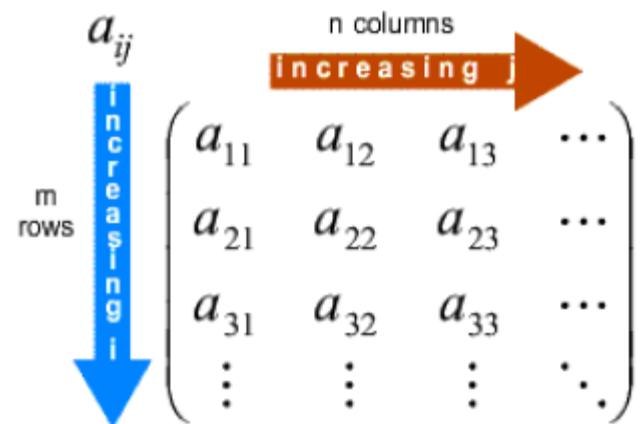


Special forms of square matrices- Sparse, Polynomial

2-Dimensional Arrays

Matrices

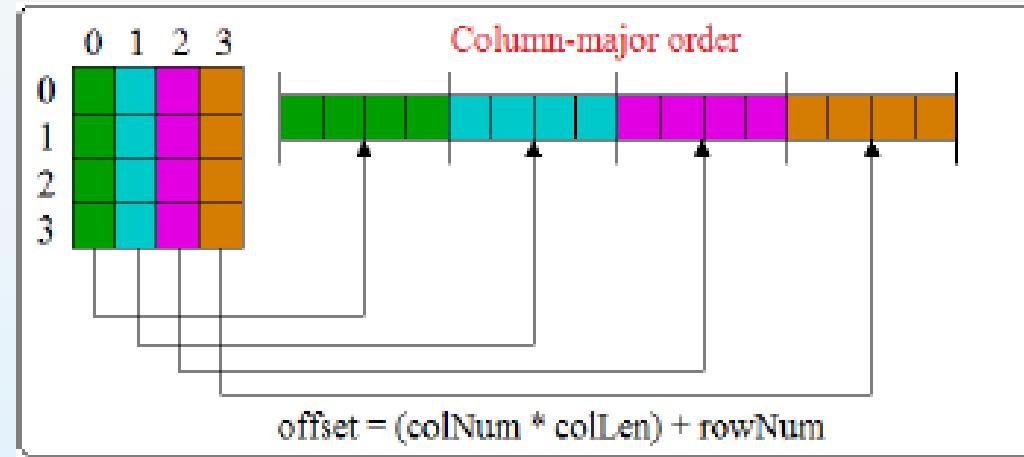
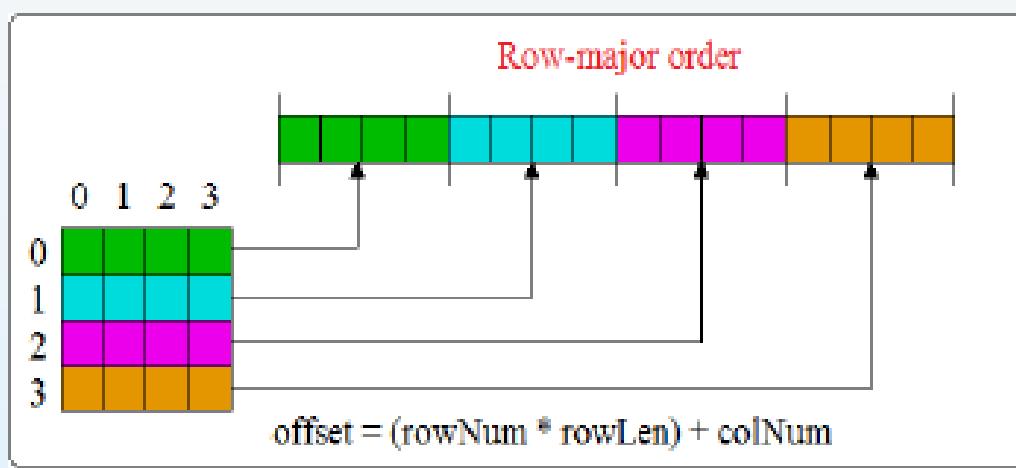
- A $m \times n$ matrix is a table with m rows and n columns (m and n are the **dimensions** of the matrix)



- Operations: addition, subtraction, multiplication, transpose, etc.

Matrices (Continued ...)

- Representations (mapped as 1-D array)



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

Row-major

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

Column-major

Special forms of square matrices ($m = n$)

- Diagonal: $M(i, j) = 0$ for $i \neq j$
 - Diagonal entries = m

$$M = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{array}{ll} M(i, j) = 0 & i \neq j \\ = A(i) & i = j \end{array}$$
$$A = [5 \ 3 \ 6 \ 0 \ 1]$$

Special forms of square matrices ($m = n$)

- Tridiagonal: $M(i, j) = 0$ for $|i - j| > 1$
 - ✓ 3 diagonals: Main $\rightarrow i = j$; Lower $\rightarrow i = j - 1$; Upper $\rightarrow i = j + 1$
 - ✓ Number of elements on three diagonals: $3*m-2$
 - ✓ Mapping: row-major, column-major or diagonals
(begin with lowest)

$$M = \begin{bmatrix} 5 & 8 & 0 & 0 & 0 \\ 7 & 3 & 0 & 0 & 0 \\ 0 & 2 & 6 & 4 & 0 \\ 0 & 0 & 9 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad M(i, j) = A(2*i + j) \quad |i - j| \leq 1$$
$$A = [\quad 5 \quad 8 \quad 7 \quad 3 \quad 0 \quad 2 \quad 6 \quad 4 \quad 9 \quad 0 \quad 3 \quad 0 \quad 1 \quad]$$

Special forms of square matrices ($m = n$)

- **Triangular** matrices
 - ✓ No. of elements in non-zero region: $n(n + 1) / 2$
 - ✓ Upper triangular: $M(i, j) = 0$ for $i > j$
 - ✓ Lower triangular: $M(i, j) = 0$ for $i < j$

$$U = \begin{bmatrix} 5 & 8 & 0 & 7 & 2 \\ 0 & 3 & 5 & 6 & 1 \\ 0 & 0 & 6 & 4 & 3 \\ 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix} \quad L = \begin{bmatrix} 8 & 0 & 0 & 0 & 0 \\ 7 & 3 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 1 & 2 & 8 & 9 & 0 \\ 5 & 6 & 0 & 1 & 0 \end{bmatrix}$$

Upper: $M(i, j) = A((n * i) + j - (i * (i+1) / 2))$ for $i \leq j$

Lower: $M(i, j) = A(i * (i+1) / 2 + j)$ for $i \geq j$

2-Dimensional Arrays

Special forms of square matrices ($m = n$)

- **Symmetric:** $M(i, j) = M(j, i)$ for all i and j
 - ✓ Can be stored as lower-or upper-triangular
 - ✓ E.g., Table of inter-city distance for a set of cities

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

Sparse Matrices

Sparse matrices

- Number of non-zero elements is very less **compared** to total number of elements
- Represented as a 1-D **array of triplets**
 - ✓ Element 1 onwards -Row, Column and Value –for all non-zero elements
 - ✓ Element 0 –number of rows, columns and non-zero elements

0	0	0	0	9	0
0	8	0	0	0	0
4	0	0	2	0	0
0	0	0	0	0	5
0	0	2	0	0	0



Row	Column	Value
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

Polynomials

Polynomial

- An expression of the form

$$a_n x^n + a_{(n-1)} x^{(n-1)} + \dots + a_1 x^1 + a_0$$

- One possible way of representation

- ✓ Store coefficients of the terms in an array element at position equal to the exponent
- ✓ Disadvantage: Waste of space specially in case of sparse polynomial

$$P(x) = 16x^{21} - 3x^5 + 2x + 6$$

6	2	0	0	0	-3	0	0	16
---	---	---	---	---	----	---	-------	---	----

WASTE OF SPACE!

Polynomial(Continued ...)

- Represented as a 1-D array of doublets
 - ✓ Element 1 onwards -**coefficient and exponent** of all terms
 - ✓ Element 0 –**number of terms** (either in coefficient or exponent field)

$$P(x) = 23x^9 + 18x^7 - 41x^6 + 163x^4 - 5x + 3$$

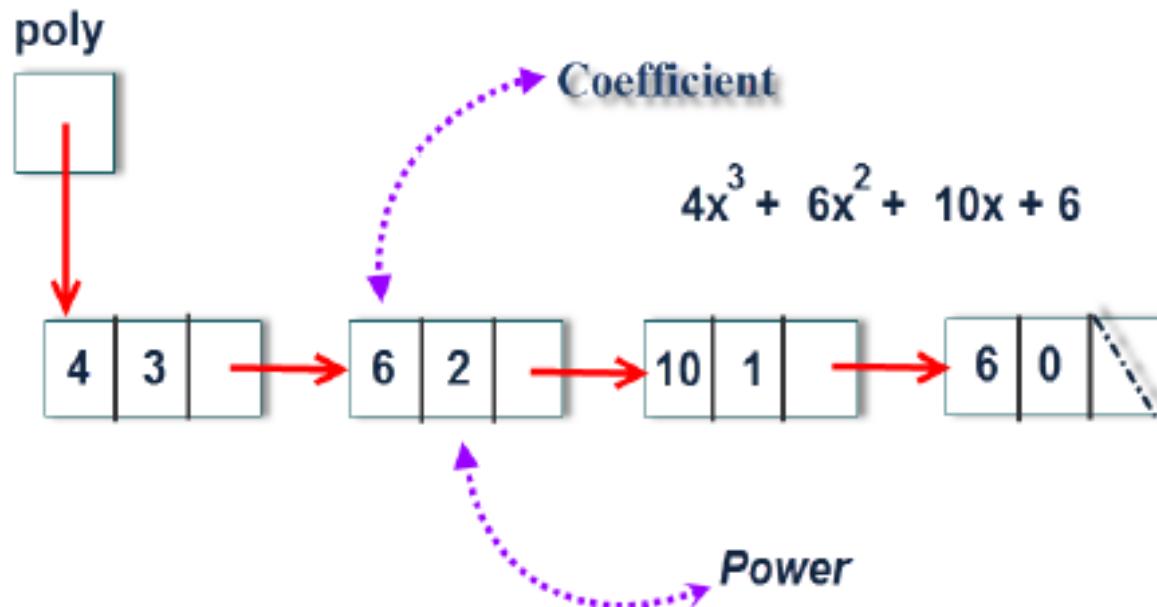
Coefficient

Exponent

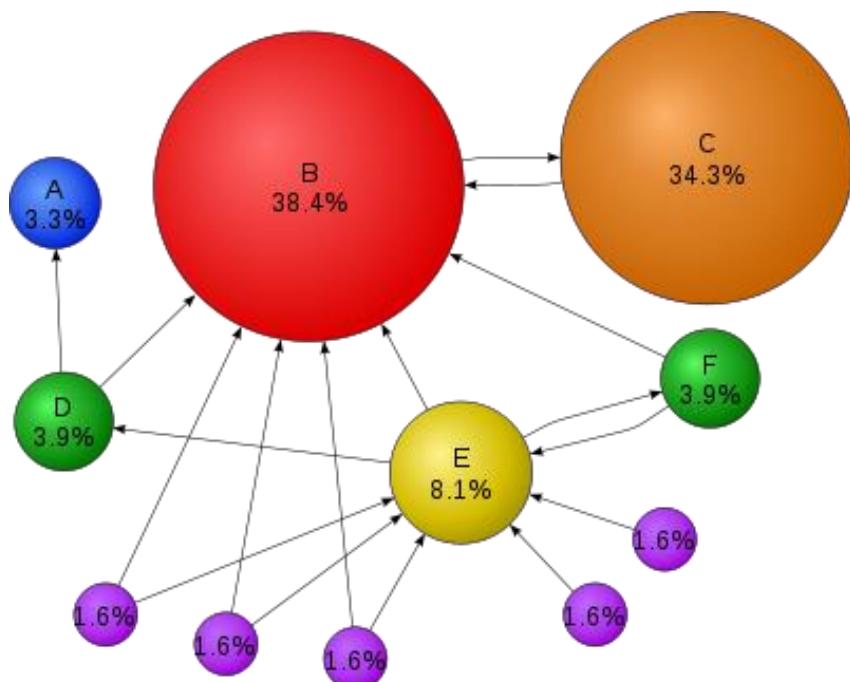
6	23	18	-41	163	-5	3
x	9	7	6	4	1	0
0	1	2	3	4	5	6

Polynomial(Continued ...)

- Represented as a **linked list**
 - ✓ Store the **coefficient** and **exponent** (power) of each term in a node of a singly linked list

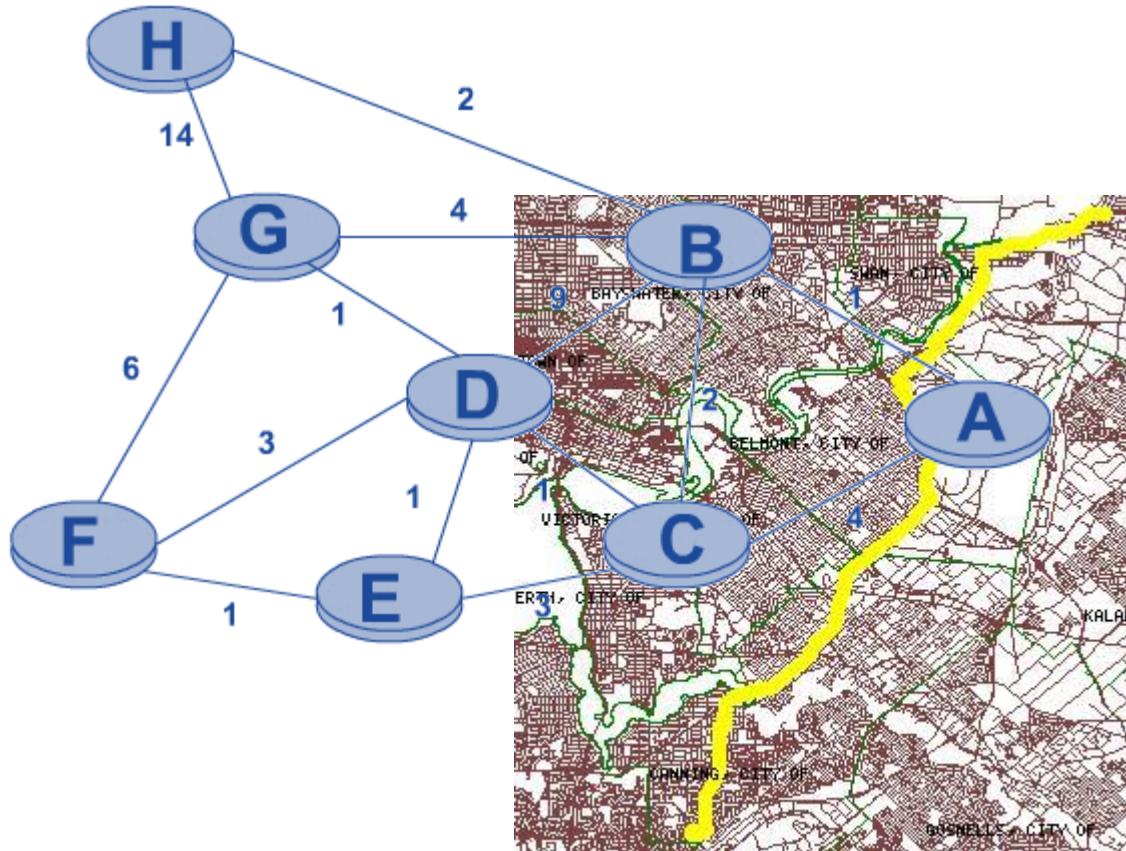


Why you want to study Algorithms?



- Making a lot of money out of a great algorithm
- \$1,000,000,000?
- Example:
- PageRank algorithm by Larry Page: The soul of Google search engine.
Google total assets: \$31 billions on 2008

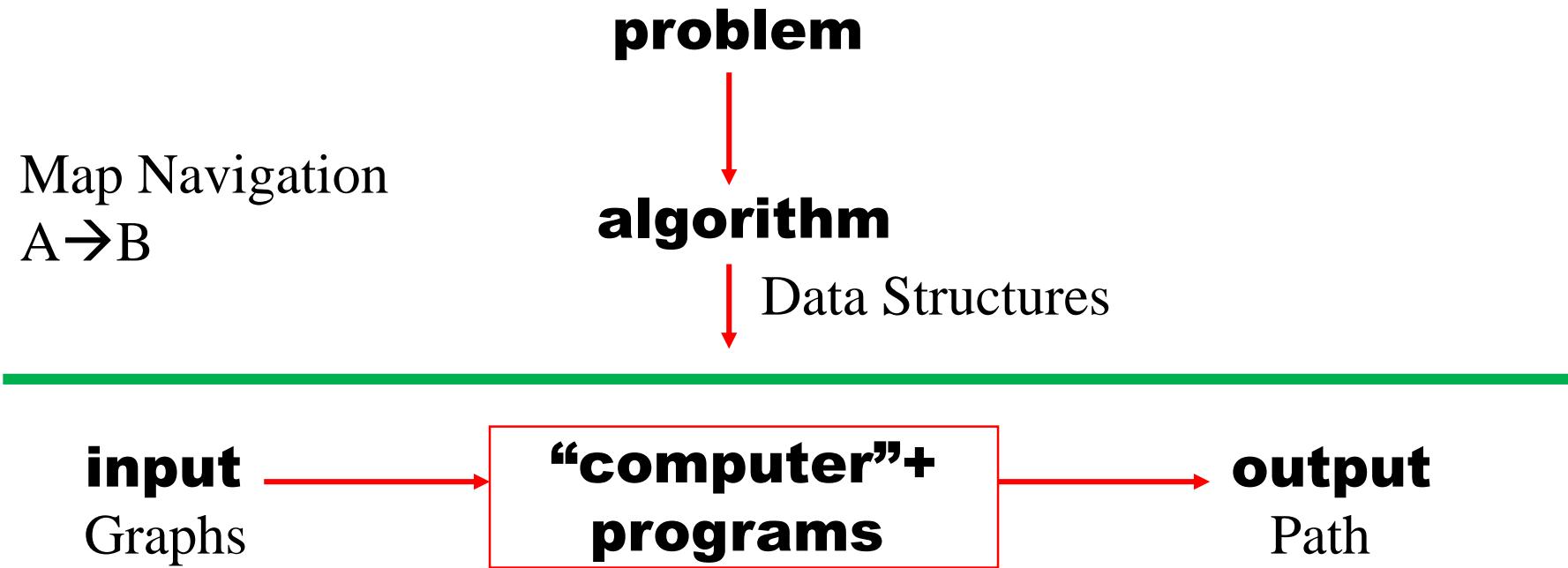
Why you want to study Algorithms?



- Simply to be cool to invent something in computer science
- Example: Shortest Path Problem and Algorithm
- Used in GPS and Mapquest or Google Maps

Algorithm and Data Structures.

- An algorithm is a sequence of unambiguous instructions/operations for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



How to study algorithms?

- ↳ Problem
 - ↳ Representation/data structure in computer
 - ↳ Operations on representations
-

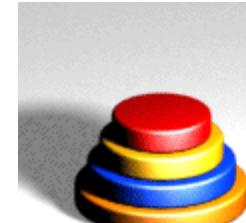
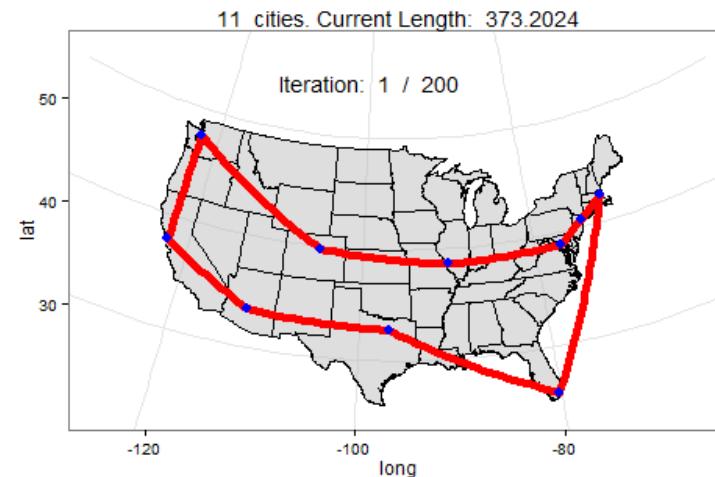
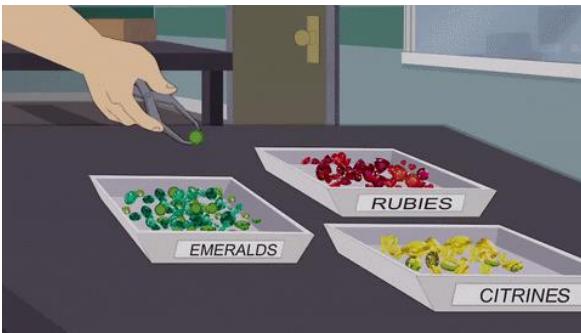
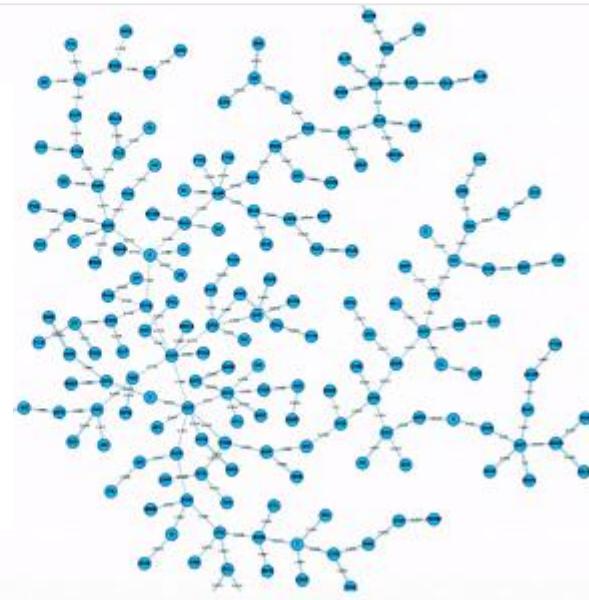
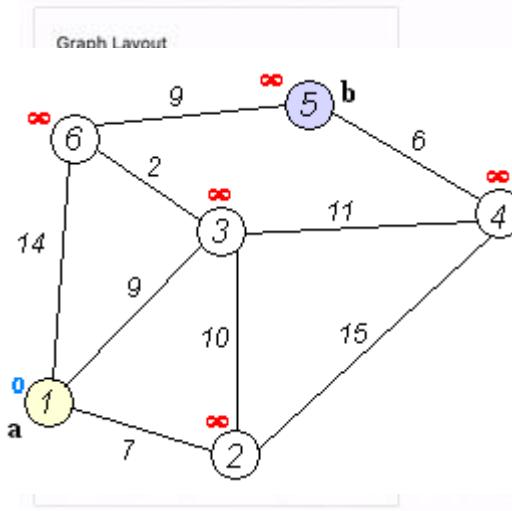
Some Important Points.

- Each step of an algorithm is unambiguous
- The range of inputs has to be specified carefully
- The same algorithm can be represented in different ways
- The same problem may be solved by different algorithms
- Different algorithms may take different time to solve the same problem – we may prefer one to the other

Fundamentals of Algorithmic Problem Solving.

- 1.Understanding the problem
- 2.Ascertaining the capabilities of a computational device
- 3.Choose between exact and approximate problem solving
- 4.Deciding on appropriate **data structure**
- 5.**Algorithm** design techniques
- 6.Methods of specifying an algorithm
 - **Pseudocode** (for, if, while //, ←, indentation...)
- 7.Prove an algorithm's correctness – mathematic induction
- 8.**Analyzing** an algorithm – Simplicity, efficiency, optimality
- 9.Coding an algorithm

Some Well-known Computational Problems.



What is data?

❶ Data

- ❷ A collection of facts from which conclusion may be drawn. Example: data: Temperature 35°C;
Conclusion: It is hot.

❸ Types of data

- ❹ Textual: For example, your name (Muhammad)
- ❹ Numeric: For example, your ID (090254)
- ❹ Audio: For example, your voice
- ❹ Video: For example, your voice and picture

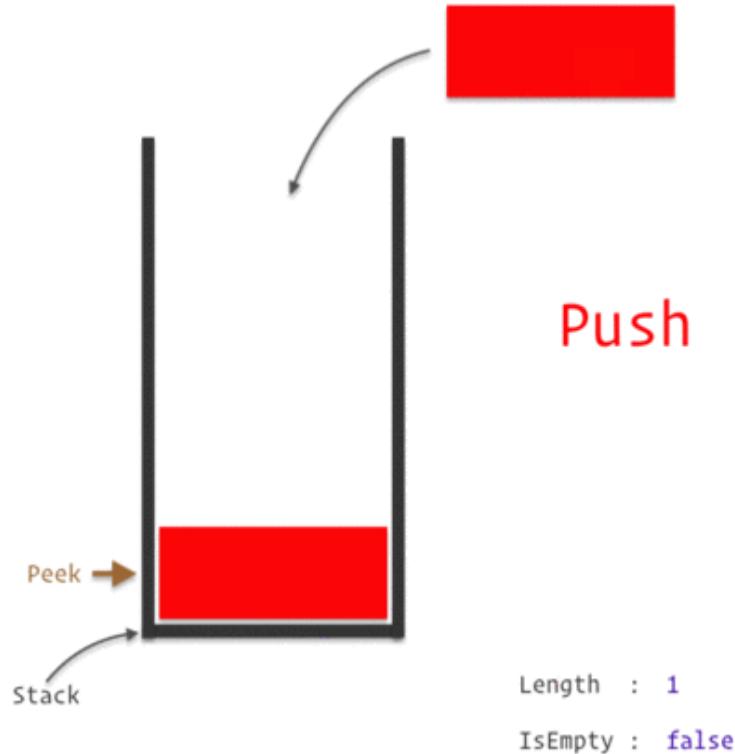


So, what is Data Type then?

- ↳ **Data Type:** Collection of objects and a set of operations that act on those objects.
- ↳ **Abstract Data Type:**
An abstract data type(ADT) is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

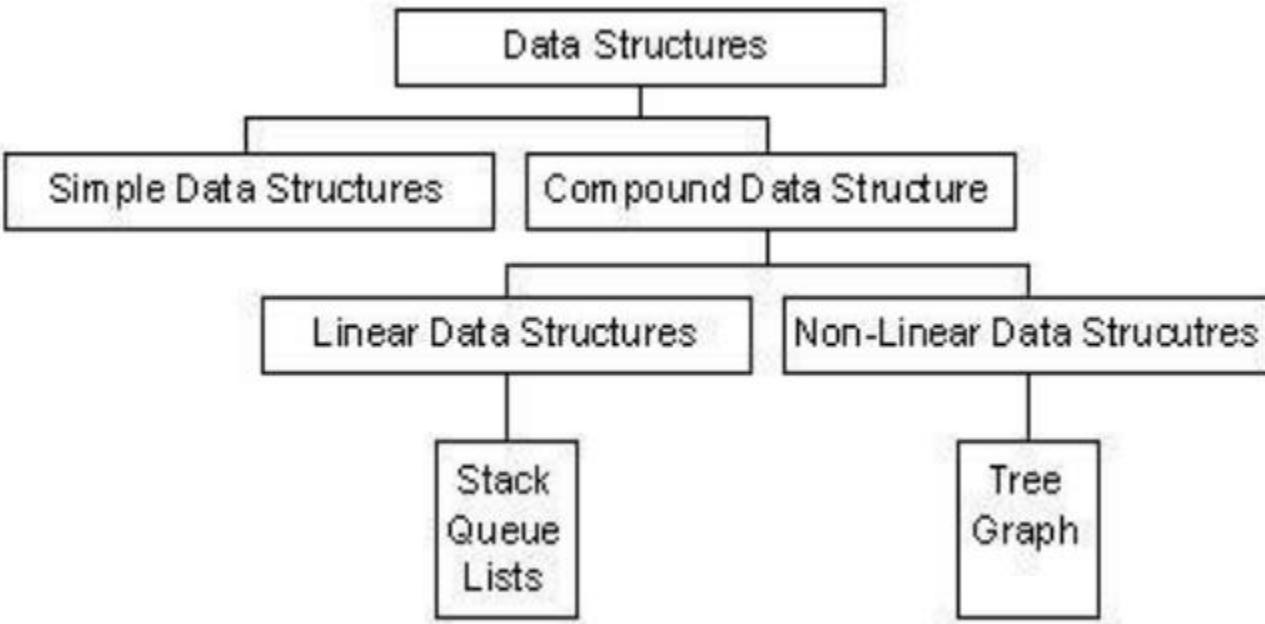
A logical view of how we view data and operations.

And, what is data structure?



- A particular way of **storing** and **organizing data** in a computer so that it can be used efficiently and effectively.
- **Data structure is the logical or mathematical model of a particular organization of data.**
- A group of data elements grouped together under one name.
 - For example, an array of integers

Classification of data structures?



- Data structures can be classified as
 1. Simple data structure.
 2. Compound data structure.
 - i. Linear data structure
 - ii. Non linear data structure

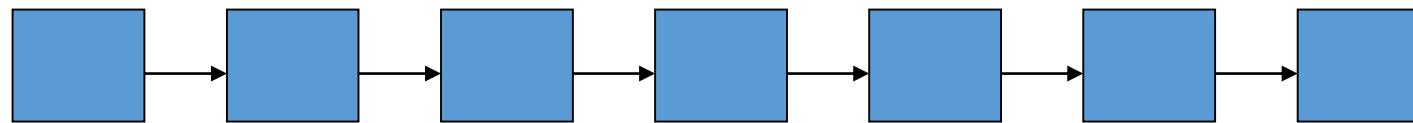
Classification of data structures?

- **Simple data structure:** Constructed with the help of primitive data structure. Examples: Variables, pointers, structures, unions.
- **Compound data structure:** Constructed with the help of any one of the primitive data structure with a specific functionality and designed by user:
 - i. **Linear data structure:** Continuous arrangement of data elements in the memory. **Relationship of adjacency** is maintained between the data elements.
 - ii. **Non linear data structure:** collection of randomly distributed set of data item joined together by using a special pointer (tag). **Relationship of adjacency** is not maintained between the data elements.

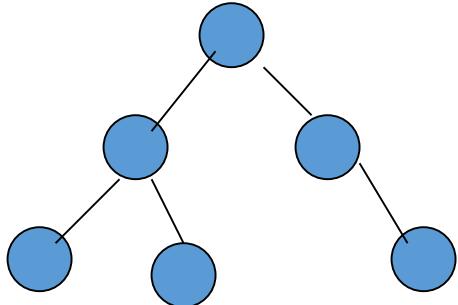
Types of data structures.



Array



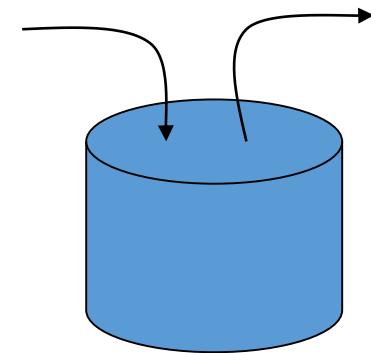
Linked List



Tree



Queue



Stack

The need for data structures.

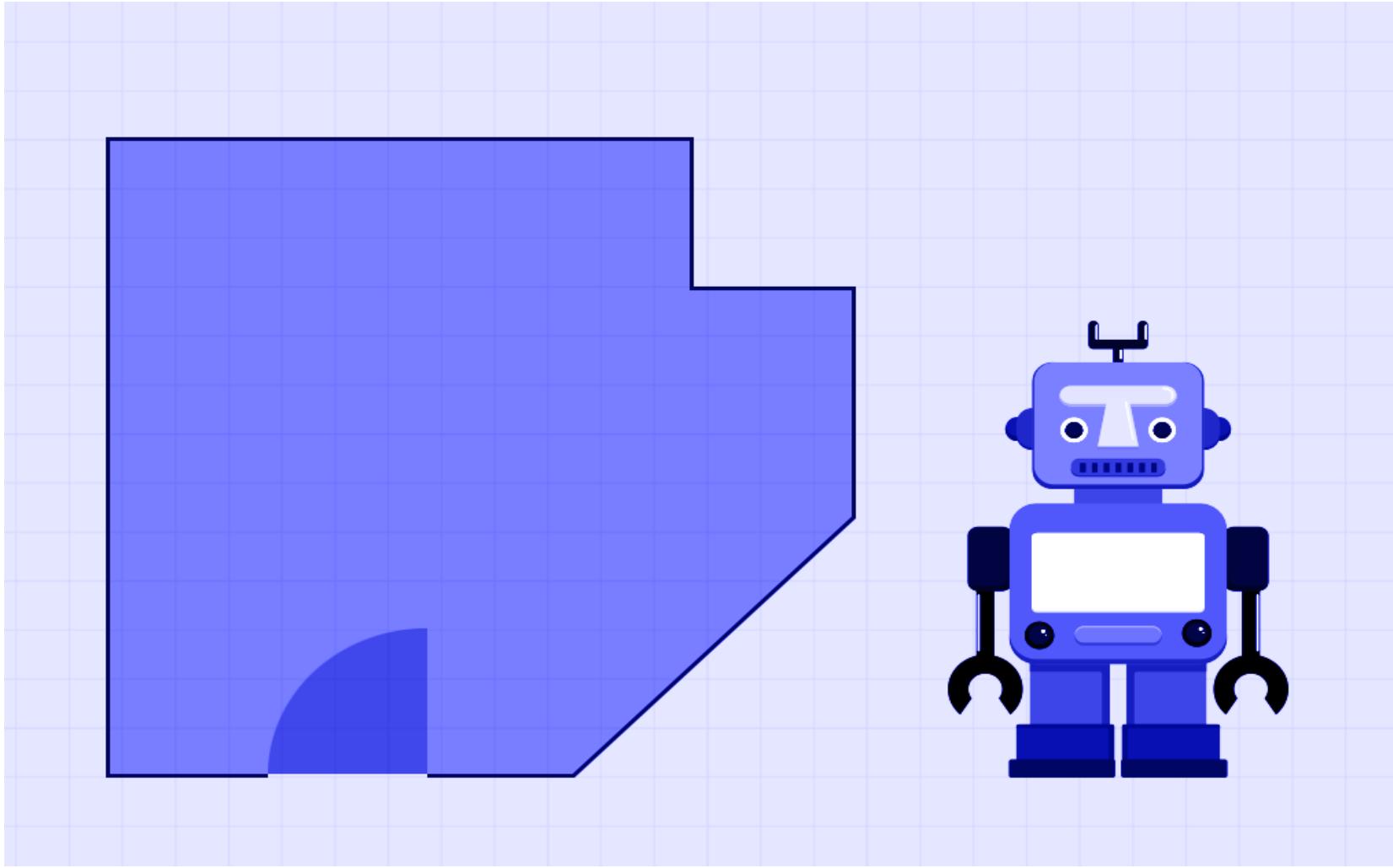
- 🔔 **Goal:** to **organize data**
- 🔔 **Criteria:** to facilitate **efficient**
 - **storage** of data
 - **retrieval** of data
 - **manipulation** of data
- 🔔 **Design Issue:**
 - **select and design** appropriate data types

This is the main motivation to learn and understand data structures.

Data Structure Operations.

- 🔔 **Traversing:** Accessing each data element exactly once so that certain items in the data may be processed
- 🔔 **Searching:** Finding the location of the data element (key) in the structure
- 🔔 **Insertion:** Adding a new data element to the structure
- 🔔 **Deletion:** Removing a data element from the structure
- 🔔 **Sorting:** Arrange the data elements in a logical order (ascending/descending)
- 🔔 **Merging:** Combining data elements from two or more data structures into one

What is algorithm?



What is algorithm?

- 🔔 A finite set of instructions which accomplish a particular task
- 🔔 A method or process to solve a problem
- 🔔 Transforms input of a problem to output

Algorithm = Input + Process + Output

- 🔔 Algorithm development is an art – **it needs practice, practice and only practice!**

What is a good algorithm?

- 🔔 It must be correct
- 🔔 It must be finite (in terms of time and size)
- 🔔 It must terminate
- 🔔 It must be unambiguous (Which step is next?)
- 🔔 It must be space and time efficient

A program is an instance of an algorithm, written in some specific programming language

A simple algorithm

🔔 Problem: Find maximum of a, b, c

🔔 Algorithm

- Input = a, b, c
- Output = max
- Process
 - Let max = a
 - If b > max then
 max = b
 - If c > max then
 max = c
 - Display max

Order is very important!!!

Algorithm development: Basics

- 🔔 Clearly identify:
 - what output is required?
 - what is the input?
 - What steps are required to transform input into output
 - The most crucial bit
 - Needs problem solving skills
 - A problem can be solved in many different ways
 - Which solution, amongst the different possible solutions is optimal?

How to express an algorithm?

- 🔔 A sequence of steps to solve a problem
- 🔔 We need a way to express this sequence of steps
 - **Natural Language** (NL) or **Programming language** (PL) is another choice, but again not a good choice. Why?
 - Algorithm should be PL independent
 - We need some balance
 - We need PL independence
 - We need clarity
 - **Pseudo-code** provides the right balance

What is Pseudo-code?

- Pseudo-code is a short hand way of describing a computer program
 - Rather than using the specific syntax of a computer language, more general wording is used
 - Mixture of NL and PL expressions, in a systematic way
 - Using pseudo-code, it is easier for a non-programmer to understand the general workings of the program
-

Components of Pseudo-code.

1. Expressions

- Standard mathematical symbols are used
 - Left arrow sign (\leftarrow) as the assignment operator in assignment statements (equivalent to the `=` operator in C++)
 - Equal sign ($=$) as the equality relation in Boolean expressions (equivalent to the `==` relation in `++`)
 - For example

Sum \leftarrow 0

Sum \leftarrow Sum + 5

What is the final value of sum?

Components of Pseudo-code.

2. Decision structures (if-then-else logic)

- if condition then true-actions [else false-actions]
- Use ***indentation*** to indicate what actions should be included in the true-actions and false-actions. For example:

```
if marks > 50 then  
    print "Congratulation, you are passed!"  
else  
    print "Sorry, you are failed!"  
end if
```

What will be the output if marks are equal to 75?

Components of Pseudo-code.

3. Loops (Repetition)

- Pre-condition loops: **While** loops
 - **while** condition **do** actions
 - Use indentation to indicate what actions to include in loop actions
 - For example

```
while counter < 5 do
    print "Welcome to MCA4252!"
    counter ← counter + 1
end while
```

What will be the output if counter is initialised to 0, 7?

Components of Pseudo-code.

3. Loops (Repetition)

- Pre-condition loops: **For** loops
 - **for** variable-increment-definition **do** actions
 - Use indentation to indicate what actions to include in loop actions
 - For example

```
for counter ← 0; counter < 5; counter ← counter + 2 do  
    print “Welcome to MCA4252!”  
end for
```

What will be the output ?

Components of Pseudo-code.

3. Loops (Repetition)

- Post-condition loops: **Do** loops
 - **do** condition **while** actions
 - Use indentation to indicate what actions to include in loop actions
 - For example

```
do  
    print "Welcome to CS204!"  
    counter ← counter + 1  
while counter < 5
```

What will be the output if counter is initialised to 0, 7?

Components of Pseudo-code.

4. Function declarations

- **return_type method_name (parameter_list) method_body**

- For example

```
integer sum ( integer num1, integer num2)
```

```
start
```

```
    result ← num1 + num2
```

```
end
```

Components of Pseudo-code.

4. Function Calls

- **object.function(arguments)**

- For example

mycalculator.sum(num1, num2)



Components of Pseudo-code.

4. Function returns

- **return value**

- For example

```
integer sum ( integer num1, integer num2)
```

```
start
```

```
    result ← num1 + num2
```

```
    return result
```

```
end
```

Add Comments.

Algorithm Design: Practice

- ↳ **Example 1: Determining even/odd number**
 - ↳ A number divisible by 2 is considered an even number, while a number which is not divisible by 2 is considered an odd number. Write pseudo-code to display first N odd/even numbers.

- ↳ **Example 2: Computing Weekly Wages**
 - ↳ Gross pay depends on the pay rate and the number of hours worked per week. However, if you work more than 40 hours, you get paid time-and-a-half for all hours worked over 40. Write the pseudo-code to compute gross pay given pay rate and hours worked

Even/ Odd Numbers

Input range

```
for num←0; num<=range; num←num+1 do
    if num % 2 = 0 then
        print num is even
    else
        print num is odd
    endif
endfor
```

Computing weekly wages

```
Input hours_worked, pay_rate
if hours_worked <= 40 then
    gross_pay ← pay_rate × hours_worked
else
    basic_pay ← pay_rate × 40
    over_time ← hours_worked - 40
    over_time_pay ← 1.5 × pay_rate × over_time
    gross_pay ← basic_pay + over_time_pay
endfor
print gross_pay
```

Homework

1. Write an algorithm to find the largest of a set of numbers. You do not know the number of numbers.
2. Write an algorithm in pseudocode that finds the average of (n) numbers.

For example: numbers are [4,5,14,20,3,6]



Analysis of Algorithms

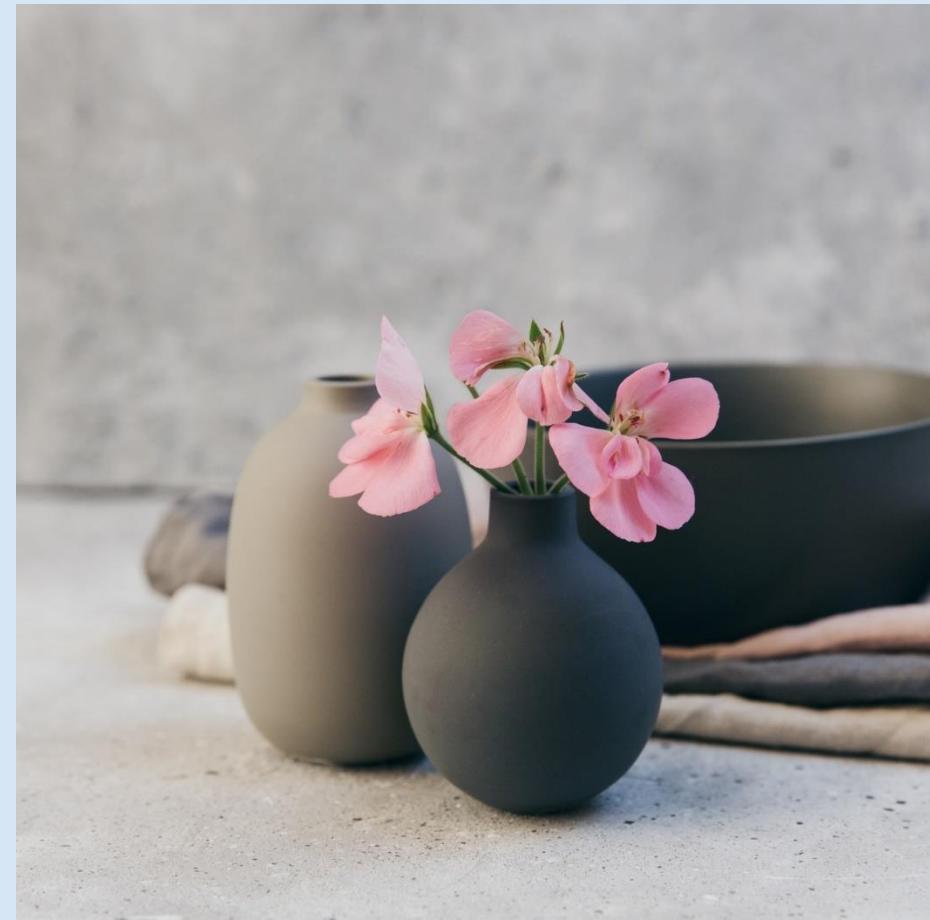
🔔 How good is the algorithm?

- Correctness
- Time efficiency
- Space efficiency

🔔 Does there exist a better algorithm?

- Lower bounds
 - Optimality
-

Thank You



The Abstract Data Type.

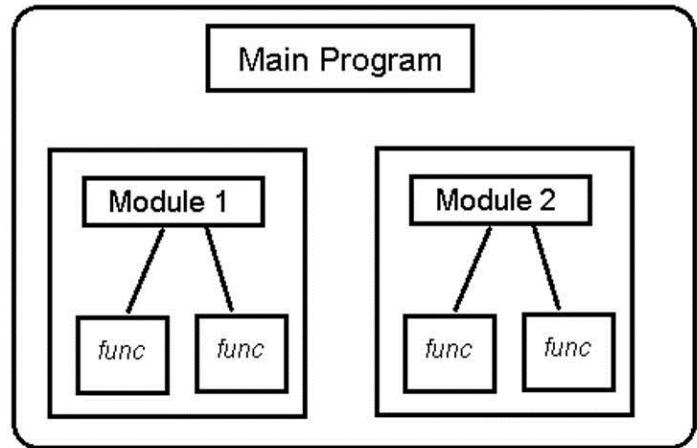
THE ADT.

```
1 C A weird program for calculating Pi written in Fortran
2 C From: Fink, D.G., Computers and the Human Mind, Anchorage, Alaska, 1973
3
4 PROGRAM PI
5 DIMENSION TERM(100)
6 N=1
7 TERM(N)=((-1)**(N+1))*(4. / (2.*N-1.))
8 N=N+1
9 IF (N>101) 3,6,6
10 N=1
11 SUM98 = SUM98+TERM(N)
12 WRITE(*,28) N, TERM(N)
13 N=N+1
14 IF (N>99) 7, 11, 11
15 SUM99=SUM98+TERM(N)
16 SUM100=SUM99+TERM(N+1)
17 IF (SUM98-3.141592) 14,23,23
18 IF (SUM99-3.141592) 23,23,15
19 IF (SUM100-3.141592) 16,23,23
20 AV89=(SUM98+SUM99)/2.
21 AV90=(SUM99+SUM100)/2.
22 COMANS=(AV89+AV90)/2.
23 IF (COMANS-3.1415920) 21,19,19
24 IF (COMANS-3.1415930) 20,21,21
25 WRITE(*,26)
26 GO TO 22
27 WRITE(*,27) COMANS
28 STOP
29 WRITE(*,25)
30 GO TO 22
31 FORMAT('ERROR IN MAGNITUDE OF SUM')
32 FORMAT('PROBLEM SOLVED')
33 FORMAT('PROBLEM UNSOLVED', F14.6)
34 FORMAT(I3, F14.6)
35 END
```

Unstructured linear programs.

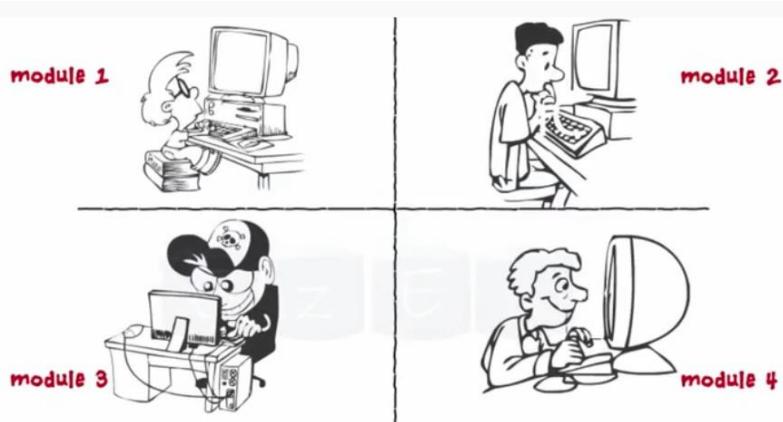


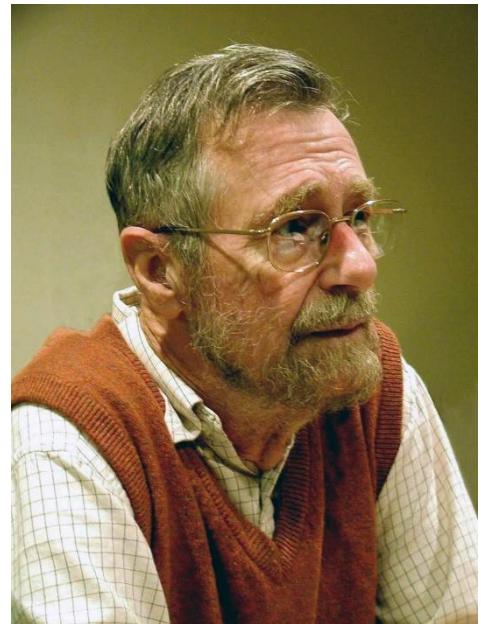
History of
programming.
SPAGHETTI CODING



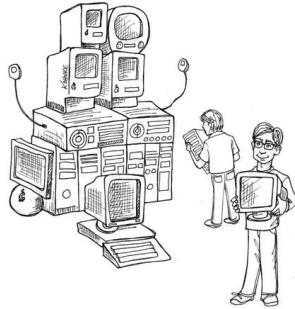
History of programming.

MODULAR PROGRAMMING





Edsger Dijkstra and Niklaus Wirth.



```
Structured:  
IF x<=y THEN  
BEGIN  
    z := y-x;  
    q :=SQRT(z);  
END  
ELSE  
BEGIN  
    z := x-y;  
    q := -SQRT(z)  
END;  
WRITELN(z,q);
```

History of programming.

STRUCTURED PROGRAMMING

Atomic data & **Composite data**

- **Atomic data** are data that consist of a single piece of information; that is, they cannot be divided into other meaningful pieces of data.
- Eg:-4 digit no
- **Composite data** can be broken out into subfields that have meaning
- Eg:- phone no, pincode

Data Type

- A **data type** consists of two parts: a set of data and the operations that can be performed on the data.
- Eg:- integer type consists of values (whole numbers in some defined range) and operations (add, subtract, multiply, divide, and any other operations appropriate for the data).

The Abstract Data Type

- An ADT consists of a **data declaration** packaged together **with the operations** that are meaningful on the data while embodying the structured principles of encapsulation and data hiding.
- In this section we define the basic parts of an ADT:
 - **Atomic and composite data**
 - **Data type**
 - **Data structure**
 - **Abstract data type**

Type	Values	Operations
integer	$-\infty, \dots, -2, -1, 0, 1, 2, \dots, \infty$	$*$, $+$, $-$, $\%$, $/$, $++$, $--$, \dots
floating point	$-\infty, \dots, 0.0, \dots, \infty$	$*$, $+$, $-$, $/$, \dots
character	$\backslash 0, \dots, 'A', 'B', \dots, 'a', 'b', \dots, \sim$	$<$, $>$, \dots

TABLE 1-1 Three Data Types

Data Structure.

- *Aggregation of atomic and composite data* into a set with defined relationships. Structure refers to a set of rules that hold the data together.
- A combination of elements in which each is either a data type or another data structure.
- A set of associations or relationship(structure) involving combined elements.

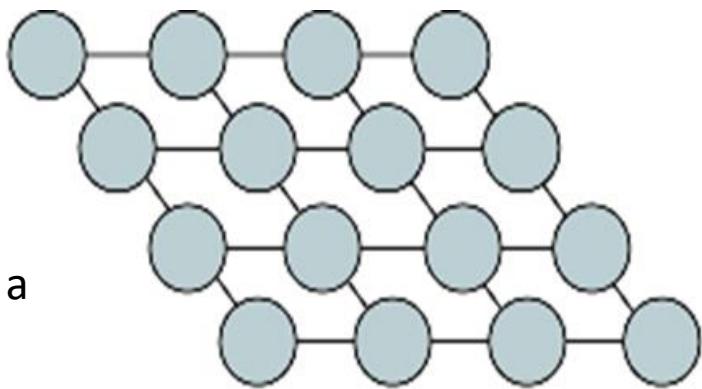
Data Structure.

Array	Record
Homogeneous sequence of data or data types known as elements	Heterogeneous combination of data into a single structure with an identified key
Position association among the elements	No association

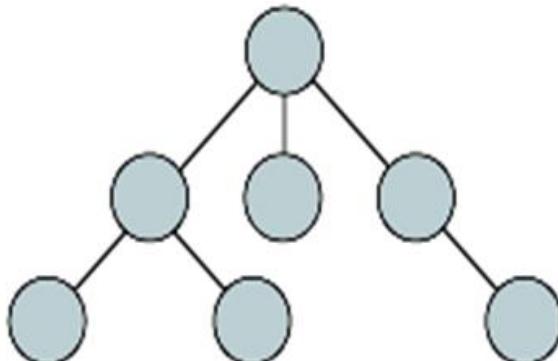
TABLE 1-2 Data Structure Examples

Four data structures
that support Use of
Lists

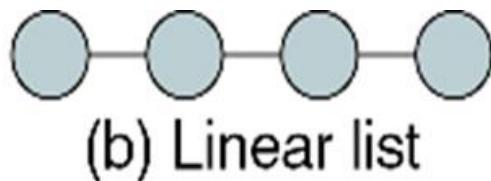
Example applications: Waiting line in a
bank (queue)



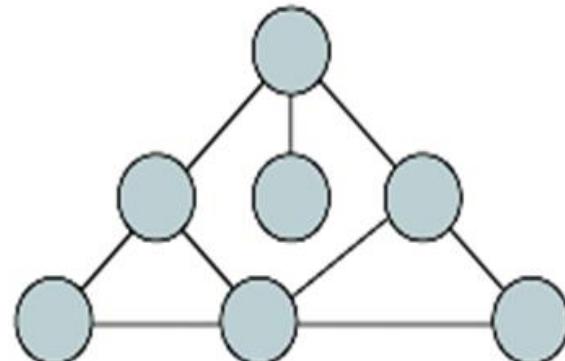
(a) Matrix



(c) Tree



(b) Linear list



(d) Graph

FIGURE 1-1 Some Data Structures

Abstract Data type.

- ADT users are NOT concerned with how the task is done but rather what it can do.
- An **abstract data type is a data declaration packaged together with the operations** that are meaningful for the data type.
- We **encapsulate the data and the operations on the data, and then hide them** from the user.
- **All references** to and manipulation of the data in a data structure are **handled through defined interfaces** to the structure.

Abstract Data type.

The concept of abstraction means:

1. We know *what* a data type can do.
2. *How* it is done is hidden.

Abstract Data Type

1. Declaration of data
2. Declaration of operations
3. Encapsulation of data and operations

Model for Abstract Data type.

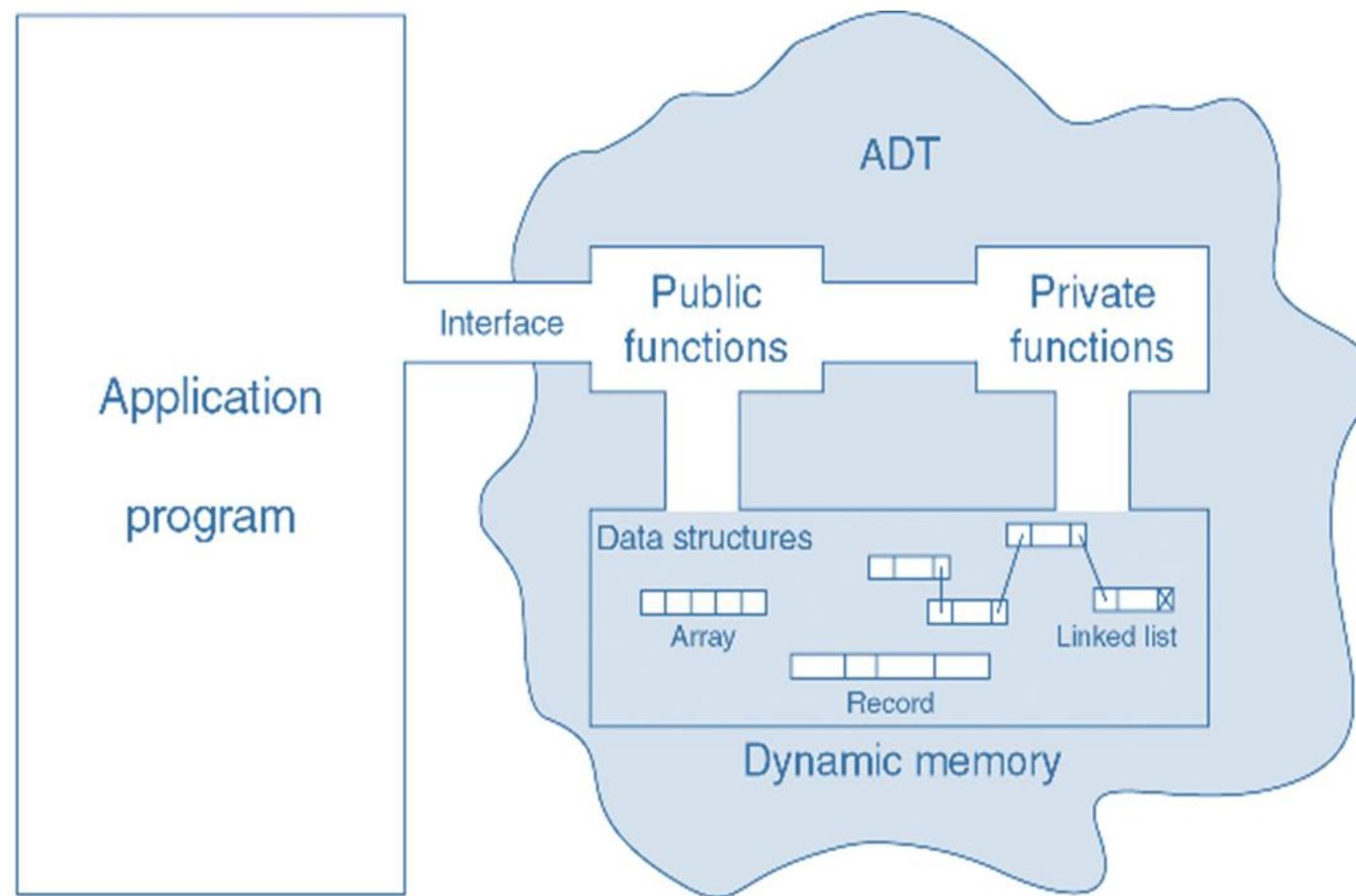


FIGURE 1-2 Abstract Data Type Model

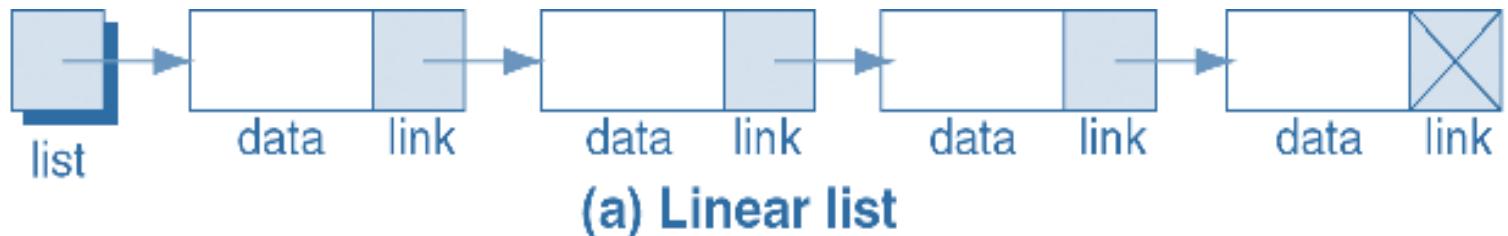
ADT Implementations

- There are two basic structures we can use to implement an ADT list: arrays and linked lists.
- In this section we discuss the basic linked-list implementation:

- **Array implementation.**

1. Sequentiality maintained by order structure of elements in the array (indexes).
2. Searching efficient.
3. Addition/deletion complex & inefficient.
4. Not preferred.

- **Linked List Implementation**



1. Ordered collection of data in which element contains the location of the next **element or elements**.
2. Each element contains two parts:
 - **data**
 - **links**
3. Data part holds application data
4. Links used to chain data together and **contain pointers** that identify next element or elements in the list.

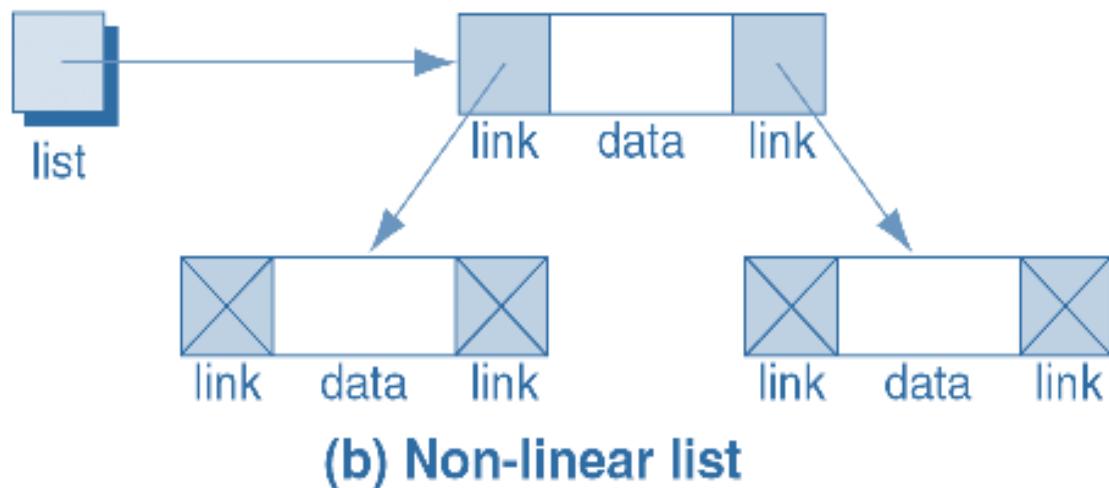


FIGURE 1-3 Linked Lists

1. Each element in the list is called a **node**.
2. Node has two parts: **data and link**.
3. Nodes are **self-referential** structures:
 - Each instance of structure contains one or more pointers to other instances of same structural type.

Data part:

- Can be a single field, multiple fields or a structure that contain several fields but acts as a single field

(a) Node in a linear list



(b) Node in a non-linear list



FIGURE 1-4 Nodes

1. Single field number and link.
2. Three fields: name, ID, gradepoints and link
3. **Recommended:** fields in structure

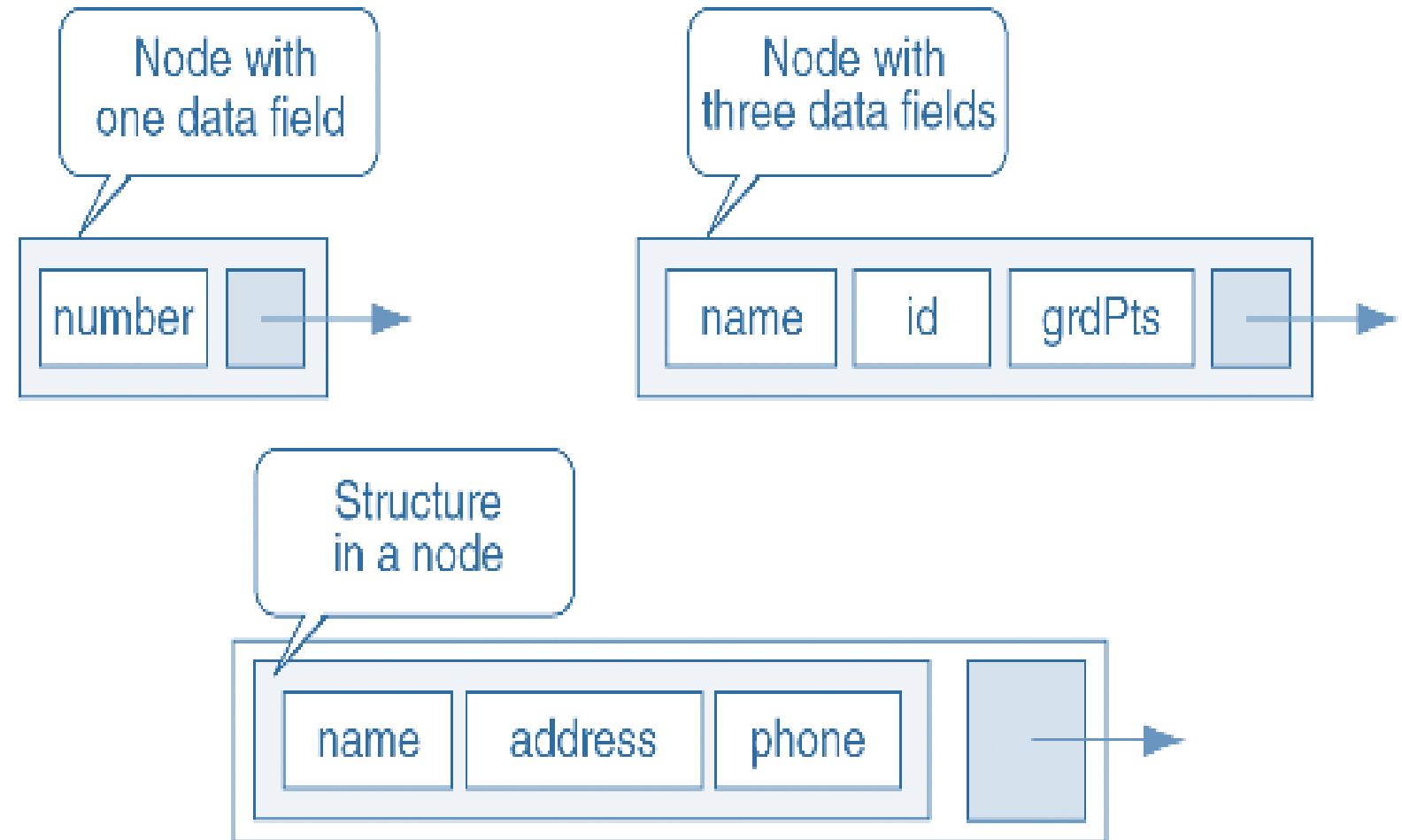


FIGURE 1-5 Linked List Node Structures

Algorithm Efficiency

Algorithmics (field):

Systematic study of the fundamental techniques used to design and analyse efficient algorithms.

- To design and implement algorithms, programmers must have a basic understanding of what constitutes good, efficient algorithms.

$$f(n) = \text{efficiency}$$

- Algorithmic efficiency defined as the function of the number of elements to be processed and the type of loop being processed.

Algorithm Efficiency: **Linear Loops**

- **Efficiency is a function of the number of instructions.**
- Loop update either adds or subtracts.

```
for ( i=0 ; i < 1000; i++)  
    application code
```

$$f(n) = n$$

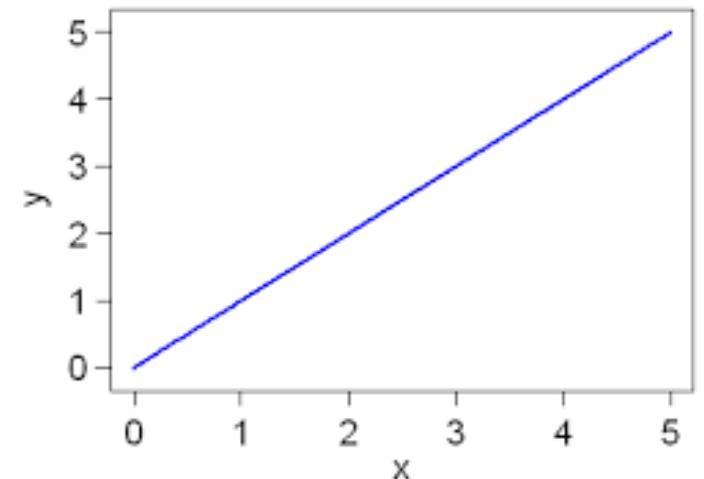
Efficiency is directly proportional
to the number of iterations

```
for ( i=0 ; i < 1000; i+=2)  
    application code
```

$$f(n) = n/2$$

Efficiency of this loop is
proportional to half the factor

Plot: Straight line



Algorithm Efficiency: **Logarithmic Loops**

- The **controlling variable is either multiplied or divided** in each iteration.
- The number of iteration is a function of the multiplier or divisor (here;2.)

Multiply loops

```
for ( i=1 ; i < 1000; i *= 2)  
    application code
```

Divide loops

```
for ( i=0 ; i < 1000; i /= 2)  
    application code
```

$$f(n) = \log n$$

multiply $2^{\text{iterations}} < 1000$
divide $1000 / 2^{\text{iterations}} \geq 1$

Analysis of multiply and divide loop

Multiply		Divide	
Iteration	Value of i	Iteration	Value of i
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

Algorithm Efficiency: Nested Loops

- The number of iterations is the total number which is the product of the number of iterations in the inner loop and number of iterations in the outer loop.

Iterations = outer loop iterations x inner loop iterations

Linear Logarithmic

```
for ( i=0 ; i < 10; i ++)  
  for ( j=0 ; j < 10; j *=2)  
    application code
```

$$f(n) = n \log n$$

Quadratic Logarithmic

```
for ( i=0 ; i < 10; i ++)  
  for ( j=0 ; j < 10; j ++)  
    application code
```

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

Dependent Quadratic

```
for ( i=0 ; i < 10; i ++)  
  for ( j=0 ; j <= i; j ++)  
    application code
```

$$f(n) = n \left(\frac{n+1}{2}\right)$$

Algorithm Efficiency: **Big-O Notation**

- Not concerned with exact measurement of efficiency but with the order of magnitude.
- This factor is the big-O, expressed as **O(n)** (**(on the order of n.)**)
 - A dominant factor determines the magnitude.
- Eg:- if an algorithm is quadratic, its efficiency is $O(n^2)$

The big-O notation can be derived from $f(n)$ using the following steps:

1. In each term, set the coefficient of the term to 1.
2. Keep the largest term in the function and discard the others. Terms are ranked from lowest to highest as shown below.

$\log n$ n $n \log n$ n^2 $n^3 \dots n^k$ 2^n $n!$

- Eg 1:- to calculate the big-O notation for

$$f(n) = n \frac{(n+1)}{2} = \frac{1}{2} n^2 + \frac{1}{2} n$$

1. first remove all coefficients

$$n^2 + n$$

2. remove the smaller factors

$$n^2$$

big-O notation is stated as

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

- Consider the polynomial expression

$$f(n) = a_j n^k + a_{j-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$$

1. First eliminate all of the coefficients

$$f(n) = n^k + n^{k-1} + \dots + n^2 + n + 1$$

2. The largest term in the above expression is the first one, so the order of a polynomial expression is:-

$$O(f(n)) = O(n^k)$$

Examples:-

1. $f(n)=3n^2+ 6n \log n + 7n + 5$

• Complexity is $O(n^2)$

2. $f(n)= n^2+ 6n$

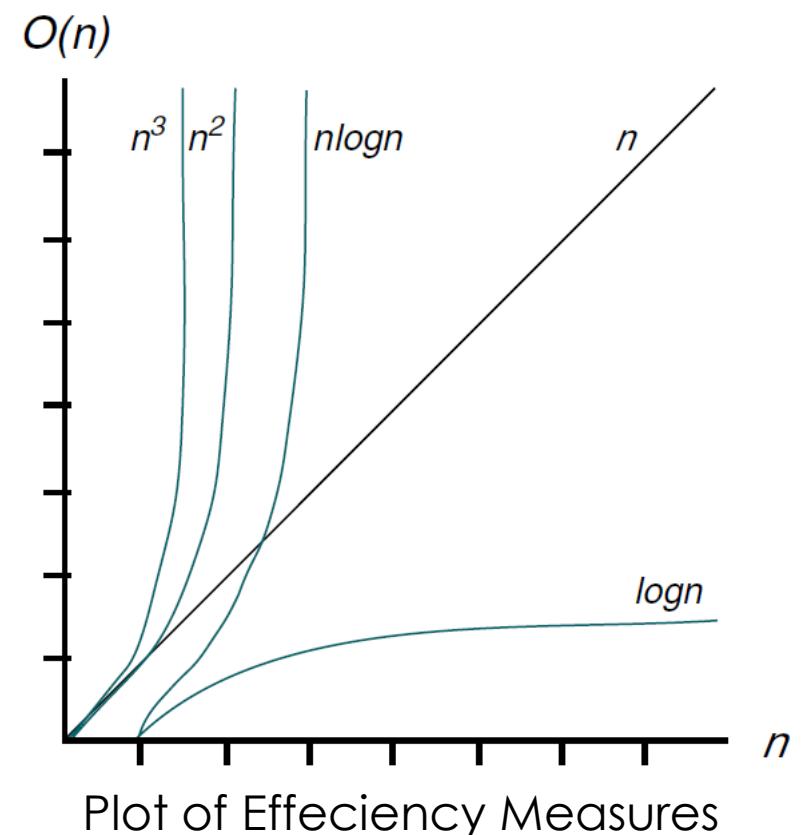
• Complexity is $O(n^2)$

3. $f(n)= 2n + 3$

• Complexity is $O(n^2)$

Standard measures of efficiency(7 categories)(order of decreasing efficiency)

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10,000^2$	minutes
Polynomial	$O(n^k)$	$10,000^k$	hours
Exponential	$O(c^n)$	$2^{10,000}$	intractable
Factorial	$O(n!)$	$10,000!$	intractable



Measures of Efficiency for $n = 10,000$

7. Reorder the following efficiencies from smallest to largest:

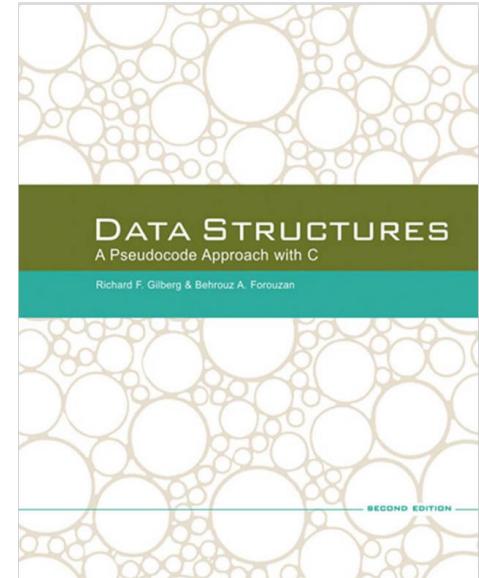
- a. $n\log(n)$
- b. $n + n^2 + n^3$
- c. 24
- d. $n^{0.5}$

8. Determine the big-O notation for the following:

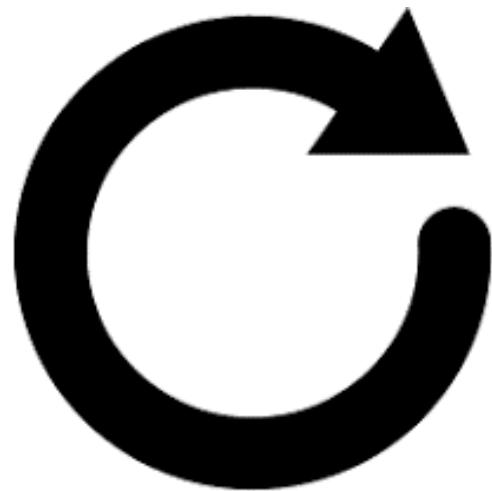
- a. $5n^{5/2} + n^{2/5}$
- b. $6\log(n) + 9n$
- c. $3n^4 + n\log(n)$
- d. $5n^2 + n^{3/2}$

- The magnitude of their efficiency for a problem containing 10,000 elements shows that the linear solution requires a fraction of a second whereas the quadratic solution requires minutes.

Review of Recursion



Two Approaches:



Iteration



Recursion

Recursion is a
repetitive process in
which an algorithm
calls itself.

Case Study:

Iterative

The definition involves only the algorithm parameter(s) and not the algorithm itself.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

FIGURE 2-1 Iterative Factorial Algorithm Definition

```
factorial(4) = 4 × 3 × 2 × 1 = 24
```

Recursive

A repetitive algorithm uses recursion whenever the algorithm appears within the definition itself.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n - 1)) & \text{if } n > 0 \end{cases}$$

FIGURE 2-2 Recursive Factorial Algorithm Definition

Recursive

Recursion is a repetitive process in which an algorithm calls itself.

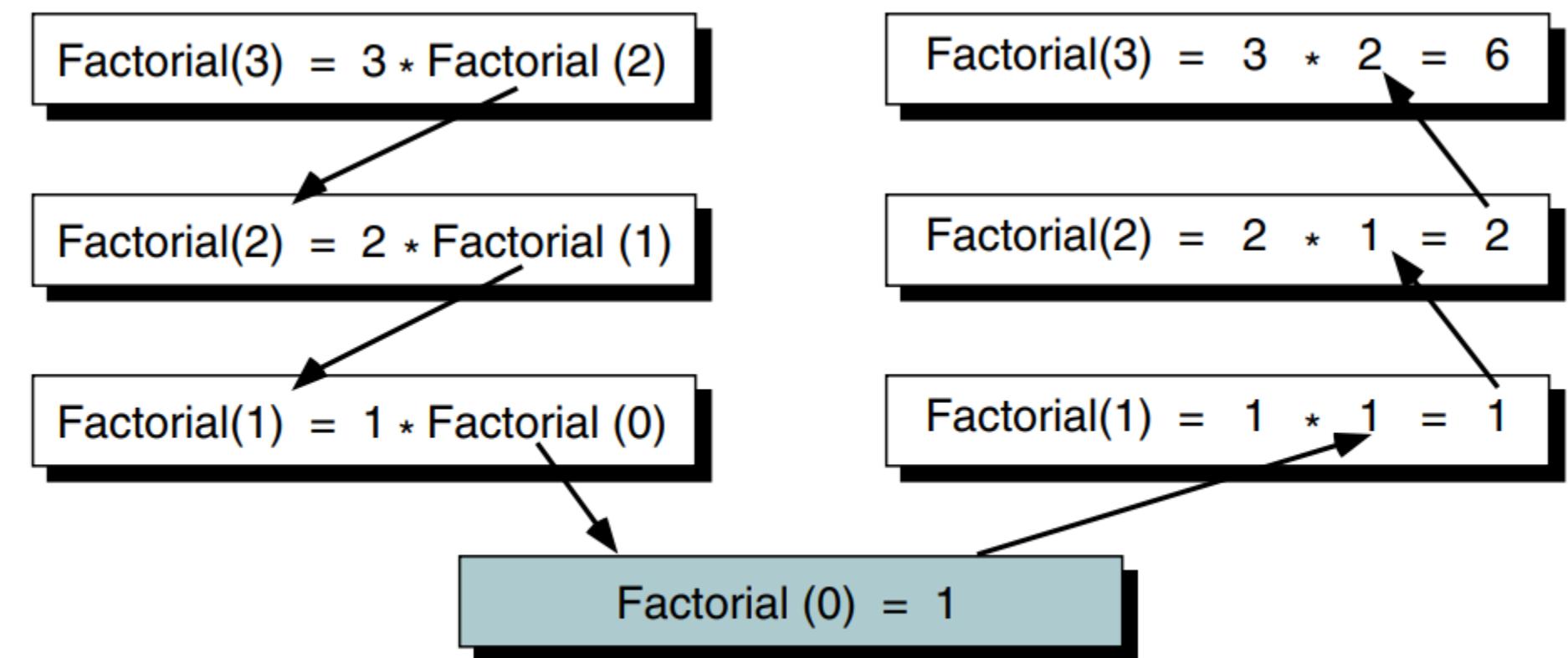


FIGURE 2-3 Factorial (3) Recursively

BE SAFE Iterative Solution

ALGORITHM 2-1 Iterative Factorial Algorithm

```
Algorithm iterativeFactorial (n)
Calculates the factorial of a number using a loop.
  Pre  n  is the number to be raised factorially
  Post n! is returned
  1 set i to 1
  2 set factN to 1
  3 loop (i <= n)
    1  set factN to factN * i
    2  increment i
  4 end loop
  5 return factN
end iterativeFactorial
```

BE SAFE Recursive Solution

ALGORITHM 2-2 Recursive Factorial

```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
    Pre    n  is the number being raised factorially
    Post   n! is returned
1  if (n equals 0)
    1  return 1
2  else
    1  return (n * recursiveFactorial (n - 1))
3  end if
end recursiveFactorial
```

Which code is simpler?

Which one does not have a loop?

Recursive Solution Analysis

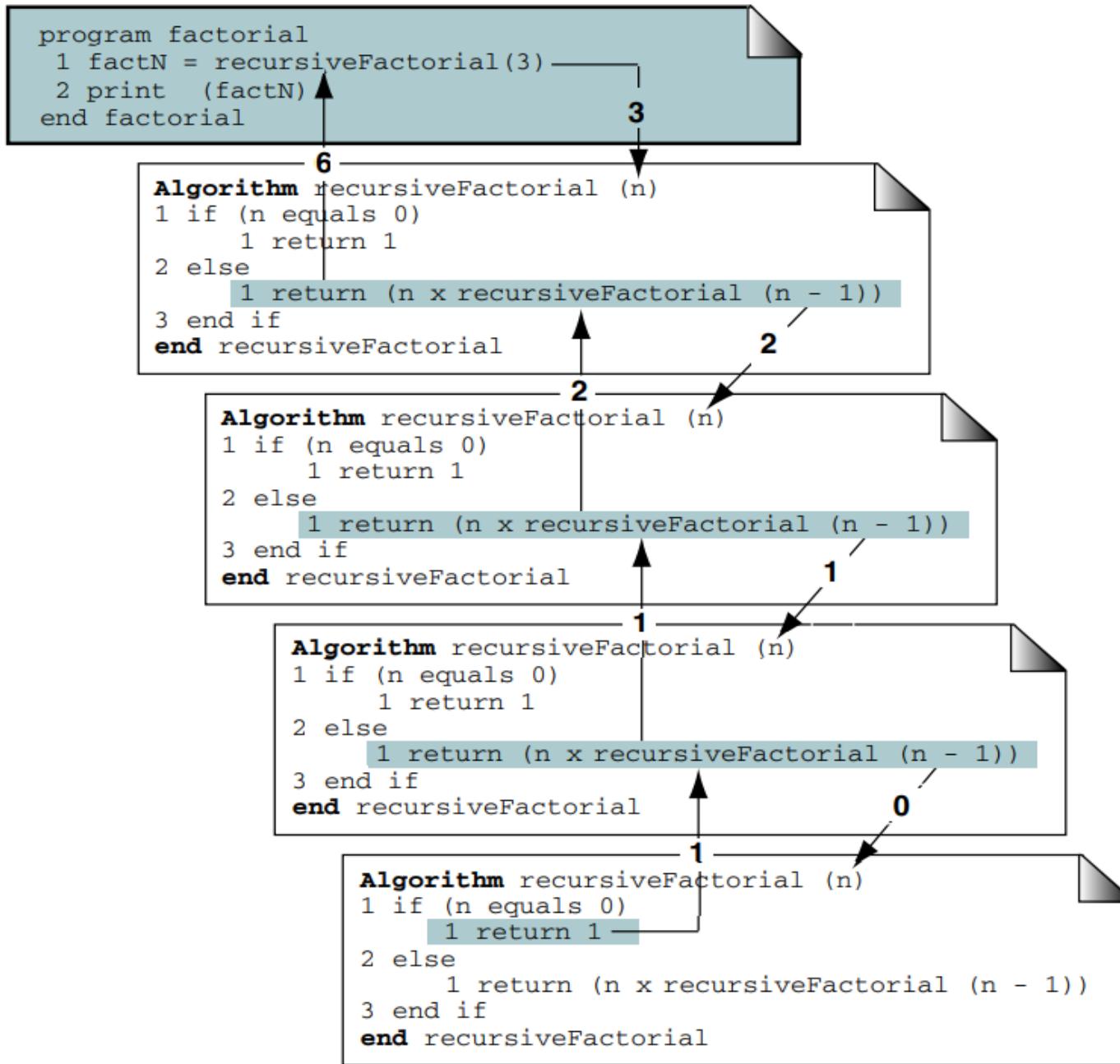


FIGURE 2-4 Calling a Recursive Algorithm

Analytic Approach:

2. Limitations of Recursion



Every recursive call **either solves a part** of the problem or it **reduce the size** of the problem.

- Base case
 - The statement that “solves” the problem.
 - Every recursive algorithm must have a base case.
- General case
 - The rest of the algorithm
 - Contains the logic needed to reduce the size of the problem.

Combine the base case and the general cases into an algorithm

Rules for designing a recursive algorithm:

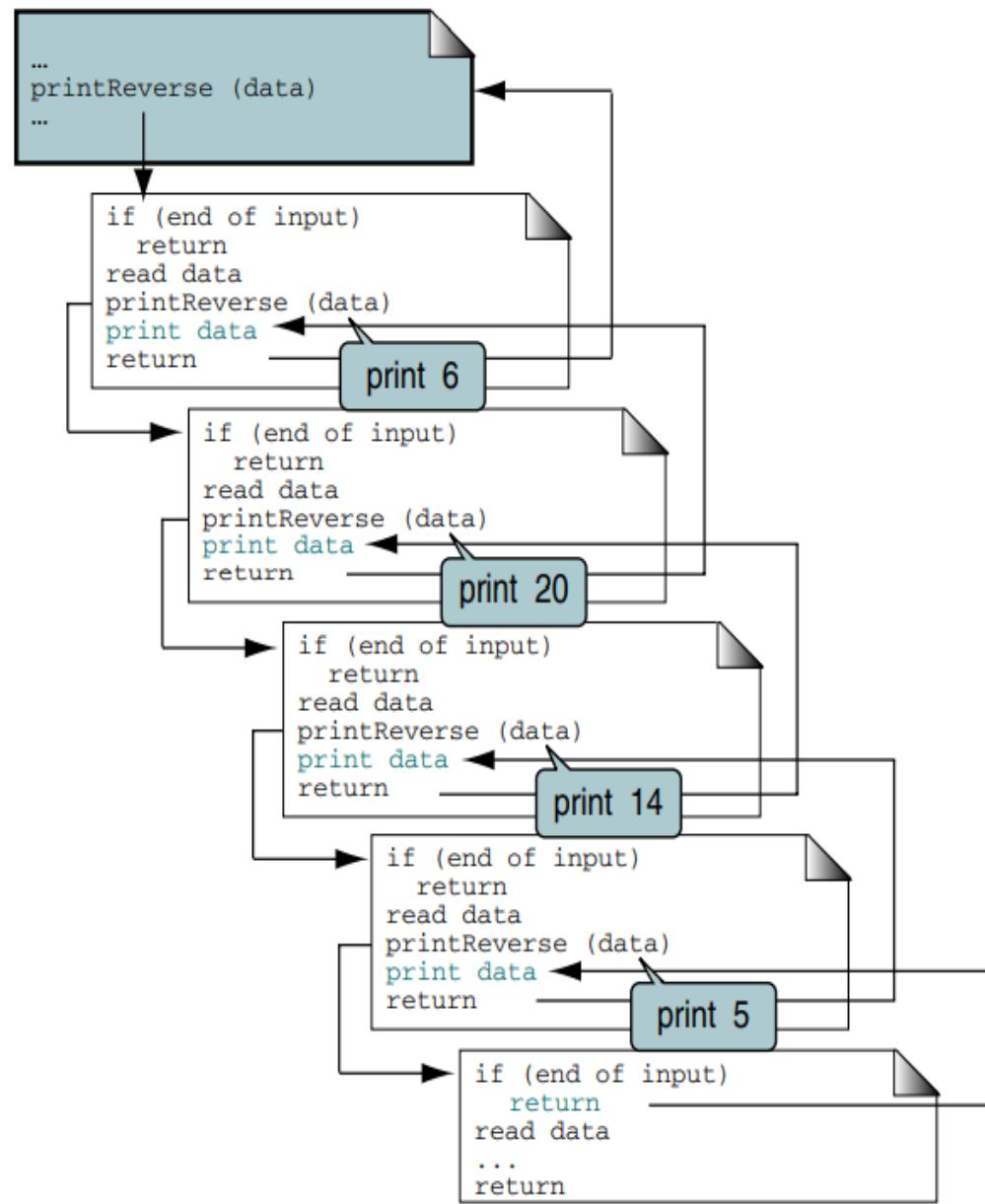
1. First, determine the base case.
2. Then determine the general case.
3. Combine the base case and the general cases into an algorithm
 - Each call must reduce the size of the problem and move it toward the base case.
 - The base case, when reached, must terminate without a call to the recursive algorithms; that is, it must execute a return.

DESIGN IMPLEMENTATION.

ALGORITHM 2-3 Print Reverse

```
Algorithm printReverse (data)
Print keyboard data in reverse.
    Pre nothing
    Post data printed in reverse
1 if (end of input)
1   return
2 end if
3 read data
4 printReverse (data)
Have reached end of input: print nodes
5 print data
6 return
end printReverse
```

DESIGN IMPLEMENTATION



Recursive calls (reads)

6
data

20
data

14
data

5
data

FIGURE 2-5 Print Keyboard Input in Reverse

Limitations

- Recursive solutions may involve extensive overhead (both time and memory) because they use calls.
- Each call takes time to execute.
- A recursive algorithm therefore generally runs more slowly than its non-recursive implementation.
- As a general rule, recursive algorithms should be used only when their efficiency is logarithmic.

Do not use recursion if answer is NO to any question below:

1. Is the algorithm or data structure naturally suited to recursion?
2. Is the recursive solution shorter and more understandable?
3. Does the recursive solution run within acceptable time and space limits?

SOME MORE EXAMPLES.

 Greatest Common Divisor

 Fibonacci Numbers

 Prefix to Postfix Conversion

 The Towers of Hanoi

SOME MORE Recursive Examples: GCD.

$$\text{gcd } (a, b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ \text{gcd } (b, a \bmod b) & \text{otherwise} \end{cases}$$

Greatest Common Divisor Recursive Definition

SOME MORE Recursive Examples: GCD.

Euclidean Algorithm for Greatest Common Divisor

```
Algorithm gcd (a, b)
Calculates greatest common divisor using the Euclidean algorithm.

    Pre a and b are positive integers greater than 0
    Post greatest common divisor returned

1 if (b equals 0)
    1 return a
2 end if
3 if (a equals 0)
    2 return b
4 end if
5 return gcd (b, a mod b)
end gcd
```

```
9 // Prototype Statements
10 int gcd (int a, int b);
11
12 int main (void)
13 {
14 // Local Declarations
15     int gcdResult;
16
17 // Statements
18     printf("Test GCD Algorithm\n");
19
20     gcdResult = gcd (10, 25);
21     printf("GCD of 10 & 25 is %d", gcdResult);
22     printf("\nEnd of Test\n");
23
24 } // main
```

```
25 /* ===== gcd =====
26     Calculates greatest common divisor using the
27     Euclidean algorithm.
28     Pre a and b are positive integers greater than 0
29     Post greatest common divisor returned
30 */
31 int gcd (int a, int b)
32 {
33     // Statements
34     if (b == 0)
35         return a;
36     if (a == 0)
37         return b;
38     return gcd (b, a % b);
39 } // gcd
```

Results:

```
Test GCD Algorithm
GCD of 10 & 25 is 5
End of Test
```

SOME MORE Recursive Examples: Fibonacci Numbers

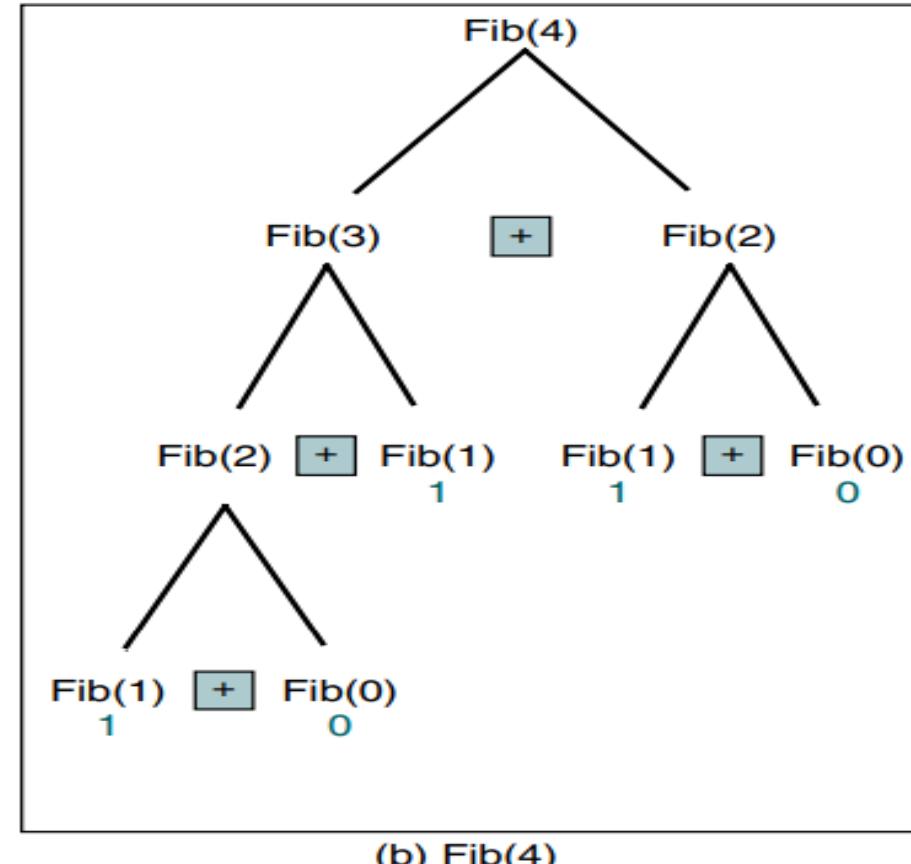
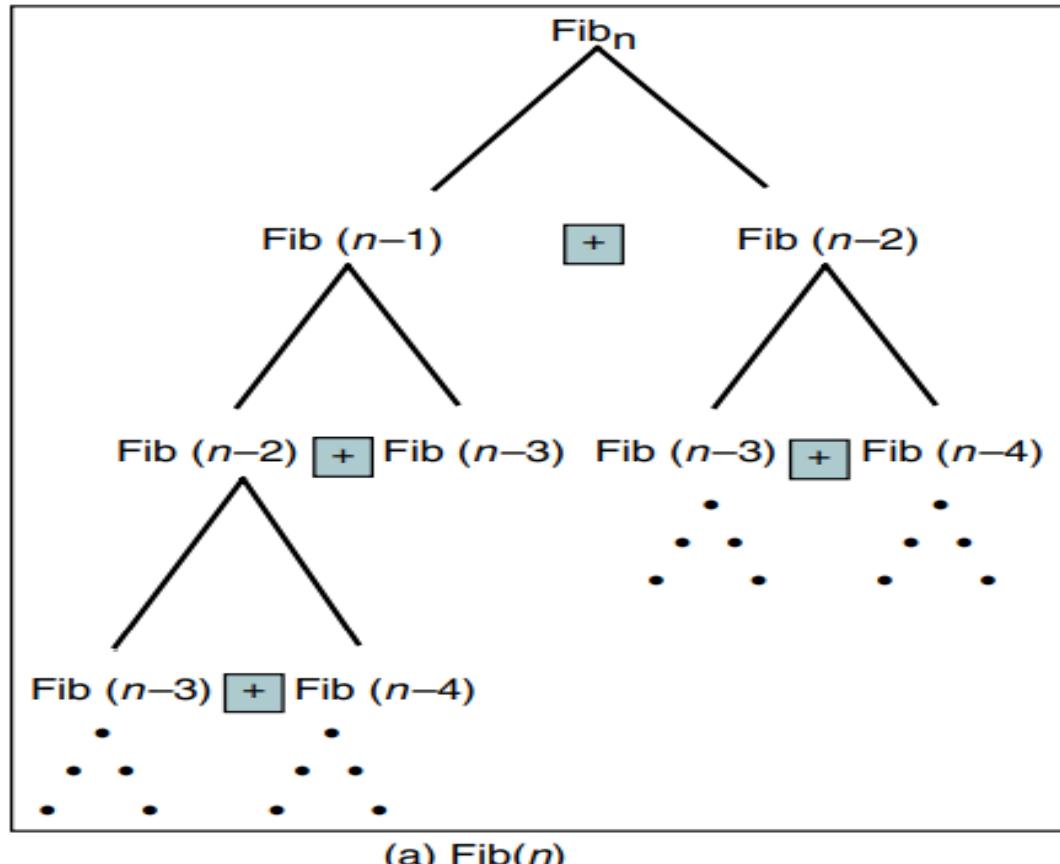
$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2) & \text{otherwise} \end{cases}$$

Fibonacci Numbers Recursive Definition

Leonardo Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci series: Design.



Fibonacci numbers

```
7 // Prototype Statements
8 long fib (long num);
9
10 int main (void)
11 {
12 // Local Declarations
13     int seriesSize = 10;
14
15 // Statements
16 printf("Print a Fibonacci series.\n");
17
18     for (int looper = 0; looper < seriesSize; looper++)
19     {
20         if (looper % 5)
21             printf(", %8ld", fib(looper));
22         else
23             printf("\n%8ld", fib(looper));
24     } // for
25     printf("\n");
26     return 0;
27 } // main
```

```
-- 29     /* ===== fib ===== */
30     Calculates the nth Fibonacci number
31     Pre  num identifies Fibonacci number
32     Post returns nth Fibonacci number
33 */
34     long fib (long num)
35     {
36 // Statements
37     if (num == 0 || num == 1)
38         // Base Case
39         return num;
40     return (fib (num - 1) + fib (num - 2));
41 } // fib
```

Results:

Print a Fibonacci series.

0,	1,	1,	2,	3
5,	8,	13,	21,	34

Fibonacci numbers

fib(<i>n</i>)	Calls	fib(<i>n</i>)	Calls
1		1	287
2		3	465
3		5	753
4		9	1219
5		15	1973
6		25	21,891
7		41	242,785
8		67	2,692,573
9		109	29,860,703
10		177	331,160,281

Prefix to Postfix Conversion

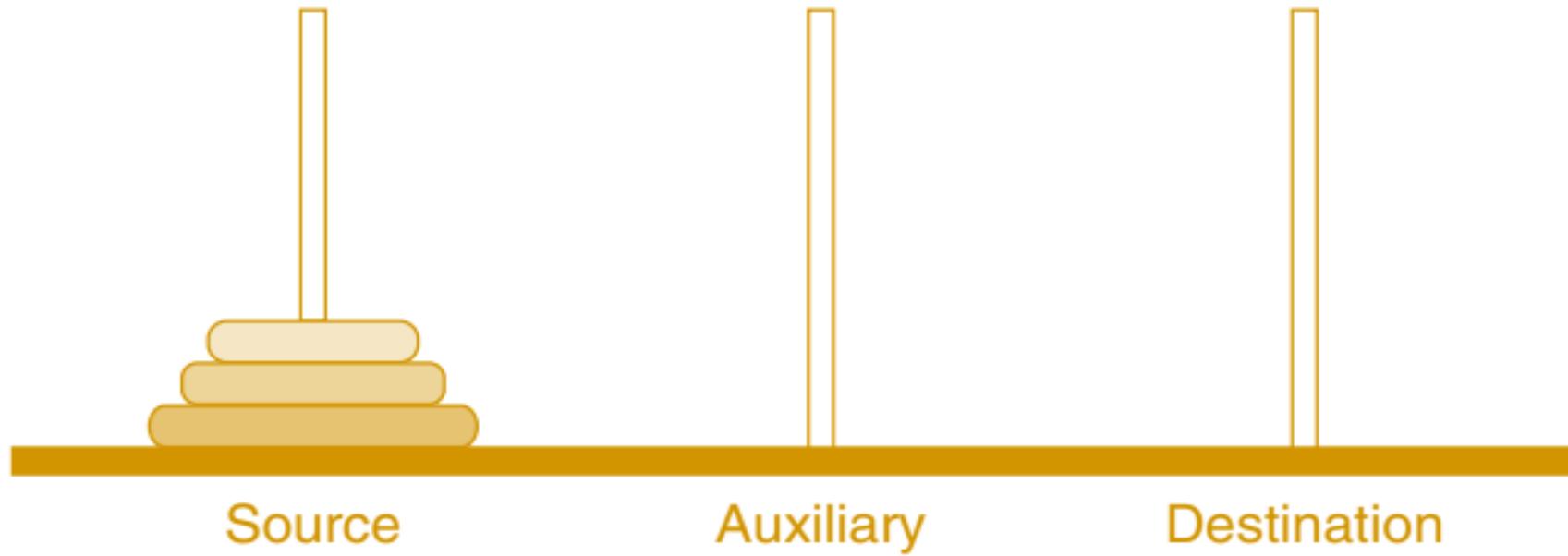
An arithmetic expression can be represented in three different formats:

1. Infix
2. Postfix
3. Prefix

Prefix:	+	A	B
Infix:	A	+	B
Postfix:	A	B	+

1. **Prefix notation:** Operator **comes before** the operands.
2. **Infix notation:** Operator **comes between** the operands.
3. **Postfix notation:** Operator **comes after** the operands

The Towers of Hanoi

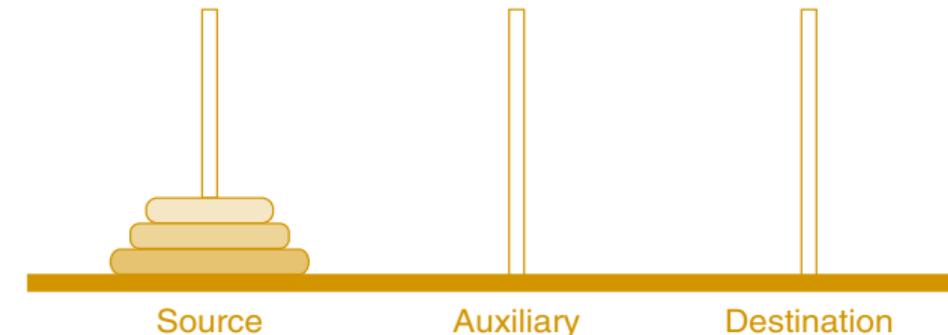


Towers of Hanoi—Start Position

According to the legend, the monks in a remote mountain monastery knew how to predict when the world would end.

The Towers of Hanoi

They had a set of **three diamond needles**. Stacked on the **first diamond needle** were **64 gold disks of decreasing size**.



Towers of Hanoi—Start Position

The legend said that when all 64 disks had been transferred to the destination needle, the stars would be extinguished and *the world would end*.

Today we know we need to have $2^{64} - 1$ moves to move all the disks.

RULES



Case: Hanoi Towers

The monks moved one disk to another needle each hour, subject to the following rules:

1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. One and only one auxiliary needle could be used for the intermediate storage of disks.

This problem is interesting for two reasons.

1. Recursive solution is much easier to code than the iterative solution would be, as is often the case with good recursive solutions.
2. Its solution pattern is different from the simple examples we have been discussing

DESIGN



CASE 1: ONE DISK

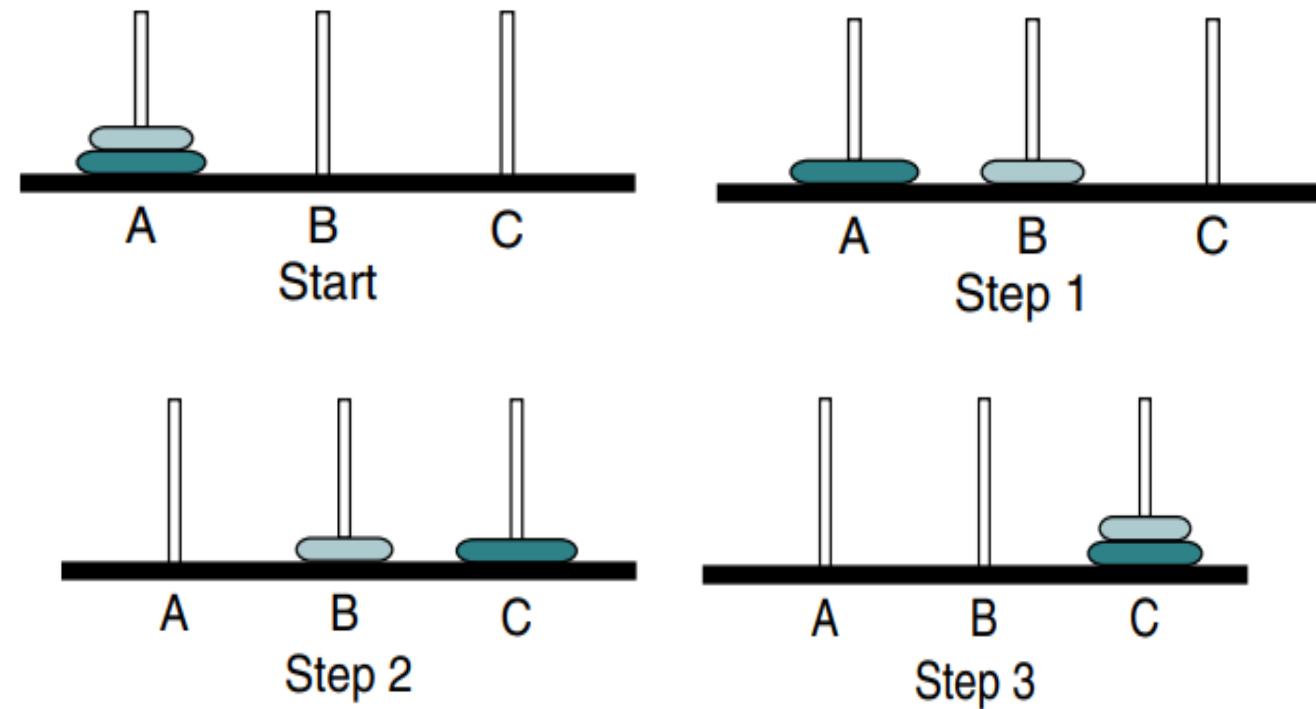
Move one disk from source to destination needle

DESIGN



CASE 1: TWO DISKS

1. Move one disk to auxiliary needle.
2. Move one disk to destination needle
3. Move one disk from auxiliary to destination needle.



Towers Solution for Two Disks



DESIGN

CASE 3: THREE DISKS

1. Move TWO disks from source to auxiliary.
2. Move one disk from source to destination.
3. Move TWO disks from auxiliary to destination.

DESIGN



CASE 3: “n” DISKS

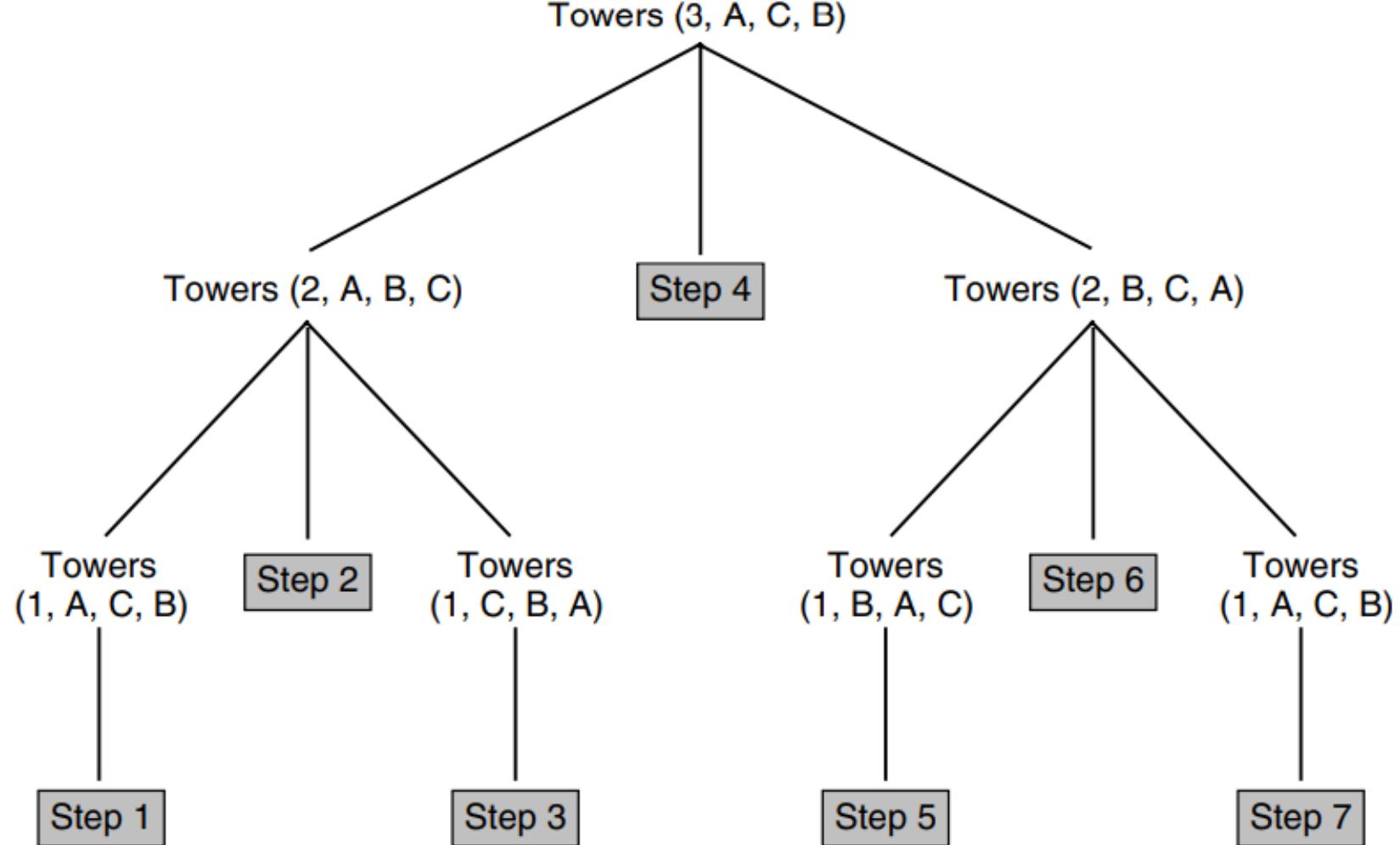
1. Move $n - 1$ disks from source to auxiliary.
2. Move one disk from source to destination.
3. Move $n - 1$ disks from auxiliary to destination.

General case

Base case

General case

Algorithm: Towers of Hanoi



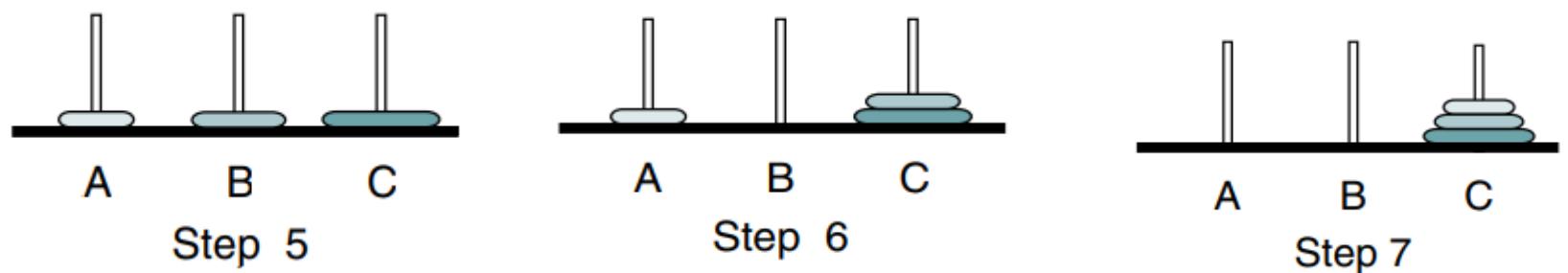
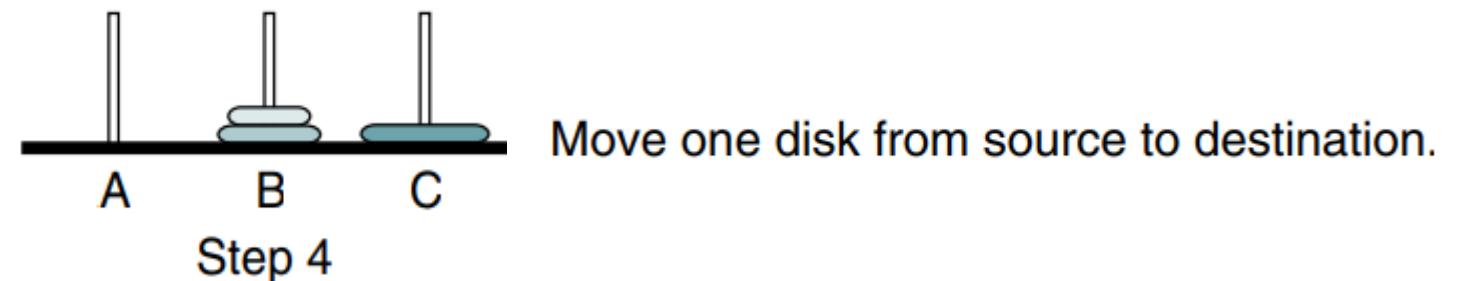
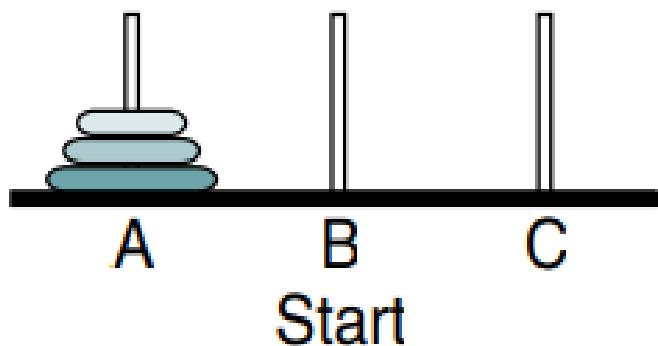
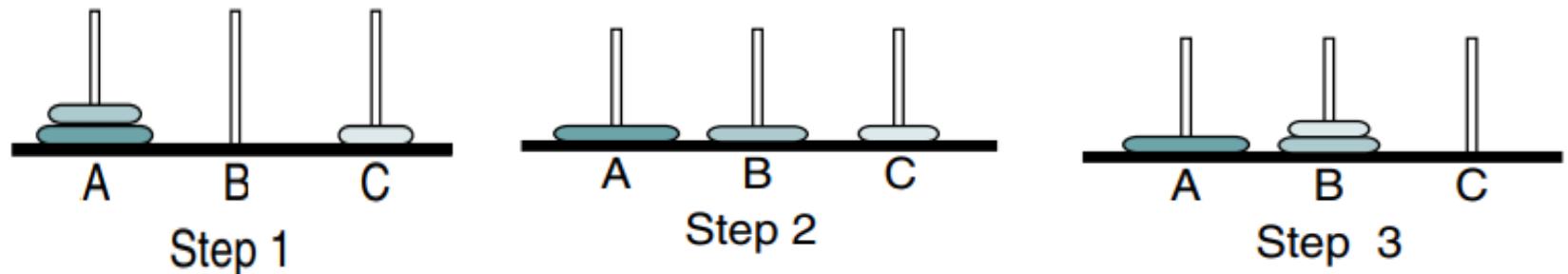
1. Move $n - 1$ disks from source to auxiliary.
2. Move one disk from source to destination.
3. Move $n - 1$ disks from auxiliary to destination.

```
Algorithm towers (numDisks, source, dest, auxiliary)
```

1. Call `Towers (n - 1, source, auxiliary, destination)`
2. Move one disk from source to destination
3. Call `Towers (n - 1, auxiliary, destination, source)`

DESIGN

CASE 3: THREE DISKS



Towers of Hanoi

```
Algorithm towers (numDisks, source, dest, auxiliary)
  Recursively move disks from source to destination.
  Pre numDisks is number of disks to be moved
    source, destination, and auxiliary towers given
  Post steps for moves printed
1 print("Towers: ", numDisks, source, dest, auxiliary)
2 if (numDisks is 1)
  1 print ("Move from ", source, " to ", dest)
3 else
  1 towers (numDisks - 1, source, auxiliary, dest, step)
  2 print ("Move from " source " to " dest)
  3 towers (numDisks - 1, auxiliary, dest, source, step)
4 end if
end towers
```

Algorithm:
Towers of Hanoi.

Calls:

Towers (3, A, C, B)

Towers (2, A, B, C)

Towers (1, A, C, B)

Towers (1, C, B, A)

Towers (2, B, C, A)

Towers (1, B, A, C)

Towers (1, A, C, B)

Output:

Move from A to C

Move from A to B

Move from C to B

Move from A to C

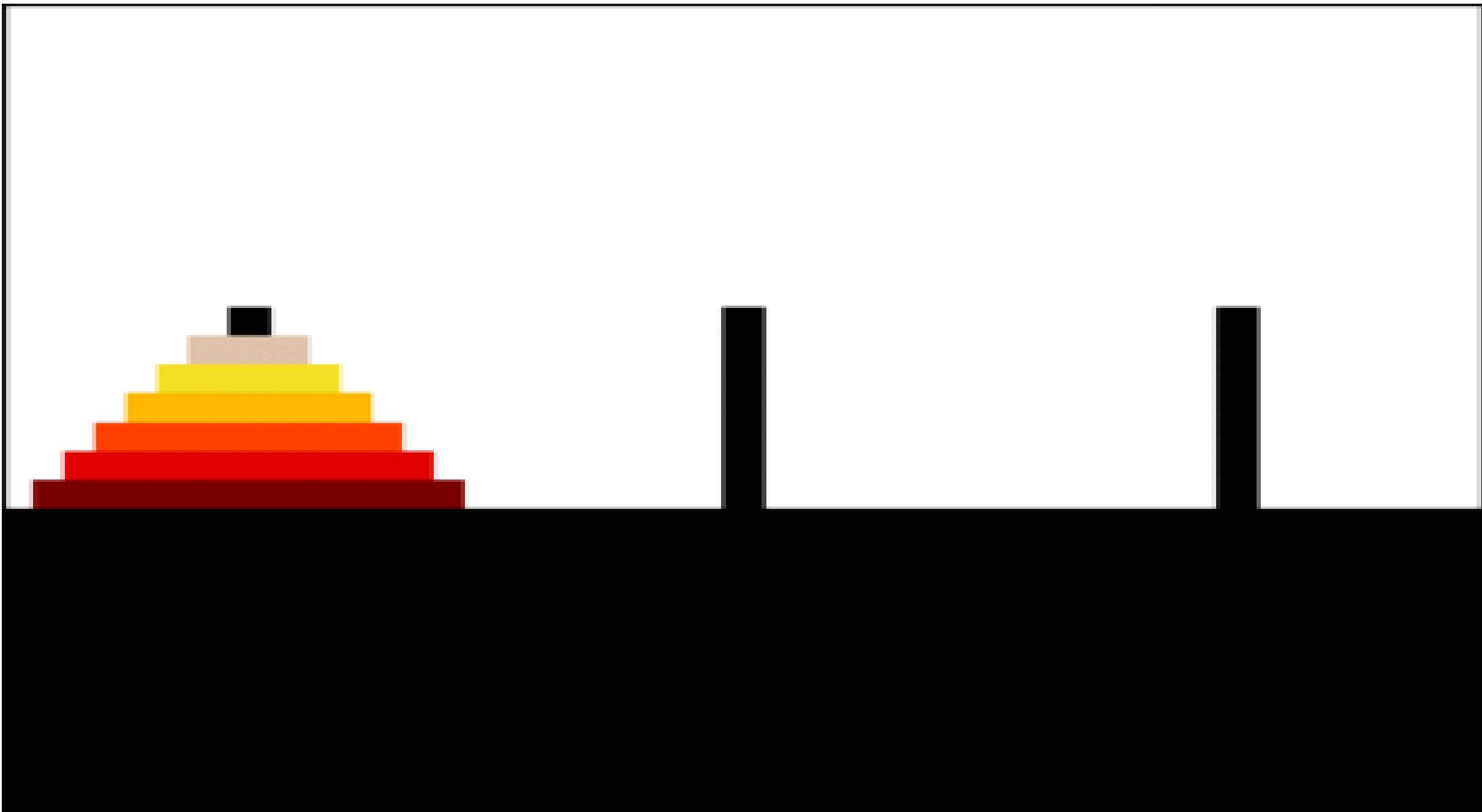
Move from B to A

Move from B to C

Move from A to C

Tracing Algorithm 2-7, Towers of Hanoi

```
Algorithm towers (numDisks, source, dest, auxiliary)
    print("Towers: ", numDisks, source, dest, auxiliary)
    if (numDisks is 1)
        1   print ("Move from ", source, " to ", dest)
    else
        1   towers (numDisks - 1, source, auxiliary, dest, s)
        2   print ("Move from " source " to " dest)
        3   towers (numDisks - 1, auxiliary, dest, source, s)
    end if
```



Iteration

- Uses loops calls
- Counter controlled and body of loop terminates when the termination condition fails.
- Execution is faster and takes less space.
- Difficult to design for some problems.

Recursion

uses if-else and repetitive function

Terminates when base condition is reached.

Consumes time and space because of push and pop.

Best suited for some problems and easy to design.

Advantages of Recursion

1. Clearer and simpler versions of algorithms can be created using recursion.
2. Recursive definition of a problem can be easily translated into a recursive function.
3. Lot of bookkeeping activities such as initialization etc required in iterative solution is avoided.

Disadvantages

1. When a function is called, the function saves formal parameters, local variables and return address and hence consumes a lot of memory.
2. Lot of time is spent in pushing and popping and hence consumes more time to compute result.

Recursive chains

- Recursive function need not call itself directly. It can call itself indirectly as shown

A(parameters)

{

.....
.....

B(arguments)

}

B(parameters)

{

.....
.....

A(arguments)

}



Exercise

1. Consider the following algorithm:

```
algorithm fun1 (x)
1 if (x < 5)
  1 return (3 * x)
2 else
  1 return (2 * fun1 (x - 5) + 7)
3 end if
end fun1
```

What would be returned if `fun1` is called as

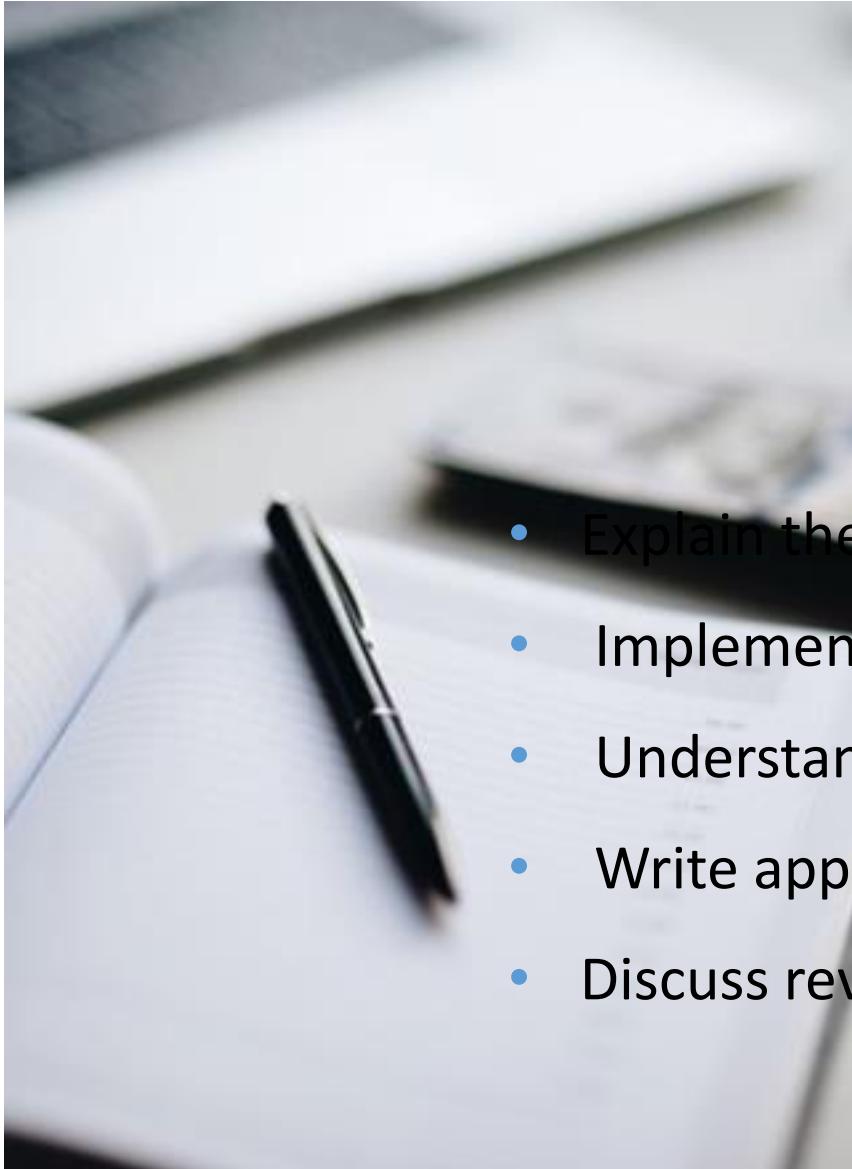
- a. `fun1 (4)?`
- b. `fun1 (10)?`
- c. `fun1 (12)?`

OF RECURSION.

STACKS

The LIFO structure





-
- Explain the design, use, and operation of a stack
 - Implement a stack using a linked list structure
 - Understand the operation of the stack ADT
 - Write application programs using the stack ADT
 - Discuss reversing data, parsing, postponing and backtracking

Stacks

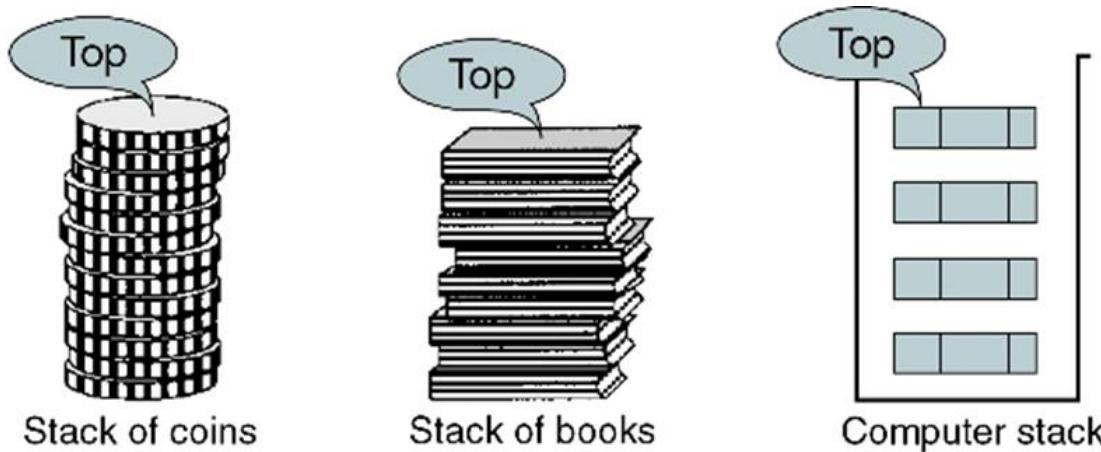


FIGURE 3-1 Stack

🔓 A stack is a **linear list** in which all additions and deletions are restricted to one end, called the top.

🔓 **Last in – first out (LIFO) data structure**

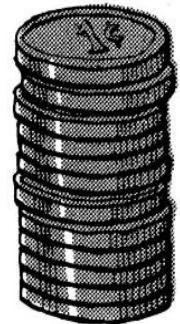
Stack Basics

- Stack is usually implemented as a list, with additions and removals taking place at one end of the list
- The **active end** of the list implementing the stack is the **top of the stack**
- Stack types:
 - **Static** – fixed size, often implemented using an array
 - **Dynamic** – size varies as needed, often implemented using a linked list

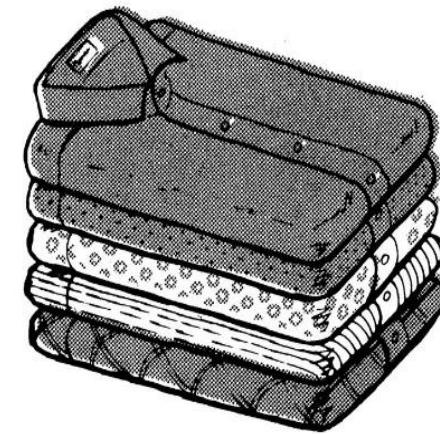
A stack of
cafeteria trays



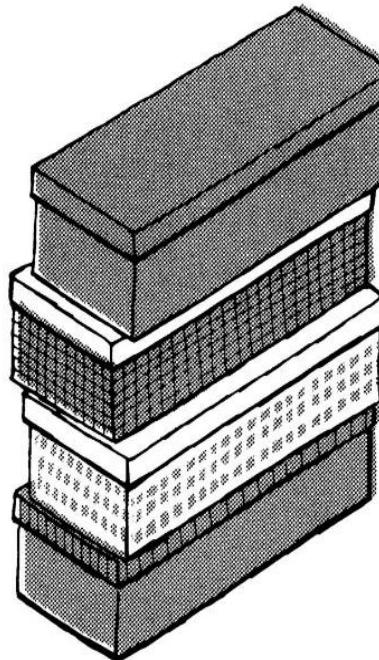
A stack
of pennies



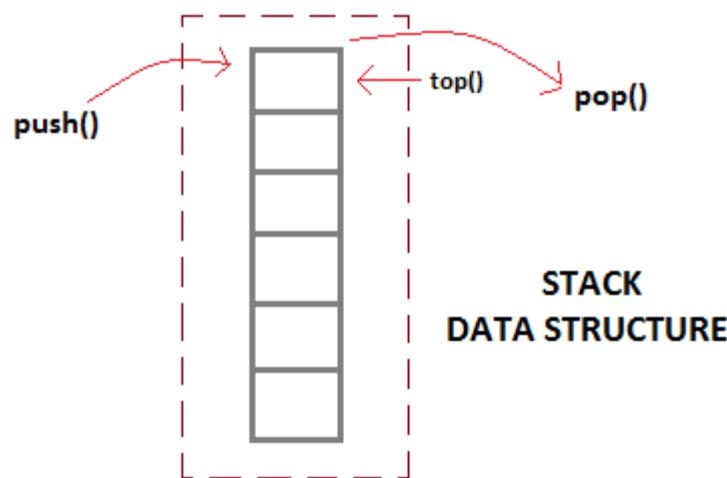
A stack of
neatly folded shirts



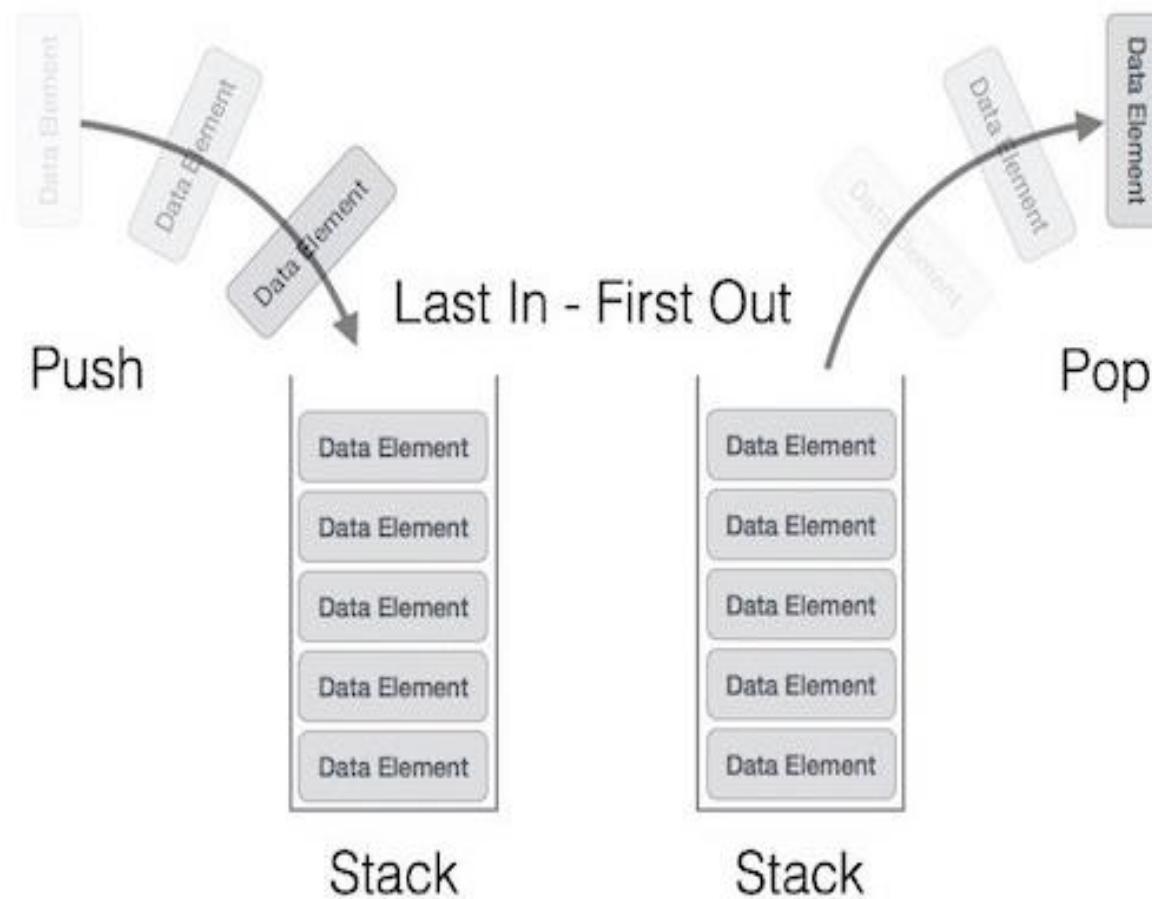
A stack of shoe boxes



Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack.

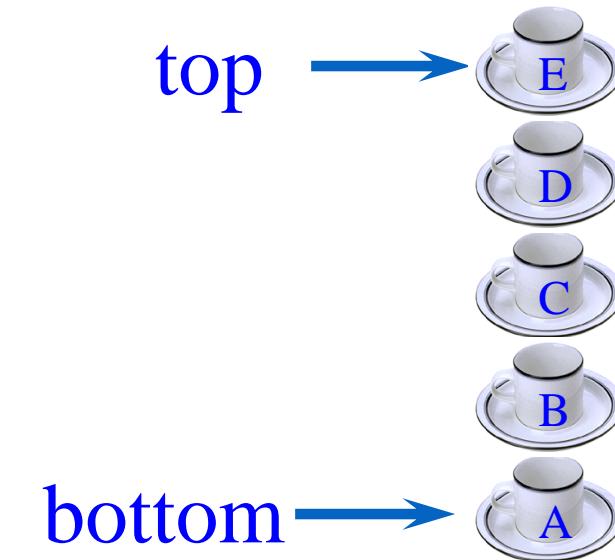


Stack Representation



Stacks fundamentals

- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from
the top end only.



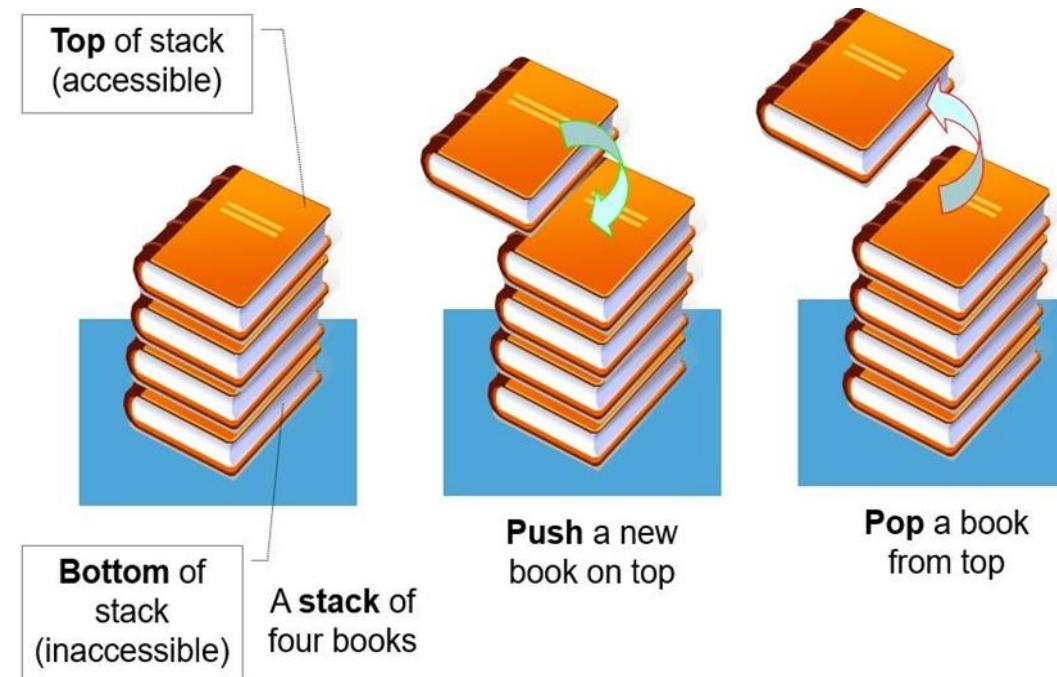
- Insert at top of stack and remove from top of stack
- Stack operations also called Last-In First-Out (LIFO)

Where can I use a stack data structure?

- 📍 Page-visited history in a Web browser
- 📍 Undo sequence in a text editor
- 📍 Saving local variables when one function calls another, and this one calls another

Stack Operations: Push and Pop

- **Push:** insert at the top/beginning of stack
- **Pop:** delete from the top/beginning of stack



Stack Operations: Push and Pop

Syntax :

stackname.push(value)

Parameters : The value of the element to be inserted is passed as the parameter.

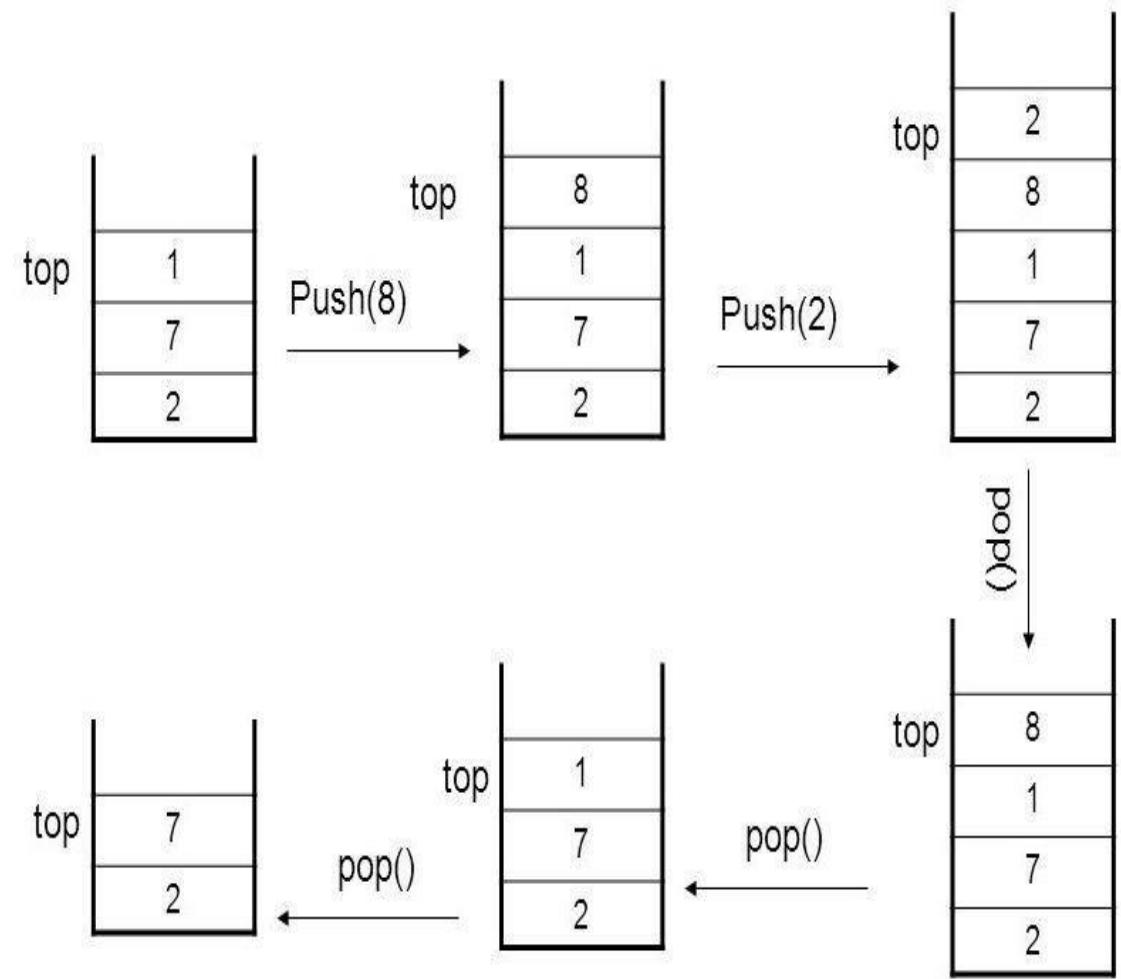
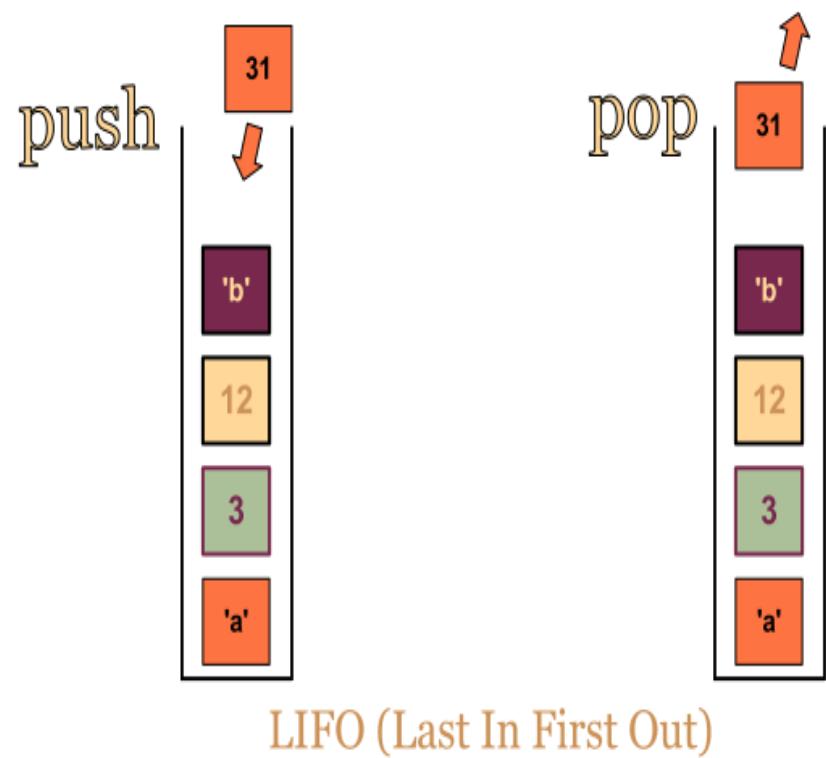
Result : Adds an element of value same as that of the parameter passed at the top of the stack.

stackname.pop()

Parameters : No parameters are passed.

