



DSE 2155 DATA STRUCTURES
[3 1 0 4]

Linda Varghese
Savitha G

Department of Computer Applications, MIT, Manipal.



Classroom Rules of Engagement.

- 🔔 All laptop computers, mobile phones, tablet computers must be closed during all classroom hours (**offline class**).
- 🔔 Computers distract the most people behind and around the user.
- 🔔 **Maintain social distance and wear masks (correctly) at all times inside and outside class.**
- 🔔 Make your own notes. Slides are not enough.
- 🔔 Self study is paramount.
- 🔔 **All homework** to be **completed** before class commences.

How do you improve your performance?

- ⌚ To transfer information from your short-term memory to your long-term memory, that information must be imposed on your mind ***at least three times.***
- ⌚ You should always try the following:
 - ⌚ Look at the notes/slides (**IF ANY**) before class.
 - ⌚ Attend all lectures (if possible).
 - ⌚ Review the lecture during the evening.
 - ⌚ Rewrite and summarize the slides in your words.
- ⌚ In addition to this, you should:
 - ⌚ Get a reasonable nights sleep (apparently this is when information is transferred to your long-term memory), and
 - ⌚ Eat a good breakfast (also apparently good for the memory)

Douglas Wilhelm Harder, MMath



Agenda

- ▶ Syllabus abstract and Textbook
- ▶ Definitions of basic terms
- ▶ Example
- ▶ Why study Data structure and algorithms?
- ▶ What are the different classifications
- ▶ Some example data structures you study in this course

MAIN TOPICS:-

DSE- 2155 : DATA STRUCTURES [3 1 0 4]

Introduction, Programming fundamentals, Recursion, Stacks, Queues and their applications, Sparse Matrix, Pointers and dynamic memory allocation,

Linked Lists: Singly linked lists, Dynamically Linked Stacks and Queues, Polynomial representation and polynomial operations using singly linked list, Singly Circular Linked List, Doubly Linked Lists,

Trees: Binary trees, Heaps, Binary Search Trees, Threaded binary trees,

Graphs: Terminologies, Depth First Search, Breadth First Search, Sorting and searching Techniques.

Text books

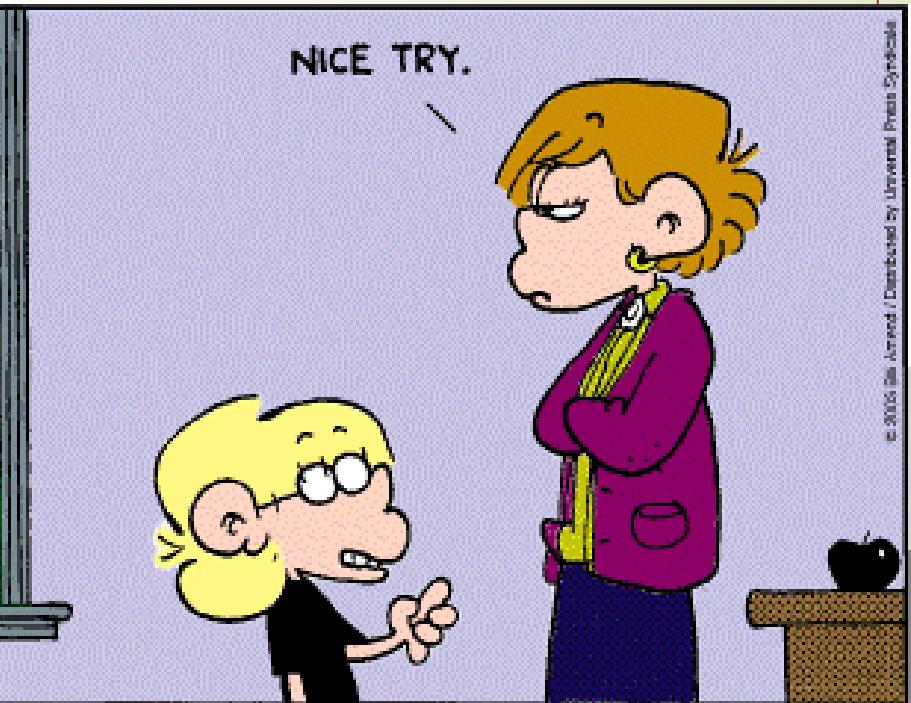
- ▶ Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C++, 2nd Edition.
- ▶ Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C, 2nd Edition.
- ▶ Behrouz A. Forouzan, Richard F. Gilberg, A Structured Programming Approach Using C, 3e, Cengage, Learning India Pvt.Ltd, India,2007.
- ▶ Behrouz A. Forouzan, Richard F. Gilberg, Data Structures, A Pseudocode approach Using C, 2e, Cengage, learning India Pvt.Ltd, India, 2009.
- ▶ Debasis Samanta, Classic Data structures- 2nd edition, PHI Learning Private Limited , 2010

Language of Implementation.

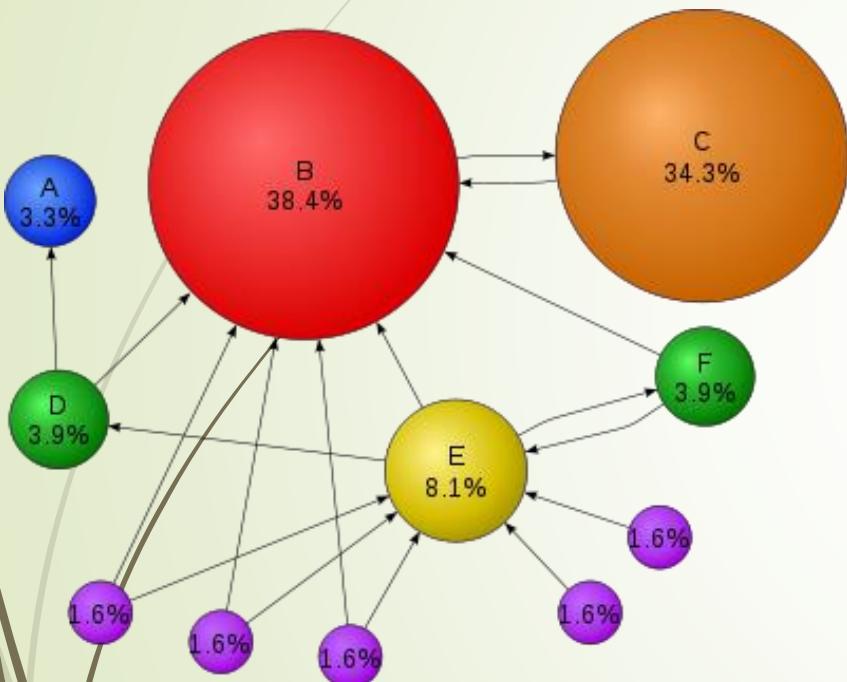
- ➊ You will be using the C++ programming language in this course
 - ➋ Knowledge of syntax of C++ pre-requisite

```
#include <stdio.h>
int main(void)
{
    int count;
    for(count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

AMEND 10-3



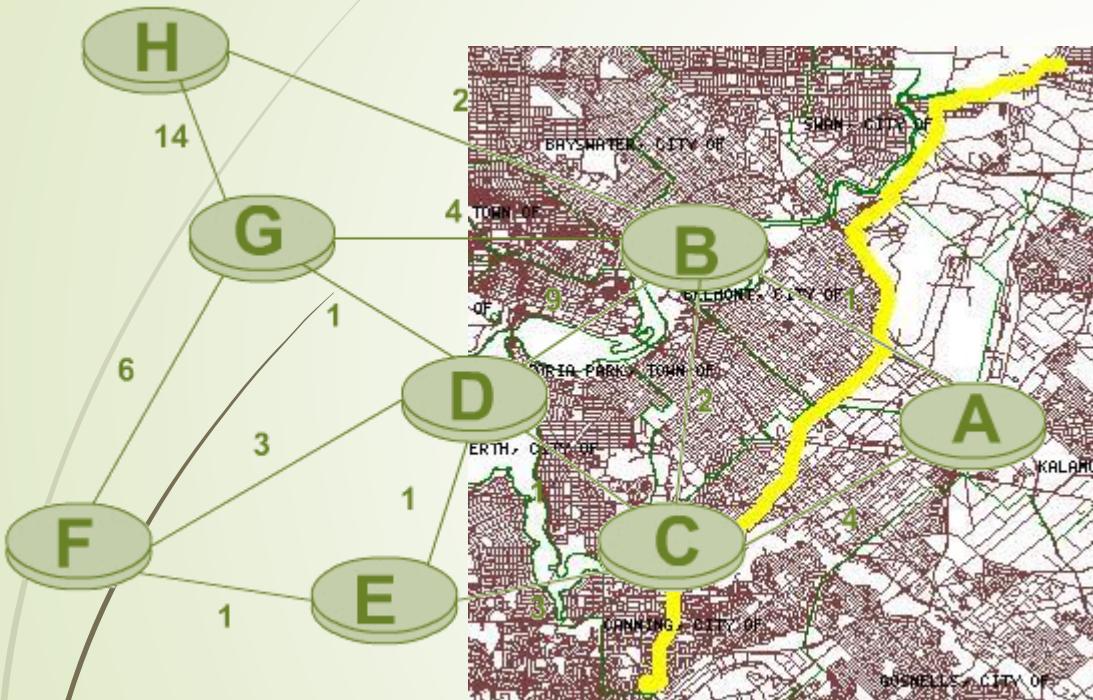
Why you want to study Algorithms?



- 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

Basic Terminologies

- ▶ **Data:** are simply a value or a set of values of different or same type which is called data types like string, integer, char etc.
- ▶ **Structure:** Way of organizing information, so that it is easier to use

Data Structure

- In simple words we can define **data structures** as
 - Its a way of storing and organizing data in such a way that it is easier to use.
 - A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently in programs or algorithms.
 - A scheme for organizing related pieces of information.



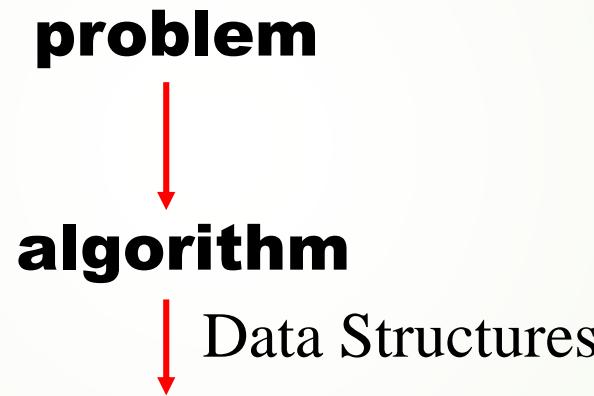
Algorithm

- Set of instructions which work on the data structure to solve a particular problem

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.

Map Navigation
A → B



input
Graphs

**“computer”+
programs**

output
Path

How to study algorithms?

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



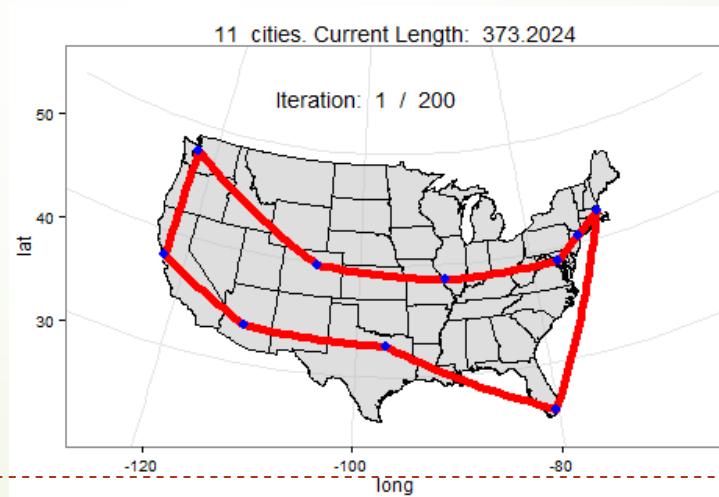
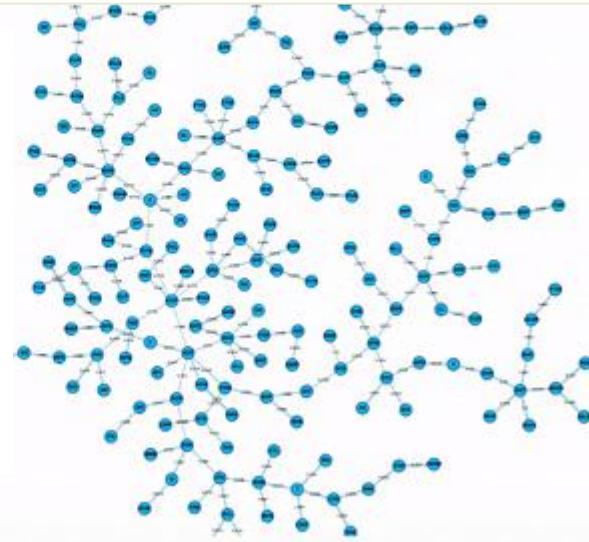
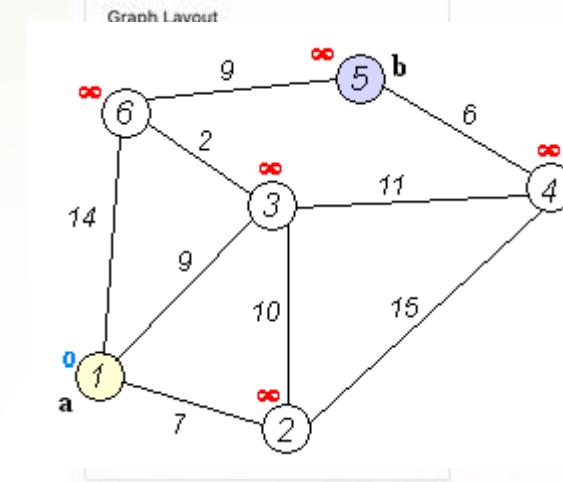
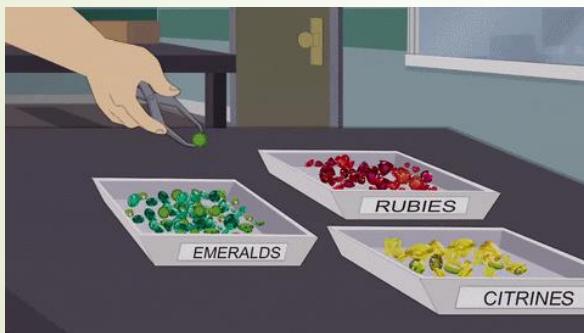
Some Important Points.

- 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 an 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?

↳ Data

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

↳ Types of data

- ↳ Textual: For example, your name (Muhammad)
- ↳ Numeric: For example, your ID (090254)
- ↳ Audio: For example, your voice
- ↳ 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:**

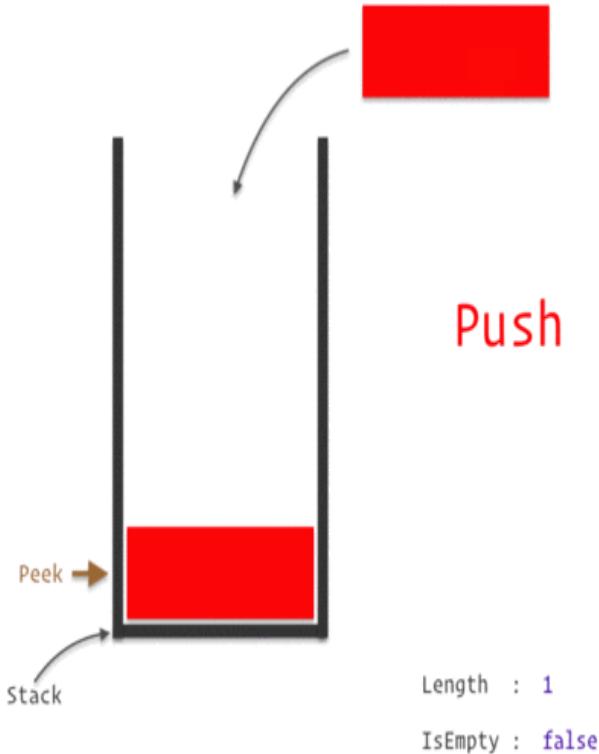
An abstract data type(ADT) is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

A logical view of how we view data and operations.

Abstract Data Type (ADT)

- An **abstract data type** is a data declaration packaged together with the operations that are meaningful on the data type (composite types).
- In other words, we encapsulate the data and the operation on data and we hide them from the user.
- ADT has
 1. **Declaration of data** (set of values on which it operates)
 2. **Declaration of operation**(set of functions)and hides the representation and implementation details

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 Structure ...

- ▶ **Simple Data Structure:** Simple data structure can be constructed with the help of primitive data types. A primitive data structure used to represent the standard data types of any one of the computer languages (integer, Character, float etc.).
- ▶ **Compound Data Structure:** Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as **Linear and Non-Linear** Data Structure.

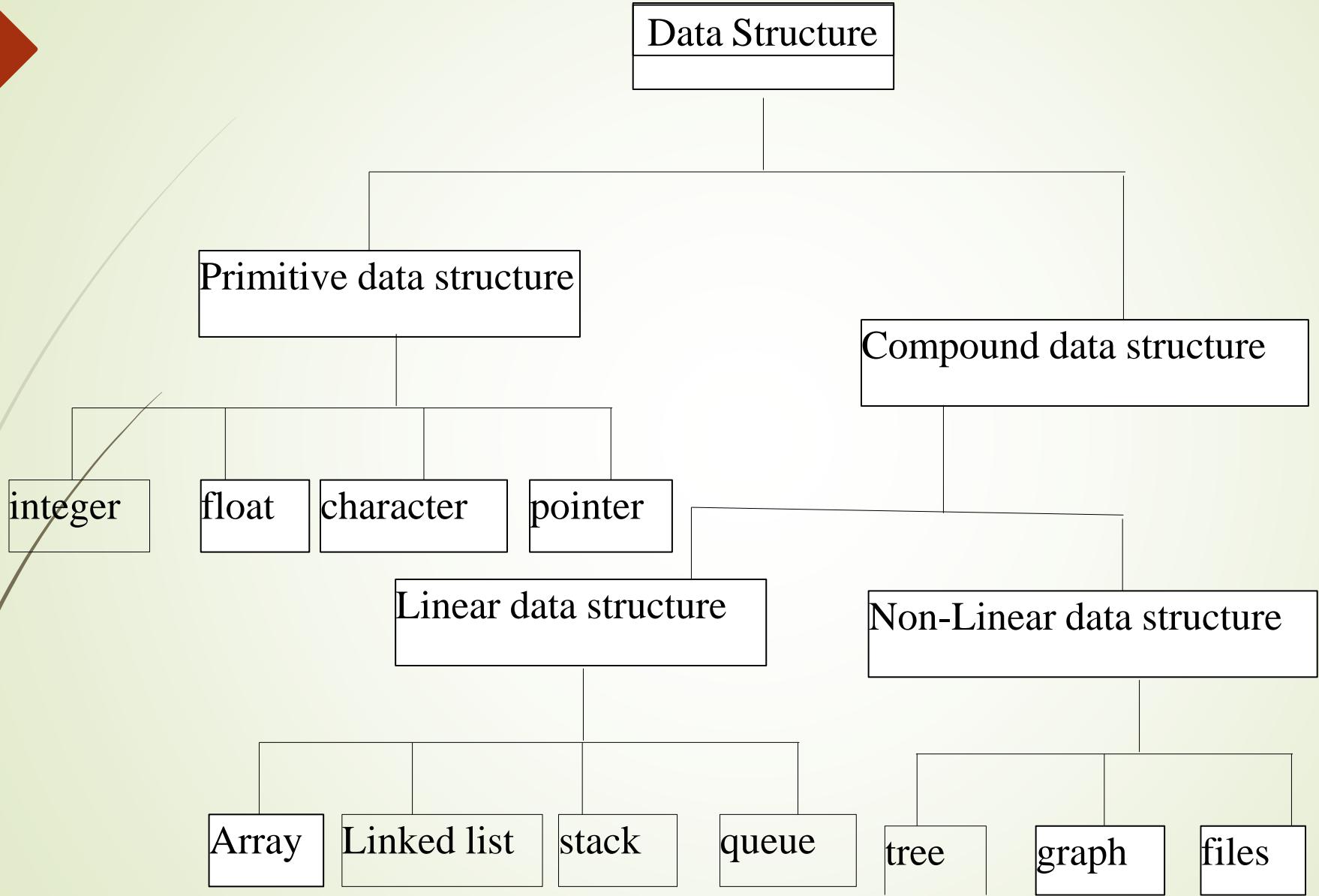
Another way of classification

- **Linear Data Structures:** A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached.
- Continuous arrangement of data elements in the memory. **Relationship of adjacency** is maintained between the data elements.

Ex: Arrays, Linked Lists

- **Non-Linear Data Structures:** Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential 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.

Ex: Trees, Graphs



Types of Data Structure

- ▶ **Array:** is commonly used in computer programming to mean a contiguous block of memory locations, where each memory location stores one fixed-length data item. e.g. Array of Integers int a[10], Array of Character char b[10]

Array of Integers									
0	1	2	3	4	5	6	7	8	9
5	6	4	3	7	8	9	2	1	2

Array of Character									
0	1	y	3	4	5	h	7	8	9
a	6	4	k	7	8	9	q	1	2

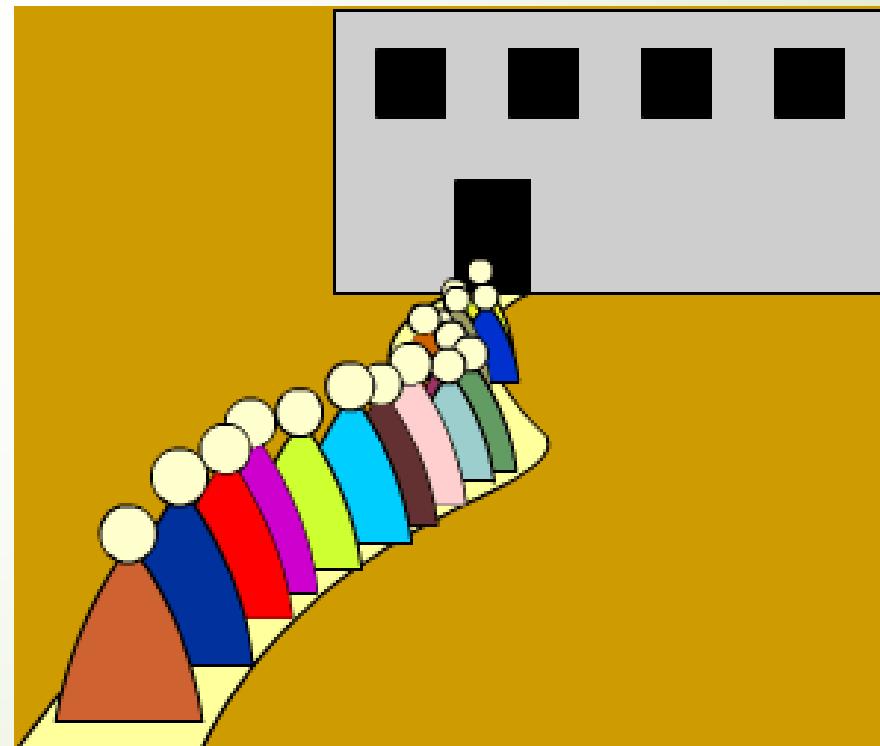
Types of Data Structure ...

- **Stack:** A stack is a data structure in which items can be inserted only from one end and get items back from the same end. There , the last item inserted into stack, is the first item to be taken out from the stack. In short its also called Last in First out [LIFO].



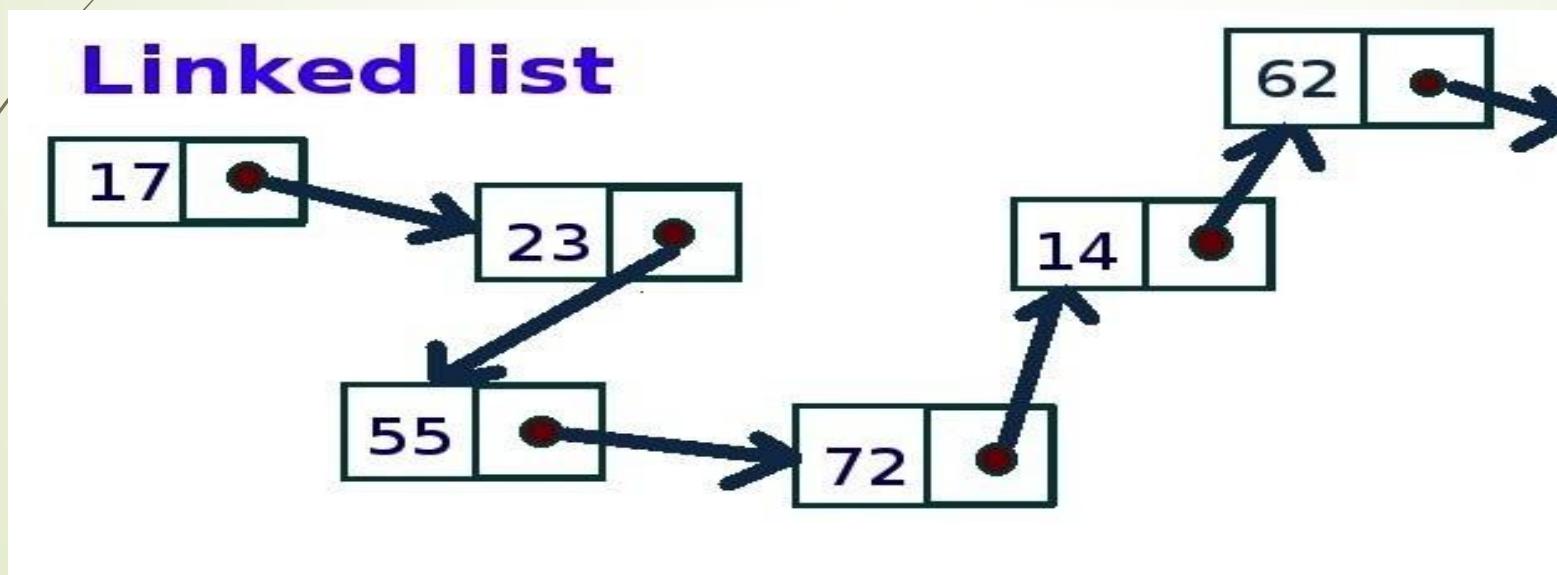
Types of Data Structure ...

- ▶ **Queue:** A queue is two ended data structure in which items can be inserted from one end and taken out from the other end. Therefore ,the first item inserted into queue is the first item to be taken out from the queue. This property is called First in First out [FIFO].



Types of Data Structure ...

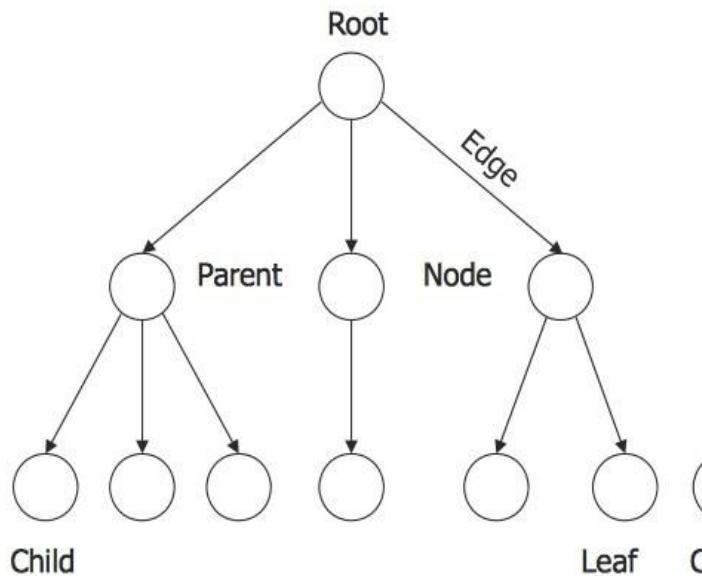
- ▶ **Linked List:** Could alternately used to store items. In linked list space to store items is created as is needed and destroyed when space no longer required to store items. Hence **linked list is a dynamic data structure, space acquired only when need.**



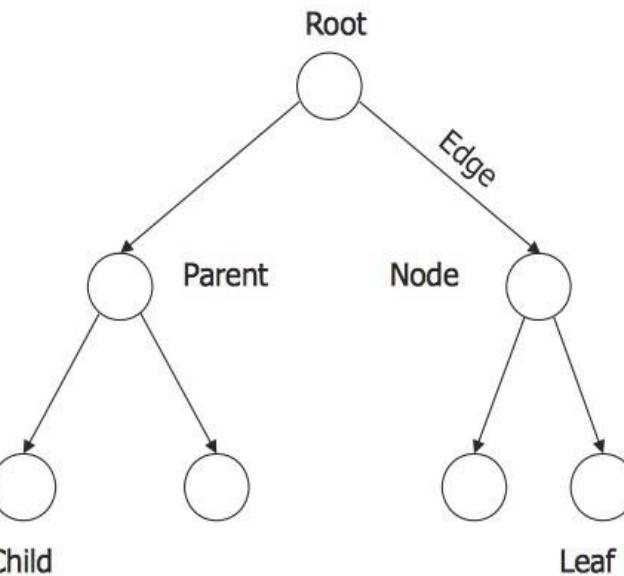
Types of Data Structure ...

- **Tree:** is a non-linear data structure which is mainly used to represent data containing a hierarchical relationship between elements.
- **Binary Tree:** A binary tree is a tree such that every node has at most 2 child and each node is labeled as either left or right child.

■ General tree

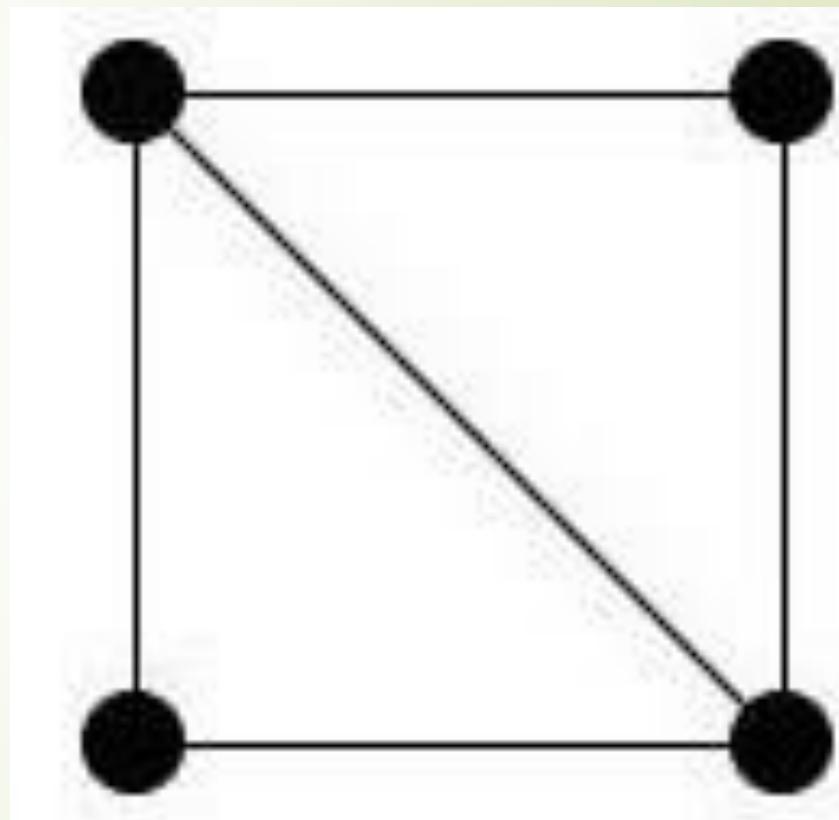


■ Binary Tree



Types of Data Structure ...

- ▶ **Graph:** It is a set of items connected by edges.
- ▶ Each item is called a vertex or node.
- ▶ Trees are just like a special kinds of graphs.
- ▶ Graphs are usually represented by $G = (V, E)$, where V is the set vertices and E is the set of Edges.



Arrays vs. Lists

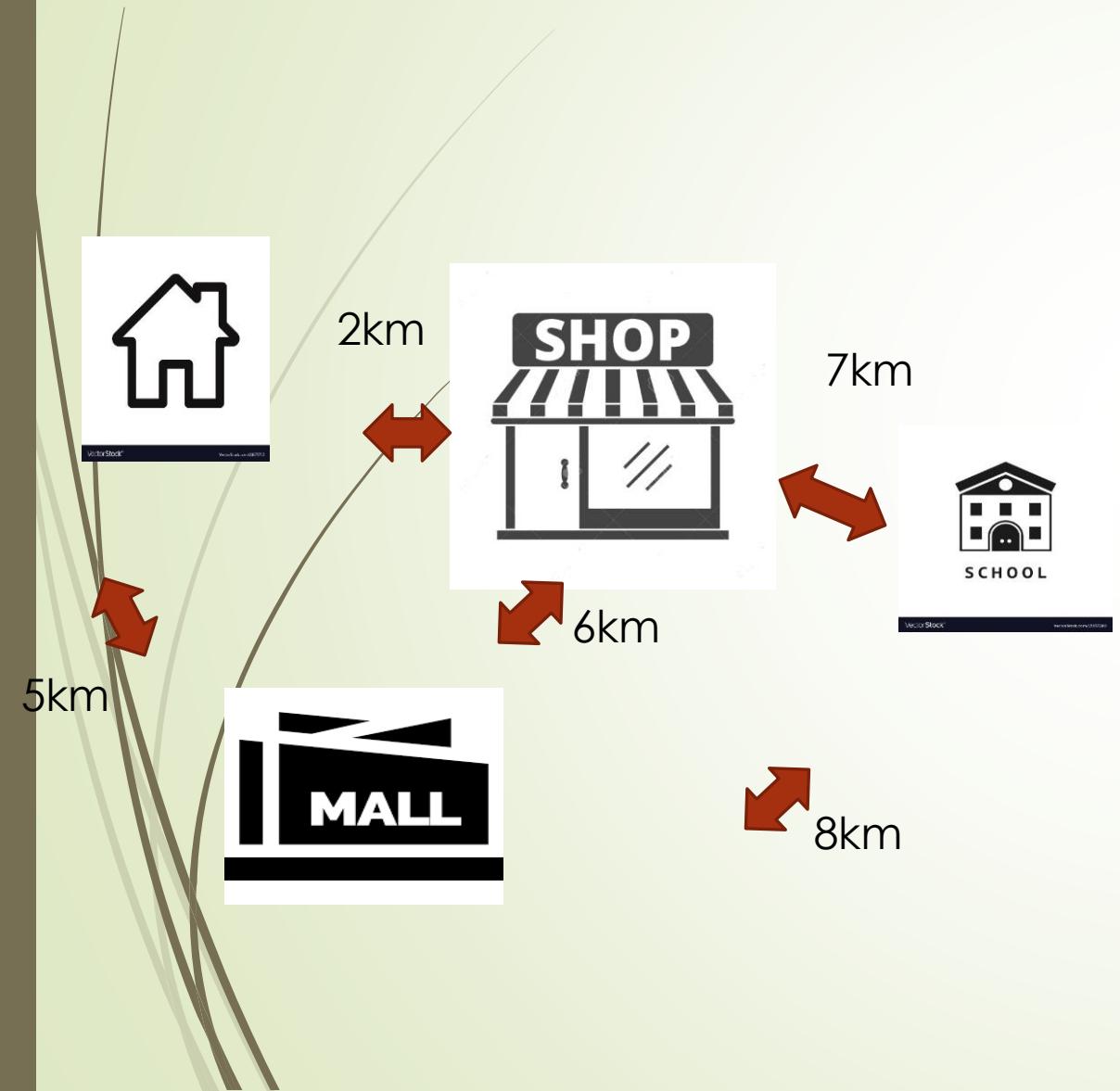
- Array as a Data structure: It is an ordered set which consist of *fixed number of Objects.*
Operations which can be performed on arrays are:
 - Create an array of some fixed size,
 - Store elements, retrieve elements, destroy an array
 - No insertion or deletion possible (fixed size)!!!!
- List: Ordered set consisting of variable number of objects.



Arrays vs. Lists

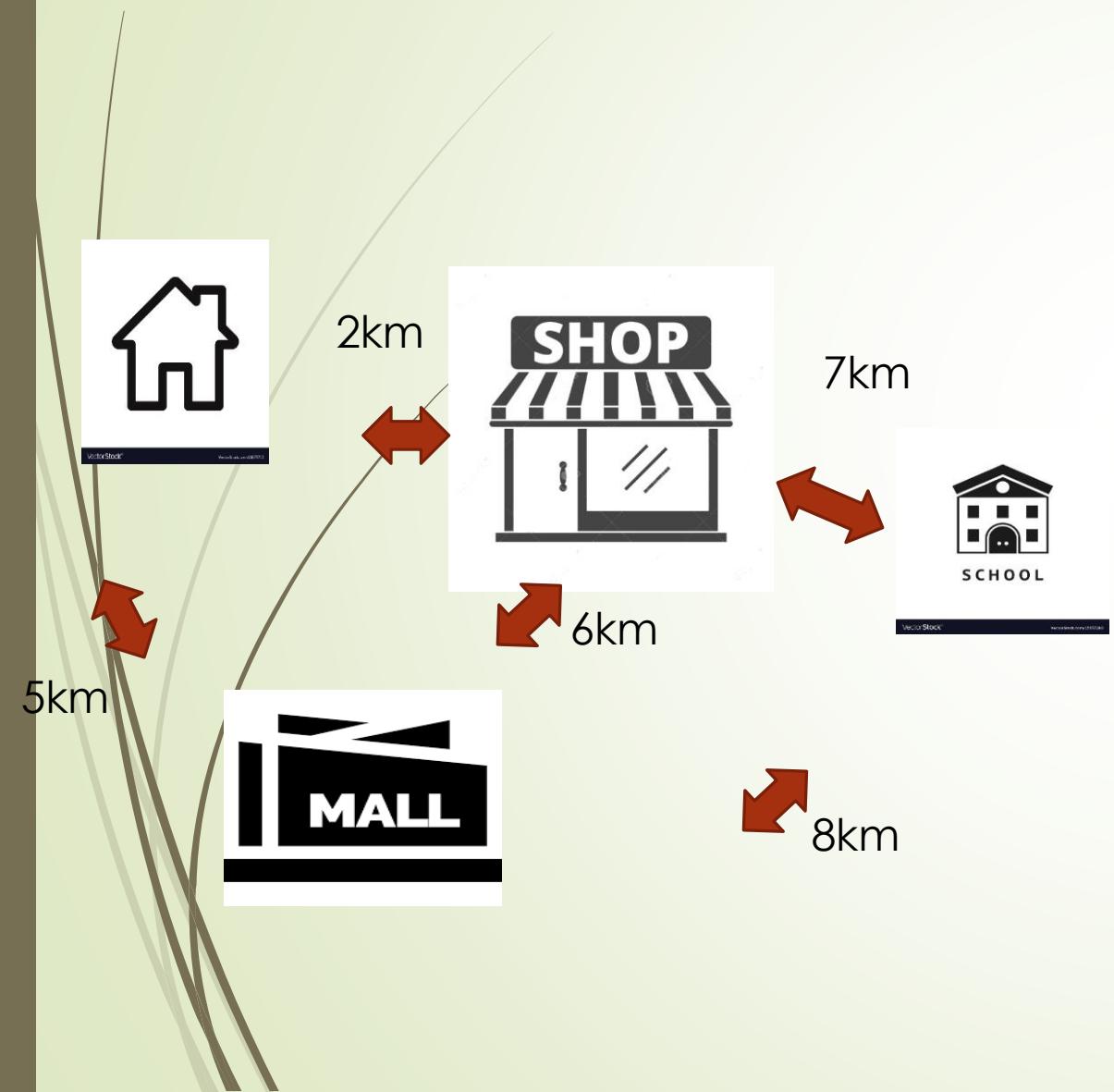
- *Operations which can be performed on Lists are:*
 - *Create a List*
 - *Insert elements, delete elements*
 - *destroy a List*
 - *Size is Not fixed size!!!!*

Example



	home	Mall	shop	school
home	0	5	2	9
Mall	5	0	6	8
shop	2	6	0	7
school	9	8	7	0

Example



	home	Mall	shop	school
home	0	5	2	infinity
Mall	5	0	6	8
shop	2	6	0	7
school	infinity	8	7	0

What is the shortest distance between home and school?

Algorithm to answer question

- Home to school direct path does not exist according to our convention
- So calculate the distance to school from home via shop
- Calculate the distance to school from home via mall
- Find smallest distance among these 2

Why study data structure and algorithms

- ▶ You know computer is a data processor. Without storing and knowing how to store data in different ways how u will process?
- ▶ How u will decide which approach is better ?
- ▶ How u will decide what data structure to use?
- ▶ Which processing method should be used?
- ▶
- ▶ So, It is very very important to know data structures

MCQ's

► The meaning of data is

- A. array B. values C. organization D. Integer

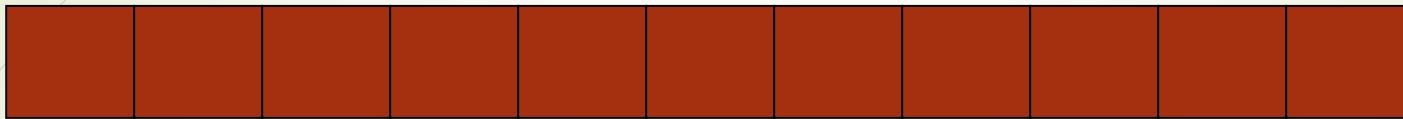
► The meaning of structure is

- A. Storing B. Organizing C. Different ways D. None of these

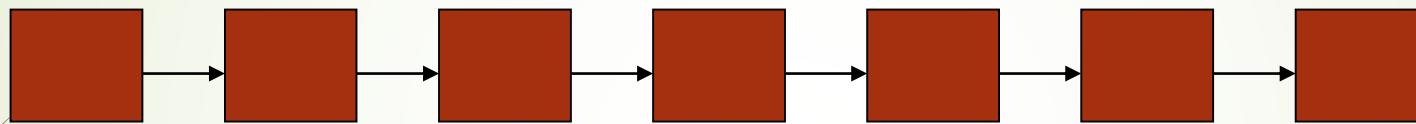
► Float is -----

- A. Value B. data type C. both A and B D. none of these

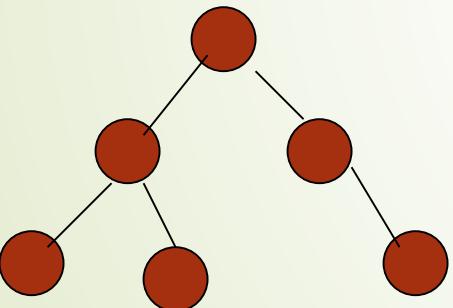
Types of data structures.



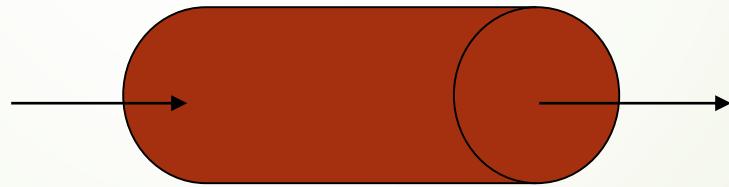
Array



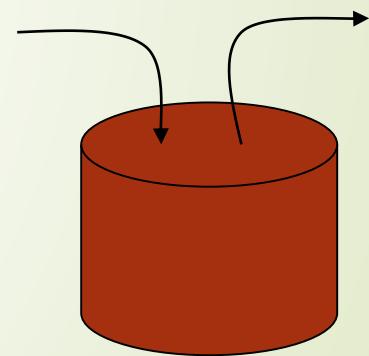
Linked List



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

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



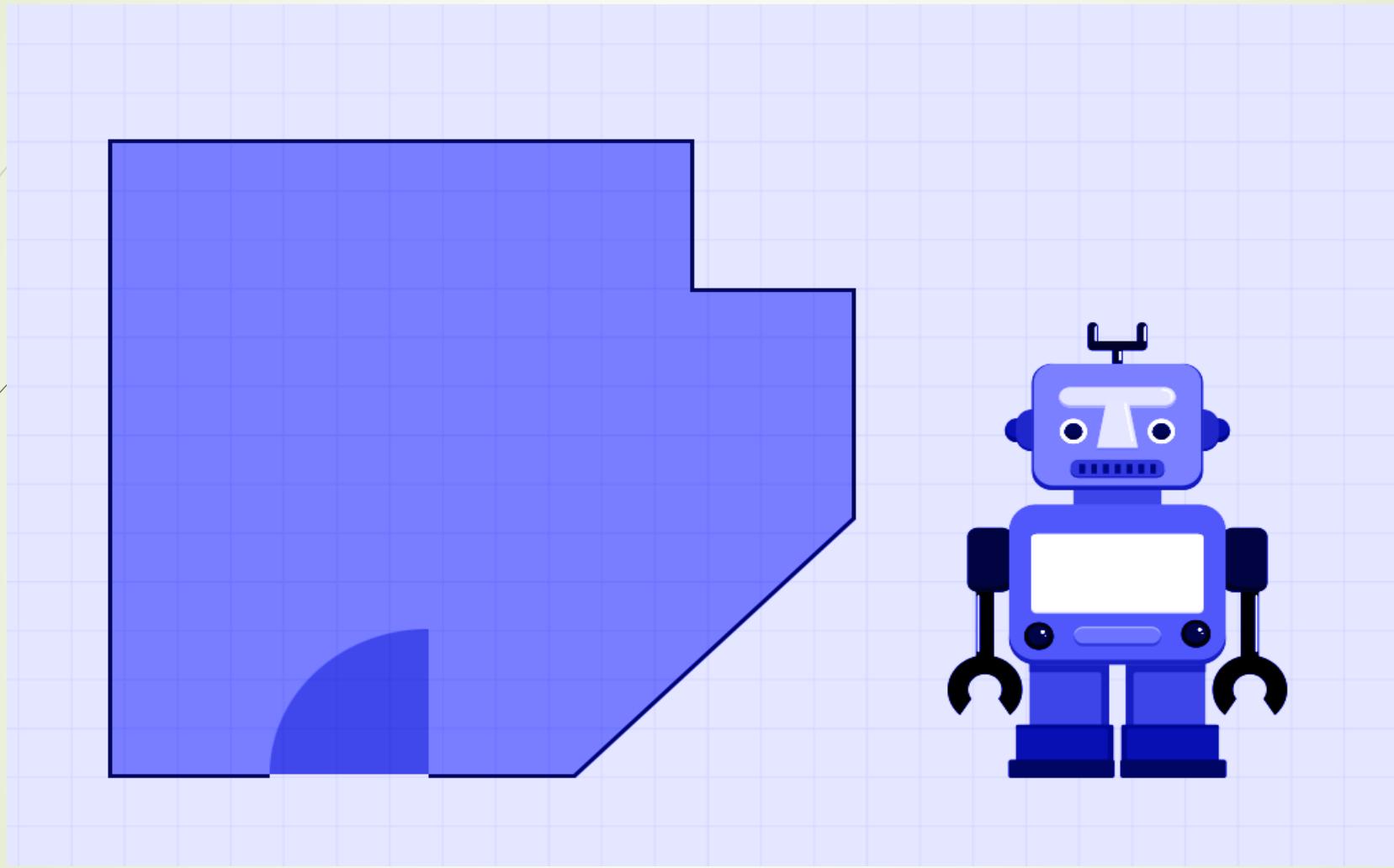
Selecting a Data Structure

- ▶ Analyze the problem to determine the resource constraints a solution must meet.
- ▶ Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
- ▶ Select the data structure that best meets these requirements.

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?

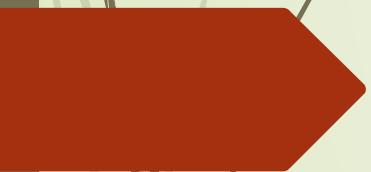
- 🔔 A finite set of instructions which accomplish a particular task
- 🔔 A method or process to solve a problem
- 🔔 Transforms input of a problem to output

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

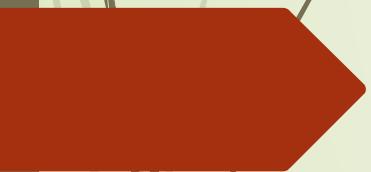
```
while counter < 5 do  
    print "Welcome to MCA4252!"  
    counter ← counter + 1  
end while
```

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 \leftarrow 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 count 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





DSE 2155 DATA STRUCTURES

[3104]

Linda Varghese
Savitha G

Department of Computer Applications, MIT, Manipal.



Classroom Rules of Engagement.

- 🔔 All laptop computers, mobile phones, tablet computers must be closed during all classroom hours **(offline class)**.
- 🔔 Computers distract the most people behind and around the user.
- 🔔 **Maintain social distance and wear masks (correctly) at all times inside and outside class.**
- 🔔 **Make your own notes**. Slides are not enough.
- 🔔 Self study is paramount.
- 🔔 **All homework** to be **completed** before class commences.

How do you improve your performance?

- ⌚ To transfer information from your short-term memory to your long-term memory, that information must be imposed on your mind *at least three times*.
- ⌚ You should always try the following:
 - ⌚ Look at the notes/slides (**IF ANY**) before class.
 - ⌚ Attend all lectures (if possible).
 - ⌚ Review the lecture during the evening.
 - ⌚ Rewrite and summarize the slides in your words.
- ⌚ In addition to this, you should:
 - ⌚ Get a reasonable nights sleep (apparently this is when information is transferred to your long-term memory), and
 - ⌚ Eat a good breakfast (also apparently good for the memory)

Agenda

- Syllabus abstract and Textbook
- Definitions of basic terms
- Example
- Why study Data structure and algorithms?
- What are the different classifications
- Some example data structures you study in this course

MAIN TOPICS:-

DSE- 2155 : DATA STRUCTURES [3 1 0 4]

Introduction, Programming fundamentals, Recursion, Stacks, Queues and their applications, Sparse Matrix, Pointers and dynamic memory allocation,

Linked Lists: Singly linked lists, Dynamically Linked Stacks and Queues, Polynomial representation and polynomial operations using singly linked list, Singly Circular Linked List, Doubly Linked Lists,

Trees: Binary trees, Heaps, Binary Search Trees, Threaded binary trees,

Graphs: Terminologies, Depth First Search, Breadth First Search, Sorting and searching Techniques.

Text books

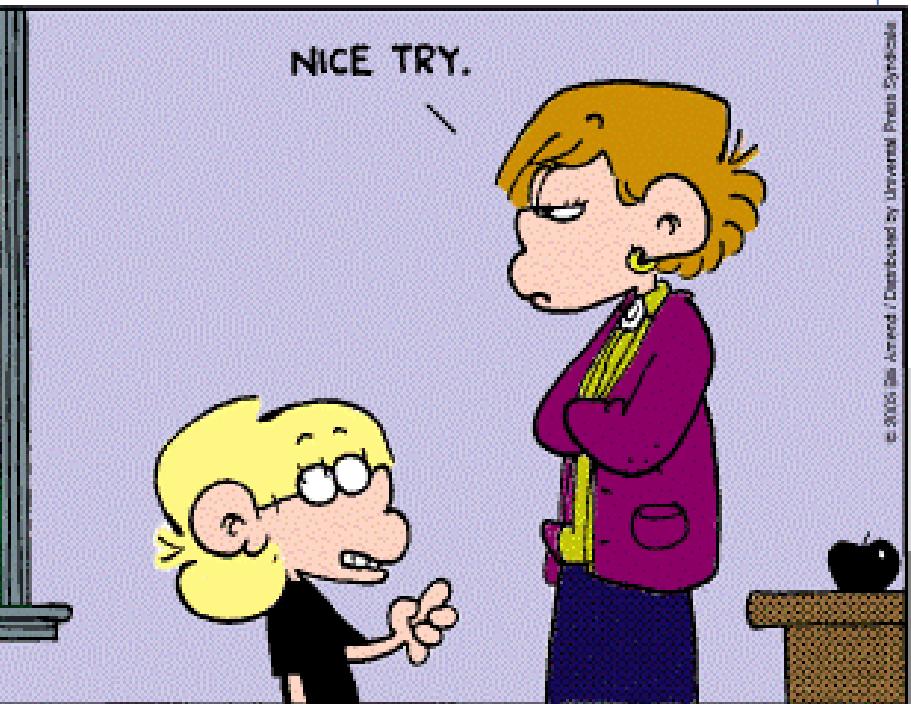
- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C++, 2nd Edition.
- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C, 2nd Edition.
- Behrouz A. Forouzan, Richard F. Gilberg, A Structured Programming Approach Using C, 3e, Cengage, Learning India Pvt.Ltd, India,2007.
- Behrouz A. Forouzan, Richard F. Gilberg, Data Structures, A Pseudocode approach Using C, 2e, Cengage, learning India Pvt.Ltd, India, 2009.
- Debasis Samanta, Classic Data structures- 2nd edition, PHI Learning Private Limited , 2010

Language of Implementation.

- ⌚ You will be using the C++ programming language in this course
 - ⌚ Knowledge of syntax of C++ pre-requisite

```
#include <csfrio.h>
int main(void)
{
    int count;
    for(count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

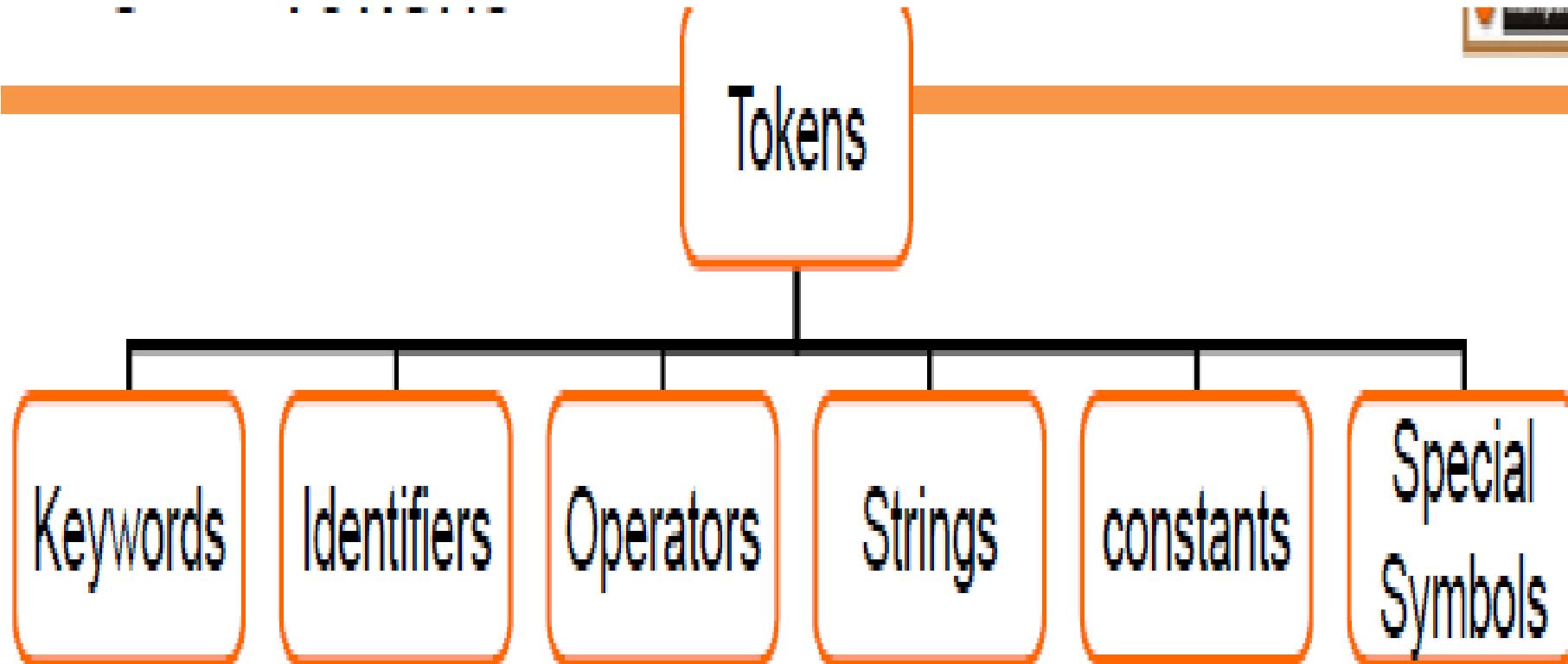
AMEND 10-3





INTRODUCTION TO BASICS OF PROGRAMMING

C++ Tokens



C++ Tokens

- (i) **Keywords**  words that are basically sequence of characters defined by a computer language and have one or more fixed meanings. They are also called as ***reserved words***. Key words cannot be changed. ex. Int, float, do-while, if, else,.....
- (ii) **Identifiers**  words which have to be identified by keywords. user defined names. ex. int amount, float avg,.....
- (iii) **Operators**  +, -, *, %, /,
- (iv) **Strings**  “Manipal”
- (v) **Constants**  -15, 10
- (vi) **Special Symbols**  { } (,.....

C++ Data Types

User Defined Type
Structure, Union, Class
enumeration

Built-in-type

Derived Type
Array ,Function, pointer

Integral type

void

Floating Type

int

float

char

double

```
// my first program in C++
```

Comments

- single line comments: `//.....`
- multiple lines : `/*.....*/`
- All lines beginning with two slash signs (`//` or `/*`) are considered as comments
- They do not have any effect on the behavior of the program
- The programmer can use them to include short explanations or observations within the source code itself.

#include <iostream>

- Lines beginning with a sign (#) are directives for the **preprocessor**.
- They are not regular code lines.
directive `#include <iostream>` tells the preprocessor to include the **iostream** standard header file.
- This specified file **(iostream) includes the declarations of the basic standard input-output library in C++**, and it is included because its functionality is going to be used later.

main()

- The **main function** is the point where all C++ programs start their execution, independently of its location within the source code.
- It is **essential** that all C++ programs have a main function.
- The word main is followed in the code by a pair of **parentheses ()**. That is because it is a **function declaration**.
- Optionally, these parentheses may enclose a list of parameters within them.
- Right after these parentheses we can find the body of the main function enclosed in **braces{ }**.

Simple C++ Program

```
// Display “This is my first C++ program”
// Single line comment
#include <iostream>
Using namespace std; // preprocessor directive
main( ) // Entry point for program execution
{Begin // block of statements:
cout << “This is my first C++ program”; // block of statements:
End}
```

Simple C++ Programs..

```
#include<iostream>
using namespace std;
int main() {
    cout<<"Enter Roll Number and marks of three subjects";
    int RollNo,marks1,marks2,marks3;
    float minimum = 35.0;
    cin>>marks1>>marks2>>marks3;
    avg = (marks1+marks2+marks3)/3;
    if (avg < minimum )
        cout<<RollNo<<"fail";
    else
        cout<<RollNo<<"pass";
    return 0;
}
```

Program to read and display a number

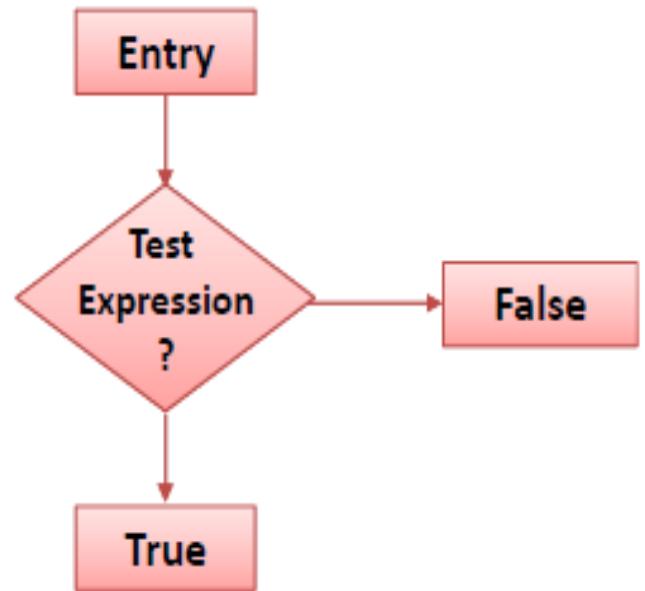
```
#include<iostream>
using namespace std;
int main() {                                //program body begins
    int number;                            //variable declaration
    cout<<"enter number";   // user friendly info display
    cin>>number;                           // reading or input value to the variable
    cout<<"\nthe no. is\n"<<number;  //writing or output variable value
    return 0;
} // end of program
```

C++ decision making and branching statements

- 1. if Statement**
- 2. switch statement**

if statement

- Used to control the *flow of execution* of statements.
- It's a *Two-way decision statement*, used in conjunction with an expression.
- It takes the form: **if(test expression)**
- It allows to *evaluate the expression first* and then, depending on the value; **true** or **false**, it transfer the control to a particular statement.
- i.e. **Two-way branching** as shown in figure



Different forms of if statement

- 1. Simple if statement.**
- 2. if...else statement.**
- 3. Nested if...else statement.**
- 4. else if ladder.**

Simple if Statement

General form of the simplest if statement:

if (*test Expression*)

{

statement-block;

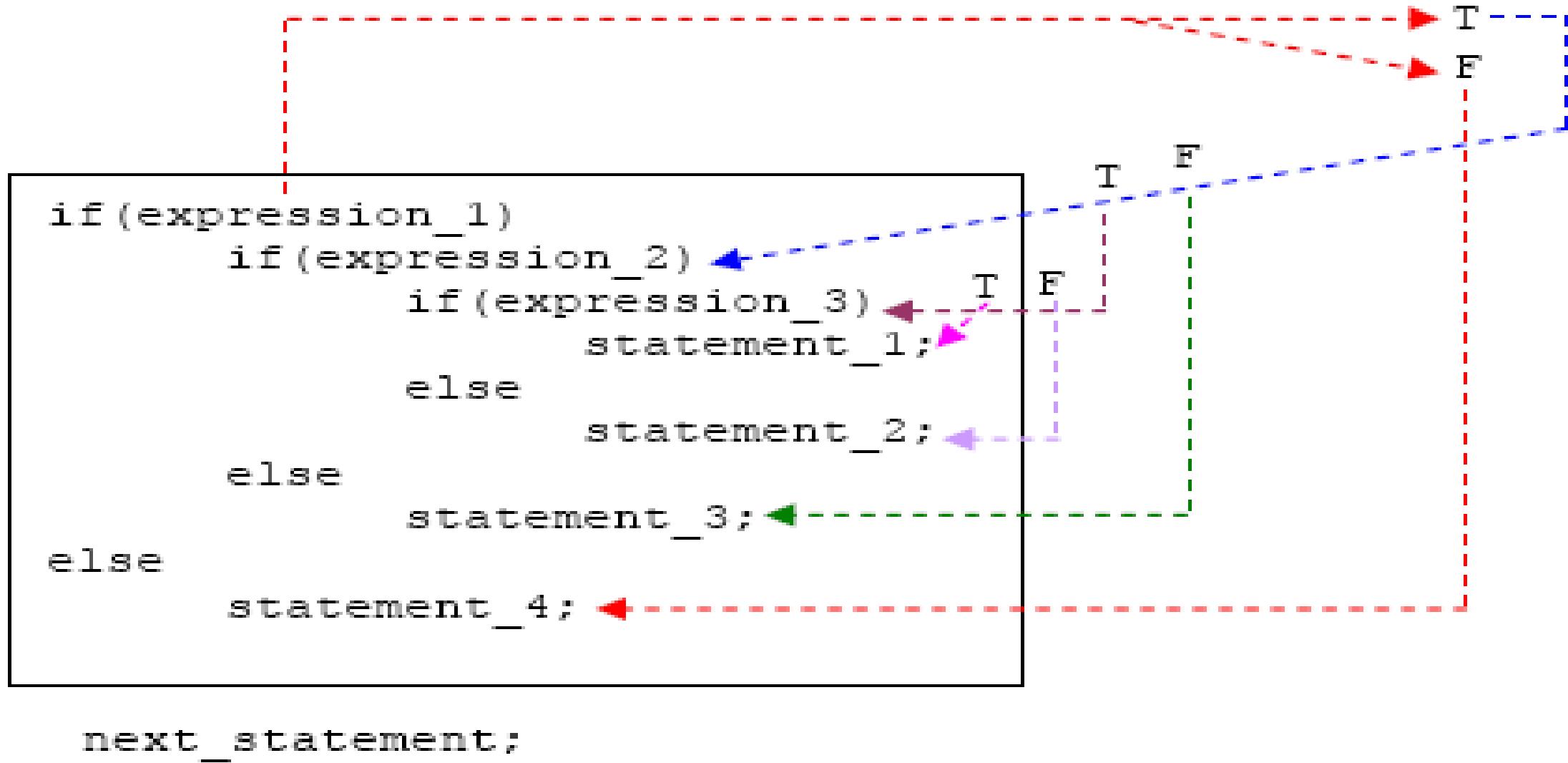
}

statement_x;

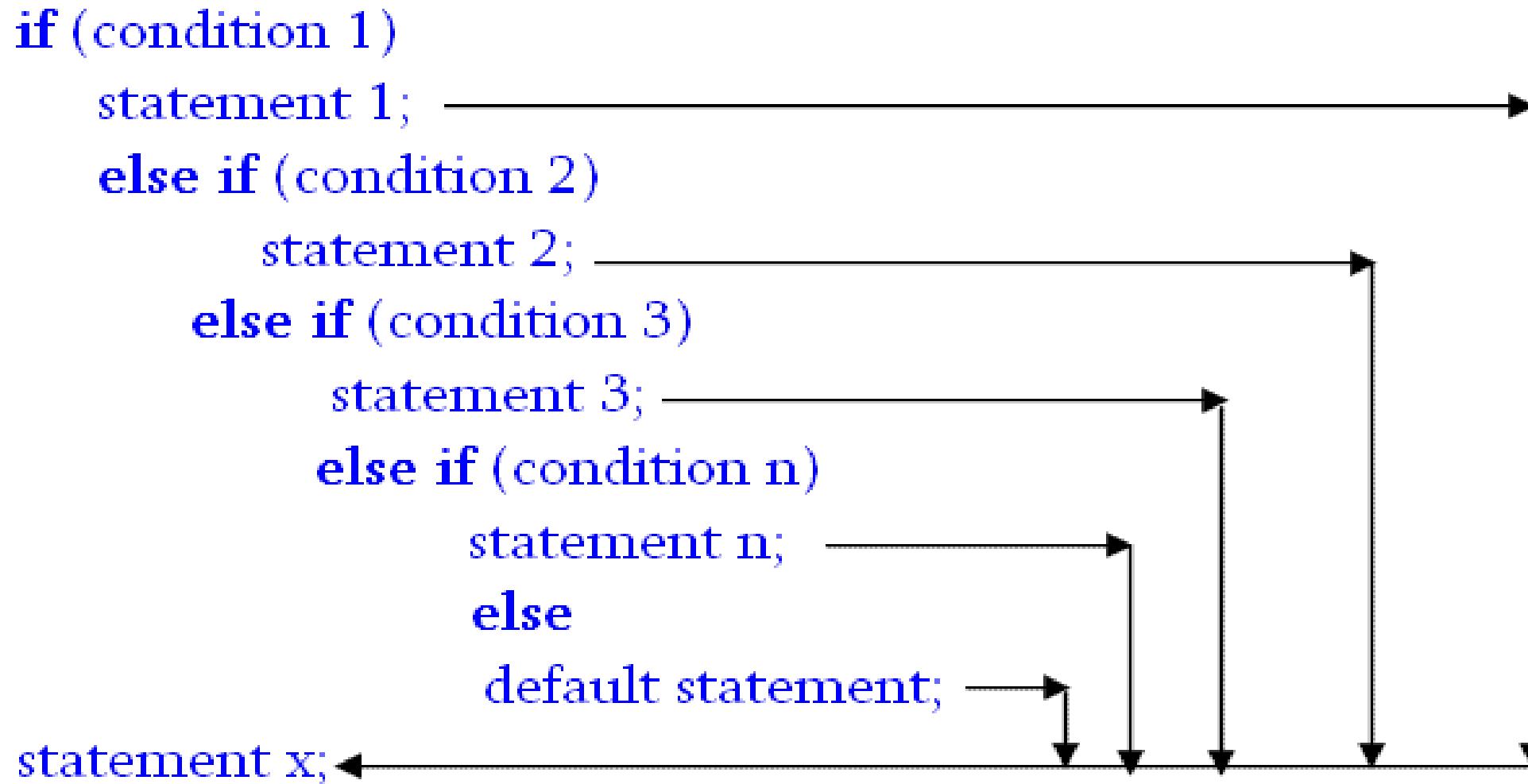
If else statement

```
if (condition)
{
    // block of code to be executed if the condition is true
} else
{
    // block of code to be executed if the condition is false
}
```

Nesting of if-else Statements



The else if Ladder



The switch Statement

- Switch is multiple–branching statement- based on a condition, the control is transferred to one of the many possible points.
- The most flexible control statement in selection structure of program control.
- Enables the program to execute different statements based on an expression that can have more than two values. Also called multiple choice statements.

General form:

```
switch(expression)
{ case value_1 : statement(s);
    break;
  case value_2 : statement(s);
    break; ... case value_n : statement(s);
    break;
  default : statement(s); }
    next_statement;
```

switch- example

```
index=mark/10;
switch (index)
{
case 10:
case 9:
case 8: grade='A';
          break;
case 7:
case 6:
          grade='B';
          break;
case 5:
          grade='C'
          break;
case 4:
          grade='D'
          break;
default: grade='F';
          break;
} cout<<grade;
```

DECISION MAKING AND LOOPING CONTROL STRUCTURES

- Iterative (repetitive) control structures are used to **repeat certain statements for a specified number of times**.
- The statements are executed as long as the **condition is true**
- These kind of control structures are also called as **loop control structures**
- Three kinds of loop control structures:
 - **while**
 - **do while**
 - **for**

While statement

Basic format:

while (test condition)

{

body of the loop

}

- **Entry controlled** loop statement
- Test condition is evaluated & if it is true, then body of the loop is executed.
- After execution, the test condition is again evaluated & if it is true, the body is executed again.
- This is **repeated until the test condition becomes false**, & control transferred out of the loop.
- **Body of loop may not be executed if the condition is false at the very first attempt.**

The do statement

General form:

```
do
{
    body of the loop
}
while (test condition);
```

The for statement

The general form:

for (initialization; test condition; increment)

{

Body of the loop

}

Next statement;

Nesting of for loop

One **for statement** within another **for statement**.

```
for (i=0; i< m; ++i)
```

```
{.....
```

```
....
```

```
for (j=0; j < n; ++j)
```

```
{.....
```

```
.....
```

```
}
```

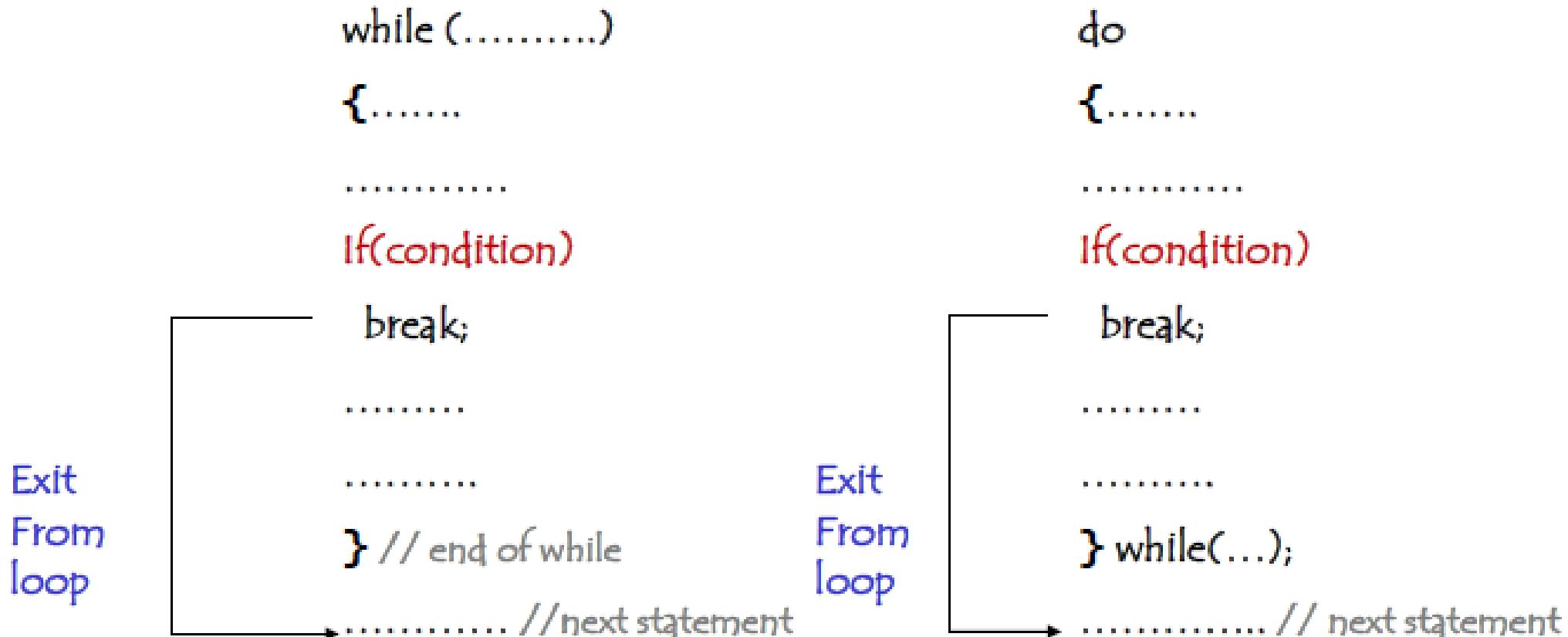
 // end of inner 'for' statement

```
// end of outer 'for' statement
```

Jumping out of a loop

- An early exit from a loop can be accomplished by using the **break** statement.
- When the break statement is encountered inside a loop, the loop is immediately exited & the program continues with the statement immediately following the loop. When the loops are nested , the break would only exit from the loop containing it.
i.e., the break will exit only a single loop.

Exiting a loop with break statement



Skipping a part of loop

- Skip part of the body of loop under certain conditions using continue statement.
- As the name implies, causes the loop to be continued with next iteration, after skipping rest of the body of the loop.

```
→ while (.....)
  {
    .....
    .....
    If(condition)
      continue;
    .....
    .....
  }
```

```
→ do
  {
    .....
    .....
    If(condition)
      continue;
    .....
    .....
  } while(...);
```

Tutorial

- Check if a given number is prime or not
- Factorial of given 10 numbers(do not use arrays)
- Print all odd numbers between m and n
- Menu driven program to sum all elements entered upto -1
- Find $\sin(x)$ using series
- Find $\cos(x)$ using series
- Find e^x using series
- Print triangle in the following form using loops until n.
Ex. If n=6

1		
2	3	
4	5	6

Arrays

- An array is a group of related data items that share a common name.
- The array elements are placed in a contiguous memory locations.
- A particular value in an array is indicated by writing an integer number called index number or subscript in square brackets after the array name.
- The least value that an index can take in array is 0.

Array Declaration:

type name [size];

- where type is a valid data type (like int, float...)
- Name is a valid identifier & size specifies how many elements the array has to contain.
- Size field is always enclosed in square brackets [] and takes static values.

Example:

- **an array salary containing 5 elements is declared as follows**
int salary [5];

One Dimensional Array

- A linear list of fixed number of data items of same type.
- These items are accessed using the same name using a single subscript. E.g. salary [1], salary [4]
- A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one dimensional array.

- Initializing one-dimensional array (compile time)

type array-name [size]={list of values}

Type → basic data type

Array-name → name of the array.

Size → maximum number of elements and may be omitted.

List of values → values separated by commas.

- E.g. `int number[3] = { 0,0,0} or {0}` will declare the variable number as an array of size 3 and will assign 0 to each element

2 – D Arrays

- It is an ordered table of homogeneous elements.
- It can be imagined as a two dimensional table made of elements, all of them of a same uniform data type.
- It is generally referred to as **matrix**, of some rows and some columns.
- It is also called as a **two-subscripted variable**.

- For example

```
int marks[5][3];
float matrix[3][3];
char page[25][80];
```

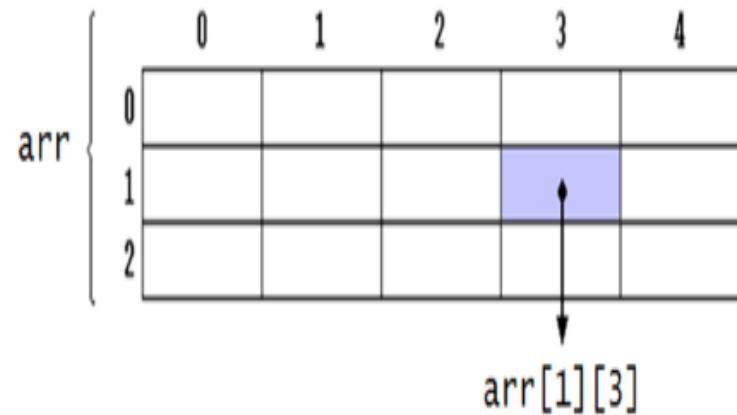
- The first example tells that marks is a 2-D array of 5 rows and 3 columns.
- The second example tells that matrix is a 2-D array of 3 rows and 3 columns.
- Similarly, the third example tells that page is a 2-D array of 25 rows and 80 columns.

- Declaration

type array_name[row_size][column_size];

- For example,

int arr [3][5];



`arr` represents a two dimensional array or table having 3 rows and 5 columns and it can store 15 integer values.

Read a matrix example

```
void main()
{
int i,j,m,n,a[100][100];
clrscr();
cout<<"enter dimension for a:";
cin>>m>>n;
cout<<"\n enter elements for a \n";
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
cin>>a[i][j];
}
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
cout<<"\t"<<a[i][j];
cout<<"\n";
}
getch();
}
```

Declaring a 3D array

- Specify data type, array name, block size, row size and column size.
- Each subscript can be written within its own separate pair of brackets.
- **Syntax: `data_type array_name[block_size][row_size][column_size];`**

Example:

```
int arr[2][3][3]; //array of type integer  
                    //number of blocks of 2D arrays:2 | rows:3 | columns:3  
                    //number of elements:2*3*3=18
```

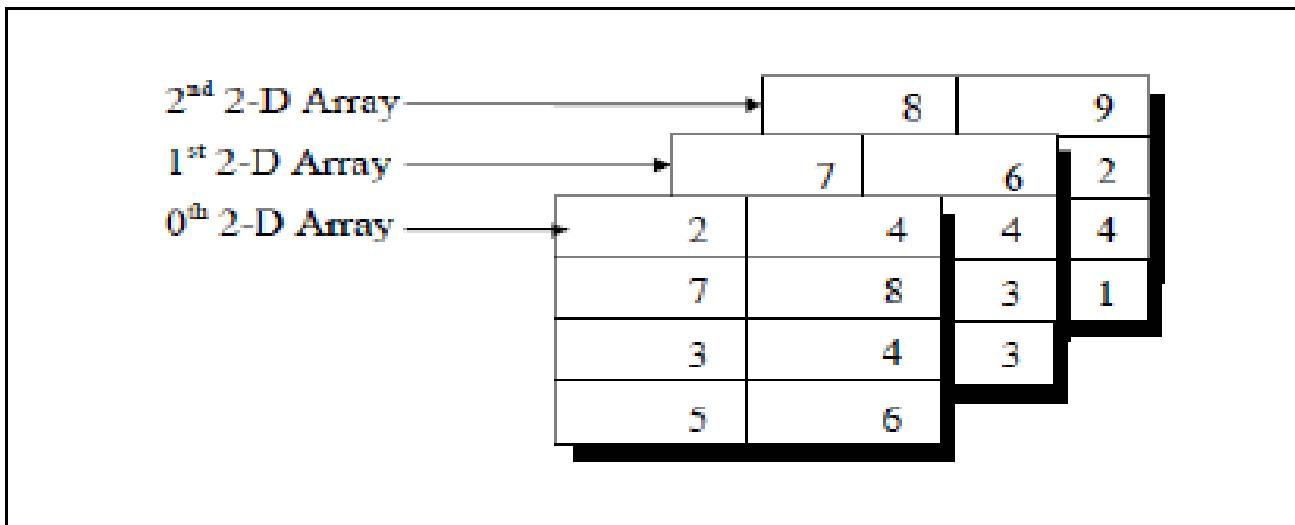
Output:

block(1) 11 22 33	block(2) 12 13 14
44 55 66	21 31 41
77 88 99	12 13 14
3x3	3x3

Multi – dimensional arrays

```
int arr[3][4][2]= {  
    {{ 2, 4 }, { 7, 8 }, { 3, 4 }, { 5, 6 } },  
    {{ 7, 6 }, { 3, 4 }, { 5, 3 }, { 2, 3 } },  
    {{ 8, 9 }, { 7, 2 }, { 3, 4 }, { 5, 1 }, }  
};
```

A three-dimensional array can be thought of as an array of arrays of arrays.



