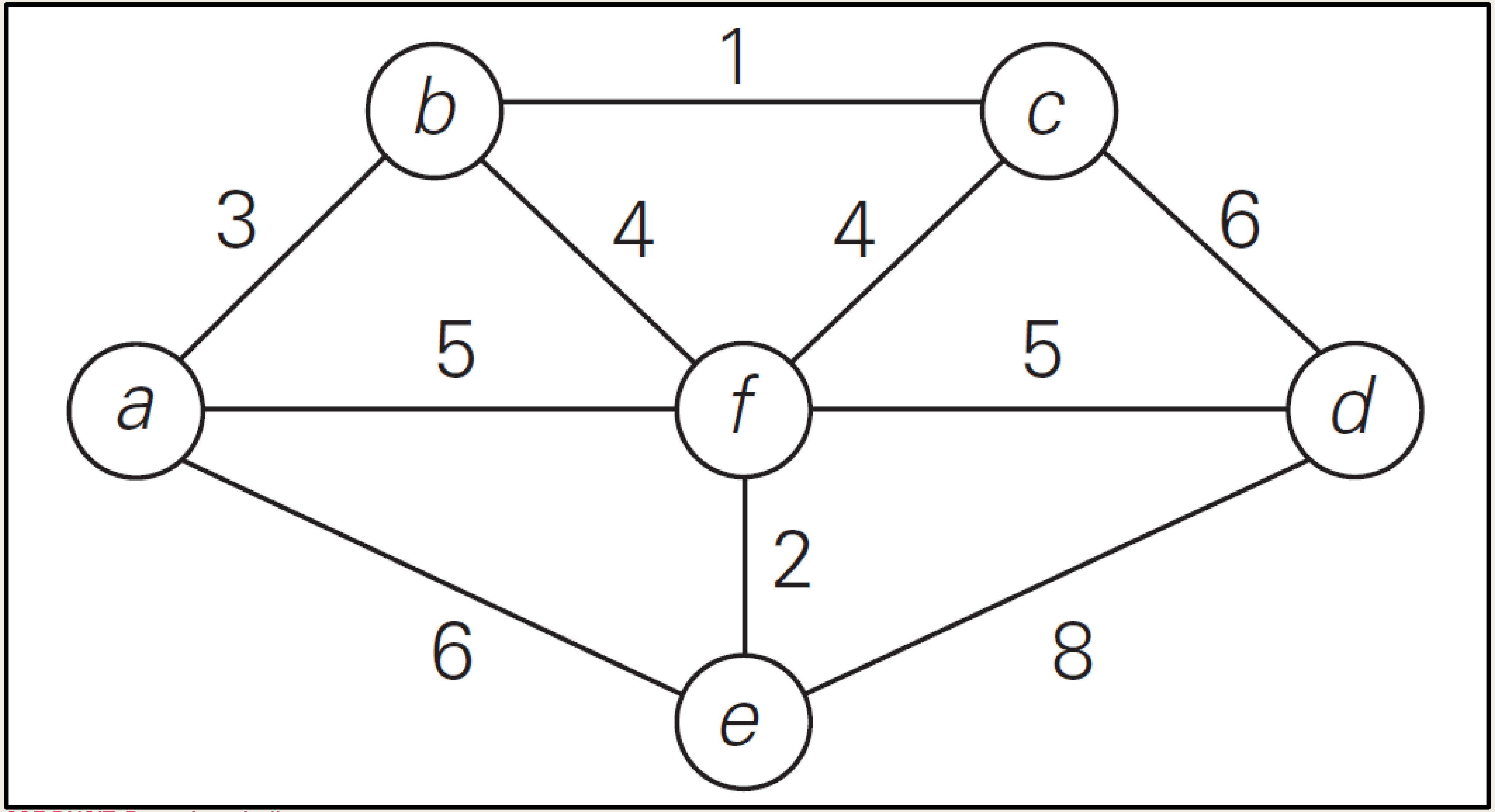
Prims Algorithm – Illustration



Tree Vertices - { r, a, b, c, d, e } – vertices that re part of the spanning tree
Fringe vertices - { h, g, f } – vertices not in the tree but adjacent to at least one vertex that is in the tree
Unseen vertices – { i } – vertices not yet affected by the algorithm

Prims Algorithm – Problem

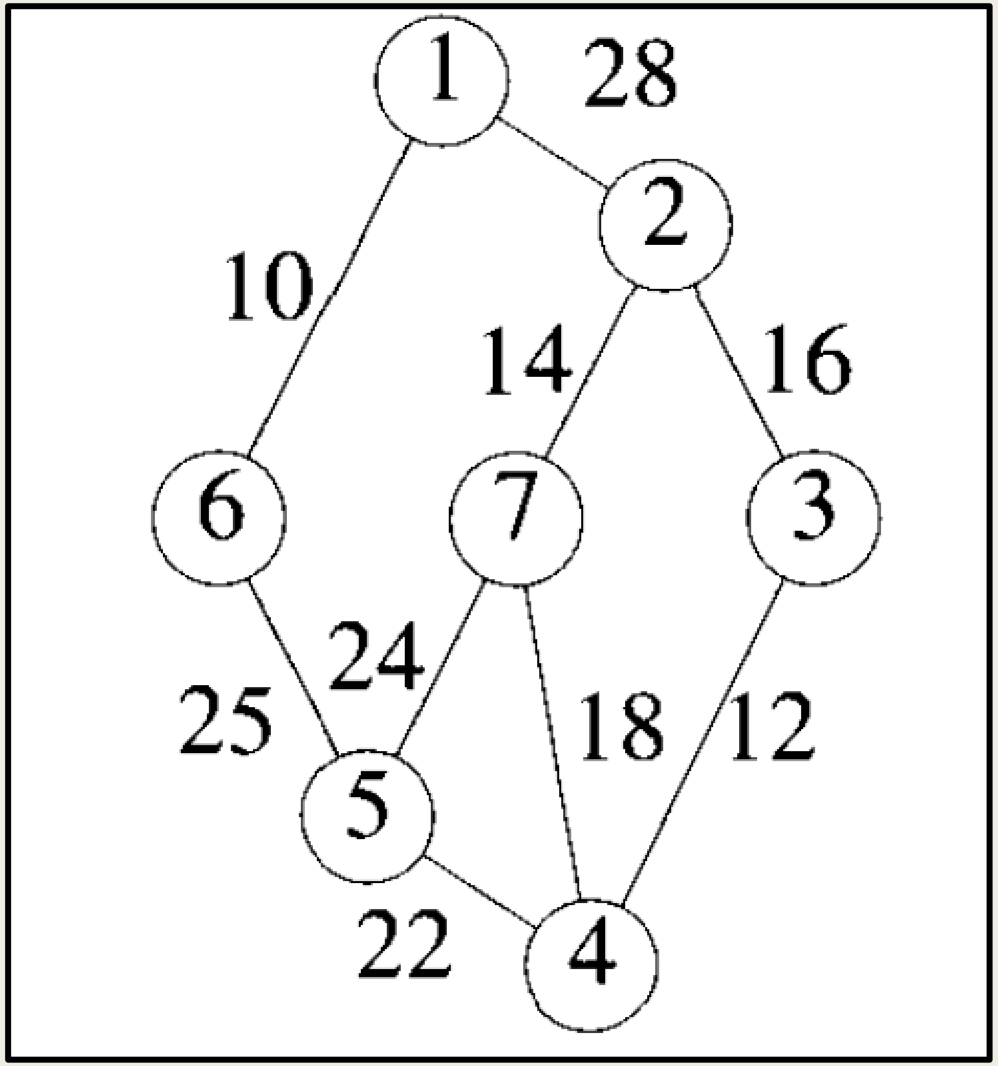
Find the minimum cost spanning tree for the following graph



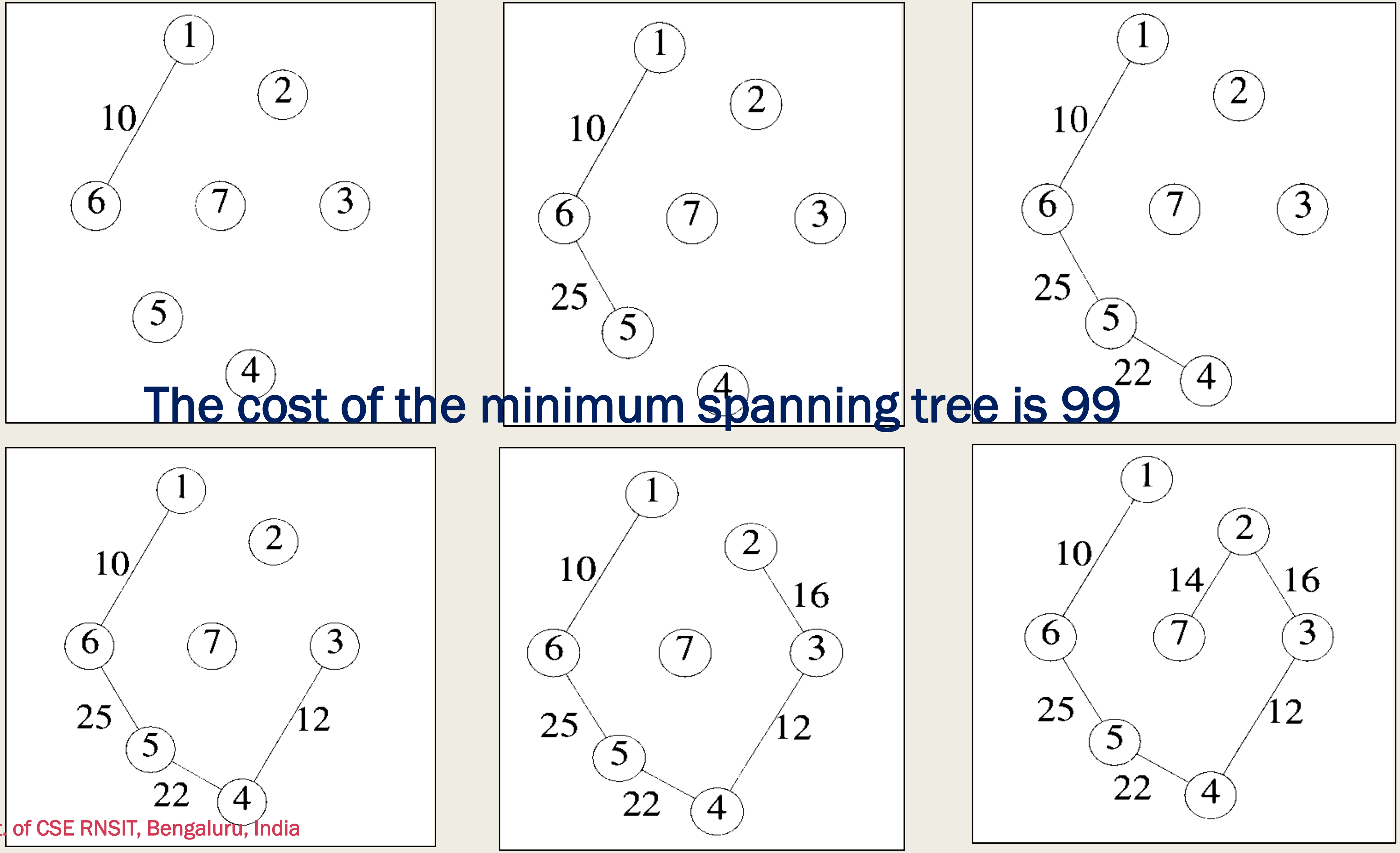
Solution

Prims Algorithm – Problem

Find the minimum cost spanning tree for the following graph



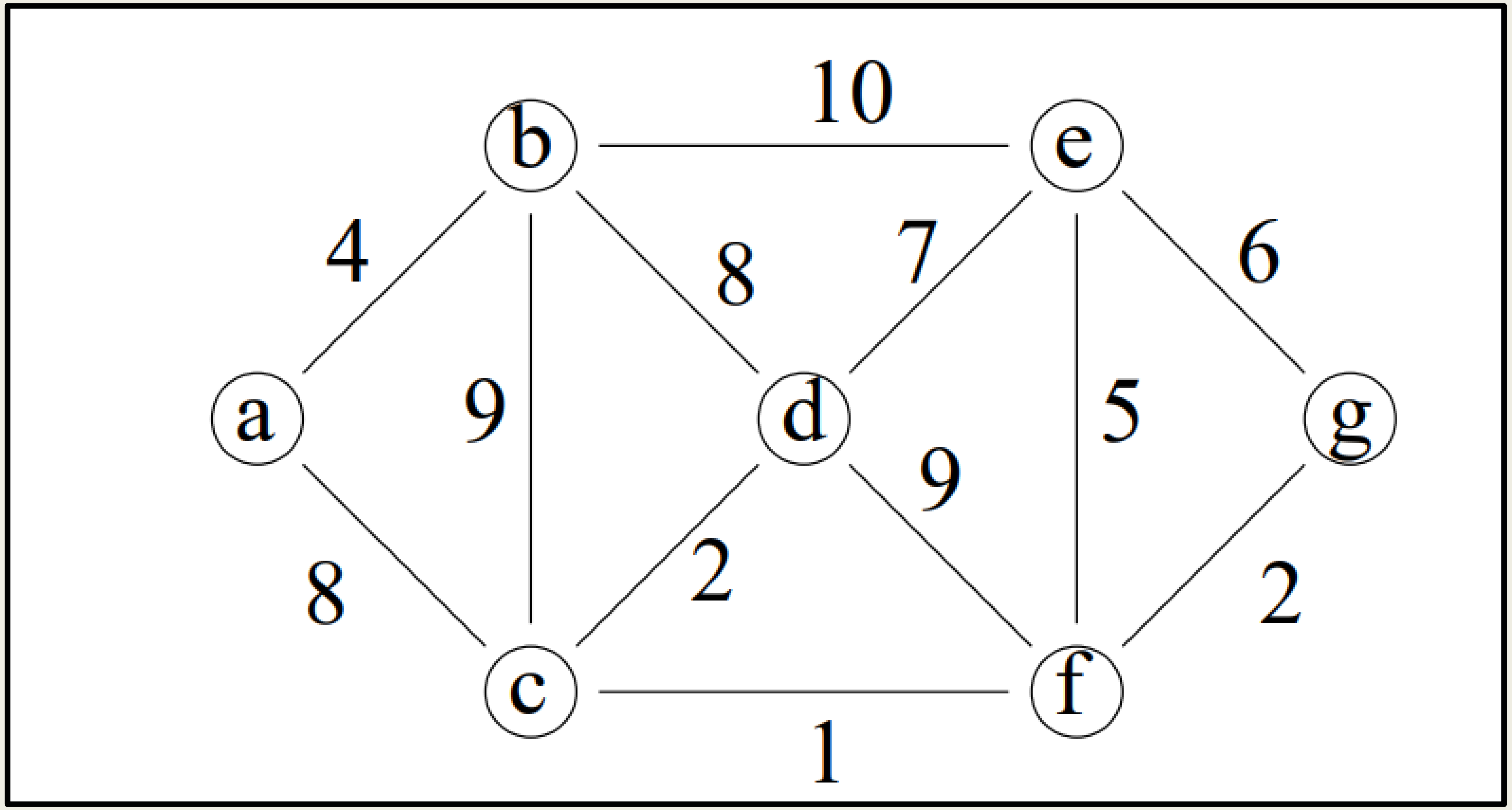
Prims Algorithm- Example



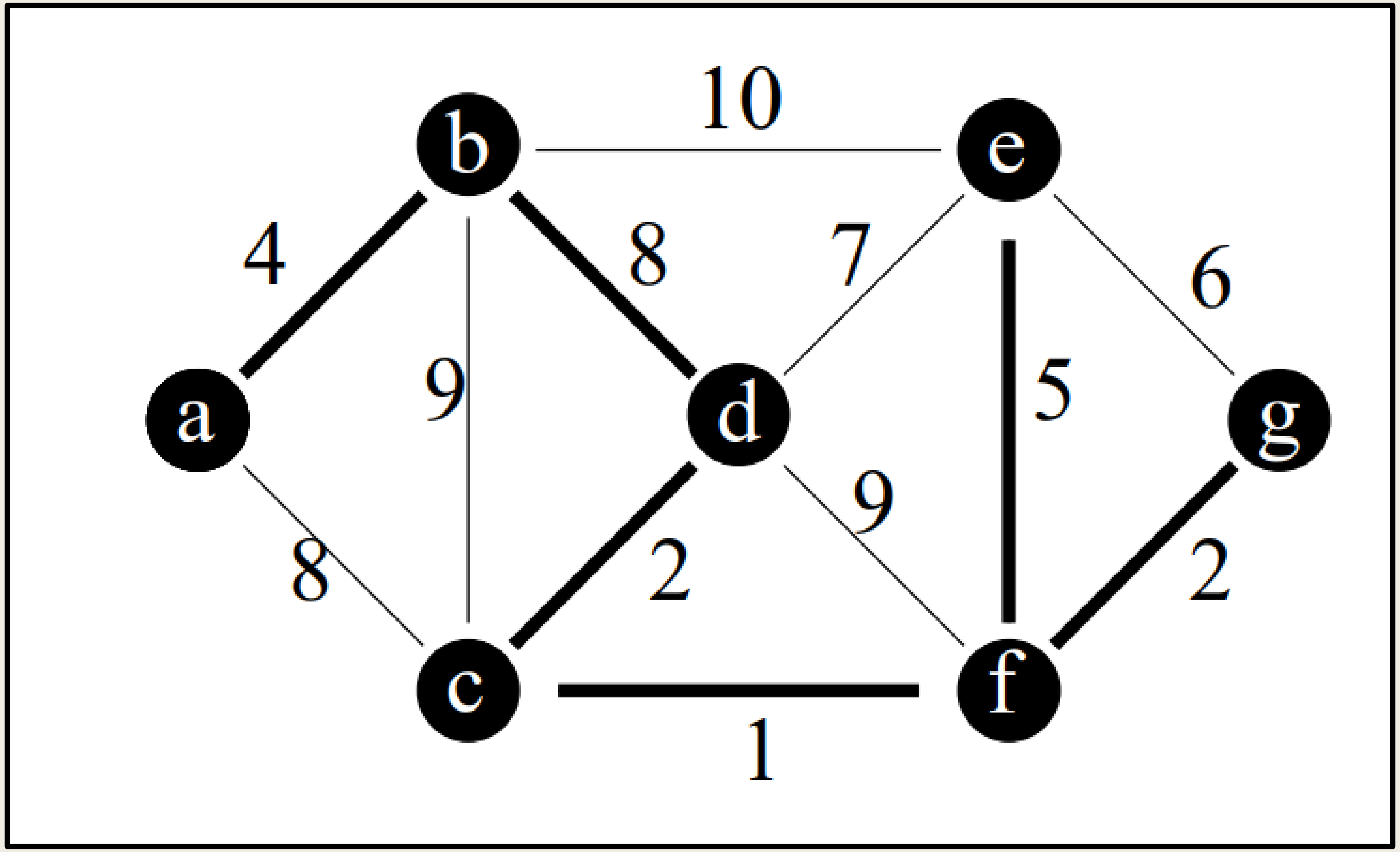
The cost of the minimum spanning tree is 99