Result : Removes the newest element in the stack or basically the top element.

STACK



Stack Operations: Push and Pop

check the status of stack

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

Stack Operations: Push and Pop

Algorithm of peek() function

```
begin procedure peek  
    return stack[top]  
end procedure
```

Algorithm of isfull() function

```
begin procedure isfull  
    if top equals to MAXSIZE  
        return true  
    else  
        return false  
    endif  
end procedure
```

```
int peek()  
{  
    return stack[top];  
}
```

```
bool isfull()  
{  
    if(top == MAXSIZE)  
        return true;  
    else  
        return false;  
}
```

Stack Operations: Push and Pop

isempty()

```
begin procedure isempty
    if top less than 1
        return true
    else return false
endif
end procedure
```

```
bool isempty()
{
    if (top == -1)
        return true;
    else
        return false;
}
```

Stack Operations: Push and Pop

Push Operation

The process of inserting a new data element onto stack is known as a Push Operation.

Push operation involves a series of steps –

- Step 1 – Checks if the stack is full.
- Step 2 – If the stack is full, produces an error and exit.
- Step 3 – If the stack is not full, increments top to point next empty space.
- Step 4 – Adds data element to the stack location, where top is pointing.
- Step 5 – Returns success.

Stack Operations: Push and Pop

Algorithm for PUSH and POP Operation

```
begin procedure push: stack, data  
    if stack is full  
        return null  
    endif  
    top ← top + 1  
    stack[top] ← data  
end procedure
```

```
begin procedure pop: stack  
    if stack is empty  
        return null  
    endif  
    data ← stack[top]  
    top ← top - 1  
    return data  
end procedure
```

Exercise: Stacks

- Describe the output of the following series of stack operations
 - Push(8)
 - Push(3)
 - Pop()
 - Push(2)
 - Push(5)
 - Pop()
 - Pop()
 - Push(9)
 - Push(1)

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List.

- **Arrays are quick, but are limited in size.**

Pros: Easy to implement. Memory saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

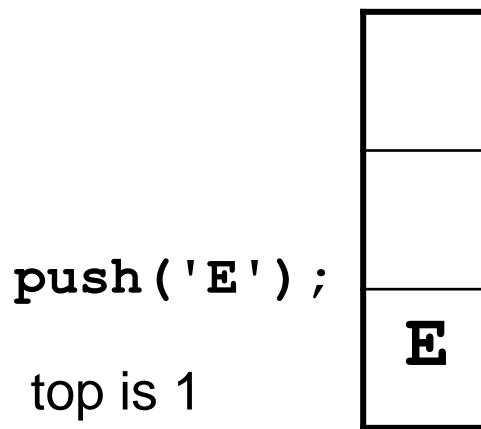
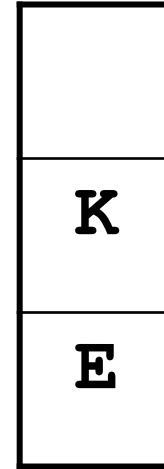
- **Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.**

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.

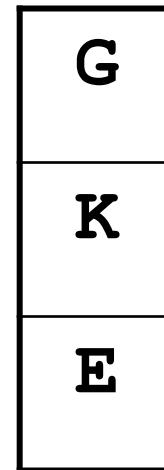
Cons: Requires extra memory due to involvement of pointers.

Array Implementation Example

- This stack has max capacity 3, initially top = 0 and stack is empty.



push ('K');
top is 2

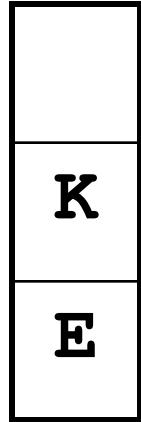


push ('G');
top is 3

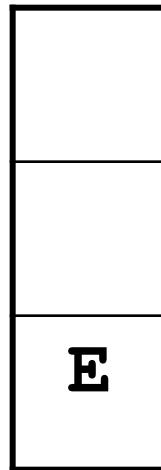
Stack Operations Example

After three pops, **top == 0** and the stack is empty

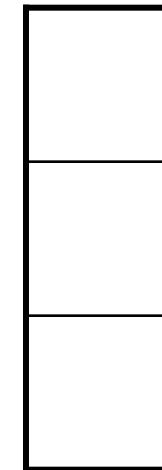
pop () ;
(remove G)



pop () ;
(remove K)



pop () ;
(remove E)



Analysis of Stack Operations

- Push Operation : $O(1)$
- Pop Operation : $O(1)$
- Top Operation : $O(1)$
- Search Operation : $O(n)$

Stack - Relavance

- Stacks appear in computer programs
 - Key to call / return in functions & procedures
 - Stack frame allows recursive calls
 - Call: push stack frame
 - Return: pop stack frame

Stack - Relavance

- Stacks appear in computer programs
 - Key to call / return in functions & procedures
 - Stack frame allows recursive calls
 - Call: push stack frame
 - Return: pop stack frame
- Stack frame
 - Function arguments
 - Return address
 - Local variables

Use of Stacks in Function call – System Stack

- Whenever a function is invoked program creates a structure called **activation record or a stack frame** and places it on top of system stack.
- Initially, the activation record for the invoked functions contains only a pointer to the previous frame and return address.
- The previous stack frame pointer points to the stack frame of invoking function.

Use of Stacks in Function call – System Stack

- While return address contains the location of the statement to be executed after the function terminates.
- Only one function executes at given time which is the top of stack.
- If this function invokes another, the non-static local variables parameters of invoking function are added to stack frame.

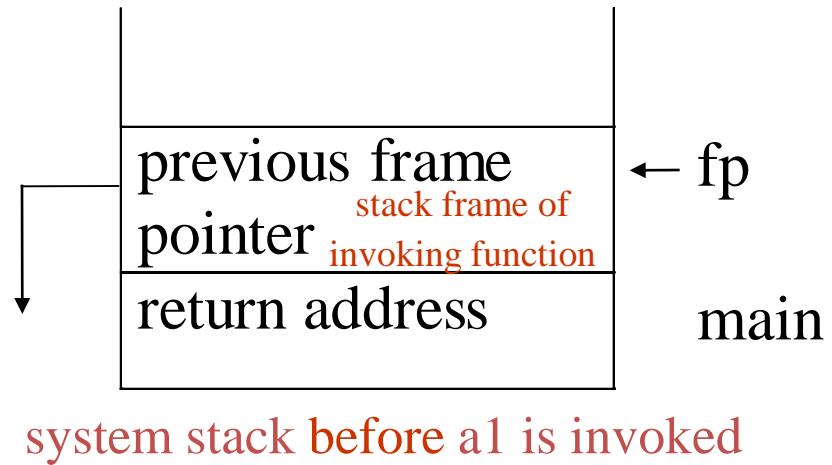
Stacks in Function call – System Stack

- Assume that main() invokes function a1.
- It creates stack frame for a1.
- Frame pointer is a pointer to the current stack frame
- Also system maintains separately a stack pointer
- When a function terminates its stack frame is removed
- Processing of invoking which is on top of stack continues

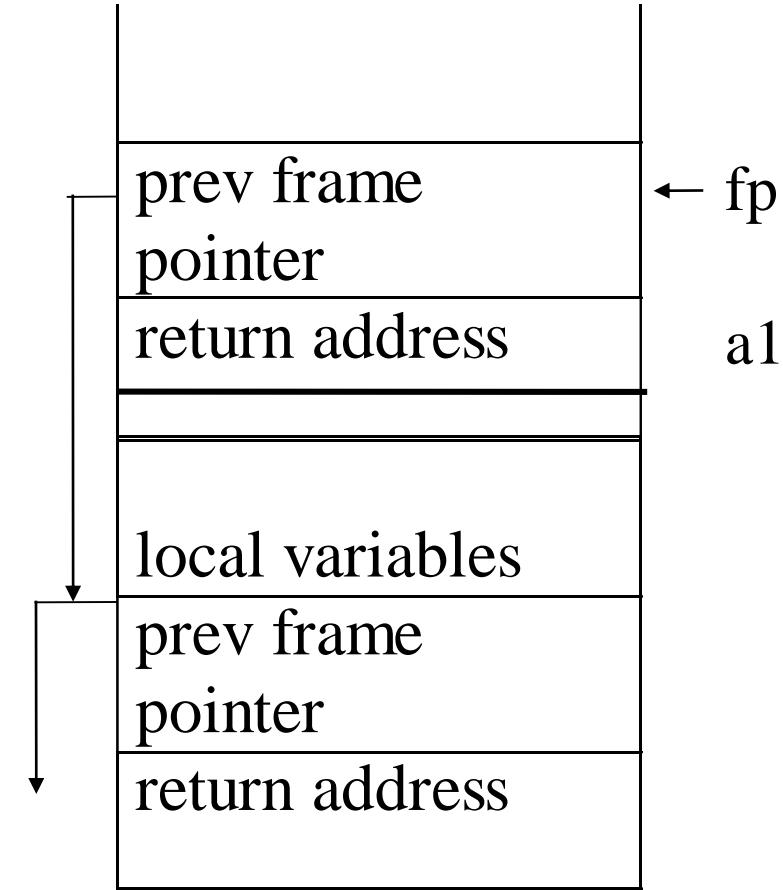
An application of stack: stack frame of function call

(activation record)

fp: a pointer to current stack frame



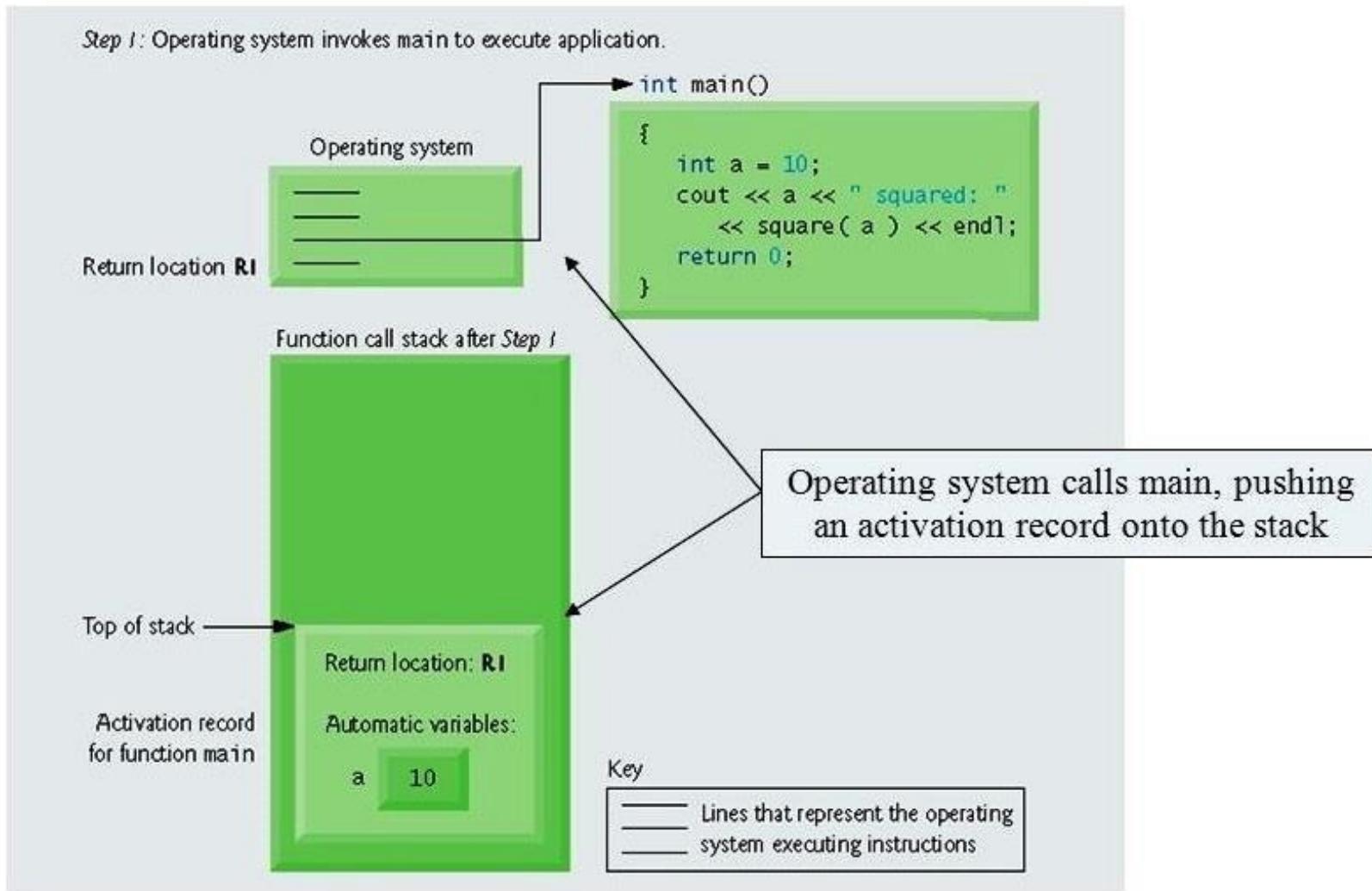
(a)



(b)
system stack after a1 is invoked

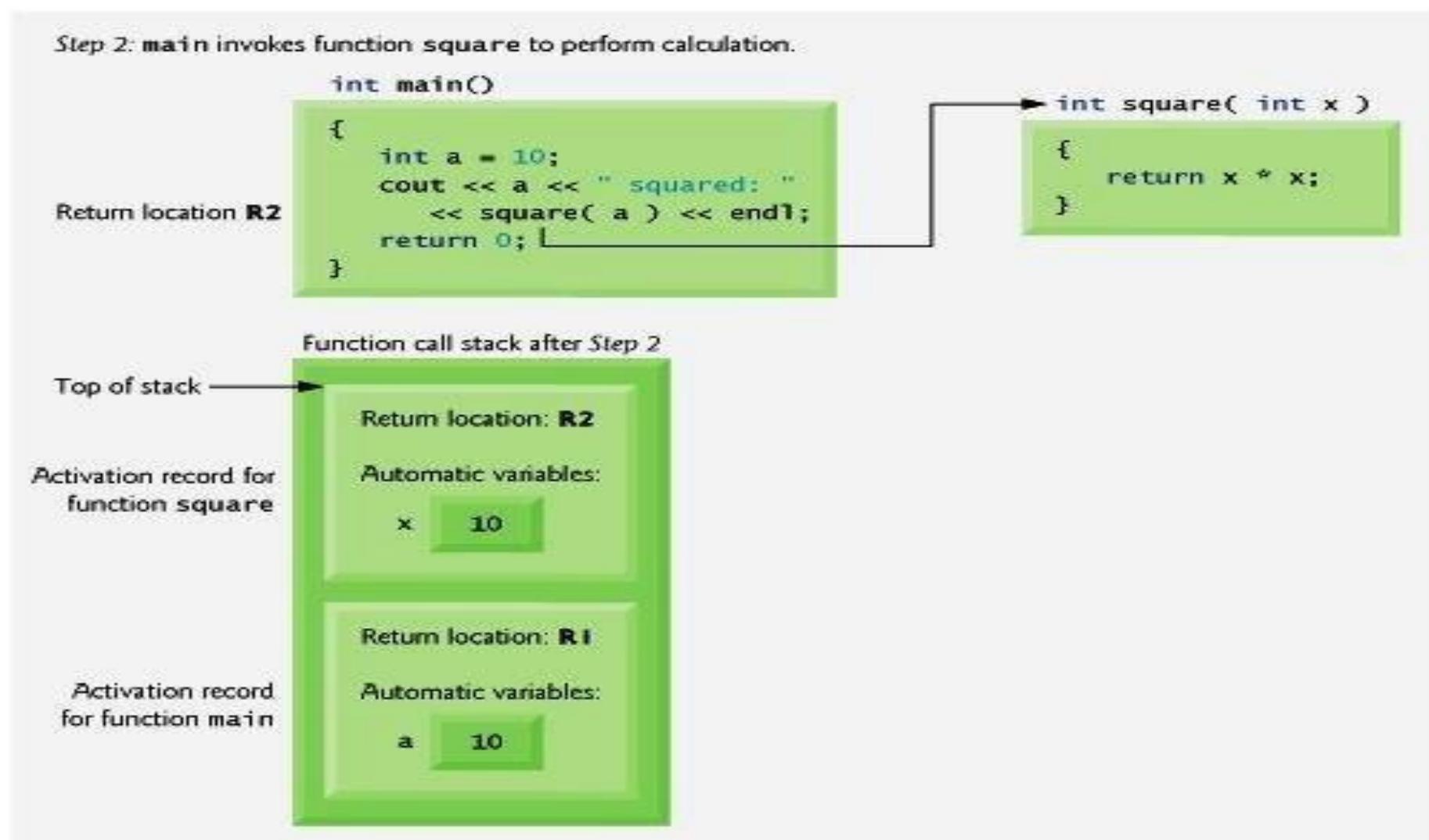
Applications of Stack

■ Function Call (Continued ...)



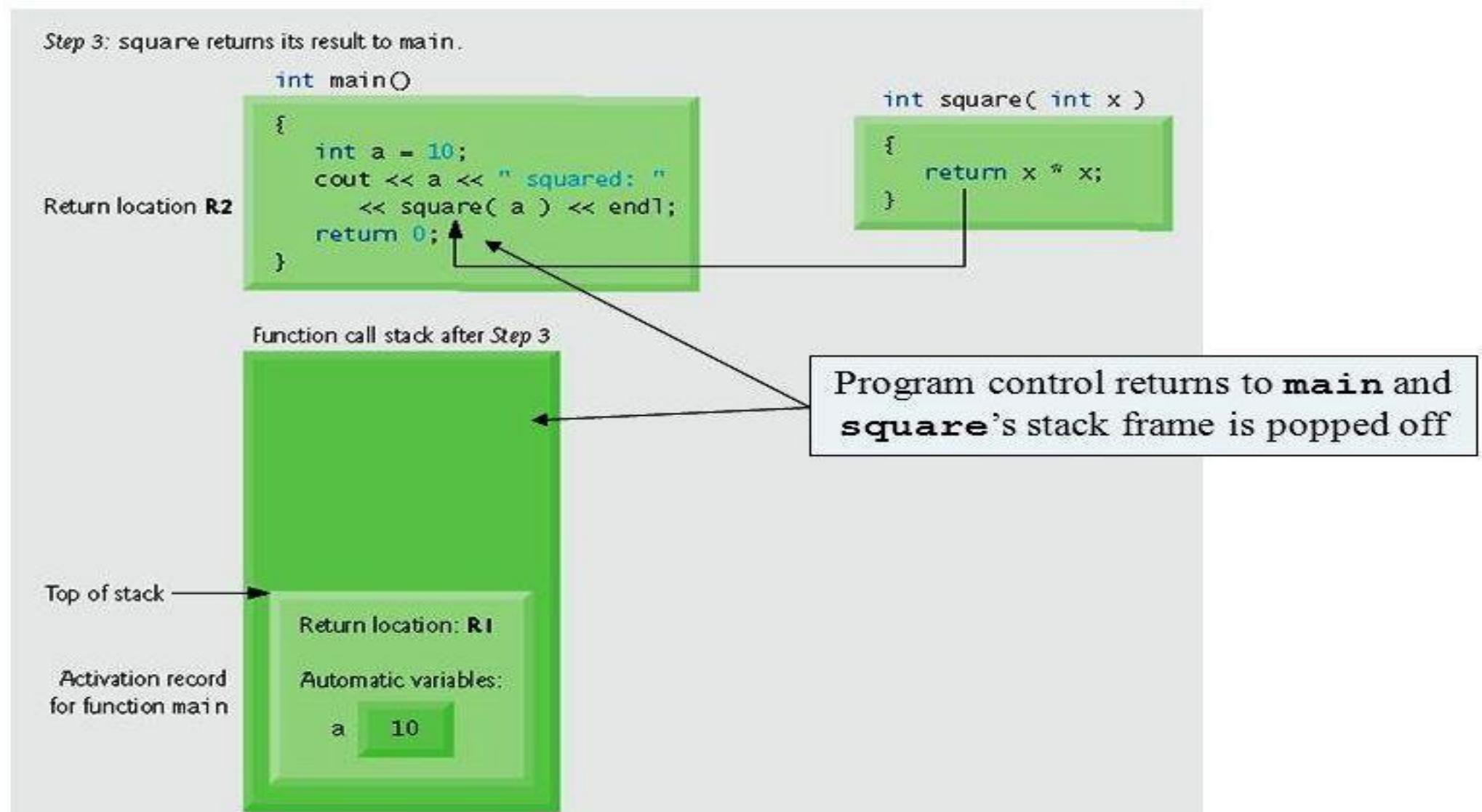
Applications of Stack

■ Function Call (Continued ...)



Applications of Stack

■ Function Call (Continued ...)



Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like:

- 1.Expression Conversion(Infix to Postfix, Postfix to Prefix etc)
- 2.Parsing
- 3.It can be used to process function calls.
- 4.Implementing recursive functions in high level languages

Application of Stack

Parentheses Matching

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

Example

- $((a+b)^*c+d-e)/(f+g)-(h+j)^*(k-l)) / (m-n)$

$\begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$

(2,6) (1,13) (15,19) (21,25) (27,31)

Applications of Stack: Symbols Matching

- Check for the following:
 - Equal number of left and right parentheses
 - Every right parenthesis is preceded by left parenthesis
- Valid expression: $7 - ((x * ((x + y) / (j - 3)) + y) / (4 - k))$
- Invalid expressions: $((a + b)$ $a + b($ $)a + b(-c$ $(a + b)) - (c + d$
- Nesting depth at a particular point:
 - Number of scopes opened and not closed
- Parenthesis count at a particular point:
 - Number of left parentheses – number of right parentheses

Applications of Stack: Symbols Matching

- Multiple types of delimiters
 - Parentheses
 - Brackets
 - Braces

Valid:

$[(a + b)]$

$\{a - (b + c)\}$

Invalid:

$(a + b]$

$[(a + b])$

$\{ a - (b] \}$

Applications of Stack

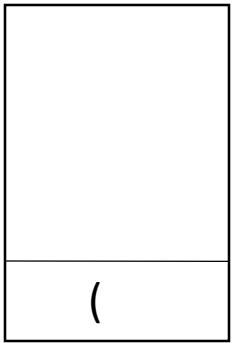
Function VALIDITY (P, VALID)

1. Set VALID := TRUE
2. Scan the expression P from **left to right** and repeat Steps 3 through 4 for each symbol read from the expression until the end of the string
3. If (**symbol = '('** or **symbol = '['** or **symbol = '{'**) then **push** the symbol onto the stack

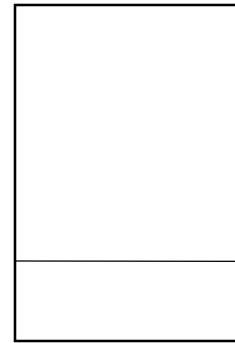
Applications of Stack

4. If (symbol = ')' or symbol = ']' or symbol = '}') then
 - If stack is empty then
 - Set VALID := FALSE;
 - else
 - {
 1. Pop an item op from the stack;
 2. If (op is not the matching opener of symbol) then Set VALID := FALSE;
 3. }
 5. If stack is not empty, then
 - Set VALID := FALSE;
 6. return

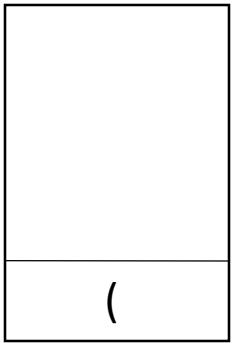
Ex1: $(a+b) * (c+d)$



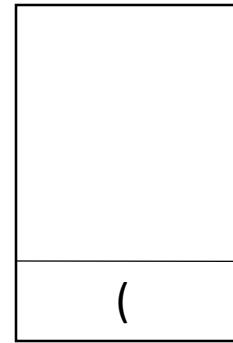
(
push '('



(a+b)
Pop '(' since
there is ')' &
continue

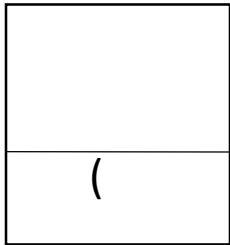


(a+b) *(
Push '('

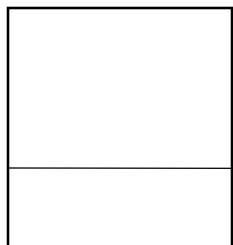


(a+b) *(c+d)
End of string reached
but stack not empty.
Hence invalid

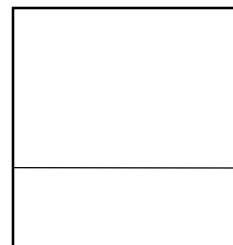
Ex2: $(a+b)*c+d)$



(
push '('

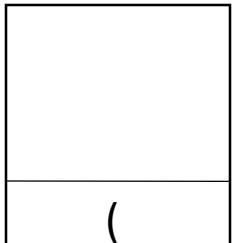


(a+b)
pop '(' and
continue

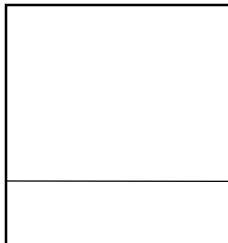


(a+b)*c+d)
Stack empty when ')' encountered. Hence invalid

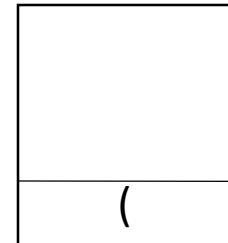
Ex3: $(a+b)^*(\{c^*d)$



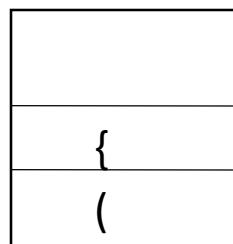
(
push '('



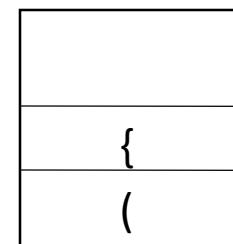
(a+b)
**pop'(' and
continue**



(a+b)*(
**push'(' and
continue**

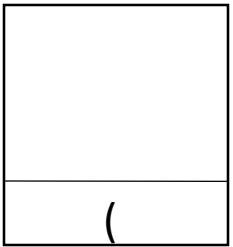


(a+b)*({
**push '{' and
continue'**

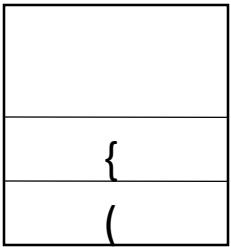


(a+b)*({c*d)
**No match between closing scope ')' and opening
scope '{'. Hence invalid.**

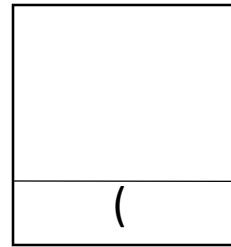
Ex4: $(a + \{b^*c\} + (c^*d))$



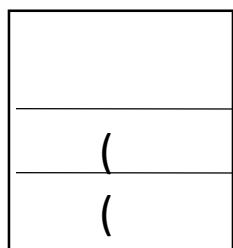
(
push '(' and
continue



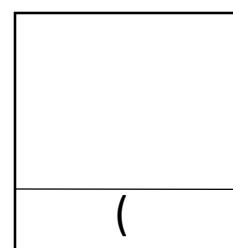
(a+{
push '{' and
continue



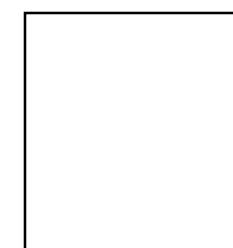
(a+{b*c}
pop '{' and continue



(a+{b*c}+
push '(' and
continue'



(a+{b*c}+(c*d)
Pop '('



(a+{b*c}+(c*d))
Pop '('.
End of string and stack empty. Hence valid

Evaluation of Expressions

$X = a / b - c + d * e - a * c$

$a = 4, b = c = 2, d = e = 3$

Interpretation 1:

$$((4/2)-2)+(3*3)-(4*2) = 0 + 9-8 = 1$$

Interpretation 2:

$$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666\cdots$$

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Token	Operator	Precedence ¹	Associativity
()	function call	17	left-to-right
[]	array element		
-> .	struct or union member		
- ++	increment, decrement ²	16	left-to-right
- - ++	decrement, increment ³	15	right-to-left
!	logical not		
-	one's complement		
- +	unary minus or plus		
& *	address or indirection		
sizeof	size (in bytes)		
(type)	type cast	14	right-to-left
* / %	multiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >=	relational	10	left-to-right
< <=			
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= =	assignment	2	right-to-left
,	comma	1	left-to-right

user

compiler

Infix	Postfix
$2+3*4$	$234*+$
$a*b+5$	$ab*5+$
$(1+2)*7$	$12+7*$
$a*b/c$	$ab*c/$
$(a/(b-c+d))*((e-a)^*c)$	$abc-d+/ea-*c^*$
$a/b-c+d^*e-a^*c$	$ab/c-de^*ac^-*$

Note--→ **Postfix: no parentheses, no precedence**

Infix to Postfix Conversion

(Intuitive Algorithm/ Manual method)

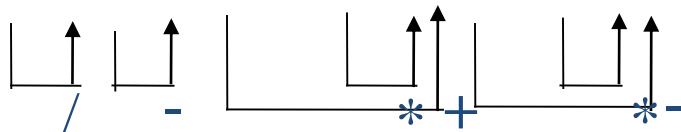
(1) **Fully parenthesize expression**

$a / b - c + d * e - a * c \rightarrow$

$((((a / b) - c) + (d * e)) - a * c))$

(2) **All operators replace their corresponding right parentheses.**

$((((a / b) - c) + (d * e)) - (a * c))$



(3) **Delete all parentheses.**

$ab/c-de^*+ac^*-$

two passes

Infix expression

- In an **expression if the binary operator, which performs an operation , is written in between the operands it is called an infix expression.** Ex: **a+b*c**

Infix, Prefix and Postfix expression

- In an expression if the binary operator, which performs an operation , is written in between the operands it is called an **infix expression**.

Ex: $a+b*c$

- If the operator is written before the operands , it is called **prefix expression**

Ex: $+a*bc$

- If the operator is written after the operands, it is called **postfix expression**

Ex: $abc^* +$

Infix, Prefix, and Postfix expression

- An expression in infix form is **dependent of precedence** during evaluation
- Ex: to evaluate $a+b*c$, sub expression $a+b$ can be evaluated only after evaluating $b*c$.
- **As soon as we get an operator we cannot perform the operation specified on the operands.**
- **So it takes more time for compilers to check precedence to evaluate sub expression.**

Infix, Prefix, and Postfix expression

- Both prefix and postfix representations are independent of precedence of operators.
- In a single scan an entire expression can be evaluated
- Takes less time to evaluate.
 - However infix expressions have to be converted to postfix or prefix.

Stack: Applications

Conversion and evaluation of expressions

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Infix Notation: operators are used **in**-between operands

e.g. $a - b + c$

- **Advantage: easy to read, write, and speak for humans**
- **Difficult and costly in terms of time and space consumption**

Postfix Notation:

The operator is written after the operands.

For example, **ab+**

Also, known as **Reversed Polish Notation**

Prefix Notation : operators are followed by operands i.e the operators are fixed before the operands.

For example, **+ab**

Also, known as **Polish Notation**

All the infix expression will be converted into post fix notation with the help of stack in any program

Parsing Expressions

To parse any arithmetic expression, we need to take care of **operator precedence** and **associativity** also.

Precedence

$$a + b * c \rightarrow a + (b * c)$$

Associativity

Rule where operators with the same precedence appear in an expression

$$a + b - c$$

$$(a + b) - c$$

Table shows the default behavior of operators

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

Altered by using parenthesis

$$a + b*c$$

$$(a + b)*c$$

Applications of Stack

■ Conversion of infix form to postfix

INFIX_POSTFIX (Q, P)

1. Push '(' onto stack, and add ')' to the end of Q.
2. Scan Q **from left to right** and repeat Steps 3 through 6 for each element of Q until the stack is empty.
3. If an **operand** is encountered, add it to the right of P.
4. If a **left parenthesis** is encountered, push it on to the stack.
5. If an **operator** *op* is encountered, then
 - a) Repeatedly pop from the stack and add to the right of P each operator (on the top of the stack) which has the same precedence as or higher precedence than *op*.
 - b) Add *op* to the stack.

Applications of Stack

■ Conversion of infix form to postfix (Continued...)

6. If a **right parenthesis** is encountered, then
 - a) Repeatedly pop from the stack and add to the right of P each operator (on the top of the stack) until a left parenthesis is encountered.
 - b) Remove the left parenthesis. [Do not add the left parenthesis to P]
7. Return

Convert infix to post fix expression

Example 1:-

$a - (b + c * d)/e$

Ch	Stack(bottom to top)	PostfixExp
a		a
-	-	a
(-()	a
b	-()	ab
+	-(+	ab
c	-(+	abc
*	-(+*	abc
d	-(+*	abcd
)	-(+	abcd*
/	-()	abcd*+
e	-	abcd*+
	-/	abcd*+
	-/	abcd*+e
		abcd*+e/-

Infix to postfix conversion: Sample Exercises

Convert the following infix expression to postfix expression

- $a+b*c+d*e$
- $a*b+5$
- $(a/(b-c+d))*((e-a)*c)$
- $a/b-c+d*e-a*c$

Applications of Stack

■ Conversion of infix form to prefix

INFIX_PREFIX (Q, P)

- 1. Push ')' onto stack, and add '(' at the beginning of Q.**
- 2. Scan Q **from right to left** and repeat Steps 3 through 6 for each element of Q until the stack is empty.**
- 3. If an **operand** is encountered, add it to the left of P.**
- 4. If a **right parenthesis** is encountered, push it onto the stack.**
- 5. If an **operator op** is encountered, then**
 - a) Repeatedly pop from the stack and add to the left of P each operator (on the top of the stack) which has higher precedence than *op*.**
 - b) Add *op* to the stack.**

Applications of Stack

■ Conversion of infix form to prefix (Continued...)

- 6. If a **left parenthesis** is encountered, then**
 - a) Repeatedly pop from the stack and add to the left of P each operator (on the top of the stack) until a **right parenthesis** is encountered.**
 - b) Remove the right parenthesis. [Do not add the right parenthesis to P]**
- 7. Return**

Convert Infix To Prefix Notation (Alternative method)

- Step 1: Reverse the infix expression

Ex. **A+B*C** will become **C*B+A**.

Note: While reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.

- Step 2: Obtain the postfix expression of the modified expression

i.e.: **CB*A+**.

- Step 3: Reverse the postfix expression.

- Hence in our example prefix is **+A*BC**.

Applications of Stack

■ Evaluation of a postfix expression

POST_EVALUATE (P, VALUE)

1. Scan P **from left to right** and repeat Steps 2 through 3 for each element of P until the **end** of the expression is encountered.
2. If an **operand** is encountered, push it to stack.
3. If an **operator *op*** is encountered, then
 - a) Pop the two top elements from the stack and assign them to ***opnd2*** and ***opnd1*** respectively.
 - b) Evaluate ***opnd1 op opnd2*** and push the result onto the stack.
1. Set **VALUE** equal to the **top element** on the stack.
2. Return

Example for evaluation of Postfix expression

2 10 + 9 6 - /

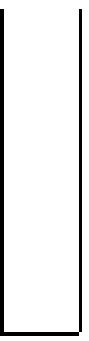
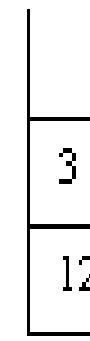
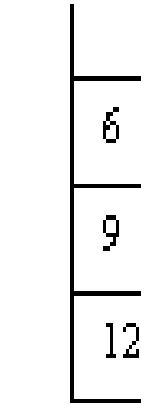
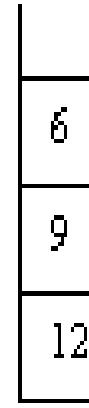
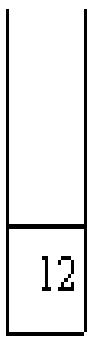
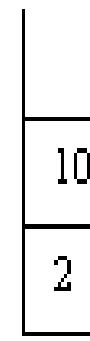
push 2
push 10

pop 10
pop 2
push 2 + 10 = 12

push 9
push 6
push 9 - 6 = 3

pop 6
pop 9
push 12 / 3 = 4

pop answer: 4



Applications of Stack

■ Evaluation of a prefix expression

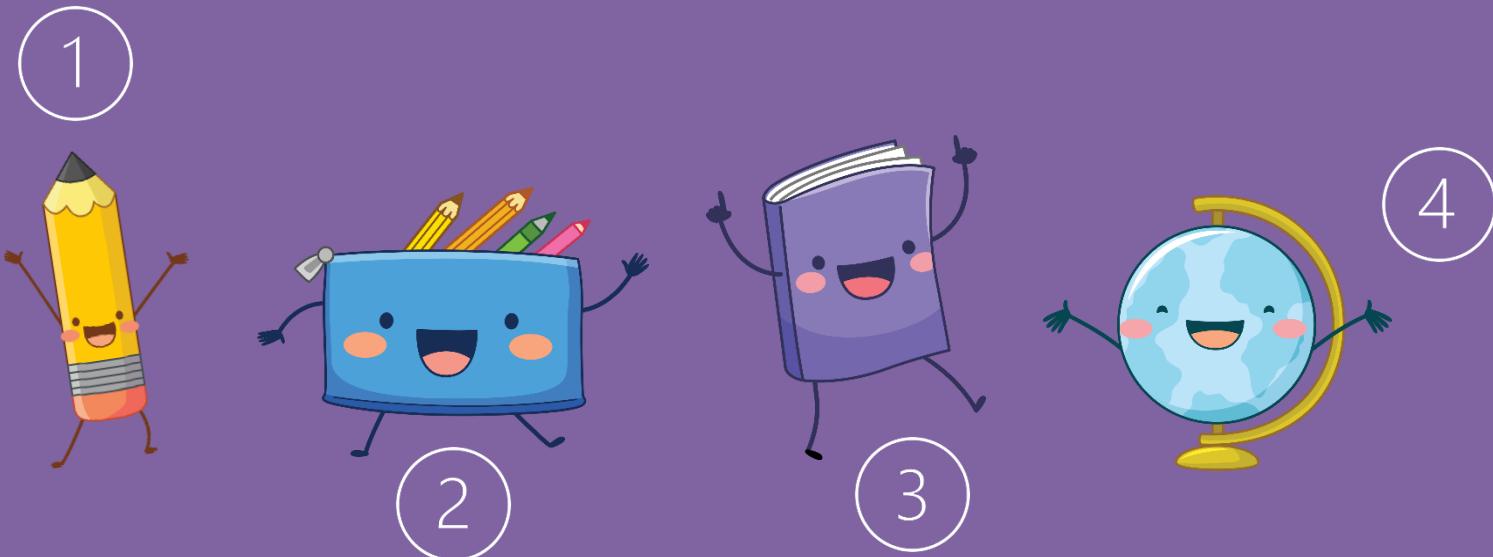
PRE_EVALUATE (P, VALUE)

1. Scan P **from right to left** and repeat Steps 2 through 3 for each element of P until the **beginning** of the expression is encountered.
2. If an **operand** is encountered, push it to stack.
3. If an **operator *op*** is encountered, then
 - a) Pop the two top elements from the stack and assign them to ***opnd1* and *opnd2*** respectively.
 - b) Evaluate ***opnd1 op opnd2*** and push the result onto the stack.
1. Set **VALUE** equal to the **top element** on the stack.
2. Return

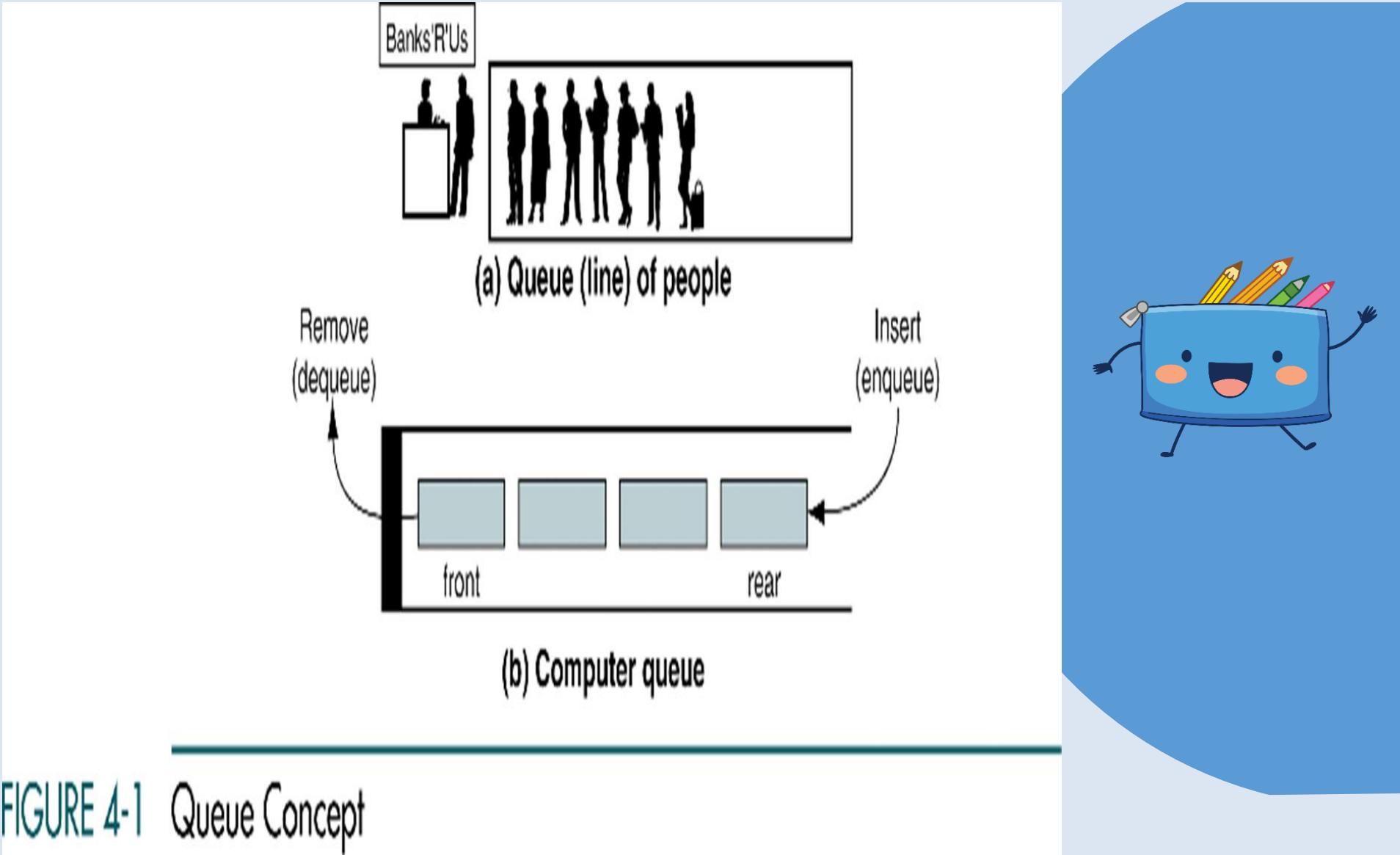
END OF STACK

LEARNING ABOUT A NEW DATA STRUCTURE

- QUEUES



What is a queue?



Bus Stop Queue



Bus
Stop

front

rear



Bus Stop Queue



front

rear



Bus Stop Queue



front

rear



Bus Stop Queue



front

rear



Basic features of Queue

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle

Basic features of Queue

- Queue is an abstract data structure
- A queue is open at both its ends
- One end (**rear**) is always used to insert data (**enqueue**) and
- The **other end (front)** is used to remove data (**dequeue**).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.
- Real-world examples can be seen as queues at the ticket windows and bus-stops.

Queues

Basic features of Queue

- Linear list.
- One end is called **front**.
- Other end is called **rear**.
- Additions are done at the **rear** only.
- Removals are made from the **front** only.

Queue

Insertion and Deletion
happen on different ends

Enqueue

Rear

Front

Dequeue

First in, first out



front

rear

0

1

2

3

4

10

20

30

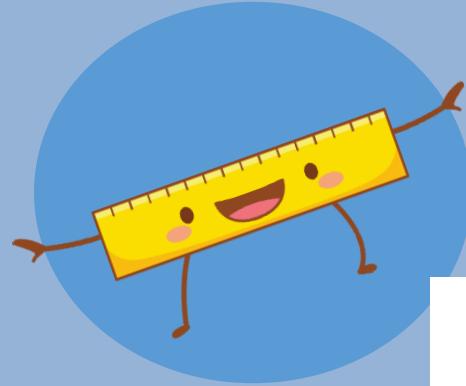
Front=-1
points to location
one before the 1st element

Rear
always points to last data

Queue Operations



- ➊ Enqueue
- ➋ Dequeue
- ➌ Queue Front
- ➍ Queue Rear
- ➎ Queue Example



ENQUEUE

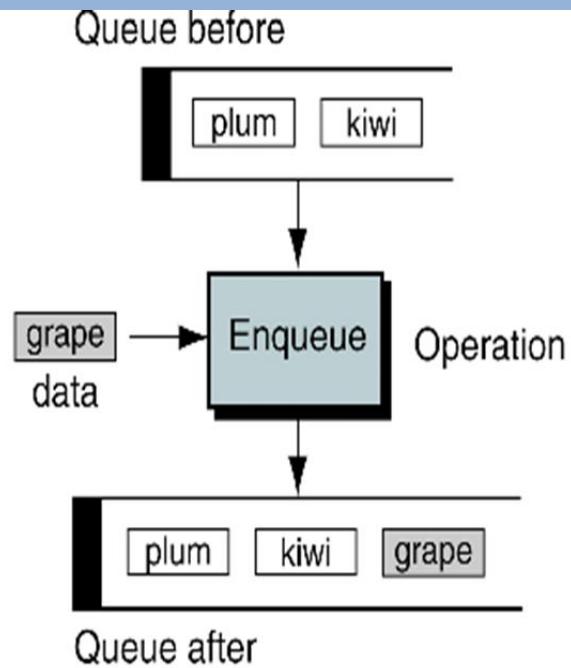


FIGURE 4-2 Enqueue

DEQUEUE

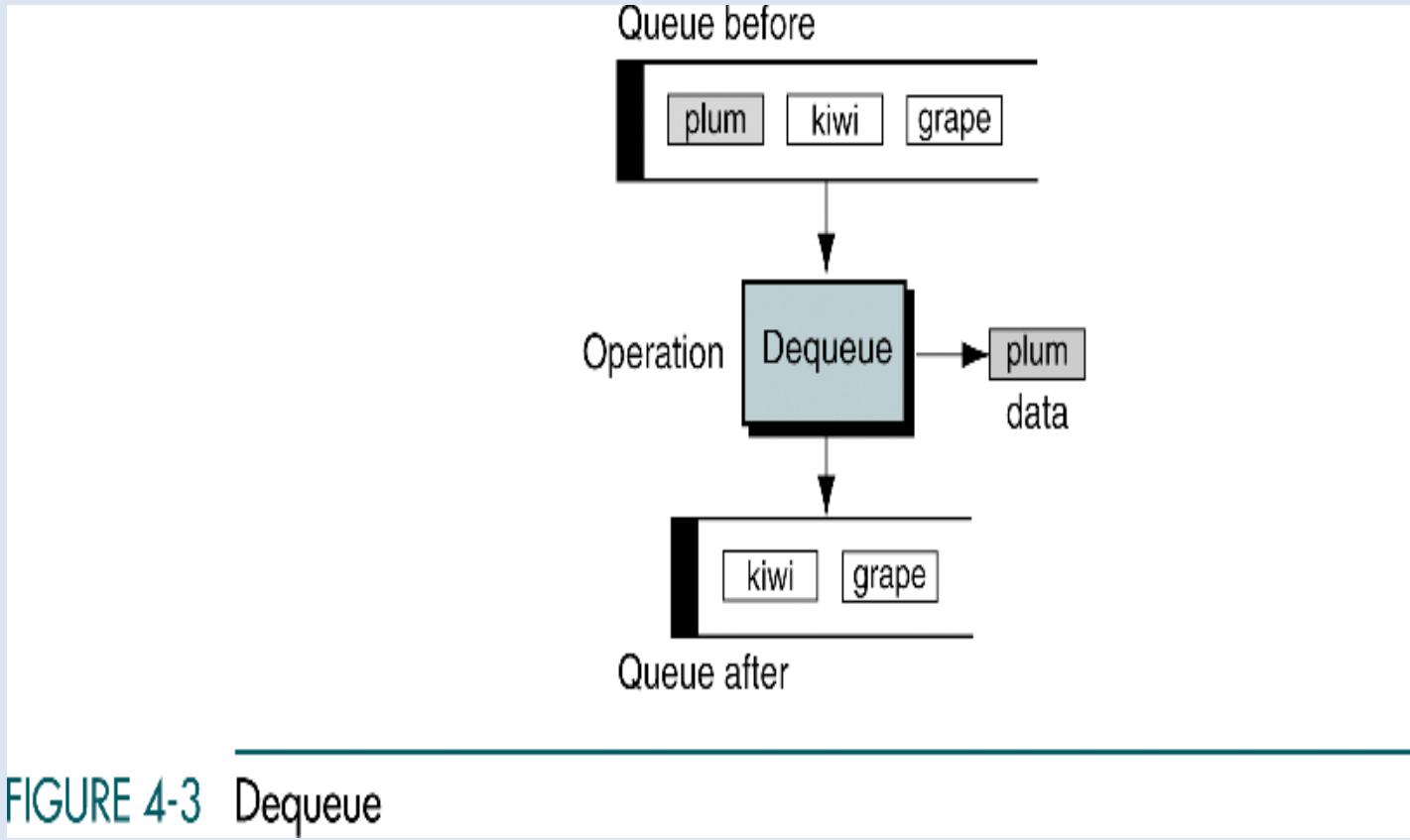
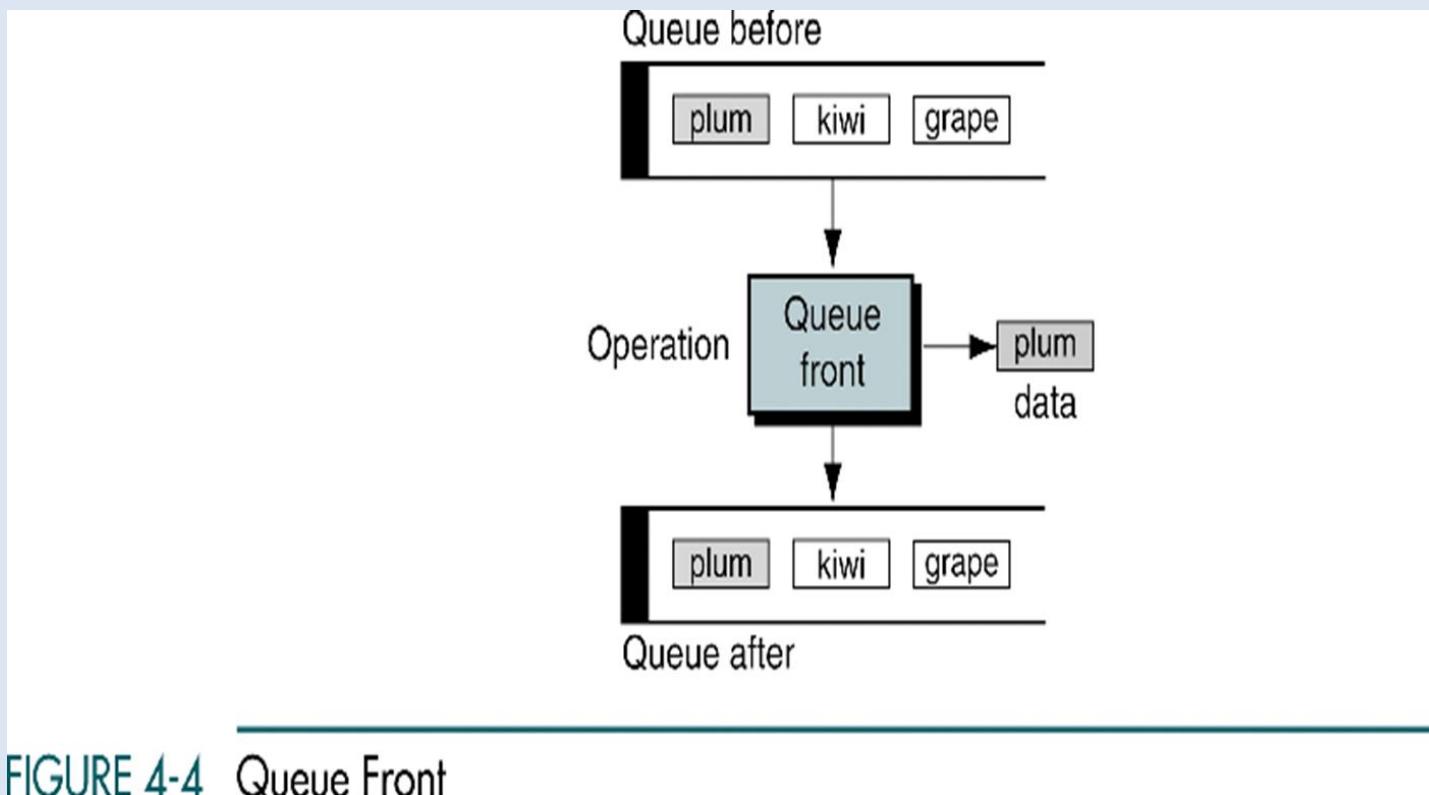


FIGURE 4-3 Dequeue

FRONT OF QUEUE



REAR OF QUEUE

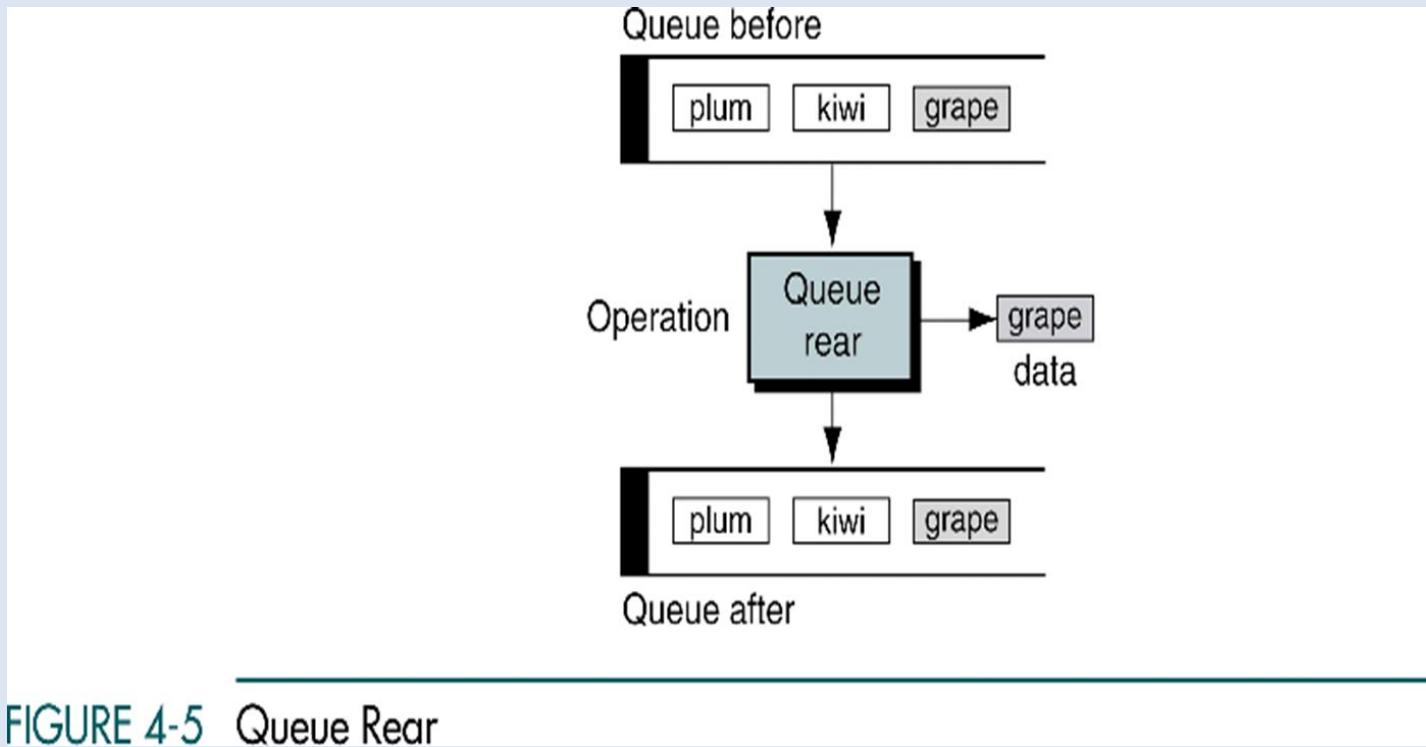
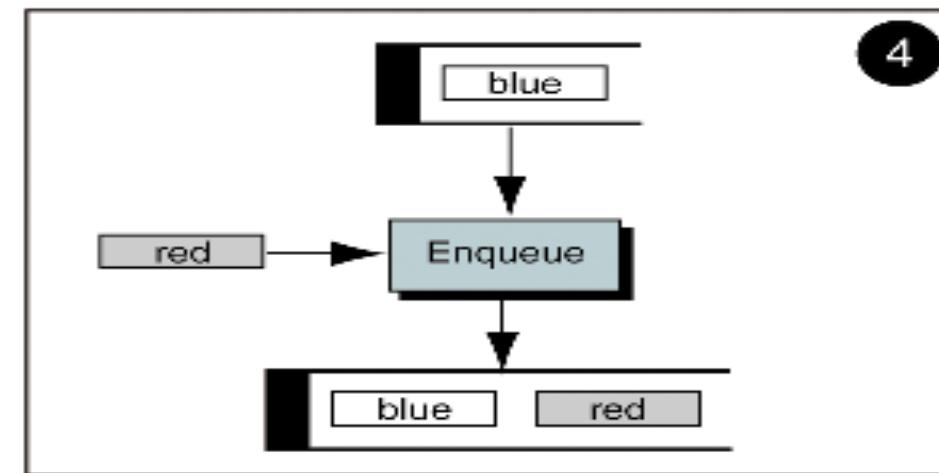
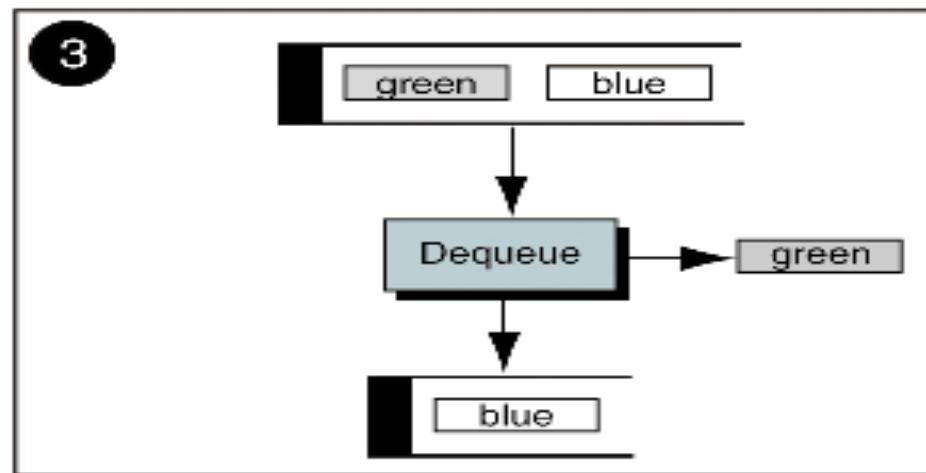
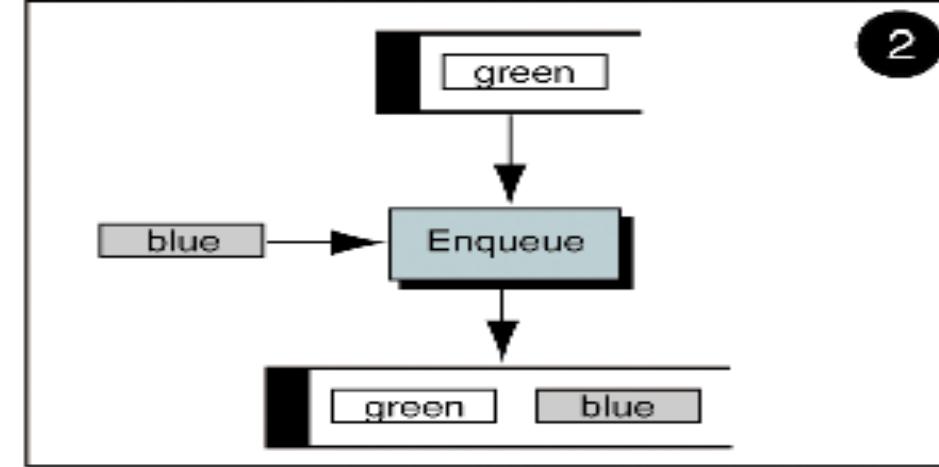
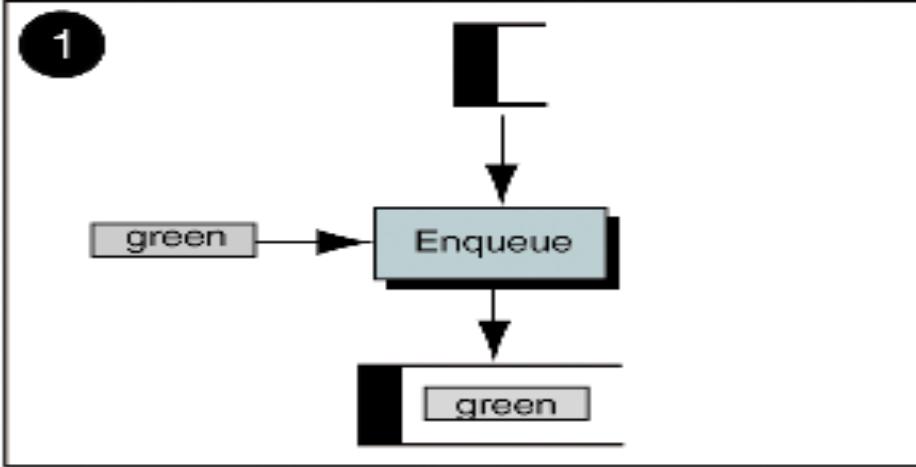


FIGURE 4-5 Queue Rear

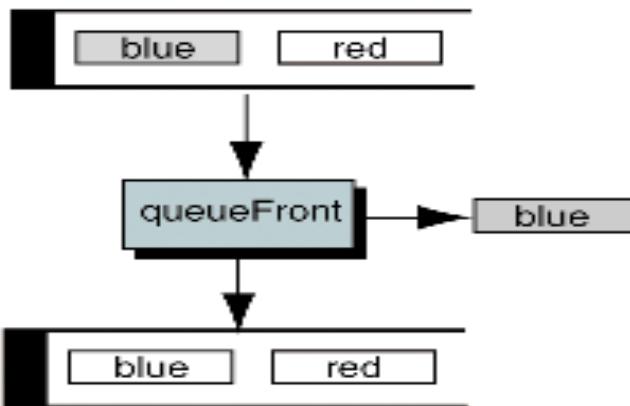


QUEUE EXAMPLE.

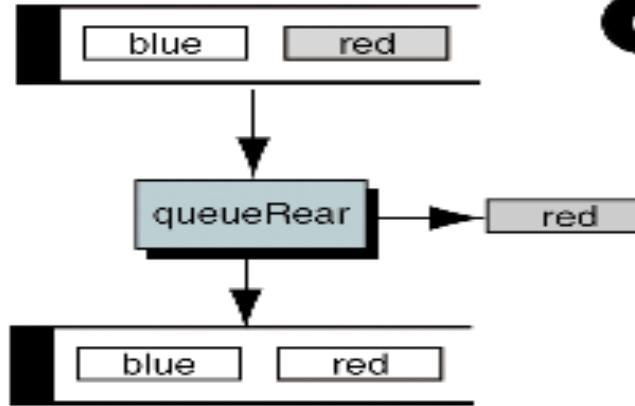


QUEUE EXAMPLE.

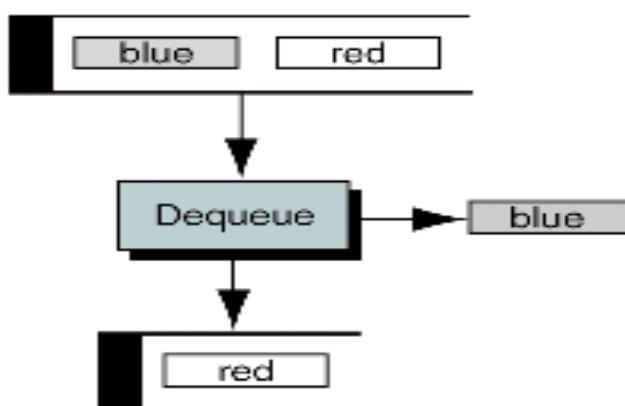
5



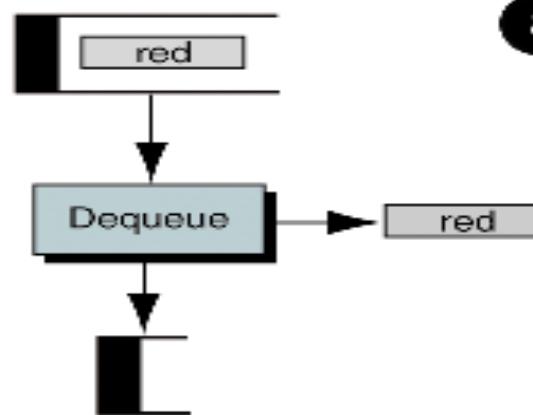
6



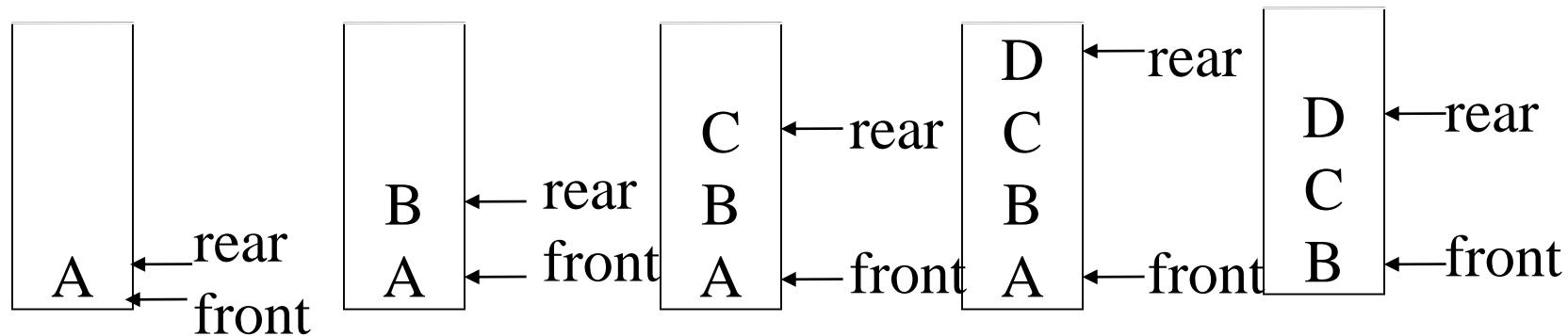
7



8



Queue: a First-In-First-Out (FIFO) list



*Figure 3.4: Inserting and deleting elements in a queue (p.108)

Queues - Application

- Used by operating system (OS) to create job queues.
- If OS does not use priorities then the jobs are processed in the order they enter the system

Application: Job scheduling

front	rear	Q1	Q2	Q3	Comments
-1	-1				queue is empty
-1	0	J1			Job1 is added
-1	1	J1	J2		Job2 is added
-1	2	J1	J2	J3	Job3 is added
0	2		J2	J3	Job1 is deleted
1	2			J3	Job2 is deleted

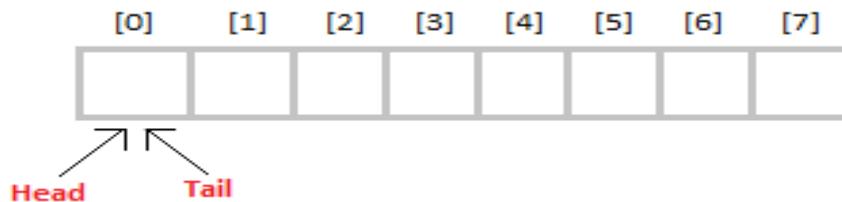
*Figure 3.5: Insertion and deletion from a sequential queue (p.117)

Applications of Queue

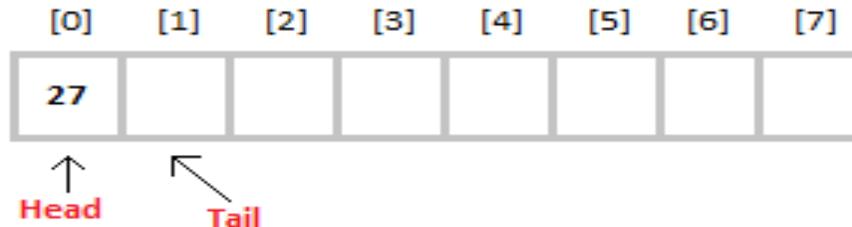
- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e. First come first served.

Implementation of Queue Data Structure

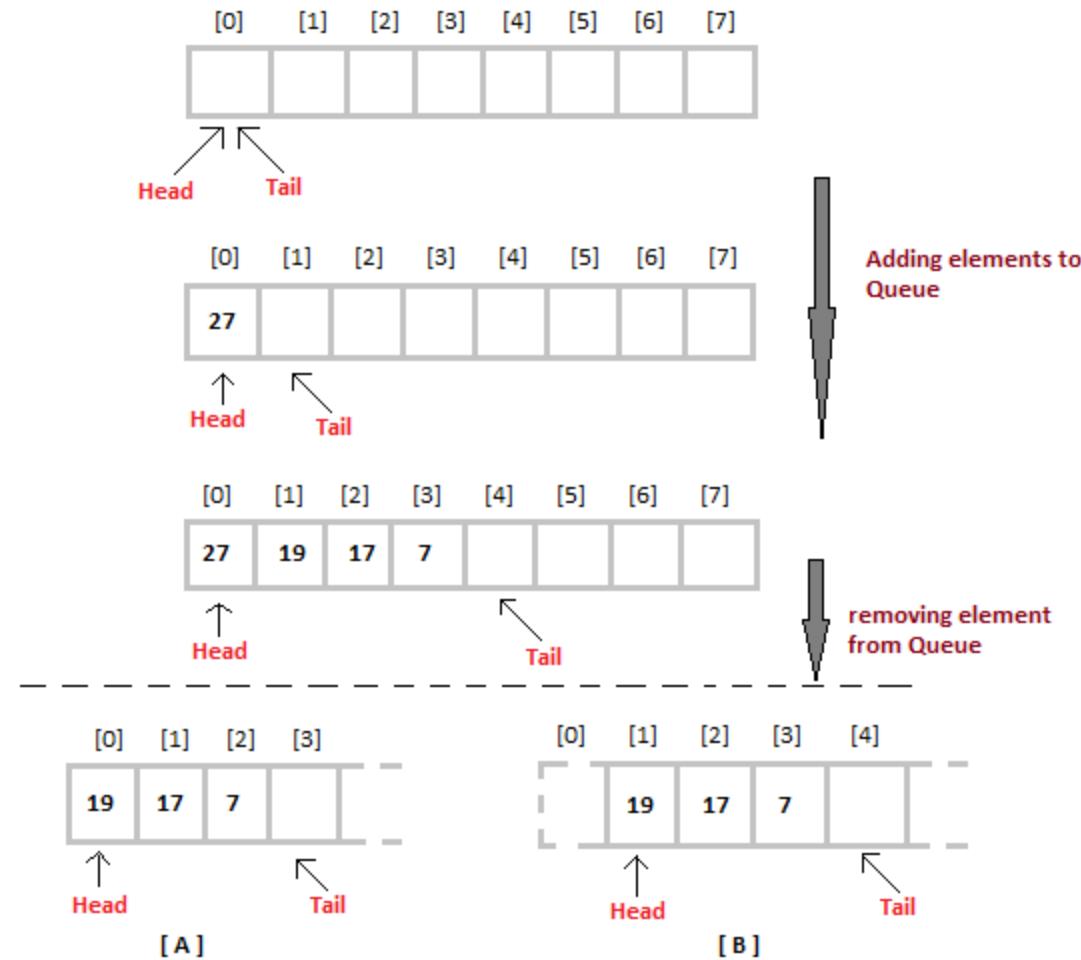
- Initially the **head(FRONT)** and the **tail(REAIR)** of the queue points at the first index of the array



- As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



Implementation of Queue Data Structure- Insertion and deletion



In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in **forward** position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Operations on a Queue

- 1.enQueue(value) - (To insert an element into the queue)
- 2.deQueue() - (To delete an element from the queue)
- 3.display() - (To display the elements of the queue)

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

Enqueue

Enqueue: If the queue is not full, this function adds an element to the back of the queue, else it prints “OverFlow”.

```
void enqueue(int queue[], int element, int& rear, int arraySize)
{
    if(rear == arraySize) // Queue is full
        printf("OverFlow\n");
    else
    {
        queue[rear] = element; // Add the element to the back
        rear++;
    }
}
```

Dequeue

Dequeue: If the queue is not empty, this function removes the element from the front of the queue, else it prints “UnderFlow”.

```
void dequeue(int queue[], int& front, int rear)
{
    if(front == rear) // Queue is empty
        printf("UnderFlow\n");
    else
    {
        queue[front] = 0; // Delete the front element
        front++;
    }
}
```

IsEmpty: If a queue is empty, this function returns 'true', else it returns 'false'.

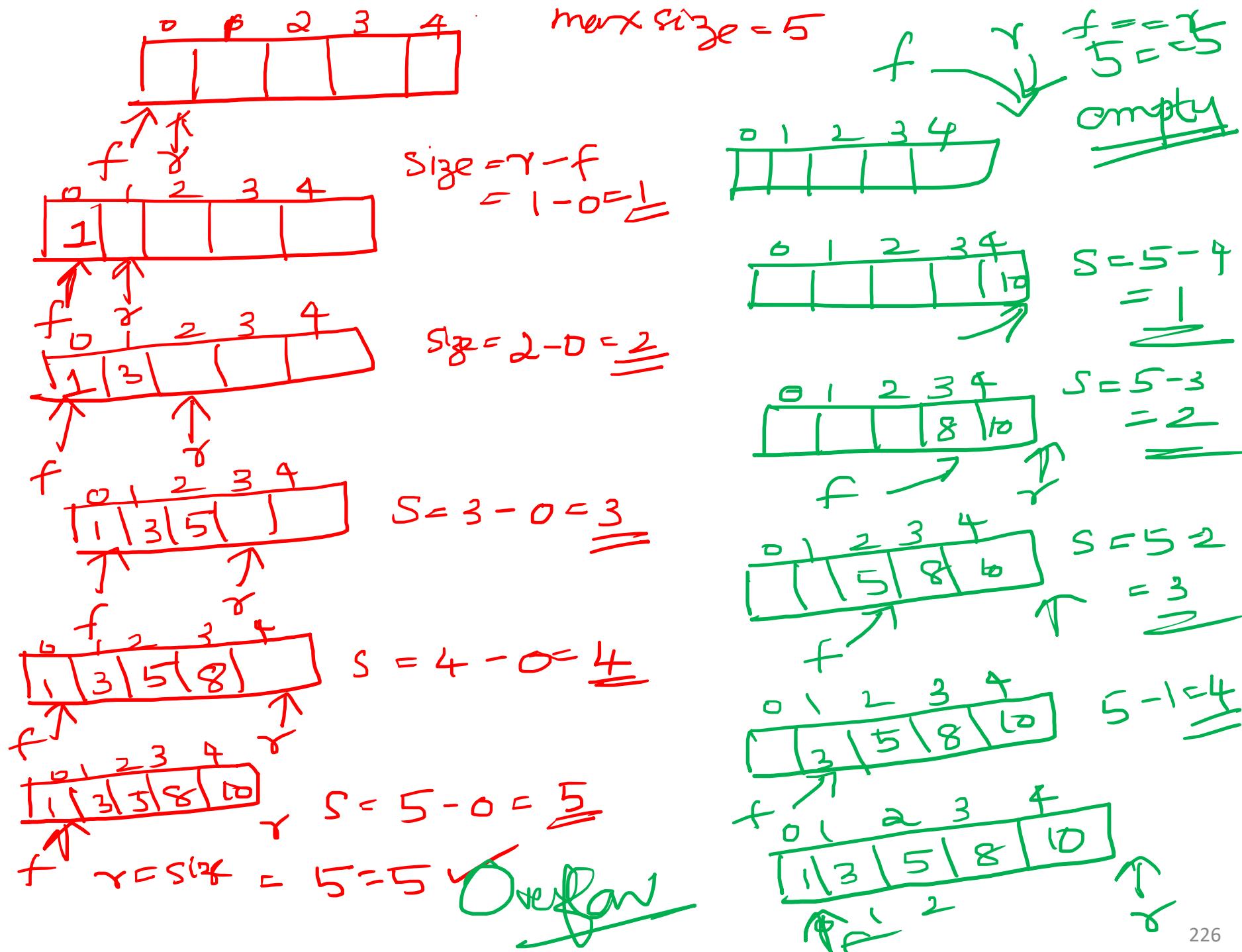
```
bool isEmpty(int front, int rear)  
{  
    return (front == rear);  
}
```

Front: This function returns the front element of the queue.

```
int Front(int queue[], int front)  
{  
    return queue[front];  
}
```

Size: This function returns the size of a queue or the number of elements in a queue.

```
int size(int front, int rear)  
{  
    return (rear - front);  
}
```



Complexity Analysis of Queue Operations

- Enqueue: **O(1)**
- Dequeue: **O(1)**
- Size: **O(1)**

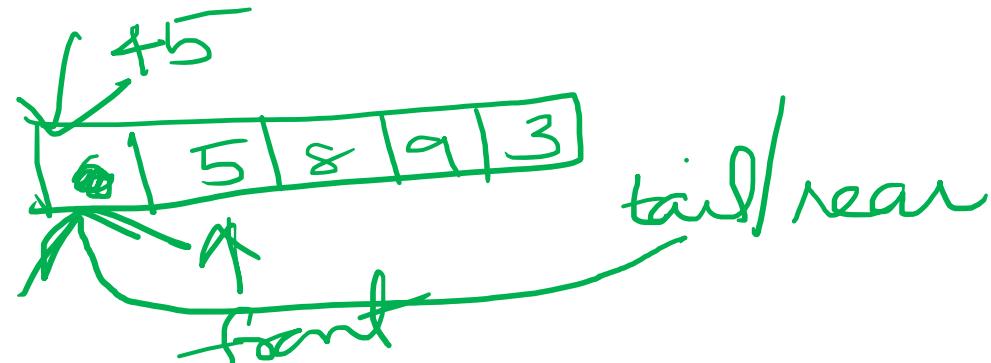
Queue variations

The standard queue data structure has the following variations:

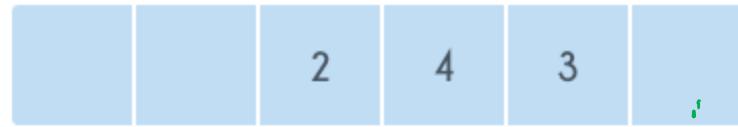
- 1.Double-ended queue
- 2.Circular queue
3. Priority queue

Circular queues

- A circular queue is an improvement over the standard queue structure.
- In a standard queue, when an element is deleted, the **vacant space is not reutilized**. However, in a circular queue, vacant spaces are **reutilized**.
- While inserting elements, when you reach the end of an array and you need to insert another element, you must insert that element at the beginning (given that the first element has been deleted and the space is vacant).

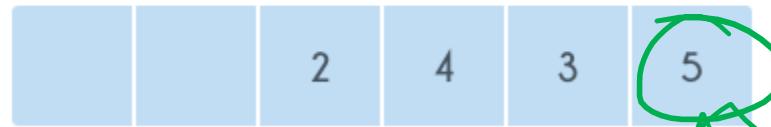


queue



↑
front rear

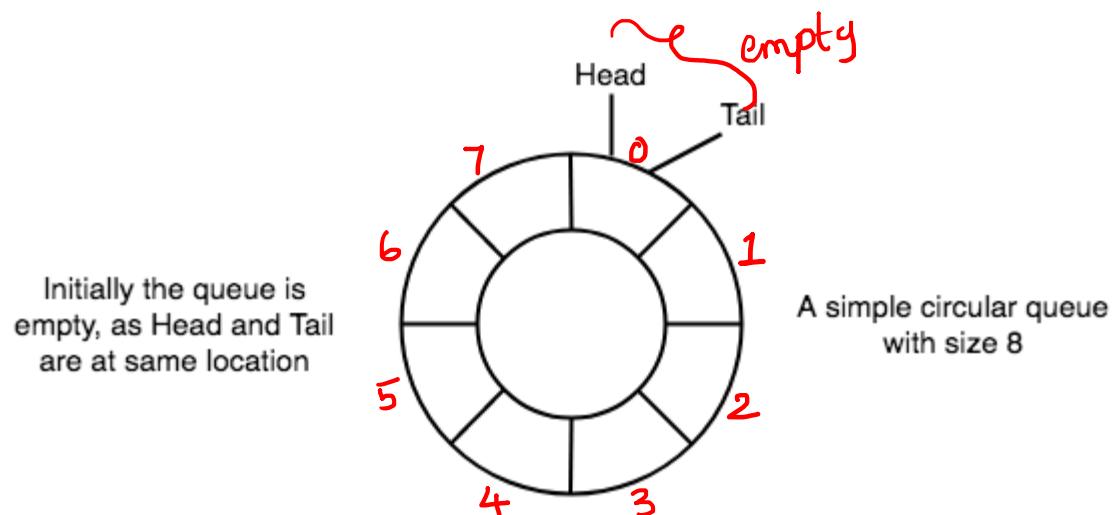
↓
insert 5



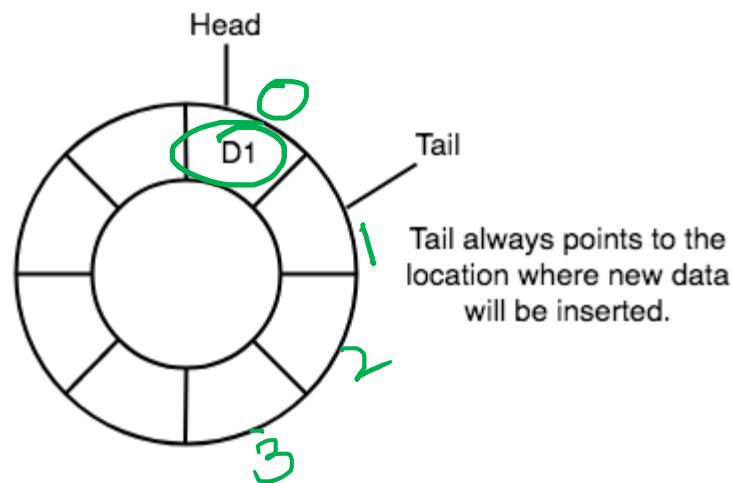
↑
rear front

Basic features of Circular Queue

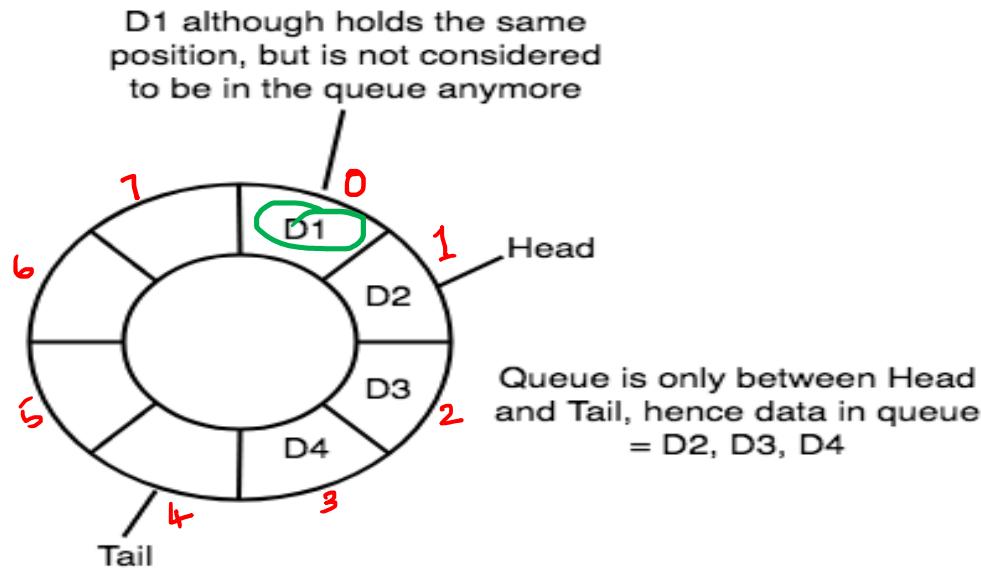
1. Head pointer will always point to the front of the queue, and
2. Tail pointer will always point to the end of the queue.
3. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



New data is always added to the location pointed by the **tail** pointer, and once the data is added, **tail** pointer is incremented to point to the next available location.

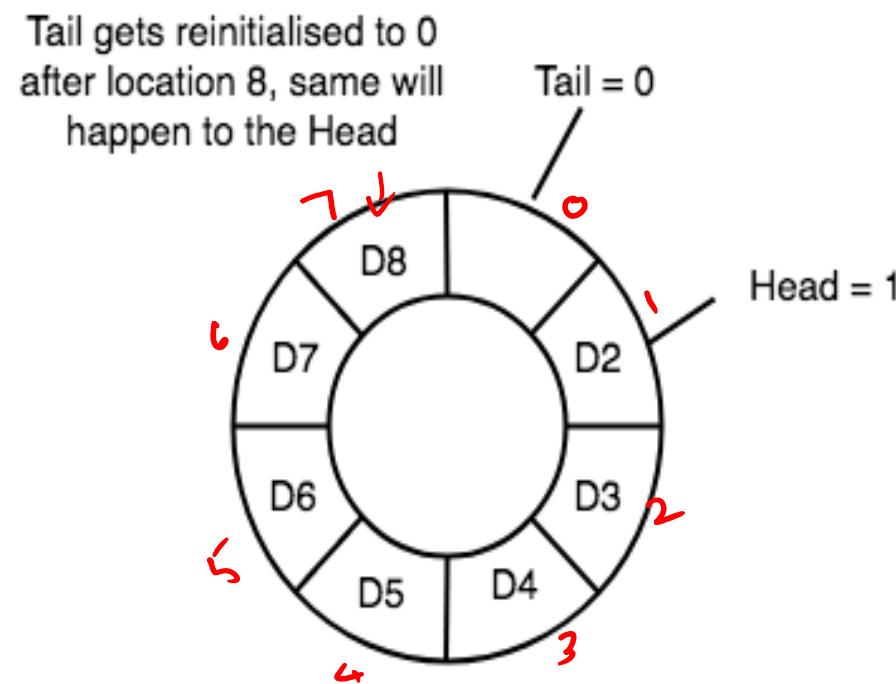


- In a circular queue, data is not actually removed from the queue.
- Only the **head** pointer is incremented by one position when **dequeue** is executed.
- As the **queue data is only the data between head and tail**, hence the data left outside is not a part of the queue anymore, hence removed.

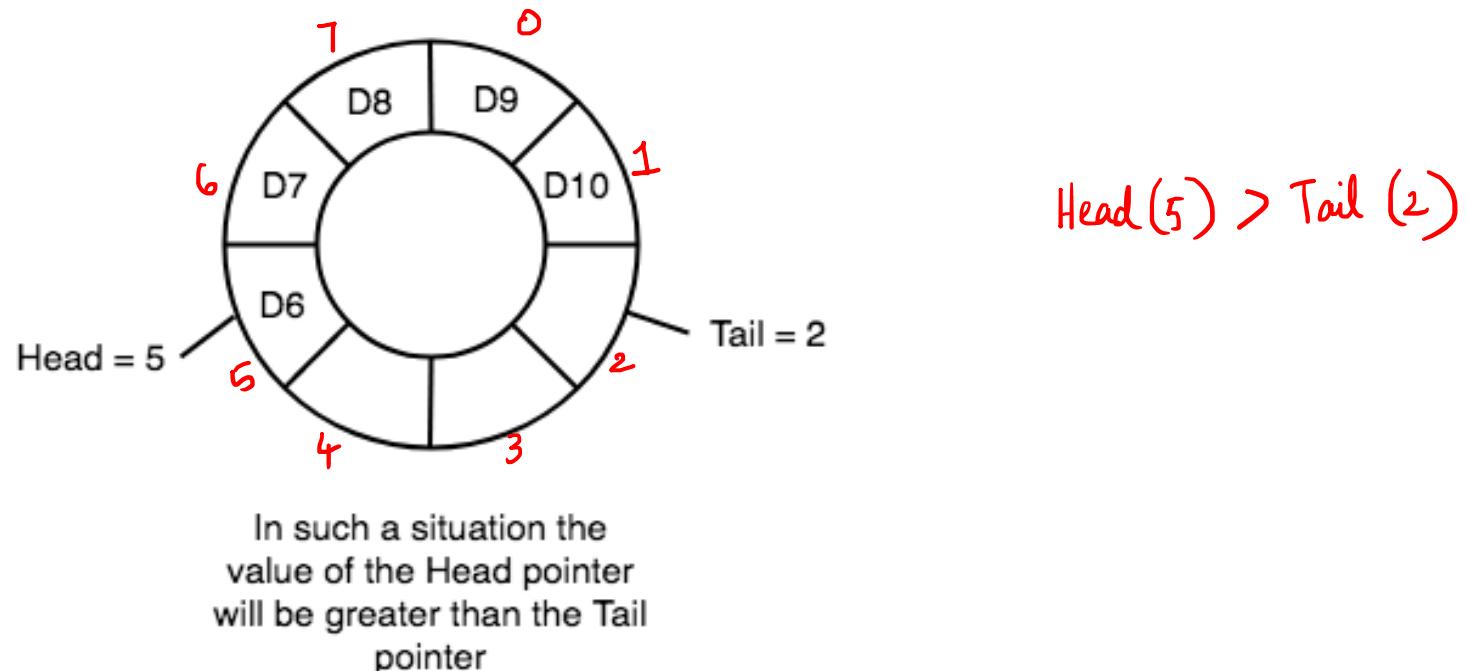


$$\begin{aligned}
 & \text{tail - head} \\
 &= 4 - 1 = 3 \\
 &\text{i.e.; 3 elements available in circular queue.} \\
 &\text{i.e., D2, D3, D4}
 \end{aligned}$$

The **head** and the **tail** pointer will get reinitialised to **0** every time they reach the end of the queue.



- Also, the **head** and the **tail** pointers can cross each other.
- In other words, **head** pointer can be greater than the **tail**. Sounds odd?
- This will happen when we dequeue the queue a couple of times and the **tail** pointer gets reinitialised upon reaching the end of the queue.



Going Round and Round

- Another very important point is keeping the value of the **tail** and the **head** pointer **within the maximum queue size**.
- In the diagrams above the queue has a size of **8**, hence, the value of **tail** and **head** pointers will always be between **0** and **7**.
- This can be controlled either by checking everytime whether **tail** or **head** have reached the **maxSize** and then setting the value **0**
- or, we have a better way, which is, for a value **x** if we divide it by **8**, the remainder will never be greater than **8**, it will always be between **0** and **7**, which is exactly what we want.

So the formula to increment the **head** and **tail** pointers to make them **go round and round** over and again will be,

head = (head+1) % maxSize ,or

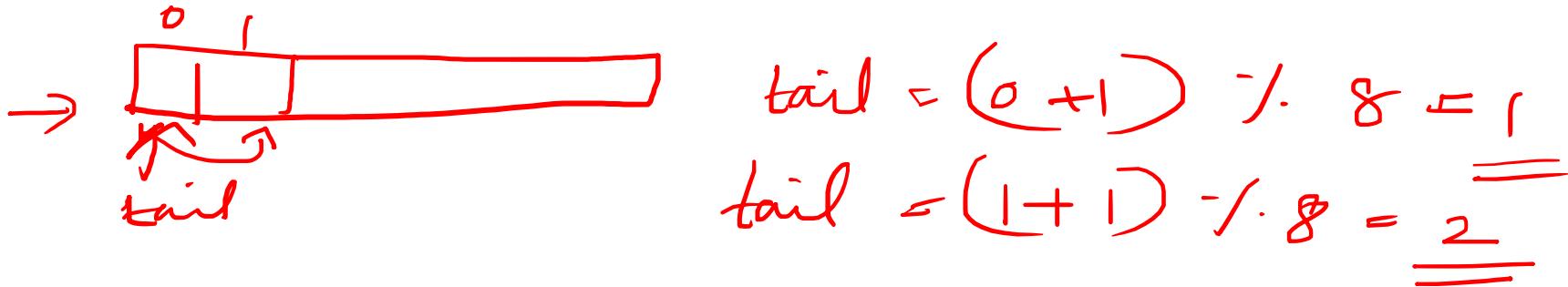
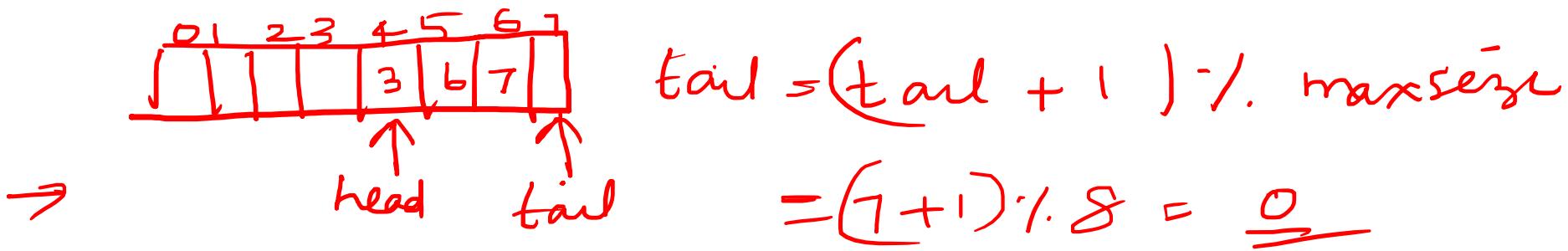
tail = (tail+1) % maxSize

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1.Computer controlled **Traffic Signal System** uses circular queue.

2.CPU scheduling and Memory management.



$\underline{\underline{\text{head} = 4}}$

$$\text{head} = (\underline{\underline{\text{head} + 1}}) \% 8$$

$$= \underline{\underline{5}}$$

$\underline{\underline{\text{head} = 7}}$

$$\text{head} = (\underline{\underline{\text{head} + 1}}) \% 8$$

$$= \underline{\underline{0}}$$

Implementation of Circular Queue

1. Initialize the queue, with size of the queue defined (**maxSize**), and **head** and **tail** pointers.

2. enqueue: Check if the number of elements is equal to **maxSize - 1**:

- If Yes, then return **Queue is full**.
- If No, then add the new data element to the location of **tail** pointer and increment the **tail** pointer.
$$(\text{tail} + 1) \% \text{max}$$

3. dequeue: Check if the number of elements in the queue is zero:

- If Yes, then return **Queue is empty**.
- If No, then increment the **head** pointer.

Enqueue

```
void enqueue(int queue[], int element, int& rear, int arraySize, int& count)
{
    if(count == arraySize) // Queue is full
        printf("OverFlow\n");
    else
    {
        queue[rear] = element;
        rear = (rear + 1)%arraySize;
        count = count + 1;
    }
}
```

Dequeue

```
void dequeue(int queue[], int& front, int rear, int& count)
{
    if(count == 0) // Queue is empty
        printf("UnderFlow\n");
    else
    {
        queue[front] = 0; // Delete the front element
        front = (front + 1)%arraySize;
        count = count - 1;
    }
}
```

Front

```
int Front(int queue[], int front)  
{  
    return queue[front];  
}
```

Size

```
int size(int count)  
{  
    return count;  
}
```

IsEmpty

```
bool isEmpty(int count)  
{  
    return (count == 0);  
}
```

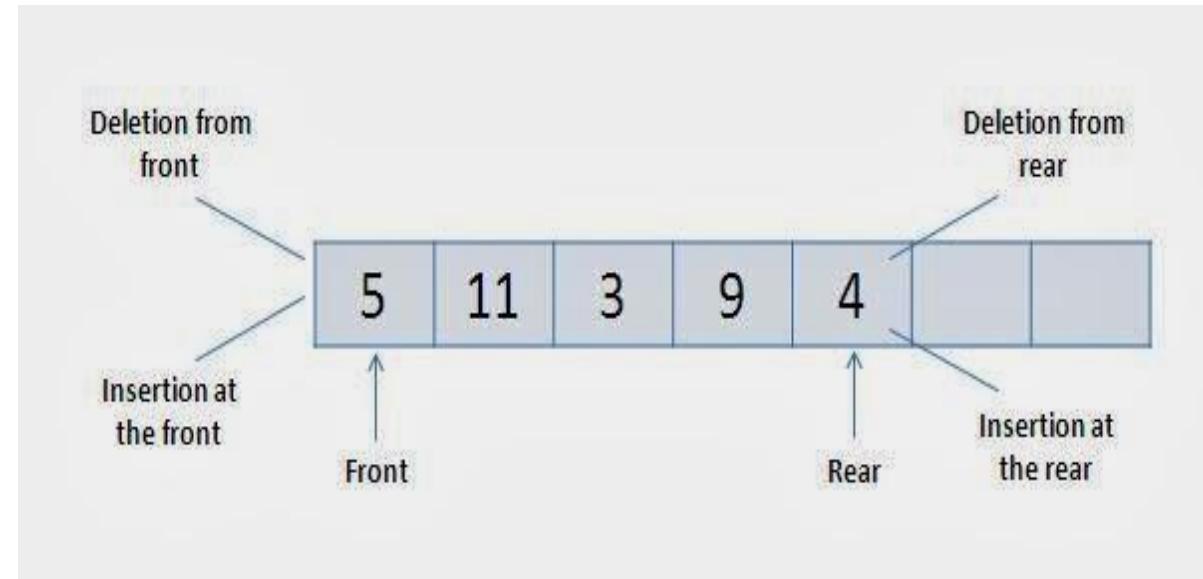
Double-ended queue

- In a standard queue, a character is **inserted at the back and deleted in the front.**

- However, in a double-ended queue, characters can be inserted and deleted from **both the front and back of the queue.**

Double Ended Queue

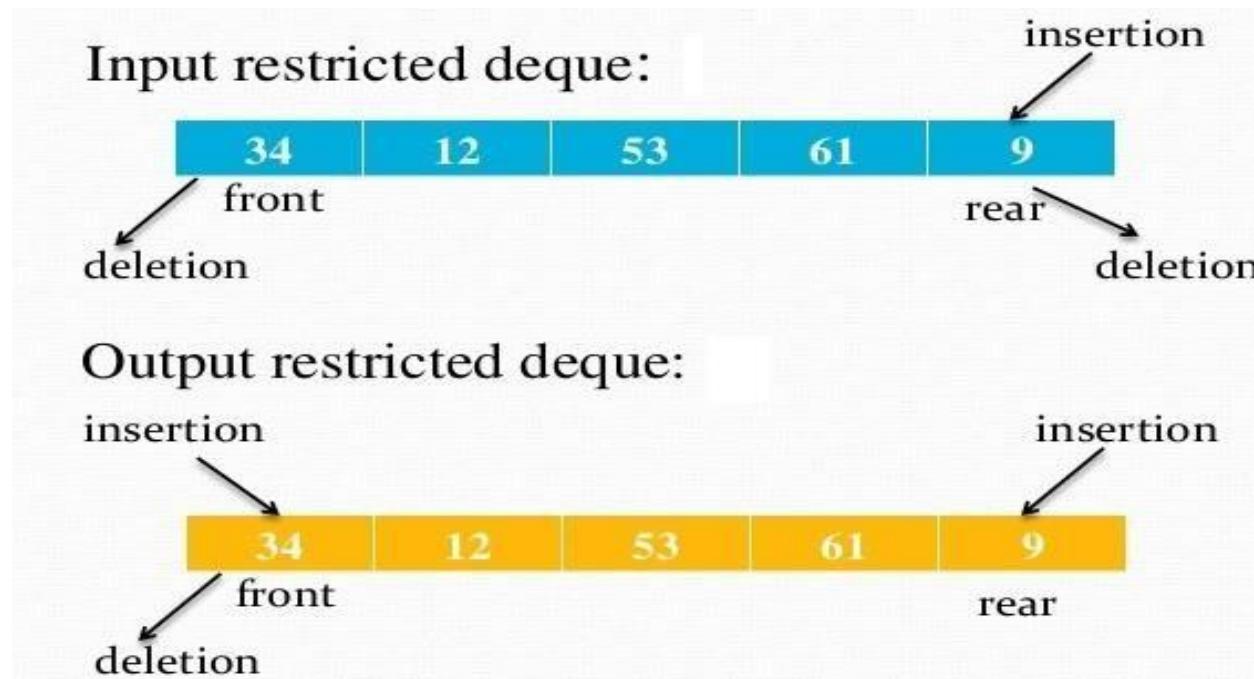
- **What is a double ended queue (deque) ?**
 - Queue in which insertion (enqueue) and deletion (dequeue) can be made at both ends
- **Operations on a deque**
 - insert_rear
 - insert_front
 - delete_front
 - delete_rear



Double Ended Queue

■ Types of deque

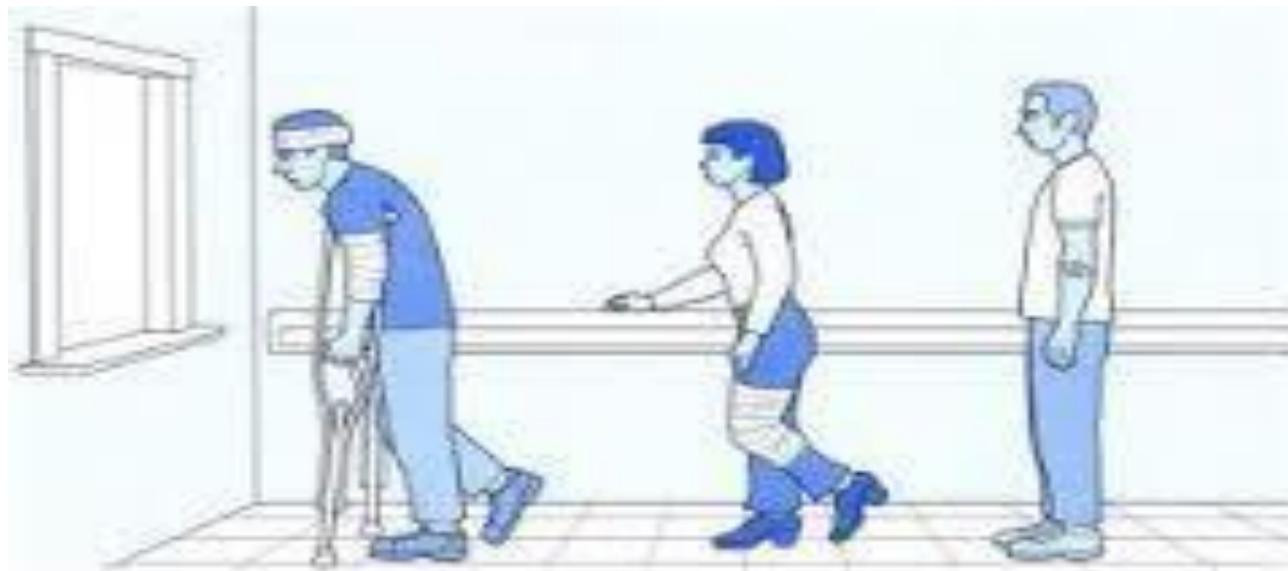
- 1. Input-restricted:** Insert at rear end only, delete from any end
- 2. Output-restricted:** Delete from front end only, insert at any end



Priority Queue

■ What is a priority queue?

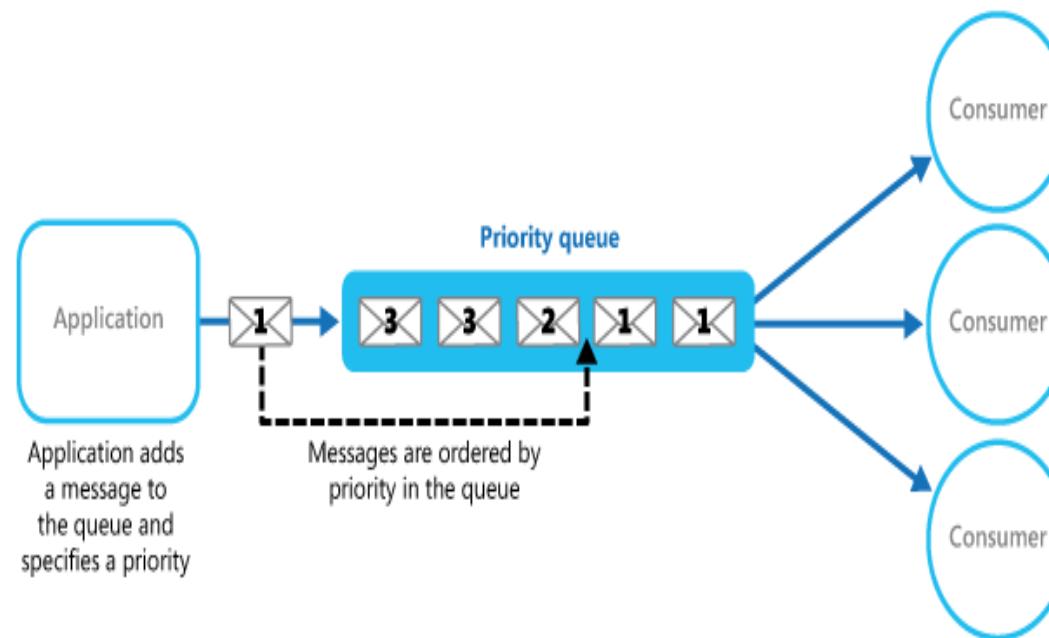
- Queue in which each element has a priority associated with it
- Priority – a unique number that determines the relative importance of that element compared to other



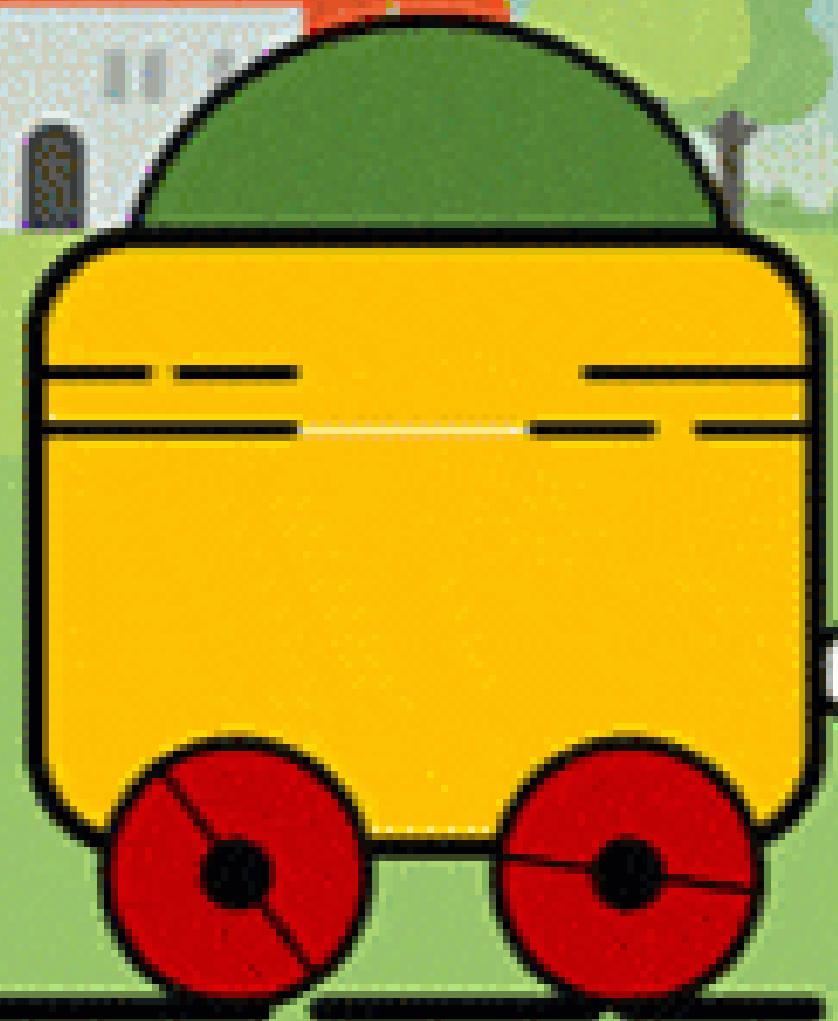
Priority Queue

■ Operations on Priority Queue

- Insertion according to priority
- Deletion as usual
- Types
 - **Max (descending) priority queue – delete element with highest priority**
 - **Min (ascending) priority queue – delete element with least priority**



END OF QUEUE



Lists.



General Linear Lists

Objectives.

- Explain the design, use, and operation of a linear list
- Implement a linear list using a linked list structure
- Understand the operation of the linear list ADT
- Write application programs using the linear list ADT
- Design and implement different link-list structures

Array versus Linked Lists

- Arrays are suitable for:
 - Inserting/deleting an element at the end.
 - Randomly accessing any element.
 - Searching the list for a particular value.
- Linked lists are suitable for:
 - Inserting an element.
 - Deleting an element.
 - Applications where sequential access is required.
 - In situations where the number of elements cannot be predicted beforehand.

List is an Abstract Data Type

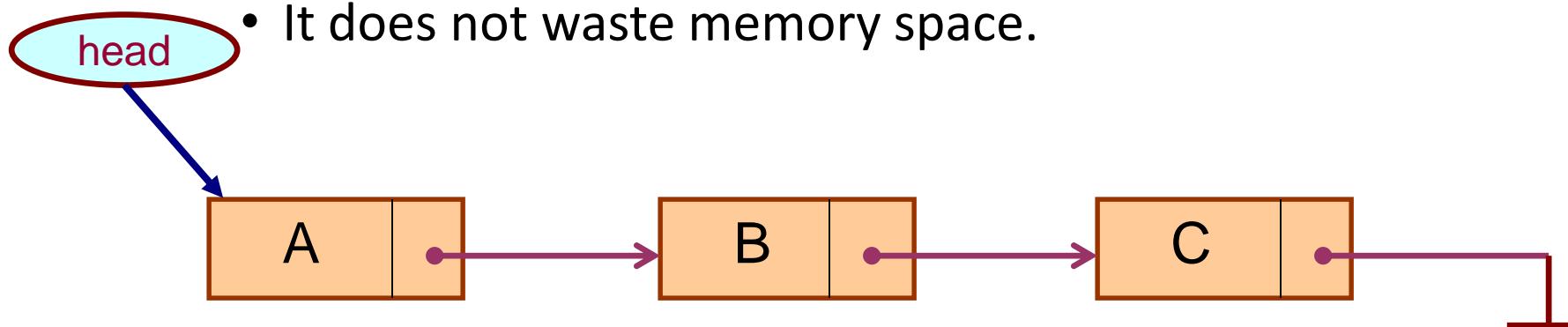
- What is an abstract data type?
 - It is a data type defined by the user.
 - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
 - Because details of the implementation are **hidden**.
 - When you do some operation on the list, say insert an element, you just call a function.
 - Details of how the list is implemented or how the insert function is written is no longer required.

General linear lists.

- A general linear list is a list in which operations can be done anywhere in the list.
- For simplicity, we refer to general linear lists as lists.

Linked List- Introduction

- A linked list is a data structure which can change during execution.
 - Successive elements are connected by pointers.
 - Last element points to `NULL`.
 - It can grow or shrink in size during execution of a program.
 - It can be made just as long as required.
 - It does not waste memory space.



Linked List- Introduction

- Keeping track of a linked list:
 - Must know the pointer to the first element of the list (called *start*, *head*, etc.).
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
 - Insert an element.
 - Delete an element.

Basic Operations.

📌 Insertion

📌 Deletion

📌 Retrieval

📌 Traversal

Insertion.

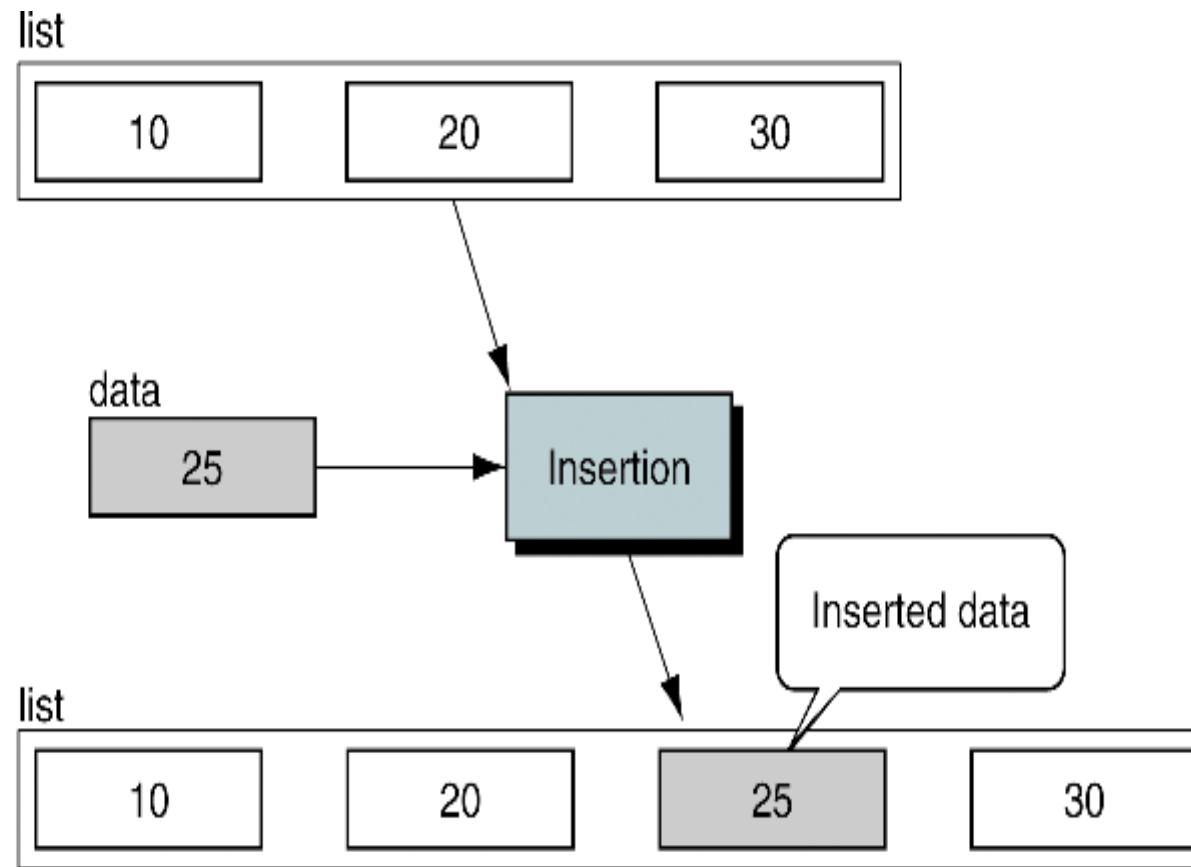


FIGURE 5-1 Insertion

Insertion is used to add a new element to the list

Deletion.

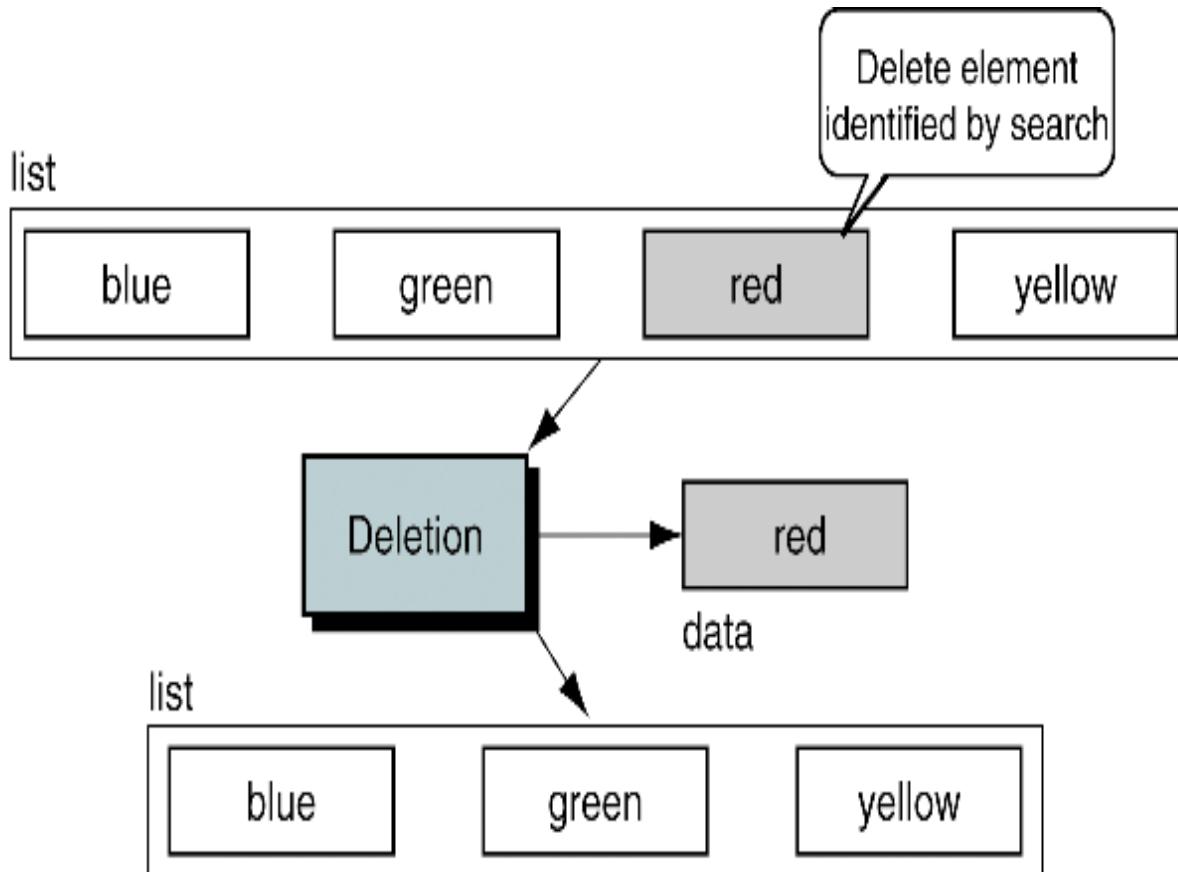


FIGURE 5-2 Deletion

Deletion is used to remove an element from the list.

Retrieval.

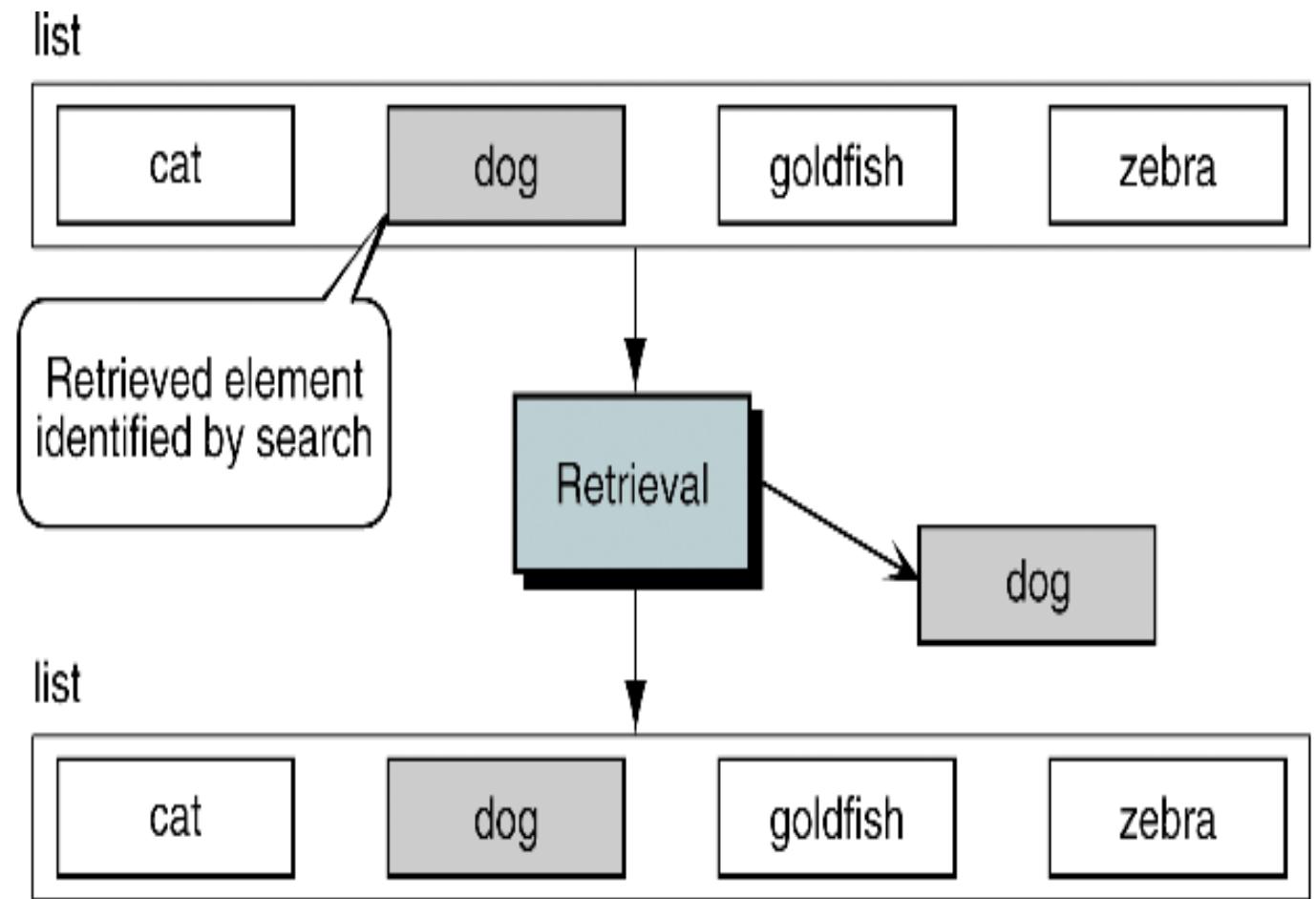
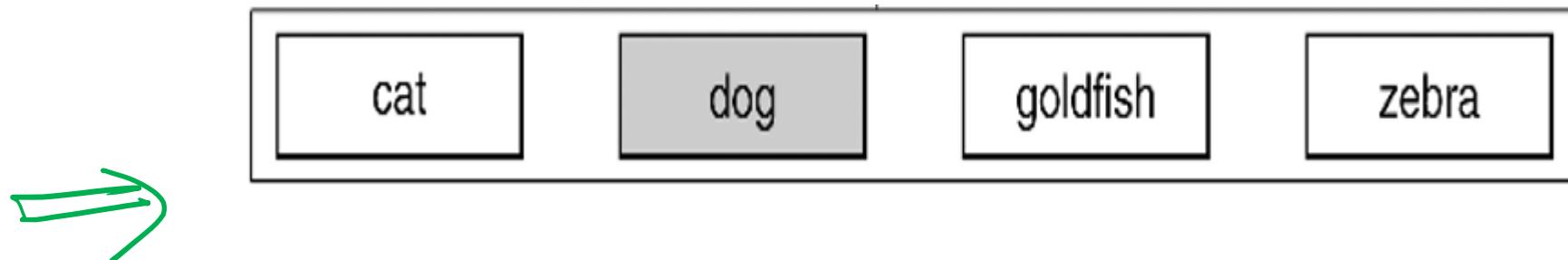


FIGURE 5-3 Retrieval

Retrieval is used to get the information related to an element without changing the structure of the list.

Traversal.

- List traversal **processes each element** in a list in sequence.



Implementation.

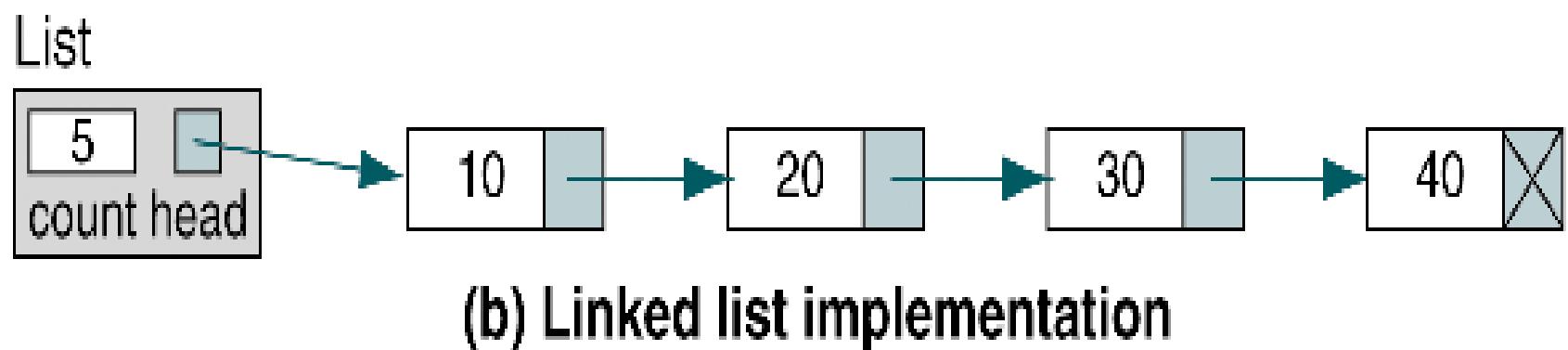
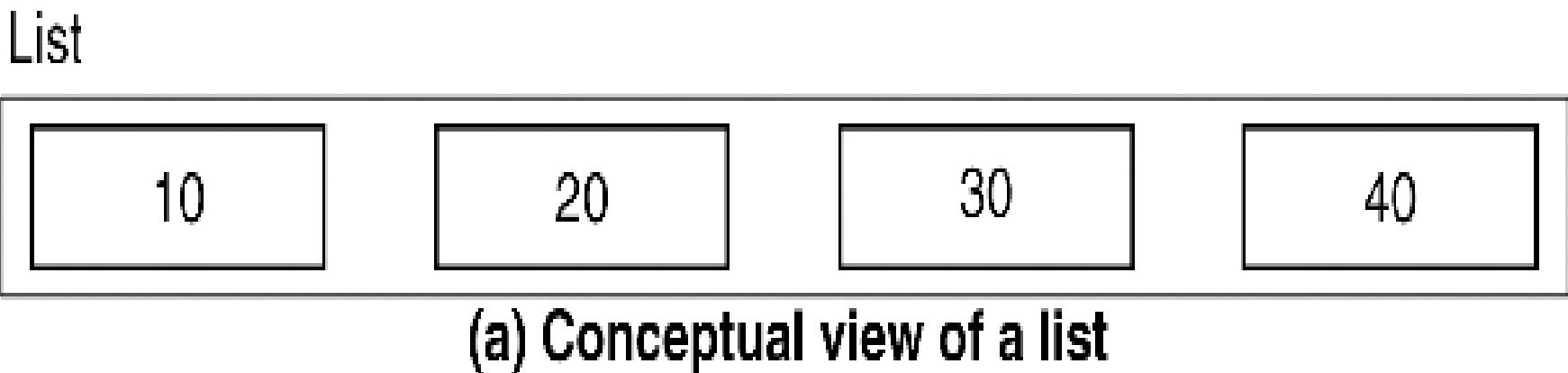
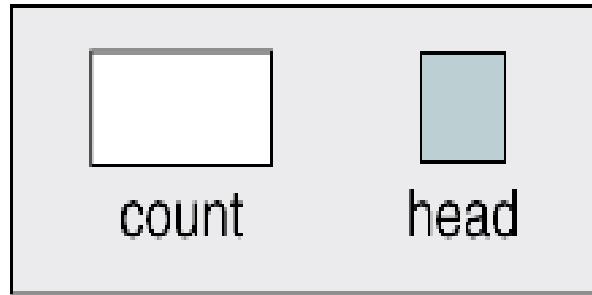
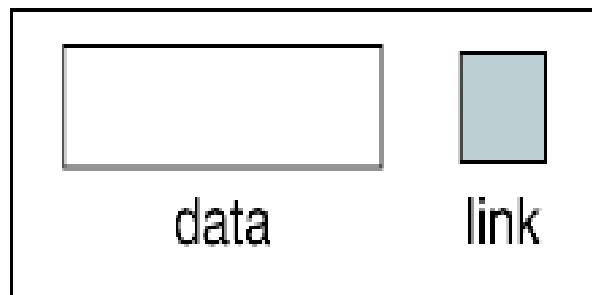


FIGURE 5-4 Linked List Implementation of a List

Data structure.



(a) Head structure



(b) Data node structure

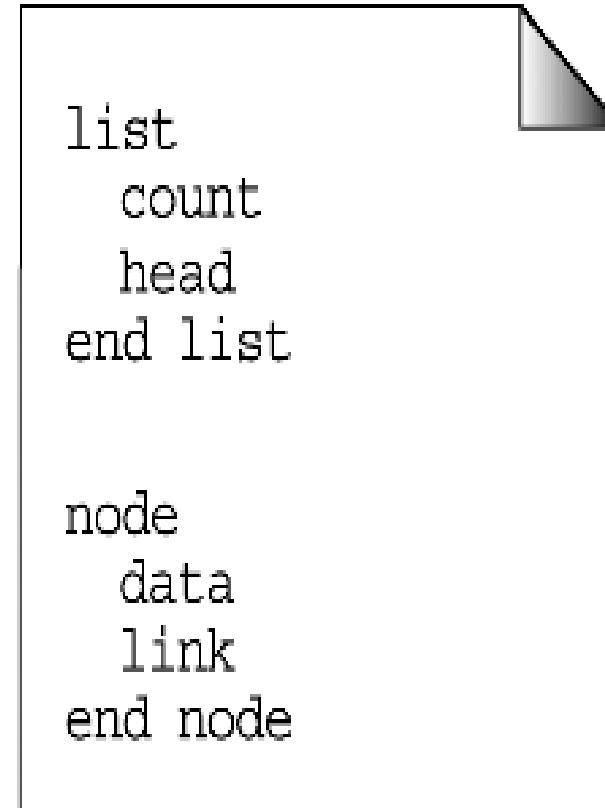
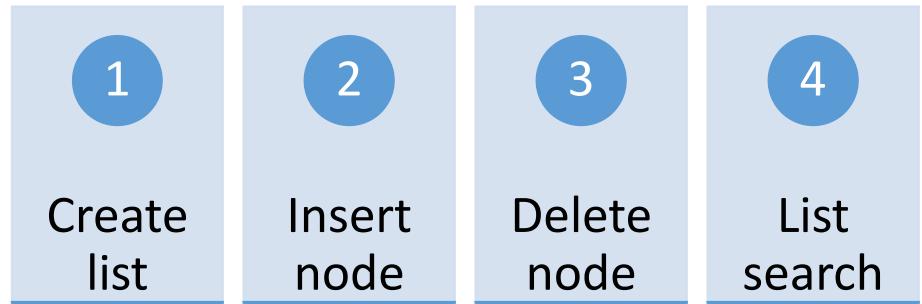


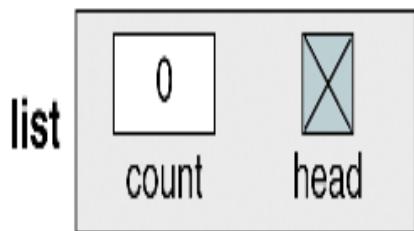
FIGURE 5-5 Head Node and Data Node

Algorithms.



