



**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# **Data Structures and Algorithms Design**

**DSECLZG519**

**Parthasarathy**





**DSECLZG519 – CS#1**

**Introduction to DSECLZG519 &**

**Algorithms**

# Agenda for CS #1

---

## 1) Introduction to DSECLZG519

- Course handout
- Books & Evaluation components
- How to make the most out of this course ?

## 2) Motivation & Synergies between Data Science & DSAD

## 3) Introduction to Algorithms

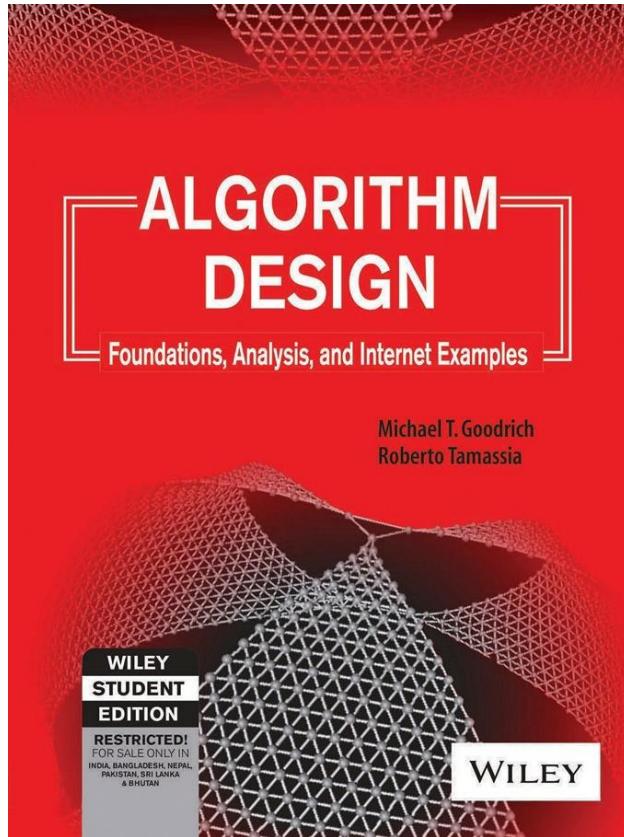
- Notion of an algorithm
- Properties of an algorithm
- Phases of program development
- Why do we have to analyze them ?

## 4) Analysis of algorithm efficiency

- Analysis Framework
- Pseudocode & Counting primitives
- Asymptotic Notations

## 5) Q&A!

# Text Book



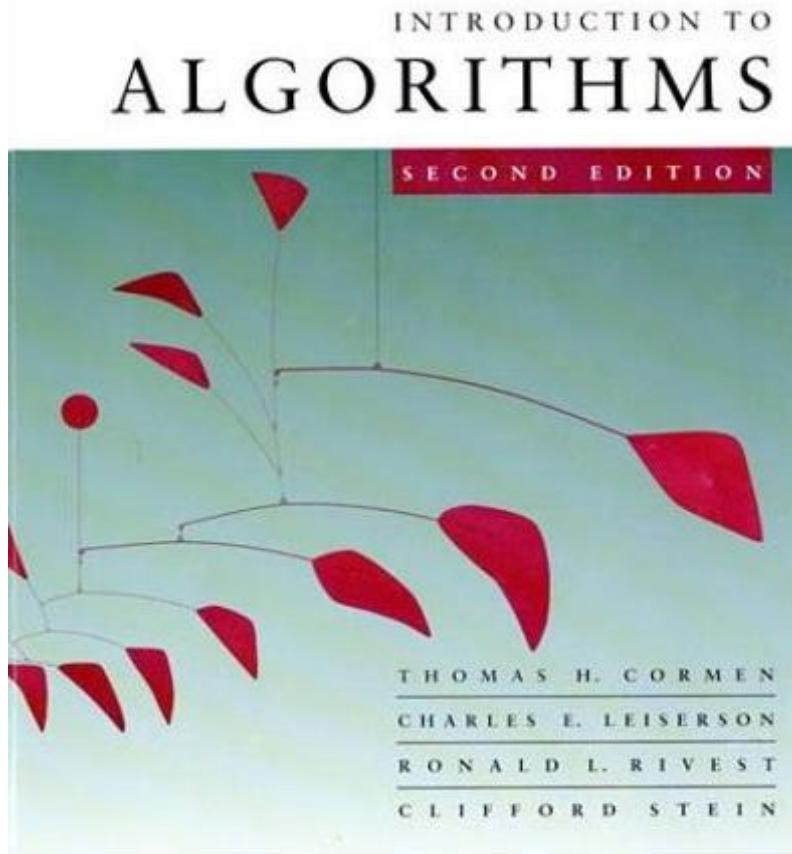
Michael T. Goodrich and Roberto Tamassia: *Algorithm Design: Foundations, Analysis and Internet examples* (John Wiley & Sons, Inc., 2002)

# Reference Books

innovate

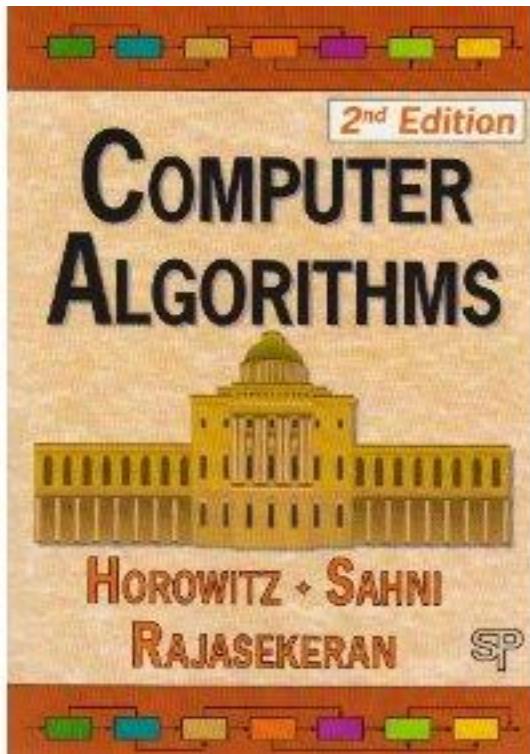
achieve

lead

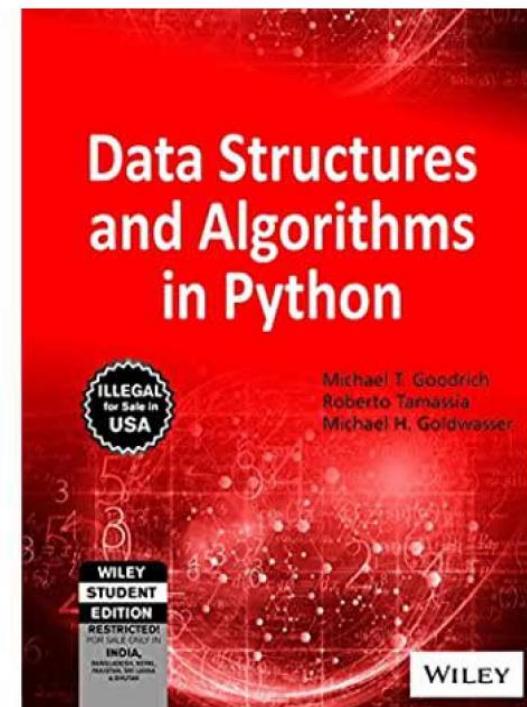


Also known as CLRS book

# Reference Books



Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran. **Computer Algorithms**



Michael T. Goodrich and Roberto Tamassia and Goldwasser: *Data structures and Algorithms in Python* (John Wiley & Sons)

# Ground Rules!

---

- Be Regular 😁
- Mentally present – Observe!! Listen!! 😊
- Keep your questions for the Q&A section ...
- Use the Discussion Forum in Canvas effectively
- Solve the exercises regularly!
- Go an extra mile ☺

$$1^{365} = 1$$

$$1.01^{365} = 37.8$$

# Motivation & DSE – DSAD ?

---

*Aspiring Data Scientist and allied areas ?*

- Awesome! Even in such a role, you would be creating solutions that inevitably have code !!
- When you want to code → You need to have a strong understanding of Data structures and algorithms.
- Data Structures and Algorithms Knowledge give us the ability to improve our solution to the problem and the ability to write much better and efficient code.
- But most importantly, it helps to build problem solving mindset ...
- Thus, learning Data Structures and Algorithms can be a major learning curve for any computer science / data science student.

# What is an Program ?

---

## Algorithm

An algorithm is a *step-by-step procedure* for solving a problem in a finite amount of time.

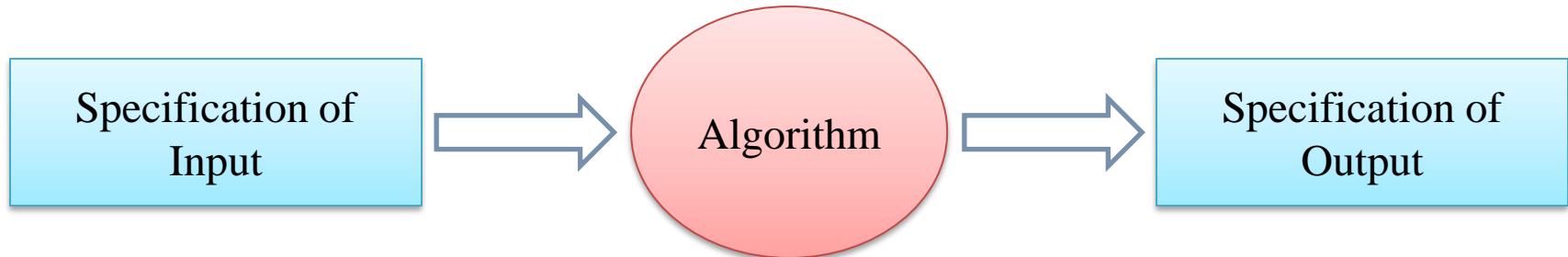
## Data Structures

Is a *systematic way of organizing and accessing data, so that data can be used efficiently*

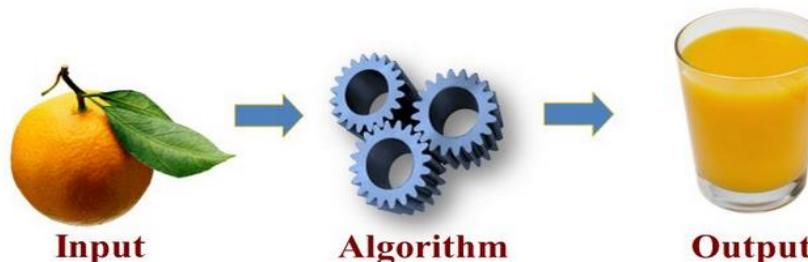
**Algorithms + Data Structures = Program**

*An algorithm is defined as a finite sequence of unambiguous instructions followed to accomplish a given task.*

# Algorithmic Solution



- Algorithm describes actions on the input instance.
- Infinitely many correct algorithm for the same problem.
- Infinite number of input instances satisfying the specification.



# Properties of an Algorithm

---

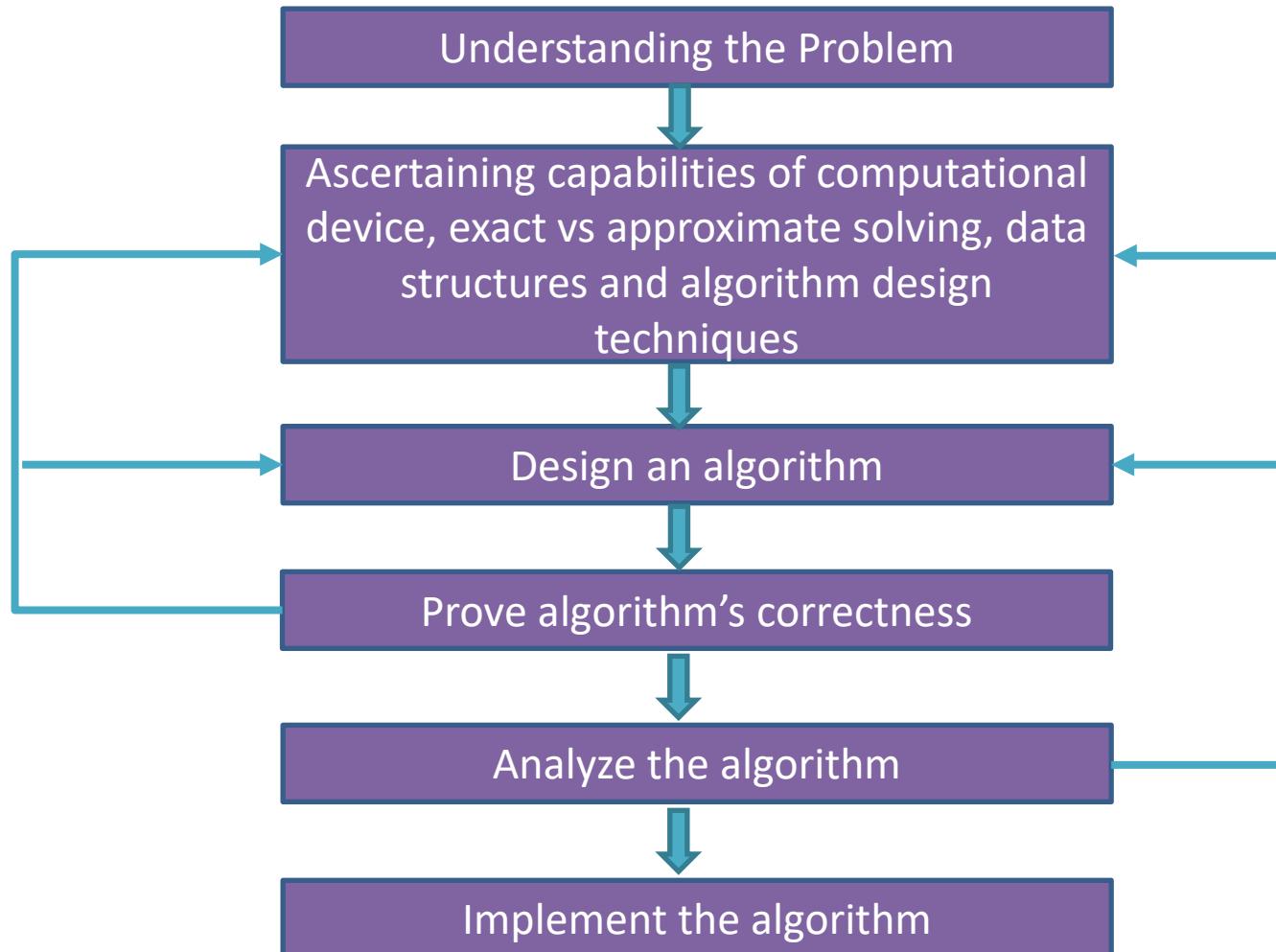
- ✓ **Input** : Each algorithm should have zero or more inputs
- ✓ **Output** : The algorithm should produce correct results. Atleast one output has to be produced
- ✓ **Definiteness** : Each instruction should be clear and unambiguous
- ✓ **Effectiveness** : The instructions should be simple and should transform the given input to the desired output.
- ✓ **Finiteness** : The algorithm must terminate after a finite sequence of instructions

# Example of an Algorithm

---

- Let us first take an example of a real-life situation for creating algorithm. Here is the algorithm for making tea!
  1. Put the teabag in a cup.
  2. Fill the kettle with water.
  3. Boil the water in the kettle.
  4. Pour some of the boiled water into the cup.
  5. Add milk to the cup.
  6. Add sugar to the cup.
  7. Stir the tea.
  8. Drink the tea.
- Some steps like 5,6 can be interchanged but some like 3,8 cannot be interchanged.

# Phases of Program Development



# Data Structures Outlook

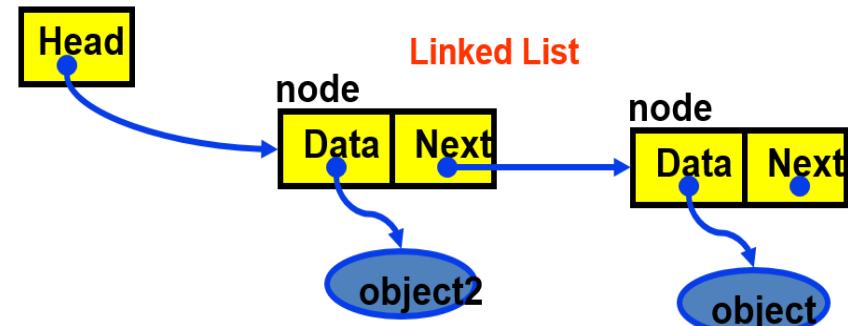
Name of array (Note that all elements of this array have the same name, **c**)

|       |      |
|-------|------|
| ↓     |      |
| c[0]  | -45  |
| c[1]  | 6    |
| c[2]  | 0    |
| c[3]  | 72   |
| c[4]  | 1543 |
| c[5]  | -89  |
| c[6]  | 0    |
| c[7]  | 62   |
| c[8]  | -3   |
| c[9]  | 1    |
| c[10] | 6453 |
| c[11] | 78   |

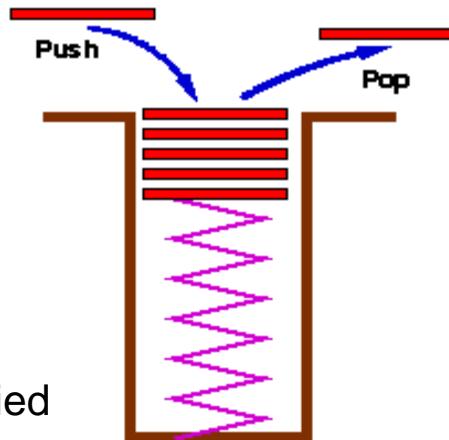
**Array**  
Linearly  
Ordered Set

Position  
number of the  
element within  
array **c**

*A data structure is a particular way of organizing data in a computer so that it can be used effectively.*

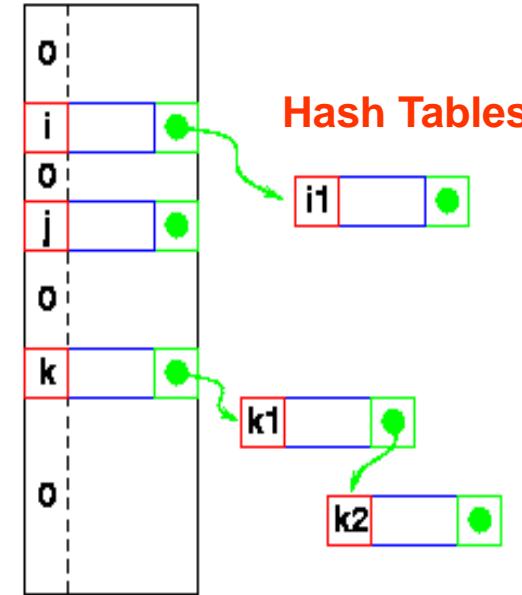


# Data Structures Outlook



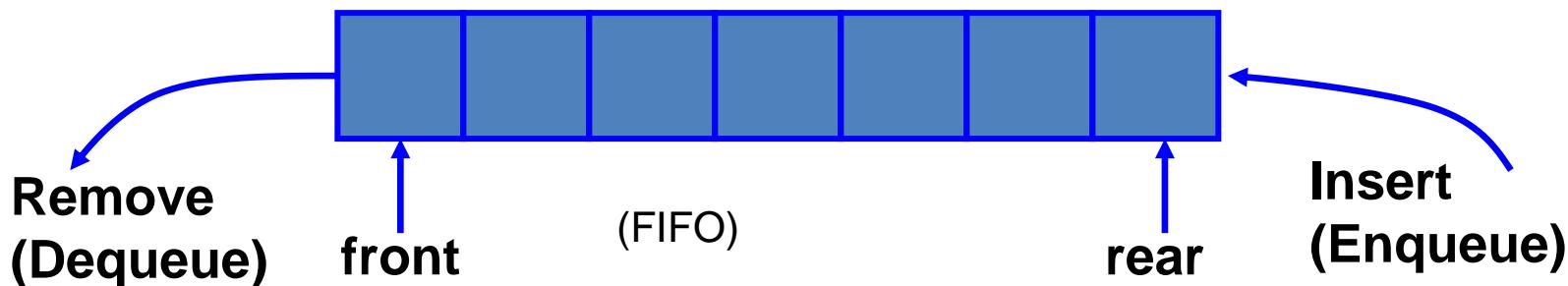
**Stack**

Set with delete operation specified (LIFO)

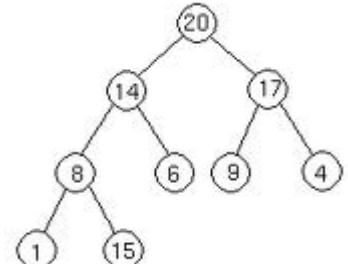
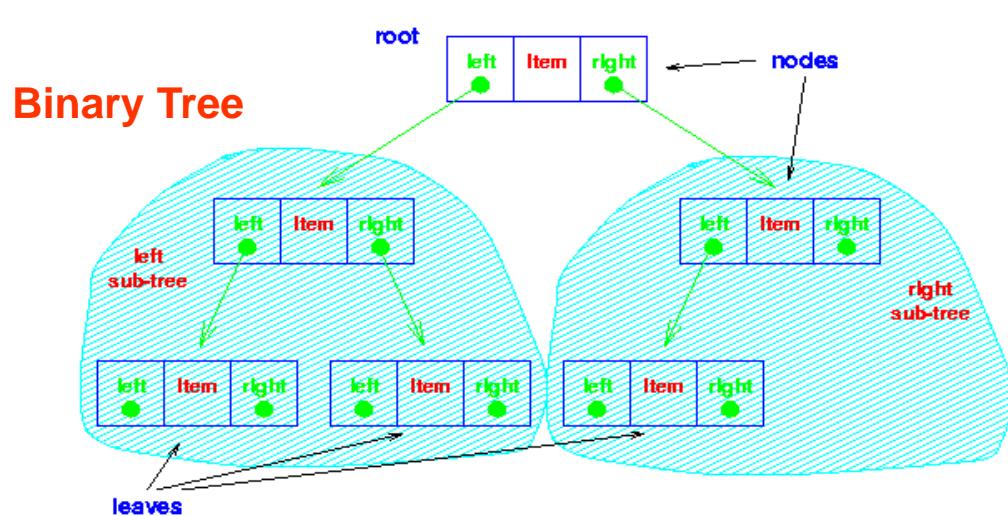


**Hash Tables**

**Queue**



# Data Structures Outlook

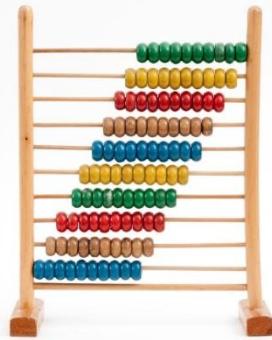


**Heap**



**AVL Tree and others ...**

# Algorithm Techniques – Brute Force

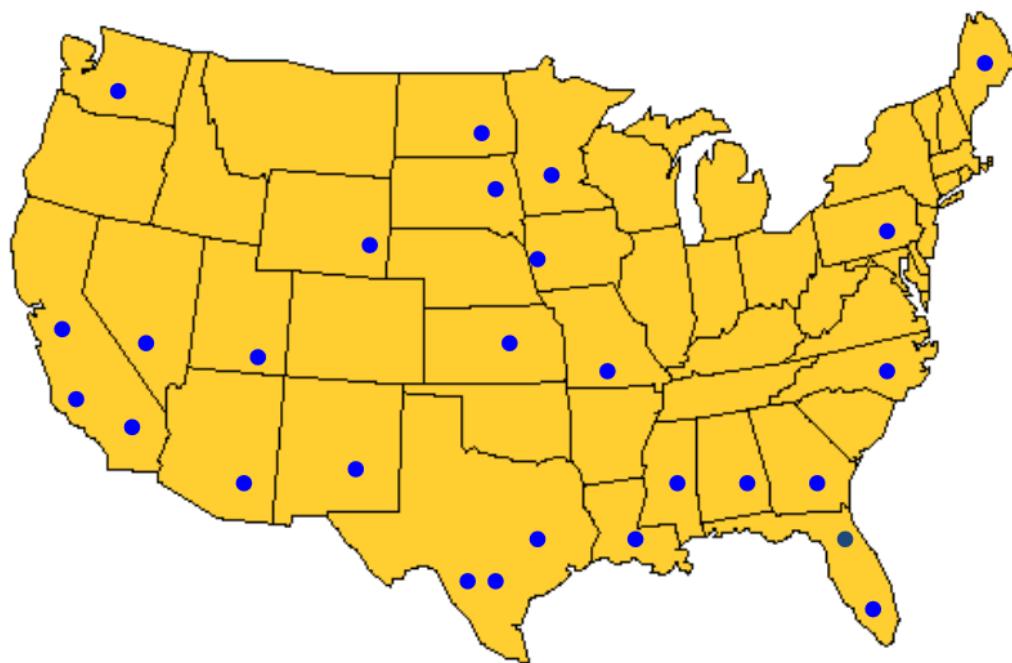


*Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.*

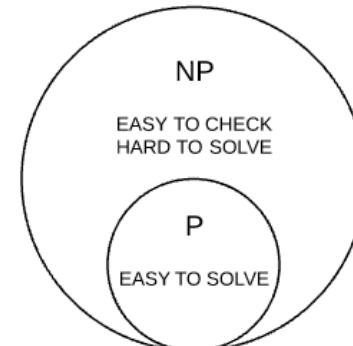
# Algorithm Techniques Outlook

- Brute Force
- Greedy Method
  - Knapsack
  - MST
  - Dijkstra's
- Divide & Conquer
  - Merge Sort
  - Quick Sort
  - Integer Multiplication Problem
- Dynamic Programming
  - Matrix Chain Multiplication
  - Floyd – Warshall's
- ...
- ...

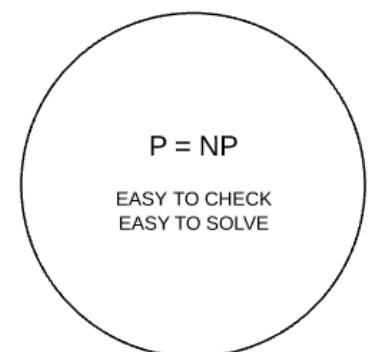
# P, NP, NP-Complete, NP Hard



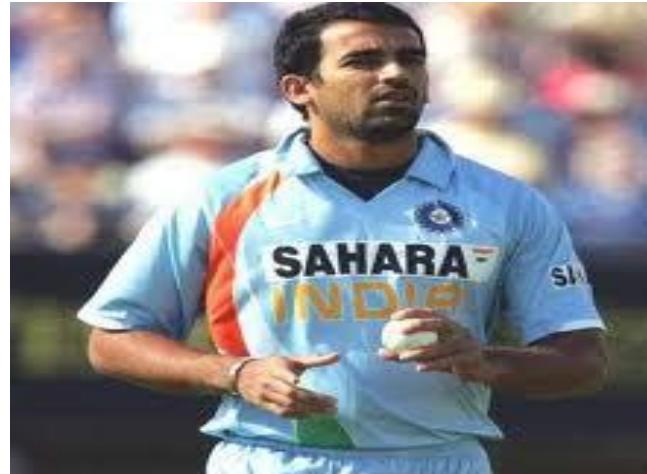
Right now



If  $P = NP$



# Who's the champion?



# Which Algorithm is better ?

Who is topper in DSAD?

Algorithm 1:

Sort A into decreasing order

Output A[1].

Which is better?     $A[i] > m$

Algorithm 2:  
int i;  
int m = A[1];  
for (i = 2; i <= n; i++)  
    if ( $A[i] > m$ )  
        m = A[i];  
return m;

# What is good algorithm?

---

- Resources Used
  - Running time ( Lesser the running time, better the algorithm)
  - Space used (Lesser the space used, better the algorithm)
- Resource Usage
  - Measured proportional to (input) size

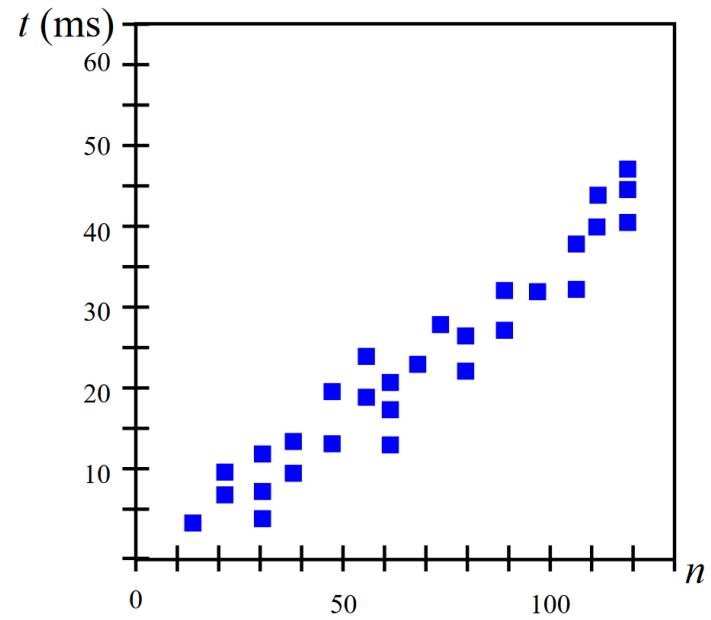
# Analysis Framework

- ✓ “Better” = more efficient
- ✓ Time
- ✓ Space

***How should we measure the running time of an algorithm?***

## Experimental Study

- Write a program that implements the algorithm
- Run the program with data sets of varying size and composition.
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time.
- The result maybe something similar to this :



# Beyond Experimental Studies

---

## Experimental studies have several limitations:

- It is necessary to implement and test the algorithm in order to determine its running time.
- Experiments can be done only on a limited set of inputs, and may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments should be used.
- Analytical Model to analyze algorithm - We will now develop a general methodology for analyzing the running time of algorithms that :
  - Uses a high-level description of the algorithm instead of testing one of its implementations.
  - Takes into account all possible inputs.
  - Allows one to evaluate the efficiency of any algorithm in a way that is independent from the hardware and software environment.

# How to Analyze time complexity ?

---

Running time depends on:

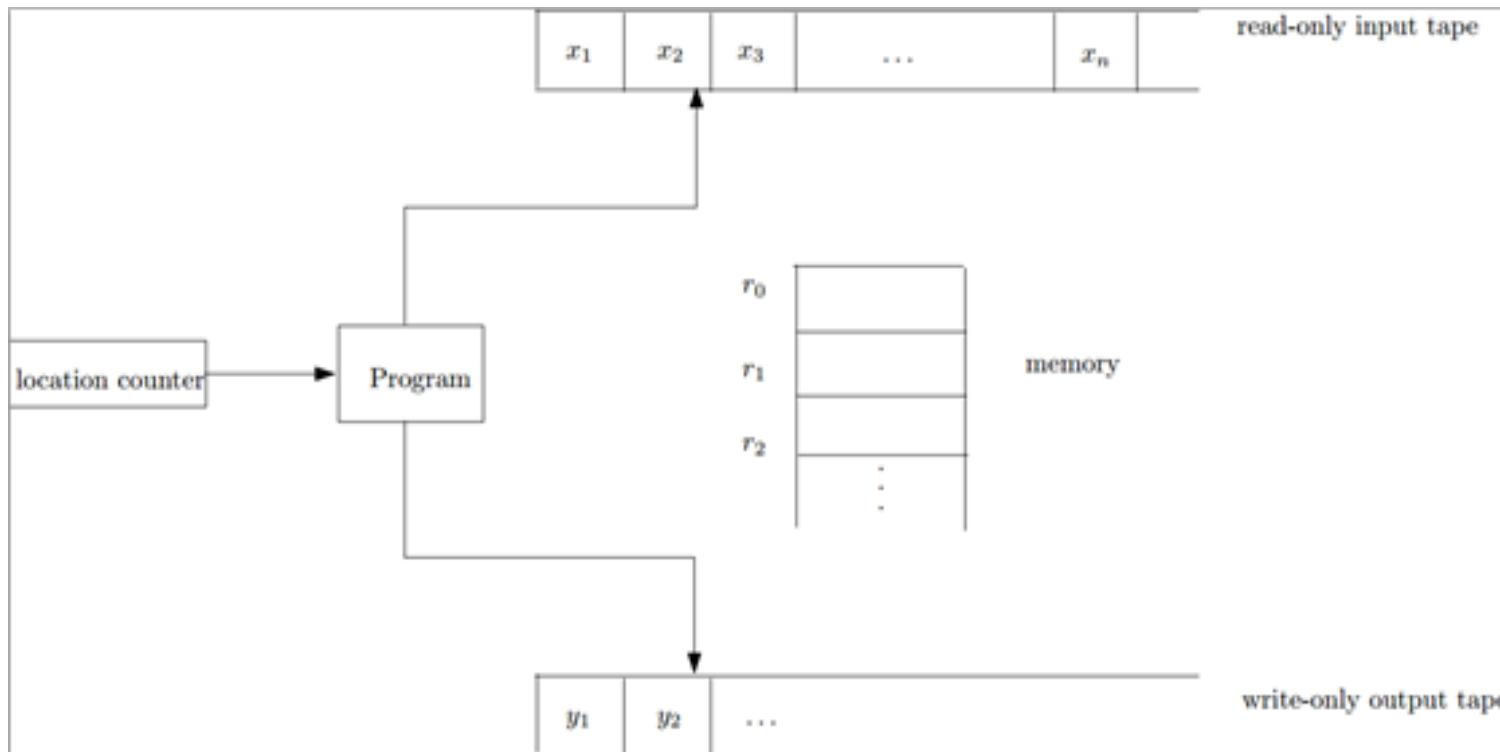
- Single vs multi processor
- Read/write speed to memory
- 32 bit or 64 bit architecture
- Input given to algorithm

A function which defines rate of growth of time w.r.t input!

# Model machine: Random Access Machine model

- Algorithms can be measured in a machine-independent way using the **Random Access Machine (RAM) model**.
- This model assumes a **single processor**.
- In the RAM model, **instructions are executed one after the other, with no concurrent operations**.
- This model of computation is an abstraction that allows us to compare algorithms on the basis of performance.
- The assumptions made in the RAM model to accomplish this are:
  - Each simple operation takes 1 time step.
  - Loops and subroutines are not simple operations.
  - Each memory access takes one time step, and there is no shortage of memory (we assume we have unbounded memory).

# Model machine: Random Access Machine model



- ◆ Time complexity (running time) = number of instructions executed
- ◆ Space complexity = the number of memory cells accessed

# Pseudo-code

A mixture of natural language and high level programming concepts that describes the main ideas behind a generic implementation of a data structure and algorithms.

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

|    |    |    |   |   |
|----|----|----|---|---|
| 0  | 1  | 2  | 3 | 4 |
| 10 | 20 | 30 | 5 | 7 |

Example: find max element of an array

```

Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A

  currentMax  $\leftarrow A[0]
  for i  $\leftarrow 1$  to n – 1 do
    if A[i] > currentMax then
      currentMax  $\leftarrow A[i]$ 
  return currentMax$ 
```

# Pseudo-code (Some Guidelines)

- Is structured than usual text but less formal than a programming language.
- Control flow
  - **if ... then ... [else ...]**
  - **while ... do ...**
  - **repeat ... until ...**
  - **for ... do ...**
  - Indentation replaces braces
- Method declaration
  - **Algorithm *method* (*arg* [, *arg*...])**
- Expressions
  - Use standard mathematical symbols to describe numeric and Boolean expressions.
    - ← Assignment  
(like = in C / Java / Python)
    - = Equality testing  
(like == in C / Java / Python)
    - n*<sup>2</sup>** Superscripts and other mathematical formatting allowed

# Primitive Operations

- **Basic** computations performed by an algorithm
- Identifiable in pseudocode
- Largely *independent* from the programming language
- Assumed to take a constant amount of time in the RAM model

## Examples:

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

# Analyzing Pseudo-code (By Counting) / Counting Primitives

---

1. For each line of pseudocode, count the number of primitive operations in it.
2. Pay attention to the word "*primitive*" here; sorting an array is not a primitive operation.
3. Multiply this count with the number of times this line is executed.
4. Sum up over all lines.

# Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

## Printing each element of an array

---

```
i=0;  
while (i<n)  
{  
    print a[i];  
    i++;  
}
```

---

# Counting Primitive Operations

Printing each element of an array

---

```
i=0;  
while (i<n)  
{  
    print a[i];  
    i++;  
}
```

---

- One Initialization of i
- n+1 comparisons
- n array indexing operations
- n invocations of print
- n increments of I

So, we write  $T(n)=1+n+1+n+n+n$   
 $T(n)=4n+2$

---

# Counting Primitives

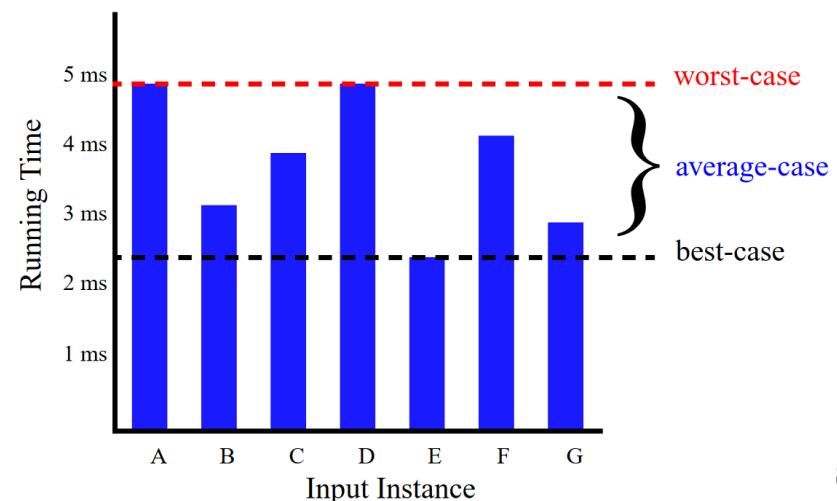
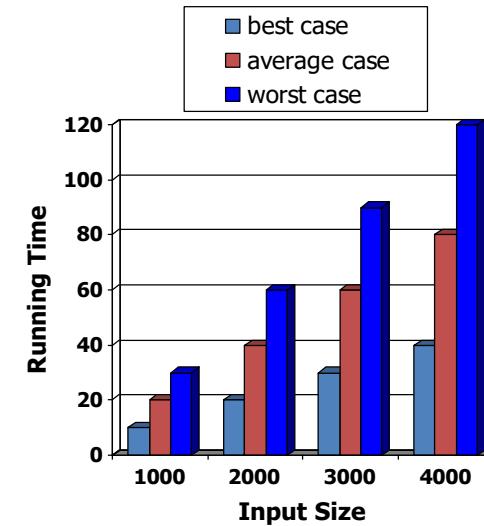
We can also use a tabular way for counting primitives as below:

| <b>Algorithm ArraySum(A, n)</b> | <b>#Operations</b> | <b>Remarks</b>                                |
|---------------------------------|--------------------|---|
| (1) Sum = A [0]                 | 2                  | Indexing , Assignment                         |
| (2) i = 1                       | 1                  | Assignment                                    |
| (3) while (i<n)                 | n                  | Comparison                                    |
| (a) Sum = Sum + A[i]            | 3 (n-1)            | (n-1) times indexing, addition and assignment |
| (b) i = i + 1                   | 2 (n-1)            | (n-1) times addition and assignment           |
| (4) return Sum                  | 1                  | 1 times returning                             |

$$\begin{aligned}
 \text{Add the Operations Column} &= 2 + 1 + n + 3n - 3 + 2n - 2 + 1 \\
 &= 6n - 1
 \end{aligned}$$

# Running Time

- What is best, average, worst case running of a problem?
- An algorithm may run faster on certain data sets than on others.
- Average case time is often difficult to determine – why?
- We focus on the worst case running time.
  - Easier to analyze and best to bet
  - Crucial to applications such as games, finance and robotics
  - Performing well in worst case means it would perform well in normal input too!



# Example

To find the sum of two numbers

Sum(a,b)

{

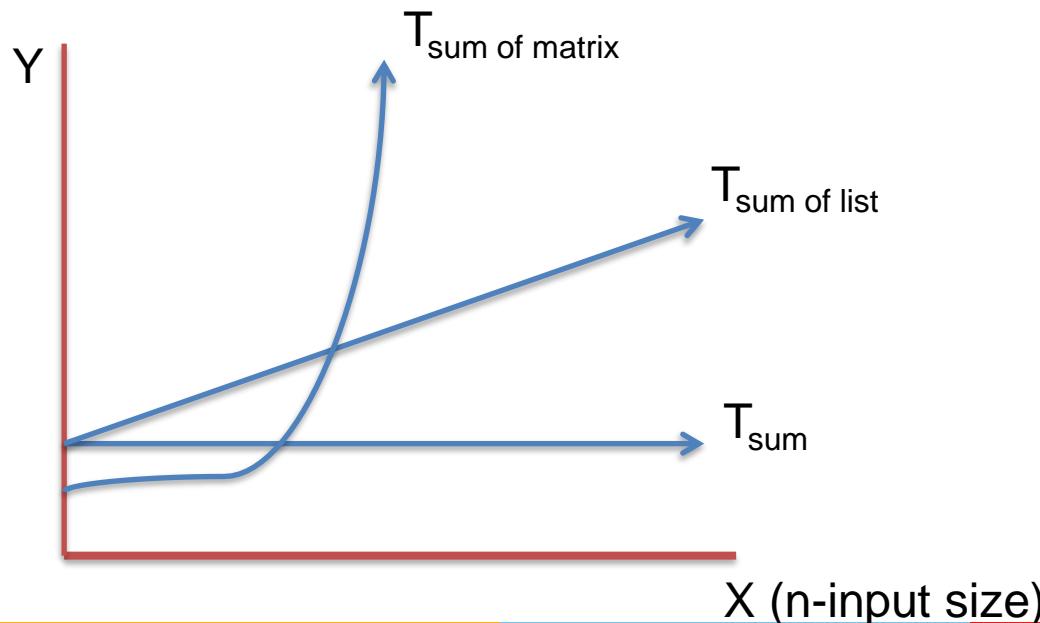
Return a+b

}

$T_{sum} = 2$  {it's a constant time algorithm})

# Example (Cont..,)

- 1)  $T_{\text{sum}} = K \rightarrow$  all the function of form some constant  $O(1)$
- 2)  $T_{\text{sum of list}} = cn + c' \rightarrow$  all linear function  $O(n)$  [linear function]
- 3)  $T_{\text{sum of matrix}} = an^2 + bn + c \rightarrow O(n^2)$  [set of all the function]



# Time complexity Analysis- Some General Rules.

We analyze time complexity

- a) Very large input-size
- b) Worst case scenario

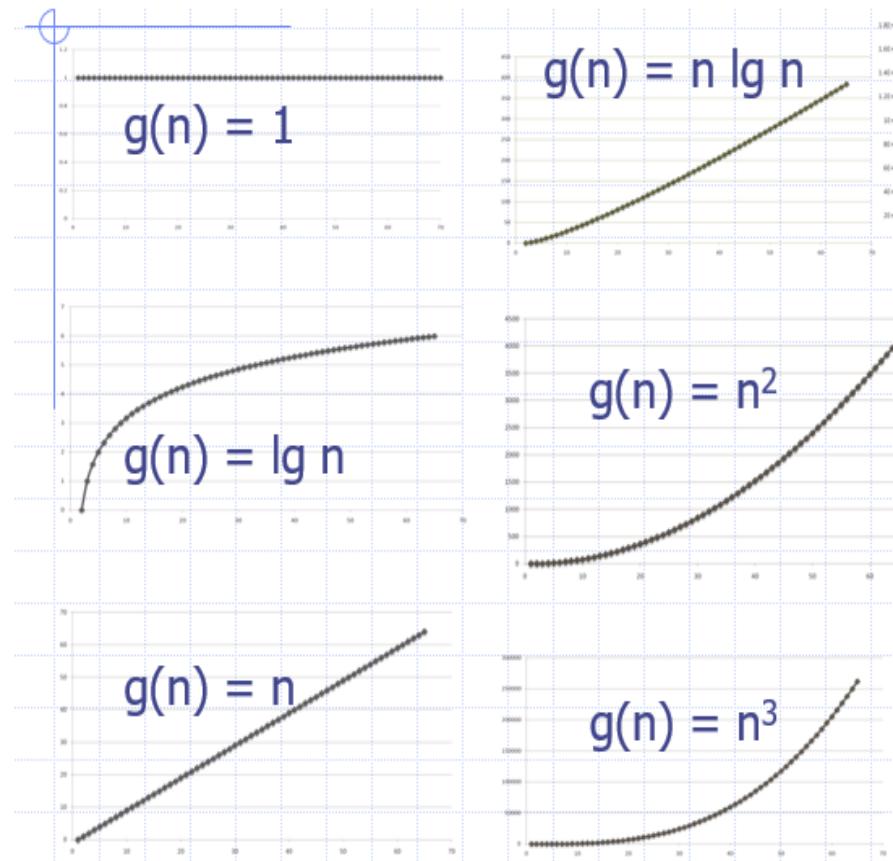
$$\begin{aligned}
 T(n) &= n^3 + 3n^2 + 4n + 2 \\
 &\approx n^3 \quad (n \rightarrow \infty) \\
 &c.n^3 \quad O(n^3)
 \end{aligned}$$

Rule:

- a) drop lower order terms
- b) drop constant multiplier

$$T(n) = 17n^4 + 3n^3 + 4n + 8 = O(n^4)$$

$$T(n) = 16n + \log n = O(n)$$



# Time complexity Analysis- Some General Rules.

## Eliminate low order terms

$$4n + 5 \Rightarrow 4n$$

$$0.5 n \log n - 2n + 7 \Rightarrow 0.5 n \log n$$

$$2^n + n^3 + 3n \Rightarrow 2^n$$

## Eliminate constant coefficients

$$4n \Rightarrow n$$

$$0.5 n \log n \Rightarrow n \log n$$

# Order of Growth

- We expect our algorithms to work faster for all values of N. Some algorithms execute faster for smaller values of N. But, as the value of N increases, they tend to be very slow.
- So, the behavior of some algorithms changes with increase in value of N.
- This change in behavior of the algorithm and algorithm's efficiency can be analyzed by considering the highest value of N.
- Order of growth :

As N increases order of growth increases

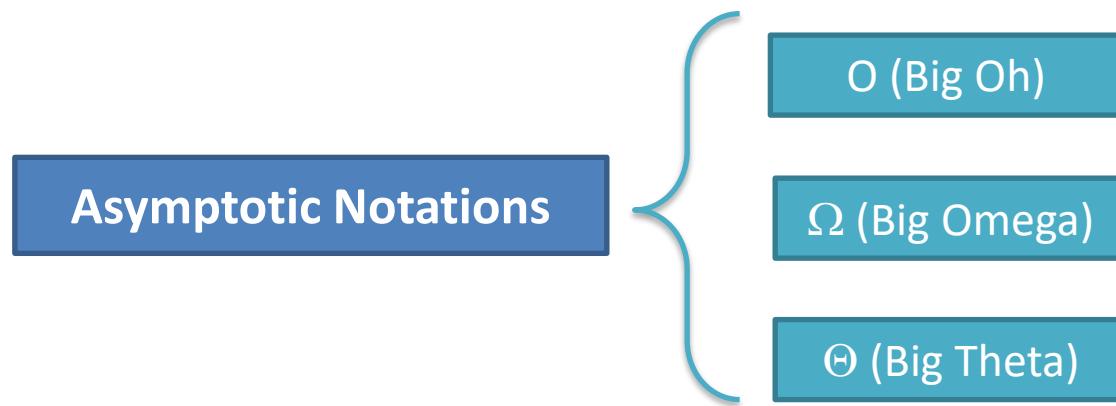
| N  | $\log N$ | N  | $N \log N$ | $N^2$ | $N^3$ | $2^N$      | N!        |
|----|----------|----|------------|-------|-------|------------|-----------|
| 1  | 0        | 1  | 0          | 1     | 1     | 2          | 1         |
| 2  | 1        | 2  | 2          | 4     | 8     | 4          | 2         |
| 4  | 2        | 4  | 8          | 16    | 64    | 16         | 24        |
| 8  | 3        | 8  | 24         | 64    | 512   | 256        | 40320     |
| 16 | 4        | 16 | 64         | 256   | 4096  | 65536      | high      |
| 32 | 5        | 32 | 160        | 1024  | 32768 | 4294967296 | very high |

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

As n Increases, order of Growth increases ...

# Asymptotic Notations

- Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.
- This is also known as an algorithm's growth rate.
- Asymptotic notations are the notations used to express the order of growth of an algorithm and it can be used to compare two algorithms with respect to their efficiency.

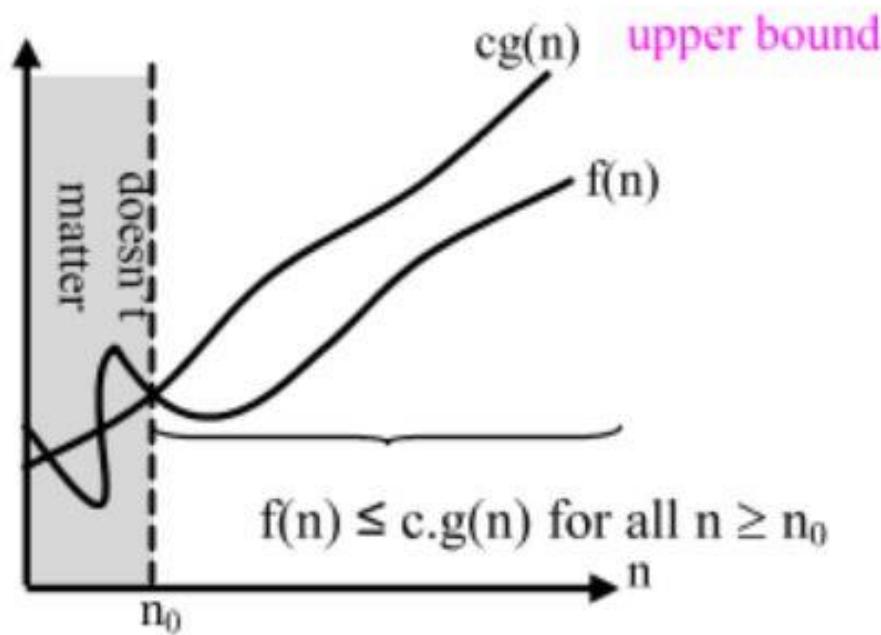


# Asymptotic Notation

- $\mathcal{O}$  notation: asymptotic “less than”: Upper Bound
  - $f(n)=\mathcal{O}(g(n))$  implies:  $f(n)$  “ $\leq$ ”  $g(n)$
- $\Omega$  notation: asymptotic “greater than”: Lower Bound
  - $f(n)=\Omega(g(n))$  implies:  $f(n)$  “ $\geq$ ”  $g(n)$
- $\Theta$  notation: asymptotic “equality”: Average Bound
  - $f(n)=\Theta(g(n))$  implies:  $f(n)$  “ $=$ ”  $g(n)$

# Big - Oh Notation

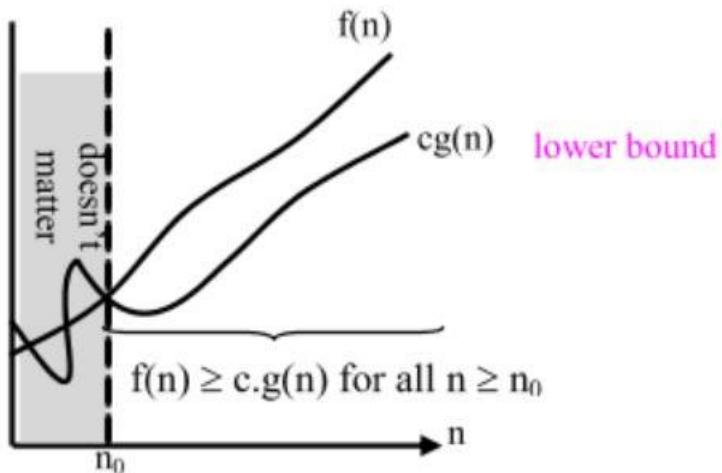
- Big – Oh is the formal method of expressing the upper bound of an algorithm's running time.
- It is a **measure of the longest amount of time** it could possibly take for the algorithm to complete.



Big Oh  
denotes the  
**worst** case  
complexity !!

# Relatives of Big-Oh – Big Omega ( $\Omega(n)$ )

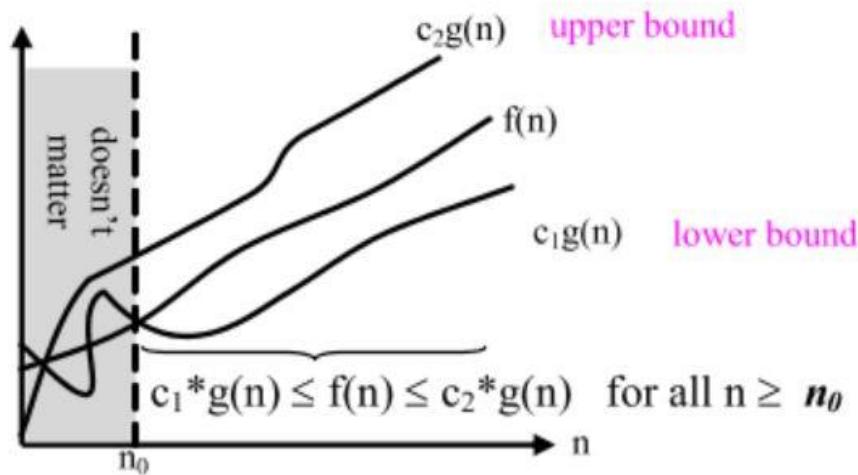
- Big – Omega is the formal method of expressing the lower bound of an algorithm's running time.
- In general, the lower bound implies that below this time the algorithm cannot perform better.
- It is a **measure of the least amount of time** it could possibly take for the algorithm to complete.



Big Omega denotes the **best** case complexity !!

# Relatives of Big-Oh – Big Theta ( $\Theta(n)$ )

- Big – theta is the formal method of expressing both the lower bound & upper bound of an algorithm's running time.
- The upper bound on  $f(n)$  indicates that function  $f(n)$  will not consume more than the specified time  $c_2 * g(n)$  & the lower bound on  $f(n)$  indicates that function  $f(n)$  in the best case will consume atleast the specified time  $c_1 * g(n)$



Big Theta  
denotes the  
**average** case  
complexity !!

# Summary for CS #1

---

## 1) Introduction to DSECLZG519

- Courseware
- Books & Evaluation components
- How to make the most out of this course ?

## 2) Motivation & Synergies between Data Science & DSAD

## 3) Introduction to Algorithms

- Notion of an algorithm
- Properties of an algorithm
- Phases of program development
- Outline of data structures & Algorithm design strategies
- Why do we have to analyze them ?

## 4) Analysis of algorithm efficiency

- Analysis Framework
- Pseudocode & Counting primitives
- Asymptotic Notations

## 5) Exercises, Q&A

# Exercises

Write the pseudocode for below and count the primitives:

- Minimum element in an array
- Maximum of 3 numbers
- Write code to count how many elements are even in the given list.
- Linear Search – Finding if an element is present in an array.

State if the below are true/False , Fill in the blanks

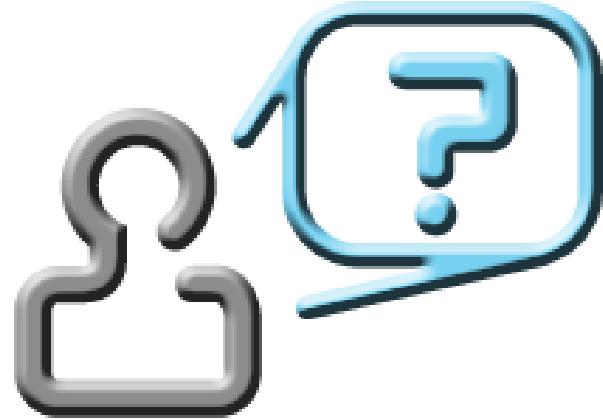
◆ Is  $T(n) = 9n^4 + 876n = O(n^4)$ ?

◆ Is  $T(n) = 9n^4 + 876n = O(n^3)$ ?

◆ Is  $T(n) = 9n^4 + 876n = O(n^{27})$ ?

◆  $T(n) = n^2 + 100n = O(?)$

◆  $T(n) = 3n + 32n^3 + 767249999n^2 = O(?)$



*We will explore more on Algorithm analysis in the next class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# **Data Structures and Algorithms Design**

**DSECLZG519**

**Parthasarathy**





# Contact Session #2

# DSECLZG519 – Analyzing Algorithms

# Agenda for CS #2

---

- 1) Recap of CS#1
- 2) General Rules and Problems
- 3) Mathematical analysis of Non-recursive algorithms
- 4) Mathematical analysis of Recursive algorithms
  - Iteration vs Recursion
  - Recurrence Tree
  - Iterative Substitution
- 5) Masters Theorem with examples
- 6) Exercises, Q&A!

# Asymptotic Notations

## Analogies

---

- “The delivery will be there within your lifetime.”
  - *(big-O, upper-bound)*
- “I can pay you at least one dollar.”
  - *(big-omega, lower bound)*
- “The high today will be 25°C and the low will be 19°C.”
  - *(big-theta, narrow)*
- “It’s a kilometer walk to the beach.”
  - *(big-theta, exact)*

# General rules to determine running time of algorithm

**Loops:** The running time of a loop is, at most, the running time of the statements inside the loop(including tests) multiplied by the number of iterations.

//executes n-times

```
for(i=1;i<=n;i++)
    ... // constant time
```

Total number

$$= \text{a constant } c \times n = cn = O(n)$$

$$F(n) = \sum_{i=1}^n 1$$

$$= n - 1 + 1$$

$$= n$$

# General rules to determine running time of algorithm (Cont...,)

---

**Nested loops:** Analyze from inside out. Total running time is the *product of the sizes* of all the loops

//outer loop executed n-times

```
for(i=1;i<=n;i++){  
    //inner loop execute n-times  
    for(j=1;j<=n;j++)  
        //constant time  
        k=k+1;  
}
```

$$\text{Total number} = c \times n \times n = cn^2 = O(n^2)$$

# General rules to determine running time of algorithm (Cont...)

---

**If-then-else statements:** worst-case running time: the test, plus either the then part or the else part (whichever is the larger).

```
//test: constant c0
If(length()==0){
    return false; // then part: constant c1
}
else { //else part : (constant c2 +constant c3)× n
    for(int n =0; n<length();n++){
        //another if: constant + constant (no else part)
        if(! List[n].equals(otherlist.list[n]))
            //constant
        return false;
    }
}
```

$$\text{Total time} = c_0 + c_1 + (c_2 + c_3) \times n = O(n)$$

# General rules to determine running time of algorithm (Cont...)

---

**Logarithmic Complexity:** an algorithm is  $O(\log n)$  if it takes constant time to cut the problem size by a fraction (usually by  $\frac{1}{2}$ )

As an example let us consider the following program

```
for(i=1; i<=n;)
    i=i*2;
```

If we observe carefully, the value of  $i$  is doubling every time initially  $i=1$ , in next step  $i=2$ , and in subsequent steps  $i = 4, 8$  and so on.

Let us assume that the loop is executing some  $k$ -times.

At  $k^{\text{th}}$  step  $2^k=n$  and we come out of loop

Taking logarithm on both sides, gives

$$\log(2^k) = \log n$$

$$k \log 2 = \log n$$

$$k = \log n // \text{if we assume base } \sim 2.$$

Total number =  $O(\log n)$

## Problem-1

---

What is the complexity of the program given below

void function (int n){

    for (i =1; i<=n; i++)

        for (j=1;j+n/2<=n;j=j++)

            //loop execute logn times

        for (k=1;k<=n;k=k\*2)

            count++;

}

Time complexity of the above function is  $O(n^2 \log n)$

More on it in the discussion thread ...

## Problem-2

```
function (int n){  
    //constant time  
    if(n==1) return;  
    //outer loop execute n times  
    for (int i =1;i<=n;i++){  
        // inner loop executes only time due to break statement  
        for (int j =1;j<=n;j++){  
            printf("*");  
            break;  
        }  
    }  
}
```

Time complexity of the above function is?

Time Complexity of the above function  $O(n)$ . *Even though the inner loop is bounded by  $n$ , but due to the break statement, it is executing only once.*

# Complexity Definition

---

- **Linear Complexity:** Processing time or storage space is directly proportional to the number of input elements to be processed.
- **Quadratic complexity:** An algorithm is of quadratic complexity if the processing time is proportional to the square of the number of input elements.
- **Cubic complexity:** In the case of cubic complexity, the processing time of an algorithm is proportional to the cube of the input elements.
- **Logarithmic complexity:** An algorithm is of logarithmic complexity if the processing time is proportional to the logarithm of the input elements. The logarithm base is typically 2

# Analysis of Algorithms

---

Techniques employed by algorithms to solve problems

- **Iterative**
- **Recursion**

## What is Recursion

- The process in which a function calls **itself** directly or indirectly is called recursion and the corresponding function is called as recursive function
- Define a procedure P that is allowed to make calls to itself, provided those calls to P are for solving sub-problems of smaller size.

## Condition for recursive procedure

- A recursive procedure should always define a base case
- Base case
  - Small sized problem which can be solved by the algorithm directly without using recursion

# Recursive Algorithm

---

1. if problem is “*small enough*”
2.       solve it *directly*
3. else
4.       break into one or more *smaller sub problems*
5.       solve each sub problem *recursively*
6.       *combine* results into solution to whole problem

# Requirements for Recursive Solution

---

- At least one “*small*” case that you can solve directly
- A way of *breaking* a larger problem down into:
  - One or more *smaller* sub problems
  - Each of the *same kind* as the original
- A way of *combining* sub problem results into an overall solution to the larger problem

# Recursive Algorithm - Example

---

***Recursive algorithm for finding length of a string:***

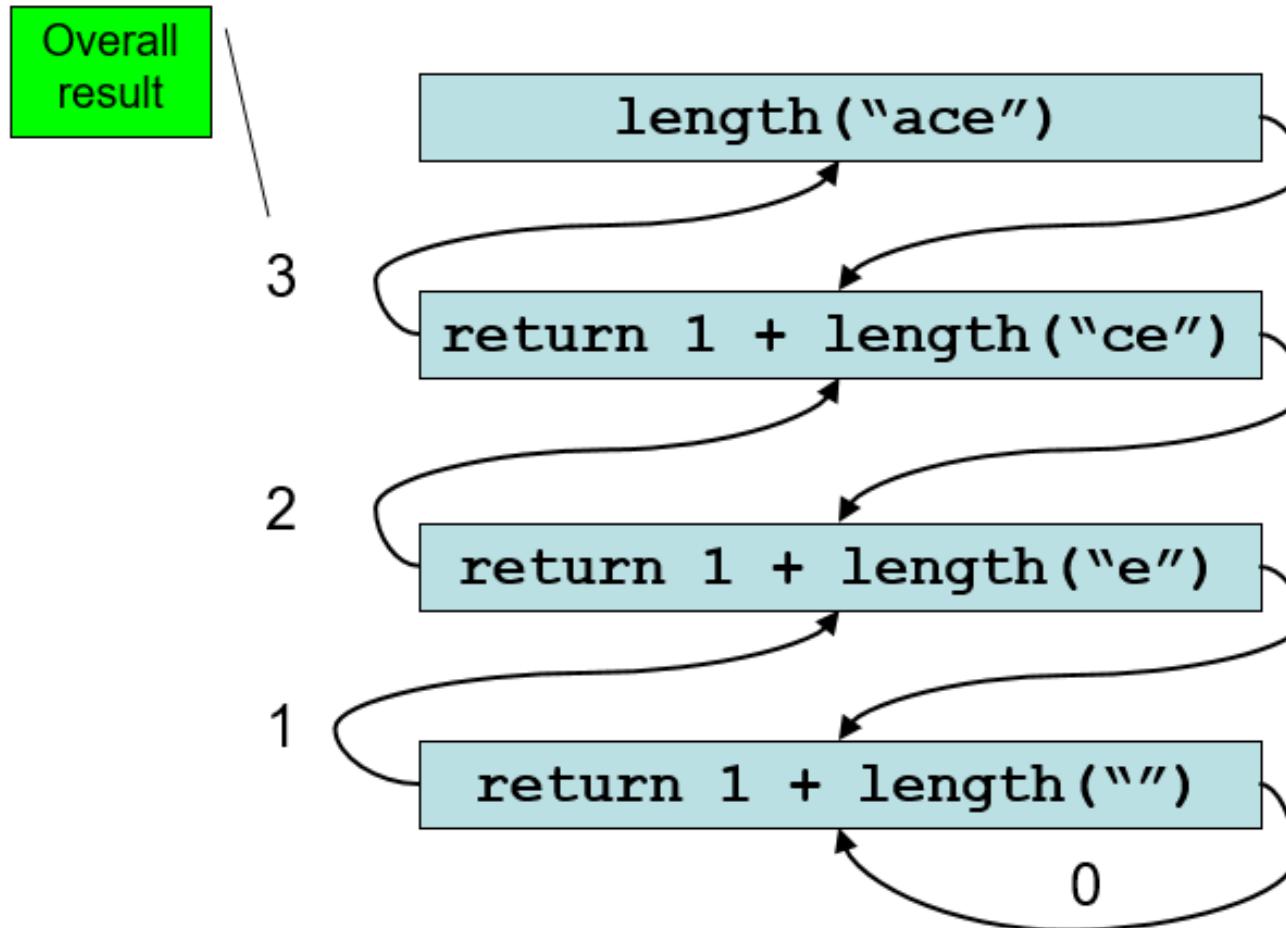
1. if string is empty (no characters)
2.     return 0     $\leftarrow$  base case
3. else     $\leftarrow$  recursive case
4.     compute length of string without first character
5.     return 1 + that length

# Recursive Design Example: Code

Recursive algorithm for finding length of a string:

```
public static int length (String str) {  
    if (str == null || str.equals("")))  
        return 0  
    else  
        return length(str.substring(1)) + 1  
}
```

# Tracing a Recursive Method



# Example

- Compute  $N! = 1 * 2 * 3 * \dots * N$

```
//An iterative solution
int factorial(int N){
    result = 1;
    for (i = 1; i <= N; i++)
        result = i * result;
    return result;
}
```

- Alternative Solution: factorial calls itself recursively

```
int factorial (int N) {
    if (N == 1) { } base case
    return 1;
} else {
    return N * factorial (N-1); } recursive case
}
```

# Analysis of non-recursive Algorithms

---

- Based on the input size, determine the number of parameters to be considered.
- Identify the basic operation in the algorithm
- Check whether the number of times the basic operation is executed depends only on the size of the input.
- Obtain the total number of times a basic operation is executed.
- Simplify using standard formulas, obtain the order of growth.

# Example

**Algorithm** Linear\_Search(key, a[], n)

**//Purpose:** To search for key in an array of  $n$  elements

**// Inputs :**

- n – the number of items present in the table
- a – the items to be searched are present in the table
- key – the item to be searched

**//Output:**

- return  $i$  whenever key is present in the list
- return -1 whenever key is not present in the list

**Step 1: [Search for the key in the array/table]**

```

for i ← 0 to n-1 do
    if (key = a[i]) return i;      // Key Found at position i
end for

```

**Step 2:[Key not found]**

```
return -1;
```

- In this problem, the size of input is  $n$  and this is the parameter to be considered.
- The statement executed in loop is the basic operation which is “ $\text{if}(\text{key} = \text{a}[i])$ ”
- The basic operation is inside a single loop and hence the complexity is  $O(n)$

# Analysis of recursive Algorithms

- Based on the input size, determine the number of parameters to be considered.
- Identify the basic operation in the algorithm
- Obtain the number of times the basic operation is executed on different inputs of the same size.
- Obtain the **recurrence relation** with an appropriate initial condition.
- Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

## Recurrence equation

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A *recurrence* is an equation or inequality that describes a function in terms of its value on smaller inputs.

# Solution to recurrence equations

$$T(n) = \begin{cases} 1 & ; n = 0 // \text{base case} \\ 1 + T(n-1) & ; n > 0 // \text{recursive case} \end{cases}$$

- Recursion tree method
- Iterative substitution method
- Master's theorem

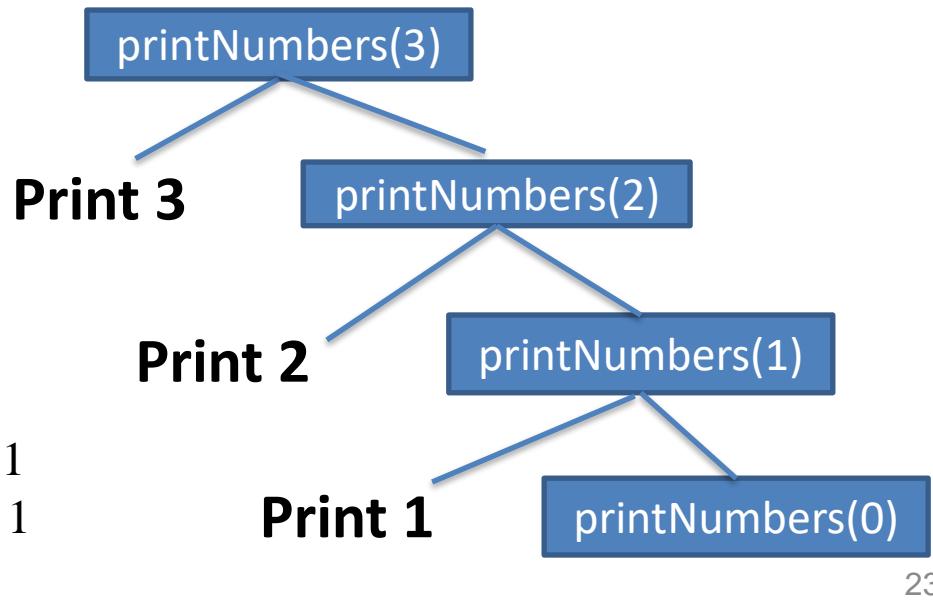
# Recursive Tree method

## Recursion tree method

- Visual method
- Converts the recurrence into a tree
- Sum all per-level cost to get the total cost of recursion

```
function printNumbers(n)
    if(n > 0)
        print n
        printNumbers(n-1)
end
```

Lets assume  $n = 3$



From the tree, its clear that it takes  $T(3)= 3 + 1$

$$\begin{aligned} T(n) &= n + 1 \\ O(n) \end{aligned}$$

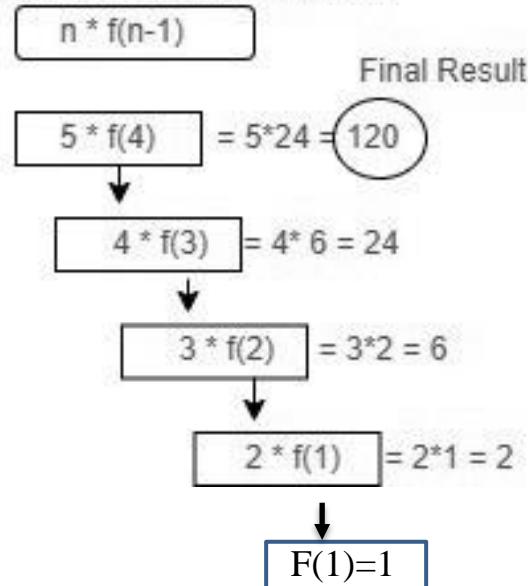
# Factorial using Recursion – Recursive Tree Method



## Recursive tree method

For user input : 5

Factorial Recursion Function



```
func factorial(n)
    if(n == 1)
        return 1
    return n * factorial(n-1)
end
```

It is clear from the recursive tree that for the recursion unfolds  $n$  levels with 1 unit in each level of the tree. Hence, the complexity is  $n * 1 = \mathbf{O(n)}$

# Iterative substitution method

## Iterative substitution method

- Known as the "plug-and-chug" method
- Substitute the general form of the recurrence for each occurrence of the function T on the right-hand side (mathematical induction ☺ )
- The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general form equation!
- Then we use the base case to solve the equation

### Example

```
function printNumbers(n)
  if(n > 0)
    print n
    printNumbers(n-1)
end
```

$$T(n) = \begin{cases} 1 & ; n = 0 // \text{base case} \\ 1 + T(n-1) & ; n > 0 // \text{recursive case} \end{cases}$$

Let us solve this recurrence relation using iterative substitution method!!

# Substitution Method

- A way to solve a linear system algebraically is to use the substitution method. The substitution method functions by substituting the one  $y$  value with the other.
- Example:

$$y = 2x + 4$$

$$3x + y = 9$$

We can substitute  $y$  in the second equation with the first equation since  $y = y$ .

$$3x + y = 9$$

$$3x + (2x + 4) = 9$$

$$5x + 4 = 9$$

$$5x = 5$$

$$x = 1$$

This value of  $x$  can then be used to find  $y$  by substituting 1 with  $x$  e.g. in the first equation

$$y = 2x + 4$$

$$y = 2 \cdot 1 + 4$$

$$y = 6$$

The solution of the linear system is  $(1, 6)$ .

# Substitution Method

$$T(n) = \begin{cases} 1 & ; n = 0 // \text{base case} \\ 1 + T(n-1) & ; n > 0 // \text{recursive case} \end{cases}$$

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

Substitute value of  $T(n-1)$  in above:

$$T(n) = [ T(n-2) + 1 ] + 1$$

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

Substitute value of  $T(n-2)$  in above:

$$T(n) = [ T(n-3) + 1 ] + 2$$

$$T(n) = T(n-3) + 3$$

.

.

.

$$\text{In general, } T(n) = T(n-k) + k \dots (2)$$

The recursion will end when the subproblem size reaches the base case, i.e. when  $n-k = 0$ , which means when  $n=k$ . Substitute the value of  $k$  in (2) and solve:

$$\begin{aligned} T(n) &= T(n-n) + n \\ &= T(0) + n \\ &= 1 + n \end{aligned}$$

What is  $T(n-1)$ ?

We know that  $T(n)=T(n-1) + 1 \dots (1)$

➤ Put  $n=n-1$  in (1) :

$$\begin{aligned} T(n-1) &= T(n-1-1) + 1 \\ &= T(n-2) + 1 \end{aligned}$$

➤ Put  $n=n-2$  in (1) :

$$\begin{aligned} T(n-2) &= T(n-2-1) + 1 \\ &= T(n-3) + 1 \end{aligned}$$

We have now solved the recurrence relation, drop the constant, we get **O(n)**

# Example 2 – Factorial using Recursion

## Iterative substitution method

```
func factorial(n)
    if(n == 1)
        return 1
    return n * factorial(n-1)
end
```

$$T(n) = \begin{cases} 1 & ; n = 1 // \text{base case} \\ 1 + T(n-1) & ; n > 1 // \text{recursive case} \end{cases}$$

function  $T(n)$  is not actually for calculating the value of an factorial but it is telling you about the time complexity of the factorial algorithm.

Let us solve this recurrence relation using iterative substitution method!!

Find attached CS#2 - PDF for solution. Complexity is  $O(n)$

# The Master's Method

- Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively.
- The masters method is like a “cook-book” / A utility method for analysis recurrence relations.
- Useful in many cases for divide and conquer algorithms.
- They come handy when the recurrence relation is of the form :

$$T(n) = aT(n/b) + f(n) \text{ where } a \geq 1, b > 1, n \geq d$$

- $n$  = the size of the current problem and  $d \geq 1$  is a constant
- $a$  = the number of sub-problems in the recursion.
- $n/b$  = the size of each sub problem
- $f(n)$  = the cost of work that has to be done outside the recursive calls (Cost of dividing + merging).

# The Master Theorem

The master method for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing  $f(n)$  to the special function  $n^{\log_b a}$

1. If there is a small constant  $\varepsilon > 0$ , such that  $f(n)$  is  $O(n^{\log_b a - \varepsilon})$ , then  $T(n)$  is  $\Theta(n^{\log_b a})$ .
2. If there is a constant  $k \geq 0$ , such that  $f(n)$  is  $\Theta(n^{\log_b a} \log^k n)$ , then  $T(n)$  is  $\Theta(n^{\log_b a} \log^{k+1} n)$ .
3. If there are small constants  $\varepsilon > 0$  and  $\delta < 1$ , such that  $f(n)$  is  $\Omega(n^{\log_b a + \varepsilon})$  and  $a f(n/b) \leq \delta f(n)$ , for  $n \geq d$ , then  $T(n)$  is  $\Theta(f(n))$ .

Case 1 characterizes the situation where  $f(n)$  is polynomially smaller than the special function,  $n^{\log_b a}$ . Case 2 characterizes the situation when  $f(n)$  is asymptotically close to the special function, and Case 3 characterizes the situation when  $f(n)$  is polynomially larger than the special function.

# Simplified version of Master Theorem

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

- ✓ Where  $f(n) = \theta(n^k \log^p n)$
- ✓  $n$  = size of the problem
- ✓  $a$  = number of subproblems in the recursion and  $a \geq 1$
- ✓  $n/b$  = size of each subproblem
- ✓  $b > 1, k \geq 0$  and  $p$  is a real number.

Case 1. if  $\log_b a > k$ , then  $T(n) = \Theta(n^{\log_b a})$

Case 2. if  $\log_b a = k$ , then

- (a) if  $p > -1$ , then  $T(n) = \Theta(n^k \log^{p+1} n)$
- (b) if  $p = -1$ , then  $T(n) = \Theta(n^k \log \log n)$
- (c) if  $p < -1$ , then  $T(n) = \Theta(n^k)$

Case 3. if  $\log_b a < k$ , then

- (a) if  $p \geq 0$ , then  $T(n) = \Theta(n^k \log^p n)$
- (b) if  $p < 0$ , then  $T(n) = \Theta(n^k)$

# Simplified version Case 1 Example

*Case 1: if  $\log_b a > k$ , then  $T(n) = \Theta(n^{\log_b a})$*

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

Solution:

$T(n) = aT(n/b) + f(n)$  where  $f(n) = \Theta(n^k \log^p n)$

**Step 01:** Here,  $a = 2$ ,  $b = 2$ ,  $f(n) = 1$  which can happen when  $\Theta(n^0 \log^0 n)$ . So,  $k=0$  and  $p=0$

**Step 02:**  $\log_b a = \log_2 2 = 1$  and  $k = 0$

**Step 03:** From step 02,  $\log_b a > k$

**Step 04:** From SMT (Simplified Master theorem) Case 1:

$$T(n) = \Theta(n^{\log_b a})$$

$$\text{Hence, } T(n) = \Theta(n^1)$$

$$= \Theta(n)$$

# Simplified version Case 2 (a)

## Example

*Case 2: if  $\log_b a = k$ , then  
(a) if  $p > -1$ , then  $T(n) = \Theta(n^k \log^{p+1} n)$*

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

Solution:

$T(n) = aT(n/b) + f(n)$  where  $f(n) = \Theta(n^k \log^p n)$

**Step 01:** Here,  $a = 2$ ,  $b = 2$ ,  $k=1$  and  $p=0$

**Step 02:**  $\log_b a = \log_2 2 = 1$  and  $k = 1$

**Step 03:** From step 02,  $\log_b a = k$  so goto case 2

**Step 04:**  $p > -1$  so goto 2(a) of simplified masters theorem.

So,  $T(n) = \Theta(n^k \log^{p+1} n)$

Hence,  $T(n) = \Theta(n^1 \log^{0+1} n)$   
 $= \Theta(n \log n)$

# Simplified version Case 2 (b)

## Example

---

*Case 2 : if  $\log_b a = k$ , then  
 (b) if  $p = -1$ , then  $T(n) = \Theta(n^k \log \log n)$*

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

Solution:

$T(n) = aT(n/b) + f(n)$  where  $f(n) = \Theta(n^k \log^p n)$

**Step 01:** Here,  $a = 2$ ,  $b = 2$ ,

$n/\log n$  is  $n^1 \cdot \log^{-1} n$  so  $k = 1$  and  $p = -1$

**Step 02:**  $\log_b a = \log_2 2 = 1$  and  $k = 1$

**Step 03:** From step 02,  $\log_b a = k$  so goto Case 2

**Step 04:**  $P = -1$  So, goto simplified Masters Theorem Case 2b.

So,  $T(n) = \Theta(n^k \log \log n)$

Hence,  $T(n) = \Theta(n^1 \log \log n)$   
 $= \Theta(n \log \log n)$

# Simplified version Case 2 (c) Example



Case 2 : if  $\log_b a = k$ , then  
(c) if  $p < -1$ , then  $T(n) = \Theta(n^k)$

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

Solution:

$T(n) = aT(n/b) + f(n)$  where  $f(n) = \Theta(n^k \log^p n)$

**Step 01:** Here,  $a = 2$ ,  $b = 2$ ,

$n/\log^2 n$  is  $n^1 * \log^{-2} n$  so  $k = 1$  and  $p = -2$

**Step 02:**  $\log_b a = \log_2 2 = 1$  and  $k = 1$

**Step 03:** From step 02,  $\log_b a = k$  so goto Case 2

**Step 04:**  $P < -1$  So, goto simplified Masters Theorem Case 2c.

So,  $T(n) = \Theta(n^k)$

Hence,  $T(n) = \Theta(n^1)$   
 $= \Theta(n)$

# Simplified version Case 3 (a)

## Example

*Case 3 : if  $\log_b a < k$ , then*  
 (a) *if  $p \geq 0$ , then  $T(n) = \Theta(n^k \log^p n)$*

**Consider**  $T(n) = 4T(n/2) + n^3 \log n$

Solution:

$T(n) = aT(n/b) + f(n)$  where  $f(n) = \Theta(n^k \log^p n)$

**Step 01:** Here,  $a = 4$ ,  $b = 2$ ,  $k = 3$  and  $p = 1$

**Step 02:**  $\log_b a = \log_2 4 = 2$  and  $k = 3$

**Step 03:** From step 02,  $\log_b a < k$  so goto Case 3

**Step 04:**  $P > 0$  So, goto simplified Masters Theorem  
Case 3a.

So,  $T(n) = \Theta(n^k \log^p n)$

$$\begin{aligned}\text{Hence, } T(n) &= \Theta(n^3 \log^1 n) \\ &= \Theta(n^3 \log n)\end{aligned}$$

# Simplified version Case 3 (b)



## Example

Case 3 : if  $\log_b a < k$ , then  
(b) if  $p < 0$ , then  $T(n) = \theta(n^k)$

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

Solution:

$T(n) = aT(n/b) + f(n)$  where  $f(n) = \theta(n^k \log^p n)$

**Step 01:** Here,  $a = 2$ ,  $b = 2$ ,  $k = 2$  and  $p = -2$  (because  $n^2 * \log^{-2} n$ )

**Step 02:**  $\log_b a = \log_2 2 = 1$  and  $k = 2$

**Step 03:** From step 02,  $\log_b a < k$  so goto Case 3

**Step 04:**  $P < 0$  So, goto simplified Masters Theorem Case 3b.

So,  $T(n) = \theta(n^k)$

$$\begin{aligned}\text{Hence, } T(n) &= \theta(n^2) \\ &= \theta(n^2)\end{aligned}$$

# Master Method Capability

---

- Not all recurrence relations can be solved with the use of the master theorem i.e. if
  - $T(n)$  is not of the form as mentioned earlier, ex:  $T(n) = \sin n$
  - $f(n)$  is not a polynomial, ex:  $T(n) = 2T(n/2) + 2^n$

# Exercises

---

Find recursive solution for following problems:

1. Return Nth Fibonacci Number
2. Power of a given number
3. Recursive array search
4. Sum of elements of an array using recursive function

# Exercises

---

- ❖ Try to run the Practice sheet 1 to get a feel of order of growth,
- ❖ Solve the below recurrence relations using the iterative substitution method:

$$1) \quad T(n) = \begin{cases} 1 & ; n = 1 \\ n + T(n-1) & ; n \geq 1 \end{cases}$$

$$2) \quad T(n) = \begin{cases} 1 & ; n = 1 \\ 2T(n-1) & ; n \geq 1 \end{cases}$$

- ❖ Tree using recurrence tree for:  $T(n) = T(n/4) + T(n/2) + n^2$
- ❖ Solve the below recurrence relations masters theorem :

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

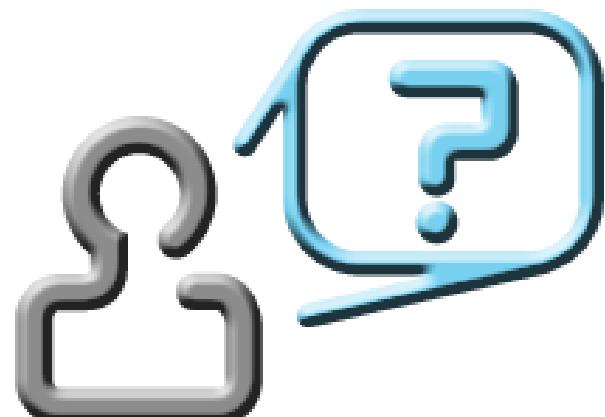
b)  $T(n) = 9T(n/3) + n$

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

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

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

f)  $T(n) = 9T(n/3) + 1$



---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# **Data Structures and Algorithms Design**

**DSECLZG519**

**Parthasarathy**



# **DSECLZG519 – Contact Session #3**

## **Linear Data Structures**

# Agenda for CS #3

- 1) Recap of CS#2
- 2) Simplified Master's Theorem
- 3) Abstract Data type
- 4) Stacks
  - o Stack implementation using array
  - o Stack applications
    - o Expression Evaluation
    - o Infix to postfix conversion [Self-Study] – Available in appendix of this presentation
- 5) Queues
  - o Queue implementation using array
  - o Circular Queue
  - o Applications of Queue
- 6) Linked List
  - o Singly Linked list
  - o Doubly Linked list – Insertion & Deletion
- 7) Stack & Queue implementation using linked list
- 8) A word on Amortized analysis
- 9) Q&A

# Recap of CS#2

---

- General Rules and Problems
- Mathematical analysis of Non-recursive algorithms
- Mathematical analysis of Recursive algorithms
  - Iteration vs Recursion
  - Recurrence Tree
  - Iterative Substitution
- Masters Theorem
  - Masters method with an example for each case

# Abstract Data Type

---

- *The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc., the space like integer will take 4-bytes, character will take 1-byte of space etc.*
- The **abstract datatype** is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes (we can perform different operations), But how those operations are working is totally hidden from the user. The ADT is made up of primitive datatypes, but operation logics are hidden.
- Abstract Data Type :
  - In computer science, an **abstract data type (ADT)** is a mathematical model for a certain class of data structures that have similar behavior.
  - A method for achieving abstraction for data structures and algorithms
  - $\text{ADT} = \text{model} + \text{operations}$
  - Describes what each operation does, but not how it does it
  - An ADT is independent of its implementation

# Operations on Data

- Typical operations on data:
  - Add data to a data collection (insert)
  - Remove data from a data collection (delete)
  - Ask questions about the data in a data collection (search, exists, update etc.)

# Linear Data Structures

- Data is organized in a linear fashion.
- Simple ADTs, example :
  - *Stack*
  - *Queue*
  - Vector
  - Lists
  - Sequences

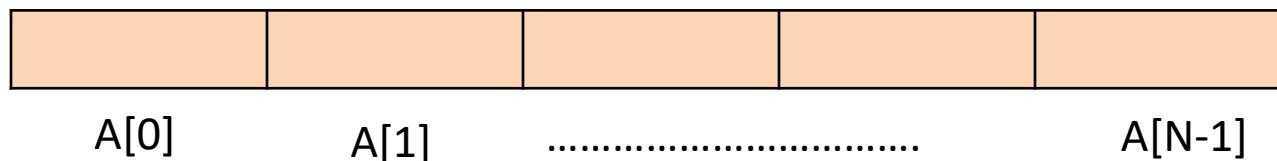
All these are called *Linear Data Structures*

# Non-linear Data Structures

- Data is not organized in a linear fashion.
- Relationship exists among data.
- Examples :
  - Trees
  - Graphs

# Arrays

- Array is a finite set of homogeneous elements stored in adjacent/contiguous memory locations. It is represented by a single name.
- Each location element is identified by a single name with its index. The first element is  $A[0]$ , the second element is  $A[1]$  and so on ..
- If there are  $N$  elements in the array, the last element will be  $A[N-1]$ .



# Stacks

---

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as “**pushing**” onto the stack. “**Popping**” off the stack is synonymous with removing an item.
- Pushing and popping happens only in **one end** called the top.

# Stacks

innovate

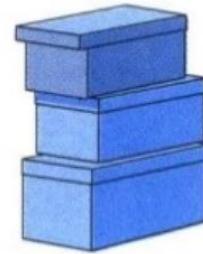
achieve

lead

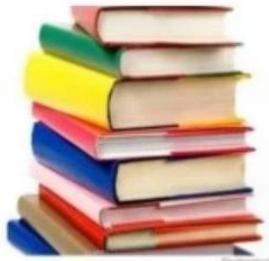
A stack of cafeteria trays



A stack of shoe boxes



A stack of neatly folded shirts



# Stacks: An Array Implementation

- Create a stack  $S$  using an array by specifying a maximum size  $N$  for our stack.
- The stack consists of an  $N$ -element array  $S$  and an integer variable  $t$ , the index of the top element in array  $S$ .

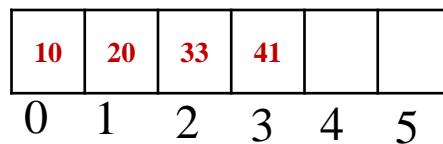


- Array indices start at 0, so we initialize  $t$  to -1

# Stacks: An Array Implementation

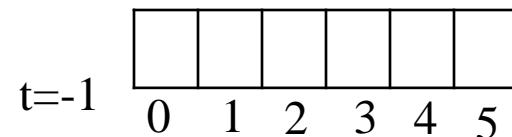
## Pseudo code

```
Algorithm size()
return t+1
```



```
Algorithm isEmpty()
```

```
if t == -1
  return true
return false
```



```
Algorithm top()
if isEmpty() then
  return Error
return S[t]
```

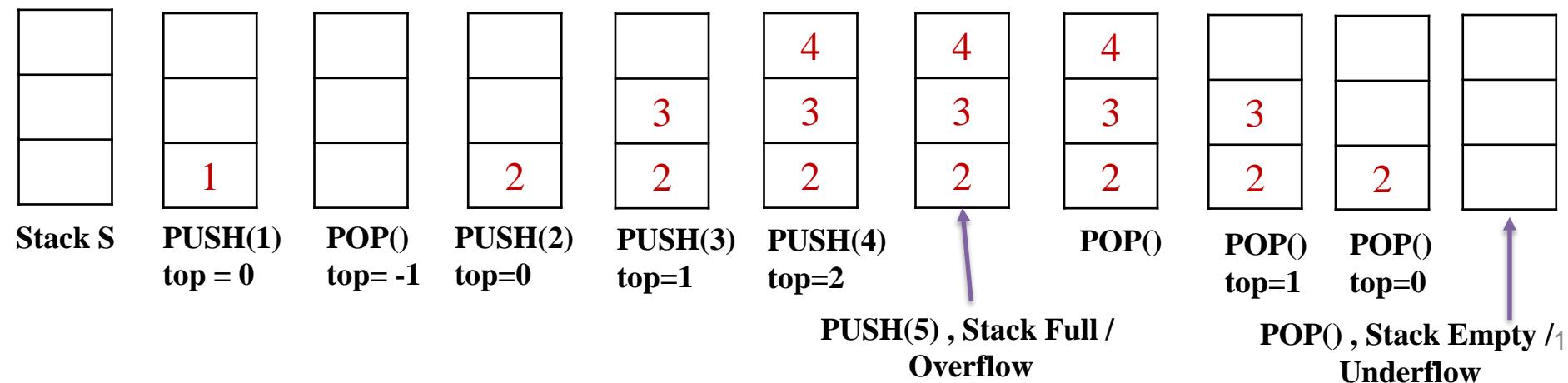
```
Algorithm push(o)
if size() == N then
  return Error/Overflow
t=t+1
S[t]=o
```

```
Algorithm pop()
if isEmpty() then
  return Error/Underflow
e= S[t]
S[t] = null
t=t-1
return e
```

# Operations on Stack Demo

Consider array based implementation of stack S. Assume that the maximum size of the stack is 3 elements. Show the contents of the STACK (trace through) at each step, for the following sequence of operations. Mention exceptions like empty/full if any.

Operations: PUSH (1), POP, PUSH (2), PUSH (3), PUSH (4), PUSH (5), POP, POP, POP, POP



# Stacks: An Array Implementation

---

The array implementation is simple and efficient  
(methods performed in  $O(1)$ ).

## Disadvantage

- There is an upper bound,  $N$ , on the size of the stack.
- The arbitrary value  $N$  may be too small for a given application  
**OR** a waste of memory.

# Applications of Stack

---

- Stacks can be used for expression evaluation. *[In class]*
- Stacks can be used for Conversion from one form of expression to another. *[In Appendix of this presentation]*
- Stacks can be used to check parenthesis matching in an expression. *[Self-study/Webinar]*
- Stack data structures are used in backtracking problems (recursion). *[Explore!]*

# Forms of Expression

Arithmetic expressions can be written in 3 different forms or notations :

- **Infix Notation** : Notation in which the operator symbol is placed in between its operands. Ex :  $A + B$  ,  $C - D$  ,  $A * D$  etc.
- **Pre-fix Notation** : It is also called as Polish notation and refers to the notation in which the operator symbol is placed **before** its operands (*when operator precedes the operands*). Ex :  $+AB$ ,  $-CD$ ,  $*AD$  etc.
- **Post-fix Notation** : It is also called as reverse polish notation and refers to the notation in which the operator symbol is placed **after** its operands (*when operator follows the operands*). Ex :  $AB+$  ,  $CD-$  ,  $AD^*$

# Algorithm: Evaluation of Postfix expression using Stack

## Algorithm:

- 1) Add a right parentheses ")" at the end of the arithmetic expression F
- 2) Scan F from left to right and repeat step 3 and step 4 for each element of F until the sentinel ")" is encountered.
- 3) If an operand is encountered, put it onto stack
- 4) If an operator  $\textcircled{x}$  is encountered, then :
  - a. Remove the 2 top elements from the stack, where n1 is the top element and n2 is the next-to-top element.
  - b. Evaluate  $n_2 \textcircled{x} n_1$
  - c. Place the result of (b) back on stack.
- Endif
- End of step 2 loop
- 5) Set value equal to the top element of stack.
- 6) Exit



: is any operator

# Example: Evaluation of Postfix expression using stack

Evaluate  $AB+C-BA+C^{\wedge}-$  where  $A=1, B=2$  and  $C=3$

→  $1\ 2\ +\ 3\ -\ 2\ 1\ +\ 3\ ^{\wedge}\ -\ )$

| Symbol Encountered | N1 (top) | N2(2 <sup>nd</sup> top) | Value = N2 op N1   | STACK |
|--------------------|----------|-------------------------|--------------------|-------|
| 1                  |          |                         |                    | 1     |
| 2                  |          |                         |                    | 1 2   |
| +                  | 2        | 1                       | $1+2 = 3$          | 3     |
| 3                  |          |                         |                    | 3 3   |
| -                  | 3        | 3                       | $3 - 3 = 0$        | 0     |
| 2                  |          |                         |                    | 0 2   |
| 1                  |          |                         |                    | 0 2 1 |
| +                  | 1        | 2                       | $2+1 = 3$          | 0 3   |
| 3                  |          |                         |                    | 0 3 3 |
| $^{\wedge}$        | 3        | 3                       | $3^{\wedge}3 = 27$ | 0 27  |
| -                  | 27       | 0                       | $0 - 27 = -27$     | -27   |
| )                  | -        | -                       | <b>-27</b>         | Empty |

Hence, the evaluation of this expression leads to the answer **-27**

# Exercises

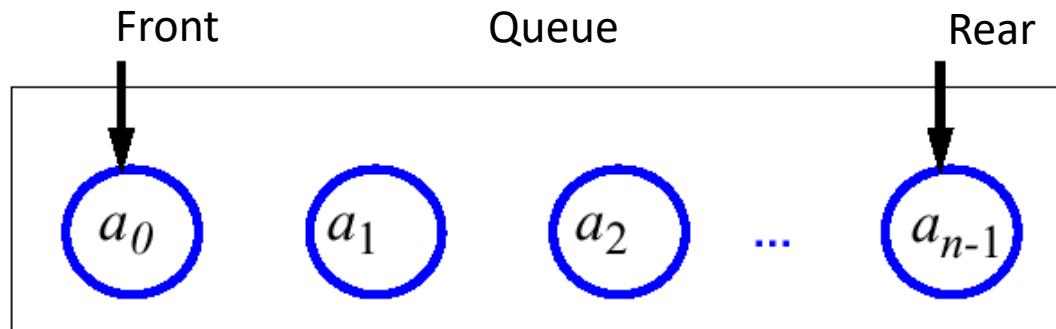
Evaluate the following postfix expressions :

- $ABC+*CBA-+*$  where A=1,B=2 and C=3
- $5\ 7\ 5\ -\ *\ 12\ 4\ *\ 24\ / \ 6\ +\ +$

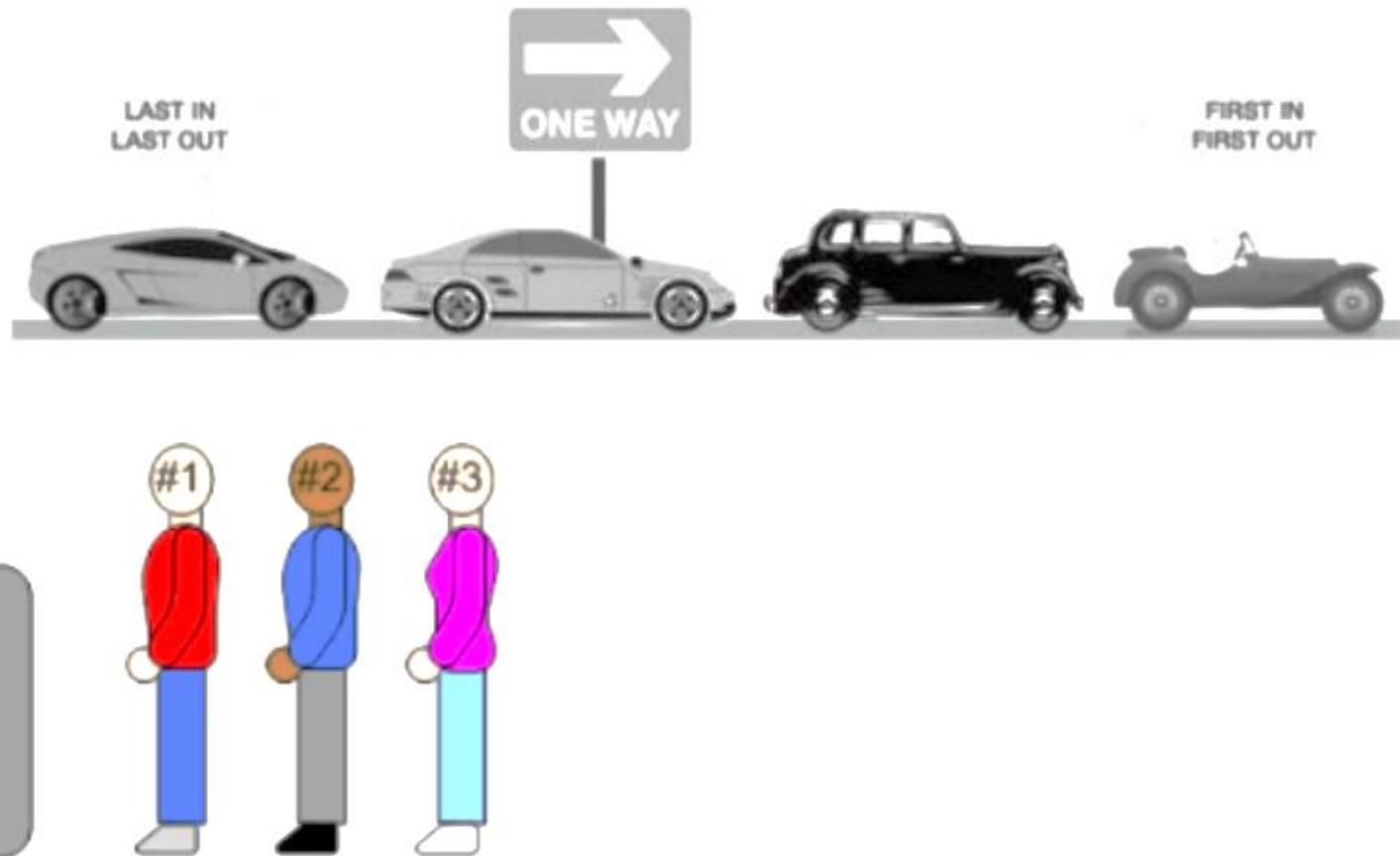
After you solve it, verify your answer by putting the postfix expression in this tool  
<https://www.free-online-calculator-use.com/postfix-evaluator.html>

# Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



# Queues



# Queues

The **queue** supports three fundamental methods:

- **Enqueue(S:ADT, o:element):** - Inserts object  $o$  at the rear of the queue; an error occurs if the queue is full
- **Dequeue(S:ADT):** - Removes the object from the front of the queue; an error occurs if the queue is empty
- **Front(S:ADT):element** - Returns, but does not remove, the front element; an error occurs if the queue is empty

# Queues: An Array Implementation

- Create a queue using an array
- A maximum size  $N$  is specified.
- The queue consists of an  $N$ -element array  $Q$  and two integer variables:
  - $f$ , index of the front element (which is the candidate to be removed by a dequeue operation)
  - $r$ , index of the next available array cell (cell which can be used in enqueue operation)
  - Initially,  $f=r=0$  and the queue is empty if  $f=r$



# Queues

## Disadvantage

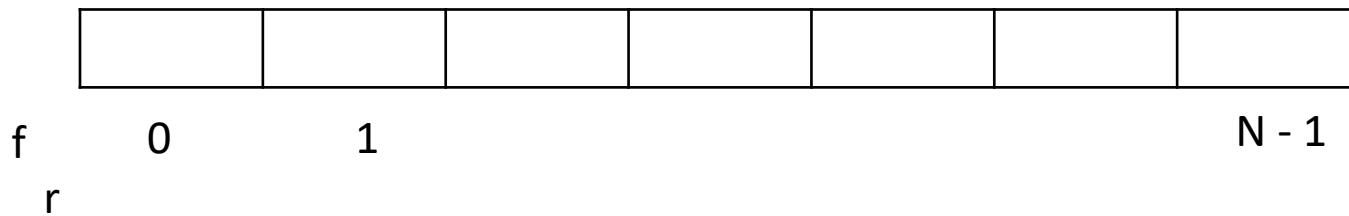
Repeatedly enqueue and dequeue a single element  $N$  times.

Finally,  $f=r=N$ .

- No more elements can be added to the queue, *though there is space in the queue !!*

## Solution

Let  $f$  and  $r$  wraparound the end of queue (circular queue).

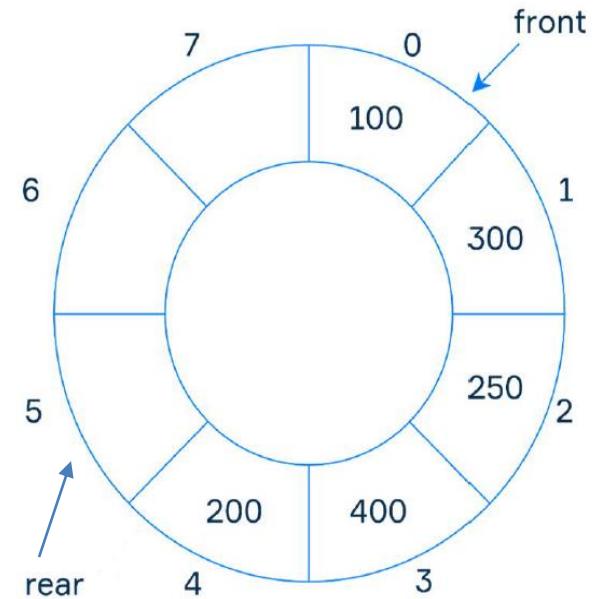


# Queues: An Array Implementation

- “wrapped around” configuration



- Each time  $r$  or  $f$  is incremented, compute this increment as  $r = (r+1) \bmod N$  or  $f = (f+1) \bmod N$
- When array is full  $f = r$  (same as empty condition), then how do we know if it's a queue full or queue empty ? There are many ways to handle this (Explore this!!). Simple approach to handle this, insist that  $Q$  can never hold more than  $N-1$  objects.



# Queues: An Array Implementation

## Pseudo code

Algorithm **size()**  
**return**  $(N-f+r) \bmod N$

Algorithm **isEmpty()**  
**return**  $(f=r)$

Algorithm **front()**  
**if** isEmpty() **then**  
**return** Error  
**return**  $Q[f]$

Algorithm **dequeue()**  
**if** isEmpty() **then**  
**return** Error  
 $o=Q[f]$   
 $Q[f]=null$   
 $f=(f+1) \bmod N$   
**return**  $o$

Algorithm **enqueue(o)**  
**if** size() =  $N - 1$  **then**  
**return** Error  
 $Q[r]=o$   
 $r=(r + 1) \bmod N$

# Applications of Queue

---

- Used in CPU scheduling
- Used in Disk scheduling
- Priority queue is used in heaps [We will look at this in later session]
- ...
- ...

# Arrays: Pluses and minuses

---

- + Fast element access.
  - Impossible to resize.
- 
- Many applications require resizing!
  - Required size not always immediately available.

# List ADT

- A sequence of items where positional order matter
- $\langle a_1, a_2, \dots, a_{n-1}, a_n \rangle$
- Lists are very general in computing
- e.g. student list, list of events, list of appointments etc

# List Operations

---

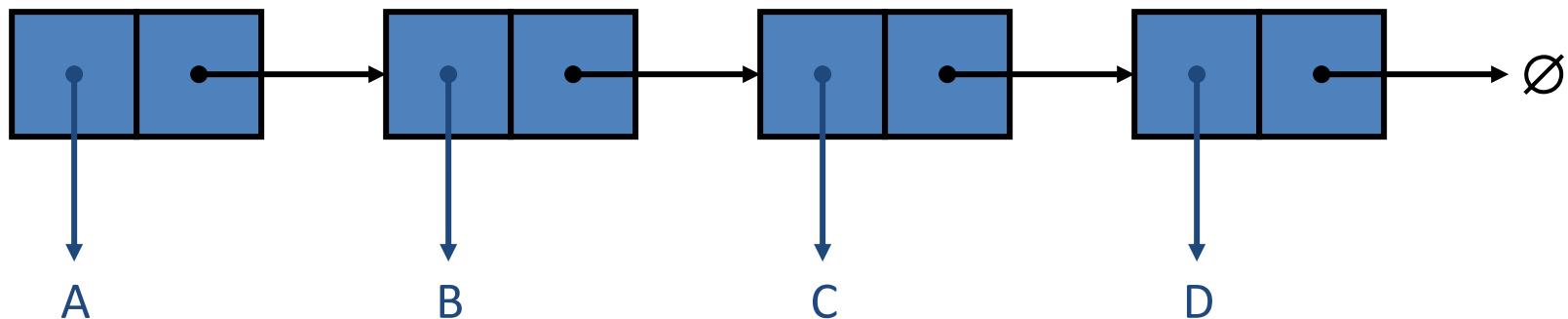
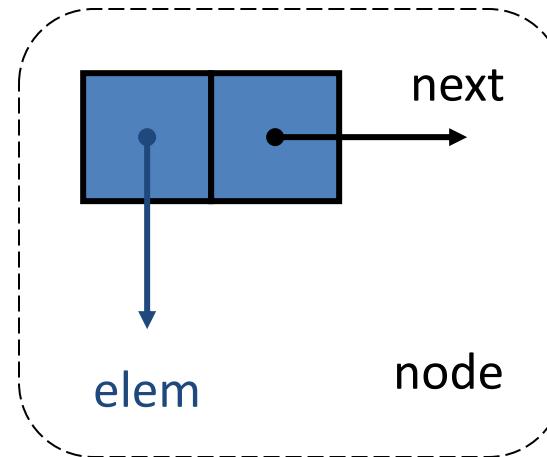
- Supports the following methods for a list S:refer to relative positions in the list
- first ( ):Return the position of the first element of S; an error occurs if S is empty.
- last() :Return the position of the last element of S; an error occurs if S is empty.
- isFirst(p ) :Return a Boolean value indicating whether the given position is the first one in the list.
- islast (p )
- before(p) :Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.
- after (p) ,size(),isEmpty(),insertAfter(), remove()

# Singly Linked Lists

A singly linked list is a concrete data structure consisting of a sequence of nodes

Each node stores

- element
- link to the next node



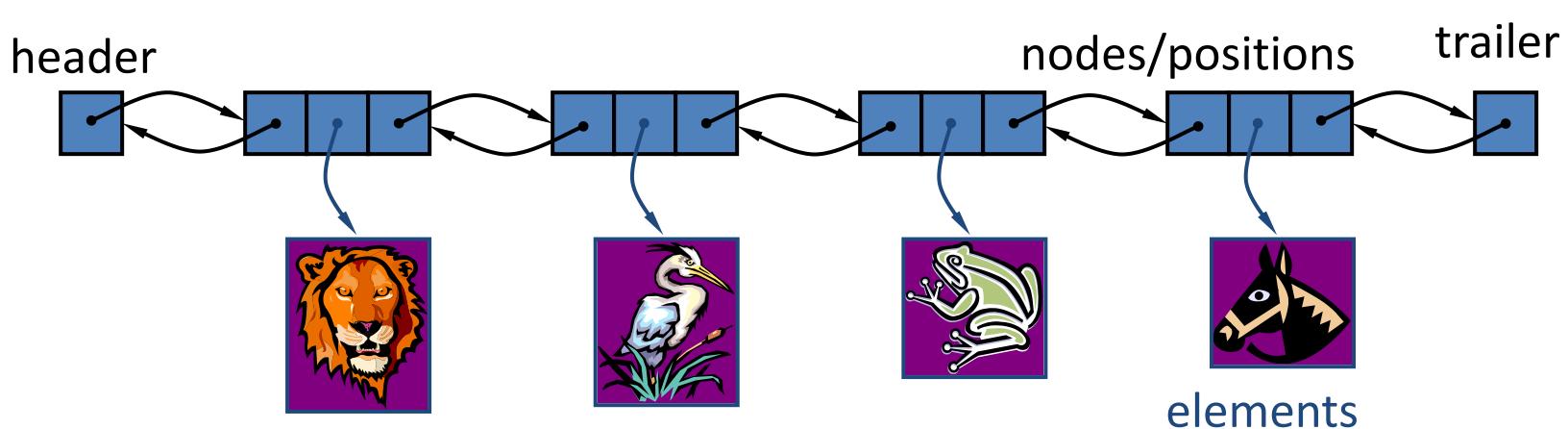
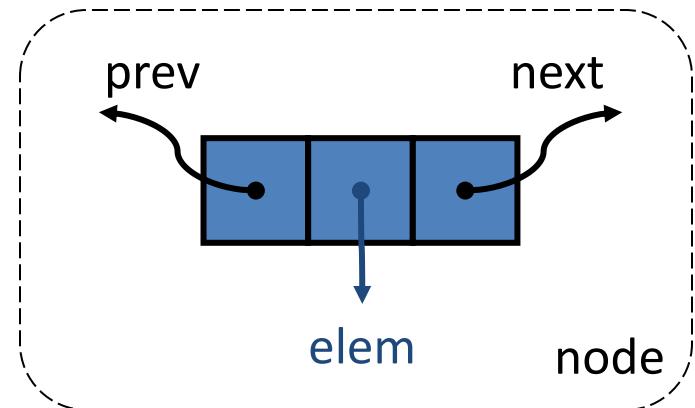
# Doubly Linked List

A doubly linked list is often more convenient!

Nodes store:

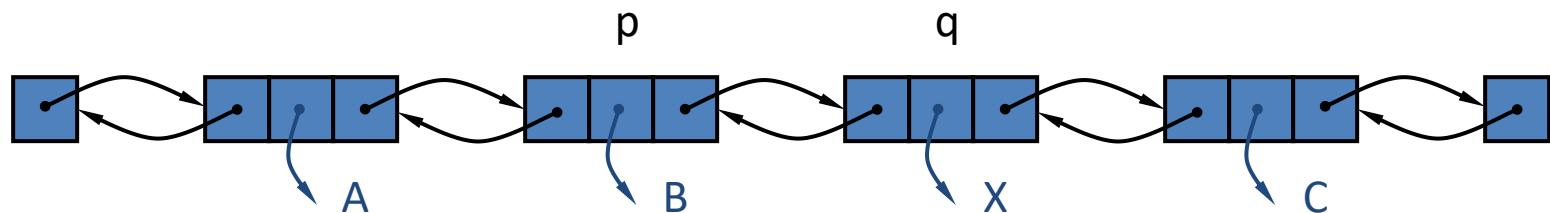
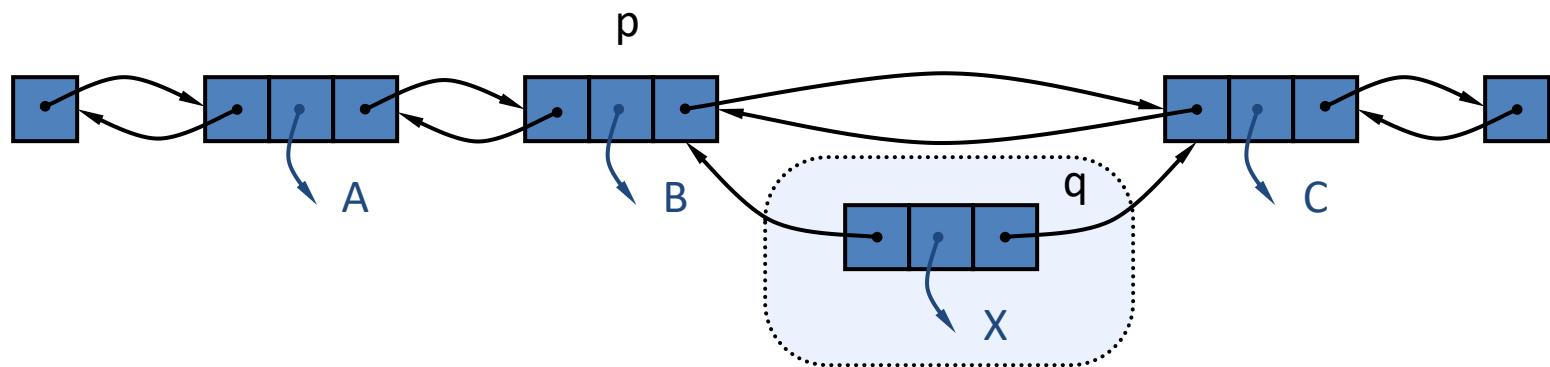
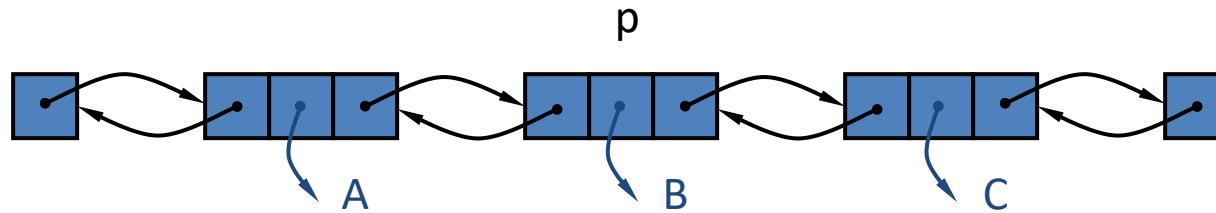
- element
- link to the previous node
- link to the next node

Special trailer and header nodes



# Insertion

We visualize operation `insertAfter(p, X)`:



# Insertion Algorithm

**Algorithm** insertAfter( $p, e$ ):

Create a new node  $v$

$v.\text{setElement}(e)$

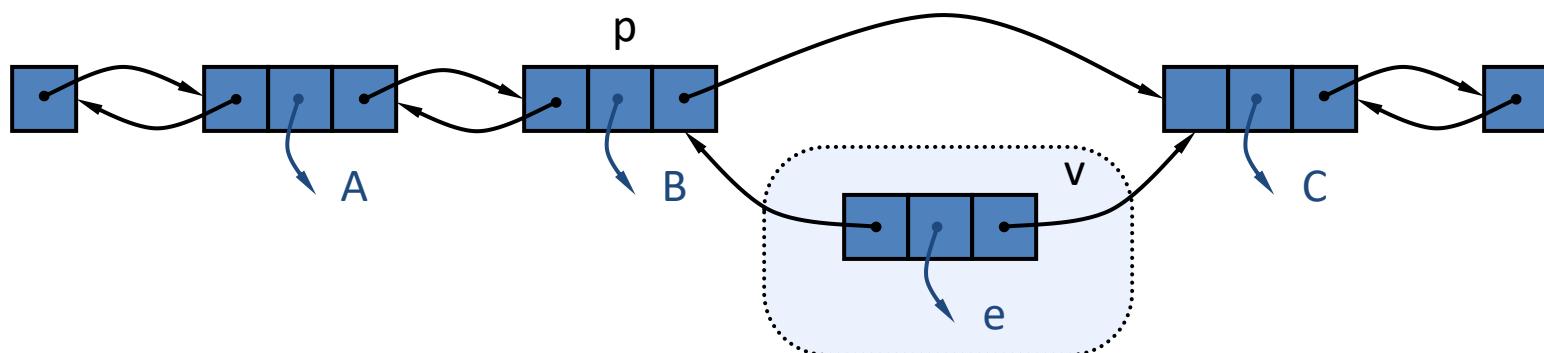
$v.\text{setPrev}(p)$  {link  $v$  to its predecessor}

$v.\text{setNext}(p.\text{getNext})$  {link  $v$  to its successor}

$(p.\text{getNext}).\text{setPrev}(v)$  {link  $p$ 's old successor to  $v$ }

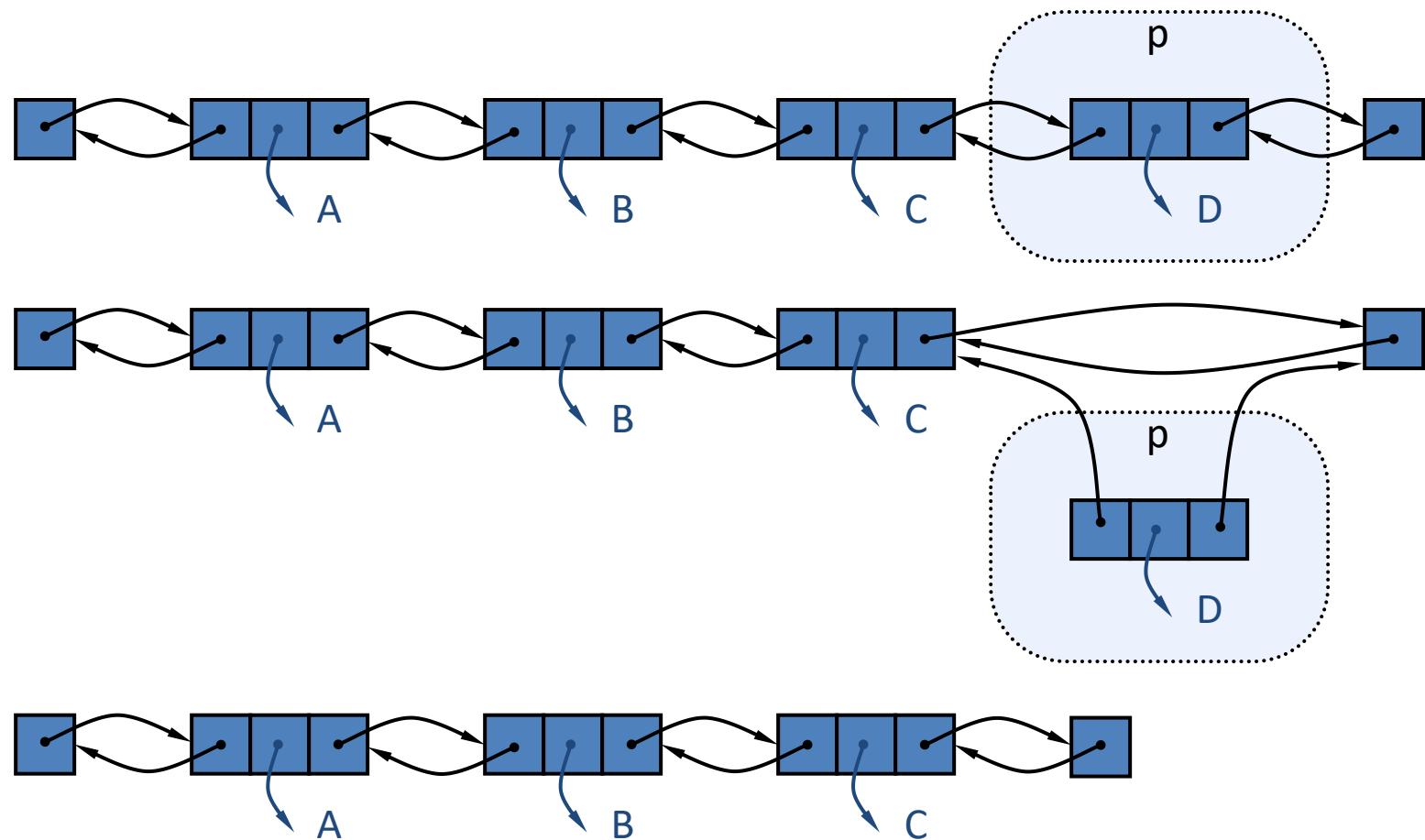
$p.\text{setNext}(v)$  {link  $p$  to its new successor,  $v$ }

**return**  $v$  {the position for the element  $e$ }



# Deletion

- We visualize  $\text{remove}(p)$

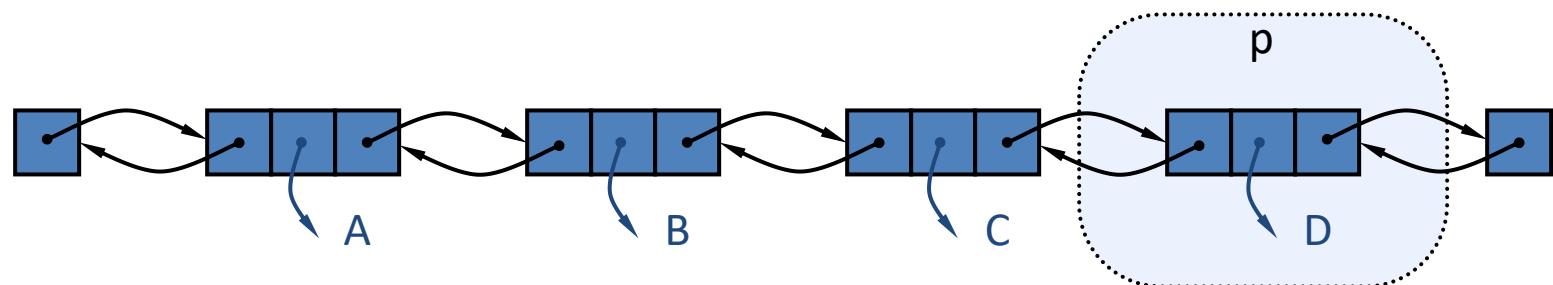


# Deletion Algorithm

**Algorithm** remove( $p$ ):

```

 $t = p.\text{element}$            {a temporary variable to hold the return value}
 $(p.\text{getPrev()}).\text{setNext}(p.\text{getNext()})$  {linking out  $p$ }
 $(p.\text{getNext()}).\text{setPrev}(p.\text{getPrev()})$  {linking out  $p$ }
 $p.\text{setPrev(null)}$        {invalidating the position  $p$ }
 $p.\text{setNext(null)}$ 
return  $t$ 
```



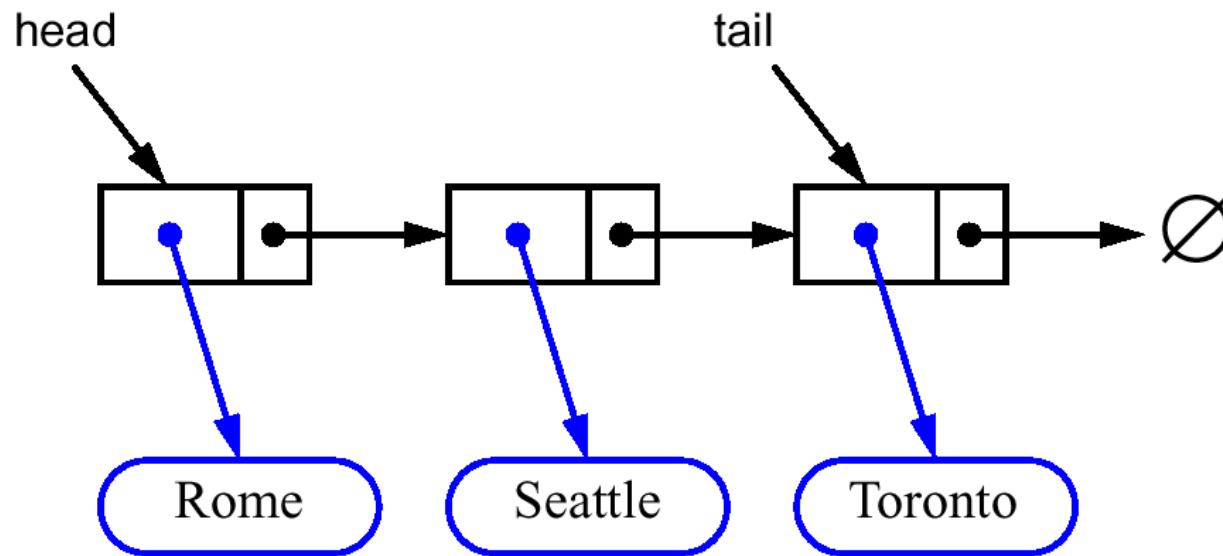
# Worst-case running time

In a doubly linked list

- + insertion at head or tail is in  $O(1)$
- + deletion at either end is on  $O(1)$
- element access is still in  $O(n)$

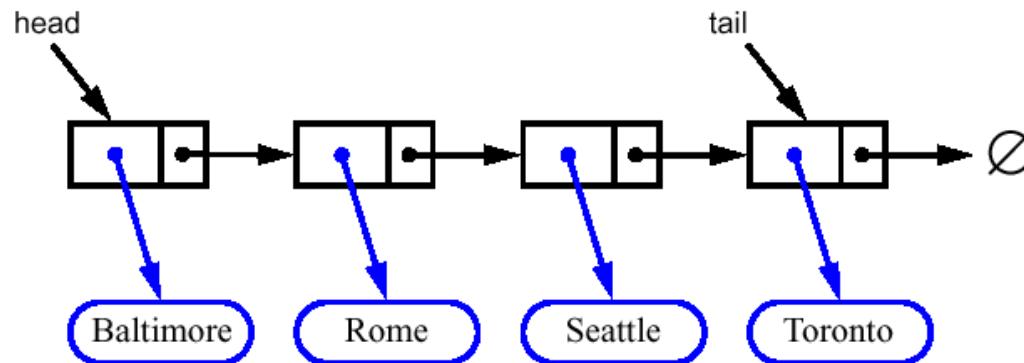
# Stacks: Singly Linked List implementation

- Nodes (*data, pointer*) connected in a chain by links

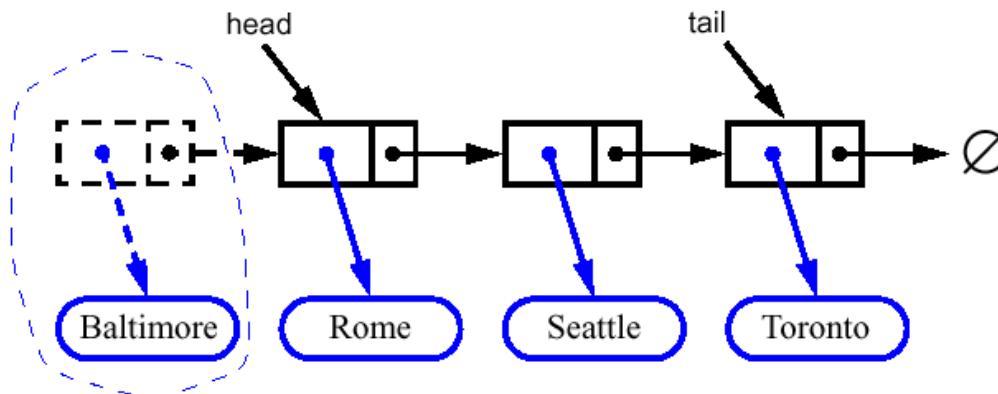


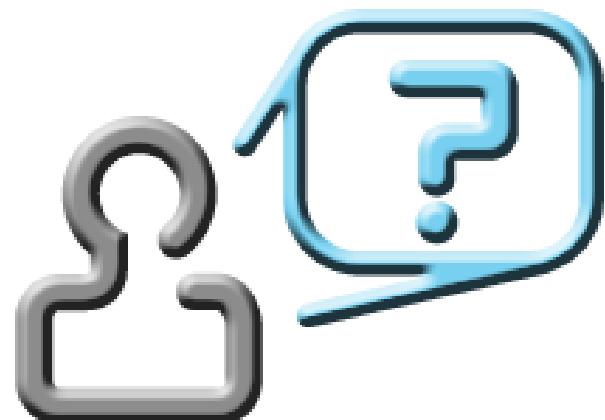
- the head or the tail of the list could serve as the top of the stack

# Queues: Linked List Implementation



- Dequeue - advance head reference





# Appendix : Forms of Expression

Examples :

| # | Infix           | Prefix       | Postfix        |
|---|-----------------|--------------|----------------|
| 1 | $(A+B) * C$     | $*+ABC$      | $AB+C*$        |
| 2 | $A + ( B * C )$ | $+A*BC$      | $ABC*+$        |
| 3 | $A ^ B * C - D$ | $- * ^ ABCD$ | $AB ^ C * D -$ |
| 4 | $(A+B)/(C-D)$   | $/+AB-CD$    | $AB+CD-/$      |

- The computer usually evaluates an arithmetic expression in infix notation in 2 steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In both these steps, the stack is used !! Let us examine how stack is used ☺

# Appendix : Algorithm: Infix to Postfix using Stack

The following algorithm transforms the infix expression Q into its equivalent postfix expression P.

- 1) Push "(" onto STACK, and add ")" to the end of Q
- 2) scan Q from left to right and repeat steps 3 to 6 for each element of Q until the stack is empty
  - 3) if an operand is encountered add it to P
  - 4) if a left parenthesis is encountered push it onto the stack
  - 5) if an operator is encountered , then
    - (a) Repeatedly pop from STACK and add to P each operator (on top of the STACK) which has same precedence as or higher precedence than the operator encountered
    - (b) Add the encountered operator to the stack  
[end of IF structure]
  - 6) if a right parenthesis is encountered, then:
    - (a) repeatedly pop from the stack and add to P each operator (on top of the STACK) until a left parenthesis is encountered
    - (b) remove the left parenthesis [do not add left parenthesis to P]  
  
[End of IF structure]  
[End of step 2 loop]
  - 7) Exit

# Appendix : Example : Converting Infix to Postfix using Stack

Consider the infix expression Q to be  $A + B * C \rightarrow A+B*C$

| # | Symbol Encountered | Postfix String P (Output) | STACK Contents |
|---|--------------------|---------------------------|----------------|
| 1 |                    |                           | (              |
| 2 | A                  | A                         | (              |
| 3 | +                  | A                         | (+             |
| 4 | B                  | AB                        | (+             |
| 5 | *                  | AB                        | (+*            |
| 6 | C                  | ABC                       | (+*            |
| 7 | )                  | ABC*                      | (+             |
| 8 | -                  | ABC*+                     | (              |
| 9 | -                  | ABC*+                     | Empty          |

Hence, the postfix expression P is  $ABC*+$

# Appendix : Example : Converting Infix to Postfix using Stack



Consider the infix expression Q to be  $(A+B) * C \rightarrow (A+B) * C$

| #  | Symbol Encountered | Postfix String P<br>(Output) | STACK Contents |
|----|--------------------|------------------------------|----------------|
| 1  |                    |                              | (              |
| 2  | (                  |                              | ((             |
| 3  | A                  | A                            | ((             |
| 4  | +                  | A                            | ((+            |
| 5  | B                  | AB                           | ((+            |
| 6  | )                  | AB+                          | (              |
| 7  | *                  | AB+                          | (*             |
| 8  | C                  | AB+C                         | (*             |
| 9  | )                  | AB+C*                        | Empty          |
| 10 | -                  | AB+C*                        | Empty          |

Hence, the postfix expression P is **AB+C\***

# Appendix Exercise

1) Convert the following infix expressions into postfix expressions using stack ( trace it as we did in class ) :

- $A + (B + C) * D$
- $B + C - D / E * F$
- $(A + B) + (C / D) * E$
- $A + B * (C + D - E) * F$

After you solve it, verify your answer by putting the infix expression in this tool :

<https://www.mathblog.dk/tools/infix-postfix-converter/>

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : [CC BY-NC-SA 4.0](#)





**BITS Pilani**

Pilani | Dubai | Goa | Hyderabad

# Data Structures and Algorithms Design

**DSECLZG519**

**Parthasarathy**





# Contact Session #4

# DSECLZG519 – Introduction to Tree

# Agenda for CS #4

---

- 1) Recap of CS#3
- 2) Introduction to Trees

- Tree terminologies
- Binary tree and types
- Representation of tree
  - Using array
  - Using linked list
- Binary tree traversals
  - Inorder
  - Preorder
  - Postorder

- 3) Applications of binary trees
- 4) Q&A

# Linear data structures

Here are some of the data structures we have studied so far:

- Arrays
- Singly-linked lists and doubly-linked lists
- Stacks and queues
- These all have the property that their elements can be adequately displayed in a straight line
- How to obtain data structures for data that have non-linear relationships?

# How should one decide which data structure to use ?

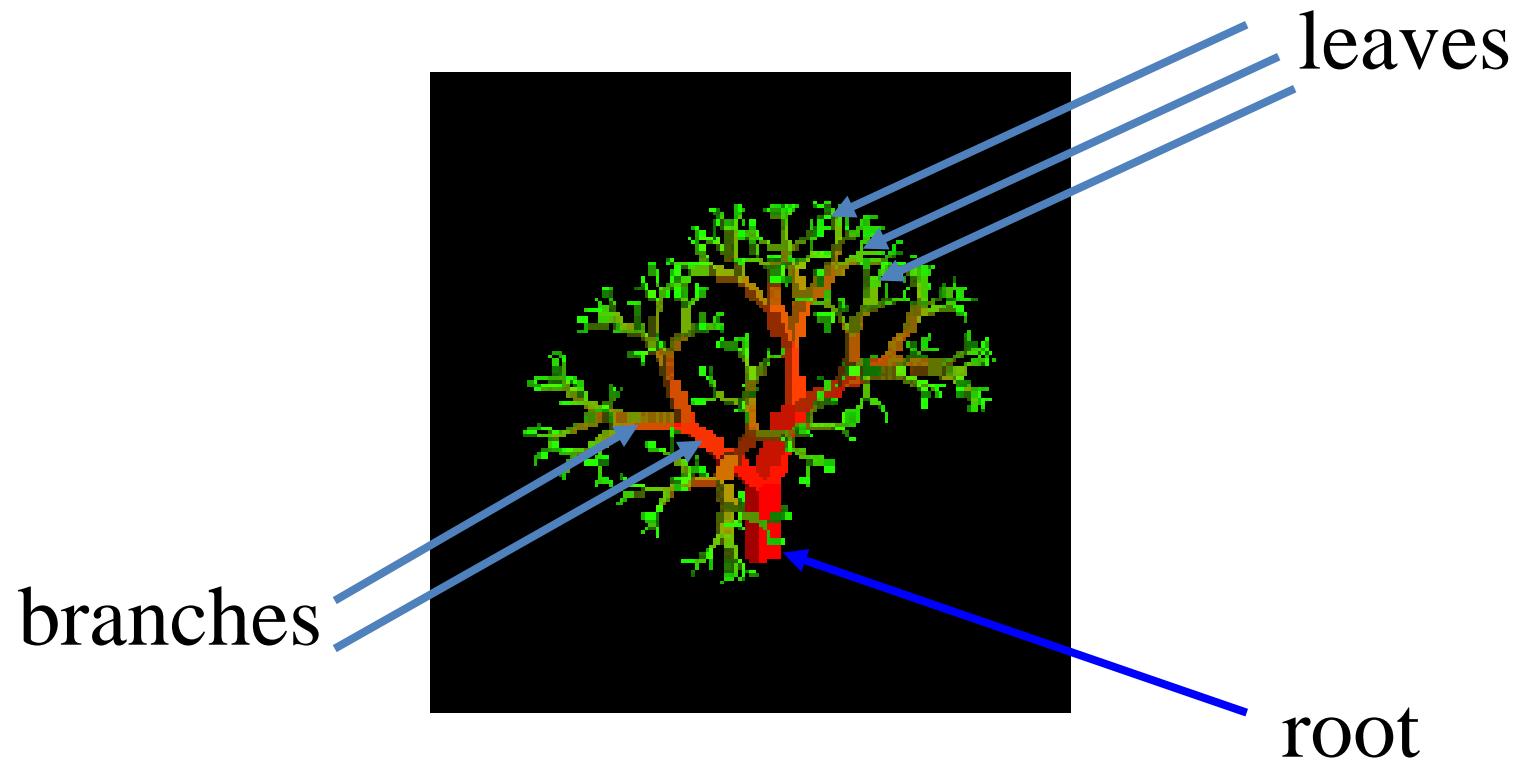
---

- What need to be stored
- Cost of operations
- Memory use
- Ease of implementation ☺

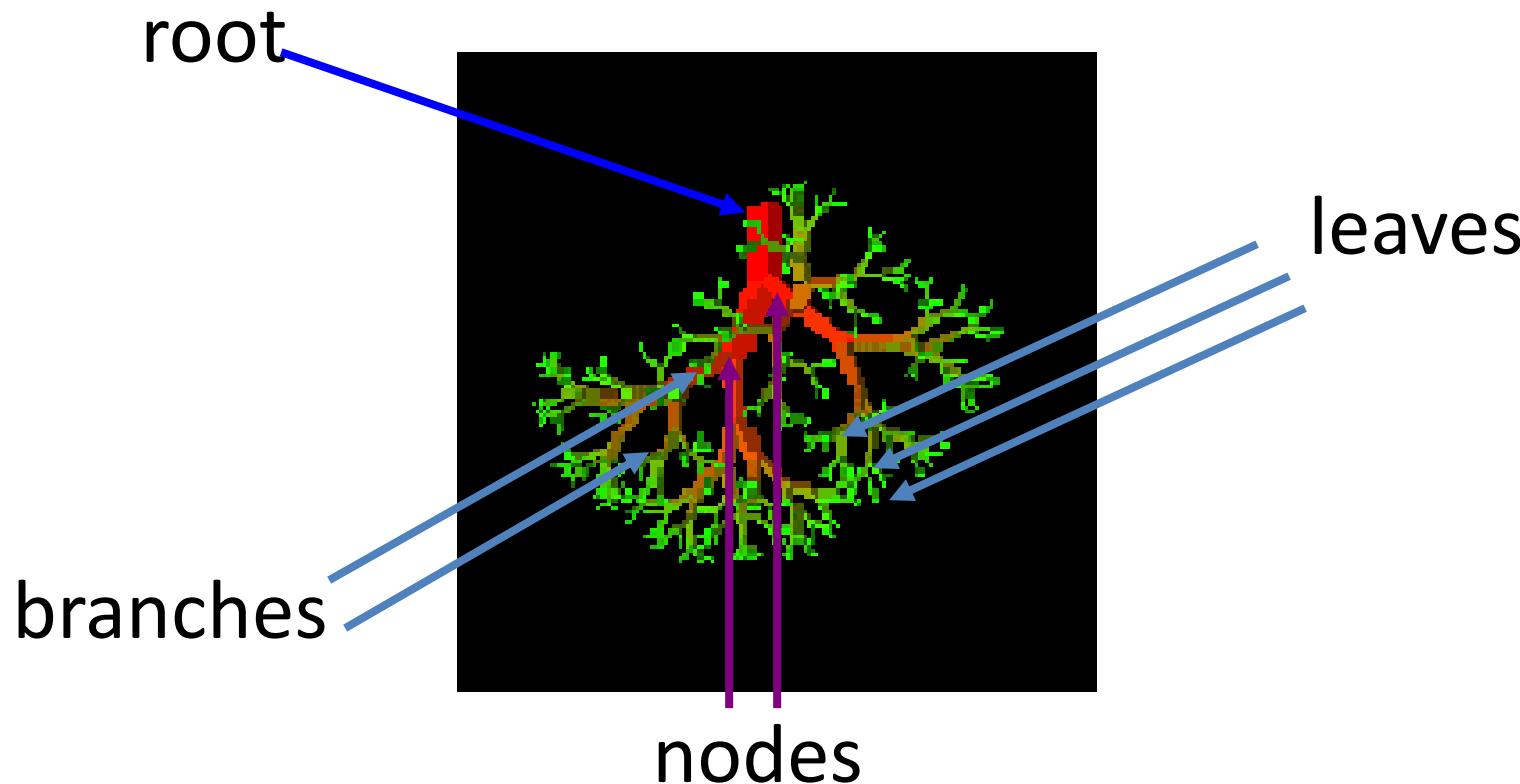
# Non-linear Data Structures

- Linear access time of linked lists is expensive
  - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is  $O(\log N)$ ?
- Trees
  - Basic concepts
  - Tree traversal
  - Binary tree
  - Binary search tree and its operations
- Stores elements hierarchically
- Top element is called as **root**
- Example
  - Directory structure
  - Family tree
  - Organizational structure , Content of a book etc.

# Natural View of a Tree

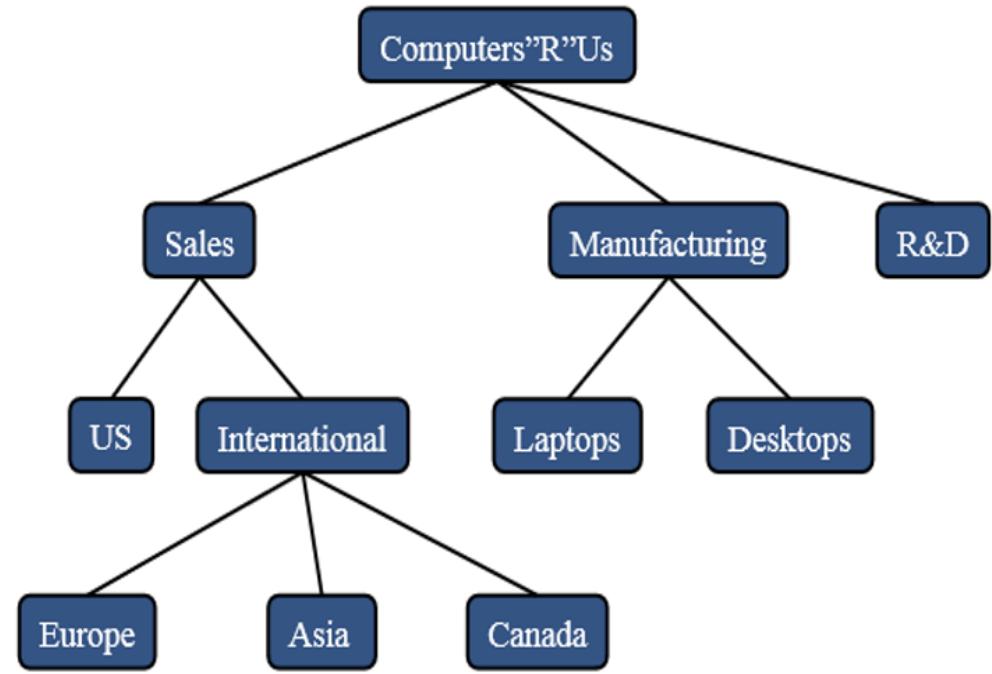


# Computer Scientist's View



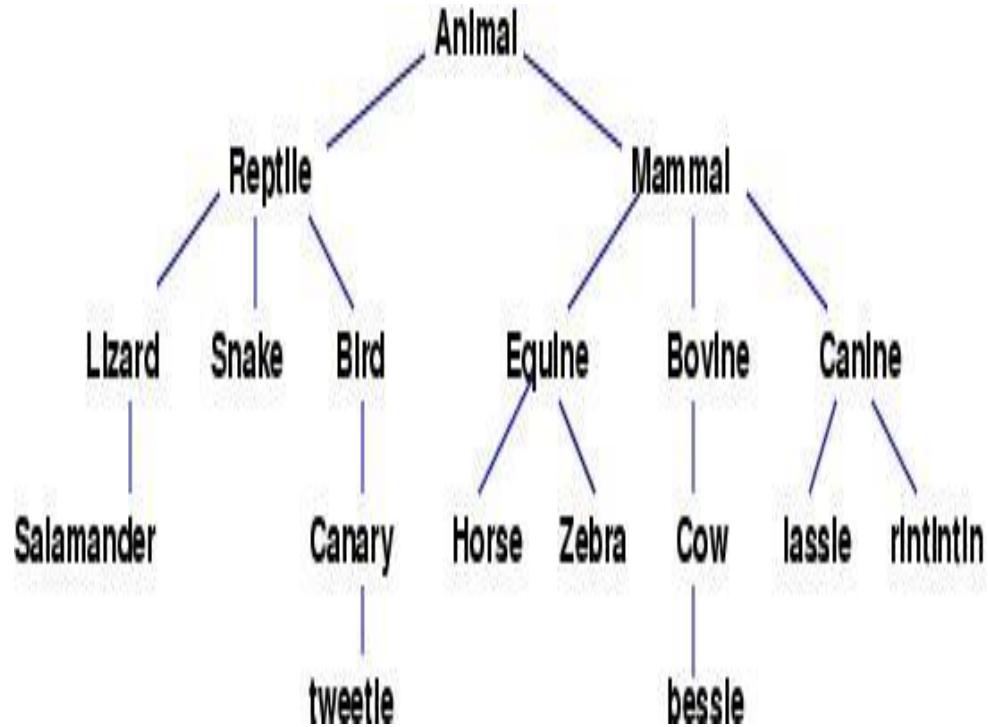
# Tree: A Hierarchical ADT

- A tree (upside down) is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relation.
- Each element (except the top element) has a parent and zero or more children elements.



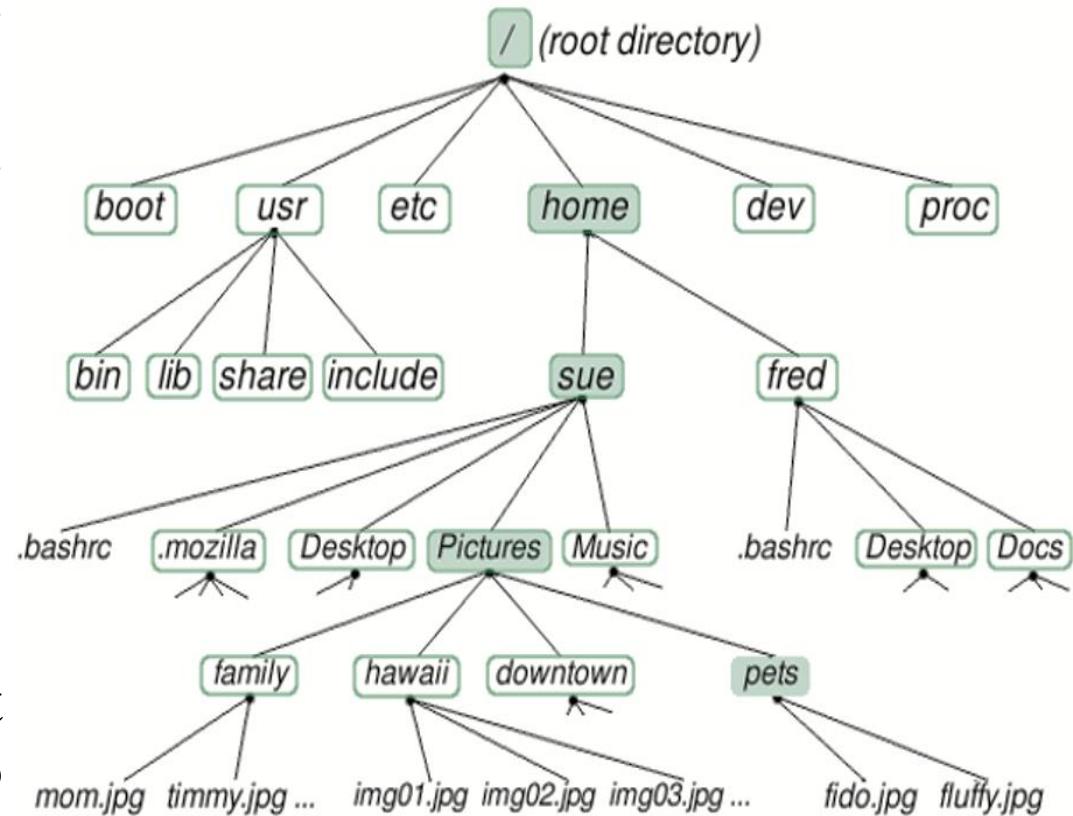
# What is a Tree

- A tree is a finite non-empty set of elements.
- It is an abstract model of a hierarchical structure.
- Consists of nodes with a parent-child relation.
- Applications:
  - Organizational data
  - File systems /storing naturally
  - Network routing algorithm.
  - Programming environments
  - Compiler Design/Text processing (syntax analysis & to display the structure of a sentence in a language)
  - Searching Algorithms and sorting
  - Evaluating a mathematical expression.



# Why do we need trees ?

- Lists, Stacks, and Queues are *linear* relationships
- Information often contains hierarchical relationships:
  - File directories or folders
  - Moves in a game
  - Hierarchies in organizations
- Can build a tree to support these relationships and also support fast searching.