Prims Algorithm – Problem Home work

Find the minimum cost spanning tree for the following graph



Prims Algorithm – Problem Home work – solution



The cost of the minimum spanning tree is 22

Prims Algorithm

ALGORITHM $Prim(G)$

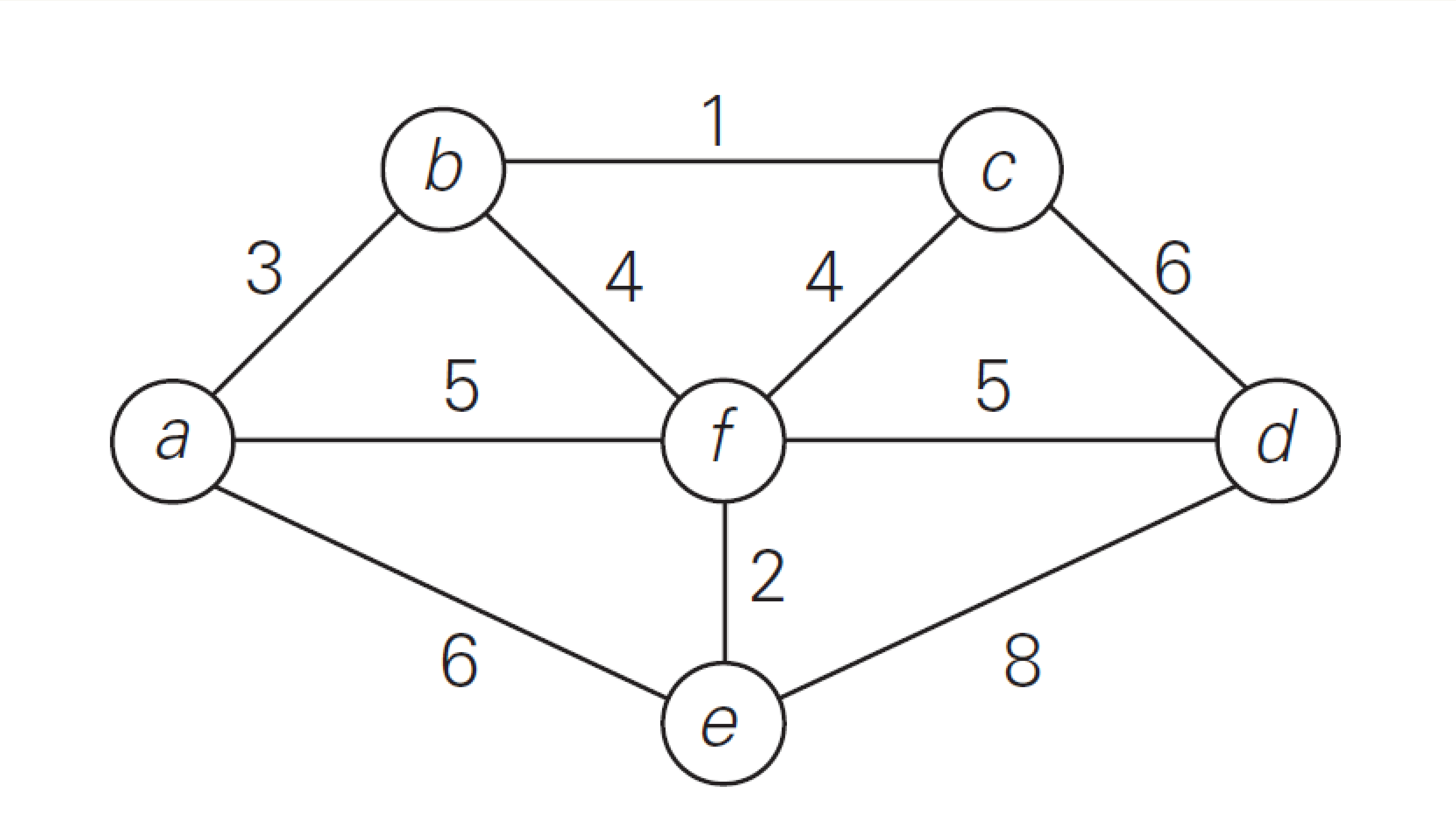
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Kruskal's Algorithm

- Joseph Kruskal developed an algorithm - Kruskal's Algorithm - to find an optimal solution to minimum spanning tree problem when he was a second year graduate student
  - Prim's algorithm grew the minimum spanning tree by including nearest vertex to the vertices already in the tree
  - In Prim's algorithm, the subgraphs were always connected in the intermediate stages
- The Kruskal's algorithm begins by sorting the edges in increasing order of their weights
- Starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise

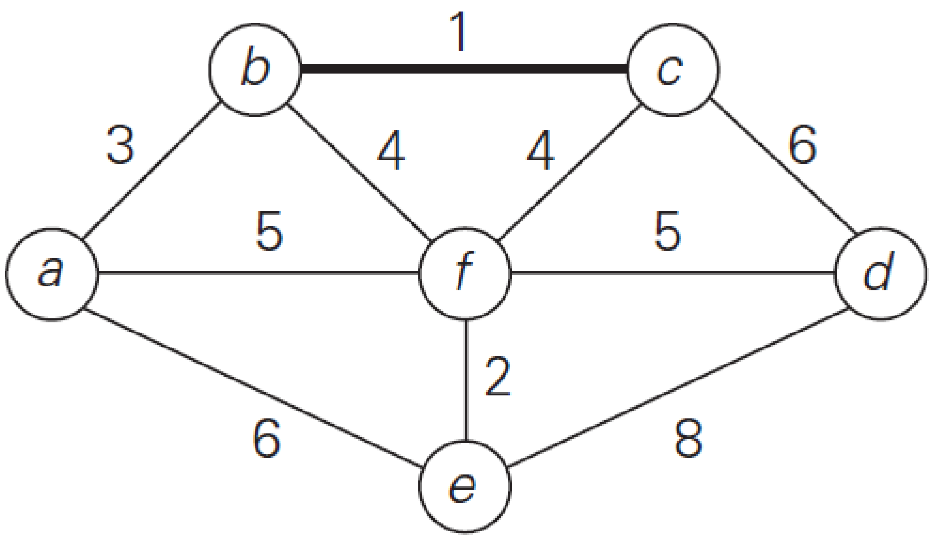
Kruskal's Algorithm - Problem

Find the minimum cost spanning tree for the following graph



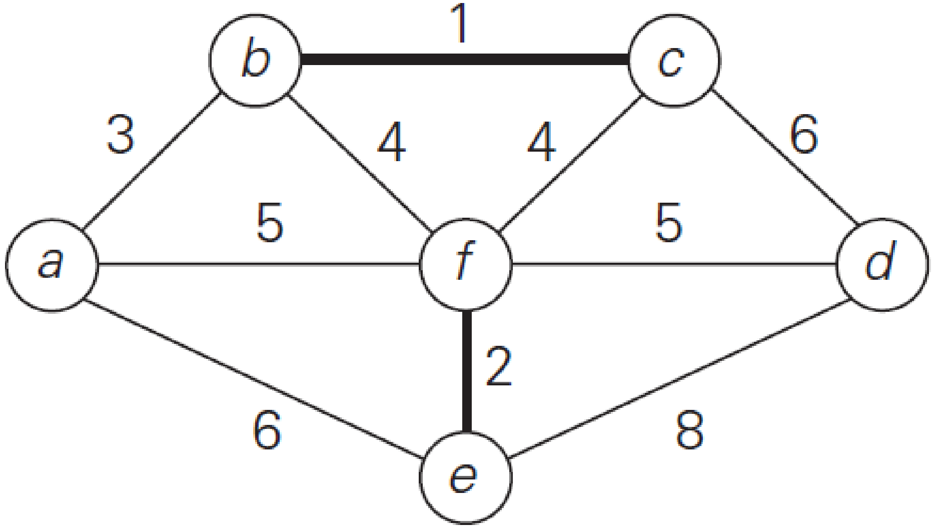
Kruskal’s Algorithm - Solution

bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



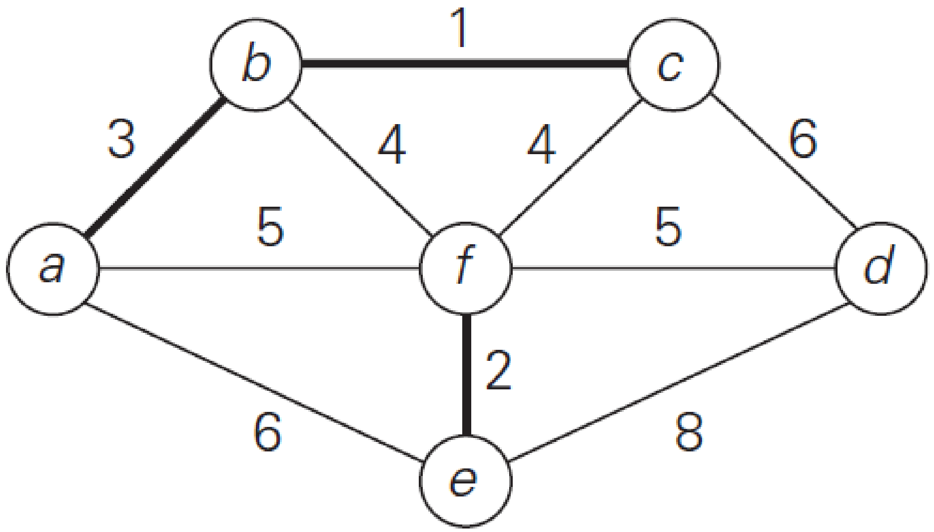
Kruskal’s Algorithm - Solution

bc	**ef**	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



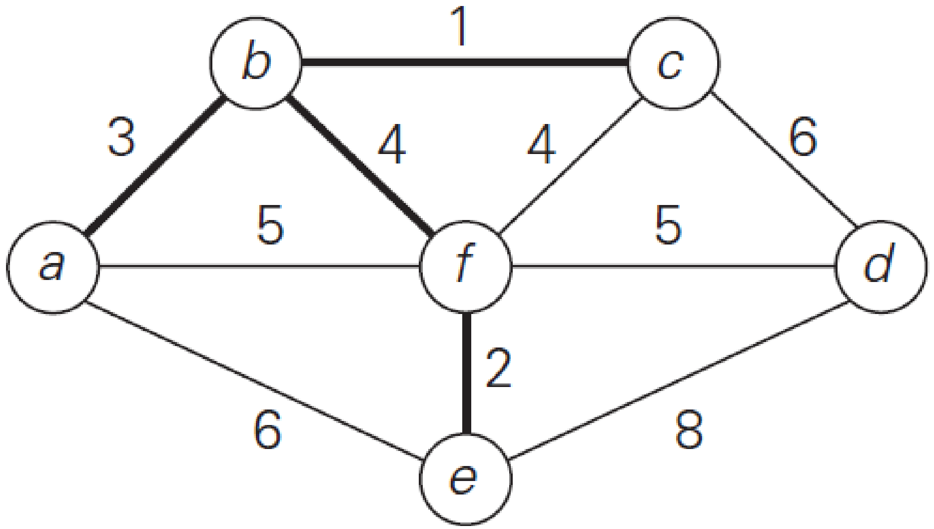
Kruskal’s Algorithm - Solution

bc	ef	**ab**	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



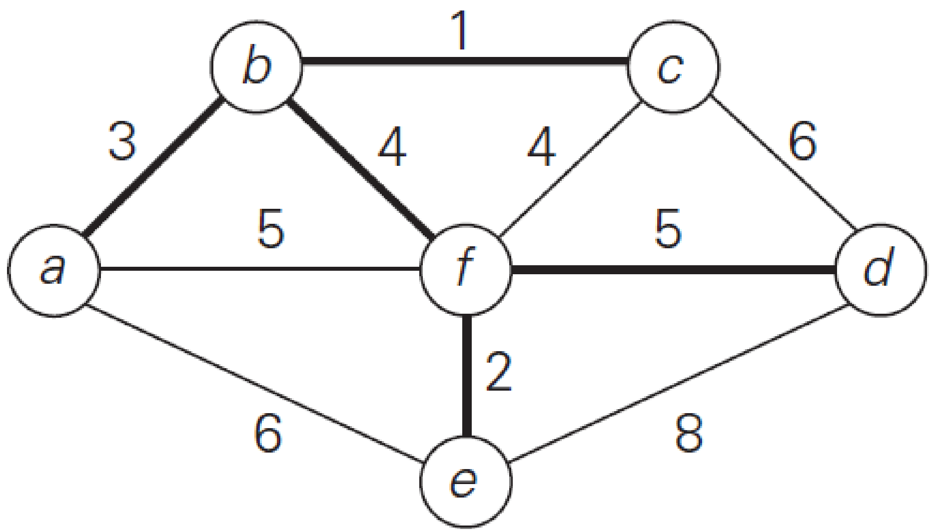
Kruskal's Algorithm - Solution

bc	ef	ab	**bf**	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



Kruskal's Algorithm - Solution

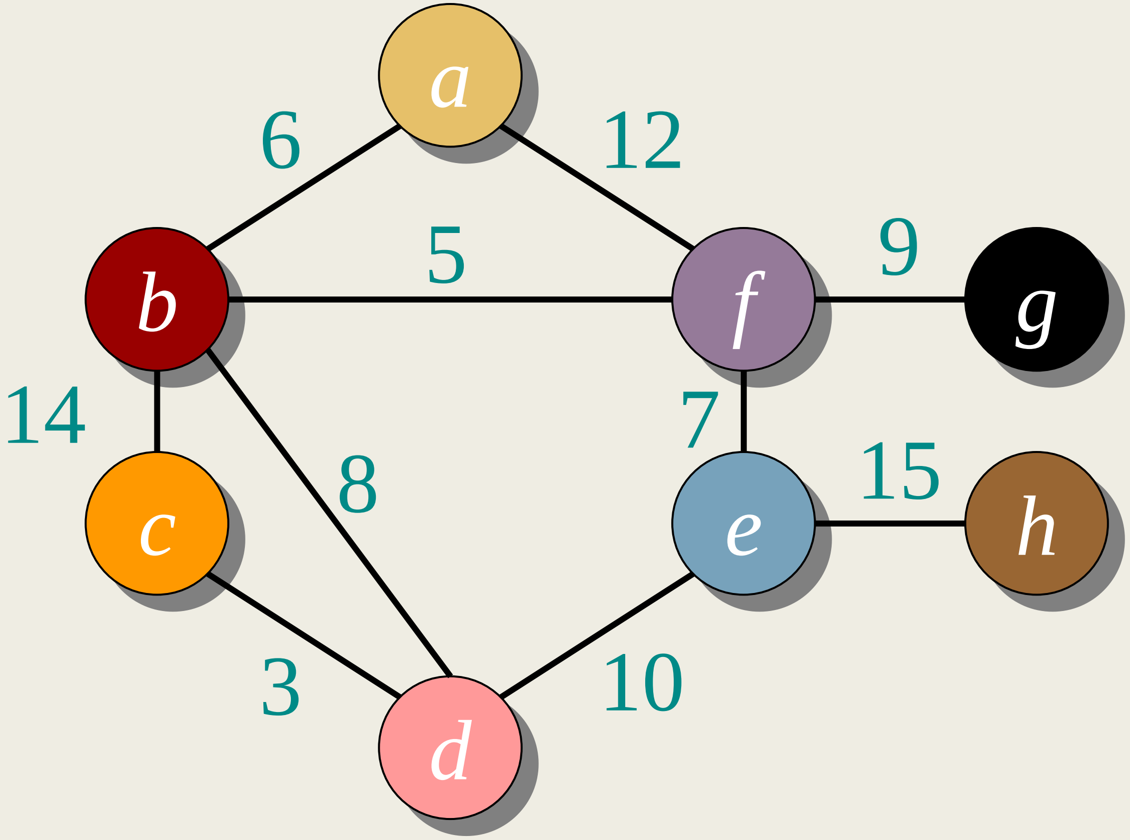
bc	ef	ab	bf	cf	af	**df**	ae	cd	de
1	2	3	4	4	5	5	6	6	8



Total cost of the minimum spanning tree= 15

Kruskal's Algorithm - Problem

- **Find the minimum cost spanning tree for the following graph**



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

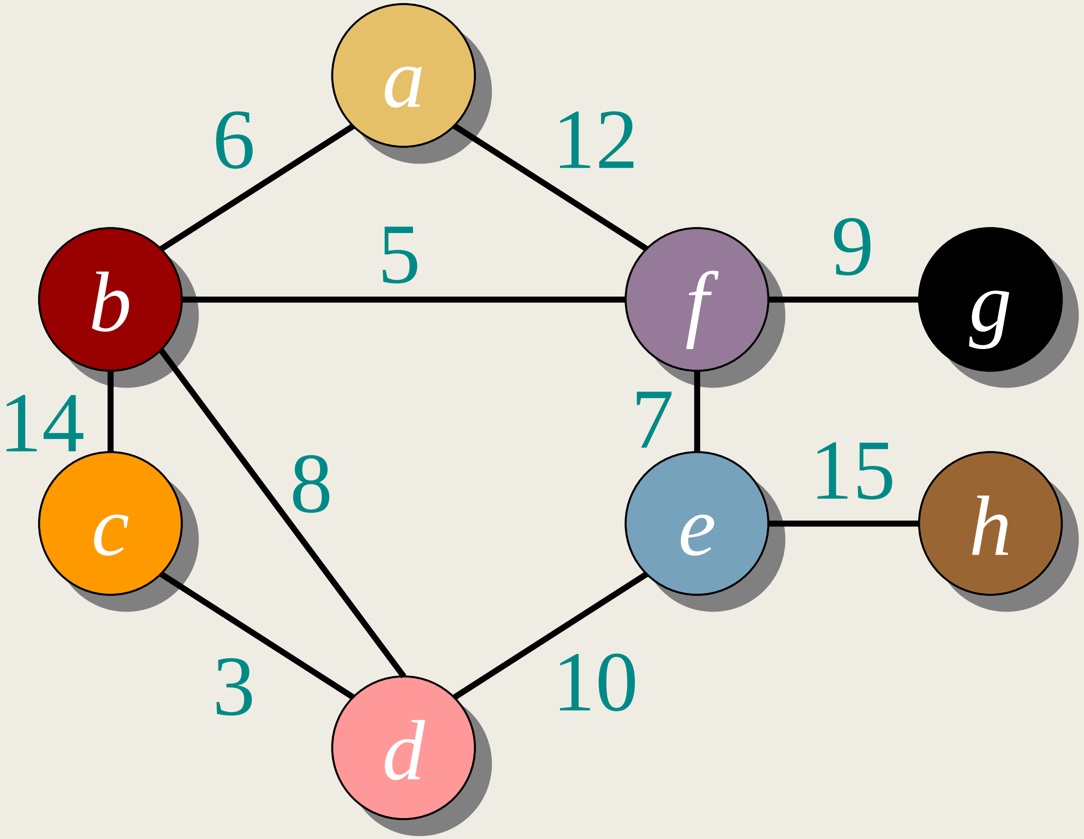
f-g: 9

d-e: 10

a-f: 12

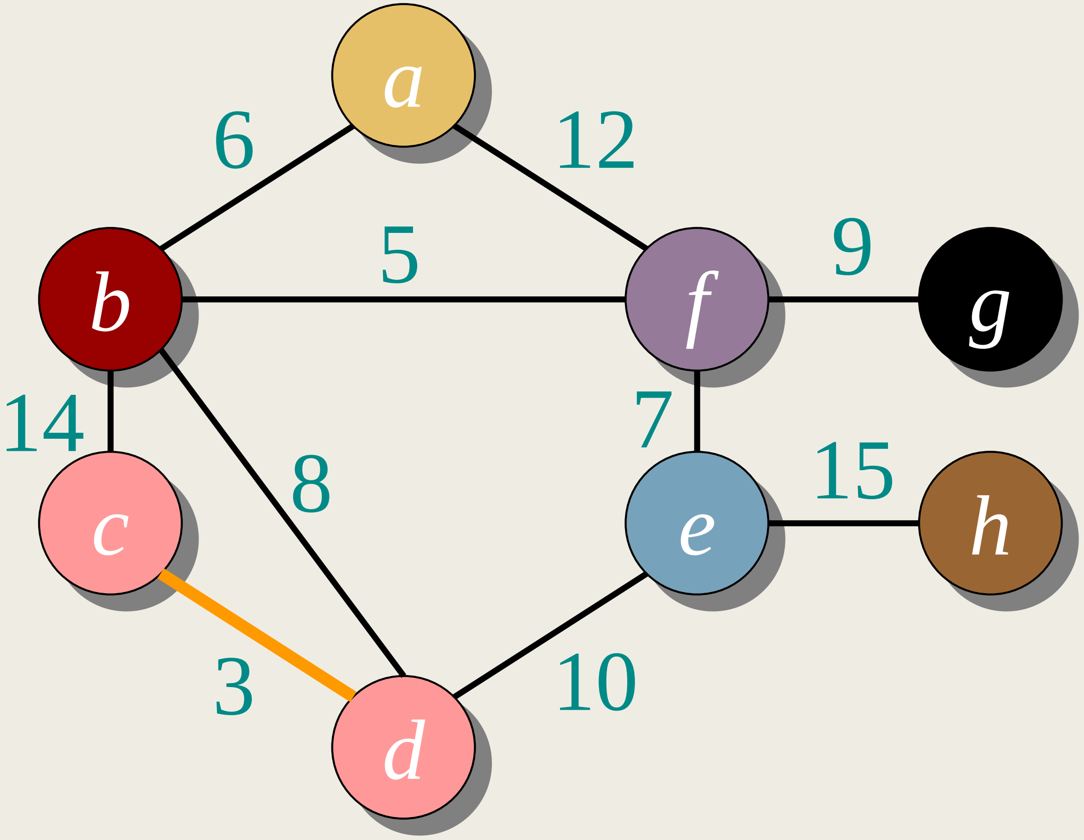
b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3
b-f: 5
b-a: 6
f-e: 7
b-d: 8
f-g: 9
d-e: 10
a-f: 12
b-c: 14
e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