Create list.

```
allocate (list)  
set list head to null  
set list count to 0
```

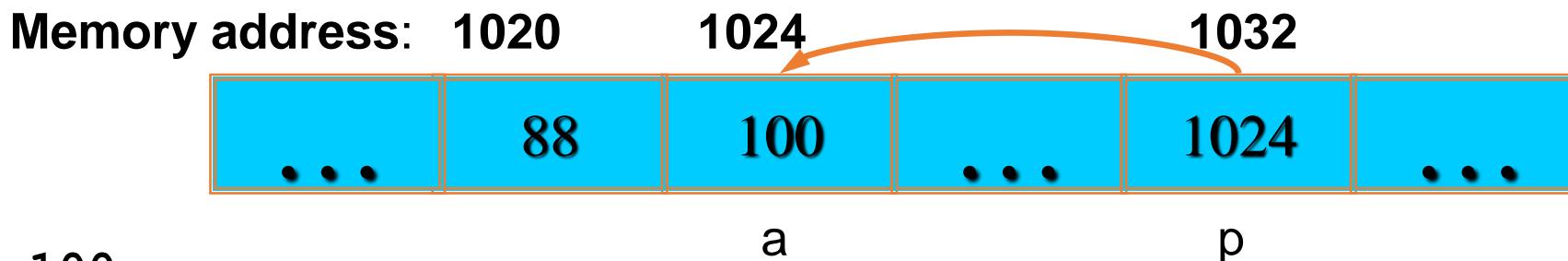


```
struct node  
{  
    int data;  
    struct node *next;  
};  
  
// Head pointer always points to  
// first element of the linked list  
  
struct node *head = new struct  
node; //create a node  
  
struct node *head = NULL;
```

FIGURE 5-6 Create List

Dereferencing Operator *

- We can access to the value stored in the variable pointed to by using the dereferencing operator (*),



```
int a = 100;  
int *p = &a;  
cout << a << endl;  
cout << &a << endl;  
cout << p << " " << *p << endl;  
cout << &p << endl;
```

□ Result is:
100
1024
1024 100
1032

Pointer to structures

- Like other types, structures can also be accessed through pointers

- Accessing structure itself by `*ptr`**
- ptr contains the address of the beginning of the structure**
- Now we do not only need to use structure name with member operator such as `sam1.x`**
- we can also use `(*ptr).x`**

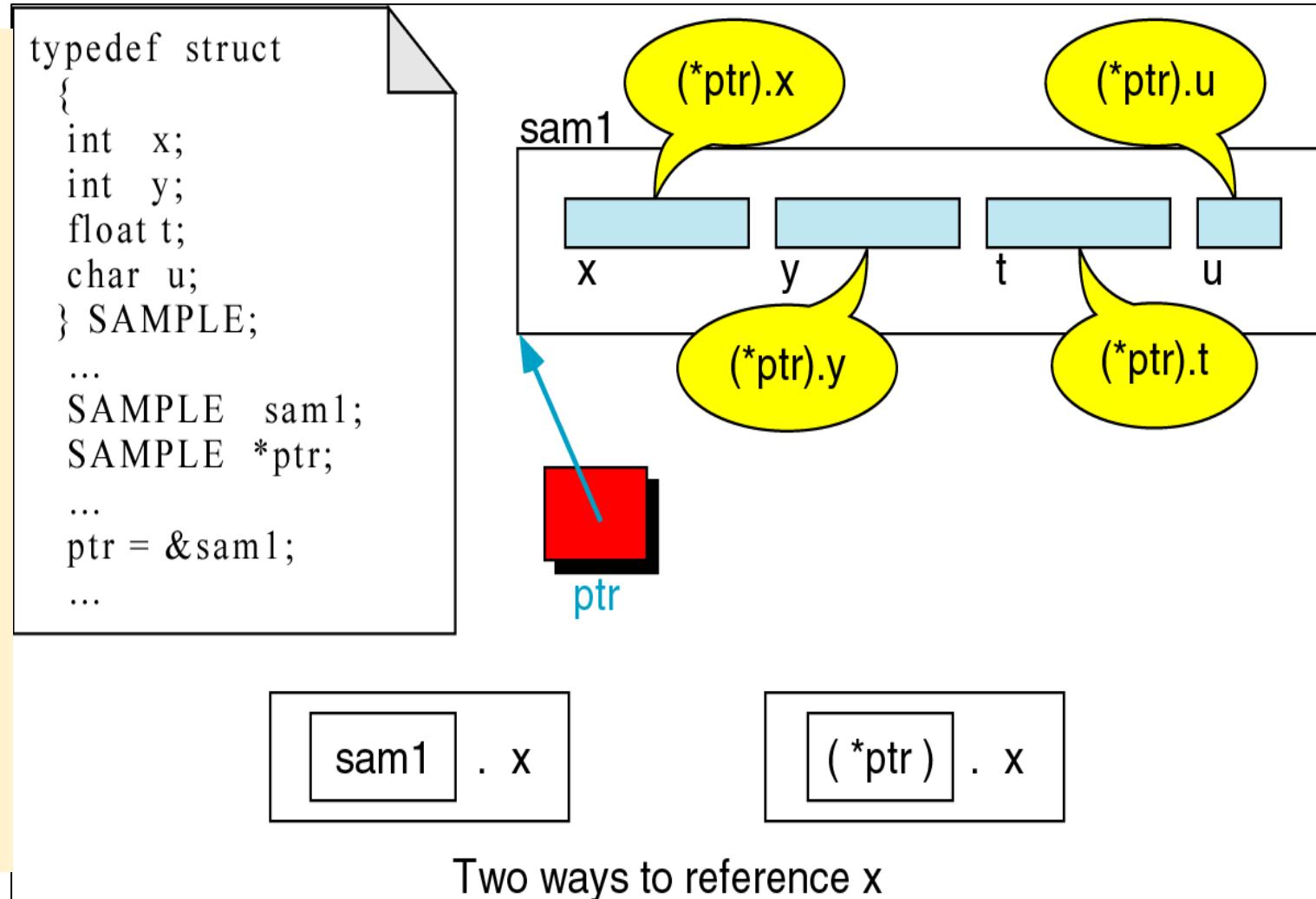
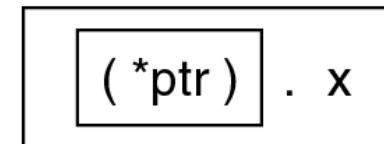
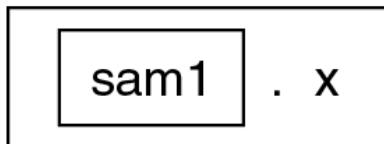
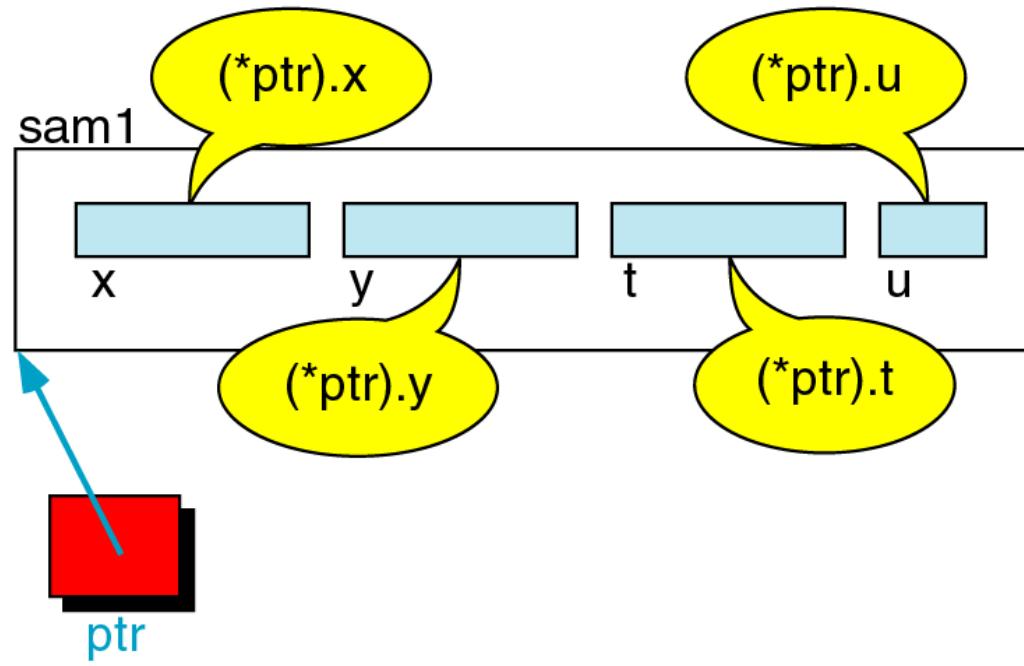


Figure 12-14 Pointers to structures

```
typedef struct
{
    int x;
    int y;
    float t;
    char u;
} SAMPLE;
...
SAMPLE sam1;
SAMPLE *ptr;
...
ptr = &sam1;
...
```



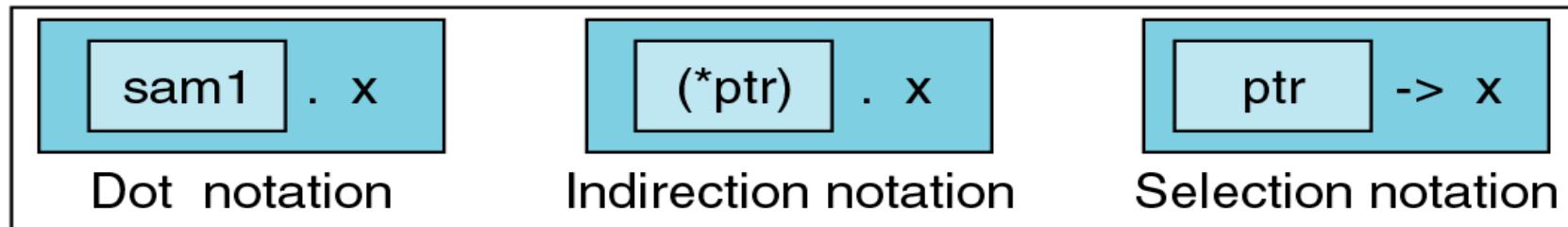
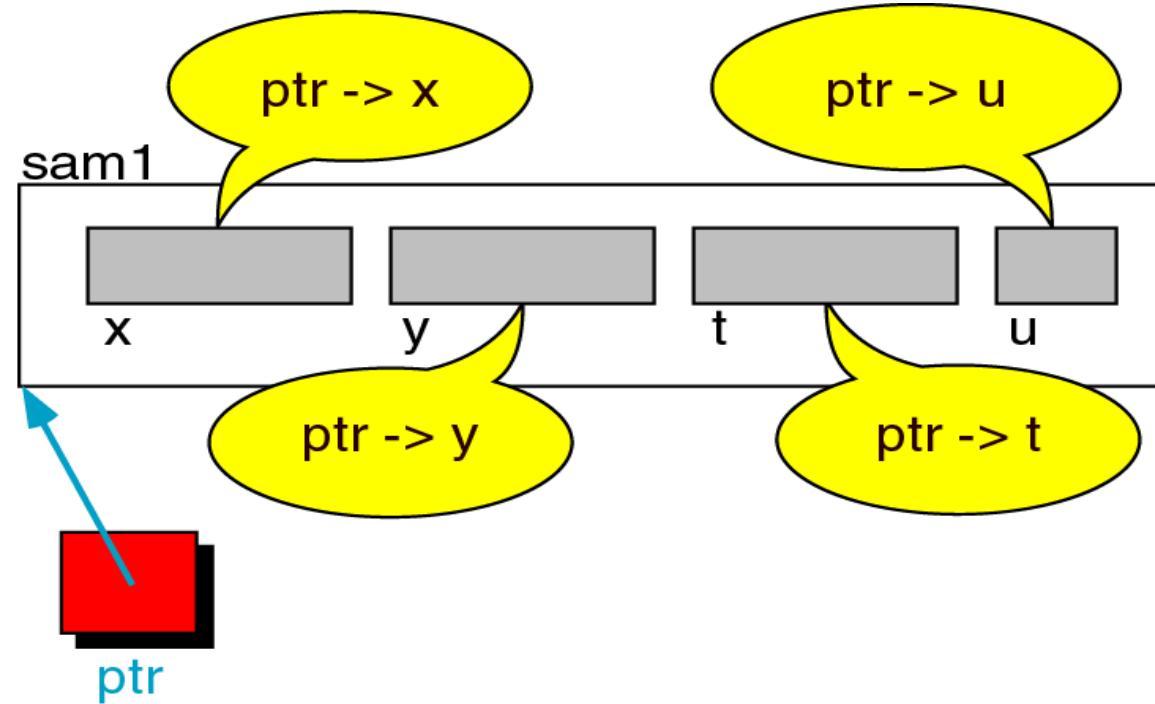
Two ways to reference x

Selection operator

- However, there is a selection operator `->` (minus sign and greater than symbol) to eliminate the problem of pointer to structures
- The priority of selection operator (`->`) and member operator(`.`) are the same
- The expressions :
 - **(*pointerName).fieldName is same as pointerName ->fieldName**
 - But `pointerName -> fieldName` is preferred

Figure 12-16 pointer selection operator

```
typedef struct
{
    int x;
    int y;
    float t;
    char u;
} SAMPLE;  
...  
SAMPLE sam1;  
SAMPLE *ptr;  
...  
ptr = &sam1;  
...
```



three ways to reference the field **x**



Pointers and Structures I

```
struct inventory
```

```
{
```

```
    char name[30];
```

```
    int number;
```

```
    float price;
```

```
}product[2],*ptr;
```

- **ptr=product;**
 - Its members are accessed using the following notation
 - ptr→name**
 - ptr→number**
 - ptr→price**



Pointers and Structures II

- The symbol → is called **arrow operator** (also known as member selection operator)
- The data members can also be accessed using **(*ptr).number**
- Parentheses is required because '.' has higher precedence than the operator *



Dynamic Memory Allocation I

- Dynamic memory allocation and deallocation is done using two operators: **new** and **delete**. An object can be created using new operator and destroyed by delete operator.
- A data object created inside a block with new will remain in existence until it is destroyed by using delete.

Pointer_variable = new data_type;

- For example:
p = new int;
q = new float;
- Alternatively, **int *p = new int;**
- **float *q = new float;**

Create list.

ALGORITHM 5-1 Create List

```
Algorithm createList (list)
Initializes metadata for list.

    Pre    list is metadata structure passed by reference
    Post   metadata initialized

1 allocate (list)
2 set list head to null
3 set list count to 0
end createList
```

Insert node.

- ⌚ Only its **logical predecessor** is needed.
- ⌚ There are three steps to the insertion:
 - ⌚ **Allocate memory for the new node** and move data to the node.
 - ⌚ Point the **new node to its successor**.
 - ⌚ Point the **new node's predecessor to the new node**.

Insert node.

1. Insert into empty list
2. Insert at beginning
3. Insert in middle
4. Insert at end

Insert into empty list.

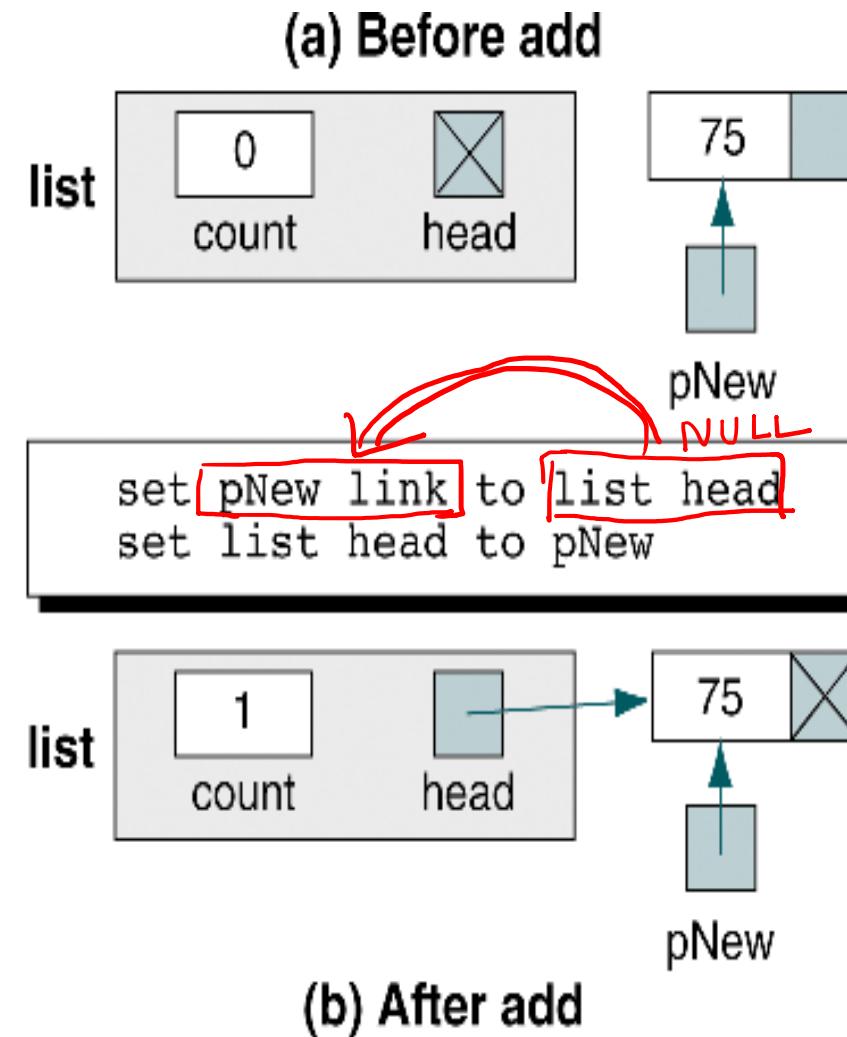


FIGURE 5-7 Add Node to Empty List

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *head = NULL;
```

```
struct node *newNode = new struct node; //create a node  
newNode->data = value;
```

```
newNode->next = head;  
head = newNode
```

Insert at beginning.

```
struct node *newNode = new struct node;  
//create a node  
newNode->data = value;  
  
newNode->next = head;  
head = newNode
```

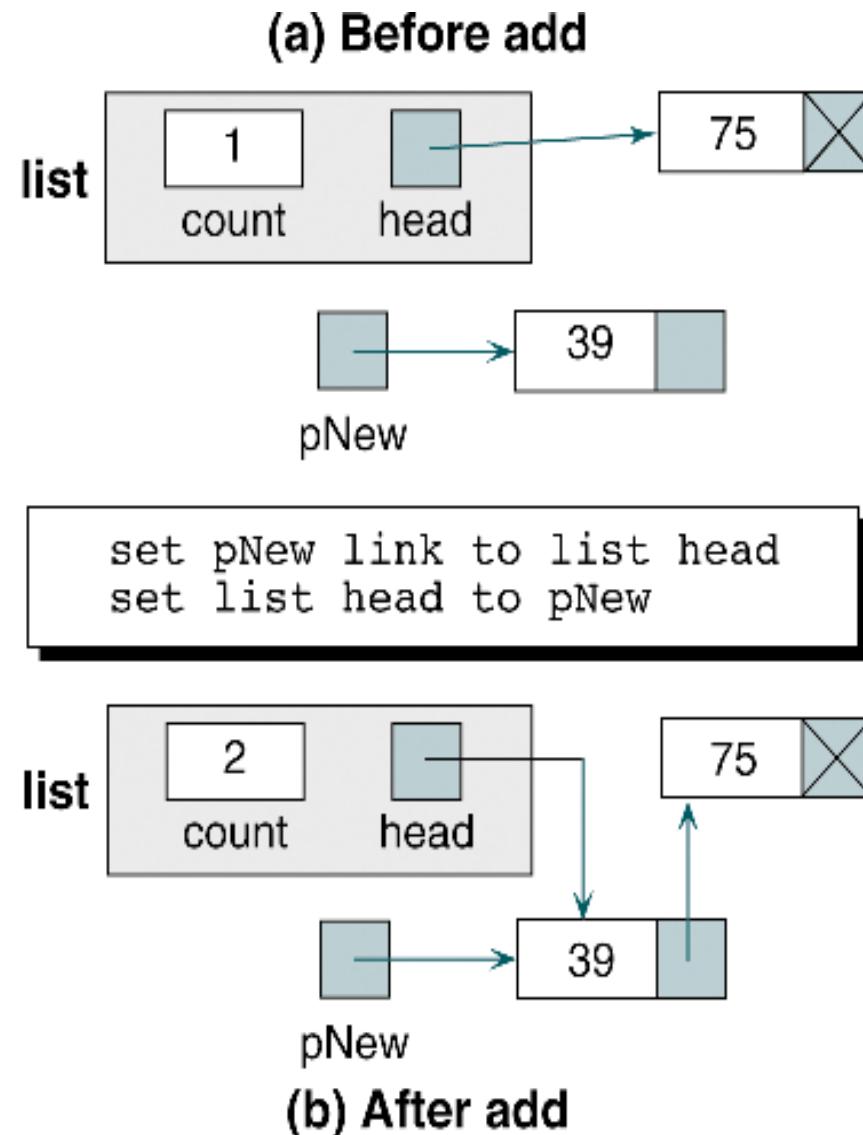


FIGURE 5-8 Add Node at Beginning

Insert in middle.

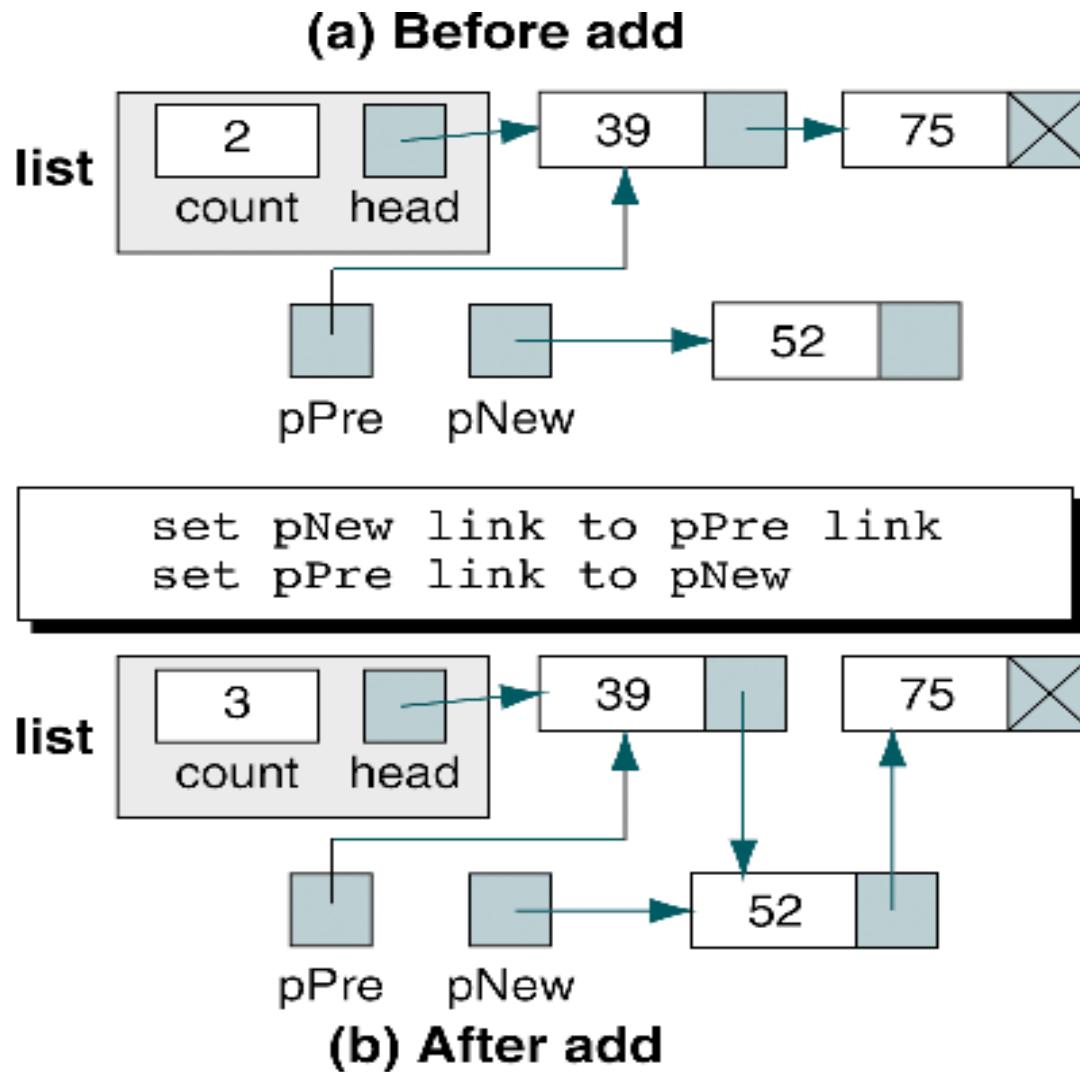


FIGURE 5-9 Add Node in Middle

Insert at the Middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode = new struct node; //create a node
newNode->data = value;
struct node *temp = head;
for(int i=1; i < position; i++)
{
if(temp->next != NULL)
{
    temp = temp->next;
}
newNode->next = temp->next;
temp->next = newNode; or temp=newNode;
```

Insert at end.

Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

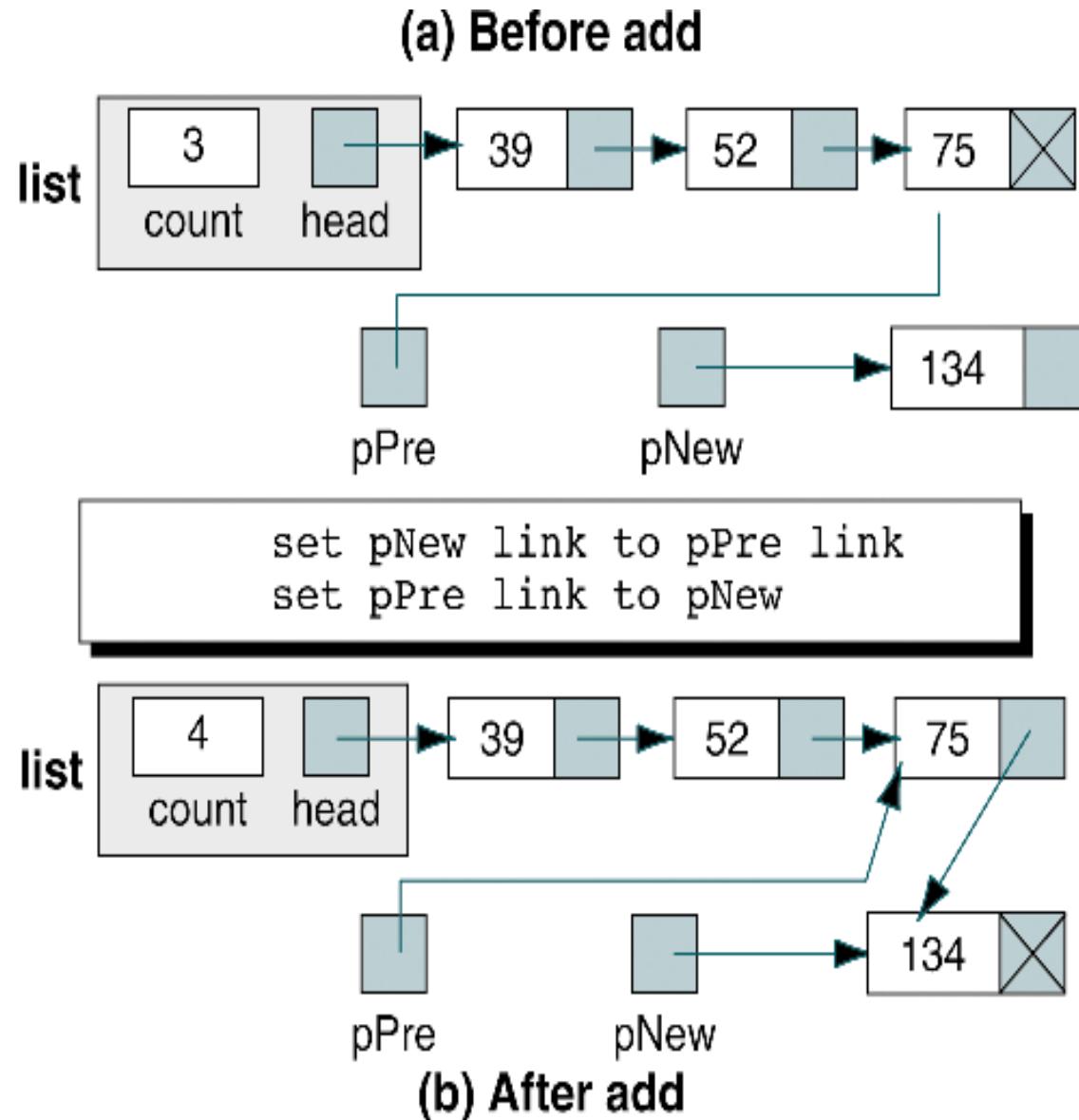


FIGURE 5-10 Add Node at End

Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode = new struct node; // Create a new node
newNode->data = data;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL)
{
    temp = temp->next;
}
temp->next = newNode; or temp=newNode;
```

ALGORITHM 5-2 Insert List Node

```
Algorithm insertNode (list, pPre, dataIn)
Inserts data into a new node in the list.

    Pre    list is metadata structure to a valid list
           pPre is pointer to data's logical predecessor
           dataIn contains data to be inserted

    Post   data have been inserted in sequence
           Return true if successful, false if memory overflow

1 allocate (pNew)
2 set pNew data to dataIn
3 if (pPre null)
    Adding before first node or to empty list.
    1 set pNew link to list head
    2 set list head to pNew
4 else
    Adding in middle or at end.
    1 set pNew link to pPre link
    2 set pPre link to pNew
5 end if
6 return true
end insertNode
```

Delete node.

1. Delete first node
2. General delete case

Delete first node

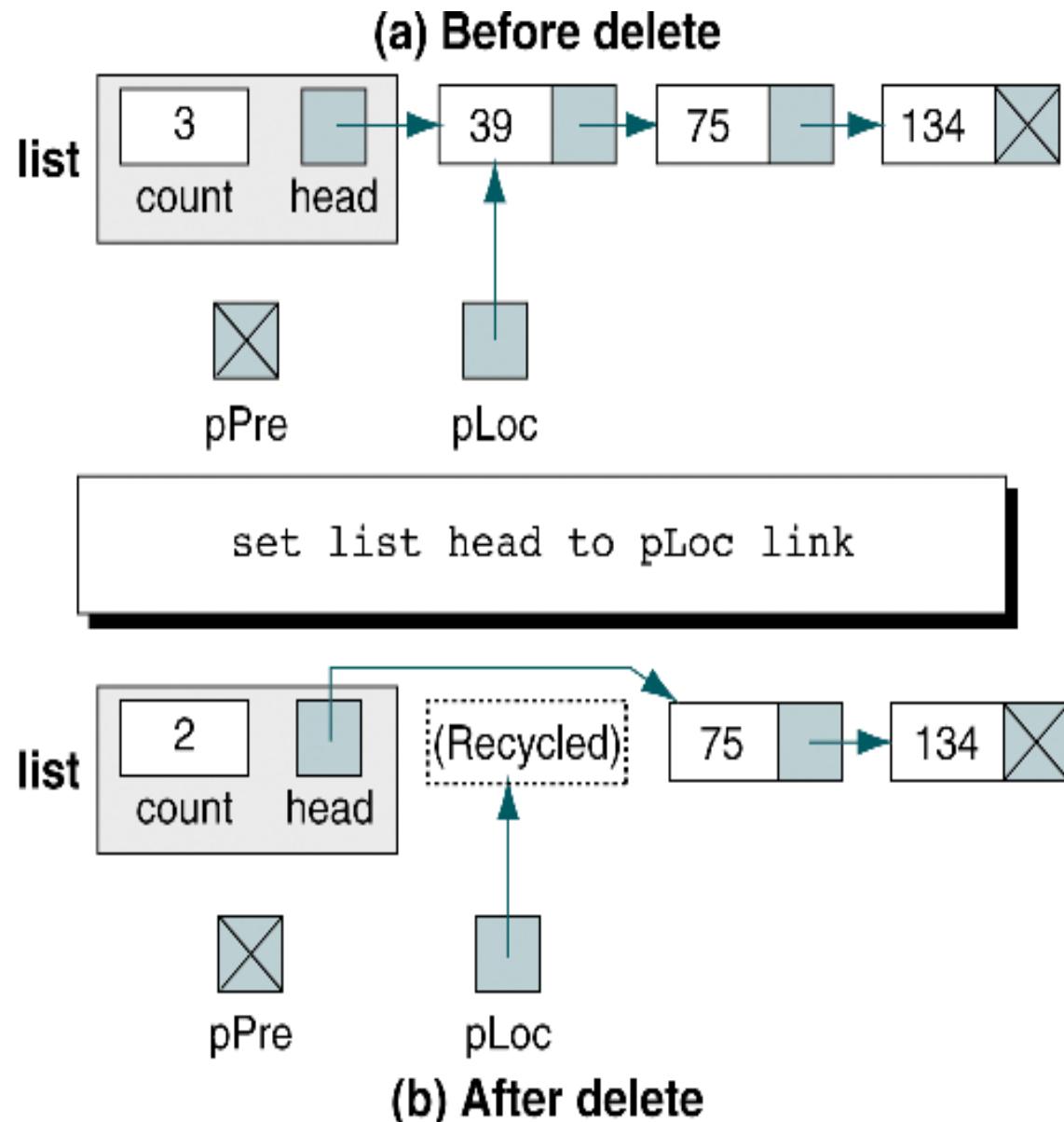


FIGURE 5-11 Delete First Node

Delete from beginning

- Point head to the second node

```
Struct node *temp = head;  
head = head->next
```

```
//free memory used by temp  
temp=NULL;  
delete temp;
```

Delete general case.

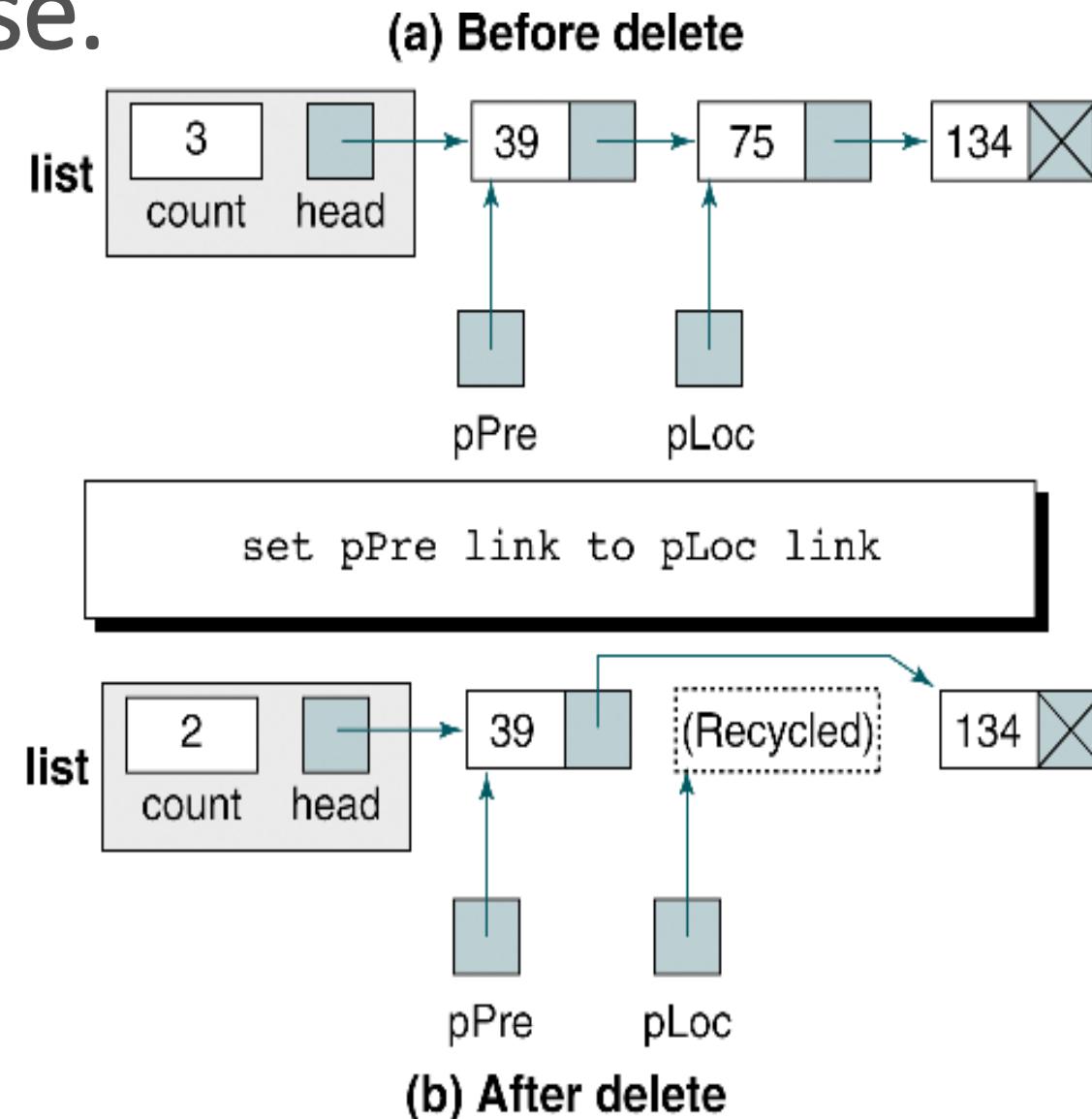


FIGURE 5-12 Delete General Case

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
struct node *p=head;  
struct node *q=head->next;  
for(i=1; i<position ;i++)  
{ p=p->next;  
    q=q->next;  
}  
p->next=q->next;  
free(q)  
return head;
```

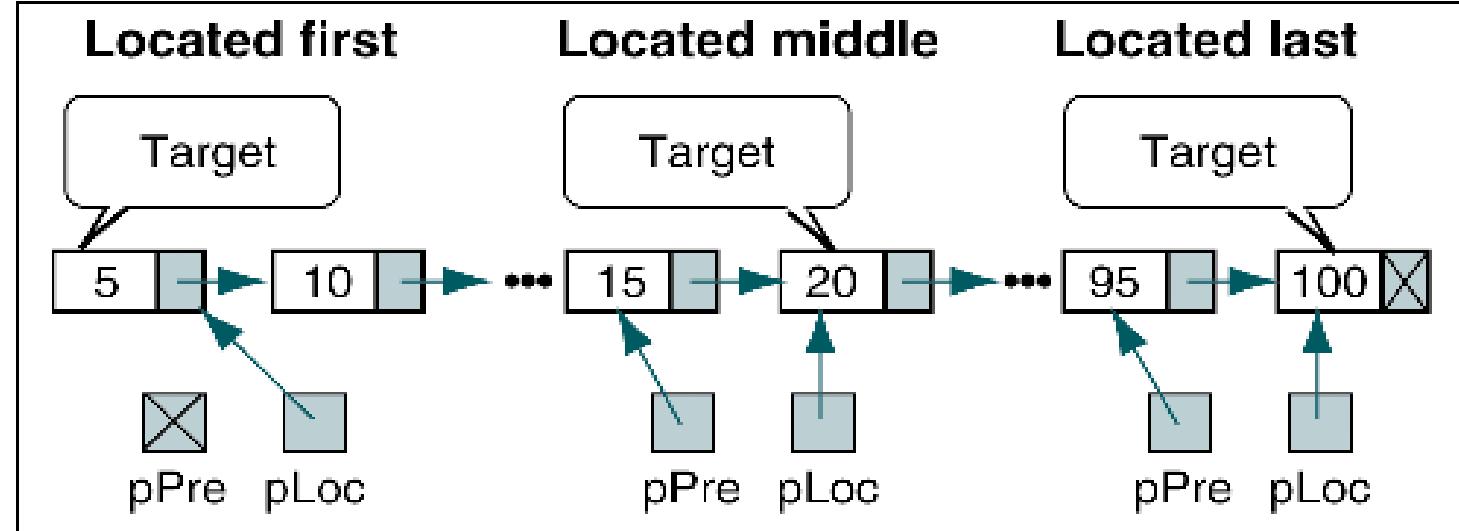
ALGORITHM 5-3 List Delete Node

```
Algorithm deleteNode (list, pPre, pLoc, dataOut)
Deletes data from list & returns it to calling module.

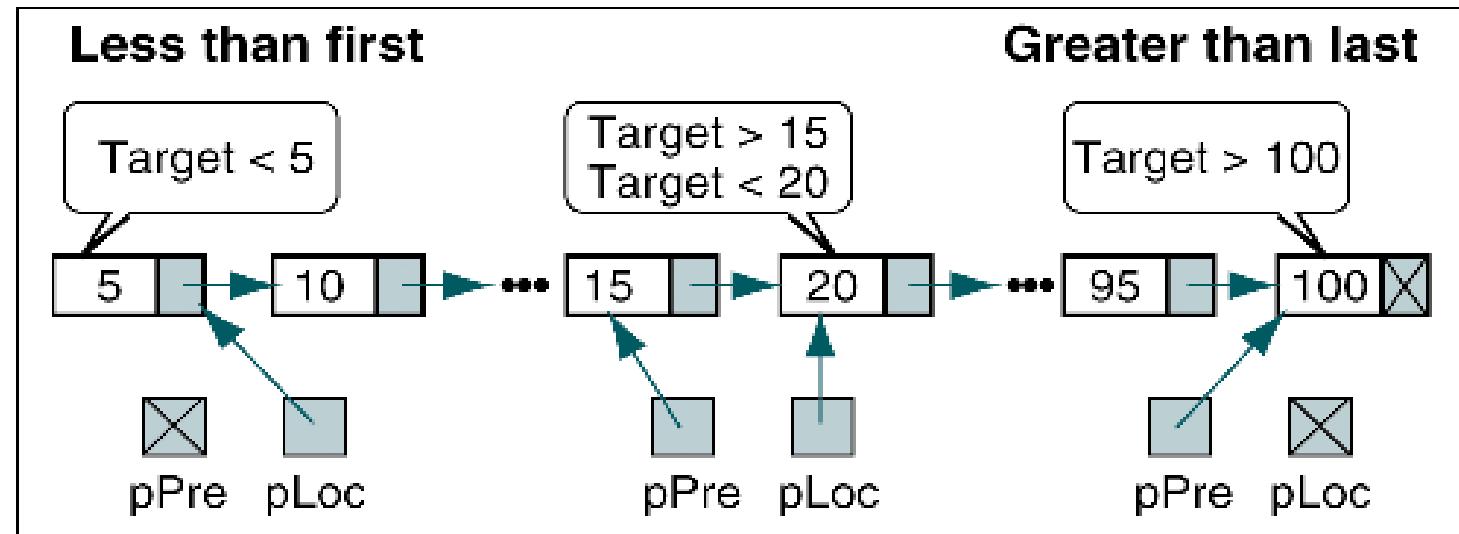
    Pre  list is metadata structure to a valid list
        Pre is a pointer to predecessor node
        pLoc is a pointer to node to be deleted
        dataOut is variable to receive deleted data
    Post data have been deleted and returned to caller

1 move pLoc data to dataOut
2 if (pPre null)
    Deleting first node
    1 set list head to pLoc link
3 else
    Deleting other nodes
    1 set pPre link to pLoc link
4 end if
5 recycle (pLoc)
end deleteNode
```

List search



(a) Successful searches (return true)



(b) Unsuccessful searches (return false)

List search.

Condition	pPre	pLoc	Return
Target < first node	Null	First node	False
Target = first node	Null	First node	True
First < target < last	Largest node < target	First node > target	False
Target = middle node	Node's predecessor	Equal node	True
Target = last node	Last's predecessor	Last node	True
Target > last node	Last node	Null	False

TABLE 5-1 List Search Results

ALGORITHM 5-4 Search List

```
Algorithm searchList (list, pPre, pLoc, target)
Searches list and passes back address of node containing
target and its logical predecessor.

Pre   list is metadata structure to a valid list
      pPre is pointer variable for predecessor
      pLoc is pointer variable for current node
      target is the key being sought

Post  pLoc points to first node with equal/greater key
      -or- null if target > key of last node
      pPre points to largest node smaller than key
      -or- null if target < key of first node

Return true if found, false if not found

1 set pPre to null
2 set pLoc to list head
3 loop (pLoc not null AND target > pLoc key)
    1 set pPre to pLoc
    2 set pLoc to pLoc link
4 end loop
5 if (pLoc null)
    Set return value
    1 set found to false
```

ALGORITHM 5-4 Search List (*continued*)

```
6 else
    1 if (target equal pLoc key)
        1 set found to true
    2 else
        1 set found to false
    3 end if
7 end if
8 return found
end searchList
```

Retrieve node.

ALGORITHM 5-5 Retrieve List Node

```
Algorithm retrieveNode (list, key, dataOut)
Retrieves data from a list.

Pre    list is metadata structure to a valid list
       key is target of data to be retrieved
       dataOut is variable to receive retrieved data
Post   data placed in dataOut
       -or- error returned if not found
Return true if successful, false if data not found

1 set found to searchList (list, pPre, pLoc, key)
2 if (found)
3     1 move pLoc data to dataOut
4 end if
5 return found
end retrieveNode
```

Empty list.

ALGORITHM 5-6 Empty List

Algorithm emptyList (list)

Returns Boolean indicating whether the list is empty.

Pre list is metadata structure to a valid list

Return true if list empty, false if list contains data

1 if (list count equal 0)

 1 return true

2 else

 1 return false

end emptyList

Full list.

ALGORITHM 5-7 Full List

```
Algorithm fullList (list)
```

Returns Boolean indicating whether or not the list is full.

Pre list is metadata structure to a valid list

Return false if room for new node; true if memory full

```
1 if (memory full)
```

```
    1 return true
```

```
2 else
```

```
    2 return false
```

```
3 end if
```

```
4 return true
```

```
end fullList
```

List count.

ALGORITHM 5-8 List Count

```
Algorithm listCount (list)
    Returns integer representing number of nodes in list.
        Pre    list is metadata structure to a valid list
        Return count for number of nodes in list
    1 return (list count)
end listCount
```

Traversal list.

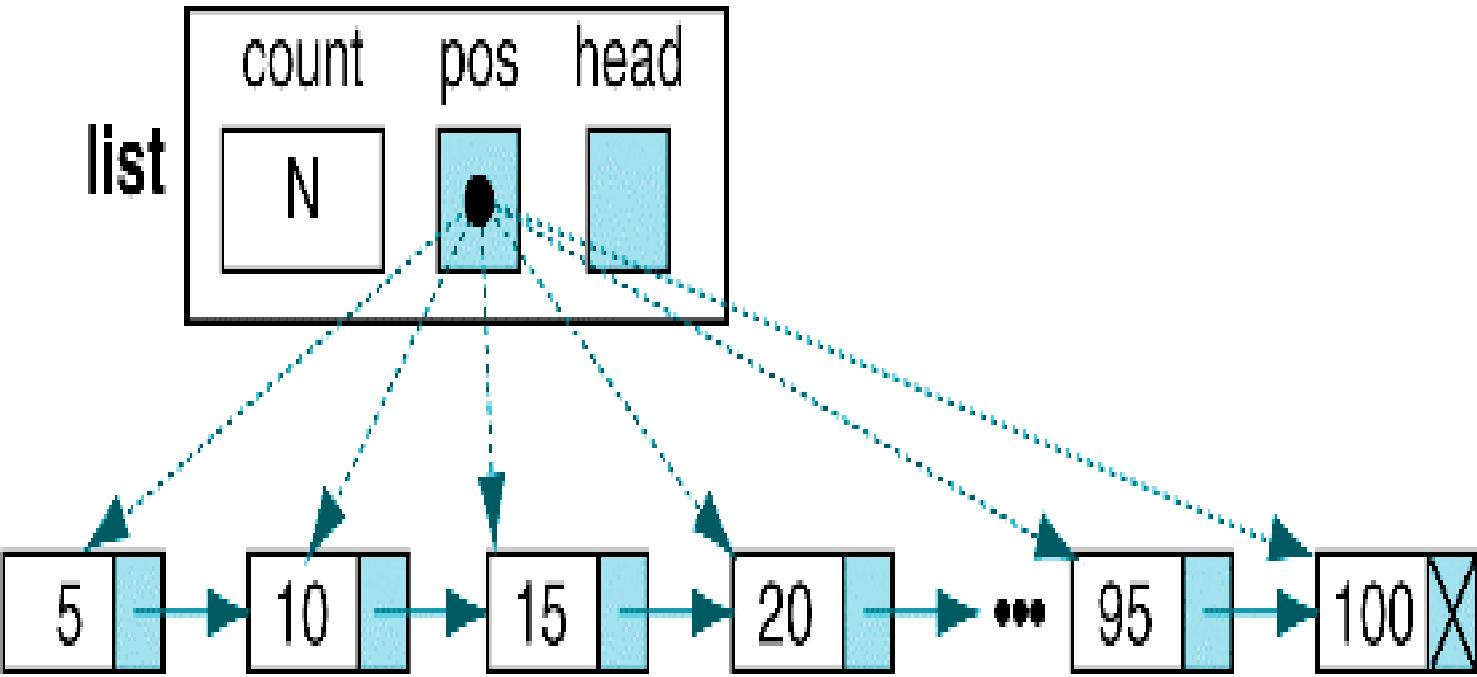


FIGURE 5-14 List Traversal

ALGORITHM 5-9 Traverse List

```
Algorithm getNext (list, fromWhere, dataOut)
Traverses a list. Each call returns the location of an
element in the list.
```

Pre list is metadata structure to a valid list
 fromWhere is 0 to start at the first element
 dataOut is reference to data variable
Post dataOut contains data and true returned
 -or- if end of list, returns false

continued

ALGORITHM 5-9 Traverse List (*continued*)

```
    Return true if next element located  
        false if end of list  
1 if (empty list)  
1 return false  
2 if (fromWhere is beginning)  
    Start from first  
1 set list pos to list head  
2 move current list data to dataOut  
3 return true  
3 else  
    Continue from pos  
1 if (end of list)  
    End of List  
1 return false  
2 else  
1 set list pos to next node  
2 move current list data to dataOut  
3 return true  
3 end if  
4 end if  
end getNext
```

Destroy list.

ALGORITHM 5-10 Destroy List

```
Algorithm destroyList (pList)
```

Deletes all data in list.

Pre list is metadata structure to a valid list

continued

ALGORITHM 5-10 Destroy List (*continued*)

Post All data deleted

1 loop (not at end of list)

 1 set list head to successor node

 2 release memory to heap

2 end loop

 No data left in list. Reset metadata.

3 set list pos to null

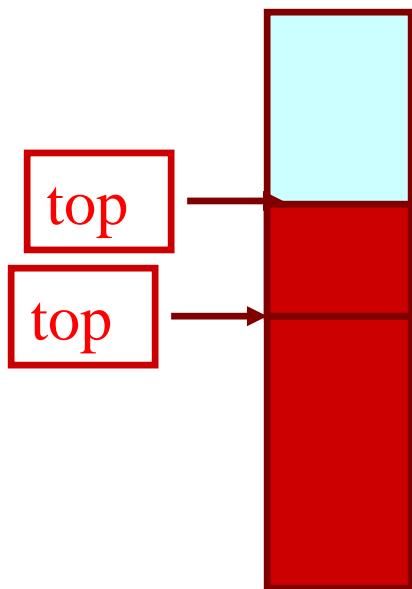
4 set list count to 0

end destroyList

Stack Implementations: Using Array and Linked List

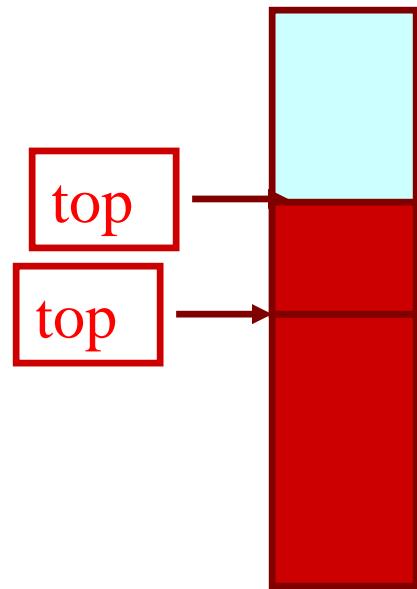
STACK USING ARRAY

PUSH



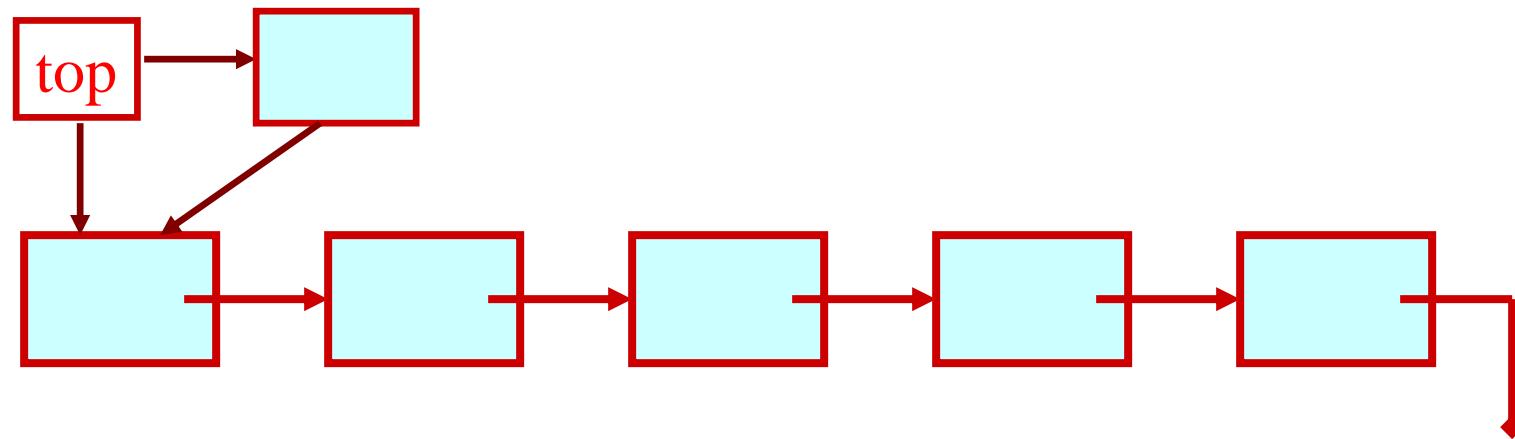
STACK USING ARRAY

POP



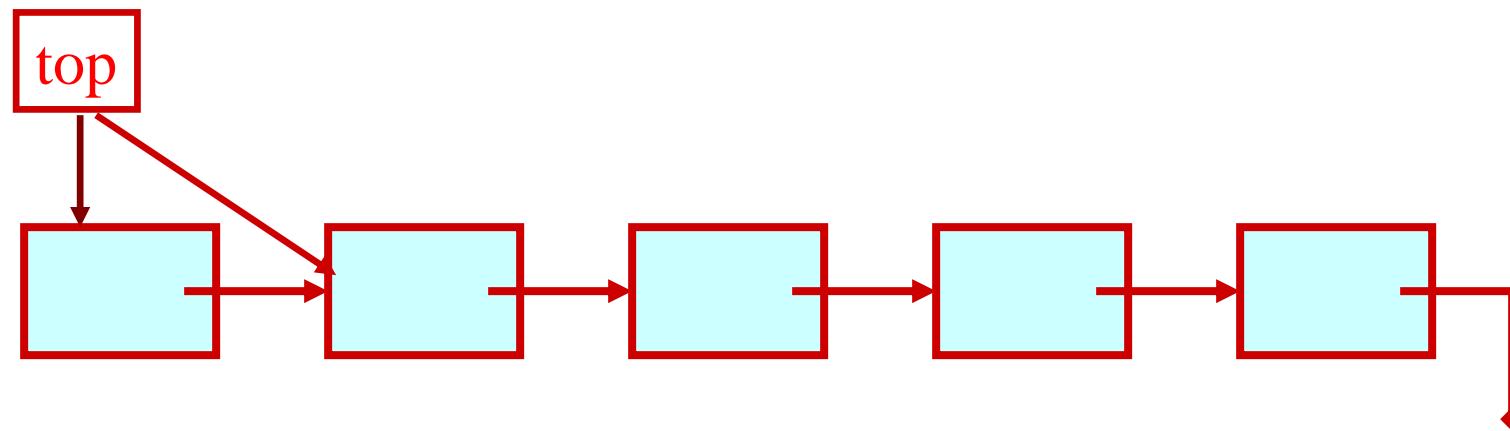
Stack: Linked List Structure

PUSH OPERATION



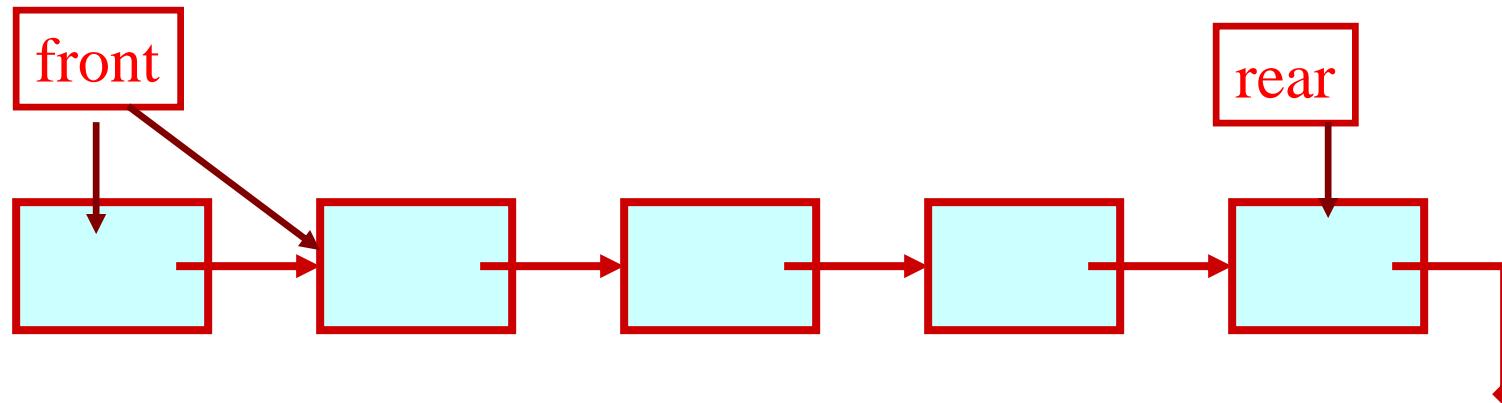
Stack: Linked List Structure

POP OPERATION



QUEUE: LINKED LIST STRUCTURE

DEQUEUE



Linked List Complexity

Time Complexity

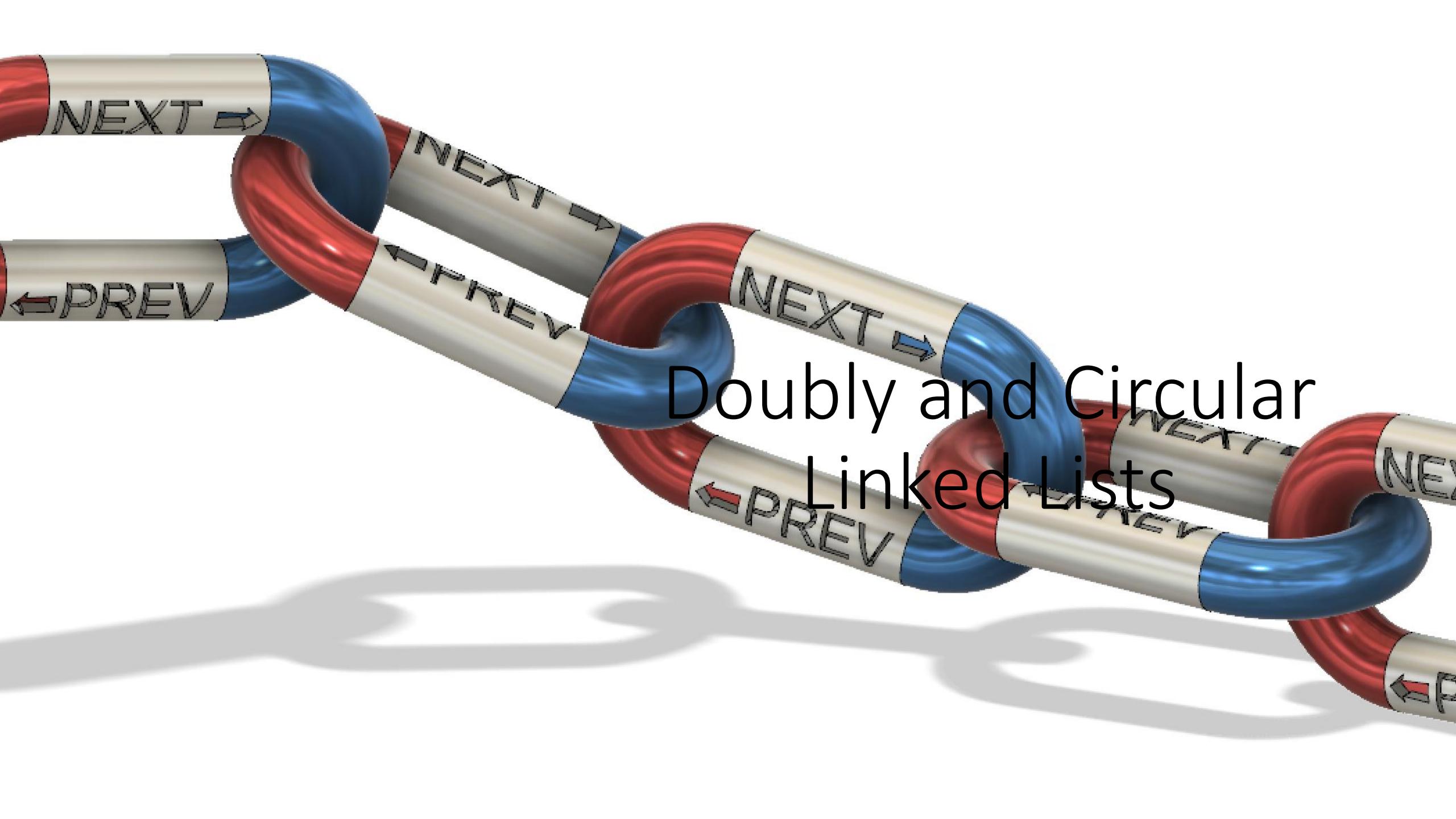
	Worst case	Average Case
Search	$O(n)$	$O(n)$
Insert	$O(1)$	$O(1)$
Deletion	$O(1)$	$O(1)$

Space Complexity: $O(n)$

Linked List Applications
Dynamic memory allocation
Implemented in stack and queue
In undo functionality of softwares
Hash tables, Graphs

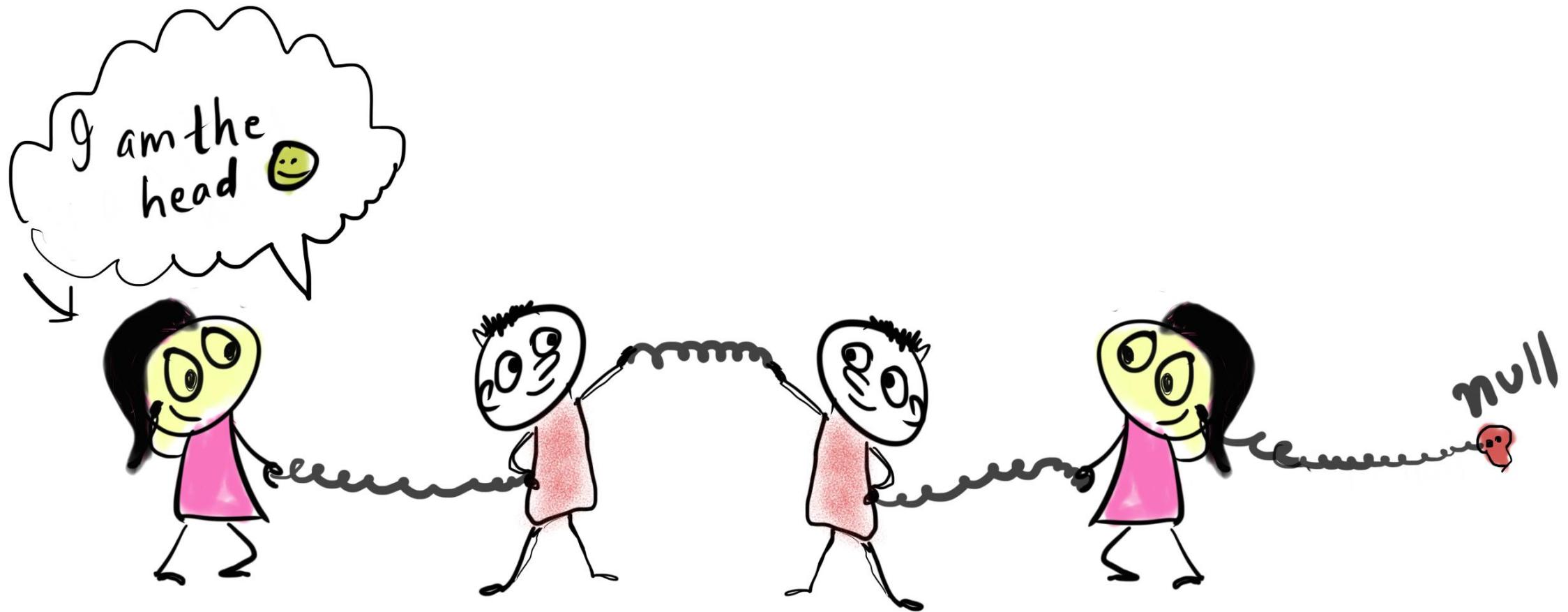


THE END.



Doubly and Circular Linked Lists

SINGLY LINKED LIST.



RECAP TO GO FORWARD.

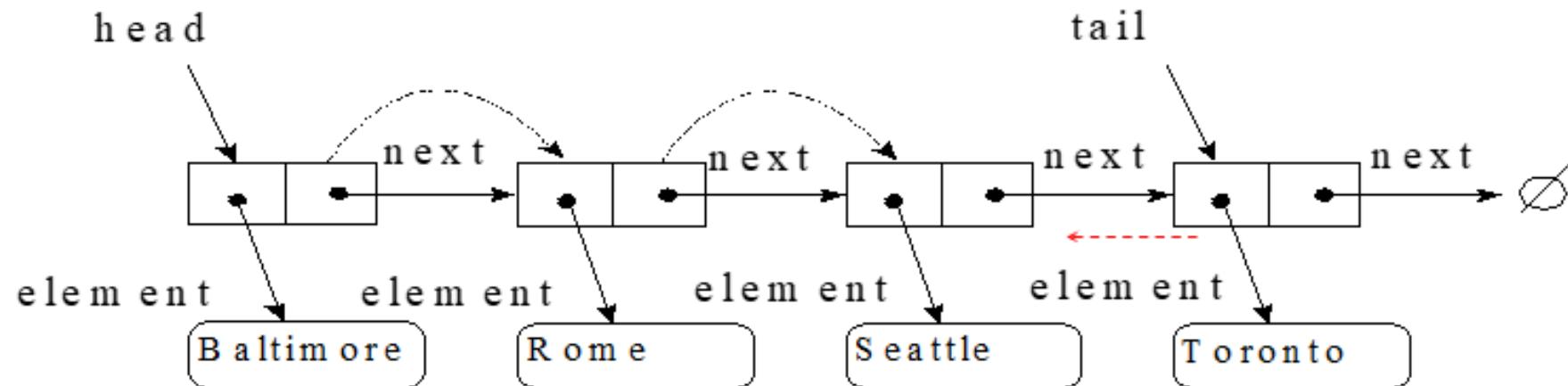
- ▀ The node at the beginning is called the head of the list
- ▀ The entire list is identified by the pointer to the head node, this pointer is called the list head.
- ▀ Nodes can be added or removed from the linked list during execution
- ▀ Addition or removal of nodes can take place at beginning, end, or middle of the list
- ▀ Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size.

TYPES OF LISTS.

- ↳ Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
 - ↳ Singly Linked List
 - ↳ Doubly Linked List
 - ↳ Circular Linked List

DOUBLY LINKED LIST.

- ⌚ Recall, deletion of element in singly linked list required link hopping to find last node(tail node).
- ⌚ Made easier with doubly linked list.

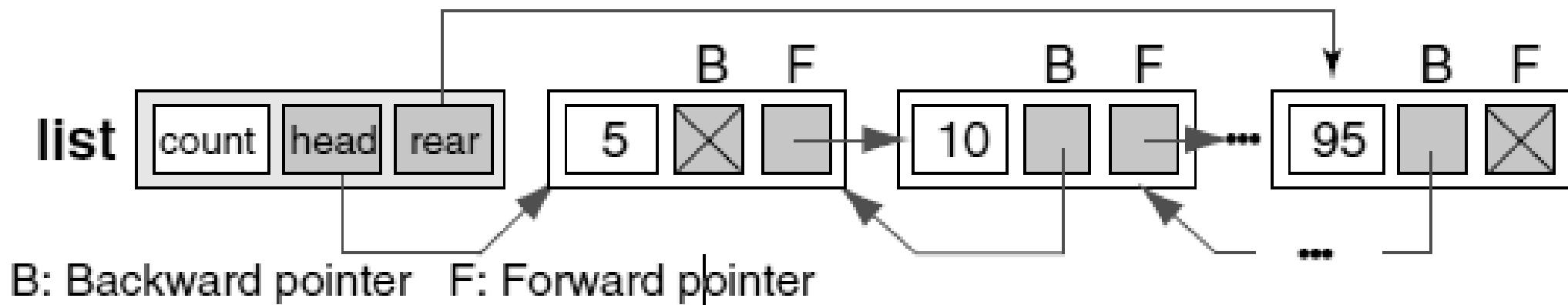


DOUBLY LINKED LIST.

- ⌚ Is a linked list structure in which each node has a pointer to both its *successor* and its *predecessor*.
- ⌚ Pointers exist between adjacent nodes in both directions.
- ⌚ The list can be traversed either forward or backward.
- ⌚ There are three pieces of metadata in the head structure: a count, a position pointer for traversals, and a rear pointer.

DOUBLY LINKED LIST.

- Although a rear pointer is not required in all doubly linked lists, it makes some of the list algorithms, such as insert and search, more efficient.
- Each node contains two pointers: a *backward pointer* to its predecessor and a *forward pointer* to its successor.



DOUBLY LINKED LIST.

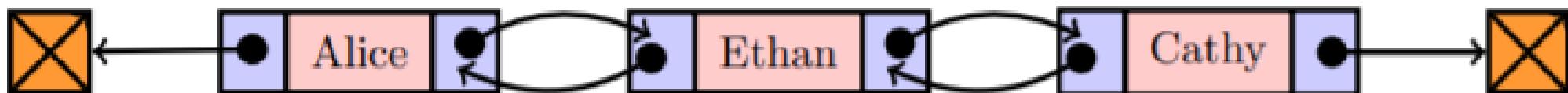
Doubly Linked List



Insert a node

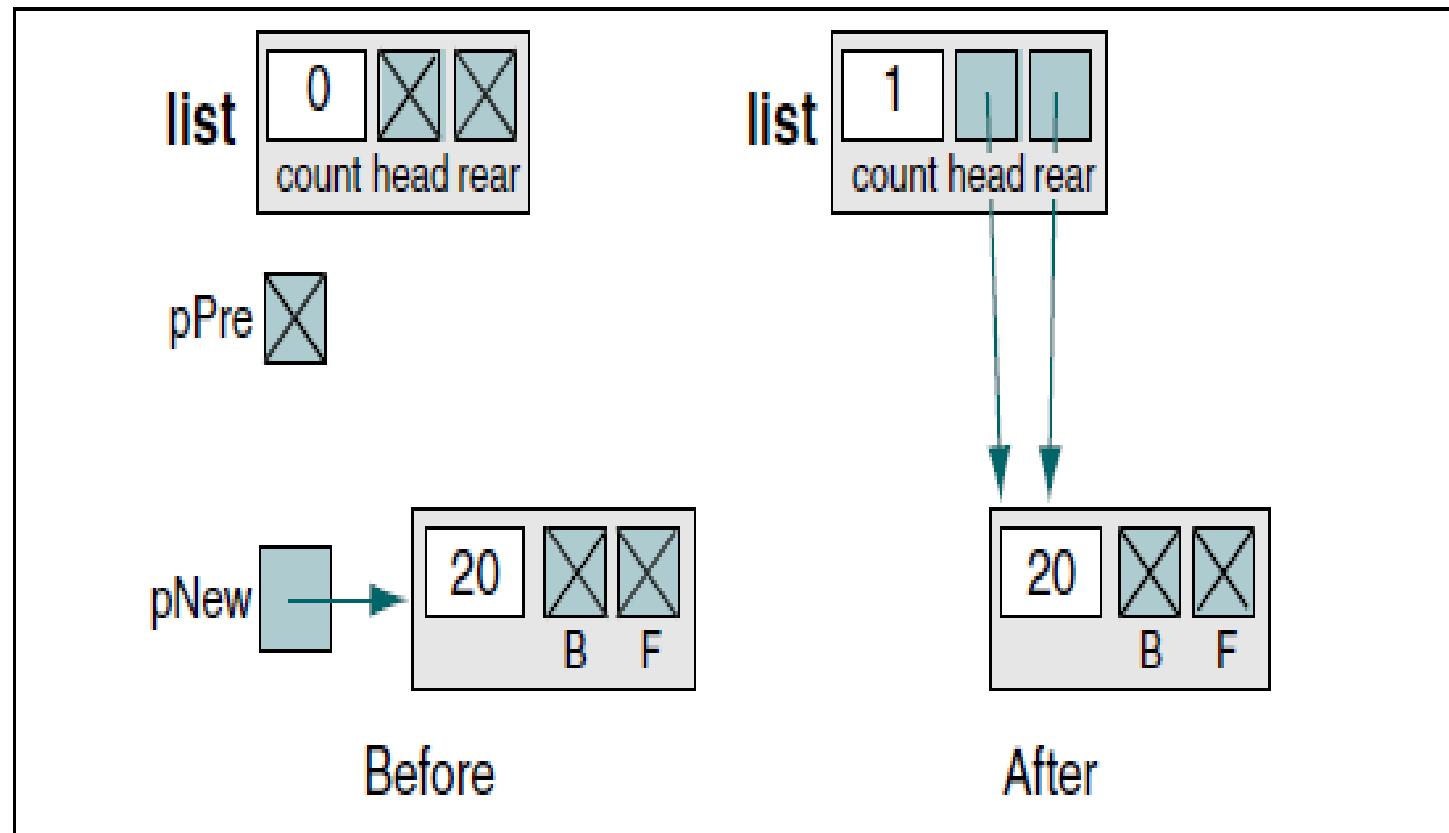


Delete a node



DOUBLY LINKED LIST - INSERTION.

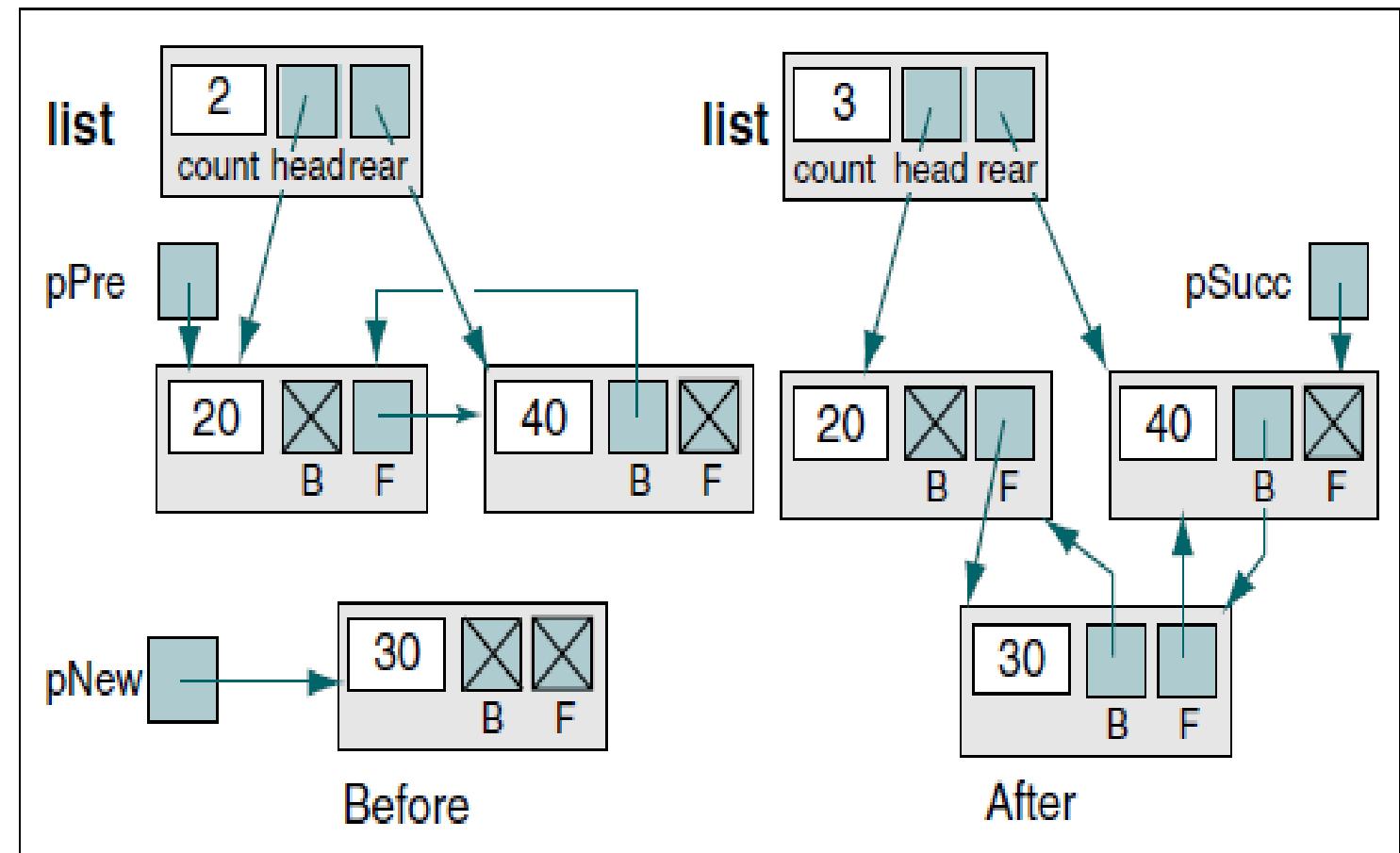
- ▶ Insertion follows the basic pattern of inserting a node into a singly linked list, but we also need to **connect both the forward and the backward pointers**.
- ▶ A null doubly linked list's **head and rear pointers are null**.
- ▶ To insert a node into a null list, we simply **set the head and rear pointers to point to the new node** and set the **forward and backward pointers of the new node to null**.



(a) Insert into null list or before first node

DOUBLY LINKED LIST - INSERTION.

- For the insertion between two nodes:
- The new node needs to be set to point to both its predecessor and its successor, and they need to be set to point to the new node.
- Because the insert is in the middle of the list, the head structure is unchanged.



(b) Insert between two nodes

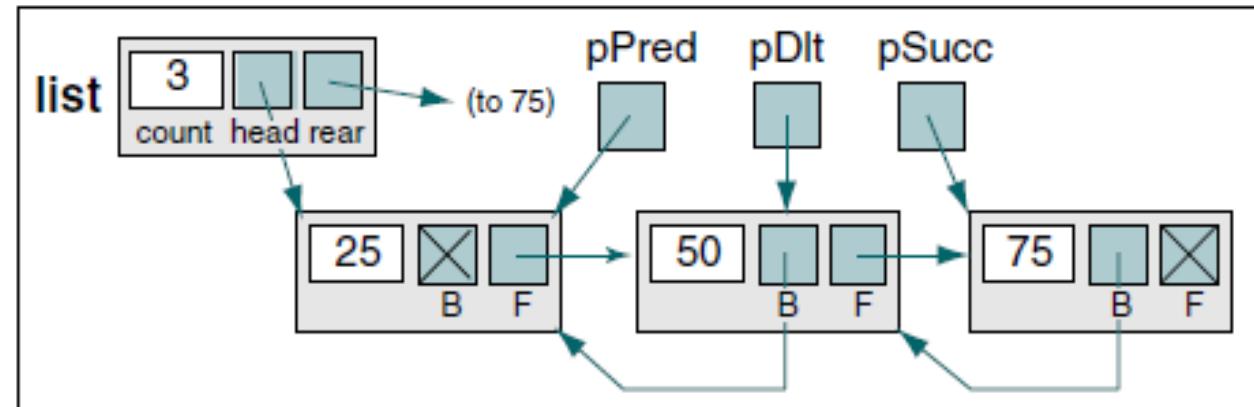
ALGORITHM 5-11 Doubly Linked List Insert

```
Algorithm insertDbl (list, dataIn)
This algorithm inserts data into a doubly linked list.
    Pre    list is metadata structure to a valid list
           dataIn contains the data to be inserted
    Post   The data have been inserted in sequence
    Return 0: failed--dynamic memory overflow
           1: successful
           2: failed--duplicate key presented
1 if (full list)
  1 return 0
2 end if
    Locate insertion point in list.
3 set found to searchList
    (list, predecessor, successor, dataIn key)
4 if (not found)
  1 allocate new node
  2 move dataIn to new node
  3 if (predecessor is null)
      Inserting before first node or into empty list
      1 set new node back pointer to null
      2 set new node fore pointer to list head
      3 set list head to new node
```

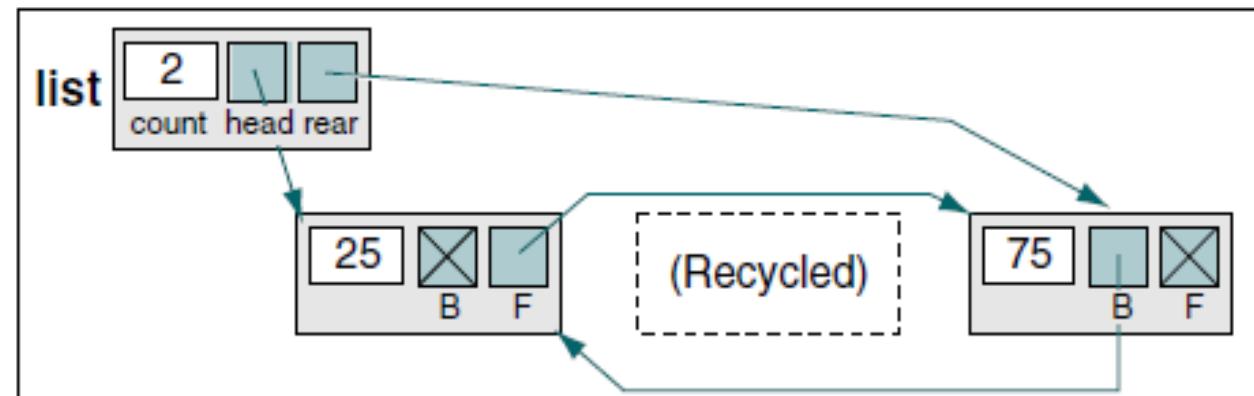
```
4 else
    Inserting into middle or end of list
    1 set new node fore pointer to predecessor fore pointer
    2 set new node back pointer to predecessor
5 end if
    Test for insert into null list or at end of list
6 if (predecessor fore null)
    Inserting at end of list--set rear pointer
    1 set list rear to new node
7 else
    Inserting in middle of list--point successor to new
    1 set successor back to new node
8 end if
9 set predecessor fore to new node
10 return 1
5 end if
    Duplicate data. Key already exists.
6 return 2
end insertDbl
```

DOUBLY LINKED LIST - DELETION.

- Deletion of nodes:
- Deleting requires that the deleted node's predecessor, if present, be pointed to the deleted node's successor and that the successor, if present, be set to point to the predecessor.
- Once we locate the node to be deleted, we simply change its predecessor's and successor's pointers and recycle the node



(a) Before delete



(b) After deleting 50

ALGORITHM 5-12 Doubly Linked List Delete

```
Algorithm deleteDbl (list, deleteNode)
```

This algorithm deletes a node from a doubly linked list.

Pre list is metadata structure to a valid list

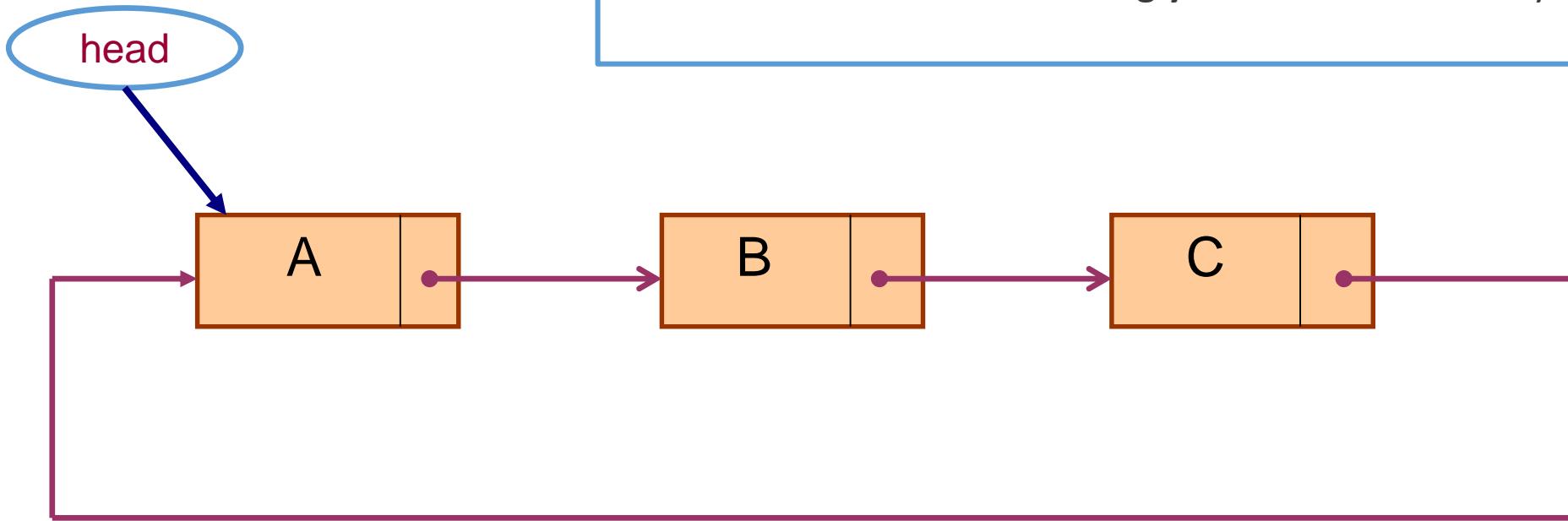
 deleteNode is a pointer to the node to be deleted

Post node deleted

```
1 if (deleteNode null)
  1 abort ("Impossible condition in delete double")
2 end if
3 if (deleteNode back not null)
  Point predecessor to successor
  1 set predecessor to deleteNode back
  2 set predecessor fore to deleteNode fore
4 else
  Update head pointer
  1 set list head to deleteNode fore
5 end if
```

```
6 if (deleteNode fore not null)
    Point successor to predecessor
        1 set successor      to deleteNode fore
        2 set successor back to deleteNode back
7 else
    Point rear to predecessor
        1 set list rear to deleteNode back
8 end if
9 recycle (deleteNode)
end deleteDbl
```

CIRCULAR LINKED LIST.

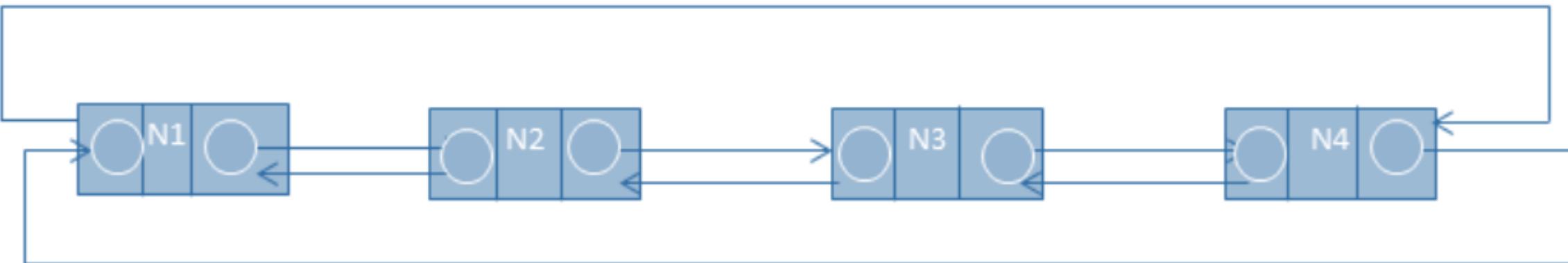


A circular linked list can be a **singly linked list** or a **doubly linked list**.

In ***singly linked list***, pointer from the last element in the list points back to the first element

CIRCULAR LINKED LIST.

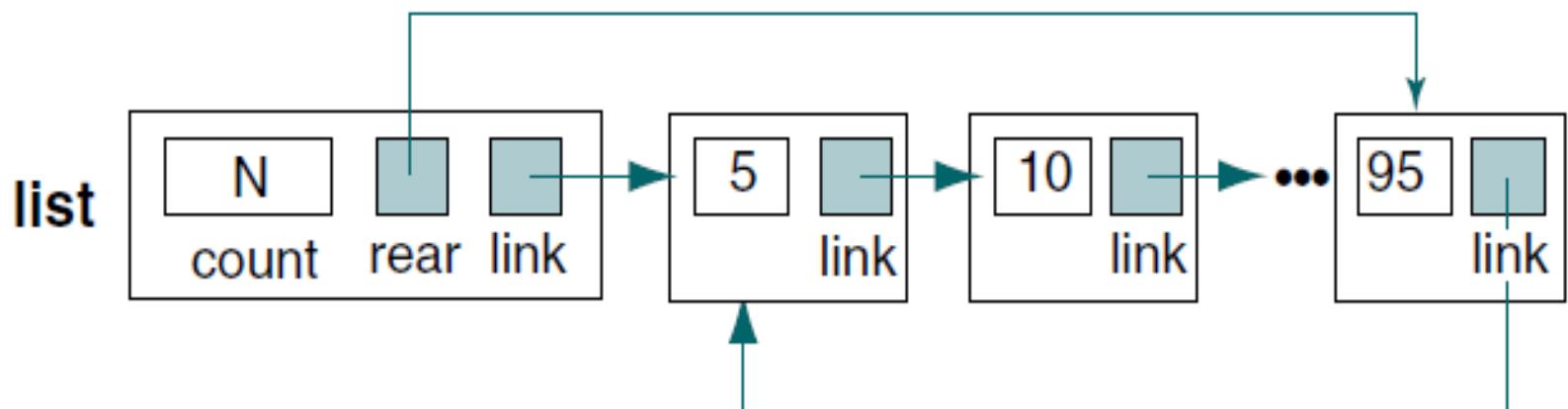
A circular linked list can be a singly linked list or a **doubly linked list**.



In a ***doubly circular linked list***, the previous pointer of the first node is connected to the last node while the next pointer of the last node is connected to the first node.

CIRCULAR LINKED LIST.

- ⌚ Circularly linked lists are primarily used in lists that allow access to nodes in the middle of the list without starting at the beginning.
- ⌚ Insertion into and deletion from a circularly linked list follow the same logic patterns used in a singly linked list except that the last node points to the first node. So, when inserting or deleting the last node, in addition to updating the rear pointer in the header, the link field must point to the first node.



Advantages of Circular Linked Lists:

1. **Any node can be a starting point.** We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of queue. Unlike usual implementation, **we don't need to maintain two pointers for front and rear if we use circular linked list.** We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
3. **Circular lists are useful in applications to repeatedly go around the list.** For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the **operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.**

The end.

A photograph of a large, mature tree with a wide, spreading canopy of dark green leaves. The tree is set against a bright blue sky with scattered white clouds. The perspective is from a low angle, looking up at the branches.

TREES.
A *non-linear data*
structure.



OBJECTIVES

- Upon completion you will be able to:
 - Understand and use basic tree terminology and concepts
 - Recognize and define the basic attributes of a binary tree
 - Process trees using depth-first and breadth-first traversals
 - Parse expressions using a binary tree
 - Design and implement Huffman trees
 - Understand the basic use and processing of general trees



Applications of Trees.

- File system
- Dynamic Spell Checking
- Network routing algorithm
- Family Tree
- Organization hierarchy

Basic Terminology.



A photograph of a large, mature tree with a dense canopy of green leaves, set against a bright blue sky with scattered white clouds. The tree is positioned centrally in the frame, with its trunk and branches visible at the top.

LOGICAL REPRESENTATION OF TREES.

TREES.

- **A non-linear list:** Each element can have ***more than one successor*** element
- Types of non-linear data structures:
 - 1. **Tree:** An element can have only one predecessor
 - ☞ ***Two-way or binary*** (up to two successors)
 - ☞ ***Multi-way trees*** (no limitation successors)
 - 2. **Graph:** An element can have one or more predecessors

Except root

Trees

- Trees are natural structures for representing certain kinds of **hierarchical data**. (How our files get saved under hierarchical directories)
- Tree is a data structure which allows you to associate a **parent-child relationship** between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion.
- Trees have many uses in **computing**. For example, a *parse-tree* can represent the structure of an expression.
- **Binary Search Trees** help to order the elements in such a way that the searching takes less time as compared to other data structures. (speed advantage over other D.S)

Basic Tree concepts.

- A **tree** consists of:
 - » Finite set of elements, called *nodes*, and
 - » Finite set of directed lines called *branches*, that connect the nodes.
 - » The number of branches associated with a node is the **degree** of the node.

Basic Tree concepts.

- When the branch is directed toward the node, it is *indegree* branch.
- When the branch is directed away from the node, it is an *outdegree* branch.
- The *sum of the indegree and outdegree* branches is the *degree* of the node.
- If the tree is not empty, the first node is called *the root*.



Basic Tree concepts.

- The *indegree of the root is, by definition, zero.*
- With the exception of the root, **all of the nodes** in a tree must have an **indegree** of exactly **one**; that is, they may have only one predecessor.
- All nodes in the tree can have zero, one, or more branches leaving them; that is, they may have outdegree of zero, one, or more.

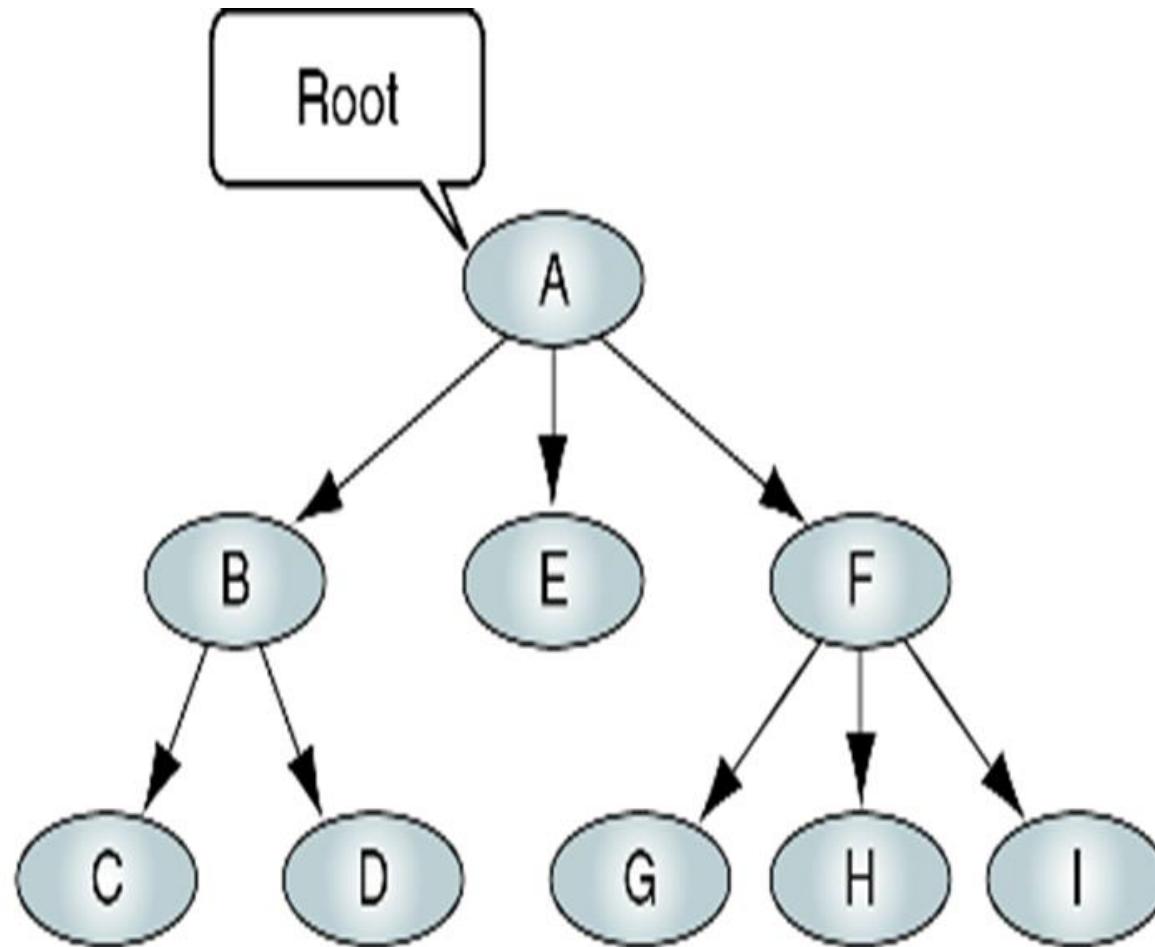
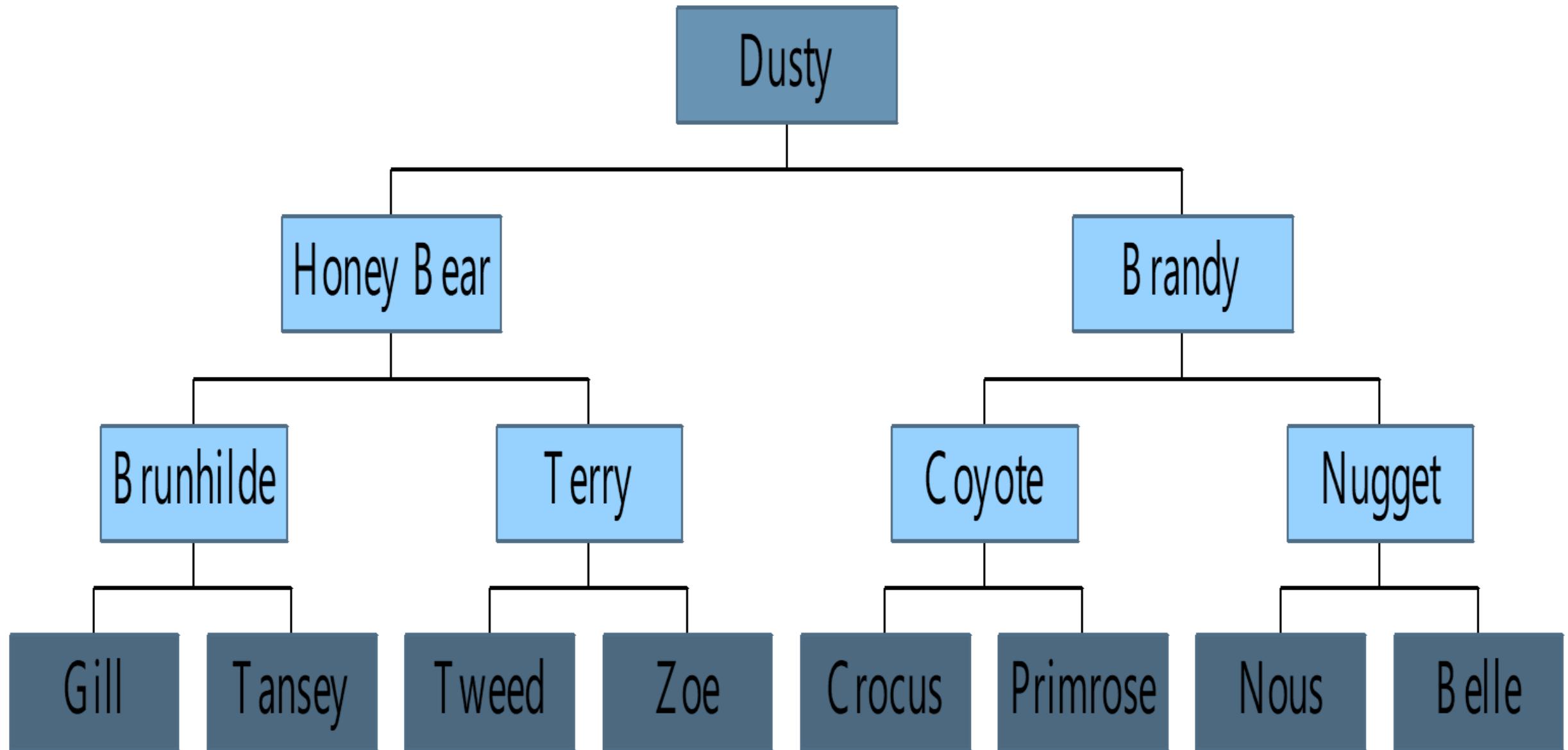


FIGURE 6-1 Tree

Tree example.



Basic Tree Concepts.

- A **leaf** is any node with an **outdegree of zero**, that is, a node with no successors.
- A node that is not a root or a leaf is known as an **internal node**.
- A node is a **parent** if it has successor nodes; that is, if it has **outdegree greater than zero**.
- A node with a predecessor is called a child.

Basic Tree Concepts.

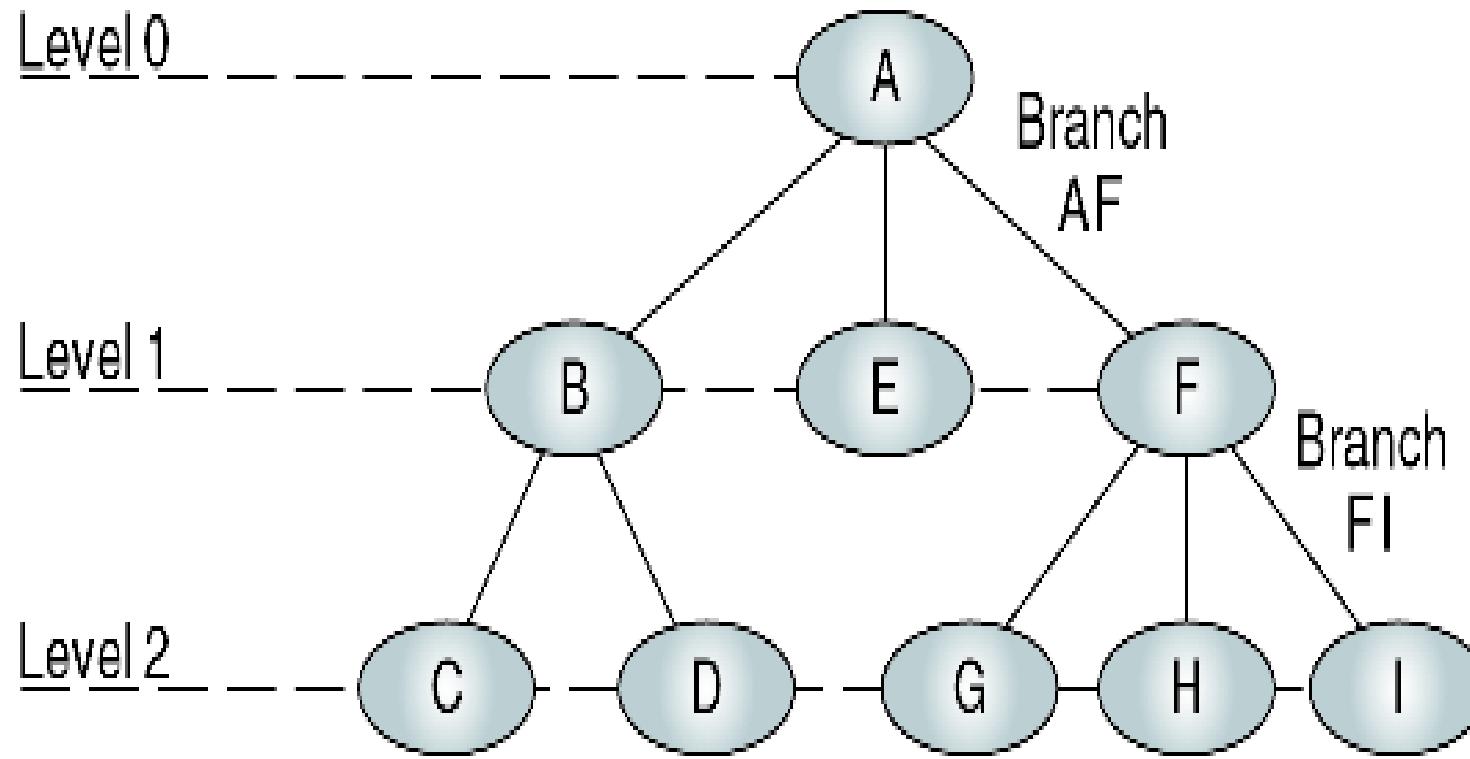
- Two or more nodes with the **same parents** are called *siblings*.
- An **ancestor** is any node in the **path from the root to the node**.
- A **descendant** is any node in the path below the parent node; that is, all nodes in the paths from a given node to a leaf are descendants of that node.

Basic Tree Concepts.

- A **path** is a sequence of nodes in which each node is adjacent to the next node.
- The level of a node is its distance from the root. The root is at level 0, its children are at level 1, etc. ...

Basic Tree Concepts.

- The **height of the tree** is the level of the leaf in the longest path from the root plus 1.
- By definition the **height of any empty tree is -1.**
- A **subtree** is any connected structure below the root.
- The **first node in the subtree** is known as the root of the subtree.

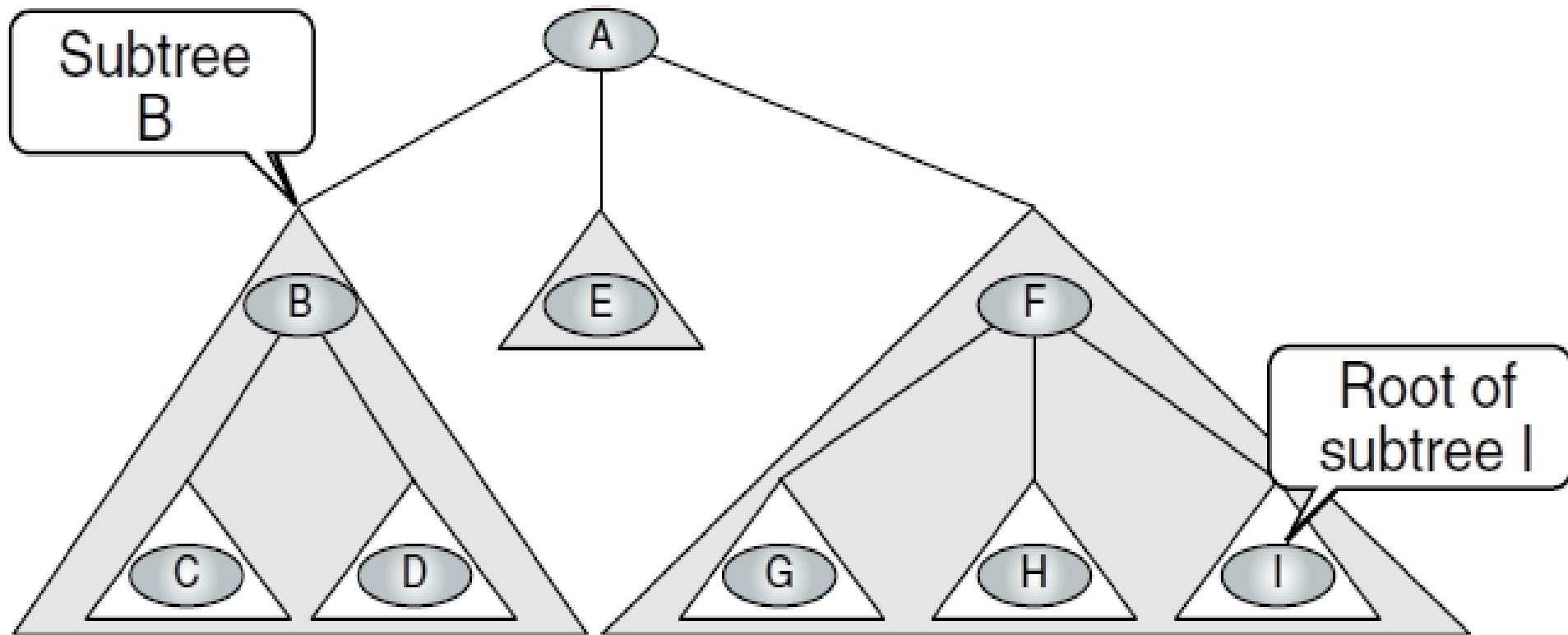


Root: A
Parents: A, B, F
Children: B, E, F, C, D, G, H, I

Siblings: {B,E,F}, {C,D}, {G,H,I}
Leaves: C,D,E,G,H,I
Internal nodes: B,F

FIGURE 6-2 Tree Nomenclature

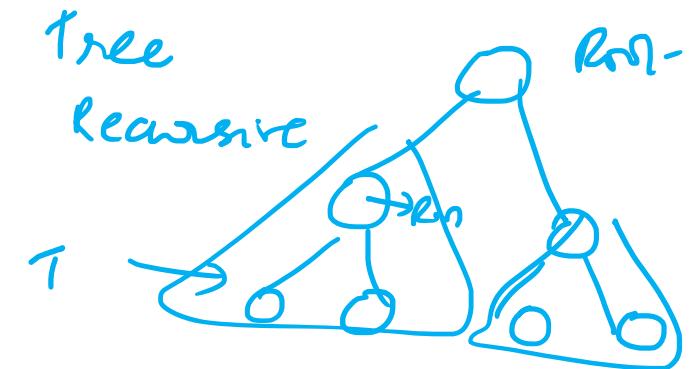
SUB TREES.



Subtrees

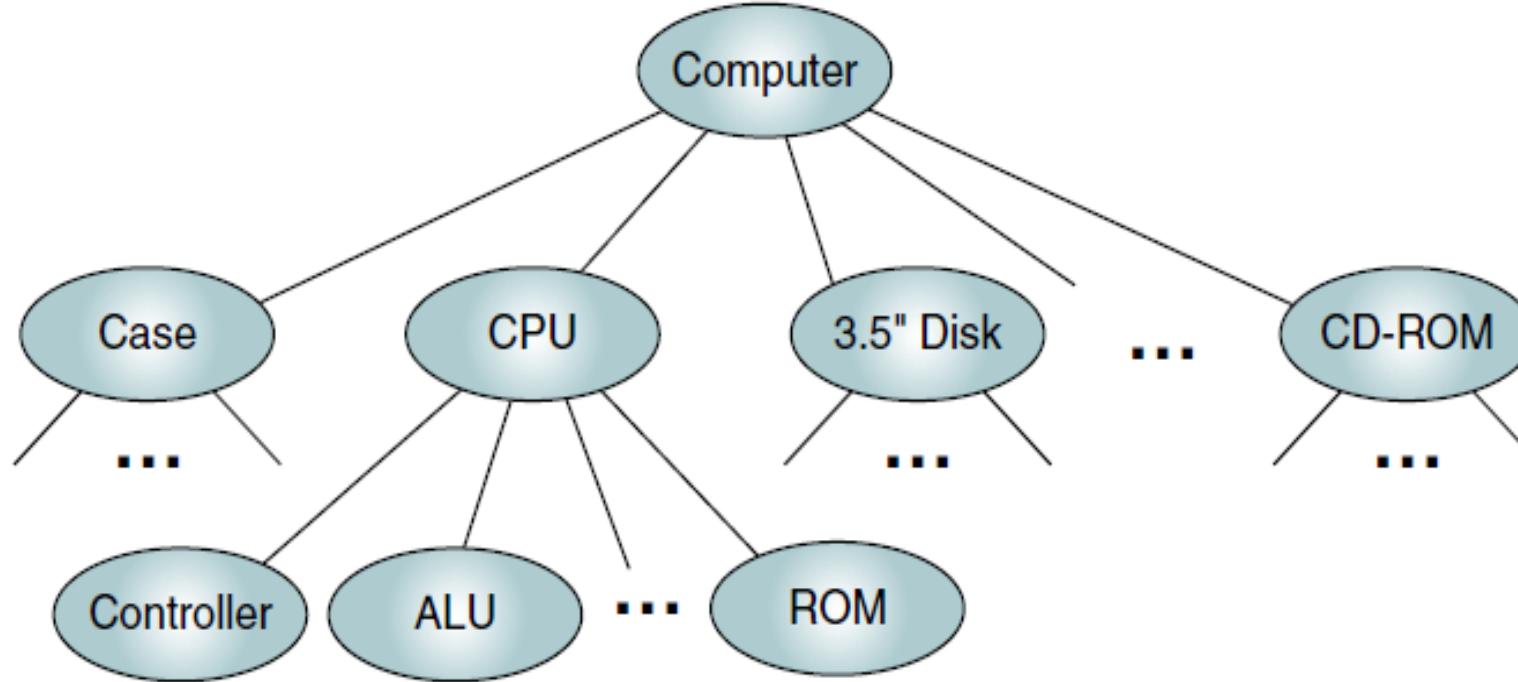
Recursive definition of a tree.

- A **tree** is a set of nodes that either:
 - is empty or
 - has a designated node, called the **root**, from which hierarchically descend **zero or more subtrees**, which are also trees.



USER REPRESENTATIONS FOR TREES: ORGANIZATION CHART FORMAT

General
Tree
format



Computer Parts List as a General Tree

This is basically the notation we use to represent trees in our figures. The term we use for this notation is **General tree**.

USER REPRESENTATIONS FOR TREES: INDENTED LIST FORMAT

2

Part number	Description
301	Computer
301-1	Case
...	...
301-2	CPU
301-2-1	Controller
301-2-2	ALU
...	...
301-2-9	ROM
301-3	3.5" Disk
...	...
301-9	CD-ROM
...	...

| Computer Bill of Materials

③

USER REPRESENTATIONS FOR TREES:

PARENTHETICAL LISTING

```
A (B (C D) E F (G H I))
```

- ↳ *Used with algebraic expressions.*
- ↳ When a tree is represented in parenthetical notation:
 - ↳ Each open parenthesis indicates the start of a new level;
 - ↳ Each closing parenthesis completes the current level and moves up one level in the tree.



The end of part 1 – Basic concept of Trees.

Binary Trees.

- A binary tree can have *no more than two descendants*. In this section we discuss the properties of binary trees, four different binary tree traversals
- What we will look at ahead:
 - » Properties
 - » Binary Tree Traversals
 - » Examples:
 - » Expression Trees
 - » Huffman Code

Binary Trees.

- A **binary tree** is a tree in which no node can have more than two subtrees; the maximum outdegree for a node is two.
- In other words, a node can have zero, one, or two subtrees.
- These subtrees are designated as the **left subtree** and the **right subtree**.

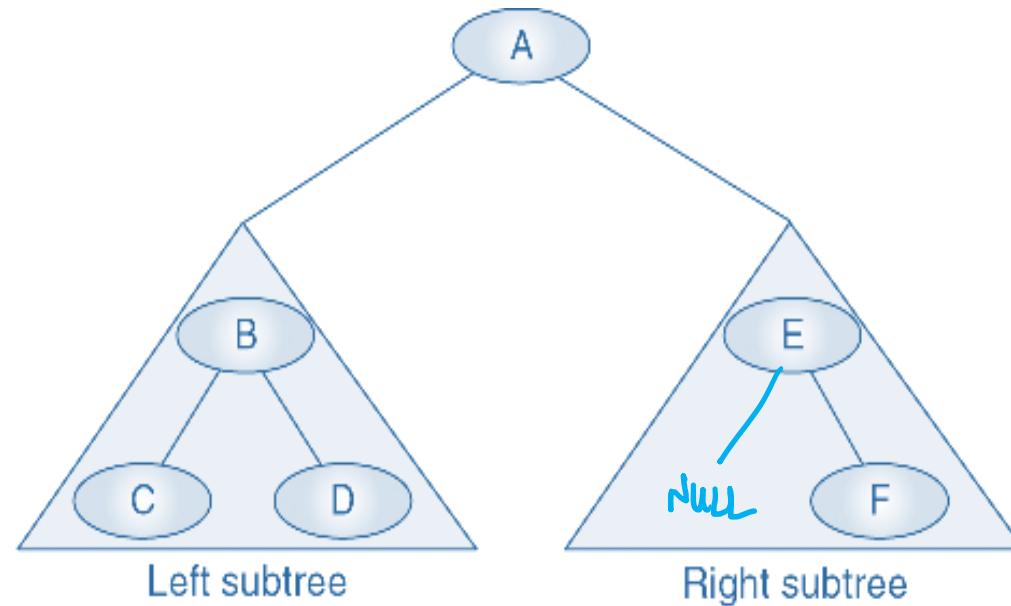
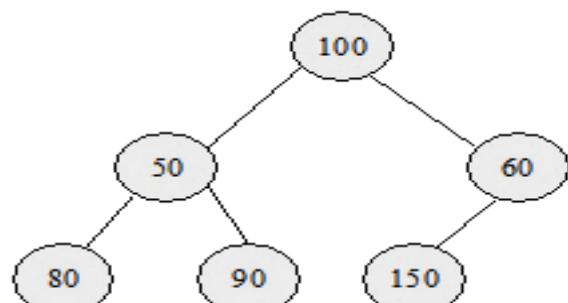


FIGURE 6-5 Binary Tree

Binary Trees

■ Binary Tree

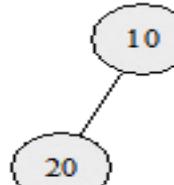
- A tree in which outdegree of each node is less than or equal to 2 (each node has 0, 1 or 2 children)



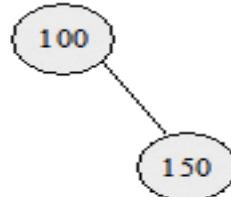
(a)



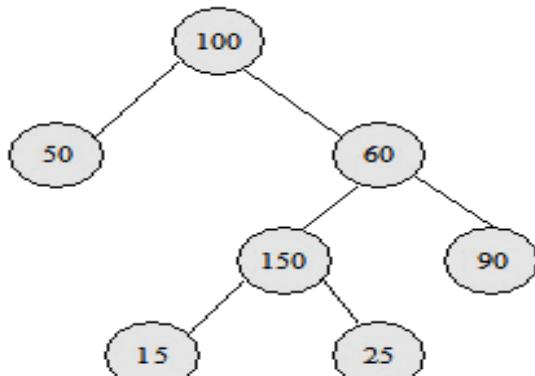
(b)



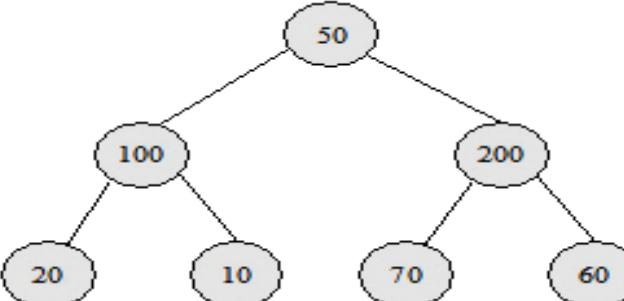
(c)



(d)



(e)



(f)

A **null tree** is a tree with no nodes

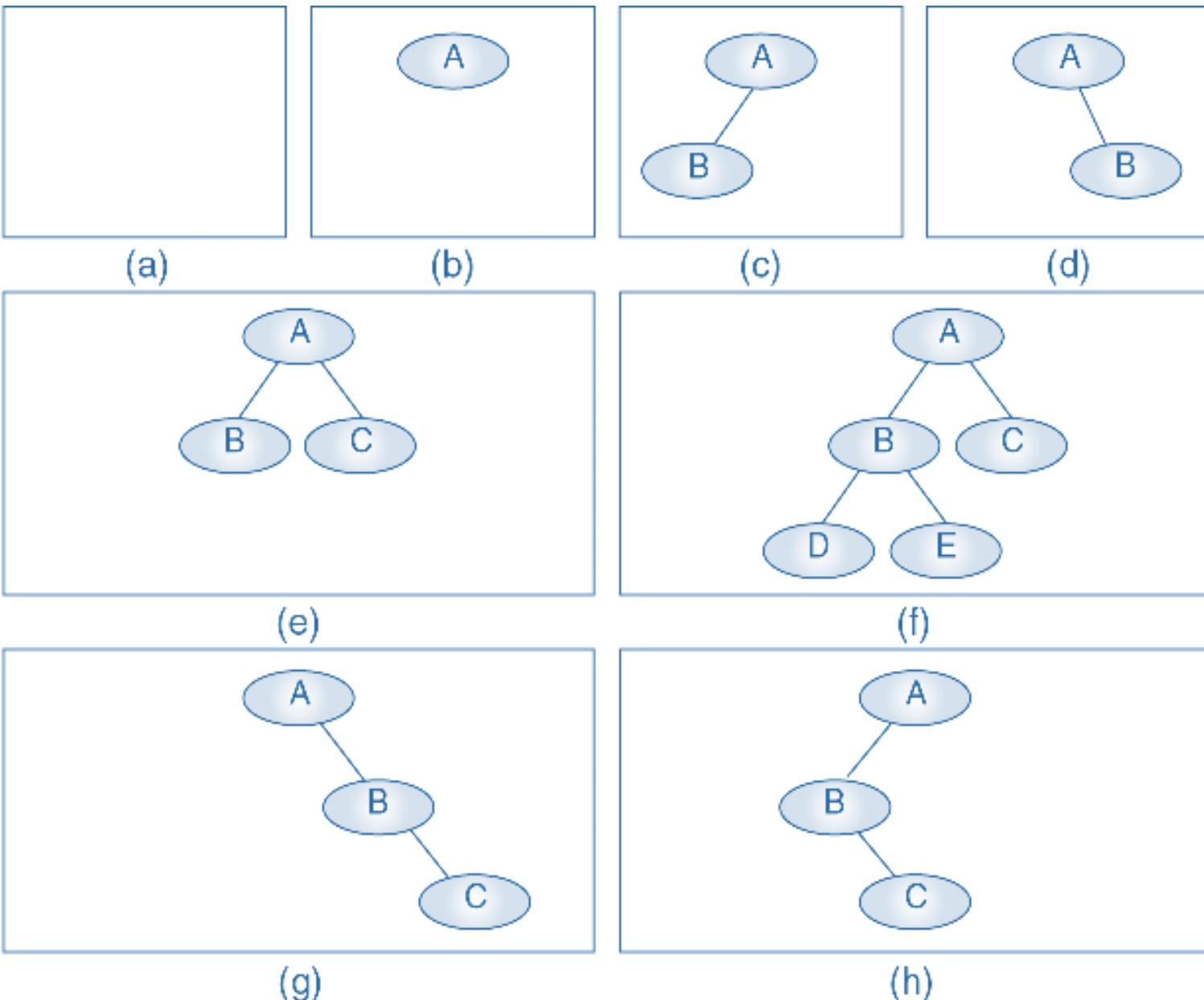
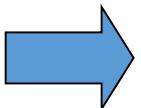


FIGURE 6-6 Collection of Binary Trees

Some Properties of Binary Trees.

- The height of binary trees can be mathematically predicted
- Given that we need to store N nodes in a binary tree, the maximum height is

$$H_{\max} = N$$

A tree with a maximum height is rare. It occurs when all of the nodes in the entire tree have only one successor.

Some Properties of Binary Trees.

- The **minimum height** of a binary tree is determined as follows:

$$H_{\min} = \lceil \log_2 N \rceil + 1$$

For instance, if there are three nodes to be stored in the binary tree ($N=3$)
then $H_{\min}=2$.

Some Properties of Binary Trees.

- Given a height of the binary tree, H , the **minimum number of nodes** in the tree is given as follows:

$$N_{\min} = H$$

Some Properties of Binary Trees.

- The formula for the maximum number of nodes is derived from the fact that each node can have only two descendants.
- Given a height of the binary tree, H , the **maximum number of nodes** in the tree is given as follows:

$$N_{\max} = 2^H - 1$$

Some Properties of Binary Trees.

- The children of any node in a tree can be accessed by following only one branch path, the one that leads to the desired node.
- The nodes at level 1, which are children of the root, can be accessed by following only one branch; the nodes of level 2 of a tree can be accessed by following only two branches from the root, etc.
- The **balance factor** of a binary tree is the difference in height between its left and right subtrees:

$$B = H_L - H_R$$

Balance of the tree.

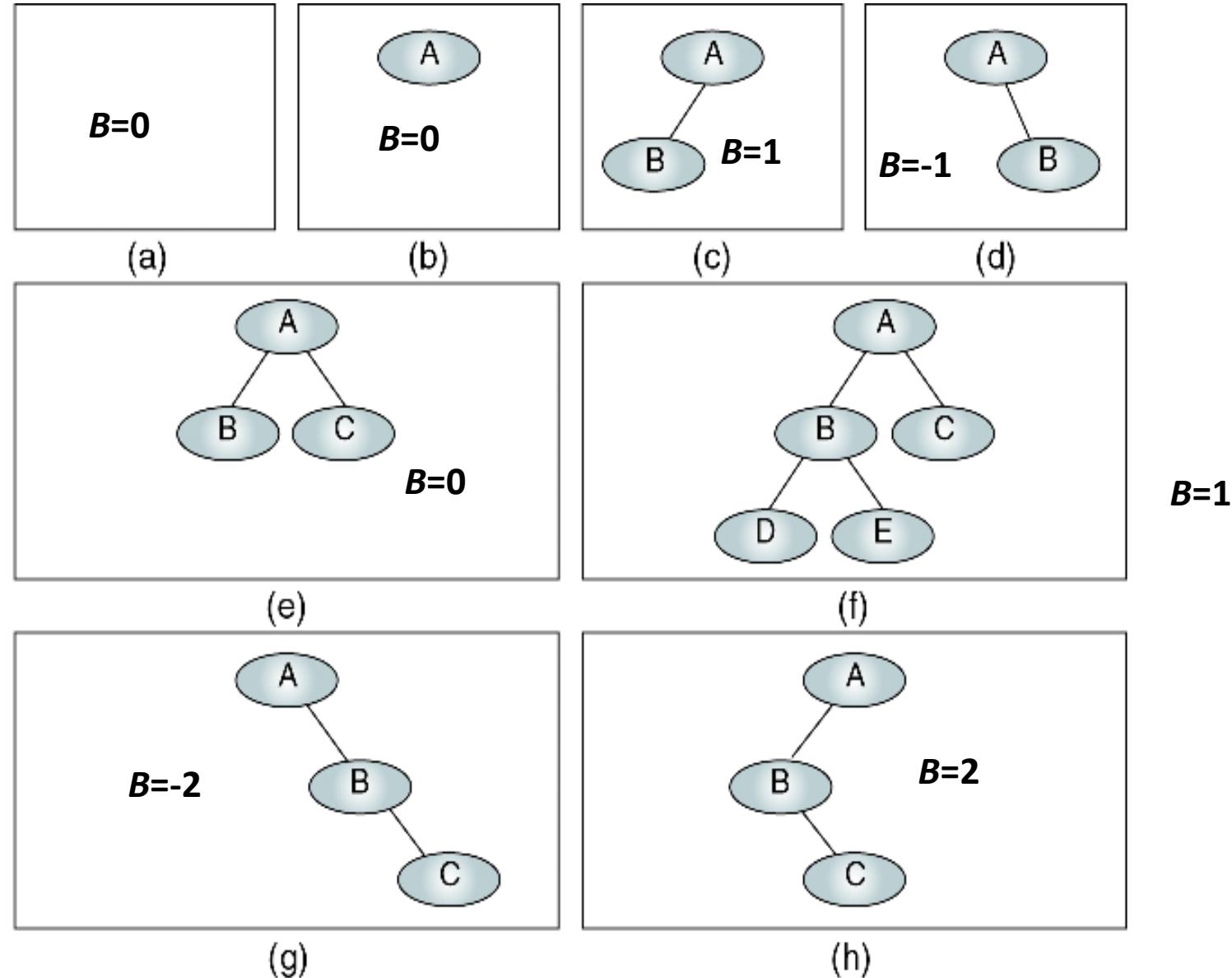


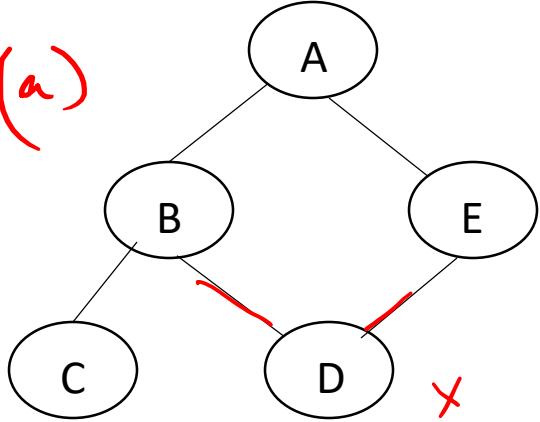
FIGURE 6-6 Collection of Binary Trees

Some Properties of Binary Trees.

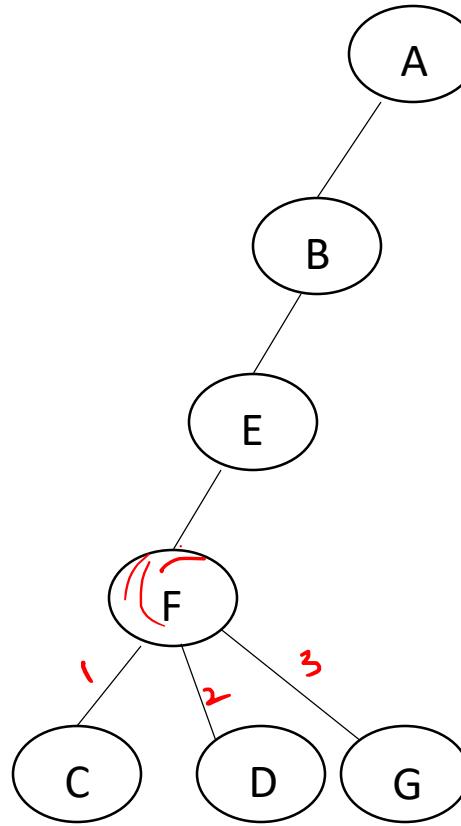
- In the ~~balanced binary tree~~ (definition of Russian mathematicians ^{AVL} Adelson-Velskii and Landis) the height of its subtrees differs by no more than one (its balance factor is -1, 0, or 1), and its subtrees are also balanced.

Examples of non binary trees:

fig (a)



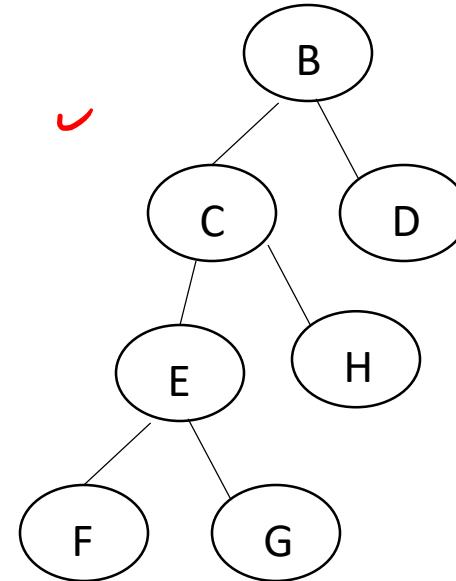
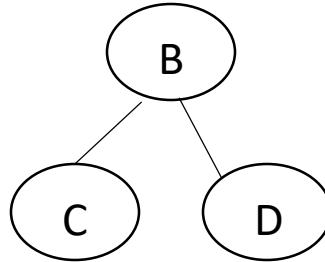
D has indegree 2, hence
not directed tree



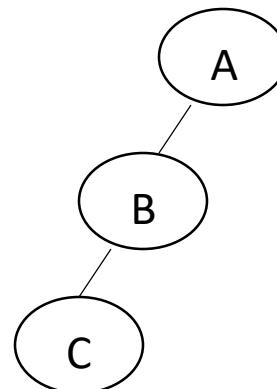
F has outdegree 3.

Strictly binary tree:

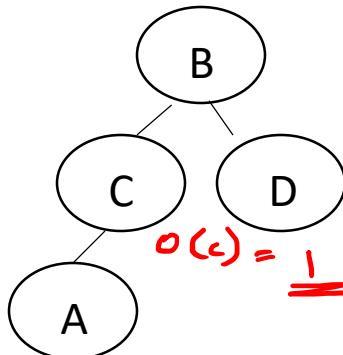
- If the out degree of every node in a tree is either 0 or 2(1 not allowed), then the tree is strictly binary tree.



Tress which are binary trees but not strictly binary:



$$o(A) = 1$$

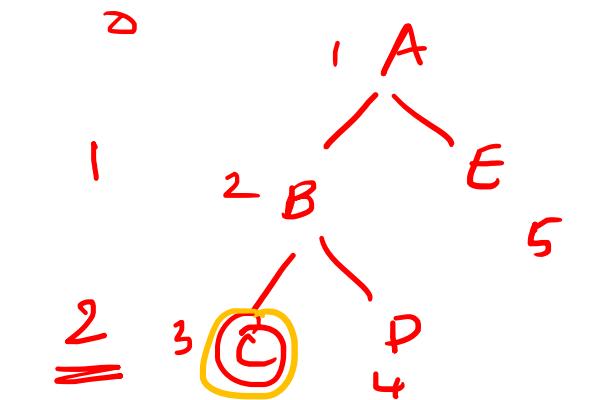


$$o(c) = \underline{\underline{1}}$$

- If every non leaf node in a BT has non empty left and right subtrees ,the tree is called strictly binary tree.

Properties

- If a SBT has n leaves then it contains $2n-1$ nodes.
- Depth:** it is the maximum level of any leaf in the tree.



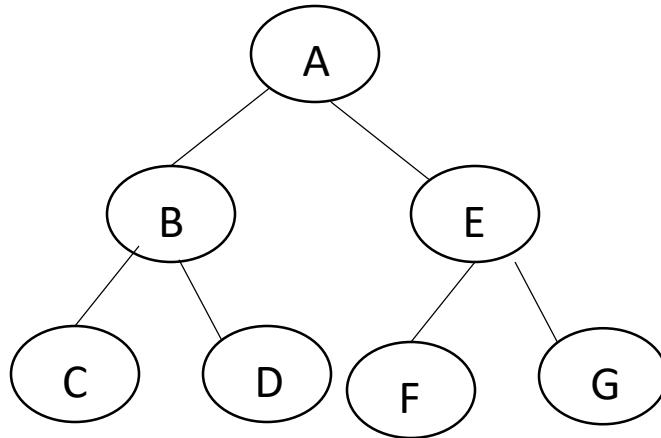
$$n = 3$$

$$(2n - 1)$$

$$2 \times 3 - 1 = \underline{\underline{5}}$$

Complete binary tree:

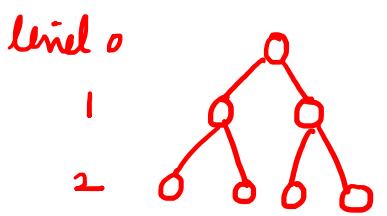
- Is a strictly binary tree in which the number of nodes at any level ‘i’ is $\text{pow}(2,i)$.



Number of nodes at level 0(root level) is $\text{pow}(2,0) \rightarrow 1$

Number of nodes at level 1(B and E) is $\text{pow}(2,1) \rightarrow 2$

Number of nodes at level 2(C,D,F and G) is $\text{pow}(2,2) \rightarrow 4$



- A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d .
- The total number of nodes at each level between 0 and d equals the sum of nodes at each level which is equal to $2^{d+1} - 1$
- No of non leaf nodes in that tree = $2^d - 1$
- No of leaf nodes = 2^d

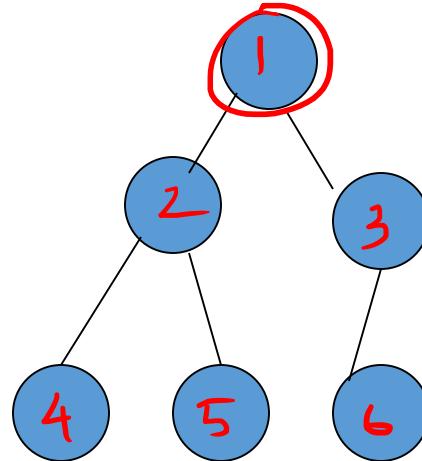
$$\begin{aligned}\text{Total no. of nodes} &= 2^3 - 1 \\ &= \underline{\underline{7}} \text{ nodes}\end{aligned}$$

$$\text{No. of non-leaf nodes} = 2^2 - 1 = \underline{\underline{3}}$$

$$\text{No. of leaf nodes} = 2^2 = \underline{\underline{4}}$$

Almost Complete BT

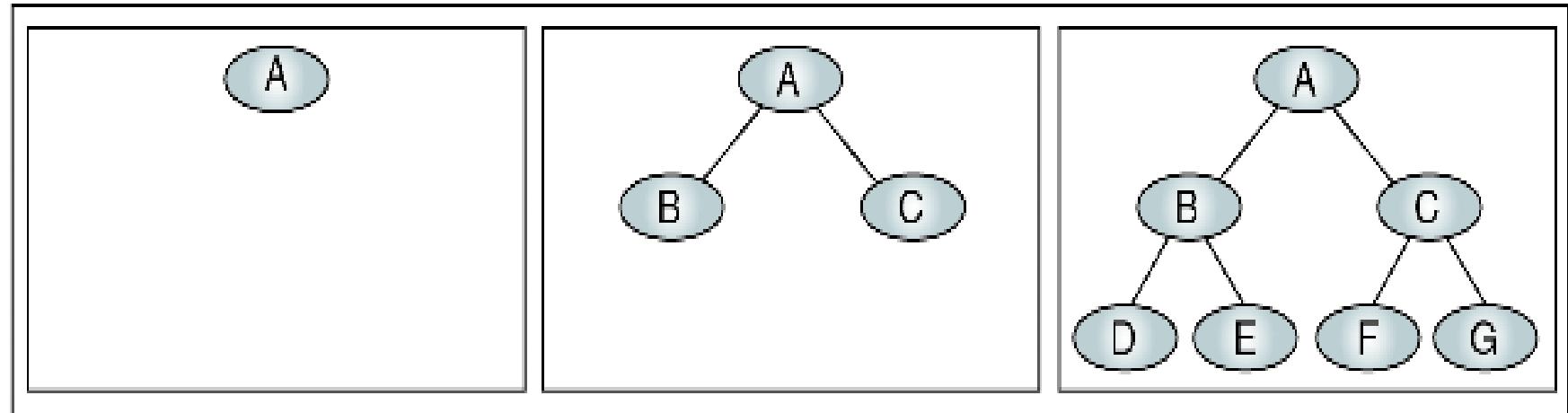
- All levels are complete except the lowest
- In the last level empty spaces are towards the right.



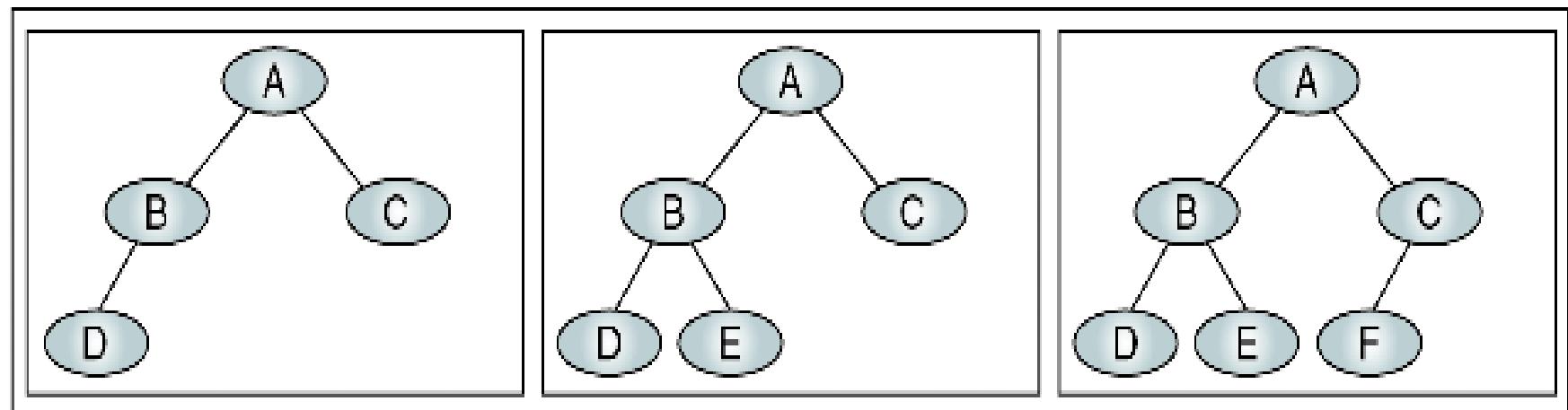
Complete and nearly complete binary trees

- A **complete tree** has the maximum number of entries for its height.
The maximum number is reached when the last level is full.
- A tree is considered **nearly complete** if it has the minimum height for its nodes and all nodes in the last level are found on the left $\overline{=}$

$$h_{\min} = \lceil \log_2 N \rceil + 1$$



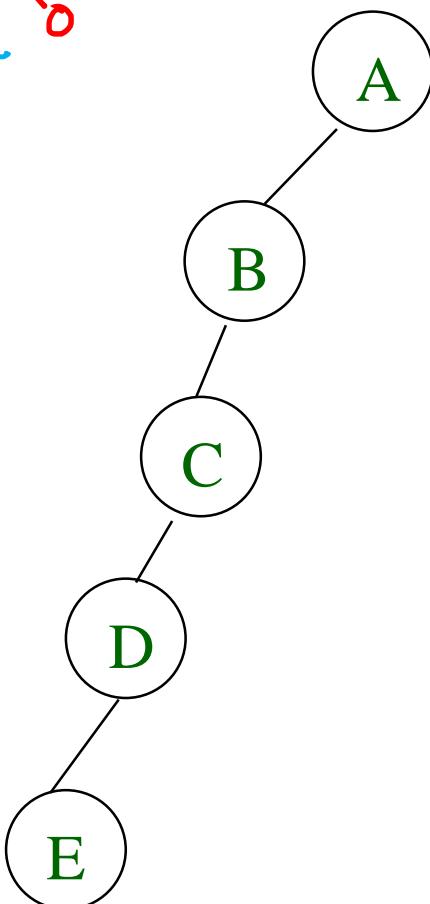
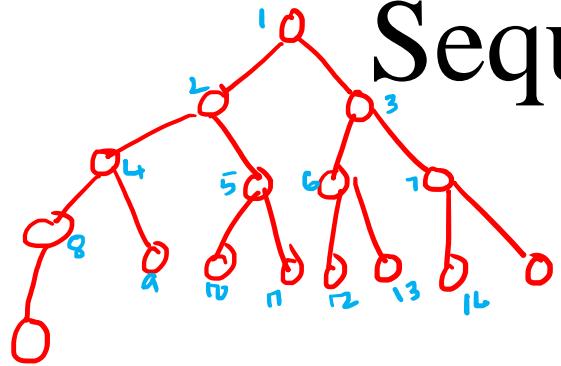
(a) Complete trees (at levels 0, 1, and 2)



(b) Nearly complete trees (at level 2)

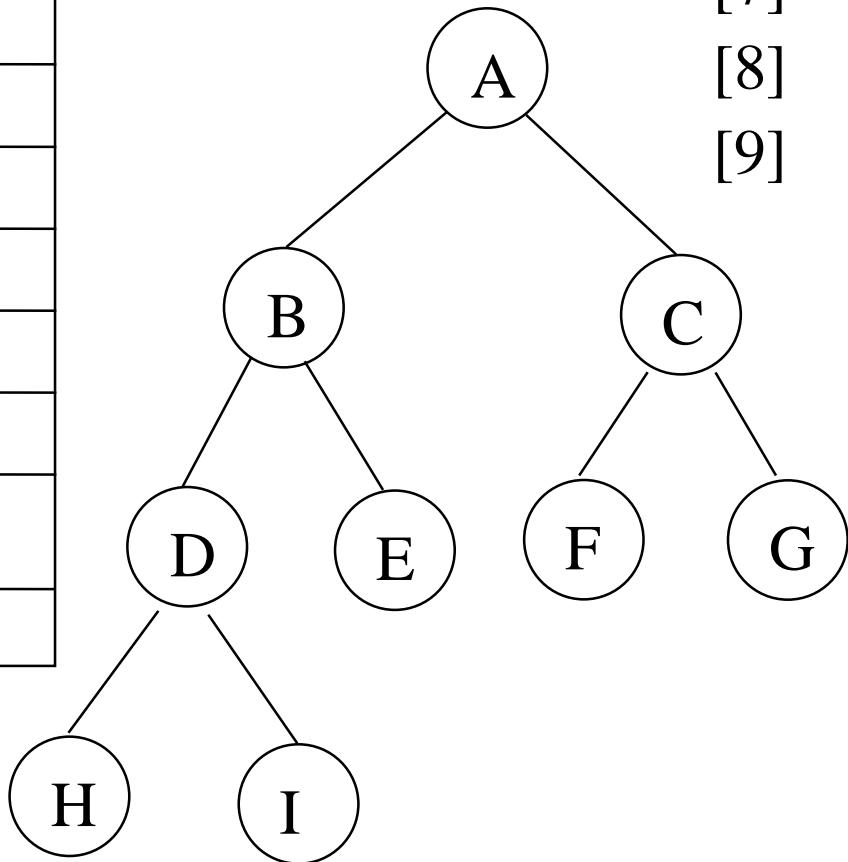
FIGURE 6-7 Complete and Nearly Complete Trees

Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

- (1) waste space
(2) insertion/deletion problem

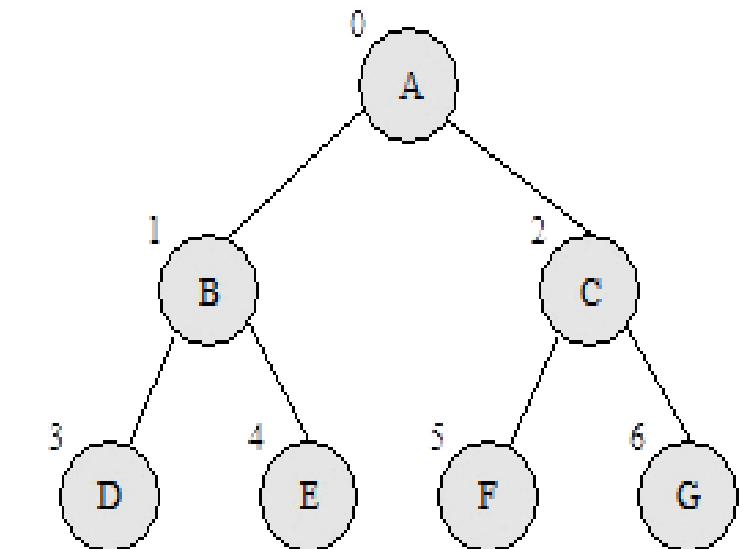


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Binary Trees

■ Storage Representation

- Sequential (array) representation

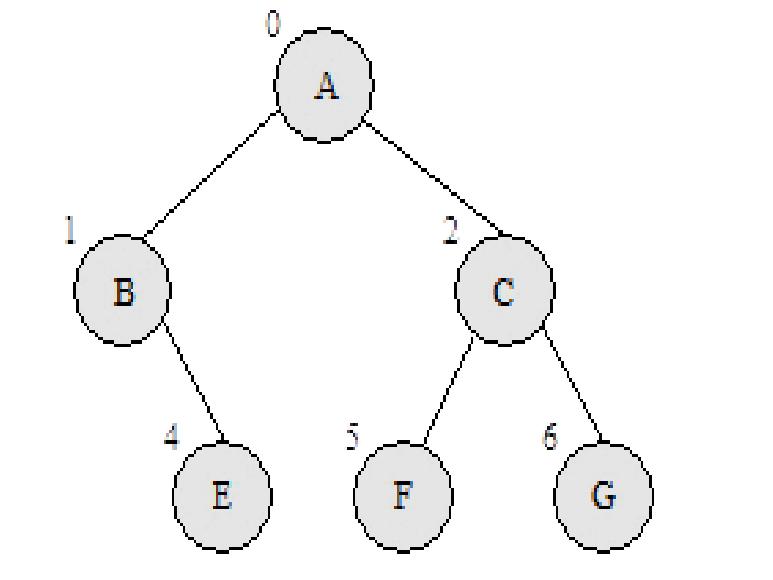


A	B	C	D	E	F	G
---	---	---	---	---	---	---

Left child $2k+1$

Right child $2k+2$

Parent $\lfloor (k-1)/2 \rfloor$



A	B	C		E	F	G
---	---	---	--	---	---	---

Binary Trees

■ Storage Representation (Continued...)

