

# CSC 4520/6520

# Design & Analysis of Algorithms

---

FINAL REVIEW

Abdullah Bal, PhD

# CSC 4520/6520

# Design & Analysis of Algorithms

---

## CHAPTER 6: TRANSFORM-AND-CONQUER (INSTANCE SIMPLIFICATION)

Abdullah Bal, PhD

# Transform and Conquer

---

- Instance simplification: This group of techniques solves a problem by a transformation to a simpler/more convenient instance of the same problem.
- Representation change: A different representation of the same instance
- Problem reduction: A different problem for which an algorithm is already available

# Instance simplification

---

Presorting

# Presorting

---

- Solve a problem's instance by transforming it into another simpler/easier instance of the same problem

## Presorting

- Many problems involving lists are easier **when list is sorted**, e.g.
- Searching
- Computing the median (selection problem)
- Checking if all elements are distinct (element uniqueness)

Also:

- Topological sorting helps solving some problems for dags.
- Presorting is used in many geometric algorithms.

# Example 1: Element Uniqueness with presorting

---

## Bruteforce

- Checking element uniqueness in an array
- The **bruteforce** algorithm compared pairs of the array's elements until either two equal elements were found or no more pairs were left.
- Its worst-case efficiency was in  $\Theta(n^2)$ .

# Example 1: Element Uniqueness with presorting

## Presorting

- Alternatively, we can sort the array first and then check only its consecutive elements:
  - if the array has equal elements, a pair of them must be next to each other, and vice versa.

## Example 1: Element Uniqueness with presorting

- The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements.

$$T(n) = T_{sort}(n) + T_{scan}(n)$$

- Since the former (sorting) requires at least  $n \log n$  comparisons and the latter (scanning) needs no more than  $n - 1$  comparisons, it is the sorting part that will determine the overall efficiency of the algorithm.



## Example 1: Element Uniqueness with presorting

- So, if we use a quadratic sorting algorithm here, the entire algorithm will not be more efficient than the brute-force one.
- But if we use a good sorting algorithm, such as merge sort, with worst-case efficiency in  $\Theta(n \log n)$ , the worst-case efficiency of the entire presorting-based algorithm will be also in  $\Theta(n \log n)$ :

# Example 1: Element Uniqueness with presorting

---

- Presorting-based algorithm

- Stage 1: sort by efficient sorting algorithm (e.g. merge sort)
- Stage 2: scan array to check pairs of adjacent elements
- Efficiency:  $\Theta(n \log n) + O(n) = \Theta(n \log n)$

- Brute force algorithm

- Compare all pairs of elements
- Efficiency:  $O(n^2)$

## Example 2: Computing a Mode

---

- A *mode* is a value that **occurs most often** in a given list of numbers.
- For example, for 5, 1, 5, 7, 6, 5, 7, **the mode is 5**.
- If several different values occur most often, any of them can be considered a mode.

# Example 2: Computing a Mode

---

## Bruteforce

- The **brute-force approach** to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.
- In order to implement this idea, we can **store the values already encountered**, along with **their frequencies**, in a **separate** list.
- On each iteration, the  $i^{\text{th}}$  element of the original list is compared with the values already encountered by traversing this **auxiliary list**.
- If a matching value is found, **its frequency is incremented**; otherwise, the current element is added to the list of distinct values seen so far **with a frequency of 1**.

## Example 2: Computing a Mode

---

- The worst-case input for this algorithm is a list with no equal elements.
- For such a list, its  $i^{\text{th}}$  element is compared with  $i - 1$  elements of the **auxiliary list of distinct values** seen so far before being added to the list with a frequency of 1.
- As a result, the worst-case number of comparisons made by this algorithm in creating the frequency list is  $\Theta(n^2)$
- The **additional  $n - 1$  comparisons** needed to find the largest frequency in the auxiliary list do **not change the quadratic worst-case** efficiency class of the algorithm.
- Efficiency:  $\Theta(n^2)$

# Example 2: Computing a Mode

---

## Presorting

- As an alternative, let us first sort the input.
- Then all equal values will be adjacent to each other.
- To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.
- The running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time.
- Consequently, with a  $n \log n$  sort, this method's worst-case efficiency will be in a better asymptotic class than the worst-case efficiency of the brute-force algorithm.
- Efficiency:  $\Theta(n \log n)$

## Example 3: Searching Problem

---

- Problem: Search for a given  $K$  in  $A[0..n-1]$

### Bruteforce

- The **brute-force solution** here is **sequential search**, which needs  $n$  comparisons in the worst case.
- Efficiency:  $\Theta(n)$

# Example 3: Searching Problem

---

- Presorting

- Presorting-based algorithm:

  - Stage 1: Sort the array by an efficient sorting algorithm

  - Stage 2: Apply binary search

- Assuming the most efficient  $n \log n$  sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{\text{sort}}(n) + T_{\text{search}}(n)$$

- Efficiency:  $\Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$



# Example 3: Searching Problem

---

- Brute force algorithm

  - Efficiency:  $\Theta(n)$

- Presorting-based algorithm

  - Efficiency:  $\Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$

- Good or bad?

- Why do we have our dictionaries, telephone directories, etc. sorted? If we are to search in the same list more than once, the time spent on **sorting might well be justified.**

# Instance simplification

---

## Gaussian Elimination

# Gaussian Elimination

---

- We are certainly familiar with systems of two **linear equations** in two unknowns:

$$\begin{aligned}a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2.\end{aligned}$$

- Unless the coefficients of one equation are proportional to the coefficients of the other, the system has a **unique solution**.
- The standard method for finding this solution is to use either equation to express one of the variables **as a function of the other** and then **substitute** the result into the other equation, yielding a linear equation whose solution is then used to find the value of the second variable.

# Gaussian Elimination (cont.)

---

- In many applications, we need to solve a system of  $n$  equations in  $n$  unknowns:

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

where  $n$  is a large number.

- Theoretically, we can solve such a system by generalizing the substitution method for solving systems of two linear equations.
- However, the resulting algorithm would be extremely cumbersome.

# Gaussian Elimination (cont.)

- There is a much **more elegant algorithm** for solving systems of linear equations called ***Gaussian elimination***.
- The idea of Gaussian elimination is to transform a system of  $n$  linear equations in  $n$  unknowns to an equivalent system (i.e., a system with the same solution as the original one) with an **upper- triangular coefficient matrix**, a matrix with **all zeros below its main diagonal**:

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

- In matrix notations, we can write this as  $Ax = b \implies A'x = b'$ ,

# Gaussian Elimination (cont.)

---

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.$$

- We can easily solve the system with an upper-triangular coefficient matrix by back substitutions as follows.

## Gaussian Elimination (cont.)

---

- First, we can immediately find the value of  $x_n$  from the last equation;
- Then we can substitute this value into the next to last equation to get  $x_{n-1}$ , and so on,
- Until we substitute the known values of the last  $n - 1$  variables into the first equation, from which we find the value of  $x_1$ .

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

# Gaussian Elimination (cont.)

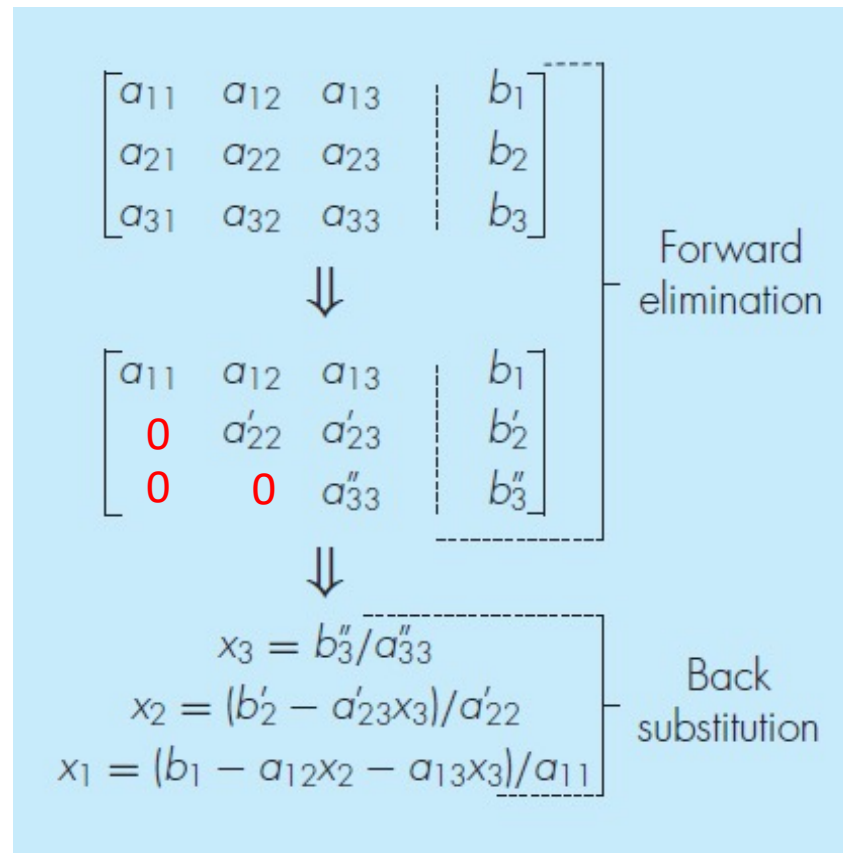
---

- How can we get from a system with an arbitrary coefficient matrix  $A$  to an equivalent system with an upper-triangular coefficient matrix  $A'$  ?
- We can do that through a series of the so-called *elementary operations*:
  - exchanging two equations of the system replacing an equation with its nonzero multiple
  - replacing an equation with a sum or difference of this equation and some multiple of another equation
  - Since no elementary operation can change a solution to a system, any system that is obtained through a series of such operations will have the same solution as the original one.



# Gaussian Elimination (cont.)

- Solve the system by Gaussian elimination.



# Gaussian Elimination (cont.)

---

- Let us see how we can get to a system with an upper-triangular matrix.
- First, we use  $a_{11}$  as a *pivot* to make all  $x_1$  coefficients zeros in the equations below the first one.
- Specifically, we replace the second equation with the difference between it and the first equation multiplied by  $a_{21}/a_{11}$  to get an equation with a zero coefficient for  $x_1$ .

# Gaussian Elimination (cont.)

---

- Doing the same for **the third, fourth, and finally  $n$ th equation**—with the multiples  $a_{31}/a_{11}$ ,  $a_{41}/a_{11}$ ,  $\dots$ ,  $a_{n1}/a_{11}$  of the first equation, respectively—makes all the coefficients of  $x_1$  below the first equation zero.
- Then we get rid of all the coefficients of  $x_2$  by subtracting an appropriate multiple of the second equation from each of the equations below the second one.
- Repeating this elimination for each of the first  $n - 1$  variables **ultimately yields a system with an upper-triangular coefficient matrix.**
- We can operate with just a system's coefficient matrix augmented, as its  $(n + 1)^{\text{st}}$  column, with the equations' right-hand side values.

# Pseudocode of Gaussian Elimination

---

- Stage 1: Reduction to an upper-triangular matrix

```
for  $i \leftarrow 1$  to  $n-1$  do
  for  $j \leftarrow i+1$  to  $n$  do
    for  $k \leftarrow i$  to  $n+1$  do
       $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$ 
```

- Stage 2: Back substitutions

```
for  $j \leftarrow n$  down to  $1$  do
   $t \leftarrow 0$ 
  for  $k \leftarrow j+1$  to  $n$  do
     $t \leftarrow t + A[j, k] * x[k]$ 
   $x[j] \leftarrow (A[j, n+1] - t) / A[j, j]$ 
```

- Efficiency:  $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

# CSC 4520/6520

# Design & Analysis of Algorithms

---

## CHAPTER 6: TRANSFORM-AND-CONQUER

### (AVL)

Abdullah Bal, PhD  
Spring 2024

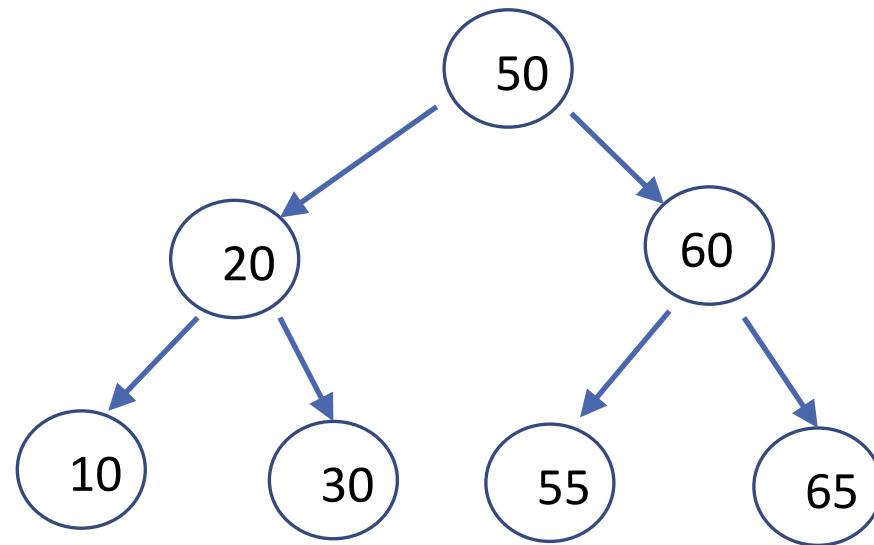
# Taxonomy of Searching Algorithms

---

- List searching
  - sequential search
  - binary search
  - interpolation search
- Tree searching
  - binary search tree
  - binary balanced trees: AVL trees, red-black trees
  - multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees
- Hashing
  - open hashing (separate chaining)
  - closed hashing (open addressing)

# Binary Search Trees

- Binary search tree is a **binary tree** whose nodes contain elements of a set of **orderable items**,
  - **One element** per node,
  - All elements in **the left subtree are smaller** than the element in the subtree's root,
  - All the elements in the **right subtree are greater** than it.



Height  
Min.  $\log n$   
Max.  $n$

# Example: Binary Search Tree

---

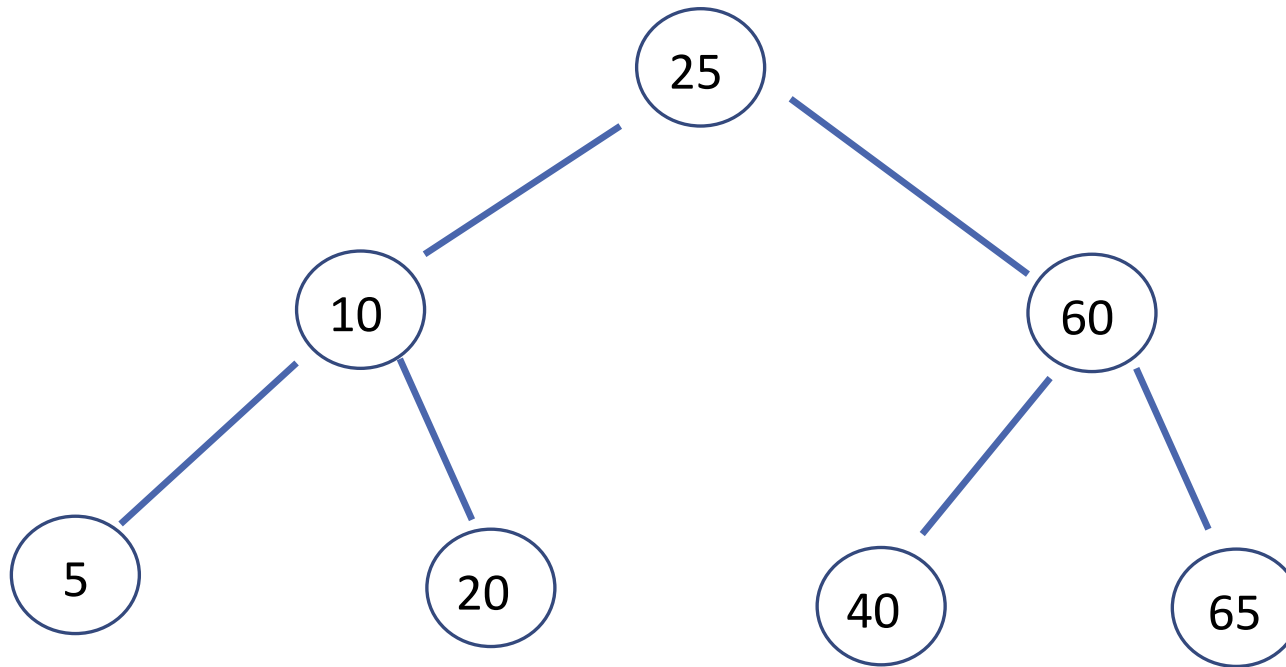
Keys: 25, 60, 10, 65, 20, 5, 40



# Example: Binary Search Tree

---

Keys: 25, 60, 10, 65, 20, 5, 40



Height:  
 $\log n$

# Example: Binary Search Tree

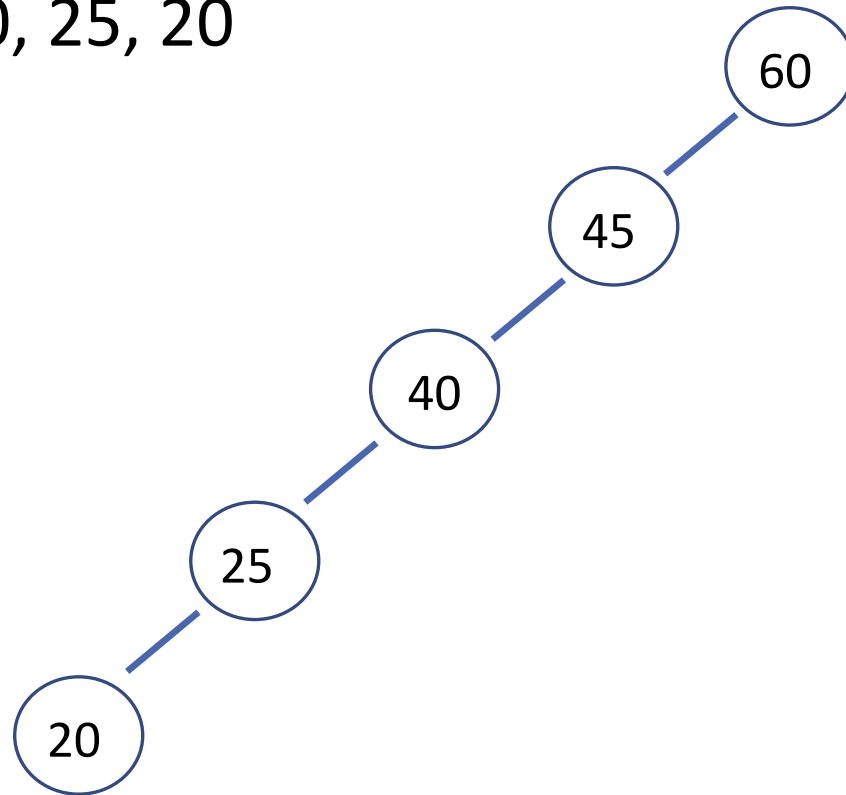
---

Keys: 60, 45, 40, 25, 20, 10, 5

# Example: Binary Search Tree

---

Keys: 60, 45, 40, 25, 20



Height  
 $n$

# Binary Search Trees

---

- This transformation from a set to a binary search tree is an example of the **representation-change technique**.
- We gain in the time efficiency of searching, insertion, and deletion, which are all in  $\Theta(\log n)$ , but only in the **average case**.
- In the **worst case**, these operations are in  $\Theta(n)$  because the tree can degenerate into a severely unbalanced one with its height equal to  $n - 1$ .

# Binary Search Trees

---

- Computer scientists have expended a lot of effort in trying to find a structure that preserves the good properties of the classical binary search tree
- Principally, the **logarithmic efficiency** of the dictionary operations **avoiding its worst-case degeneracy**.
- They have come up with **two approaches**:
  - **Instance-simplification** variety
  - **Representation-change** variety

# SELF-BALANCING AVL TREE



# The Instance-simplification Variety (AVL)

---

- An unbalanced binary search tree is transformed into a balanced one.
- Such trees are called *self-balancing*.
- An *AVL tree* requires the difference between the heights of the left and right subtrees of every node *never exceed 1*.
- If an insertion or deletion of a new node creates a tree with a violated balance requirement, the tree is restructured by one of a family of special transformations called *rotations* that restore the balance required.

# The Representation-change Variety (2-3 Trees)

- Allow **more than one element in a node** of a search tree.
- Specific cases of such trees are **2-3 trees**, **2-3-4 trees**, and more general and important **B-trees**.
- They differ in the number of elements admissible in a single node of a search tree, but **all are perfectly balanced**.
- We discuss the simplest case of such trees, the 2-3 tree, in this section, leaving the discussion of *B*-trees for Chapter 7.



# Balanced Search Trees

---

- Attractiveness of binary search tree is marred by the **worst-case efficiency**.  
Two ideas to overcome it are:
- To **rebalance** binary search tree when a new insertion makes the tree “too unbalanced”
  - AVL trees
  - Red-black trees
- To allow **more than one key** per node of a search tree
  - 2-3 trees
  - 2-3-4 trees
  - B-trees

# Balanced trees: AVL

---

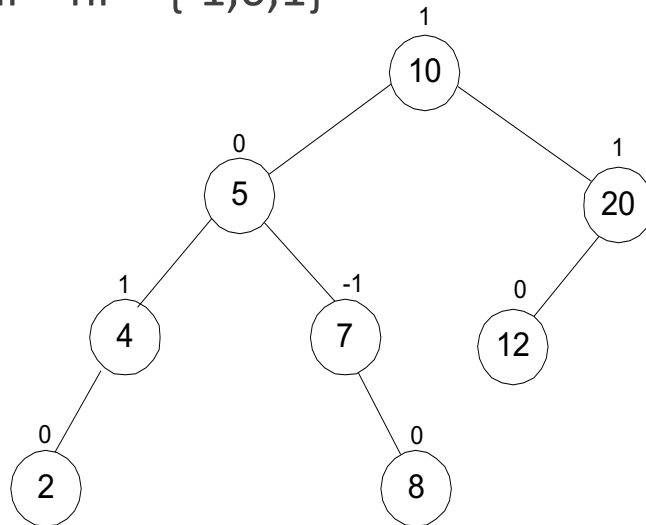
- AVL (**A**delson-**V**elsky-**L**andis) trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis, after whom this data structure is named.

# AVL trees

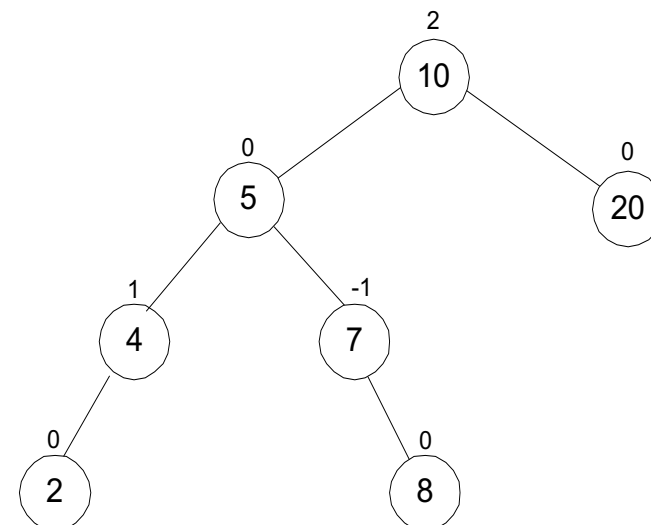
- An AVL tree is a binary search tree in which the **balance factor** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1.

**balance factor** = height of the left subtree – height of the right subtree

**balance factor** =  $hl - hr = \{-1, 0, 1\}$



AVL Tree



Not AVL Tree

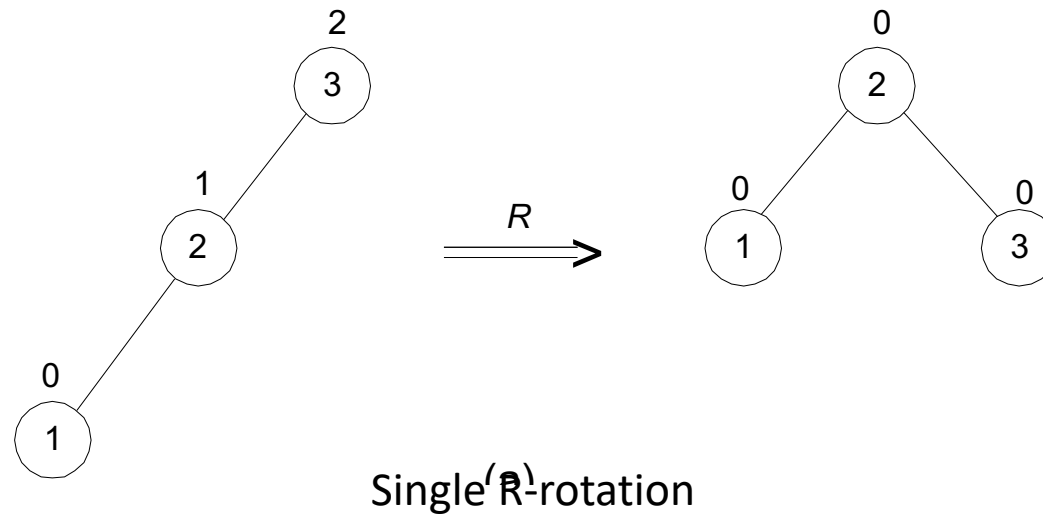
# Rotations

---

- If an **insertion** of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.
- If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.
- There are only **four types of rotations**; in fact, two of them are mirror images of the other two.

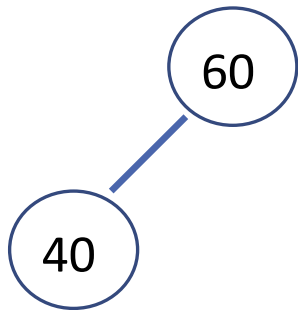
# Single Right Rotation (R-Rotation)

- The first rotation type is called the *single right rotation*, or *R-rotation*.
- Note that this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

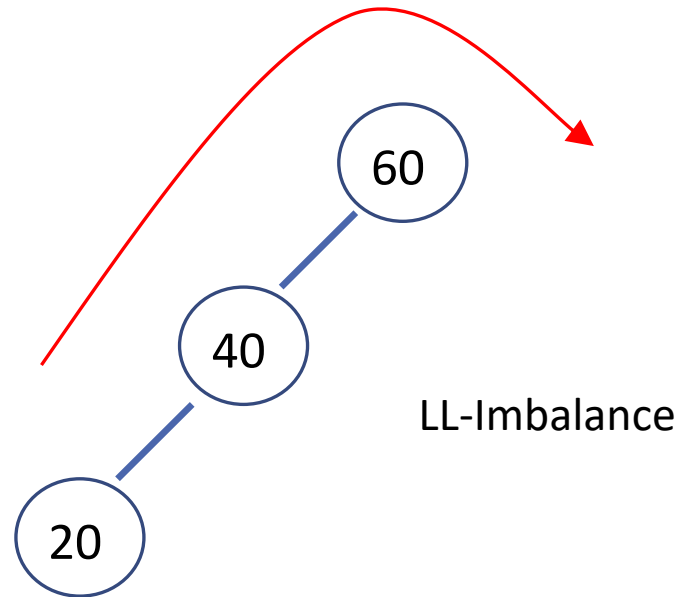


# Example: Single R Rotation

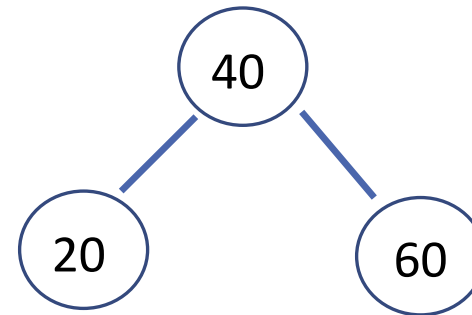
Keys: 60,40



Insert: 20

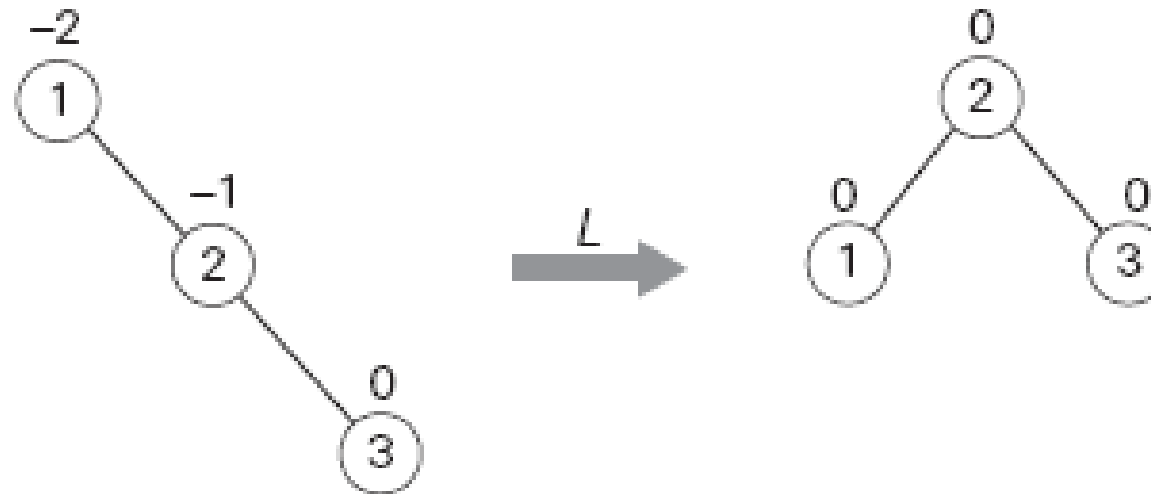


R- Rotation



# Single Left Rotation

- The symmetric *single left rotation*, or *L-rotation*, is the **mirror image** of the single R-rotation.
- It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of  $-1$  before the insertion.

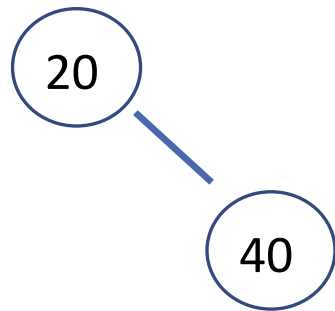


Single L-rotation

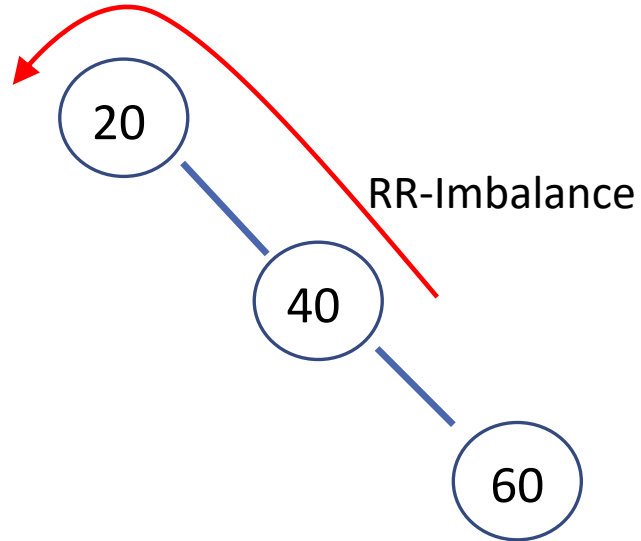
# Example: Single L Rotation

---

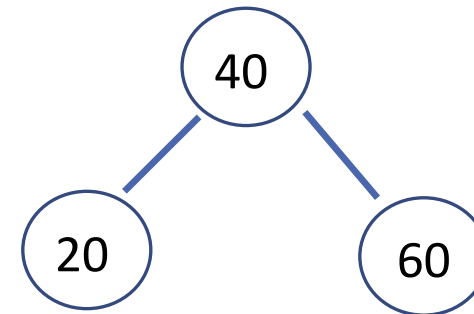
Keys: 20,40



Insert: 60



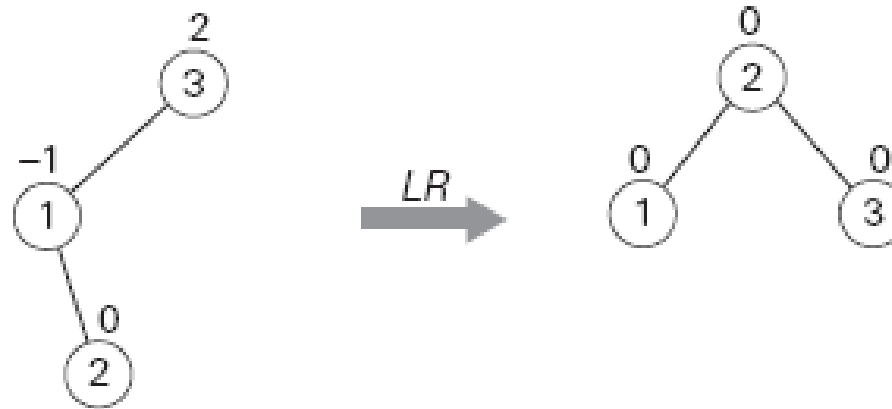
L- Rotation





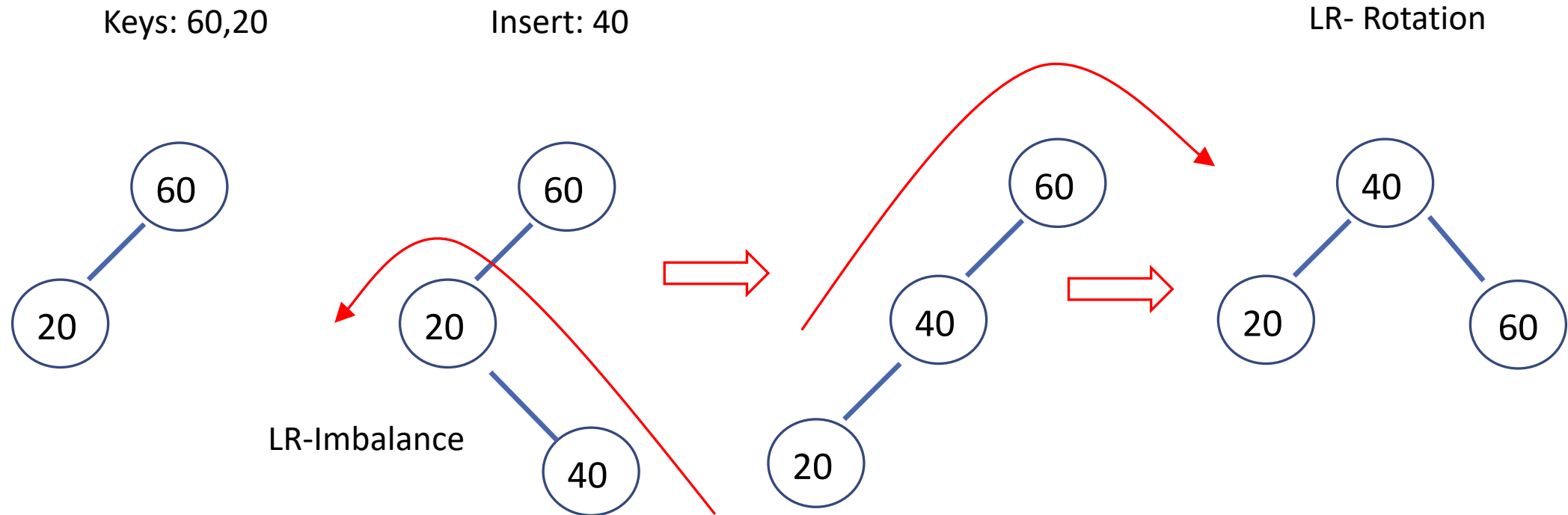
# Double Left-Right (LR) Rotation

- The second rotation type is called the **double left-right rotation (LR-rotation)**.
- We perform the *L*-rotation of the left subtree of root *r* followed by the *R*-rotation of the new tree rooted at *r*.
- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



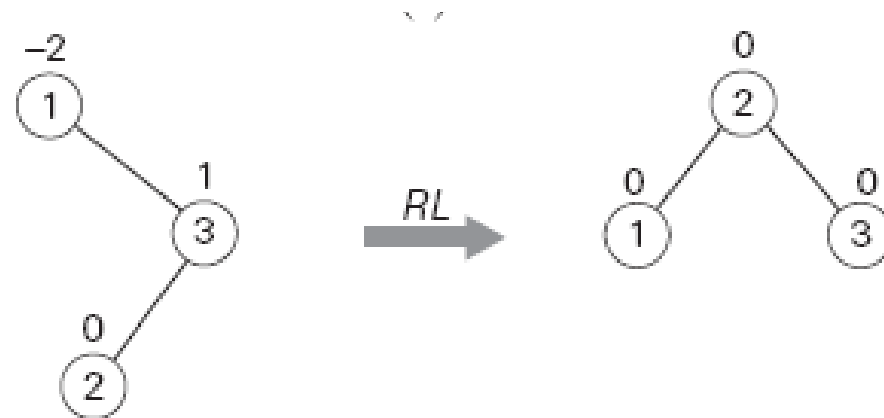
Double LR-rotation

# Example: Double LR Rotation



# Double Right-Left (RL) Rotation

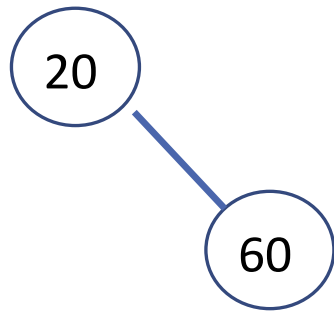
- The **double right-left rotation (RL-rotation)** is the mirror image of the double LR-rotation.
- We perform the *L*-rotation of the left subtree of root *r* followed by the *R*-rotation of the new tree rooted at *r*.
- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



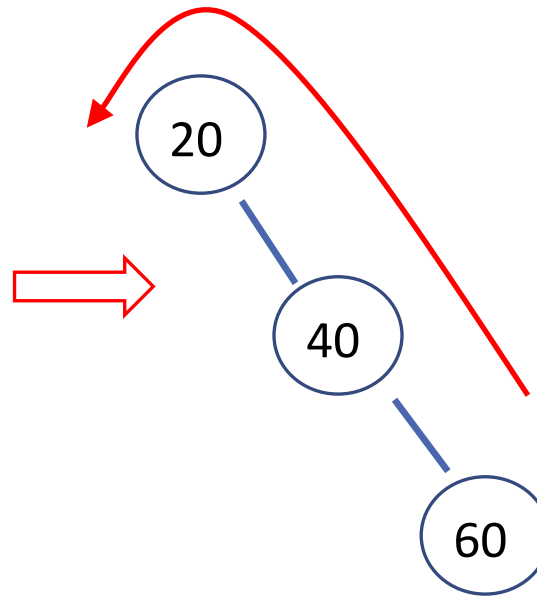
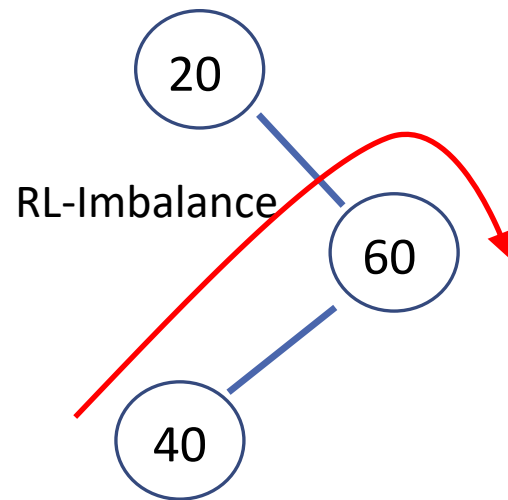
Double RL-rotation