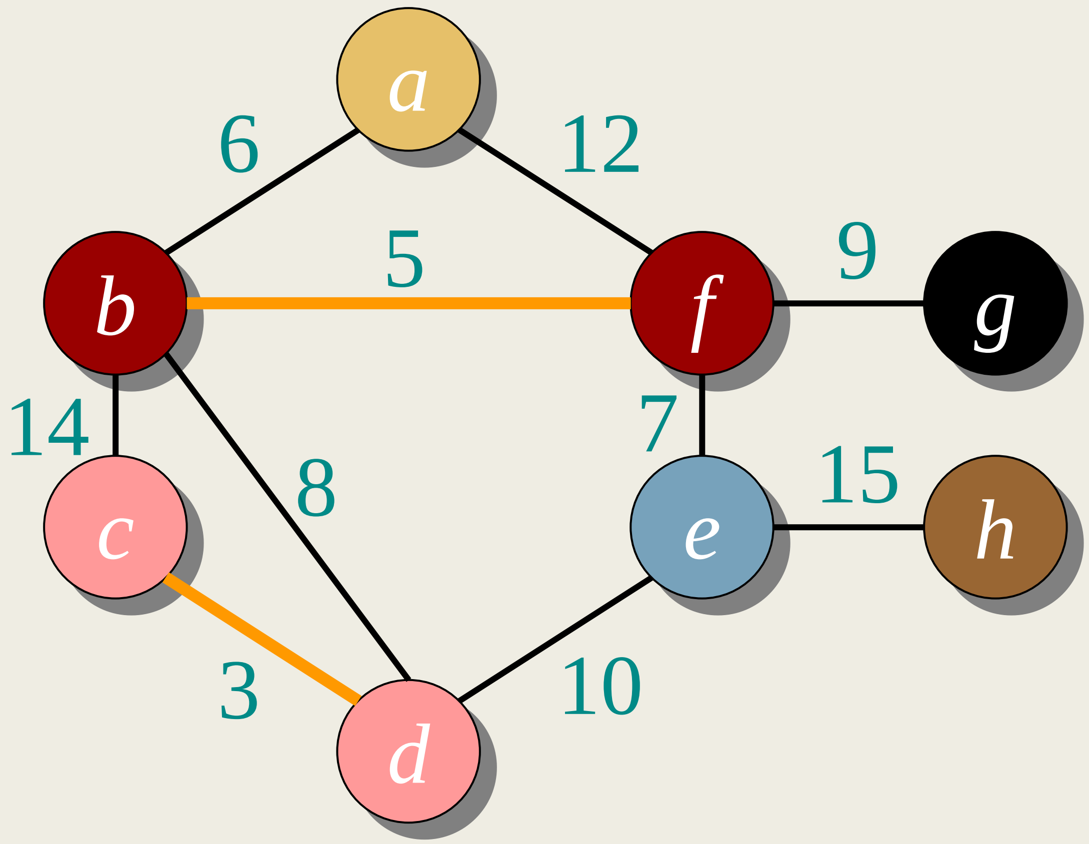
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

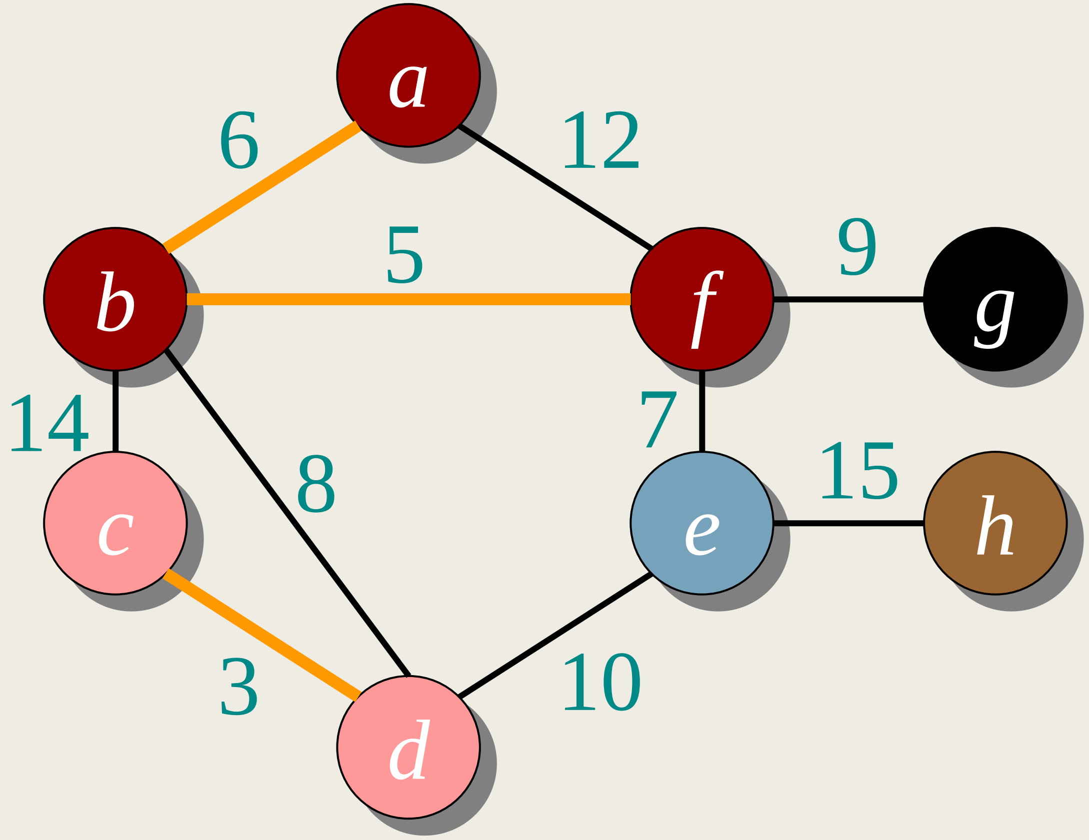
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal’s Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

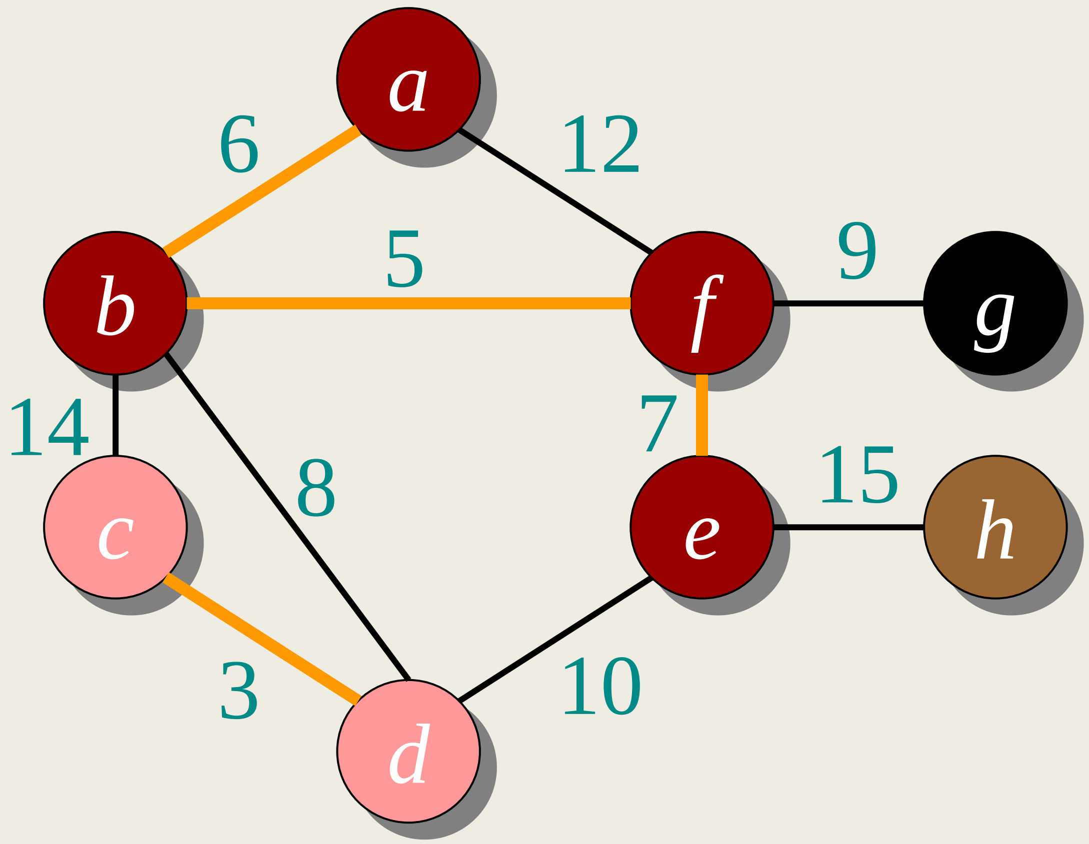
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

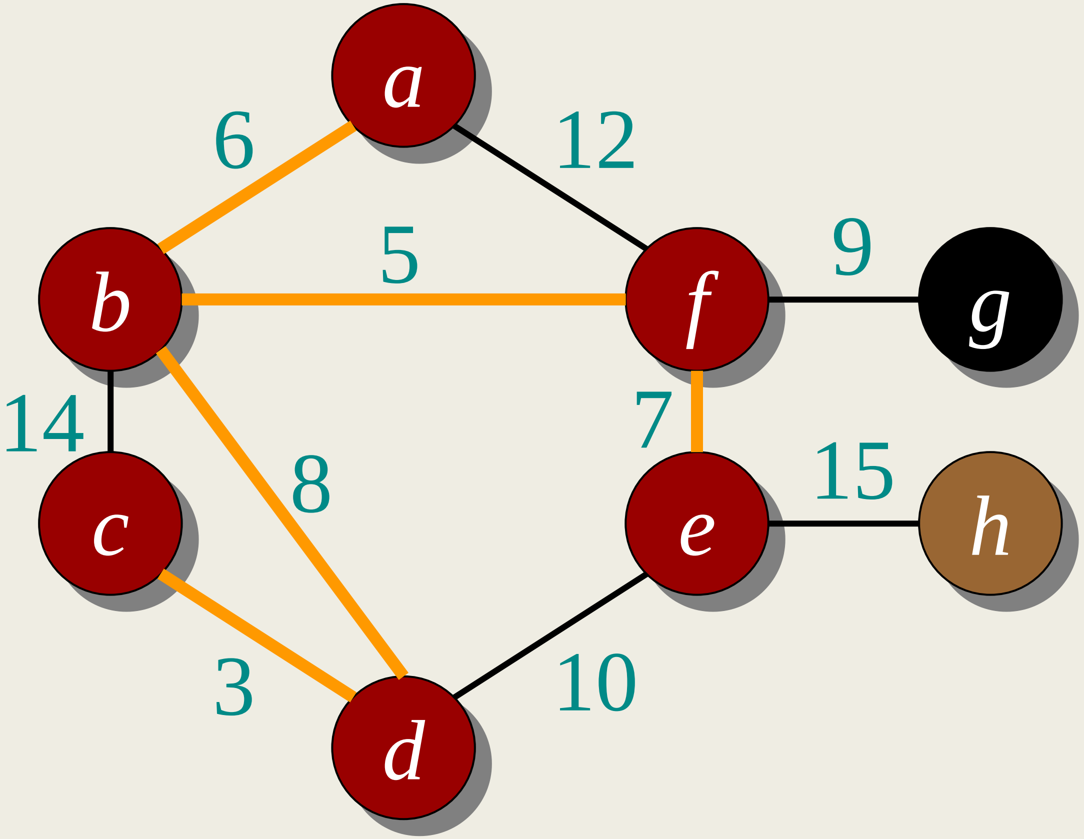
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

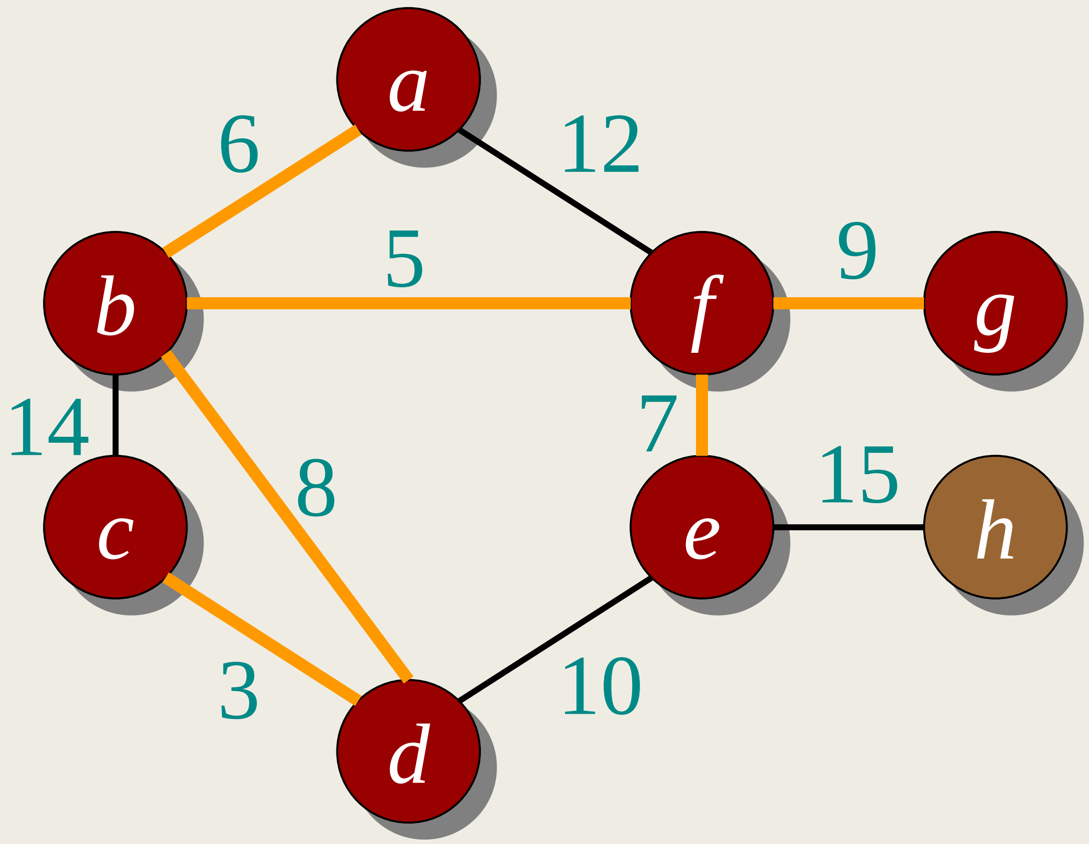
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

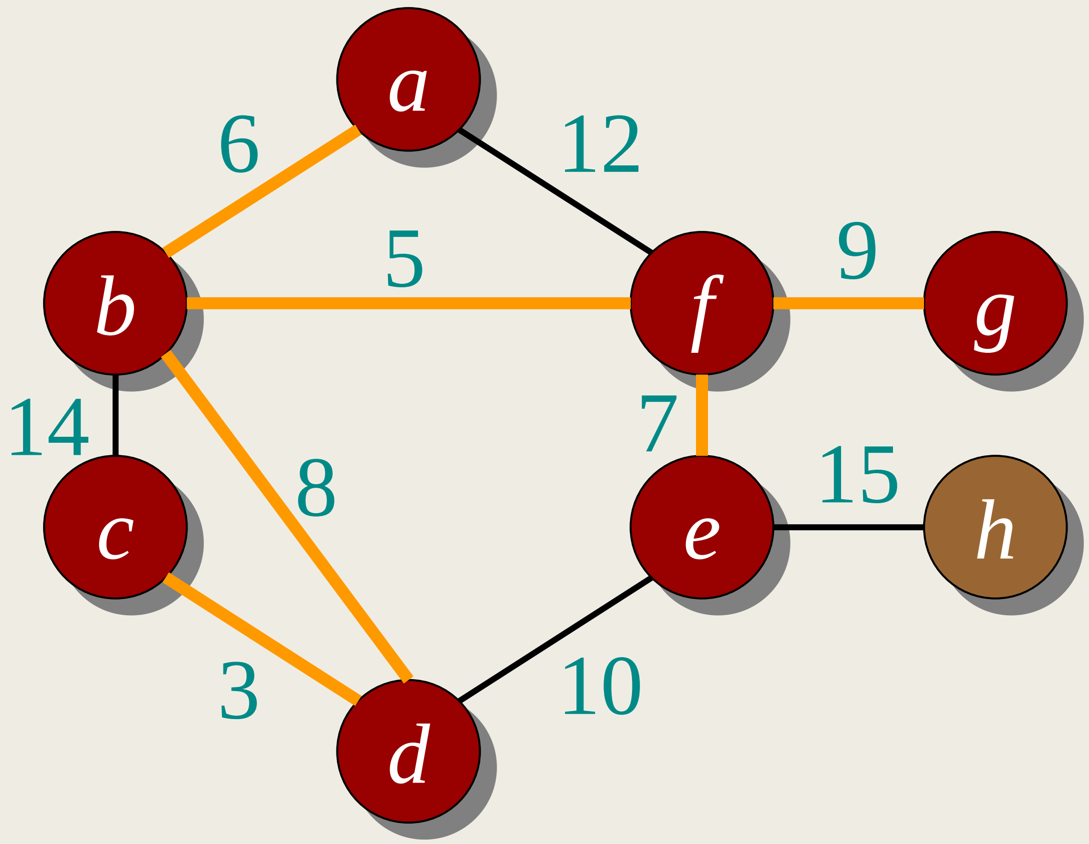
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