# Tree Applications : Outline as Tree



## Features of Object-orientation

### I. Introduction

- A. Vocabulary
- B. Topics

### II. Encapsulation

- A. Abstraction
- B. Example

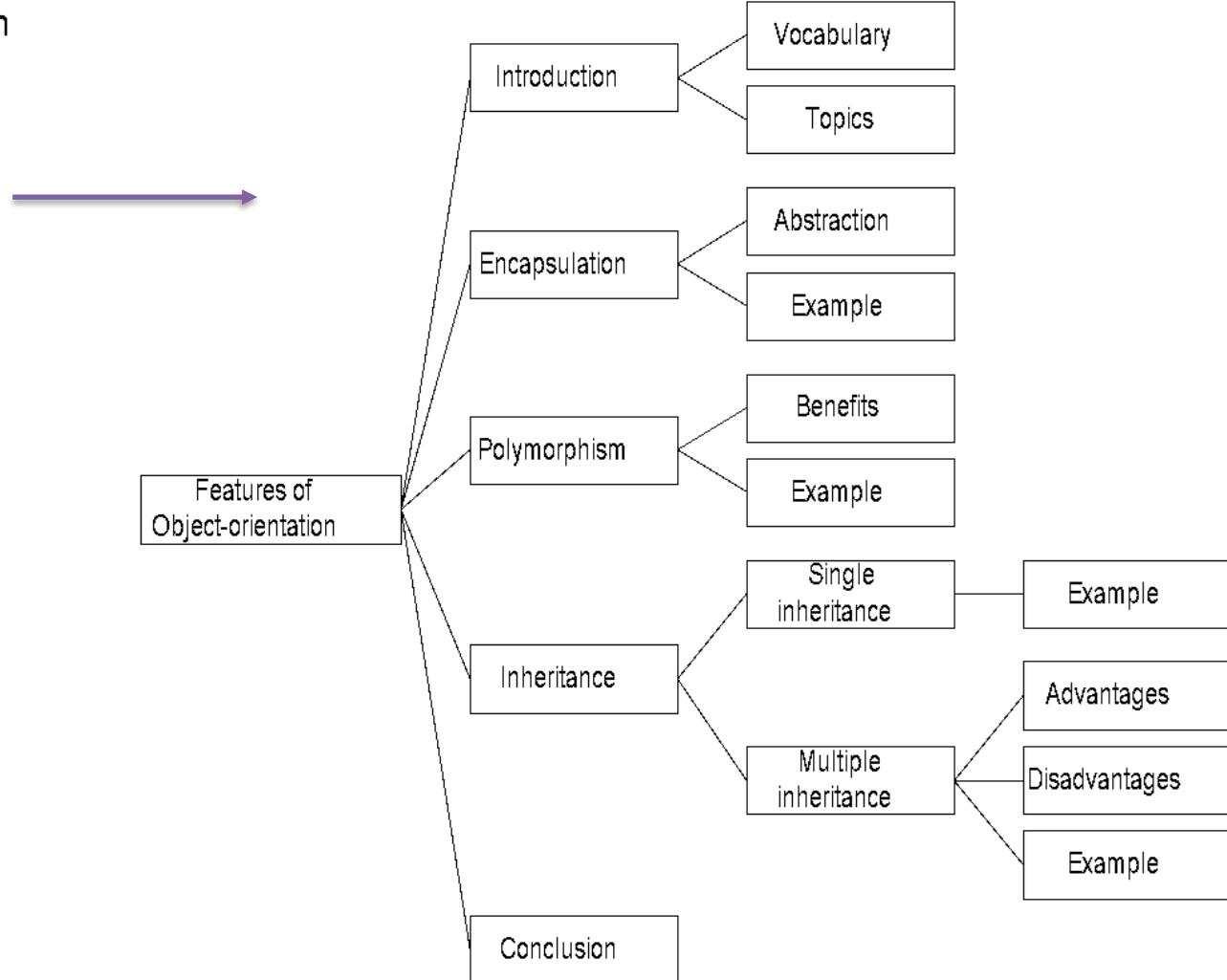
### III. Polymorphism

- A. Benefits
- B. Example

### IV. Inheritance

- A. Single inheritance
  - 1. Example
- B. Multiple inheritance
  - 1. Advantages
  - 2. Disadvantages
  - 3. Example

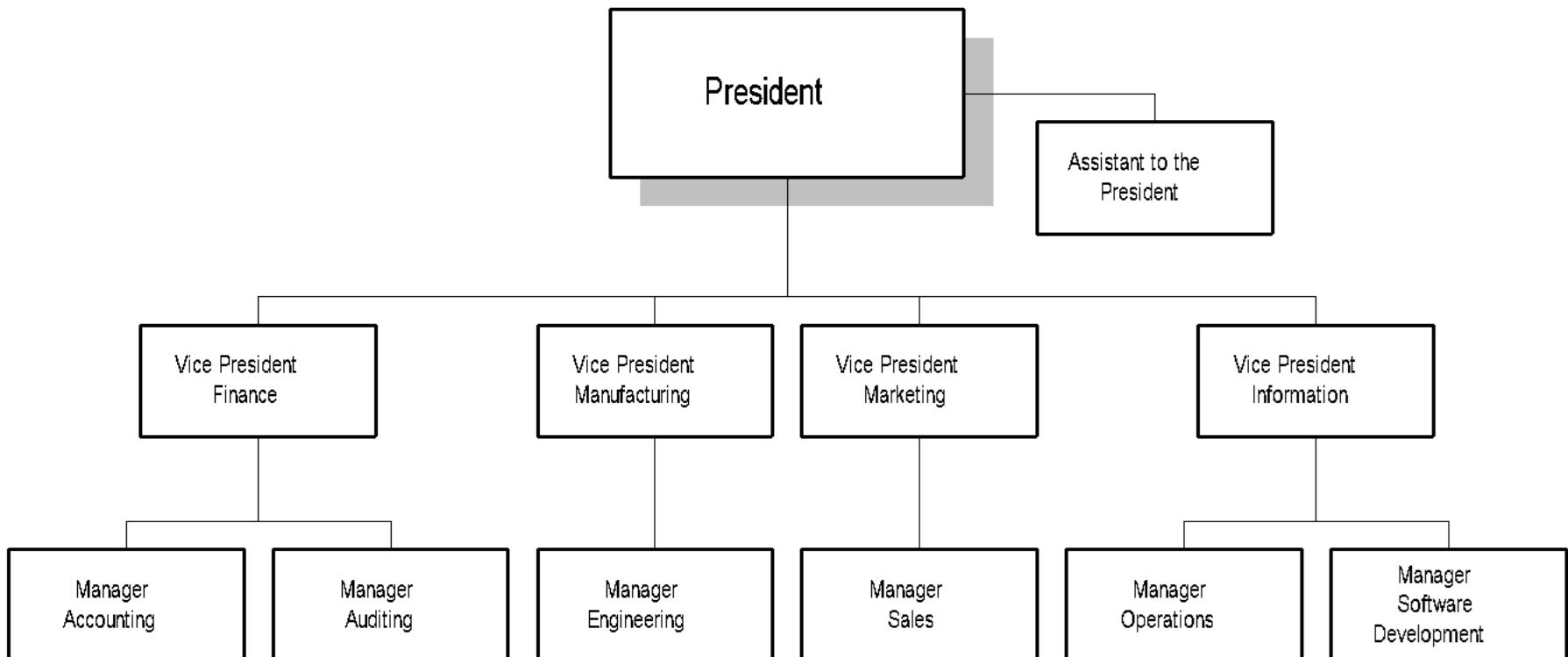
### V. Conclusion



# Tree Applications : Organizational Chart



## XYZ Corporation

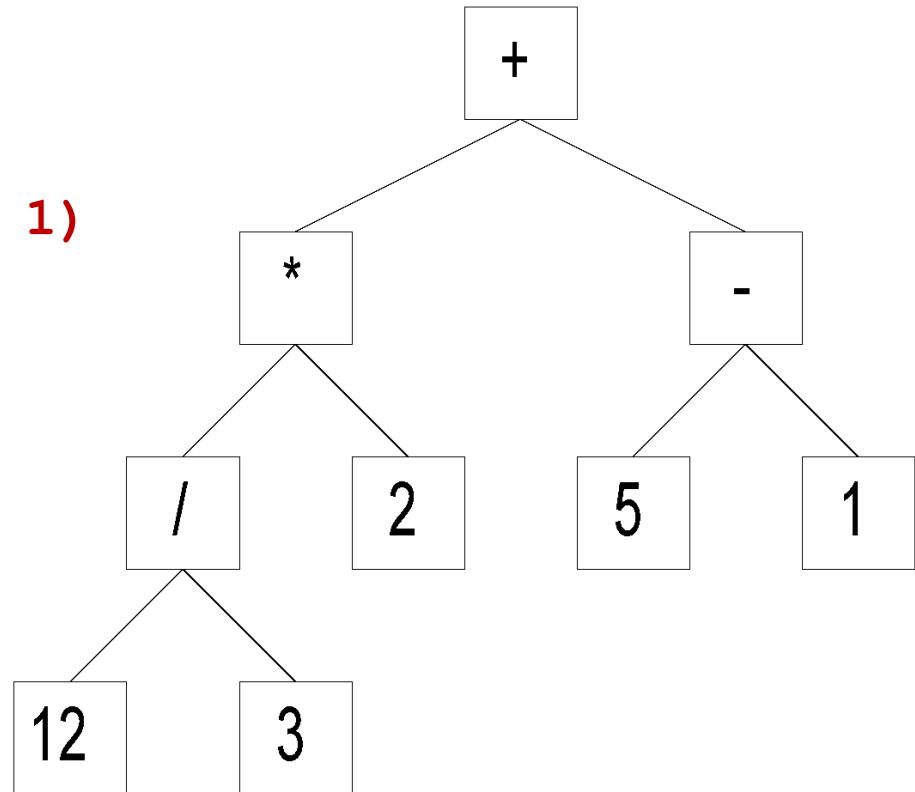


# Tree Applications : Expression Tree



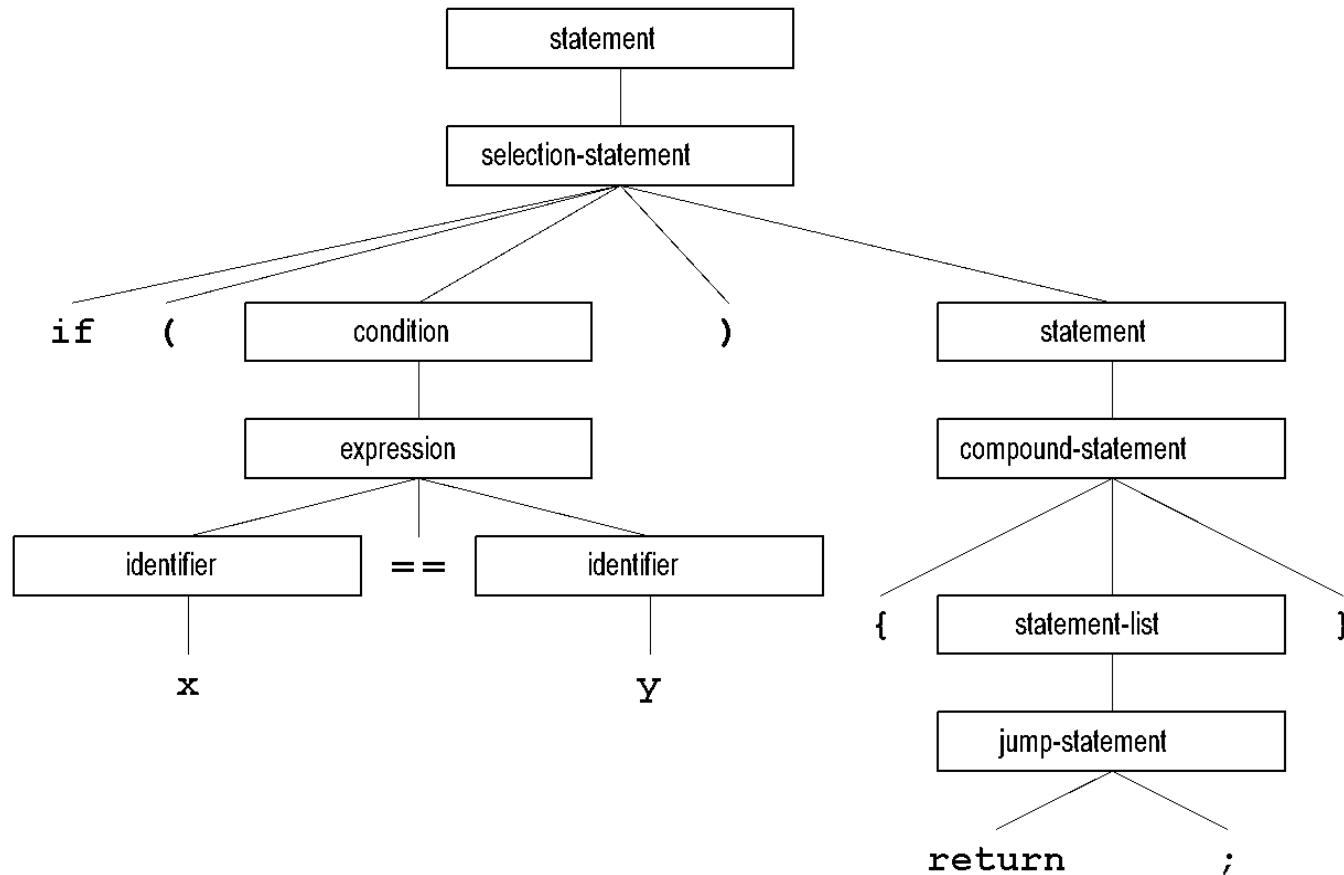
Can represent an arithmetic expression as a tree

((12 / 3) \* 2) + (5 - 1)



# Tree Applications : Programs as Trees

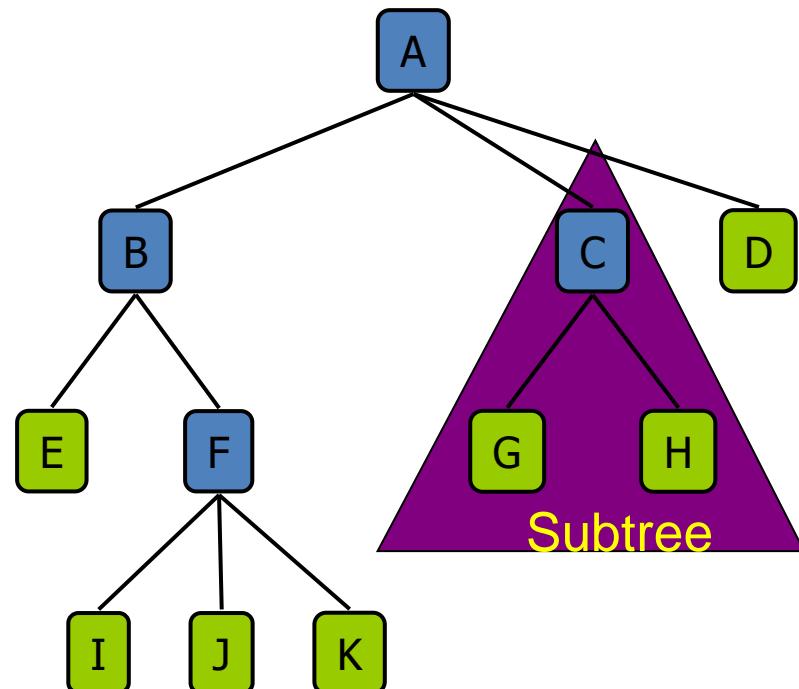
- Many compilers represent programs as trees



# Tree Terminology

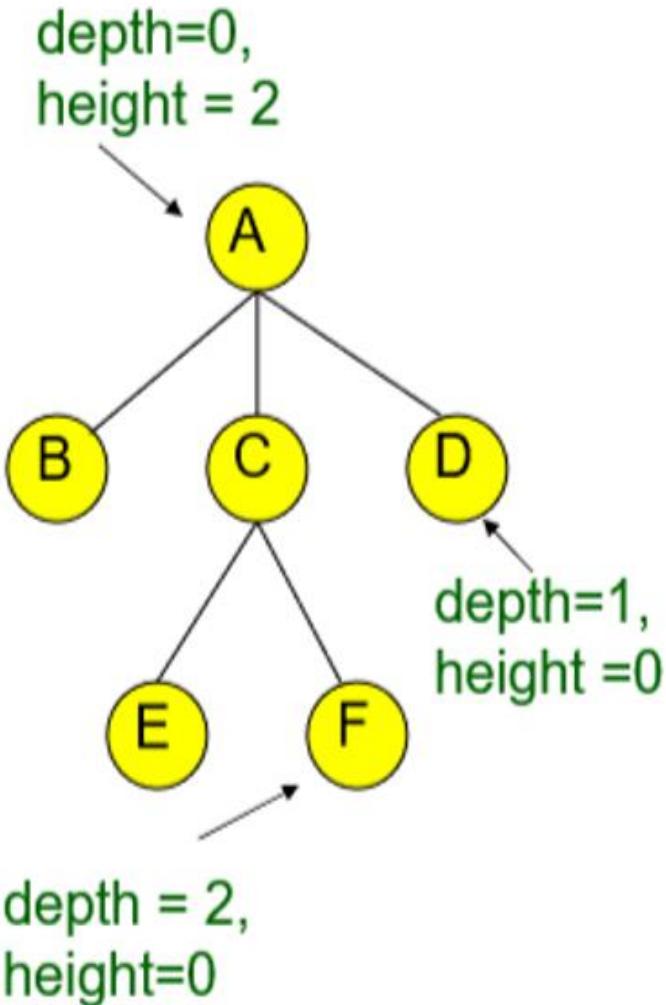
- ✓ **Root:** node without parent (A)
- ✓ **Siblings:** nodes share the same parent (B,C,D)
- ✓ **Internal node:** node with at least one child (A, B, C, F)
- ✓ **External node (leaf):** node without children (E, I, J, K, G, H, D)
- ✓ **Ancestors** of a node: parent, grandparent, great-grandparent, etc. (For E, B and A are ancestors).
- ✓ **Descendant** of a node: child, grandchild, great-grandchild, etc. (For B, E and F,I,J,K are descendants)
- ✓ **Degree** of a node: the number of its children ( For A its 3, F its 3, C its 2).
- ✓ **Depth of x** = no of edges in path from root to x. ( depth of F is 2 )
- ✓ **Height of x**= no of edges in longest path from x to a leaf. ( height of B is 2, height of A is 3).

- ✓ **Sub-tree:** tree consisting of a node and its descendants
- ✓ **Level:** set of all nodes at the same depth. Root at 0.
- ✓ **Forest:** set of disjoint trees.

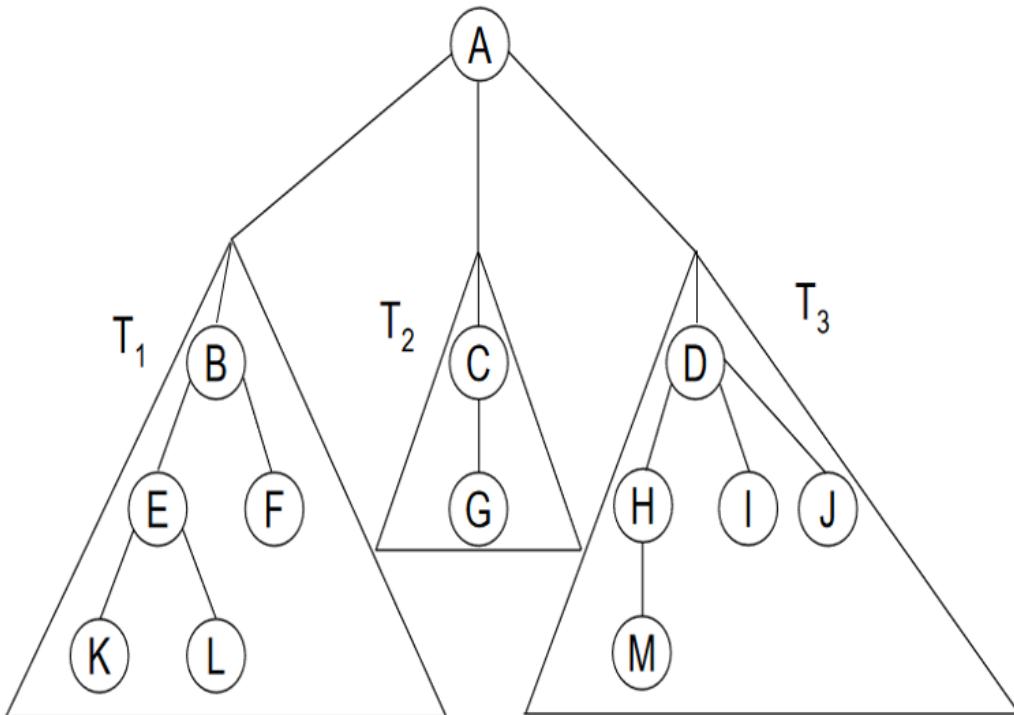


# More Tree Jargon

- *Length of a path* = number of edges
- *Depth of a node N* = length of path from root to N
- *Height of node N* = length of longest path from N to a leaf
- *Depth of a tree* = depth of deepest node
- *Height of a tree* = height of root



# Tree Terminology Activity



- ✓ Root ?
- ✓ Internal nodes ?
- ✓ Leaf's ?
- ✓ Ancestors of M ?
- ✓ Descendants of E ?
- ✓ Degree of A ?
- ✓ Height of tree ?
- ✓ I is in which level ?
- ✓ Depth of H ?
- ✓ Height of D ?

# Tree ADT

➤ We use positions to abstract nodes

➤ Generic methods:

***integer size():***Return the number of nodes in the tree.

***boolean isEmpty() :*** Return if tree is empty

***ObjectIterator elements():***Return an iterator of all the elements stored at nodes of the tree.

➤ Accessor methods:

***position root():***Return the root of the tree

***position parent(p):***Return the parent of node p; error if p is root.

***positionIterator children(p):***Return an iterator of the children of node p.

# Tree ADT

---

- **Query methods:**
  - boolean `isInternal(v)`: Test whether node v is internal.
  - boolean `isExternal(v)`: Test whether node v is external
  - boolean `isRoot(v)`: Test whether node v is the root.
- **Update methods:**
  - `swapElements(v, w)`: Swap the elements stored at the nodes v and w.
  - object `replaceElement(v, e)`: Replace with e and return the element stored at node v

# TREE ADT-Depth and Height

- Let  $v$  be a node of a tree  $T$ .
- The depth of  $v$  is the number of ancestors of  $v$ , excluding  $v$  itself
- If  $v$  is the root, then the depth of  $v$  is 0.
- Otherwise, the depth of  $v$  is one plus the depth of the parent of  $v$ .

*Algorithm depth( $T, v$ ):*

```

if  $T$ . isRoot( $v$ ) then
    return 0
else
    return 1 + depth ( $T, T$ . parent( $v$ ))
```

- The running time of algorithm  $\text{depth} (T, v)$  is  $O(1 + d_v)$ , where  $d_v$  denotes the depth of the node  $v$  in the tree  $T$ .
- In the worst case, the depth algorithm runs in  $O (n)$  time, where  $n$  is the total number of nodes in the tree  $T$

# TREE ADT- Depth and Height

- The *height of a tree* T is equal to the maximum depth of an external node of T.
- If v is an external node, then the height of v is 0.
- Otherwise, the height of v is one plus the maximum height of a child of v.

***Algorithm height(T, v) :***

```

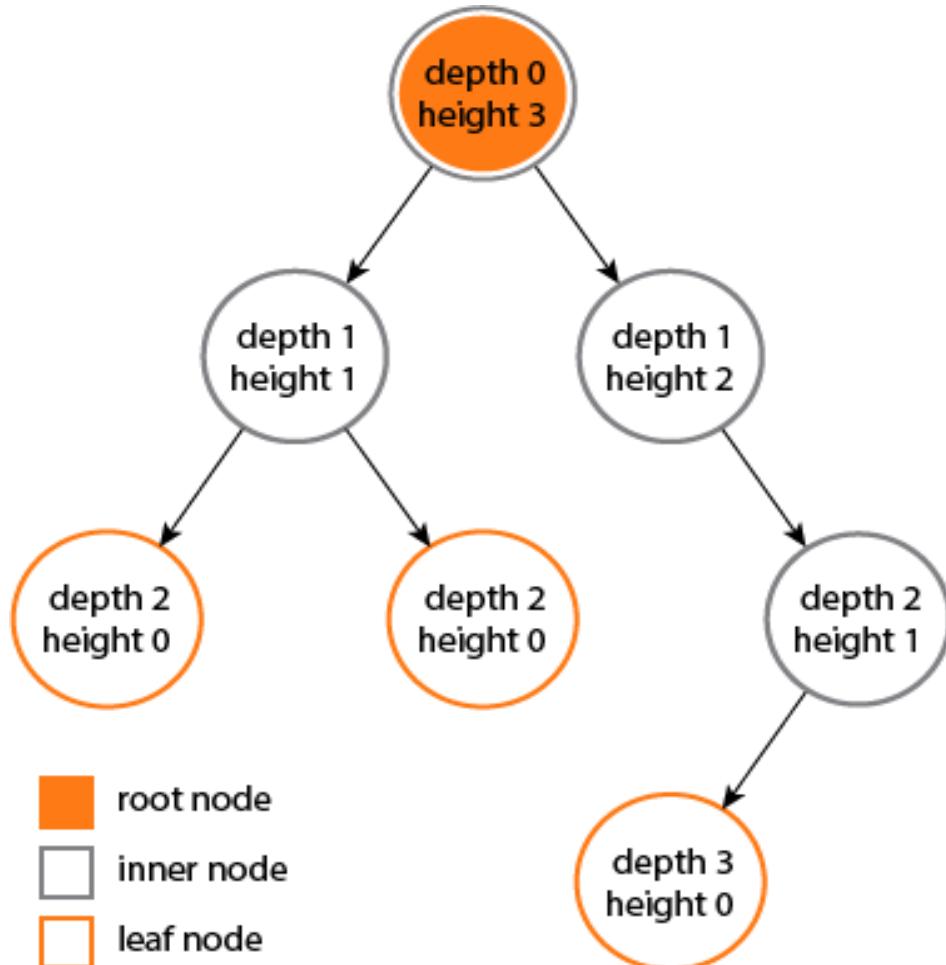
if T. isExternal (v) then
    return 0
else
    h = 0
    for each w ∈ T.children (v) do
        h = max(h, height(T, w))
    return 1 + h

```

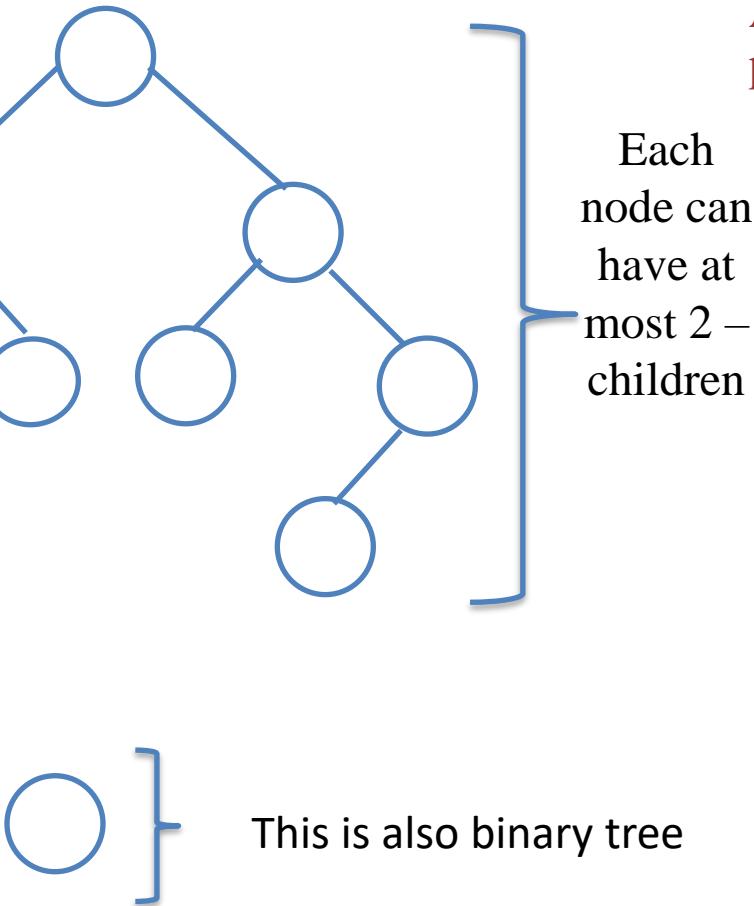
*The height of a tree T is obtained by calling height(T,T.root())*

Complexity of this algorithm is also O(n), Refer Chapter 2, T1. (This involves Amortized analysis).

# TREE ADT-Dept h and Height

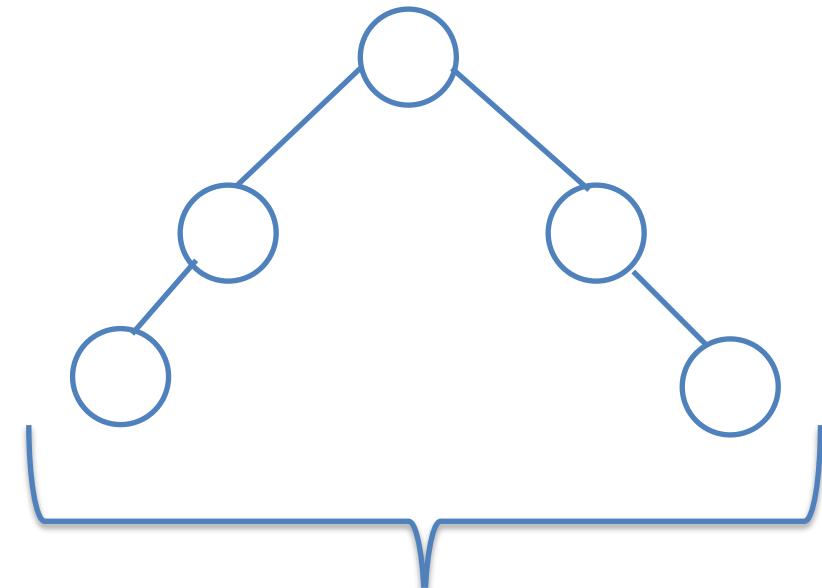


# Binary Tree



A binary tree is a tree with the following properties:

- Each internal node has **at most two** children.
- The children of a node are called as left child and right child.

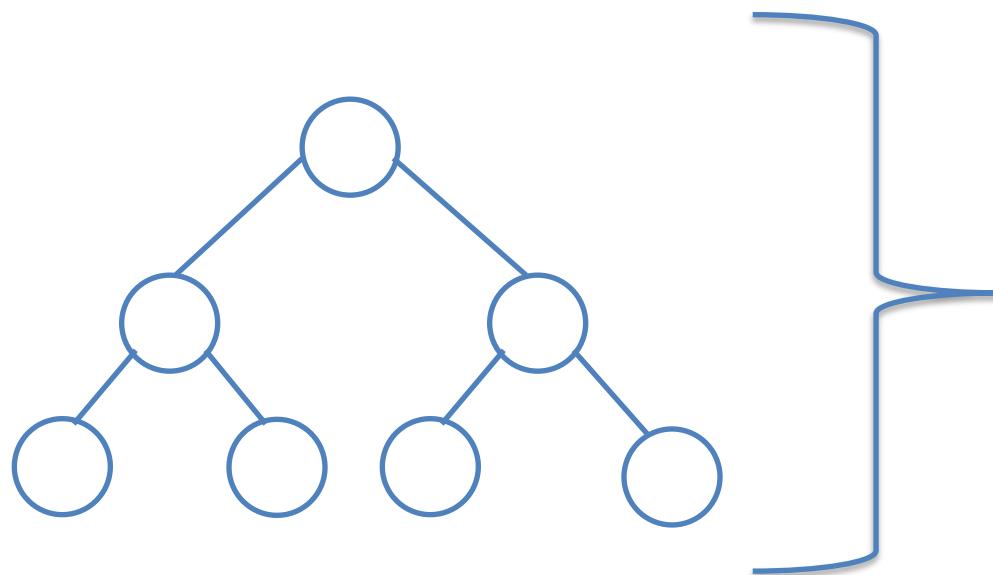


What about this ?

# Strict / Proper/ Full Binary tree

## Proper Binary Tree

- Each internal node has **exactly** two children

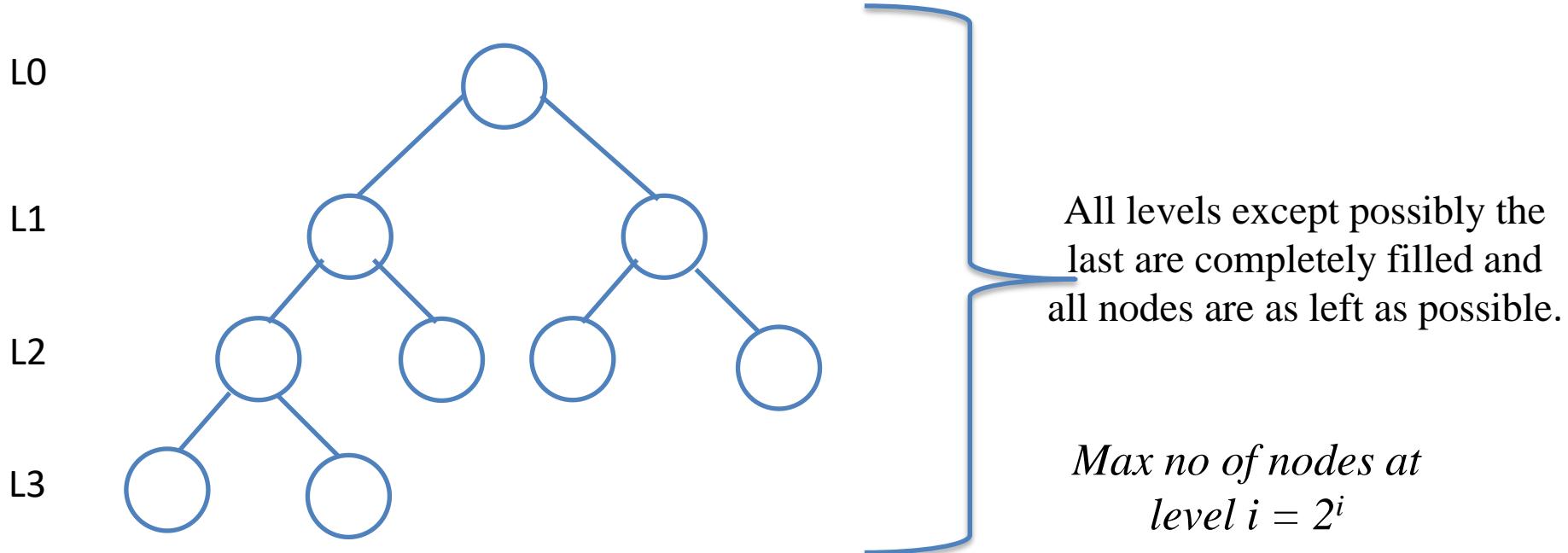


Each node can have either 2 or 0 children.

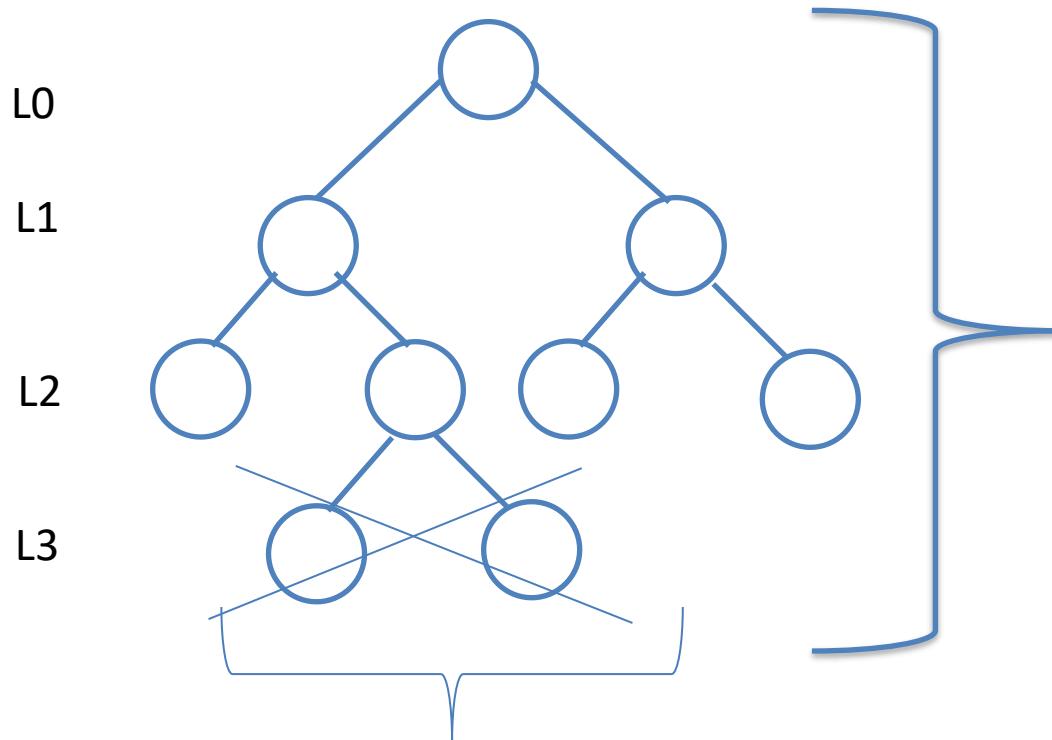
# Complete Binary tree

## Complete binary tree

- A binary tree in which every level, except the lowermost, is completely (max) filled
- At the lowermost level nodes must be filled from left to right



# Complete Binary tree

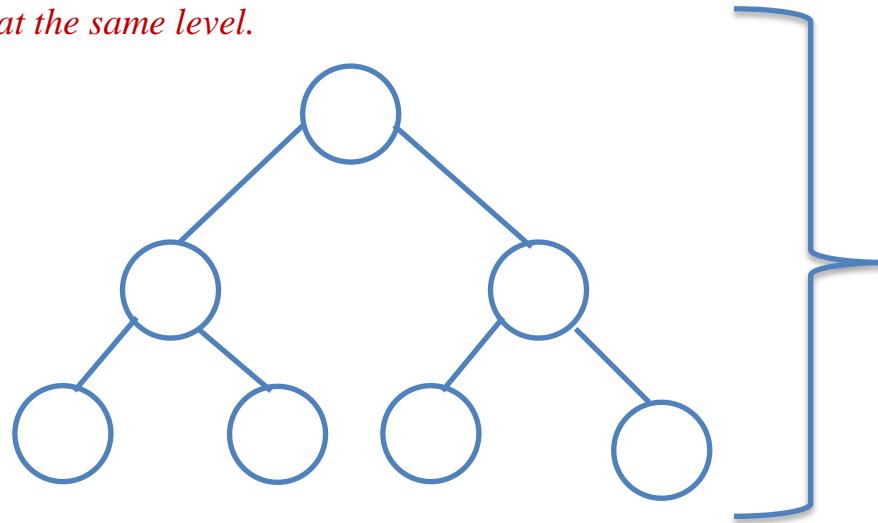


All levels except possibly the last are completely filled and all nodes are not as left as possible. So this is not a complete Binary tree

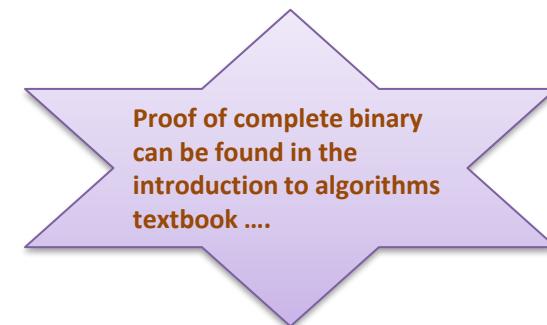
This is not the left  
as possible

# Perfect binary tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



In perfect binary all levels will be completely filled.



Maximum no of nodes in a binary tree with height h

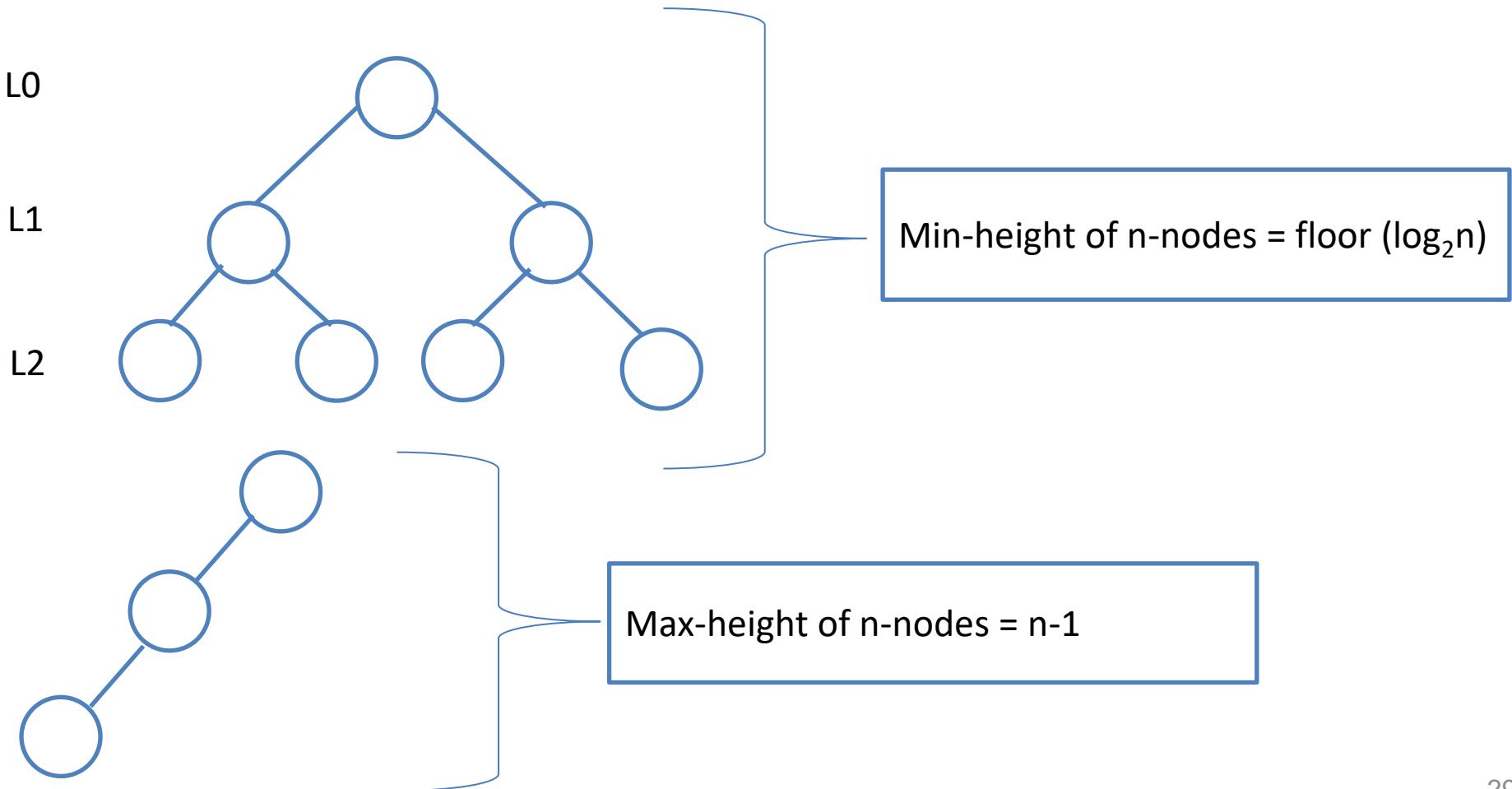
$$\begin{aligned}
 &= 2^0 + 2^1 + \dots + 2^h \\
 &= 2^{h+1} - 1 \\
 &= 2^{(\text{no of levels})} - 1 \\
 n &= 2^{h+1} - 1
 \end{aligned}$$

- ❖ Now you can find the height of a tree, with n
 
$$n = 2^{h+1} - 1$$

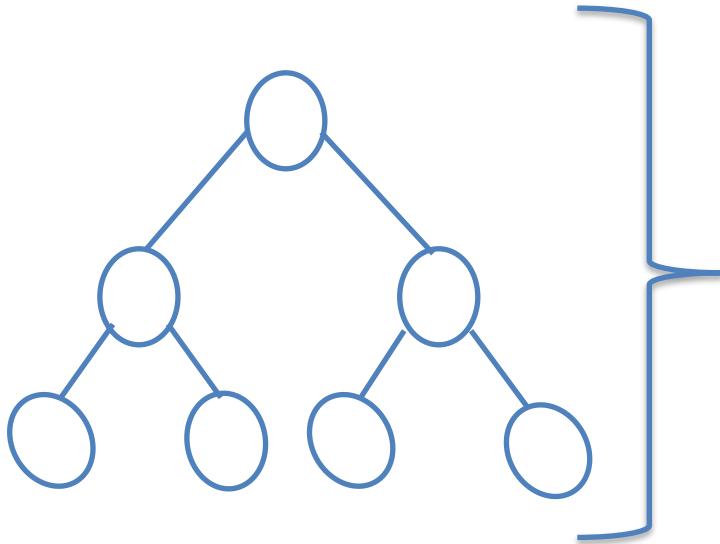
$$2^{h+1} = n+1$$

$$h = \log_2(n+1) - 1$$
  - ❖ Height of complete binary tree  $h = \text{floor}(\log_2 n)$

# Min and max height of a binary tree



# Balanced binary tree



Difference between height of left & right subtree for every node is not more than k (mostly 1).

- ❖ Height → no of edges in longest path from root to a leaf.
- ❖ Height of an empty tree = -1
- ❖ Height of tree with one node = 0

# Binary tree implementation

---

We can implement binary tree using

- Linked List [Dynamically created nodes]
- Arrays [this will be efficient only for complete binary tree]

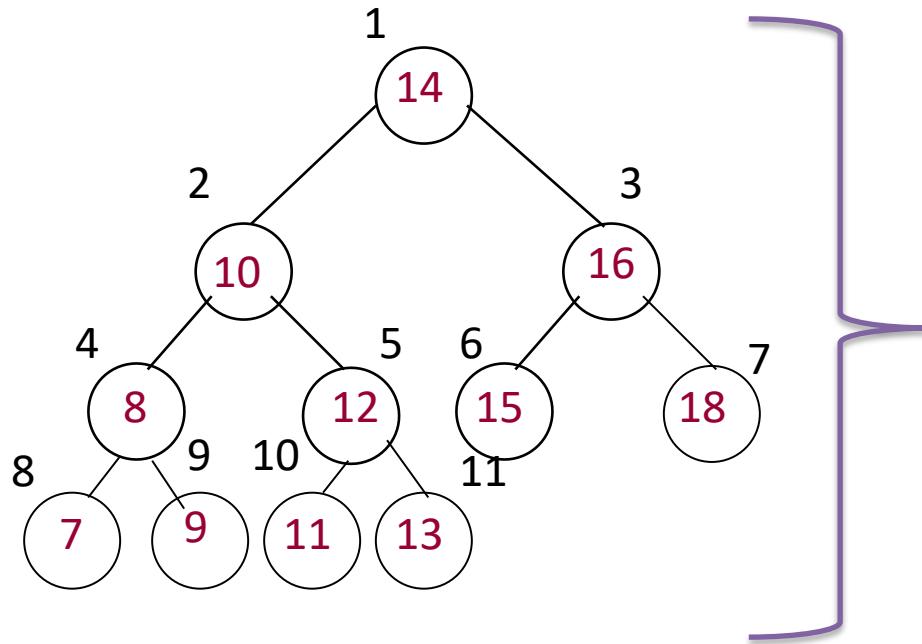
# Complete Binary Trees: Array Representation



Complete Binary Trees can be represented in memory with the use of an array A so that all nodes can be accessed in  $O(1)$  time:

- Root will be the first position in A
- Label nodes sequentially top-to-bottom and left-to-right
- Left child of A[i] is at position A[2i]
- Right child of A[i] is at position A[2i + 1]
- Parent of A[i] is at A[i/2]

# Complete Binary Trees: Array Representation

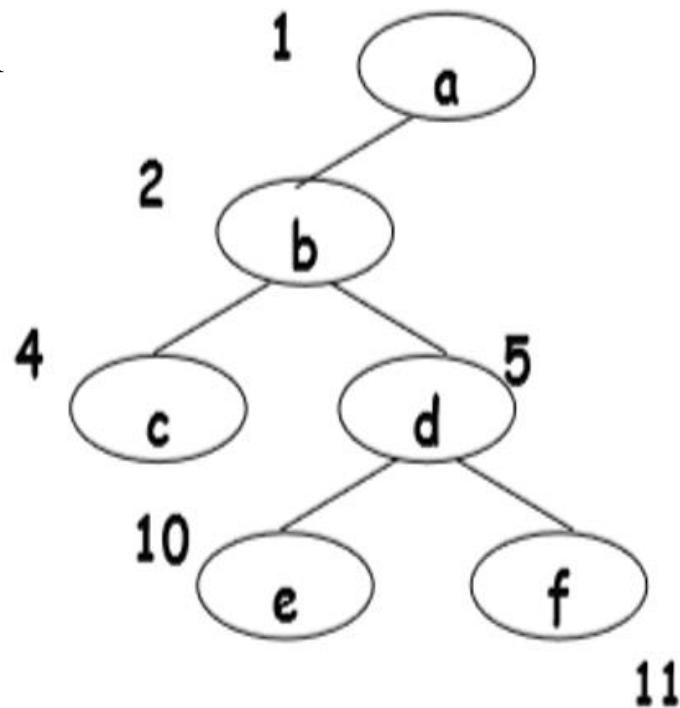


- Left child of  $A[i]$  is at position  $A[2i]$
- Right child of  $A[i]$  is at position  $A[2i + 1]$
- Parent of  $i$  is at position  $\text{floor}(i/2)$

|    |    |    |   |    |    |    |   |   |    |    |
|----|----|----|---|----|----|----|---|---|----|----|
| 14 | 10 | 16 | 8 | 12 | 15 | 18 | 7 | 9 | 11 | 13 |
| 1  | 2  | 3  | 4 | 5  | 6  | 7  | 8 | 9 | 10 | 11 |

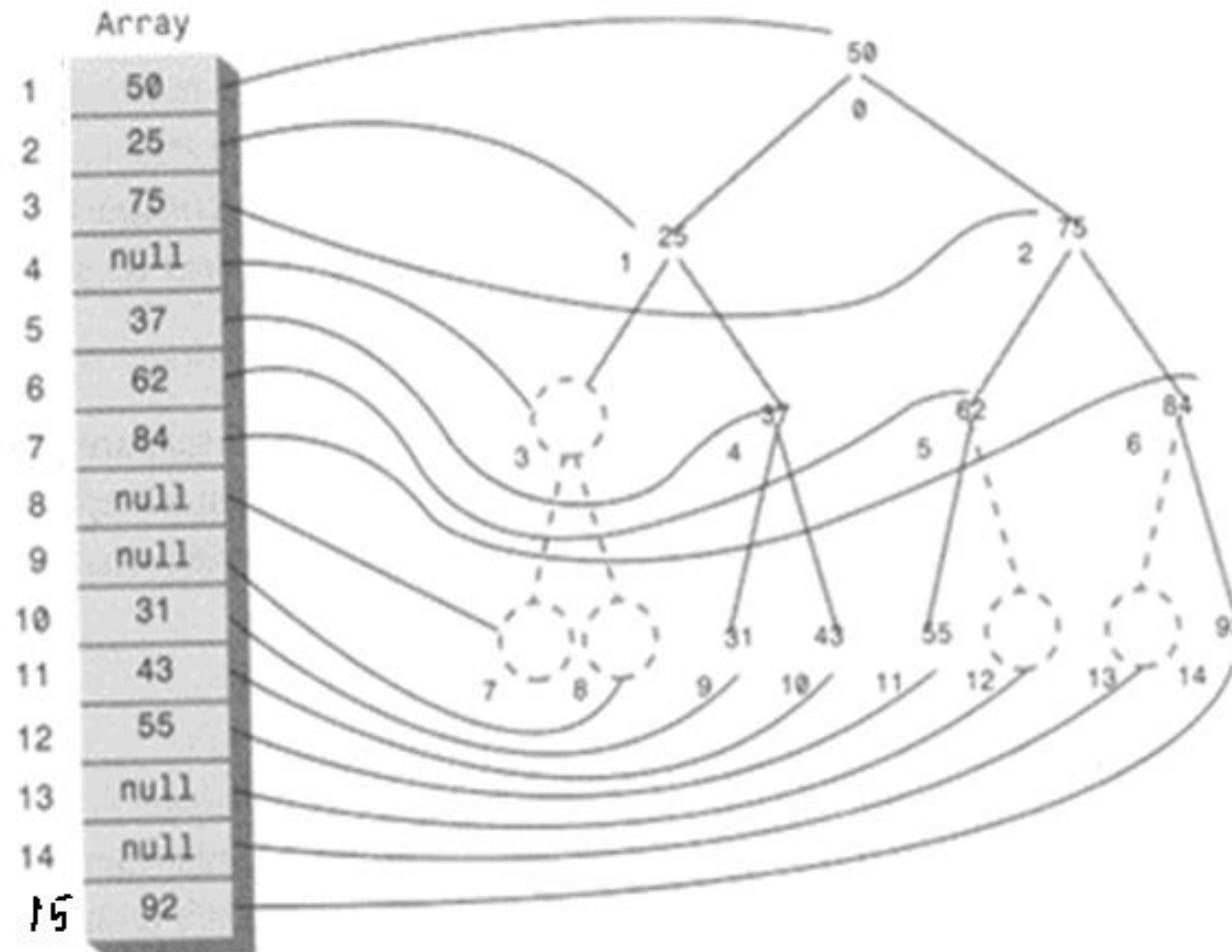
# Binary Trees: Array Representation

Sequential representation of a tree with **depth d** will require an array with approx.  $(2^{d+1} - 1)$  elements



|   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| a | b |   | c | d |   |   |   |   | e  | f  |

# Binary Trees: Array Representation



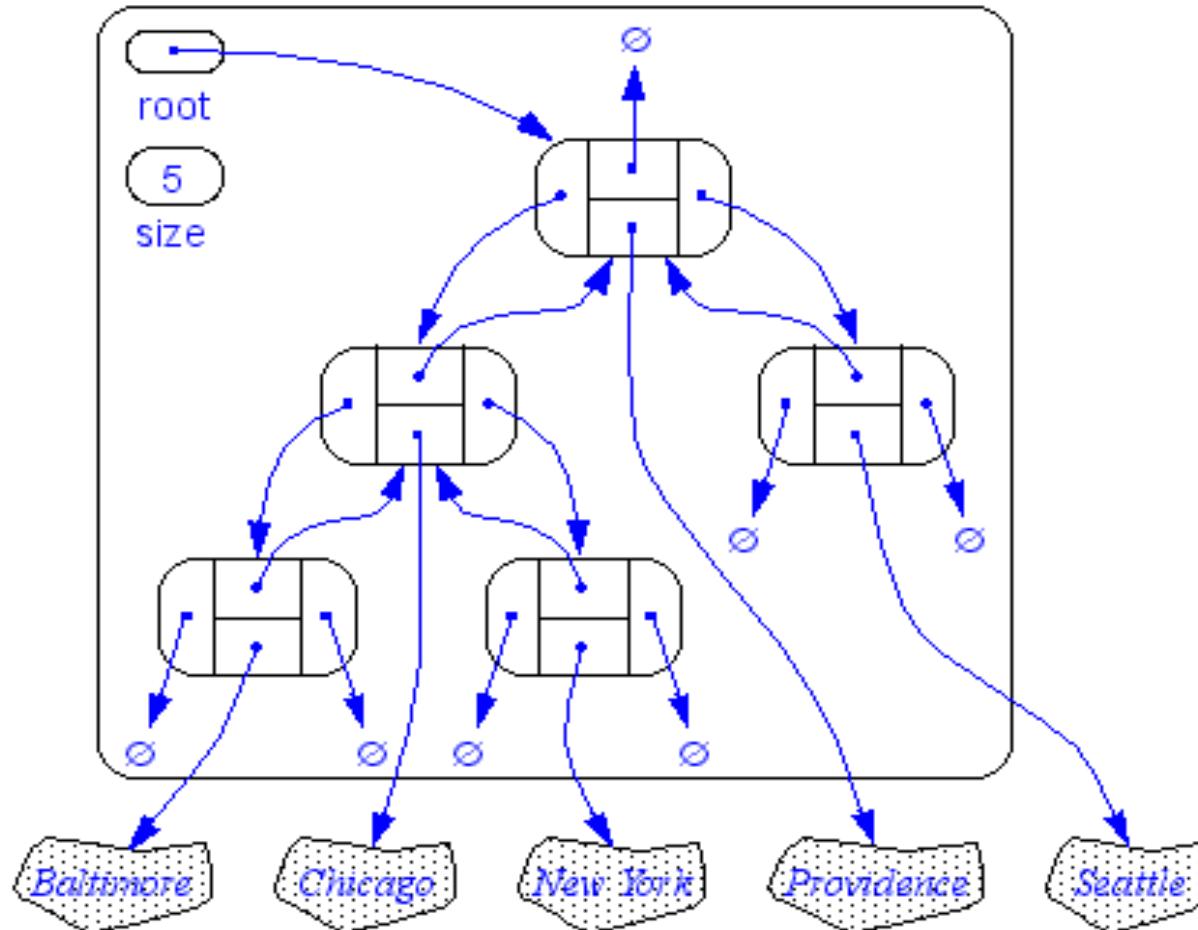
# Binary Trees: Linked List Representation

---

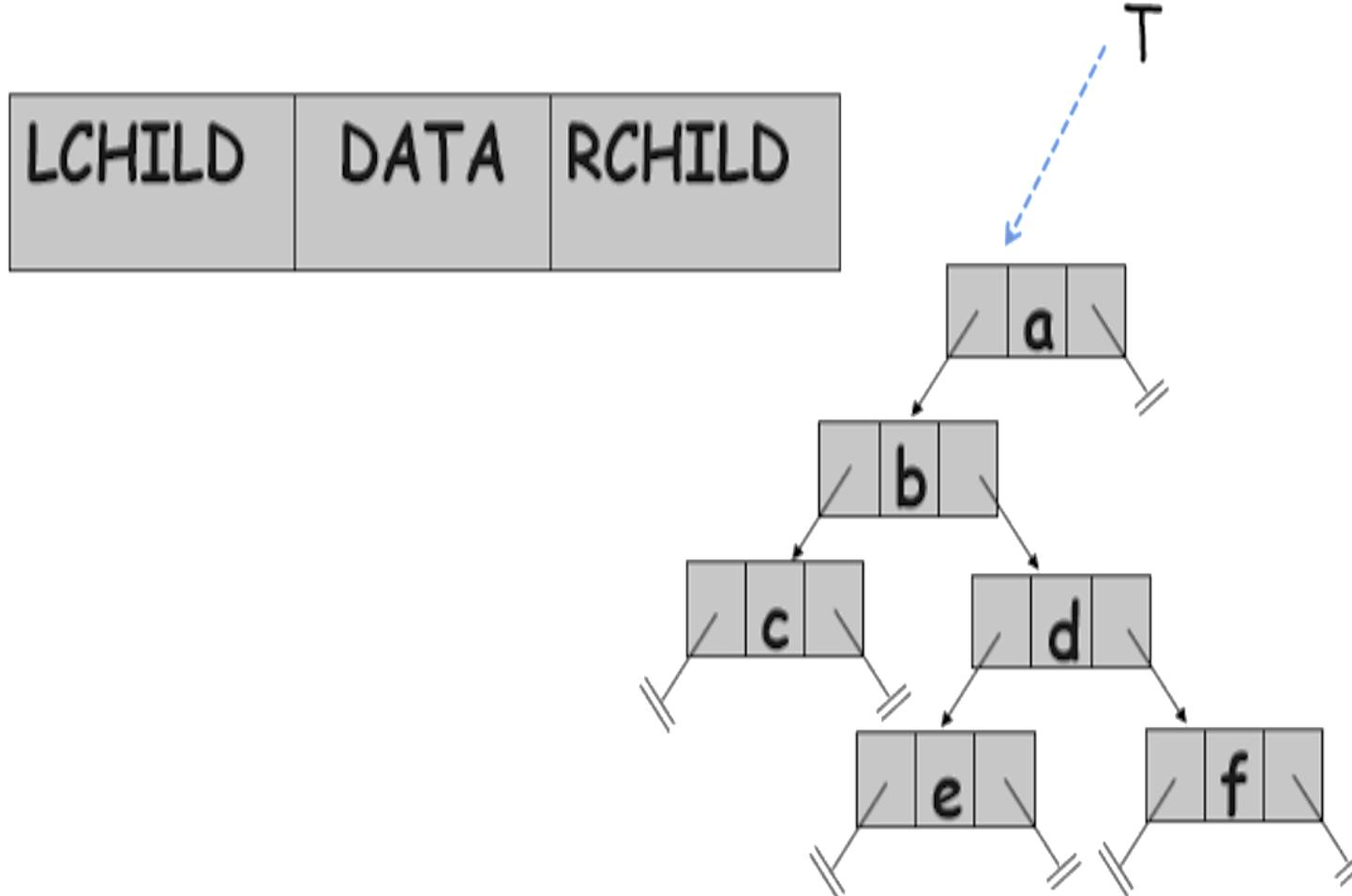
A **Binary tree** is a linked data structure. Each node contains data ,and pointers left, right and p.

- *Left points to the left child of the node.*
- *Right points to the right child of the node.*
- *p points to the parent of the node.*
- If a child is missing, the pointer is NIL.
- If a parent is missing, p is NIL.
- The **root** of the tree is the only node for which p is NIL.
- Nodes for which both left and right are NIL are **leaves**.

# Binary Trees: Linked List Representation

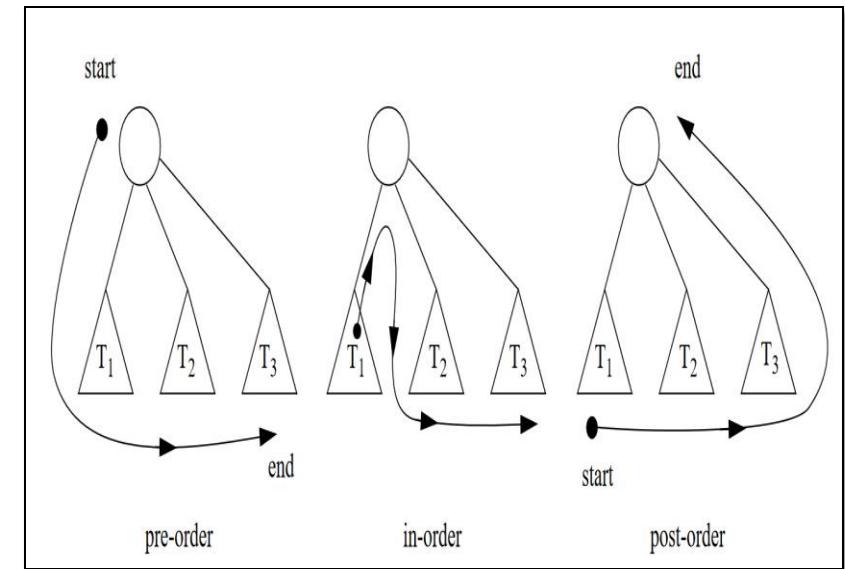


# Binary Trees: Linked Representation



# Binary Tree Traversals

- To **traverse** (or **walk**) the binary tree is to visit each node in the binary tree exactly once.
- Tree traversals are naturally recursive.
- Since a binary tree has three parts, the ways to traverse the binary tree are:
  - ✓ root, left, right : Preorder
  - ✓ left, root, right: Inorder
  - ✓ left, right, root: Postorder



# Binary Tree Traversals

Three ways of traversing the binary tree T with root R:

## Preorder

- Process the root R
- Traverse the left sub-tree of R in preorder
- Traverse the right sub-tree of R in preorder

a.k.a node-left-right  
**(NLR)**

## In-order

- Traverse the left sub-tree of R in in-order
- Process the root R
- Traverse the right sub-tree of R in in-order

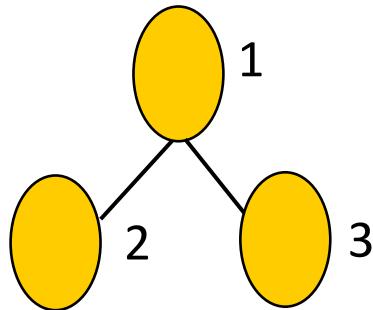
a.k.a left-node-right  
**(LNR)**

## Post-order

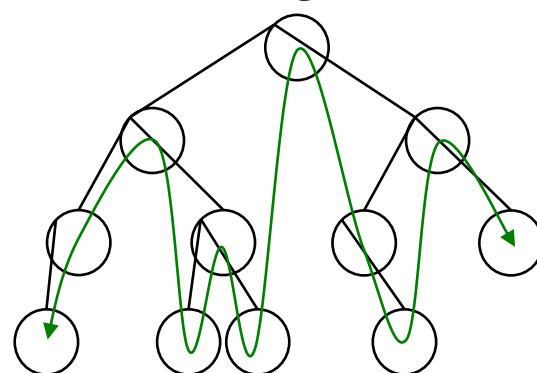
- Traverse the left sub-tree of R in post-order
- Traverse the right sub-tree of R in post-order
- Process the root R

a.k.a left-right-node  
**(LRN)**

# Binary Tree Traversals

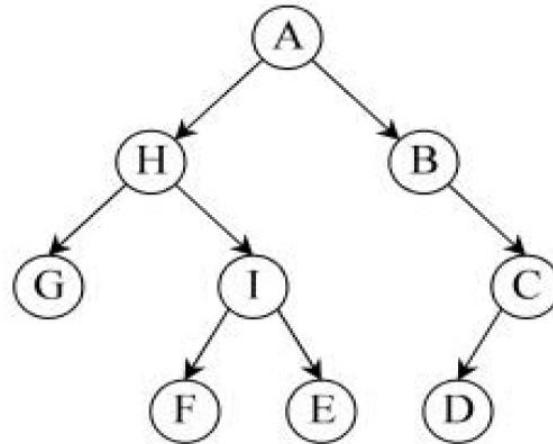


- Preorder(Root(N),left (L),right (R)): 1 2 3
- Inorder(left(L),root(N),right(R)): 2 1 3
- Postorder(left(L),right(R),root(N)): 2 3 1

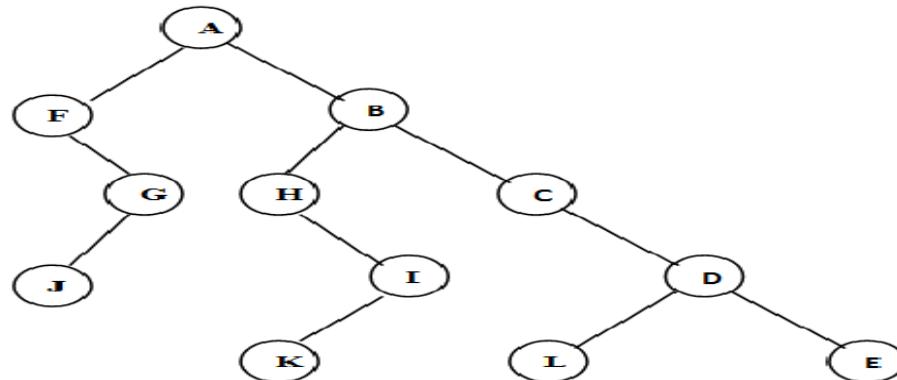


## Exercises for CS #4

1) Perform the 3 traversals on the below:



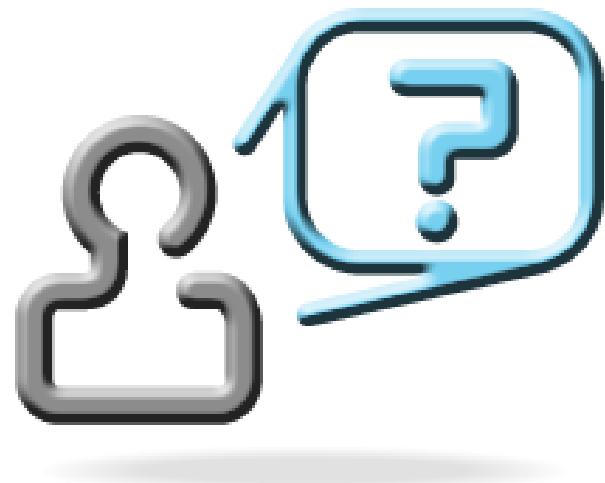
2) Represent below in array and Linked list:



# Summary for CS #4

---

- 1) Recap of CS#3
- 2) Introduction to Trees
  - Tree terminologies
  - Binary tree and types
  - Representation of tree
    - Using array
    - Using linked list
- 3) Q&A



*See you in the next class to explore heaps!*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# **Data Structures and Algorithms Design**

**DSECLZG519**

**Parthasarathy**





# Contact Session #5

# DSECLZG519 – Heaps & Heap Sort

# Agenda for CS #5

---

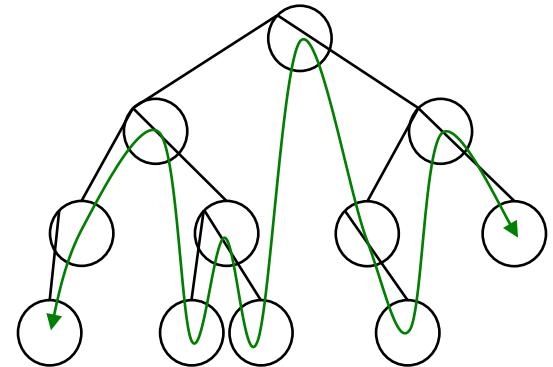
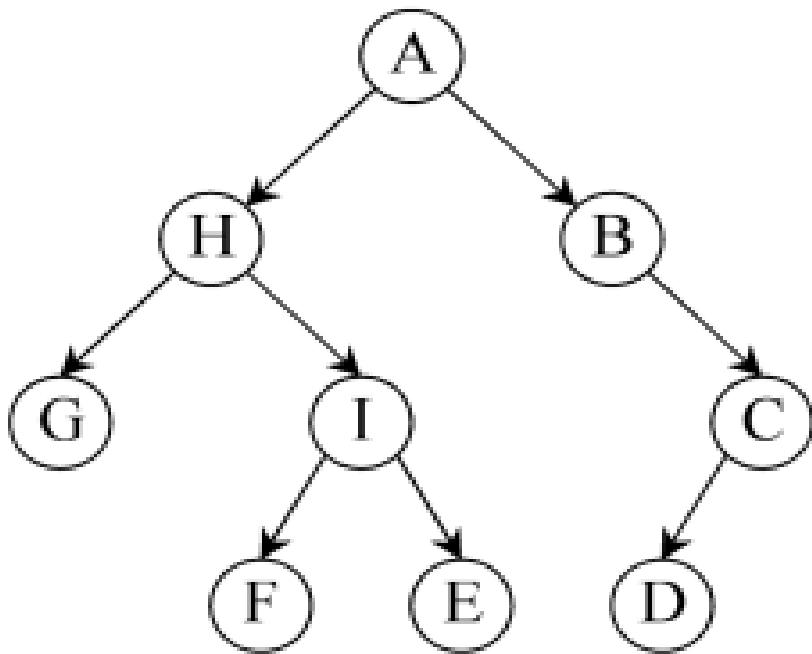
- 1) Recap of CS#4
  - 2) Introduction to Heap
    - What is Heap ?
    - Types of Heap
    - Heapification
    - Build a Heap
    - Insertion into a Heap
    - Removal from a Heap
    - Exercises
  - 3) Heap Sort
  - 4) Q&A
-

# Tree Traversal: In-order

# In-order

- Traverse the left sub-tree of R in in-order
  - Process the root R
  - Traverse the right sub-tree of R in in-order

a.k.a left-node-right (**LNR**)

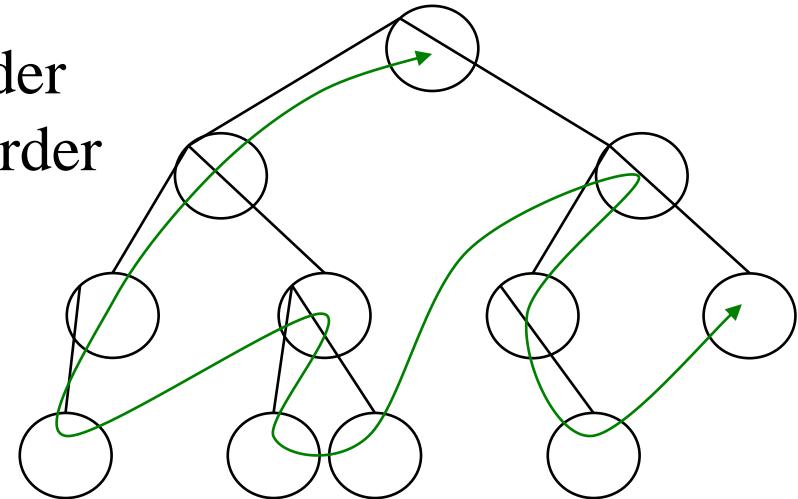
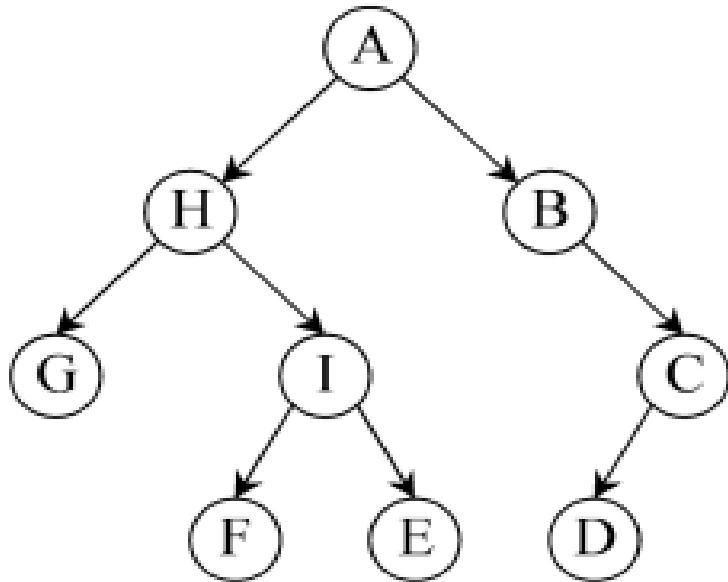


In-order (LNR) traversal yields:  
**G, H, F, I, E, A, B, D, C**

# Tree Traversal: Pre-order

## Preorder

- Process the root R
  - Traverse the left sub-tree of R in preorder
  - Traverse the right sub-tree of R in preorder
- a.k.a node-left-right (**NLR**)

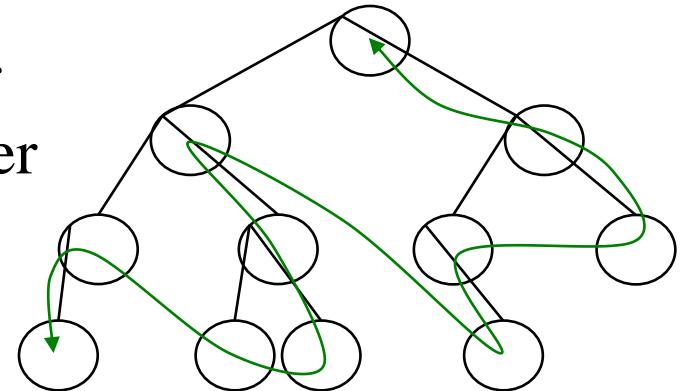
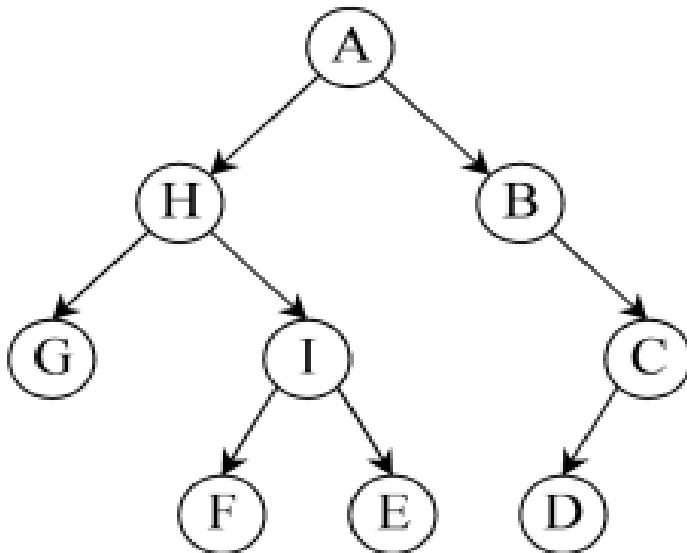


Preorder (NLR) traversal  
yields: **A, H, G, I, F, E, B, C, D**

# Tree Traversal: Post-Order

## Post-order

- Traverse the left sub-tree of R in post-order
  - Traverse the right sub-tree of R in post-order
  - Process the root R
- a.k.a left-right-node (**LRN**)



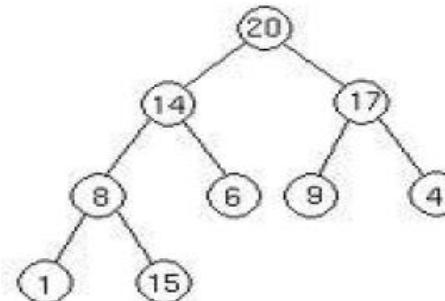
Postorder (LRN) traversal  
yields: **G, F, E, I, H, D, C, B,A**

# Application of Binary Trees

- Data Base indexing
- In video games
- Path finding algorithms in AI applications
- Huffman Coding
- Heaps
- Syntax tree.
- ....
- ...

# Heap

- Heap is a special tree-based data structure, that satisfies the following special heap properties:
  - *Shape Property*
  - *Heap Property*

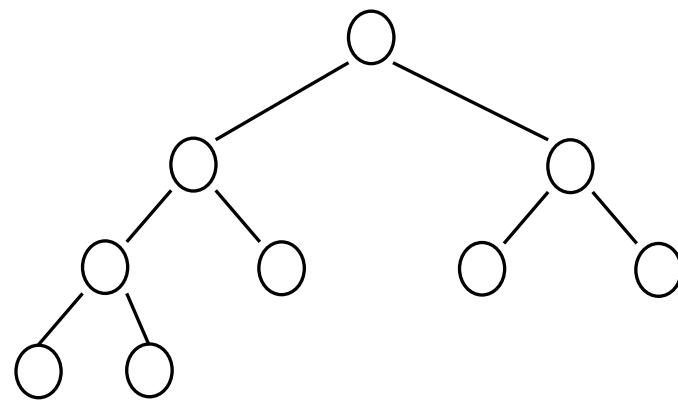


Heap



# Shape Property

Heap data structure is always a **complete binary tree**, which means all levels of the tree are fully filled till  $h-1$  and at level  $h$  (last level), the nodes are filled from left to right.



Complete Binary tree

# Heap Property

- All nodes are either [greater than or equal to] or [less than or equal to] each of its children.
- If the parent nodes are greater than their children, then such a heap is called: **Max-Heap**. *So, The root of any sub-tree holds the **greatest** value in the sub-tree*
- If the parent nodes are smaller than their children, then such a heap is called: **Min-Heap**. *So, the root of any sub-tree holds the **least** value in that sub-tree.*

- **Max-Heap**

for every node  $v$  other than the root

$$\text{element}(\text{parent}(v)) \geq \text{element}(v)$$

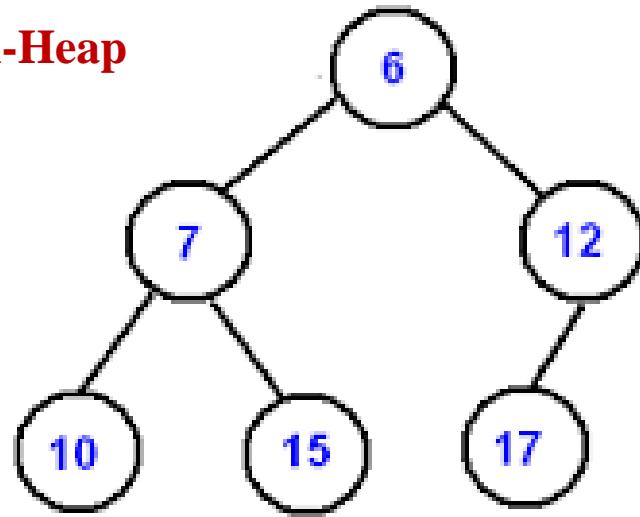
- **Min-Heap**

for every node  $v$  other than the root

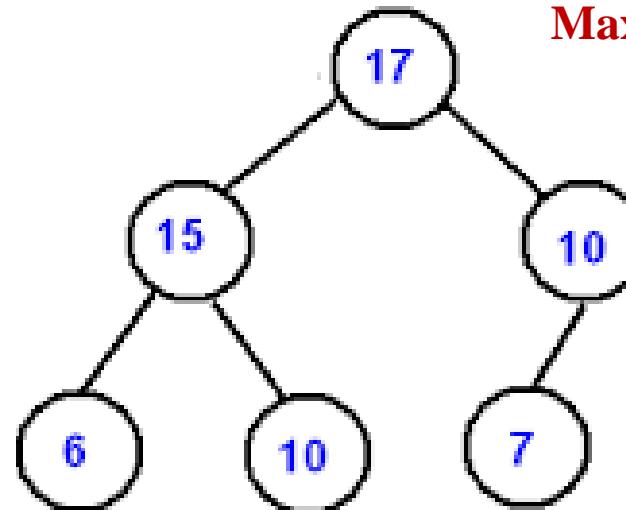
$$\text{element}(\text{parent}(v)) \leq \text{element}(v)$$

# Examples of min-heap and max-heap

Min-Heap

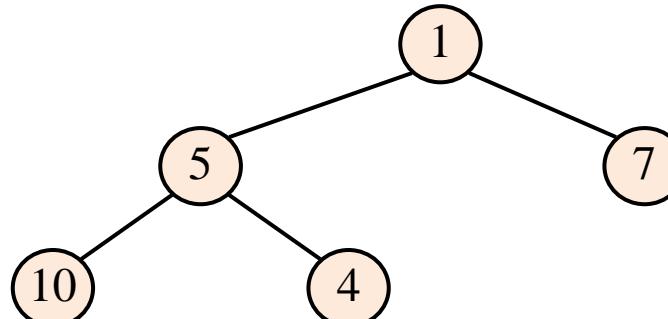
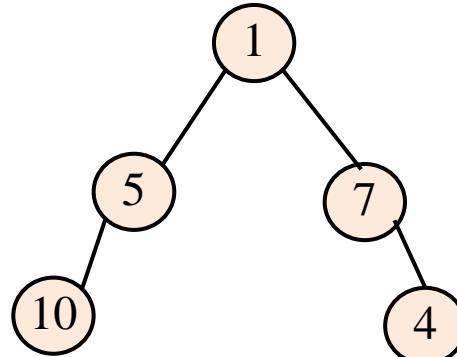


Max-Heap



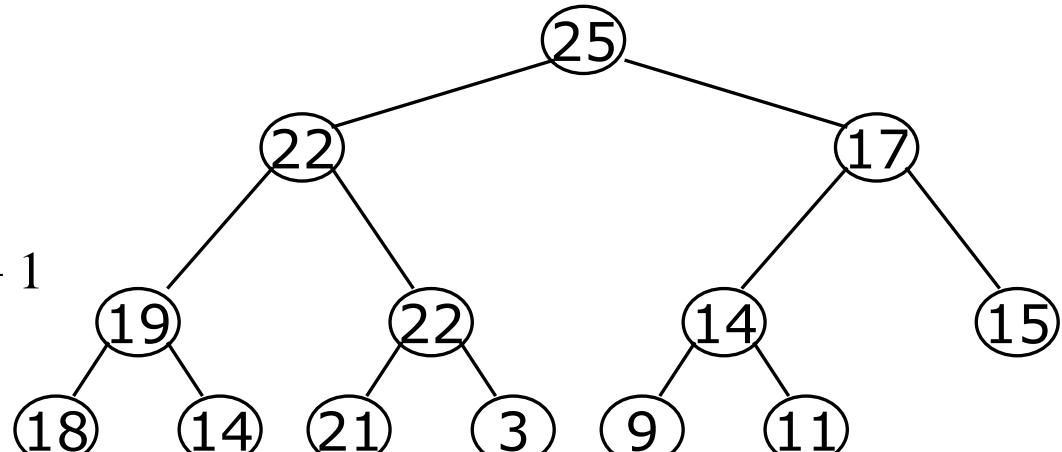
Check :

*These  
are not  
heaps!!*



# Heap Representation

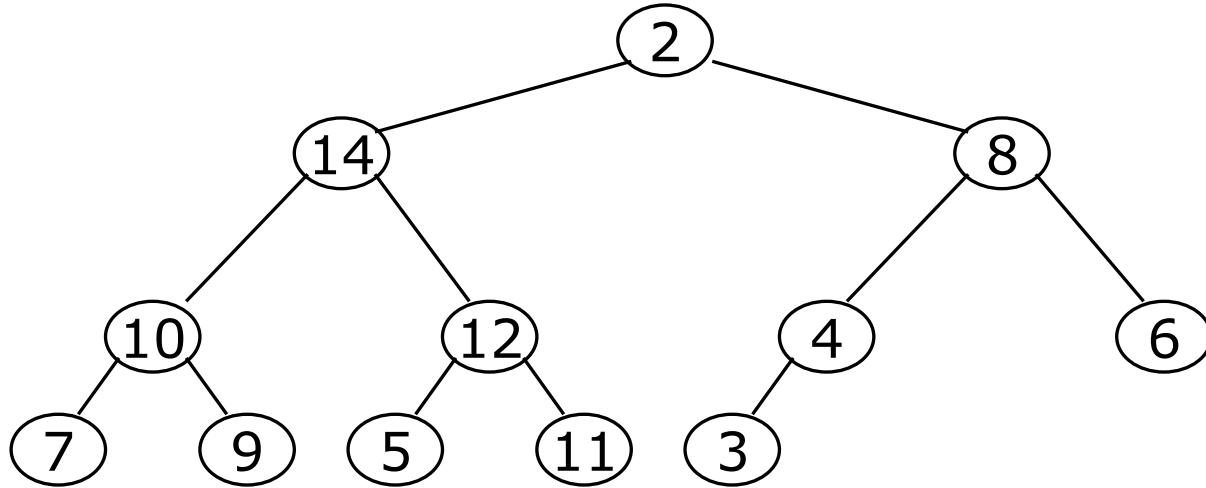
- Since, a heap is always a complete binary tree, can we represent heap as array easily and effectively ? **YES**
- Similar to array implementation of binary trees
- Root is at index 1
- For any node at index  $i$ 
  - The left child is at index  $2i$
  - The right child is at index  $2i + 1$
  - Parent is at  $\text{floor}(i/2)$



|    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 25 | 22 | 17 | 19 | 22 | 14 | 15 | 18 | 14 | 21 | 3  | 9  | 11 |

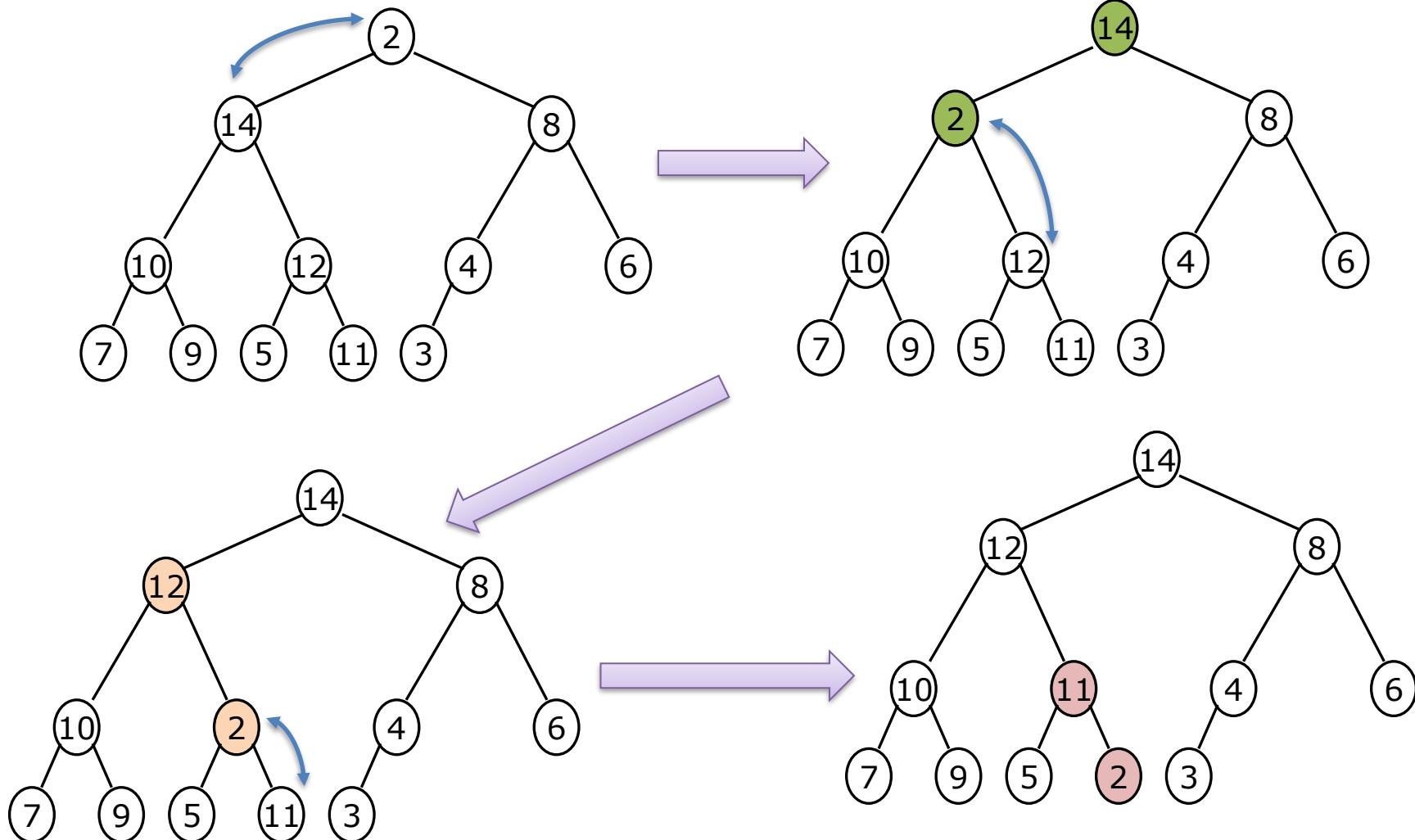
# Heapification (Max-Heapify)

- Before discussing the method for building heap of an arbitrary complete binary tree, we discuss a simpler problem.
- *Let us consider a binary tree in which left and right subtrees of the root satisfy the heap property but **not the root**.* See the following fig:



- Now the Question is how to transform the above tree into a Heap ?
- Heapification !! Commonly referred as Max-Heapify()

# Sequence Depicting the Heapification process



# Algorithm: Max-Heapify(B, s)

## Algorithm 1: Max-Heapify Pseudocode

**Data:**  $B$ : input array;  $s$ : an index of the node

**Result:** Heap tree that obeys max-heap property

**Procedure Max-Heapify( $B, s$ )**

```

left = 2s;
right = 2s + 1;
if left ≤ B.length and B[left] > B[s] then
  | largest =left;
else
  | largest =s;
end
if right ≤ B.length and B[right] > B[largest] then
  | largest =right;
end
if largest ≠ s then
  | swap(B[s], B[largest]);
  | Max-Heapify(B, largest);
end
end
  
```

The time complexity of max-Heapify is **O(log n)**

*\*Since the complete binary tree is perfectly balanced, shifting up a single node takes O(log n) time.*

# Build Heap

- Heap building can be done efficiently with **bottom up fashion**.
- Given an arbitrary complete binary tree, we can assume each leaf is a heap
- Start building the heap from the *parents of these leaves* i.e., Max-Heapify subtrees rooted at the parents.
- The Heapify process continues till we reach the root of the tree.

---

## Algorithm 2: Building a Max-Heap Pseudocode

**Data:**  $B$ : input array

**Result:** Heap tree

**Procedure Max-Heap-Building( $B$ )**

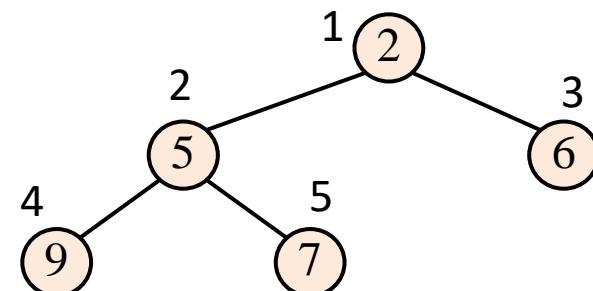
```
 $B.\text{heapsize} = B.\text{length};$ 
```

```
for  $k = B.\text{length}/2$  down to 1 do  
| Max-Heapify( $B$ ,  $k$ );
```

```
end
```

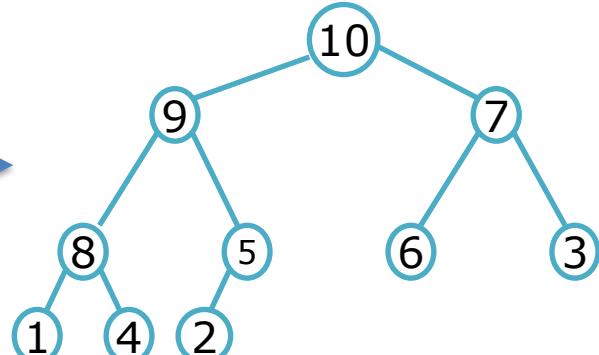
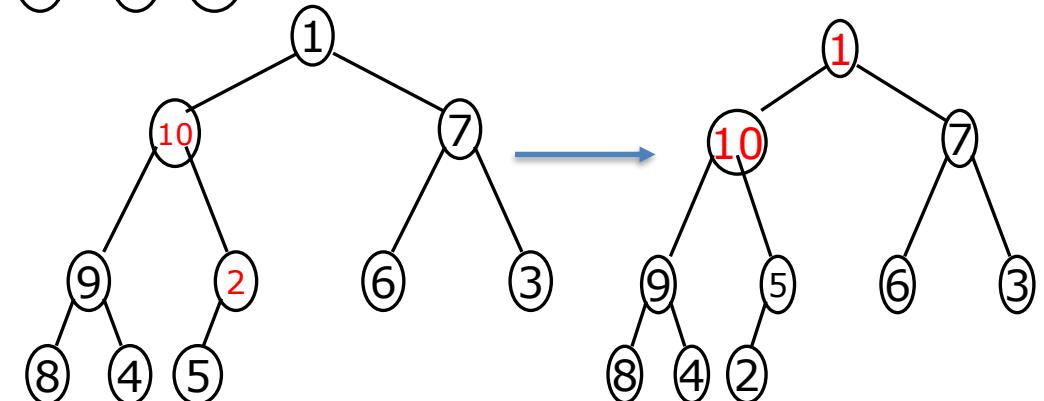
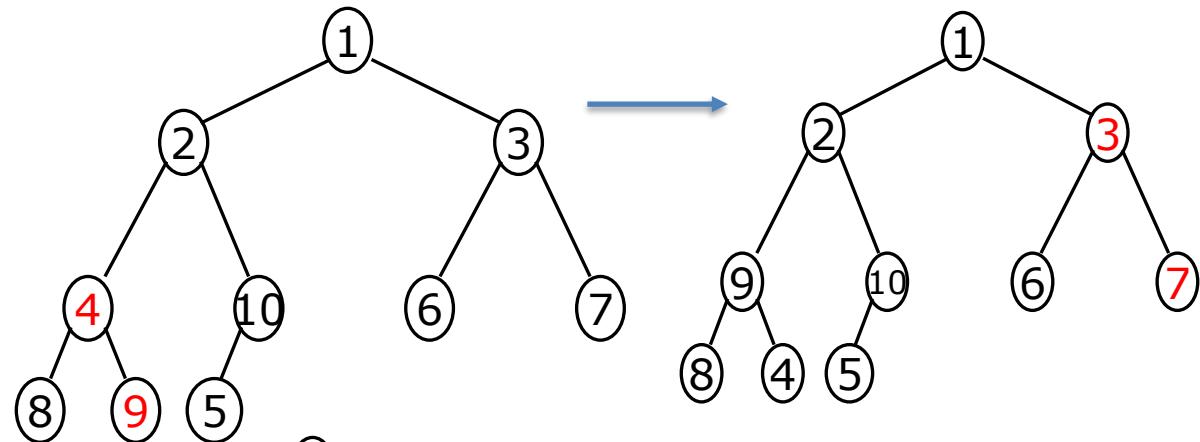
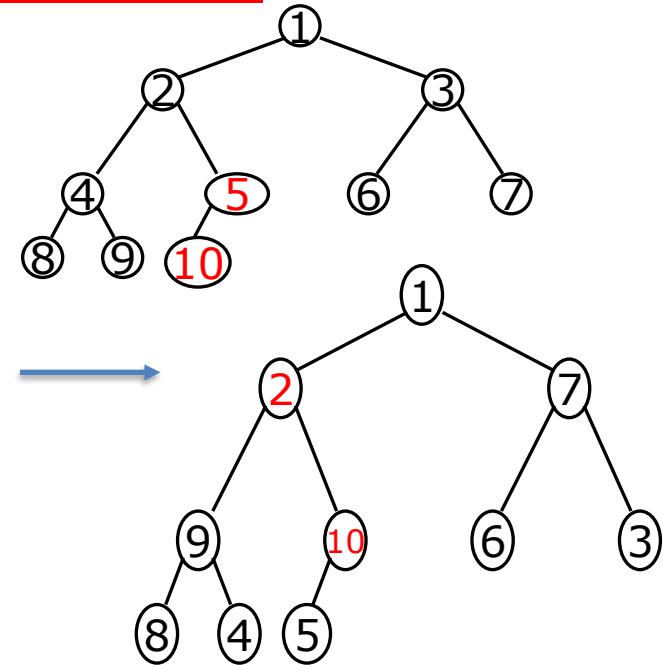
```
end
```

All leaf nodes are from  $[n/2] + 1$  to  $n$   
 All non-leaf nodes are from 1 to  $[n/2]$

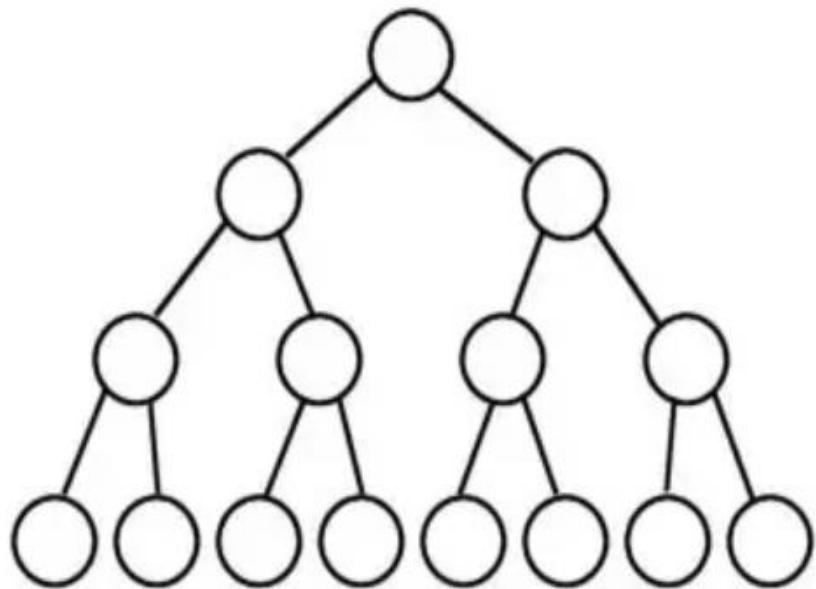


# Build Heap

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |



# Build Heap Analysis



We are calling *MAX\_HEAPIFY()* on  $n/2$  nodes (as leaf's are already a heap).

Since we call *MAX\_HEAPIFY*  $O(n)$  times and *MAX\_HEAPIFY* takes  $O(\log n)$ , the overall complexity is  $O(n \log n)$ .

But this is not tight! We can also prove that building a heap is not  $O(n \log n)$  but just  **$O(n)$** .

There is a beautiful proof about this in CLRS using progression. Please refer the same and start a discussion if required!

# Insertion into a Heap

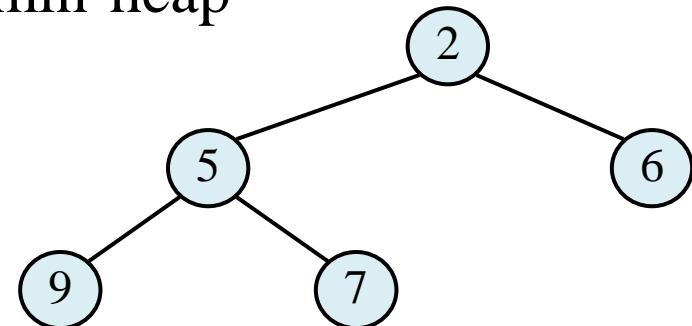
Inserting an element  $e$  in the heap has

- **Three steps**

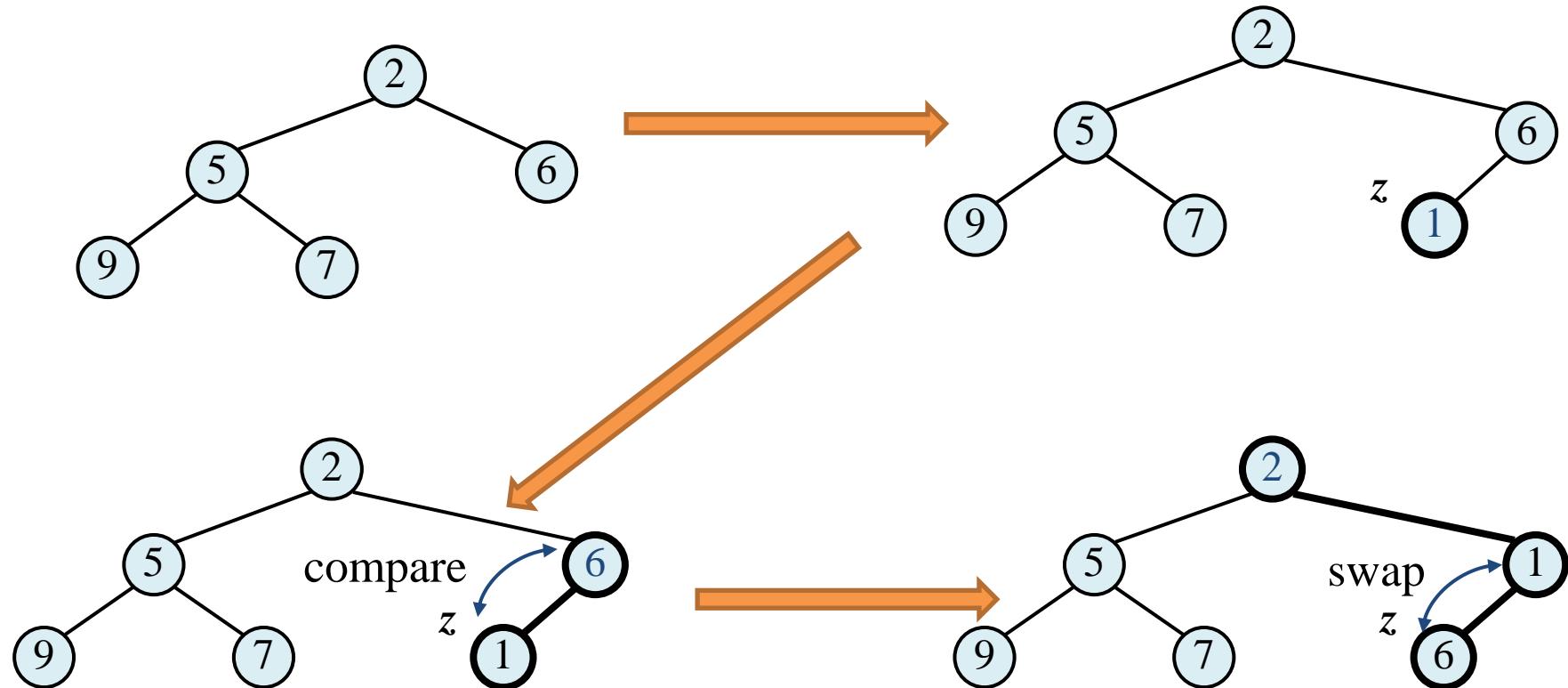
- Find the insertion point  $z$ 
  - So that we maintain complete binary tree property
- Store  $e$  at insertion point  $z$
- Check if the heap follows heap-order property
  - Restore the heap-order property by *Up-heap bubbling*

Example:

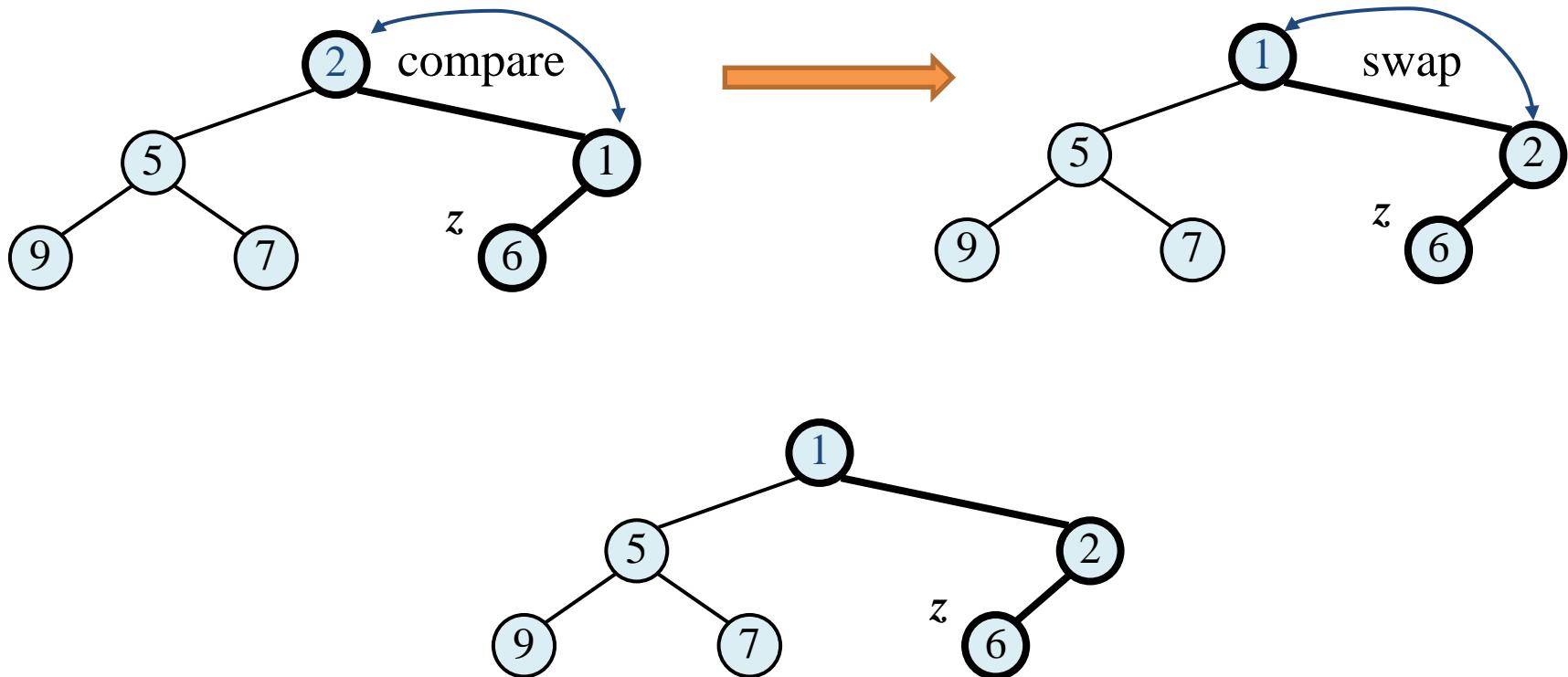
- Insert the element 1 into the min-heap



# Insertion into a Heap



# Insertion into a Heap



# Insertion into a Heap

- After the insertion of a new element  $e$ , the heap-order property may be violated.
- Up-heap bubbling restores the heap-order property
  - Compare and swap  $e$  along an upward path from the insertion point
  - Up-heap bubbling terminates when the element  $e$  reaches
    - the root
    - a node where the heap order property is satisfied
- Since the heap has a complete binary tree structure, its height =  $\log n$  (where  $n$  is no of elements). In the worst case (element inserted at the bottom has to be swapped at every level from bottom to top up to the root node to maintain the heap property), 1 swap is needed on every level. Therefore, the maximum no of times this swap is performed is  $\log n$ . Hence, Insertion in a heap takes **O(log n) time**.

# Removal from a Heap

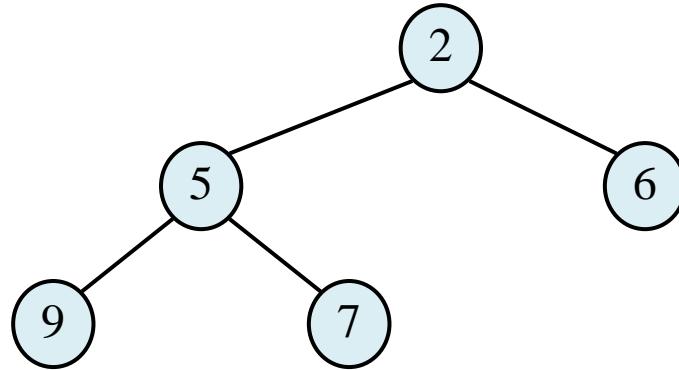
---

## Three steps

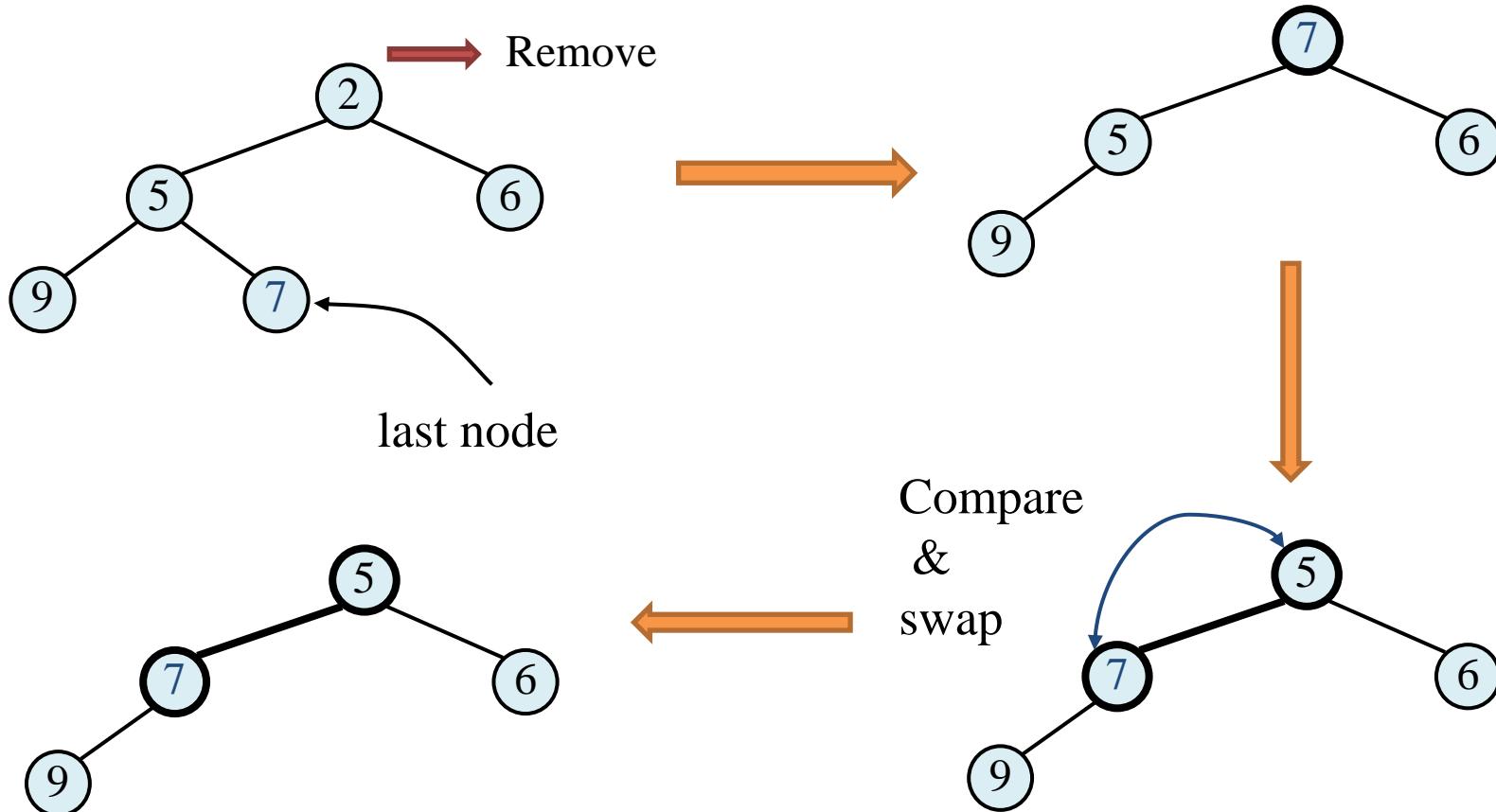
- Remove the element **at the root node** from the heap
- Fill the root node with the element from **the last node**
  - maintain Complete binary tree property
- Check if the heap follows the heap-order property
  - Restore the heap-order property by *down-heap bubbling*

# Removal from a Heap

Example: Perform a delete operation on the given min-heap



# Removal from a Heap



# Removal from a Heap

---

- After replacing the root with the element from the last node  $e$ , the heap-order property may be violated
- **Down-heap bubbling** restores the heap-order property
  - Compare and swap  $e$  along a downward path from the root node
    - Choose the eligible (min/max) child of  $e$  and swap it with  $e$
  - Down-heap terminates when the element  $e$  reaches
    - A leaf
    - A node where the heap order property is satisfied
- Here again, in worst case, we may have to perform down-heap bubbling till the node reaches the leaf. Complexity is  **$O(\log n)$**

# Exercise 1

---

## Min-Heap

- Illustrate the result of inserting the elements 35, 33, 42, 10, 14, 19 and 27 one at a time, into an initially empty binary min-heap in that order. Draw the resulting min-heap after each insertion.
- Perform 2 delete operation for the min-heap constructed in the earlier example.

## Max Heap

- Illustrate the result of inserting the elements 35, 33, 42, 10, 14, 19 and 27 one at a time, into an initially empty binary max-heap in that order. Draw the resulting max-heap after each insertion.
- Perform 2 delete operation for the max-heap constructed in the earlier example.

# Exercise 2

---

Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?

25, 14, 16, 13, 10, 8, 12

25, 12, 16, 13, 10, 8, 14

25, 14, 12, 13, 10, 8, 16

25, 14, 13, 16, 10, 8, 12

Draw the heap structure and find out the right answer/s

# Heap-Sort Algorithm

In-Place: A sorting algorithm is said to be “in-place” if it moves the items within the array itself and, thus, requires only a small  $O(1)$  amount of extra storage.

- Heap sort is one of the best sorting methods being **in-place** and with no quadratic worst-case scenarios.
- Heap sort is divided into two basic parts :
  - Creating a heap of the unsorted list
  - Then a sorted array is created by repeatedly removing the largest/smallest element form the heap and inserting it into the array
  - *Heap is reconstructed after each removal*

# Why study Heapsort?

---

- It is a well-known, traditional sorting algorithm you will be expected to know!!
- Heapsort is *always*  $O(n \log n)$
- Heapsort is a *really cool* algorithm!

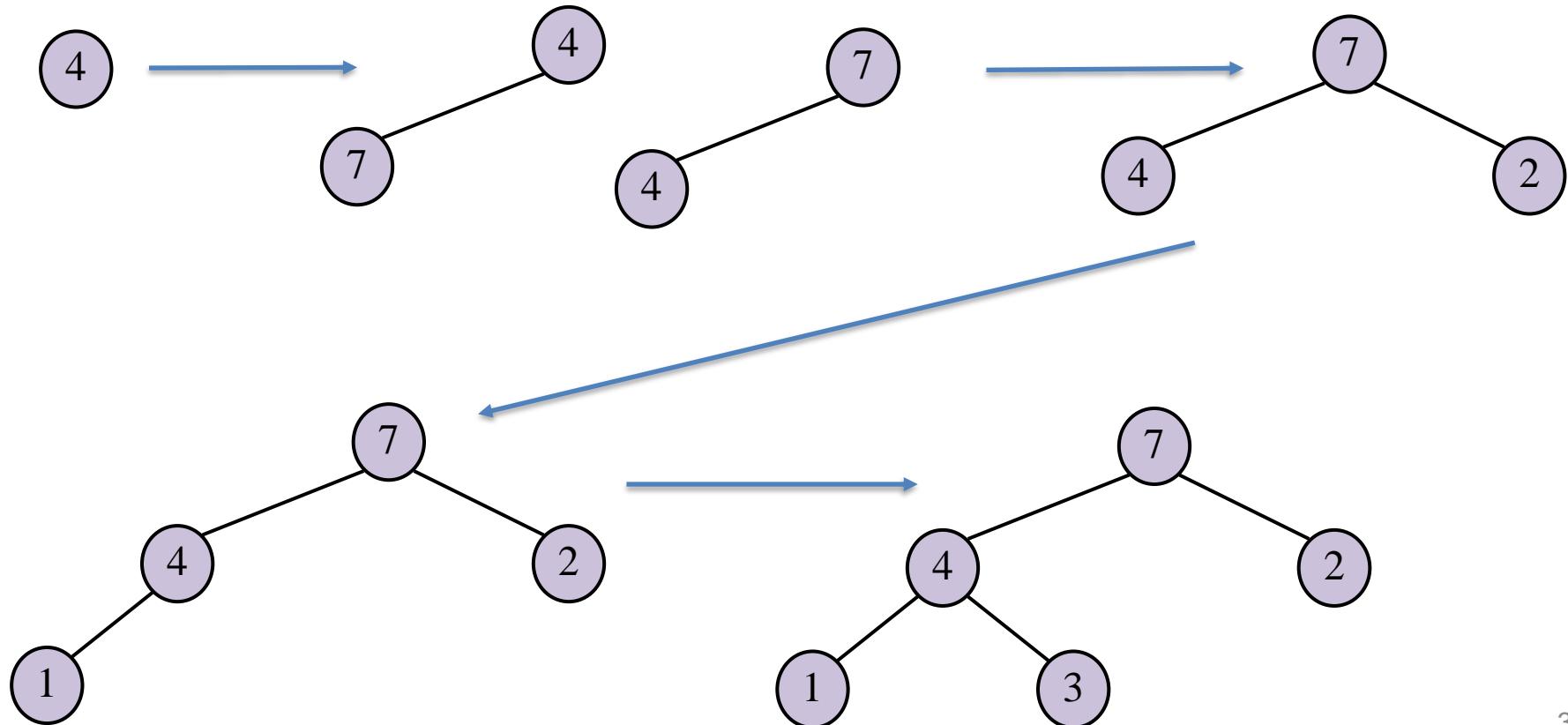
# Heap Sort

- Given an array of n element, first we build the heap ..
- The largest element is at the root, but its position in sorted array should be at last. So swap the root with the last.
- We have placed the highest element in its correct position we left with an array of n-1 elements. Repeat the same of these remaining n-1 element to place the next largest elements in its correct position.
- Repeat the above step till all elements are placed in their correct positions.
- For increasing (ascending) order → Create a Max-Heap
- For a decreasing (descending) order → Create a Min-Heap

# Heap Sort Example

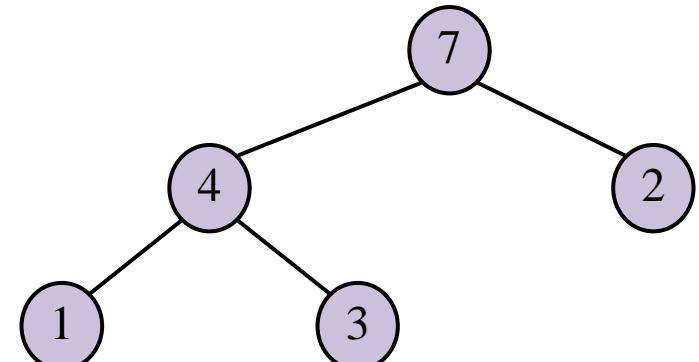
Illustrate heap sort for the given array  $S = [4, 7, 2, 1, 3]$ . Sort  $S$  in increasing order.

First phase: Build max-heap



# Heap Sort Example

|         | S[1]                                  | S[2] | S[3] | S[4] | S[5] |
|---------|---------------------------------------|------|------|------|------|
| Initial | 4                                     | 7    | 2    | 1    | 3    |
| i=1     | 4                                     | 7    | 2    | 1    | 3    |
| i =2    | 7                                     | 4    | 2    | 1    | 3    |
| i =3    | 7                                     | 4    | 2    | 1    | 3    |
| i =4    | 7                                     | 4    | 2    | 1    | 3    |
| i =5    | 7                                     | 4    | 2    | 1    | 3    |
|         | Represents heap                       |      |      |      |      |
|         | Represents array elements not in heap |      |      |      |      |

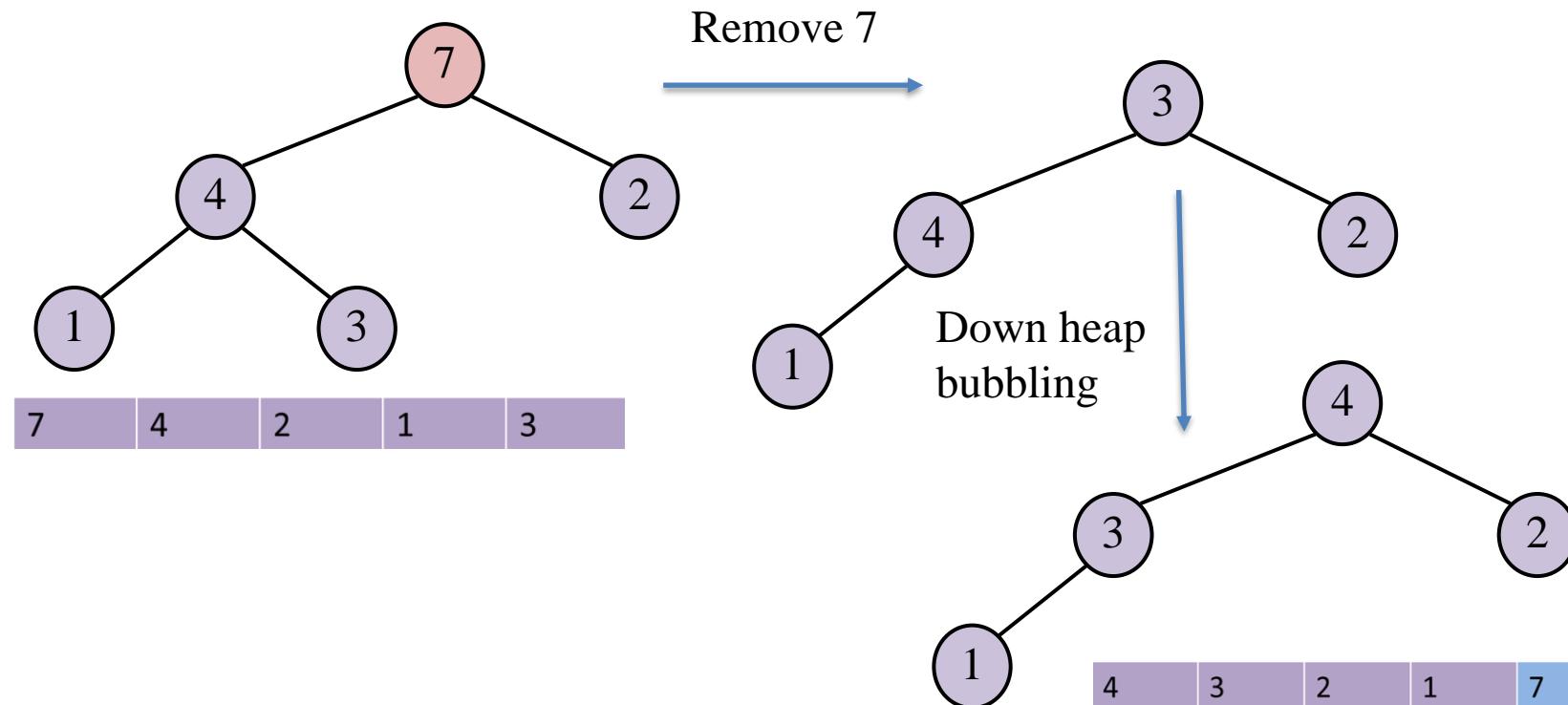


*Phase 1: Heap Creation is completed!*

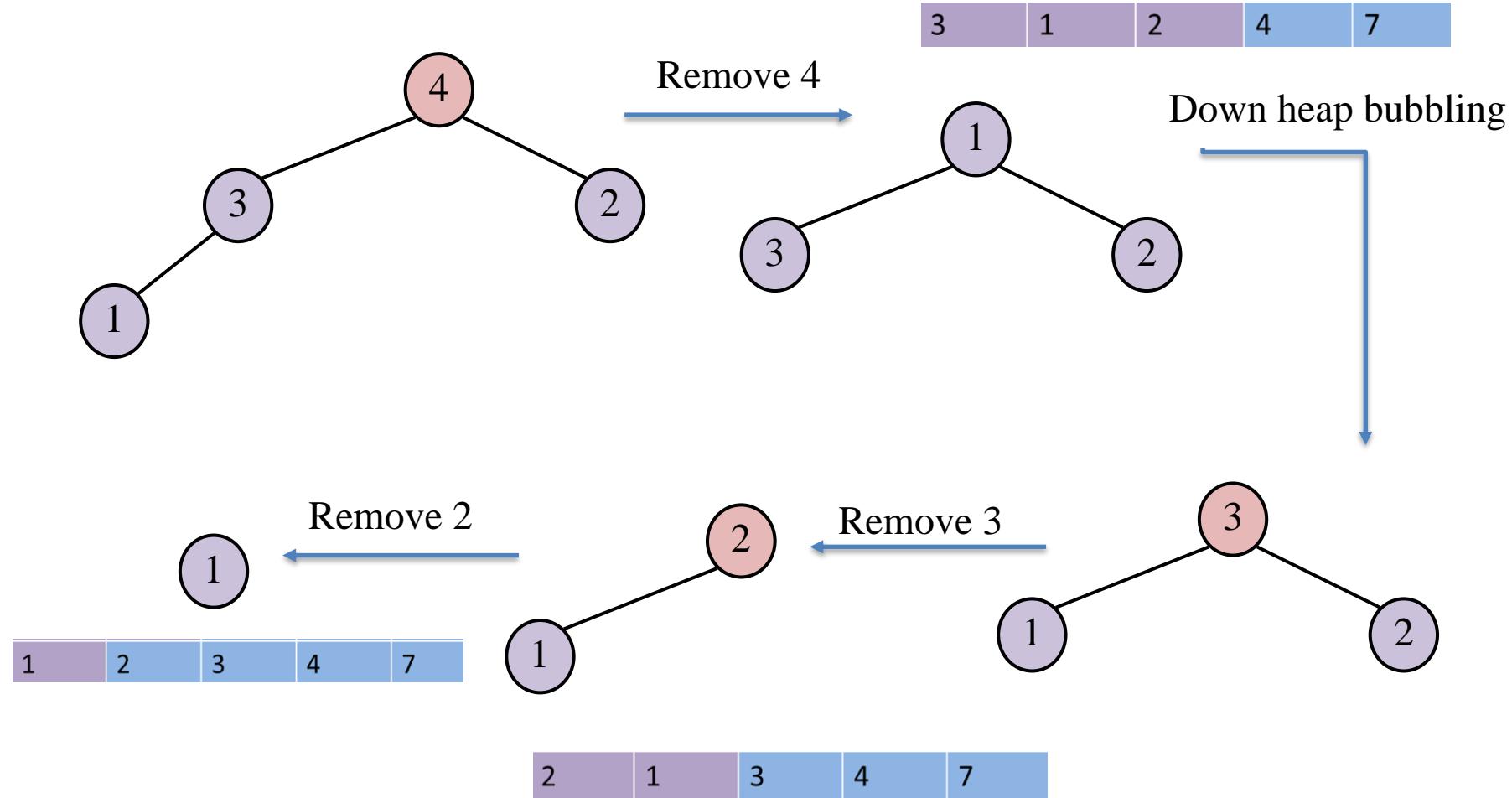
# Heap Sort Example

Heap S= [7, 4, 2, 1, 3]

Second phase: Remove elements from heap and add them to the sorted array



# Heap Sort Example



# Heap Sort Example

|         | S[1]            | S[2] | S[3] | S[4] | S[5] |
|---------|-----------------|------|------|------|------|
| Initial | 7               | 4    | 2    | 1    | 3    |
| i=1     | 4               | 3    | 2    | 1    | 7    |
| i =2    | 3               | 1    | 2    | 4    | 7    |
| i =3    | 2               | 1    | 3    | 4    | 7    |
| i =4    | 1               | 2    | 3    | 4    | 7    |
| i =5    | 1               | 2    | 3    | 4    | 7    |
|         | Represents heap |      |      |      |      |
|         | Sorted array    |      |      |      |      |

*Phase 2: Heap Deletion is completed! And result is sorted elements!!*

# Pseudocode for Heap Sort

## Heapsort(A)

1. Build-Max-Heap(A)
2. for  $i \leftarrow \text{length}[A]$  **downto** 2
3.     **do** exchange  $A[1] \leftrightarrow A[i]$
4.                  $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5.     Max-Heapify(A,1)

*The time complexity of the heap sort algorithm is in ?*

*Phase 1 – Building the heap takes  $O(n)$*

*Phase 2 – Remove the roots till we are left with only 1 element ( $\log n$ )*

*Overall Complexity :  $O(n \log n)$*

# Exercise 3

---

- Given set of elements: 16,14,10,8,7,9,3,2,4,1

## Exercise 1:

- Implement Heap Sort by showing each step and the resultant must be in increasing order. Hint: Create max-heap!

## Exercise 2:

- Implement Heap Sort by showing each step and the resultant must be in decreasing order. Hint: Create min-heap!

# Exercise 4

Find K'th smallest element in an array using Heap.

**Input:**

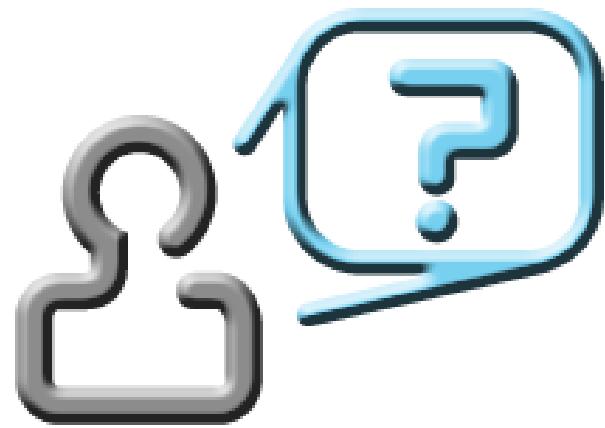
```
arr = [7, 4, 6, 3, 9, 1]  
k = 3
```

**Output:**

```
k'th smallest element in the array is 4
```

**Procedure**

1. Construct a min-heap of size 'N'
2. Pop first K-1 elements from it
3. Now K'th smallest element will reside at the root of the min-heap.



*See you in the next class to explore Graphs!*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : CC BY-NC-SA 4.0





**BITS Pilani**

Pilani | Dubai | Goa | Hyderabad

# Data Structures and Algorithms Design

**DSECLZG519**

**Parthasarathy**





# Contact Session #6

# DSECLZG519 – Introduction to Graphs

# Agenda for CS #6

---

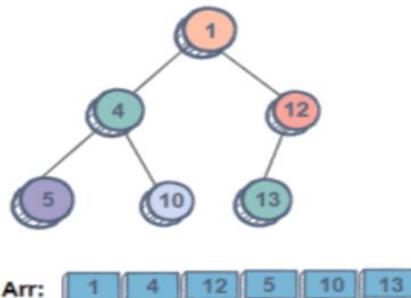
- 1) Recap of CS#5
- 2) Priority Queues & Heaps
- 3) Introduction to Graphs
  - What is a Graph ?
  - Types of Graph
  - Terminologies
  - Applications
- 4) Graph Implementation
  - Adjacency matrix
  - Adjacency List
  - Edge List
- 5) Exercises

# Recap of #5

- Heaps & Types
- Heapification
  - Max-Heapify()
  - Min-Heapify()
- Building Heap
  - Using *bottom up* using appropriate heapification
  - Using *top down* using repetitive insertion
- Insertion
  - Up Heap bubbling
- Deletion
  - Down Heap bubbling
- Application 1: Heap Sort

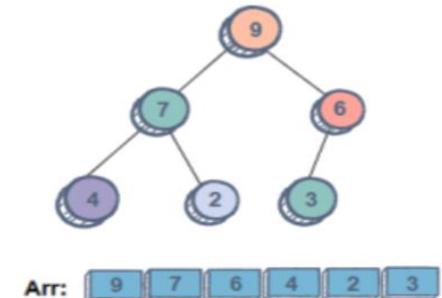
## Min-Heap

- The root node has the **minimum** value.
- The value of each node is equal to or greater than the value of its parent node.
- A complete binary tree.



## Max-Heap

- The root node has the **maximum** value.
- The value of each node is equal to or less than the value of its parent node.
- A complete binary tree.



# Priority Queues

## (An Application of Heap)

- A priority queue is a data structure for maintaining a set  $S$  of elements, each with an associated value called a key/priority.
- There's no “real” FIFO rule anymore.
- Two kinds : max-priority queues and min-priority queues, according to max-heaps and min-heaps.
- The key denotes the **priority**
- Max-heap is used to implement a max-Priority Queue
  - Always deletes & returns (extracts) an element with maximum priority
- Min-heap is used to implement a min-Priority Queue
  - Always deletes & returns (extracts) an element with minimum priority

# Max – Priority Queue Applications



Example: **job scheduling** on shared computer

- jobs have priorities, are stored in a max-priority queue
- each time a new job is to be scheduled, it's got to be one of highest priority (**Extract-Max** operation)
- new jobs can be inserted using **Insert** operation
- in order to avoid “starvation”, priorities can be increased (**Increase-Key** operation)

# Max-Priority Queue Operations

---

- **Insert**( $S, x$ ) inserts element  $x$  into set  $S$
- **Maximum**( $S$ ) returns element of  $S$  with largest key
- **Extract-Max**( $S$ ) removes and returns element of  $S$  with largest key
- **Increase-Key**( $S, x, k$ ) increases  $x$ 's key to new value  $k$ , assuming  $k$  is at least as large as  $x$ 's old key

*Min-priority queues offer Insert, Minimum, Extract-Min, and Decrease-Key.*

# Max – Priority Queue

---

Heaps are very convenient here:

- using max-heaps, we know that the largest element is in  $A[1]$ : we have  $O(1)$  access to largest element
- removing/inserting elements and increasing keys means that we (basically) can call **Max-Heapify** at the right place (relatively efficient operation)

# Max-Priority Queue Implementation

Heap-Maximum(A)

<----- O(1)

1. **return** A[1]

Heap-Extract-Max(A)

<----- O(log n)

1. if  $heap\text{-size}[A] < 1$
2.     **then error** “heap underflow”
3.  $max \leftarrow A[1]$
4.  $A[1] \leftarrow A[heap\text{-size}[A]]$
5.  $heap\text{-size}[A] \leftarrow heap\text{-size}[A] - 1$
6. Max-Heapify(A,1)
7. **return** max



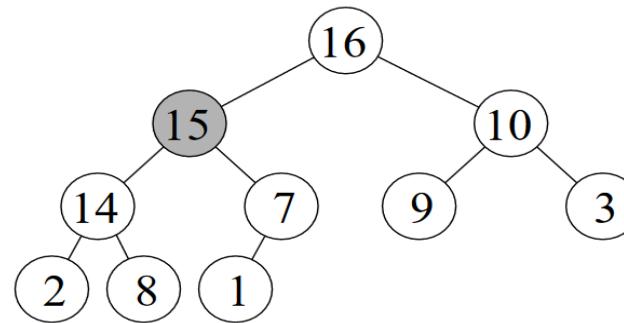
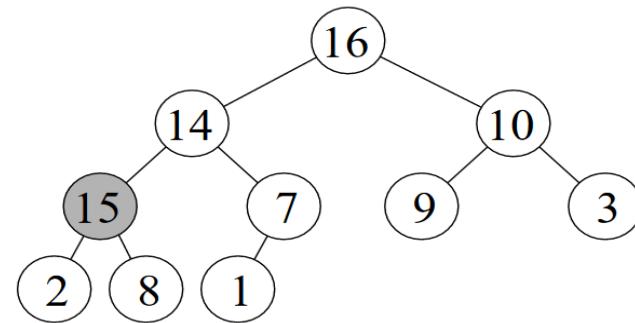
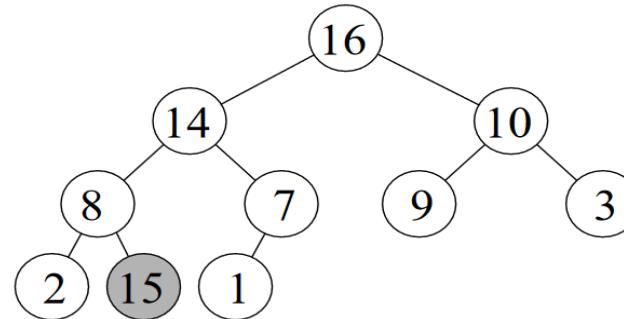
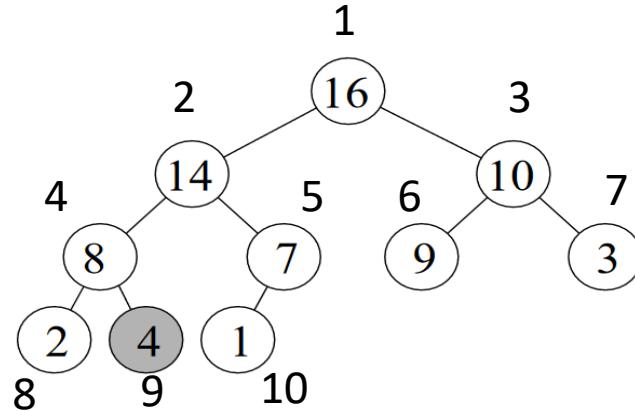
*Technically, this part is removal from heap!*

# Example of Heap-Increase-Key

## Example:

Heap-Increase-Key( $A, 9, 15$ )

Updates node 9 from 4 to 15



# Max-Priority Queue Implementation

Heap-Increase-Key(A,i,key)

1. **if** key < A[i]
2.     **then error** “new key is smaller than current key”
3. A[i]  $\leftarrow$  key
4. **while** i > 1 and A[Parent(i)] < A[i]
5.     **do exchange** A[i]  $\leftrightarrow$  A[Parent(i)]
6.         i  $\leftarrow$  Parent(i)

<----- O(log n)  
Height of the tree

} *Technically, this part is insertion into heap!*

Max-Heap-Insert(A,key)

1. heap-size[A]  $\leftarrow$  heap-size[A]+1
2. A[heap-size[A]]  $\leftarrow -\infty$
3. Heap-Increase-Key(A, heap-size[A], key)

<----- O(log n)

# Applications of Heap

---

- Heaps are used in heapsort!
- *Priority Queues*: Priority queues can be efficiently implemented using Heaps
- *Order statistics*: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.
- Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.

# Exercise 1

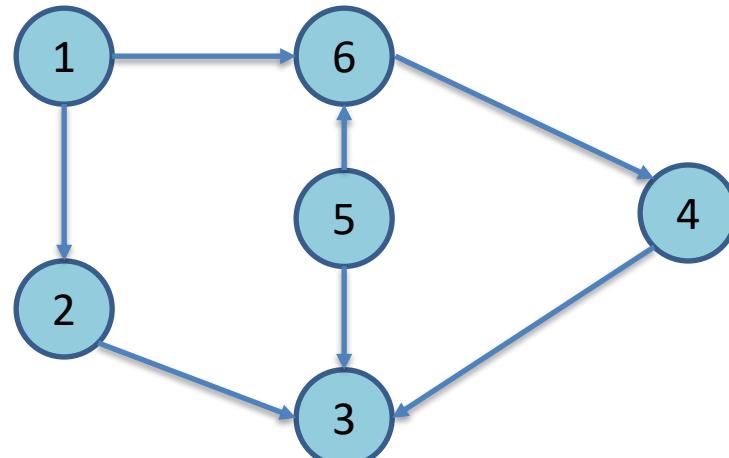
---

- Write an algorithm to compute product of three minimum numbers [least 3 numbers] of a given array using min-heap.
- Example:
  - Original array elements:  
**[12, 74, 9, 50, 61, 41]**
  - Product of three minimum numbers of the given array:  
4428 [ 9 X 12 X 41]
- Write both steps and the algorithm
- Analyze the complexity of your solution

# Graph

A graph  $G$  is defined as a pair of two sets  $V$  and  $E$ , where

- $V$  is a set of nodes, called **vertices**
- $E$  is a collection of pairs of vertices, called **edges**

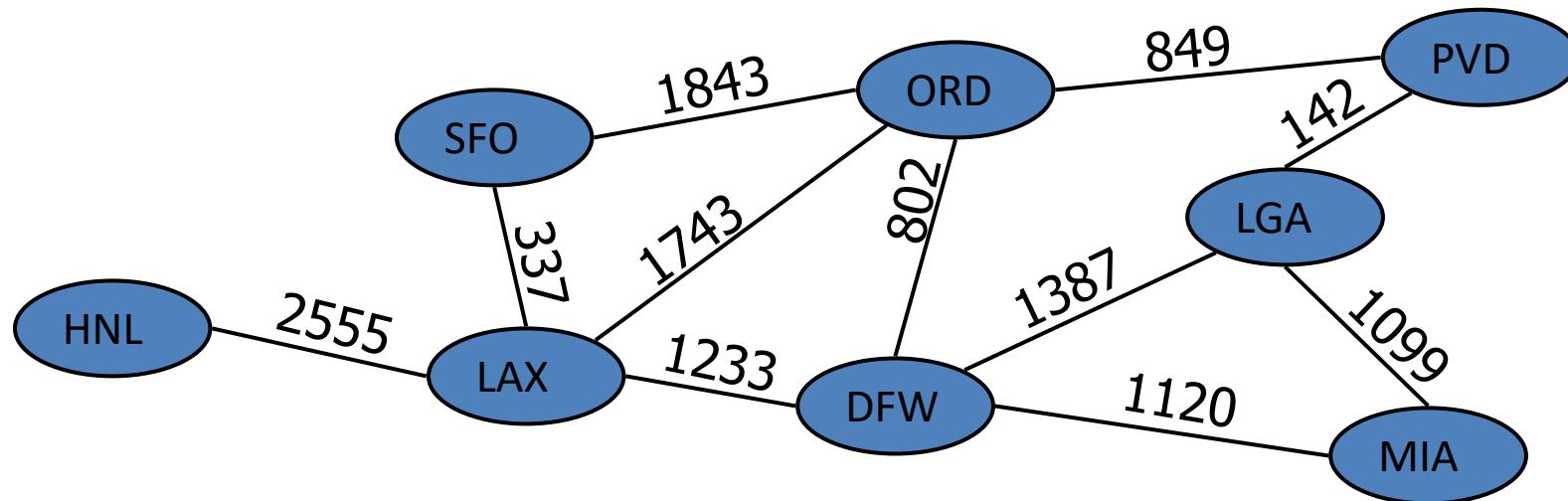


- $G = (V, E)$
- $V = \{ 1, 2, 3, 4, 5, 6 \}$  and  $|V| = 6$
- $E = \{ (1, 6), (1, 2), (2, 6), (2, 3), (3, 6), (3, 4), (4, 6), (5, 6), (5, 3) \}$  and  $|E| = 7$

# Example

## Example:

- A **vertex** represents an **airport** and stores the **three-letter airport code**
- An **edge** represents a flight route between two airports and stores the **mileage** of the route



# Other Examples

---

Can you think of other examples of graphs modelling real - world situations ?

- Google maps !
- Facebook and friends ☺
- World Wide Web and Hyperlinks ...
- Operating Systems for resource allocation
- ...
- ....
- ...

# Applications

## Electronic circuits

- Printed circuit board
- Integrated circuit

## Transportation networks

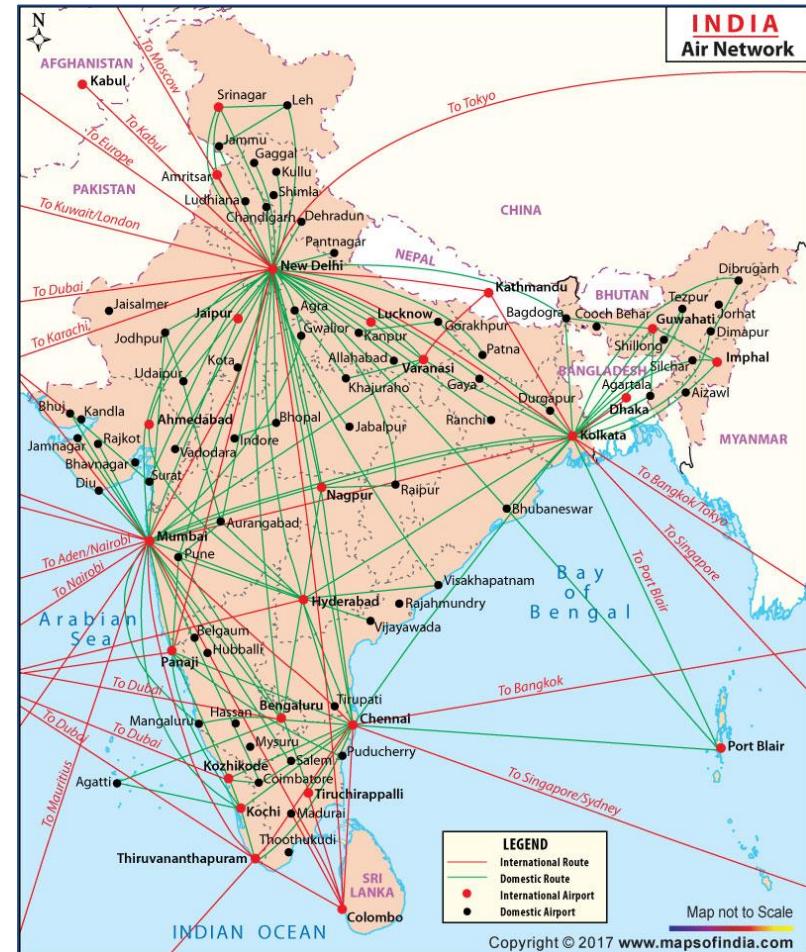
- Highway network
- Flight network

## Computer networks

- Local area network
- Internet
- Web

## Databases

- Entity-relationship diagram



Source- Maps of India

# Tree vs Graph

| BASIS FOR COMPARISON | TREE                                     | GRAPH  |
|----------------------|--|--|
| Path                 | Only one between two vertices.           | More than one path is allowed.               |
| Root node            | It has exactly one root node.            | Graph doesn't have a root node.              |
| Loops                | No loops are permitted.                  | Graph can have loops.                        |
| Complexity           | Less complex                             | More complex comparatively                   |
| Traversal techniques | Pre-order, In-order and Post-order.      | Breadth-first search and depth-first search. |
| Number of edges      | $n-1$ (where $n$ is the number of nodes) | Not defined                                  |
| Model type           | Hierarchical                             | Network                                      |

# Edge Types

## Directed edge (with arrow)

- ordered pair of vertices  $(u,v)$
- first vertex  $u$  is the origin
- second vertex  $v$  is the destination
- e.g., a flight



## Undirected edge (no arrow)

- unordered pair of vertices  $(u,v)$
- e.g., a flight route



## Directed graph

- all the edges are directed
- e.g., flight network

### Analogy:

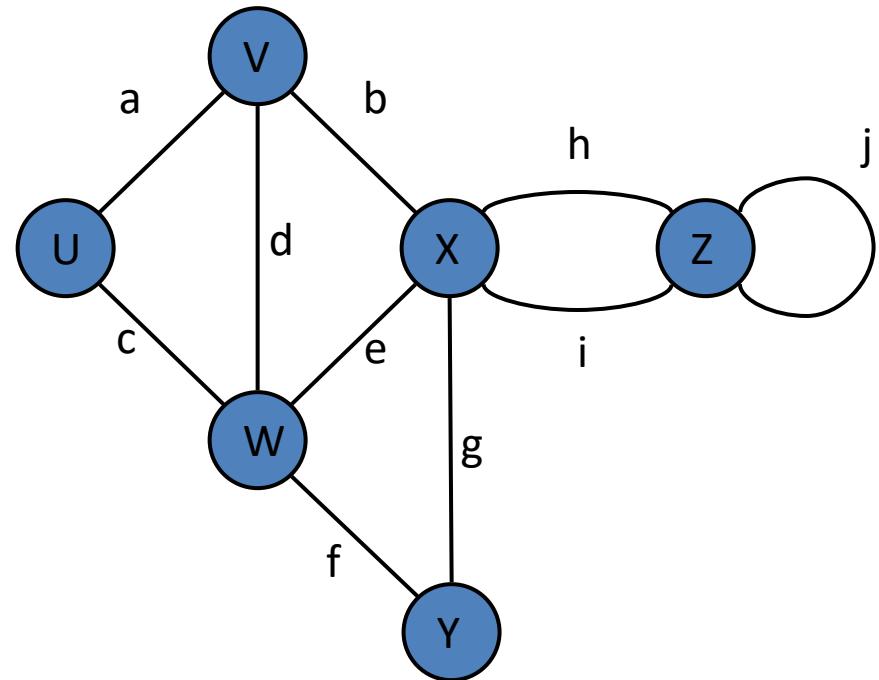
- A road between two points could be **one way** (directed) Or **two way** (undirected)
- Could have more than one edge (e.g. toll road and public road)

## Undirected graph

- all the edges are undirected
- e.g., route network

# Graph Terminology

- **End vertices** (or endpoints) of an edge
  - Ex:  $U$  and  $V$  are the endpoints of a
- **Edges incident on a vertex**
  - Ex:  $a, d,$  and  $b$  are incident on  $V$
- **Adjacent vertices**
  - Ex:  $U$  and  $V$  are adjacent
- **Degree of a vertex**
  - Ex:  $X$  has degree 5
- **Parallel edges**
  - Ex:  $h$  and  $i$  are parallel edges
- **Self-loop**
  - Ex:  $j$  is a self-loop



# Graph Terminology

## Path

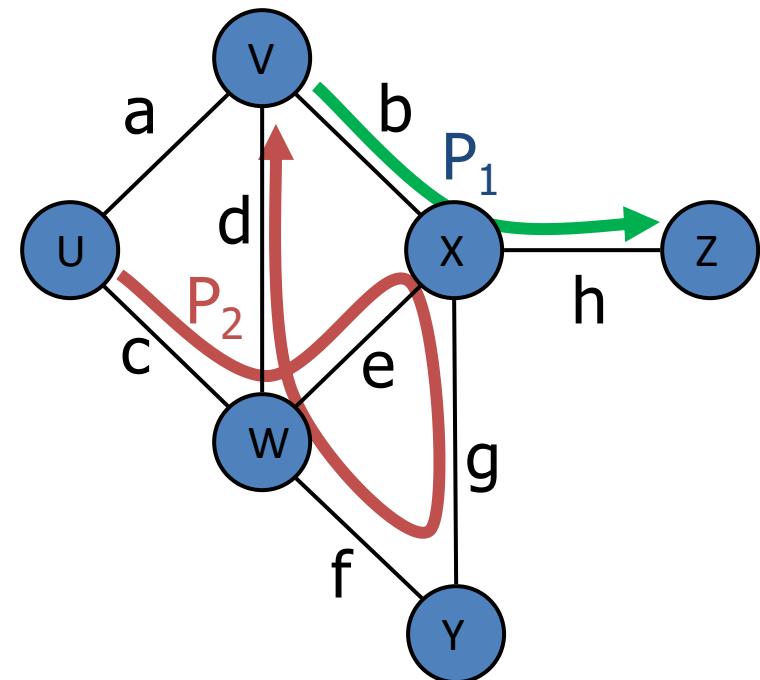
- Sequence of alternating vertices and edges
- Begins with a vertex
- Ends with a vertex
- Each edge is preceded and followed by its endpoints

## Simple path

- Path such that all its vertices and edges are *distinct*

## Examples

- $P_1 = (V, b, X, h, Z)$  is a **simple** path
- $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$  is a path that is **not simple**



# Graph Terminology

## Cycle

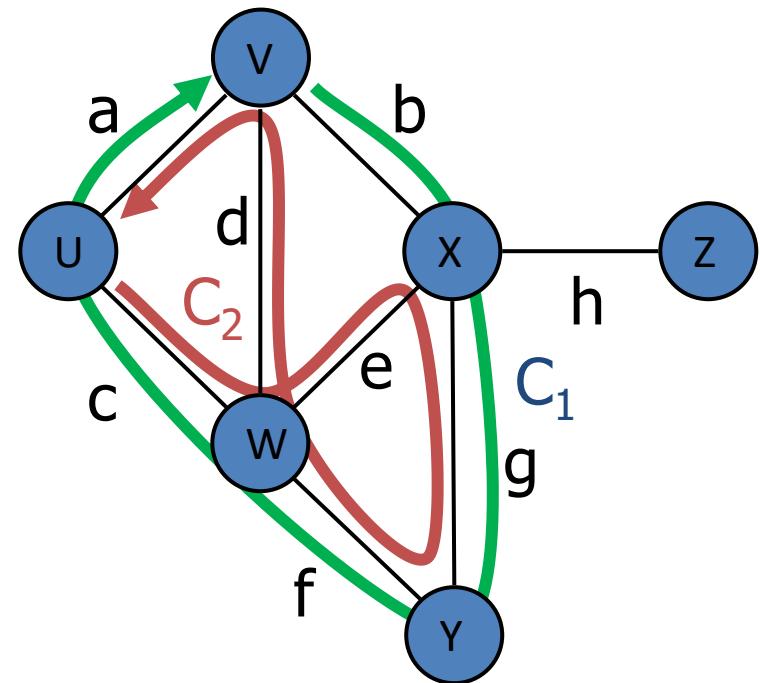
- **Circular** sequence of alternating vertices and edges
- each edge is preceded and followed by its endpoints

## Simple Cycle

- Cycle such that all its vertices and edges are **distinct**

## Examples

- $C_1 = (V, b, X, g, Y, f, W, c, U, a, \leftarrow)$  is a **simple cycle**
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \leftarrow)$  is a cycle that is **not simple**



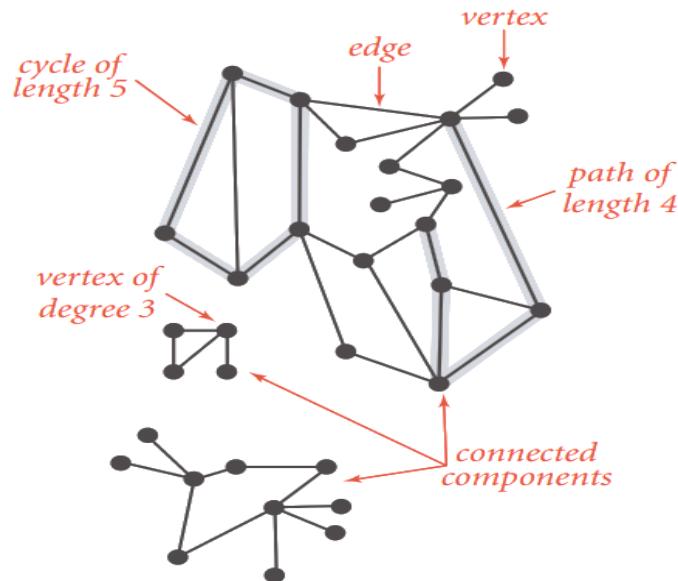
# Graph Terminology

## Graph terminology

**Path.** Sequence of vertices connected by edges.

**Cycle.** Path whose first and last vertices are the same.

Two vertices are **connected** if there is a path between them.