# Example: Double RL Rotation

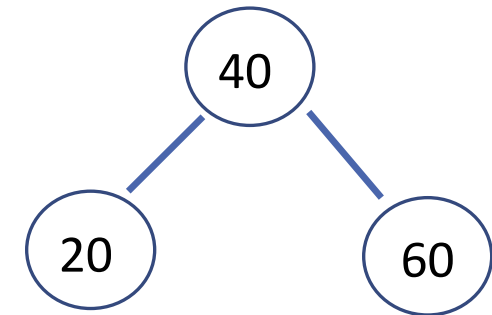
Keys: 20,60



Insert: 40

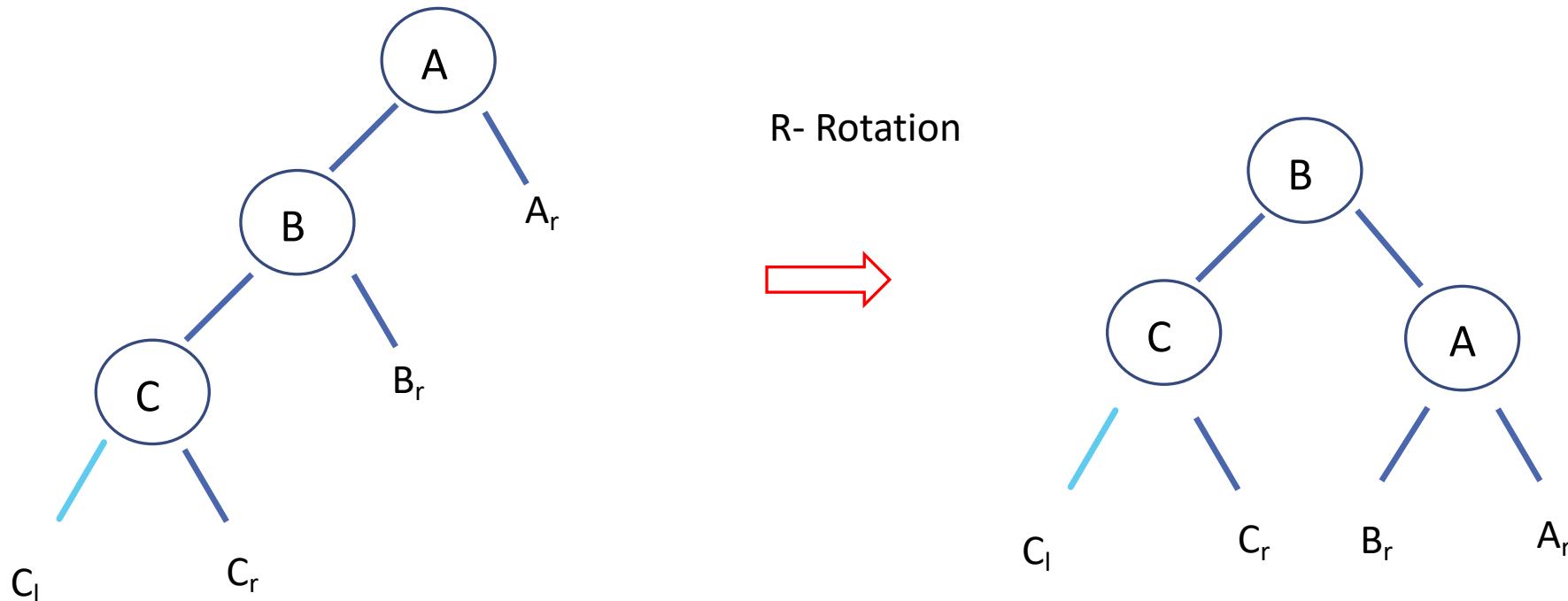


RL- Rotation



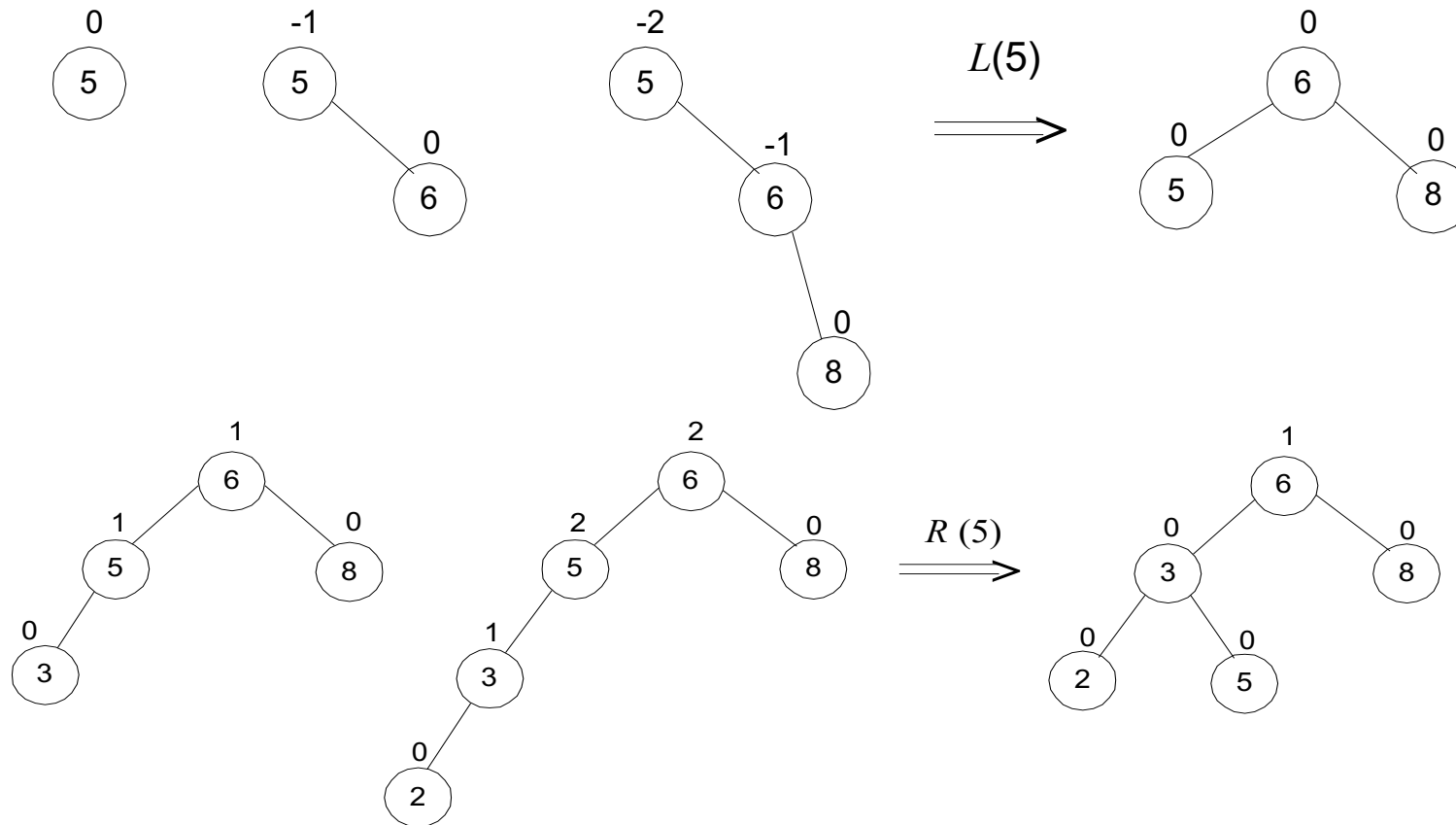
# General Case: Single R Rotation

- Rotations guarantee that a resulting tree is balanced, but they should also preserve the basic requirements of a binary search tree.
- And the same relationships among the key values hold, as they must, for the balanced tree after the rotation.



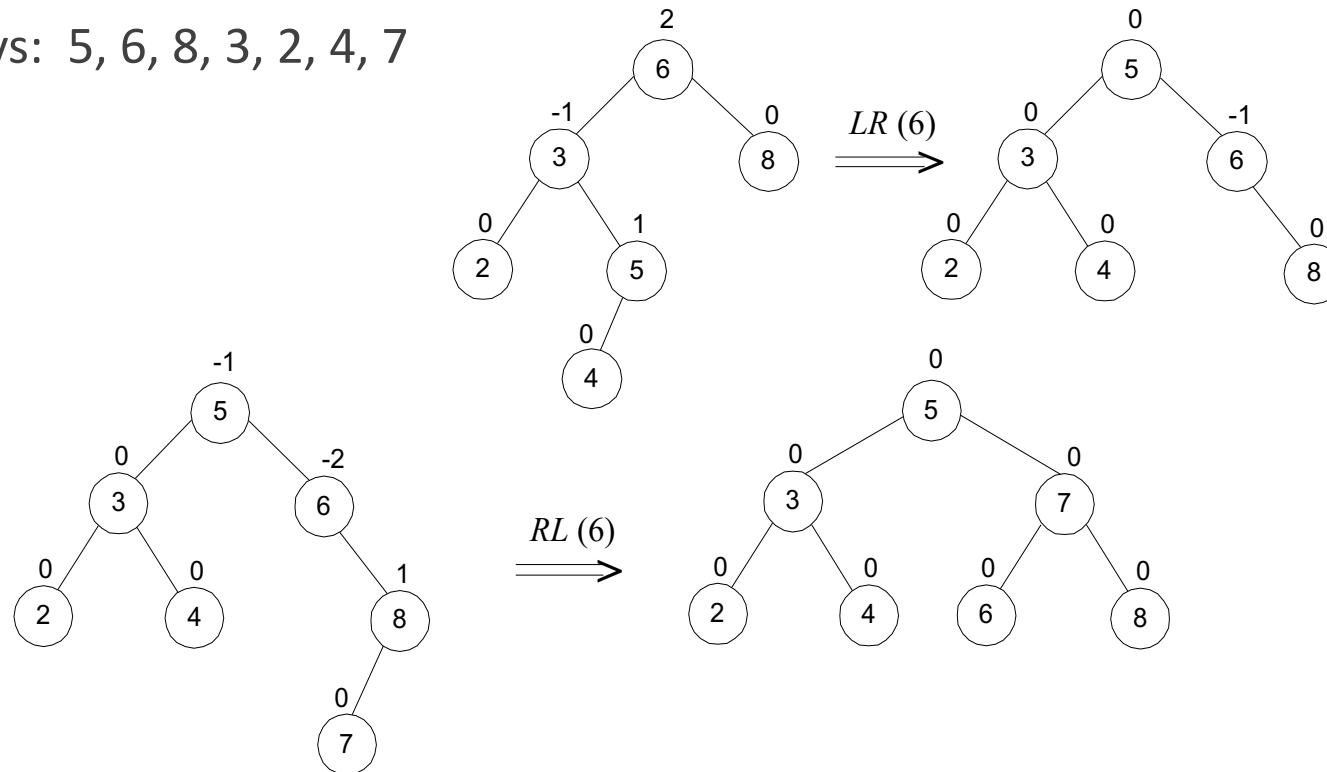
# Example 1: AVL tree construction

- Construct an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



# Example 1: AVL tree construction (cont.)

- If there are several nodes with the  $\pm 2$  balance, the rotation is done for the tree rooted at the unbalanced node that is the **closest to the newly inserted leaf**.
- Keys: 5, 6, 8, 3, 2, 4, 7



## Example 2: AVL tree construction

---

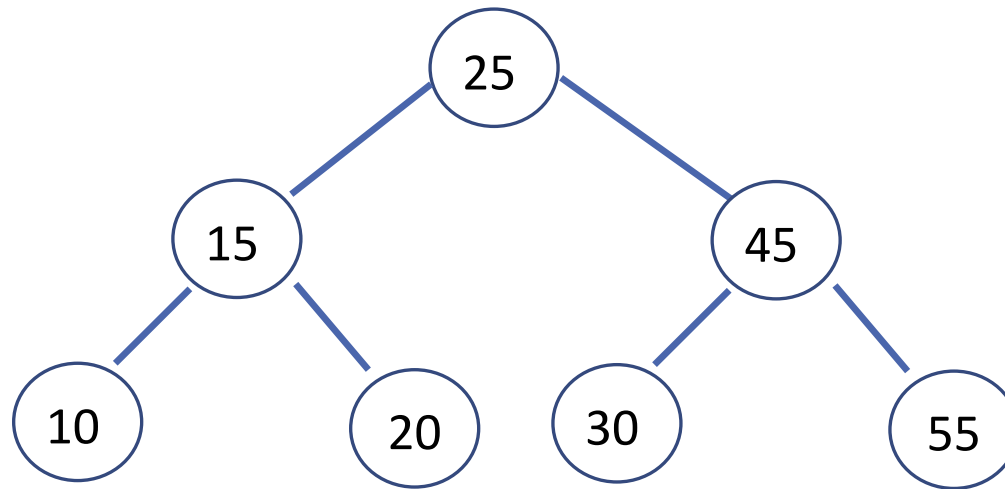
Construct an AVL tree for the list 45, 15, 10, 25, 30, 20, 55



## Example 2: AVL Tree

---

AVL tree for the list 45, 15, 10, 25, 30, 20, 55



# Analysis of AVL trees

---

- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$
- average height:  $1.01 \log_2 n + 0.1$  for large  $n$  (found empirically)
- Search and insertion are  $O(\log n)$
- Deletion is more complicated but is also  $O(\log n)$
- Disadvantages:
  - frequent rotations
  - Complexity
- A similar idea: *red-black trees*

# CSC 4520/6520

# Design & Analysis of Algorithms

---

CHAPTER 6: TRANSFORM-AND-CONQUER

(REPRESENTATION CHANGE-2\_3 TREE)

Abdullah Bal, PhD

# 2-3 Tree

---

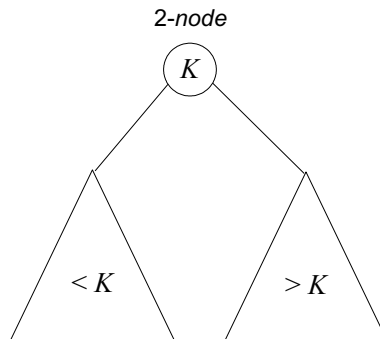
- The simplest implementation of multiway search trees is **2-3 trees**, introduced by the U.S. computer scientist John Hopcroft in 1970.
- A **2-3 tree** is a tree that can have nodes of two kinds:
  - 2-nodes
  - 3-nodes.

# 2-Node

---

- A **2-node** contains a single key  $K$  and has two children:
  - The left child serves as the root of a subtree whose keys are less than  $K$ ,
  - The right child serves as the root of a subtree whose keys are greater than  $K$ .
- A 2-node is the same kind of node we have in the classical binary search tree.

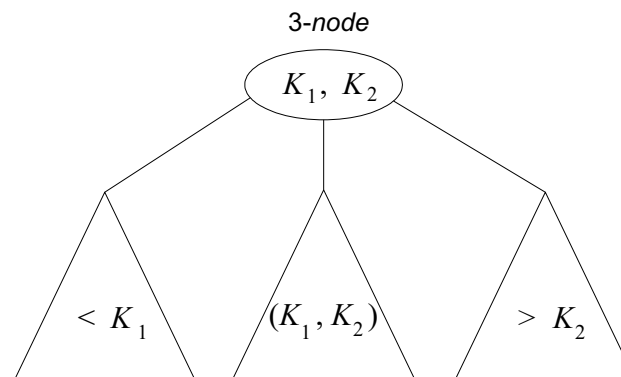
○



# 3-Node

---

- A **3-node** contains
  - Two ordered keys  $K_1$  and  $K_2$  ( $K_1 < K_2$ )
  - Has three children.
- The leftmost child serves as the root of a subtree with keys less than  $K_1$ ,
- The middle child serves as the root of a subtree with keys between  $K_1$  and  $K_2$ ,
- The rightmost child serves as the root of a subtree with keys greater than  $K_2$ .

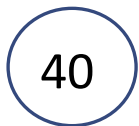


# 2-3 Tree Properties

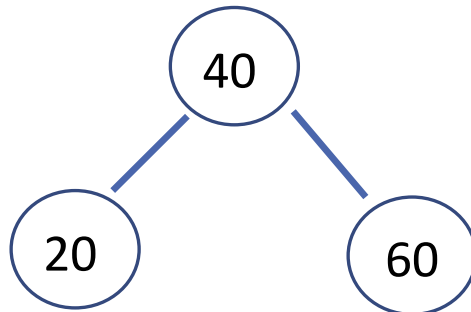
---

1. Each node can either be a leaf node, 2- node, or 3-node
2. Keys is stored in **sorted order**.
3. It is a **balanced tree**.
4. All the leaf nodes are at **same level**.
5. Always **insertion** is **done at leaf**.

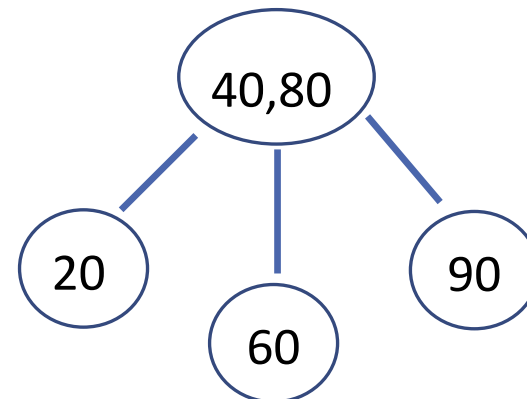
1 Leaf node



1 Key, 2 Children



2 Keys, 3 Children



# 2-3 Tree Searching

---

## Searching:

- We start at the root.
  - If the root is a 2-node, we act as if it were a **binary search** tree:
    - We either stop if  $K$  is equal to the root's key or continue the search in the left or right subtree if  $K$  is, respectively, smaller or larger than the root's key.
  - If the root is a 3- node, we know after no more than two key comparisons whether the search can be stopped (if  $K$  is equal to one of the root's keys) or in which of the root's three subtrees it needs to be continued.

○



# 2-3 Tree Inserting

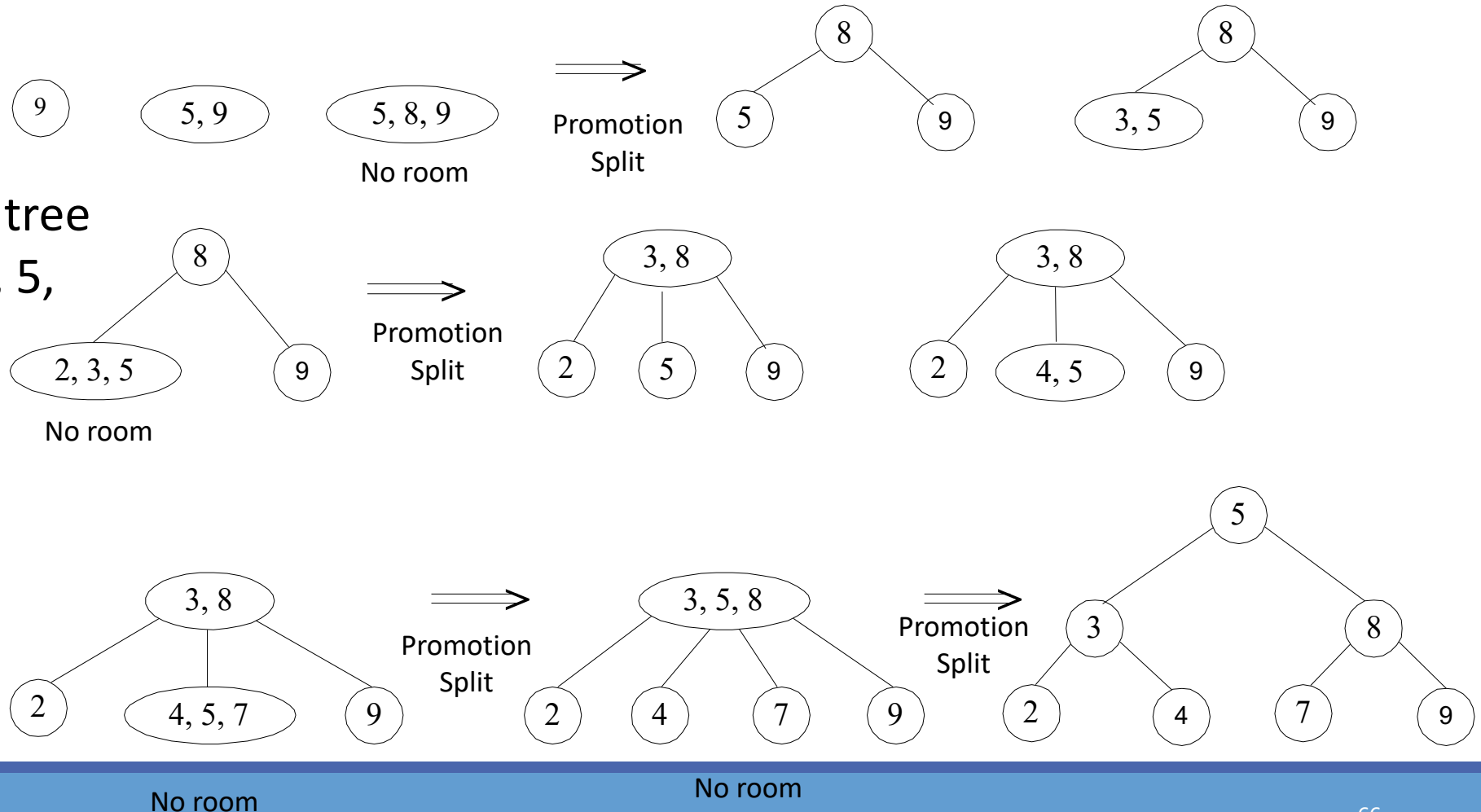
---

## Inserting:

- We always insert a new key K in a leaf, except for the empty tree.
- The appropriate leaf is found by performing a search for K.
  - If the leaf in question is a 2-node, we insert K there as either the first or the second key, depending on whether K is smaller or larger than the node's old key.
  - If the leaf is a 3-node, we split the leaf in two:
    - The smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, and the middle key is promoted to the old leaf's parent.
    - If the leaf happens to be the tree's root, a new root is created to accept the middle key.
    - Note that promotion of a middle key to its parent can cause the parent's overflow (if it was a 3-node) and hence can lead to several node splits along the chain of the leaf's ancestors.

# Example 1: 2-3 tree construction

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

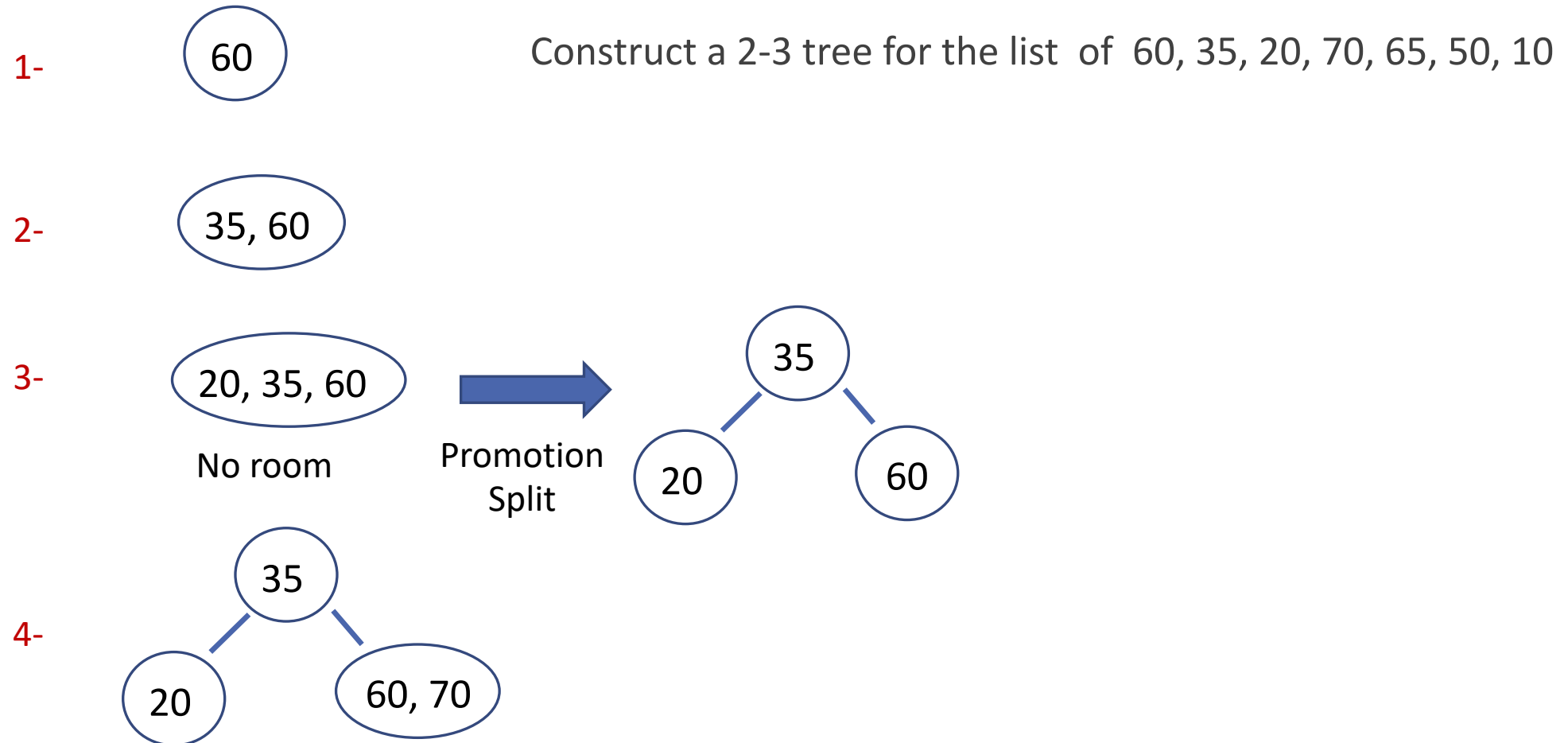


## Example 2: 2-3 tree construction

---

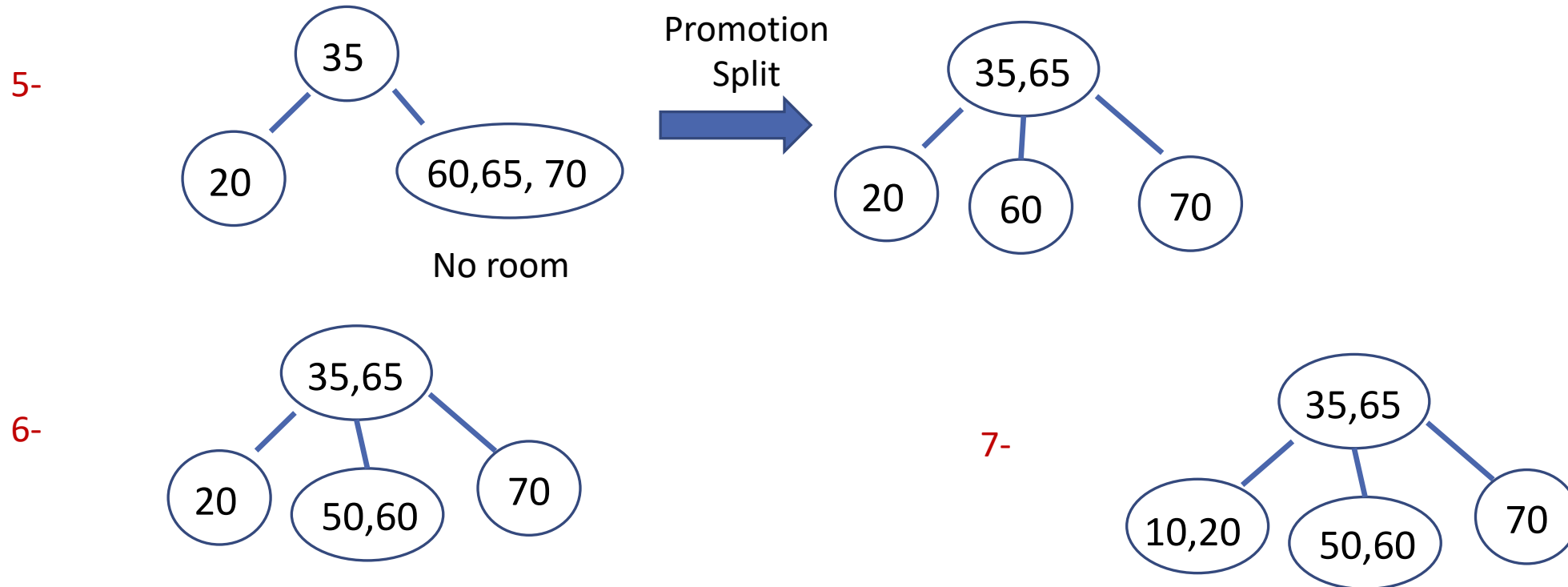
Construct a 2-3 tree for the list of 60, 35, 20, 70, 65, 50, 10

## Example 2: 2-3 tree construction



## Example 2: 2-3 tree construction

Construct a 2-3 tree for the list of 60, 35, 20, 70, 65, 50, 10



## Example 3: 2-3 tree construction

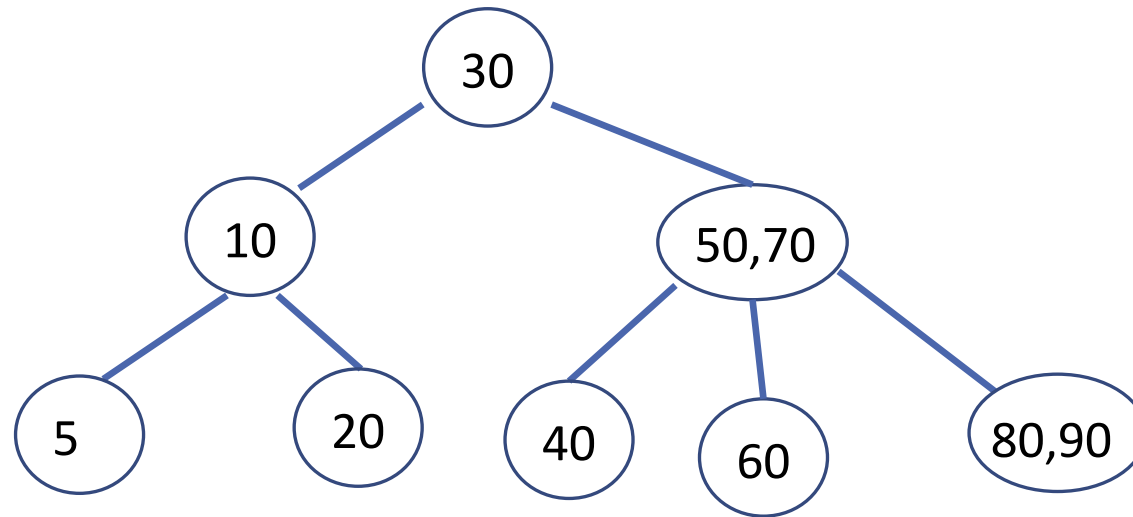
---

Construct a 2-3 tree for the list of 40, 50, 60, 30, 20, 10, 5, 70, 80, 90

## Example 3: 2-3 tree construction

Construct a 2-3 tree for the list of 40, 50, 60, 30, 20, 10, 5, 70, 80, 90

**Solution:**



# Analysis of 2-3 trees

---

- As for any search tree, the efficiency of the dictionary operations depends on the **tree's height**.
- A 2-3 tree of height  $h$  with the smallest number of keys is a full tree of 2-nodes (such as the final tree in the previous figure for  $h = 2$ ).
- Therefore, for any 2-3 tree of height  $h$  with  $n$  nodes, we get the inequality

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

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

- On the other hand, a 2-3 tree of height  $h$  with the largest number of keys is a full tree of 3-nodes, each with two keys and three children.

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

- Therefore, for any 2-3 tree with  $n$  nodes,  $h \geq \log_3(n + 1) - 1.$



# Analysis of 2-3 trees

---

- These lower and upper bounds on height  $h$ ,

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

- The time efficiencies of searching, insertion, and deletion are all in  $\Theta(\log n)$  in both the worst and average case.

# CSC 4520/6520

# Design & Analysis of Algorithms

---

## CHAPTER 6: TRANSFORM-AND-CONQUER

### (HEAPS - PROBLEM REDUCTION)

Abdullah Bal, PhD

# Heaps

---

The data structure called the “heap” is definitely **not a disordered pile** of items as the word’s definition in a standard dictionary might suggest.

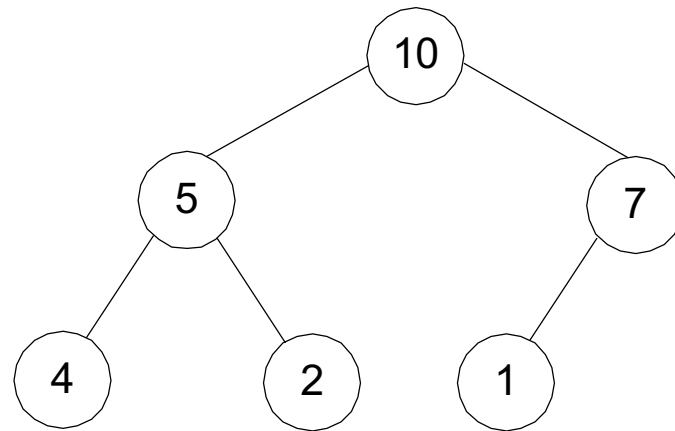
Rather, it is a clever, partially ordered data structure that is especially suitable for **implementing priority queues**.

- It is primarily an efficient implementation of these operations that makes the **heap both interesting and useful**.
- Priority queues arise naturally in such applications as **scheduling job executions** by computer operating systems and **traffic management by communication networks**.
- They also arise in several important algorithms, e.g., **Prim’s algorithm, Dijkstra’s algorithm, Huffman encoding, and branch-and-bound applications**.
- The heap is also the data structure that serves as a cornerstone of a theoretically important sorting algorithm called **heapsort**.

# Heaps

Definition **A heap is a binary tree** with keys at its nodes (one key per node) such that:

1. It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing

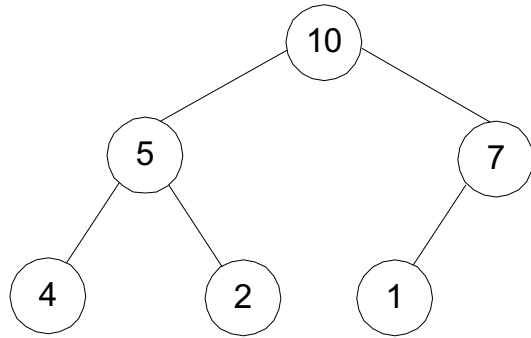


2. Max Heap: The key at each node is  $\geq$  keys at its children

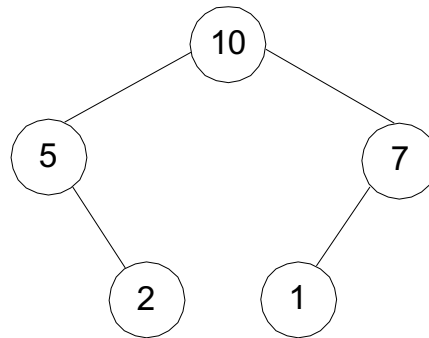
Min Heap: The key at each node is  $\leq$  keys at its children

# Illustration of the heap's definition

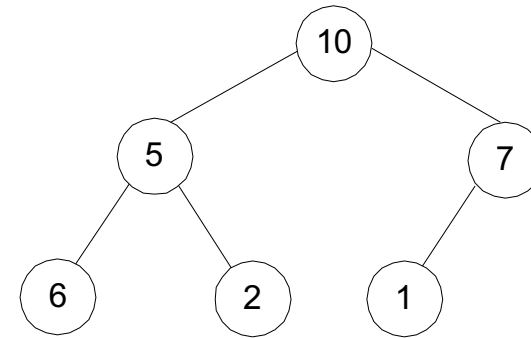
---



a heap



not a heap



not a heap

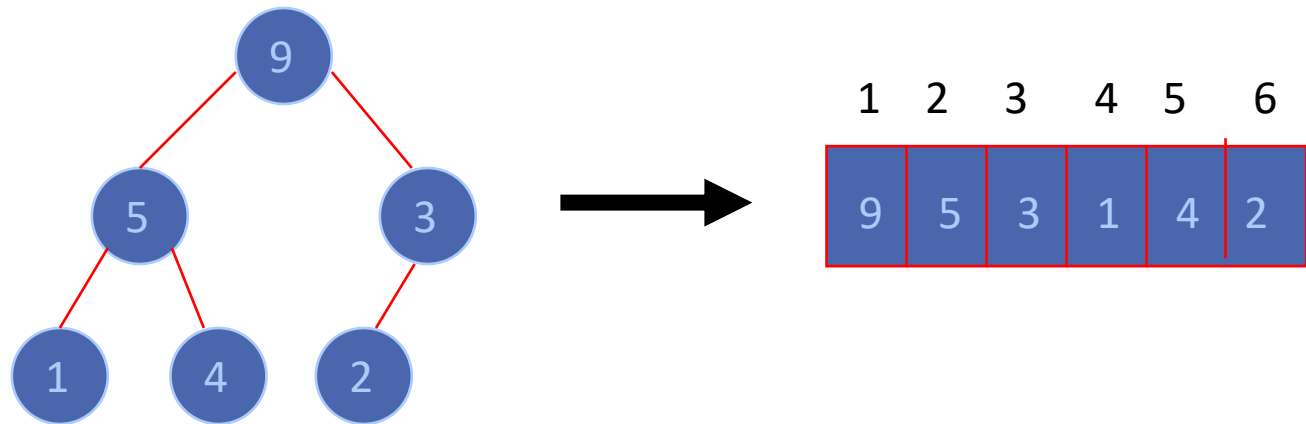
Note: Heap's elements are **ordered top down** (along any path down from its root), but they are **not ordered left to right**

# Heap's Array Representation

- Store heap's elements in an array (whose elements indexed, for convenience, 1 to  $n$ ) in top-down left-to-right order

- Example:

- Left child of node  $j$  is at  $2j$
- Right child of node  $j$  is at  $2j+1$
- Parent of node  $j$  is at floor  $\lfloor j/2 \rfloor$  where  $j$  is an arbitrary index.