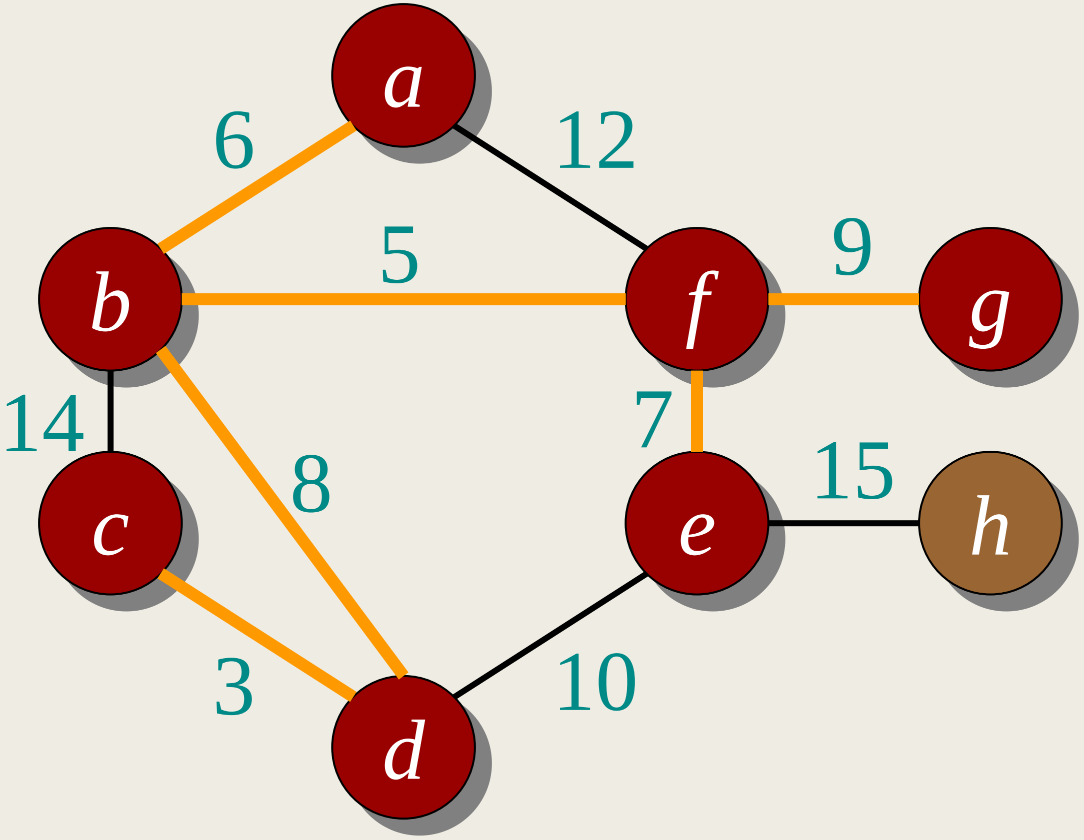
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

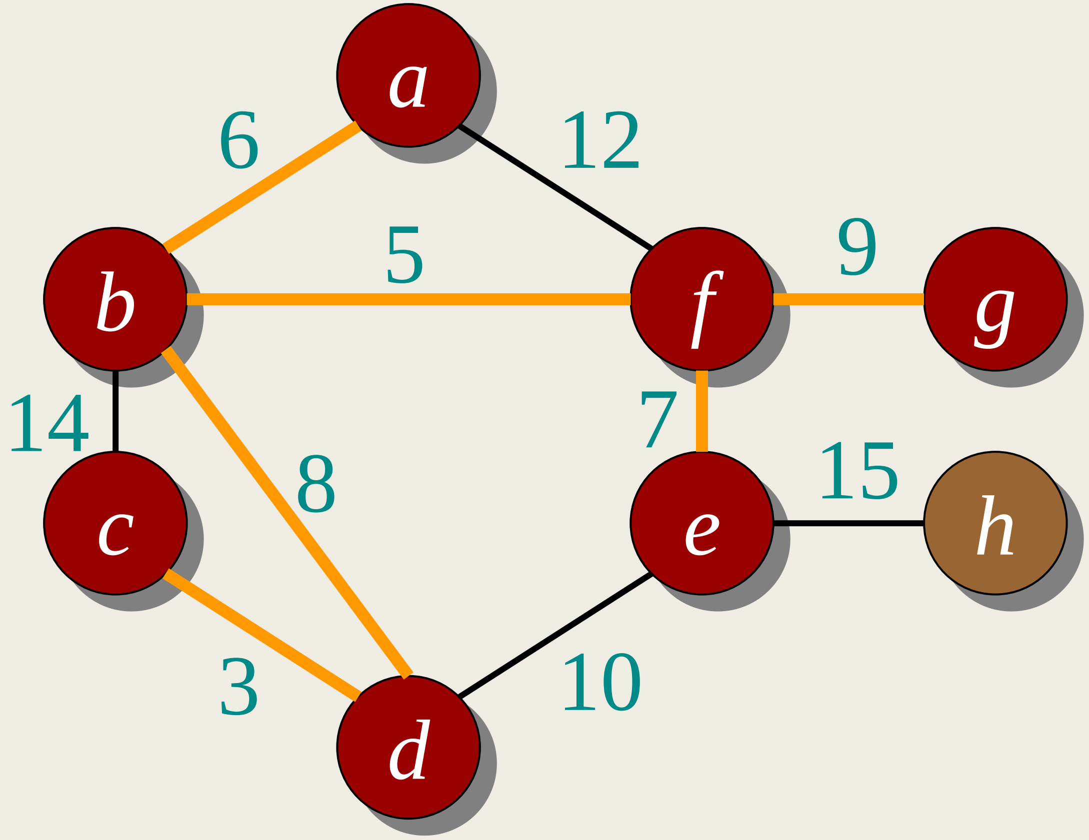
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Kruskal's Algorithm - Solution

c-d: 3

b-f: 5

b-a: 6

f-e: 7

b-d: 8

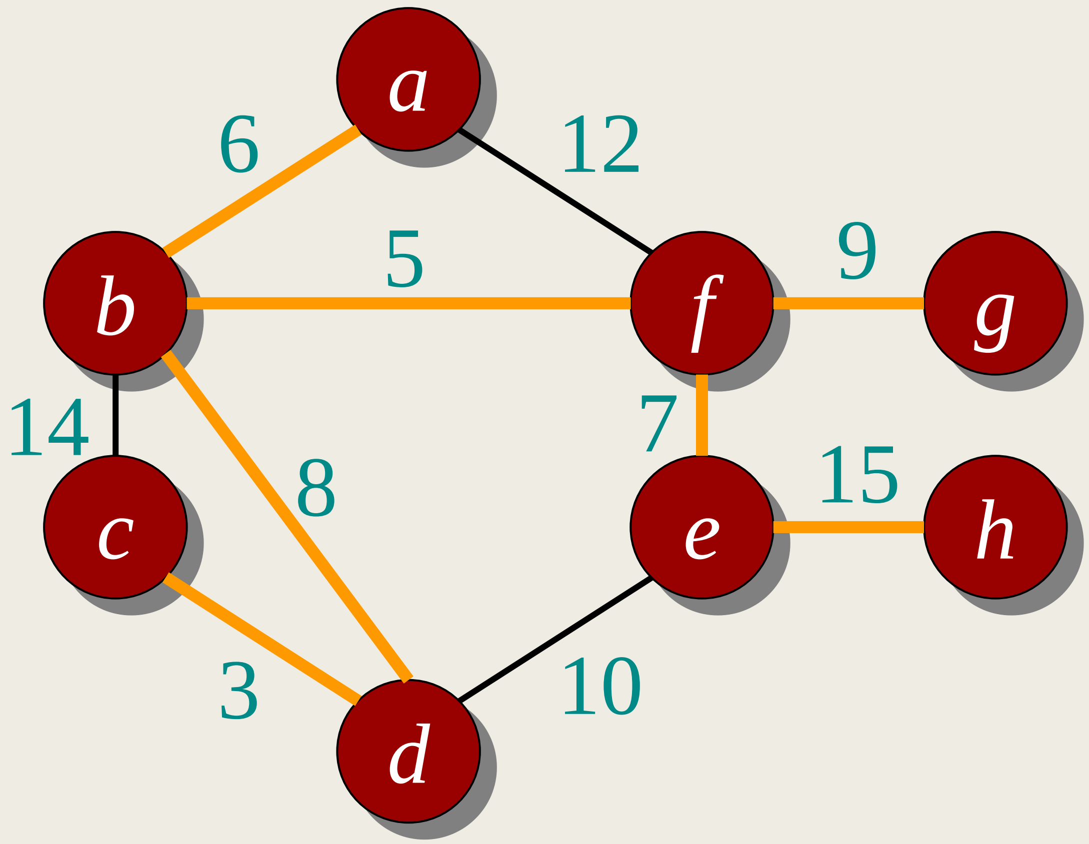
f-g: 9

d-e: 10

a-f: 12

b-c: 14

e-h: 15



Total cost of the minimum spanning tree= 53

Kruskal's Algorithm

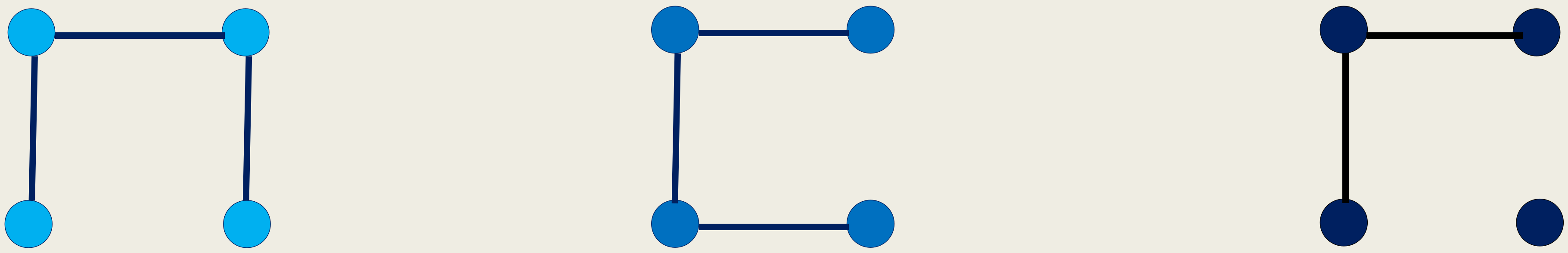
ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $\quad ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
  $k \leftarrow k + 1$
  if $E_T \cup \{e_{i_k}\}$ is acyclic
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $\quad ecounter \leftarrow ecounter + 1$
return E_T

Kruskal's Algorithm

The implementation catch

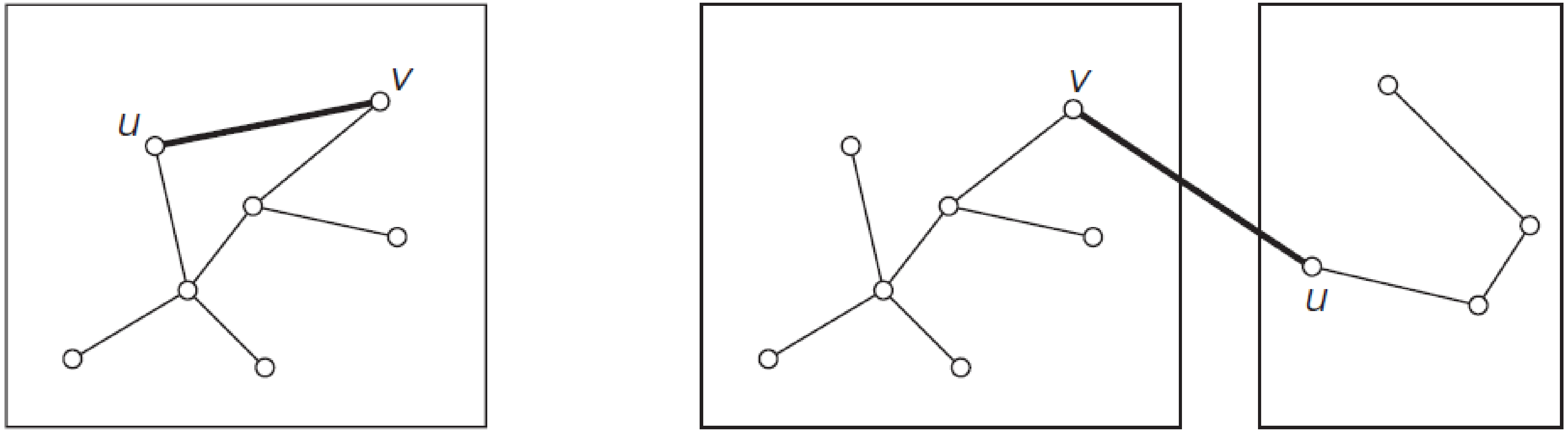
- Kruskal's approach gives away an impression of being very easy to implement. It is really so?
- Each time / iteration the algorithm should check whether adding the next edge to the edges already selected creates a cycle?
- Remember a cycle is created iff the new edge connects two vertices which are already connected by a path



Kruskal's Algorithm

The implementation catch

- New cycle is created if and only if the two vertices belong to the same connected component
- Each connected component of a subgraph generated by Kruskal's algorithm is a tree because it has no cycles.



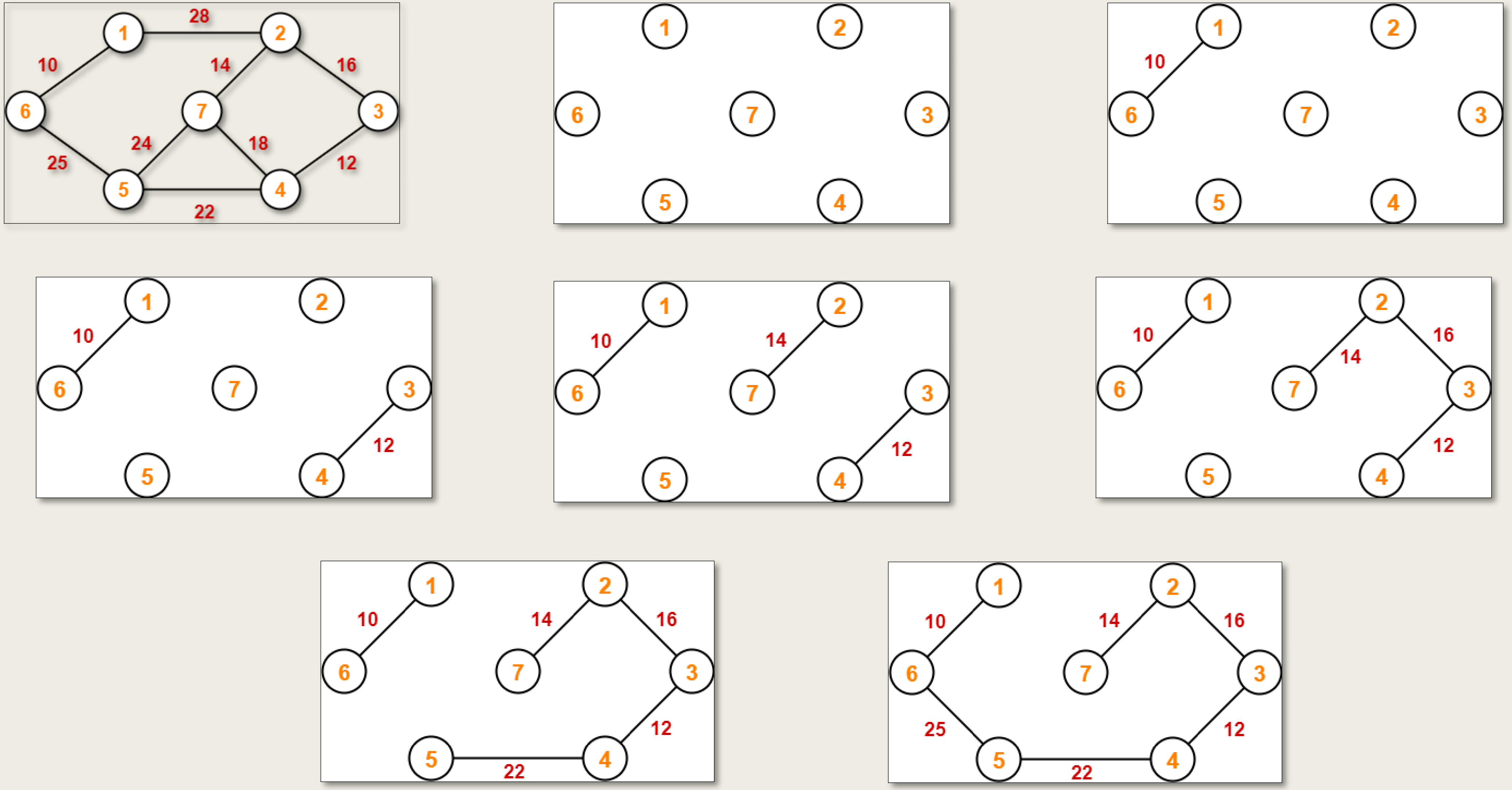
(a) (b)

Kruskal's Algorithm

The implementation catch

- Consider the algorithm's operations as a progression through a series of forests containing all the vertices of a given graph and some of its edges.
- The initial forest consists of $|V|$ trivial trees, each comprising a single vertex of the graph.
- The final forest consists of a single tree, which is a minimum spanning tree of the graph.
- On each iteration, the algorithm takes
  - *the next edge (u, v) from the sorted list of the graph's edges,*
  - *finds the trees containing the vertices u and v, and,*
  - *if these trees are not the same, unites them in a larger tree by adding the edge (u, v).*

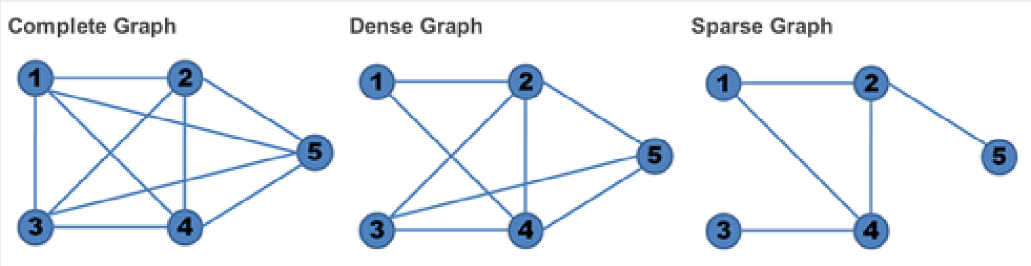
Kruskal’s Algorithm



Differences between Prim's and Kruskal's

PRIM'S ALGORITHM	KRUSKAL'S ALGORITHM
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V^2), V being the number of vertices.	Kruskal's algorithm's time complexity is O(logV), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

A *graph* in which the number of *edges* is much less than the possible number of edges



A *graph* in which the number of *edges* is close to the possible number of edges.

Job Sequencing with deadlines

- The sequencing of jobs (for execution) on a single processor with deadline constraints is called as Job Sequencing with Deadlines.
- Problem statement

"Given,

- *set of n jobs*
- *a deadline associated with each job i, $d_i \geq 0$*
- *profit associated with each job i , $p_i > 0$*
  - For any job i the profit p_i is earned iff the job is completed by its deadline
  - To complete a job, one has to process the job on a machine for one unit of time
  - Only one machine is available for processing job

Obtain the sequence of jobs that yields optimal solution (max profit)"

Job Sequencing with deadlines

What is the feasible solution ?