Conceptual
View



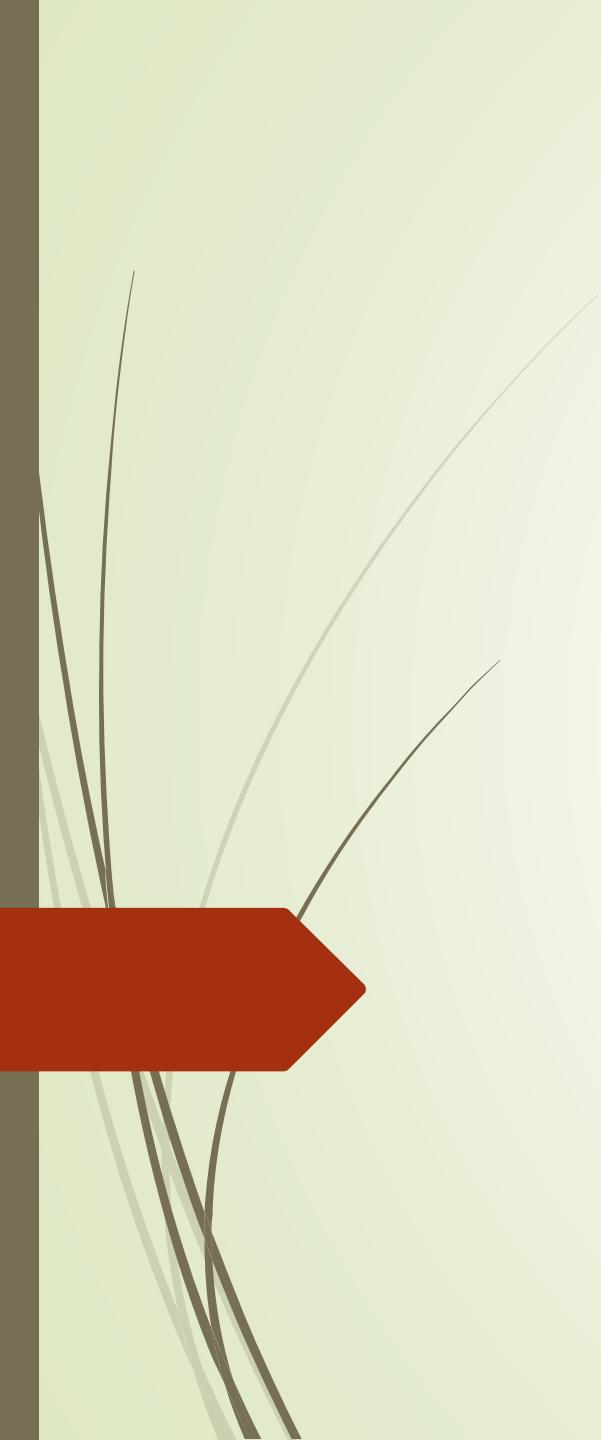
PROGRAMMING AND DATA STRUCTURES



Introduction to three dimensional
Arrays



STRINGS



String Definition

- A string is an array of characters.
- Any group of characters (except double quote sign) defined between double quotation marks is a constant string.
- Character strings are often used to build meaningful and readable programs.
- The common operations performed on strings are
 - Reading and writing strings
 - Combining strings together
 - Copying one string to another
 - Comparing strings to another
 - Extracting a portion of a string ..etc.

- 
- ▶ Declaration and initialization

char string_name[size];

The size determines the number of characters in the string_name.

- ✓ The character sequences "**Hello**" and "**Merry Christmas**" represented in an array **name** as follows :

name

H	e	l	l	o	\0														
---	---	---	---	---	----	--	--	--	--	--	--	--	--	--	--	--	--	--	--

M	e	r	r	y		C	h	r	i	s	t	m	a	s	\0			
---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	----	--	--	--

Example

```
#include <iostream.h>
void main ()
{
    char question[] = "Please, enter your first name: ";
    char greeting[] = "Hello, ";
    char yourname [80];
    cout << question;
    cin >> yourname;
    cout << greeting << yourname << "!";
    getch();
}
```

Reading Multiple Lines

- ▶ We use third argument in **cin.get()** function
- ▶ **cin.get(array_name, size, stop_char)**
- ▶ The argument **stop_char** specifies the character that tells the function to stop reading. The default value for this argument is the newline ('`\n`') character, but if you call the function with some other character for this argument, the default will be overridden by the specified character.

Example

```
#include <iostream>
using namespace std;
int main(){
    const int len = 80;           //max characters in string  char str[MAX];
                                //string variable str
    cout << "\nEnter a string:";
    char str[len];
    cin.get(str, len);
    cout << "\n" << str;
    return 0
}
```

Length of string

```
#include <iostream>
using namespace std;
int main(){
    char a[30];
    int i, c=0;
    cout<<"Enter a string:";
    cin.get(a,30);
    for(i=0;a[i]!='\0';i++)
        c++;
    cout<<"\nLength of the string "<<a<<" is "<<c;
    return 0;
}
```

String concatenation

```
#include<iostream>
using namespace std;
int main() {
    char str1[55],str2[25];
    int i=0,j=0;
    cout<<"\nEnter First String:";
    cin>>str1;
    cout<<"\nEnter Second String:";
    cin>>str2;
    while(str1[i]!='\0')
        i++;
```

```
while(str2[j]!='\0')
{
    str1[i]=str2[j];
    j++;
    i++;
}
str1[i]='\0';
cout<<"\nConcatenated String
is\n"<<str1;
return 0;}
```

Library functions: String Handling functions(built-in)

These in-built functions are used to manipulate a given string. These functions are part of **string.h** header file.

- **strlen ()**
✓ gives the length of the string. E.g. **strlen(string)**
- **strcpy ()**
✓ copies one string to other. E.g. **strcpy(Dstr1,Sstr2)**
- **strcmp ()**
✓ compares the two strings. E.g. **strcmp(str1,str2)**
- **strcat ()**
✓ Concatinate the two strings. E.g. **strcat(str1,str2)**

Library function: **strlen()**

- String length can be obtained by using the following function

n=strlen(string);

This function counts and returns the number of characters in a string, where n is an integer variable which receives the value of the length of the string. The argument may be a string constant.

- Copying a String the Hard Way

The best way to understand the true nature of strings is to deal with them character by character. The following program copies one string to another character by character.

Copies a string using a for loop

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[ ] = "Manipal Institute of Technology";
    char str2[100];
    int j;
    for(j=0 ; j<strlen(str1); j++)
        str2[j] = str1[j];
    str2[j] = '\0';
    cout << str1 << "\n" << str2 << endl;
    return 0;
}
```

Copies a string using a for loop

- ▶ The copying is done one character at a time, in the Statement
`str2[j] = str1[j];`
- ▶ The copied version of the string must be terminated with a null.
- ▶ However, the string length returned by **`strlen()`** does not include the null.
- ▶ We could copy one additional character, but it's safer to insert the null explicitly. We do this with the line **`str2[j] = '\0';`**
- ▶ If you don't insert this character, you'll find that the string printed by the program includes all sorts of weird characters following the string you want.
- ▶ The **<<** just keeps on printing characters, whatever they are, until by chance it encounters a '**`\0`**'.

Library function: **strcpy()**

Copying a String the Easy Way: **strcpy(destination, source)**

- The strcpy function works almost like a string assignment operator and assigns the contents of source to destination.
- ✓ Destination may be a character array variable or a string constant.
e.g., **strcpy(city,"DELHI");**
will assign the string “DELHI” to the string variable city.
- ✓ Similarly, the statement **strcpy(city1,city2);**
will assign the contents of the string variable city2 to the string variable city1.
Note:- The size of the array city1 should be large enough to receive the contents of city2.

strcpy(): Example

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[ ] = "Manipal Institute of Technology";
    char str2[100];
    strcpy(str2,str1);
    cout << str1 << "\n" << str2 << endl;
    return 0;
}
```

Library function: **strcmp()**

The **strcmp** function compares two strings identified by the arguments and **has a value 0 if they are equal.**

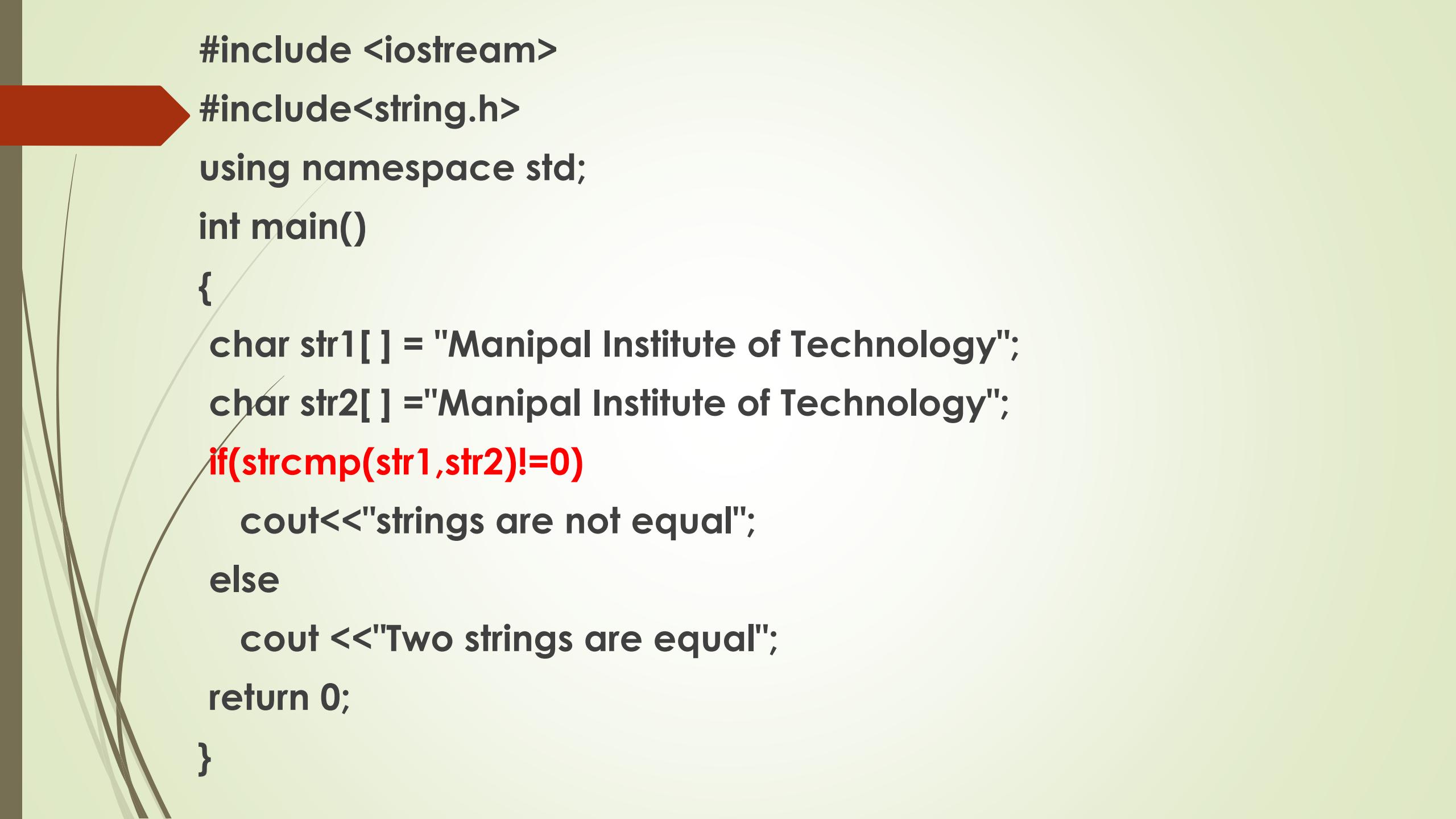
- If they are not, it has the numeric difference between the first non matching characters in the strings.

strcmp(string1,string2);

String1 and string2 may be string variables or string constants.

e.g., **strcmp("their","there");** will return a value of -9 which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" with respect to ASCII code is -9.

Note:- If the value is negative, string1 is alphabetically above string2.



```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[ ] = "Manipal Institute of Technology";
    char str2[ ] ="Manipal Institute of Technology";
    if(strcmp(str1,str2)!=0)
        cout<<"strings are not equal";
    else
        cout <<"Two strings are equal";
    return 0;
}
```

Library function: **strcat()**

The **strcat function** joins two strings together.

- It takes the following form:

► **strcat(string1,string2);**

- string1 and string2 are character arrays.

- ✓ When the function **strcat** is executed, string2 is appended to a string1.
- ✓ It does so **by removing the null character at the end of string1 and placing string2 from there.**
- ✓ The string at string2 remains unchanged.



```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
    char str1[ ] = "Manipal";
    char str2[] =" Institute of Technology";
    strcat(str1,str2);
    cout<<"concatenated string is \n"<<str1;
    return 0;
}
```

Reading integer followed by sentences

```
#include <iostream>

using namespace std;

//reading integer followed by sentences

int main(){

    int n;

    char s1[10],s2[10];

    cout<<"Enter integer";

    cin>>n;

    fflush(stdin);

    gets(s1);

    fflush(stdin);

    gets(s2);

    cout<<"s1="<<s1<<endl;

    cout<<"s2="<<s2<<endl;

    return 0;
}
```



Reading integer followed by multiline input strings

```
#include <iostream>
using namespace std;

int main(){
    int n;    char s1[10],s2[10];
    cout<<"Enter integer";
    cin>>n;
    fflush(stdin);
    cin.get(s1,10,'$');
    fflush(stdin);
    cin.get(s2,10,'$');
    cout<<"s1="<<s1<<endl;
    cout<<"s2="<<s2<<endl;
    return 0;
}
```

Insert a substring into a given string:

Steps to be followed:

- 1. Take a string, substring and position where the substring is to be inserted in main string as input.
- 2. Copy all the characters from the given position to a temporary array.
- 3. Substring should be inserted at the given position character by character to main string.
- 4. Now to this array copy all characters from the temporary array.

Delete substring

1. Take a string and its substring as input.
2. Take two control variable “i” for main string and “j” for substring
3. Compare $m[i]$ and $s[j]$.
 - If match is found increment both control variables.
 - If match is not found put $m[i]$ into the new array $n[k]$ and increment i .
 - Keep a flag variable to ensure that complete substring is present in the main string
4. If the end of substring is reached update the index j to 0 and again compare $m[i]$ with $s[j]$ and so on....



THANK YOU

Modular Programming

Lengthier programs

- Prone to errors
- tedious to locate and correct the errors

To overcome this

Programs broken into a number of smaller logical components, each of which serves a specific task.

Modularization

- ◆ Process of splitting the lengthier and complex programs into a number of smaller units is called **Modularization**.
- ◆ Programming with such an approach is called **Modular programming**

Advantages of modularization

- Reusability
- Debugging is easier
- Build library

Functions

- ◆ A *function* is a set of instructions to carryout a particular task.
- ◆ Using functions we can structure our programs in a more modular way.

Functions

- ◆ Standard functions
(library functions or built in functions)
- ◆ User-defined functions
Written by the user(programmer)

Defining a Function

✓ Name

- You should give functions descriptive names
- Same rules as variable names, generally

✓ Return type

- Data type of the value returned to the part of the program that activated (called) the function.

Functions

✓ Parameter list

- A list of variables that hold the values being passed to the function

✓ Body

- Statements enclosed in curly braces that perform the function's operations(tasks)

The general form of a function definition

```
return_type function_name(parameter_definition)
{
    variable declaration;
    statement1;
    statement2;
    .
    .
    .
    .
    .
    return(value_computed);
}
```

Functions: understanding

The diagram illustrates the structure of a C++ main function. It shows the declaration `void main (void)` with three parts labeled: "Return type" pointing to `void`, "Function name" pointing to `main`, and "Parameter List" pointing to `(void)`. Below this, the function body is shown as `{ cout << "hello world\n"; }`. A large brace on the right side groups the code block and is labeled "Body".

```
void main (void){    cout << "hello world\n";}
```

Functions

```
Return type      Function name      Parameter List  
void DisplayMessage(void)  
{   cout << "Hello from the function"  
    << "DisplayMessage.\n";  
}  
} Body
```

```
void main()  
{ cout << "Hello from main";  
    DisplayMessage(); // FUNCTION CALL  
    cout << "Back in function main again.\n";  
}
```

Functions

```
Return type      Function name      Parameter List  
void DisplayMessage(void)  
{ cout << "Hello from the function"  
  << "DisplayMessage.\n";  
}
```

} Body

```
void main()  
{ cout << "Hello from main";  
  DisplayMessage(); // FUNCTION CALL  
  cout << "Back in function main again.\n";  
}
```

Functions

```
Return type      Function name      Parameter List  
void DisplayMessage(void)  
{   cout << "Hello from the function"  
    << "DisplayMessage.\n";  
}  
} Body
```

```
void main()  
{ cout << "Hello from main";  
  DisplayMessage(); // FUNCTION CALL  
  cout << "Back in function main again.\n";  
}
```

Return type Function name Parameter List

```
void DisplayMessage(void)
{ cout << "Hello from the function"
"DisplayMessage.\n";
}
```

} Body <<

```
void main()
{ cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
}
```

Return type Function name Parameter List

→ **void DisplayMessage(void)**

```
{ cout << "Hello from the function"  
  << "DisplayMessage.\n";  
}
```

} Body

```
void main()  
{ cout << "Hello from main";  
  DisplayMessage(); // FUNCTION CALL  
  cout << "Back in function main again.\n";  
}
```

Return type Function name Parameter List

→ void DisplayMessage(void)

```
{ cout << "Hello from the function"  
     << "DisplayMessage.\n"; }
```

}

void main()

```
{ cout << "Hello from main";  
    DisplayMessage(); // FUNCTION CALL  
    cout << "Back in function main again.\n"; }
```

The diagram illustrates the structure of a C++ function definition. A blue arrow points from the text "Return type" to the keyword "void" in the function declaration. Another blue arrow points from "Function name" to "DisplayMessage". A third blue arrow points from "Parameter List" to the parentheses "()" in the declaration. To the right of the function body, a blue brace groups the code block, labeled "Body".

```
void DisplayMessage(void)
{ cout << "Hello from the function"
  << "DisplayMessage.\n";
}
```

```
void main()
{ cout << "Hello from main";
  DisplayMessage(); // FUNCTION CALL
  cout << "Back in function main again.\n";
}
```

The diagram illustrates the structure of a C++ function definition. At the top, three labels with arrows point to specific parts of the code: "Return type" points to "void", "Function name" points to "DisplayMessage", and "Parameter List" points to "(void)". Below this, the full function definition is shown:

```
void DisplayMessage(void)
{ cout << "Hello from the function"
  << "DisplayMessage.\n";
}
```

A large curly brace on the right side of the code is labeled "Body". A blue bracket on the left side of the code, starting from the opening brace of the main function and ending at the closing brace of the main function body, encloses the entire function definition.

Below the function definition, the main function is shown:

```
void main()
{ cout << "Hello from main";
  DisplayMessage(); // FUNCTION CALL
  cout << "Back in function main again.\n";
}
```

The line "DisplayMessage(); // FUNCTION CALL" is highlighted with a purple rectangular background.

```
// FUNCTION DEFINITION
```

```
void DisplayMessage(void)
```

```
{   cout << "Hello from the function"
```

```
      << "DisplayMessage.\n";
```

```
}
```

```
void main()
```

```
{ cout << "Hello from main";
```

```
    DisplayMessage(); // FUNCTION CALL
```

```
    cout << "Back in function main again.\n";
```

```
}
```

```
void DisplayMessage(void); //fn declaration
```

```
void main()
{ cout << "Hello from main";
  DisplayMessage(); // FUNCTION CALL
  cout << "Back in function main again.\n";
```

```
}
```

```
// FUNCTION DEFINITION
```

```
void DisplayMessage(void)
```

```
{   cout << "Hello from the function"
      << "DisplayMessage.\n";
```

```
}
```

```
→void First (void)
{ cout << "I am now inside function First\n"; }

→void Second (void)
{ cout << "I am now inside function Second\n";
  } // book has now as not →

void main ()
{ cout << "I am starting in function main\n";
  First ();
  Second (); ←
  cout << "Back in function main again.\n"; }
```

Function- definition & call

Return type Function name Parameter List

```
→ void DisplayMessage(void)
{   cout << "Hello from the function"
    << "DisplayMessage.\n";
}
```

}

Body

```
void main()
{ cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
}
```

Arguments (parameters)

- Both arguments (parameters) are variables used in a **program** & **function**.
- Variables used in the *function reference* or *function call* are called as **arguments**. These are written within the parenthesis followed by the name of the function. They are also called **actual parameters**.
- Variables used in *function definition* are called **parameters**, They are also referred to as **formal parameters**.

Home Work: To be solved ...Functions

Write appropriate functions to

1. Find the factorial of a number ‘n’.
2. Reverse a number ‘n’.
3. Check whether the number ‘n’ is a palindrome.
4. Generate the Fibonacci series for given limit ‘n’.
5. Check whether the number ‘n’ is prime.
6. Generate the prime series using the function written for prime check, for a given limit.

To be solved ... Functions

Factorial of a given number ‘n’

```
long factFn(int); //prototype
```

```
void main() {  
    long n, f;  
    cout<<"Enter the number to  
    evaluate its factorial:";  
    cin>>n;  
    f=factFn(n); //function call  
    cout<<"\nFact of "<<n<<" is "<< f;  
}
```

```
long factFn(int num) //factorial calculation  
{  
    int i, fact=1;  
  
    for (i=1; i<=num; i++)  
        fact=fact * i;  
  
    return (fact); //returning the factorial  
}
```

To be solved ... Functions

Try to convert all (non function) programs
into functions...

Function- definition & call

```
Formal parameters → void dispChar(int n, char c); //proto-type  
declaration  
void dispChar(int n, char c) // function  
definition  
{  
    cout<<" You have entered "<< n << "&" <<c;  
}  
void main(){ //calling program  
int no; char ch;  
cout<<"\nEnter a number & a character: \n";  
cin>>no>>ch;  
dispChar( no, ch); //Function reference  
}
```

Actual parameters

Functions- points to note

1. The parameter list must be separated by commas.

dispChar(int n, char c);

2. The parameter names do not need to be the same in the prototype declaration and the function definition.

3. The **types must match the types of parameters** in the function definition, **in number and order**.

void dispChar (int n, char c);//proto-type

void dispChar (int num, char letter) // function definition
{ cout<<" You have entered "<< n<< "&" <<c; }

4. Use of parameter names in the declaration is optional.

void dispChar (int , char);//proto-type

Functions- points to note

4. If the function has no formal parameters, the list is written as (void).
5. The return type is optional, when the function returns **int** type data.
6. The return type must be **void** if no value is returned.
7. When the declared types do not match with the types in the function definition, compiler will produce error.

Functions- Categories

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and one return value.
4. Functions with no arguments but return a value.
5. Functions that return multiple values (later).

Fn with No Arguments/parameters & No return values

```
void dispPattern(void )  
{ int i;  
    for (i=1;i<=20 ; i++)  
        cout << "*";}
```

```
void dispPattern(void); // prototype  
void main()  
{ cout << "fn to display a line of stars\n";  
    dispPattern();  
}
```

Fn with Arguments/parameters & No return values

```
void dispPattern(char c )  
{ int i;  
    for (i=1;i<=20 ; i++)  
        cout << c;  
    cout<<“\n”;}
```

```
void dispPattern(char ch); // prototype  
void main()  
{ cout << “\nfn to display a line of patterns\n”;  
    dispPattern('#');  
    dispPattern('*');  
    dispPattern('@'); }
```

Fn with Arguments/parameters & One return value

```
int fnAdd(int x, int y )  
{ int z;  
    z=x+y  
    return(z);}
```

```
int fnAdd(int a, int b); // prototype  
void main()  
{ int a,b,c;  
cout << “\nEnter numbers to be added\n”;  
cin>>a>>b;  
c=fnAdd(a,b);  
cout<<“Sum is “<< c;}
```

Fn with No Arguments but A return value

```
int readNum()  
{ int z;  
    cin>>z;  
    return(z); }
```

```
int readNum(void); // prototype  
void main(){  
    int c;  
    cout << “\nEnter a number \n”;  
    c=readNum();  
    cout<<“\nThe number read is “<<c;  
}
```

Fn that return multiple values

Will see later...

Passing 2D-Array to Function

Rules to pass a 2D- array to a function

- The function must be called by passing only the array name.
- In the function definition, we must indicate that the array has two-dimensions by including two set of brackets.
- The size of the second dimension must be specified.
- The prototype declaration should be similar to function header.

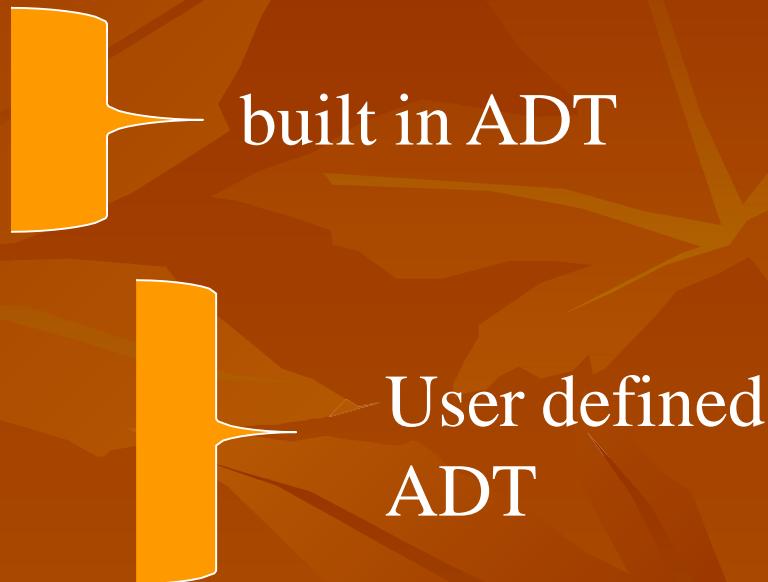
CLASS & OBJECT

- Class: A Class is a user defined data type to implement an abstract object. Abstract means to hide the details. A Class is a combination of data and functions.
- Data is called as data members and functions are called as member functions.

② Abstract data type:-

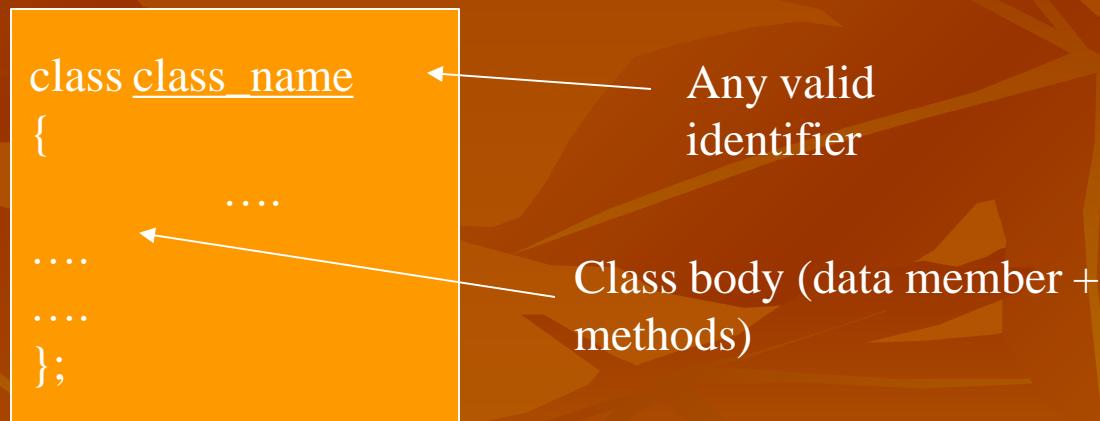
- ② A data type that separates the logical properties from the implementation details called Abstract Data Type(ADT).
- ② An abstract data type is a set of objects and an associated set of operations on those objects.
- ② ADT supports data abstraction, encapsulation and data hiding.

- Examples of ADT are:-
 - Boolean
 - Integer
 - Array
 - Stack
 - Queue
 - Tree search structure
- Boolean {operations are AND,OR,NOT and values are true and false }
- Queues{operations are create , dequeue,inqueue and values are queue elements }



Class definition

- A class definition begins with the keyword *class*.
- The body of the class is contained within a set of braces, { } ; (notice the semi-colon).



The diagram illustrates the structure of a class definition. It shows a code snippet within a yellow box:

```
class class_name
{
    ...
};
```

Annotations with arrows point to specific parts of the code:

- An arrow points to the identifier `class_name` with the label "Any valid identifier".
- An arrow points to the entire block between the braces with the label "Class body (data member + methods)".

- Within the body, the keywords *private*: and *public*: specify the access level of the members of the class.
 - the default is private.
- Usually, the data members of a class are declared in the *private*: section of the class and the member functions are in *public*: section.

- Data member or member functions may be public, private or protected.
- Public means data members or member functions defined inside the class can be used outside the class.(in different class and in main function)
- Member access specifiers
 - **public:** can be accessed outside the class directly.

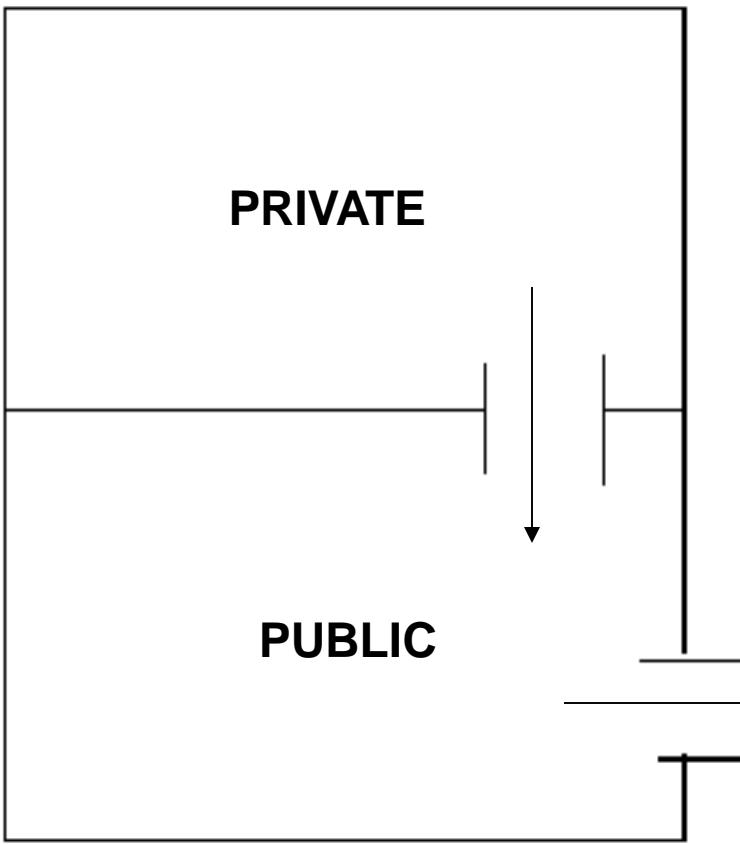
private:

- Accessible only to member functions of class
 - Private members and methods are for internal use only.
-
- Private means data members and member functions can't be used outside the class.

□ Protected means data member and member functions can be used in the same class and its derived class (at one level) (not in main function).

PRIVATE

PUBLIC



```
class class_name  
{  
    private:  
        ...  
        ...  
        ...  
    public:  
        ...  
        ...  
        ...  
};
```

private members or methods

Public members or methods

- This class example shows how we can encapsulate (gather) a circle information into one package (unit or class)

```
class Circle
{
    private:
        double radius;
    public:
        void setRadius(double r);
        double getDiameter();
        double getArea();
        double getCircumference();
};
```

No need for other classes to access and retrieve its value directly. The class methods are responsible for that only.

They are accessible from outside the class, and they can access the member (radius)

Class Example (Problem)

```
#include<iostream>
#include<stdio.h>
class student
{
    int rollno;
    char name[20];
};
```

```
int main()
{
    student s;
    cout<<"enter the rollno.:";
    cin>>s.rollno;
    cout<<"enter the name:";
    gets(s.name);
    cout<<"rollno:"<<s.rollno;
    cout<<"\nname:";
    puts(s.name);
    return 0;
}
```

Class Example (Solution)

```
#include<iostream>
#include<stdio.h>
class student
{
public:
    int rollno;
    char name[20];
};
```

```
int main()
{
    student s;
    cout<<"enter the rollno.:";
    cin>>s.rollno;
    cout<<"enter the name:";
    gets(s.name);
    cout<<"rollno:"<<s.rollno;
    cout<<"\nname:";
    puts(s.name);
    return 0;
}
```

Implementing class methods

- There are two ways:
 1. Member functions defined outside class
 - Using Binary scope resolution operator (::)
 - “Ties” member name to class name
 - Uniquely identify functions of particular class
 - Different classes can have member functions with same name
 - Format for defining member functions

```
ReturnType ClassName :: MemberFunctionName( ) {  
    ...  
}
```

Member Function

Defining Inside the Class

```
class student
{
    int rollno;
    char name[20];
public:
    void getdata()
    {
        cout<<"enter the rollno.:";
        cin>>rollno;
        cout<<"enter the name:";
        gets(name);
    }
    void putdata()
    {
        cout<<"rollno:"<<rollno;
        cout<<"\nname:";
        puts(name);
    }
};
```

Data Members (Private : in this example)

Member Functions (Public: in this example)

Calling member function

```
void main()
{
    student s;
    s.getdata();
    s.putdata();
}
```

Member Function

Defining Outside the Class

```
class student
{
int rollno;
char name[20];
public:
void getdata();
void putdata();
};

void student :: getdata()
{
cout<<"enter the rollno.:";
cin>>rollno;
cout<<"enter the name:";
gets(name);
}
```

```
void student :: putdata()
{
cout<<"rollno:"<<rollno;
cout<<"\nname:";
puts(name);
}

void main()
{
student s;
s.getdata();
s.putdata();
}
```

Characteristics of member function

- Different classes have same function name, the “membership label” will resolve their scope.
- Member functions can access the private data of the class. A non member function cannot do this.(friend function can do this.)
- A member function can call another member function directly, without using the dot operator.

Accessing Class Members

- ❑ Operators to access class members
 - ❑ Identical to those for **structs**
 - ❑ Dot member selection operator (.)
 - ❑ Object
 - ❑ Reference to object
 - ❑ Arrow member selection operator (->)
 - ❑ Pointers

Objects

- An object is an instance of a class.
- An object is a class variable.
- It can be uniquely identified by its name.
- Every object have a state which is represented by the values of its attributes. These state are changed by function which applied on the object.

Creating an object of a Class

- Declaring a variable of a class type creates an **object**. You can have many variables of the same type (class).
 - *Also known as Instantiation*
- Once an object of a certain class is instantiated, a new memory location is created for it to store its data members and code
- You can instantiate many objects from a class type.
 - Ex) Circle c; Circle *c;

Class item

```
{ .....,  
- -----  
}x,y,z;
```

We have to declare objects close to the place where they are needed because it makes easier to identify the objects.

Memory Allocation of Object

```
class student
```

```
{  
    int rollno;  
    char name[20];  
    int marks;  
};
```

```
student s;
```



Array of objects

The array of class type variable is known as array of object.

- ② We can declare array of object as following way:-
Class_name object [length];
Employee manager[3];

- 1. We can use this array when calling a member function
2. Manager[i].put data();
3. The array of object is stored in memory as a multi-dimensional array.

Object as function arguments

- This can be done in two ways:-
- A copy of entire object is passed to the function.
(pass by value)
- Only the address of the object is transferred to the function. (pass by reference)

(pass by value)

- A copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call function.

(pass by reference)

- When an address of object is passed, the called function works directly on the actual object used in the call. Means that any change made inside the function will reflect in the actual object.

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};

void Complex :: getdata( )
{
    cout<<"enter real part:" ;
    cin>>real;
    cout<<"enter imaginary part:" ;
    cin>>imag;
}

void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<"+ "<<imag<<"i";
    else
        cout<<real<<imag<<"i";
}
```

Passing Object

```
void Complex :: sum ( Complex A, Complex B)
{
    real = A.real + B.real;
    imag= A.imag + B.imag;
}

int main( )
{
    Complex X,Y,Z;
    X.getdata( );
    Y.getdata( );
    Z.sum(X,Y);
    Z.putdata( );
    return 0;
}
```

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>imag;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Passing Object

```
void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```



X

Y

Z

```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>ima g;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Passing Object

```
void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```

5
6

7
8

X

Y

Z

```

#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>ima g;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}

```

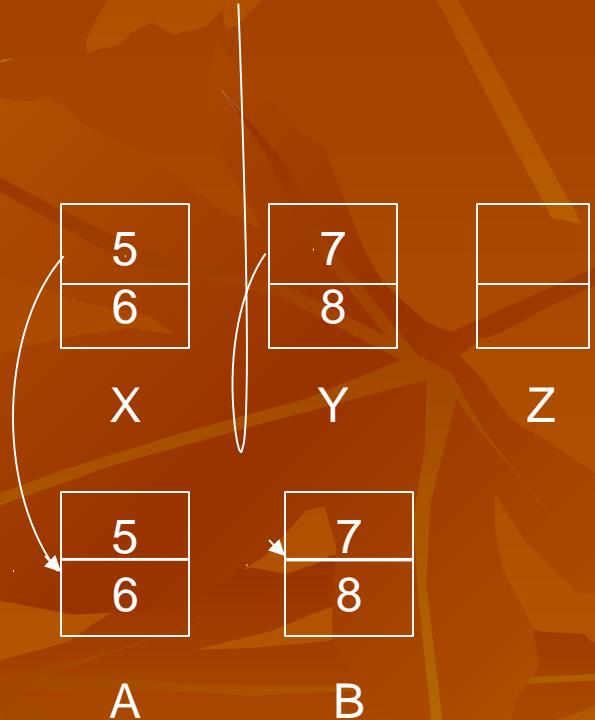
Passing Object

```

void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}

```



```

#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>ima g;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}

```

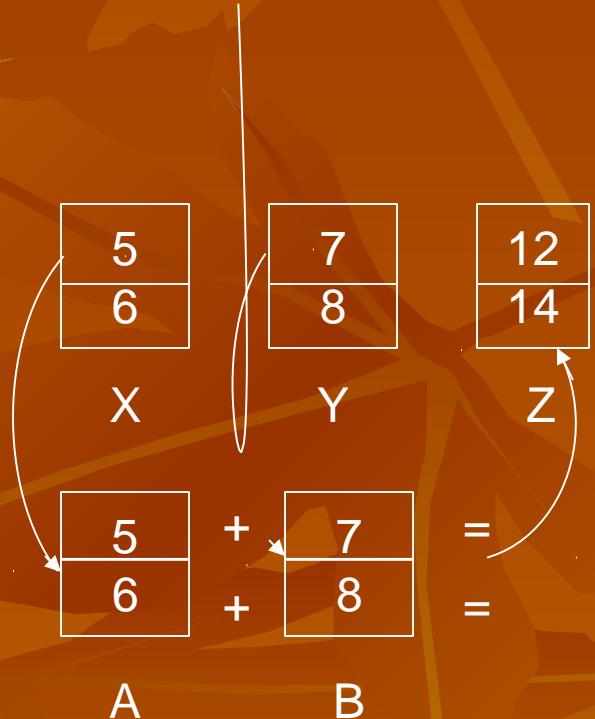
Passing Object

```

void Complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}

```



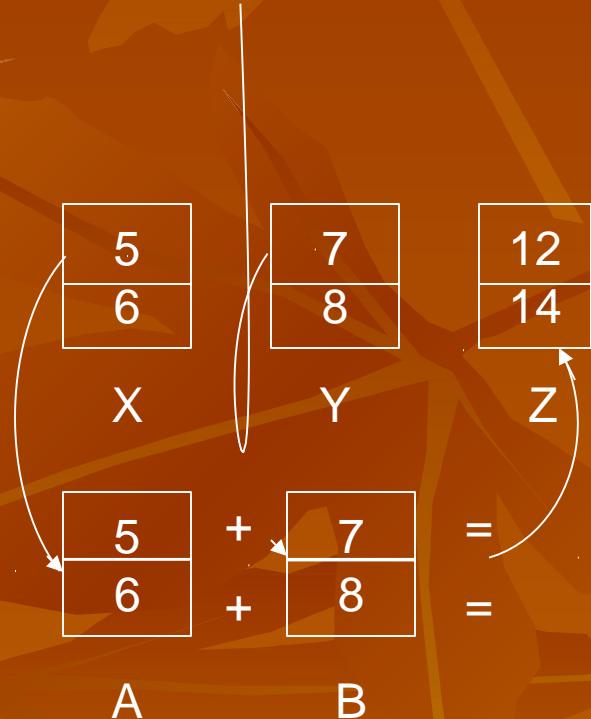
```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>imag;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Passing Object

```
void complex :: sum ( Complex A, Complex B)
{
real = A.real + B.real;
imag= A.imag + B.imag;
}

void main()
{
Complex X,Y,Z;
X.getdata();
Y.getdata();
Z.sum(X,Y);
Z.putdata();
}
```

$$12 + 14 i$$



```
#include<iostream.h>
class Complex
{
float real, imag;
public:
void getdata( );
void putdata( );
void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
cout<<“enter real part:”;
cin>>real;
cout<<“enter imaginary part:”;
cin>>ima g;
}
void Complex :: putdata( )
{
if (imag>=0)
cout<<real<<“+”<<imag<<“i”;
else
cout<<real<<imag<<“i”;
}
```

Returning Object

```
Complex Complex :: sum
(Complex B)
{
Complex temp;
temp.real=real + B.real;
temp.imag= imag + B.imag;
return temp;
}
void main ()
{
Complex X, Y, Z;
X.Getdata();
Y. getdata();
Z= X.sum (Y);
Z.putdata();
}
```

Returning Object

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<"enter real part:";
    cin>>real;
    cout<<"enter imaginary part:";
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<"+"<<imag<<"i";
    else
        cout<<real<<imag<<"i";
}
```

```
Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ( )
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}
```



X

Y

Z

Returning Object

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}
```

```
Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}
```

5
6

7
8

X

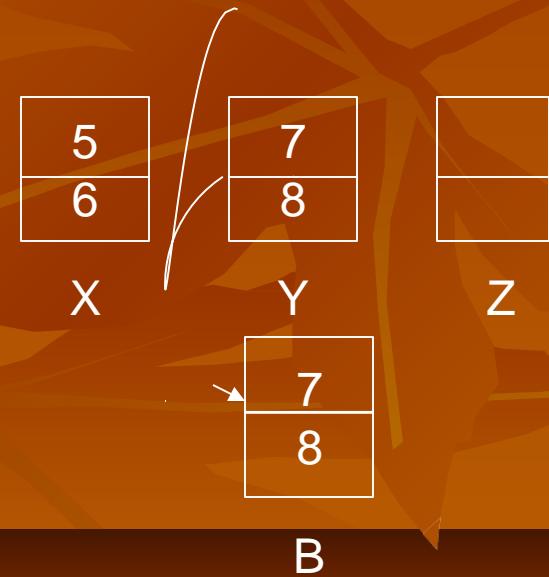
Y

Z

Returning Object

```
class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}
```

```
Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.Getdata();
    Y.getdata();
    Z= X.sum (Y);
    Z.putdata();
}
```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

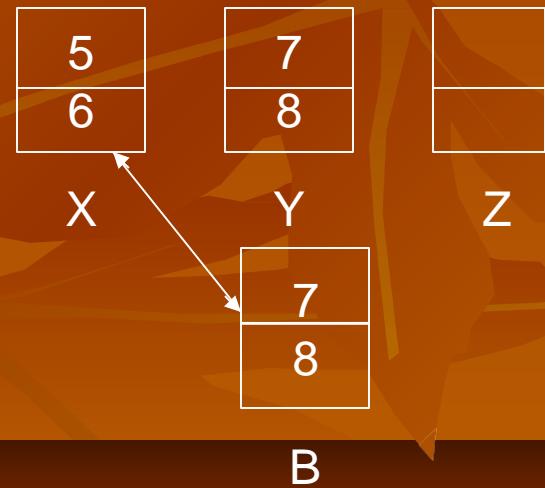
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

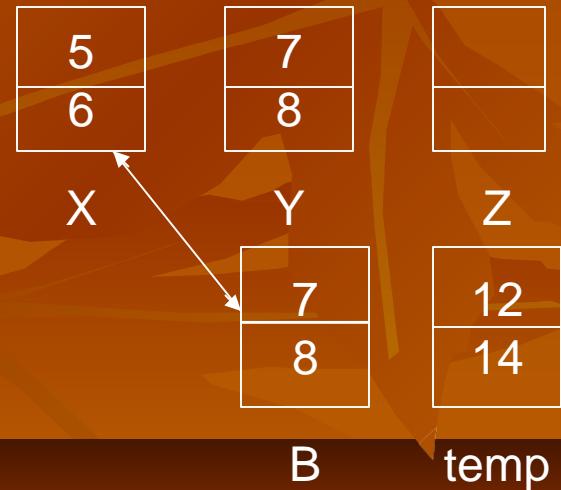
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

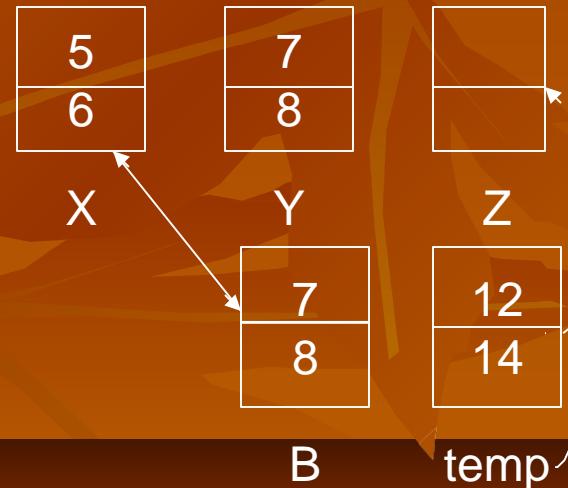
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

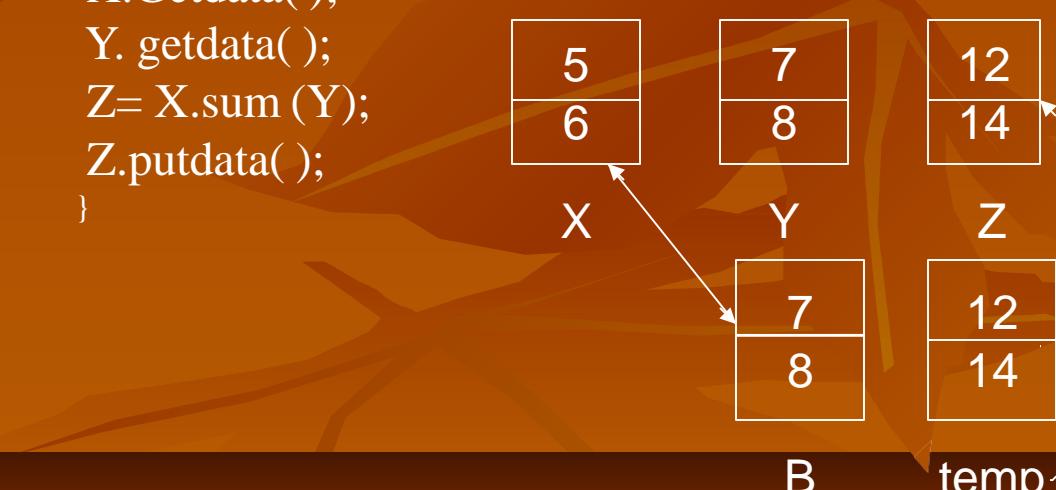
```

Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.Getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```



```

class Complex
{
    float real, imag;
public:
    void getdata( );
    void putdata( );
    void sum (Complex A, Complex B);
};
void Complex :: getdata( )
{
    cout<<“enter real part:”;
    cin>>real;
    cout<<“enter imaginary part:”;
    cin>>imag;
}
void Complex :: putdata( )
{
    if (imag>=0)
        cout<<real<<“+”<<imag<<“i”;
    else
        cout<<real<<imag<<“i”;
}

```

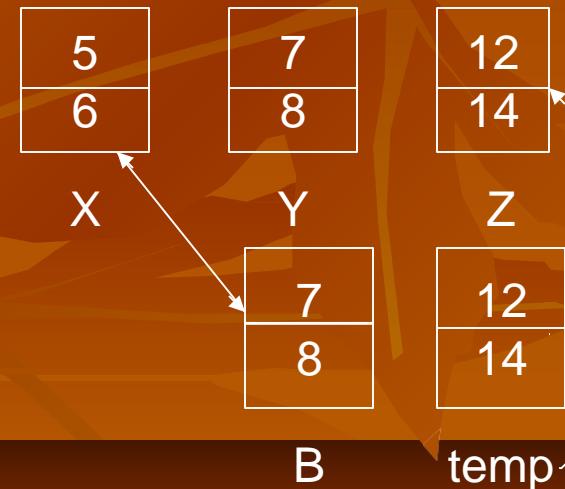
Returning Object

```

Complex Complex :: sum (Complex B)
{
    Complex temp;
    temp.real=real + B.real;
    temp.imag= imag + B.imag;
    return temp;
}
void main ()
{
    Complex X, Y, Z;
    X.Getdata( );
    Y.getdata( );
    Z= X.sum (Y);
    Z.putdata( );
}

```

$12 + 14i$





Constructors

and

Destructors

Constructor

- It is a member function which initializes the objects of its class.
- A constructor has:
 - (i) the same name as the class itself
 - (ii) no return type ,not even void.
- It constructs the values of data member so that it is called constructor.

- A constructor is called automatically whenever a new object of a class is created.
- You must supply the arguments to the constructor when a new object is created.
- If you do not specify a constructor, the compiler generates a default constructor for you (expects no parameters and has an empty body).

```
void main()
{
    rectangle rc(3.0, 2.0);

    rc.posn(100, 100);
    rc.draw();
    rc.move(50, 50);
    rc.draw();
}
```

□ *Warning:* attempting to initialize a data member of a class explicitly in the class definition is a syntax error.

Declaration and definition

class complex

{

 int m, n;

 public:

 complex();

};

complex :: complex ()

{

 m=0; n=0;

}

- A constructor that accepts no parameters is called default constructor.

characteristics

1. They should be declared in public section.
2. Invoked automatically when class objects are created.
3. They do not have return types, not even void and they can't return any value.
4. They cannot be inherited, though a derived class can call the base class constructors.

5. They also can have default arguments like other functions.
6. They are implicitly called when the NEW and DELETE operators execute when memory allocation is required.
7. Constructors can not be virtual.

Parameterized constructors

- The constructors that can take arguments are called parameterized constructors.
- It is used when we assign different value to the data member for different object.
- We must pass the initial values as arguments to the constructors when an object is declared.

- ② This can be done in two ways:-
 - ② By **calling the constructors implicitly**
 - ② Class_name object(arguments);
 - ② Ex:- simple s(3, 67);
 - ② This method also known as shorthand.
 - ② By **calling the constructors explicitly**
 - ② Class_name object =constructor(arguments);
 - ② Ex:- simple s=simple(2,67);
 - ② This statement create object s and passes the values 2 and 67 to it.

Example:-

```
#include<iostream.h>
class integer
{
    int m,n;
public:
    integer(int,int);
    void display()
    {
        cout<<"m"<<m;
        cout<<"n"<<n;
    }
    integer::integer(int x,int y)
    {
        m=x;
        n=y;
    }
}
```

```
int main()
{
    integer i1(10,100);
    integer i2=integer(33,55);
    cout<<"object 1";
    i1.display();
    cout<<"object 2";
    i2.display();
    return 0;
}
```

Notes:-

A constructor function can also be defined as **INLINE** function.

```
Class integer
{
    int m,n;
    public:
        integer (int x,int y)
    {
        m=x;
        n=y;
    };
```

A class can accept a reference of its own class as parameter.

```
Class A  
{
```

```
.....
```

```
.....  
Public:  
A(A&);  
};
```

is valid

In this case the constructor is called as copy constructor.

Copy constructor

- When a class reference is passed as parameters in constructor then that constructor is called Copy constructor.
- A copy constructor is used to declare and initialize an object from another object.

Syntax:-

Constructor _name (class_name & object); Integer (integer
&i);

```
Integer i2(i1); /integer i2=i1;
```

Define object i2 and initialize it with i1.

The process of initialization object through copy constructor is known as copy initialization.

A copy constructor takes a reference to an object of the same class as itself as argument.

```
#include<iostream.h>

Class person
{
    public: int
        age;
    Person(int a)
    { age = a; } Person(person
    & x)
    { age=x.age;
    }
};
```

```
int main()
{
    Person timmy(10); Person
    sally(15);
    Person timmy_clone = timmy;
    cout << timmy.age << " " << sally.age << " "
    << timmy_clone.age << endl;
    timmy.age = 23;
    cout << timmy.age << " " << sally.age << " "
    << timmy_clone.age << endl;
}
```

Destructors

- A destructor is used to destroy the objects that have been created by constructor.
- It is also a member function of class whose name same as class name but preceded by tiled sign(~).
- It never takes any arguments nor return any value
- It will be invoked implicitly by the compiler upon exit from the program to clean up the storage which is allocated

The new operator is used in constructor to allocate memory and delete is used to free in destructors.

Example:-

```
~assign()  
{  
    Delete p;  
}
```

```
#include<iostream.h>
Int count=0;
Class try
{
    public:
        try()
        {
            count++;
Cout<<“no of objects created”<<count;
}
~try()
{
    cout<<“no of object destroyed”<<count;
Count- -;
}};
```

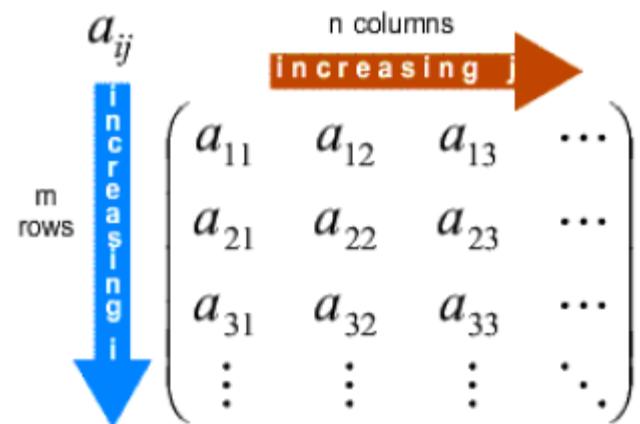
```
int main()
{
    cout<<“enter main”;
try t1,t2,t3,t4;
{
    cout<<“block1”;
try t5;
}
{
    cout<<“block 2”;
try t6;
}
cout<<“again in main”;
Return 0;
}
```

Special forms of square matrices- Sparse, Polynomial

2-Dimensional Arrays

Matrices

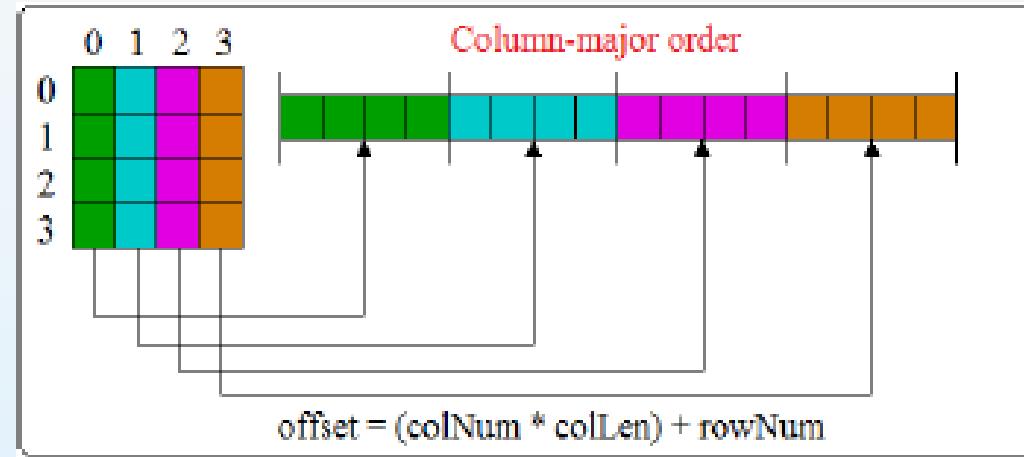
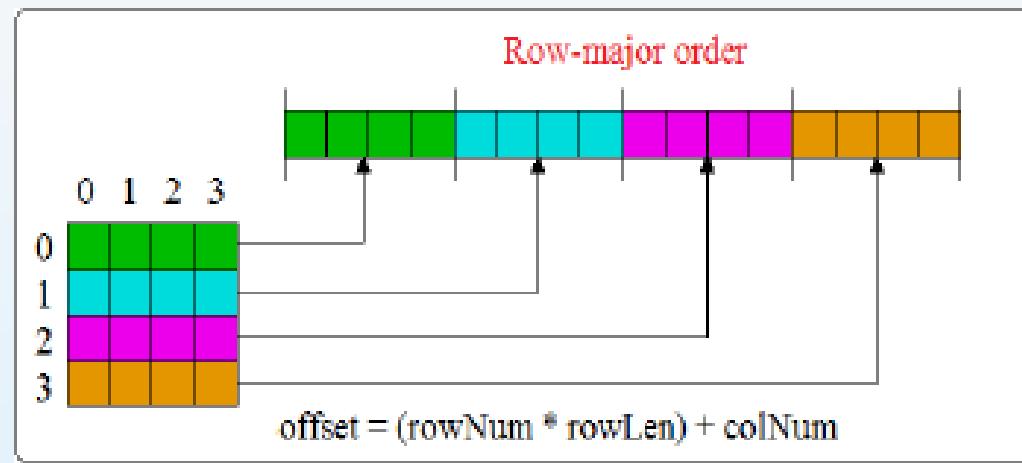
- A $m \times n$ matrix is a table with m rows and n columns (m and n are the dimensions of the matrix)



- 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

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

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

- ① Accept order of matrix r_{ow} , c_{ol}
- ② Accept matrix values
 $\{ \text{if } A[i][j] \neq 0 \} \text{ count++};$
- ③ Store sparse $[count];$
 $s[0] \cdot r = r_{ow}; s[0] \cdot c = c_{ol}, s[0] \cdot v = count;$
- ④ Create 1D array + nplets
- ⑤ Display " "
- ⑥ Reconstruct sparse matrix from 1D array of tuples
- ⑦ Display newly constructed

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)

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

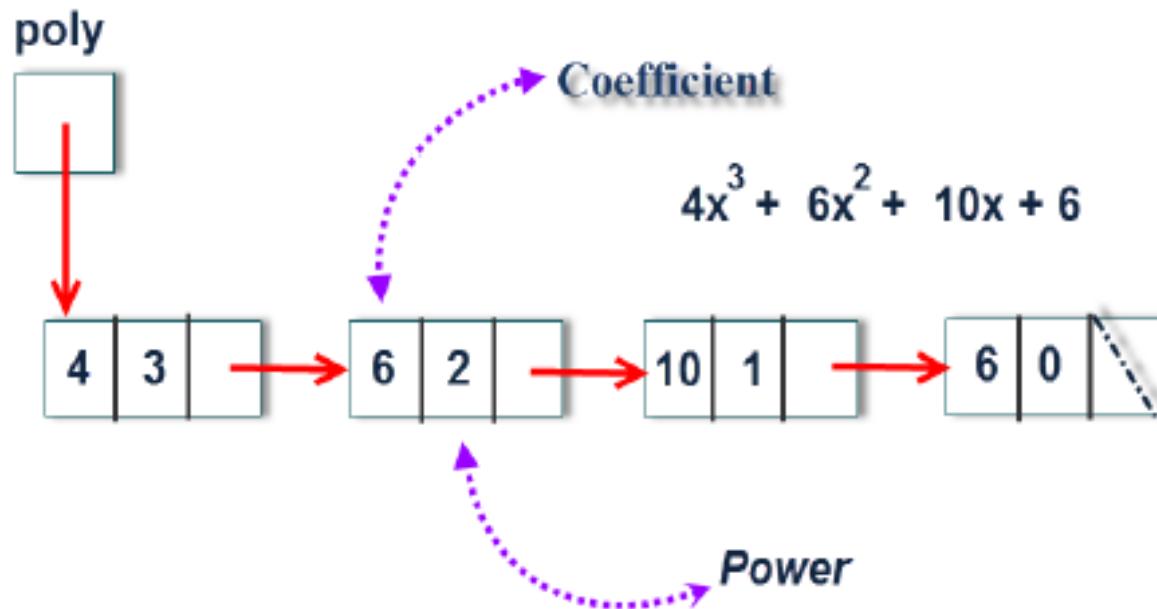
Coefficient

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



	col	exp
0	4	
1	3	6
2	4	4
3	3	2
4	2	0

P_1

	col	exp
0		
1		
2		
3		
4		

P_3

	col	exp
0	3	
1	5	5
2	2	4
3	3	0

$$P_1 = 3x^6 + 4x^4 + 3x^2 + 2$$

$$P_2 = 5x^5 + 2x^4 + 3x$$

```
//add polynomial as 1d array
#include <stdio.h>
typedef struct
{
    int coeff, exp;
} poly;
```

```
int main()
{
    poly p1[10],p2[10],p3[20];
    printf("1st polynomial:\n");
    getpoly(p1);
    printf("\n2nd polynomial:\n");
    getpoly(p2);
    printf("\n1st polynomial: ");
    showpoly(p1);
    printf("\n2nd polynomial: ");
    showpoly(p2);
    add(p1,p2,p3);
    printf("\nSum : ");
    showpoly(p3);
    printf("\n");
    return 0;
}
```

```
void getpoly(poly p[])
{
    int n,i;
    printf("Enter the number of terms: ");
    scanf("%d",&n);
    printf("Enter the terms (coefficient, exponent):\n");
    p[0].coeff=n;
    for(i=1;i<=n;i++)
        scanf("%d %d",&p[i].coeff,&p[i].exp);
}
void showpoly(poly p[])
{
    int i;
    for(i=1;i<=p[0].coeff;i++)
        printf("(%d,%d) ",p[i].coeff,p[i].exp);
}
```

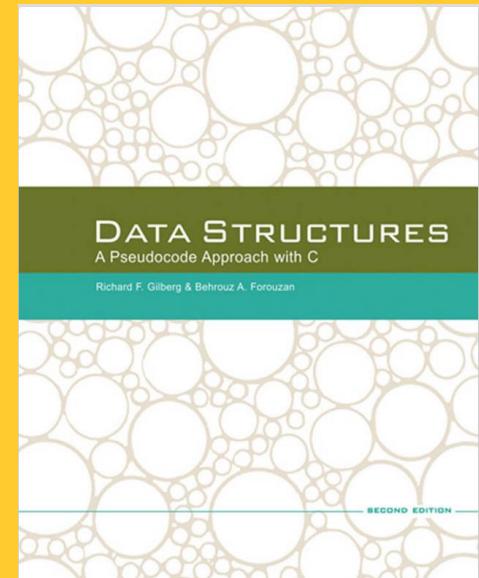
```
void add(poly p1[],poly p2[],poly p3[])
{
    int i=1,j=1,n1=p1[0].coeff,
        n2=p2[0].coeff,n3=0;
    while(i<=n1 && j<=n2)
    {
        if(p1[i].exp>p2[j].exp)
        {
            n3++;
            p3[n3].coeff=p1[i].coeff;
            p3[n3].exp=p1[i].exp;
            i++;
        }
        else if(p1[i].exp<p2[j].exp)
        {
            n3++;
            p3[n3].coeff=p2[j].coeff;
            p3[n3].exp=p2[j].exp;
            j++;
        }
    }
}
```

```
else {
    int sum=p1[i].coeff+p2[j].coeff;
    if(sum!=0)
    {
        n3++;
        p3[n3].coeff=sum;
        p3[n3].exp=p1[i].exp;
    }
    i++;
    j++;
}
```

```
while(i<=n1)
{
    n3++;
    p3[n3].coeff=p1[i].coeff;
    p3[n3].exp=p1[i].exp;
    i++;
}
while(j<=n2)
{
    n3++;
    p3[n3].coeff=p2[j].coeff;
    p3[n3].exp=p2[j].exp;
    j++;
}
p3[0].coeff=n3;
```

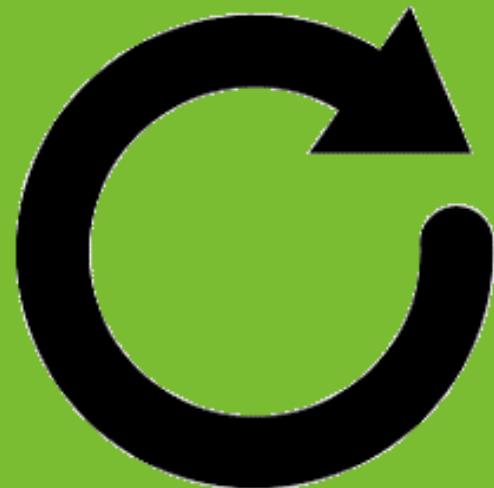
Review of

Recursion



How to write repetitive algorithms?

Two Approaches:



Iteration



Recursion

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

Recursion: What is it?

Case Study: Factorial Algorithm.

- 1. Iterative Solution**
- 2. Recursive Definition**
- 3. Recursive Solution**

Factorial: Iterative Definition.

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

$$5 = 5 \times 4 \times 3 \times 2 \times 1$$

$$\text{factorial}(4) = 4 \times 3 \times 2 \times 1 = 24$$

Factorial: Recursive Definition.

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

Factorial: Recursive Definition.

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

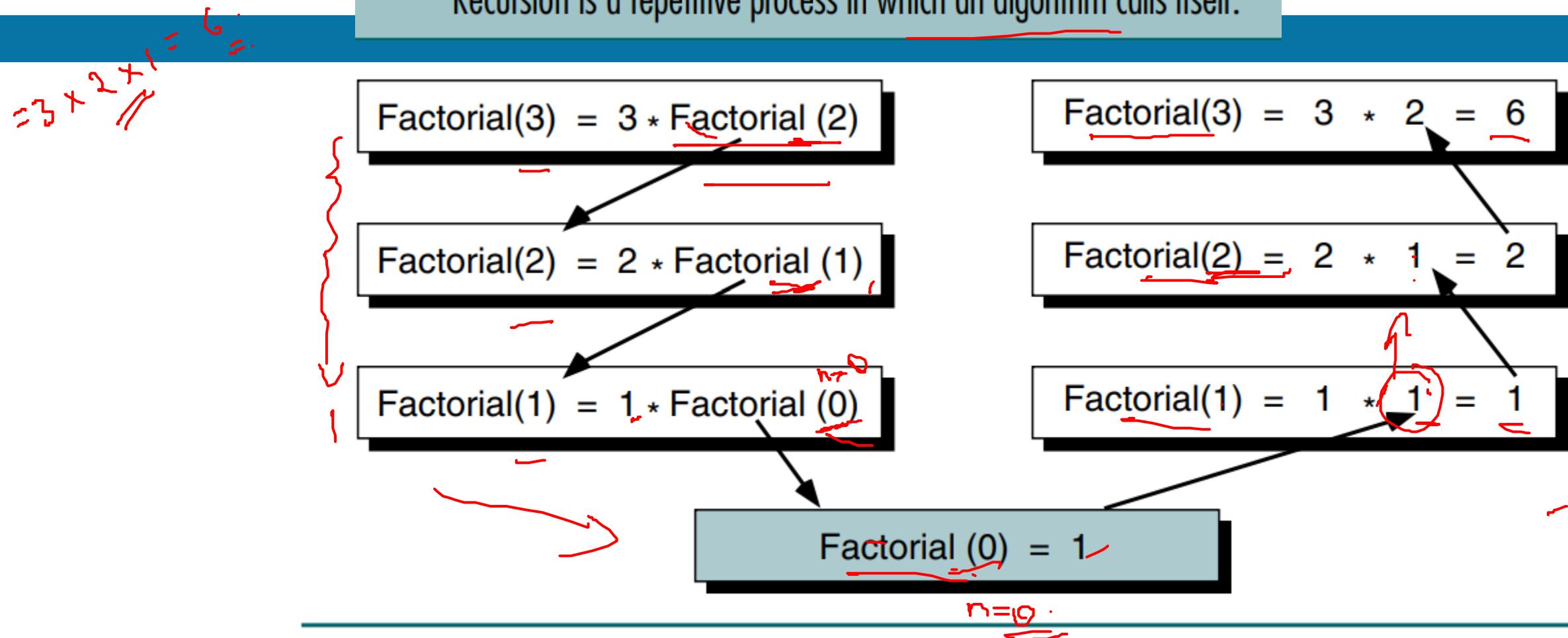


FIGURE 2-3 Factorial (3) Recursively

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
```

*K = 3, 2L ≈ 3
F = (x) - 1
Y*

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

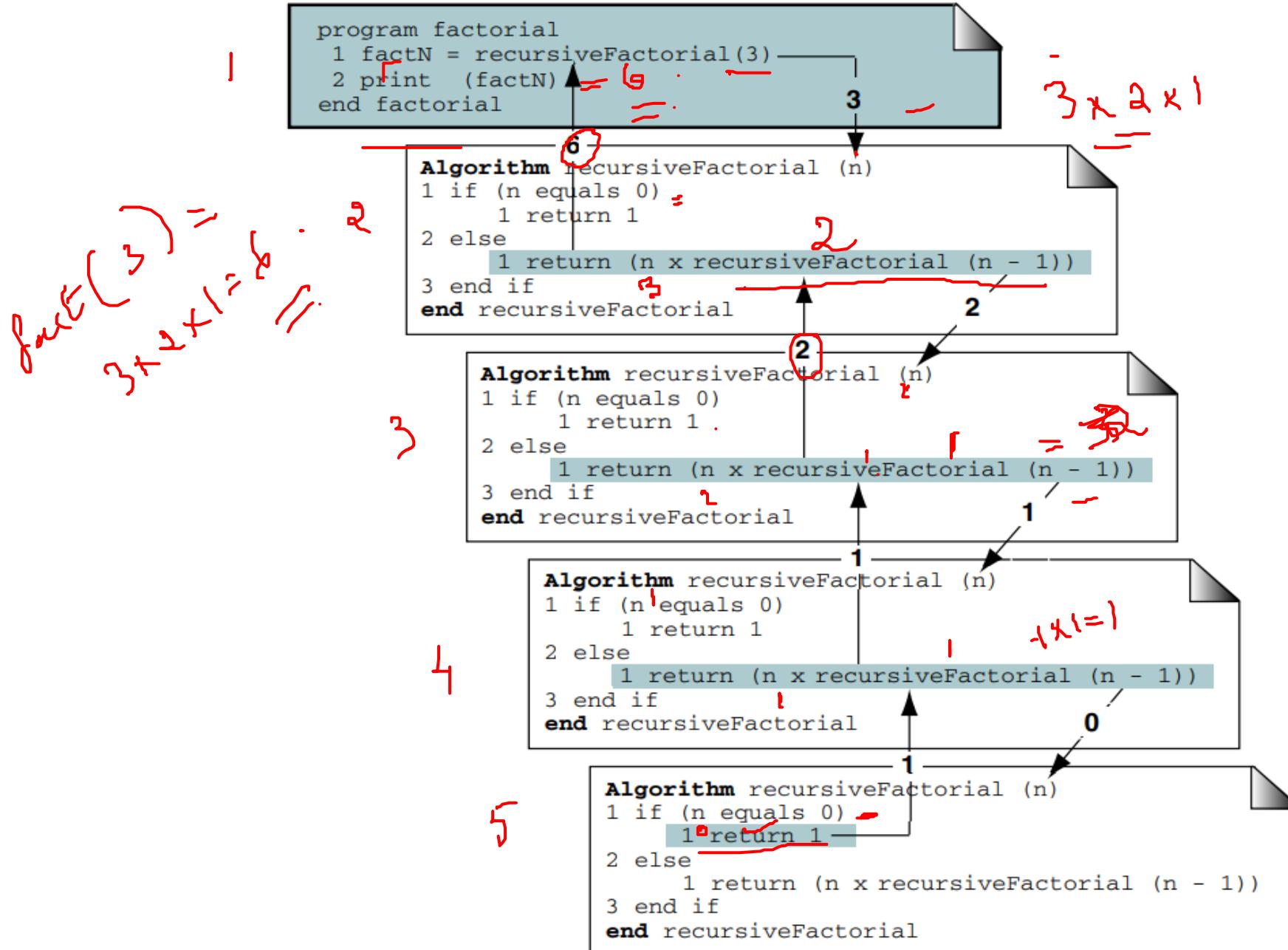


FIGURE 2-4 Calling a Recursive Algorithm

Figure 2-4 traces the recursion and the parameters for each individual call.

Designing Recursive Algorithms.

Analytic Approach:

1. The Design Methodology
2. Limitations of Recursion
3. Design Implementation

THE DESIGN METHODOLOGY.

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.

LIMITATIONS OF RECURSION.

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?

DESIGN IMPLEMENTATION.

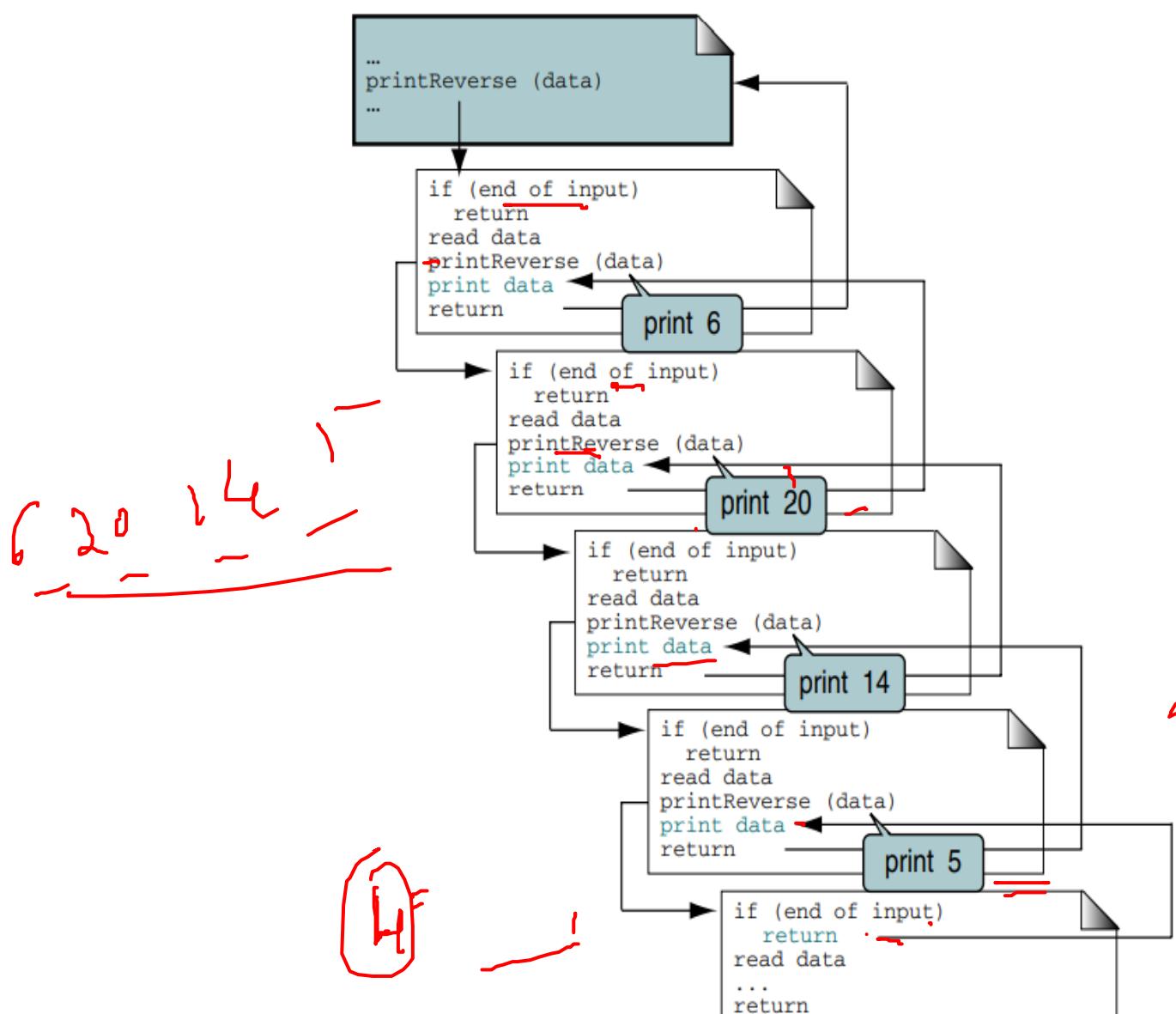
ALGORITHM 2-3 Print Reverse

```
Algorithm printReverse (data)
Print keyboard data in reverse.
Pre nothing
Post data printed in reverse
1 if (end of input) = 4
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
```

↓ | ↓ | 2 3 4 | → D C B A
O P Q 3 2 1 | → D C B A

↑ 3 2 1

DESIGN IMPLEMENTATION.



Recursive returns (prints)

6
data

20
data

14
data

5
data

FIGURE 2-5 Print Keyboard Input in Reverse

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

We use the **Euclidean algorithm** to determine the greatest common divisor between two nonnegative integers.

Given two integers, a and b , the greatest common divisor is recursively found using the formula given above.

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
```

$$\left\{ \begin{array}{l} \text{gcd}(10, 25) \\ \Rightarrow \text{gcd}(25, 10 \bmod 15) = \text{gcd}(25, 10) \\ \Rightarrow \text{gcd}(25, 10, 25 \bmod 10) = \text{gcd}(10, 5) \\ \Rightarrow \text{gcd}(10, 5, 10 \bmod 5) = \text{gcd}(5, 0) \\ b = 0 \quad a = 5 \end{array} \right.$$

GCD IMPLEMENTATION.

```
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) {BASE}
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

$l = 0, 1, 1, 2, 3 \dots$

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

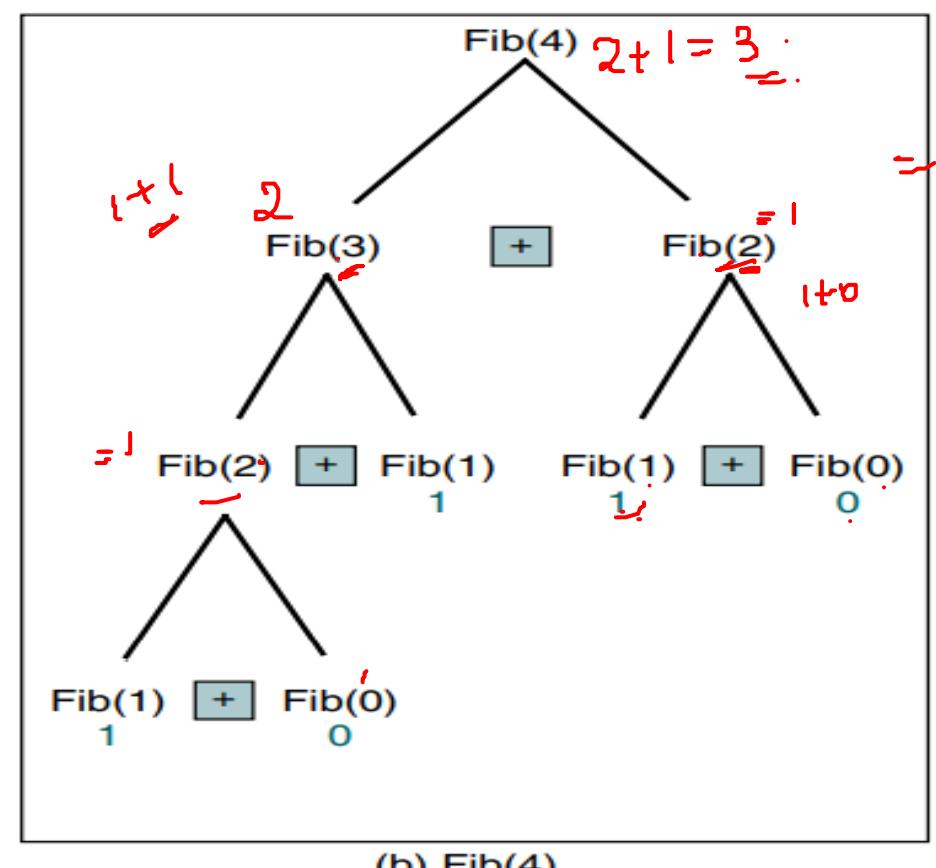
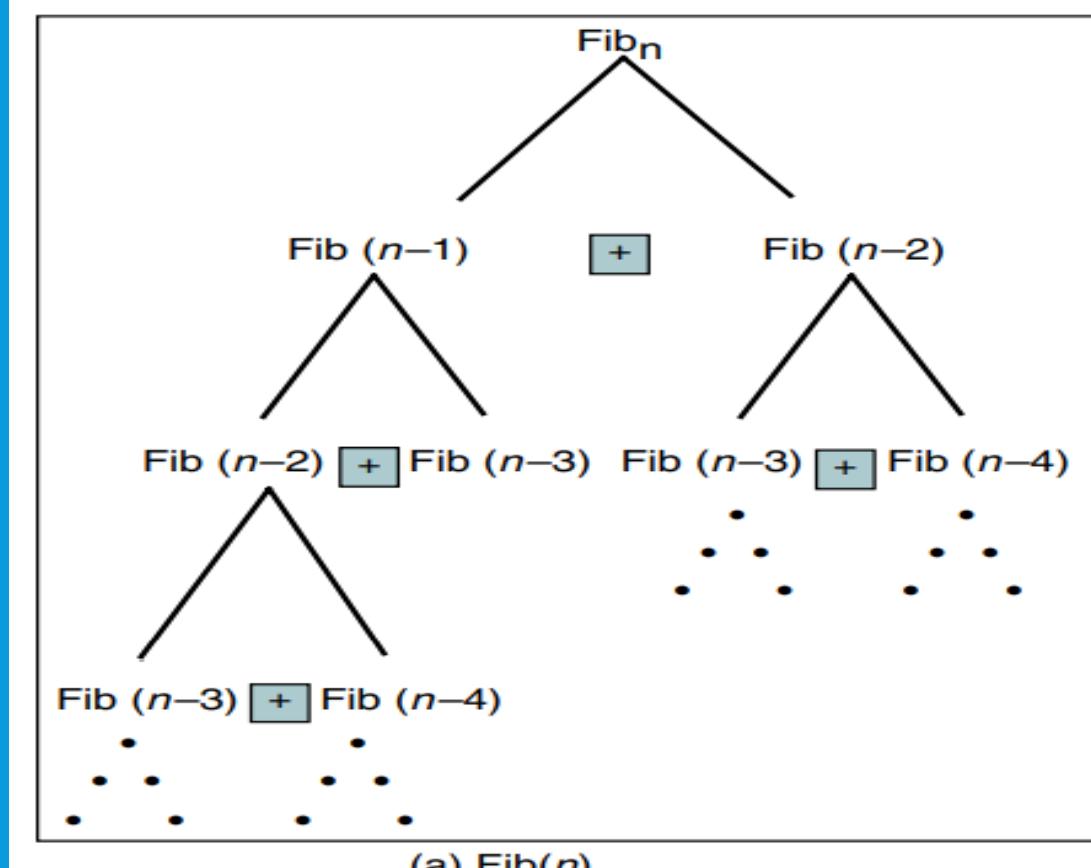
Fibonacci Numbers Recursive Definition

$\underbrace{\text{Fibonacci } (n - 1) + \text{Fibonacci } (n - 2)}_{\text{Recursive Call}} \quad \underline{n}$

Fibonacci numbers are named after **Leonardo Fibonacci**, an Italian mathematician who lived in the early thirteenth century. In this series each number is the sum of the previous two numbers. The first few numbers in the Fibonacci series are:

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$

FIBONACCI SERIES: DESIGN.



FIBONACCI NUMBERS IMPLEMENTATION.