# Graph Types

**Null Graph :** A graph having no edges.



**Trivial Graph :** A graph with only one vertex.



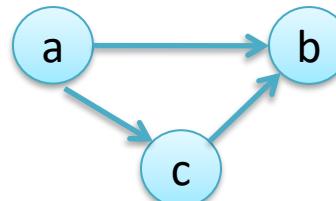
**Undirected Graph :** A graph that contains edges but the edges are not directed ones.



**Directed Graph :** A graph that contains edges which have direction.



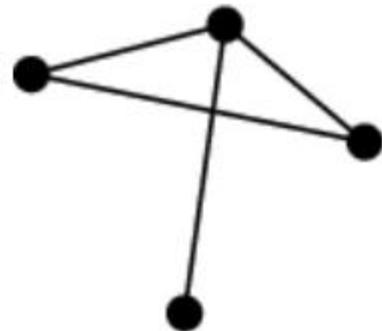
**Simple Graph :** A graph that contains no loops and no parallel edges.



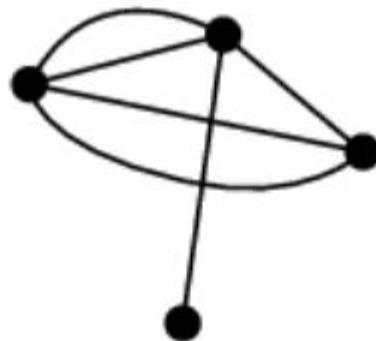
**Many more .....**

# Quick Check

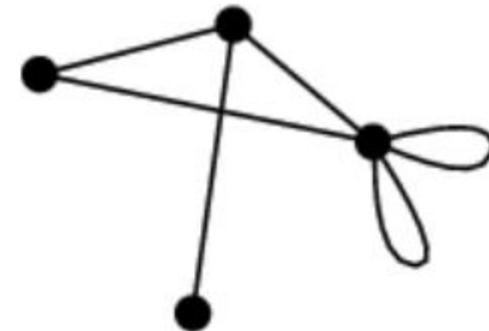
Which of the below is a Simple Graph ?



Simple Graph



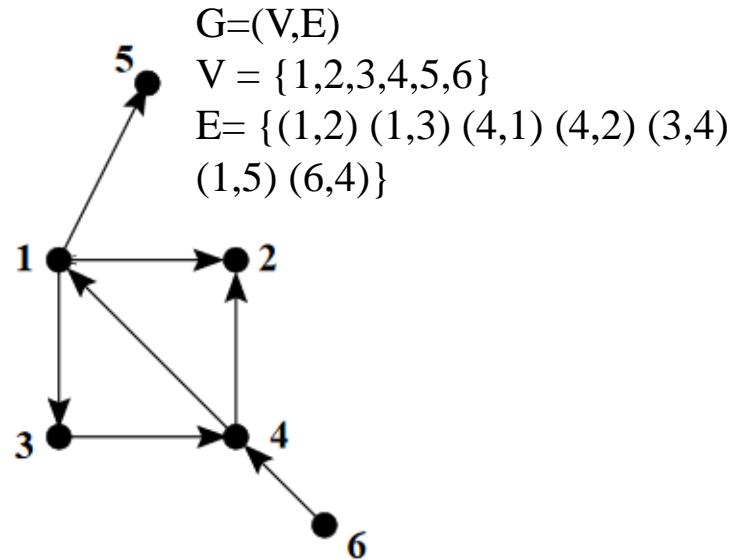
Non-Simple Graph  
with multiple (parallel)  
edges



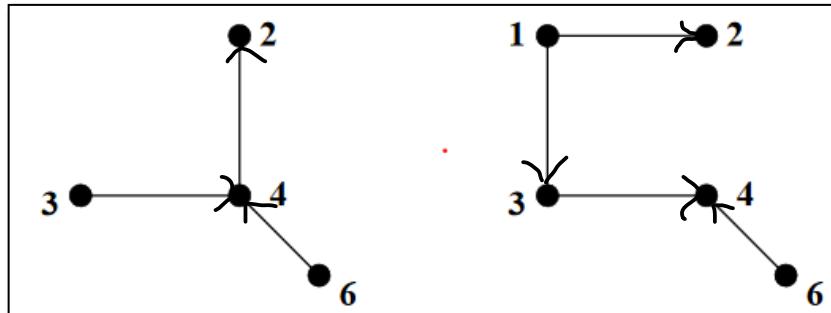
Non-Simple Graph  
with loops

# Subgraphs

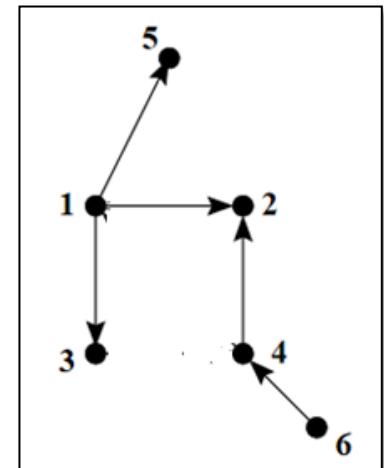
- A **subgraph** S of a graph G is a graph such that :
  - The **vertices** of S are a **subset** of the vertices of G
  - The **edges** of S are a **subset** of the edges of G
- A **spanning subgraph** of G is a subgraph that contains all the **vertices** of G.



## Subgraphs :

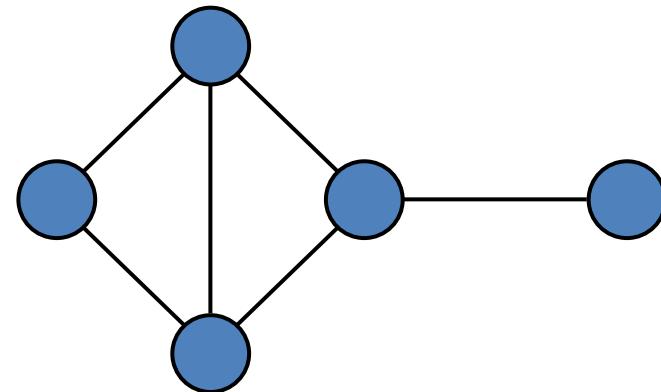


## Spanning subgraph:

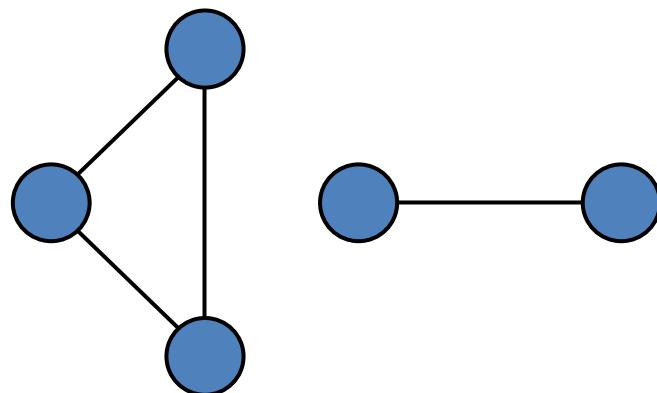


# Connectivity

- A graph is **connected** if there is a path between every pair of vertices.
- From every vertex to any other vertex, there should be some path to traverse.
- That is called the connectivity of a graph. A graph with multiple disconnected vertices and edges is said to be disconnected.



**Connected** graph

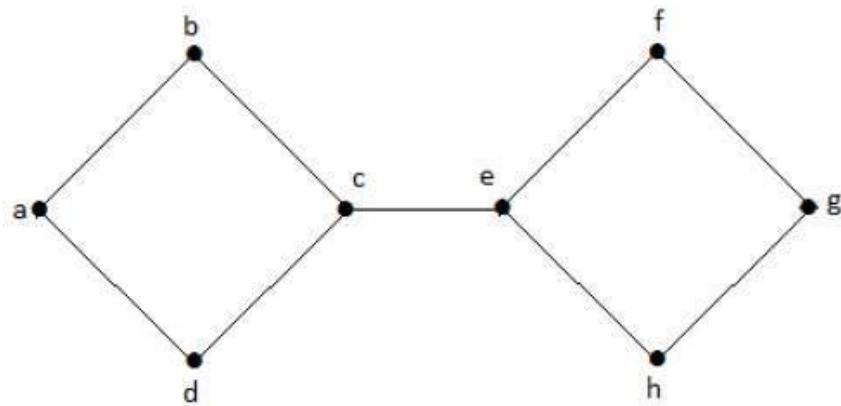


**Non connected** graph with two connected components

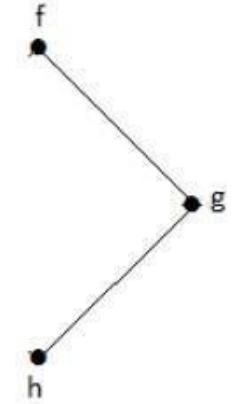
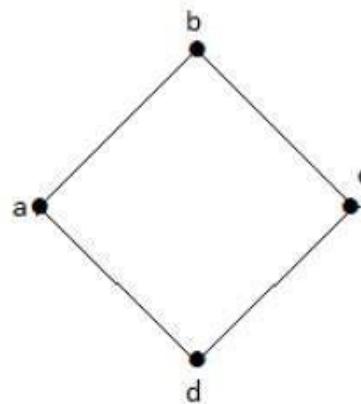
# Connectivity – Cut Vertex

- Let ‘G’ be a connected graph. A vertex  $V \in G$  is called a cut vertex of ‘G’, if ‘ $G-V$ ’ (Delete ‘V’ from ‘G’) results in a disconnected graph. Removing a cut vertex from a graph breaks it in to two or more graphs.

*Note – Removing a cut vertex will render a graph disconnected.*



By removing ‘e’ or ‘c’, the graph will become a disconnected graph.

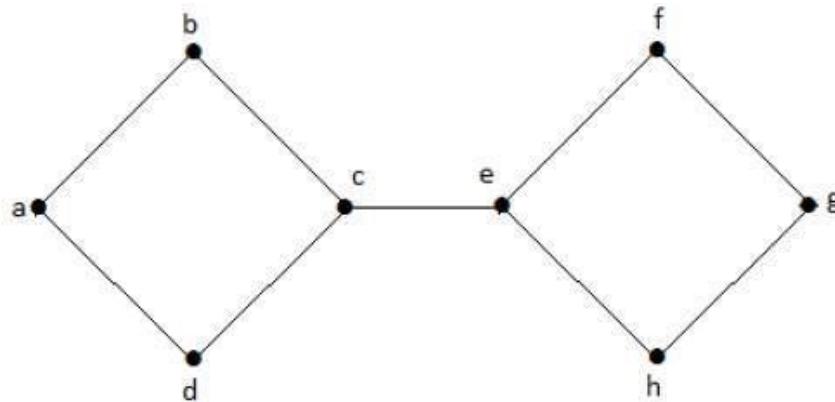


Hence it is a disconnected graph with cut vertex as ‘e’. Similarly, ‘c’ is also a cut vertex for the above graph.

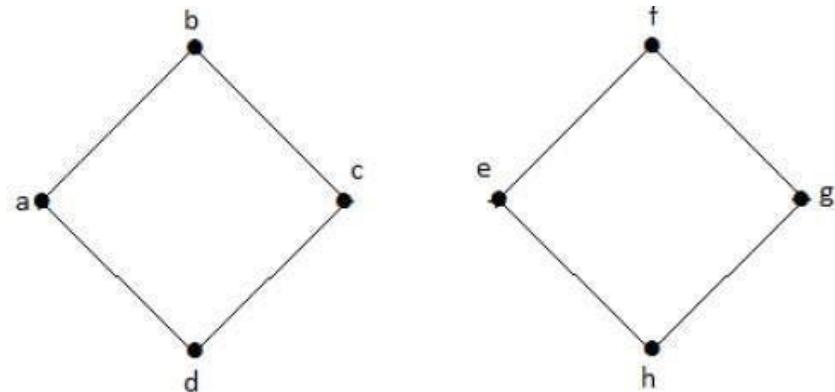
# Connectivity – Cut Edge

- Let ‘G’ be a connected graph. An edge ‘e’  $\in G$  is called a cut edge if ‘G-e’ results in a disconnected graph.
- If removing an edge in a graph results in two or more graphs, then that edge is called a Cut Edge.

*Note – Removing a cut edge will render a graph disconnected.*



By removing the edge (c, e) from the graph, it becomes a disconnected graph.

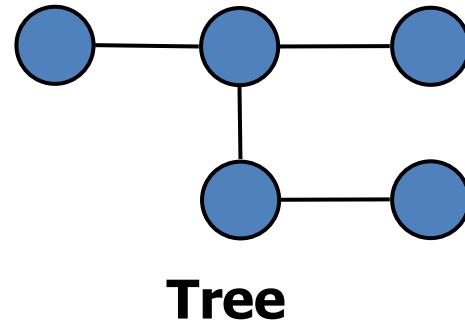


In the above graph, removing the edge (c, e) breaks the graph into two which is nothing but a disconnected graph. Hence, the edge (c, e) is a cut edge of the graph.

# Trees and Forests

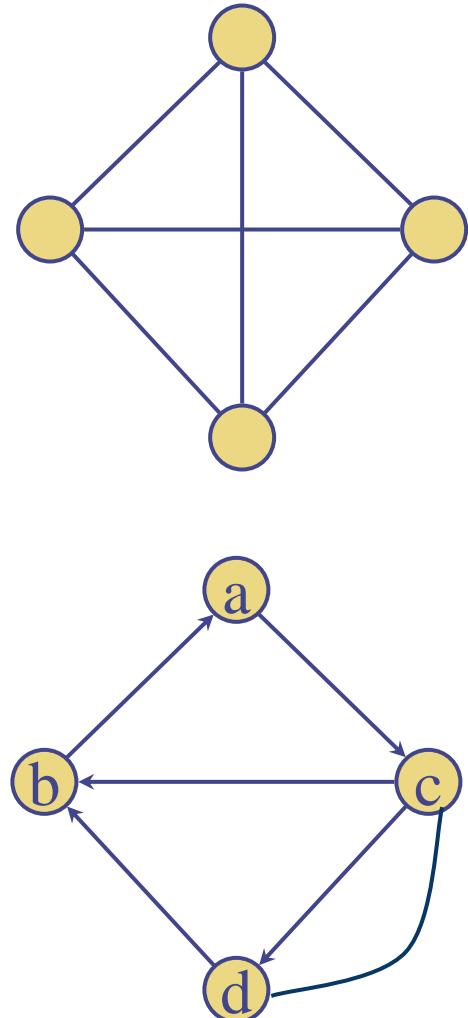
- A (free) **tree** is an undirected graph  $T$  such that
  - $T$  is **connected**
  - $T$  has **no cycles**

*[This definition of tree is different from the one of a rooted tree]*



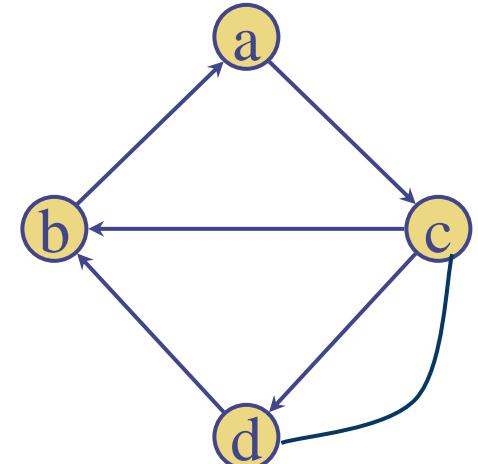
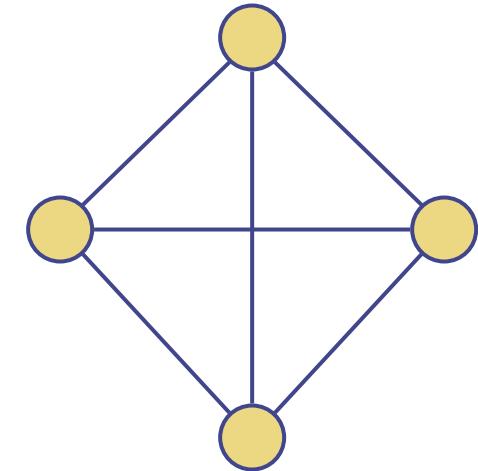
# Operations on a Graph

- Return the number,  $n$ , of vertices in  $G$ .
- Return the number,  $m$ , of edges in  $G$ .
- Return a set or list containing all  $n$  vertices in  $G$ .
- Return a set or list containing all  $m$  edges in  $G$ .
- Return some vertex,  $v$ , in  $G$ .
- Return the degree,  $\deg(v)$ , of a given vertex,  $v$ , in  $G$ .
- Return a set or list containing all the edges incident upon a given vertex,  $v$ , in  $G$ .
- Return a set or list containing all the vertices adjacent to a given vertex,  $v$ , in  $G$ .
- Return the two end vertices of an edge,  $e$ , in  $G$ ; if  $e$  is directed, indicate which vertex is the origin of  $e$  and which is the destination of  $e$ .
- Return whether two given vertices,  $v$  and  $w$ , are adjacent in  $G$ .



# Operations on a Graph

- Indicate whether a given edge,  $e$ , is directed in  $G$ .
- Return the in-degree of  $v$ ,  $\text{inDegree}(v)$ .
- Return a set or list containing all the incoming (or outgoing) edges incident upon a given vertex,  $v$ , in  $G$ .
- Return a set or list containing all the vertices adjacent to a given vertex,  $v$ , along incoming (or outgoing) edges in  $G$ .
- Insert a new directed (or undirected) edge,  $e$ , between two given vertices,  $v$  and  $w$ , in  $G$ .
- Insert a new (isolated) vertex,  $v$ , in  $G$ .
- Remove a given edge,  $e$ , from  $G$ .
- Remove a given vertex,  $v$ , and all its incident edges from  $G$ .

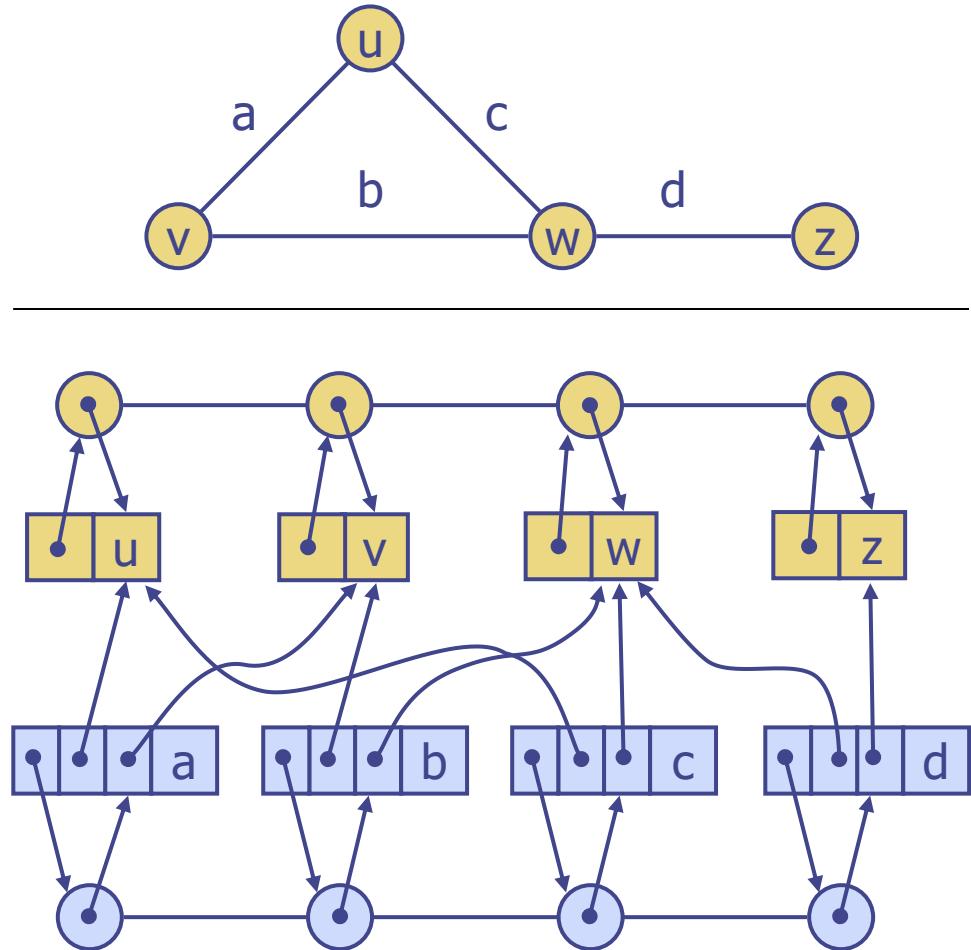


# Graph Representation Strategies

- Edge List
- Adjacency Matrix
- Adjacency List

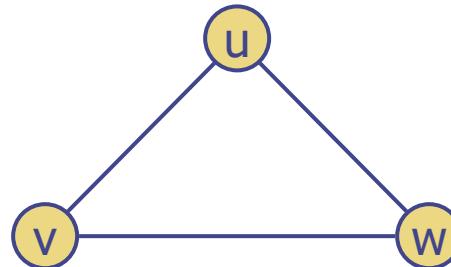
# Edge List

- **Vertex object**
  - element
  - reference to position in vertex sequence
- **Edge object**
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- **Vertex sequence**
  - sequence of vertex objects
- **Edge sequence**
  - sequence of edge objects



# Edge List Simplified

```
[  
    (u, v) ,  
    (v, w) ,  
    (u, w)  
]
```



Example 2:

edge\_list = [(0, 1), (1, 2), (2, 3), (0, 2), (3, 2), (4, 5), (5, 4)]

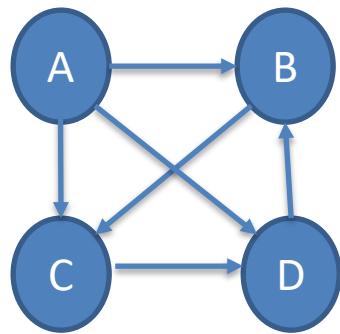
# Adjacency Matrix

The adjacency matrix A of a graph G is formally defined as :

$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j \\ 0 & \text{if there is no edge from vertex } i \text{ to vertex } j \end{cases}$$

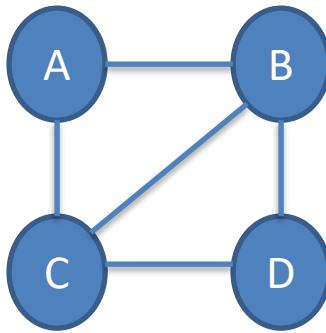
- It is clear from the definition that an adjacency matrix of a graph with n vertices is a boolean square matrix with n rows and n columns with entries 1's and 0's ( bit-matrix)
- Note that the above definition holds true for both directed and undirected graph. *If it's a adjacency matrix of a undirected graph, then the matrix will look symmetric and diagonal being 0's.*
- Note: If there are multiple edges from vertex u to v, then the number of edges can also be used in the matrix.

# Adjacency Matrix Examples



Adjacency Matrix

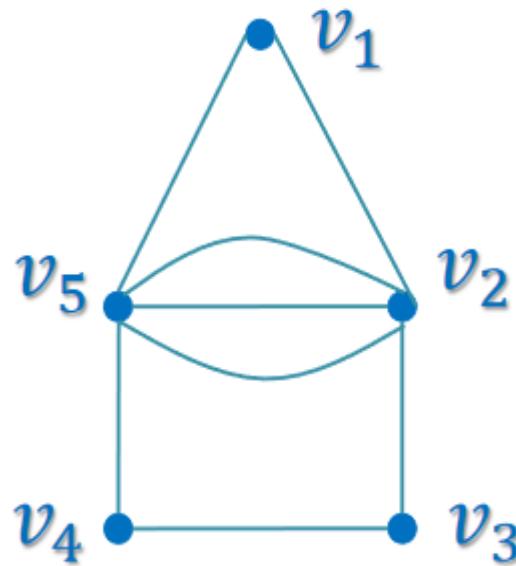
|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 |
| B | 0 | 0 | 1 | 0 |
| C | 0 | 0 | 0 | 1 |
| D | 0 | 1 | 0 | 0 |



Adjacency Matrix

|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 |
| B | 1 | 0 | 1 | 1 |
| C | 1 | 1 | 0 | 1 |
| D | 0 | 1 | 1 | 0 |

# Adjacency Matrix Examples



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 3 & 0 & 1 & 0 \end{pmatrix}$$

# Adjacency Matrix: Pros and Cons

---

## *Advantages*

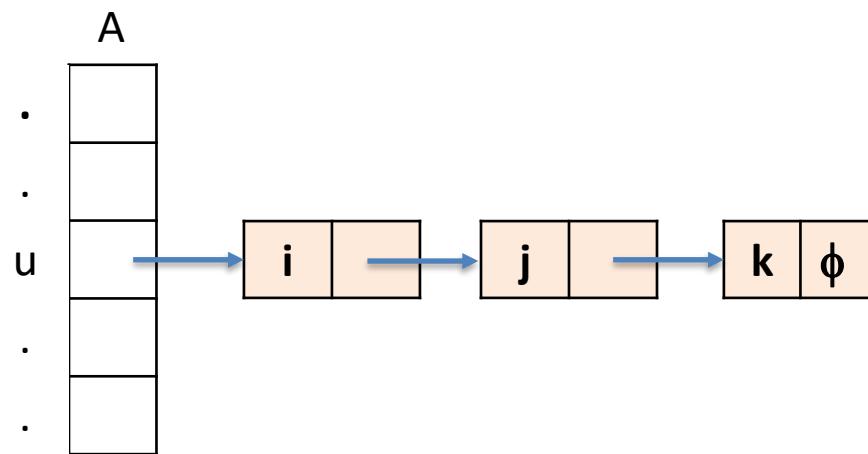
- Fast to tell whether edge exists between any two vertices  $i$  and  $j$  (and to get its weight)

## *Disadvantages*

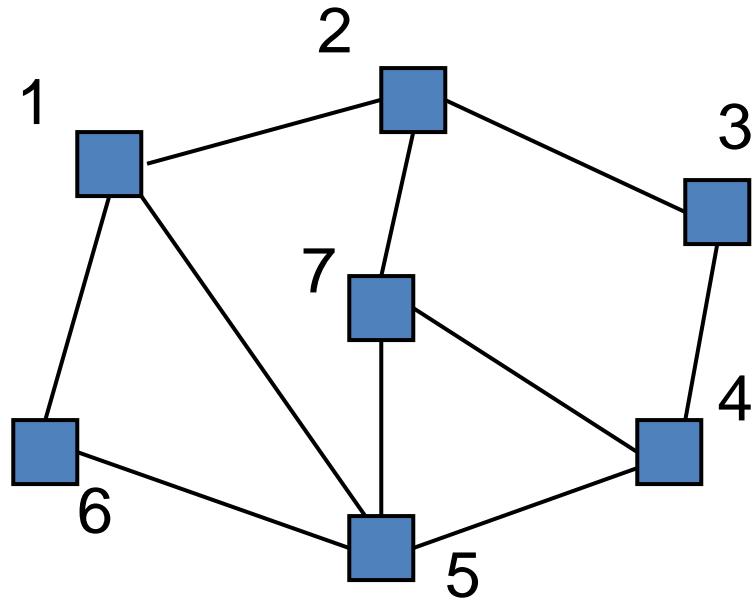
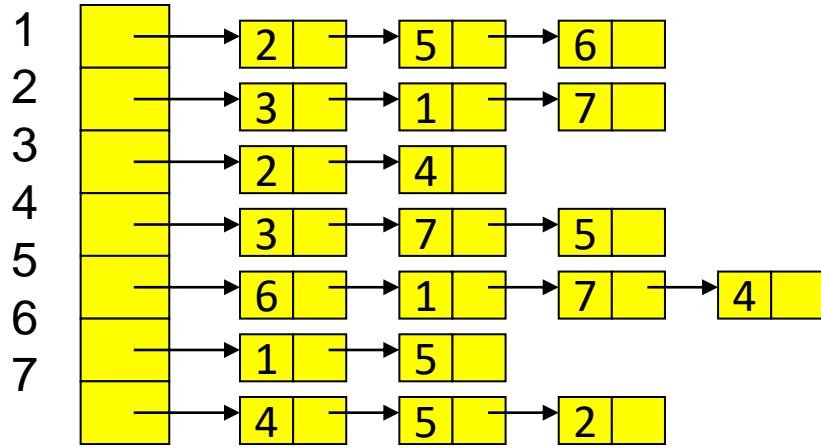
- Consumes a lot of memory on **sparse** graphs (ones with few edges)
- Redundant information for undirected graphs

# Adjacency Lists

- An adjacency linked list is an array of  $n$  linked lists where  $n$  is the number of vertices in graph  $G$ . Each location of the array represents a vertex of the graph. For each vertex  $u \in V$ , a linked list consisting of all the vertices adjacent to  $u$  is created and stored in  $A[u]$ . The resulting array  $A$  is an adjacency list.
- Note: It is clear from the def. that if  $i, j$  and  $k$  are the vertices adjacent to the vertex  $u$ , then  $i, j$  and  $k$  are stored in a linked list and starting address of linked list is stored in  $A[u]$  as shown below:



# Adjacency List Example



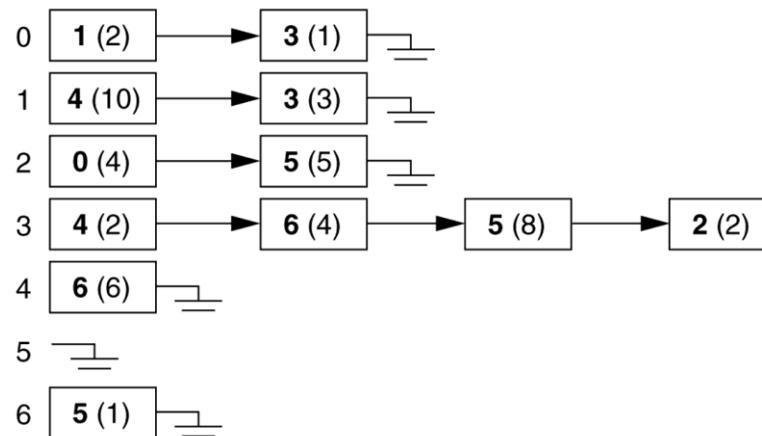
# Adjacency List: Pros and Cons

## *Advantages:*

- new nodes can be added easily
- new nodes can be connected with existing nodes easily
- "who are my neighbors" easily answered

## *Disadvantages:*

- determining whether an edge exists between two nodes:  
 $O(\text{average degree})$



# Asymptotic Performance

| $\nabla$ $n$ vertices, $m$ edges<br>➤ no parallel edges<br>➤ no self-loops<br>➤ Bounds are “big-Oh” | Edge List | Adjacency List           | Adjacency Matrix |
|---|-----------|--------------------------|------------------|
| Space   | $n + m$   | $n + m$                  | $n^2$            |
| incidentEdges( $v$ )  | $m$       | $\deg(v)$                | $n$              |
| areAdjacent ( $v, w$ )  | $m$       | $\min(\deg(v), \deg(w))$ | 1                |
| insertVertex( $o$ )   | 1         | 1                        | $n^2$            |
| insertEdge( $v, w, o$ )   | 1         | 1                        | 1                |
| removeVertex( $v$ )   | $m$       | $\deg(v)$                | $n^2$            |
| removeEdge( $e$ )   | 1         | 1                        | 1                |

# Exercise 2

Consider the following two adjacency matrices:

|       | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| 1     | 0 | 1 | 0 | 0 | 1 | 0 |
| 2     | 1 | 0 | 1 | 0 | 0 | 1 |
| 3     | 0 | 1 | 0 | 0 | 0 | 1 |
| 4     | 0 | 0 | 0 | 0 | 1 | 0 |
| 5     | 1 | 0 | 0 | 1 | 0 | 1 |
| (1) 6 | 0 | 1 | 1 | 0 | 1 | 0 |

|       | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|
| 1     | 0 | 1 | 0 | 0 | 0 | 0 |
| 2     | 1 | 0 | 0 | 0 | 0 | 0 |
| 3     | 0 | 1 | 0 | 0 | 0 | 1 |
| 4     | 1 | 0 | 0 | 1 | 1 | 0 |
| 5     | 1 | 0 | 0 | 1 | 0 | 1 |
| (2) 6 | 0 | 1 | 1 | 0 | 0 | 0 |

For each, answer the following questions:

- (a) Can this matrix be the adjacency matrix of an undirected graph? Could it represent a directed graph? Why or why not?
- (b) For both adjacency matrices draw the corresponding undirected graph if possible, otherwise draw the corresponding directed graph.
- (c) Give the adjacency list representation for both graphs.

# Exercise 3

Create graph from given adjacency matrix:

a)

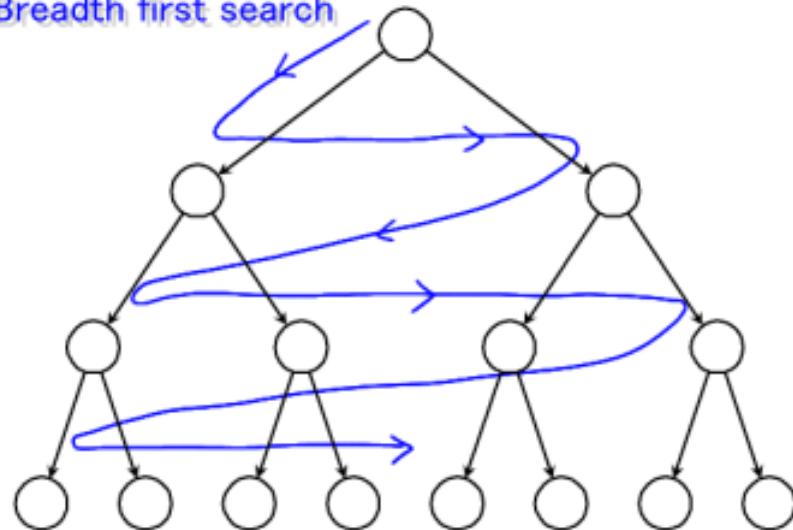
|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 |

b)

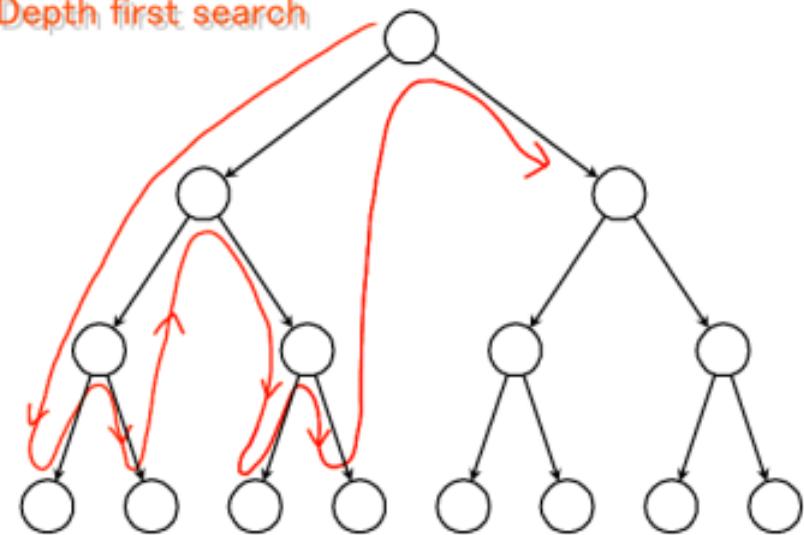
$$A = \begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

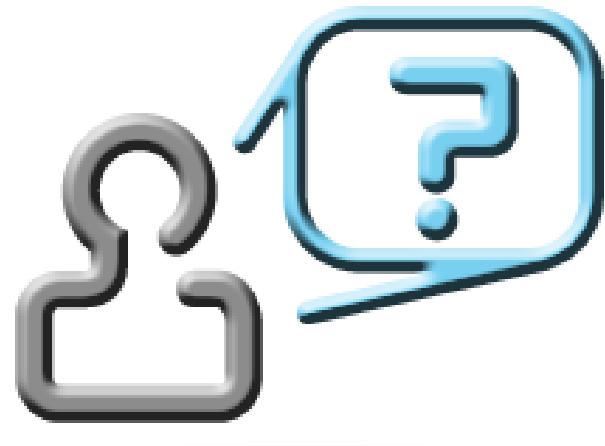
# Graph Traversals

Breadth first search



Depth first search





*See you in the next class to explore more on Graphs!*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : CC BY-NC-SA 4.0





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# **Data Structures and Algorithms Design**

## **DSECLZG519**

**Parthasarathy**

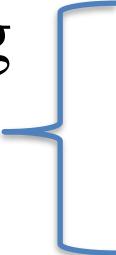




# Contact Session #7

# DSECLZG519 – Graph Traversals & Hashing

# Agenda for CS #7

- 
- 1) Recap and continuation of CS#6
    - BFS and DFS
  - 2) Dictionary ADT
  - 3) Hashing and Collision
    - What is collision ?
    - Types of collision handling
      - Separate Chaining
      - Open addressing
  - 4) Exercises
  - 5) Q&A
- 

# Breadth First Search (BFS)

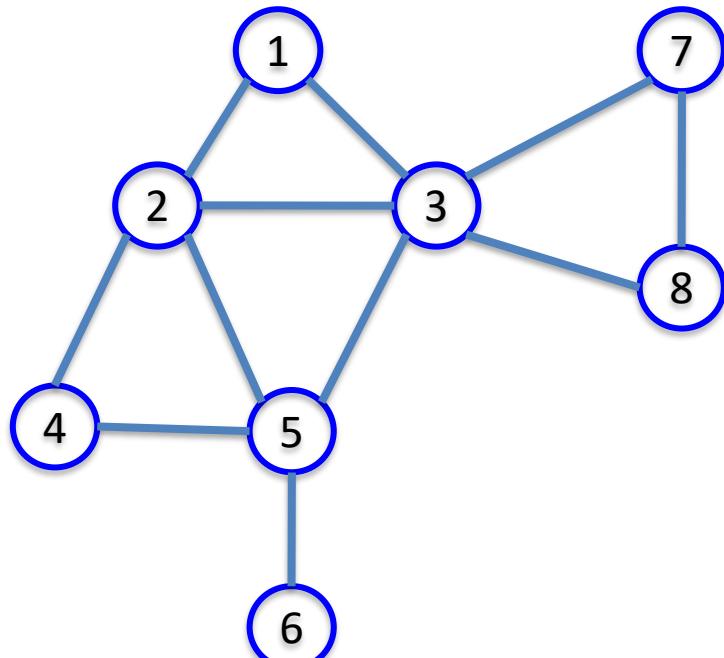
- Breadth-first search (BFS) is a general technique for traversing a graph.
- A BFS traversal of a graph G:
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with  $n$  vertices and  $m$  edges takes  $O(n + m)$  time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

# Breadth First Search (BFS)

---

- It is very common to use a *queue* to keep track of:
  - nodes to be visited next, or
  - nodes that we have already visited.
- Typically, use of a queue leads to a *breadth-first* visit order.
- Breadth-first visit order is “cautious” in the sense that it examines every path of length  $i$  before going on to paths of length  $i+1$ .
- Dotted lines depict *Cross edges* and solid lines depict the *discovery edges*.

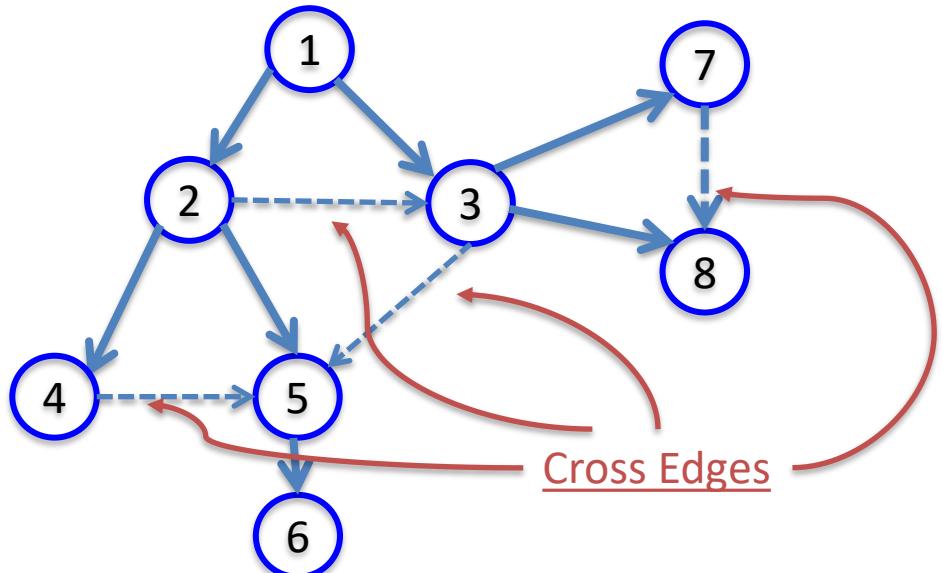
# BFS - Demo



Queue - FIFO



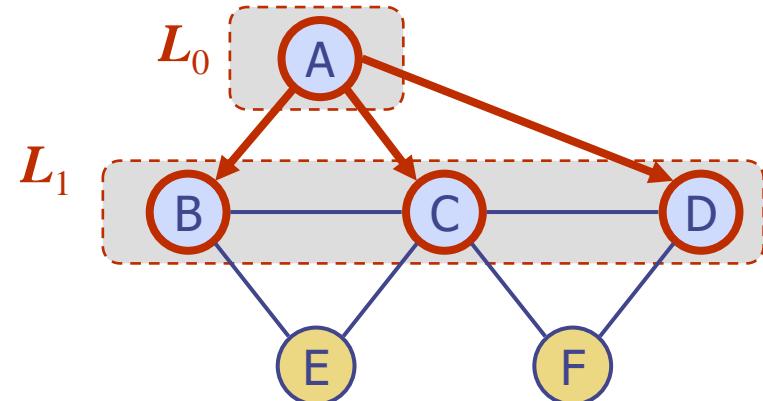
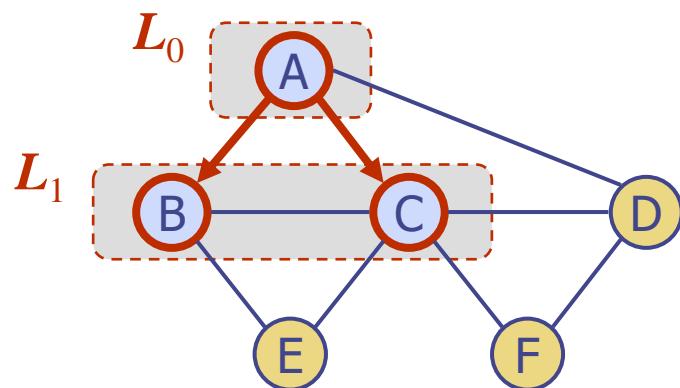
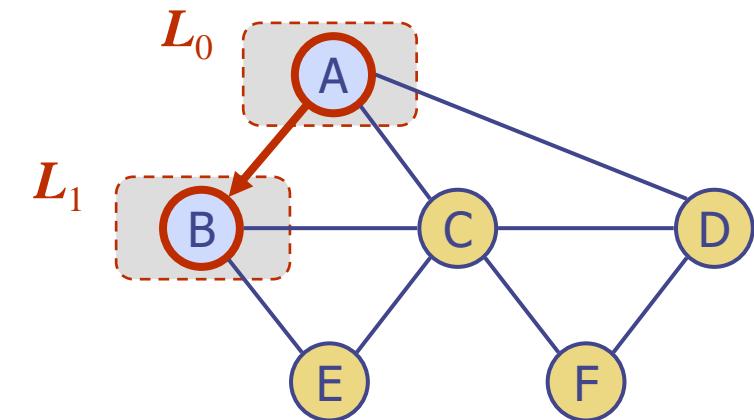
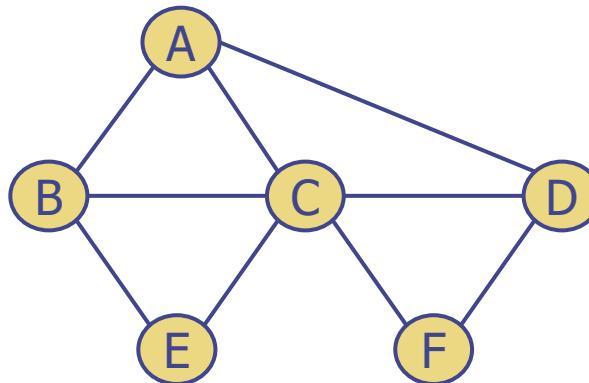
BFS Spanning Tree



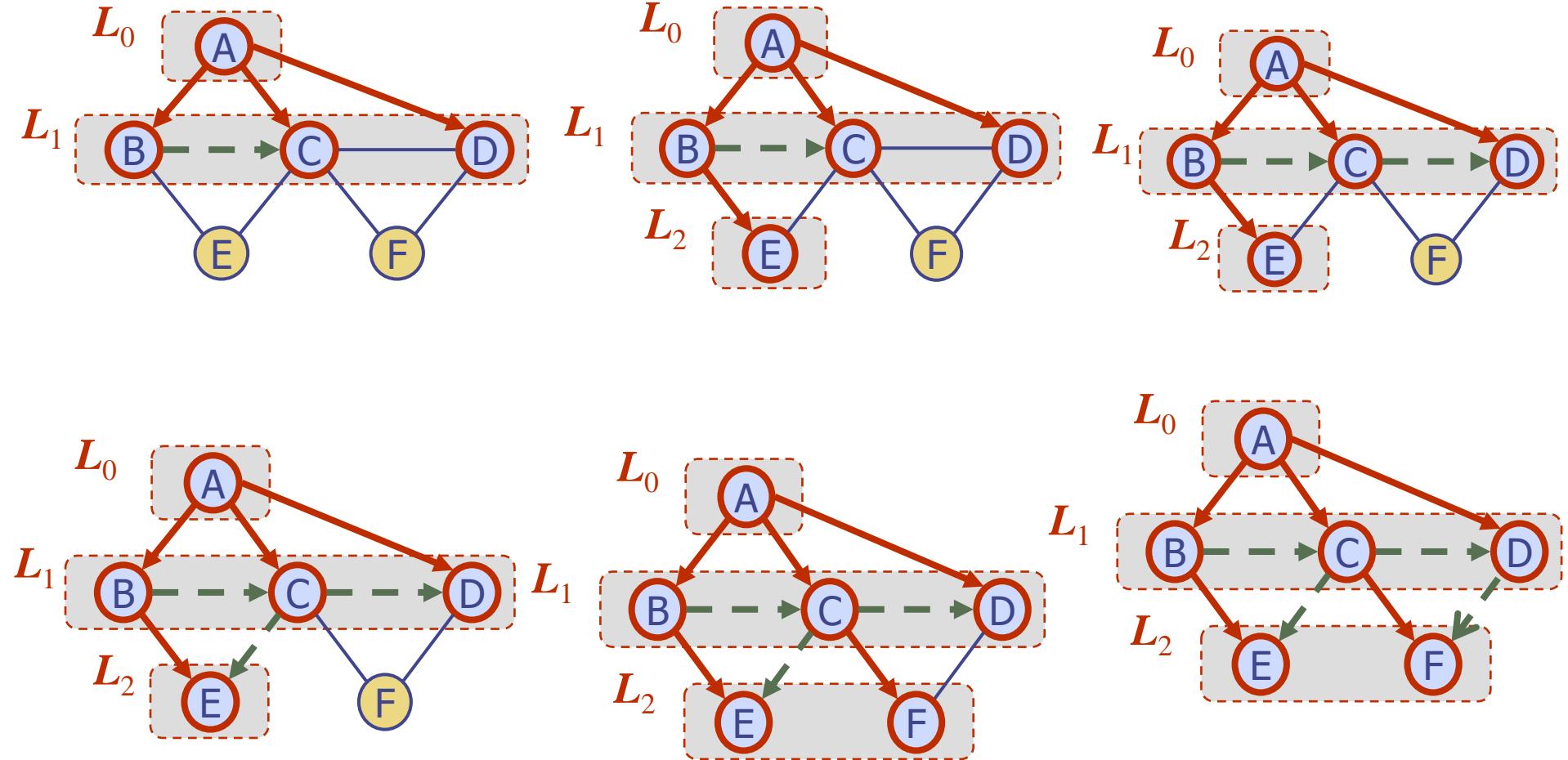
BFS: 1,2,3,4,5,7,8,6

# BFS – Example 2

- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- cross edge



# BFS – Example 2



# BFS- Pseudocode

```

BFS (G, s)                                //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
        v = Q.dequeue( )

        //processing all the neighbours of v
        for all neighbours w of v in Graph G
            if w is not visited
                Q.enqueue( w )           //Stores w in Q to further visit its neighbour
                mark w as visited.
  
```

The time complexity of BFS is **O(V+E)** where **V** is the number of nodes and **E** is the number of edges.

# BFS Applications

---

- We can specialize the BFS traversal of a graph  $G$  to solve the following problems in  $O(n + m)$  time
  - Compute the connected components of  $G$
  - Compute a spanning forest of  $G$
  - Find a simple cycle in  $G$ , or report that  $G$  is a forest
  - Given two vertices of  $G$ , find a path in  $G$  between them with the minimum number of edges, or report that no such path exists

*Good Design exercise! Try the above problems using BFS traversal*

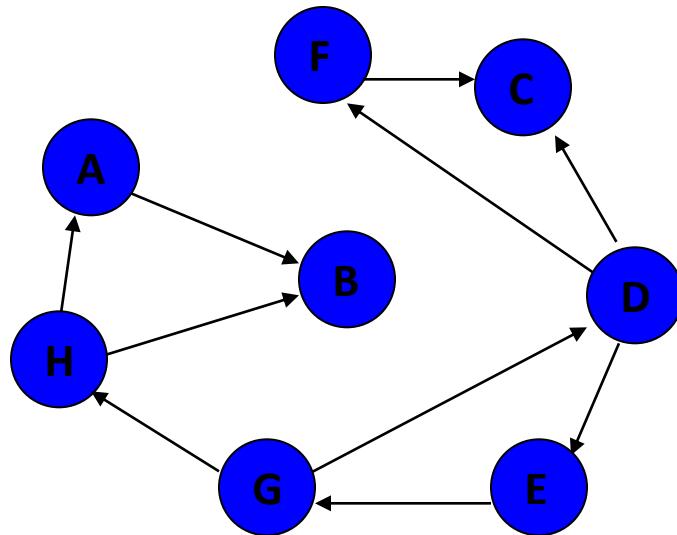
# Depth First Search (DFS)

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph  $G$ 
  - Visits all the vertices and edges of  $G$
  - Determines whether  $G$  is connected
  - Computes the connected components of  $G$
  - Computes a spanning forest of  $G$
- DFS on a graph with  $n$  vertices and  $m$  edges takes  $O(n + m)$  time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph

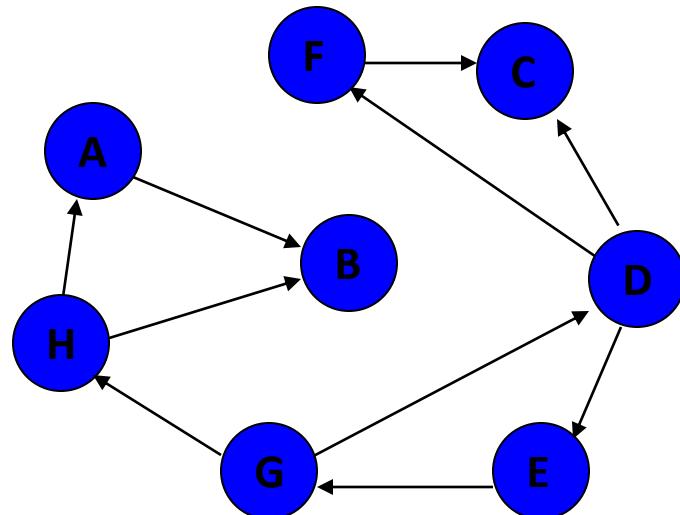
# DFS

- It is very common to use a *stack* to keep track of:
  - nodes to be visited next, or
  - nodes that we have already visited.
- Typically, use of a stack leads to a *depth-first* visit order.
- Depth-first visit order is “aggressive” in the sense that it examines complete paths.
- Solid lines depict the *discovery edges* and dotted lines depict the *Back edges*.

Consider this graph below and perform DFS with D as start node.

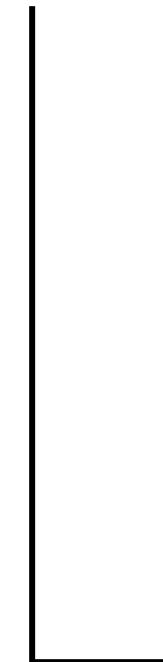


# Walk-Through



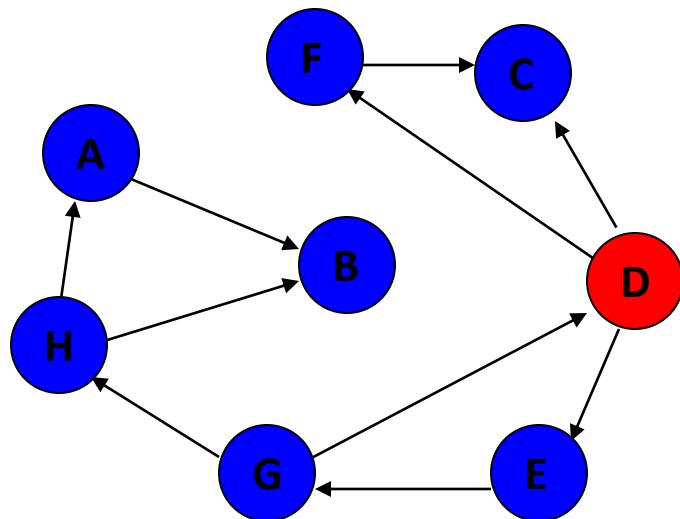
Visited Array

|   |  |
|---|--|
| A |  |
| B |  |
| C |  |
| D |  |
| E |  |
| F |  |
| G |  |
| H |  |



**Task: Conduct a depth-first search of the graph starting with node D**

# Walk-Through



The order nodes are visited:

D

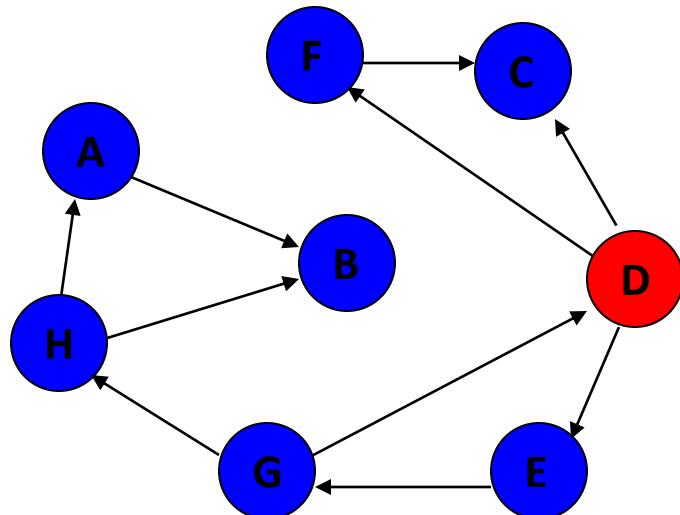
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C |   |
| D | ✓ |
| E |   |
| F |   |
| G |   |
| H |   |

|   |
|---|
| D |
|---|

**Visit D**

# Walk Through



The order nodes are visited:

D

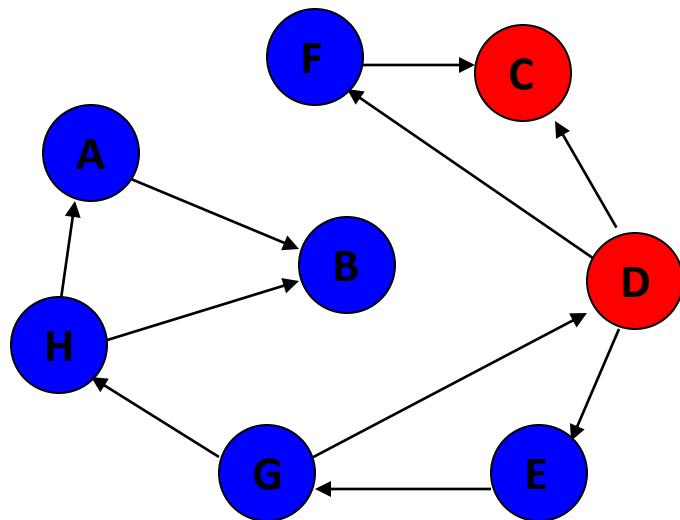
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C |   |
| D | ✓ |
| E |   |
| F |   |
| G |   |
| H |   |

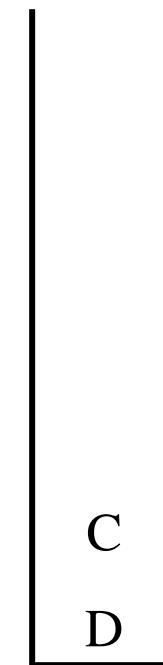
|   |
|---|
| D |
|---|

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)

# Walk Through



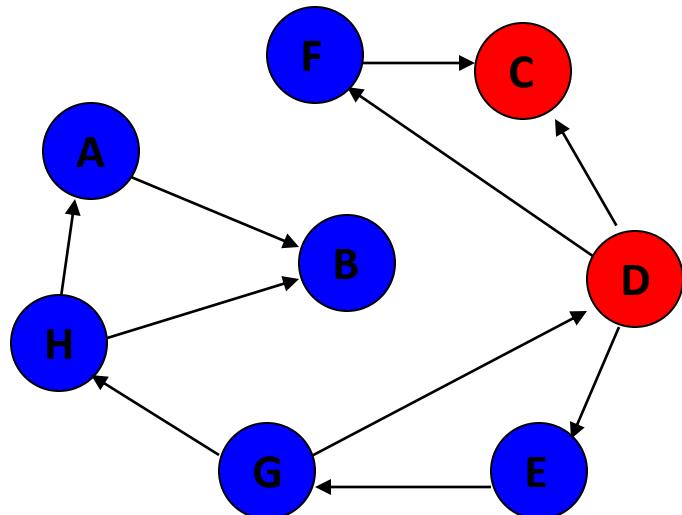
|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E |   |
| F |   |
| G |   |
| H |   |



Visit C

The order nodes are visited:  
D, C

# Walk-Through



The order nodes are visited:

D, C

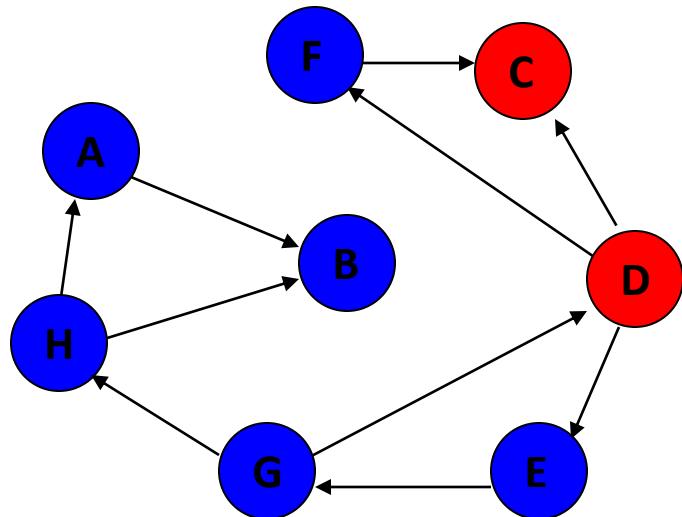
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E |   |
| F |   |
| G |   |
| H |   |

|   |
|---|
| C |
| D |

No nodes adjacent to C; cannot continue → *backtrack*, i.e., pop stack and restore previous state

# Walk-Through



The order nodes are visited:  
D, C

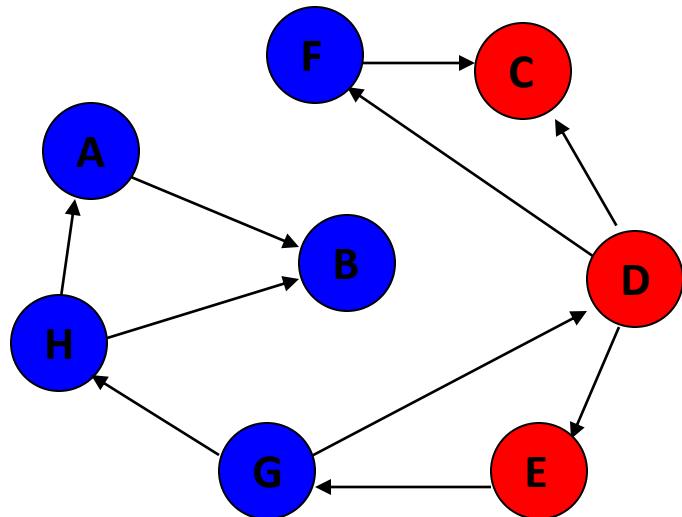
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E |   |
| F |   |
| G |   |
| H |   |

|   |
|---|
| D |
|   |

**Back to D – C has been visited,  
decide to visit E next**

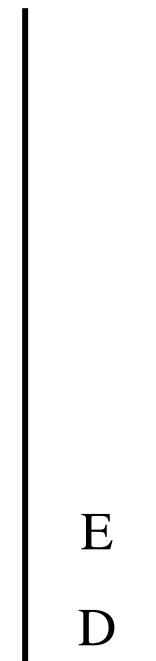
# Walk Through



The order nodes are visited:  
D, C, E

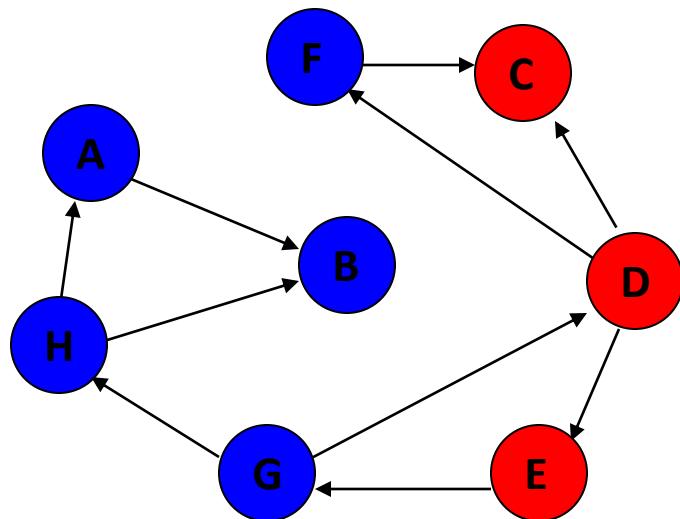
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G |   |
| H |   |



**Back to D – C has been visited,  
decide to visit E next**

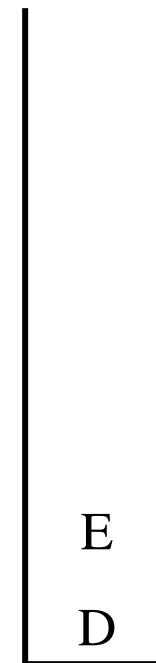
# Walk-Through



The order nodes are visited:  
D, C, E

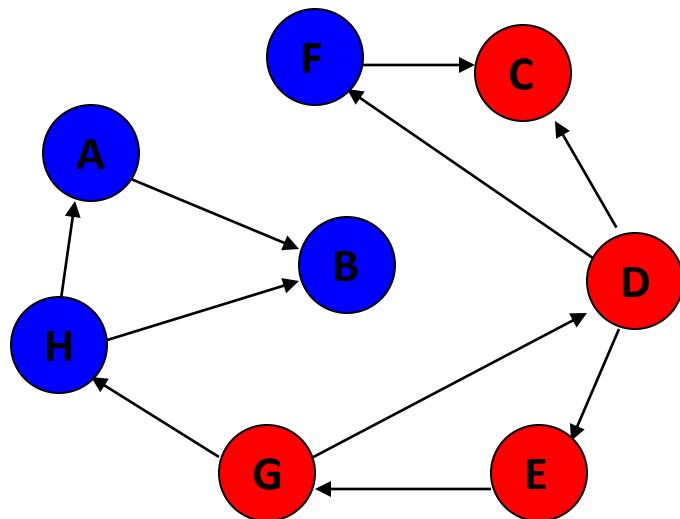
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G |   |
| H |   |



**Only G is adjacent to E**

# Walk-Through



The order nodes are visited:

D, C, E, G

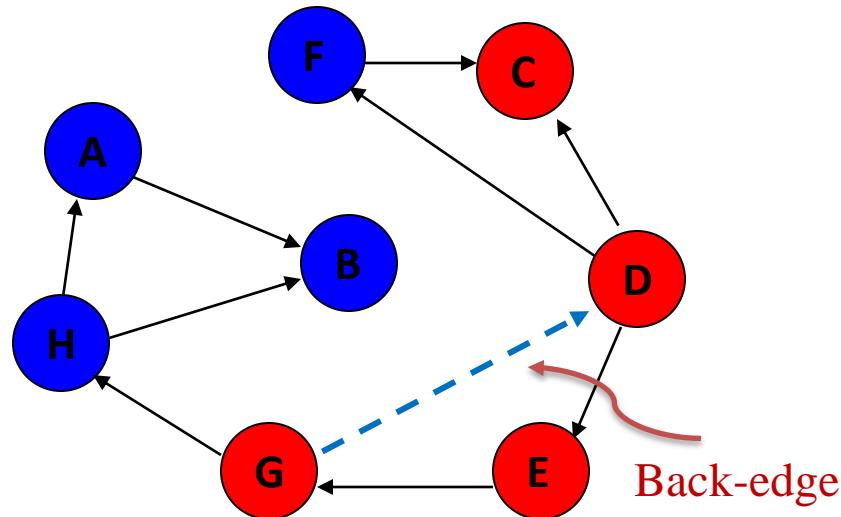
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H |   |

|   |
|---|
| G |
| E |
| D |

Visit G

# Walk-Through



Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H |   |

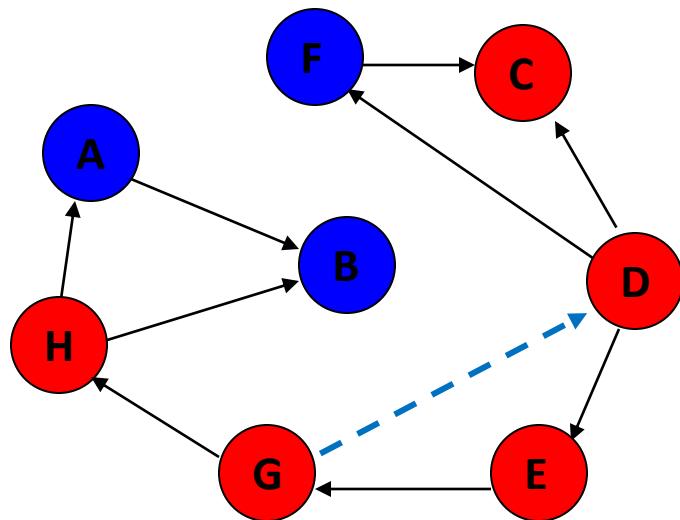
|   |
|---|
| G |
| E |
| D |

The order nodes are visited:

D, C, E, G

**Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.**

# Walk-Through



The order nodes are visited:

D, C, E, G, H

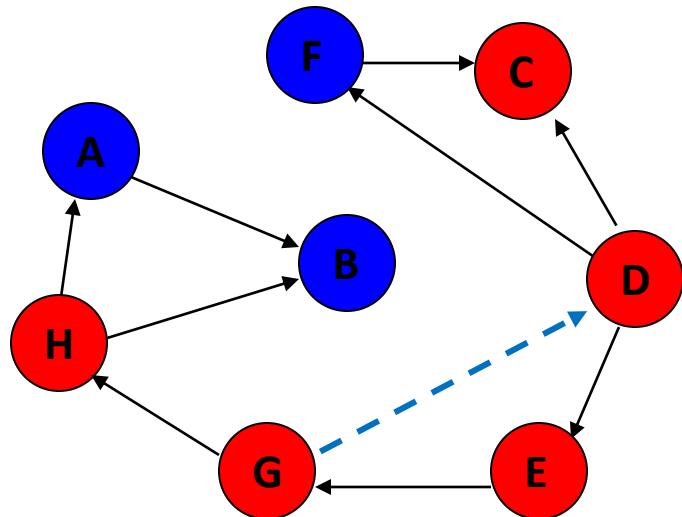
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |

|   |
|---|
| H |
| G |
| E |
| D |

Visit H

# Walk-Through



The order nodes are visited:

D, C, E, G, H

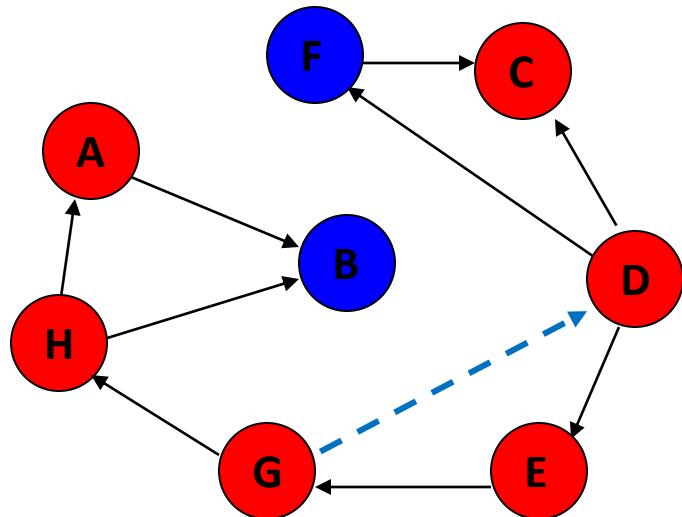
Visited Array

|   |   |
|---|---|
| A |   |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |

|   |
|---|
| H |
| G |
| E |
| D |

**Nodes A and B are adjacent to F.  
Decide to visit A next.**

# Walk-Through



The order nodes are visited:

D, C, E, G, H, A

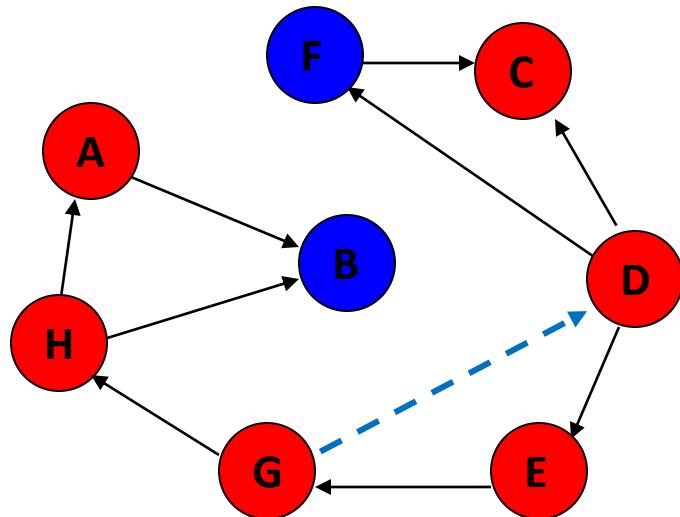
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |

|   |
|---|
| A |
| H |
| G |
| E |
| D |

**Visit A**

# Walk-Through



The order nodes are visited:  
D, C, E, G, H, A

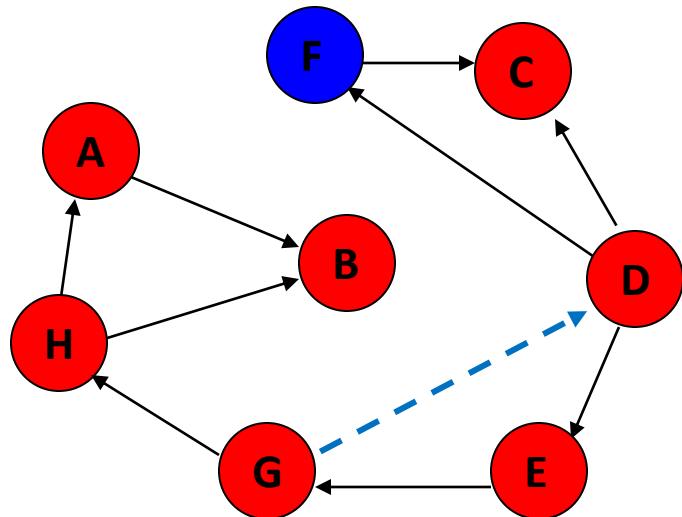
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B |   |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |

|   |
|---|
| A |
| H |
| G |
| E |
| D |

**Only Node B is adjacent to A.  
Decide to visit B next.**

# Walk Through

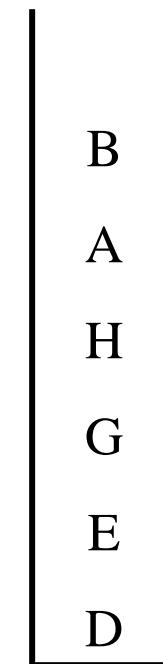


The order nodes are visited:

D, C, E, G, H, A, B

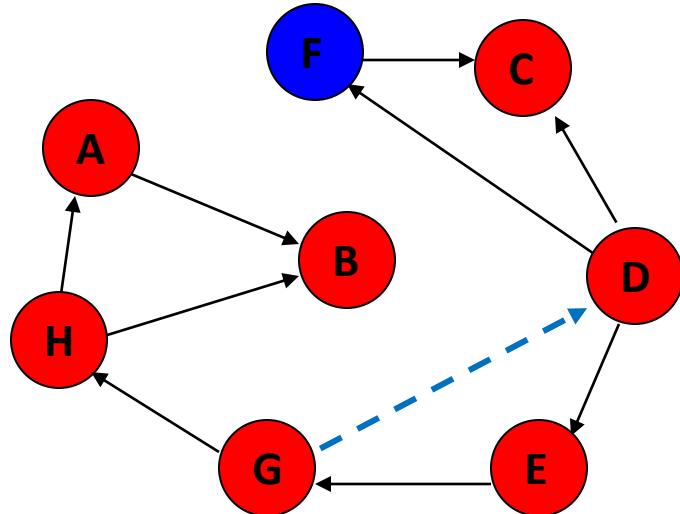
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |



**Visit B**

# Walk Through



The order nodes are visited:  
D, C, E, G, H, A, B

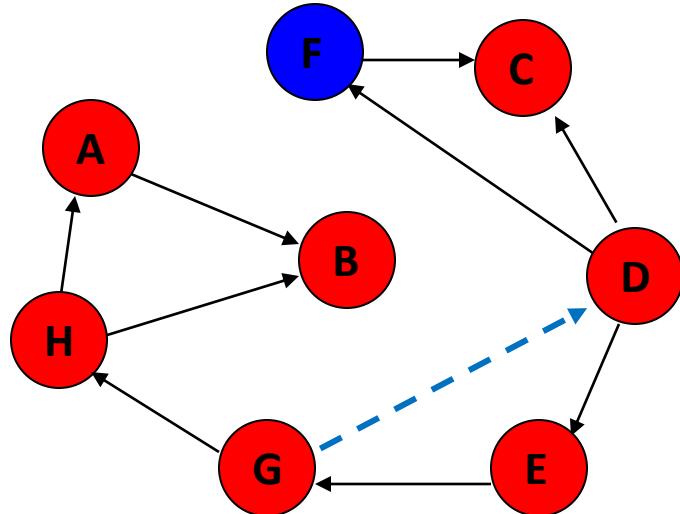
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |

|   |
|---|
| A |
| H |
| G |
| E |
| D |

**No unvisited nodes adjacent to  
B. Backtrack (pop the stack).**

# Walk Through



The order nodes are visited:

D, C, E, G, H, A, B

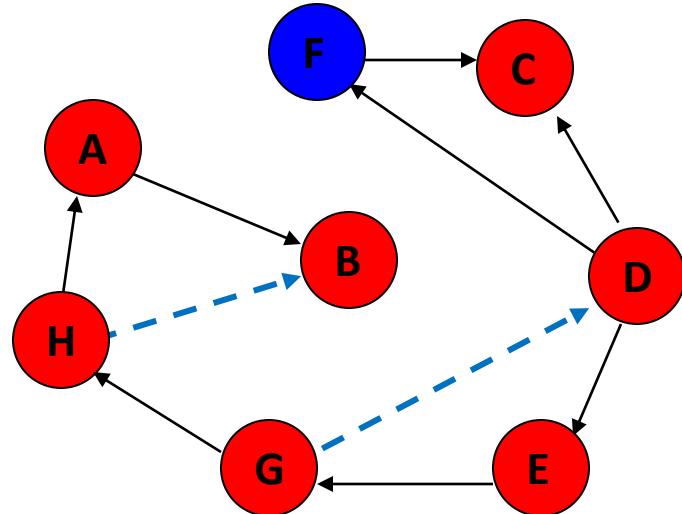
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |

|   |
|---|
| H |
| G |
| E |
| D |

**No unvisited nodes adjacent to A. Backtrack (pop the stack).**

# Walk Through

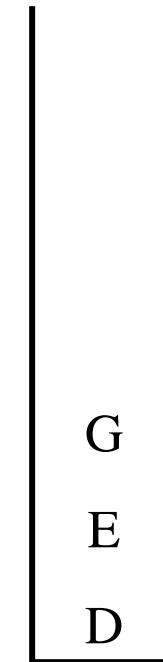


The order nodes are visited:

D, C, E, G, H, A, B

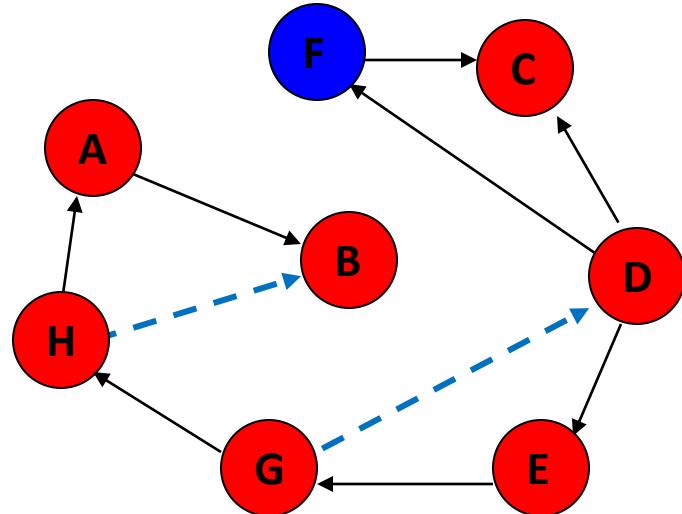
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |



**No unvisited nodes adjacent to H. Backtrack (pop the stack).**

# Walk-Through

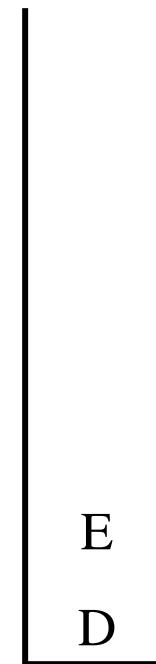


The order nodes are visited:

D, C, E, G, H, A, B

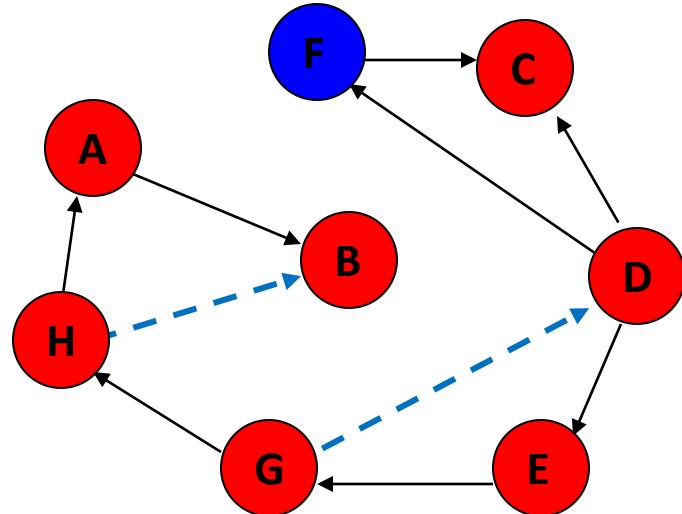
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |



**No unvisited nodes adjacent to G. Backtrack (pop the stack).**

# Walk-Through



The order nodes are visited:  
D, C, E, G, H, A, B

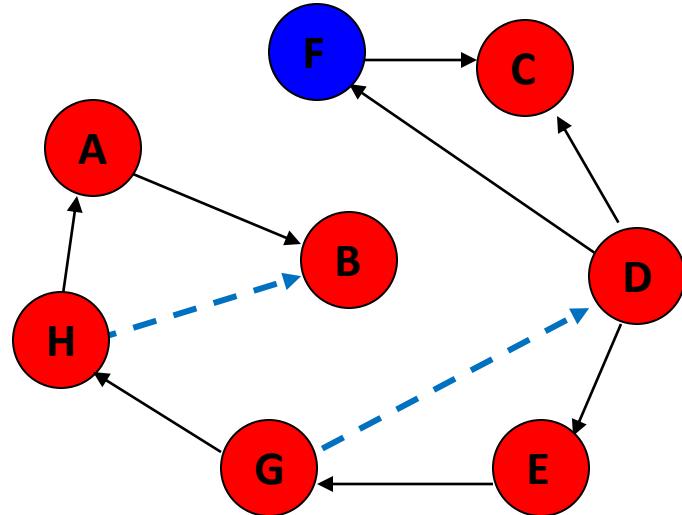
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |



**No unvisited nodes adjacent to E. Backtrack (pop the stack).**

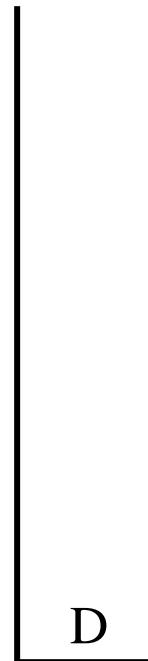
# Walk-Through



The order nodes are visited:  
D, C, E, G, H, A, B

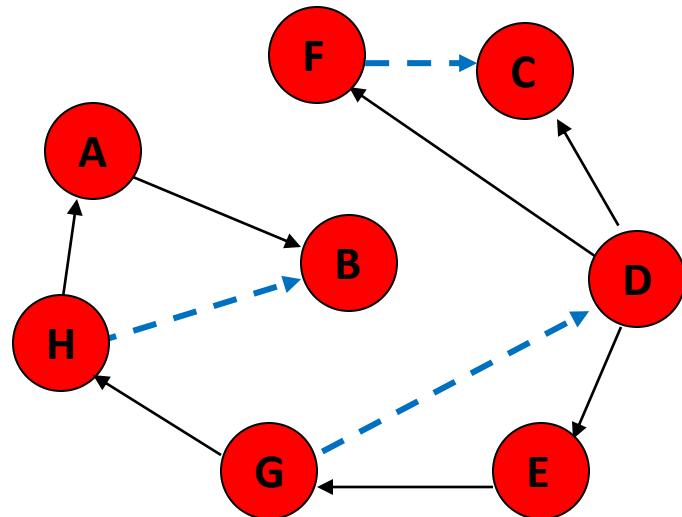
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F |   |
| G | ✓ |
| H | ✓ |



**F is unvisited and is adjacent to  
D. Decide to visit F next.**

# Walk-Through

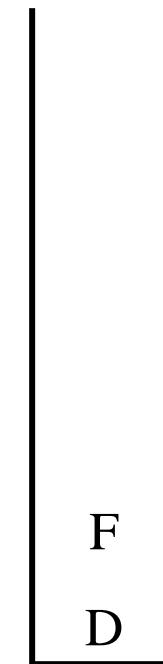


The order nodes are visited:

D, C, E, G, H, A, B, F

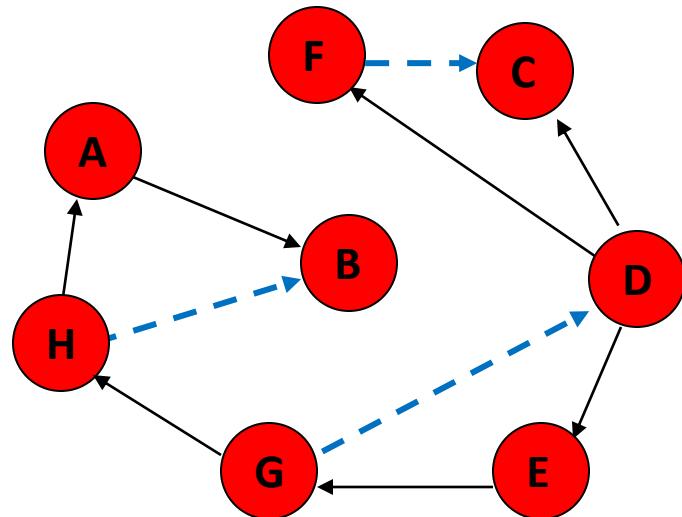
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F | ✓ |
| G | ✓ |
| H | ✓ |



Visit F

# Walk-Through



The order nodes are visited:

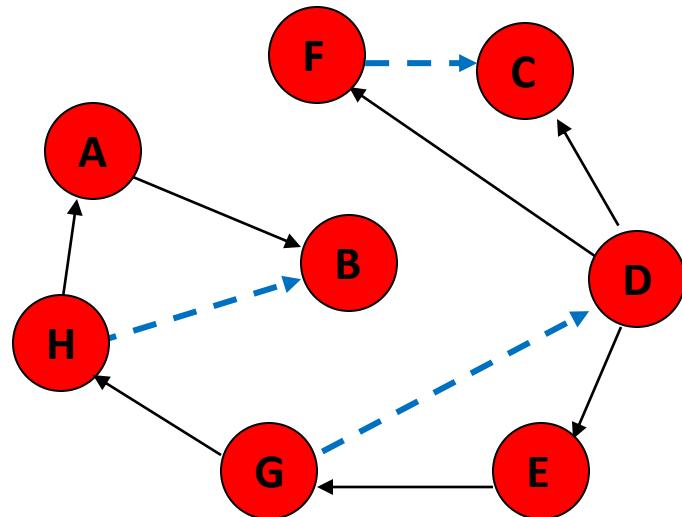
D, C, E, G, H, A, B, F

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F | ✓ |
| G | ✓ |
| H | ✓ |

|   |
|---|
| D |
|---|

**No unvisited nodes adjacent to F. Backtrack.**

# Walk-Through

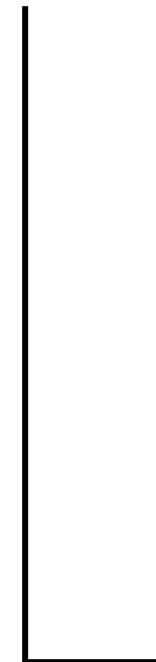


The order nodes are visited:

D, C, E, G, H, A, B, F

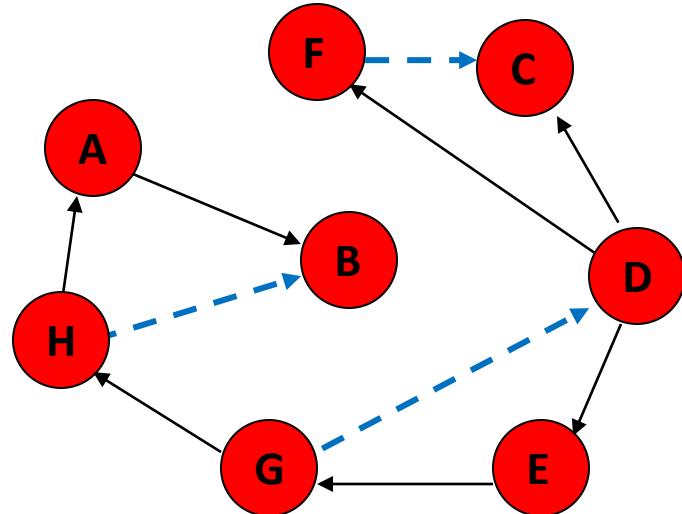
Visited Array

|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F | ✓ |
| G | ✓ |
| H | ✓ |



**No unvisited nodes adjacent to  
D. Backtrack.**

# Walk-Through



The order nodes are visited:  
**D, C, E, G, H, A, B, F**

Visited Array

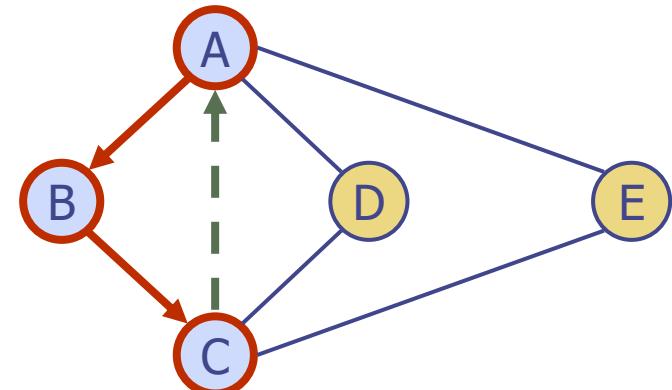
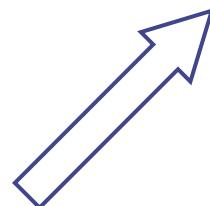
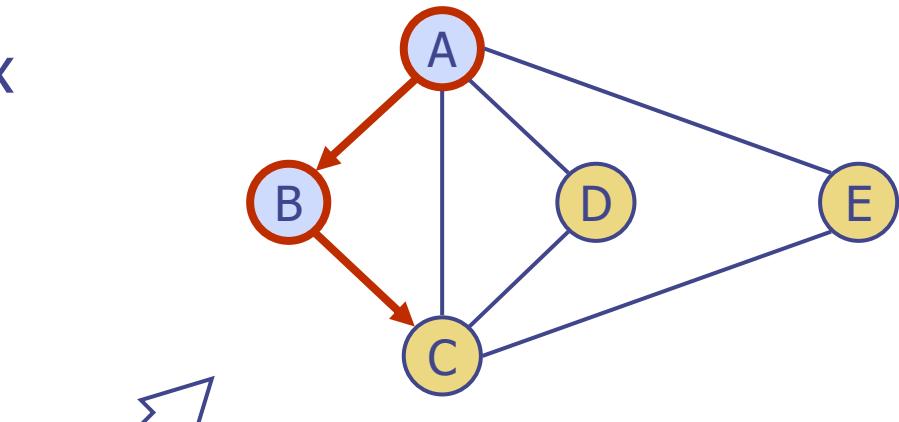
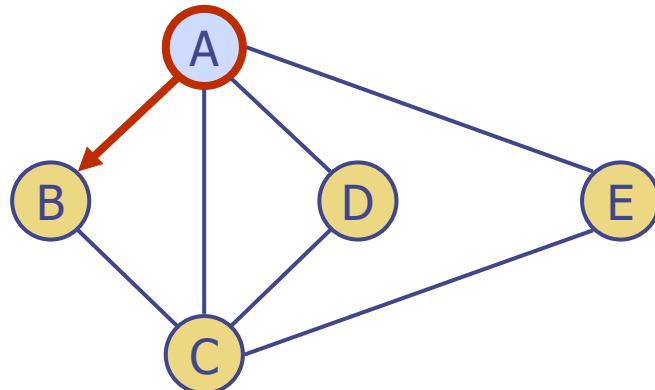
|   |   |
|---|---|
| A | ✓ |
| B | ✓ |
| C | ✓ |
| D | ✓ |
| E | ✓ |
| F | ✓ |
| G | ✓ |
| H | ✓ |



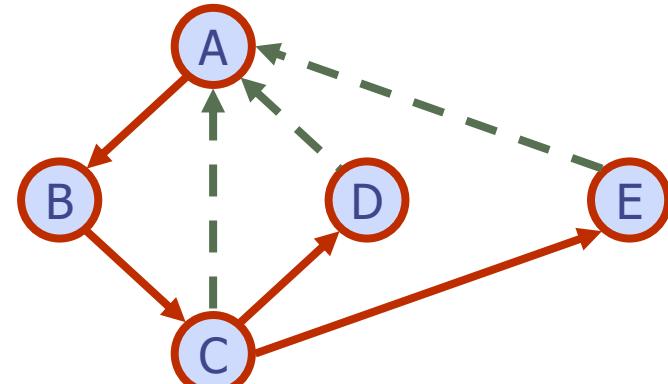
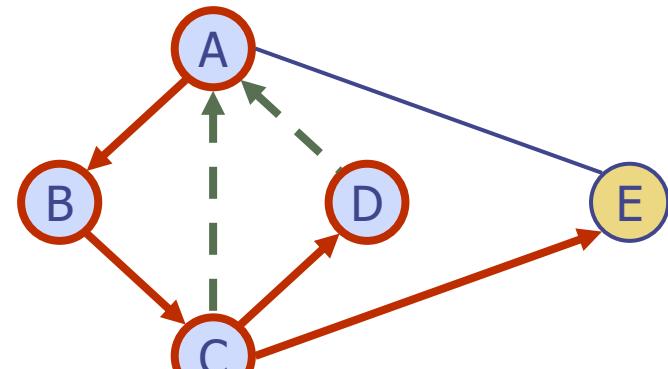
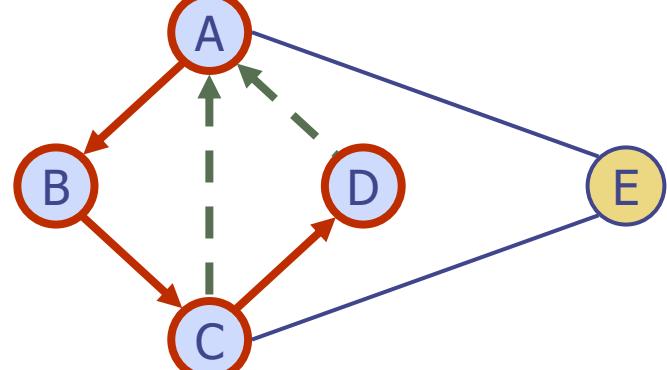
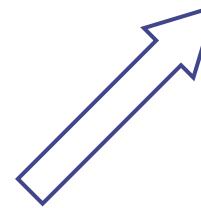
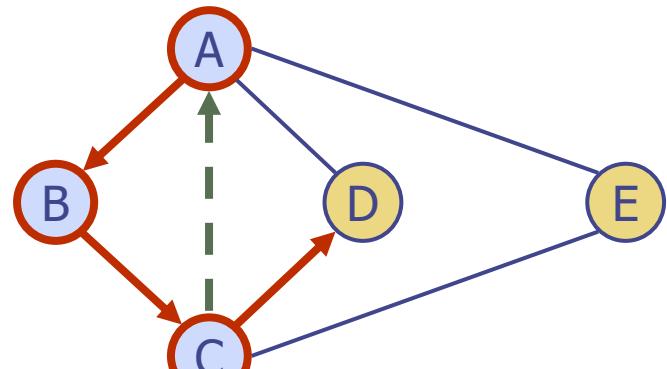
**Stack is empty. Depth-first traversal is done.**

# DFS Example 2

- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- - - → back edge



# DFS Example 2



# DFS Pseudocode

```

DFS-iterative (G, s):                                //Where G is graph and s is source vertex
  let S be stack
  S.push( s )           //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
    //Pop a vertex from stack to visit next
    v = S.top()
    S.pop()
    //Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
      if w is not visited :
        S.push( w )
        mark w as visited

DFS-recursive(G, s):
  mark s as visited
  for all neighbours w of s in Graph G:
    if w is not visited:
      DFS-recursive(G, w)
  
```

The time complexity of DFS is **O(V+E)** where **V** is the number of nodes and **E** is the number of edges.

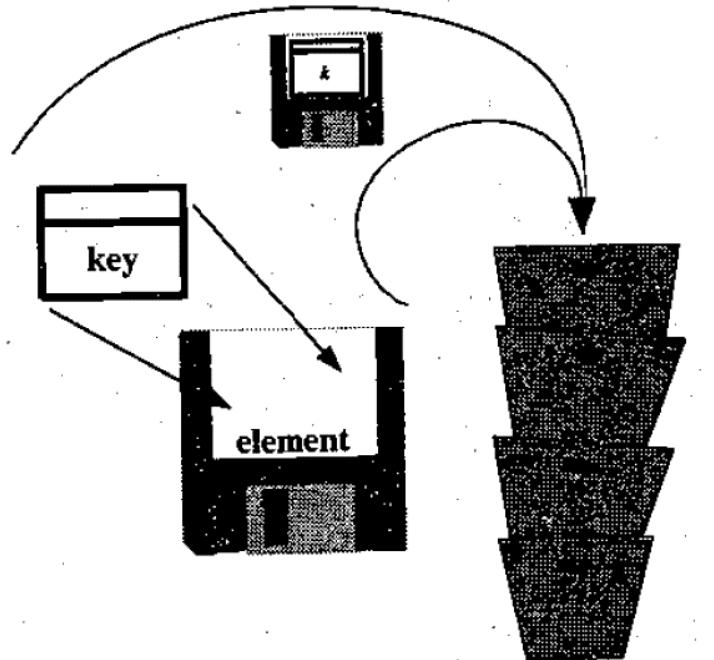
# DFS Applications

---

- Let  $G$  be a graph with  $n$  vertices and  $m$  edges represented with the adjacency list structure. A DFS traversal of  $G$  can be performed in  $O(n + m)$  time. Also, there exist  $O(n + m)$  -time algorithms based on DFS for the following problems:
  - Testing whether  $G$  is connected
  - Computing a spanning forest of  $G$
  - Computing the connected components of  $G$
  - Computing a path between two vertices of  $G$  , or reporting that no such path exists [ *Ex: Good algorithm design exercise*]
  - Computing a cycle in  $G$  , or reporting that  $G$  has no cycles. [ *Ex: Good algorithm design exercise.*]

# Dictionary ADT

- A dictionary stores key-element pairs ( $k$ ,  $e$ ), which we call items, where  $k$  is the key and  $e$  is the element.
- For example, in a dictionary storing student records (such as the student's name, address, and course grades), the key might be the student's ID number.
- The main operations of a dictionary are searching, inserting, and deleting items
- Store elements so that they can be quickly located using keys.
- Typically, useful additional information in addition to the element.
- Sets and Hash Tables are common dictionaries.
- Examples:
  - Bank accounts with SSN as key
  - Student records with RID as key



# Dictionary ADT methods

---

- **-findElement(k):** if the dictionary has an item with key k, returns its element, else, returns the special element NO\_SUCH\_KEY
- **-insertItem(k, e):** Insert an item with element e and key k into D
- **-removeElement(k):** if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO\_SUCH\_KEY
- **-size(), isEmpty()**
- **-keys(), elements()-Iterators**

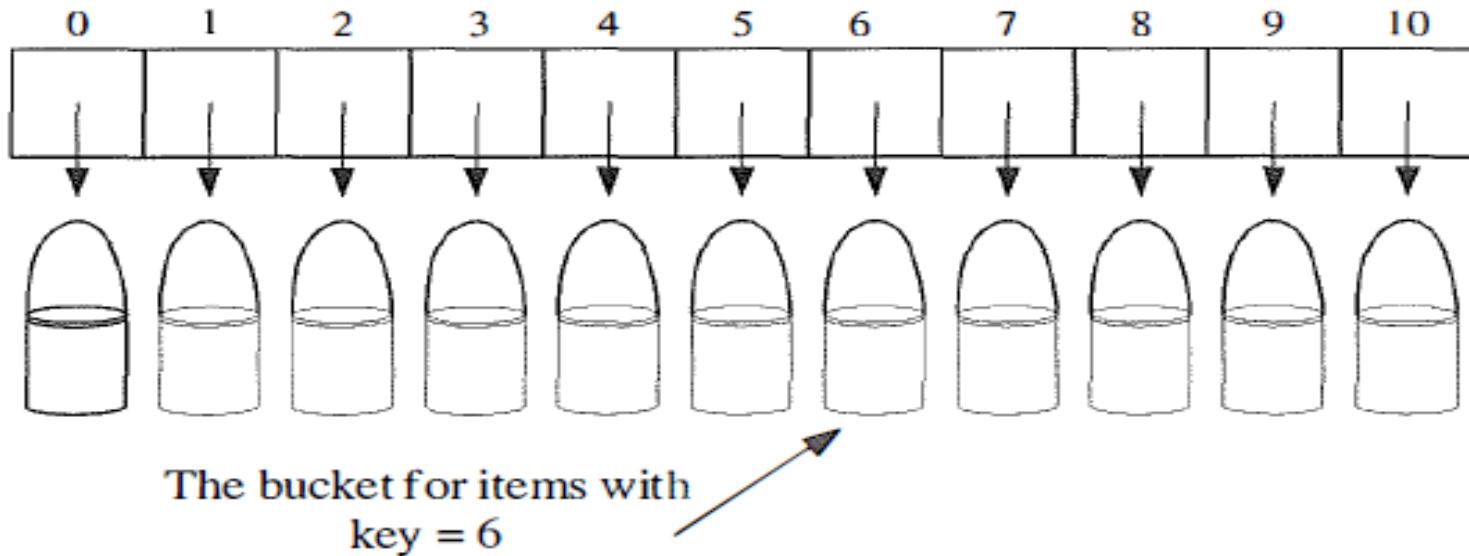
# Hash Table

---

- A hash table is a data structure used to store key-element ( $k, e$ ) pairs
- The key and element together is called an item
- Keys are used to insert, search and remove elements
- Applications :
  - Address book
  - Mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)
- A hash table consists of two major components
  - Hash function
  - (Bucket) Array (called table)

# Bucket Array

- An (bucket) array  $A$  of size  $N$ 
  - Each cell is thought of as a bucket
  - $N$  defines the capacity of the array
  - If the keys are integers well distributed in the range  $[0, N - 1]$ 
    - An element  $e$  with key  $k$  is inserted into the bucket  $A [k]$



# Array vs Hash Table

Array

| Value      |
|------------|
| New York   |
| Boston     |
| Mexico     |
| Kansas     |
| Detroit    |
| California |

Hash Table

| Key | Value      |
|-----|------------|
| 1   | New York   |
| 2   | Boston     |
| 3   | Mexico     |
| 4   | Kansas     |
| 5   | Detroit    |
| 6   | California |

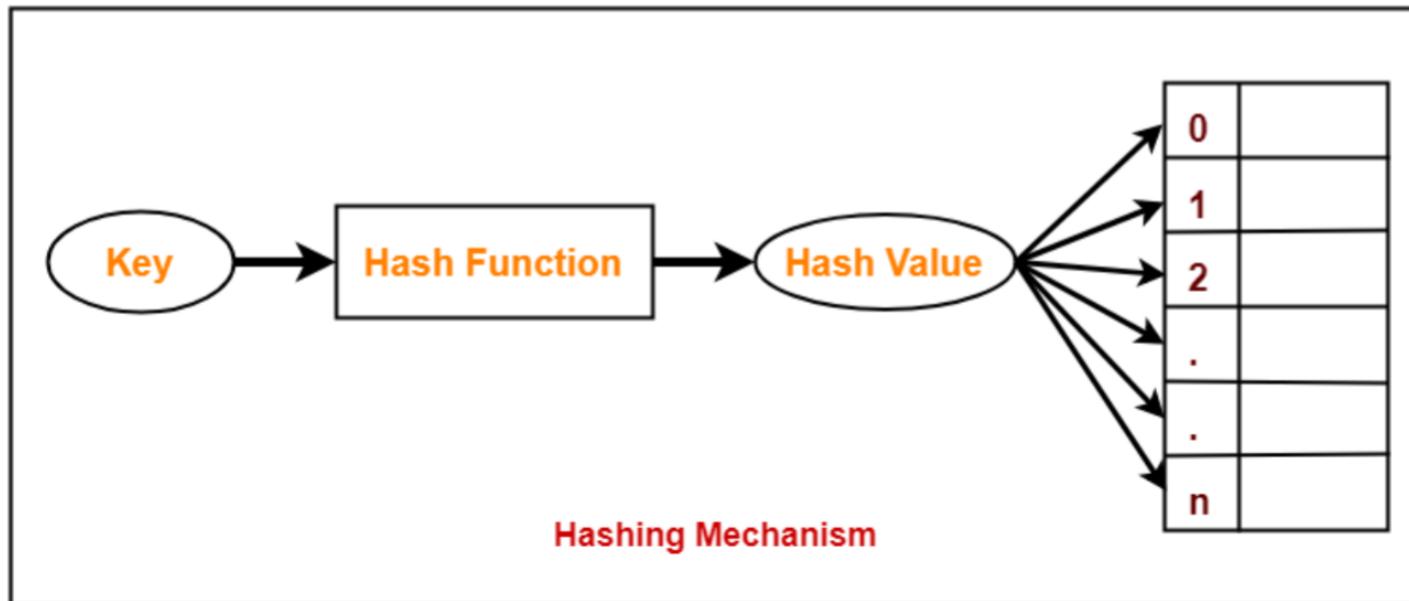
# Hash Function

- The keys are not always in the range  $[0, N - 1]$ .
- Hash function helps to map each key  $k$  to an integer in the range  $[0, N - 1]$ .
- Example:  $h(x) = x \bmod N$  is a hash function and the integer  $h(x)$  is called the hash value of key  $x$ .
- Hash function, maps each key  $k$  in our dictionary to an integer in the range  $[0, N - 1]$ , where  $N$  is the capacity of the bucket array for this table.
- Use the hash function value,  $h(k)$ , as an index to bucket array,  $A$ , instead of the key  $k$  (which is most likely inappropriate for use as a bucket array index). That is, store the item( $k, e$ ) in the bucket  $A[h(k)]$ .
- Requirements of good hash function
  - Minimize collisions as much as possible
  - Fast and easy to compute

# Hash Function

## Division:

- $h(k) = |k| \bmod N$
- $N$  is the size of the hash table
- $N$  is usually chosen to be a prime that helps to “spread out” the hashed values



# Hash Function

- 
- Consider the following keys: 200, 205, 210, 300, 305, 310, 400, 405, 410

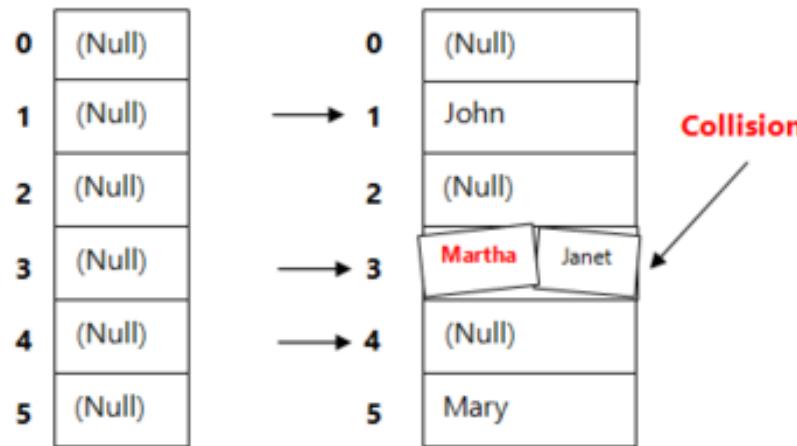
Apply the hash function  $h(k) = |k| \bmod N$  when  $N=100$   
and  $N=101$  and compute the hashed values

- $N=100$ 
  - hashed values are 0, 5, 10, 0, 5, 10, 0, 5, 10
- $N=101$ 
  - hashed values are 99, 3, 8, 98, 2, 7, 97, 1, 6

# Collision

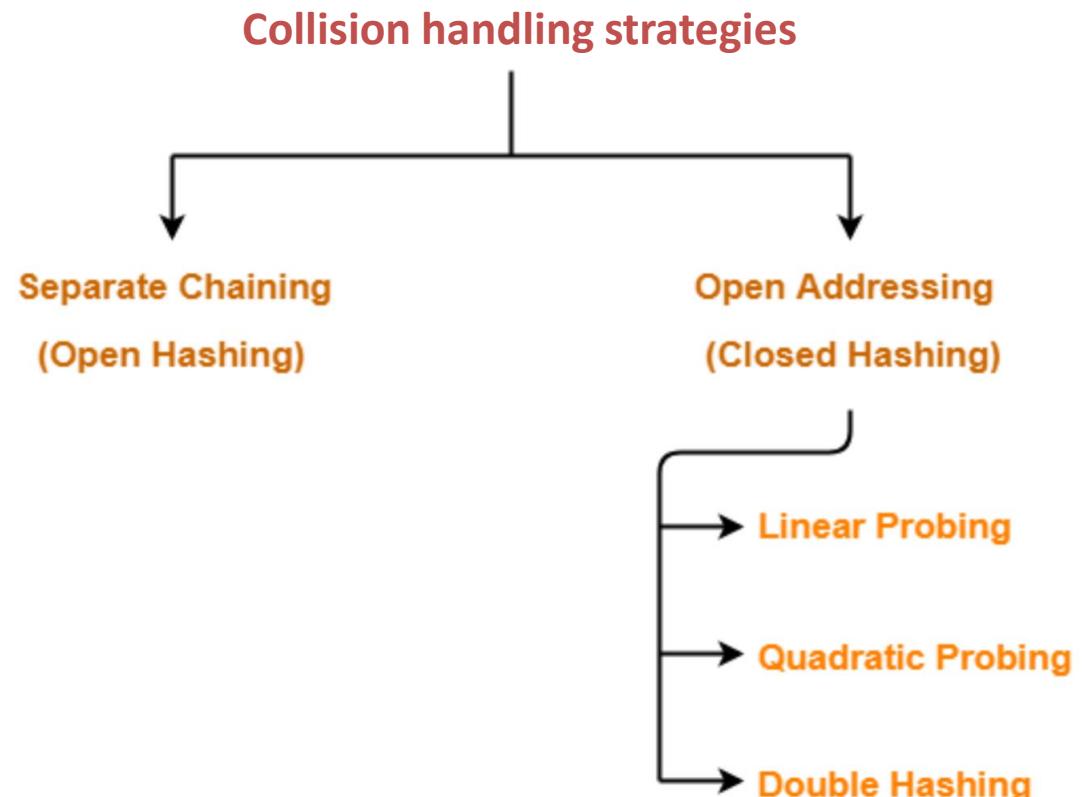
- A bucket can store only one item 😊
- Collision occurs when
  - If two different elements have same keys
  - When two distinct keys,  $k_1$  and  $k_2$ , are mapped to the same bucket in the hash table such that  $h(k_1)$  is equal to  $h(k_2)$

Figure 1: Hash Table: Empty / Collision Occurrence



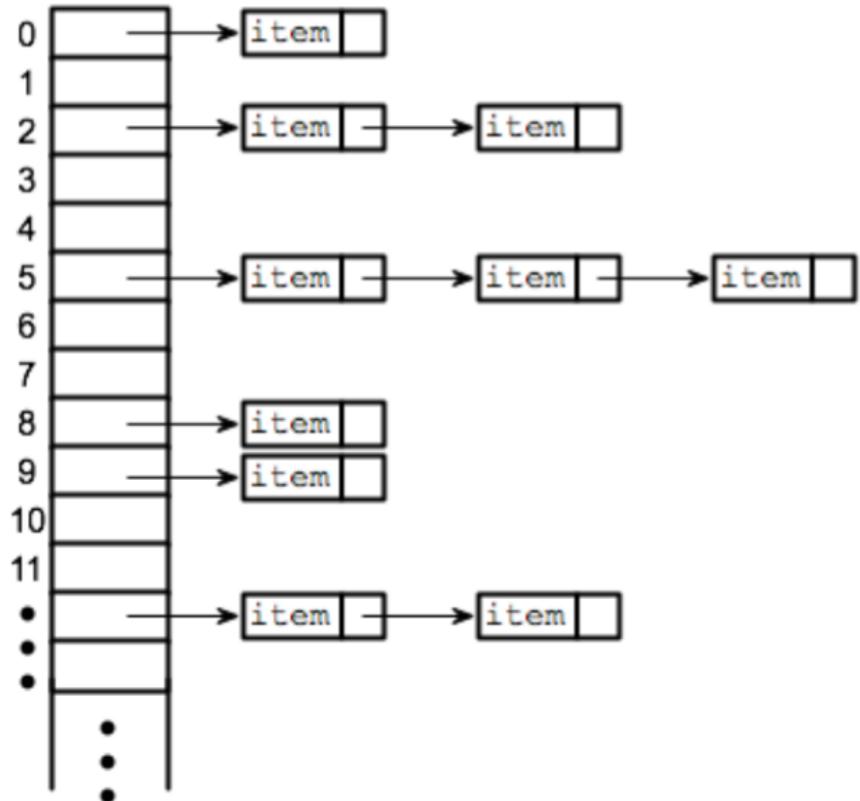
# Collision handling strategies

- Separate chaining
- Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing



# Separate chaining

- Colliding item is placed in a linked list
- Each cell in the array
  - Stores a pointer that points to a linked list of elements that are mapped to that cell
  - Otherwise stores NULL
- Requires additional memory apart from the table.



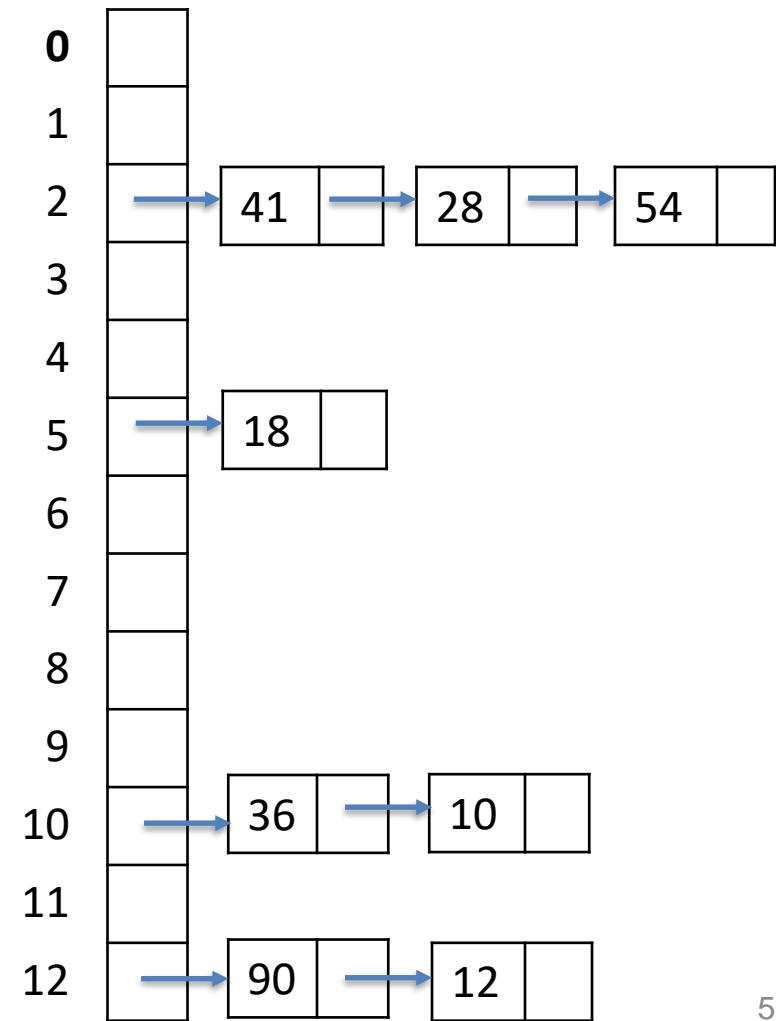
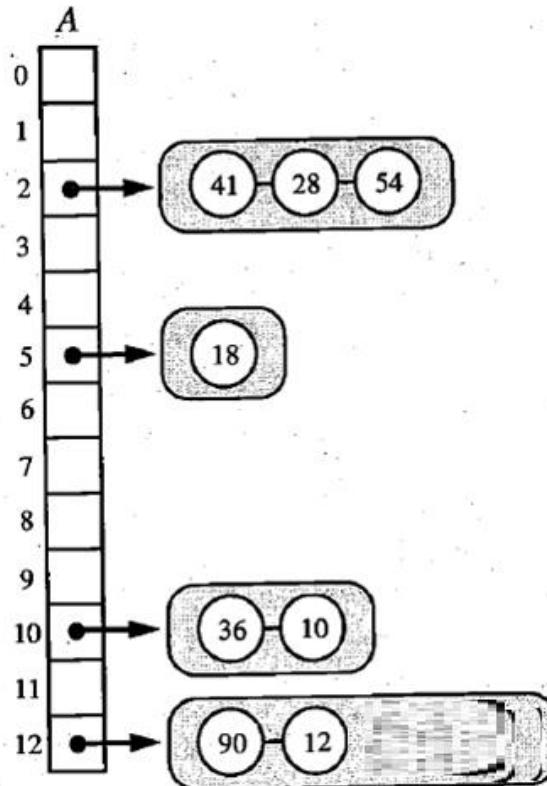
# Separate chaining

---

- Insert the elements using separate chaining in this order into an initially empty hash table using the hash function  $h(k) = k \bmod 13$ . Show your work step by step.
- Keys: 90, 41, 36, 18, 28, 12, 54, 10
- Hash values = 12, 2, 10, 5, 2, 12, 2, 10

# Separate chaining

- Keys: 90, 41, 36, 18, 28, 12, 54, 10
- Hash values = 12, 2, 10, 5, 2, 12, 2, 10



# Separate Chaining Insert Algorithm



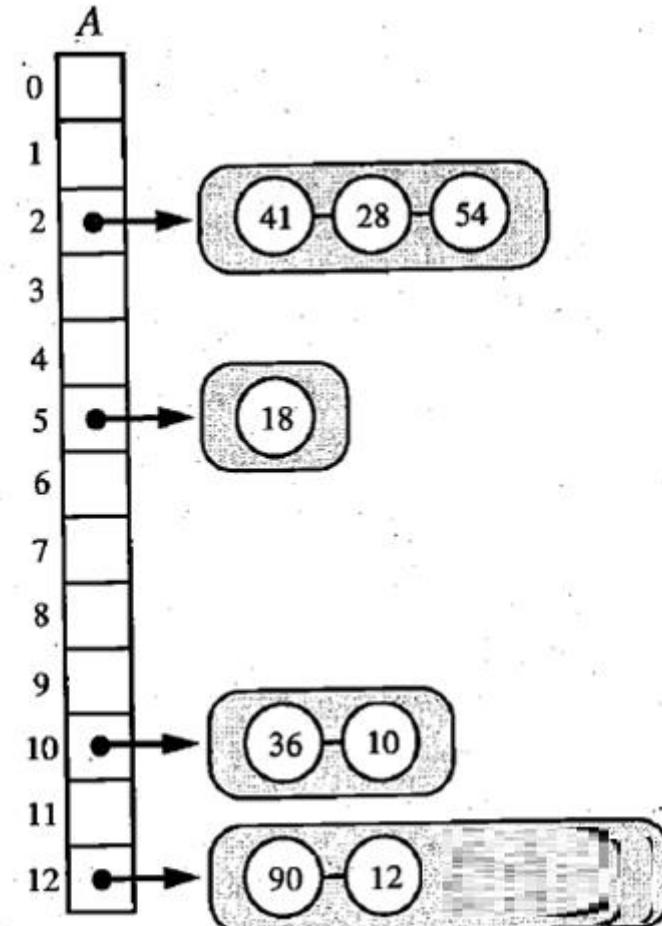
1. Declare an array of a linked list with the hash table size.
2. Initialize an array of a linked list to NULL.
3. Find hash key.
4. If  $\text{chain}[\text{key}] == \text{NULL}$   
    Make  $\text{chain}[\text{key}]$  points to the key node.
5. Otherwise(collision),  
    Insert the key node at the end of the  $\text{chain}[\text{key}]$ .

# Separate chaining

- Search (k)
  - Compute  $i = h(k)$
  - Go to cell  $A[i]$
  - if  $A[i]$  is not null traverse the list and search for the key
    - If the key exists return index ( $i$ ), KEY PRESENT
  - Else return NO SUCH KEY

# Separate chaining

- Search(10)
- ✓  $i=h(10) = 10 \bmod 13 = 10$
- ✓ *Goto  $A[i]=A[10]$  & check if 10 exists.*
- Search(20)
- ✓  $i=h(20)=20 \bmod 13 = 7$
- ✓ *Goto  $A[i]=A[7]$  & check if 20 exists.*
- Search(15)
- ✓  $i=h(15) = 15 \bmod 13 = 2$
- ✓ *Goto  $A[i] = A[2]$  & check if 15 exists*



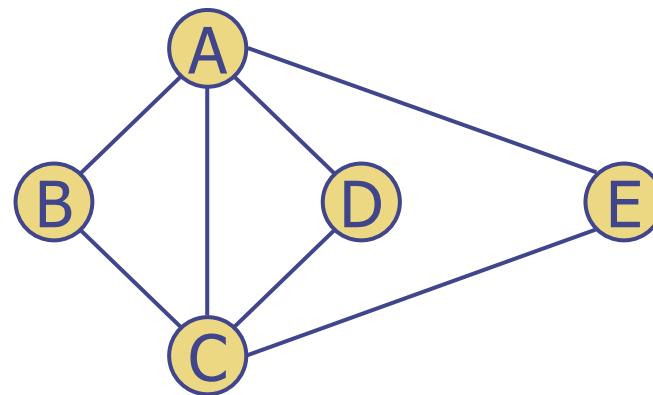
# Open addressing

- Colliding item is placed in a different cell of the array

| <b>Separate Chaining</b>   | <b>Open Addressing</b>   |
|--|--|
| Keys are stored inside the hash table as well as outside the hash table.                                       | All the keys are stored only inside the hash table.<br><br>No key is present outside the hash table. |
| The number of keys to be stored in the hash table can even exceed the size of the hash table.                  | The number of keys to be stored in the hash table can never exceed the size of the hash table.       |
| Deletion is easier.  | Deletion is difficult.   |
| Extra space is required for the pointers to store the keys outside the hash table.                             | No extra space is required.  |
| Cache performance is poor.<br><br>This is because of linked lists which store the keys outside the hash table. | Cache performance is better.<br><br>This is because here no linked lists are used.                   |
| Some buckets of the hash table are never used which leads to wastage of space.                                 | Buckets may be used even if no key maps to those particular buckets.                                 |

# Exercise 1

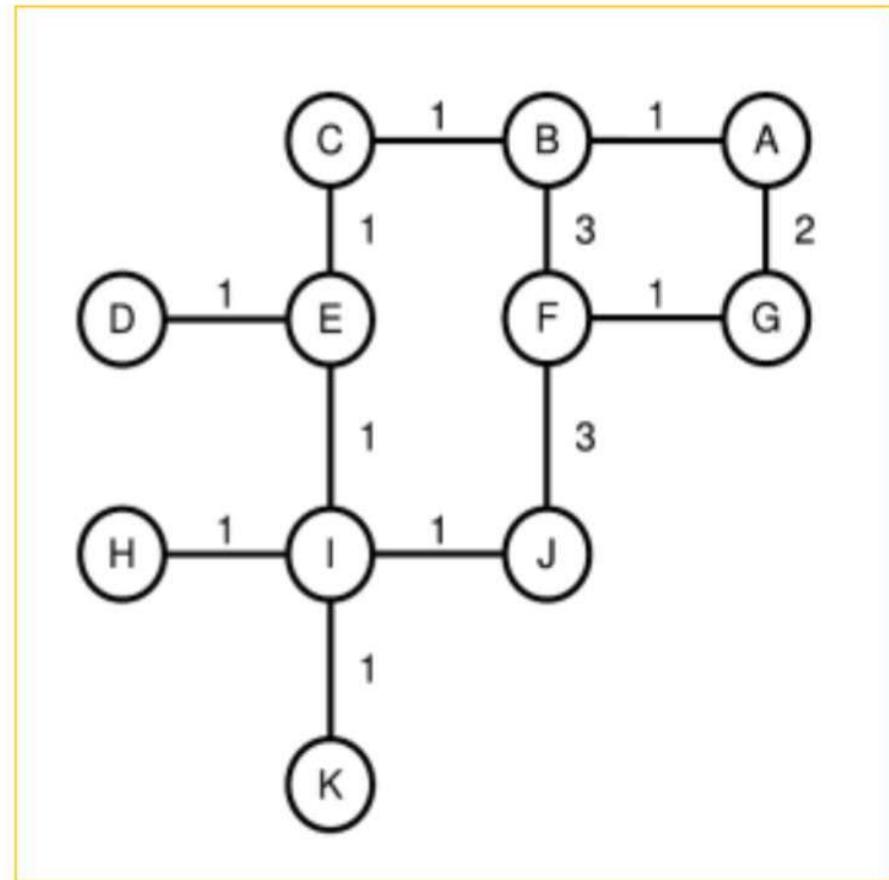
- Perform BFS and DFS on the below and assume A as the start node. Also show the final result and the cross-edges.



# Exercise 2

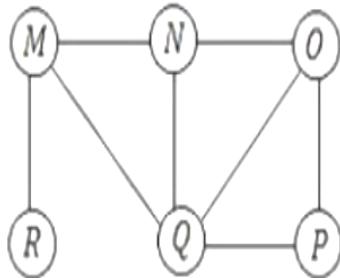
List the labels in the order they would be visited when performing:

- Breadth First Search
- Depth First Search
- Source vertex: A
- Assume that neighbors of the same node are visited in alphabetical order



# Exercise 3

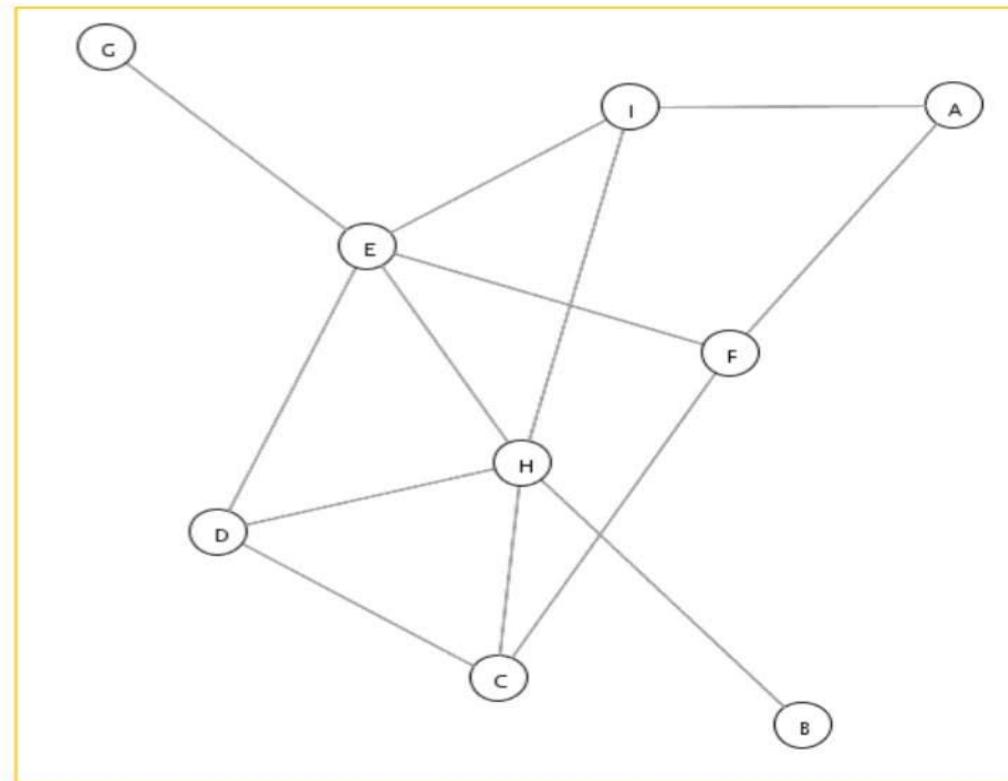
The Breadth First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below?



- A** MNOPQR
- B** NQMPOR
- C** QMNROP
- D** POQNMR

# Exercise 4

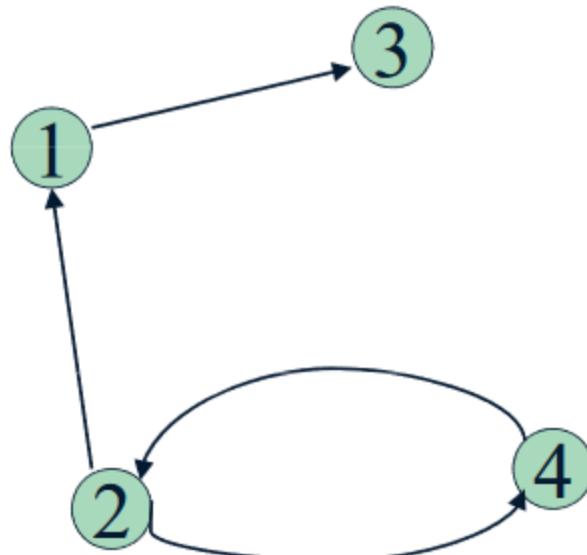
- Implement both DFS and BFS traversal. Assume C to be the source vertex.



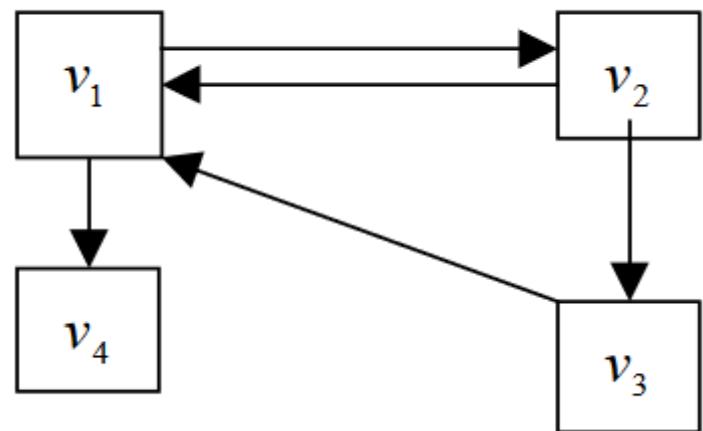
# Exercise 5

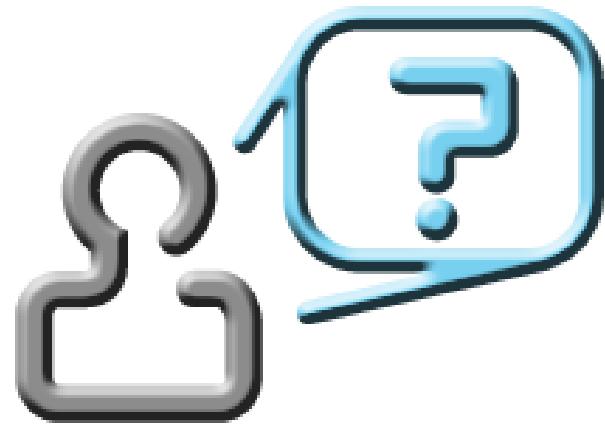
Apply the Warshall algorithm to find the transitive Closure matrix T for the following digraph's:

1)



2)





*See you in the next class to explore more on Hashing !*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : CC BY-NC-SA 4.0





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# **Data Structures and Algorithms Design**

**DSECLZG519**

**Parthasarathy**





# Contact Session #8

# DSECLZG519 – Hashing & Collisions!

# Agenda for CS #8

- 1) Recap of CS#7
  - 2) Open addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
  - 3) Load Factor & Rehashing
  - 4) Journey so far & Mid-Sem discussion
  - 5) Exercises
  - 6) Q&A

# Linear Probing

---

- Linear Probing handles collisions by placing the colliding item in the next (circularly) available table cell.
- Each table cell inspected is referred to as “probe”.
- In this method, if we try to insert an element  $(k,v)$  into a bucket  $A[i]$  that is already occupied, where  $i=h(k)$ , then we try next at  $A[(i+1) \bmod N]$ .
- This process will continue until we find an empty bucket that can accept the new entry.

# Linear probing

## ➤ Insert an item $(k, e)$

- Compute  $i = h(k)$
- If  $A[i]$  is not occupied then place element  $e$  in  $A[i]$
- If  $A[i]$  is already occupied (collision) then try the buckets  $A[(i+1)\text{mod } N]$
- If  $A[(i+ 1) \text{ mod } N]$  is occupied, then we try  $A[(i + 2) \text{ mod } N]$ , and so on, until we find an empty bucket in  $A$  that can accept the new item or we have inspected the entire array  $A[(i + N-1) \text{ mod } N]$

# Linear Probing

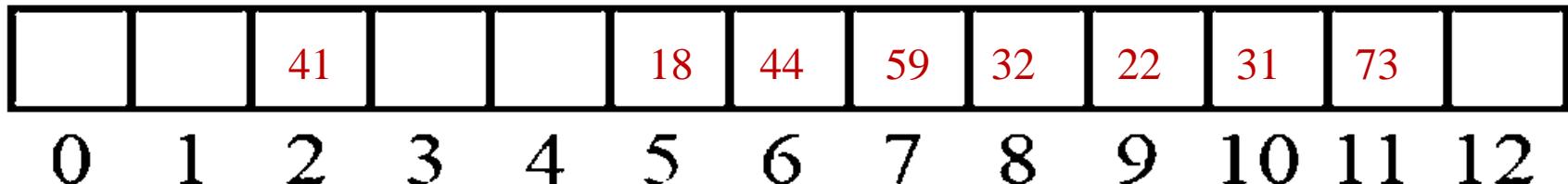
---

- Probe: Each table cell inspected to place the item is referred to as a “probe”
- *Disadvantage:* Colliding items lump together (primary clustering), causing future collisions to cause a longer sequence of probes during insertion

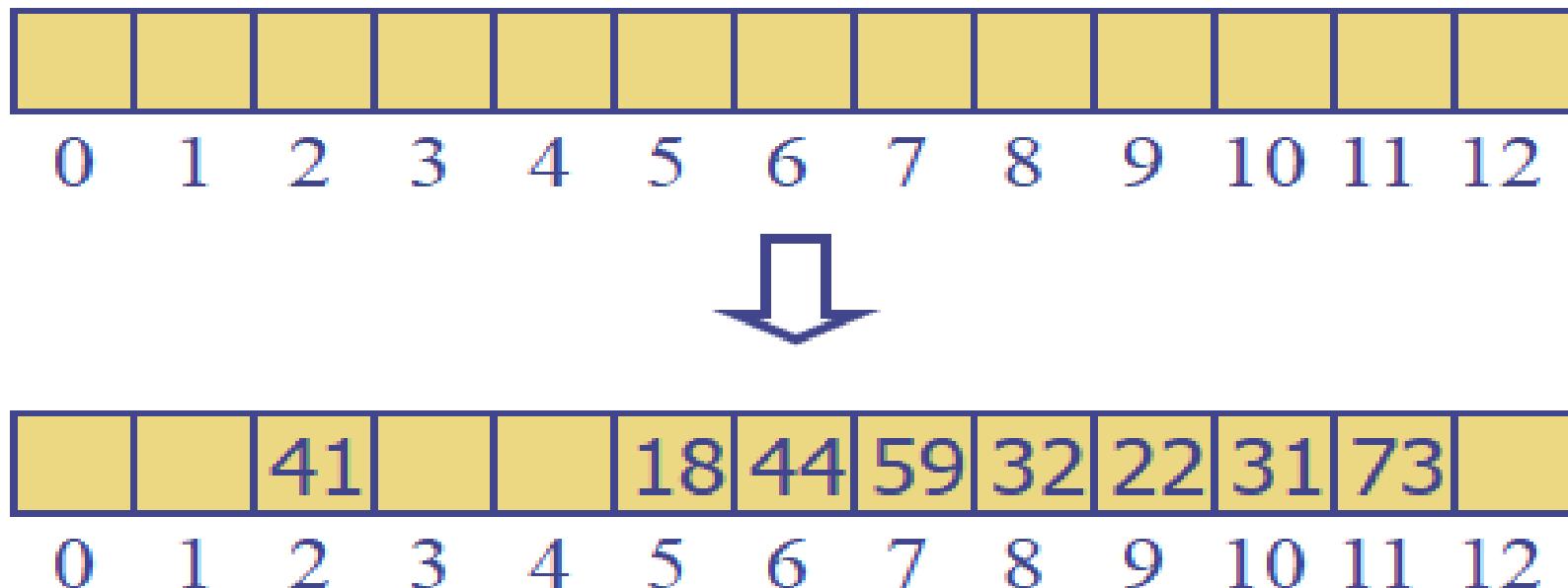
# Linear probing

- Insert the keys using linear probing in this order into a hash table using the hash function  $h(k) = k \bmod 13$ . Show your work step by step. Keys: 18, 41, 22, 44, 59, 32, 31, 73
  - $- h(k) = (5, 2, 9, 5, 7, 6, 5, 8)$

Linear Probing  
in action!



# Linear probing

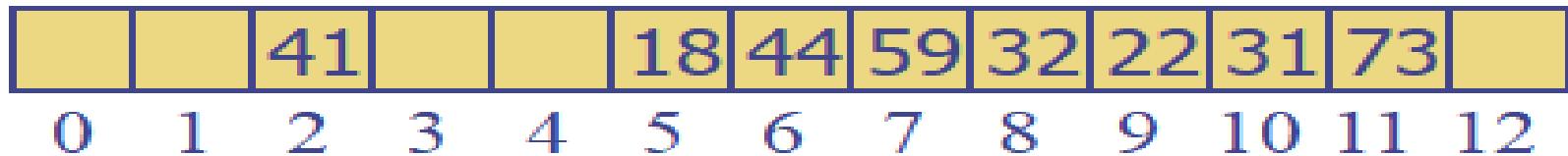


# Linear probing

- 
- **Search ( $k$ )**
    - Compute  $i = h(k)$
    - Start at array cell  $A[i]$
    - Probe consecutive locations until one of the following occurs
      - An item with key  $k$  is found return **KEY PRESENT**
      - An empty cell is found return **NO SUCH KEY**
      - $N$  cells have been unsuccessfully probed return **NO SUCH KEY**

# Linear probing

- Consider the following hash table and the hash function  $h(k) = k \bmod 13$ . Search/Find 18,31 and 57



- Search(18) = 18 mod 13 = 5, look at A[5] ; 18 == 18 , “Success”
- Search(31) = 31 mod 13 = 5, look at A[5] ; 18 != 31, Look at A[6] and so on .. A[10] = 31, “ Success”
- Search(57) = 57 mod 13 = 5, look at A[5]; 18 != 57, Look at A[6] and so on ... until its found or an empty location is found. Here, A[12] = null and hence “57 not found”

# Linear probing

Delete ( $k$ ) is (1) Search( $k$ ) (2) Once found, delete it!

## Delete ( $k$ )

- Consider the following hash table and the hash function  $h(k) = k \bmod 13$

|   |   |    |   |    |    |    |    |    |    |    |    |    |
|---|---|----|---|----|----|----|----|----|----|----|----|----|
|   |   | 41 |   | 18 | 44 | 59 | 32 | 22 | 31 | 73 |    |    |
| 0 | 1 | 2  | 3 | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

- **Delete(18)** =  $18 \bmod 13 = 5$ , goto A[5] and 18 == 18, “Delete 18”

|   |   |    |   |   |   |    |    |    |    |    |    |    |
|---|---|----|---|---|---|----|----|----|----|----|----|----|
|   |   | 41 |   |   |   | 44 | 59 | 32 | 22 | 31 | 73 |    |
| 0 | 1 | 2  | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

- **Search(31)** =  $31 \bmod 13 = 5$ , goto A[5] and its empty!
- Oops!! 31 not found

# Linear probing

---

- Cannot just delete the key we want
- Doing so might make it impossible to retrieve any key  $k$  during whose insertion we had probed this slot & found it occupied
- Use a special value **AVAILABLE** when we delete a key from a slot
  - **Search** should treat **AVAILABLE** as though the slot holds a key that does not match the one being searched for
  - **Insert** should treat **AVAILABLE** as though the slot were empty, so that it can be reused

# Linear probing

- Example
  - Consider the following hash table and the hash function  $h(k) = k \bmod 13$

|   |   |    |   |   |    |    |    |    |    |    |    |    |
|---|---|----|---|---|----|----|----|----|----|----|----|----|
|   |   | 41 |   |   | 18 | 44 | 59 | 32 | 22 | 31 | 73 |    |
| 0 | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

**Delete(18)** =  $18 \bmod 13 = 5$  , goto A[5] and delete element and mark as A

|   |   |    |   |   |   |    |    |    |    |    |    |    |
|---|---|----|---|---|---|----|----|----|----|----|----|----|
|   |   | 41 |   |   | A | 44 | 59 | 32 | 22 | 31 | 73 |    |
| 0 | 1 | 2  | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

**Search(31)**

treat **AVAILABLE** as though the slot holds a key that does not match the one being searched for and continue .. A[10] = 31

# Linear probing

## Example

- Consider the following hash table and the hash function  $h(k) = k \bmod 13$

|   |   |    |   |   |    |    |    |    |    |    |    |    |
|---|---|----|---|---|----|----|----|----|----|----|----|----|
|   |   | 41 |   |   | 18 | 44 | 59 | 32 | 22 | 31 | 73 |    |
| 0 | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

- Delete(18)**

|   |   |    |   |   |   |    |    |    |    |    |    |    |
|---|---|----|---|---|---|----|----|----|----|----|----|----|
|   |   | 41 |   |   | A | 44 | 59 | 32 | 22 | 31 | 73 |    |
| 0 | 1 | 2  | 3 | 4 | 5 | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

- Insert(57)**

|   |   |    |   |   |    |    |    |    |    |    |    |    |
|---|---|----|---|---|----|----|----|----|----|----|----|----|
|   |   | 41 |   |   | 57 | 44 | 59 | 32 | 22 | 31 | 73 |    |
| 0 | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

# Quadratic probing

- Insert an item  $(k, e)$ 
  - Compute  $i = h(k)$
  - If  $A[i]$  is not occupied then place element  $e$  in  $A[i]$
  - If  $A[i]$  is already occupied (collision)
    - Iteratively try the buckets  $A[(i + f(j)) \bmod N]$ ,
    - for  $j = 0, 1, 2, \dots, N-1$ , where  $f(j) = j^2$ ,
    - until finding an empty bucket or have examined the entire array
- Can only guarantee an successful insertion when the hash table is atmost half full !
- Can suffer from secondary clustering
  - *If two keys have the same initial position, then their probe sequences are the same*

# Quadratic probing

---

- Insert the keys using quadratic probing in this order into a hash table using the hash function  $h(k) = k \bmod 7$ . Show your work step by step. Keys: 14, 8, 21, 2, 7
  - $h(k) = 0, 1, 0, 2, 0$

# Quadratic probing

$$i = h(k) = k \bmod 7$$

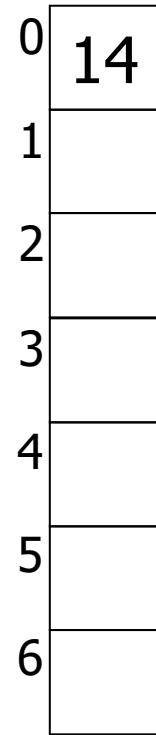
$$A[(i + j^2) \bmod N]$$

$$j = 0 \rightarrow A[0+0 \bmod 7] = A[0]$$

$$j = 1 \rightarrow A[0+1 \bmod 7] = A[1]$$

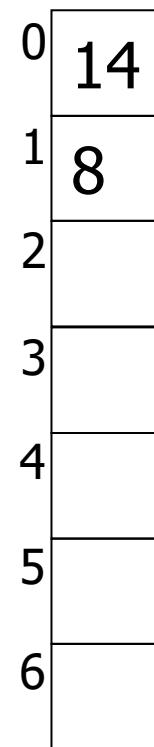
$$j = 2 \rightarrow A[0+4 \bmod 7] = A[4]$$

insert(14)  
 $14 \% 7 = 0$



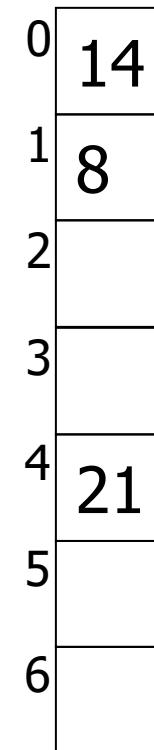
1

insert(8)  
 $8 \% 7 = 1$



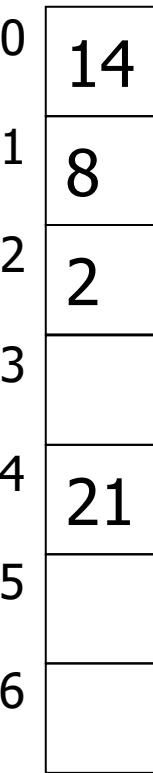
1

insert(21)  
 $21 \% 7 = 0$



3

insert(2)  
 $2 \% 7 = 2$



1

# Quadratic probing

insert(7)

$$7 \% 7 = 0$$

|   |    |
|---|----|
| 0 | 14 |
| 1 | 8  |
| 2 | 2  |
| 3 |    |
| 4 | 21 |
| 5 |    |
| 6 |    |

| j | Cells tried<br>$h(k) = (h(k) + j^2) \bmod N$ |
|---|--|
| 0 | 0 mod 7 = 0                                  |
| 1 | (0+1)mod 7 = A[1]                            |
| 2 | (0+4)mod 7 = A[4]                            |
| 3 | (0+9)mod 7 = A[2]                            |
| 4 | (0+16)mod 7 = A[2]                           |
| 5 | (0+25)mod 7 = A[4]                           |
| 6 | (0+36)mod 7 = A[1]                           |

# Linear vs Quadratic Probing

---

- An advantage of linear probing is that it can reach every location in the hash table.
- This property is important since it guarantees the success of the *insertItem* operation when the hash table is not full.
- Quadratic probing can only guarantees a successful *insertItem* operation when the hash table is atmost half full.