- **A feasible solution for this problem is a subset J of jobs such that each Job in this subset can be completed by its deadline**
- **The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$**
- **An optimal solution is a feasible solution with maximum value.**

Job Sequencing with deadlines

Brute force approach

- **Solve the following instance of job sequencing problem**

Let n = 4, (p1,p2,p3, p4)= (100,10,15, 27), (d1, d2, d3, d4)=(2, 1, 2, 1)

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Job Sequencing with deadlines

Greedy Approach

- ***Sort all the given jobs in decreasing order of their profit.***
- ***Check the value of maximum deadline.***
- ***Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.***
- ***Select the jobs one by one.***
- ***Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.***

Job Sequencing with deadlines

- **Solve the following instance of job sequencing with deadline problem.**

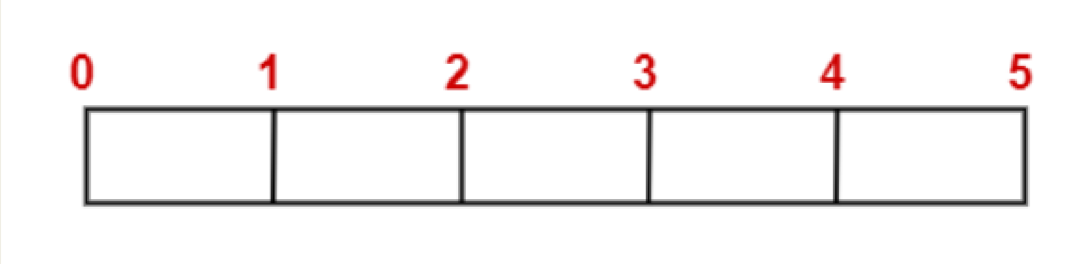
Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

- **Solution**
  - ***Sort all the given jobs in decreasing order of their profit.***

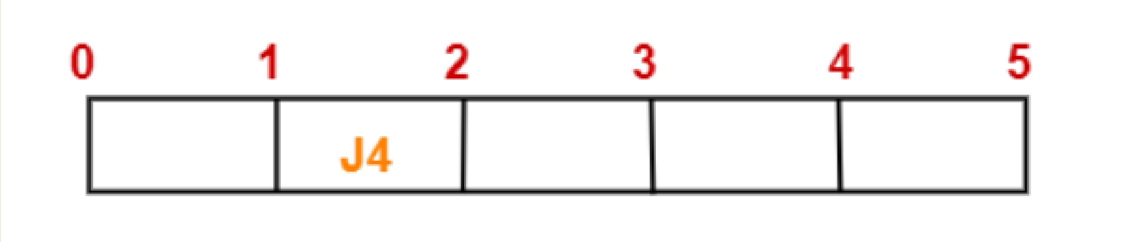
Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Job Sequencing with deadlines

- Value of maximum deadline = 5.
- So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown below

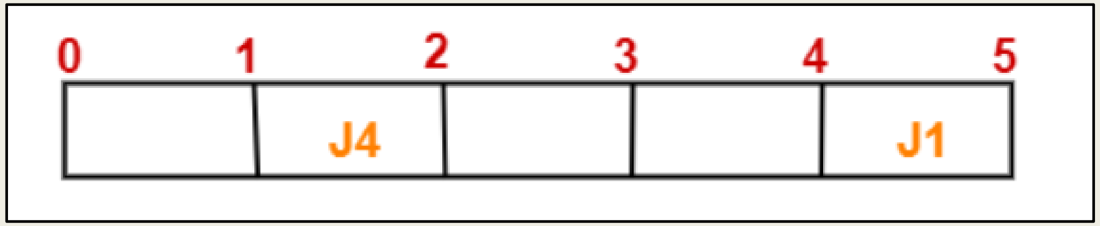


- Select the jobs as per the ordered profits and place them in the chart accordingly
- Select J4 (d4=2)

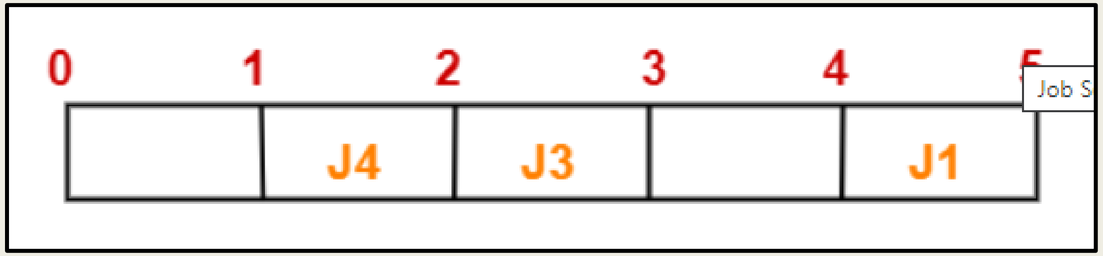


Job Sequencing with deadlines

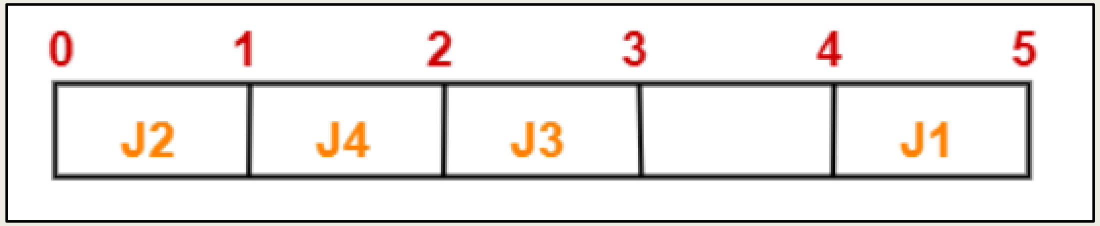
- Select J1 (d1=5)



- Select J3 (d3=3)



- Select J2 (d2=3)



Job Sequencing with deadlines

- Select J5 (d5=4)

0 1 2 3 4 5

J2	J4	J3	J5	J1

- The left over job is J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.
- Hence the optimal sequence of jobs are <J2, J4, J3, J5, J1>
- The profit earned by this sequence is 990 units

Job Sequencing with deadlines

- **Solve the following instance of job sequencing problem**

Let $n = 4$, $(p1,p2,p3, p4)= (100,10,15, 27)$, $(d1, d2, d3, d4)=(2, 1, 2, 1)$

Solution ?

Huffman coding

Situation

- How will you compactly store a data file of 100000 characters?
- The file consists of only 6 different characters
- Their occurrences (frequencies) are as follows

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

- Designing a binary character code (in which each character is represented by a unique binary string, which we call a *Codeword*) is one among multiple solutions

Huffman coding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- **Fixed length code (e.g. ASCII)**
  - **3 bits to represent 6 characters**
  - **a = 000, b = 001, . . . , f = 101.**
  - **This method requires 300,000 bits to code the entire file**
  - **Can you do better ?**

Huffman coding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- **Variable length code (Morse code)**
  - **assign frequent characters short codewords and infrequent characters long codewords**
  - **This code requires (45.1+13.3+12.3+16.3+9.4+5.4).1,000= 224,000 bits**
  - **When compared to fixed length code, the variable length code saved the storage by 25%**

Huffman coding

Prefix codes

- A code is called a prefix (free) code if no codeword is a prefix of another one. (typically of variable length)
- Example

A: 00
B: 010
C: 011
D: 10
E: 11

Prefix code

A: 00
B: 010
C: 001 # 011 -> 001
D: 10
E: 11

Not a Prefix code

{3, 11, 22} Prefix code

{1, 12, 13} Not a Prefix code

Huffman coding

Prefix codes

- **Life is easy with prefix code**
- **Let [1, 2, 33, 34, 50, 61] be the codewords**
- **Let the sequence of number received be 1611333425012**
- **Decoding ?**

1 61 1 33 34 2 50 1 2

- https://gist.github.com/joepie91/26579e2f73ad903144dd5d75e2f03d83
- https://leimao.github.io/blog/Huffman-Coding/

Huffman coding

David Huffman invented a greedy algorithm to

"Construct a tree that would assign shorter bit strings to high-frequency symbols and longer ones to low-frequency symbols"

- **He invented this when he was a graduate student at MIT**
- **The two major steps in Huffman algorithm are**
  - **Building a Huffman Tree from the input characters.**
  - **Assigning code to the characters by traversing the Huffman Tree.**

Huffman coding

1. Create a leaf node for each character of the text.
2. Arrange all the nodes in increasing order of their frequency value.
3. Considering the first two nodes having minimum frequency,
   a. Create a new internal node.
   b. The frequency of this new node is the sum of frequency of those two nodes.
   c. Make the first node as a left child and the other node as a right child of the newly created node.
4. Keep repeating Step-2 and Step-3 until all the nodes form a single tree.

https://www.gatevidyalay.com/huffman-coding-huffman-encoding/

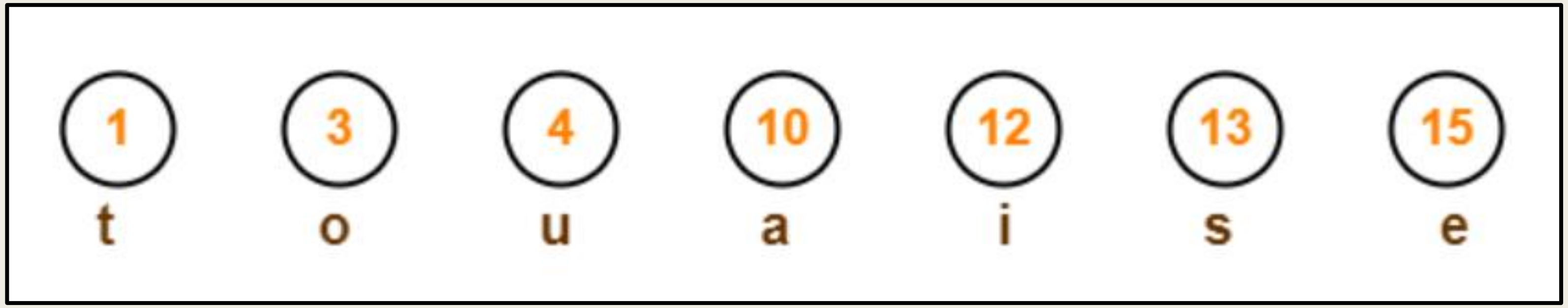
Huffman coding - Problem

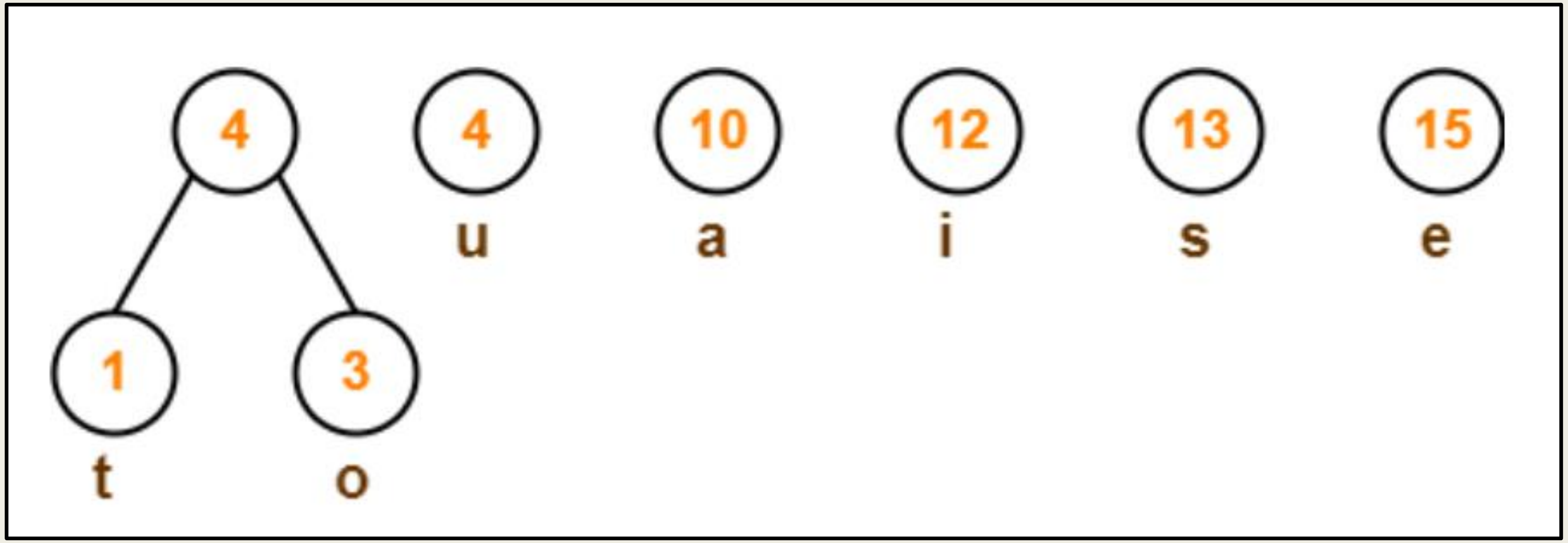
- A file contains the following characters with the frequencies as shown. If Huffman Coding is used for data compression, determine-
  - Huffman Code for each character
  - Average code length
  - Length of Huffman encoded message (in bits)