```
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     } // fib
41     return (fib (num - 1) + fib (num - 2));
```

Results:

Print a Fibonacci series.

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

FIBONACCI NUMBERS IMPLEMENTATION CALLS.

$\text{fib}(n)$	Calls	$\text{fib}(n)$	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	no=40	331,160,281

Recursive program for multiplying 2 numbers

```
int mul(int m, int n) {  
    int y;  
    if(m==0 || n==0)  
        return 0;  
    if(n==1)  
        return m;  
    else{  
        y=mul(m, n-1);  
        return(y+m);  
    }  
}
```

$$0 + \dots = 0$$

$$1 \times \dots = 1$$

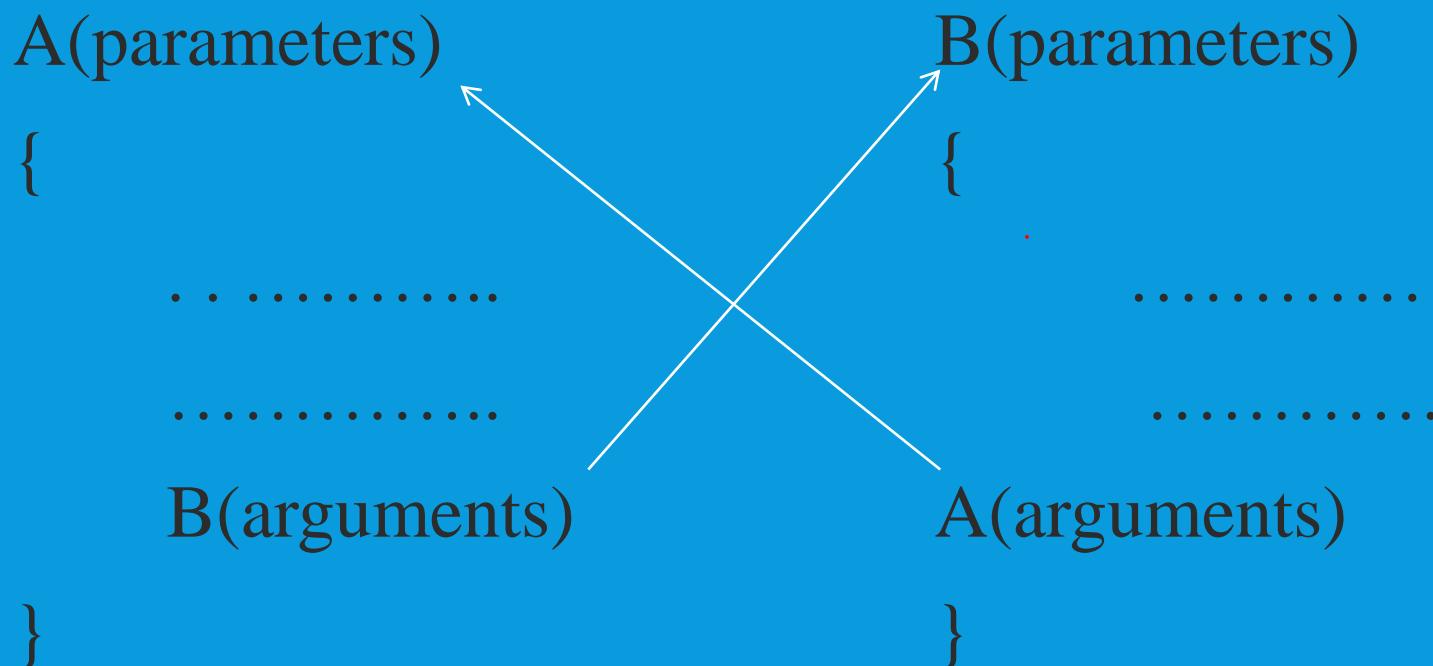


Recursive program to find the nth fibonacci number

```
int fib(int n)
{
    if(n==0)
        return 0;
    if(n==1)
        return 1;
    else
    {
        return (fib(n-1) + fib(n-2));
    }
}
```

Recursive chains

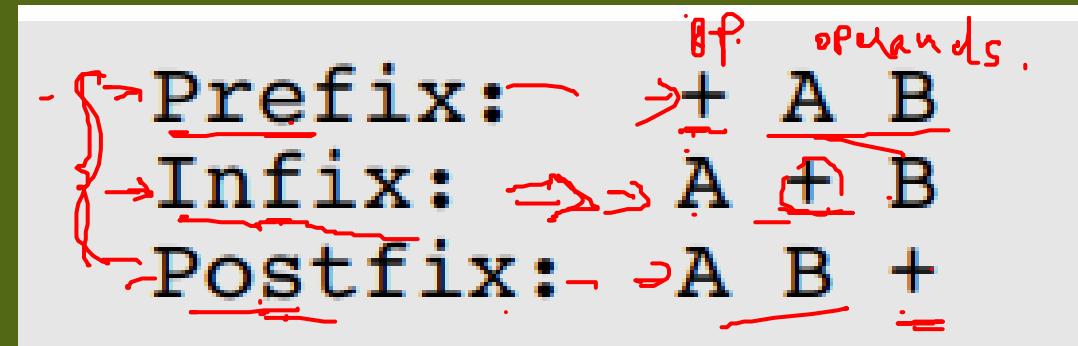
- Recursive function need not call itself directly. It can call itself indirectly as shown



SOME MORE RECURSIVE EXAMPLES: PREFIX TO POSTFIX CONVERSION

An arithmetic expression can be represented in three different formats:

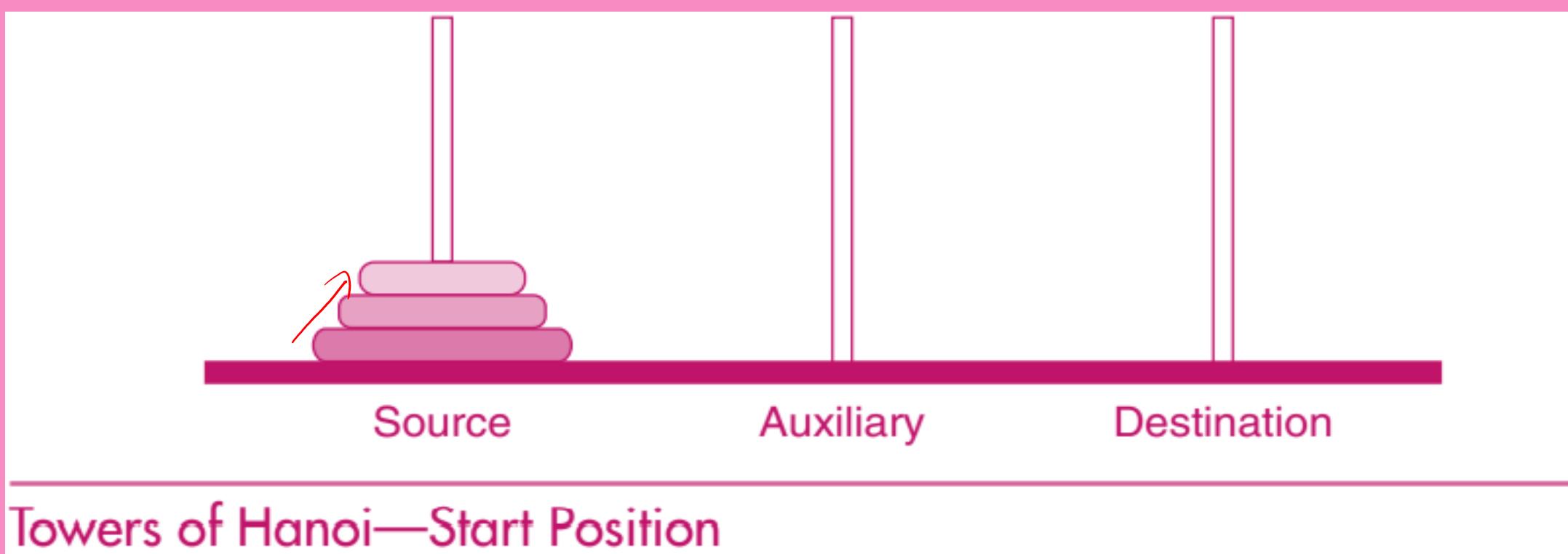
- 1. Infix
- { 2. Postfix
- 3. Prefix



1. **Prefix notation:**
2. **Infix notation:**
3. **Postfix notation:**

Operator **comes before** the operands.
Operator **comes between** the operands.
Operator **comes after** the operands

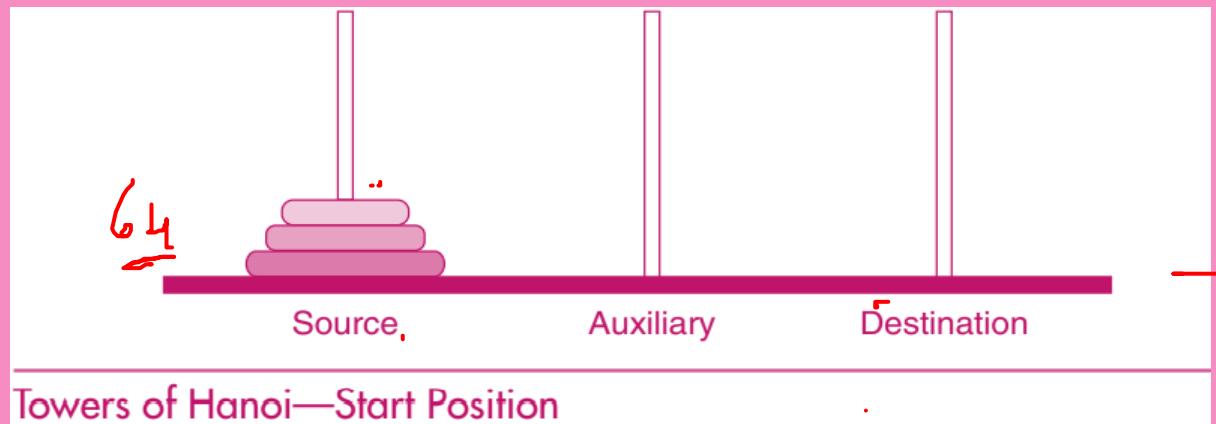
SOME MORE RECURSIVE EXAMPLES: THE TOWERS OF HANOI



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

SOME MORE RECURSIVE EXAMPLES: 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.**



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.

RECURSIVE TOWERS OF HANOI: 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

RECURSIVE TOWERS OF HANOI: DESIGN.



CASE 1: ONE DISK

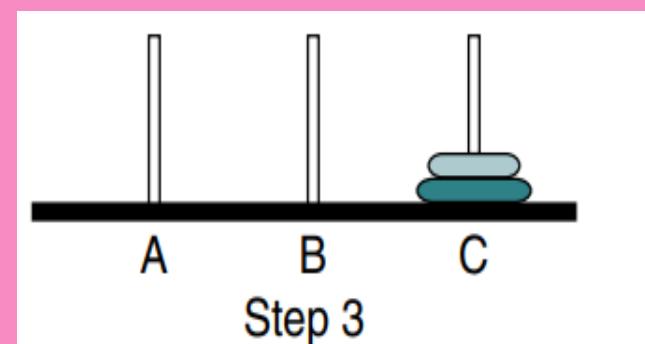
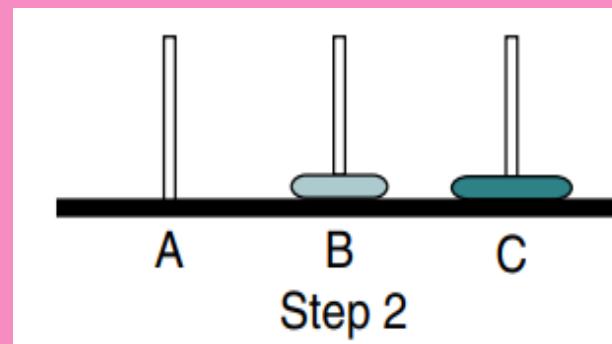
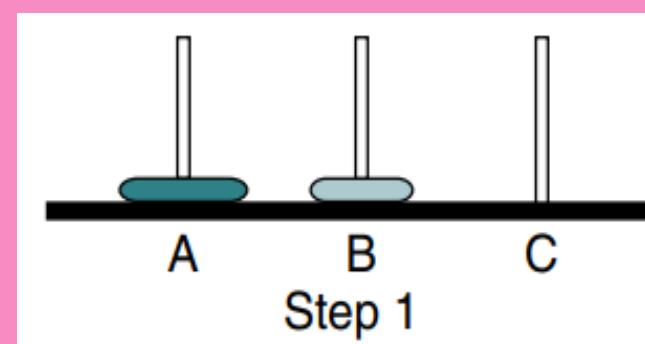
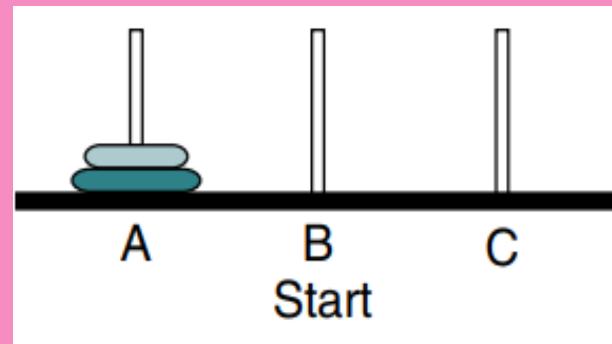
Move one disk from source to destination needle

RECURSIVE TOWERS OF HANOI: 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

RECURSIVE TOWERS OF HANOI: DESIGN.



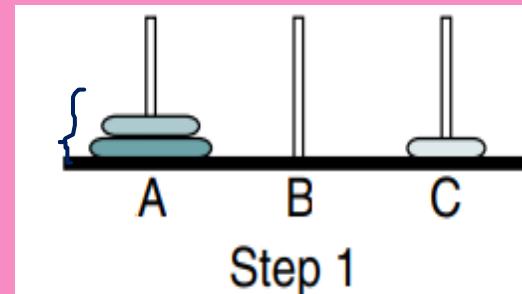
CASE 3: THREE DISKS

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

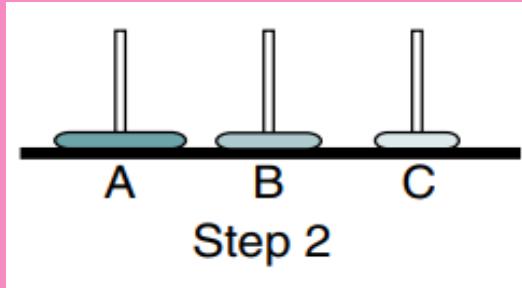
RECURSIVE TOWERS OF HANOI: DESIGN.

CASE 3: THREE DISKS

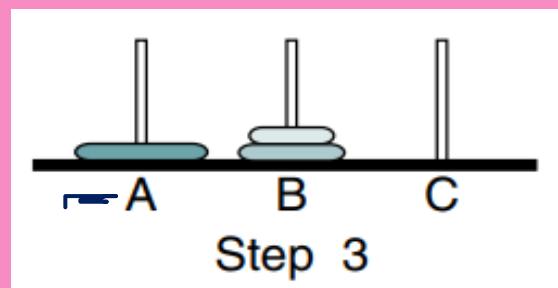
case 2



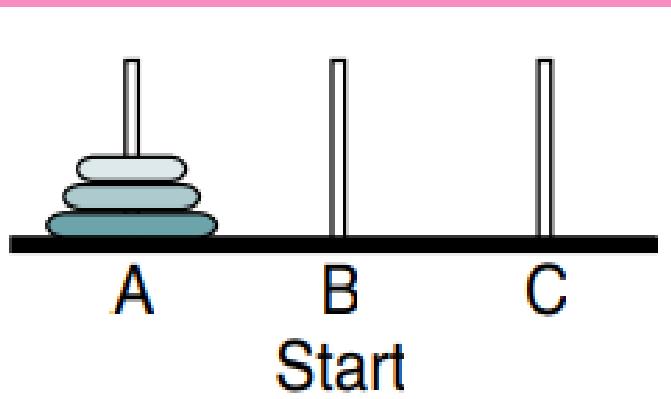
Step 1



Step 2

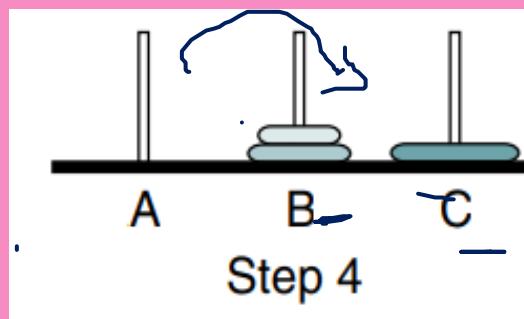


Step 3



Start

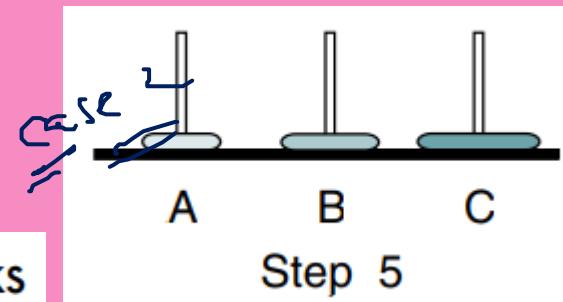
case 1



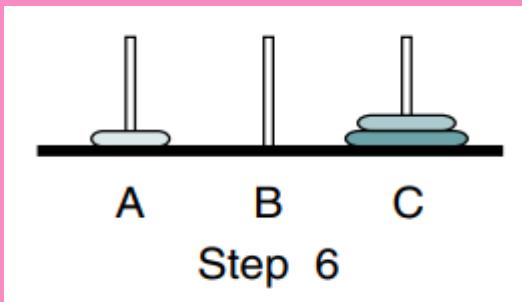
Step 4

Move one disk from source to destination.

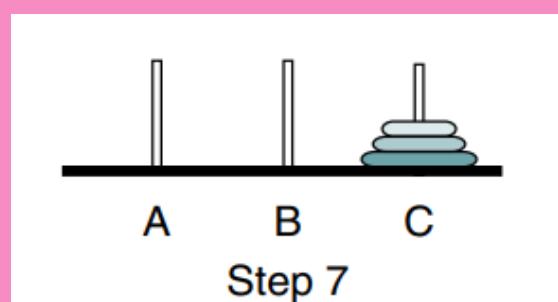
$n=3$



Step 5



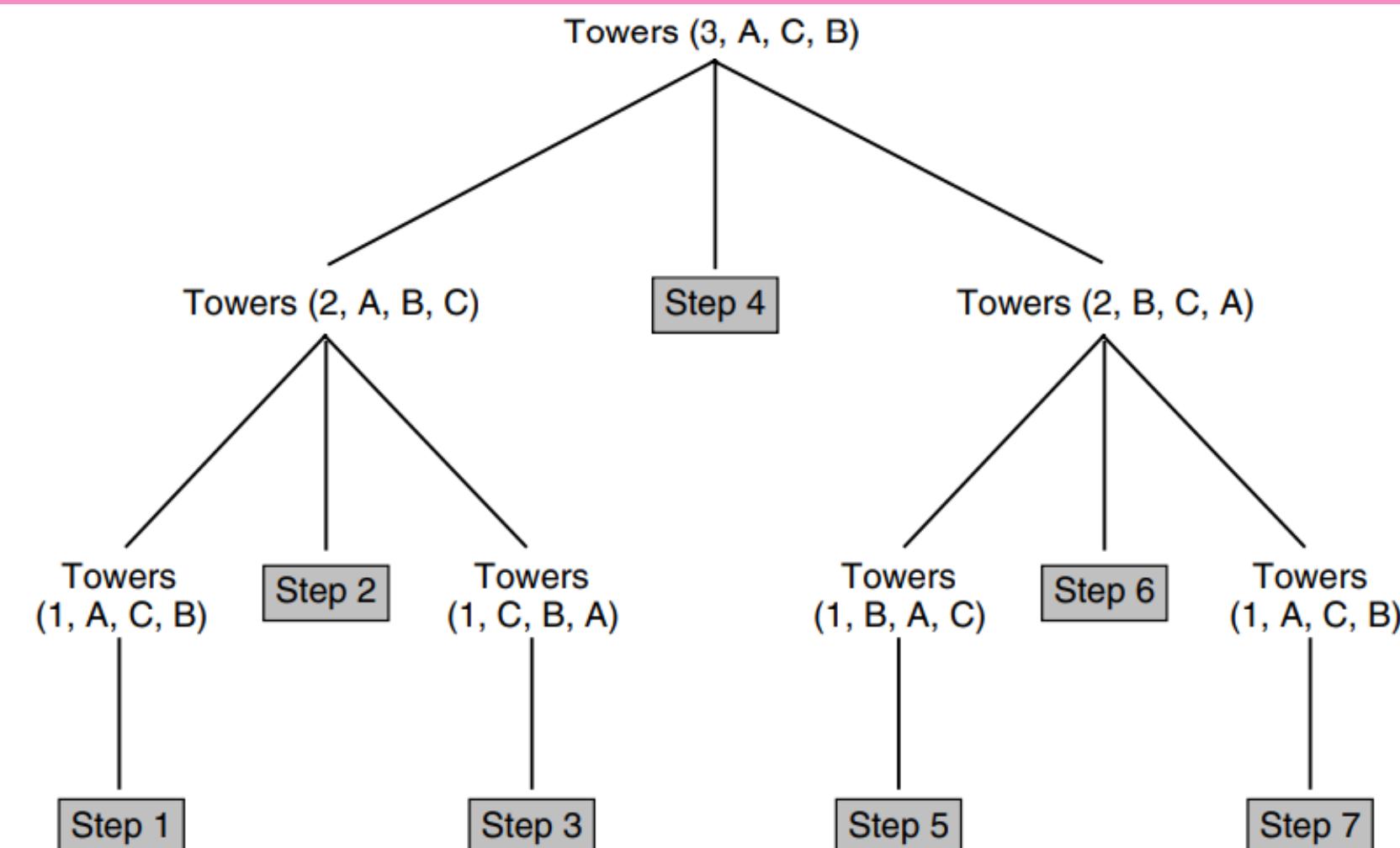
Step 6



Step 7

Towers Solution for Three Disks

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)

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

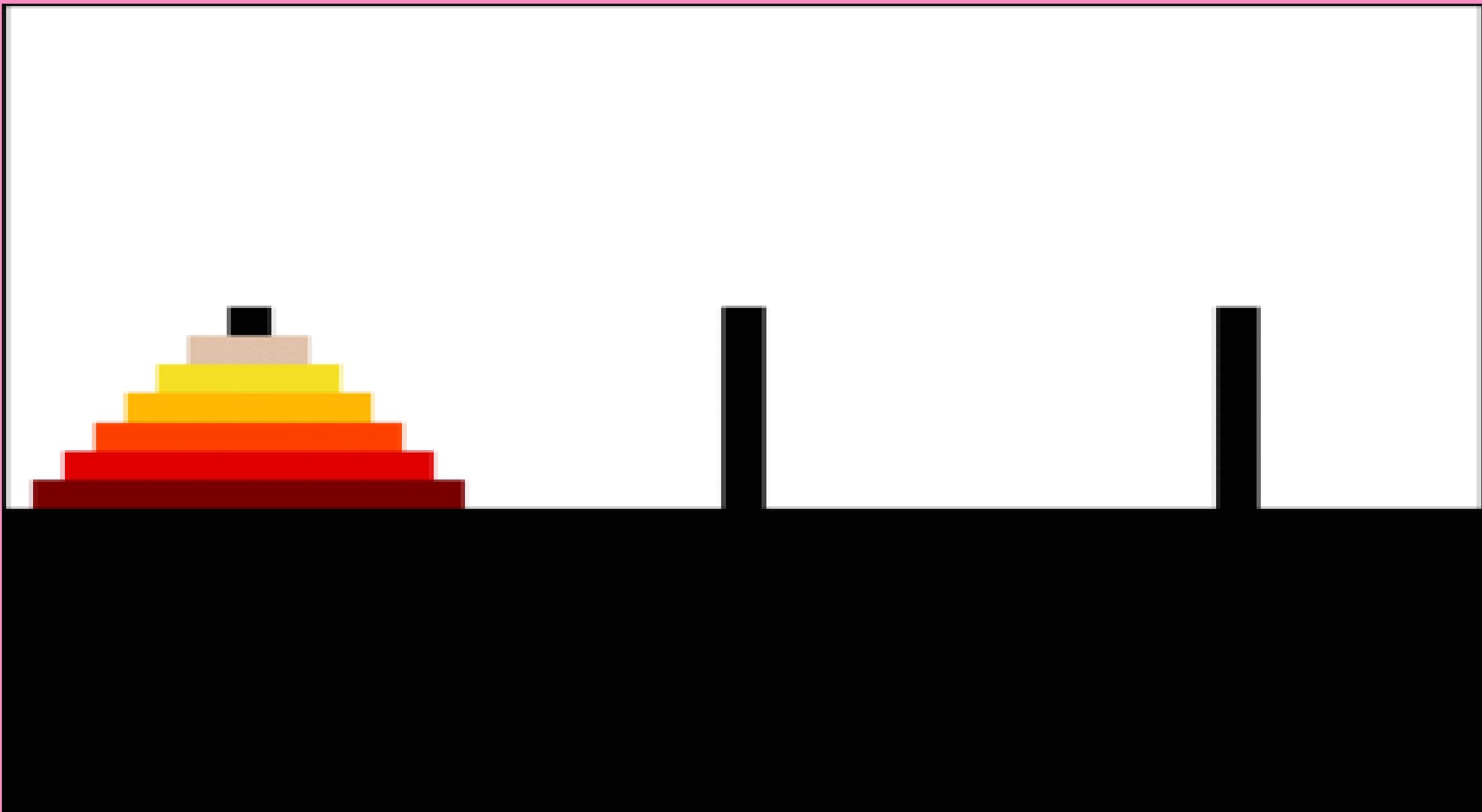
Move from B to A

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
```



C program for tower of hanoi problem

```
void tower (int n, char source, char temp, char destination) {  
    if(n==1) {  
        cout<<“move disk 1 from “<<source<<“ to “<<destination<<endl;  
        return;  
    }  
  
    /*moving n-1 disks from A to B using C as auxiliary*/  
    tower(n-1, source, destination, temp);  
  
    cout<<“move disk “<<n<<“ from “<<source<<“ to  
    “<<destination<<endl;  
  
    /*moving n-1 disks from B to C using A as auxiliary*/  
    tower(n-1, temp, source, destination);  
}
```

LENGTH OF A STRING USING RECURSION

```
int StrLen(str[], int index)
{
    if (str[index] == '\0') return 0;

    return (1 + StrLen(str, index + 1));
}
```

LENGTH OF A STRING USING RECURSION USING STATIC VARIABLE

```
int StrLen(char *str)
{
    static int length=0;
    if(*str != '\0')
    {
        length++;
        StrLen(++str);
    }
    return length;
}
```

TO CHECK WHETHER A GIVEN STRING IS PALINDROME OR NOT USING RECURSION

```
int isPalindrome(char *inputString, int leftIndex, int rightIndex) {  
    /* Recursion termination condition */  
    if(leftIndex >= rightIndex) return 1;  
    if(inputString[leftIndex] == inputString[rightIndex]) {  
        return isPalindrome(inputString, leftIndex + 1, rightIndex - 1);  
    }  
    return 0;  
}
```

```
int main(){
    char inputString[100];
    printf("Enter a string for palindrome check\n");
    scanf("%s", inputString);
    if(isPalindrome(inputString, 0, strlen(inputString) - 1))
        printf("%s is a Palindrome \n", inputString);
    else
        printf("%s is not a Palindrome \n", inputString);
    getch();
    return 0;
}
```

TO COPY ONE STRING TO ANOTHER USING RECURSION

```
void copy(char str1[], char str2[], int index)
{
    str2[index] = str1[index];
    if (str1[index] == '\0') return;
    copy(str1, str2, index + 1);
}
```

Advantages of Recursion

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 book keeping 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.

Iteration

Uses loops

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 calls

Terminates when base condition is reached.

Consumes time and space because of push and pop.

Best suited for some problems and easy to design.

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

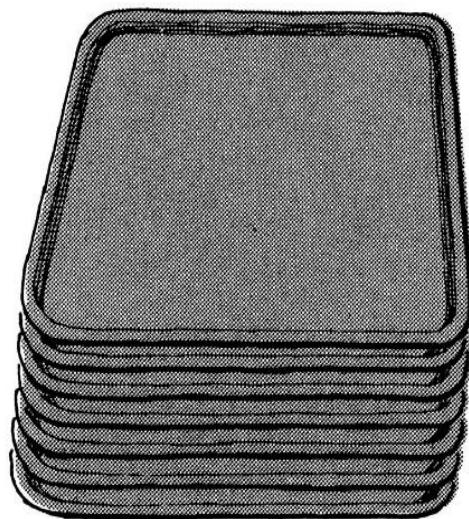
- a. `fun1 (4)?`
- b. `fun1 (10)?`
- c. `fun1 (12)?`

The End.
OF RECURSION.



STACKS

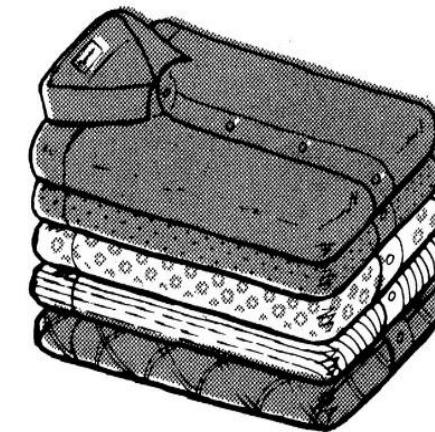
A stack of
cafeteria trays



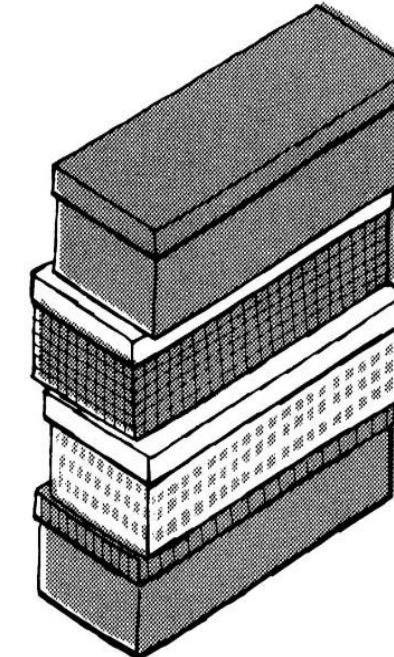
A stack
of pennies



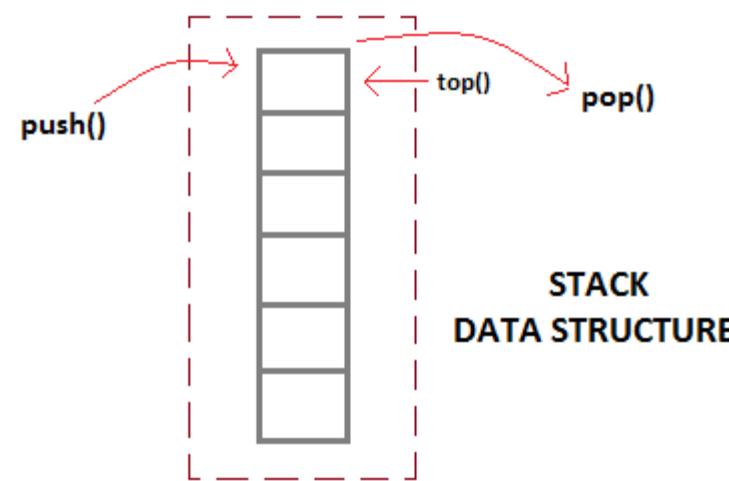
A stack of
neatly folded shirts



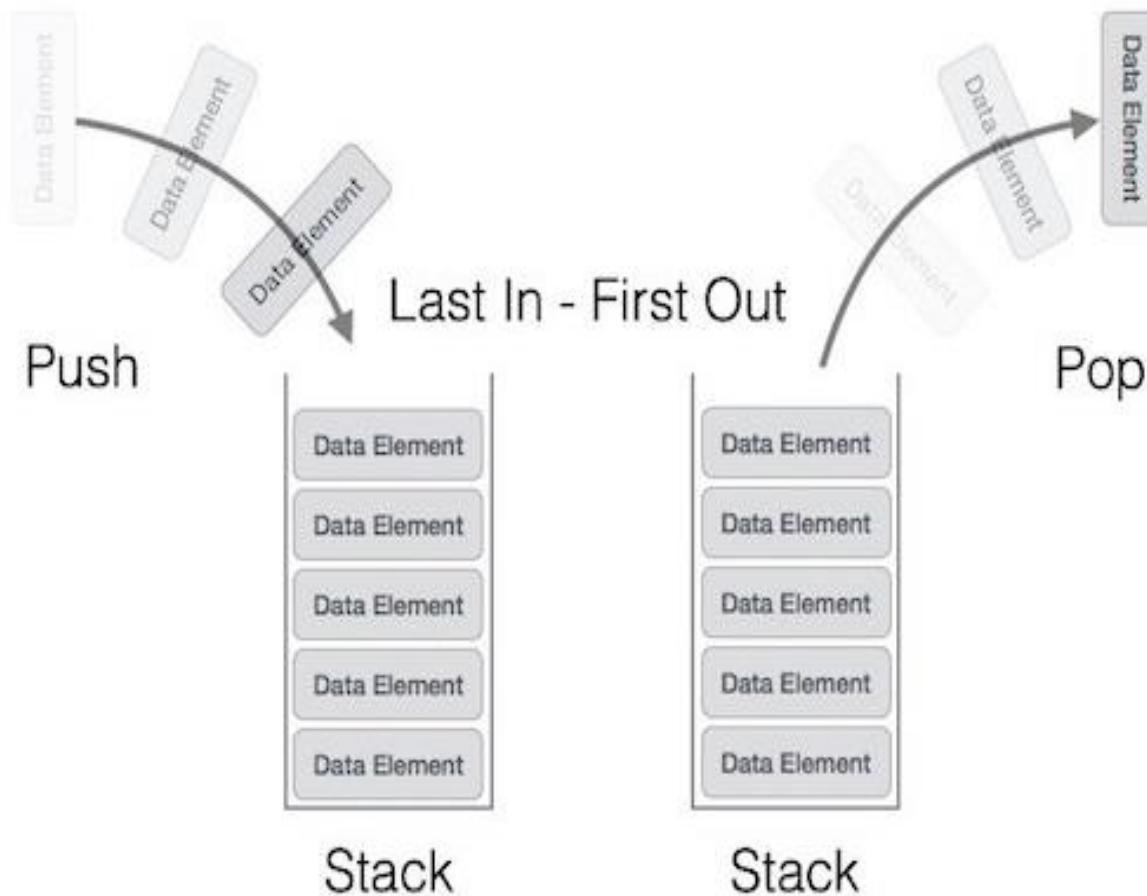
A stack of shoe boxes



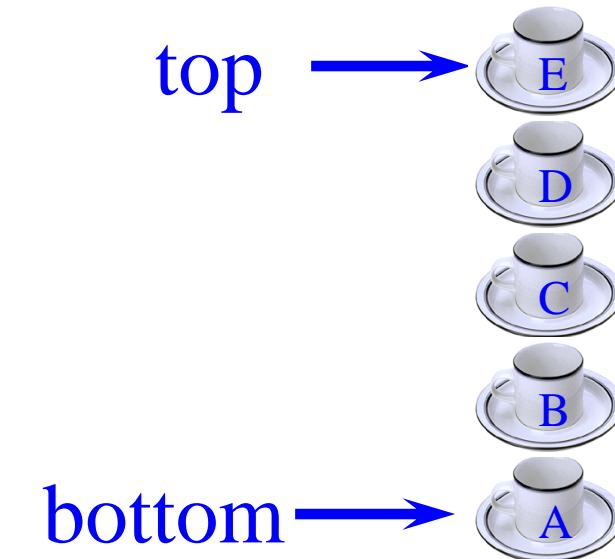
Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the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



Stack Representation



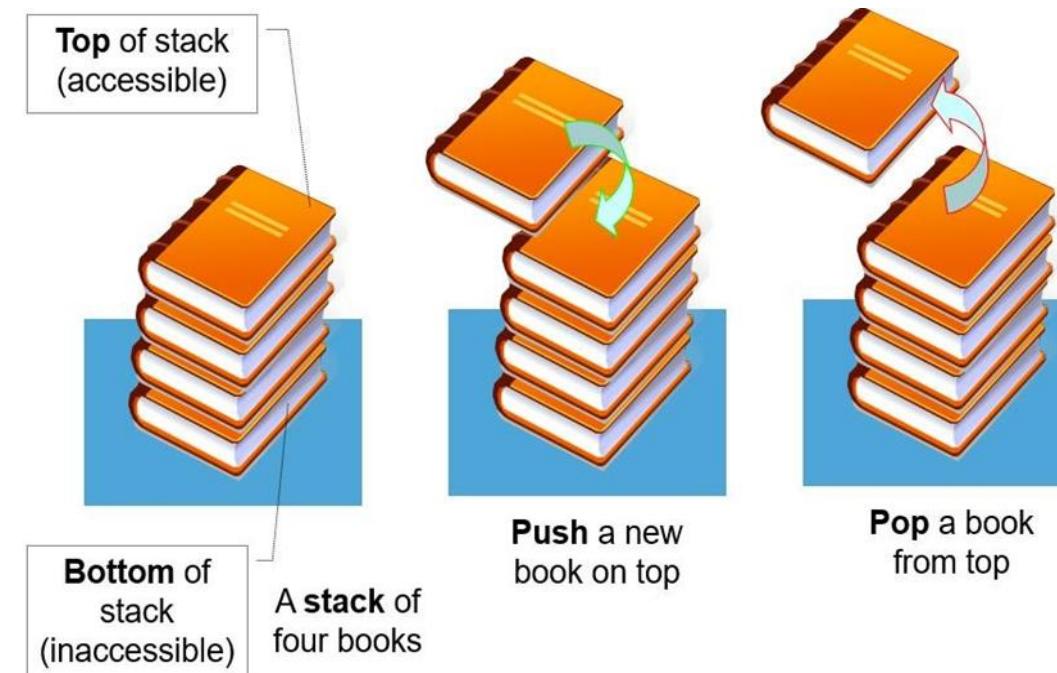
- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from
the top end only.



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

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.

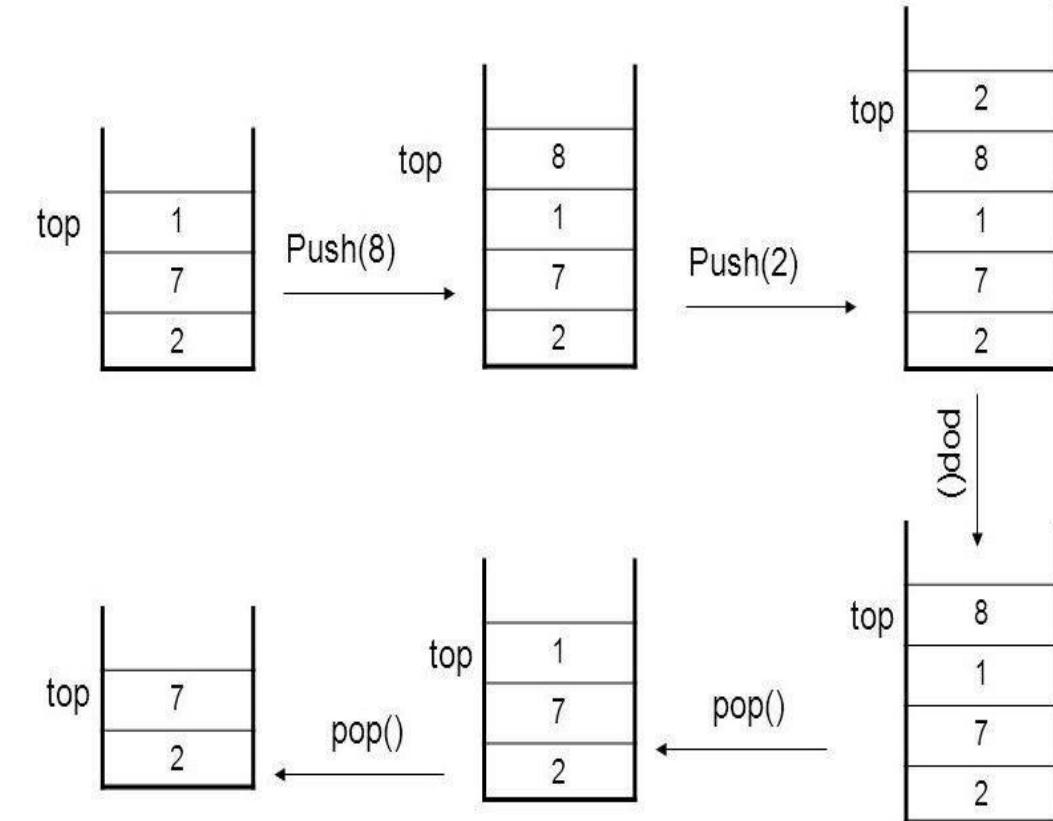
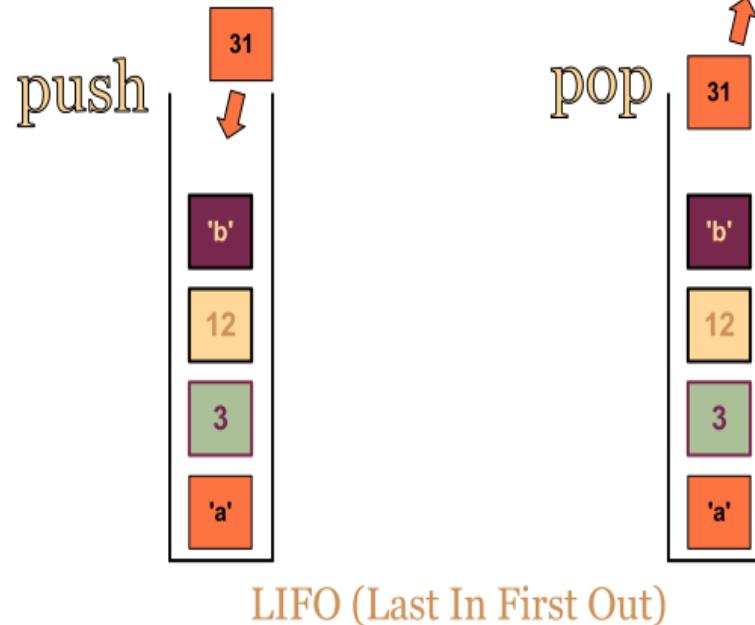
Result : Adds an element of value same as that of the parameter passed at the top of the stack.

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 putting 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
```



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.



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

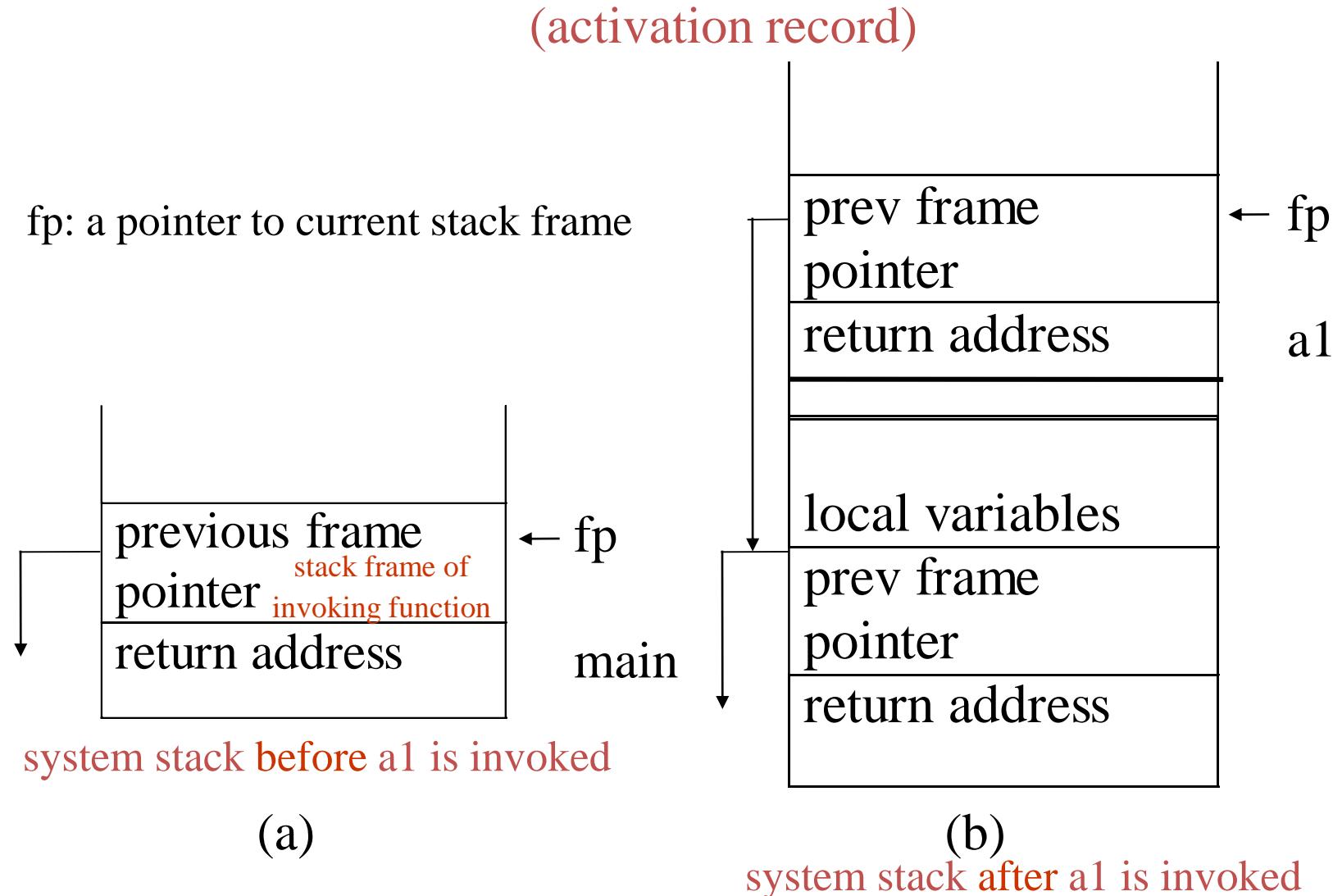
- While return address contains the location of the statement to be executed after the function terminates.
- Only one function executes at given time which is the top of stack.
- If this function invokes another, the non-static local variables parameters of invoking function are added to stack frame.



Stacks in Function call – System Stack

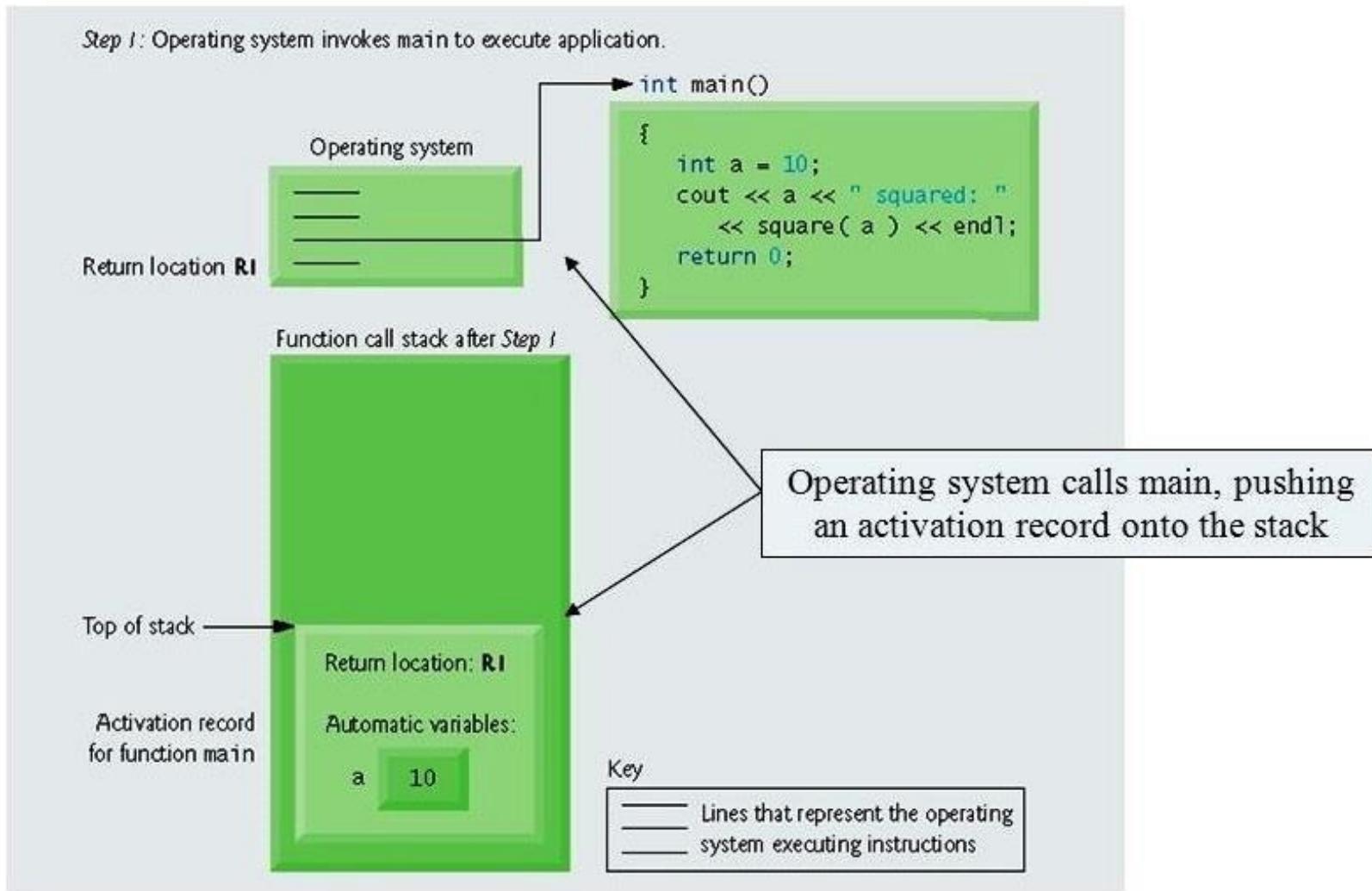
- Assume that main() invokes function a1.
- It creates stack frame for a1.
- Frame pointer is a pointer to the current stack frame
- Also system maintains separately a stack pointer
- When a function terminates its stack frame is removed
- Processing of invoking function which is on top of stack continues

An application of stack: stack frame of function call



Applications of Stack

■ Function Call (Continued ...)



Applications of Stack

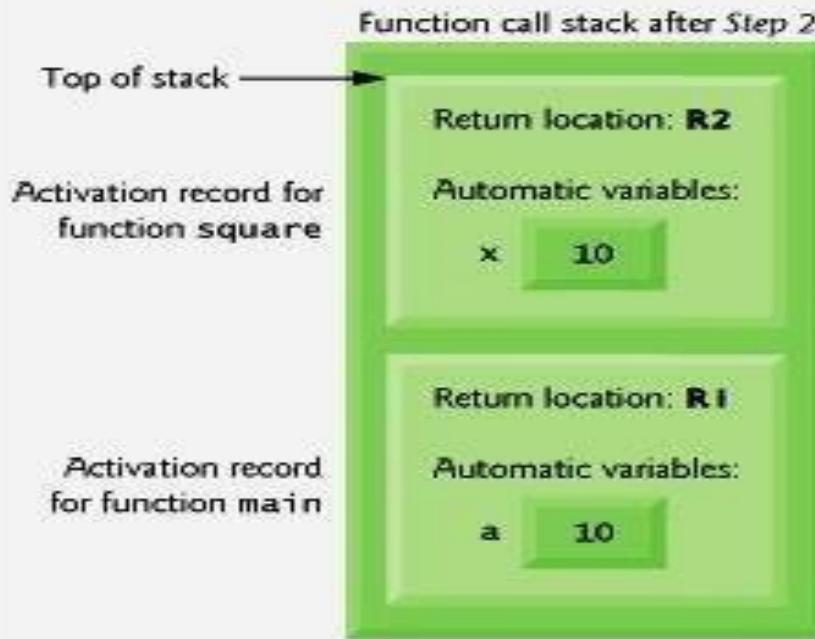
■ Function Call (Continued ...)

Step 2: main invokes function square to perform calculation.

```
int main()
{
    int a = 10;
    cout << a << " squared: "
        << square( a ) << endl;
    return 0;
}
```

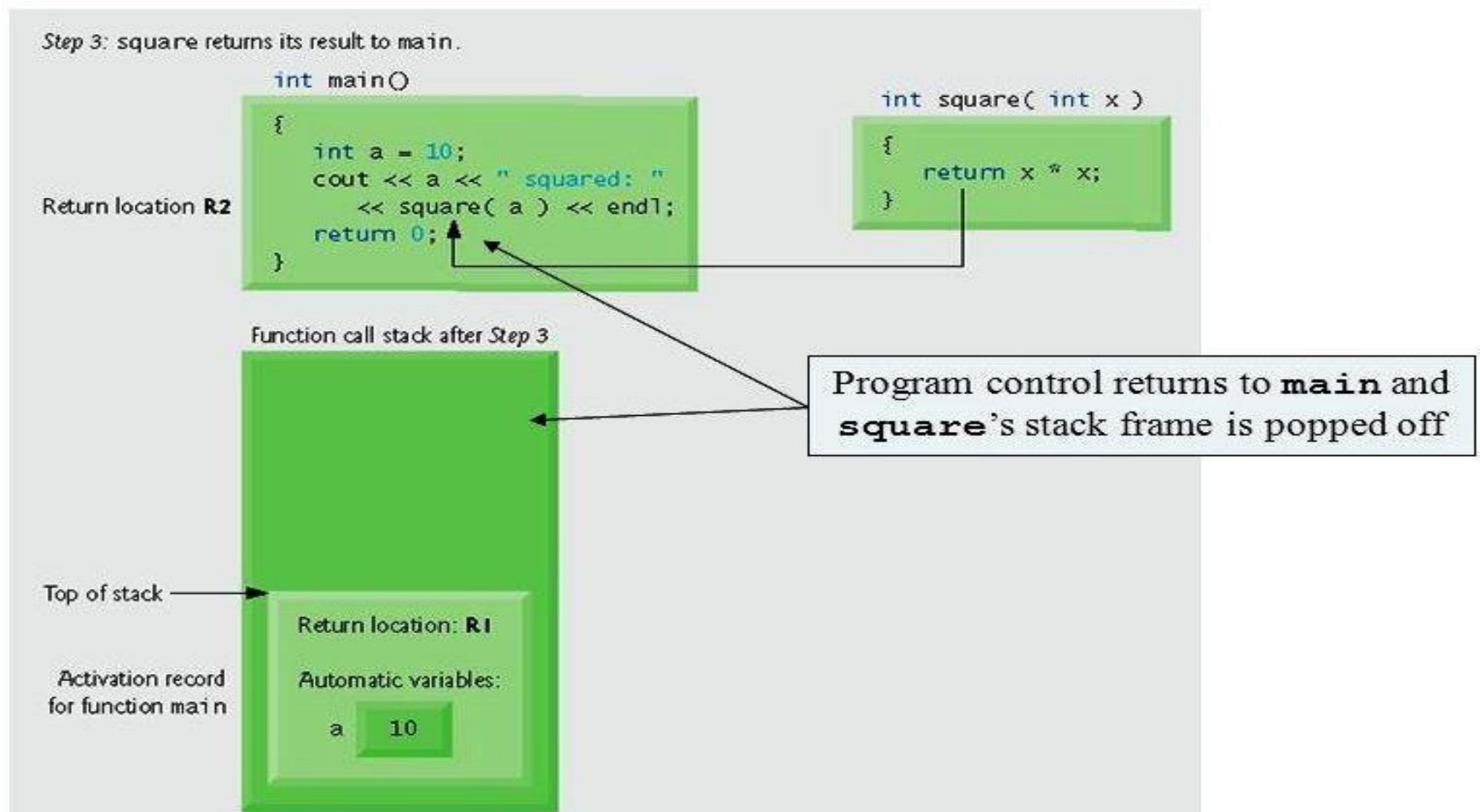
Return location **R2**

→ int square(int x)
{
 return x * x;
}



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



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

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

2
1
0

(2,6) (1,13) (15,19) (21,25)(27,31) (0,32) (34,38)



Algorithm for checking expression

1. Scan the expression from left to right.
2. Whenever a scope opener(‘(’, ’{’, ’[‘) is encountered while scanning the expression, it is pushed to stack.
3. Whenever a scope ender(‘)’, ’}’, ’]’) is encountered, the stack is examined. If stack is empty, scope ender does not have a matching opener and hence string is invalid. If stack is non empty, we pop the stack and check whether the popped item corresponds to scope ender. If a match occurs, we continue. If it does not, the string is invalid.
4. When the end of string is reached, the stack must be empty; otherwise 1 or more scopes have been opened which have not been closed and string is invalid.



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

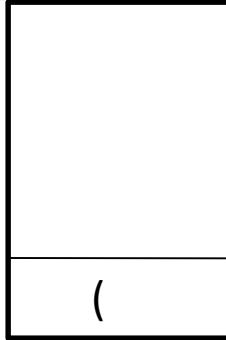


Applications of Stack

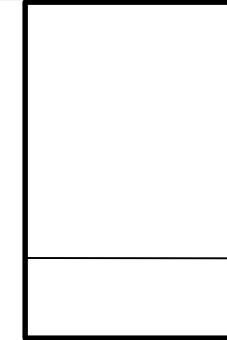
4. If (symbol = ')' or symbol = ']' or symbol = '}') then
 - If stack is empty then
 - Set VALID := FALSE;
 - else
 - {
 1. Pop an item op from the stack;
 2. If (op is not the matching opener of symbol) then Set
 3. VALID := FALSE;
 4. }
5. If stack is not empty, then
 - Set VALID := FALSE;
6. return



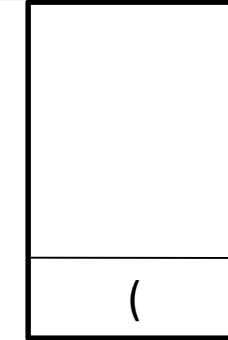
Ex1: $(a+b) * (c+d)$



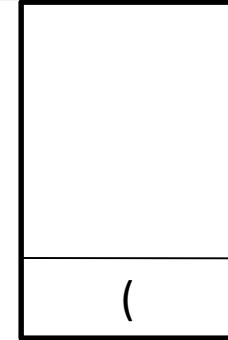
(
push '('



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

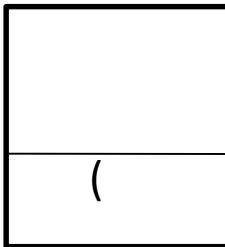


(a+b) *(
Push '('



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

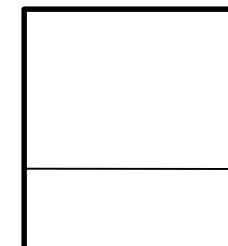
Ex2: $(a+b)*c+d)$



(
push '('



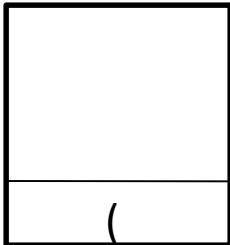
(a+b)
pop '(' and
continue



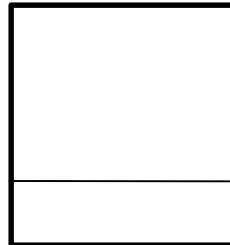
(a+b)*c+d)
Stack empty when ')' encountered. Hence invalid



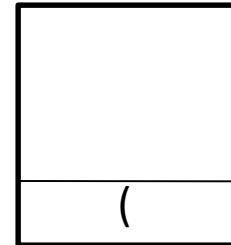
Ex3: $(a+b)^*(\{c^*d)$



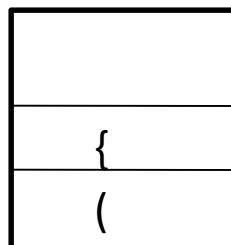
(
push '('



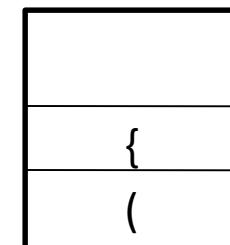
(a+b)
**pop '(' and
continue**



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



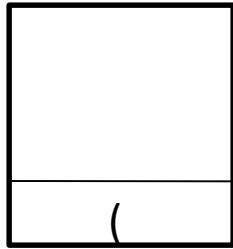
(a+b)*({
**push '{' and
continue'**



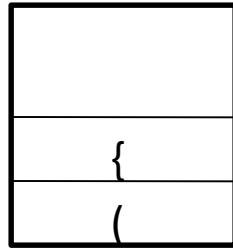
(a+b)*({c*d)
**No match between closing scope ')' and opening
scope '{'. Hence invalid.**



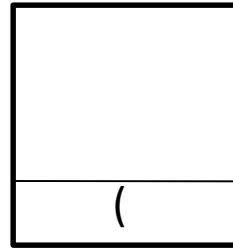
Ex4: $(a+\{b*c\}+(c*d))$



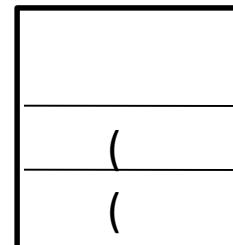
(
push '(' and continue



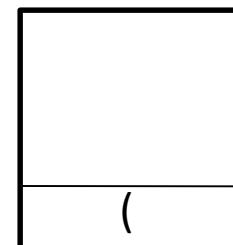
(a+{
push '{' and continue



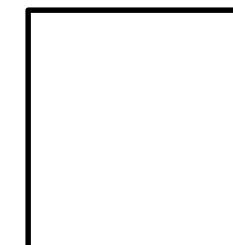
(a+{b*c}
pop '{' and continue



(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



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:

$$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.6666\cdots$$

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

Postfix: no parentheses, no precedence

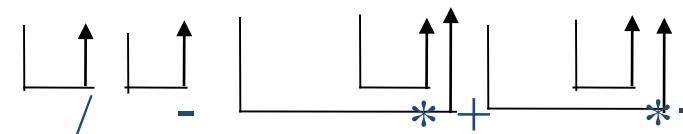
- (1) Fully parenthesize expression

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

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

- (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.



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

$$a + b*c$$

$$(a + b)*c$$



Convert infix to post fix expression

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

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/-



■ 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.



■ 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



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.



■ 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

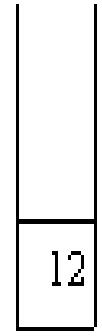
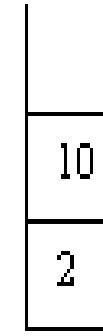


Example for evaluation of Postfix expression

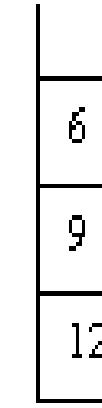
2 10 + 9 6 - /

push 2
push 10

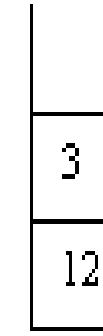
pop 10
pop 2
push 2 + 10 = 12



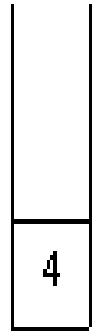
push 9
push 6
push 9 - 6 = 3



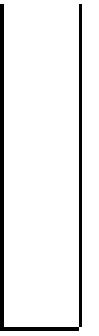
pop 6
pop 9
push 9 - 6 = 3



pop 3
pop 12
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



Multiple Stacks



Method 1 (Divide the space in two halves)

- Divide the array in two halves and assign the half half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.
- The problem with this method is inefficient use of array space.
- A stack push operation may result in stack overflow even if there is space available in arr[].



Method 2 (A space efficient implementation)

- Stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0.
- The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1).
- Both stacks grow (or shrink) in opposite direction.
- To check for overflow, all we need to check is for space between top elements of both stacks.



Multiple Stacks

```
class twoStacks
{
    int arr[size];
    int top1, top2;
public:
    twoStacks() // constructor
    {
        top1 = -1;
        top2 = size;
    }
```



Multiple Stack implementation in single array

```
// Method to push an element x to stack1
void push1(int x)
{
    // There is at least one empty space for new element
    if(top1 < top2 - 1)
    {
        top1++;
        arr[top1] = x;
    }
    else
    {
        cout << "Stack Overflow";
    }
}
```



```
// Method to push an element x to stack2
void push2(int x)
{
    // There is at least one empty space for new element
    if (top1 < top2 - 1)
    {
        top2--;
        arr[top2] = x;
    }
    else
    {
        cout << "Stack Overflow";
    }
}
```



```
// Method to pop an element from first stack
void pop1()
{
    if (top1 >= 0 )
    {
        int x = arr[top1];
        top1--;
        cout<<"popped element is"<<x;
    }
    else
    {
        cout << "Stack UnderFlow";
    }
}
```



```
// Method to pop an element from second stack
void pop2()
{
    if (top2 < size)
    {
        int x = arr[top2];
        top2++;
        cout<<"popped element is"<<x;
    }
    else
    {
        cout << "Stack UnderFlow";
    }
}
```



```
int main()
{
    twoStacks ts;
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is " << ts.pop1();
    ts.push2(40);
    cout << "\nPopped element from stack2 is " << ts.pop2();
return 0;
```



END



Queues

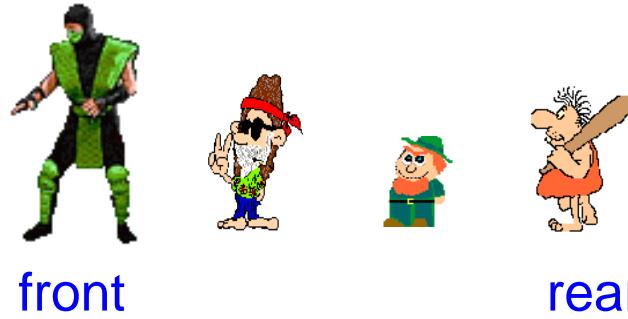


Bus Stop Queue





Bus Stop Queue



front

rear





Bus Stop Queue



front

rear





Bus Stop Queue



front

rear



Queue

Insertion and Deletion
happen on different ends

Enqueue

Rear

Front

Dequeue

First in, first out



0

1

2

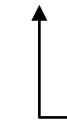
3

4



Front=-1

points to location
one before the 1st element



Rear

always points to last data



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.



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.

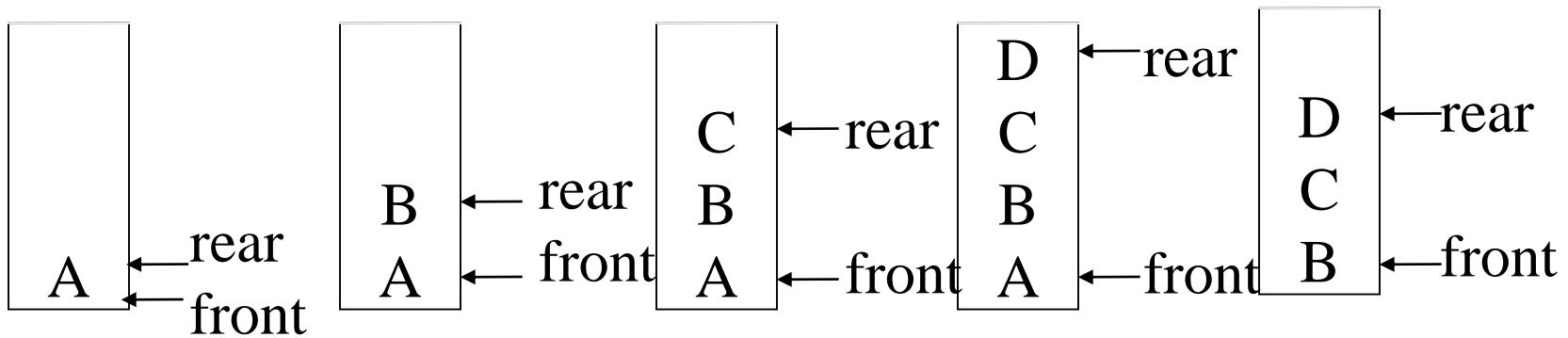


Basic features of Queue

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

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	L	Job 1 is added
-1	1	L D	Job 2 is added
-1	2	L D B	Job 3 is added
0	2	D B	Job 1 is deleted
1	2	B	Job 2 is deleted

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



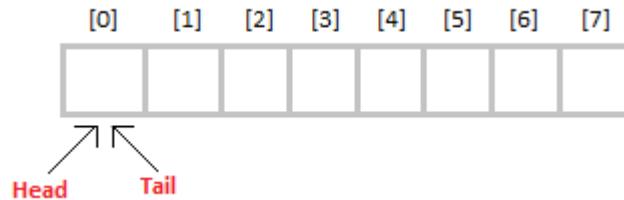
Applications of Queue

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

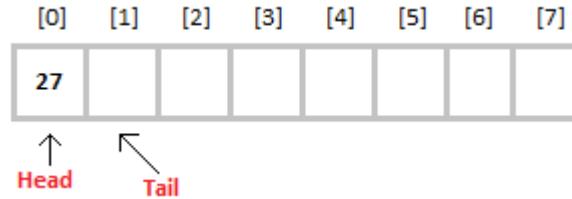


Implementation of Queue Data Structure

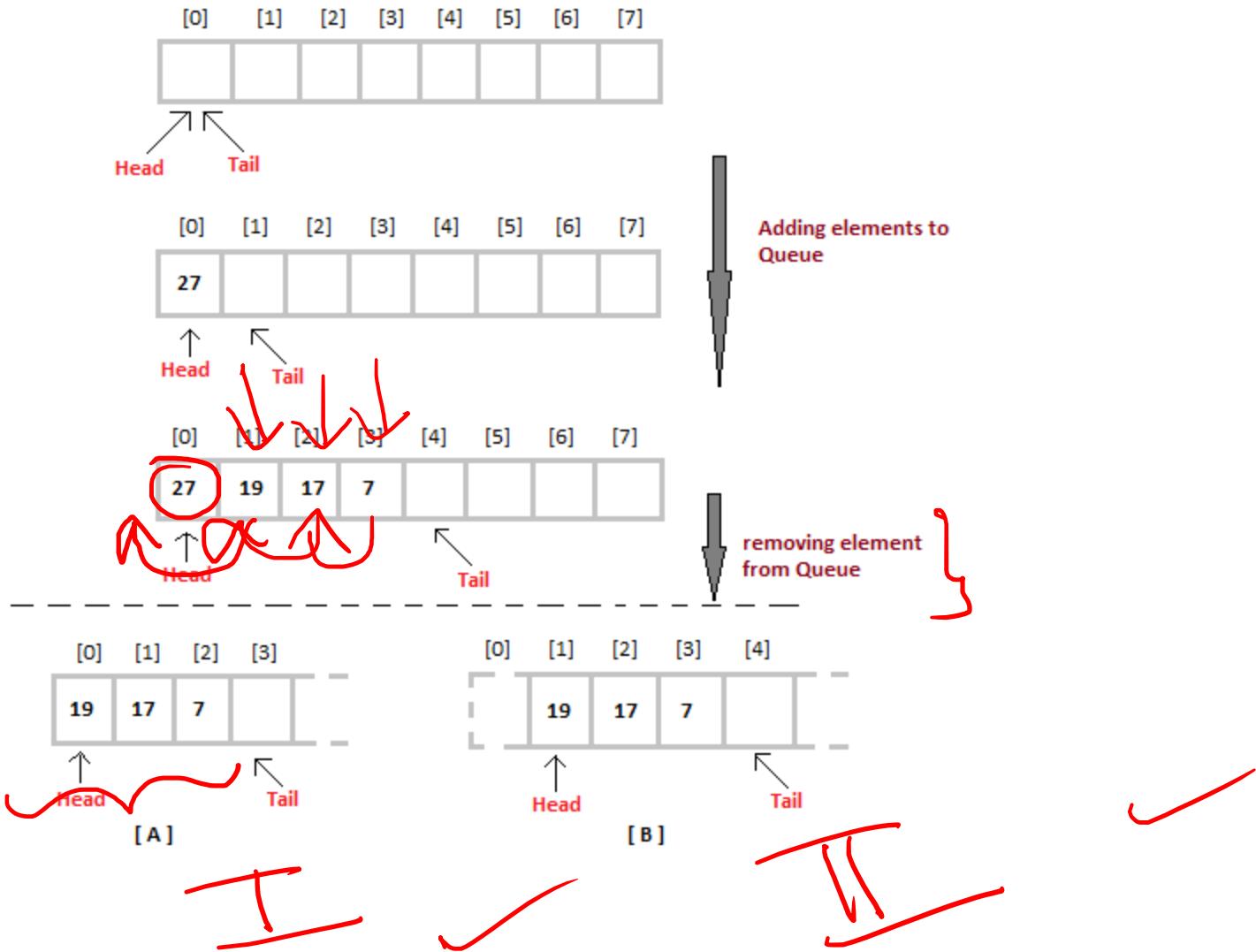
- Initially the **head(FRONT)** and the **tail(REAR)** of the queue points at the first index of the array



- 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 queuefront;
}
```