# Double hashing

- Double hashing uses a secondary hash function  $d(k)$
- Handles collisions by placing an item in the first available cell of the series

$$(i + j d(k)) \bmod N$$

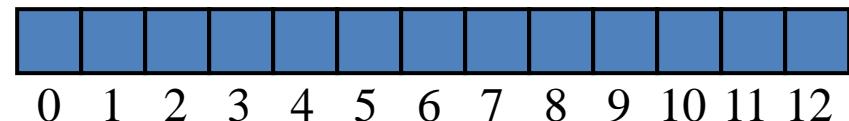
where  $j = 0, 1, \dots, N - 1$  and  $i = h(k)$  and  $d(k)$  is secondary hash function

- The secondary hash function cannot have zero values.
- The table size  $N$  must be a prime to allow probing of all the cells.
- Choose a secondary hash function that will attempt to minimize clustering as much as possible. Usually taken as  $q - (k \bmod q)$  where  $q$  is prime and  $q < N$

# Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - (k \bmod 7)$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order into the hash table

| $k$ | $h(k)$ | $d(k)$ |
|-----|--------|--------|
| 18  | 5      | 3      |
| 41  | 2      | 1      |
| 22  | 9      | 6      |
| 44  | 5      | 5      |
| 59  | 7      | 4      |
| 32  | 6      | 3      |
| 31  | 5      | 4      |
| 73  | 8      | 4      |

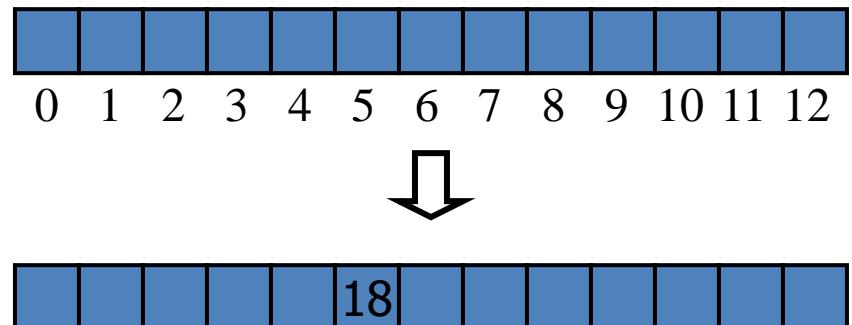


# Example of Double Hashing

## Insert key 18

- place an item in the first available cell of the series  $(i + j d(k)) \bmod N$  for  $j = 0, 1, \dots, N - 1$
- $i = h(k) = 5; j = 0; d(k) = 3;$   
 $N = 13$
- $5+0 \bmod 13 = 5$
- $A[5] = 18$

| $k$ | $h(k)$ | $d(k)$ | Probes |
|-----|--------|--------|--------|
| 18  | 5      | 3      | 5      |
| 41  | 2      | 1      | 2      |
| 22  | 9      | 6      | 9      |
| 44  | 5      | 5      | 5 10   |
| 59  | 7      | 4      | 7      |
| 32  | 6      | 3      | 6      |
| 31  | 5      | 4      | 5 9 0  |
| 73  | 8      | 4      | 8      |

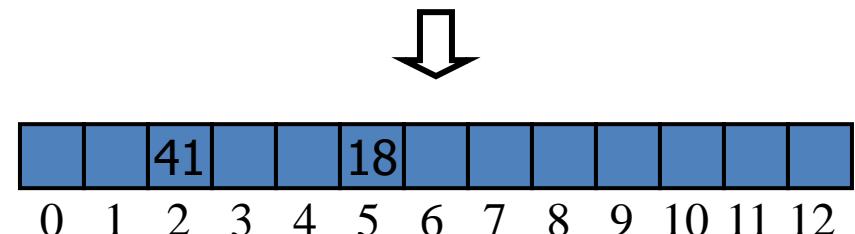
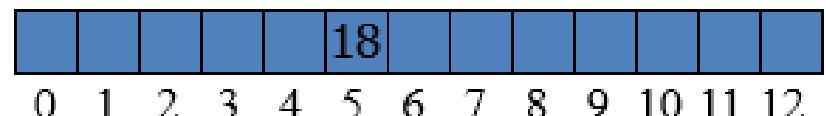


# Example of Double Hashing

## Insert key 41

- place an item in the first available cell of the series  
 $(i + jd(k)) \text{ mod } N$   
for  $j = 0, 1, \dots, N - 1$
- $i = 2; j = 0; d(k) = 1; N = 13$
- $2+0 \text{ mod } 13 = 2$
- $A[2] = 41$

| $k$ | $h(k)$ | $d(k)$ | Probes |
|-----|--------|--------|--------|
| 18  | 5      | 3      | 5      |
| 41  | 2      | 1      | 2      |
| 22  | 9      | 6      | 9      |
| 44  | 5      | 5      | 5 10   |
| 59  | 7      | 4      | 7      |
| 32  | 6      | 3      | 6      |
| 31  | 5      | 4      | 5 9 0  |
| 73  | 8      | 4      | 8      |

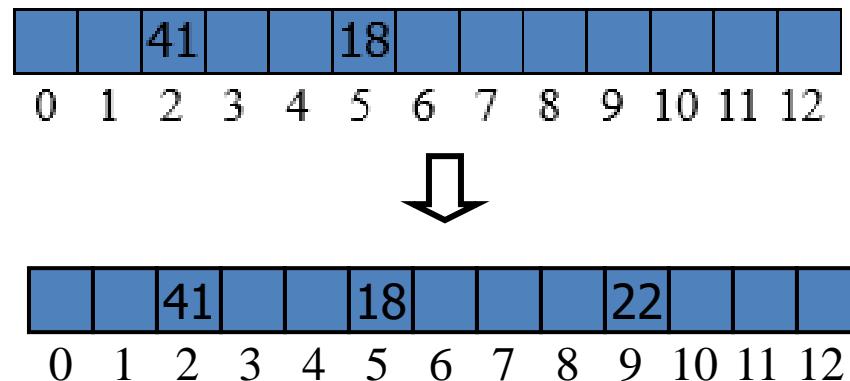


# Example of Double Hashing

## Insert key 22

- place an item in the first available cell of the series  $(i + jd(k)) \bmod N$  for  $j = 0, 1, \dots, N - 1$
- $i = 9; j = 0; d(k) = 6; N = 13$
- $9+0 \bmod 13 = 9$
- $A[9] = 22$

| $k$ | $h(k)$ | $d(k)$ | Probes |
|-----|--------|--------|--------|
| 18  | 5      | 3      | 5      |
| 41  | 2      | 1      | 2      |
| 22  | 9      | 6      | 9      |
| 44  | 5      | 5      | 5 10   |
| 59  | 7      | 4      | 7      |
| 32  | 6      | 3      | 6      |
| 31  | 5      | 4      | 5 9 0  |
| 73  | 8      | 4      | 8      |

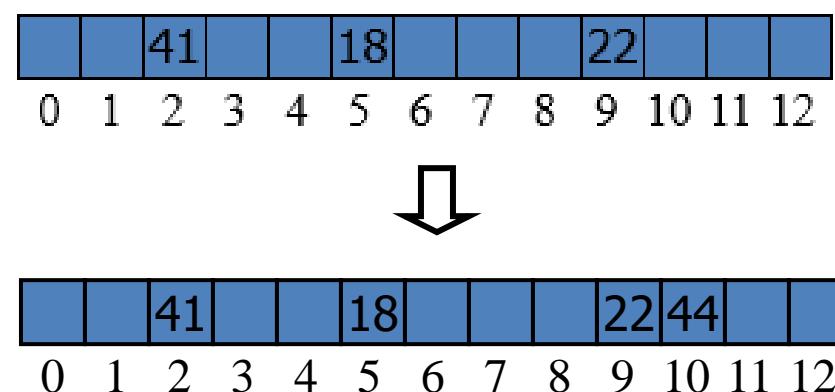


# Example of Double Hashing

## Insert key 44

- place an item in the first available cell of the series  
 $(i + jd(k)) \text{ mod } N$   
 for  $j = 0, 1, \dots, N - 1$
- $i = 5; j = 0; d(k) = 5; N = 13$
- $5+0 \text{ mod } 13 = 5; A[5] - \text{occupied}$
- Try with  $j=1$*
- $5+(1*5) \text{ mod } 13 = 10$
- $A[10] = 44$

| $k$ | $h(k)$ | $d(k)$ | Probes |     |
|-----|--------|--------|--------|-----|
| 18  | 5      | 3      | 5      |     |
| 41  | 2      | 1      | 2      |     |
| 22  | 9      | 6      | 9      |     |
| 44  | 5      | 5      | 5      | 10  |
| 59  | 7      | 4      | 7      |     |
| 32  | 6      | 3      | 6      |     |
| 31  | 5      | 4      | 5      | 9 0 |
| 73  | 8      | 4      | 8      |     |

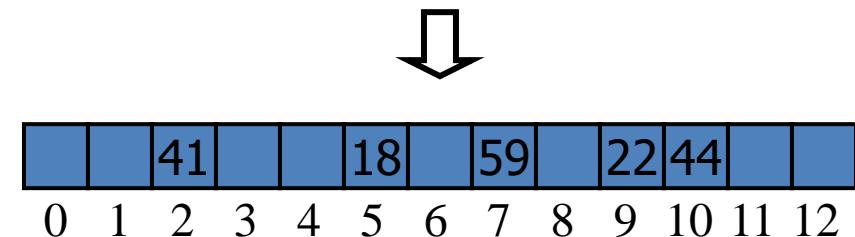


# Example of Double Hashing

## Insert key 59

- place an item in the first available cell of the series  $(i + jd(k)) \text{ mod } N$   
for  $j = 0, 1, \dots, N - 1$
- $i = 7; j = 0; d(k) = 4; N = 13$
- $7+0 \text{ mod } 13 = 7$
- $A[7] = 59$

| $k$ | $h(k)$ | $d(k)$ | Probes |
|-----|--------|--------|--------|
| 18  | 5      | 3      | 5      |
| 41  | 2      | 1      | 2      |
| 22  | 9      | 6      | 9      |
| 44  | 5      | 5      | 5 10   |
| 59  | 7      | 4      | 7      |
| 32  | 6      | 3      | 6      |
| 31  | 5      | 4      | 5 9 0  |
| 73  | 8      | 4      | 8      |

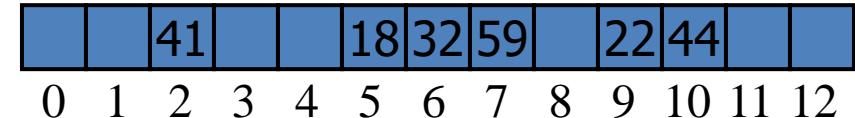
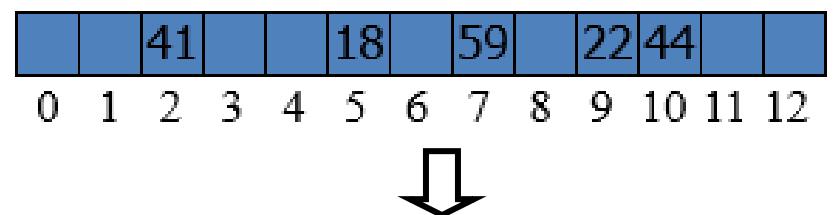


# Example of Double Hashing

## Insert key 32

- place an item in the first available cell of the series  $(i + jd(k)) \bmod N$  for  $j = 0, 1, \dots, N - 1$
- $i = 6; j = 0; d(k) = 3; N = 13$
- $6+0 \bmod 13 = 6$
- $A[6] = 32$

| $k$ | $h(k)$ | $d(k)$ | Probes |
|-----|--------|--------|--------|
| 18  | 5      | 3      | 5      |
| 41  | 2      | 1      | 2      |
| 22  | 9      | 6      | 9      |
| 44  | 5      | 5      | 5 10   |
| 59  | 7      | 4      | 7      |
| 32  | 6      | 3      | 6      |
| 31  | 5      | 4      | 5 9 0  |
| 73  | 8      | 4      | 8      |

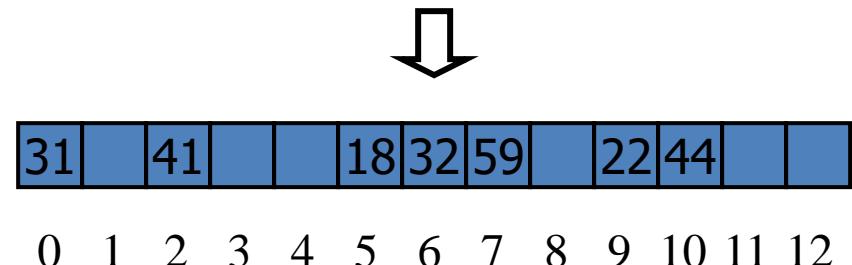


# Example of Double Hashing

## Insert key 31

- place an item in the first available cell of the series  $(i + jd(k)) \text{ mod } N$  for  $j = 0, 1, \dots, N - 1$
- $i = 5; j = 0; d(k) = 4; N = 13$
- $5+0 \text{ mod } 13 = 5$
- $5+(1*4) \text{ mod } 13 = 9 \ (j=1)$
- $5+(2*4) \text{ mod } 13 = 0 \ (j=2)$
- $A[0] = 31$

| $k$ | $h(k)$ | $d(k)$ | Probes |
|-----|--------|--------|--------|
| 18  | 5      | 3      | 5      |
| 41  | 2      | 1      | 2      |
| 22  | 9      | 6      | 9      |
| 44  | 5      | 5      | 5 10   |
| 59  | 7      | 4      | 7      |
| 32  | 6      | 3      | 6      |
| 31  | 5      | 4      | 5 9 0  |
| 73  | 8      | 4      | 8      |



# Example of Double Hashing

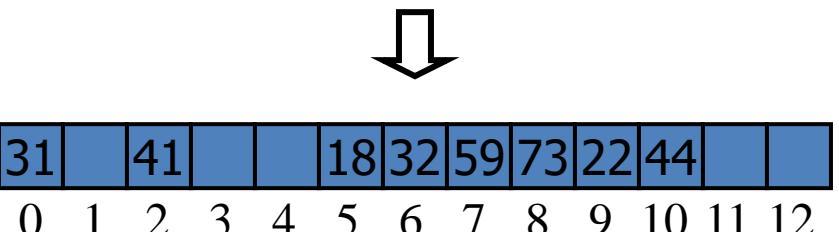
## Insert key 73

- place an item in the first available cell of the series  $(i + jd(k)) \bmod N$  for  $j = 0, 1, \dots, N - 1$
- $i = 8; j = 0; d(k) = 4; N = 13$
- $8+0 \bmod 13 = 8$
- $A[8] = 73$

| $k$ | $h(k)$ | $d(k)$ | Probes |
|-----|--------|--------|--------|
| 18  | 5      | 3      | 5      |
| 41  | 2      | 1      | 2      |
| 22  | 9      | 6      | 9      |
| 44  | 5      | 5      | 5 10   |
| 59  | 7      | 4      | 7      |
| 32  | 6      | 3      | 6      |
| 31  | 5      | 4      | 5 9 0  |
| 73  | 8      | 4      | 8      |



0 1 2 3 4 5 6 7 8 9 10 11 12



# Load factor

---

- Suppose that
  - the bucket array is of capacity  $N$
  - there are  $n$  *entries in the hash table*
- The load factor is defined as  $\lambda = n/N$
- Load factor is kept below a small constant, preferably below 1 (i.e.)  $\lambda < 1$
- Usually  $\lambda$  is kept as 0.75
  - But this requires additional work

# Rehashing

- Whenever we add elements, it is important to keep the load factor below a specified constant
  - Increase the size of bucket array
    - double the size of the original array, choosing the size of the new array to be a prime number
  - Change hash function to match this new size
  - Insert all the existing hash table elements into the new bucket array using the new hash function
- Such a size increase and hash table rebuild is called **rehashing**

# Rehashing

- What is the load factor of the hash table? Elements of the hash table are rehashed into another hash table of size 17 using linear probing. Let the new hash function be  $h(x) = x \bmod 17$ .

|   |    |
|---|----|
| 0 | 6  |
| 1 | 15 |
| 2 | 23 |
| 3 | 24 |
| 4 |    |
| 5 |    |
| 6 | 13 |

# Rehashing

|   |    |
|---|----|
| 0 | 6  |
| 1 | 15 |
| 2 | 23 |
| 3 | 24 |
| 4 |    |
| 5 |    |
| 6 | 13 |

$$\lambda = n/N \\ = 5/7 = 0.7$$



|    |    |
|----|----|
| 0  |    |
| 1  |    |
| 2  |    |
| 3  |    |
| 4  |    |
| 5  |    |
| 6  | 6  |
| 7  | 23 |
| 8  | 24 |
| 9  |    |
| 10 |    |
| 11 |    |
| 12 |    |
| 13 | 13 |
| 14 |    |
| 15 | 15 |
| 16 |    |

$$\lambda = n/N \\ = 5/17 = 0.3$$

# Hash Table

- The worst case running time of operations like insert, delete and search is  $O(n)$  where n is the number of items in the hash table.

| Method            | Worst case performance |
|-------------------|------------------------|
| Separate Chaining | $O(N)$                 |
| Linear Probing    | $O(N)$                 |
| Quadratic Probing | $O(N)$                 |
| Double Hashing    | $O(N)$                 |

# Applications

---

- DNS resolution
- Password Verification (local and server-side)
- Compilers: Symbol/keyword table
- Online spell check tools
- Pattern matching and plagiarism check

# Journey so far ...

CS#1  
Introduction & Analysis framework

CS#2  
Analysis and Master's Theorem

CS#3  
Linear Data structures

CS#4  
Introduction to Trees & traversals

CS#5  
Heaps , Heap Sort!

CS#6  
Priority Queues, Introduction to Graphs

CS#7  
Exploring graph traversals

CS#8  
Hashing & Collision. Webinar  
1: Stacks & Amortized. Webinar  
2: BST

Mid Semester Examination – All the Best!

# Mid-Semester Examination

- Portions for mid-semester examination will be till CS#8 (todays session). Portions has been clearly put as an announcement as well in canvas by the IC. Do have an eye on it !
- Sample/past paper will be available in canvas - 2 weeks before exam date.
- Weightage : 30%
- Nature: Subjective
- Mode : Online via Wheebbox
- Type : Open book
- Duration : 2Hours

# Exercise 1

1. Implement Double hashing for the below given list:

- Table Size 15
- Insert 16, 7, 28, 31, 67, 28, 29, 73, 99, 43, 218
- Use below given hash functions.
  - $k \bmod 15$
  - $7 - k \bmod 7$

- Visualizations:  
<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>
- Credits: USFCA
- Purpose: Many concepts are illustrated visually here. Have a look if this interests you ☺

# Exercise 2

---

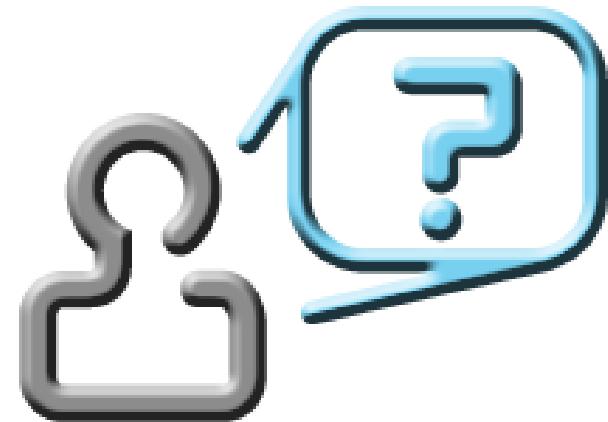
1. Implement Double hashing for the below given list:
  - Table Size 15
  - Insert 16, 7, 28, 31, 67, 28, 29, 73, 99, 43, 218
  - Use below given hash functions.
    - $k \bmod 15$
    - $7 - k \bmod 7$
2. Given the input {42,39,57,3,18,5,67,13,70,26}, and a fixed table size of 13, and a hash-function  $H(X) = X \bmod 13$ , show the resulting
  - 1.Linear probing hash-table (Circle the elements which causes collision)
  - 2.Separate chaining hash-table

# Exercise 3

---

Given the input {4371,1323,6173,4199,4344,9679,1989}, and a fixed table size of 10, and a hash-function  $H(X) = X \bmod 10$ , show the resulting

- 1.Linear probing hash-table
- 2.Quadratic probing hash-table
- 3.Separate chaining hash-table



*All the Best !!*

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : CC BY-NC-SA 4.0





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# **Data Structures and Algorithms Design**

## **DSECLZG519**

**Parthasarathy**





# Contact Session #9

## DSECLZG519 – Bloom Filters, Kd and AVL Trees!

# Journey So Far ...

CS#1  
Introduction & Analysis framework

CS#2  
Analysis and Master's Theorem

CS#3  
Linear Data structures

CS#4  
Introduction to Trees & traversals

CS#5  
Heaps , Heap Sort!

CS#6  
Priority Queues, Introduction to Graphs

CS#7  
Exploring graph traversals

CS#8  
Hashing & Collision. Webinar  
1: Stacks & Amortized. Webinar  
2: BST

Mid Semester Examination – All the Best!

# Agenda for Session # 9

- 
- Mid-semester Discussion
  - Esoteric Data structures:
    - Bloom Filters
    - Kd Trees
  - BST Recap
  - AVL Trees
  - Exercises

# Esoteric Data Structures

- In DSECLZG519, we have been talking about many standard, famous and commonly used data structures like Array, Linked list, trees with its different types, Hash tables and Graphs.
- However, we only scratched the surface of Data structures and DS research is still alive 😊
- We will learn two such esoteric data structures in our course : Bloom filters and kd-trees.

difference-list  
table  
b-tree  
2-3-heap  
heightmap  
splay-tree  
lookup  
hash  
rolling hash  
skip-list  
kd-tree

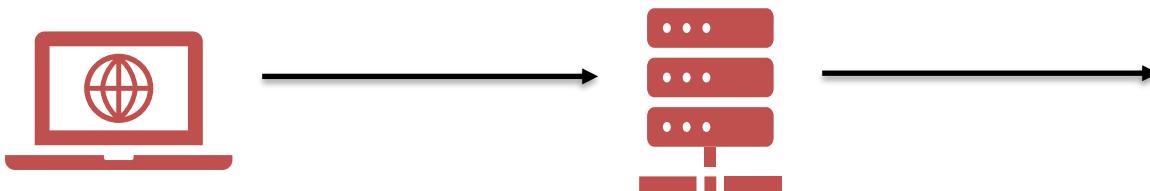
*Esoteric : intended for or likely to be understood by only a small number of people with a specialized knowledge or interest.*

# Motivation for Bloom Filters ?



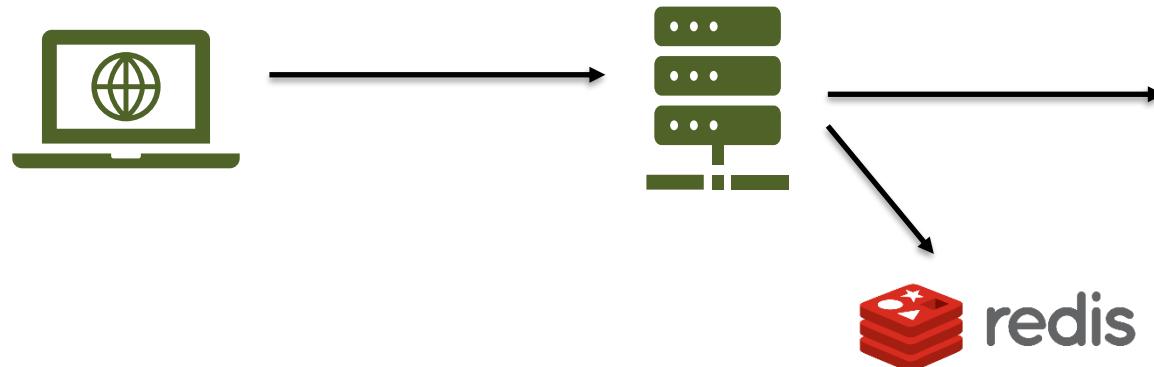
*Check if the username “Atul” exists ?*

**GET call for “Atul” :**



**SLOW !!!**

**GET call for “Atul” :**



**Inefficient  
memory wise !!!**

# Motivation for Bloom Filters ?

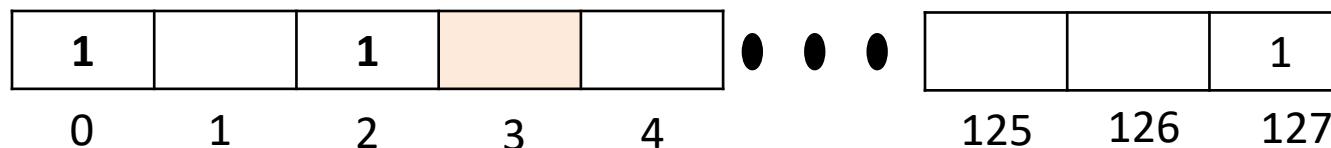
Check if the username “Atul” exists ?

**GET** call for “Atul” :



Efficient ... If  
value is 0 in bit  
array, no need  
to hit DB!

$$\text{Hash(Atul)} = 3$$



$$\text{Hash(Ravi)} = 0$$

Note : The hash values 3 and 1 are assumption here. In reality this value depends on hash function used.

# Bloom Filters

---

- Our first esoteric data structure is called a bloom filter, named for its creator, Burton Howard Bloom, who invented the data structure in 1970.
- A bloom filter is a space efficient, probabilistic data structure that is used *to tell whether a member is in a set*.
- Bloom filters are a bit odd because they can **definitely** tell you whether an element is not in the set, but can only say whether the element is **possibly** in the set.
- In other words: “*false positives*” are possible, but “*false negatives*” are not possible with bloom filters.
- A *false positive* would say that the element is in the set when it isn’t, and a *false negative* would say that the element is not in the set when it is.

# Bloom Filters

- The idea is that we have a “bit array.” We will model a bit array with a regular array

a bit array :

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

- Bloom Filters: start with an empty bit array (all zeros), and k hash functions.
- $k1 = (13 - (x \% 13)) \% 7$ ,
- $k2 = (3 + 5x) \% 7$ , etc.

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

# Bloom Filters

- The hash functions should be independent, and the optimal amount is calculable based on the number of items you are hashing, and the length of your table (*see Wikipedia for details*).
- Values then get hashed by all k hashes, and the bit in the hashed position is set to 1 in each case.
- Example : Insert **129** → x=129

$$\begin{aligned}
 k1 &= (13 - (x \% 13)) \% 7 \\
 &= (13 - (129 \% 13)) \% 7 \\
 &= (13 - 12) \% 7 \\
 &= 1 \% 7 \\
 &= 1
 \end{aligned}$$

$$\begin{aligned}
 k2 &= (3 + 5*x) \% 7 \\
 &= (3 + 5*129) \% 7 \\
 &= 648 \% 7 \\
 &= 4
 \end{aligned}$$

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

$k1 == 1$ , so we change bit 1 to a 1  
 $k2 == 4$ , so we change bit 4 to a 1

# Bloom Filters

Insert **479** :  $x = 479$  and  $k1 = (13 - (x \% 13)) \% 7$ ,  $k2 = (3 + 5x) \% 7$

$$\begin{aligned}
 k1 &= (13 - (x \% 13)) \% 7 \\
 &= (13 - (479 \% 13)) \% 7 \\
 &= (13 - 11) \% 7 \\
 &= 2
 \end{aligned}$$

$$\begin{aligned}
 k2 &= (3 + 5x) \% 7 \\
 &= (3 + 5*479) \% 7 \\
 &= (2398) \% 7 \\
 &= 4
 \end{aligned}$$

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

$k1 == 2$ , so we change bit 2 to a 1

$k2 == 4$ , so we would change bit 4 to a 1, but it is already a 1

# Bloom Filters

- To check if 129 is in the table, just hash again using k1 and k2 and check the bits in the bit array.
- In this case, k1=1, k2=4: ***probably*** in the table!

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

- To check if 123 is in the table, hash and check the bits.  
k1=0, k2=2: ***definitely cannot be*** in table because the 0 bit is still 0

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

# Bloom Filters

- To check if 402 is in the table, hash and check the bits.  
 $k1=1, k2=4$ : ***Probably*** in the table (**but isn't! False positive!**)

$$\begin{aligned}
 k1 &= (13 - (x \% 13)) \% 7 \\
 &= (13 - (402 \% 13)) \% 7 \\
 &= (13 - 12) \% 7 \\
 &= 1 \% 7 \\
 &= 1
 \end{aligned}$$

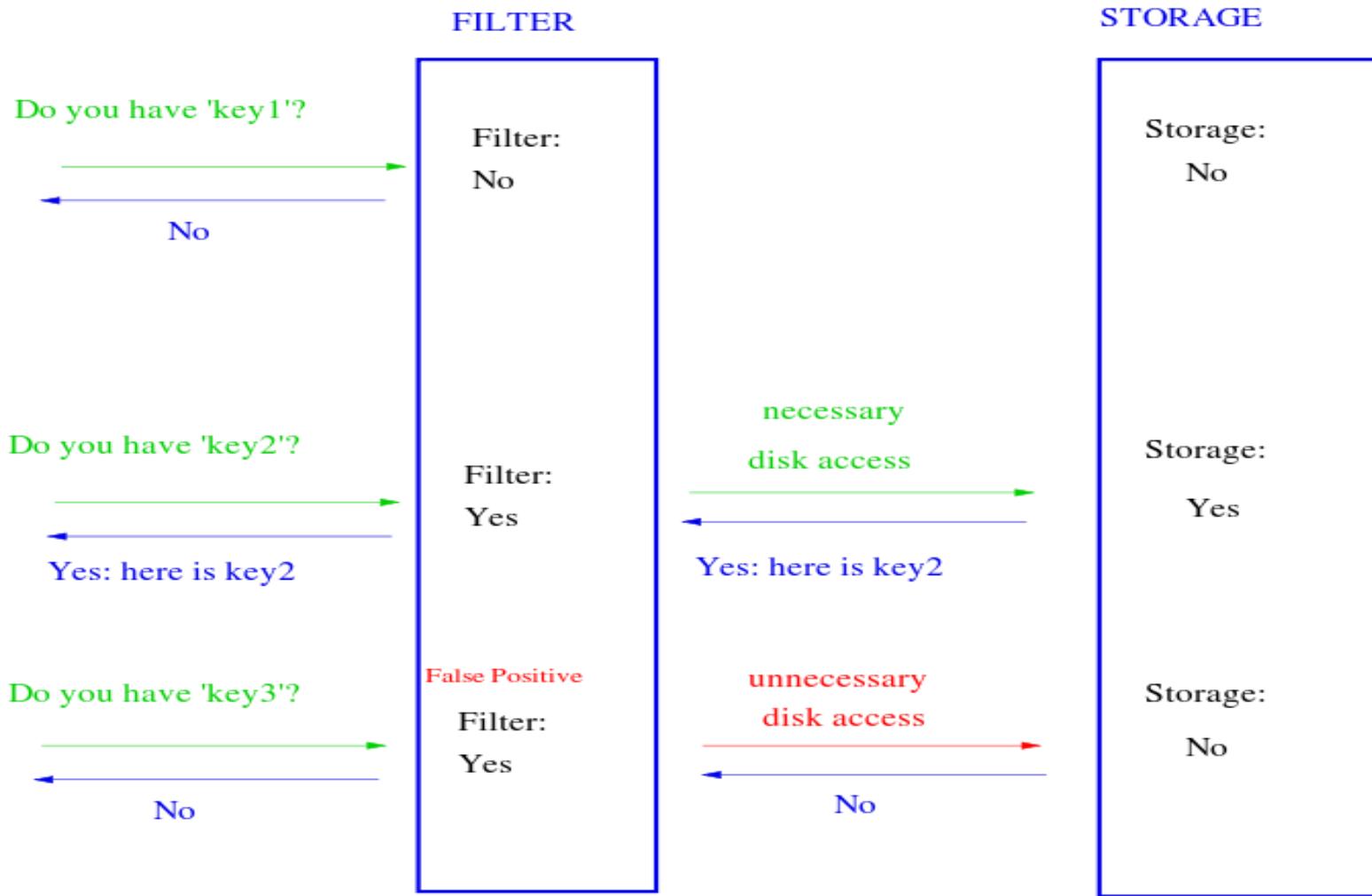
$$\begin{aligned}
 k2 &= (3 + 5*x) \% 7 \\
 &= (3 + 5*402) \% 7 \\
 &= 2013 \% 7 \\
 &= 4
 \end{aligned}$$

129 and 402  
result in same bits

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

Interactive Example : <https://llimllib.github.io/bloomfilter-tutorial/>

# Bloom Filters – Pictorial



# Bloom Filters: Probability of a False Positive



What is the probability that we have a false positive?

- ✓ If  $m$  is the number of bits in the array, then the probability that a bit is not set to 1 is

$$1 - \frac{1}{m}$$

- ✓ If  $k$  is the number of hash functions, the probability that the bit is not set to 1 by any hash function is

$$\left(1 - \frac{1}{m}\right)^k$$

- ✓ If we have inserted  $n$  elements, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn}$$

# Bloom Filters

- ✓ To get the probability that a bit is 1 is just 1- the answer on the previous slide:

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

- ✓ Now test membership of an element that is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, (false positive):

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

# Bloom Filters

Lets put this formula in practice ! For our example :

- The bloom filter size is 7, so  $m = 7$
- We inserted two elements (129, 479), so  $n = 2$
- We used two hash function, so  $k = 2$

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \Rightarrow \left(1 - \left(1 - \frac{1}{7}\right)^{2 \cdot 2}\right)^2 \Rightarrow \left(1 - \left(1 - \frac{1}{7}\right)^4\right)^2 \\ \Rightarrow (1 - (0.857142)^4)^2 \\ \Rightarrow (1 - 0.539775093)^2 \\ \Rightarrow (0.460224906)^2 \\ \Rightarrow 0.2118 \text{ or } \sim 21\% \text{ of the time} \\ \text{we will get a false positive.}$$

# Bloom Filters Applications

Why would we want a structure that can produce false positives ?

- Example: Google Chrome uses a local Bloom Filter to check for malicious URLs — if there is a hit, a stronger check is performed!
- Weak password detection
- Internet Cache Protocol
- Wallet synchronization in Bitcoin
- Cyber security like virus scanning
- Facebook uses bloom filters for typeahead search, to fetch friends and friends of friends to a user typed query. The bloom filter is just 16 bits per friend connection (an edge in the facebook social graph) and they have called it "world's smallest bloom" ☺

*Tip : Just google about these applications to understand better as to why these use bloom filters!*

# Bloom Filters

---

## Positives :

- You have to perform  $k$  hashing functions for an element, and then either flip bits, or read bits. Therefore, they perform in  $O(k)$  time, which is independent of the number of elements in the structure. Additionally, because the hashes are independent, they can be parallelized, which gives drastically better performance with multiple processors

## Negatives :

- There is one more ( $1^{\text{st}}$  is false positives) negative issue with a Bloom Filter: you can't delete! If you delete, you might delete another inserted value, as well! You could keep a second bloom filter of removals, but then you could get false positives in that filter...

# Bloom Filters - Exercise

---

- Visit following website:
- <http://llimllib.github.io/bloomfilter-tutorial/>
  
- Add few strings say: “hello”, ‘hi”, “bye”
  
- Then check for possibility of following strings:
  - “hi”
  - “hello”
  - “good”
- Observe the result

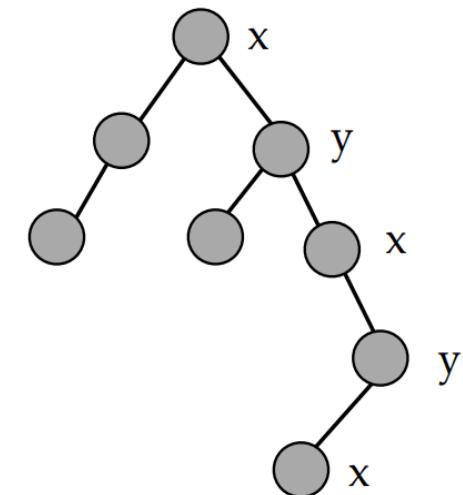
# What is k-dimensional data?

---

- If we have a set of ages say, {20, 45, 36, 75, 87, 69, 18}, these are one dimensional data.
- Because each data in the array is a single value that represents age.
- What if instead of only age we have to also store the salary for a person?
- The data would look like [{20, 1500}, {45, 5000}, {36, 4000}, {75, 2000}, {87, 0}, {18, 1000}].
- This data is two dimensional as each data set contains two values.
- Similarly, if we add one more attribute to it, say education it would be a 3 dimensional data and so on.

# K-d Trees

- K-d trees is a data structure for storing finite set of multi-dimensional [k-dimensional] points. It was invented in 1970s by Jon Bentley.
- Name originally meant “3d-trees, 4d-trees” where k was the # of dimensions
- A Binary tree
  - Each level of the tree compares against 1 dimension, also called as “cutting/splitting” dimension
  - Each node contains a point  $P = (x,y)$
- Idea: Each level of the tree compares against 1 dimension.



# K-d Trees

---

- A k-d tree is a data structure that partitions space by repeatedly choosing a point in the data set to split the current partition in half. It accomplishes this by alternating the dimension which performs the split.
- A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning data structure for organizing points in a K-Dimensional space.
- A recursive space partitioning tree.
  - Partition along x and y axis in an alternating fashion.
  - Each internal node stores the splitting node along x (or y)

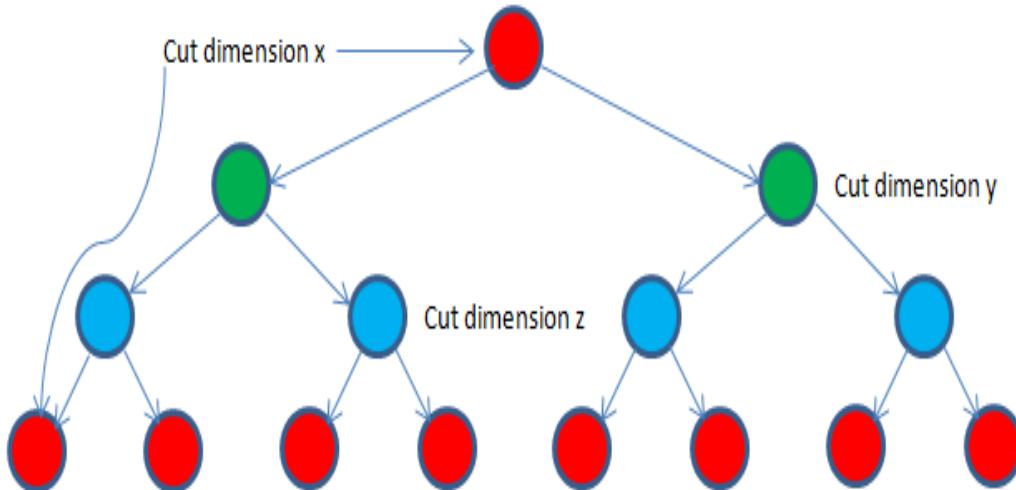
# How do we split the points?

---

- We can extend the idea of BST to do this!
- This is what Jon Louis Bentley created in 1975.
- K-d tree is called 2-d tree or k-d tree with 2-dimension when  $k = 2$  and so on.
- In BST, at each level of the tree we split the data points based on the data value.
- Since, BST deals with just one dimension the question does not arise which dimension.
- In k-d tree since we have more than one dimension.
- At each level we can choose to split the data based on only one dimension.

# How do we split the points?

- So if we have 3 dimensions: x, y and z, at first level we split the data sets using x dimension. At 2<sup>nd</sup> level we do so using y dimension and at 3<sup>rd</sup> level we use z dimension.
- At 4<sup>th</sup> level we start again with x dimension and so on.
- If we are splitting the points based on x dimension for a certain level then we call x the *cutting dimension* for this level.

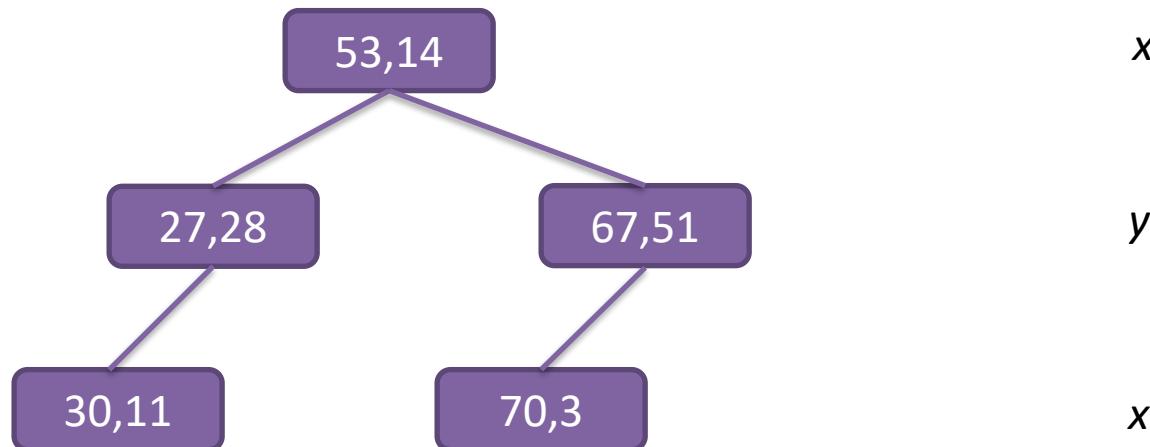


The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

# K-d Tree Construction

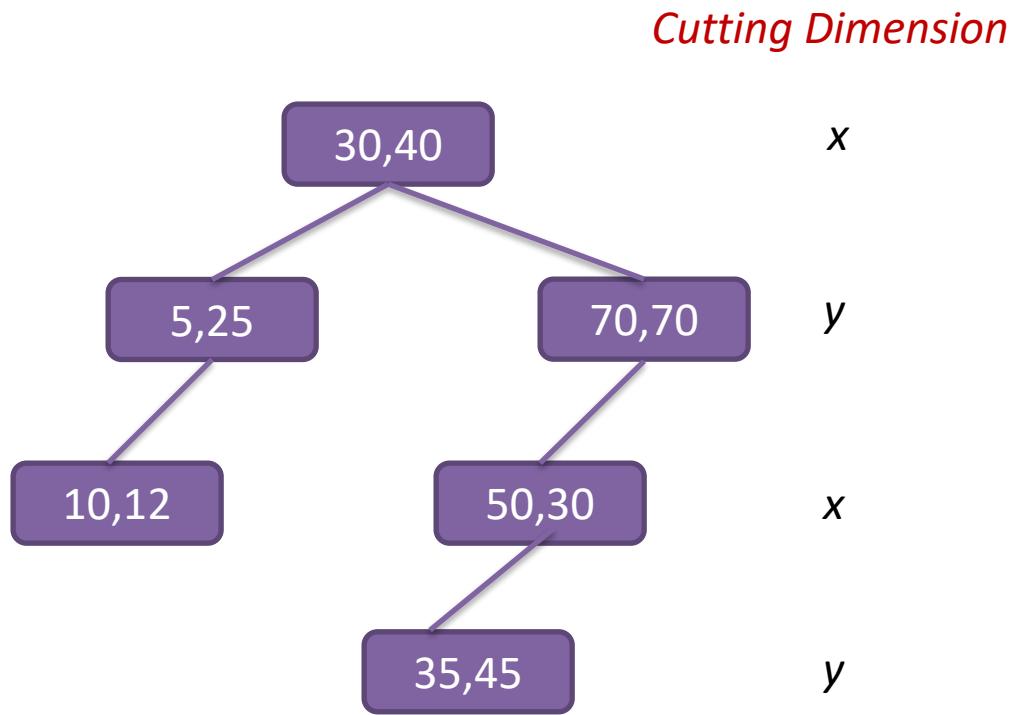
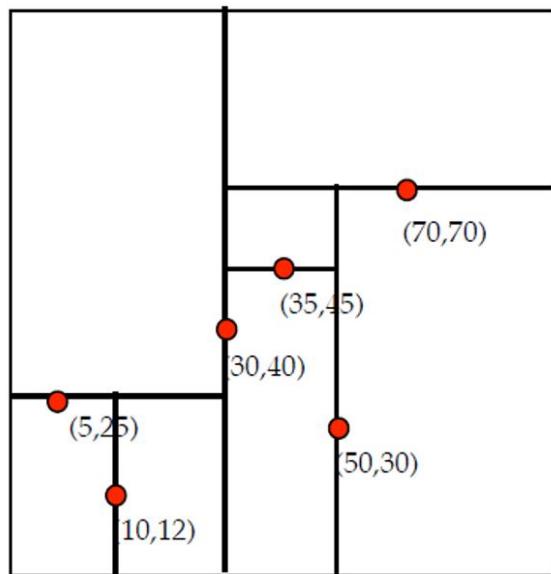
- Insert the following data points into an initially empty 2d tree :  
 $(53,14)$  ,  $(27,28)$  ,  $(30,11)$  ,  $(67,51)$  ,  $(70,3)$

*Cutting Dimension*



# K-d Tree Construction Example 2

- Insert the following data points into an initially empty 2d tree :  
 $(30,40)$  ,  $(5,25)$  ,  $(10,12)$  ,  $(70,70)$  ,  $(50,30)$  ,  $(35,45)$

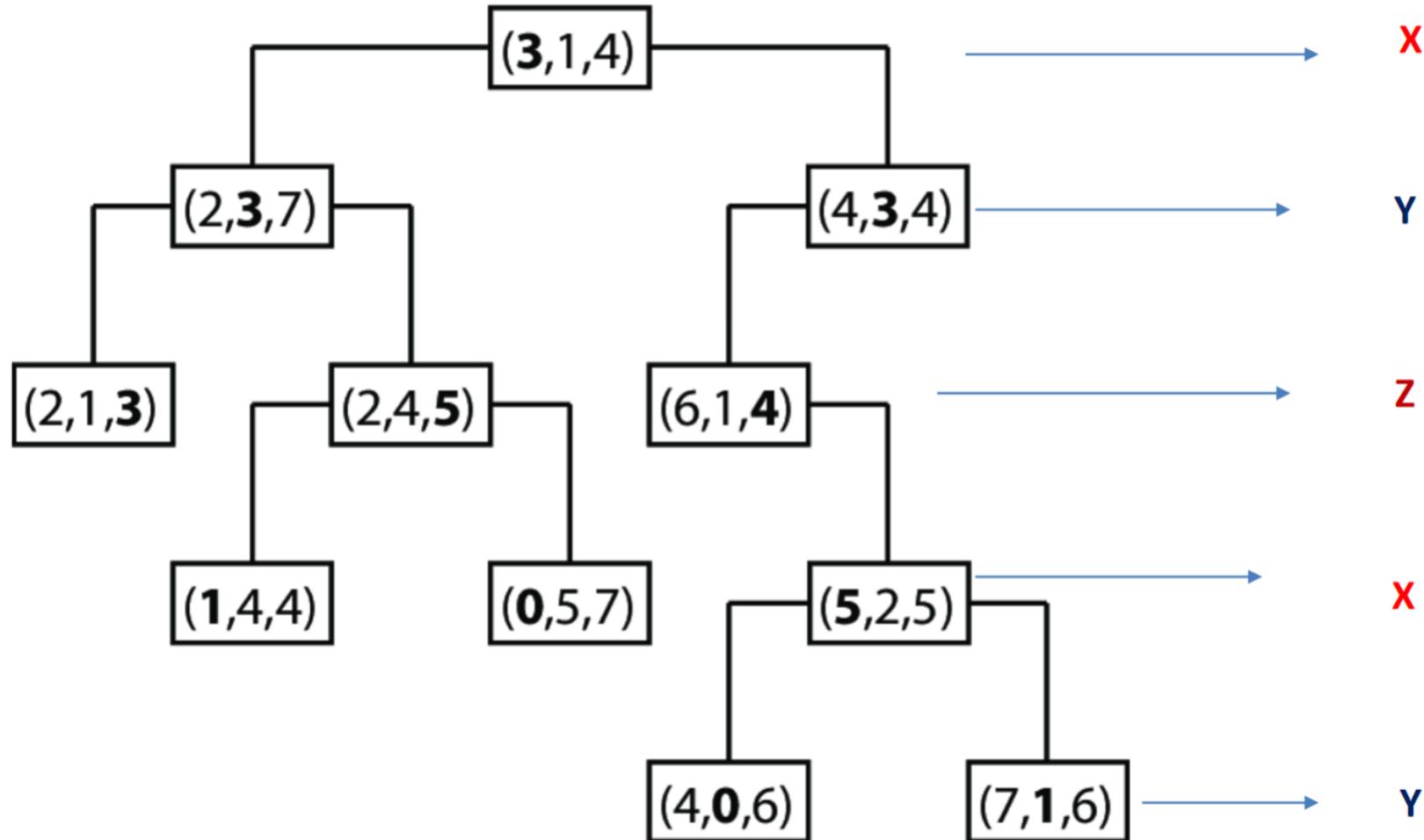


# K-d Trees – Insertion Algorithm

```
insert(Point x, KDNode t, int cd) {  
    if t == null  
        t = new KDNode(x)  
    else if (x == t.data)  
        // error! duplicate  
    else if (x[cd] < t.data[cd])  
        t.left = insert(x, t.left, (cd+1) % DIM)  
    else  
        t.right = insert(x, t.right, (cd+1) % DIM)  
    return t  
}
```

*Time Complexity for insertion into a K-d tree :  $O(\log n)$  or in worst case  $O(n)$*

# K-d Tree with 3 Dimensions

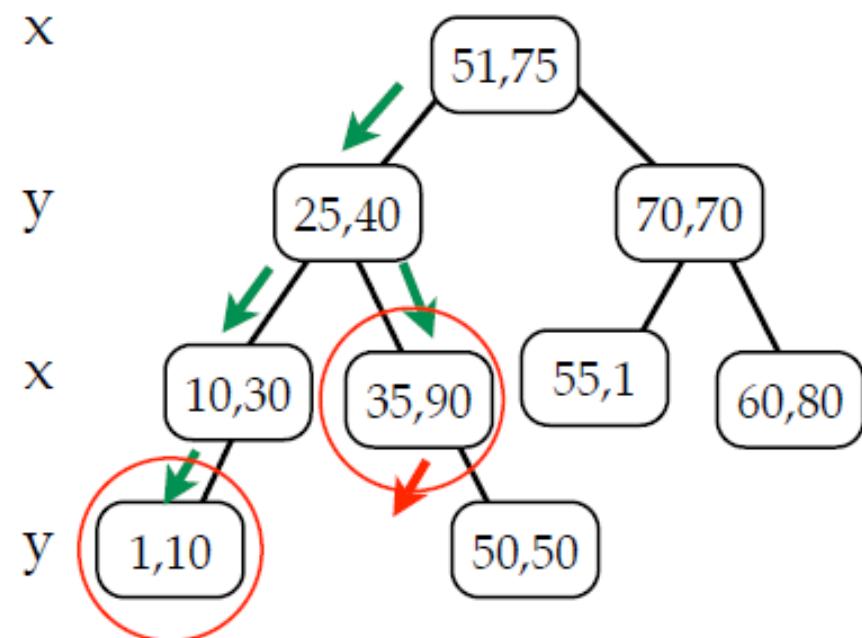
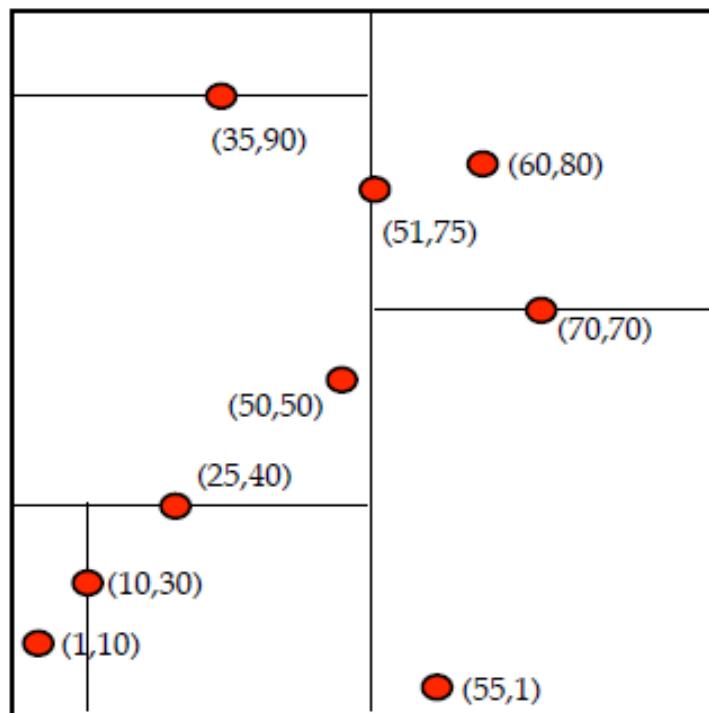


# FindMin in k-d-trees

- FindMin( $d$ ): find the point with the smallest value in the  $d^{\text{th}}$  dimension.
- Recursively traverse the tree!
- If  $\text{cutdim}(\text{current\_node}) = d$ , then
  - the minimum can't be in the right subtree, so recurse on just the left subtree.
  - if no left subtree, then current node is the min for tree rooted at this node.
- If  $\text{cutdim}(\text{current\_node}) \neq d$ , then
  - minimum could be in *either* subtree, so recurse on both subtrees,
  - or current node may also be minimum. So we take minimum of three and return.

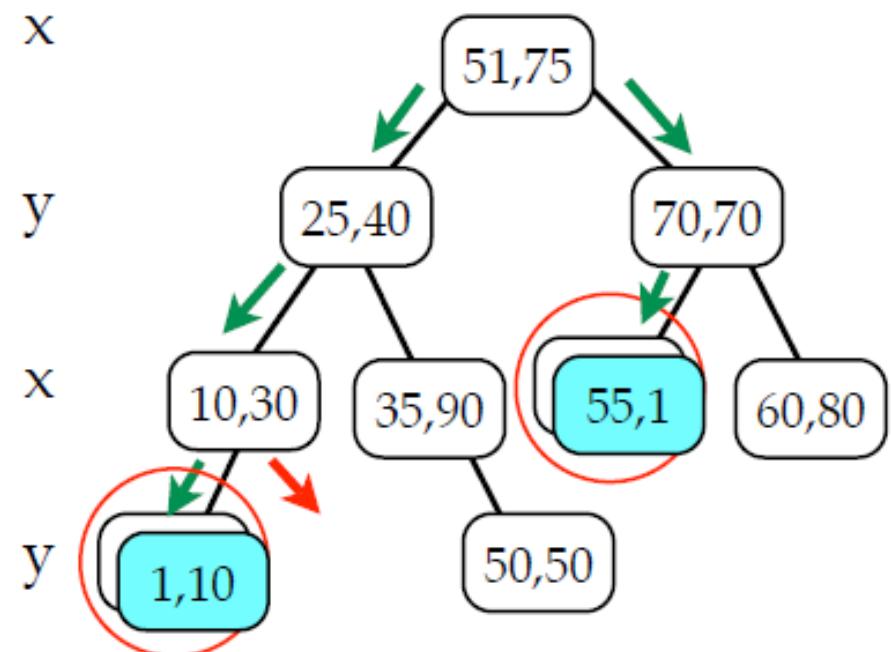
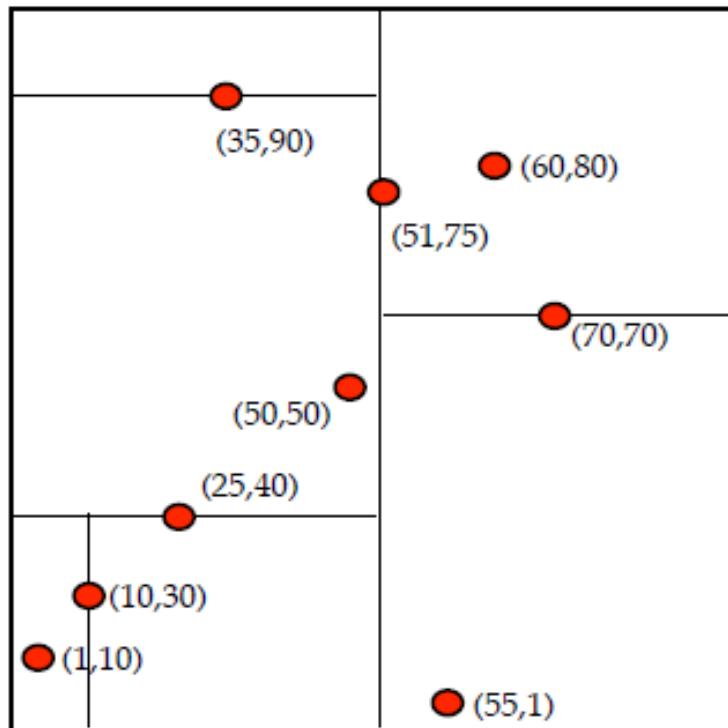
# FindMin in k-d Trees

FindMin( $x$ -dimension):



# FindMin in k-d Trees

FindMin(y-dimension):



# FindMin in k-d Trees Algorithm

```
Point findmin(Node T, int dim, int cd):
    // empty tree
    if T == NULL: return NULL

    // T splits on the dimension we're searching
    // => only visit left subtree
    if cd == dim:
        if t.left == NULL: return t.data
        else return findmin(T.left, dim, (cd+1)%DIM)

    // T splits on a different dimension
    // => have to search both subtrees
    else:
        return minimum(
            findmin(T.left, dim, (cd+1)%DIM),
            findmin(T.right, dim, (cd+1)%DIM)
            T.data
        )
```

# K-d Trees – Deletion

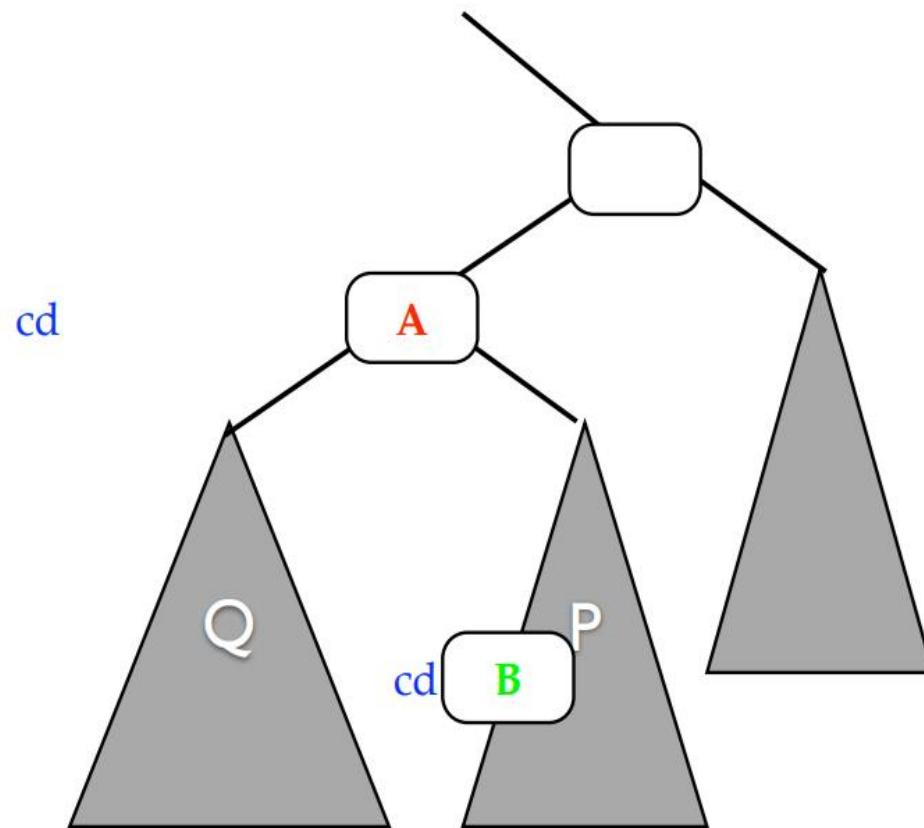
- Deletion from a k-d tree has many cases (just as in BST). Lets look at each of them.

Want to delete node **A**.

Assume cutting dimension of **A** is **cd**

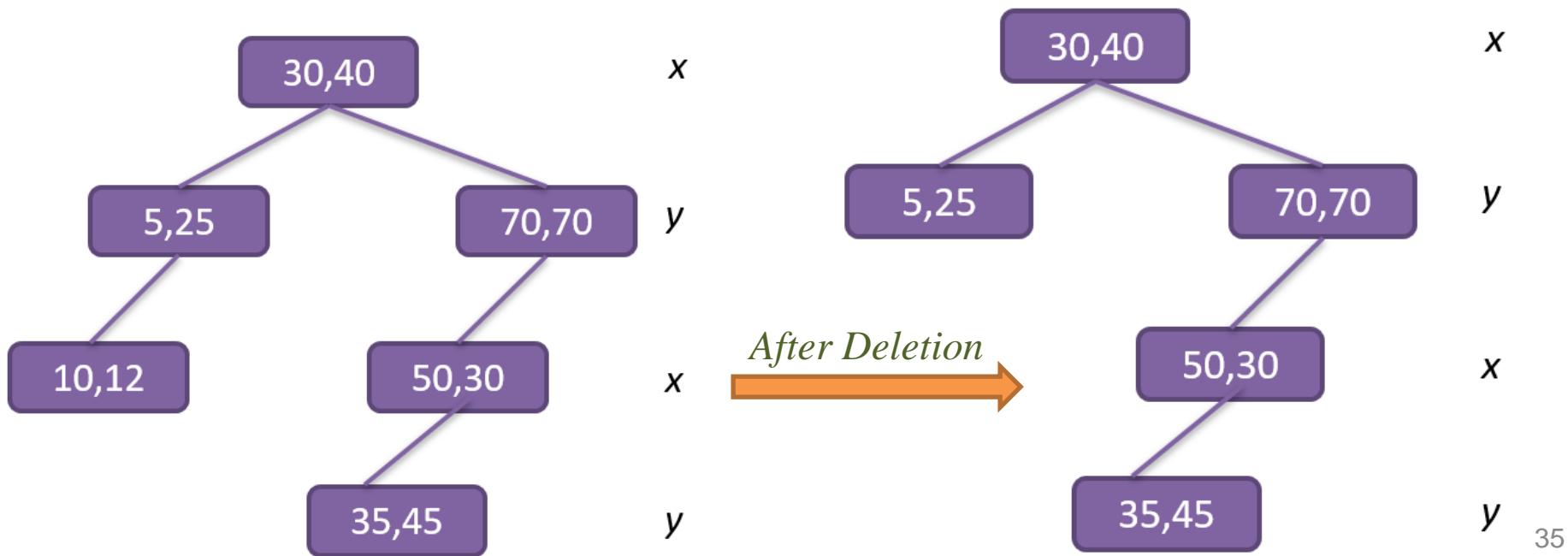
In BST, we'd  
*findmin(A.right)*.

Here, we have to  
*findmin(A.right, cd)*



# K-d Tree – Deletion – Case 1

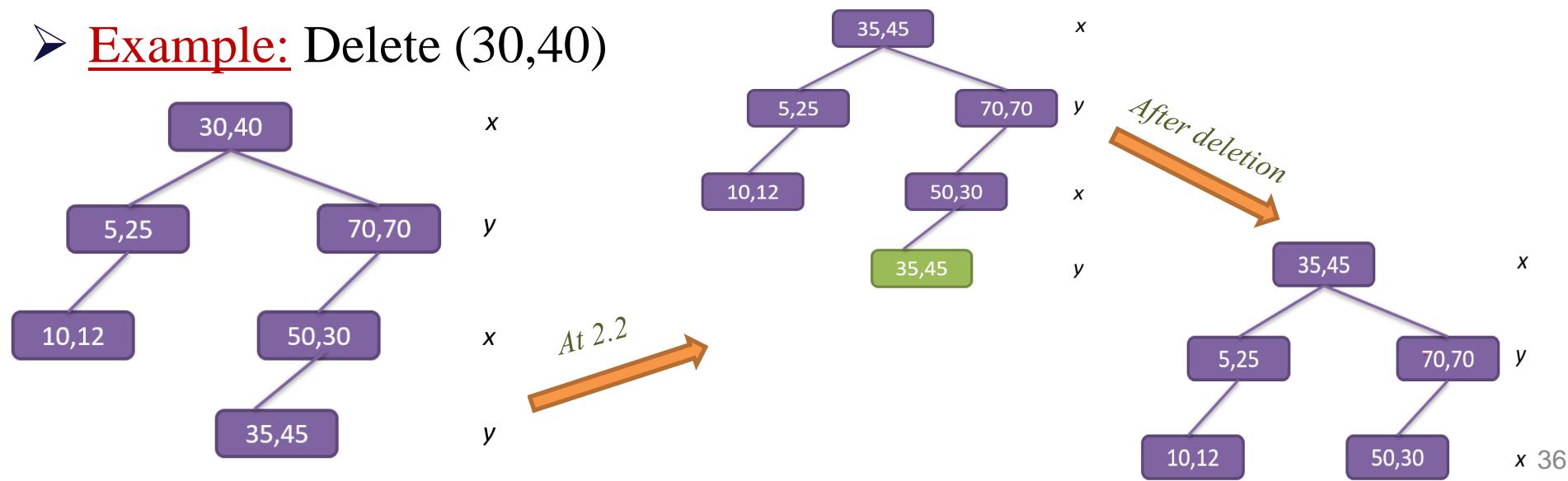
- Case 1: If the current node contains the point to be deleted and the node to be deleted is a leaf node, just delete it.
- Example: Delete (10,12)



# K-d Tree – Deletion – Case 2

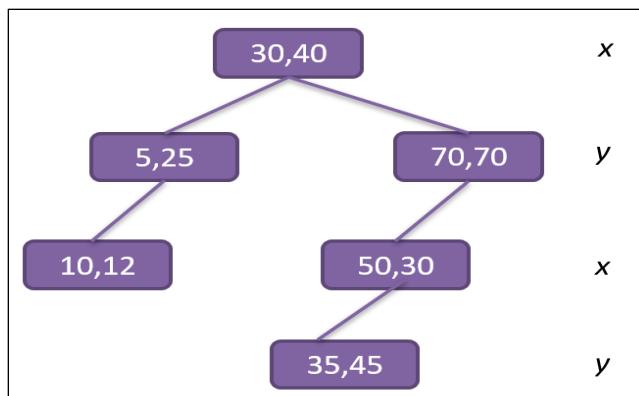
- Case 2: If the current node contains the point to be deleted and the node to be deleted has a right subtree, then:
  - 2.1 Find the minimum of current node's dimension in **right** subtree i.e. `FindMin(currentNode's dim)` in right subtree.
  - 2.2 Replace the node with the node found in 2.1 and recursively delete minimum in right subtree.

- Example: Delete (30,40)

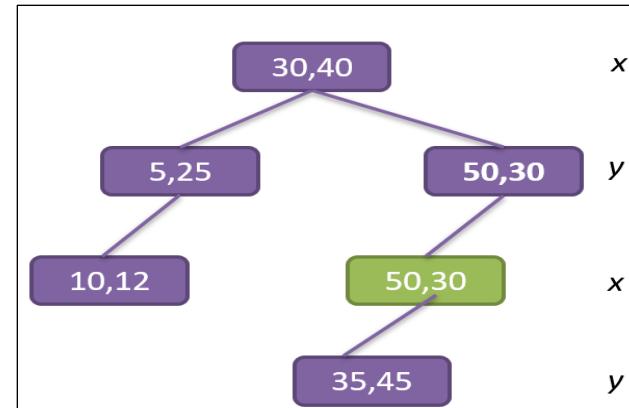


# K-d Tree – Deletion – Case 3

- Case 3: If the current node contains the point to be deleted and the node to be deleted has a left subtree (no right subtree), then:
  - 3.1 Find the minimum of current node's dimension in **left** subtree i.e. `FindMin(currentNode's dim)` in left subtree.
  - 3.2 Replace the node with the node found in 3.1 and recursively delete minimum in left subtree.
  - 3.3 Make new left subtree as right child of current node [*Caution!*]
- Example: Delete (70,70)

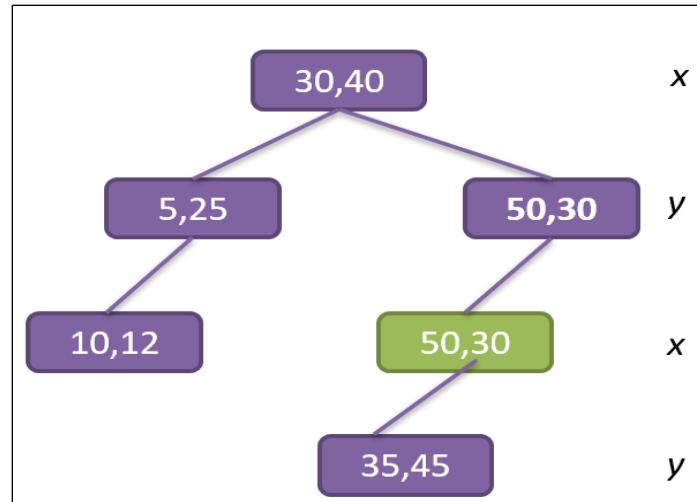


At 3.2,  
Min is (50,30)

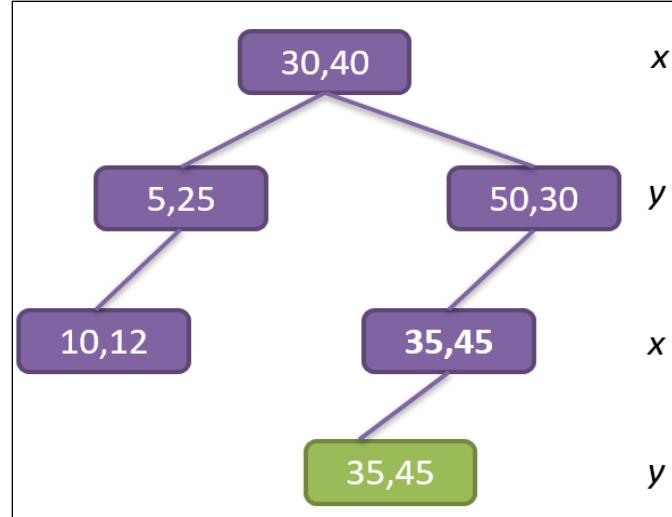


# K-d Tree – Deletion – Case 3

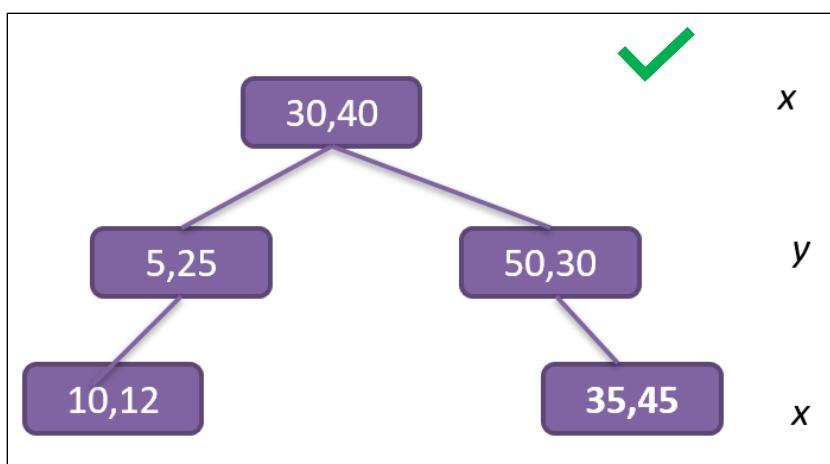
Recursively delete in left-subtree !!



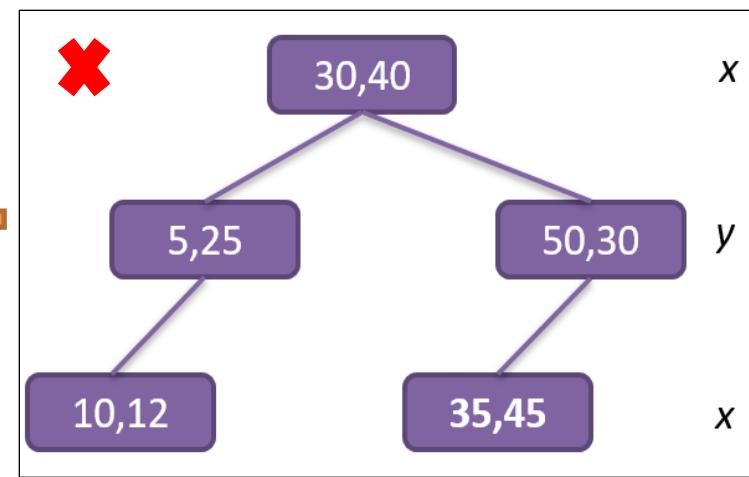
At 3.2,  
Min is (35,45)



After  
deletion



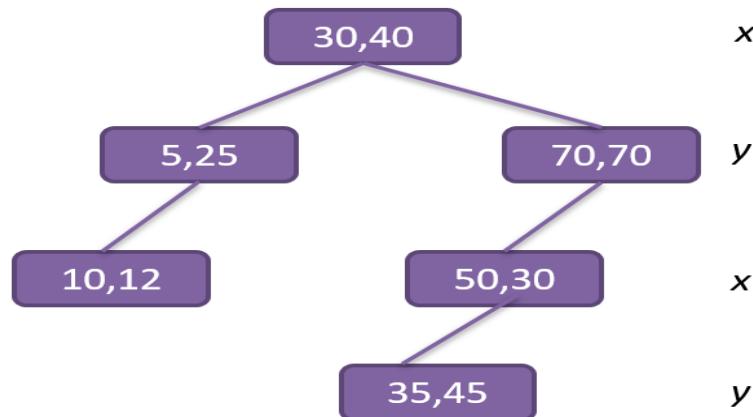
After  
3.3



# K-d Tree – Deletion – Case 4

Case 4: If the current node does not contain the point to be deleted, then:

- 4.1 If the node to be deleted is smaller than the current node on current dimension, recur for the left subtree. Otherwise, recur for the right subtree.
- 4.2 Eventually after 4.1, we will reach any of the other three cases. Use that case and perform deletion!
- Example: Delete (70,70) and current node is (30,40)



*Here,  $70 > 30$ , hence recur on right of (30,40) to find (70,70). Then, perform Case 3.*

# K-d Tree – Deletion Algorithm

```
Point delete(Point x, Node T, int cd):
    if T == NULL: error point not found!
    next_cd = (cd+1)%DIM

    // This is the point to delete:
    if x = T.data:
        // use min(cd) from right subtree:
        if t.right != NULL:
            t.data = findmin(T.right, cd, next_cd)
            t.right = delete(t.data, t.right, next_cd)
        // swap subtrees and use min(cd) from new right:
        else if T.left != NULL:
            t.data = findmin(T.left, cd, next_cd)
            t.right = delete(t.data, t.left, next_cd)
        else
            t = null      // we're a leaf: just remove

    // this is not the point, so search for it:
    else if x[cd] < t.data[cd]:
        t.left = delete(x, t.left, next_cd)
    else
        t.right = delete(x, t.right, next_cd)

return t
```

# K-d Tree Applications

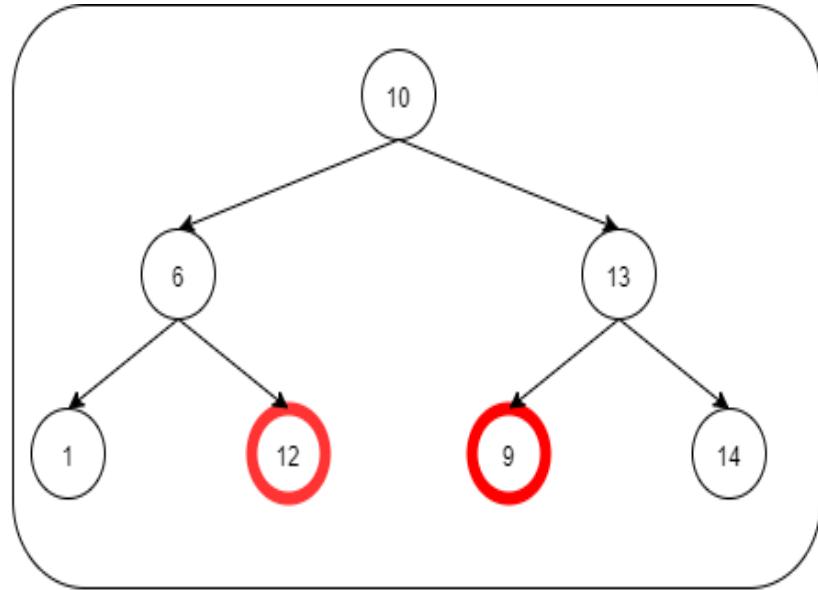
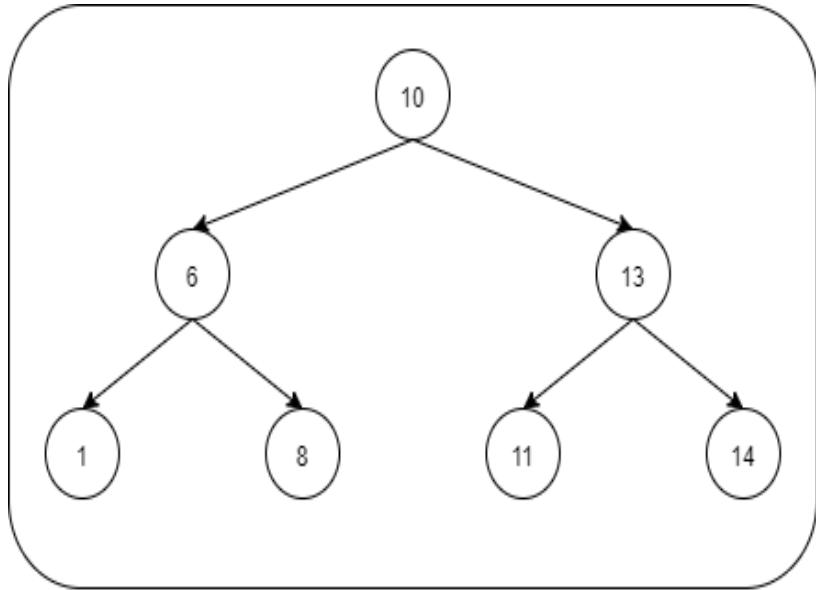
- Nearest Neighbor [Explore & Self-Study! Will upload some slides as appendix]
- Range Queries
- Search Engines
- Image Search
- Machine learning
- ...
- ...

# Recap [Binary Search Tree – Property]



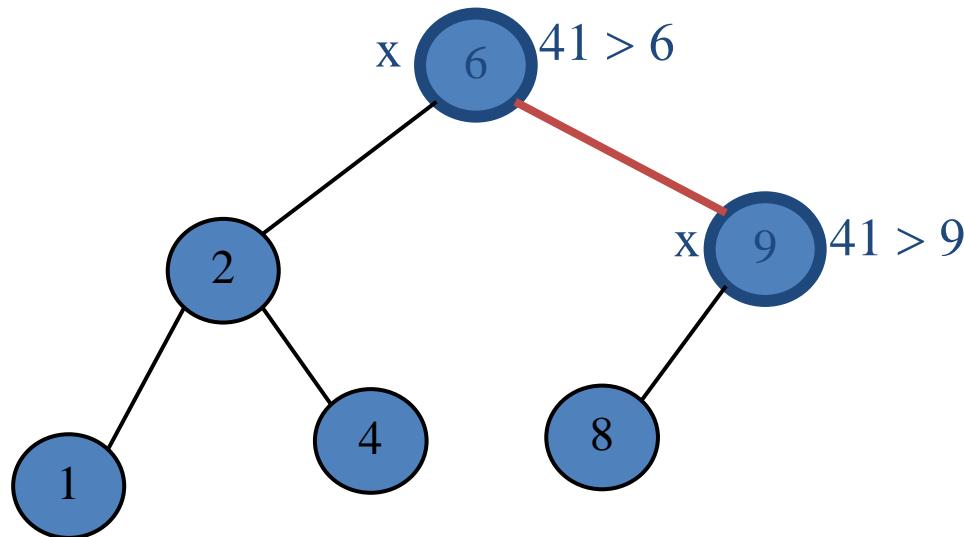
- In binary search tree, *all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data.*
  - This is called binary search tree property, note that, this property should be satisfied at **every node** in the tree.
    - The left subtree of a node, contains only nodes with keys **less** than the nodes key.
    - The right subtree of a node, contains only nodes with keys **greater** than the nodes key.
    - Both the left and right subtrees must also be binary search trees.

# Binary Search Trees

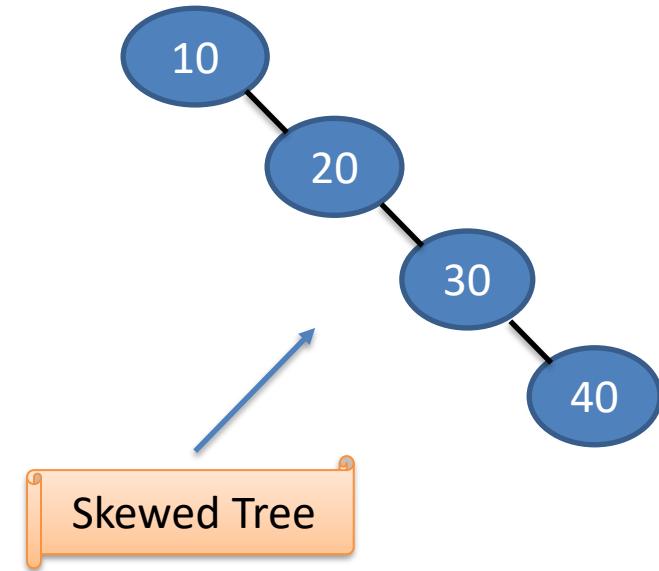


# Find in BST

Search element 41



Search element 41



*What is the time complexity of finding the element in binary search tree ?  
O(n) Why??*

# Applications of BST

BSTs are used for a lot of applications due to its ordered structure :

- BSTs are used for indexing and multi-level indexing in DB.
- They are also helpful to implement various searching algorithms.
- It is helpful in maintaining a sorted stream of data.
- TreeMap and TreeSet data structures offered by high level languages are internally implemented using self-balancing BSTs

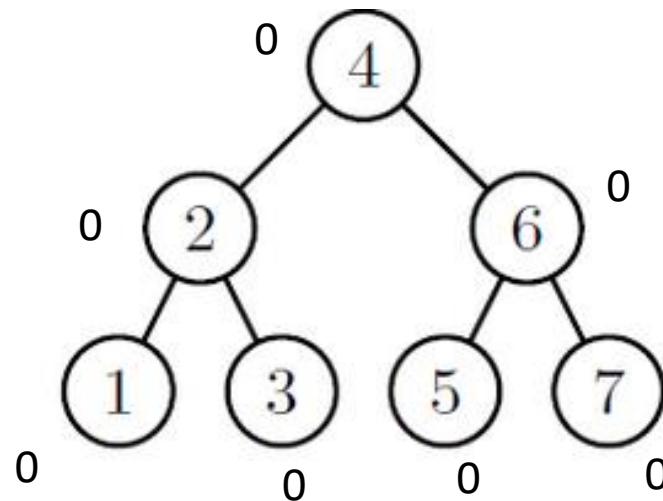
# Balanced Binary Search Trees

- We have seen different trees whose worst case complexity is  $O(n)$  ( including BST ).
- Where  $n$  is the number of nodes in the tree. This happens when the trees are skew trees ☹
- Will try to reduce this worst case complexity to  $O(\log n)$  by *imposing restrictions on the heights*.
- In general, the height balanced trees, are represented with **HB(K)**, where K is the difference between left subtree height and right subtree height.
- Sometimes K is called *balance factor*.

Balance Factor = Height(Left subtree) – Height(right subtree)

# Balanced Binary Search Trees

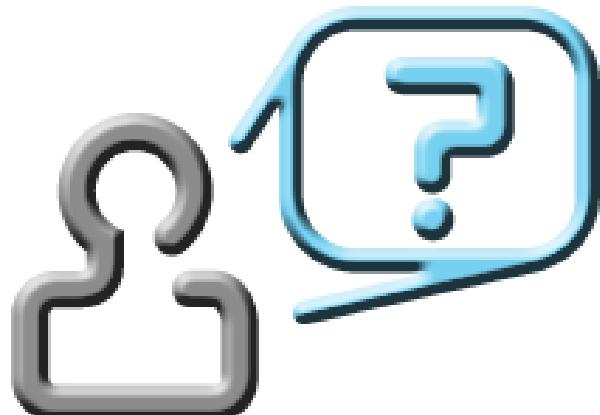
- If HB(K), if K = 0( if balance factor is zero), then we call such binary search trees as full balanced binary search trees.
- That means, in HB(0) binary search tree, the difference between left subtree and right subtree height should be *at most zero*.



*Note: Numbers next to the node indicate the balance factor*

# AVL (Adelson-Velsky and Landis) Trees

- In a height balanced tree HB(K), if  $K = 1$ (if balance factor is one), such binary search tree is called an AVL tree.
- That means an AVL tree is a binary search tree with a balance condition the difference between left subtree height and right subtree height is **at most 1**
- Hence, in an AVL tree, each node is associated with a balance factor which can be :
  - **Balance factor of 0** : If the height of the left subtree and height of the right subtree are same.
  - **Balance factor of 1** : If the height of the left subtree is 1 more than the height of the right subtree.
  - **Balance factor of -1** : If the height of the left subtree is 1 less than the height of the right subtree.



*We will rotate around AVL trees in the next class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : CC BY-NC-SA 4.0





**BITS** Pilani

Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

**DSECLZG519**

Parthasarathy





# Contact Session #10

## DSECLZG519 – AVL Trees

# Agenda for Session # 10

- Balance Factor ?
- AVL Trees
  - Motivation for AVL ?
  - Types of Rotations
    - LL
    - RR
    - LR
    - RL
  - Operations on AVL
    - Insertion
    - Removal
- Analysis of AVL tree
- Exercises

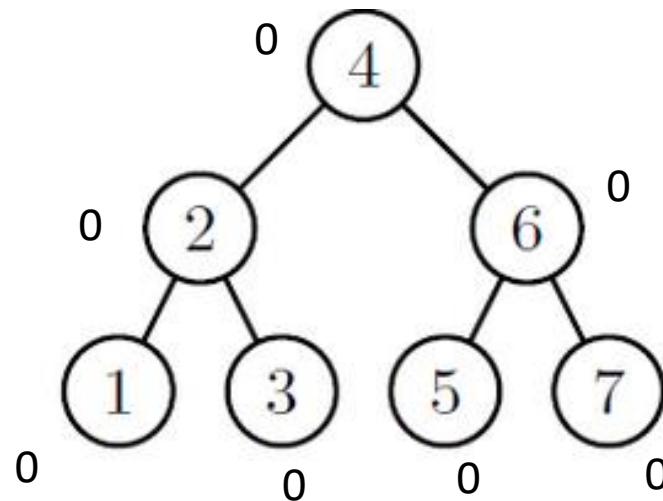
# Balanced Binary Search Trees

- We have seen different trees whose worst case complexity is  $O(n)$  ( including BST ).
- Where  $n$  is the number of nodes in the tree. This happens when the trees are skew trees ☹
- Will try to reduce this worst case complexity to  $O(\log n)$  by *imposing restrictions on the heights*.
- In general, the height balanced trees, are represented with **HB(K)**, where K is the difference between left subtree height and right subtree height.
- Sometimes K is called *balance factor*.

Balance Factor = Height(Left subtree) – Height(right subtree)

# Balanced Binary Search Trees

- If HB(K), if K = 0( if balance factor is zero), then we call such binary search trees as full balanced binary search trees.
- That means, in HB(0) binary search tree, the difference between left subtree and right subtree height should be *at most zero*.



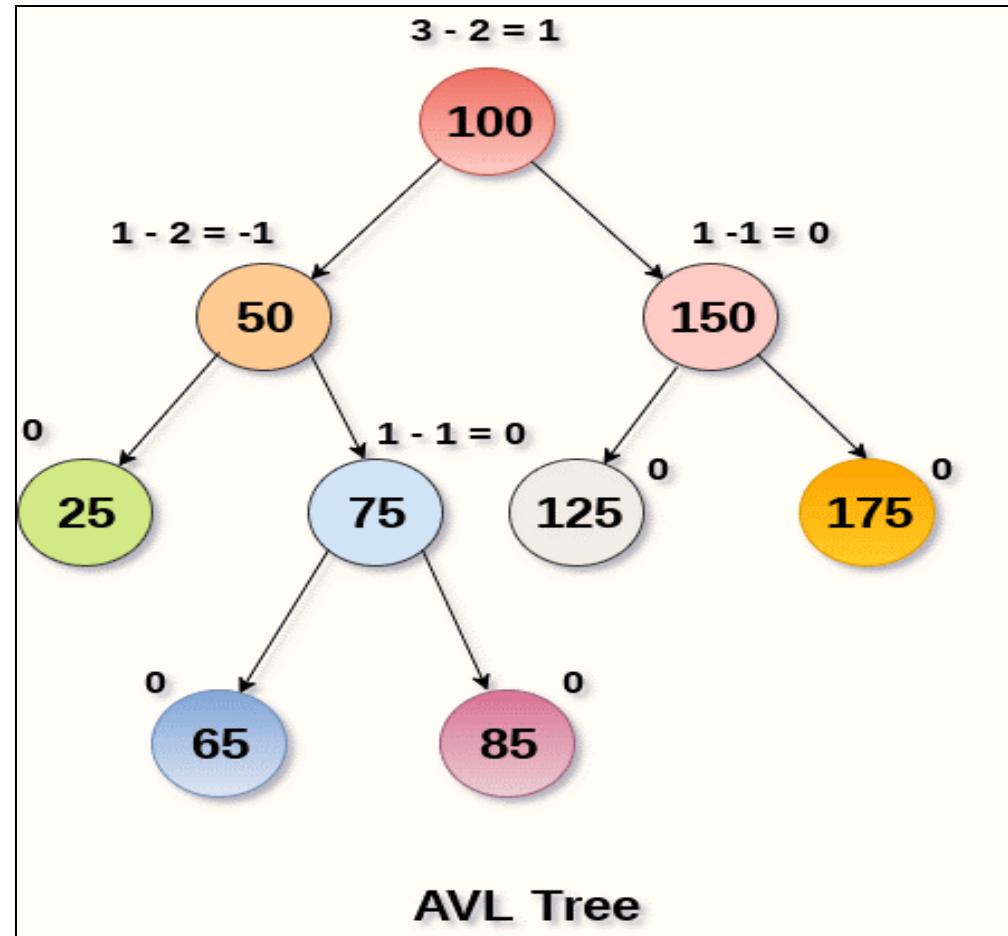
*Note: Numbers next to the node indicate the balance factor*

# AVL (Adelson-Velsky and Landis) Trees

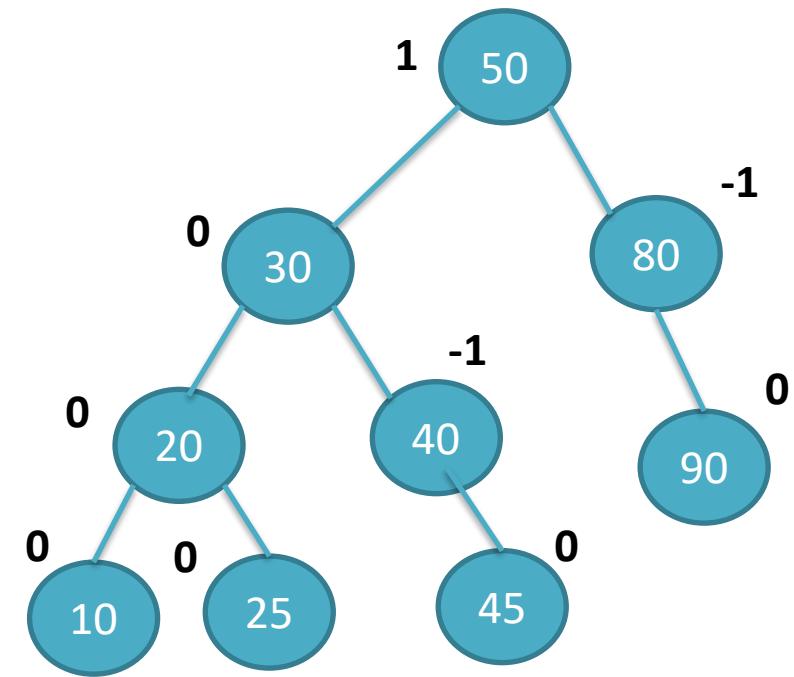
- In a height balanced tree HB(K), if  $K = 1$ (if balance factor is one), such binary search tree is called an AVL tree.
- That means an AVL tree is a binary search tree with a balance condition the difference between left subtree height and right subtree height is **at most 1**
- Hence, in an AVL tree, each node is associated with a balance factor which can be :
  - **Balance factor of 0** : If the height of the left subtree and height of the right subtree are same.
  - **Balance factor of 1** : If the height of the left subtree is 1 more than the height of the right subtree.
  - **Balance factor of -1** : If the height of the left subtree is 1 less than the height of the right subtree.

# Properties of AVL Trees

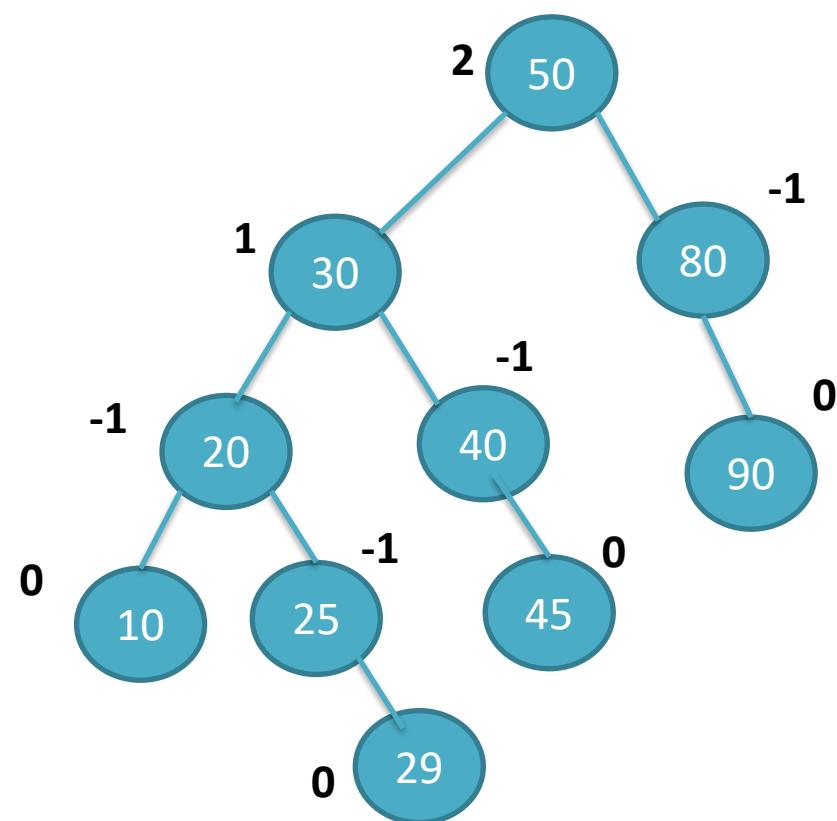
- It is a binary search tree, and
- For any node X, the height of left subtree of X and height of right subtree of X differ by ***at most 1***.
- i.e. An AVL tree is a height balanced BST !



# Examples



AVL  
Tree

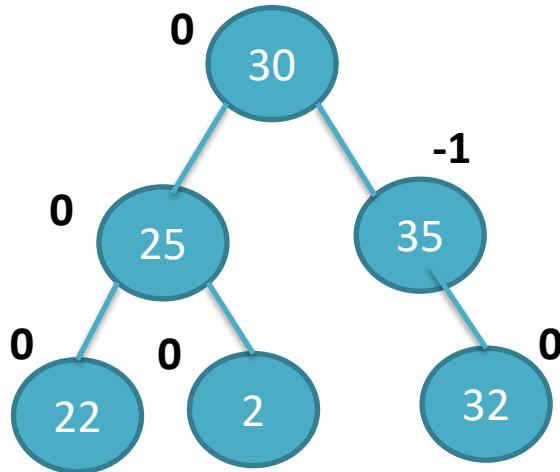


AVL  
Tree



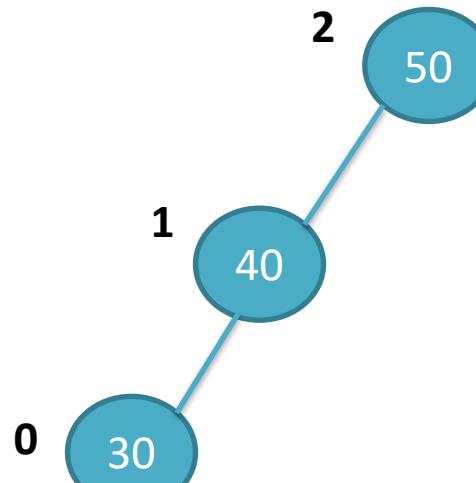
*Reason : Balance  
factor of 50 is 2*

# Examples



*Reason : This is not a BST !*

AVL  
Tree 



*Reason : Balance factor of 50 is 2*

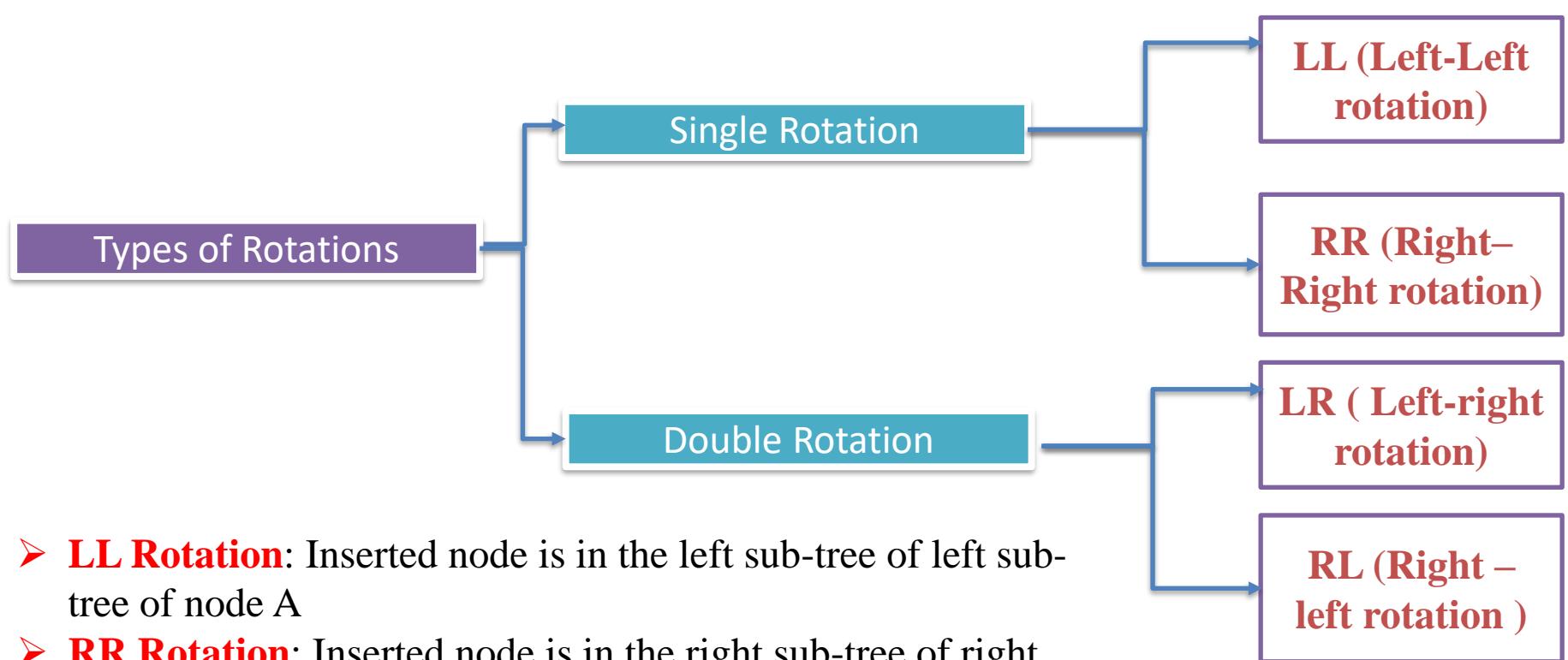
AVL  
Tree 

# Creating an AVL Tree

---

- Creating an AVL tree is nothing but repeated insertion of elements / items into binary search tree [BST]. But, after adding each element / item into the tree, a check must be made to ensure that the balance factor of **each node** is -1 or 0 or 1.
- If so, then the tree is balanced and the tree is a AVL Tree.
- If the balance factor of *any* node is not either -1 or 0 or 1, it is necessary to rearrange the nodes in the tree so that the balance factor of all the nodes will either be -1 or 0 or 1.
- This rearrangement is achieved using “*rotations*”.
- The different types of rotations are LL , RR , LR and RL.

# Rotations in AVL Tree



- **LL Rotation:** Inserted node is in the left sub-tree of left sub-tree of node A
- **RR Rotation:** Inserted node is in the right sub-tree of right sub-tree of node A
- **LR Rotation:** Inserted node is in the right sub-tree of left sub-tree of node A
- **RL Rotation:** Inserted node is in the left sub-tree of right sub-tree of node A

# Which rotation is required ?

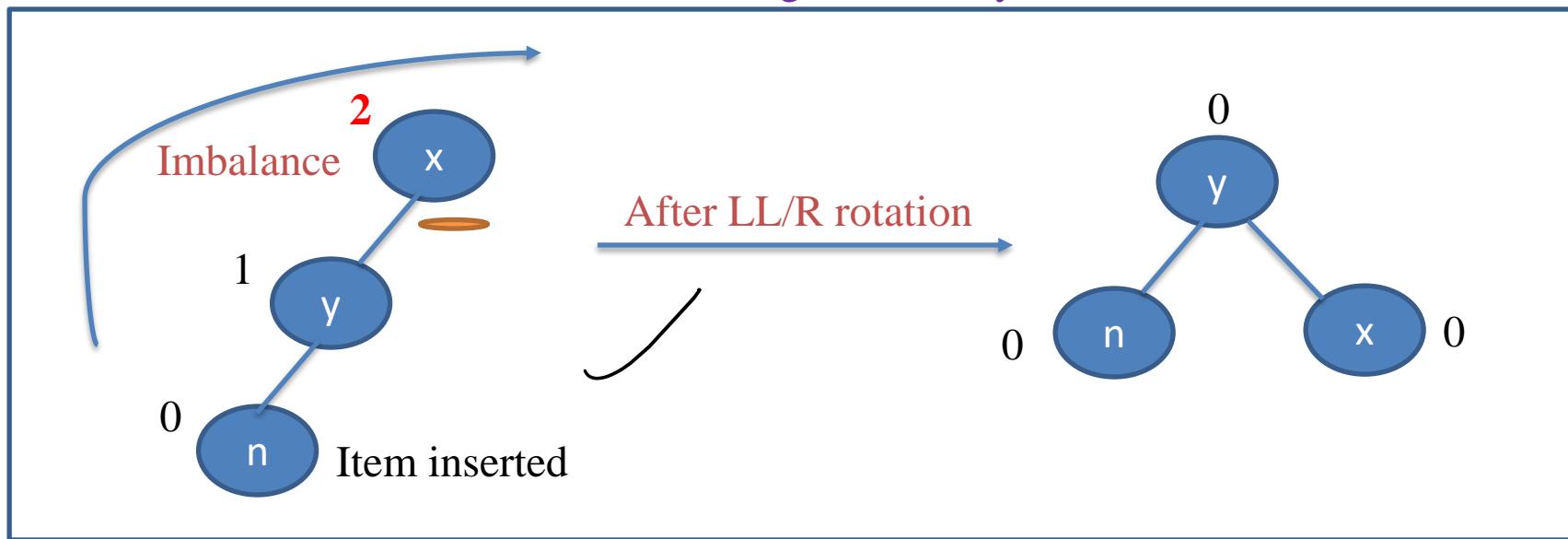
---

- After inserting a node, trace backward towards the root and compute the balance factors along the path.
- If we find a node whose balance factor is other than -1 or 0 or 1, then rotation is required. But how do we find which of the four rotations is required ? We follow the below process :
  - Step 1: Identify the first node where balance factor is **not** -1/0/1 in the path.
  - Step 2: Identify two other nodes below the node where imbalance occurs in the path.
- If the three nodes identified after Step (1) and (2) are in a straight line, single rotation is required (Either LL/R-rotation or RR/L rotation).
- If the three nodes identified are **not** in a straight line, we will need double rotation (LR rotation or RL rotation).

# LL imbalance (R rotation) in AVL Tree

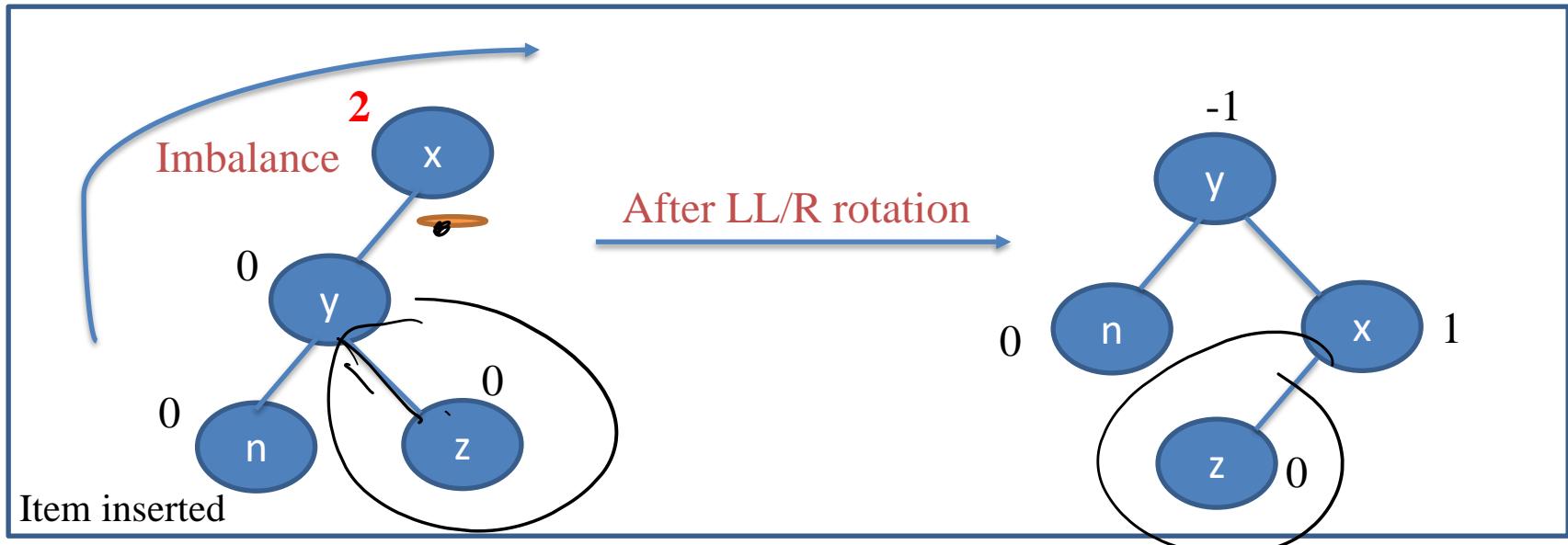
- If the balance factor of a node where the imbalance occurs is 2, then the tree is heavy towards left. So, this is called left-left imbalance and to balance it we rotate right.
- This is called as LL rotation or R-rotation. This is a **single** rotation and is in clock wise.

Case 1: No right child of y

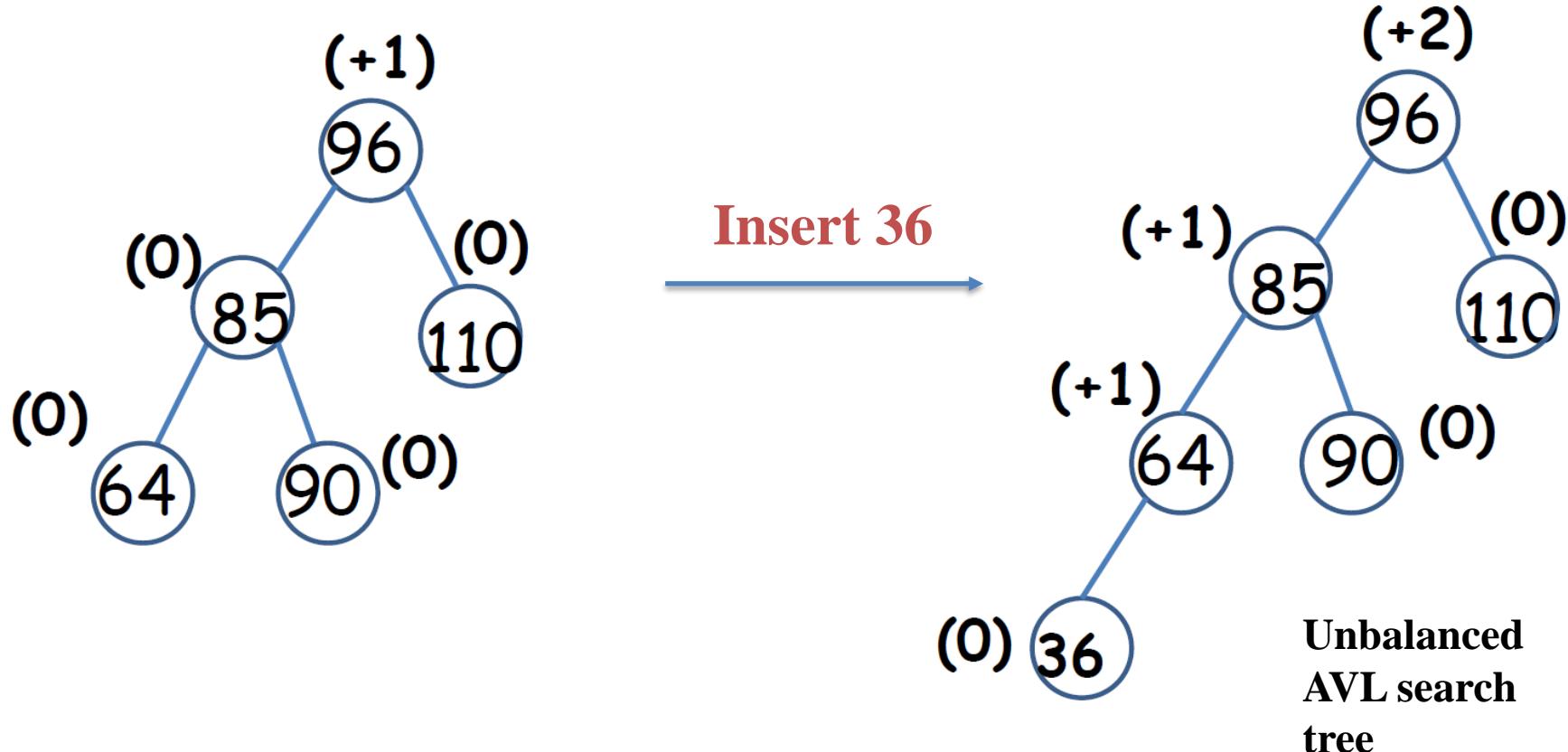


# LL imbalance (R rotation) in AVL Tree

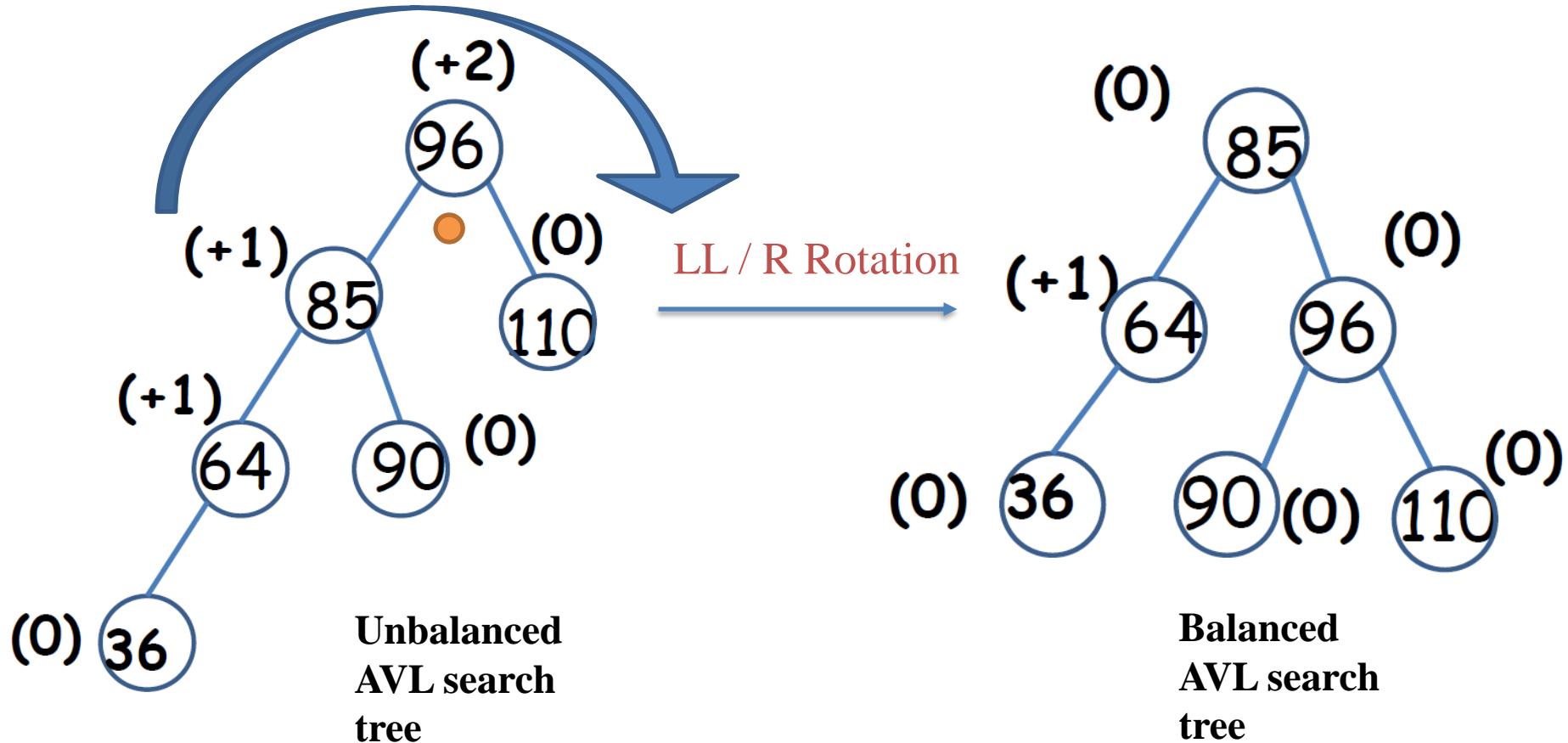
Case 2: There is a right child of y



# LL / R Rotation Example



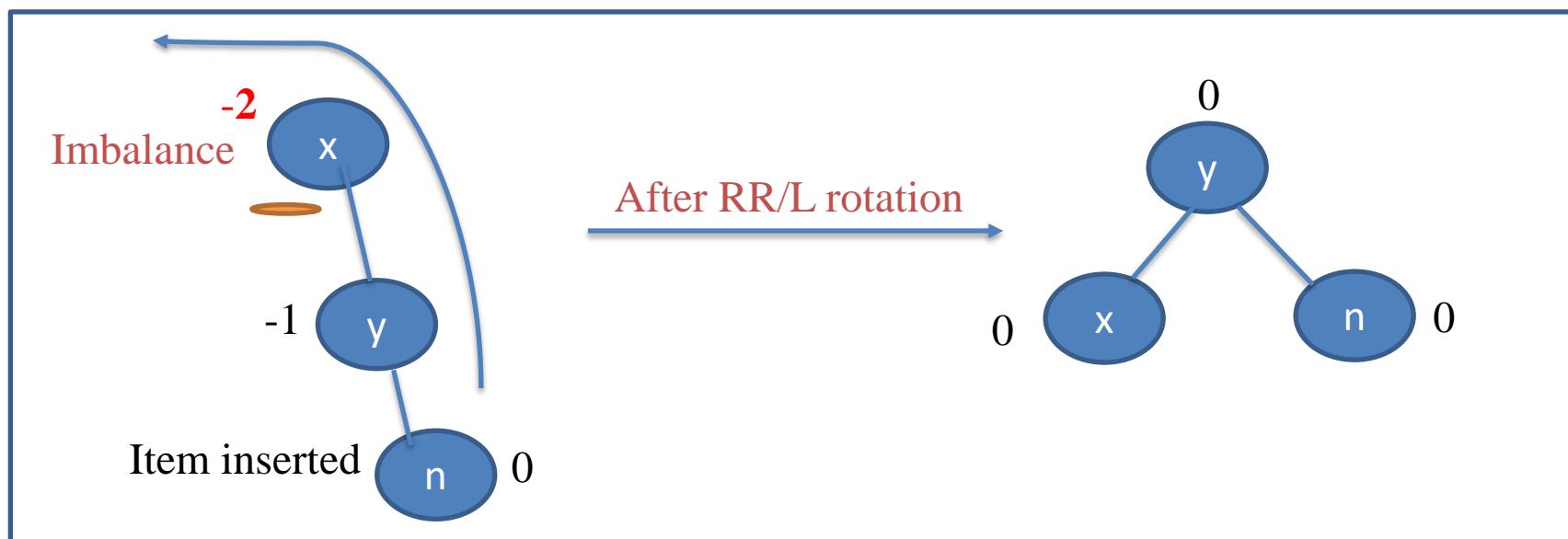
# LL / R Rotation Example



# RR Imbalance (L-rotation) in AVL Tree

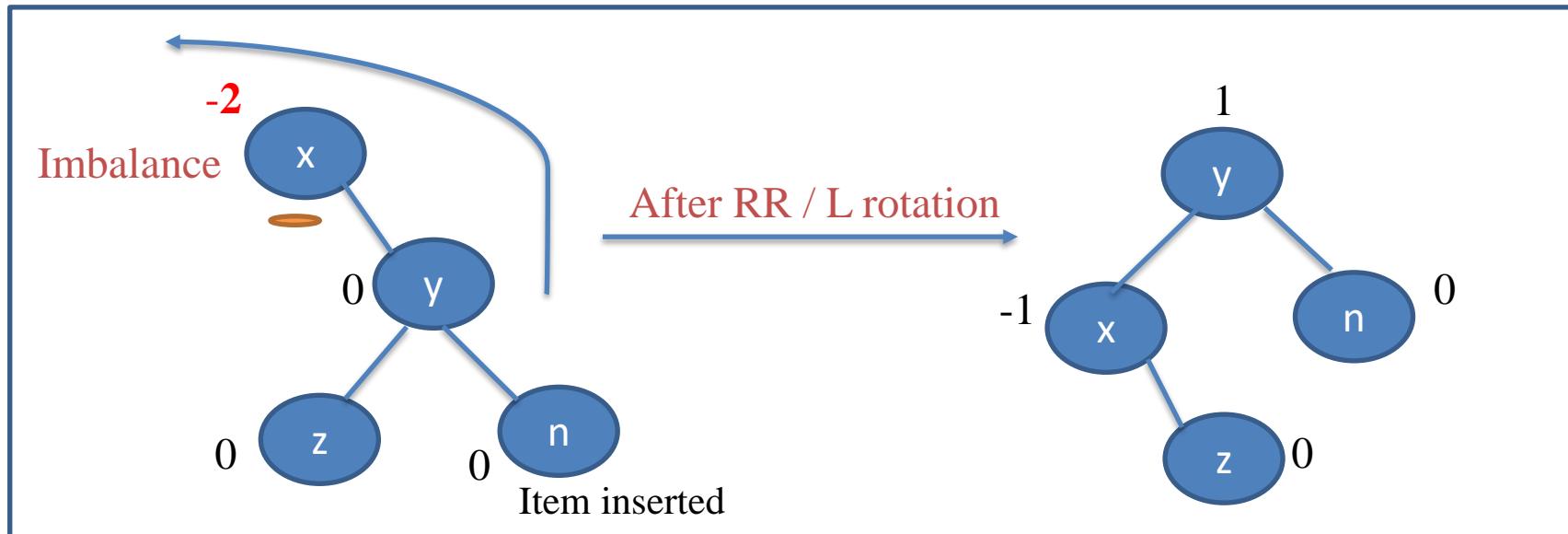
- If the balance factor of a node where imbalance occurs is **-2**, then the tree is heavy towards right. So, this is called right-right imbalance and to balance it we rotate left.
- This is called RR-rotation or L rotation. This is a **single** rotation and is anti-clock wise.

Case 1: No left child of y

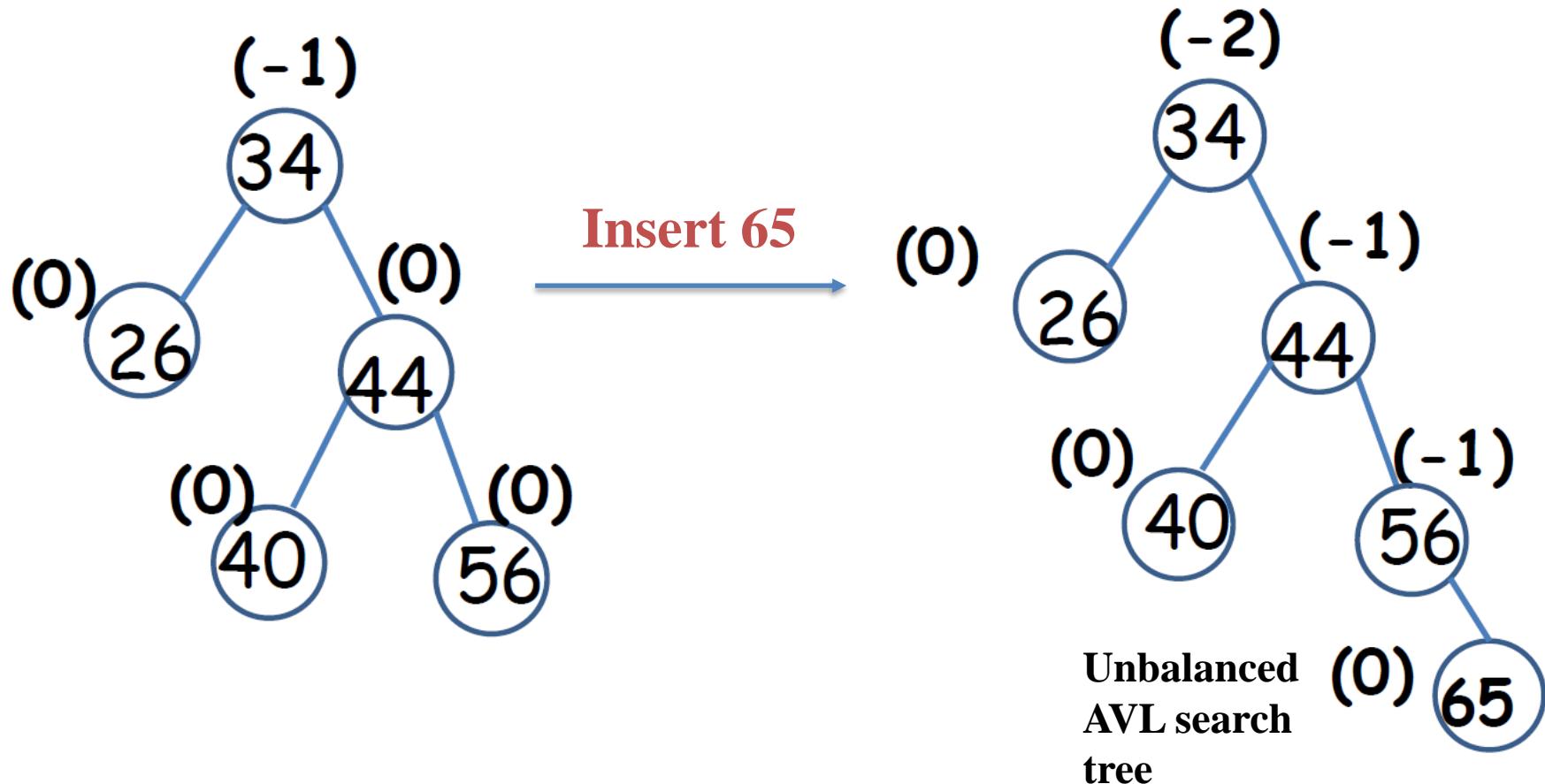


# RR Imbalance (L-rotation) in AVL Tree

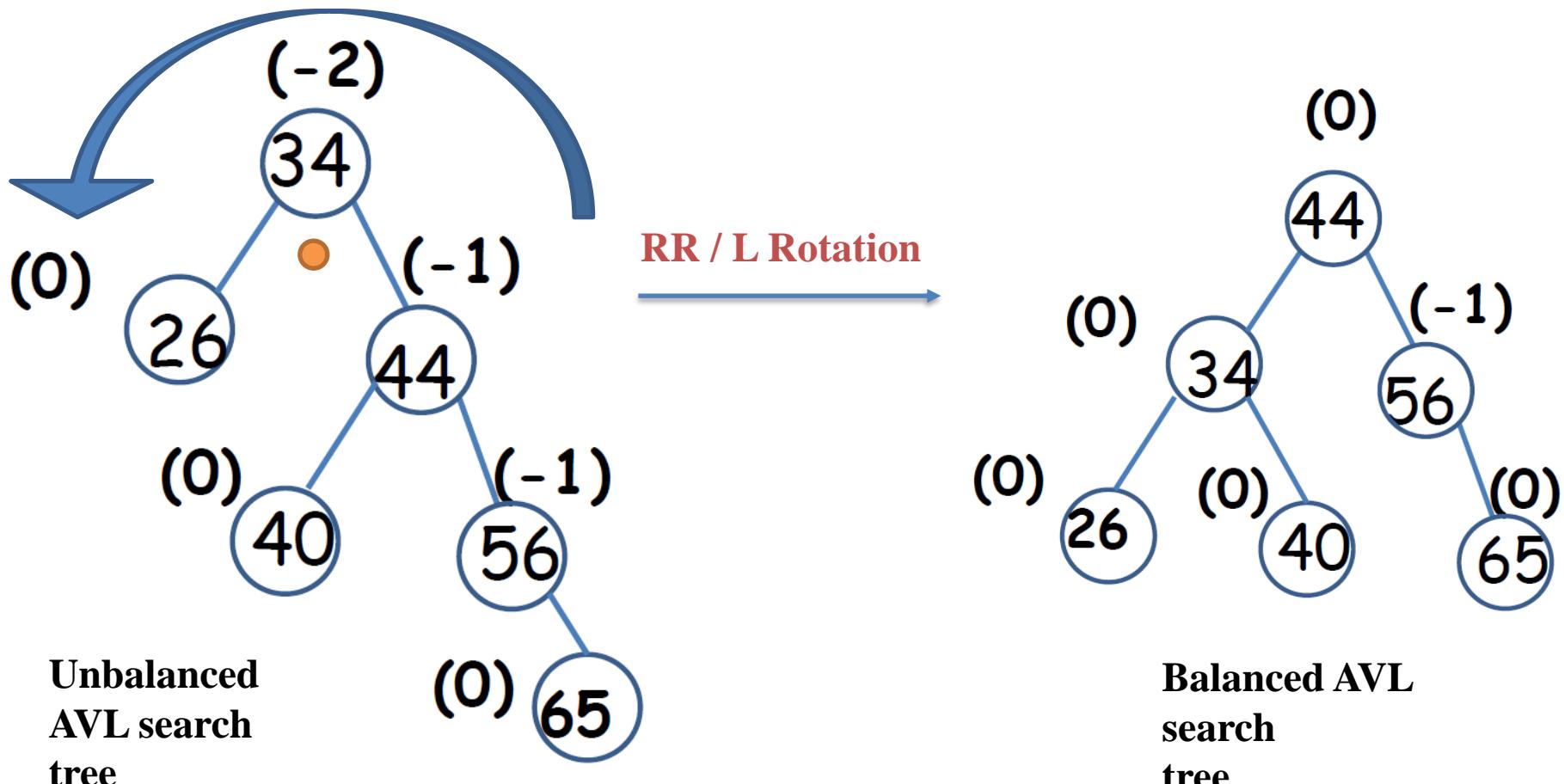
Case 2: There is a left child of y



# RR / L Rotation Example



# RR / L Rotation Example

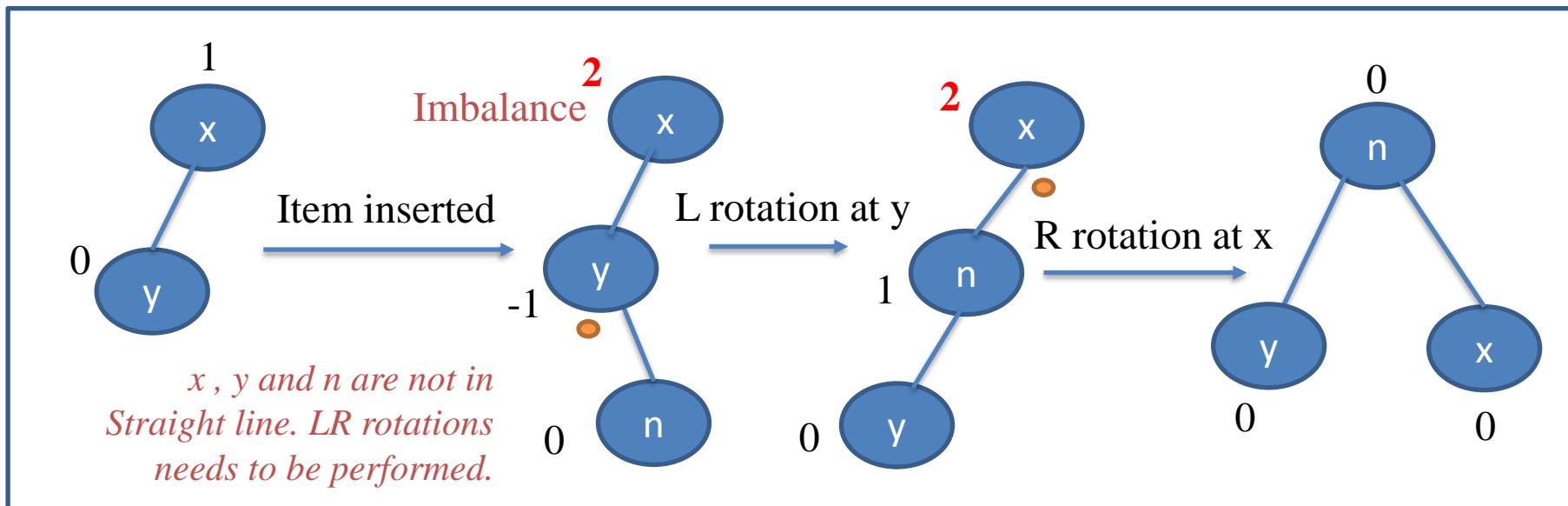


Unbalanced  
AVL search  
tree

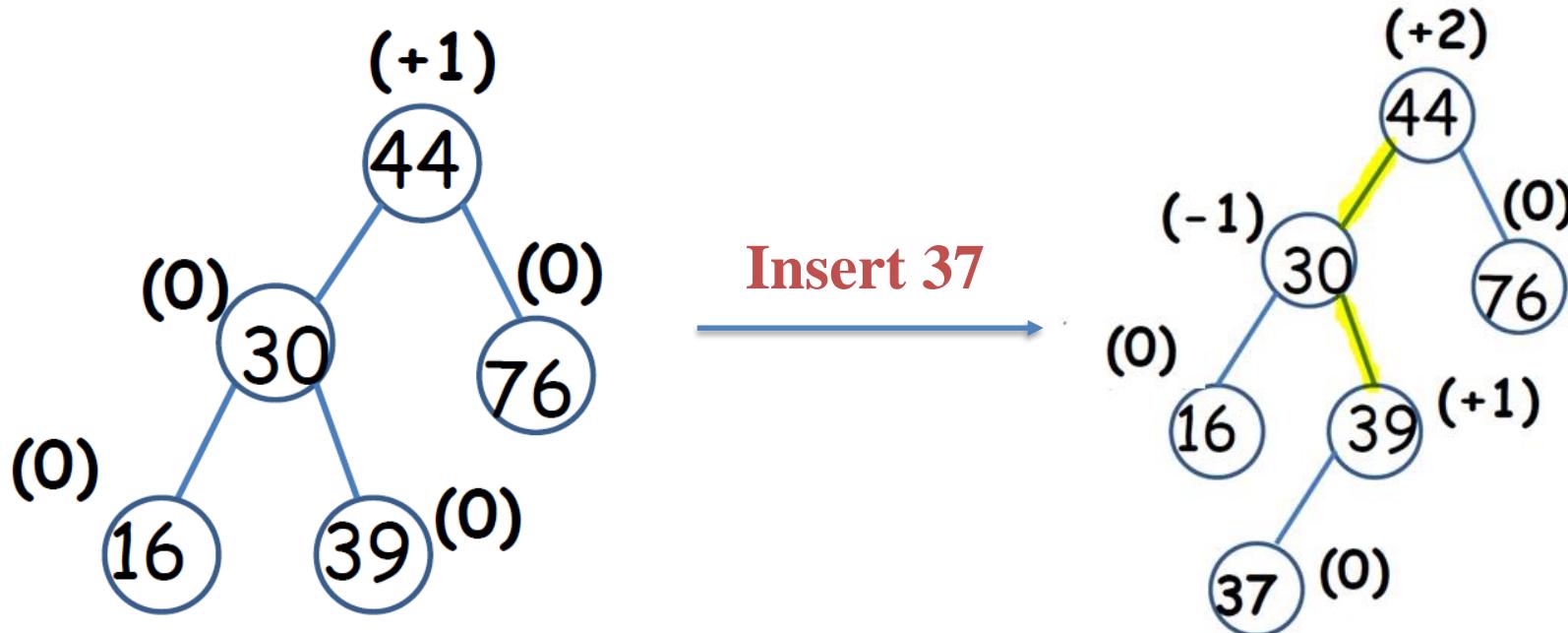
Balanced AVL  
search  
tree

# Left Right rotation (LR) in AVL Tree

- If the three nodes identified are ***not in a straight line***, then double rotation is required. A double-rotation is a combination of two single-rotations. If the order of imbalance is LR, then LR rotation is needed.
- The first step will be L rotation (anti) and then the R(clock) rotation.

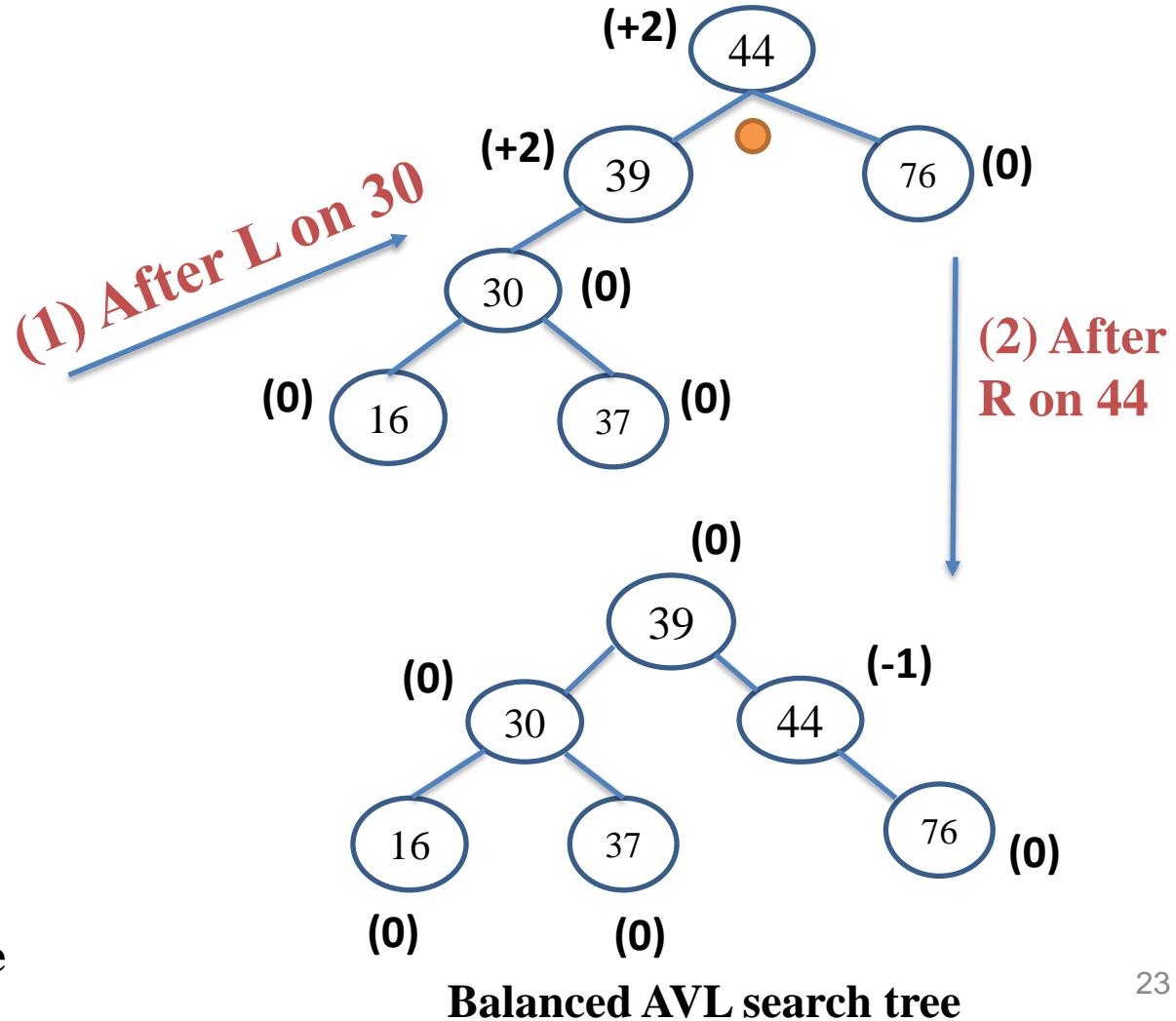
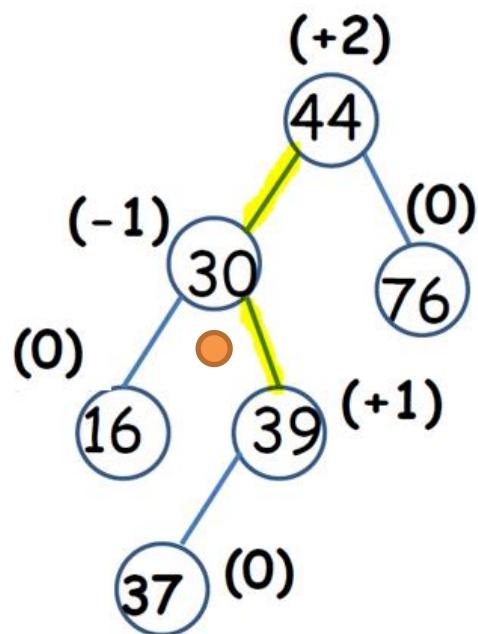


# (LR) rotation Example



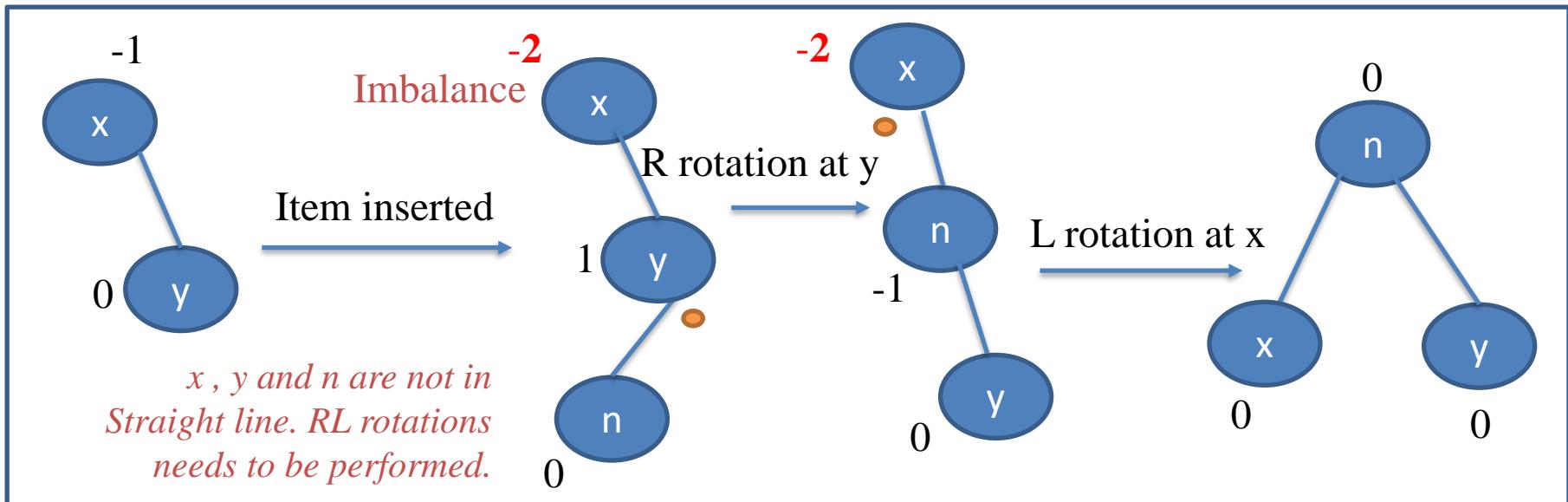
Unbalanced AVL search tree

# (LR) rotation Example

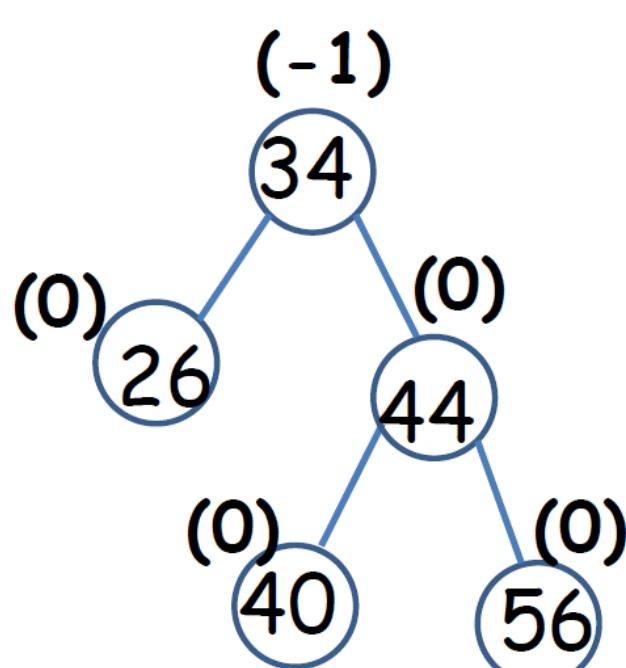


# Right Left rotation (RL) in AVL Tree

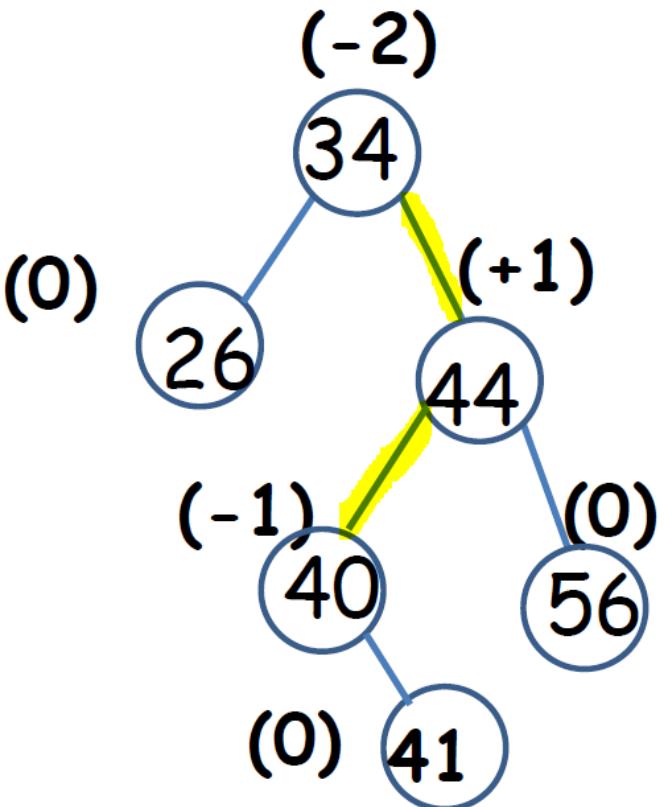
- If the three nodes identified are not in a straight line, then double rotation is required. A double-rotation is a combination of two single-rotations. If the order of imbalance is RL, then RL rotation is needed.
- The first step will be R rotation (clockwise) and then the L(anti-clock) rotation.



# (RL) rotation Example

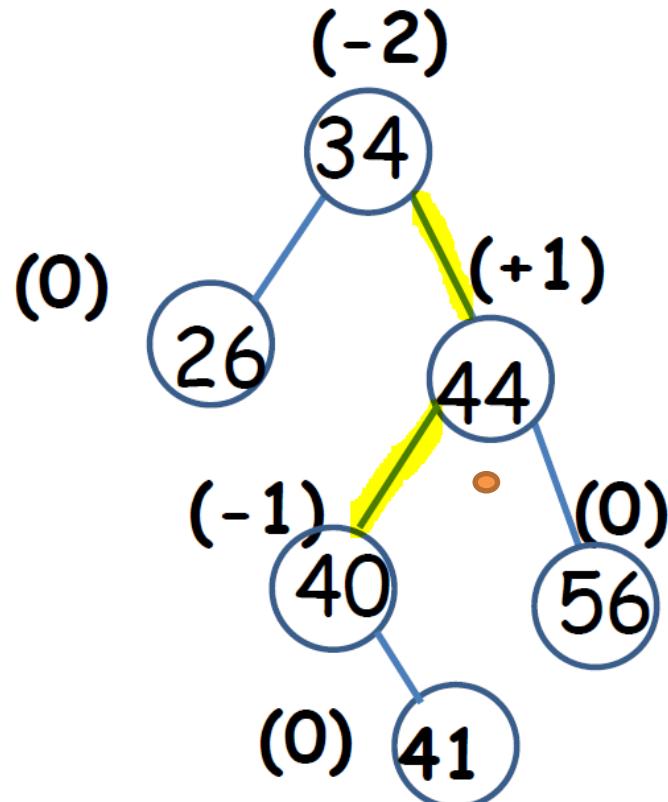


**Insert 41**

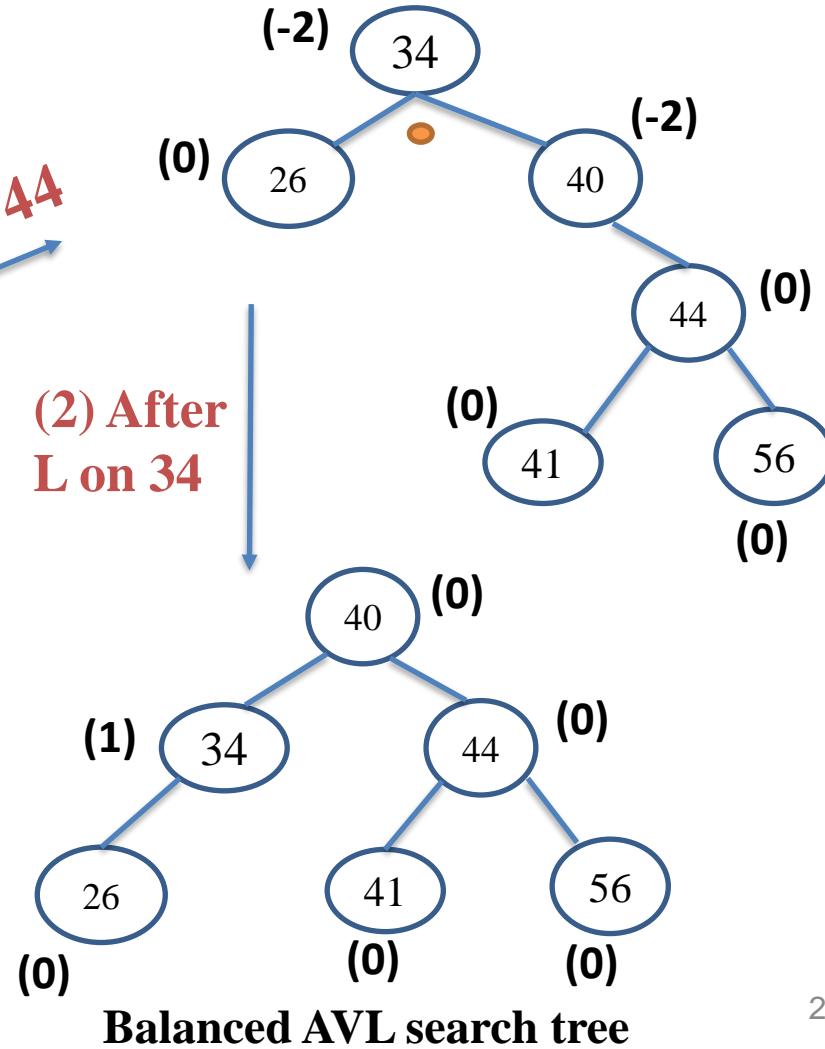


**Unbalanced AVL search tree**

# (RL) rotation Example



*(1) After R on 44*



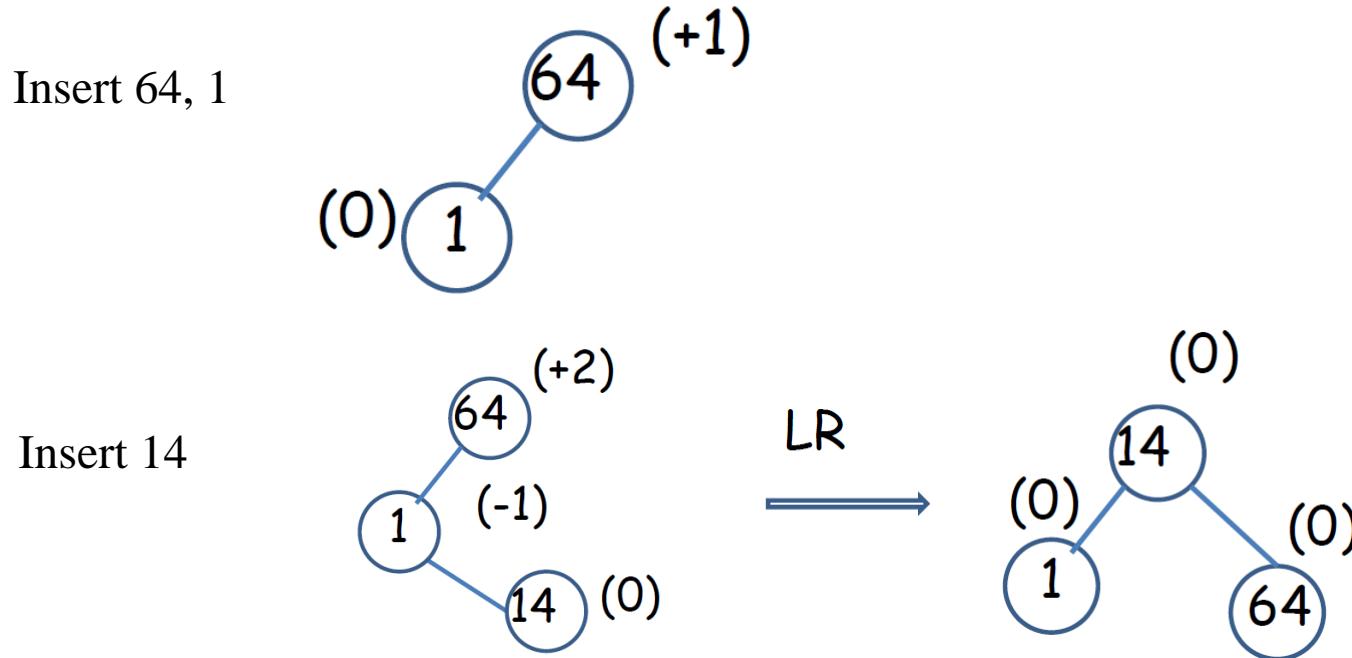
# Ready Reckoner ☺

| Involved Nodes in Straight line ? | Balance Factor | Type of rotation | Rotation Direction               |
|-----------------------------------|----------------|------------------|----------------------------------|
| Yes                               | +2             | R / LL rotation  | Clockwise                        |
| Yes                               | -2             | L / RR rotation  | Anti-Clockwise                   |
| No                                | +2             | LR rotation      | (1) L rotation<br>(2) R rotation |
| No                                | -2             | RL rotation      | (1) R rotation<br>(2) L rotation |

# Insertion Example

- Construct an AVL search tree by inserting the following elements in the order of their occurrence :

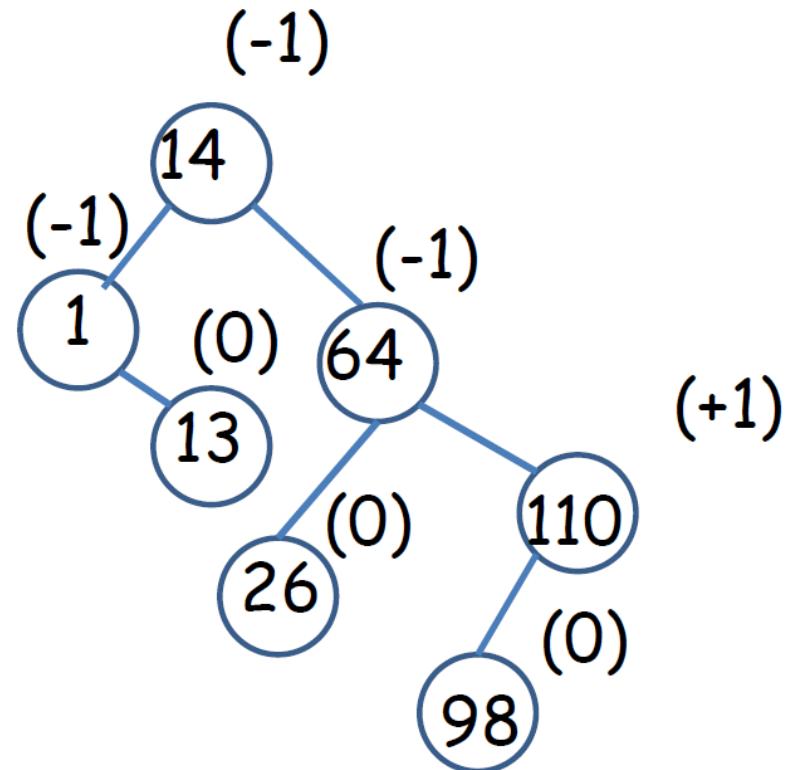
64, 1, 14, 26, 13, 110, 98, 85



# Insertion Example

64, 1, 14, 26, 13, 110, 98, 85

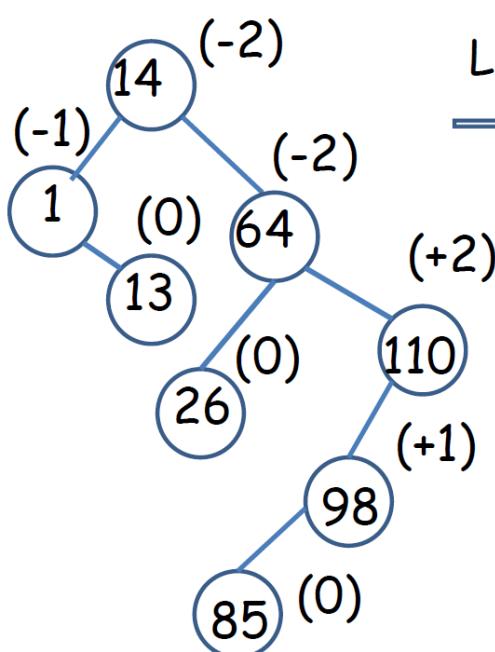
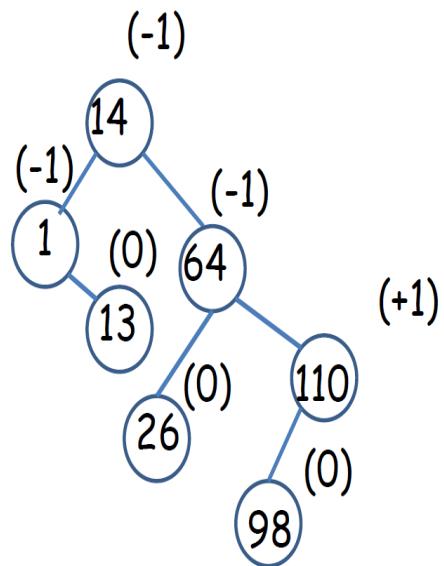
Insert 26, 13, 110, 98



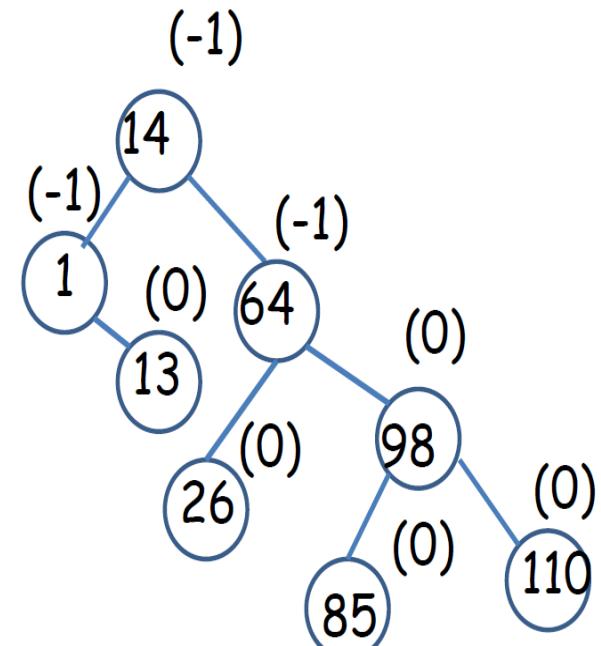
# Insertion Example

64, 1, 14, 26, 13, 110, 98, 85

Insert 85



LL

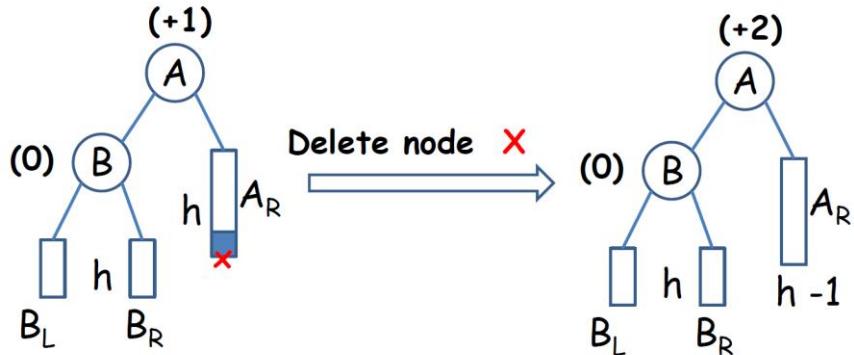


Balanced AVL search tree

# Removal from AVL Tree

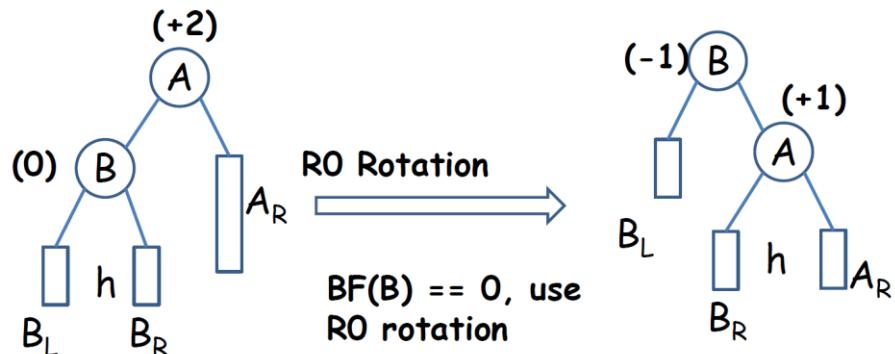
- *Deletion in AVL search tree proceed the same way as the deletion in binary search tree (So, all the 3 cases of BST is applicable here as well – Node to be deleted has no child, 1 child and 2 child).*
- However, in the event of imbalance due to deletion, one or more rotation need to be applied to balance the AVL tree.
- Let **A** be the closest ancestor node on the path from **X** (deleted node) to the root with a balancing factor +2 or -2
- If deleted node are from left subtree of A then It is called **Type L** delete otherwise it is called **Type R** delete.
- Depending on the BF(B), where B is the sibling of the deleted node (the one below the critical imbalanced node A in the opposite side of deletion), we have R0,R1 and R-1 type and L0, L1 and L-1 types.