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

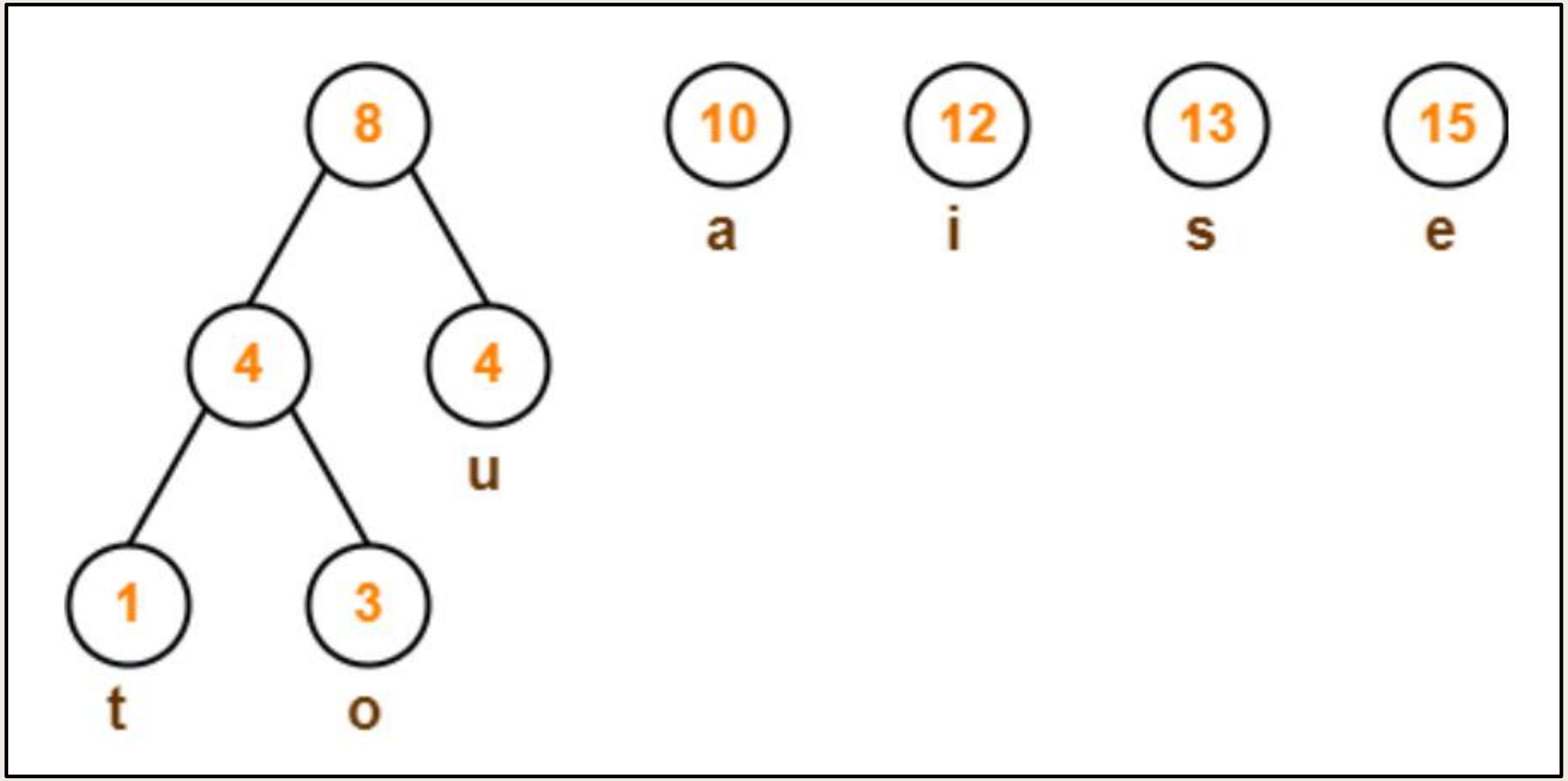
Huffman coding

Construction of Huffman tree

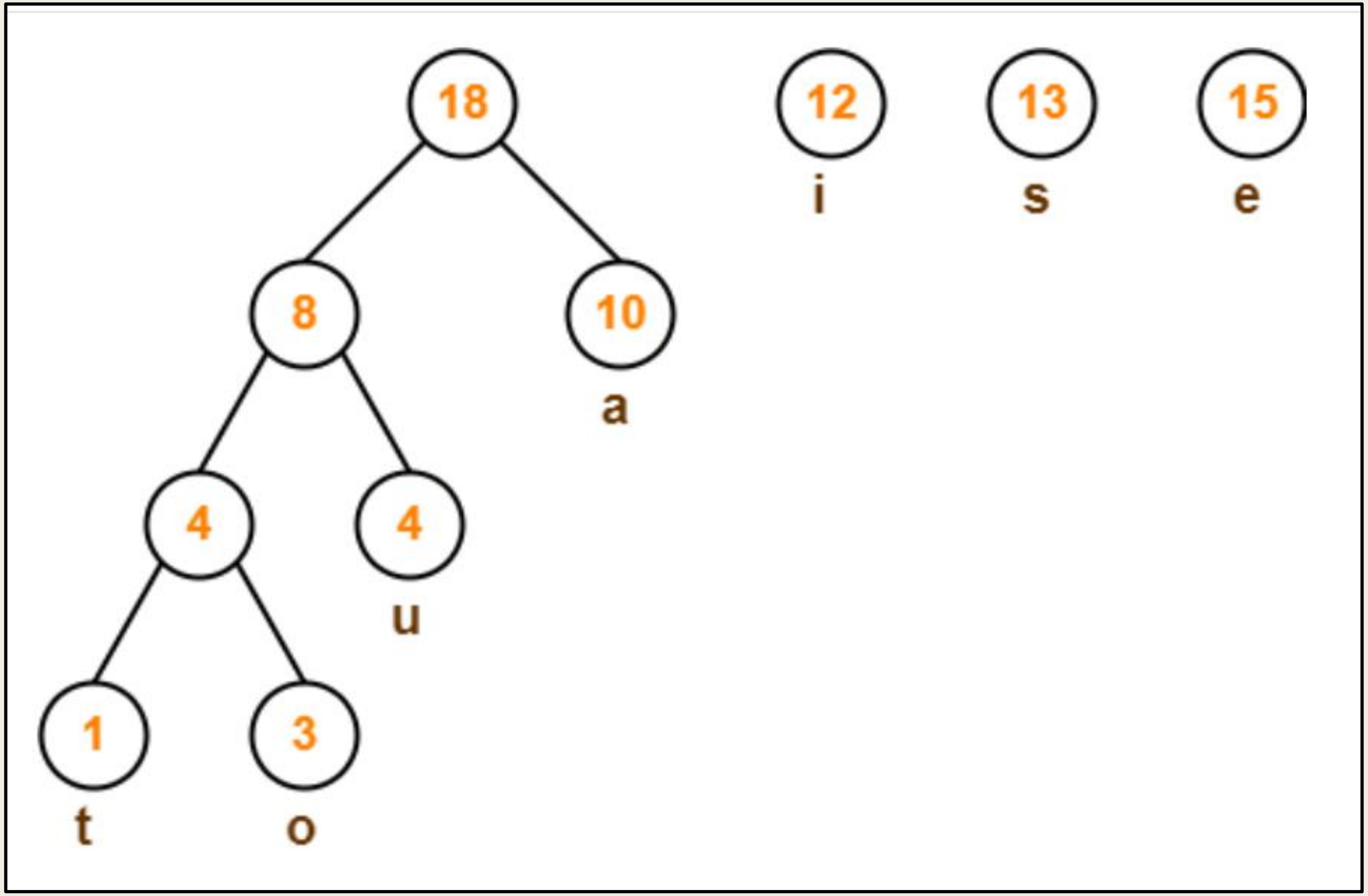




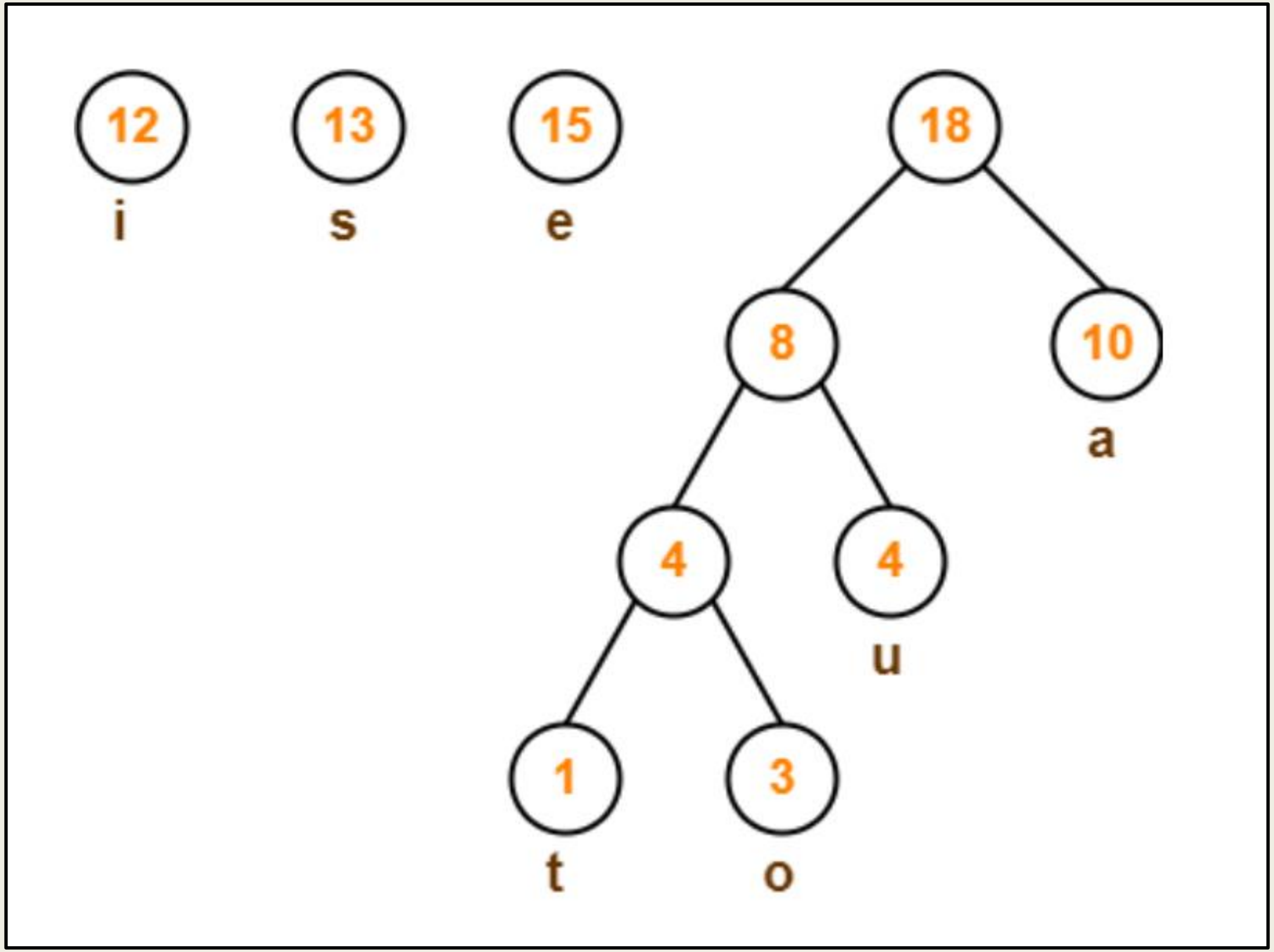
Huffman coding



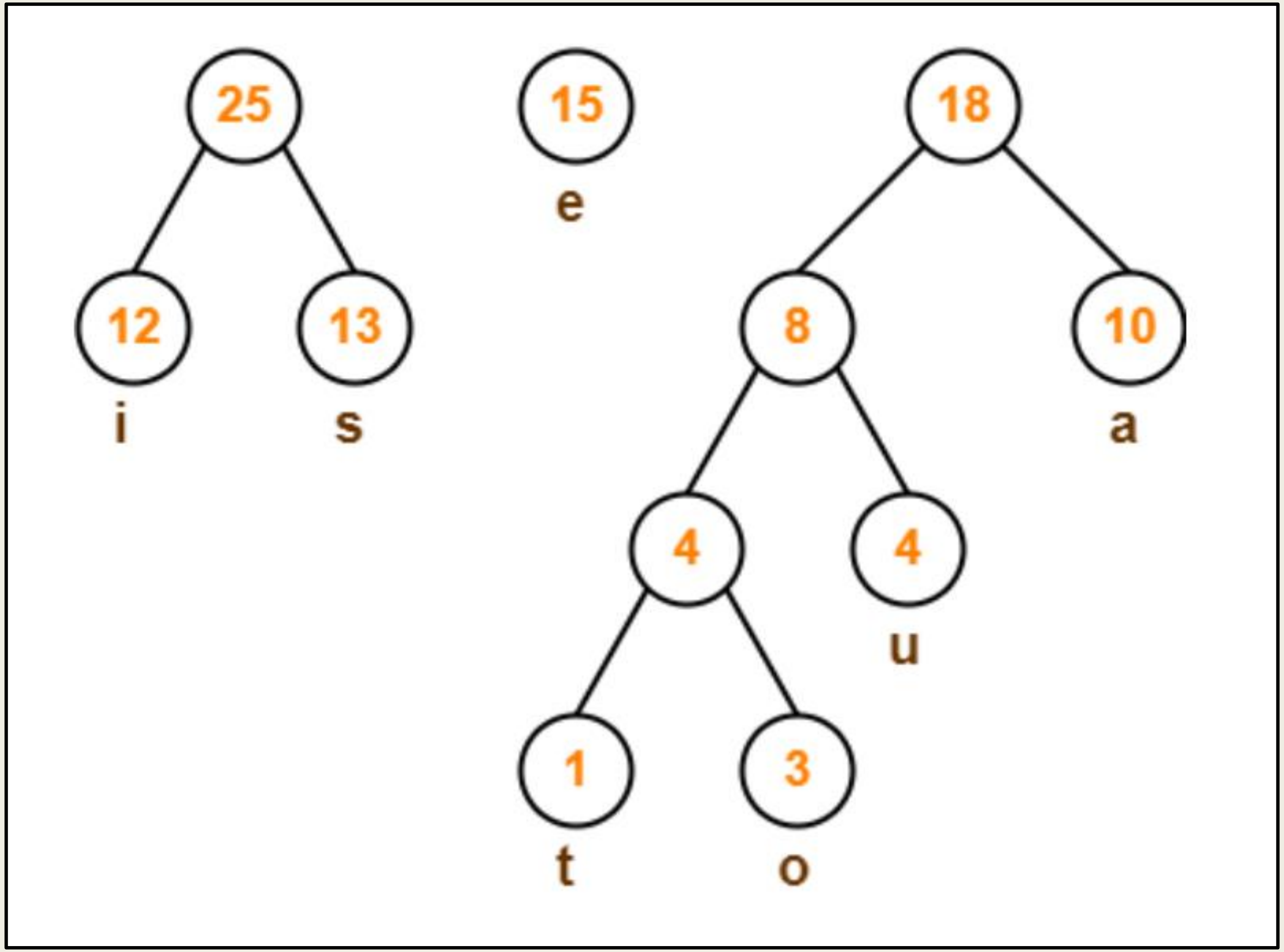
Huffman coding



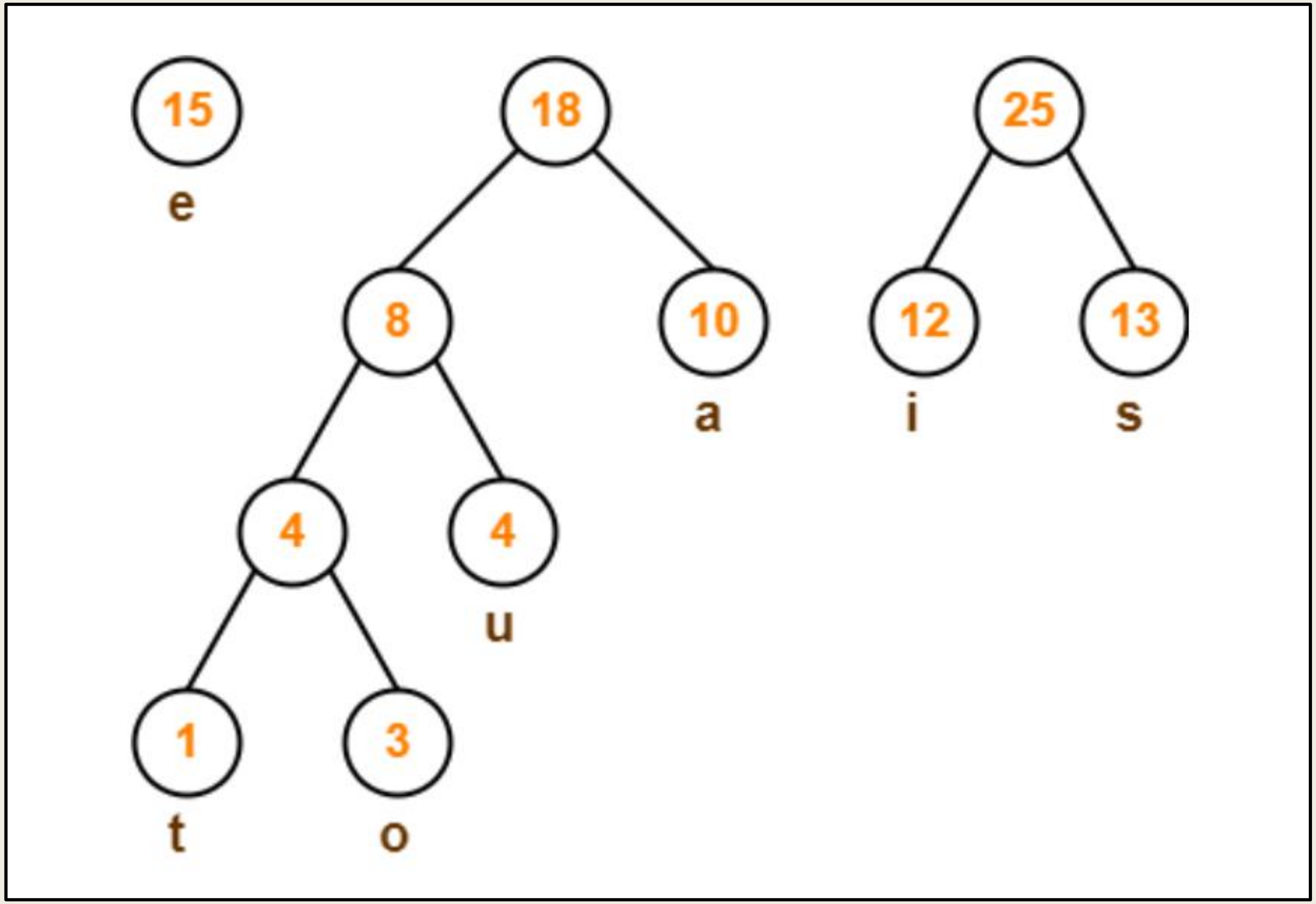
Huffman coding



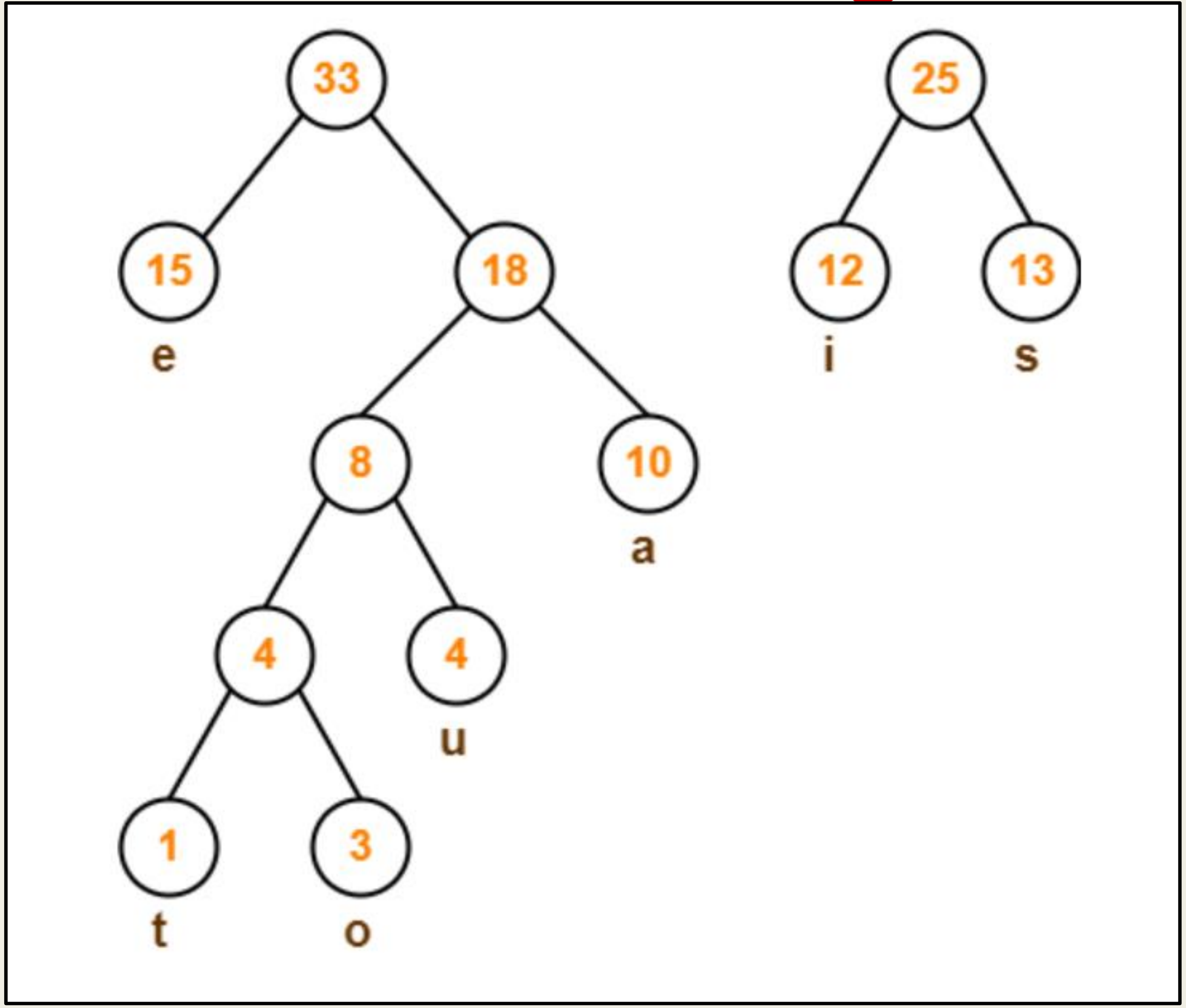
Huffman coding



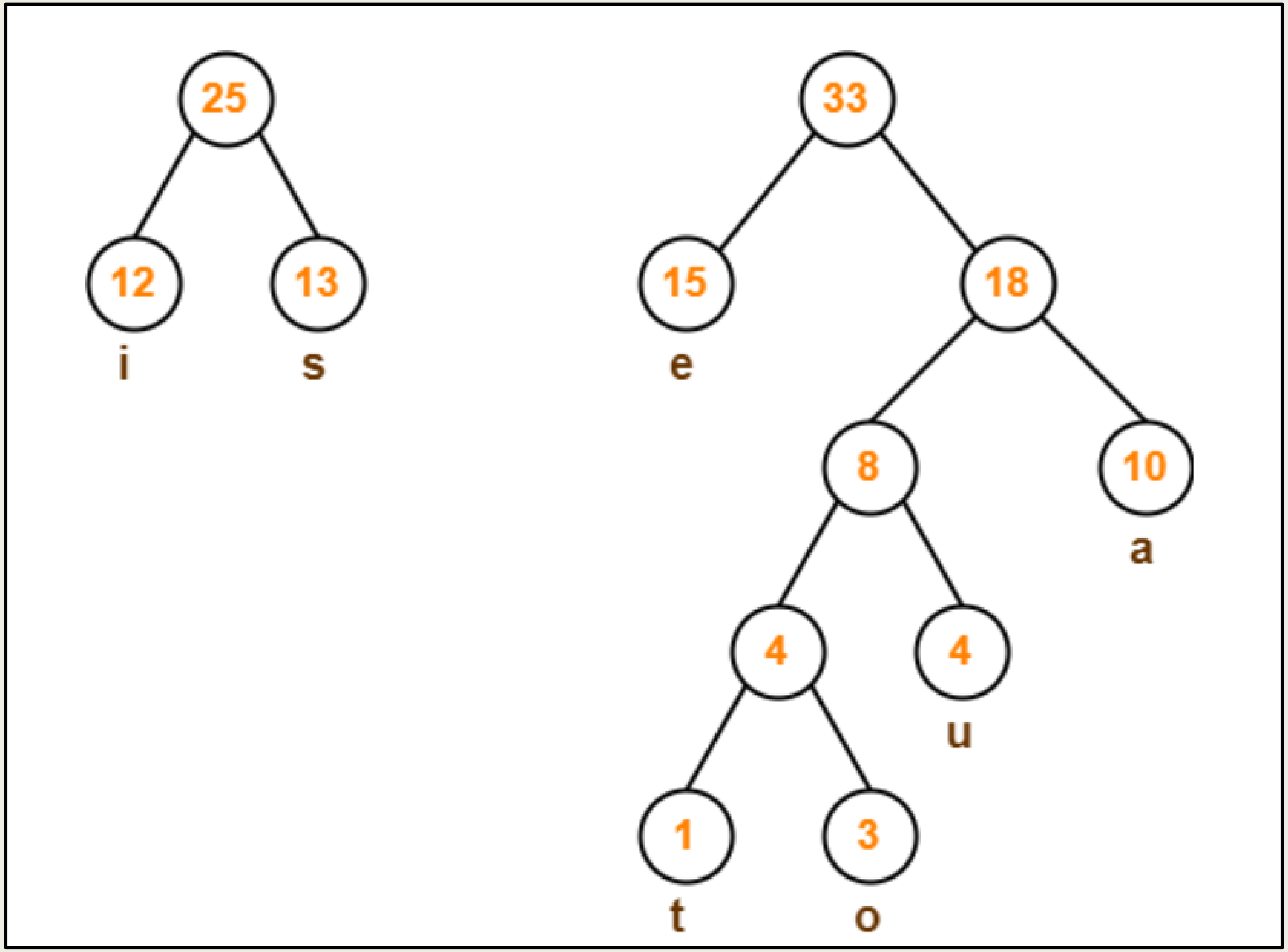
Huffman coding



Huffman coding

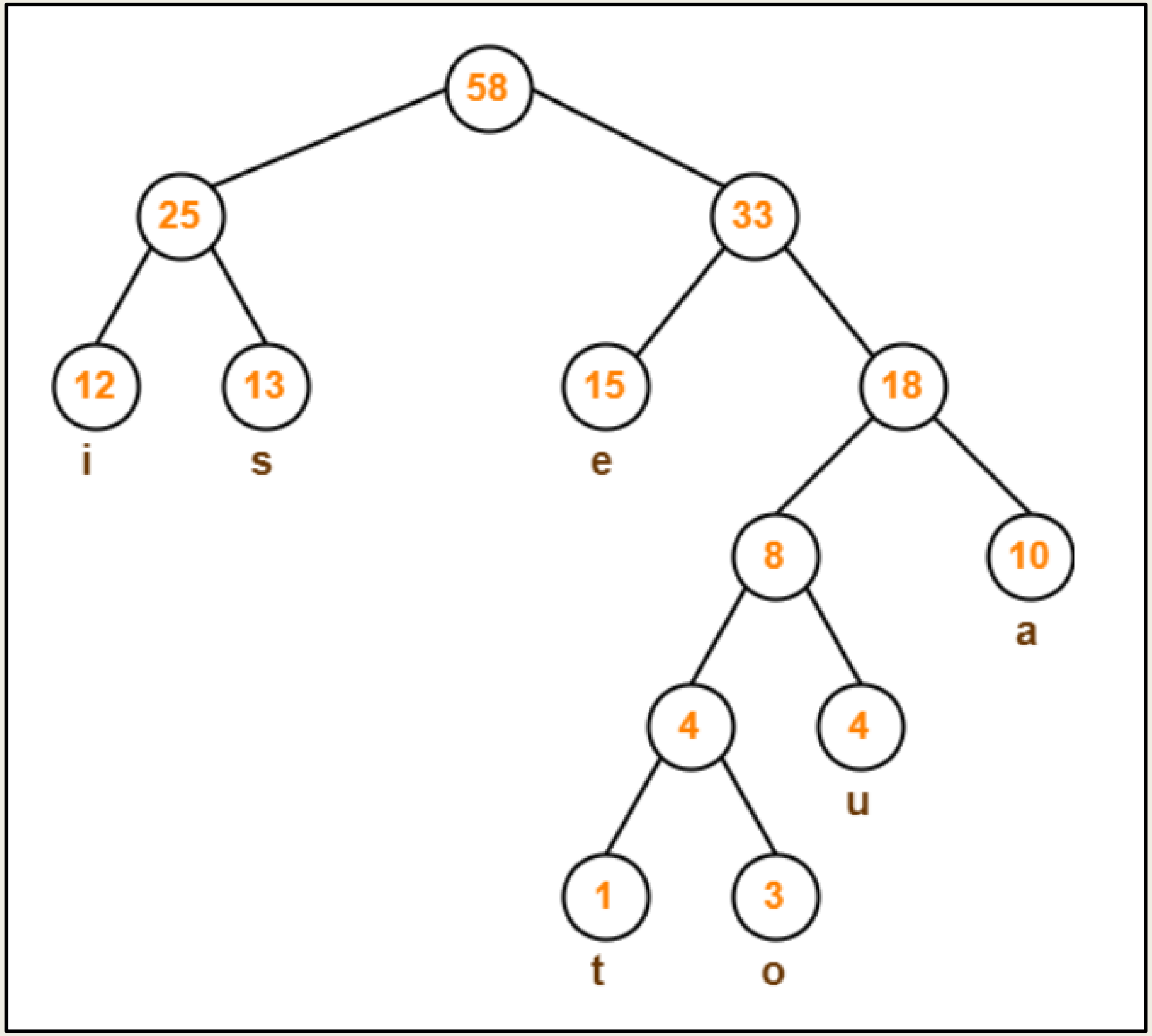


Huffman coding



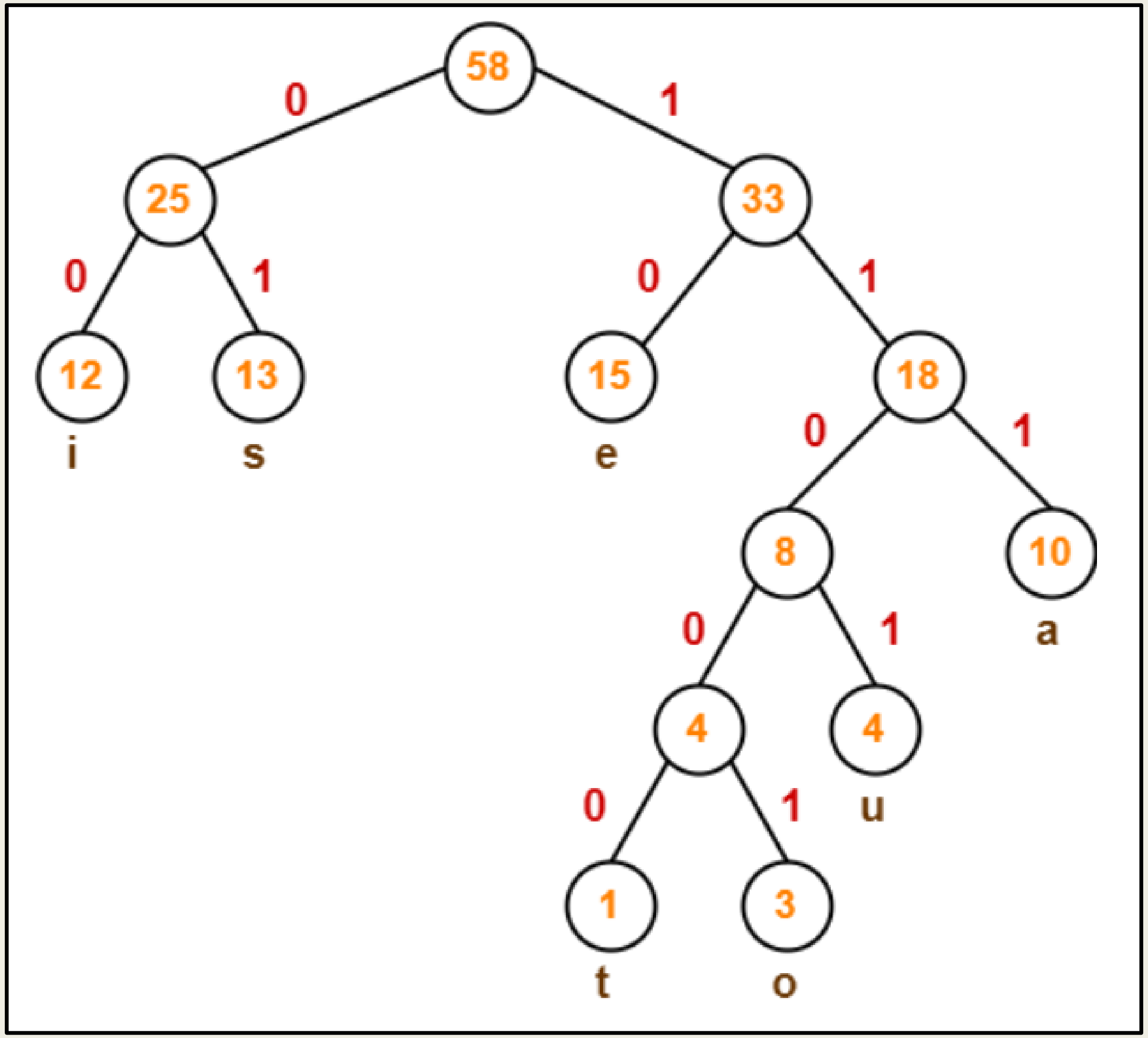
Huffman coding

Huffman Tree



Huffman coding

Assign weight '0' to the left edges and weight '1' to the right edges of the Huffman Tree



a = 111

e = 10

i = 00

o = 11001

u = 1101

s = 01

t = 11000

Huffman coding

- **Construct the Huffman tree and hence obtain the codewords for the following population**

Value	A	B	C	D	E	F
Frequency	5	25	7	15	4	12

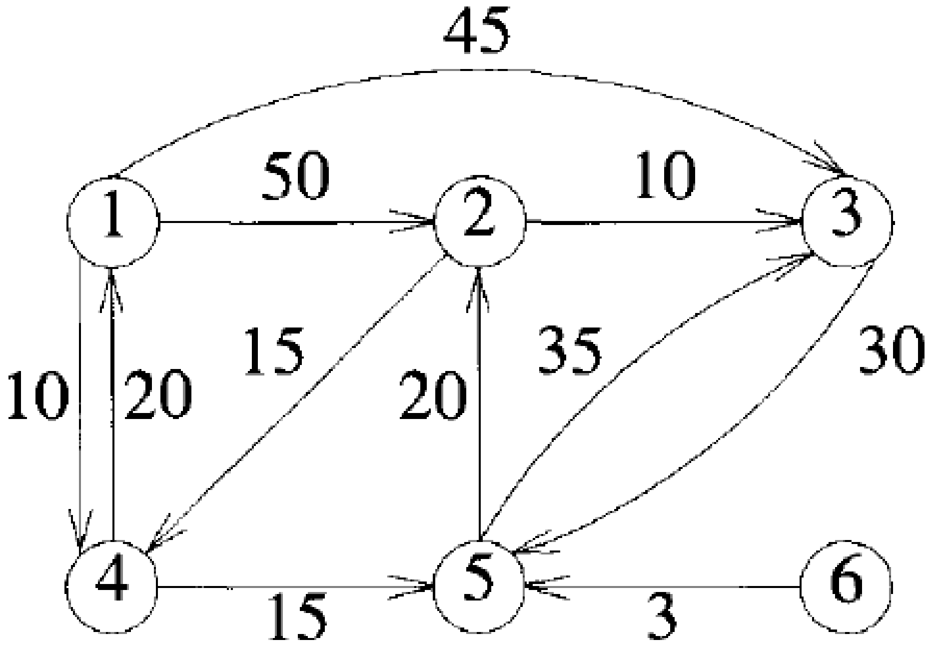
Single source shortest path

- Graphs can be used to model highway structure of a state/country.
- Vertices representing cities and edges representing the connecting roads
- The edges can be assigned weights which might represent distance/cost/time to drive across the cites
- If Mr. X wants to drive from point A to point B
- Following are the questions raised
- Is there a path from A to B?
- If there are multiple paths then which is the shortest path?

Single source shortest path

- **Graphs can be used to model highway structure of a state/country.**
- **Vertices representing cities and edges representing the connecting roads**
- **The edges can be assigned weights which might represent distance/cost/time to drive across the cites**
- **If Mr. X wants to drive from point A to point B**
- **Following are the questions raised**
  - ***Is there a path from A to B?***
  - ***If there are multiple paths then which is the shortest path?***

Single source shortest path



(a) Graph

	Path	Length
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

Single source shortest path

- The problem statement
- Given a weighted connected graph G=(V, E) and a source vertex s, determine the shortest paths from *s* to all the remaining vertices.