- **Linked** representation

- ▶ Node has three fields

- info
 - leftChildPtr
 - rightChildPtr

- ▶ Implement using structure

```
struct NODE
```

```
{
```

```
    int info;
```

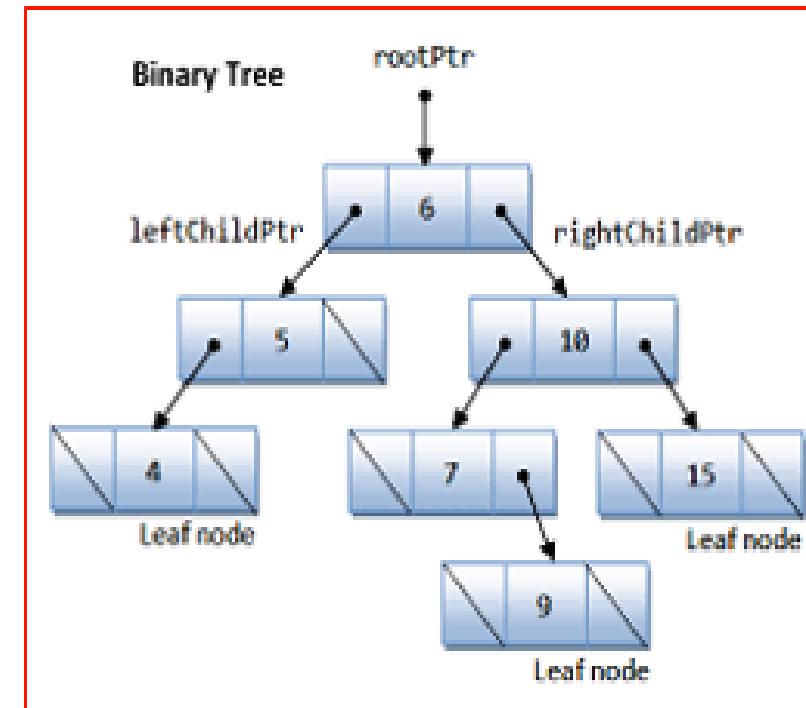
```
    struct NODE *leftChildPtr;
```

```
    struct NODE *rightChildPtr;
```

```
} ;
```

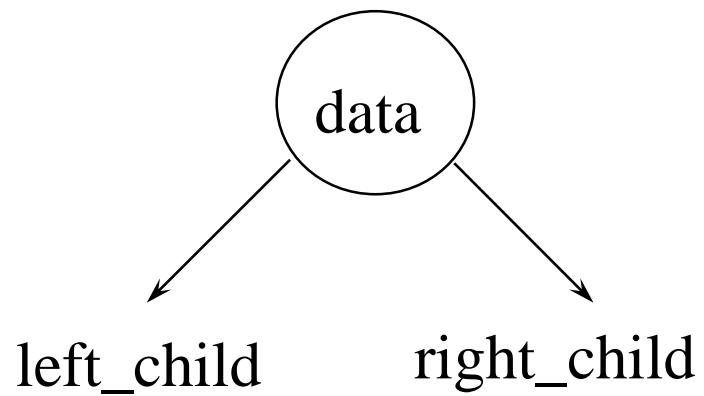
```
struct NODE node;
```

```
struct node *rootPtr = NULL;
```



Linked Representation

```
class node {  
    int data;  
    node *left_child, *right_child;  
};
```



Storage representation of binary trees:

- Trees can be represented using **sequential allocation techniques**(arrays) or by **dynamically allocating memory**.
- In 2nd technique, node has 3 fields
 1. Info : which contains actual information.
 2. llink :contains address of left subtree. //lchild
 3. rlink :contains address of right subtree. //rchild

```
struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};

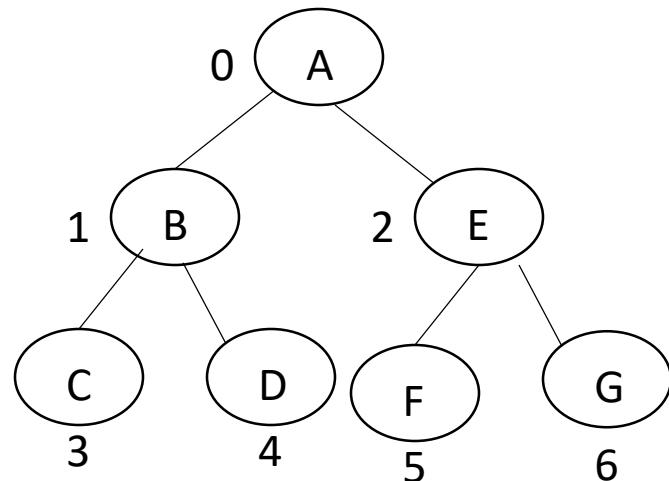
typedef struct node *NODEPTR;
```

Implementing a binary tree:

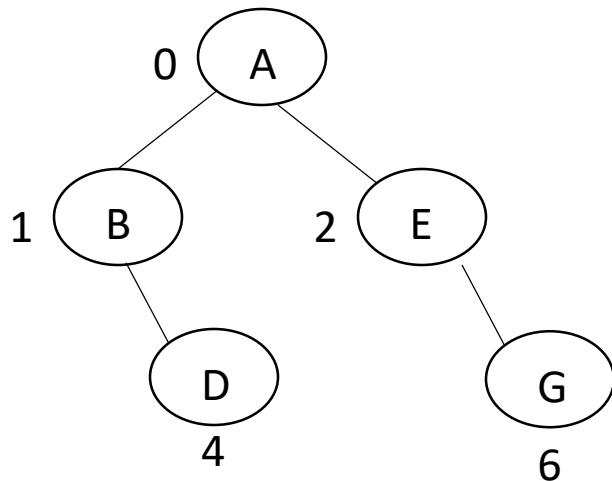
- A pointer variable root is used to point to the root node.
- Initially root is NULL, which means the tree is empty.

NODEPTR root=NULL;

Array representation of binary tree:



0	1	2	3	4	5	6
A	B	E	C	D	F	G



0	1	2	3	4	5	6
A	B	E		D		G

- Given the position ‘i’ any node, $2i+1$ gives the position of left child and $2i+2$ gives the position of right child.

In the above diagram, B’s position is 1. $2*1+2$ gives 4, which is the position of its right child D.

- Given the position ‘i’ of any node, $(i-1)/2$ gives the position of its parent. Position of E is 2. $(2-1)/2$ gives 0, which is the position of its parent A.

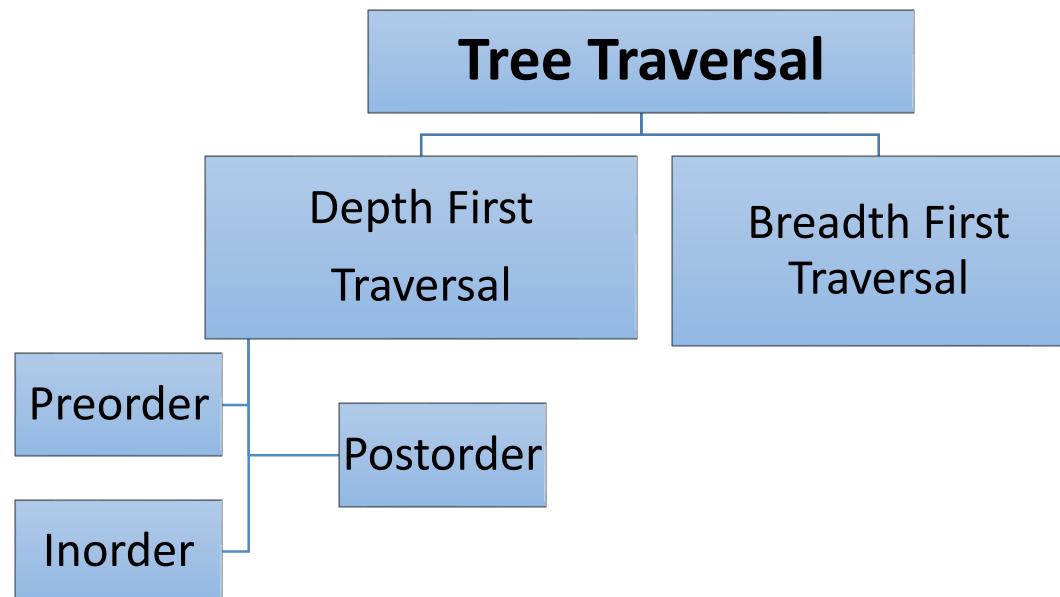
Various operations that can be performed on binary trees:

- Insertion : inserting an item into the tree.
- Traversal : visiting the nodes of the tree one by one.
- Search : search for the specified item in the tree.
- Copy : to obtain exact copy of given tree.
- Deletion : delete a node from the tree.

```
void insert_rec(NodePtr *root, int data) {  
  
    if(*root==NULL) {  
        NodePtr tmp = malloc(sizeof(Node));  
        tmp->info= data;  
        tmp->llink= tmp->rlink= NULL;  
  
        *root = tmp;  
        return;  
    }  
  
    printf("Insert to the left or right of %d?\n", (*root)->data);  
    char str[10];  
    scanf("%s", str);  
    if(strcmp(str,"left")==0) {  
        insert_rec(&((*root)->llink), data);  
    } else {  
        insert_rec(&((*root)->rlink), data);  
    }  
}
```

Binary Tree Traversal.

- A **binary tree traversal** requires that **each node of the tree be processed once and only once** in a predetermined sequence.

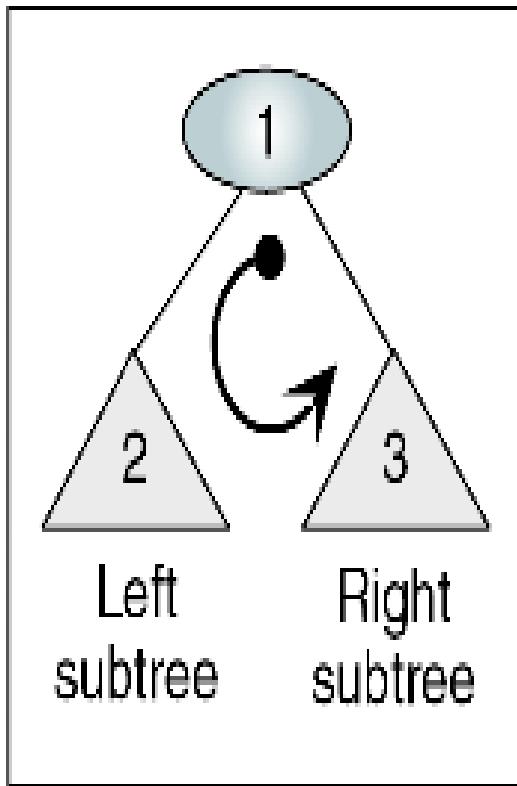


Depth First Traversal.

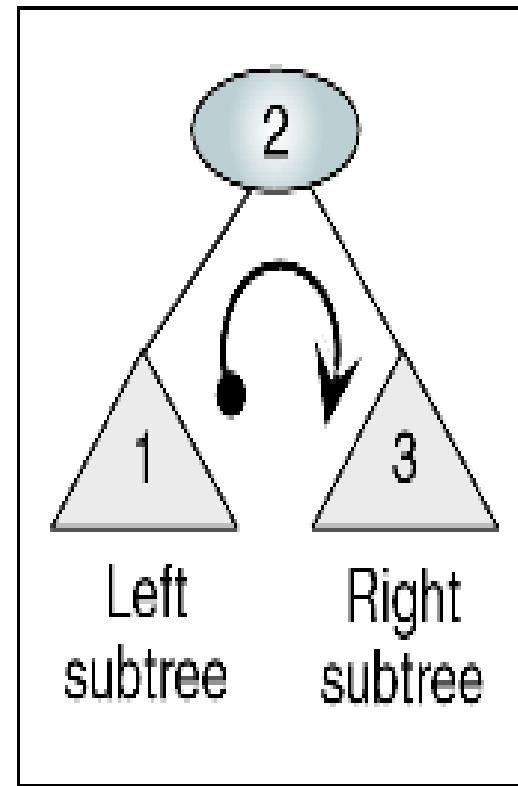
- In the **depth-first traversal** processing proceeds along a path from the **root through one child to the most distant descendant** of that first child before processing a second child.

OR

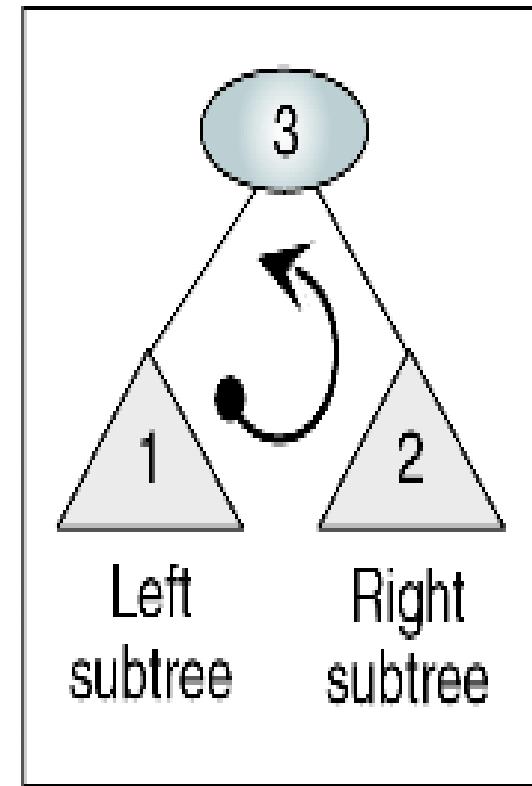
- We process all the descendants of a child before going on to the next child.



(a) Preorder traversal



(b) Inorder traversal



(c) Postorder traversal

FIGURE 6-8 Binary Tree Traversals

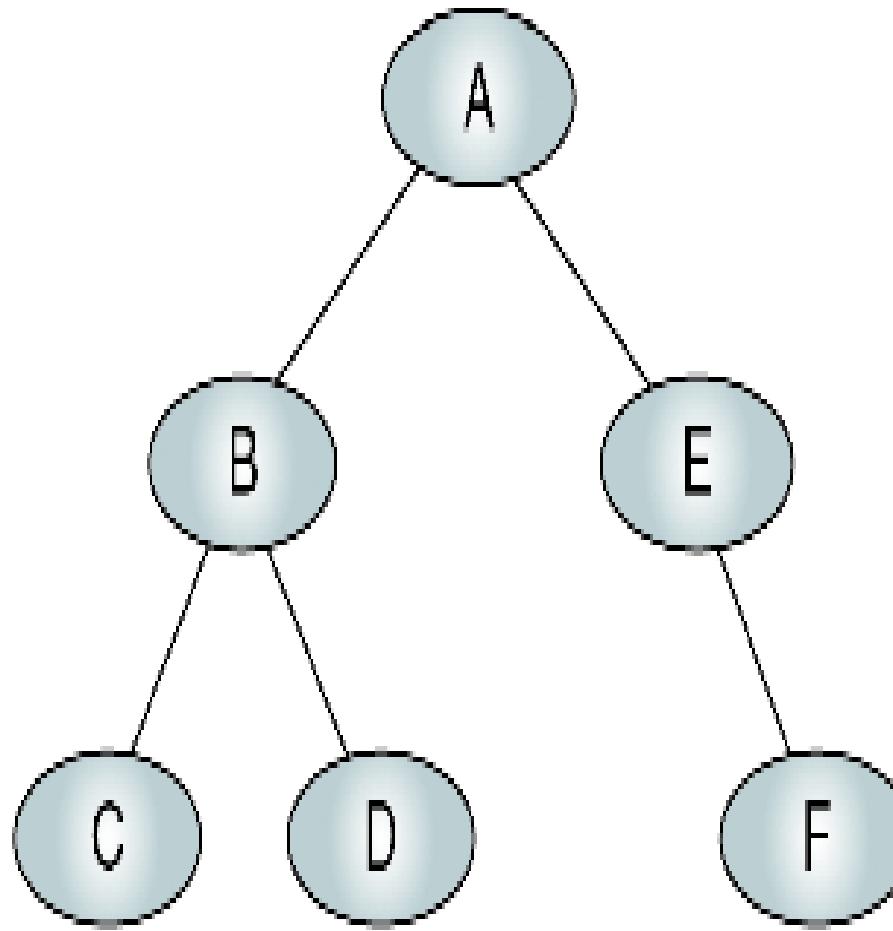
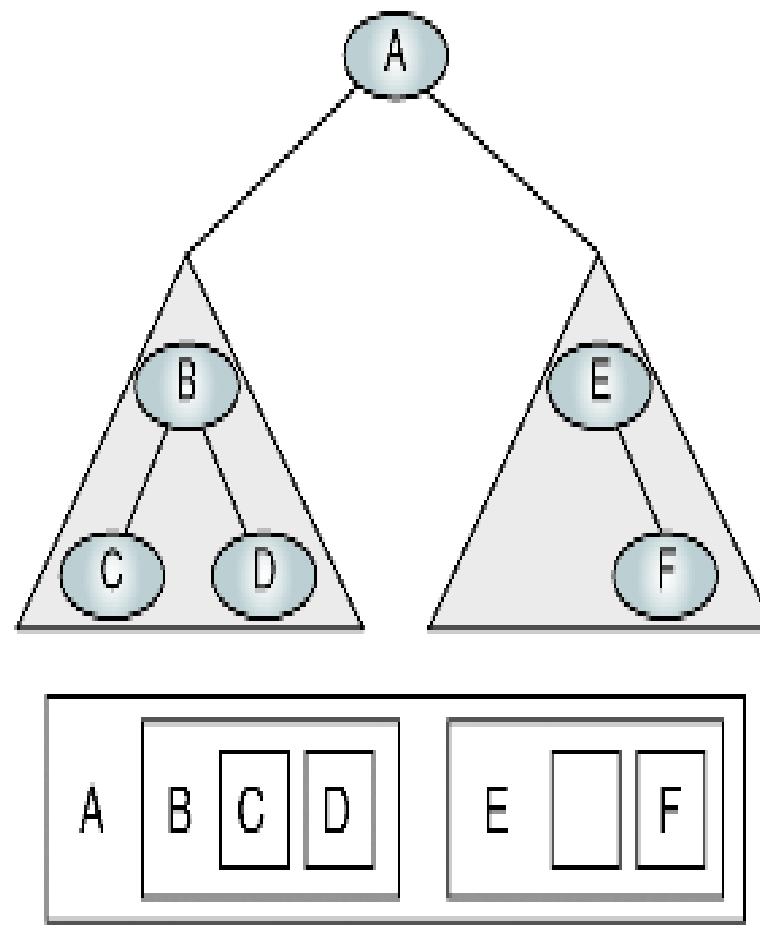


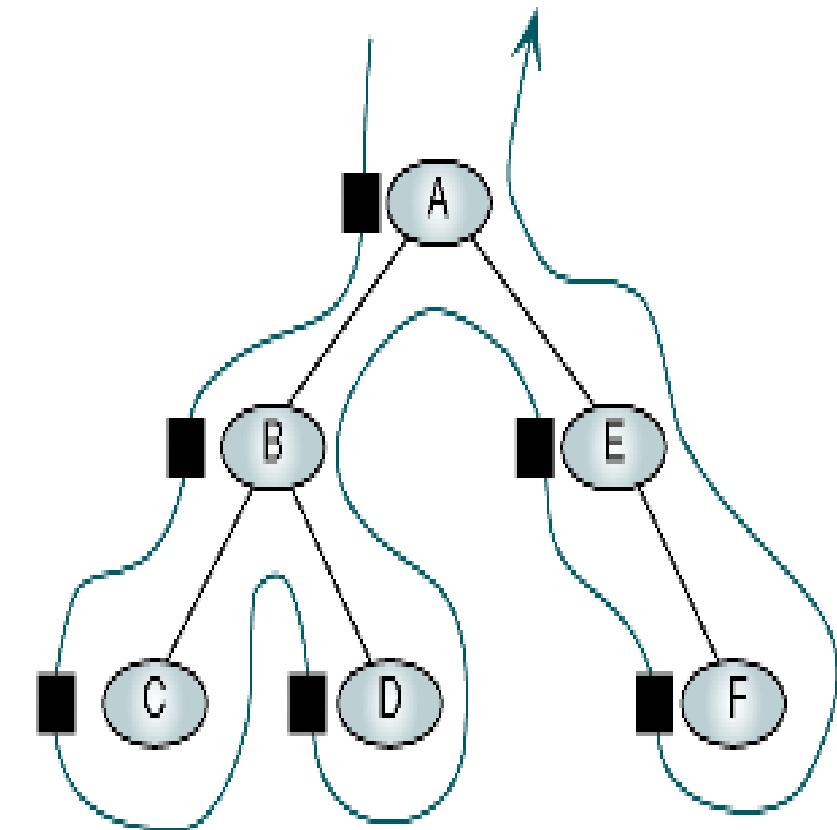
FIGURE 6-9 Binary Tree for Traversals

ALGORITHM 6-2 Preorder Traversal of a Binary Tree

```
Algorithm preOrder (root)
Traverse a binary tree in node-left-right sequence.
    Pre  root is the entry node of a tree or subtree
    Post each node has been processed in order
1 if (root is not null)
    1 process (root)
    2 preOrder (leftSubtree)
    3 preOrder (rightSubtree)
2 end if
end preOrder
```

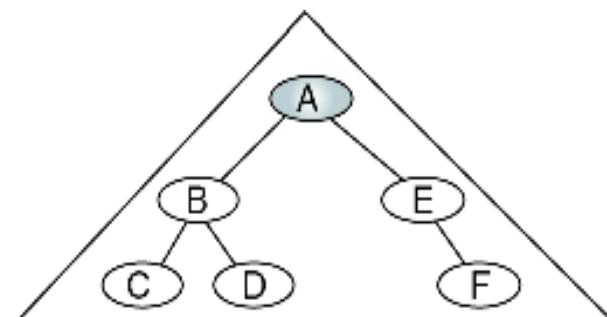


(a) Processing order

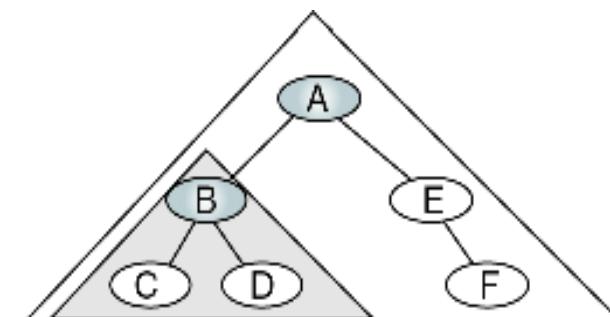


(b) "Walking" order

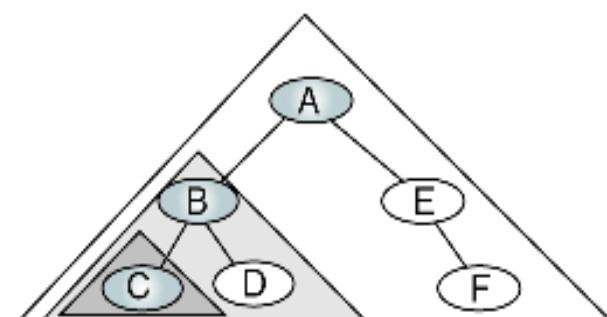
FIGURE 6-10 Preorder Traversal—A B C D E F



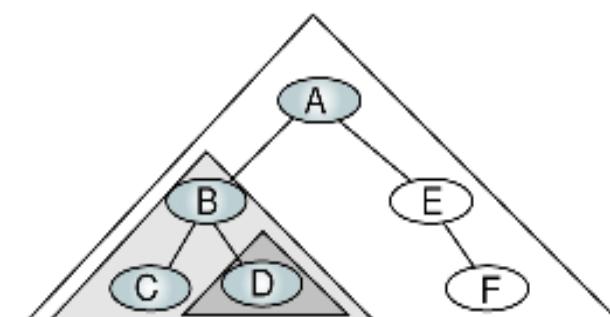
(a) Process tree A



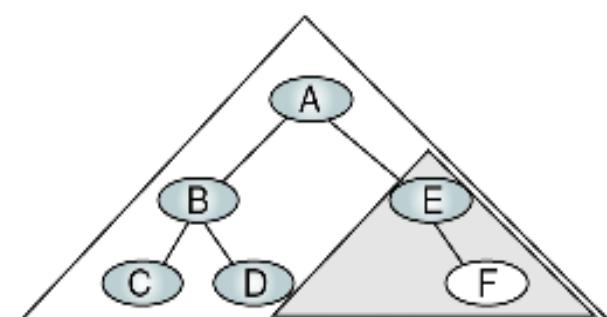
(b) Process tree B



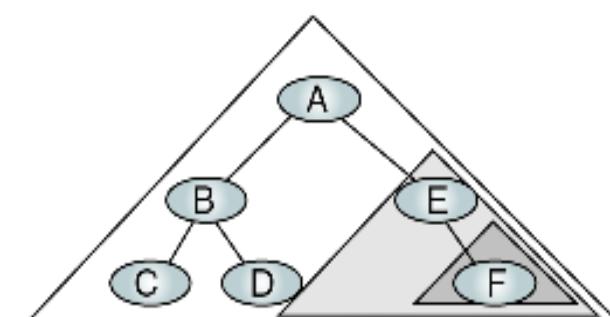
(c) Process tree C



(d) Process tree D



(e) Process tree E

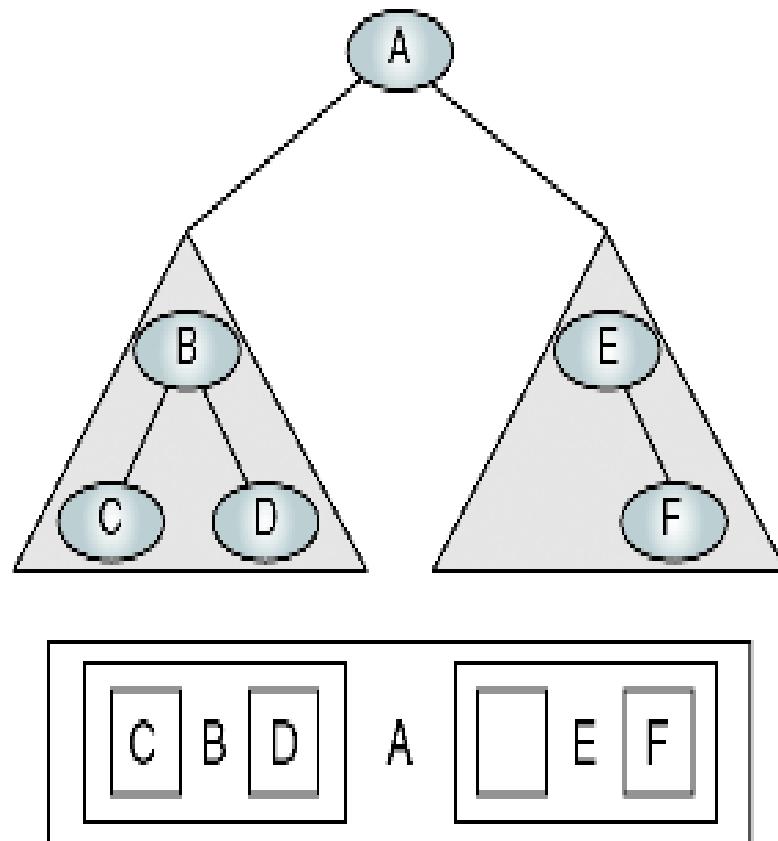


(f) Process tree F

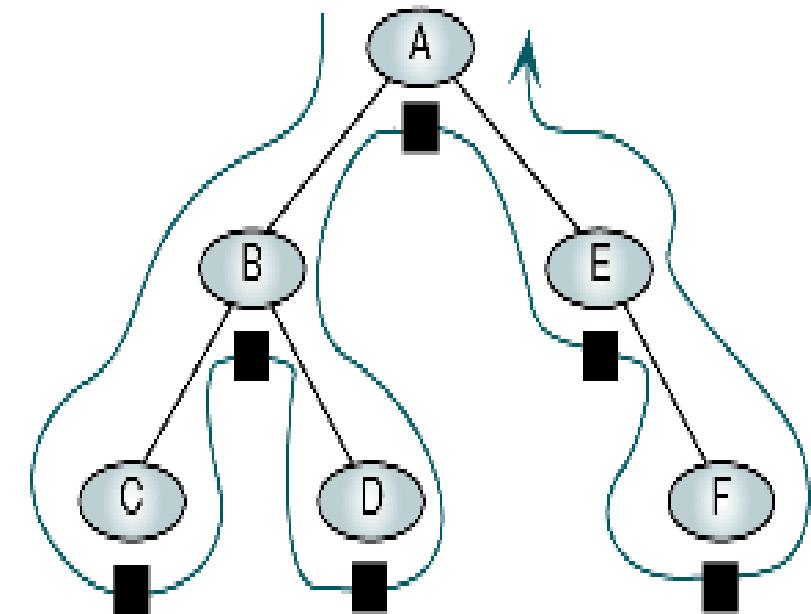
FIGURE 6-11 Algorithmic Traversal of Binary Tree

ALGORITHM 6-3 Inorder Traversal of a Binary Tree

```
Algorithm inOrder (root)
Traverse a binary tree in left-node-right sequence.
Pre root is the entry node of a tree or subtree
Post each node has been processed in order
1 if (root is not null)
    1 inOrder (leftSubTree)
    2 process (root)
    3 inOrder (rightSubTree)
2 end if
end inOrder
```



(a) Processing order



(b) "Walking" order

FIGURE 6-12 Inorder Traversal—C B D A E F

ALGORITHM 6-4 Postorder Traversal of a Binary Tree

Algorithm postOrder (root)

Traverse a binary tree in left-right-node sequence.

Pre root is the entry node of a tree or subtree

Post each node has been processed in order

1 if (root is not null)

 1 postOrder (left subtree)

 2 postOrder (right subtree)

 3 process (root)

2 end if

end postOrder

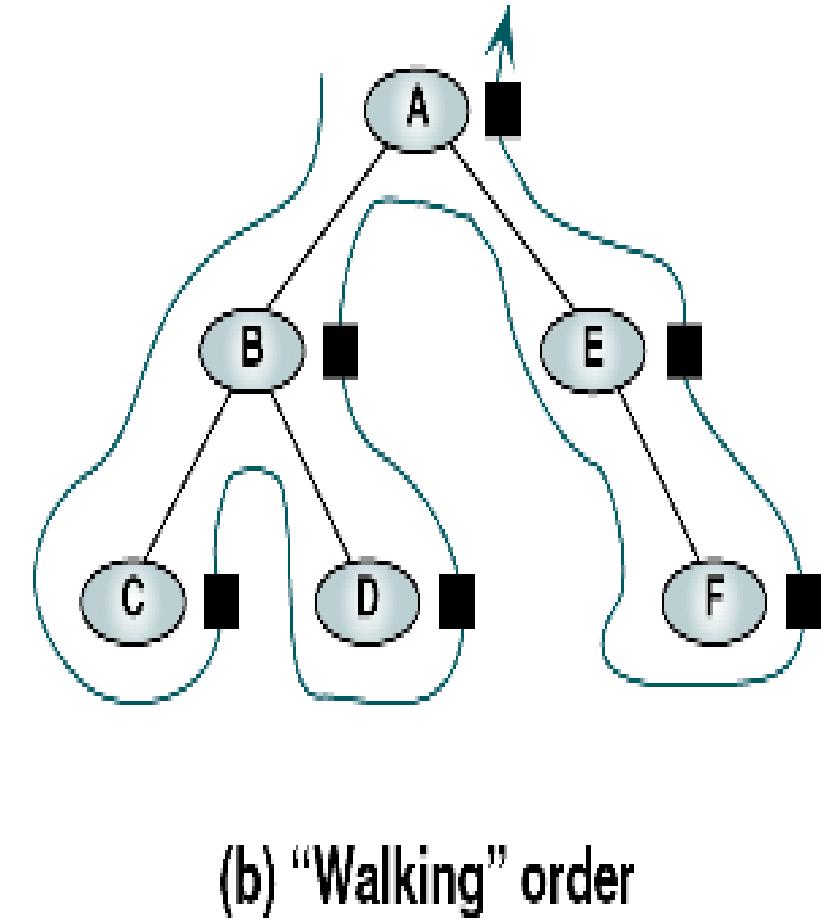
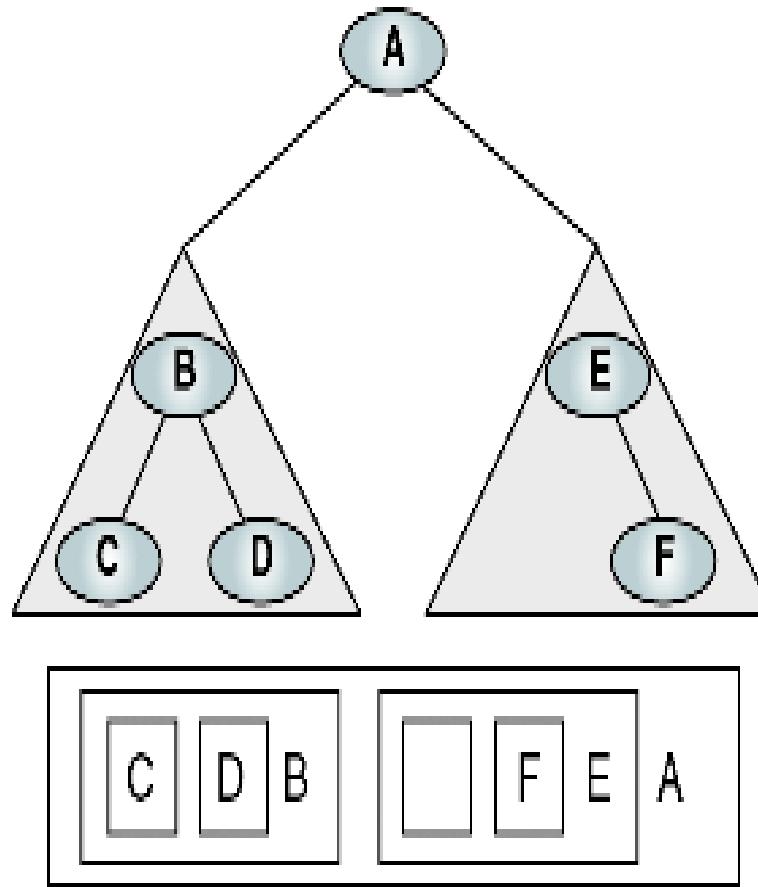
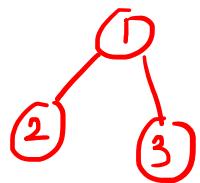
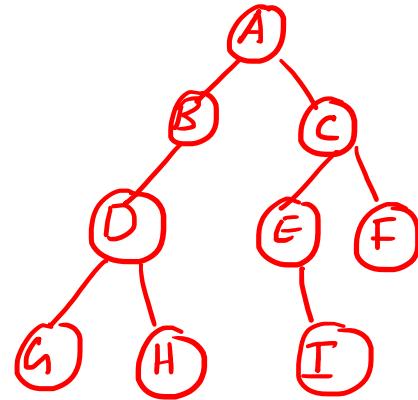


FIGURE 6-13 Postorder Traversal—CDBFEA

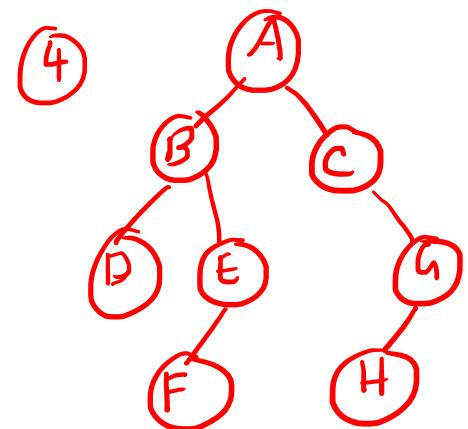
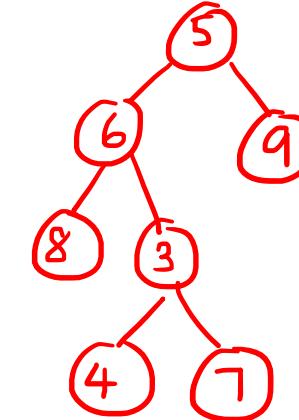
Ex :- 1



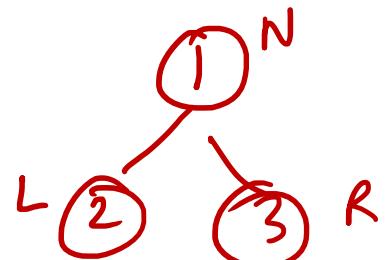
Ex :- 2



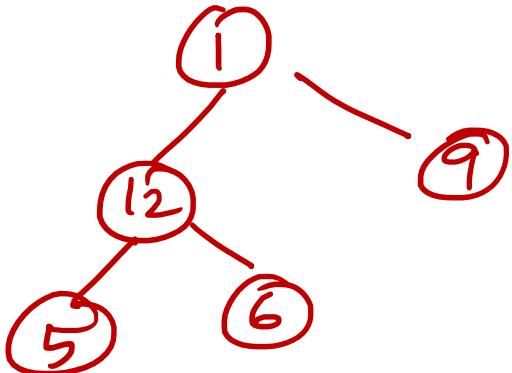
Ex :- 3



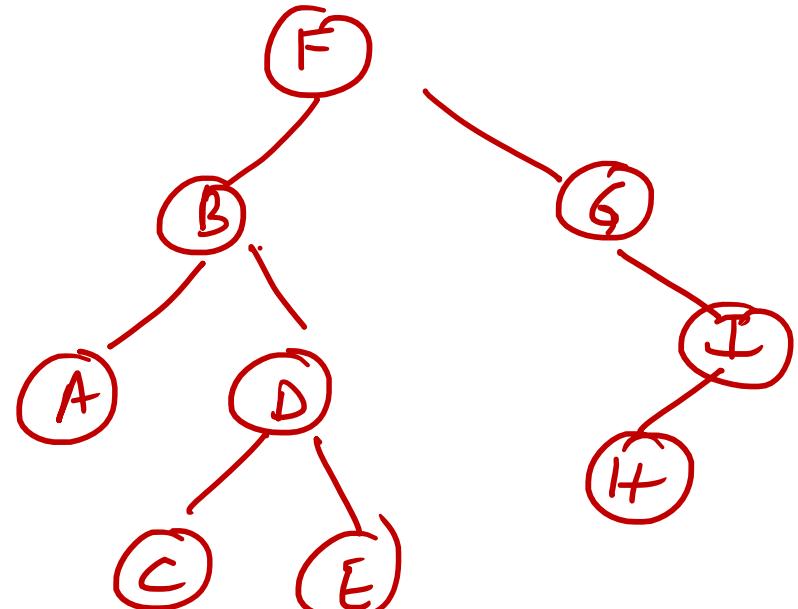
More Examples.



I 1, 2, 3
II 2, 1, 3
III 2, 3, 1

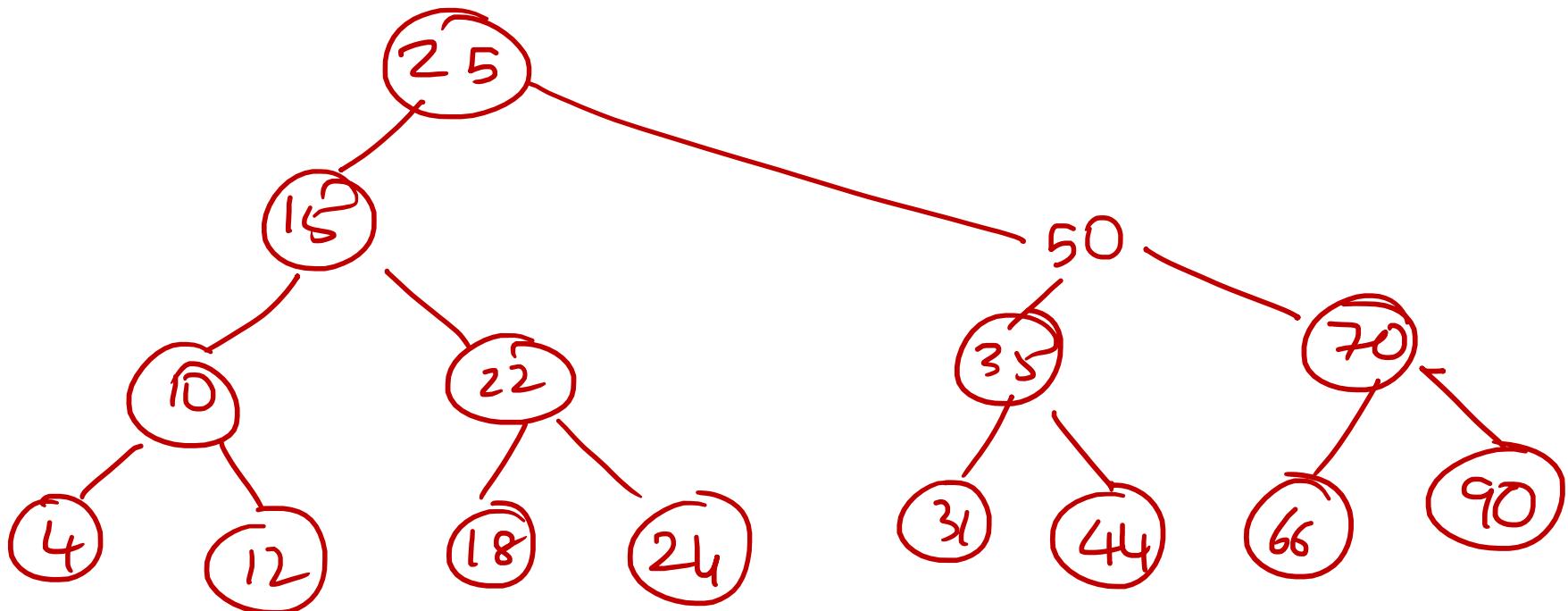


I 1, 12, 5, 6, 9
II 5, 12, 6, 1, 9
III 5, 6, 12, 9, 1



I F, B, A, D, C, E, G, I H
II A, B, C, D, E, F, G, H, I
III A, C, E, D, B, H, I, G, F

More Examples.



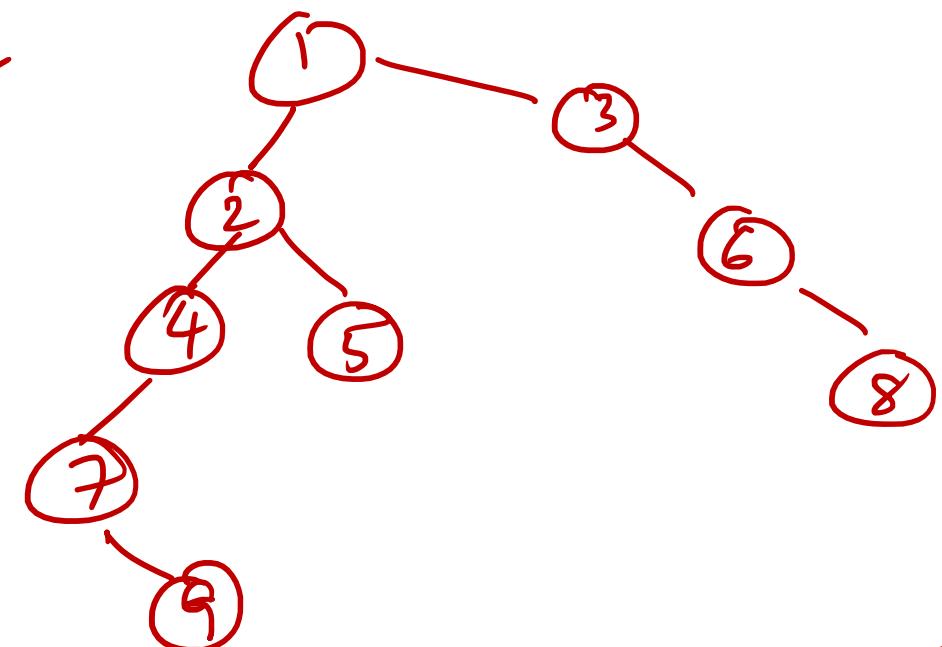
I 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90.

II 4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

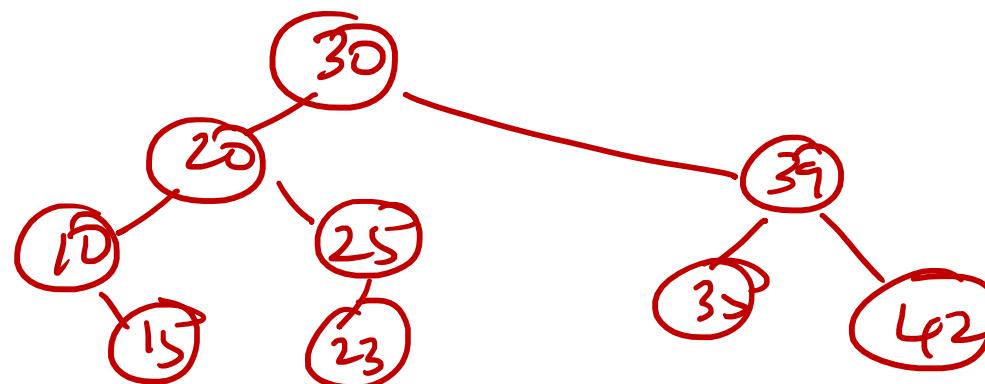
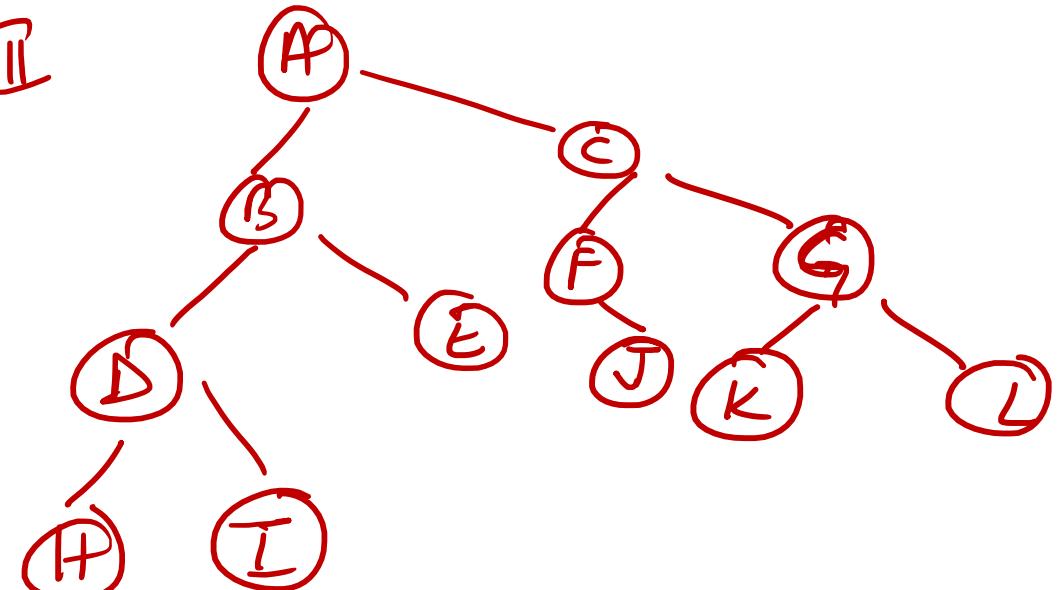
III 4, 12, 10, 18, 12, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

More Examples.

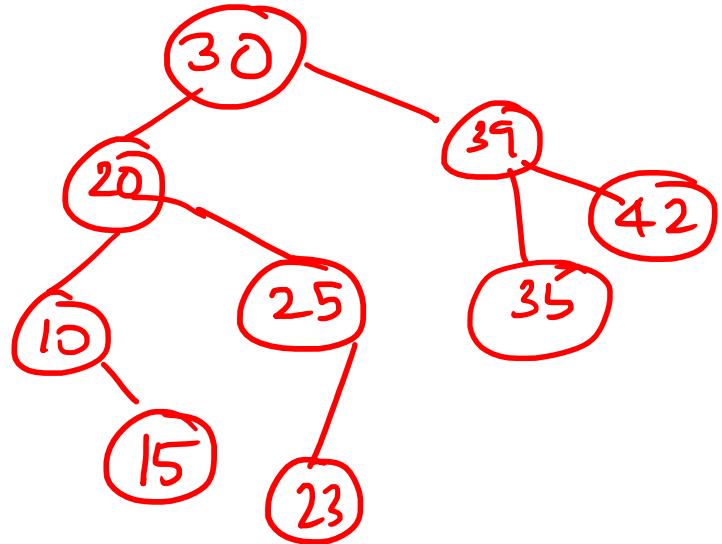
I



II



NLR .

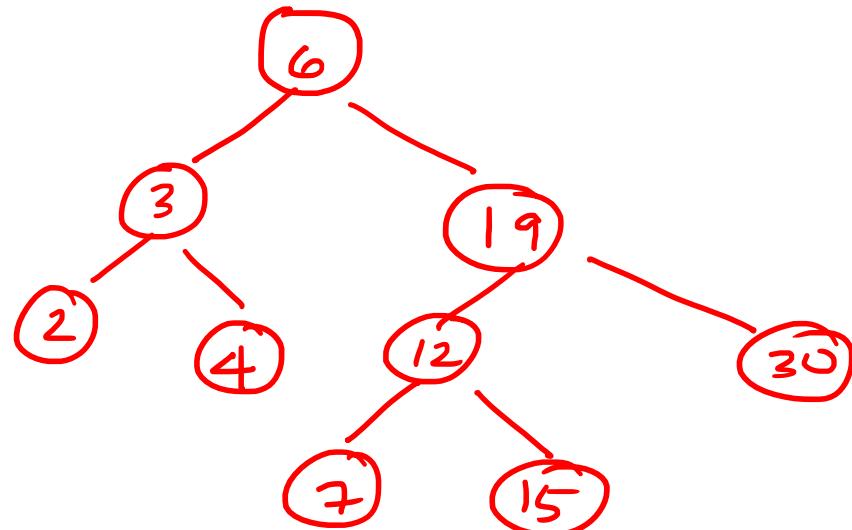


I) 30, 20, 10, 15, 25, 23, 39, 35, 42

II) 10, 15, 20, 23, 25, 30, 35, 39, 42

III) 15, 10, 23, 25, 35, 42, 39, 30

LRN

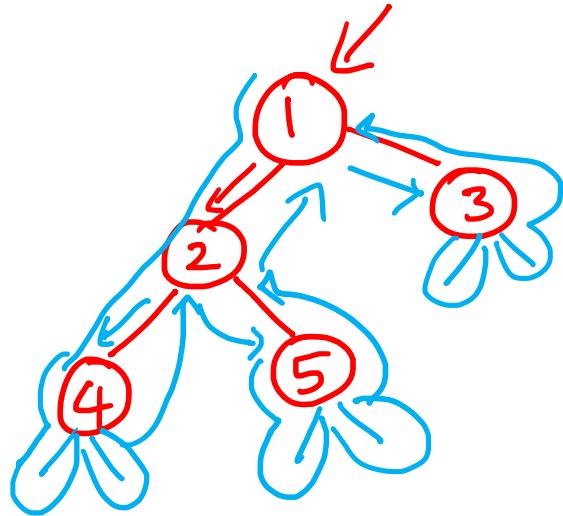


I) 6, 3, 2, 4, 19, 12, 7, 15, 30

II) 2, 3, 4, 6, 7, 12, 15, 19, 30

III) 2, 4, 3, 7, 15, 12, 30, 19, 6.

Binary Tree



Preorder - RLR → First
Inorder → LNR → 2nd
Postorder → LRN → 3rd

Euler's

Preorder - 1, 2, 4 , 5, 3

Inorder - 4, 2, 5, 1 , 3

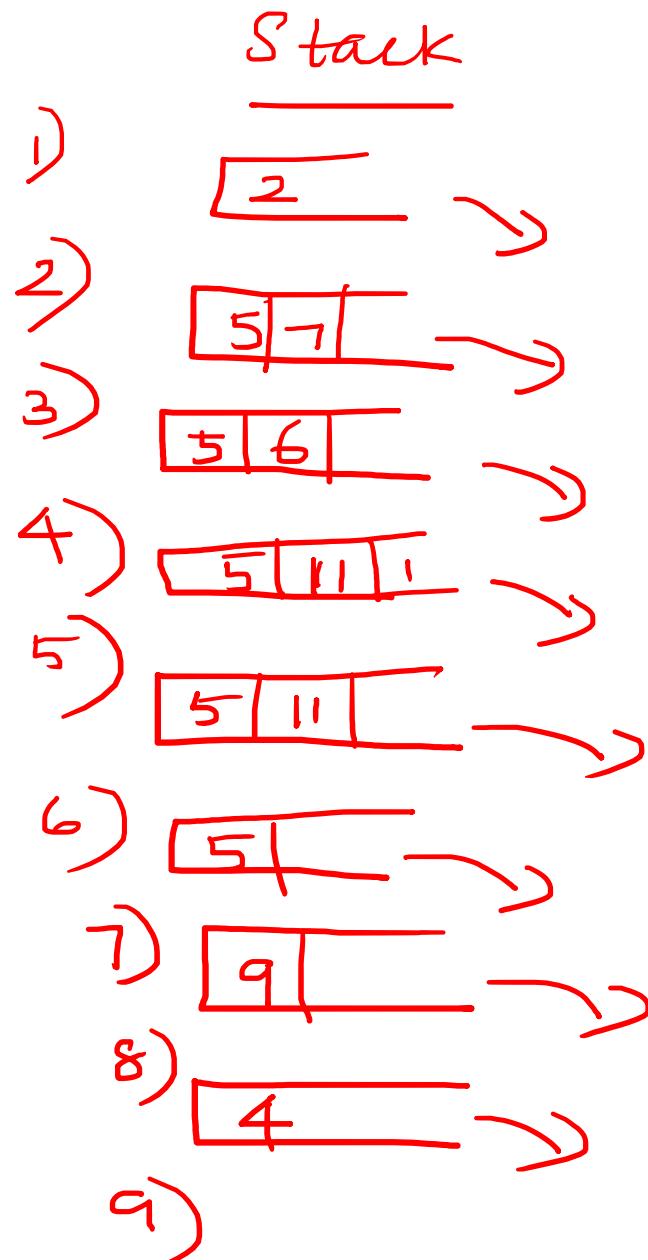
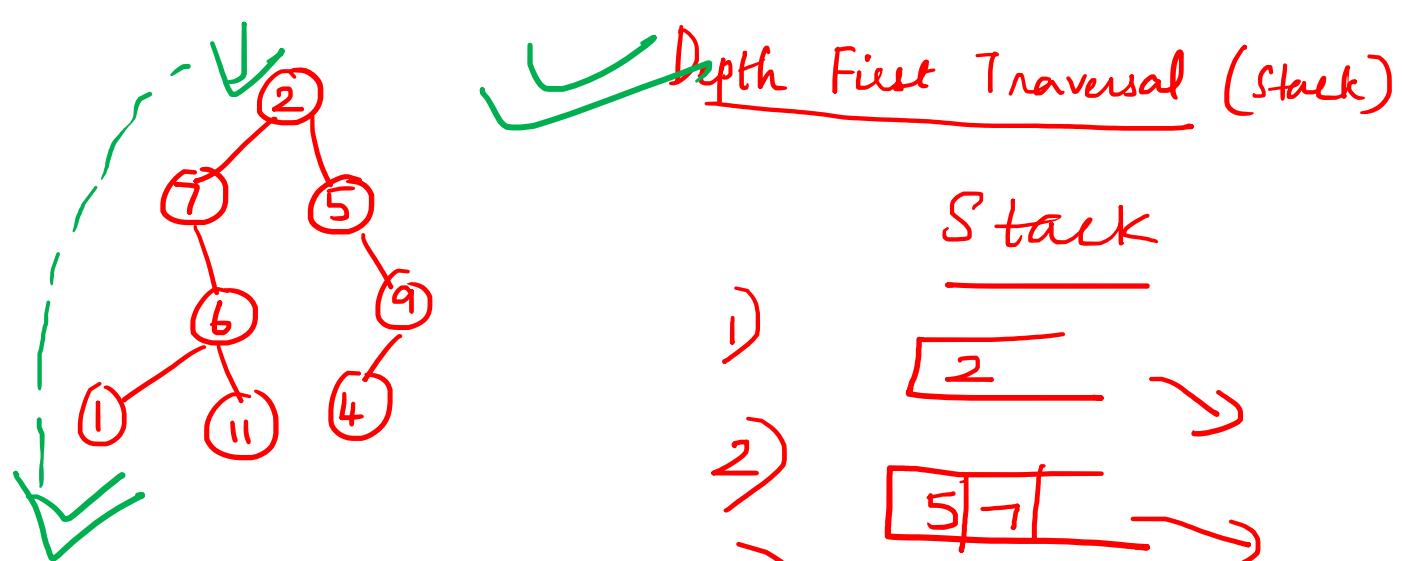
Postorder - 4, 5, 2 , 3 , 1

DEPTH FIRST TRAVERSAL ALGORITHM

- USE Stack.

Steps:

- Add root to the Stack.
- Pop out an element from Stack , process it, and add its **right** and **left children** to stack.
- Repeat the above two steps until the Stack is empty.



Output String

2

2 7

2 7 L

2 7 6 1

2 7 6 1 11

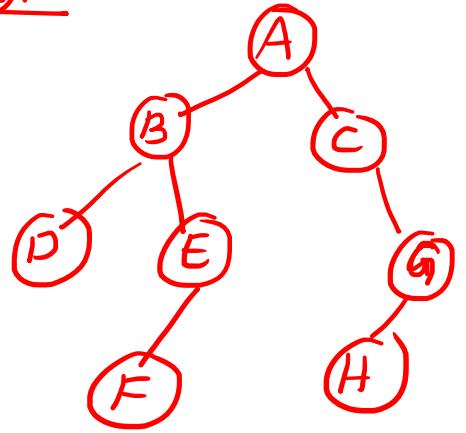
2 7 6 1 11 5

2 7 6 1 11 5 9

Preorder

2 7 6 1 11 5 9 4

Ex ②:



BREADTH FIRST TRAVERSAL ALGORITHM

(Level Order
Traversal)

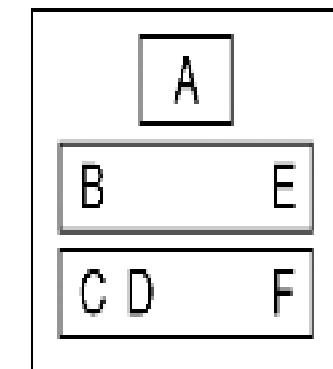
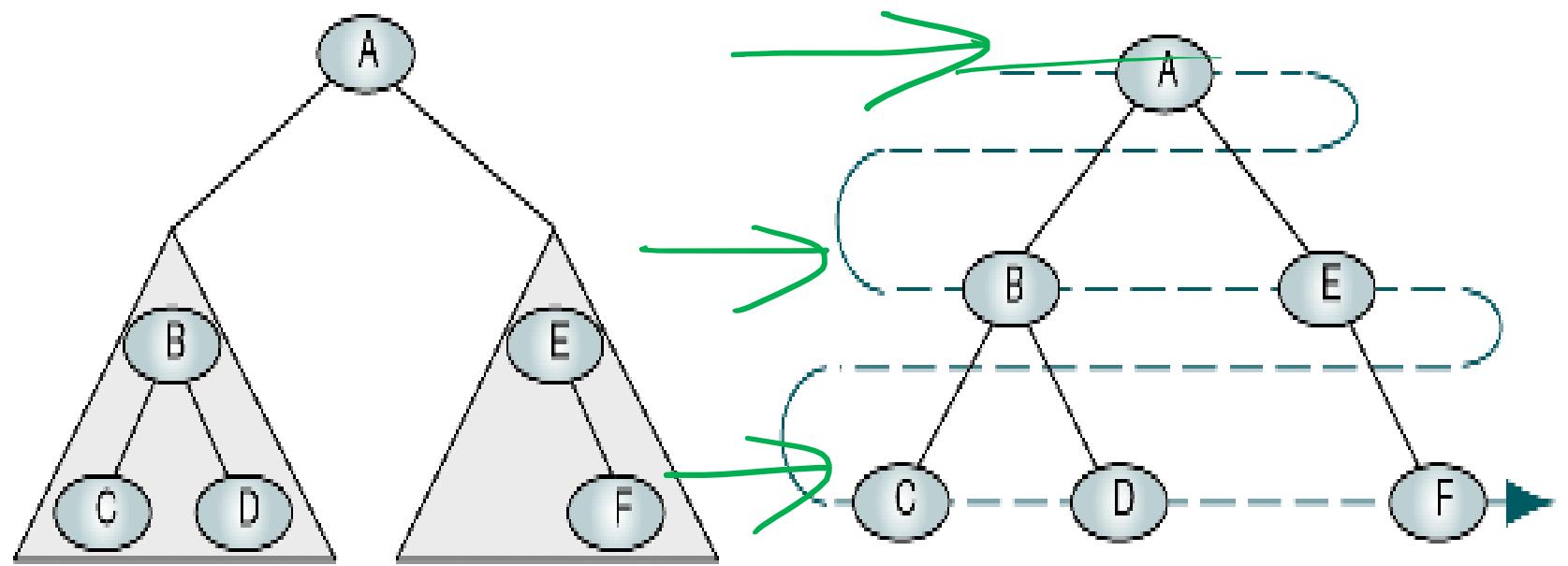
- USE Queue.

Steps:

- ~~Remove~~ • Add root to the Queue.

- Pop out an element from queue, process it, and add its left and right children to the queue.
- Repeat the above two steps until the queue is empty.

S

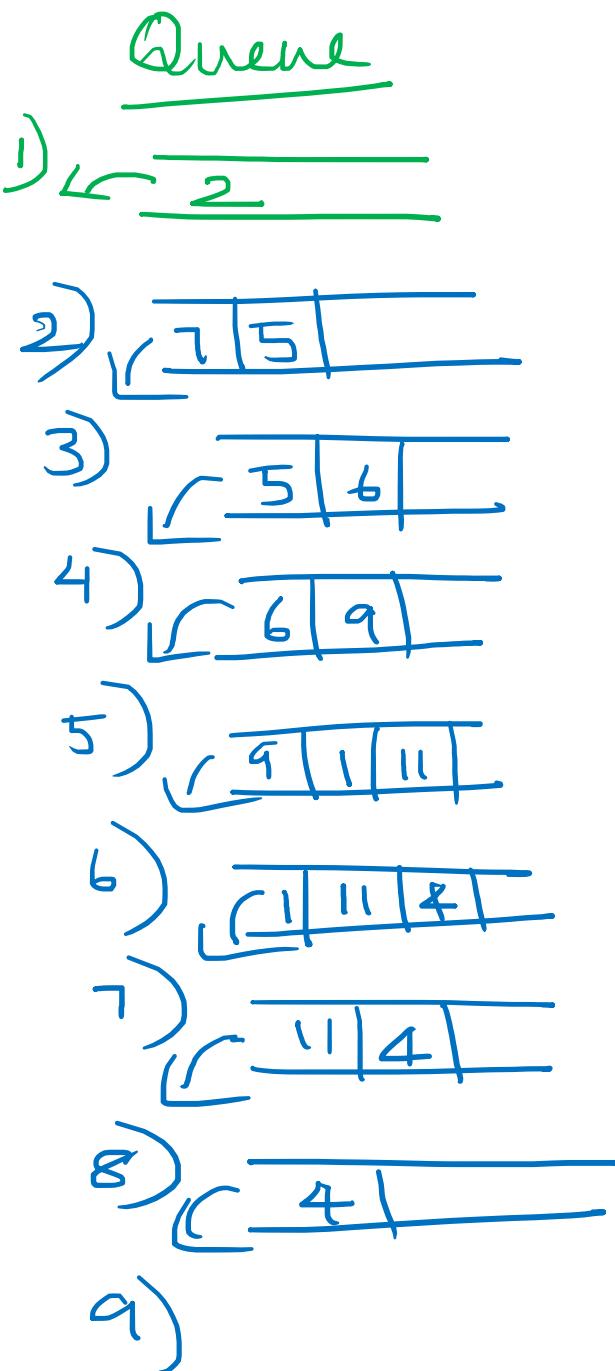
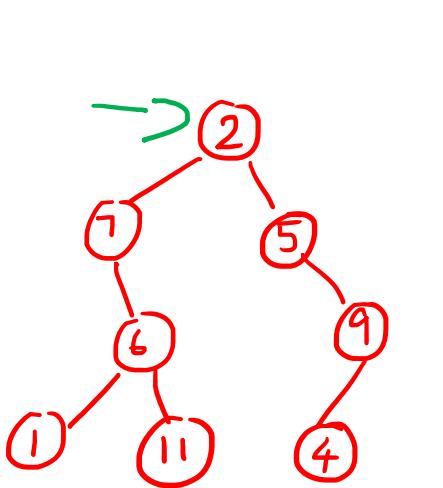


(a) Processing order

(b) "Walking" order

FIGURE 6-14 Breadth-first Traversal

Ex: ①



Output String

2 .

2 7

2 7 5

2 7 5 6

2 7 5 6 9

2 7 5 6 9 1

2 7 5 6 9 1 11

2 7 5 6 9 1 11 4

DEPTH FIRST SEARCH ALGORITHM

{ STACK }

search key element

Put the root node onto the stack
while (stack is not empty)

do

remove a node from the stack

if (node is a goal node)

return success

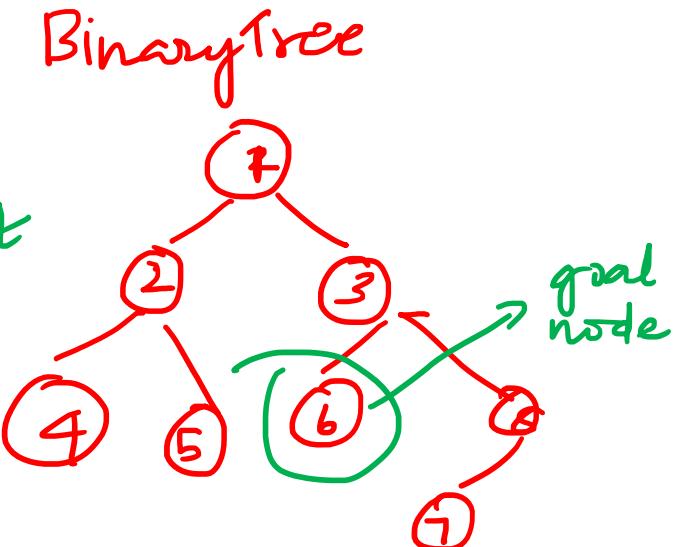
Put its right and left children onto the stack

done

return failure

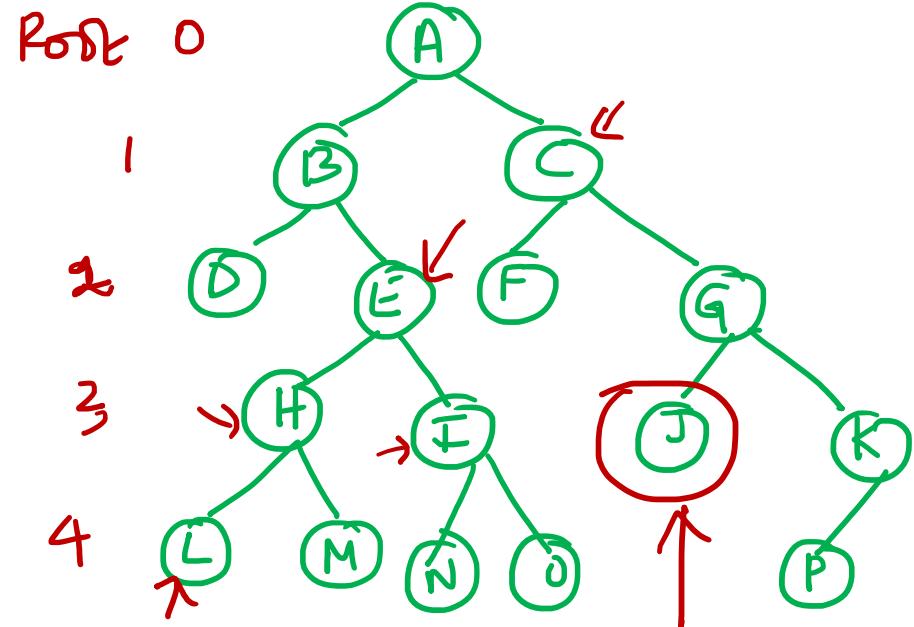
Not found

right → first
left → next



Key element = 6

Example of DFS



Height ?
Balance factor ?

found

Goal Node = J

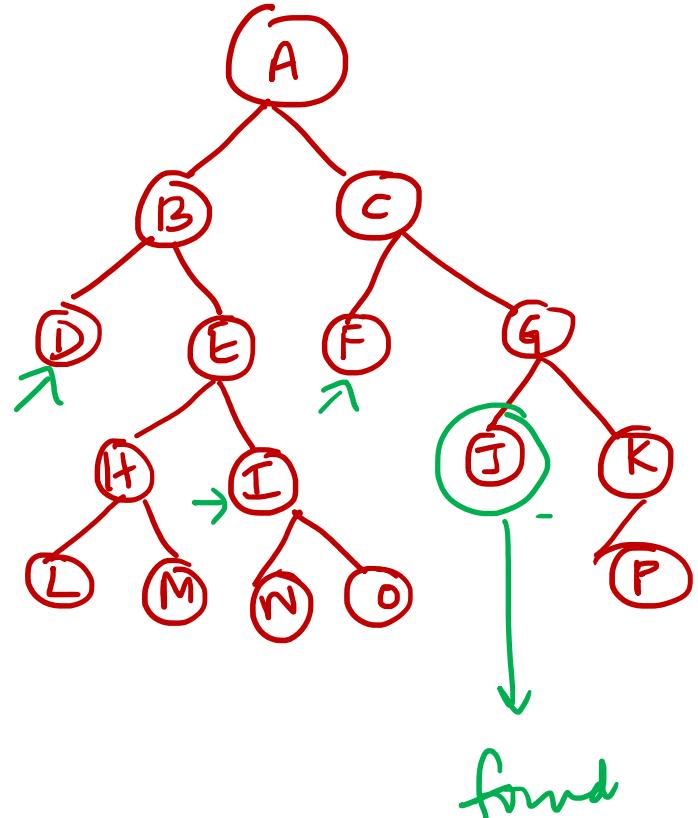
	Stack	Check
1.	A	
2.	C B	A = J ? No
3.	C E D	B = J ? N
4.	C E	D = J ? N
5.	C I H	E = J ? N
6.	C I M L	H = J ? N
7.	C I M	L = J ? N
8.	C I	M = J ? N
9.	C O N	I = J ? N
10.	C O T	N = J ? N
11.	C F	O = J ? N
12.	G F	C = J = N
13.	G	
14.	K J	G = J / N
15.	J = J	
		SUCCESS

Level Order Traversal (Also called)

BREADTH FIRST SEARCH ALGORITHM.

```
Add root to queue  
while (queue is not empty)  
do  
    remove a node from the queue  
    if (node is a goal node)  
        return success  
    put its left and right children onto the queue  
done  
return failure
```

Example for BFS



Search element = J → Goal Node

	<u>Queue</u>	<u>Check</u>	
1.	F R ← A	-	At one end (front)
2.	← B C	A = J? N	
3.	← C D E	B = J? N	
4.	← D E F G	C = J? N	
5.	← E F G	D = J? N	
6.	← F G H I	E = J? N	
7.	← G H I	F = J? N	
8.	← H I J K	G = J? N	
9.	← I J K L M	H = J? N	
10.	← J K L M N O	I = J? N	
11.		J = J	Succes → Found

BINARY TREE CONSTRUCTION.

1. Inorder and Preorder $\{ \text{LNR} \}$ $\{ \text{NLR} \}$

2. Inorder and Postorder $\{ \text{LRN} \}$

3. Preorder and Postorder

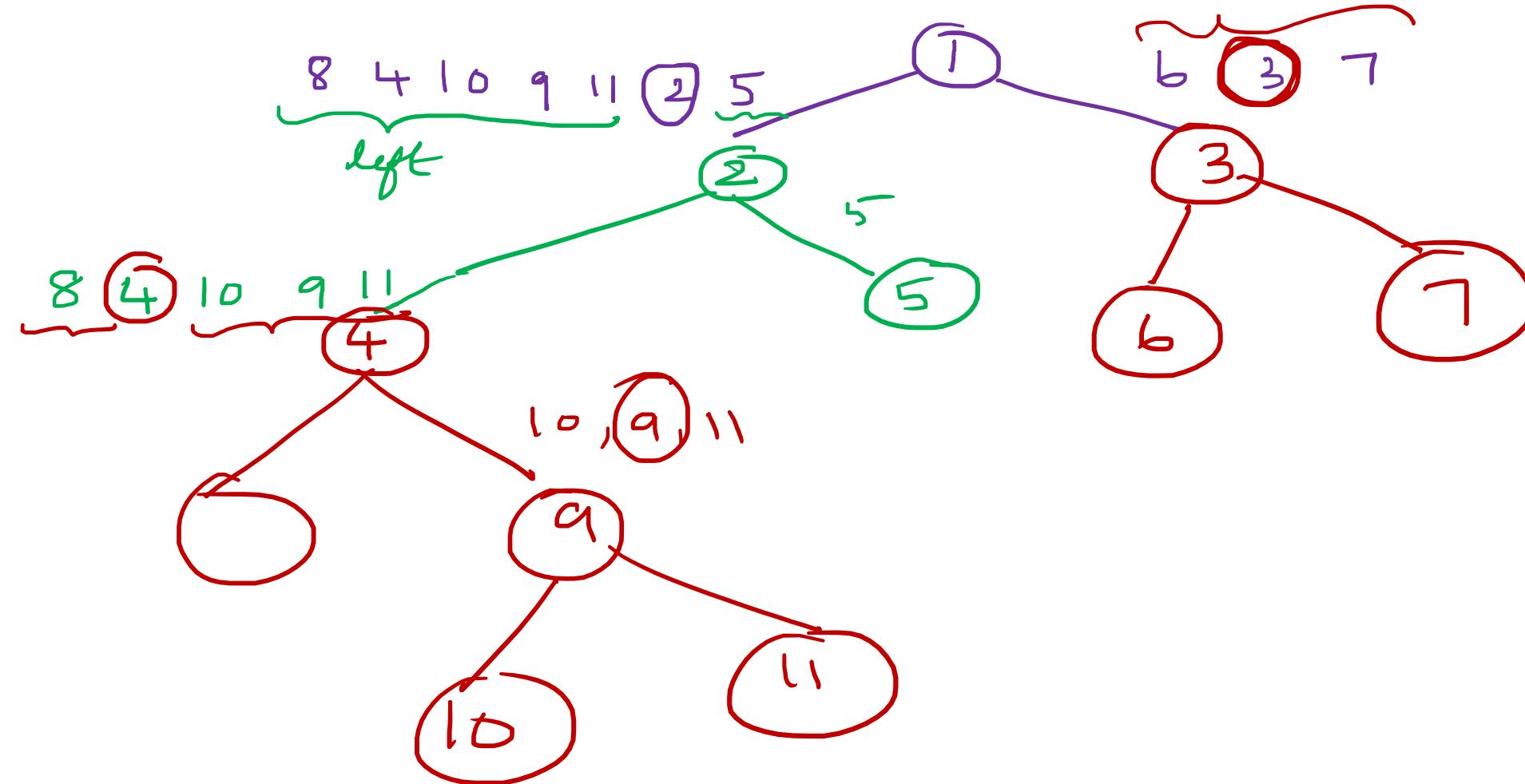
- ① Find root (Preorder)
 - ② Scan L → R
 - ③ Find left & Right subtrees from Inorder

Preorder :-

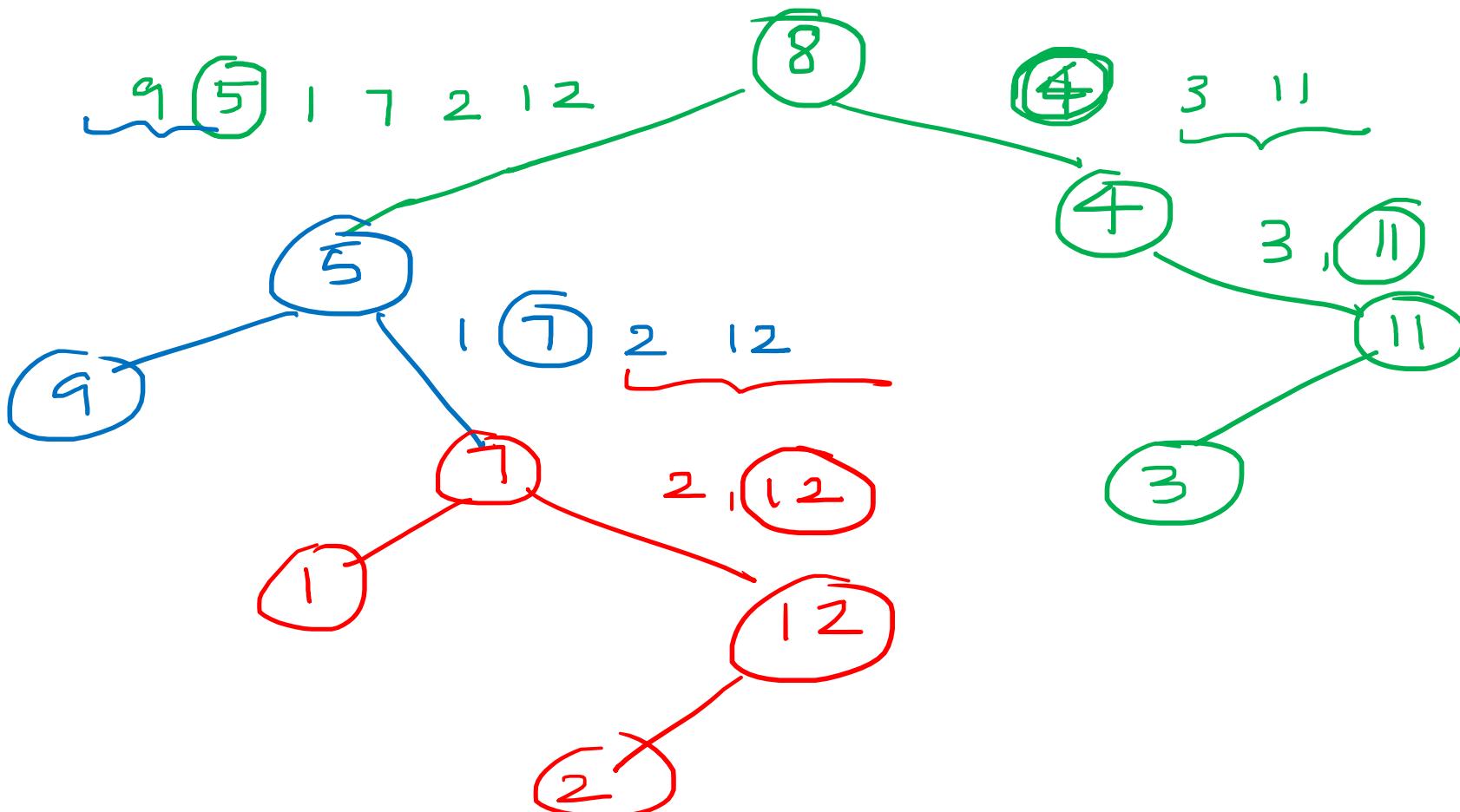
Inorder :-

(Parent left Right)

(left Parent Right)



- ① Find root (postorder) }
 ② Scan from R → L
 ③ Find left & Right subtrees
 from Inorder
- Construct Binary tree from Postorder & Inorder
- Postorder :- 9 1 2 12 7 5 3 11 4 8 (left Right Parent)
 Inorder :- 9 5 1 7 2 12 8 4 3 11 (left Parent Right)



HW Construct Binary tree

① Inorder :- E A C K F H D B G

Preorder :- F A E K C D H G B

② Inorder :- E A C K F H D B G

Postorder:- E C K A H B G D F

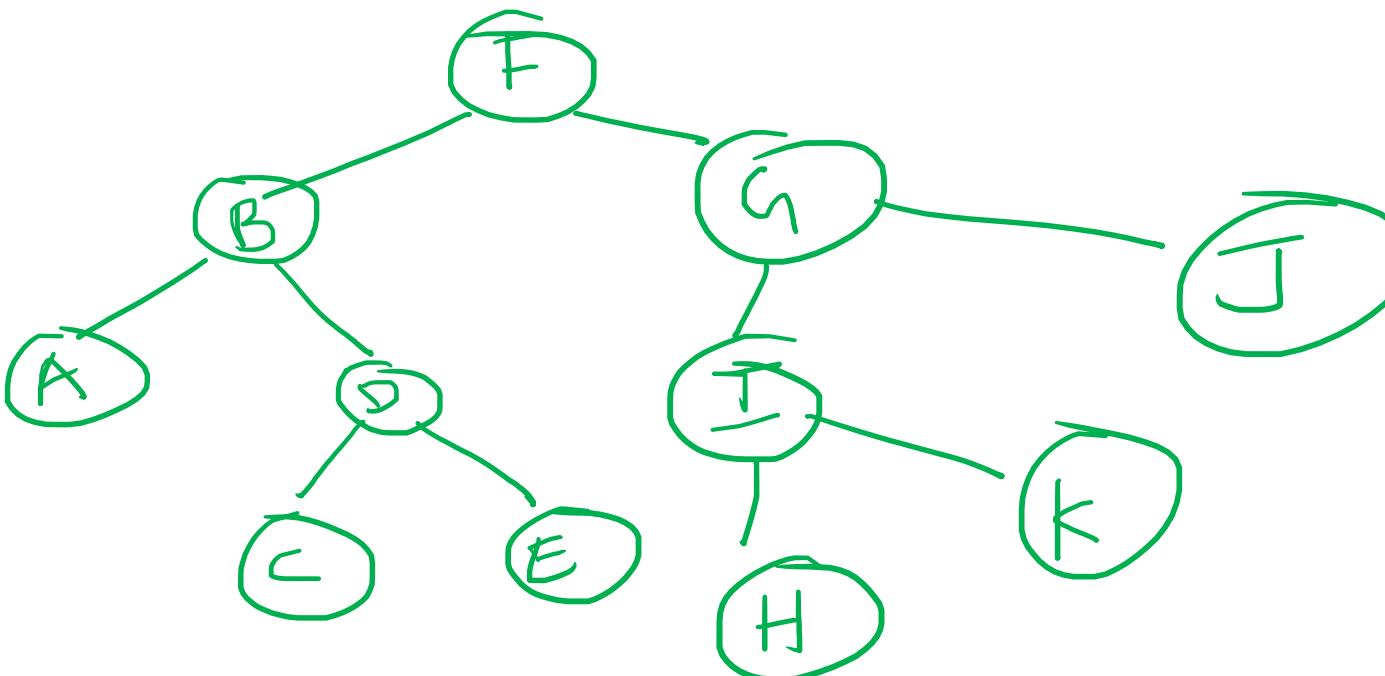
Construct unique full binary tree from preorder and postorder.

① Scan preorder L→R

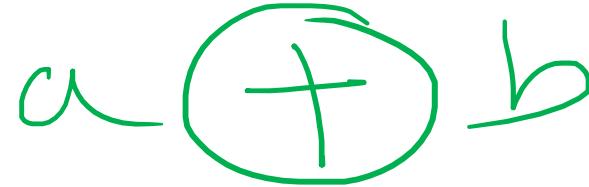
L → R

Preorder :- F B A D C E G I H K J (Parent Left Right)

Postorder :- A C E D B H K I J G F (Left Right Parent)



EXPRESSION TREES.



- One interesting application of binary trees is expression trees.
- An expression is a sequence of tokens that follow prescribed rules.
- A **token** may be either an operand or an operator.
- In this discussion we consider only binary arithmetic operators in the form operand-operator-operand.
- The standard arithmetic operators are +, -, *, and /.
- An **expression tree** is a binary tree with the following properties:
 - » Each leaf is an **operand**.
 - » The root and internal nodes are **operators**.
 - » Subtrees are subexpressions, with the root being an **operator**.

EXPRESSION TREES.

- For an expression tree, the three standard depth-first traversals represent the three different expression formats: infix, postfix, and prefix.
- The **inorder traversal** produces the **infix expression**.
- The **postorder traversal** produces the **postfix expression**, and
- The **preorder traversal** produces the **prefix expression**.

a × (b + c) + d

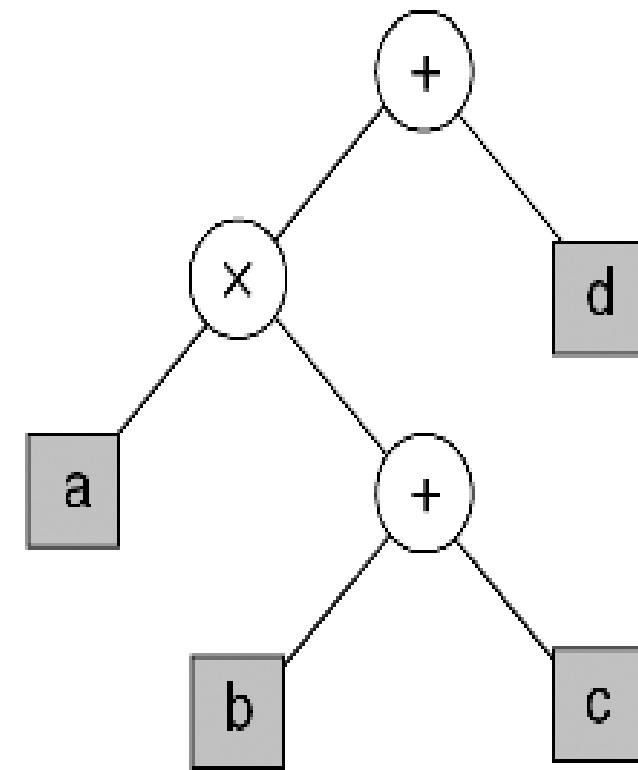
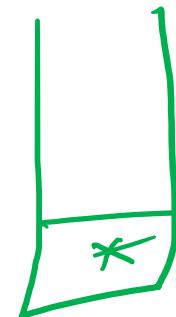
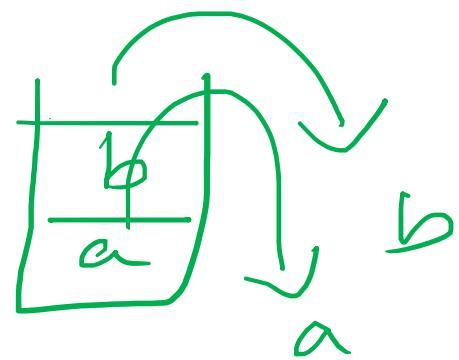


FIGURE 6-15 Infix Expression and Its Expression Tree

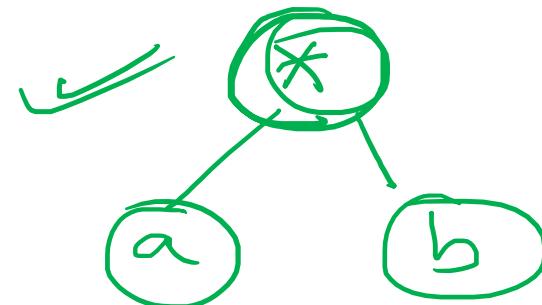
$$① \quad a + b * c - d / e + f$$

$$\textcircled{2} \quad A + ((B - C) * D) \wedge (E + F)$$

$a * b$ Postfix $\Rightarrow a b *$



\Rightarrow Stack
 $\Rightarrow L \rightarrow R$
 \Rightarrow operands push to stack
 \Rightarrow operator
pop 2 elts from stack, const tree



L \rightarrow R

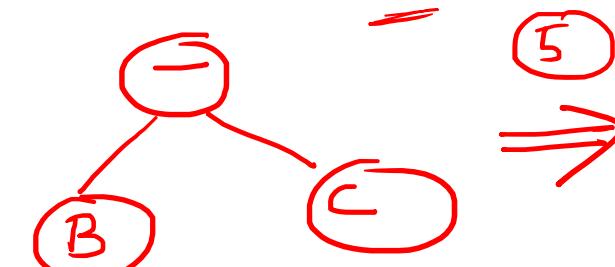
$$ABC - D * EF + \wedge +$$

①

-	C
B	A

④

⑥



⑤

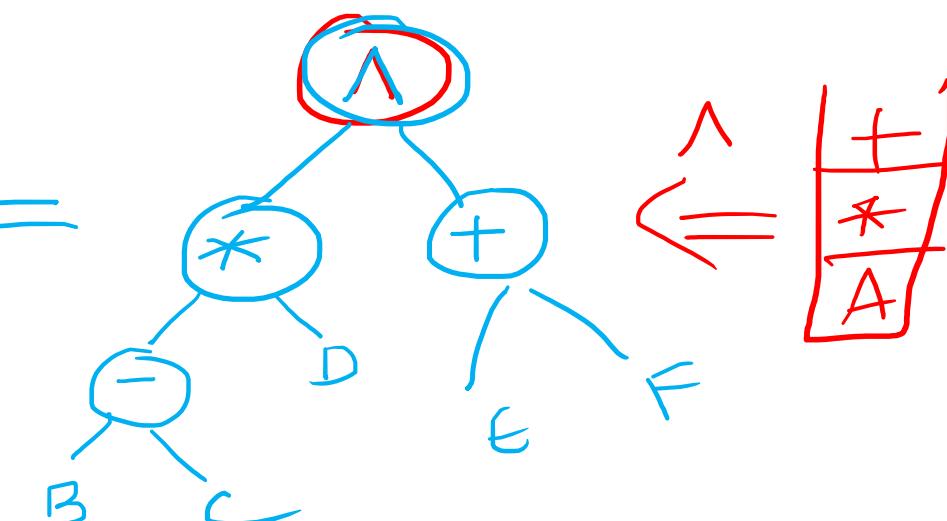
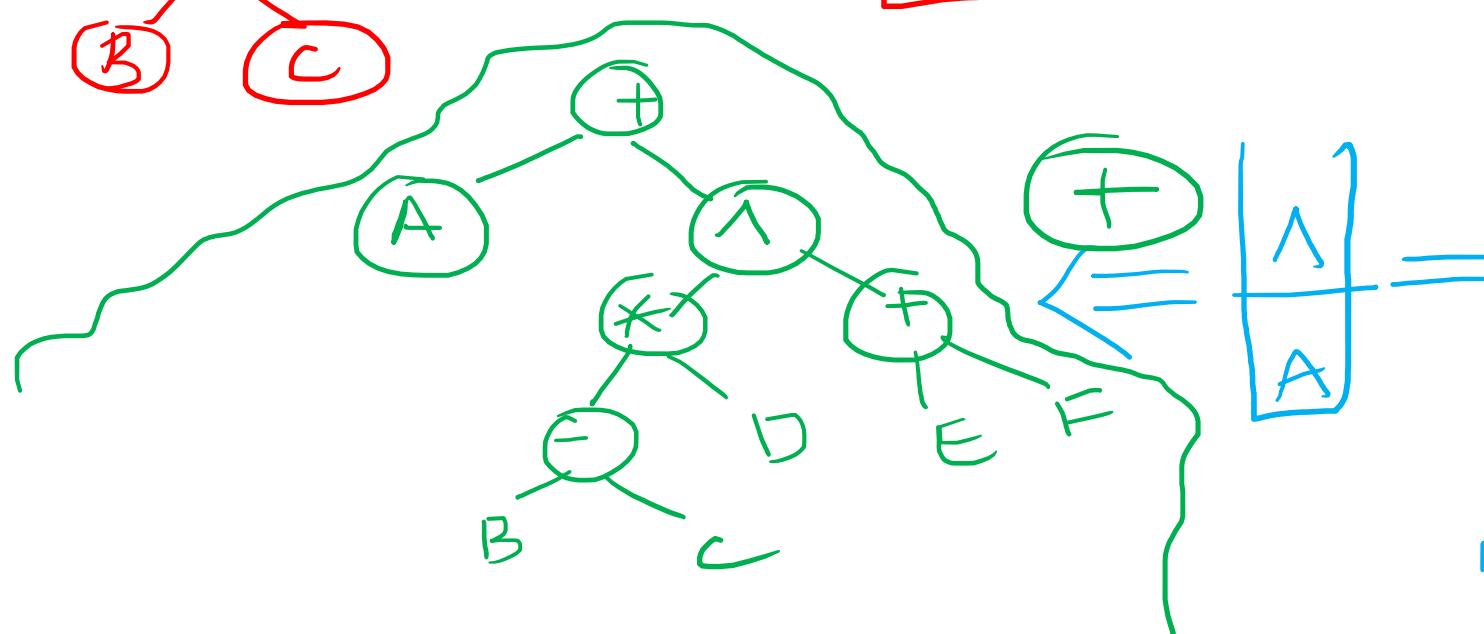
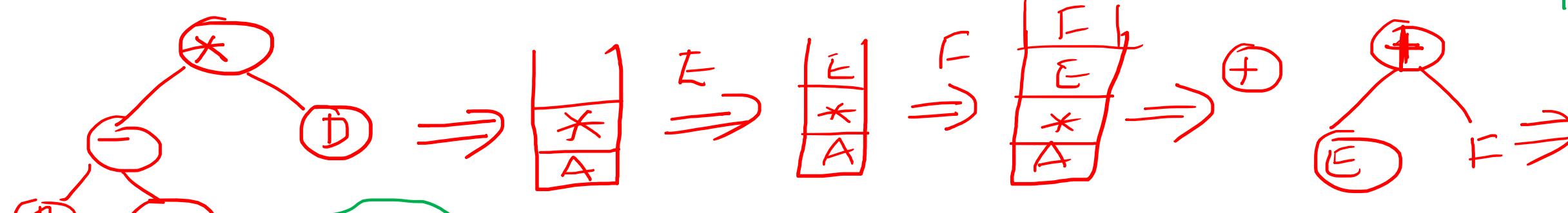
-	1
A	

⑥

D	1
A	

⑦

~



Note: When we print the infix expression tree, we must add an opening parenthesis at the beginning of each expression and a closing parenthesis at the end of each expression. Because the root of the tree and each of its subtrees represents a subexpression, we print the opening parenthesis when we start a tree or a subtree and the closing parenthesis when we have processed all of its children.

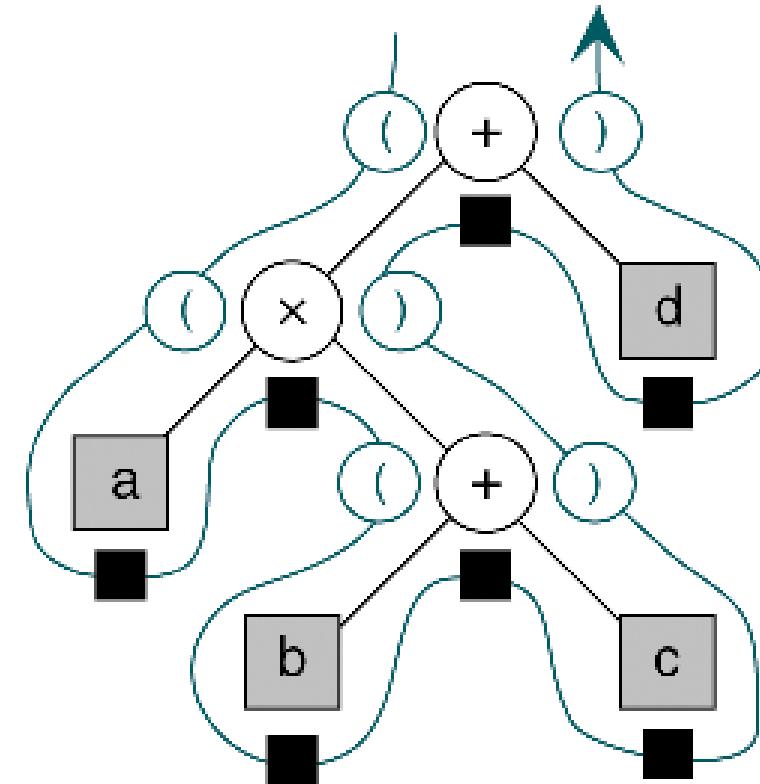
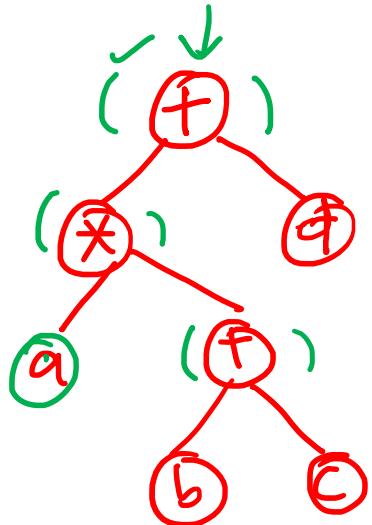
$$((a \times (b + c)) + d)$$


FIGURE 6-16 Infix Traversal of an Expression Tree

ALGORITHM 6-6 Infix Expression Tree Traversal

$((a * (b + c)) + d)$



Algorithm infix (tree)

Print the infix expression for an expression tree.

Pre tree is a pointer to an expression tree

Post the infix expression has been printed

```
→ 1 if (tree not empty)
    1 if (tree token is an operand) №      LNR
        1 print (tree-token)
    2 else
        1 print (open parenthesis)    $((a * (b + c)) + d)$ 
        2 infix (tree left subtree)
        3 print (tree token)
        4 infix (tree right subtree)
        5 print (close parenthesis)
    3 end if
2 end if
end infix
```

$\frac{+}{\text{ }} \quad \frac{+}{\text{ }}$

ALGORITHM 6.7 Postfix Traversal of an Expression Tree

Algorithm postfix (tree)

Print the postfix expression for an expression tree.

Pre tree is a pointer to an expression tree

Post the postfix expression has been printed

→ 1 if (tree not empty)

1 postfix (tree left subtree) \leftarrow LRN

2 postfix (tree right subtree) \leftarrow

3 print (tree token)

2 end if

end postfix

root | L(R)
| a |
key

ALGORITHM 6-8 Prefix Traversal of an Expression Tree

Algorithm prefix (tree)

Print the prefix expression for an expression tree.

Pre tree is a pointer to an expression tree

Post the prefix expression has been printed

```
1 if (tree not empty)
    1 print (tree token)           NLR
    2 prefix (tree left subtree)
    3 prefix (tree right subtree)
2 end if
end prefix
```



Binary Search Trees.

The ordered nodes.





Objectives.



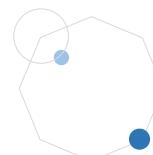
Create and implement binary search trees



Understand the operation of the binary search tree ADT



Write application programs using the binary search tree ADT





Basic Concepts.

- **Binary search trees** provide an excellent structure for searching a list and at the same time for inserting and deleting data into the list.
- A **binary search tree (BST)** is a **binary tree** with following properties:
 - All items in the **left of the tree** are **less than the root**.
 - All items in the **right subtree** are **greater than or equal to the root**.
 - **Each subtree** is itself a **binary search tree**.

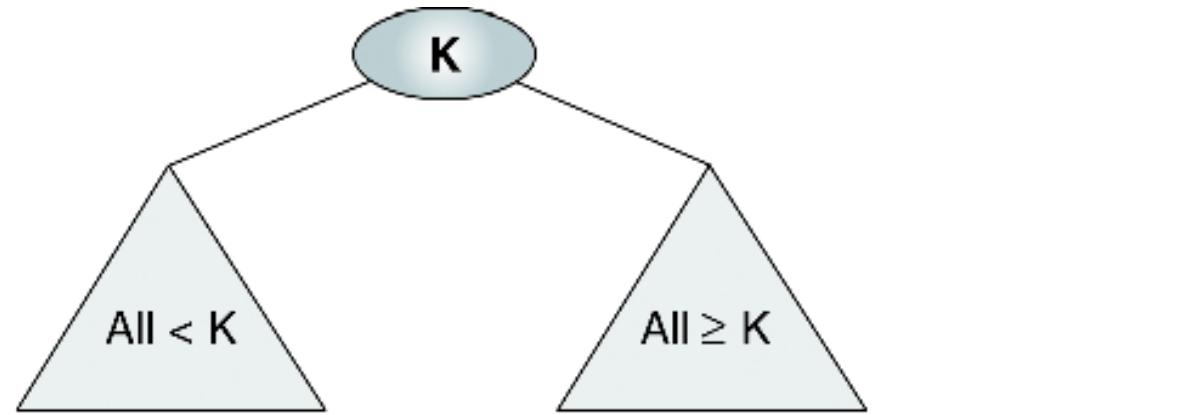


FIGURE 7-1 Binary Search Tree

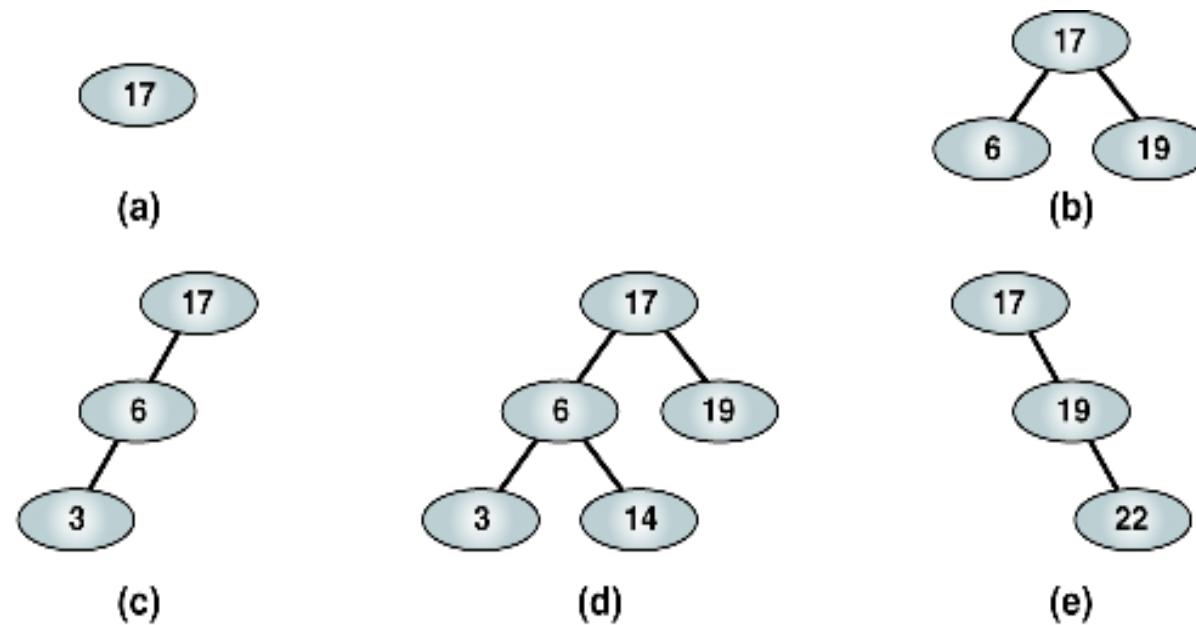
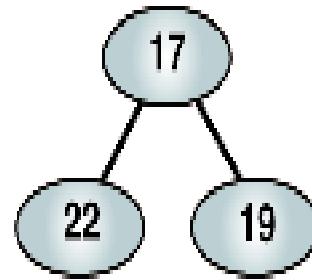
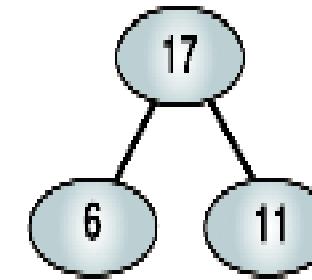


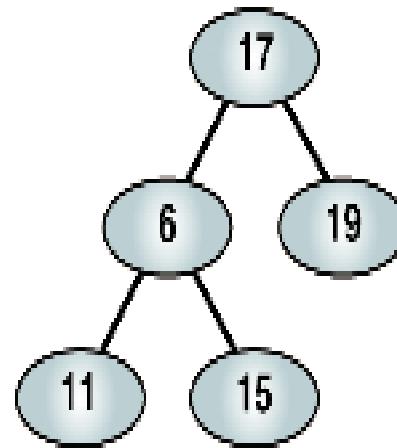
FIGURE 7-2 Valid Binary Search Trees



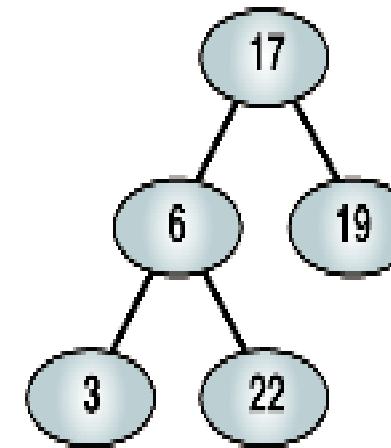
(a)



(b)



(c)



(d)

FIGURE 7-3 Invalid Binary Search Trees



BST OPERATIONS.

There are *four* basic BST operations:

- **Traversal**
- **Search**
- **Insert, and**
- **Delete.**

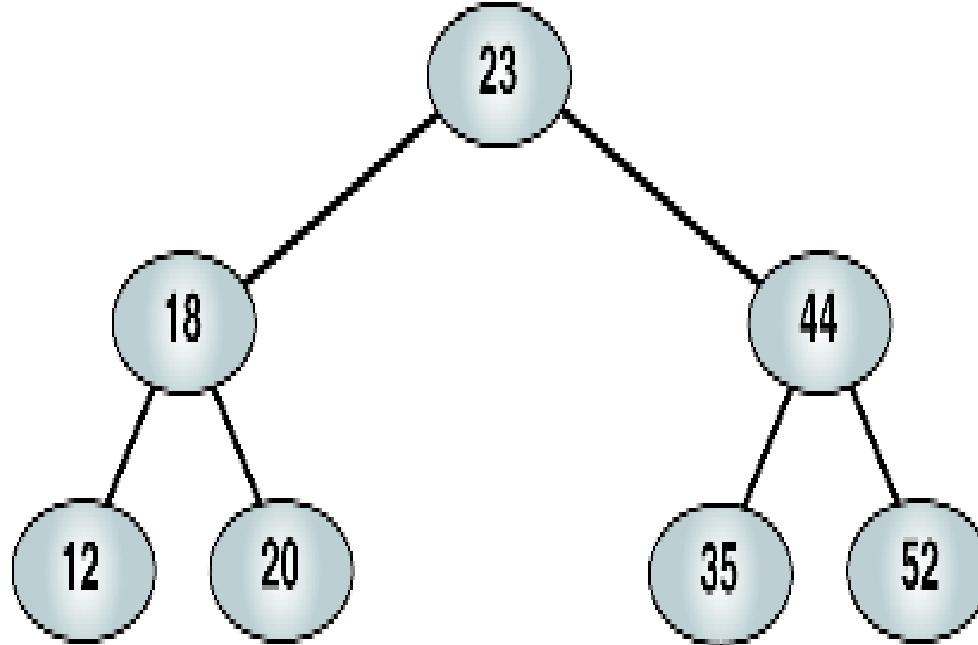


FIGURE 7-4 Example of a Binary Search Tree

Preorder Traversal : 23 18 12 20 44 35 52

Inorder Traversal: 12 18 20 23 35 44 52

Postorder Traversal: 12 20 8 35 52 44 23



SEARCH OPERATIONS.

Three search algorithms:

- find the smallest node
- find the largest node
- find a requested node (BST search).

ALGORITHM 7-1 Find Smallest Node in a BST

Find the smallest node

Algorithm findSmallestBST (root)

This algorithm finds the smallest node in a BST.

Pre root is a pointer to a nonempty BST or subtree

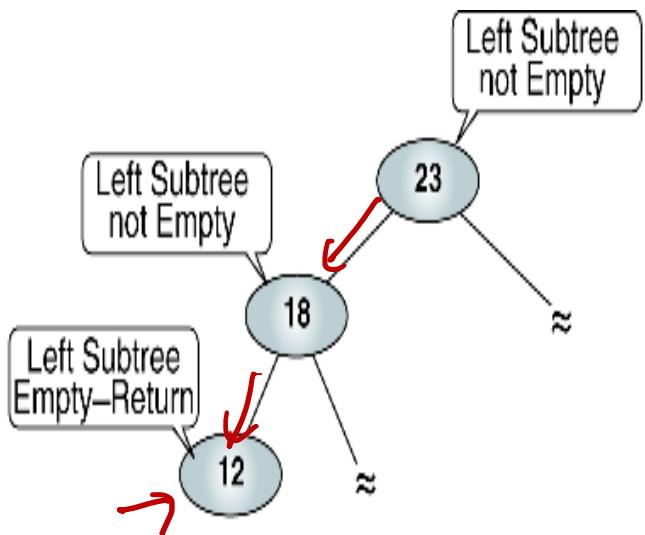
Return address of smallest node

1 if (left subtree empty)
1 return (root)

2 end if

3 return findSmallestBST (left subtree)

end findSmallestBST

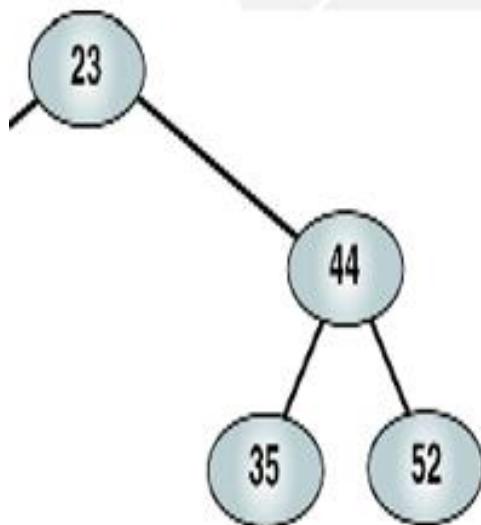


- Node with the smallest value (12) is the ***far-left leaf node*** in the tree.
- The find smallest node operation, therefore, simply follows the left branches until a leaf is reached..

FIGURE 7-5 Find Smallest Node in a BST

ALGORITHM 7-2 Find Largest Node in a BST

- Node with the largest value (52) is the ***far-right leaf node*** in the tree.
- This operation, therefore, simply follows the right branches until a leaf is reached.



Algorithm findLargestBST (root)

This algorithm finds the largest node in a BST.

Pre root is a pointer to a nonempty BST or subtree
Return address of largest node returned

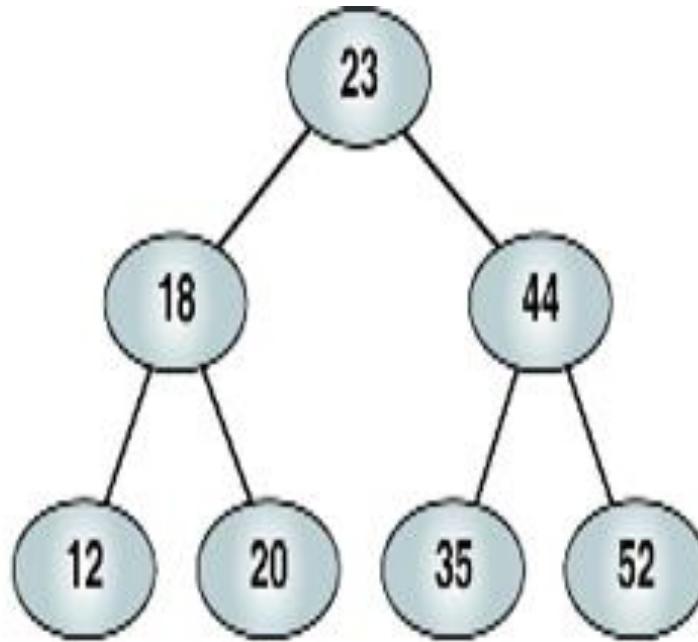
```
1 if (right subtree empty)
    1   return (root)
2 end if
3 return findLargestBST (right subtree)
end findLargestBST
```

Find the largest node.

binary search algorithm

Sequenced array

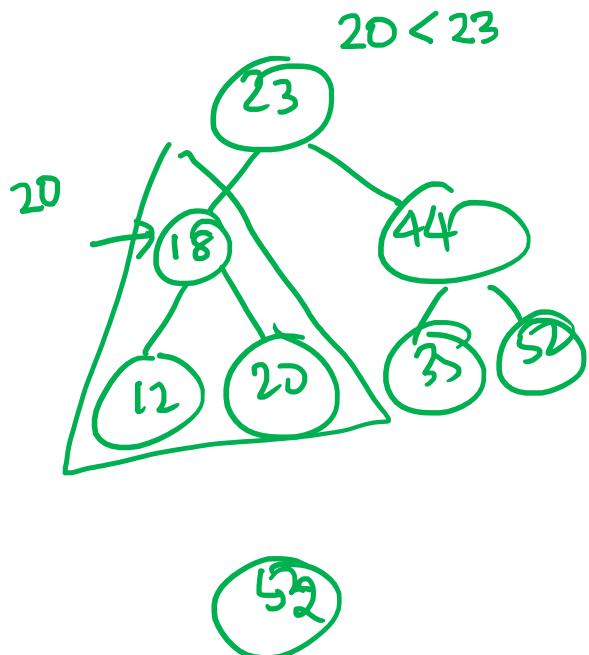
12	18	20	23	35	44	52
----	----	----	----	----	----	----



Search points in binary search

FIGURE 7-6 BST and the Binary Search

ALGORITHM 7-3 Search BST



```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
Pre    root is the root to a binary tree or subtree
           targetKey is the key value requested
Return the node address if the value is found
           null if the node is not in the tree
1  if (empty tree)
    Not found
    1 return null
2 end if
3 if (targetKey < root)
    1 return searchBST (left subtree, targetKey)
4 else if (targetKey > root)
    1 return searchBST (right subtree, targetKey)
5 else
    Found target key
    1 return root
6 end if
end searchBST
```

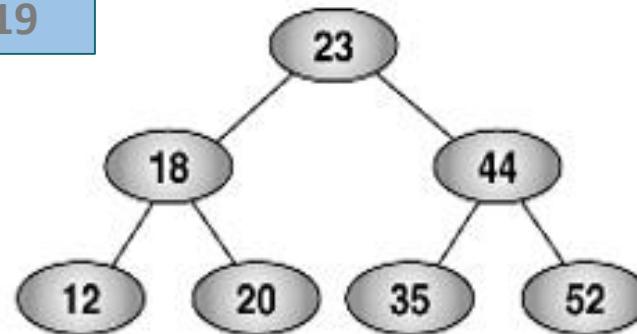
20 < root
20 > root
20



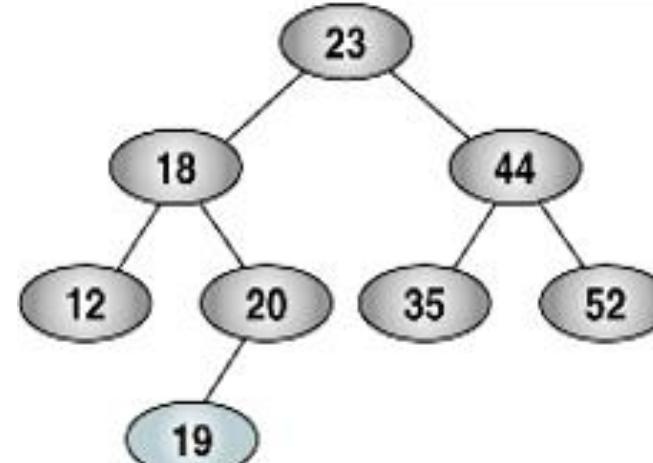
INSERTION OPERATIONS.

All BST insertions take place at a leaf or a leaflike node.

Inserting 19

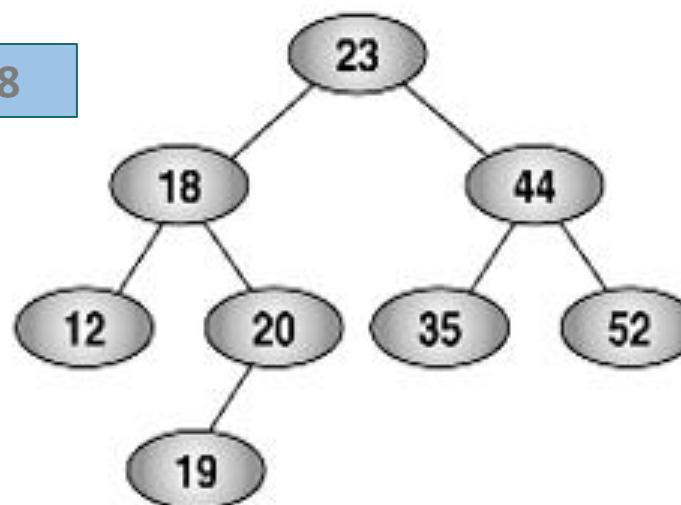


(a) Before inserting 19

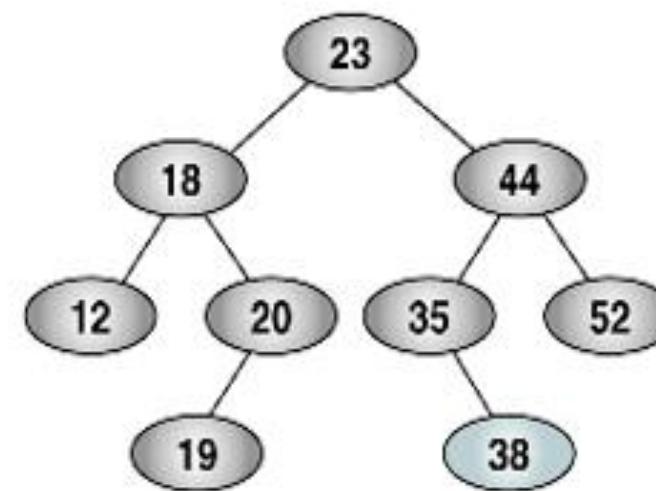


(b) After inserting 19

Inserting 38



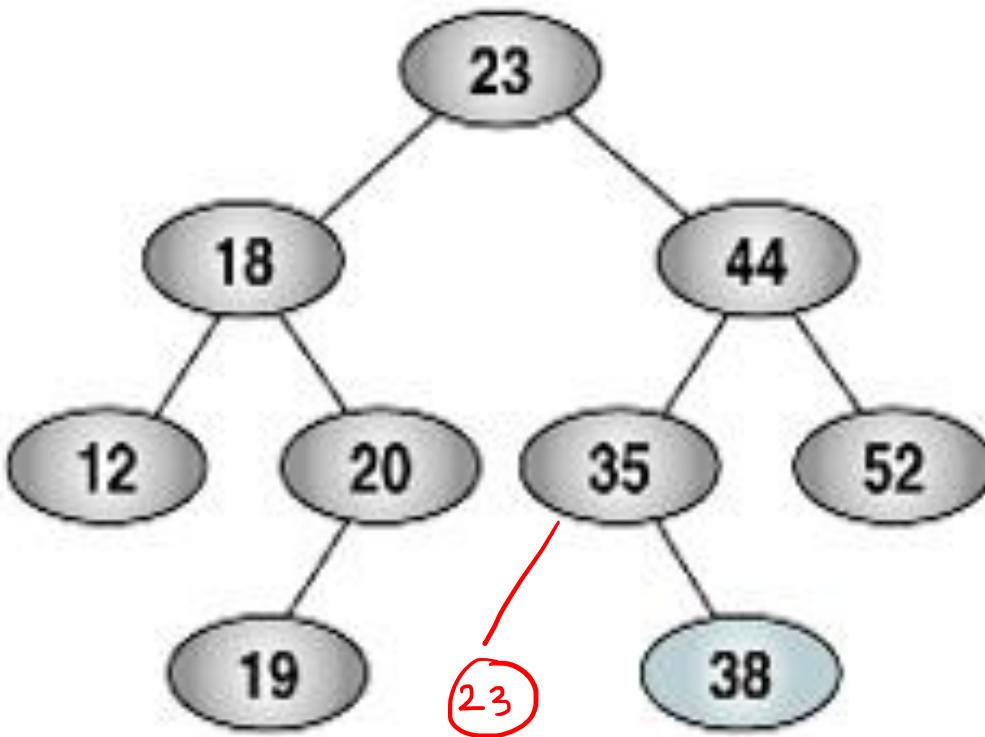
(c) Before inserting 38



(d) After inserting 38

FIGURE 7-8 BST Insertion

Inserting 23 ?