# Deletion – R0 Case



R0 Case : Use LL / R rotation !!

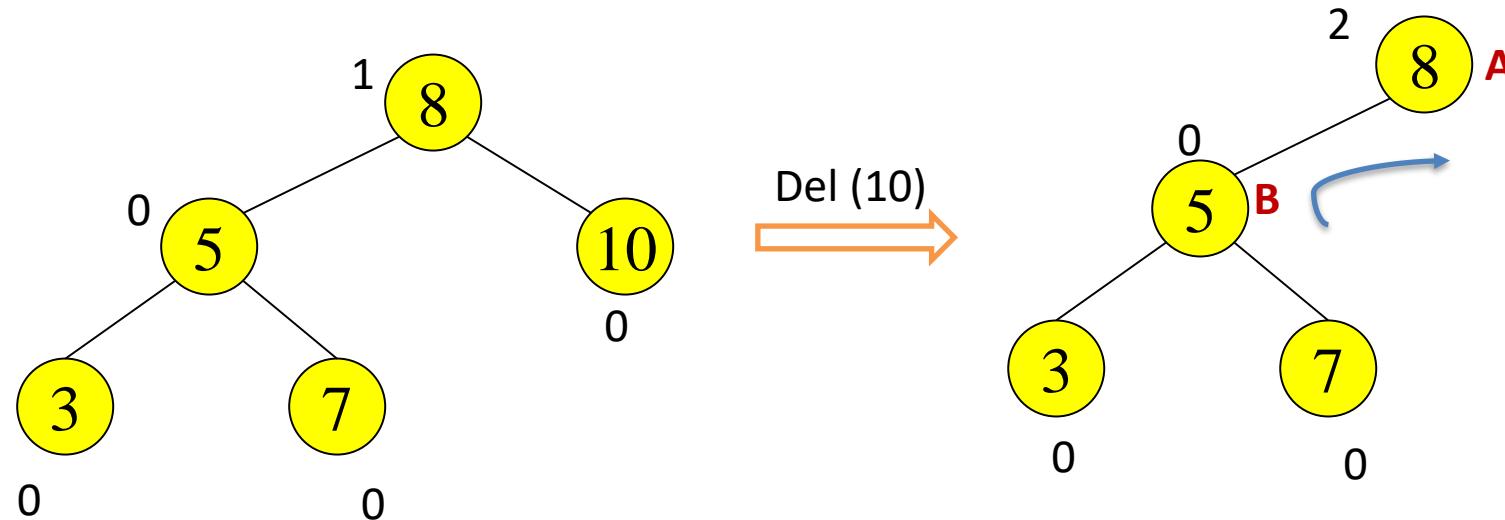
Unbalanced AVL search tree after deletion of node x



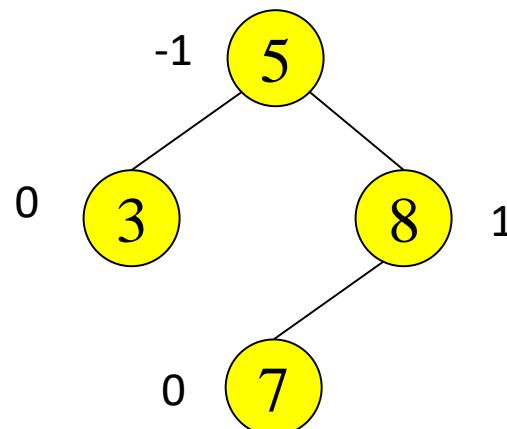
Unbalanced AVL search tree after deletion of x

Balanced AVL search tree after rotation

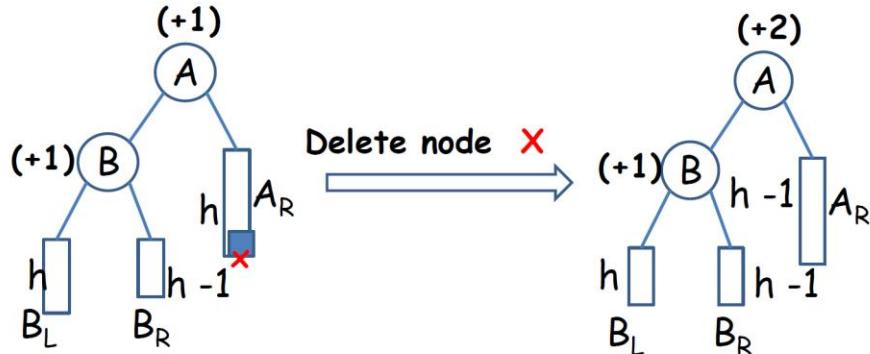
# Deletion – R0 Case Example



The BF(5) is 0 and Hence, it R(0) type . So, Apply R rotation on A

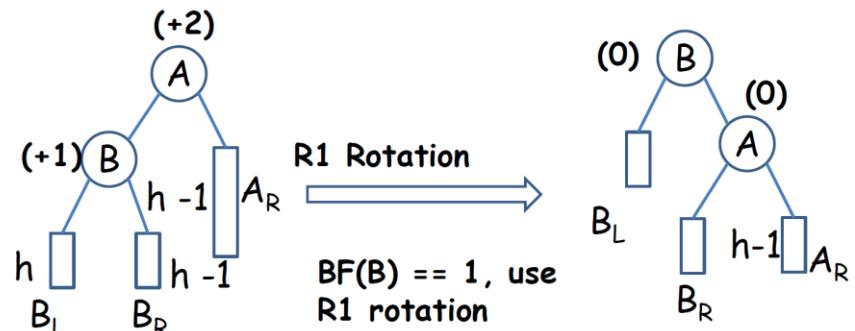


# Deletion – R1 Case



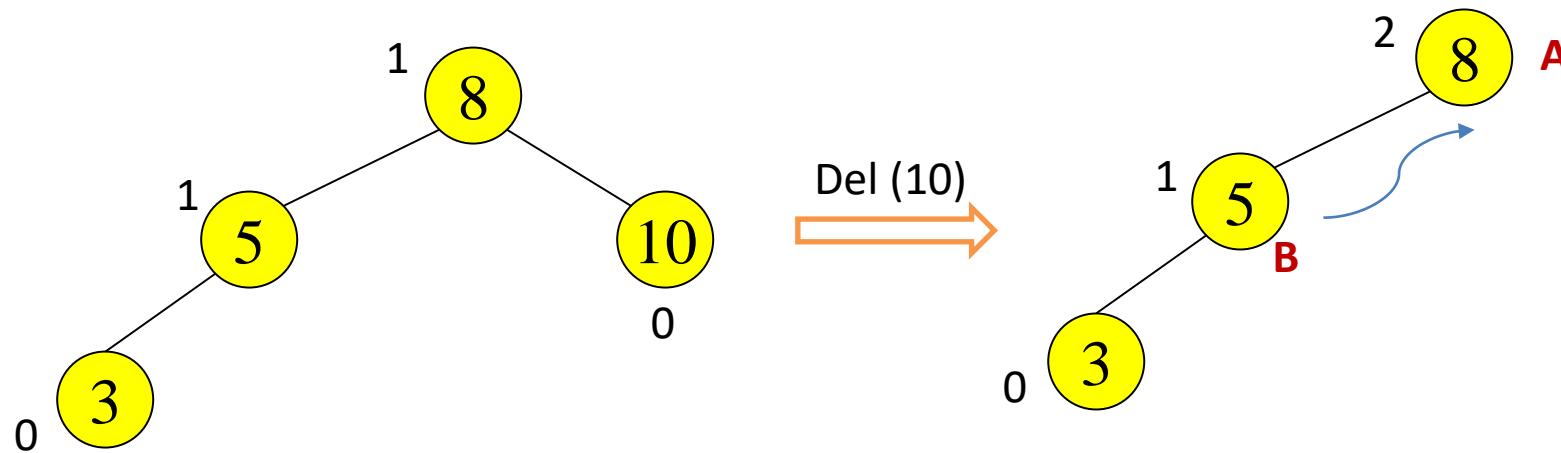
Unbalanced AVL search tree after deletion of node x

R1 Case : Use LL / R rotation !!

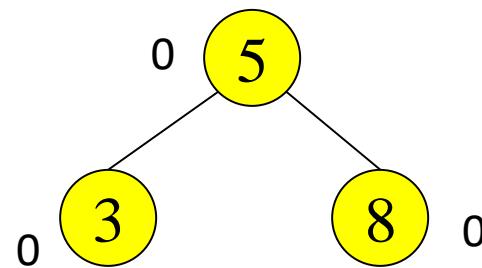


Balanced AVL search tree after rotation

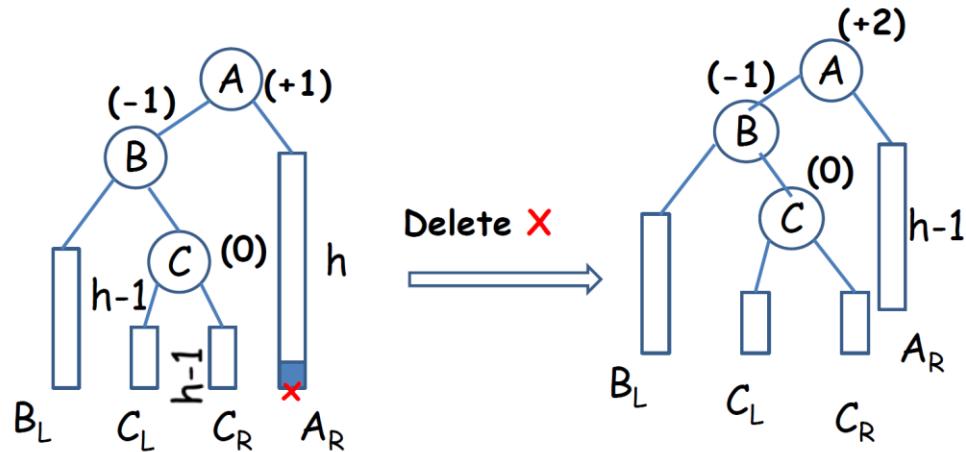
# Deletion – R1 Case Example



$\text{BF}(5) = 1$  , Hence its R(1) type – So, Apply right rotation on A

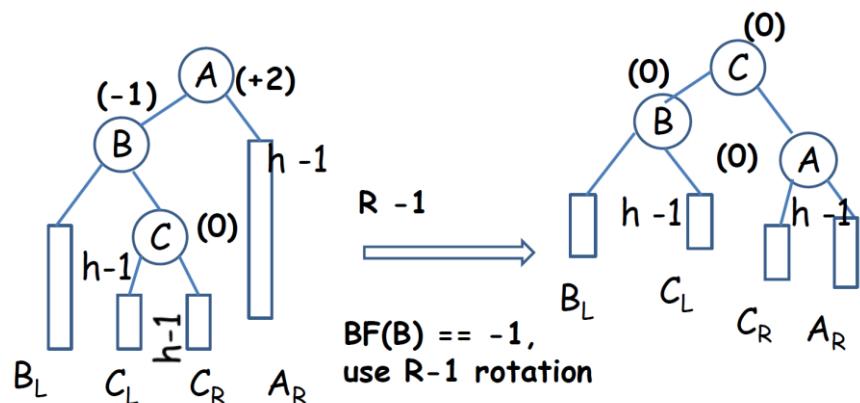


# Deletion R-1 Case



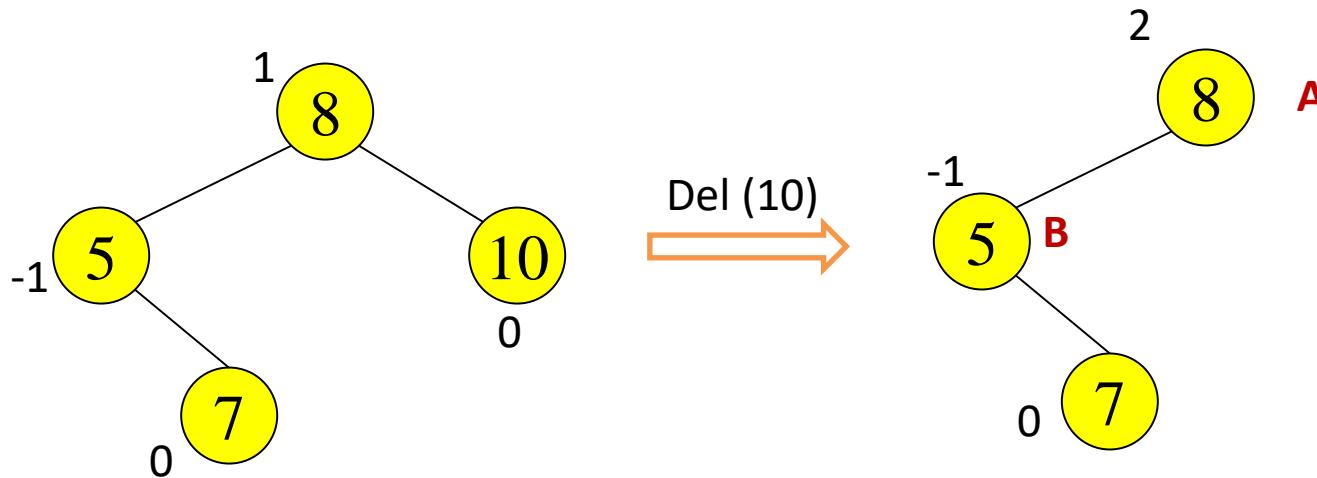
R-1 Case : Use LR rotation !!

Unbalanced AVL search tree after deletion

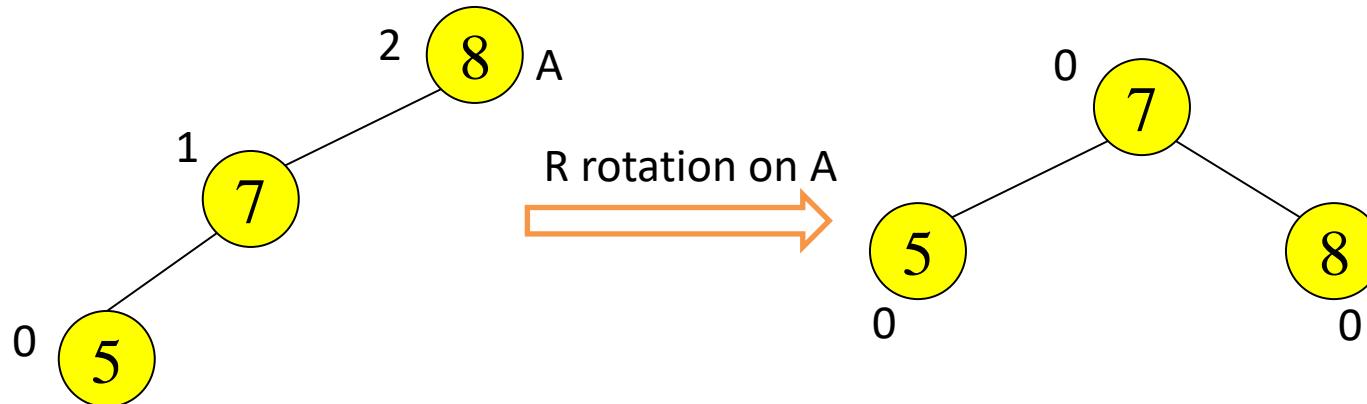


Balanced AVL search tree after Rotation

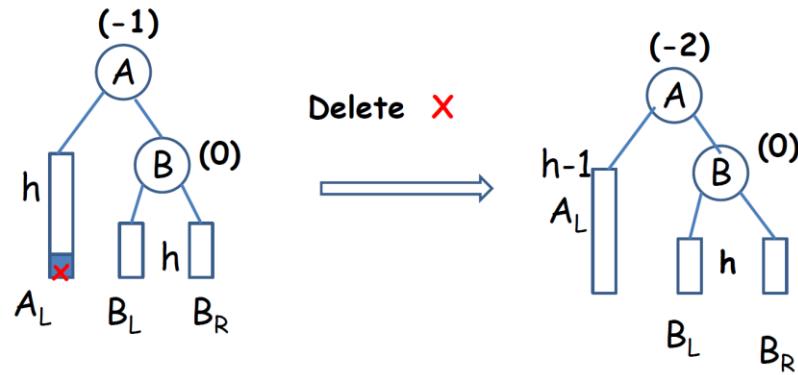
# Deletion R-1 Case Example



$\text{BF}(5) = -1$ , Hence its R(-1) type. So, Apply left Rotation on left child of node A i.e. on 5 and then right rotation on node A

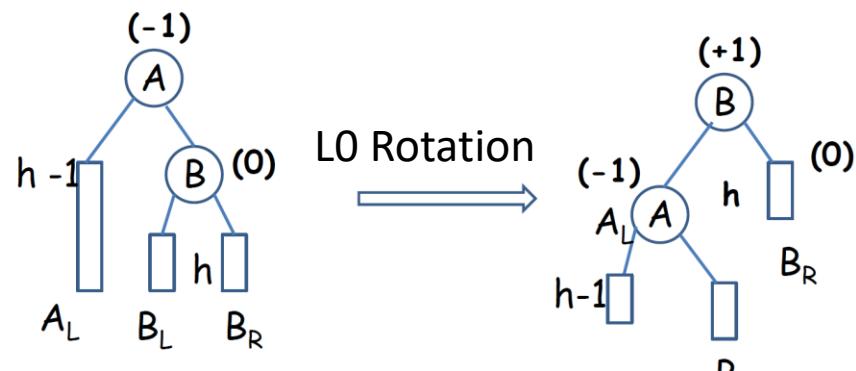


# Deletion L0 Case



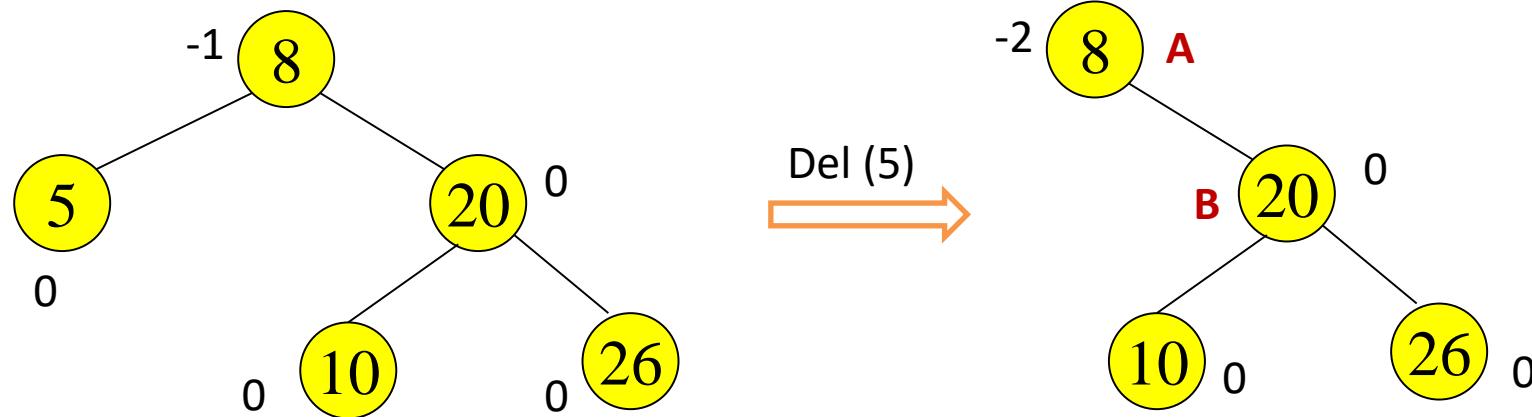
Unbalanced AVL search tree after deletion

L0 Case : Use RR/ L rotation !!

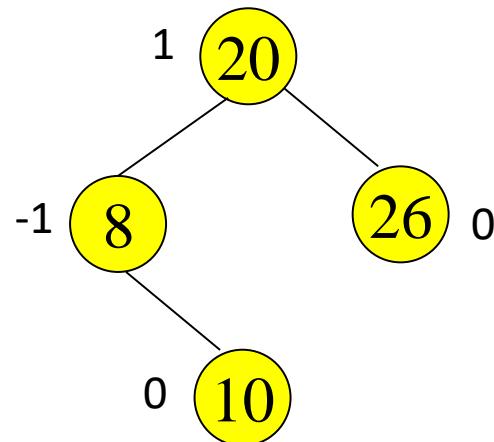


Balanced AVL search tree after deletion

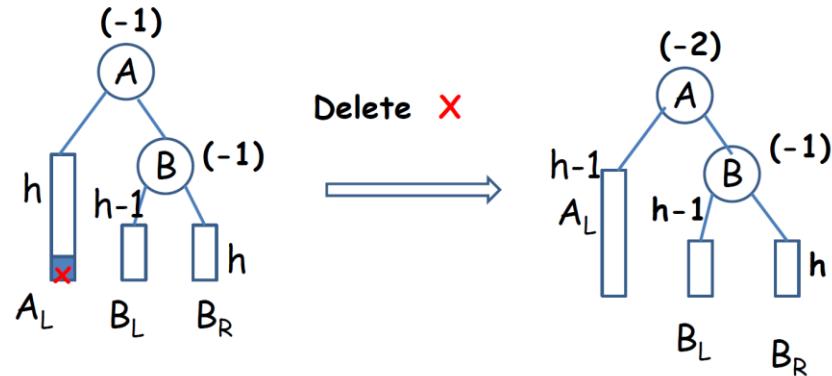
# Deletion L0 Case Example



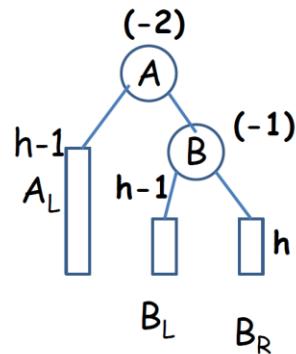
$\text{BF}(20) = 0$ , Hence its L(0) type. So, Apply Left rotation on A



# Deletion L-1 Case

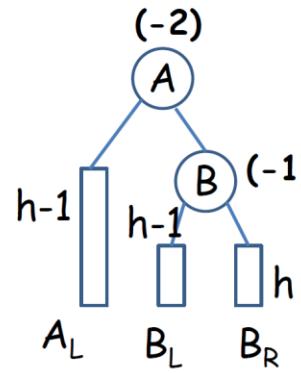


Delete X

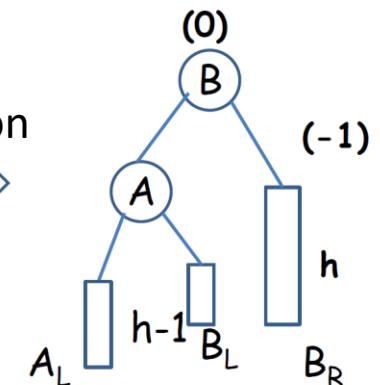


Unbalanced AVL search tree after deletion

L-1 Case : Use RR/L rotation !!

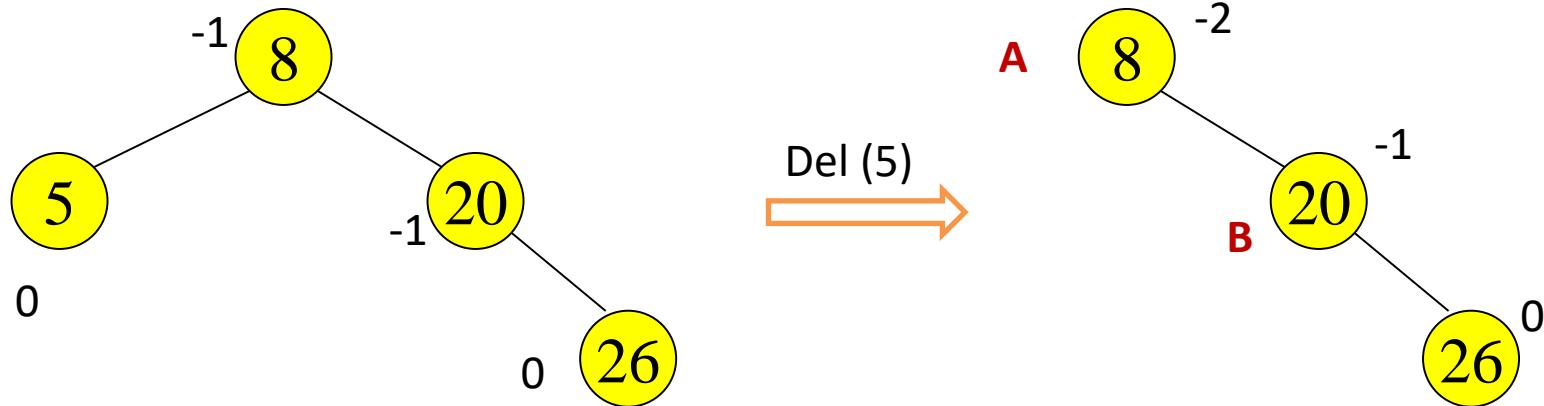


L-1 Rotation

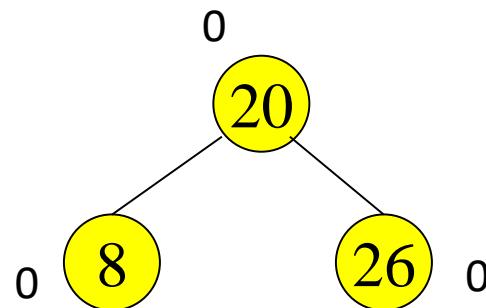


Balanced AVL search tree after deletion

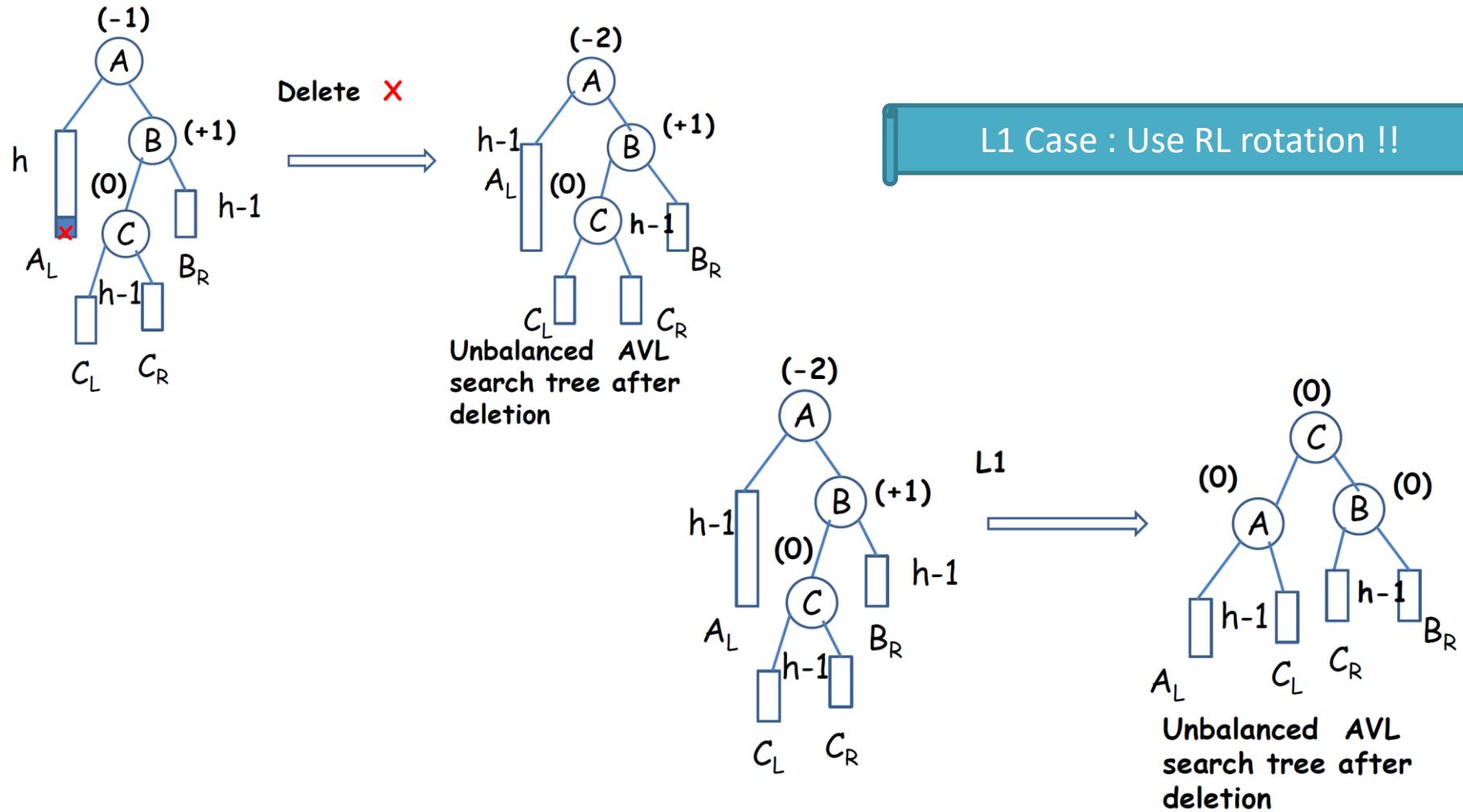
# Deletion L-1 Case Example



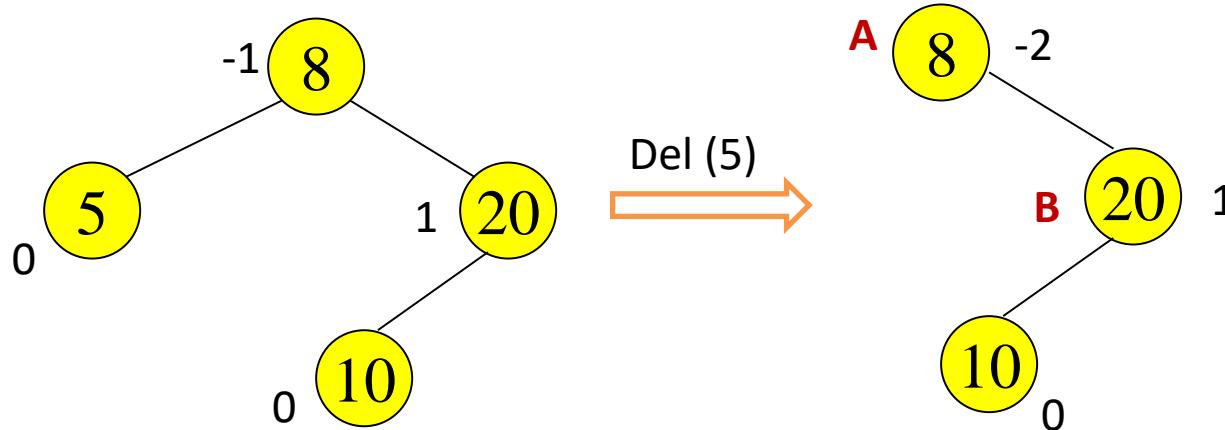
Since it  $L(-1)$  = case apply Left rotation on A



# Deletion L1 Case

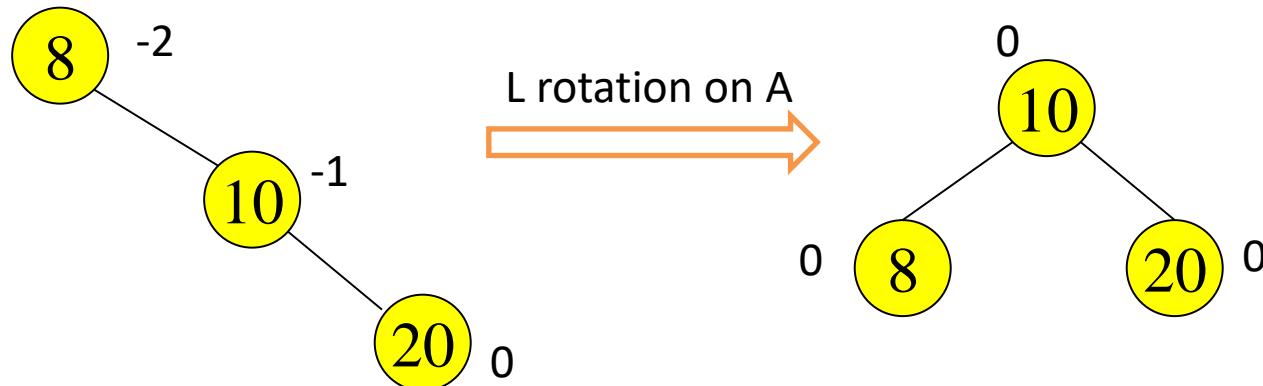


# Deletion L1 Case Example



Since it L(1), In L<sub>1</sub> case we have to solve in two steps,

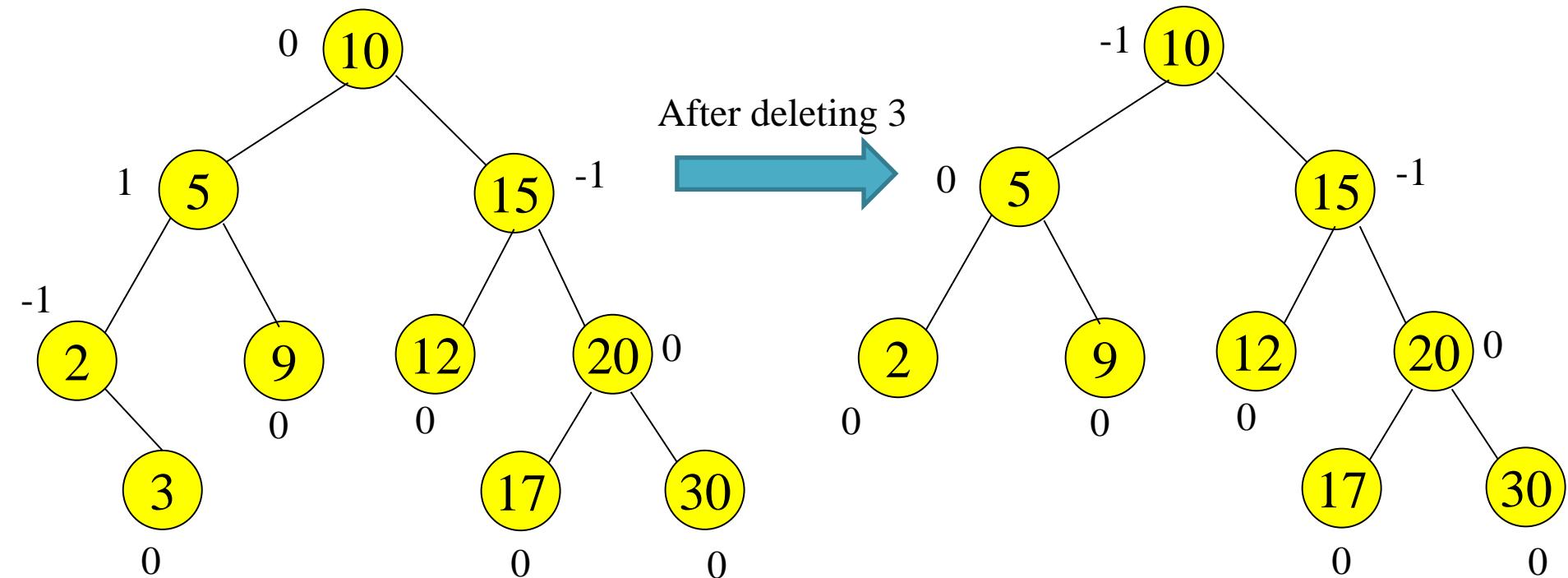
Step1: Right Rotation at right child of 'A' i.e. 20 Step2: Left rotation at node A



# AVL Deletion

Delete 3 from :

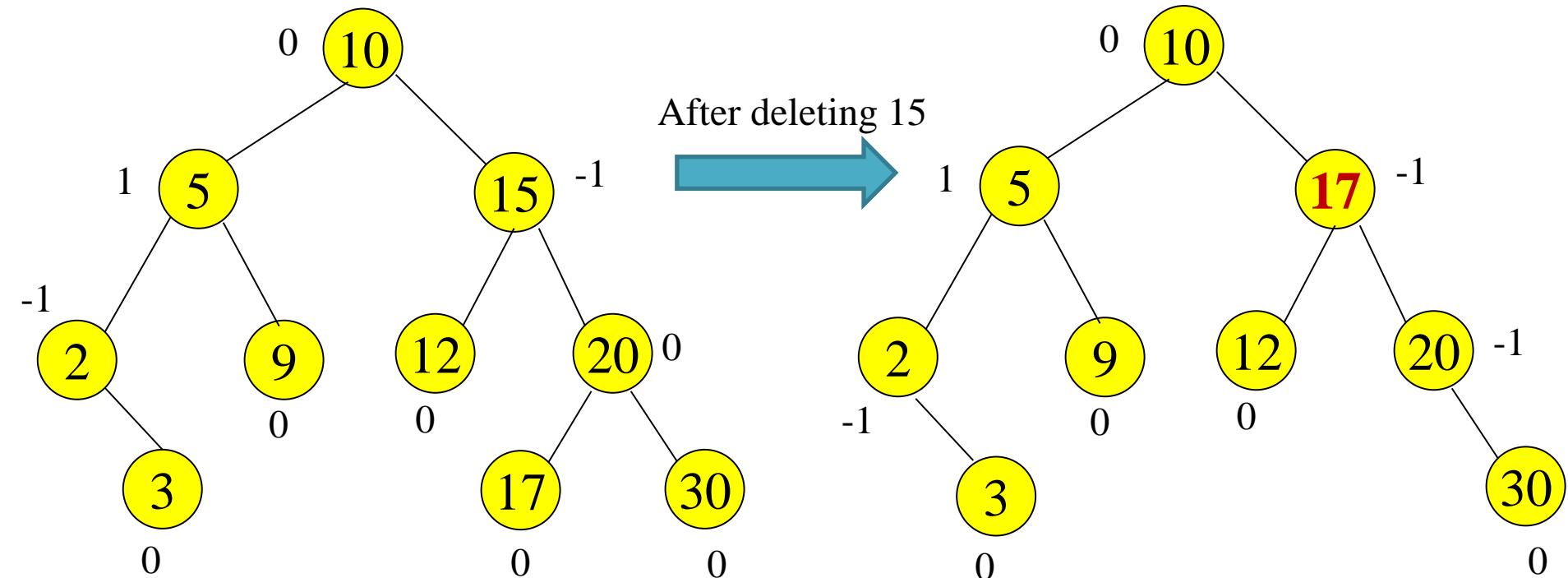
*3 has no children!! Easy delete & recalculate Balance factor. No need of rotations 😊*



# AVL Deletion

Delete 15 from :

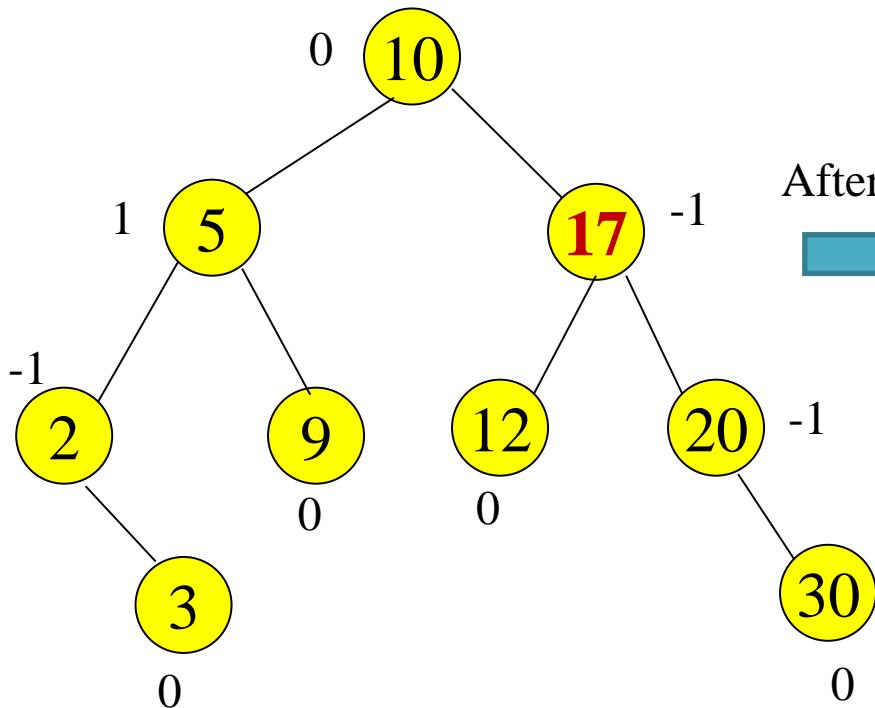
*15 has 2 children!! Find the inorder successor and place in position of 15*



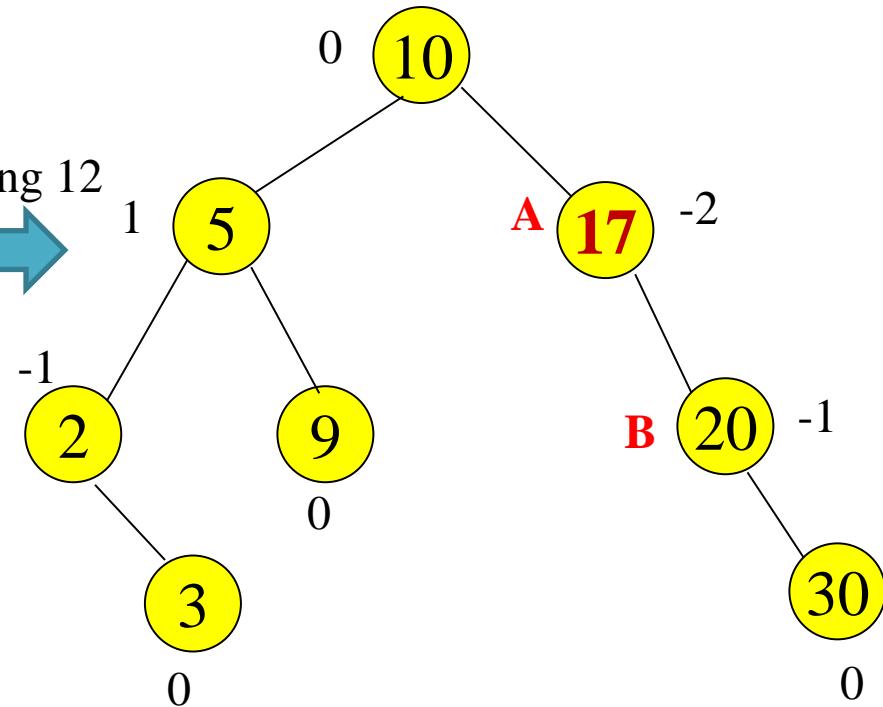
# AVL Deletion

Delete 12 from :

*12 has no children!! Easy delete & recalculate Balance factor. There is need of rotations ☹*



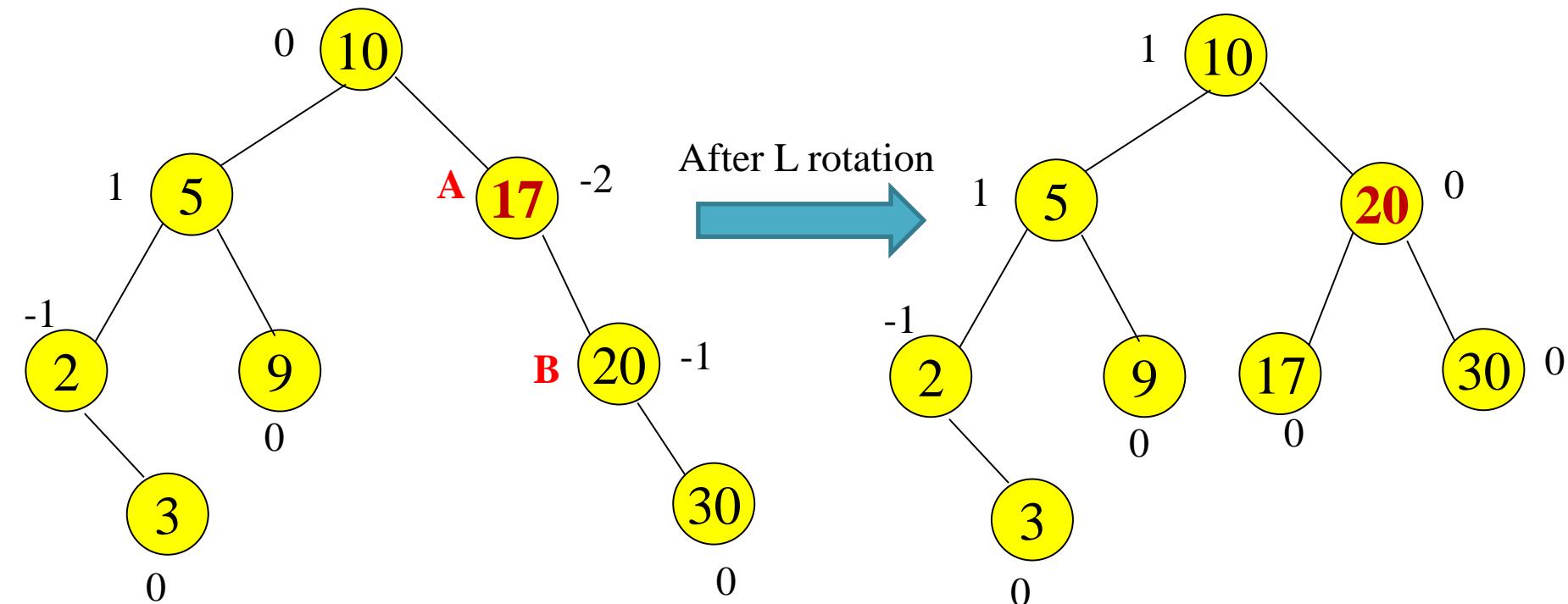
After deleting 12



We are in L-1 Case, so perform L rotation

# AVL Deletion

## Delete 12 from :



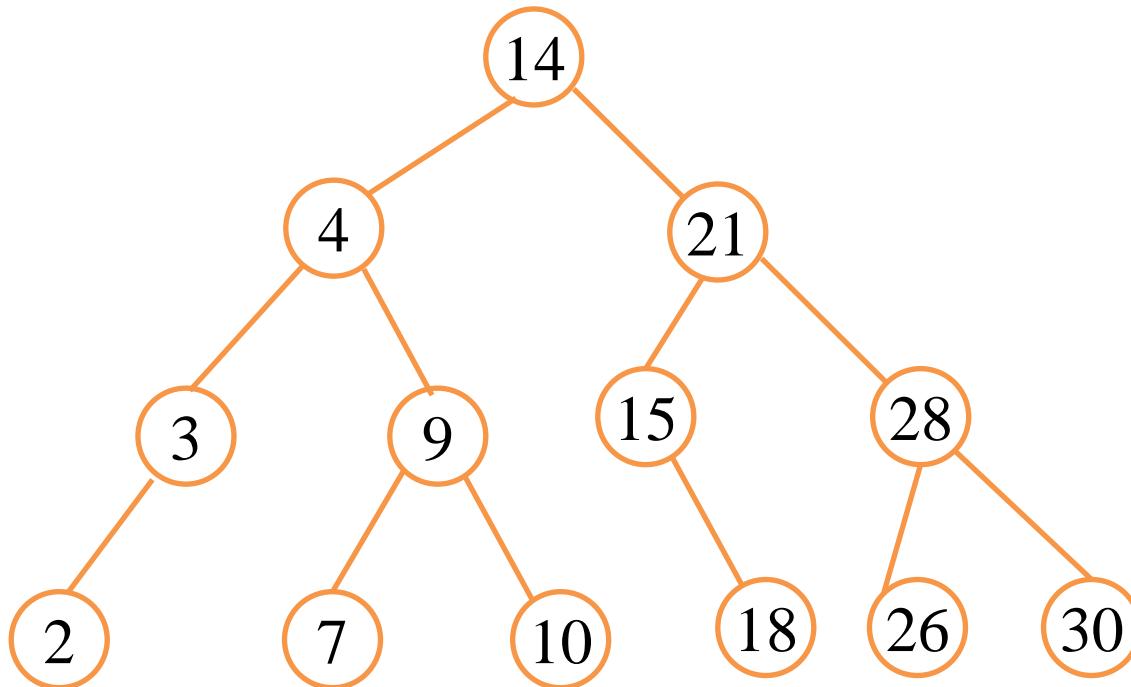
# Analysis of AVL Trees

---

- Due to the balancing property, the insertion, deletion and search operations take  $O(\log n)$  in both the average and the worst cases. Therefore, AVL trees give us an edge over Binary Search Trees which have an  $O(n)$  time complexity in the worst case scenario.
- The space complexity of an AVL tree is  $O(n)$  in both the average and the worst case.

# AVL Deletion - Exercise

- Delete 2,3,10,18,4,9,14,7 and 15 in the same order from the below AVL and show all the steps !



# Exercises

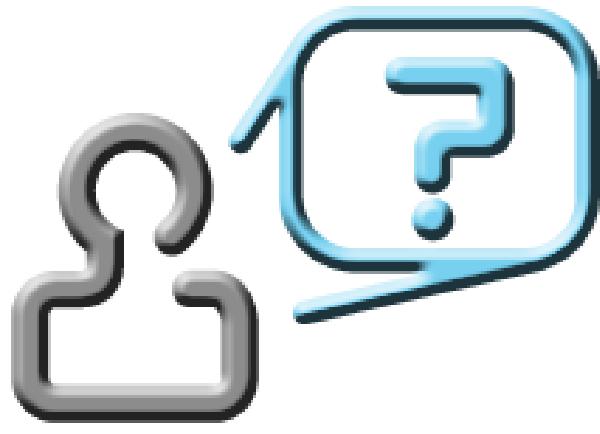
---

1. Write an algorithm to check whether a given a binary tree is a valid binary search tree (BST) or not.
2. Write an algorithm to find  $K^{\text{th}}$  smallest element in BST
3. Construct the AVL tree for : **63, 9, 19, 27, 18, 108, 99, 81**
4. Construct the AVL tree for : **H, I, J, B, A, E, C, F, D, G, K, L**
5. Delete F, G, L from the tree constructed in (4)

## Interactive Example

1. Try <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html> and add some elements and remove some elements and observe rotations happen!

**Exploration** : Explore uses of AVL.



*We will start with algorithms from the next class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS** Pilani  
Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

## DSECLZG519

Parthasarathy





# Contact Session #11

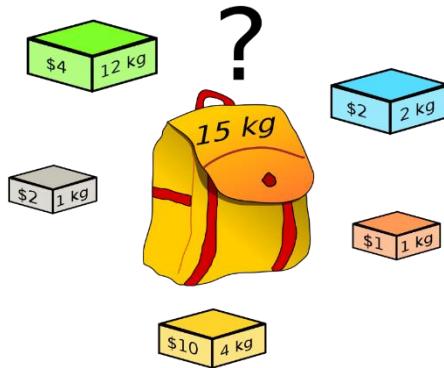
# DSECLZG519 – Greedy Techniques

# Agenda for Session # 11

---

- Recap CS#10
- Fractional Knapsack problem
- Job Sequencing problem
- Minimum Spanning Tree
  - Kruskal's Algorithm
  - Prim's Algorithm
- Single Source Shortest Path
- Exercises ☺

# Knapsack Problem or Container Problem



Based on the nature of the items, Knapsack problems are categorized as :

- Fractional Knapsack
- 0/1 Knapsack

| Objects O | 1  | 2 | 3  | 4 | 5 | 6  | 7 |
|-----------|----|---|----|---|---|----|---|
| Profits P | 10 | 5 | 15 | 7 | 6 | 18 | 3 |
| Weights W | 2  | 3 | 5  | 7 | 1 | 4  | 1 |

$$\begin{aligned} N &= 7 \\ W &= 15 \end{aligned}$$

## *Constraint:*

The sum of the weights of the objects we place in the knapsack must be less than Or equal to the capacity of the knapsack i.e.  $\sum x_i w_i \leq m$

## *Objective*

Maximize the profit :  $\sum X_i P_i$

# Knapsack Problem Example

|           |    |   |    |   |   |    |   |
|-----------|----|---|----|---|---|----|---|
| Objects O | 1  | 2 | 3  | 4 | 5 | 6  | 7 |
| Profits P | 10 | 5 | 15 | 7 | 6 | 18 | 3 |
| Weights W | 2  | 3 | 5  | 7 | 1 | 4  | 1 |

$$N=7 \\ W = 15$$

|     |   |     |   |   |   |     |   |
|-----|---|-----|---|---|---|-----|---|
| P/W | 5 | 1.3 | 3 | 1 | 6 | 4.5 | 3 |
| x   | 1 |     |   |   | 1 | 1   |   |

- Step 1: Select Object whose P/W is highest. Here, O5 is selected and put it fully in Knapsack.  
Now the Available capacity in Knapsack is  $15 - 1 = 14\text{Kg}$
- Step 2: Select the next Object whose P/W is highest. Here, O1 is selected and put it fully in Knapsack. Now the Available capacity in Knapsack is  $14 - 2 = 12\text{Kg}$
- Step 3: Select the next Object whose P/W is highest. Here, O6 is selected and put it fully in Knapsack. Now the Available capacity in Knapsack is  $12 - 4 = 8\text{Kg}$

# Knapsack Problem Example

|           |    |     |    |   |   |     |   |
|-----------|----|-----|----|---|---|-----|---|
| Objects O | 1  | 2   | 3  | 4 | 5 | 6   | 7 |
| Profits P | 10 | 5   | 15 | 7 | 6 | 18  | 3 |
| Weights W | 2  | 3   | 5  | 7 | 1 | 4   | 1 |
| P/W       | 5  | 1.3 | 3  | 1 | 6 | 4.5 | 3 |
| x         | 1  | 2/3 | 1  | 0 | 1 | 1   | 1 |

N = 7  
W = 15

- Step 4: Select Object whose P/W is highest. Here, O3 and O7 both yield the same. We select O3 put it fully in Knapsack. Now the Available capacity in Knapsack is  $8 - 5 = 3\text{Kg}$
- Step 5: Select the next Object whose P/W is highest. Here, O7 is selected and put it fully in Knapsack. Now the Available capacity in Knapsack is  $3 - 1 = 2\text{Kg}$
- Step 6: Select the next Object whose P/W is highest. Here, O2 is selected but we cannot place it fully in the knapsack as the capacity available in knapsack is just 2Kg. Hence, We take 2/3 of O2. Knapsack is now full!

# Knapsack Problem Example

|           |    |     |    |   |   |     |   |
|-----------|----|-----|----|---|---|-----|---|
| Objects O | 1  | 2   | 3  | 4 | 5 | 6   | 7 |
| Profits P | 10 | 5   | 15 | 7 | 6 | 18  | 3 |
| Weights W | 2  | 3   | 5  | 7 | 1 | 4   | 1 |
| P/W       | 5  | 1.3 | 3  | 1 | 6 | 4.5 | 3 |
| x         | 1  | 2/3 | 1  | 0 | 1 | 1   | 1 |

$$N = 7 \\ W = 15$$

- $\sum X_i Wi = (1 * 2) + \left(\frac{2}{3} * 3\right) + (1 * 5) + (0 * 7) + (1 * 1) + (1 * 4) + (1 * 1)$   
 $= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$ , so the constraint  $\sum x_i wi \leq m$  is satisfied,  
*where m is the capacity of the knapsack.*
- $\sum X_i Pi = (1 * 10) + \left(\frac{2}{3} * 5\right) + (1 * 15) + (0 * 7) + (1 * 6) + (1 * 18) + (1 * 3)$   
 $= 10 + 3.33 + 15 + 0 + 6 + 18 + 3 = 55.33$ , so the objective was to maximize  
 $\sum X_i P_i$  which is also done. Maximum Profit is **55.33**

# Job Sequencing with Deadlines

- Problem: Given an array of jobs/tasks where every job has a deadline associated with it. If the job is finished before/on the deadline, there is a profit associated with it. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. how to maximize total profit if only one job can be scheduled at a time ?

N=5

| Jobs ID  | 1  | 2  | 3  | 4 | 5 |
|----------|----|----|----|---|---|
| Profit   | 20 | 15 | 10 | 5 | 1 |
| Deadline | 2  | 2  | 1  | 3 | 3 |

8AM ————— 9AM ————— 10AM ————— 11AM

# Job Sequencing with Deadlines

N=5

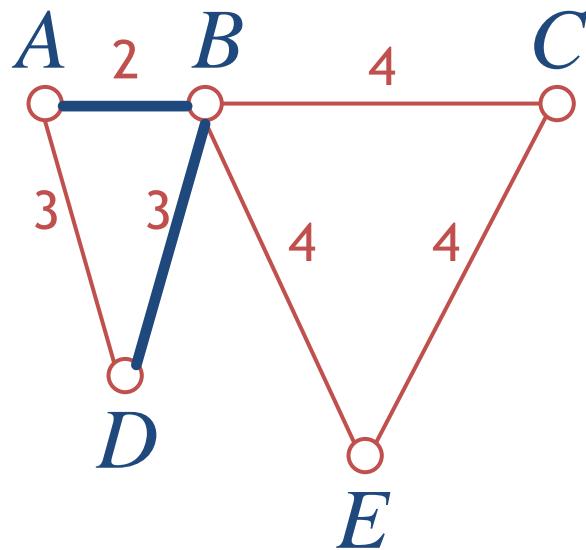
| Jobs ID  | 1  | 2  | 3  | 4 | 5 |
|----------|----|----|----|---|---|
| Profit   | 20 | 15 | 10 | 5 | 1 |
| Deadline | 2  | 2  | 1  | 3 | 3 |



| Considered | Slot Assigned                           | Solution        | Profit     |
|------------|---|-----------------|------------|
| -          | -                                       | Null            | 0          |
| J1         | [1-2]                                   | J1              | 20         |
| J2         | [0-1],[1-2]                             | J1,J2           | 20+15 = 35 |
| J3         | J3 Rejected (as deadline can't be meet) | J1,J2           | 35         |
| J4         | [0-1],[1-2],[2-3]                       | J1,J2,J4        | 35+5 = 40  |
| J5         | J5 Rejected (as deadline can't be meet) | <b>J1,J2,J4</b> | <b>40</b>  |

# Traveling Salesman

- A salesman must visit every city (starting from city  $A$ ), and wants to cover the least possible distance
  - He can revisit a city (and reuse a road) if necessary
- He does this by using a greedy algorithm: He goes to the next nearest city from wherever he is

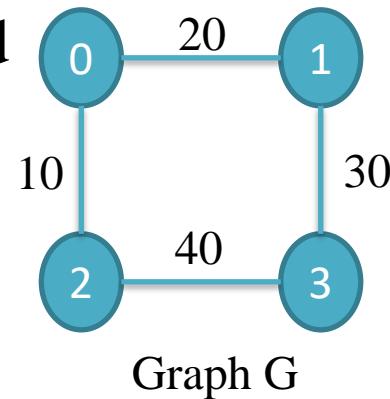


- From  $A$  he goes to  $B$
- From  $B$  he goes to  $D$
- This is *not* going to result in a shortest path!
- The best result he can get now will be  $ABDBCE$ , at a cost of 16
- An actual least-cost path from  $A$  is  $ADBCE$ , at a cost of 14

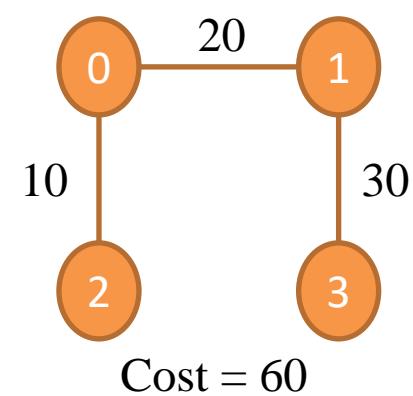
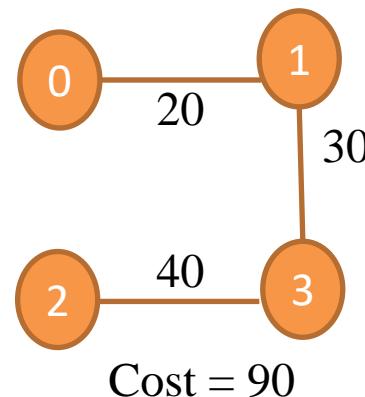
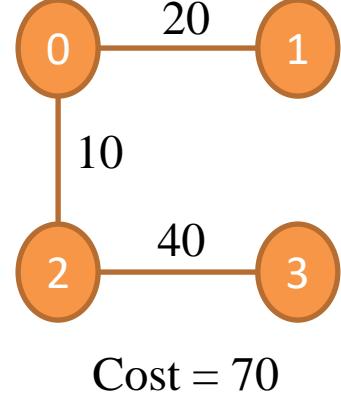
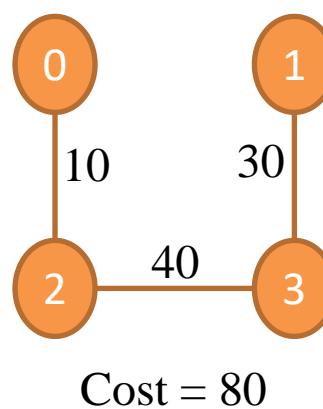
# Spanning Tree

A spanning tree is a subgraph of a graph such that the subgraph spans (includes) all nodes, is connected and is acyclic ( does not contain cycle/loop/circuit). Formally, a spanning tree of graph  $G$  is defined as a sub graph  $G' = (V',E')$  with the following properties :

- ✓  $G'$  is connected
- ✓  $V' = V$
- ✓  $G'$  is acyclic

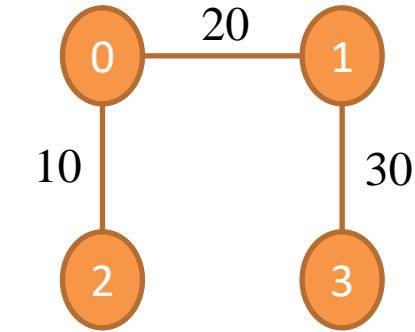
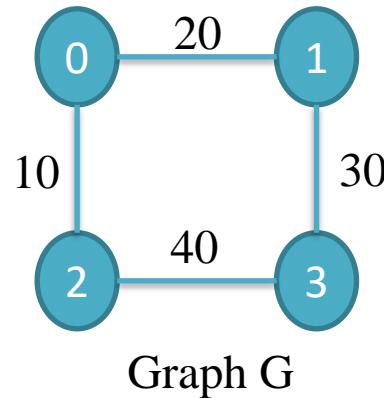


Spanning Trees



# Minimum Spanning Tree (MST)

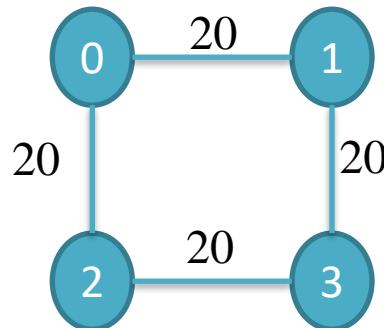
- *A minimum spanning tree of a given graph G is a spanning tree whose cost is minimum.*
- For the previous Graph G, the minimum spanning tree is



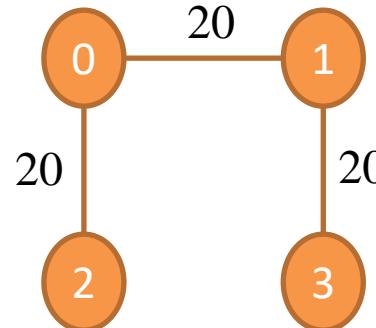
- A graph will have only one MST iff the weights associated with all the edges in the graph are distinct. Example for above Graph G, there is only one MST with cost 60.

# Minimum Spanning Tree (MST)

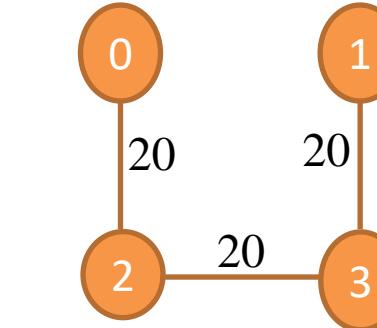
- Consider the below Graph G,



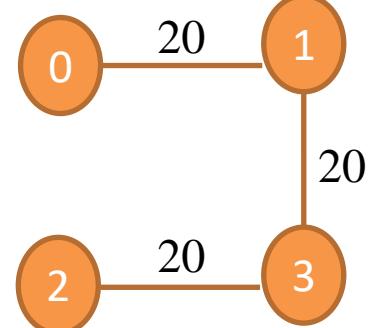
Graph G



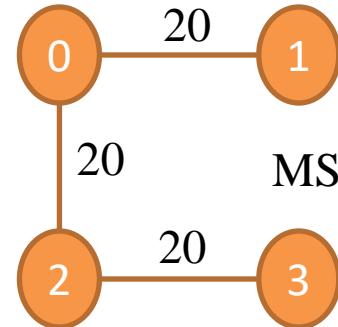
MST with Cost = 60



MST with Cost = 60



MST with Cost = 60

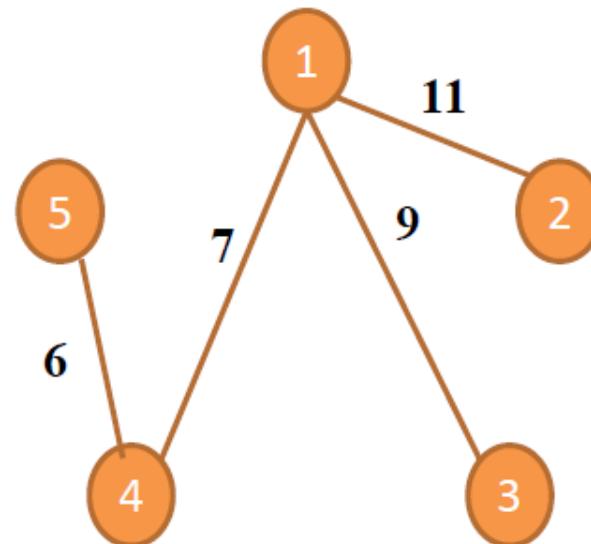
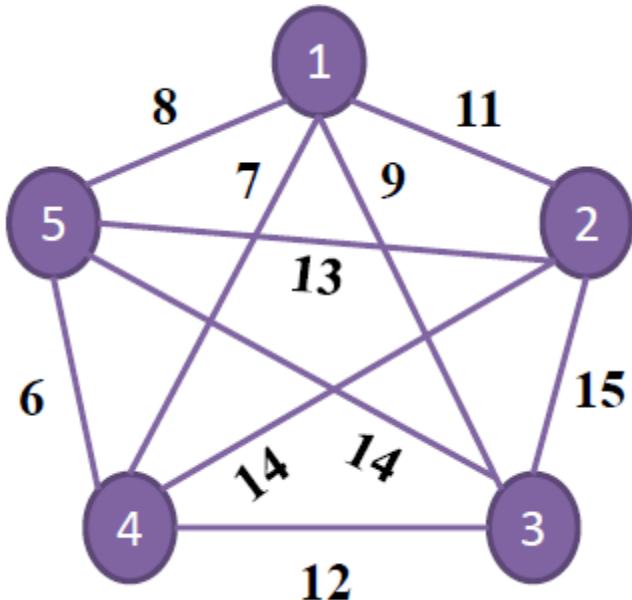


MST with Cost = 60

- Here, since the weights are not distinct, we have more than 1 MST.

# Classroom Exercise [3 min]

Draw the MST for the below Graph G and also provide its cost :



*MST with Cost =  $6+7+9+11 = 33$  units*

# Minimum Spanning Tree

There are plenty of algorithms for finding MST like Kruskal, Prim-Jarnik, Borůvka's, Edmonds' and so on.

We will look at two most popular greedy algorithms to find the MST:

- **Kruskal's Algorithm:**

Consider edges in ascending order of cost. Add the next edge to T unless doing so would create a cycle.

- **Prim's Algorithm:**

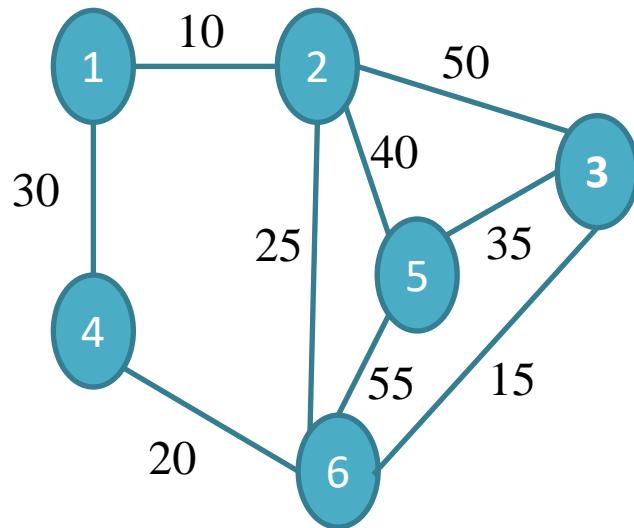
Start with any vertex S and greedily grow a tree T from S. At each step, add the cheapest edge to T that has exactly one endpoint in T.

- Both the algorithms produce optimal solutions and are greedy techniques.
- Both the algorithms *work with undirected graphs only !!* (We will see why these algorithms cannot be used for directed graphs).

# Kruskal's Algorithm

- Kruskal's algorithm is a greedy algorithm used to obtain MST using a direct generic method. Here, all the edges are considered in non-decreasing (increasing) order and the edges to be included are based on the order.
- Procedure for Kruskal's approach to MST:
  - Step 01: Construct an edge list E in the increasing order of their costs.
  - Step 02: Pick one edge at a time from the list starting from the least cost edge.
  - Step 03: Check if the inclusion of the edge causes a cycle or a loop in the output.
  - Step 04: If it causes a cycle, discard it. If it does not generate a cycle, include it into the tree.
  - Step 05: Repeat the steps 2 – 4 until a tree is generated with all nodes of the graph.

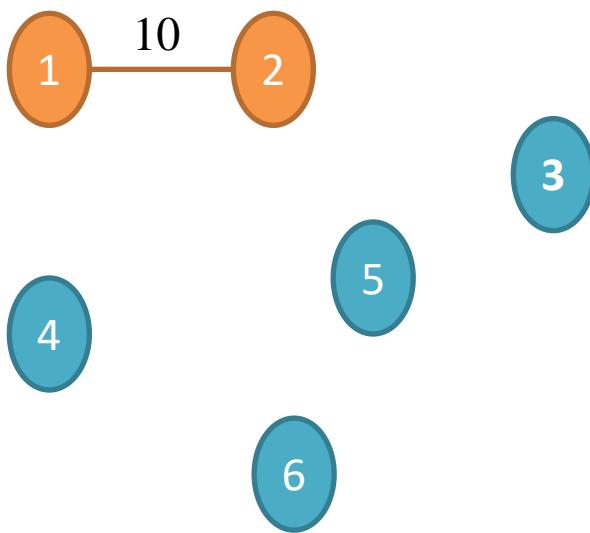
# Kruskal's Example 1



Edge List E in Increasing Order

| Edge  | Cost |
|-------|------|
| 1 – 2 | 10   |
| 3 – 6 | 15   |
| 4 – 6 | 20   |
| 2 – 6 | 25   |
| 1 – 4 | 30   |
| 3 – 5 | 35   |
| 2 – 5 | 40   |
| 2 – 3 | 50   |
| 5 – 6 | 55   |

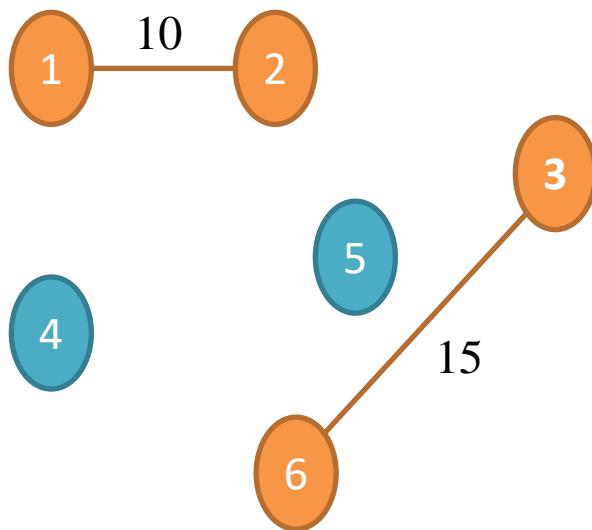
# Kruskal's Example 1



Kruskal in Action

| Edge  | Cost | Include ? | Status |
|-------|------|-----------|--------|
| 1 – 2 | 10   | Yes       | ✓      |
| 3 – 6 | 15   |           |        |
| 4 – 6 | 20   |           |        |
| 2 – 6 | 25   |           |        |
| 1 – 4 | 30   |           |        |
| 3 – 5 | 35   |           |        |
| 2 – 5 | 40   |           |        |
| 2 – 3 | 50   |           |        |
| 5 – 6 | 55   |           |        |

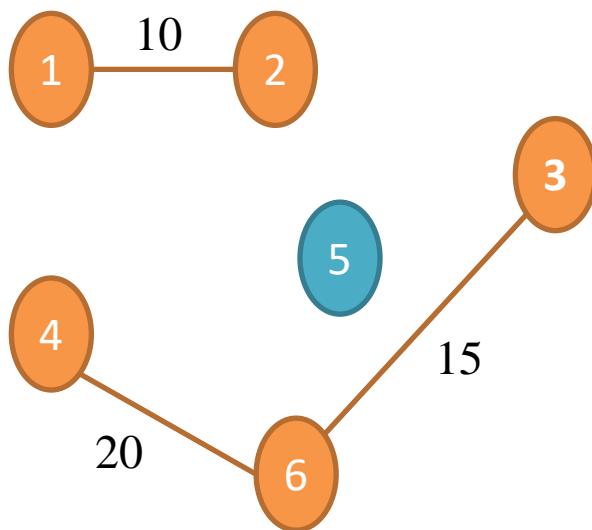
# Kruskal's Example 1



Kruskal in Action

| Edge  | Cost | Include ? | Status |
|-------|------|-----------|--------|
| 1 – 2 | 10   | Yes       | ✓      |
| 3 – 6 | 15   | Yes       | ✓      |
| 4 – 6 | 20   |           |        |
| 2 – 6 | 25   |           |        |
| 1 – 4 | 30   |           |        |
| 3 – 5 | 35   |           |        |
| 2 – 5 | 40   |           |        |
| 2 – 3 | 50   |           |        |
| 5 – 6 | 55   |           |        |

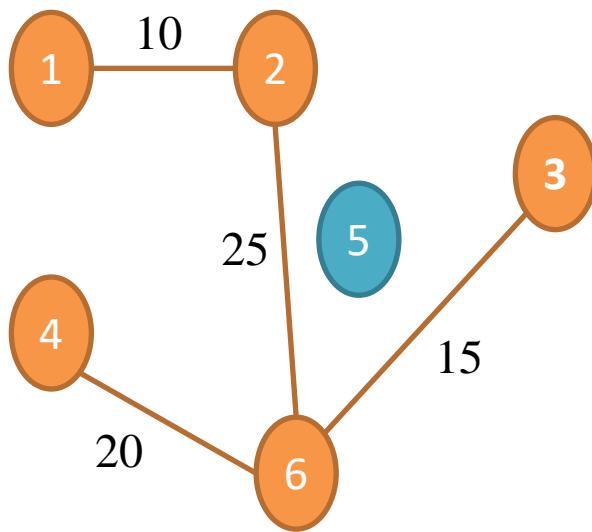
# Kruskal's Example 1



Kruskal in Action

| Edge  | Cost | Include ? | Status |
|-------|------|-----------|--------|
| 1 – 2 | 10   | Yes       | ✓      |
| 3 – 6 | 15   | Yes       | ✓      |
| 4 – 6 | 20   | Yes       | ✓      |
| 2 – 6 | 25   |           |        |
| 1 – 4 | 30   |           |        |
| 3 – 5 | 35   |           |        |
| 2 – 5 | 40   |           |        |
| 2 – 3 | 50   |           |        |
| 5 – 6 | 55   |           |        |

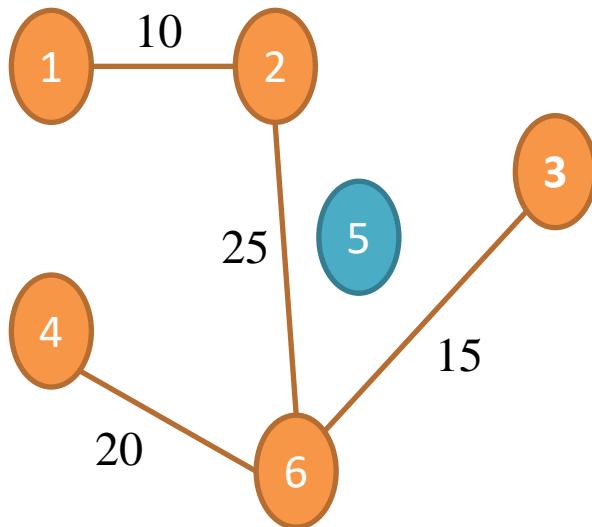
# Kruskal's Example 1



Kruskal in Action

| Edge  | Cost | Include ? | Status |
|-------|------|-----------|--------|
| 1 – 2 | 10   | Yes       | ✓      |
| 3 – 6 | 15   | Yes       | ✓      |
| 4 – 6 | 20   | Yes       | ✓      |
| 2 – 6 | 25   | Yes       | ✓      |
| 1 – 4 | 30   |           |        |
| 3 – 5 | 35   |           |        |
| 2 – 5 | 40   |           |        |
| 2 – 3 | 50   |           |        |
| 5 – 6 | 55   |           |        |

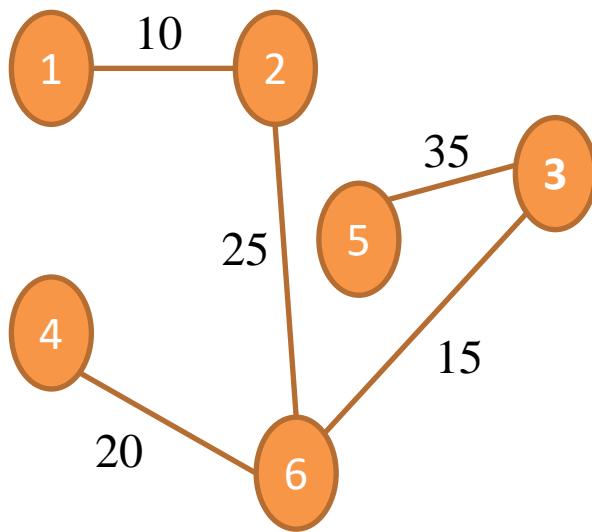
# Kruskal's Example 1



Kruskal in Action

| Edge  | Cost | Include ?           | Status |
|-------|------|---------------------|--------|
| 1 – 2 | 10   | Yes                 | ✓      |
| 3 – 6 | 15   | Yes                 | ✓      |
| 4 – 6 | 20   | Yes                 | ✓      |
| 2 – 6 | 25   | Yes                 | ✓      |
| 1 – 4 | 30   | No – Leads to Cycle | ✗      |
| 3 – 5 | 35   |                     |        |
| 2 – 5 | 40   |                     |        |
| 2 – 3 | 50   |                     |        |
| 5 – 6 | 55   |                     |        |

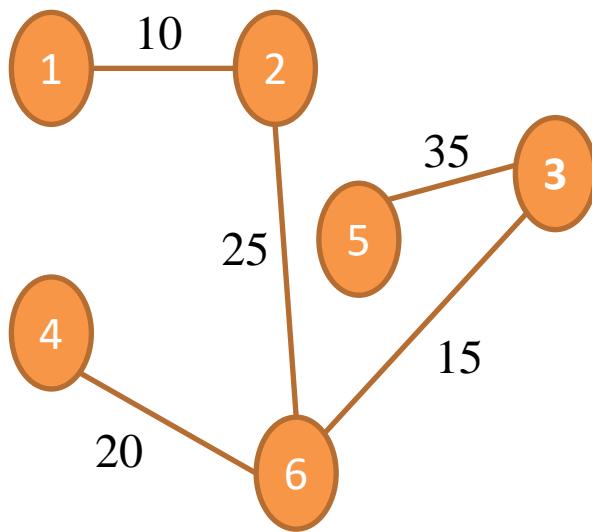
# Kruskal's Example 1



Kruskal in Action

| Edge  | Cost | Include ?           | Status |
|-------|------|---------------------|--------|
| 1 – 2 | 10   | Yes                 | ✓      |
| 3 – 6 | 15   | Yes                 | ✓      |
| 4 – 6 | 20   | Yes                 | ✓      |
| 2 – 6 | 25   | Yes                 | ✓      |
| 1 – 4 | 30   | No – Leads to Cycle | ✗      |
| 3 – 5 | 35   | Yes                 | ✓      |
| 2 – 5 | 40   |                     |        |
| 2 – 3 | 50   |                     |        |
| 5 – 6 | 55   |                     |        |

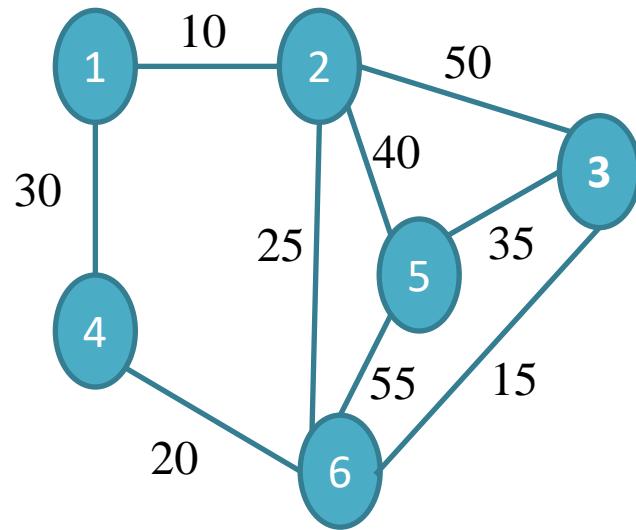
# Kruskal's Example 1



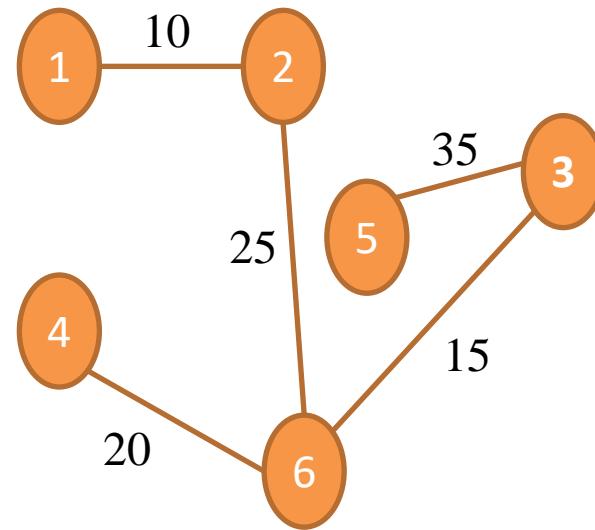
Kruskal in Action

| Edge  | Cost | Include ?  | Status |
|-------|------|--|--------|
| 1 – 2 | 10   | Yes  | ✓      |
| 3 – 6 | 15   | Yes  | ✓      |
| 4 – 6 | 20   | Yes  | ✓      |
| 2 – 6 | 25   | Yes  | ✓      |
| 1 – 4 | 30   | No – Leads to Cycle                                    | ✗      |
| 3 – 5 | 35   | Yes  | ✓      |
| 2 – 5 | 40   | Not Considered as already all Nodes are covered in MST |        |
| 2 – 3 | 50   |  |        |
| 5 – 6 | 55   |  |        |

# Kruskal's Example 1



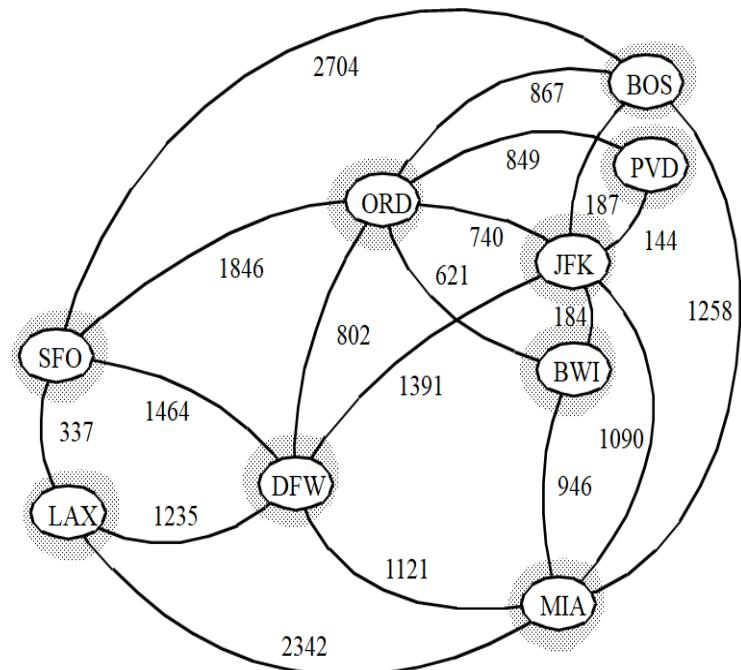
*Graph G*



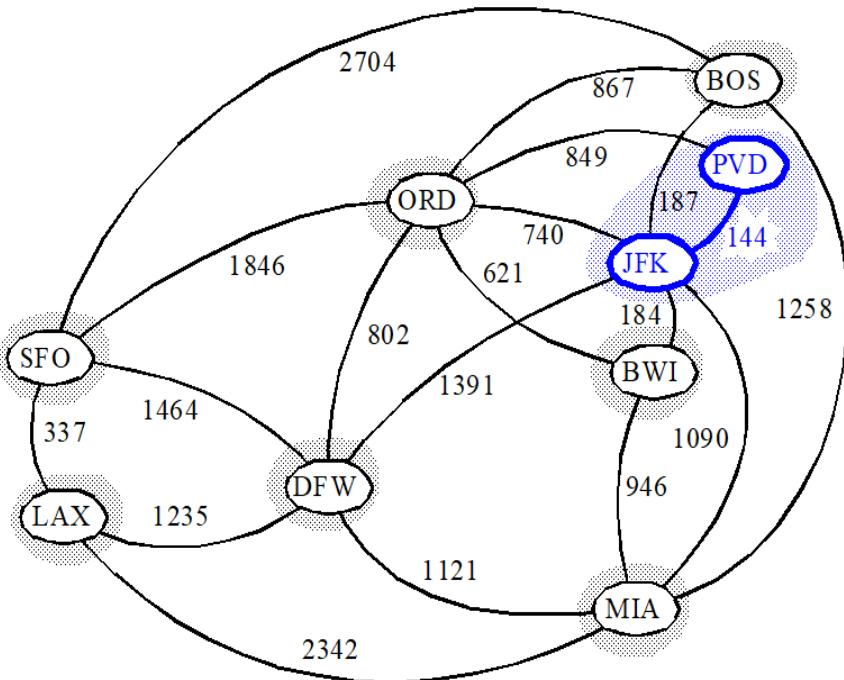
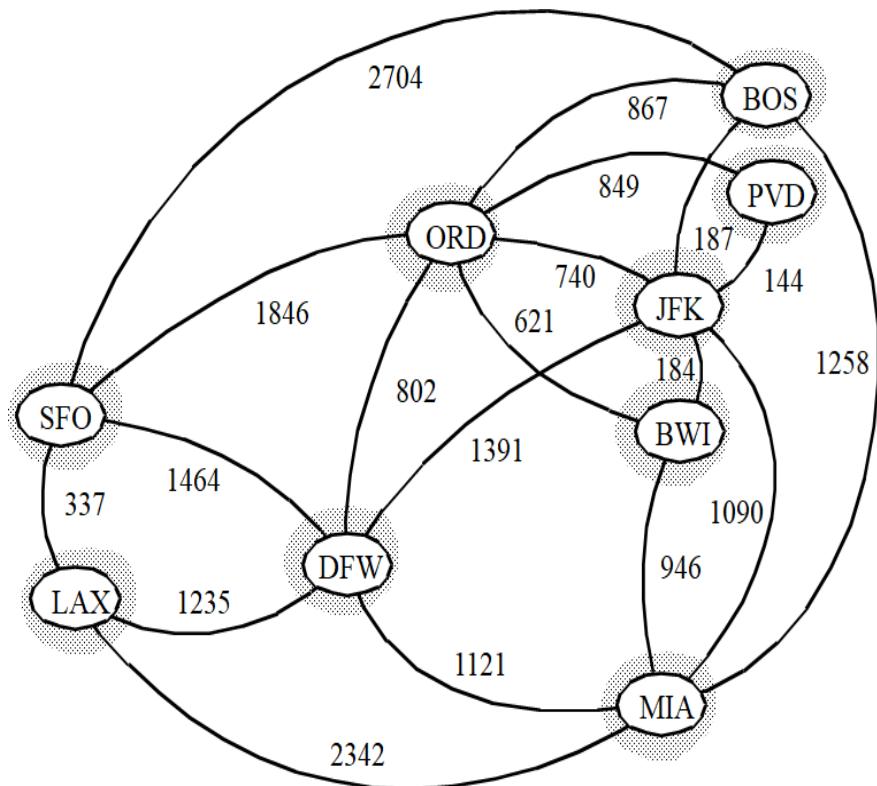
*MST with Cost =  $10+15+20+25+35 = 105$  units*

# Kruskal's Example 2

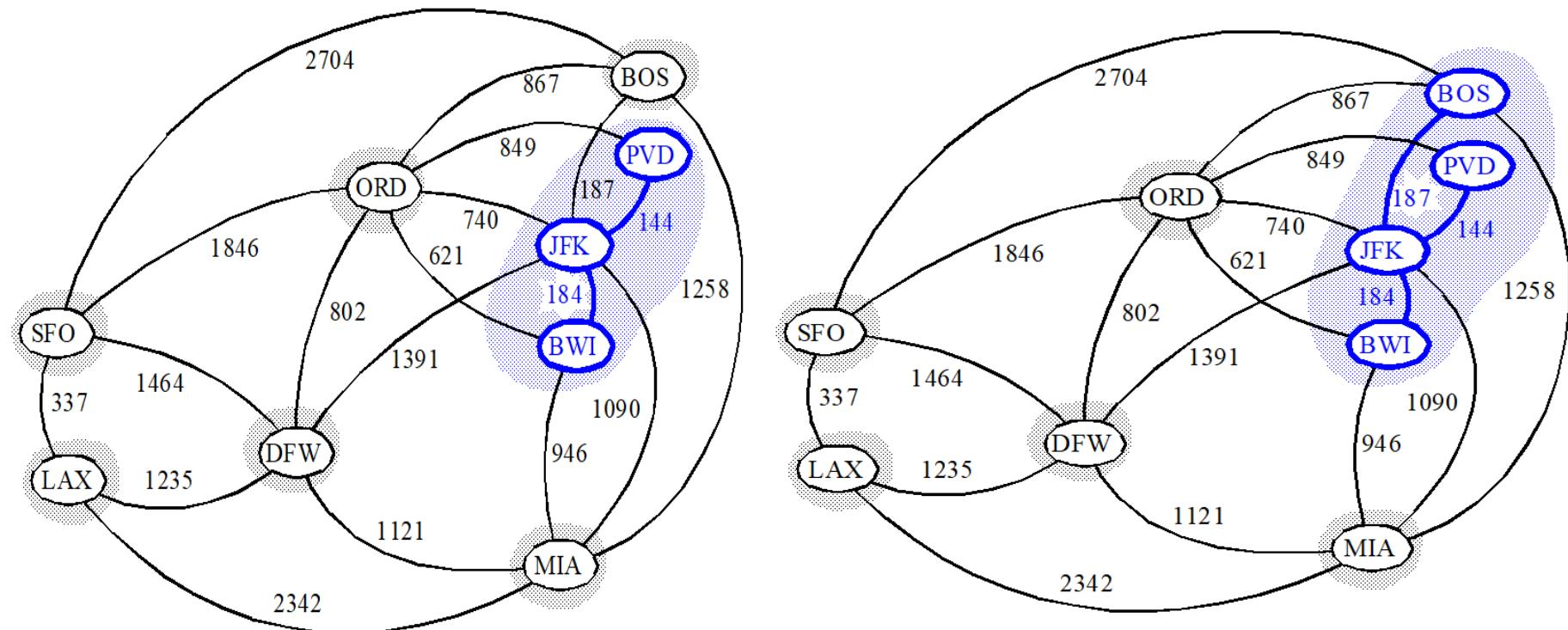
- Kruskal's algorithm is used to build the MST in *clusters*. Initially, each vertex is in its own cluster all by itself.
- The algorithm then considers each edge in turn, ordered by increasing weight.
- If an edge  $e$  connects two different clusters, then  $e$  is added to the set of edges of the MST, and the two clusters connected by  $e$  are merged into a single cluster!
- If, on the other hand,  $e$  connects two vertices that are already in same cluster, then  $e$  is discarded.
- Once the algorithm has added enough edges to form a spanning tree, it terminates and outputs this tree as the MST.



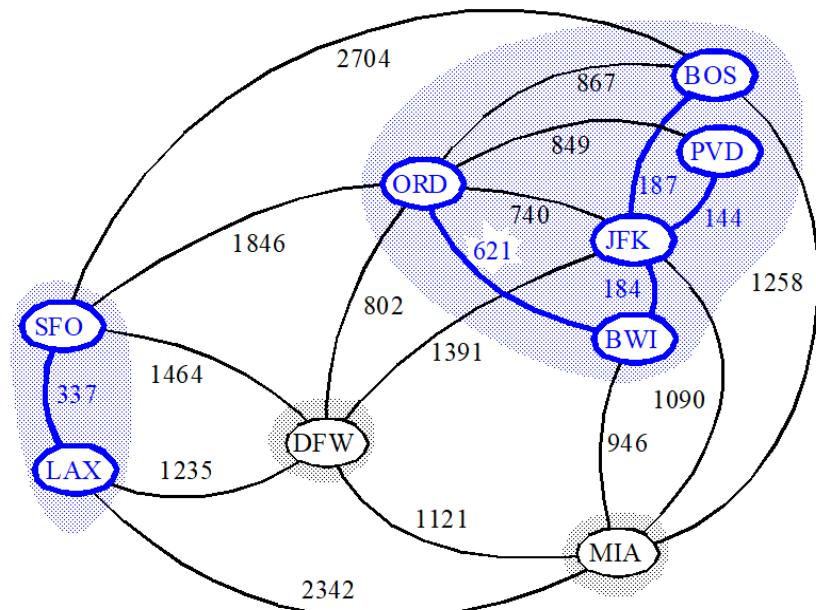
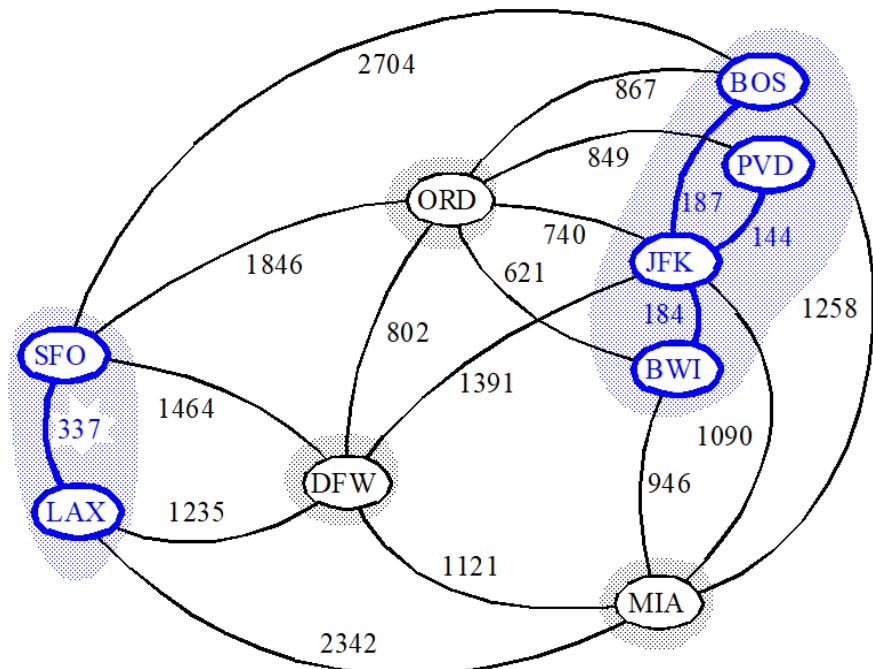
# Kruskal's Example 2



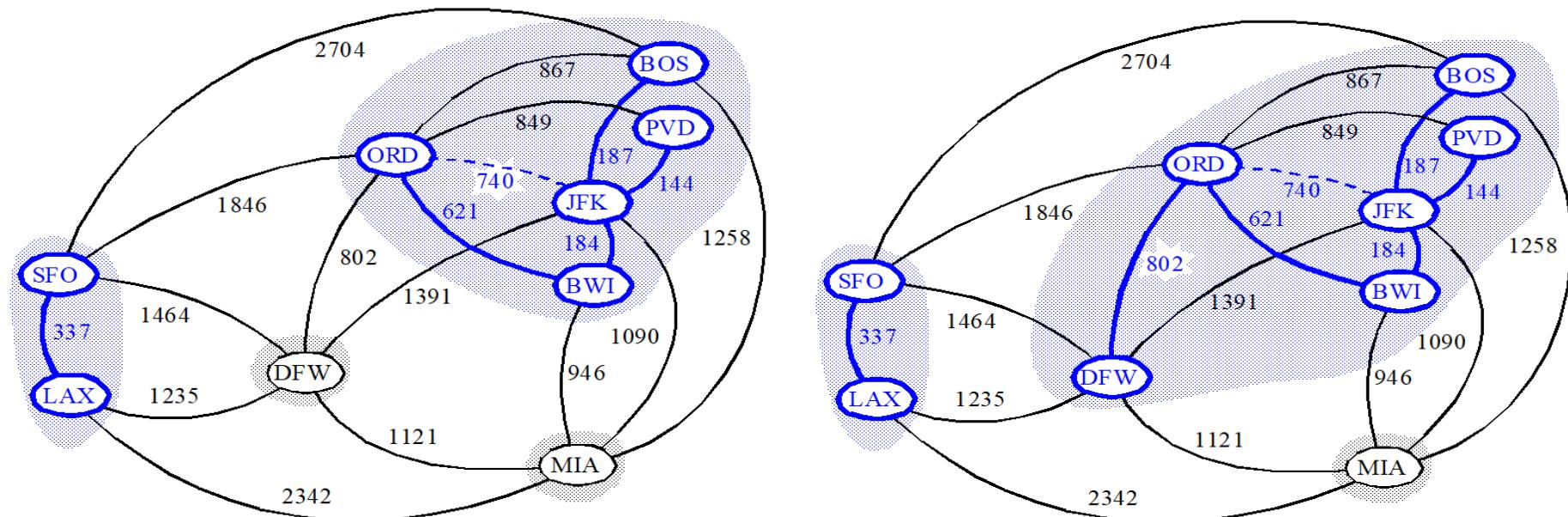
# Kruskal's Example 2



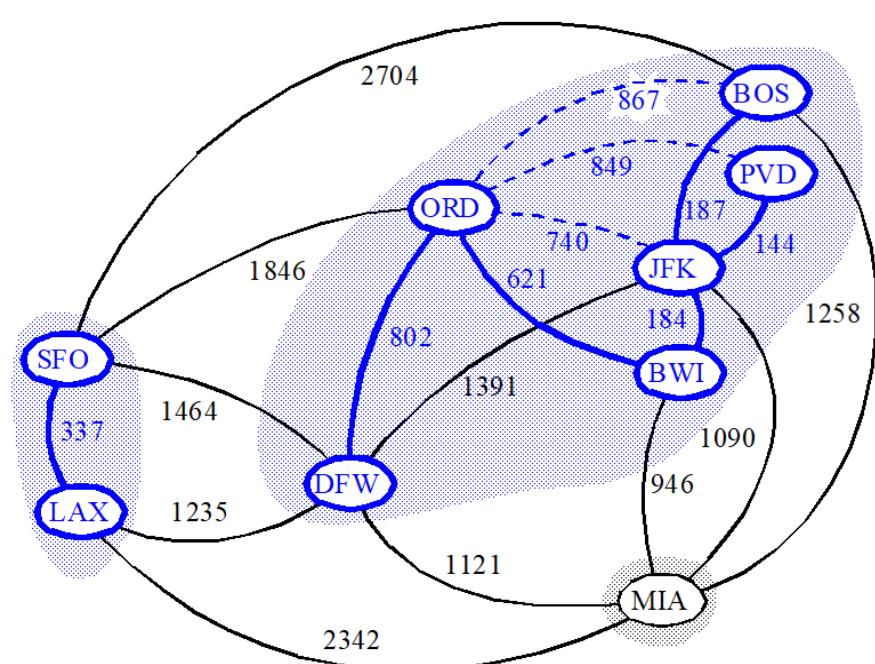
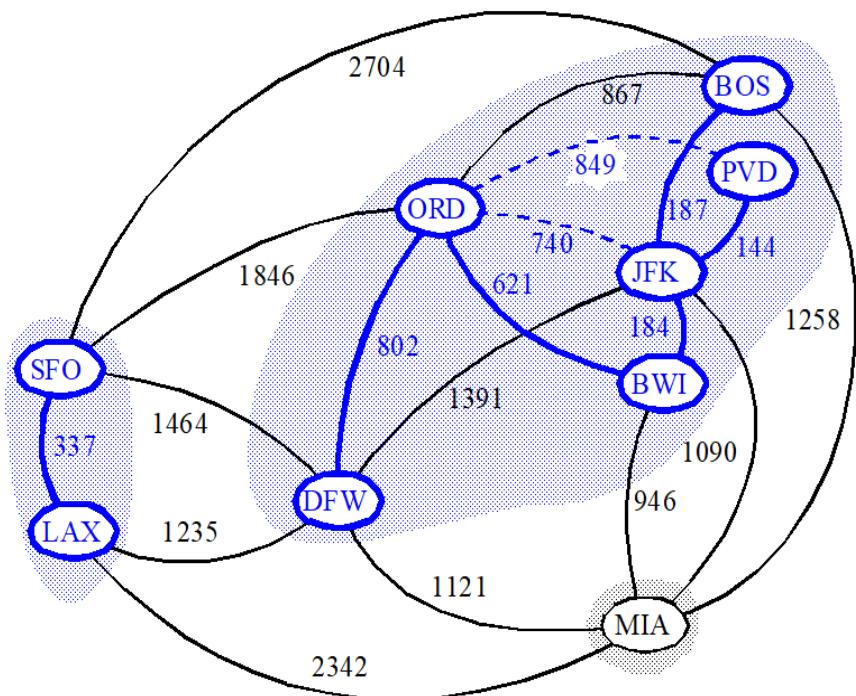
# Kruskal's Example 2



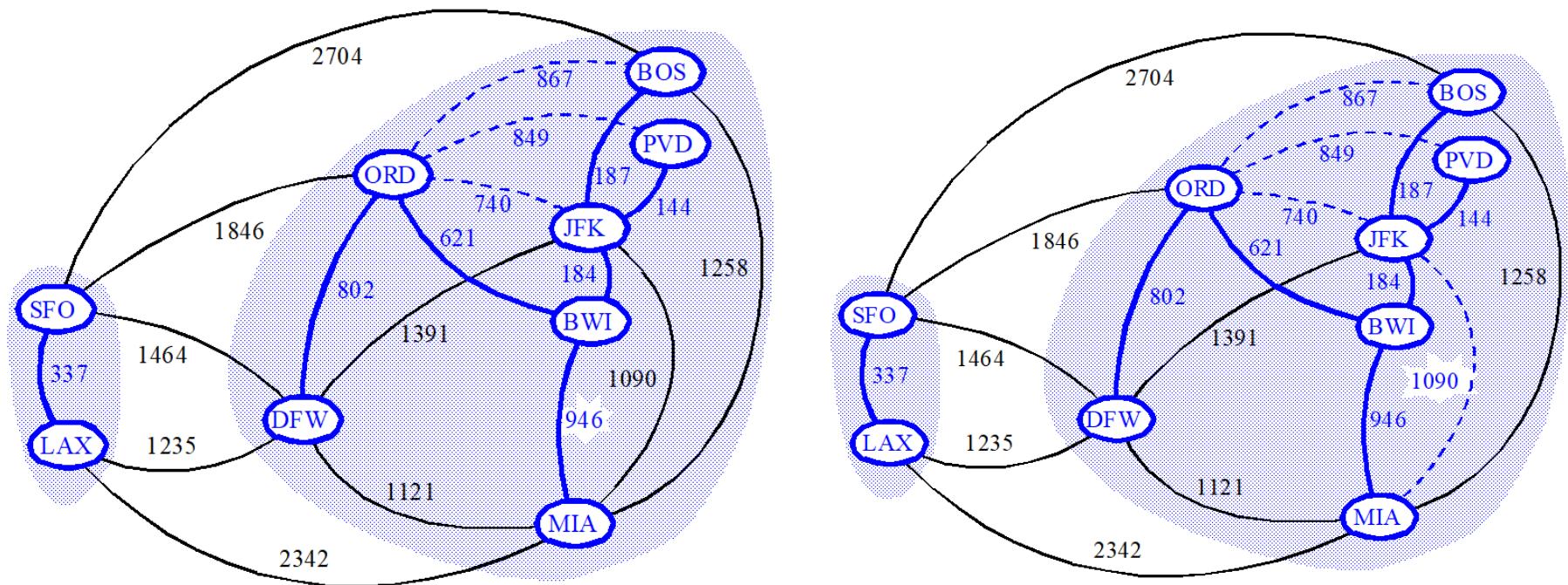
# Kruskal's Example 2



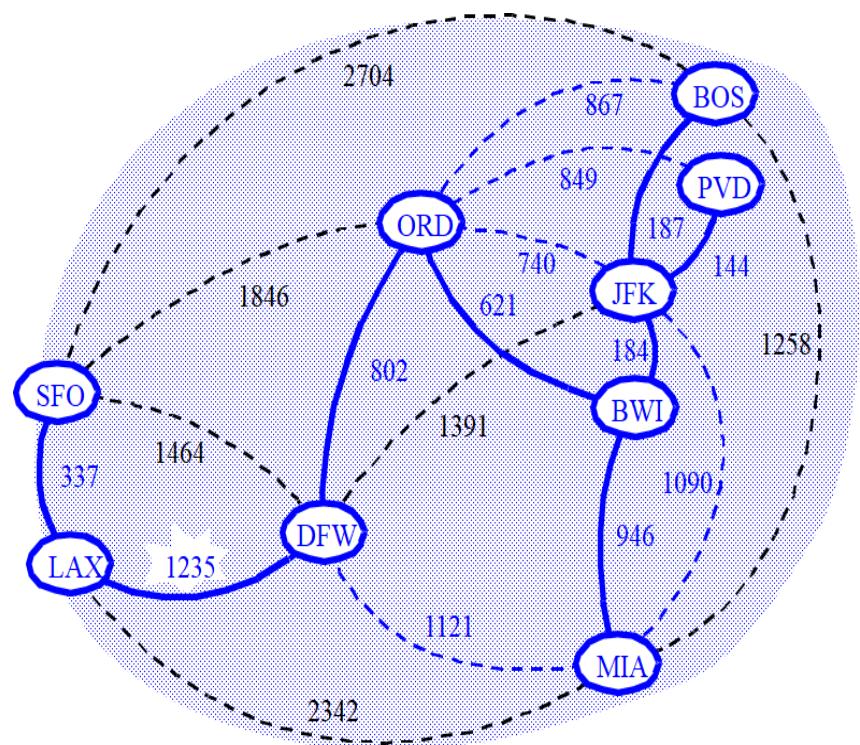
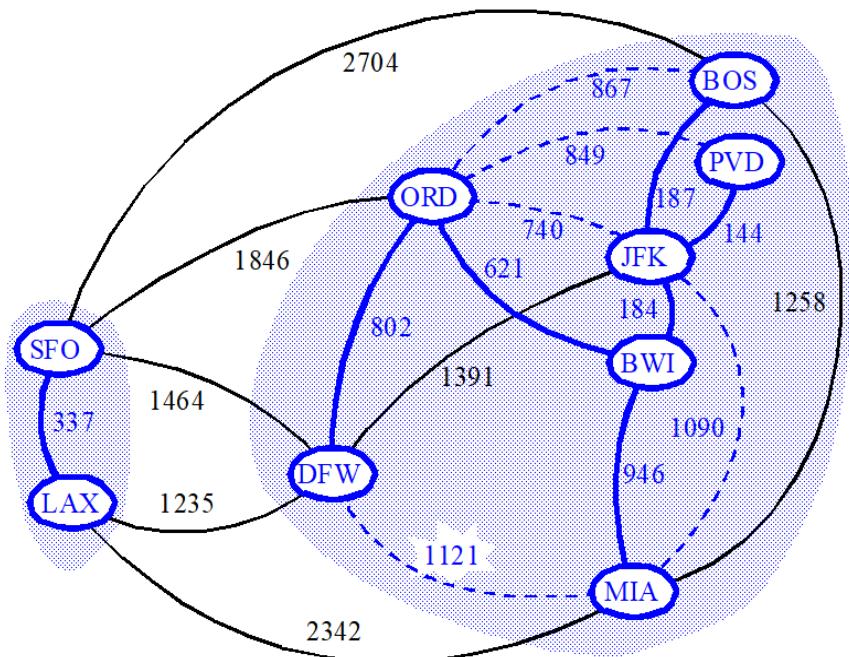
# Kruskal's Example 2



# Kruskal's Example 2



# Kruskal's Example 2



$$\begin{aligned}
 \text{MST Cost} &= 144 + 184 + 187 + 337 + 621 + 802 + 946 + 1235 \\
 &= 4456 \text{ units}
 \end{aligned}$$

# Kruskal's Algorithm

**Algorithm KruskalMST( $G$ ):**

**Input:** A simple connected weighted graph  $G$  with  $n$  vertices and  $m$  edges

**Output:** A minimum spanning tree  $T$  for  $G$

**for** each vertex  $v$  in  $G$  **do**

    Define an elementary cluster  $C(v) \leftarrow \{v\}$ .

Let  $Q$  be a priority queue storing the edges in  $G$ , using edge weights as keys

$T \leftarrow \emptyset$     //  $T$  will ultimately contain the edges of the MST

**while**  $T$  has fewer than  $n - 1$  edges **do**

$(u|v) \leftarrow Q.\text{removeMin}()$

    Let  $C(v)$  be the cluster containing  $v$

    Let  $C(u)$  be the cluster containing  $u$

**if**  $C(v) \neq C(u)$  **then**

        Add edge  $(v, u)$  to  $T$

        Merge  $C(v)$  and  $C(u)$  into one cluster, that is, union  $C(v)$  and  $C(u)$

**return** tree  $T$

$O(m \log n)$

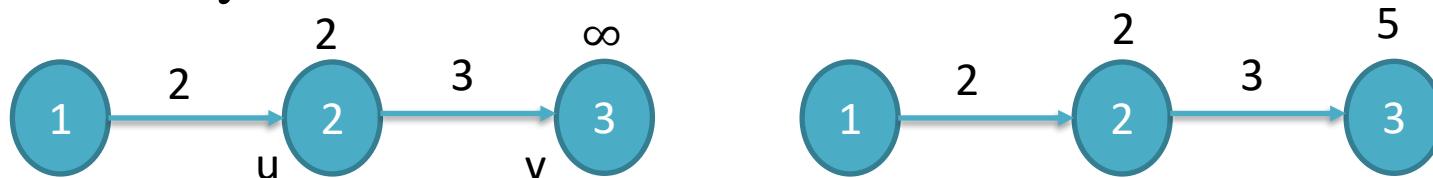
$O(n \log n)$

Explore!!

Total:  $O(m+n \log n) = O(m \log n)$

# Edge Relaxation

- Generally, we can solve all the shortest path problems by using edge relaxation. The *edge relaxation* is the operation to calculate the reaching cost to the vertex with the aim to lower the cost.
- Consider the below example:
- Here, initially the cost from  $1 \rightarrow 2$  is 2 and  $1 \rightarrow 3$  is  $\infty$



- Once 2 is visited (added into source), the cost from  $1 \rightarrow 3$  is reduced to 5 ( $2+3$ ) from  $\infty$ . This process is called edge relaxation! More formally,

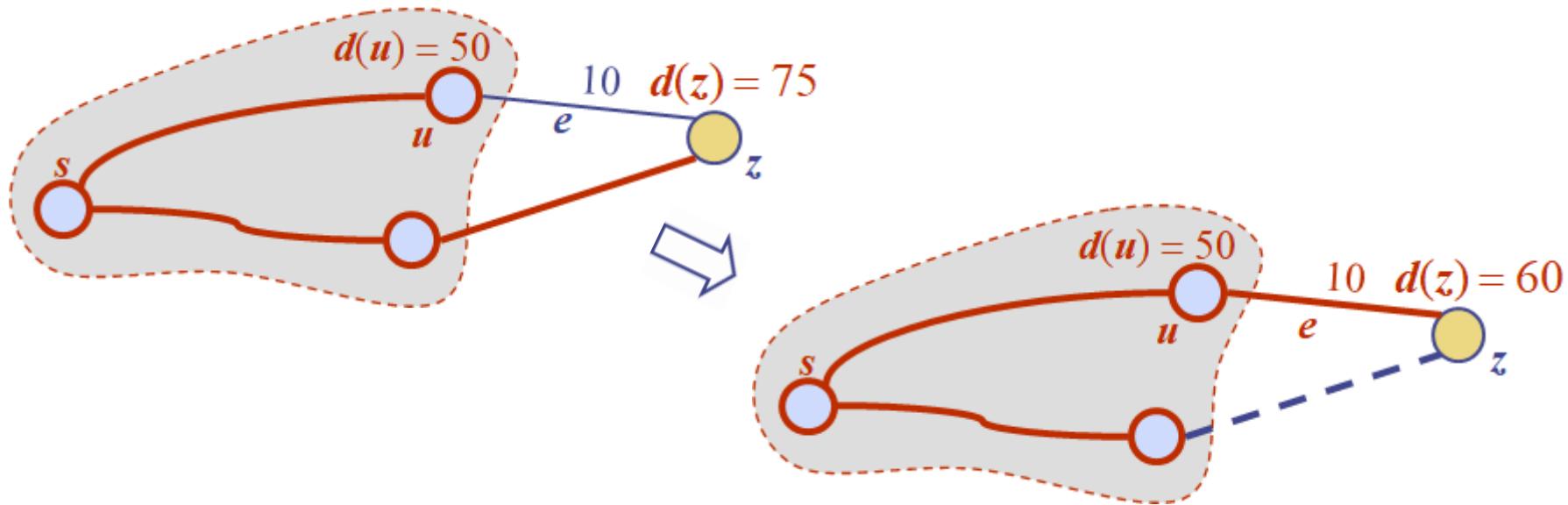
For the edge from the vertex  $u$  to the vertex  $v$ , if  $d[u] + w(u,v) < d[v]$  is satisfied, update  $d[v]$  to  $d[u] + w(u,v)$

If ( $d[u] + c(u,v) < d[v]$ )  
then  $d[v] = d[u] + c(u,v)$

# Edge Relaxation

- Consider an edge  $e = (u,z)$  such that
  - $u$  is the vertex most recently added to the cloud
  - $z$  is not in the cloud
- The relaxation of edge  $e$  updates distance  $d(z)$  as follows:

$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$

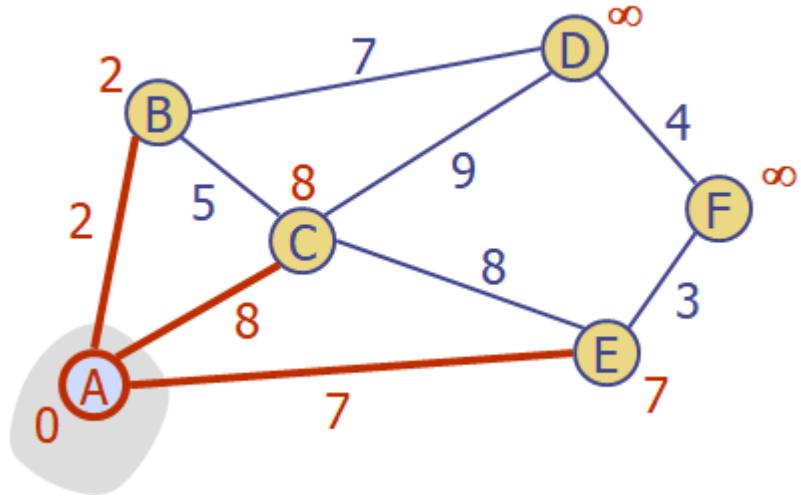


# Prim's Algorithm / Prim-Jarnik Algorithm

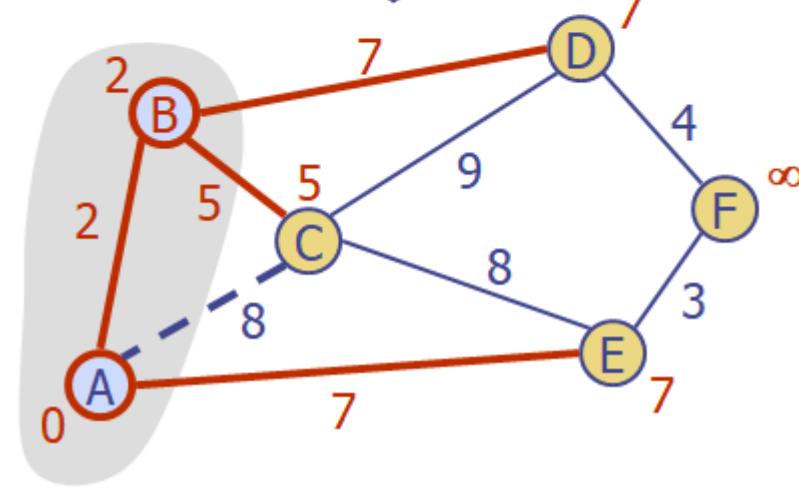
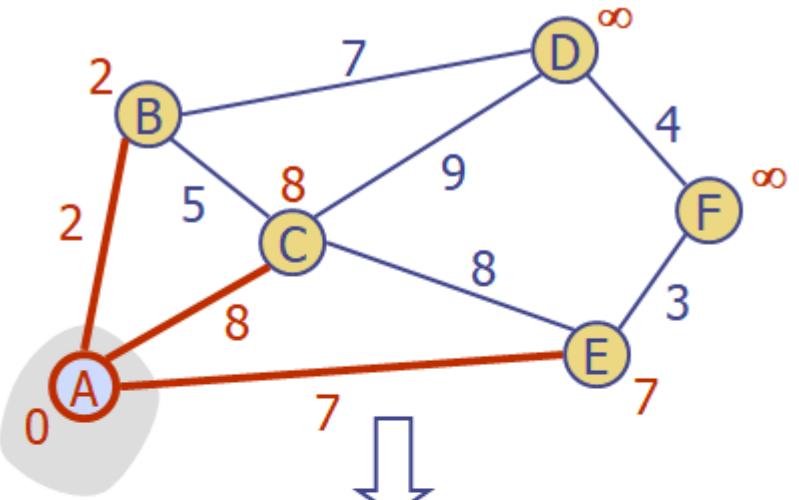


- Prim's algorithm is a greedy strategy used to obtain the MST from a graph edge by edge. The edge to be included is chosen according to the optimization criteria i.e., to choose an edge that results in a minimum increase in the sum of costs of the edges included so far.
- If A is the set of edges selected so far, then A forms a tree, the next edge  $(u, v)$  to be included in A is a minimum cost edge not in A, with the property that  $A \cup \{(u, v)\}$  is also a tree.
- Steps in Prim's Algorithm:
  - Step 01: Select any vertex (or alphabetically or given vertex) as the source.
  - Step 02: Compute the distances from the source to each other vertices of the graph.
  - Step 03: Choose the vertex which leads to a minimum increase in the sum of costs of the edges included. Add the vertex to the tree iff the addition does not lead to a cycle.
  - Step 04: Repeat the steps 2 and 3 until a tree is generated with all the vertices of the graph.

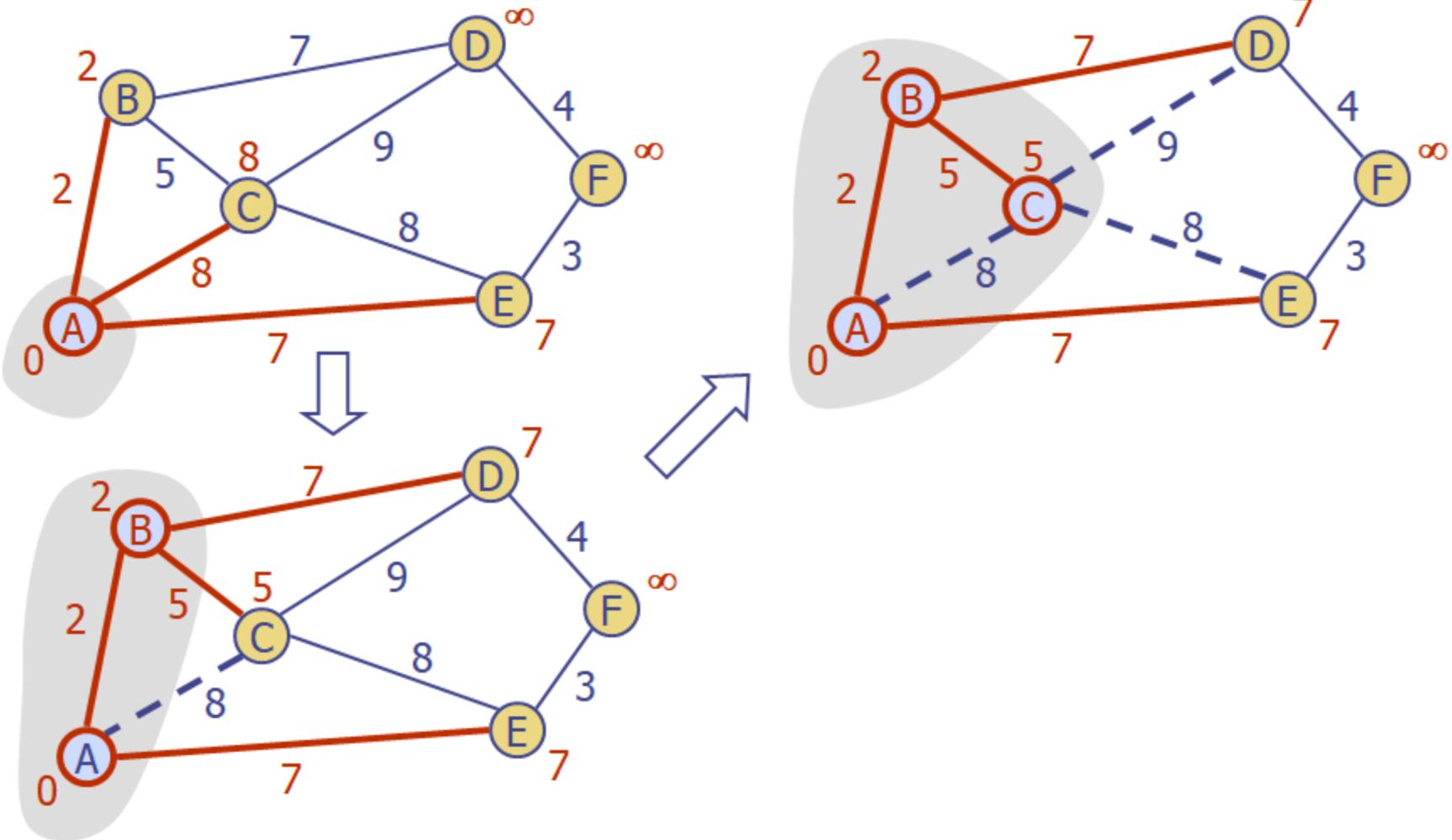
# Prim's Algorithm Example



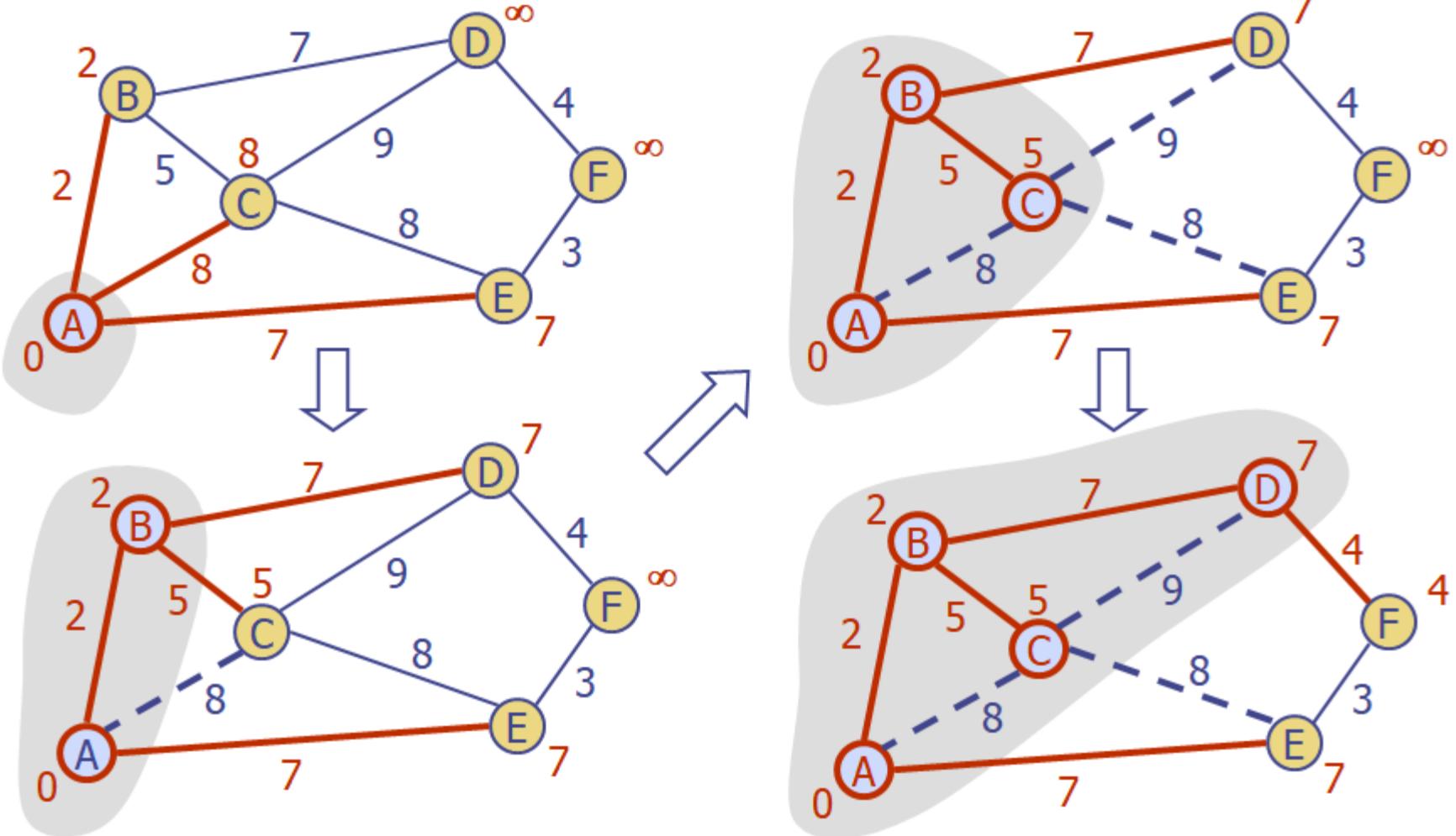
# Prim's Algorithm Example



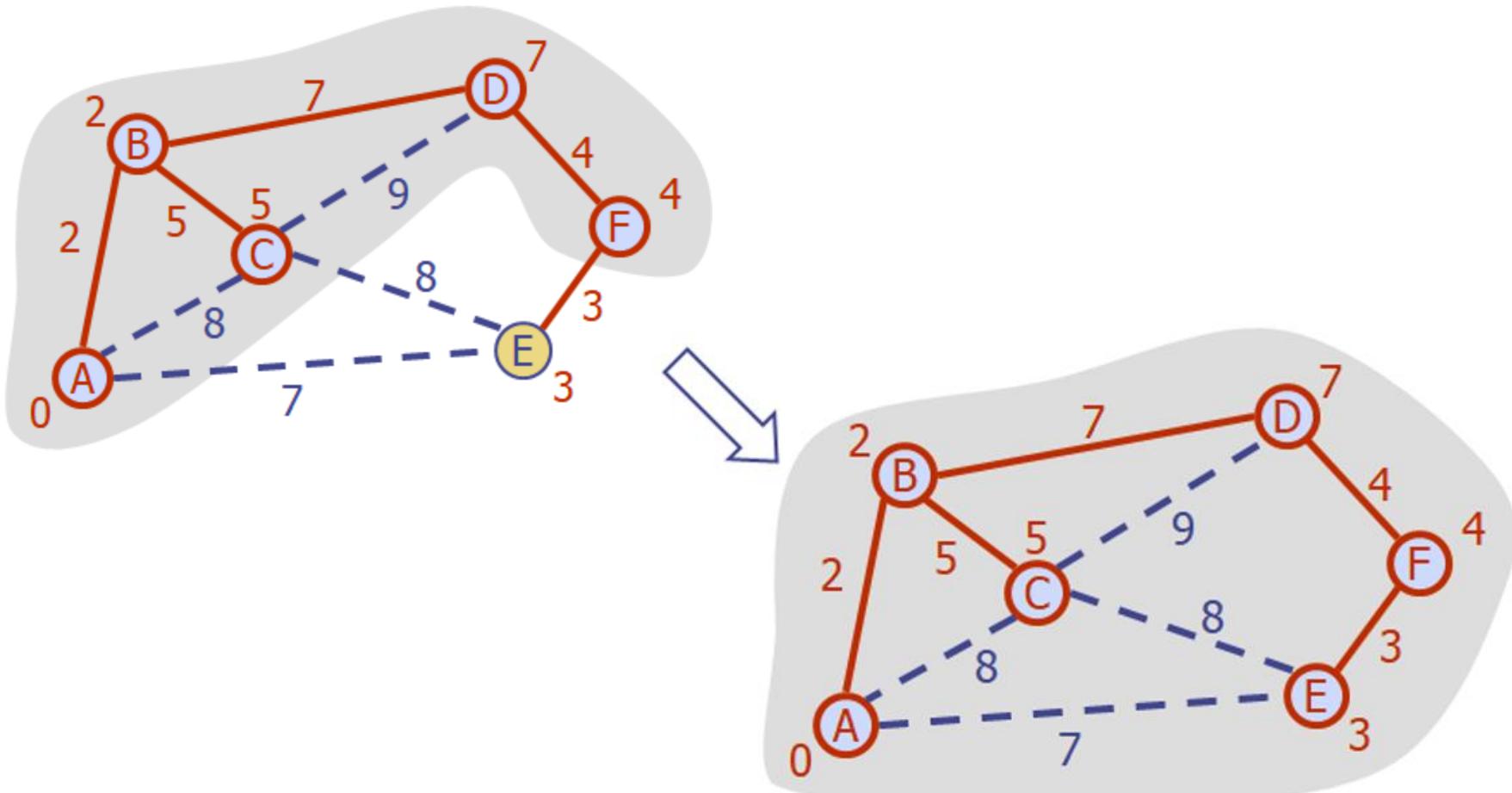
# Prim's Algorithm Example



# Prim's Algorithm Example



# Prim's Algorithm Example



# Prim's / PrimJarník Algorithm

**Algorithm** PrimJarníkMST( $G$ ):

**Input:** A weighted connected graph  $G$  with  $n$  vertices and  $m$  edges

**Output:** A minimum spanning tree  $T$  for  $G$

Pick any vertex  $v$  of  $G$

$D[v] \leftarrow 0$

**for** each vertex  $u \neq v$  **do**

$D[u] \leftarrow +\infty$

Initialize  $T \leftarrow \emptyset$ .

Initialize a priority queue  $Q$  with an item  $((u, \text{null}), D[u])$  for each vertex  $u$ , where  $(u, \text{null})$  is the element and  $D[u]$  is the key.

**while**  $Q$  is not empty **do**

$(u, e) \leftarrow Q.\text{removeMin}()$

Add vertex  $u$  and edge  $e$  to  $T$ .

Total:  $O(m+n \log n) = O(m \log n)$

**for** each vertex  $z$  adjacent to  $u$  such that  $z$  is in  $Q$  **do**

// perform the relaxation procedure on edge  $(u, z)$

**if**  $w((u, z)) < D[z]$  **then**

$D[z] \leftarrow w((u, z))$

Change to  $(z, (u, z))$  the element of vertex  $z$  in  $Q$ .

Change to  $D[z]$  the key of vertex  $z$  in  $Q$ .

**return** the tree  $T$

# Kruskal vs Prim's Algorithm

| Prims Algorithm   | Kruskal Algorithm   |
|---|---|
| It starts to build the MST from any of the Node.  | It starts to build the MST from Minimum weighted vertex in the graph.                     |
| Adjacency Matrix, Binary Heap or Fibonacci Heap is used in Prims algorithm                    | Disjoint Set is used in Kruskal Algorithm.  |
| Prims Algorithm runs faster in dense graphs   | Kruskal Algorithm runs faster in sparse graphs  |
| Time Complexity is $O(EV \log V)$ with binary heap and $O(E + V \log V)$ with fibonacci heap. | Time Complexity is <b><math>O(E \log V)</math></b>  |
| The next Node included must be connected with the node we traverse                            | The next edge included may or may not be connected but should not form the cycle.         |
| It traverses the one node several times in order to get its minimum distance                  | It traverses the edge only once and based on cycle it will either reject it or accept it, |
| Greedy Algorithm  | Greedy Algorithm  |

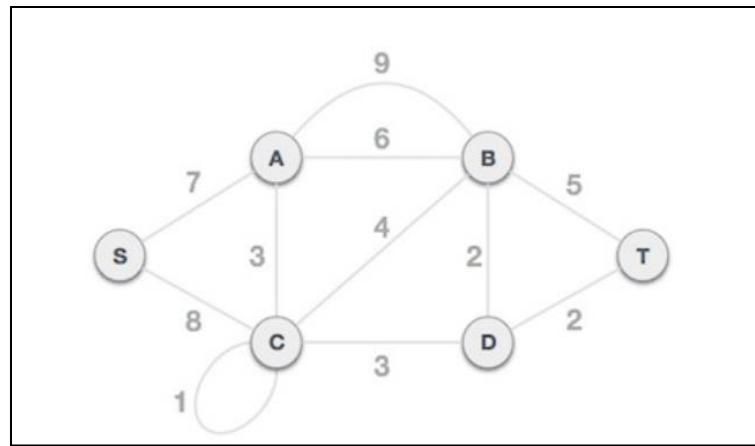
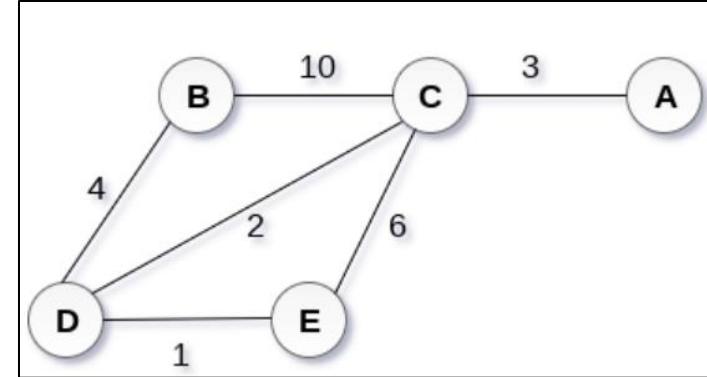
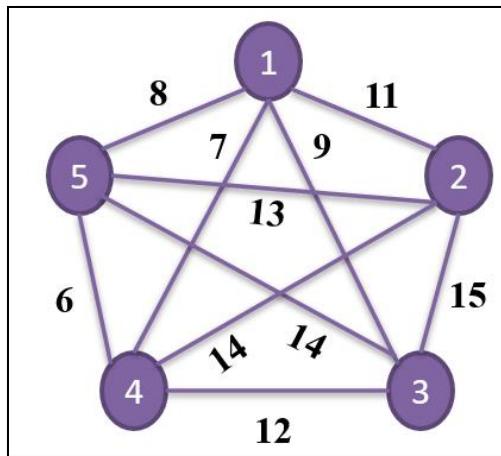
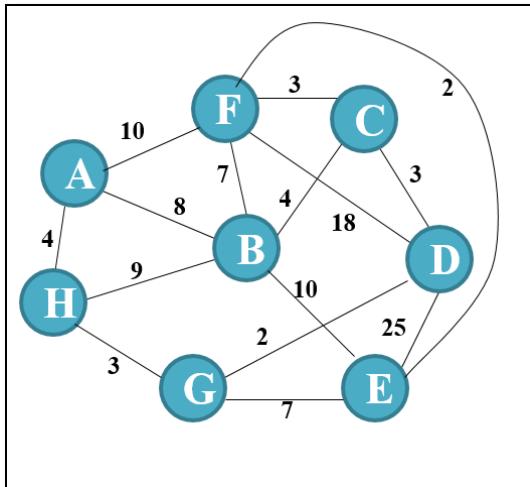
# Exercises 1

---

1. For the fractional knapsack problem,  $n=3, m=20$ ,  $(p_1, p_2, p_3) = (25, 24, 15)$  and  $(w_1, w_2, w_3) = (18, 15, 10)$ , find the feasible and optimal solution.
2. Consider a fractional knapsack problem of finding the optimal solution, where  $m=15$ ,  $(p_1, p_2, \dots, p_7) = (10, 5, 15, 7, 6, 18, 3)$  and  $(w_1, w_2, \dots, w_7) = (2, 3, 5, 7, 1, 4, 1)$ .

# Exercises Set - 2

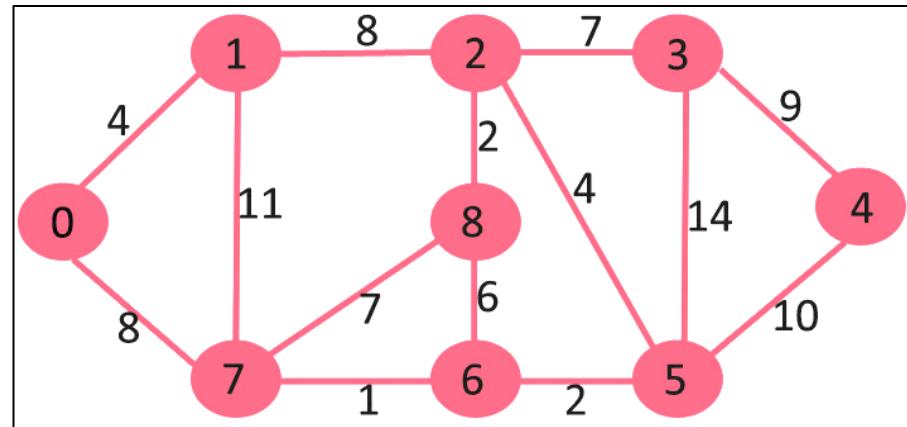
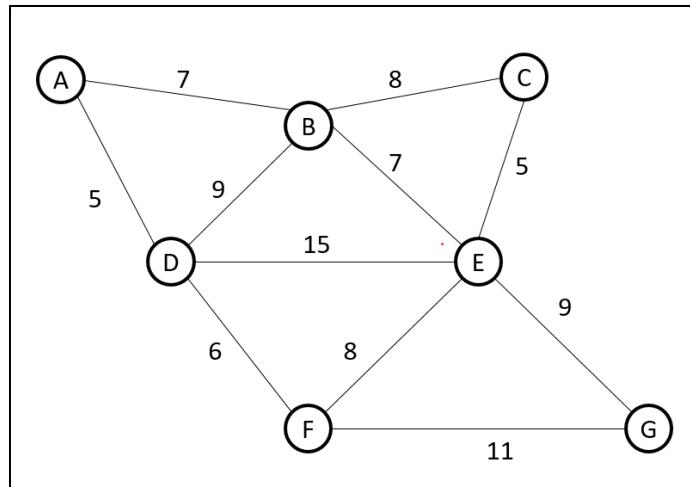
1) Apply Prim's algorithm and Kruskal's algorithm on the given graphs.



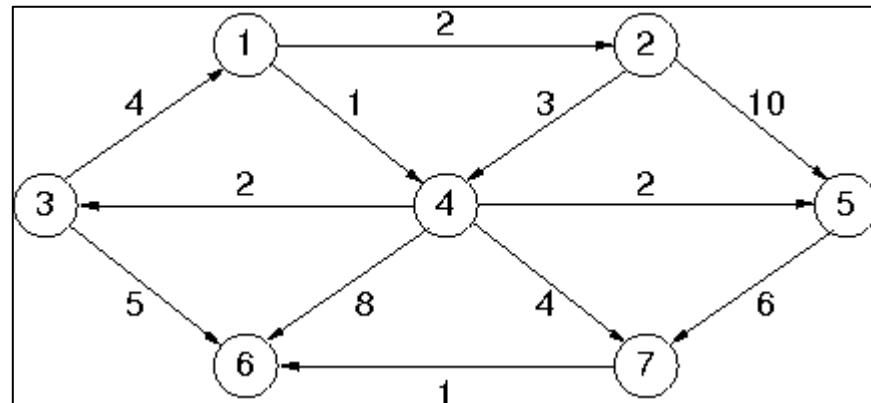
2. Both MST algorithms presented assumed to use Adjacency List. Investigate will the complexity change changing this to adjacency matrix? Explain.
3. How do you adapt algorithms for finding MST to find k-clusters? Explain
4. Assume you have a graph G with a negative weighted edge(s). Will the algorithms for MST covered in this class still work? What about directed edges. Explain. Hint: Refer Appendix of this slide.

# Exercises Set 3

Consider A, 0, 3 as Source vertex in each of the below Graphs and Apply Dijkstra's Algorithm :

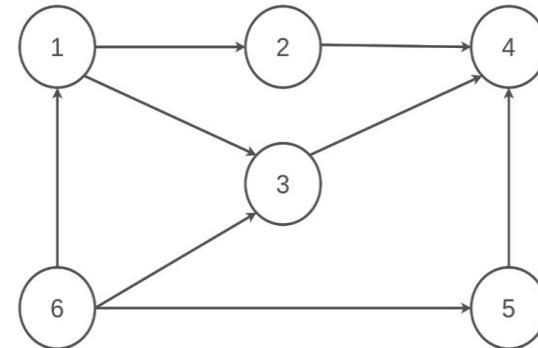


Hint for Undirected Graph – Dijkstra's :  
 In above,  $A \rightarrow B$  and  $B \rightarrow A$  is 7  
 Basically, You can convert each undirected edge into 2 direct edges ☺

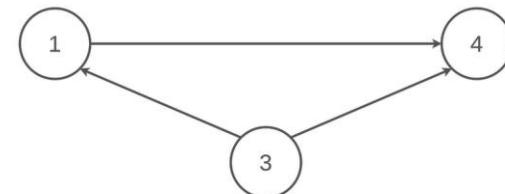


# Appendix: Why Prim's & Kruskal's Fail for Directed Graphs ?

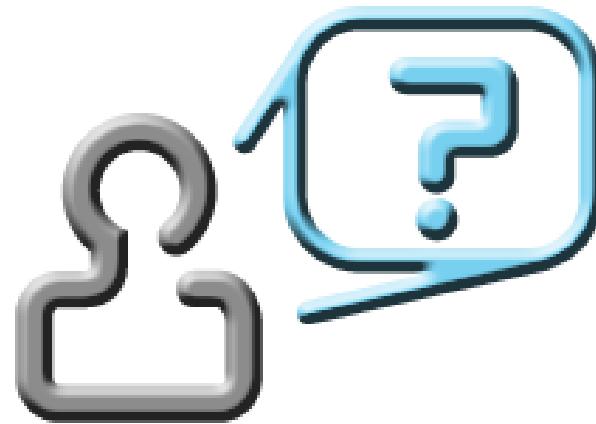
- Prim's algorithm assumes that all vertices are connected. But in a directed graph, every node need not be reachable from every other node. Hence, Prim's algorithm fails due to this reason.
- In Kruskal's algorithm, In each step, it is checked that if the edges form a cycle with the spanning-tree formed so far. But Kruskal's algorithm fails to detect the cycles in a directed graph as there are cases when there is no cycle between the vertices but Kruskal's Algorithm assumes it to be a cycle and doesn't take consider some edges into account due to which Kruskal's Algorithm fails for directed graph.
- Edmonds' algorithm can be used for finding MST for a digraph. {Not in the scope of this course}



As No node is reachable from node 4 and Prim's Algorithm assumes that every node is reachable due to which Prim's Algorithm fails for directed graph



As There is no cycle in this directed graph but Kruskal's Algorithm assumes it a cycle by union-find method due to which Kruskal's Algorithm fails for directed graph



*We will Greedily learn D&C in the next class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : CC BY-NC-SA 4.0





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

## DSECLZG519

Parthasarathy





# Contact Session #12

# DSECLZG519 – Divide & Conquer

# Agenda for Session # 12

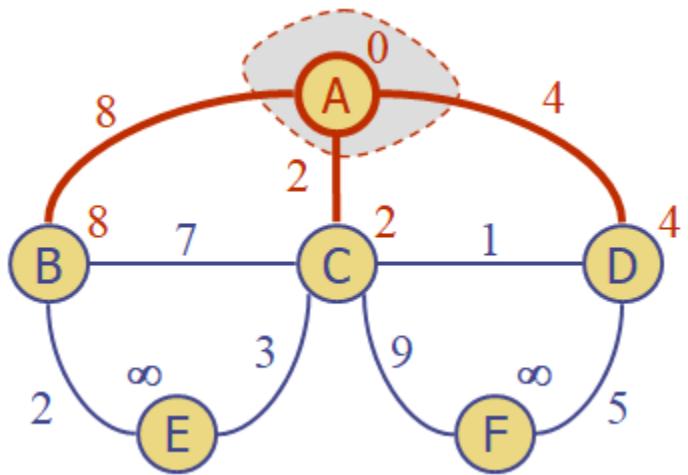
---

- Recap of CS#11
- Single source shortest path
- Divide & Conquer
- Merge Sort
- Quick Sort
- Integer Multiplication Problem (Karatsuba)
- Exercises

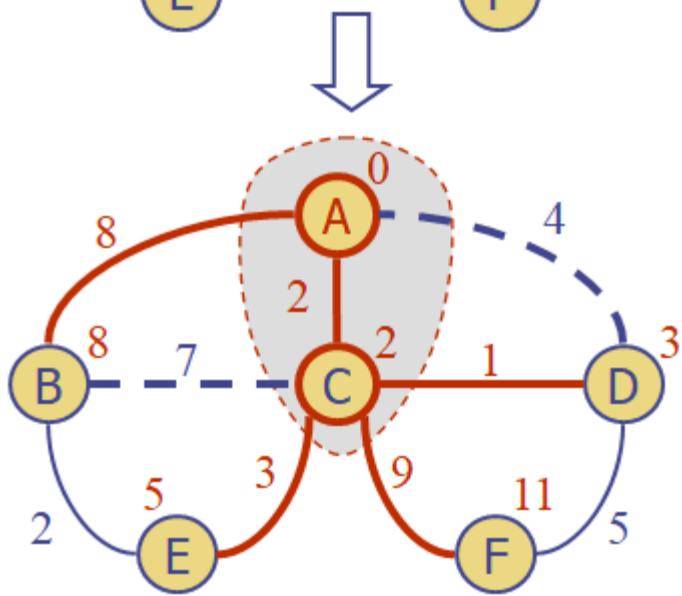
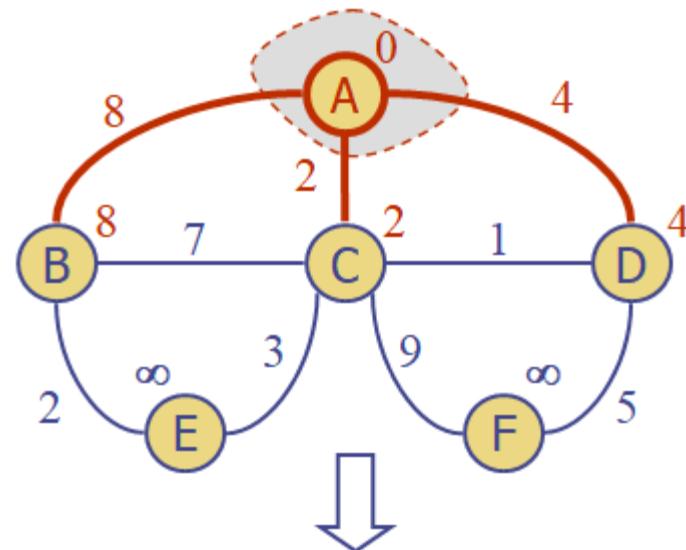
# Single Source Shortest Path

- The aim of the single source shortest path problem is to find the shortest path between a selected source and every other vertex in the graph. To solve this problem, Edsger-w-Dijkstra has devised an amazing algorithm in the 1950's which is still popular and widely used.
- The problem can be stated as “For a given weighted graph  $G$  with vertices, a shortest path has to be computed from a given vertex  $v$  (called source) to all the other vertices”.
- The shortest path computed can either be a direct path or an indirect path.
- To formulate a greedy paradigm solution for this problem, a multistage solution which is optimal in every stage has to be devised.
- First the shortest path from the starting vertex to the nearest vertex has to be computed, then the next shortest and so on.