- The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.
- Major applications
  - *Transportation planning*
  - *Packet routing in Internet*
  - *Social networks*
  - *Speech recognition*
  - *Path finding video games*

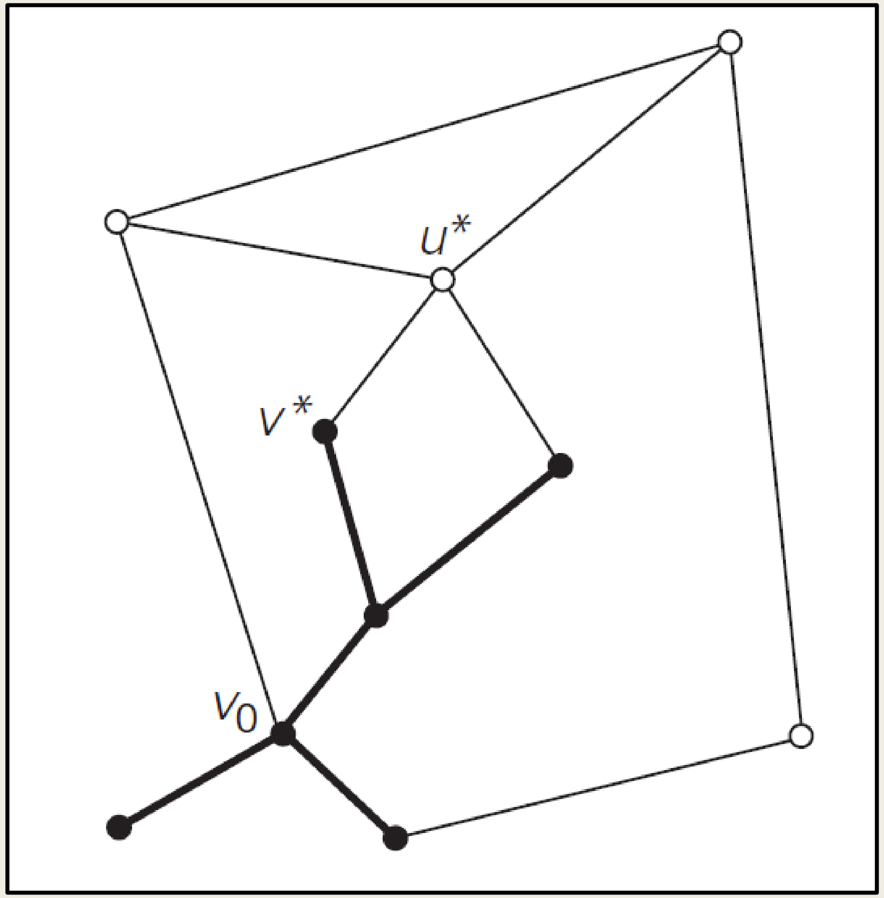
Single source shortest path

Dijkstra’s Algorithm

- Best-known algorithm for the single-source shortest-paths problem
- This algorithm is applicable to undirected and directed graphs with nonnegative weights only
- Dijkstra’s algorithm finds the shortest paths to a graph’s vertices in order of their distance from a given source
- First, it finds the shortest path from the source to a vertex nearest to it then to a second nearest, and so on
- In general, before its ith iteration commences, the algorithm has already identified the shortest paths to $i - 1$ other vertices nearest to the source.

Single source shortest path

- **Working of Dijkstra's algorithm**
- **Fringe vertices?**
  - *The set of vertices adjacent to the vertices in Ti*
- **How to identify next nearest vertex? u***



Single source shortest path

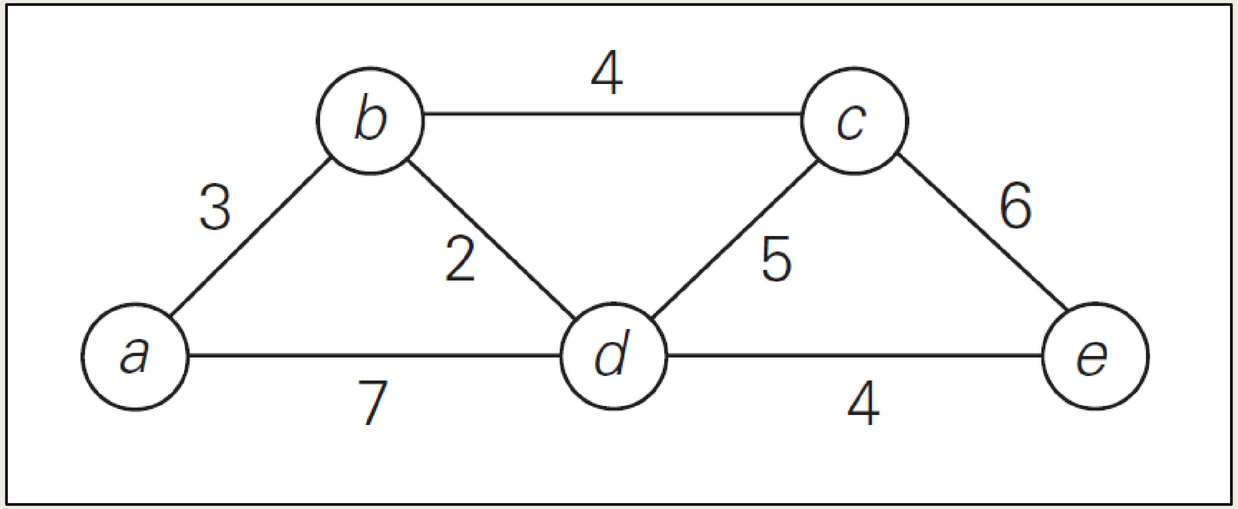
- After identifying do the following
- Move u* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u* by an edge of weight w(u*, u) such that du* + w(u*, u) < du, update the labels of u by u* and du* + w(u*, u), respectively.

Single source shortest path

- **Find the shortest paths from the source vertex a to all other vertices**

- **Adjacency Matrix**

$$\begin{matrix} 0 & 3 & \infty & 7 & \infty \\ 3 & 0 & 4 & 2 & \infty \\ \infty & 4 & 0 & 5 & 6 \\ 7 & 2 & 5 & 0 & 4 \\ \infty & \infty & 6 & 4 & 0 \end{matrix}$$

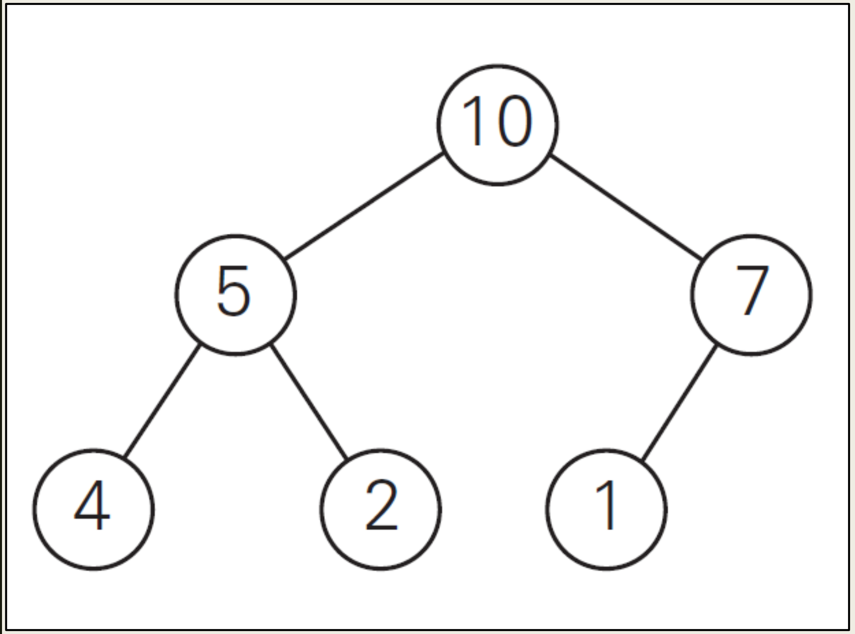


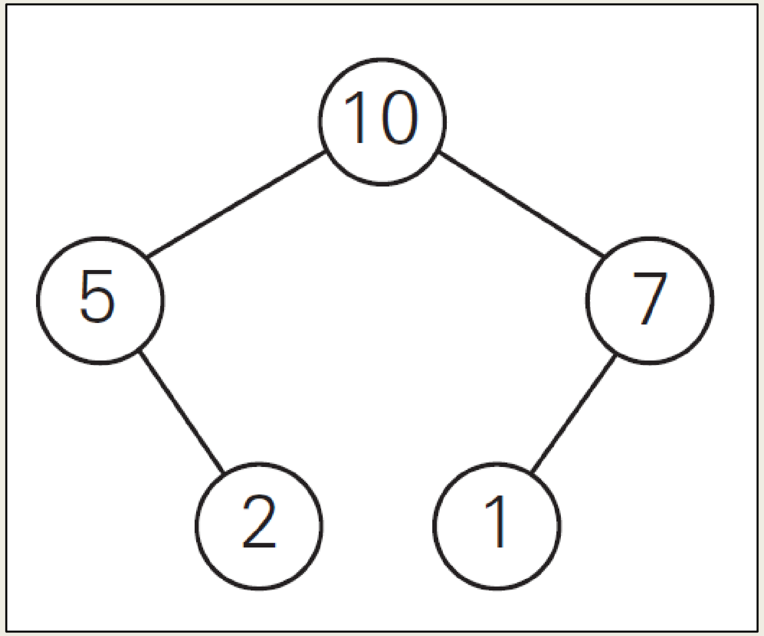
Solution

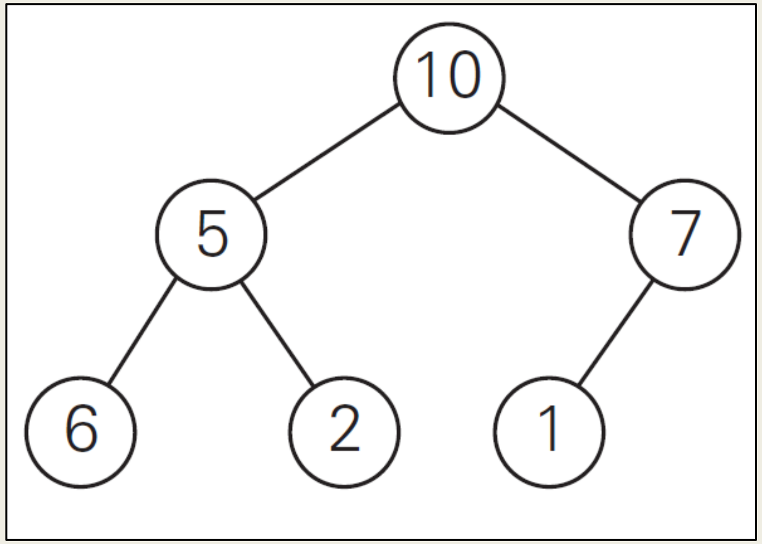
Heaps and Heap Sort

- **Partially ordered data structure that is especially suitable for implementing priority queues**
- **A heap can be defined as a binary tree with keys assigned to its nodes, one key per node, provided these two conditions are met:**
- **The shape property**
  - **The binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.**
- **The parental dominance or heap property**
  - **The key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)**