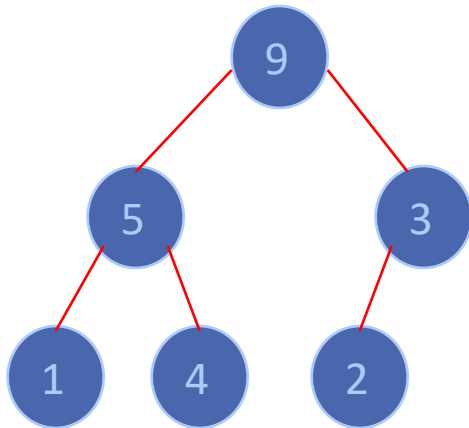
# Some Important Properties of a Heap

---

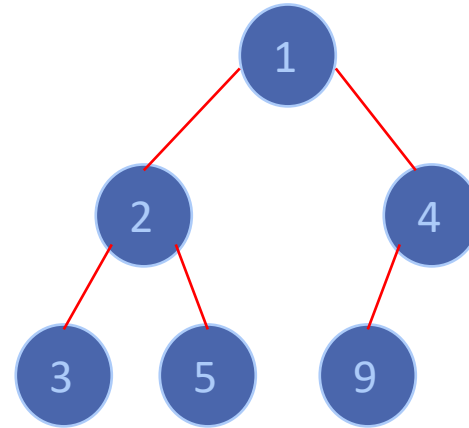
- Given  $n$ , there exists a unique binary tree with  $n$  nodes that is essentially complete, with  $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array

# Max Heap and Min Heap

---



Max Heap

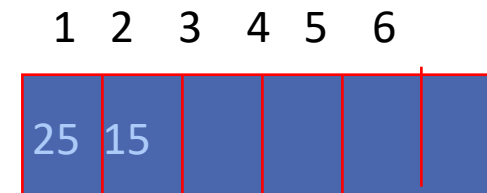
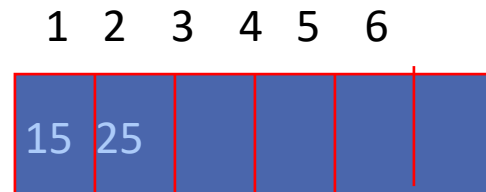
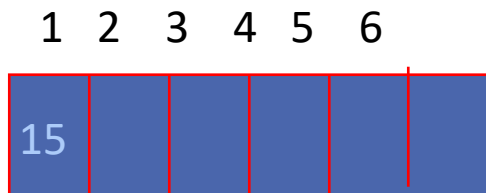
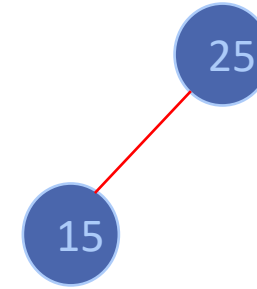
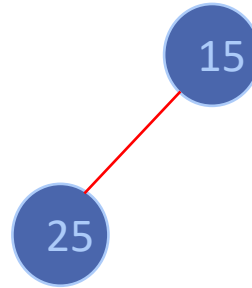


Min Heap



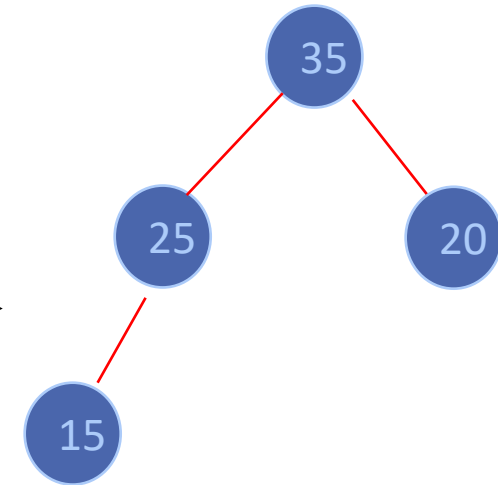
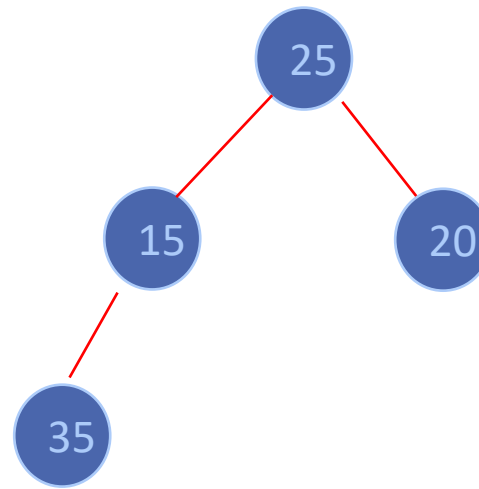
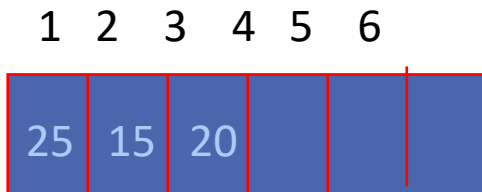
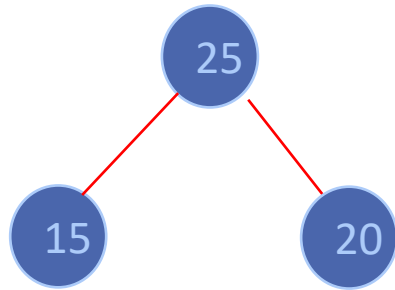
# Example: Heap Creation (Top-Down)

Create a max heap for the list 15, 25, 20, 35, 45, 10



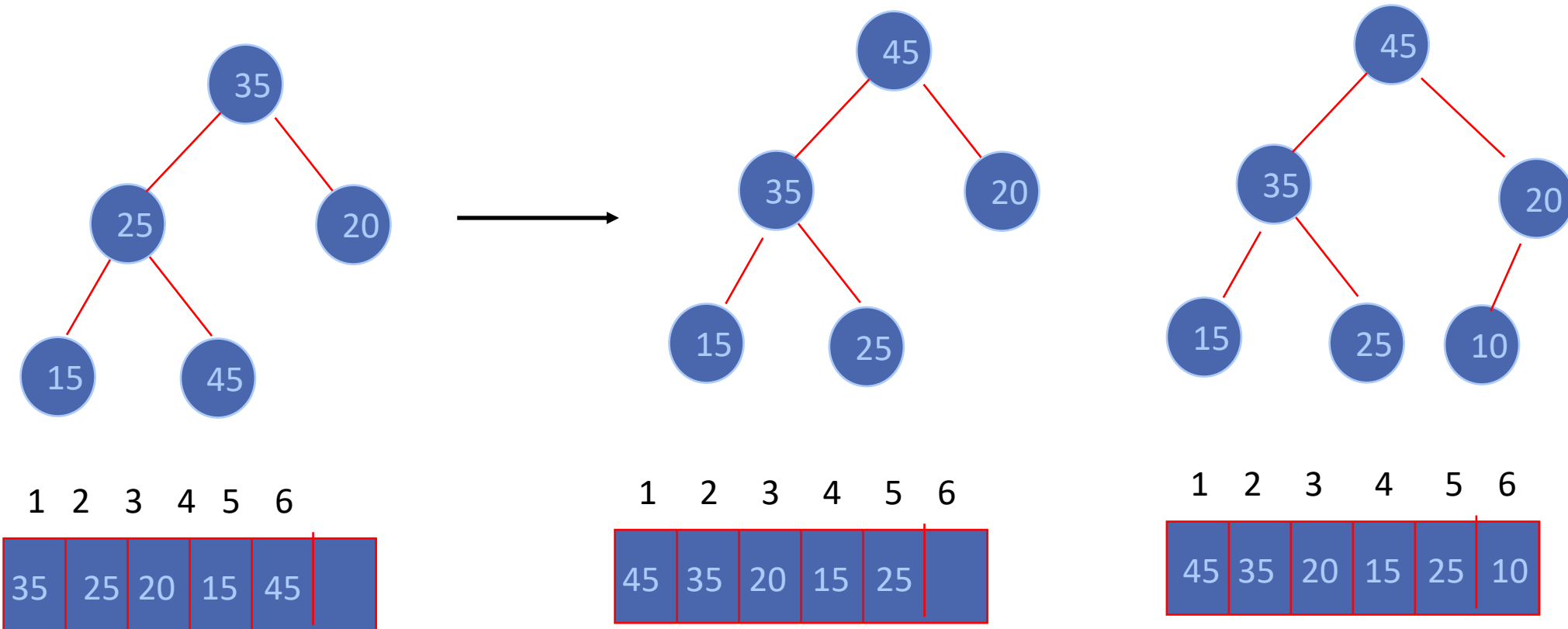
# Example: Heap Creation (Top-Down)

Create a max heap for the list 15, 25, 20, 35, 45, 10



# Example: Heap Creation (Top-Down)

Create a max heap for the list 15, 25, 20, 35, 45, 10



# Heap Construction (Bottom-up)

---

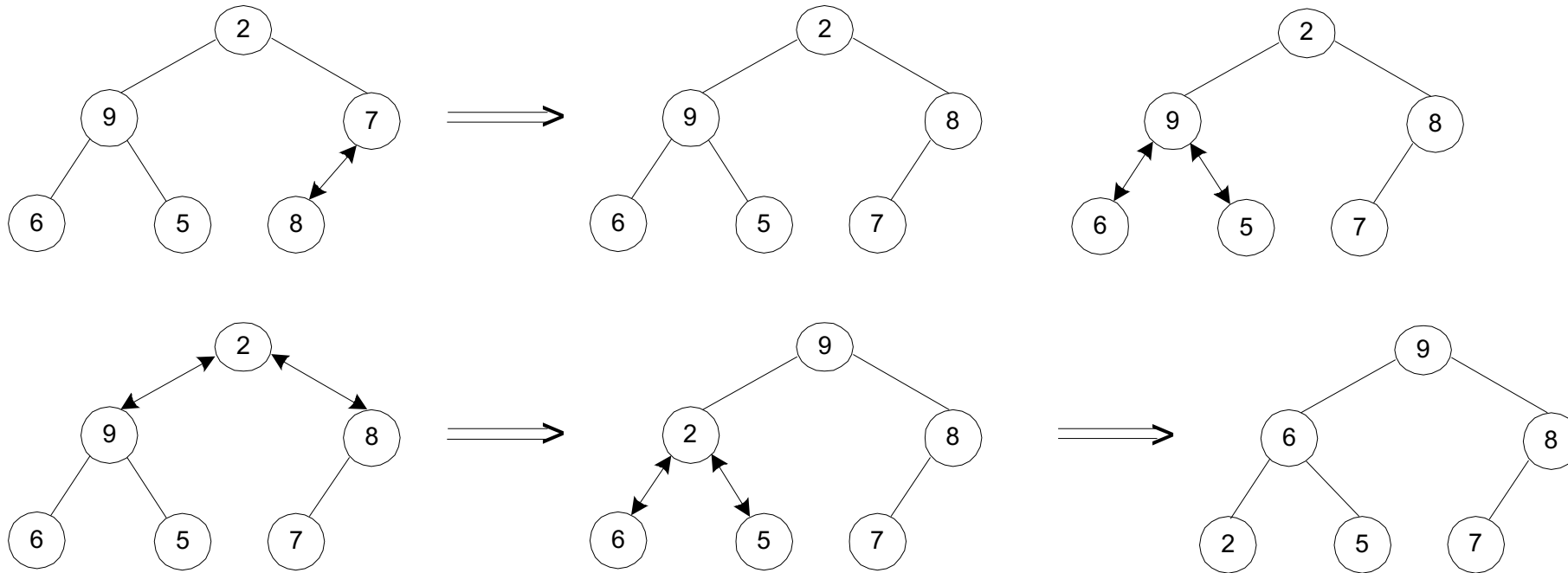
**Step 0:** Initialize the structure with keys in the order given

**Step 1:** Starting with the last (rightmost) parental node, fix the heap rooted at it,  
if it doesn't satisfy the heap condition: keep exchanging it with its largest child until  
the heap condition holds

**Step 2:** Repeat Step 1 for the preceding parental node

# Example: Heap Construction (Bottom-up)

Construct a max heap for the list 2, 9, 7, 6, 5, 8



# Deletion

---

- Start to delete by max element of the heap which is the root's key in Max. 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.

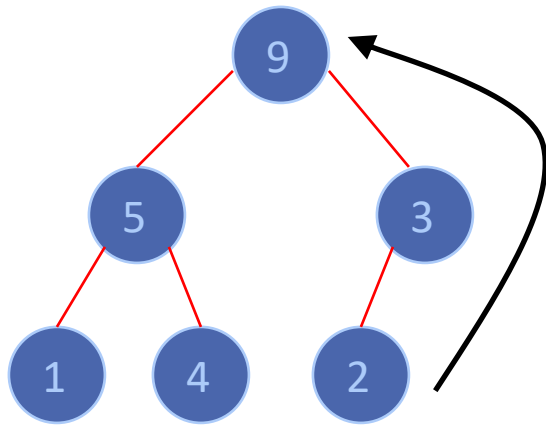
**Step 3:** “Heapify” the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm.

# Deletion

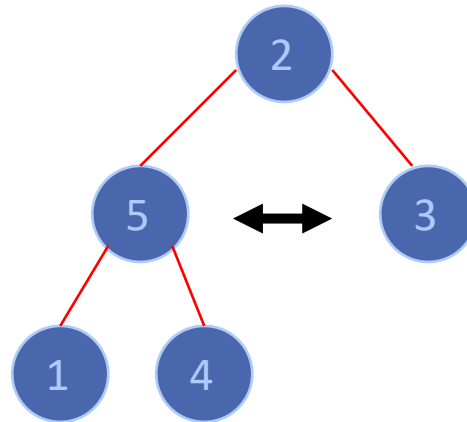
---

- 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.
- The efficiency of deletion is determined by the number of key comparisons needed to “heapify” the tree after the swap has been made and the size of the tree is decreased by 1.
- Since this cannot require more key comparisons than twice the heap’s height, the time efficiency of deletion is in  $O(\log n)$  as well.

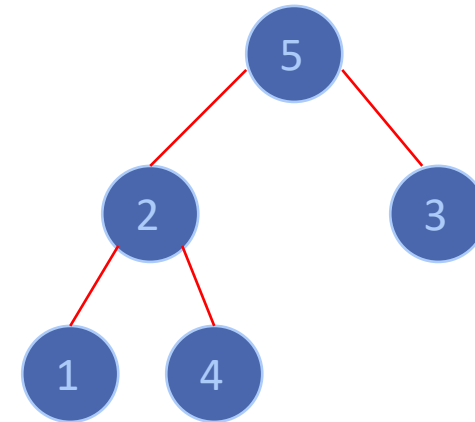
# Deletion



1	2	3	4	5	6
9	5	3	1	4	2



1	2	3	4	5	6
2	5	3	1	4	

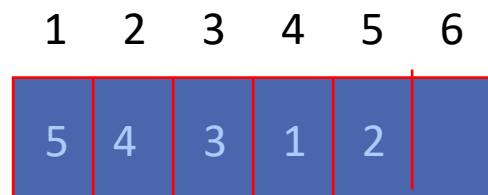
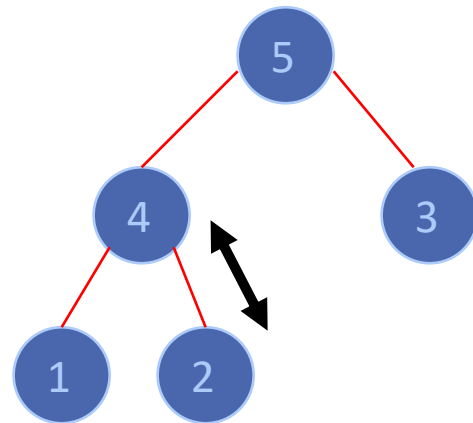


1	2	3	4	5	6
5	2	3	1	4	



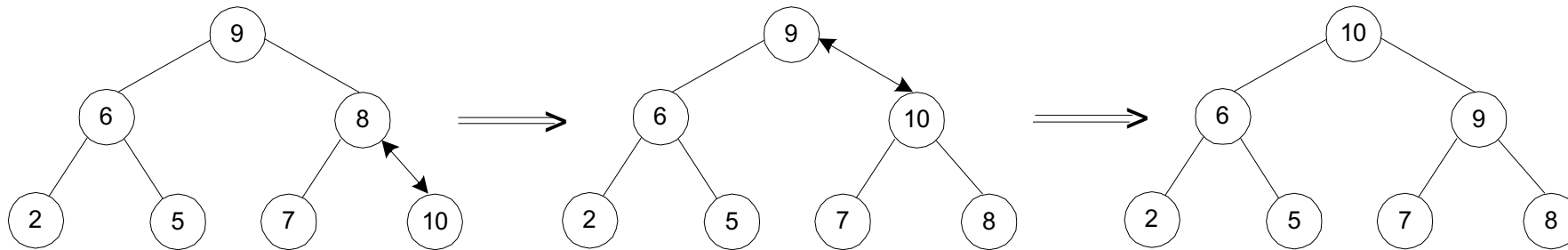
# Deletion

---



# Insertion of a New Element into a Heap

- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied
- Example: Insert key 10



- Efficiency:  $O(\log n)$

---

# HEAPSORT

# Heapsort

---

**Stage 1:** Construct a heap for a given list of  $n$  keys

**Stage 2:** Repeat operation of root removal  $n-1$  times:

- Exchange keys in the root and in the last (rightmost) leaf
- Decrease heap size by 1
- If necessary, swap new root with larger child until the heap condition holds

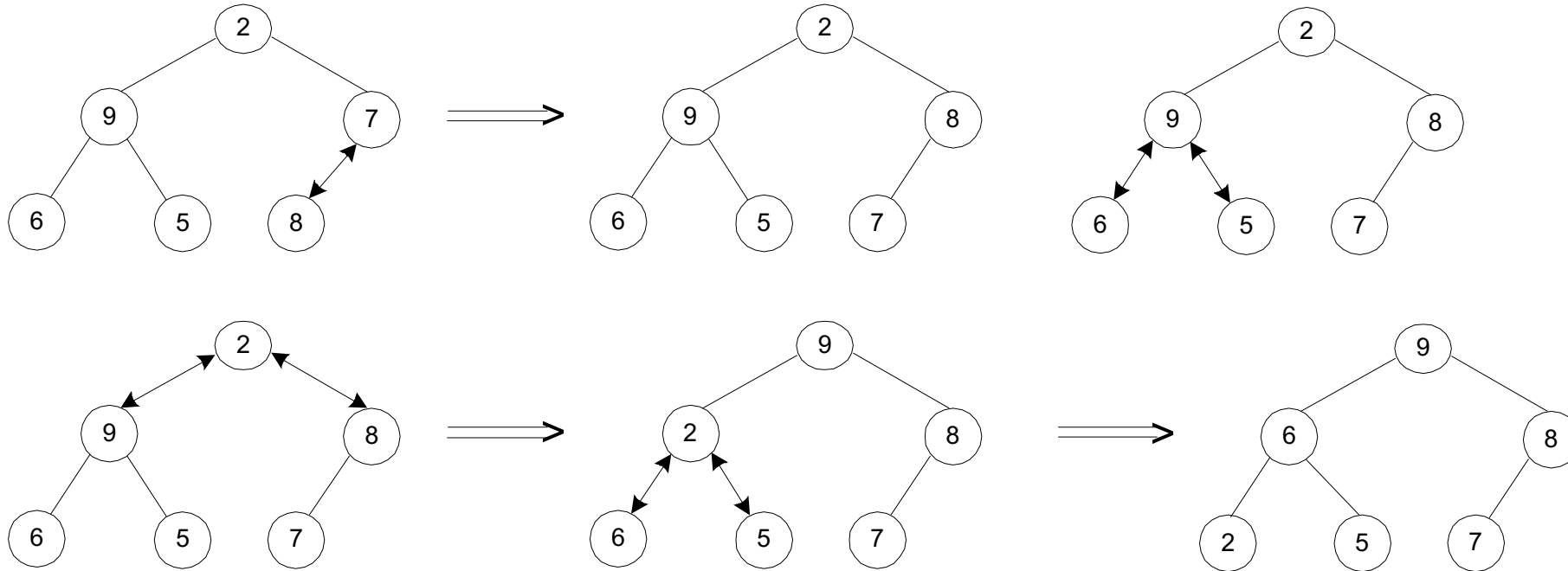
# Example: Sorting by Heapsort

---

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

# Stage 1: Heap Construction (Bottom-up)

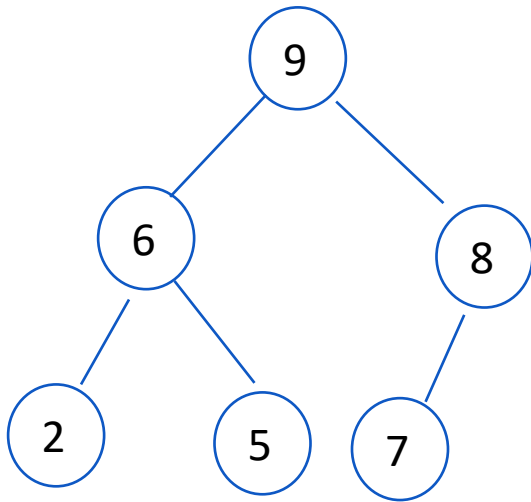
Heap construction for 2, 9, 7, 6, 5, 8



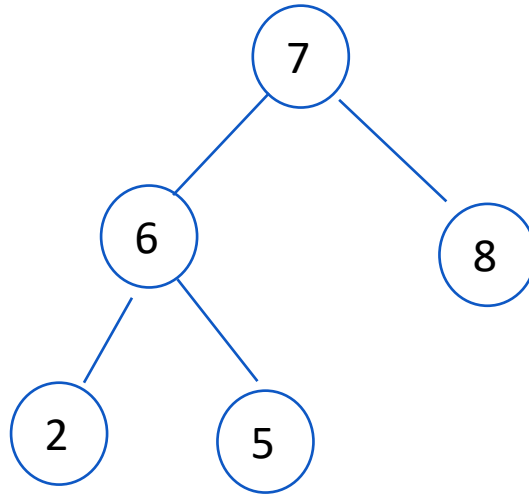
Constructed Heap: 9 6 8 2 5 7

## Stage 2: Root/Max Removal

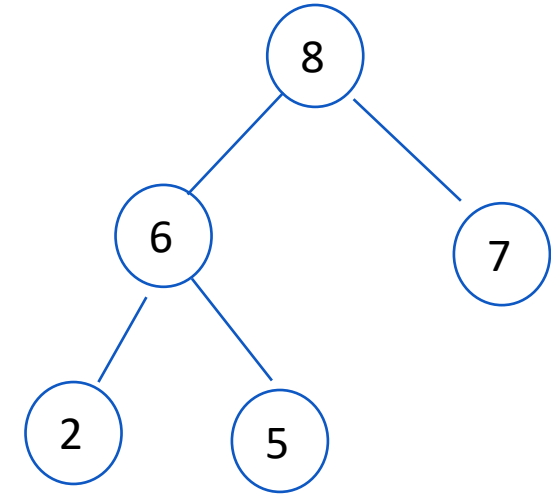
---



9 6 8 2 5 7



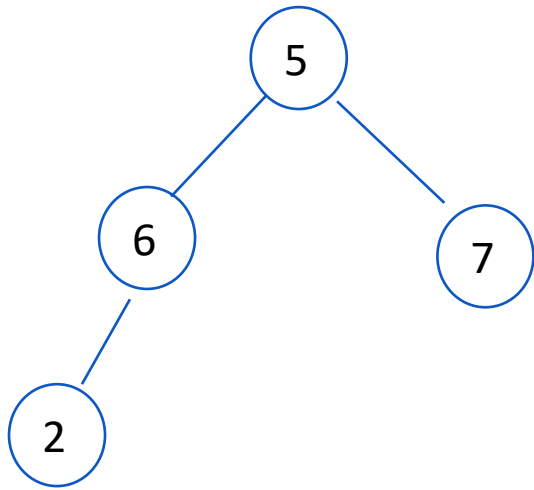
7 6 8 2 5 | 9



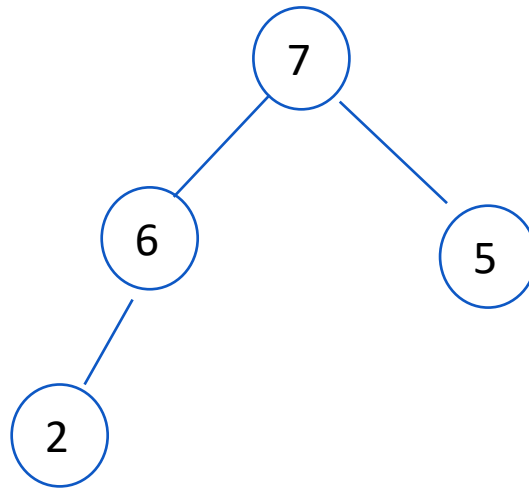
8 6 7 2 5 | 9

## Stage 2: Root/Max Removal

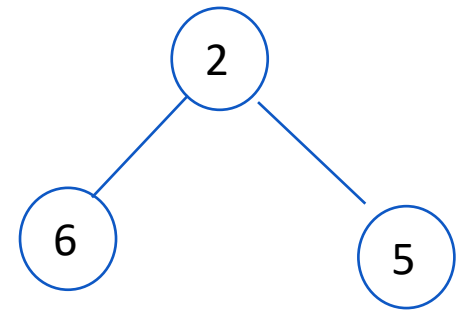
---



5 6 7 2 | 8 9



7 6 5 2 | 8 9

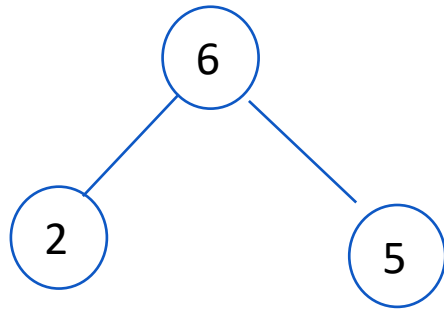


2 6 5 | 7 8 9

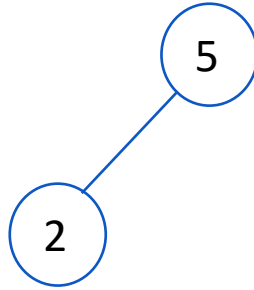


## Stage 2: Root/Max Removal

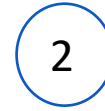
---



6 2 5 | 7 8 9



2 5 | 6 7 8 9



2 | 5 6 7 8 9

Sorted Array: 2 5 6 7 8 9

# Analysis of Heapsort

---

**Stage 1:** Build heap for a given list of  $n$  keys

worst-case

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

# Analysis of Heapsort

---

**Stage 2:** Repeat operation of root removal  $n-1$  times (fix heap).

- We have to investigate just the time efficiency of the second stage.
- For the number of key comparisons,  $C(n)$ , needed for eliminating the root keys from the heaps of diminishing sizes from  $n$  to  $2$ , we get the following inequality:

$$C(n) \leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \leq 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\leq 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \leq 2n \log_2 n.$$

$$C(n) \in O(n \log n)$$

# Analysis of Heapsort

---

- For both stages, we get  $O(n) + O(n \log n) = O(n \log n)$ .
- A more detailed analysis shows that the time efficiency of heapsort is, in fact, in  $\Theta(n \log n)$  in both the worst and average cases.
- Thus, heapsort's time efficiency falls in the same class as that of **merge sort**.
- Unlike the latter, heapsort is in-place, i.e., it **does not require any extra storage**.
- Timing experiments on random files show that heapsort runs **more slowly than quick sort** but can be competitive with **merge sort**.

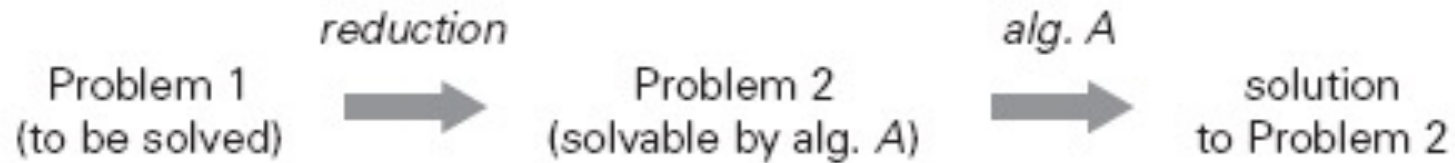
---

# Problem Reduction

# Problem Reduction

---

- This variation of transform-and-conquer solves a problem by transforming it into **different problem** for which an algorithm is **already available**.



- To be of practical value, **the combined time of the transformation and solving the other problem** should be **smaller** than solving the problem as given by another method.

# Examples of Solving Problems by Reduction

---

- Computing  $\text{lcm}(m, n)$  (the least common multiple) via computing  $\text{gcd}(m, n)$  (the greatest common divisor)
- Transforming a maximization problem to a minimization problem and vice versa (also, min-heap construction)
- Counting number of paths of length  $n$  in a graph by raising the graph's adjacency matrix to the  $n$ -th power
- Linear programming
- Reduction to graph problems (e.g., solving puzzles via state-space graphs)

# Computing The Least Common Multiple

- The **least common multiple** of two positive integers  $m$  and  $n$ , denoted  $\text{lcm}(m, n)$ , is defined as the **smallest integer that is divisible** by both  $m$  and  $n$ .

- For example,

$$\text{lcm}(24, 60) = 120,$$

$$\text{lcm}(11, 5) = 55.$$

- Given the **prime factorizations** of  $m$  and  $n$ , compute the **product** of all the common prime factors of  $m$  and  $n$ , all the prime factors of  $m$  that are not in  $n$ , and all the prime factors of  $n$  that are not in  $m$ .

- For example,

$$24 = 2 \cdot 2 \cdot 2 \cdot 3,$$

$$60 = 2 \cdot 2 \cdot 3 \cdot 5,$$

$$\text{lcm}(24, 60) = (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 = 120.$$

- As a computational procedure, this algorithm has the **same drawbacks** as the middle-school algorithm for computing the **greatest common divisor**.
- It is **inefficient** and requires a **list of consecutive primes**.



# Computing The Least Common Multiple (Cont.)

- A much more efficient algorithm for computing the least common multiple can be devised by using problem reduction.
- There is a very efficient algorithm (Euclid's algorithm) for finding the greatest common divisor, which is a product of all the common prime factors of  $m$  and  $n$ .
- Can we find a formula relating  $\text{lcm}(m, n)$  and  $\text{gcd}(m, n)$ ?
- It is not difficult to see that the product of  $\text{lcm}(m, n)$  and  $\text{gcd}(m, n)$  includes every factor of  $m$  and  $n$  exactly once and hence is simply equal to the product of  $m$  and  $n$ .

- $$m \cdot n = \text{lcm}(m, n) \cdot \text{gcd}(m, n)$$

- This observation leads to the formula

$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)},$$

where  $\text{gcd}(m, n)$  can be computed very efficiently by Euclid's algorithm.

# Reduction of Optimization Problems

---

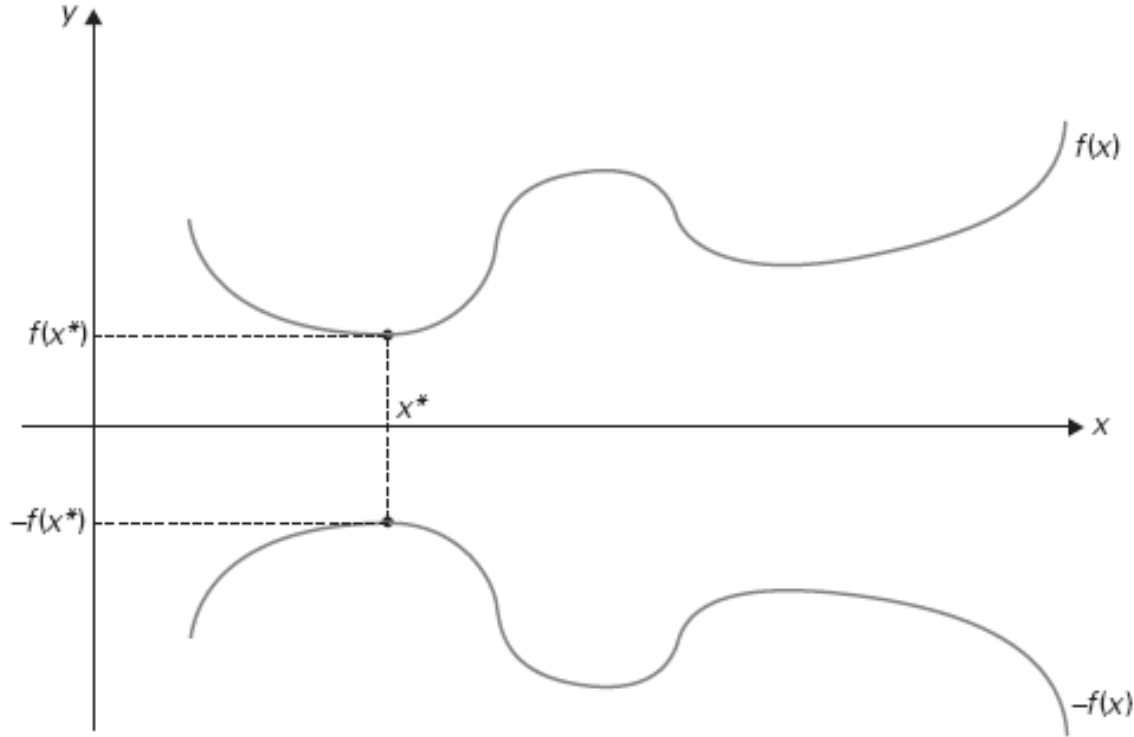
- If a problem asks to find a maximum of some function, it is said to be a ***maximization problem***;
- If it asks to find a function's minimum, it is called a ***minimization problem***.
- Suppose now that we need to find a minimum of some function  $f(x)$  and we have an algorithm for function maximization.
- How can you take advantage of the latter?
- The answer lies in the simple formula :

$$\min f(x) = - \max[-f(x)].$$

- In other words, to minimize a function, we can maximize its negative instead and, to get a correct minimal value of the function itself, change the sign of the answer.

# Reduction of Optimization Problems (Cont.)

- Relationship between minimization and maximization problems:



# CSC 4520/6520

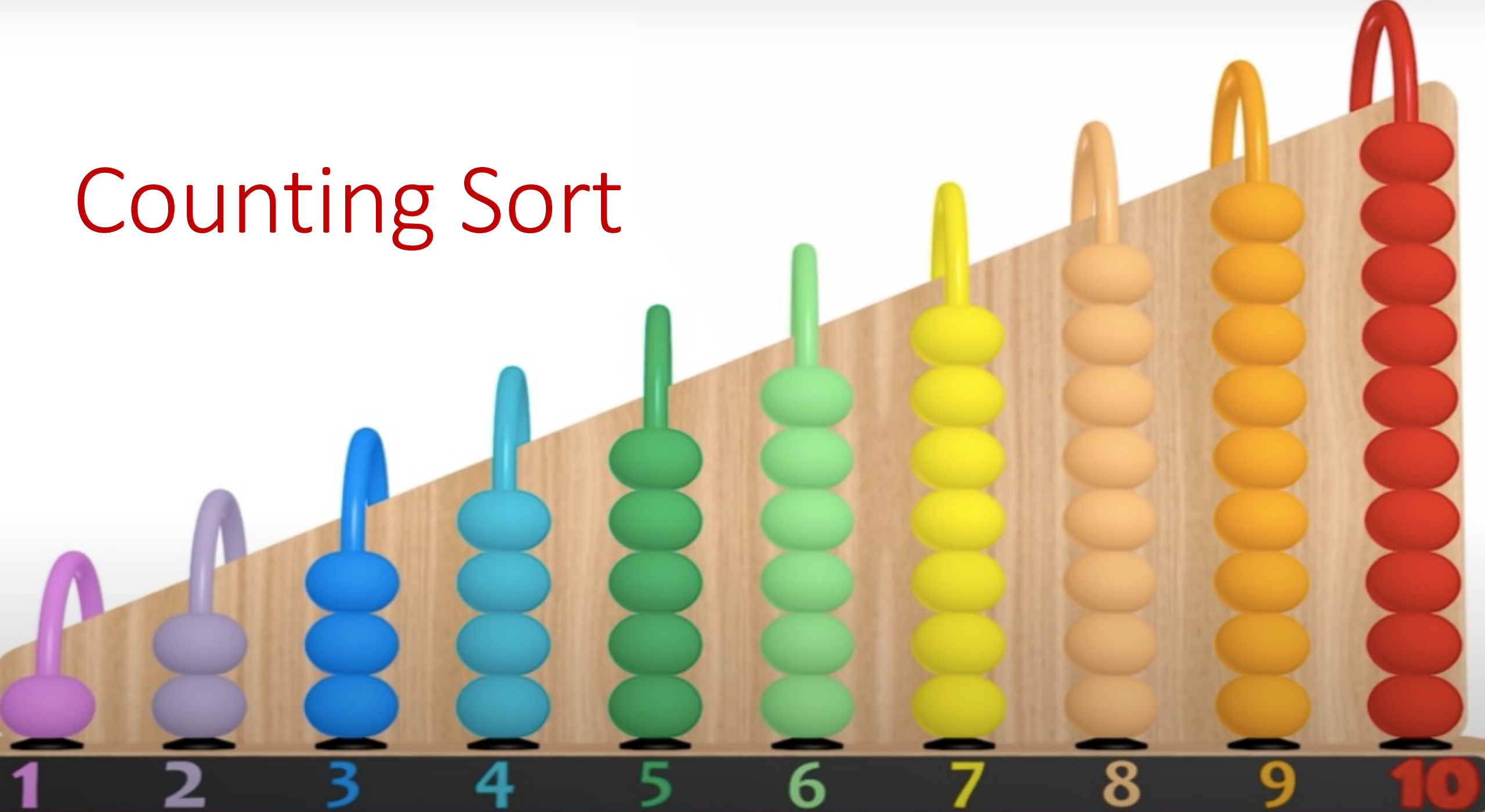
# Design & Analysis of Algorithms

---

## CHAPTER 7: SPACE AND TIME TRADE-OFFS

Abdullah Bal, PhD

# Counting Sort



# Sorting by Counting

---

- Counting sort is an example of **input enhancement** algorithm design techniques.
- Assume, for example, that we have to sort a list whose values can be either 1 or 2.
- We can scan the list to **compute the number of 1's** and **the number of 2's in it** and then, on the second pass, simply make the appropriate number of the first elements equal to 1 and the remaining elements equal to 2.
- More generally, if element values are integers between some lower bound  $l$  and upper bound  $u$ , we can **compute the frequency** of each of those values and **store them** in array  $F [0..u - l]$ .
- Then the first  $F [0]$  positions in the sorted list must be filled with  $l$ , the next  $F [1]$  positions with  $l + 1$ , and so on.
- All this can be done, of course, only if we can overwrite the given elements.

# Example 1: Sorting by Counting (First Alg.)

- Consider sorting the array

13	11	12	13	12	12
----	----	----	----	----	----

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting.

- The frequency and distribution arrays are as follows:

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

- Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array.
- If we index array positions from 0 to  $n - 1$ , the distribution values must be reduced by 1 to get corresponding element positions.

## Example 1: Sorting by Counting (First Alg.)

- It is more convenient to process the input array right to left. 

13	11	12	13	12	12
----	----	----	----	----	----
- For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position  $4-1 = 3$  of the array S that will hold the sorted list.
- Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array.

Iteration	Sort List					
1				12		
2			12	12		
3			12	12		13
4		12	12	12		13
5	11	12	12	12		13
6	11	12	12	12	13	13



## Example 1: Sorting by Counting (Second Alg.)

- Consider sorting the array

13	11	12	13	12	12
----	----	----	----	----	----

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting.

- The frequency array is as follows:

Array values	11	12	13
Frequencies	1	3	2

- The sorted array according to frequency list as follows:

11	12	12	12	13	13
----	----	----	----	----	----

## Example 2: Sorting by Counting

---

Sort the following array using counting sort:

1	0	2	5	3	7	3	4	0	6
---	---	---	---	---	---	---	---	---	---

# Analysis of Counting Sort

---

- Assuming that the range of array values is fixed, this is obviously a **linear algorithm** because it makes just **two consecutive passes** through its input array  $A$   $\Theta(n+k)$
- This is a **better time-efficiency class** than that of the most efficient sorting algorithms—**mergesort, quicksort, and heapsort**—we have encountered.
- It is important to remember, however, that this efficiency is obtained by exploiting the specific nature of inputs for which sorting by distribution counting works, in addition **to trading space for time**.



# String Searching

# Review: String searching by brute force

---

*pattern*: a string of  $m$  characters to search for

*text*: a (long) string of  $n$  characters to search in

## **Brute force algorithm**

Step 1    Align pattern at beginning of text

Step 2    Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step 3    While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

The time complexity of brute force is  $O(n*m)$

# Horspool's Algorithm

---

- Preprocesses pattern to **generate a shift table** that determines how much to shift the pattern when a **mismatch occurs**.
- Always makes a shift based on **the text's character  $c$**  aligned with the last character in the pattern according to the shift table's entry for  $c$ .

# Horspool's Algorithm

---

- If a mismatch occurs, we need to shift the pattern to the right.
- Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text.
- Horspool's algorithm determines the size of such a shift by looking at the character  $c$  of the text that is aligned against the last character of the pattern.
- This is the case even if character  $c$  itself matches its counterpart in the pattern.

# Horspool's Algorithm

---

- These examples clearly demonstrate that right-to-left character comparisons can lead to **farther shifts of the pattern than the shifts by only one position** always made by the brute-force algorithm.
- However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority.
- Fortunately, the idea of input enhancement makes **repetitive comparisons unnecessary**.
- We can **precompute** shift sizes and store them in a table.



# Shift Table

---

- For example, for the pattern BARBER, “\*\*” will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

## Shift Table

Character c	B	A	R	E	**
Shift t(c)	2	4	3	1	6

- Scan the pattern right to left starting second endmost character. Compute the distance from c to end of the pattern.

# Horspool's Algorithm

---

**Step 1 :** For a given pattern of length  $m$  and the alphabet used in both the pattern and text, construct the shift table as described above.

**Step 2 :** Align the pattern against the beginning of the text.

**Step 3 :** Repeat the following until either a matching substring is found, or the pattern reaches beyond the last character of the text.

- Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all  $m$  characters are matched (then stop) or a mismatching pair is encountered.
- In the latter case, retrieve the entry  $t(c)$  from the  $c$ 's column of the shift table where  $c$  is the text's character currently aligned against the last character of the pattern, and shift the pattern by  $t(c)$  characters to the right along the text.