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);  
}
```





Queue variations

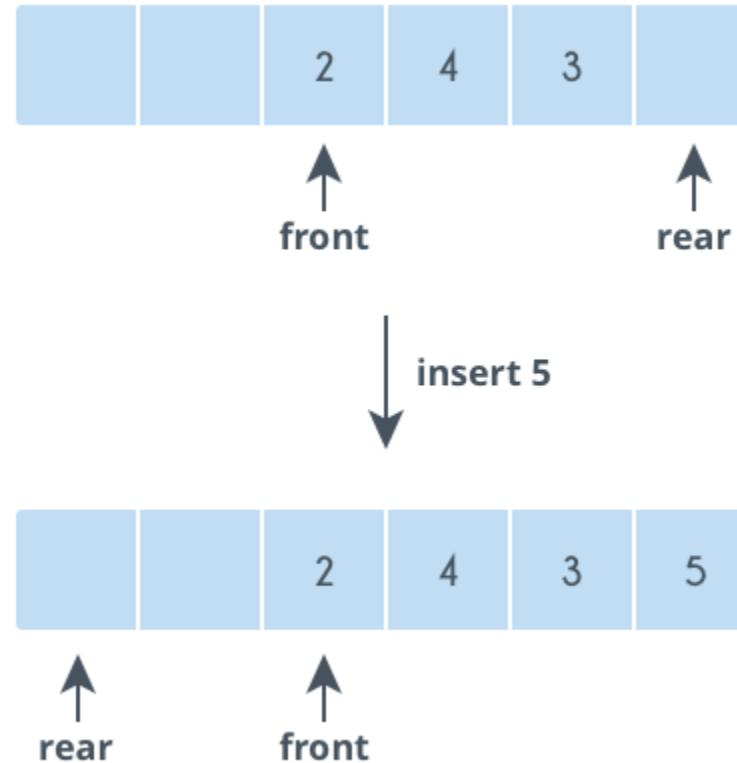
The standard queue data structure has the following variations:

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



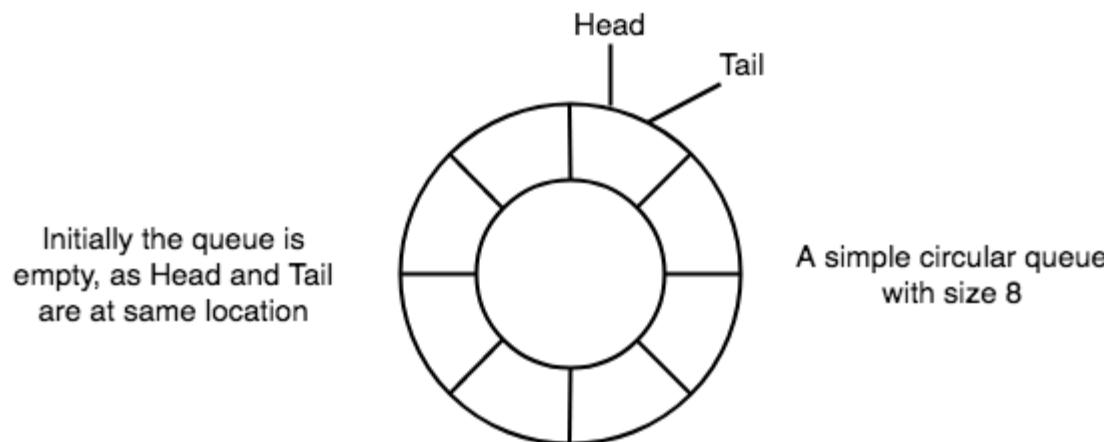
Circular queues

- 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).

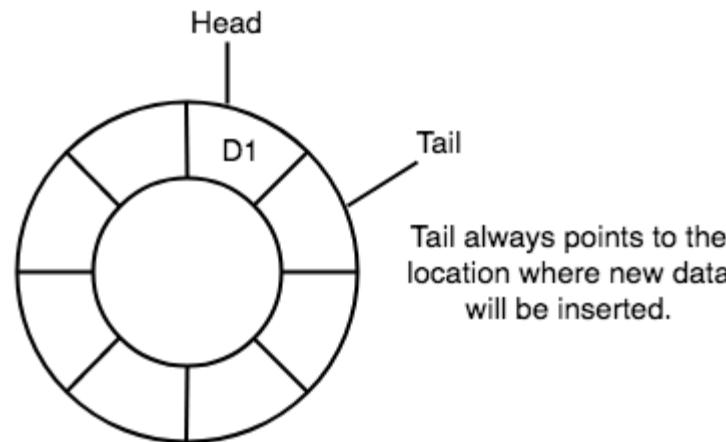


Basic features of Circular Queue

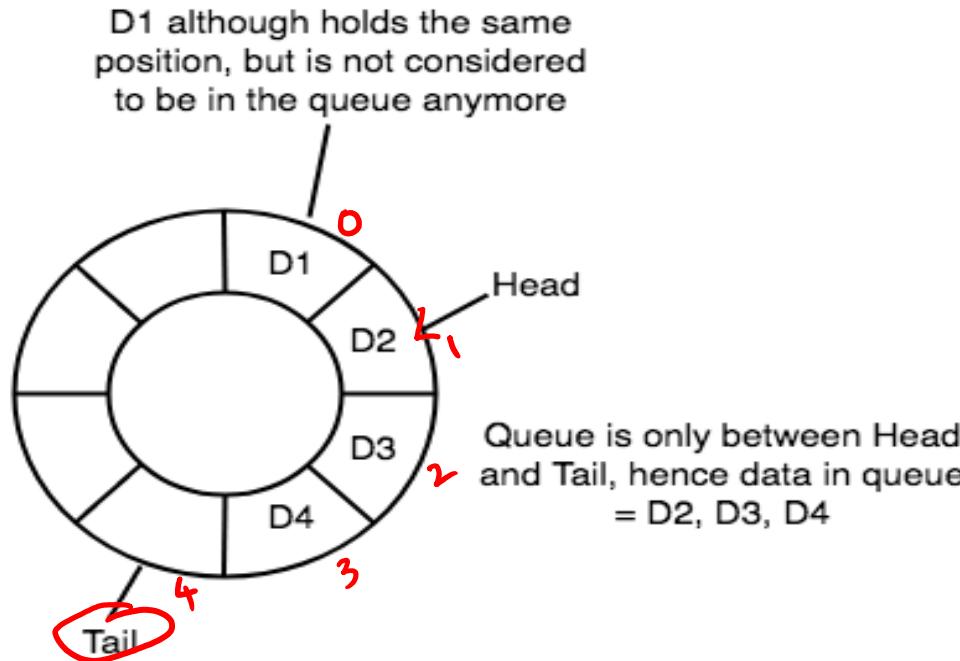
1. Head pointer will always point to the front of the queue, and
2. Tail pointer will always point to the end of the queue.
3. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



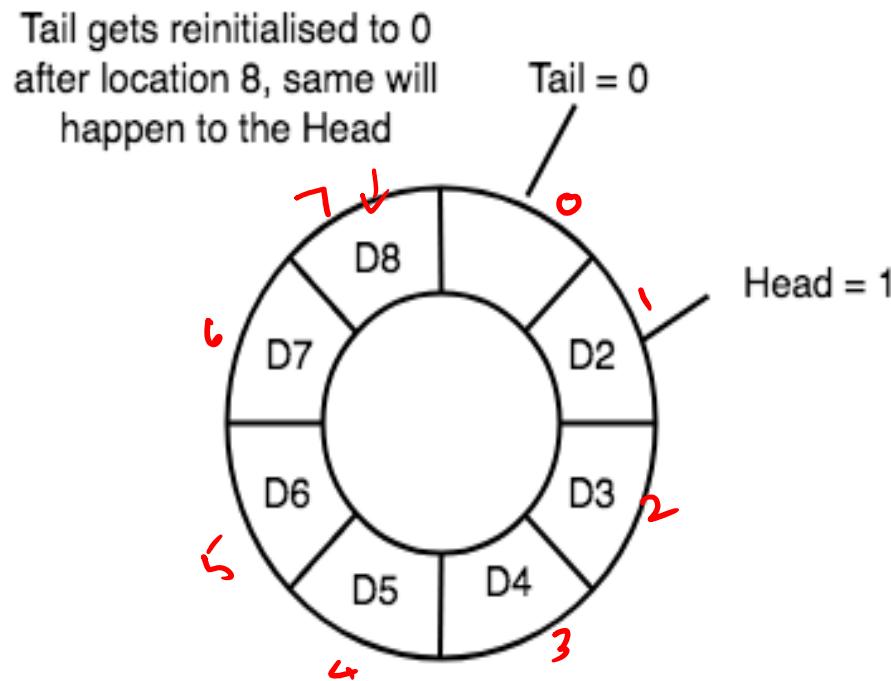
New data is always added to the location pointed by the **tail** pointer, and once the data is added, **tail** pointer is incremented to point to the next available location.



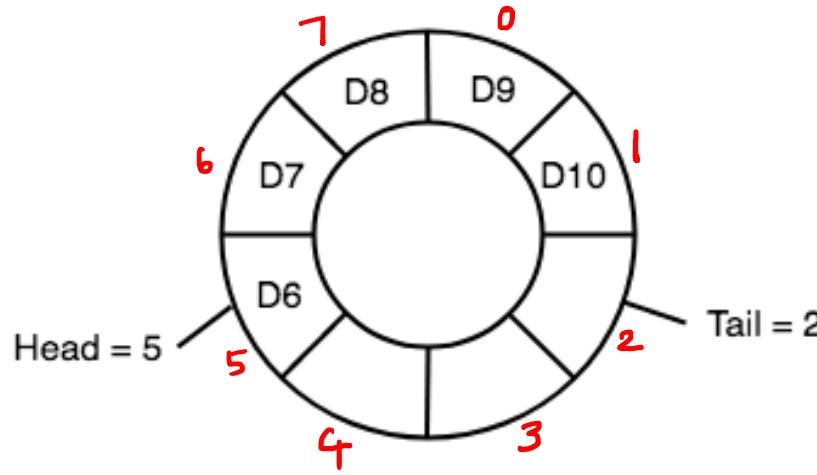
- In a circular queue, data is not actually removed from the queue.
- Only the head pointer is incremented by one position when **dequeue** is executed.
- As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.



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.



In such a situation the value of the Head pointer will be greater than the Tail pointer



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.



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.

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

■ Types of deque

- **Input-restricted:** Insert at rear end only, delete from any end
- **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

■ Operations on Priority Queue

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



Priority Queue and Heap

■ Priority Queue

- Queue: FIFO data structure
- Priority Queue: **Order of deletion** is determined by the element's priority
 - ▶ Deleted either in increasing or decreasing priority
 - ▶ Efficiently implemented with heap data structure
 - Heap is a complete binary tree; can be implemented using the array representation

Other representations are leftist trees, fibonacci heaps, binomial heaps, skew heaps and pairing heaps

- Implemented with linked data structures



END OF QUEUE

Pointers



Pointer Introduction

- Pointer is a memory location or a variable which stores the memory address of another variable in memory.
- Pointers are more commonly used in C than other languages (such as BASIC, Pascal, and certainly Java, which has no pointers).

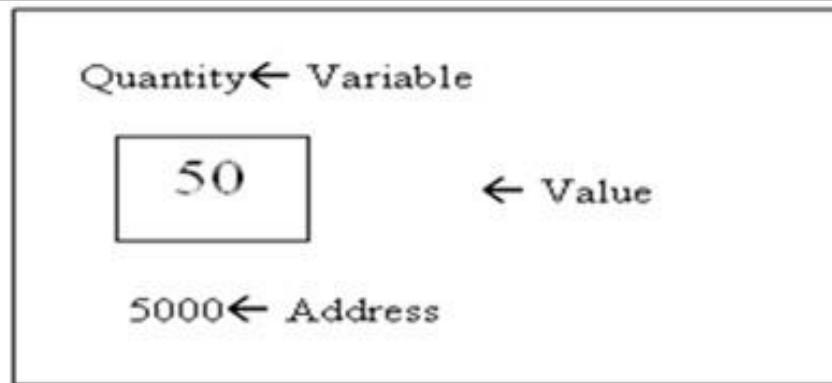
Common Uses of Pointers

- Accessing array elements.
- Passing arrays and strings to functions
- Passing arguments to a function when the function needs to modify the original argument.
- Obtaining memory from the system.
- Creating data structures such as linked lists.

Why Pointers?

- It enables the access to a variable that is defined outside the function.
- Pointers reduce the length and complexity of a program.
- They increase the execution speed.
- The use of a pointer array to character strings results in saving of data storage space in memory

Concept I



int Quantity = 50;

This statement instructs the system to find a location for the integer variable **Quantity** and puts the value **50** in that location.

- Assume that system has chosen address location 5000 for **Quantity**.

Concept II

- During Execution of the program, the system always associates the name **Quantity** with the **address 5000**.
- We may have access to the **value 50** by using either the name of the **variable Quantity** or the **address 5000**.
- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory, like any other variable.
- To assign the **address 5000** (the location of **Quantity**) to a **variable p**, we can write:

p = &Quantity ;

Such variables that hold memory addresses are called **Pointer Variables**

Concept III

Variable	Value	Address
Quantity	50	5000
p	5000	5048

The operator & can be called as **address of** and can be used only with a simple variable or an array element.



Declaring and Initializing Pointers I

- Declaration Syntax:
<datatype> *pt_name;
- This tells the compiler 3 things about the pt name:
 - The **asterisk(*)** tells the variable **pt_name** is a pointer variable.
 - **Pt_name** needs a memory location.
 - **Pt_name** points to a variable of type **data type**
- Example:
`int *p; //declares a variable p as a pointer variable that points to an integer data type.`
- `float *x; // declares x as a pointer to floating point variable.`



Declaring and Initializing Pointers II

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement:

int quantity;

p = &quantity;

- p now contains the address of quantity. This is known as **pointer initialization**.



To be taken care..

- Before a pointer is initialized, it should not be used.
- We must ensure that the pointer variables always point to the corresponding type of data.
- Assigning an absolute address to a pointer variable is prohibited.

i.e **p=5000**

- A pointer variable can be initialized in its declaration itself.
- Example: **int x, *p=&x;** declares **x** as an integer variable and then initializes **p** to the address of **x**.
- **int *p = &x, x;** //not valid.



Accessing a variable through its pointer

- When the operator ***** is placed before a pointer variable in an expression on the R.H.S of equal sign, the pointer returns the value of the variable **quantity**
- The ***** can be remembered as value at address. (***p** where **p** is the address!)
- Example:

p = &quantity;

n = *p; is equivalent to

n = * &quantity; which in turn equal to

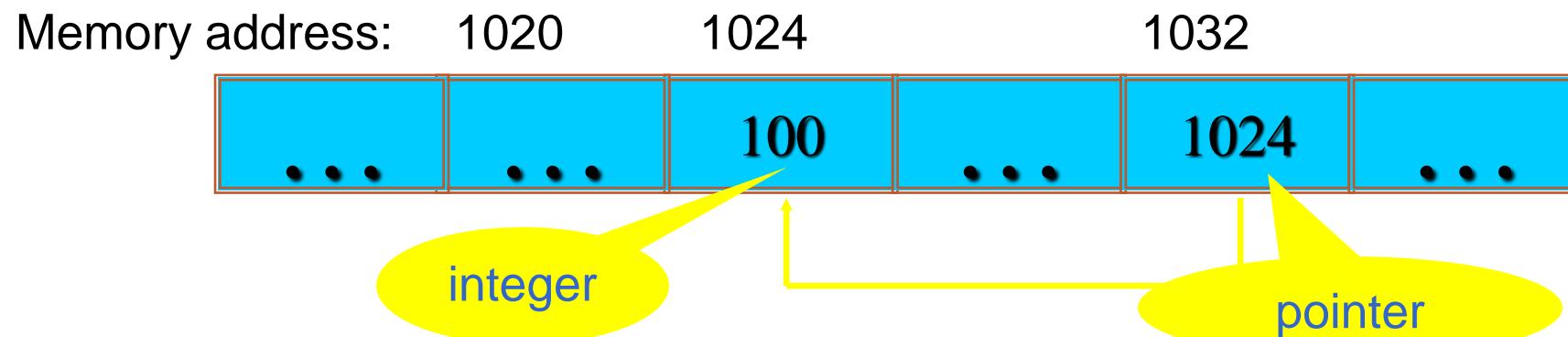
n = quantity;

- **&** is the '**reference**' operator and can be read as "**address of**"
- ***** is the '**dereference**' operator and can be read as "**value pointed by**"

Pointers

A pointer is a variable used to store the address of a memory cell.

We can use the pointer to reference this memory cell

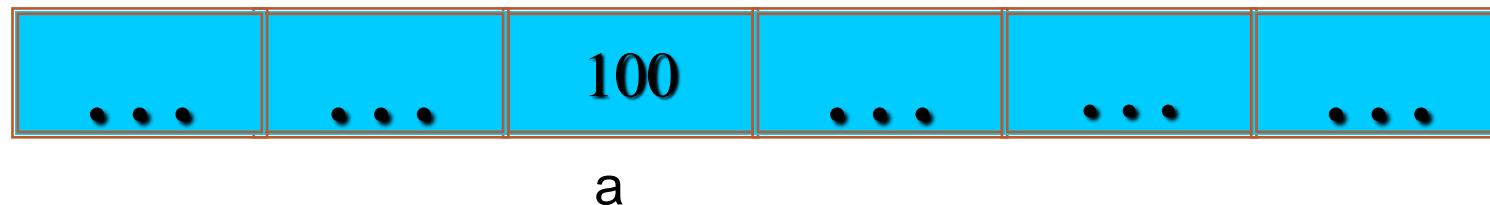


Address Operator &

The "*address of*" operator (&) gives the memory address of the variable

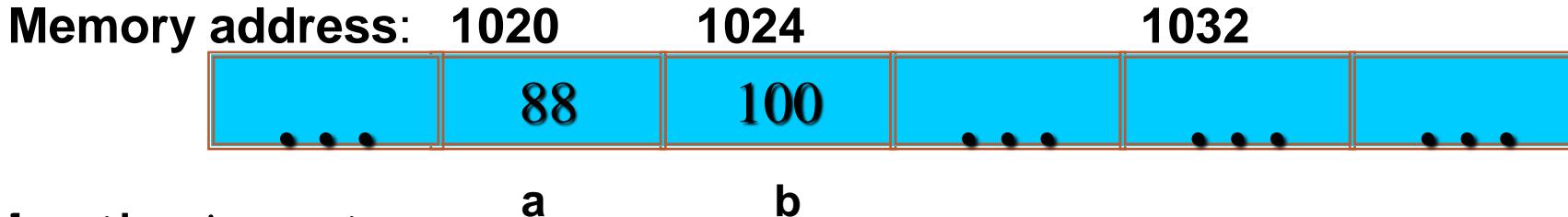
- Usage: **&variable_name**

Memory address: 1020 1024



```
int a = 100;  
//get the value,  
cout << a;      //prints 100  
//get the memory address  
cout << &a;      //prints 1024
```

Address Operator &



```
#include <iostream>

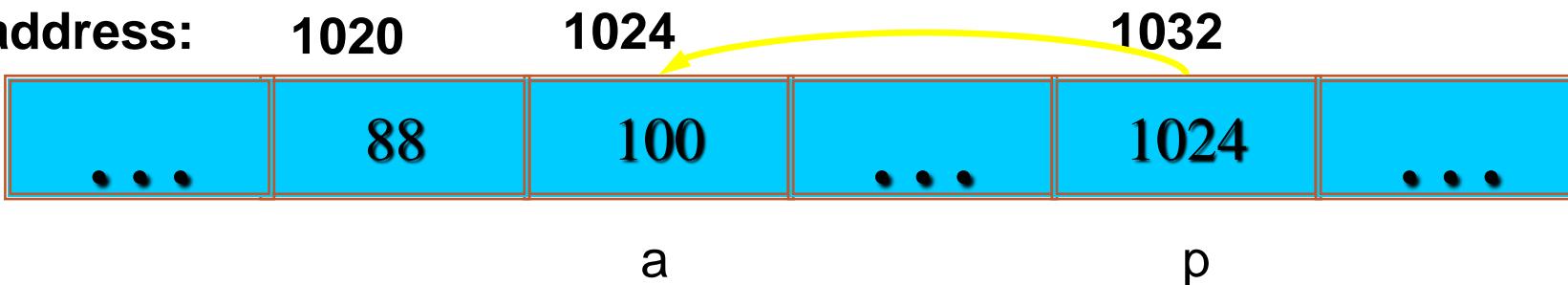
using namespace std;

void main() {
    int a, b;
    a = 88;
    b = 100;
    cout << "The address of a is: " << &a << endl;
    cout << "The address of b is: " << &b << endl;
}
```

□ Result is:
The address of a is: 1020
The address of b is: 1024

Pointer Variables

Memory address:

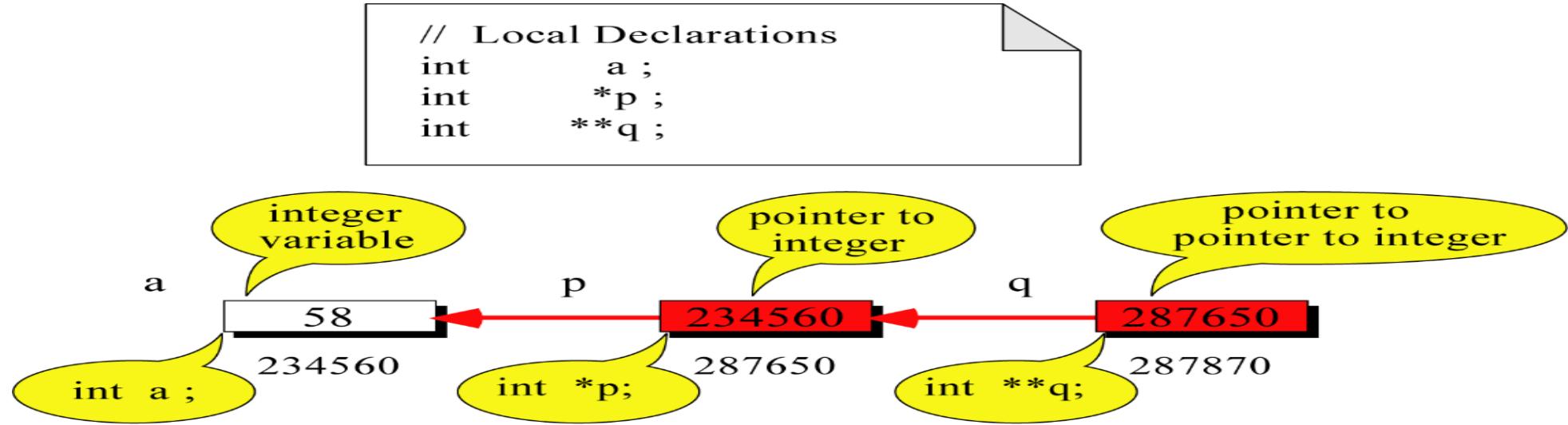


```
int a = 100;  
int *p = &a;  
cout << a << " " << &a << endl;  
cout << p << " " << &p << endl;
```

Result is:
100 1024
1024 1032

- ▶ The value of pointer p is the address of variable a
- ▶ A pointer is also a variable, so it has its own memory address

Pointer to Pointer



What is the output?

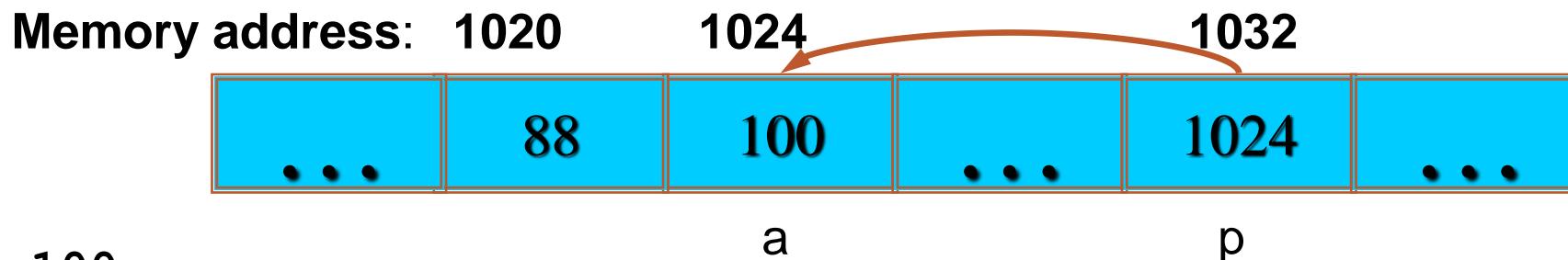
58 58 58

// Statements

```
a = 58 ;
p = &a ;
q = &p ;
cout <<      a << " ";
cout <<      *p << " ";
cout <<      **q << " ";
```

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

A Pointer Example

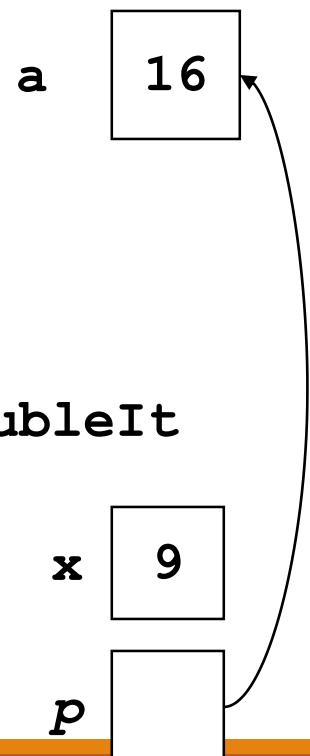
Output=a gets 18

The code

```
void doubleIt(int x, int * p)  
{  
    *p = 2 * x;  
}  
  
int main()  
{  
    int a = 16;  
    doubleIt(9, &a);  
    return 0;  
}
```

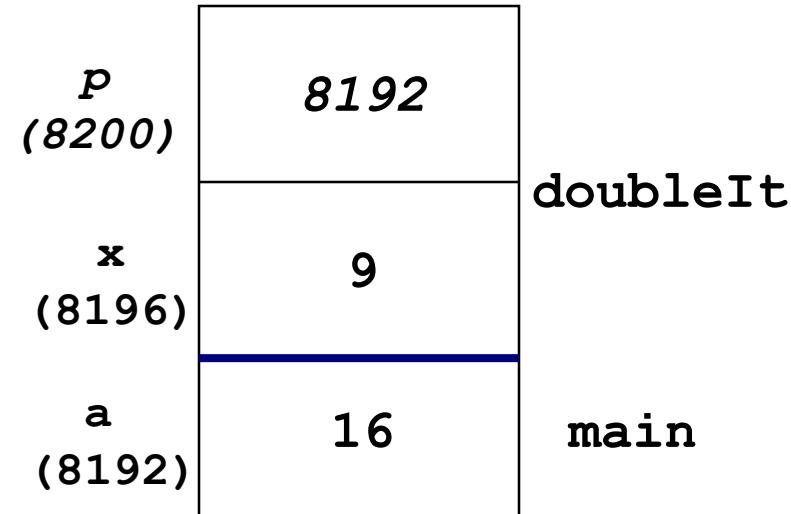
Box diagram

main



doubleIt

Memory Layout





Understanding Pointers

```
int main()
```

```
{  
    int firstvalue = 5, secondvalue = 15;  
    int * p1, *p2;  
    p1 = &firstvalue;           // p1 = address of firstvalue  
    p2 = &secondvalue;         // p2 = address of secondvalue  
    *p1 = 10;                 // value pointed by p1 = 10  
    *p2 = *p1;                // value pointed by p2 = value pointed by p1  
    p1 = p2;                  // p1 = p2 (value of pointer is copied)  
    *p1 = 20;                 // value pointed by p1 = 20  
    cout << "firstvalue is " << firstvalue;  
    cout << "secondvalue is " << secondvalue;  
    return 0;  
}
```

Pointer Usage (pass by value)

```
void IndirectSwap(int p, int q){  
    int temp = p;  
    p = q;  
    q= temp;  
    cout << p << q << endl;  
}  
  
int main() {  
    int a = 10;  
    int b = 20;  
    IndirectSwap(a, b);  
    cout << a << b << endl;  
    return 0;  
}
```

Output:

p=20, q=10

a=10, b=20

Pointer Usage (pass by address)

```
void IndirectSwap(char *Ptr1, char *Ptr2){  
    char temp = *Ptr1;  
    *Ptr1 = *Ptr2;  
    *Ptr2 = temp;  
}
```

```
int main() {  
    char a = 'y';  
    char b = 'n';  
    IndirectSwap(&a, &b);  
    cout << a << b << endl;  
    return 0;  
}
```

Output:
a=n
b=y

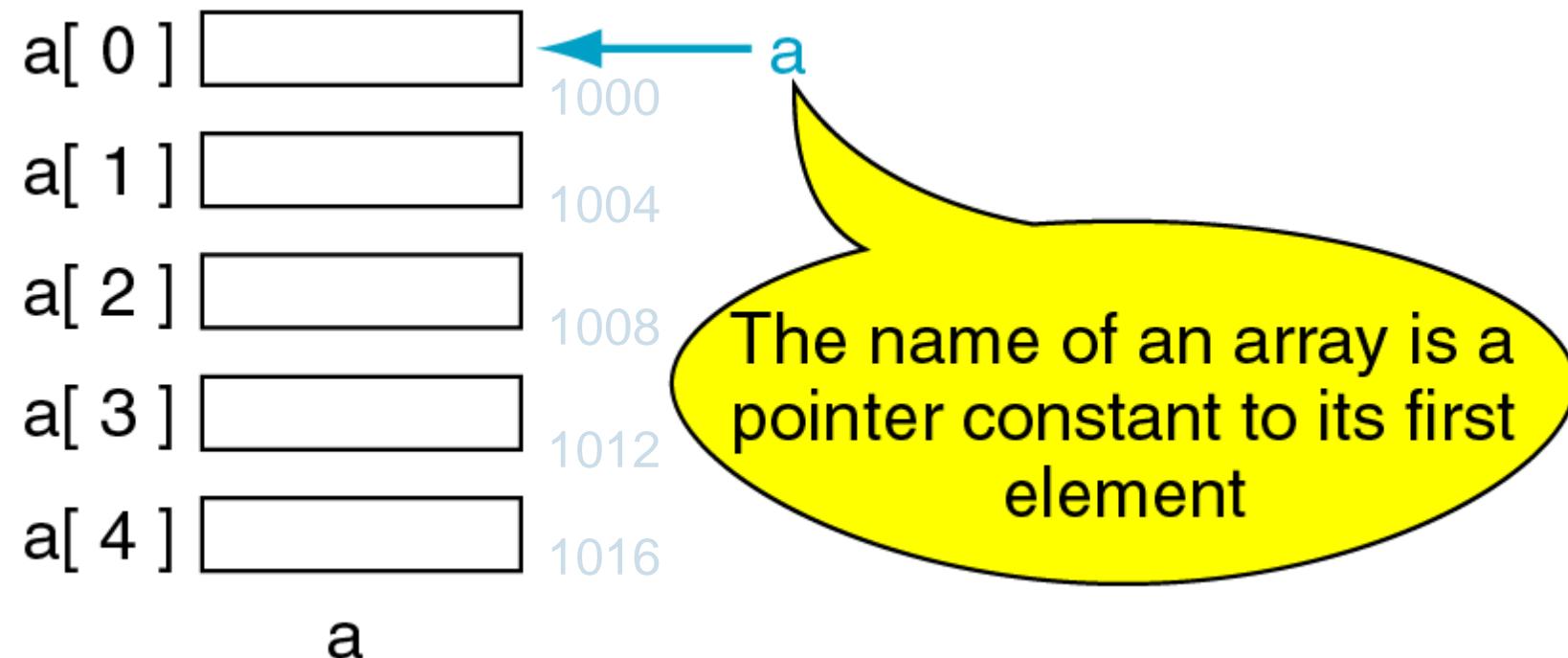
Pass by Reference (only in C++)

```
void IndirectSwap(char & x, char & y) {  
    char temp = x;  
    x = y;  
    y = temp; }  
  
int main() {  
    char a = 'm';  
    char b = 'n';  
    IndirectSwap(a, b);  
    cout << a << b << endl;  
    return 0;  
}
```

Output:
a=n
b=m

Pointers and Arrays

- The name of an array points only to the first element not the whole array.





Pointers and Arrays I

- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- An array **x** is declared as follows and assume the base address of **x** is 1000.
int x[5]={1,2,3,4,5};
- Array x, is a constant pointer, pointing to the first element **x[0]**. Value of **x** is **1000** (Base Address), the location of **x[0]** i.e. **x = &x[0] = 1000**

Elements	x[0]	x[1]	x[2]	x[3]	x[4]
Value	1	2	3	4	5
Address	1000 ↑ Base Address	1004	1008	1012	1016



Pointers and Arrays II (pointer to array)

- Use a pointer to an array, and then use that pointer to access the array elements.
- An integer pointer variable p, can be made to point to an array as follows:

int x[5] = { 1,2,3,4,5};

int *p;

p = x; OR p = &x[0];

p = &x ;//Invalid

- Successive array elements can be accessed by writing:

cout<< *p; p++; or

cout <<*(p+i); i++;

```
#include<stdio.h>
void main()
{
    int x[3] = {7, 9, 45};
    int *p = x;
    for (int i = 0; i < 3; i++)
    { printf("%d", *p);
        p++;
    }
    return 0;
}
```

***(p+i)//pointer
with an array is
same as x[i]**



Pointers and Arrays III

- Address of an element of x is given by:

Address of $x[i] = \text{base address} + i * \text{scale factor of (int)}$

Address of $x[3] = 1000 + (3 * 2) = 1006$

- The relationship between p and x is shown below.

$p = \&x[0] (=1000)$

$p+1 = \&x[1] (=1002)$

$p+2 = \&x[2] = 1004$

$p+3 = \&x[3] (=1006)$

$p+4 = \&x[4]$

- $*(p+3)$ is equivalent to $x[3]$.**

Array Name is a pointer constant

```
#include <iostream>
using namespace std;

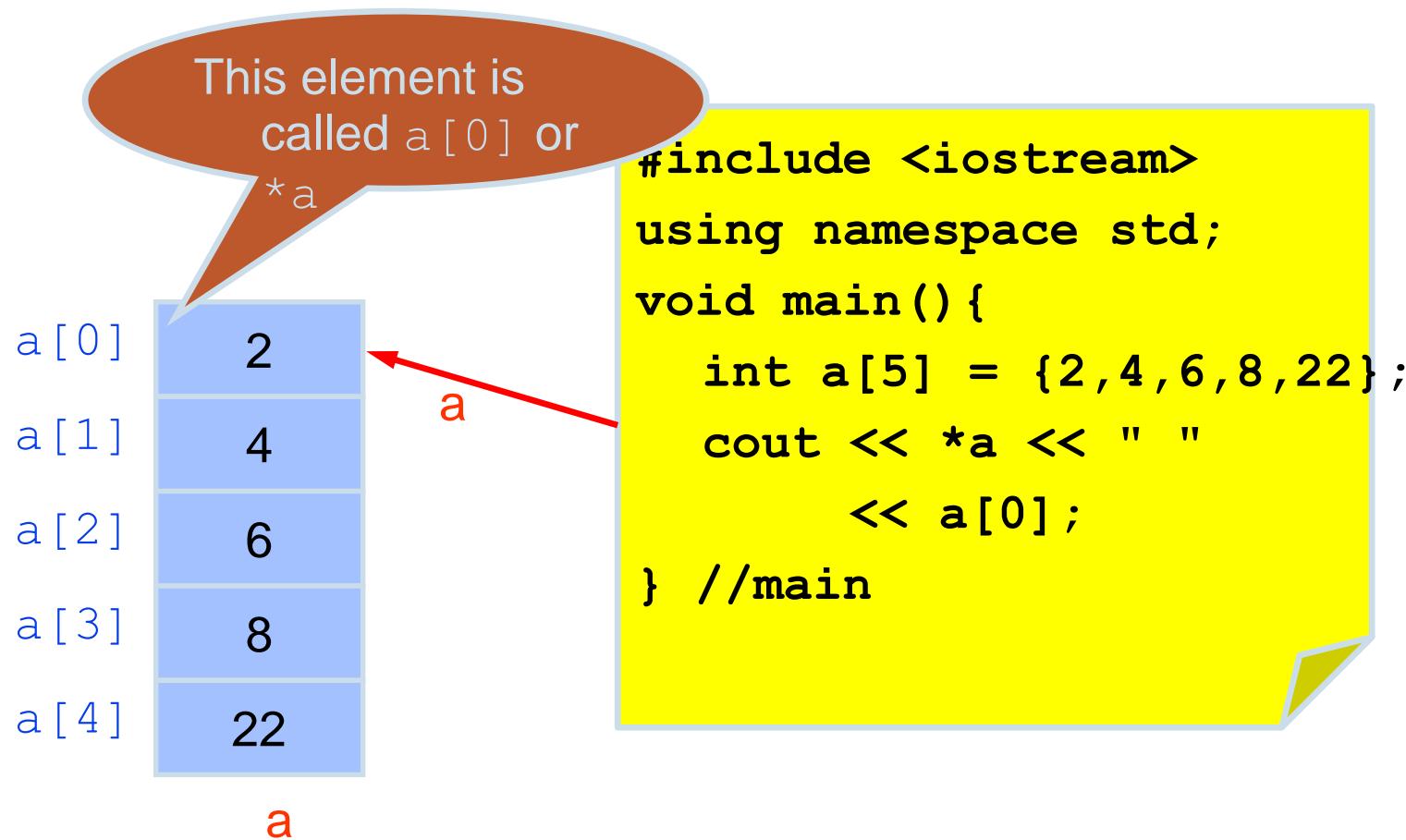
void main (){
    int a[5];
    cout << "Address of a[0]: " << &a[0] << endl
        << "Name as pointer: " << a << endl;
}
```

Result:

Address of a[0]: 0x0065FDE4

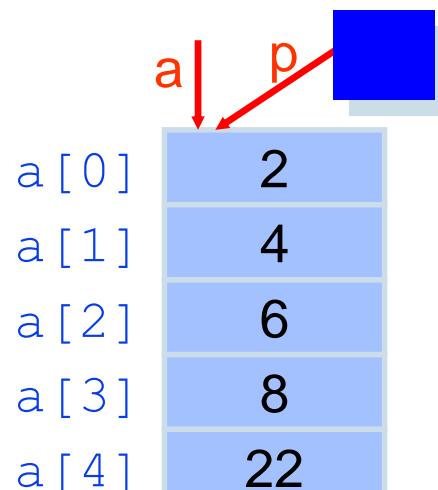
Name as pointer: 0x0065FDE4

Dereferencing An Array Name



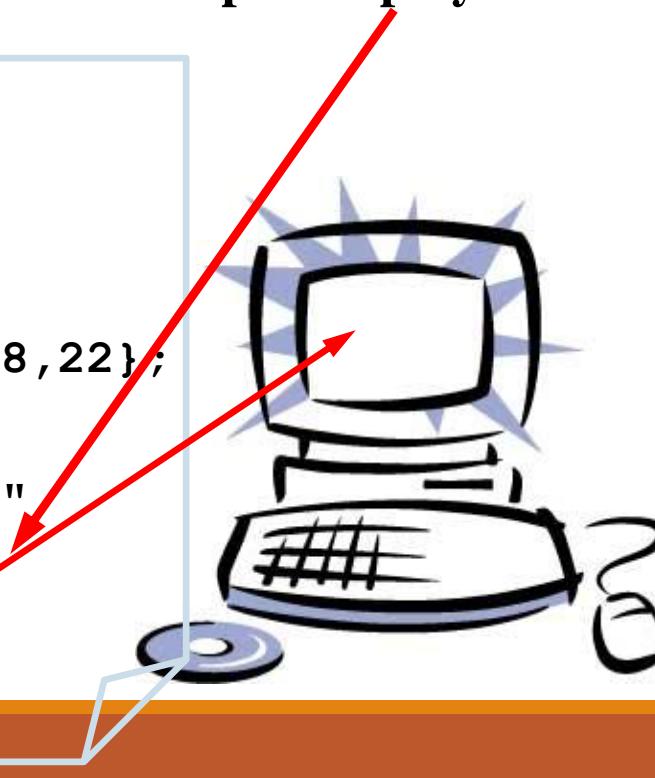
Array Names as Pointers

- To access an array, any pointer to the first element can be used instead of the name of the array.



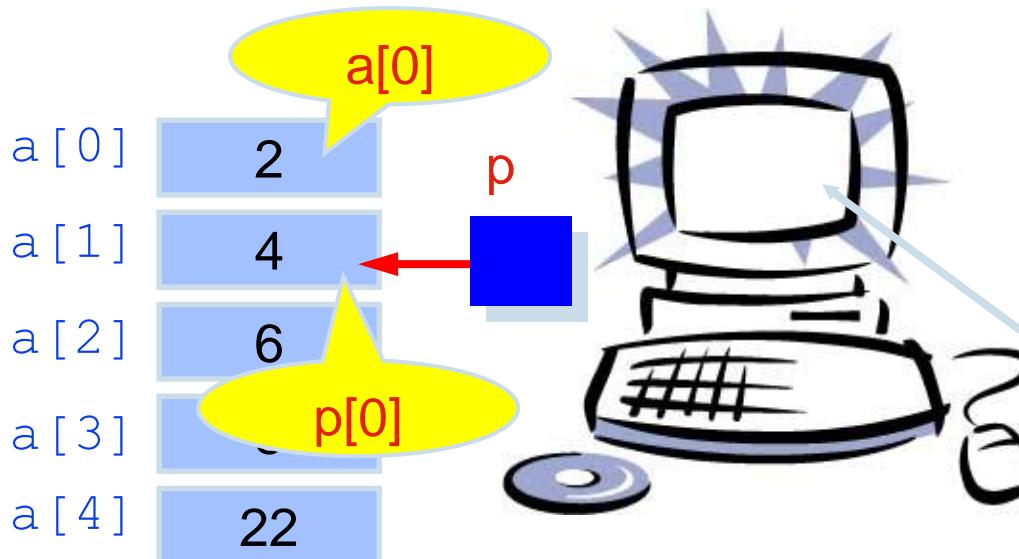
```
#include <iostream>
using namespace std;
void main(){
    int a[5] = {2,4,6,8,22};
    int *p = a;
    cout << a[0] << " "
        << *p;
}
```

We could replace `*p` by `*a`



Multiple Array Pointers

- Both a and p are pointers to the same array.



```
#include <iostream>
using namespace std;
void main() {
    int a[5] = {2,4,6,8,22};
    int *p = &a[1];
    cout << a[0] << " "
        << p[-1];
    cout << a[1] << " "
        << p[0];
}
```



Operations on Pointers I

- Pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then following are valid:
 - y= *p1* *p2 ; same as (*p1) * (*p2)**
 - sum=sum+ *p1;**
 - z=5 * - *p2 / *p1; same as (5 * (- (*p2)))/(*p1)**
 - Note:-There is a blank space between / and *; otherwise treated as comment.**
 - *p2 =*p2 +10;**
 - Addition of integers: **p1 + 2 //address increments by 4 bytes**
 - Subtraction of integers from pointers: **p2 - 2 //address decrements by 4 bytes**



Operations on Pointers II

- Subtraction of one pointer from another:

p1 - p2 //difference between two addresses

- Short hand operators may also be used with the pointers.

p1++;

- -p2;

Sum+=*p2;

- Pointers can also be used to compare things using relational operators.

p1>p2; p1==p2;

p1!=p2;



Operations on Pointers III

- Invalid statements:
 - Pointers are not used in division and multiplication.
p1/*p2; p1*p2;
p1/3; not allowed!
 - Two pointers can not be added.
p1 + p2 is illegal.



Scale Factor

- A scale factor is an increment or decrement given to a pointer to memory address which changes the address of the pointer to next memory address.
- When we increment a pointer , its value is increased by the **length** of the **data type** that it points to. This **length** is called the **scale factor**.
- For example,
p1++; where p1 is a pointer pointing to some integer variable with an initial value, say **2800**, then after **p1 = p1 + 1**, the value of **p1** will be **2802**.
- We refer to this addition as **pointer arithmetic**.

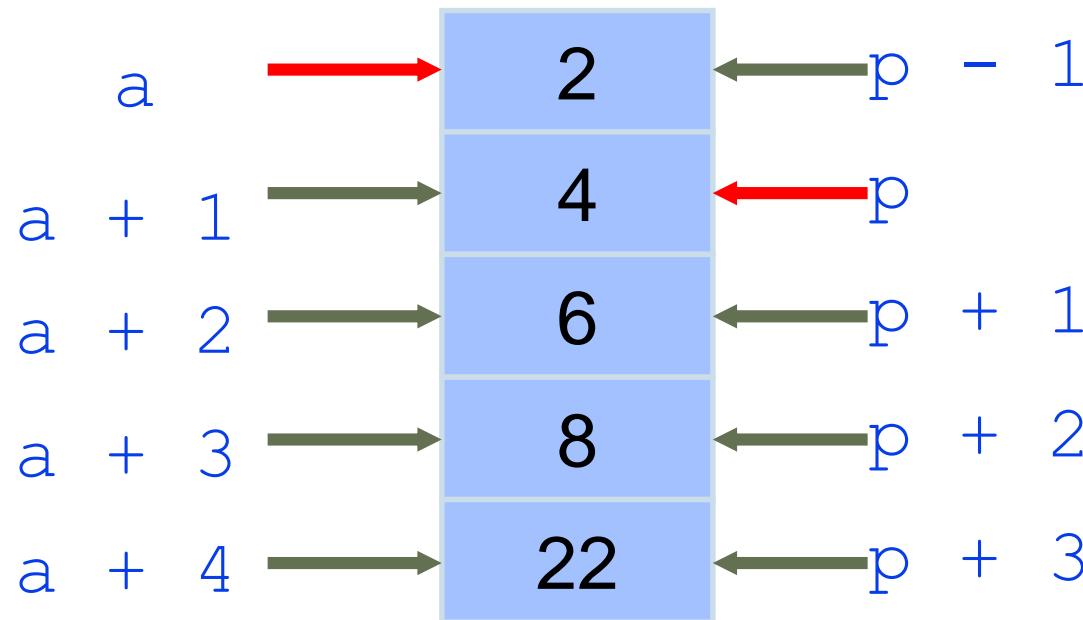


Valid and Invalid cases

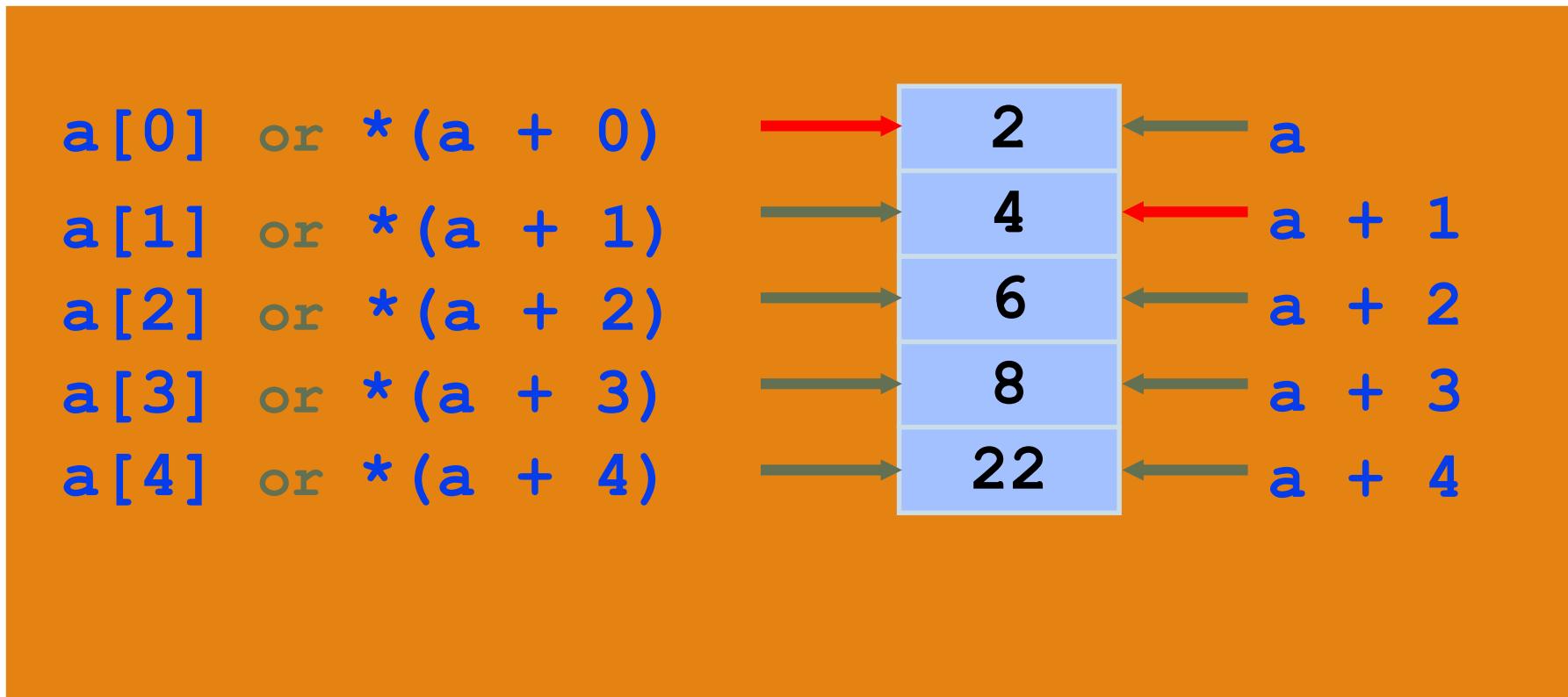
- Invalid Cases:
 - `&50` (pointing at constant)
 - `int x[10];`
`&x` (pointing at array name)
 - `&(x + y)` (pointing at expressions).
- If **x** is an array, then expressions such as **&x[0]** and **&x[i+3]** are valid and represent the addresses of **0th** and **(i+3)th** elements of **x**.

Pointer Arithmetic

- Given a pointer p , $p+n$ refers to the element that is offset from p by n positions.



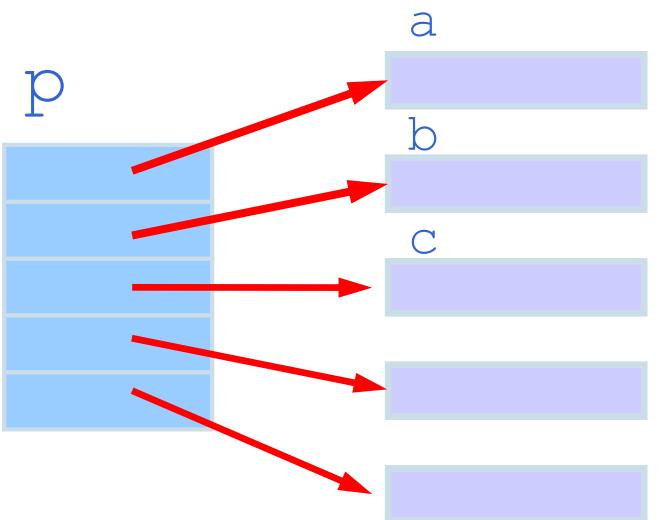
Dereferencing Array Pointers



$*(\text{a}+n)$ is identical to $\text{a}[n]$

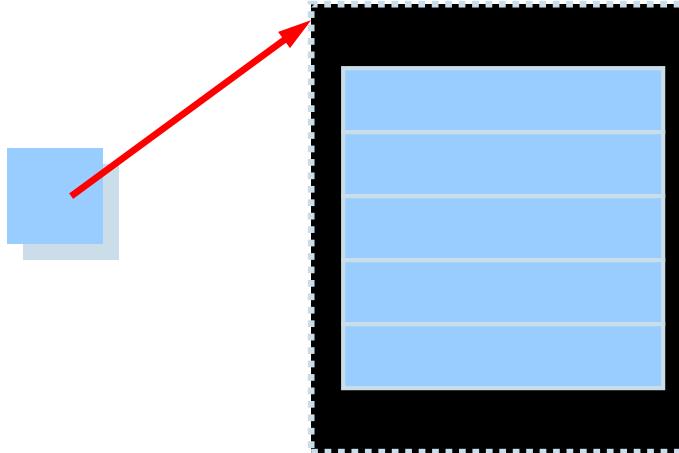
□ Note: flexible pointer syntax

Array of Pointers & Pointers to Array



An array of Pointers

```
int a = 1, b = 2, c = 3;  
  
int *p[5];  
  
p[0] = &a;  
p[1] = &b;  
p[2] = &c;
```



A pointer to an array

```
int list[5] = {9, 8, 7, 6, 5};  
  
int *p;  
  
P = list;//points to 1st entry  
P = &list[0];//points to 1st entry  
P = &list[1];//points to 2nd entry  
P = list + 1; //points to 2nd entry
```

NULL pointer

NULL is a special value that indicates an empty pointer

If you try to access a NULL pointer, you will get an error

```
int *p;  
p = 0;  
cout << p << endl; //prints 0  
cout << &p << endl;//prints address of p  
cout << *p << endl;//Error!
```



Example

```
#include <iostream.h>
void main()
{
    int arr[5] = { 31, 54, 77, 52, 93 };
    for(int j=0; j<5; j++) //for each element,
        cout << *(arr+j); //print value
}
```

”arr” itself is a constant pointer which can be used to access the elements.



Example

```
// array accessed with pointer
#include <iostream.h>
void main()
{
    int arr[] = { 31, 54, 77, 52, 93 }; //array
    int *ptr; //pointer to arr
    ptr = arr; //points to arr
    for(int j=0; j<5; j++) //for each element,
        cout << *(ptr++); //print value
}
```

”ptr” is a pointer which can be used to access the elements.



Problems

- Write a program using pointers to exchange two values (values stored in variables x & y).
- Write a program using pointers to find the sum of all elements stored in an integer array.



Pointers and Character Strings

- The statement **char *cptr =name;** declares cptr as a pointer to a character and assigns address of the first character of name as the initial value
- The statement **while(*cptr]!='\0')** is true until the end of the string is reached.
- When the while loop is terminated, the pointer **cptr** holds the address of the null character **[\0]**.
- The statement **length = cptr - name;** gives the length of the string name.
- A constant character string always represents a pointer to that string.
- The following statements are valid: **char *name; name="Delhi";**

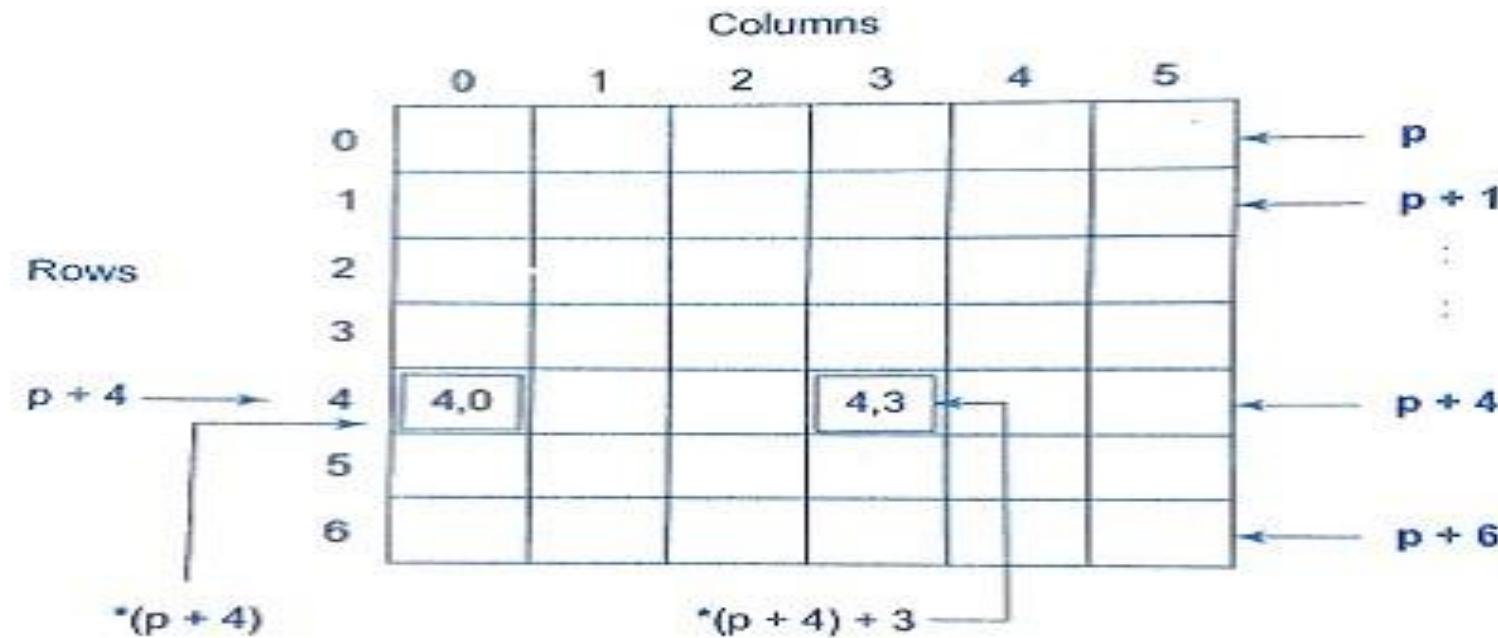


Pointers and 2D array I

```
int a[][2]={ {12,22},  
            {33,44} };  
  
int (*p)[2];  
p=a;    // initialization
```

- Element in 2d represented as
 $*(*(a+i)+j)$ or
 $*(*(p+i)+j)$

Pointers and 2D array II



- p → pointer to first row
- $p + i$ → pointer to i th row
- $*(p + i)$ → pointer to first element in the i th row
- $*(p + i) + j$ → pointer to j th element in the i th row
- $*(*(p + i) + j)$ → value stored in the cell (i,j)
(i th row and j th column)



Pointers and 2D array III

```
#include <iostream.h>
void main()
{
    int i,j,(*p)[2], a[][2]={{12, 22}, {33, 44}};
    p=a;
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
            cout<<*(*(p+i)+j)<<"\t";
        cout<<"\n";
    }
}
```



Array of Pointers I

- We can use pointers to handle a table of strings.

char name[3][25];

name is a table containing 3 names, each with a maximum length of 25 characters (including '`\0`')

Total storage requirement for name is **75 bytes**. But rarely all the individual strings will be equal in lengths.

- We can use a pointer to a string of varying length as

char *name[3] = { "New Zealand", "Australia", "India" };

2D Array for strings

- Not all strings are long enough to fill all the rows of the array, compiler fills these empty spaces with '\0'.
- The total size of the sports array is 75 bytes but only 34 bytes is used, 41 bytes is wasted.
- Therefore, we need a **jagged array**: A 2-D array whose rows can be of different length.

The `sports` array is stored in the memory as follows:

```
char sports[5][15] = {  
    "golf",  
    "hockey",  
    "football",  
    "cricket",  
    "shooting"  
};
```

sports[5][15]

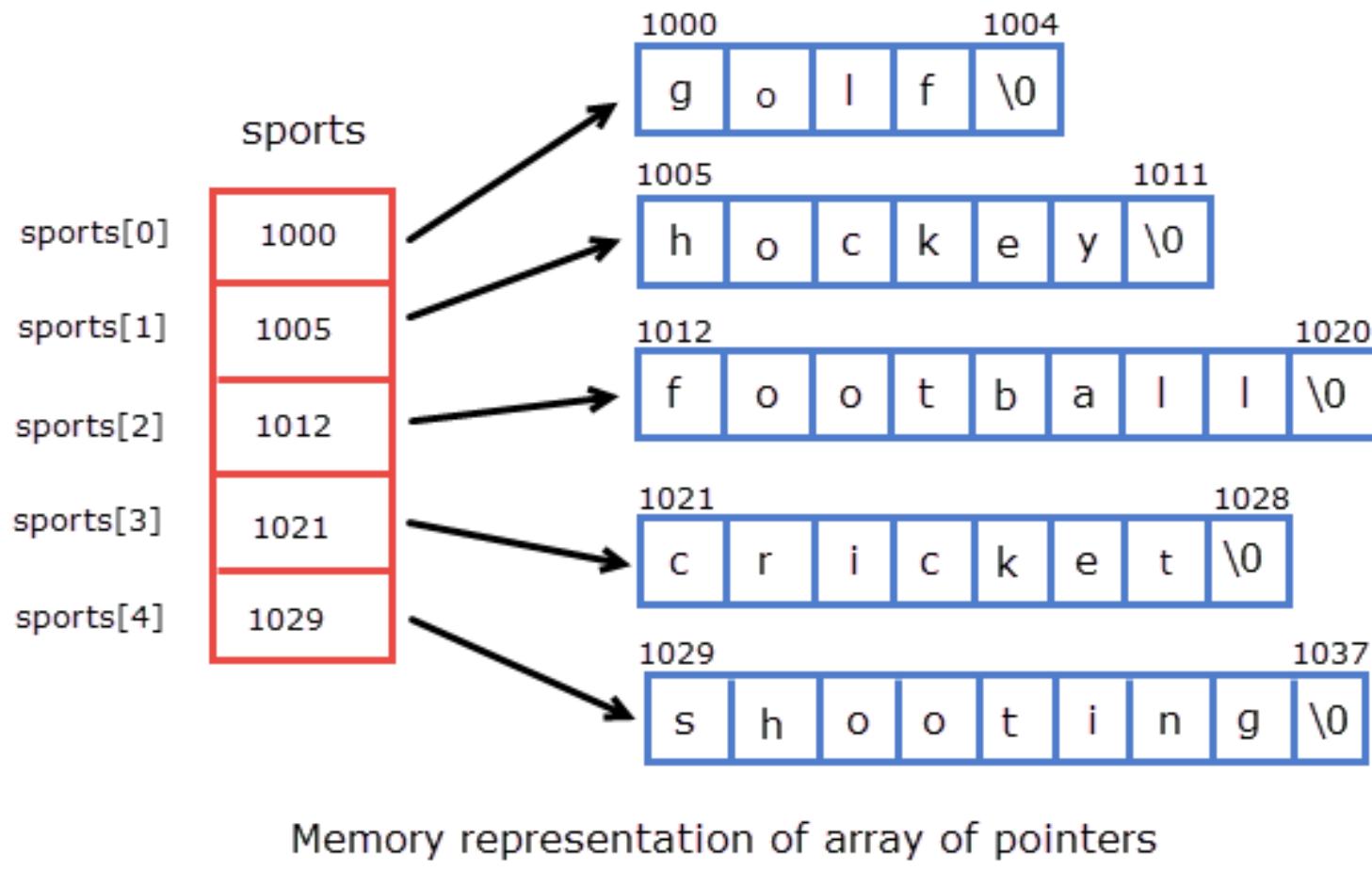
1000	g	o	l	f	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	1015
1016	h	o	c	k	e	y	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	1031
1032	f	o	o	t	b	a	i	i	\0	\0	\0	\0	\0	\0	\0	\0	\0	1047
1048	c	r	i	c	k	e	t	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	1063
1064	s	h	o	o	t	i	n	g	\0	\0	\0	\0	\0	\0	\0	\0	\0	1079

Memory representation of an array of strings or 2-D array of characters

The 0th element i.e **arr[0]** points to the base address of string "golf".

Similarly, the 1st element i.e **arr[1]** points to the base address of string "hockey" and so on.

Here is how an array of pointers to string is stored in memory.





Array of Pointers II

- Declares name to be an array of 3 pointers to characters, each pointer pointing to a particular name.

name[0]→New Zealand

name[1]→Australia

name[2]→India

This declaration allocates **28 bytes**. - **Ragged Array**

- The following statement would print out all the 3 names.

```
for(i=0; i<=2;i++)  
    cout<<name[i]; or *(name + i);
```

To access the j^{th} character in the i^{th} name, we may write as ***(name[i] + j)**

Array of Pointers to Strings

- An array of pointers to strings is an array of character pointers where each pointer points to the first character of the string or the base address of the string.

- To declare and initialize an array of pointers to strings

```
char *sports[5] = { "golf", "hockey", "football", "cricket", "shooting" };
```

- If initialization of an array is done at the time of declaration then the size of an array can be omitted

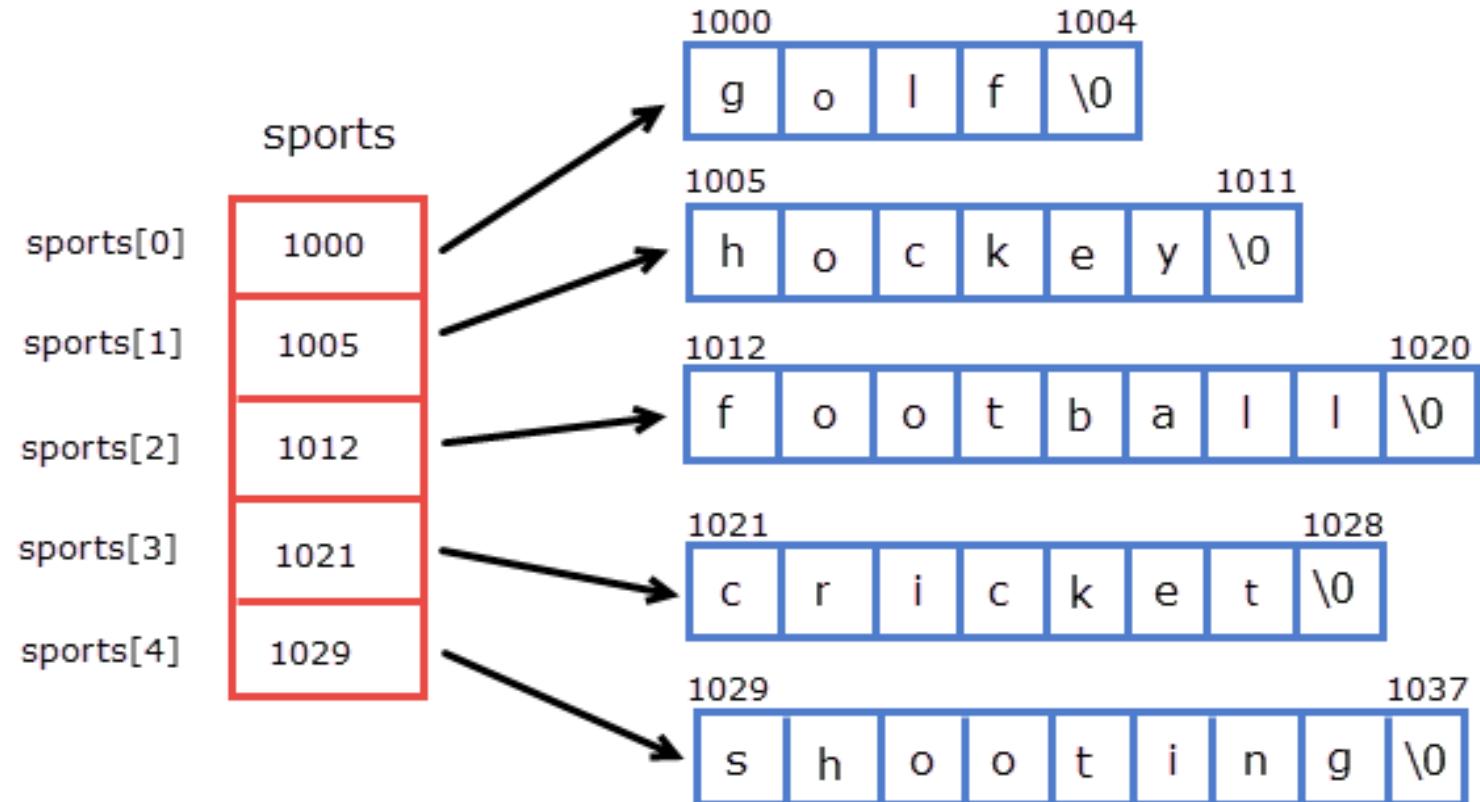
```
char *sports[ ] = { "golf", "hockey", "football", "cricket", "shooting" };
```

- Space utilized is : $34 + 20 = 54$

In this case, all string literals occupy **34 bytes** and **20 bytes** are occupied by the array of pointers **i.e sports**. So, just by creating an array of pointers to string instead of 2-D array of characters **21 bytes** ($75-54=21$) of memory is saved.

- The 0th element i.e **arr[0]** points to the base address of string "**golf**".
- The 1st element i.e **arr[1]** points to the base address of string "**hockey**" and so on.
- An array of pointers to string is stored in memory in this following manner:

```
char *sports[] = {
    "golf",
    "hockey",
    "football",
    "cricket",
    "shooting"
};
```



Memory representation of array of pointers

Structures

Structure

Structure is a collection of related elements, possibly of different types, having a single name

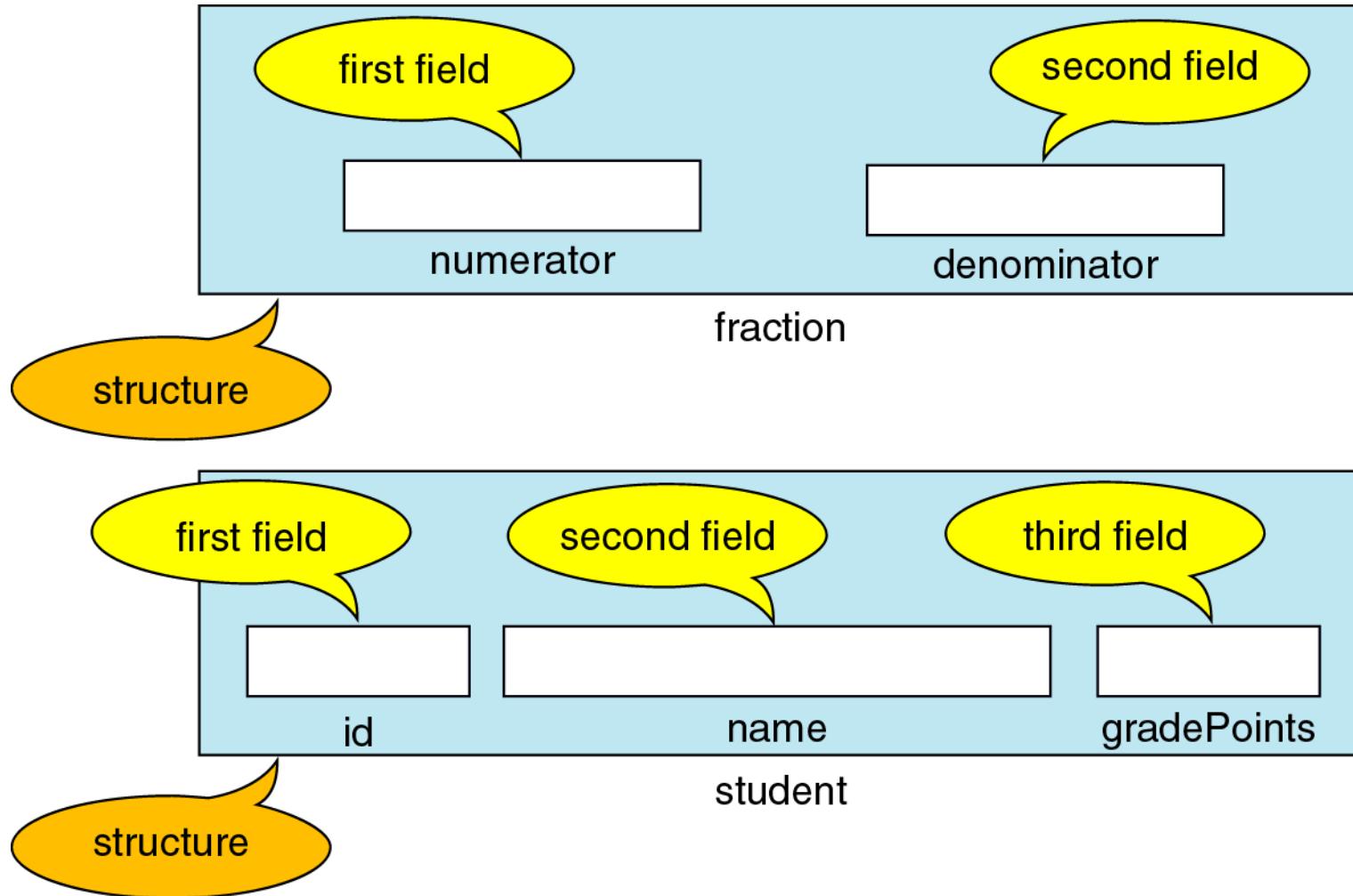
Field – each element in a structure is called a field

- Same as variable in that it has type and exists in memory
- Differs from a variable in that it is a part of structure

Structure vs. array

- Both are derived data types that can hold multiple pieces of data
- All elements in an array must be of the same type, while the elements in a structure can be of the same or different types

Figure 12-6 Structure Examples



Note: the data in a structure should all be related to one object

Ex1: both integers belong to the same fraction

Ex2: all data relate to one student

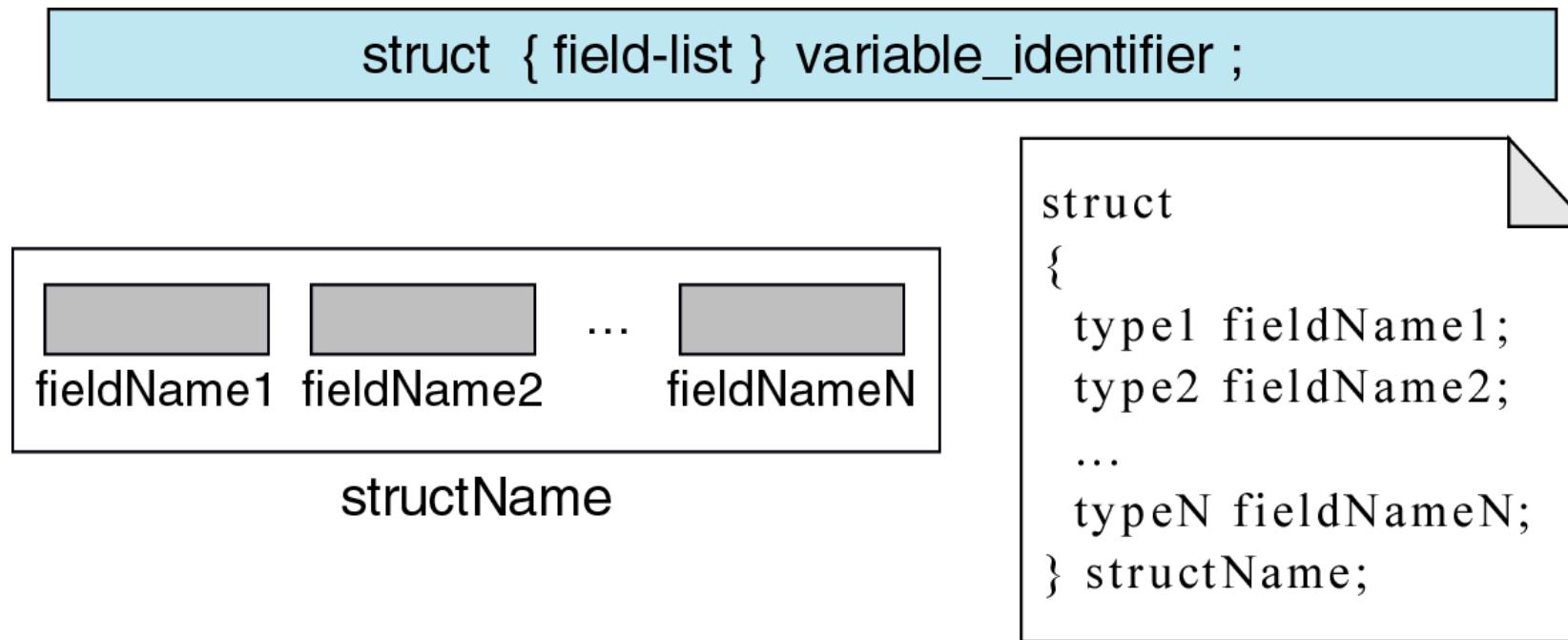
Structure – declaration and definition

Keyword struct – informs the compiler that it is a collection of related data

3 ways to declare/define a structure in C

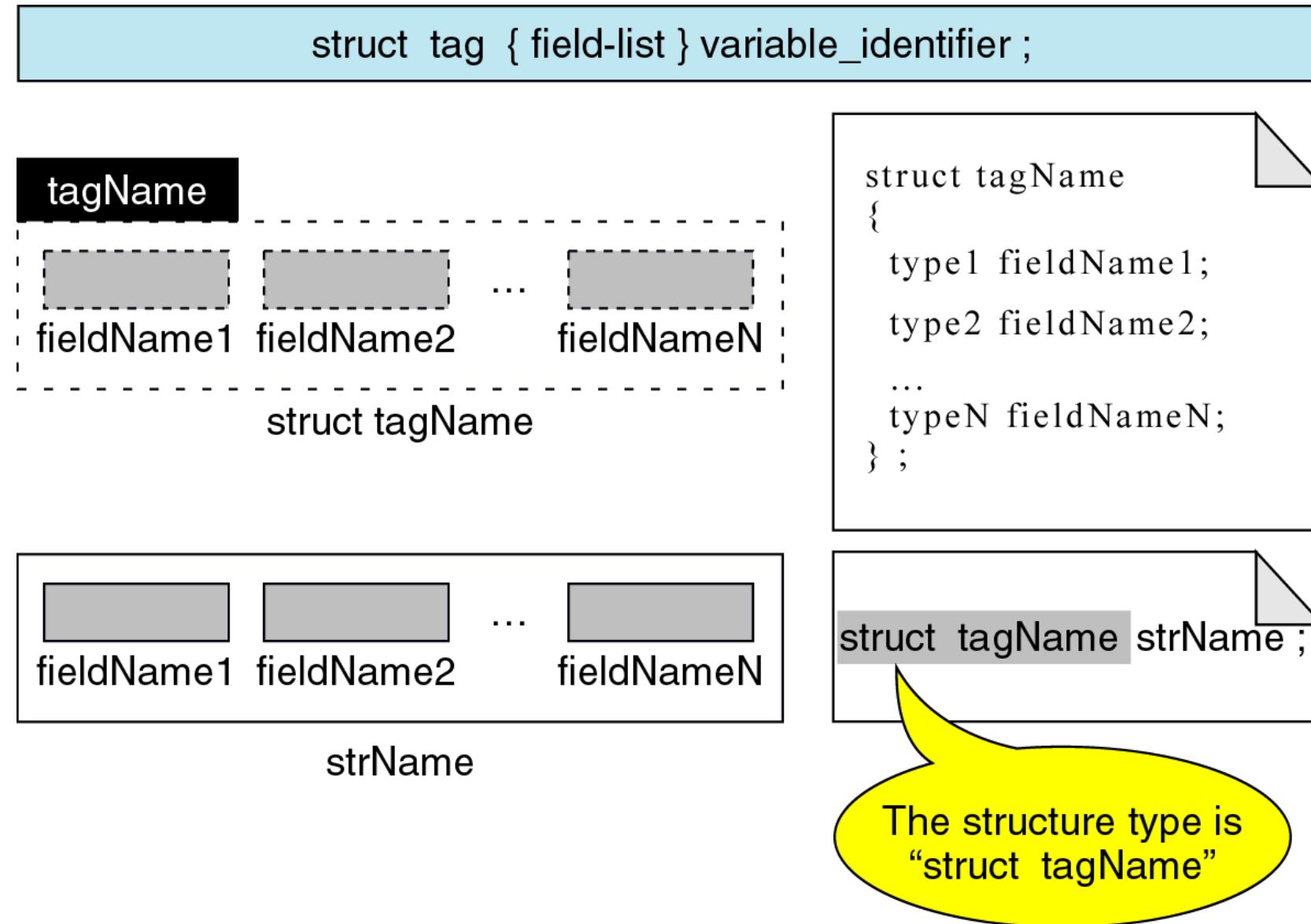
1. Structure variable
2. Tagged structure
3. Type-defined structure

Figure 12-7 structure variable



- **Note:** The above definition creates a structure for only one variable definition
- As there is no structure identifier (tag), it cannot be shared
- So, it is not really a type
- This type of declaration format is not to be used

Figure 12-8 Tagged structure



Tagged Structure – declaration and definition

struct tagname – tagname is the identifier for the structure

- allows to define variables, parameters, and return types

If a struct is concluded with a semicolon after the closing brace, no variables are defined

- So structure is a type template with no associated storage

To define a variable at the same time we declare the structure, list the variables by comma separation

Tagged Structure – declaration and definition

Ex: declare and use student structure:

```
struct student{  
    char id[10];  
    char name[20];  
    int gpa;  
};  
struct student astudent;  
void print_Student(struct student stu);
```

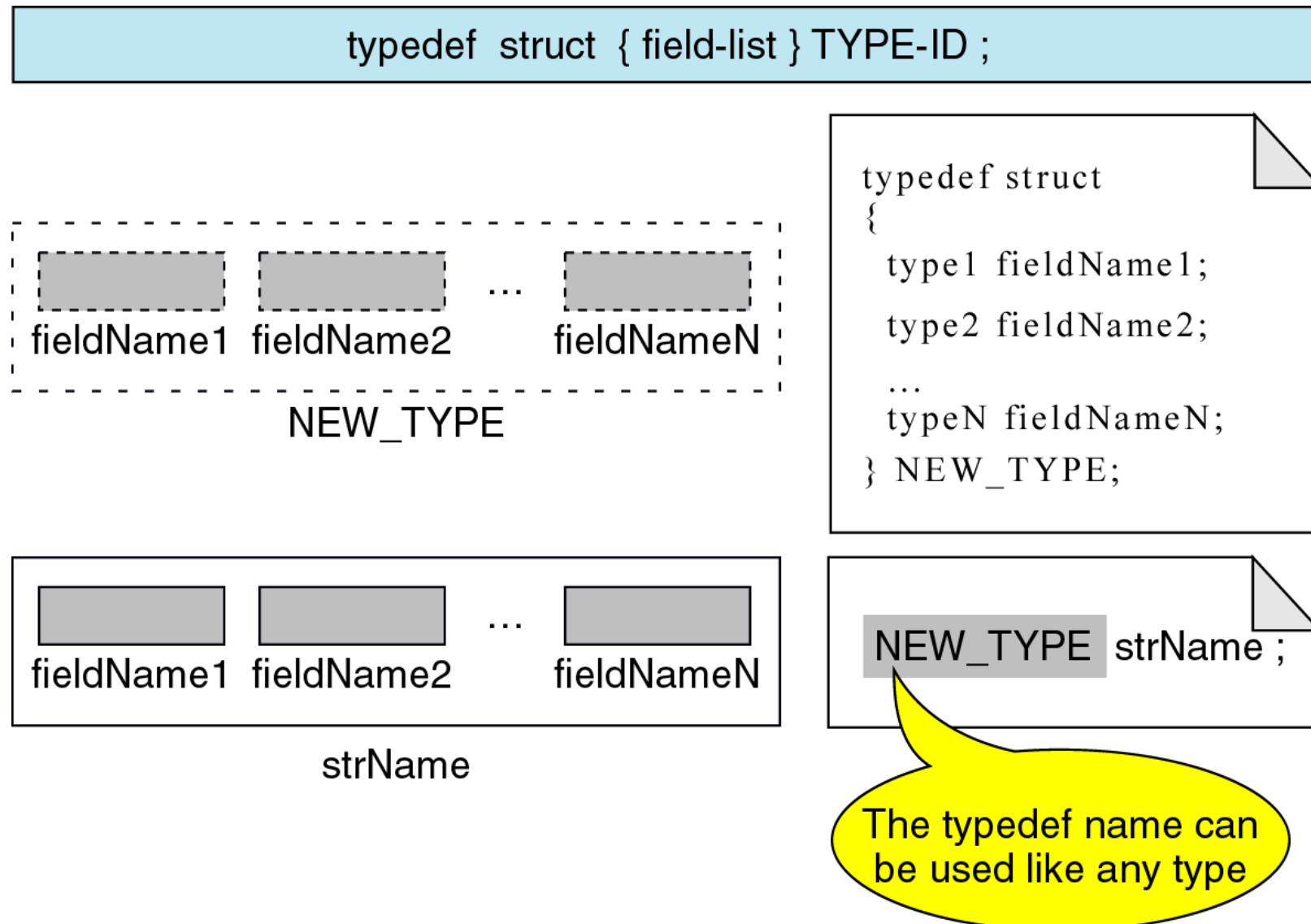
Tagged Structure – declaration and definition