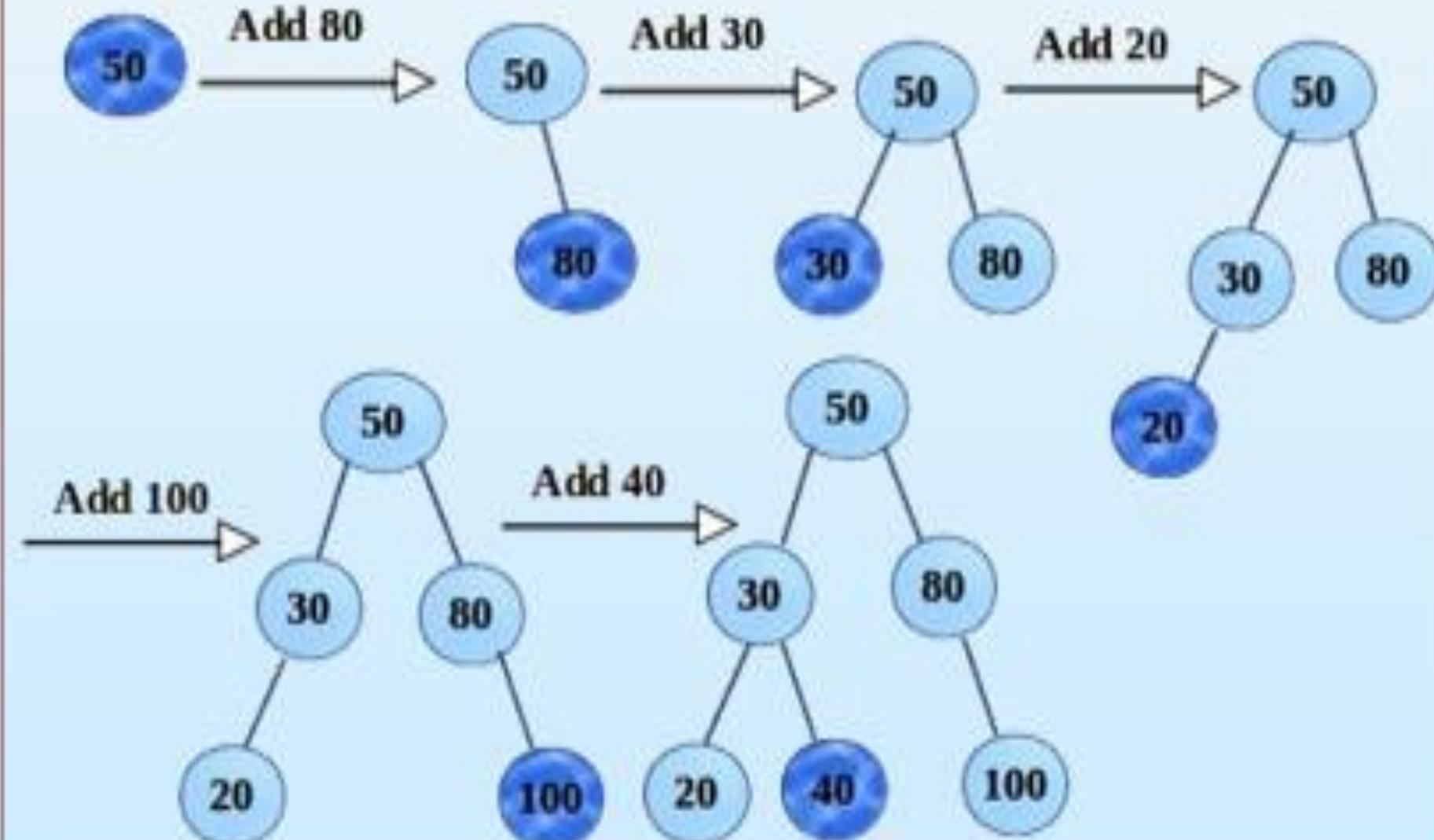
(d) After inserting 38

then insert 23

FIGURE 7-8 BST Insertion

50 ✓
80 ✓
30 ✓
20 ✓
100 ✓
40 ✓

example 2



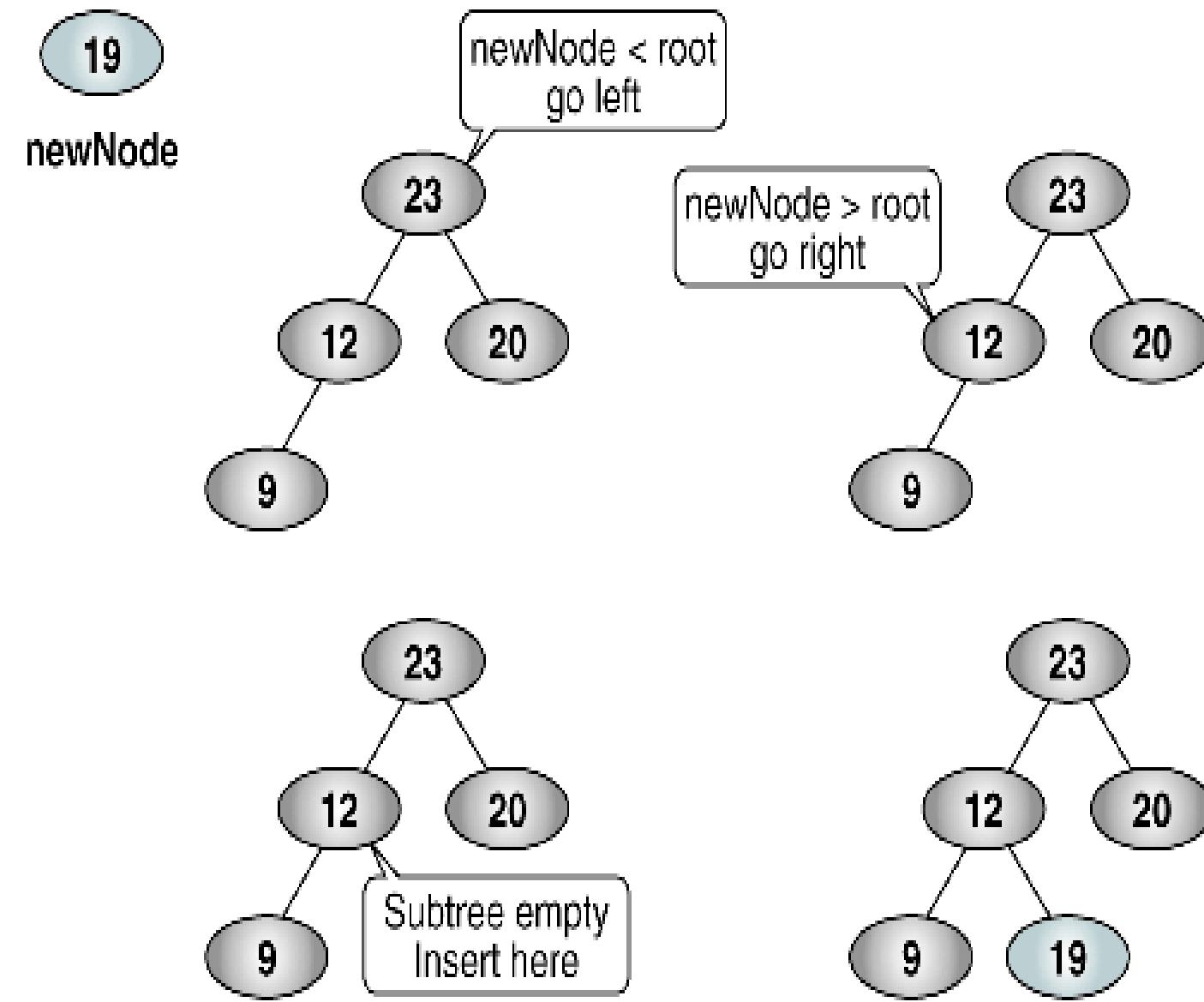


FIGURE 7-9 Trace of Recursive BST Insert

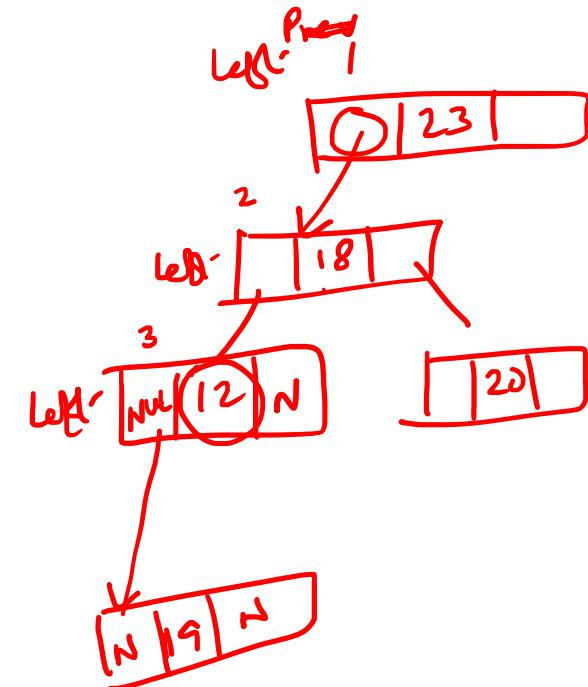
ALGORITHM 7-4 Add Node to BST

23

19

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
Pre    root is address of current node in a BST
        newNode is address of node containing data
Post   newNode inserted into the tree
Return address of potential new tree root
1 if (empty tree)
  1 set root to newNode
  2 return newNode
2 end if
```

Leaf
node



(1) Implementing

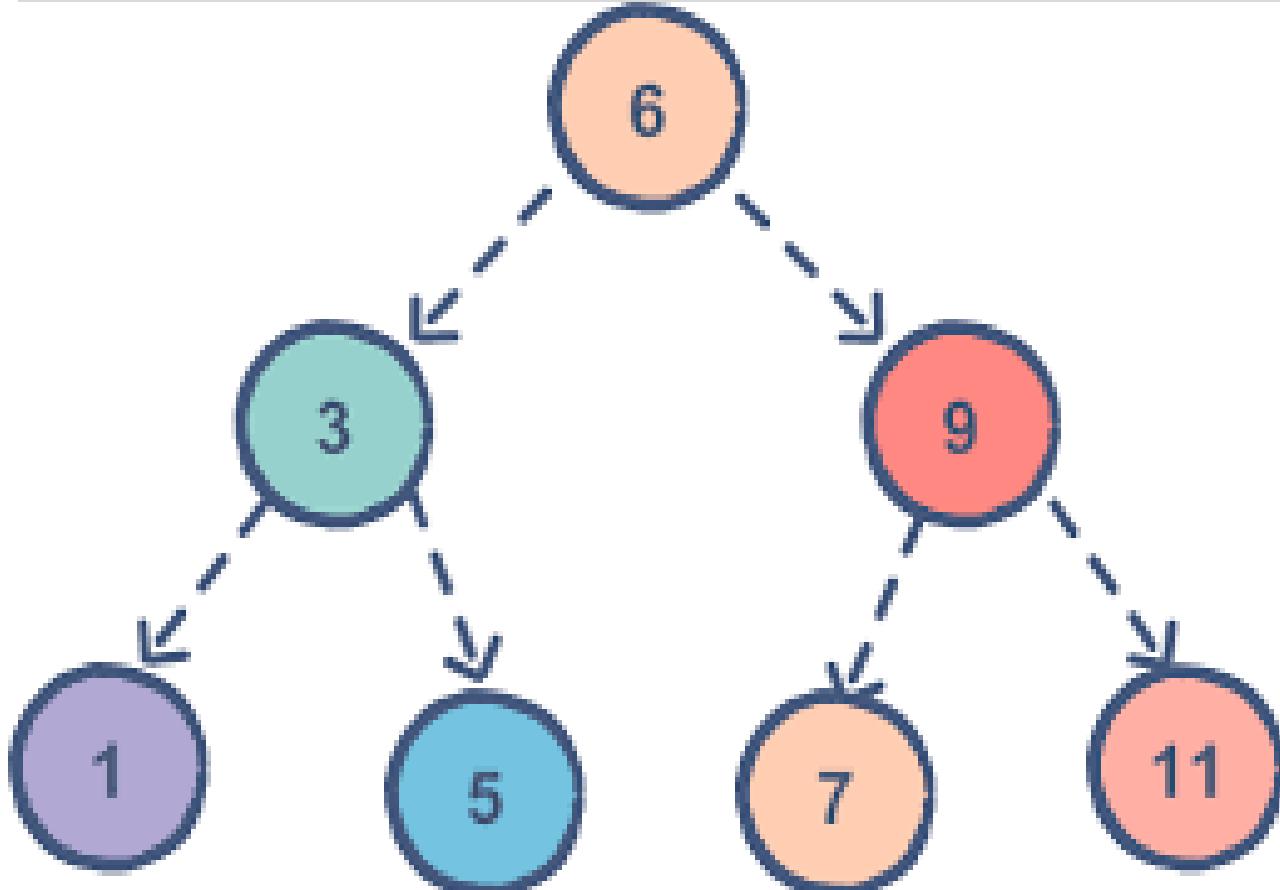
19 < 23
=

19 < 23
=

ALGORITHM 7-4 Add Node to BST (continued)

```
Locate null subtree for insertion
3 if (newNode < root) 19 < 23
  1 return addBST (left subtree, newNode)
4 else
  1 return addBST (right subtree, newNode)
5 end if
end addBST
```

18, 19

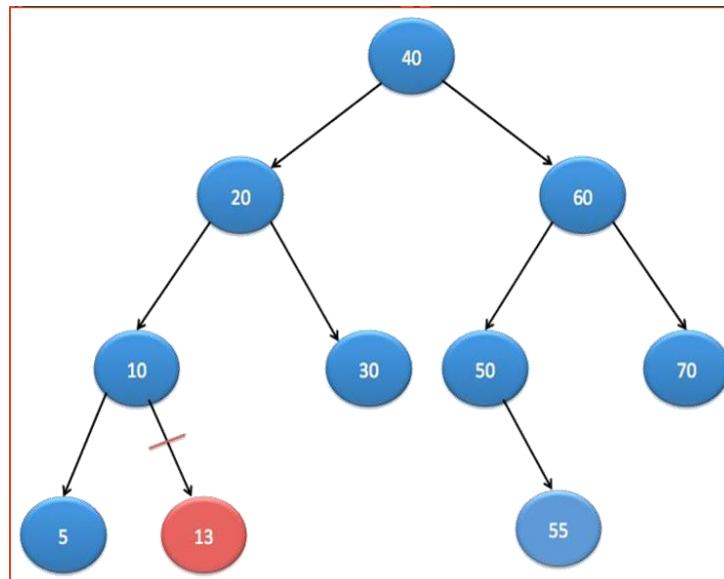


An example of a binary search tree

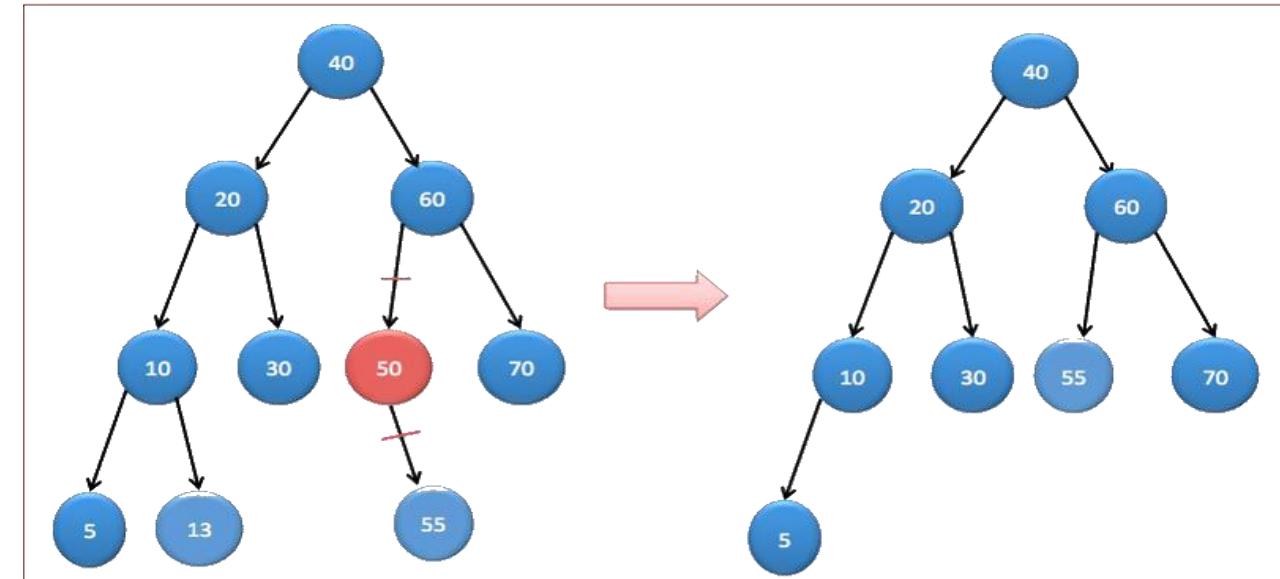
Break.

Deletion.

- To **delete a node** from a binary search tree, we must **first locate it**.
- There are **four cases** for deletion:
 1. The node to be deleted has **no children**, which means ***delete the node/ leaf node***.
 2. The node to be deleted has **only a right subtree**. Delete the node and **attach the right subtree** to the deleted node's parent.



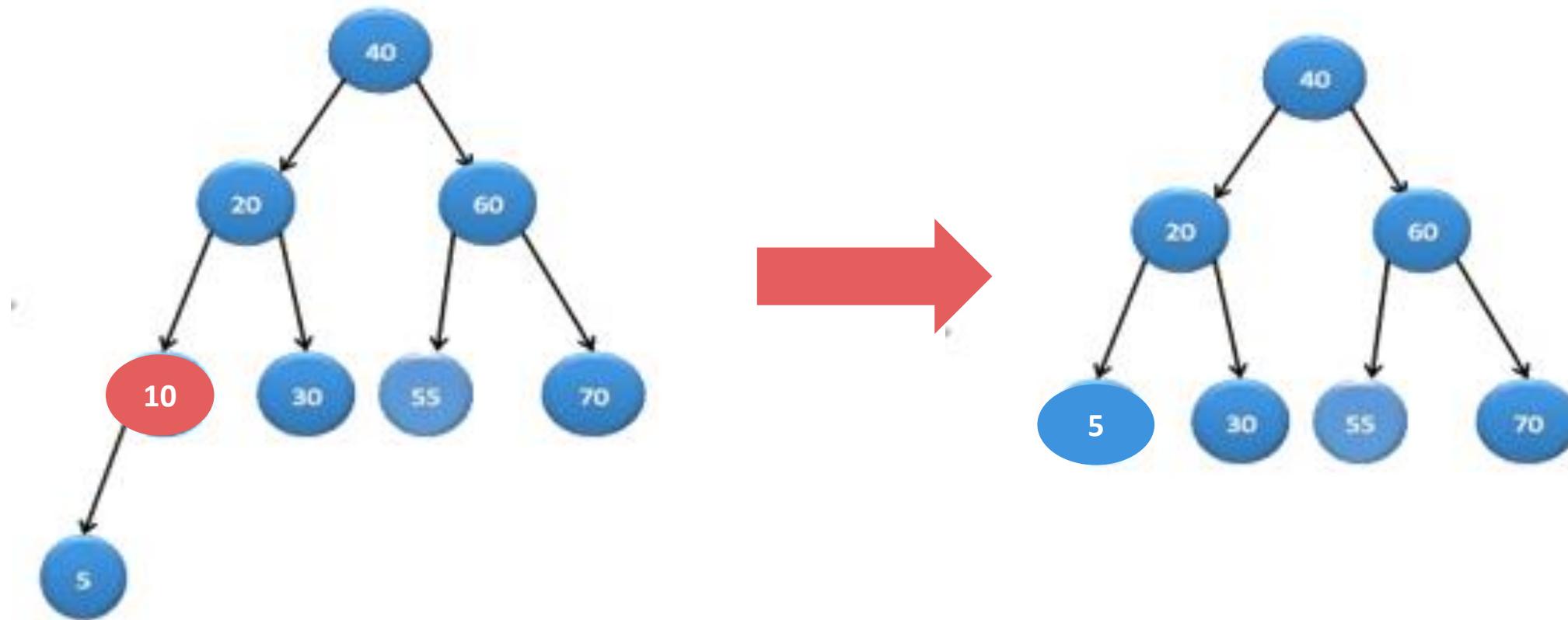
No children



Node with right subtree

Deletion.

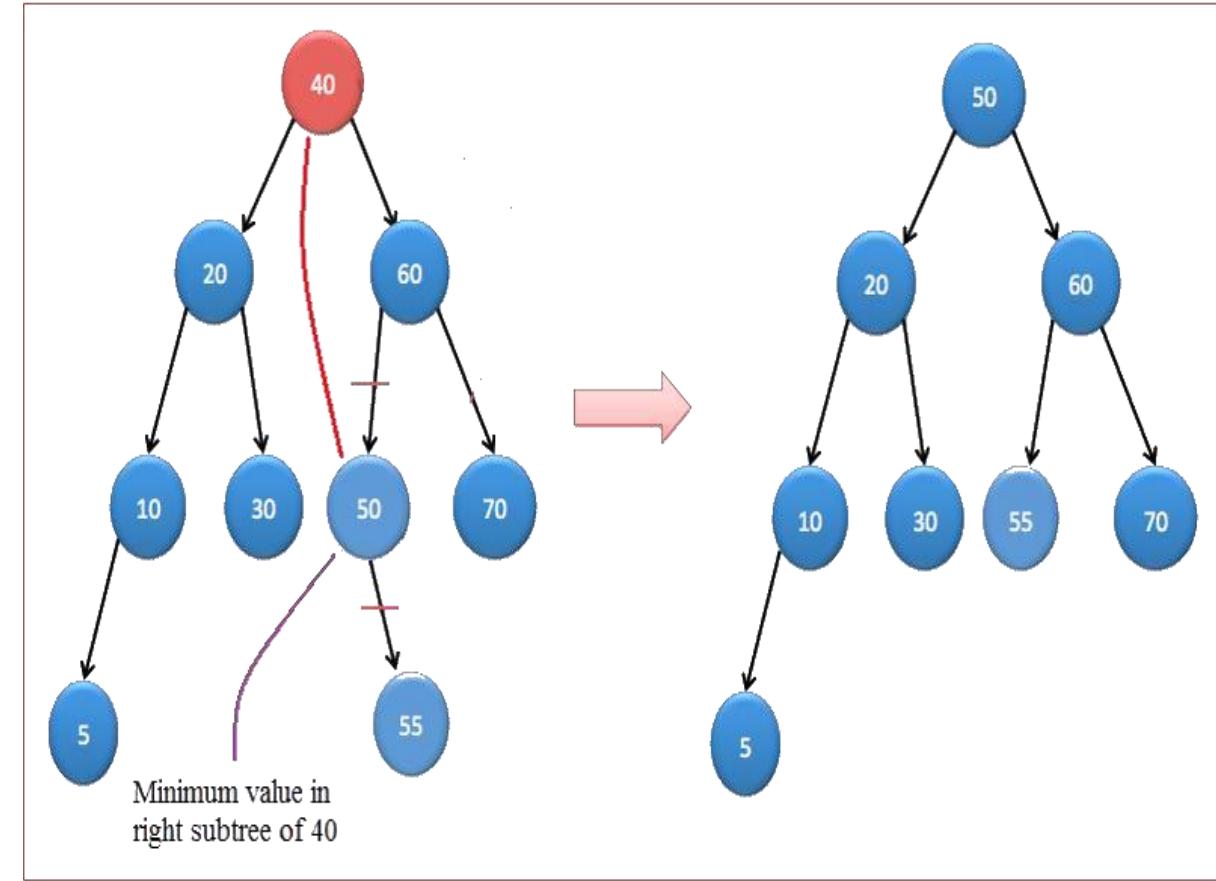
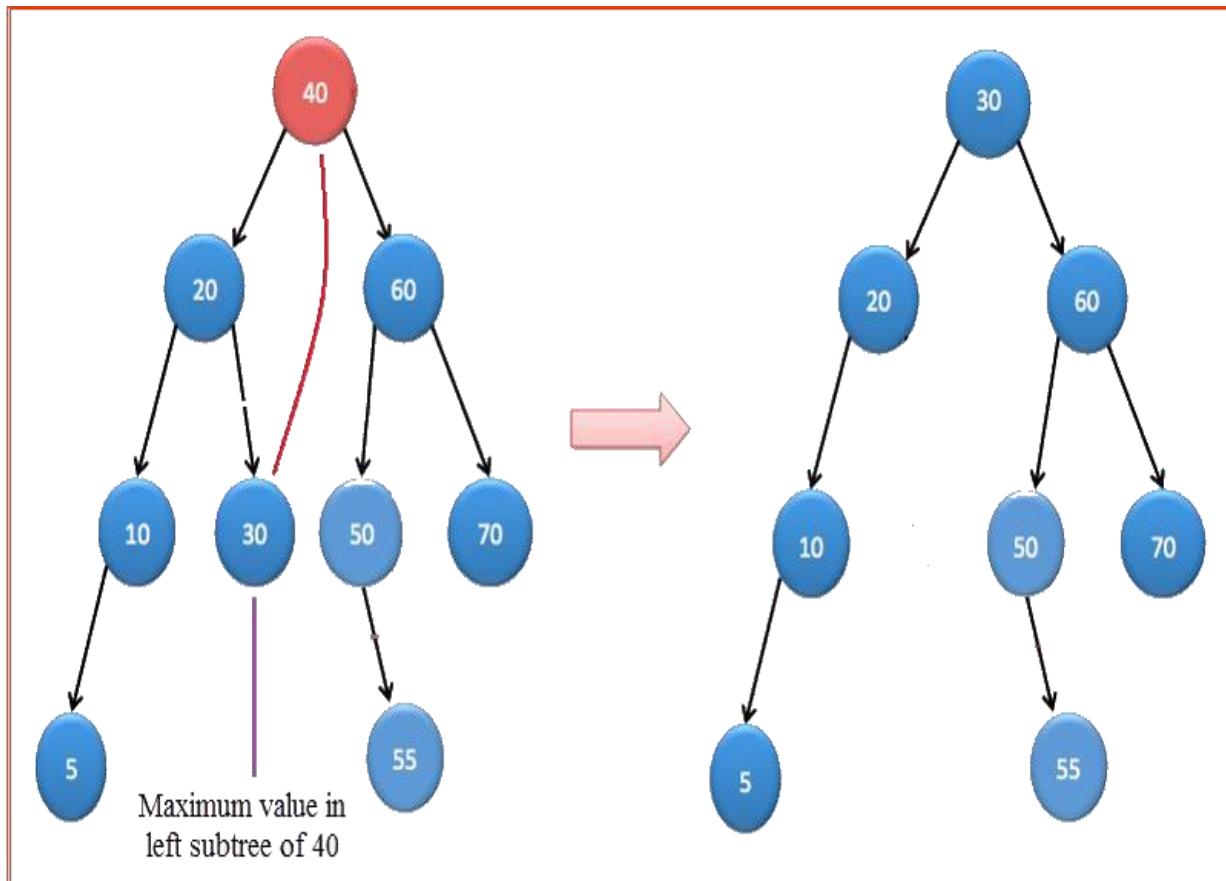
3. The node to be deleted has **only a left subtree**. Delete the node and attach the left subtree to the deleted node's parent.



Node with left subtree

Deletion.

4. The node to deleted has **two subtrees**. Tree becomes unbalanced.



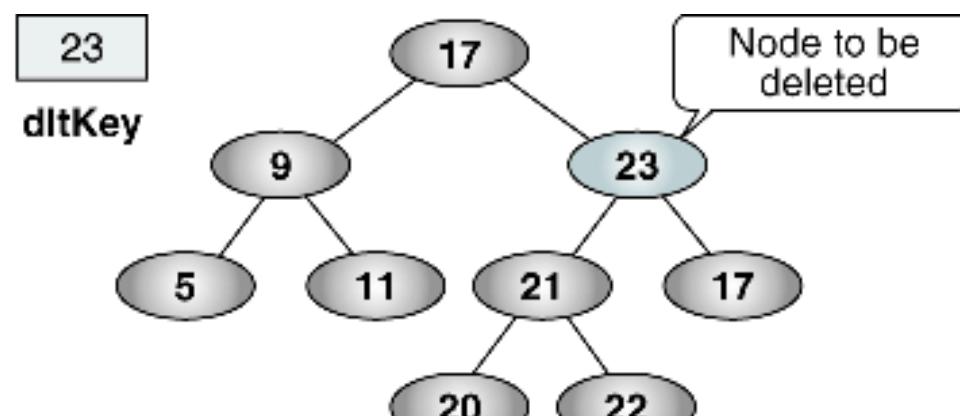
Node with two subtrees

Deletion of node with two subtrees – two ways

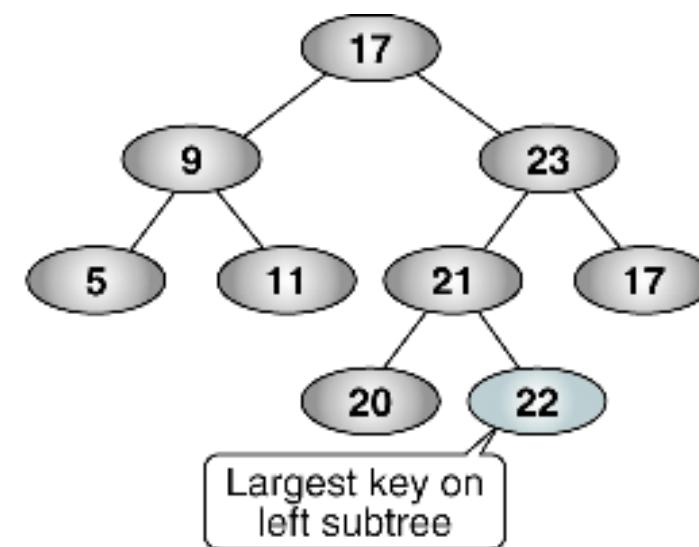
- When the node to deleted has two subtrees.
- Find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.
- Do the **inorder traversal** -> **5, 10, 20, 30, 40, 50, 55, 60, 70**.
- Find inorder predecessor of root, replace 40 by 30.

Or

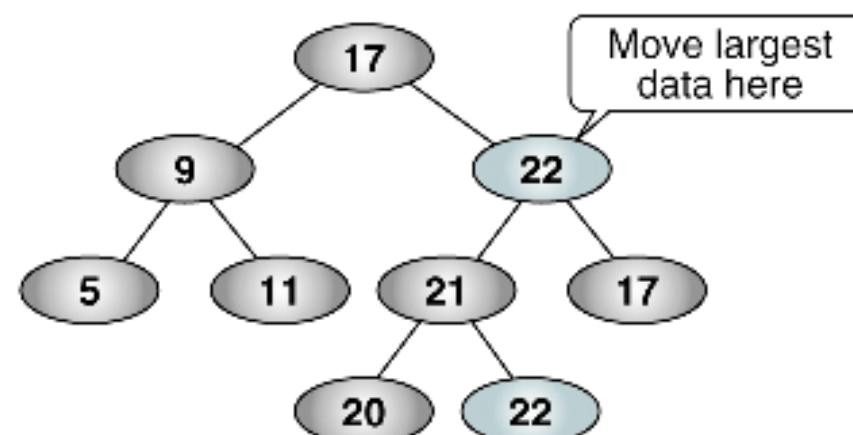
- Find the smallest node in the deleted node's right subtree and move its data to replace the deleted node's data.
- Do the **inorder traversal** -> **5, 10, 20, 30, 40, 50, 55, 60, 70**.
- Find inorder successor of root, replace 40 by 50.



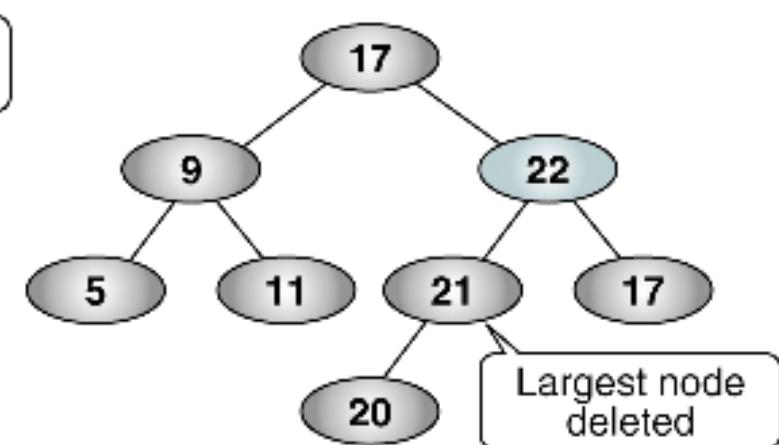
(a) Find dltKey



(b) Find largest



(c) Move largest data



(d) Delete largest node

FIGURE 7-10 Delete BST Test Cases

ALGORITHM 7-5 Delete Node from BST

```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
    Pre    root is reference to node to be deleted
           dltKey is key of node to be deleted
    Post   node deleted
           if dltKey not found, root unchanged
           Return true if node deleted, false if not found
1  if (empty tree)
   1  return false
2 end if
3 if (dltKey < root)
   1  return deleteBST (left subtree, dltKey)
4 else if (dltKey > root)
   1  return deleteBST (right subtree, dltKey)
5 else
    Delete node found--test for leaf node
    1 If (no left subtree)
       1 make right subtree the root
       2 return true
```

ALGORITHM 7-5 Delete Node from BST (continued)

```
2 else if (no right subtree)
    1 make left subtree the root
    2 return true
3 else
    Node to be deleted not a leaf. Find largest node on
    left subtree.
    1 save root in deleteNode
    2 set largest to largestBST (left subtree)
    3 move data in largest to deleteNode
    4 return deleteBST (left subtree of deleteNode,
                        key of largest
4 end if
6 end if
end deleteBST
```

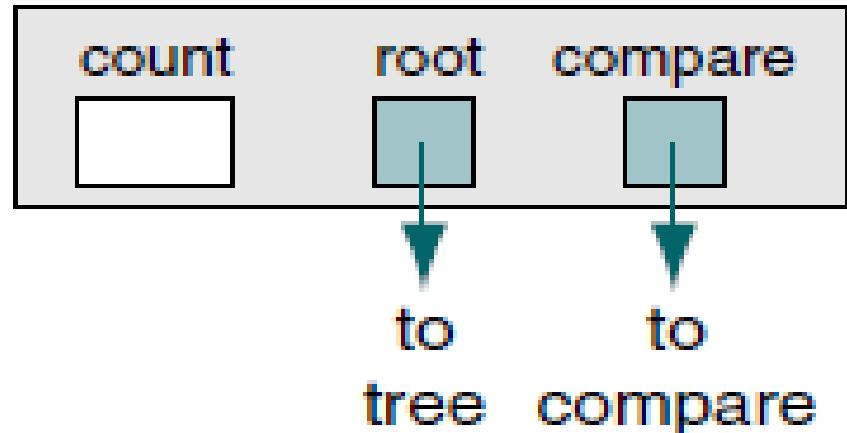
Binary Search Tree ADT.

Head: Count, a root pointer, and the address of the compare function needed to search the list.

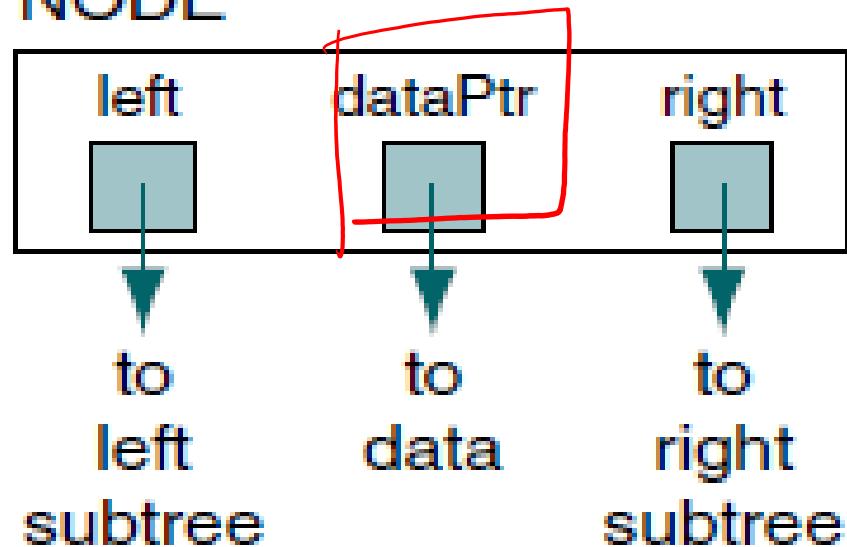
Data Structure

Node: data pointer and two self-referential pointers to the left and right subtrees..

BST_TREE



NODE



BST APPLICATIONS.

Integer Application

Student List Application

Integer Application:

Reading integers (values) from the keyboard and constructing BST

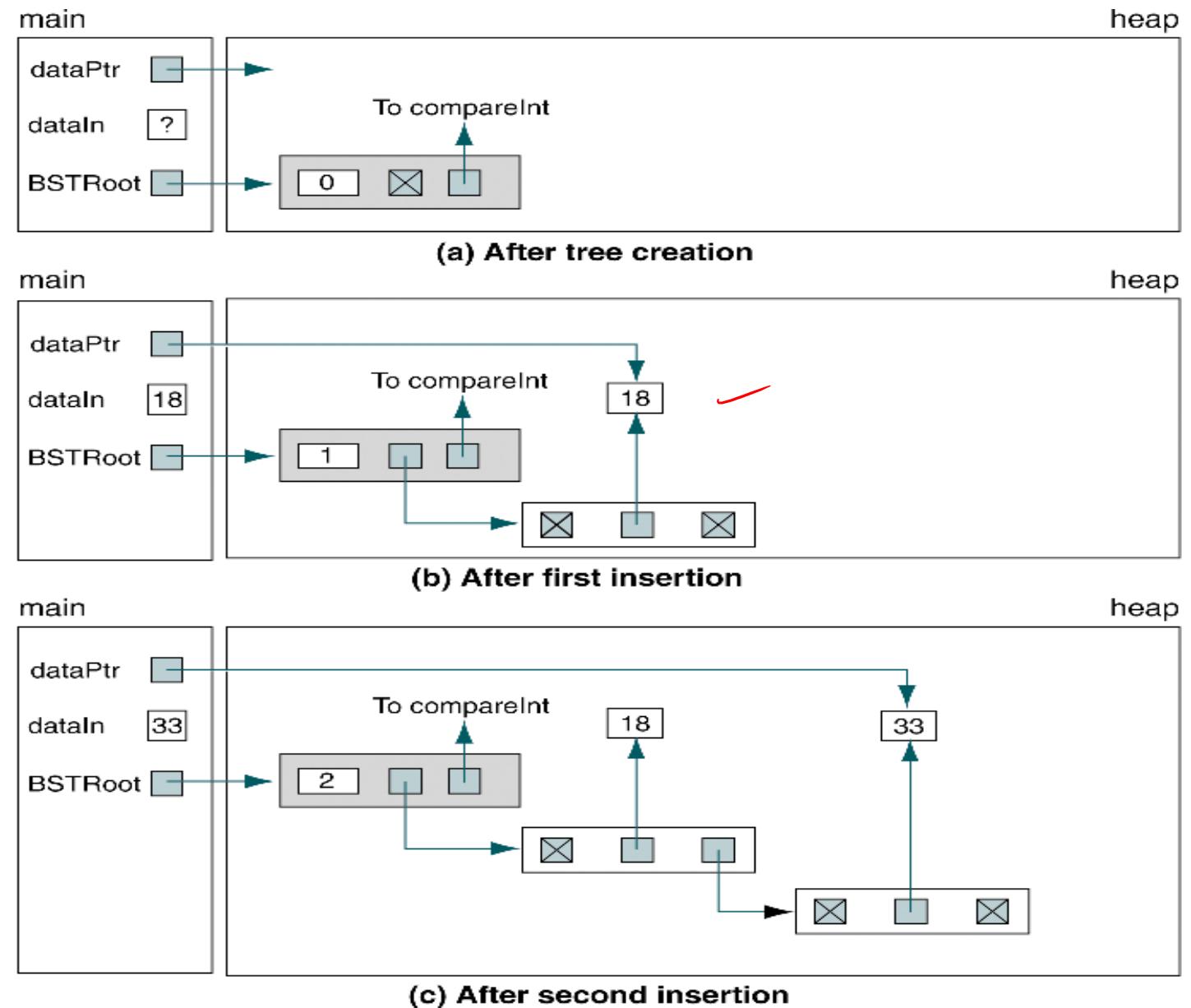


FIGURE 7-13 Insertions into a BST

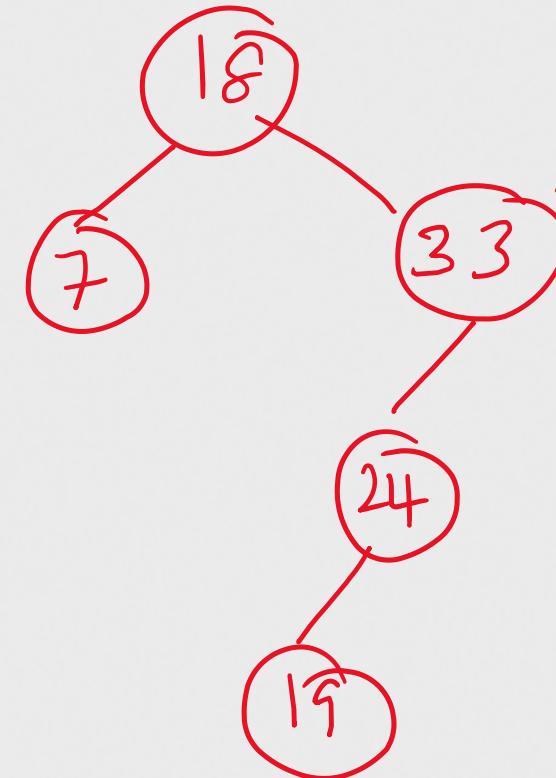
PROGRAM 7-16

```
Results:  
Begin BST Demonstation  
Enter a list of positive integers;  
Enter a negative number to stop.  
Enter a number: 18
```

```
Enter a number: 33 ✓  
Enter a number: 7 ✓  
Enter a number: 24 ✓  
Enter a number: 19  
Enter a number: -1
```

BST contains:

7
18
19
24
33



End BST Demonstration

STUDENT APPLICATION.

Three pieces of data: Student's ID, the student's name, and the student's grade-point average.

- Students are added and deleted from the keyboard.
- They can be retrieved individually or as a list

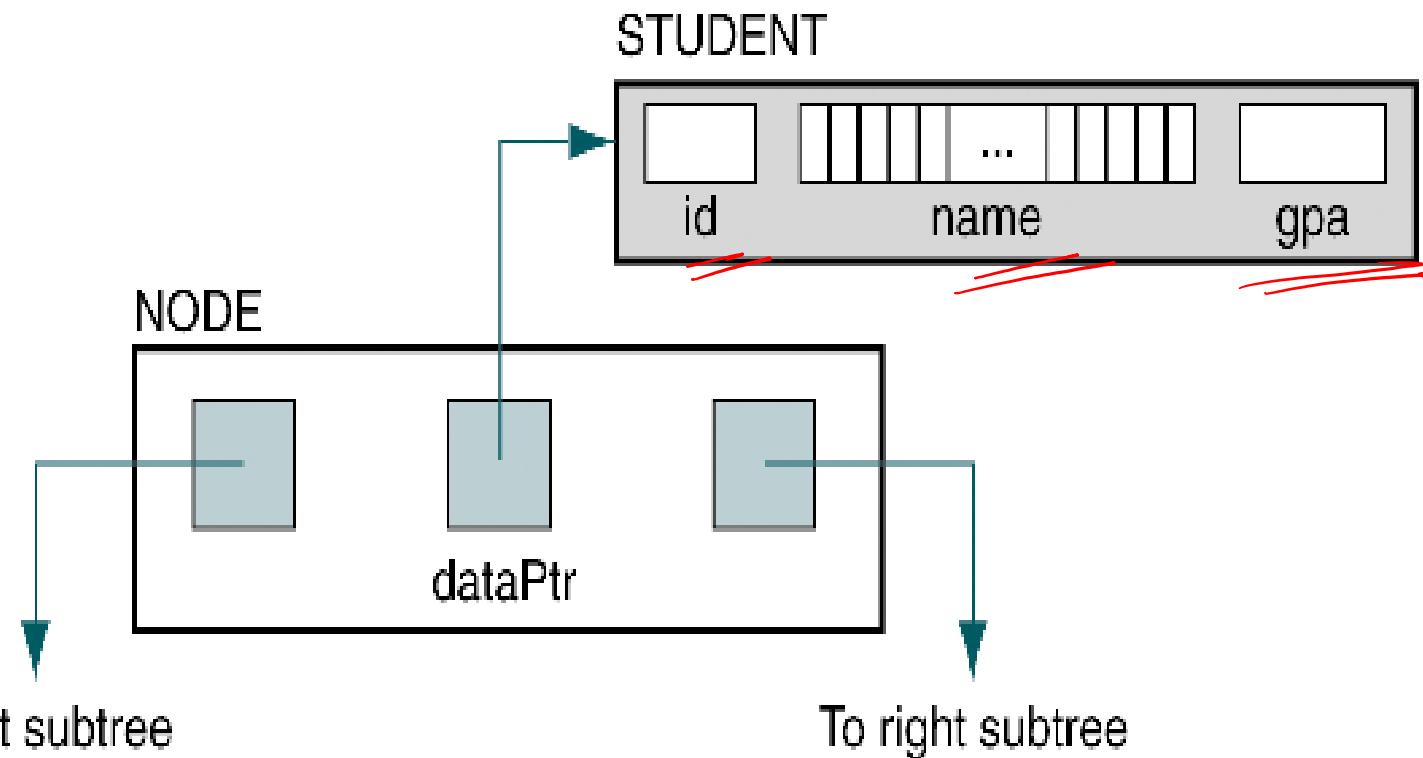


FIGURE 7-14 Student Data in ADT

GENERAL TREES.

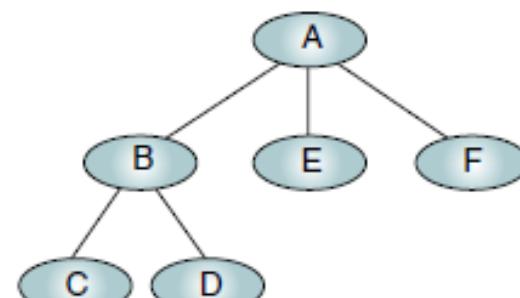
- A *general tree* is a tree in which each node can have an **unlimited outdegree**.
- Each node may have **as many children** as is necessary to satisfy its requirements.

INSERTION INTO GENERAL TREES.

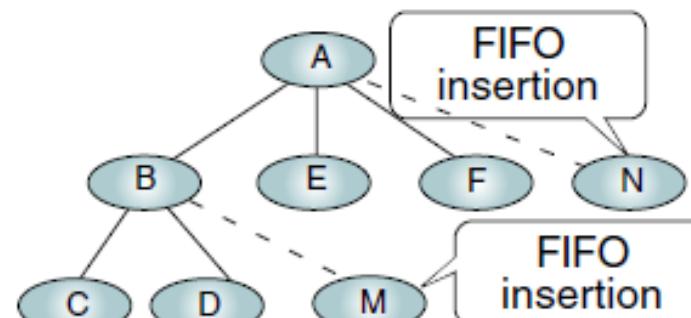
- To insert a node into a general tree, *the user must supply the parent of the node.*
- Given the parent, **three** different rules may be used:
 - (1) **first in–first out (FIFO) insertion,**
 - (2) **last in–first out (LIFO) insertion, and**
 - (3) **key-sequenced insertion**

FIFO INSERTION.

- When using FIFO insertion, **Insert the nodes at the end of the sibling list**, much as we insert a new node at the **rear of a queue**.
- When the list is then processed, the siblings are processed in FIFO order. FIFO order is used when the application requires that the data be processed in the order in which they were input.
- Given its parent as A, node N has been inserted into level 1 after node F; and, given its parent as B, node M has been inserted at level 2 after node D.



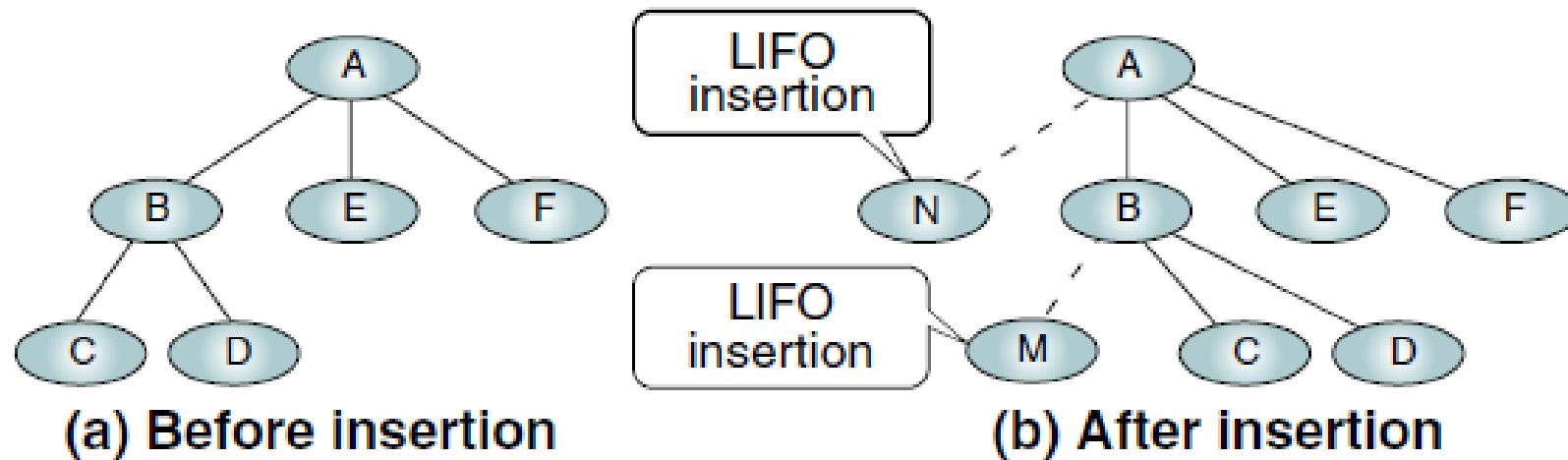
(a) Before insertion



(b) After insertion

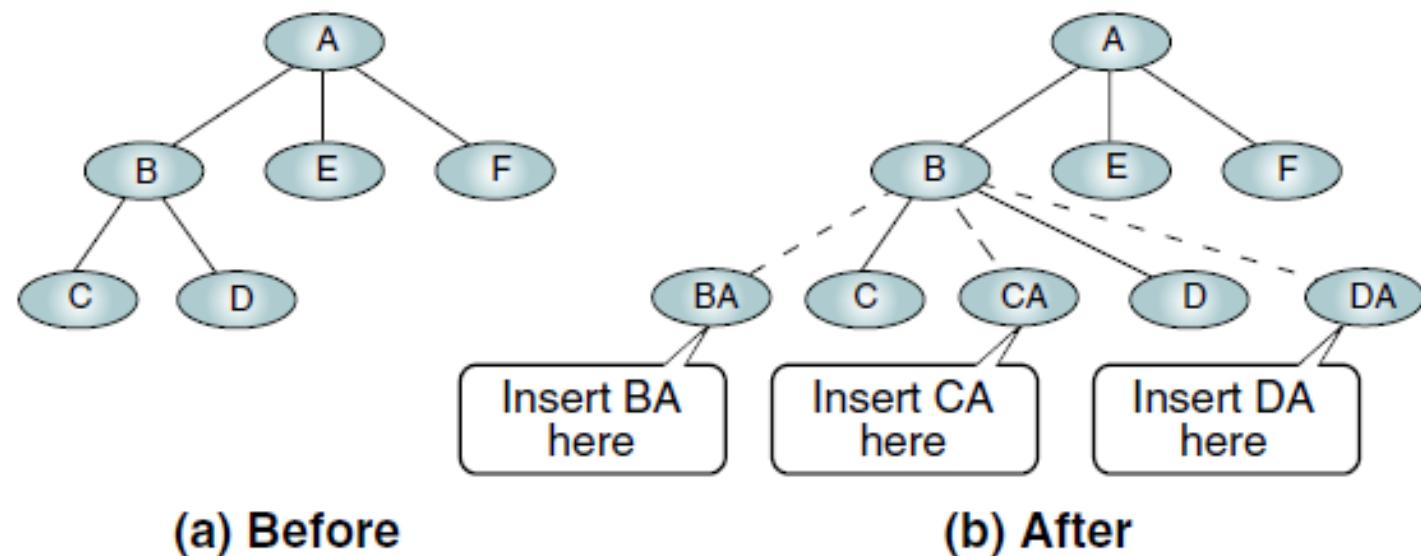
LIFO INSERTION.

- To process sibling lists in the opposite order in which they were created, we use LIFO insertion.
- LIFO insertion places the new node at the beginning of the sibling list.
- It is the equivalent of a stack. EXAMPLE shows the insertion points for a LIFO tree.



KEY SEQUENCED INSERTION.

- Most common of the insertion rules, key-sequenced insertion *places the new node in key sequence* among the sibling nodes.
- The logic for inserting in key sequence is similar to that for insertion into a linked list. **Starting at the parent's first child, we follow the sibling (right) pointers until we locate the correct insertion point and then build the links with the predecessors and successors** (if any). Example shows the correct key-sequenced insertion locations for several different values in a general tree.

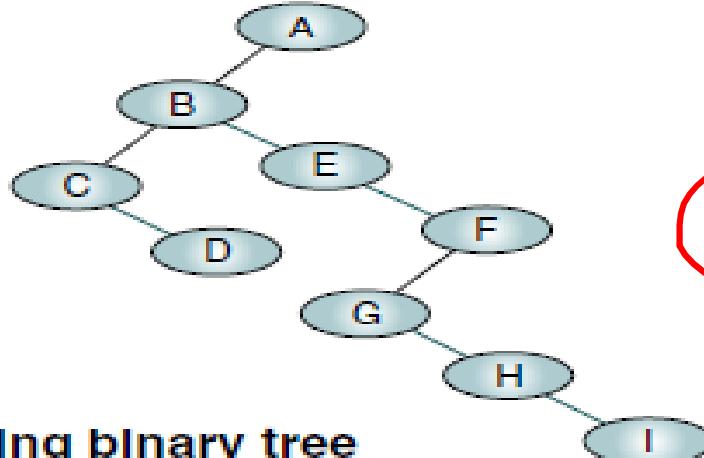
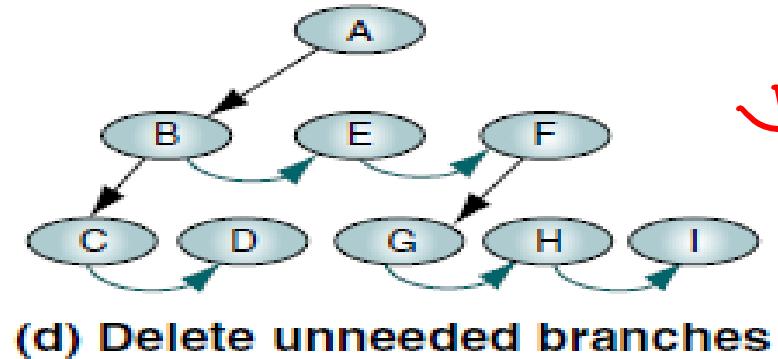
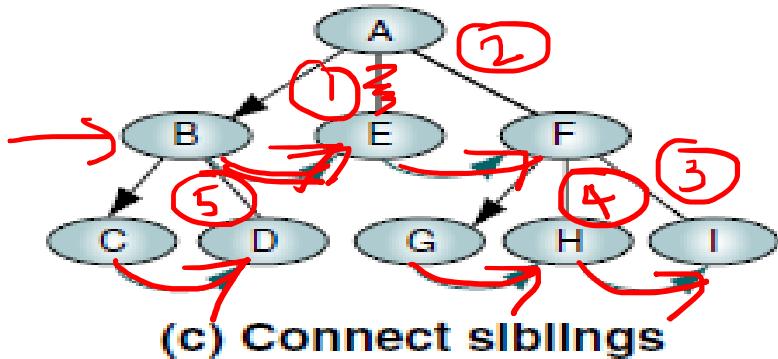
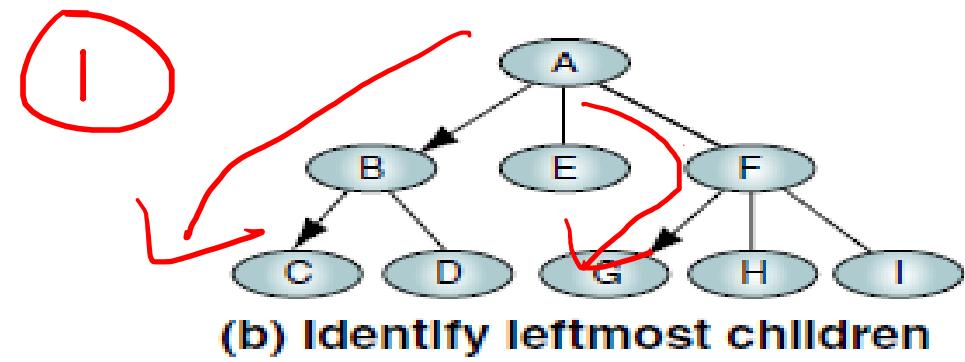
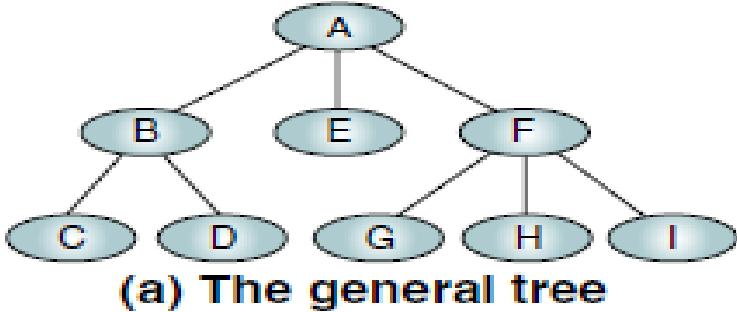


GENERAL TREE DELETIONS.

- No standard rules for general tree insertions,
- BUT **standard deletion rules are:**
 1. **A node may be deleted only if it is a leaf.** In the general tree, this means a node cannot be deleted if it has any children. If the user tries to delete a node that has children, the program provides an error message that the node cannot be deleted until its children are deleted.
 2. It is then the user's responsibility to first delete any children. As an alternative, the application could be programmed to delete the children first and then delete the requested node. If this alternative is used, it should be with a different user option, such as purge node and children, and not the simple delete node option.

CHANGING GENERAL TREES TO BINARY TREES.

- It is considerably **easier to represent binary trees in programs** than it is to represent general trees.
- Better to represent general trees using a binary tree format.
- The binary tree format can be adopted by changing the meaning of the left and right pointers. **In a general tree, two relationships: parent to child and sibling to sibling are used.** Using these two relationships, we can represent any general tree as a binary tree.



Converting General Trees to Binary Trees

SUMMARY of GENERAL TREES.

- To change a general tree to a binary tree, we identify the first child of each node, connect the siblings from left to right, and delete the connection between each parent and all children except the first.
- The three approaches for inserting data into general trees are FIFO, LIFO, and key sequenced.
- To delete a node in a general tree, we must ensure that it does not have a child.



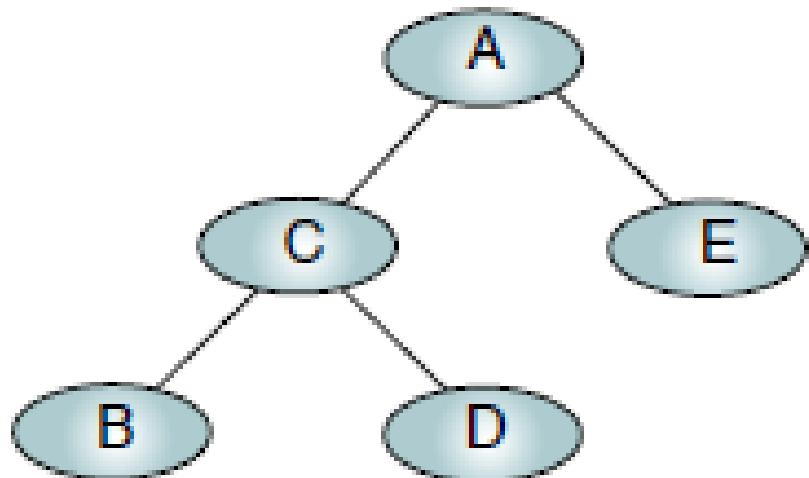
THREADED TREES.

THREADED TREES.

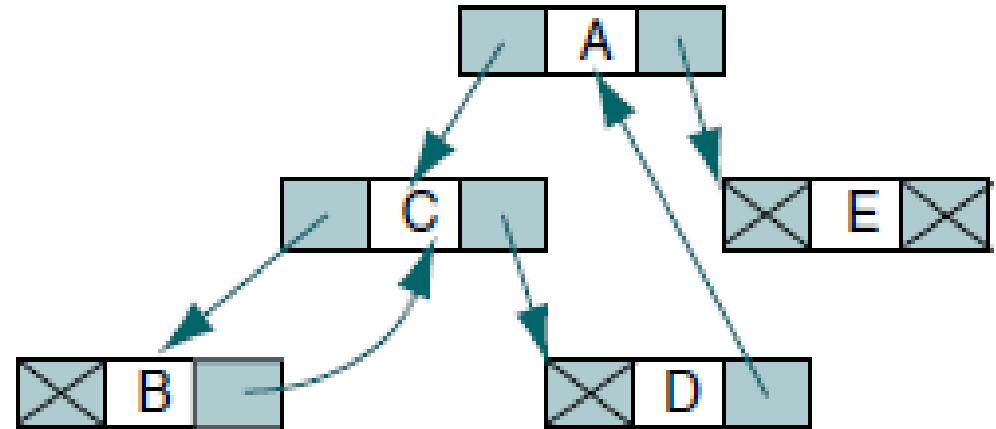
- Binary tree traversal algorithms are written using either recursion or programmer-written stacks. If the tree must be traversed frequently, **using stacks** rather than recursion may be more **efficient**.
- A third alternative is a ***threaded tree***. In a threaded tree, **null pointers are replaced with pointers to their successor nodes**.
- To **build a threaded tree** ----> first build a standard binary search tree.
- Then **traverse the tree**, changing the null right pointers to point to their successors.
- The traversal for a threaded tree is straightforward. Once you locate the far-left node, loop happens, following the thread (the right pointer) to the next node. No recursion or stack is needed. When you find a null thread (right pointer), the traversal is complete.

Inorder traversal (LNR)

THREADED TREES.



(a) Binary tree



(b) Threaded binary tree

To find the far-left leaf, **backtracking** is performed to process the right subtrees while navigating downwards. This is especially inefficient when the parent node has no right subtree.

Binary Trees

■ Threaded Binary Tree

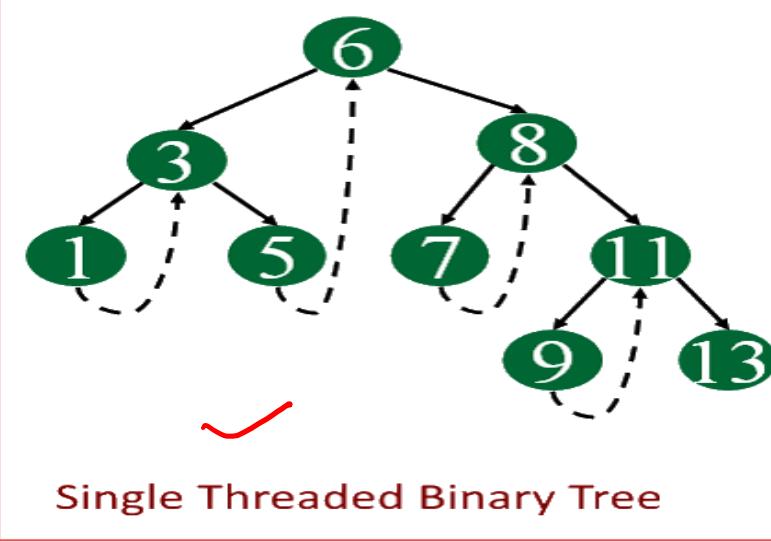
- A binary tree with n nodes has $n + 1$ null pointers

‣ Waste of space due to null pointers

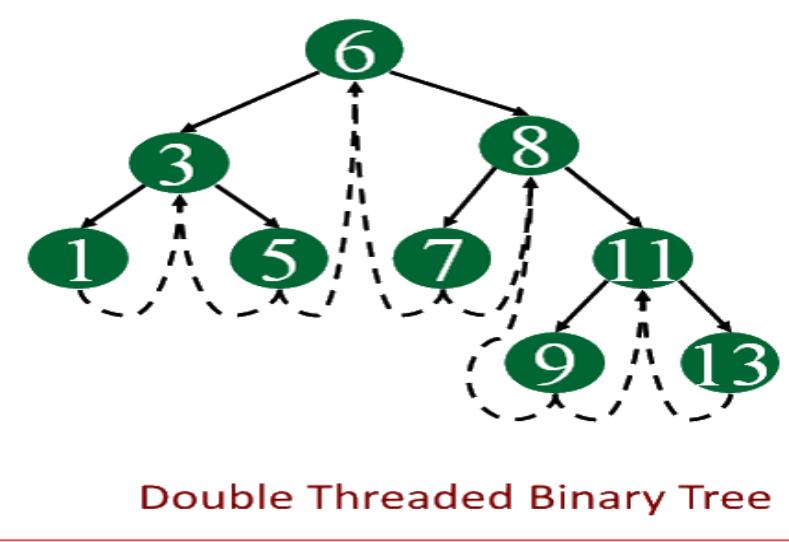
‣ Replace by threads pointing to inorder successor and/or inorder predecessor (if any)

INORDER TRAVERSAL IS:-

1 3 5 6 7 8 9 11 13



Single Threaded Binary Tree



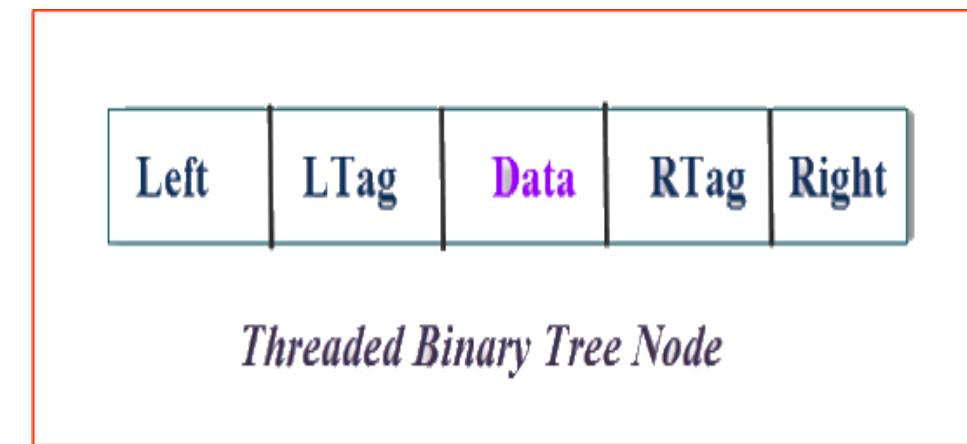
Double Threaded Binary Tree

- Single threaded: Makes inorder traversal easier
- Double threaded: Makes inorder and postorder easier

Binary Trees

■ Threaded Binary Tree (Continued...)

- Implementation requirements
 - ▶ Use a **boolean value – thread or child pointer (0 : child, 1 : thread)**



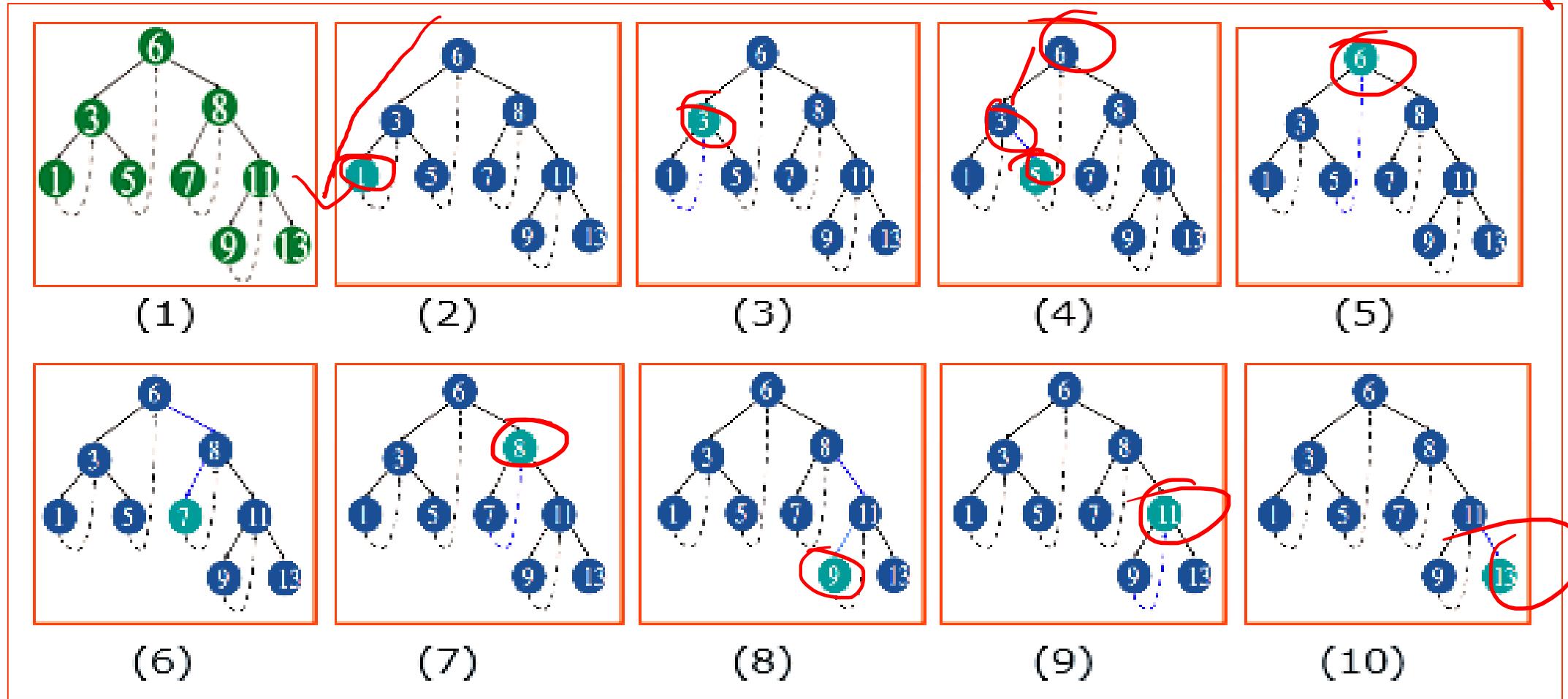
- **Advantage: Stack not required for inorder traversal**
- **Disadvantage: Adjustment of pointers during insertion and deletion of nodes**

Binary Trees

INORDER TRAVERSAL IS:-

■ Threaded Binary Tree (Continued...)

- Example (for inorder traversal)



AVL TREES.

While the binary search tree is simple and easy to understand, it has one major problem.

Not balanced.

AVL TREES.

Also called **AVL Search Trees**.

In **1962**, two Russian mathematicians,
G. M. Adelson-Velskii and **E. M. Landis**, created
the **balanced binary tree structure** named
after them
“the AVL tree”

AVL SEARCH TREES.

An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1.

{-1, 0, 1}

It is thus a **balanced binary tree**.

-1: Right High (RH)

LST is shorter than RST

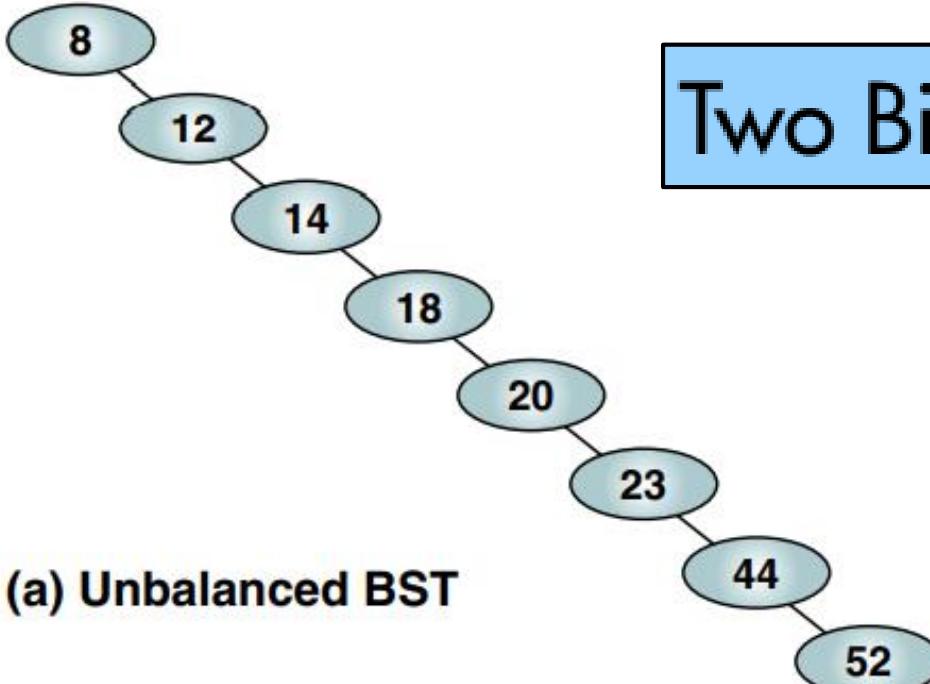
0: Even High (EH)

LST is equal to RST

+1: Left High (LH)

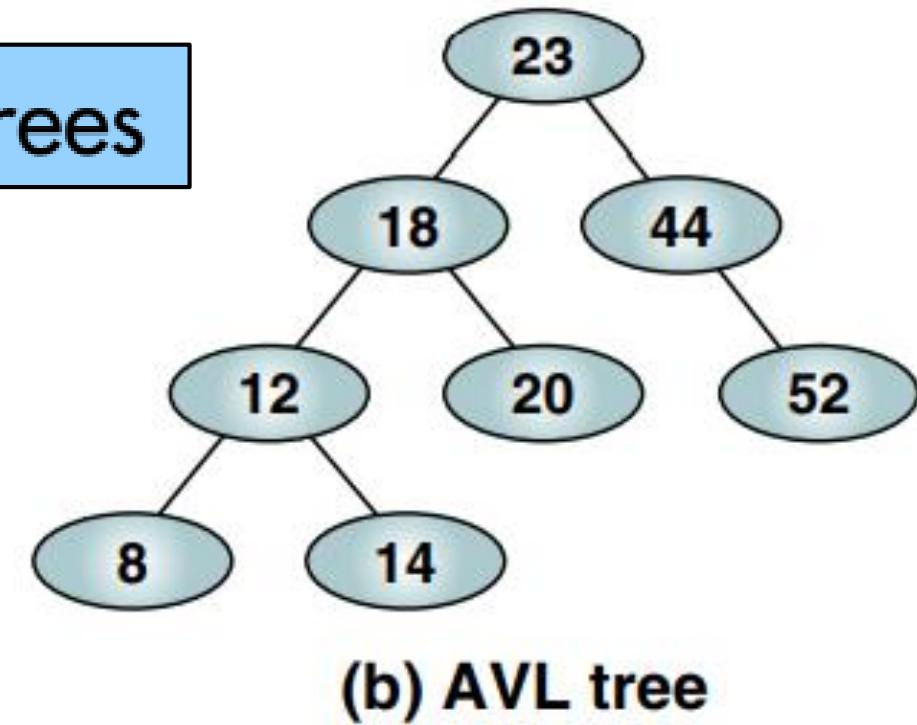
LST is longer than RST

It takes **2** tests to locate 12.
It takes **3** tests to locate 14.
It takes **8** tests to locate 52.



Two Binary Trees

It takes **4** tests to locate 8 and 14.
It takes **3** tests to locate 20 and 52.
The maximum search effort is either 3 or 4.



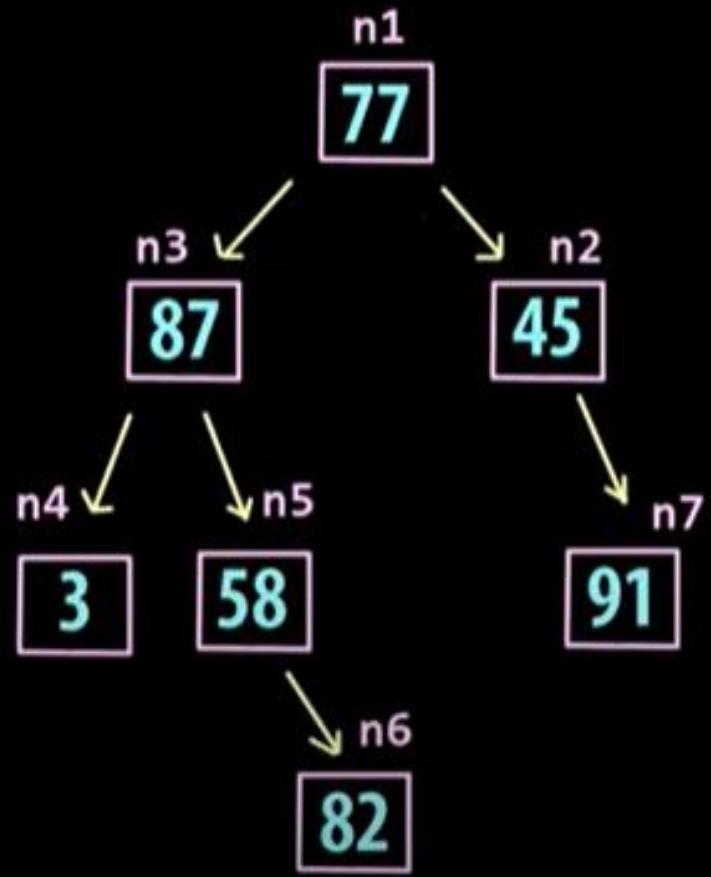
Search effort for this binary search tree is **O(n)**

Search effort for this AVL tree is **O(log n)**

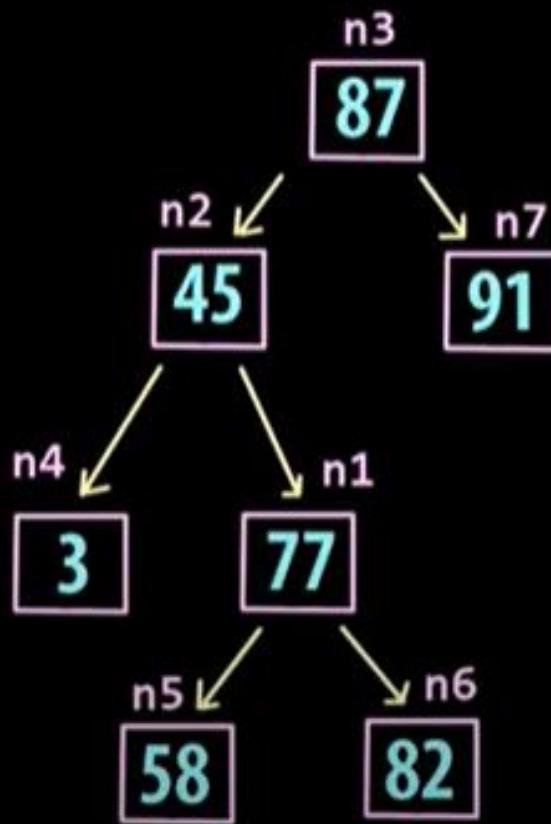
Examples for AVL Tree

An AVL tree is a height-balanced binary search tree

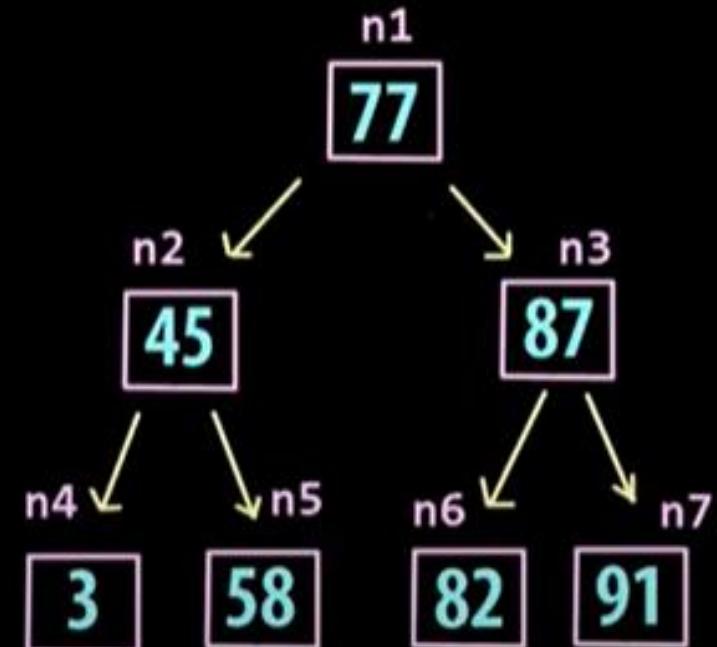
Binary Tree



Binary SEARCH Tree



AVL Tree



AVL SEARCH TREES – Definition.

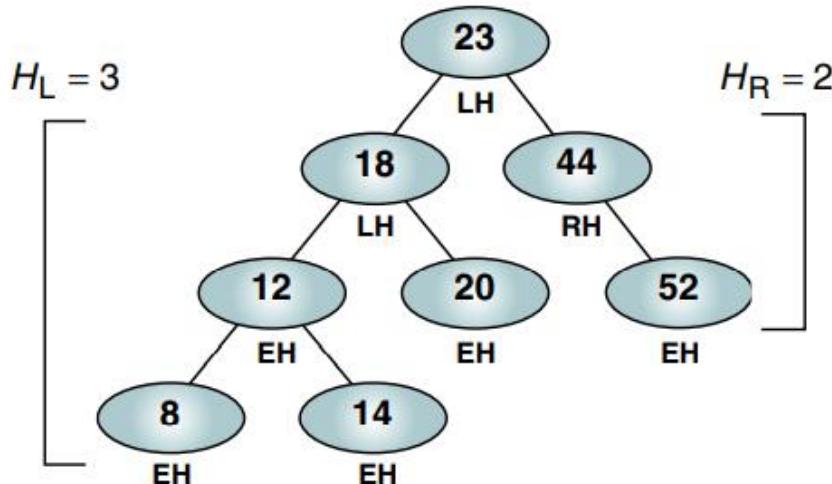
An AVL tree is a binary tree that either is empty or consists of two AVL subtrees, TL, and TR,
whose heights differ by no more than 1.

$$| H_L - H_R | \leq 1$$

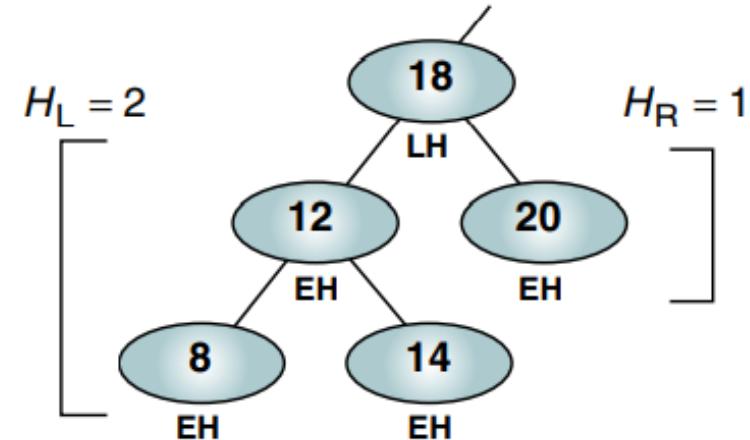
H_L is the *height of the left subtree* and H_R is the *height of the right subtree*.

Because AVL trees are balanced by working with their height, they are also known as **height-balanced trees**.

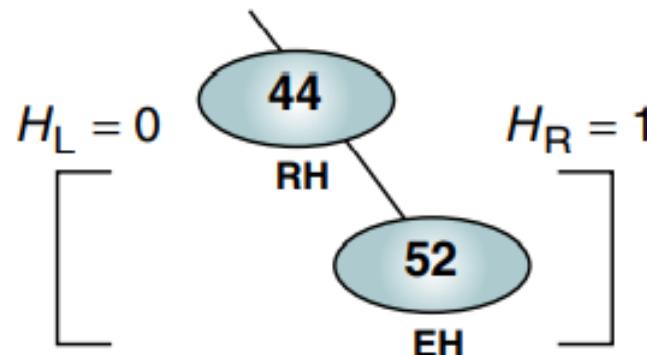
BALANCE FACTORS of AVL Trees



(a) Tree 23 appears balanced: $H_L - H_R = 1$



(b) Subtree 18 appears balanced:
 $H_L - H_R = 1$



(c) Subtree 44 is balanced:
 $| H_L - H_R | = 1$

Why BALANCING TREES?

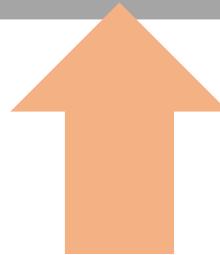
- Whenever we **insert a node** into a tree or delete a node from a tree, the *resulting tree may be unbalanced.*
- When we detect that a ***tree has become unbalanced***, we must ***rebalance*** it.
- AVL trees are balanced by rotating nodes either to the left or to the right.**

Consider the basic balancing algorithms for four cases of unbalancing:

UNBALANCED

Right of right

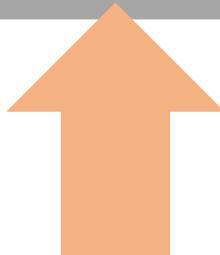
A subtree of a tree that is right high has also become right high.



Left Rotation

Left of left

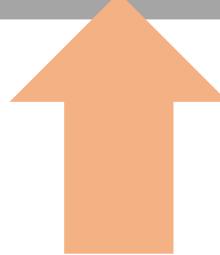
A subtree of a tree that is left high has also become left high.



Single Rotation Left

Right of left:

A subtree of a tree that is left high has become right high.



Left Right Rotation

Left of right:

A subtree of a tree that is right high has become left high.



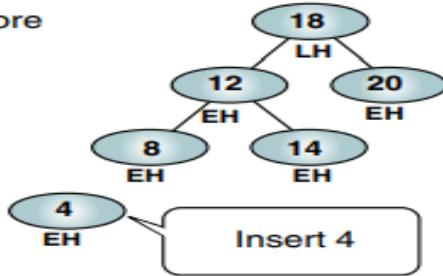
Right Left Rotation

Single Rotation Right

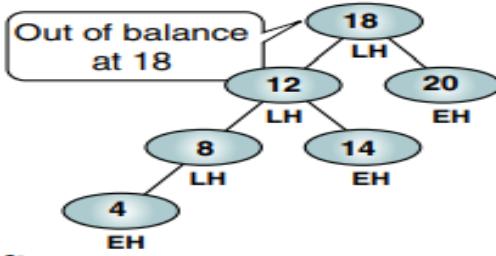
Double Rotation

Double Rotation

Before

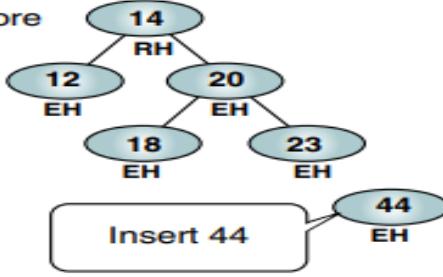


After

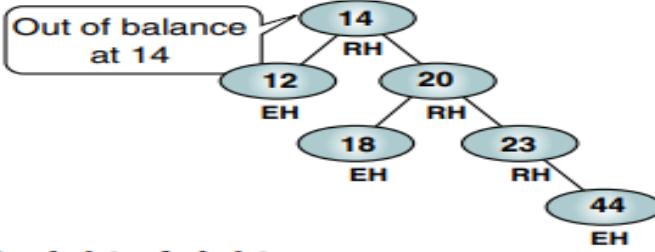


(a) Case 1: left of left

Before

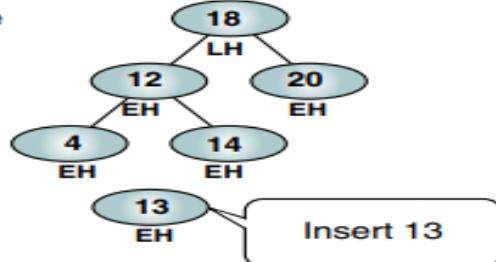


After

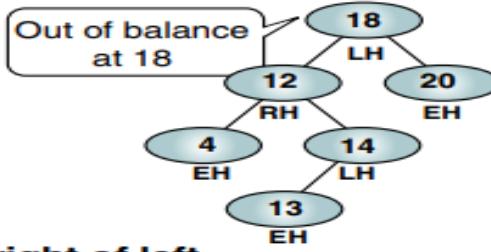


(b) Case 2: right of right

Before

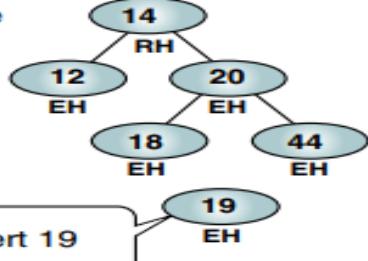


After

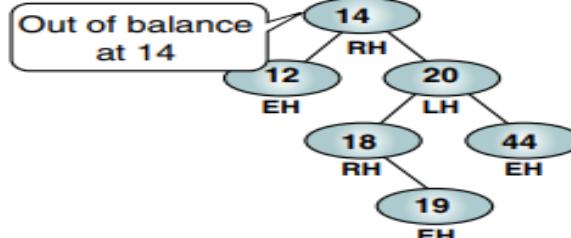


(c) Case 3: right of left

Before



After



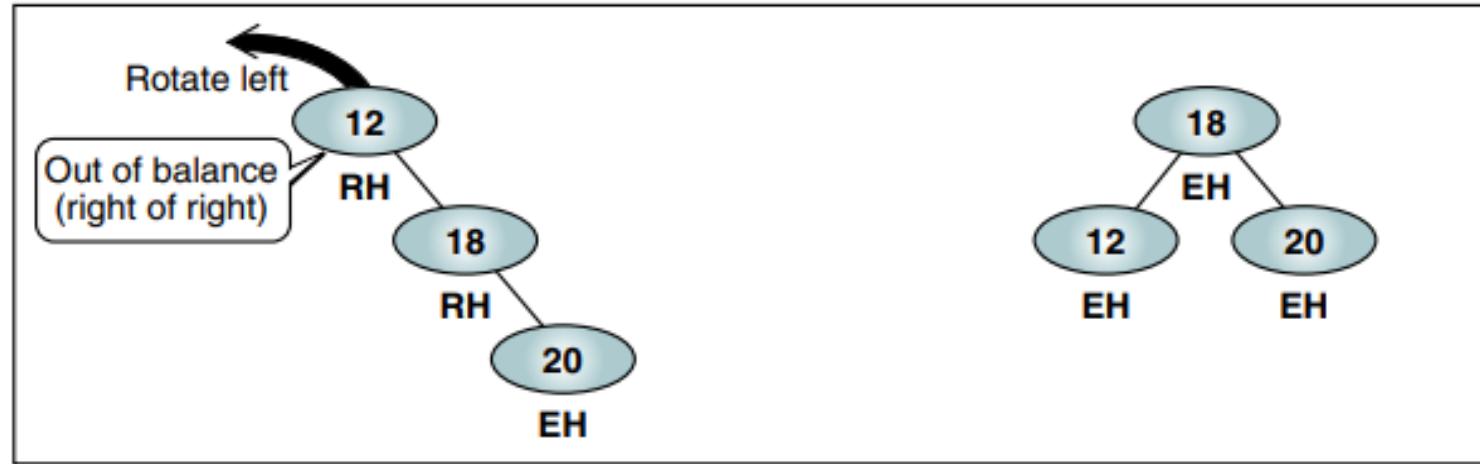
(d) Case 4: left of right

Out-of-balance AVL Trees

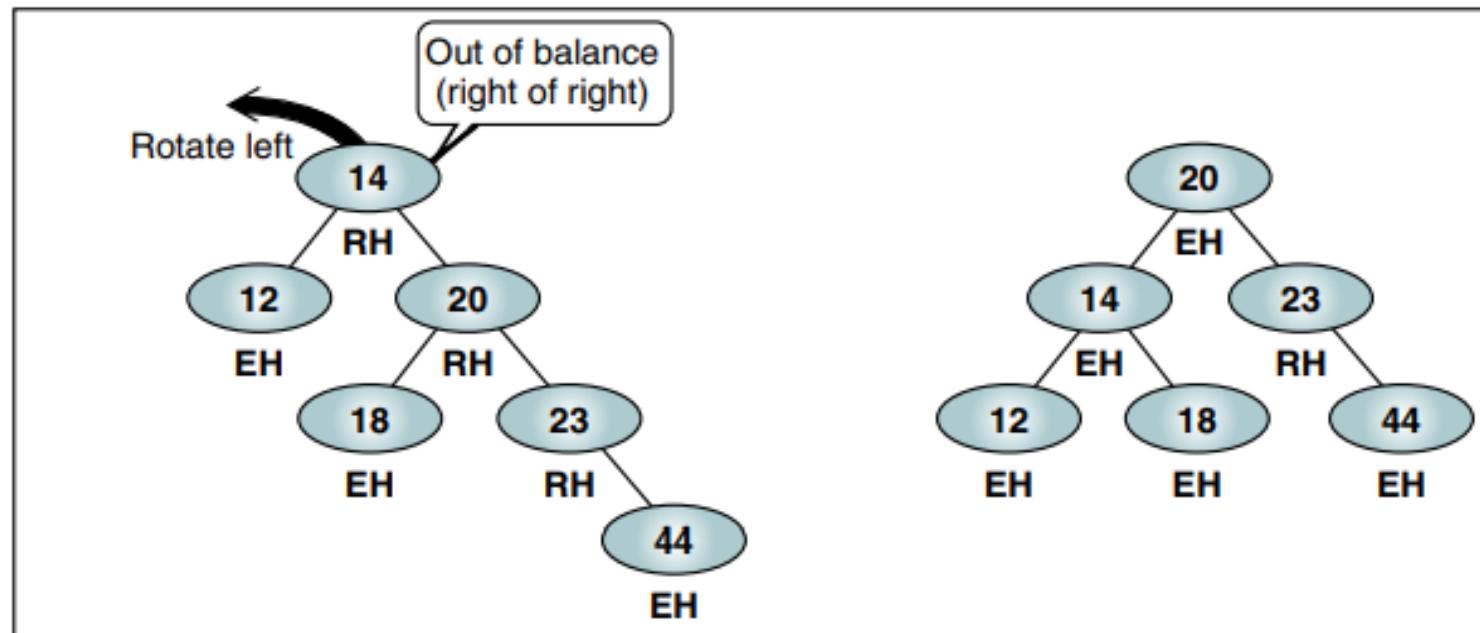
Case 1: Left Rotation or Single Rotation Left

Insertion is Right of Right

Case 1: Left Rotation or Single Rotation Left



(a) Simple left rotation

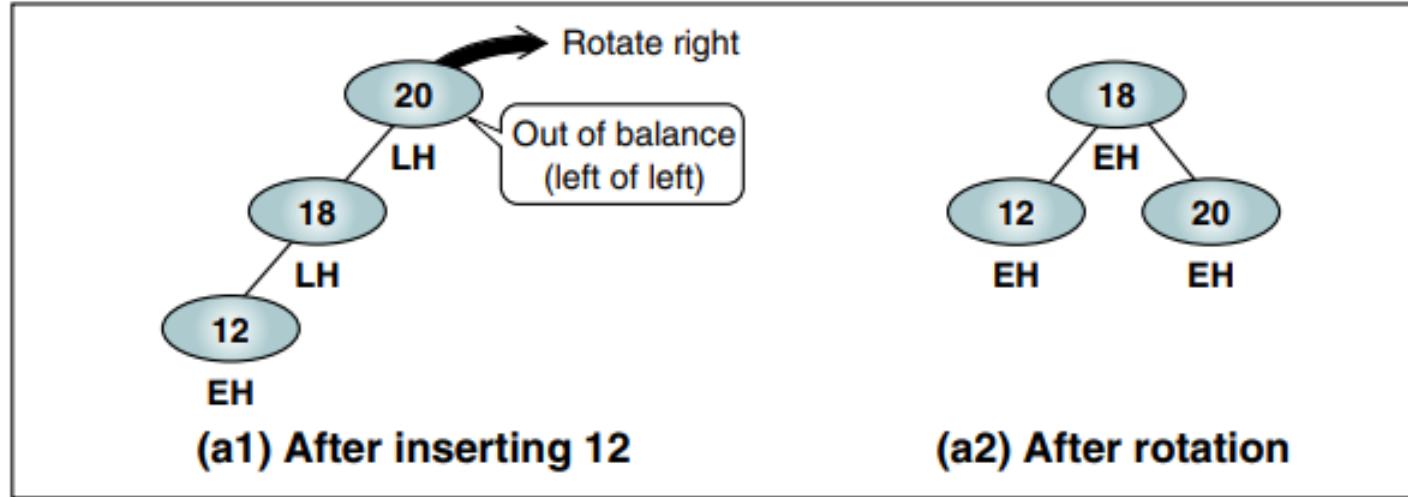


(b) Complex left rotation

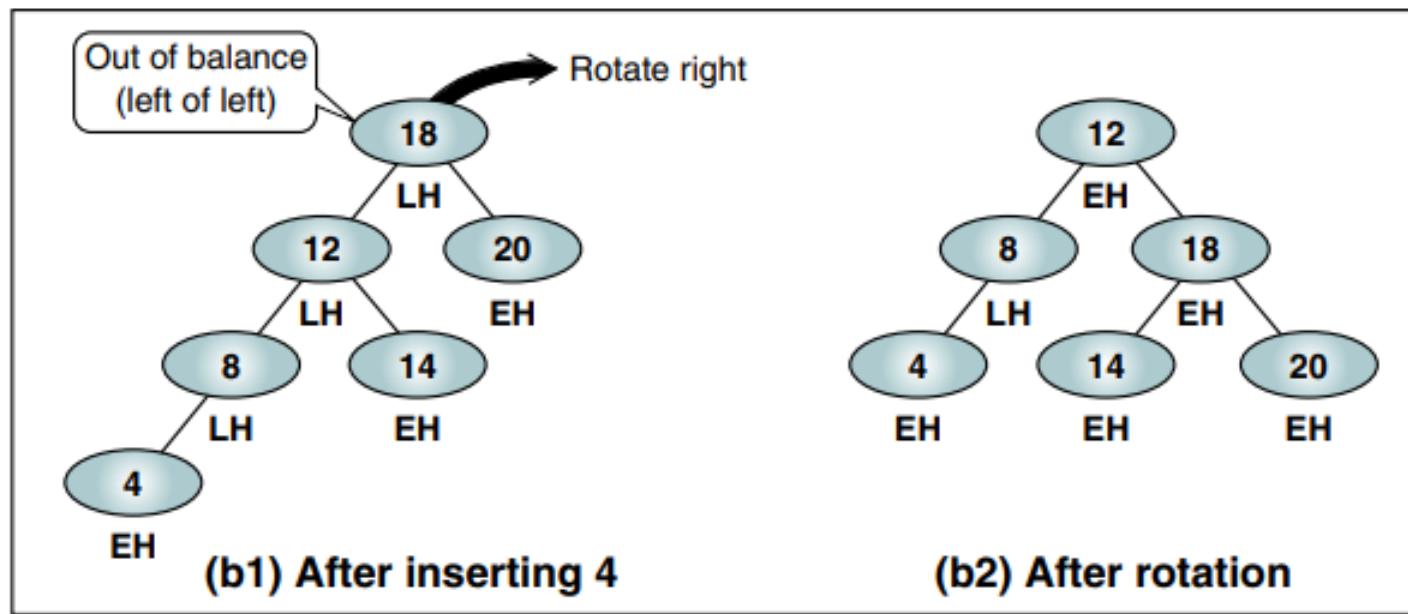
Case 2: Right Rotation or Single Rotation Right

Insertion is Left of Left

Case 2: Right Rotation or Single Rotation Right



(a) Simple right rotation

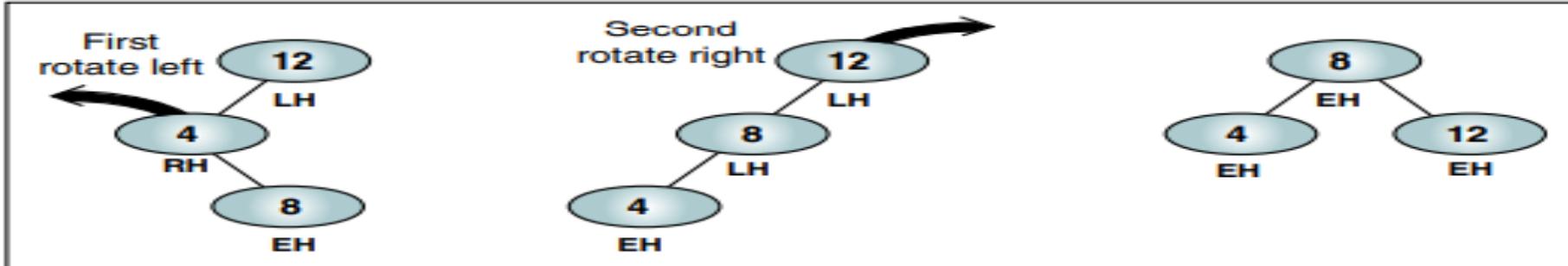


(b) Complex right rotation

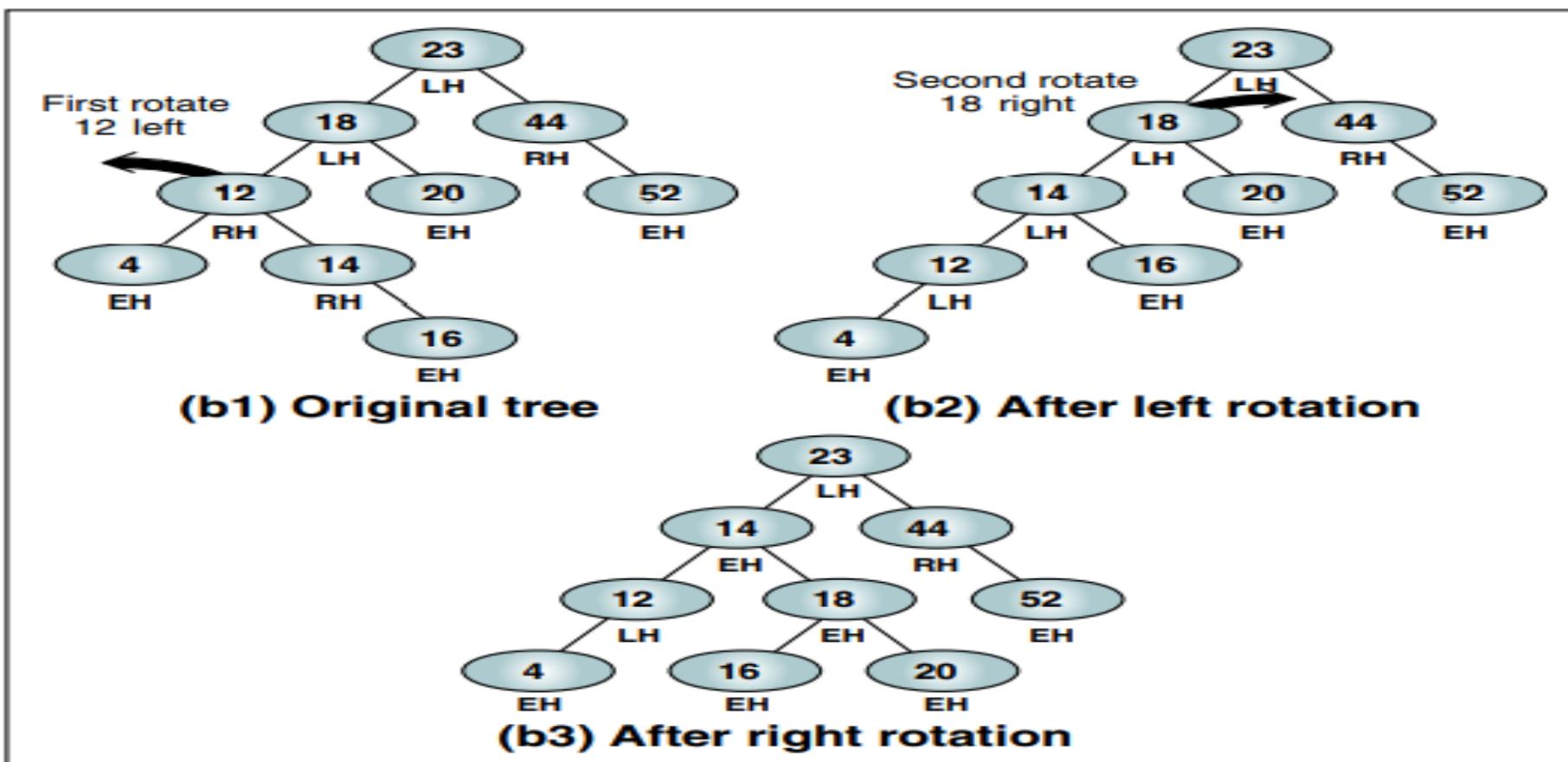
Case 3: Left - Right Rotation or Double Rotation

Insertion is Left and then Right (Right of left Insertion).

Case 3: Left - Right Rotation or Double Rotation Left



(a) Simple double rotation right

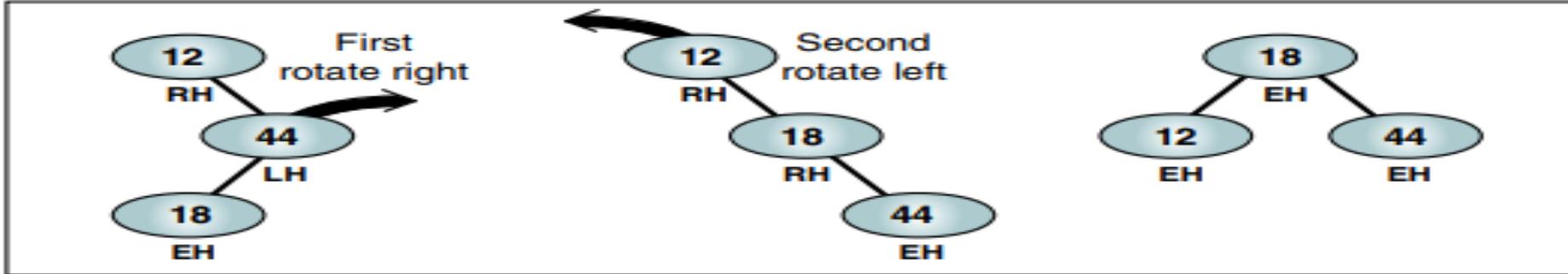


(b) Complex double rotation right

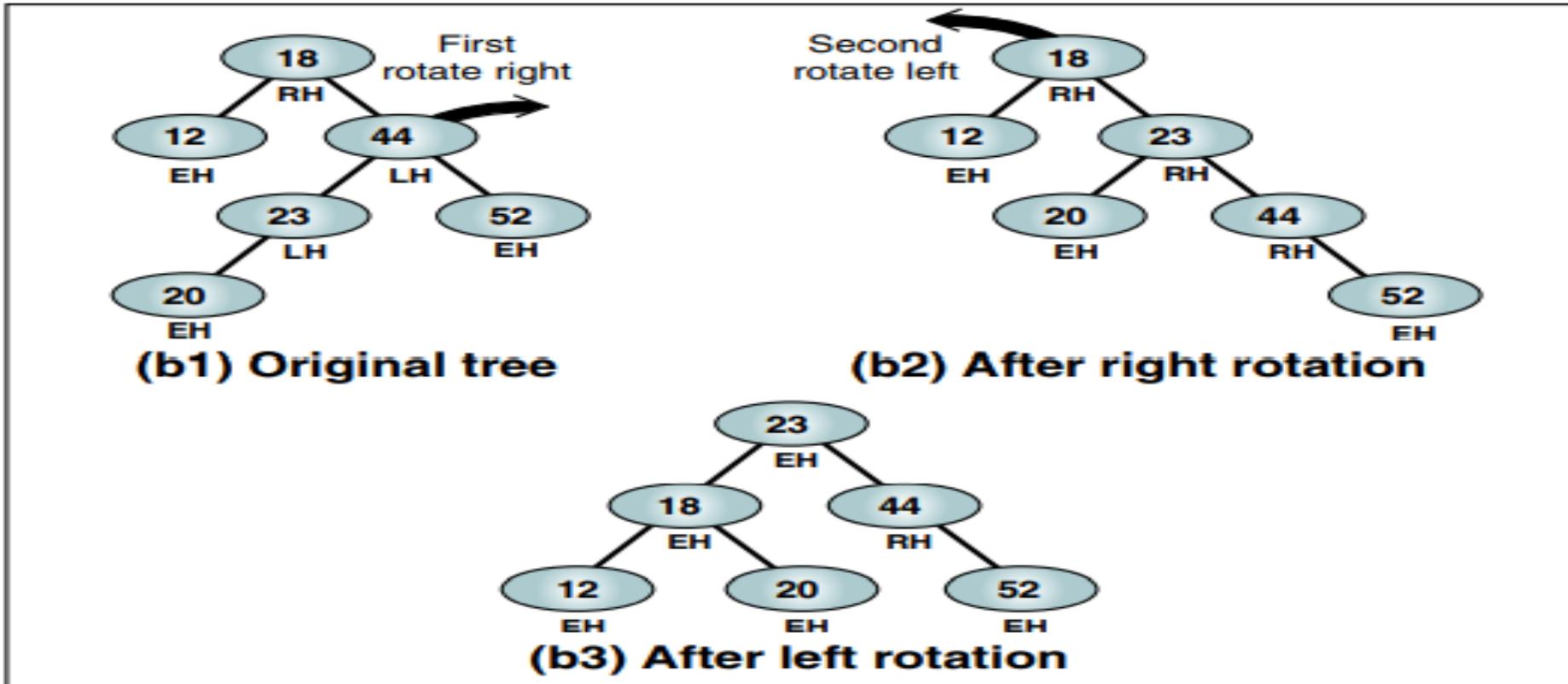
Case 4: Right-Left Rotation or Double Rotation

Insertion is Right and then left (**Left of Right Insertion**).

Case 4: Right-Left Rotation or Double Rotation



(a) Simple double rotation right



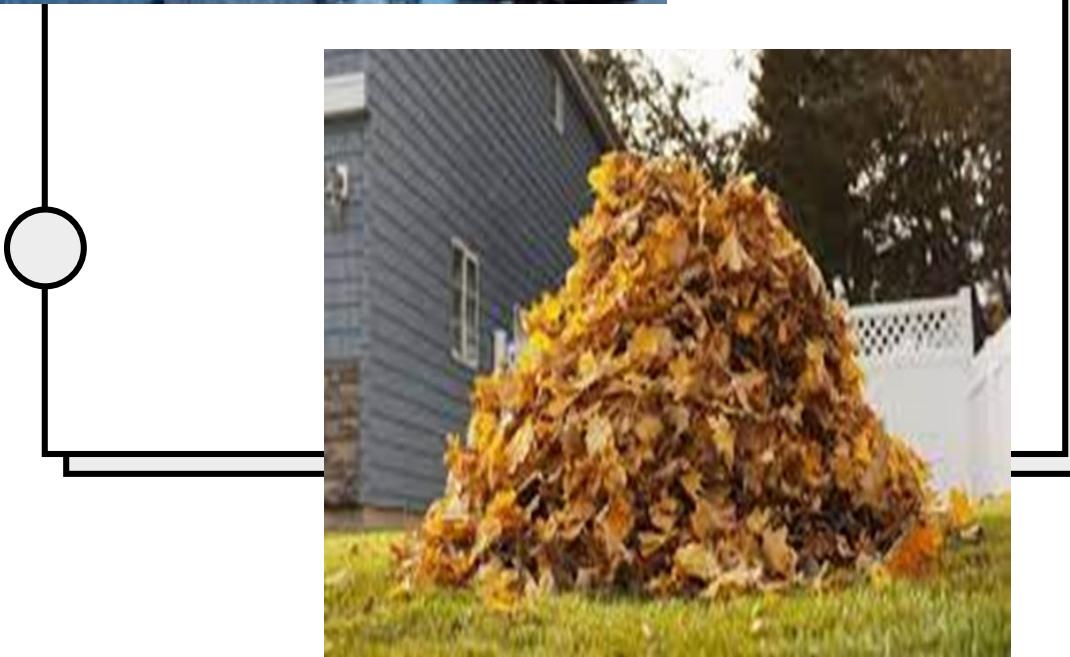
(b) Complex double rotation right

Insertion



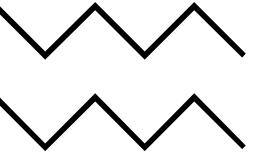
Thank You.



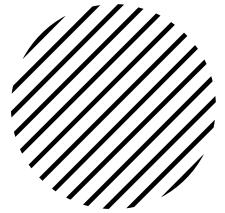


HEAPS.
ALSO, A TREE.





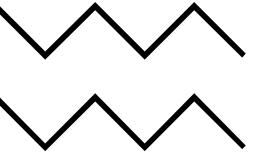
Terminology.



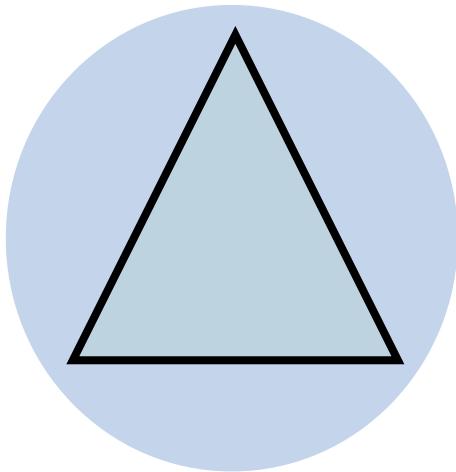
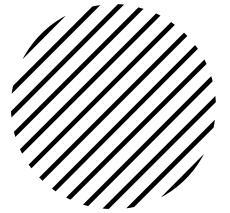
Full Binary Tree.

Complete Binary Tree.



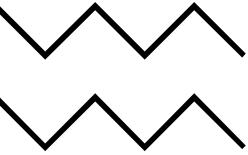


Full Binary Tree.

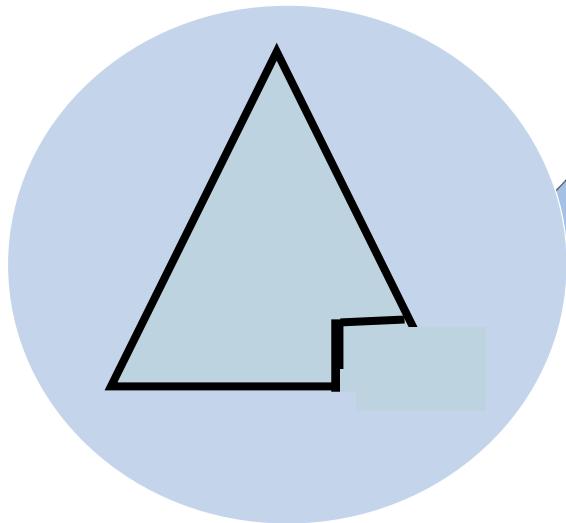
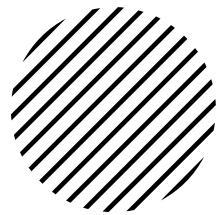


Every non-leaf node has two children
All the leaves are on the same level



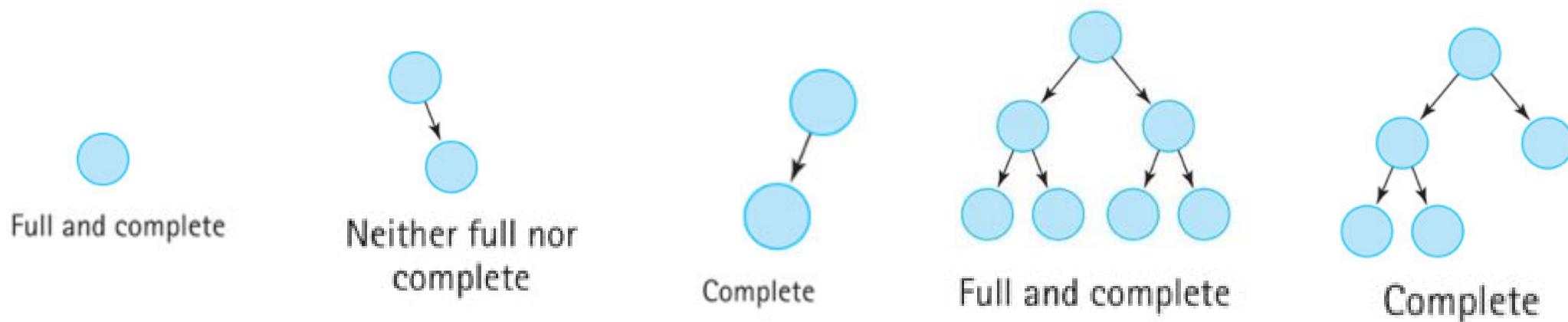


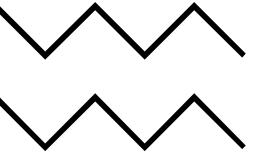
Complete Binary Tree.



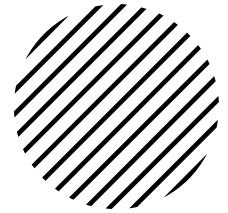
A binary tree that is either full or full through the next-to-last level

The last level is full from left to right (i.e., leaves are as far to the left as possible)



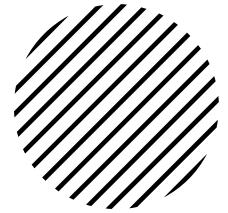
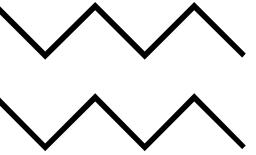


Heaps.



- ↳ A ***third type of tree*** is a heap.
- ↳ A heap is a binary tree whose left and right subtrees have values less than their parents.
- ↳ The root of a heap is guaranteed to hold the largest node in the tree; its subtrees contain data that have lesser values.
- ↳ Unlike the binary search tree, however, the **lesser-valued nodes of a heap can be placed on either the right or the left subtree**. Therefore, both the left and the right branches of the tree have the same properties.

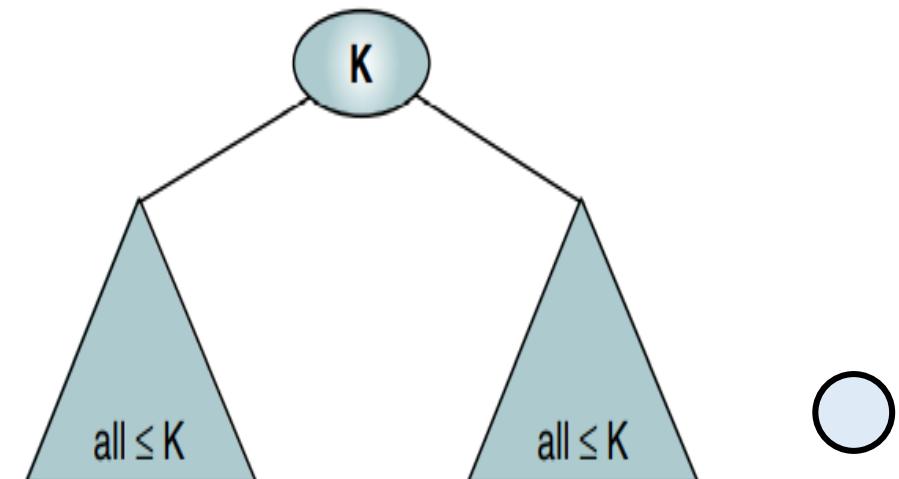


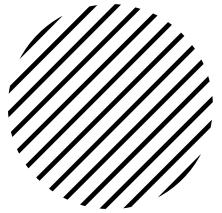
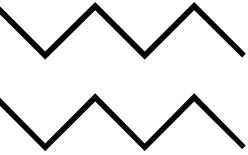


Basic Concepts.

A heap is a binary tree structure (*heap ordered*) with the following properties:

1. The tree is ***complete or nearly complete***.
2. The key value of each node is ***greater than or equal to the key*** value in each of its descendants.

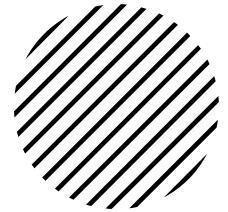
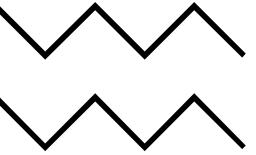




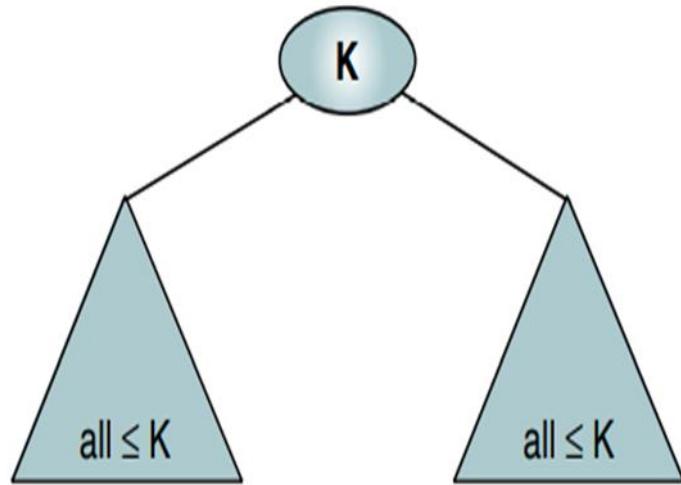
Basic Concepts - Definition.

A heap is a ***complete or nearly complete binary tree*** in which the key value in a node is greater than or equal to the key values in all of its subtrees, and the subtrees are in turn heaps.



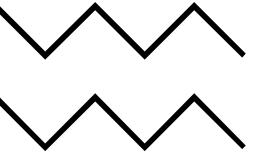


Heaps.

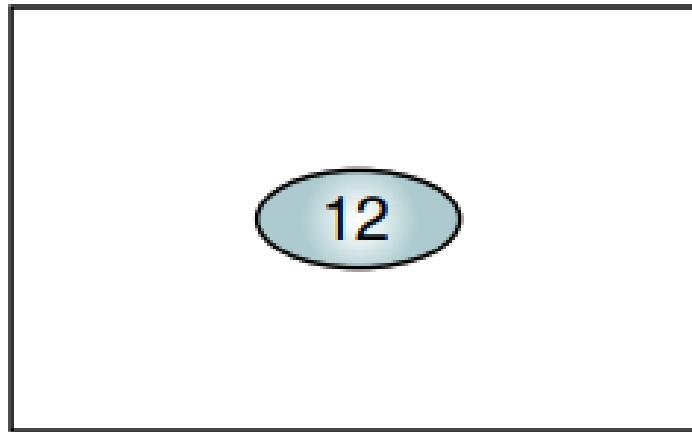
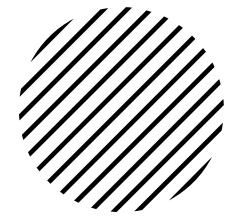


- ↳ Sometimes this structure is called a **max-heap**.
- ↳ The second property of a heap “key value is greater than the keys of the subtrees” can be reversed to create a min-heap.
- ↳ **Min heap:** *Key value in a node is less than the key values in all of its subtrees.*
- ↳ Generally speaking, whenever the term **heap** is used by itself, it **refers to a max-heap**.

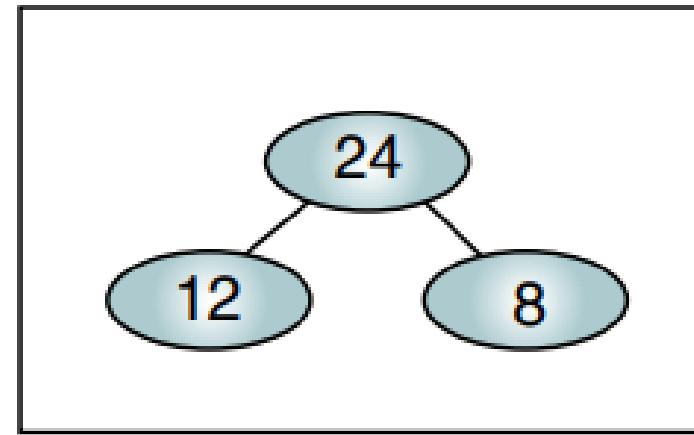




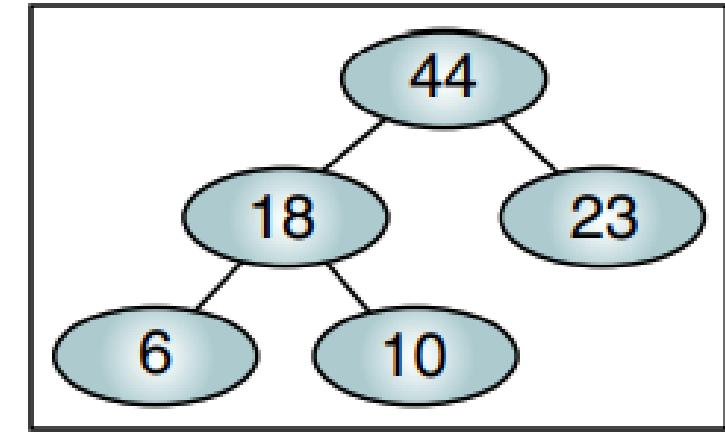
Examples of *Valid* Heap trees.



(a) Root-only heap

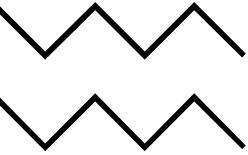


(b) Two-level heap

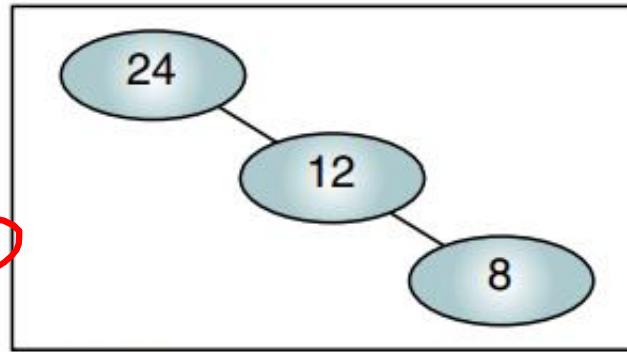
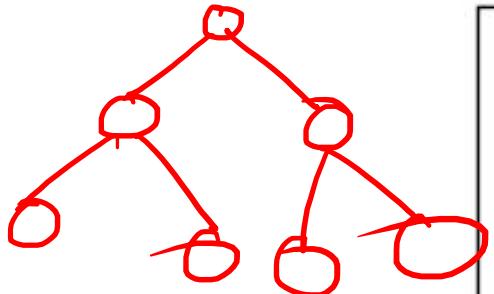
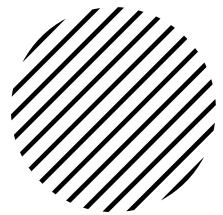


(c) Three-level heap

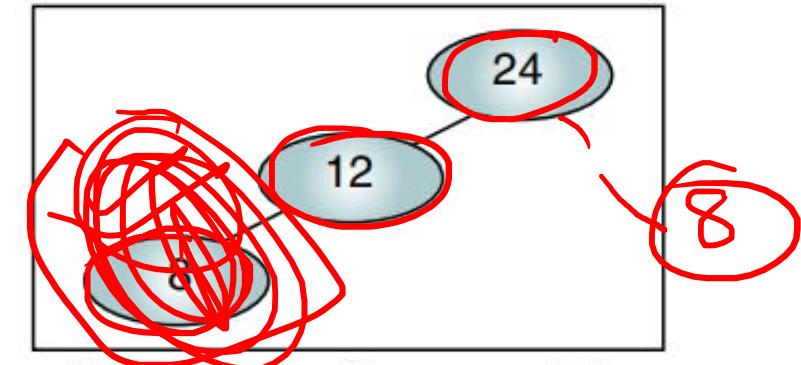




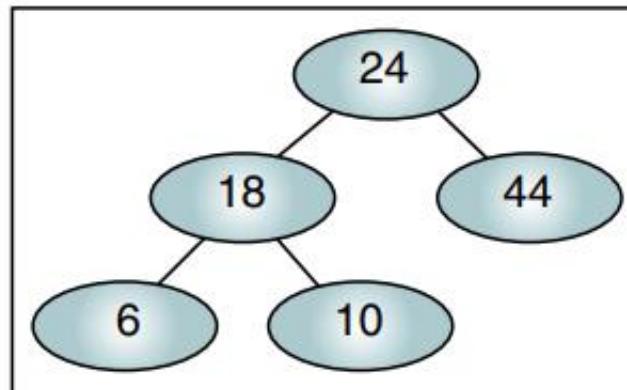
Examples of *Invalid* Heap trees.



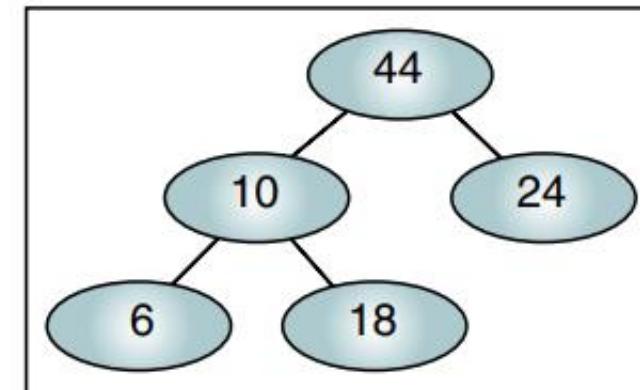
**(a) Not nearly complete
(rule 1)**



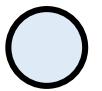
**(b) Not nearly complete
(rule 1)**

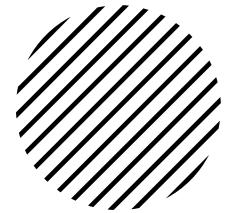
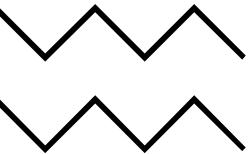


**(c) Root not largest
(rule 2)**

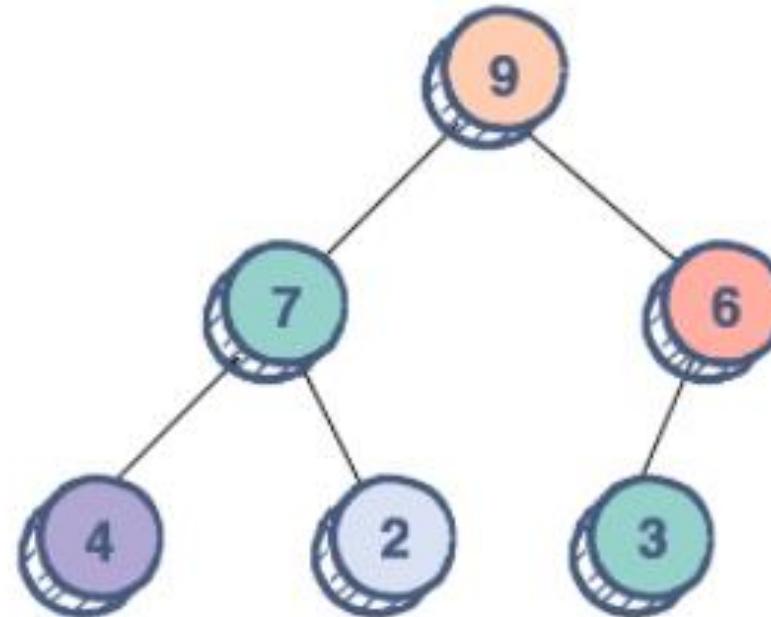


**(d) Subtree 10 not a heap
(rule 2)**



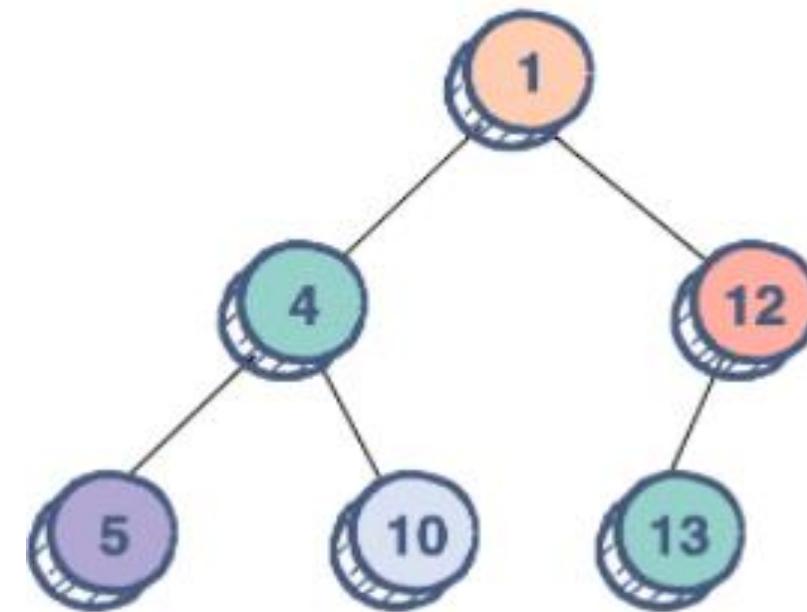


Max Heap and Min Heap.



If Node A has a child node B,
then,

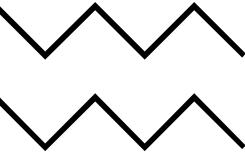
$$\text{key}(A) \geq \text{key}(B)$$



If Node A has a child node B,
then,

$$\text{key}(A) \leq \text{key}(B)$$



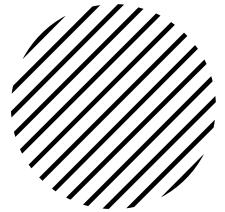
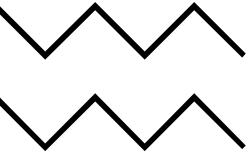


Some more terms.

Heapify: To rearrange a heap to maintain the heap property, that is, the key of the root node is more extreme (greater or less) than or equal to the keys of its children. If the root node's key is not more extreme, swap it with the most extreme child key, then recursively heapify that child's subtree. The child subtrees must be heaps to start.

or

✓ Heapify is the process of converting a binary tree into a Heap data structure

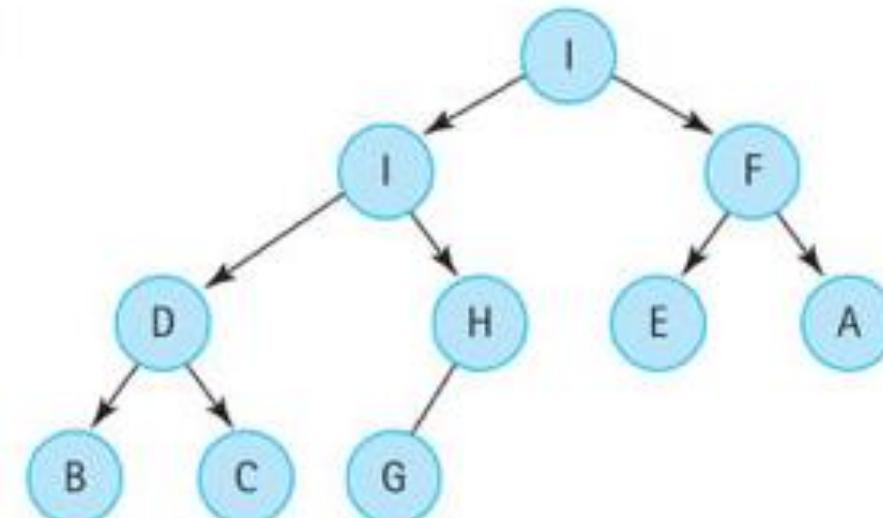
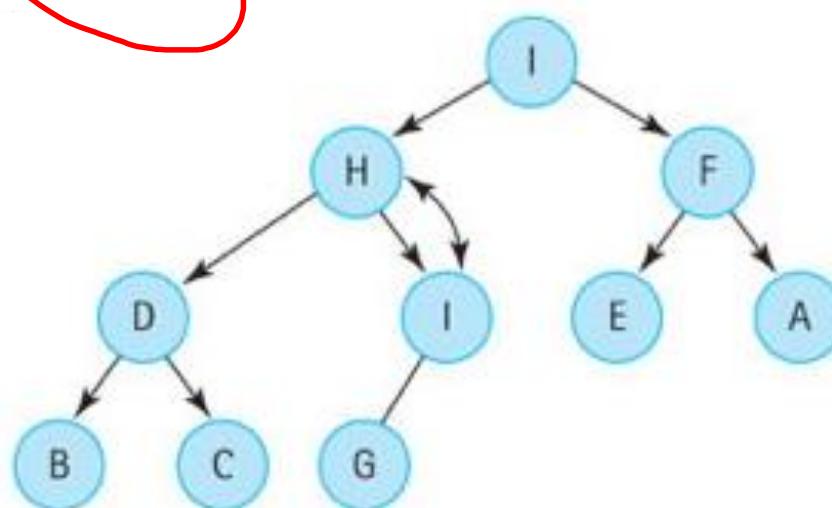
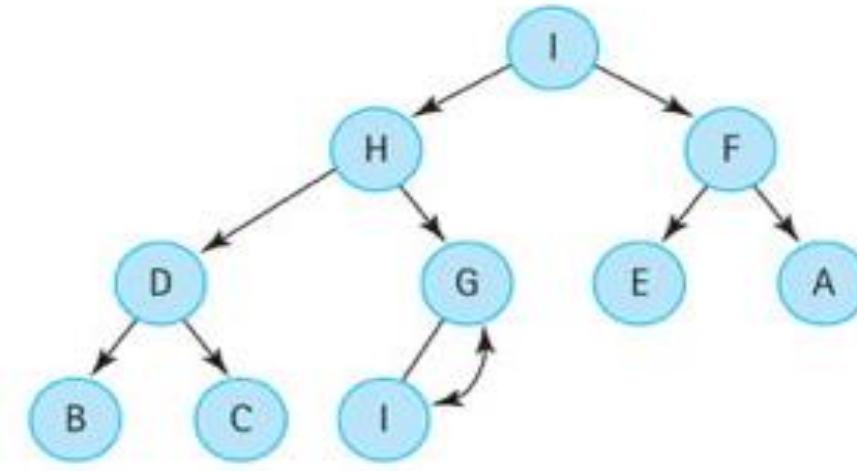
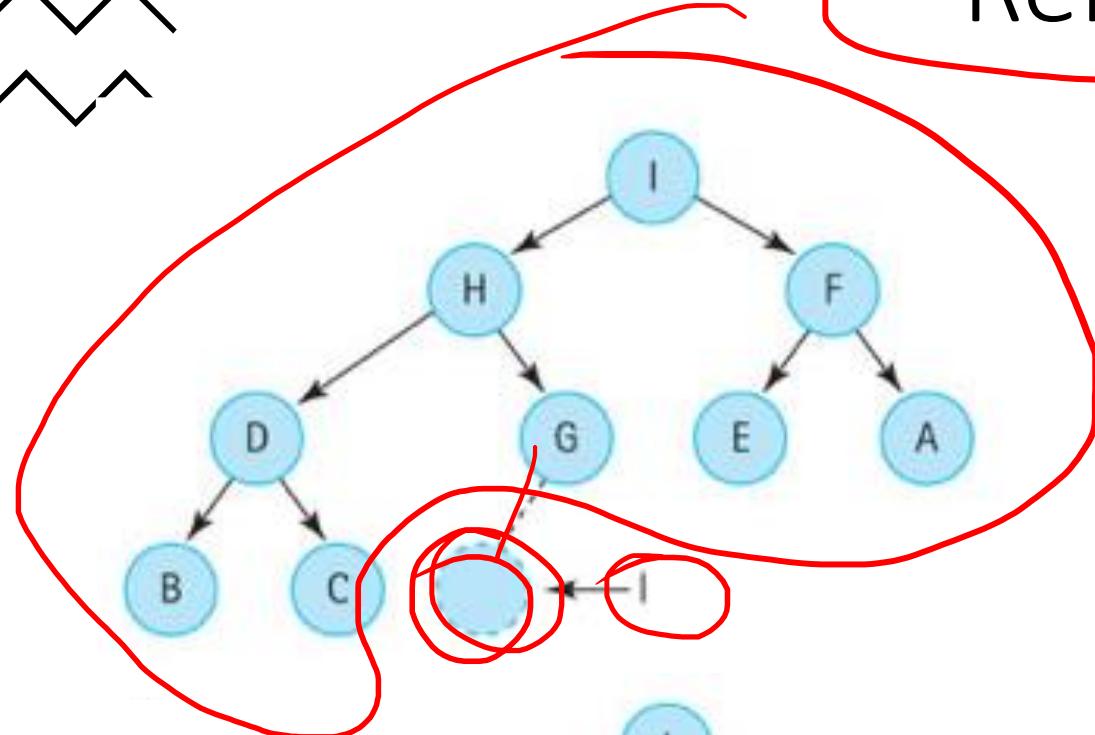
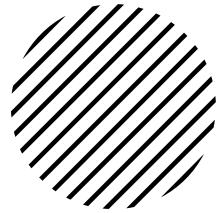
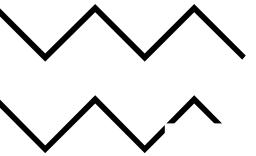


Some more terms.

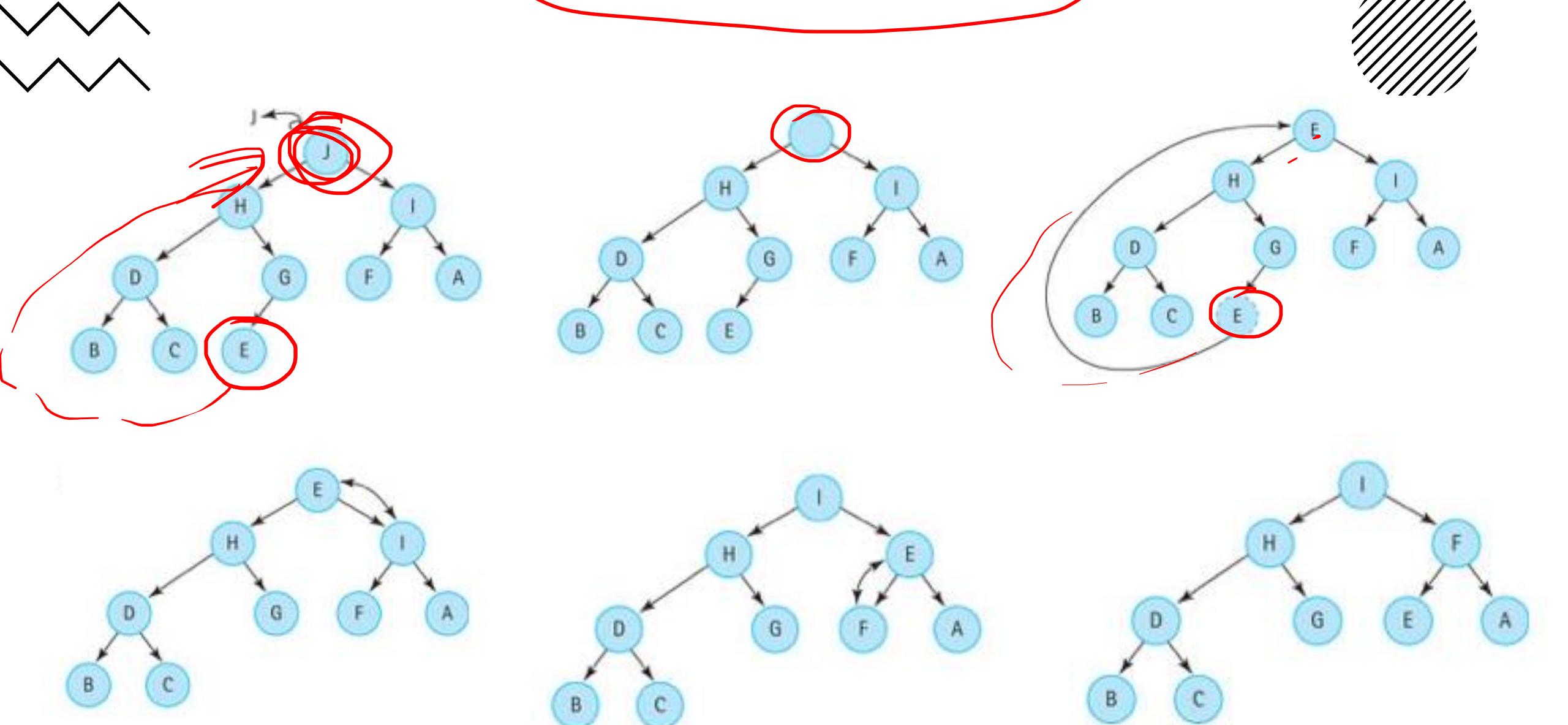
ReheapUp: Add the new item to the first empty leaf at the bottom level and re-establish the heap characteristic by swapping with the parent if the child should be the new parent. Repeat upward until the root is reached or no swap is necessary

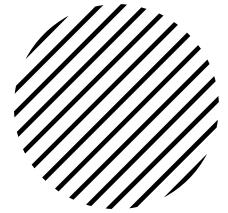
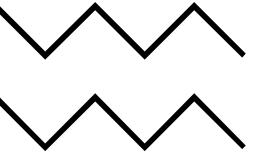
Doh! Doh!
ReheapDown: Root is removed and then the heap is re-established by pulling upward the child that has the largest value. Repeat until no shift upward is necessary or the leaf is reached. Use the rightmost, bottom level value as the fill-in.

Reheap Up.



Reheap Down.





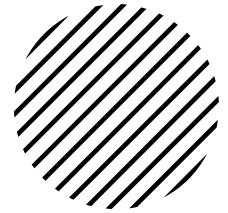
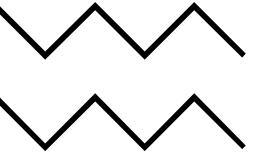
Heap Implementation.

💡 Although a heap can be built in a dynamic tree structure, it is most often **implemented in an array**.

💡 How ?

A heap can be implemented in an array because it must be a complete or nearly complete binary tree, which allows a fixed relationship between each node and its children.

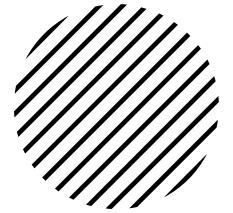
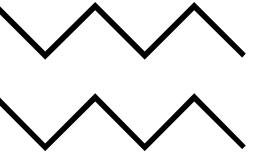




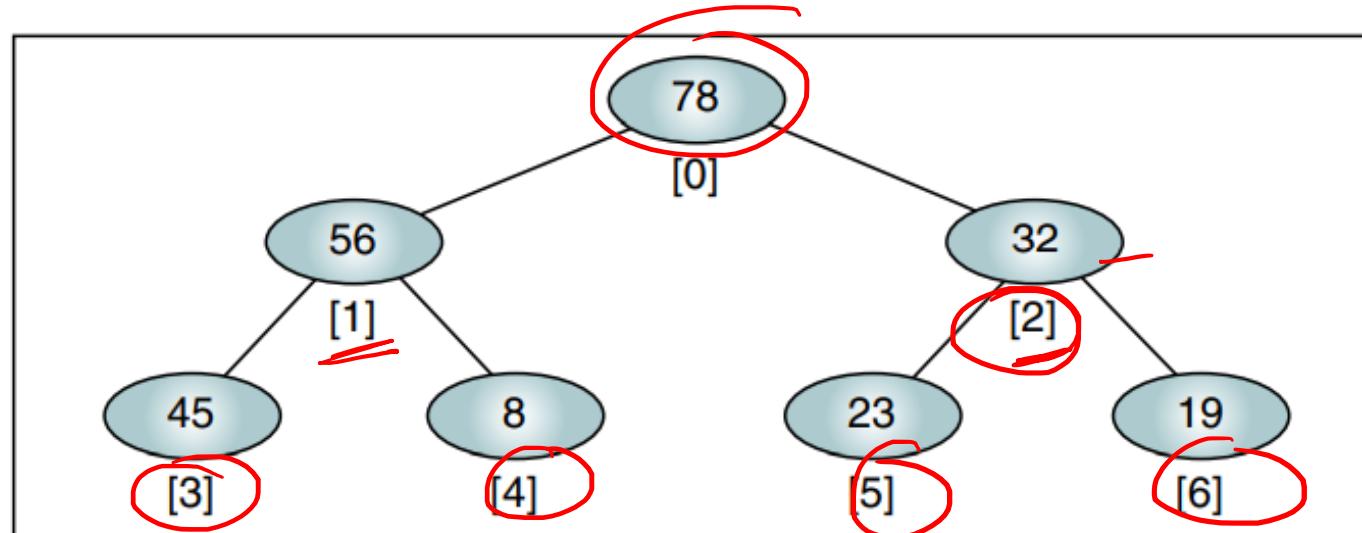
Heap Implementation Relationships.

1. For a node located at index i , its children are found at:
 - a. Left child: $2i + 1$
 - b. Right child: $2i + 2$
2. The parent of a node located at index i is located at $\lfloor (i - 1) / 2 \rfloor$.
3. Given the index for a left child, j , its right sibling, if any, is found at $j + 1$. Conversely, given the index for a right child, k , its left sibling, which must exist, is found at $k - 1$.
4. Given the size, n , of a complete heap, the location of the first leaf is $\lfloor (n / 2) \rfloor$.
5. Given the location of the first leaf element, the location of the last nonleaf element is one less.

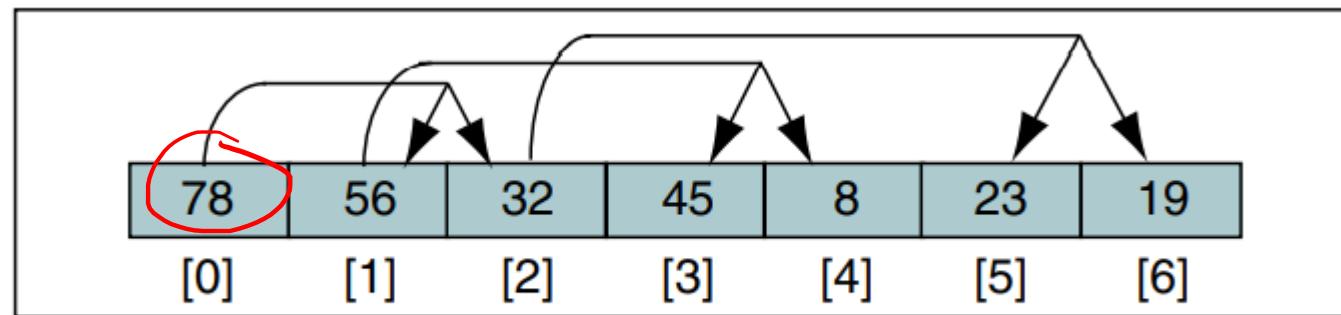




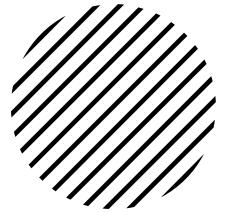
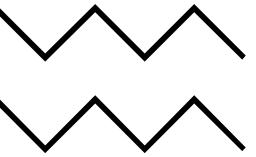
Heaps in Arrays.



(a) Heap in its logical form



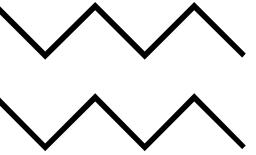
(b) Heap in an array



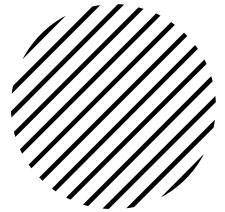
Heaps in Arrays.

1. The index of 32 is 2, so the index of its left child, 23, is $2 \times 2 + 1$, or 5. The index of its right child, 19, is $2 \times 2 + 2$, or 6 (Relationship 1).
2. The index of 8 is 4, so the index of its parent, 56, is $\lfloor(4 - 1) / 2\rfloor$, or 1 (Relationship 2).
3. In the first example, we found the address of the left and the right children. To find the right child, we could also have used the location of the left child (5) and added 1 (Relationship 3).
4. The total number of elements is 7, so the index of the first leaf element, 45, is $\lfloor(7 / 2)\rfloor$, or 3 (Relationship 4).
5. The location of the last nonleaf element, 32, is $3 - 1$, or 2 (Relationship 5).





Operations on Heaps.



①

Insertion

②

Deletion

③

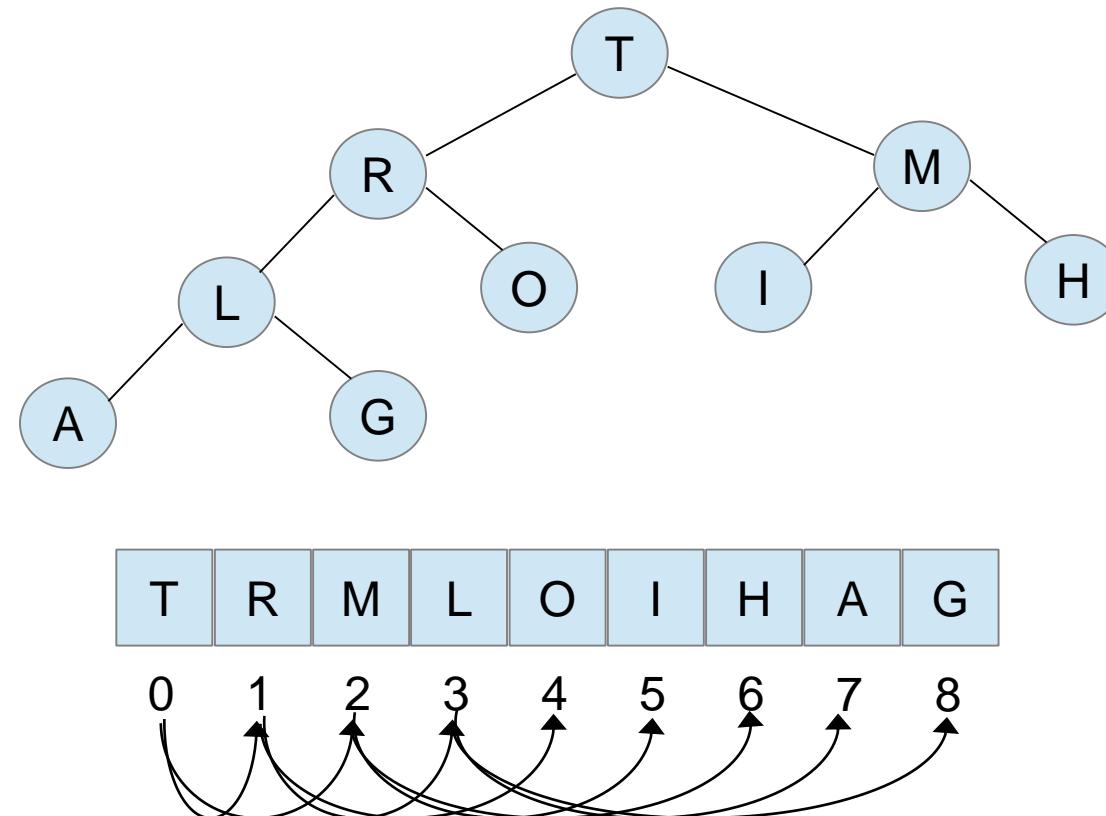
Construction

Initialization

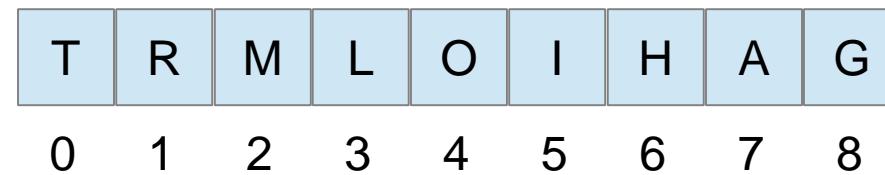
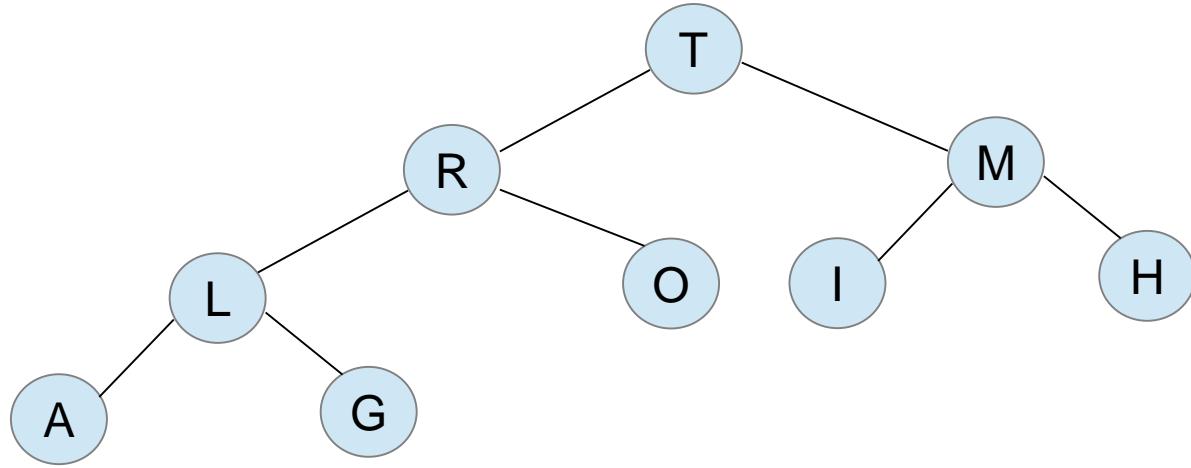
as Heap



Heap Implementation (Example 3).

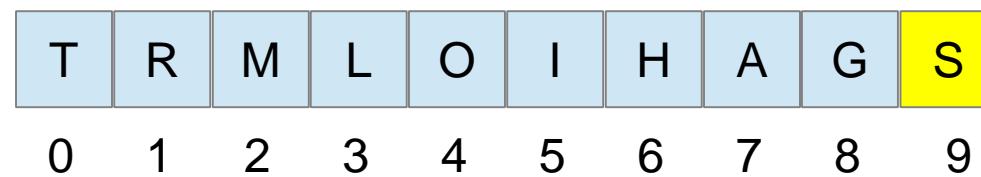
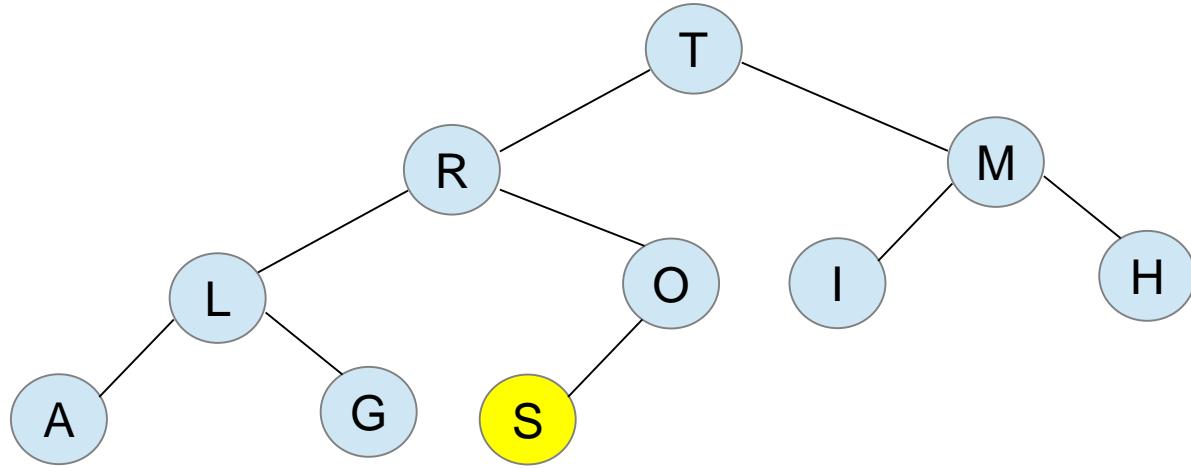


Heapification.



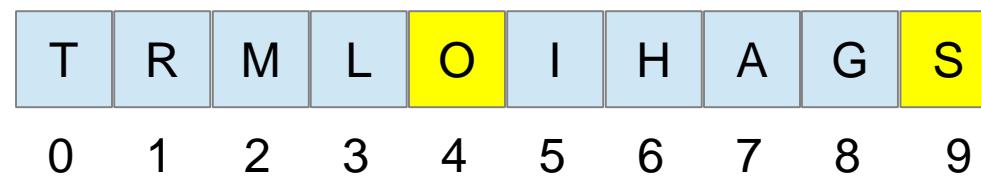
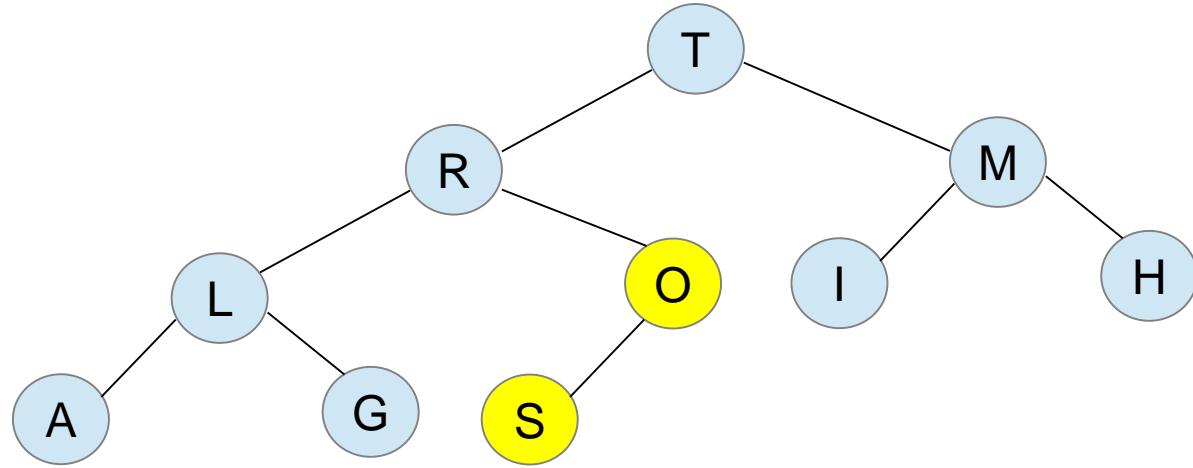
Heap-broken

Heapification.



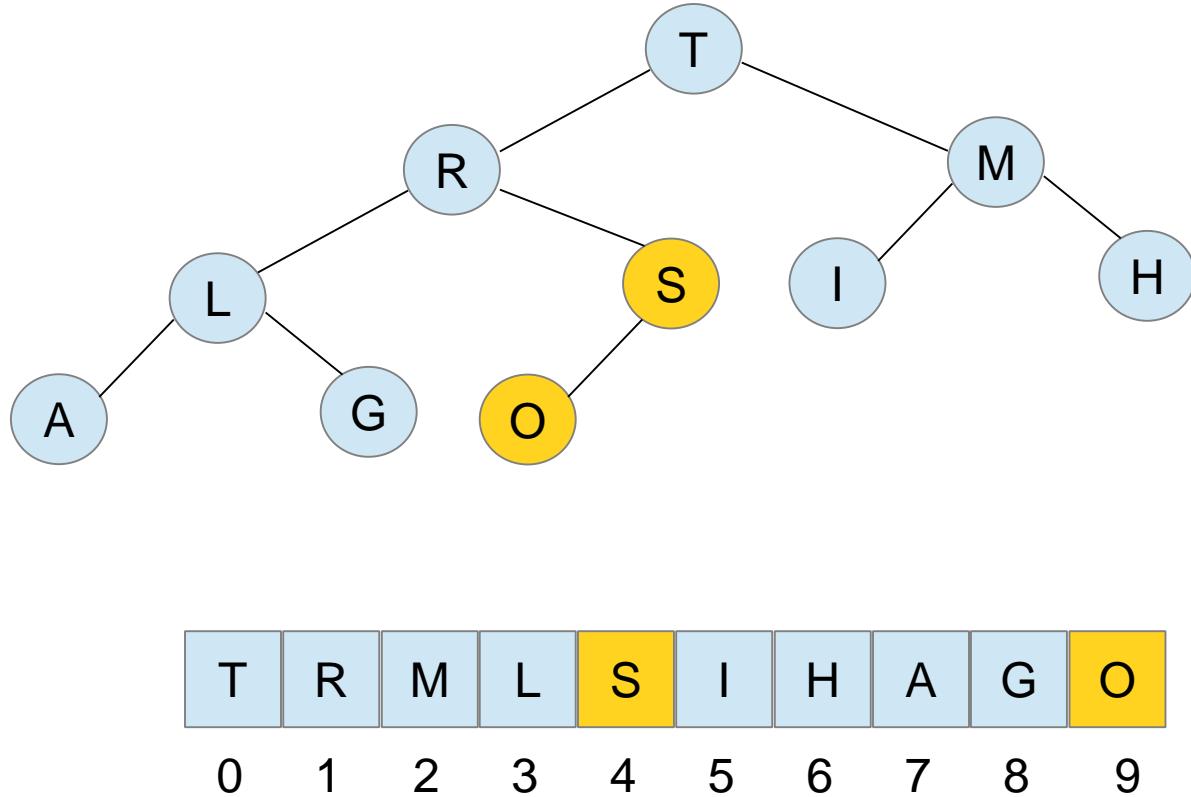
Heap-broken

Heapification.



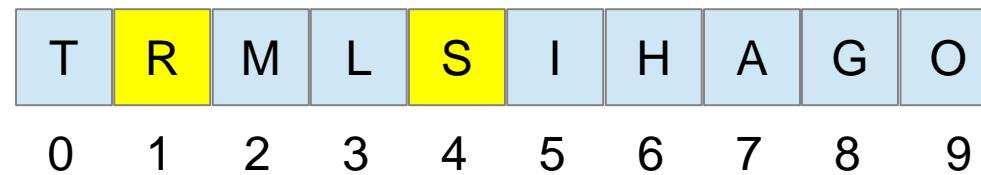
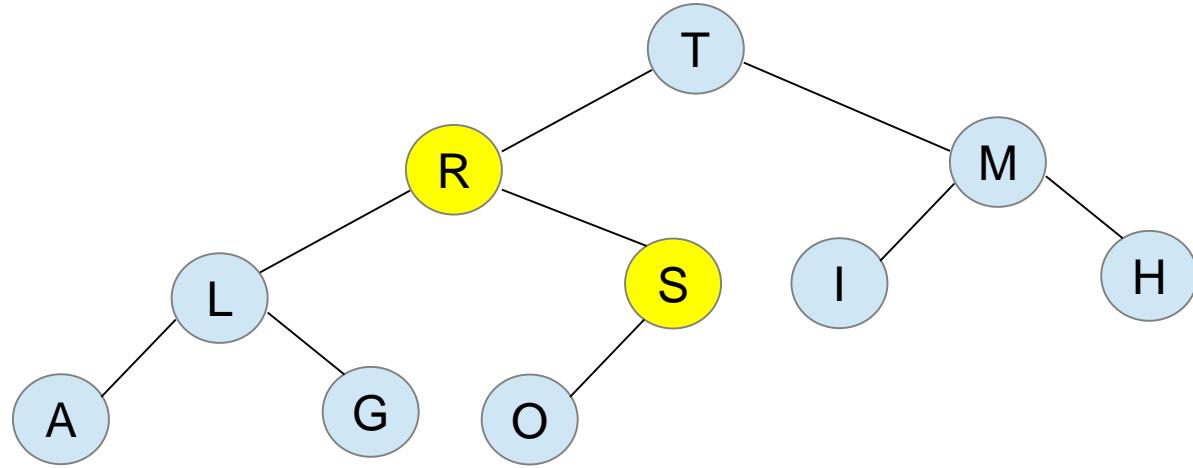
Heap-broken

Heapification.



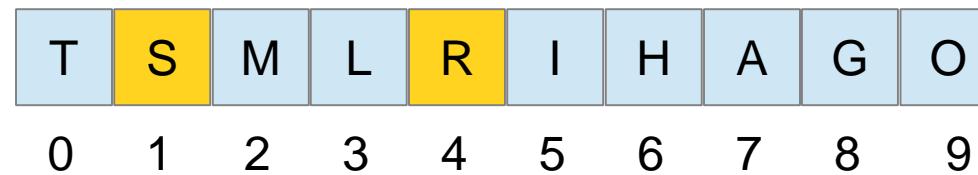
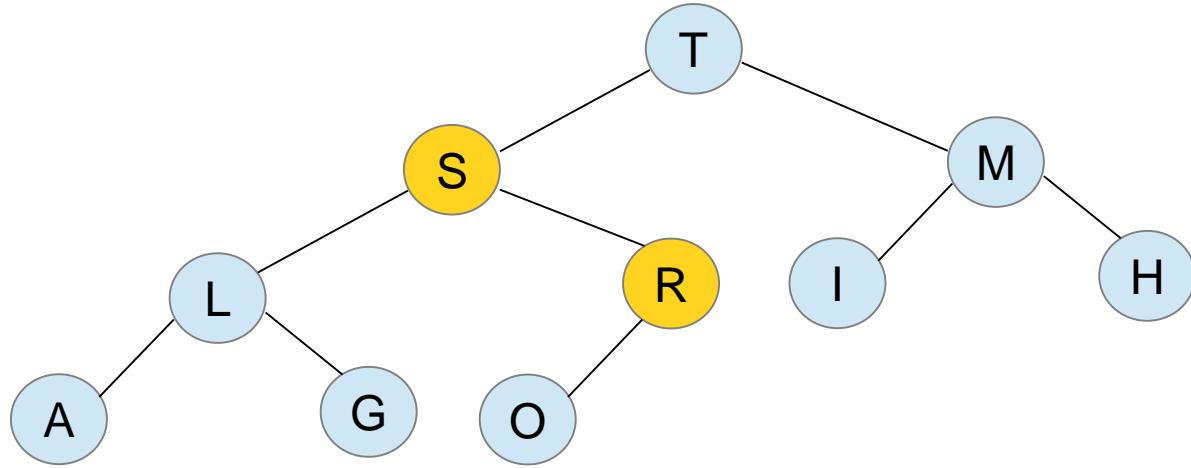
Heap-broken

Heapification.



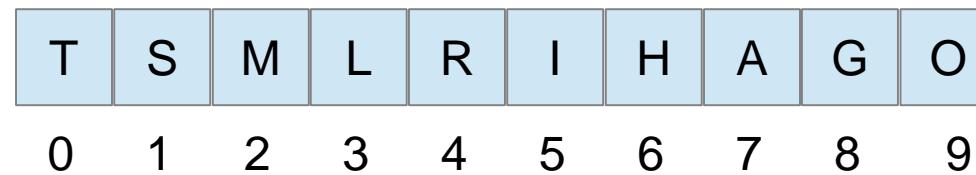
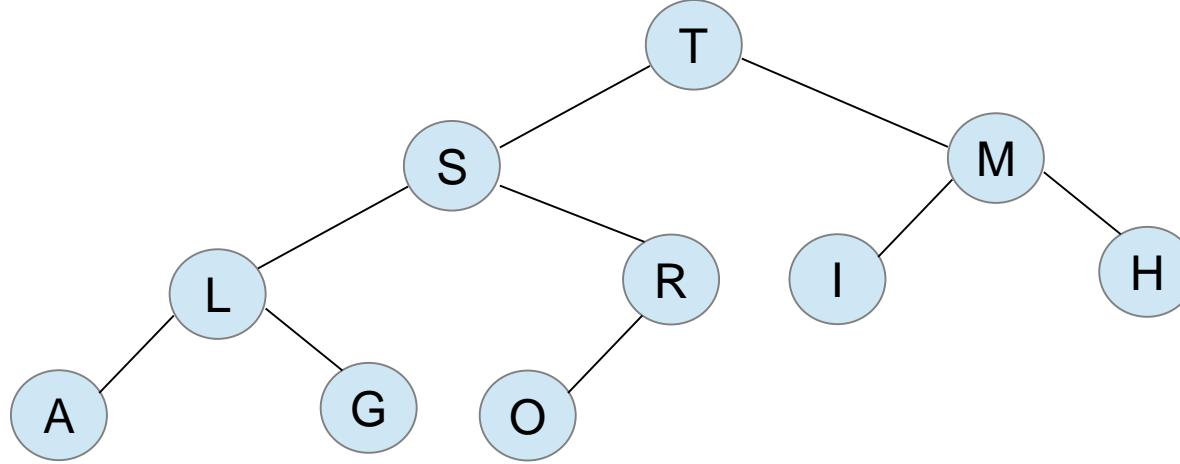
Heap-broken

Heapification.

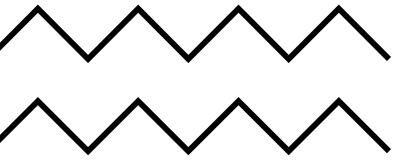


Heap-broken

Heapification.



Heap-fixed.



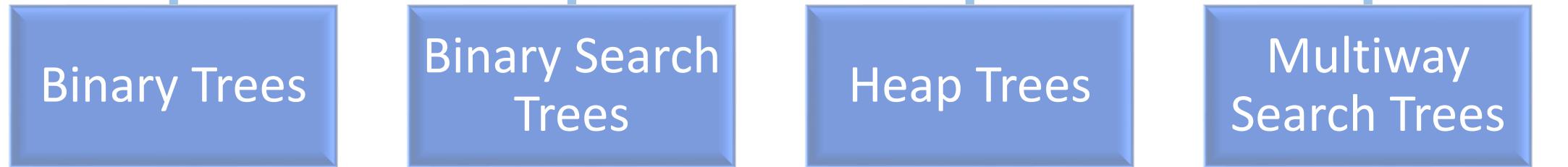
THANK
YOU

Multiway trees.

M-WAY SEARCH TREES.



Types of Trees





M-WAY TREES.

An m-way tree is a ***search tree*** in which each node can have from **0 to m** subtrees, where **m** is defined as the B-tree order.

Given a nonempty multiway tree, we can identify the following properties:

Each node
0 to m
subtrees.

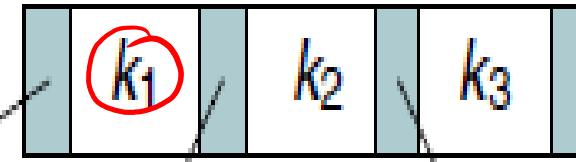
A node with
 $k < m$ subtrees
contains
k subtrees and
 $k - 1$
data entries.

Key values in the
first subtree \leq key value in the first entry
Key values in **other subtrees \geq key value in their parent entry**.

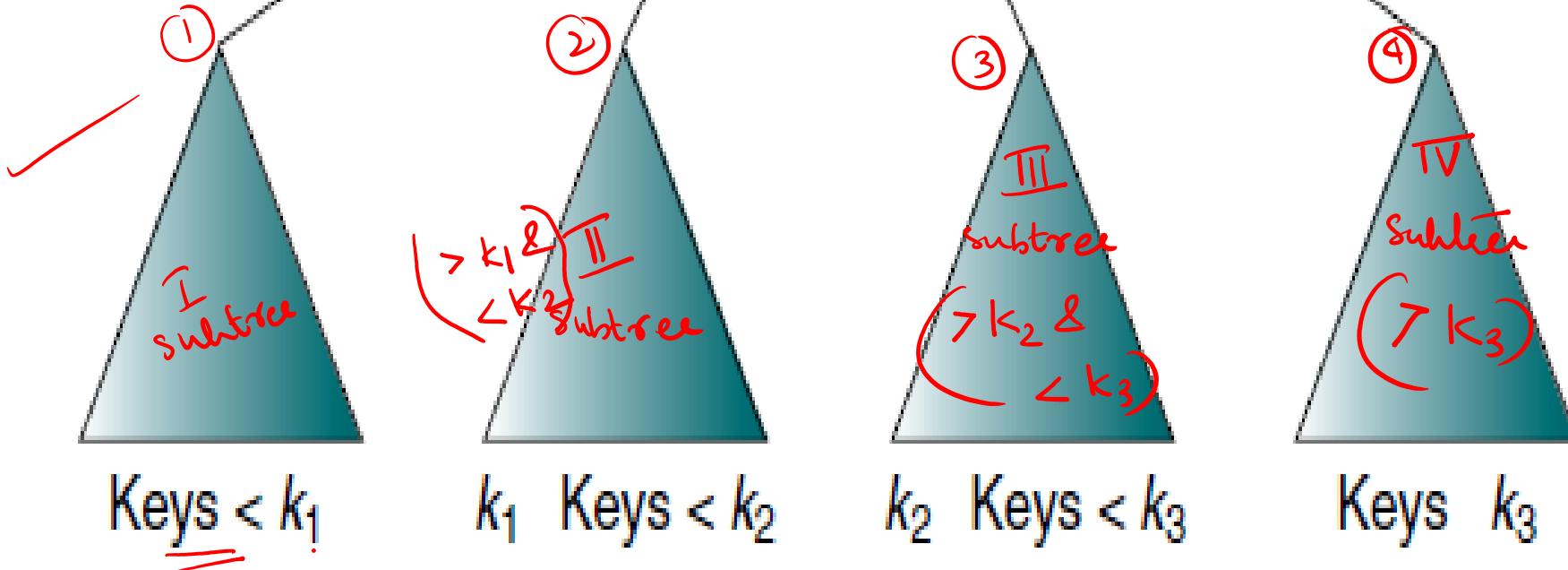
Keys of the
data entries
are ordered
 $key_1 \leq key_2 \leq \dots \leq key_k$.

All subtrees
are themselves
multiway
trees.

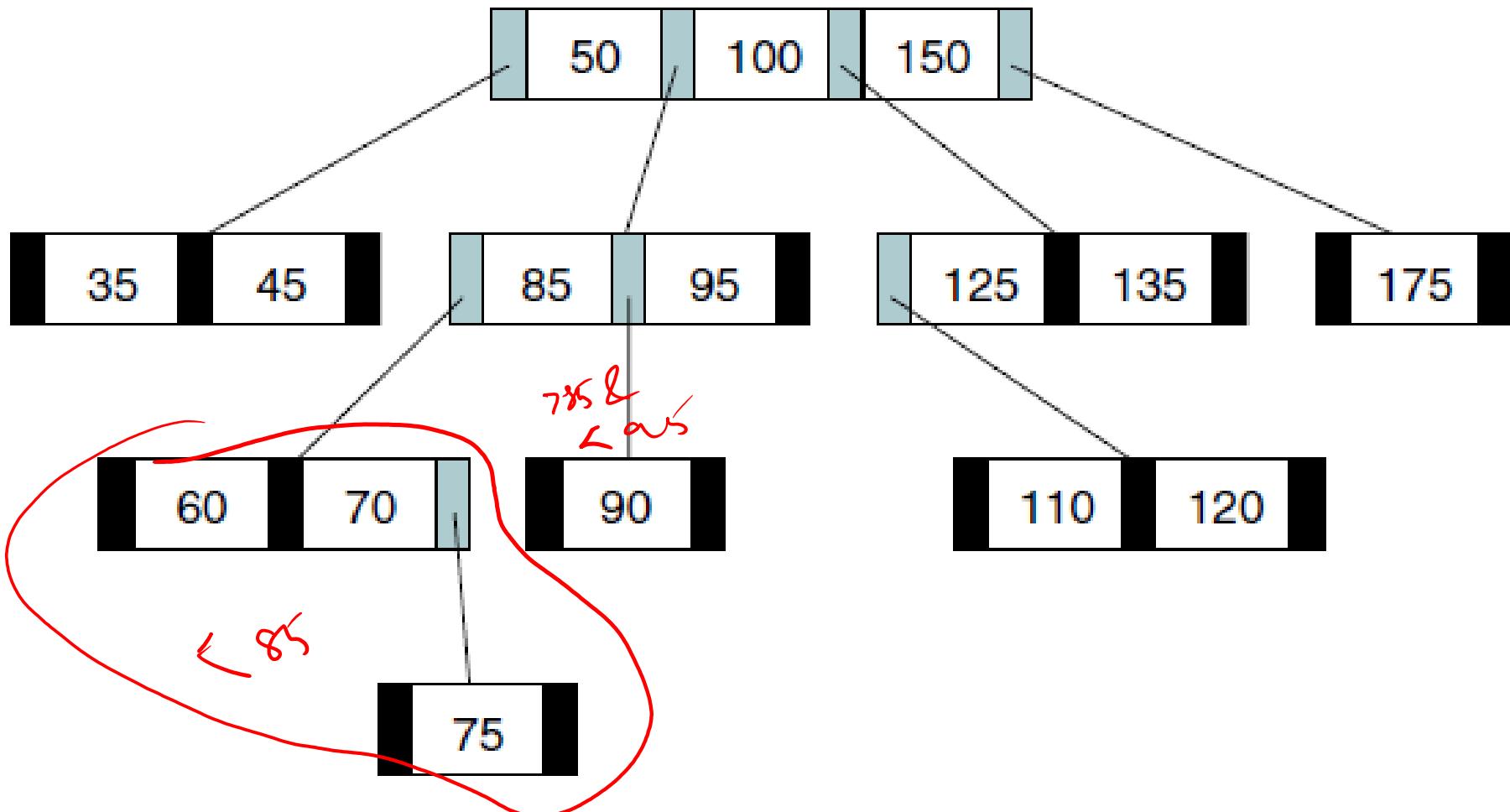
$m=4$
 $m-1 \rightarrow$ keys
= 3 keys
+ subtrees / children



$k_1 \leq k_2 \leq k_3$

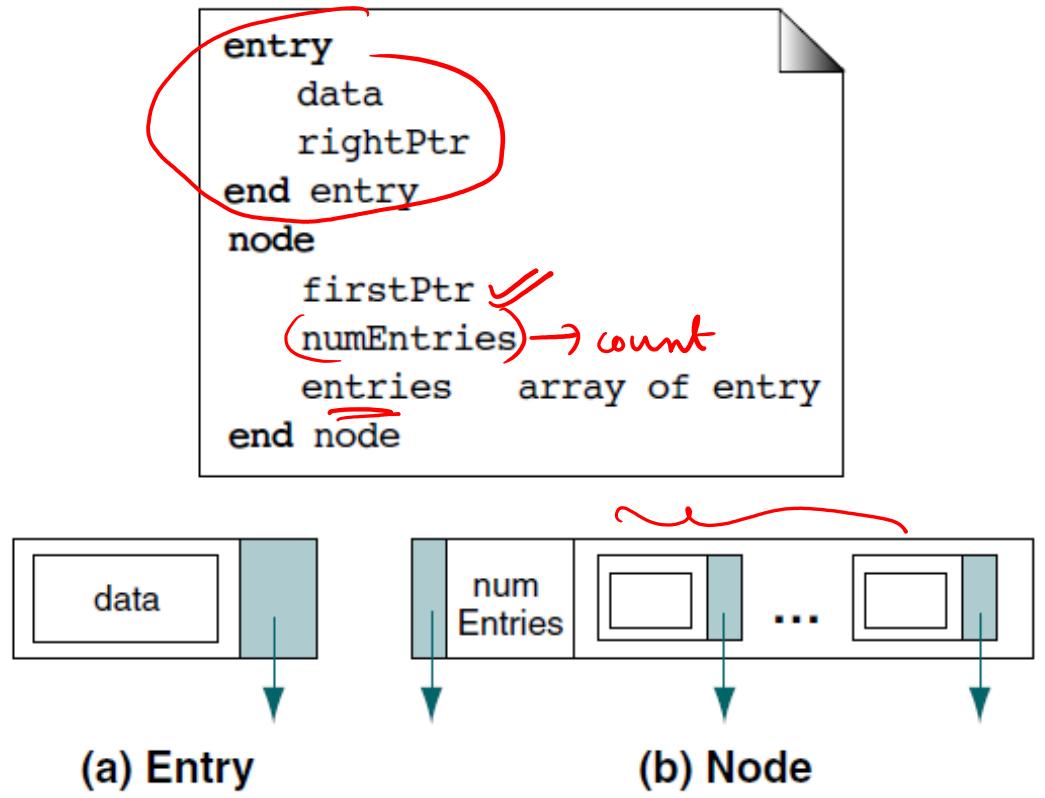


M-way Tree of Order 4



Four-way Tree

Data structure.



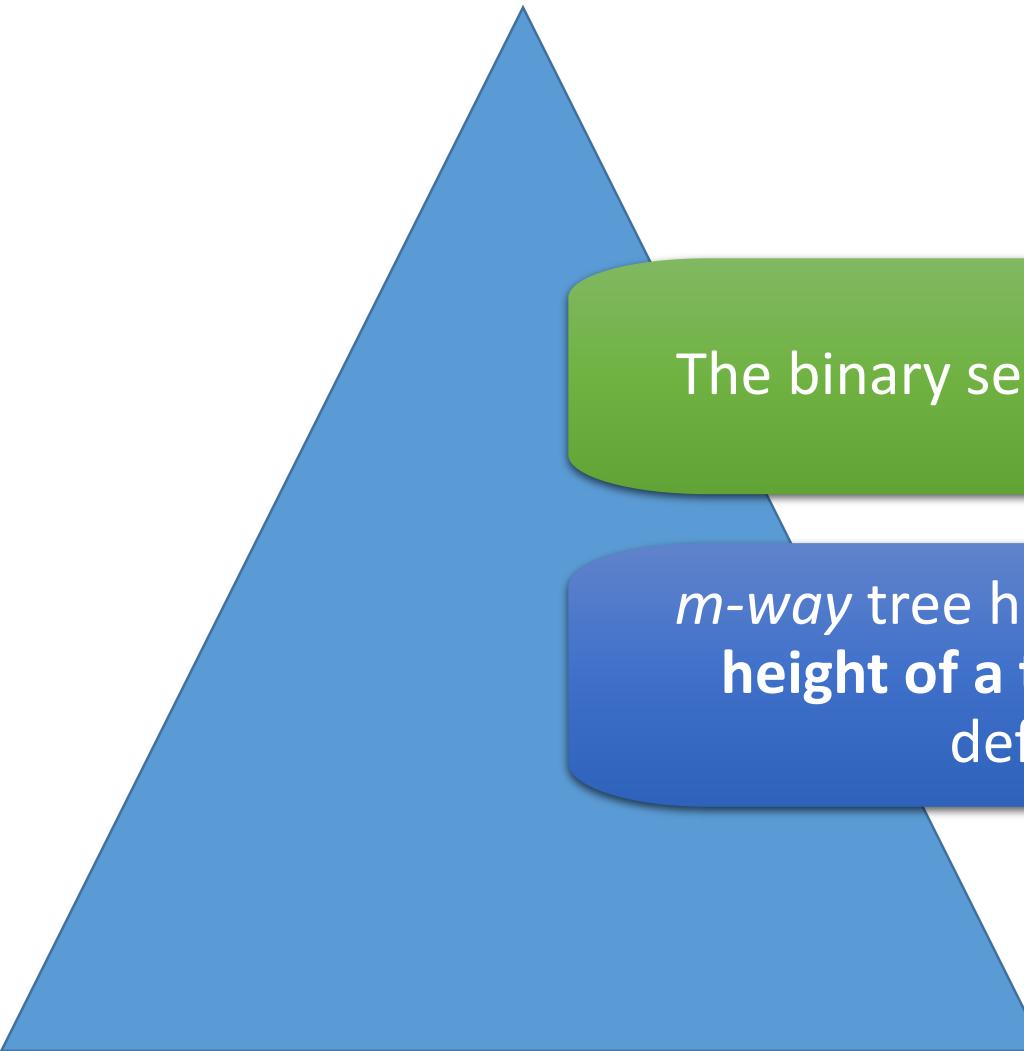
The first pointer to the subtree with entries less than the key of the first entry, a count of the number of entries currently in the node, and the array of entries.

The array must have room for $m - 1$ entries.

order = m

M-way Node Structure

B-TREES.



The binary search tree is an m-way tree of order 2.

m-way tree has the potential to greatly **reduce the height of a tree**. However, it still has one major deficiency: **it is not balanced**.

B-trees.

In 1970, two computer scientists working for the Boeing Company in Seattle, Washington, created a new tree structure they called the **B-tree**.

A ***B-tree is a special m-way search tree*** with the following additional properties:

The root is either a leaf or it has ***2 ... m subtrees***.

All internal nodes have at least ceil of($m/2$) nonnull subtrees and at most ***m nonnull subtrees***.

All leaf nodes are at the same level; that is, the ***tree is perfectly balanced***.

Every node has at least ceil of($m/2$)-1 and at most $m - 1$ entries.(Root min->1 key)

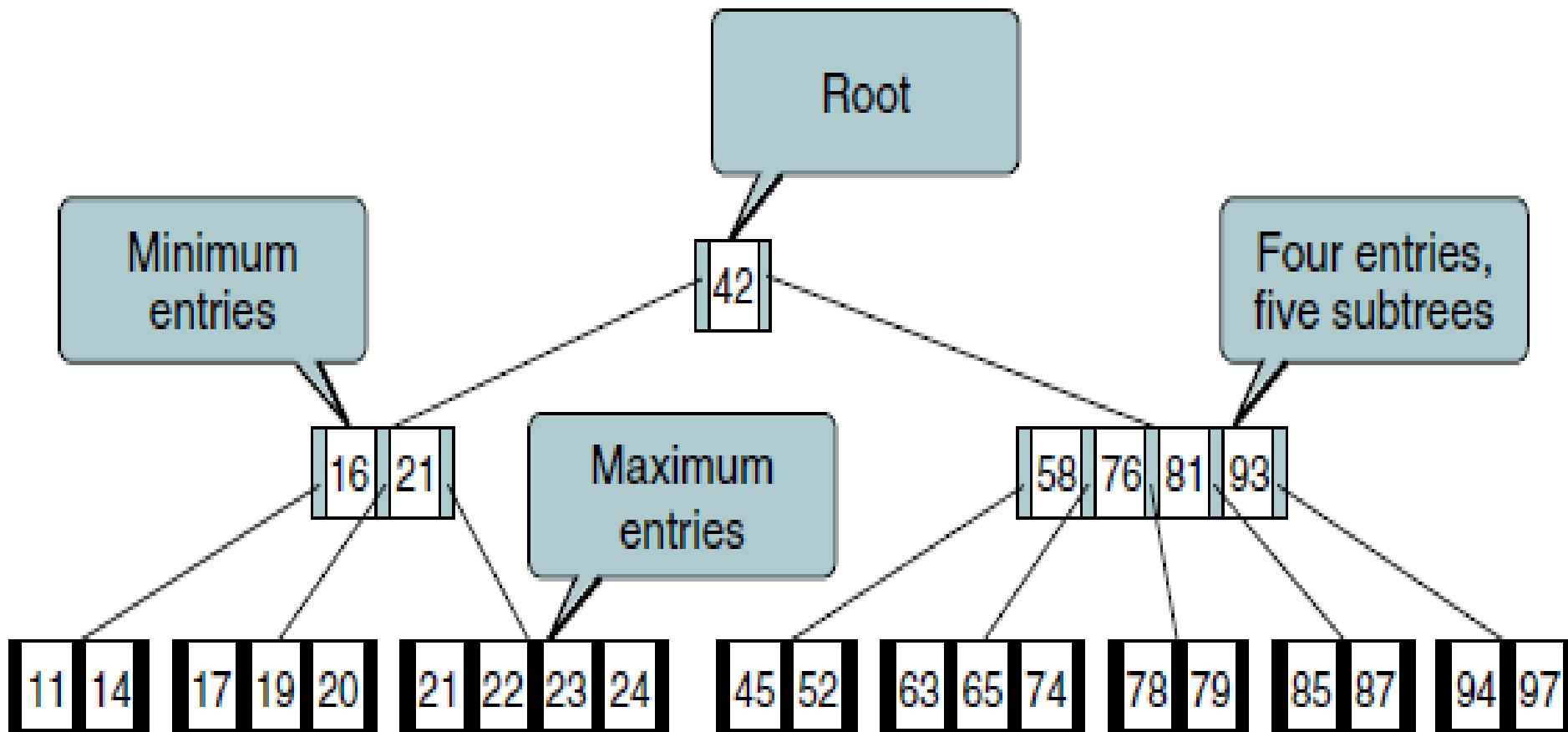
A B-tree is a perfectly balanced m-way tree in which each node, with the possible exception of the root, is at least half full.

minimum and maximum
numbers of subtrees in a non root node for B-trees of different orders.

Order	Number of subtrees		Number of entries	
	Minimum	Maximum	Minimum	Maximum
3	2	3	1	2
4	2	4	1	3
5	3	5	2	4
6	3	6	2	5
...
m	$\lceil m / 2 \rceil$	m	$\lceil m / 2 \rceil - 1$	$m - 1$

Entries in B-trees of Various Orders

Check for m-Way rules.



A B-tree of Order 5

B-tree implementation.

The four basic operations for B-trees are:

Insert

Delete

Traverse

Search

Like the binary search tree,
B-tree insertion takes place at a **leaf node**.

The first step, is to **locate the leaf node** for the data being inserted.

If the node is not full (**overflow**) or if it has fewer than $m - 1$ entries, the new data are simply inserted in sequence in the node.

Overflow requires that the **leaf node be split into two nodes**, each containing half of the data.

Then the median data entry is **inserted into the parent node**.

After the data have been split, the new entry is inserted into either the original or the new node, depending on its key value.

To split the node, we allocate a new node from the available memory and then copy the data from the end of the full node to the new node.

B-trees grow in a balanced fashion from the bottom up.
When the root node of a B-tree overflows and the median entry is pushed up, a new root node is created and the tree grows one level.

B-Tree insertion.

B-tree insertion.



(a) Insert 11



(b) Insert 21



(c) Insert 14



(d) Insert 78



(e) Ready to insert 97



Original node



New data

(f) Create new right subtree



Original node



New node

(g) Insert median into parent (new root)

B-tree deletion.

Three considerations when deleting a data entry from a B-tree node.

First,

Ensure that the data to be deleted are actually in the tree.

Second,

If the node **does not have enough entries after the deletion or *underflow*** correct the structural deficiency.

A deletion that results in a node with fewer than the minimum number of entries is an underflow.

Third (like BST)

Delete only from a leaf node.

Therefore, if the data to be deleted are in an internal node, we must find a data entry to take their place.

B-Tree non-leaf node deletion.

When the data to be deleted are not in a leaf node, ***Find substitute data.***

Two choices for substitute data:

Either the ***immediate predecessor*** or ***immediate successor***.

Either will do, but it is more efficient to use the immediate predecessor because it is always the last entry in a node and no shifting is required when it is deleted. We therefore use the immediate predecessor.

The **immediate predecessor** is the **largest node on the left subtree** of the entry to be deleted.

The **immediate successor** is the **smallest node on the right subtree** of the entry to be deleted.

B-TREE Deletion term: REFLOW

When underflow has occurred, we need to bring the underflowed node up to a minimum state by adding at least one entry to it. This process is known as reflow.

Option 1: Balance

Shifts data among nodes to reestablish the **integrity of the tree**. Because it does not change the structure of the tree, it is less disruptive and therefore preferred.



Rotating an entry from one sibling to another through the parent.

Option 2: Combine

Joins the **data from an underflowed entry, a minimal sibling, and a parent in one node**.

Combine results in one node with the maximum number of entries and an empty node that must be recycled.



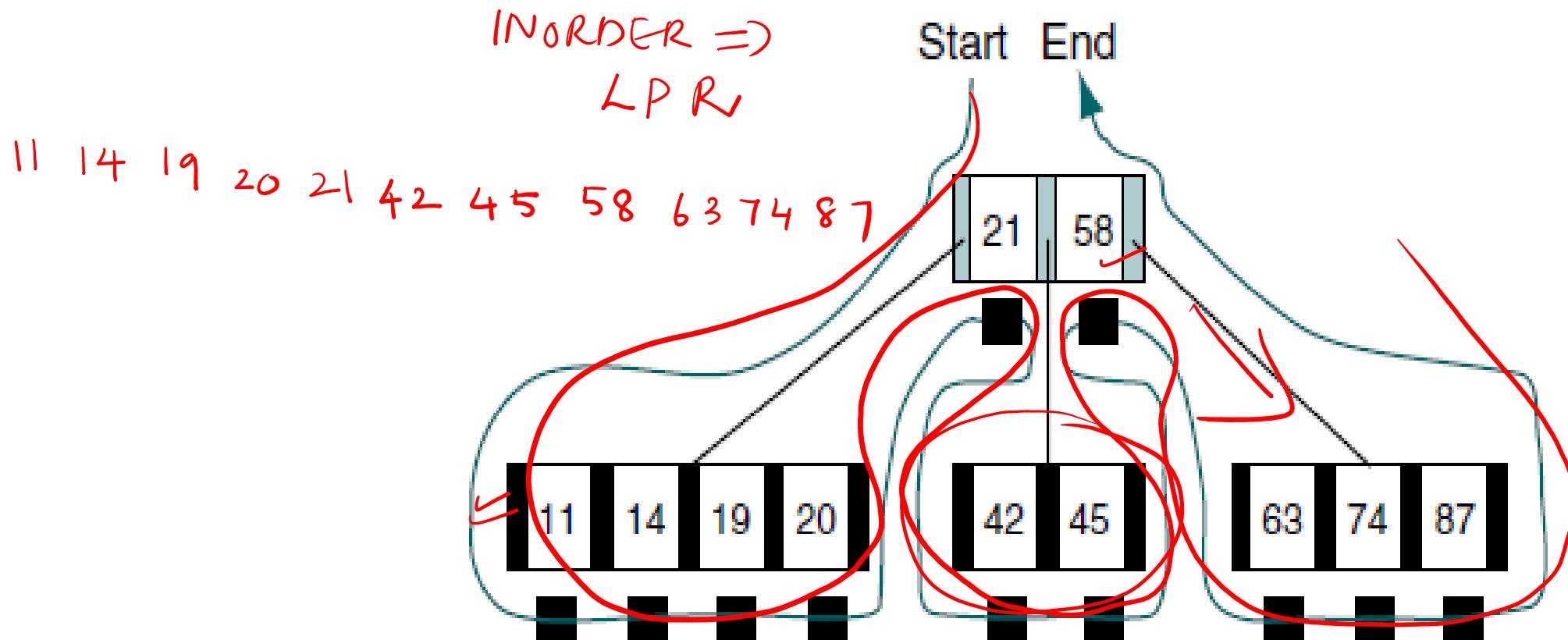
When we can't balance, we must combine nodes.

B-Tree traversal.

Because a B-tree is built on the same structure as the binary search tree, we can use the same basic traversal design: **INORDER traversal**.

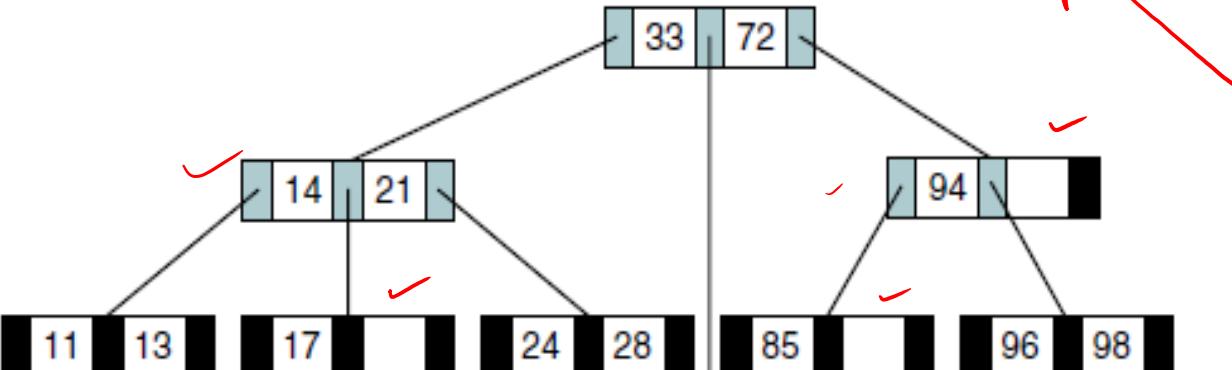
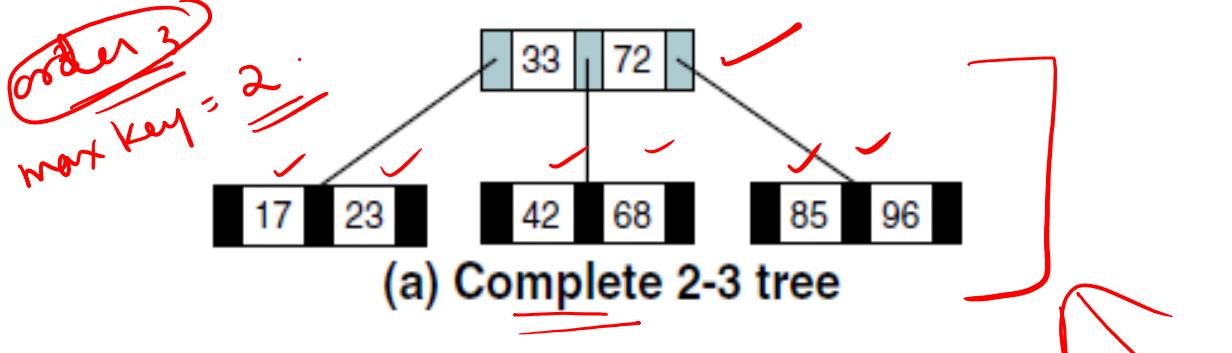
The major difference, however, is that with the *exception of leaf nodes*, we **don't process all of the data in a node at the same time**. Therefore, we must remember which entry was processed whenever we return to a node and continue from that point.

B-Tree traversal.



Basic B-tree Traversal

2-3 TREES.



(b) 2-3 tree with empty entries

Order m
order 3 → 2-3
order 4 → 2 - 3 - 4
order 5 → 2 3 3 - 4 - 5

The 2-3 tree is a B-tree of order 3, 3-4-5

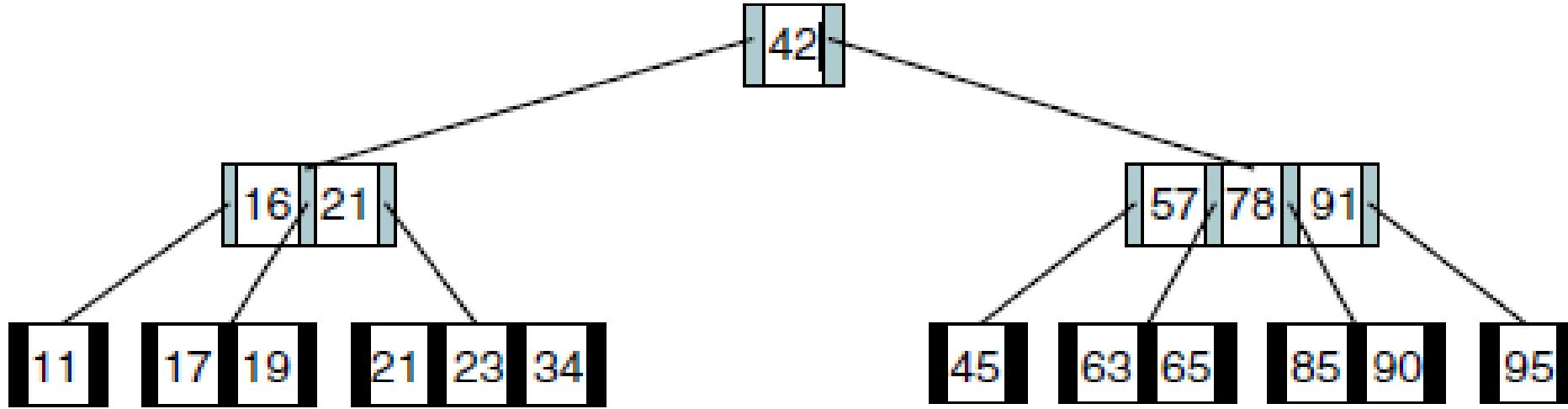
It gets its name because each non-root node has either two or three subtrees (the root may have zero, two, or three subtrees).

The complete 2-3 tree has the maximum number of entries for its height.

Nearly complete 2-3 tree has twice as many entries, but some of the entries are empty. Note also that subtree

Order 5
~~2-3-4 TREES.~~

Order - 4
 $\approx 2-3-4$



A B-tree of order 4 is sometimes called a 2-3-4 tree.

Because each node can have two, three, or four children.

B-TREE VARIATIONS.

B* Tree

B+Tree

Lexical
Search Tree

B-tree deletion.

Three considerations when deleting a data entry from a B-tree node.

First,

Ensure that the data to be deleted are actually in the tree.

Second,

If the node **does not have enough entries after the deletion or *underflow*** correct the structural deficiency.

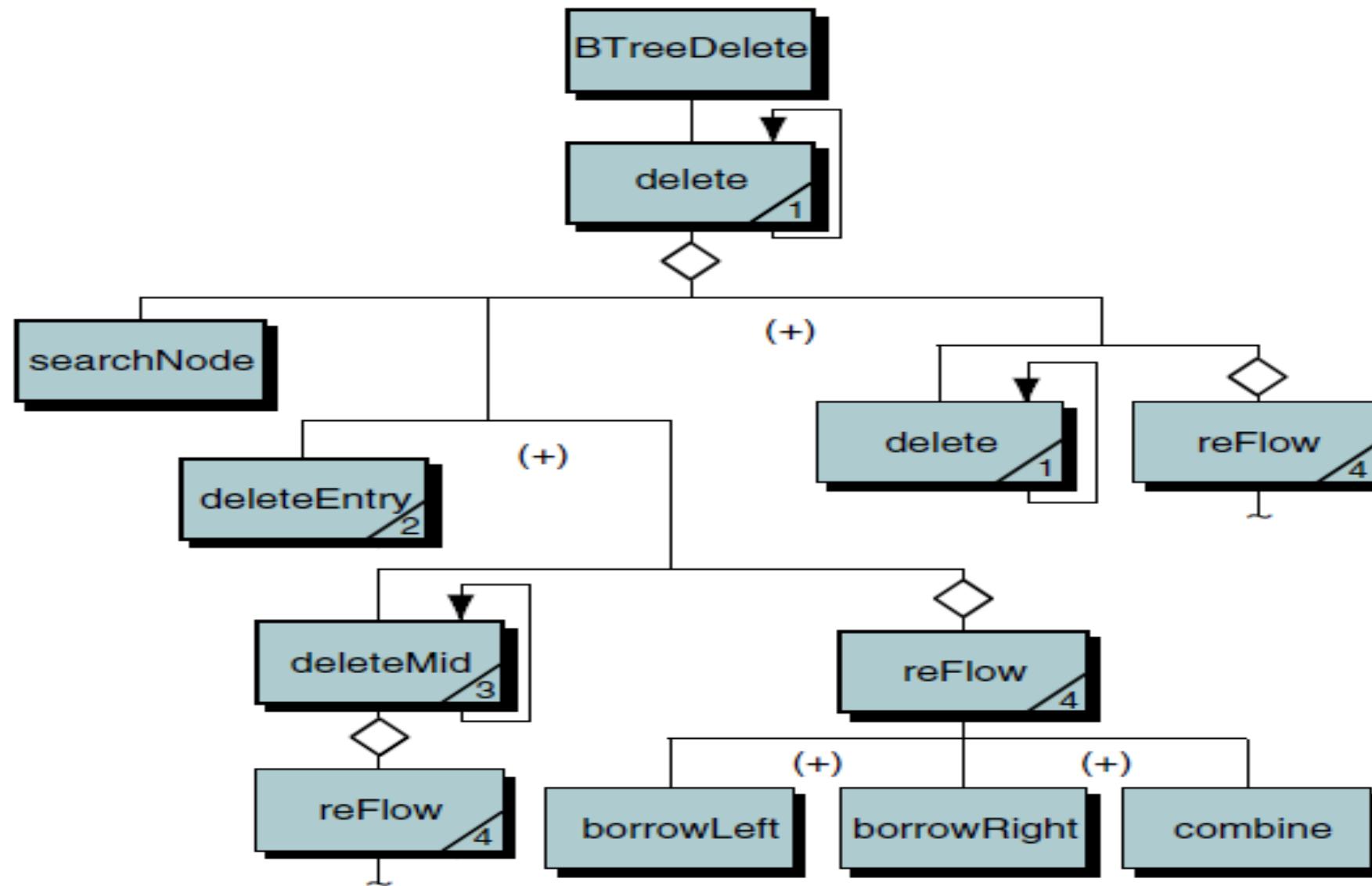
A deletion that results in a node with fewer than the minimum number of entries is an underflow.

Third (like BST)

Delete only from a leaf node.

Therefore, if the data to be deleted are in an internal node, we must find a data entry to take their place.

B-TREE DELETION design.



B-Tree non-leaf node deletion.

When the data to be deleted are not in a leaf node, ***Find substitute data.***

Two choices for substitute data:

Either the ***immediate predecessor*** or ***immediate successor***.

Either will do, but it is more efficient to use the immediate predecessor because it is always the last entry in a node and no shifting is required when it is deleted. We therefore use the immediate predecessor.

The **immediate predecessor** is the **largest node on the left subtree** of the entry to be deleted.

The **immediate successor** is the **smallest node on the right subtree** of the entry to be deleted.

B-TREE Deletion term: REFLOW

When underflow has occurred, we need to bring the underflowed node up to a minimum state by adding at least one entry to it. This process is known as reflow.

Option 1: Balance

Shifts data among nodes to reestablish the **integrity of the tree**. Because it does not change the structure of the tree, it is less disruptive and therefore preferred.



Rotating an entry from one sibling to another through the parent.

Option 2: Combine

Joins the **data from an underflowed entry, a minimal sibling, and a parent in one node**.

Combine results in one node with the maximum number of entries and an empty node that must be recycled.



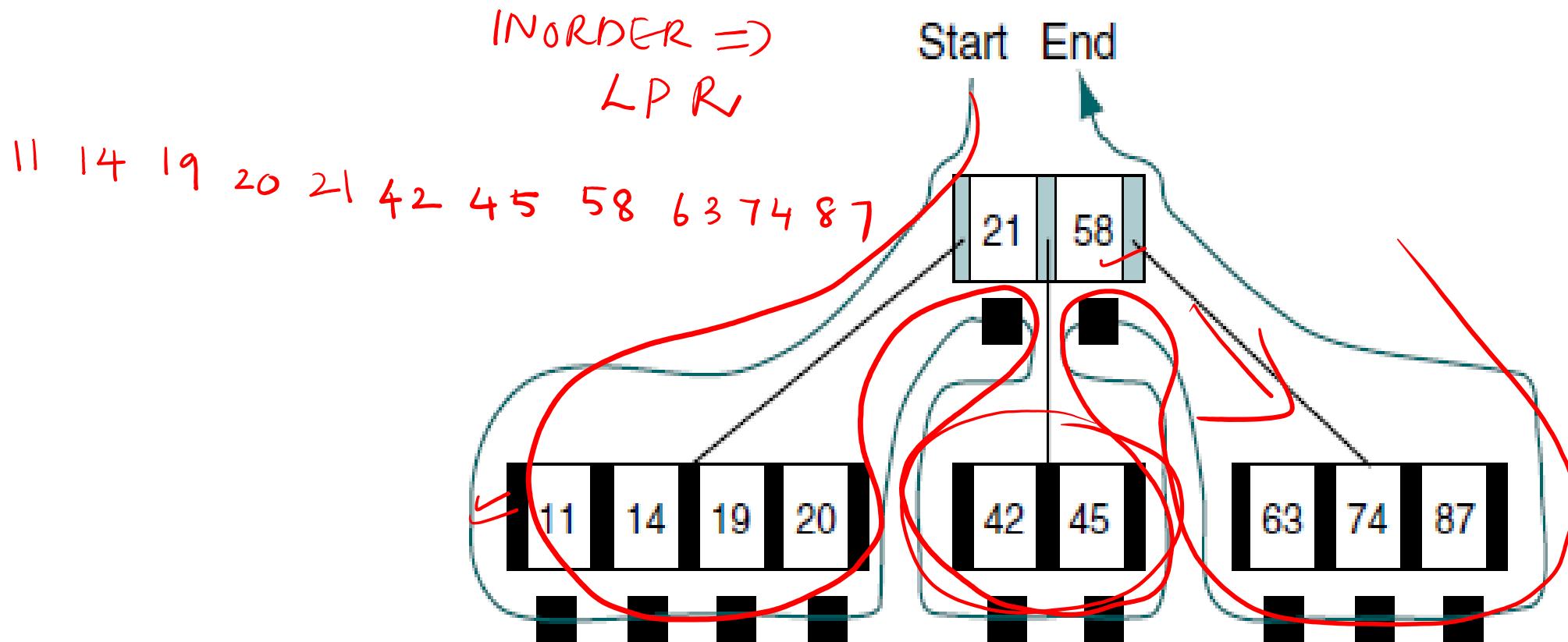
When we can't balance, we must combine nodes.

B-Tree traversal.

Because a B-tree is built on the same structure as the binary search tree, we can use the same basic traversal design: **INORDER traversal**.

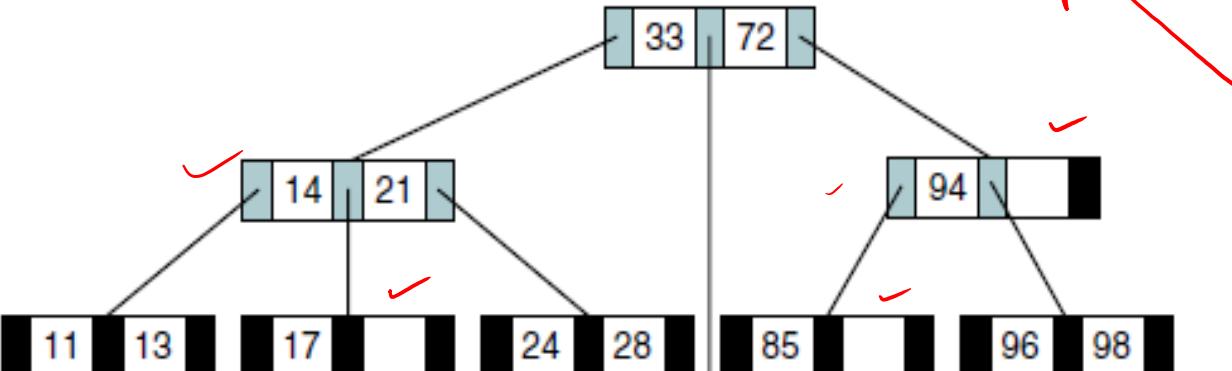
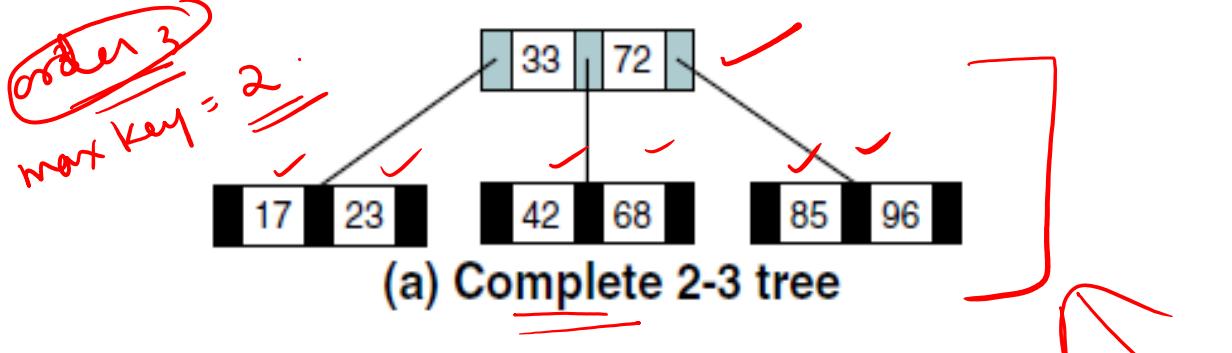
The major difference, however, is that with the *exception of leaf nodes*, we **don't process all of the data in a node at the same time**. Therefore, we must remember which entry was processed whenever we return to a node and continue from that point.

B-Tree traversal.



Basic B-tree Traversal

2-3 TREES.



(b) 2-3 tree with empty entries

Order m
order 3 → 2-3
order 4 → 2 - 3 - 4
order 5 → 2 3 3 - 4 - 5

The 2-3 tree is a B-tree of order 3, 3-4-5

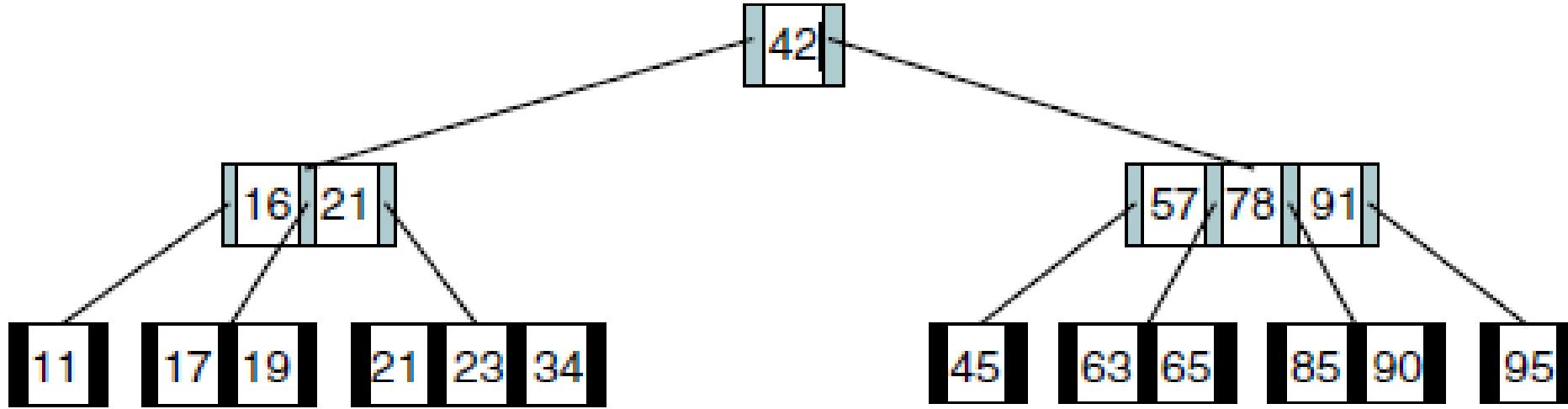
It gets its name because each non-root node has either two or three subtrees (the root may have zero, two, or three subtrees).

The complete 2-3 tree has the maximum number of entries for its height.

Nearly complete 2-3 tree has twice as many entries, but some of the entries are empty. Note also that subtree

Order 5
~~2-3-4 TREES.~~

Order - 4
 $\approx 2 - 3 - 4$



A B-tree of order 4 is sometimes called a 2-3-4 tree.

Because each node can have two, three, or four children.

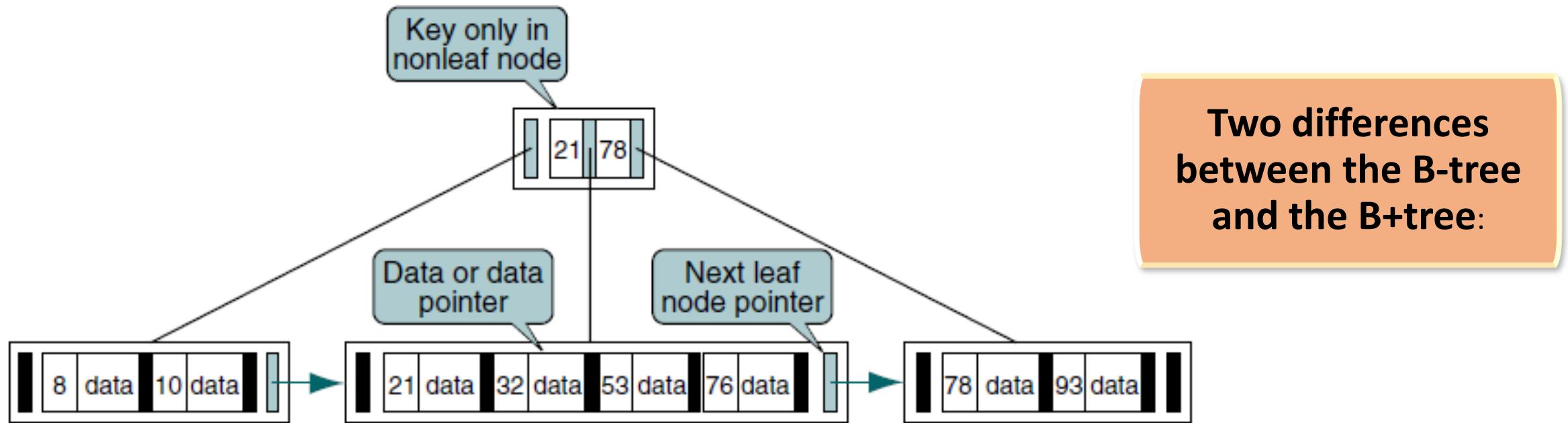
B-TREE VARIATIONS.

B* Tree

B+Tree

Lexical
Search Tree

B+TREE.



Each data entry must be represented at the leaf level, even though there may be internal nodes with the same keys. Because the internal nodes are used only for searching, they generally do not contain data.

Each leaf node has one additional pointer, which is used to move to the next leaf node in sequence.

SORTING & SEARCHING.

Looking for data.



SORTING.

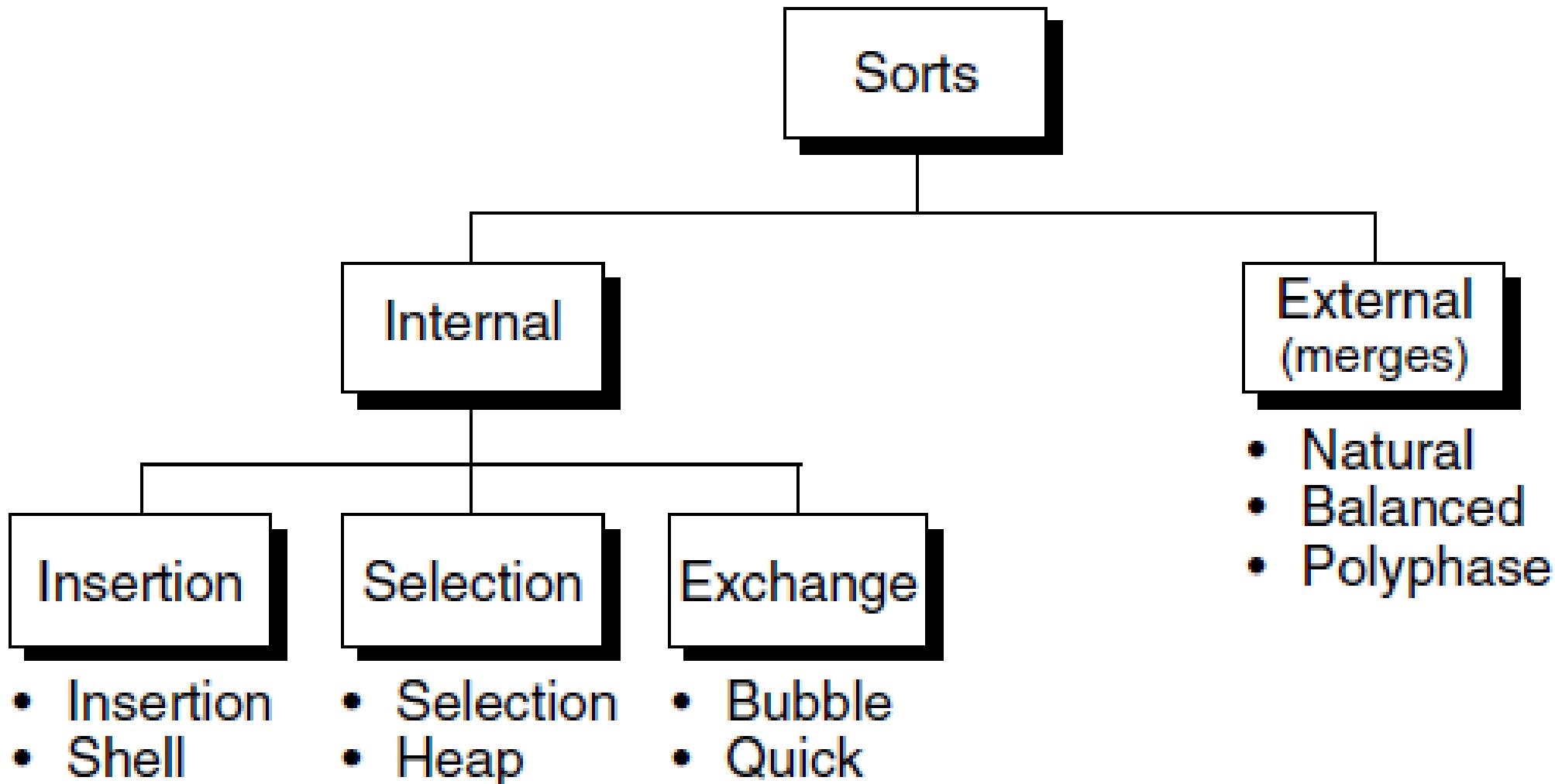
One of the most common data-processing applications.

The process through which data are arranged according to their values is called ***SORTING***.

If data were not ordered, hours spent on trying to find a single piece of information.

Example: The difficulty of finding someone's telephone number in a telephone book that had no internal order.

SORT CLASSIFICATIONS.



TYPES OF SORTS.

All data are held in
primary memory during
the sorting process

Internal

Uses **primary memory**
for the current data
being sorted.

Secondary storage for
data not fitting in
primary memory

External

THREE INTERNAL SORTS.

Selection sort

Insertion sort

Bubble sort

Shell sort
Heap sort
Quick sort

SORT ORDER.

Data may be sorted in either **ascending** or **descending** sequence.

If the order of the sort is not specified, it is **assumed** to be ***ascending***.

Examples of common data sorted in ascending sequence are the dictionary and the telephone book.

Examples of common descending data are percentages of games won in a sporting event such as baseball or grade-point averages for honor students.

SORT STABILITY.

Is an attribute of a sort, indicating that **data with equal keys maintain their relative input order in the output.**

input
order

365	blue
212	green
876	white
212	yellow
119	purple
737	green
212	blue
443	red
567	yellow

(a) Unsorted data

119	purple
212	green
212	yellow
212	blue
365	blue
443	red
567	yellow
737	green
876	white

(b) Stable sort

119	purple
212	blue
212	green
212	yellow
365	blue
443	red
567	yellow
737	green
876	white

output

(c) Unstable sort

SORT EFFICIENCY.

Is a measure of the relative efficiency of a sort, usually an estimate of the number of comparisons and moves required to order an unordered list.

Best possible sorting algorithms are the **$O(n \log n)$** sorts.

PASSES.

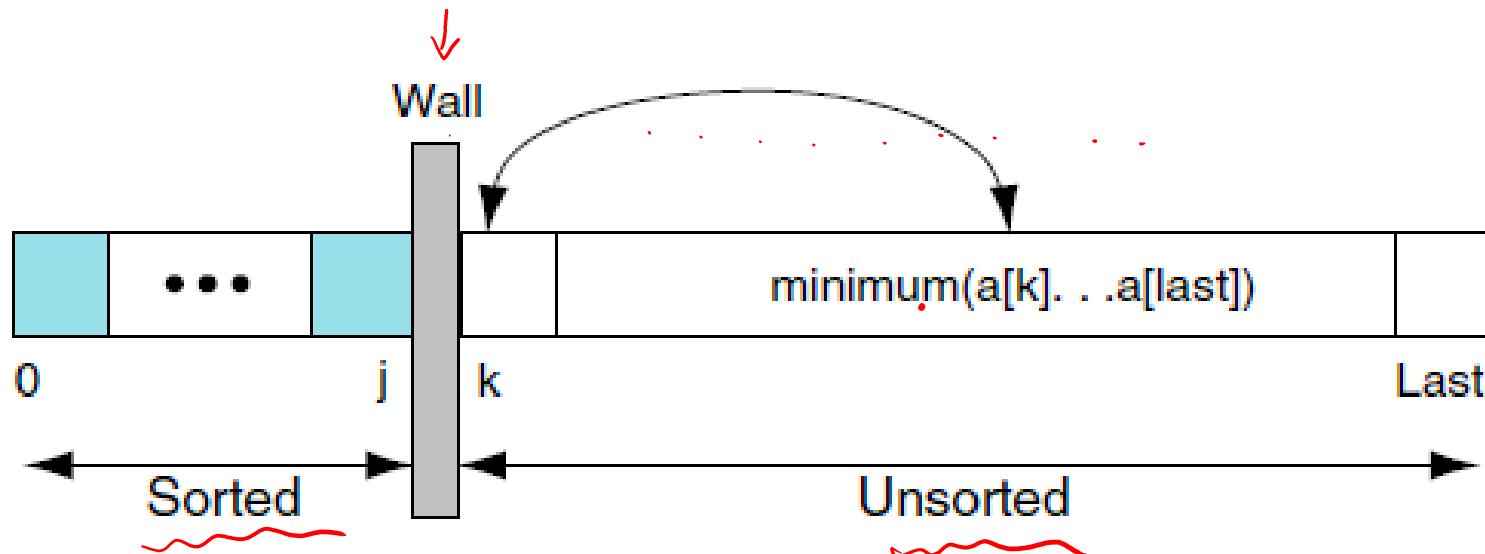
During the sorting process, the data are traversed many times. Each traversal of the data is referred to as a **sort pass**.

Depending on the algorithm, the sort pass may traverse the whole list or just a section of the list.

Also, a characteristic of a sort pass is the placement of one or more elements in a sorted list.

SELECTION SORT.

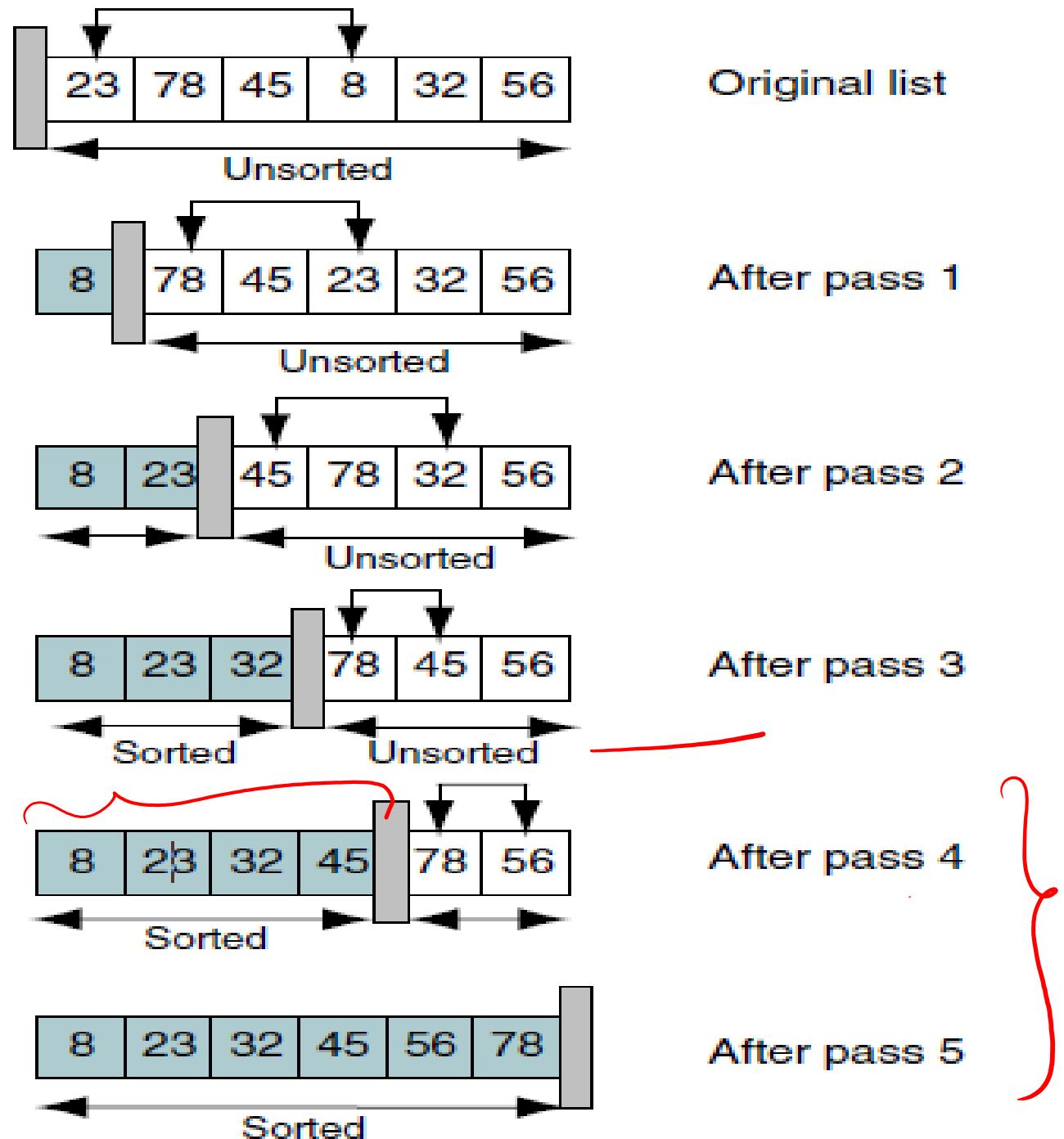
- x In each pass of the selection sort, the smallest element is selected from the unsorted sublist and exchanged with the ~~element at the beginning of the unsorted list~~.
- x If there is a list of ~~n~~ elements, therefore, ~~n - 1~~ passes are needed to completely rearrange the data.



SELECTION SORT.

5 3 4 1 2

Example 2



Selection Sort