# Example 1: String Matching

- Consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores).
- The shift table, as we mentioned, is filled as follows:

Character c	B	A	R	E	**
Shift t(c)	2	4	3	1	6

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
      B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```

## Example 2: String Matching

---

Text: DESIGN & ANALYSIS OF ALGORITHMS

Pattern: ALGO

# Example 2: String Matching

Text: DESIGN & ANALYSIS OF ALGORITHMS

Pattern: ALGO

Character c	A	L	G	**
Shift t(c)	3	2	1	4

D	E	S	I	G	N		&		A	N	A	L	Y	S	I	S		O	F		A	L	G	O	R	I	T	H	M	S
A	L	G	O																											
				A	L	G	O																							
								A	L	G	O																			
											A	L	G	O																
														A	L	G	O													
																	A	L	G	O										
																		A	L	G	O									
																				A	L	G	O							

## Example 3: String Matching

---

Text: GEORGIA STATE UNIVERSITY COMPUTER SCIENCE

Pattern: SCIENCE

# Example 3: String Matching

Text: GEORGIA STATE UNIVERSITY COMPUTER SCIENCE

Pattern: SCIENCE

Character c	S	C	I	E	N	**
Shift t(c)	6	1	4	3	2	7

G	E	O	R	G	I	A		S	T	A	T	E		U	N	I	V	E	R	S	I	T	Y		C	O	M	P	U	T	E	R		S	C	I	E	N	C	E	
S	C	I	E	N	C	E																																			
							S	C	I	E	N	C	E																												
														S	C	I	E	N	C	E																					
																				S	C	I	E	N	C	E															
																											S	C	I	E	N	C	E								
																													S	C	I	E	N	C	E						
																																			S	C	I	E	N	C	E

## Example 4: String Matching

---

Text: WELCOME TO ART COLLEGE

Pattern: COLL



# Example 4: String Matching

Text: WELCOME TO ART COLLEGE

Pattern: COLL

Character c				C	O				L				**								
Shift t(c)				3	2				1				4								
W	E	L	C	O	M	E		T	O		A	R	T		C	O	L	L	E	G	E
C	O	L	L																		
			C	O	L	L															
							C	O	L	L											
											C	O	L	L							
															C	O	L	L			

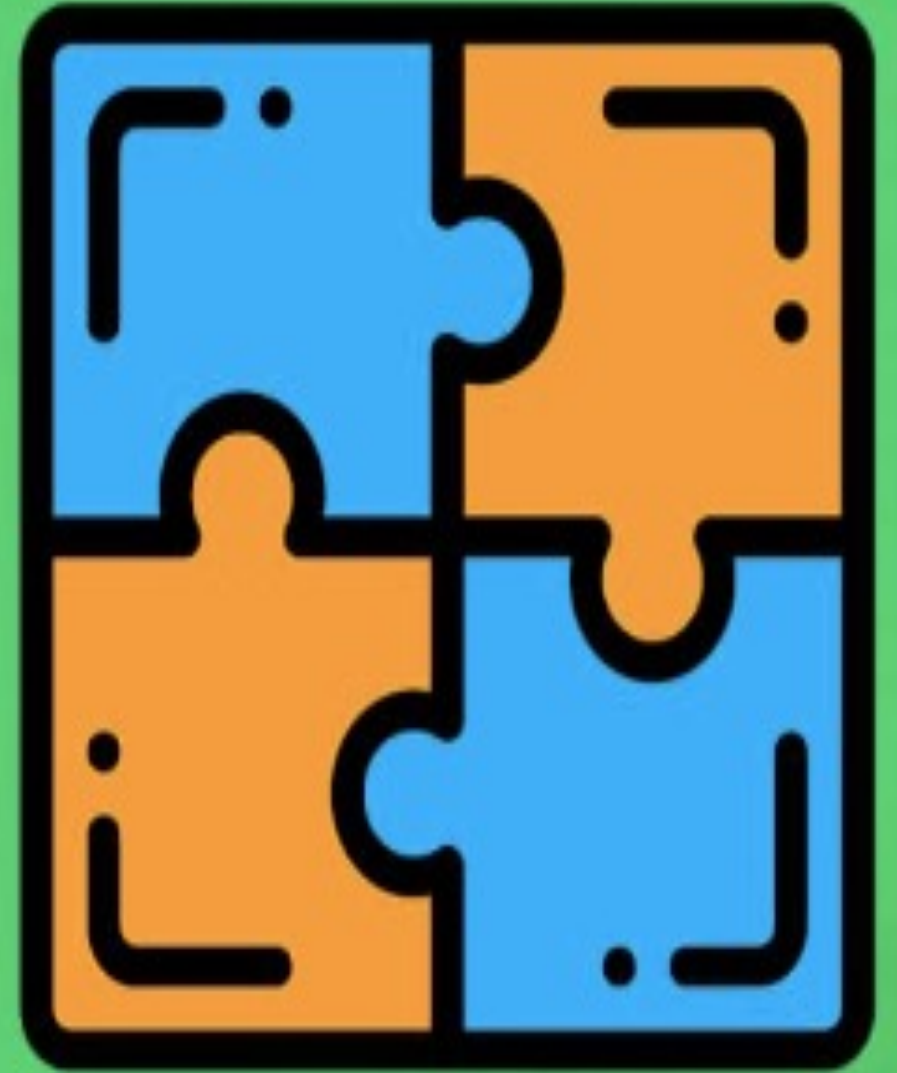
# Analysis of Horspool's Algorithm

---

- A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in  $O(nm)$ .
- But for random texts, it is in  $\Theta(n)$ , and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm.
- In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.



Hashing

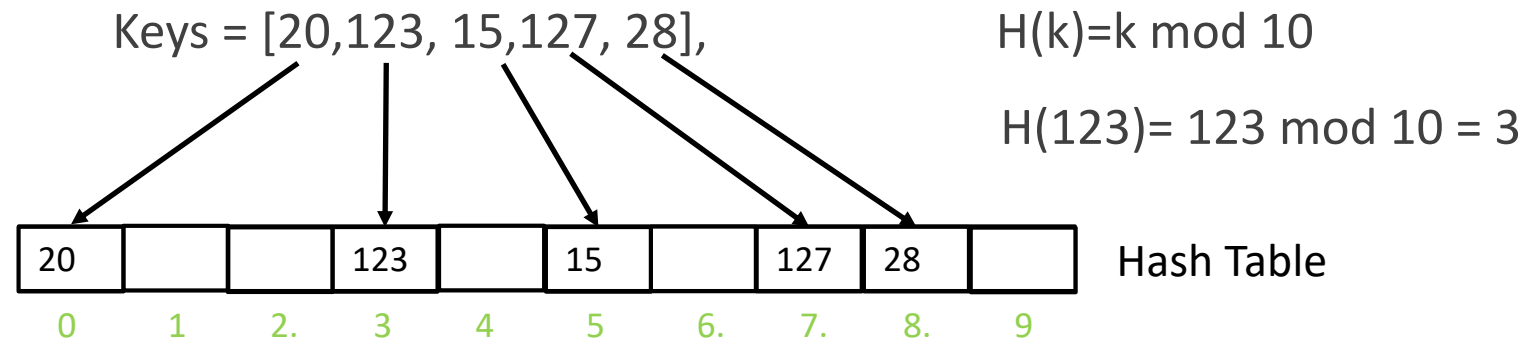
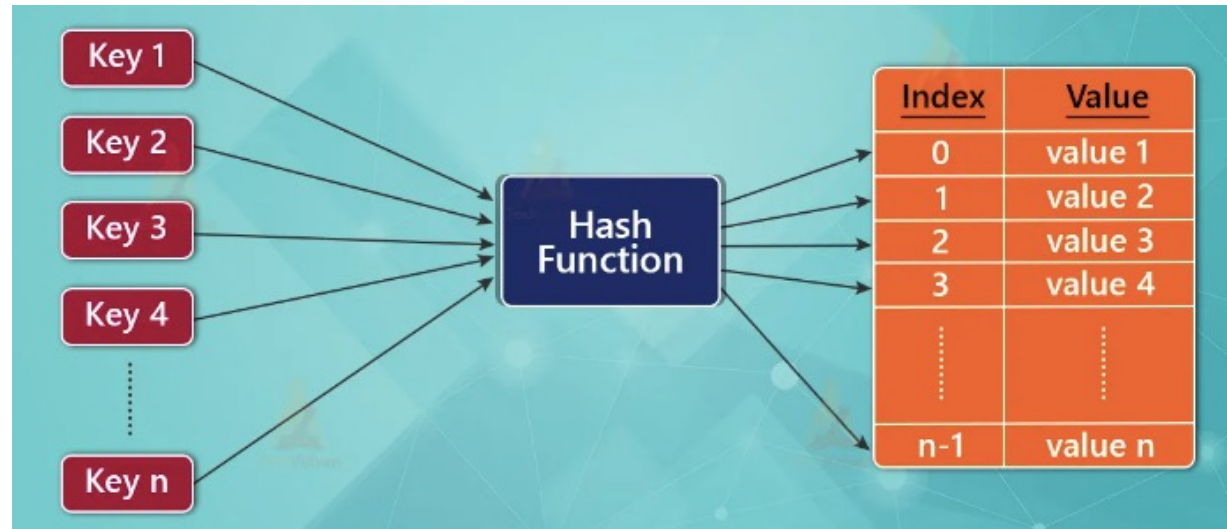


# Hashing

---

- The idea of hashing is to map keys of a given file of size  $n$  into a table of size  $m$ , called the hash table, by using a predefined function, called the hash function.
- It's discovered in the 1950s by IBM researchers.
- A very efficient method for implementing a *dictionary*, (student records in a school, citizen records in a governmental office, book records in a library )

# Hash tables and hash functions



# Hash functions

- A hash function can be a form of key

$$h(K) = K \bmod m$$

- If keys are **letters** of some alphabet, we can first assign a letter its position in the alphabet, denoted **ord(K)**, and then apply the same kind of a function used for integers.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

- Finally, if  $K$  is a character string  $c_0 c_1 \dots c_{s-1}$ , we can use, as a very unsophisticated option,

$$\sum_{i=0}^{s-1} \text{ord}(c_i) \bmod m$$

$$h(\text{ARE}) = (1 + 18 + 5) \bmod 13 = 11$$

$$h(\text{SOON}) = (19 + 15 + 15 + 14) \bmod 13 = 11.$$

# Hash functions

---

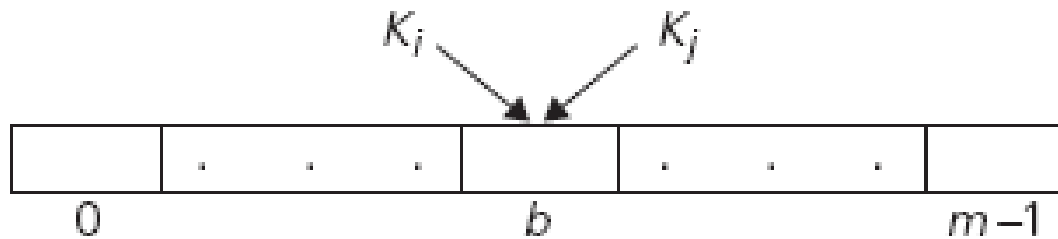
- In general, a hash function needs to satisfy somewhat conflicting requirements:
  - A hash table's size should **not be excessively large** compared to the **number of keys**, but it should be sufficient to not jeopardize the implementation's time efficiency.
  - A hash function needs to **distribute keys** among the cells of the hash table **as evenly as possible**.
  - A hash function has to be **easy to compute**.
- Example: student records, key = SSN.
- Hash function:

$h(K) = K \bmod m$  where  $m$  is some integer (typically, prime)

If  $m = 1000$ , where is record with SSN= 314159265 stored?

# Collisions

- If we choose a hash table's size  $m$  to be smaller than the number of keys  $n$ , we will get **collisions**—a phenomenon of two (or more) keys being hashed into the same cell of the hash table:  $h(K_i) = h(K_j)$ .

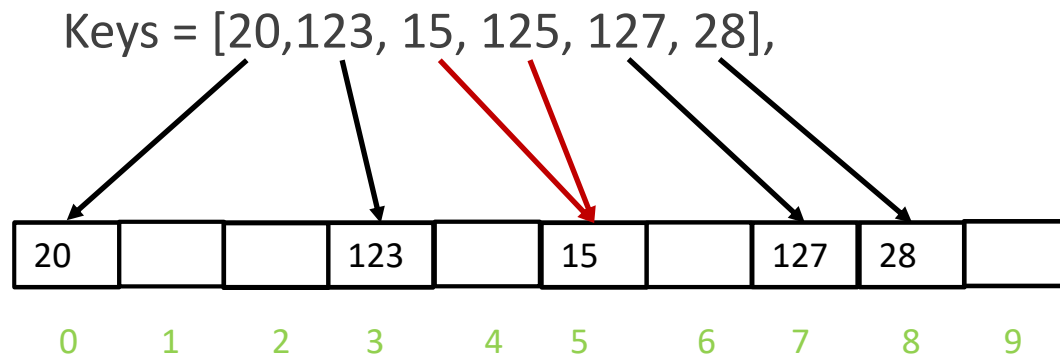


- However, collisions should be expected even if  $m$  is considerably larger than  $n$ .
- In fact, in **the worst case**, all the keys could be hashed to the same cell of the hash table.
- Fortunately, with an appropriately chosen hash table size and a good hash function, this situation **happens very rarely**.



# Collisions

---



$$H(k) = k \bmod 10$$

$$H(125) = 125 \bmod 10 = 5$$

Hash Table

# Collisions

---

- Every hashing scheme must have a **collision resolution** mechanism.
- This mechanism is different in the two principal versions of hashing:
  - ***open hashing*** (also called ***separate chaining***)
    - each cell is a header of linked list of all keys hashed to it
  - ***closed hashing*** (also called ***open addressing***).
    - one key per cell
    - in case of collision, finds another cell by
      - *linear probing*: use next free bucket
      - *double hashing*: use second hash function to compute increment

# Open hashing (Separate chaining)

---

- Keys are stored in **linked lists** attached to cells of a hash table.
- Each list contains all the keys hashed to its cell.
- Consider, as an example, the following list of words:

A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

$h(K)$  = sum of  $K$  's letters' positions in the alphabet MOD 13

# Open hashing (Separate chaining)

A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

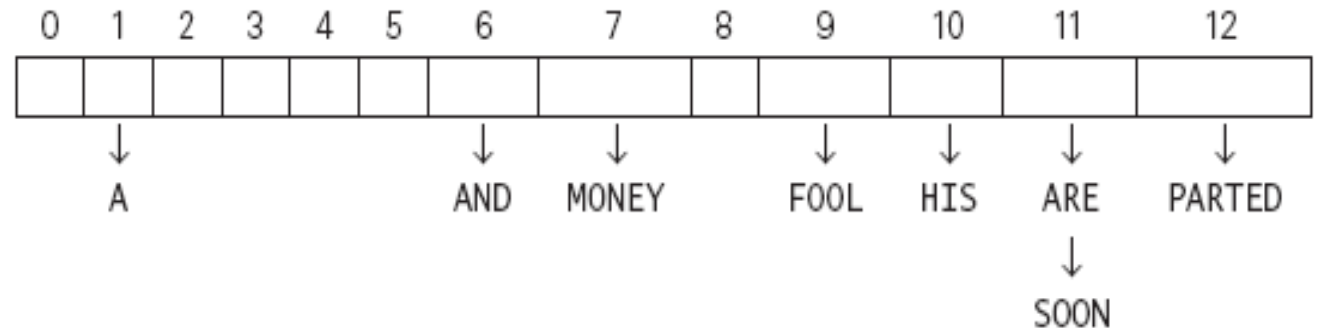
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

- The first key is the word A; its hash value is  $h(A) = 1 \bmod 13 = 1$ .
- The second key—the word FOOL—is installed in the ninth cell since  $(6 + 15 + 15 + 12) \bmod 13 = 9$
- There is a collision of the keys ARE and SOON because

$$h(ARE) = (1 + 18 + 5) \bmod 13 = 11$$

$$h(SOON) = (19 + 15 + 15 + 14) \bmod 13 = 11.$$

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12



# Searching- Open hashing

---

- How do we search in a dictionary implemented as such a table of linked lists?
- We do this by simply applying to a search key the same procedure that was used for creating the table.
- To illustrate, if we want to search for the **key KID** in the hash table, we first compute the value of the same hash function for the key:  $h(KID) = 11$ .
- Since the list attached to cell 11 is not empty, its linked list may contain the search key.
- But because of possible collisions, we cannot tell whether this is the case until we traverse this linked list.
- After comparing the string KID first with the string ARE and then with the string SOON, we end up with an **unsuccessful search**.

# Searching - Open hashing

---

- In general, the efficiency of searching depends on the **lengths of the linked lists**, which, in turn, depend on the dictionary and table sizes, as well as the **quality of the hash function**.
- If the hash function distributes  **$n$  keys among  $m$  cells** of the hash table about evenly, each list will be about  $n/m$  keys long.
- The ratio  $\alpha = n/m$ , called the ***load factor*** of the hash table, plays a crucial role in the efficiency of hashing.
- They are almost identical to **searching sequentially in a linked list**; what we have gained by hashing is a **reduction in average list size by a factor of  $m$** , the size of the hash table.

# Searching - Open hashing

---

- Normally, we want the load factor to be **not far from 1**.
- Having it too small would imply a lot of empty lists and hence inefficient use of space; having it too large would mean longer linked lists and hence longer search times.
- But if we do have the load factor around 1, we have an amazingly efficient scheme that makes it possible to search for a given key for, on average, the **price of one or two comparisons**!
- True, in addition to comparisons, we need to spend time on computing the value of the hash function for a search key, but it is a constant-time operation, independent from  $n$  and  $m$ .
- Note that we are getting this remarkable efficiency not only as a result of the method's ingenuity but also at the expense of extra space.

# Insertion and Deletion - Open hashing

---

- The two other dictionary operations—insertion and deletion—are almost identical to searching.
- Insertions are normally done at the end of a list.
- Deletion is performed by searching for a key to be deleted and then removing it from its list.
- Hence, the efficiency of these operations is identical to that of searching, and they are all  $\Theta(1)$  in the average case if the number of keys  $n$  is about equal to the hash table's size  $m$ .



# Closed hashing

---

- In closed hashing, all keys are stored in the hash table itself without the use of linked lists.
- Different strategies can be employed for collision resolution.
  - **Linear probing:** use next free bucket
  - **Double hashing:** use second hash function to compute increment

# Closed hashing – Linear Probing

---

- The simplest one—called ***linear probing***—checks the cell following the one where the collision occurs.
- If that cell is empty, the new key is installed there;
- If the next cell is already occupied, the availability of that cell's immediate successor is checked, and so on.
- If the end of the hash table is reached, the search is wrapped to the beginning of the table; i.e., it is treated as a **circular array**.

# Closed hashing – Linear Probing

Linear probing: Keys are stored inside a hash table.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON
0	1	2	3	4	5	6	7	8	9	10	11	12
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

# Searching - Linear Probing

---

- To search for a given key  $K$ , we start by computing  $h(K)$  where  $h$  is the hash function used in the table construction.
- If the cell  $h(K)$  is **empty**, the search is **unsuccessful**.
- If the cell is not empty, we must compare  $K$  with the **cell's occupant**: if they are equal, we have found a matching key;
- If they are not, we compare  $K$  with a key in the next cell and continue in this manner until we encounter either a matching key (a successful search) or an empty cell (unsuccessful search).

# Searching - Linear Probing

- For example, if we search for the word LIT in the table, we will get

$$h(\text{LIT}) = (12 + 9 + 20) \bmod 13 = 2$$

- Since cell 2 is empty, we can stop immediately.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

0	1	2	3	4	5	6	7	8	9	10	11	12
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

# Searching - Linear Probing

---

- If we search for KID with

$$h(KID) = (11 + 9 + 4) \bmod 13 = 11,$$

- We will have to compare KID with ARE, SOON, PARTED, and A before we can declare the search unsuccessful.

0	1	2	3	4	5	6	7	8	9	10	11	12
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

# Deletion - Linear Probing

---

- Although the search and insertion operations are straightforward for this version of hashing, **deletion is not**.
- For example, if we **simply delete the key ARE** from the last state of the hash table, we will be **unable to find the key SOON** afterward.
- Indeed, after computing  $h(\text{SOON}) = 11$ , the algorithm would find this location empty and report the unsuccessful search result.
- A simple solution is to use “**lazy deletion**,” i.e., to mark previously occupied locations by a special symbol to **distinguish them from locations that have not been occupied**.

# Linear Probing - Cluster Problem

---

- As the hash table gets closer to being full, the performance of **linear probing deteriorates** because of a phenomenon called **clustering**.
- A **cluster** in linear probing is a sequence of contiguously occupied cells.
- Clusters are bad news in hashing because they make the **dictionary operations less efficient**.
- As clusters become larger, the probability that a new element will be attached to a cluster increases;
- In addition, large clusters increase the probability that **two clusters will coalesce** after a new key's insertion, causing even more clustering.



# Closed hashing – Double Hashing

---

- One of the collision resolution strategies is ***double hashing***.
- Under this scheme, we use another hash function,  $s(K)$ , to determine a fixed increment for the probing sequence to be used after a collision at location

$$l = h(K) \quad (l + s(K)) \bmod m, \quad (l + 2s(K)) \bmod m, \quad \dots\dots$$

- To guarantee that every location in the table is probed by this sequence, the increment  $s(k)$  and the table size  $m$  must be relatively prime, i.e., their only common divisor must be 1.
- This condition is satisfied automatically if  $m$  itself is prime.
- Some functions recommended in the literature are

$$s(k) = m - 2 - k \bmod (m - 2) \text{ and } s(k) = 8 - (k \bmod 8) \text{ for small tables}$$

$$s(k) = k \bmod 97 + 1 \text{ for larger ones.}$$

# Analysis of Closed hashing – Rehashing

---

- Mathematical analysis of double hashing has proved to be quite difficult.
- Some partial results and considerable practical experience with the method suggest that with good hashing functions—both primary and secondary—**double hashing is superior to linear probing.**
- But its performance also **deteriorates when the table gets close to being full.**
- A natural solution in such a situation is ***rehashing***: the current table is scanned, and all its keys are **relocated into a larger table.**

# Analysis of Hashing

---

- It is worthwhile to compare the main properties of hashing **with balanced search trees**—its principal competitor for implementing dictionaries.
- **Asymptotic time efficiency** with hashing, searching, insertion, and deletion can be implemented to take  $\Theta(1)$  time on the average but  $\Theta(n)$  time in the very unlikely worst case.
- For balanced search trees, the average time efficiencies are  $\Theta(\log n)$  for both the average and worst cases.

# CSC 4520/6520

# Design & Analysis of Algorithms

---

## CHAPTER 8: DYNAMIC PROGRAMMING

Abdullah Bal, PhD

# Problem

---

- How many coins, at a minimum, are required to achieve a total of 6?”
- What coins meet the condition mentioned above?



# Dynamic Programming

---

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

# Change Making Problem



# Change Making Problem

---

- Give change for amount  $n$  using the minimum number of coins of denominations
- We consider a dynamic programming algorithm for the general case, assuming availability of unlimited quantities of coins for each of the  $m$  denominations

$$d_1 < d_2 < \dots < d_m$$



# Change Making Problem

---

- Let  $F(n)$  be the minimum number of coins whose values add up to  $n$ ; it is convenient to define

$$F(0) = 0$$

- The amount  $n$  can only be obtained by adding one coin of denomination

$$d_j \text{ to the amount } n - d_j \quad \text{for } j = 1, 2, \dots, m \quad n \geq d_j$$

- Therefore, we can consider all such denominations and select the one minimizing

$$F(n - d_j) + 1$$

- Since 1 is a constant, we can, find the smallest  $F(n - d_j)$  first and then add 1 to it.

# Change Making Problem

---

- We have the following recurrence for  $F(n)$ :

$$F(n) = \min_{j: n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

- We can compute  $F(n)$  by filling a **one-row table** left to right,
- Computing a table entry here requires finding the minimum of up to  $m$  numbers.

# Example 1: Change Making Problem

- The application of the algorithm to amount  $n = 6$  and denominations 1, 3, 4

$$F[0] = 0$$

$n$	0	1	2	3	4	5	6
$F$	0						

$$F[1] = \min\{F[1 - 1]\} + 1 = 1$$

$n$	0	1	2	3	4	5	6
$F$	0	1					

$$F[2] = \min\{F[2 - 1]\} + 1 = 2$$

$n$	0	1	2	3	4	5	6
$F$	0	1	2				

$$F[3] = \min\{F[3 - 1], F[3 - 3]\} + 1 = 1$$

$n$	0	1	2	3	4	5	6
$F$	0	1	2	1			

$$F[4] = \min\{F[4 - 1], F[4 - 3], F[4 - 4]\} + 1 = 1$$

$n$	0	1	2	3	4	5	6
$F$	0	1	2	1	1		

$$F[5] = \min\{F[5 - 1], F[5 - 3], F[5 - 4]\} + 1 = 2$$

$n$	0	1	2	3	4	5	6
$F$	0	1	2	1	1	2	

$$F[6] = \min\{F[6 - 1], F[6 - 3], F[6 - 4]\} + 1 = 2$$

$n$	0	1	2	3	4	5	6
$F$	0	1	2	1	1	2	2



$n=6$

$$F(n) = \min_{j: n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

Result:  $3 + 3 = 6$

# Example 1: Change Making Problem

---

- The answer it yields is two coins.
- To find the coins of an optimal solution, we need to **backtrace** the computations to see which of the denominations produced the minima in the formula.
- For the instance considered, the last application of the formula (for  $n = 6$ ), the minimum was produced by  $d_2 = 3$ .
- The second minimum (for  $n = 6 - 3$ ) was also produced for a coin of that denomination.
- Thus, the minimum-coin set for  $n = 6$  is two 3's.

## Example 2: Change Making Problem

- The application of the algorithm to amount  $n = 7$  and denominations 1, 2, 4

$$F(n) = \min_{j: n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

$n$	0	1	2	3	4	5	6	7
$F$	0							



$n=7$

## Example 2: Change Making Problem

- The application of the algorithm to amount  $n = 7$  and denominations 1, 2, 4

$$F(0)=0$$

$$F(1)=\min\{F(1-1)\}+1=1$$

$$F(2)=\min\{F(2-1), F(2-2)\}+1=1$$

$$F(3)=\min\{F(3-1), F(3-2)\}+1=2$$

$$F(4)=\min\{F(4-1), F(4-2), F(4-4)\}+1=1$$

$$F(5)=\min\{F(5-1), F(5-2), F(5-4)\}+1=2$$

$$F(6)=\min\{F(6-1), F(6-2), F(6-4)\}+1=2$$

$$F(7)=\min\{F(7-1), F(7-2), F(7-4)\}+1=3$$

$$F(n) = \min_{j: n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

$n$	0	1	2	3	4	5	6	7
$F$	0	1	1	2	1	2	2	3

Result:  $4 + 2 + 1 = 7$



$n=7$

# Example 3: Change Making Problem

- The application of the algorithm to amount  $n = 10$  and denominations 1, 5, 6, 9

$$F(0)=0$$

$$F(1)=\min\{F(1-1)\}+1=1$$

$$F(2)=\min\{F(2-1)\}+1=2$$

$$F(3)=\min\{F(3-1)\}+1=3$$

$$F(4)=\min\{F(4-1)\}+1=4$$

$$F(5)=\min\{F(5-1), F(5-5)\}+1=1$$

$$F(6)=\min\{F(6-1), F(6-5), F(6-6)\}+1=2$$

$$F(7)=\min\{F(7-1), F(7-5), F(7-6)\}+1=2$$

$$F(8)=\min\{F(8-1), F(8-5), F(8-6)\}+1=3$$

$$F(9)=\min\{F(9-1), F(9-5), F(9-6), F(9-9)\}+1=1$$

$$F(10)=\min\{F(10-1), F(10-5), F(10-6), F(10-9)\}+1=2$$

$$F(n) = \min_{j: n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

$n$	0	1	2	3	4	5	6	7	8	9	10
$F$	0	1	2	3	4	1	2	2	3	1	2

Result: **9 + 1**

or

**5 + 5**



$n=10$

# Change Making Problem

---

**ALGORITHM** *ChangeMaking*( $D[1..m], n$ )

//Applies dynamic programming to find the minimum number of coins  
//of denominations  $d_1 < d_2 < \dots < d_m$  where  $d_1 = 1$  that add up to a  
//given amount  $n$

//Input: Positive integer  $n$  and array  $D[1..m]$  of increasing positive  
// integers indicating the coin denominations where  $D[1] = 1$

//Output: The minimum number of coins that add up to  $n$

$F[0] \leftarrow 0$

**for**  $i \leftarrow 1$  **to**  $n$  **do**

$temp \leftarrow \infty; j \leftarrow 1$

**while**  $j \leq m$  **and**  $i \geq D[j]$  **do**

$temp \leftarrow \min(F[i - D[j]], temp)$

$j \leftarrow j + 1$

$F[i] \leftarrow temp + 1$

**return**  $F[n]$

- The time efficiency of the algorithm is  $\Theta(nm)$ .





# Coin-collecting by robot

- Several coins are placed in cells of an  $n \times m$  board, **no more than one coin per cell**.
- A robot, located in the upper left cell of the board, needs to collect **as many of the coins as possible** and bring them to the bottom right cell.
- On each step, the robot can move either **one cell to the right or one cell down** from its current location.
- When the robot visits a cell with a coin, it **always picks up** that coin.
- Design an algorithm to find the **maximum number** of coins the robot can collect and **a path it needs to follow to do this**.

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

# Coin-collecting problem

---

- Let  $F(i,j)$  be the largest number of coins the robot can collect and bring to cell  $(i,j)$  in the  $i$ th row and  $j$ th column.
- The largest number of coins that can be brought to cell  $(i,j)$ :
  - from the left neighbor ?
  - from the neighbor above?

- The recurrence:

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \leq i \leq n, \quad 1 \leq j \leq m$$

$$F(0, j) = 0 \quad \text{for } 1 \leq j \leq m \quad \text{and} \quad F(i, 0) = 0 \quad \text{for } 1 \leq i \leq n,$$

where  $c_{ij}$  = value of the coin, if there is a coin in cell  $(i,j)$

$c_{ij} = 0$  otherwise

# Coin-collecting problem

---

- Tracing the computations backward makes it possible to get an optimal path:
- If  $F(i - 1, j) > F(i, j - 1)$ , an optimal path to cell  $(i, j)$  must come down from the adjacent cell above it;
- If  $F(i - 1, j) < F(i, j - 1)$ , an optimal path to cell  $(i, j)$  must come from the adjacent cell on the left;
- If  $F(i - 1, j) = F(i, j - 1)$ , it can reach cell  $(i, j)$  from either direction.
- This yields two optimal paths for the instance.

# Example 1: Coin-collecting problem

- We can fill in the  $n \times m$  table of  $F(i, j)$  values either row by row or column by column, as is typical for dynamic programming algorithms involving two-dimensional tables.

	1	2	3	4	5	6
1					1	
2		1		1		
3				1		1
4			1			1
5	1				1	

Coins to collect

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

Dynamic programming algorithm results

	1	2	3	4	5	6
1					1	
2		1		1		
3				1		1
4			1			1
5	1				1	










Two paths to collect 5 coins, the maximum number of coins possible.

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \leq i \leq n, \quad 1 \leq j \leq m$$










$$F(0, j) = 0 \quad \text{for } 1 \leq j \leq m \quad \text{and} \quad F(i, 0) = 0 \quad \text{for } 1 \leq i \leq n,$$

## Example 2: Coin-collecting problem

---

	1	2	3	4	5	6
1					 6	
2		 5		 4		
3				 3		 8
4			 9			 3
5	 10				 9	

## Example 2: Solution of the coin-collecting problem

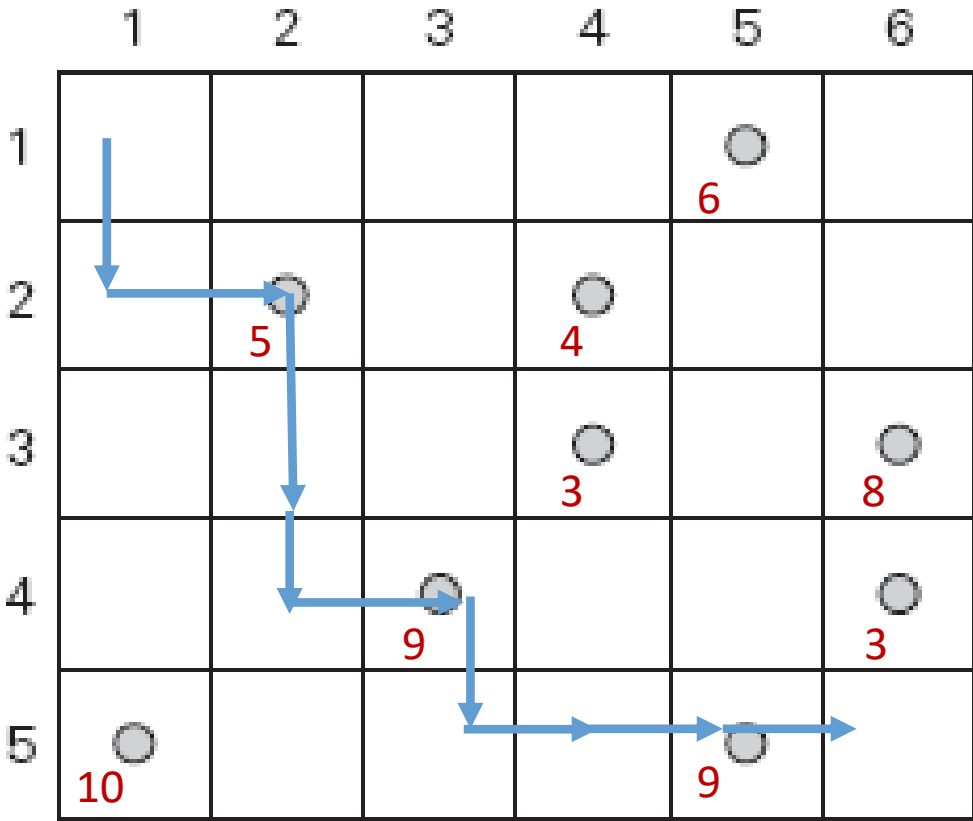
	1	2	3	4	5	6
1					 6	
2		 5		 4		
3				 3		 8
4			 9			 3
5	 10				 9	

	1	2	3	4	5	6
1	0	0	0	0	6	6
2	0	5	5	9	9	9
3	0	5	5	12	12	20
4	0	5	14	14	14	23
5	10	10	14	14	23	<b>23</b>

# Example 2: Solution of the coin-collecting problem

	1	2	3	4	5	6
1	0	0	0	0	6	6
2	0	5	5	9	9	9
3	0	5	5	12	12	20
4	0	5	14	14	14	23
5	10	10	14	14	23	<b>23</b>

Two equal routs are possible.



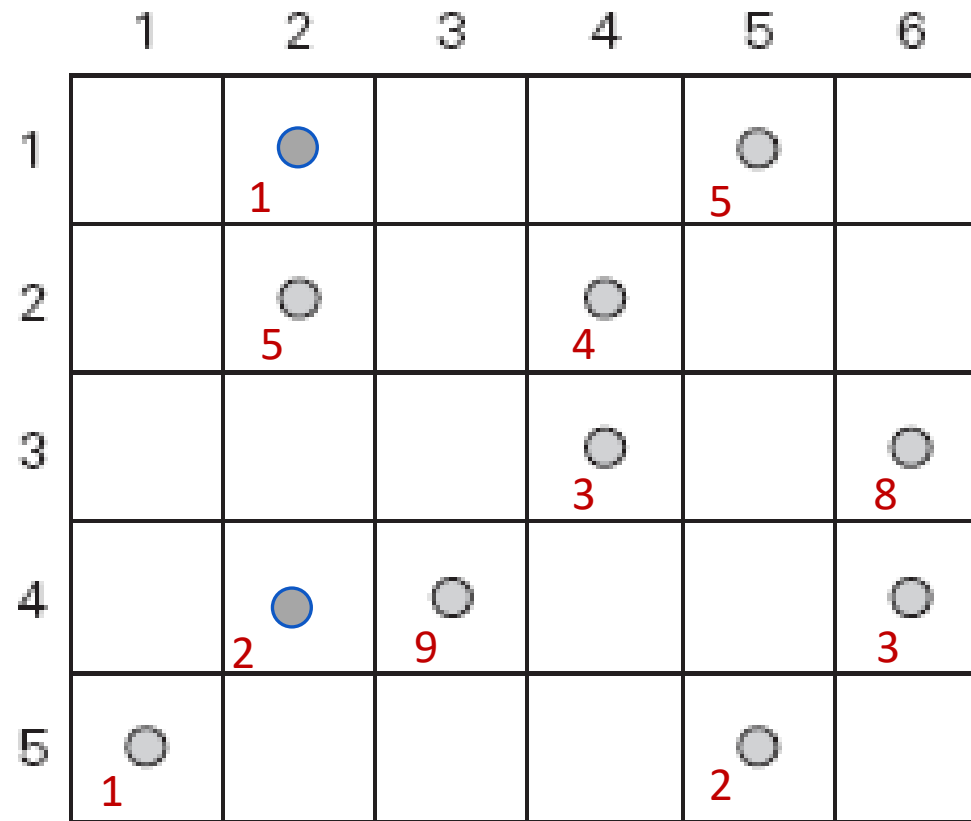
Total:  $5 + 9 + 9 = 23$  or

Total:  $3 + 8 + 3 + 4 + 5 = 23$














## Example 3: Coin-collecting problem

---



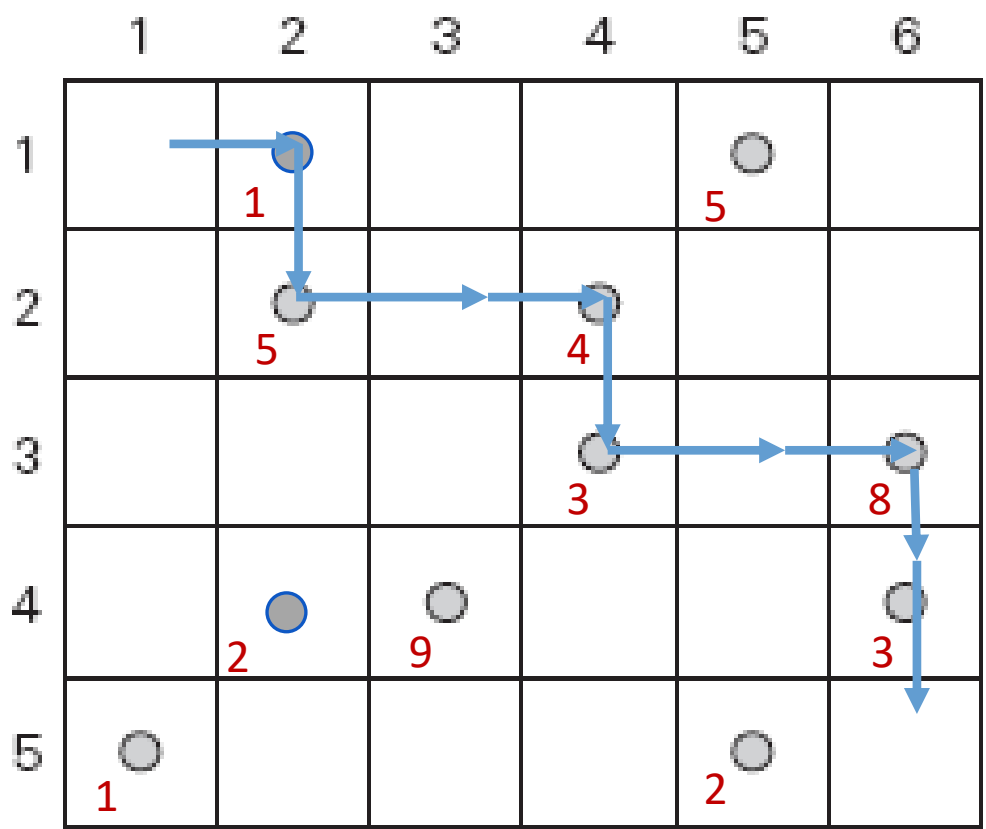
## Example 3: Solution of the coin-collecting problem

	1	2	3	4	5	6
1		 1			 5	
2		 5		 4		
3				 3		 8
4		 2	 9			 3
5	 1				 2	

	1	2	3	4	5	6
1	0	1	1	1	6	6
2	0	6	6	10	10	10
3	0	6	6	13	13	21
4	0	8	17	17	17	24
5	1	8	17	17	19	<b>24</b>

# Example 3: Solution of the coin-collecting problem

	1	2	3	4	5	6
1	0	1	1	1	6	6
2	0	6	6	10	10	10
3	0	6	6	13	13	21
4	0	8	17	17	17	24
5	1	8	17	17	19	24



Total:  $1 + 5 + 4 + 3 + 8 + 3 = 24$

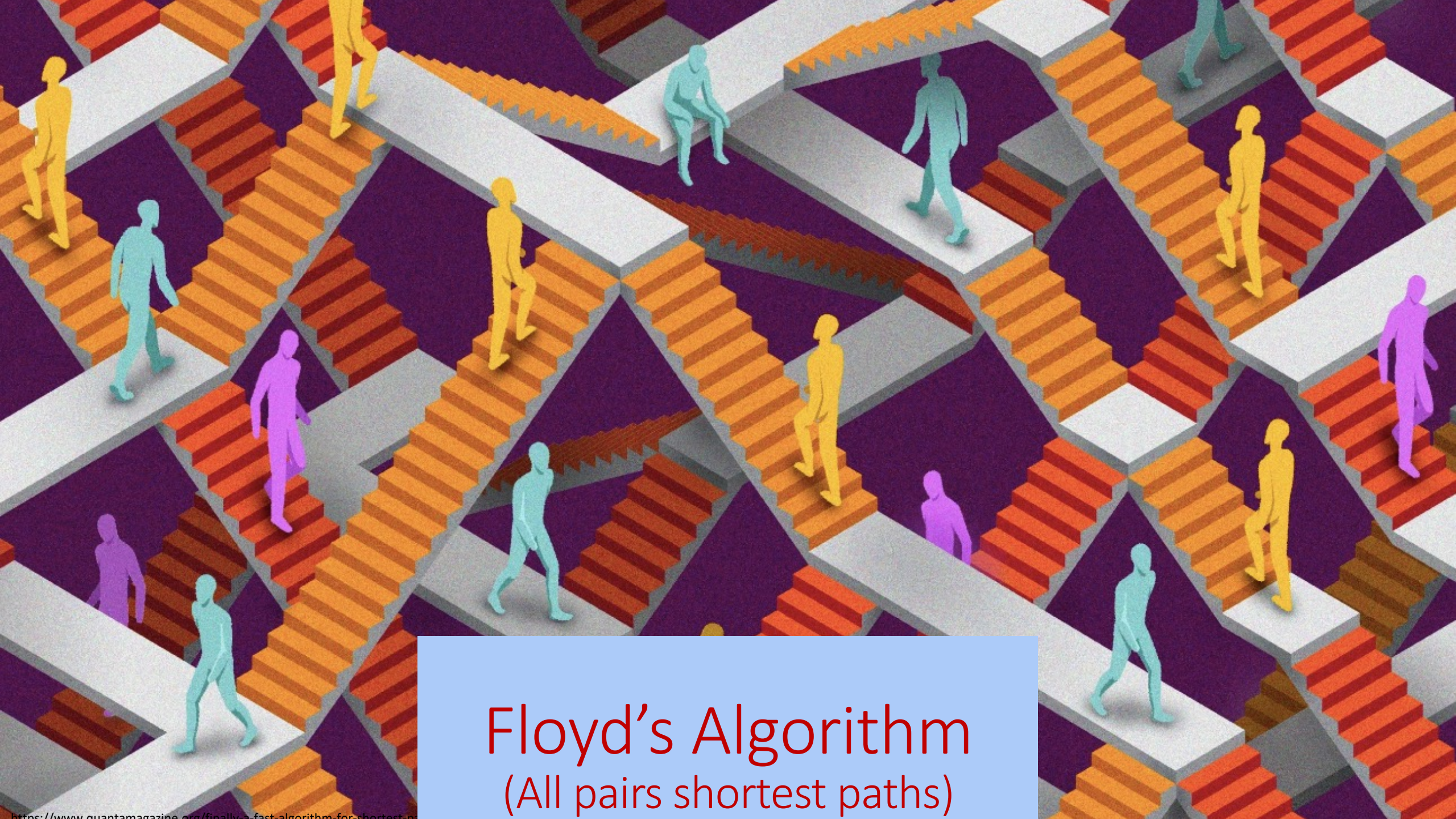
# The coin-collecting problem

- Since computing the value of  $F(i, j)$  by the formula for each cell of the table takes constant time, the time efficiency of the algorithm is  $\Theta(nm)$ .
- Its space efficiency is, obviously, also  $\Theta(nm)$ .