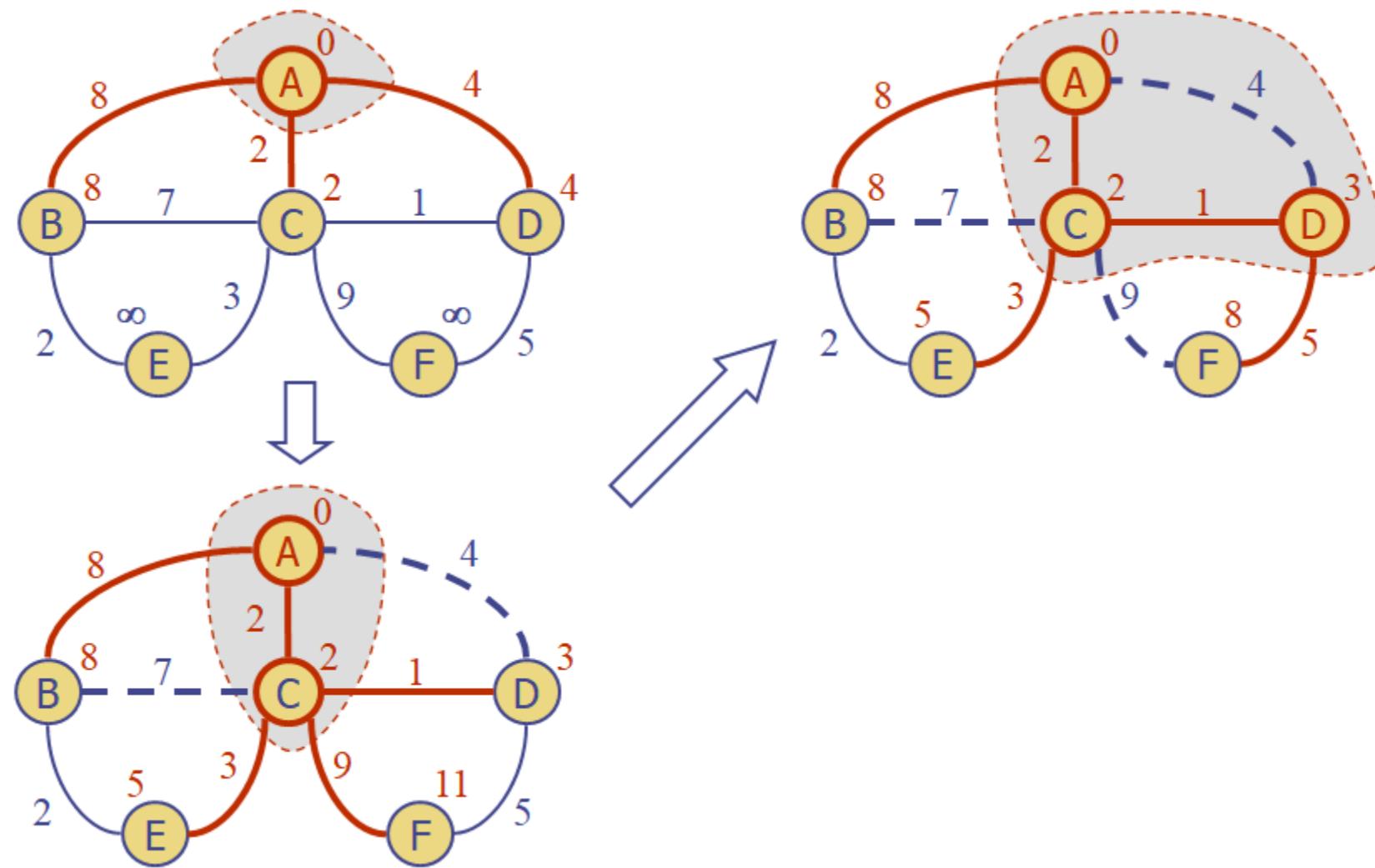
# Dijkstra's Algorithm Example 1



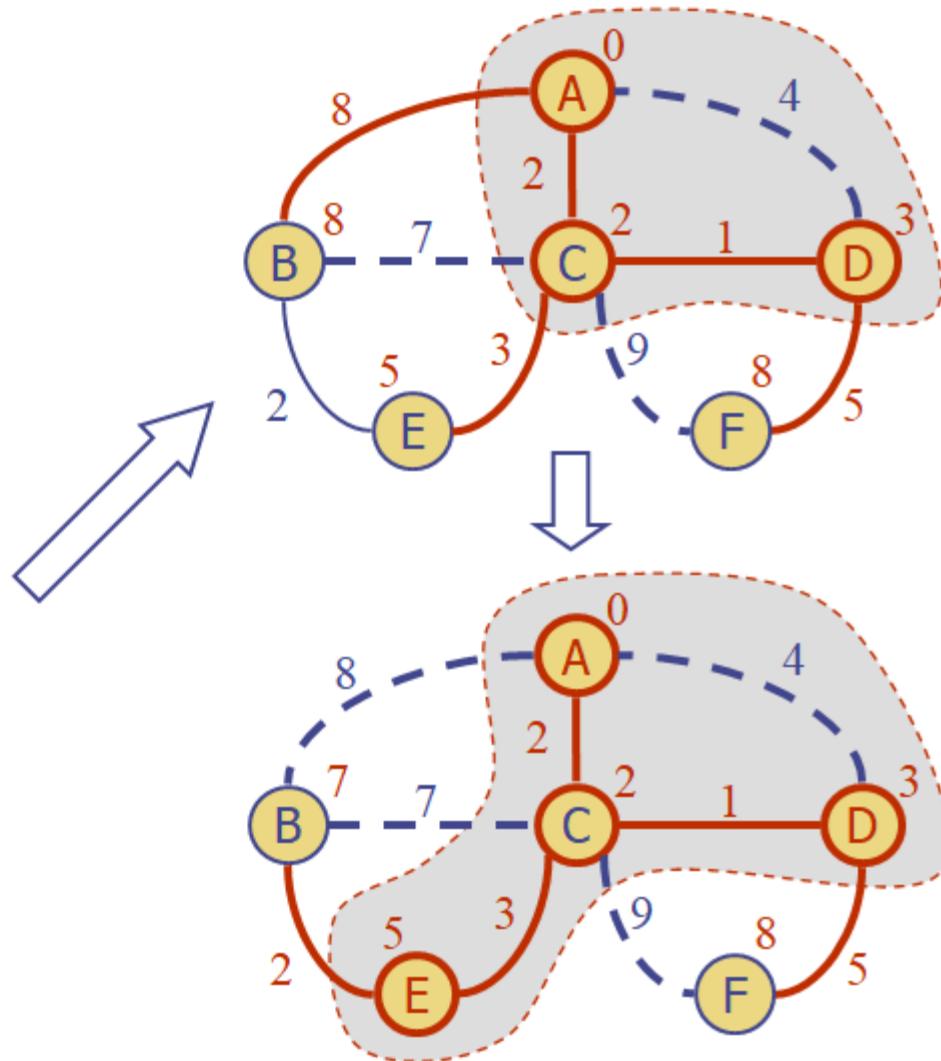
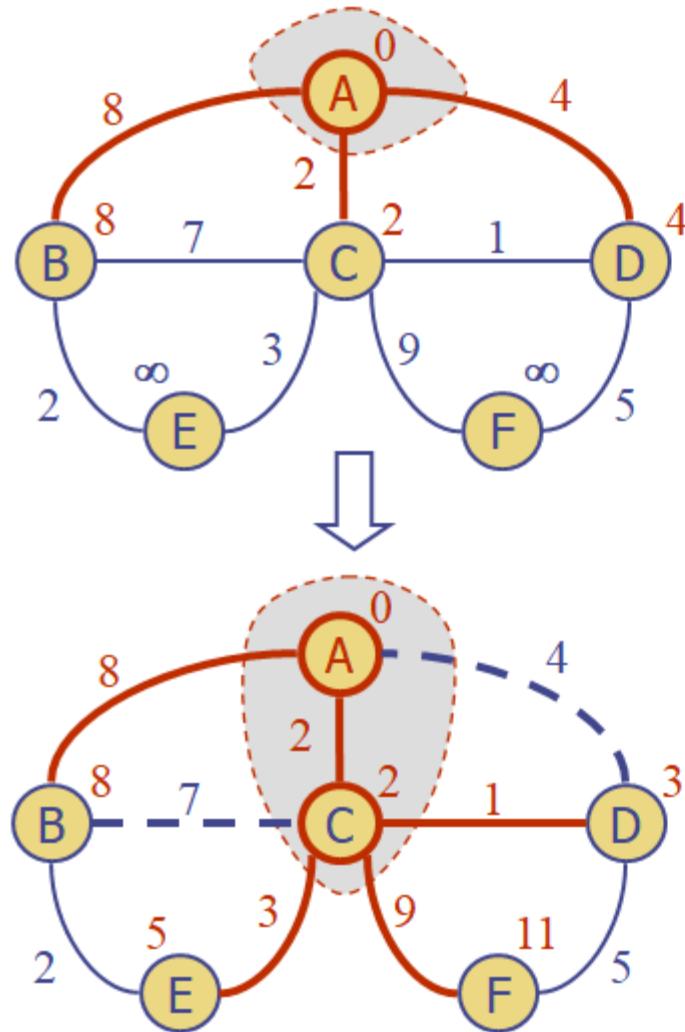
# Dijkstra's Algorithm Example 1



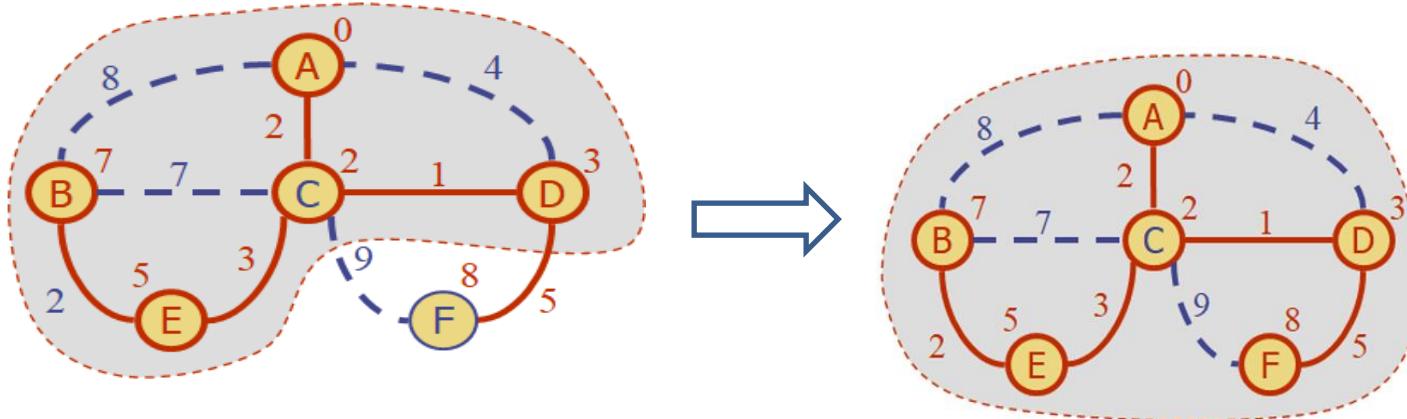
# Dijkstra's Algorithm Example 1



# Dijkstra's Algorithm Example 1



# Dijkstra's Algorithm Example 1



*The Optimal Solution or Shortest Path from Source Vertex (A) is :*

| Path              | Shortest Path                                 | Shortest Cost/Distance |
|-------------------|---|------------------------|
| $A \rightarrow A$ | -   | 0                      |
| $A \rightarrow B$ | $A \rightarrow C \rightarrow E \rightarrow B$ | 7                      |
| $A \rightarrow C$ | $A \rightarrow C$                             | 2                      |
| $A \rightarrow D$ | $A \rightarrow C \rightarrow D$               | 3                      |
| $A \rightarrow E$ | $A \rightarrow C \rightarrow E$               | 5                      |
| $A \rightarrow F$ | $A \rightarrow C \rightarrow D \rightarrow F$ | 8                      |

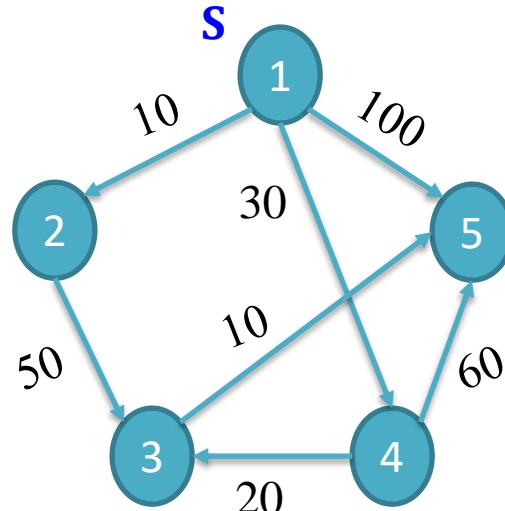
# Dijkstra's Algorithm

- Dijkstra's Algorithm will work for both Directed and undirected graphs but only when the weights are positive!

Procedure:

- **Step 01:** Construct the cost adjacency matrix for the given graph.
- **Step 02:** Assume a vertex as the source (alphabetically) (if source is not mentioned) and compute the distance from the source to all other vertices as  $D[w]=c(s,w)$  or  $c(s,u)+c(u,w)$  i.e. direct distance or indirect distance. This is also known as relaxation.
- **Step 03:** Pick the shortest path of the computed distance.
- **Step 04:** The vertex causing the shorted path is also included into the source.
- **Step 05:** Repeat the steps till all shortest paths are evaluated.

# Dijkstra's Algorithm Example 2



Cost Adjacency Matrix

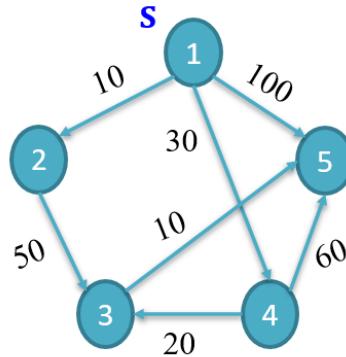
|   | 1        | 2        | 3        | 4        | 5        |
|---|----------|----------|----------|----------|----------|
| 1 | 0        | 10       | $\infty$ | 30       | 100      |
| 2 | $\infty$ | 0        | 50       | $\infty$ | $\infty$ |
| 3 | $\infty$ | $\infty$ | 0        | $\infty$ | 10       |
| 4 | $\infty$ | $\infty$ | 20       | 0        | 60       |
| 5 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

Step 01 :

| S (Source) | V = V - S | Distance D[v]            | u | D[u] |
|------------|-----------|--------------------------|---|------|
| 1          | 2         | D[2] = C(1,2) = 10 ✓     |   |      |
|            | 3         | D[3] = C(1,3) = $\infty$ |   |      |
|            | 4         | D[4] = C(1,4) = 30       | 2 | 10   |
|            | 5         | D[5] = C(1,5) = 100      |   |      |

Add u (which is 2) to the source and continue ...

# Dijkstra's Algorithm Example 2



Cost Adjacency Matrix

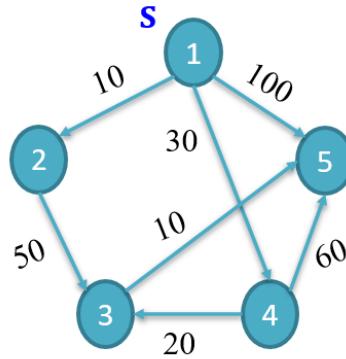
|   | 1        | 2        | 3        | 4        | 5        |
|---|----------|----------|----------|----------|----------|
| 1 | 0        | 10       | $\infty$ | 30       | 100      |
| 2 | $\infty$ | 0        | 50       | $\infty$ | $\infty$ |
| 3 | $\infty$ | $\infty$ | 0        | $\infty$ | 10       |
| 4 | $\infty$ | $\infty$ | 20       | 0        | 60       |
| 5 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

Step 02 :

| S<br>(Source) | V = V - S   | Distance D[v]  | U | D[u] |
|---------------|-------------|--|---|------|
| 1,2           | 3<br>4<br>5 | D[3] = min{prev value, C(1,2)+C(2,3)}=min( $\infty$ , 10 + 50) = 60<br>D[4] = min{prev value, C(1,2)+C(2,4)}=min(30,10+ $\infty$ ) = 30 ✓<br>D[5] =min{prev value, C(1,2)+C(2,5)}=min(100,10+ $\infty$ ) = 100 | 4 | 30   |

Add u (which is 4) to the source and continue ...

# Dijkstra's Algorithm Example 2



Cost Adjacency Matrix

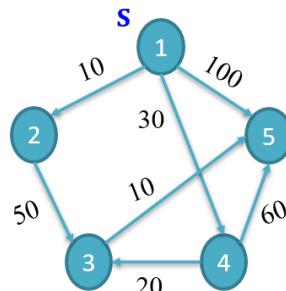
|   | 1        | 2        | 3        | 4        | 5        |
|---|----------|----------|----------|----------|----------|
| 1 | 0        | 10       | $\infty$ | 30       | 100      |
| 2 | $\infty$ | 0        | 50       | $\infty$ | $\infty$ |
| 3 | $\infty$ | $\infty$ | 0        | $\infty$ | 10       |
| 4 | $\infty$ | $\infty$ | 20       | 0        | 60       |
| 5 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

Step 03 :

| S<br>(Source) | V = V - S | Distance D[v]  | U | D[u] |
|---------------|-----------|--|---|------|
| 1,2,4         | 3<br>5    | $D[3] = \min\{\text{prev value}, C(1,4)+C(4,3)\} = \min(60, 30 + 20) = 50 \checkmark$<br>$D[5] = \min\{\text{prev value}, C(1,4)+C(4,5)\} = \min(100, 30 + 60) = 90$ | 3 | 50   |

Add u (which is 3) to the source and continue ...

# Dijkstra's Algorithm Example 2



Cost Adjacency Matrix

|   | 1        | 2        | 3        | 4        | 5        |
|---|----------|----------|----------|----------|----------|
| 1 | 0        | 10       | $\infty$ | 30       | 100      |
| 2 | $\infty$ | 0        | 50       | $\infty$ | $\infty$ |
| 3 | $\infty$ | $\infty$ | 0        | $\infty$ | 10       |
| 4 | $\infty$ | $\infty$ | 20       | 0        | 60       |
| 5 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

Step 04 :

| S<br>(Source) | V = V - S | Distance D[v]  | U | D[u] |
|---------------|-----------|--|---|------|
| 1,2,4,3       | 5         | $D[5] = \min\{\text{prev value}, C(1,3)+C(3,5)\} = \min(90, 50+10) = 60$ | 5 | 60   |

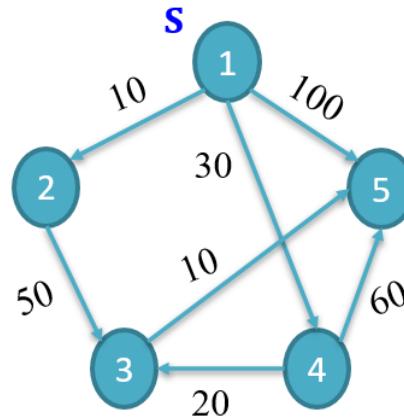
Add u (which is 5) to the source and continue ...

Step 05 :

| S (Source) | V = V - S | Distance D[v] | U | D[u] |
|------------|-----------|---------------|---|------|
| 1,2,4,3,5  | $\phi$    | -             | - | -    |

Since V is null, stop the process.

# Dijkstra's Algorithm Example 2



Cost Adjacency Matrix

|   | 1        | 2        | 3        | 4        | 5        |
|---|----------|----------|----------|----------|----------|
| 1 | 0        | 10       | $\infty$ | 30       | 100      |
| 2 | $\infty$ | 0        | 50       | $\infty$ | $\infty$ |
| 3 | $\infty$ | $\infty$ | 0        | $\infty$ | 10       |
| 4 | $\infty$ | $\infty$ | 20       | 0        | 60       |
| 5 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        |

The Optimal Solution or Shortest Path from Source Vertex (1) is :

| Path              | Shortest Path                                 | Shortest Cost/Distance |
|-------------------|---|------------------------|
| $1 \rightarrow 1$ | -   | 0                      |
| $1 \rightarrow 2$ | $1 \rightarrow 2$                             | 10                     |
| $1 \rightarrow 3$ | $1 \rightarrow 4 \rightarrow 3$               | 50                     |
| $1 \rightarrow 4$ | $1 \rightarrow 4$                             | 30                     |
| $1 \rightarrow 5$ | $1 \rightarrow 4 \rightarrow 3 \rightarrow 5$ | 60                     |

# Dijkstra's Algorithm

**Algorithm DijkstraShortestPaths( $G, v$ ):**

**Input:** A simple undirected weighted graph  $G$  with nonnegative edge weights, and a distinguished vertex  $v$  of  $G$

**Output:** A label,  $D[u]$ , for each vertex  $u$  of  $G$ , such that  $D[u]$  is the distance from  $v$  to  $u$  in  $G$

```

 $D[v] \leftarrow 0$ 
for each vertex  $u \neq v$  of  $G$  do
     $D[u] \leftarrow +\infty$ 

```

Let a priority queue,  $Q$ , contain all the vertices of  $G$  using the  $D$  labels as keys.

```

while  $Q$  is not empty do
    // pull a new vertex  $u$  into the cloud
     $u \leftarrow Q.\text{removeMin}()$ 
    for each vertex  $z$  adjacent to  $u$  such that  $z$  is in  $Q$  do
        // perform the relaxation procedure on edge  $(u, z)$ 
        if  $D[u] + w((u, z)) < D[z]$  then
             $D[z] \leftarrow D[u] + w((u, z))$ 
            Change the key for vertex  $z$  in  $Q$  to  $D[z]$ 

```

**return** the label  $D[u]$  of each vertex  $u$

$O(n \log n)$

$O(m \log n)$

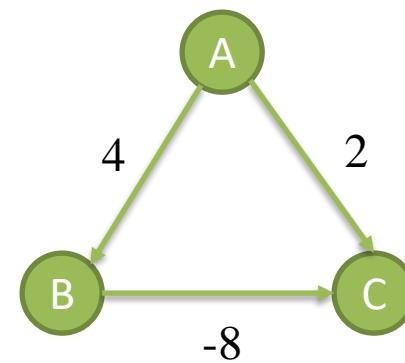
$O((m+n) \log n) \rightarrow O(m \log n)$

# Dijkstra's Algorithm Fails !

- Note that the Dijkstra's Algorithm will fail if the weights of the graph are negative!
- Recall that in Dijkstra's algorithm, *once a vertex is marked as "closed" (and out of the open set) - the algorithm found the shortest path to it*, and will never have to develop this node again - it assumes the path developed to this path is the shortest. But with negative weights - it might not be true.

*Dijkstra from A will first develop C (and add C to the source), and will later fail to find  $A \rightarrow B \rightarrow C$*

*Hence, Dijkstra's algorithm will only work when the weights are positive. If they are negative then Bellman Ford Algorithm (Dynamic Programming technique) helps us!*

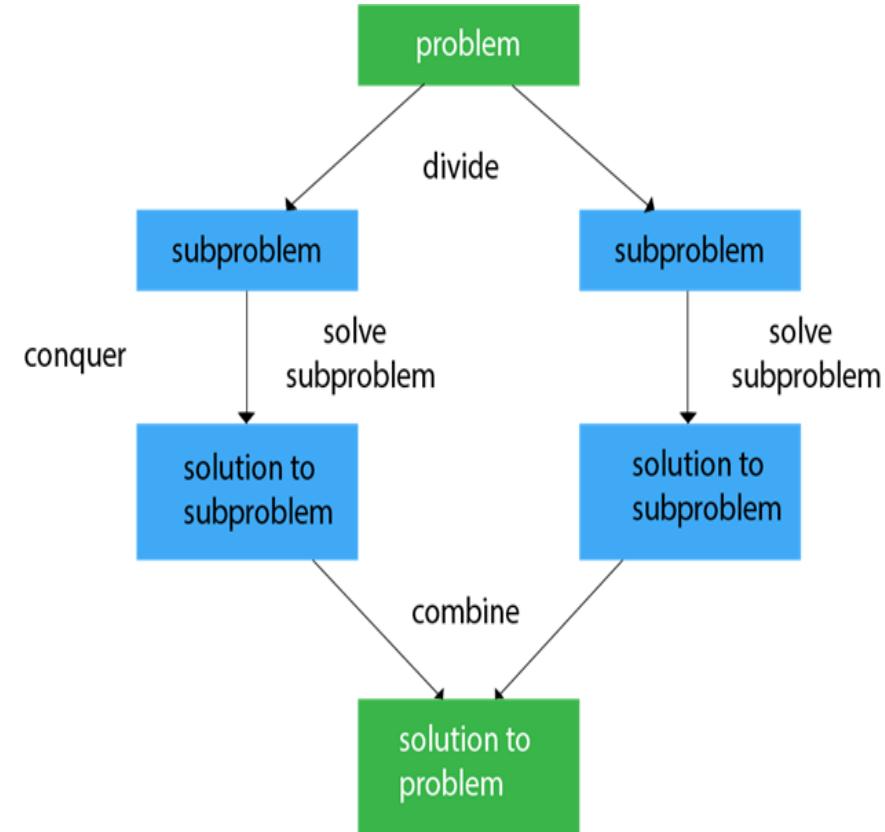


# Divide & Conquer Strategy

- Divide and Conquer (D&C) is an important algorithm design strategy based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem.
- The divide and conquer approach yields some of the most important and efficient algorithms in Computer Science / Data Science ☺
- This technique is the basis of efficient algorithms for all kinds of problems, such as sorting, multiplying large numbers, syntactic analysis, and in scientific research.

# Divide & Conquer Strategy

- Divide and Conquer Algorithms work in the following manner :
- 1) A problem's instance is divided into several smaller instances of the same problem, ideally of the same size (**Divide**).
  - 2) The smaller instances are solved, typically recursively. (**Conquer**).
  - 3) The solutions obtained for the smaller instances are combined to get a solution to the original instance of the problem (**Combine**).



# Advantages of Divide & Conquer Strategy

- **Solving Difficult Problems:** D&C is a powerful tool for solving conceptually difficult problems, such as the classic towers of Hanoi problem. All it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and the combining sub-problems to the original form.
- **Algorithm Efficiency:** D&C technique often helps in the discovery of efficient algorithms. It was the key to Karatsuba's multiplication, quick sort, merge sort algorithms and others. Often D&C led to an improvement in the asymptotic cost of the solution.
- **Parallelism:** D&C algorithms are naturally adapted for execution in multi-processor machines.
- **Efficient Memory access**

# Some common Divide & Conquer Algorithms

---

- Binary Search
- Merge Sort
- Quick Sort
- Integer Multiplication – Karatsuba's Algorithm
- Finding maximum and minimum from a list [Exercise Question]
- Strassen's Matrix Multiplication [Self – Study]
- ...
- ...

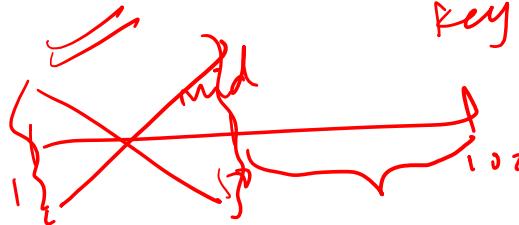
# Search Techniques

---

- We already know linear search in which we start from the leftmost element and compare it with the search term (Key);
- If the search term matches the number on the index we are currently on, then the search operation is successful and the index is returned!
- But, if the numbers don't match, then we go to the number on the next index and follow the same procedure till the number is found or we have reached the end of the list.
- If the number isn't present, we return that the search was unsuccessful. The time complexity of linear sort is  $O(n)$ . This may hence take enormous time when there are many inputs.
- Which Algorithmic Strategy is linear Search ? *Brute Force !!*

# Binary Search

Key = 75



- Binary search is one such divide and conquer algorithm to assist with the given problem; note that a sorted array should be used in this case.
- This search algorithm recursively divides the array into two sub-arrays that may contain the search term.
- It discards one of the sub-array by utilizing the fact that items are sorted.
- It continues halving the sub-arrays until it finds the search term or it narrows down the list to a single item.
- Since binary search discards the sub-array it's pseudo Divide & Conquer algorithm. What makes binary search efficient is the fact that if it doesn't find the search term in each iteration, it just reduces the array/list to its half for the next iteration.
- The time complexity of binary search is  $O(\log n)$ , where  $n$  is the number of elements in an array. If the search term is at the center of the array, it's considered to be the best case since the element is found instantly in a go. Hence the best case complexity will be  $O(1)$

# Merge Sort

- Merge-sort on an input sequence  $S$  with  $n$  elements consists of three steps:
  - **Divide**: partition  $S$  into two sequences  $S_1$  and  $S_2$  of about  $n/2$  elements each
  - **Recur**: recursively sort  $S_1$  and  $S_2$
  - **Conquer**: merge  $S_1$  and  $S_2$  into a unique sorted sequence

**Algorithm**  $\text{mergeSort}(S, C)$

**Input** sequence  $S$  with  $n$  elements, comparator  $C$

**Output** sequence  $S$  sorted according to  $C$

**if**  $S.\text{size}() > 1$

$(S_1, S_2) \leftarrow \text{partition}(S, n/2)$

$\text{mergeSort}(S_1, C)$

$\text{mergeSort}(S_2, C)$

$S \leftarrow \text{merge}(S_1, S_2)$

# Merge Sort

- The *conquer step of merge-sort consists of merging two sorted sequences A and B* into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with  $n/2$  elements and implemented by means of a doubly linked list, takes  $O(n)$  time

**Algorithm *merge(A, B)***

**Input** sequences  $A$  and  $B$  with  $n/2$  elements each

**Output** sorted sequence of  $A \cup B$

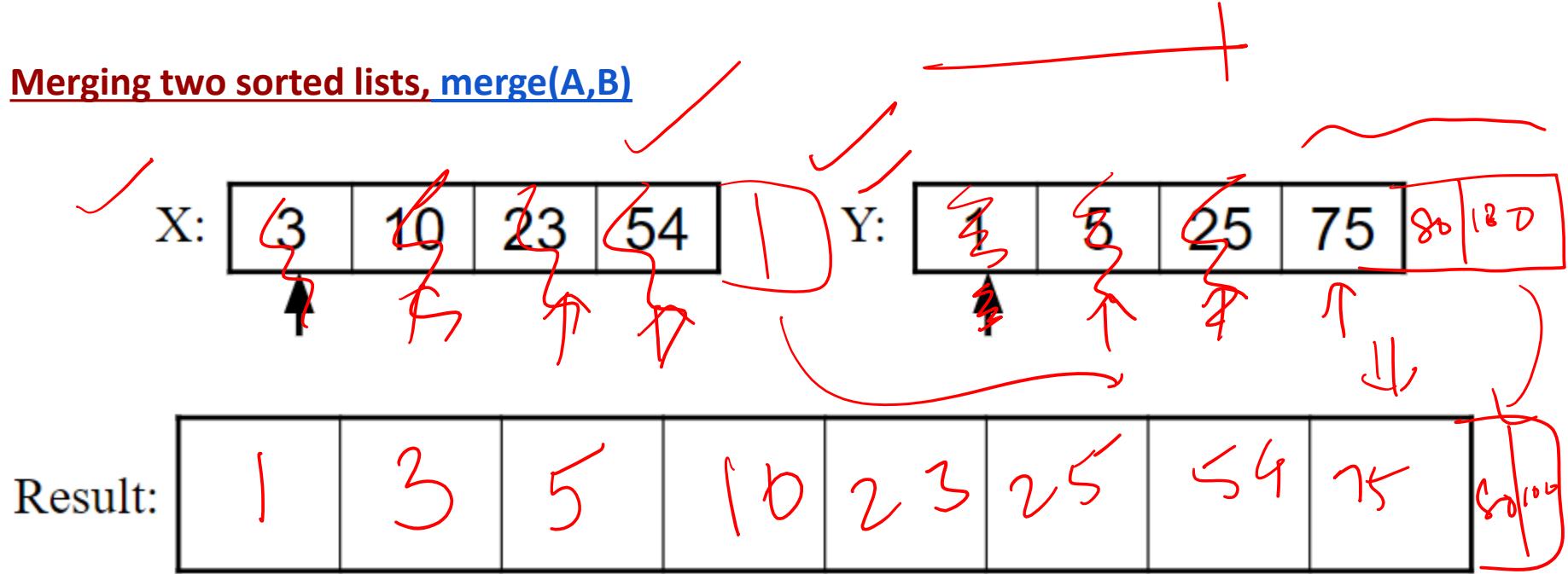
```

res S ← empty sequence
while ¬A.isEmpty() ∧ ¬B.isEmpty()
  if A.first().element() < B.first().element()
    S.insertLast(A.remove(A.first()))
  else
    S.insertLast(B.remove(B.first()))
  while ¬A.isEmpty()
    S.insertLast(A.remove(A.first()))
  while ¬B.isEmpty()
    S.insertLast(B.remove(B.first()))
return S
  
```

*sorted !!*

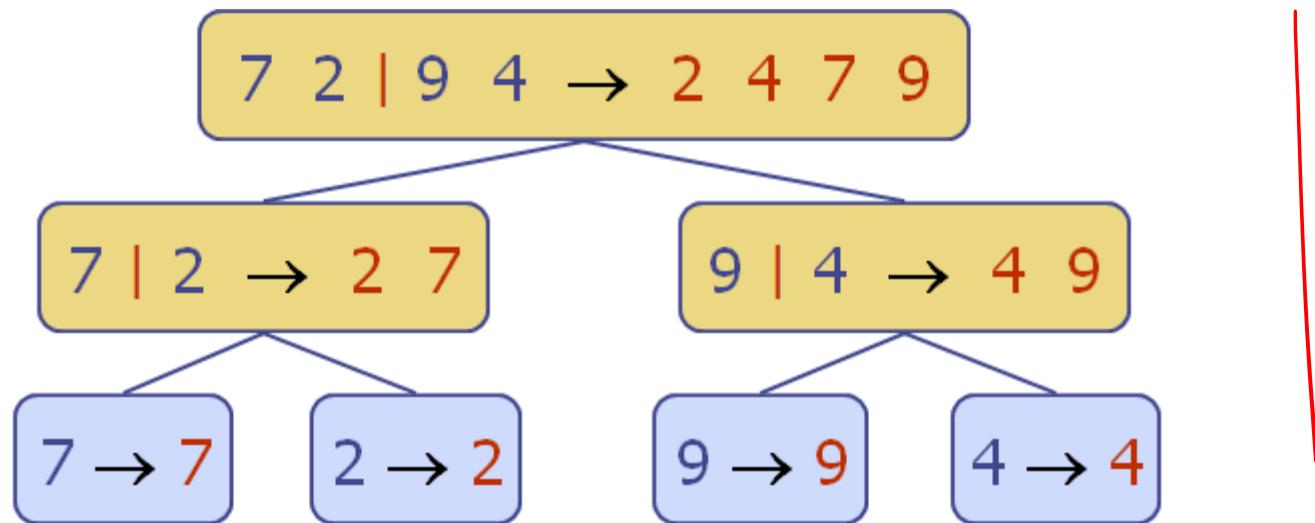
# Merge Sort

Merging two sorted lists, merge(A,B)

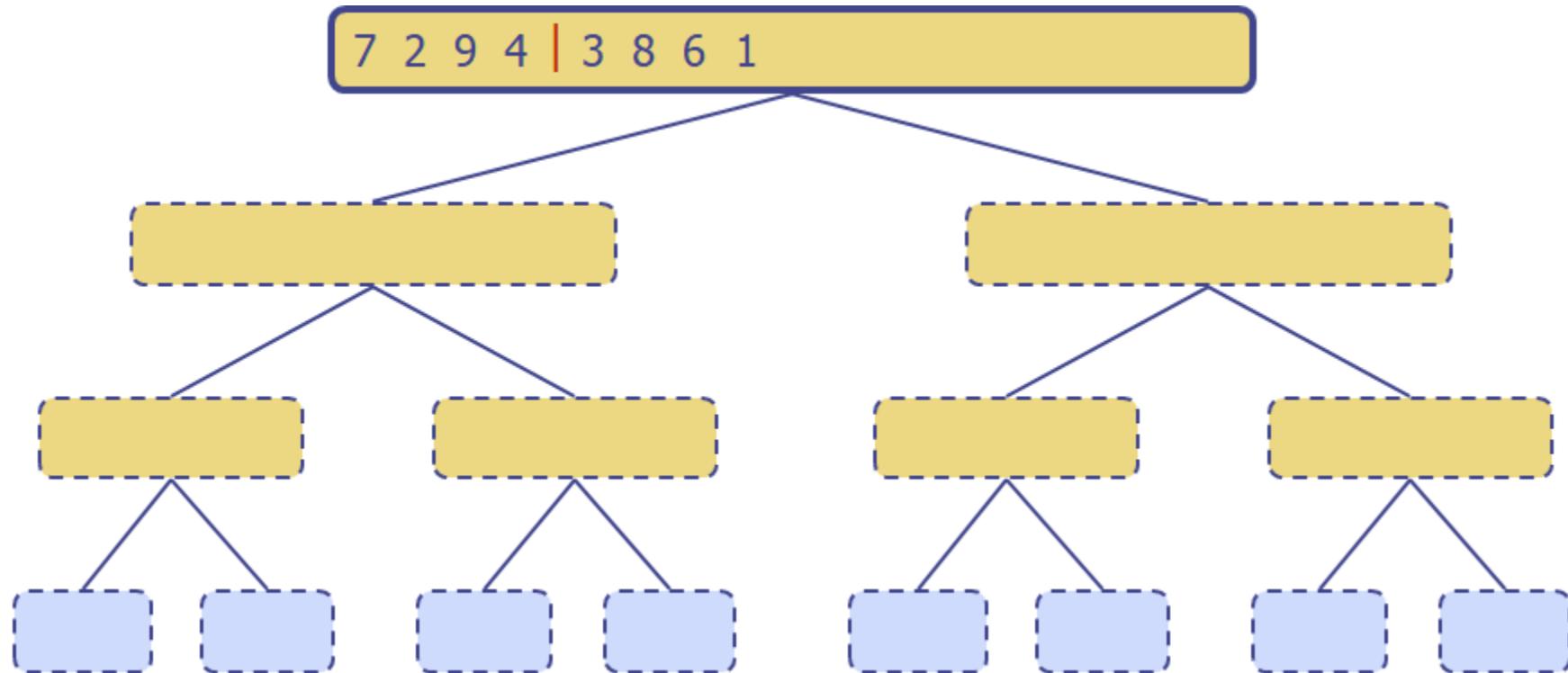


# Merge Sort Example 1

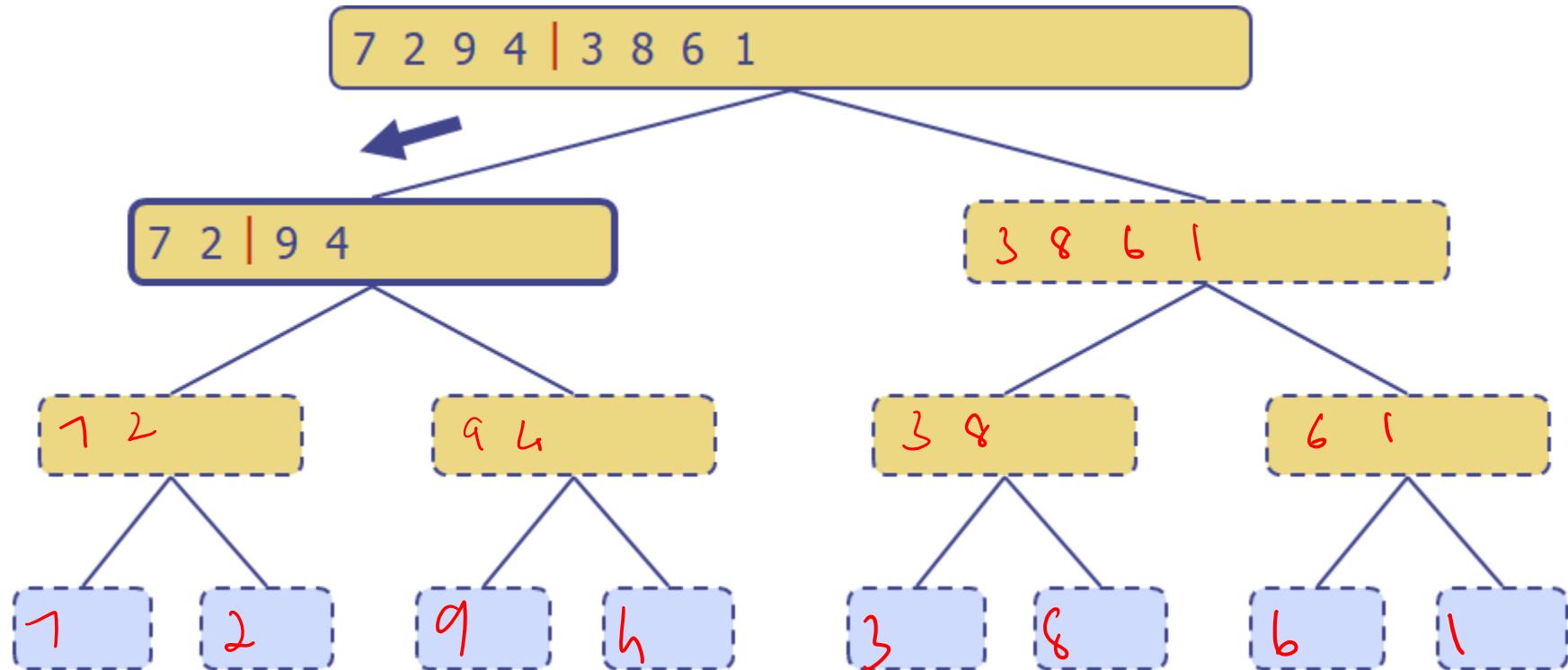
- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



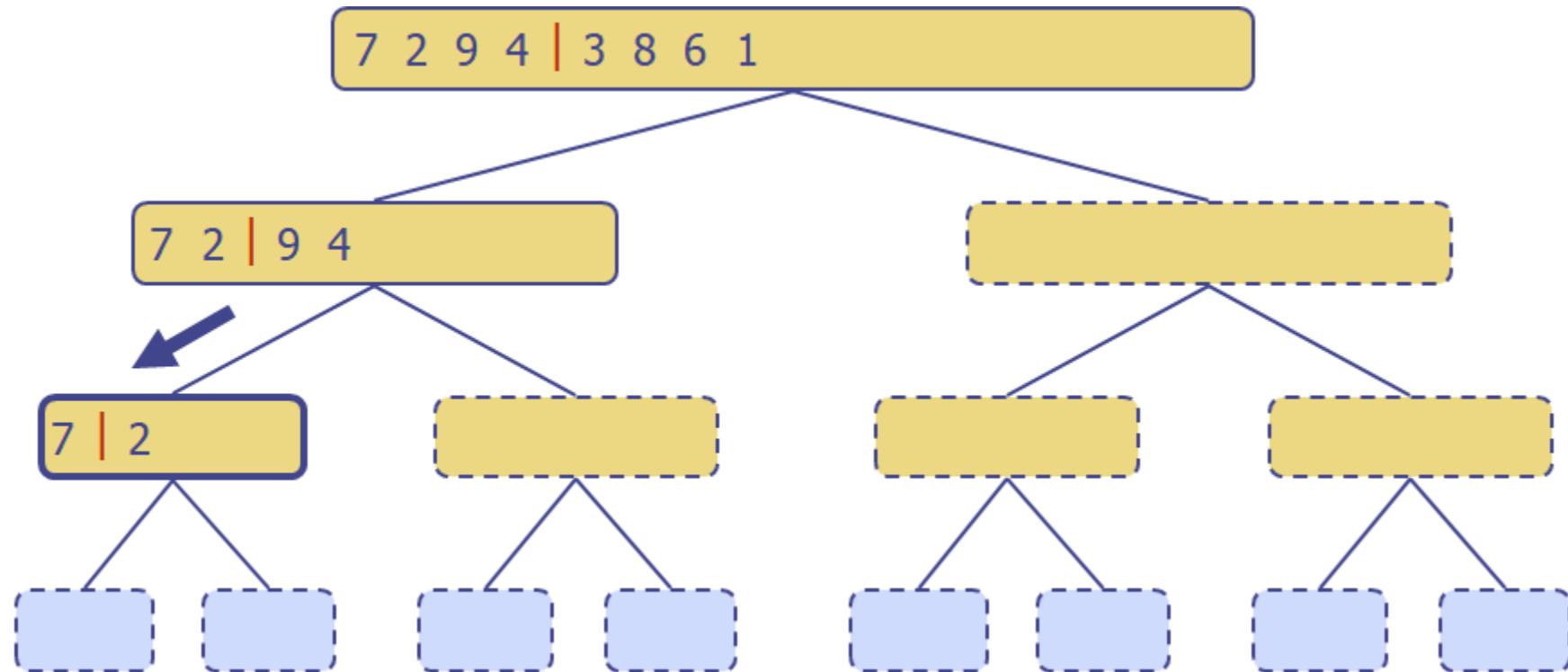
# Merge Sort Example 1



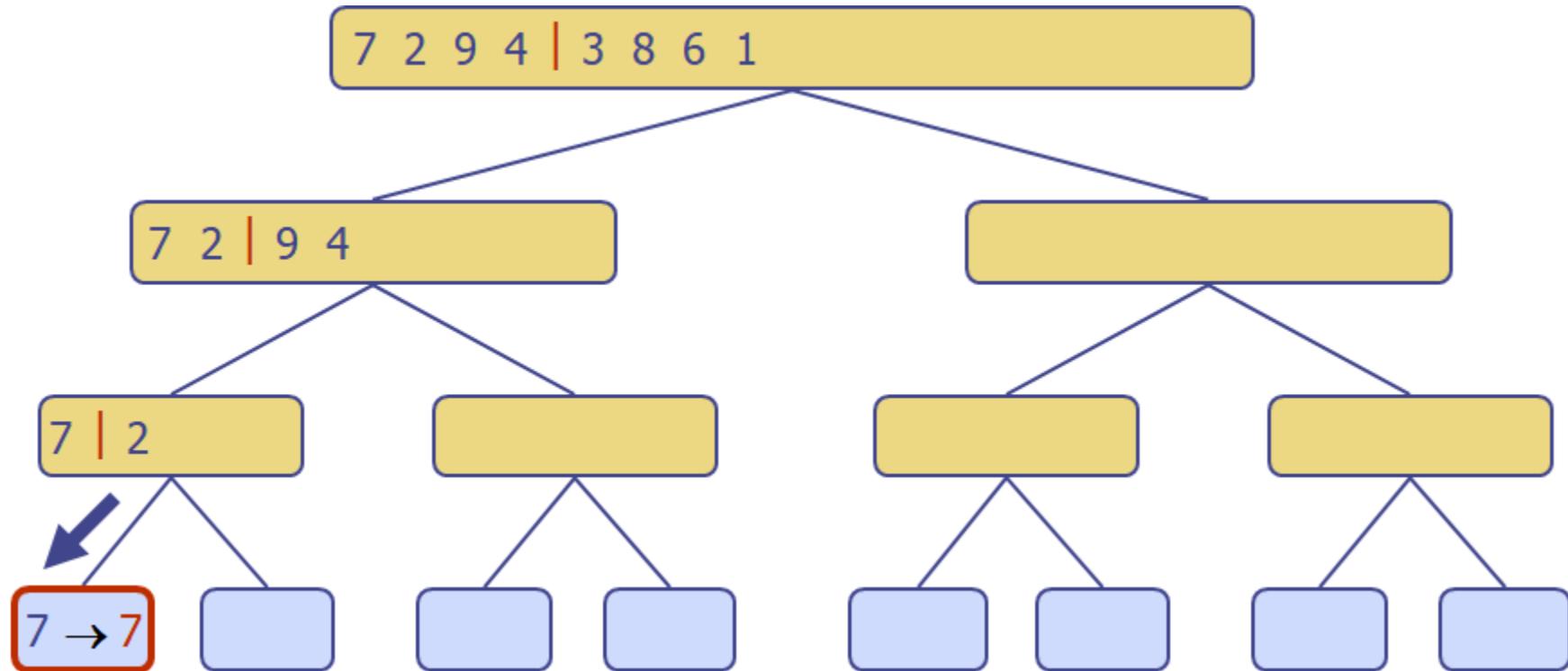
# Merge Sort Example 1



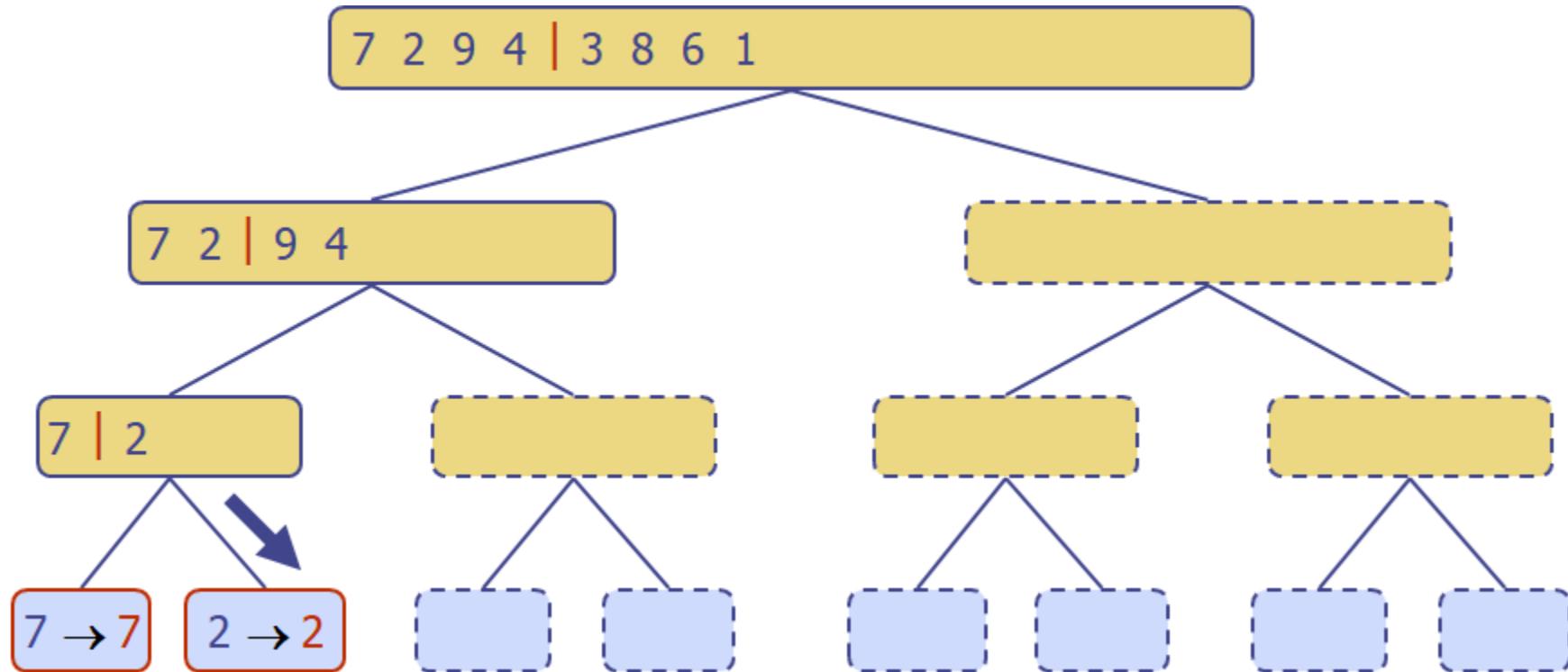
# Merge Sort Example 1



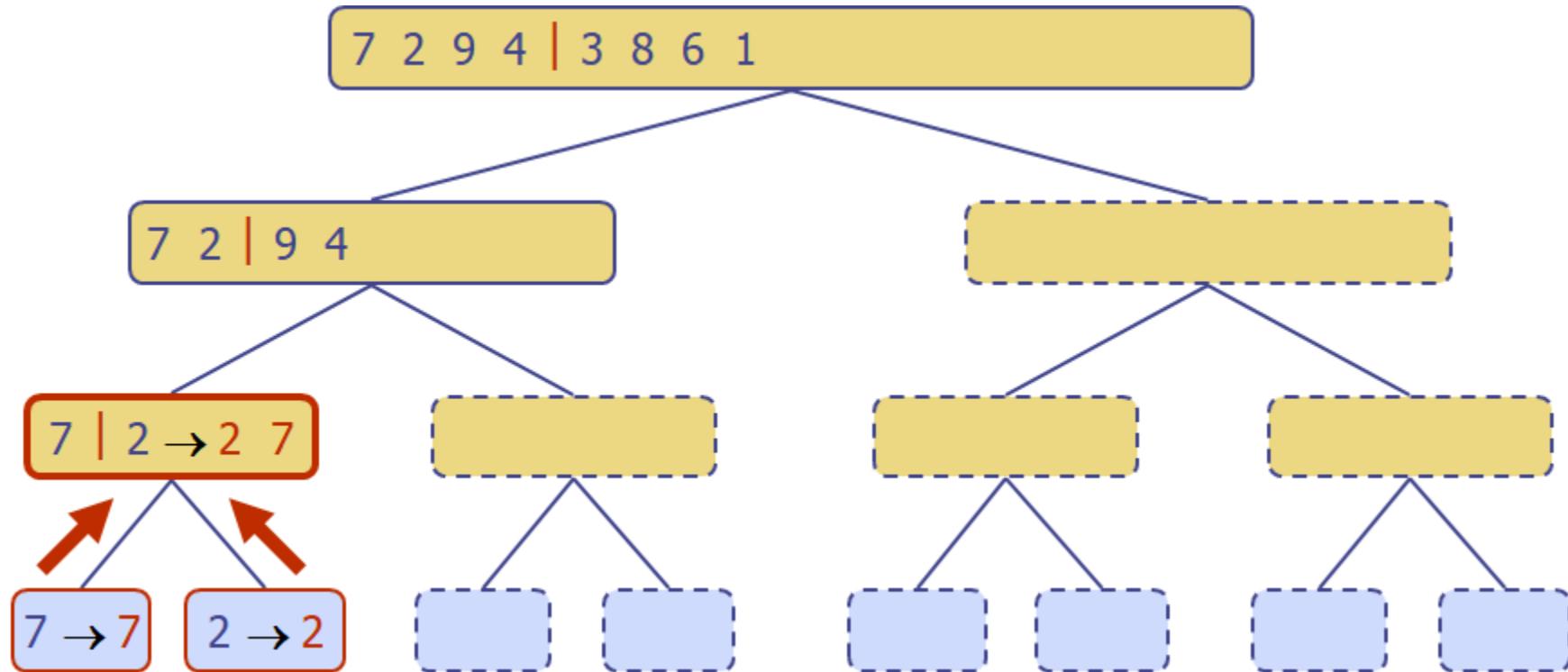
# Merge Sort Example 1



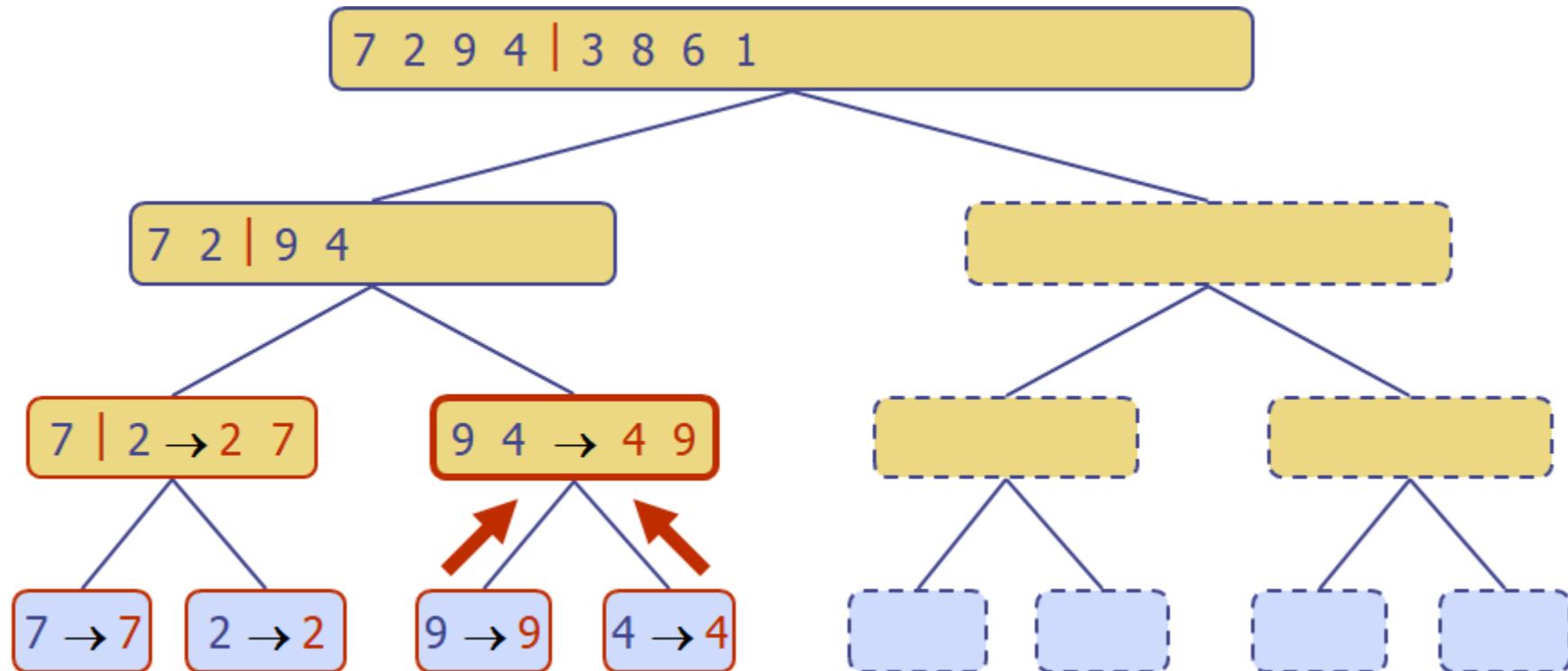
# Merge Sort Example 1



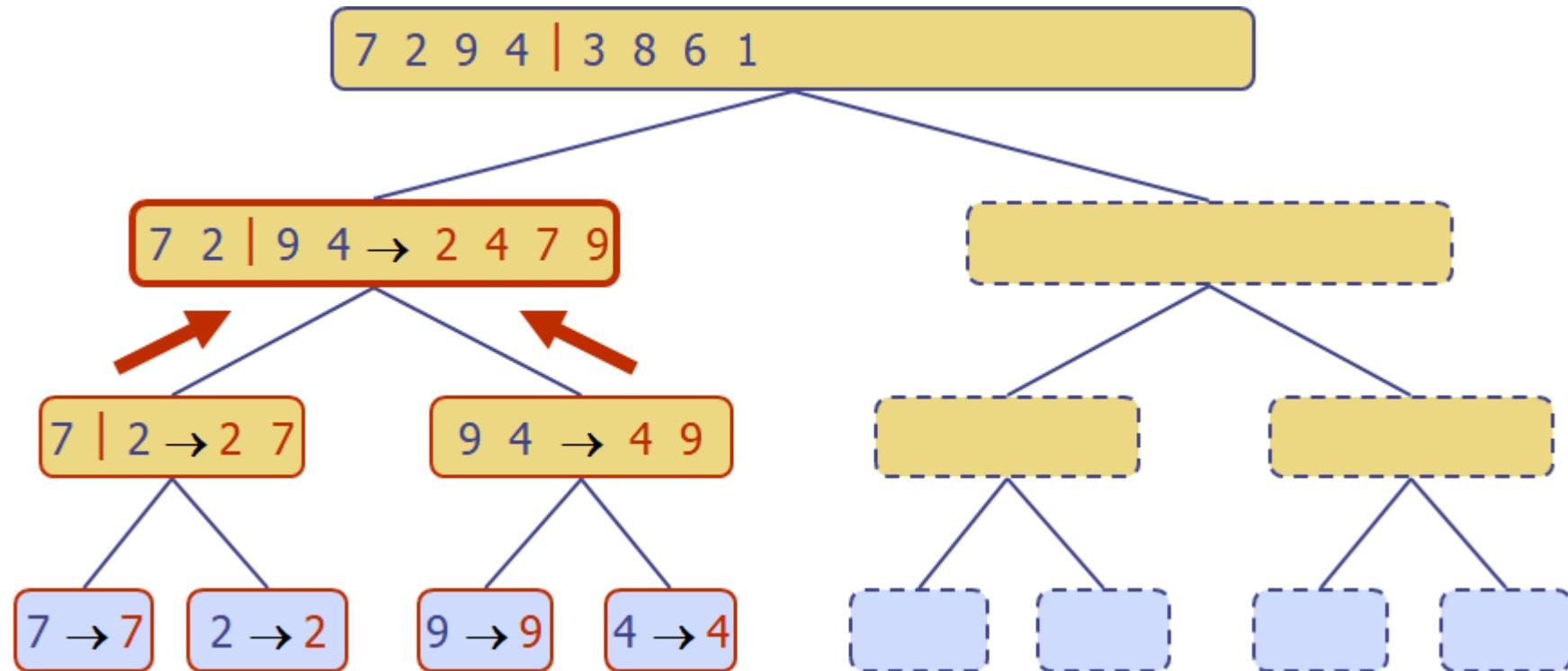
# Merge Sort Example 1



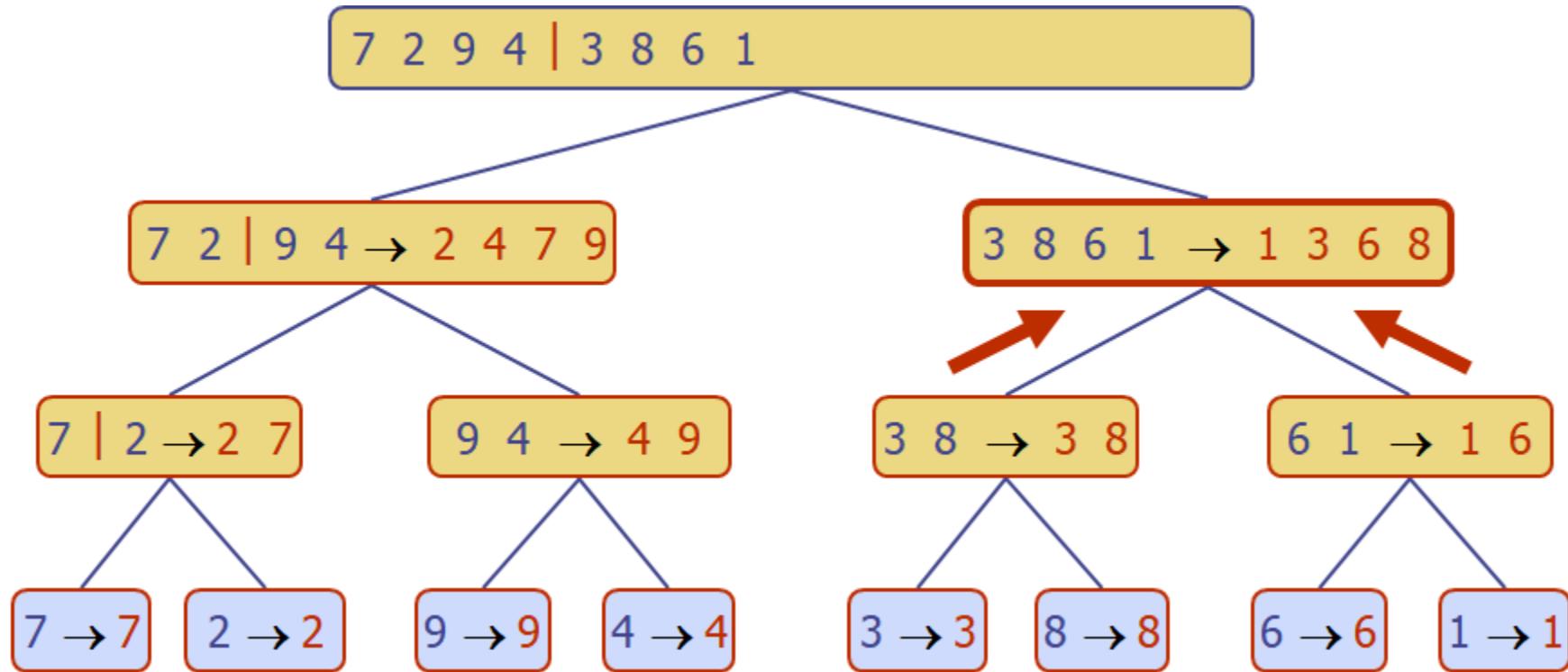
# Merge Sort Example 1



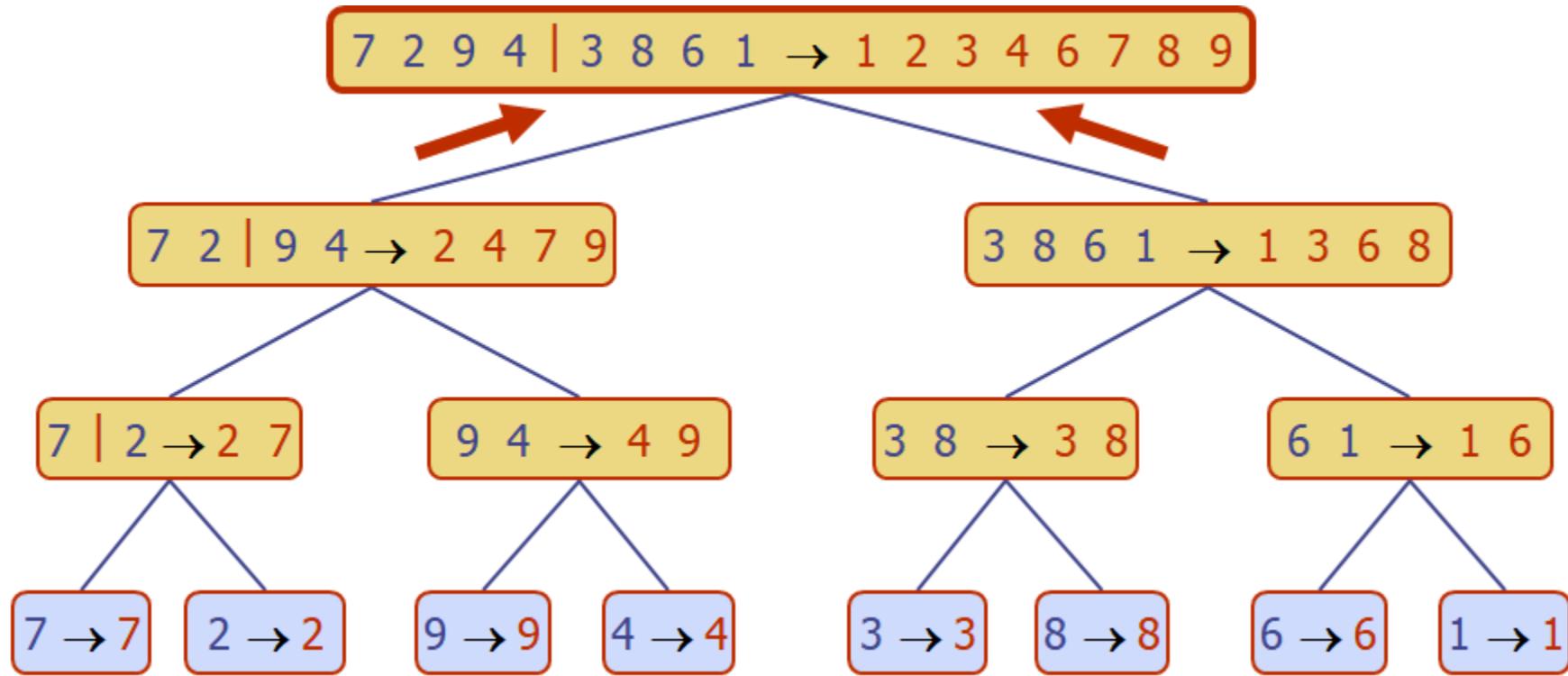
# Merge Sort Example 1



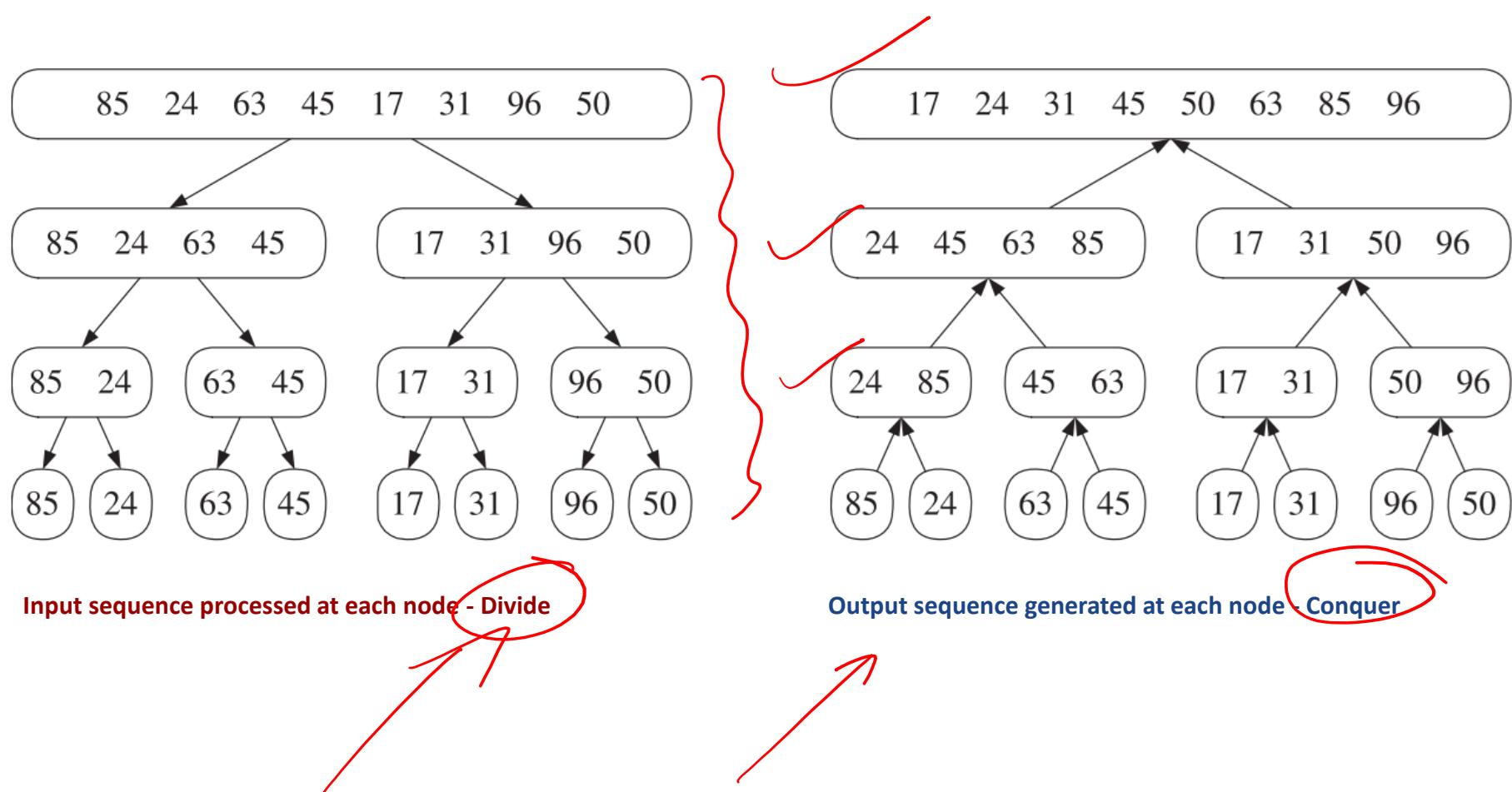
# Merge Sort Example 1



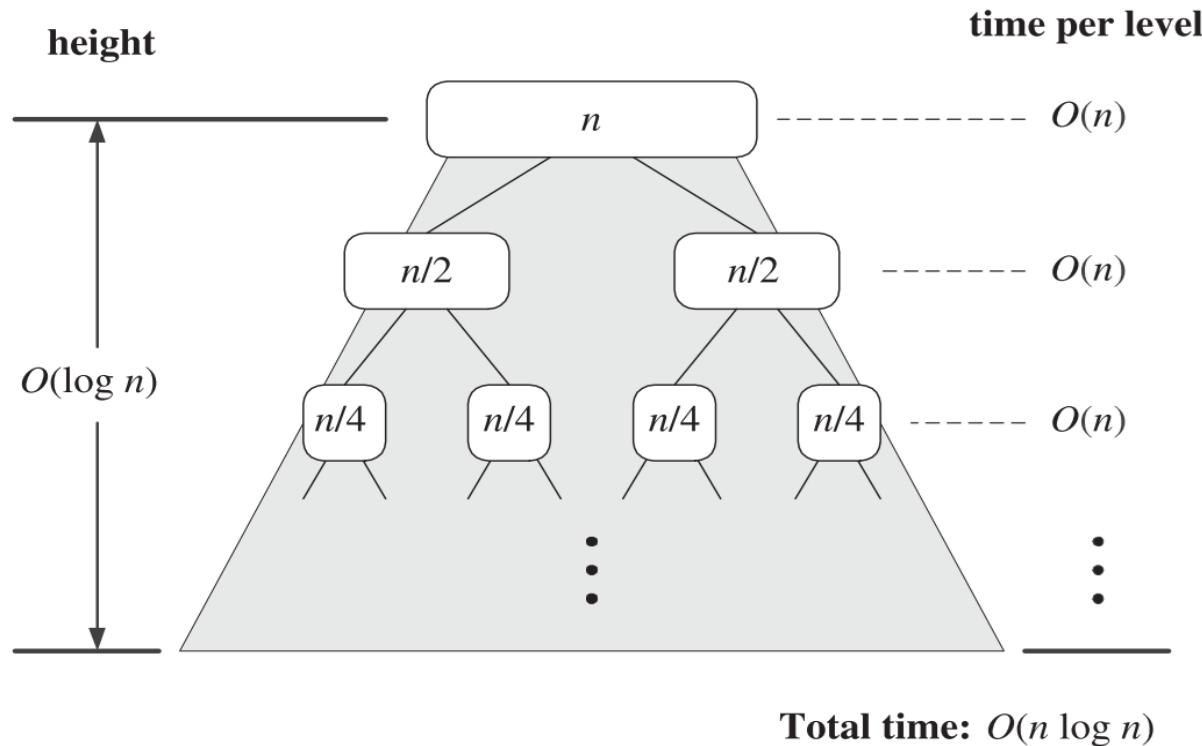
# Merge Sort Example 1



# Merge Sort Example 2



# Merge Sort



$$t(n) = \begin{cases} b & \text{if } n = 1 \text{ or } n = 0 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{otherwise} \end{cases}$$

**Assume,  $n$  is a power of 2 :**  $t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$

# Merge Sort Analysis

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

**By repeated substitution,**

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2t(n/2^2) + 2cn. \end{aligned}$$

**Substituting Again...**

$$t(n) = 2^3t(n/2^3) + 3cn$$

**Substituting Again...**

$$t(n) = 2^4t(n/2^4) + 4cn$$

**Generalizing it ...**

$$t(n) = 2^i t(n/2^i) + icn$$

when  $i = \log n$ ,  $2^i = n$

$$\begin{aligned} t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n)cn \\ &= nt(1) + cn \log n \\ &= nb + cn \log n. \end{aligned}$$

**This implies,**

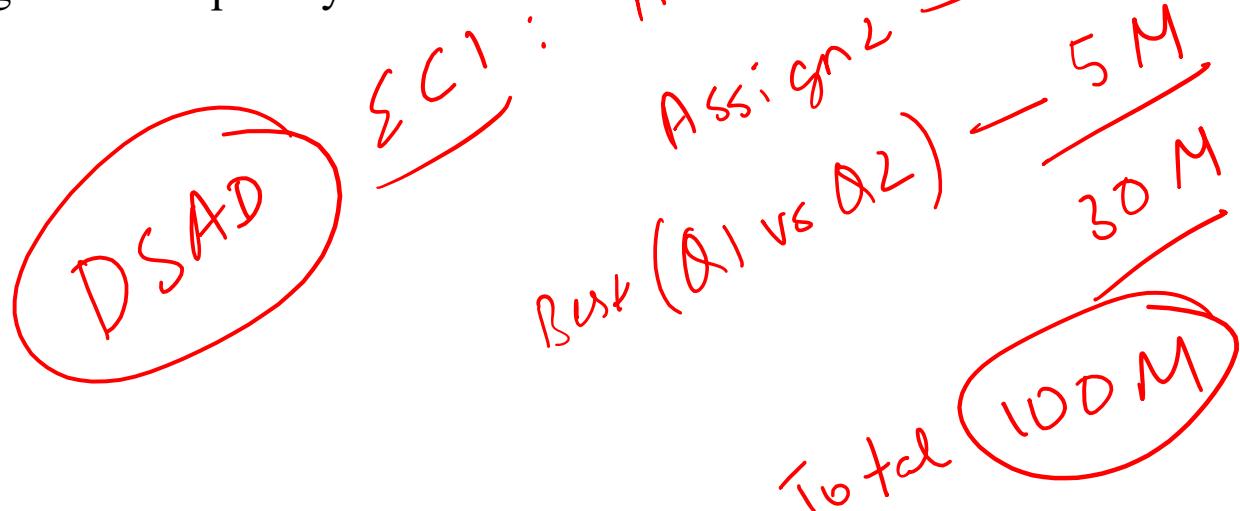
$$t(n) \text{ is } O(n \log n)$$

# Merge Sort Analysis

$$t(n) = \begin{cases} b & \text{if } n = 1 \text{ or } n = 0 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{otherwise} \end{cases}$$

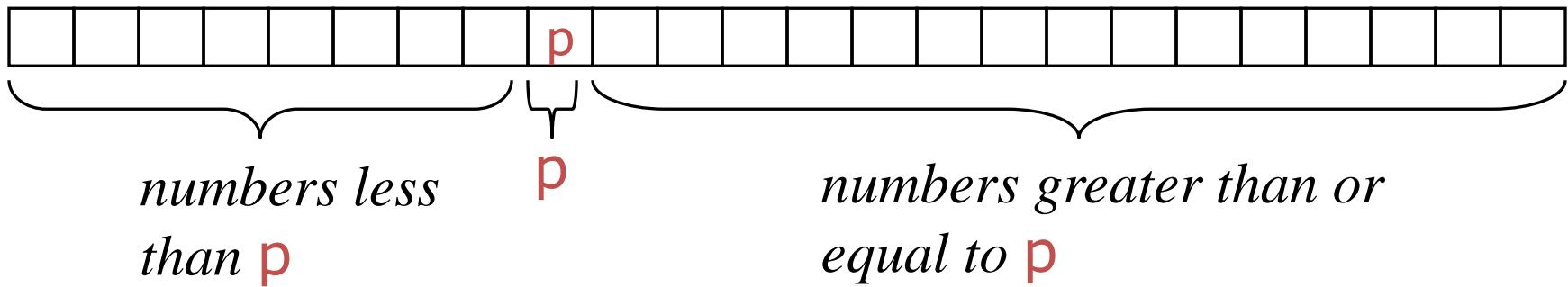
$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

Apply Master's theorem to get the complexity !!



# Quick Sort

- Quick is a very popular sorting algorithm which is based on the divide and conquer strategy. The steps involved here are :
- Pick some number  $p$  from the array
- Move all numbers less than  $p$  to the beginning of the array
- Move all numbers greater than (or equal to)  $p$  to the end of the array
- Quicksort the numbers less than  $p$
- Quicksort the numbers greater than or equal to  $p$



# Quick Sort Analogy – Assembly Line



*Compare and get into **right** position !!*

*Right position means anyone in front / left of you is smaller than you  
and anyone after you / behind you are greater than you!*



# Quick Sort Algorithm

---

- Given an array of  $n$  elements (e.g., integers):
- If array only contains one element, return
- Else
  - pick one element to use as *pivot*.
  - **Partition** elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

# Quick Sort Example

We are given array of n integers to sort:

|    |    |    |    |    |    |   |    |     |
|----|----|----|----|----|----|---|----|-----|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|----|----|----|----|----|----|---|----|-----|

- *Pick Pivot Element:* There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

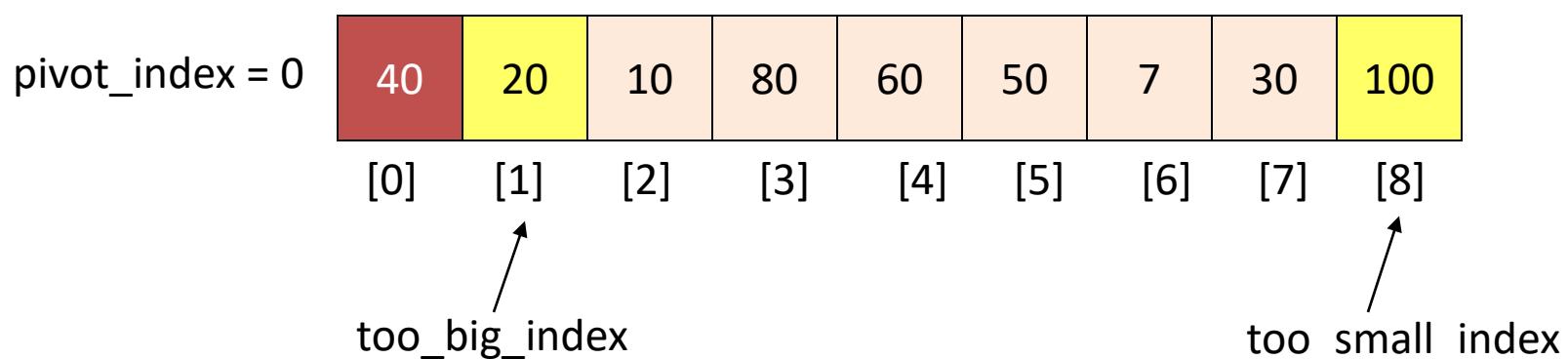
|    |    |    |    |    |    |   |    |     |
|----|----|----|----|----|----|---|----|-----|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|----|----|----|----|----|----|---|----|-----|

# Partitioning Array in Quick Sort



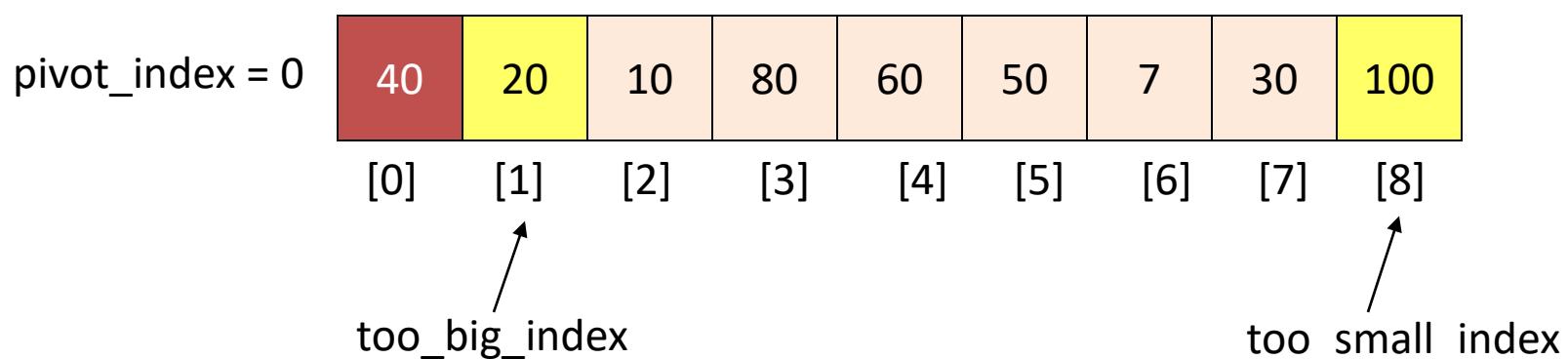
- Given a pivot, partition the elements of the array such that the resulting array consists of:
  - One sub-array that contains elements  $\geq$  pivot
  - Another sub-array that contains elements  $<$  pivot
- The sub-arrays are stored in the original data array.
- Partitioning loops through, swapping elements below/above pivot.

# Quick Sort



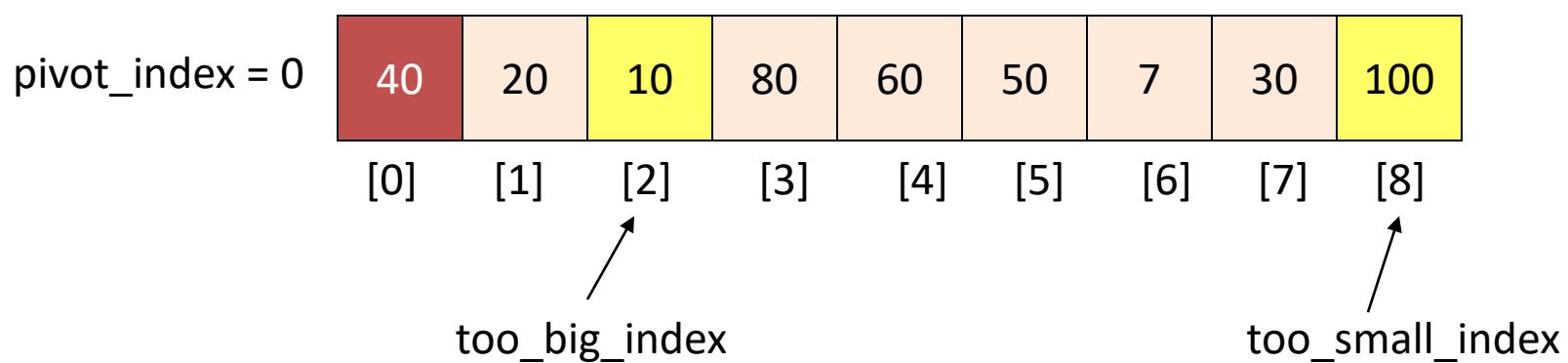
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



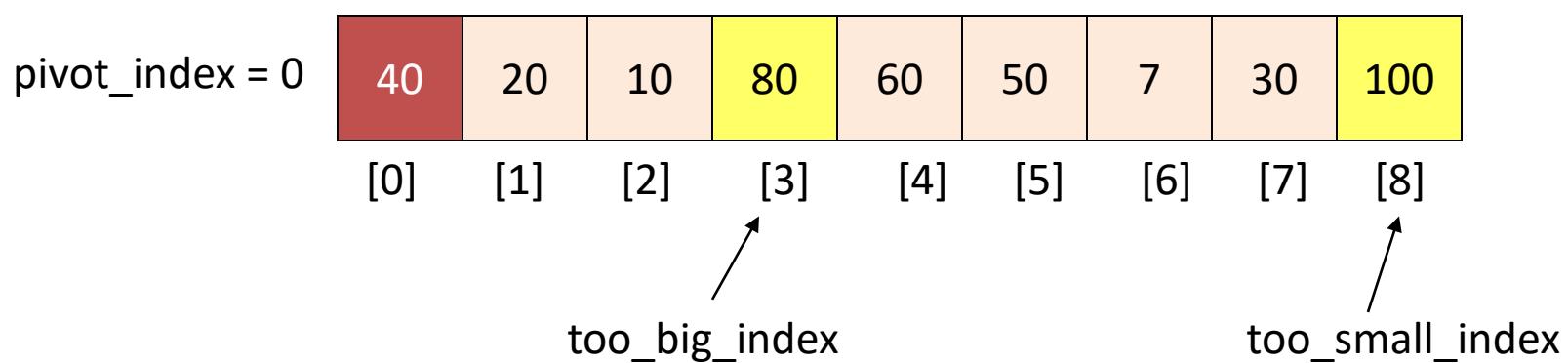
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



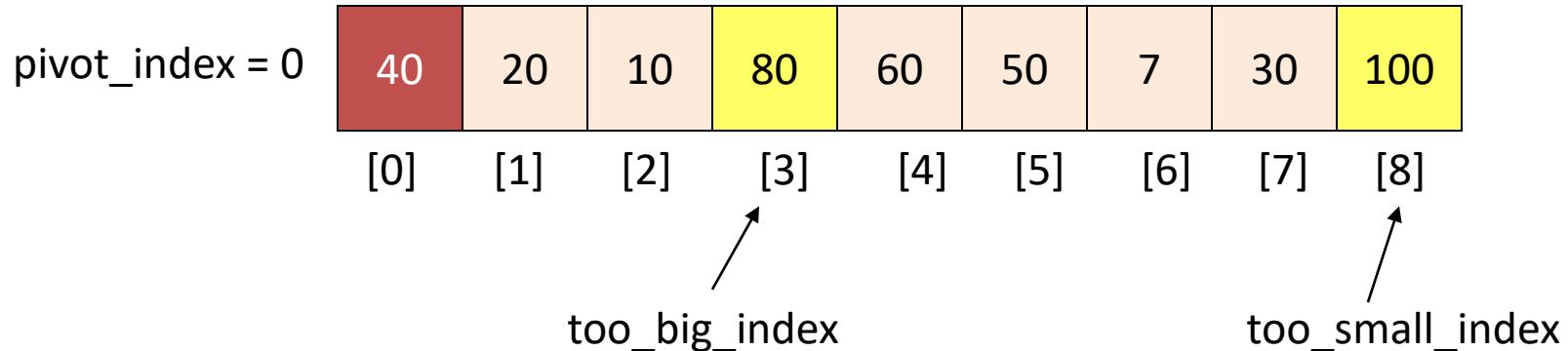
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



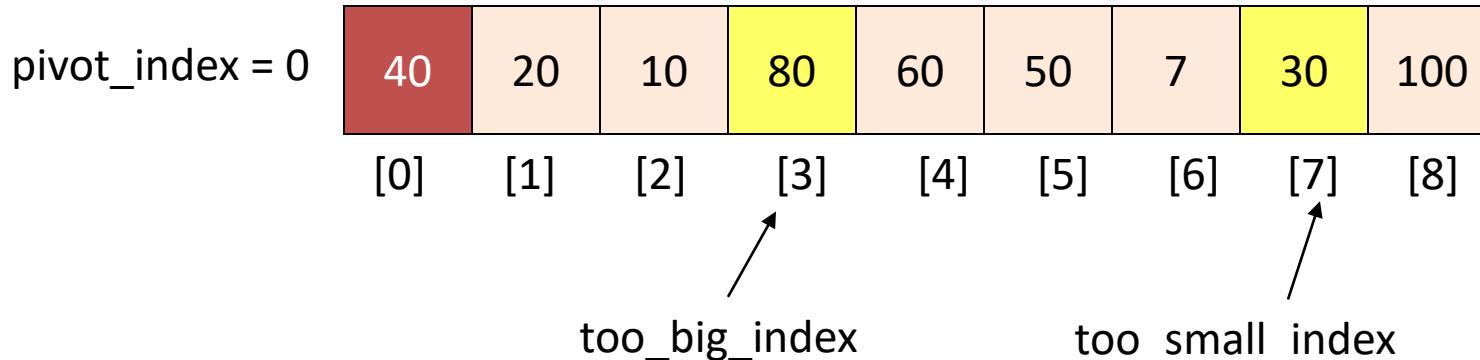
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$



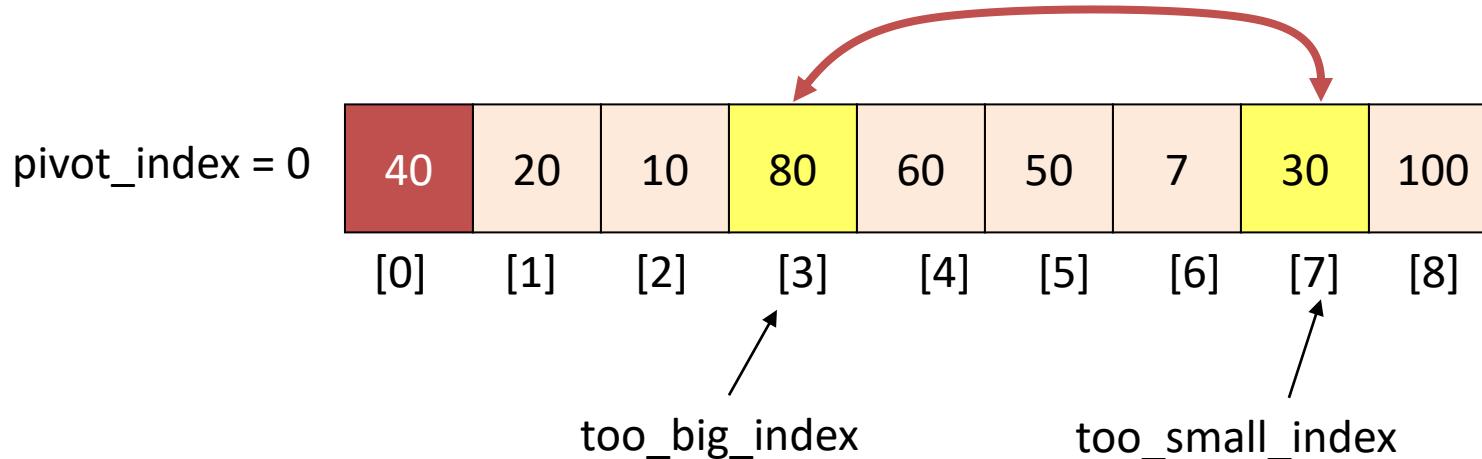
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$



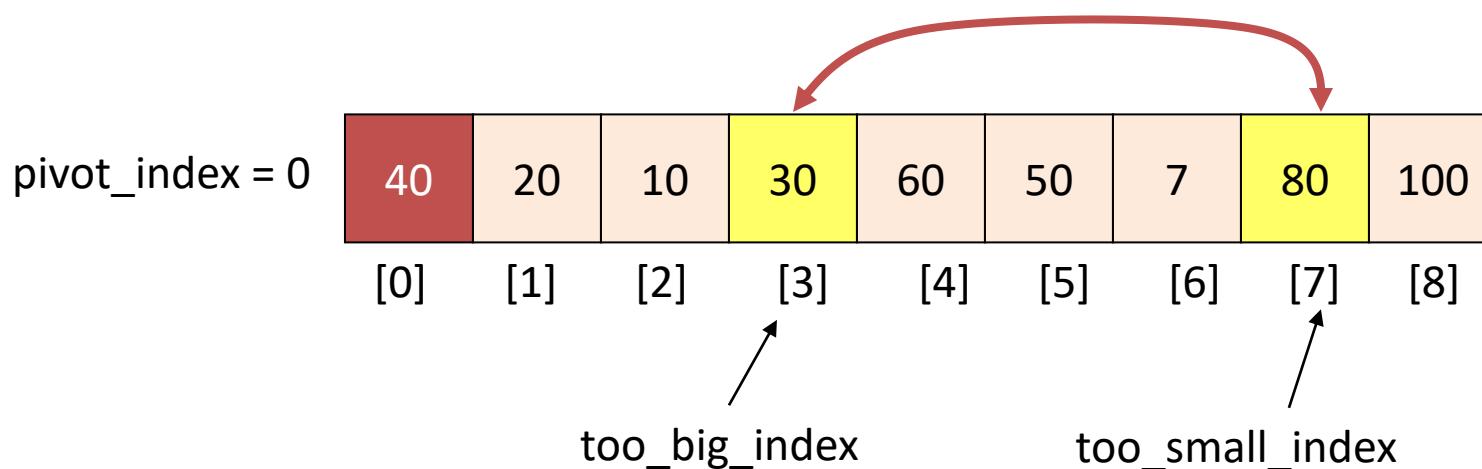
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$



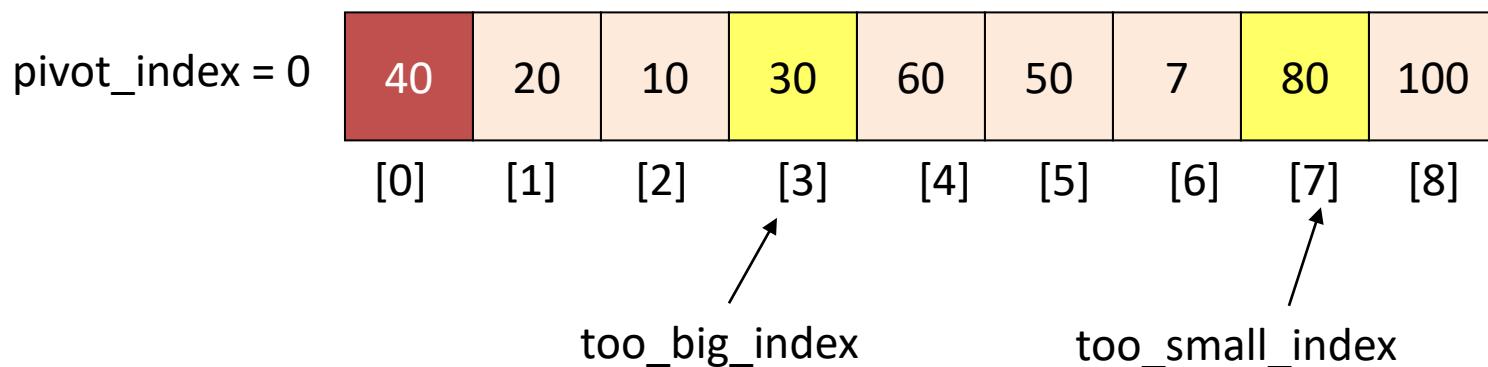
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$



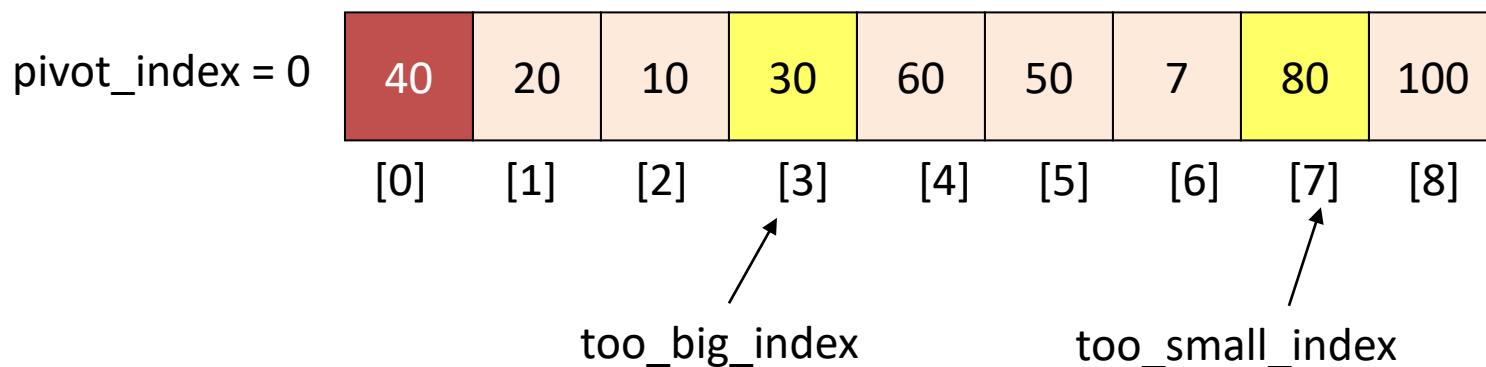
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

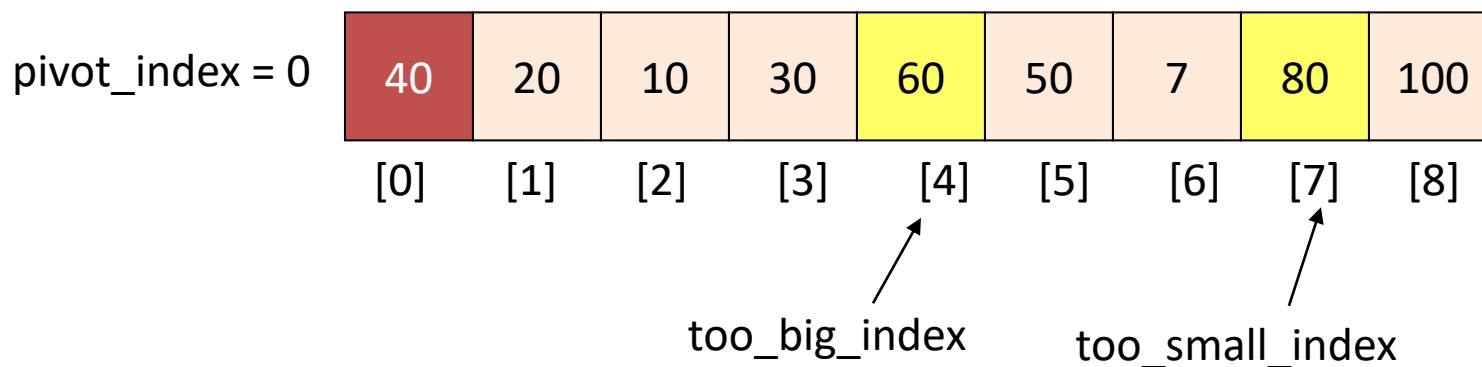


# Quick Sort

- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
 $\quad \quad \quad \text{swap data[too\_big\_index] and data[too\_small\_index]}$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

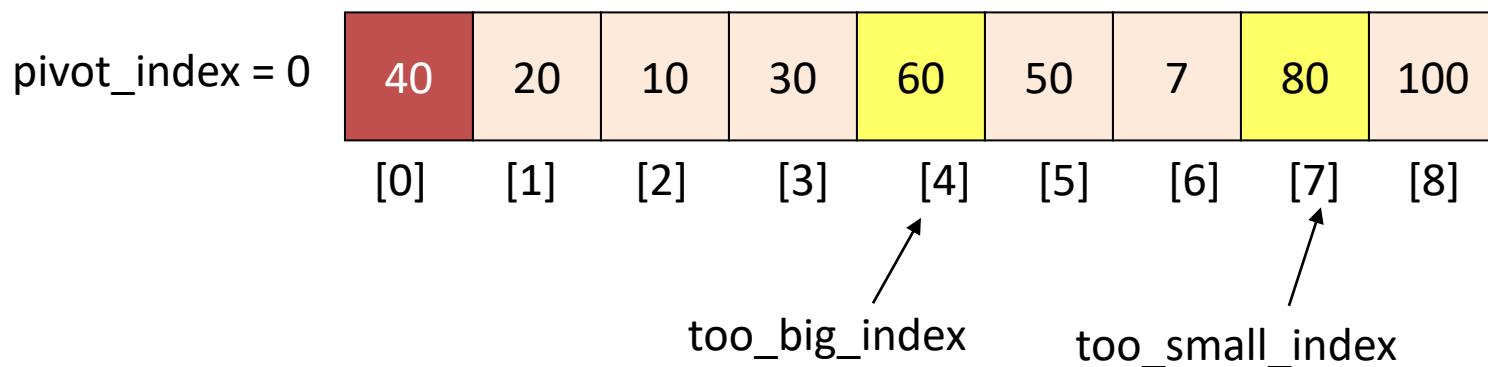


# Quick Sort



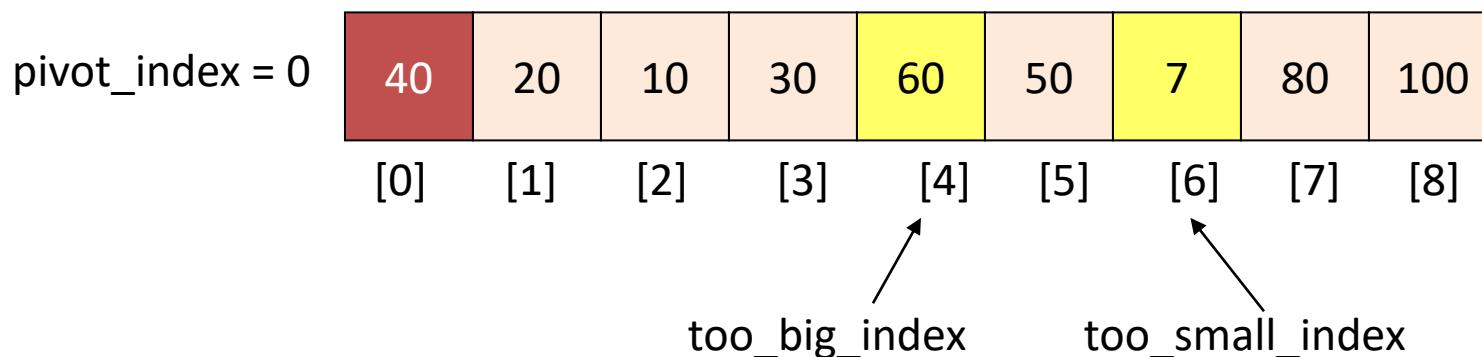
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



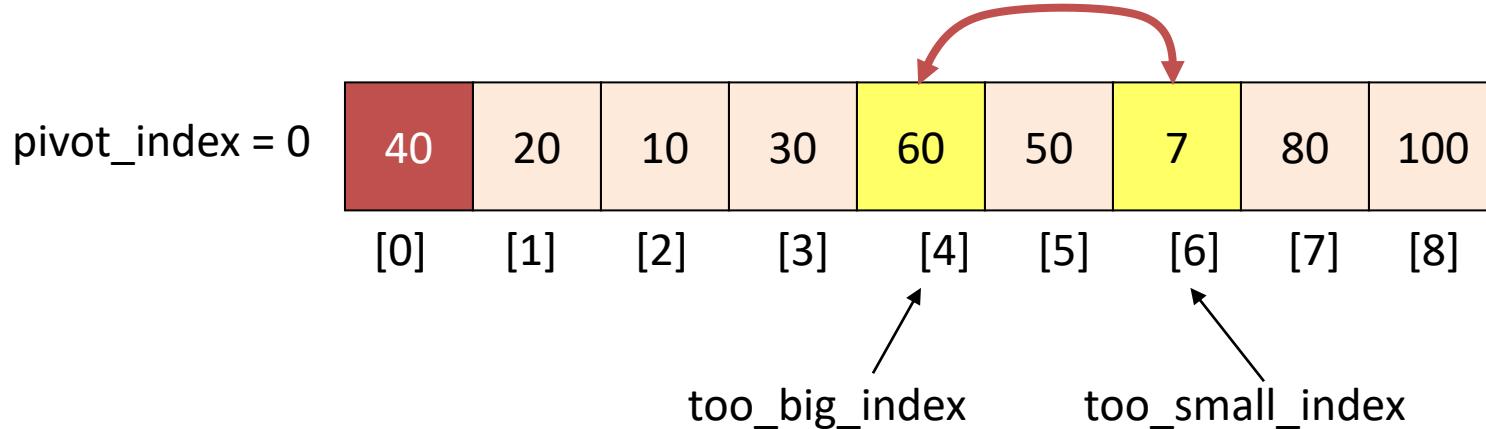
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  - 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



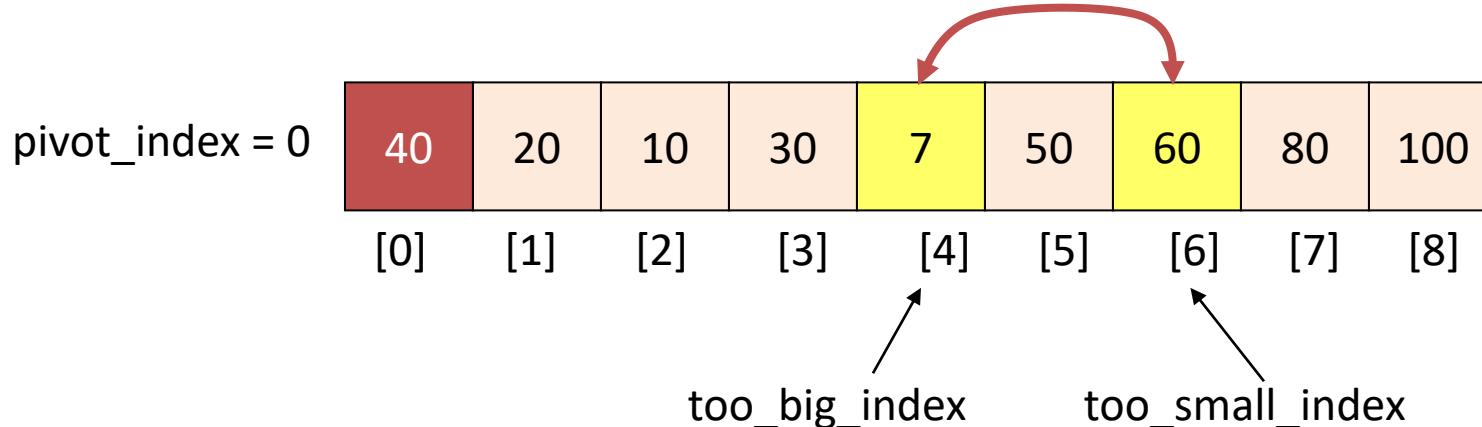
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



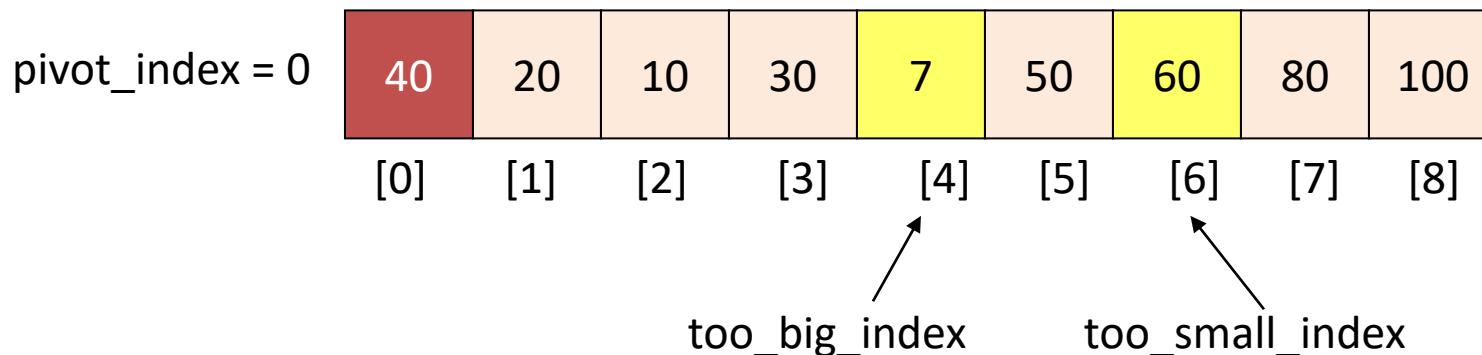
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

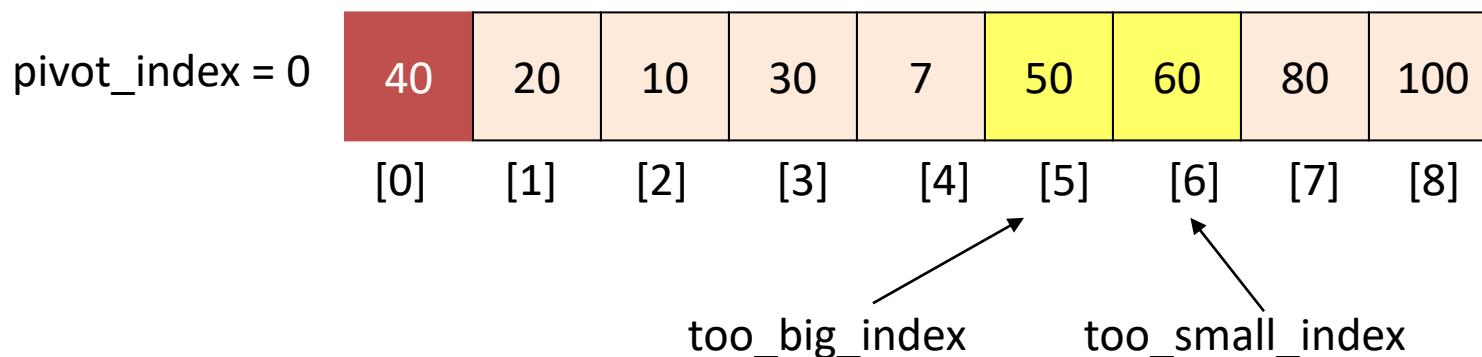


# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
  - 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

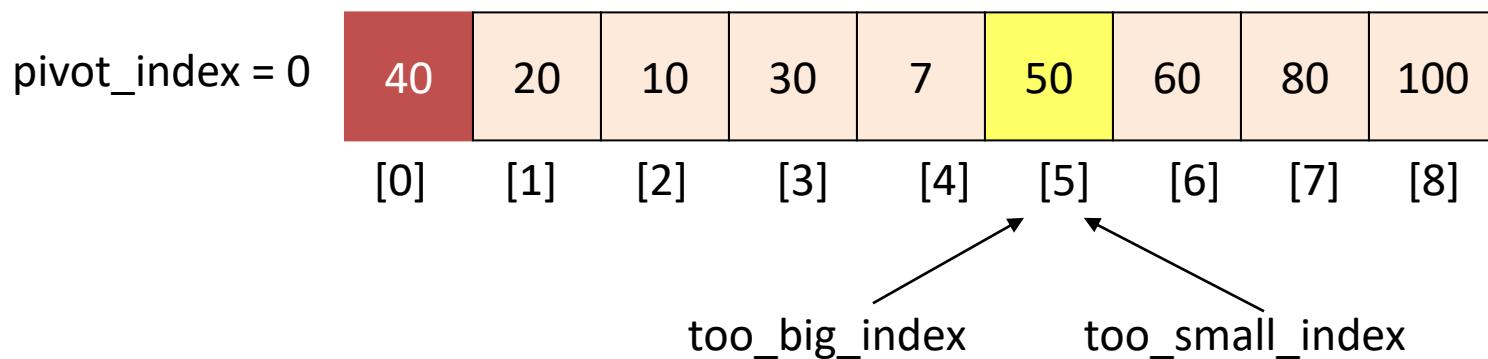


# Quick Sort



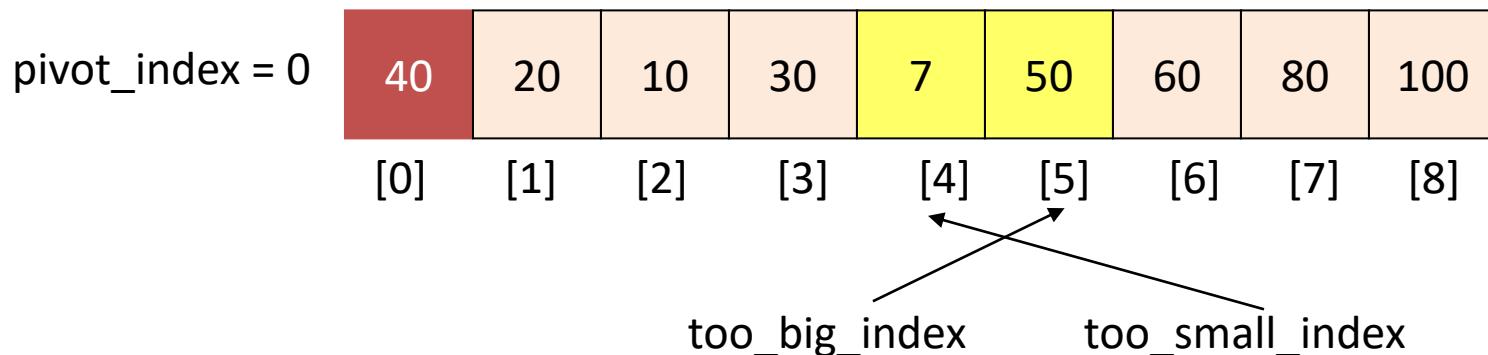
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



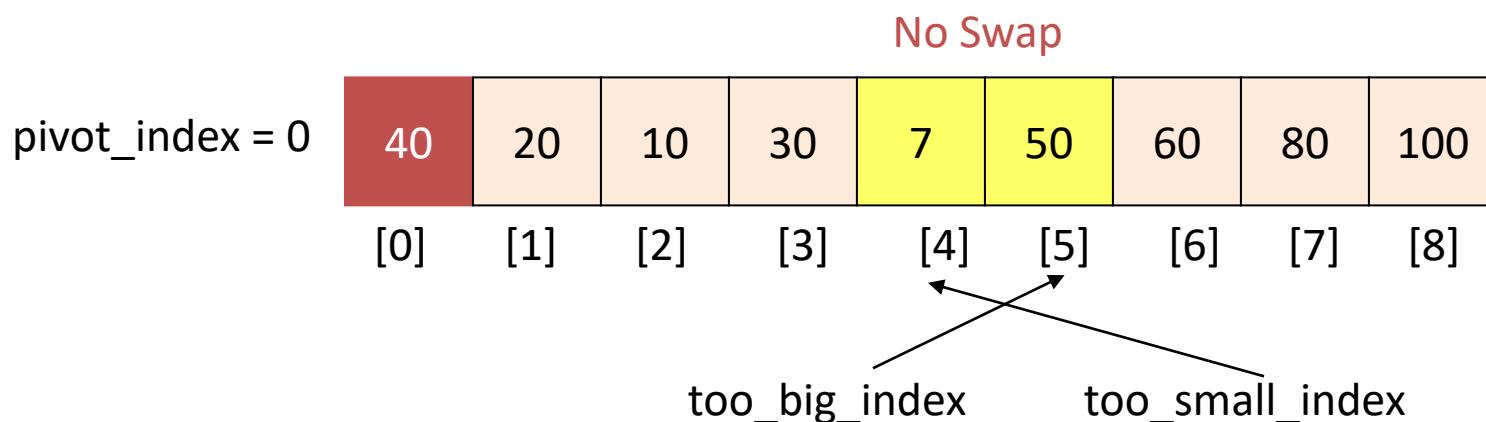
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



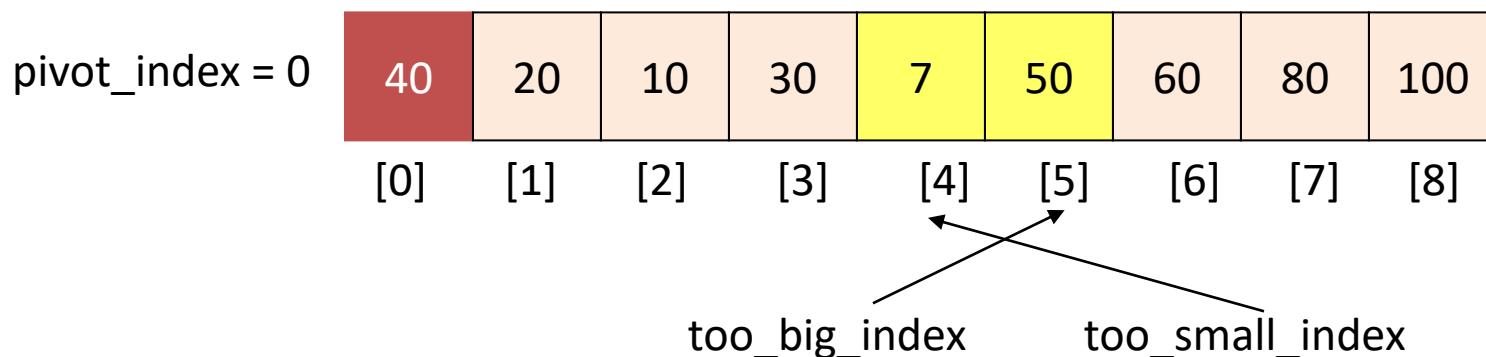
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



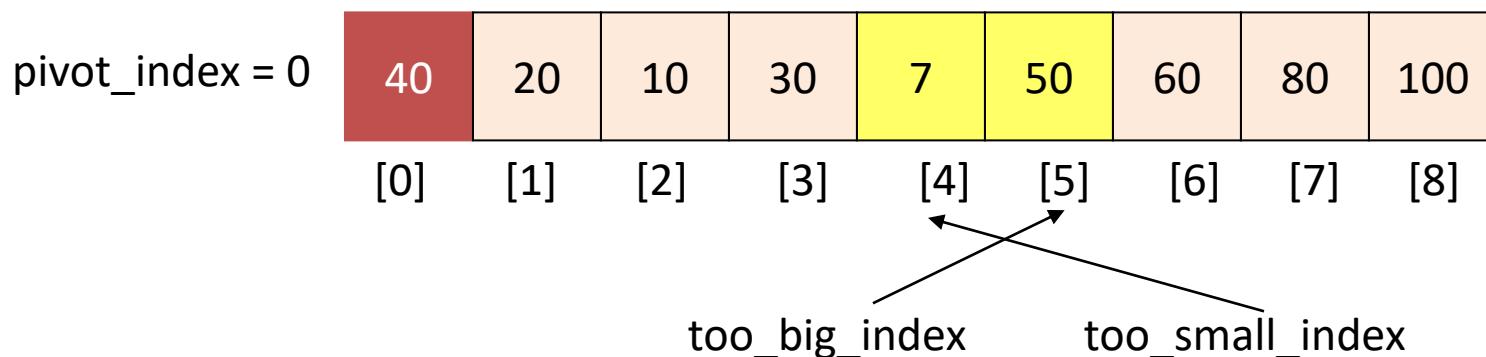
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
  - 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



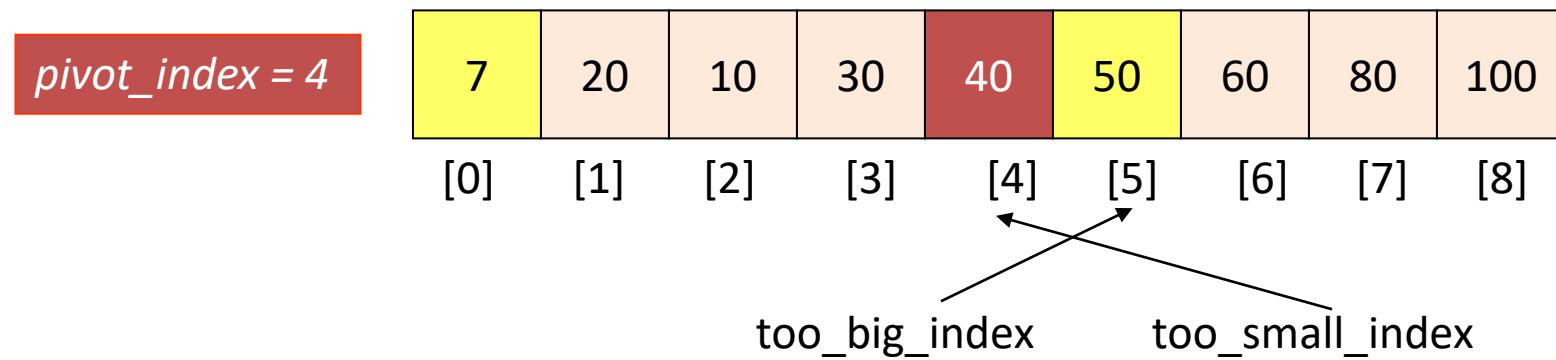
# Quick Sort

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
  5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$

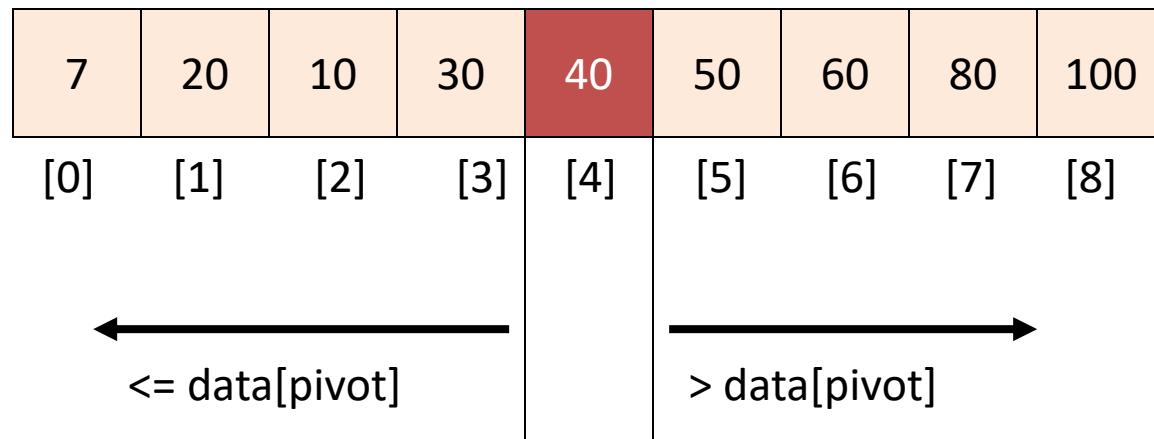


# Quick Sort

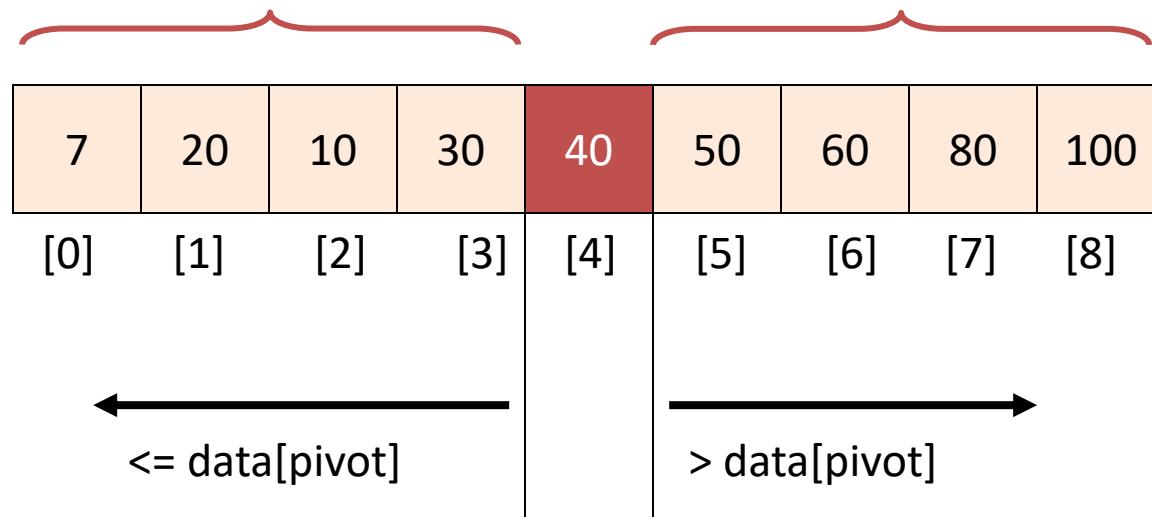
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



# Quick Sort Partition Result



# Recursion: Quick Sort Sub-Arrays



*Complete it as home work !!*

# Quick Sort Algorithm

```

procedure quickSort(left, right)

  if right-left <= 0
    return
  else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
  end if

end procedure

```

```

function partitionFunc(left, right, pivot)
  leftPointer = left
  rightPointer = right - 1

  while True do
    while A[++leftPointer] < pivot do
      //do-nothing
    end while

    while rightPointer > 0 && A[--rightPointer] > pivot do
      //do-nothing
    end while

    if leftPointer >= rightPointer
      break
    else
      swap leftPointer,rightPointer
    end if

  end while

  swap leftPointer,right
  return leftPointer

end function

```

# Quick Sort Analysis

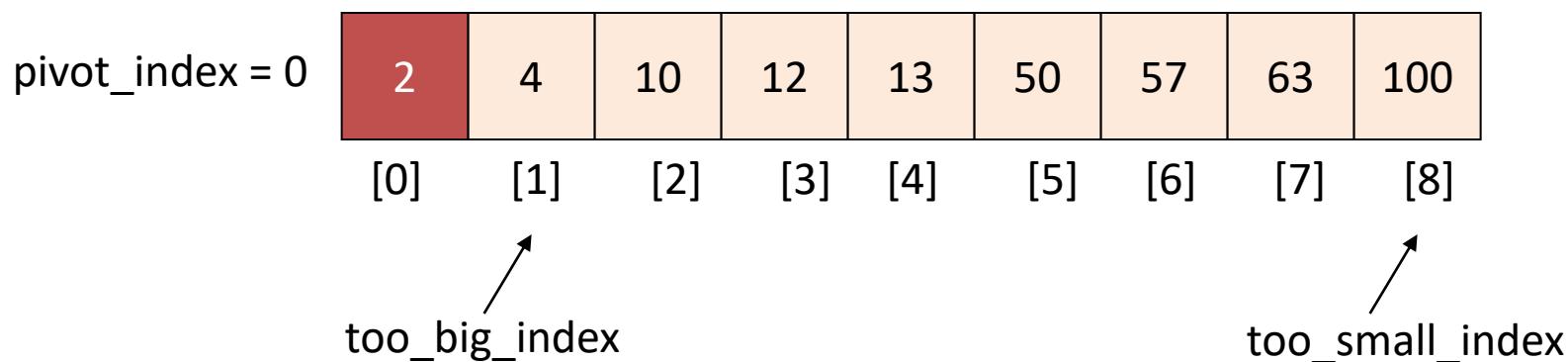
- Assume that keys are random, uniformly distributed.
- What is best case / average running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $\Theta(\log_2 n)$
  - Number of accesses in partition?  $\Theta(n)$
- Best case running time:  $\Theta(n \log_2 n)$
- Unable to get it intuitively, then solve :

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \text{ (sorting not required)} \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases} \quad \Rightarrow \quad \Theta(n \log_2 n)$$

# Quick Sort Worst Case Analysis

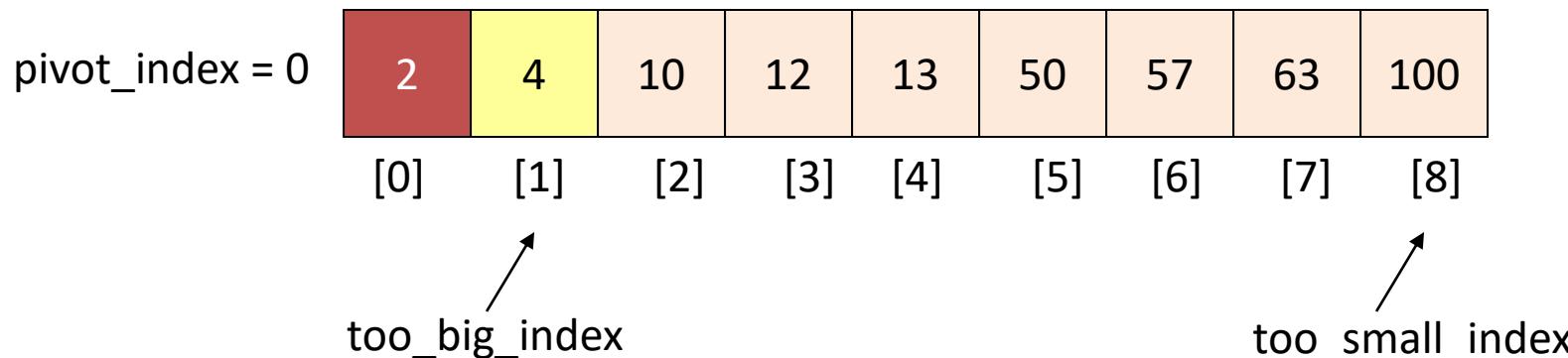


- Assume first element is chosen as pivot.
  - Assume we get array that is already in order:



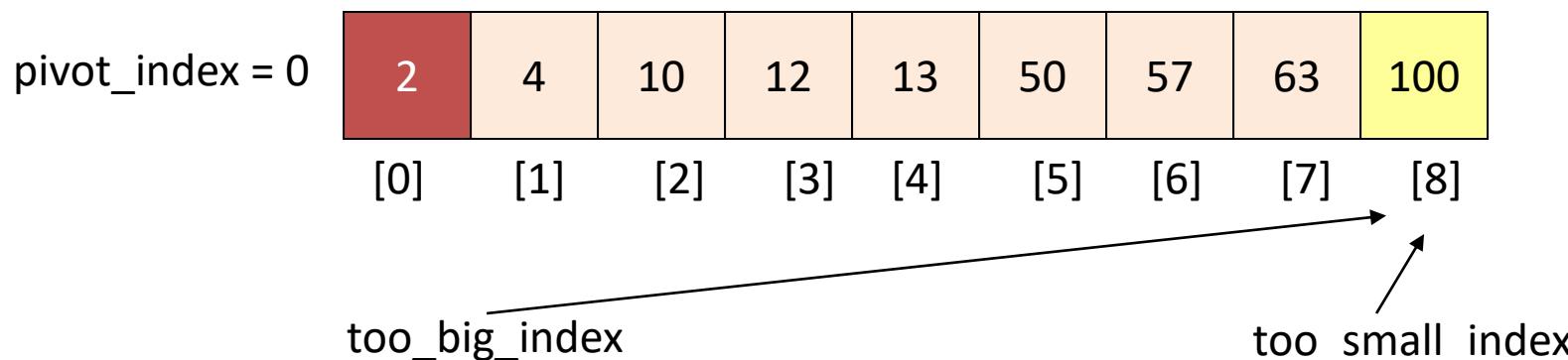
# Quick Sort Worst Case Analysis

- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$ 
  - $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$ 
  - $\text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$ 
  - swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
- 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



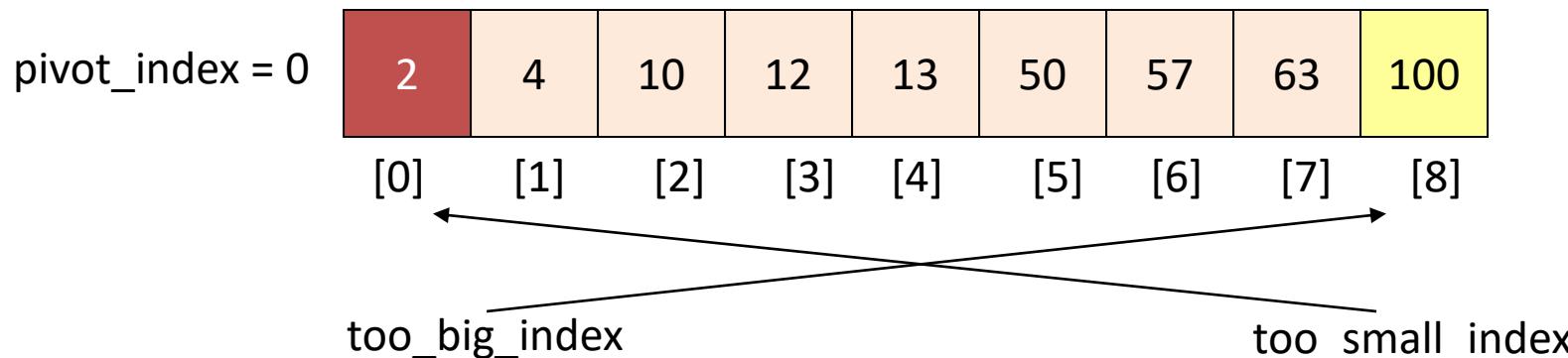
# Quick Sort Worst Case Analysis

- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$ 
  - $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$ 
  - $\text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$ 
  - swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
- 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



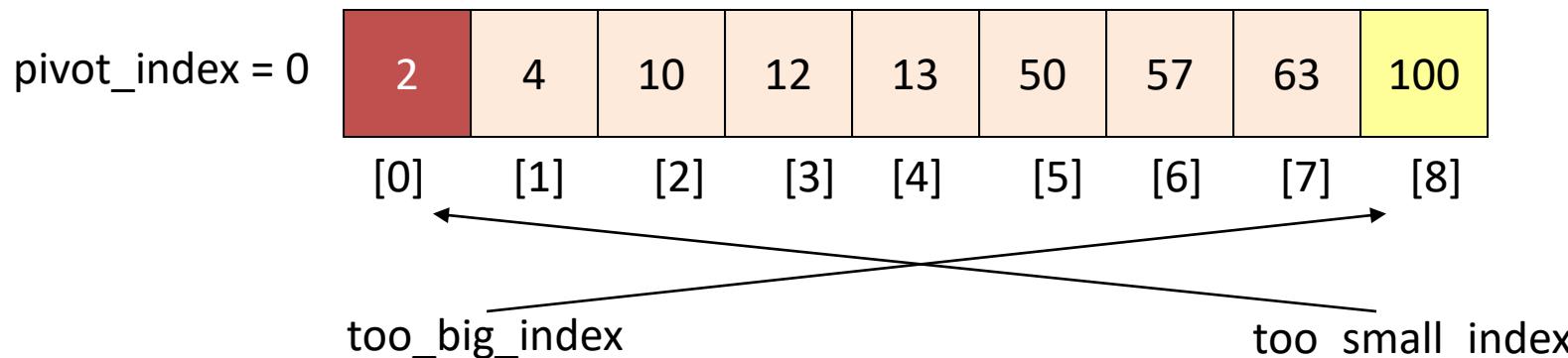
# Quick Sort Worst Case Analysis

1. While  $\text{data}[\text{too\_big\_index}] <= \text{data}[\text{pivot}]$ 
    - $\text{++too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$ 
    - $\text{--too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$ 
    - swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
  5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



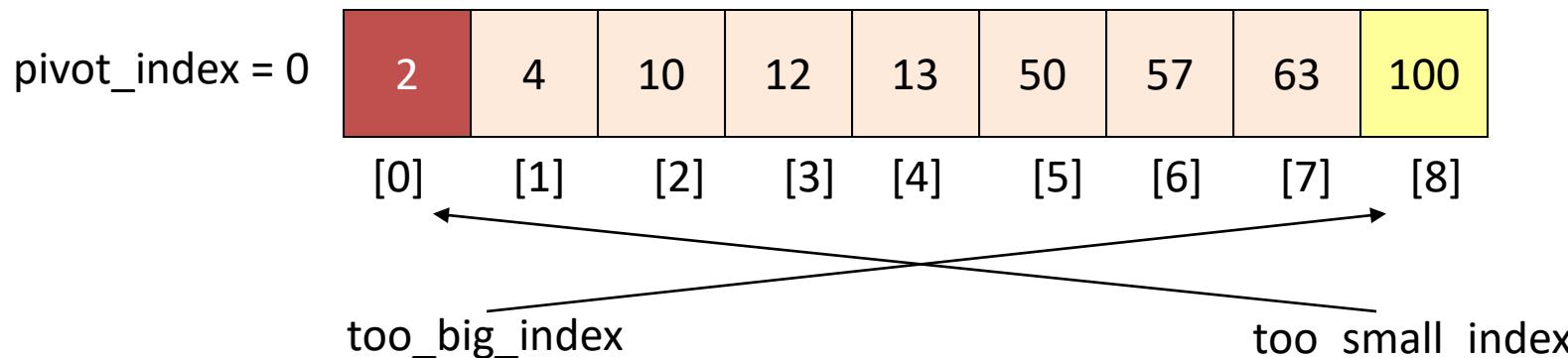
# Quick Sort Worst Case Analysis

1. While  $\text{data}[\text{too\_big\_index}] <= \text{data}[\text{pivot}]$ 
    - $\text{++too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$ 
    - $\text{--too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$ 
    - swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
  5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



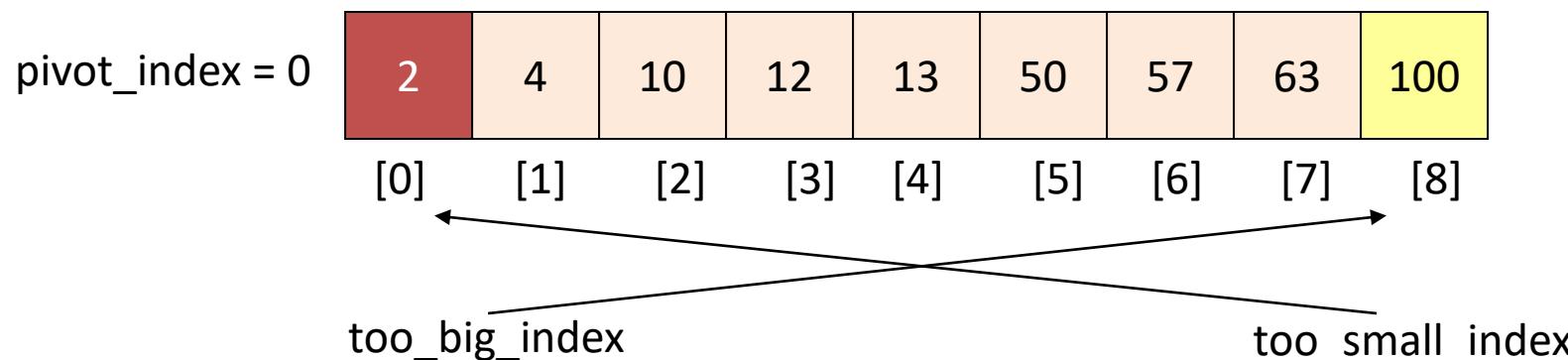
# Quick Sort Worst Case Analysis

1. While  $\text{data}[\text{too\_big\_index}] <= \text{data}[\text{pivot}]$ 
    - $\text{++too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$ 
    - $\text{--too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$ 
    - swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
  5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



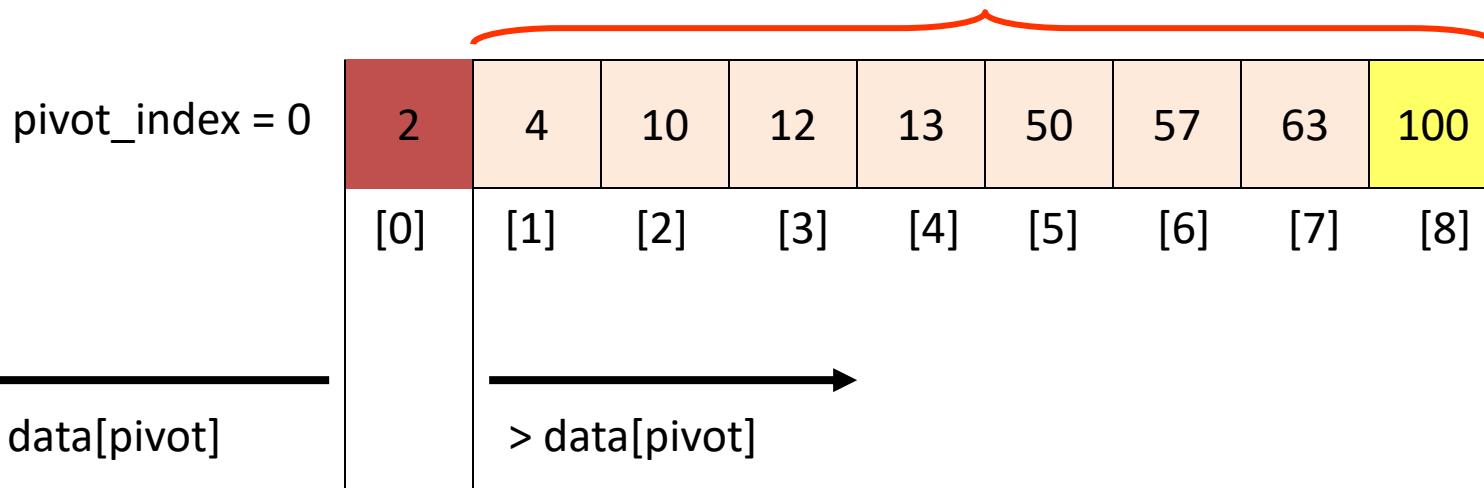
# Quick Sort Worst Case Analysis

1. While  $\text{data}[\text{too\_big\_index}] <= \text{data}[\text{pivot}]$ 
    - $\text{++too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$ 
    - $\text{--too\_small\_index}$
  3. If  $\text{too\_big\_index} < \text{too\_small\_index}$ 
    - swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
  - 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



# Quick Sort Worst Case Analysis

1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$ 
  - $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$ 
  - $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$ 
  - swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



# Quick Sort Worst Case Analysis



- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?  **$O(n)$**
  - Number of accesses per partition?  **$O(n)$**
  - Overall :  **$O(n^2)$**

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(0) + T(n - 1) + n & \text{otherwise} \end{cases}$$

➤  **$O(n^2)$**

# Quick Sort Worst Case Analysis



- Assume that keys are random, uniformly distributed.
- Best case running time:  $\Theta(n \log_2 n)$
- Worst case running time? **O( $n^2$ )**
- How to avoid the worst case ?
  - One approach is to improve the pivot selection :
    - Pick median value of three elements from data array data[0], data[n/2], and data[n-1]. Use this median value as pivot.
- Use Randomization!

# Randomized Quick Sort

- The goal of randomized quick sort is to decrease the chances of getting the worst case complexity.
- In the below example, quick sort will perform poorly and complexity would be  $O(n^2)$ .

Example :

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|

- In randomized Quick sort, either make the input randomized by shuffling the input or by choosing the pivot randomly.
- Even with Randomized Quick sort, there can be a situation that the complexity is  $O(n^2)$  but experientially it is proven that that case is rare! Probabilistically also its rare !! (Try to think ...)
- *A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. More on it in your next semester!*

# Integer Multiplication

**Add:** Given two n-bit integers a and b, compute  $a + b$ .

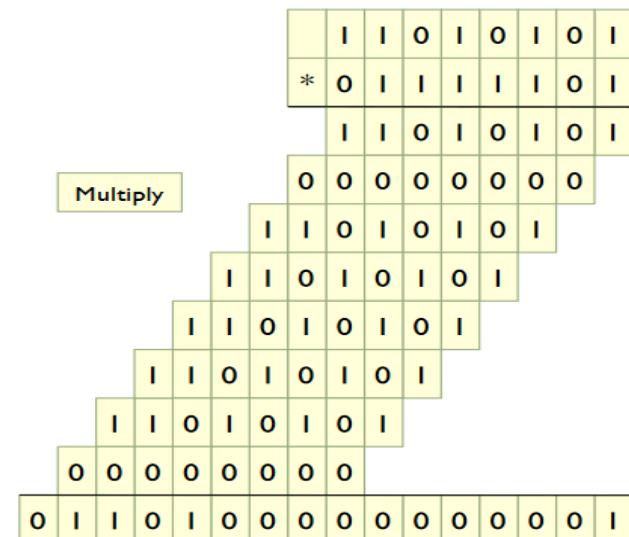
|     |   |   |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|---|--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|---|---|---|---|---|---|---|
| Add | <table style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td></td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>+</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td></td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> </table> | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |  | 1 | 1 | 0 | 1 | 0 | 1 | 0 | + | 0 | 1 | 1 | 1 | 1 | 1 | 0 |  | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1   | 1   | 1 | 1 | 1 | 1 | 0 | 1 |   |   |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |
|     | 1   | 1 | 0 | 1 | 0 | 1 | 0 |   |   |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |
| +   | 0   | 1 | 1 | 1 | 1 | 1 | 0 |   |   |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |
|     | 1   | 0 | 1 | 0 | 1 | 0 | 0 |   |   |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |

$\Theta(n)$  bit operations.

**Multiply:** Given two n-bit integers a and b, compute  $a \times b$ .

The “grade school” method:

$\Theta(n^2)$  bit operations.



# Integer Multiplication

To multiply two 2-digit integers:

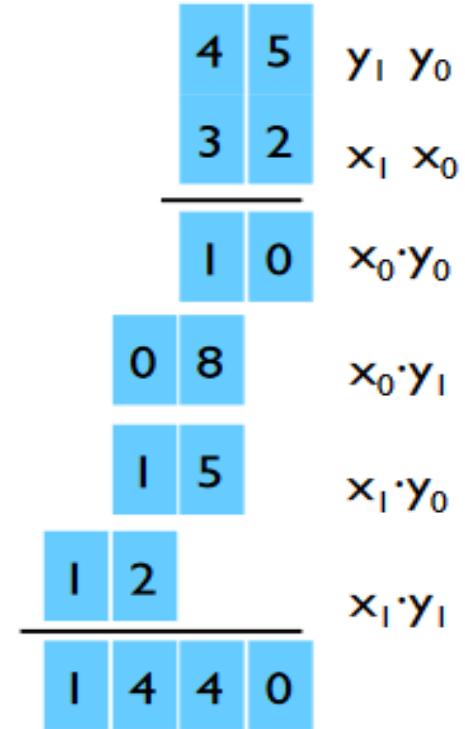
- Multiply *four* 1-digit integers.
- Add, shift some 2-digit integers to obtain result.

$$x = 10 \cdot x_1 + x_0$$

$$y = 10 \cdot y_1 + y_0$$

$$xy = (10 \cdot x_1 + x_0)(10 \cdot y_1 + y_0)$$

$$= 100 \cdot x_1 y_1 + 10 \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0$$



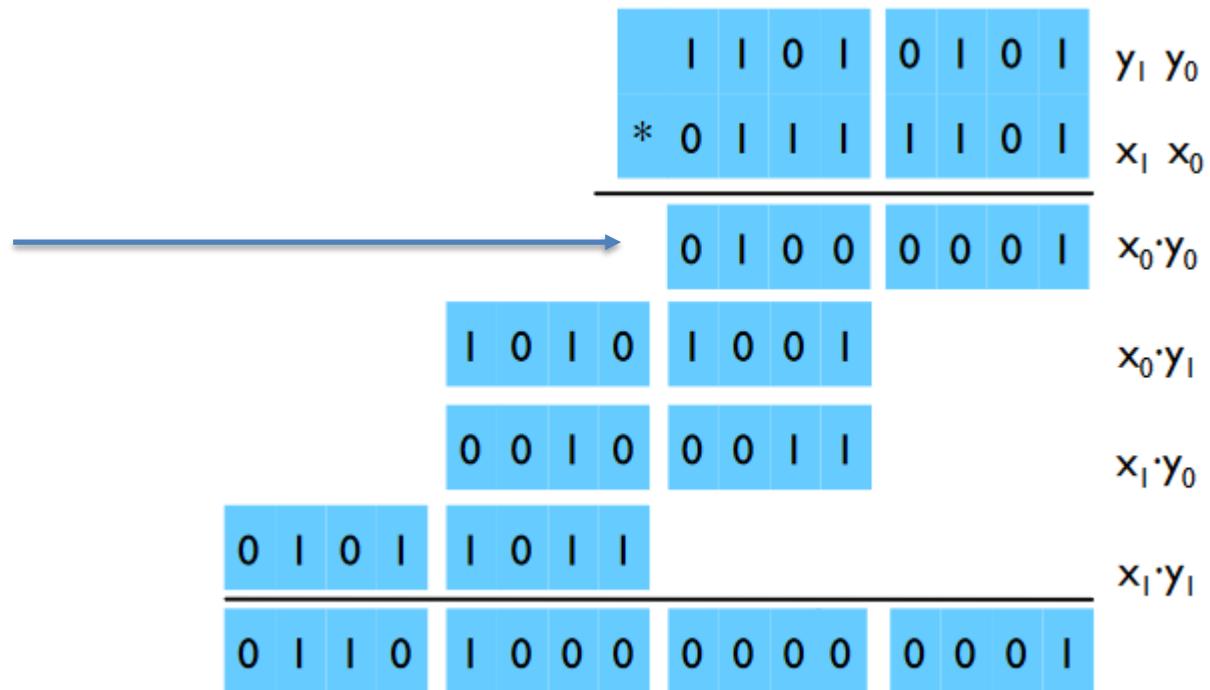
Same idea works for long integers – Can split them into 4 half-sized integers !!

# Integer Multiplication

To multiply two n-bit integers:

- Multiply *four*  $\frac{1}{2}$  n-bit integers.
- Add two  $\frac{1}{2}$  n-bit integers, and shift to obtain result.

$$\begin{array}{r}
 0101 \\
 1101 \\
 \hline
 ,0101 \\
 ,0000 * \\
 ,0101 ** \\
 ,0101 *** \\
 \hline
 1000001 \\
 \hline
 \Rightarrow 0100\ 0001
 \end{array}$$



# Integer Multiplication

To multiply two n-bit integers:

- Multiply *four*  $\frac{1}{2}$  n-bit integers.
- Add two  $\frac{1}{2}n$ -bit integers, and shift to obtain result.

$$x = 2^{n/2} \cdot x_1 + x_0$$

$$y = 2^{n/2} \cdot y_1 + y_0$$

$$\begin{aligned} xy &= (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) \\ &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \end{aligned}$$

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)$$

|   |   |   |   |   |   |   |                 |
|---|---|---|---|---|---|---|-----------------|
|   | 1 | 1 | 0 | 1 | 0 | 1 | $y_1 \cdot y_0$ |
| * | 0 | 1 | 1 | 1 | 1 | 0 | $x_1 \cdot x_0$ |
|   | 0 | 1 | 0 | 0 | 0 | 0 | $x_0 \cdot y_0$ |
|   | 1 | 0 | 1 | 0 | 1 | 0 | $x_0 \cdot y_1$ |
|   | 0 | 0 | 1 | 0 | 0 | 1 | $x_1 \cdot y_0$ |
|   | 0 | 1 | 0 | 1 | 1 |   | $x_1 \cdot y_1$ |
|   | 0 | 1 | 1 | 0 | 1 | 0 |                 |
|   | 1 | 0 | 0 | 0 | 0 | 1 |                 |

# Integer Multiplication Key Trick

Key trick: 2 multiplies for the price of 1

$$\begin{aligned}
 x &= 2^{n/2} \cdot x_1 + x_0 \\
 y &= 2^{n/2} \cdot y_1 + y_0 \\
 xy &= (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) \\
 &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0
 \end{aligned}$$

*Instead of 4 subproblems,  
we only need 3 (with the  
help of clever insight).*

$$\begin{aligned}
 \alpha &= x_1 + x_0 \\
 \beta &= y_1 + y_0 \\
 \alpha\beta &= (x_1 + x_0)(y_1 + y_0) \\
 &= x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
 (x_1 y_0 + x_0 y_1) &= \alpha\beta - x_1 y_1 - x_0 y_0
 \end{aligned}$$

# Karatsuba Multiplication

To multiply two n-bit integers:

- Add two  $\frac{1}{2}n$  bit integers.
- Multiply three  $\frac{1}{2}n$ -bit integers.
- Add, subtract, and shift  $\frac{1}{2}n$ -bit integers to obtain result.

$$x = 2^{n/2} \cdot x_1 + x_0$$

$$y = 2^{n/2} \cdot y_1 + y_0$$

$$\begin{aligned} xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\ &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) + x_0 y_0 \end{aligned}$$

**A**

**B**

**A**

**C**

**C**

$$\begin{aligned} T(n) &= 3 T(n/2) + O(n) \\ &= O(n^{\log 3}) \\ &= \mathbf{O(n^{1.584...})} \end{aligned}$$

# Karatsuba Multiplication Simplified

Three subproblems:

$$- a = x_H y_H$$

$$- d = x_L y_L$$

$$- e = (x_H + x_L)(y_H + y_L) - a - d$$

$$\text{Then } xy = a r^n + e r^{n/2} + d$$

Where,  $r$  is the radix/base of the number system.

H/1 stands for higher order terms

L/0 stands for lower order terms

# Worked Example

Compute  $1234 * 4321$  using Karatsuba's Insight!