```
struct student{  
    char id[10];  
    char name[20];  
    int gpa;  
}; struct student astudent;  
void print Student( struct student stu);
```

Follow the above format of declaring structure first and then define variables

- Declare structure declaration in the global area of the program before main
- So structure declaration scope is global and can be shared by all functions

Figure 12-9 Type defined structure



Type defined Structure – declaration and definition

Ex: declare and use student structure:

```
typedef struct {
    char id[10];
    char name[20];
    float gpa;
} STUDENT;
STUDENT astudent;
void printStudent(STUDENT stu);
```

Type defined Structure – declaration and definition

The most powerful way to declare a structure is by **typedef**

Typedef is to be added before struct

- Identifier at the end of the block is the type definition name not a variable
- Cannot define a variable with the typedef declaration

```
typedef struct {  
    char id[10];  
    char name[20];  
    float gpa;  
} STUDENT;  
STUDENT astudent;  
void print Student(STUDENT stu);
```

Type defined Structure – declaration and definition

It is possible to combine the tagged structure and type definition structure in a tagged type definition

The difference between tagged type definition and a type defined structure is

- Here, the structure has a tag name in tagged type definition

```
typedef struct tag {
    char id[10];
    char name[20];
    float gpa;
} STUDENT;
STUDENT astudent;
void printStudent(STUDENT stu);
```

Figure 12-10 struct format variations

```
struct {  
    ...  
} variable_identifier;
```

structure variable

```
struct tag  
{  
    ...  
} variable_identifier;
```

```
struct tag variable_identifier;
```

tagged structure

```
typedef struct  
{  
    ...  
} TYPE_ID;
```

```
TYPE_ID variable_identifier;
```

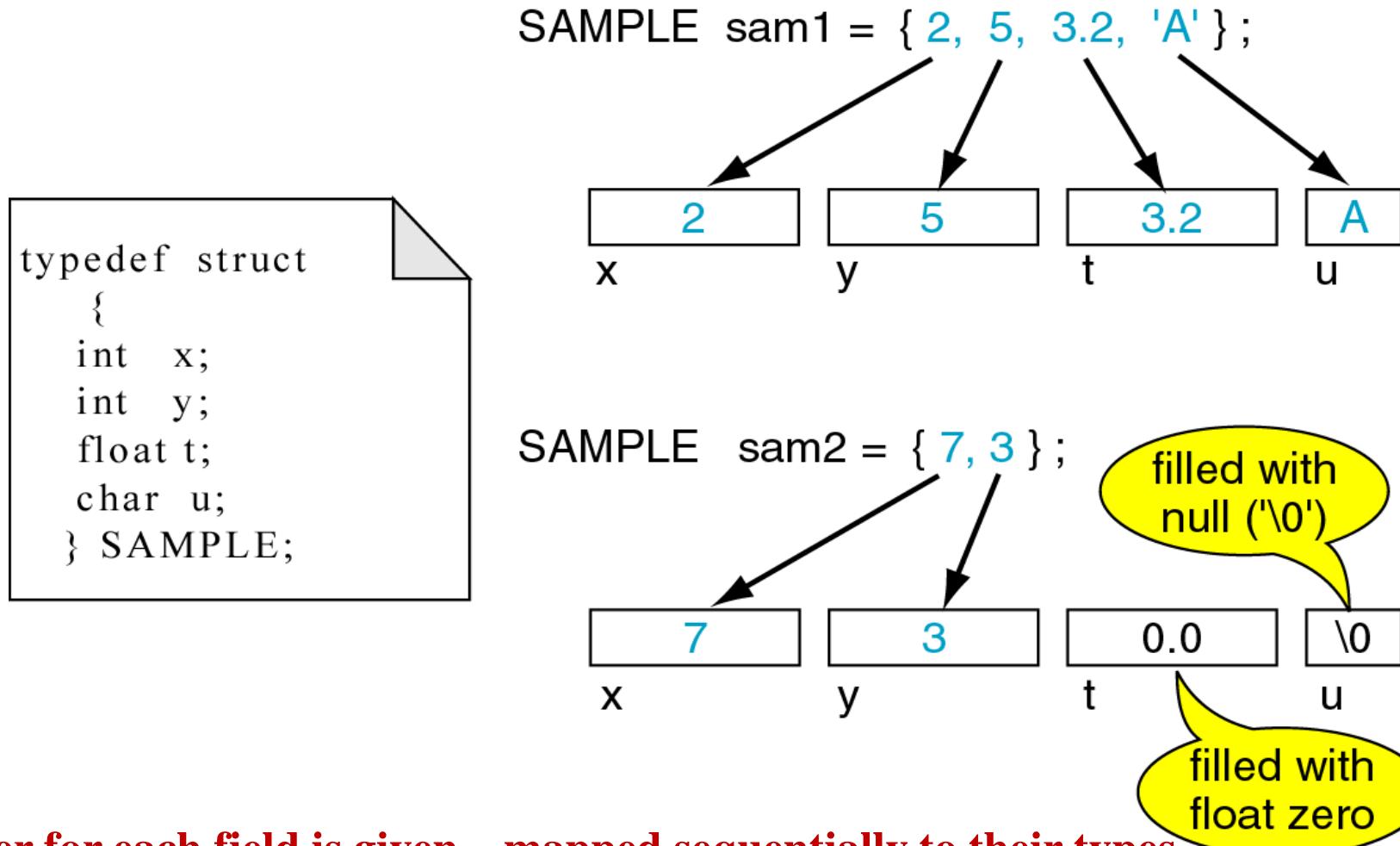
type-defined structure

Initializing structures

Rules for structure initialization are similar to rules of array initialization

- The initializers are enclosed in braces and comma separated
- They must match their corresponding types in the structure definition
- For a nested structure, the nested initializers must be enclosed in their own set of braces

Figure 12-11 Initializing structures



- **Ex1:** initializer for each field is given – mapped sequentially to their types
- **Ex2:** initializer for some fields are only given – structure elements will be assigned null values – zero for integers and floating point numbers, \0 for chars and strings (same as in arrays)

Accessing structures

- Same way as we manipulate variables using expressions and operators, the structure fields can also be operated

- **Example:**

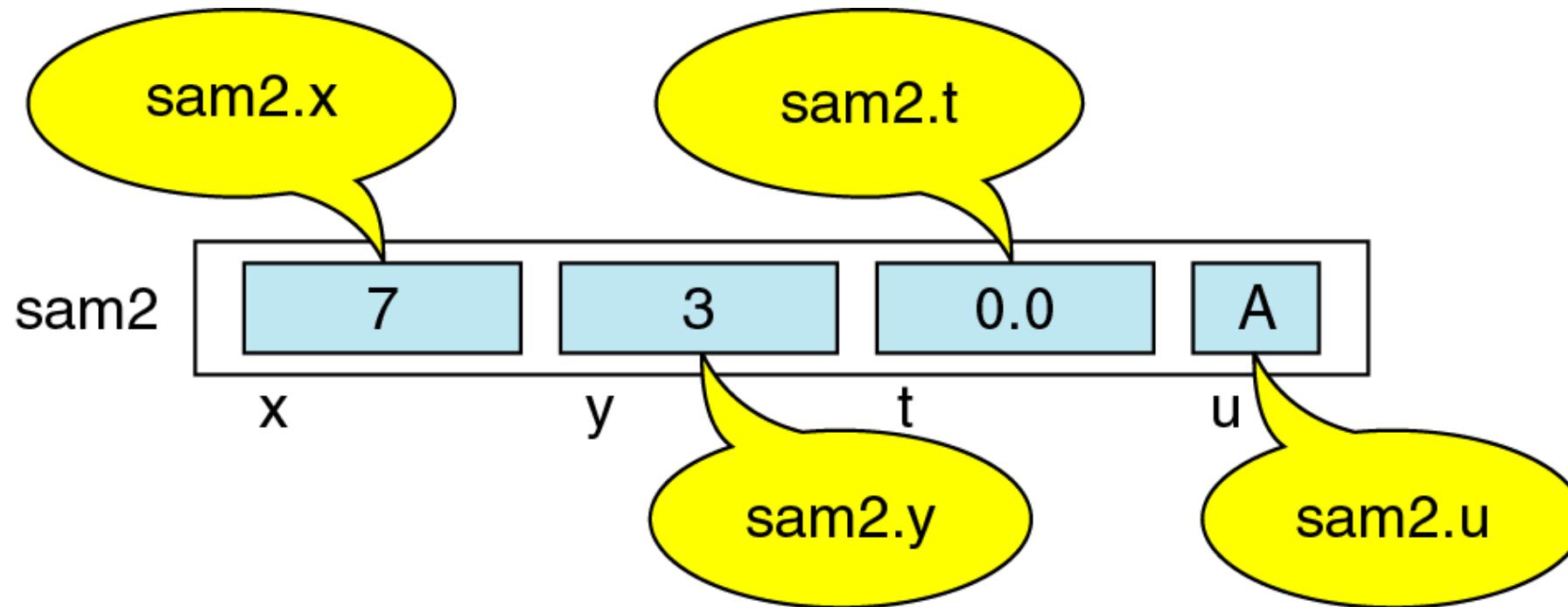
```
typedef struct {  
    char id[10];  
    char name[20];  
    float gpa;  
} STUDENT;
```

```
STUDENT astudent;
```

- Refer to fields by

```
astudent.id, astudent.name, astudent.gpa
```

Figure 12-12 structure member operator



Ex: Reading data into and writing data from structure members is same as done for variables

```
scanf("%d %d %f %c", &sam2.x, &sam2.y, &sam2.t, &sam2.u);
```

Precedence of Member operator

Similar to **operator []** for array indexing, **dot** is member operator for structure reference

- 1) The precedence of member operator is higher than that of increment

Ex: **sam2.x++;** **++sam2.x;**

No parentheses are required

- 2) **&sam1.x is eqt to &(sam1.x).** So **dot** has higher priority than **&** operator

Operations on Structures

Assignment operation : assigning one structure to another

Structure is treated as one entity and only one operation i.e., assignment is allowed on the structure itself

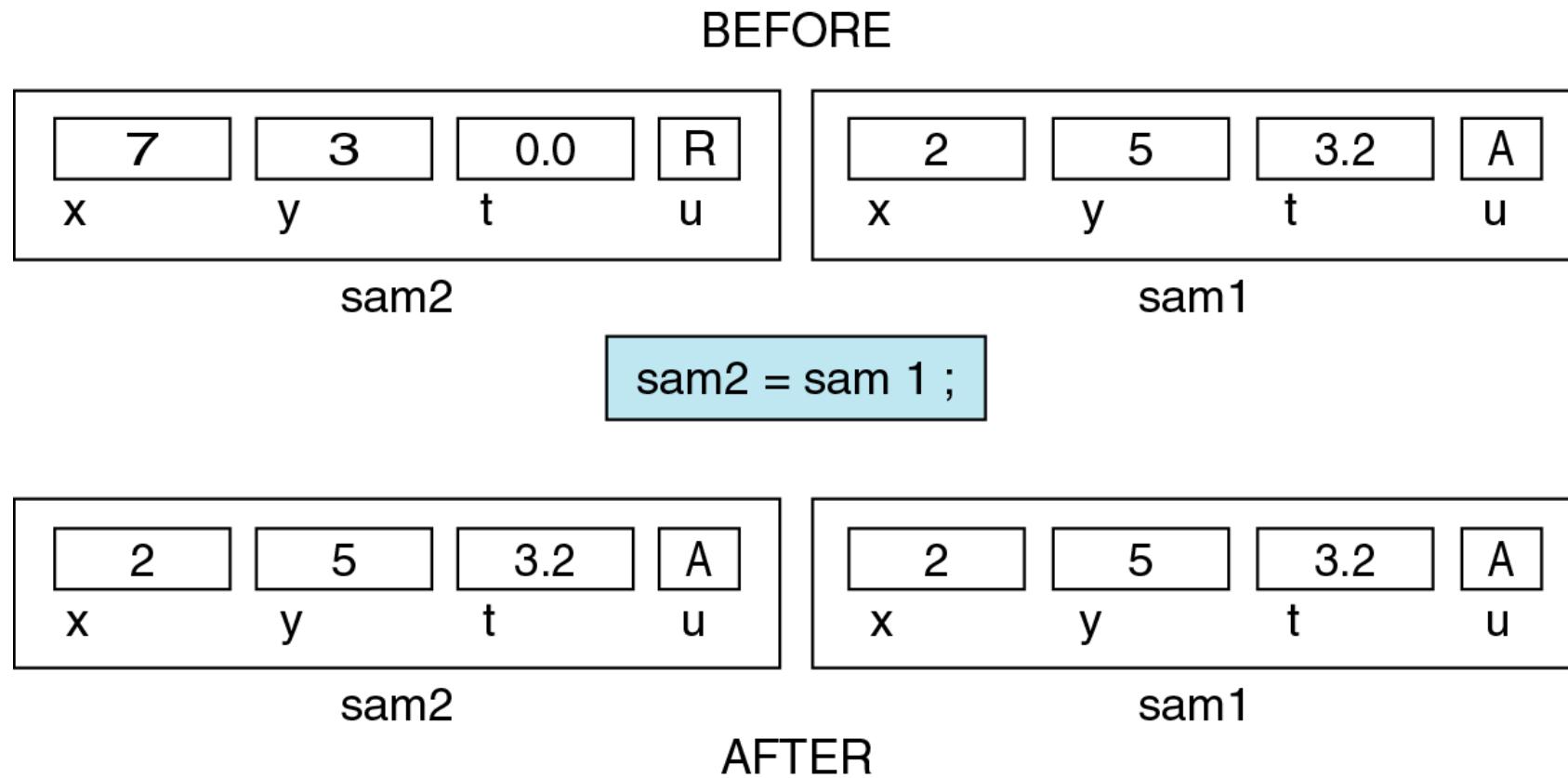
To copy one structure to another structure of the same type

- Rather than assigning individual members
- Assign one to another

Ex: read values into sam1 from the keyboard. Now copy sam1 to sam2 by

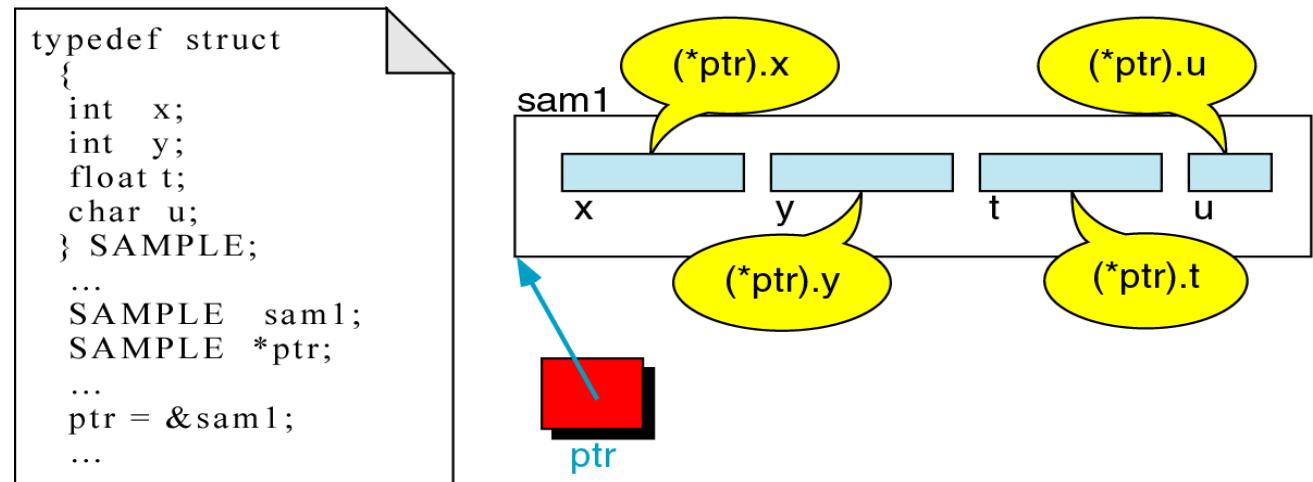
sam2=sam1;

Figure 12-13 Copying a structure

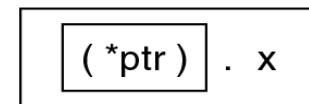


Pointer to structures

Like other types, structures can also be accessed through pointers



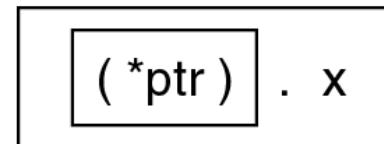
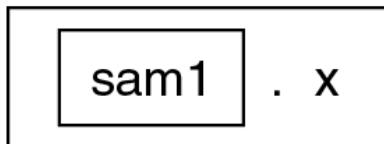
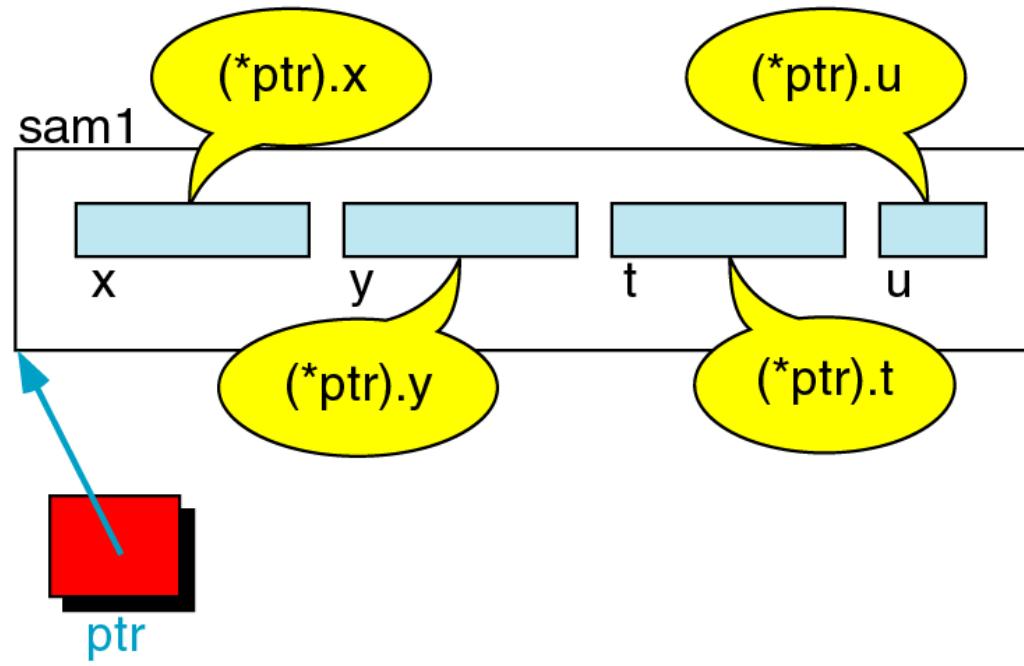
- **Accessing structure itself by `*ptr`**
- **ptr contains the address of the beginning of the structure**
- **Now we do not only need to use structure name with member operator such as `sam1.x`**
- **we can also use `(*ptr).x`**



Two ways to reference x

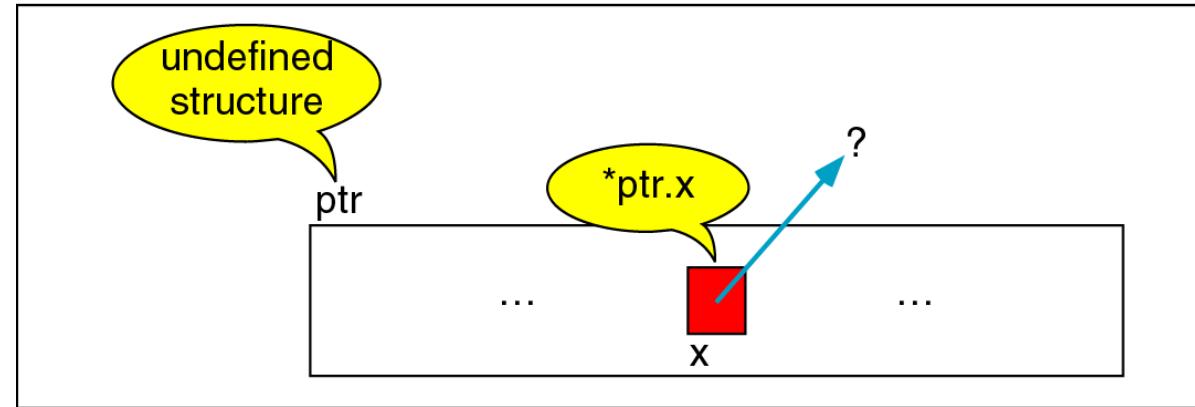
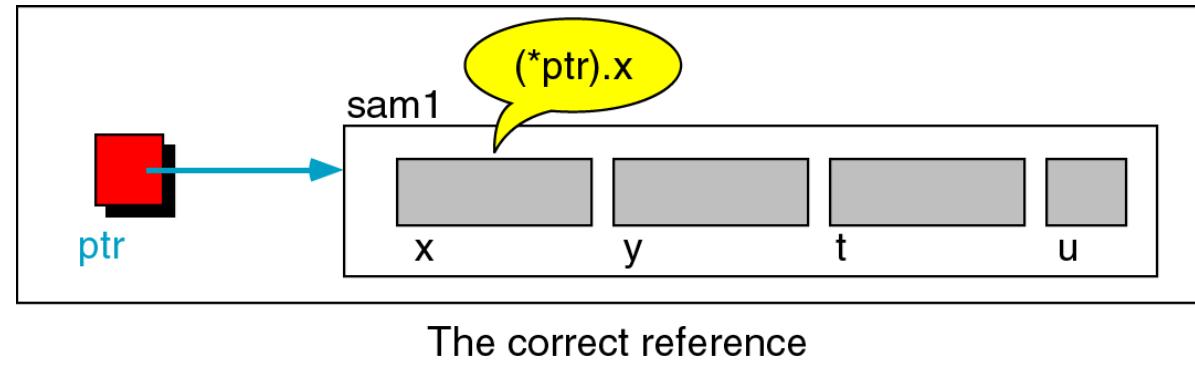
Figure 12-14 Pointers to structures

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



Two ways to reference x

Figure 12-15 Interpretation of invalid pointer use



The wrong way to reference the component

- In the expression `(*ptr).x`, parentheses are necessary as member operator has more priority than indirection operator
- Default interpretation of `*ptr.x` is `*(ptr.x)` which is error because it means that there is a structure called `ptr` (undefined here) containing a member `x` which must be a pointer
- So, a compile-time error is generated as it is not the case

Selection operator

However, there is a selection operator **->** (minus sign and greater than symbol) to eliminate the problem of pointer to structures

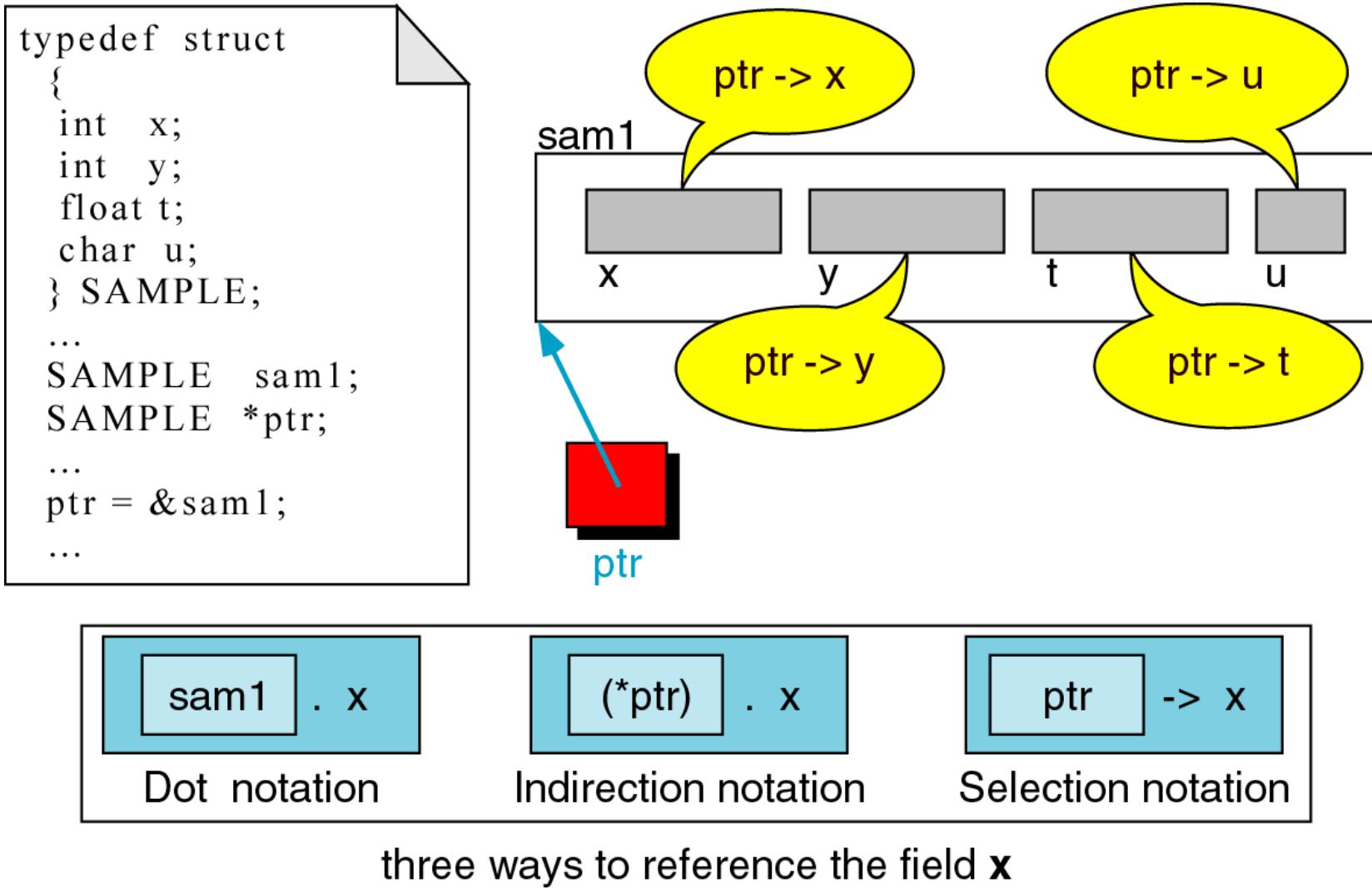
The priority of selection operator (**->**) and member operator(.) are the same

The expressions :

(*pointerName).fieldName is same as pointerName->fieldName

But **pointerName -> fieldName** is preferred

Figure 12-16 pointer selection operator





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 *



Function returning Pointer

```
int *larger(int *, int *);
void main()
{
    int a=10, b=20, *p; p=larger (&a, &b); cout<<*p;
}
int *larger(int *x, int *y)
{
    if (*x > *y)
        return(x); else
        return(y);
}
```



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;**

textbook

Structured programming programing using C, 2 edition, Behrouz ferouzan



Dynamic Memory Allocation II

- Initializing memory using new:

ptr variable = new data type(value);

int *p = new int(25);

- The constructor will be called implicitly, to initialize the values which are being passed into the object.
- To create arrays,

ptr variable = new datatype [size];



Dynamic Memory Allocation III

- Creating the multidimensional array all the sizes must be supplied. i.e

arr ptr = new int[3][4][5]; // Legal

arr ptr = new int[m][4]; //Legal

arr ptr = new int[][][5]; // Illegal

arr ptr = new int[4][]; // Illegal

- The first dimension may be a variable whose value is supplied at runtime, whereas the others must be constants.

DYNAMIC OBJECTS

Memory Management

Static Memory Allocation

- Memory is allocated at compilation time

Dynamic Memory

- Memory is allocated at running time

Static vs. Dynamic Objects

► Static object

(variables as declared in function calls)

► Memory is acquired automatically

► Memory is returned automatically when object goes out of scope

► Dynamic object

► Memory is acquired by program with an allocation request

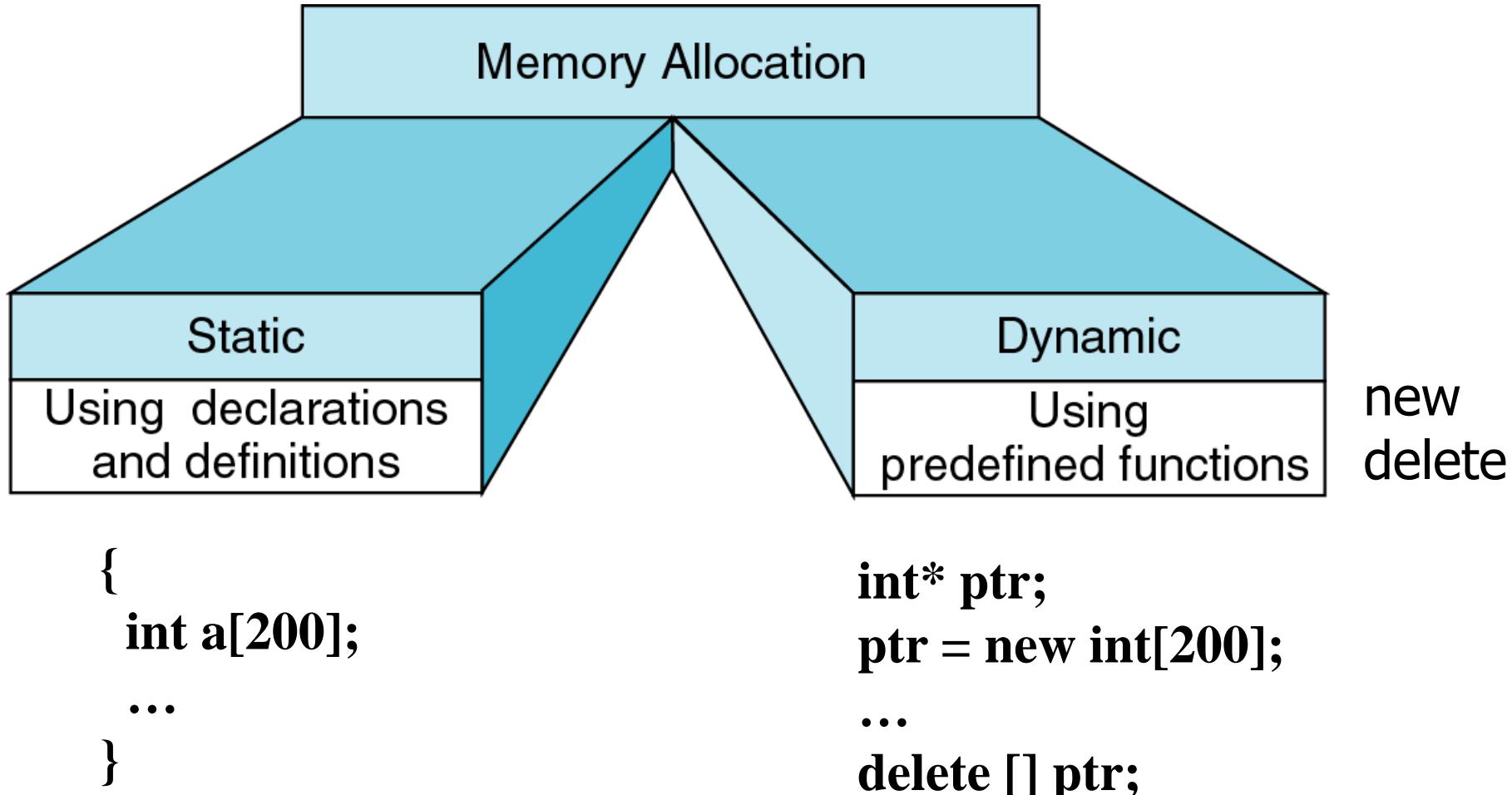
► **new operation**

► Dynamic objects can exist beyond the function in which they were allocated

► Object memory is returned by a deallocation request

► **delete operation**

Memory Allocation



Object (variable) creation: New

Syntax

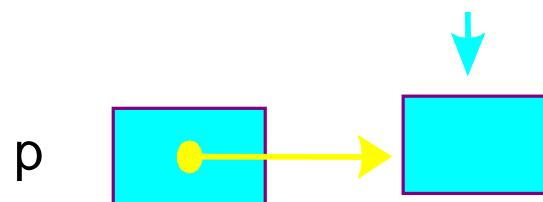
ptr = new SomeType;

where ptr is a pointer of type SomeType

Example:

int* p = new int;

Uninitialized int variable



Object (variable) destruction: Delete

Syntax

delete p;

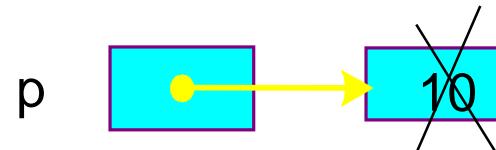
storage pointed to by p is returned to free store and p is now undefined

Example:

int* p = new int;

***p = 10;**

delete p;



Array of New: Dynamic arrays

Syntax:

P = new SomeType[Expression];

- Where
 - P is a pointer of type SomeType
 - Expression is the number of objects to be constructed -- we are making an array

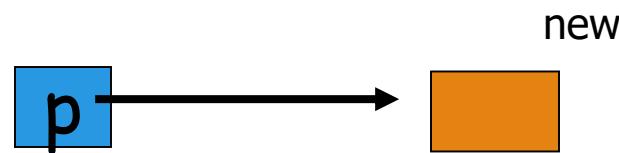
Because of the flexible pointer syntax, P can be considered to be an array

Example

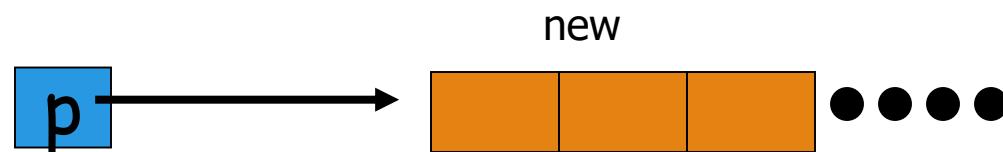
Dynamic Memory Allocation

- Request for “unnamed” memory from the Operating System

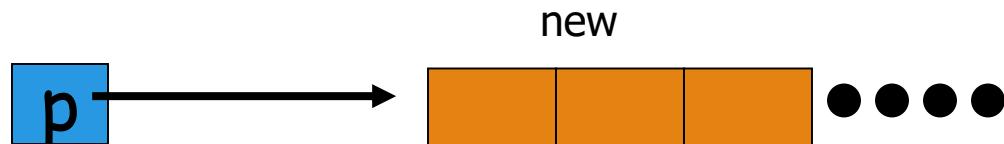
□ `int *p, n=10;`
`p = new int;`



`p = new int[100];`



`p = new int[n];`

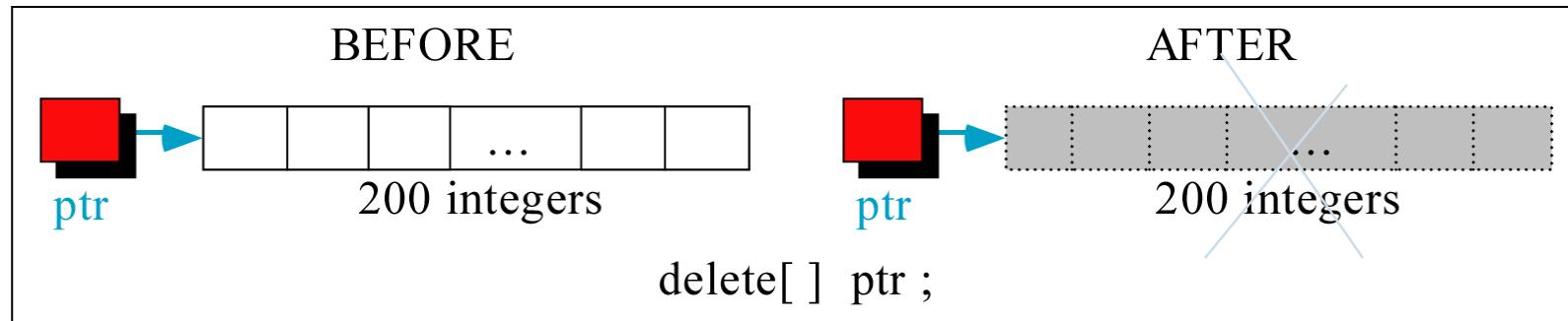
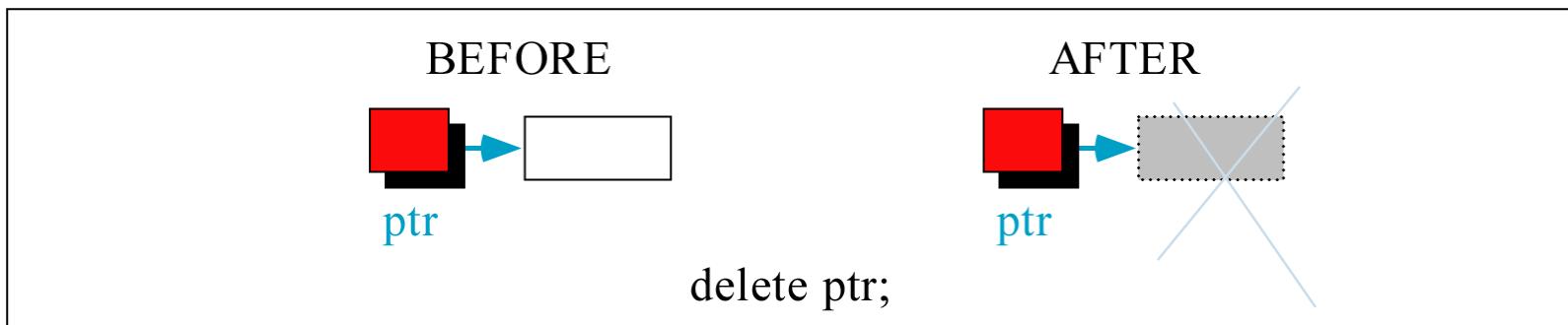


Memory Allocation Example

Want an array of unknown size

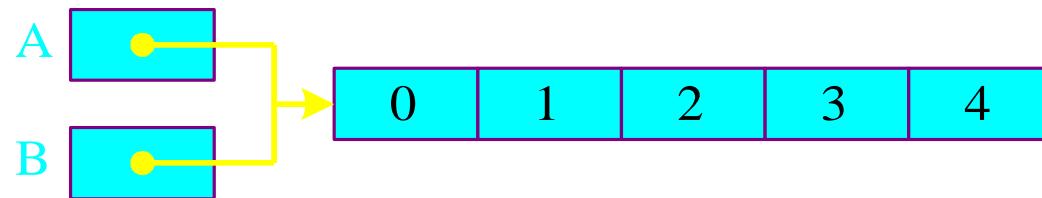
```
#include <iostream>
using namespace std;
void main()
{
    int n;
    cout << "How many students? ";
    cin >> n;
    int *grades = new int[n];
    for(int i=0; i < n; i++){
        int mark;
        cout << "Input Grade for Student" << (i+1) << " ? :";
        cin >> mark;
        grades[i] = mark;
    }
    . . .
    printMean( grades, n ); // call a function with dynamic array
    . . .
}
```

Freeing (or deleting) Memory



Dangling Pointer Problem

```
int *A = new int[5];
for(int i=0; i<5; i++)
    A[i] = i;
int *B = A;
```



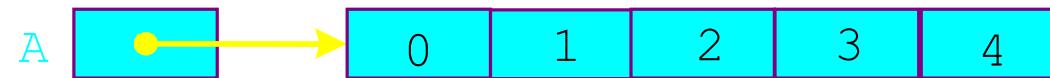
Locations do not belong to program

```
delete [] A;
B[0] = 1; // illegal!
```



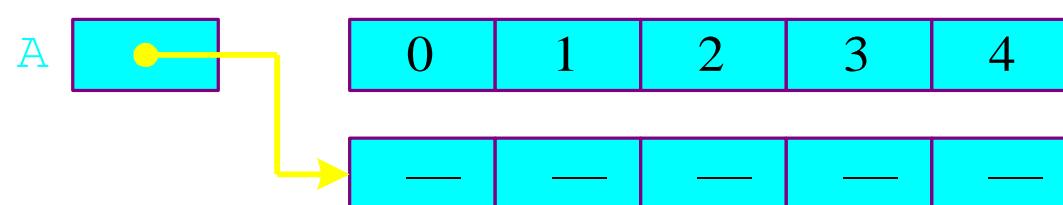
Memory Leak Problem

```
int *A = new int [5];  
for(int i=0; i<5; i++)  
    A[i] = i;
```



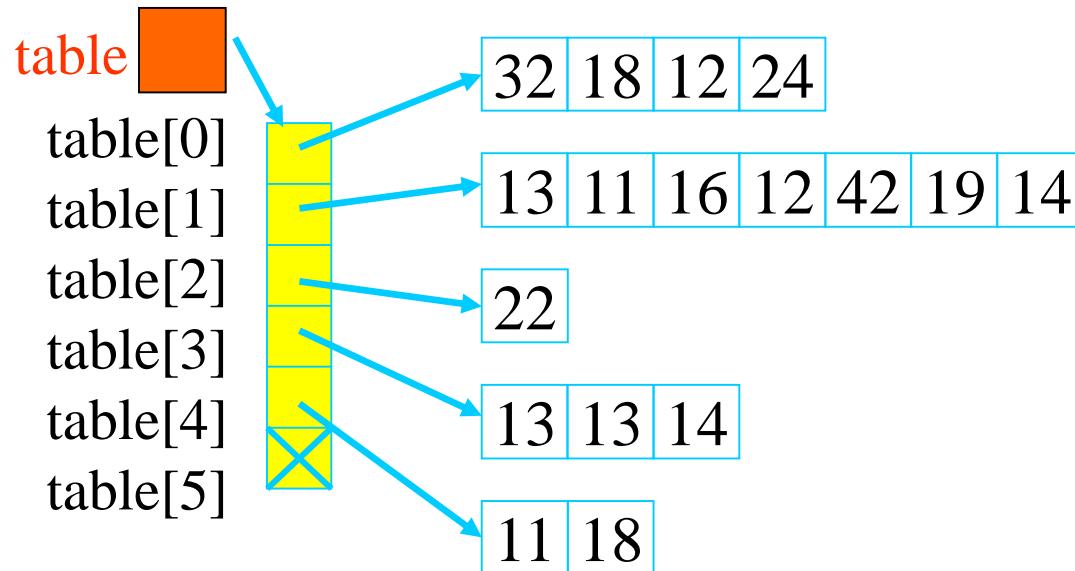
```
A = new int [5];
```

These locations cannot be
accessed by program



A Dynamic 2D Array

- A dynamic array is an array of pointers to save space when not all rows of the array are full.
- **int **table;**



```
table = new int*[6];  
...  
table[0] = new int[4];  
table[1] = new int[7];  
table[2] = new int[1];  
table[3] = new int[3];  
table[4] = new int[2];  
table[5] = NULL;
```

Memory Allocation

```
int **table;
```

```
table = new int*[6];
```

```
table[0]= new int[3];
```

```
table[1]= new int[1];
```

```
table[2]= new int[5];
```

```
table[3]= new int[10];
```

```
table[4]= new int[2];
```

```
table[5]= new int[6];
```

```
table[0][0] = 1; table[0][1] = 2;  
table[0][2] = 3;
```

```
table[1][0] = 4;
```

```
table[2][0] = 5; table[2][1] = 6;  
table[2][2] = 7; table[2][3] = 8;  
table[2][4] = 9;
```

```
table[4][0] = 10; table[4][1] = 11;
```

```
cout << table[2][5] << endl;
```

Memory Deallocation

Memory leak is a serious bug!

Each row must be deleted individually

Be careful to delete each row before deleting the table pointer.

- **for(int i=0; i<6; i++)
 delete [] table[i];
delete [] table;**

Create a matrix of any dimensions, m by n:

```
int m, n;  
cin >> m >> n >> endl;  
  
int** mat;  
  
mat = new int*[m];  
  
for (int i=0;i<m;i++)  
    mat[i] = new int[n];
```

Put it into a function:

```
int m, n;  
cin >> m >> n >> endl;  
int** mat;  
mat = imatrix(m,n);  
...  
int** imatrix(nr, nc) {  
    int** m;  
    m = new int*[nr];  
    for (int i=0;i<nr;i++)  
        m[i] = new int[nc];  
    return m; }
```

Pointers to objects

Pointers to objects

Any type that can be used to declare a variable/object can also have a pointer type.

Consider the following class:

```
class Rational
{
    private:
        int numerator;
        int denominator;
    public:
        Rational(int n, int d);
        void Display();
};
```

Pointers to objects (Cont..)

→ Rational *rp = NULL;
Rational r(3,4);
rp = &r;

<i>rp</i>	
FFF0	0
FFF1	
FFF2	
FFF3	
FFF4	
FFF5	
FFF6	
FFF7	
FFF8	
FFF9	
FFFA	
FFFB	
FFFC	
FFFD	

Pointers to objects (Cont..)

Rational *rp = NULL;
→ Rational r(3,4);
rp = &r;

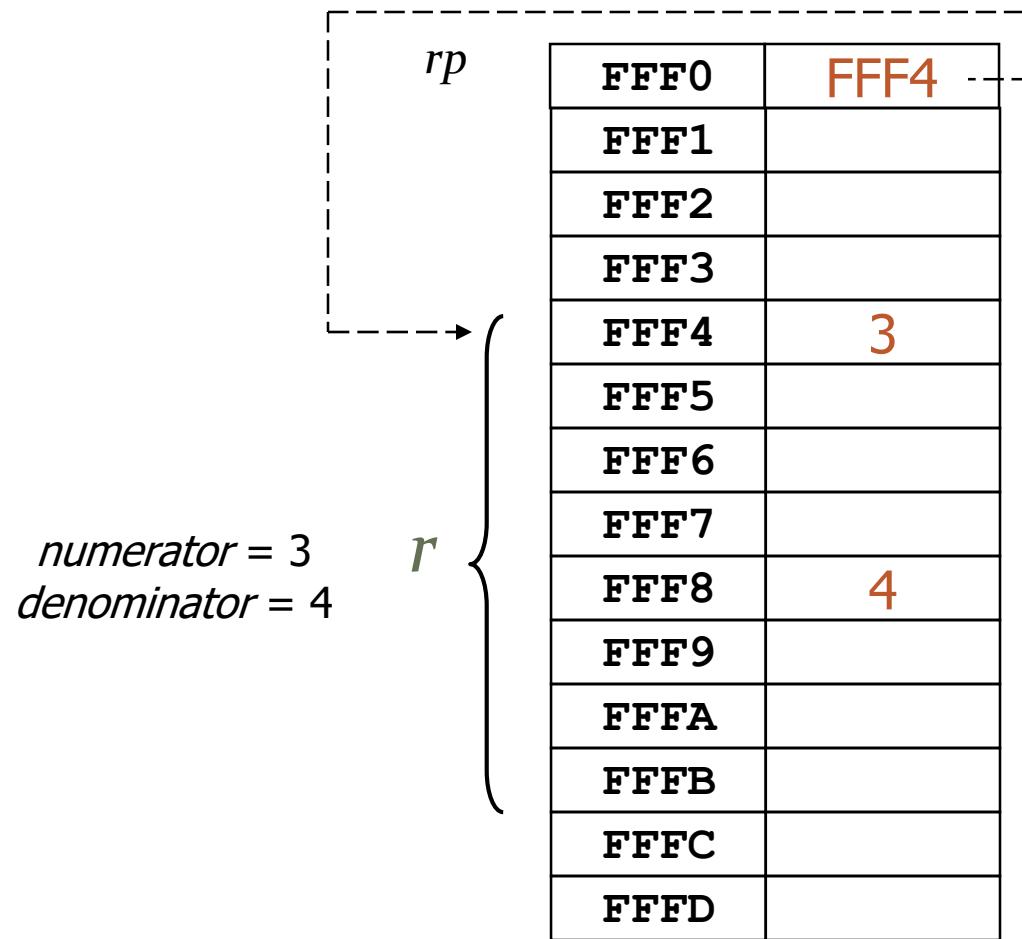
*numerator = 3
denominator = 4*

<i>rp</i>	
FFF0	0
FFF1	
FFF2	
FFF3	
FFF4	3
FFF5	
FFF6	
FFF7	
FFF8	4
FFF9	
FFFA	
FFFB	
FFFC	
FFFD	

r {

Pointers to objects (Cont..)

```
Rational *rp = NULL;  
Rational r(3,4);  
rp = &r;
```



Pointers to objects (Cont..)

If **rp** is a pointer to an object, then two notations can be used to reference the instance/object **rp** points to.

Using the *de-referencing operator* *

(*rp).Display();

Using the *member access operator ->*

rp -> Display();

Dynamic Allocation of a Class Object

Consider the Rational class defined before:

```
Rational *rp;  
int a, b;  
cin >> a >> b;  
rp = new Rational(a,b);  
(*rp).Display(); // rp->Display();  
delete rp;  
rp = NULL;
```

End of Pointers



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

Linear list

- **Linear list** is a list in which each element has a unique successor.
- 2 categories:-
 - restricted and general list
- In **restricted list**, addition and deletion of data are restricted to the ends of the list.(stack, queue)
- In **general lists**, data can be inserted and deleted anywhere in the list(beginning, middle, end).

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

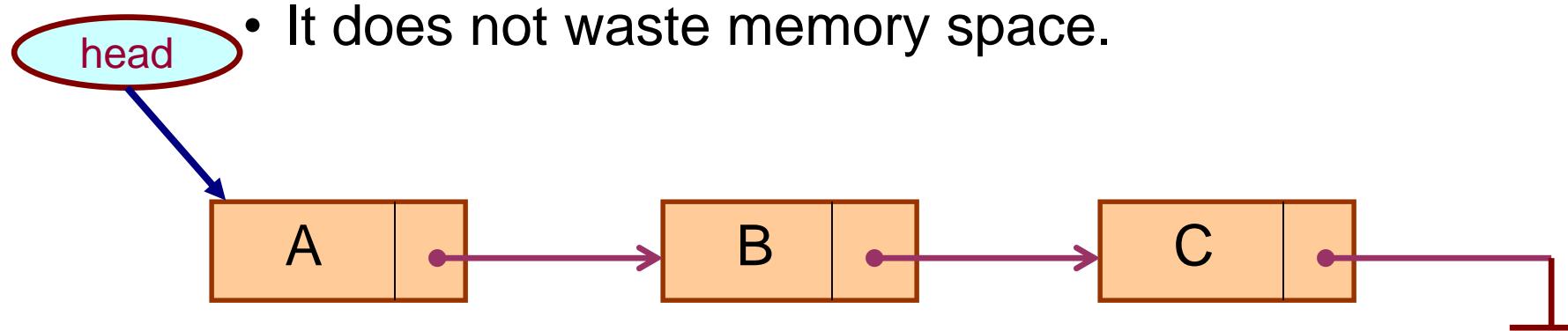
- What is an abstract data type?
 - It is a data type defined by the user.
 - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
 - Because details of the implementation are **hidden**.
 - When you do some operation on the list, say insert an element, you just call 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**.

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.



Introduction- contd...

- 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.

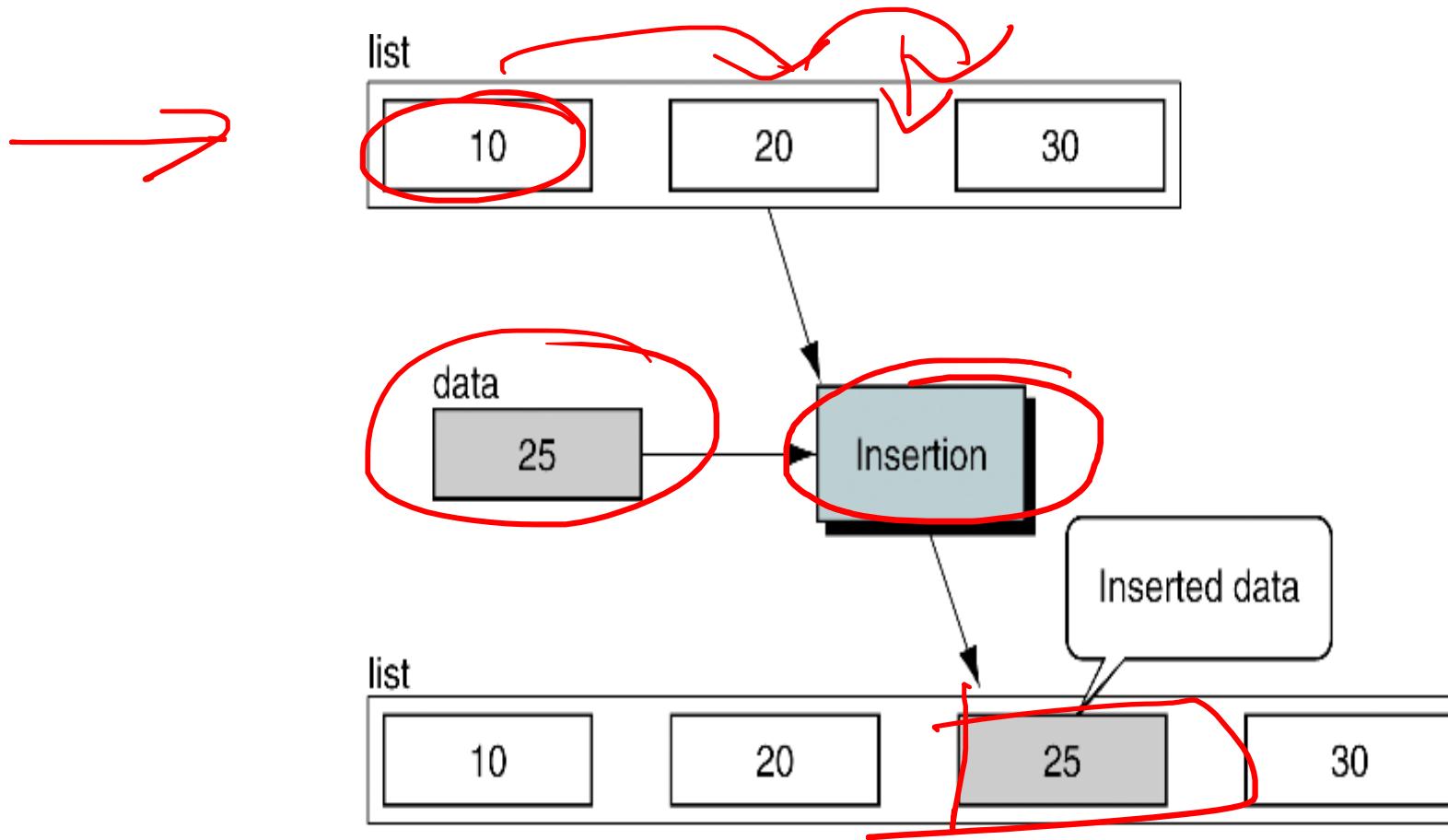


FIGURE 5-1 Insertion

Insertion is used to add a new element to the list

Deletion.

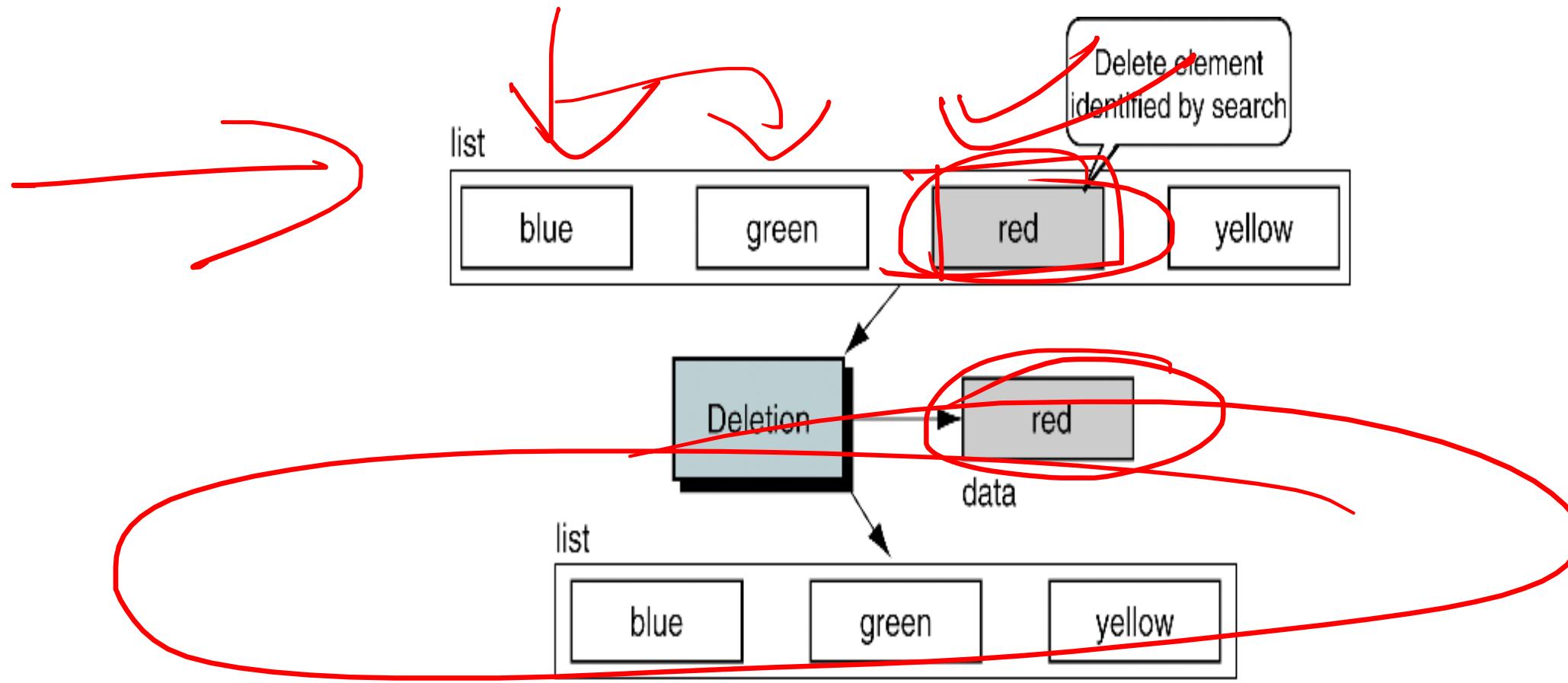


FIGURE 5-2 Deletion

Deletion is used to remove an element from the list.

Retrieval.

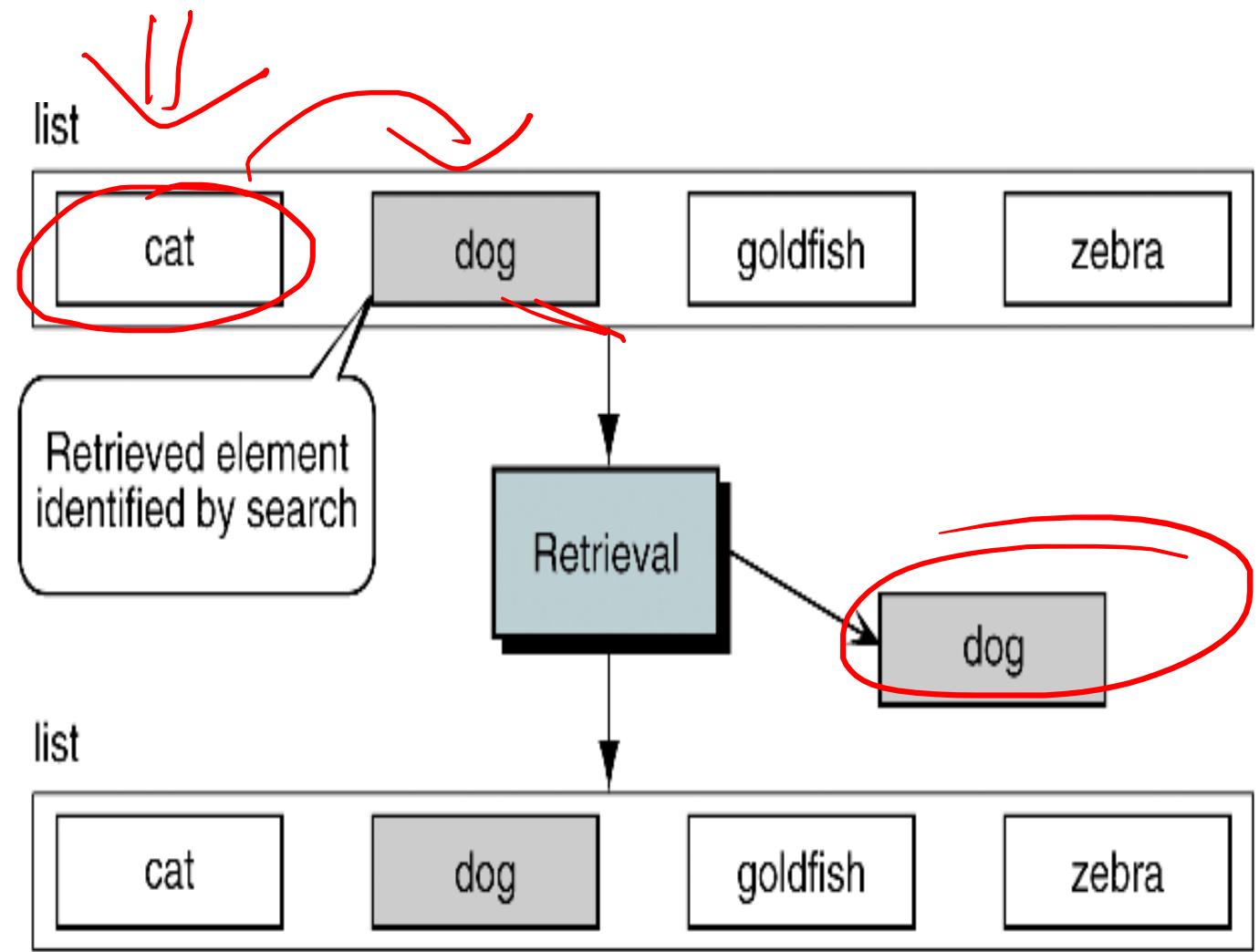
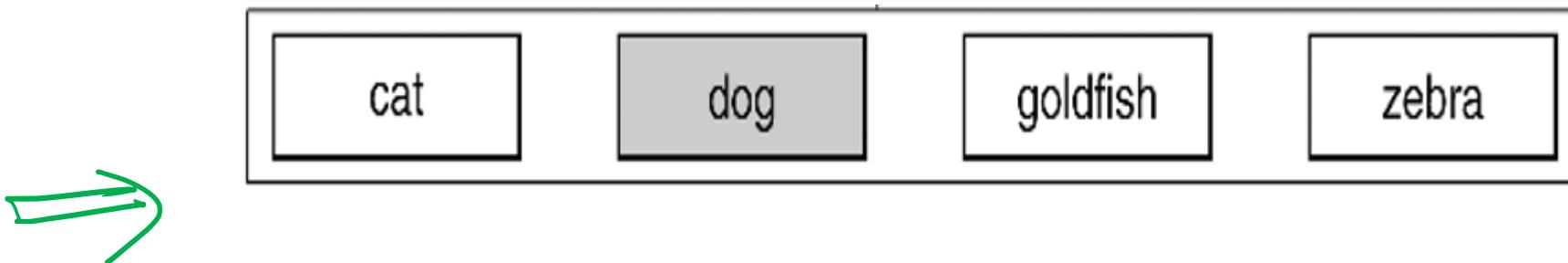


FIGURE 5-3 Retrieval

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.

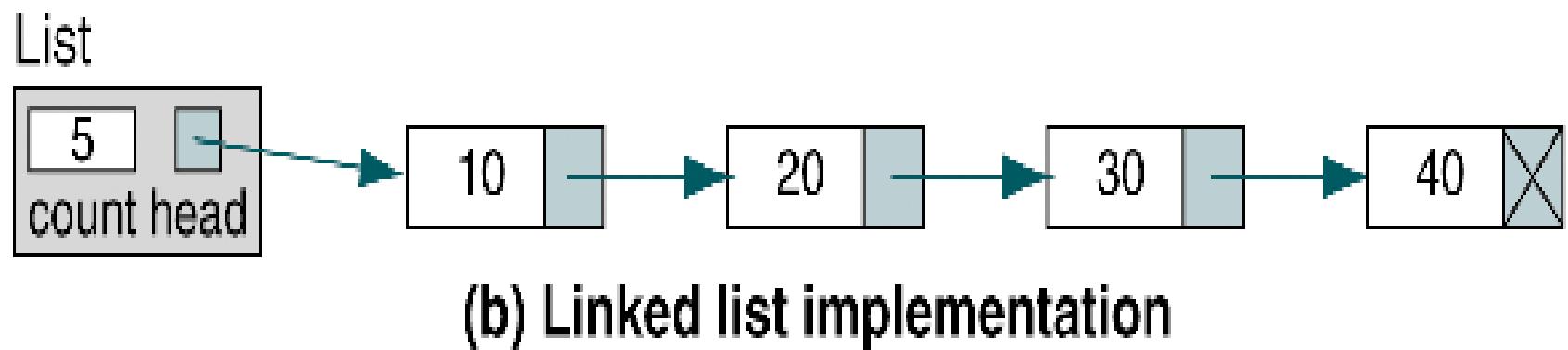
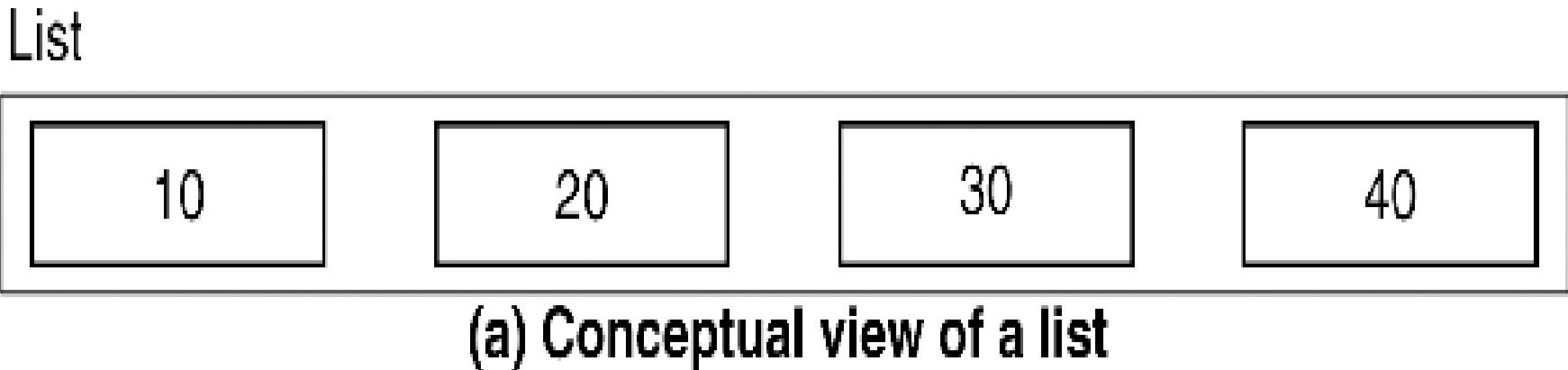
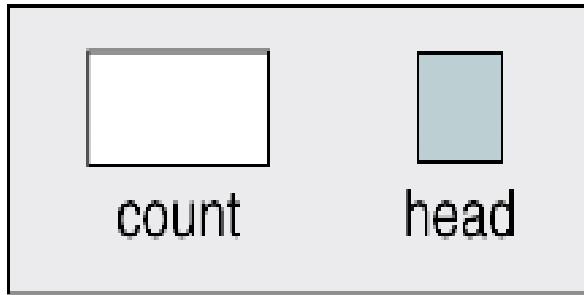
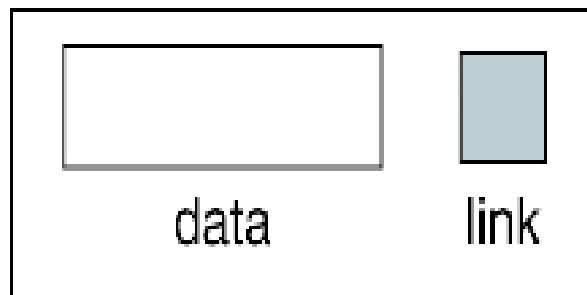


FIGURE 5-4 Linked List Implementation of a List

Data structure.



(a) Head structure



(b) Data node structure

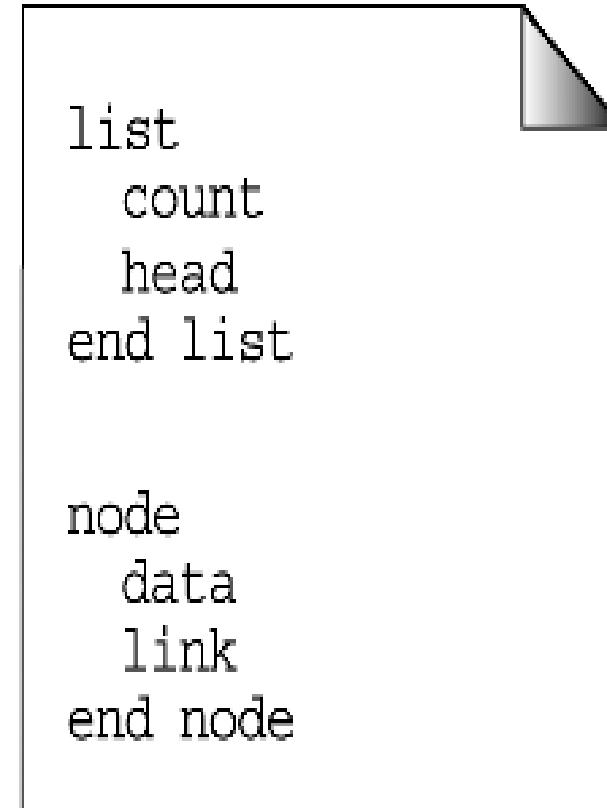
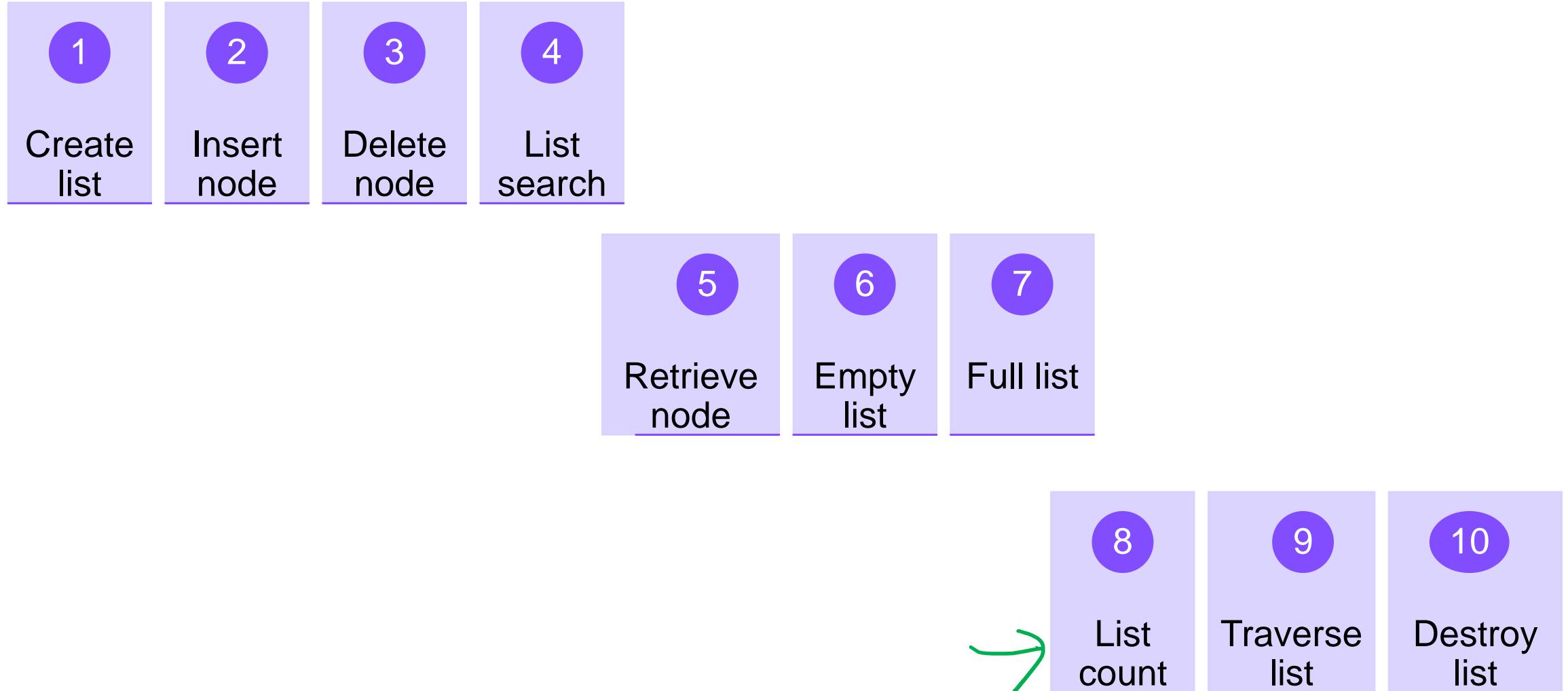


FIGURE 5-5 Head Node and Data Node

Algorithms.



Create list.

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.

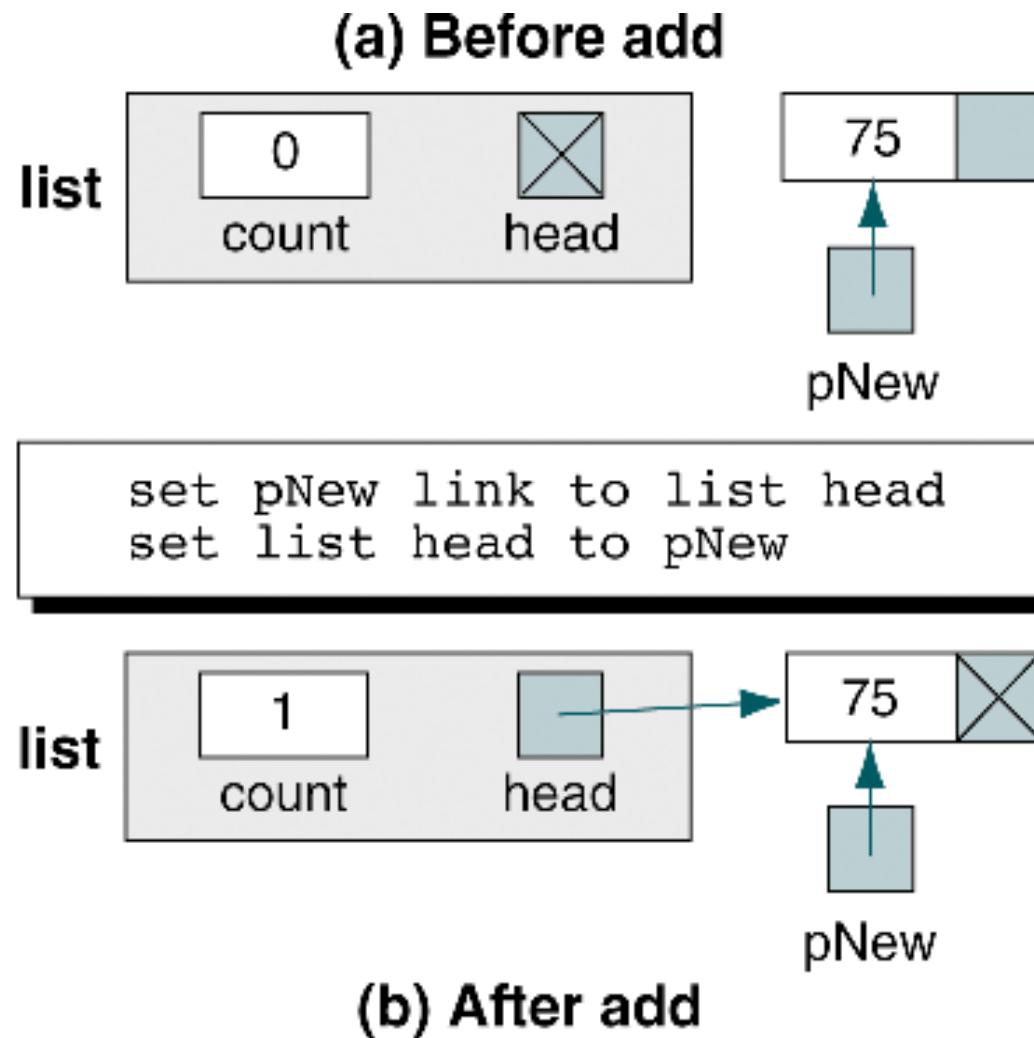


FIGURE 5-7 Add Node to Empty List

- 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 = data;
```

```
newNode->next = head; //null copied to next part of newNode from head  
head = newNode //address of newNode copied to head
```

Insert at beginning.

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

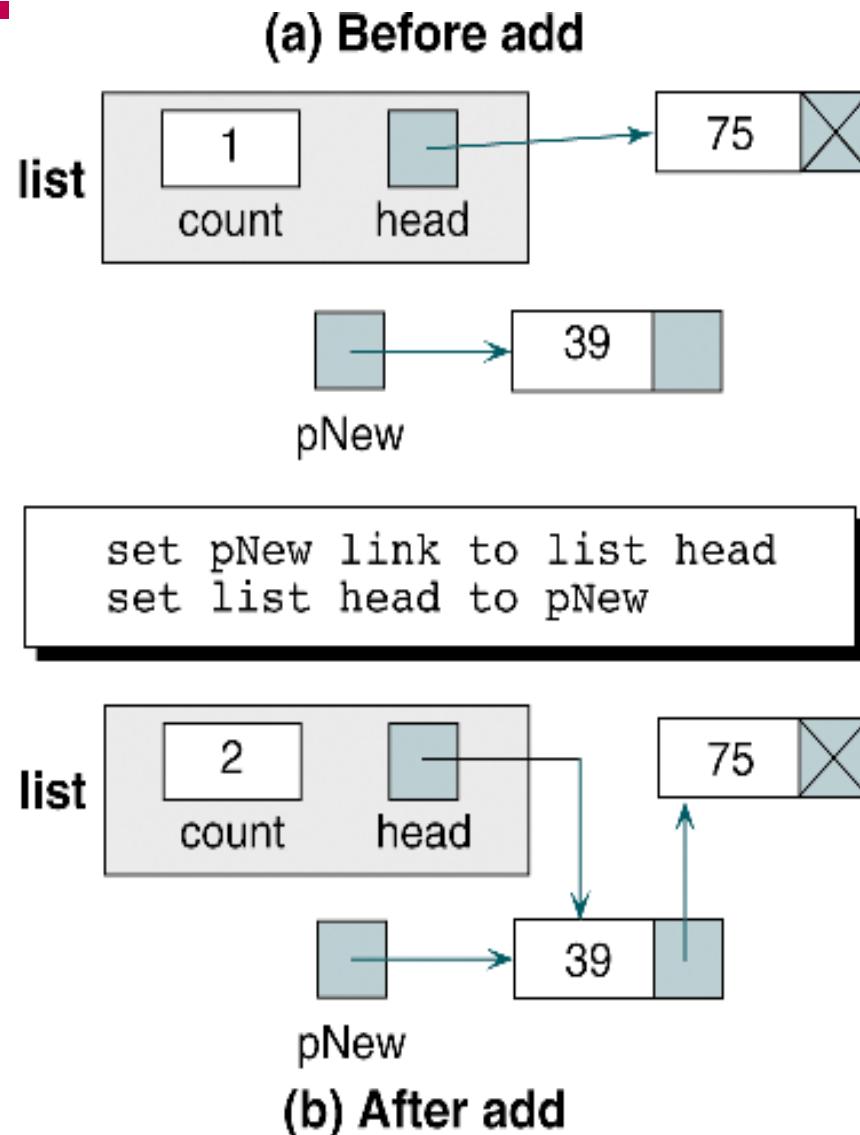


FIGURE 5-8 Add Node at Beginning

Insert in middle.