**ALGORITHM** *RobotCoinCollection*( $C[1..n, 1..m]$ )

```
//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an  $n \times m$  board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix  $C[1..n, 1..m]$  whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell  $(n, m)$ 
 $F[1, 1] \leftarrow C[1, 1]$ ; for  $j \leftarrow 2$  to  $m$  do  $F[1, j] \leftarrow F[1, j - 1] + C[1, j]$ 
for  $i \leftarrow 2$  to  $n$  do
     $F[i, 1] \leftarrow F[i - 1, 1] + C[i, 1]$ 
    for  $j \leftarrow 2$  to  $m$  do
         $F[i, j] \leftarrow \max(F[i - 1, j], F[i, j - 1]) + C[i, j]$ 
return  $F[n, m]$ 
```



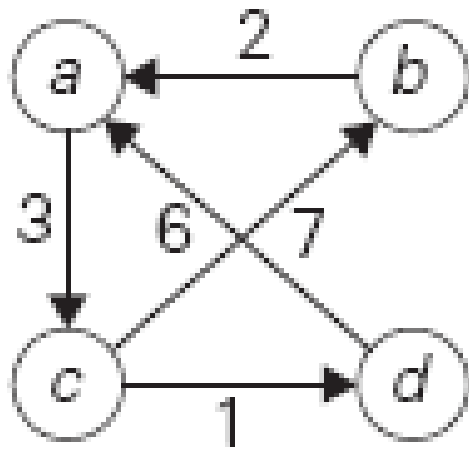


# Floyd's Algorithm

(All pairs shortest paths)



# All Pairs Shortest Paths Problem



$$W = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

$$D = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{matrix}$$

C → a ?

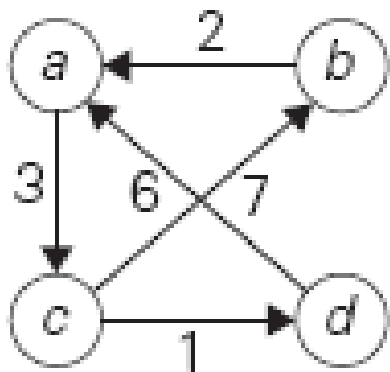
# All pairs shortest paths

---

- Given a weighted connected graph (**undirected or directed**), the ***all-pairs shortest-paths problem*** asks to find the distances—i.e., the **lengths of the shortest paths**— **from each vertex to all other vertices**.
- Because of its important applications to communications, transportation networks, and operations research, it has been thoroughly studied over the years.
- Among recent applications of the all-pairs shortest-path problem is **precomputing distances** for motion planning **in computer games**.

# All pairs shortest paths

- It is convenient to record the lengths of shortest paths in an  $n \times n$  matrix  $D$  called the ***distance matrix***:
- The element  $d_{ij}$  in the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column of this matrix indicates the length of the shortest path from the  $i^{\text{th}}$  vertex to the  $j^{\text{th}}$  vertex.



$$W = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

$$D = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{matrix}$$



# Floyd's Algorithm: All pairs shortest paths

---

- We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm.
- It is called ***Floyd's algorithm*** after its co-inventor Robert W. Floyd.
- The algorithm can be enhanced to find not only the lengths of the shortest paths for all vertex pairs but also **the shortest paths themselves**.

# Floyd's Algorithm: All pairs shortest paths

- Floyd's algorithm computes the distance matrix of a weighted graph with  $n$  vertices through a series of  $n \times n$  matrices:

$$D^{(0)}, \dots, D^{(k-1)}, D^{(k)}, \dots, D^{(n)}.$$

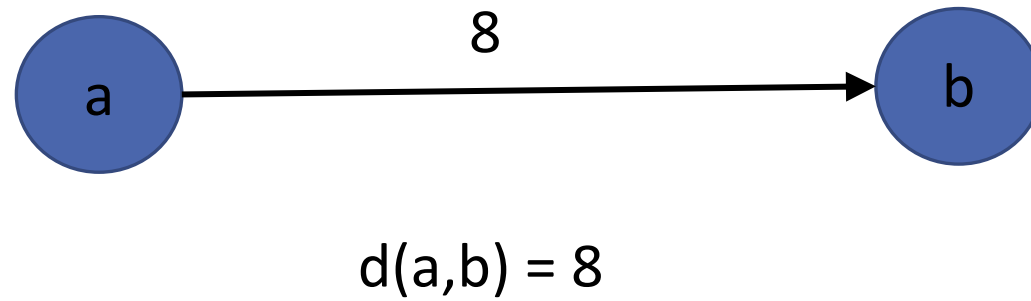
- On the  $k$ -th iteration, the algorithm determines shortest paths between every pair of vertices  $i, j$  that use only vertices among  $1, \dots, k$  as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

- We can compute all the elements of each matrix  $D^{(k)}$  from its immediate predecessor  $D^{(k-1)}$

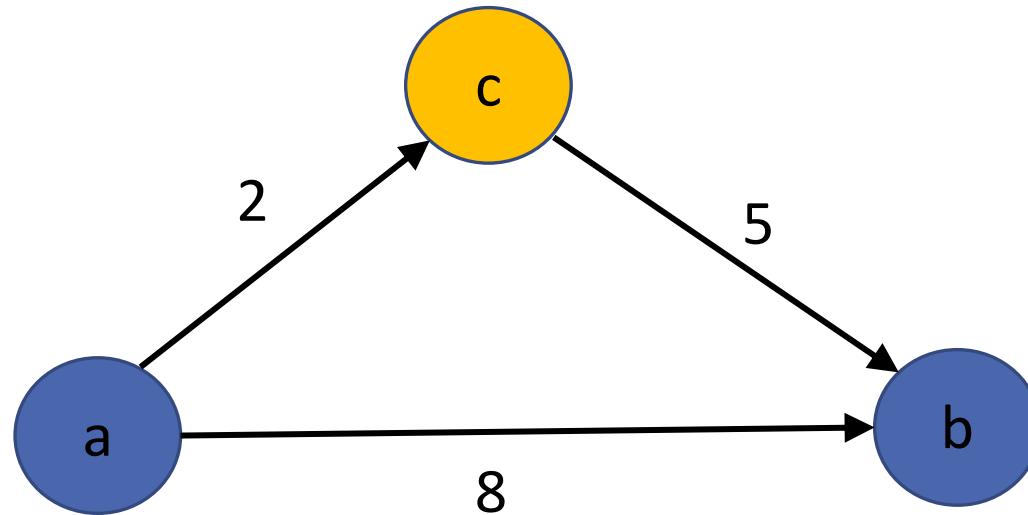
# Floyd's Algorithm: All pairs shortest paths

---



# Floyd's Algorithm: All pairs shortest paths

---

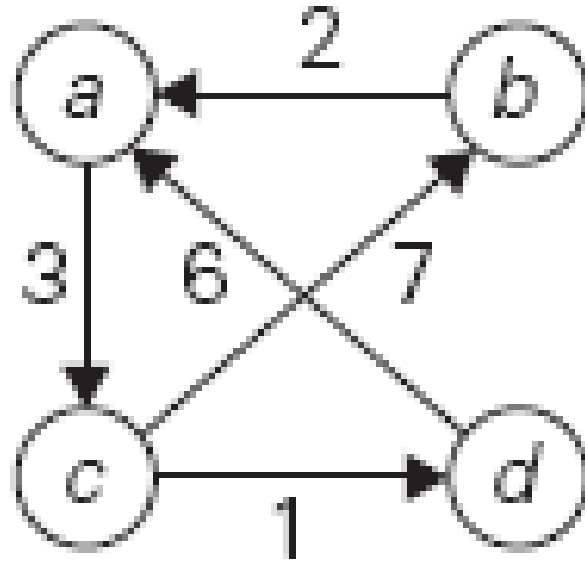


$$\begin{aligned} d(a,b) &= \min\{d(a,b) , (d(a,c) + d(c,b)) \} \\ &= \min\{8 , (2+5) \}=7 \end{aligned}$$

# Example 1: Floyd's Algorithm

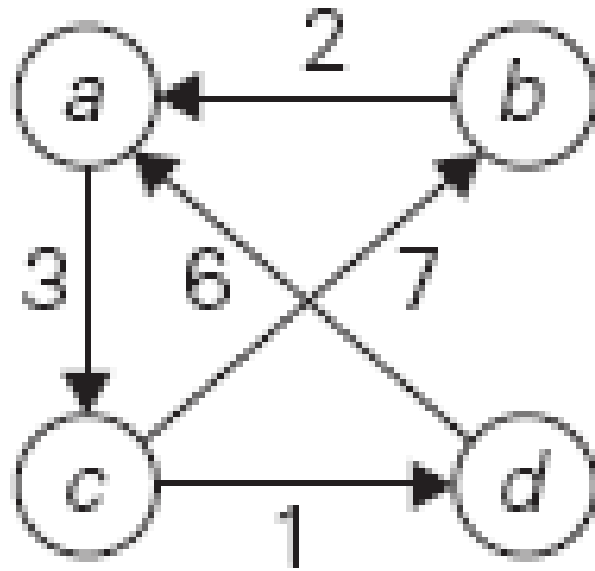
---

- In a weighted (di)graph, find shortest paths between every pair of vertices



# Example 1: Floyd's Algorithm

- Step 1: Generate weight matrix  $D^{(0)}$  (Adjacency matrix ).



$D^{(0)} =$

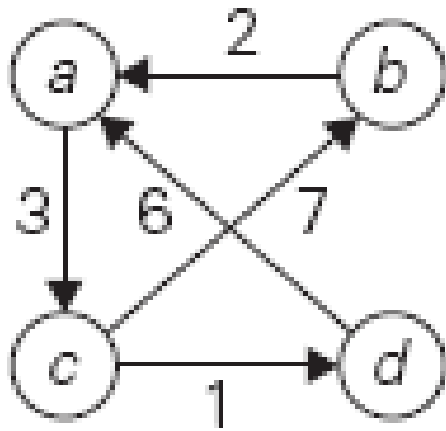
	a	b	c	d
a	0	$\infty$	3	$\infty$
b	2	0	$\infty$	$\infty$
c	$\infty$	7	0	1
d	6	$\infty$	$\infty$	0

Formula of D:

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

# Example 1: Floyd's Algorithm

**Step 2:** Generate  $D^{(1)}$  considering the shortest path for each vertex through vertex **a**.



$D^{(0)} =$

	a	b	c	d
a	0	$\infty$	3	$\infty$
b	2	0	$\infty$	$\infty$
c	$\infty$	7	0	1
d	6	$\infty$	$\infty$	0

$D^{(1)} =$

	a	b	c	d
a	0	$\infty$	3	$\infty$
b	2	0	5	$\infty$
c	$\infty$	7	0	1
d	6	$\infty$	9	0

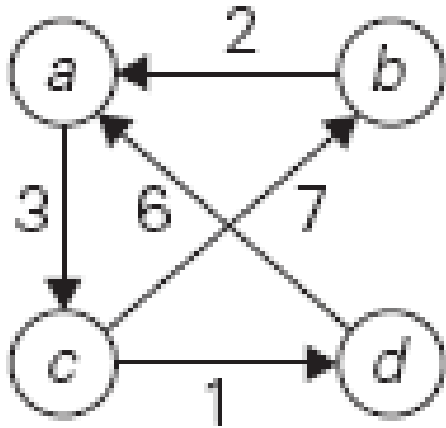
- $b, c = \min\{(b \rightarrow c), (b \rightarrow a + a \rightarrow c)\} = \min\{\infty, 2 + 3\} = 5$
- $b, d = \min\{(b \rightarrow d), (b \rightarrow a + a \rightarrow d)\} = \min\{\infty, 2 + \infty\} = \infty$

- $c, b = \min\{(c \rightarrow b), (c \rightarrow a + a \rightarrow b)\} = \min\{7, \infty + \infty\} = 7$
- $c, d = \min\{(c \rightarrow d), (c \rightarrow a + a \rightarrow d)\} = \min\{1, \infty + \infty\} = 1$

- $d, b = \min\{(d \rightarrow b), (d \rightarrow a + a \rightarrow b)\} = \min\{\infty, 6 + \infty\} = \infty$
- $d, c = \min\{(d \rightarrow c), (d \rightarrow a + a \rightarrow c)\} = \min\{\infty, 6 + 3\} = 9$

# Example 1: Floyd's Algorithm

**Step 3:** Generate  $D^{(2)}$  considering the shortest path for each vertex through vertex **b**.



$D^{(1)} =$

	a	b	c	d
a	0	$\infty$	3	$\infty$
b	2	0	5	$\infty$
c	$\infty$	7	0	1
d	6	$\infty$	9	0

$D^{(2)} =$

	a	b	c	d
a	0	$\infty$	3	$\infty$
b	2	0	5	$\infty$
c	9	7	0	1
d	6	$\infty$	9	0

- $a, c = \min\{(a \rightarrow c), (a \rightarrow b + b \rightarrow c)\} = \min\{3, \infty + 5\} = 3$
- $a, d = \min\{(a \rightarrow d), (a \rightarrow b + b \rightarrow d)\} = \min\{\infty, \infty + \infty\} = \infty$

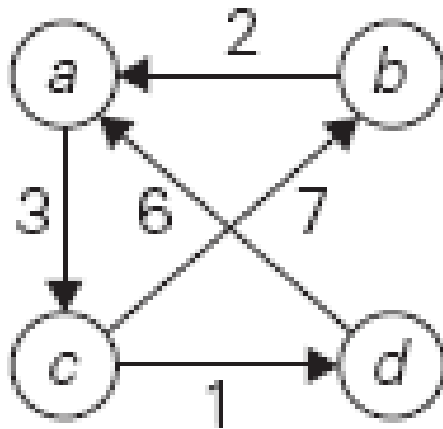
- $c, a = \min\{(c \rightarrow a), (c \rightarrow b + b \rightarrow a)\} = \min\{\infty, 7 + 2\} = 9$
- $c, d = \min\{(c \rightarrow d), (c \rightarrow b + b \rightarrow d)\} = \min\{1, 7 + \infty\} = 1$

- $d, a = \min\{(d \rightarrow a), (d \rightarrow b + b \rightarrow a)\}$   
 $= \min\{6, \infty + 2\} = 6$
- $d, c = \min\{(d \rightarrow c), (d \rightarrow b + b \rightarrow c)\}$   
 $= \min\{9, \infty + 5\} = 9$



# Example 1: Floyd's Algorithm

**Step 4:** Generate  $D^{(3)}$  considering the shortest path for each vertex through vertex  $c$ .



$D^{(2)} =$

	a	b	c	d
a	0	$\infty$	3	$\infty$
b	2	0	5	$\infty$
c	9	7	0	1
d	6	$\infty$	9	0

$D^{(3)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

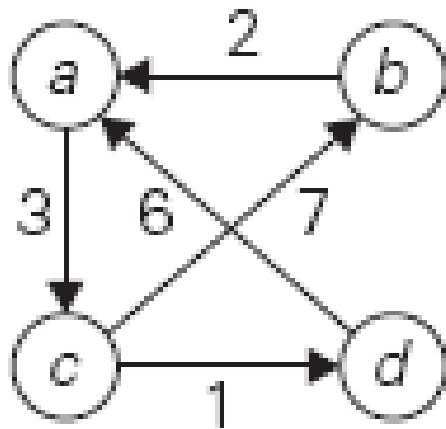
- $a, b = \min\{(a \rightarrow b), (a \rightarrow c + c \rightarrow b)\} = \min\{\infty, 3 + 7\} = 10$
- $a, d = \min\{(a \rightarrow d), (a \rightarrow c + c \rightarrow d)\} = \min\{\infty, 3 + 1\} = 4$

- $b, a = \min\{(b \rightarrow a), (b \rightarrow c + c \rightarrow a)\} = \min\{2, 5 + 9\} = 2$
- $b, d = \min\{(b \rightarrow d), (b \rightarrow c + c \rightarrow d)\} = \min\{\infty, 5 + 1\} = 6$

- $d, a = \min\{(d \rightarrow a), (d \rightarrow c + c \rightarrow a)\} = \min\{6, 9 + 9\} = 6$
- $d, b = \min\{(d \rightarrow b), (d \rightarrow c + c \rightarrow b)\} = \min\{\infty, 9 + 7\} = 16$

# Example 1: Floyd's Algorithm

**Step 5:** Generate  $D^{(4)}$  considering the shortest path for each vertex through vertex d.



$D^{(3)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

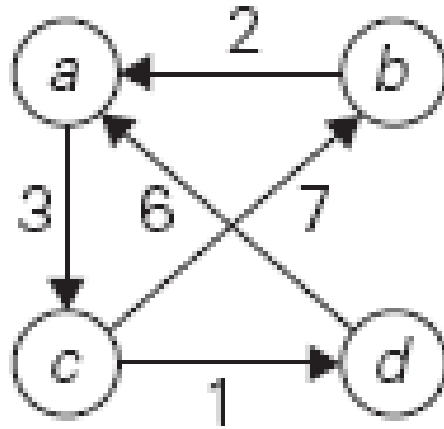
- $a, b = \min\{(a \rightarrow b), (a \rightarrow d + d \rightarrow b)\} = \min\{10, 4 + 16\} = 10$
- $a, c = \min\{(a \rightarrow c), (a \rightarrow d + d \rightarrow c)\} = \min\{3, 4 + 9\} = 3$

- $b, a = \min\{(b \rightarrow a), (b \rightarrow d + d \rightarrow a)\} = \min\{2, 6 + 6\} = 2$
- $b, c = \min\{(b \rightarrow c), (b \rightarrow d + d \rightarrow c)\} = \min\{5, 6 + 9\} = 5$

- $c, a = \min\{(c \rightarrow a), (c \rightarrow d + d \rightarrow a)\}$   
 $= \min\{9, 1 + 6\} = 7$
- $c, b = \min\{(c \rightarrow b), (c \rightarrow d + d \rightarrow b)\}$   
 $= \min\{7, 1 + 16\} = 7$

# Example 1: Floyd's Algorithm

- Final distance matrix for the given graph:



$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

# Floyd's Algorithm (pseudocode and analysis)

---

**ALGORITHM** *Floyd*( $W[1..n, 1..n]$ )

//Implements Floyd's algorithm for the all-pairs shortest-paths problem

//Input: The weight matrix  $W$  of a graph with no negative-length cycle

//Output: The distance matrix of the shortest paths' lengths

$D \leftarrow W$  //is not necessary if  $W$  can be overwritten

**for**  $k \leftarrow 1$  **to**  $n$  **do**

**for**  $i \leftarrow 1$  **to**  $n$  **do**

**for**  $j \leftarrow 1$  **to**  $n$  **do**

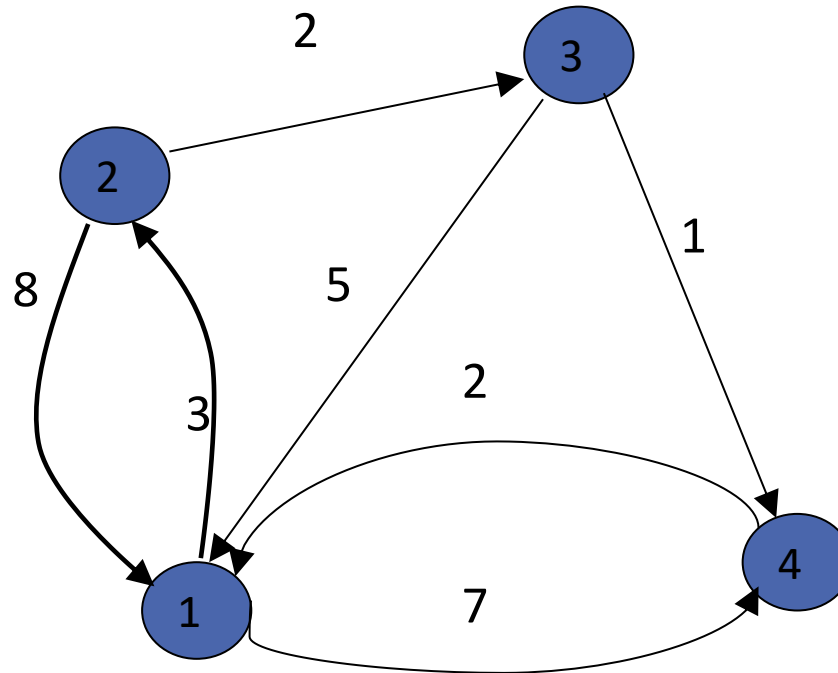
$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

**return**  $D$

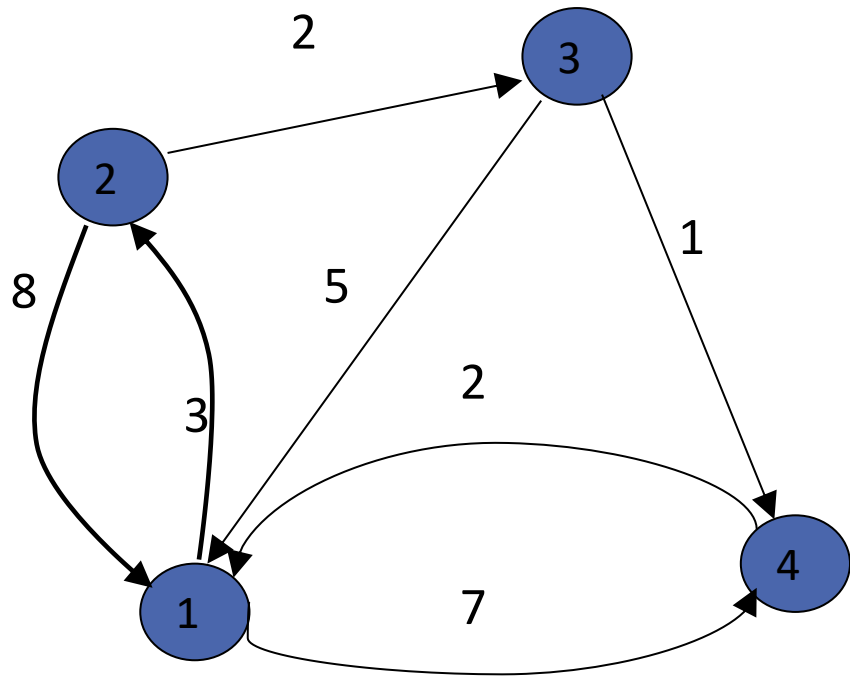
- Time efficiency:  $\Theta(n^3)$
- Space efficiency: Matrices can be written over their predecessors

## Example 2: Floyd's Algorithm

- Find shortest paths between every pair of vertices



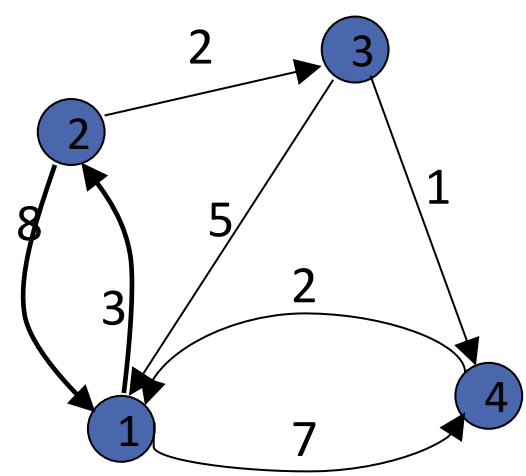
## Example 2: Floyd's Algorithm



$D^{(0)} =$

	1	2	3	4
1	0	3	$\infty$	7
2	8	0	2	$\infty$
3	5	$\infty$	0	1
4	2	$\infty$	$\infty$	0

# Example 2: Floyd's Algorithm



D<sup>(0)</sup>=

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

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

D<sup>(1)</sup>

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

D<sup>(2)</sup>

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

D<sup>(3)</sup>

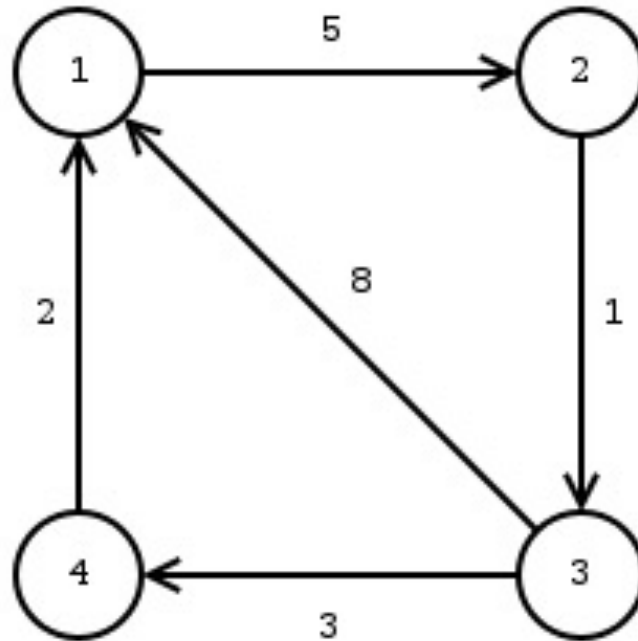
	1	2	3	4
1	0	3	5	6
2	5	0	2	3
3	3	6	0	1
4	2	5	7	0

D<sup>(4)</sup>

# Example 3: Floyd's Algorithm

---

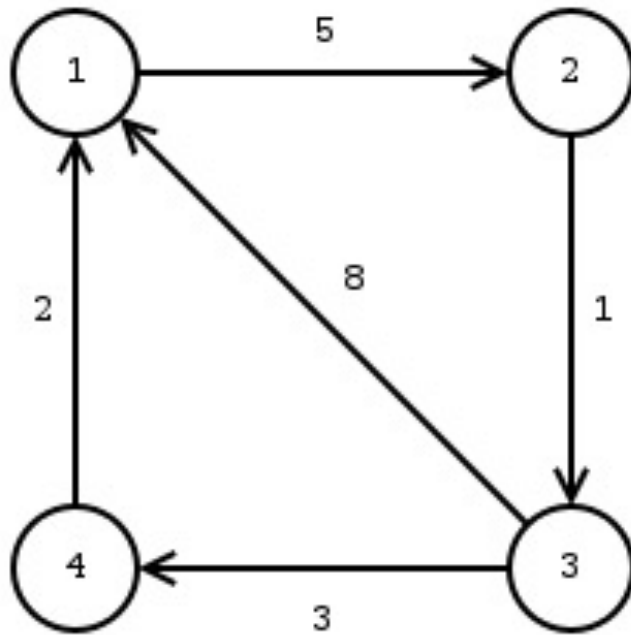
- Find shortest paths between every pair of vertices





# Example 3: Floyd's Algorithm

- Solution



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

$D^{(4)}$

# CSC 4520/6520

# Design & Analysis of Algorithms

---

## CHAPTER 9: GREEDY TECHNIQUE

Abdullah Bal, PhD

# Greedy Technique

---

- This change-making problem solution technique is called *greedy*.
- Computer scientists consider it a general design technique despite the fact that it is applicable to optimization problems only.
- The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- On each step—and this is the central point of this technique—the choice made must be:
  - feasible, i.e., it has to satisfy the problem's constraints
  - locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step
  - irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm
- For some problems, yields an optimal solution for every instance.
- For most, does not but can be useful for fast approximations.

# Applications of the Greedy Strategy

---

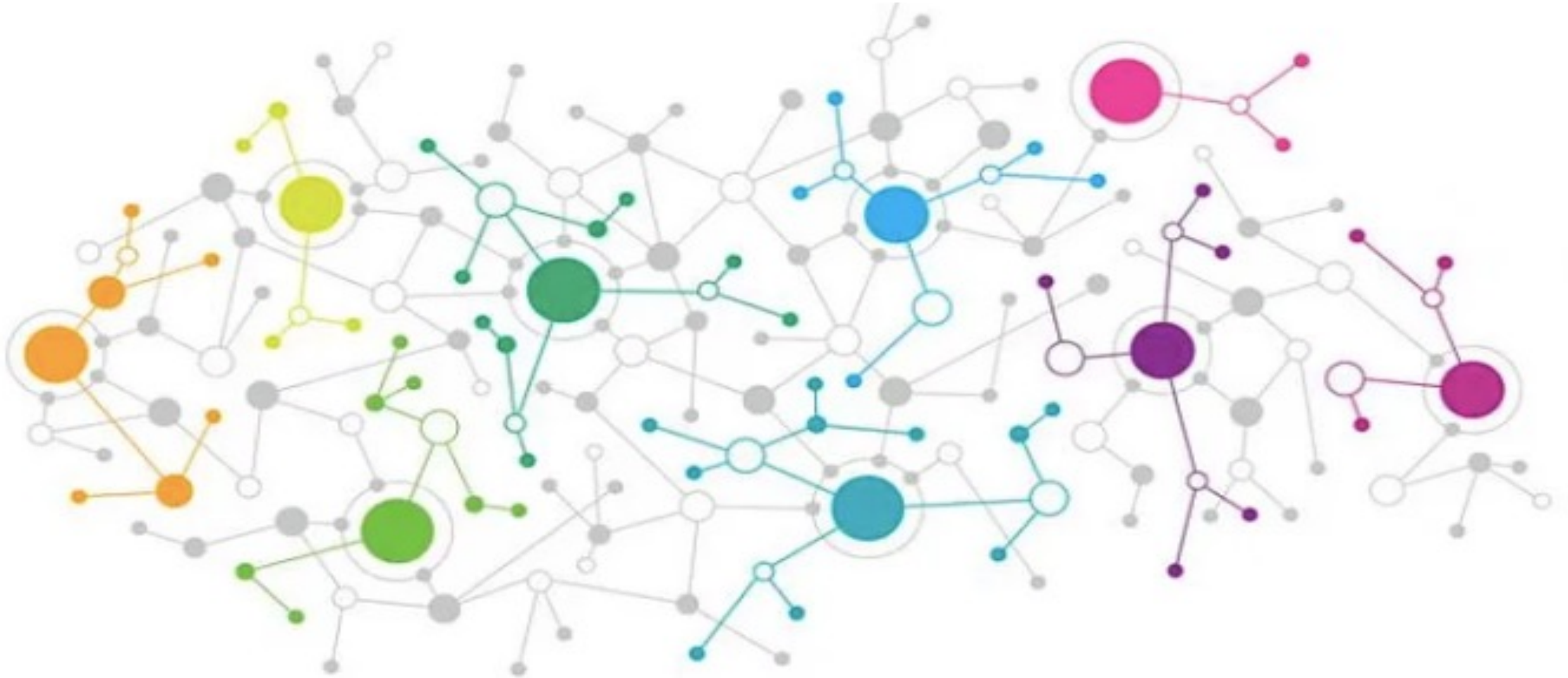
## **Optimal solutions:**

- change making for “normal” coin denominations
- **minimum spanning tree (MST)**
- **single-source shortest paths**
- simple scheduling problems
- **Huffman codes**

## **Approximations:**

- traveling salesman problem (TSP)
- knapsack problem
- other combinatorial optimization problems

# Prim's Algorithm (Minimum Spanning Tree)



# Minimum Spanning Tree (MST)

---

Spanning tree of a connected graph  $G$ :

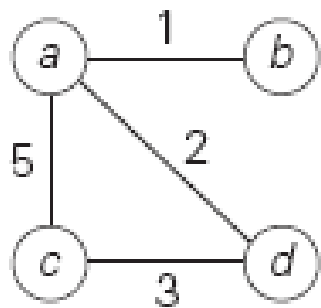
- a connected **acyclic** subgraph of  $G$  that includes all of  $G$ 's vertices

Minimum spanning tree of a weighted, connected graph  $G$ :

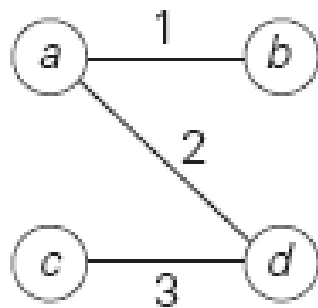
- a spanning tree of  $G$  of **minimum total weight**

# Minimum Spanning Tree (MST) Problem

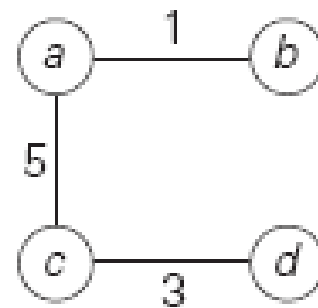
- Graph and its spanning trees, with  $T_1$  being the minimum spanning tree.



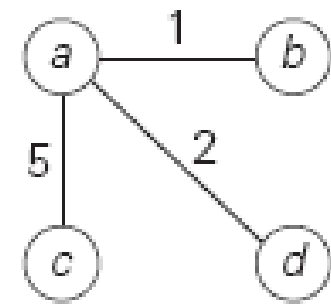
graph



$w(T_1) = 6$



$w(T_2) = 9$



$w(T_3) = 8$

# Minimum Spanning Tree (MST)

---

**DEFINITION** 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.

- If such a graph has weights assigned to its edges, 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.



# Prim's MST algorithm

---

- If we were to try constructing a minimum spanning tree by **exhaustive search**, we would face two serious obstacles.
  - First, the number of spanning trees **grows exponentially with the graph size**.
  - Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a *minimum* spanning tree for a weighted graph by using one of several efficient algorithms available for this problem.
- In this section, we outline ***Prim's algorithm***, which goes back to at least 1957.

# Prim's MST algorithm

---

- Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.
- The initial subtree in such a sequence consists of a single vertex **selected arbitrarily** from the set  $V$  of the graph's vertices.
- On each iteration, the algorithm **expands the current tree in the greedy manner** by simply attaching to it the nearest vertex not in that tree.
- By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree **by an edge of the smallest weight**.
- The algorithm **stops after all the graph's vertices have been included** in the tree being constructed.
- Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is  $n - 1$ , where  $n$  is the number of vertices in the graph.

# Prim's MST algorithm

---

- We can provide such information by attaching two labels to a vertex: **the name of the nearest tree vertex** and **the length (the weight) of the corresponding edge**.
- Vertices that are not adjacent to any of the tree vertices can be given the  **$\infty$  label** indicating their “infinite” distance to the tree vertices and a **null label** for the name of the nearest tree vertex.
- With such labels, finding the next vertex to be added to the current tree becomes a simple task of finding a vertex with the smallest distance label in the set  $V - V_T$ .