```
Algorithm selectionSort (list, last)
Sorts list array by selecting smallest element in
unsorted portion of array and exchanging it with element
at the beginning of the unsorted list.

Pre  list must contain at least one item
      last contains index to last element in the list
Post list has been rearranged smallest to largest

1 set current to 0
2 loop (until last element sorted)
  1 set smallest to current
  2 set walker to current + 1
  3 loop (walker <= last)
    1 if (walker key < smallest key)
      1 set smallest to walker
      2 increment walker
  4 end loop
    Smallest selected: exchange with current element.
  5 exchange (current, smallest)
  6 increment current
3 end loop
end selectionSort
```

{Selection Sort} 5 passes

$n=6$

0	1	2	3	4	5
23	78	45	8	32	56

I Pass

$i \ min \ j \ j < n \ a[j] < a[min] \ exchange$

0 0 1 $1 < 6$ ✓ $78 < 23$ F -

2 2 $2 < 6$ $45 < 23$ F -

3 3 $3 < 6$ ✓ $8 < 23$ T -

4 4 $4 < 6$ $32 < 8$ F -

5 5 $5 < 6$ $56 < 8$ F -

6 6 $6 < 6$ F exchange(0,3)

min
min

Algorithm

$i \leftarrow 0$

Loop until last element sorted

{

$min \leftarrow i$

$j \leftarrow i+1$

Loop ($j < n$)

{

if ($a[j] < a[min]$)
{ $min \leftarrow j$ }

j++ ✓

exchange(i, min)
i++

8 23 32 45 56 78

II Pass

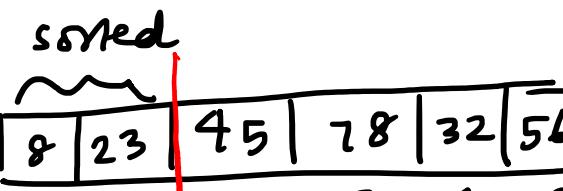
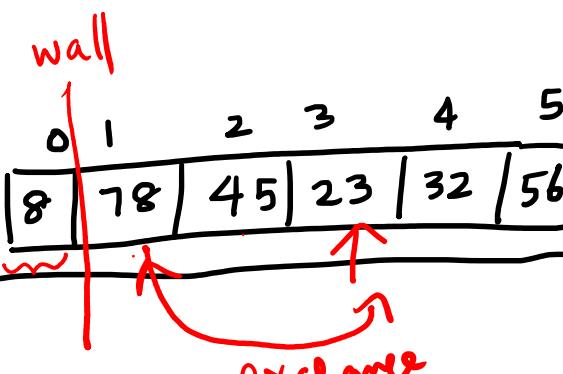
1 1 2 $2 < 6$ $45 < 18$ T

2 3 $3 < 6$ $23 < 45$ T

3 4 $4 < 6$ $32 < 23$ F

5 5 $5 < 6$ $56 < 23$ F

6 6 $6 < 6$ F exchange(1, 3)

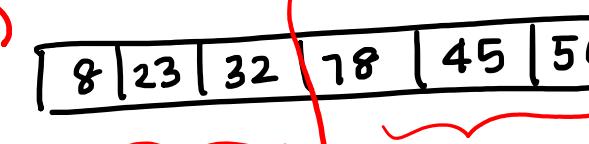


III Pass

2 2 3 $3 < 6$ $78 < 45$ F

4 4 $4 < 6$ $32 < 45$ T

5 5 $5 < 6$ $56 < 32$ F, 6 →

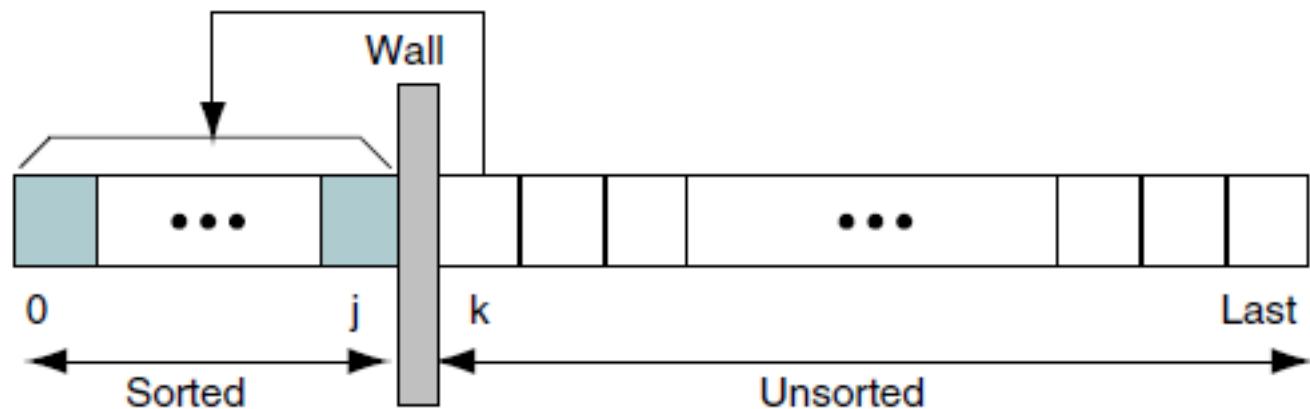


Selection Sort Efficiency.

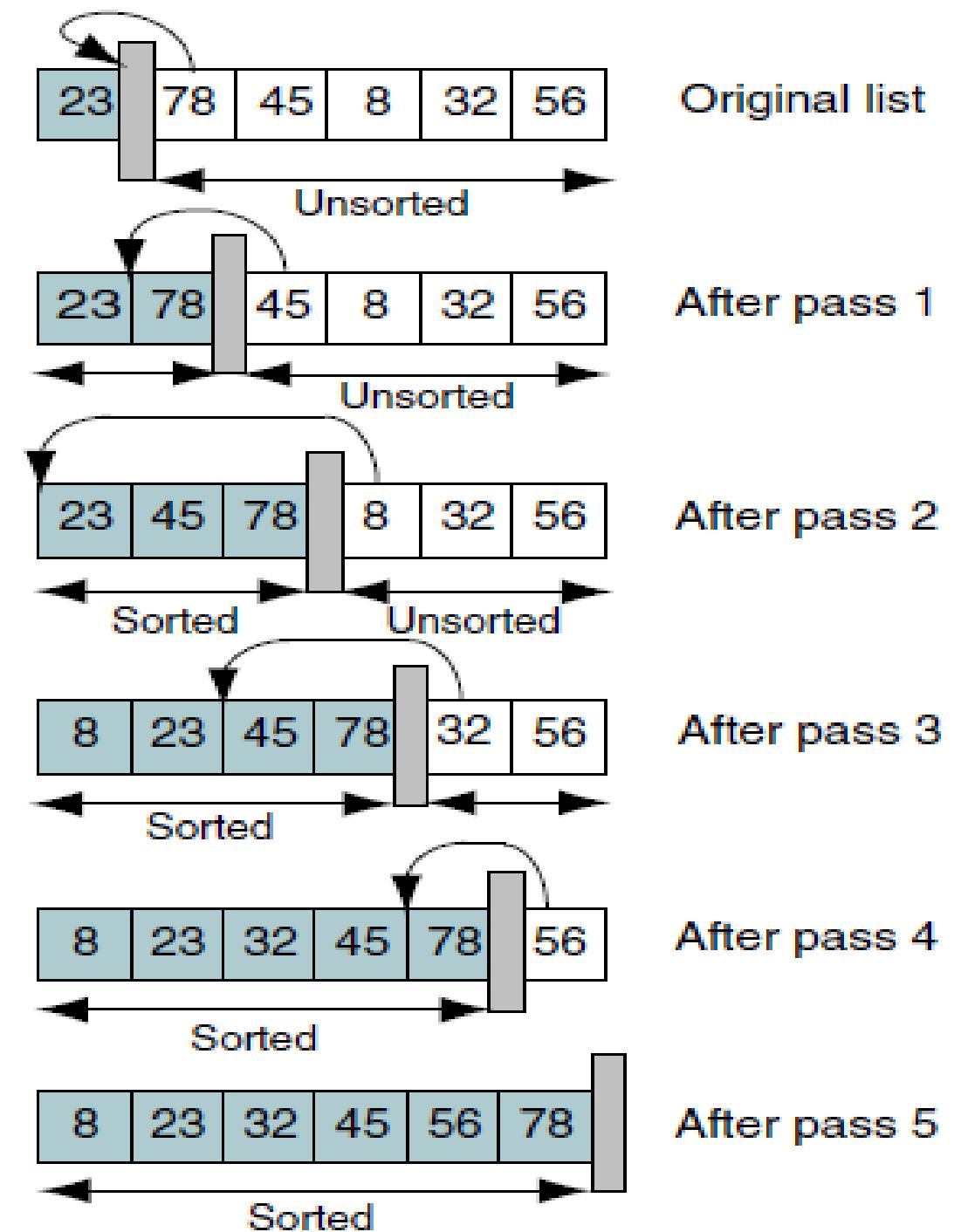
- x It contains the two loops.
- x The outer loop executes $n - 1$ times. The inner loop also executes $n - 1$ times.
- x This is a classic example of the quadratic loop. Its search effort therefore, using big-O notation, is $O(n^2)$.

INSERTION SORT.

- x Given a list, it is divided into two parts: sorted and unsorted.
- x In each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.
- x If list has n elements, it will take at most $n - 1$ passes to sort the data.



INSERTION SORT.



Straight Insertion Sort

```
Algorithm insertionSort (list, last)
Sort list array using insertion sort. The array is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list.

    Pre  list must contain at least one element
        last is an index to last element in the list
    Post list has been rearranged

1 set current to 1
2 loop (until last element sorted)
    1 move current element to hold
    2 set walker to current - 1
    3 loop (walker >= 0 AND hold key < walker key)
        1 move walker element right one element
        2 decrement walker
    4 end loop
    5 move hold to walker + 1 element
    6 increment current
3 end loop
end insertionSort
```

$n=6$, no. of pass = 5

Original
 $\Rightarrow 78 = 78$

0	1	2	3	4	5
23	78	45	8	32	56

$i \ a[i] \ t \ j \ j \geq 0 \ \& \ t < a[j]$

1 78 78 0 $0 \geq 0 \ \& \ 78 < 23$ F

2 45 45 1 $1 \geq 0 \ \& \ 45 < 78$ T
 $0 \geq 0 \ \& \ 45 < 23$ F

3 8 8 2 $2 \geq 0 \ \& \ 8 < 78$ T
 $1 \geq 0 \ \& \ 8 < 45$ T
 $0 \geq 0 \ \& \ 8 < 23$ T

4 32 32 3 $3 \geq 0 \ \& \ 32 < 78$ T
 $2 \geq 0 \ \& \ 32 < 45$ T
 $1 \geq 0 \ \& \ 32 < 23$ F

5 56 56 4 $4 \geq 0 \ \& \ 56 < 78$ T
 $3 \geq 0 \ \& \ 56 < 45$ F \Rightarrow

$i=1$
 $\Rightarrow 78 = 78$

0	1	2	3	4	5
23	78	45	8	32	56

0	1	2	3	4	5
23	45	78	9	32	56

0	1	2	3	4	5
23	45	78	9	32	56

0	1	2	3	4	5
8	23	45	78	32	56

0	1	2	3	4	5
8	23	32	45	78	56

Algorithm

$i=1 \ \{ \text{2nd element} \}$

loop until last ele sorted
 $\{$
 $t \leftarrow a[i]$
 $j \leftarrow i-1$

loop ($j \geq 0 \ \& \ t < a[j]$)
 $\{$
 $a[j+1] = a[j]$
 $j--$

$\rightarrow a[j+1] = t // \text{Always}$
 $i++$

8 23 32 45 56 78

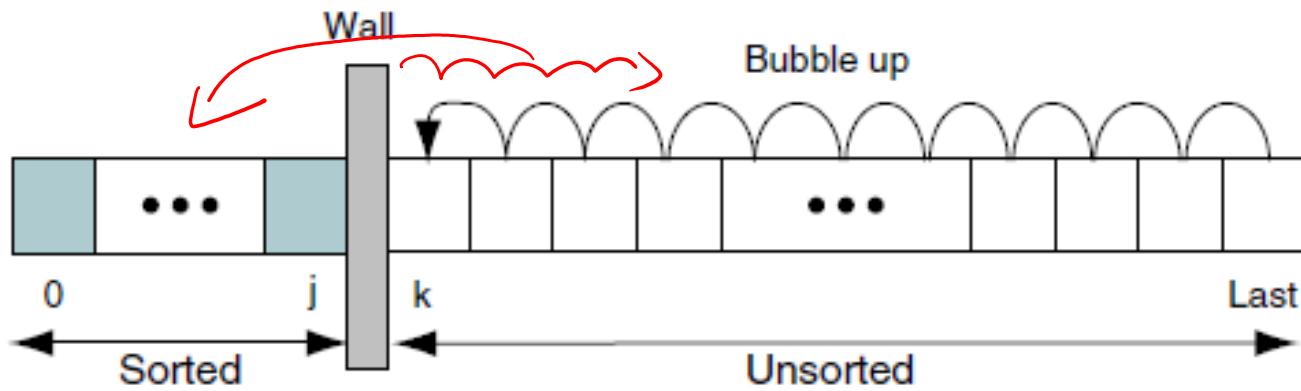
INSERTION SORT EFFICIENCY.

- x The outer loop executes $n - 1$ times, from 1 through the last element in the list.
- x For each outer loop, the inner loop executes from 0 to current times, depending on the relationship between the hold key and the walker key.
- x On the average, the inner loop processes through the data in half of the sorted list. Because the inner loop depends on the setting for current, which is controlled by the outer loop, we have a dependent quadratic loop, which is mathematically stated as

$$f(n) = n \left(\frac{n + 1}{2} \right)$$

- x In big-O notation the dependent quadratic loop is $O(n^2)$.
- x **Therefore, the insertion sort efficiency is $O(n^2)$.**

BUBBLE SORT.

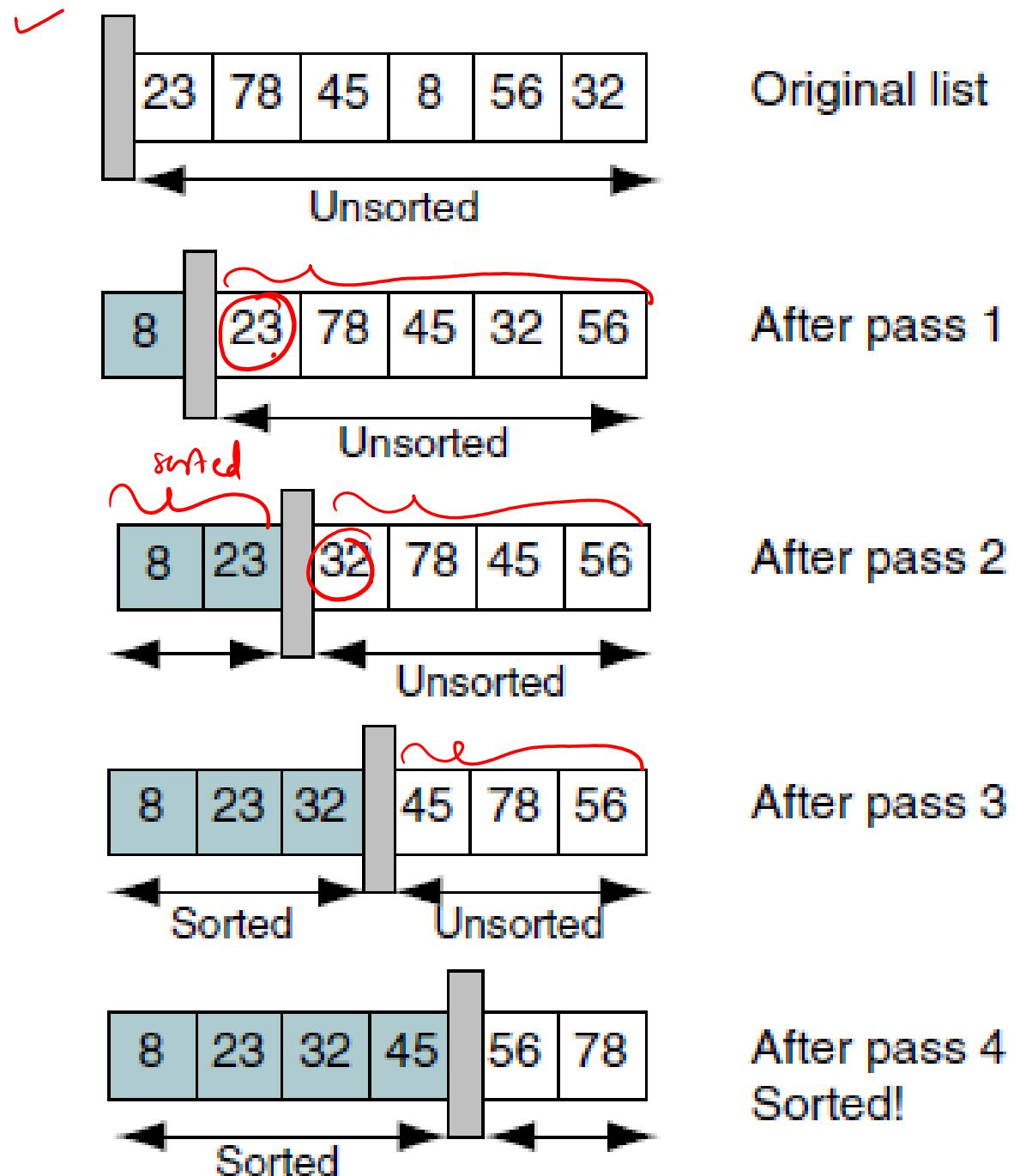


- x The list is divided into two sublists: sorted and unsorted.
- x The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist.
- x After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the number of unsorted ones.
- x Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed.
- x Given a list of n elements, the bubble sort requires up to $n - 1$ passes to sort the data.

BUBBLE SORT.

or:

6 5 3 1 8 7 2 4



```
Algorithm bubbleSort (list, last)
Sort an array using bubble sort. Adjacent
elements are compared and exchanged until list is
completely ordered.
```

```
    Pre list must contain at least one item
        last contains index to last element in the list
    Post list has been rearranged in sequence low to high
1 set current to 0
2 set sorted to false
3 loop (current <= last AND sorted false)
    Each iteration is one sort pass.
    1 set walker to last
    2 set sorted to true
    3 loop (walker > current)
        1 if (walker data < walker - 1 data)
            Any exchange means list is not sorted.
            1 set sorted to false
            2 exchange (list, walker, walker - 1)
        2 end if
        3 decrement walker
    4 end loop
    5 increment current
4 end loop
end bubbleSort
```

i sorted $j > i \ a[j] < a[j-1]$ exchange

0 F 5 5 > 0 T 32 < 56 T
 \downarrow
 0 < 5 F 4 4 > 0 T 32 < 8 F

3 3 > 0 T 8 < 45 T exchange(3,2)

2 2 > 0 T 8 < 78 T exchange(2,1)

1 1 > 0 T 8 < 23 T exchange(1,0)

0 0 > 0 F

$n=6$

23	78	45	8	56	32
0	1	2	3	4	5

23	78	45	8	32	56
0	1	2	3	4	5

23	78	8	45	32	56
0	1	2	3	4	5

23	8	78	45	32	56
0	1	2	3	4	5

8	23	78	45	32	56
0	1	2	3	4	5

Algorithm

$i < 0$

sorted \leftarrow false

Loop ($i \leq n-1 \ \& \ \& \ \text{sorted} = \text{false}$)
 {

$j \leftarrow n-1$

sorted \leftarrow true

loop ($j > i$)
 {

if ($a[j] < a[j-1]$)

{ sorted \leftarrow false

exchange($a, j, j-1$)

}

$j--$

}

$i++$

$n=6$

$n-1$

= 5

BUBBLE SORT EFFICIENCY.

- x Uses two loops to sort the data.
- x The outer loop tests two conditions: the current index and a sorted flag.
- x Assuming that the list is not sorted until the last pass, we loop through the array n times. The number of loops in the inner loop depends on the current location in the outer loop. It therefore loops through half the list on the average. The total number of loops is the product of both loops, making the bubble sort efficiency to

$$n \left(\frac{n+1}{2} \right) \quad n^2$$

- x **The bubble sort efficiency is $O(n^2)$.**

QUICK SORT.

- x In Quick sort, also an exchange sort method, developed by C. A. R. Hoare in 1962.
- x Quick sort is an exchange sort in which a pivot key is placed in its correct position in the array while rearranging other elements widely dispersed across the list.
- x More efficient than the bubble sort because a typical exchange involves *elements that are far apart*, so *fewer exchanges* are required to correctly position an element.

QUICK SORT.

- x Also called **partition-exchange sort**.
- x **Each iteration** of the quick sort **selects an element**, known as **pivot**, and divides the list into three groups:
 - **Partition of elements** whose **keys are less than the pivot's key**,
 - **Pivot element** placed in its ultimately correct location in the list,
 - **Partition of elements** greater than or equal to the **pivot's key**.
- x **Pivot** element can be **any element from the array**, it can be the **first** element, the **last** element or any **random** element.
- x Approach is **recursive**.

LOGIC OF PARTITION.

- x In the array $\{52, 37, 63, 14, 17, 8, 6, 25\}$, 25 is taken as **pivot**.
- x **First pass:** $\{6, 8, 17, 14, 25, 63, 37, 52\}$
- x After the first pass, **pivot** will be **set at its position** in the final sorted array, with all the **elements smaller** to it on its **left** and all the **elements larger** than to its **right**.
- x Next, $\{6 \ 8 \ 17 \ 14\}$ and $\{63 \ 37 \ 52\}$ are considered as two separate subarrays
- x Same **recursive logic** will be applied on them, and keep doing this until the complete array is sorted.

HOW DOES QUICK SORT WORK?

Selection of pivot to partition the array

Pivot (key) = First element; Find a position (for its final correct position)

Left partition < pivot, Right partition \geq pivot

Repeat recursively for Left and Right partitions

Can be used for finding the k^{th} smallest OR largest element in an array without completely sorting the array of size n.

When the pivot element is placed in $(k-1)^{\text{th}}$ position, it will be the k^{th} smallest element

When the pivot element is placed in $(n-k)^{\text{th}}$ position, it will be the k^{th} largest element

QUICK SORT ALGORITHM.

- x Two Algorithms:
 - Quick Sort Recursive
 - Partition

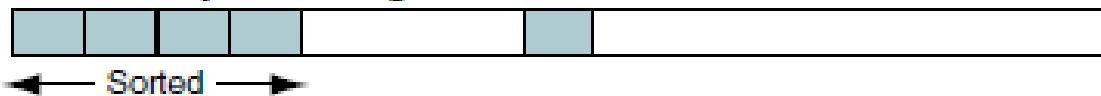
After first partitioning



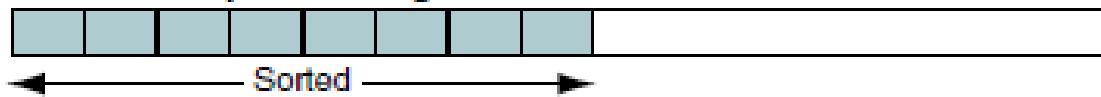
After second partitioning



After third partitioning



After fourth partitioning



After fifth partitioning



After sixth partitioning



After seventh partitioning



QUICK SORT EXAMPLES.

0	15	12	3	21	25	3	9	8	18	28	5
---	----	----	---	----	----	---	---	---	----	----	---

1	9	12	3	5	8	3	15	25	18	28	21
---	---	----	---	---	---	---	----	----	----	----	----

2	8	3	3	5	9	12	15	21	18	25	28
---	---	---	---	---	---	----	----	----	----	----	----

3	5	3	3	8	9	12	15	18	21	25	28
---	---	---	---	---	---	----	----	----	----	----	----

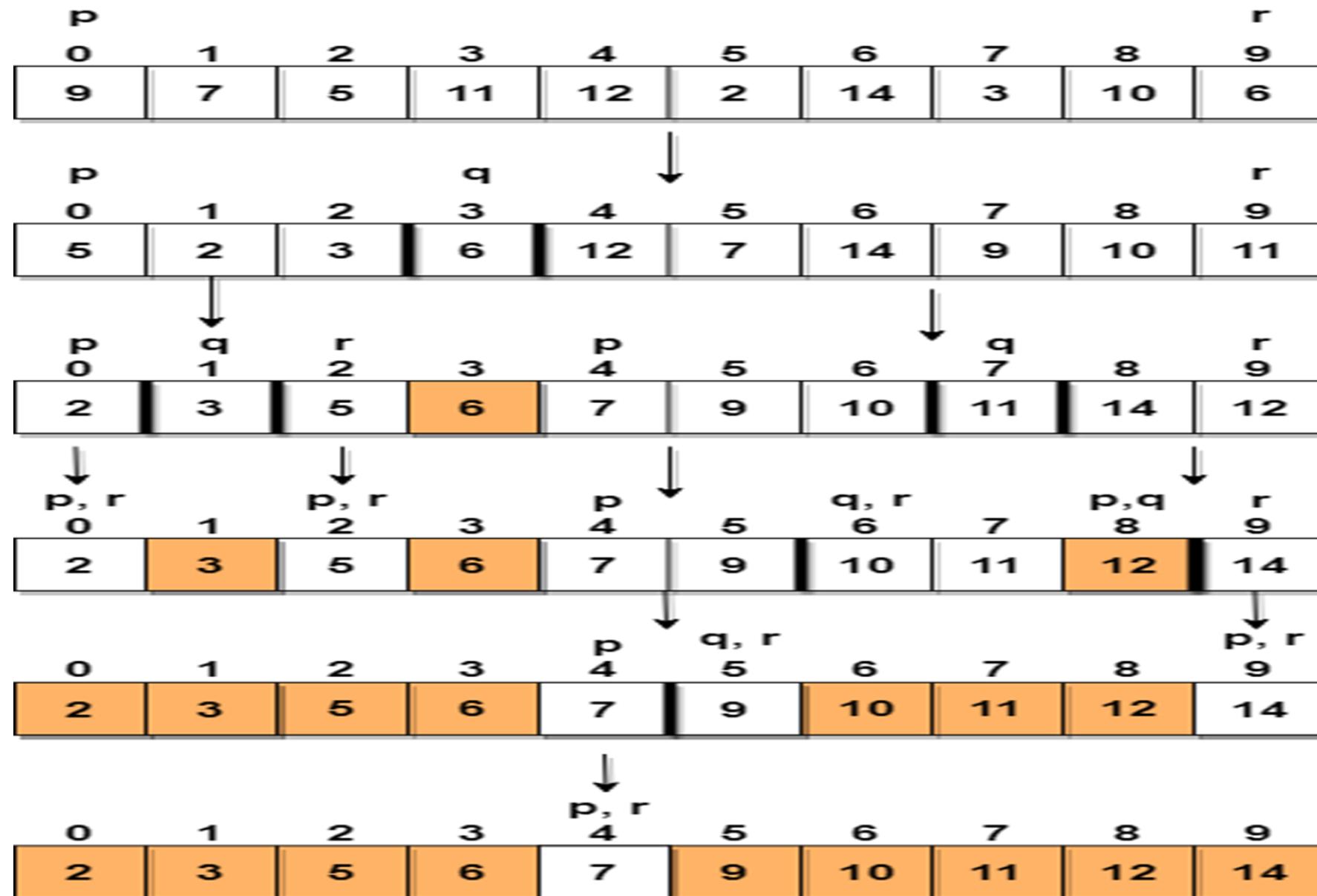
4	3	3	5	8	9	12	15	18	21	25	28
---	---	---	---	---	---	----	----	----	----	----	----

5	3	3	5	8	9	12	15	18	21	25	28
---	---	---	---	---	---	----	----	----	----	----	----

6	3	3	5	8	9	12	15	18	21	25	28
---	---	---	---	---	---	----	----	----	----	----	----

QUICK SORT EXAMPLES.

Pivot Element = 6



QUICK SORT ALGORITHM.

Algorithm Quicksort (Array, Low, High)

1. If ($\text{Low} < \text{High}$) Then
 1. Set $\text{Mid} = \text{Partition} (\text{Array}, \text{Low}, \text{High})$
 2. $\text{Quicksort} (\text{Array}, \text{Low}, \text{Mid} - 1)$
 3. $\text{Quicksort} (\text{Array}, \text{Mid} + 1, \text{High})$
2. End

QUICK SORT ALGORITHM.

Algorithm Partition (Array, Low, High)

1. Set Key = Array[low], I = Low + 1, J = High
2. Repeat Steps A through C
 - A. while ($I < High \ \&\& \ Key \geq Array[i]$) $i++$
 - B. while ($Key < Array[j]$) $j- -$
 - C. if ($I < J$) then
 swap Array[i] with Array[j]
 else
 swap Array[low] with Array[j]
 return j {Position For KEY}
3. End

QUICK SORT EXAMPLES. (PARTITIONS)

LOW	I	J, HIGH							
42	37	11	98	36	72	65	10	88	78
LOW	I				J	HIGH			
42	37	11	98	36	72	65	10	88	78
LOW	I				J	HIGH			
42	37	11	10	36	72	65	98	88	78
LOW	J I				HIGH				
42	37	11	10	36	72	65	98	88	78
LOW	J I				HIGH				
36	37	11	10	42	72	65	98	88	78
<42					>42				

0	1	2	3	4	5	6	7	8	9
42	37	11	98	36	72	65	10	88	78

$n = 10$, low = 0, high = 9
 $key = a[0] = 42$, i = 1, j = 9

while ($i < 9$ && $42 > a[i]$) ✓ i = 2
 $2 < 9$ && $42 > a[2]$ ✓ i = 3
 $3 < 9$ && $42 > a[3]$ ✓ j = 8
while $42 < a[9]$ ✓ j = 8
 $42 < a[8]$ ✓ j = 7
 $42 < a[7]$ ✓ j = 6
if $3 < 7$ ✓ swap($a[3], a[7]$)
i.e.; swap(98, 10)

0	1	2	3	4	5	6	7	8	9
42	37	11	10	36	72	65	98	88	78

while $3 < 9$ && $42 > a[3]$ ✓ i = 4
 $4 < 9$ && $42 > a[4]$ ✓ j = 5
 $5 < 9$ && $42 > a[5]$ ✓ j = 6
while $42 < a[7]$ ✓ j = 6
 $42 < a[6]$ ✓ j = 5
 $42 < a[5]$ ✓ j = 4
 $42 < a[4]$ ✓ j = 3

if ($5 < 4$) ✗
swap ($a[0], a[4]$)
i.e.;
swap (42, 36)

Resultant Array

36	37	11	10	42	72	65	98	88	78
0	1	2	3	4	5	6	7	8	9

return (4)

if ($0 < 9$)
mid = 4
Quicksort ($a, 0, 3$)
Quicksort ($a, 5, 9$)

1. Set Key = Array[low], I = Low + 1, J = High
2. Repeat Steps A through C
 - A. while ($I < High$ && Key ≥ Array[i]) i++
 - B. while ($Key < Array[j]$) j - -
 - C. if ($I < J$) then
swap Array[i] with Array[j]
else
swap Array[low] with Array[j]
return j {Position For KEY}

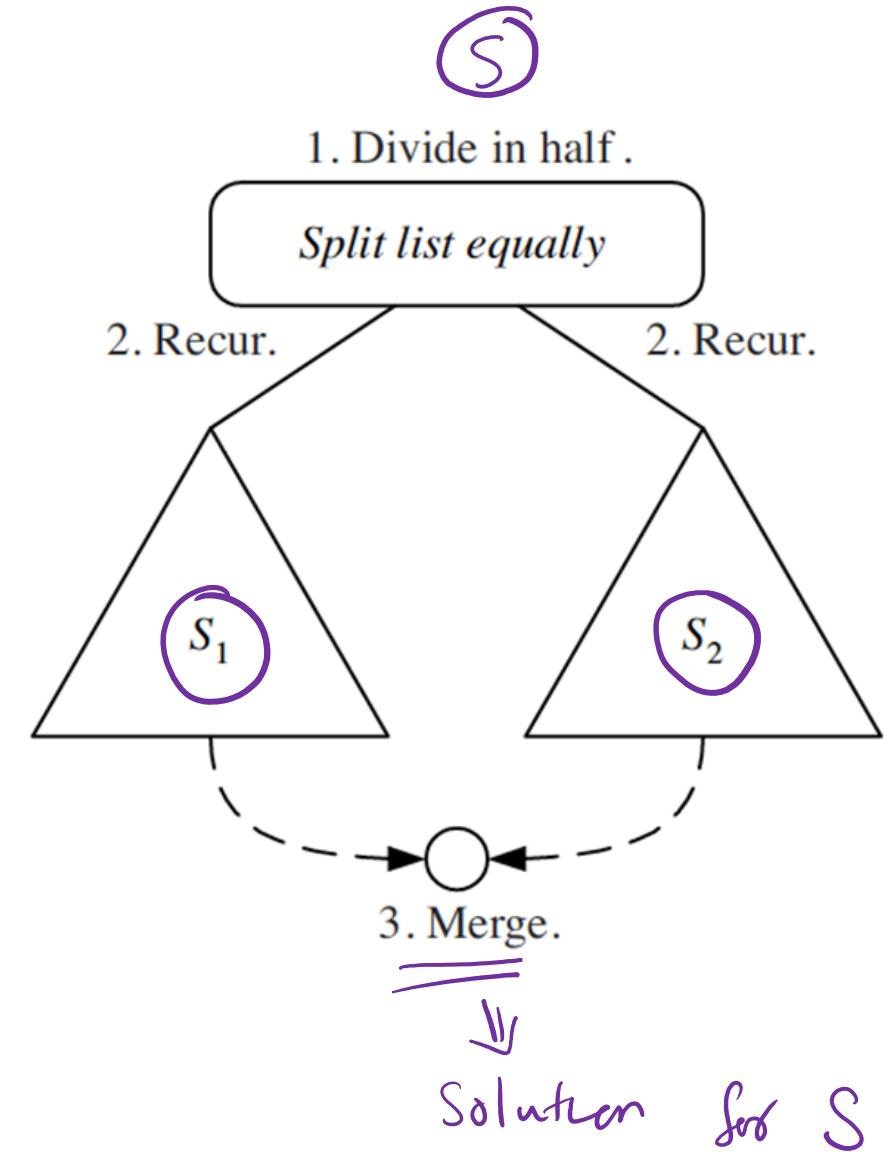
Algorithm Quicksort (Array, Low, High)
If (Low < High) Then
Set Mid = Partition (Array, Low, High)
Quicksort (Array, Low, Mid - 1)
Quicksort (Array, Mid + 1, High)
End

QUICK SORT EFFICIENCY.

- x Quick sort is considered the best general-purpose sort known today.
- x To calculate the complexity of quick sort, the number of comparisons to sort an array of n elements (index ranges from 0 to $n - 1$) is $f(n)$, given the following:
 - An array of zero or one element is already sorted. This means $f(0) = f(1) = 0$.
 - If pivot is at index i , two subarrays exist. The left subarray has (i) elements, and the right subarray has $(n - 1 - i)$ elements.
- x The number of comparisons to sort the left subarray is $f(i)$, and the number of comparisons to sort the right subarray is $f(n - 1 - i)$, where i can be between 0 to $n - 1$.
- x To continuously divide the array into subarrays, n comparisons needed.
- x The efficiency of quick sort is **$O(n \log n)$** .

DIVIDE & CONQUER.

- x Divide-and conquer is a general algorithm design paradigm.
- x **Divide:** divide the input data S in two disjoint subsets S_1 and S_2 .
- x **Recur:** solve the subproblems associated with S_1 and S_2 .
- x **Conquer:** combine the solutions for S_1 and S_2 into a solution for S .
- x Base case for recursion are subproblems of size 0 or 1.



MERGE SORT.

- x Invented by *John von Neumann* in 1945, example of Divide & Conquer.
- x Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.
- x **Basic algorithm:**
 - Divide the list into two roughly equal halves.
 - Sort the left half.
 - Sort the right half.
 - Merge the two sorted halves into one sorted list.
- x Often implemented **recursively**
- x Runtime: $O(n \log n)$.

MERGE SORT ALGORITHM.

Algorithm mergeSort(S)

Input: Sequence S with n elements

Output: Sequence S sorted

if $S.size() > 1$

$(S_1, S_2) \leftarrow \text{partition}(S, n/2)$

mergeSort(S_1)

mergeSort(S_2)

$S \leftarrow \text{merge}(S_1, S_2)$

Algorithm merge(S_1, S_2, S)

Input: Two arrays S_1 & S_2 of size n_1 and n_2 sorted in

non decreasing order and an empty array S of size at least (n_1+n_2)

Output: S containing elements from S_1 & S_2 in sorted order.

$i \leftarrow 1$

$j \leftarrow 1$

 while ($i \leq n_1$) and ($j \leq n_2$)

 if $S_1[i] \leq S_2[j]$ then

$S[i+j-1] \leftarrow S_1[i]$

$i \leftarrow i + 1$

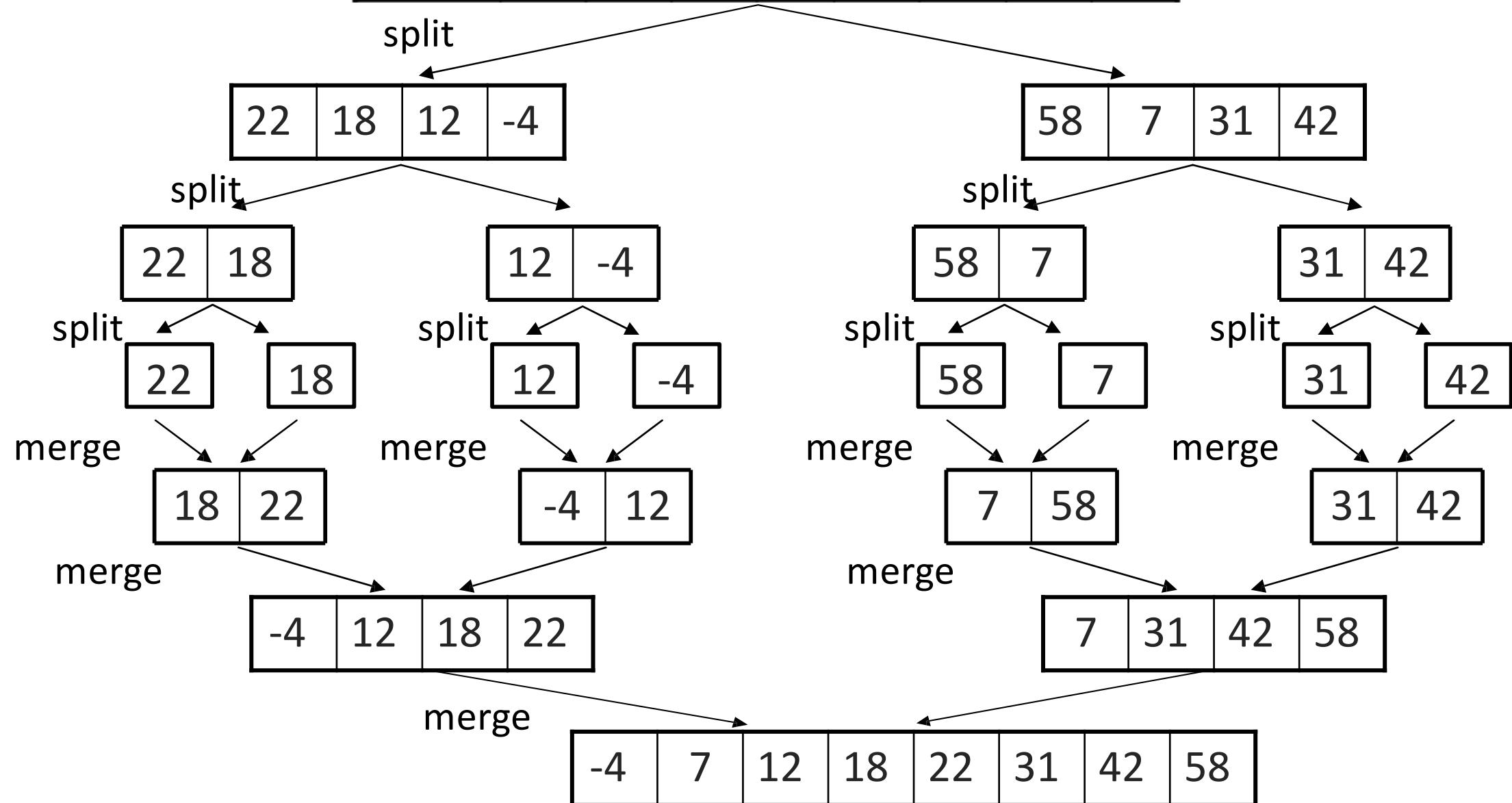
 else

$S[i+j-1] \leftarrow S_2[j]$

$j \leftarrow j + 1$

Merge Sort example.

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



S_1

Subarrays

 S_2

Next include

Final list
 S

Merged array

Sort example.

0	1	2	3
14	32	67	76
i1			

0	1	2	3
23	41	58	85
	i2		

14 from left

14	32	67	76
14	32	67	76
i1			

23	41	58	85
	i2		

23 from right

14	32	67	76
14	32	67	76
i1			

23	41	58	85
	i2		

32 from left

14	32	67	76
14	32	67	76
i1			

23	41	58	85
	i2		

41 from right

14	32	67	76
14	32	67	76
i1			

23	41	58	85
	i2		

58 from right

14	32	67	76
14	32	67	76
i1			

23	41	58	85
	i2		

67 from left

14	32	67	76
14	32	67	76
i1			

23	41	58	85
	i2		

76 from left

14	32	67	76
14	32	67	76
i1			

23	41	58	85
	i2		

85 from right

0 1 2 3 4 5 6 7

14

i

14 23

i

14 23 32

i

14 23 32 41

i

14 23 32 41 58

i

14 23 32 41 58 67

i

14 23 32 41 58 67 76

i

14 23 32 41 58 67 76 85

i

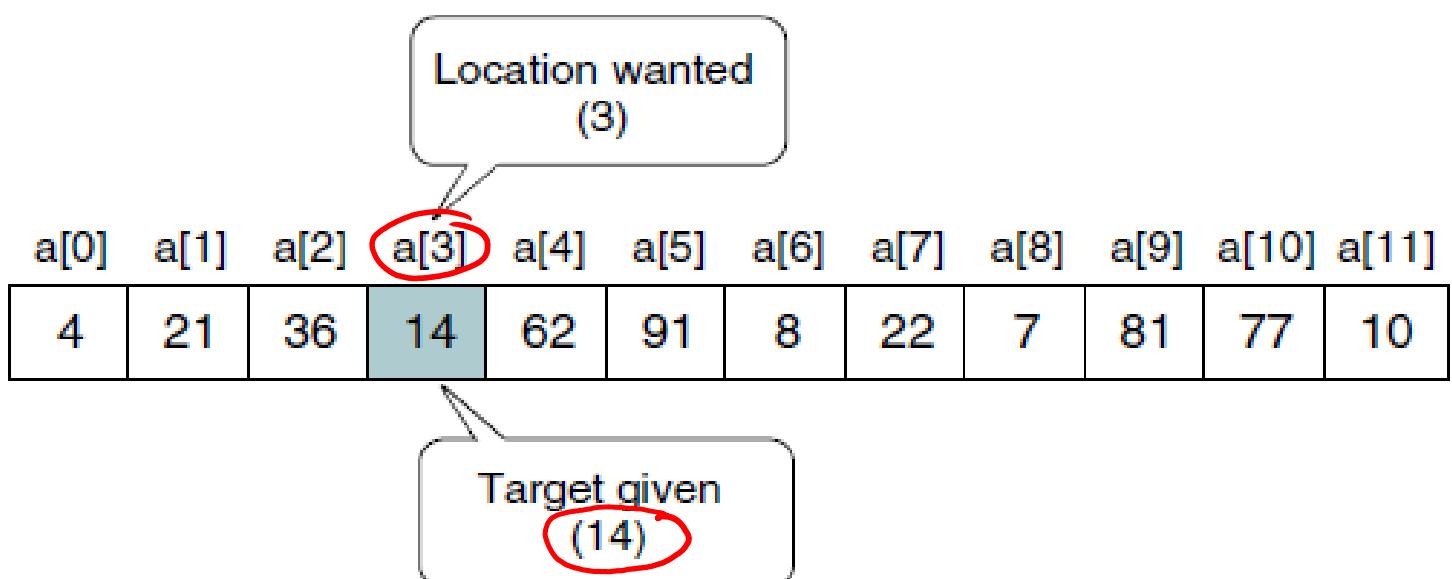
S

SEARCHING.

- x One MORE common and time-consuming operations in computer science is searching
- x The process used to find the ~~location~~ of a target among a list of objects.
- x The two basic search algorithms:
- x **Sequential search** including three interesting variations and,
- x **Binary search.**

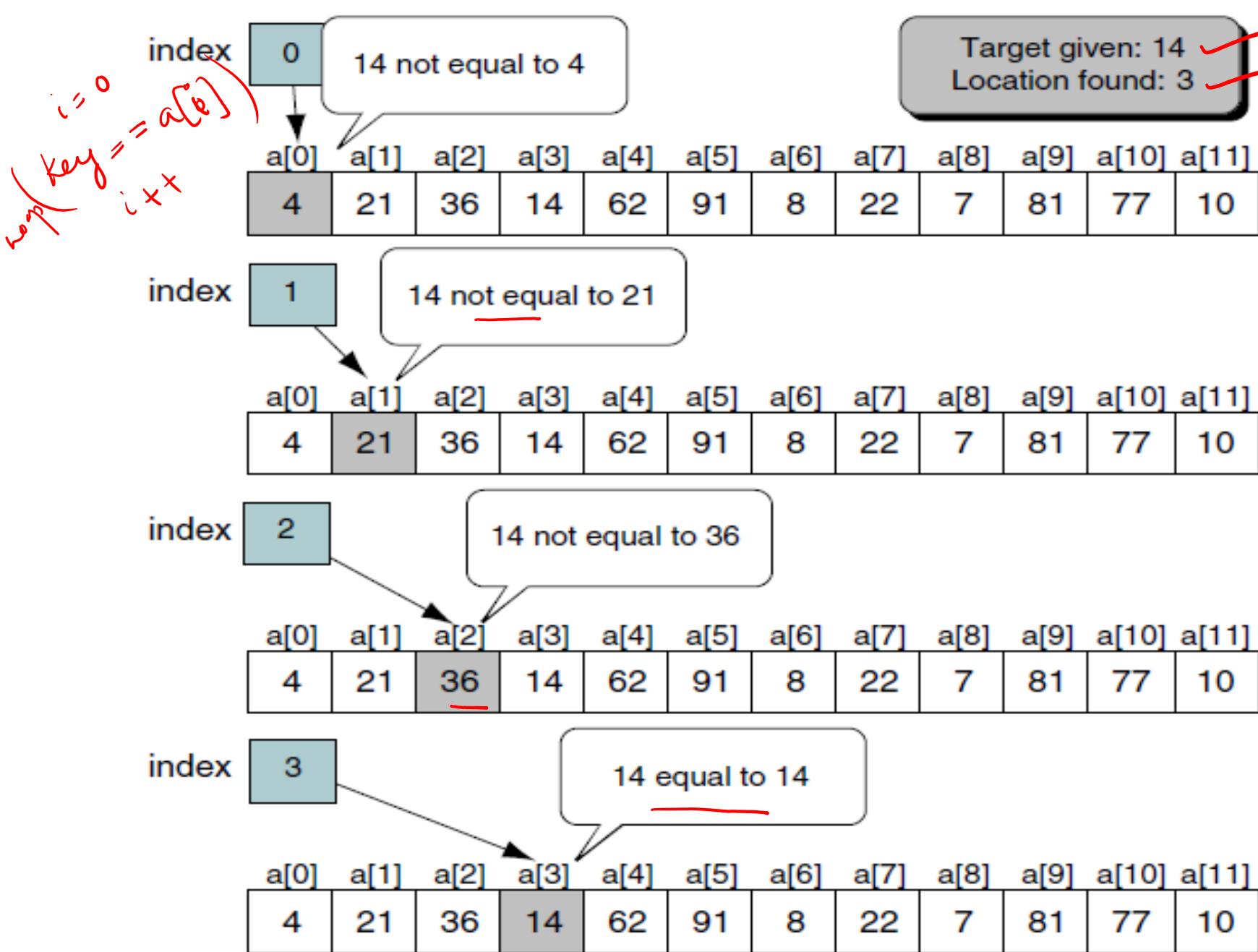
SEQUENTIAL SEARCH.

- x Used whenever the list is not ordered.
- x Generally, technique used only for small lists or lists that are not searched often.
- x Process: Start searching for the target at the beginning of the list and continue until target found or it is not in the list.
- x This approach has two possibilities: Find element (**successful**) or reach end of list (**unsuccessful**).

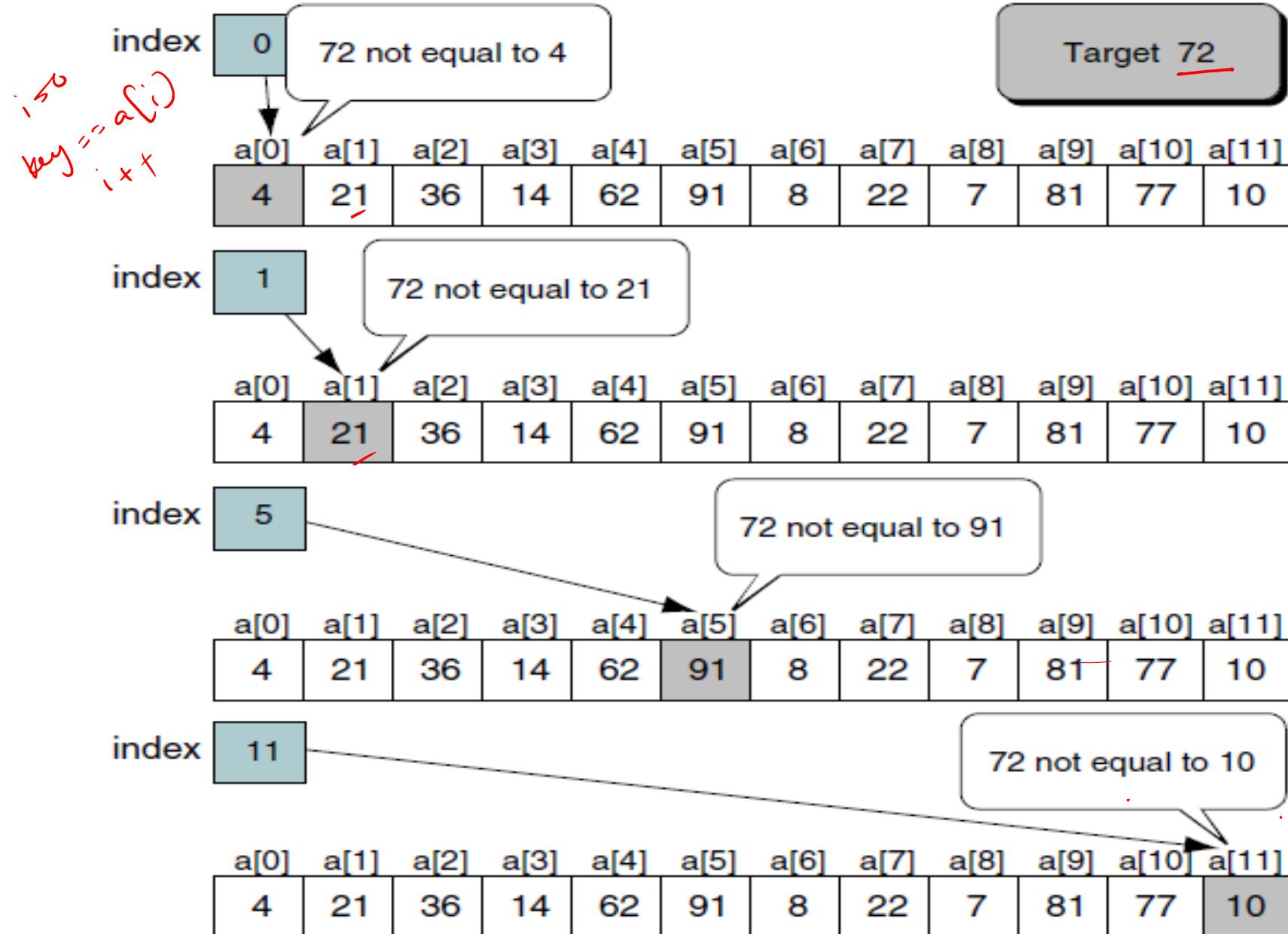


Sequential Search

Example for Successful search.



Sequential Search Example for Unsuccessful search.



Note: Not all test points are shown.

Algorithm seqSearch (list, last, target, locn)

Locate the target in an unordered list of elements.

Pre list must contain at least one element
 last is index to last element in the list
 target contains the data to be located
 locn is address of index in calling algorithm
Post if found: index stored in locn & found true
 if not found: last stored in locn & found false

Return found true or false

```
1 set looker to 0      i < 0
2 loop (looker < last AND target not equal list[looker])
    1 increment looker
3 end loop
4 set locn to looker
5 if (target equal list[looker])
    1 set found to true
6 else
    1 set found to false
7 end if
8 return found
end seqSearch
```

i < n & key != a[i]

Efficiency of the
sequential search is
O(n).

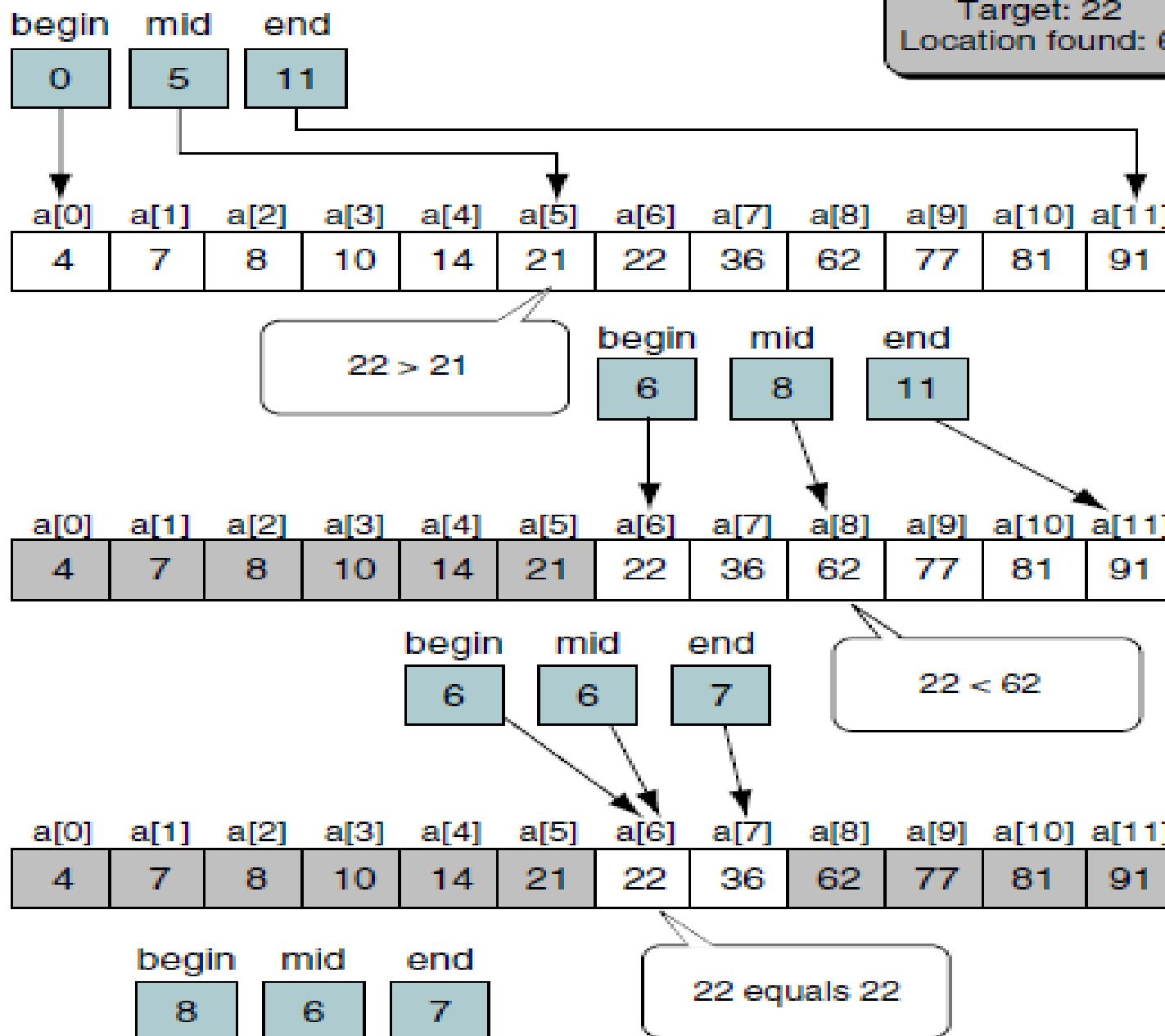
BINARY SEARCH.

- x Sequential search algorithm is very slow. If an array of 1000 elements, exists, 1000 comparisons are made in worst case.
- x If the array is not sorted, the sequential search is the only solution.
- x However, if the array is sorted, we can use a more efficient algorithm called *binary search*.
- x Generally speaking, Binary search used whenever the list starts to become large.

BINARY SEARCH.

- x Begins by testing the data in the element at the middle of the array to determine if the ***target is in the first or the second half of the list.***
- x If **target in first half**, there is NO need to check the second half.
- x If **target in second half**, NO need to test the first half.
- x In other words, half the list is eliminated from further consideration with just one comparison.
- x This process repeated, eliminating half of the remaining list with each test, until target if found or does not exist in the list.
- x To find the middle of the list, three variables needed: one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list.

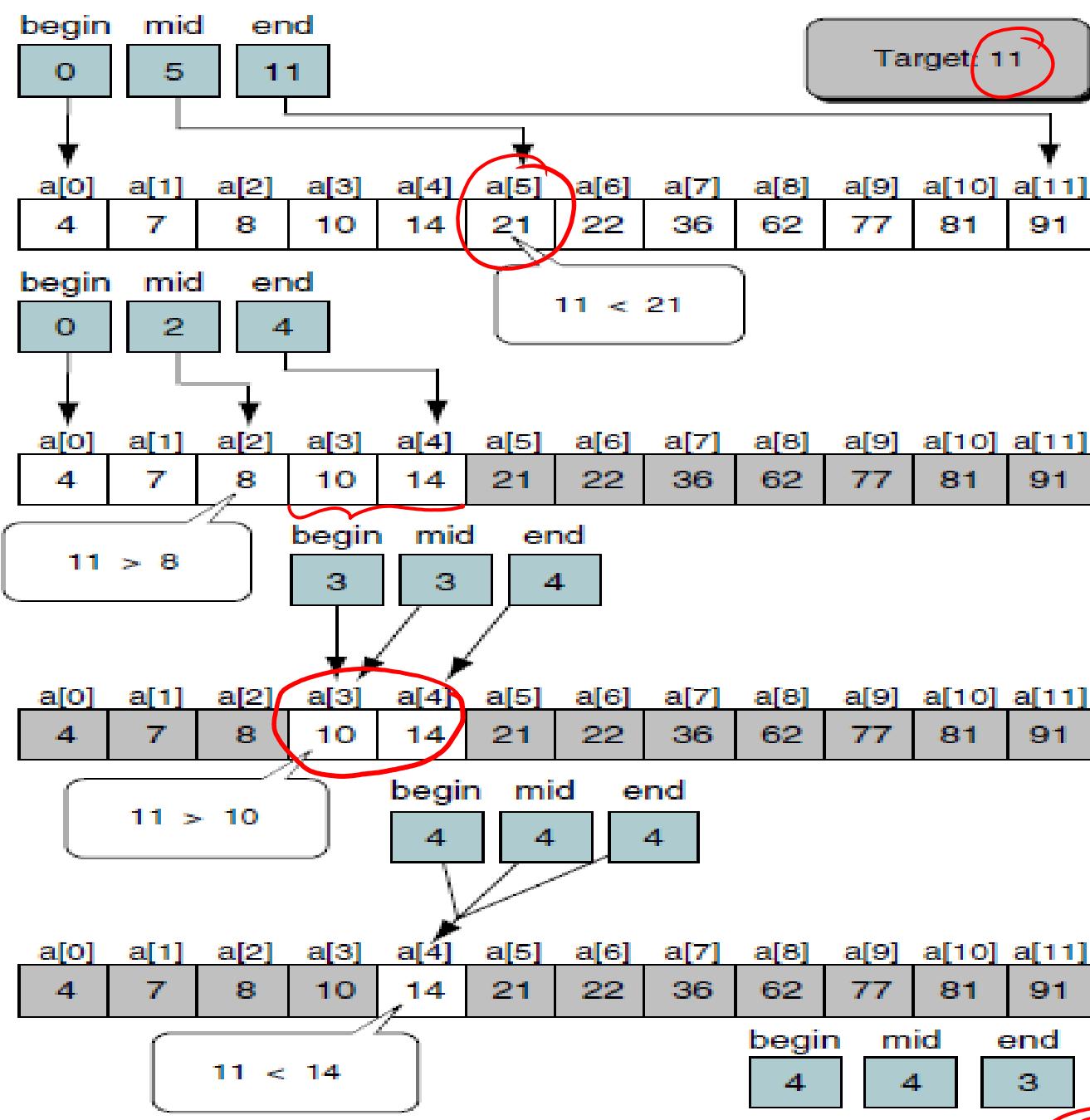
Binary Search Example for Successful search.



Successful Binary Search Example

$$\text{mid} = \lfloor (\text{begin} + \text{end}) / 2 \rfloor = \lfloor (0 + 11) / 2 \rfloor = 5$$

Binary Search Example for Unsuccessful search.



Unsuccessful Binary Search Example

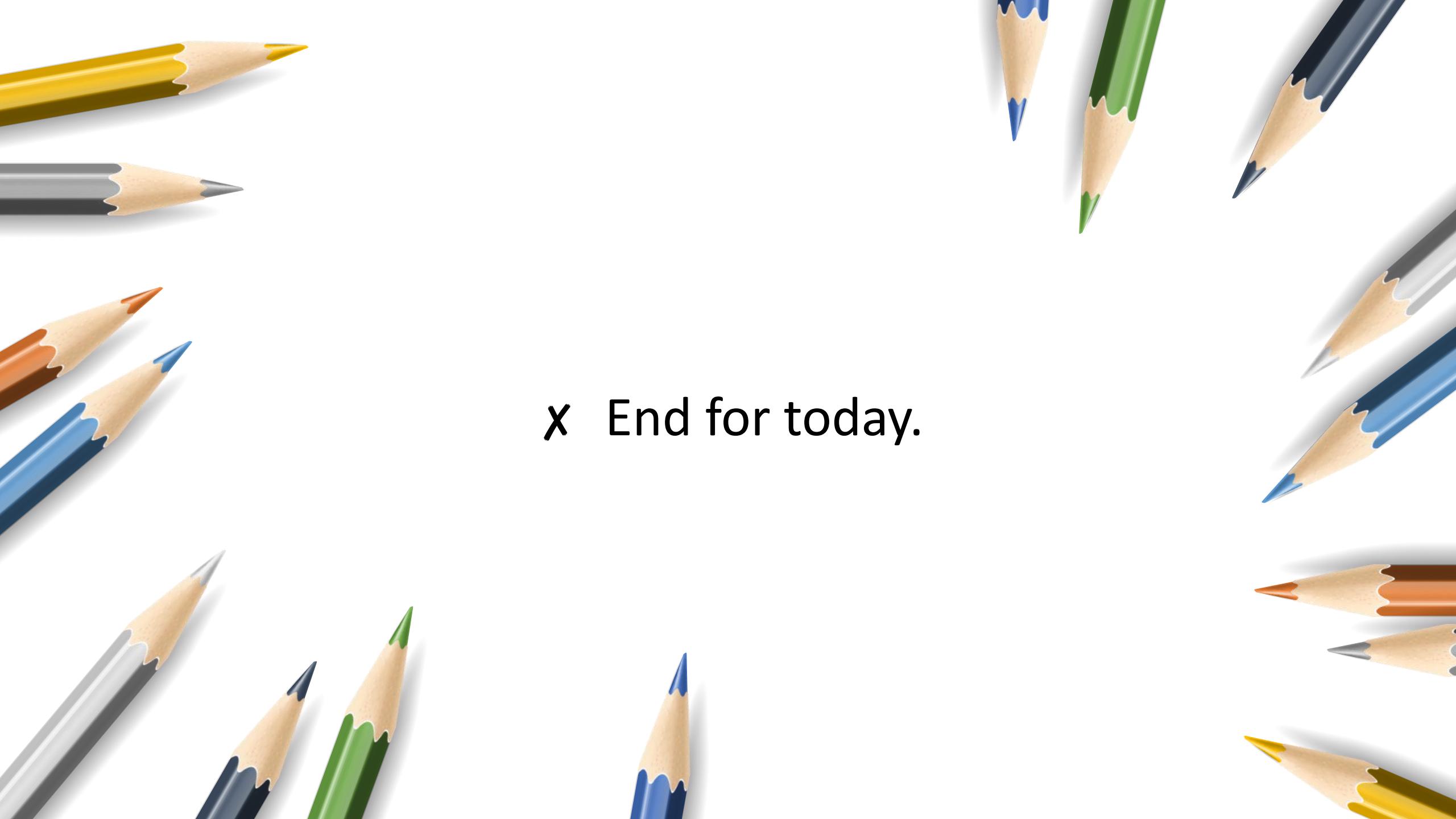
$$\text{mid} = \lfloor (\text{begin} + \text{end}) / 2 \rfloor = \lfloor (0 + 11) / 2 \rfloor = 5$$