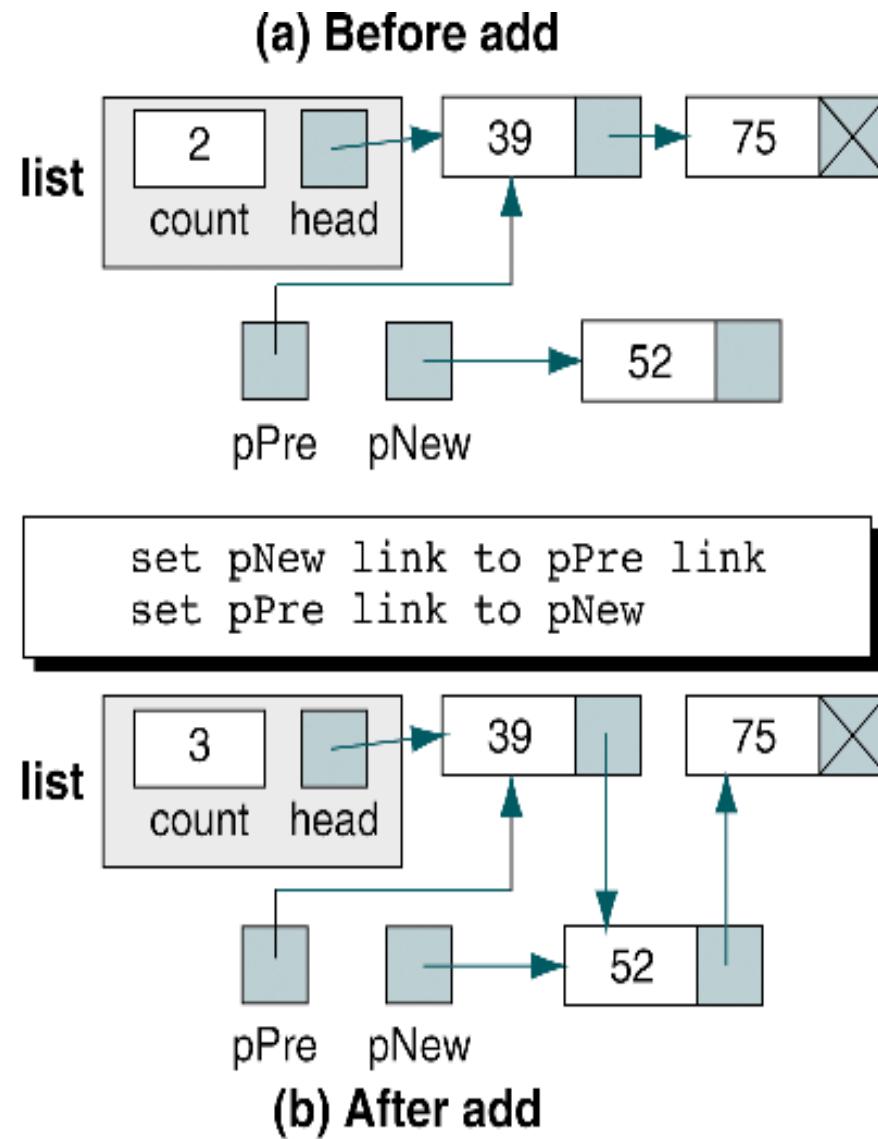


FIGURE 5-9 Add Node in Middle

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*

→ **newNode->data = data;**

struct node *temp = head;

for(int i=1; i < position; i++)

{

if(temp->next != NULL)

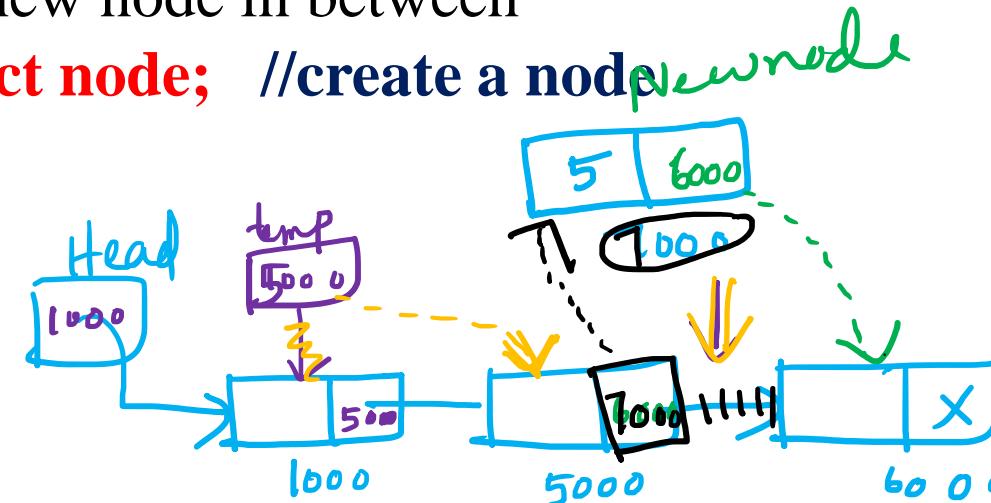
{

→ **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

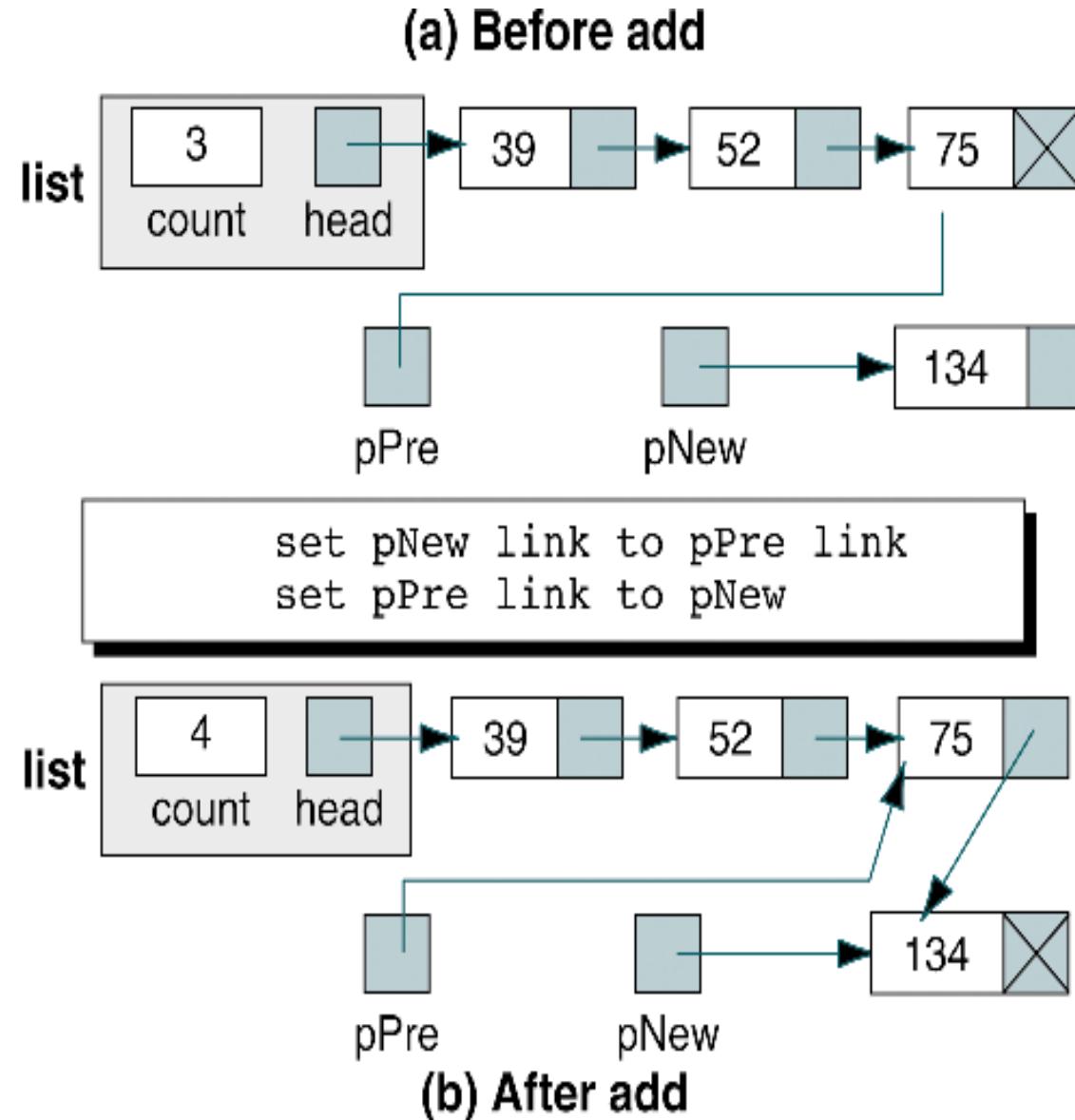
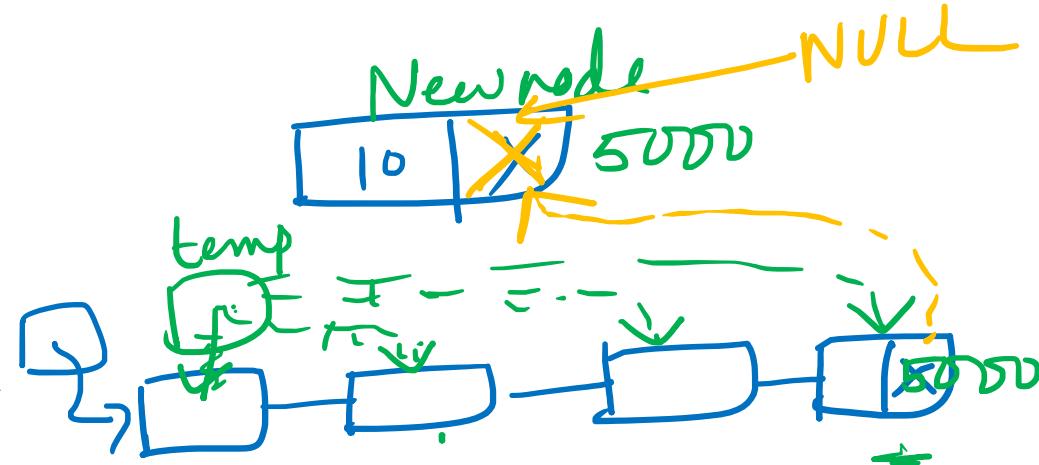


FIGURE 5-10 Add Node at End

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

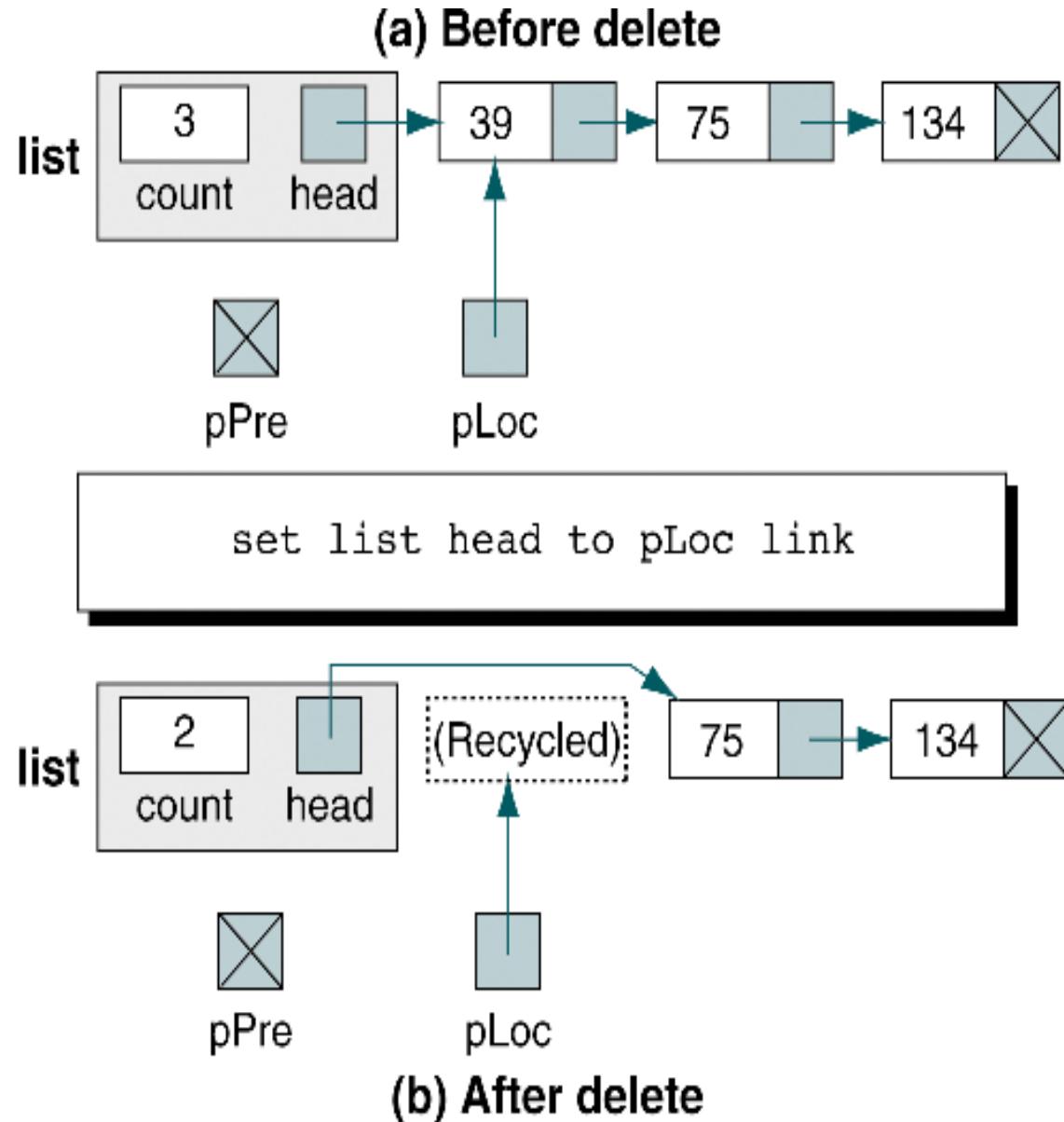


FIGURE 5-11 Delete First Node

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.

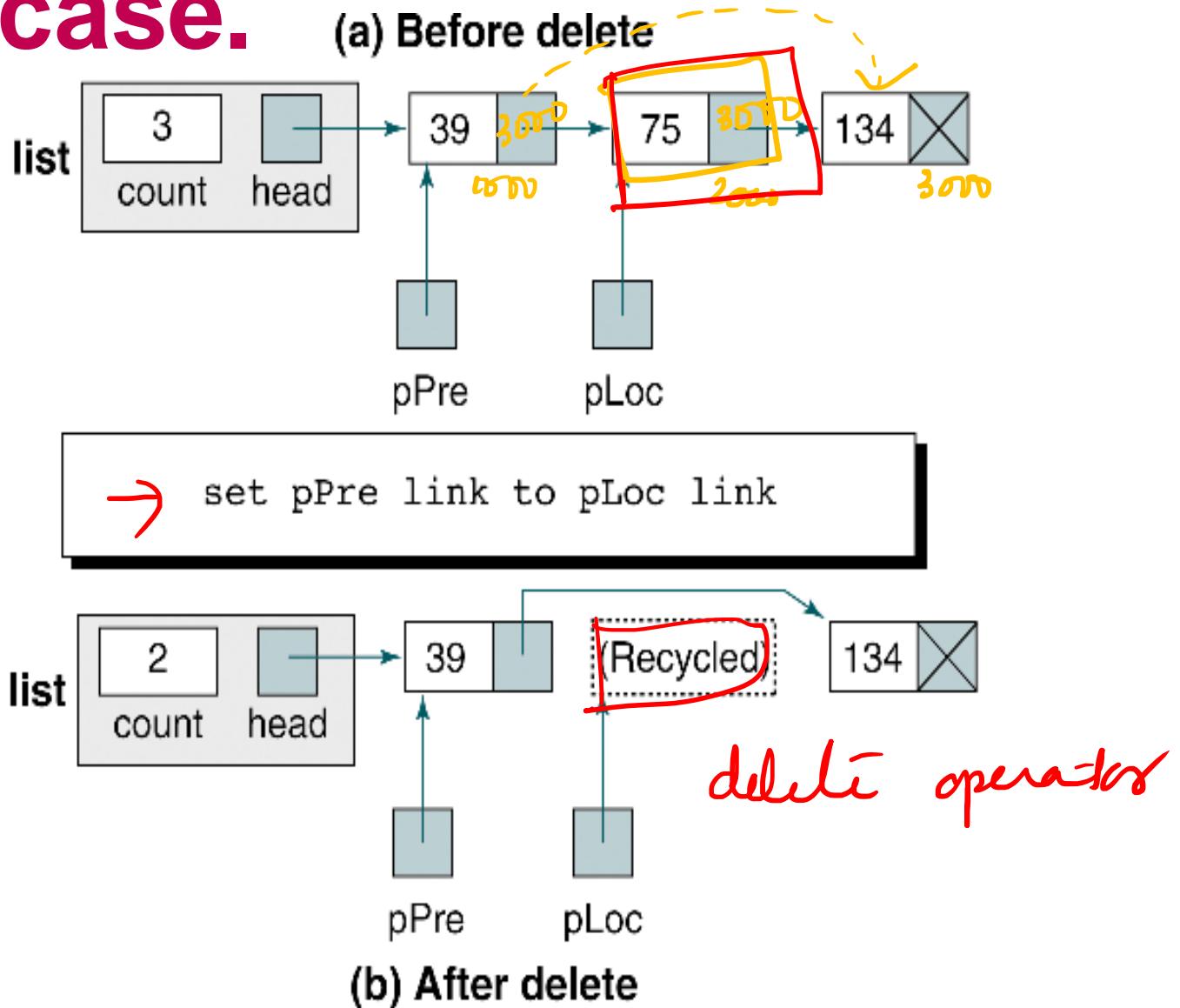
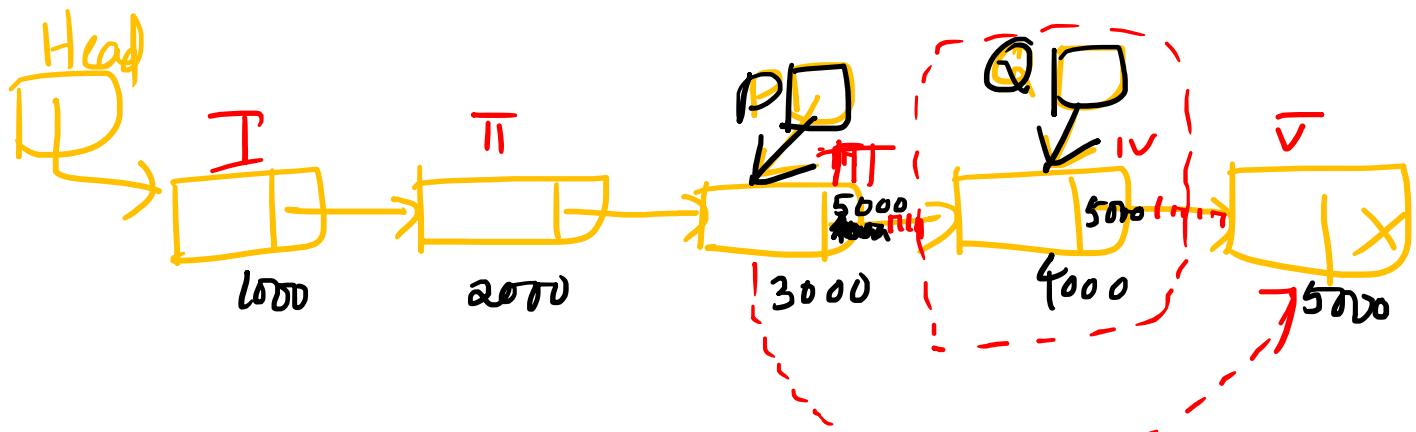
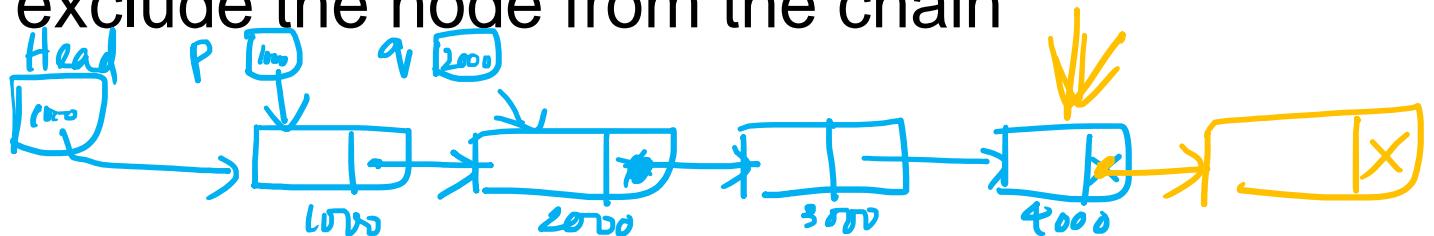


FIGURE 5-12 Delete General Case

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=0; 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.

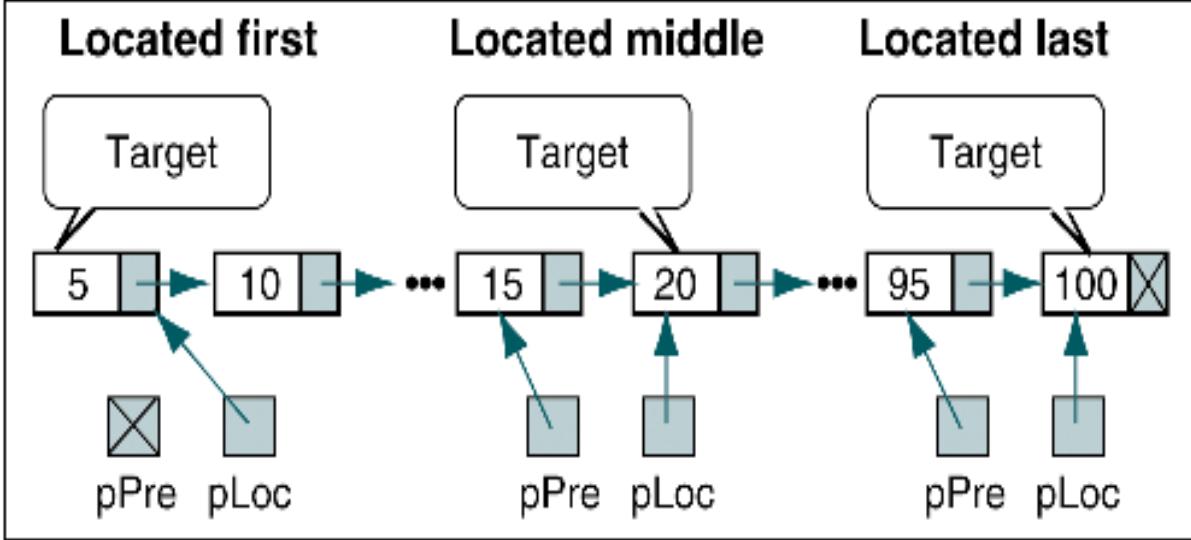
    Pre  list is metadata structure to a valid list
        Pre is a pointer to predecessor node
        pLoc is a pointer to node to be deleted
        dataOut is variable to receive deleted data
    Post data have been deleted and returned to caller

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
```

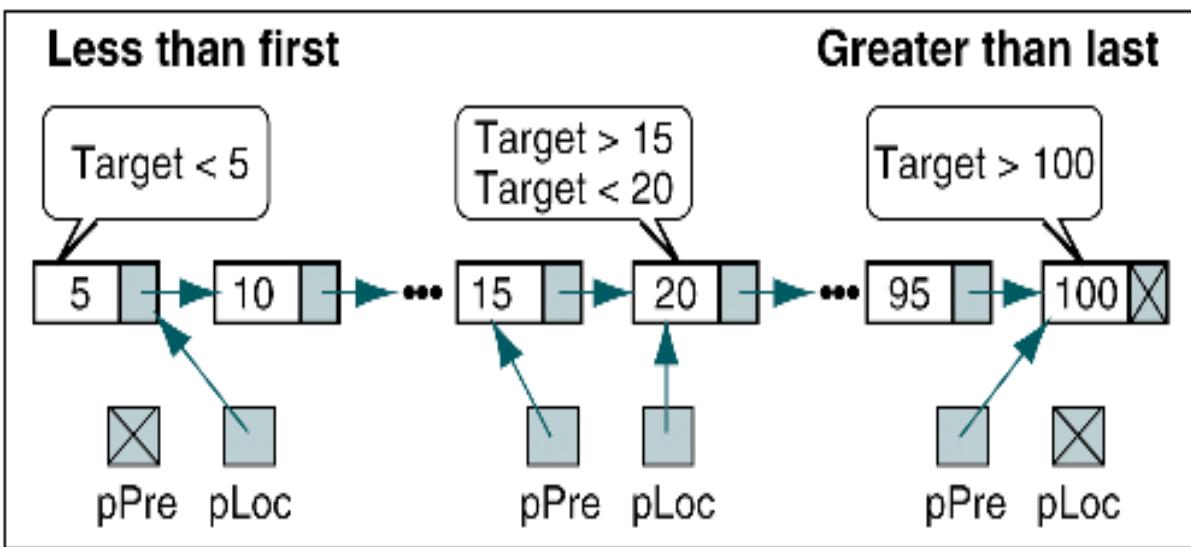
List search.

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



(a) Successful searches (return true)



(b) Unsuccessful searches (return false)

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)
    1 move pLoc data to dataOut
3 end if
4 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.

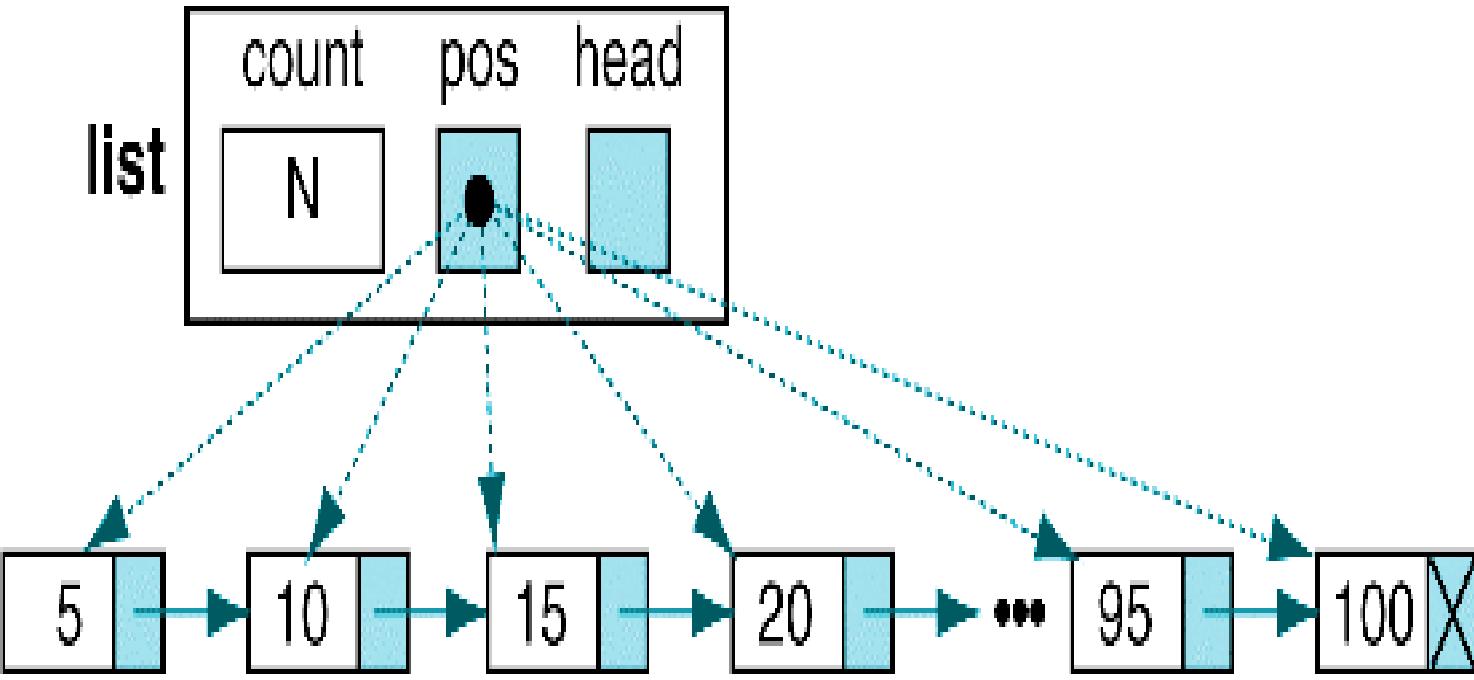


FIGURE 5-14 List Traversal

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

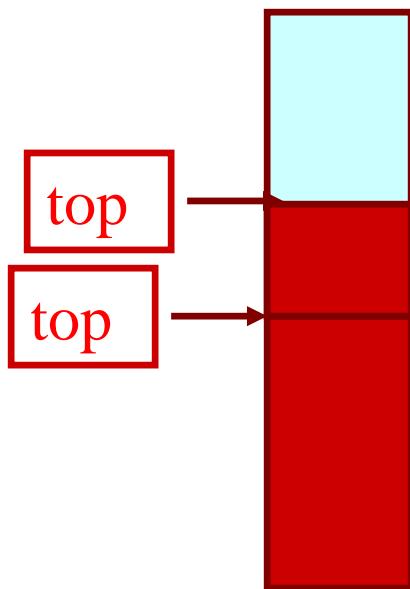
end destroyList



STACK IMPLEMENTATIONS: USING ARRAY AND LINKED LIST

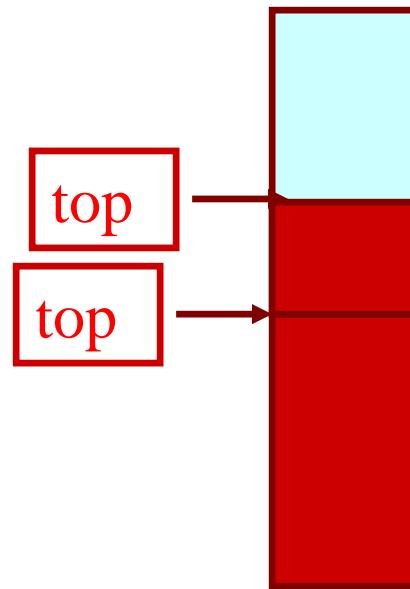
STACK USING ARRAY

PUSH



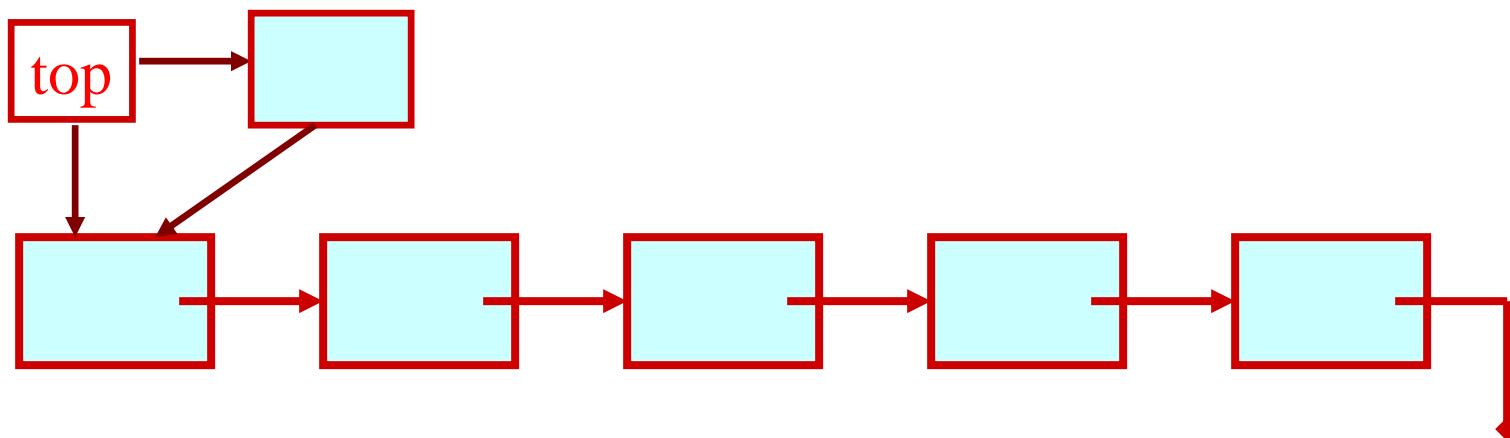
STACK USING ARRAY

POP



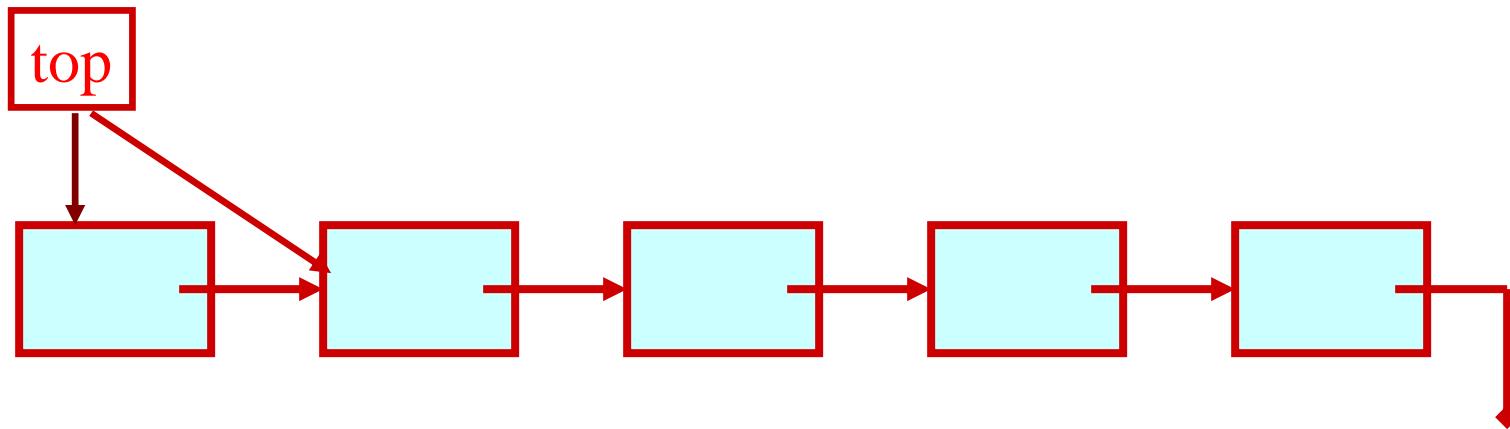
Stack: Linked List Structure

PUSH OPERATION



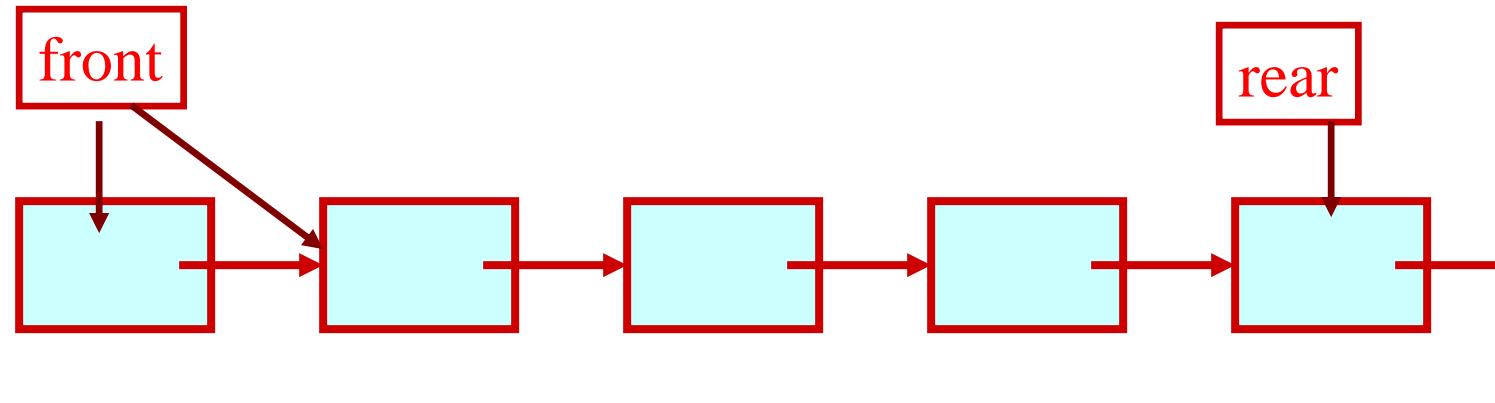
Stack: Linked List Structure

POP OPERATION



QUEUE: LINKED LIST STRUCTURE

DEQUEUE



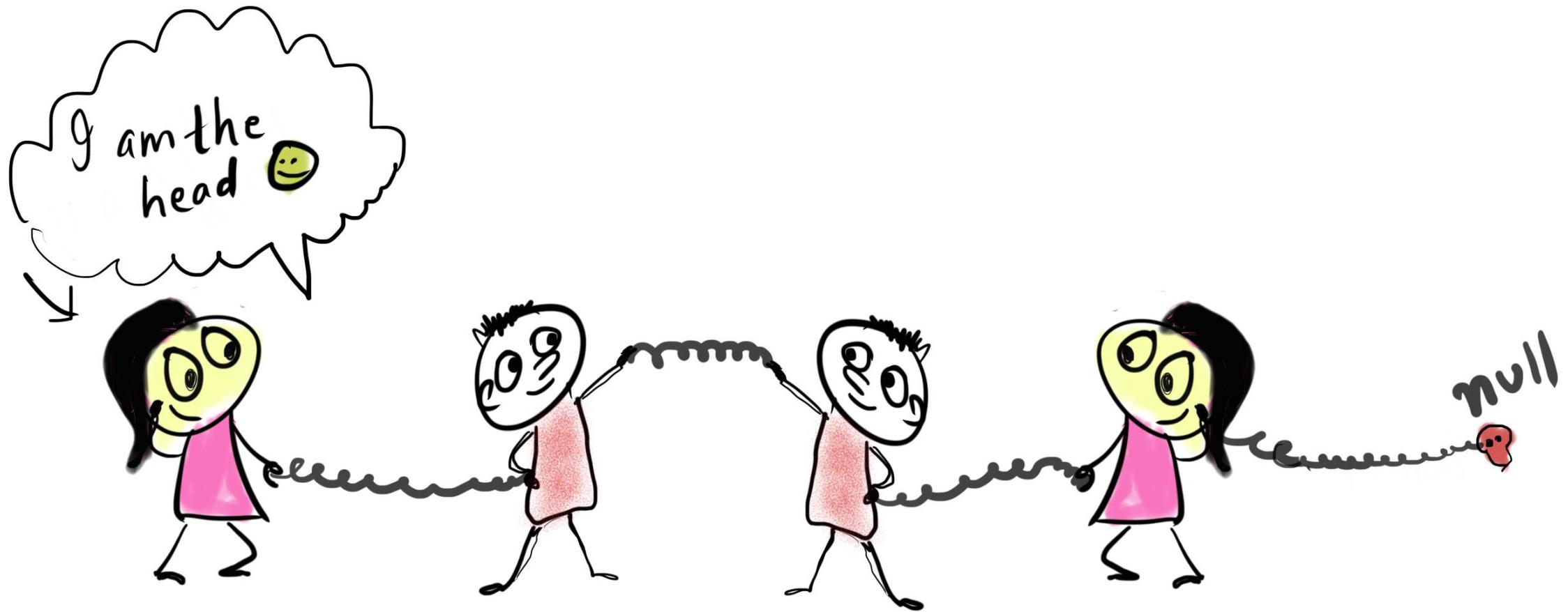


THE END.



DOUBLY AND CIRCULAR LINKED LISTS

SINGLY LINKED LIST.



RECAP TO GO FORWARD.

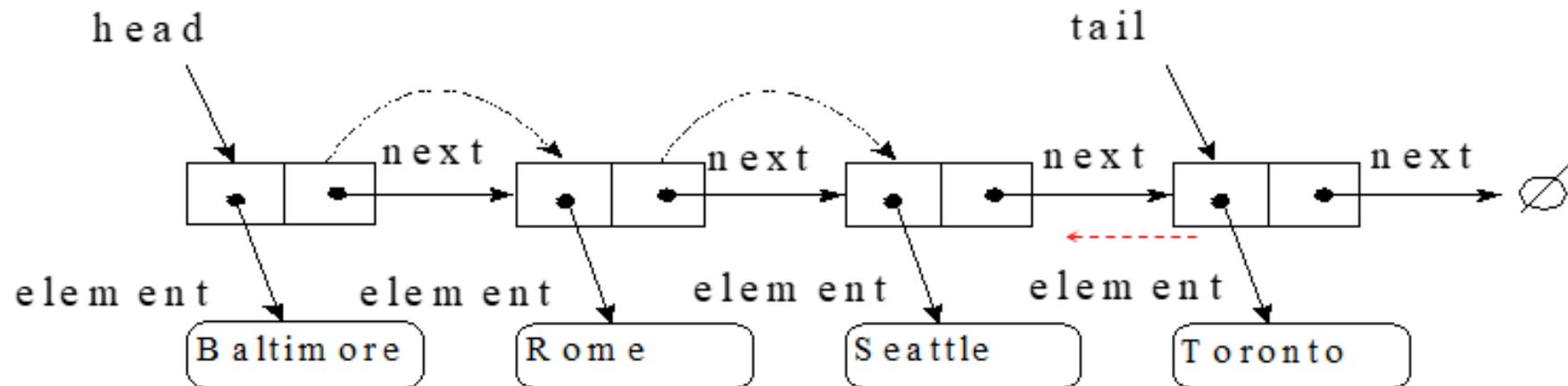
- 🔔 The node at the beginning is called the head of the list
- 🔔 The entire list is identified by the pointer to the head node, this pointer is called the list head.
- 🔔 Nodes can be added or removed from the linked list during execution
- 🔔 Addition or removal of nodes can take place at beginning, end, 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.

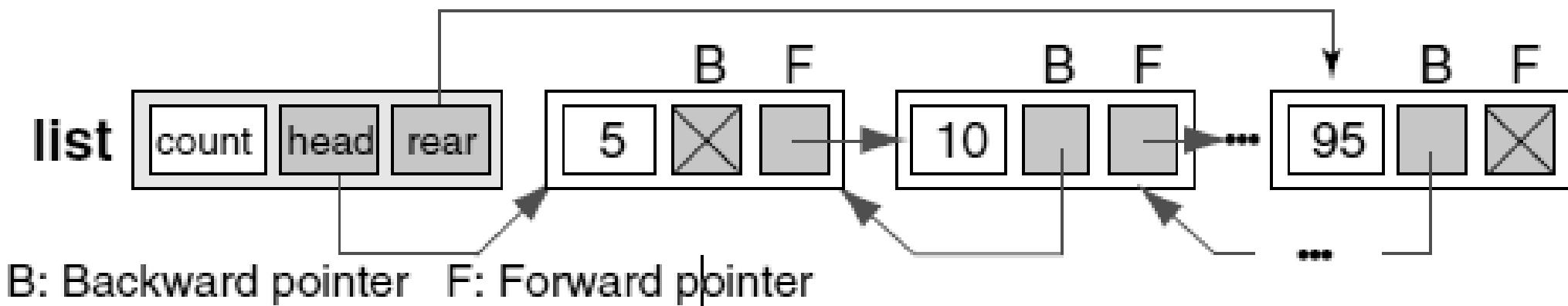


DOUBLY LINKED LIST.

- ⌚ 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

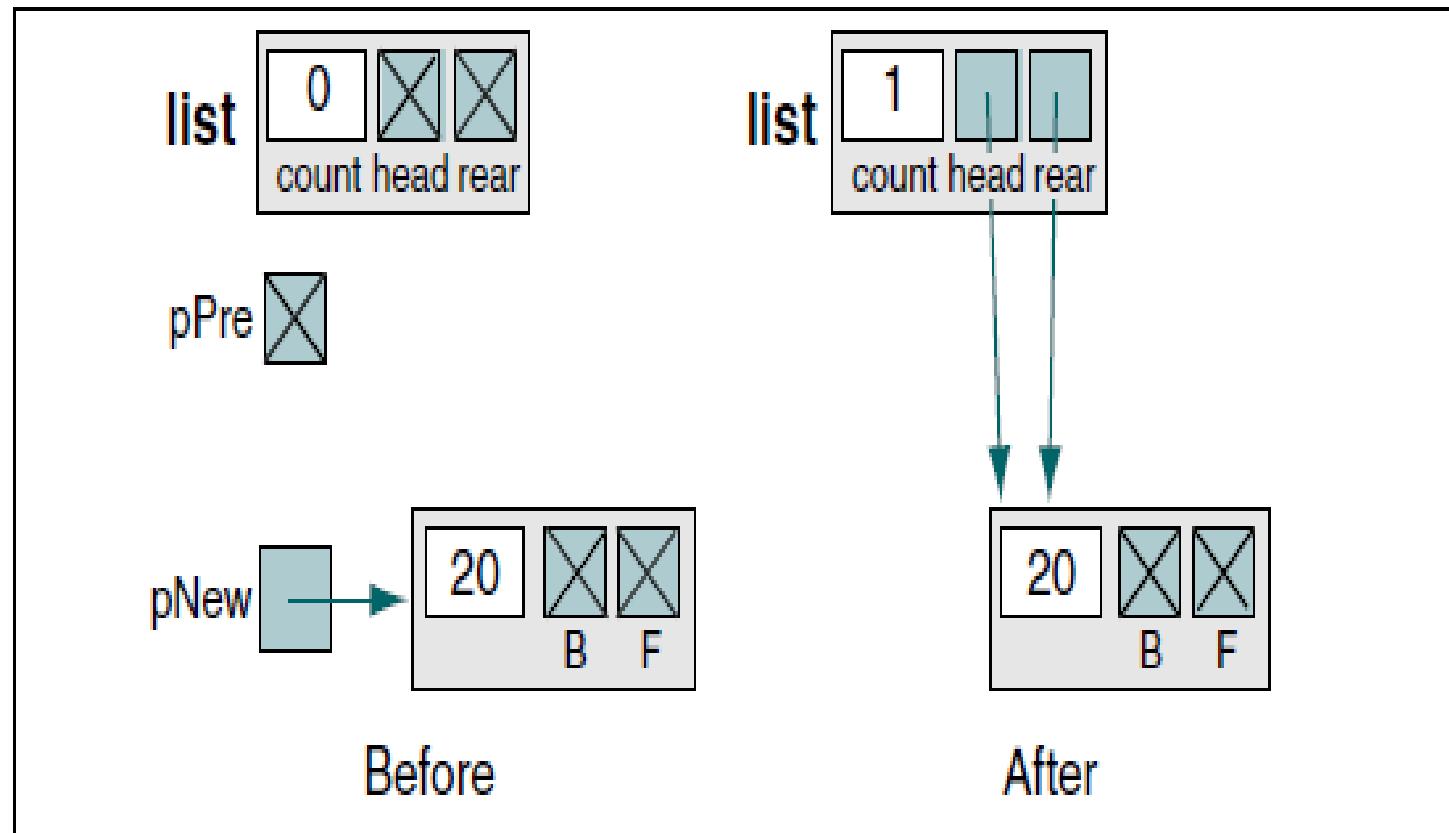


Delete a node



DOUBLY LINKED LIST - INSERTION.

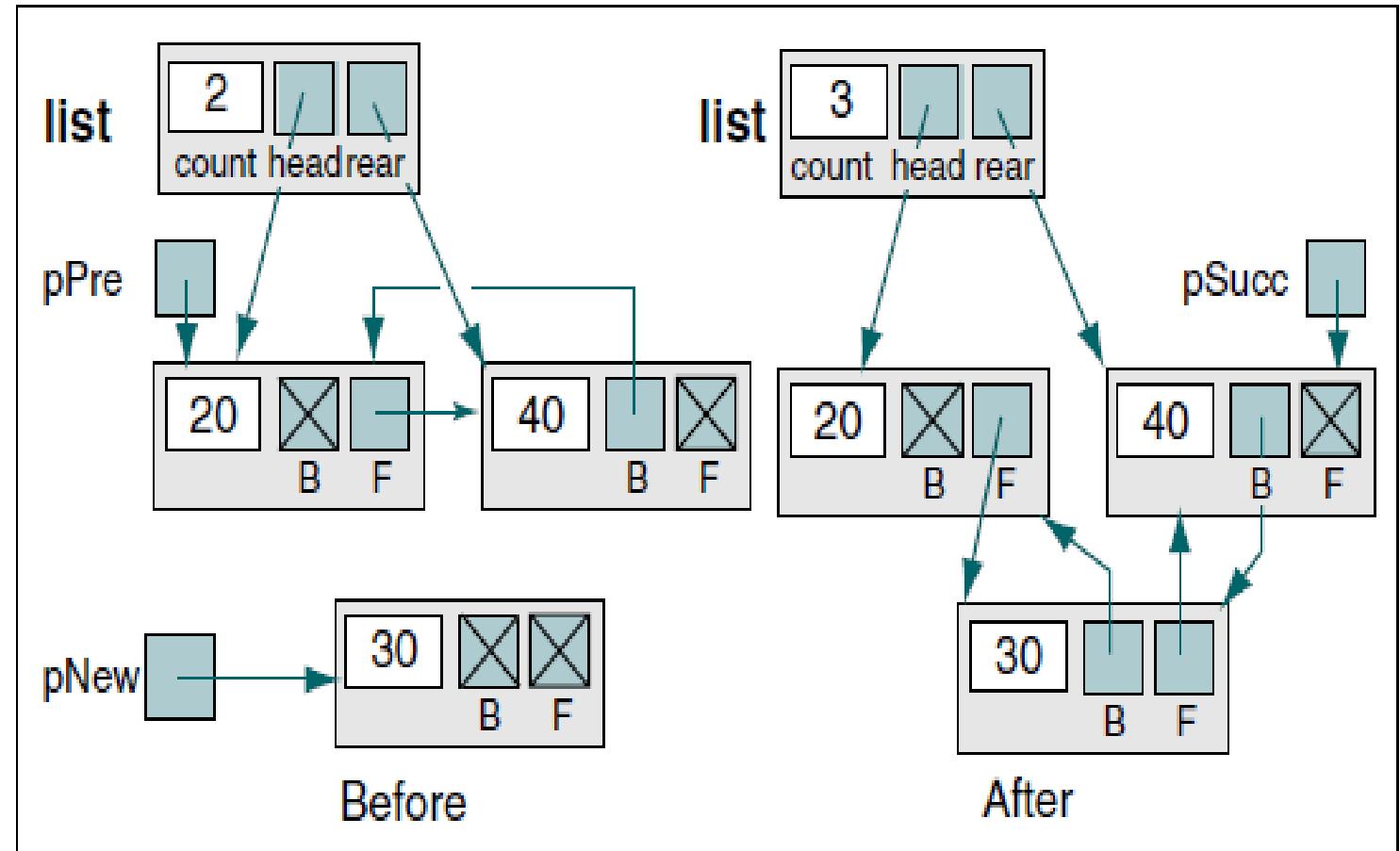
- Inserting follows the basic pattern of inserting a node into a singly linked list, but we also need to **connect both the forward and the backward pointers**.
- A null doubly linked list's **head and rear pointers are null**.
- To insert a node into a null list, we simply **set the head and rear pointers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

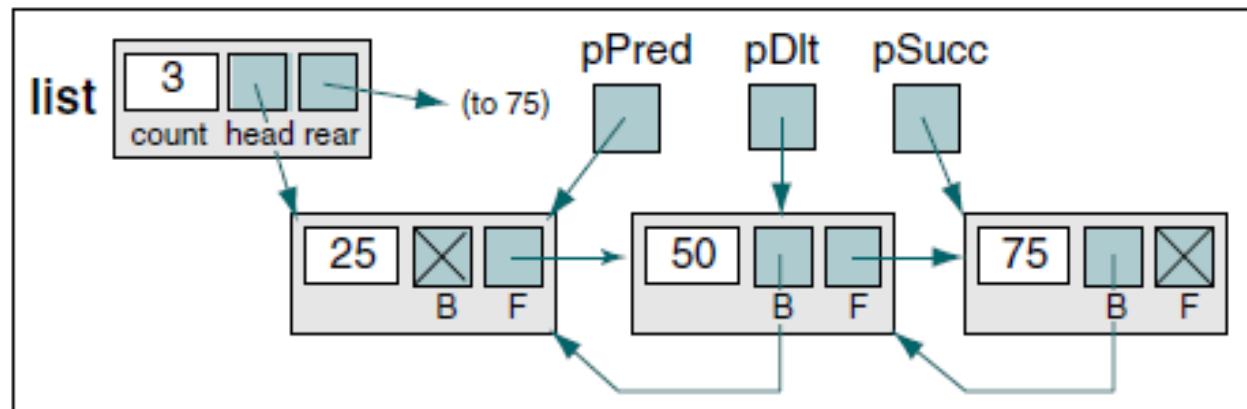
ALGORITHM 5-11 Doubly Linked List Insert

```
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
```

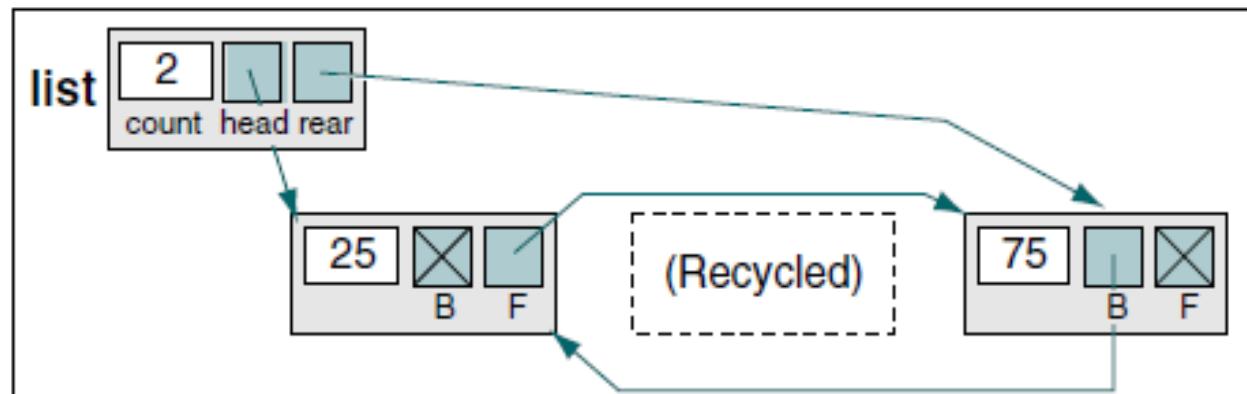
```
4 else
    Inserting into middle or end of list
    1 set new node fore pointer to predecessor fore pointer
    2 set new node back pointer to predecessor
5 end if
    Test for insert into null list or at end of list
6 if (predecessor fore null)
    Inserting at end of list--set rear pointer
    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

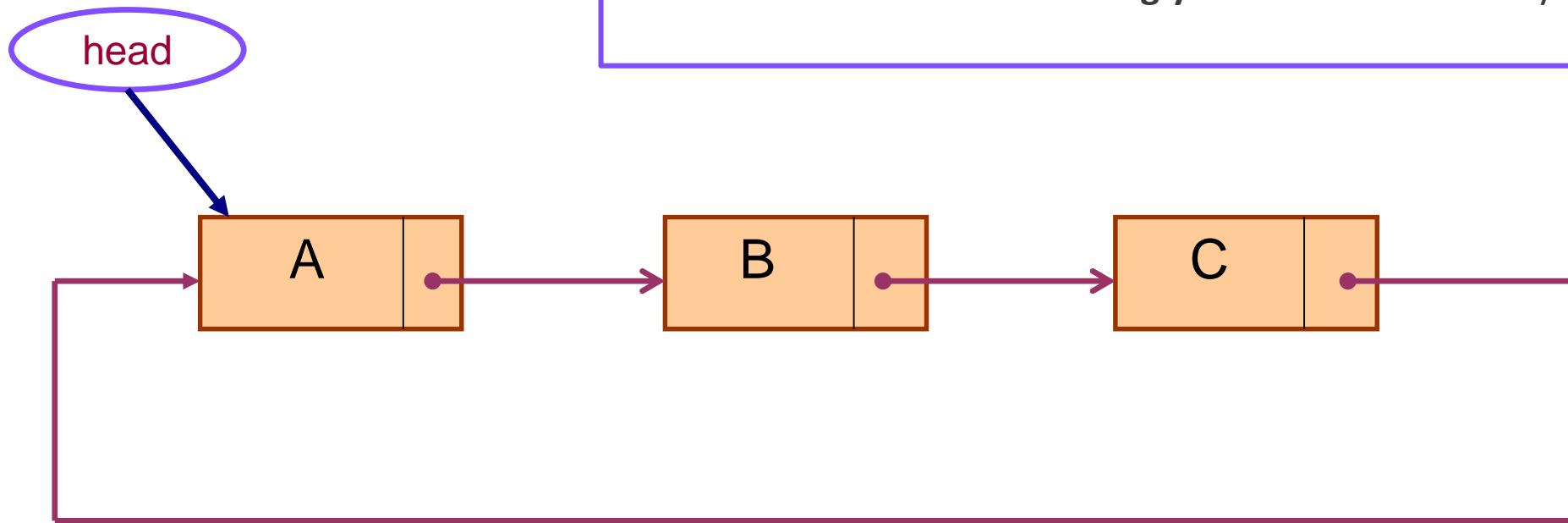
deleteNode is a pointer to the node to be deleted

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.

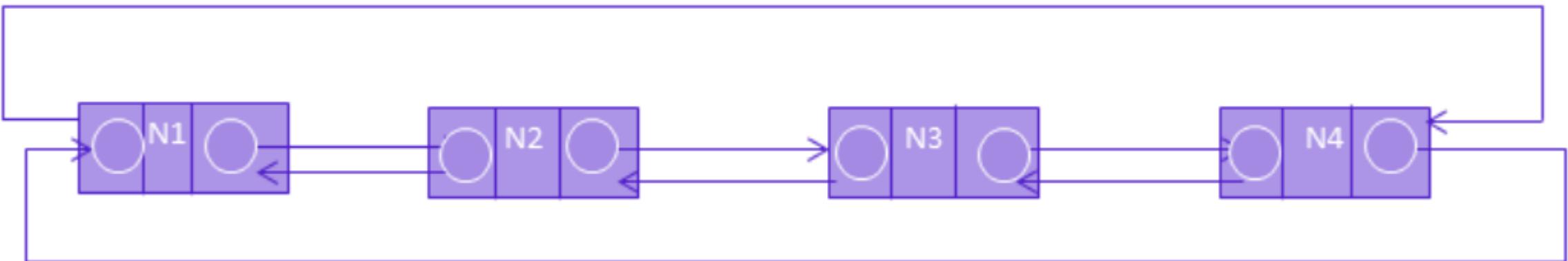


A circular linked list can be a **singly linked list** or a **doubly linked list**.

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

CIRCULAR LINKED LIST.

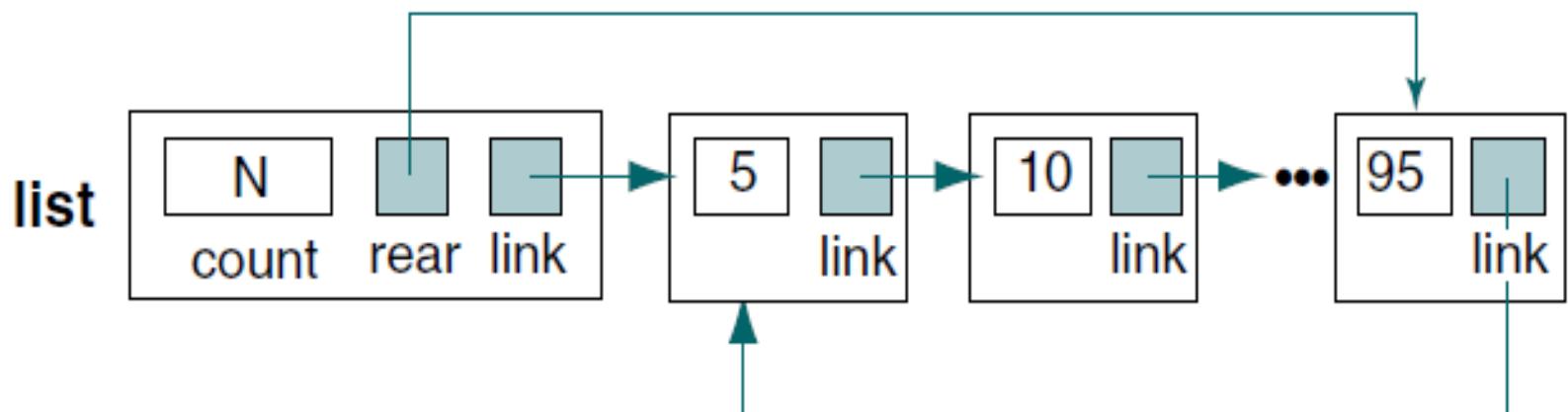
A circular linked list can be a singly linked list or a **doubly linked list**.



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.



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
 - Understand the basic use and processing of general trees



Applications of Trees.

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

Basic Terminology.

About Trees.





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.

- » Linked list is a linear D.S and for some problems it is not possible to maintain this linear ordering.
- » Using non linear D.S such as trees and graphs more complex relations can be expressed.

| Basic Tree concepts.

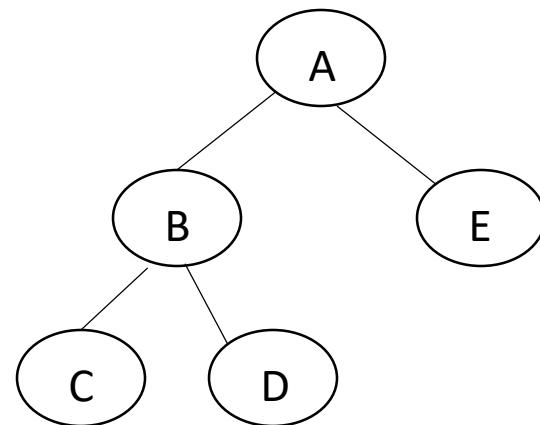
- » When the branch is directed toward the node, it is ***indegree*** branch.
- » When the branch is directed away from the node, it is an ***outdegree*** branch.
- » The ***sum of the indegree and outdegree*** branches is the ***degree*** of the node.
- » If the tree is not empty, the first node is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

Indegree and Outdegree of a node:

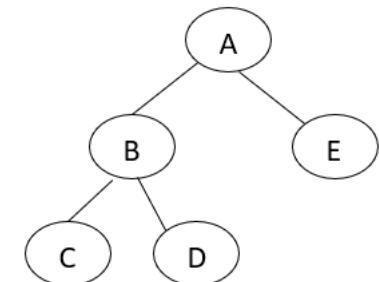
- Indegree of B, C, D and E is 1 and that of A is 0.
- Outdegree of A and B is 2 and that of C, D and E is 0.



Directed tree:

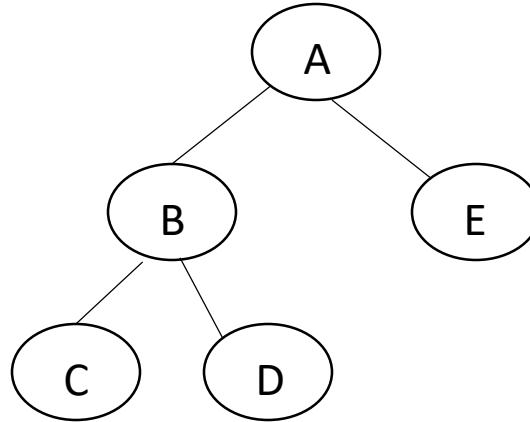
- » Is a tree which has only one node with indegree 0 and all other nodes have indegree 1.
- » The node with indegree 0 is called the root and all other nodes are reachable from root.

Ex: The tree in the above diagram is a directed tree with A as the root and B, C, D and E are reachable from the root.



Ancestors and descendants of a node:

- » In a tree, all the nodes that are reachable from a particular node are called the descendants of that node.



- » Descendants of A are B, C, D and E.
Descendants of B are C and D.
- » Nodes from which a particular node is reachable starting from root are called ancestors of that node.
- » Ancestors of D and C are B and A.
Ancestors of E and B is A.

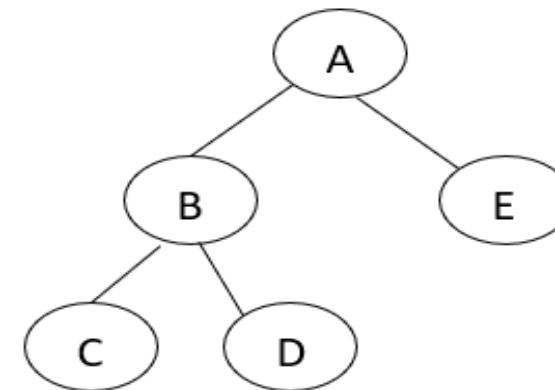
Children of a node:

» The nodes which are reachable from a particular node using only a single edge are called children of that node and this node is called the father of those nodes.

Example:

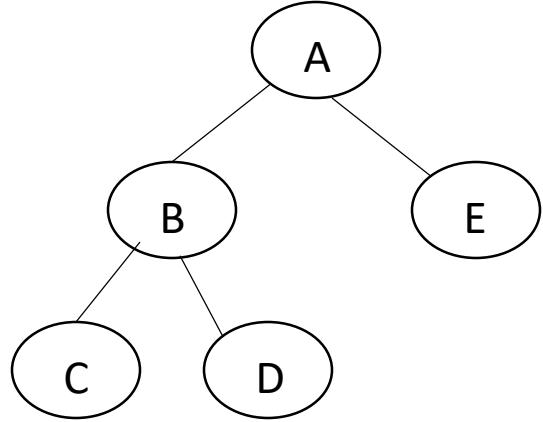
Children of A are B and E.

Children of B are C and D.

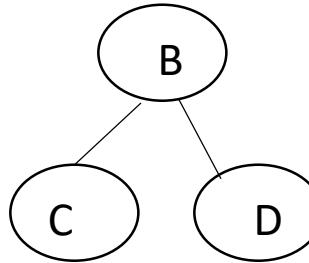


Left subtree and right subtree of a node:

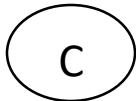
- » All nodes that are all left descendants of a node form the left subtree of that node.



» left subtree of A is

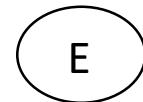


» left subtree of B is

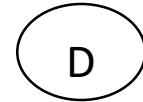


» All nodes that are right descendants of a node form the right subtree of that node.

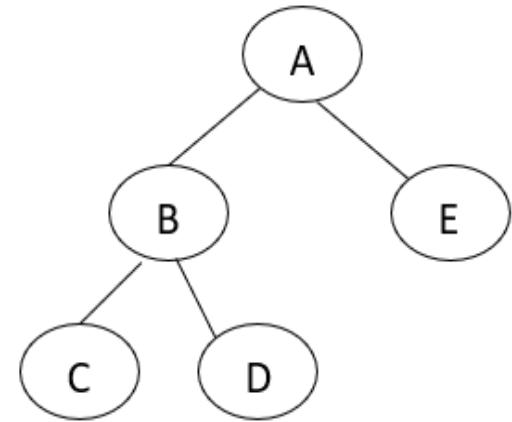
» right subtree of A is



» right subtree of B is



» right subtree of E is empty tree.



Terminal node or leaf node:

- » All nodes in a tree with outdegree zero are called the terminal nodes, leaf nodes or external nodes of the tree.
- » All other nodes other than leaf nodes are called non leaf nodes or internal nodes of the tree.

In the previous tree, C, D and E are leaf nodes and A, B are non leaf nodes.

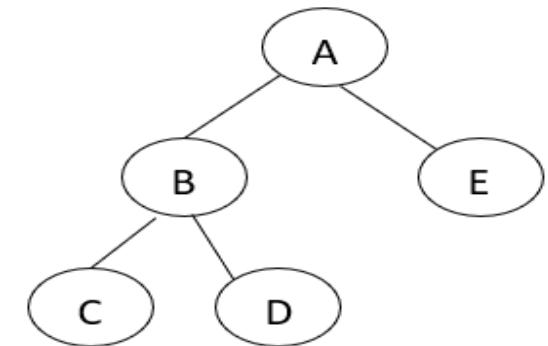
Levels of a tree:

- » Level of a node is the number of edges in the path from root node.

Level of A is 0.

Level of B and E is 1.

Level of C and D is 2.



Height of a tree:

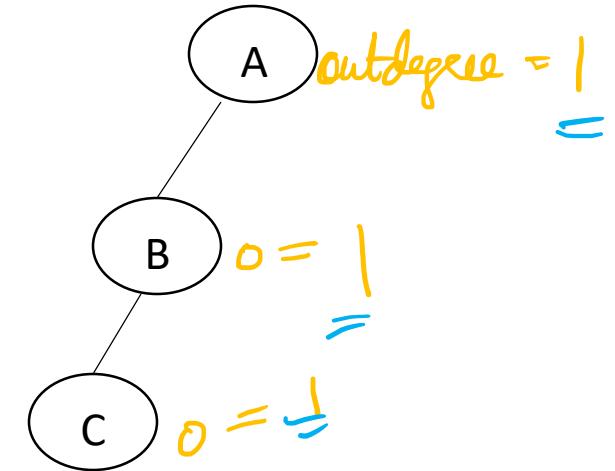
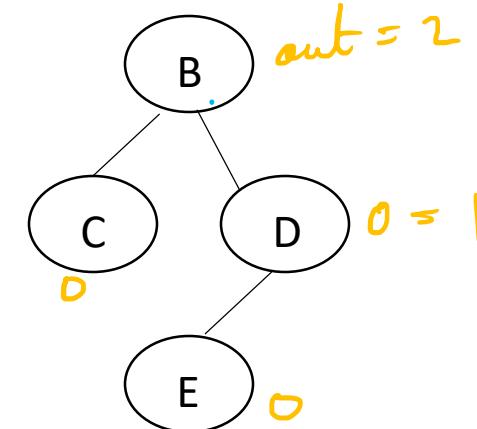
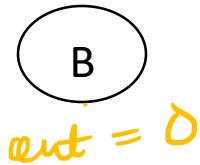
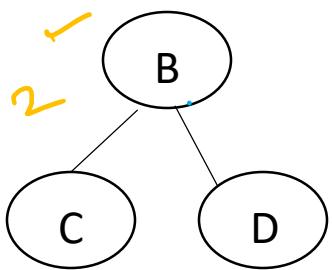
- » Height of a tree is one more than the maximum level in the tree.

In the previous tree , maximum level is 2 and height is 3

Binary trees:

- » A binary tree is a directed tree in which outdegree of each node is less than or equal to 2. i.e each node can have 0, 1 or 2 children.

Examples of binary tree:



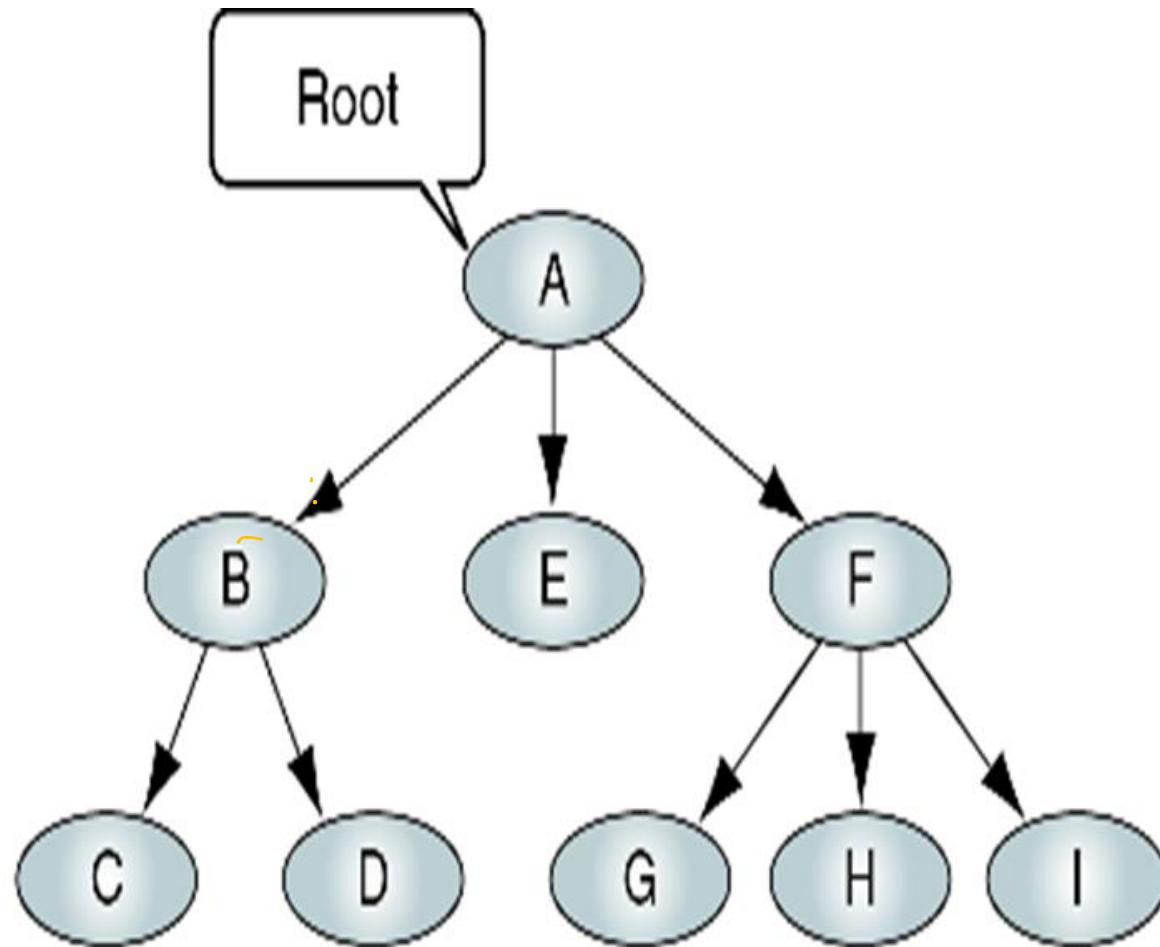
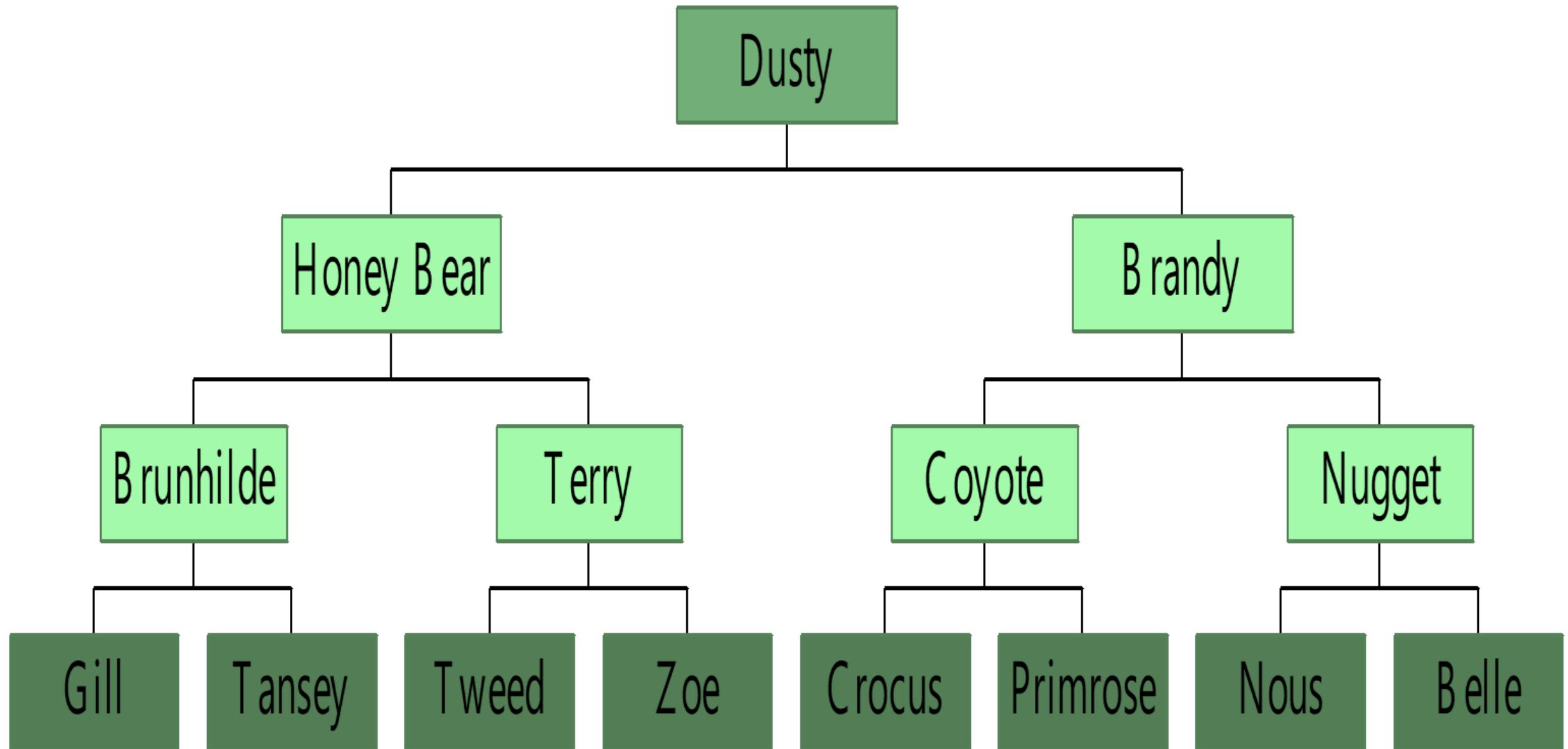


FIGURE 6-1 Tree

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.