# Prim's MST algorithm

---

- Start with tree  $T_1$  consisting of **one (any) vertex** and “grow” tree one vertex at a time to produce MST through a series of expanding subtrees

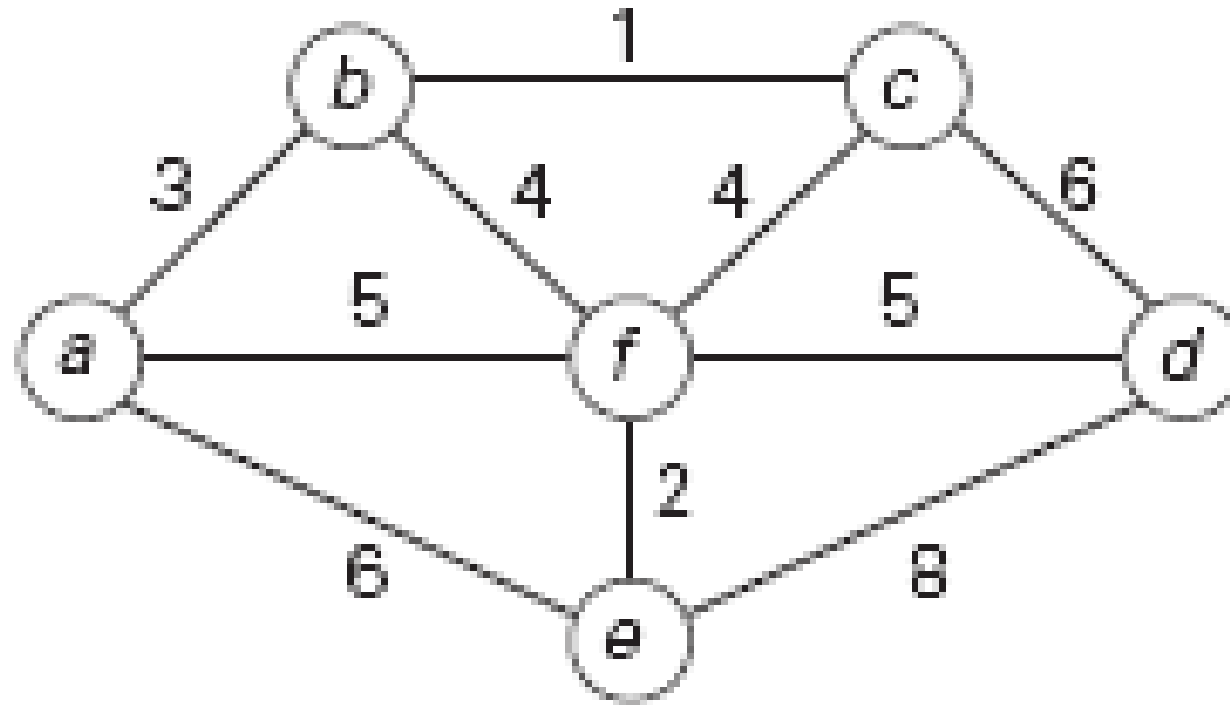
$$T_1, T_2, \dots, T_n$$

- On each iteration, construct  $T_{i+1}$  from  $T_i$  by adding vertex not in  $T_i$  that is closest to those already in  $T_i$  (this is a “greedy” step!)
- **Stop when all vertices are included**

# Example 1: Prim's MST algorithm

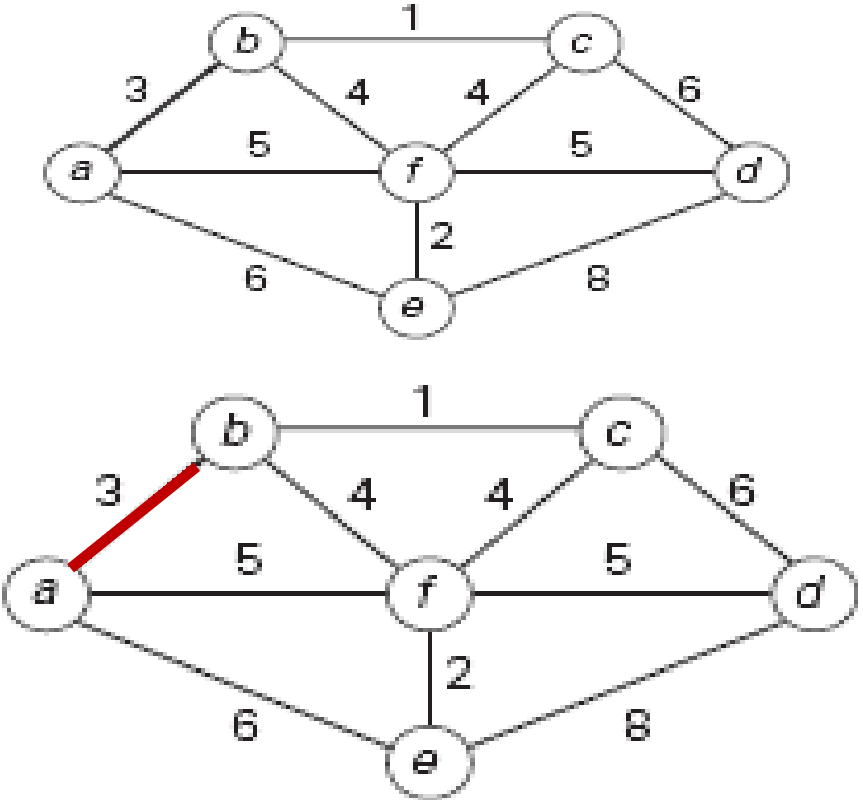
---

- Construct the minimum spanning tree for following graph:



# Example 1: Prim's MST algorithm

## Step 1

Tree vertices	Remaining vertices	Illustration
$a(-, -)$	$b(a, 3)$ $c(-, \infty)$ $d(-, \infty)$ $e(a, 6)$ $f(a, 5)$	 <p>The illustration shows two states of a weighted undirected graph with 6 vertices: a, b, c, d, e, and f. The edges and their weights are: (a,b) weight 3, (a,f) weight 5, (a,e) weight 6, (b,c) weight 1, (b,f) weight 4, (c,d) weight 6, (c,f) weight 4, (d,f) weight 5, (d,e) weight 8, and (e,f) weight 2. The top diagram shows the initial graph. The bottom diagram shows the graph after selecting vertex 'a' and its incident edges (a-b, a-f, a-e). The edge (a-b) is highlighted in red.</p>
Selected Vertex $b(a, 3)$		

# Example 1: Prim's MST algorithm

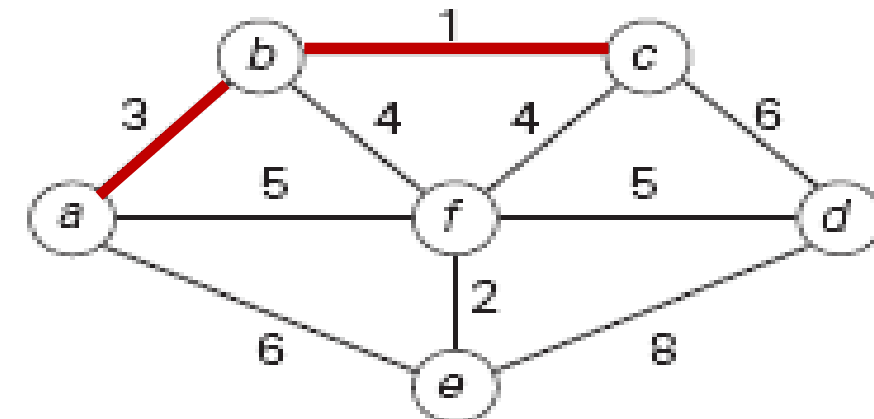
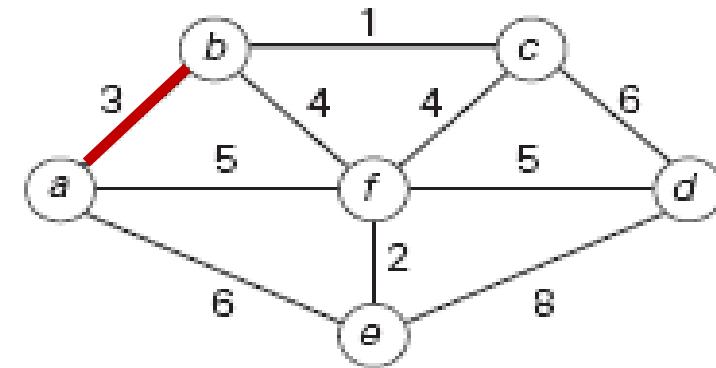
## Step 2

**Tree vertices**

**Remaining vertices**

**Illustration**

$c(b, 1)$   $d(-, \infty)$   $e(a, 6)$   
 $f(b, 4)$



Selected Vertex  
 $c(b, 1)$

# Example 1: Prim's MST algorithm

Step 3

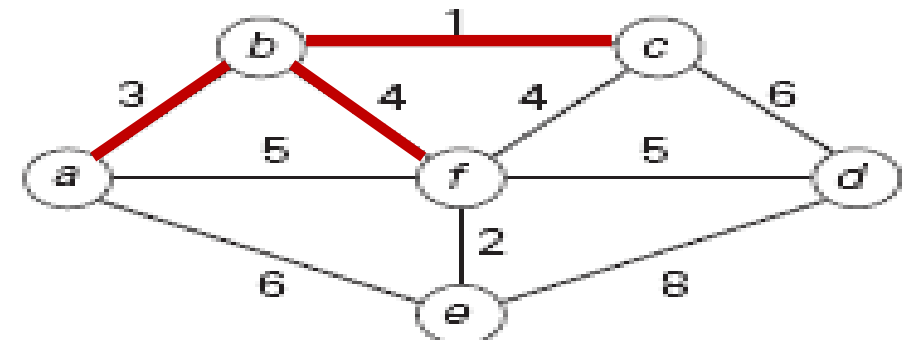
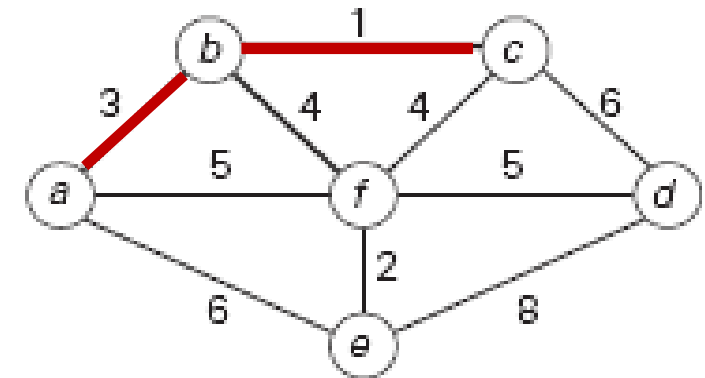
**Tree vertices**

**Remaining vertices**

**Illustration**

$d(c, 6)$   $e(a, 6)$   $f(b, 4)$

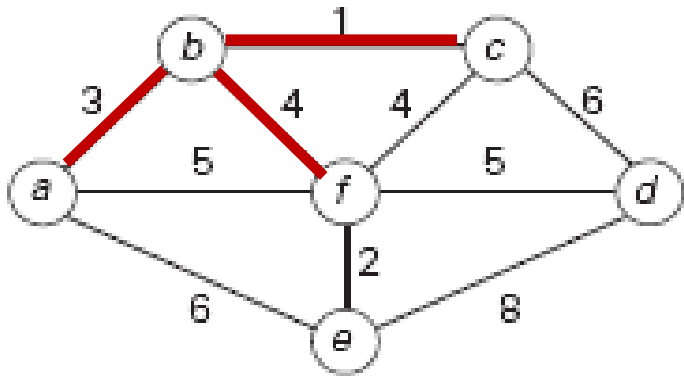
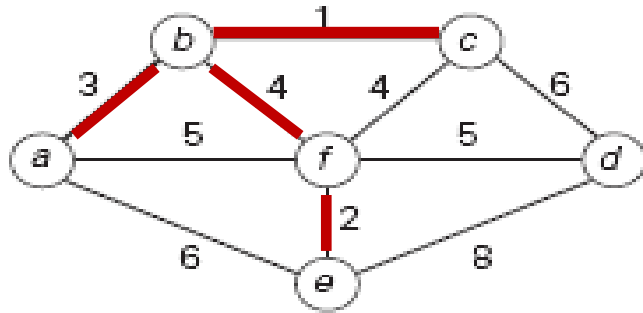
Selected Vertex  
 $f(b, 4)$





# Example 1: Prim's MST algorithm

## Step 4

Tree vertices	Remaining vertices	Illustration
	$d(f, 5) \quad e(f, 2)$	
Selected Vertex $e(f, 2)$		

# Example 1: Prim's MST algorithm

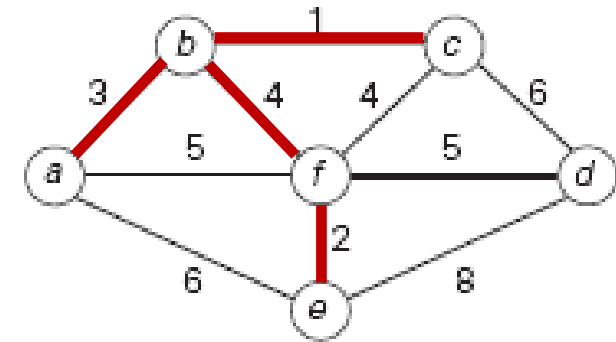
## Step 5

**Tree vertices**

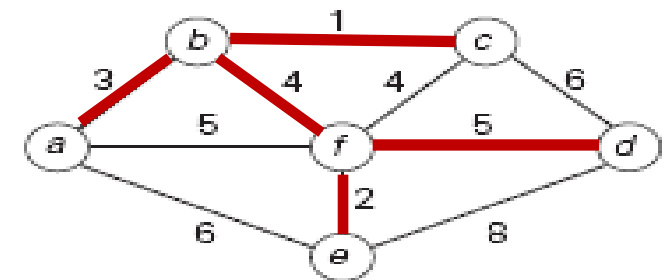
**Remaining vertices**

**Illustration**

$d(f, 5)$

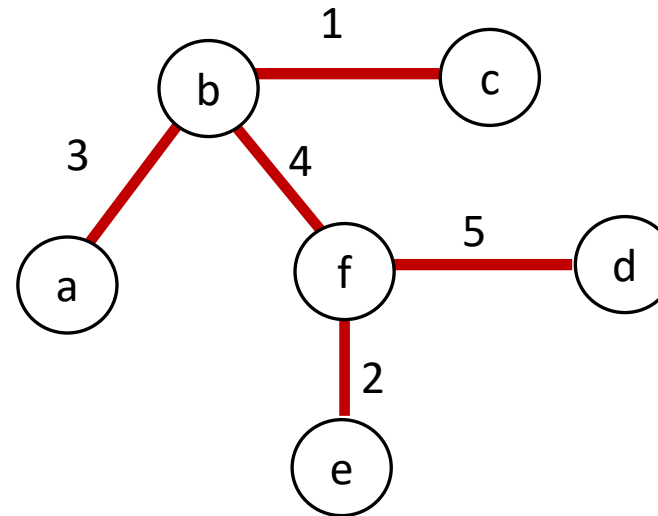


Selected Vertex  
 $d(f, 5)$



# Example 1: Prim's MST algorithm

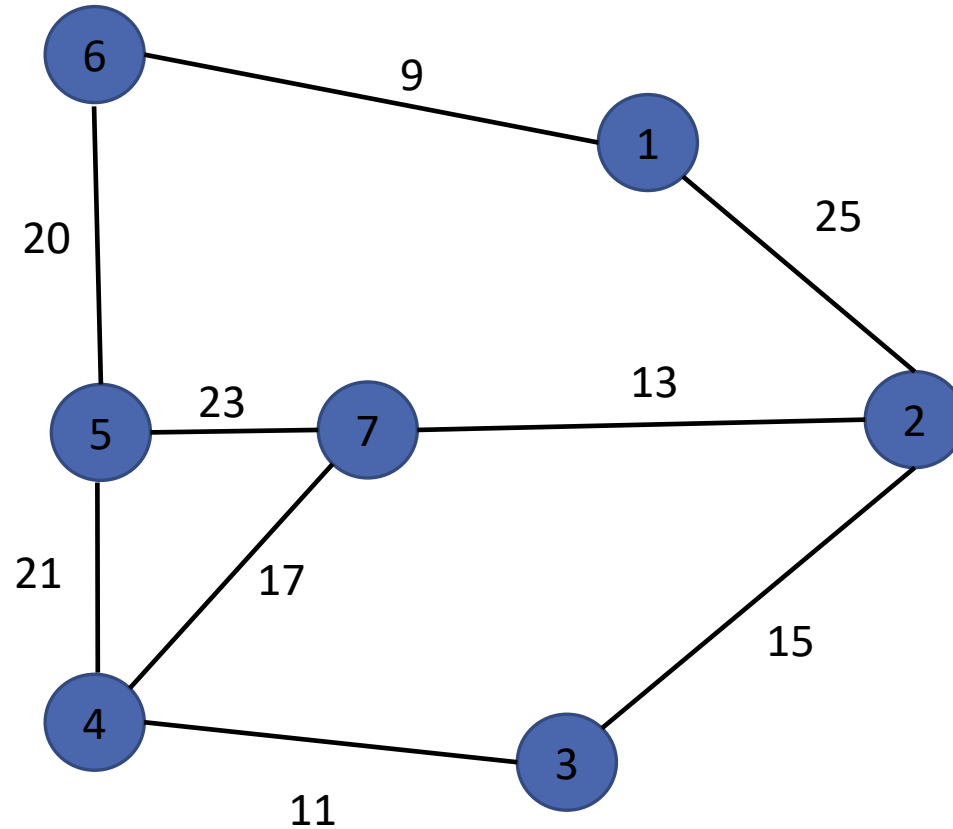
- Constructed MST:



$$W(T) = 3 + 1 + 4 + 5 + 2 = 15$$

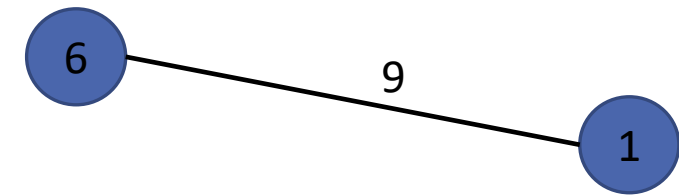
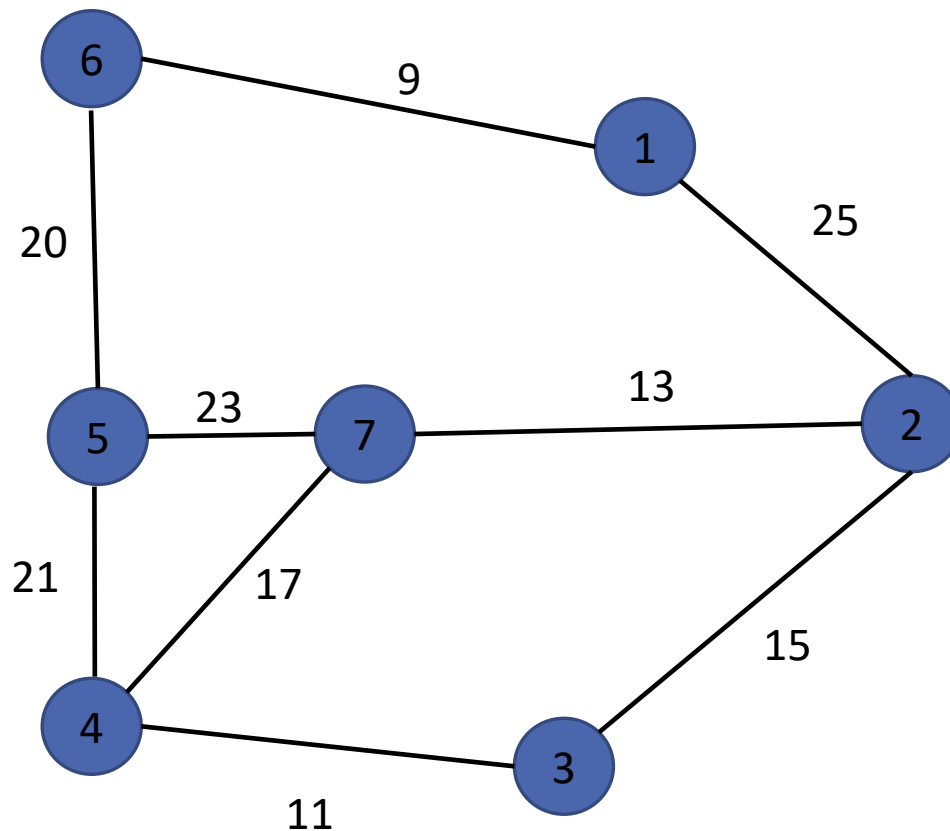
## Example 2: Prim's MST algorithm

- Construct the minimum spanning tree for following graph:



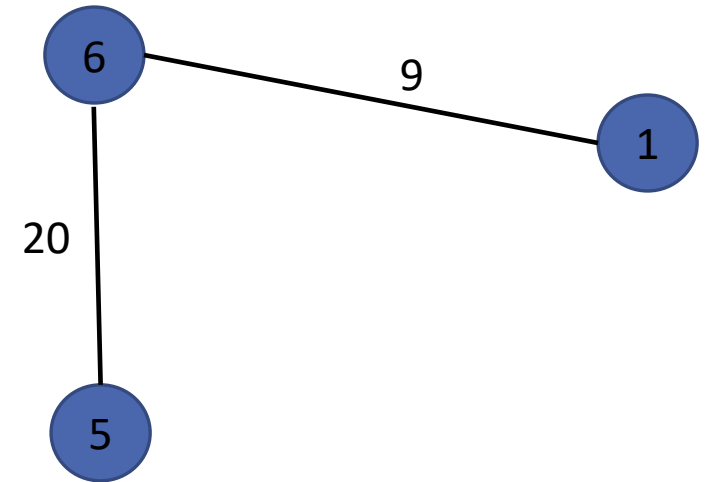
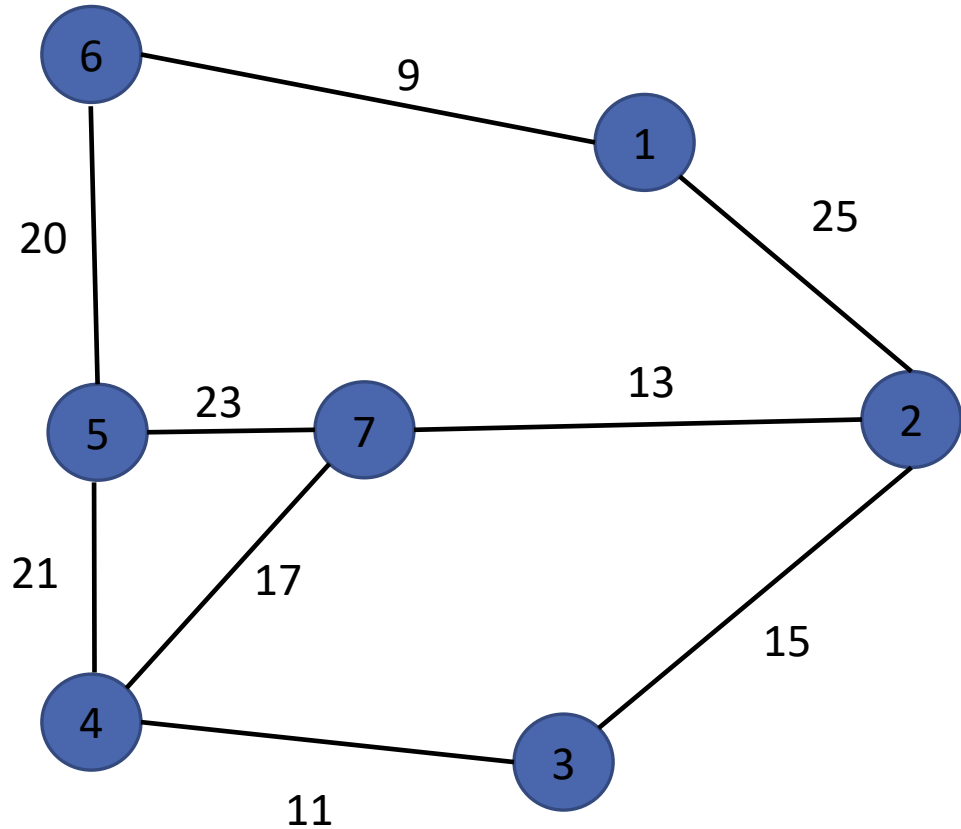
# Example 2: Prim's MST algorithm

## Step 1



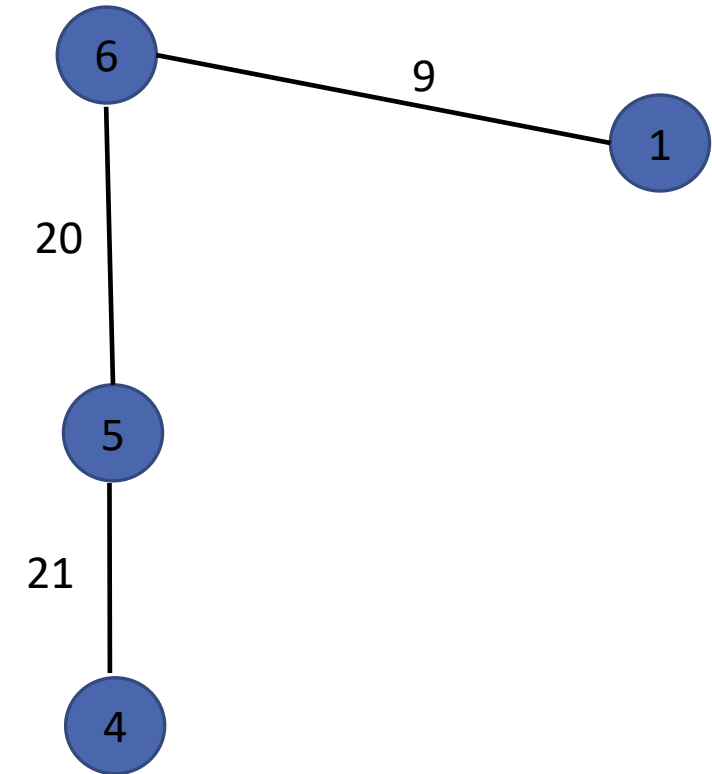
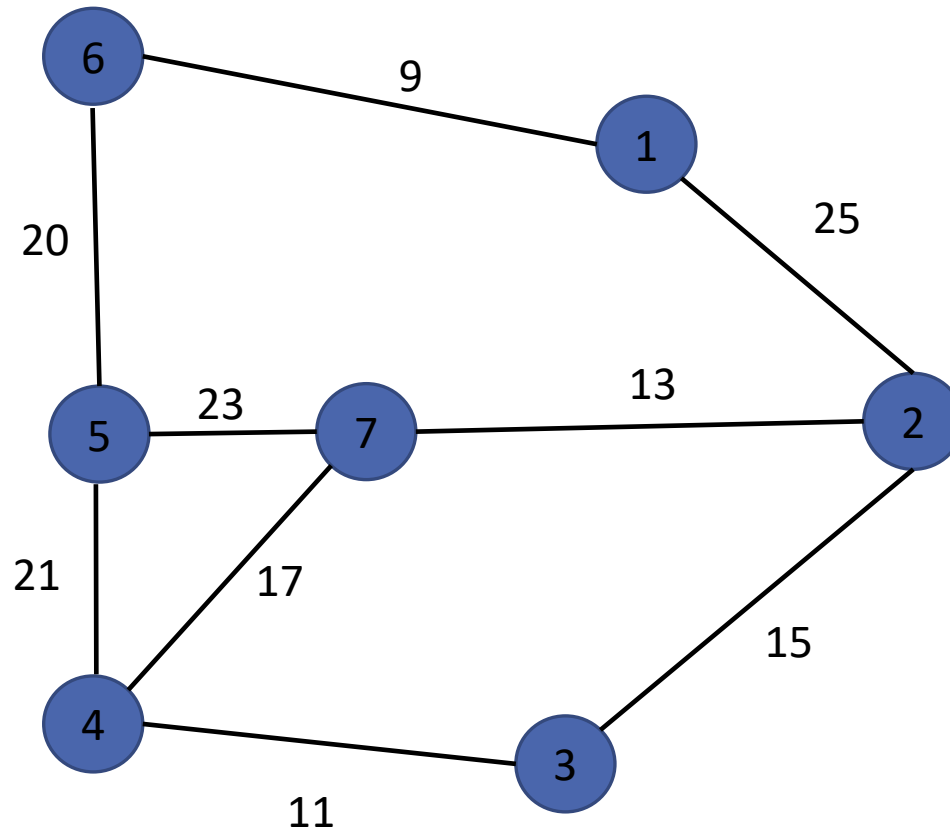
# Example 2: Prim's MST algorithm

## Step 2



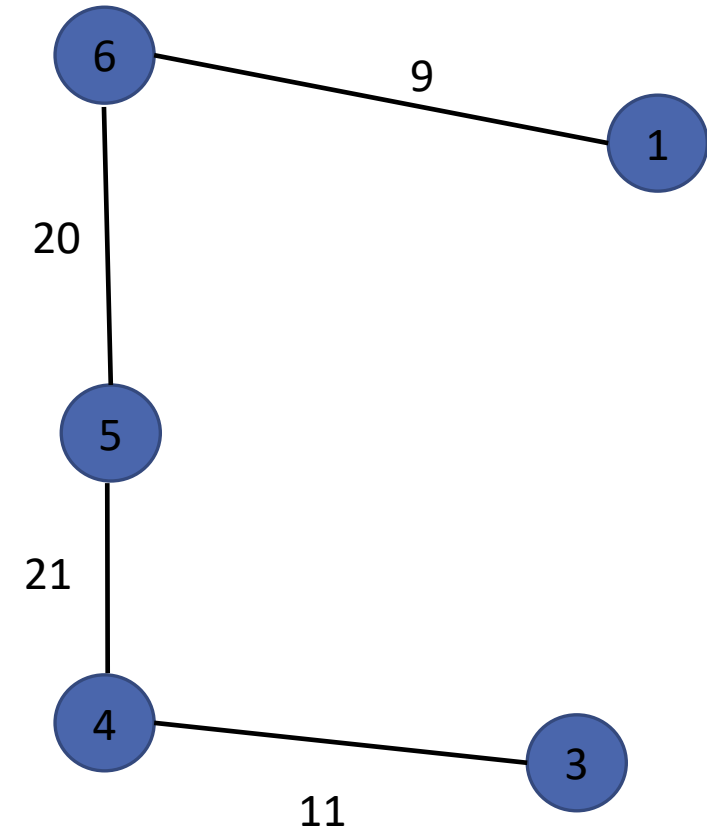
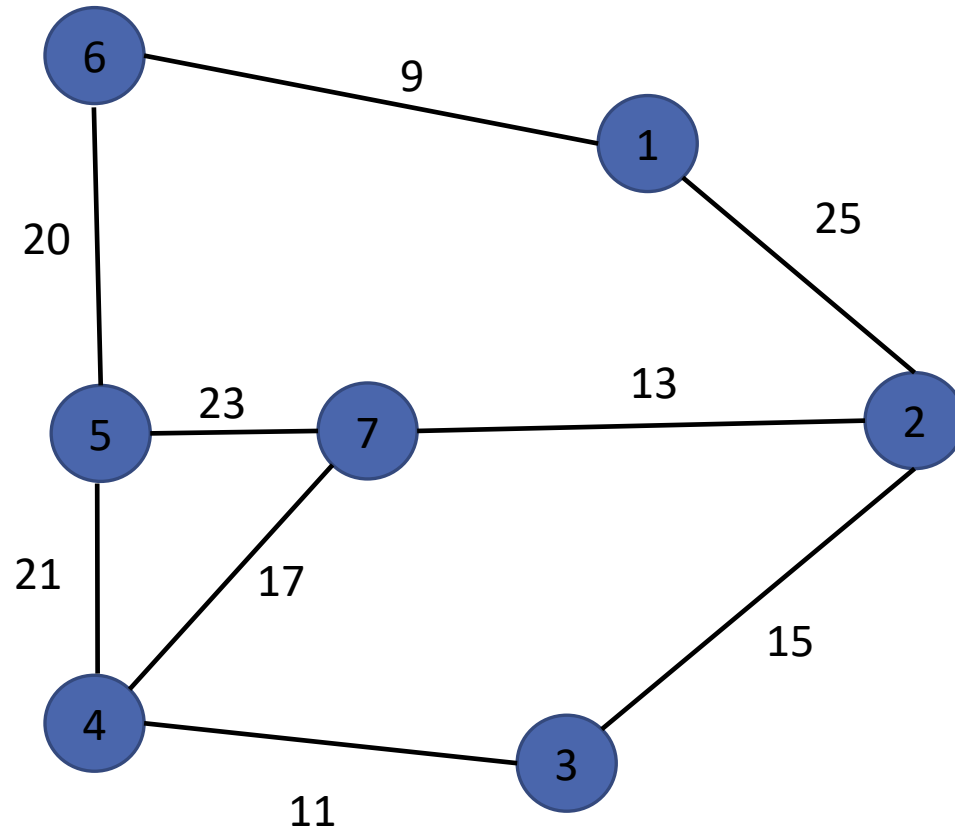
# Example 2: Prim's MST algorithm

## Step 3



## Example 2: Prim's MST algorithm

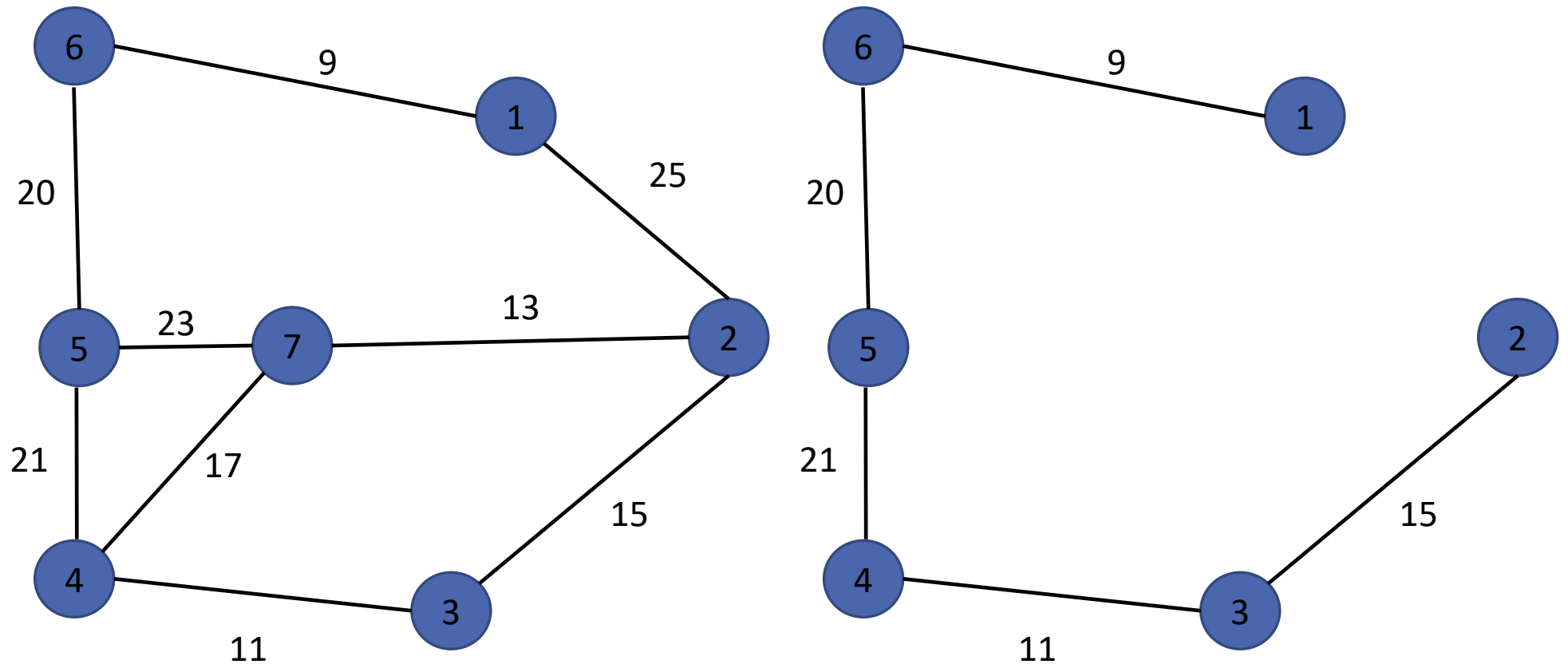
Step 4





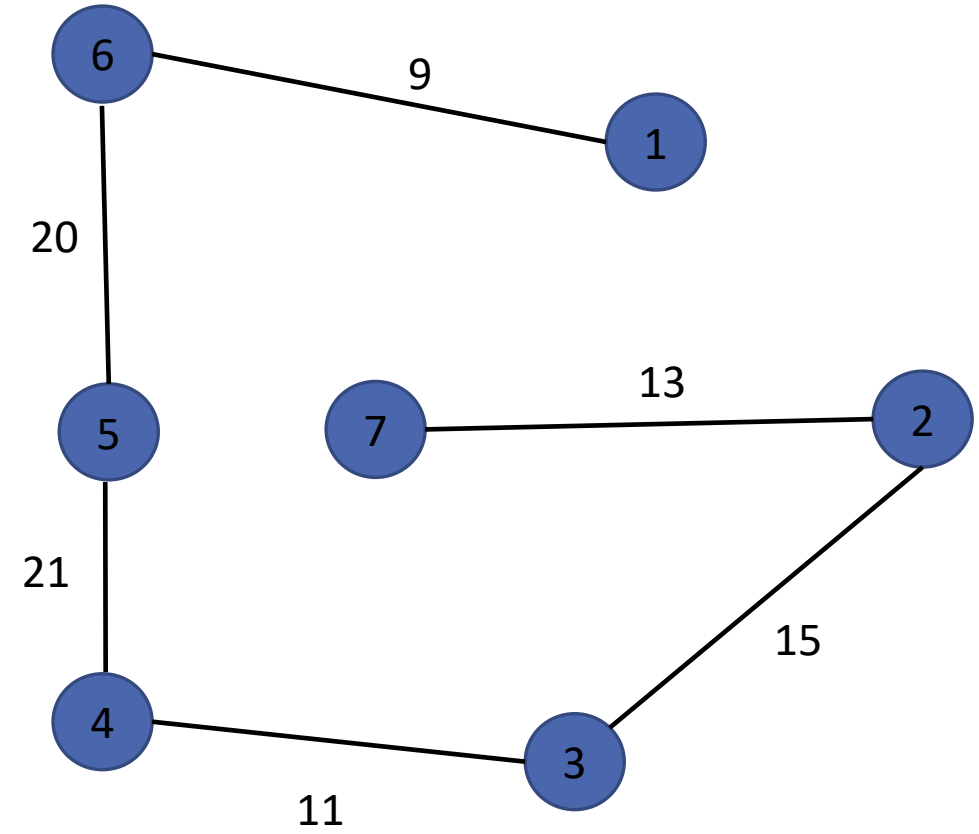
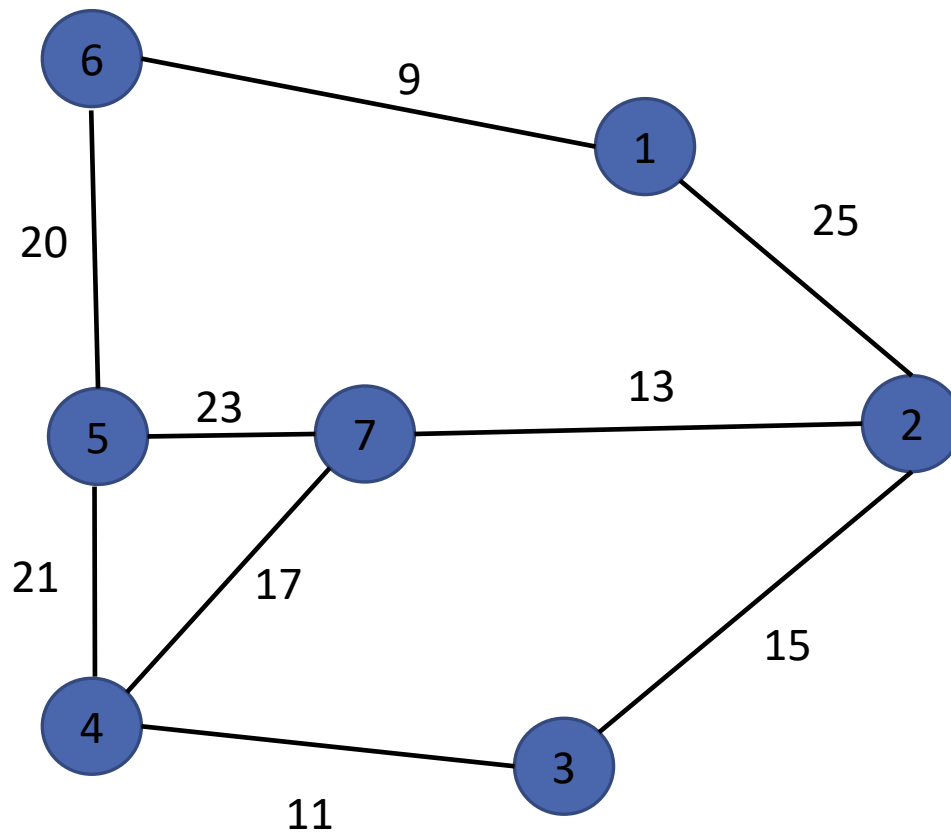
# Example 2: Prim's MST algorithm

## Step 5



# Example 2: Prim's MST algorithm

Step 6

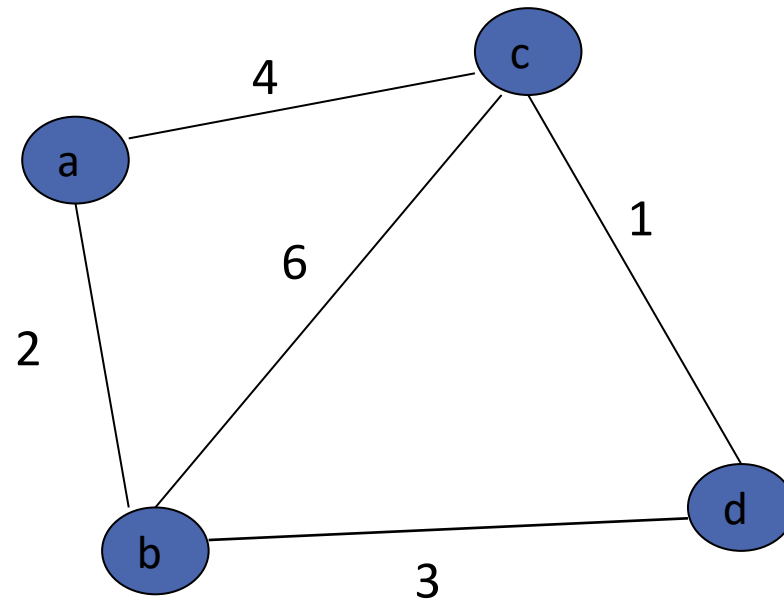


$$W(T) = 9 + 20 + 21 + 11 + 15 + 13 = 89$$

## Example 3: Prim's MST algorithm

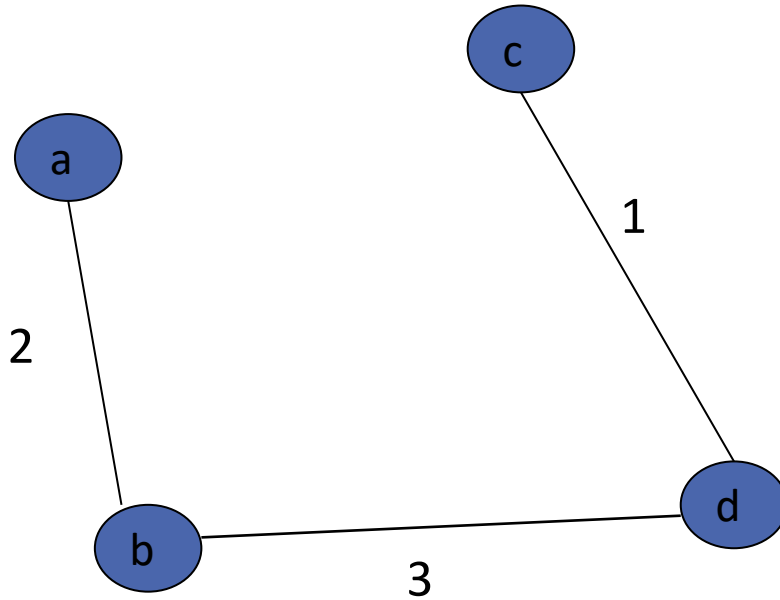
---

- Construct the minimum spanning tree for following graph:



# Example 3: Prim's MST algorithm

- Constructed MST:



$$W(T) = 1 + 2 + 3 = 6$$

# Notes about Prim's algorithm

---

- Needs priority queue for locating closest suitable vertex
- Efficiency
  - $O(n^2)$  for weight matrix representation of graph and array implementation of priority queue
  - $O(m \log n)$  for adjacency list representation of graph with  $n$  vertices and  $m$  edges and min-heap implementation of priority queue

D

# Dijkstra's algorithm

S

— : 14 min 53 sec

- - - : 15 min 30 sec

... : 15 min 48 sec

# Single-source Shortest-paths Problem

---

- ***Single-source shortest-paths problem***: for a given vertex called the ***source*** in a weighted connected graph, **find shortest paths to all its other 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.
- The obvious but probably most widely used applications are **transportation planning** and **packet routing in communication networks**, including the **Internet**.
- Multitudes of less obvious applications include finding shortest paths in **social networks**, **speech recognition**, **document formatting**, and **robotics**.
- In the world of entertainment, one can mention **pathfinding in video games** and finding best solutions to puzzles using their **state-space graphs**.