```

Algorithm binarySearch (list, last, target, locn)
Search an ordered list using Binary search
  Pre      list is ordered; it must have at least 1 value
            last is index to the largest element in the list
            target is the value of element being sought
            locn is address of index in calling algorithm
  Post     FOUND: locn assigned index to target element
            found set true
            NOT FOUND: locn = element below or above target
            found set false
  Return   found true or false
1  set begin to 0
2  set end to last
3  loop (begin <= end)
    1  set mid to (begin + end) / 2
    2  if (target > list[mid])
        Look in upper half
        1  set begin to (mid + 1)
    3  else if (target < list[mid])
        Look in lower half
        1  set end to mid - 1
    4  else
        Found: force exit
        1  set begin to (end + 1)
    5  end if
4  end loop
5  set locn to mid
6  if (target equal list [mid])
    1  set found to true
7  else
    1  set found to false
8  end if
9  return found
end binarySearch

```

Efficiency of the
binary search is
O(log n).



x End for today.

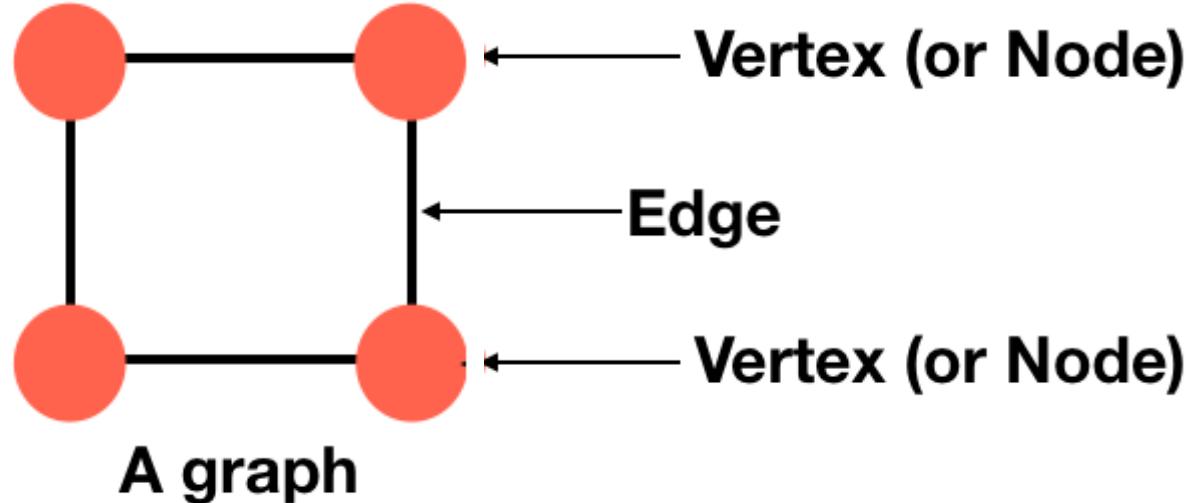
The background of the image features a complex network graph composed of numerous small gray dots connected by thin gray lines. Several larger, semi-transparent circles of varying sizes are overlaid on the graph, some containing solid dark blue or cyan dots. One large cyan circle is in the bottom left, one large dark blue circle is at the top center, and several smaller cyan and gray circles are scattered throughout.

Useful data structures.

What we know already.

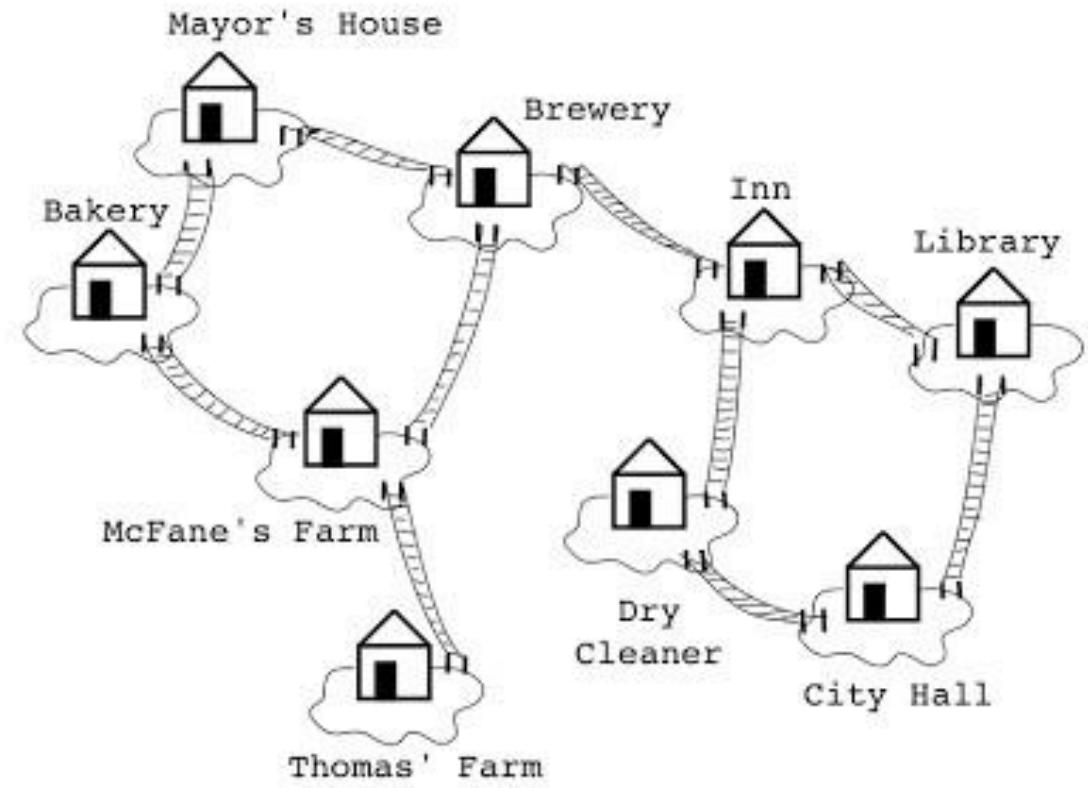
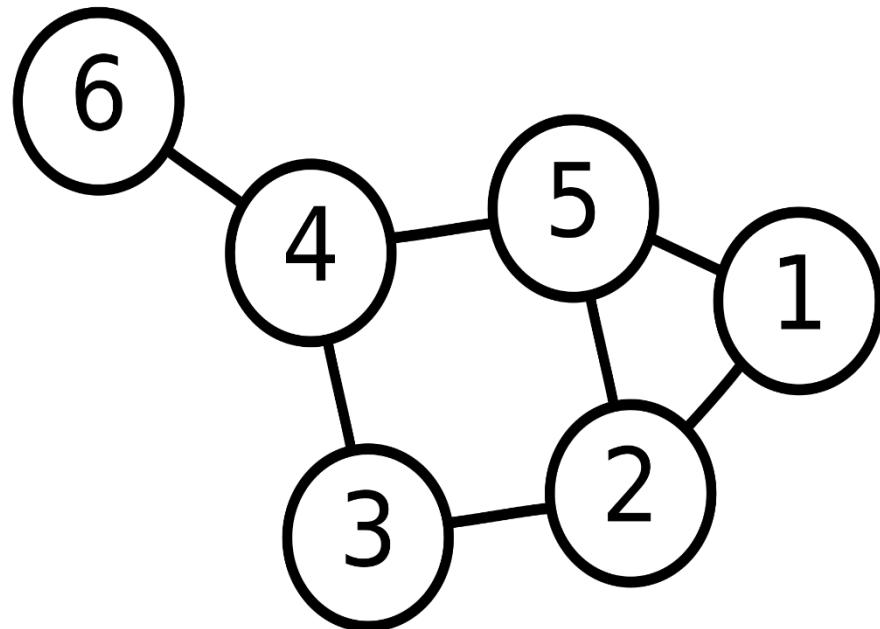
- Need for ADTs and data structures:
 - Regular Arrays (with dynamic sizing)
 - Linked Lists
 - Stacks, Queues
 - Heaps
 - Unbalanced and Balanced Search Trees, B-tree, B+-tree
- Some algorithms like Tree traversals

BASIC CONCEPTS: GRAPHS.



A graph is a **data structure** that consists of a **set of nodes or vertices** and a **set of edges** that *relate the nodes to each other*.

BASIC CONCEPTS: GRAPHS.



The set of edges describes relationships among the nodes.

MATHEMATICAL or formal definition.

- 💡 A graph G is defined as follows:

$$G = (V, E)$$

$V(G)$: a finite, nonempty set of vertices

$$V = \{v_1, v_2, \dots, v_n\}$$

$E(G)$: a set of edges (pairs of vertices)

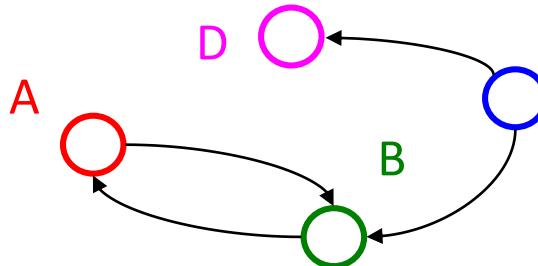
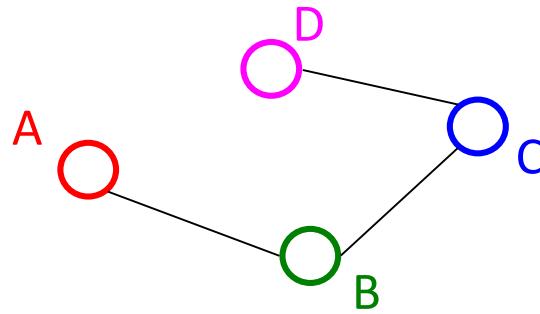
$$E = \{e_1, e_2, \dots, e_m\}$$

An edge "connects" the vertices

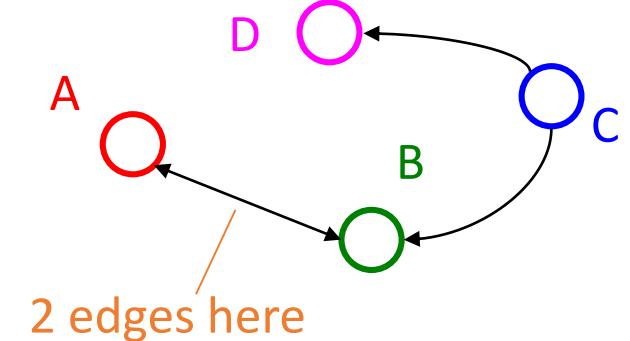
Graphs can be directed or undirected

MATHEMATICAL or formal definition.

- ↳ **Undirected graphs:** edges have no specific direction.
- ↳ Edges are always "two-way"
- ↳ Thus, $(u, v) \in E$ implies $(v, u) \in E$.
- ↳ Only one of these edges needs to be in the set, the other is implicit.



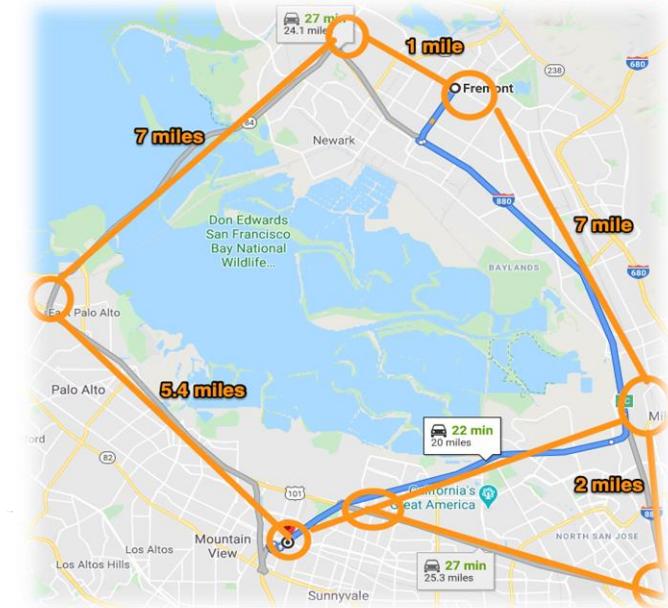
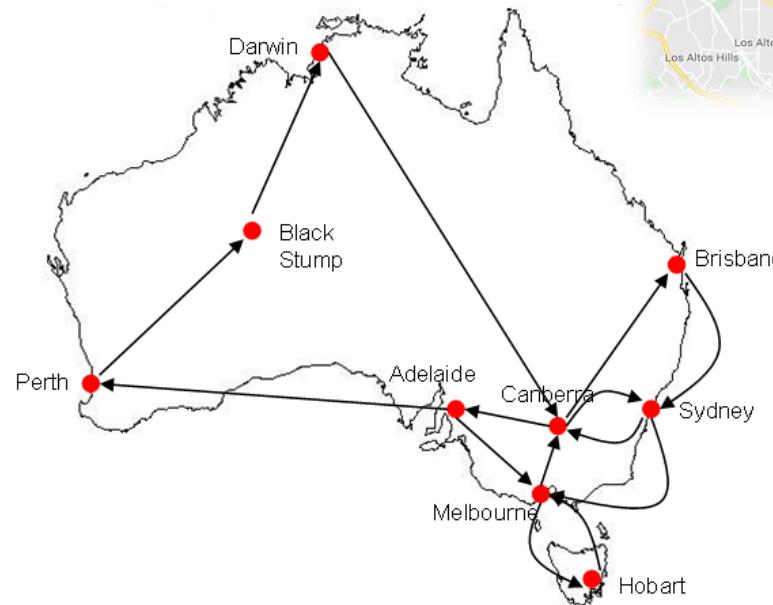
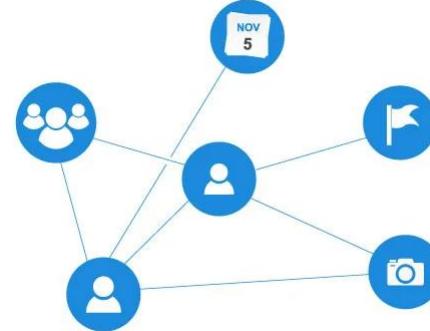
or



- ↳ **Directed graphs or digraphs:** edges have direction
 - ↳ Thus, $(u, v) \in E$ does not imply $(v, u) \in E$.
 - ↳ Let $(u, v) \in E$ mean $u \rightarrow v$ Where u is the source, and v the destination

Applications of graphs

- Web pages with links
- Facebook friends
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites



Applications of graphs.

Graphs can be used to ***solve complex routing problems***, such as designing and routing airlines among the airports they serve.

Similarly, graphs can be used to route messages over a computer network from one node to another.

Trees & Graphs

- In a non-linear list, each element can have more than one successor.
- In a tree, an element can have only one predecessor.
- In a graph, an element can have one or more predecessors.

One final point: ***A tree is a graph*** in which each vertex has only one predecessor; however, ***a graph is not a tree***.

Basic concepts in Computer science.

A **graph** is a *collection of nodes called vertices*, and a *collection of segments called lines, connecting pairs of vertices*.

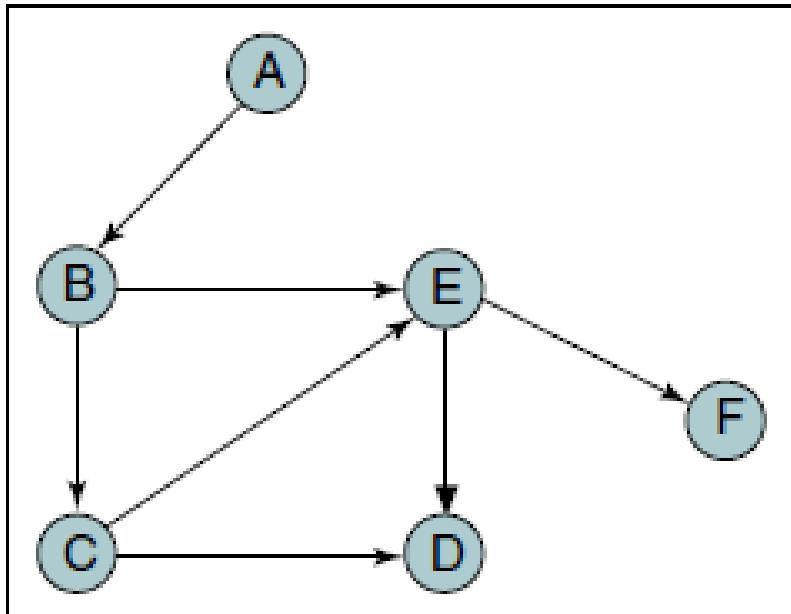


FIGURE 11-1 (a) **Directed graph**

So, a graph consists of two sets, a set of vertices and a set of lines (we know that)

A directed graph, or digraph for short, is a graph in which each line has a direction (arrow head) to its successor.

The **lines** in a directed graph are known as **arcs**. In a directed graph, the flow along the arcs between two vertices can follow only the indicated direction.

Basic concepts.

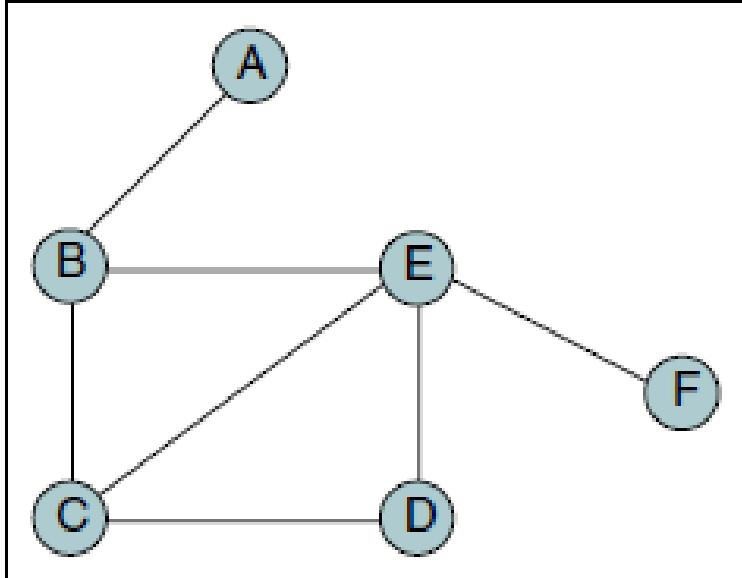


FIGURE 11-1 (b) Undirected graph

{A, B, C, E} is one path and {A, B, E, F} is another.

An undirected graph is a graph in which there is no direction (arrow head) on any of the lines, which are known as edges.

In an undirected graph, the flow between two vertices can go in either direction.

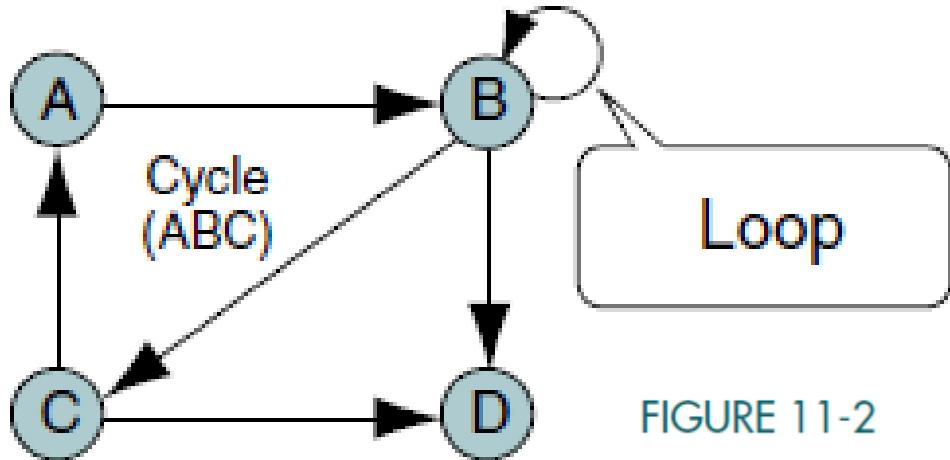
A path is a sequence of vertices in which each vertex is adjacent to the next one.

Two vertices in a graph are said to be adjacent vertices (or neighbors) if there is a path of length 1 connecting them.

In (a), B is adjacent to A, whereas E is not adjacent to D; on the other hand, D is adjacent to E

In (b), E and D are adjacent, but D and F are not.

Cycles and loops.



In Figure 11-1(a), ***same vertices do not constitute a cycle*** because in a digraph a path can follow only the direction of the arc, whereas in an undirected graph a path can move in either direction along the edge.

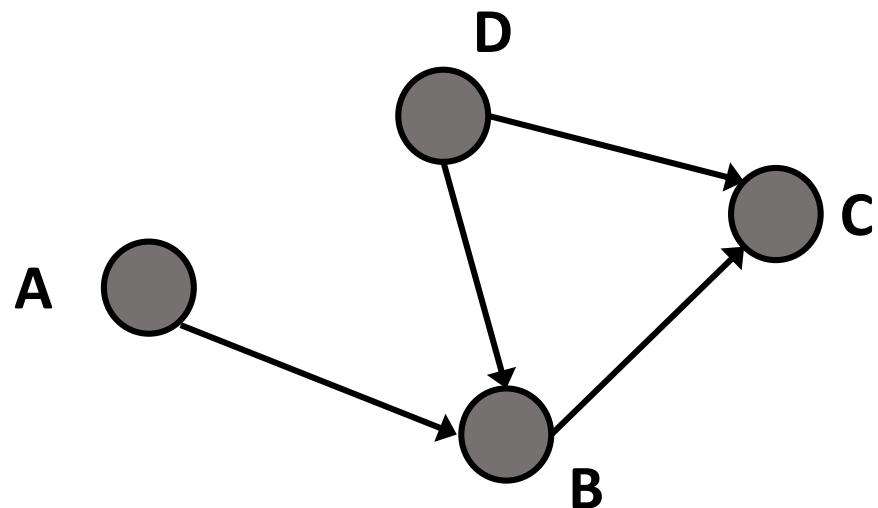
A ***cycle*** is a path consisting of at least three vertices that starts and ends with the same vertex.

In Fig.11-1(b) : B, C, D, E, B is a cycle

A ***loop*** is a special case of a cycle in which a single arc begins and ends with the same vertex. In a loop the end points of the line are the same.

Length: Length of a path is the **number of edges** in the path

Paths and Cycles in Directed Graphs

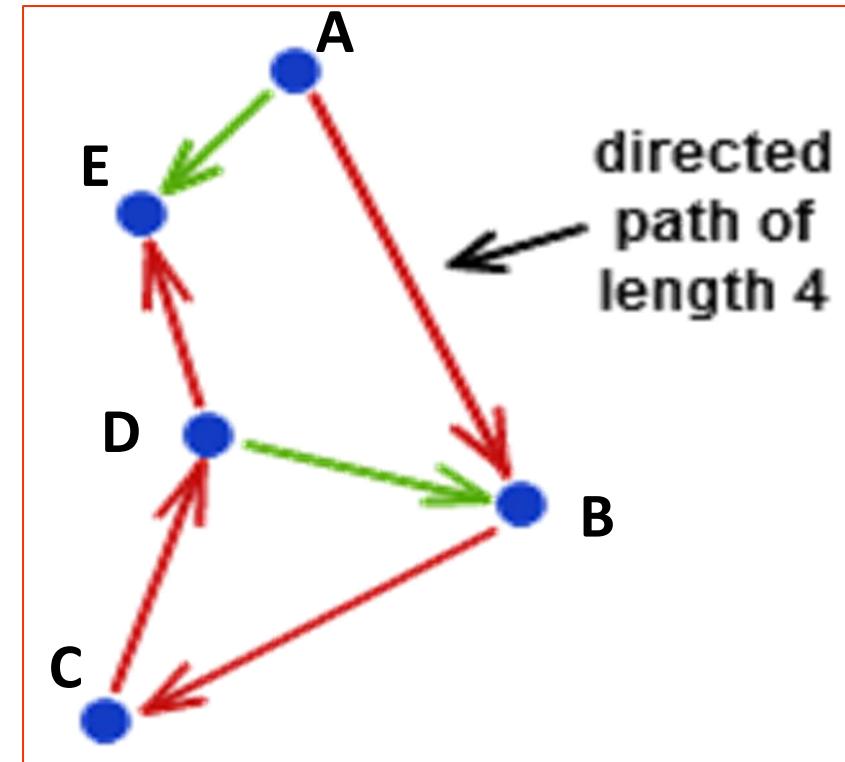


Is there a path from A to D?

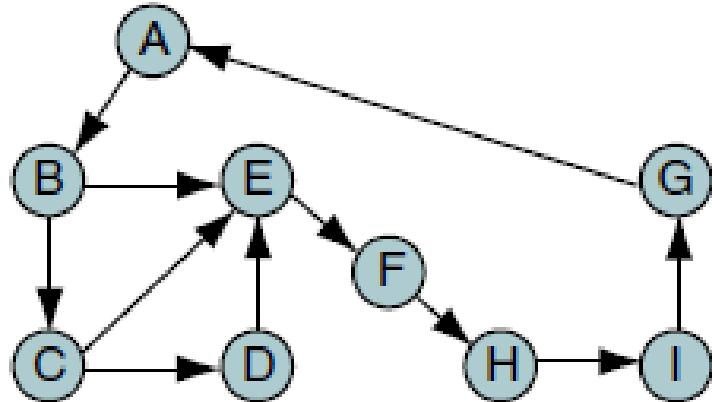
No

Does the graph contain any cycles?

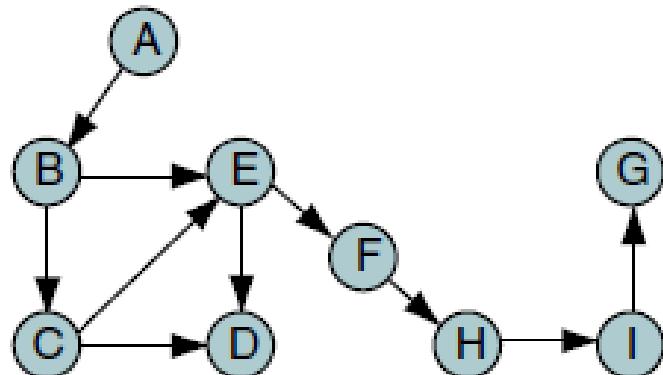
No



Connected graphs.



(b) Strongly connected



(a) Weakly connected

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

A graph is said to be **connected** if, ignoring direction, there is a path from any vertex to any other vertex.

A directed graph is **strongly connected** if there is a path from each vertex to every other vertex in the digraph.

A directed graph is **weakly connected** if at least two vertices are not connected.

A connected undirected graph would always be strongly connected, so the concept is not normally used with undirected graphs.

disjoint graphs.

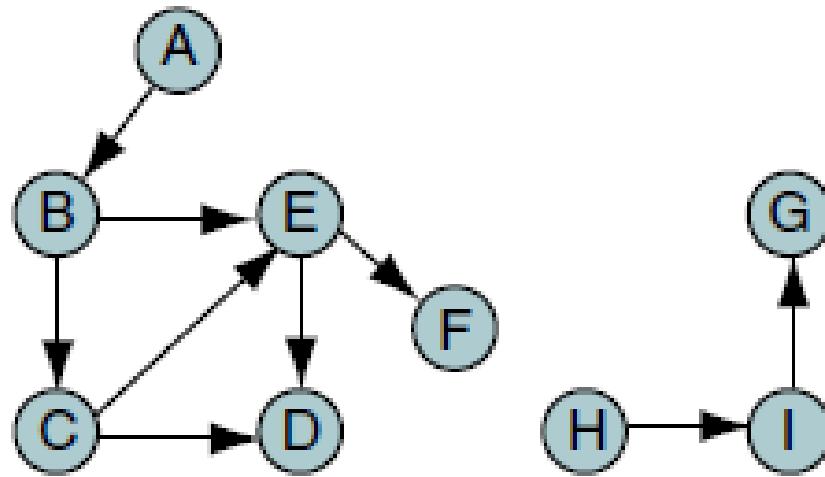


FIGURE 11-3

(c) Disjoint graph

A graph is a disjoint graph if it is not connected.

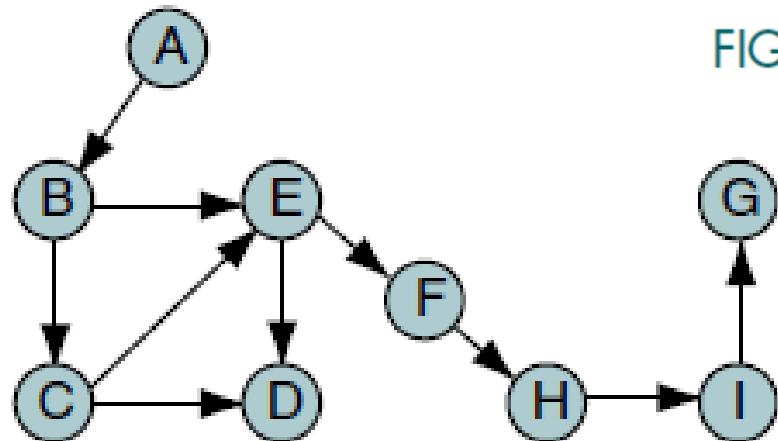
Degree of a vertex.

Number of lines incident to it.

The **indegree** is the number of arcs entering the vertex.

The **outdegree** of a vertex in a digraph is the number of arcs leaving the vertex.

FIGURE 11-3



(a) **Weakly connected**

The degree of vertex B is 3
The degree of vertex E is 4.

The indegree of vertex B is 1 and its outdegree is 2

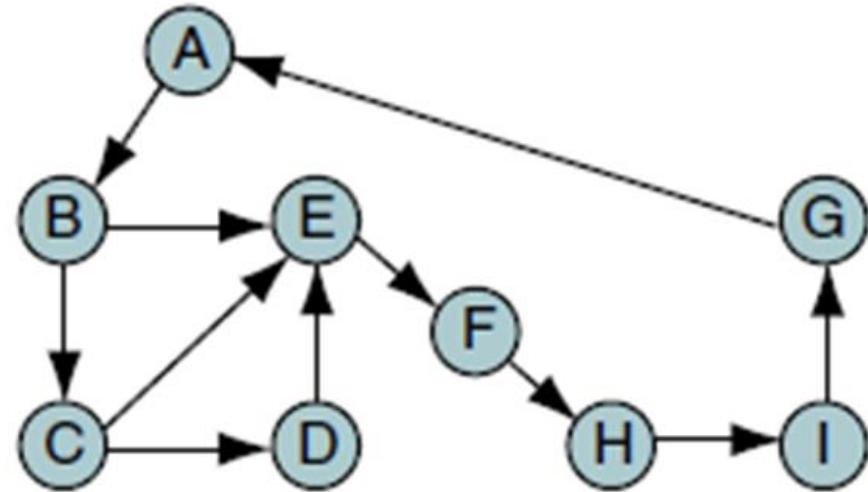
Degree of a vertex.

Number of lines incident to it.

The indegree is the number of arcs entering the vertex.

The outdegree of a vertex in a digraph is the number of arcs leaving the vertex.

FIGURE 11-3



(b) Strongly connected

The indegree of vertex E is 3 and its outdegree is 1.

Operations.

There are six primitive graph operations that provide the basic modules needed to maintain a graph:

Insert a vertex

Delete a vertex

Add an edge

Delete an edge

Find a vertex

Traverse a graph

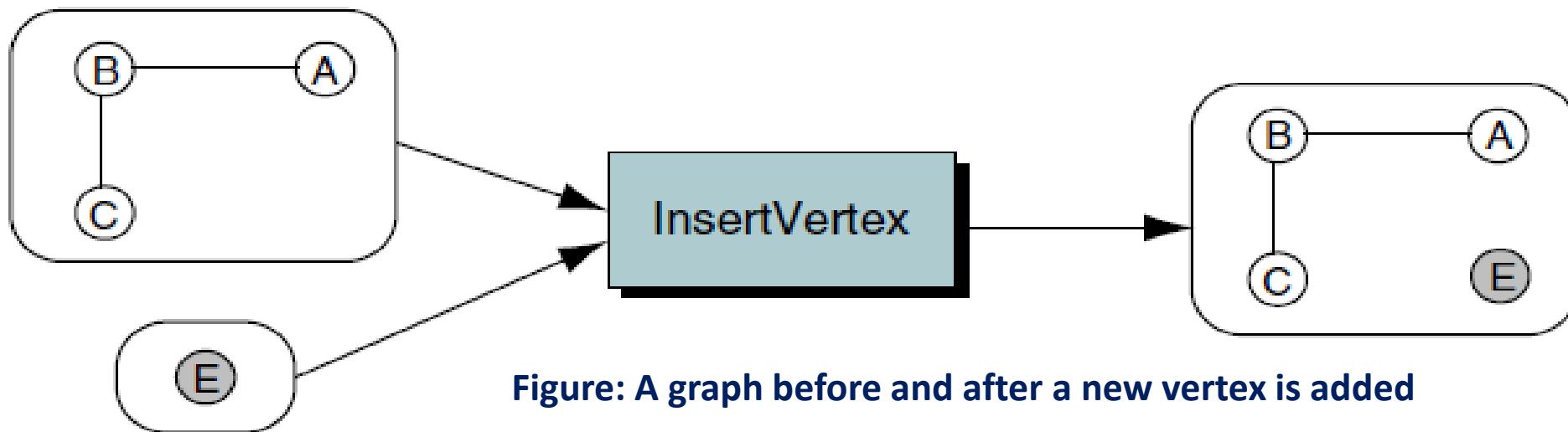
Insert vertex.

Adds a new vertex to a graph.

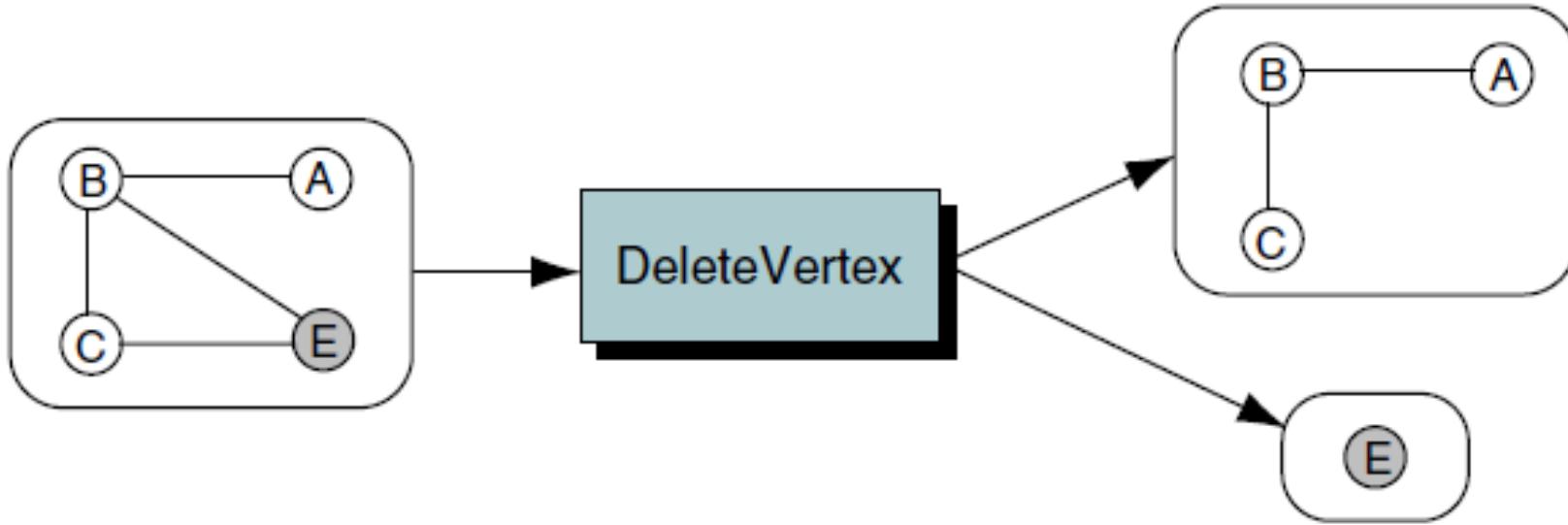
When a vertex is inserted, it is disjoint.
That is, it is not connected to any other vertices in the list.

Inserting a vertex is just the first step in the insertion process.

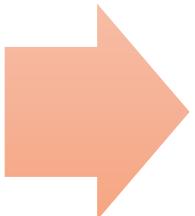
After a vertex is inserted, it must be connected.



Delete Vertex.



Delete vertex removes a vertex from the graph.



When a vertex is deleted, all connecting edges are also removed.

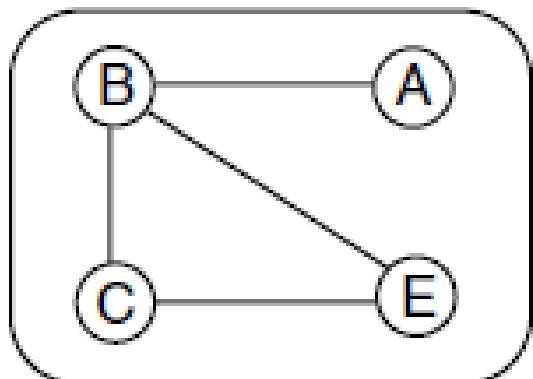
Add Edge.

Add edge connects a vertex to a destination vertex.

If a vertex requires multiple edges, add an edge must be called once for each adjacent vertex.

To add an edge, two vertices must be specified.

If the graph is a digraph, one of the vertices must be specified as the source and one as the destination.



AddEdge

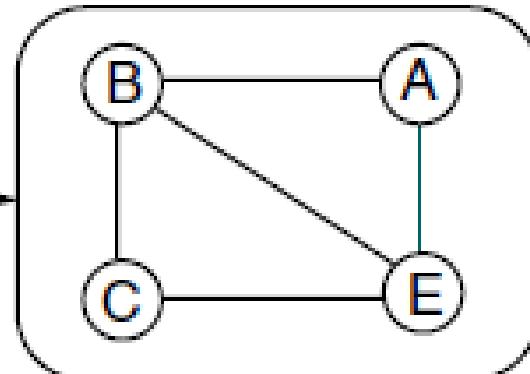


Figure: Adding an edge, {A, E}, to the graph.

Delete Edge.

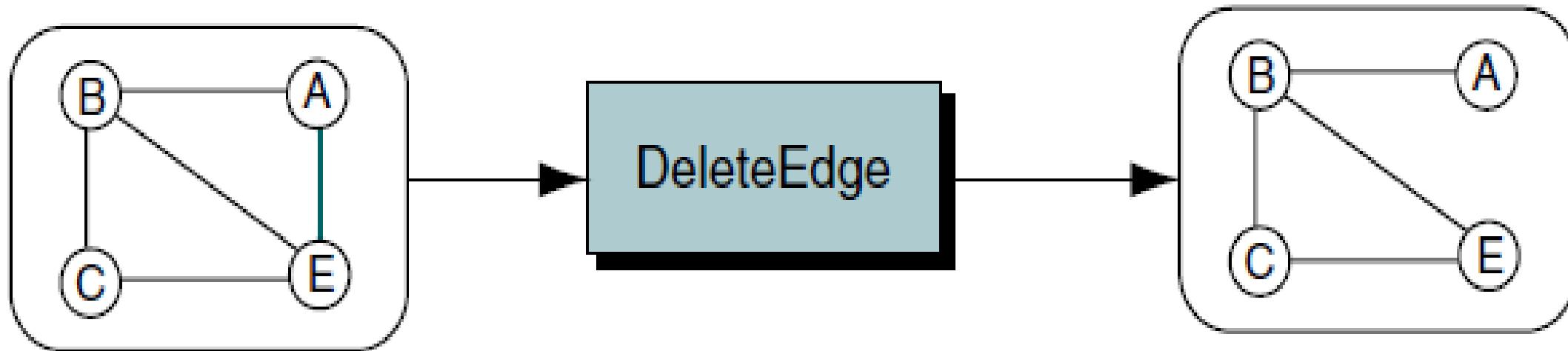
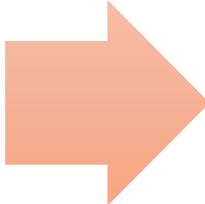


Figure: Deleting the edge, {A, E}, to the graph.

Delete edge removes one edge from a graph.

Find vertex.

Find vertex traverses a graph, looking for a specified vertex.



If the vertex is found, its data are returned. If it is not found, an error is indicated.

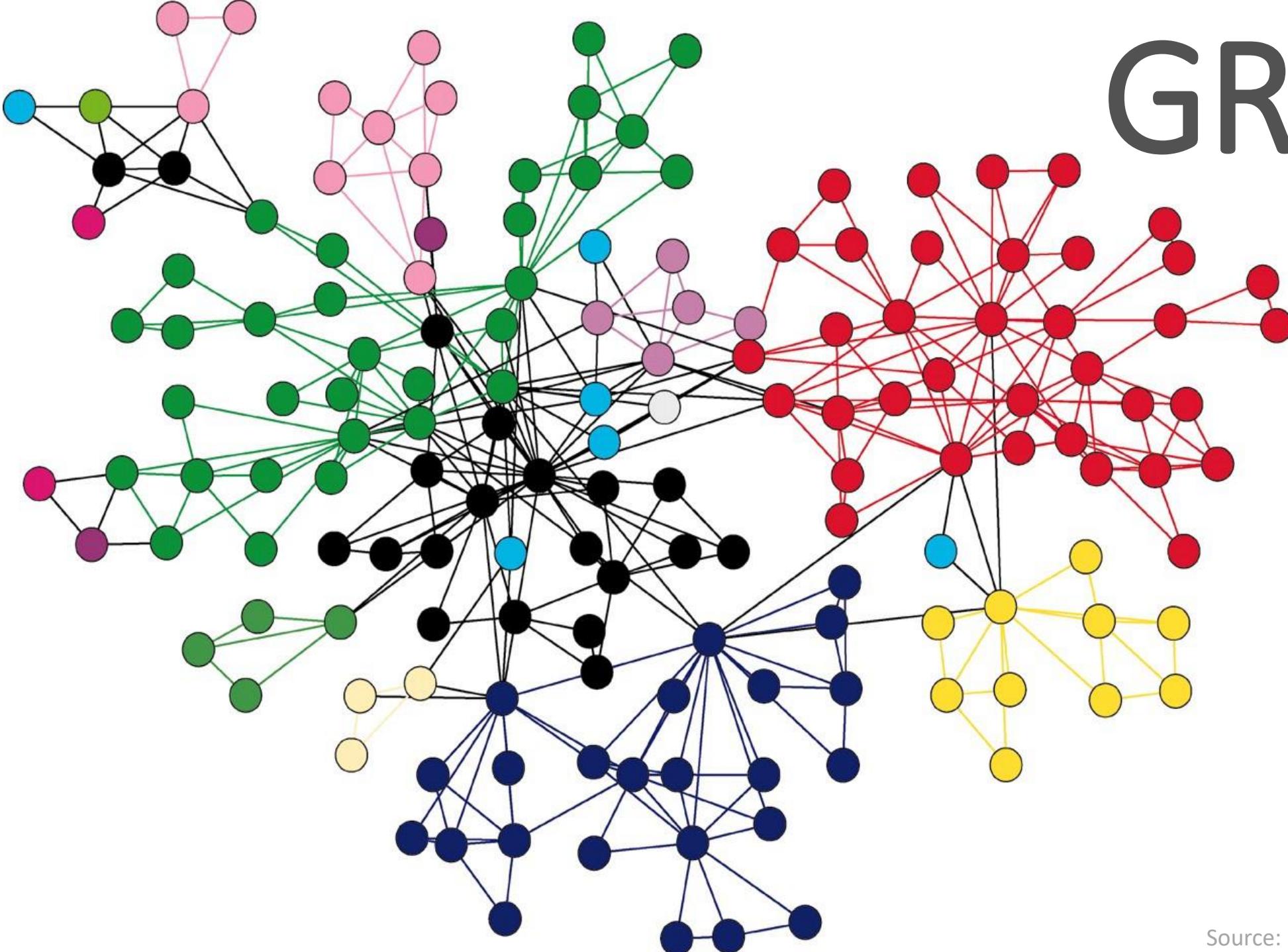


Figure: Find vertex traverses the graph, looking for vertex C..

End.

GRAPHS.

MORE
OPERATIONS.

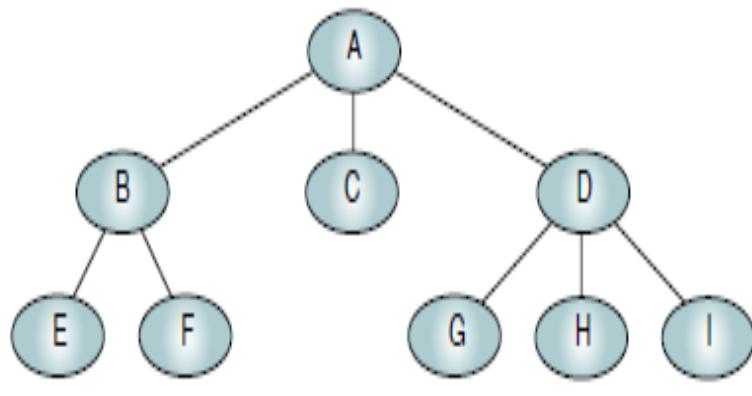


We will look into:

- Graph Traversal
- Graph Storage Structures
 - Adjacency Matrix.
 - Adjacency List.
- Graph Algorithms.
- Networks
 - City Network.
 - Minimum Spanning Tree
 - Shortest Path Algorithm

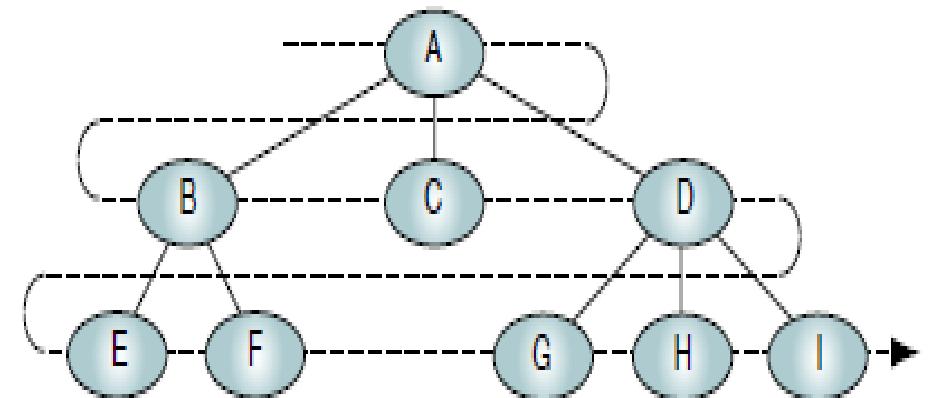
Graph Traversal

- Two Commonly used Traversal Techniques are
 - **Depth First Search (DFS)**
 - Process all of a vertex's descendants before moving to an adjacent vertex.



Depth-first traversal: A B E F C D G H I

vertex I



Breadth-first traversal: A B C D E F G H I

Depth-first Traversal of a Tree

Breadth-first Traversal of a Tree

Depth First Search (DFS)

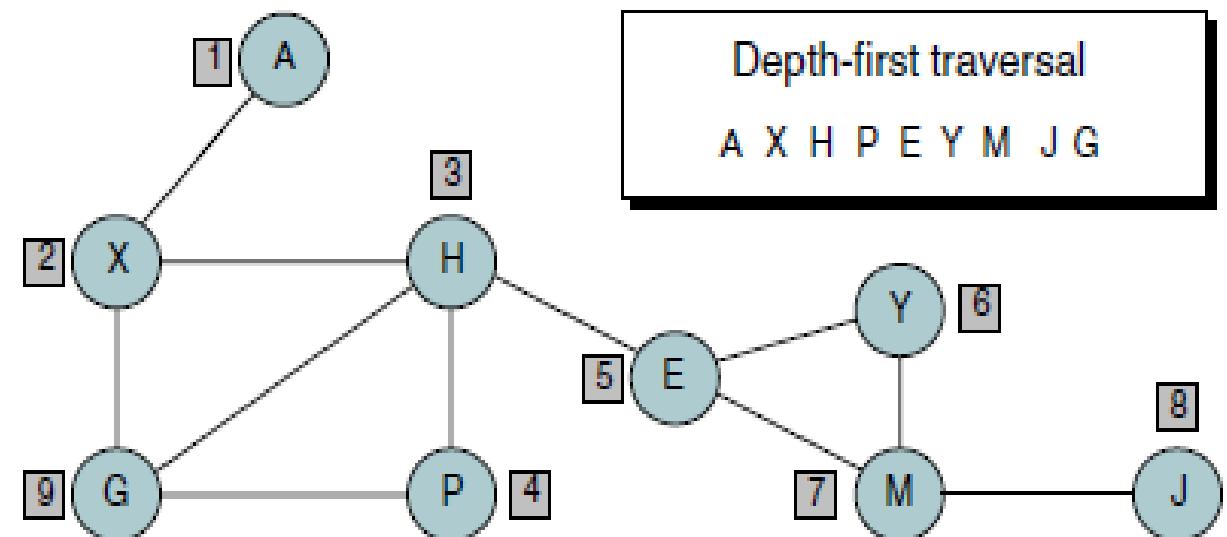
Steps:

1. Begin by pushing the first vertex **A** into the stack.

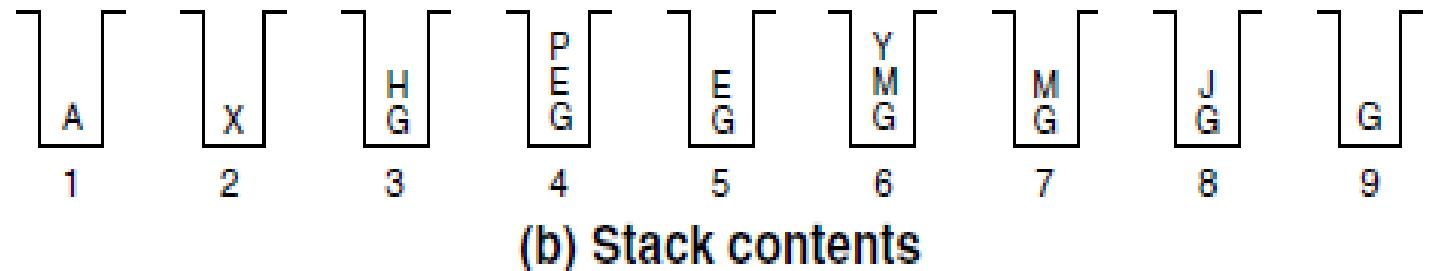
2. Then loop. Pop stack. After processing the vertex, **push all of the adjacent vertices into the stack**. To process **X** at step 2, therefore, pop **X** from the stack, process it, and then push **G & H** into the stack, giving the stack contents for step 3.

3. When **stack is empty**, traversal is complete.

Depth-first Traversal of a Graph

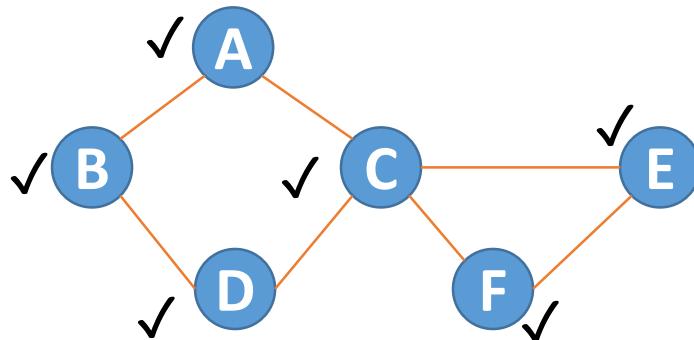
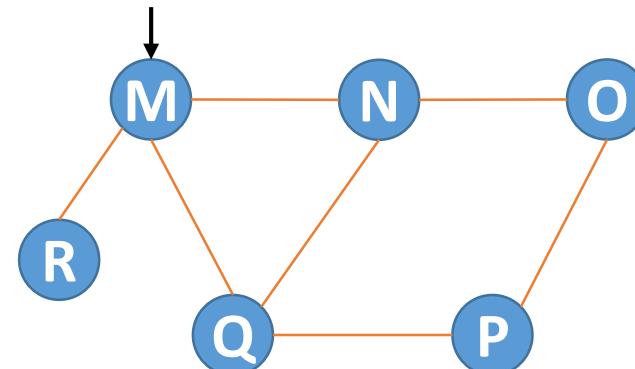
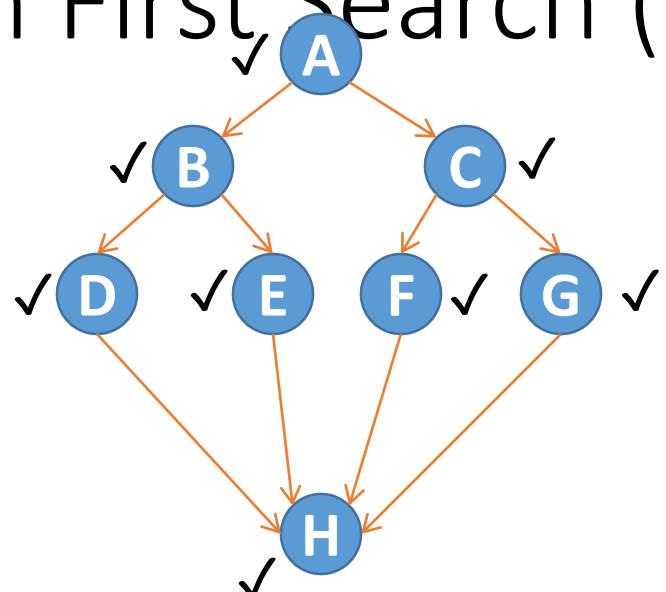


(a) Graph



(b) Stack contents

Depth First Search (DFS)



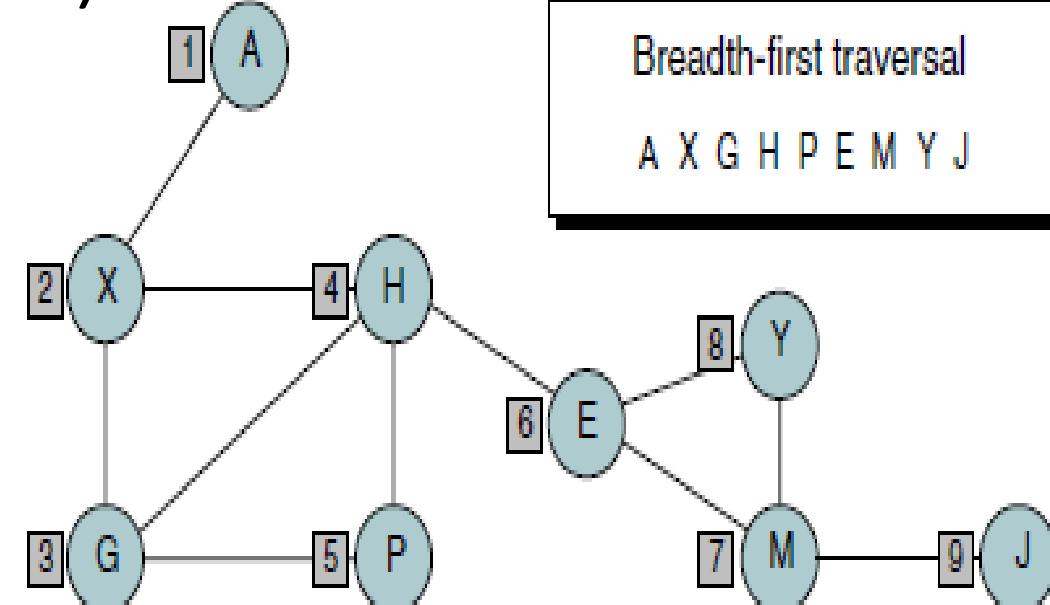
A B D C F E

Breadth First Search (BFS)

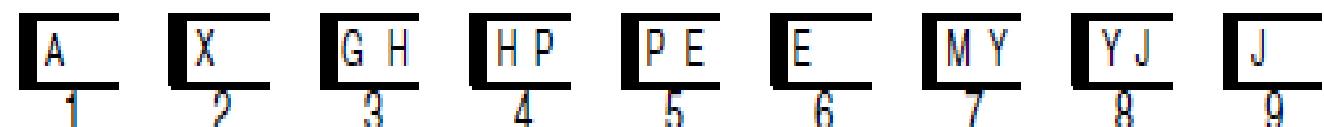
- Steps:

1. Begin by enqueueing vertex A in the queue.
2. Then loop, dequeuing the queue and processing the vertex from the front of the queue. After processing the vertex, **place all of its adjacent vertices into the queue**. Thus, at step 2, dequeue vertex X, process it, and then place vertices G & H in the queue. In step 3, process vertex G.

3. When **queue is empty**, **traversal is complete**.

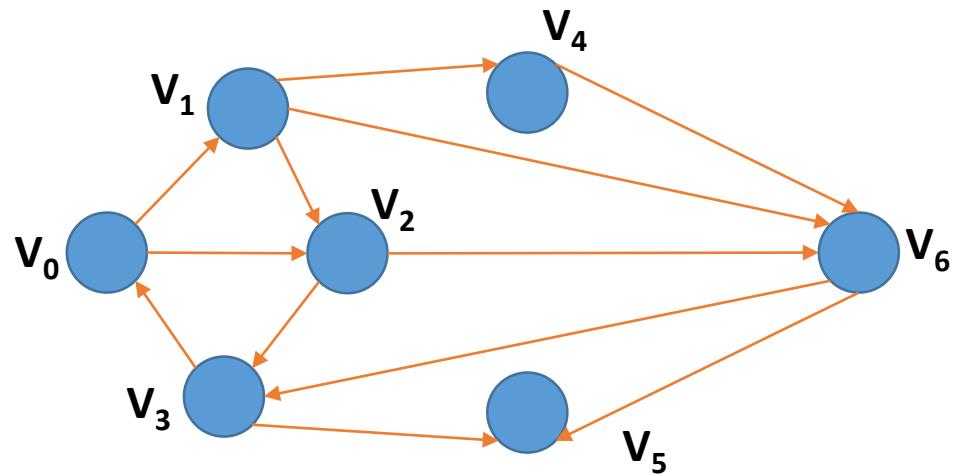
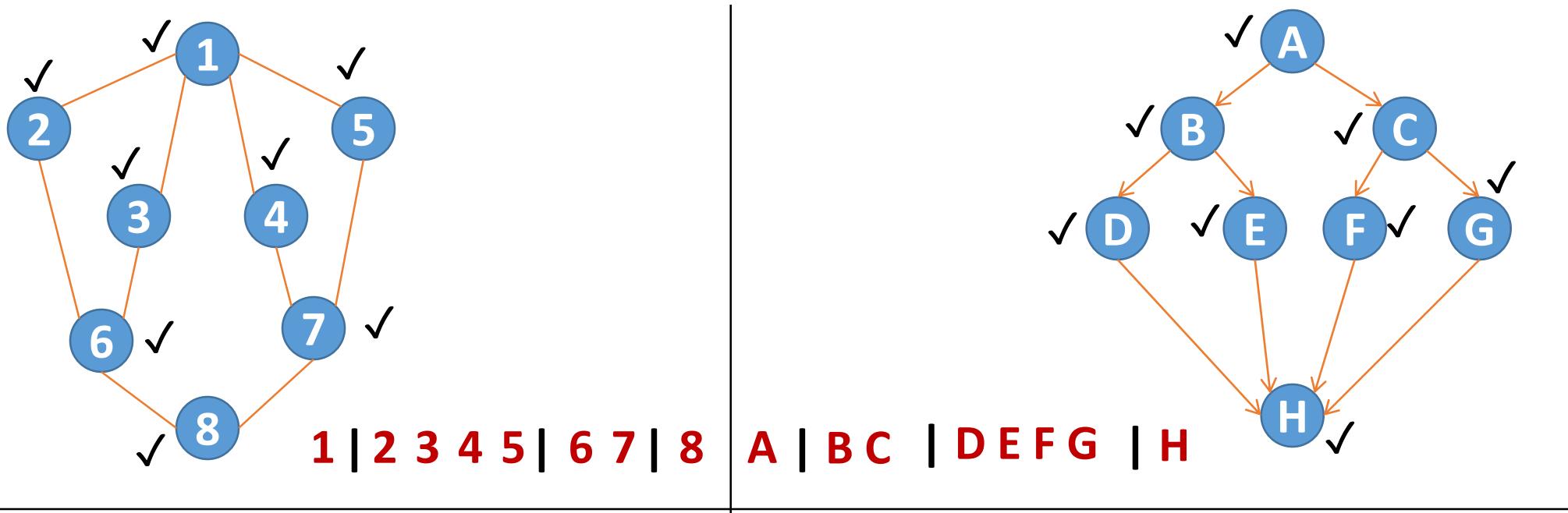


(a) Graph



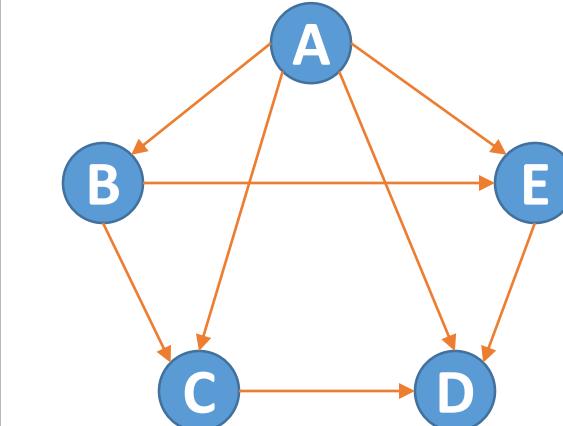
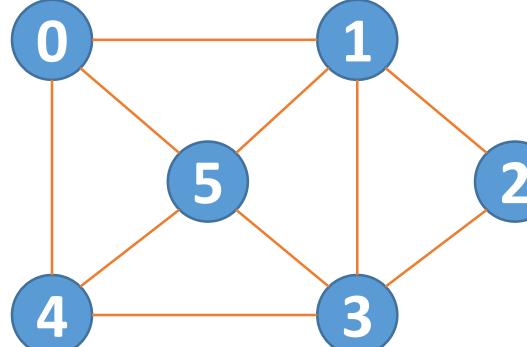
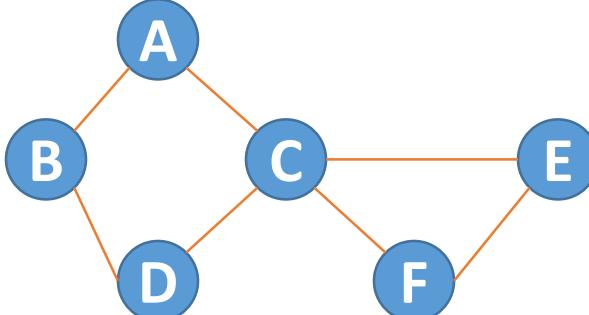
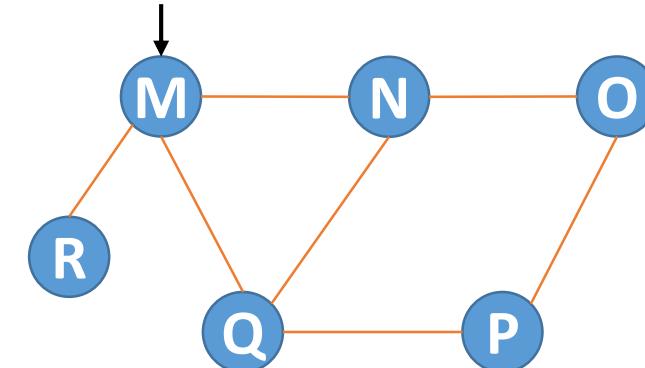
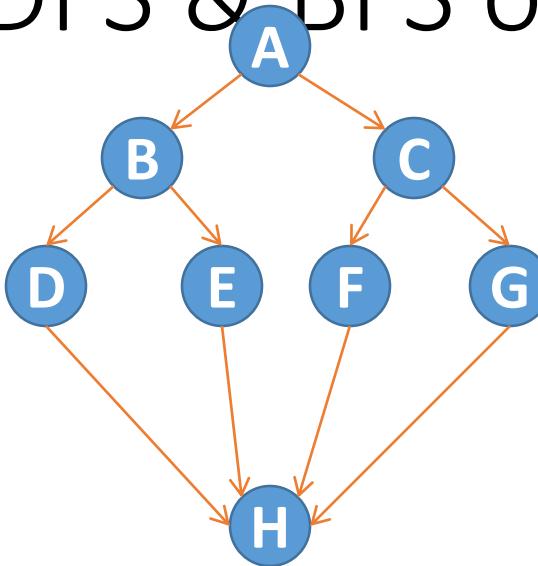
(b) Queue contents

Breadth First Search (BFS)



$v_0 | v_1 v_2 | v_4 v_6 v_3 | v_5$

Write DFS & BFS of following Graphs



End.

Graph storage structures.

Representations of graphs

- Choice between two standard ways to represent a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$:
 - As an adjacency matrix, or
 - As a collection of adjacency lists.
- Either way applies to both directed and undirected graphs.
- Because the adjacency-list representation provides a compact way to represent **sparse graphs**, those for which $|\mathbf{E}|$ is much less than $|\mathbf{V}|^2$, it is usually the method of choice.
- An adjacency-matrix representation is chosen, however, when **the graph is dense**, $|\mathbf{E}|$ is close to $|\mathbf{V}|^2$ or when we need to be able to tell quickly if there is an **edge connecting two given vertices**.

Graph Storage Structures

- To represent a graph, need to store ***two sets***.
 - First set represents the **vertices of the graph**, and
 - Second set represents the **edges or arcs**
- The two most common structures used to store these sets are arrays and linked lists.
- This is a major limitation.
 - Although the arrays offer some simplicity and processing efficiencies, the number of vertices must be known in advance.
 - Only one edge can be stored between any two vertices.

1. Adjacency Matrix.

- The adjacency matrix uses:
 - **A vector** (one-dimensional array) for the **vertices**, and
 - **A matrix** (two-dimensional array) to store the **edges**.
- If two vertices are adjacent, that is, if there is **an edge** between them, the intersect has a **value of 1**
- If there is **no edge** between them, the intersect is **set to 0**.
- If the graph is directed, the **intersection** in the adjacency matrix indicates the **direction**.

1. Adjacency Matrix.

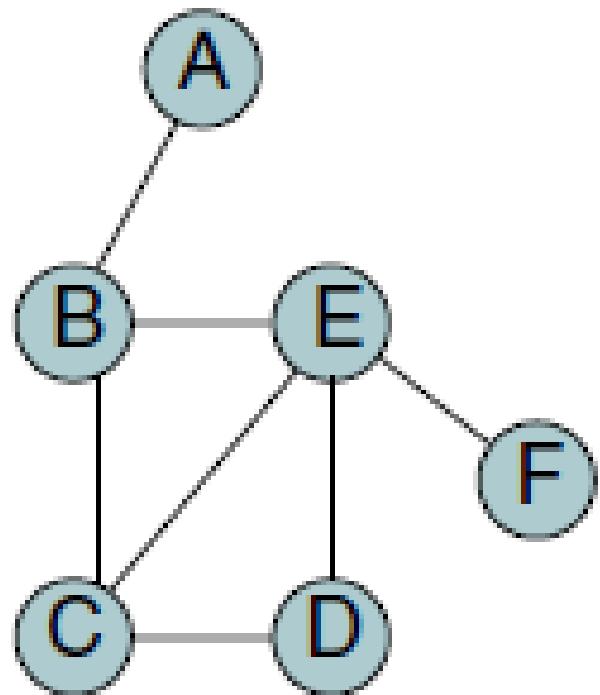
- For an adjacency matrix representation of a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, we assume that the vertices are numbered $1, 2, \dots, |\mathbf{V}|$ in some arbitrary manner.
- Then the adjacency matrix representation of a graph G ,
 - Consists of $|\mathbf{V}| * |\mathbf{V}|$ matrix $A = (a_{ij})$ such that,

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E ; \\ 0 & \text{otherwise} \end{cases}$$

1. Adjacency Matrix.

- Number of 1's in the adjacency matrix:
 - **Undirected graph:** Sum of degrees of all vertices
 - **Directed graph:** Sum of outdegrees of all vertices

Adjacency Matrix for undirected graph.



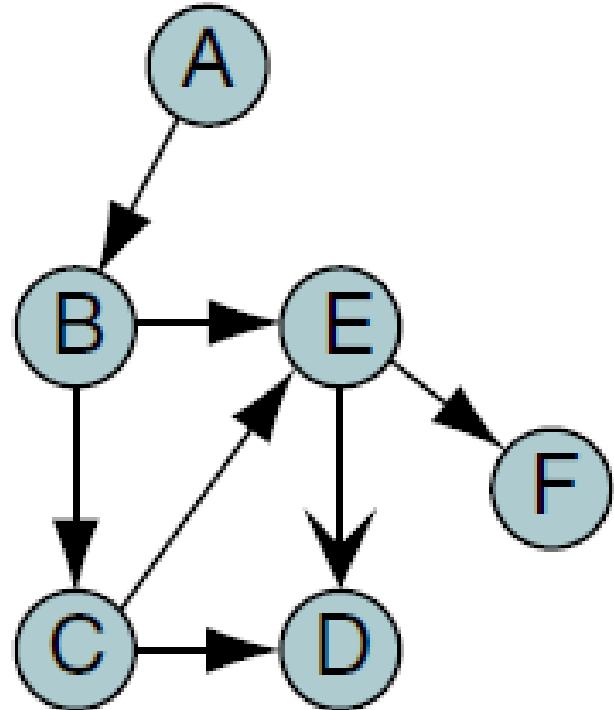
A
B
C
D
E
F

Vertex vector

Adjacency matrix for nondirected graph

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency Matrix for directed graph.



A
B
C
D
E
F

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Vertex vector

Adjacency matrix for nondirected graph

2. Adjacency List.

- The adjacency List uses:
 - *A linked list* to store the *vertices*, and
 - *A two-dimensional linked list* to store the *arcs*.
- The **vertex list** is a *singly linked list* of the vertices in the list.
- Depending on the application, it could also be implemented using doubly linked lists or circularly linked lists.
- The **pointer at the left** of the list *links the vertex* entries.
- The **pointer at the right** in the vertex is a *head pointer to a linked list of edges from the vertex*.

2. Adjacency List.

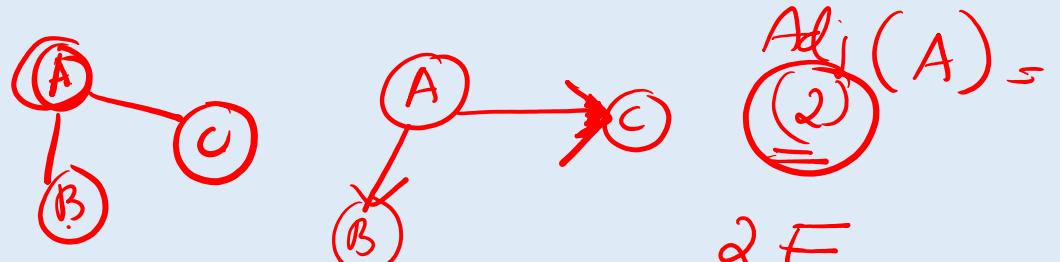
- The adjacency-list representation of a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of an array **Adj** of $|\mathbf{V}|$ lists, one for each vertex in \mathbf{V} .

- For each $u \in \mathbf{V}$, the adjacency list,

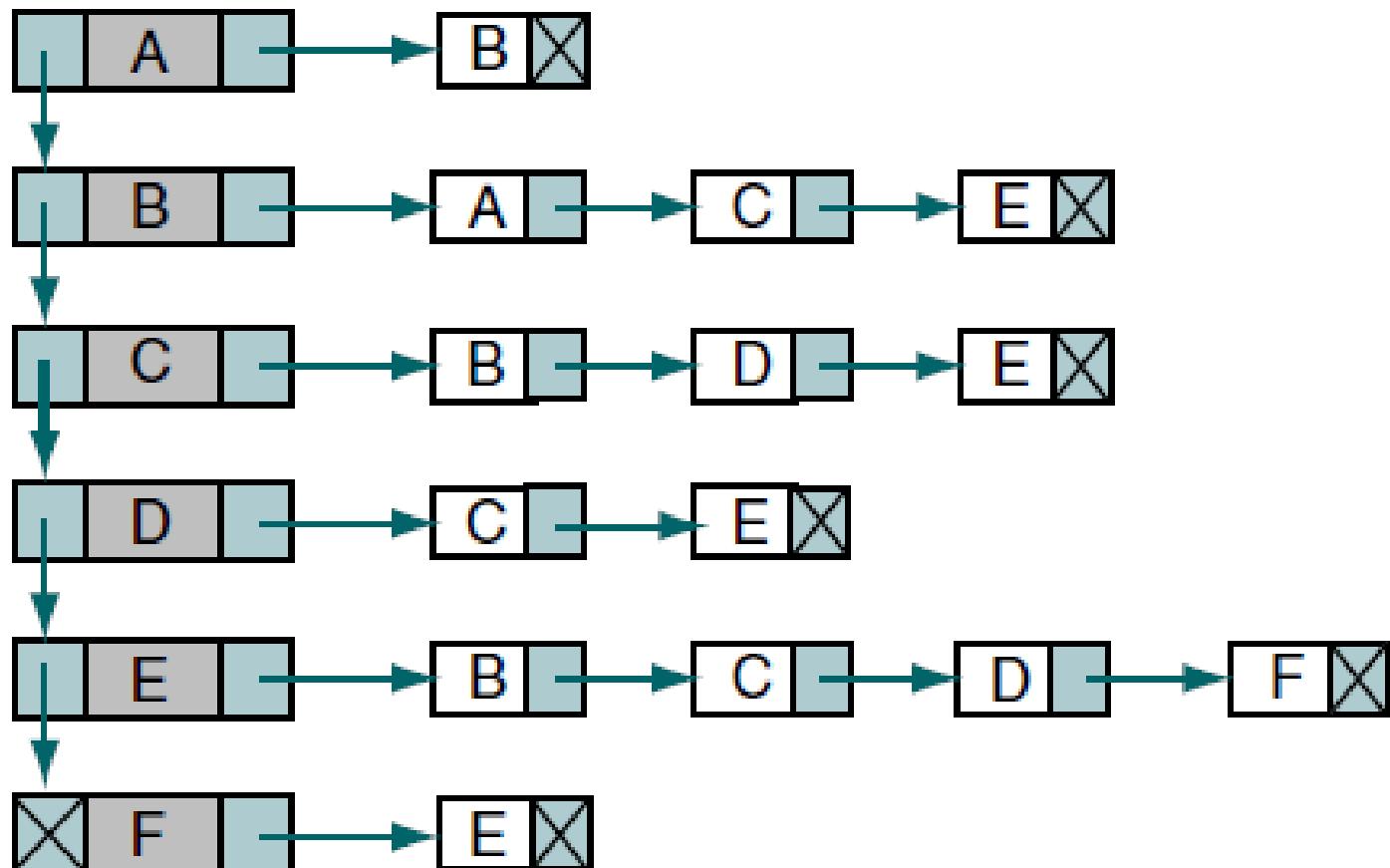
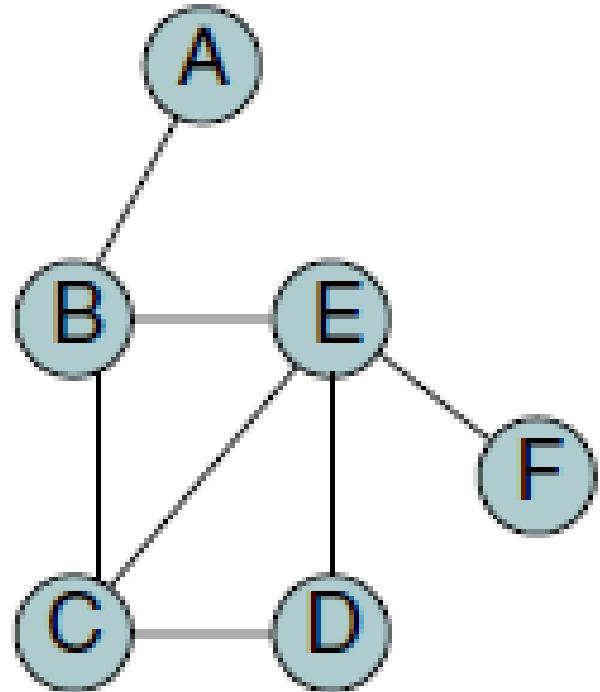
- $\text{Adj}[u]$ contains all the vertices v s.t. there is an edge $(u, v) \in E$. That is, $\text{Adj}[u]$ consists of all the vertices adjacent to u in G . (Alternatively, it may contain pointers to these vertices).

- Sum of lengths of all adjacency lists

- $|E|$ for **directed graph** (an edge (u, v) appears in $\text{Adj}[u]$ and not in $\text{Adj}[v]$)
 - $2|E|$ for G is **undirected** (an edge (u, v) appears in both $\text{Adj}[u]$ and $\text{Adj}[v]$)



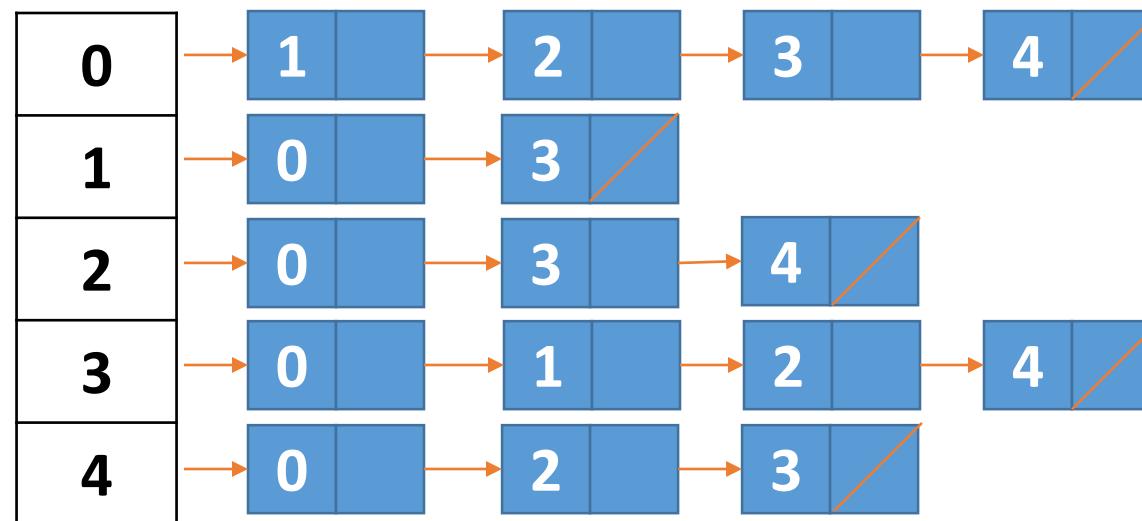
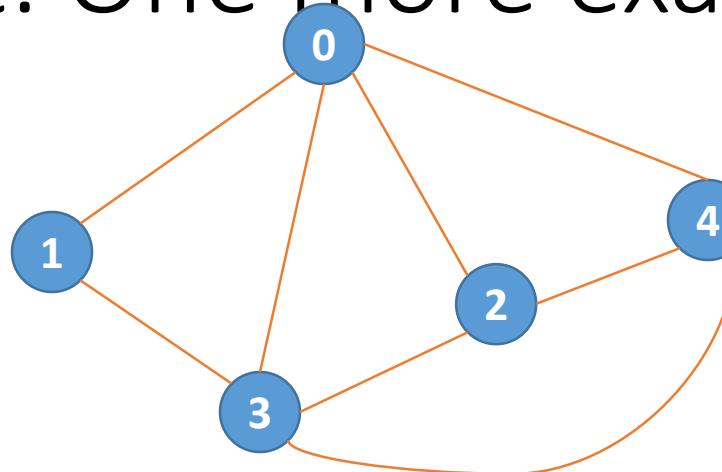
Adjacency List for undirected graph.



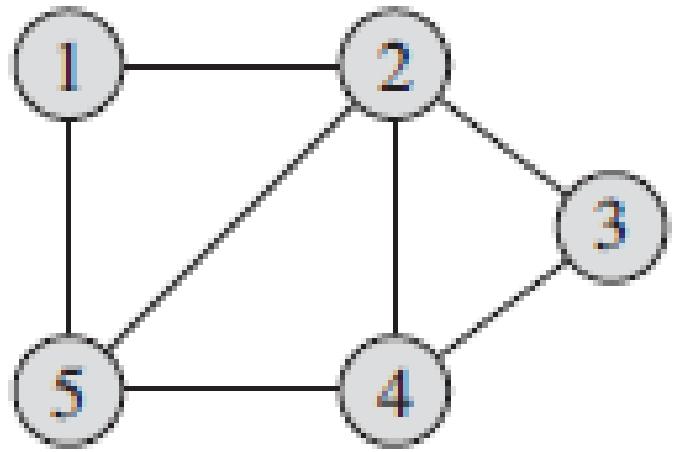
Vertex list

Adjacency list

Adjacency List: One more example.

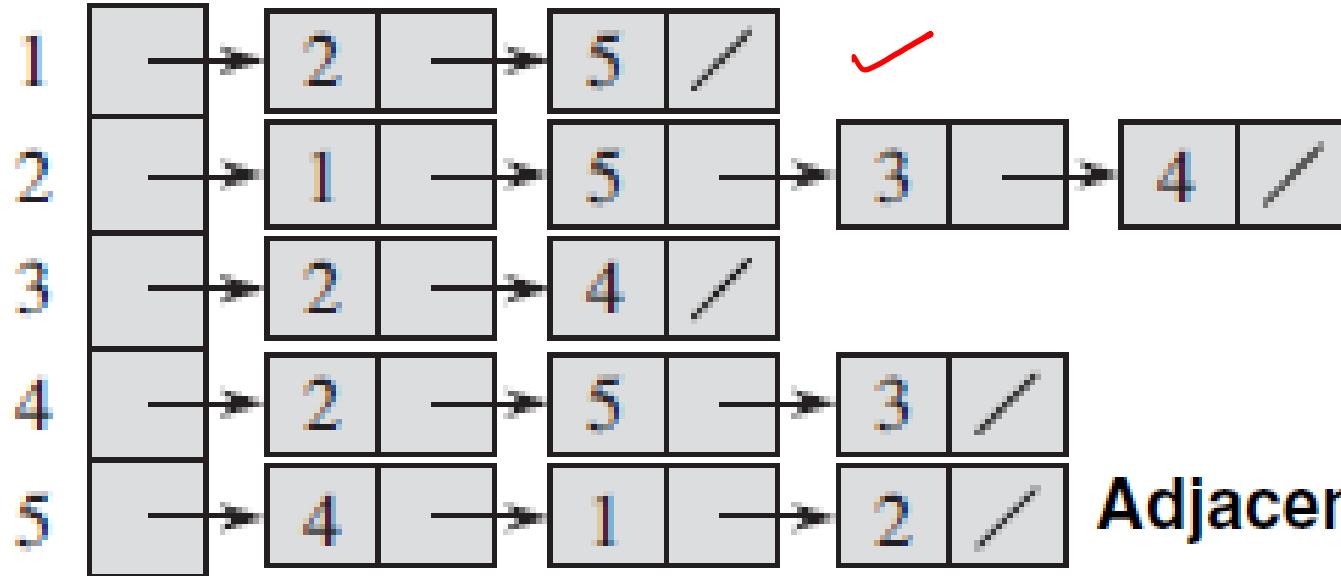


Representations for undirected graph: Example 1.



1
2
3
4
5

Vertex vector

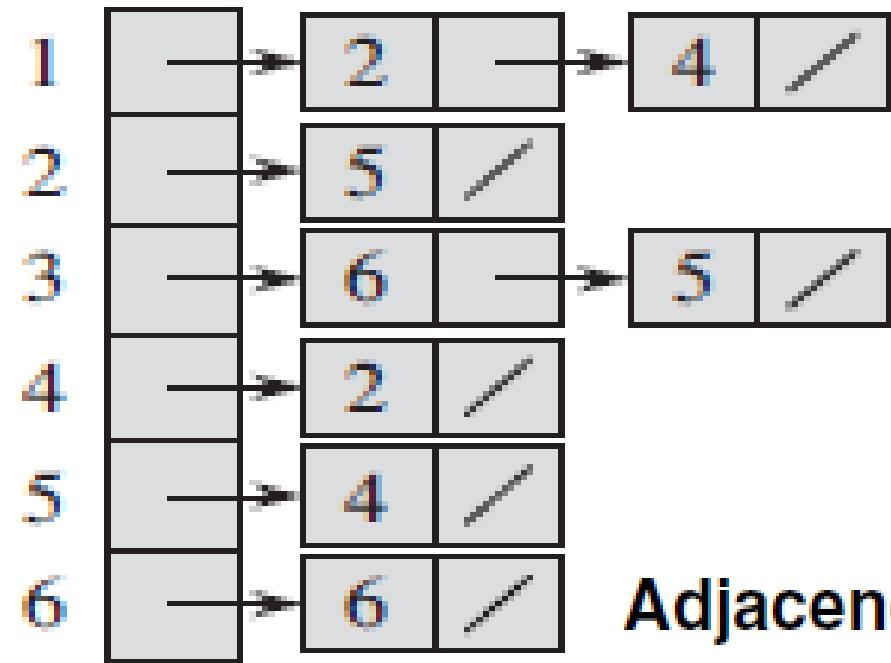
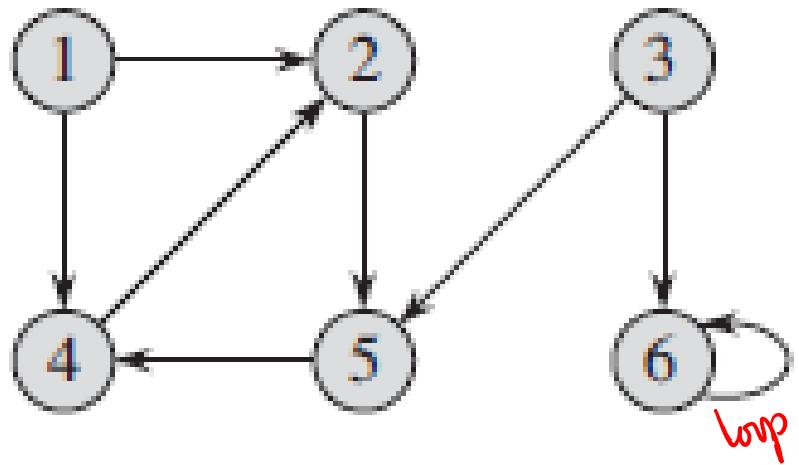


Adjacency list

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

Adjacency matrix

Representations for directed graph: Example 2.



Adjacency list

1
2
3
4
5
6

Vertex vector

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



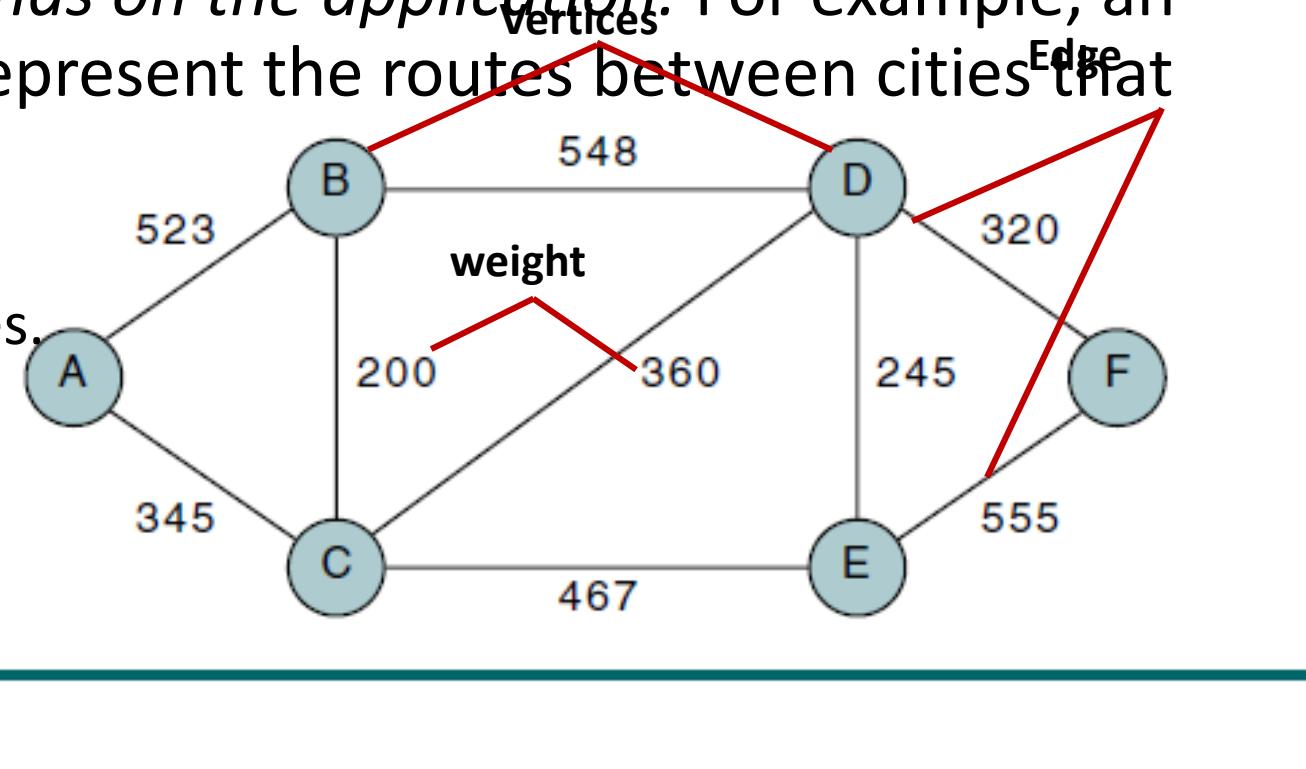
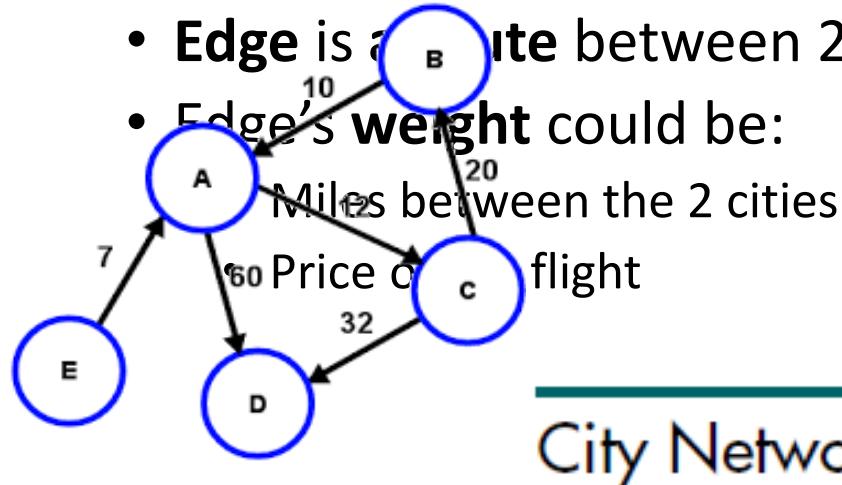
Adjacency matrix

Networks.

Networks.

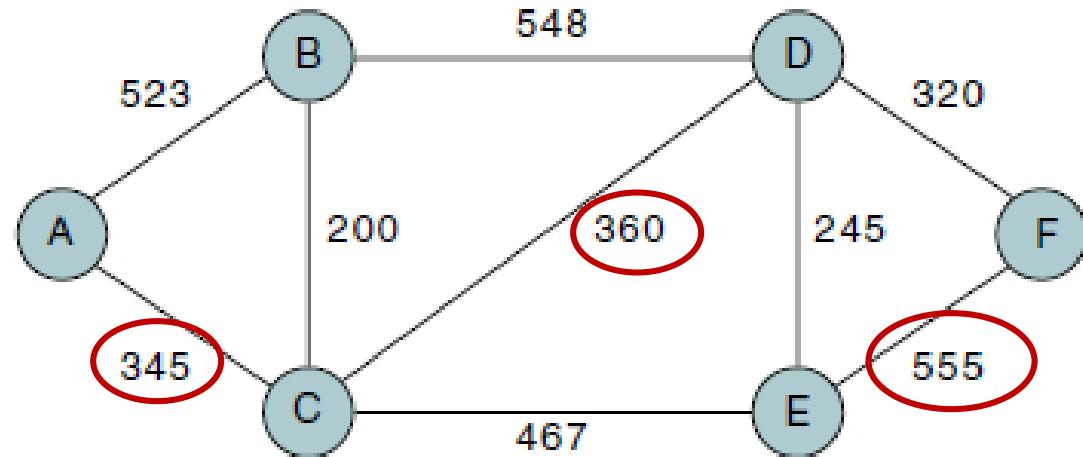
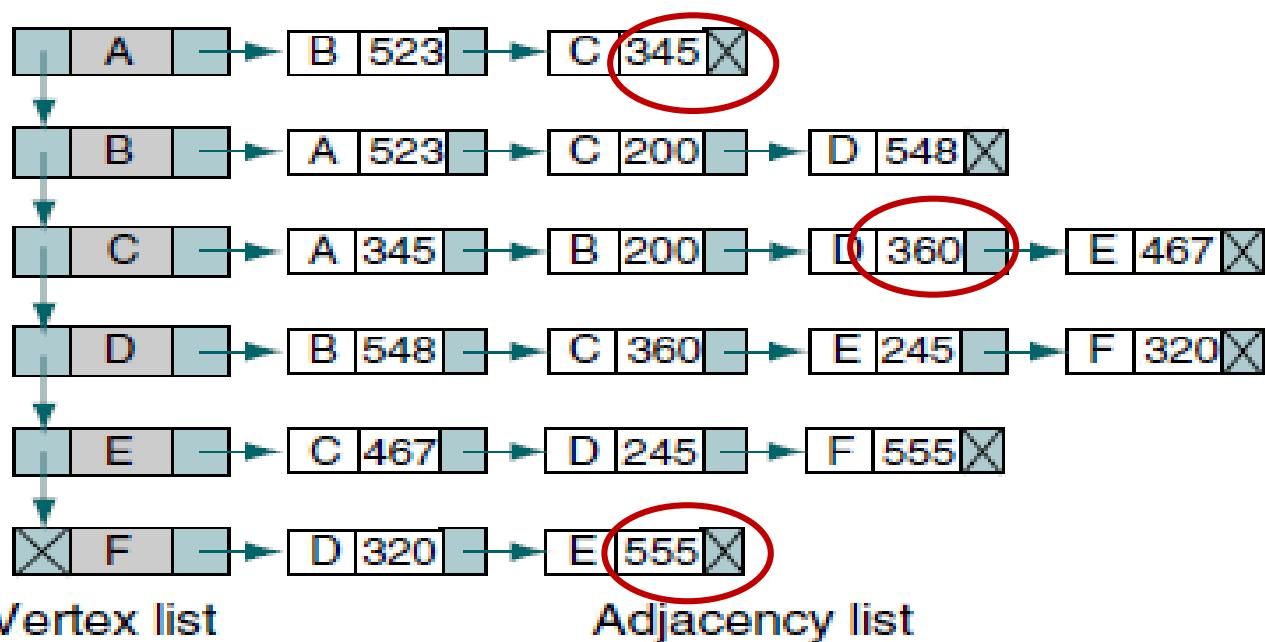
- A **network** is a **graph** whose **lines** are **weighted** or called **weighted graph**.
- *Meaning of the weights depends on the application.* For example, an airline might use a graph to represent the routes between cities that it serves. Here,

- Vertices represent **cities**, and
- Edge is a **line** between 2 cities.
- Edge's **weight** could be:



STORING WEIGHTS IN GRAPH STRUCTURES

- Since **weight is an attribute** of an edge, it is stored in the structure that contains the edge.
- In adjacency matrix, weight is stored as the intersection value.
- In adjacency list, stored as the value in the adjacency linked list.



	A	B	C	D	E	F
A	0	523	345	0	0	0
B	523	0	200	548	0	0
C	345	200	0	360	467	0
D	0	548	360	0	245	320
E	0	0	467	245	0	555
F	0	0	0	320	555	0

Vertex vector

Adjacency matrix

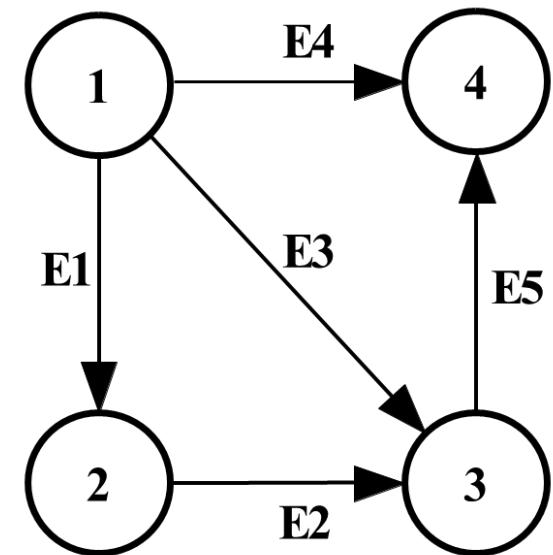
INCIDENCE MATRIX OF A DIGRAPH.

Incidence matrix is a $|E| \times |V|$ matrix $B = (b_{ij})$ such that,

	1	2	3	4
E1	-1	1	0	0
E2	0	-1	1	0
E3	-1	0	1	0
E4	-1	0	0	1
E5	0	0	-1	1

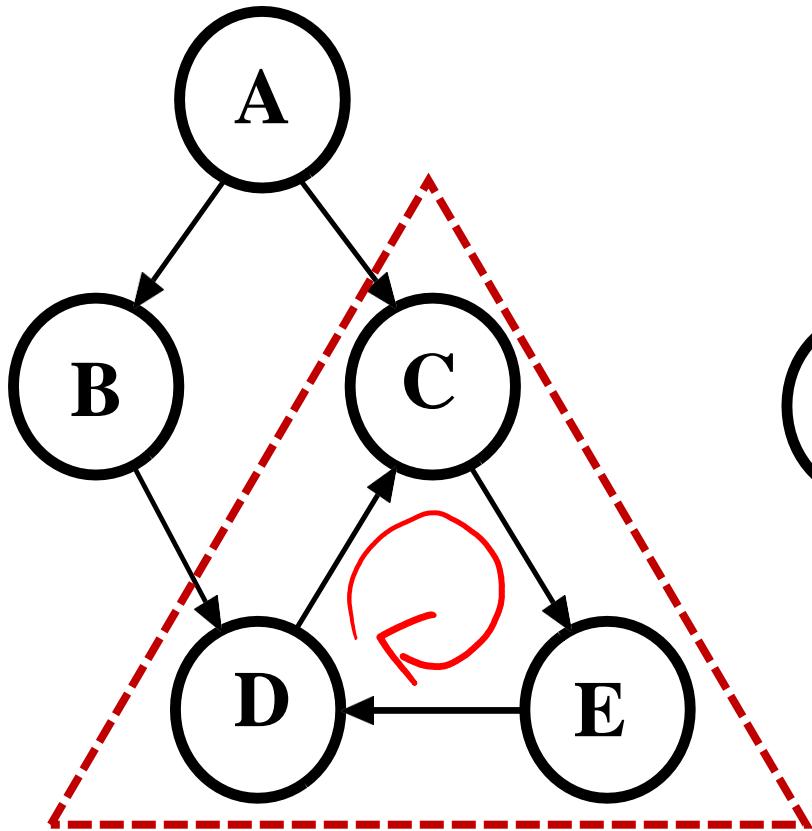
$$b_{ij} = \begin{cases} -1 & \text{if edge } i \text{ leaves vertex } j \\ 1 & \text{if edge } i \text{ enters vertex } j \\ 0 & \text{otherwise} \end{cases}$$

It has one column for each vertex and one row for each edge



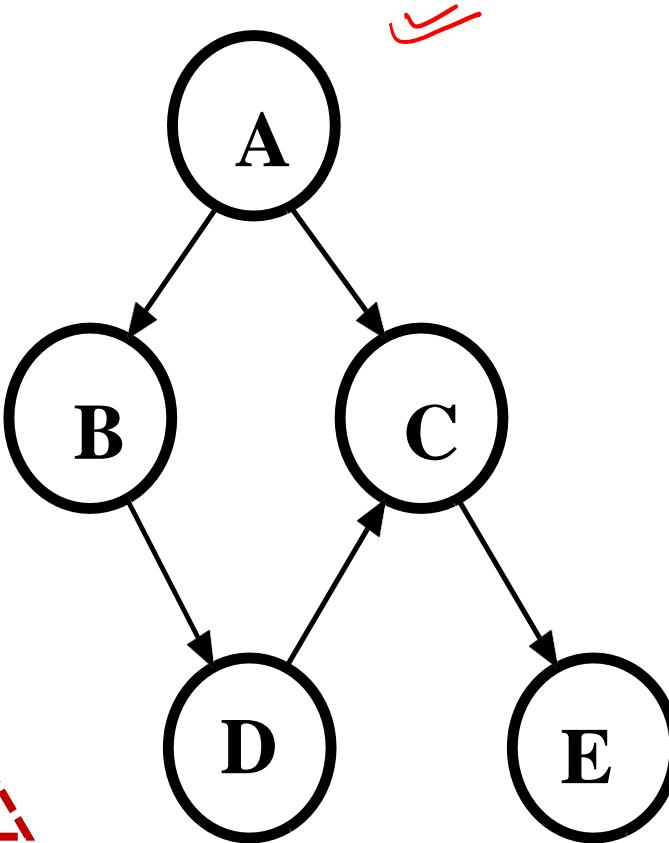
With no self loops

- Directed Acyclic Graphs(DAG).

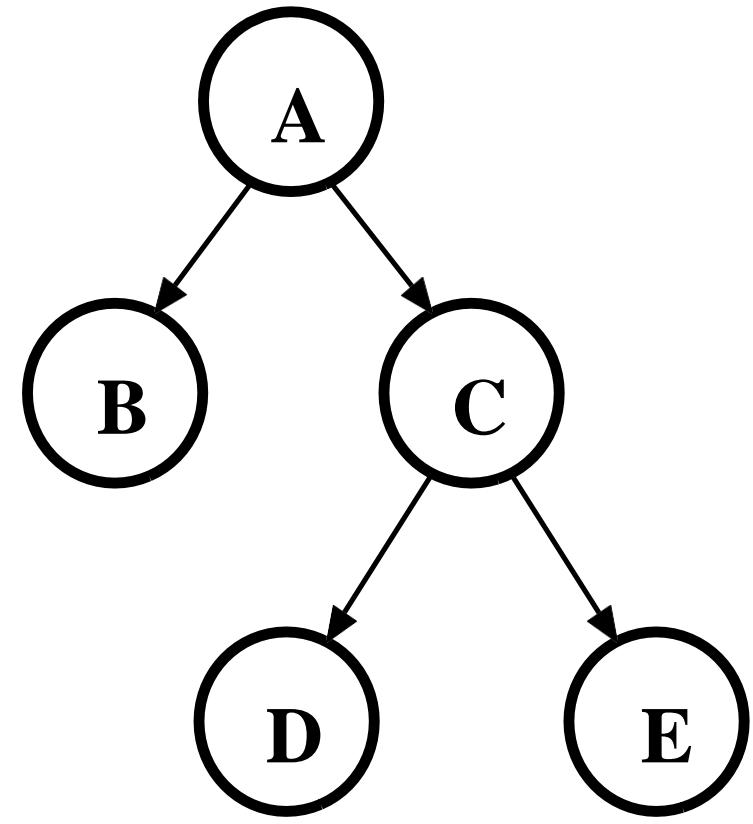


Directed Graph

DAG
+



Directed Acyclic Graph



Tree

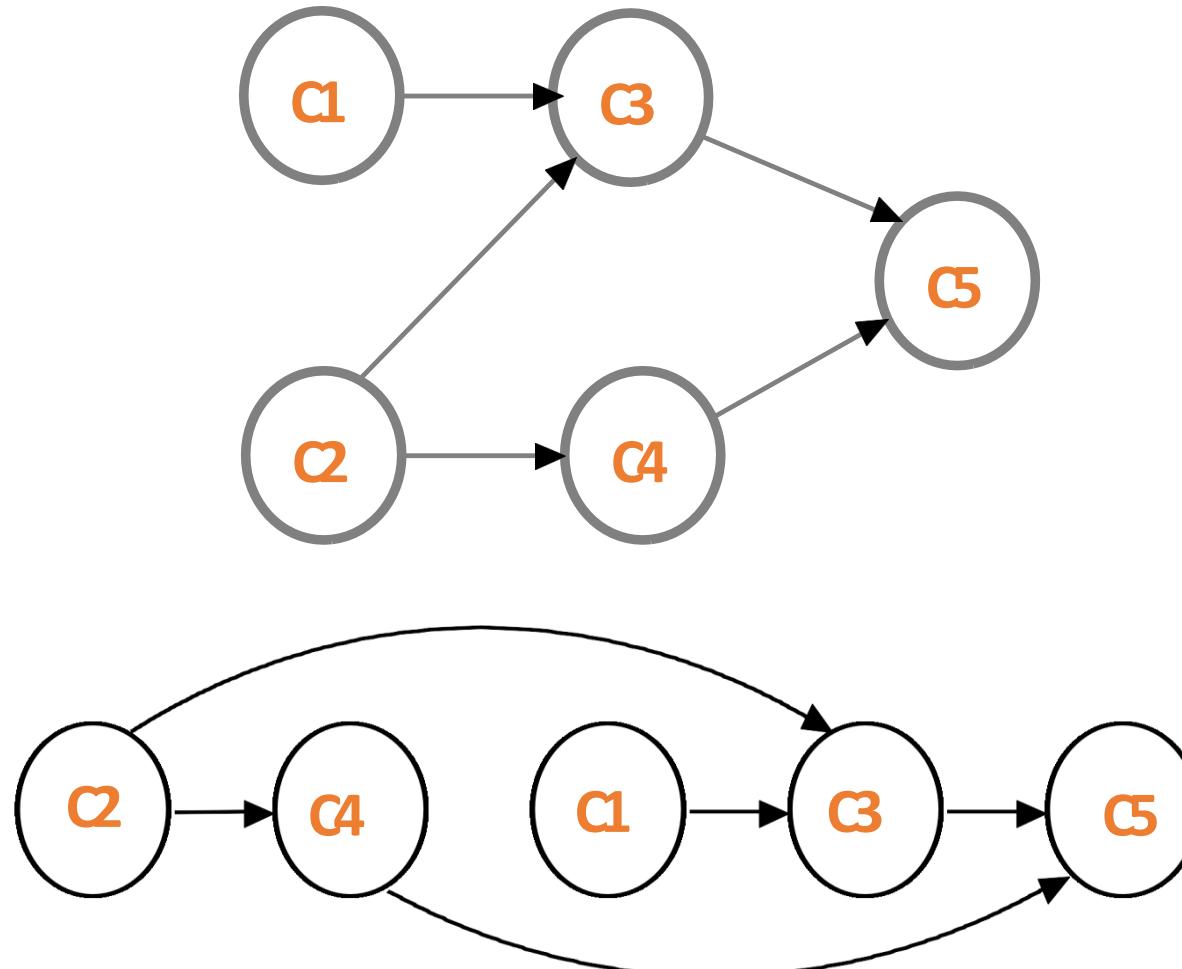
Topological sort (Kahn's Algorithm)

- *Another Graph Search Method.*
- **A topological sort (ordering)** of a directed acyclic graph(DAG) $G = (V, E)$ is:
 - A linear ordering of all its vertices such that if G contains an edge (u, v) , then u appears before v in the ordering.
 - Vertices are **ordered along a horizontal line** so that all **directed edges** go from **left to right**: shows the **precedence** among events
 - If a **graph contains a cycle**, linear **ordering** is **not possible**.

Topological sort (Ex 1)

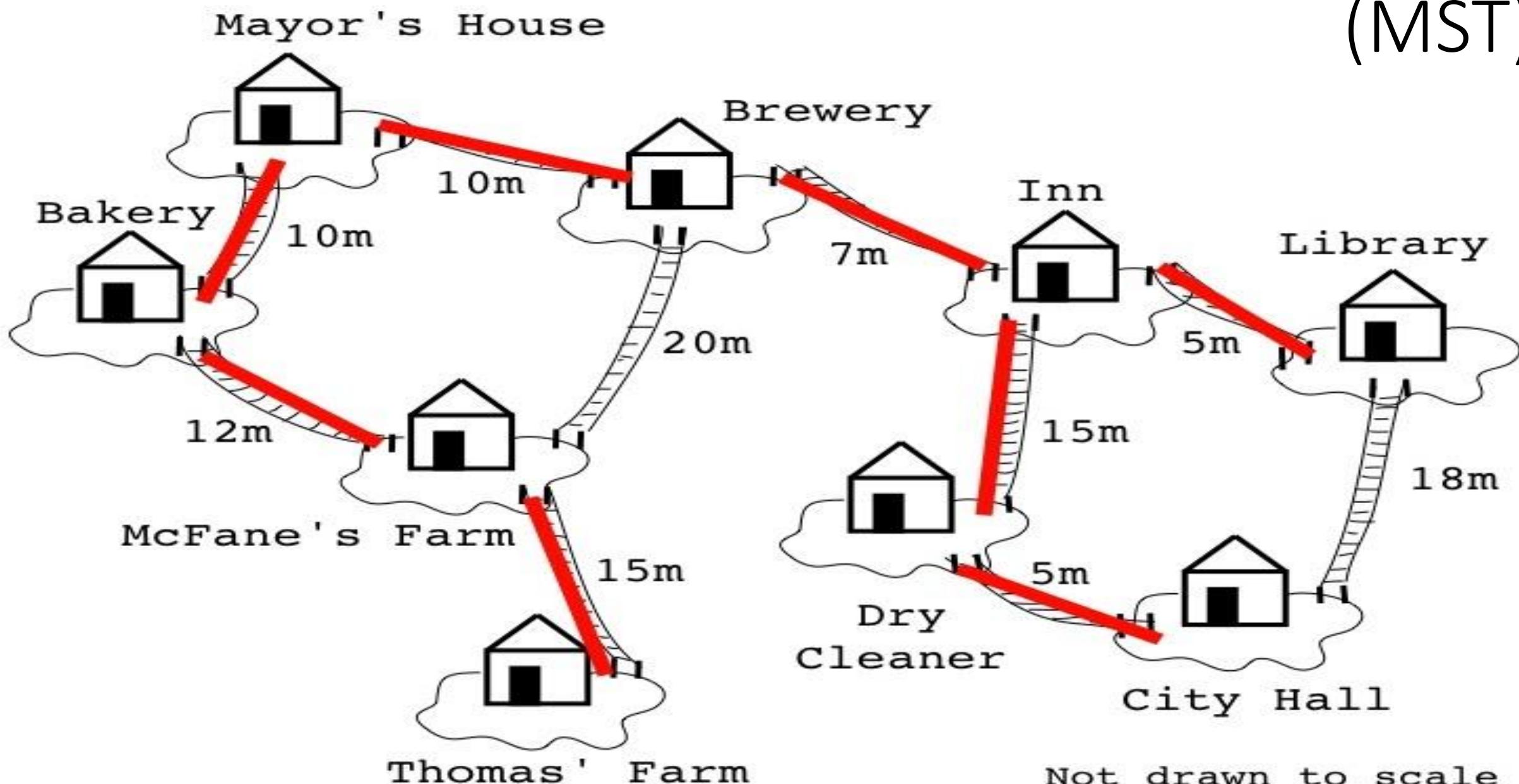
(A University curriculum may have courses that require other courses as prerequisites.

Course	Prerequisites
C1	None
C2	None
C3	C1, C2
C4	C2
C5	C3, C4



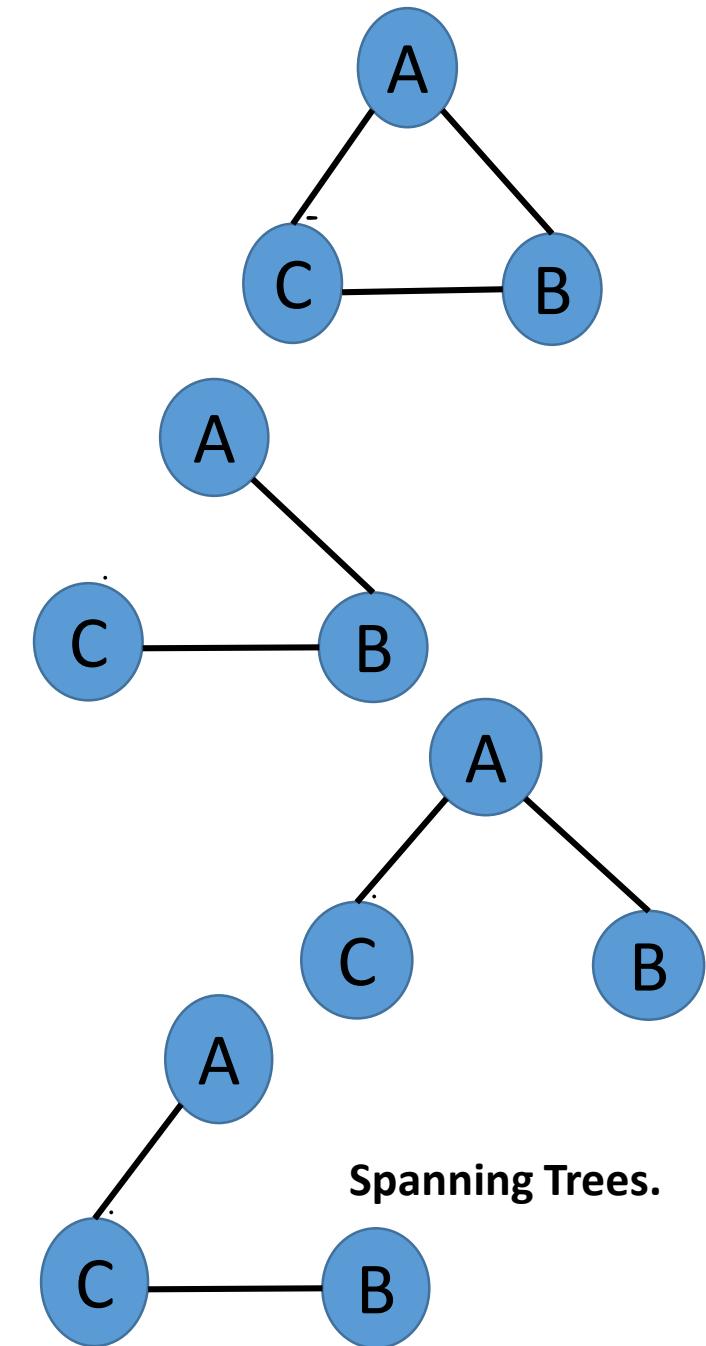
Minimum spanning Trees.

MINIMUM Spanning trees (MST).



Spanning Tree.

- Let $G = (V, E)$ be an **connected undirected** graph.
- A subgraph of G that is a tree and contains all vertices of G is **a spanning tree of G** .
- A spanning tree contains all vertices in a graph.
- A spanning tree has **$n-1$ edges**, where n is the number of nodes (vertices).
- Every **connected and undirected graph** has ***at least one spanning tree***.
- A graph can have **more than one** spanning tree.
- Spanning tree **should not contain any cycle**. (acyclic).
- **Disconnected** graph does **not have** any spanning tree.

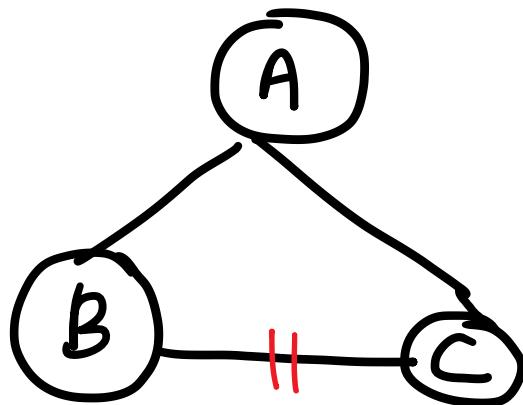
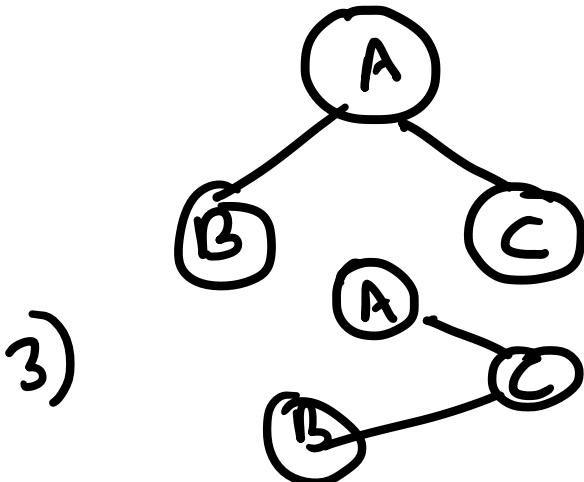


Spanning Tree.

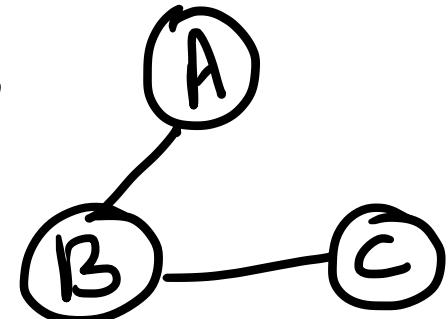
- All possible spanning trees of graph G, have the same number of edges and vertices.
- **Removing one edge** from the spanning tree will make it disconnected. i.e. the spanning tree is **minimally connected**.
- **Adding one edge** to a spanning tree will create a cycle, i.e. the spanning tree is **maximally acyclic**.
- If each has distinct weight, then there will be one and unique Minimum Spanning Tree (MST).
- A **complete undirected graph** can have n^{n-2} number of spanning trees, where n is the number of vertices.
- A spanning tree can be constructed from a complete graph by removing maximum of **e-n+1** edges.

- # Spanning Tree
- A graph can be represented by $G(V, E)$ where V is vertex set and E is edge set.
 - Spanning Tree of this graph $G'(V', E')$ where $V' = V$ and E' is a subset of E ;
 - Also, $E' = |V| - 1$

1) Remove one edge



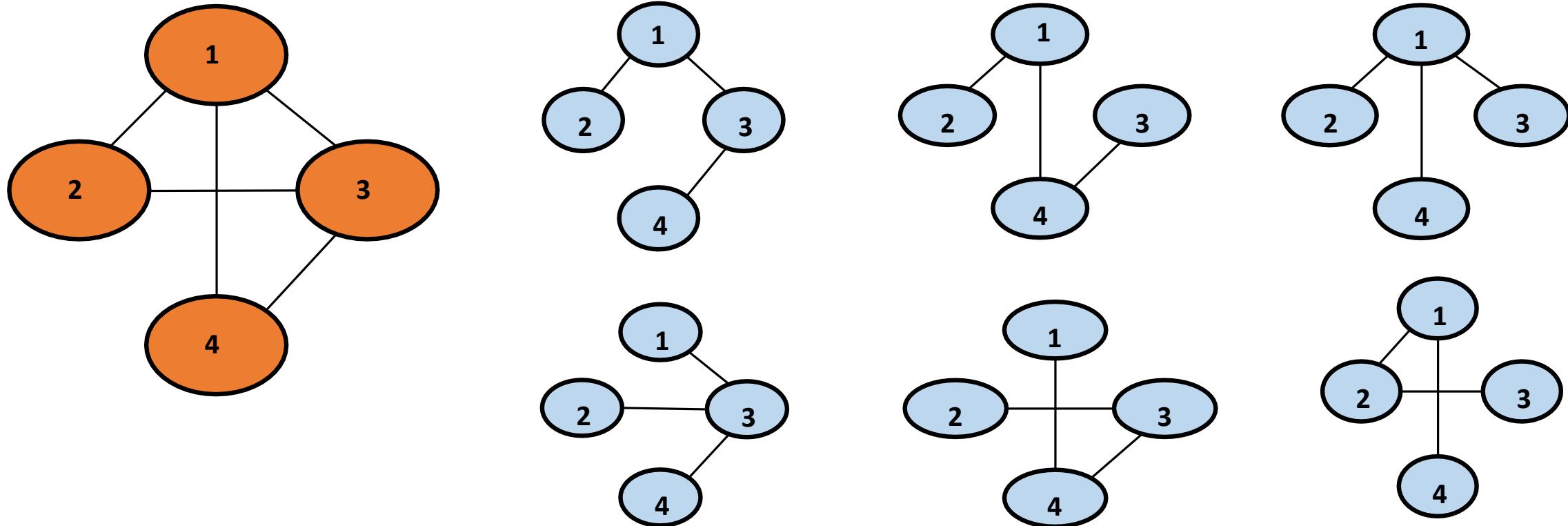
2) Remove another edge



$$n = 3$$

$$\begin{aligned} \text{No. of spanning trees} &= 3^{3-2} \\ &= 3^1 = 3 \end{aligned}$$

Spanning Tree – Connected.



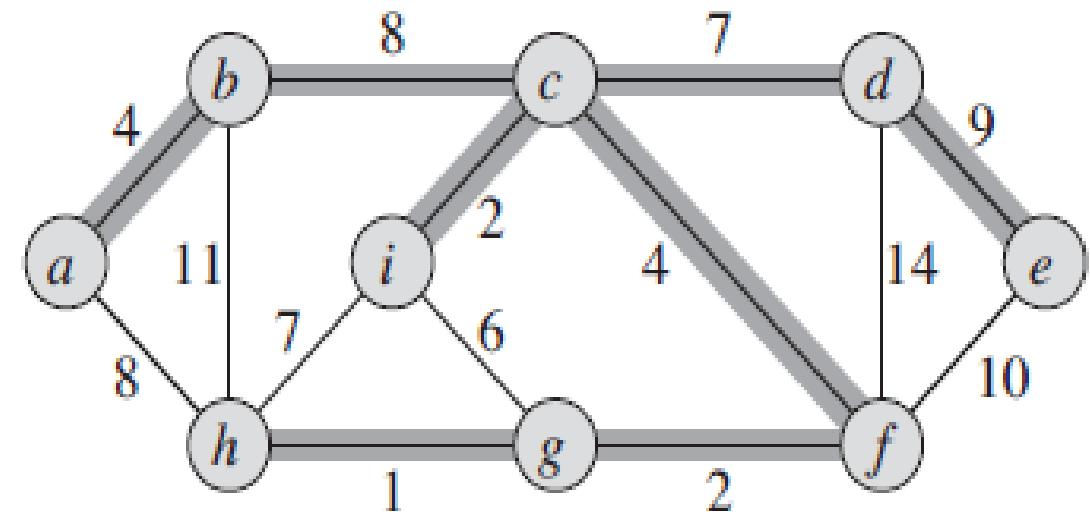
All edges are assumed to have **same weight** (cost)

- If the edges have different (non-negative) weight (cost), a **minimum-cost spanning tree** can be constructed.
- Let $G = (V, E)$ be an weighted, connected and undirected graph. For each edge $(u, v) \in E$, the weight $w(u, v)$ specifies the cost to connect u to v .
- **The acyclic subset (tree) $T \subseteq E$ that connects all of the vertices and whose total weight is minimum $w(T)$ is called the minimum spanning tree of graph G .**

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

A MST for a connected graph.

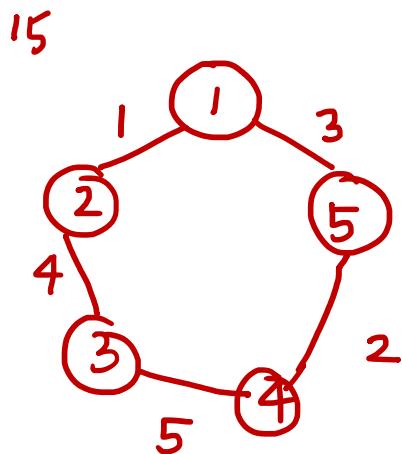
The weights on edges are shown, and the edges in a minimum spanning tree are shaded.



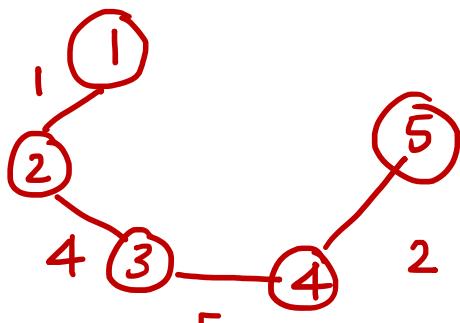
The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

Minimum Spanning Tree or MST

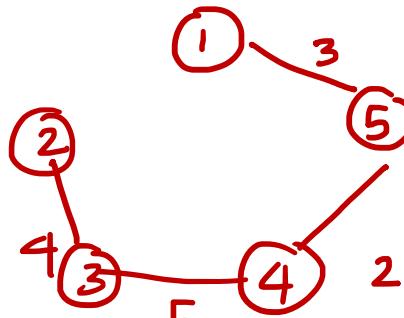
If the edges have different (non-negative) weight (cost), a minimum-cost spanning tree can be constructed.



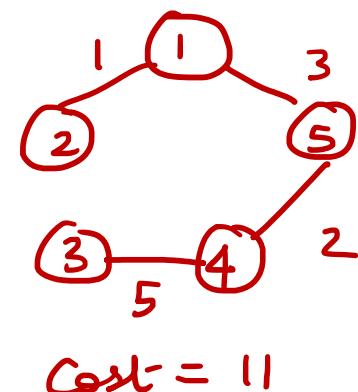
Start at any vertex, remove one edge



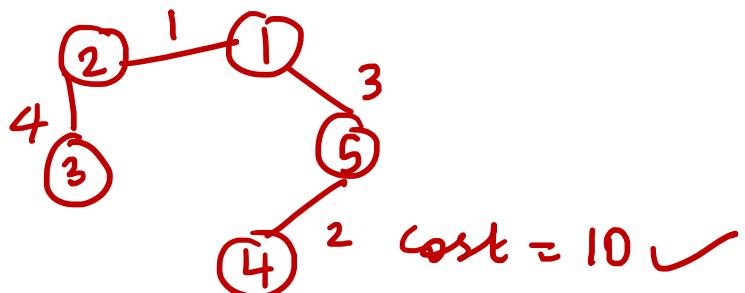
Cost = 12



Cost = 14 ✓



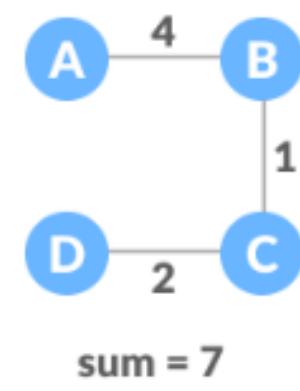
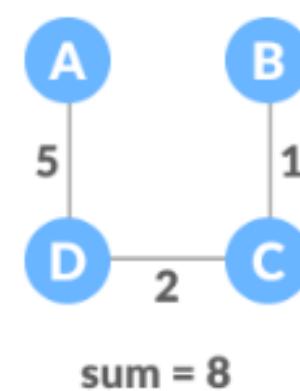
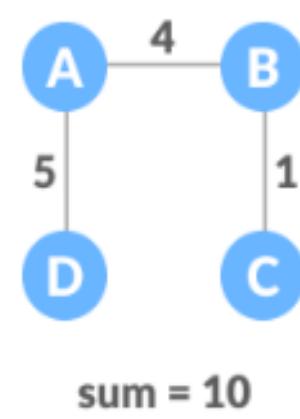
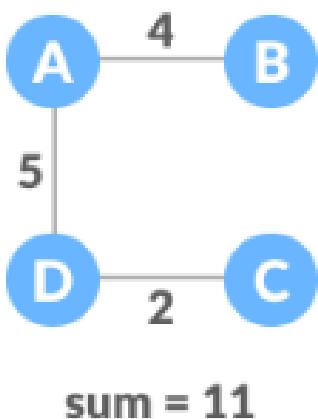
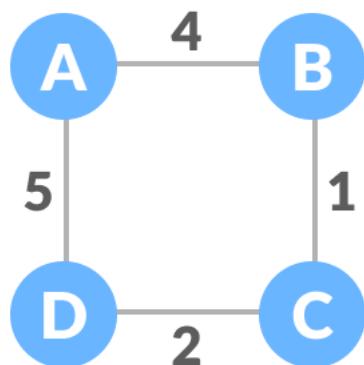
Cost = 11



MST

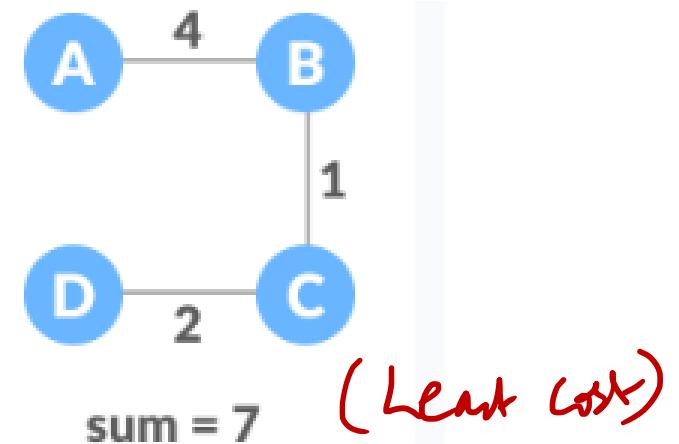
MST with costs

The cost of the spanning tree is the sum of the weights of all the edges in the tree.

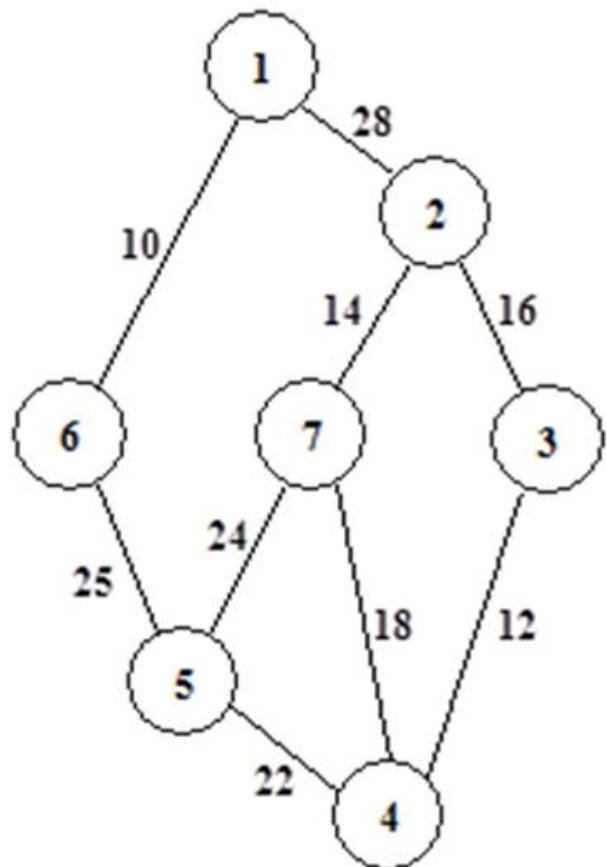


12

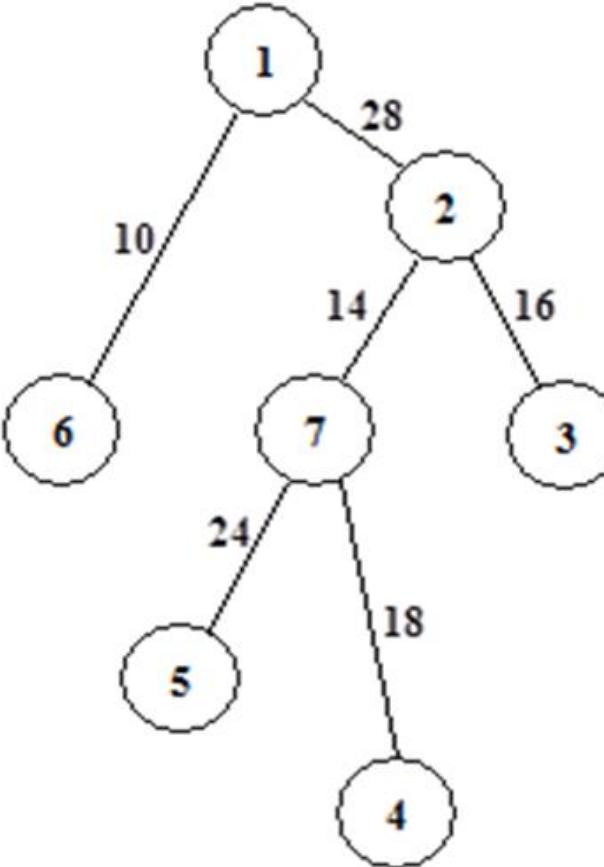
MST from the above spanning trees is:



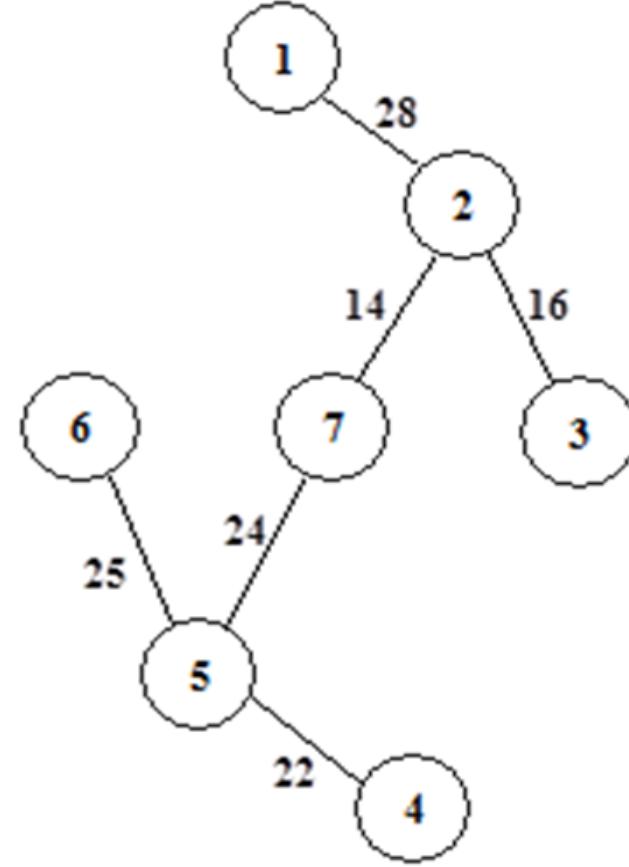
Minimum Spanning Tree or MST.



Graph G



Spanning Tree (Cost =110)

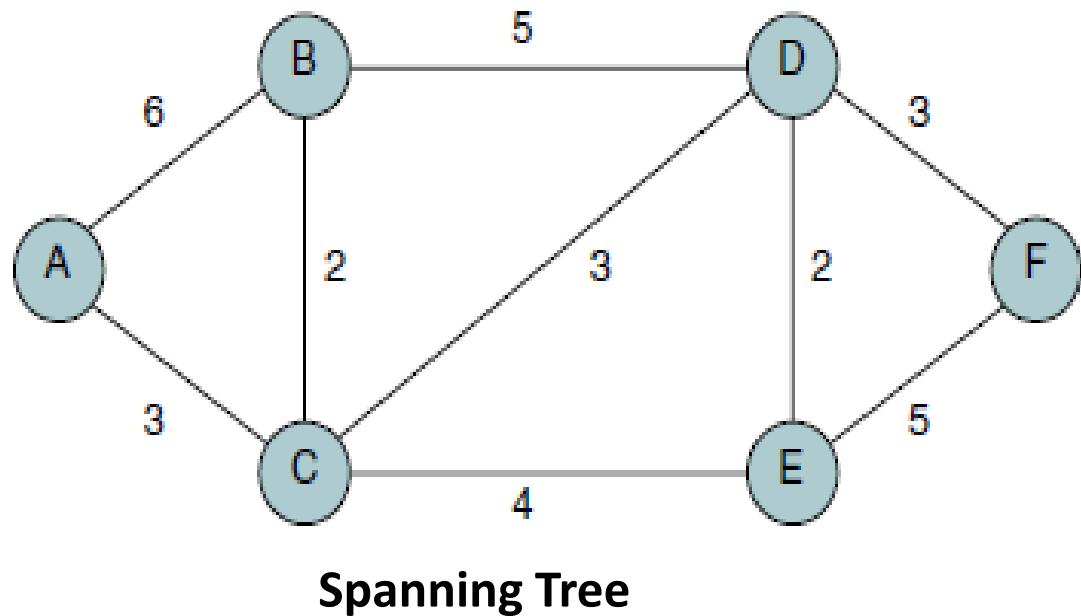


Spanning Tree (Cost =129)

Minimum Spanning Tree Applications

- Given a **network of computers**, tree that connects all computers is created. The MST gives the shortest length of cable that can be used to connect all of the computers while ensuring that there is a path between any two computers.
- Electronic circuit designs** often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, an arrangement of $n-1$ wires can be used, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.
- A Telecommunications company trying to **lay cable** in a new neighborhood, with constraint to bury cable only along certain paths (e.g. roads). Then a graph containing points (e.g. houses) connected by those paths. Some paths are more expensive with longer lengths, or cable to be buried deeper; these paths would be represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible. A MST would be one with the lowest total cost, representing the least expensive path for laying the cable.

- **Minimum Spanning Tree: Creation.** To create a minimum spanning tree in a strongly connected network, that is, in a network in which there is a path between any two vertices, the edges for the minimum spanning tree are chosen so that the following properties exist:
 1. Every vertex is included.
 2. The total edge weight of the spanning tree is the minimum possible that includes a path between any two vertices.

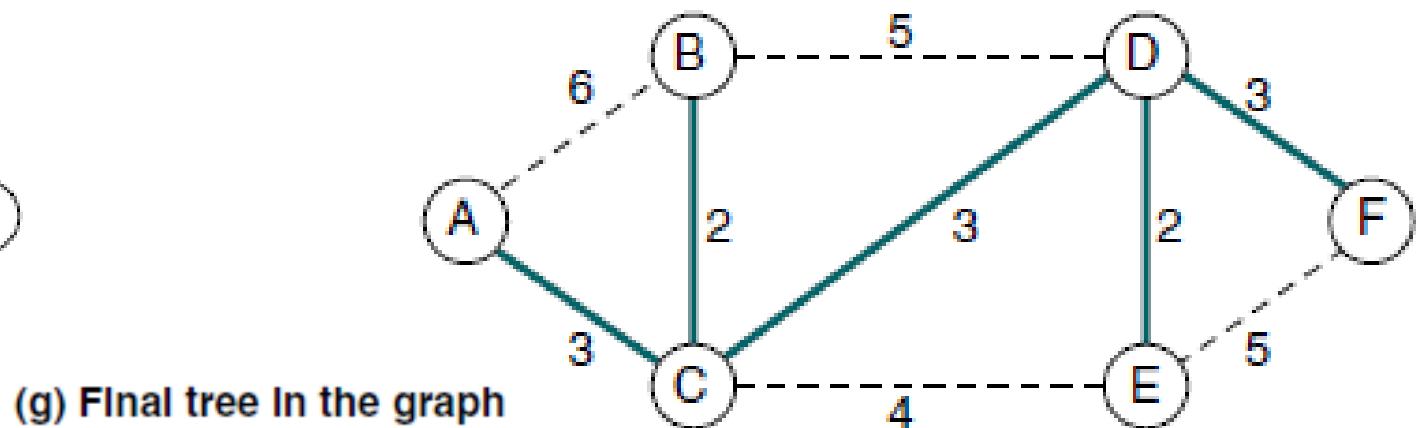
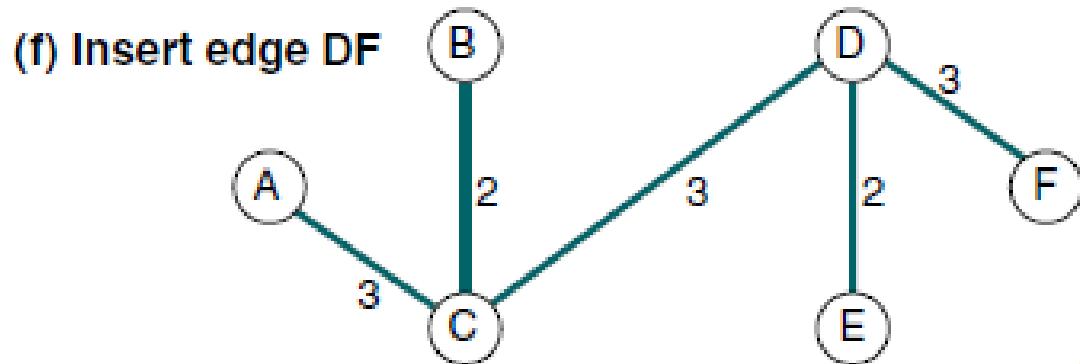
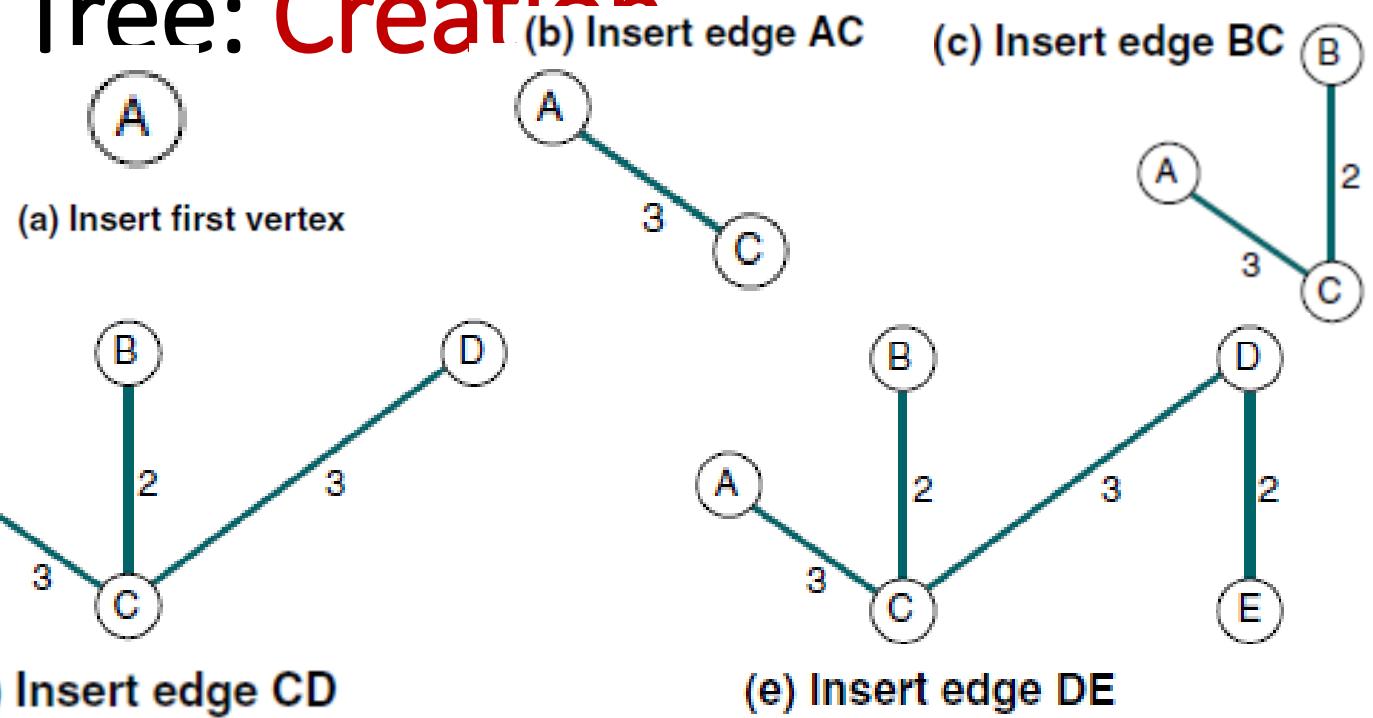
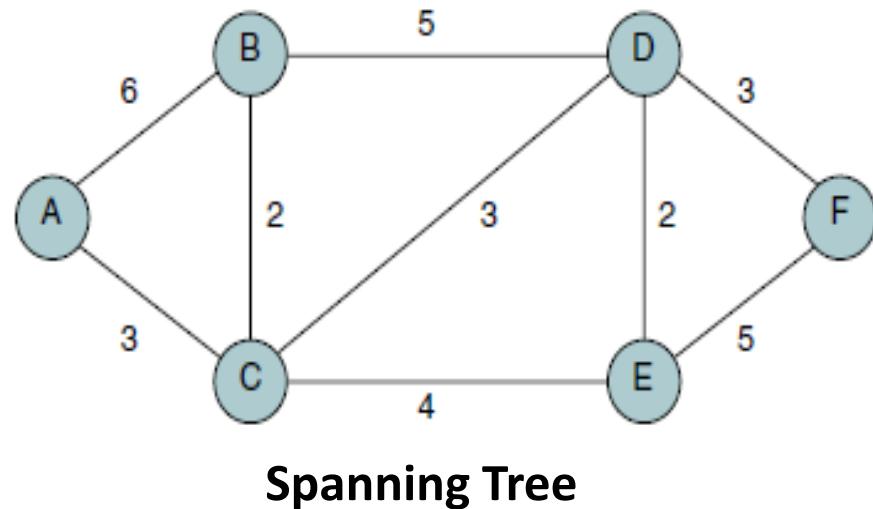


Rules:

1. Start with any vertex.
2. From all of the vertices currently in the tree, select the edge with the minimal value to a vertex not currently in the tree and insert it into the tree.

To develop the algorithm for the minimum spanning tree, a storage structure, is needed. Adjacency list is used because it is the most flexible.

Minimum Spanning Tree: Creation



Minimum Spanning-Tree Algorithms.

Kruskal's Algorithm

Prim's Algorithm

Both are greedy algorithms

- Growing a minimum cost spanning tree
- Assume there is a connected, undirected graph $G = (V, E)$ with a weight function $w: E \rightarrow \mathbb{R}$, and a *minimum spanning tree for G has to be found.*
- The two algorithms use a greedy approach to the problem, although they differ in how they apply this approach.
- This greedy strategy is captured by the following generic method, **which grows the minimum spanning tree one edge at a time.**
- **The generic method manages a set of edges A, maintaining the following loop invariant:**

Prior to each iteration, A is a subset of some minimum spanning tree.

- At each step, an edge is determined (u, v) such that it can be added to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.
- Such an edge is a **safe edge for A**, since it can be added safely to A while maintaining the invariant.

- Growing a minimum spanning tree
- Input: Let $G = (V, E)$ be a connected undirected graph with a weight
- Purpose: Grow a minimum spanning tree, one edge at a time