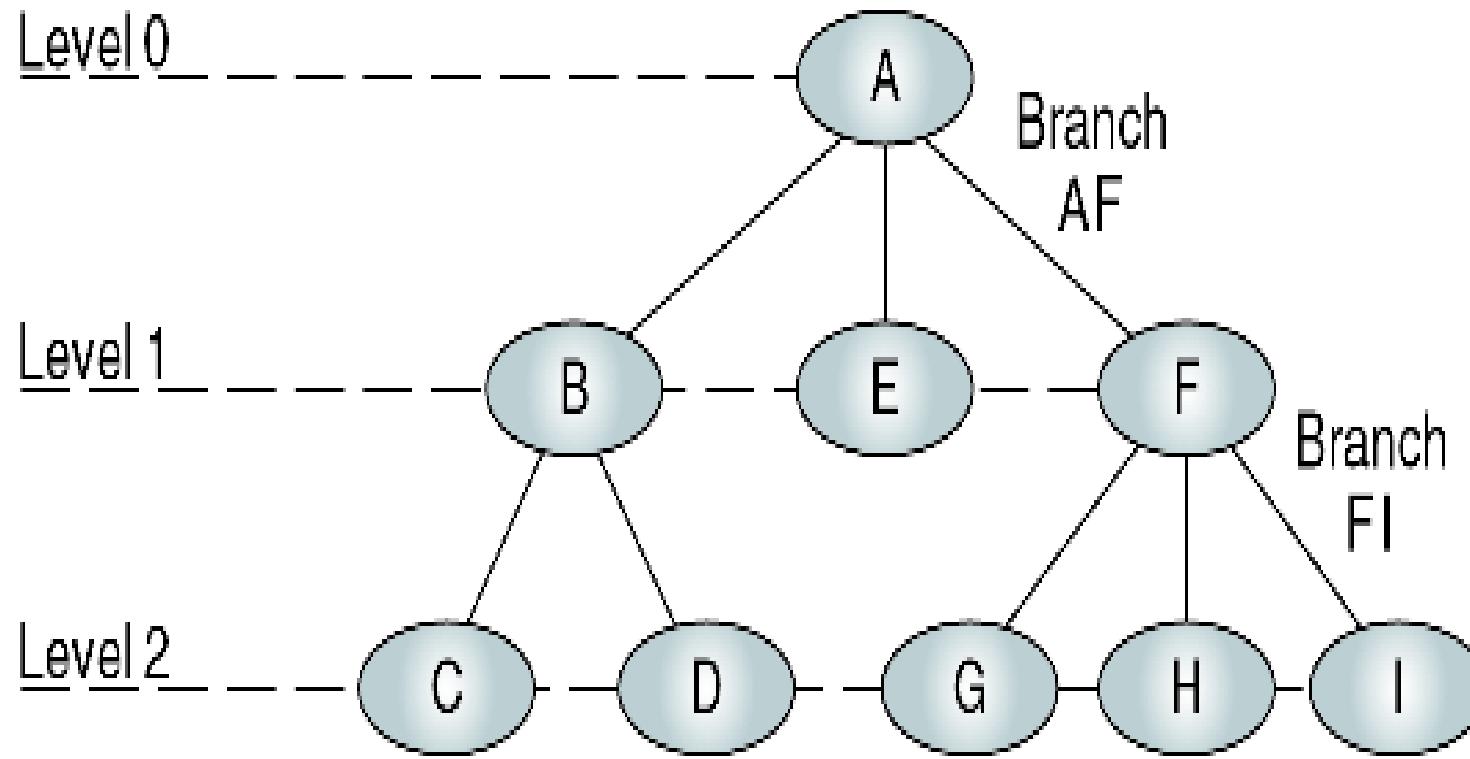
- » The **height of the tree** is the level of the leaf in the longest path from the root plus 1.
- » By definition the **height** of any **empty tree** is -1.
- » (Empty (Null)-tree: a tree without any node)
- » A **subtree** is any connected structure below the root.
- » The **first node in the subtree** is known as the ~~root~~ of the subtree.

BINARY TREE

A binary tree is a tree in which no node can have more than two children. Typically these children are described as “left child” and “right child” of the parent node.

A binary tree T is defined as a finite set of elements, called nodes, such that :

1. T is empty (i.e., if T has no nodes called the *null tree* or *empty tree*).
2. T contains a special node R, called root node of T, and the remaining nodes of T form an ordered pair of disjoined binary trees T₁ and T₂, and they are called left and right sub tree of R. If T₁ is non empty then its root is called the left successor of R, similarly if T₂ is non empty then its root is called the right successor of R.



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

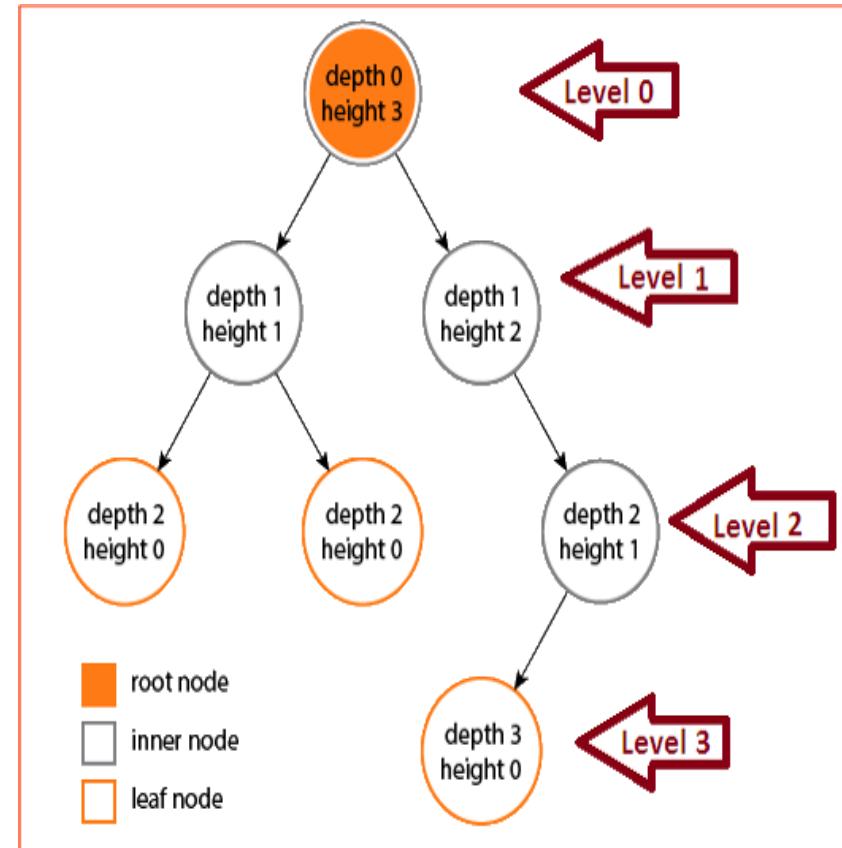
Siblings: {B,E,F}, {C,D}, {G,H,I}
Leaves: C,D,E,G,H,I
Internal nodes: B,F

FIGURE 6-2 Tree Nomenclature

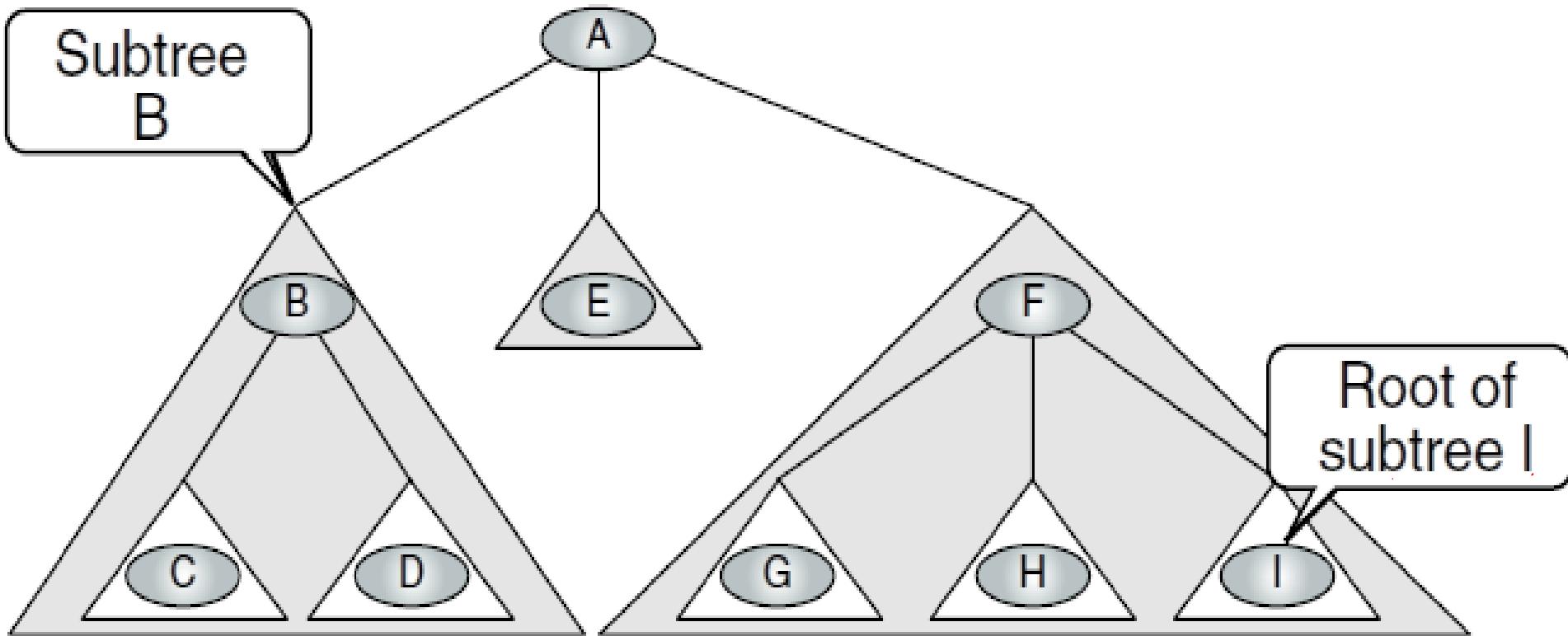
Trees

■ Tree (Continued...)

- Height of a node
 - ▶ No. of edges from the node to the deepest leaf
- Depth of a node
 - ▶ No. of edges from the root to the node
- Height or depth of a tree
 - ▶ Maximum level in the tree + 1



SUB TREES.



Subtrees

Binary tree

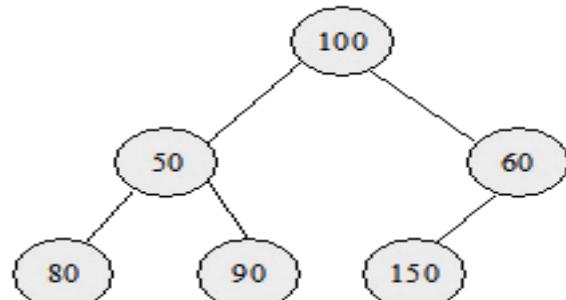
Definition:

- » A binary tree is a finite set of elements that is either empty or partitioned into 3 disjoint subsets. The first subset contains root of the tree and other two subsets themselves are binary trees called left and right subtree. Each element of a binary tree is called a **NODE** of the tree.

Binary Trees

■ Binary Tree

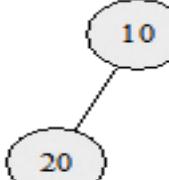
- A tree in which outdegree of each node is less than or equal to 2 (each node has 0, 1 or 2 children)



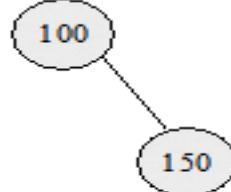
(a)



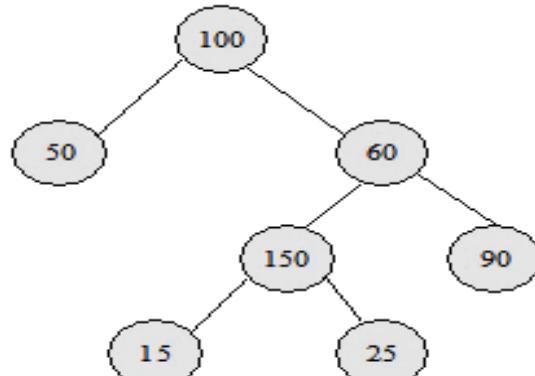
(b)



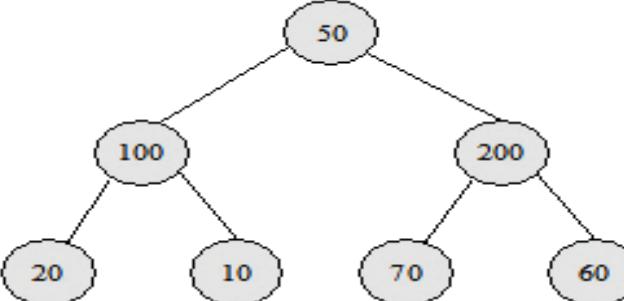
(c)



(d)



(e)



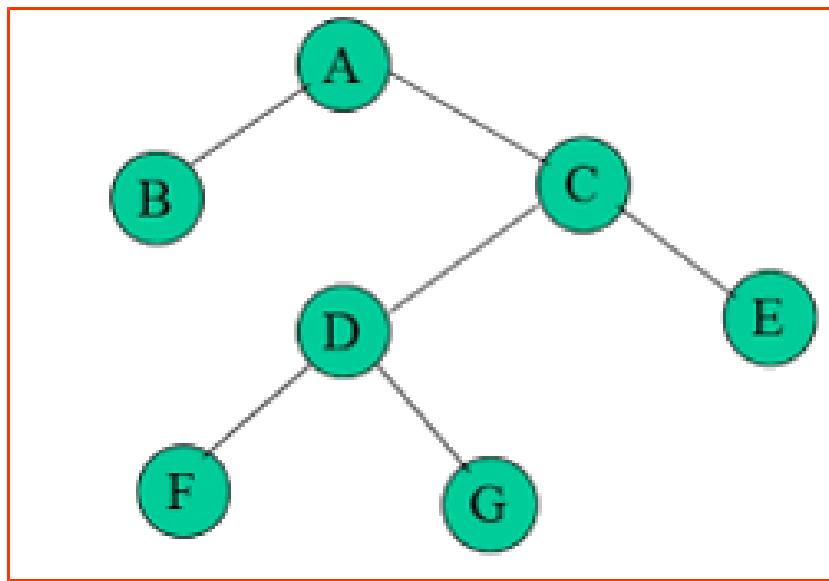
(f)

Binary Trees

■ Binary Tree (Continued...)

- Strict binary tree:

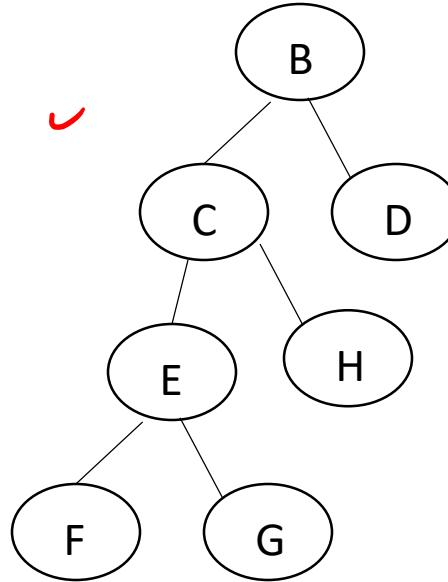
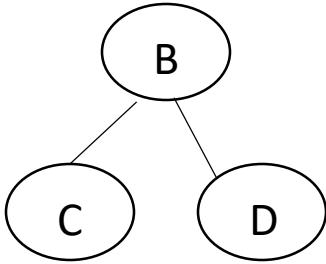
- ▶ A binary tree in which every node has either 0 or 2 children



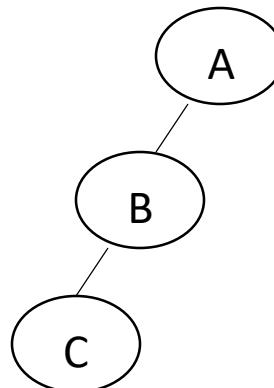
- ▶ Examples: (b), (e) and (f)

Strictly binary tree:

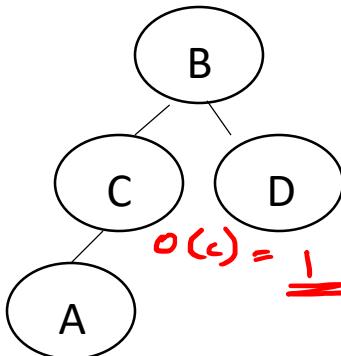
» If the out degree of every node in a tree is either 0 or 2(1 not allowed), then the tree is strictly binary tree.



Tress which are binary trees but not strictly binary:



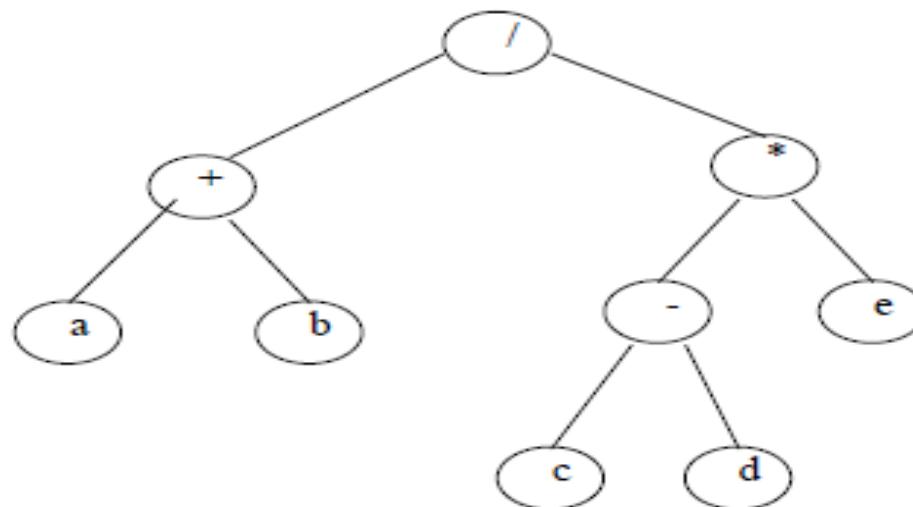
$$o(A) = 1$$



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

Strictly binary tree

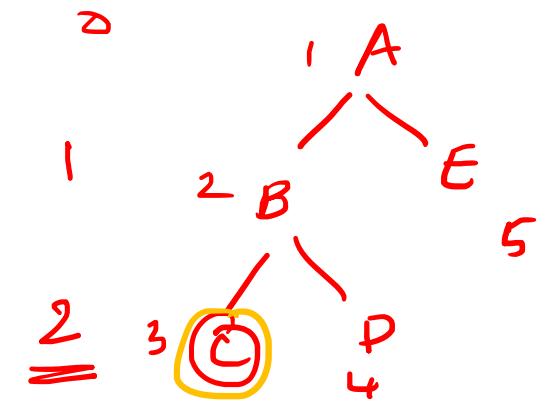
- » It is also called as **2-tree or extended binary tree**.
- » The main application of a 2-tree is to represent and compute any algebraic expression using binary operation.
- » Consider an algebraic expression E. $E = (a + b)/((c - d)^*e)$
- » E can be represented by means of the extended binary tree T.
- »
- » Each variable or **constant in E appears as an internal node** in T whose left and right sub tree corresponds to the **operands** of the operation.



- » 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.

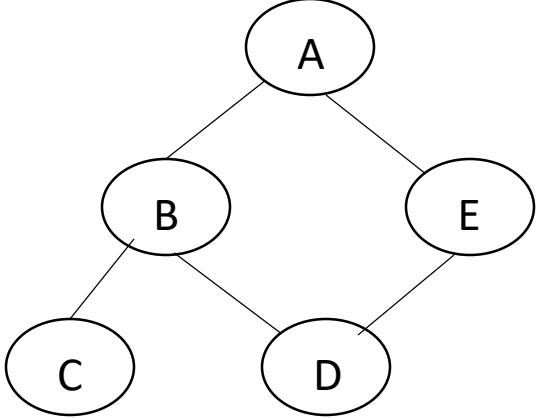


$$n = 3$$

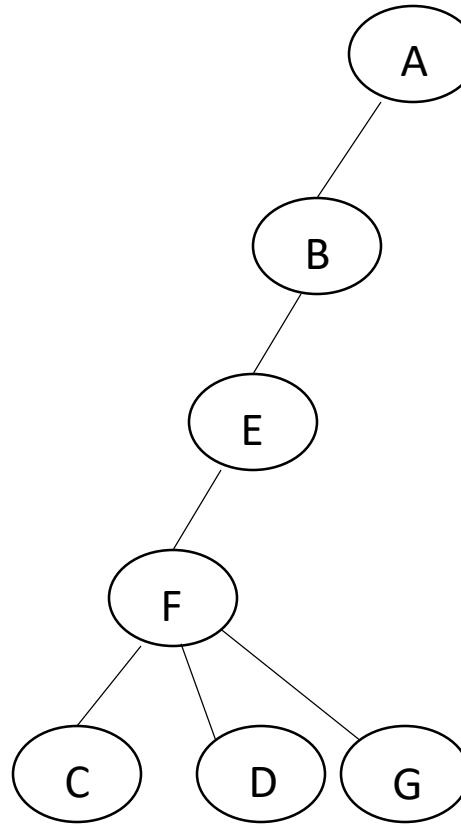
$$(2n - 1)$$

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

Examples of non binary trees:



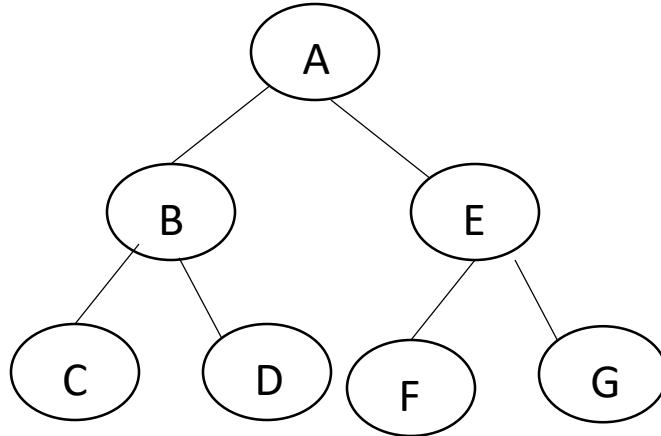
D has indegree 2, hence
not directed tree



F has outdegree 3.

Complete binary tree:

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



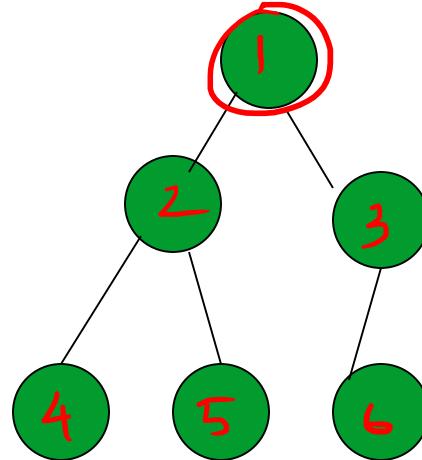
Number of nodes at level 0(root level) is $\text{pow}(2,0) \rightarrow 1$

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 $\text{pow}(2,2) \rightarrow 4$

Almost Complete BT(nearly complete)

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



| 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.

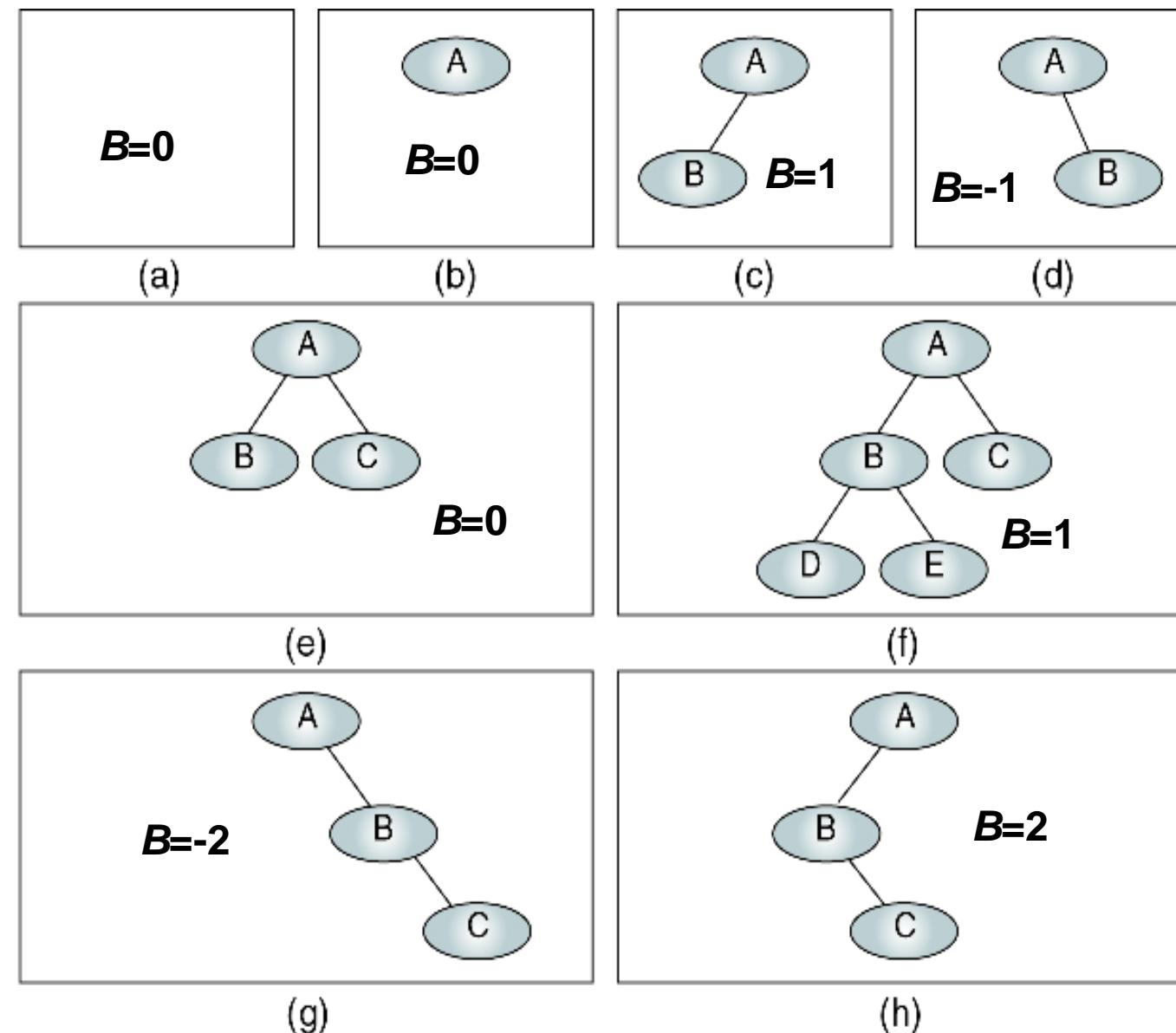
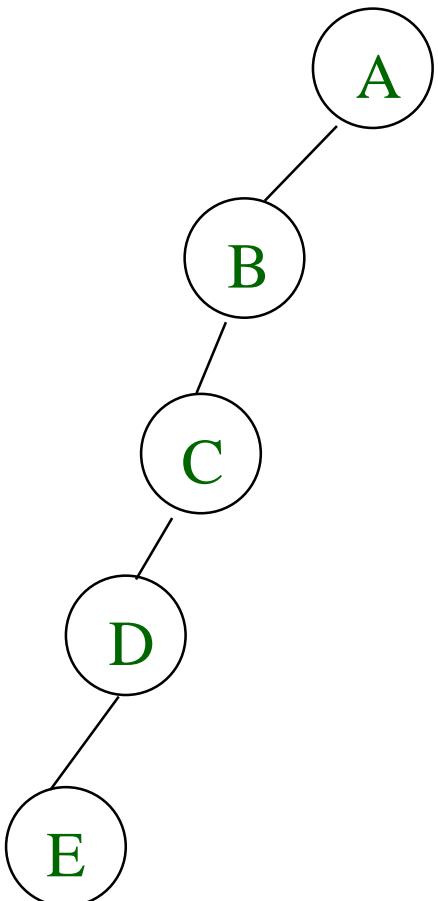


FIGURE 6-6 Collection of Binary Trees

Some Properties of Binary Trees.

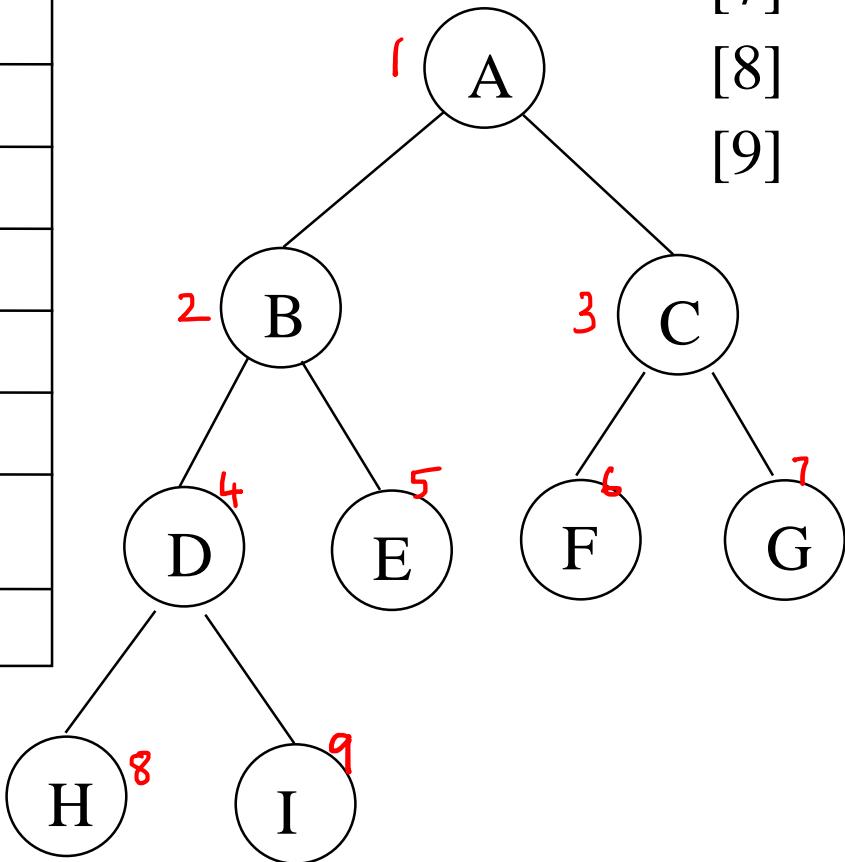
» In the **balanced binary tree** (definition of Russian mathematicians Adelson-Velskii and Landis) the height of its subtrees differs by no more than one (its balance factor is -1, 0, or 1), and its subtrees are also **balanced**.

Sequential Representation



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

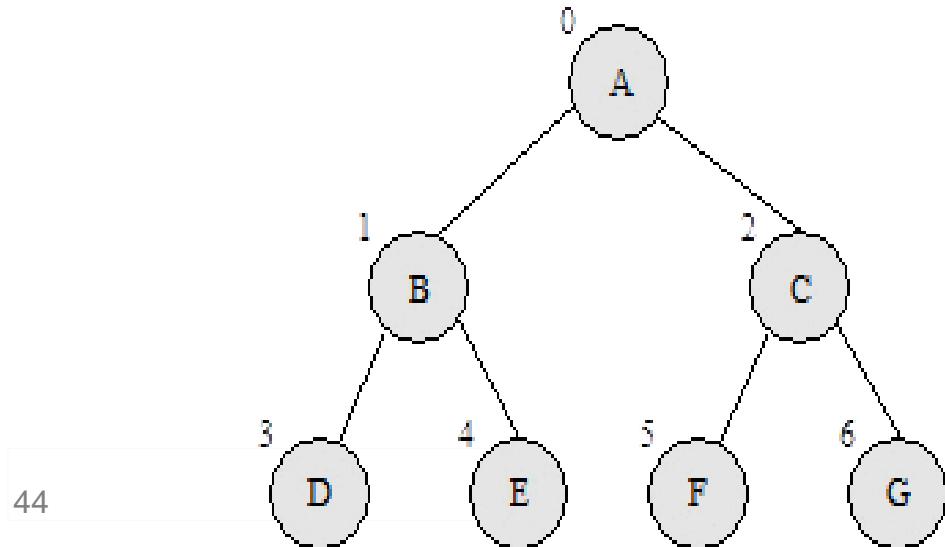
- (1) waste space
(2) insertion/deletion problem



Binary Trees

■ Storage Representation

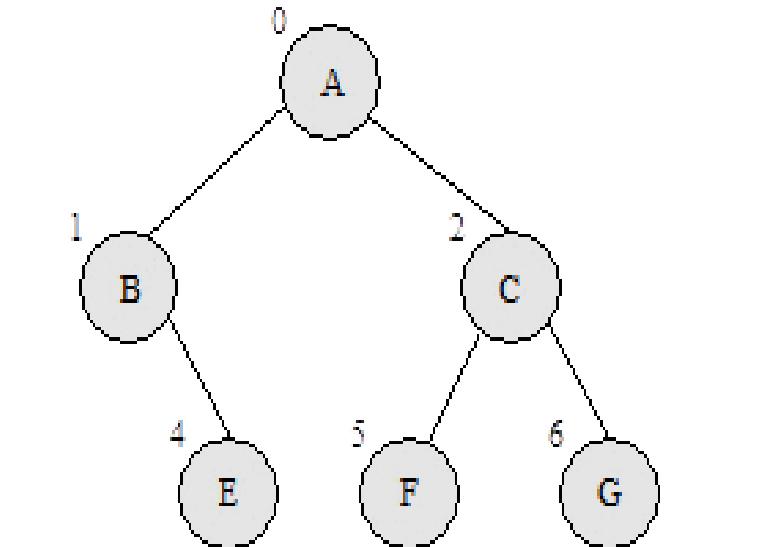
- Sequential (array) representation



44

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

Left child $2k+1$



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

Right child $2k+2$

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

Binary Trees

■ Storage Representation (Continued...)

- Linked representation

- ▶ Node has three fields

- info
 - leftChildPtr
 - rightChildPtr

- ▶ Implement using structure

```
struct NODE
```

```
{
```

```
    int info;
```

```
    struct NODE *leftChildPtr;
```

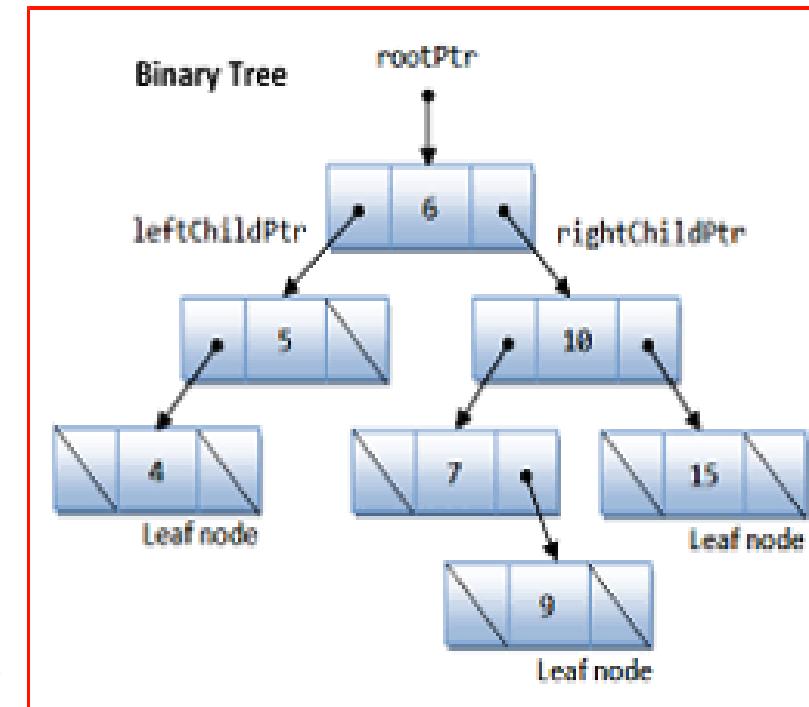
```
    struct NODE *rightChildPtr;
```

```
} ;
```

```
struct NODE node;
```

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

45



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;
};

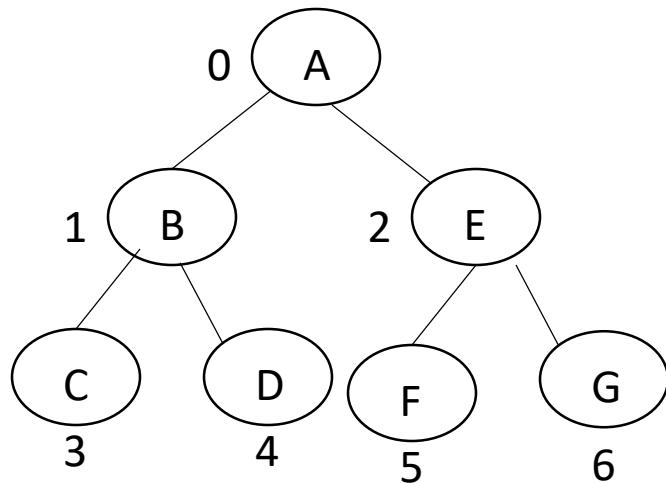
typedef struct node *NODEPTR;
```

Implementing a binary tree:

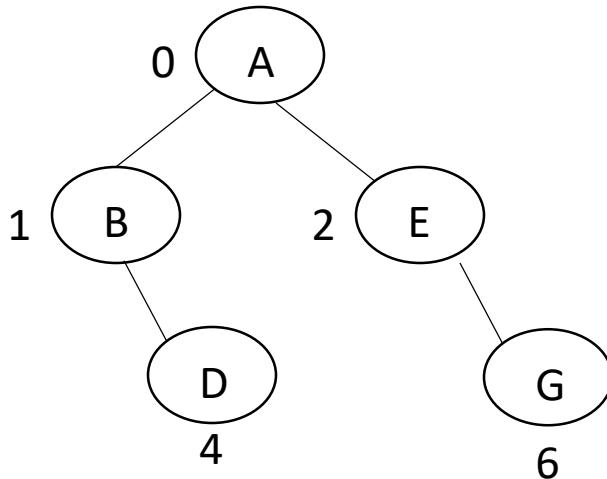
- » A pointer variable root is used to point to the root node.
- » Initially root is NULL, which means the tree is empty.

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’ of 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 father.

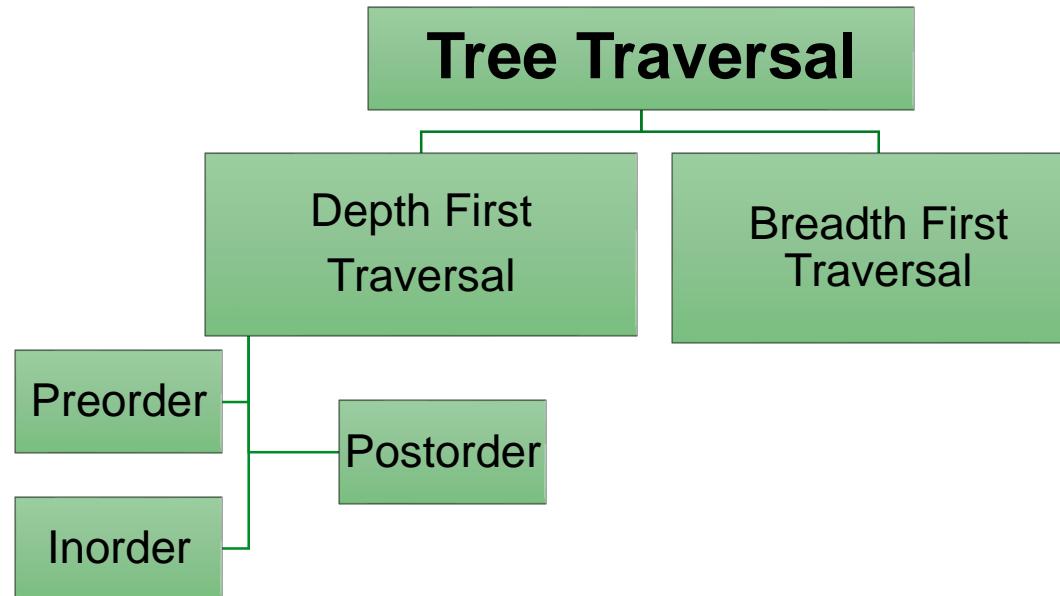
Position of E is 2. $(2-1)/2$ gives 0, which is the position of its father A.

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.

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 ALGORITHM

» USE Stack.

Steps:

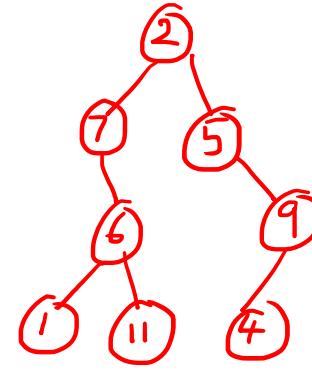
- » Add root to the Stack.
- » Pop out an element from Stack , process it, and add its right and left children to stack.
- » Repeat the above two steps until the Stack id empty.

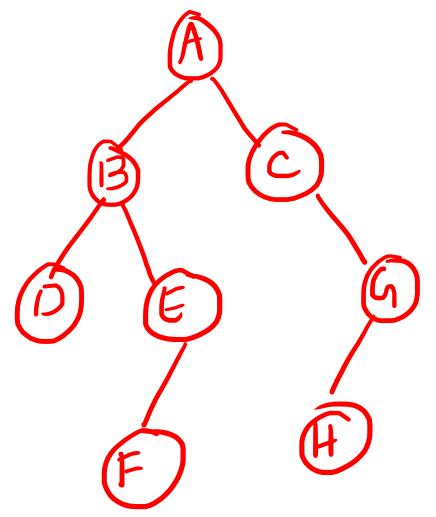
Example 1 :-

Steps
1.



Output string





Homework -

BREADTH FIRST TRAVERSAL ALGORITHM

» USE Queue.

Steps:

- » 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.

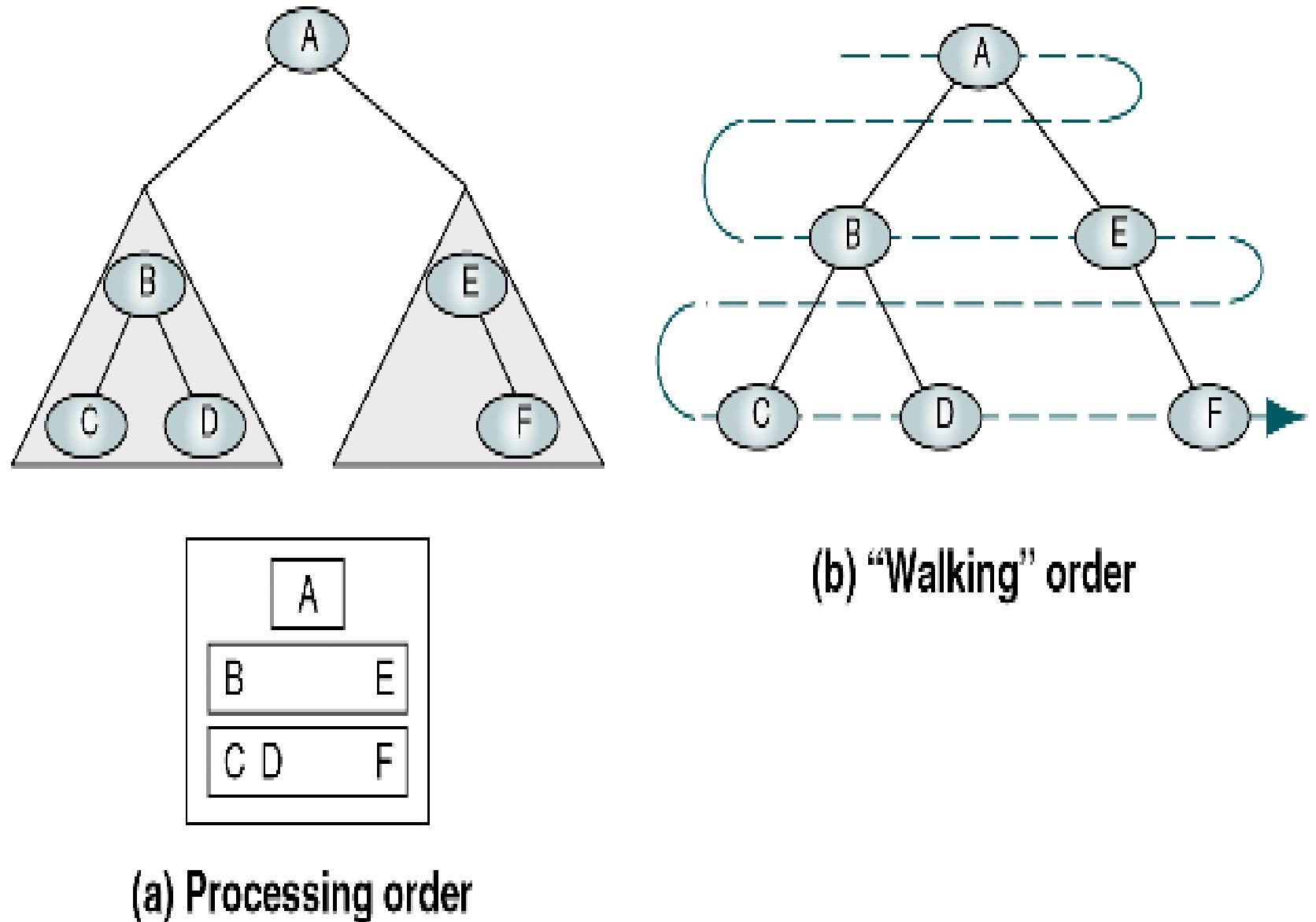
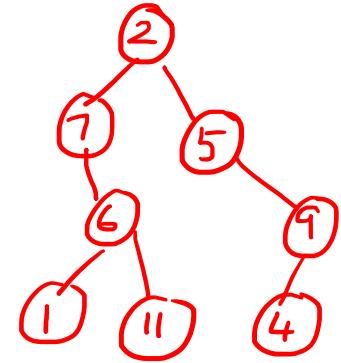
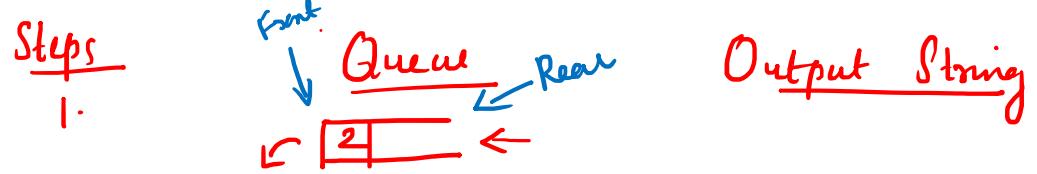
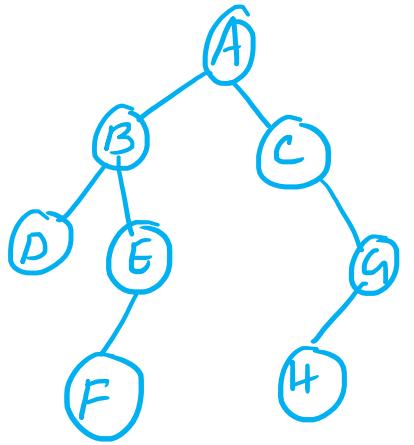


FIGURE 6-14 Breadth-first Traversal

Example 1:-



Homework



DEPTH FIRST SEARCH ALGORITHM

Put the root node onto the stack

while (stack is not empty)

do

 remove a node from the stack;

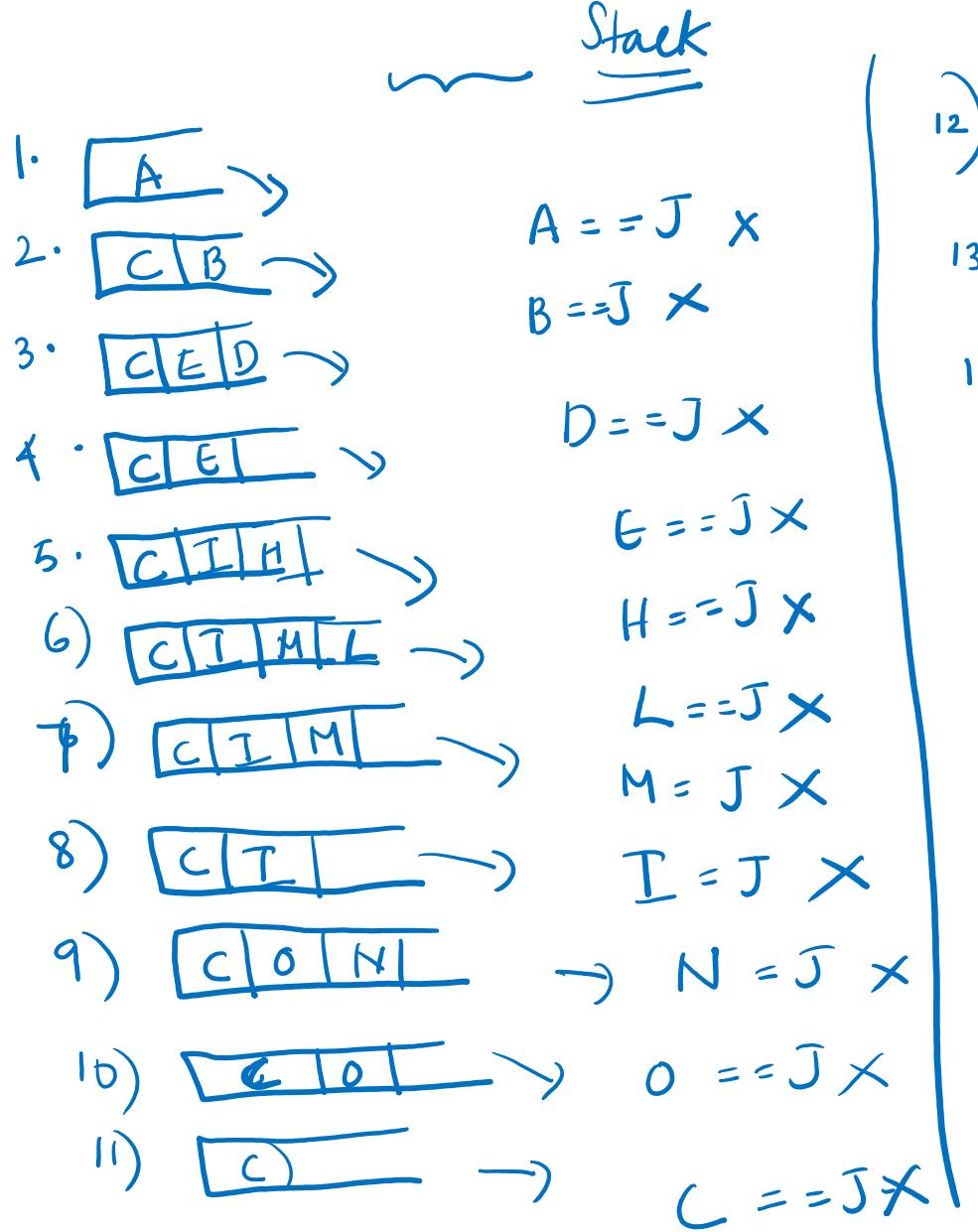
 if (node is a goal node)

 return success;

 Put its **right and left children** onto the stack;

done

return failure;



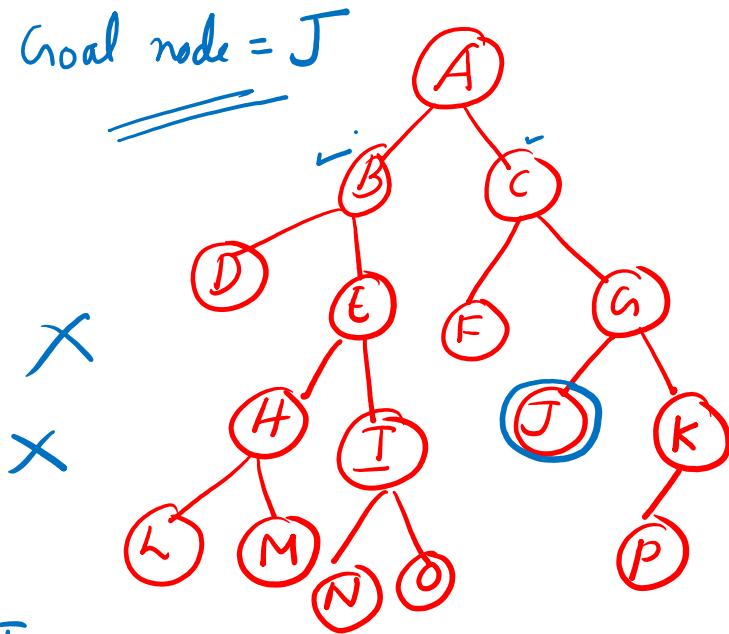
12.) $\boxed{G | F | I} \rightarrow$

13.) $\boxed{G | I} \rightarrow$

14.) $\boxed{K | J | I} \rightarrow$

$f == J \times$
 $g == J \times$

$J == J \checkmark$



BREADTH FIRST SEARCH ALGORITHM.

Add root to queue

while (queue is not empty)

do

remove a node from the queue;

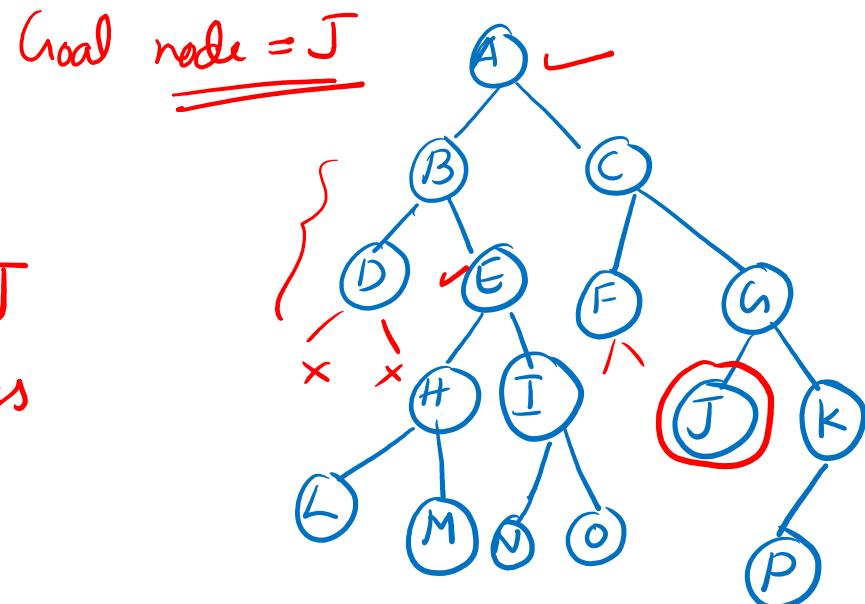
if (node is a goal node) return success;

put its **left and right children** onto the queue;

done

return failure;

- Queue
1. $\xleftarrow{\text{Front}} \boxed{A} \xrightarrow{\text{Rear}}$ $A == J \times$
 2. $\xleftarrow{\text{Front}} \boxed{B | C} \xrightarrow{\text{Rear}}$ $B == J \times$
 3. $\xleftarrow{\text{Front}} \boxed{C | D | E} \xrightarrow{\text{Rear}}$ $C == J \times$
 4. $\xleftarrow{\text{Front}} \boxed{D | E | F | G} \xrightarrow{\text{Rear}}$ $D == J \times$
 5. $\xleftarrow{\text{Front}} \boxed{E | F | G} \xrightarrow{\text{Rear}}$ $E == J \times$
 6. $\xleftarrow{\text{Front}} \boxed{F | G | H | I} \xrightarrow{\text{Rear}}$
 7. $\xleftarrow{\text{Front}} \boxed{G | H | I} \xrightarrow{\text{Rear}}$ $F == J \times$
 8. $\xleftarrow{\text{Front}} \boxed{H | I | J | K} \xrightarrow{\text{Rear}}$ $G == J \times$
 9. $\xleftarrow{\text{Front}} \boxed{I | J | K | L | M} \xrightarrow{\text{Rear}}$ $H == J \times$
 10. $\xleftarrow{\text{Front}} \boxed{J | K | L | M | N | O} \xrightarrow{\text{Rear}}$ $I == J \times$
- (11) $J == J$
 $\xrightarrow{\text{Rear}}$ K | L | M | N | O success



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

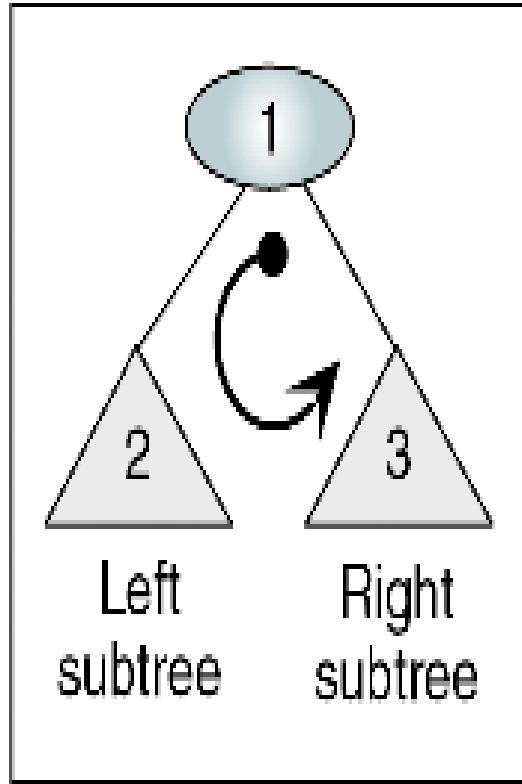
- » We process all the descendants of a child before going on to the next child.

- » Is visiting each node exactly once systematically one after the other.

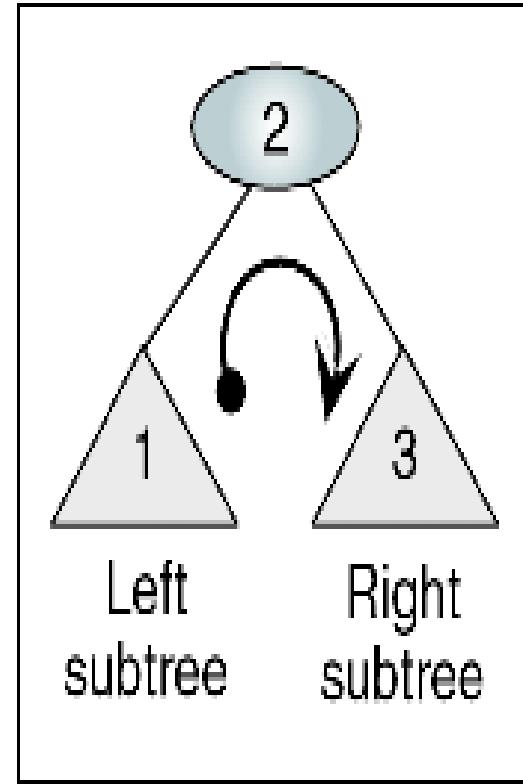
There are 3 main traversals techniques

- » **Inorder traversal**
- » **Preorder traversal**
- » **Postorder traversal**

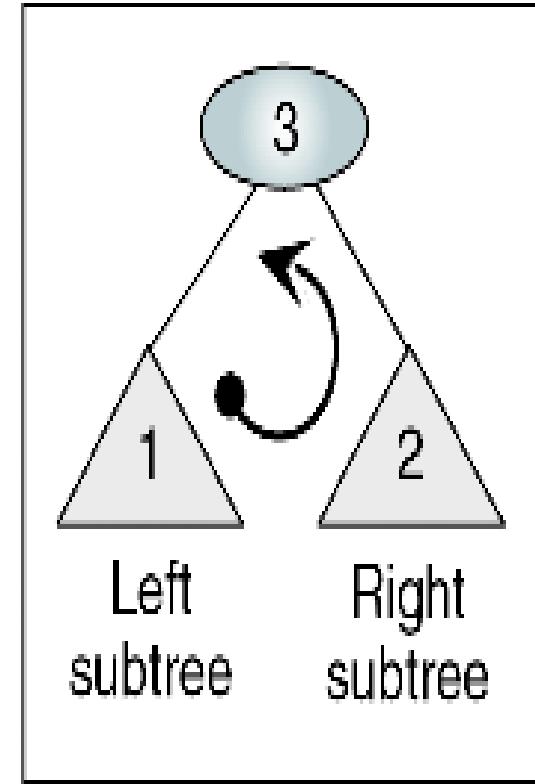
P L R



L P R



L R P



(a) Preorder traversal

(b) Inorder traversal

(c) Postorder traversal

FIGURE 6-8 Binary Tree Traversals

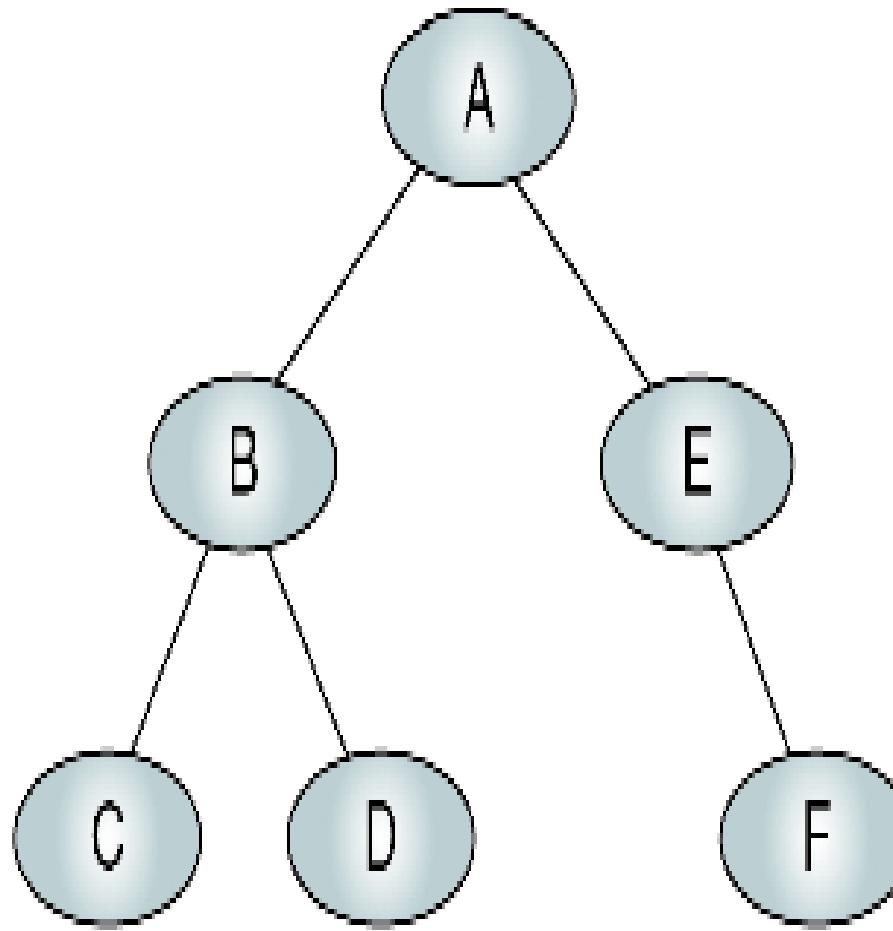
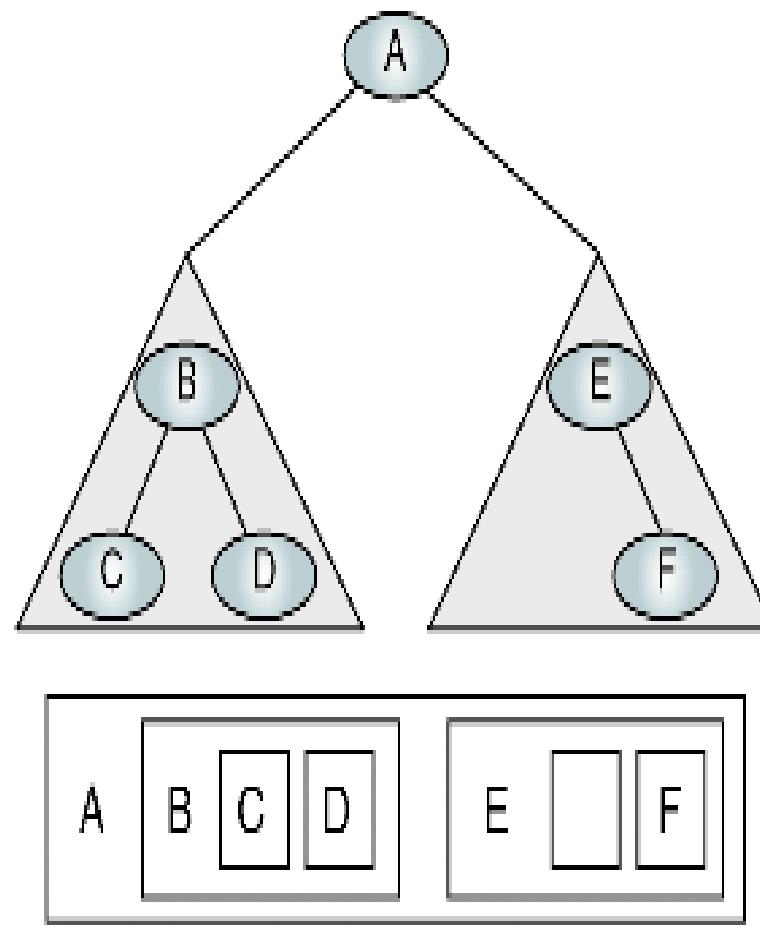


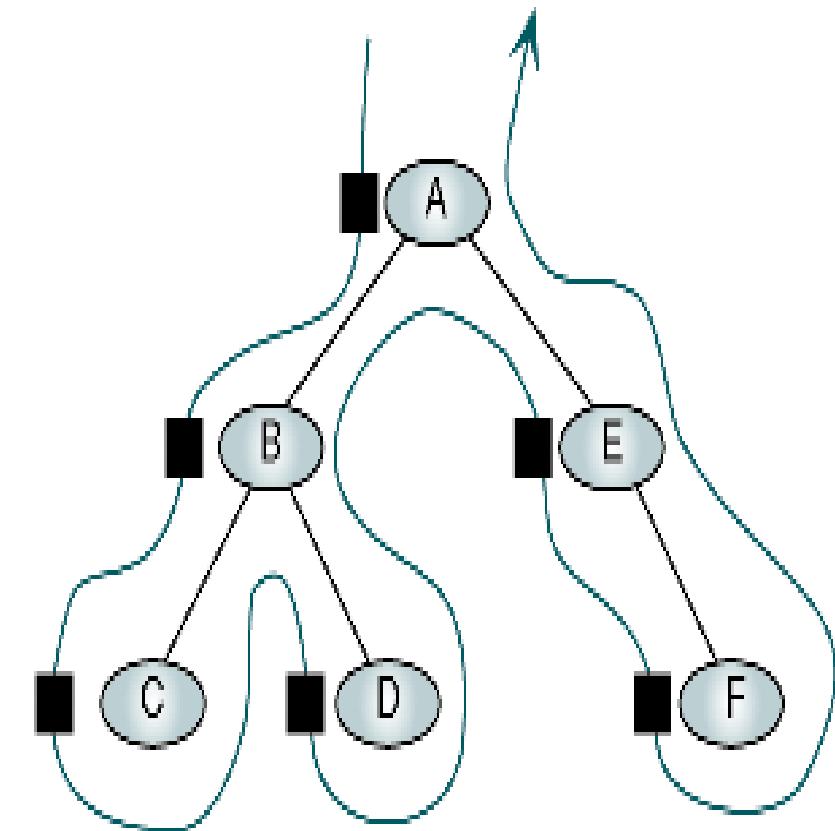
FIGURE 6-9 Binary Tree for Traversals

Preorder Traversal

1. Process the node.
 2. Traverse the left subtree in preorder.
 3. Traverse the right subtree in preorder.
-
- In preorder, we first visit the node, then move towards left and then to the right recursively.

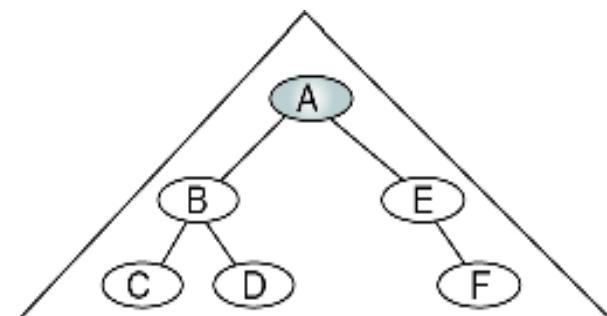


(a) Processing order

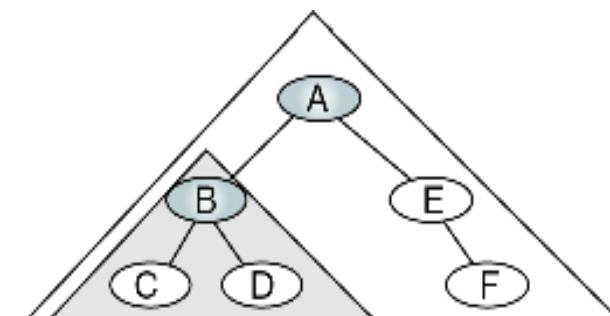


(b) "Walking" order

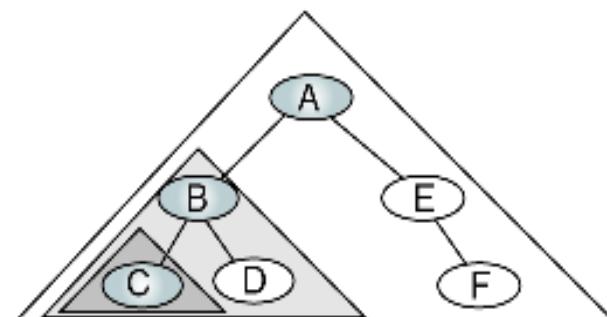
FIGURE 6-10 Preorder Traversal—A B C D E F



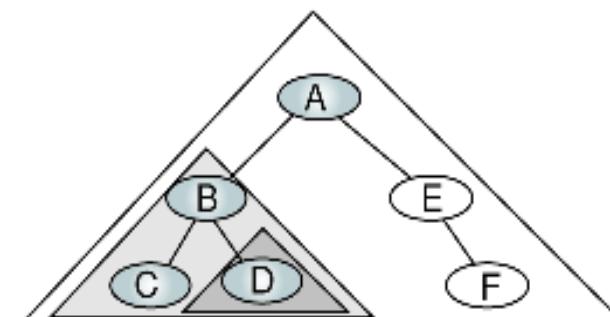
(a) Process tree A



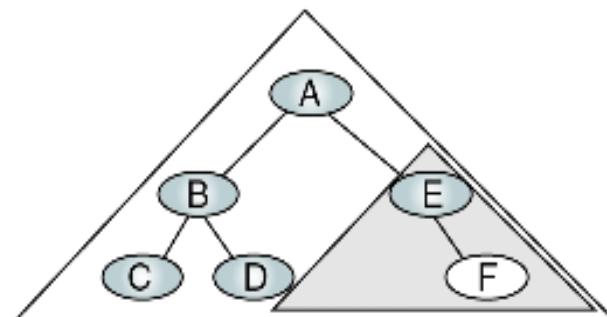
(b) Process tree B



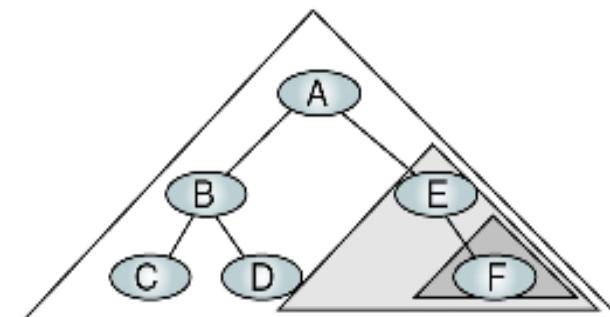
(c) Process tree C



(d) Process tree D



(e) Process tree E



(f) Process tree F

FIGURE 6-11 Algorithmic Traversal of Binary Tree

ALGORITHM 6-2 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

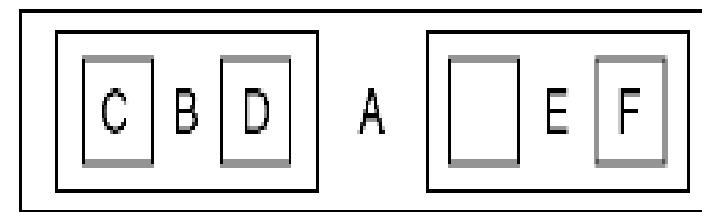
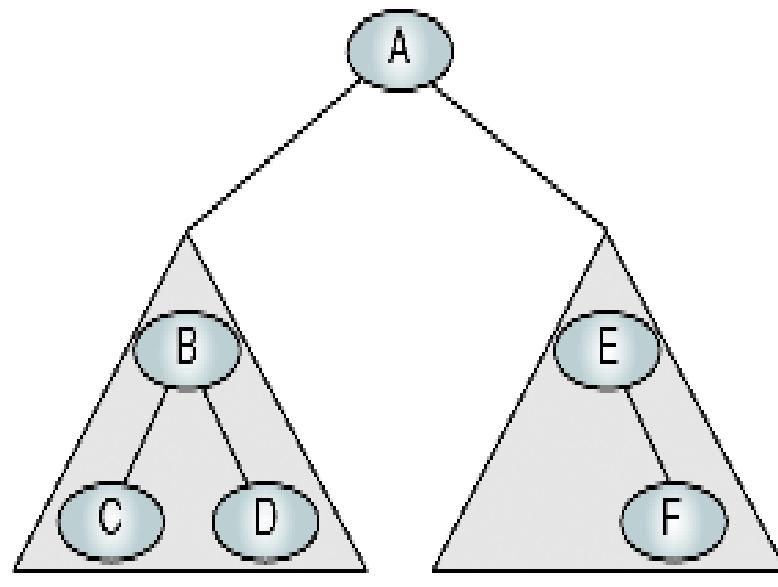
Preorder Traversal (recursive version)

```
void preorder(node *root)
/* preorder tree traversal */
{
    if (root) {
        cout<<root->data;
        preorder(root->left_child);
        preorder(root->right_child);
    }
}
```

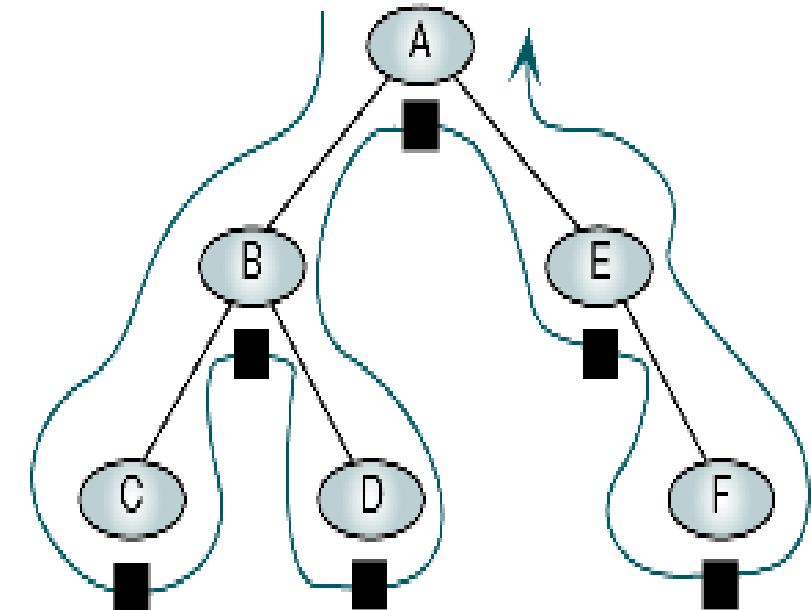
+ * * / A B C D E

Inorder traversal

1. Traverse the left subtree in inorder.
 2. Process the root node.
 3. Traverse the right subtree in inorder.
-
- In inorder traversal, move towards the left of the tree(till the leaf node), display that node and then move towards right and repeat the process
 - Since same process is repeated at every stage, recursion will serve the purpose.



(a) Processing order



(b) "Walking" order

FIGURE 6-12 Inorder Traversal—CBDAEF

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
```

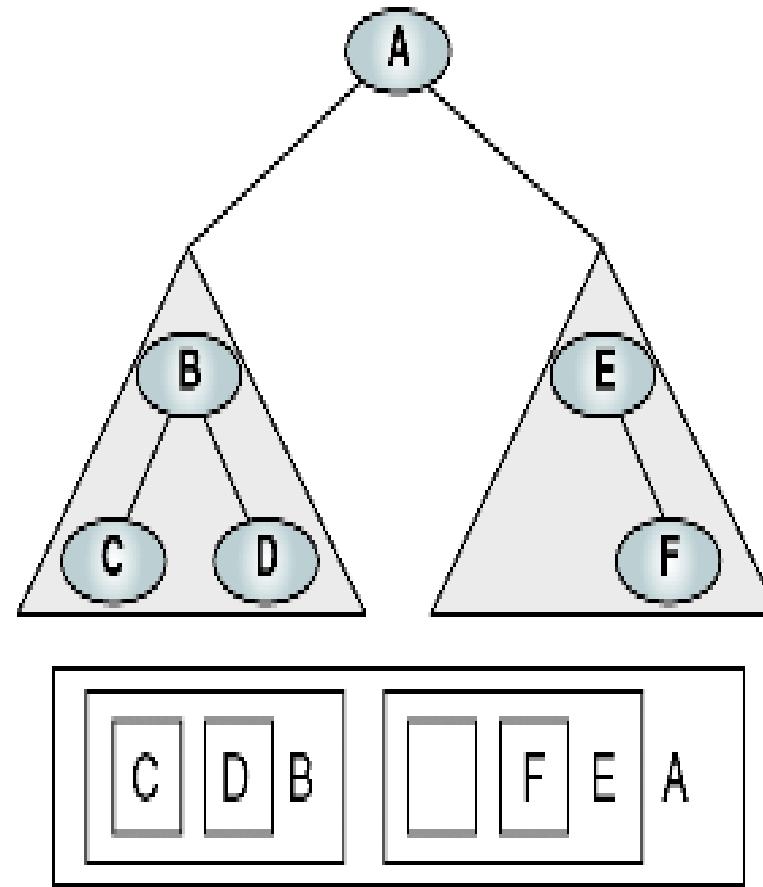
Inorder Traversal (recursive version)

```
void inorder(node *root)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(root->left_child);
        cout<<root->data;
        inorder(root->right_child);
    }
}
```

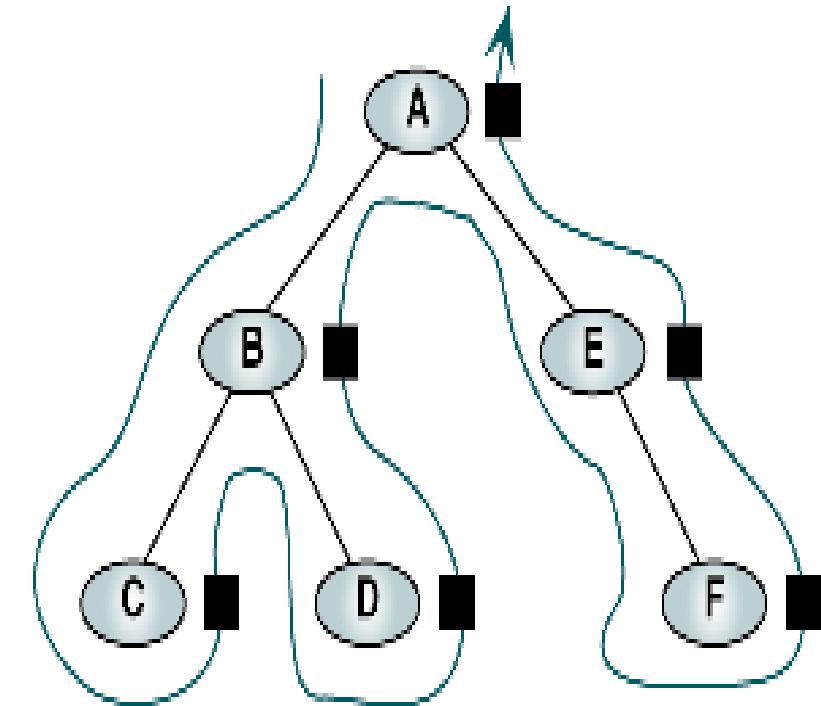
$$A / B * C * D + E$$

Post order traversal

1. Traverse the left subtree in postorder.
 2. Traverse the right subtree in postorder.
 3. Process the root node.
-
- In post order traversal, we first traverse towards left, then move to right and then visit the root. This process is repeated recursively.