# Shortest paths – Dijkstra's algorithm

---

- There are several well-known algorithms for finding shortest paths, including **Floyd's algorithm** for the more general **all-pairs shortest-paths problem**.
- We consider the best-known algorithm for the **single-source shortest-paths problem**, called ***Dijkstra's algorithm***.
- This algorithm is applicable to **undirected and directed graphs with nonnegative weights only**.
- Since in most applications this condition is satisfied, the limitation has **not impaired** the popularity of Dijkstra's algorithm.
- Edsger W. Dijkstra (1930–2002), a noted Dutch pioneer of the science and industry of computing, discovered this algorithm in the **mid-1950s**.
- Dijkstra's algorithm finds **the shortest paths to a graph's vertices in order of their distance from a given source**.



# Shortest paths – Dijkstra's algorithm

---

- 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  $i^{\text{th}}$  iteration commences, the algorithm has already identified the shortest paths to  $i - 1$  other vertices nearest to the source.
- These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree  $T_i$  of the given graph.
- Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of  $T_i$ .
- The set of vertices adjacent to the vertices in  $T_i$  can be referred to as “fringe vertices”; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.
- To identify the  $i^{\text{th}}$  nearest vertex, the algorithm computes, for every fringe vertex  $u$ , the sum of the distance to the nearest tree vertex  $v$  (given by the weight of the edge  $(v, u)$ ) and the length  $d_v$  of the shortest path from the source to  $v$  and then selects the vertex with the smallest such sum.

# Shortest paths – Dijkstra's algorithm

---

- To facilitate the algorithm's operations, we **label** each vertex with **two labels**.
- **The numeric label**  $d$  indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree,  $d$  indicates the length of the shortest path from the source to that vertex.
- **The other label** indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed.
- With such labeling, finding the next nearest vertex  $u^*$  becomes a simple task of finding a fringe vertex with the smallest  $d$  value. Ties can be broken arbitrarily.

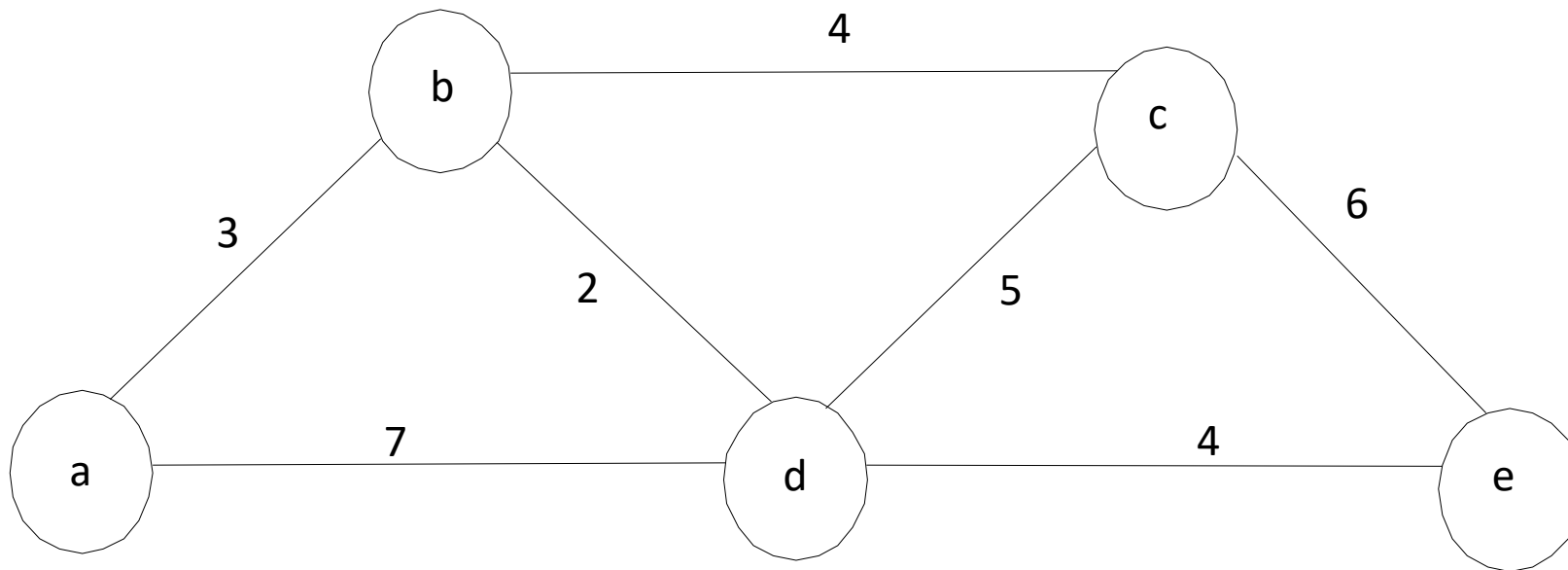
# Shortest paths – Dijkstra's algorithm

---

- After we have identified a vertex  $u^*$  to be added to the tree, we need to perform two operations:
  - 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
    - $d_{u^*} + w(u^*, u) < d_u$ ,
    - update the labels of  $u$  by  $u^*$  and  $d_{u^*} + w(u^*, u)$ , respectively.

# Example 1: Dijkstra's Algorithm

- Solve the single-source shortest-paths problem in the following graph with vertex "a" as the source:



# Example 1: Dijkstra's Algorithm

## Step 1

**Tree vertices**

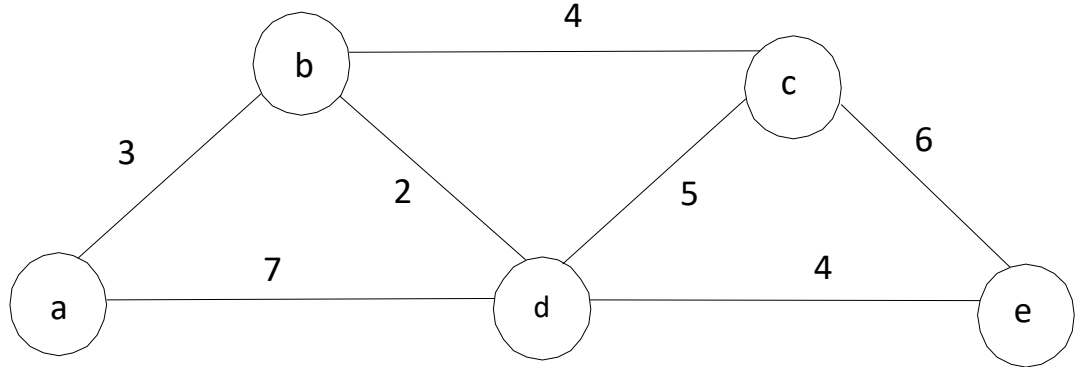
**Remaining vertices**

**Illustration**

Source Vertex

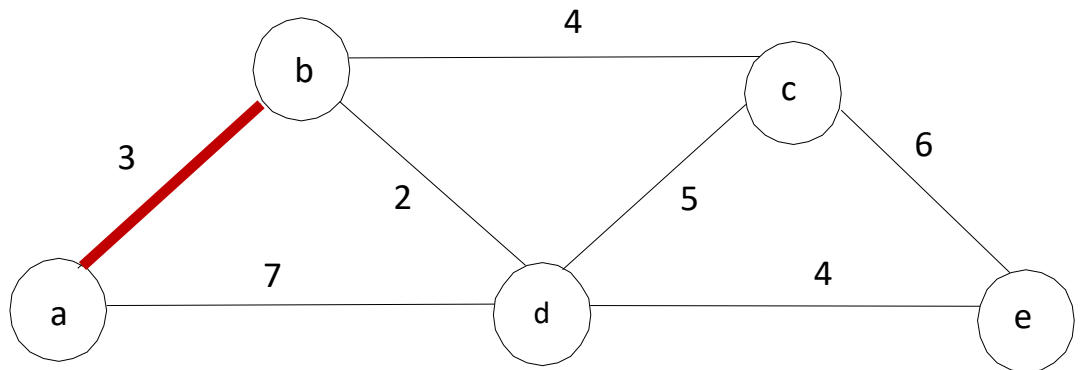
$a(-,0)$

$b(a,3)$   $c(-,\infty)$   $d(a,7)$   $e(-,\infty)$



Selected Vertex

$b(a,3)$



# Example 1: Dijkstra's Algorithm

Step 2

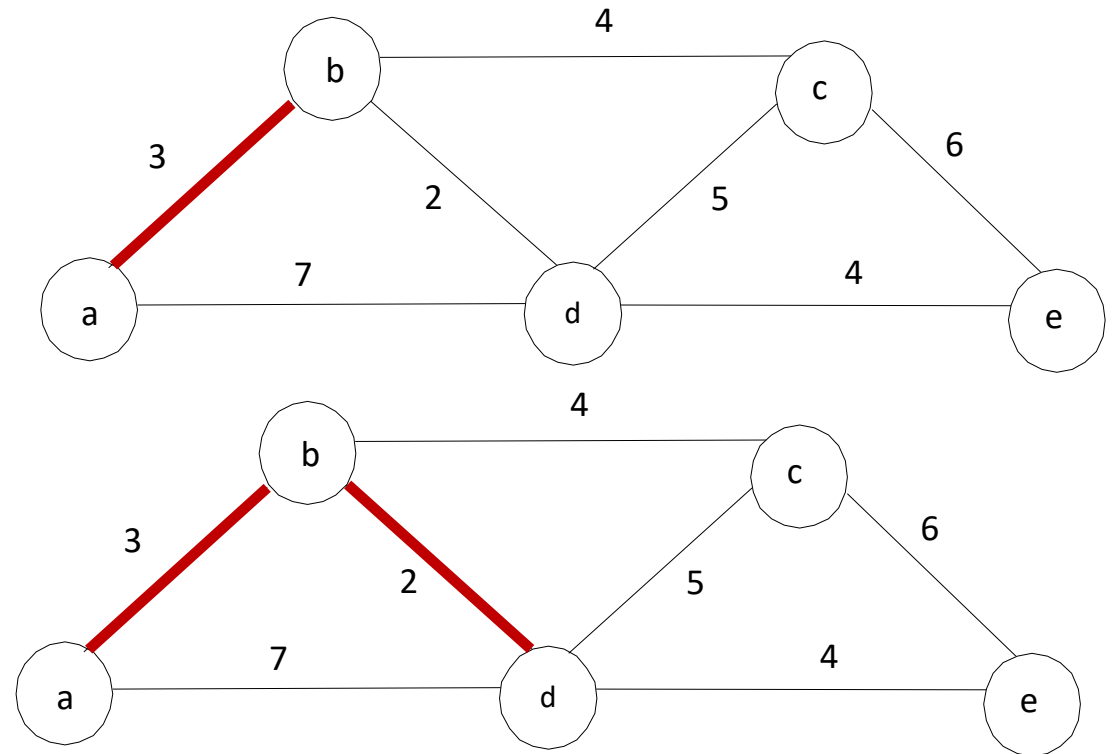
Tree vertices

Remaining vertices

Illustration

$c(b, 3+4)$   $d(b, 3+2)$   $e(-, \infty)$

Selected Vertex  
 $d(b, 5)$



Step 3

# Example 1: Dijkstra's Algorithm

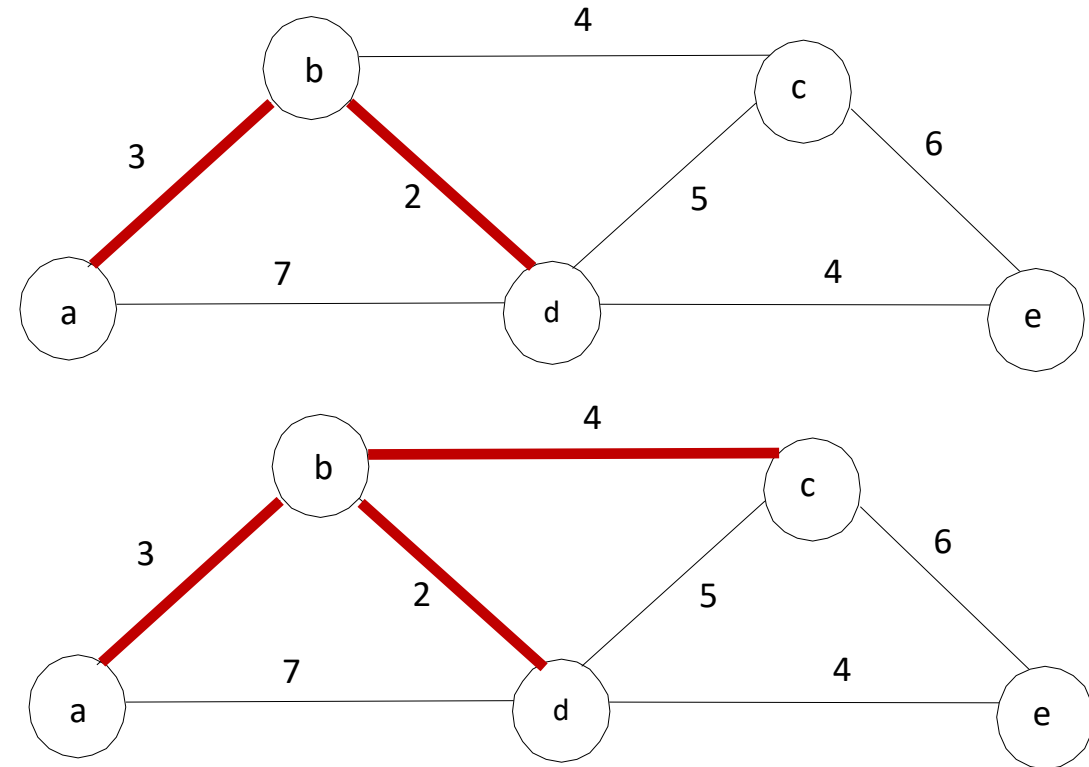
Tree vertices

Remaining vertices

Illustration

c(b,7) e(d,5+4)

Selected Vertex  
c(b,7)



# Example 1: Dijkstra's Algorithm

Step 4

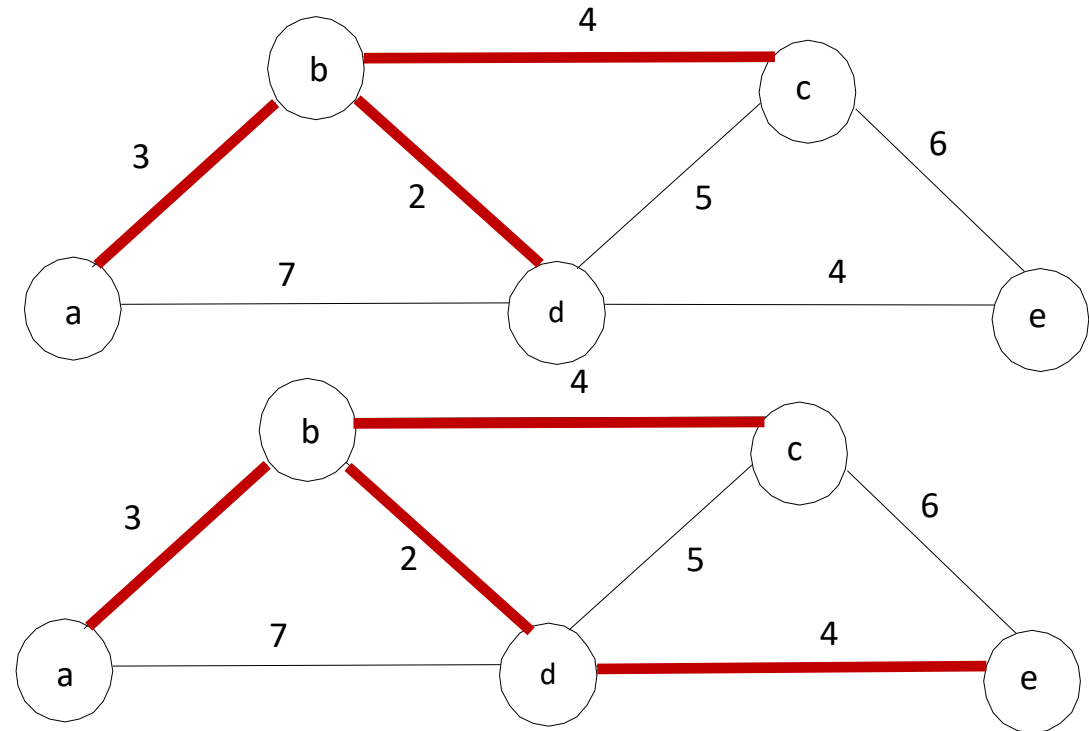
Tree vertices

Remaining vertices

Illustration

$e(d,9)$

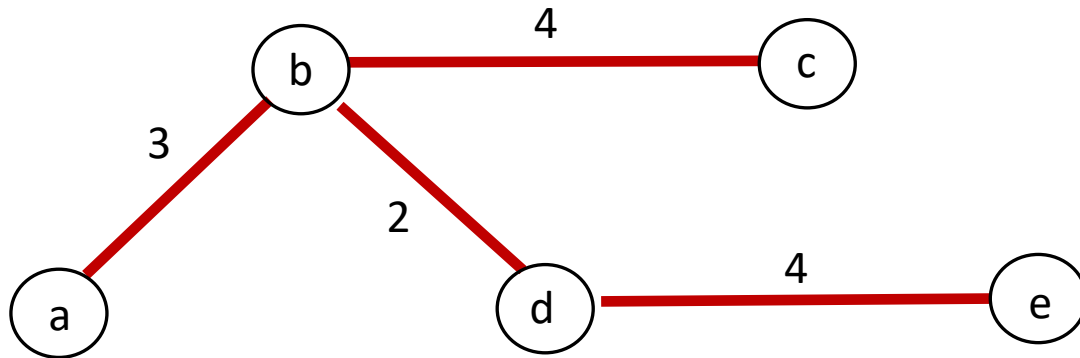
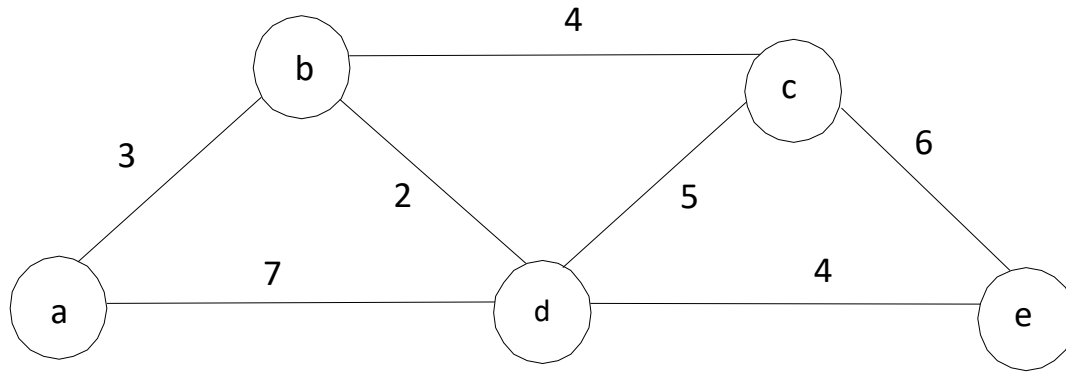
Selected Vertex  
 $e(d,9)$





# Example 1: Dijkstra's Algorithm

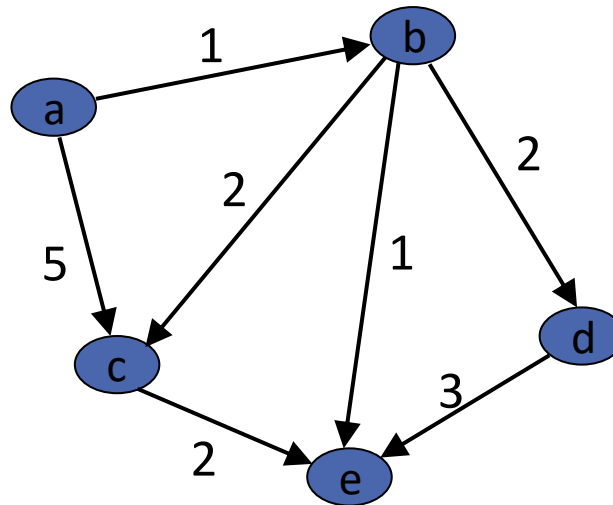
---



From a to b: a-b of Length 3  
From a to d: a-b-d of Length 5  
From a to c: a-b-c of Length 7  
From a to e: a-b-d-e of Length 9

## Example 2: Dijkstra's Algorithm-Directed Graph

- Solve the single-source shortest-paths problem in the following graph with vertex "a" as the source: :

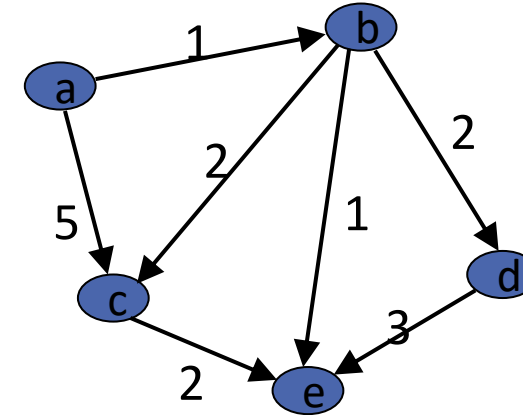


# Example 2: Dijkstra's Algorithm-Directed Graph

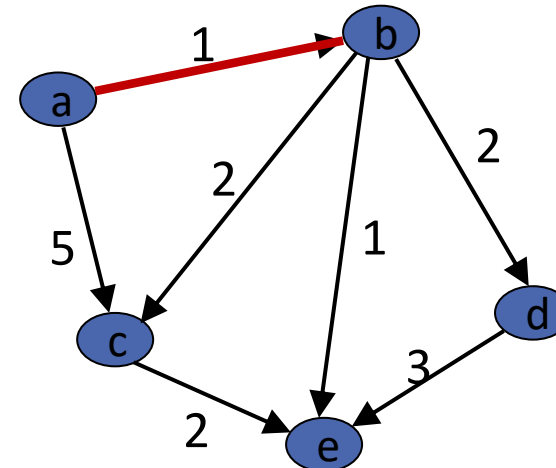
## Step 1

$a(-,0)$

$b(a,1)$   $c(a,5)$   $d(-,\infty)$   $e(-,\infty)$



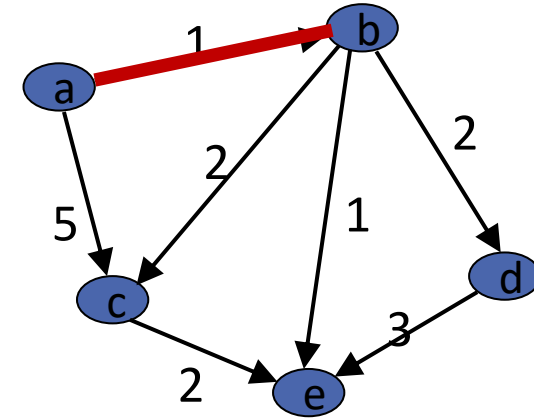
Selected Vertex  
 $b(a,1)$



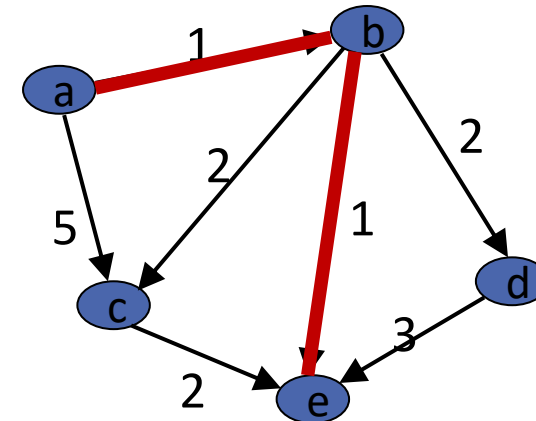
# Example 2: Dijkstra's Algorithm-Directed Graph

## Step 2

$c(a, 2+1)$   $d(b, 2+1)$   $e(b, 1+1)$



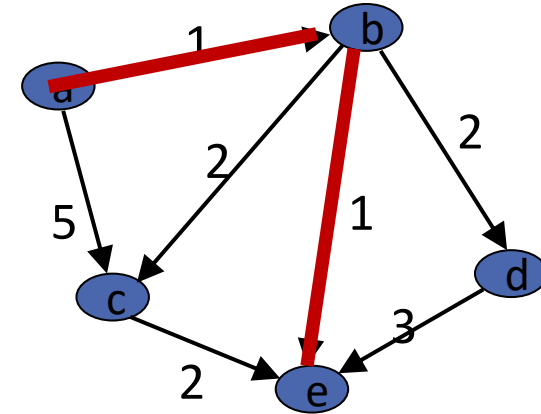
Selected Vertex  
 $e(b, 2)$



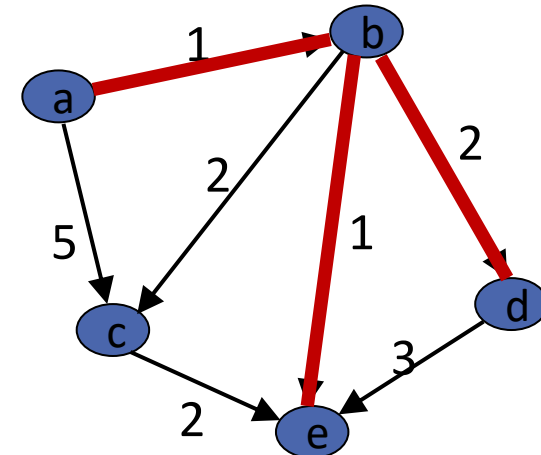
# Example 2: Dijkstra's Algorithm-Directed Graph

## Step 3

$c(a,3)$   $d(b,2+1)$



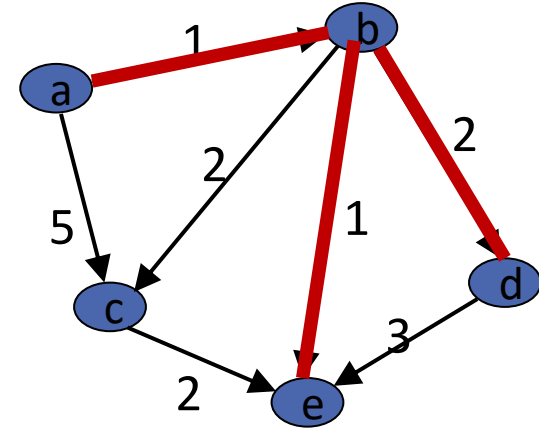
Selected Vertex  
 $d(b,3)$



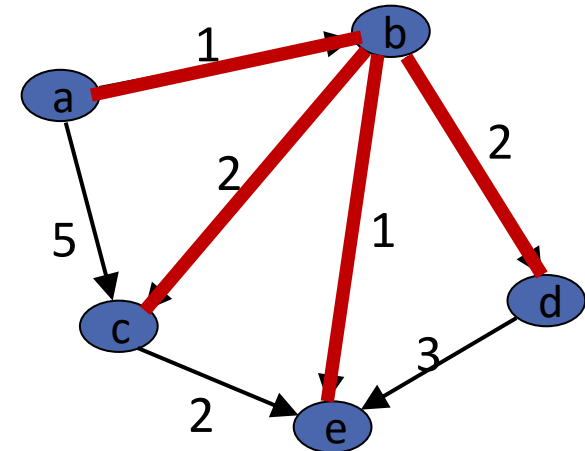
# Example 2: Dijkstra's Algorithm-Directed Graph

## Step 4

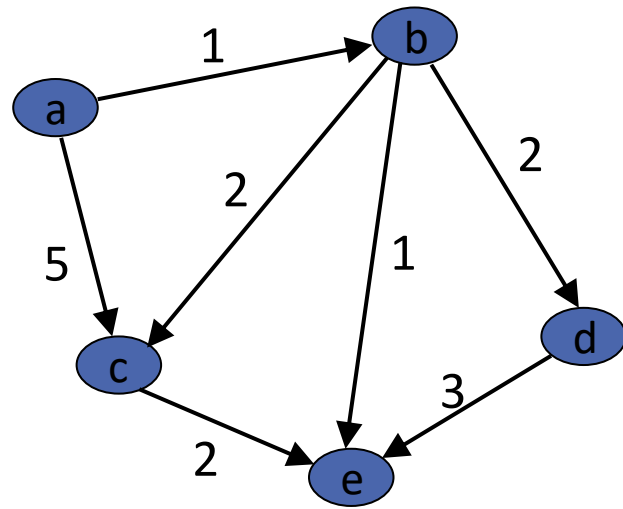
$c(b,3)$



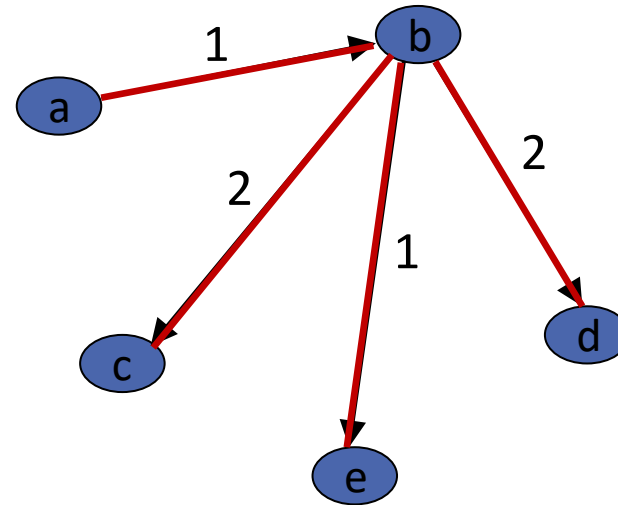
Selected Vertex  
 $c(b,3)$



## Example 2: Dijkstra's Algorithm-Directed Graph



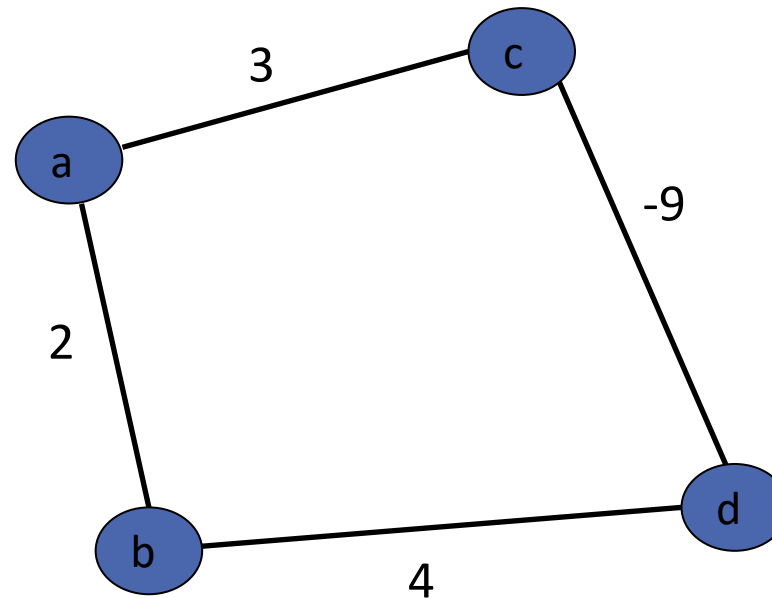
From a to b: a-b  
From a to c: a-b-c  
From a to e: a-b-e  
From a to d: a-b-d



of Length 1  
of Length 3  
of Length 2  
of Length 3

## Example 3: Dijkstra's Algorithm-Negative Edge

- Solve the single-source shortest-paths problem in the following graph with vertex "a" as the source:



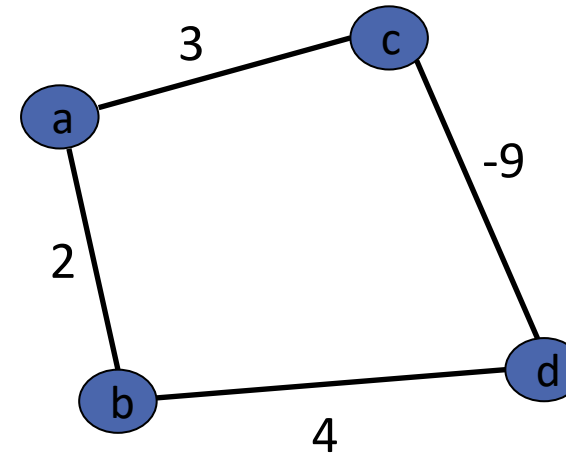


# Example 3: Dijkstra's Algorithm-Negative Edge

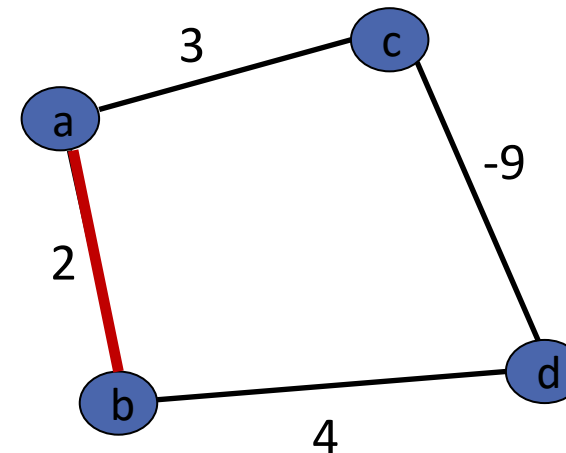
## Step 1

$a(-,0)$

$b(a,2)$   $c(a,3)$   $d(-,\infty)$



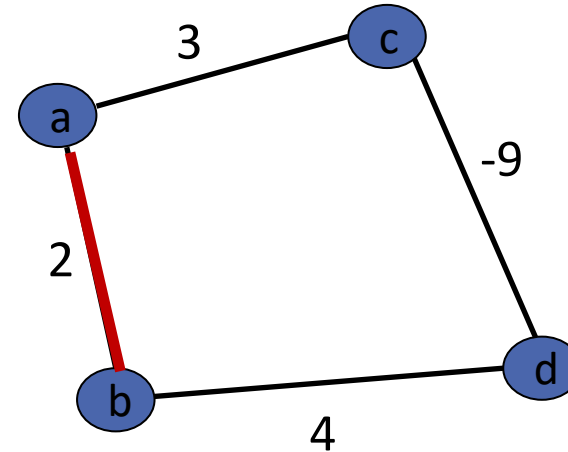
Selected Vertex  
 $b(a,2)$



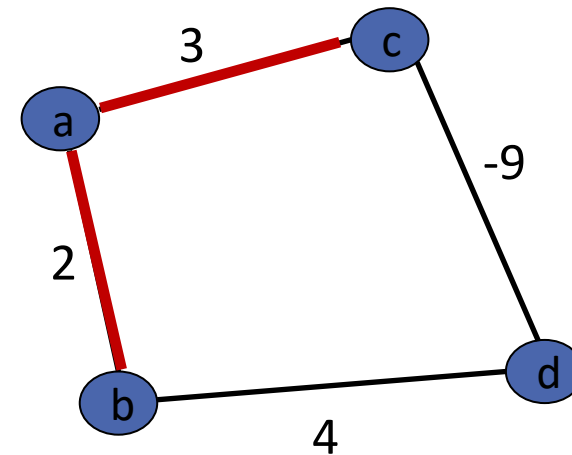
# Example 3: Dijkstra's Algorithm-Negative Edge

## Step 2

$c(a,3)$   $d(b,4 + 2)$



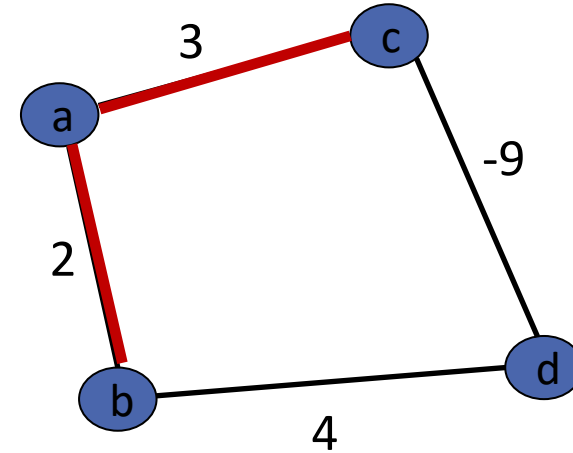
Selected Vertex  
 $c(a,3)$



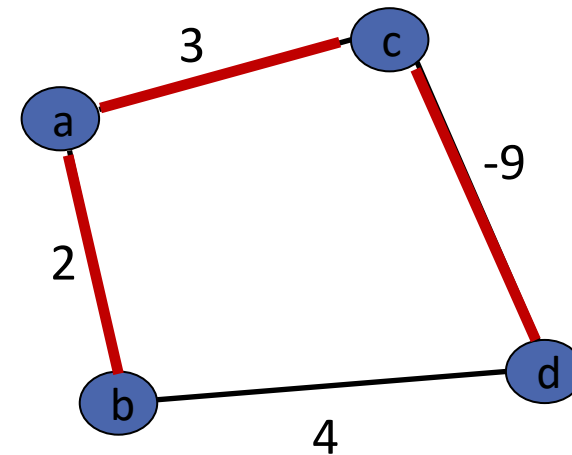
# Example 3: Dijkstra's Algorithm-Negative Edge

## Step 3

$d(c, -9+3)$

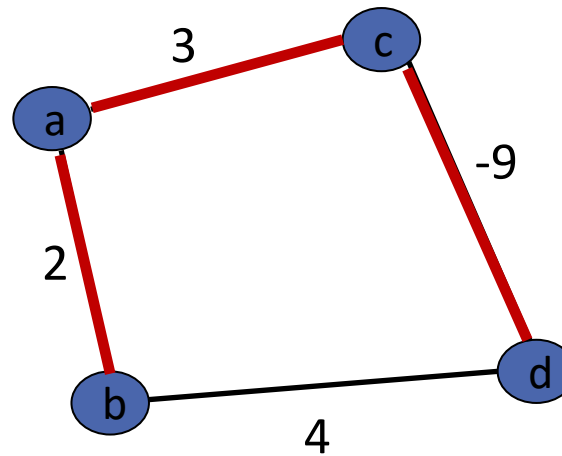


Selected Vertex  
 $d(c, -6)$



## Example 3: Dijkstra's Algorithm-Negative Edge

- Dijkstra's algorithm can not be applied to a graph with negative edge(s).