```
GENERIC-MST( $G, w$ )
```

- 1 $A = \emptyset$
- 2 **while** A does not form a spanning tree
- 3 find an edge (u, v) that is safe for A
- 4 $A = A \cup \{(u, v)\}$
- 5 **return** A

Note:

The edge (u, v) that is added to A is called safe, because $A \cup \{(u, v)\}$ is also a subset of minimum spanning tree.

- **Kruskal's Algorithm**
- Algorithm finds the minimum cost spanning tree and it uses the greedy approach.
- Special case of the generic minimum-spanning-tree method
- Algorithm treats the graph as a forest and every node it has as an individual tree.
- **Forest is a collection of disjoint trees.** Eg:- removing the root from a binary tree, we get a forest of two trees.
- A tree connects to another if and only if, it has the least cost among all available options and does not violate MST properties
- Set A is a forest whose vertices are all those of the given graph G.
- *Algorithm finds a safe edge to add to the growing forest*
- This **edge is always a least-weight edge** in the graph that connects two distinct components (two different trees in the forest)
- Kruskal's algorithm is a greedy algorithm because at each step it adds to the forest an edge of least possible weight

Simple Steps for implementing Kruskal's algorithm.

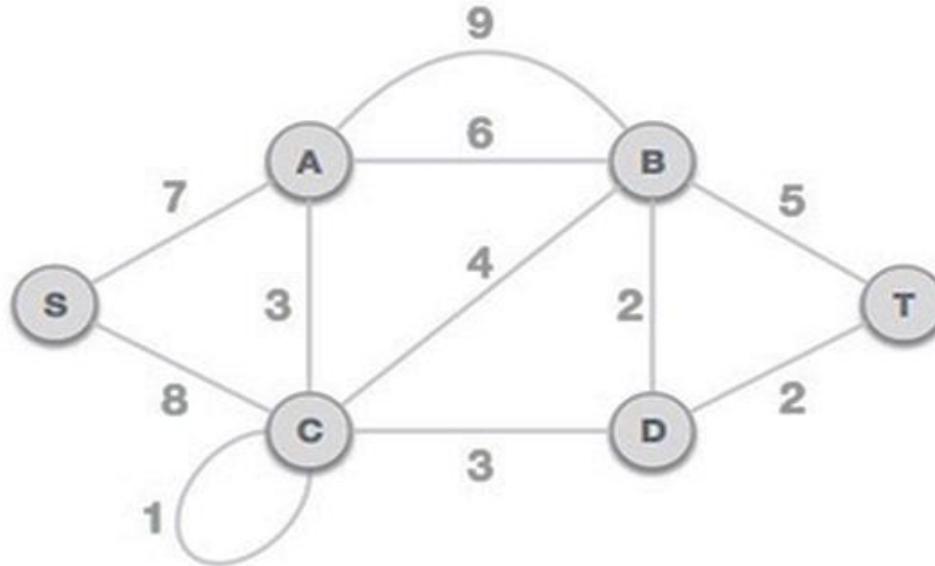
Sort all the edges from low weight to high

Take the edge with the lowest weight and add it to the spanning tree.
If adding the edge creates a cycle, then reject this edge.

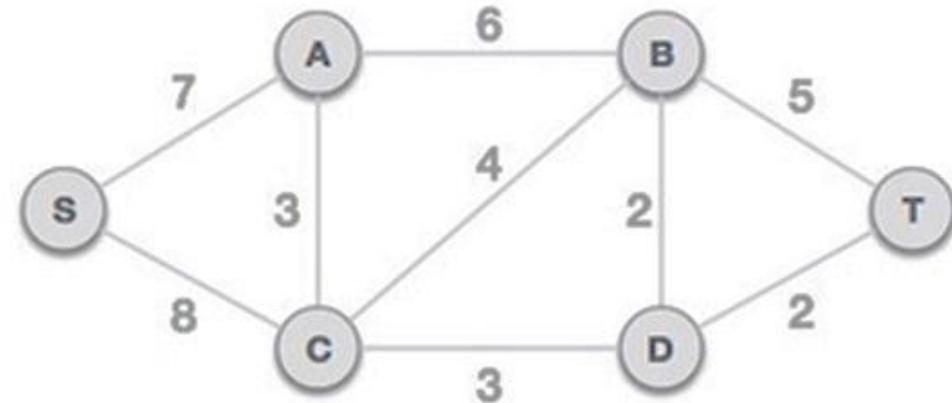
Keep adding edges until we reach all vertices.

Kruskal's algorithm example

Consider the graph G.



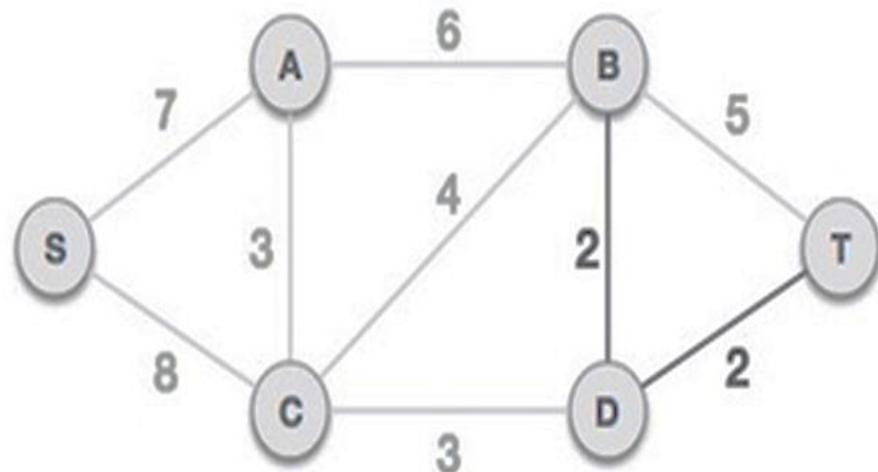
Step 1: Remove all loops and Parallel Edges



Step 3 - Add the edge which has the least weightage

Step 2: Arrange all edges in increasing order of weights.

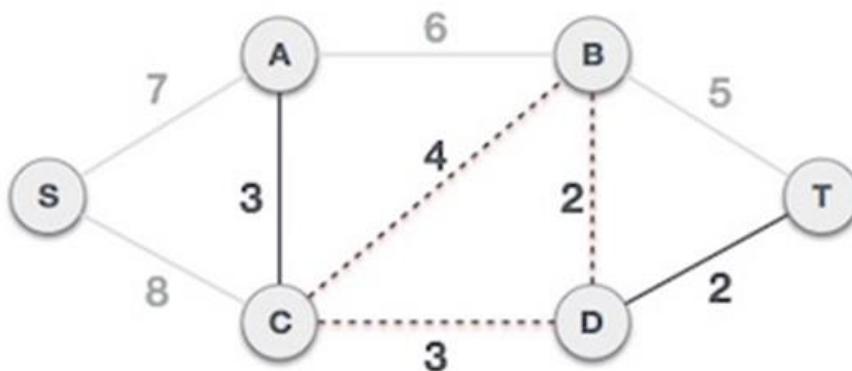
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8



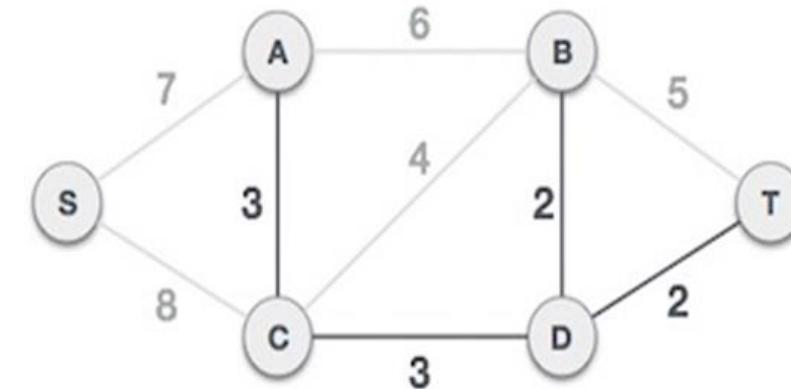
When edges are selected, edges that create cycles have to be avoided. If there is a cycle present, ignore the edge.

Kruskal's algorithm example

Step 4: Next cost is 3, and associated edges are A,C and C,D

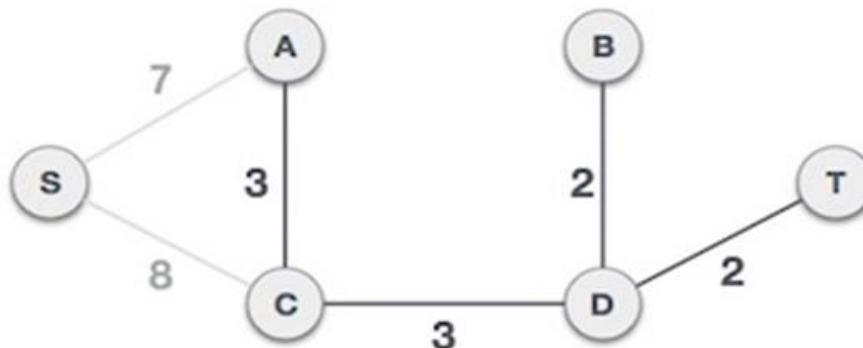


Step 5: Next cost is 4, and it will create a circuit in the graph.

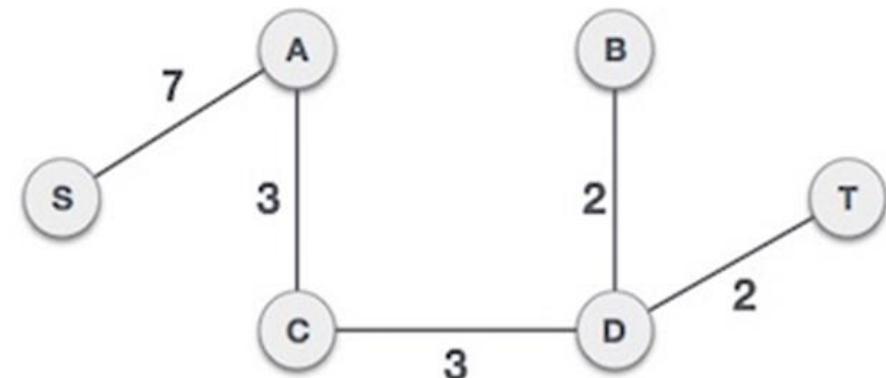


If a graph has "n" vertices, , then the MST must contain n vertices and (n-1) edges

Step 6: Between the two least cost edges 7 and 8, add the edge with cost 7.



Step 7: All vertices of the graph have included and a minimum cost spanning tree created. Therefore, **Cost of MST=17**



PRIM's ALGORITHM.

PRIM'S ALGORITHM.

- To ***find minimum cost spanning tree*** (as Kruskal's algorithm) the greedy approach is used.
- Prim's algorithm, in contrast with Kruskal's algorithm, treats the **nodes(vertices)** as a **single tree** and keeps on ***adding new nodes*** to the spanning tree from the given graph.
- Rather than build a subgraph one edge at a time, Prim's algorithm ***builds a tree one vertex at a time.***
- That is, in Prim's Algorithm a spanning tree grows from a arbitrary starting position(root) and a vertex is added to the growing spanning tree in Prim's (Unlike an edge in Kruskal's).

PRIM'S ALGORITHM.

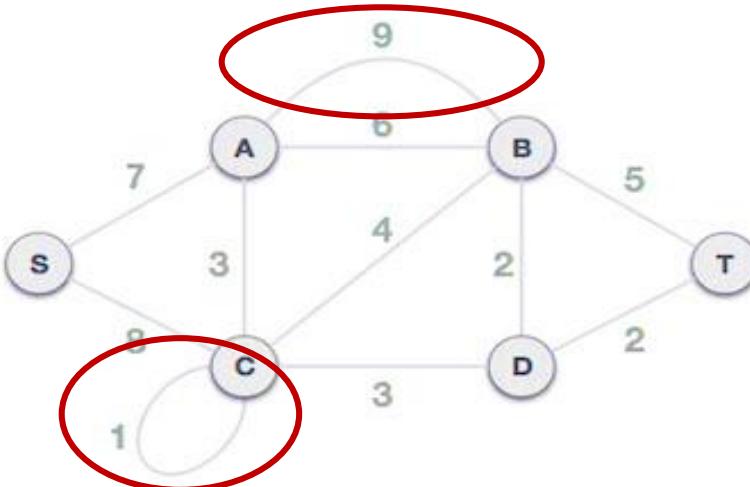
- Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method.
- Prim's algorithm has the property that the edges in the set A always form a single tree.
- The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V .
- Each step adds to the tree A, a light edge that connects A to an isolated vertex: one on which no edge of A is incident.
- This rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

~~DDIM'A'S ALGORITHM~~

1. Keep a track of all the vertices that have been visited and added to the spanning tree.
2. Initially the spanning tree is empty.
3. Choose a random vertex, and add it to the spanning tree. This is **the root node**.
4. Add a new vertex, say **vx**, such that **vx** is not in the already built spanning tree.
 - **vx** is connected to the built spanning tree using minimum weight edge. (So, remember, that **vx** can be adjacent to any of the nodes that have already been added in the spanning tree).
 - Make sure adding vertex **vx** to the spanning tree should not form cycles.
5. Repeat the Step 4, till all the vertices of the graph are added to the spanning tree.
6. Print the total cost of the spanning tree.

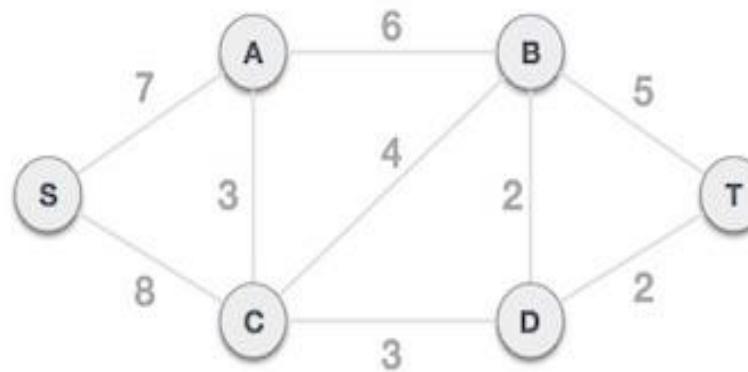
PRIM'S ALGORITHM example 1.

Consider the graph G:-



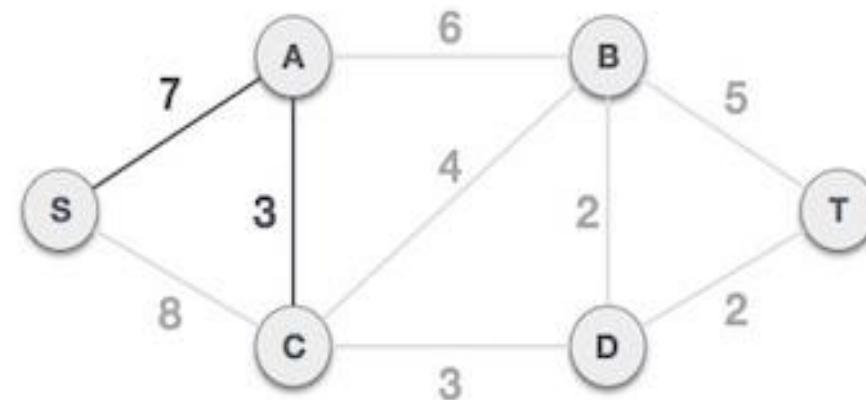
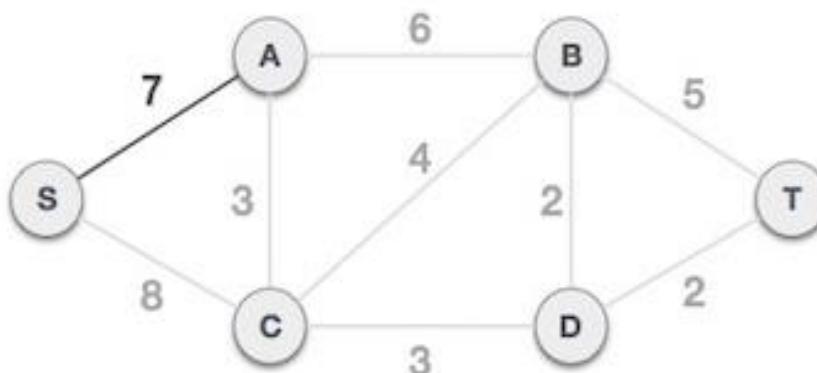
Step 3: Check outgoing edges and select the one with less cost. Choose the edge S,A. Why?

Step 1: Remove all loops and Parallel Edges



Step 2: Choose any arbitrary node as root node
Here choose S node as the root node

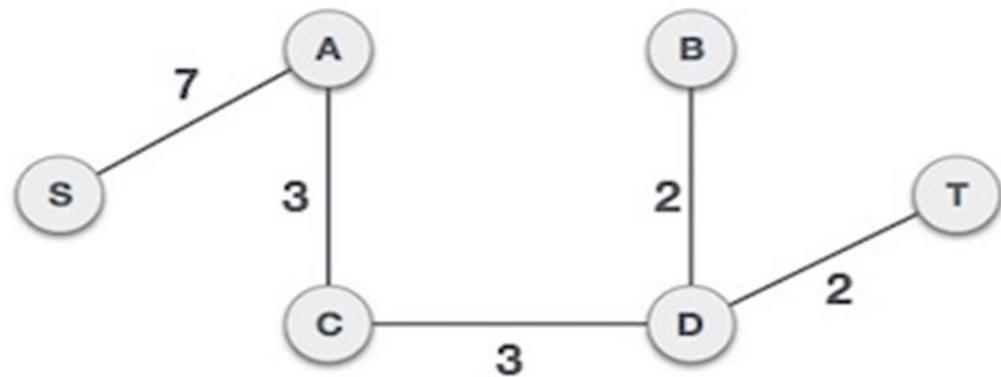
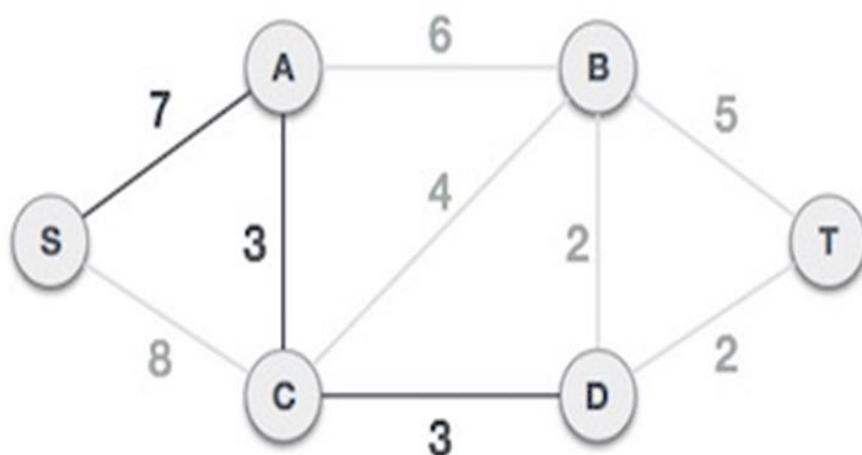
Step 4: The tree S-7-A is treated as one node. Check for all outgoing edges. Select the one with lowest cost



PRIM'S ALGORITHM example 1.

Step 5: S-7-A-3-C tree is formed, treat it as a node and check all the edges again, choose only the least cost edge.

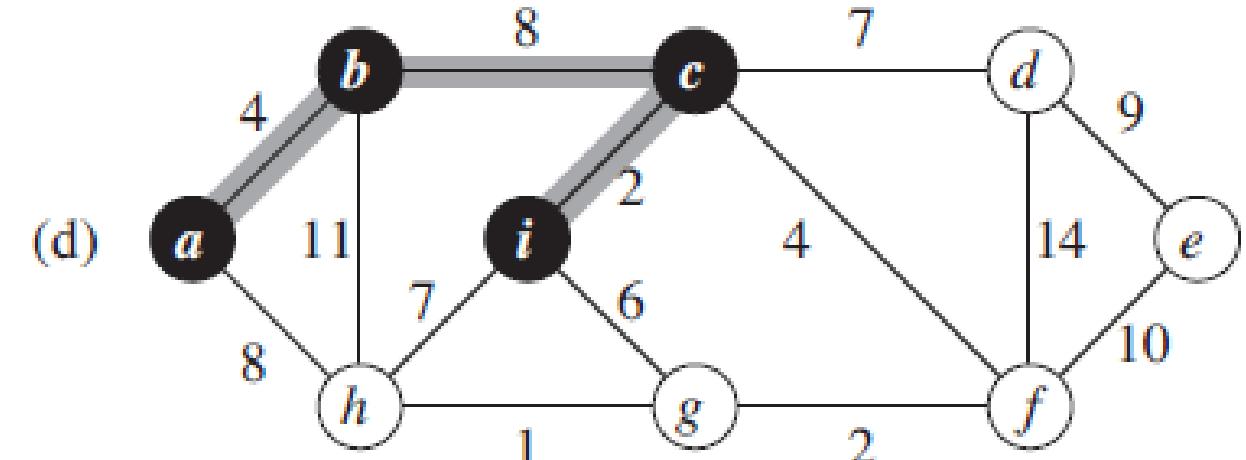
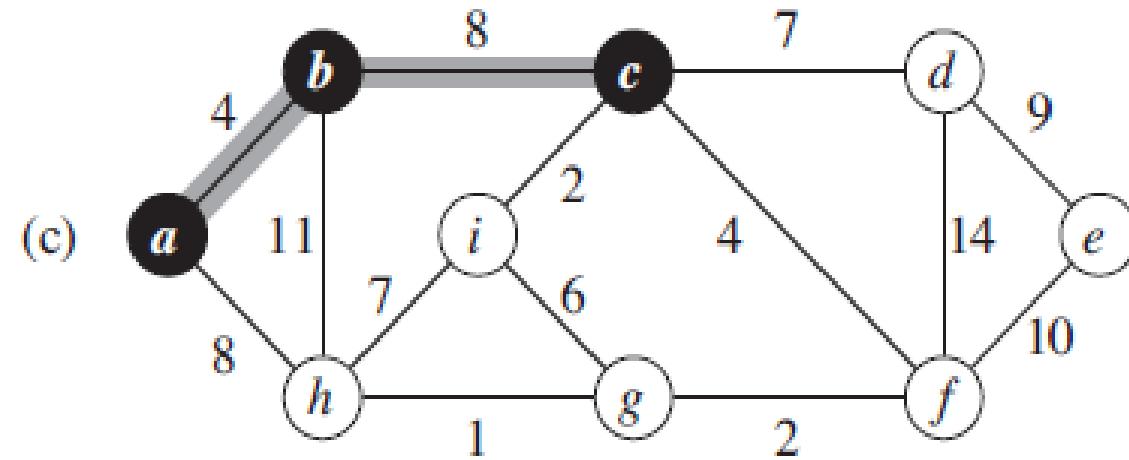
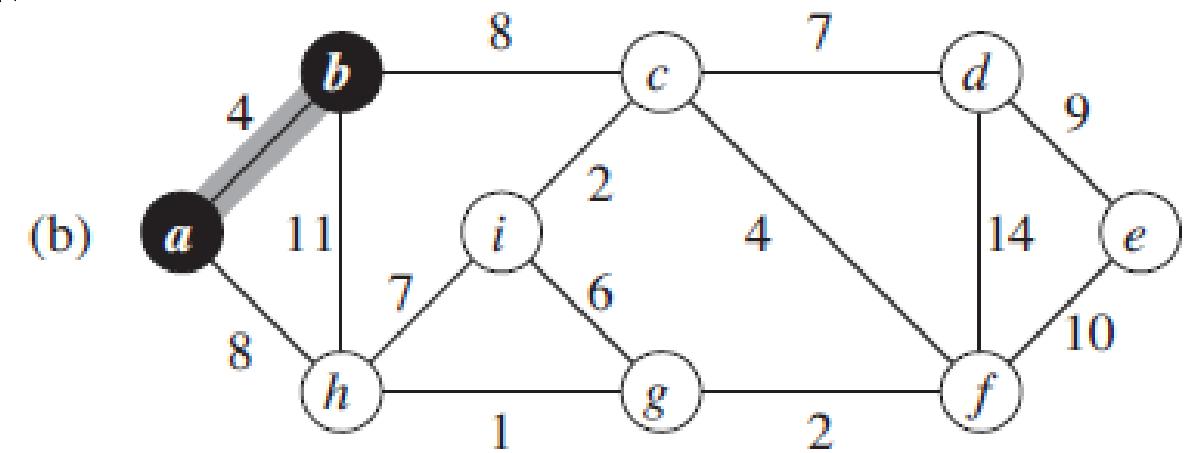
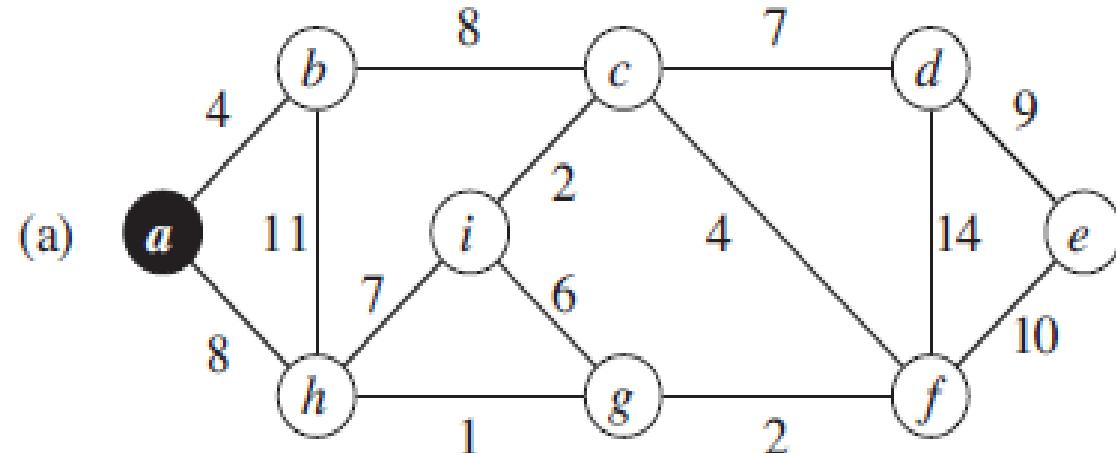
Step 6: All vertices of the graph have been included in the MST, which means the total edge weight or cost is minimum.



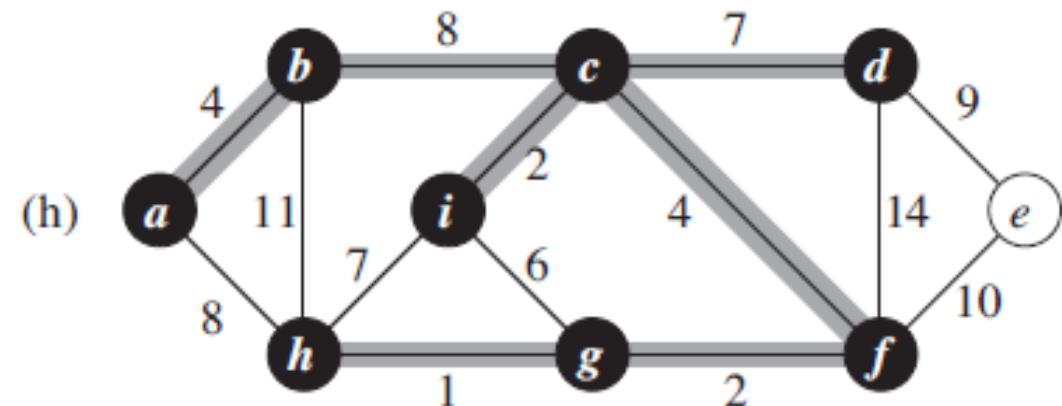
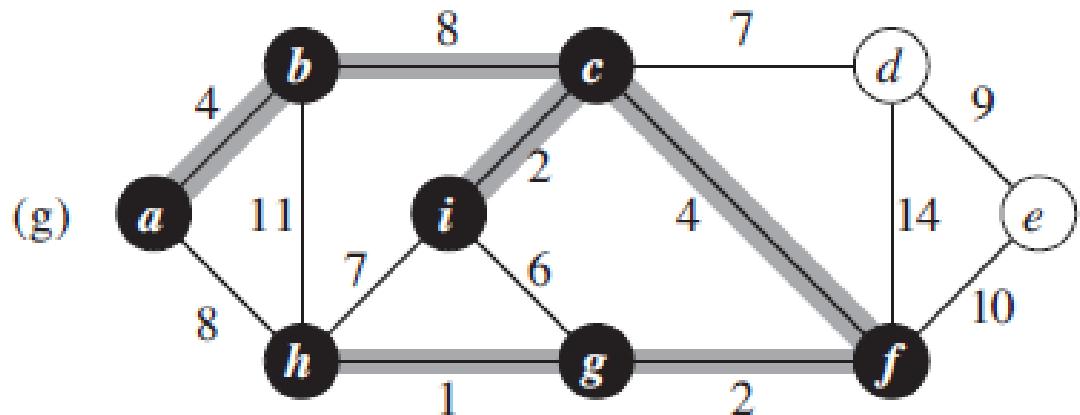
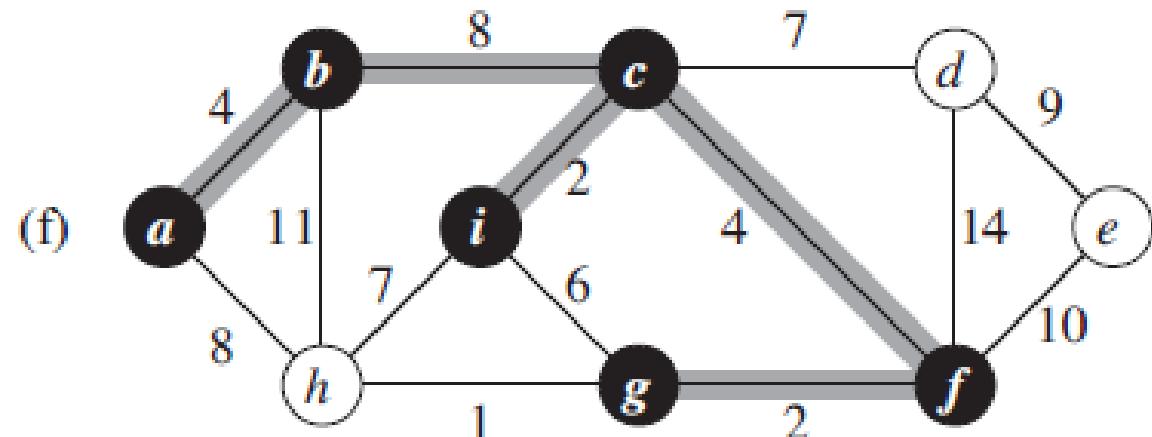
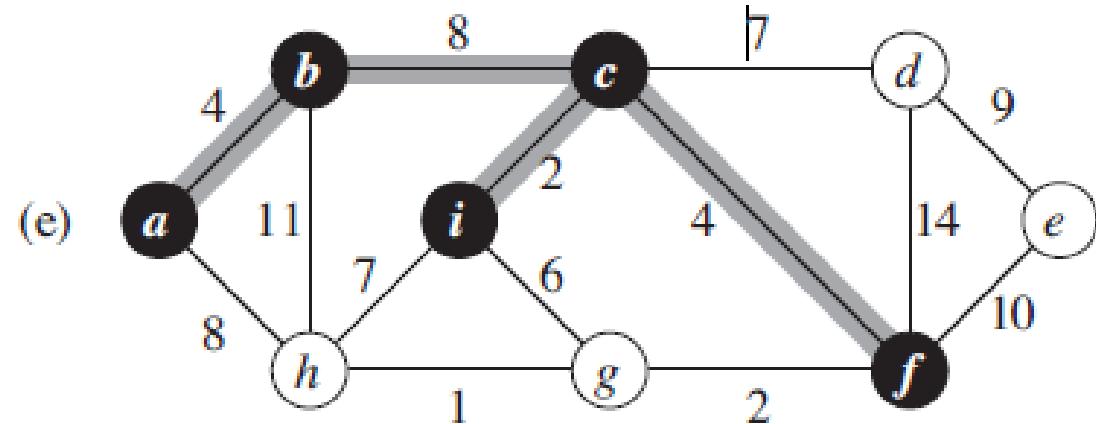
Cost of min. spanning tree

$$7+3+3+2+2 = 17$$

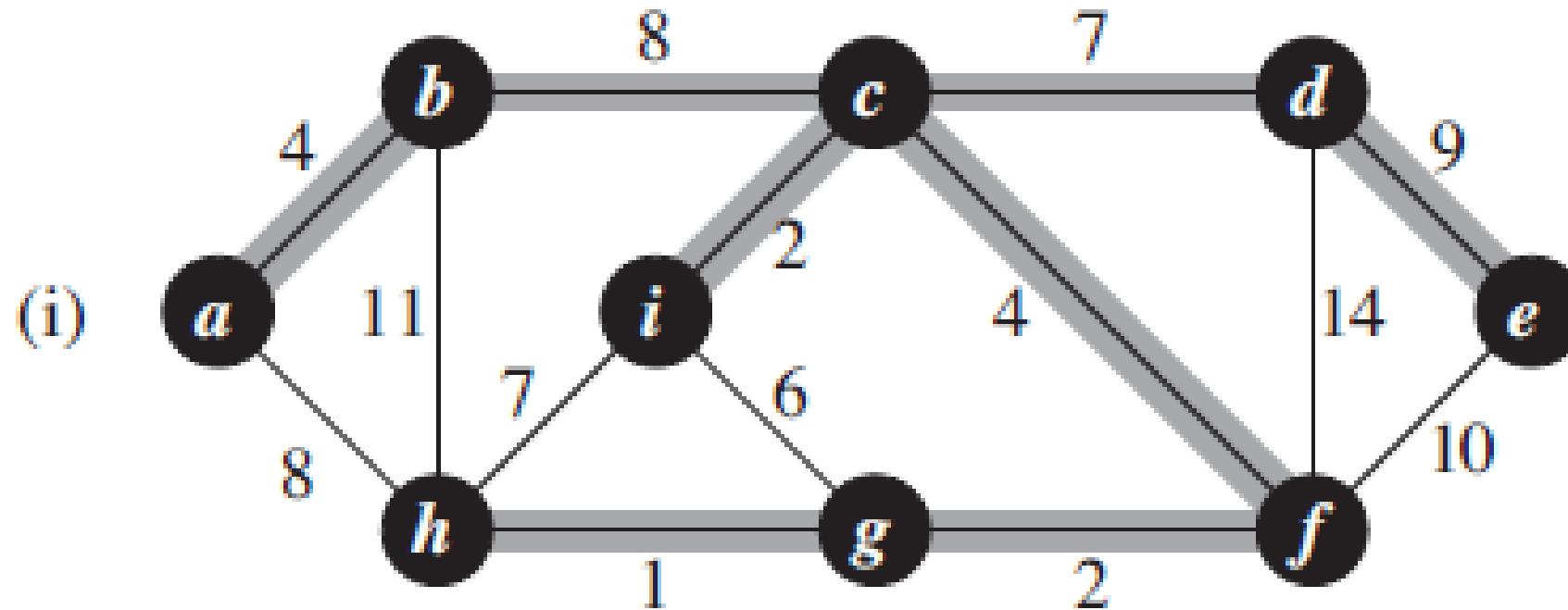
PRIM'S ALGORITHM example 2-1



PRIM'S ALGORITHM example 2-2



PRIM'S ALGORITHM example 2-3

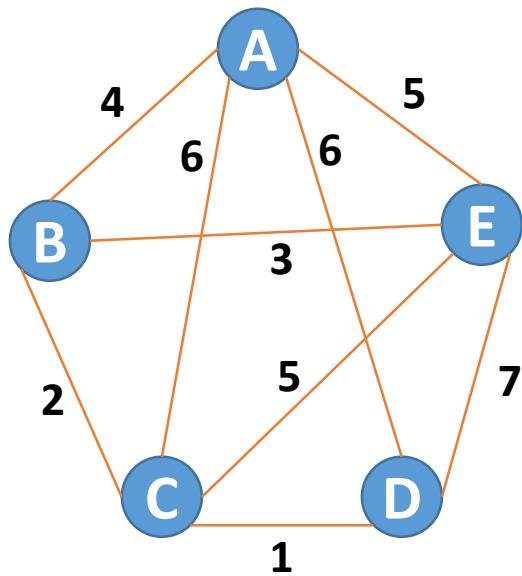


The above segments showed the execution of Prim's algorithm on the graph. The root vertex is **a**. Shaded edges are in the tree being grown, and black vertices are in the tree.

Cost of min. spanning tree

$$4+8+2+4+2+1+7+9 = 37$$

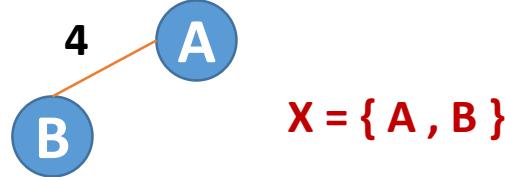
Prims Algorithm



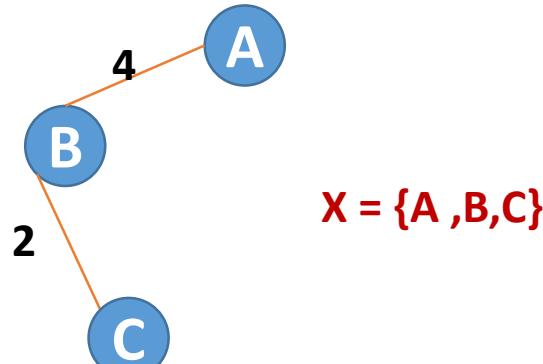
Let X be the set of nodes explored, initially $X = \{ A \}$



Step 1: Taking minimum Weight edge of all Adjacent edges of $X=\{A\}$

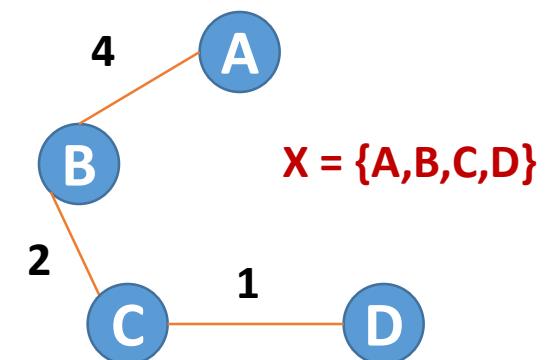


Step 2: Taking minimum weight edge of all Adjacent edges of $X = \{ A , B \}$

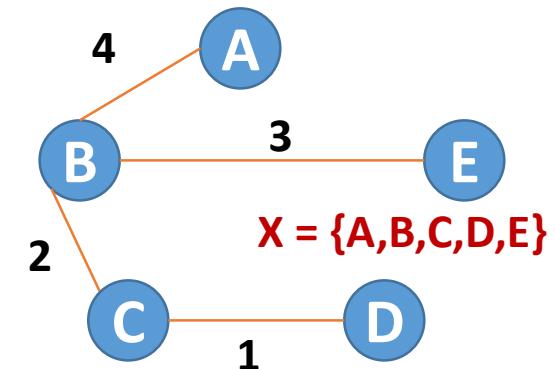


We obtained minimum spanning tree of cost:
 $4 + 2 + 1 + 3 = 10$

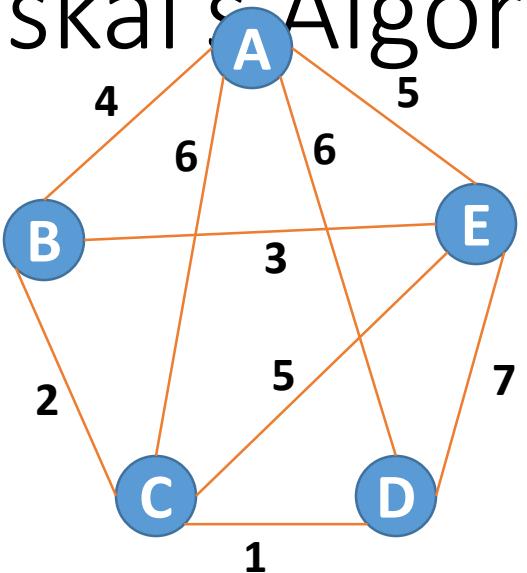
Step 3: Taking minimum weight edge of all Adjacent edges of $X = \{ A , B , C \}$



Step 4: Taking minimum weight edge of all Adjacent edges of $X = \{ A , B , C , D \}$



Kruskal's Algorithm

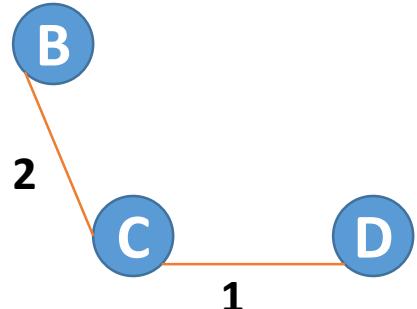


Step 1: Taking min edge (C,D)

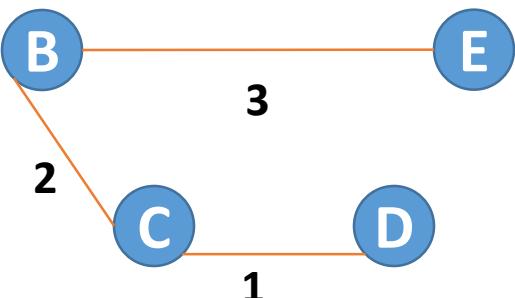


C, D	B, C	B, E	A, B	E, C	A, E	C, A	A, D	E, D
1	2	3	4	5	5	6	6	7

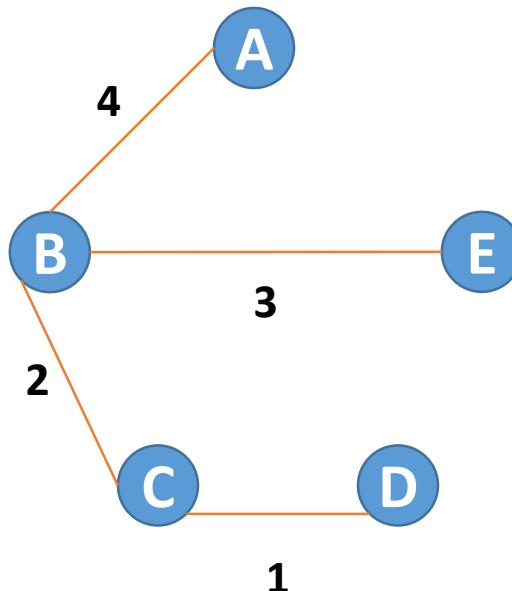
Step 2: Taking next min edge (B,C)



Step 3: Taking next min edge (B,E)

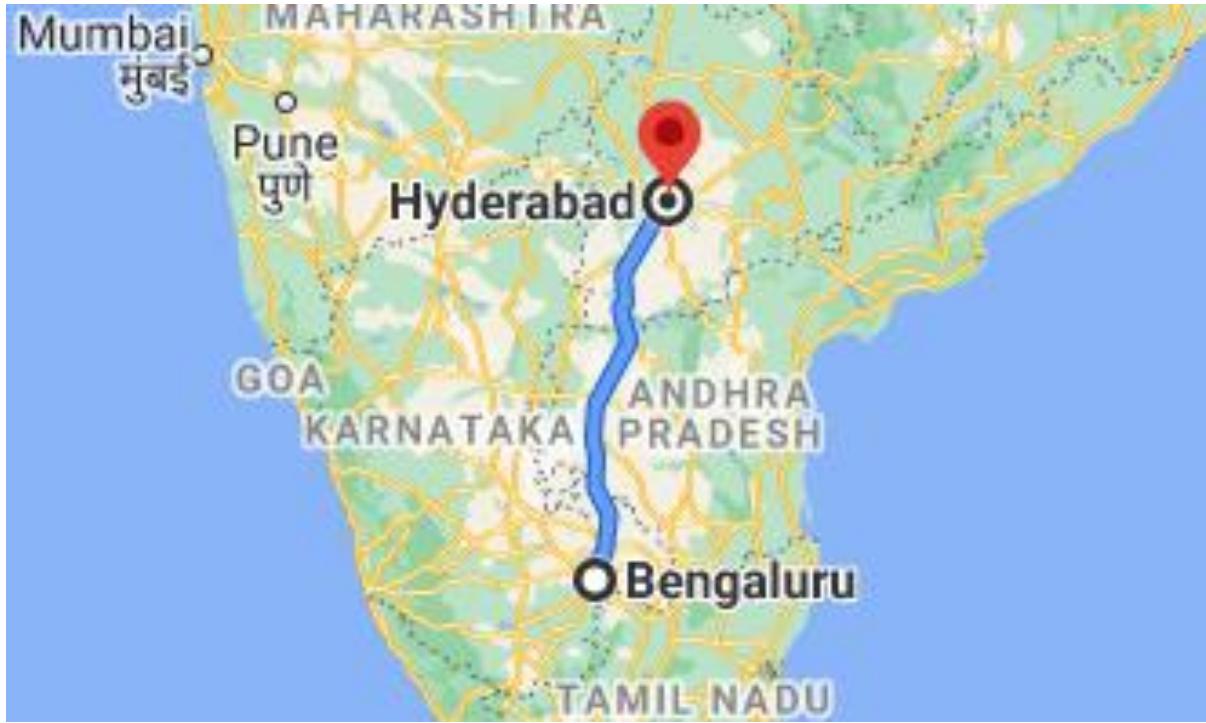


Step 4: Taking next min edge (A,B)



so we obtained minimum spanning tree of cost:
 $4 + 2 + 1 + 3 = 10$

THANK YOU



Shortest Path Problems.

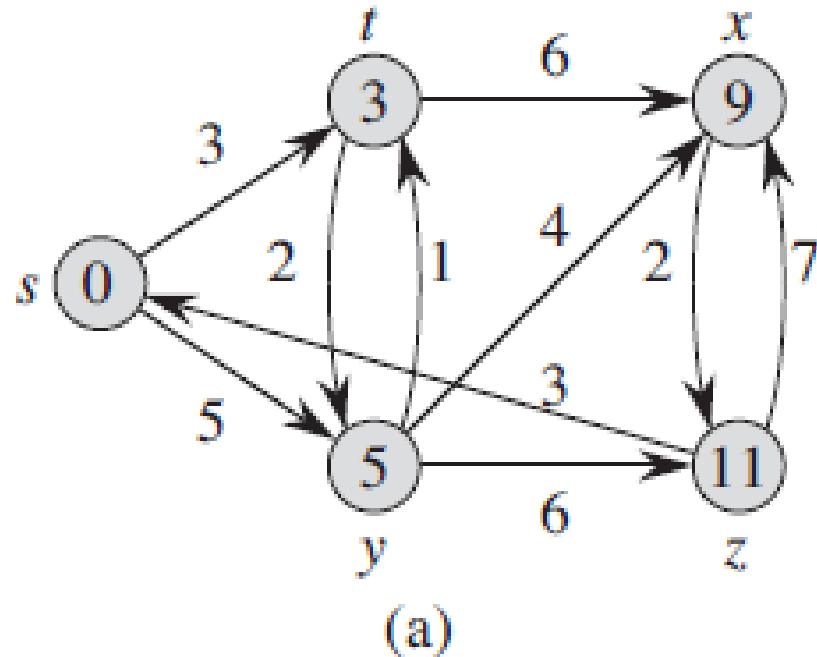
- **Single Source Shortest Path Problems.** Given a graph $G = (V, E)$, find the shortest path from a given source $s \in V$ to each vertex $v \in V$
- Variants of the problem:

Single-destination shortest-path problems: Find a shortest path to a given destination vertex t from each vertex v

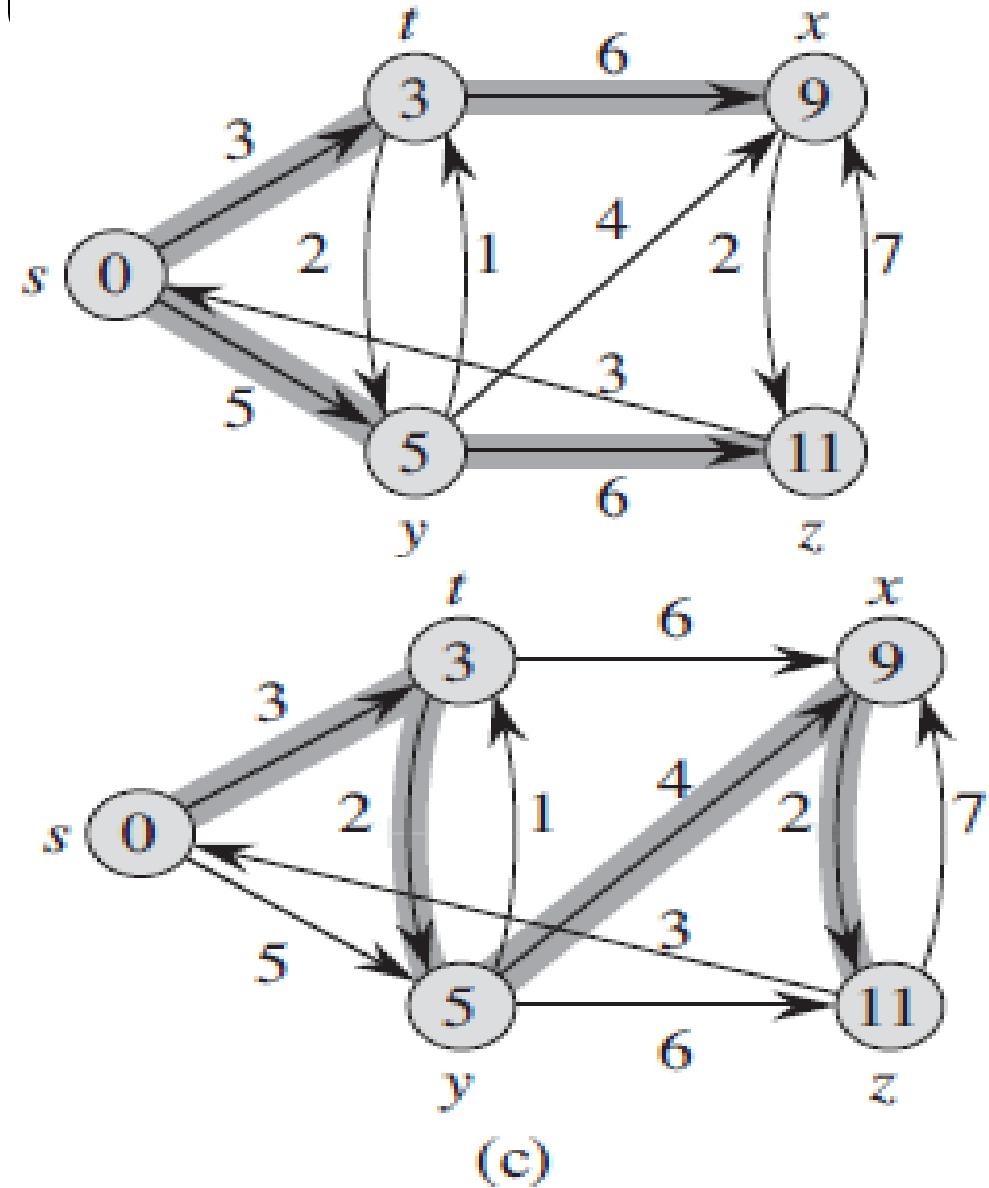
Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v

All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v

Single Source Shortest Path Problems



- a) A weighted, directed graph with shortest-path weights from source s.
- b) The shaded edges form a shortest-paths tree rooted at the source s.
- c) Another shortest-paths tree with the same root.

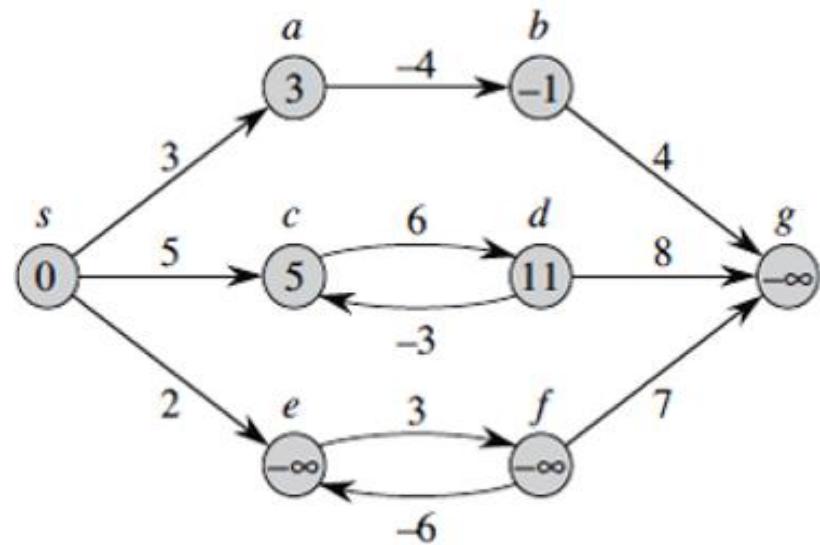


Single Source Shortest Path Problems.

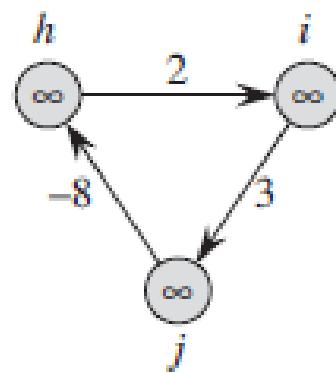
Negative-weight cycles:

- Some instances of the single-source shortest-paths problem may include edges whose weights are negative.
- If graph $G=(V, E)$ contains no negative weight cycles reachable from the source s , then for all $v \in V$, shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
- If the graph contains a negative-weight cycle reachable from s , however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path, a path with lower weight can be found by following the proposed “shortest” path and then traversing the negative-weight cycle.
- If there is a negative weight cycle on some path from s to v , we define $\delta(s, v) = -\infty$

Single-Source Shortest-Path Problems



- Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s , they have shortest-path weights of $-\infty$.
- Because vertex g is reachable from a vertex whose shortest path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$.
- Vertices such as h , i , and j are not reachable from s , and so their shortest-path weights are ∞ , even though they lie on a negative-weight cycle.



Note: Some shortest-paths algorithms, such as **Dijkstra's algorithm**, assume that all **edge weights** in the input graph are **nonnegative**, as in the road-map example.

Others, such as the **Bellman-Ford algorithm**, allow **negative-weight** edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source.

In case of negative-weight cycles, shortest-path weights are **not well defined**

Shortest Path Algorithm

- Let $\mathbf{G = (V,E)}$ be a simple diagraph with n vertices
- The problem is to find out shortest distance from a vertex to all other vertices of a graph.
- **Dijkstra Algorithm** – it is also called Single Source Shortest Path Algorithm

Single Source Shortest Path Problem Solutions.

- The algorithms for finding the single source shortest path use the technique of relaxation:
 - **Dijkstra Algorithm**
 - **Bellman Ford Algorithm**
- For each vertex, $v \in V$, an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v . **$v.d$ is called a shortest-path estimate.**

$$5 + 2 < 6 \times$$

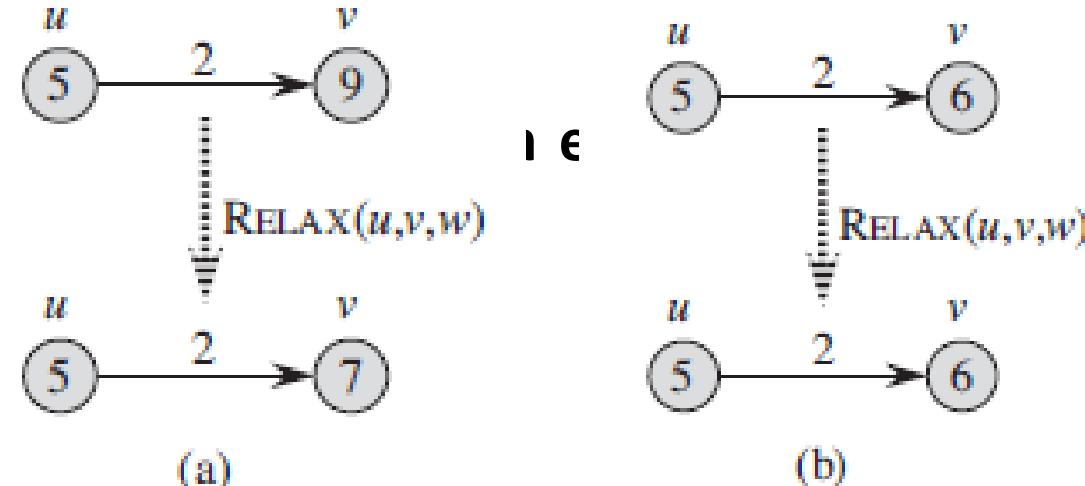
Relaxation of edges.

- Process of relaxing an edge (u, v) consists of testing whether **the shortest path can be improved** to v , found so far by going through u and, if so, updating edges.
- Relaxation step may decrease the $v.d$

- Algorithm **RELAX** (u, v, w)

if $u.d + w(u, v) < v.d$

$v.d = u.d + w(u, v)$



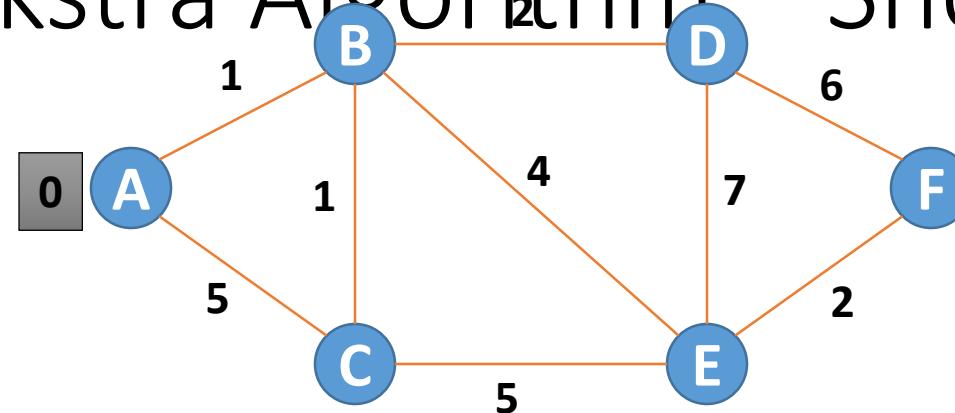
Relaxing an edge (u,v) with weight $w(u,v)=2$.

The shortest-path estimate of each vertex appears within the vertex.

(a) Because $v.d > u.d + w(u,v)$ prior to relaxation, the value of $v.d$ decreases.

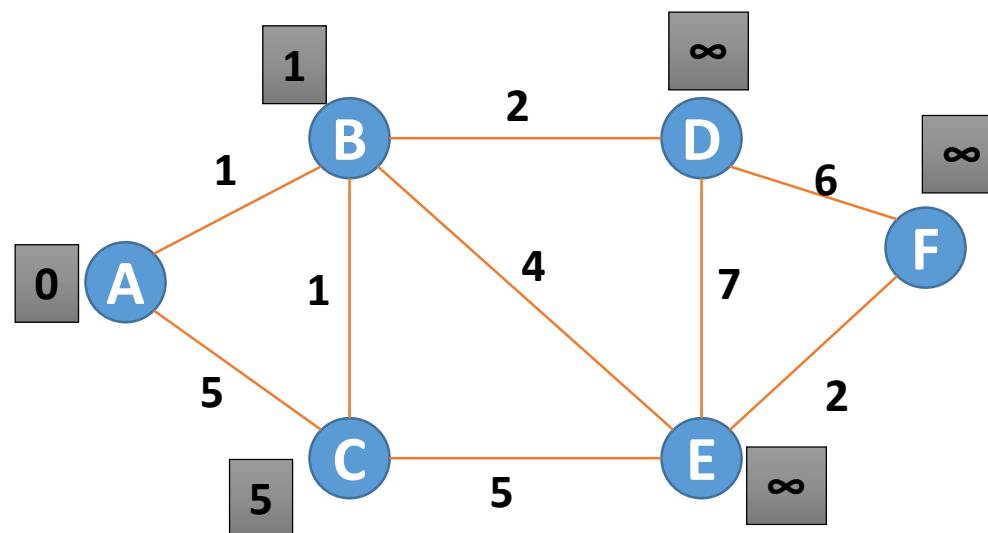
(b) Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation step leaves $v.d$ unchanged.

Dijkstra Algorithm – Shortest Path



	A	B	C	D	E	F
Distance	0	∞	∞	∞	∞	∞
Visited	0	0	0	0	0	0

1st Iteration: Select Vertex A with minimum distance



	A	B	C	D	E	F
Distance	0	1	5	∞	∞	∞
Visited	1	0	0	0	0	0

Dijkstra Algorithm – Shortest Path

2nd Iteration: Select Vertex B with minimum distance

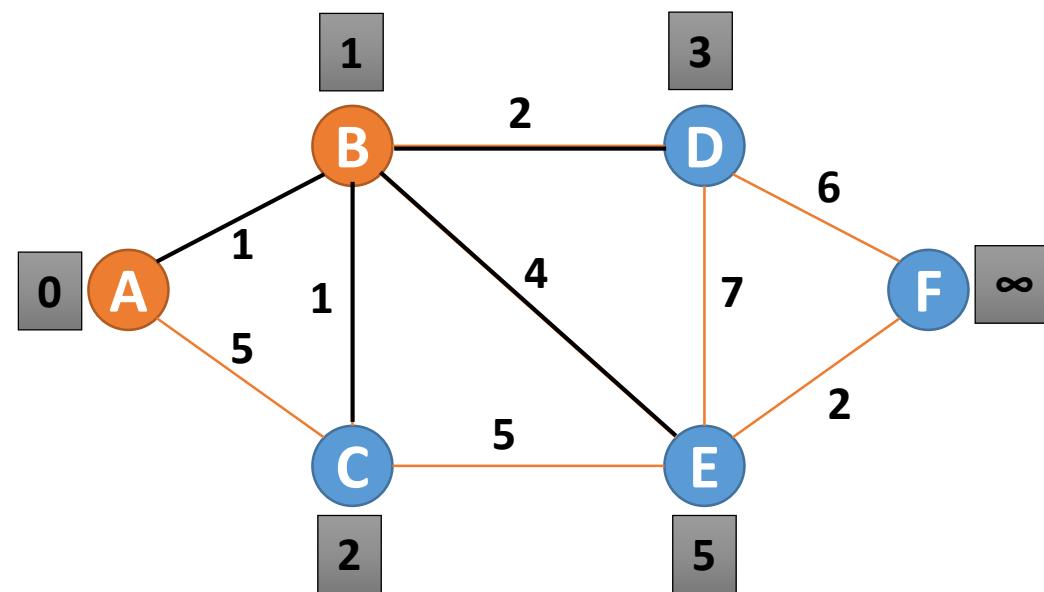
Cost of going to C via B = $\text{dist}[B] + \text{cost}[B][C] = 1 + 1 = 2$

Cost of going to D via B = $\text{dist}[B] + \text{cost}[B][D] = 1 + 2 = 3$

Cost of going to E via B = $\text{dist}[B] + \text{cost}[B][E] = 1 + 4 = 5$

Cost of going to F via B = $\text{dist}[B] + \text{cost}[B][F] = 1 + \infty = \infty$

	A	B	C	D	E	F
Distance	0	1	5	∞	∞	∞
Visited	1	0	0	0	0	0



	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	0	0	0	0

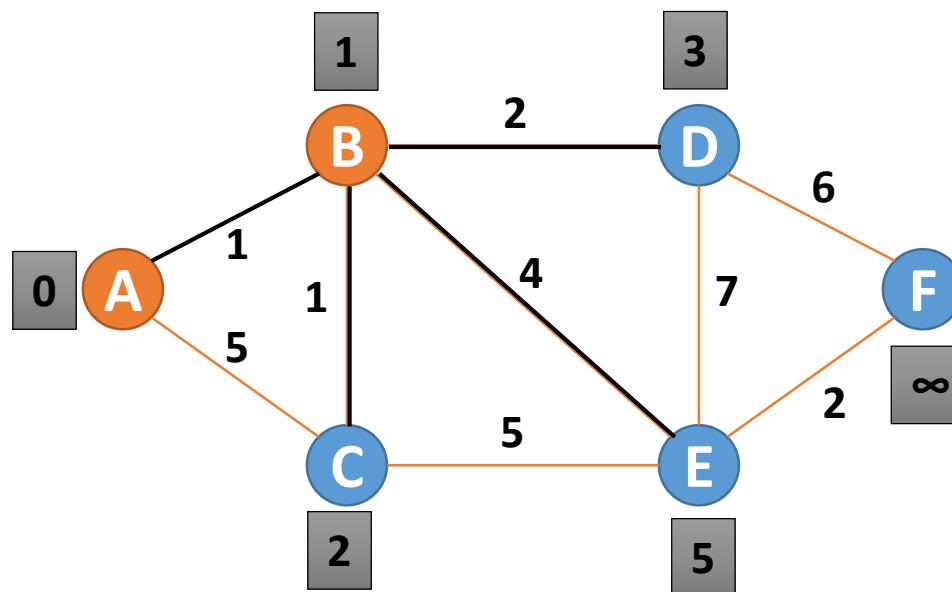
Dijkstra Algorithm – Shortest Path

3rd Iteration: Select Vertex C via B with minimum distance

Cost of going to D via C = $\text{dist}[C] + \text{cost}[C][D] = 2 + \infty = \infty$

Cost of going to E via C = $\text{dist}[C] + \text{cost}[C][E] = 2 + 5 = 7$

Cost of going to F via C = $\text{dist}[C] + \text{cost}[C][F] = 2 + \infty = \infty$



	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	0	0	0	0

	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	1	0	0	0

$$\begin{aligned} D - E - 3 + 7 &< 5 \times \\ D - F - 3 + 6 &< \infty \checkmark \end{aligned}$$

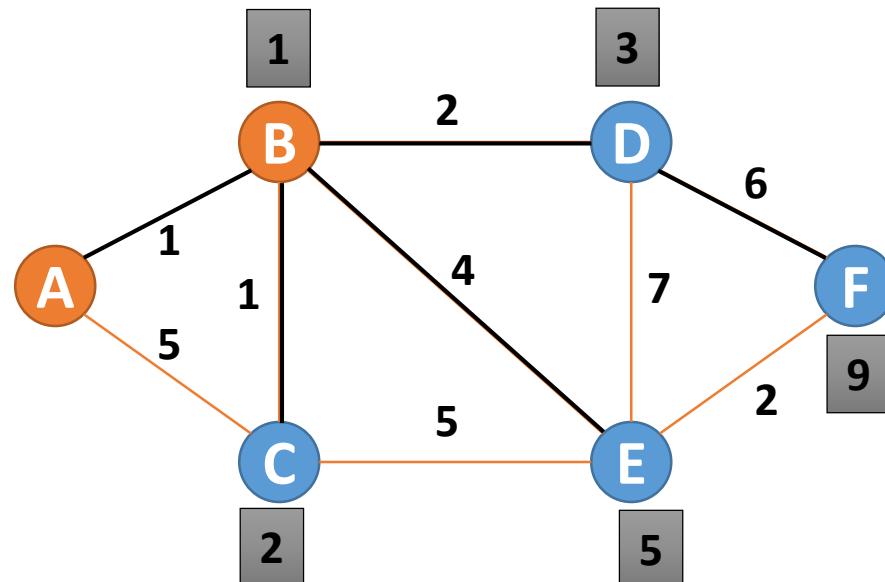
Dijkstra Algorithm – Shortest Path

4th Iteration: Select Vertex D via path A - B with minimum distance

Cost of going to E via D = $\text{dist}[D] + \text{cost}[D][E] = 3 + 7 = 10$

Cost of going to F via D = $\text{dist}[D] + \text{cost}[D][F] = 3 + 6 = 9$

	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	1	0	0	0

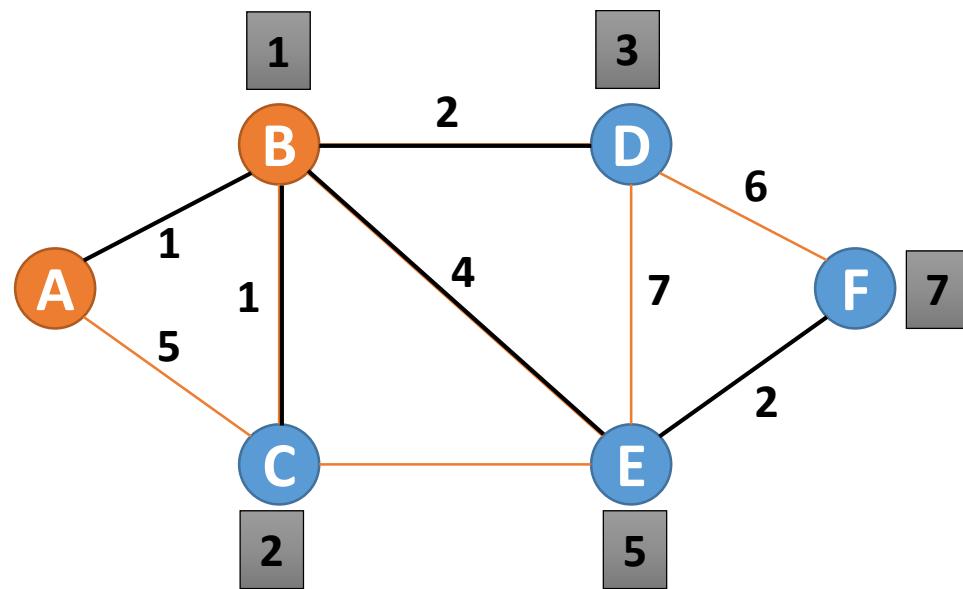


	A	B	C	D	E	F
Distance	0	1	2	3	5	9
Visited	1	1	1	1	0	0

Dijkstra Algorithm – Shortest Path

4th Iteration: Select Vertex E via path A – B – E with minimum distance

Cost of going to F via E = dist[E] + cost[E][F] = 5 + 2 = 7



	A	B	C	D	E	F
Distance	0	1	2	3	5	9
Visited	1	1	1	1	0	0

	A	B	C	D	E	F
Distance	0	1	2	3	5	7
Visited	1	1	1	1	1	0

Shortest Path from A to F is
 $A \rightarrow B \rightarrow E \rightarrow F = 7$

Shortest Path Tree.

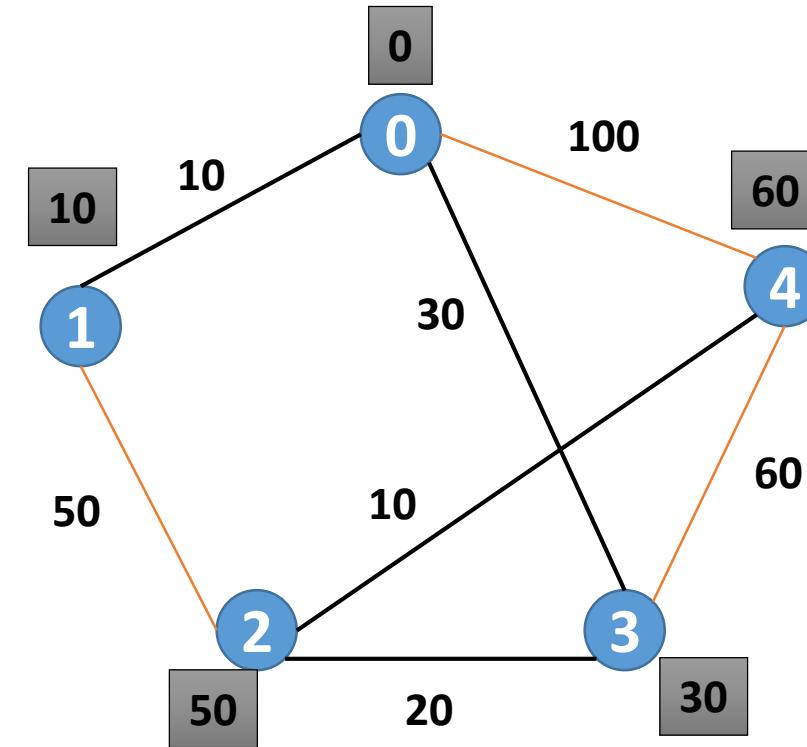
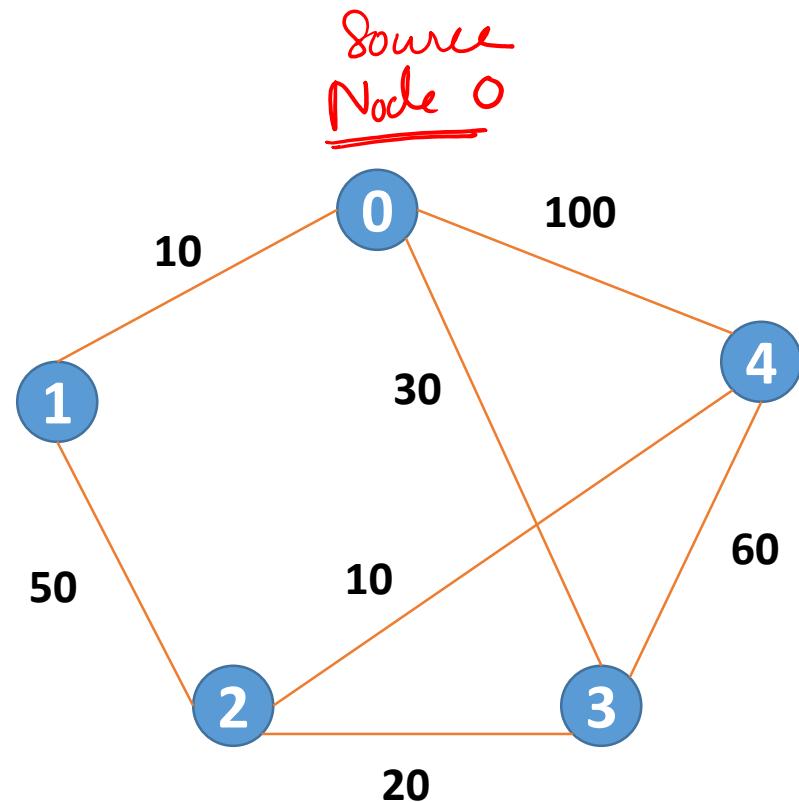
G

$E' V'$

- It is a tree with root at s and is ~~a subgraph~~ $G' = (V', E')$, where V' is a subset V and E' is a subset of E , such that
 - V' is the set of vertices reachable from s in G
 - G' forms a tree with root at s
 - For all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G

Shortest Path

Find out shortest path from node 0 to all other nodes using
Dijkstra Algorithm



Shortest Path Problems --Bellman-Ford Algorithm

- Solves the **single-source** shortest-paths problem in the general case in which the edge weights may be **negative**
- Inputs:
 - Weighted directed graph $G = (V, E)$ with source s
 - Weight function w
- Outputs:
 - Returns a **boolean value** indicating whether or not there is a **negative-weight cycle** (that is reachable from the source)
 - Cycle exists: no solution
 - No cycle: Produces the shortest paths and their weights
- Method:
 - **Relaxes edges** progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$

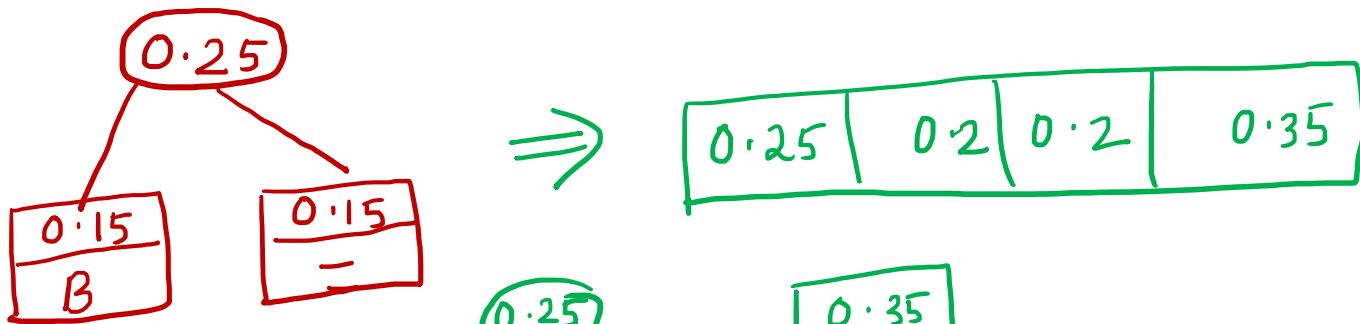
HUFFMAN CODING:- Application of trees

Example $\Rightarrow \{ A, B, C, D, - \}$

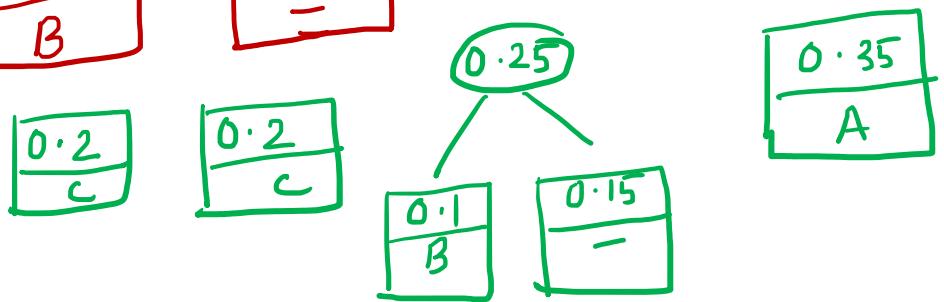
Step 1 :-

B	-	C	D	A
0.1	0.15	0.2	0.2	0.35

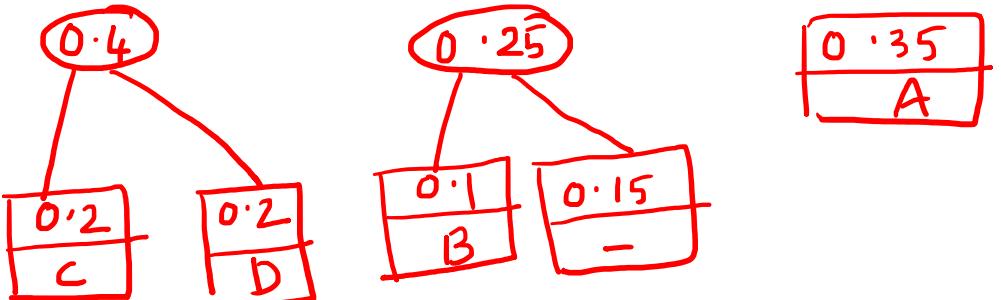
Step 2 :-



Step 3 :-

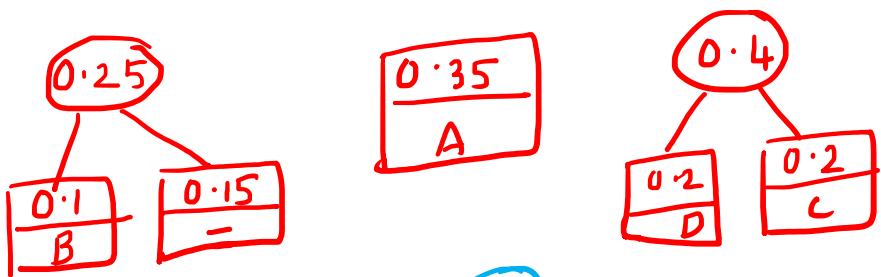


Step 4 :-

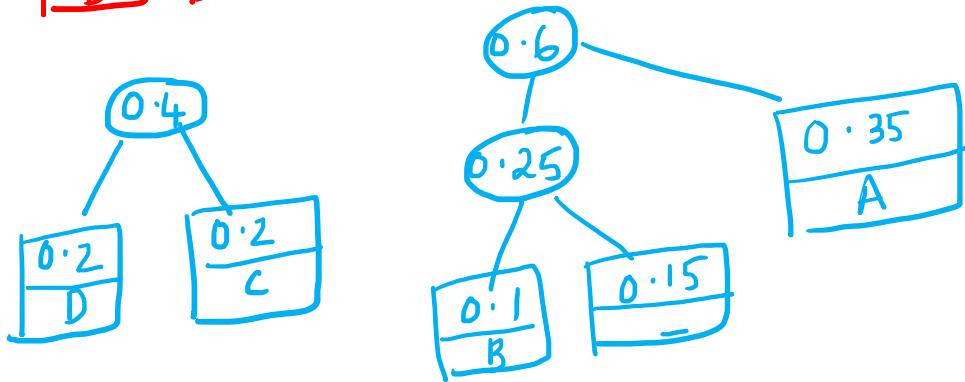


char	probability
A	0.35
B	0.1
C	0.2
D	0.2
-	0.15

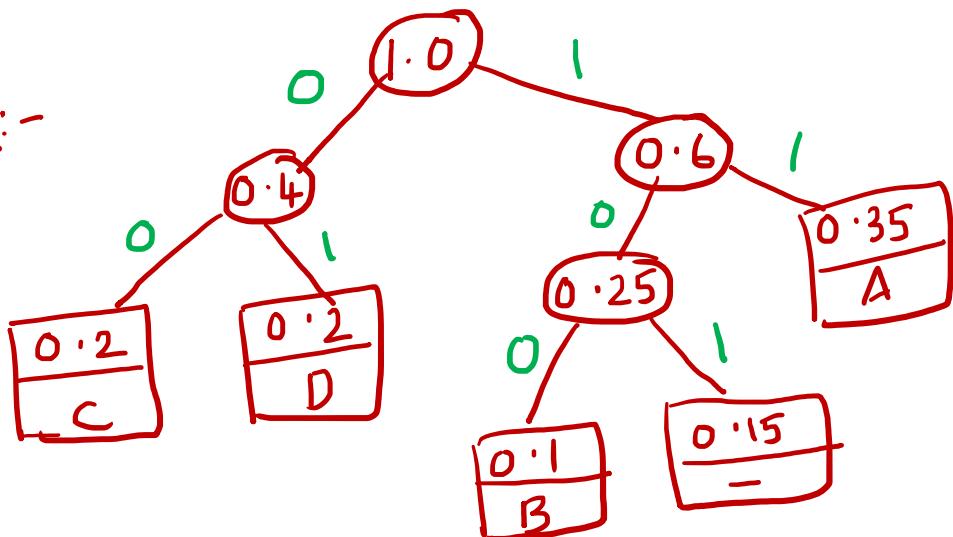
Step 5 :-



Step 6 :-



Step 7 :-



Huffman tree

NOTE

- ① Characters always present in the leaf node.
- ② Assign '0' to left tree edge and '1' to right tree edge.
- ③ Minimum values will be attached to left subtree always.

Huffman code

B	-	C	D	A
100	101	00	01	11

Note:- Character that occurs more frequently has less no.of bits in Huffman coding. Eg:- A,C,D has 2 bits.

Expected no. of bits per character =
probability/frequency * no. of bits

$$= 1 \times 3 + 0.15 \times 3 + 0.2 \times 2 + 0.2 \times 2 + 0.35 \times 2$$

$$= \underline{\underline{2.25}} \text{ bits per character.}$$

Total no. of bits transferred to send table to receiver =

$$= \text{no. of characters} * \text{ASCII length} + \text{length of codeword}$$

$$= 5 \times 8 + 12 = \underline{\underline{52}}$$

∴ Total bits to be transferred from sender to receiver

$$= 2.25 * 52$$

$$= \underline{\underline{117}} \text{ bits required to transfer from sender to receiver.}$$

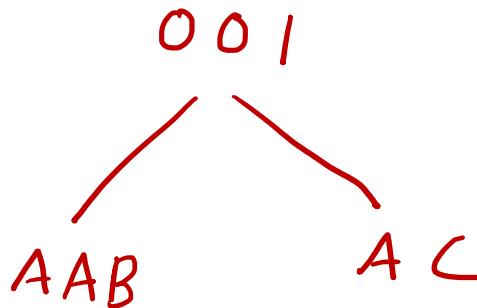
Condition to be satisfied - PREFIX CODE

eg :-

A - 0

B - 1

C - 01



i.e.; one code can be decoded in 2 ways

Reason :- 0 is prefix of another code.

Note :- This problem can be solved by huffman coding.

HUFFMAN CODE always follows PREFIX CODE.

Prefix code(prefix tree):- In a prefix code, no codeword is a prefix of a codeword of another character.



THANK YOU

HASHING.

Ideal Searching.

What is Hashed Search?

In an ideal search, one knows exactly where the data are and can go directly there.

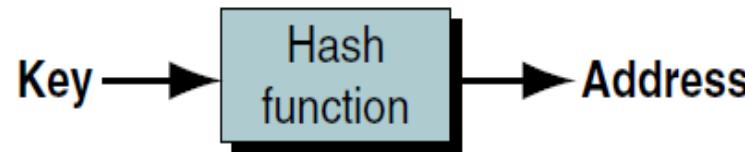
Goal of a hashed search: to find the data with only one test.

In a hashed search, the key (through an algorithmic function) determines the location of the data.

While searching an array, a **hashing algorithm** is used to transform the key into the index that contains the data we need to locate.

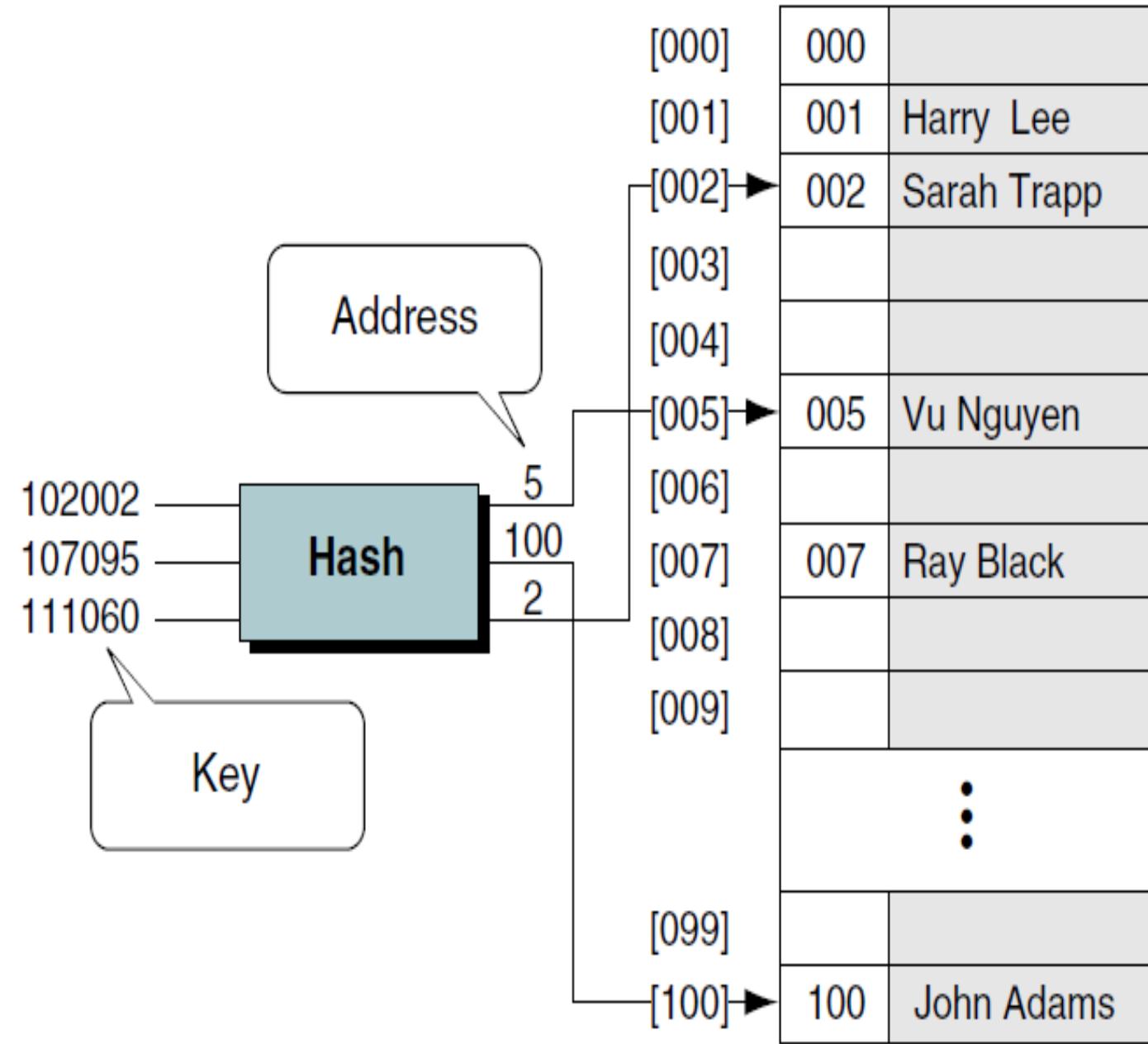
Hashing is also known as a **key-to-address transformation** in which the keys map to addresses in a list.

Hash Concept.



General representation of
Hashing concept

Hashing is a **key-to-address**
mapping process.



How to hash?

If we have an **array of 50 students** for a class in which the students are identified by the last four digits of their Aadhar numbers, there are **200 possible keys** for each element in the array ($10,000 / 50$).

Because there are many keys for each index location in the array, more than one student may hash to the same location in the array.

Set of keys that hash to the same location in a list are called **synonyms**.

If the actual data that inserted into a list contain two or more synonyms, then there are **collisions**.

How to hash?

A collision occurs when a hashing algorithm produces an address for an insertion key and that address is already occupied.

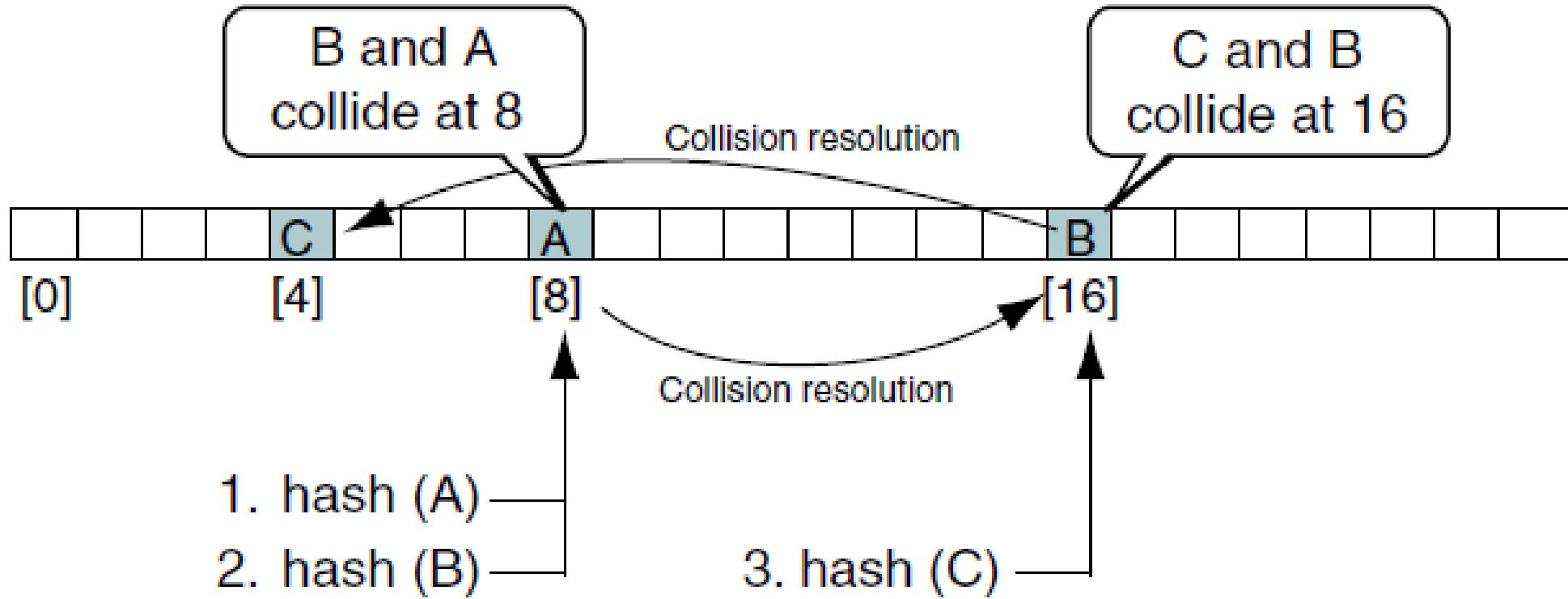
The address produced by the hashing algorithm is known as the **home address**.

The memory that contains all of the home addresses is known as the **prime area**.

When **two keys collide at a home address**, it must be resolved by placing one of the keys and its data in another location.

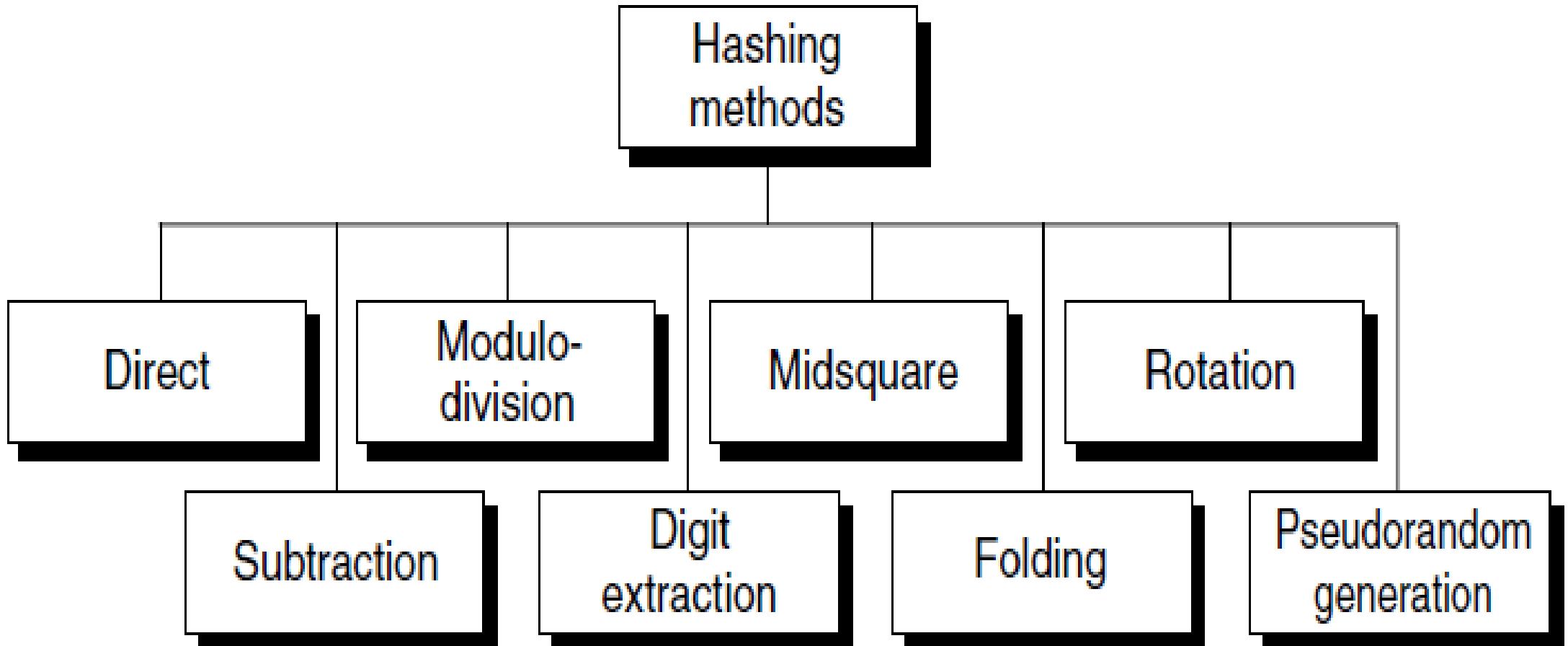
Each calculation of an address and test for success is known as a **probe**.

Collision Resolution Concept.



Collision Resolution Concept

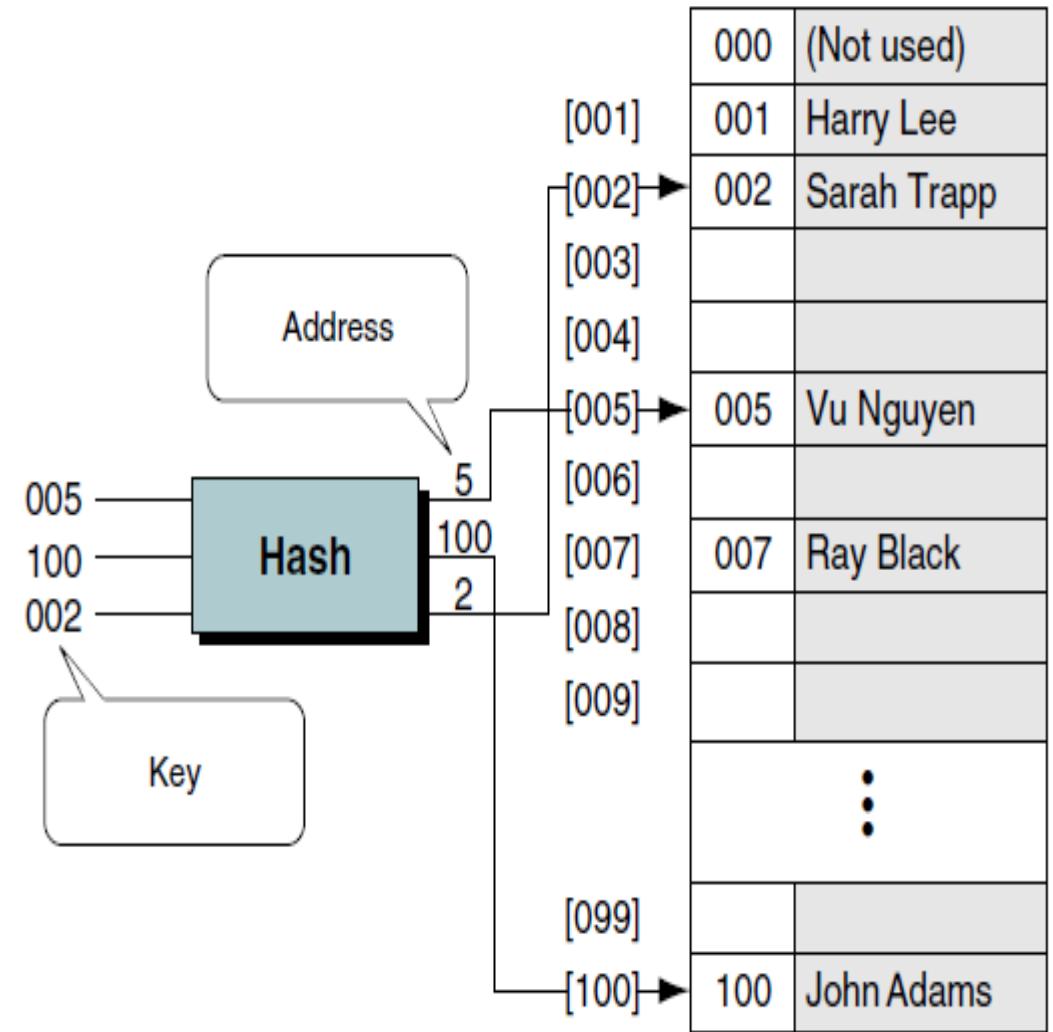
Hashing Methods.



Direct Method.

In direct hashing the key is the address without any algorithmic manipulation.

The data structure must therefore contain an element for every possible key.



Direct Hashing of Employee Numbers

Subtraction Method.

Sometimes keys are consecutive but do not start from 1.

For example, a company may have only 100 employees, but the employee numbers start from 1001 and go to 1100. In this case **subtraction hashing** is used that subtracts 1000 from the key to determine the address.

Limitation: Both direct & subtraction hashing, can be used only for small lists in which the keys map to a densely filled list.

The direct and subtraction hash functions both guarantee a search effort of one with no collisions.

They are **one-to-one hashing methods**: only one key hashes to each address.

Modulo Division Method.

Also known as **division remainder**

This method **divides the key by the array size and uses the remainder for the address.**

Simple hashing algorithm shown below in which listSize is the number of elements in the array:

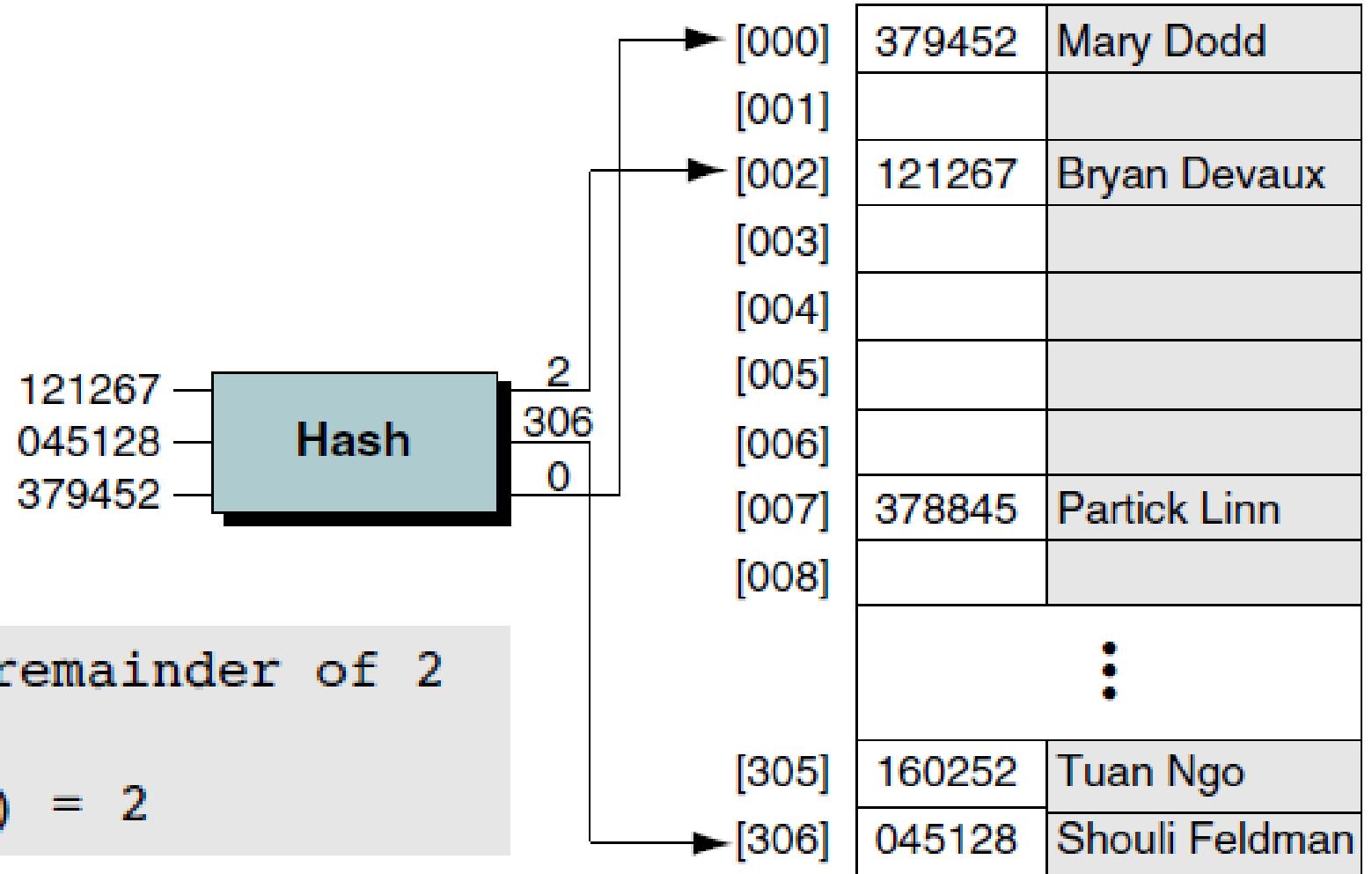
```
address = key MODULO listSize
```

This algorithm works with any list size, but a list size that is a prime number produces fewer collisions than other list sizes.

Whenever possible, to make the array size a prime number.

Modulo Division Method Example

To demonstrate,
hash Bryan Devaux's employee
number, 121267.



$$121267 / 307 = 395 \text{ with remainder of } 2$$

Therefore: $\text{hash}(121267) = 2$

Digit Extraction Method.

Using digit extraction selected digits are extracted from the key and used as the address. For example, using our six-digit employee number to hash to a three-digit address (000–999), we could select the first, third, and fourth digits (from the left) and use them as the address.

379452	→	394
121267	→	112
378845	→	388
160252	→	102
045128	→	051

Midsquare Method.

In **midsquare hashing** the **key is squared** and the address is selected from the middle of the squared number.

Limitation: Size of the key.

Given a key of six digits, the product will be 12 digits, which is beyond the maximum integer size of many computers. Given a key of 9452, the midsquare address calculation is shown below using a four-digit address (0000–9999).

$$9452^2 = 89340304 : \text{address is } 3403$$

Select a portion of the key, such as the middle three digits, and then use them rather than the whole key

379452:	$379^2 = 143641$	364
121267:	$121^2 = 014641$	464
378845:	$378^2 = 142884$	288
160252:	$160^2 = 025600$	560
045128:	$045^2 = 002025$	202

Folding Method.

Fold shift

In **fold shift** the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part.

For example, imagine that we want to map Social Security numbers into three-digit addresses. We divide the nine-digit Social Security number into three three-digit numbers, which are then added. If the resulting sum is greater than 999, we discard the leading digit.

Fold boundary

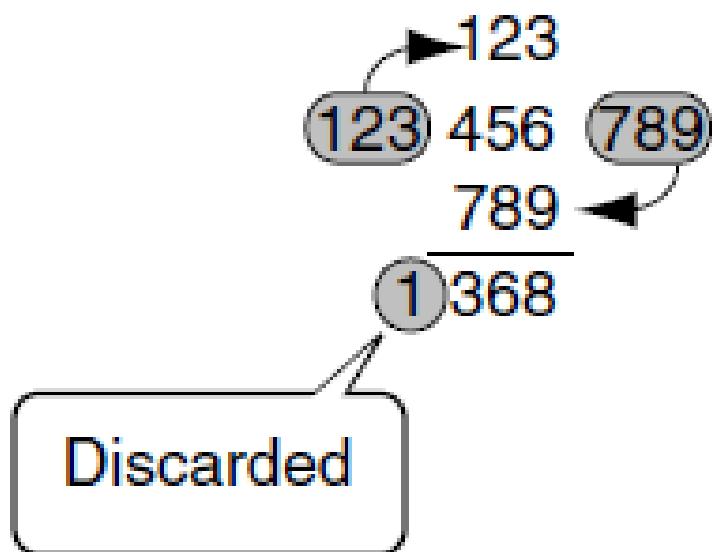
In fold boundary the left and right numbers are folded on a fixed boundary between them and the center number. The two outside values are thus reversed,

Example 123 is folded to 321 and 789 is folded to 987.

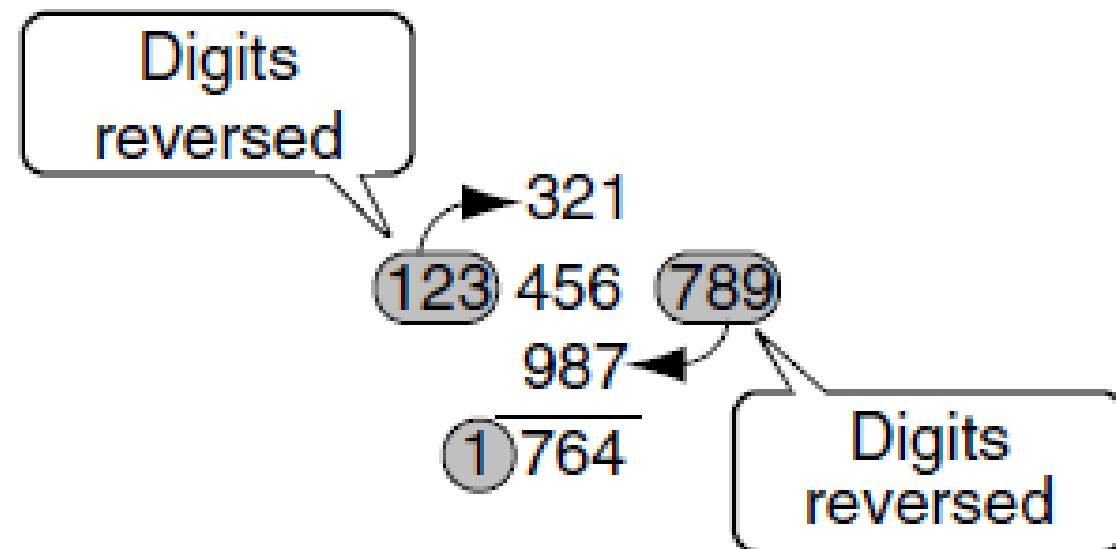
Bolding methods give different hashed addresses.

Folding Method.

Key
123456789



(a) Fold shift



(b) Fold boundary

Rotation Method.

Rotation hashing is generally not used by itself but rather is incorporated in **combination** with other hashing methods. It is most useful when keys are assigned serially, such as we often see in employee numbers and part numbers.

Rotating the last character to the front of the key minimizes the effect of creating synonyms when hashing keys are identical except for the last character

Original key	Rotation	Rotated key
600101	↓	160010
600102		260010
600103		360010
600104		460010
600105		560010

Pseudorandom Generation Method.

The key is used as the seed in a pseudorandom-number generator, and the resulting random number is then scaled into the possible address range using modulo-division.

A common random-number generator

$$y = ax + c$$

To use the pseudorandom-number generator as a hashing method, we set x to the key, multiply it by the coefficient a , and then add the constant c . The result is then divided by the list size, with the remainder being the hashed address.

```
y = ((17 * 121267) + 7) modulo 307
y = (2061539 + 7) modulo 307
y = 2061546 modulo 307
y = 41
```

Hash functions.

All hash functions except direct hashing and subtraction hashing are **many-to-one functions**: many keys hash to one address.

“

The end for today.

”

“

Collision Resolution

”

- Collision Resolution.
- Nature of hashing algorithms **requires some empty elements** in a list at all times.
- In fact, a **full list is defined** as a list with all elements except one contain data.
- **Rule of thumb:** A hashed list **should not be allowed to become more than 75% full.**
- **Load factor:** Number of elements in the list divided by the number of physical elements allocated for the list, expressed as a percentage.
- Traditionally, load factor (symbol alpha α). The formula in which k represents the number of filled elements in the list and n represents the total number of elements allocated to the list is

$$\alpha = \frac{k}{n} \times 100$$

Collision Resolution.

- **Clustering:** As data are added to a list and collisions are resolved, some hashing algorithms tend to cause data to group within the list.
- This tendency of data to build up unevenly across a hashed list is known as clustering.
- Clustering is a concern because it is usually created by collisions.
- If the list contains a high degree of clustering, the number of probes to locate an element grows and *reduces the processing efficiency* of the list.

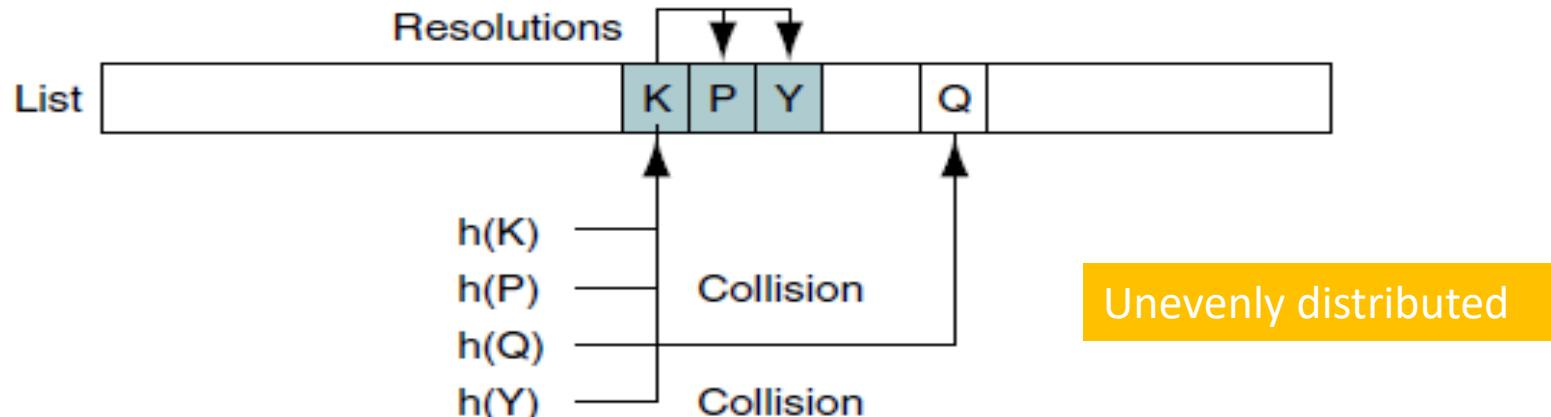
Clustering.

Clustering is the tendency of data to build up unevenly across a hashed list.

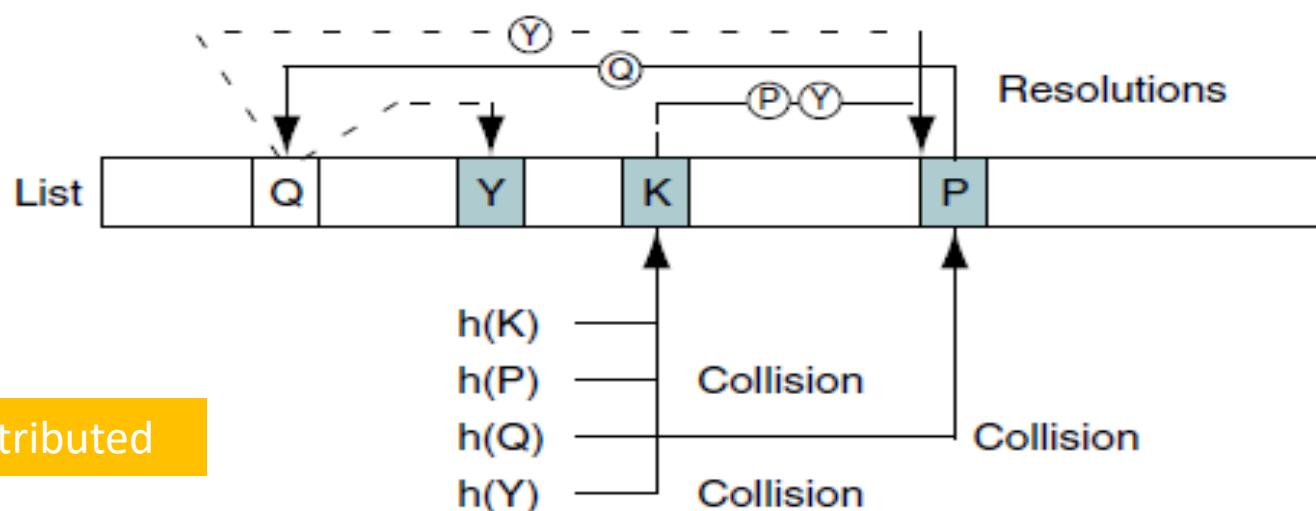
Primary clustering occurs when data build up around a home address.

Secondary clustering occurs when data build up along a collision path in the list.

Clustering



(a) Primary clustering



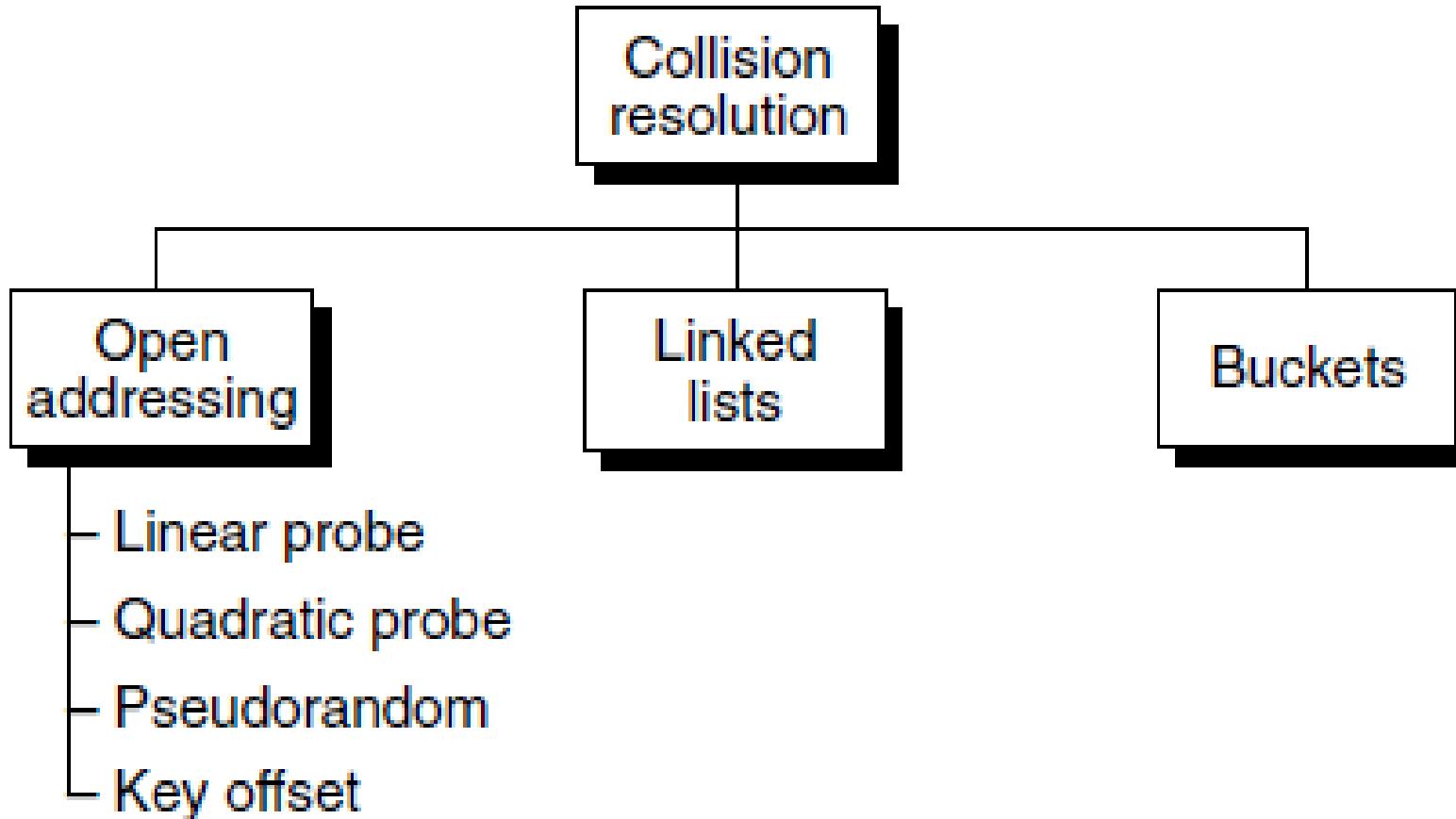
(b) Secondary clustering

Secondary Clustering.

- Effect of secondary clustering, consider an **extreme example**: *Assume that a hashing algorithm hashes each key to same home address.*
- Locating the **first element** inserted into the list takes only **one probe**.
- Locating the **second element** takes **two probes**.
- Carrying the analogy to its conclusion, locating the **n th element** added to the list takes **n probes**, even if the data are widely distributed across the addresses in the list.

Objective: Design hashing algorithms to minimize clustering, both primary and secondary.

Collision Resolution



Collision Resolution Methods

Open Addressing.

The first collision resolution method, **open addressing**, resolves collisions **in the prime area**, that is, the area that contains all of the home addresses.

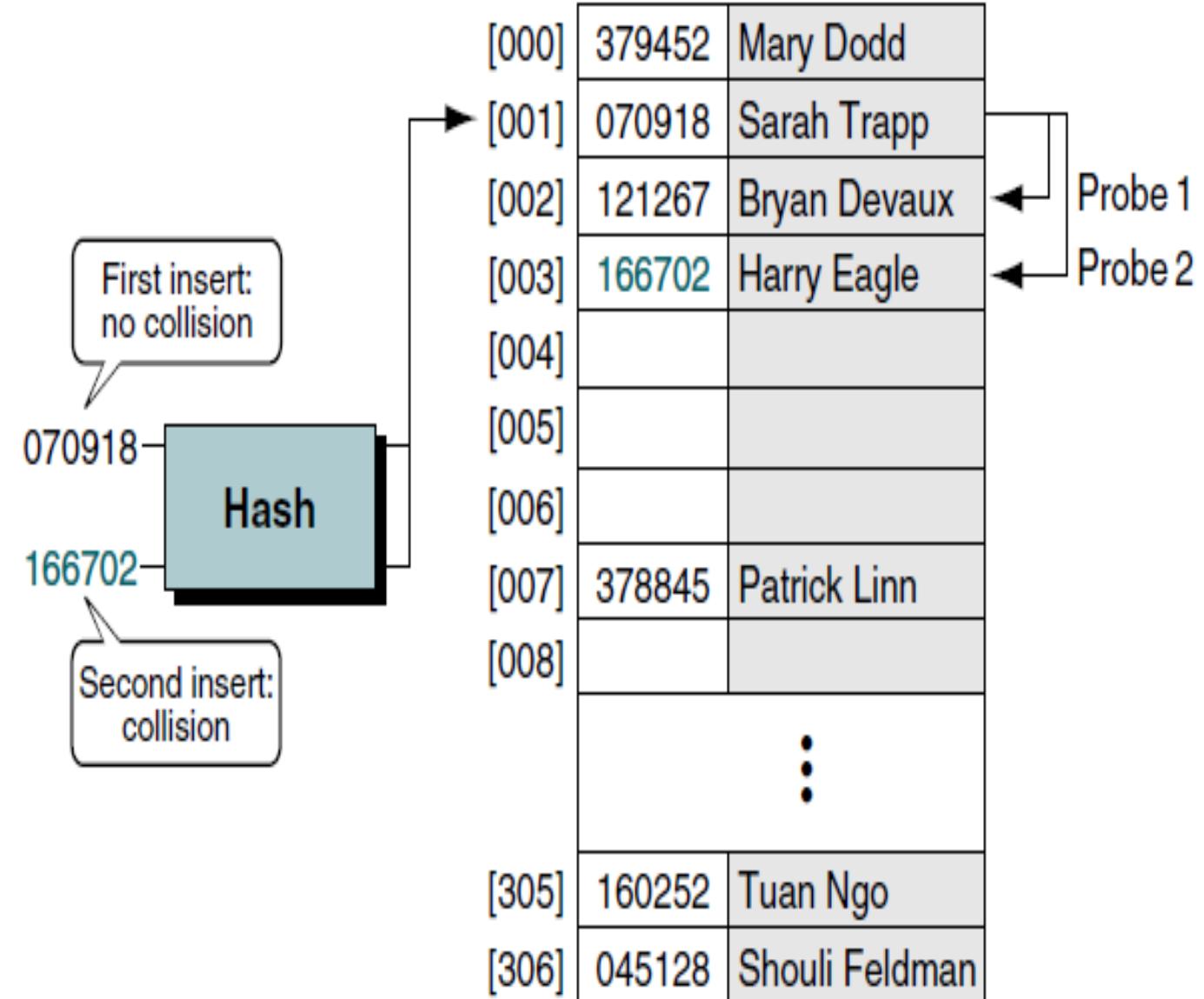
When a collision occurs, the **prime area addresses** are searched for an **open or unoccupied element** where the **new data** can be **placed**.

Linear Probe

Simple linear probe: When data cannot be stored in the home address, it can be resolved by adding 1 to the current address.

Alternative: Add 1, subtract 2, add 3, subtract 4, and so forth until an empty element located.

For example, given a collision at location 341, Try 342, 340, 343, 339, and so forth until an empty element found.



Linear Probe Collision Resolution

Quadratic Probe

The **increment** is the collision probe number squared.

Thus for the first probe we add 1^2 , for the second collision probe we add 2^2 , for the third collision probe we add 3^2 , and so forth until we either find an empty element or we exhaust the possible elements.

To ensure that end of the address list not reached, use **the modulo of the quadratic sum** for the new address.

Probe number	Collision location	Probe ² and increment	New address
1	1	$1^2 = 1$	$1 + 1 \leftarrow 02$
2	2	$2^2 = 4$	$2 + 4 \leftarrow 06$
3	6	$3^2 = 9$	$6 + 9 \leftarrow 15$
4	15	$4^2 = 16$	$15 + 16 \leftarrow 31$
5	31	$5^2 = 25$	$31 + 25 \leftarrow 56$
6	56	$6^2 = 36$	$56 + 36 \leftarrow 92$
7	92	$7^2 = 49$	$92 + 49 \leftarrow 41$
8	41	$8^2 = 64$	$41 + 64 \leftarrow 05$
9	5	$9^2 = 81$	$5 + 81 \leftarrow 86$
10	86	$10^2 = 100$	$86 + 100 \leftarrow 86$

Quadratic Collision Resolution Increments

Disadvantage: Time required to square the probe number.

Limitation: Not possible to generate a new address for every element in the list.

Double Hashing.

1. Pseudorandom Collision Resolution

2. Key Offset

In each method, rather than use an arithmetic probe function, the address is rehashed.

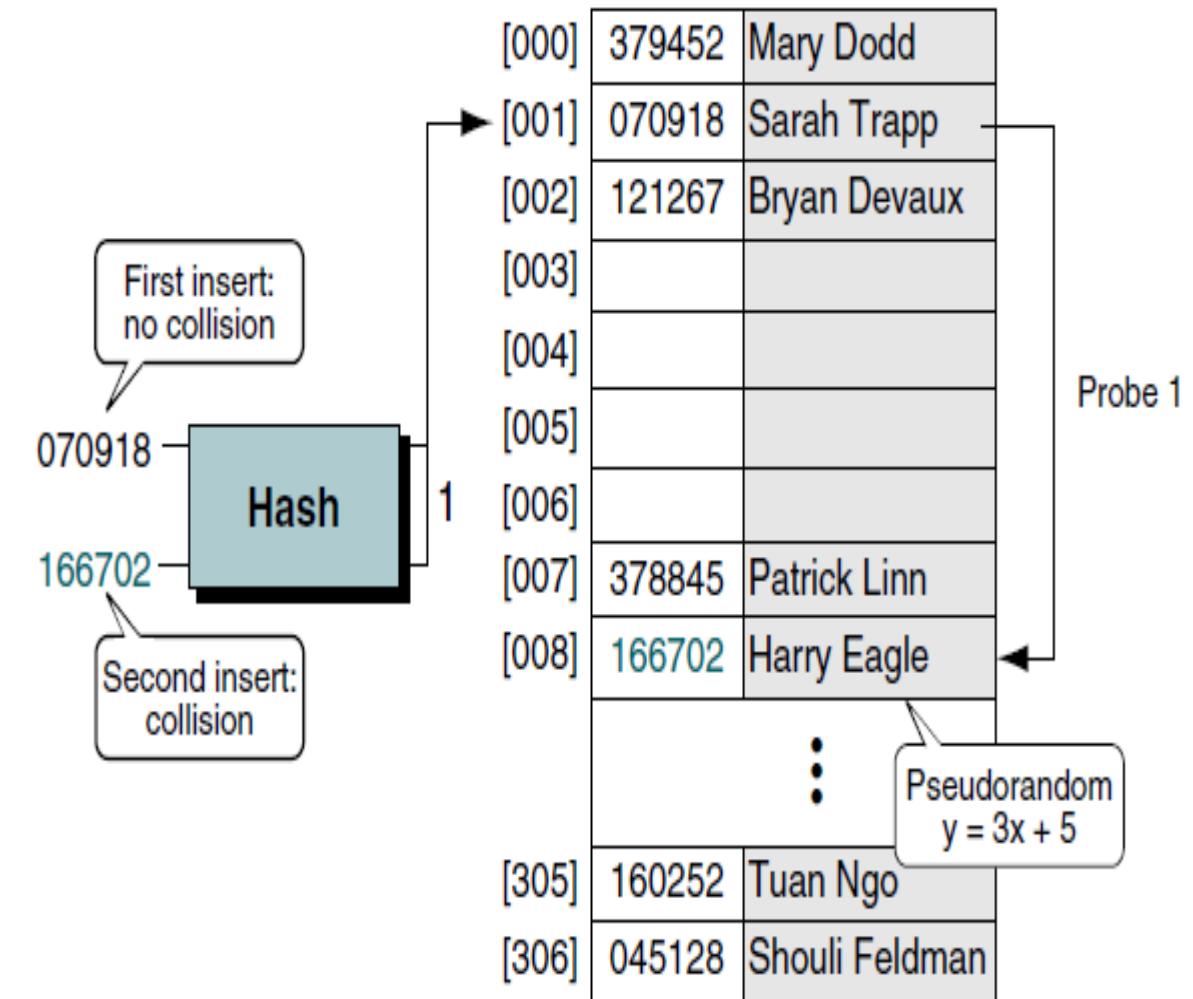
Both methods prevent primary clustering.

Pseudorandom Collision Resolution.

- Uses a pseudorandom number to resolve the collision.
- In this case, rather than use the key as a factor in the random-number calculation, we use the collision address.
- Collision can be resolved using the following pseudorandom-number generator, where a is 3 and c is 5:

```
y = (ax + c) modulo listSize  
= (3 × 1 + 5) Modulo 307  
= 8
```

- Limitation: all keys follow only one collision resolution path through the list.



Key Offset

- Key offset is a double hashing method that produces different collision paths for different keys.
- While the pseudorandom-number generator produces a new address as a function of the previous address, key offset calculates the new address as a function of the old address and the key.
- One of the simplest versions simply adds the quotient of the key divided by the list size to the address to determine the next collision resolution address.

```
offSet = ⌊key/listSize⌋  
address = ((offSet + old address) modulo listSize)
```

- For example, when the key is 166702 and the list size is 307, using the modulo-division hashing method generates an address of 1. This is synonym of 070918 producing a collision at address 1. Using key offset to calculate the next address to get 237.

```
offSet = ⌊166702/307⌋ = 543  
address = ((543 + 001) modulo 307) = 237
```

Key Offset.

- If 237 were also a collision, repeat the process to locate the next address

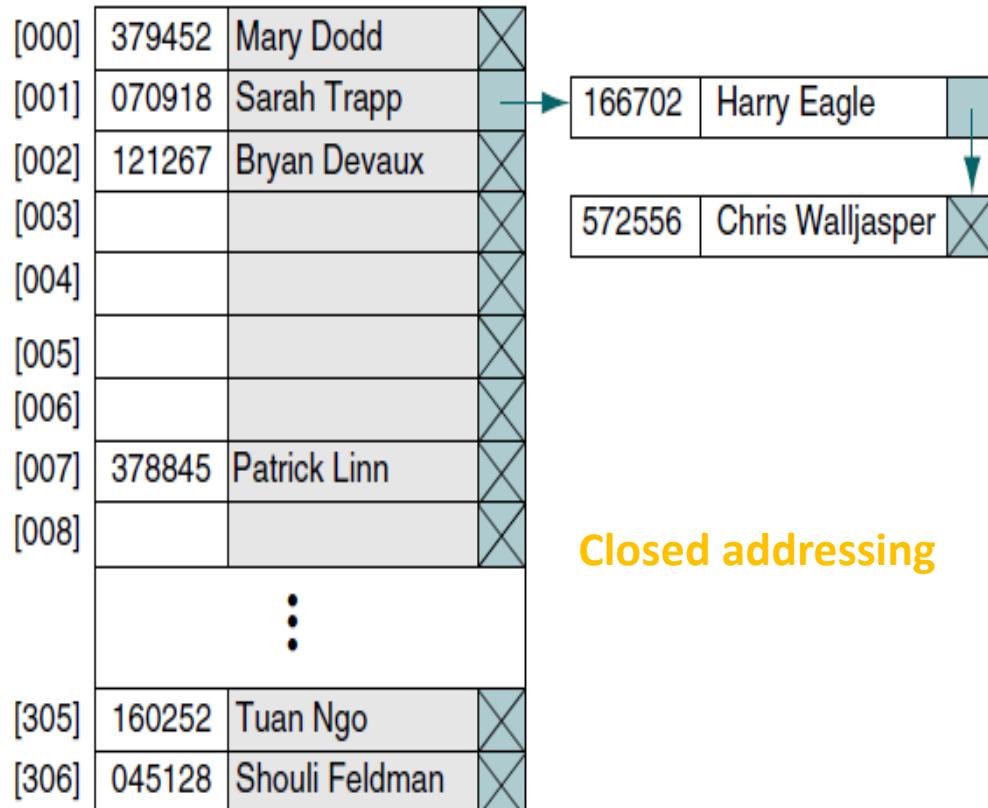
$$\text{offset} = \lfloor 166702 / 307 \rfloor = 543$$

$$\text{address} = ((543 + 237) \bmod 307) = 166$$

Key	Home address	Key offset	Probe 1	Probe 2
166702	1	543	237	166
572556	1	1865	024	047
067234	1	219	220	132

Each key resolves its collision at a different address for both the first and the second probes

Linked List Collision Resolution



The linked list data can be stored in any order, but a last in–first out (LIFO) sequence or a key sequence is the most common.

A major disadvantage to open addressing is that each collision resolution increases the probability of future collisions.

- **Linked list collision resolution uses a separate area to store collisions and chains all synonyms together in a linked list.**
- It uses two storage areas: the **prime area** and the **overflow area**. Each element in the prime area contains an additional field—a link head pointer to a linked list of overflow data in the overflow area.
- **When a collision occurs, one element is stored in the prime area and chained to its corresponding linked list in the overflow area.**
- Although the overflow area can be any data structure, it is typically implemented as a linked list in dynamic memory.

Bucket Hashing

- Another approach to handling the **collision problems** is **bucket hashing**, in which keys are hashed to buckets, nodes that accommodate multiple data occurrences.
- Because a **bucket can hold multiple data, collisions are postponed until the bucket is full.**
- Example: Data for three entries, all of which hashed to address 1. No collision until the fourth key, 572556 in our example, is inserted into the list. When a collision finally occurs—that is, when the bucket is full—any of the collision resolution methods may be used.

[000]	Bucket 0	379452	Mary Dodd
[001]	Bucket 1	070918	Sarah Trapp
		166702	Harry Eagle
		367173	Ann Giorgis
[002]	Bucket 2	121267	Bryan Devaux
		572556	Chris Walljasper
			⋮
[307]	Bucket 307	045128	Shouli Feldman

Linear probe placed here

When a collision finally occurs—that is, when the bucket is full—any of the collision, resolution methods may be used.

The end.