## Solution:

Firstly,  $X = 1234$  and  $Y = 4321$

$X_H = 12$ ,  $X_L = 34$  and  $Y_H = 43$ ,  $Y_L = 21$

- $a_1 = 12 * 43$
- $d_1 = 34 * 21$
- $e_1 = (12+34) * (43+21) - a_1 - d_1$   
 $= 46 * 64 - a_1 - d_1$

➤ *Need to recursively solve these !! We have 3 subproblems,  $a_1$ ,  $d_1$  and  $e_1$*

$$\begin{aligned} - a &= x_H y_H \\ - d &= x_L y_L \\ - e &= (x_H + x_L)(y_H + y_L) - a - d \end{aligned}$$

Then  $xy = a r^n + e r^{n/2} + d$

# Worked Example

*Solving for 1<sup>st</sup> subproblem  $a_1$  :*

Firstly,  $X = 12$  and  $Y = 43$

$X_H = 1$  ,  $X_L = 2$  and  $Y_H = 4$  ,  $Y_L = 3$

$$a_1 = 12 * 43$$

Subproblems:

- $a_2 = 1 * 4 = 4$
- $d_2 = 2 * 3 = 6$
- $e_2 = (1+2)(4+3) - a_2 - d_2$   
 $= 21 - 4 - 6 = 11$

Solution to  $a_1 = 4 * 10^2 + 11 * 10 + 6$  [ 10 because base is decimal,  $n=2$  as its two digit multiplication ]

$$= 516$$

$$\begin{aligned} - a &= x_H y_H \\ - d &= x_L y_L \\ - e &= (x_H + x_L)(y_H + y_L) - a - d \end{aligned}$$

$$\text{Then } xy = a r^n + e r^{n/2} + d$$

# Worked Example

*Solving for 2<sup>nd</sup> subproblem  $d_1$  :*

Firstly,  $X = 34$  and  $Y = 21$

$X_H = 3$ ,  $X_L = 4$  and  $Y_H = 2$ ,  $Y_L = 1$

$$d_1 = 34 * 21$$

-  $a = X_H Y_H$   
 -  $d = X_L Y_L$   
 -  $e = (X_H + X_L)(Y_H + Y_L) - a - d$   
 Then  $xy = a r^n + e r^{n/2} + d$

## Subproblems:

- $a_2 = 3 * 2 = 6$
- $d_2 = 4 * 1 = 4$
- $e_2 = (3+4)(2+1) - a_2 - d_2$   
 $= 21 - 6 - 4 = 11$

Solution to  $d_1 = 6 * 10^2 + 11 * 10 + 4$  [ 10 because base is decimal,  
 $n=2$  as its two digit multiplication ]

$$= 714$$

# Worked Example

*Solving for 3<sup>rd</sup> subproblem  $e_1$  :*

Firstly,  $X = 46$  and  $Y = 64$

$X_H = 4$ ,  $X_L = 6$  and  $Y_H = 6$ ,  $Y_L = 4$

$e_1 = 46 * 64 - a_1 - d_1$  (where  $a_1, d_1$  are solutions of previous subproblems)

## Subproblems:

- $a_2 = 4 * 6 = 24$
- $d_2 = 6 * 4 = 24$
- $e_2 = (4+6)(6+4) - a_2 - d_2$   
 $= 100 - 24 - 24$   
 $= 52$

–  $a = x_H y_H$   
 –  $d = x_L y_L$   
 –  $e = (x_H + x_L)(y_H + y_L) - a - d$   
 Then  $xy = a r^n + e r^{n/2} + d$

Solution to  $e_1 = [ 24 * 10^2 + 52 * 10 + 24 ] - 516 - 714$   
 $= 1714$

# Worked Example

*Solving for final answer:*

- $a = x_H y_H$
- $d = x_L y_L$
- $e = (x_H + x_L)(y_H + y_L) - a - d$

Then  $xy = a r^n + e r^{n/2} + d$

|  |  |
|--|--|
| <p>Solution of 1<sup>st</sup> Subproblem</p>   | <p>Solution of 3<sup>rd</sup> Subproblem</p> |
| $1234 * 4321 = 516 * 10^4 + 1714 * 10^2 + 714$ | $= 5,332,114$                                |

Solution of 2<sup>nd</sup> Subproblem

# Integer Multiplication Algorithm

---

function multiply( $x, y$ )

Input: Positive integers  $x$  and  $y$ , in binary

Output: Their product

$n = \max(\text{size of } x, \text{ size of } y)$

if  $n = 1$ : return  $xy$

$x_L, x_R = \text{leftmost } \lceil n/2 \rceil, \text{ rightmost } \lfloor n/2 \rfloor \text{ bits of } x$

$y_L, y_R = \text{leftmost } \lceil n/2 \rceil, \text{ rightmost } \lfloor n/2 \rfloor \text{ bits of } y$

$P_1 = \text{multiply}(x_L, y_L)$

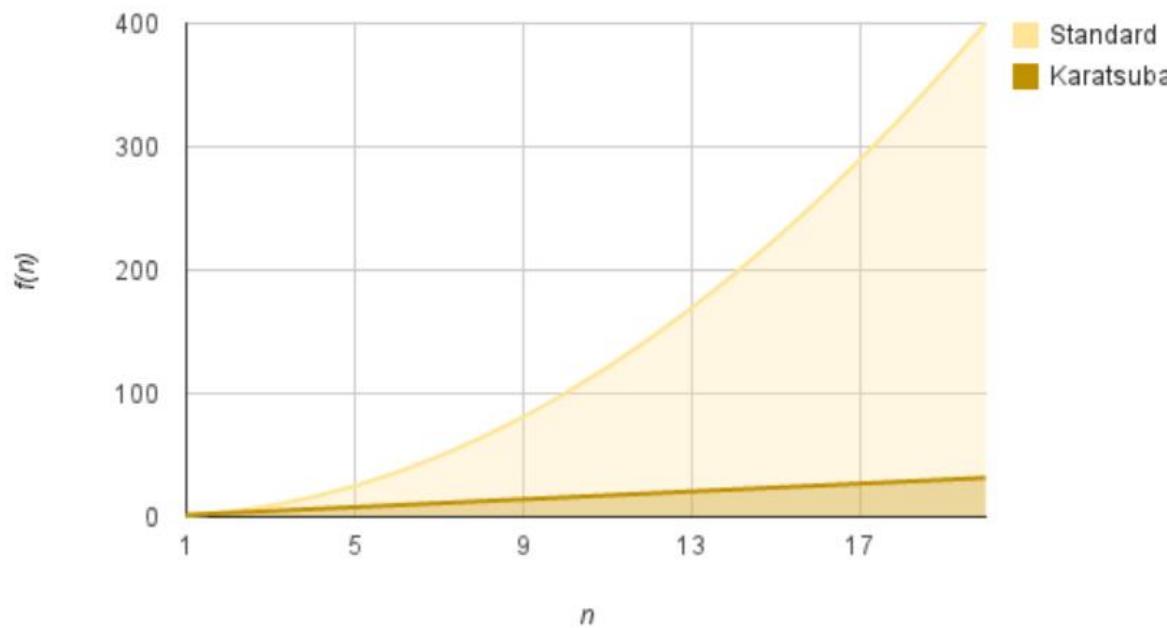
$P_2 = \text{multiply}(x_R, y_R)$

$P_3 = \text{multiply}(x_L + x_R, y_L + y_R)$

return  $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

# Power of Karatsuba's Algo

- Assuming that we replace two of the multiplications with only one makes the program faster. The question is how fast. Karatsuba improves the multiplication process by replacing the initial complexity of  $O(n^2)$  by  $O(n^{1.59})$ , which as you can see on the diagram below is much faster for big n.



# Exercises Set 1

---

## Exercise-1:

Assume that the sequence to be sorted is represented as a doubly linked list.

Given this,

- (a) What is the complexity of divide step?
- (b) What is the complexity of merge step?
- (c) Characterize the performance of merge sort on doubly linked list as a recurrence & solve it.
- (d) Characterize the result obtained in (c) using Big Oh notation

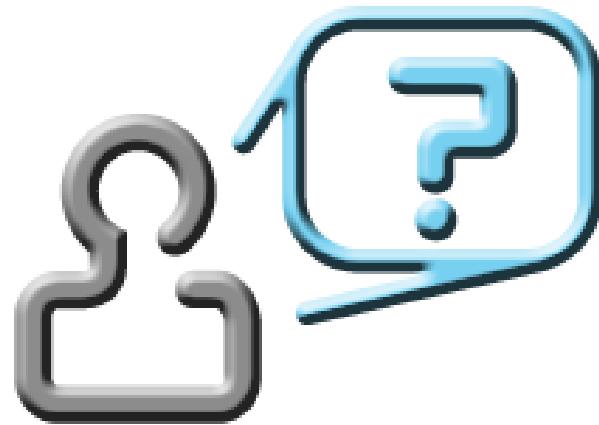
## Exercise-2:

Assume you are given a sequence of keys with duplicates. How will you adapt merge sort algorithm to produce sorted array as output which has no duplicate entries? Write the modified algorithm. What is its complexity?

# Exercise Set 2

---

- Apply Merge sort and quick sort on the following numbers and show the tracing :
  - a) 10,76,83,2,1,68,94,1003,368,53
  - b) 45,68,21,6,-6,2,78,3,6,457,1
  
- Apply Karatsuba's multiplication to multiply :
  - a) 4321 \* 5678
  - b) 98741 \* 78125 ( if n is odd, you can add a 0 in front and make it even ☺ )



*We will explore DP in the next class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS** Pilani

Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

**DSECLZG519**

Parthasarathy



# Contact Session #13

# DSECLZG519 –Dynamic Programming

# Agenda for Session # 13

---

- Recap of CS#12
  - Example of Integer multiplication
- Introduction to DP
- Example of DP using Fibonacci
- Transitive Closure - Warshall's Algorithm
- All Pair Shortest Path – Floyd's Algorithm
- Matrix chain multiplication using DP
- Exercises

# Motivation for DP

- What is “ $1+1+1+1+1=$ ” ?
- Counting “*Five!*”
- I add another “ $1+$ ” on the left.
- “What about that answer now?”
- “*Six!*”
- “How did you answer that it’s six so fast ?”
- “You just added one more!” “So you didn’t need to recount because you remembered there were Five!”
- Dynamic Programming is just a fancy way to say: remembering things to save time later!”

*Those who cannot remember the past are condemned to repeat it.*

# Introduction to Dynamic Programming



- Dynamic programming is a method of solving the problem with overlapping subproblems. In this approach, we precompute and **store** similar subproblem solutions to build the solution of a complex problem.
- This method works by dividing the problem into sub-problem and getting the solution for the sub-problems using which the solution for the given problem can be obtained.
- Once a sub-problem is solved, the result is stored in a table and never recalculated. When a sub-problem of earlier instance is encountered, instead of recalculating the values, it is simply retrieved from the table thus saving time.
- *Note: The word programming in dynamic programming has no particular connection to computer programming. Here, programming is the process of obtaining the plan to get the optimal solution.*

# Where do we need Dynamic Programming?

---

- If we are given a problem that can be broken down into smaller sub-problems and these smaller sub-problems can still be broken into smaller ones, and if we manage to find out that there are some **over-lapping** sub-problems, then we've encountered a DP problem.
- Dynamic Programming combines solutions to sub-problems. It is mainly used when solutions of the **same** subproblems are needed again and again. In this, computed solutions to subproblems are stored in a table so that these don't have to be recomputed.
- So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point in storing the solutions if they are not needed again. For example, Binary Search does not have an overlapping sub-problem. Whereas the recursive program of Fibonacci numbers has many overlapping sub-problems!
- DP can be used when the sub-problems are dependent. When they are independent subproblems, we can use D&C ☺
- The core idea of dynamic programming is to avoid repeated work by remembering partial results.

# Applications of DP Strategy

---

## ➤ Areas :

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ....

# Some Common Problems solved using DP



- Fibonacci Series
- Transitive Closure – Warshall's Algorithm
- All pair Shortest path – Floyd Warshall's Algorithm
- 0/1 Knapsack Problem
- Matrix Chain Multiplication
- Sum of subset problem
- Binomial co-efficient
- Longest common Sequence
- Bellman Ford Algorithm
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- And many more .....



*Self-Study & Next Semesters!*

# Fibonacci Series

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n - 2) + \text{fib}(n - 1) & \text{if } n > 1 \end{cases}$$

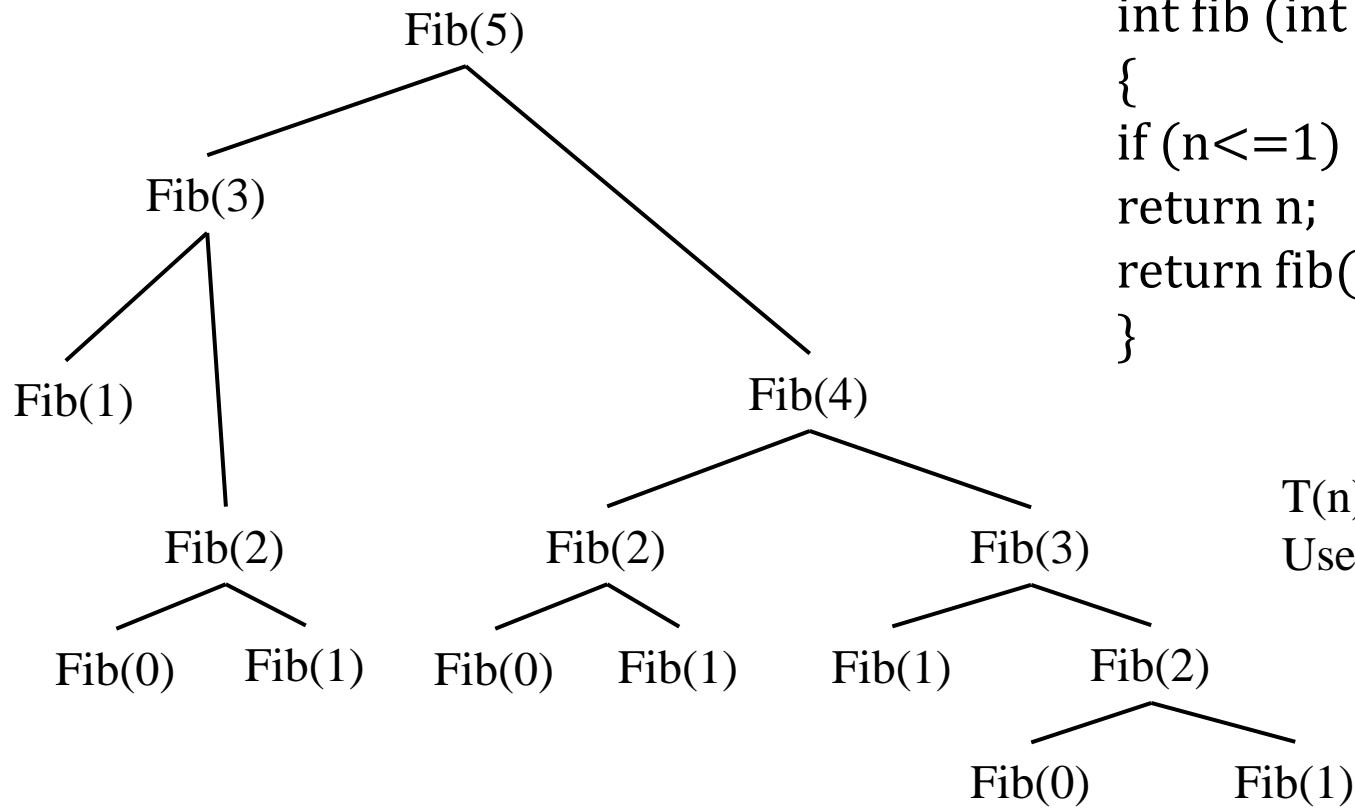
```
int fib (int n)
{
    if (n<=1)
        return n;
    return fib(n-2)+fib(n-1);
}
```

## The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377...

|                |                    |
|----------------|--------------------|
| <b>1+1=2</b>   | <b>13+21=34</b>    |
| <b>1+2=3</b>   | <b>21+34=55</b>    |
| <b>2+3=5</b>   | <b>34+55=89</b>    |
| <b>3+5=8</b>   | <b>55+89=144</b>   |
| <b>5+8=13</b>  | <b>89+144=233</b>  |
| <b>8+13=21</b> | <b>144+233=377</b> |

# Fibonacci Series



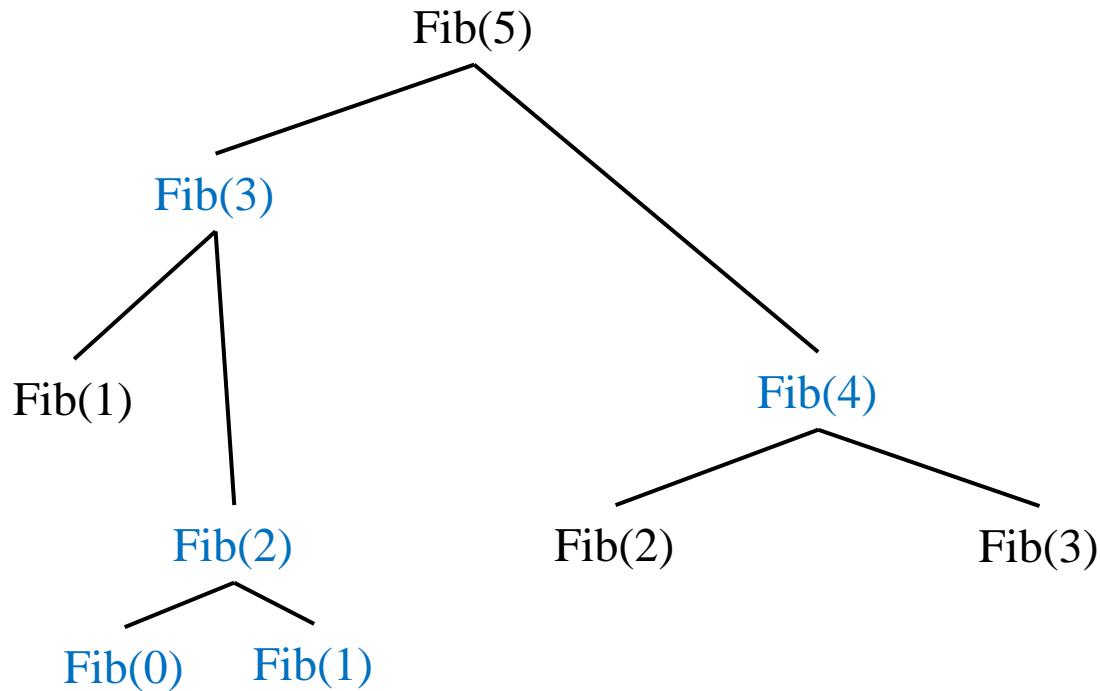
```

int fib (int n)
{
if (n<=1)
return n;
return fib(n-2)+fib(n-1);
}
  
```

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

Use master theorem =  $O(2^n)$

# Recursion with Memoization



```

int fib (int n)
{
if (n<=1)
return n;
If fn is in memory return fn
else
fn ←fib(n-2)+fib(n-1);
Save fn in memory
return fn
}
  
```

From the above figure it can be seen, subtrees that correspond to subproblems that have already been solved are pruned from this recursive call tree. The time complexity is thus **O(n)**. In short, **DP = recursion + re-use**

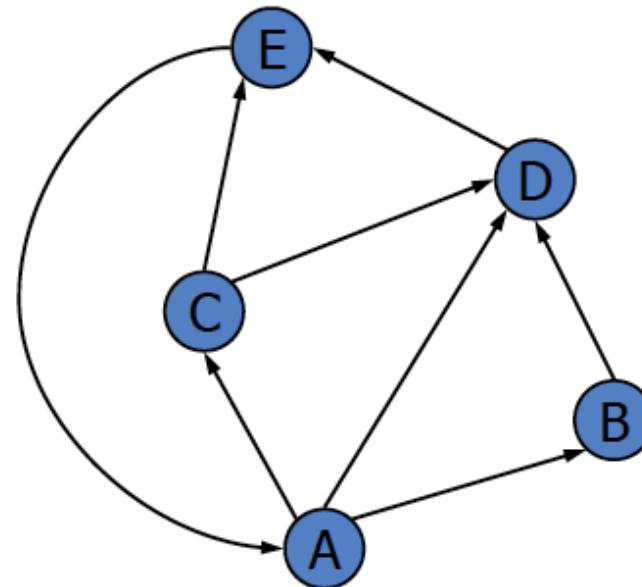
# Steps of Dynamic Programming

---

- Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the optimal solution for all smaller sub-problems in a bottom-up approach (preferably by using a table).
  - Construct an optimal solution from computed information
- 
- The common theme of any problem solved using dynamic programming is to computing solutions for smaller sub-problems and filling it into the table for future reference (while solving other subproblems).

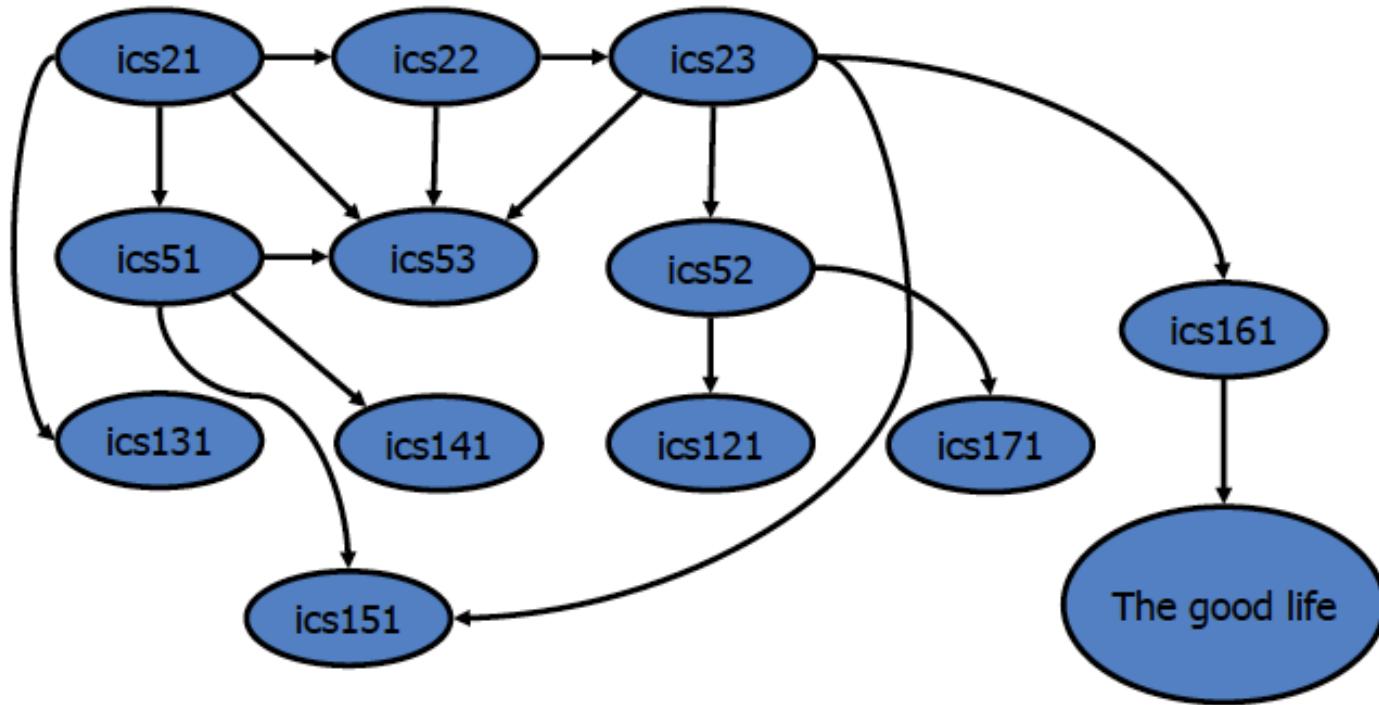
# Directed Graph

- A **digraph** is a graph whose edges are all directed ..
- Short for “directed graph”
- Applications
  - One-way streets
  - Flights
  - Task scheduling



# Digraph Application

Task / Job : Scheduling: Edge(a,b) means task **a** must be completed before **b** can be started.

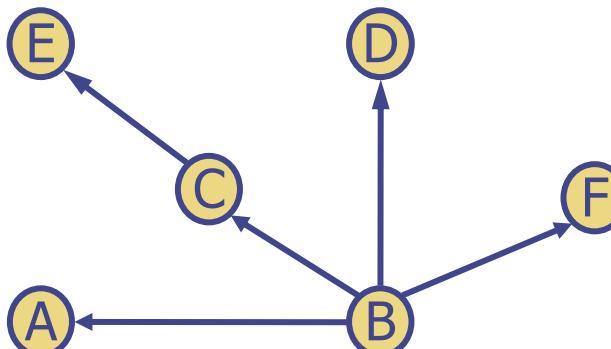
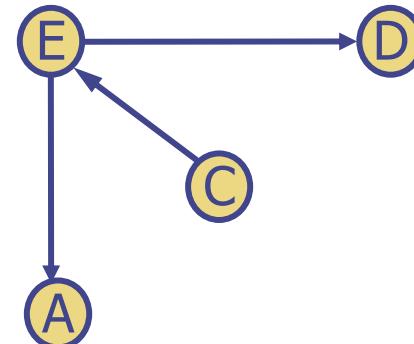
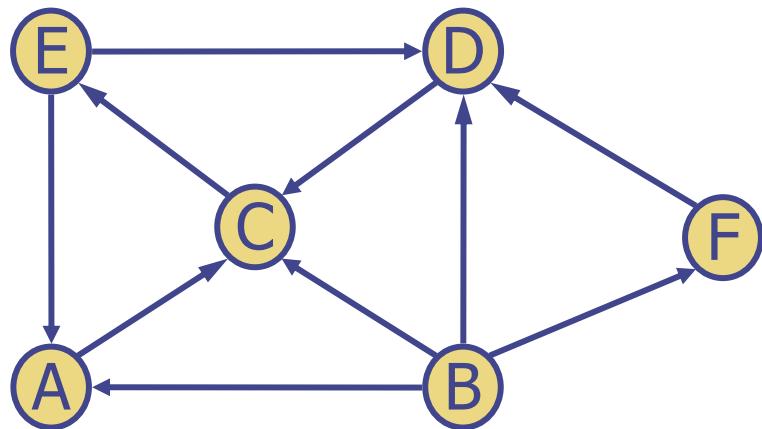


# Reachability

---

- A fundamental issue with directed graphs is the notion of reachability
- Reachability, which deals with determining where we can get to in a directed graph.
- Given vertices  $u$  and  $v$  of a digraph  $G$ , we say that  $u$  reaches  $v$  ( and  $v$  is **reachable** from  $u$ ) if  $G$  has a directed path from  $u$  to  $v$ .
- A vertex  $v$  reaches an edges  $(w,z)$  if  $v$  reaches the origin vertex  $w$  of the edge .

# Reachability Examples



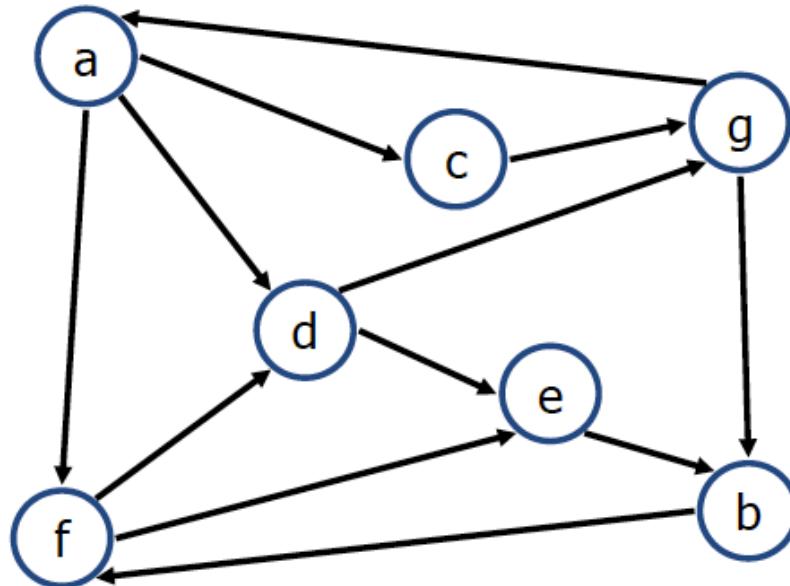
# Reachability

---

- Interesting problems that deal with reachability in a digraph G including the following
  - *Given vertices u and v, determine whether u reaches v ?*
- Find all the vertices of G that are reachable from a given vertex S.
- Determine whether G is strongly connected.
- Compute the transitive closure  $G^*$  of G.

# Strongly connected

- A digraph G is strongly connected, if for any two vertices u and v of G :  
    u reaches v **and** v reaches u.
- Each vertex can reach all other vertices:

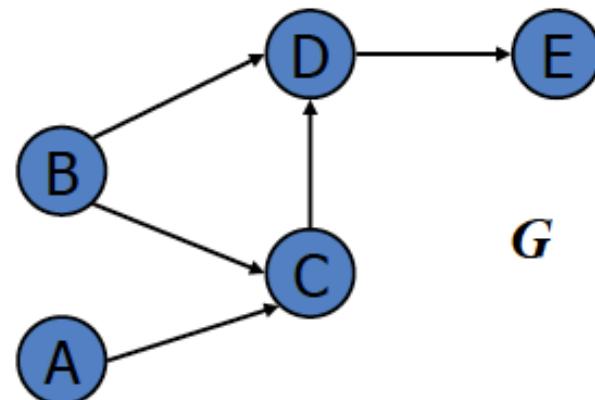


# Transitive closure

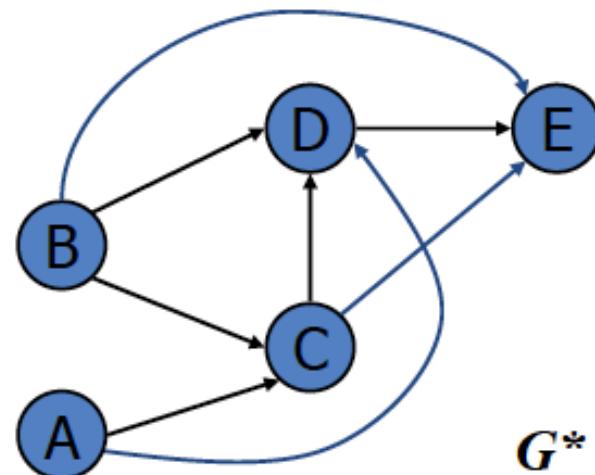
---

- The transitive closure provides reachability information about a digraph.
- Given a digraph  $G$ , the transitive closure of  $G$  is the digraph  $G^*$  such that
  - ✓  $G^*$  has the same vertices as  $G$
  - ✓ If  $G$  has a directed path from  $u$  to  $v$  ( $u \neq v$ ),  $G^*$  has a directed edge from  $u$  to  $v$ .

# Transitive closure (Example)

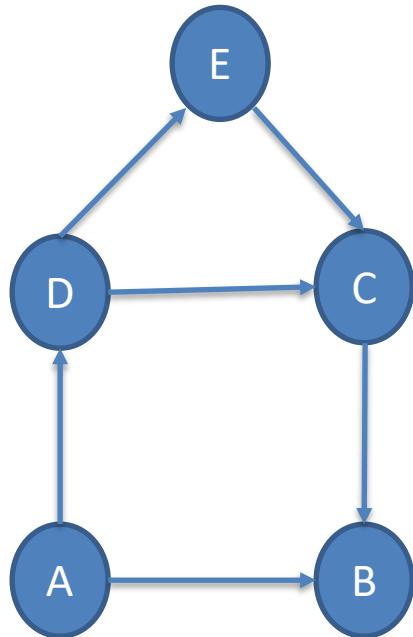


$G$



$G^*$

# Transitive Closure & Reachability Matrix



Adjacency Matrix A :

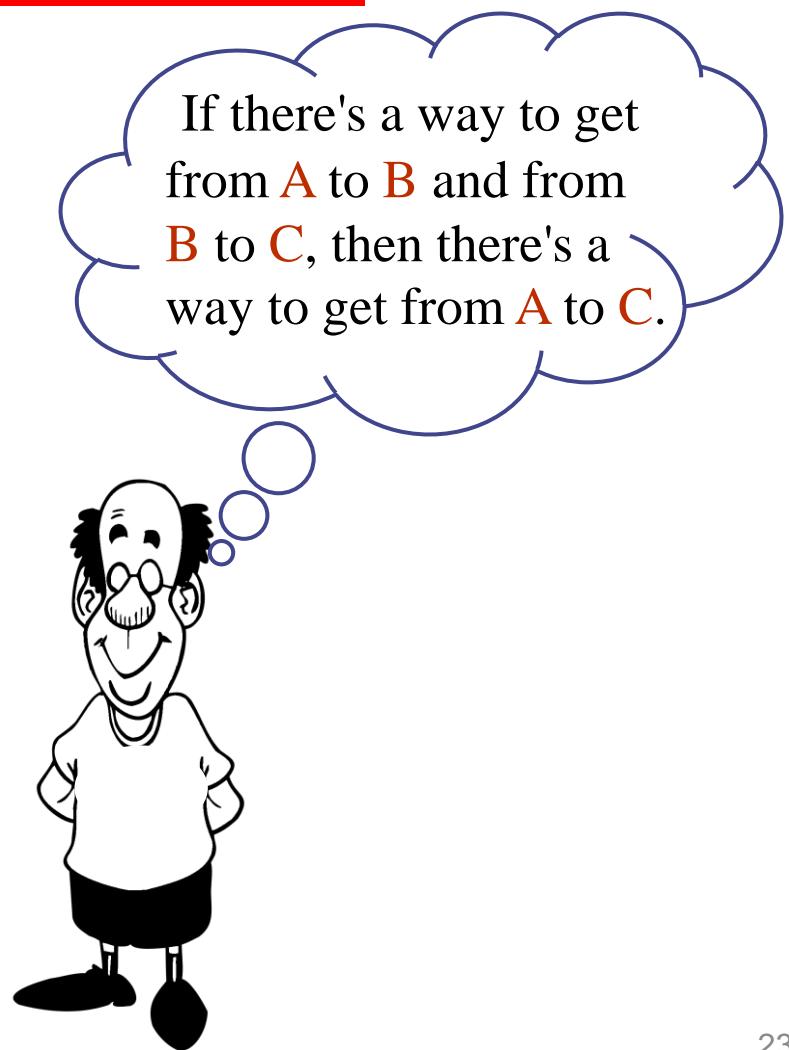
|   | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 | 0 |
| B | 0 | 0 | 0 | 0 | 0 |
| C | 0 | 1 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | 0 | 1 |
| E | 0 | 0 | 1 | 0 | 0 |

Reachability Matrix/  
Transitive Closure T :

|   | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 1 |
| B | 0 | 1 | 0 | 0 | 0 |
| C | 0 | 1 | 1 | 0 | 0 |
| D | 0 | 1 | 1 | 1 | 1 |
| E | 0 | 1 | 1 | 0 | 1 |

# Computing the transitive Closure

- We can perform DFS starting at each vertex to see which vertices w are reachable from v, adding an edge  $(v,w)$  to the transitive closure for each such W.
  - $O(n(n+m))$
- Use Warshall's algorithm.



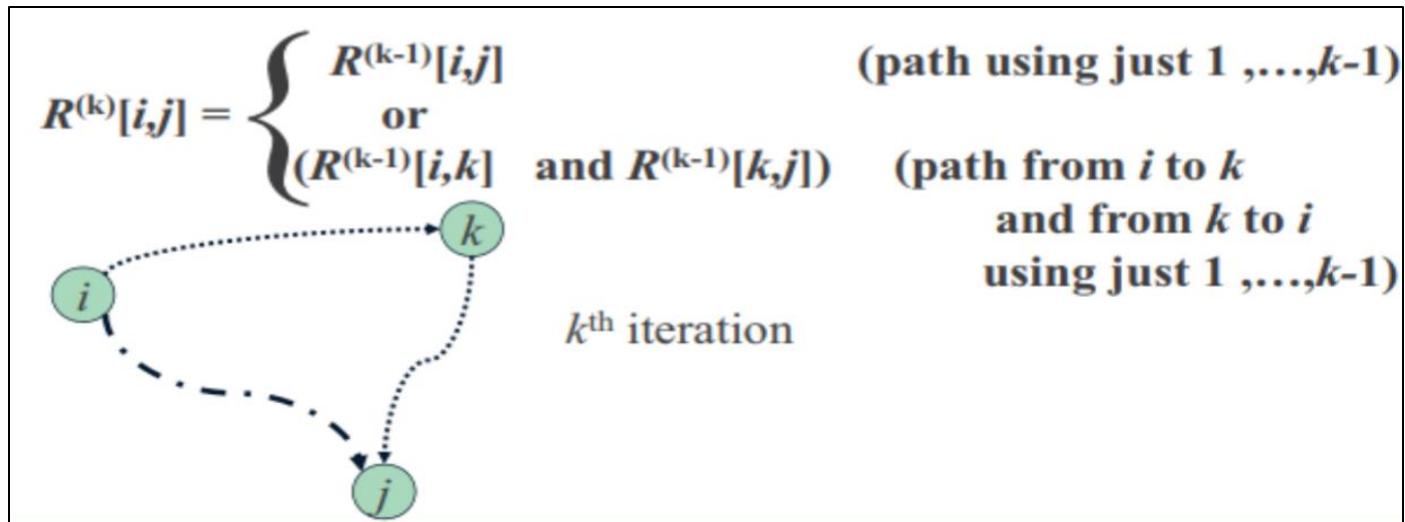
# Floyd-Warshall Transitive Closure

- Constructs the transitive closure of a given directed graph by a sequence of matrices :

$$\mathbf{R^0}, \mathbf{R^1}, \mathbf{R^2}, \mathbf{R^3} \dots\dots\dots \mathbf{R^{(n-1)}}, \mathbf{R^n}$$

- ✓ each of them are of order  $n \times n$  where
- ✓ ‘n’ - number of vertices in digraph.

- $\mathbf{R^n}$  – Transitive closure.
- Vertices are numbered from 1 to ‘n’.



# Floyd-Warshall Transitive Closure Formula for Matrix Generation



$$R^k[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

So,

$k=0$ , Adjacency Matrix

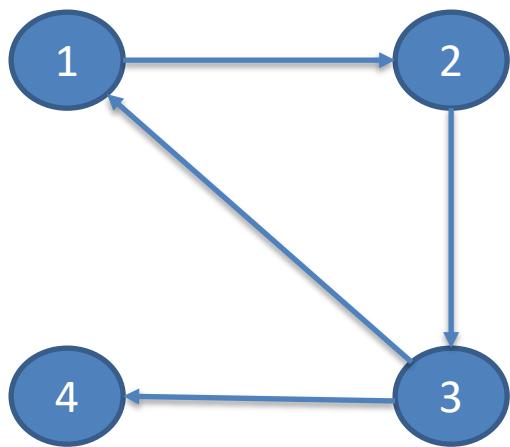
$K=1$ ,  $R^1[i,j] = R^{(0)}[i,j] \text{ or } (R^{(0)}[i,1] \text{ and } R^{(0)}[1,j])$

$K=2$ ,  $R^2[i,j] = R^{(1)}[i,j] \text{ or } (R^{(1)}[i,2] \text{ and } R^{(1)}[2,j])$

$K=3$ ,  $R^3[i,j] = R^{(2)}[i,j] \text{ or } (R^{(2)}[i,3] \text{ and } R^{(2)}[3,j])$

▪  
▪  
▪

# Example



$$R^0 =$$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^0$  is the adjacency matrix of the digraph.

# Example – Generate $R^1$ (Row1)

Now find  $R^1$  Matrix !

$$k=1, R^1[i,j] = R^{(0)}[i,j] \text{ or } (R^{(0)}[i,1] \text{ and } R^{(0)}[1,j])$$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^0 =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

$R^{(1)} =$

- ✓  $R^1[1,1] = R^{(0)}[1,1]$  (false). So, Check condition 2, ( $R^{(0)}[1,1]$  and  $R^{(0)}[1,1]$  are not 1 in  $R^0$ ) Hence,  $R^1[1,1] = 0$
- ✓  $R^1[1,2] = R^{(0)}[1,2]$  (true) . Hence,  $R^1[1,2] = 1$
- ✓  $R^1[1,3] = R^{(0)}[1,3]$  (false). So, Check condition 2, ( $R^{(0)}[1,1]$  and  $R^{(0)}[1,3]$  are not 1 in  $R^0$ ) Hence,  $R^1[1,3] = 0$
- ✓  $R^1[1,4] = R^{(0)}[1,4]$  (false). So, Check condition 2, ( $R^{(0)}[1,1]$  and  $R^{(0)}[1,4]$  are not 1 in  $R^0$ ) Hence,  $R^1[1,4] = 0$

# Example – Generate $R^1$ (Row 2)

Now find  $R^1$  Matrix !

$k=1, R^1[i,i] = R^{(0)}[i,i]$  or  $(R^{(0)}[i,1] \text{ and } R^{(0)}[1,j])$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^0 =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

$R^{(1)} =$

- ✓  $R^1[2,1] = R^{(0)}[2,1]$  (false). So, Check condition 2, ( $R^{(0)}[2,1]$  and  $R^{(0)}[1,1]$  are not 1 in  $R^0$ ) Hence,  $R^1[2,1] = 0$
- ✓  $R^1[2,2] = R^{(0)}[2,2]$  (false) .So, Check condition 2, ( $R^{(0)}[2,1]$  and  $R^{(0)}[1,2]$  are not 1 in  $R^0$ ) Hence,  $R^1[2,2] = 0$
- ✓  $R^1[2,3] = R^{(0)}[2,3]$  (true). Hence,  $R^1[2,3] = 1$
- ✓  $R^1[2,4] = R^{(0)}[2,4]$  (false). So, Check condition 2, ( $R^{(0)}[2,1]$  and  $R^{(0)}[1,4]$  are not 1 in  $R^0$ ) Hence,  $R^1[2,4] = 0$

# Example – Generate $R^1$ (Row 3)

Now find  $R^1$  Matrix !

$k=1, R^1[i,j] = R^{(0)}[i,j]$  or  $(R^{(0)}[i,1] \text{ and } R^{(0)}[1,j])$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^0 =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 |
| 4 |   |   |   |   |

$R^{(1)} =$

- ✓  $R^1[3,1] = R^{(0)}[3,1]$  (true). Hence,  $R^1[3,1] = 1$
- ✓  $R^1[3,2] = R^{(0)}[3,2]$  (false) . So, Check condition 2, ( $R^{(0)}[3,1]$  and  $R^{(0)}[1,2]$  are 1 in  $R^0$ ) Hence,  $R^1[3,2] = 1$
- ✓  $R^1[3,3] = R^{(0)}[3,3]$  (false). So, Check condition 2, ( $R^{(0)}[3,1]$  and  $R^{(0)}[1,3]$  are not 1 in  $R^0$ ) Hence,  $R^1[3,3] = 0$
- ✓  $R^1[3,4] = R^{(0)}[3,4]$  (true). Hence,  $R^1[3,4] = 1$

# Example – Generate $R^1$ (Row 4)

Now find  $R^1$  Matrix !

$$k=1, R^1[i,j] = R^{(0)}[i,j] \text{ or } (R^{(0)}[i,1] \text{ and } R^{(0)}[1,j])$$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^0 =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

- ✓  $R^1[4,1] = R^{(0)}[4,1]$  (false). So, Check condition 2, ( $R^{(0)}[4,1]$  and  $R^{(0)}[1,1]$  are not 1 in  $R^0$ ) Hence,  $R^1[4,1] = 0$
- ✓  $R^1[4,2] = R^{(0)}[4,2]$  (false) . So, Check condition 2, ( $R^{(0)}[4,1]$  and  $R^{(0)}[1,2]$  are not 1 in  $R^0$ ) Hence,  $R^1[4,2] = 0$
- ✓  $R^1[4,3] = R^{(0)}[4,3]$  (false). So, Check condition 2, ( $R^{(0)}[4,1]$  and  $R^{(0)}[1,3]$  are not 1 in  $R^0$ ) Hence,  $R^1[4,3] = 0$
- ✓  $R^1[4,4] = R^{(0)}[4,4]$  (false). So, Check condition 2, ( $R^{(0)}[4,1]$  and  $R^{(0)}[1,4]$  are not 1 in  $R^0$ ) Hence,  $R^1[4,4] = 0$  30

# Example – Generate R<sup>2</sup> (Row 1)

Now find R<sup>2</sup> Matrix

$$k=2, R^2[i,j] = R^{(1)}[i,j] \text{ or } (R^{(1)}[i,2] \text{ and } R^{(1)}[2,j])$$

$R^{(1)} =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^{(2)} =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

- ✓  $R^2[1,1] = R^{(1)}[1,1]$  (false). So, Check condition 2, ( $R^{(1)}[1,2]$  and  $R^{(1)}[2,1]$  are not 1 in  $R^1$ ) Hence,  $R^2[1,1] = 0$
- ✓  $R^2[1,2] = R^{(1)}[1,2]$  (true) . Hence,  $R^2[1,2] = 1$
- ✓  $R^2[1,3] = R^{(1)}[1,3]$  (false). So, Check condition 2, ( $R^{(1)}[1,2]$  and  $R^{(1)}[2,3]$  are 1 in  $R^1$ ) Hence,  $R^2[1,3] = 1$
- ✓  $R^2[1,4] = R^{(1)}[1,4]$  (false). So, Check condition 2, ( $R^{(1)}[1,2]$  and  $R^{(1)}[2,4]$  are not 1 in  $R^1$ ) Hence,  $R^2[1,4] = 0$

# Example – Generate $R^2$

Now find  $R^2$  Matrix

$k=2, R^2[i,j] = R^{(1)}[i,j] \text{ or } (R^{(1)}[i,2] \text{ and } R^{(1)}[2,j])$

$R^{(1)} =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^{(2)} =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

Similarly, we calculate and fill all the cells of  $R^{(2)}$

*Tip : Try them offline and verify the  $R^{(2)}$  matrix that we have got here ☺*

# Example – Generate $R^3$ (Row1)

Now find  $R^3$  Matrix

$k=3, R^3[i,j] = R^{(2)}[i,j]$  or  $(R^{(2)}[i,3] \text{ and } R^{(2)}[3,j])$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

- ✓  $R^3[1,1] = R^{(2)}[1,1]$  (false). So, Check condition 2,  $(R^{(2)}[1,3] \text{ and } R^{(1)}[3,1] \text{ are 1 in } R^2)$  Hence,  $R^3[1,1] = 1$
- ✓  $R^3[1,2] = R^{(2)}[1,2]$  (true) . Hence,  $R^3[1,2] = 1$
- ✓  $R^3[1,3] = R^{(2)}[1,3]$  (true). Hence,  $R^3[1,3] = 1$
- ✓  $R^3[1,4] = R^{(2)}[1,4]$  (false). So, Check condition 2,  $(R^{(2)}[1,3] \text{ and } R^{(2)}[3,4] \text{ are 1 in } R^2)$  Hence,  $R^3[1,4] = 1$

# Example – Generate $R^3$

Now find  $R^3$  Matrix

$k=3, R^3[i,j] = R^{(2)}[i,j] \text{ or } (R^{(2)}[i,3] \text{ and } R^{(2)}[3,j])$

$R^{(2)} =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

$R^{(3)} =$

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

Similarly, we calculate and fill all the cells of  $R^{(3)}$

*Tip : Try them offline and verify the  $R^{(3)}$  matrix that we have got here ☺*

# Example – Generate R<sup>4</sup>

Now find R<sup>4</sup> Matrix

k=4, R<sup>4</sup>[i,j] = R<sup>(3)</sup>[i,j] or (R<sup>(3)</sup>[i,4] and R<sup>(3)</sup>[4,j])

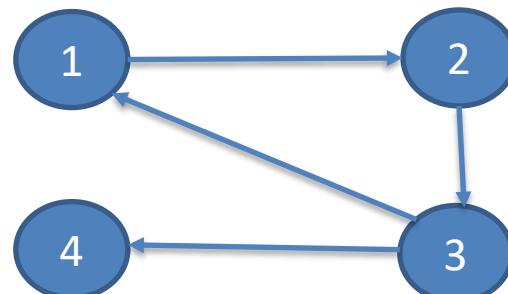
|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

R<sup>(3)</sup> =

R<sup>(4)</sup> =



# Warshall's Algorithm

```
Algorithm: warshall (A[1..n,1..n])
//implements warshall's algorithm for computing the transitive closure.
//input: the adjacency matrix A of a digraph with n-vertices
//output: the transitive closure of the digraph.
```

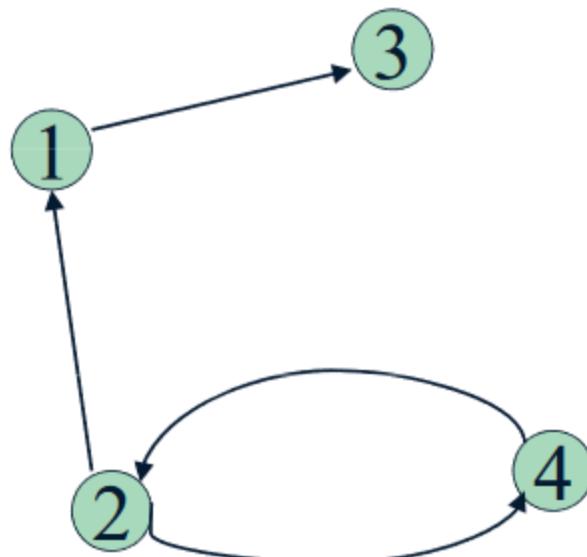
```
R(0) ← A
for K ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R(k) [i,j] ← R(k-1) [i,j] or (R(k-1) [i,k] and R(k-1) [k,j])
return R(n)
```

Now its your turn to find the time complexity: 😊

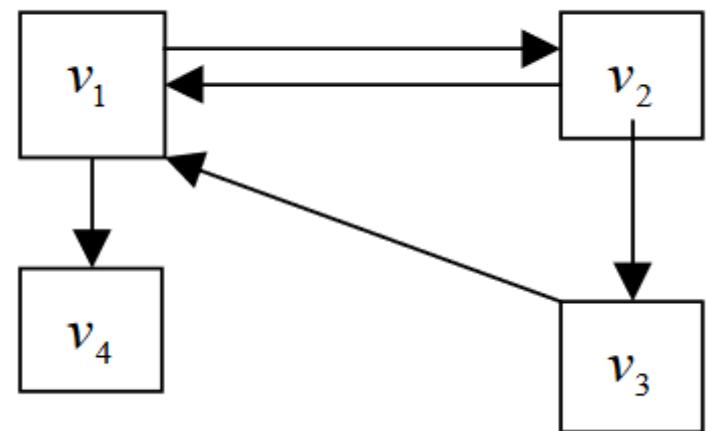
# Exercises

Apply the Warshall algorithm to find the transitive Closure matrix T for the following digraph's:

1)



2)



# All Pair Shortest Path Problem

---

- Let  $G = (V,E)$  be a digraph with  $n$  vertices. Let “C” be the cost adjacency matrix for  $G$ , such that  $C[i,i]$  i.e. cost of self loops is 0. The  $C[i,j]$  is the cost of the edge  $(i,j) \in E$ .
- The  $C[i,j] = \infty$  if there does not exist a path between vertex  $i$  and vertex  $j$ .
- All pairs shortest path problem is to determine a matrix  $A$  such that  $A[i,j]$  is the length of a shortest path from  $i$  to  $j$ .
- In other words, all pair shortest path problem is one where we want to find the shortest path from every node to every other node.
- This can be obtained greedily also! If we run the Dijkstra's algorithm by selecting every vertex as source then we can achieve this task with a complexity of  $O(n^2) * n = O(n^3)$

# All Pair Shortest Path Problem

- An alternate solution to the same problem can be obtained by dynamic programming strategy and the algorithm is called Floyd's algorithm or Floyd – Warshall's algorithm.
- Floyd's algorithm works on the same principles and is also referred at times as Floyd – Warshall's algorithm.

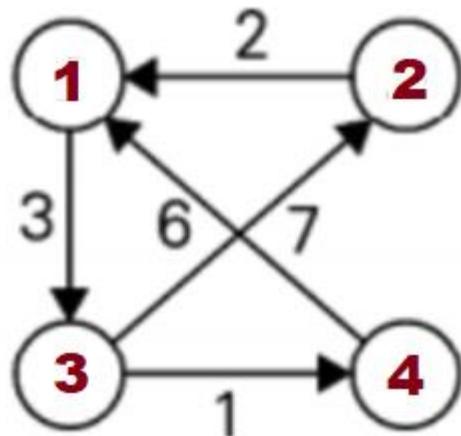
# Floyd's Algorithm: All Pairs Shortest Paths



- Problem: In a weighted (di)graph, find the shortest paths between every pair of vertices.
- *Same idea:* construct solution through a series of matrices  $D^{(0)}$ ,  $D^{(1)}$ , ...  $D^{(n)}$  using increasing subsets of the vertices allowed as intermediate. [ Note: Few books might name these matrices as A or Cost instead of D ].
- On the  $k^{\text{th}}$  iteration, the algorithm determines shortest paths between every pair of vertices  $i,j$  that use only vertices among  $1 \dots k$  as intermediate.

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

# Floyd's Algorithm Example



| $D^0$ | 1        | 2        | 3        | 4        |
|-------|----------|----------|----------|----------|
| 1     | 0        | $\infty$ | 3        | $\infty$ |
| 2     | 2        | 0        | $\infty$ | $\infty$ |
| 3     | $\infty$ | 7        | 0        | 1        |
| 4     | 6        | $\infty$ | $\infty$ | 0        |

$D^0$  is the cost -adjacency matrix of the digraph.

# Floyd's Example – Matrix Generation D<sup>(1)</sup>

Now, find D<sup>1</sup> Matrix !

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

$$k=1, D^1[i,j] = \min \{ D^{(0)}[i,j], D^{(0)}[i,1] + D^{(0)}[1,j] \}$$

| D <sup>0</sup> | 1        | 2        | 3        | 4        |
|----------------|----------|----------|----------|----------|
| 1              | 0        | $\infty$ | 3        | $\infty$ |
| 2              | 2        | 0        | $\infty$ | $\infty$ |
| 3              | $\infty$ | 7        | 0        | 1        |
| 4              | 6        | $\infty$ | $\infty$ | 0        |

|   | 1        | 2        | 3 | 4        |
|---|----------|----------|---|----------|
| 1 | 0        | $\infty$ | 3 | $\infty$ |
| 2 | 2        | 0        |   |          |
| 3 | $\infty$ |          | 0 |          |
| 4 | 6        |          |   | 0        |

- ✓ D<sup>1</sup>[2, 3] = min { D<sup>(0)</sup>[2,3] , D<sup>(0)</sup>[2,1] + D<sup>(0)</sup>[1,3] } = min {  $\infty$  , 2 + 3 } = 5
- ✓ D<sup>1</sup>[2, 4] = min { D<sup>(0)</sup>[2,4] , D<sup>(0)</sup>[2,1] + D<sup>(0)</sup>[1,4] } = min {  $\infty$  , 2 +  $\infty$  } =  $\infty$
- ✓ D<sup>1</sup>[3, 2] = min { D<sup>(0)</sup>[3,2] , D<sup>(0)</sup>[3,1] + D<sup>(0)</sup>[1,2] } = min { 7 ,  $\infty$  +  $\infty$  } = 7
- ✓ D<sup>1</sup>[3, 4] = min { D<sup>(0)</sup>[3,4] , D<sup>(0)</sup>[3,1] + D<sup>(0)</sup>[1,4] } = min { 1 ,  $\infty$  +  $\infty$  } = 1
- ✓ D<sup>1</sup>[4, 2] = min { D<sup>(0)</sup>[4,2] , D<sup>(0)</sup>[4,1] + D<sup>(0)</sup>[1,2] } = min {  $\infty$  , 6 +  $\infty$  } =  $\infty$
- ✓ D<sup>1</sup>[4, 3] = min { D<sup>(0)</sup>[4,3] , D<sup>(0)</sup>[4,1] + D<sup>(0)</sup>[1,3] } = min {  $\infty$  , 6 + 3 } = 9

# Floyd's Example – Matrix Generation D<sup>(2)</sup>

Now, find D<sup>2</sup> Matrix !

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

$$k=2, D^2[i,j] = \min \{ D^{(1)}[i,j], D^{(1)}[i,2] + D^{(1)}[2,j] \}$$

| D <sup>1</sup> | 1        | 2        | 3 | 4        |
|----------------|----------|----------|---|----------|
| 1              | 0        | $\infty$ | 3 | $\infty$ |
| 2              | 2        | 0        | 5 | $\infty$ |
| 3              | $\infty$ | 7        | 0 | 1        |
| 4              | 6        | $\infty$ | 9 | 0        |

| D <sup>2</sup> | 1 | 2        | 3 | 4        |
|----------------|---|----------|---|----------|
| 1              | 0 | $\infty$ |   |          |
| 2              | 2 | 0        | 5 | $\infty$ |
| 3              |   | 7        | 0 |          |
| 4              |   | $\infty$ |   | 0        |

- ✓  $D^2[1, 3] = \min \{ D^{(1)}[1,3], D^{(1)}[1,2] + D^{(1)}[2,3] \} = \min \{ 3, \infty + 5 \} = 3$
- ✓  $D^2[1, 4] = \min \{ D^{(1)}[1,4], D^{(1)}[1,2] + D^{(1)}[2,4] \} = \min \{ \infty, \infty + \infty \} = \infty$
- ✓  $D^2[3, 1] = \min \{ D^{(1)}[3,1], D^{(1)}[3,2] + D^{(1)}[2,1] \} = \min \{ \infty, 7 + 2 \} = 9$
- ✓  $D^2[3, 4] = \min \{ D^{(1)}[3,4], D^{(1)}[3,2] + D^{(1)}[2,4] \} = \min \{ 1, 7 + \infty \} = 1$
- ✓  $D^2[4, 1] = \min \{ D^{(1)}[4,1], D^{(1)}[4,2] + D^{(1)}[2,1] \} = \min \{ 6, \infty + 2 \} = 6$
- ✓  $D^2[4, 3] = \min \{ D^{(1)}[4,3], D^{(1)}[4,2] + D^{(1)}[2,3] \} = \min \{ 9, \infty + 5 \} = 9$

# Floyd's Example – Matrix Generation $D^{(3)}$

Now, find  $D^3$  Matrix !

$$k=3, D^3[i,j] = \min\{ D^{(2)}[i,j], D^{(2)}[i,3] + D^{(2)}[3,j] \}$$

| $D^2$ | 1 | 2        | 3 | 4        |
|-------|---|----------|---|----------|
| 1     | 0 | $\infty$ | 3 | $\infty$ |
| 2     | 2 | 0        | 5 | $\infty$ |
| 3     | 9 | 7        | 0 | 1        |
| 4     | 6 | $\infty$ | 9 | 0        |

$$D^{(3)} =$$

| $D^3$ | 1 | 2 | 3 | 4 |
|-------|---|---|---|---|
| 1     | 0 |   | 3 |   |
| 2     |   | 0 | 5 |   |
| 3     | 9 | 7 | 0 | 1 |
| 4     |   |   | 9 | 0 |

- ✓  $D^3[1, 2] = \min\{ D^{(2)}[1,2], D^{(2)}[1,3] + D^{(2)}[3,2] \} = \min\{ \infty, 3 + 7 \} = 10$
- ✓  $D^3[1, 4] = \min\{ D^{(2)}[1,4], D^{(2)}[1,3] + D^{(2)}[3,4] \} = \min\{ \infty, 3 + 1 \} = 4$
- ✓  $D^3[2, 1] = \min\{ D^{(2)}[2,1], D^{(2)}[2,3] + D^{(2)}[3,1] \} = \min\{ 2, 5 + 9 \} = 2$
- ✓  $D^3[2, 4] = \min\{ D^{(2)}[2,4], D^{(2)}[2,3] + D^{(2)}[3,4] \} = \min\{ \infty, 5 + 1 \} = 6$
- ✓  $D^3[4, 1] = \min\{ D^{(2)}[4,1], D^{(2)}[4,3] + D^{(2)}[3,1] \} = \min\{ 6, 9 + 9 \} = 6$
- ✓  $D^3[4, 2] = \min\{ D^{(2)}[4,2], D^{(2)}[4,3] + D^{(2)}[3,2] \} = \min\{ \infty, 9 + 7 \} = 16$

# Floyd's Example – Matrix Generation D<sup>(4)</sup>

Now, find D<sup>4</sup> Matrix !

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

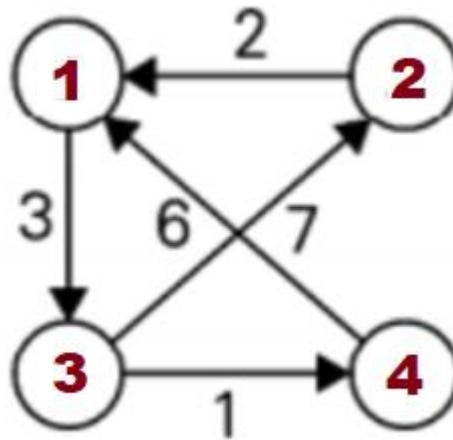
$$k=4, D^4[i,j] = \min \{ D^{(3)}[i,j], D^{(3)}[i,4] + D^{(3)}[4,j] \}$$

| D <sup>3</sup> | 1 | 2  | 3 | 4 |
|----------------|---|----|---|---|
| 1              | 0 | 10 | 3 | 4 |
| 2              | 2 | 0  | 5 | 6 |
| 3              | 9 | 7  | 0 | 1 |
| 4              | 6 | 16 | 9 | 0 |

| D <sup>4</sup> | 1 | 2  | 3 | 4 |
|----------------|---|----|---|---|
| 1              | 0 |    |   | 4 |
| 2              |   | 0  |   | 6 |
| 3              |   |    | 0 | 1 |
| 4              | 6 | 16 | 9 | 0 |

- ✓  $D^4[1, 2] = \min \{ D^{(3)}[1,2], D^{(3)}[1,4] + D^{(3)}[4,2] \} = \min \{ 10, 4 + 16 \} = 10$
- ✓  $D^4[1, 3] = \min \{ D^{(3)}[1,3], D^{(3)}[1,4] + D^{(3)}[4,3] \} = \min \{ 3, 4 + 9 \} = 3$
- ✓  $D^4[2, 1] = \min \{ D^{(3)}[2,1], D^{(3)}[2,4] + D^{(3)}[4,1] \} = \min \{ 2, 6 + 6 \} = 2$
- ✓  $D^4[2, 3] = \min \{ D^{(3)}[2,3], D^{(3)}[2,4] + D^{(3)}[4,3] \} = \min \{ 5, 6 + 9 \} = 5$
- ✓  $D^4[3, 1] = \min \{ D^{(3)}[3,1], D^{(3)}[3,4] + D^{(3)}[4,1] \} = \min \{ 9, 1 + 6 \} = 7$
- ✓  $D^4[3, 2] = \min \{ D^{(3)}[3,2], D^{(3)}[3,4] + D^{(3)}[4,2] \} = \min \{ 7, 1 + 16 \} = 7$

# Floyd's Example



| $D^4$ | 1 | 2  | 3 | 4 |
|-------|---|----|---|---|
| 1     | 0 | 10 | 3 | 4 |
| 2     | 2 | 0  | 5 | 6 |
| 3     | 7 | 7  | 0 | 1 |
| 4     | 6 | 16 | 9 | 0 |

$D^4$  matrix presents the shortest path cost between every pair

# Floyd's Algorithm

**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 ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Time complexity:  $O(n^3)$

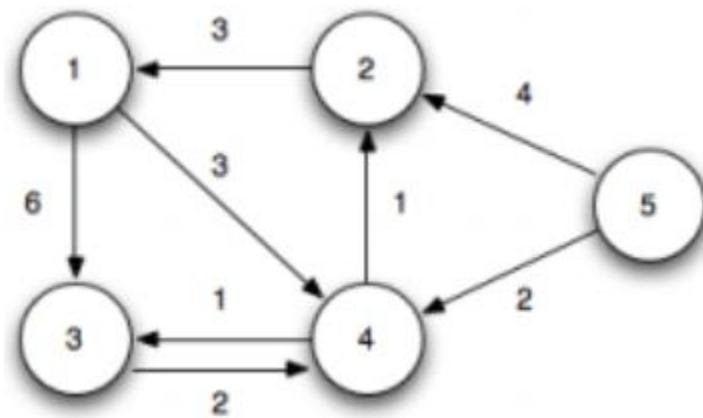
# Floyd's vs Dijkstra's Algorithm

---

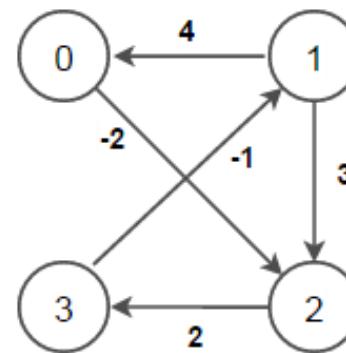
- Floyd's algorithm is used for all pair shortest path problem.
  - This is a DP strategy.
- Dijkstra's algorithm is used for single source shortest path problem.
  - This is a greedy strategy.
- The Dijkstra's algorithm can also be used for all pair shortest path problem if we run the algorithm on every vertex. Doing so we get a complexity of  $O(n^3)$ .
- Floyd's algorithm also produces the same complexity  $O(n^3)$ . Then why do we prefer Floyd's algorithm ?
- We prefer Floyd's algorithm over Dijkstra's for APSP problem because :
  - ✓ Floyd's works with negative weights as well. (But not negative weight cycles) but Dijkstra's may not work if weights are negative.
  - ✓ Floyd's algorithm can be implemented in a distributed system unlike Dijkstra's.

# Exercises

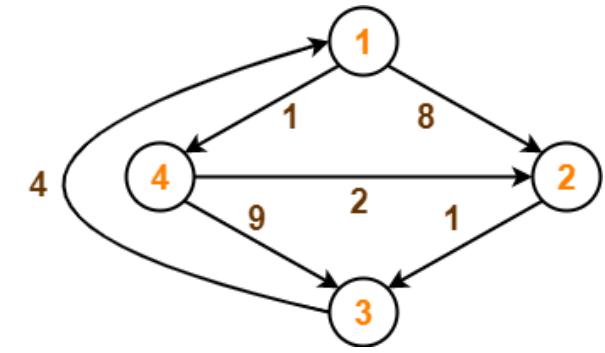
(1) Apply Floyd's algorithm to find the all pair shortest path:



(a)



(b)

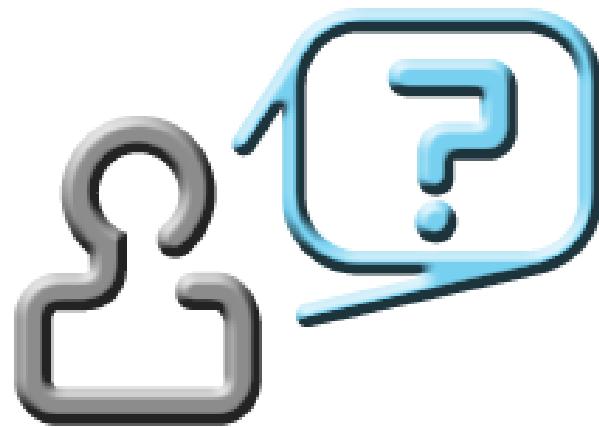


(c)

(2) Learn if there other algorithms for all pair shortest path problem.

Google about: Johnson's algorithm!

(3) Learn how to use DP strategy for Travelling Salesman Problem



*We will explore further in the next class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

**DSECLZG519**

**Parthasarathy**



# Contact Session #14

# DSECLZG519 –Dynamic Programming

# Agenda for Session # 14

---

- Recap of CS#13
- Matrix chain multiplication using DP
- 0/1 Knapsack
- Bellman Ford
- Game!
- Exercises

# Floyd's vs Dijkstra's Algorithm

---

- Floyd's algorithm is used for all pair shortest path problem.
  - This is a DP strategy.
- Dijkstra's algorithm is used for single source shortest path problem.
  - This is a greedy strategy.
- The Dijkstra's algorithm can also be used for all pair shortest path problem if we run the algorithm on every vertex. Doing so we get a complexity of  $O(n^3)$ .
- Floyd's algorithm also produces the same complexity  $O(n^3)$ . Then why do we prefer Floyd's algorithm ?
- We prefer Floyd's algorithm over Dijkstra's for APSP problem because :
  - ✓ Floyd's works with negative weights as well. (But not negative weight cycles) but Dijkstra's may not work if weights are negative.
  - ✓ Floyd's algorithm can be implemented in a distributed system unlike Dijkstra's.

# Matrix Multiplication

- Let A be an  $p \times q$  matrix, and B be a  $q \times r$  matrix. Their product is defined to be the  $p \times r$  matrix C where

$$C(i,j) = \sum_{k=1}^q A(i,k) \cdot B(k,j)$$

$$\begin{array}{l} \text{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \quad \text{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \\ \text{2 X 3} \qquad \qquad \qquad \text{3 X 2} \\ \boxed{\text{Must be same!}} \end{array}$$

$$\begin{array}{l} \text{C} = \text{A} \times \text{B} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{bmatrix} \\ \text{2 X 2} \end{array}$$

- Number of Multiplications =  $12 ( p \times q \times r )$

# Matrix Chain Product / Matrix Chain Multiplication

Matrices :  $[A] \times [B] \times [C]$

Dimensions:  $2 \times 3$      $3 \times 4$      $4 \times 2$



Is Matrix Multiplication Associative ?

**Yes!**

So, we can perform either

$(A \times B) \times C$  or  $A \times (B \times C)$

*Number of individual multiplications needed ?*

# Matrix Chain Product / Matrix Chain Multiplication

Matrices :  $[A] \times [B] \times [C]$

Dimensions:  $2 \times 3 \quad 3 \times 4 \quad 4 \times 2$



$(A \times B) \times C \text{ or } A \times (B \times C)$

Number of Multiplications = ?  
 $(A \times B) = 2 * 3 * 4 = 24$  & resultant  
 Dimension will be  $2 \times 4$   
 $(A \times B) \times C = 2 * 4 * 2 = 16$   
 So, a total of  $24+16 = 40$  multiplications

Number of Multiplications = ?  
 $(B \times C) = 3 * 4 * 2 = 24$  & resultant  
 Dimension will be  $3 \times 2$   
 $A \times (B \times C) = 2 * 3 * 2 = 12$   
 So, a total of  $24+12 = 36$  multiplications

*Number of multiplications : 40 and 36*

# Matrix Chain Multiplication

- Now, let us consider we need to evaluate a long product of matrices :

$$A_1 \cdot A_2 \cdot A_3 \dots A_{n-1} \cdot A_n$$

- We can place the parentheses wherever we like ☺ [Since, matrix multiplication is associative].
- But, our motive is to place the parentheses in such a way that the total number of individual/scalar multiplications needed is minimum.
- *So, the matrix chain multiplication problem is to determine the parenthesization of the expression defining the product that minimizes the total number of scalar multiplications performed.*
- Fact Check : Can we rearrange the terms ?
  - No, matrix multiplication is not commutative
- How many ways are there to fully parenthesize this product ?

# Matrix Chain Multiplication

- How many ways are there to fully parenthesize an expression ?
- Equivalently: *How many full/proper binary trees on n leaves are there?*

Let us say, we have to multiply 4 matrices :

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4$$

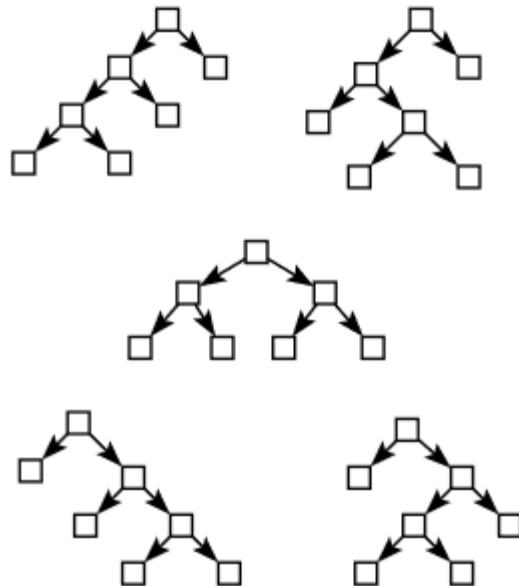
- ✓  $(A_1 \cdot A_2) (A_3 \cdot A_4)$
- ✓  $A_1 (A_2 (A_3 \cdot A_4))$
- ✓  $A_1 ((A_2 \cdot A_3) A_4)$
- ✓  $((A_1 \cdot A_2) A_3) A_4$
- ✓  $(A_1 (A_2 \cdot A_3)) A_4$



*When n=4, we can have 5 different ways to parenthesize it!*

# Matrix Chain Multiplication

*Even with tree approach, we get 5 when n=4*



- In Simple, if we want to know the number of different ways the parenthesization can be done is the modified *Catalan* number formula which is :

$$\frac{2(n - 1)C_{(n - 1)}}{n}$$

So, When n=4,

$$\frac{^6C_3}{4} = \frac{20}{4} = 5$$

So, When n=5,

$$\frac{8C_4}{5} = \frac{70}{5} = 14$$

# Matrix Chain Multiplication

Suppose we have matrices  $A_1, A_2, \dots, A_{n-1}, A_n$  of size  $p_0, p_1, \dots, p_n$ , so that  $A_i$  has dimension  $p_{i-1} \times p_i$ .

Choosing a particular parenthesization affects the total number of multiplication.

Example:

$$A_1 : 10 \times 100$$

$$A_2 : 100 \times 5$$

$$A_3 : 5 \times 50$$

$$(A_1 A_2) A_3 : 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$$

$$A_1 (A_2 A_3) : 100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$$

This is vaguely reminiscent of Huffman coding, or merging a collection of already sorted lists.

But there are important differences: we must keep the matrices in order. We can choose the shape of the tree, but not the leaf-order.

Also, the cost function is more complicated:

If  $A$  is  $p \times q$  matrix,  $B$   $q \times r$  then the cost of multiplying them is  $p \cdot q \cdot r$  and the result has dimension  $p \times r$ .

As a consequence, the simple Greedy Algorithm in Huffman coding won't work here.

# Matrix Chain Multiplication

- The brute force way is very costly and runs in exponential time.  
We can improve the performance by making few observations about the nature of the matrix chain-product problem.
- The first observation is that the problem can be split into subproblems which are related.
- Hence, we can use the Dynamic Programming Strategy here!
- Let us derive the formula and algorithm on how to apply the DP strategy to the matrix chain multiplication problem.
- Let us consider the previous example and derive the formula ...

# Matrix Chain Multiplication

Matrices :  $[A] \times [B] \times [C]$

Dimensions:  $2 \times 3$      $3 \times 4$      $4 \times 2$



$(A \times B) \times C$  or  $A \times (B \times C)$



Number of Multiplications = ?

$(A \times B) = 2 * 3 * 4 = 24$  & resultant  
Dimension will be  $2 \times 4$

$(A \times B) \times C = 2 * 4 * 2 = 16$

So, a total of  $24+16 = 40$  multiplications

Number of Multiplications = ?

$(B \times C) = 3 * 4 * 2 = 24$  & resultant  
Dimension will be  $3 \times 2$

$A \times (B \times C) = 2 * 3 * 2 = 12$

So, a total of  $24+12 = 36$  multiplications

*Number of multiplications : 40 and 36*

# Matrix Chain Multiplication

$\text{Cost}[1,3] = ?$   
 $\text{C}[1,3] = ?$

Matrices :  $[A] \times [B] \times [C]$   
 Dimensions:  $2 \times 3 \quad 3 \times 4 \quad 4 \times 2$   
 $d_0 \ d_1 \quad d_1 \ d_2 \quad d_2 \ d_3$

$(A \times B) \times C$

$A \times (B \times C)$

- $(A \times B) = 2 * 3 * 4 = 24$ , which is  $C[1,2]$
- $C = 0$ , as its single matrix (no multiplication involved)  $\rightarrow C[3,3]$
- Resultant of  $A \times B$  is dimension  $2 \times 4$
- Now,  $(A \times B) \times C$  :
- Dimensions :  $2 \times 4 \quad 4 \times 2$   
 $d_0 \ d_2 \quad d_2 \ d_3$
- $(A \times B) \times C = 2 * 4 * 2 = 16$   
 $\text{C}[1,2] + \text{C}[3,3] + d_0 \times d_2 \times d_3 = 40$

- $A = 0$ , as its single matrix (no multiplication involved)  $\rightarrow C[1,1]$
- $(B \times C) = 3 * 4 * 2 = 24$ , which is  $C[2,3]$
- Resultant of  $B \times C$  is dimension  $3 \times 2$
- Now,  $A \times (B \times C)$  :
- Dimensions :  $2 \times 3 \quad 3 \times 2$   
 $d_0 \ d_1 \quad d_1 \ d_3$
- $A \times (B \times C) := 2 * 3 * 2 = 12$   
 $\text{C}[1,1] + \text{C}[2,3] + d_0 \times d_1 \times d_3 = 36$

# Matrix Chain Multiplication

$\text{Cost}[1,3] = ?$

$C[i,j] = ?$

Matrices :  $[A] \times [B] \times [C]$

Dimensions:  $2 \times 3 \quad 3 \times 4 \quad 4 \times 2$   
 $d_0 \ d_1 \quad d_1 \ d_2 \quad d_2 \ d_3$

$(A \times B) \times C$

$A \times (B \times C)$

$$C[1,2] + C[3,3] + d_0 \times d_2 \times d_3 = 40$$

i                j

$$C[1,1] + C[2,3] + d_0 \times d_1 \times d_3 = 36$$

i                j

$$C[1,2] + C[3,3] + d_0 \times d_2 \times d_3 = 40$$

i                j             $d_{i-1}$              $d_j$

$$C[1,1] + C[2,3] + d_0 \times d_1 \times d_3 = 36$$

i                j             $d_{i-1}$              $d_j$

$$C[1,2] + C[3,3] + d_0 \times d_2 \times d_3 = 40$$

i    k             $k+1$     j             $d_{i-1}$      $d_k$      $d_j$

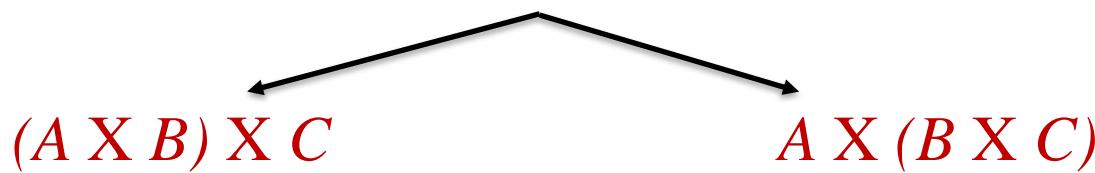
$$C[1,1] + C[2,3] + d_0 \times d_1 \times d_3 = 36$$

i    k             $k+1$     j             $d_{i-1}$      $d_k$      $d_j$

# Matrix Chain Multiplication

$\text{Cost}[1,3] = ?$   
 $C[i,j] = ?$

Matrices :  $[A] \times [B] \times [C]$   
 Dimensions:  $2 \times 3 \quad 3 \times 4 \quad 4 \times 2$   
 $d_0 \ d_1 \quad d_1 \ d_2 \quad d_2 \ d_3$



$C[1,2] + C[3,3] + d_0 \times d_2 \times d_3 = 40$

↳ i k k+1 j  $d_{i-1}$   $d_k$   $d_j$

$C[1,1] + C[2,3] + d_0 \times d_1 \times d_3 = 36$

↳ i k k+1 j  $d_{i-1}$   $d_k$   $d_j$

$$C[i,j] = \min_{i \leq k < j} \{C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j\}$$

*This is the DP formula for Matrix Chain Multiplication ☺*

# Matrix Chain Multiplication

- Let us consider the same example where we want to multiply 4 matrices.

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4$$

- Let us use the formula that we derived.

$$C[i,j] = \min_{i \leq k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} X d_k X d_j \}$$

$$C[1,4] = \min_{1 \leq k < 4} \begin{cases} k=1 & C[1,1] + C[2,4] + d_0 X d_1 X d_4, \\ k=2 & C[1,2] + C[3,4] + d_0 X d_2 X d_4, \\ k=3 & C[1,3] + C[4,4] + d_0 X d_3 X d_4 \end{cases}$$

$A_1 (A_2 A_3 A_4)$

$(A_1 A_2) (A_3 A_4)$

$(A_1 A_2 A_3) A_4$

# Matrix Chain Multiplication

$$C[i,j] = \min_{i \leq k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

$$C[1,4] = \min_{1 \leq k < 4} \begin{cases} k=1 & C[1,1] + C[2,4] + d_0 \times d_1 \times d_4, \\ k=2 & C[1,2] + C[3,4] + d_0 \times d_2 \times d_4, \\ k=3 & C[1,3] + C[4,4] + d_0 \times d_3 \times d_4 \end{cases}$$

Here,  $C[1,1]$  and  $C[4,4]$  are zero. Others are unknown! We need to apply the formula again.

$$C[2,4] = \min_{2 \leq k < 4} \begin{cases} k=2 & C[2,2] + C[3,4] + d_1 \times d_2 \times d_4, \\ k=3 & C[2,3] + C[4,4] + d_1 \times d_3 \times d_4, \end{cases}$$

# Matrix Chain Multiplication

$$C[i,j] = \min_{i \leq k < j} \{C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j\}$$

$$C[2,4] = \min_{2 \leq k < 4} \begin{cases} k=2 \\ k=3 \end{cases} \left\{ \begin{array}{l} C[2,2] + C[3,4] + d_1 \times d_2 \times d_4, \\ C[2,3] + C[4,4] + d_1 \times d_3 \times d_4, \end{array} \right.$$

- Here again,  $C[2,2]$  and  $C[4,4]$  are zero.  $C[3,4]$  is unknown !

# Matrix Chain Multiplication

- But, this is a lengthy process 😞
- DP Strategy states us to start with smaller sub-problems and then go stage by stage. In this case, instead of calculating  $C[1,4]$  in the beginning, we shall calculate smaller subproblems such as  $C[3,4]$  ,  $C[1,2]$  etc.

$$C[1,4] = \min_{1 \leq k < 4} \begin{cases} k=1 & C[1,1] + C[2,4] + d_0 \times d_1 \times d_4, \\ k=2 & C[1,2] + C[3,4] + d_0 \times d_2 \times d_4, \\ k=3 & C[1,3] + C[4,4] + d_0 \times d_3 \times d_4 \end{cases}$$

- In this approach, one can also reuse the solutions of smaller subproblems later when needed.
- A table is preferred to solve such dynamic strategy problems! So, that it's easier to reuse the solutions.

# Matrix Chain Multiplication

- For the matrix chain multiplication problem, we will use two tables as below:

|      |   | j |   |   |   |
|------|---|---|---|---|---|
|      |   | 1 | 2 | 3 | 4 |
| Cost | 1 | 0 |   |   |   |
|      | 2 | - | 0 |   |   |
| 3    | - | - | 0 |   |   |
| 4    | - | - | - | 0 |   |

This table would be used to denote which yielded the minimum solution.

We will start filling the table from smaller subproblem to bigger. Ex: (1,2) (2,3) (3,4) then (1,3) (2,4) and finally (1,4)

|   |   | k |   |   |   |
|---|---|---|---|---|---|
|   |   | 1 | 2 | 3 | 4 |
| 1 | 1 | 0 |   |   |   |
|   | 2 | - | 0 |   |   |
| 3 | - | - | 0 |   |   |
| 4 | - | - | - | 0 |   |

# Matrix Chain Multiplication Example



- We are given the matrices orders in a sequence { 3,2,4,2,5 }.
- This means that the matrices have sizes
  - 3X2, 2X4, 4X2, 2X5
- Are these dimensions valid and matrix multiplication can be performed on this ?
  - Yes!
- Aim: Find the minimum number of scalar multiplications needed and the parenthesization.
- We know that cost of C[1,1] , C[2,2], C[3,3] and C[4,4] is 0. With this in mind, lets starting using the formula and fill the Cost/Matrix table.
- Also, as per DP, we will start with smaller subproblems (where difference is small) like C[1,2] , C[2,3] and then proceed to C[1,3] etc.

# Matrix Chain Multiplication Example

$$C[i,j] = \min_{i \leq k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} X d_k X d_j \}$$

$k=1$

$$C[1,2] = \min_{1 \leq k < 2} \{ C[1,1] + C[2,2] + d_0 X d_1 X d_2 \} \\ = \min \{ 0 + 0 + 3 * 2 * 4 \} = 24$$

$k=2$

$$C[2,3] = \min_{2 \leq k < 3} \{ C[2,2] + C[3,3] + d_1 X d_2 X d_3 \} \\ = \min \{ 0 + 0 + 2 * 4 * 2 \} = 16$$

$k=3$

$$C[3,4] = \min_{3 \leq k < 4} \{ C[3,3] + C[4,4] + d_2 X d_3 X d_4 \} \\ = \min \{ 0 + 0 + 4 * 2 * 5 \} = 40$$

- So, we have solved three sub-problems and stored their solutions. Now we will move to slightly bigger sub-problem. *Can you guess which is that?*

| A                   | X                   | B                   | X                   | C                   | X                   | D                   |
|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| 3<br>d <sub>0</sub> | 2<br>d <sub>1</sub> | 2<br>d <sub>1</sub> | 4<br>d <sub>2</sub> | 4<br>d <sub>2</sub> | 2<br>d <sub>3</sub> | 2<br>d <sub>3</sub> |
|                     |                     |                     |                     |                     |                     | 5<br>d <sub>4</sub> |

Cost →

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| i | 1 | 2 | j | 3 | 4 |
|   | 0 |   |   |   |   |
|   | - | 0 |   |   |   |
|   | - | - | 0 |   |   |
|   | - | - | - | 0 |   |

k    1    2    3    4

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 |   |   |   |
| 2 | - | 0 |   |   |
| 3 | - | - | 0 |   |
| 4 | - | - | - | 0 |

# Matrix Chain Multiplication Example

$$C[i,j] = \min_{1 \leq k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} X d_k X d_j \}$$

$k=1,2$

$$\begin{aligned} C[1,3] &= \min_{1 \leq k < 3} \{ C[1,1] + C[2,3] + d_0 X d_1 X d_3, \\ &\quad C[1,2] + C[3,3] + d_0 X d_2 X d_3 \} \\ &= \min \{ 0 + 16 + 3 * 2 * 2, \\ &\quad 24 + 0 + 3 * 4 * 2 \} = \min\{28,48\} \\ &= 28 \end{aligned}$$

$k=2,3$

$$\begin{aligned} C[2,4] &= \min_{2 \leq k < 4} \{ C[2,2] + C[3,4] + d_1 X d_2 X d_4, \\ &\quad C[2,3] + C[4,4] + d_1 X d_3 X d_4 \} \\ &= \min \{ 0 + 40 + 2 * 4 * 5, \\ &\quad 16 + 0 + 2 * 2 * 5 \} = \min\{80,36\} \\ &= 36 \end{aligned}$$

What next ?

| A                   | X                   | B                   | X                   | C                   | X                   | D                   |
|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| 3<br>d <sub>0</sub> | 2<br>d <sub>1</sub> | 2<br>d <sub>1</sub> | 4<br>d <sub>2</sub> | 4<br>d <sub>2</sub> | 2<br>d <sub>3</sub> | 2<br>d <sub>3</sub> |
|                     |                     |                     |                     |                     |                     | 5<br>d <sub>4</sub> |

Cost

|   |   | j |   |    |   |
|---|---|---|---|----|---|
|   |   | 1 | 2 | 3  | 4 |
|   |   | 1 | 0 | 24 |   |
| i | 2 | - | 0 | 16 |   |
| 3 | - | - | 0 | 40 |   |
| 4 | - | - | - | 0  |   |

|   |   | k |   |   |   |
|---|---|---|---|---|---|
|   |   | 1 | 2 | 3 | 4 |
|   |   | 1 | 0 | 1 |   |
| 2 | - | 0 | 2 |   |   |
| 3 | - | - | 0 | 3 |   |
| 4 | - | - | - | 0 |   |

# Matrix Chain Multiplication Example

$$C[i,j] = \min_{i \leq k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} X d_k X d_j \}$$

| A                   | X                   | B                   | X                   | C                   | X                   | D                   |
|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| 3<br>d <sub>0</sub> | 2<br>d <sub>1</sub> | 2<br>d <sub>1</sub> | 4<br>d <sub>2</sub> | 4<br>d <sub>2</sub> | 2<br>d <sub>3</sub> | 2<br>d <sub>3</sub> |
|                     |                     |                     |                     |                     |                     | 5<br>d <sub>4</sub> |

k= 1,2,3

$$\begin{aligned}
 C[1,4] &= \min_{1 \leq k < 4} \{ C[1,1] + C[2,4] + d_0 X d_1 X d_4 , \\
 &\quad C[1,2] + C[3,4] + d_0 X d_2 X d_4 , \\
 &\quad C[1,3] + C[4,4] + d_0 X d_3 X d_4 \} \\
 &= \min \{ 0 + 36 + 3 * 2 * 5 , \\
 &\quad 24 + 40 + 3 * 4 * 5 , \\
 &\quad 28 + 00 + 3 * 2 * 5 \} \\
 &= \min \{ 66, 124, 58 \} = 58
 \end{aligned}$$

Cost

|   | 1 | 2         | j         | 3         | 4 |
|---|---|-----------|-----------|-----------|---|
| i | 0 | <b>24</b> | <b>28</b> |           |   |
| 1 | - | 0         | <b>16</b> | <b>36</b> |   |
| 2 | - | -         | 0         | <b>40</b> |   |
| 3 | - | -         | -         | 0         |   |
| 4 | - | -         | -         | 0         |   |

| k | 1 | 2        | 3        | 4        |
|---|---|----------|----------|----------|
| 1 | 0 | <b>1</b> | <b>1</b> |          |
| 2 | - | 0        | <b>2</b> | <b>3</b> |
| 3 | - | -        | 0        | <b>3</b> |
| 4 | - | -        | -        | 0        |

# Matrix Chain Multiplication Example

$$C[i,j] = \min_{i \leq k < j} \{C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j\}$$

| A                   | X                   | B                   | X                   | C                   | X                   | D                   |
|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| 3<br>d <sub>0</sub> | 2<br>d <sub>1</sub> | 2<br>d <sub>1</sub> | 4<br>d <sub>2</sub> | 4<br>d <sub>2</sub> | 2<br>d <sub>3</sub> | 2<br>d <sub>4</sub> |

- From the solved table, we can see that  $C[1,4]$  is 58.
- Hence, the minimum number of scalar multiplications needed to multiply four matrices of the given dimension is **58**.
- Parenthesization:
  - The  $k[1,4] = 3$  & hence, parenthesize at 3 ☺
  - $(A \times B \times C) \times (D)$
  - How to further parenthesize  $A \times B \times C$  ?
  - $K[1,3]$  will give us where to parenthesize which is 1 here.
  - So, **( (A) X (B X C) ) X (D)**

→

|      |   | j |   |    |    |    |
|------|---|---|---|----|----|----|
|      |   | 1 | 2 | 3  | 4  |    |
| Cost | i | 1 | 0 | 24 | 28 | 58 |
|      |   | 2 | - | 0  | 16 | 36 |
|      | 3 | - | - | 0  | 40 |    |
|      | 4 | - | - | -  | 0  |    |

↓

|   |   | k |   |   |   |   |
|---|---|---|---|---|---|---|
|   |   | 1 | 2 | 3 | 4 |   |
| k | 1 | 1 | 0 | 1 | 1 | 3 |
|   |   | 2 | - | 0 | 2 | 3 |
| 2 | 3 | - | - | 0 | 3 |   |
| 3 | 4 | - | - | - | 0 |   |

# Matrix Chain Multiplication Algorithm



Bottom-up approach: compute an array  $C[i, j]$ .

For  $i = j$  can set  $C(i, i) = 0$ .

Then compute all  $C(i, j)$  for  $d = j - i$ ,  $d = 1, 2, \dots, n - 1$ .

```
for d = 1,...,n-1 do
  for i = 1,...,n-d do
    j = i + d;
    for k = i,...,j-1 do
    {
      c = C[i,k] + C[k+1,j] + p[i-1] p[k] p[j];
      if( c < C[i,j] )
        C[i,j] = c;
    }
}
```

Running time is clearly  $O(n^3)$  and in fact  $\Theta(n^3)$ .

# 0/1 Knapsack Problem

---

- Given a knapsack with maximum capacity  $W$ , and a set  $S$  consisting of  $n$  items
- Each item  $i$  has some weight  $w_i$  and benefit value  $b_i$  (**all  $w_i$  and  $W$  are integer values**)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?
- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a “0-1” problem, because each item must be entirely accepted or rejected.

# 0/1 Knapsack Problem : Brute force Approach

---

- Let's first solve this problem with a straightforward algorithm
- Since there are  $n$  items, there are  $2^n$  possible combinations of items.
- We go through all combinations and find the one with maximum value and with total weight less or equal to  $W$
- Running time will be  $\mathbf{O}(2^n)$

# 0/1 Knapsack Problem : Dynamic Programming Approach

If items are labeled 1..n, then a subproblem would be to find an optimal solution for  $S_k = \{ \text{items labeled } 1, 2, \dots k \}$

- Let's add another parameter: w, which will represent the maximum weight for each subset of items
- The subproblem then will be to compute  $V[k,w]$ , i.e., to find an optimal solution for  $S_k = \{ \text{items labeled } 1, 2, \dots k \}$  in a knapsack of size w
- Assuming knowing  $V[i, j]$ , where  $i=0,1, 2, \dots k-1$ ,  $j=0,1,2, \dots w$ , how to derive  $V[k,w]$ ?
- Recursive formula for subproblems:

$$V [k , w ] = \begin{cases} V [k - 1, w ] & \text{if } w_k > w \\ \max\{ V [k - 1, w ], V [k - 1, w - w_k ] + b_k \} & \text{else} \end{cases}$$

# 0/1 Knapsack : DP Approach Example



Let's run our algorithm on the following data:

- $n = 4$  (# of elements)
- $W = 5$  (max weight of the knapsack)
- Elements (weight, benefit):  
 $(2,3), (3,4), (4,5), (5,6)$

# 0/1 Knapsack : DP Approach Example



| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   |   |   |   |   |   |   |
| 2   |   |   |   |   |   |   |
| 3   |   |   |   |   |   |   |
| 4   |   |   |   |   |   |   |

for  $w = 0$  to  $W$

$$V[0,w] = 0$$

# 0/1 Knapsack : DP Approach Example



| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 |   |   |   |   |   |
| 2   | 0 |   |   |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

for  $i = 1$  to  $n$   
 $V[i,0] = 0$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 |   |   |   |   |
| 2   | 0 |   |   |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

$$i=1$$

$$b_i = 3$$

$$w_i = 2$$

$$w = 1$$

$$w - w_i = -1$$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example



| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 |   |   |   |
| 2   | 0 |   |   |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=1$   
 $b_i=3$   
 $w_i=2$   
 $w=2$   
 $w-w_i = 0$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 |   |   |
| 2   | 0 |   |   |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=1$   
 $b_i=3$   
 $w_i=2$   
 $w=3$   
 $w-w_i = 1$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$$V[i, w] = b_i + V[i-1, w - w_i]$$

else

$$V[i, w] = V[i-1, w]$$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 |   |
| 2   | 0 |   |   |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=1$   
 $b_i=3$   
 $w_i=2$   
 $w=4$   
 $w-w_i=2$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$$V[i, w] = b_i + V[i-1, w - w_i]$$

else

$$V[i, w] = V[i-1, w]$$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 |   |   |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=1$   
 $b_i=3$   
 $w_i=2$   
 $w=5$   
 $w-w_i = 3$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 |   |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=2$   
 $b_i=4$   
 $w_i=3$   
 $w=1$   
 $w-w_i = -2$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 |   |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=2$   
 $b_i=4$   
 $w_i=3$   
 $w=2$   
 $w-w_i = -1$

if  $w_i \leq w$  // item i can be part of the solution  
 if  $b_i + V[i-1, w-w_i] > V[i-1, w]$   
 $V[i, w] = b_i + V[i-1, w-w_i]$   
 else  
 $V[i, w] = V[i-1, w]$   
**else**  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 |   |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=2$   
 $b_i=4$   
 $w_i=3$   
 $w=3$   
 $w-w_i = 0$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 |   |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=2$   
 $b_i=4$   
 $w_i=3$   
 $w=4$   
 $w-w_i = 1$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$$V[i, w] = b_i + V[i-1, w-w_i]$$

else

$$V[i, w] = V[i-1, w]$$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 |   |   |   |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=2$   
 $b_i=4$   
 $w_i=3$   
 $w=5$   
 $w-w_i = 2$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$$V[i, w] = b_i + V[i-1, w-w_i]$$

else

$$V[i, w] = V[i-1, w]$$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 |   |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=3$   
 $b_i=5$   
 $w_i=4$   
 $w=1..3$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 |   |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=3$   
 $b_i=5$   
 $w_i=4$   
 $w=4$   
 $w - w_i = 0$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$$V[i, w] = b_i + V[i-1, w - w_i]$$

else

$$V[i, w] = V[i-1, w]$$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 |   |   |   |   |   |

Items:

|          |
|----------|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

$i=3$   
 $b_i=5$   
 $w_i=4$   
 $w= 5$   
 $w - w_i = 1$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 |   |

$i=4$

$b_i=6$

$w_i=5$

$w=1..4$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w - w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

Items:

|               |
|---------------|
| $i=4$         |
| $b_i=6$       |
| $w_i=5$       |
| $w= 5$        |
| $w - w_i = 0$ |

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + V[i-1, w-w_i] > V[i-1, w]$

$V[i, w] = b_i + V[i-1, w-w_i]$

else

$V[i, w] = V[i-1, w]$

else  $V[i, w] = V[i-1, w]$  //  $w_i > w$

# 0/1 Knapsack : DP Approach

$$V[k, w] = \begin{cases} V[k-1, w] & \text{if } w_k > w \\ \max\{V[k-1, w], V[k-1, w-w_k] + b_k\} & \text{else} \end{cases}$$

- The best subset of  $S_k$  that has the total weight  $\leq w$ , either contains item  $k$  or not.
- First case:  $w_k > w$ . Item  $k$  can't be part of the solution, since if it was, the total weight would be  $> w$ , which is unacceptable.
- Second case:  $w_k \leq w$ . Then the item  $k$  can be in the solution, and we choose *the case with greater value*.

# 0/1 Knapsack : DP Approach Algorithm



for  $w = 0$  to  $W$

$$V[0,w] = 0$$

for  $i = 1$  to  $n$

$$V[i,0] = 0$$

for  $i = 1$  to  $n$

$O(n*W)$

for  $w = 0$  to  $W$

if  $w_i \leq w$  // item  $i$  can be part of the solution

$$\text{if } b_i + V[i-1, w-w_i] > V[i-1, w]$$

$$V[i, w] = b_i + V[i-1, w-w_i]$$

else

$$V[i, w] = V[i-1, w]$$

$$\text{else } V[i, w] = V[i-1, w] \text{ // } w_i > w$$

# 0/1 Knapsack : DP Approach

---

- How to find out which items are in the optimal subset ?
- This algorithm only finds the max possible value that can be carried in the knapsack
  - i.e., the value in  $V[n, W]$
- To know the items that make this maximum value, an addition to this algorithm is necessary ☺

# 0/1 Knapsack : DP Approach

## How to find actual Knapsack Items

---

- All of the information we need is in the table.
- $V[n, W]$  is the maximal value of items that can be placed in the Knapsack.
- Let  $i=n$  and  $k=W$

if  $V[i, k] \neq V[i-1, k]$  then

    mark the  $i^{\text{th}}$  item as in the knapsack

$k = k - w_i$ ,  $i = i-1$

else

$i = i-1$  // Assume the  $i^{\text{th}}$  item is not in the knapsack

# 0/1 Knapsack : DP Approach

## Finding the Items - Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

i=4

k=5

b<sub>i</sub>=6

w<sub>i</sub>=5

V[i,k] = 7

V[i-1,k] = 7

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i=n, k=W

while i,k > 0

if V[i,k] ≠ V[i-1,k] then

mark the *i*<sup>th</sup> item as in the knapsack

k = k-w<sub>i</sub>, i = i-1

else

i = i-1

# 0/1 Knapsack : DP Approach

## Finding the Items - Example

Items:

|    |       |
|----|-------|
| 1: | (2,3) |
| 2: | (3,4) |
| 3: | (4,5) |
| 4: | (5,6) |

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

$$i=4$$

$$k=5$$

$$b_i=6$$

$$w_i=5$$

$$V[i,k] = 7$$

$$V[i-1,k] = 7$$

$i=n, k=W$

while  $i, k > 0$

if  $V[i,k] \neq V[i-1,k]$  then

mark the  $i^{\text{th}}$  item as in the knapsack

$$k = k - w_i, i = i - 1$$

else

$$i = i - 1$$

# 0/1 Knapsack : DP Approach

## Finding the Items - Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

i=3

k= 5

b<sub>i</sub>=5

w<sub>i</sub>=4

V[i,k] = 7

V[i-1,k] = 7

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

i=n, k=W

while i,k > 0

if V[i,k] ≠ V[i-1,k] then

mark the *i*<sup>th</sup> item as in the knapsack

$k = k - w_i$ ,  $i = i - 1$

else

$i = i - 1$

# 0/1 Knapsack : DP Approach

## Finding the Items - Example

Items:

|    |       |
|----|-------|
| 1: | (2,3) |
| 2: | (3,4) |
| 3: | (4,5) |
| 4: | (5,6) |

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

i=2

k= 5

b<sub>i</sub>=4

w<sub>i</sub>=3

V[i,k] = 7

V[i-1,k] = 3

k - w<sub>i</sub>=2

i=n, k=W

while i,k > 0

if V[i,k] ≠ V[i-1,k] then

mark the *i*<sup>th</sup> item as in the knapsack

k = k-w<sub>i</sub>, i = i-1

else

i = i-1

# 0/1 Knapsack : DP Approach

## Finding the Items - Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

$$i=1$$

$$k=2$$

$$b_i=3$$

$$w_i=2$$

$$V[i,k] = 3$$

$$V[i-1,k] = 0$$

$$k - w_i = 0$$

$i=n, k=W$

while  $i,k > 0$

if  $V[i,k] \neq V[i-1,k]$  then

mark the  $i^{\text{th}}$  item as in the knapsack

$k = k - w_i, i = i-1$

else

$i = i-1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

# 0/1 Knapsack : DP Approach

## Finding the Items - Example

Items:

|    |       |
|----|-------|
| 1: | (2,3) |
| 2: | (3,4) |
| 3: | (4,5) |
| 4: | (5,6) |

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

i=0

k= 0

i=n, k=W

while i,k > 0

if  $V[i,k] \neq V[i-1,k]$  then

mark the  $i^{\text{th}}$  item as in the knapsack

$k = k - w_i$ ,  $i = i-1$

else

$i = i-1$

The optimal knapsack  
should contain {1, 2}

# 0/1 Knapsack : DP Approach

## Finding the Items - Example

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|---|---|---|---|---|---|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 |
| 1   | 0 | 0 | 3 | 3 | 3 | 3 |
| 2   | 0 | 0 | 3 | 4 | 4 | 7 |
| 3   | 0 | 0 | 3 | 4 | 5 | 7 |
| 4   | 0 | 0 | 3 | 4 | 5 | 7 |

i=n, k=W

while i,k > 0

if  $V[i,k] \neq V[i-1,k]$  then

mark the  $i^{\text{th}}$  item as in the knapsack

$k = k - w_i$ ,  $i = i - 1$

else

$i = i - 1$

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

The optimal knapsack  
should contain {1, 2}

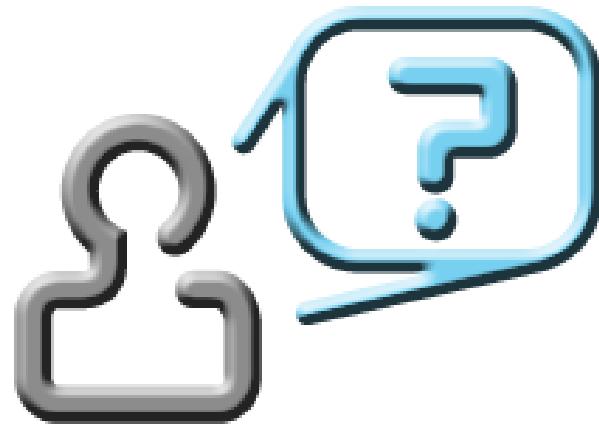
# 0/1 Knapsack Example 2

Apply the DP algorithm to the following 0/1 Knapsack instance :

Let  $W = 10$  and

| $i$   | 1  | 2  | 3  | 4  |
|-------|----|----|----|----|
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5  | 4  | 6  | 3  |

*Solved using Excel ☺*



*We will explore Complexity Classes in the next class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS** Pilani

Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

**DSECLZG519**

Parthasarathy





# Contact Session #15

## DSECLZG519 – Motivation to Complexity Classes

# Agenda for Session # 15

---

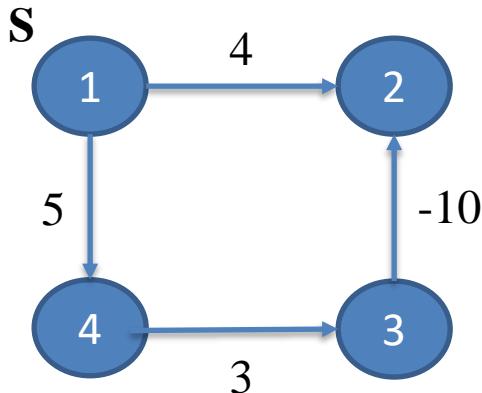
- Recap of Session #14
- Overview of Bellman Ford (Not in course)
- Quick Game
- Motivation to Complexity Classes
- Exercises

# Bellman Ford Algorithm

Not in course ...

- Solves single shortest path problem in which edge weight may be negative but no negative cycle exists.
- This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination.
- It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges.
- This algorithm detects the negative cycle in a graph and reports their existence.
- Based on the "***Principle of Relaxation***" in which more accurate values gradually recovered an approximation to the proper distance by until eventually reaching the optimum solution.

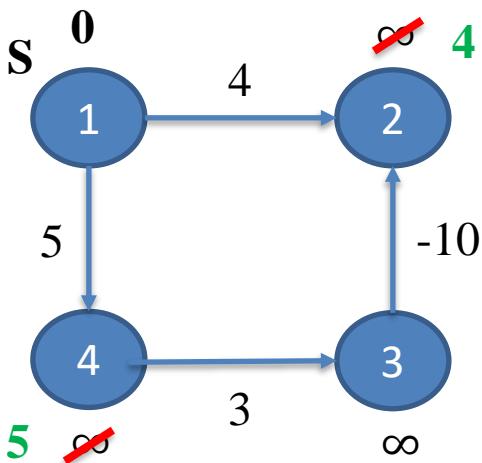
# Bellman Ford Algorithm Example



Here, we have 4 nodes, so we will relax  $n - 1$ ,  
Which is 3 times.

Write down all the edges: (In any order, doesn't matter) : (3,2) (4,3) (1,4) (1,2)

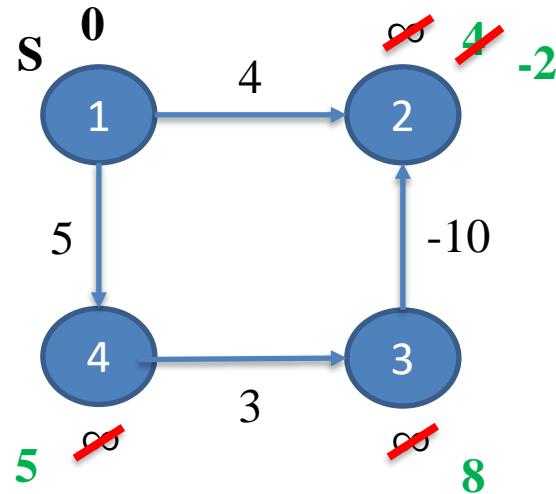
Mark path cost as 0 for Source and  $\infty$  for other nodes. Then start relaxing ☺ for 3 times.



## Relaxation – 1<sup>st</sup> Iteration

- $(3,2) = \infty - 10 < \infty$  (false), No change
- $(4,3) = \infty + 3 < \infty$  (false), No change
- $(1,4) = 0 + 5 < \infty$  (true). So, change cost at 4 from  $\infty$  to 5
- $(1,2) = 0 + 4 < \infty$  (true). So, change cost at 2 from  $\infty$  to 4

# Bellman Ford Algorithm Example



## Relaxation – 2<sup>nd</sup> Iteration

- $(3,2) = \infty - 10 < 4$  (false) , No change
- $(4,3) = 5 + 3 < \infty$  (true), So, change cost at 3 from  $\infty$  to 8
- $(1,4) = 0 + 5 < 5$  (false). No change.
- $(1,2) = 0 + 4 < 4$  (false). No change.

## Relaxation – 3<sup>rd</sup> Iteration

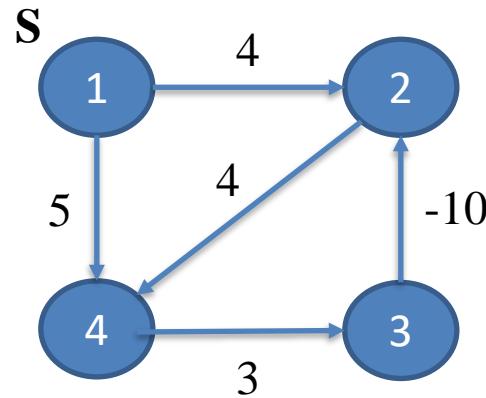
- $(3,2) = 8 - 10 < 4$  (true) , So, change cost at 2 from 4 to -2
- $(4,3) = 5 + 3 < 8$  (false). No change.
- $(1,4) = 0 + 5 < 5$  (false). No change.
- $(1,2) = 0 + 4 < -2$  (false). No change.

*Try to relax one more time ! Are answers changing ? No!*

| Path              | Cost |
|-------------------|------|
| $1 \rightarrow 1$ | 0    |
| $1 \rightarrow 2$ | -2   |
| $1 \rightarrow 3$ | 8    |
| $1 \rightarrow 4$ | 5    |

# Negative Edge Cycle ?

- A negative edge cycle is a cycle whose total weight cost is negative.



## Observations:

- There exists a cycle  $2 - 4 - 3 - 2$
- Its total cost is  $4 + 3 + (-10) = -3$
- Such a cycle is called as a negative cycle / negative edge cycle.

- If there is a graph G with negative edge cycle, then even after the  $(n - 1)$  iteration, the values would still get relaxed. Bellman Ford can detect it cannot be used to find the shortest path! ( Since, the edges involved in the negative edge cycle will keep on relaxing ).

# Quick Game ...

| Algorithm                     | Purpose | Strategy |
|-------------------------------|---------|----------|
| <i>Kruskal's</i>              |         |          |
| <i>Prim's</i>                 |         |          |
| <i>Dijkstra's</i>             |         |          |
| <i>Fractional Knapsack</i>    |         |          |
| <i>Huffman</i>                |         |          |
| <i>Karatsuba</i>              |         |          |
| <i>Merge &amp; Quick Sort</i> |         |          |
| <i>Bellman Ford</i>           |         |          |
| <i>Floyd's</i>                |         |          |
| <i>Warshall's</i>             |         |          |
| <i>Matrix Chain Product</i>   |         |          |
| <i>0/1 Knapsack</i>           | []      |          |

# Answers ...

| Algorithm                     | Purpose   | Strategy | Complexity   |
|-------------------------------|---|----------|--|
| <i>Kruskal's</i>              | Minimum Spanning tree   | Greedy   | Fill it as an exercise (It also depends on the Data structure used) !! |
| <i>Prim's</i>                 | Minimum Spanning tree   | Greedy   |  |
| <i>Dijkstra's</i>             | Single Source shortest path for G which has no negative weights               | Greedy   |  |
| <i>Fractional Knapsack</i>    | Maximize the profit, given the size of knapsack and objects with w and p      | Greedy   |  |
| <i>Huffman</i>                | Encode with variable number of bits per character                             | Greedy   |  |
| <i>Karatsuba</i>              | Integer Multiplication  | D&C      |  |
| <i>Merge &amp; Quick Sort</i> | Sorting   | D&C      |  |
| <i>Bellman Ford</i>           | Single Source shortest path for G which has positive & negative weights       | DP       |  |
| <i>Floyd's</i>                | All pair shortest path with negative & positive weights.                      | DP       |  |
| <i>Warshall's</i>             | Transitive Closure  | DP       |  |
| <i>Matrix Chain Product</i>   | To optimally parenthesize such that total scalar multiplications is minimized | DP       |  |
| <i>0/1 Knapsack</i>           | Maximize the profit, objects cannot be partitioned                            | DP       |  |

# Some other Algorithms to Explore



- Kruskal and Prims do not work for directed graph, explore about Edmond's algorithm.
- There is another algorithm to find out all pair shortest path problem – Johnson's algorithm.
- Travelling Sales man problem (DP)
- Strassen's Matrix multiplication (D&C)
- String pattern matching Algorithms like:
  - Horspool's algorithm
  - Boyer-more algorithm (Covered in Webinar 4)
  - Knuth-morris-pratt algorithm
- You will learn a lot more algorithm which are '*Data science*' specific in future semesters. Do ask yourself its strategy & complexity and complexity varies if you change the underlying Data structure ☺

# Complexity Classes Motivation

---

- ***Input:*** A set of Jobs  $J_1, J_2, \dots, J_n$  and each of them have a size associated with them say  $S_1, S_2, \dots, S_n$ .
- There are two processors P1 and P2 and your task is to automate the process of scheduling i.e. by writing a program which takes the sizes of the jobs as input (say from an array or file or screen) and schedule them on the processors P1 and P2.
- Assume Pre-emption is not allowed, meaning if we start a job – we cannot stop until it completes.
- The goal is that the last job finishes fastest.
- What are the different ways to do this?
- **Attempt 1:** Say you don't use P2, and schedule every job on P1. Then the time taken will be  $S_1 + S_2 + \dots + S_n$

# Complexity Classes Motivation

- *Example:*  $S_1=10, S_2=4, S_3=2, S_4=14, S_5=20$
- You schedule everything in P1 and hence at time unit 50 ( $10+4+2+14+20$ ), all jobs will be completed.
- **Attempt 2:** You want to optimize this approach by using both the processors. You decide to schedule all the odd jobs in P1 and even jobs in P2.

| P1    | P2    |
|-------|-------|
| $S_1$ | $S_2$ |
| $S_3$ | $S_4$ |
| $S_5$ |       |

- Now, at time unit 32, all the jobs are completed. You are happy and inform your success to your friends.

# Complexity Classes Motivation

- Your friends give an input as follows:
- $S_1=1, S_2=85, S_3=2, S_4=90, S_5=3, S_6=100$
- Your algorithm schedules  $S_1, S_3$  and  $S_5$  on P1 and  $S_2, S_4$  and  $S_6$  on P2

| P1    | P2    |
|-------|-------|
| $S_1$ | $S_2$ |
| $S_3$ | $S_4$ |
| $S_5$ | $S_6$ |
| 6     | 275   |

- The time taken is 275 units to complete the jobs! This is clearly not best!
- *For Example:* If  $S_1, S_3, S_4, S_6$  was scheduled in P1 and  $S_2, S_5$  was scheduled in P2 then by 193 units, all jobs would have got completed!

# Complexity Classes Motivation

- Based on the previous failure, you realize that you must evenly distribute the jobs (as much as possible) among the processors.
- You come up with another approach!
- **Attempt 3:** If there are  $P_i$  processors, you schedule  $i$  jobs in the  $P_i$  processors. After this for the other jobs, you schedule them on the least lightly loaded processor!
- *Example:*  $S_1, S_2, S_3, \dots$  There are 2 processors, so you schedule  $S_1$  in P1 and  $S_2$  in P2.
- From  $S_3$  onwards, we see the least lightly loaded processor and schedule it there. i.e. say if  $S_1 > S_2$  then we put  $S_3$  in  $P_2$  and continue this way ... For scheduling  $S_4$  Again check if  $S_1 > S_2 + S_3$  if so schedule  $S_4$  on  $P_2$  else on  $P_1$ . Break ties as you want and continue the process ...

| P1    | P2    |
|-------|-------|
| $S_1$ | $S_2$ |
|       |       |
|       |       |

| P1    | P2                      |
|-------|-------------------------|
| $S_1$ | $S_2$                   |
|       | $S_3$ (if $S_1 > S_2$ ) |
|       |                         |

# Complexity Classes Motivation

---

- You show this algorithm to the friends and this seems to work!  
You show this to the boss! All good 😊
- You again face an issue the next day where your algorithm takes  $T$  units of time and the boss is able to produce another schedule which is less than  $T$ .
- *Reason?*
- Maybe because the job sizes was a mixture and your algorithm did not work.
- **Attempt 4:** You now sort the jobs according to size such that  $S_1 < S_2 < S_3 \dots$  and follow the same approach.
- As usual, you show it to your friends and they give you jobs with sizes 2,3,2,3,2.

# Complexity Classes Motivation

- You sort them and make it 2,2,2,3,3 and schedule them as follows:

| P1 | P2 |
|----|----|
| 2  | 2  |
| 2  | 3  |
| 3  |    |

*At time unit 7, all the jobs got completed.*

- Again mockery from friends because optimal for these jobs is to assign 2,2,2 in P1 and 3,3 in P2. In which case, at time unit 6, all the jobs will get completed. So, attempt 4 fails as well.
- By the way, the approach that you are attempting is which strategy?
- Greedy! You are greedily assigning the job to the processor ( putting in least lightly loaded processor).

# Complexity Classes Motivation

---

- You try to come up with some other approaches ... Any other algorithmic strategy that can come handy ?
- Probably not ☹ You can surely come up with a smart solution for a given case but an algorithm which works optimally to schedule the jobs always is possibly impossible !!
- *Brute Force the saviour.*
- Final Attempt (after you are frustrated):
  - $S_1, S_2, \dots, S_n$
  - Take all the possible subsets of  $S$
  - Choose the minimum and schedule them on the processor.
- Will this always give the correct answer ? Yes
- All good and submitted the solution to the boss!

# Complexity Classes Motivation

---

- The boss tests your solution by giving some inputs and everything works like a charm!
- Until ...
- The boss gives an input with 1000 Jobs !
- The algorithm runs for
  - 10 min ...
  - 1 hour ...
  - 10 hours ....
  - 2 days ...
  - 10 days ... and still the algorithm has not stopped
- Now, he calls you for a 1 – 1 meeting ☺
- He asks you the golden question : “*When will the algorithm stop ?*”

# Complexity Classes Motivation

---

- $N = 1000$
- We are create all possible subsets!
- If there are  $N$  numbers in a set, how many subsets can be formed ?

$$2^N$$

In this case,  $2^{1000}$

- Lets assume that in 1 instruction, the computer can process 1 of these subsets.
- Lets say the fastest computer can perform  $10^{20}$  instructions/sec.
- Lets say we use ALL the computers in this world to solve this ( currently we have around 2+ billion computers)  $\rightarrow 10^{20}$  computers

# Complexity Classes Motivation

- So per second we can do  $10^{40}$  instructions if we use all the computers. ( $10^{20} * 10^{20}$  )
- What about a year ? How many instructions can we perform in a year ?
- $10^{40} * 60 * 60 * 24 * 365$  instructions / year
- Lets generalize this and let's be very generous here :

$$\begin{array}{c}
 10^{40} * 60 * 60 * 24 * 365 \\
 \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
 10^{40} * 10^2 * 10^2 * 10^2 * 10^3 \rightarrow \sim 10^{50} \text{ instructions / year}
 \end{array}$$

We are very generous here!

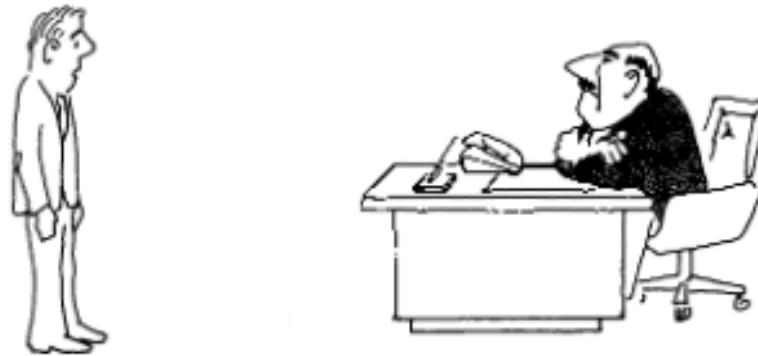
- So maximum we can perform  $10^{50}$  instructions per year and we have  $2^{1000}$  instructions to perform.
- How many years will this take ?

# Complexity Classes Motivation

- Required to execute :  $2^{1000}$  instructions
- Capable of executing :  $10^{50}$  instructions/year  $\rightarrow \sim 2^{4.50} \rightarrow \sim 2^{200}$  instructions/year
- Number of years required ?
- Number of years required = 
$$\frac{\text{Number of Instructions}}{\text{Number of Instructions per year}}$$
- Number of years required = 
$$\frac{2^{1000}}{2^{200}} = 2^{1000-200}$$
- So, the number of years needed is  **$2^{800}$**
- *Fun Fact:*  **$2^{800}$**  is ~ equal to number of atoms in this universe ☺ ).  
Realize the blunder of such algorithms?

# Complexity Classes Motivation

- If you tell this to your boss, what's the consequence ?
- So, what do we tell the boss ?



**"I can't find an efficient algorithm,  
I guess I'm just too dumb."**

# What you would ideally like to Say?



"I can't find an efficient algorithm,  
because no such algorithm is possible!"

*This is where Complexity Classes come into picture ☺*

# Complexity Theory

- We would tell the boss that this is a “*hard*” problem! We will tell that this has not been solved efficiently by anyone yet. If this gets solved, then a lot of similar “*hard*” problems would also get solved.
- We tell his and prove this and save the job!



“I can’t find an efficient algorithm,  
but neither can all these famous people.”

# Complexity Theory

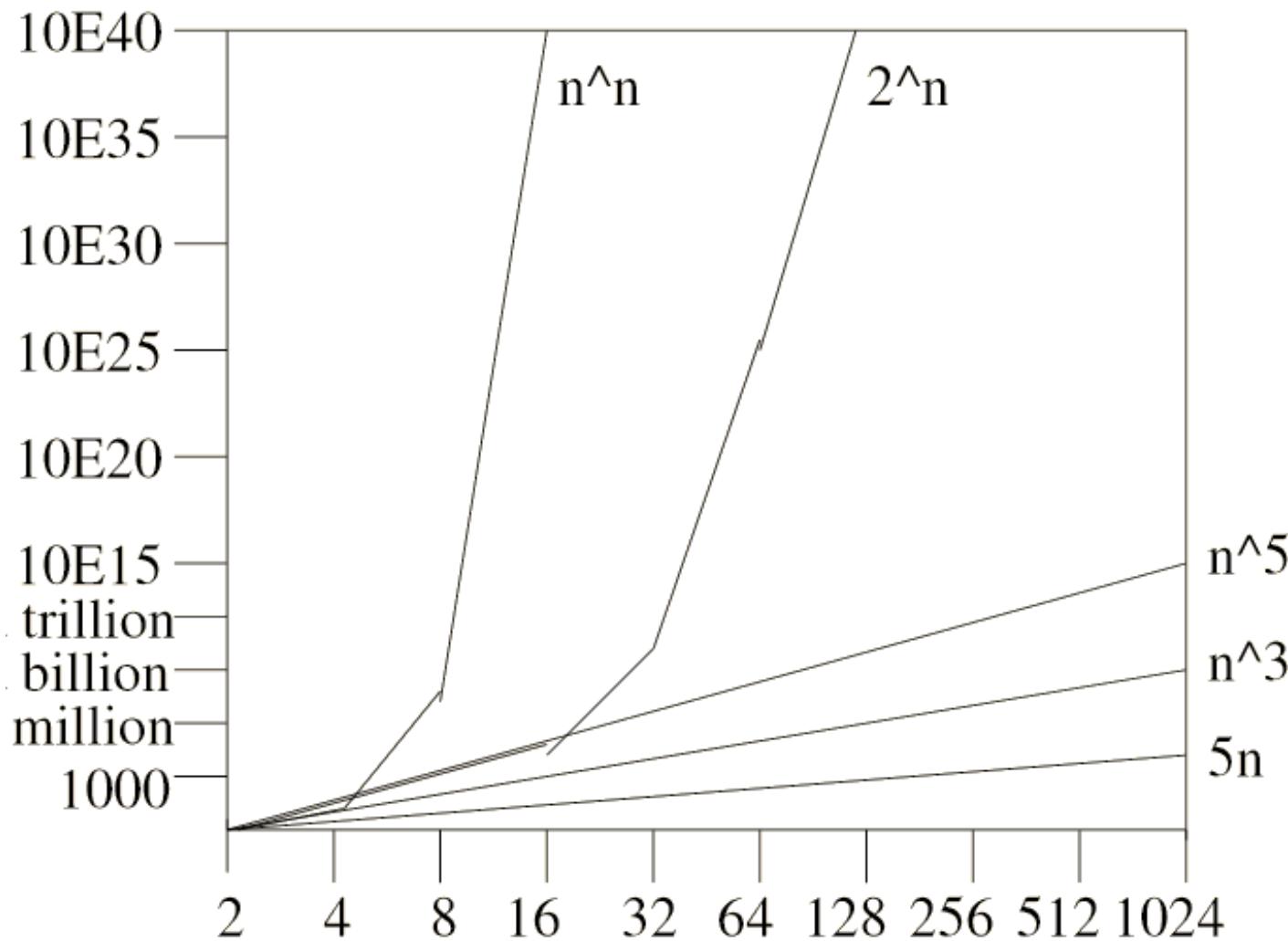
## Polynomial Time Taking

- Linear Search –  $n$
- Binary Search –  $\log n$
- Insertion Sort –  $n^2$
- Bubble Sort –  $n^2$
- Merge Sort –  $n \log n$
- Matrix Multiplication –  $n^3$
- .....
- .....
- .....

## Exponential Time Taking

- Job Scheduling –  $2^n$
- Sum of Subsets –  $2^n$
- Travelling Salesman –  $2^n$
- Hamiltonian Cycle –  $2^n$
- Graph coloring –  $2^n$
- ...
- ...
- ...

# Complexity Theory



# Complexity Theory

---

- On the basis of this classification of functions into polynomial and exponential, we can classify algorithms:
- **Polynomial-time Algorithms:**  
An algorithm whose order of magnitude time performance is bounded from above by a polynomial function  $n$ , where  $n$  is the size of the inputs.
- **Exponential Algorithms:**  
An algorithm whose order of magnitude time performance is not bounded from above by a polynomial function of  $n$ .

# Complexity Theory

- **Tractable Problem:** a problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial. Example : Searching, sorting, multiplication of integers, finding a MST etc.
- **Intractable Problem:** a problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential. Example: Towers of Hanoi, SAT, TSP etc.
- **Tractable:** controllable, manageable, malleable, governable, yielding, amenable, complaisant, compliant, adjustable, docile, submissive, obedient, tame, meek, easily handled.
- **Intractable:** unmanageable, uncontrollable, ungovernable, out of control, out of hand, impossible to cope with; difficult, awkward, complex, troublesome, demanding, burdensome, “their problems have become more severe and intractable.

# P- Class

---

- The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time  $O(n^k)$  in worst-case, where  $k$  is constant.
- For those problem belonging in this class, the below properties holds good:
  - We can solve them in polynomial time
  - We already know a deterministic algorithm for the same.
- Examples of problems which belong to this class are Binary Search, Sorting, Matrix multiplication, All pair shortest path problem etc.

- Now there are a lot of programs that don't (necessarily) run in polynomial time on a regular computer, but do run in polynomial time on a non-deterministic Turing machine. These programs solve problems in **NP**, which stands for **non-deterministic polynomial time**. A non-deterministic Turing machine can do everything a regular computer can and more. This means all problems in P are also in NP.
- An equivalent way to define NP is by pointing to the problems that can be **verified in polynomial time**. This means there is not necessarily a polynomial-time way to find a solution, but once you have a solution it only takes polynomial time to verify that it is correct.
- *Verifiable?*
  - If we are somehow given a ‘certificate’ of a solution we can verify the legitimacy in polynomial time.
- *A Turing machine is a hypothetical machine thought of by the mathematician Alan Turing in 1936. Despite its simplicity, the machine can simulate ANY computer algorithm, no matter how complicated it is!*

- NP class consists of those problems which can be solved by a *non-deterministic algorithm in polynomial time*.
- NP stands for **N**on-deterministic **P**olynomial
- The class NP consists of those problems that are *verifiable* in polynomial time.
- Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.
- Every problem in this class can be solved in exponential time using exhaustive search.
- Lets consider what is non-deterministic algorithm with an example.

# NP - Class

The kinds of problems we deal with in complexity theory come in pairs:  
a "search" version and a "verification" version. For example:

- **Problem:** Sorting.  
**Search version:** Input a list of numbers X and output the same list in sorted order (call it Y).  
**Verification version:** input a list of numbers X and another list Y, and output "YES" if Y is the sorted version of X and "NO" otherwise.
- **Problem:** graph coloring.  
**Search version:** Input a network of nodes and edges X, and output colors Y for each node such that no adjacent nodes have the same color.  
**Verification version:** input a network X and a set of colors Y and output "YES" if all adjacent nodes have different colors and "NO" otherwise.

# Non-Deterministic

## DETERMINISTIC ALGORITHM

For a particular input the computer will give always same output.

Can determine the next step of execution.

## NON-DETERMINISTIC ALGORITHM

For a particular input the computer may give different output on different execution.

Cannot determine the next step of execution due to more than one path the algorithm can take.

# Non-Deterministic

- In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in case of **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs.
- *Meaning, non-deterministic algorithms have some degree of randomness.*
- Some of the terms related to the non-deterministic algorithm are defined below:

  - **choice(X)** : chooses any value randomly from the set X.
  - **failure()** : denotes the unsuccessful solution.
  - **success()** : Solution is successful and current thread terminates.

Let's say we have an array 'a' with 'n' elements and a key 'x' is to be searched in the array 'a'. A non-deterministic algorithm for this is as follows:

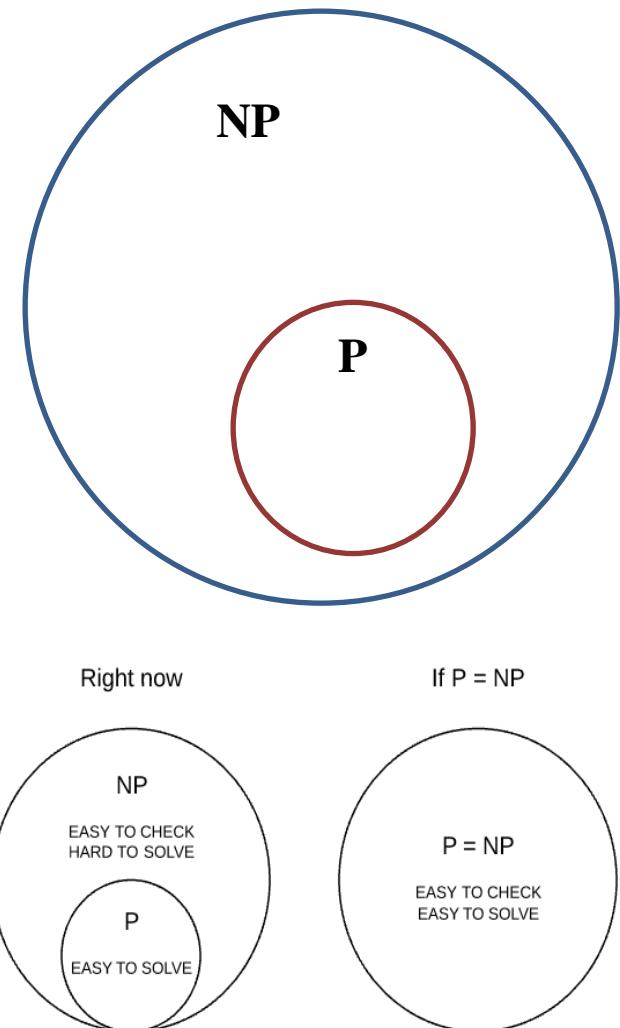
```

1. j= choice(a, n)
2. if(A[j]==x) then
    {
        write(j);
        success();
    }
3. write(0); failure();

```

# P and NP

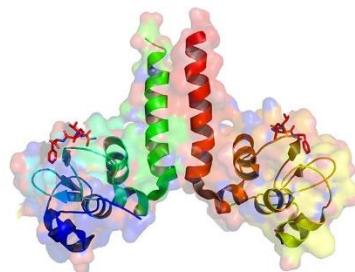
- The problem belongs to class **P** if it's easy to find a solution for the problem. The problem belongs to **NP**, if it's easy to check a solution that may have been very tedious to find.
- Clearly,  $P \subseteq NP$
- The P vs NP is a famous unsolved Computer Science problem which asks whether every problem whose solution can be quickly (in polynomial time) verified (NP problem) can also be solved quickly (P class).
- In other words, will NP become equal to P ?
- Time will answer this! ☺



# P and NP

If  $P = NP$ , then the world will be different!

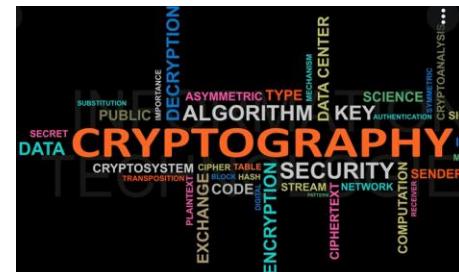
- Transport routing problems will be efficient
  - Job scheduling's will be best
  - Massive advances in ML and AI
  - Protein folding problem's could be solved too!
  - Many more ...



<http://www.claymath.org/millennium-problems>

## Downsides of P = NP :

- Cryptography will completely go for a toss ☹
  - If  $P = NP$ , then we could break most cryptography algorithms!



# Exercises

Apply DP algorithm to the following 0/1 Knapsack instance.

| item | weight | value |
|------|--------|-------|
| 1    | 3      | \$25  |
| 2    | 2      | \$20  |
| 3    | 1      | \$15  |
| 4    | 4      | \$40  |
| 5    | 5      | \$50  |

, capacity  $W = 6$ .

2) N=4, Max W= 8

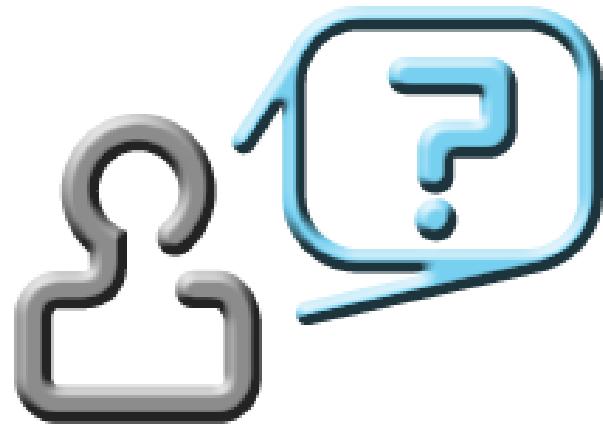
3)

| $i$ | 1  | 2  | 3  | 4  | 5  |
|-----|----|----|----|----|----|
| $p$ | 30 | 20 | 40 | 70 | 60 |
| $w$ | 4  | 1  | 2  | 5  | 3  |

| Pi | 1 | 2 | 5 | 6 |
|----|---|---|---|---|
| Wi | 2 | 3 | 4 | 5 |

$n = 5$

$c = 10$



*We will explore further in the next(last) class ...*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : [CC BY-NC-SA 4.0](#)





**BITS** Pilani  
Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

## DSECLZG519

Parthasarathy





# Contact Session #16

# DSECLZG519 – Complexity Classes

# Agenda for Session # 16

---

- Journey so far ...
- Revisit Motivation for Complexity Classes
- Terms involved in Complexity Classes
  - P, NP
  - Reductions
  - CNF-SAT
  - NP-Hard and NP-Complete
- Reductions
- Gyan & Feedback!

# Journey So Far ...

CS#1  
Introduction & Analysis framework

CS#2  
Analysis and Master's Theorem

CS#3 Linear Data structures

CS#4  
Introduction to Trees

CS#5  
Heaps , Heap Sort and Priority Queues!

CS#6  
Introduction to Graphs

CS#7  
Exploring Graphs & Traversals

CS#8  
Hashing and Collisions

Mid Semester Examination

# Journey So Far ...

CS#9  
Esoteric  
Data  
structures

CS#10  
Introduction  
to AVL Trees

CS#11  
Introduction  
to Greedy  
Techniques

CS#12  
Exploring  
Greedy –  
MST,  
Dijkstra's

CS#13  
Divide and  
Conquer

CS#14  
Dynamic  
Programming

CS#15  
Completed  
DP &  
Motivation to  
Complexity  
Classes

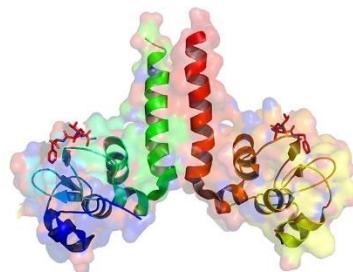
CS#16  
Completion  
of  
Complexity  
Classes &  
Discussion,  
Feedbacks

End Semester Examination

# P and NP

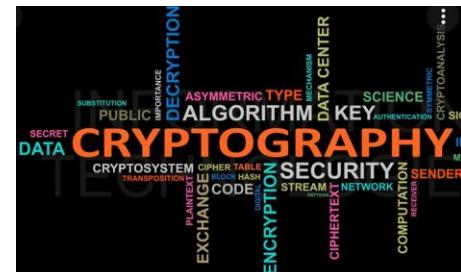
If  $P = NP$ , then the world will be different!

- Transport routing problems will be efficient
- Job scheduling's will be best
- Massive advances in ML and AI
- Protein folding problem's could be solved too!
- Many more ...



## Downsides of $P = NP$ :

- Cryptography will completely go for a toss 😞
- If  $P = NP$ , then we could break most cryptography algorithms!



<http://www.claymath.org/millennium-problems>

# Reductions

- Instead of solving every NP problem separately, it makes sense to show that they are all related so that if one is solved, all of them gets solved!
- How do we show that they are related / associated to one another ?
- We use a technique called **Reduction**.
- Reductions are a very common and powerful idea in mathematics and computer science.
- The idea is to solve a new problem by reducing (mapping) it to one for which we already know how to solve it.
- We will ideally ‘reduce’ any already known ‘hard’ problem to our problem then ‘our’ problem and the ‘hard’ problem are related.
- So, once the hard problem is solved, our problem is also solved.

# Reductions

---

- A problem Q can be reduced to another problem Q' if any instance of Q can be “*easily rephrased*” as an instance of Q', the solution to which provides a solution to the instance of Q
- Is a quadratic equation reducible to a linear equation?
  - *Sure! Let co-efficient of the square term be 0*

# CNF-SAT Problem

- CNF – Boolean formula is in Conjunctive normal form (CNF) if it is formed as a collection of sub-expressions, called clauses, that are combined using AND, with each clause formed as the OR of boolean variables or their negation, called literals. Ex:  $(x_1 + x_2)$   $(x_2' + x_3)$ .
- The CNF-SAT (CNF-Satisfiability) problem takes a boolean formula in CNF as input and asks if there is an assignment of boolean values to its variables so that the formula evaluates to 1.
- Ex:  $x_i = \{ x_1, x_2, x_3 \}$  and the CNF =  $(x_1 \vee x_2 \vee x_3')$   $\wedge (x_1' \vee x_2' \vee x_3)$  with 2 clauses.



Clause C1



Clause C2

# CNF-SAT Problem

- Problem is to check for which values of  $x_1, x_2, x_3$  the CNF holds true (or CNF results in 1).
- $\text{CNF} = (x_1 \vee x_2 \vee x_3') \wedge (x_1' \vee x_2' \vee x_3)$
- There are 3 variables. How many possible combinations ?
- 8 (which is nothing but  $2^3$  )
- Which means that we need to check  $2^3$  combinations to check if any of them satisfy the CNF.
- If we have a CNF-SAT problem with n variables, then, we will have to check  $2^n$  combinations.

| <b>X<sub>1</sub></b> | <b>X<sub>2</sub></b> | <b>X<sub>3</sub></b> |
|----------------------|----------------------|----------------------|
| 0                    | 0                    | 0                    |
| 0                    | 0                    | 1                    |
| 0                    | 1                    | 0                    |
| 0                    | 1                    | 1                    |
| 1                    | 0                    | 0                    |
| 1                    | 0                    | 1                    |
| 1                    | 1                    | 0                    |
| 1                    | 1                    | 1                    |

*So, the CNF-SAT problem also takes exponential time!*

# CNF-SAT Problem

---

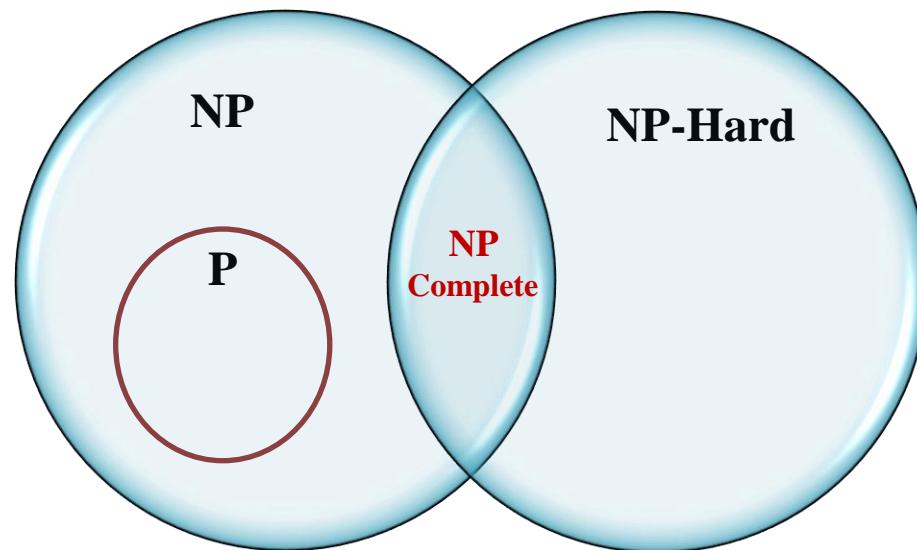
- It is already proved that CNF-SAT is NP-Hard and NP-Complete.  
[Refer text for proof!]
- We can now reduce other problem instances (A) from CNF-SAT and if we are able to do that, then the other problem (A) also becomes NP-Hard.

# Transitive Reductions

- 
- Reductions are transitive in nature!
  - $\text{Sat} \times L_1$  (Satisfiability reduces to  $L_1$ . Then  $L_1$  is NP-Hard).
  - $L_1 \times L_2$  ( $L_1$  to  $L_2$ . Then  $L_2$  is also NP-Hard)

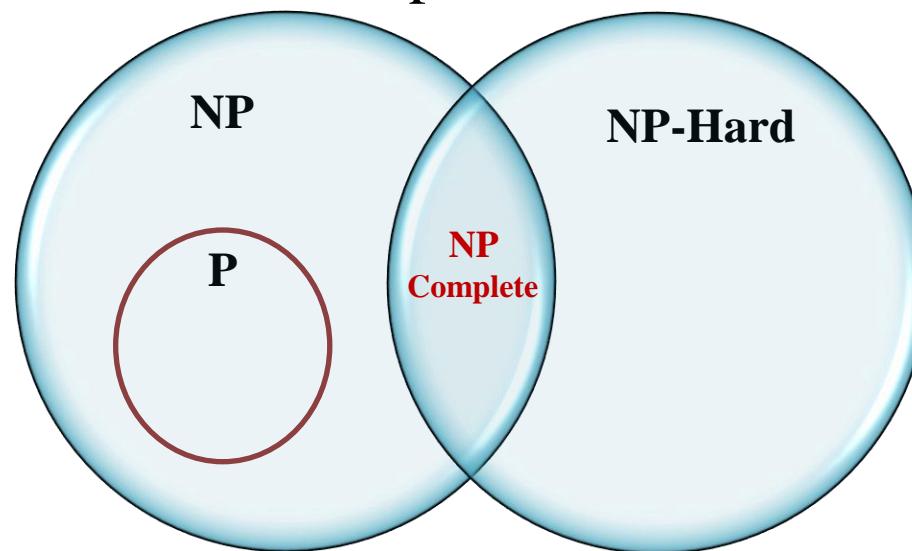
# NP-Hard & NP-Completeness

- Let  $B$  be a decision problem.
- $B$  is called **NP-hard** if  $A \leq_p B$  for all problems  $A \in \text{NP}$ .
- $B$  is called **NP-complete** if  $B \in \text{NP}$  and  $B$  is NP-hard.
- NP-hard problems are “*at least as hard*” as all problems in NP.
- NP-complete problems are “*the hardest*” problems in NP.



# NP-Hard & NP-Completeness

- A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.
- If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.



# Reductions and NP-Completeness

## Theorem

If  $Y$  is NP-complete, and

- ①  $X$  is in NP
- ②  $Y \leq_P X$

then  $X$  is NP-complete.

In other words, we can prove a new problem is NP-complete by reducing some other NP-complete problem to it.

# Cook's Theorem

---

## Cook's Theorem:

If SAT has an efficient algorithm then so does other problems in NP.

## Cook-Levin Theorem:

CNF – Satisfiability is NP-Complete.

# Proving New Problems as NP-Hard / NP-Complete

The recipe to prove NP-Hardness of a problem X :

- Find an already known NP-Hard problem Y.
- Show that  $Y \leq X$

The recipe to prove NP-Completeness of a problem P :

Let  $P' \leq_P P$  and let  $P'$  be NP-complete. Then  $P$  is NP-hard.

Let  $P' \leq_P P$  and let  $P'$  be NP-complete, and  $P \in NP$ .  
Then  $P$  is NP-complete.

**Warning:** You should reduce the *known* NP-complete problem to the problem you are interested in. (You *will* mistakenly do this backwards sometimes.)

# Amusing Analogy

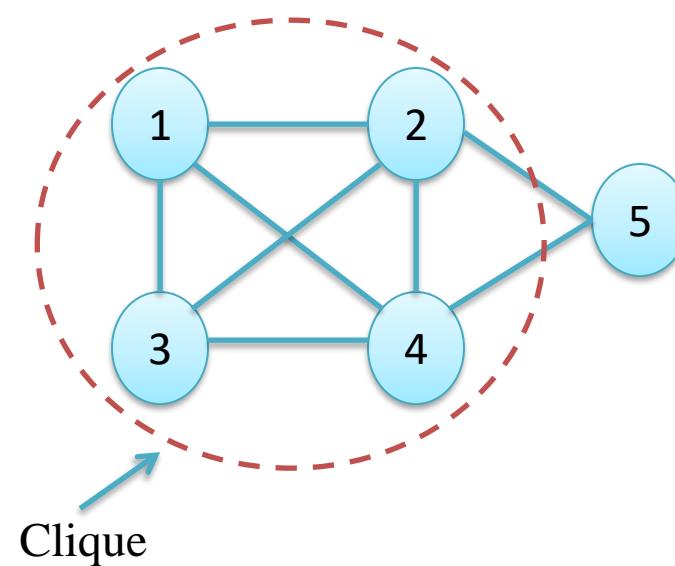
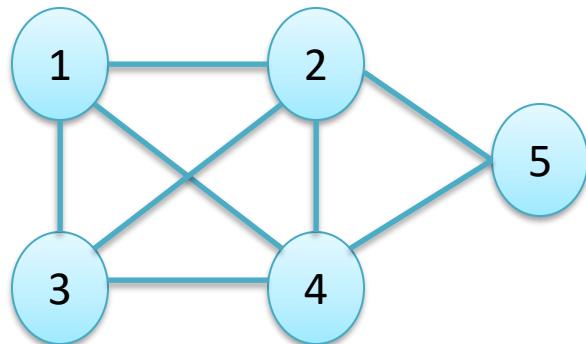
---

- Students believe that every problem assigned to them is NP-complete in difficulty level, as they have to find the solutions 😕
- Teaching Assistants, on the other hand, find that their job is only as hard as NP, as they only have to verify the student's answers 😎
- Professors know that all the assigned problems are in class P 😜
- When some students confound the TAs, even verification becomes hard ..

# Clique

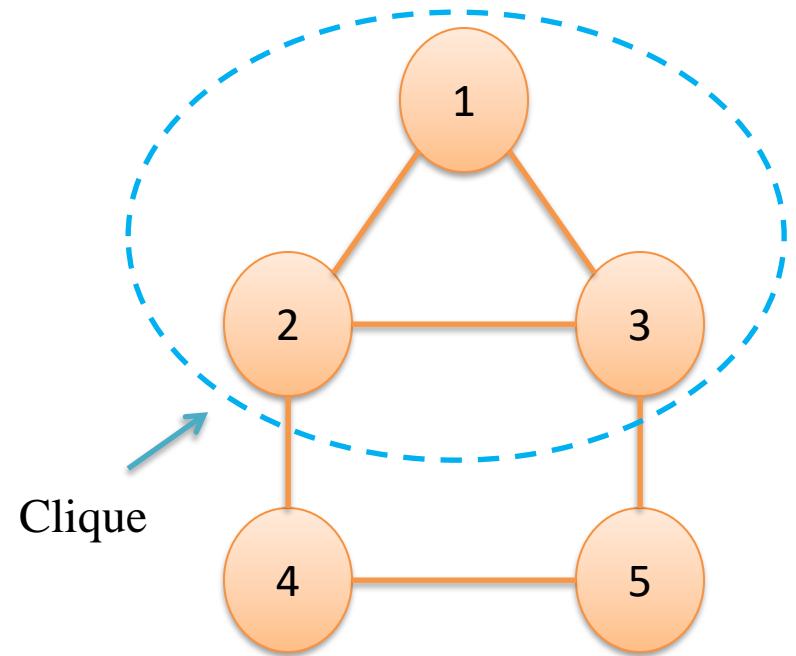
## Clique

- A clique in an undirected graph is a subset of vertices such that each pair is connected by an edge.



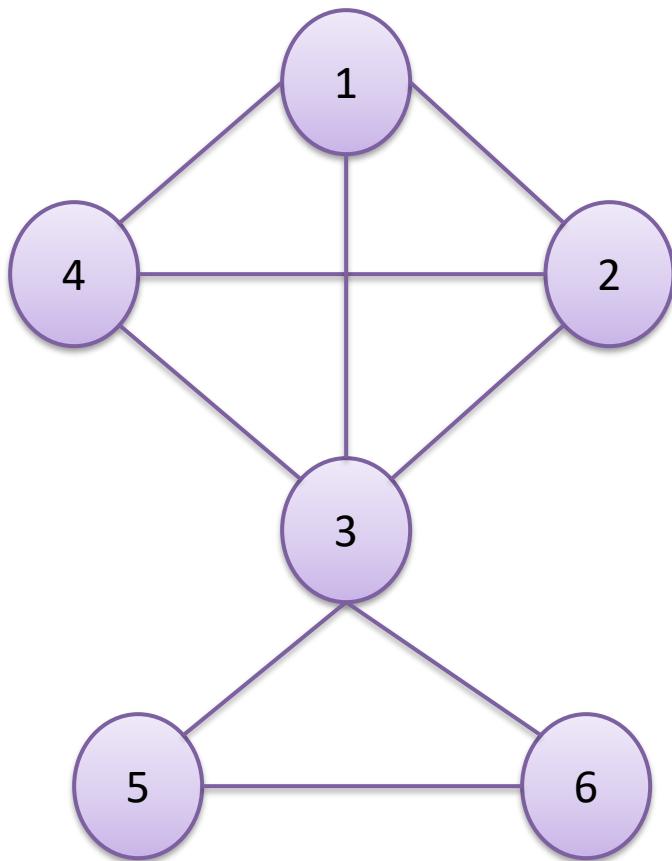
- In other words, Clique is a subgraph of a graph which is complete!

# Clique



- Is there a subgraph in this graph which is complete ?
- In other words, is there a clique ?
- ***Yes!***
- How many vertices are part of this clique ?
- ***Three* ...**

# Clique



- Is there a subgraph in this graph which is complete ?
- In other words, is there a clique ?
- ***Yes!***
- How many cliques are present ?
- $K = 4$  (*Clique with 4 vertices*)
- $K = 3$  (*Clique with 3 vertices*)
- $K = 2$  (*Clique with 2 vertices*)

# Decision Problem vs Optimization Problem

---

- Problems whose answers are boolean (Yes/No) is a decision problem.
- Example:
  - *Is there a clique of size 4 in the Graph G ?*
  - *Is there a clique of size 5 in the Graph G ?*
  - *Is there a clique of size 3 in the Graph G ?*
- Problems whose answers require us to perform maximization / minimization are optimization problems.
- Example:
  - *Find the maximum sized clique in the Graph G*
- In general in our study of complexity classes, we will focus on decision problems.

# Clique is NP Complete ?

The recipe to prove NP-Hardness of a problem X :

- *Find an already known NP-Hard problem Y.*
- *Show that  $Y \not\leq_p X$*

- So, in our case  $X = \text{Clique}$  decision problem.
- We need to take any  $Y$  which is already a known NP Hard problem and reduce it to Clique decision problem.
- We know that 3SAT is a NP Hard problem. Hence, lets reduce 3SAT to clique and if we are able to do it, then Clique is also NP-Hard.