From a to b: a-b of Length 2

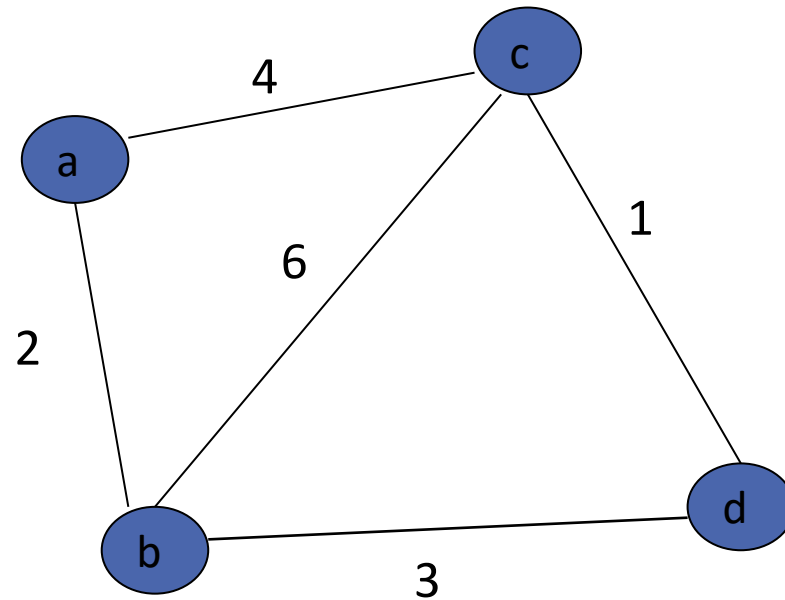
?

From a to b: a-c-d-b of Length -2

## Example 4: Dijkstra's Algorithm

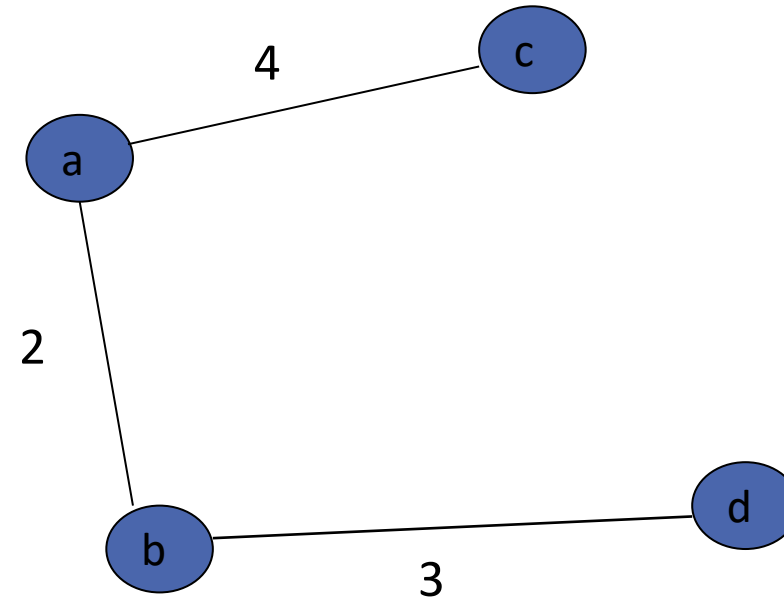
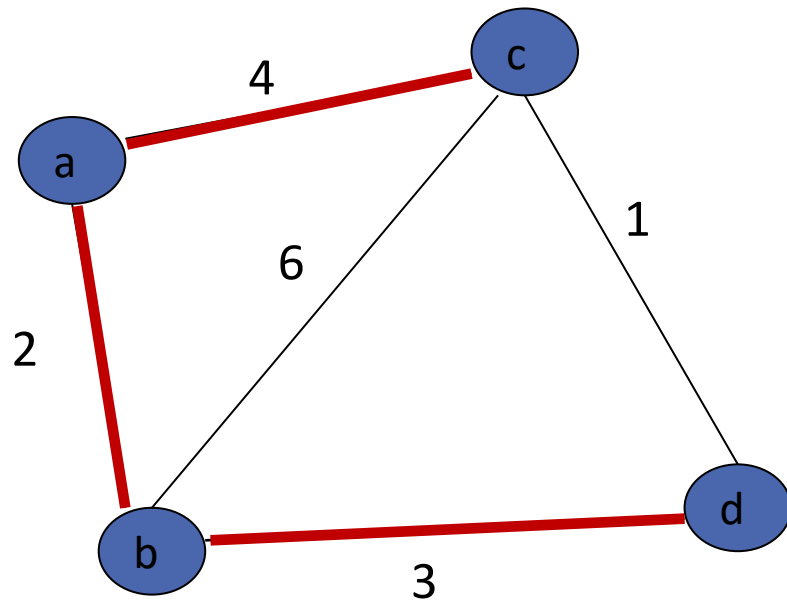
---

- Solve the single-source shortest-paths problem in the following graph with vertex "a" as the source:



## Example 4: Dijkstra's Algorithm

---



# Pseudocode of Dijkstra's algorithm

---

**ALGORITHM** *Dijkstra*( $G, s$ )

//Dijkstra's algorithm for single-source shortest paths

//Input: A weighted connected graph  $G = \langle V, E \rangle$  with nonnegative weights

// and its vertex  $s$

//Output: The length  $d_v$  of a shortest path from  $s$  to  $v$

// and its penultimate vertex  $p_v$  for every vertex  $v$  in  $V$

*Initialize*( $Q$ ) //initialize priority queue to empty

**for** every vertex  $v$  in  $V$

$d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$

*Insert*( $Q, v, d_v$ ) //initialize vertex priority in the priority queue

$d_s \leftarrow 0$ ; *Decrease*( $Q, s, d_s$ ) //update priority of  $s$  with  $d_s$

$V_T \leftarrow \emptyset$

**for**  $i \leftarrow 0$  **to**  $|V| - 1$  **do**

$u^* \leftarrow \text{DeleteMin}(Q)$  //delete the minimum priority element

$V_T \leftarrow V_T \cup \{u^*\}$

**for** every vertex  $u$  in  $V - V_T$  that is adjacent to  $u^*$  **do**

**if**  $d_{u^*} + w(u^*, u) < d_u$

$d_u \leftarrow d_{u^*} + w(u^*, u)$ ;  $p_u \leftarrow u^*$

*Decrease*( $Q, u, d_u$ )

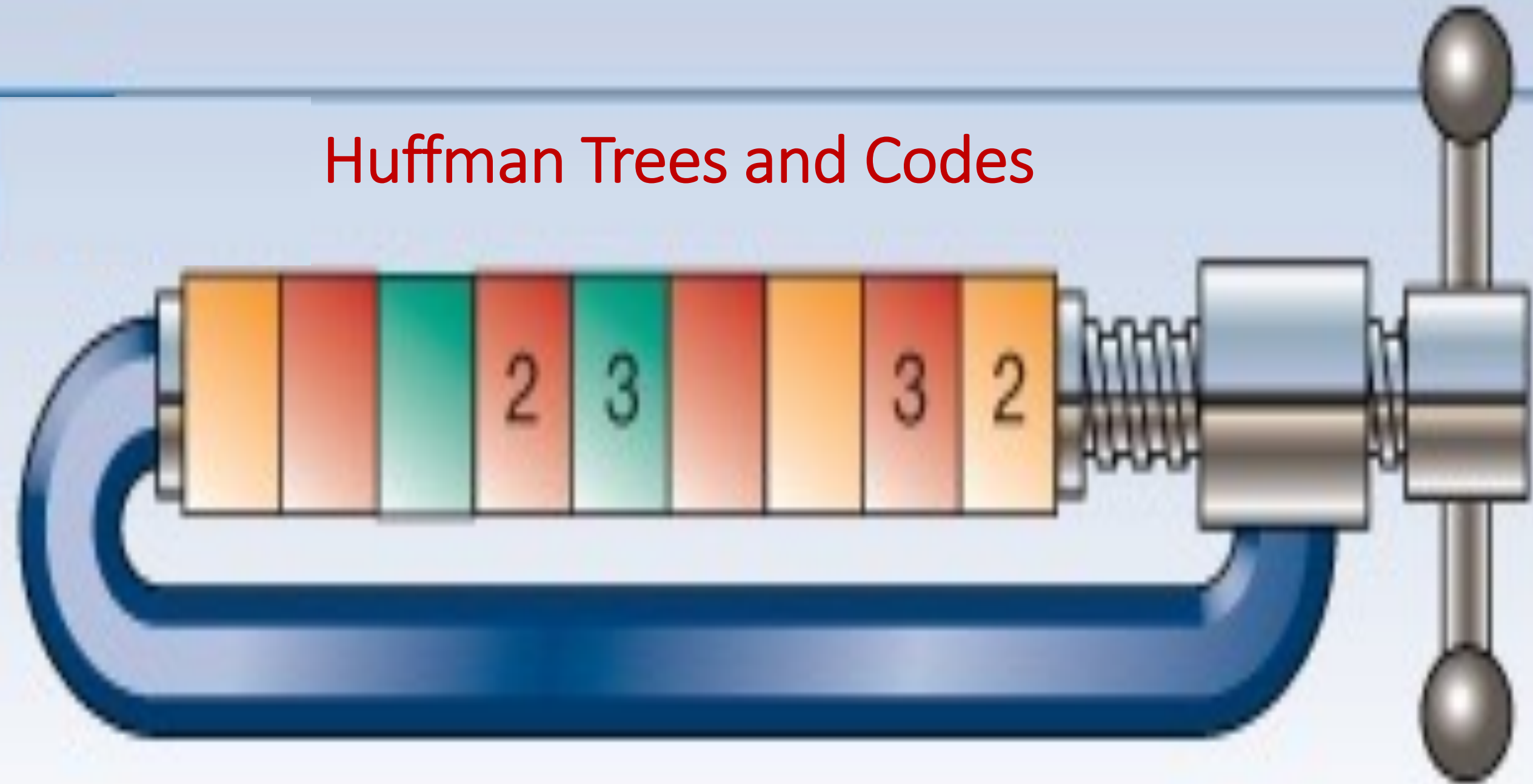
# Analysis of Dijkstra's algorithm

---

- The time efficiency of Dijkstra's algorithm depends on the data structures used for **implementing the priority queue** and for representing an input graph itself.
- It is in  $\Theta(|V|^2)$  for graphs represented by their weight matrix and the **priority queue implemented as an unordered array**.
- For graphs represented by their adjacency lists and the priority queue **implemented as a min-heap**, it is in  $O(|E| \log |V|)$ .



# Huffman Trees and Codes



# Coding Problem

---

○ Message: A B B A C C D A A B E

Characters	ASCII	8-bit	3-bit
A	65	01000001	000
B	66	01000010	001
C	67	01000011	010
D	68	01000100	011
E	69	01000101	100

○ Total for 8-bit:  $10 * 8 = 80$

○ Total for 3-bit:  $10 * 3 = 30$

# Coding Problem

---

- Suppose we have to **encode** a text that comprises symbols from some  $n$ -symbol alphabet by assigning to each of the text's symbols some sequence of bits called the **codeword**.
- We can use a **fixed-length encoding** that assigns to each symbol a bit string of the same length  $m$  ( $m \geq \log_2 n$ ).
- This is exactly what the standard ASCII code does.

# Coding Problem

---

- One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of **assigning shorter codewords to more frequent symbols** and **longer codewords to less frequent symbols**.
- This idea was used, in particular, in the **telegraph code** invented in the mid-19th century by Samuel Morse.
- In that code, **frequent letters** such as e (.) and a (.-) are assigned short sequences of dots and dashes while **infrequent letters** such as q (- - .-) and z (- - ..) have longer ones.
- ***Variable-length encoding***, which assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding does not have.

# Coding Problem

---

- If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's symbols with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1.
- The codeword of a symbol can then be obtained by recording the labels on the simple path from the root to the symbol's leaf.
- Among the many trees that can be constructed in this manner for a given alphabet with known frequencies of the symbol occurrences, how can we construct a tree that would assign shorter bit strings to high-frequency symbols and longer ones to low-frequency symbols?
- It can be done by the following greedy algorithm, invented by David Huffman while he was a graduate student at MIT.

# Huffman codes

---

- **Huffman's algorithm**
- **Step 1** : Initialize  $n$  one-node trees and label them with the symbols of the alphabet given.
- Record the frequency of each symbol in its tree's root to indicate the tree's **weight**. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)
- **Step 2** : Repeat the following operation until a single tree is obtained.
- Find two trees with the smallest weight.
- Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

# Huffman codes

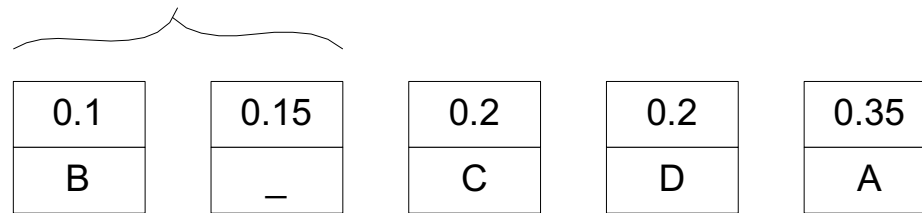
---

- A tree constructed by the above algorithm is called a ***Huffman tree***. It defines—in the manner described above—a ***Huffman code***.
- A ***Huffman tree*** is a **binary tree** that minimizes the weighted path length from the root to the leaves of predefined weights.
- The most important application of Huffman trees is **Huffman codes**.
- A ***Huffman code*** is an optimal prefix-free variable-length encoding scheme that assigns bit strings to symbols based on their frequencies in a given text.
- This is accomplished by a greedy construction of a binary tree whose leaves represent the alphabet symbols and whose edges are labeled with 0's and 1's.

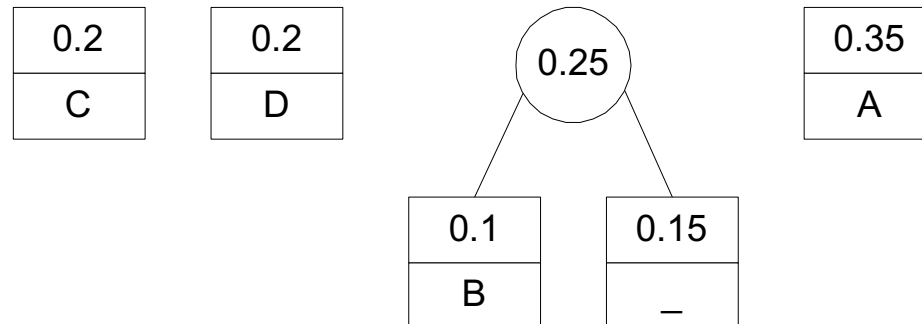
# Example 1: Huffman Coding

## Step 1

character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15



- 1- Order according to the frequencies
- 2- Select minimum two frequencies and create a tree

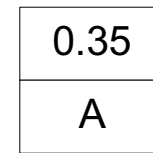
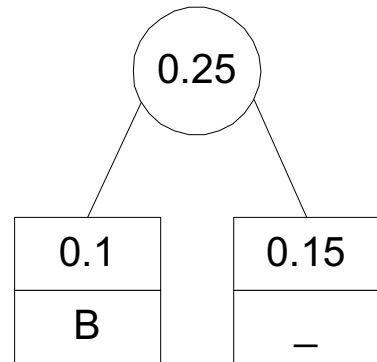
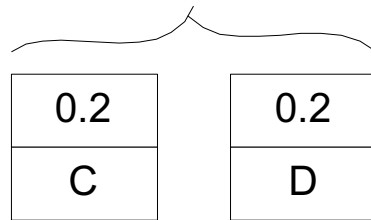


- 3- Reorder according to the frequencies

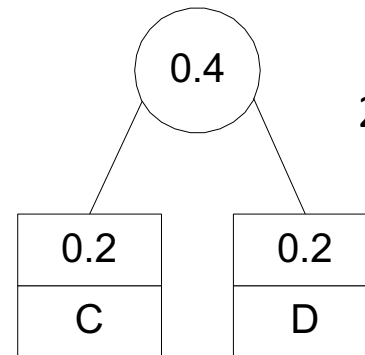
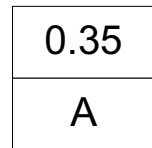
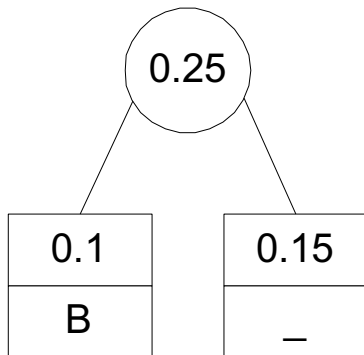


# Example 1: Huffman Coding

Step 2



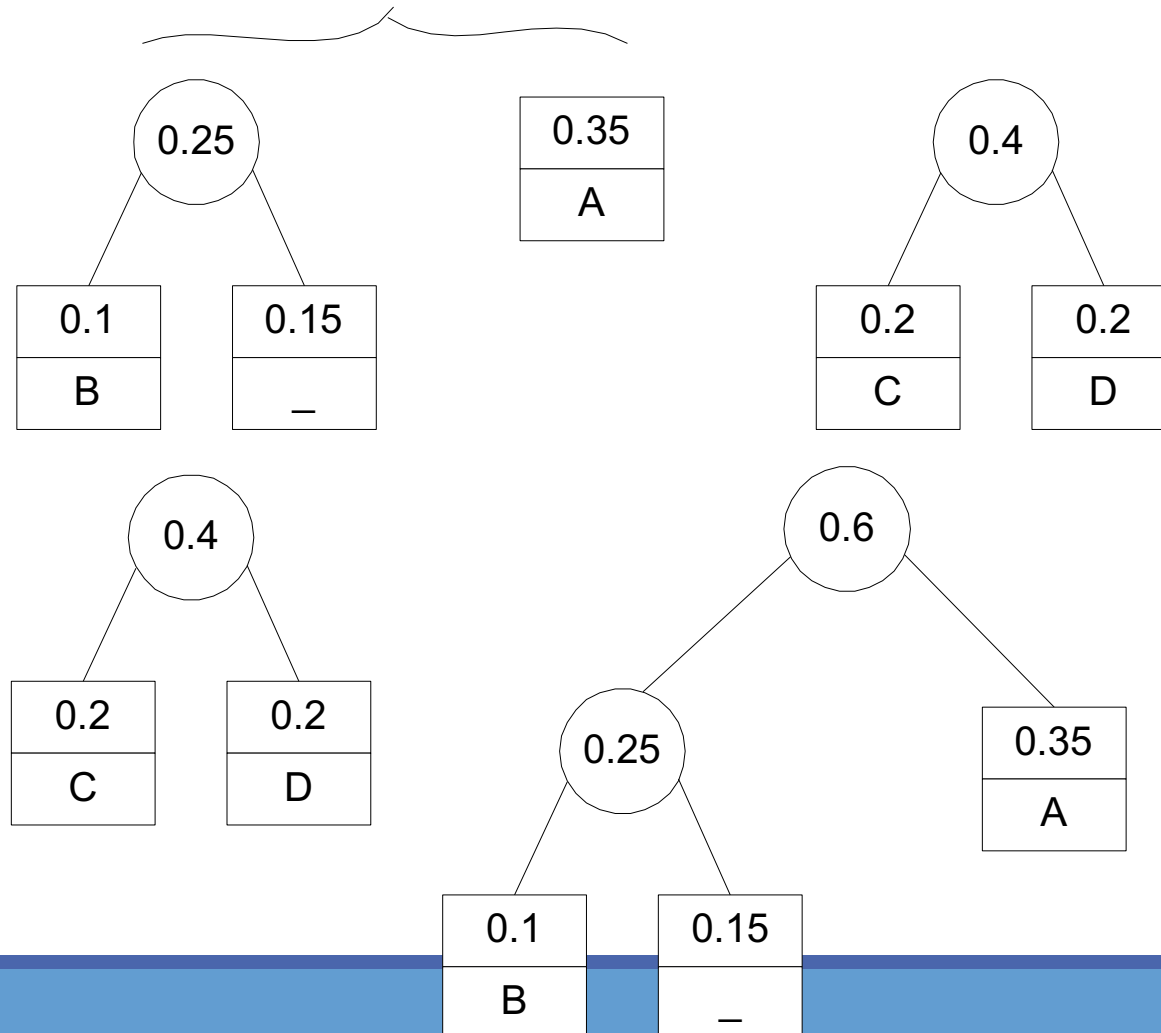
1- Select minimum two frequencies and create a tree



2- Reorder according to the frequencies

# Example 1: Huffman Coding

Step 3

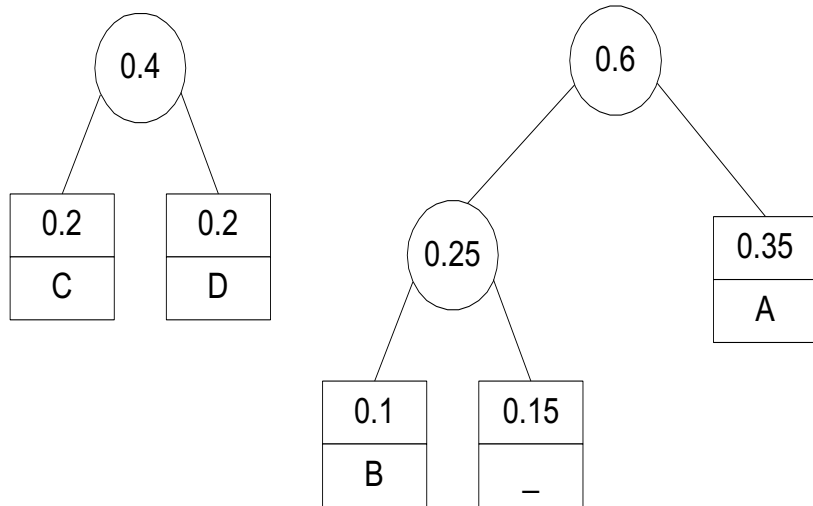


1- Select minimum two frequencies and create a tree

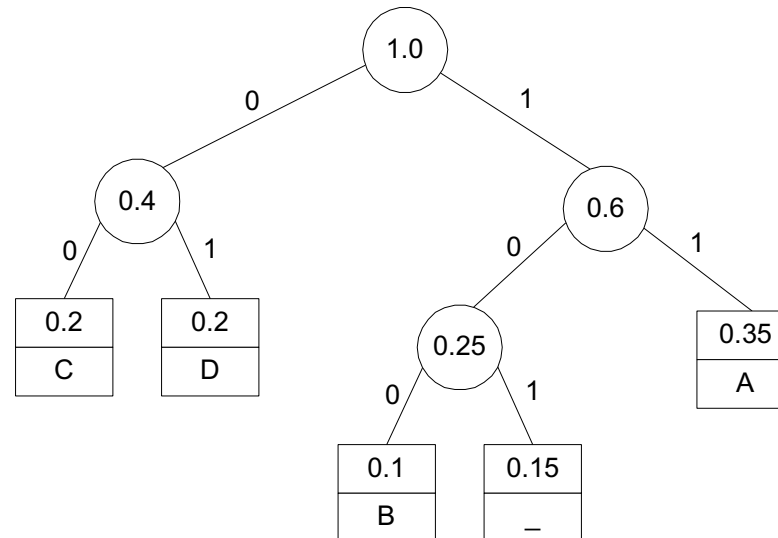
2- Reorder according to the frequencies

# Example 1: Huffman Coding

Step 4



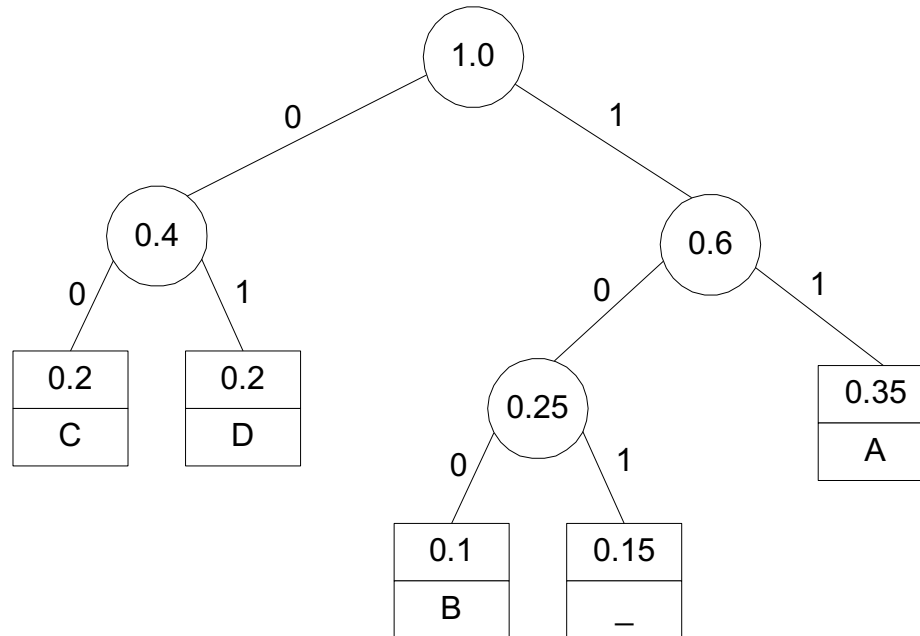
1- Select minimum two frequencies and create a tree



2- Reorder according to the frequencies  
3- If root is one, It is done.

# Example 1: Huffman Coding

Step 5



All the left edges are labeled by 0  
All the right edges are labeled by 1.

The resulting codewords

symbol	A	B	C	D	—
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

# Example 1: Huffman Coding

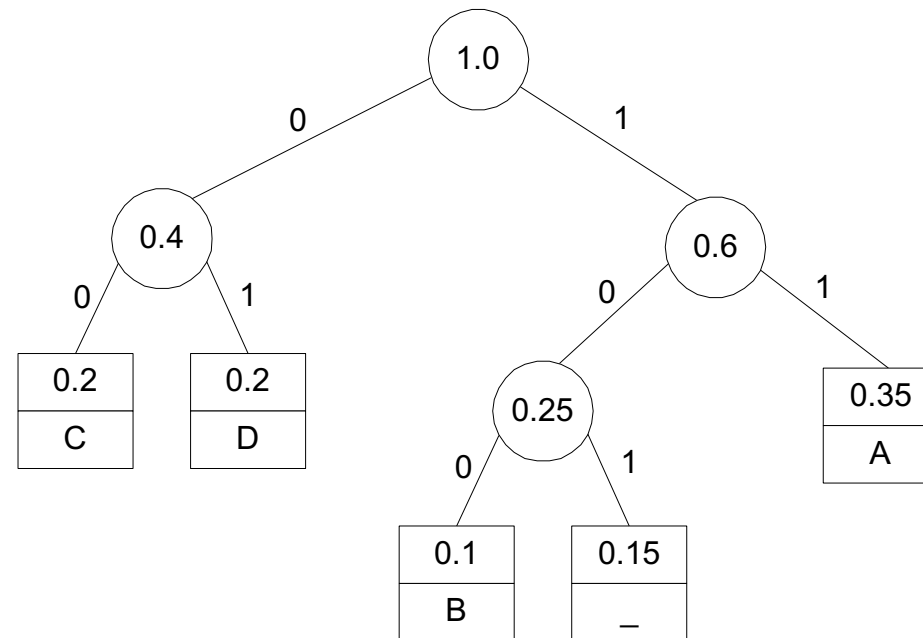
---

character	A	B	C	D	_
codeword	11	100	00	01	101

- With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is
$$2 * 0.35 + 3 * 0.1 + 2 * 0.2 + 2 * 0.2 + 3 * 0.15 = 2.25$$
- Had we used a fixed-length encoding for the same alphabet, we would have to use at least **3 bits per each symbol**.
- For this example, Huffman's code achieves the **compression ratio**—a standard measure of a compression algorithm's effectiveness—of
$$(3 - 2.25)/3 \cdot 100\% = 25\%.$$
- In other words, Huffman's encoding of the text will use 25% less memory than its fixed-length encoding.

# Example 1: Huffman Coding

- DAD is encoded as 011101
- 10011011011101 is decoded as BAD\_AD.



## Example 2: Huffman Coding

---

- Construct a Huffman code for the following message

Message:    A B B A C C D A A B

# Example 2: Huffman Coding

---

## Step 1

Message: A B B A C C D A A B

character	A	B	C	D
frequency	4/10	3/10	2/10	1/10

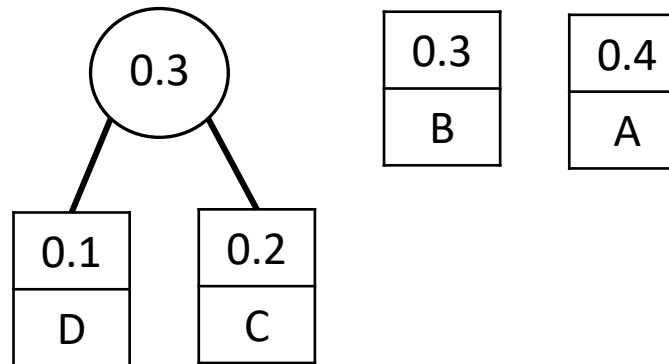
0.1	0.2	0.3	0.4
D	C	B	A



# Example 2: Huffman Coding

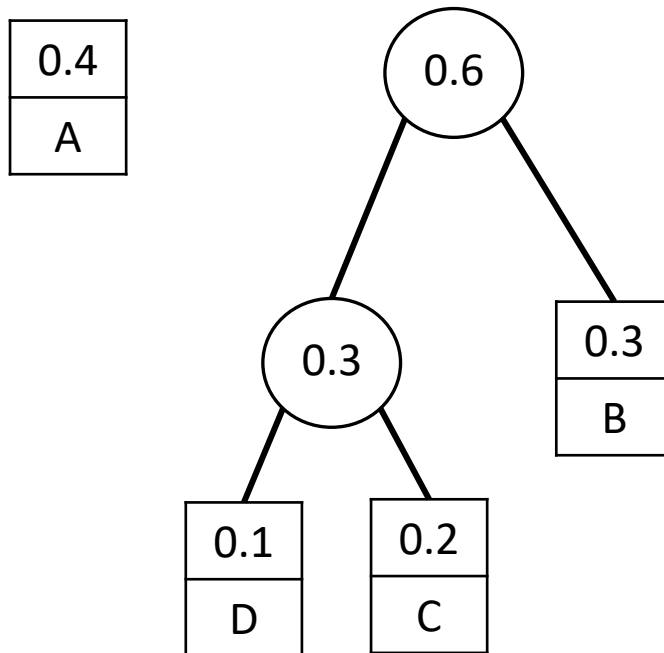
---

## Step 2



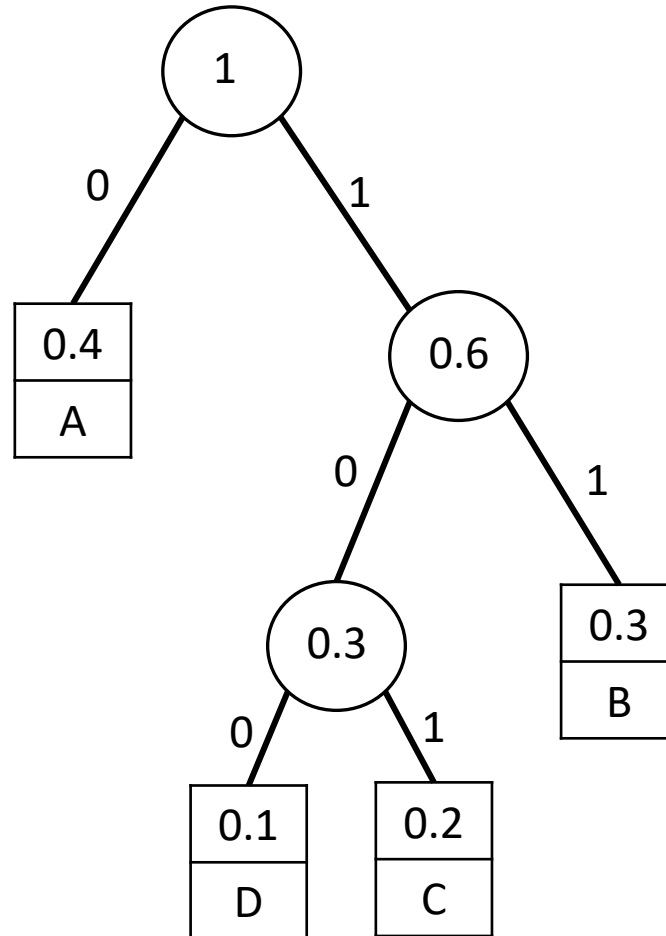
# Example 2: Huffman Coding

## Step 3



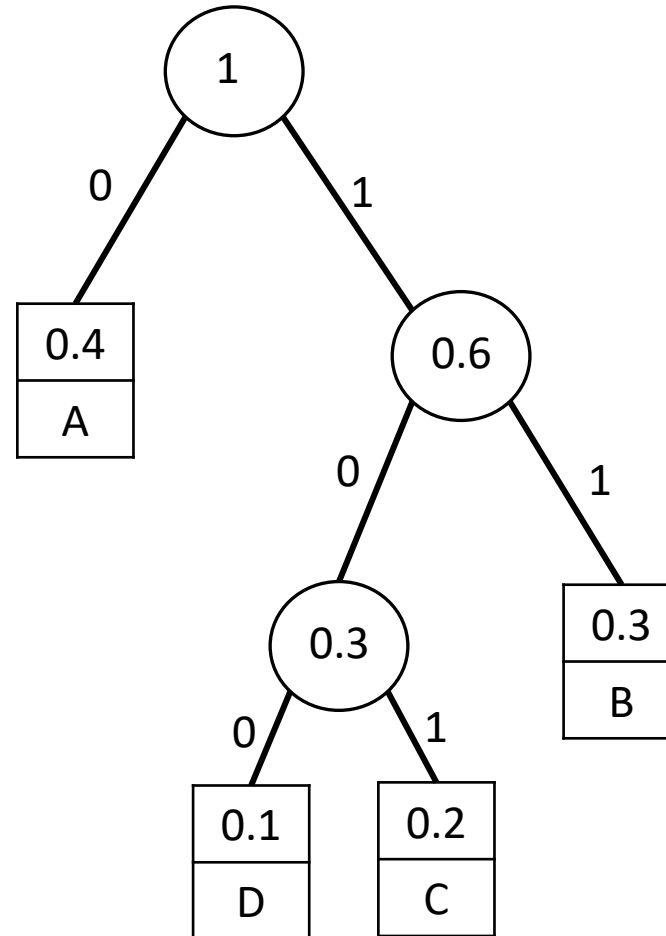
# Example 2: Huffman Coding

Step 4



## Example 2: Huffman Coding

Step 5



A → 0  
B → 11  
C → 101  
D → 100

## Example 2: Encoding

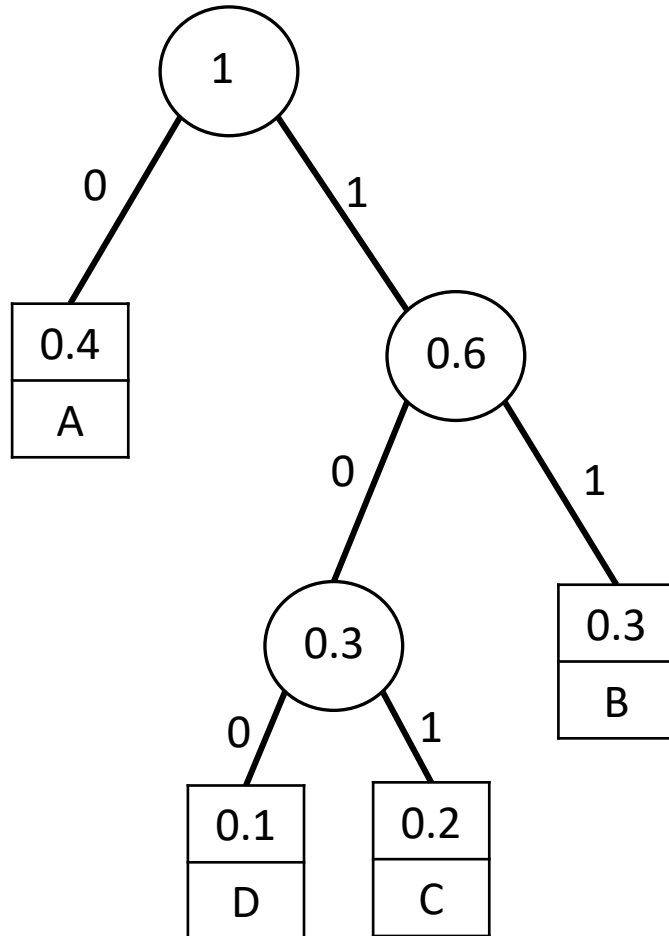
---

A → 0  
B → 11  
C → 101  
D → 100

Message (10 Characters): A B B A C C D A A B

Codeword (19 bits). : 0111101011011000011

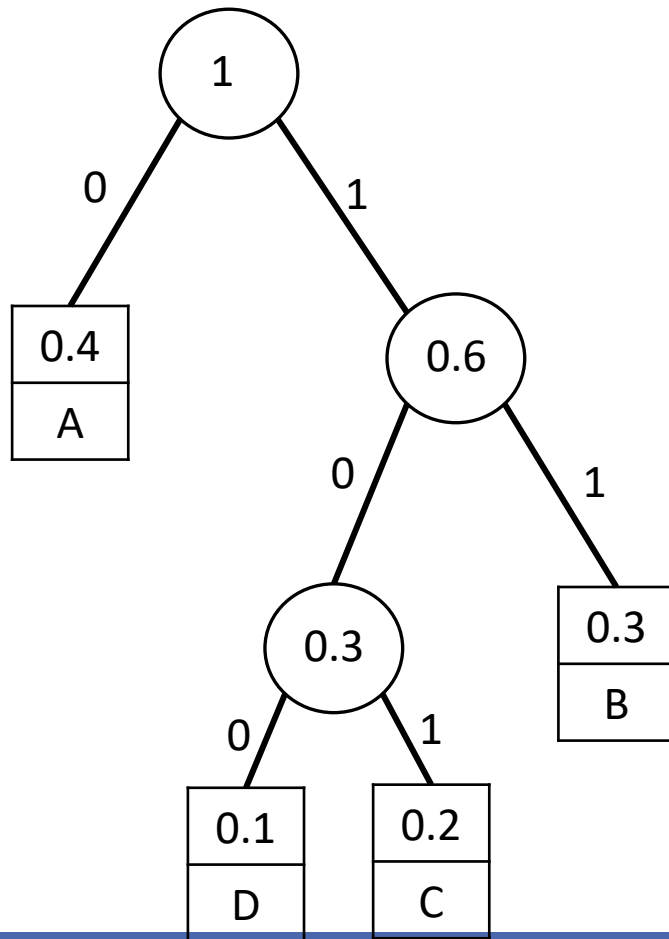
## Example 2: Decoding



Codeword : 011011110101101100

Message : ?

## Example 2: Decoding



Codeword : 011011110101101100

Message : A B A B B A C C D

# Supportive Materials

---

- *Introduction to The Design and Analysis of Algorithms (3rd Edition)* by Anany Levitin, Pearson.
- *Introduction to Algorithms*, by T.Cormen, C. Leiserson, R.Rivest and C.Stein, MIT Press, 3rd Edition.