Heaps and Heap Sort







- Note
  - key values in a heap are ordered top down
  - there is no left-to-right order in key values;

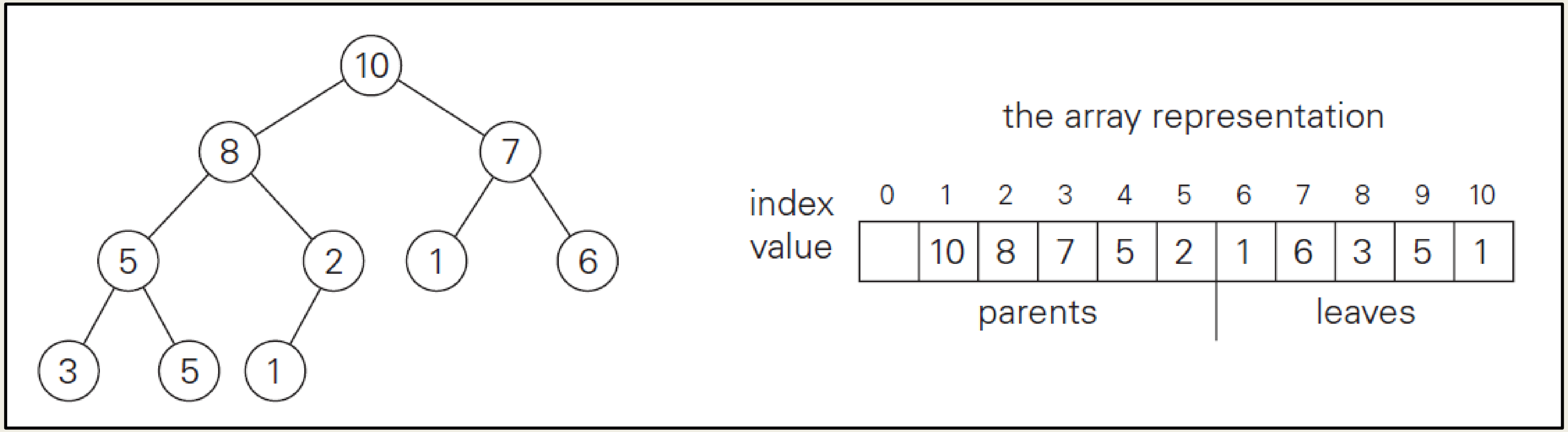
Heaps and Heap Sort

Properties of Heap

- There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\log_2 n$.
- The root of a heap always contains its largest element.
- A node of a heap considered with all its descendants is also a heap.
- A heap can be implemented as an array by recording its elements in the top down, left-to-right fashion
- It is convenient to store the heap's elements in positions 1 through n of such an array, leaving H[0] either unused

Heaps and Heap Sort

- **In an array representation of heaps**
  - the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
  - the children of a key in the array's parental position $i\ (1 \leq i \leq n/2)$ willbe in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position $i\ (2 \leq i \leq n)$ will be in position $\lfloor i/2 \rfloor$



Heaps and Heap Sort

ESTD:2001
An Institute with a Difference

How to construct a heap for a given list of keys?

- bottom-up heap construction
  - It initializes the essentially complete binary tree with n nodes by placing keys in the order given
  - then "heapifies" the tree
- top-down heap construction
  - Successive insertions of a new key into a previously constructed heap

Bottom-up heap construction

Example : construct a heap for the following list of numbers

2, 9, 7, 6, 5, 8.

Heaps and Heap Sort

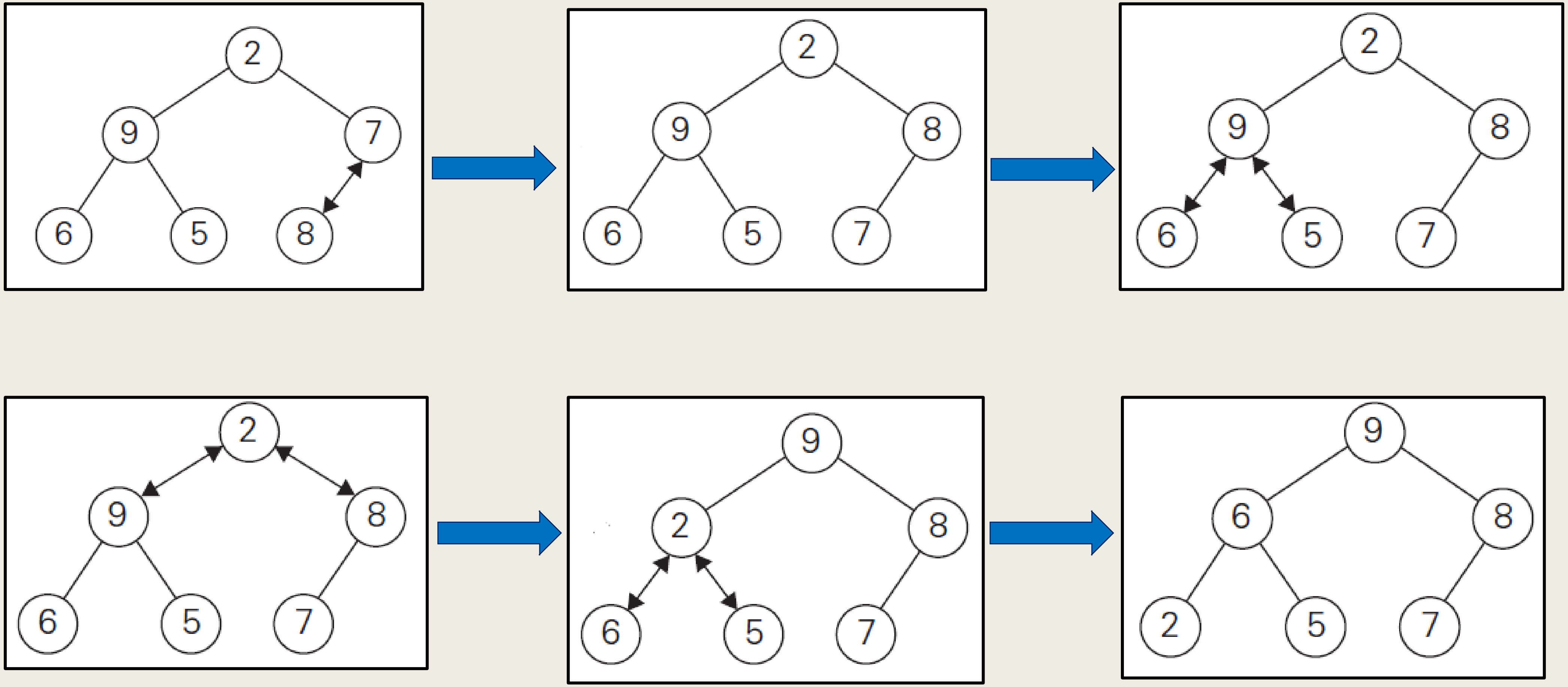
Bottom-up heap construction

- **It initializes the essentially complete binary tree with n nodes by placing keys in the order given**
- **Heapify**
  - Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node
  - If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position
  - This process continues until the parental dominance for K is satisfied.
  - After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor
  - The algorithm stops after this is done for the root of the tree.

Heaps and Heap Sort

Example : Construct a heap for the following list of numbers

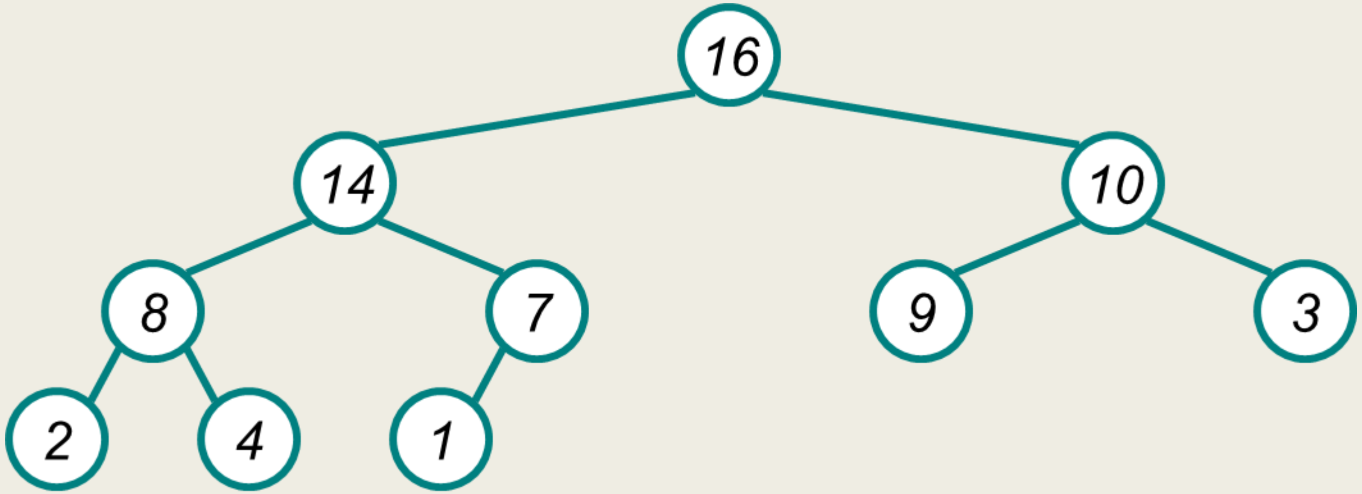
2, 9, 7, 6, 5, 8.



Heaps and Heap Sort

- **construct a heap for the following list of numbers**

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



Heaps and Heap Sort

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

//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$

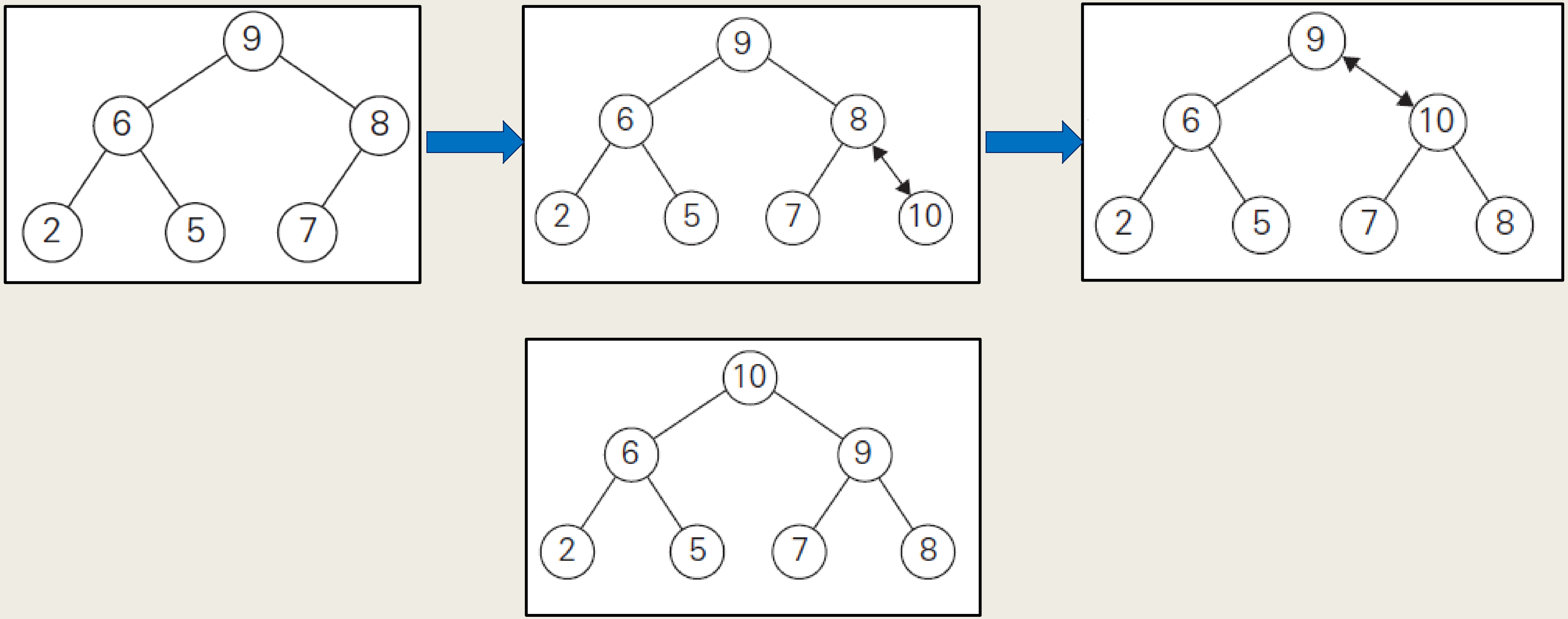
```
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

Heaps and Heap Sort

- **Top-down heap construction**
  - **Successive insertions of a new key into a previously constructed heap**
- **How to insert a new key K into a heap?**
  - First, attach a new node with key **K** in it after the last leaf of the existing heap.
  - Then shift **K** up to its appropriate place in the new heap as follows.
    - Compare K with its parent’s key: if the latter is greater than or equal to K, stop (the structure is a heap);
    - Otherwise, swap these two keys and compare K with its new parent.
    - This swapping continues until K is not greater than its last parent or it reaches the root

Heaps and Heap Sort

- **top-down heap construction**
  - **Successive insertions of a new key into a previously constructed heap**



Heaps and Heap Sort

- **How to delete an item form the heap?**
- **Special case- Deletion of root**

Maximum Key Deletion from a heap

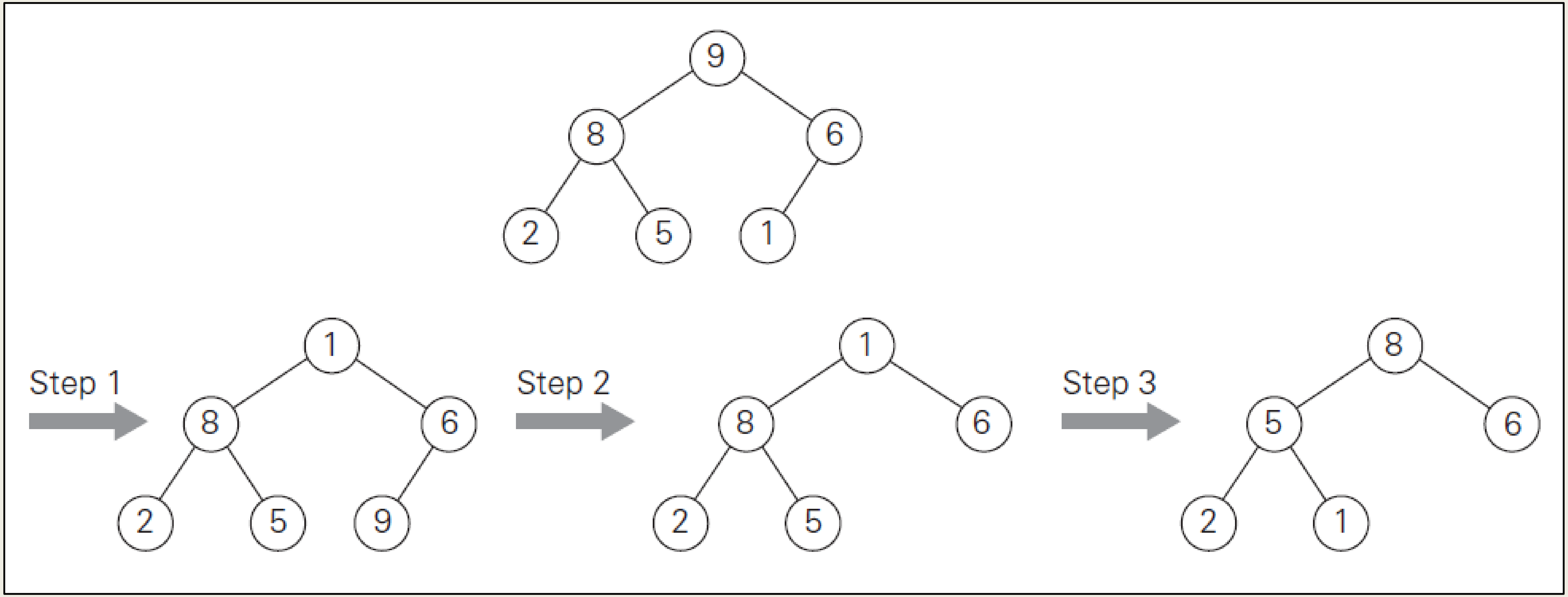
Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Heaps and Heap Sort

Steps illustrating the deletion of root element from the heap



Heaps and Heap Sort

Heap Sort

- **An interesting sorting algorithm discovered by J. W. J. Williams**
- **This is a two-stage algorithm that works as follows.**
  - **Stage 1** (heap construction): **Construct a heap for a given array.**
  - **Stage 2** (maximum deletions): **Apply the root-deletion operation n − 1 times to the remaining heap.**

Heaps and Heap Sort

Sort the following elements using heap sort

2, 9, 7, 6, 5, 8

Stage 1 (heap construction)

2	9	**7**	6	5	8
2	**9**	8	6	5	7
2	9	8	6	5	7
9	**2**	8	6	5	7
9	6	8	2	5	7

→

Stage 2 (maximum deletions)

9	6	8	2	5	7		
7	6	8	2	5	I	**9**	
8	6	7	2	5			
5	6	7	2	I	**8**		
7	6	5	2				
2	6	5	I	**7**			
6	2	5					
5	2	I	**6**				
5	2						
2	I	**5**					
2							

GREEDY METHOD