# Reduce 3-CNF SAT to Clique

- Let's consider the below 3SAT:

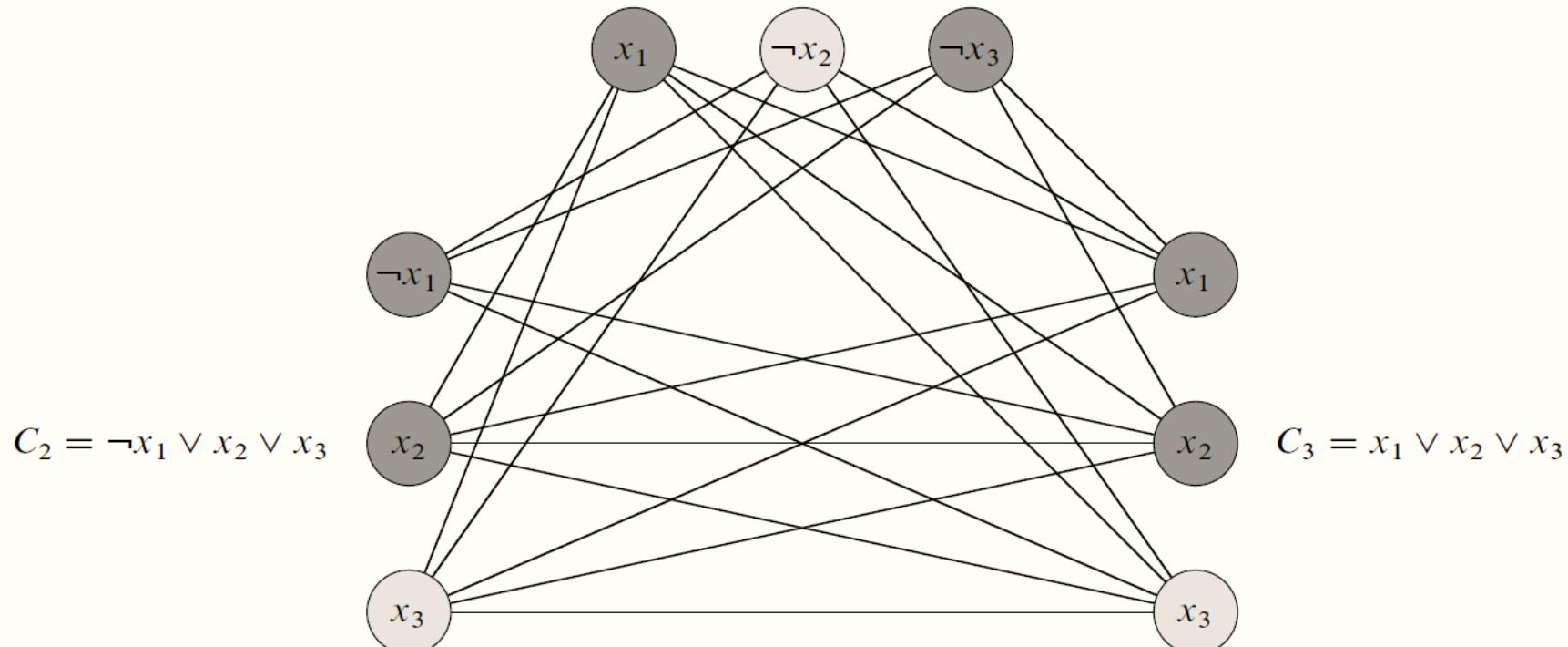
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

- From this CNF, if we are able to draw a Graph which is having a clique of size 3 (here there are 3 clauses), then we are showing that 3SAT reduces to Clique.
- For each clause, create a vertex for each literal
- For the edges:
  - *Connect vertices if they come from different clauses*
  - *Even if the vertices come from different clauses, do not connect if it results in incompatibility. No variable should be connected to its negation.*

# Reduce 3-CNF SAT to Clique

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

$$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$$



# Reduce 3-CNF SAT to Clique

---

- From the previous slide, we have proved that Satisfiability problem reduces to Clique.
- Hence, *Clique is NP-Hard*.
- Writing a NP algorithm for Clique is easy! You can give it a try ...
- Hence, Clique is NP complete.

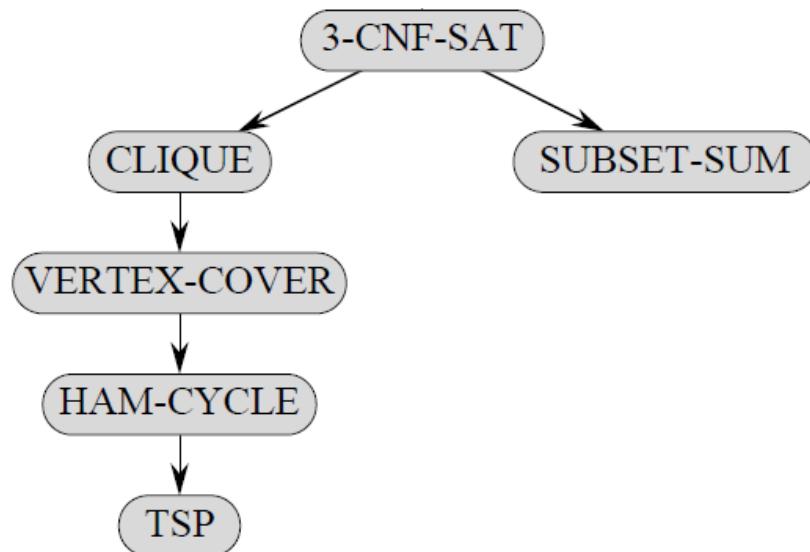
# List of common NP-Complete Problems



- SAT
- Clique
- Vertex Cover Problem
- Hamiltonian Cycle
- TSP (decision version)
- Sum of subset
- Independent set problem
- Graph coloring problem
- Subgraph isomorphism problem
- ... and many more !

# The Reducibility 'tree'

- Richard Karp proved 21 problems to be NP complete in a seminal 1971 paper
- Not that hard to read actually!
- Definitely not hard to read it to the point of knowing what these problems are.
- [Karp's paper](#)



# Concept of Approximation Algorithms



- Many problems of practical significance are NP-complete but are too important and cannot be abandoned merely because obtaining an optimal solution is intractable.
- If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it exactly, but even so, there may be a hope.
- There are at least three approaches to getting around NP-completeness.
- First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
- Second, we may be able to isolate important special cases that are solvable in polynomial time.
- Third, it may still be possible to find near-optimal solutions in polynomial time (either in the worst case or on average).
- In practice, near-optimality is often good enough. An algorithm that returns near-optimal solutions is called an approximation algorithm.

# NP Completeness in 'everyday' life

- Assigning university classes to timeslots to minimize scheduling conflicts
- Wedding planning where you want to seat friends together at the same table but not any enemies (along the lines of CLIQUE)



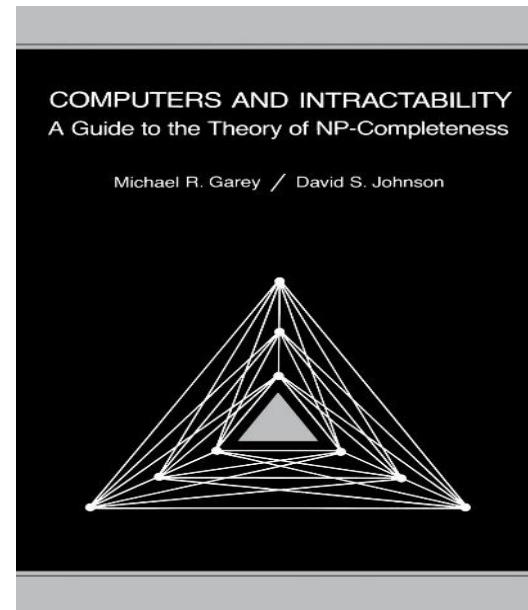
# NP Complete problems

---

- Painful to solve
- Even when we reduce complexity (by DP for instance) we still have non polynomial run times
- But these are practical problems!
- How to solve?
  - Approximation algorithms
  - Heuristics
  - Randomized algorithms
- They are ‘unsolved’ so new and interesting results can be shown about them.
- People pay good money for solving these problems

# Further Readings

- Chapter 13 from Textbook [ Algorithm Design by Goodrich and Tamassia].
- Chapter 34 from Introduction to Algorithms by CLRS
- Computers and intractability - A Guide to the Theory of NP-Completeness-W. H. Freeman (1979)



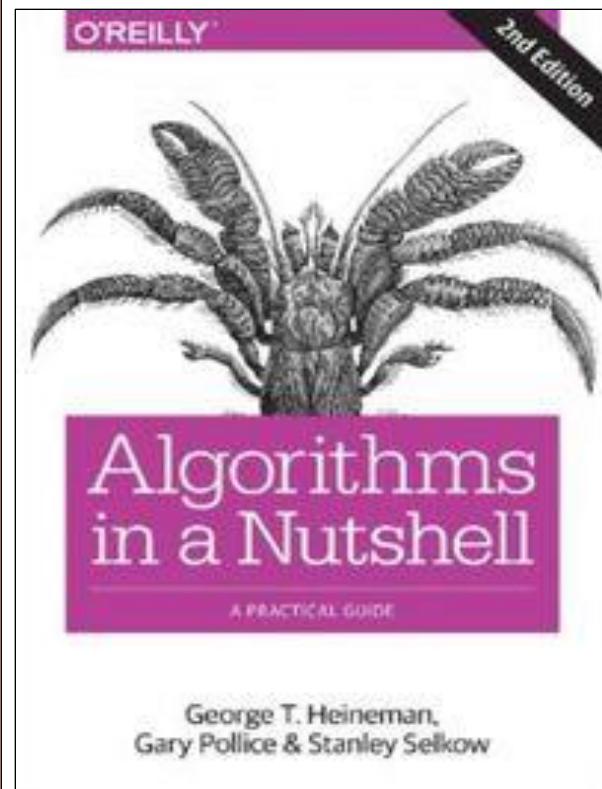
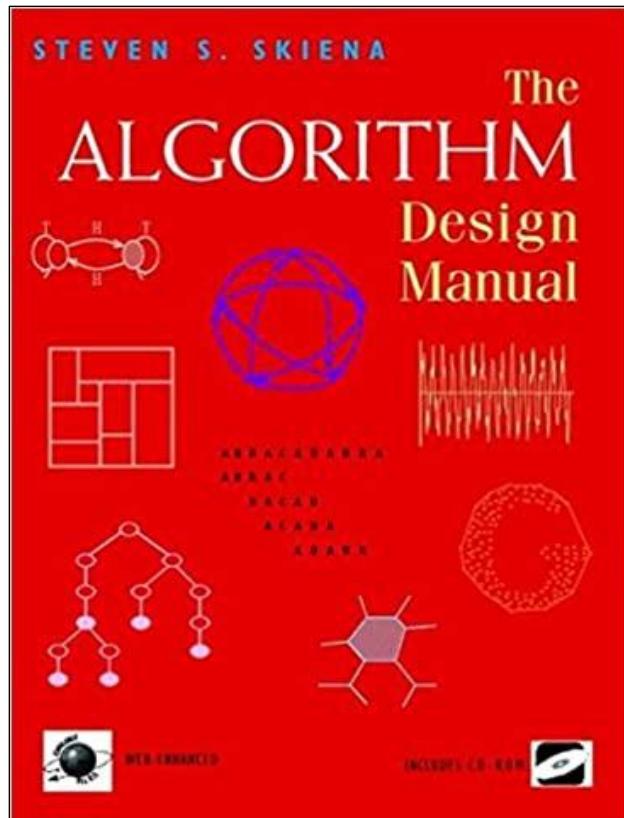
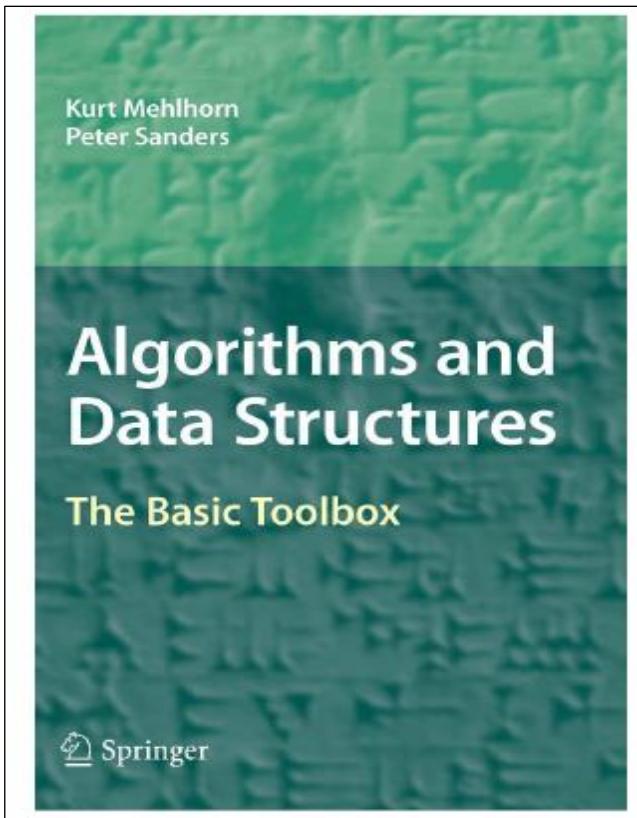
# Some other Algorithms Strategies to Explore

---

- Backtracking
- Branch & Bound
- Approximation Algorithms
- Randomized Algorithms
- Genetic Algorithms

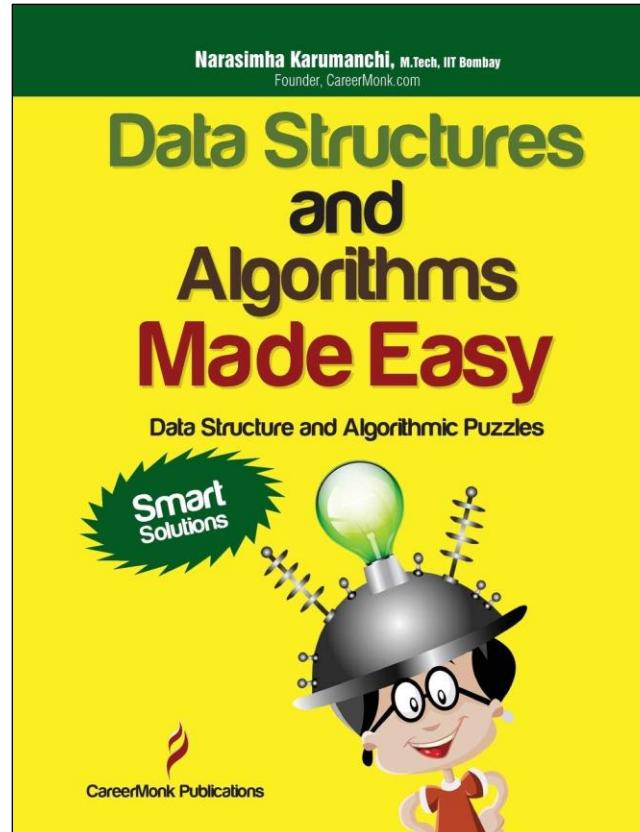
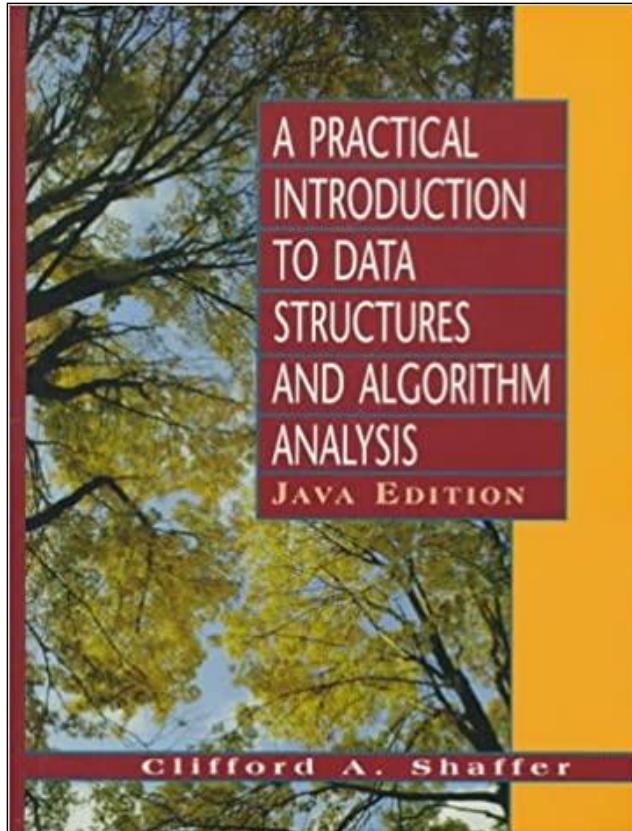
You will learn a lot more algorithm which are '*Data science*' specific in future semesters. Do ask yourself its strategy & complexity 😊

# Going “extra” mile ...



*Note: These are not the prescribed text/references. Please refer to the course handout for the textbook and references. These are in response to your requests to go the ‘extra-mile’.*

# Going “extra” mile ...



*And numerous  
other books &  
websites ...*

*Note: These are not the prescribed text/references. Please refer to the course handout for the textbook and references. These are in response to your requests to go the ‘extra-mile’.*

# Way ahead ?

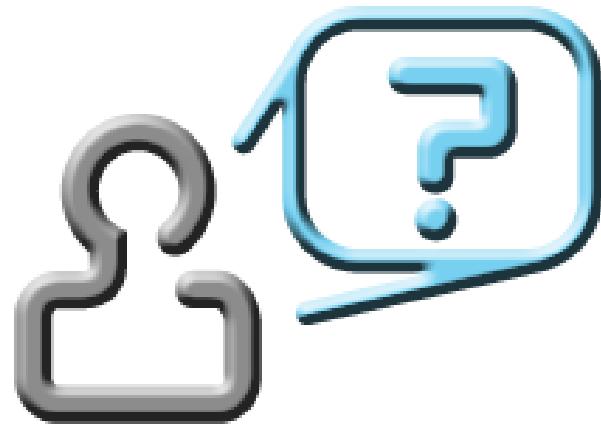
## Master DSAD ? Practice!

- Practice DSAD Problems on:
- <https://www.algoexpert.io/product>
- <https://leetcode.com/problemset/algorithms/>
- <https://www.hackerrank.com/domains/algorithms>
- <https://thinkster.io/tutorials/100-algorithms-challenge>
- <https://www.logicmojo.com/>
- <https://www.educative.io/>
- <https://www.geeksforgeeks.org/>

*Neither BITS Pilani nor Faculties vouch for these websites.  
It's your personal choice.*

## Research

- <https://arxiv.org/>
  - cs.DS
  - cs.CG
  - cs.IT
  - cs.LG
  - cs.AI
- Search for DSAD topics in journal publication portals, join forums/societies.
- <https://shodhganga.inflibnet.ac.in/> - Search for DSAD topics



*Until we cross paths again, Ciao!*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)





**BITS Pilani**

Pilani|Duba|Goa|Hyderabad

# Data Structures and Algorithms Design

## DSECLZG519

# Applications of Stack

---

- Stacks can be used for expression evaluation.
- Stacks can be used for Conversion from one form of expression to another.
- Stacks can be used to check parenthesis matching in an expression.
- Stack data structures are used in backtracking problems (recursion).

# Forms of Expression

---

Arithmetic expressions can be written in 3 different forms or notations :

- **Infix Notation** : Notation in which the operator symbol is placed in between its operands. Ex : A + B , C – D , A \* D etc.
- **Pre-fix Notation** : It is also called as Polish notation and refers to the notation in which the operator symbol is placed **before** its operands (*when operator precedes the operands*). Ex : +AB, -CD, \*AD etc.
- **Post-fix Notation** : It is also called as reverse polish notation and refers to the notation in which the operator symbol is placed **after** its operands (*when operator follows the operands*). Ex : AB+ , CD- , AD\*

# Forms of expression

- There's no real reason to put the operation between the variables or values.
- They can just as well precede or follow the operands.
- You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

| Infix                 | Prefix        | Postfix       |
|-----------------------|---------------|---------------|
| $a + b$               | + a b         | a b +         |
| $a + b * c$           | + a * b c     | a b c * +     |
| $(a + b) * (c - d)$   | * + a b - c d | a b + c d - * |
| $b * b - 4 * a * c$   |               |               |
| $x * x + y / (z - 1)$ |               |               |
| $40 - 3 * 5 + 1$      |               |               |

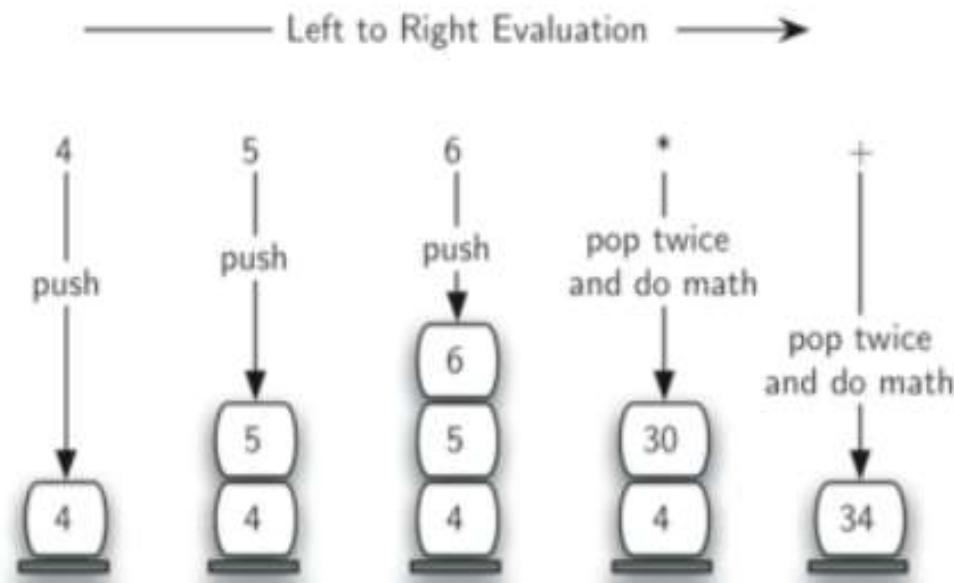
# Evaluation of Postfix expression

---

1. Create an empty stack
  2. Scan the token list from left to right.
    - i. If the token is an operand, push the value onto the Stack.
    - ii. If the token is an operator, \*, /, +, or -, it will need two operands. Pop the Stack twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the Stack.
  3. When the input expression has been completely processed, the result is on the stack. Pop the stack and return the value.
-

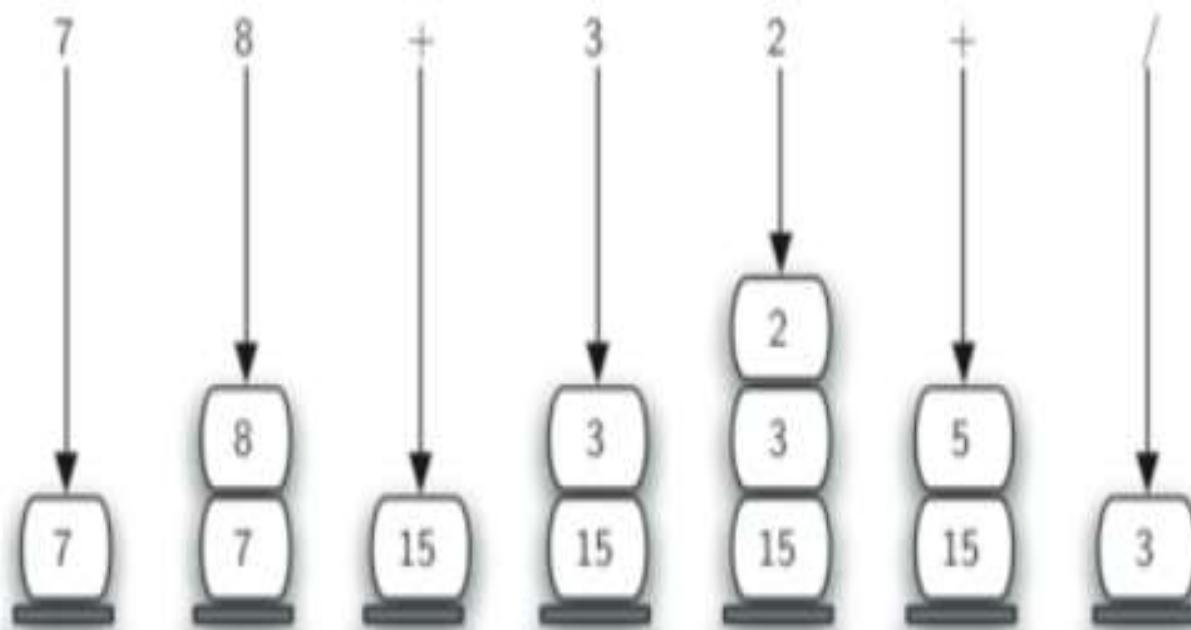
# Example 1

4 5 6 \* +



# Example 2

$$78 + 32 + /$$



# Example: Evaluation of Postfix expression using stack

Evaluate  $AB+C-BA+C^{\wedge}-$  where  $A=1, B=2$  and  $C=3$

→  $1\ 2\ +\ 3\ -\ 2\ 1\ +\ 3\ ^{\wedge}\ -$

| Symbol Encountered | Op 2 | Op1 | Value Op1 op Op2   | STACK |
|--------------------|------|-----|--------------------|-------|
| 1                  |      |     |                    | 1     |
| 2                  |      |     |                    | 1 2   |
| +                  | 2    | 1   | $1+2 = 3$          | 3     |
| 3                  |      |     |                    | 3 3   |
| -                  | 3    | 3   | $3 - 3 = 0$        | 0     |
| 2                  |      |     |                    | 0 2   |
| 1                  |      |     |                    | 0 2 1 |
| +                  | 1    | 2   | $2+1 = 3$          | 0 3   |
| 3                  |      |     |                    | 0 3 3 |
| $^{\wedge}$        | 3    | 3   | $3^{\wedge}3 = 27$ | 0 27  |
| -                  | 27   | 0   | $0 - 27 = -27$     | -27   |

Hence, the evaluation of this expression leads to the answer **-27**

# Forms of Expression



## Examples :

| # | Infix           | Prefix       | Postfix        |
|---|-----------------|--------------|----------------|
| 1 | $(A+B) * C$     | $*+ABC$      | $AB+C*$        |
| 2 | $A + ( B * C )$ | $+A*BC$      | $ABC*+$        |
| 3 | $A ^ B * C - D$ | $- * ^ ABCD$ | $AB ^ C * D -$ |
| 4 | $(A+B)/(C-D)$   | $/+AB-CD$    | $AB+CD-/$      |

- The computer usually evaluates an arithmetic expression in infix notation in 2 steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In both these steps, the stack is used !! Let us examine how stack is used

# Infix transformation to Postfix

---

Consider the expression  $A + B * C$ .

$A B C * +$  is the postfix equivalent.

The operands A, B, and C stay in their relative positions. It is only the operators that change position.

In the infix expression the first operator that appears from left to right is  $+$ . However, in the postfix expression,  $+$  is at the end since the next operator,  $*$ , has precedence over addition. The order of the operators in the original expression is reversed in the resulting postfix expression.

# Infix transformation to Postfix

---

As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet.

Also, the order of these saved operators may need to be reversed due to their precedence.

This is the case with the addition and the multiplication in this example. Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used. Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

---

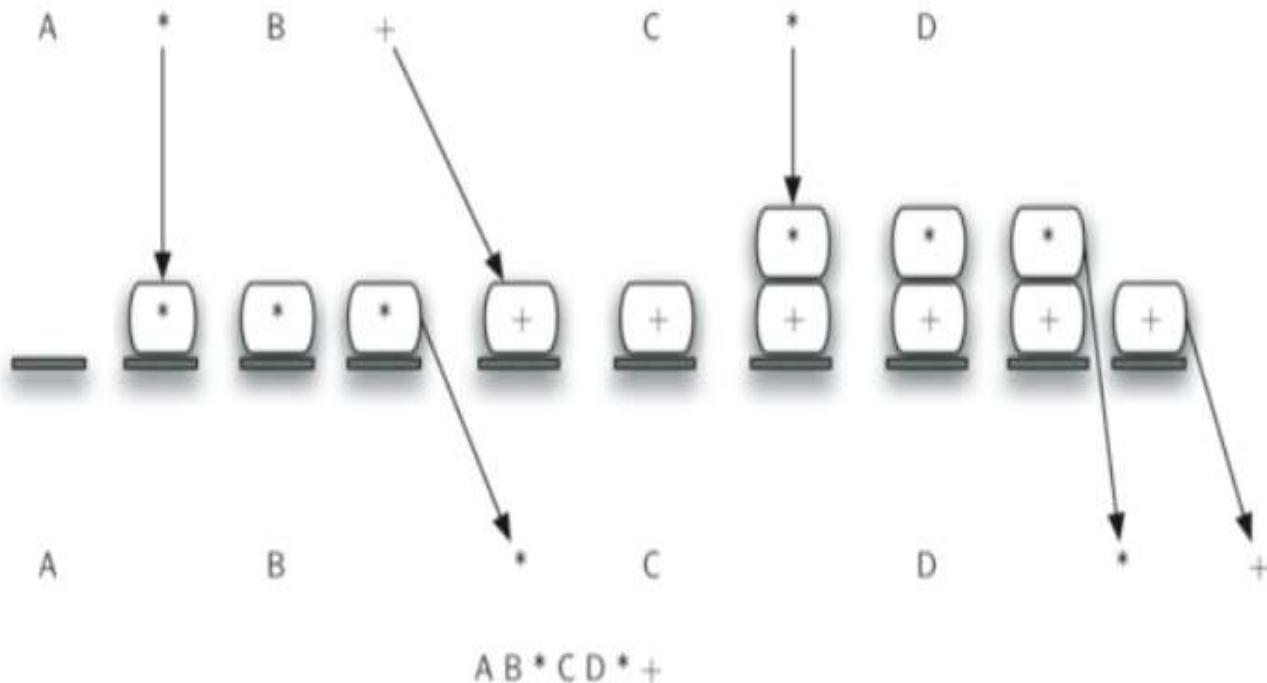
# Infix transformation to Postfix



1. Create an empty stack for keeping operators. Create an empty list for output.
2. Scan the token list from left to right.
  - i. If the token is an operand, append it to the end of the output list.
  - ii. If the token is a left parenthesis, push it on the stack.
  - iii. If the token is a right parenthesis, pop the stack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
  - iv. If the token is an operator, \*, /, +, or -, push it on the stack. However, first remove any operators already on the stack that have higher or equal precedence and append them to the output list.
3. When the input expression has been completely processed, check the stack. Any operators still on the stack can be removed and appended to the end of the output list.

# Example

$$A * B + C * D$$



# Example : Converting Infix to Postfix using Stack

Consider the infix expression Q to be  $A + B * C$

| # | Symbol Encountered | Postfix String P (Output) | STACK Contents |
|---|--------------------|---------------------------|----------------|
| 1 | A                  | A                         |                |
| 2 | +                  | A                         | +              |
| 3 | B                  | AB                        | +              |
| 4 | *                  | AB                        | +*             |
| 5 | C                  | ABC                       | +*             |
| 6 | -                  | ABC*+                     | Empty          |

Hence, the postfix expression P is  $ABC*+$

# Example : Converting Infix to Postfix using Stack

Consider the infix expression Q to be  $(A+B) * C$

| # | Symbol Encountered | Postfix String P<br>(Output) | STACK Contents |
|---|--------------------|------------------------------|----------------|
| 1 | (                  |                              | (              |
| 2 | A                  | A                            | (              |
| 3 | +                  | A                            | (+             |
| 4 | B                  | AB                           | (+             |
| 5 | )                  | AB+                          |                |
| 6 | *                  | AB+                          | *              |
| 7 | C                  | AB+C                         | *              |
| 8 | -                  | AB+C*                        | Empty          |

Hence, the postfix expression P is **AB+C\***

# Exercises

Evaluate the following postfix expressions :

- $ABC+*CBA-+*$  where A=1,B=2 and C=3
- $5\ 7\ 5\ -\ *\ 12\ 4\ *\ 24\ / \ 6\ +\ +$

After you solve it, verify your answer by putting the postfix expression in this tool  
<https://www.free-online-calculator-use.com/postfix-evaluator.html>

# Appendix Exercise

1) Convert the following infix expressions into postfix expressions using stack ( trace it as we did in class ) :

- $A + (B + C) * D$
- $B + C - D / E * F$
- $(A + B) + (C / D) * E$
- $A + B * (C + D - E) * F$

After you solve it, verify your answer by putting the infix expression in this tool :

<https://www.mathblog.dk/tools/infix-postfix-converter/>

# Amortized Analysis

- **Amortized analysis** is a method of analyzing the costs associated with a data structure that averages the worst operations out over time.
- Often, a data structure has one particularly costly operation, but it doesn't get performed very often.
- That data structure shouldn't be labeled a costly structure just because that one operation, that is seldom performed, is costly.
- Ex : Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or joining a course etc.. In other months, he or she saves money for the expensive month.

# Amortized Analysis – Real Life Example



Let's say you want to make a cake for the bake sale. Cake-making is pretty complex, but it's essentially two main steps:

- **Mix batter (fast).**
- **Bake in an oven (slow, and you can only fit one cake in at a time).**

Mixing the batter takes relatively little time when compared with baking. Afterwards, you reflect on the cake-making process. When deciding if it is slow, medium, or fast, you choose medium because you average the two operations—slow and fast—to get medium.

Now let's say you wanted to make 100 cakes. You have two options for how to bake 100 cakes. You can mix the batter for a single cake, bake it, and repeat. Or, you can mix the batter for all 100 cakes, then bake all of them, one after another. Are these methods slow, medium, or fast?

Amortized analysis tells us that these two methods should both be described as "medium", *even though you might have to bake 100 cakes sequentially*. Even though you might have to work through 100 slow operations in a row, they were preceded by 100 fast operations, so the average is still medium.



*Worst-case means that it is not possible to dream up a worse sequence of events. It doesn't make any sense, for instance, to skip the batter mixing operation and simply bake 100 cakes. That would be a slow baking process, but it doesn't make any sense, so it's not worth analyzing. The cake-baking process is a medium process because mixing cake batter and baking the cake have a logical ordering that cannot be reversed.*

# Amortized cost per operation

- The amortized cost per operation for a sequence of  $n$  operations is the total cost of the operations divided by  $n$ .
- For example, if we have 100 operations at cost 1, followed by one operation at cost 100, the amortized cost per operation is  $200/101 < 2$ .
- The reason for considering amortized cost is that we will be interested in data structures that occasionally can incur a large cost as they perform some kind of rebalancing or improvement of their internal state, but where such operations cannot occur too frequently
- In this case, amortized analysis can give a much tighter bound on the true cost of using the data structure than a standard worst-case-per-operation bound.

# Techniques used for amortized analysis

---

1. The aggregate method, where the total running time for a sequence of operations is analyzed.
  2. The accounting (or banker's) method, where an amortized cost per operation is used to "pay" for the actual work that must be done. The central idea is to save up small payments for each operation to use in times when a lot of work needs to be done.
  3. The potential method, which derives a function characterizing the "capacity" for a data structure to be transformed as each operation is performed. This potential either increases or decreases with each successive operational, but cannot be negative.
-

## Example #1: implementing a stack as an array

---

We have an array A, with a variable top that points to the top of the stack

To implement push(x), we just need to perform:

```
A[top] = x;  
top++;
```

To implement x=pop(), we just need to perform:

```
top--;  
x = A[top];
```

**Cost model:** Let's say that inserting into the array costs 1, taking an element out of the array costs 1, and the cost of resizing the array is the number of elements moved.

# Amortized Analysis-Dynamic table

---

The idea is to increase size of table whenever it becomes full.

Following are the steps to follow when **table becomes full**.

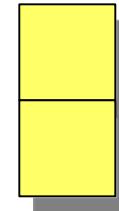
- Allocate memory for a larger table of size, typically twice the old table.
- Copy the contents of old table to new table.
- Free the old table.

If the table has space available, we simply insert new item in available space.

# Example of a dynamic table

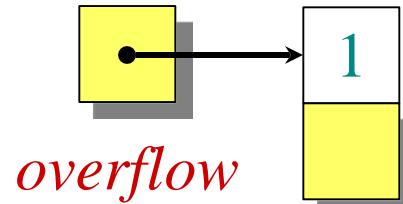
1. INSERT
2. INSERT

1  
*overflow*



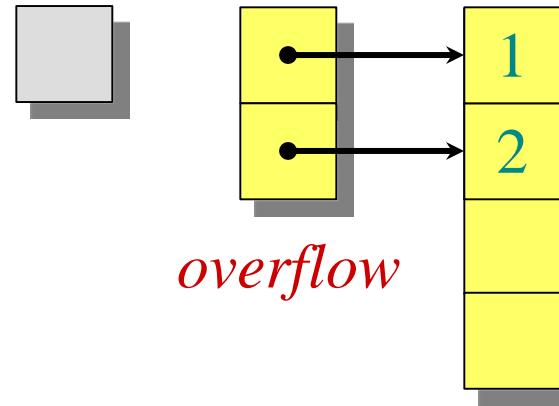
# Example of a dynamic table

1. INSERT
2. INSERT



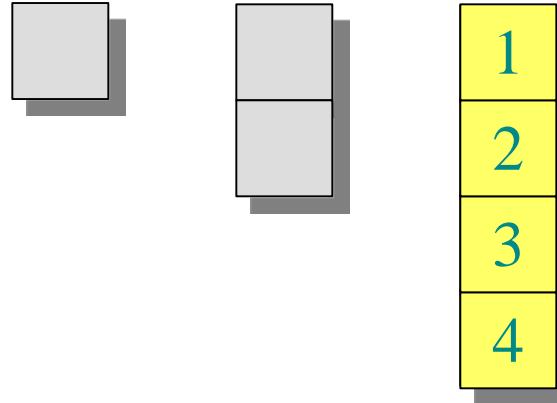
# Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



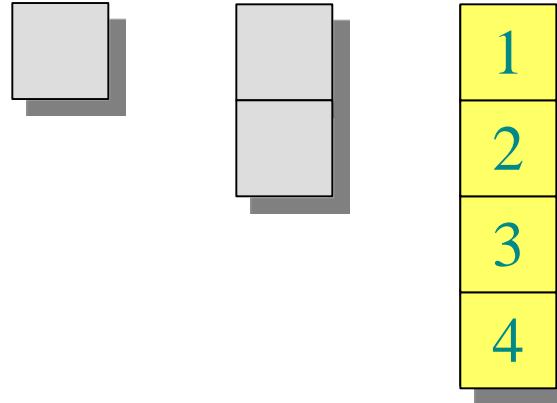
# Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT



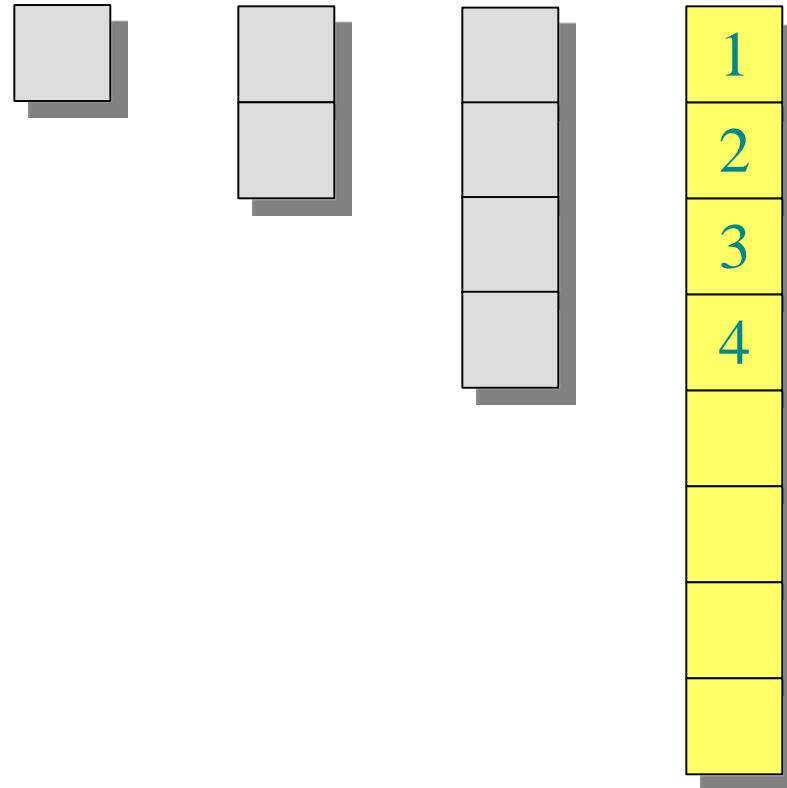
# Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT



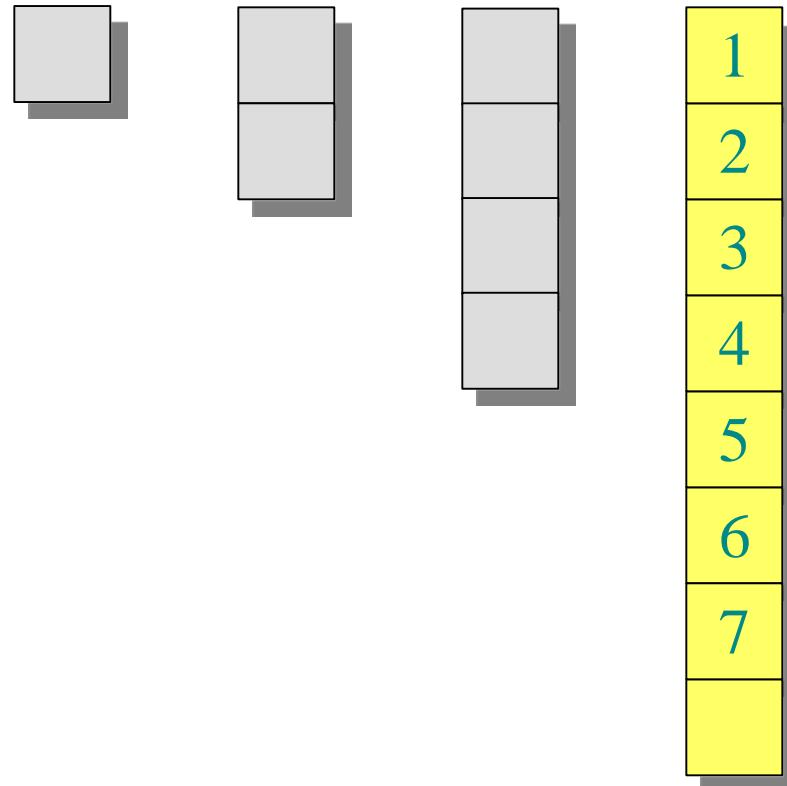
# Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



# Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



# Worst-case analysis

---

Consider a sequence of  $n$  insertions. The worst-case time to execute one insertion is  $\Theta(n)$ . Therefore, the worst-case time for  $n$  insertions is  $n \cdot \Theta(n) = \Theta(n^2)$ .

**WRONG!** In fact, the worst-case cost for  $n$  insertions is only  $\Theta(n) \ll \Theta(n^2)$ .

Let's see why.

---

# Tighter analysis

Let  $c_i$  = the cost of the  $i$ th insertion  
 $= \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$

| $i$      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 |
|----------|---|---|---|---|---|---|---|---|----|----|
| $size_i$ | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| $c_i$    | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9  | 1  |

# Tighter analysis

$$\begin{aligned}\text{Cost of } n \text{ insertions} &= \sum_{i=1}^n c_i \\ &\leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\ &\leq 3n \\ &= \Theta(n).\end{aligned}$$

Thus, the average cost of each dynamic-table operation is  $\Theta(n)/n = \Theta(1)$ .

# Types of amortized analysis

Three common amortization arguments:

- the *aggregate* method,
- the *accounting* method,
- the *potential* method.

We've just seen an aggregate analysis.

The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific *amortized cost* to be allocated to each operation.

# Accounting Method

---

- Charge  $i$ th operation a fictitious ***amortized cost***  $\hat{c}_i$ , where \$1 pays for 1 unit of work (*i.e.*, time).
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the ***bank*** for use by subsequent operations.
- The bank balance must not go negative ! We must ensure that

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i$$

for all  $n$ .

- Thus, the total amortized costs provide an upper bound on the total true costs.
-

# Accounting Analysis

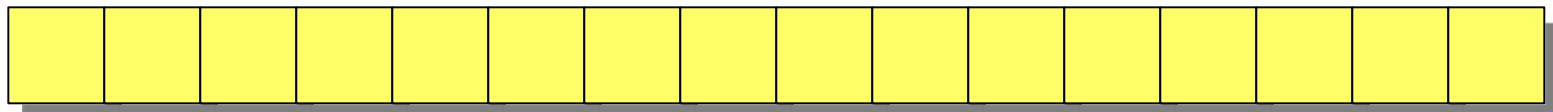
---

Charge an amortized cost of  $\hat{c}_i = \$3$  for the  $i$ th insertion.

- $\$1$  pays for the immediate insertion.
- $\$2$  is stored for later table doubling.

When the table doubles,  $\$1$  pays to move a recent item, and  $\$1$  pays to move an old item.

## Example:



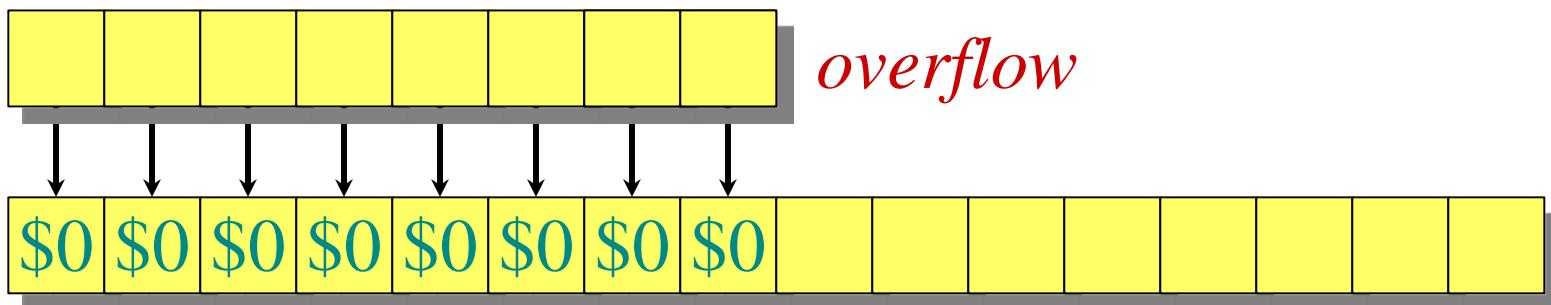
# Accounting Analysis

Charge an amortized cost of  $\hat{c}_i = \$3$  for the  $i$ th insertion.

- $\$1$  pays for the immediate insertion.
- $\$2$  is stored for later table doubling.

When the table doubles,  $\$1$  pays to move a recent item, and  $\$1$  pays to move an old item.

## Example:



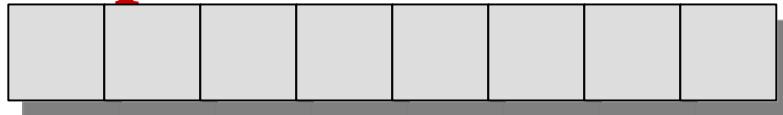
# Accounting Analysis

Charge an amortized cost of  $\hat{c}_i = \$3$  for the  $i$ th insertion.

- $\$1$  pays for the immediate insertion.
- $\$2$  is stored for later table doubling.

When the table doubles,  $\$1$  pays to move a recent item, and  $\$1$  pays to move an old item.

## Example:



|     |     |     |     |     |     |     |     |     |     |     |  |  |  |  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|--|--|--|
| \$0 | \$0 | \$0 | \$0 | \$0 | \$0 | \$0 | \$0 | \$2 | \$2 | \$2 |  |  |  |  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|--|--|--|

# Accounting Analysis

**Key invariant:** Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

|             |    |   |   |   |   |   |   |   |    |    |
|-------------|----|---|---|---|---|---|---|---|----|----|
| $i$         | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 |
| $size_i$    | 1  | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| $c_i$       | 1  | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9  | 1  |
| $\hat{c}_i$ | 2* | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3  | 3  |
| $bank_i$    | 1  | 2 | 2 | 4 | 2 | 4 | 6 | 8 | 2  | 4  |

The first operation costs only \$2, not \$3.



# Amortized Analysis

---

- Amortized costs can provide a clean abstraction of data-structure performance.
  - Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest.
  - Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.
-



**BITS** Pilani

# Data Structures and Algorithm Design



# Agenda – session 9

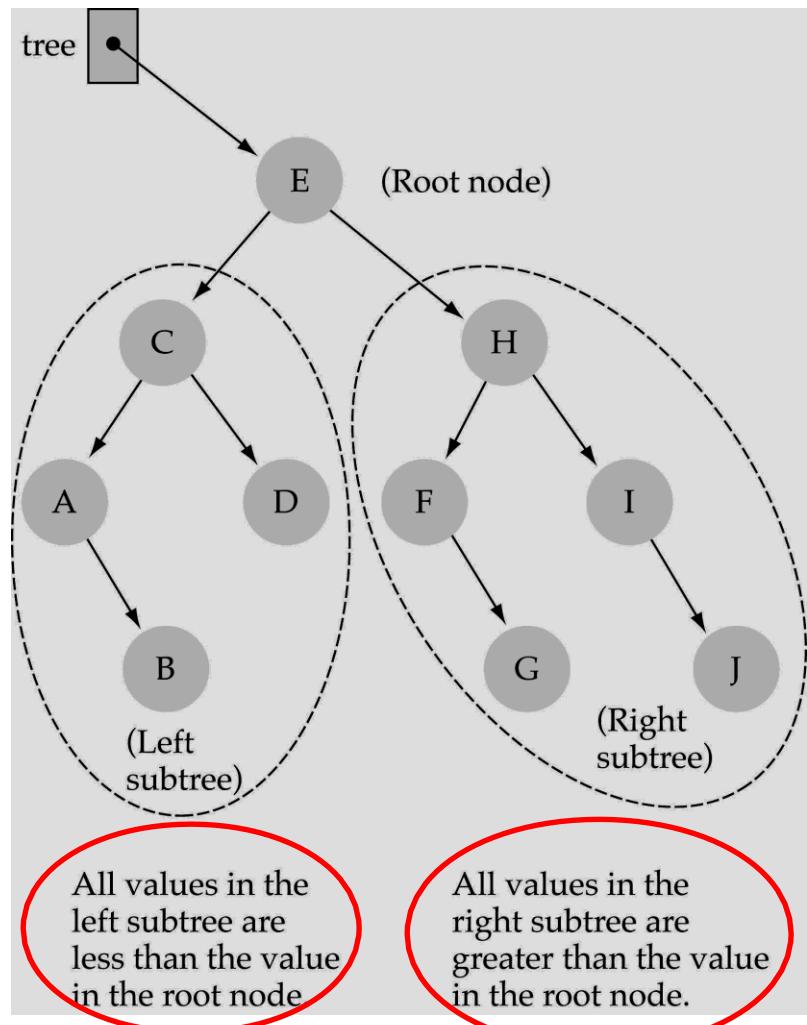
---

- What is BST
- BST-Search(Algorithm -analysis)
- BST-Insertion(Algorithm -analysis)
- BST-Deletion(Algorithm -analysis)
- Balanced Tree
- Rank,Find k-th smallest element in BST
- Range query
- Range Search – Example
- BST-Few applications
- k-d Trees

# Binary Search Trees

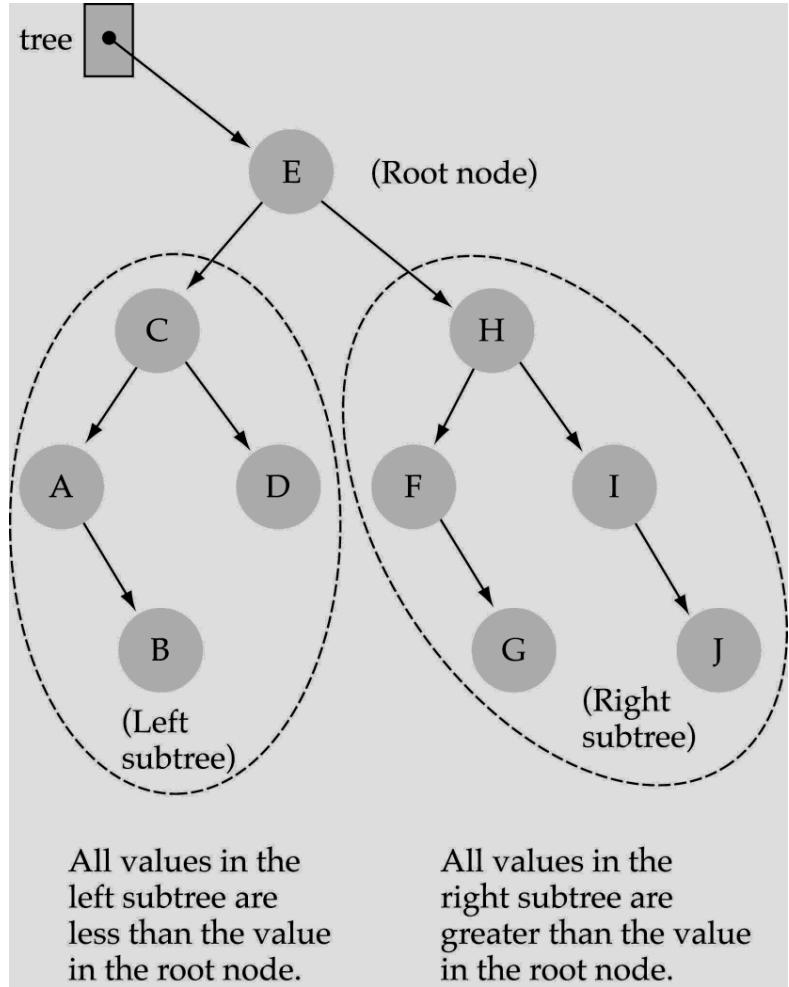
## ⊕ Binary Search Tree Property:

The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



# Binary Search Trees

- ♣ In a BST, the value stored at the root of a subtree is *greater* than any value in its left subtree and *less* than any value in its right subtree!



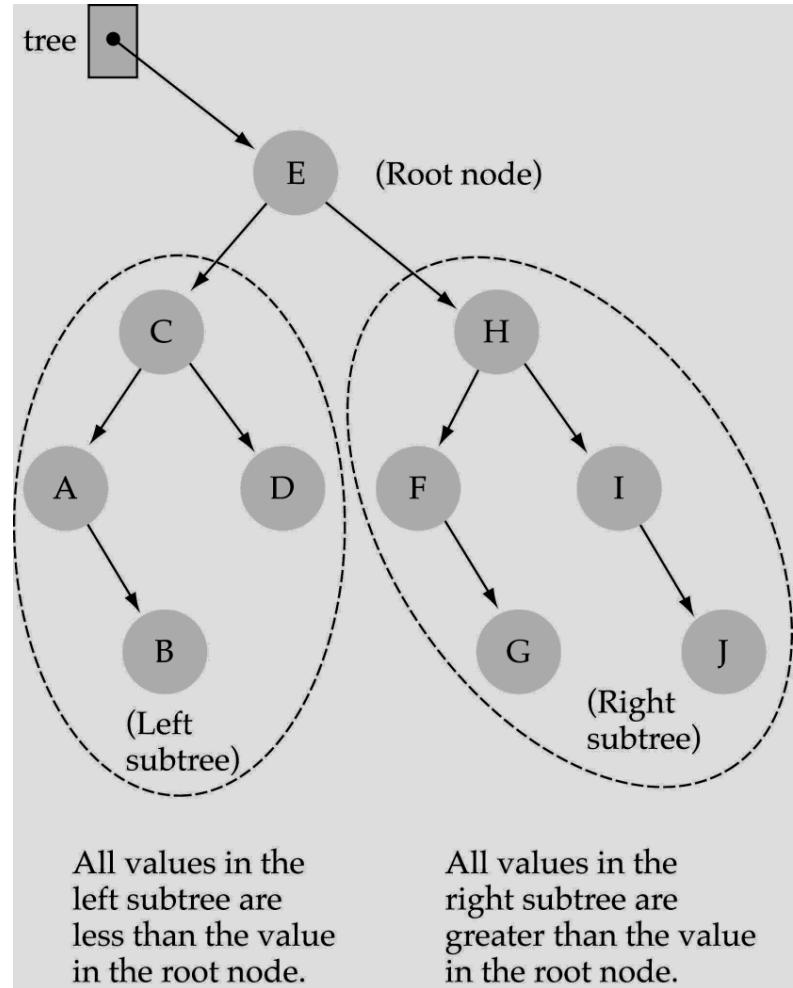
# Binary Search Trees

Where is the smallest element?

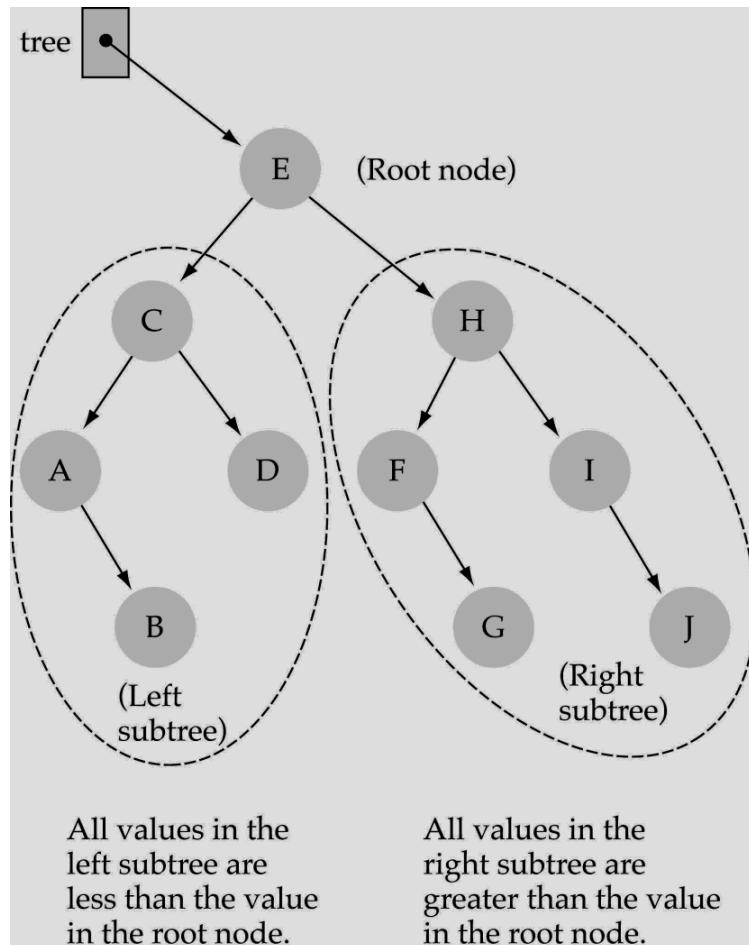
Ans: leftmost element

Where is the largest element?

Ans: rightmost element

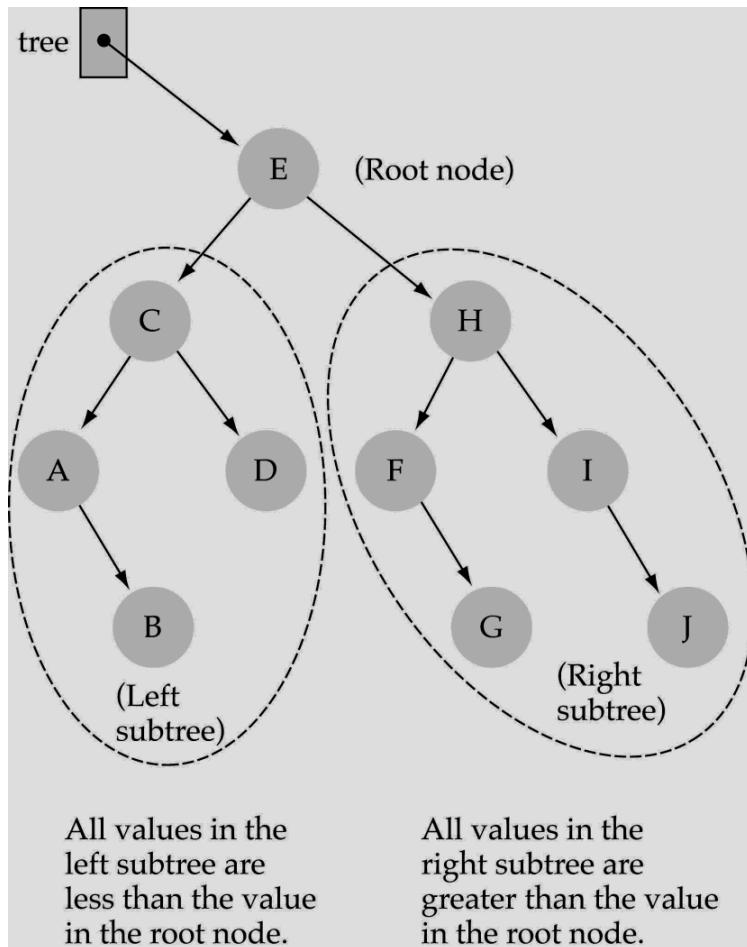


# Howto search a binary search tree?



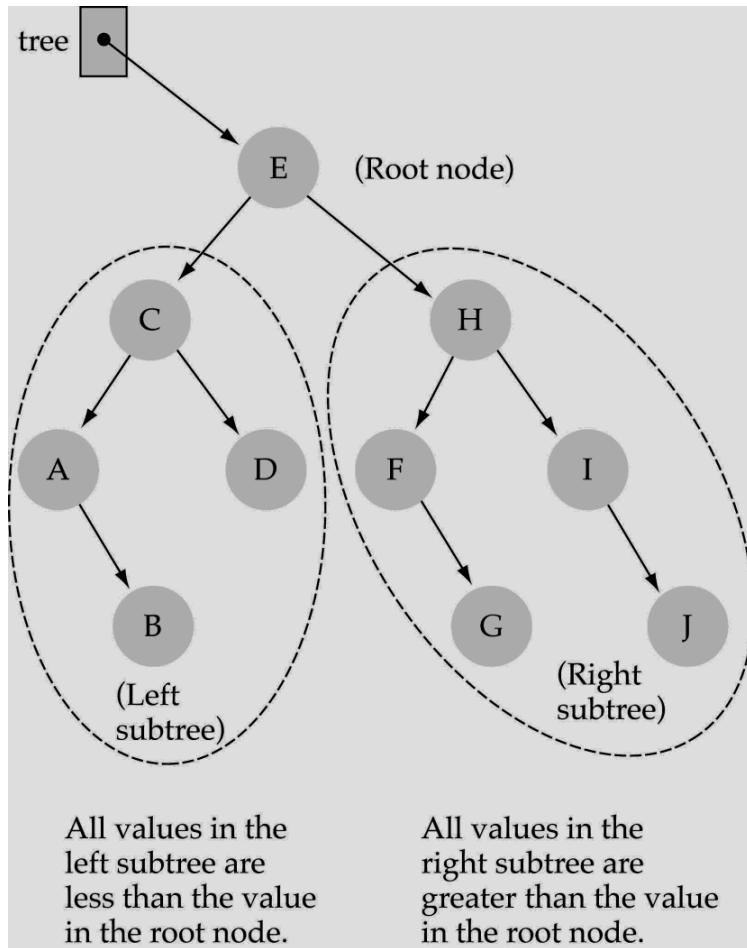
1. Start at the root
2. Compare the value of the item you are searching for with the value stored at the root
3. If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*

# Howto search a binary search tree?



4. If it is less than the value stored at the root, then search the left subtree
5. If it is greater than the value stored at the root, then search the right subtree
6. Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

# How to search a binary search tree?



Is this better than searching  
a linked list?

**Yes !!** ◊ ◊ **O(logN)**

# Difference between BT and BST

- ♣ A binary tree is simply a tree in which each node can have at most two children.
- ♣ A binary search tree is a binary tree in which the nodes are assigned values, with the following restrictions :
  1. No duplicate values.
  2. The left subtree of a node can only have values less than the node
  3. The right subtree of a node can only have values greater than the node and recursively defined
  4. The left subtree of a node is a binary search tree.
  5. The right subtree of a node is a binary search tree.

# Binary Tree Search Algorithm

- ♣ Let  $x$  be a node in a binary search tree and  $k$  is the value, we are supposed to search.
- ♣ Then according to the binary search tree property we know that: if  $y$  is a node in the left subtree of  $x$ , then  $y.key \leq x.key$ . If  $y$  is a node in the right subtree of  $x$ , then  $y.key \geq x.key$ . ( $x.key$  denotes the value at node  $x$ )
- ♣ To search the location of given data  $k$ , the binary search tree algorithm begins its search at the root and traces the path downward in the tree.
- ♣ For each node  $x$  it compares the value  $k$  with  $x.key$ . If the values are equal then the search terminates and  $x$  is the desired node.

# Binary Tree Search Algorithm

- ♣ If  $k$  is smaller than  $x.key$ , then the search continues in the left subtree of  $x$ , since the binary search tree property implies that  $k$  could not be in the right subtree.
- ♣ Symmetrically, if  $k$  is larger than  $x.key$ , then the search continues in the right subtree.
- ♣ The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is  $O(h)$ , where  $h$  is the height of the tree.

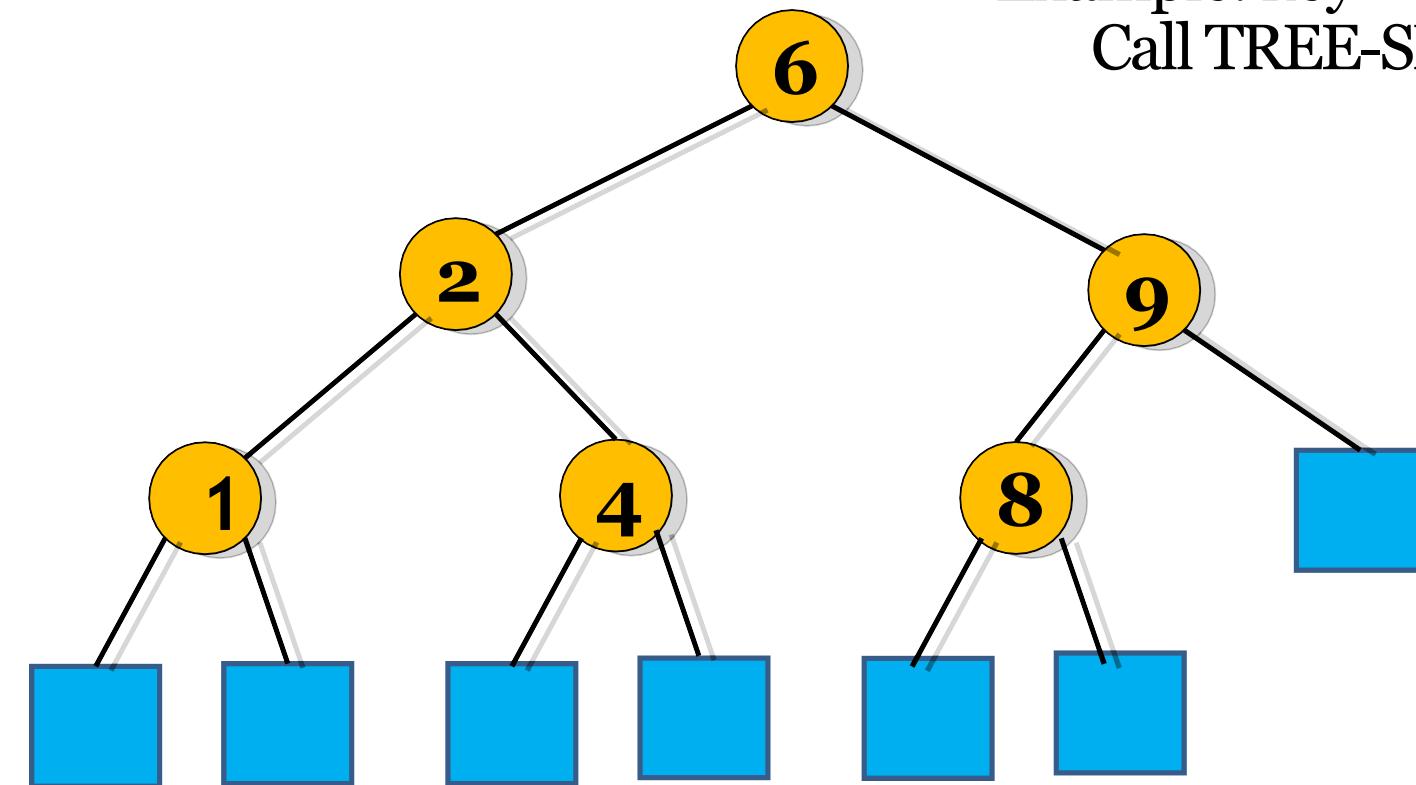
# Binary Tree Search Algorithm

TREE-SEARCH( $x, k$ )

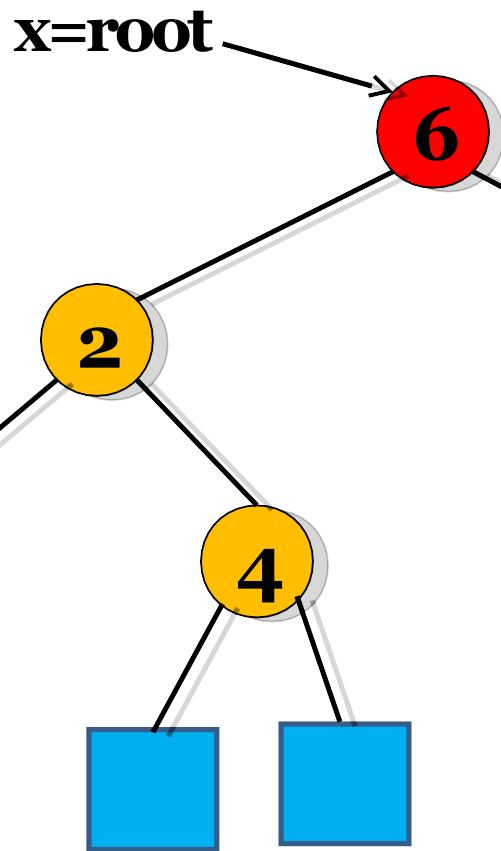
1. If  $x == \text{NIL}$  or  $k == x.\text{key}$
2. return  $x$
3. If  $k < x.\text{key}$
4.     return TREE-SEARCH( $x.\text{left}, k$ )
5. else return TREE-SEARCH( $x.\text{right}, k$ )

# Binary Tree Search Algorithm

Example: key =4 then find(4)  
Call TREE-SEARCH(x,k)



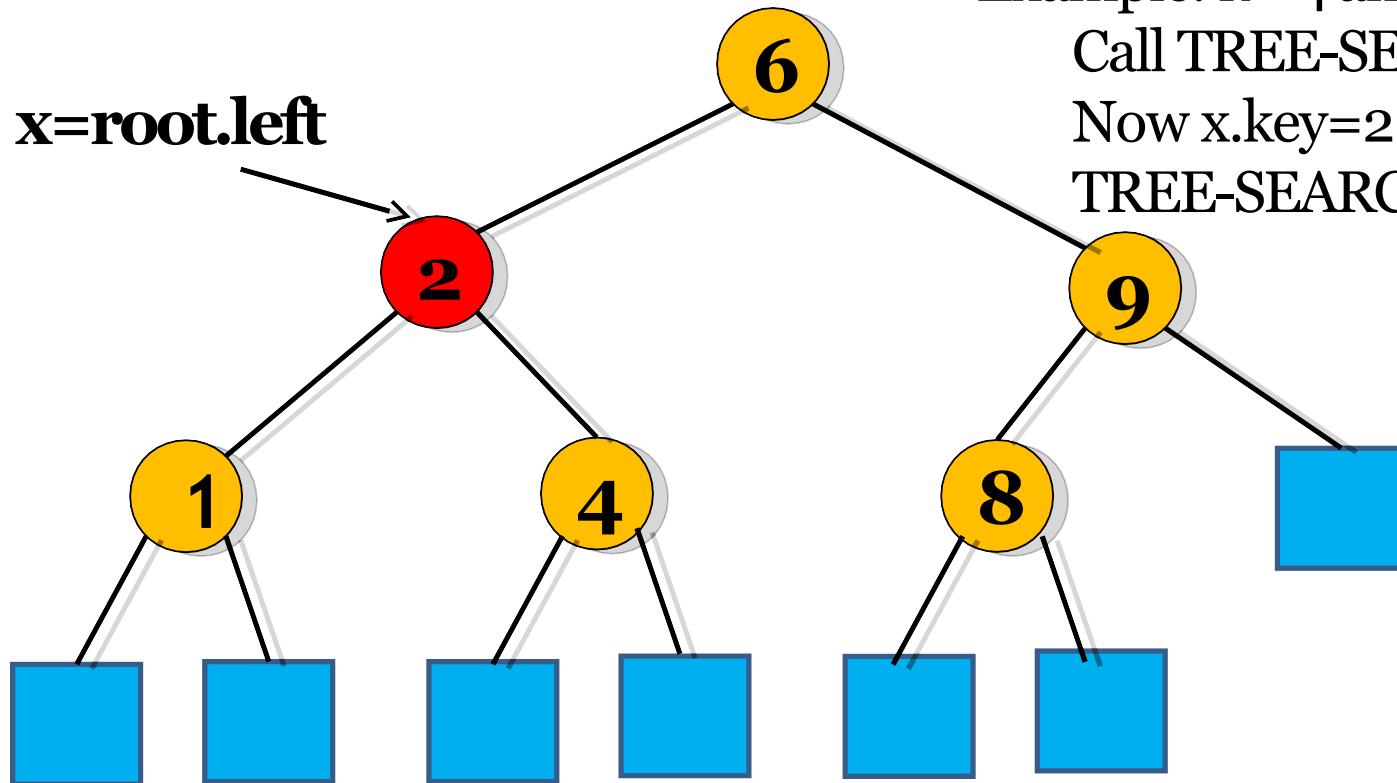
# Binary Tree Search Algorithm



**x=root**

Example:  $k = 4$  and  $x=\text{root}$   
Call TREE-SEARCH(root,4)  
Now  $x.\text{key}=6$  then  $k < x.\text{key}$   
So call TREE-SEARCH(x.left,k)

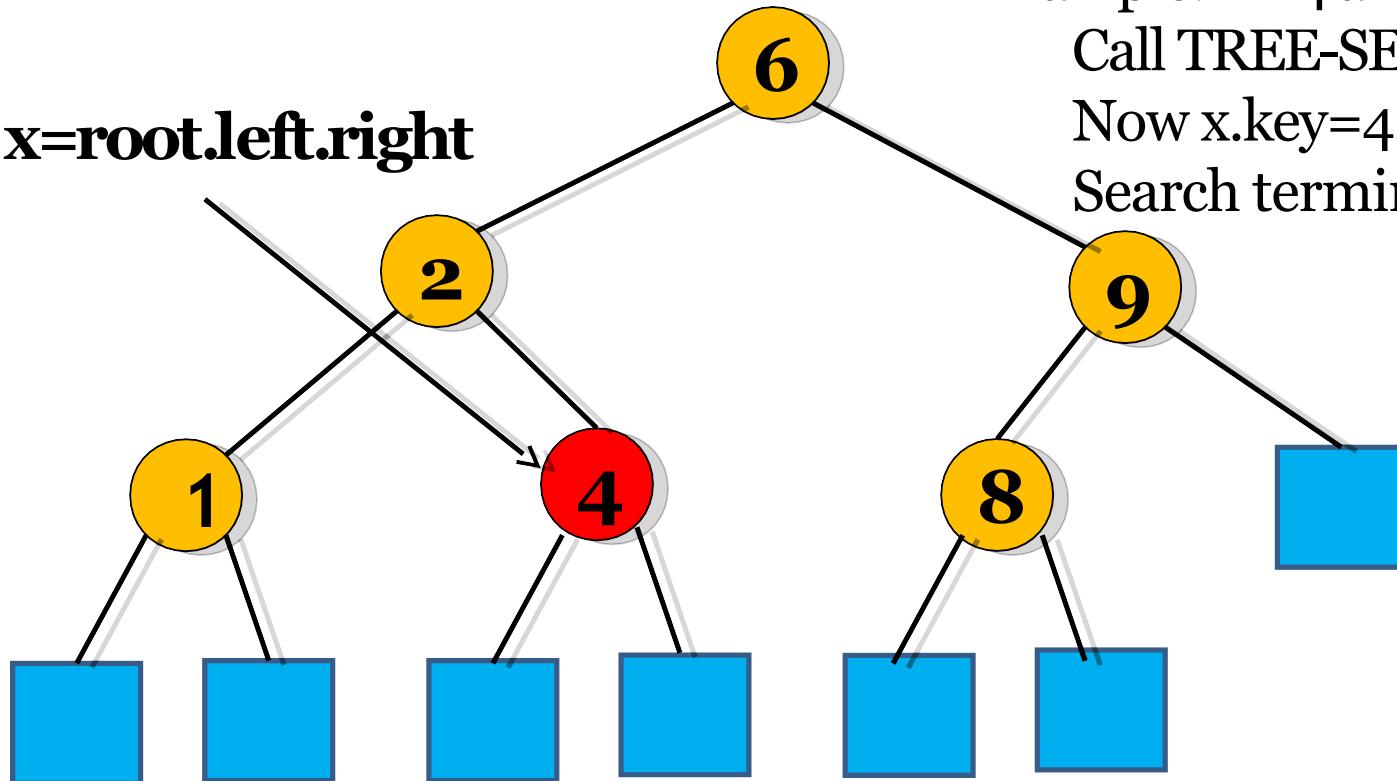
# Binary Tree Search Algorithm



Example:  $k = 4$  and  $x = \text{root.left}$   
Call TREE-SEARCH( $x, 4$ )  
Now  $x.\text{key} = 2$  then  $k > x.\text{key}$   
TREE-SEARCH( $x.\text{right}, k$ )

# Binary Tree Search Algorithm

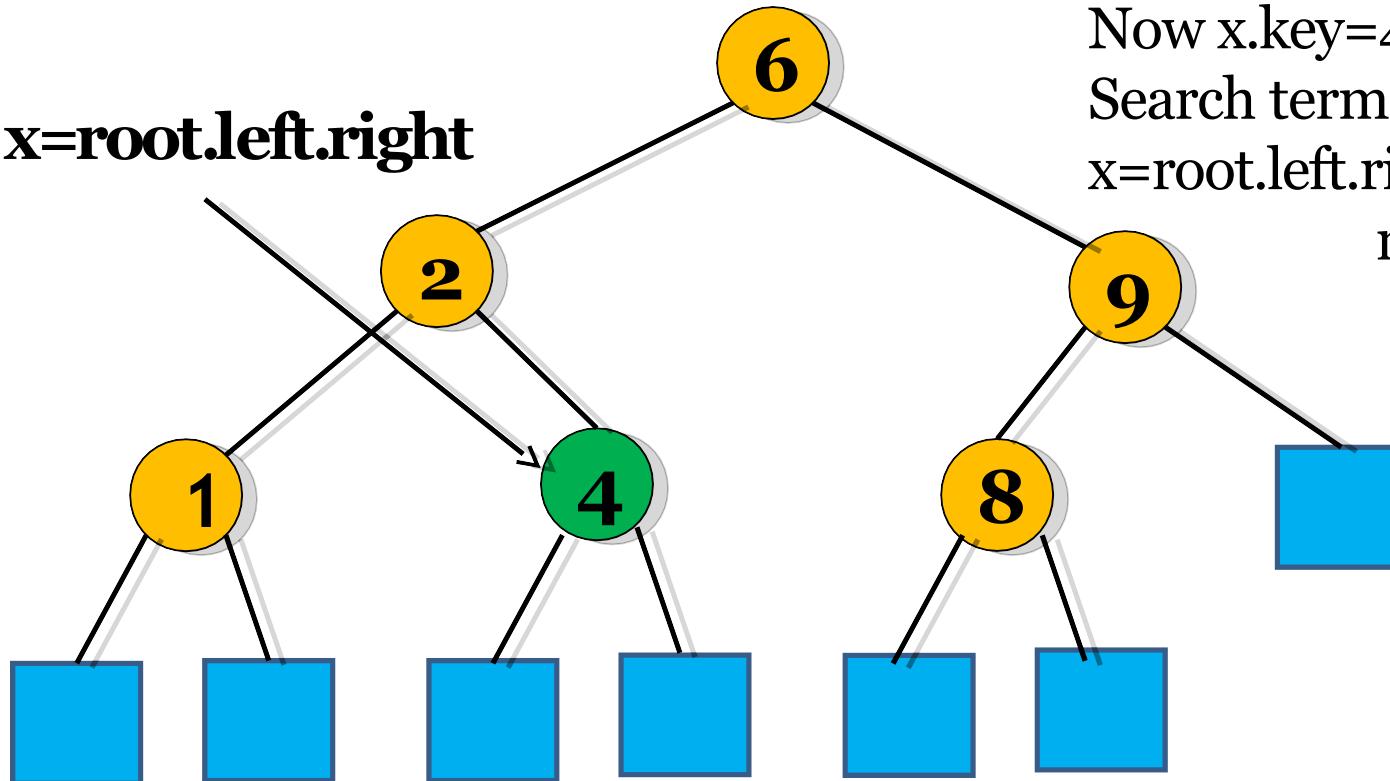
**x=root.left.right**



Example:  $k = 4$  and  $x = \text{root.left.right}$   
Call TREE-SEARCH( $x, 4$ )  
Now  $x.\text{key} = 4$  then  $k = x.\text{key}$   
Search terminates

# Binary Tree Search Algorithm

**x=root.left.right**



Example:  $k = 4$  and  $x = \text{root.left.right}$   
Now  $x.\text{key} = 4$  then  $k = x.\text{key}$   
Search terminates and  
 $x = \text{root.left.right}$  is the desired  
node or location

# BST – algorithm

**if** the tree is empty  
    return NULL

**else if** the key value in the node(root) equals the target  
    return the node value

**else if** the key value in the node is greater than the target  
    return the result of searching the left subtree

**else if** the key value in the node is smaller than the target  
    return the result of searching the right subtree

# Minimum Key or Element

- ♣ We can always find an element in a binary search tree whose key is minimum by following the left children from the root until we encounter a NIL.
- ♣ Otherwise if a node  $x$  has no left subtree then the value  $x.key$  contained in root  $x$  is the minimum key or element.  
The procedure for finding the minimum key:
  - ♣ **TREE-MINIMUM( $x$ )**
  - 1. while  $x.left \neq \text{NIL}$
  - 2.        $x = x.left$
  - 3. return  $x$

# Maximum Key or Element

- ♣ We can always find an element in a binary search tree whose key is maximum by following the right children from the root until we encounter a NIL.
- ♣ Otherwise if a node  $x$  has no right subtree then the value  $x.key$  contained in root  $x$  is the maximum key or element.  
The procedure for finding the maximum key:
  - ♣ **TREE-MAXIMUM( $x$ )**
  - 1. while  $x.right \neq \text{NIL}$
  - 2.        $x = x.right$
  - 3. return  $x$

# Insert a value into the BST

- ♣ To insert a new value  $v$  into a binary search tree  $T$ , we use the procedure TREE-INSERT.
- ♣ The procedure takes a node  $z$  for which  $z.key=v$  , $z.left=NIL$  and  $z.right=NIL$ .
- ♣ It modifies  $T$  and some of the attributes of  $z$  in such a way that it inserts  $z$  into an appropriate position in the tree

# Insert a value into the BST

- ♣ Suppose  $v$  is the value we want to insert and  $z$  is the node (New or NIL) we are supposed to find to insert the value  $v$ .  
 $z.p$  denotes the parent of  $z$ .
- ♣  $X$  is a pointer that traces a simple path downward the tree and  $y$  is the trailing pointer as the parent of  $x$ .  $T.root$  denote the root of the tree.
- ♣ Now the intention is to find a new or NIL node that will satisfy the BST property after placing the value  $v$ . The procedure first consider  $x$  as the root of the tree thus the parent of the root  $y=NIL$ .
- ♣ In steps 3 to 7 the procedure causes the two pointer  $y$  and  $x$  to move down the tree, going left or right depending on the comparison of  $z.key$  with  $x.key$ , until  $x$  becomes NIL.

# Insert a value into the BST

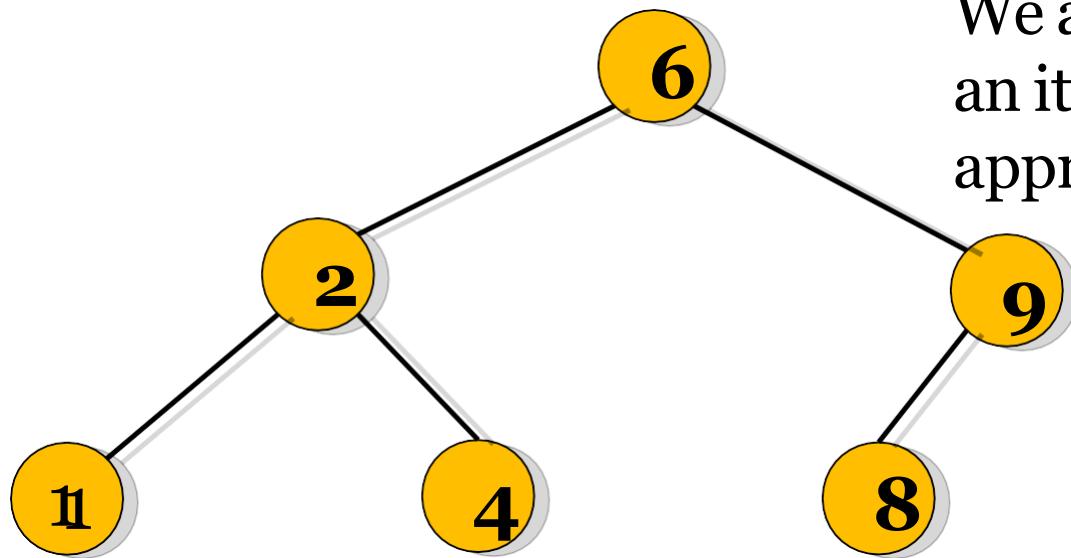
- ♣ Now this NIL occupies the position z, where we wish to place the input item.
- ♣ At this time we need y the parent of the desired node. This is why at step four we always storing the parent of current node x while moving downward. At the end of step 7 (in step 8) we make this node the parent of z (z.p).
- ♣ From steps 9 to 11:
  - ♣ Now **if** tree is empty ( $y==\text{NIL}$ ) then *create a root* node with the new key( $T.\text{root}=z$ )
  - ♣ If the value v is less than the value of the parent( $z.\text{key} < y.\text{key}$ ) then make it as the left-child of the parent( $y.\text{left}=z$ )
  - ♣ If the value v is greater than the value of the parent( $z.\text{key} > y.\text{key}$ ) then make it as the right-child of the parent( $y.\text{right}=z$ )

# Insert a value into the BST

TREE-INSERT( $T, z$ )

1.  $y = \text{NIL}$
2.  $x = T.\text{root}$
3. While  $x \neq \text{NIL}$ 
  4.      $y = x$
  5.     if  $z.\text{key} < x.\text{key}$
  6.          $x = x.\text{left}$
  7.         else  $x = x.\text{right}$
8.  $z.p = y$
9. if  $y == \text{NIL}$
10.  $T.\text{root} = z$
11. elseif  $z.\text{key} < y.\text{key}$
12.          $y.\text{left} = z$
13.         else  $y.\text{right} = z$

# BST Insertion Algorithm

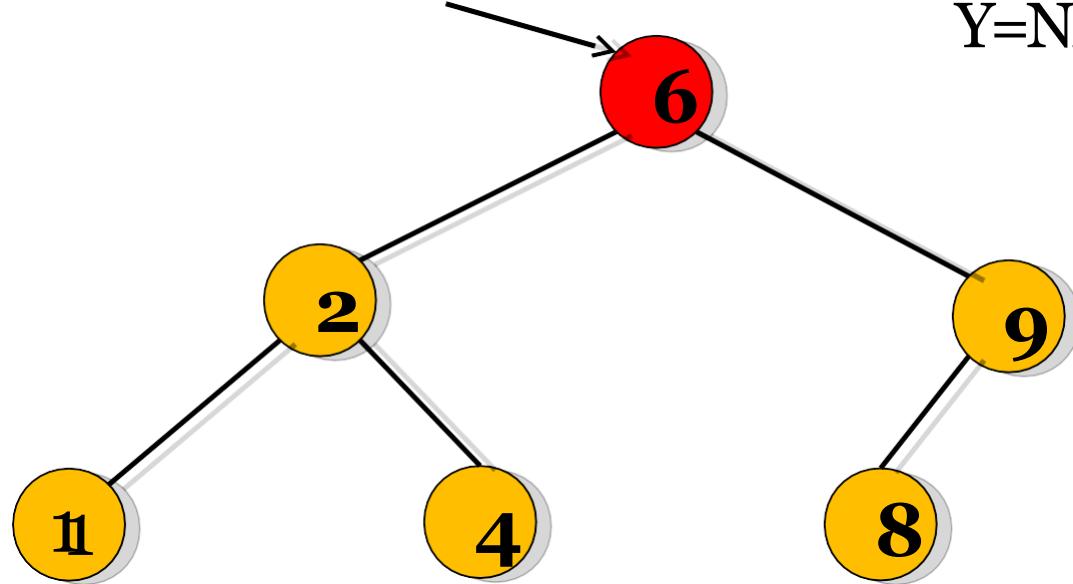


We are supposed to insert an item value 5 and find an appropriate node z for it

# BST Insertion Algorithm

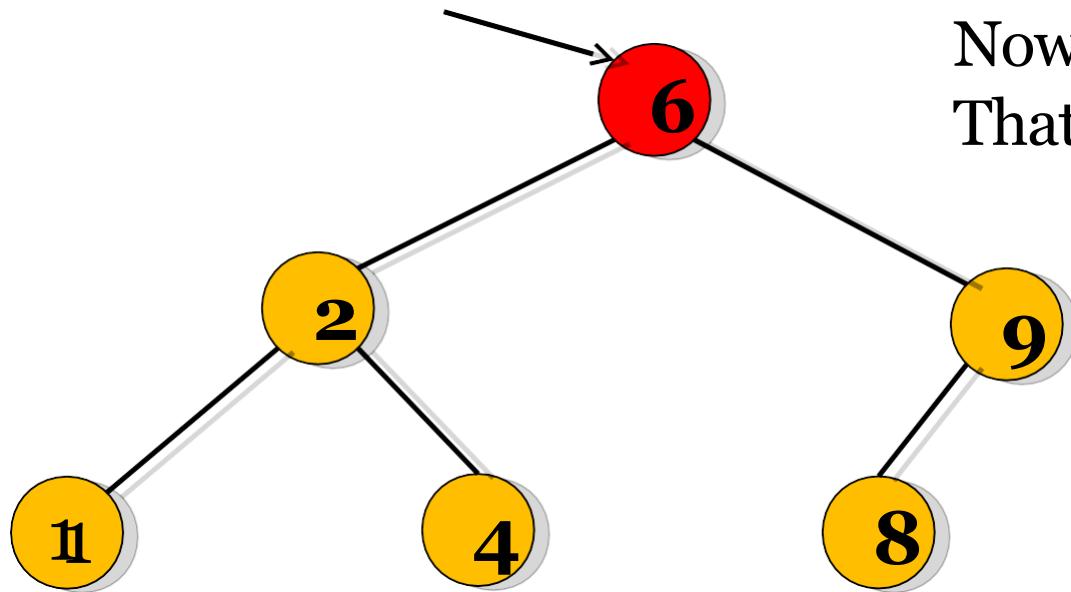
$x = T.root$

$Y = \text{NIL}$  and  $x = T.root$



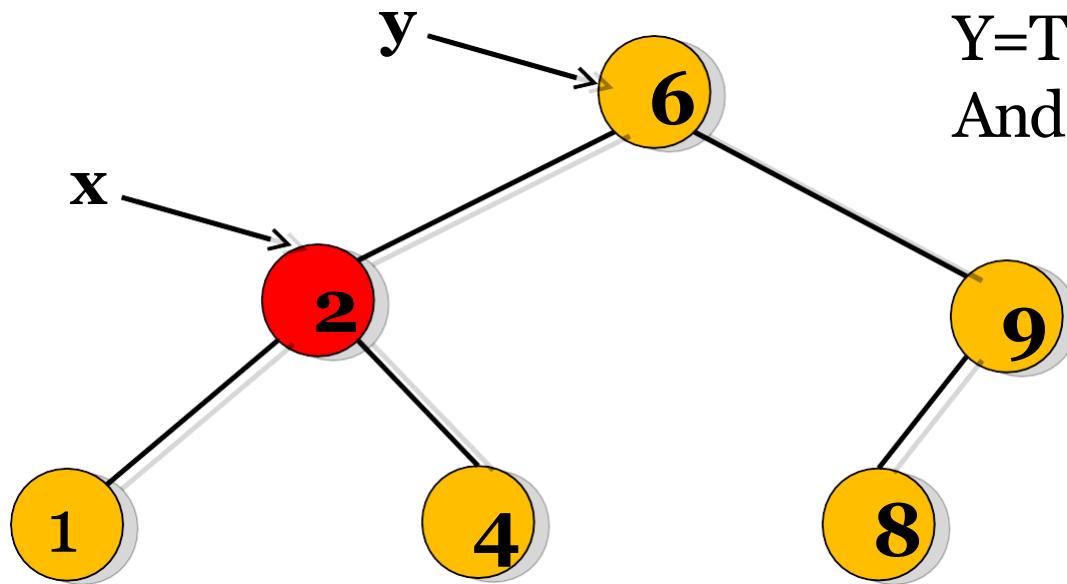
# BST Insertion Algorithm

$x=T.root$



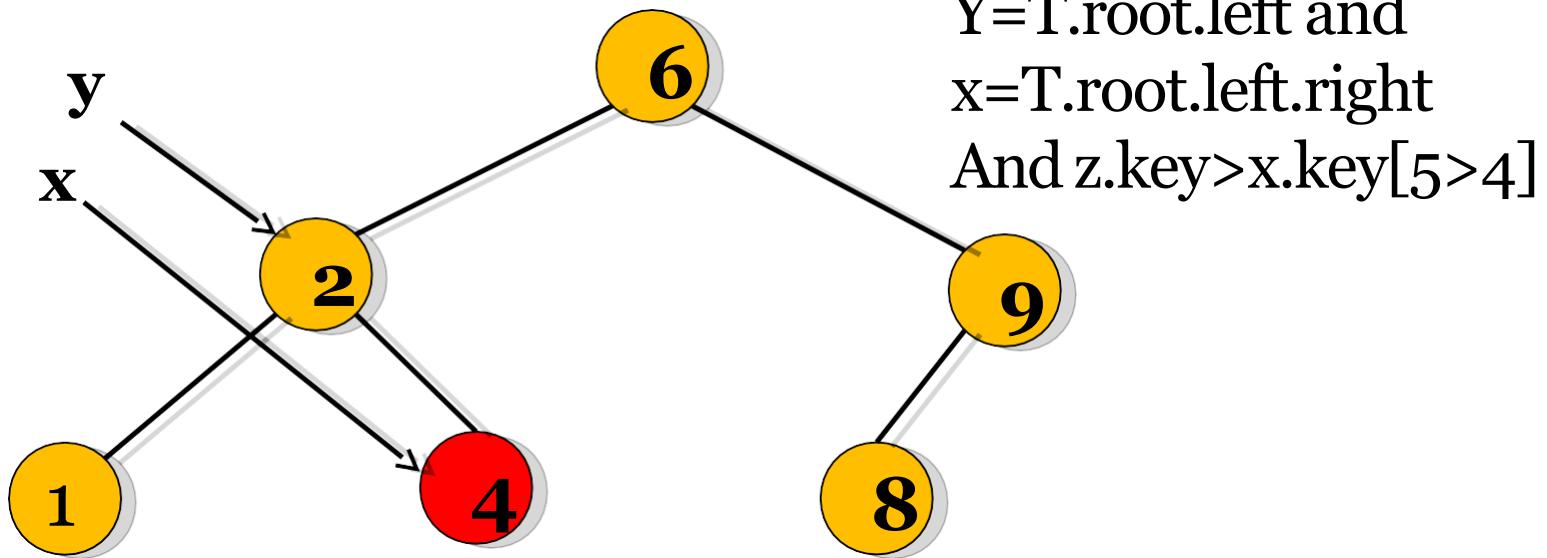
Now  $x \neq \text{NIL}$  and  $z.\text{key} < x.\text{key}$   
That is  $[5 < 6]$

# BST Insertion Algorithm

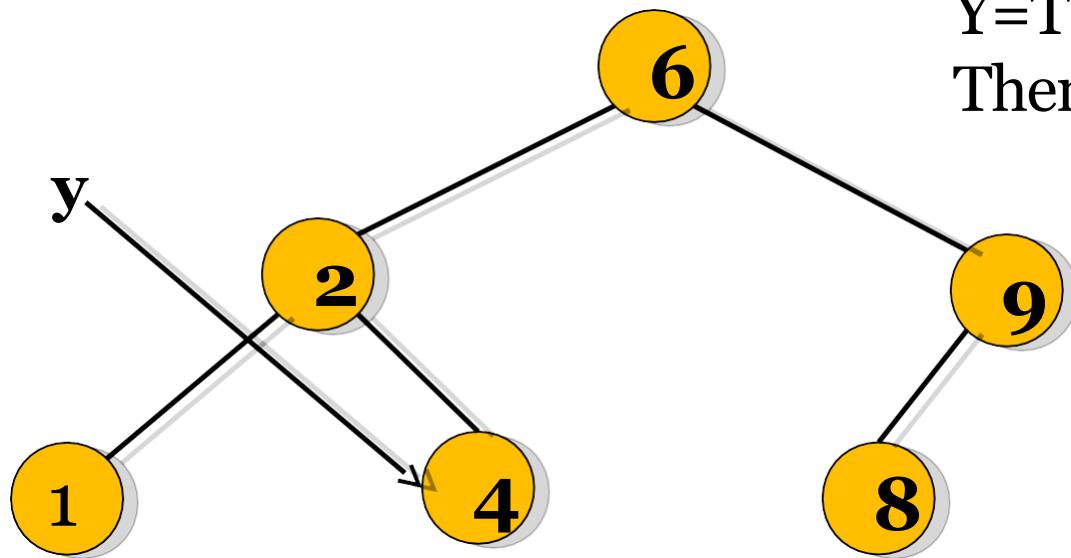


$Y = T.root$  and  $x = T.root.left$   
And  $z.key > x.key [5 > 2]$

# BST Insertion Algorithm

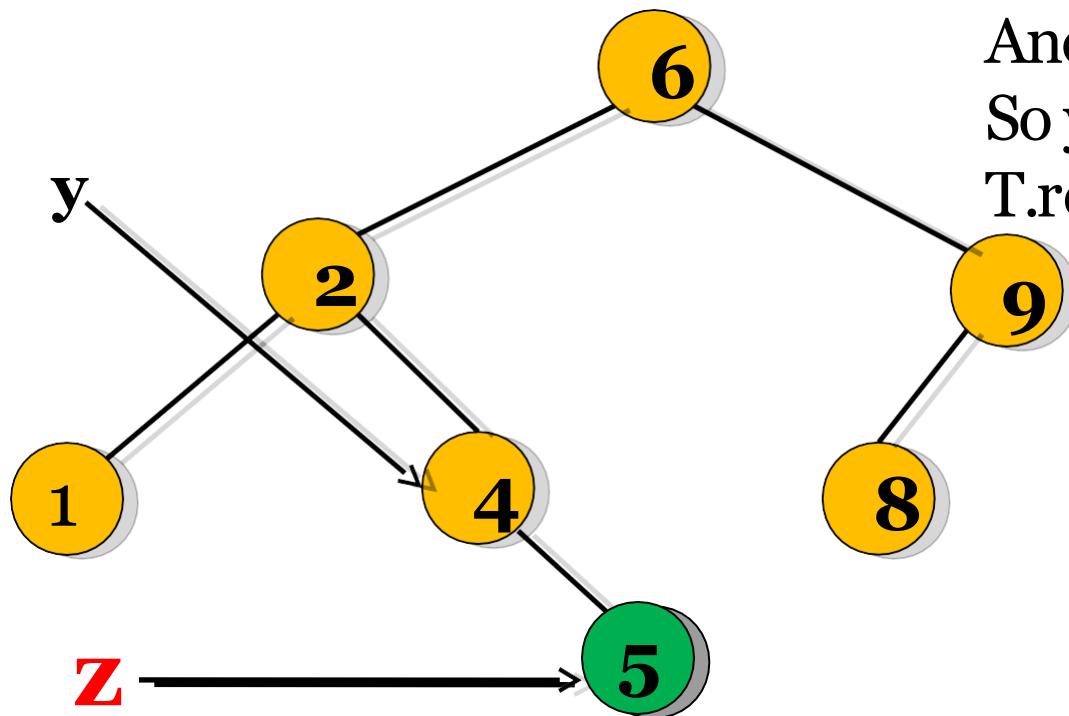


# BST Insertion Algorithm



$Y = T.root.left.right$  and  $x = \text{NIL}$   
Then  $z.p = T.root.left.right$

# BST Insertion Algorithm



$Y = T.root.left.right$  [ $y \neq \text{NIL}$ ]  
And  $z.key > y.key$  [ $5 > 4$ ]  
So  $y.right = z$  that is  
 $T.root.left.right.right = z$

# Insertion in BST - Algorithm

**if** tree is empty

*create a root node with the new key*

**else**

*compare key with the top node*

**if key = node key**

    replace the node with the new value

**else if key > node key**

*compare key with the right subtree:*

**if** subtree is empty create a leaf node

**else add key** in right subtree

**else key < node key**

*compare key with the left subtree:*

**if** the subtree is empty create a leaf node

**else add key** to the left subtree

# Delete a value from the BST

- ♣ Removing a node from a BST is a bit more complex, since we do not want to create any "holes" in the tree. The intention is to **remove** the specified item from the BST and **adjusts** the tree
- ♣ The binary search algorithm is used to locate the target item: **starting at the root** it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)
- ♣ If the node has one child then the child is spliced to the parent of the node. If the node has two children then its successor has no left child; copy the successor into the node and delete the successor instead **TREE-DELETE (T, z)** removes the node pointed to by z from the tree T. IT returns a pointer to the node removed so that the node can be put on a free-node list
- ♣ The overall strategy for deleting a node or item from a binary search tree can be described through some cases.

# Delete a value from the BST

## Experimenting the cases:

- ♣ if the tree is empty return false
- ♣ else Attempt to locate the node containing the target using the binary search algorithm:
  - if the target is not found return false
  - else the target is found, then remove its node. Now while removing the node four cases may happen

# Delete a value from the BST

**Case 1:** if the node has 2 empty subtrees

- replace the link in the parent with null

**Case 2:** if the node has a left and a right subtree

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

**Case 3:** if the node has no left child

- link the parent of the node to the right (non-empty) subtree

**Case 4:** if the node has no right child

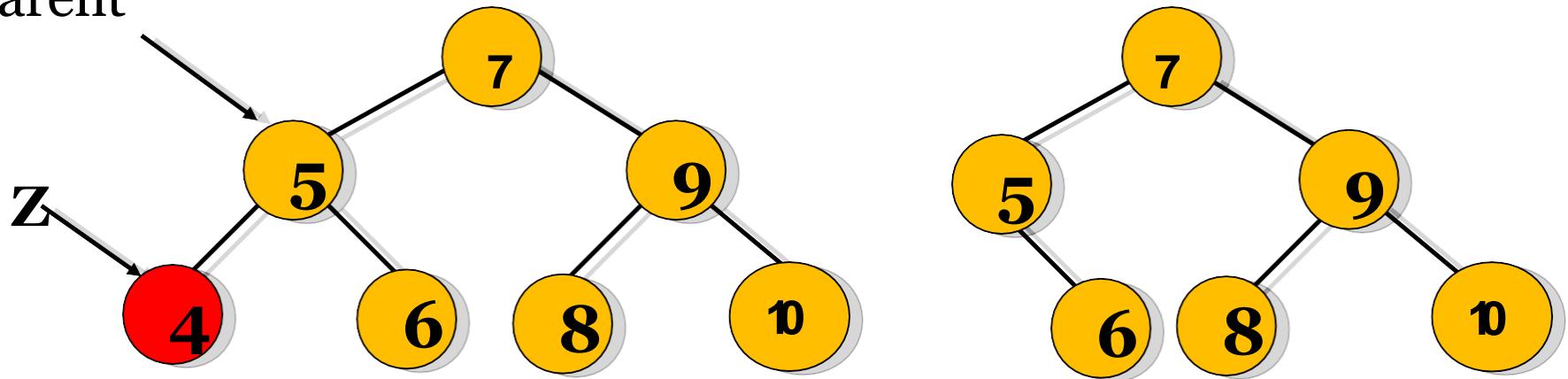
- link the parent of the target to the left (non-empty) subtree

# Delete a value from the BST

**Case 1:** removing a node with 2 EMPTY SUBTREES  
-replace the link in the parent with null

## Removing 4

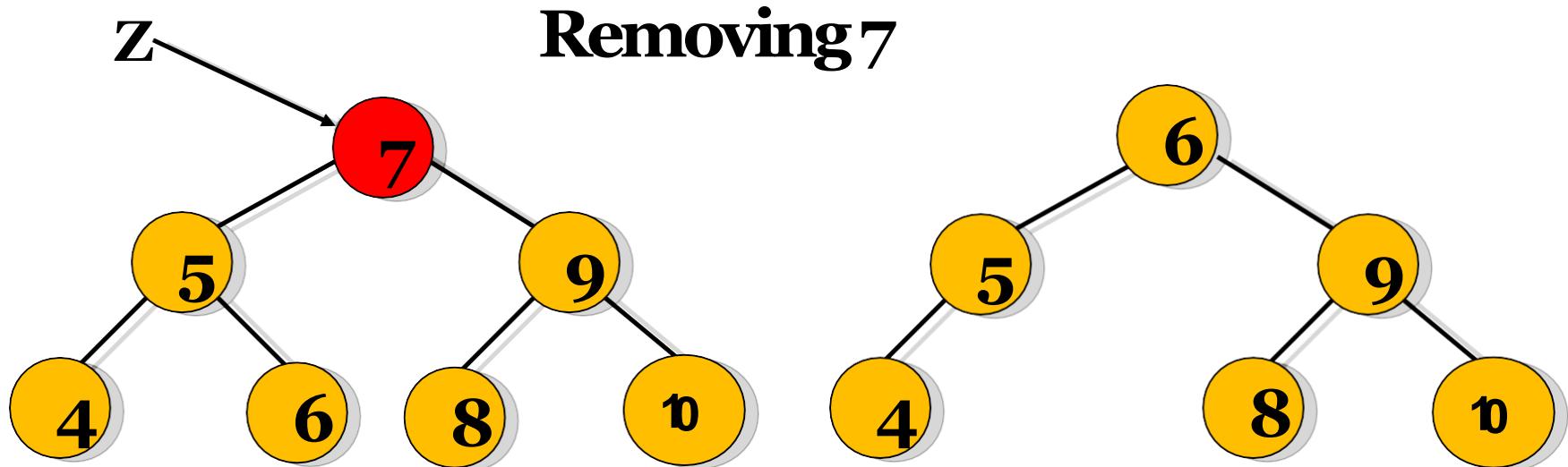
Parent



# Delete a value from the BST

**Case 2:** removing a node with 2 SUBTREES

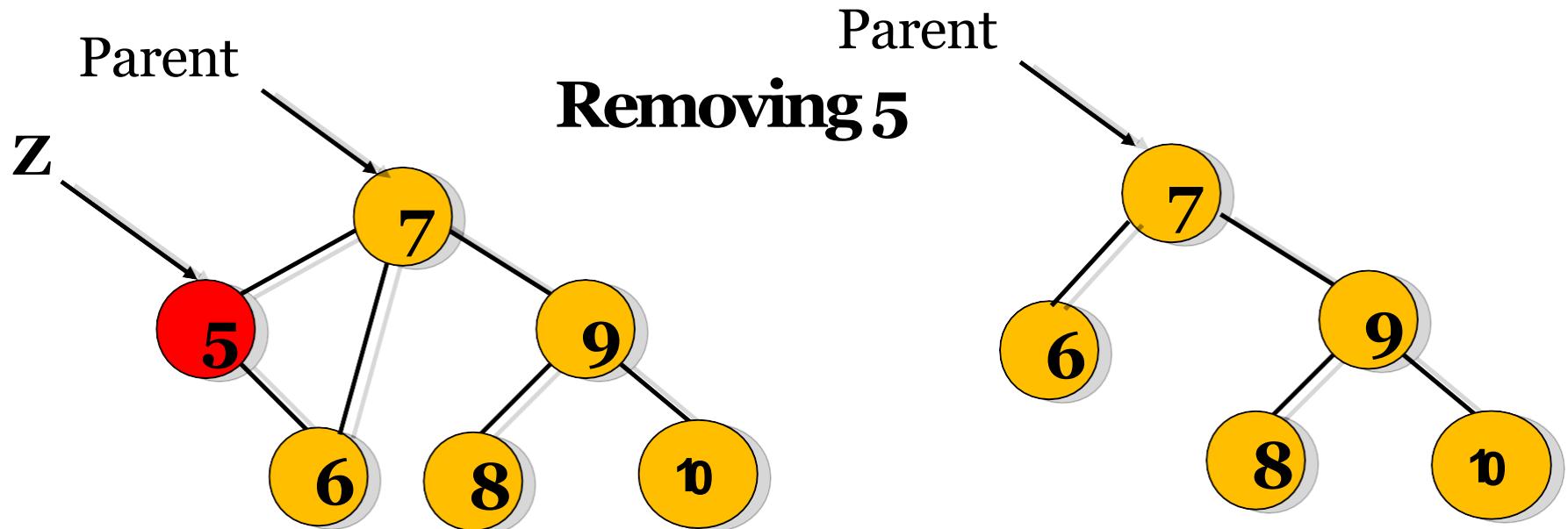
- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree



# Delete a value from the BST

**Case 3:** if the node has no left child

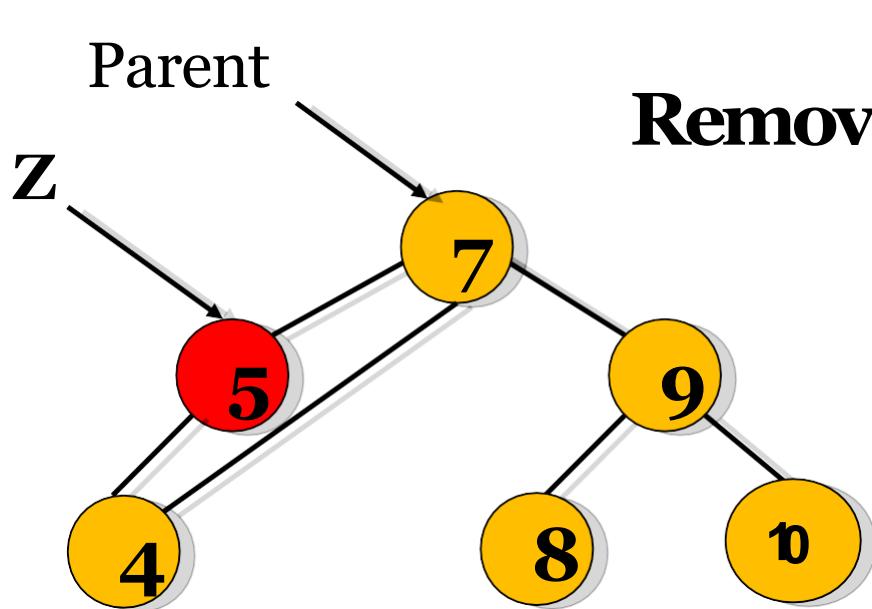
- link the parent of the node to the right (non-empty) subtree



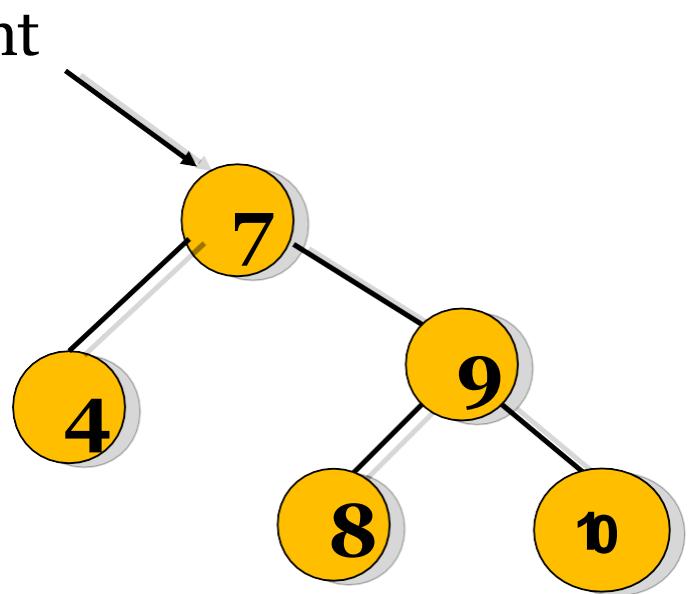
# Delete a value from the BST

**Case 4:** if the node has no right child

- link the parent of the node to the left (non-empty) subtree



**Removing 5**



# BST Deletion Algorithm

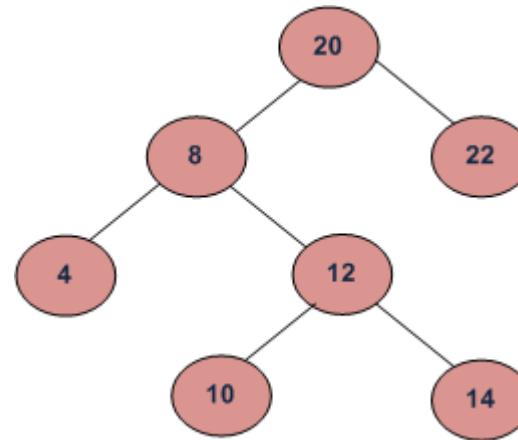
- **TREE-DELETE(T, z)**
  1. if left [z] = NIL .OR. right[z] = NIL
  2. then y  $\leftarrow$  z
  3. else y  $\leftarrow$  TREE-SUCCESSOR(z)
  4. if left [y]  $\neq$  NIL
  5. then x  $\leftarrow$  left[y]
  6. else x  $\leftarrow$  right [y]
  7. if x  $\neq$  NIL
  8. then p[x]  $\leftarrow$  p[y]
  9. if p[y] = NIL
  10. then root [T]  $\leftarrow$  x
  11. else if y = left[p[y]]
  12. then left [p[y]]  $\leftarrow$  x
  13. else right [p[y]]  $\leftarrow$  x
  14. if y  $\neq$  z
  15. then key [z]  $\leftarrow$  key [y]
  16. if y has other field, copy them, too
  17. return y

# Analysis of BST Operations

- ♣ The complexity of operations **search, insert** and **remove** in BST is  $O(h)$  , where h is the height.
- ♣ When the tree is balanced then it is  $O(\log n)$ . The updating operations cause the tree to become unbalanced.
- ♣ The tree can degenerate to a linear shape and the operations will become  $O(n)$

# Find k-th smallest element in BST

Given root of binary search tree and K as input, find K-th smallest element in BST. For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



We know that an [in-order traversal](#) of a binary search tree returns the nodes in ascending order. Thus, to find k'th smallest element, we can perform [in-order traversal](#) and store the inorder order in an array. Finally we traverse the array and return k'th element from starting. The problem with this approach is that it takes extra space.

We can solve this problem without using any extra space by keeping track of number of nodes processed so far while traversing the tree in [in-order fashion](#). When the number of nodes becomes equal to K, the current node is k'th smallest.

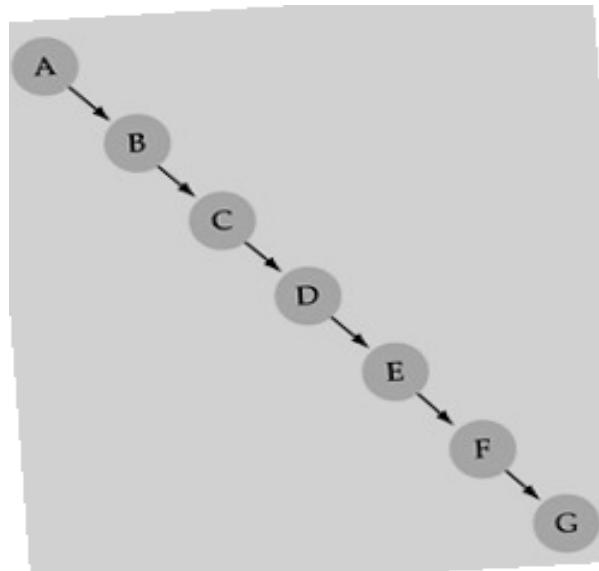
---

# Exercise:

## Find k-th Largest element in BST

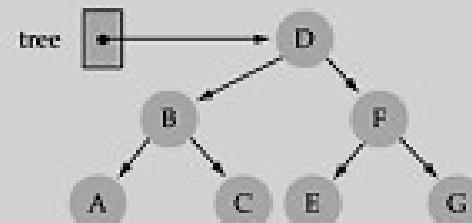
# What is a Degenerate BST?

- ♣ A degenerate binary search tree is one where most or all of the nodes contain only one sub node.
- ♣ It is unbalanced and, in the worst case, performance degrades to that of a linked list.
- ♣ If your add node function does not handle rebalancing then you can easily construct a degenerate tree by feeding it data that is already sorted.

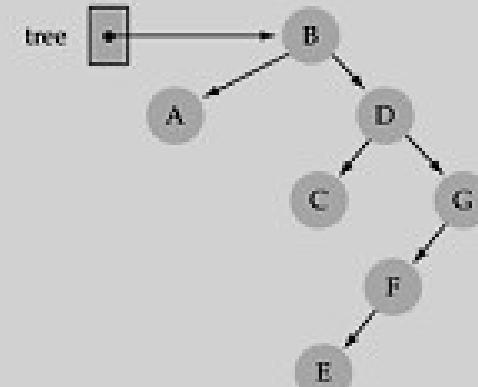


# Does the order of inserting elements into a tree matter?

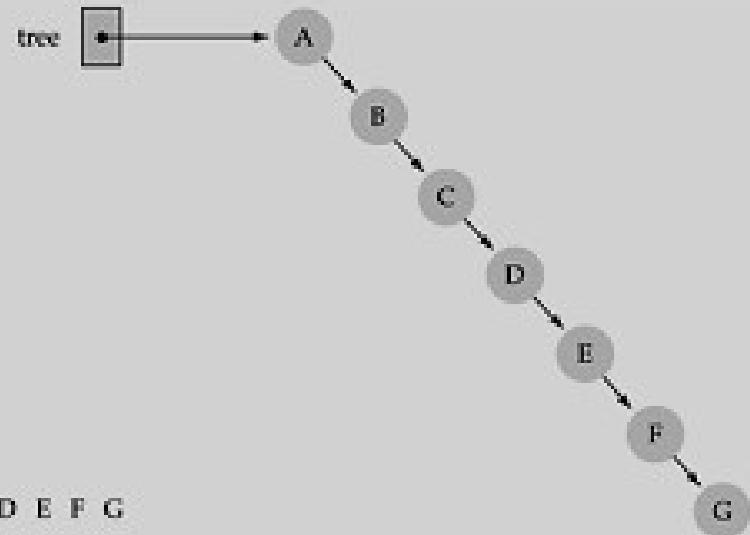
(a) Input: D B F A C E G



(b) Input: B A D C G F E



(c) Input: A B C D E F G



# Does the order of inserting elements into a tree matter?

- ♣ Yes, certain orders might produce very unbalanced trees or degenerated trees!
- ♣ Unbalanced trees are not desirable because search time increases!
- ♣ Advanced tree structures, such as **red-black trees**, **AVLtrees**, guarantee balanced trees.

# Better Search Trees

Prevent the degeneration of the BST :

- ♣ A BST can be set up to maintain balance during updating operations (insertions and removals)
- ♣ Types of ST which maintain the optimal performance in other words balanced trees:
  - splay trees
  - AVL trees
  - 2-4 Trees
  - Red-Black trees
  - B-trees

# Minimize The Search Time of Binary Search Tree In Dynamic Situation

- From the previous few examples, we know that the average and maximum search time will be minimized if the binary search tree is maintained as a complete binary search tree at all times.
- However, to achieve this in a dynamic situation, we have to pay a high price to restructure the tree to be a complete binary tree all the time.
- In 1962, Adelson-Velskii and Landis introduced a binary tree structure that is balanced with respect to the heights of subtrees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in  $O(\log n)$  time if the tree has  $n$  nodes. The resulting tree remains height-balanced. This is called an AVL tree.

# Balancing BST – (AVL Tree)

---

[Click Reference \(lab6ASD.pdf\)](#)

Last Topic - Session 9

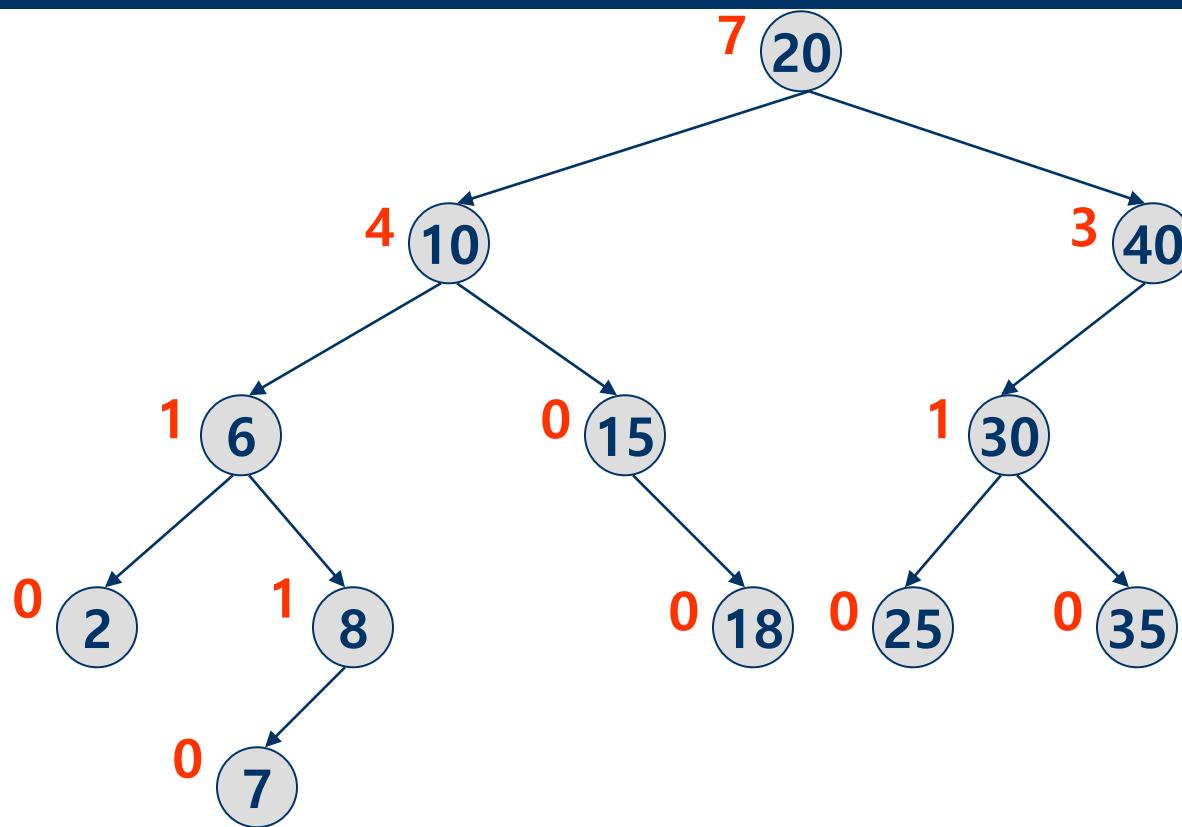
Remaining Slides in this PPT are  
out of syllabus.  
If interested you can do self study.

# Indexed Binary Search Trees

## Definition

- Binary search tree.
- Each node has an additional field ‘**LeftSize**’.
- Support search and delete operations **by rank** as well as all the binary search tree operations.
- **LeftSize**
  - the number of elements in its **left subtree**
  - the rank of an element with respect to the elements in its subtree (e.g., the fourth element in sorted order)

# Indexed Binary Search Tree Example



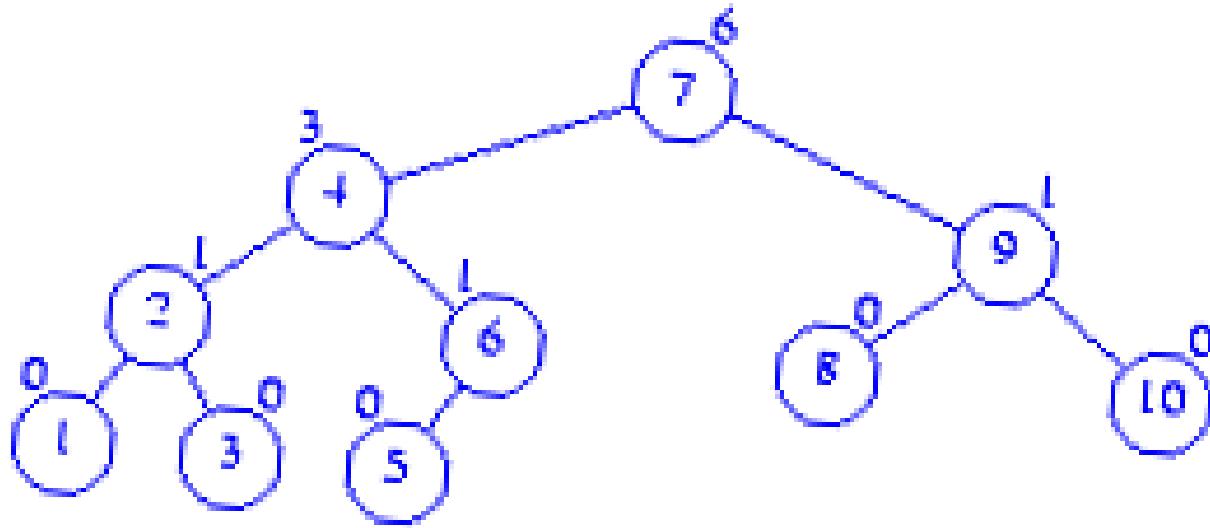
- What is the Leftsize for each node?
- LeftSize values are in red.

# LeftSize and Rank

---

- Rank of an element is its position in inorder  
(inorder = ascending key order).  
[2,6,7,8,10,15,18,20,25,30,35,40]
- rank(2)=0
- rank(15)=5
- rank(20)=7
- LeftSize( $x$ ) = rank( $x$ ) with respect to elements in the subtree rooted at  $x$

# Exercise

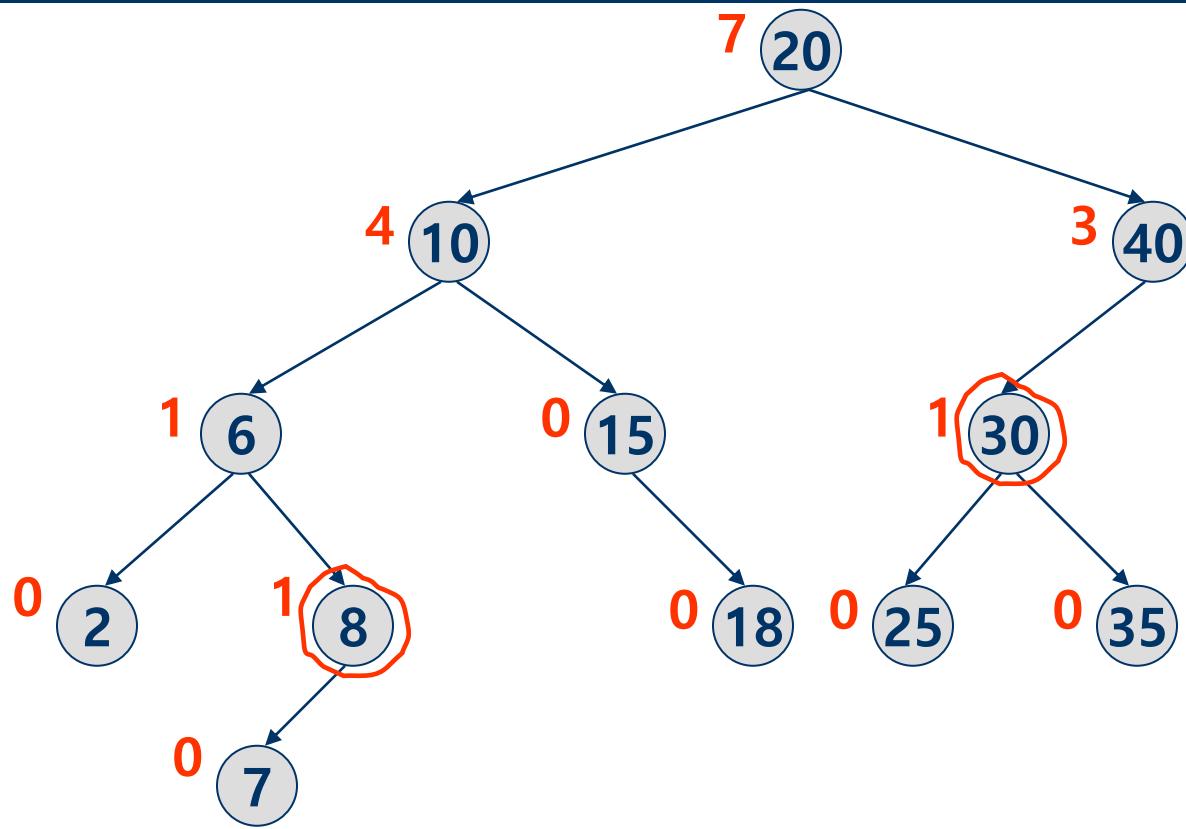


## Indexed Binary Search Tree – Search & Delete

---

- IndexSearch(rank) returns the rank<sup>th</sup> element
- IndexDelete(rank) deletes the rank<sup>th</sup> element
- If rank = x.LeftSize, desired element is x.element.
- If rank < x.LeftSize, desired element is rank<sup>th</sup> element in left subtree of x.
- If rank > x.LeftSize, desired element is  $(\text{rank} - (\text{x.LeftSize} + 1))$ <sup>th</sup> element in right subtree of x.

# Indexed Binary Search Example



- What is the 3<sup>rd</sup> element in this IndexedBST?
- What is the 9<sup>th</sup> element in this IndexedBST?

# Range queries

---

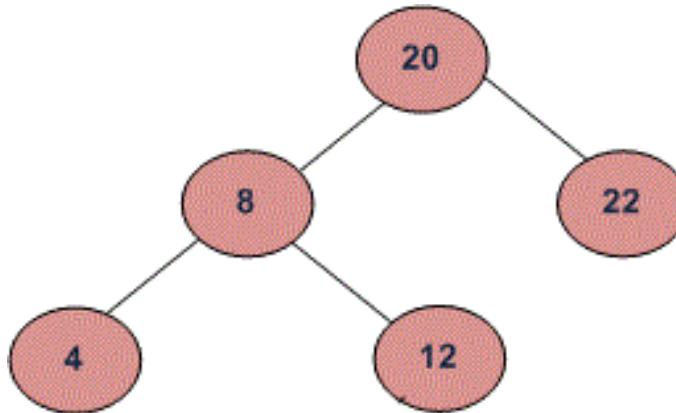
A [range query](#) on a range tree reports the set of points that lie inside a given interval. To report the points that lie in the interval  $[x_1, x_2]$ , we start by searching for  $x_1$  and  $x_2$

# Print BST keys in the given range



Given two values  $k_1$  and  $k_2$  (where  $k_1 < k_2$ ) and a root pointer to a Binary Search Tree. Print all the keys of tree in range  $k_1$  to  $k_2$ . i.e. print all  $x$  such that  $k_1 \leq x \leq k_2$  and  $x$  is a key of given BST. Print all the keys in increasing order.

For example, if  $k_1 = 10$  and  $k_2 = 22$ , then your function should print 12, 20 and 22.



**Approach - ???**

**Algorithm:**

- 1) If value of root's key is greater than  $k_1$ , then recursively call in left subtree.
- 2) If value of root's key is in range, then print the root's key.
- 3) If value of root's key is smaller than  $k_2$ , then recursively call in right subtree.

# Uses for Binary Search Trees



Use for storing and retrieving information  
n Insert, delete, and search faster than with a linked list  
Take advantage of log height  
Idea: Store information in an ordered way (keys)

---

Thanks!!!  
Queries?



**BITS Pilani**

Pilani|Dubai|Goa|Hyderabad

# Data Structures and Algorithms Design

**DSECLZG519**

Parthasarathy

BITS Pilani - WILP Division



# Extra Slides on Huffman Encoding

# Huffman Encoding

- In telecommunication, how do we represent a set of messages, each with an access frequency, by a sequence of 0's and 1's ?
- **Approach 1 / Static:** To use ASCII (American Standard Code for Information Interchange) code for each character of message. This is a static way (each code will be of 8bits). Can we reduce the bits needed ?
- **Approach 2 / Dynamic :** To minimize the transmission and decoding costs, we may use short strings (codes) to represent more frequently used messages.
- This problem can be solved by using an extended binary tree which is used in the 2-way merging problem and then creating the codes. This was proposed by Huffman and is a greedy method.

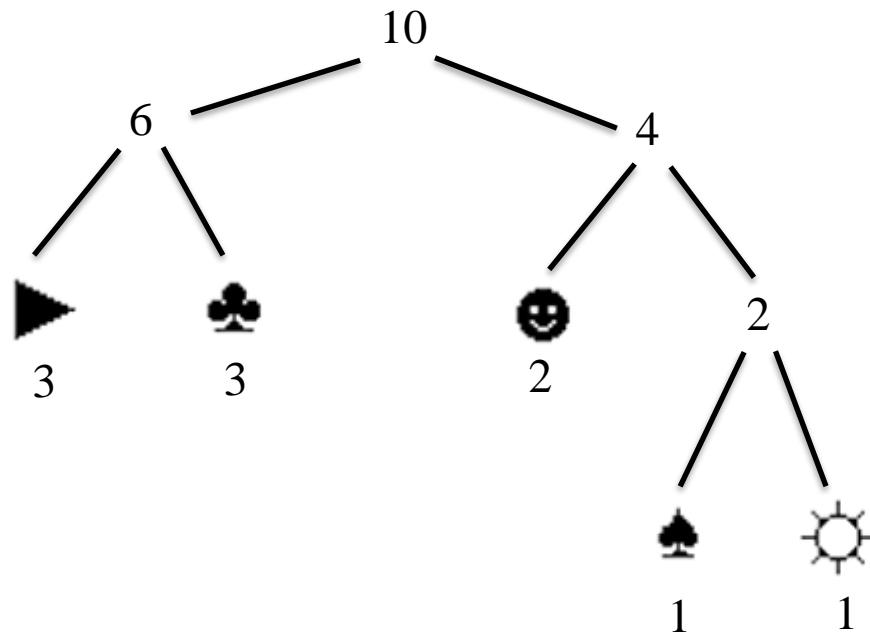
# Huffman Encoding Idea

- Suppose we have a message consisting of 5 symbols, e.g.  
[▶♣♣♣☺▶♣☀▶☺]
- How can we code this message using 0/1 so the coded message will have minimum length (for transmission or saving!)
- 5 symbols → at least 3 bits .For a simple encoding, length of code is  $10*3=30$  bits

|   |     |
|---|-----|
| ▶ | 000 |
| ♣ | 001 |
| ☺ | 010 |
| ♠ | 011 |
| ☀ | 100 |

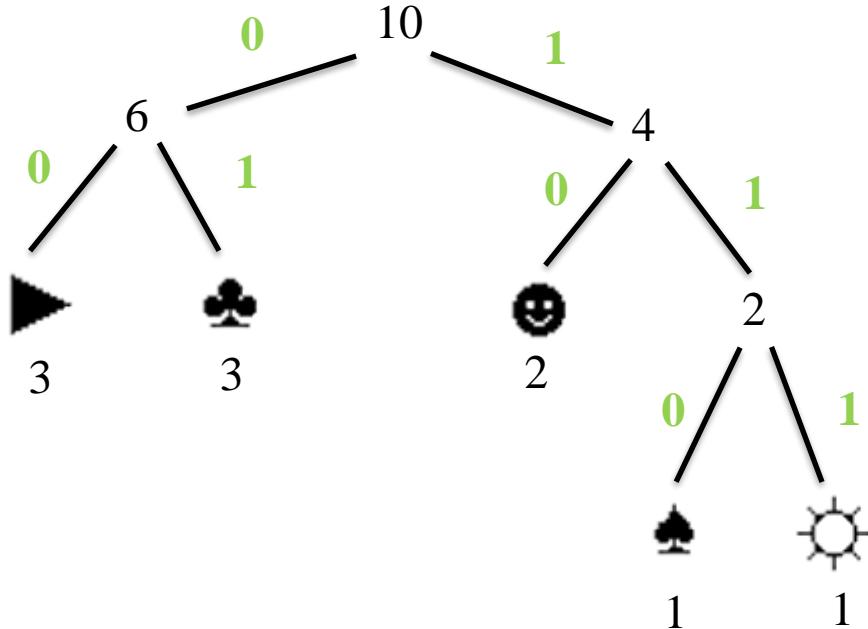
# Huffman Encoding Idea

*Intuition:* Those symbols that are more frequent should have smaller codes, yet since their length is not the same, there must be a way of distinguishing each code!



| Symbol | Freq. |
|--------|-------|
| ▶      | 3     |
| ♣      | 3     |
| 😊      | 2     |
| ♠      | 1     |
| ☀️     | 1     |

# Huffman Encoding Idea



| Symbol | Freq. | Code |
|--------|-------|------|
| ►      | 3     | 00   |
| ♣      | 3     | 01   |
| ☺      | 2     | 10   |
| ♠      | 1     | 110  |
| ☀      | 1     | 111  |

For Huffman code, length of encoded message for ►♣♣♣☺►♣☀►♠ will be  
 $= (3*2) + (3*2) + (2*2) + (1*3) + (1*3)$   
 $= 6+6+4+3+3$   
 $= 22 \text{ bits}$

# Huffman Encoding – Prefix codes

---

The variable-length codes assigned to input characters are Prefix Codes, meaning, the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

# Huffman Coding Example 1

- Assume that a message is made up of following characters and their frequencies are as mentioned.
- How many bits are needed if ASCII encoding is used?
- Draw the Huffman tree and code and explain how many bits are needed?
- Compare number of bits needed in ASCII vs Huffman
- Encode **aeaist** and decode it using Huffman.

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

# Huffman Coding Example 1

- How many bits are needed if ASCII encoding is used?

Answer:

In ASCII, each character is represented using 8bits.

So, Total number of bits needed for this message is :

$$\rightarrow (10*8)+(15*8)+(12*8)+(3*8)+(4*8) + (13*8)+(1*8)$$

$$\rightarrow 80+120+96+24+32+104+8$$

$$\rightarrow \mathbf{464 \text{ bits}}$$

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

# Huffman Coding Example 1

## Solution-

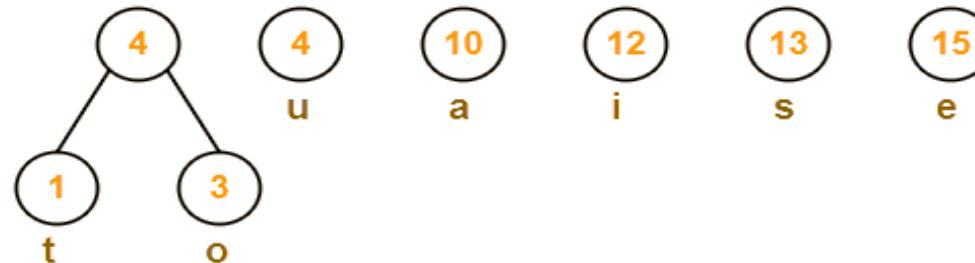
First let us construct the Huffman Tree.

Huffman Tree is constructed in the following steps-

### Step-01:

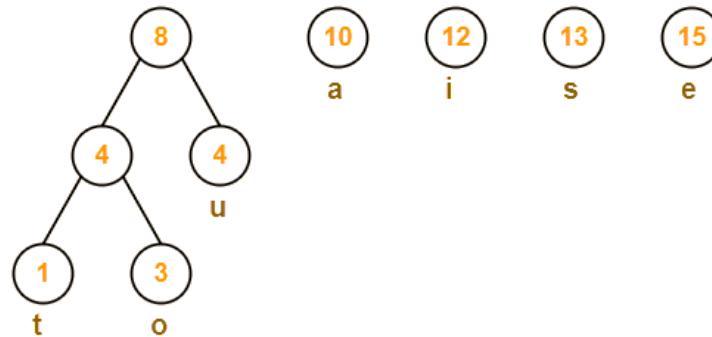


### Step-02:

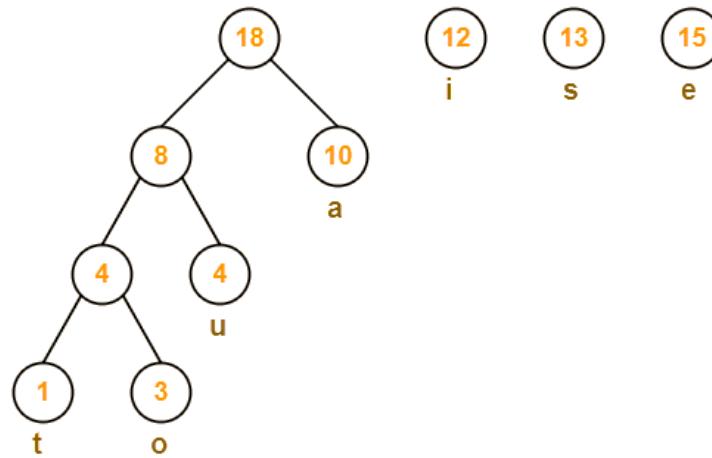


# Huffman Coding Example 1

Step-03:

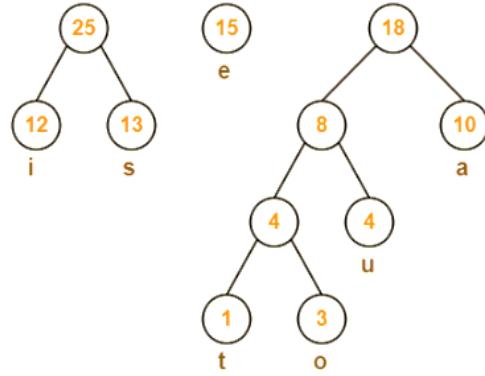
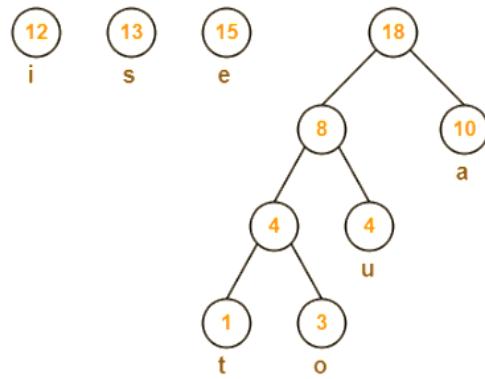


Step-04:

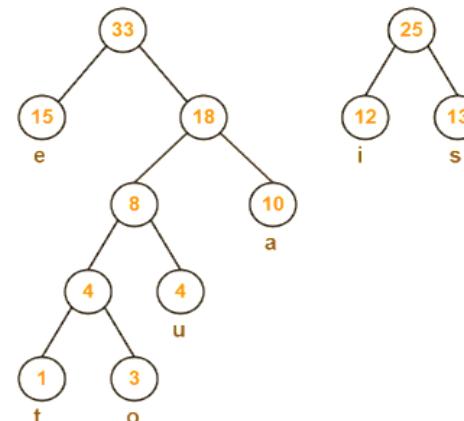
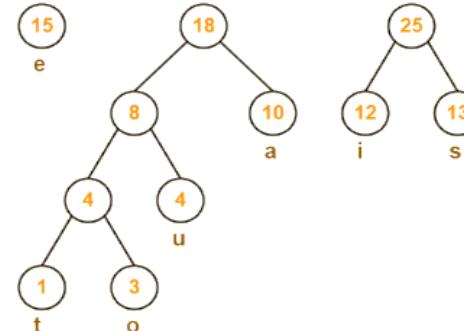


# Huffman Coding Example 1

Step-05:

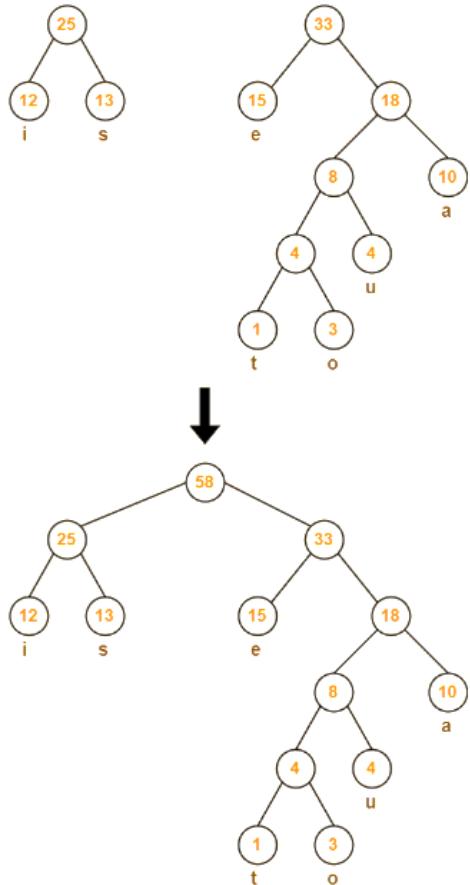


Step-06:



# Huffman Coding Example 1

Step-07:



Now,

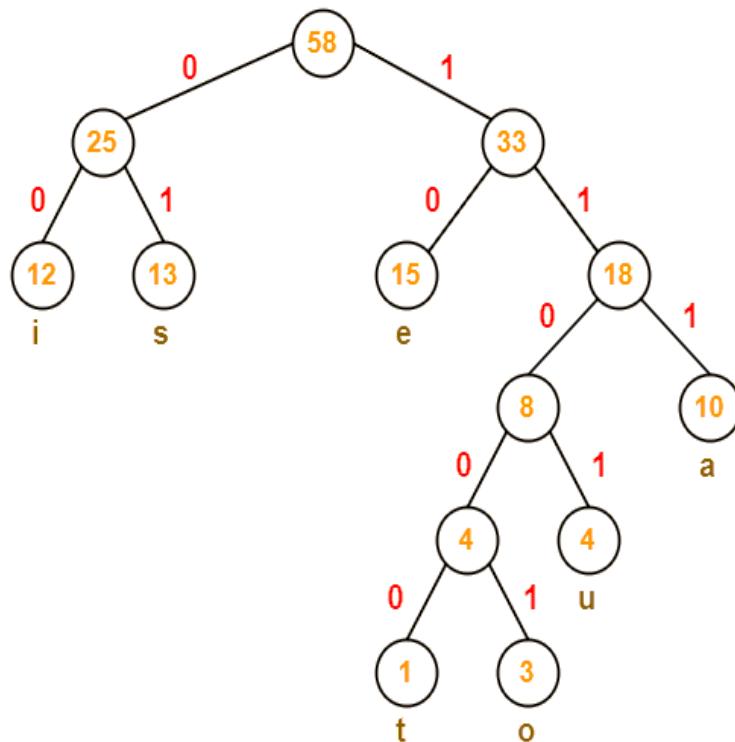
- We assign weight to all the edges of the constructed Huffman Tree.
- Let us assign weight '0' to the left edges and weight '1' to the right edges.

### Rule

- If you assign weight '0' to the left edges, then assign weight '1' to the right edges.
- If you assign weight '1' to the left edges, then assign weight '0' to the right edges.
- Any of the above two conventions may be followed.
- But follow the same convention at the time of decoding that is adopted at the time of encoding.

# Huffman Tree and Table

After assigning weight to all the edges, the modified Huffman Tree is-



Huffman Tree

| Character | Bit Rep | #Bits |
|-----------|---------|-------|
| a         | 111     | 3     |
| e         | 10      | 2     |
| i         | 00      | 2     |
| o         | 11001   | 5     |
| u         | 1101    | 4     |
| s         | 01      | 2     |
| t         | 11000   | 5     |
| Total:    |         | 23    |

# Huffman Coding Example 1

Following this rule, the Huffman Code for each character is-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.

# Huffman Coding Example 1

- How many bits are needed if Huffman encoding is used?

Answer:

So, Total number of bits needed for this message is :

$$\begin{aligned}
 & \rightarrow \{(10*3)+(15*2)+(12*2)+(3*5)+(4*4)+ \\
 & \quad (13*2)+(1*5)\} + 23 \\
 & \rightarrow \{30+30+24+15+16+26+5\} + 23 \\
 & \rightarrow 146 + 23 = \text{169 bits}
 \end{aligned}$$

Number of bits reduced from ASCII to Huffman :  $464 - 169 = \text{295 bits!!}$

Thanks to Huffman ☺

| C      | Freq | Bit Rep | #Bits |
|--------|------|---------|-------|
| a      | 10   | 111     | 3     |
| e      | 15   | 10      | 2     |
| i      | 12   | 00      | 2     |
| o      | 3    | 11001   | 5     |
| u      | 4    | 1101    | 4     |
| s      | 13   | 01      | 2     |
| t      | 1    | 11000   | 5     |
| Total: |      |         | 23    |

# Huffman Coding Example 1

- Encode **aeaist** and decode it.

Answer:

Encoding :

a → 111 , e → 10 , i → 00 , s → 01 , t → 11000

So, **aeaist** will be encoded as 111 10 111 00 01 11000 →  
**11110111000111000**

Decoding:

- **11110111000111000**
  - 111 → a
- **10111000111000**
  - 10 → e
- **111000111000**
  - 111 → a
- **000111000**
  - 00 → i
- **0111000**
  - 01 → s
- **11000 → t**

**So message decoded as aeaist**

| C      | Bit Rep | #Bits |
|--------|---------|-------|
| a      | 111     | 3     |
| e      | 10      | 2     |
| i      | 00      | 2     |
| o      | 11001   | 5     |
| u      | 1101    | 4     |
| s      | 01      | 2     |
| t      | 11000   | 5     |
| Total: |         | 23    |

# Exercises

- Create Huffman tree and generate binary digits for given 2 questions:

(1)

Letter frequency table

| Letter    | Z | K | M  | C  | U  | D  | L  | E   |
|-----------|---|---|----|----|----|----|----|-----|
| Frequency | 2 | 7 | 24 | 32 | 37 | 42 | 42 | 120 |

(2)

| character | Frequency |
|-----------|-----------|
| a         | 5         |
| b         | 9         |
| c         | 12        |
| d         | 13        |
| e         | 16        |
| f         | 45        |

## Interactive Example:

- Visit following URL and build tree for below text:

<https://people.ok.ubc.ca/yilucet/DS/Huffman.html>

“abababcccccccccdddadaaaaaaaaadx”

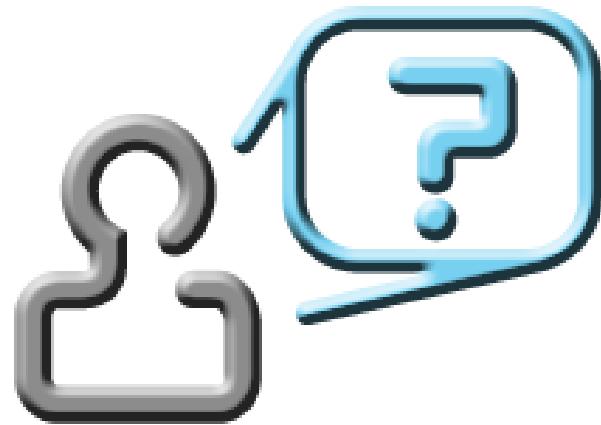
# Exercises

---

Construct the Huffman tree and table for :

|         |         |         |         |
|---------|---------|---------|---------|
| [a, 20] | [b, 21] | [c, 22] | [d, 23] |
| [e, 24] | [f, 25] | [g, 26] | [h, 27] |
| [i, 28] | [j, 29] | [k, 30] | [l, 31] |
| [m, 32] | [n, 33] | [o, 34] | [p, 35] |

- a) Encode *pagckbfjodi* and decode the Huffman code back.
- b) Compare the bits needed for ASCII and Huffman code.
- c) Self Study :
  - a) Try the complexity of Huffman Code creation and think of which DS to use
  - b) Universal Hashing



*Start a discussion if this seems complex!*

---

Thank You for your  
time & attention !

Contact : [parthasarathypd@wilp.bits-pilani.ac.in](mailto:parthasarathypd@wilp.bits-pilani.ac.in)

Slides are Licensed Under : [CC BY-NC-SA 4.0](#)