(a) Processing order



(b) "Walking" order

FIGURE 6-13 Postorder Traversal—C D B F E A

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)

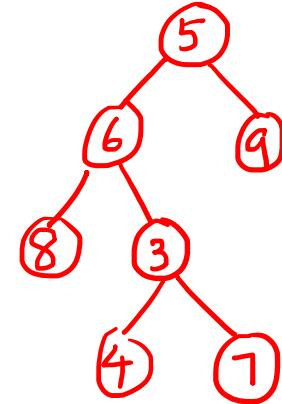
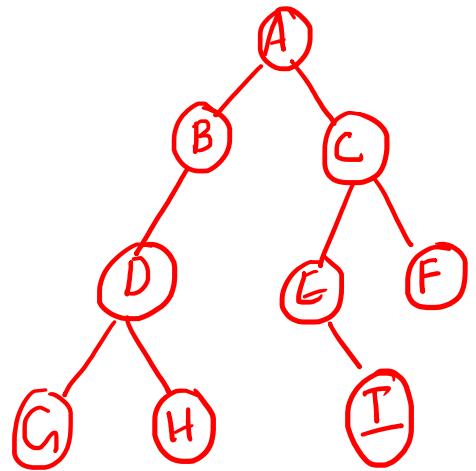
2 end if

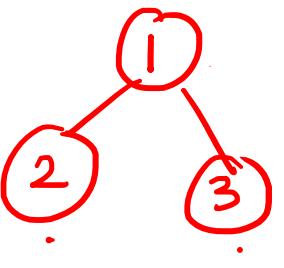
end postOrder

Postorder Traversal (recursive version)

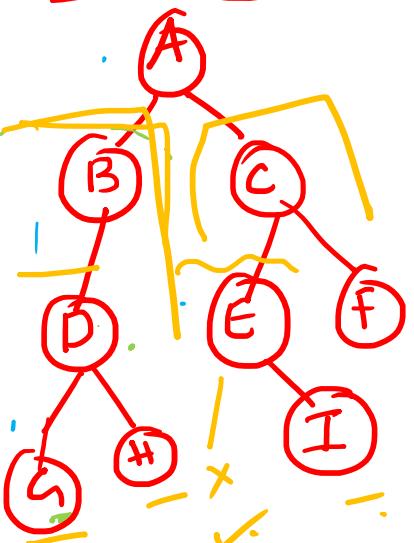
```
void postorder(node *root)
/* postorder tree traversal */
{
    if (root) {
        postorder(root->left_child);
        postorder(root->right_child);
        cout<<root->data;
    }
}
```

A B / C * D * E +

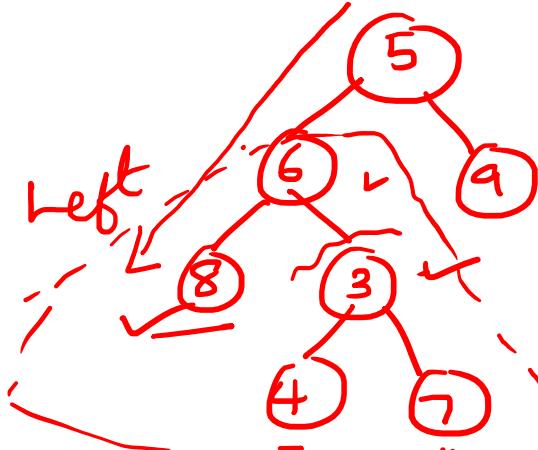




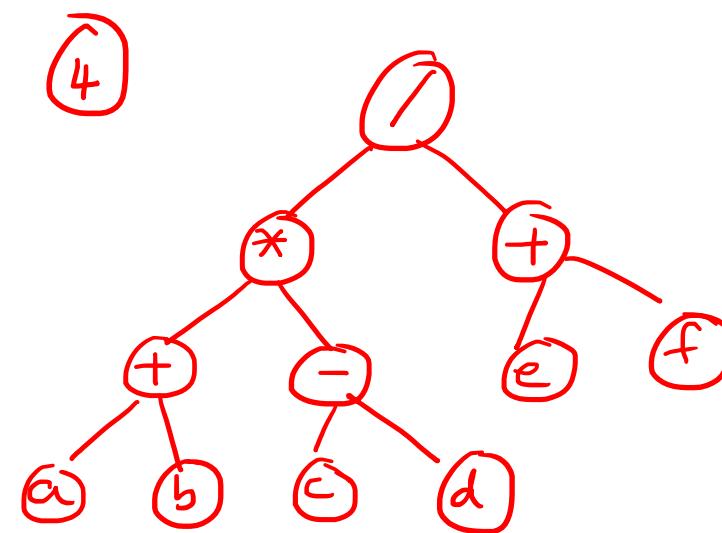
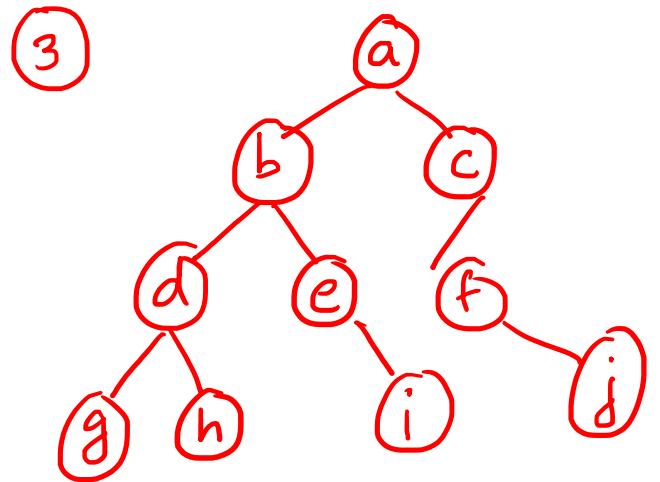
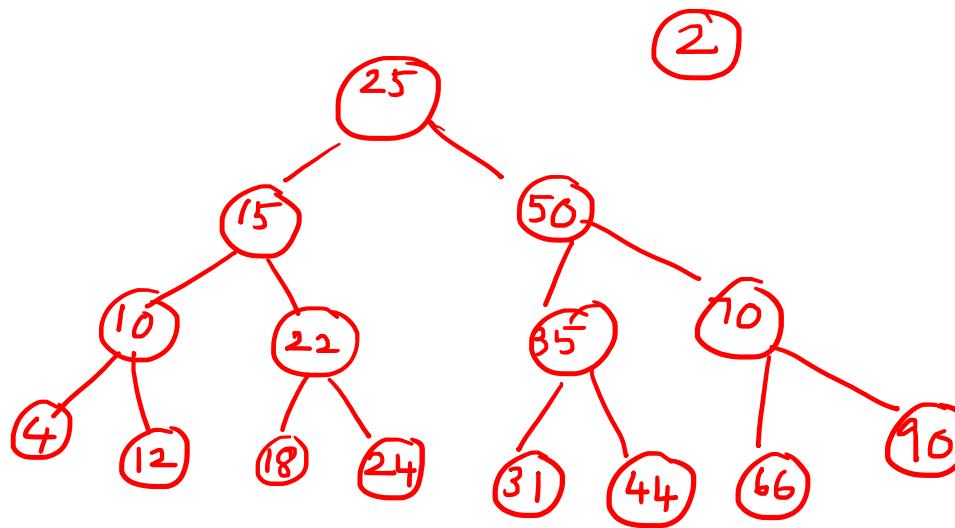
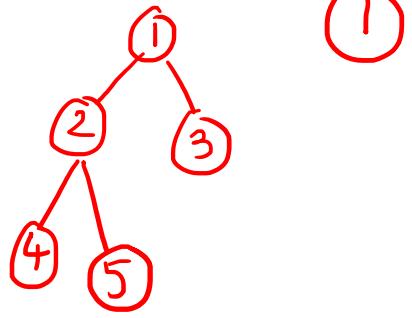
Preorder (PLR) ^{right} $\Rightarrow 123$
 Inorder (LPR) $\Rightarrow 213$
 Postorder (LRP) $\Rightarrow 231$



Preorder (PLR)
 $\rightarrow A B D G H C E I F$
Inorder (LPR)
 $G D H B A G I \underline{C} F$
Postorder (LRP)
 $G H D B I \underline{E F} C A$

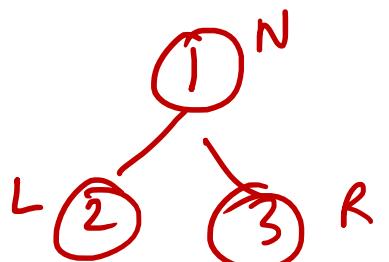


Preorder (PLR)
 $5 6 8 3 4 7 9$
Inorder (LPR)
 $8 6 4 3 7 5 9$
Postorder (LRP)
 $8 4 7 3 6 9 5$



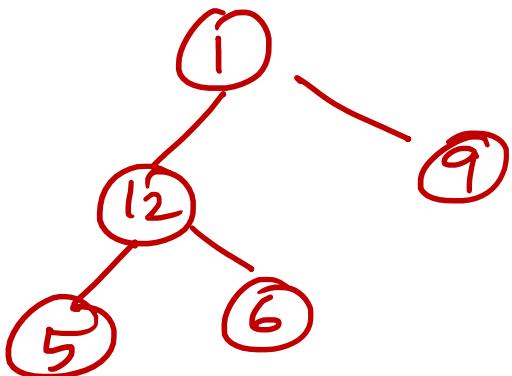
| Searching:

- » Searching an item in the tree can be done while traversing the tree in inorder, preorder or postorder traversals.
- » While visiting each node during traversal, instead of printing the node info, it is checked with the item to be searched.
- » If item is found, search is successful.

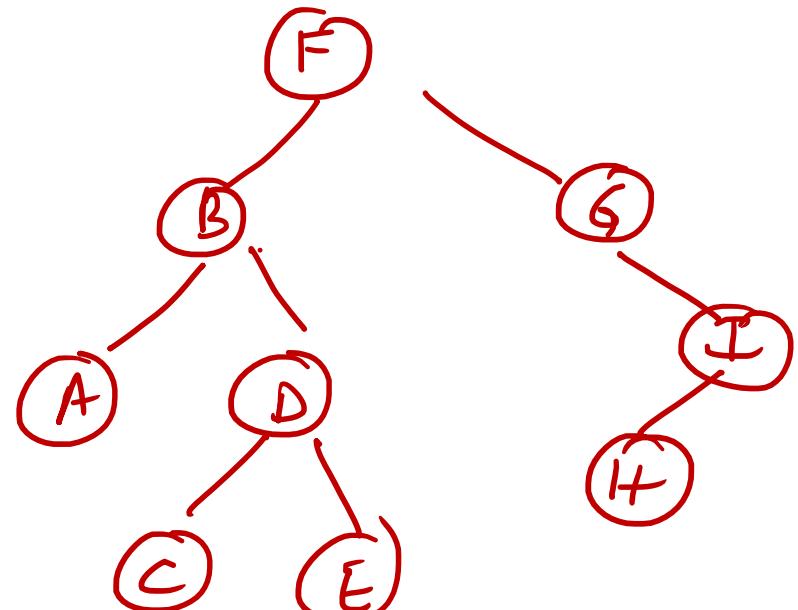


I 1, 2, 3
II 2, 1, 3
III 2, 3, 1

R|LR LNR LRN

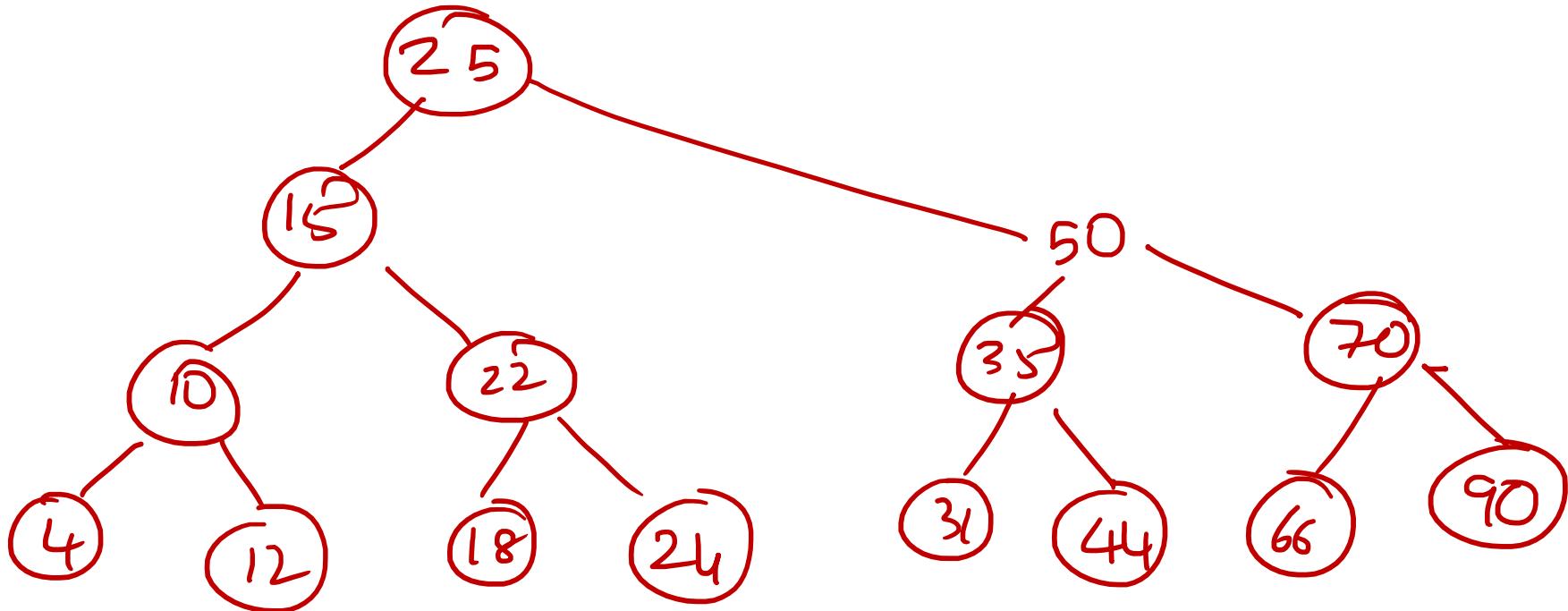


I 1, 12, 5, 6, 9
II 5, 12, 6, 1, 9
III 5, 6, 12, 9, 1



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.



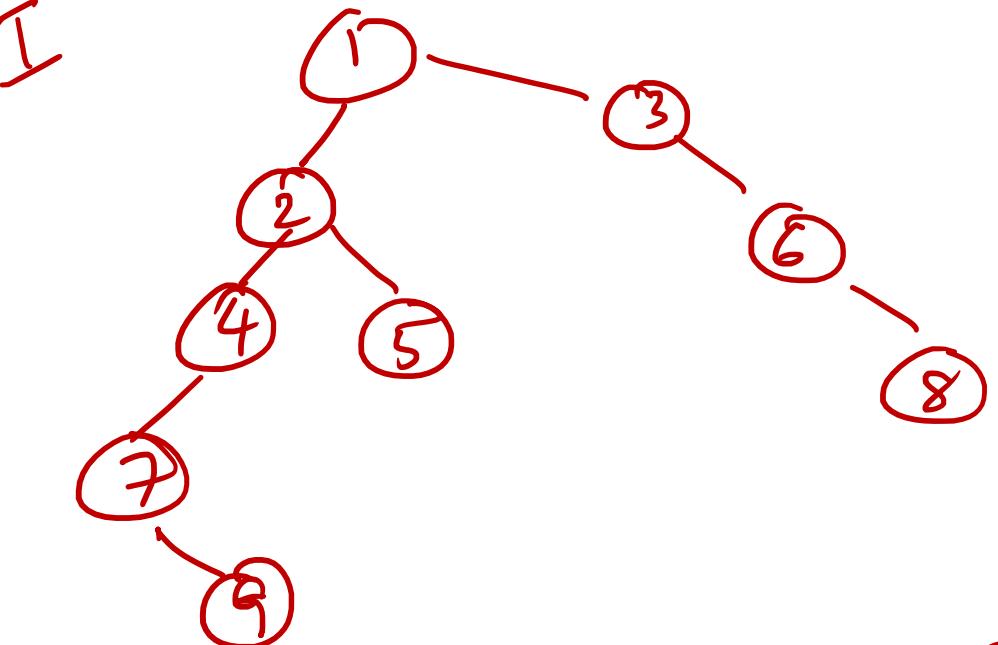
I 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90.

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

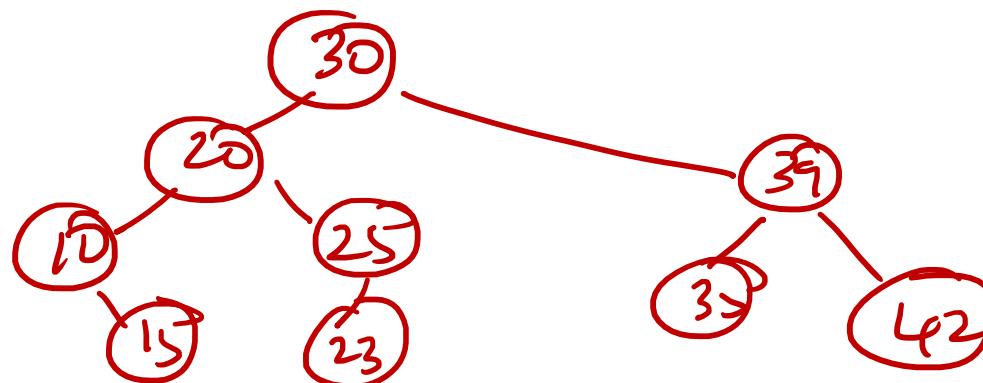
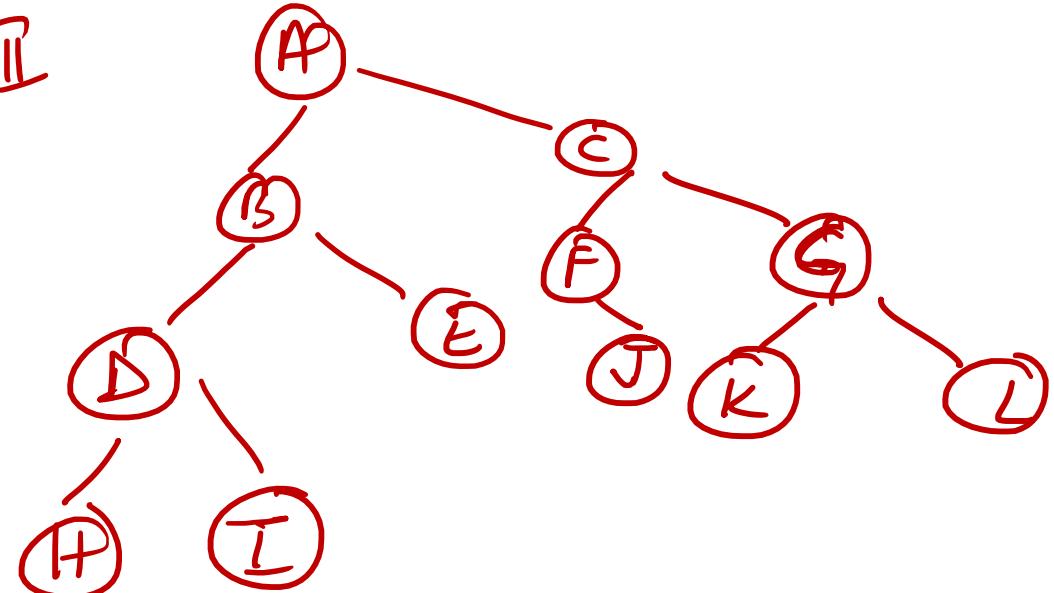
III 4, 12, 10, 18, 12, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

More Examples.

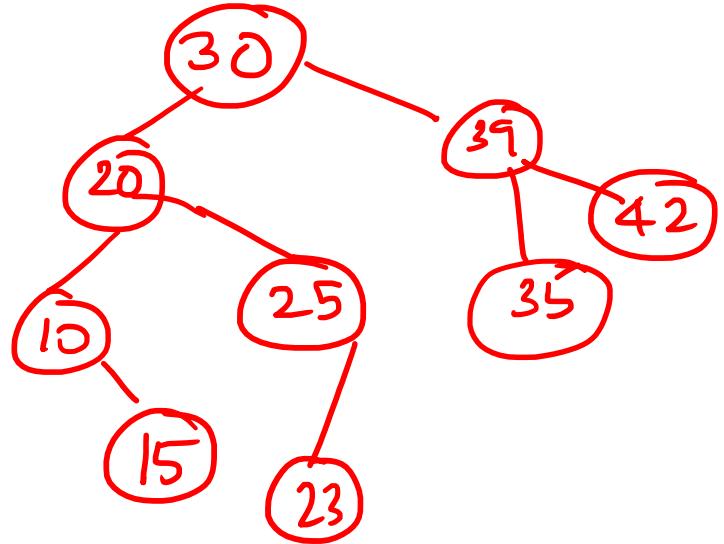
I



II



NLR .

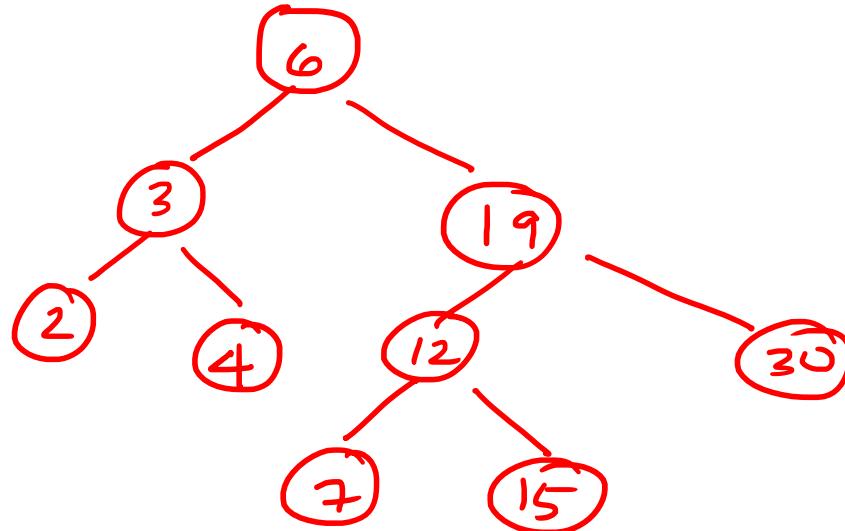


I) 30, 20, 10, 15, 25, 23, 39, 35, 42

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 CONSTRUCTION.

① →

Inorder and Preorder

② ⇒

Inorder and Postorder

③ ⇒

Preorder and Postorder

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

Preorder :- 1 2 4 8 9 10 11 5 3 6 7 (Parent left Right)
 Inorder :- 8 4 10 9 11 2 5 1 6 3 7 (left Parent Right)

- Construct Binary tree from Postorder & Inorder
- ① Find root (postorder) }
② Scan from R → L
③ Find left & Right subtrees
from 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

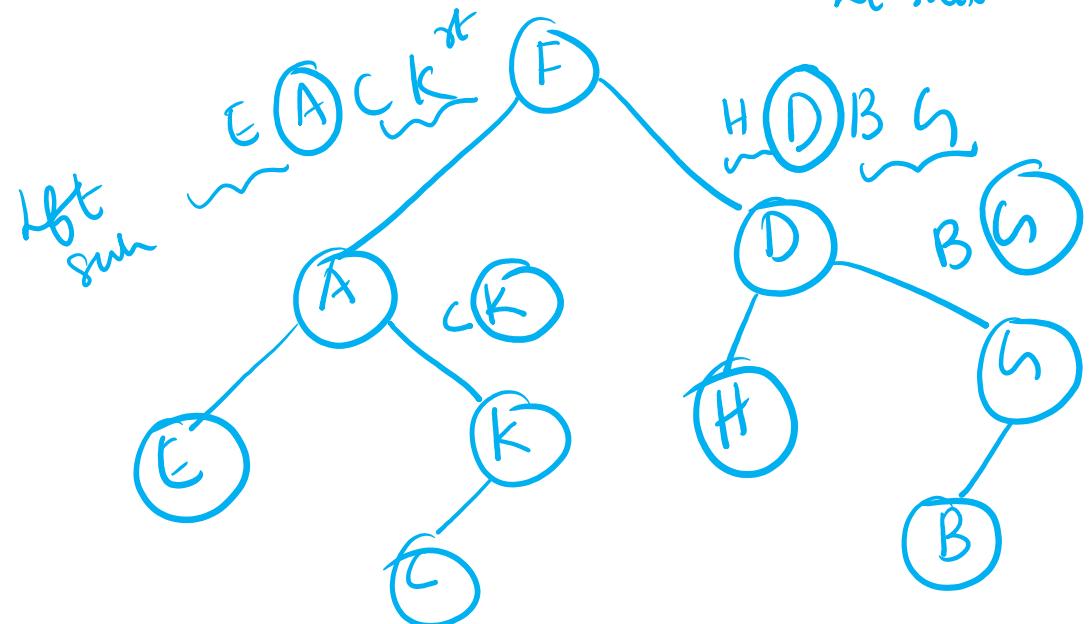
Postorder:- E C K A H B G D F

1. Binary Tree construction from Preorder and Inorder – Example 2.

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

Inorder :- E A C K F H D B G
(LPR)

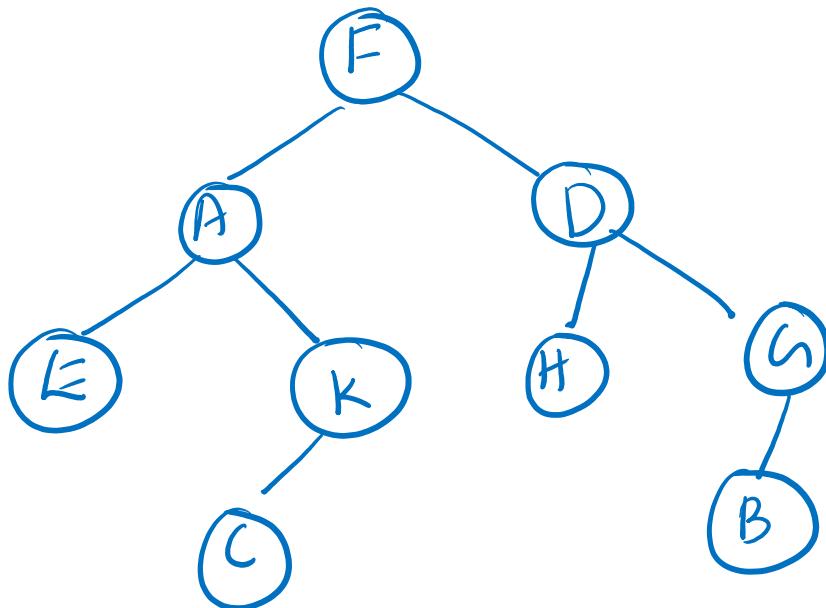
ht sub RT sub



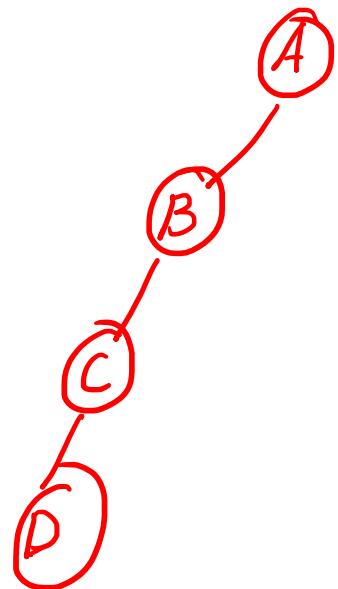
2. Binary Tree construction from Postorder and Inorder – Example 2.

POST \rightarrow E C K A H B G D F

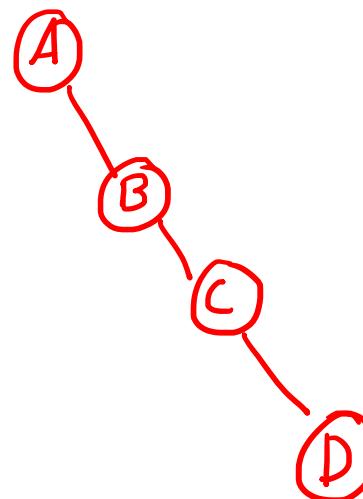
IN \rightarrow E A C K F H D B G



Construction of binary tree from preorder and postorder .



Preorder $\Rightarrow ABCD$
Post $\Rightarrow DCBA$



Preorder $\Rightarrow ABCD$
Postorder $\Rightarrow DCBA$

Construct a full BT from the given preorder and postorder traversals

Preorder \Rightarrow F B A D C E G I H (PLR)

Postorder \Rightarrow A C E D B I H G F (LRP)

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*.
- » Here, 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**.

Construct binary expression tree from the given infix expression

- ① $A + B * C$ ② $(A + B) * C$

Expression tree construction from the given infix expression

a × (b + c) + d

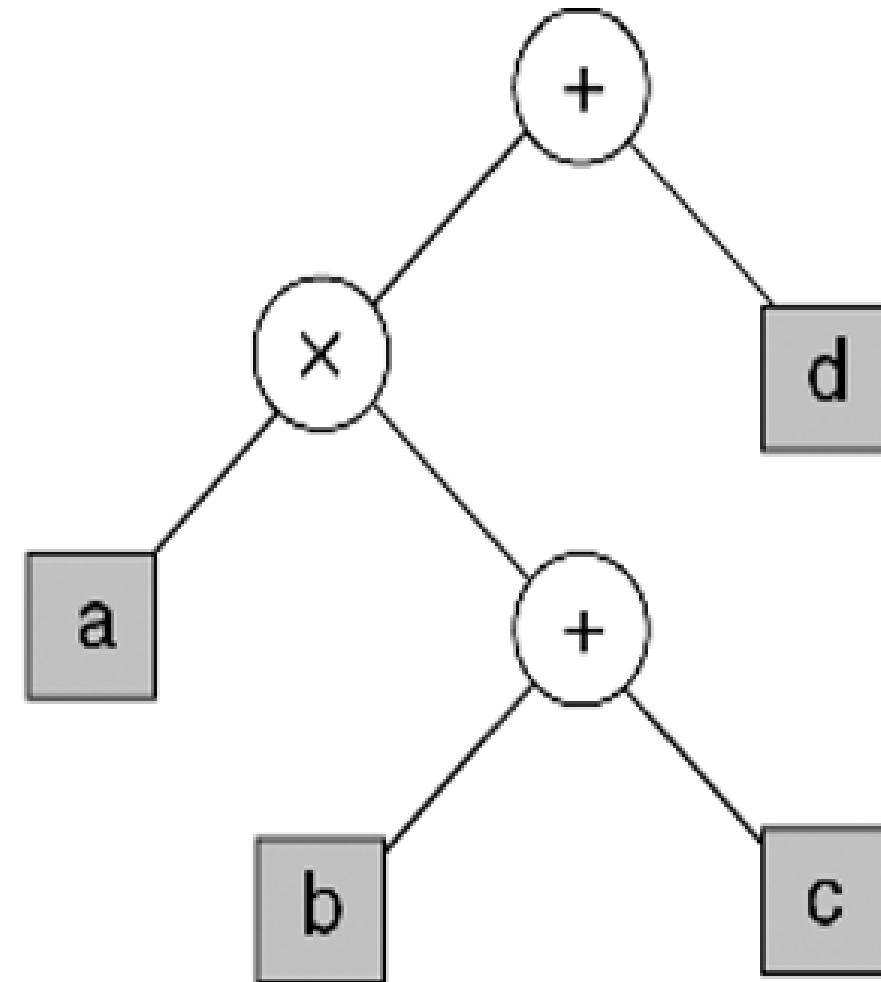


FIGURE 6-15 Infix Expression and Its Expression Tree

$$\textcircled{3} \quad a * (b + c) + d$$

$$\textcircled{4} \quad a + b * c - d / e + f$$

⑤

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

Binary Expr'n tree from Postfix

Postfix \Rightarrow Expr'n Tree \Rightarrow Stack

\Rightarrow Scan $L \rightarrow R$

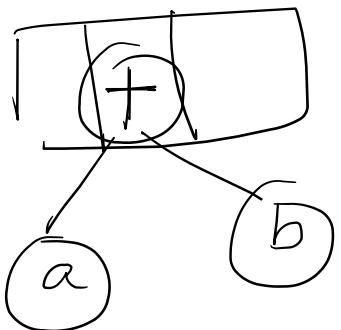
\Rightarrow operand \rightarrow push

\Rightarrow operator \rightarrow pop 2 element
and construct tree

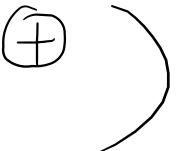
\Rightarrow ab+
 $\xrightarrow{L \rightarrow R}$

2 times
eg: 

(+) \Rightarrow pop twice \Rightarrow



then push the pointer
into the stack

(eg: - pointer of


Construct binary expression tree from postfix expression

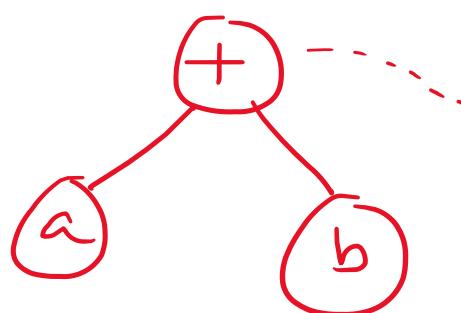
ABC - D * EF + Λ +



Binary expⁿ tree from Preorder

$\Rightarrow +ab$

$\boxed{a} \ b \Rightarrow + \Rightarrow \text{tree}$



Push the
pointer for +
into stack

\rightarrow Stack DS

$\rightarrow R \rightarrow L$

\rightarrow operands \rightarrow push

\rightarrow operators \rightarrow pop 2 times
& construct the tree

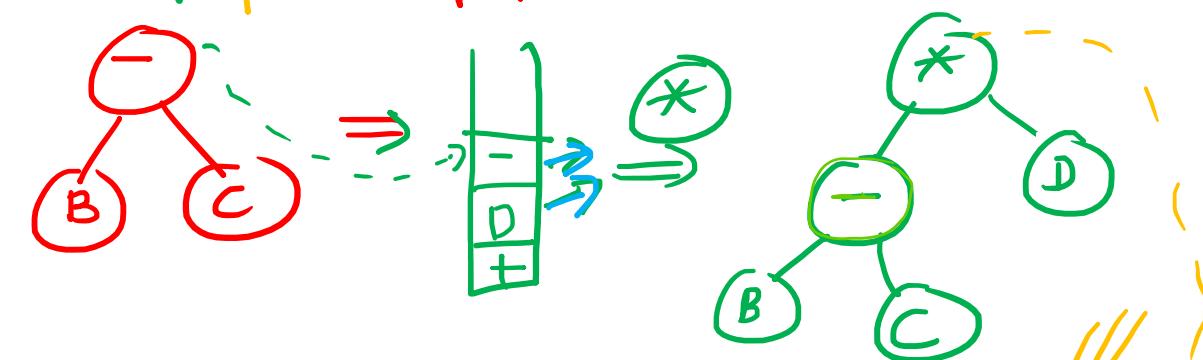
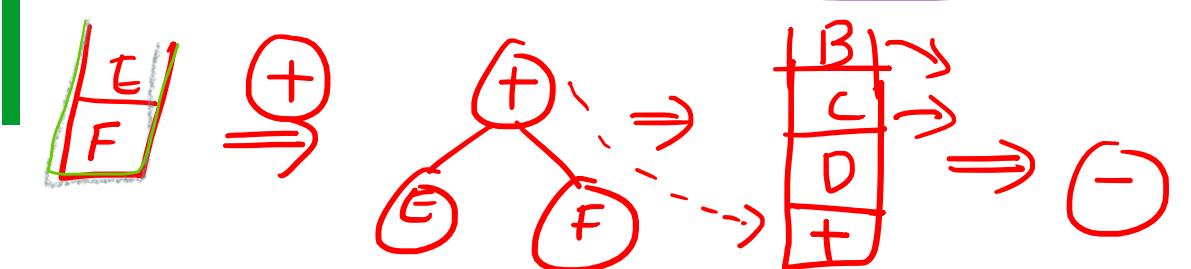
Preorder \Rightarrow

$+ A \wedge * - B C D + E F$

~~PLR~~

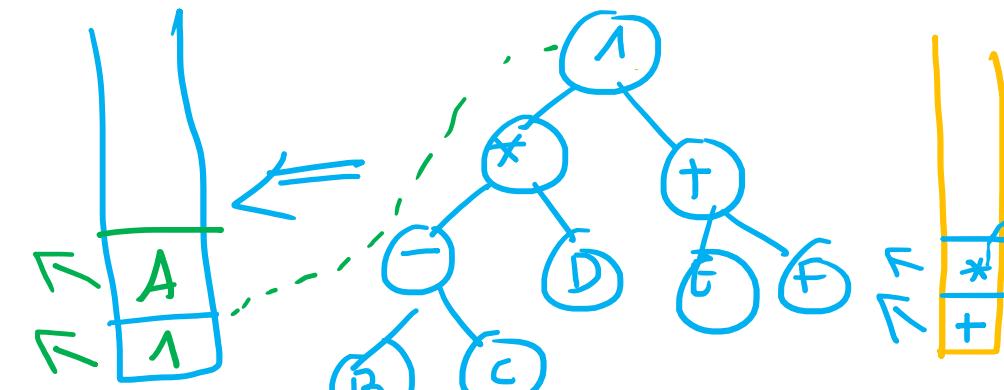
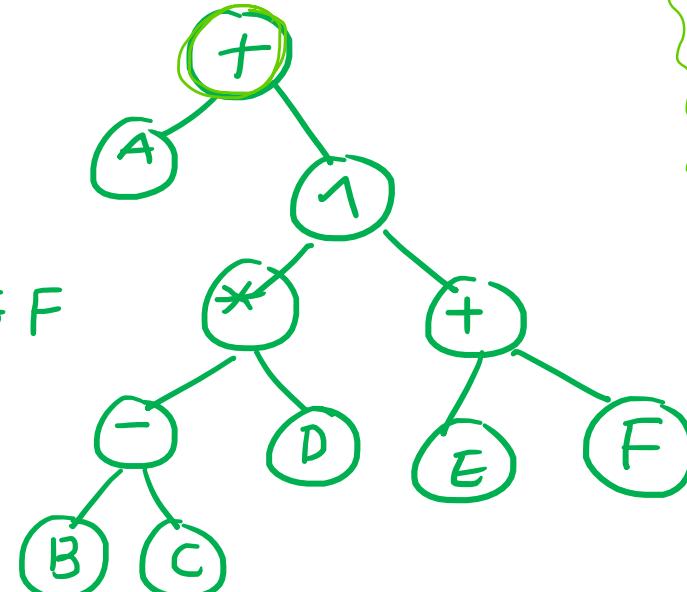
Binary exp'n tree from Preorder $\Rightarrow + A \wedge * - B C D + E F$

$R \rightarrow L$
Stack



Preorder (PLR)

$+ A \wedge * - B C D + E F$



How $\text{pre} \Rightarrow /* + ab - cd + ef$

$\Rightarrow \text{post} \Rightarrow ab + cd - * ef +)$

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.

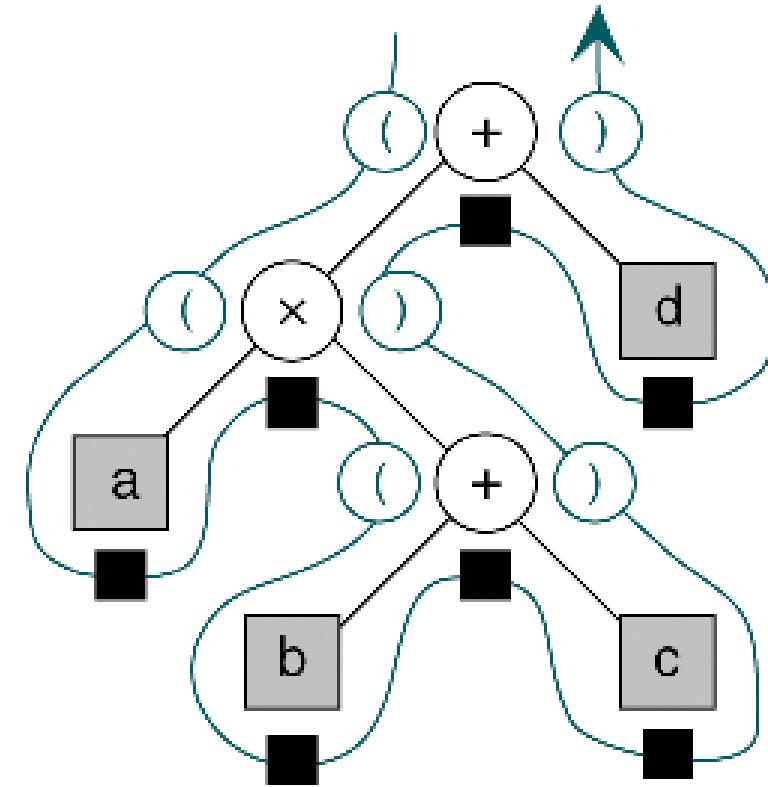
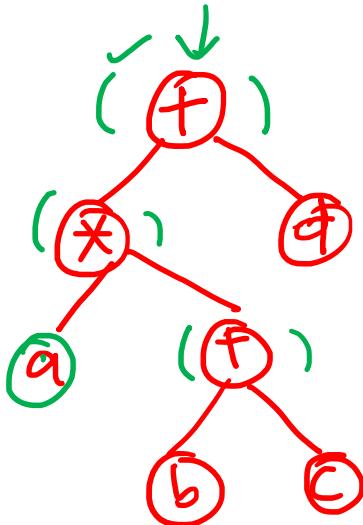
$$((a \times (b + c)) + d)$$


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
```

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)

 2 postfix (tree right subtree)

 3 print (tree token)

 2 end if

end postfix

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

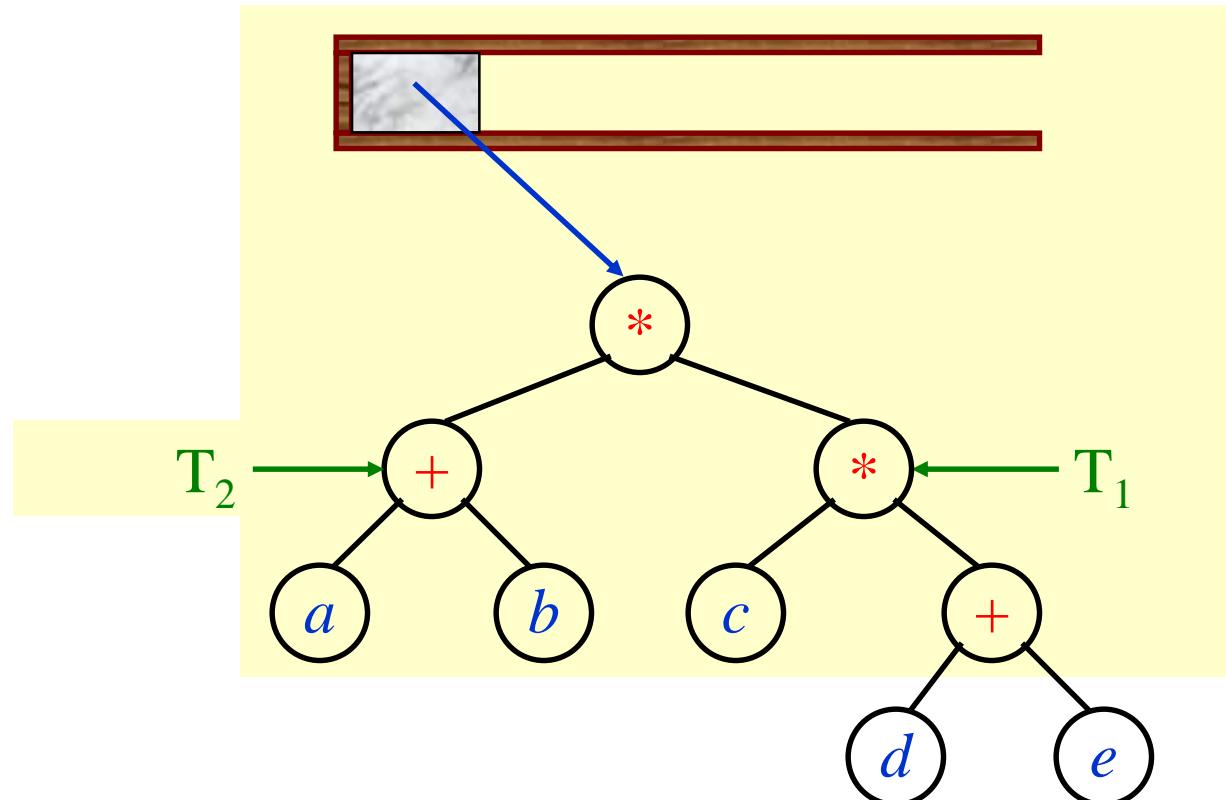
Post the prefix expression has been printed

```
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
```

END OF BINARY TREES

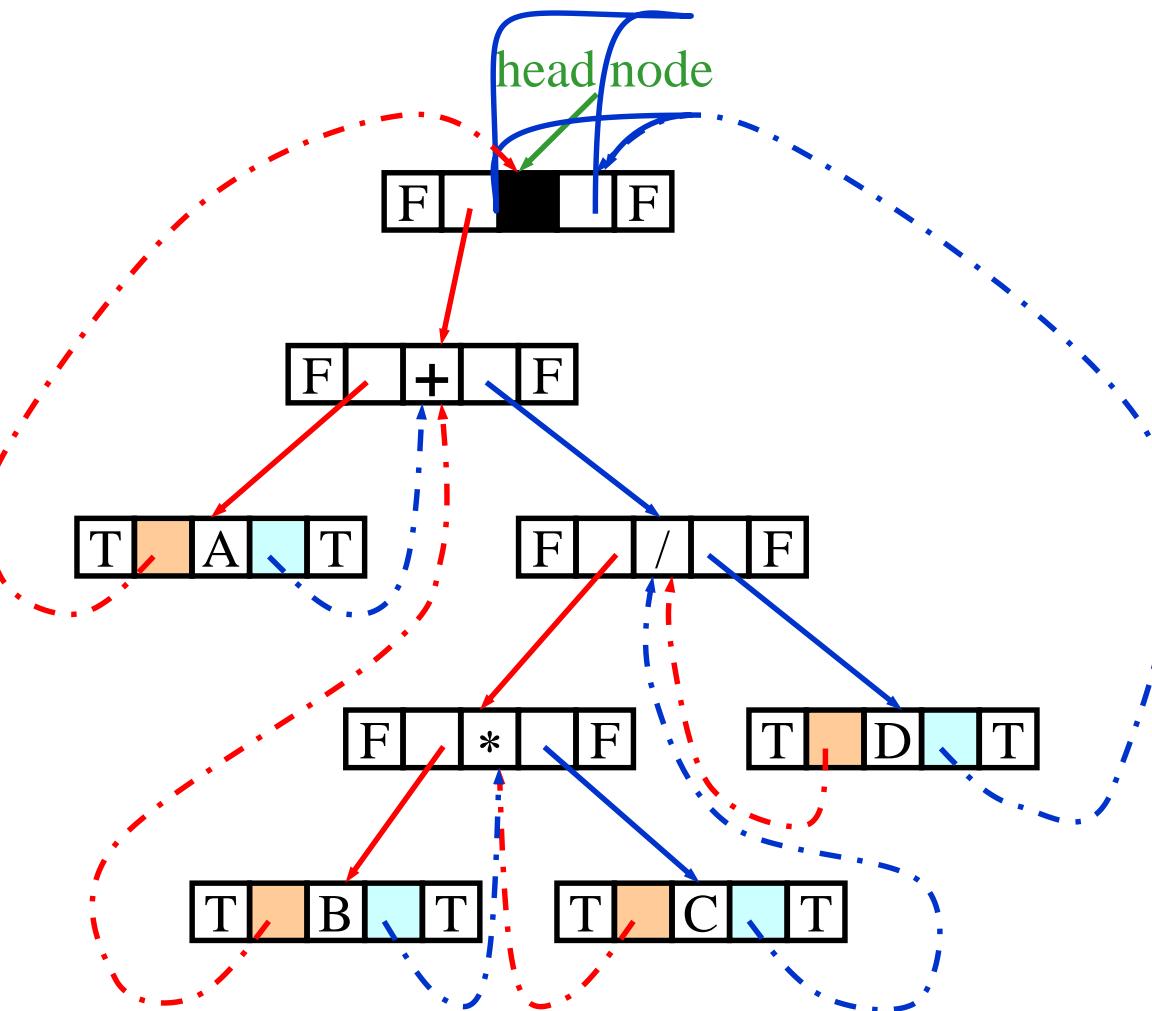
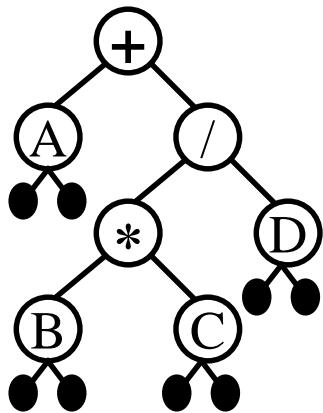
Expression Trees (syntax trees)

〔Example〕 $(a + b) * (c * (d + e)) = ab + cde + * *$



〔Example〕 Given the syntax tree of an expression (infix)

$$A + B * C / D$$



a × (b + c) + d

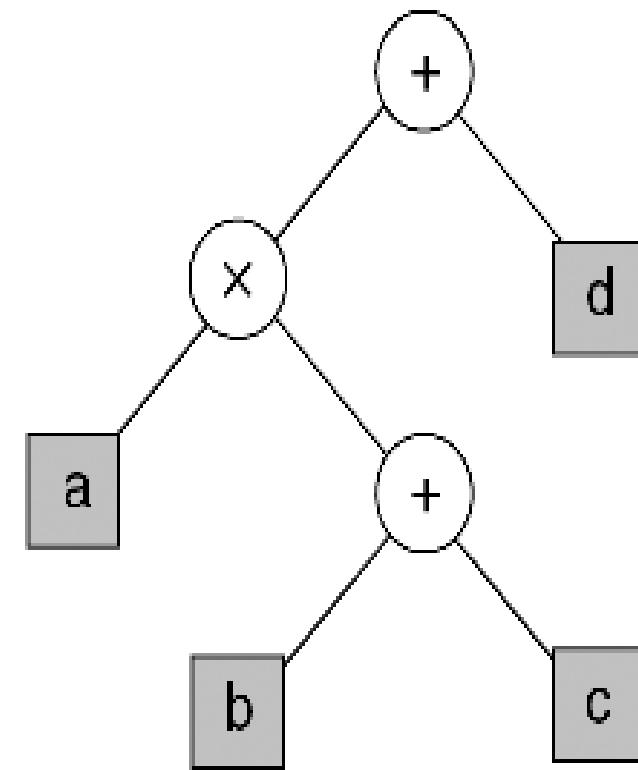


FIGURE 6-15 Infix Expression and Its Expression Tree

| GENERAL TREES.

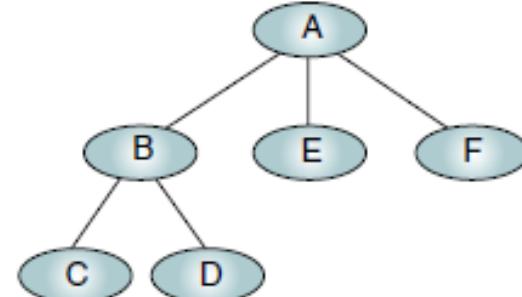
- » A *general tree* is a tree in which each node can have an **unlimited outdegree**.
- » Each node may have **as many children** as is necessary to satisfy its requirements.

INSERTION INTO GENERAL TREES.

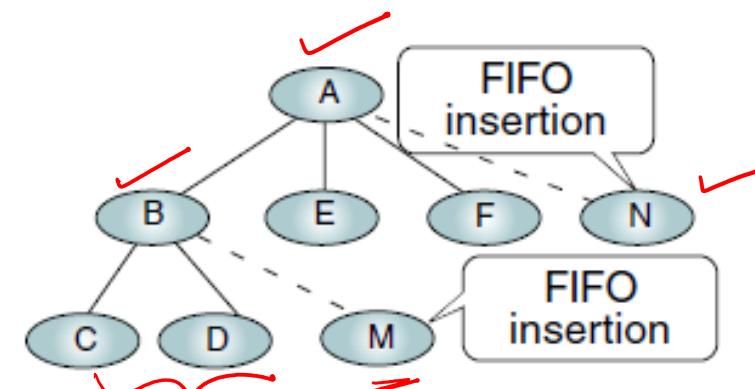
- » To insert a node into a general tree, ***the user must supply the parent of the node.***
- » Given the parent, **three** different rules may be used:
 - (1) first in–first out (FIFO) insertion,
 - (2) last in–first out (LIFO) insertion, and
 - (3) key-sequenced insertion

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

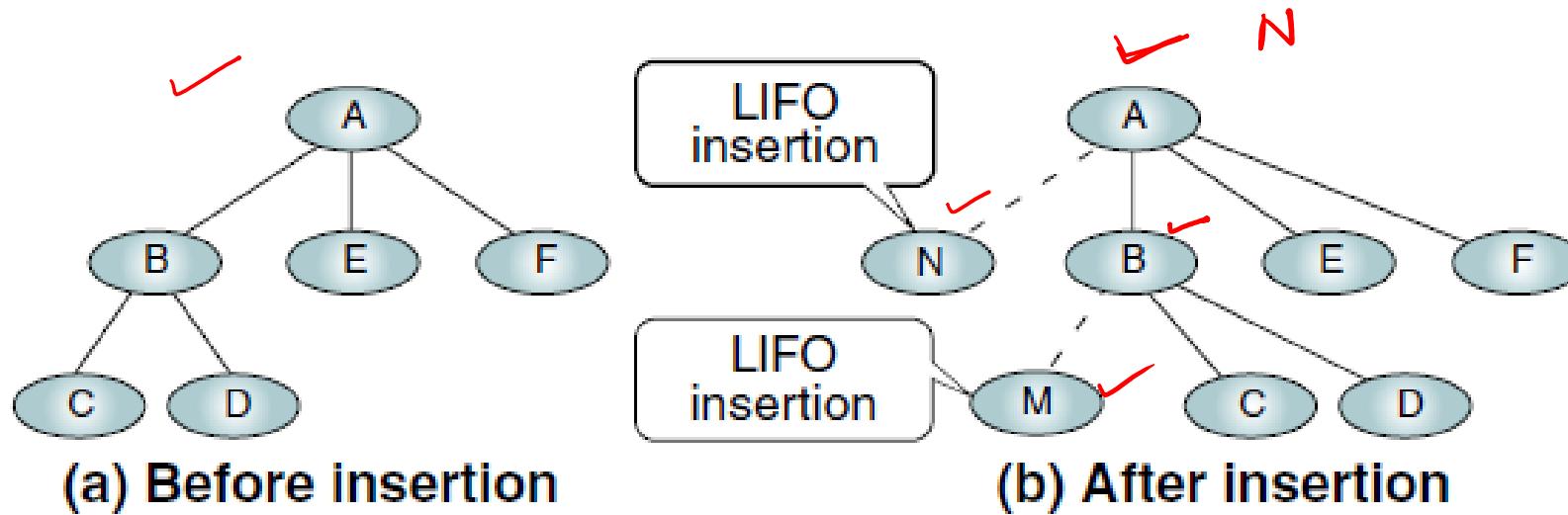


(b) After insertion

Stack

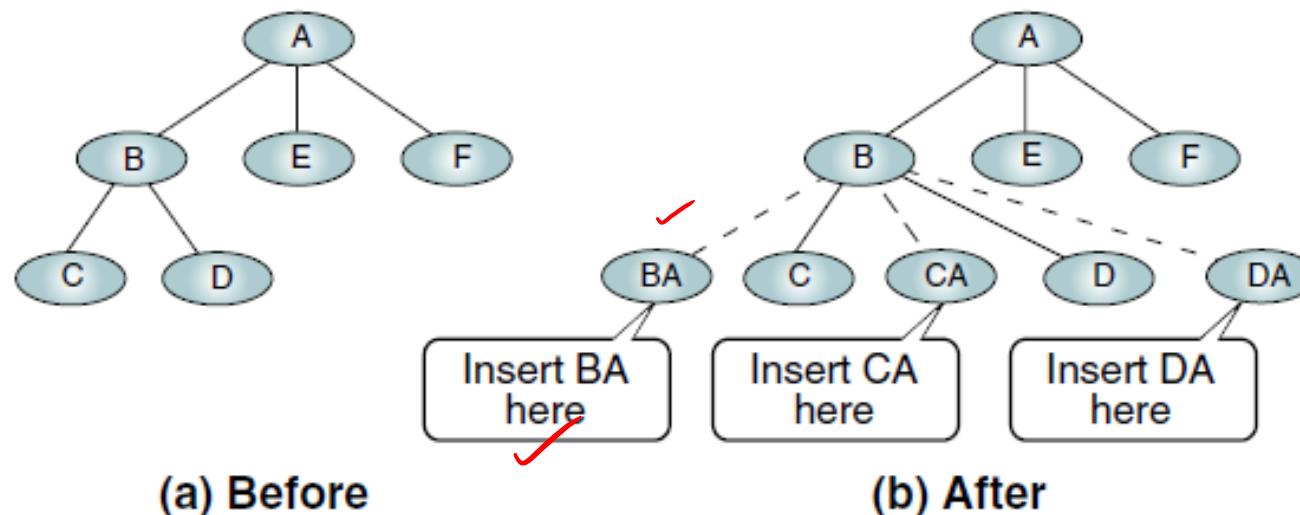
LIFO INSERTION.

- » 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,

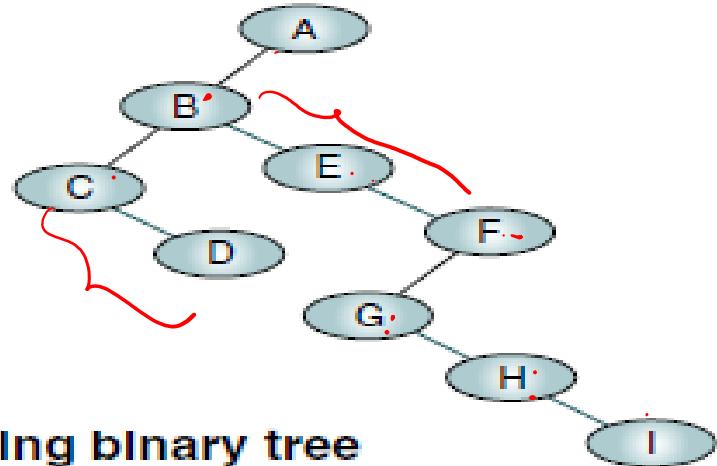
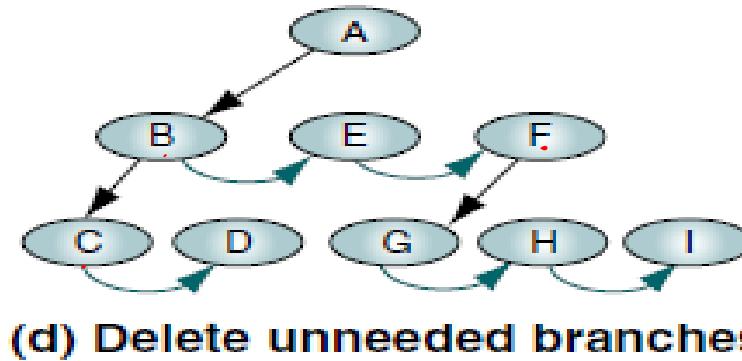
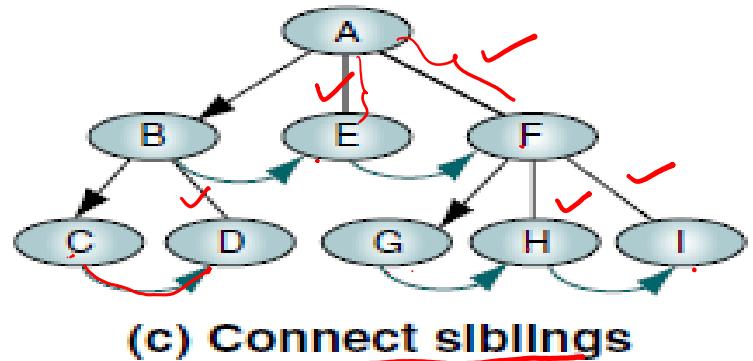
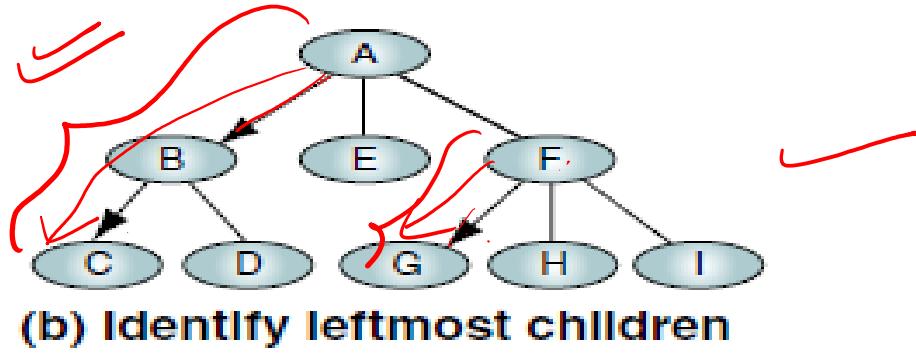
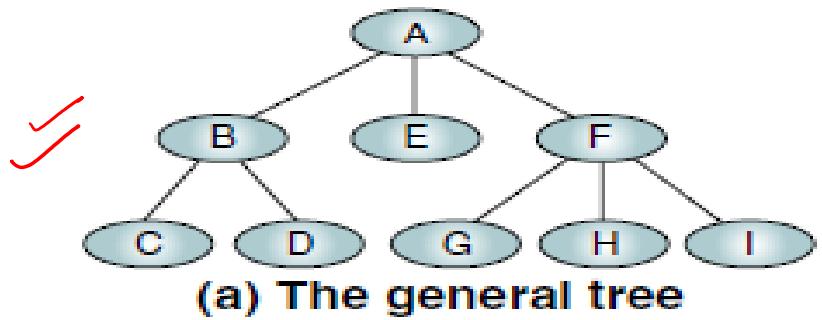
- » BUT **standard deletion rules are:**

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, using two relationships: parent to child and sibling to sibling. Using these two relationships, we can represent any general tree as a binary tree.



Converting General Trees to Binary Trees

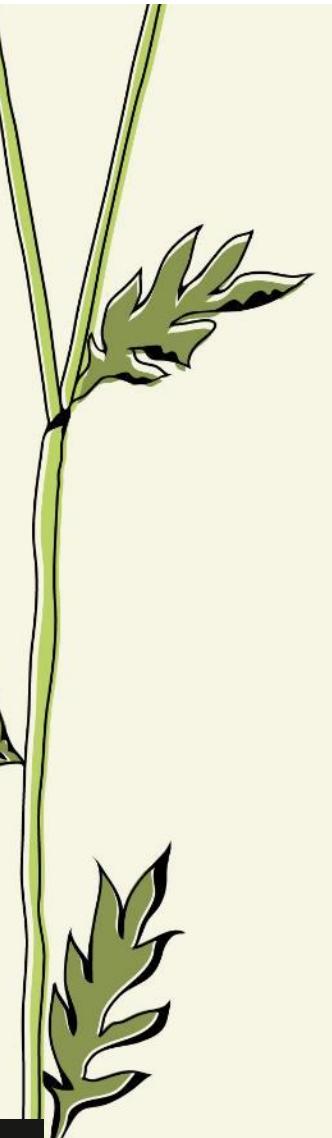
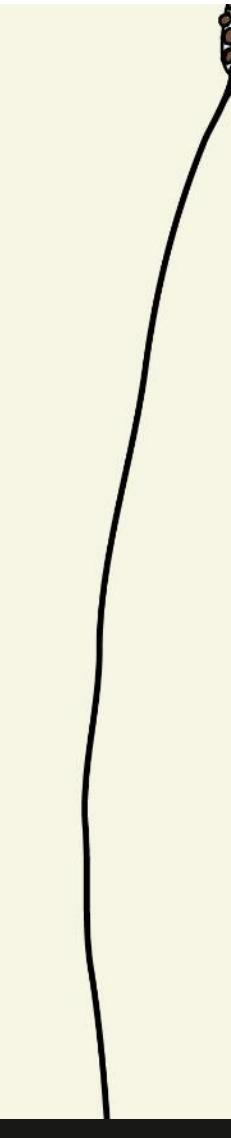
SUMMARY of GENERAL TREES.

- » 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.





The end of part 1 - concept of Trees.



Binary "Search" Trees.



The ordered nodes.



Objectives.



Create and implement binary search trees

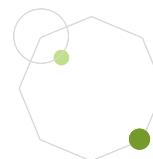


Understand the operation of the binary search tree ADT



Write application programs using the binary search tree ADT

Design and implement a list using a BST



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**.



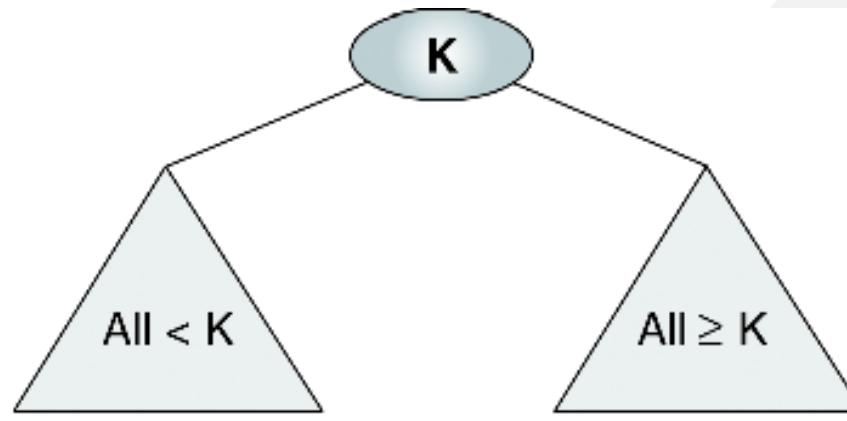


FIGURE 7-1 Binary Search Tree

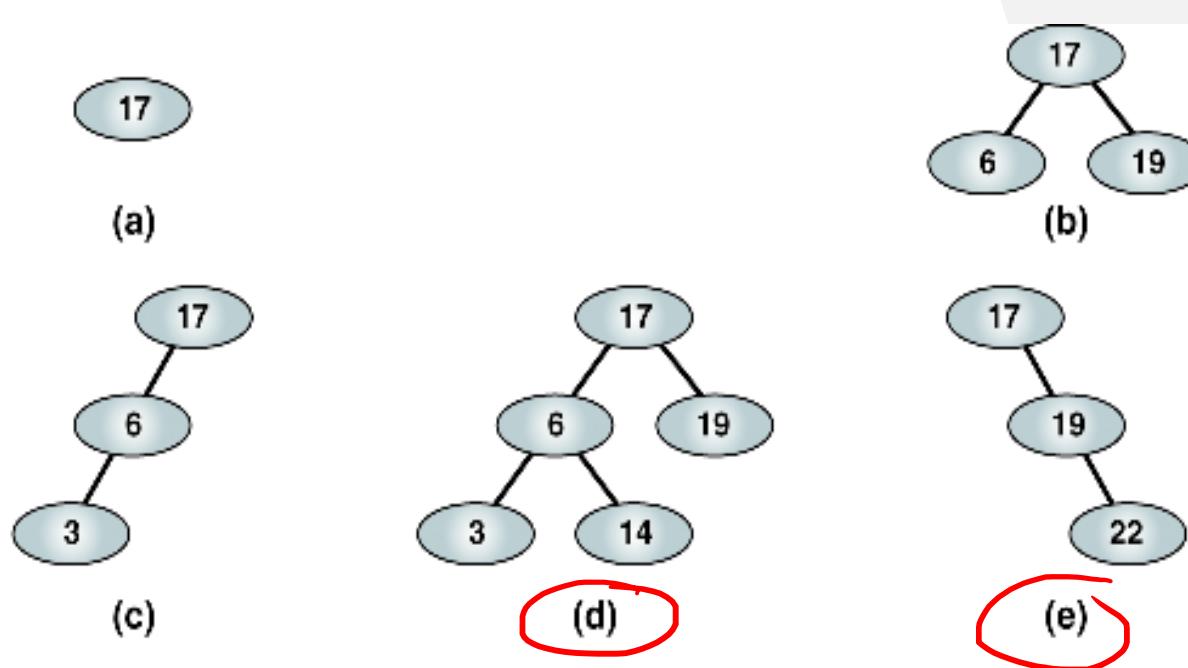
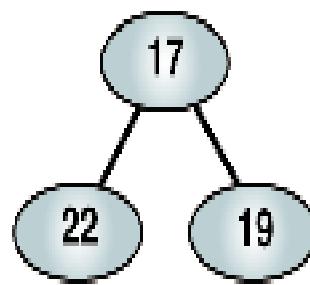
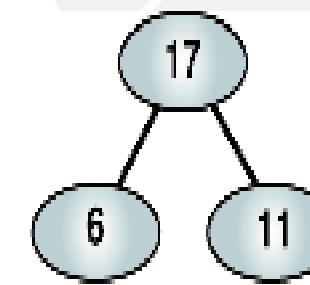


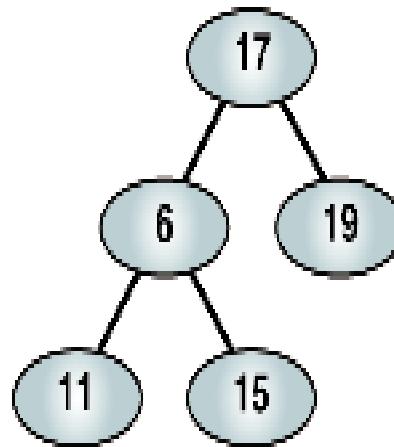
FIGURE 7-2 Valid Binary Search Trees



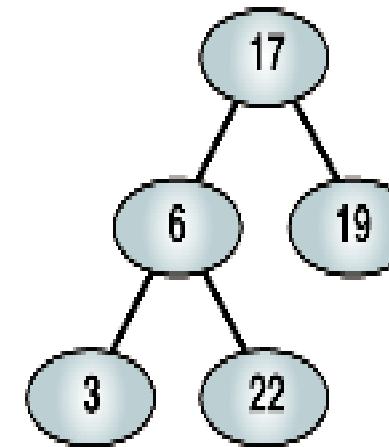
(a)



(b)

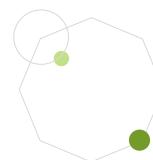


(c)



(d)

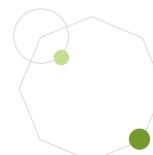
FIGURE 7-3 Invalid Binary Search Trees



BST OPERATIONS.

There are *four* basic BST operations:

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



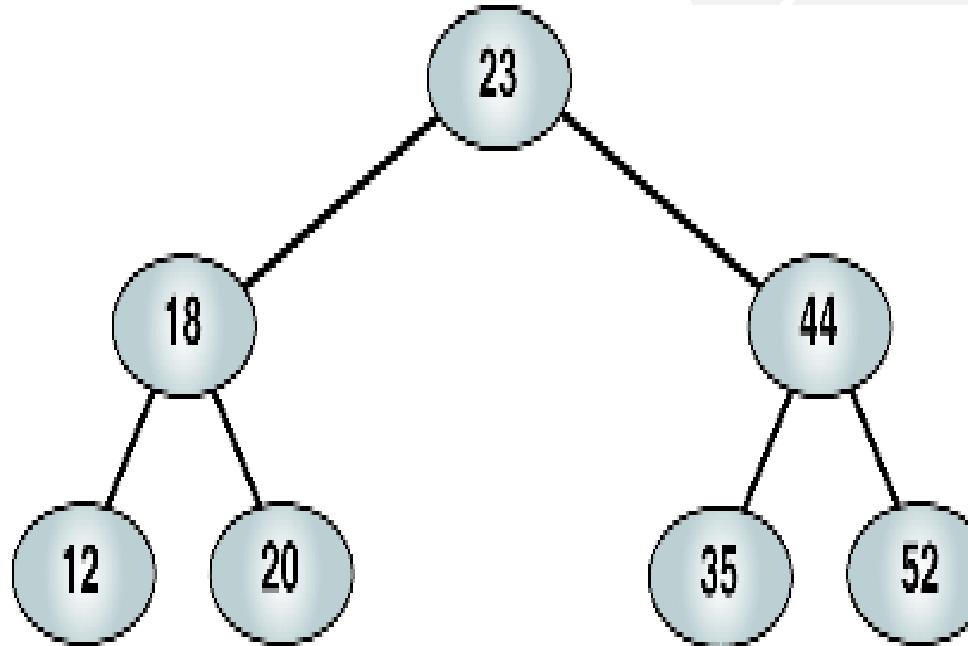


FIGURE 7-4 Example of a Binary Search Tree

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

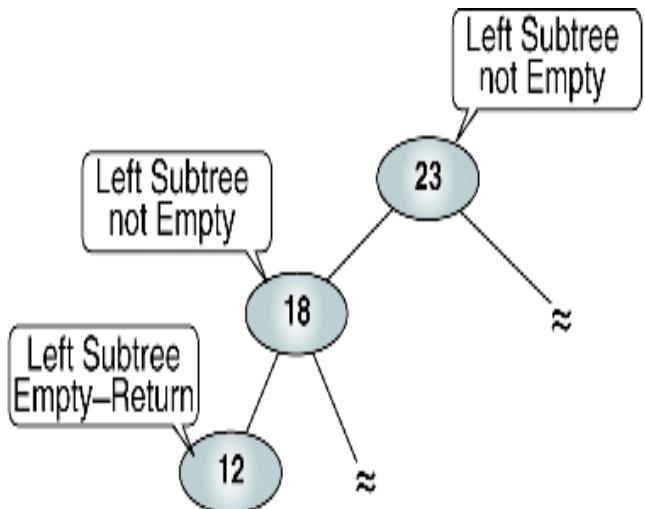
Return address of smallest node

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

2 end if

3 return findSmallestBST (left subtree)

end findSmallestBST

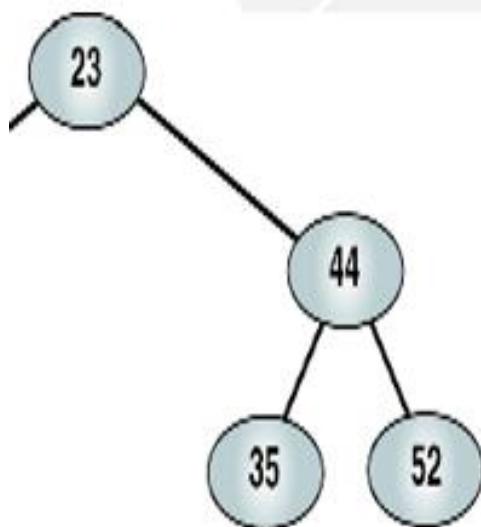


- Node with the smallest value (12) is the **far-left leaf node** in the tree.
- The find smallest node operation, therefore, simply follows the left branches until a leaf is reached..

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
----	----	----	----	----	----	----

~~20~~

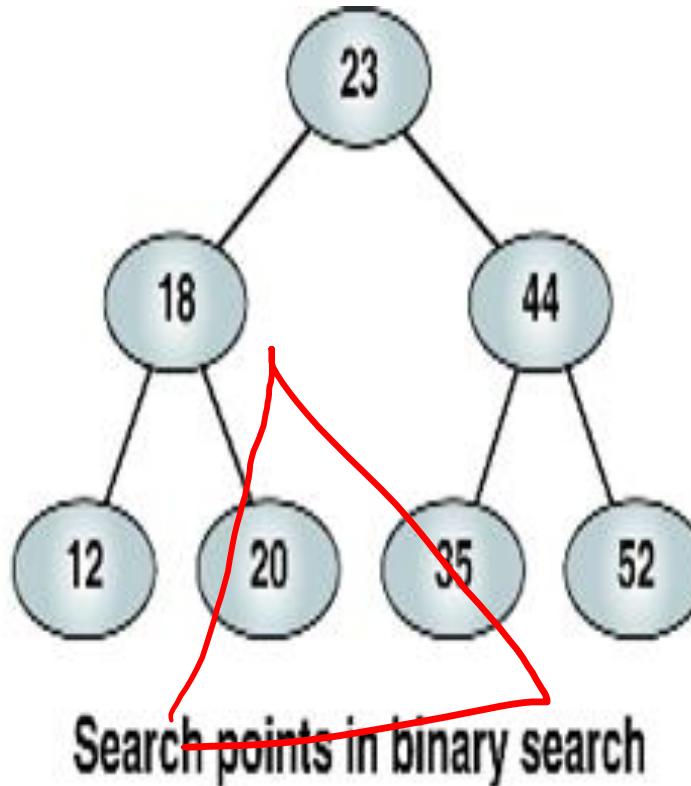
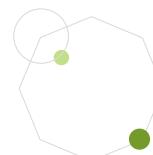
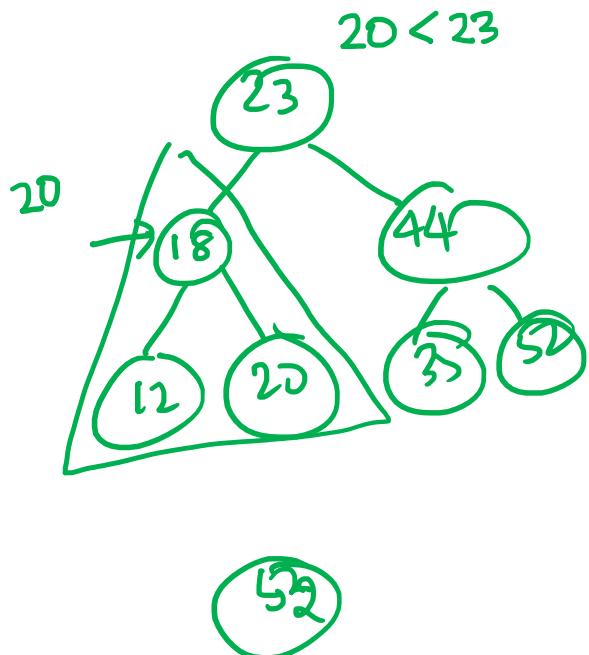


FIGURE 7-6 BST and the Binary Search



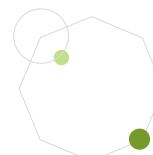
ALGORITHM 7-3 Search BST



```
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
```

Handwritten annotations on the right side of the algorithm:

- A green circle with the number 2D has an arrow pointing to the first condition in step 3.
- Handwritten text "2D < root" is written below the first condition.
- Handwritten text "2D > root" is written below the second condition.



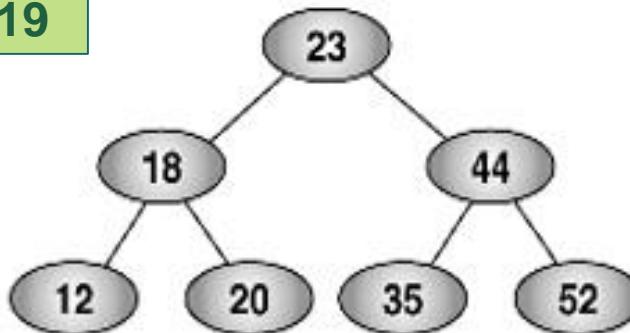
INSERTION OPERATIONS.



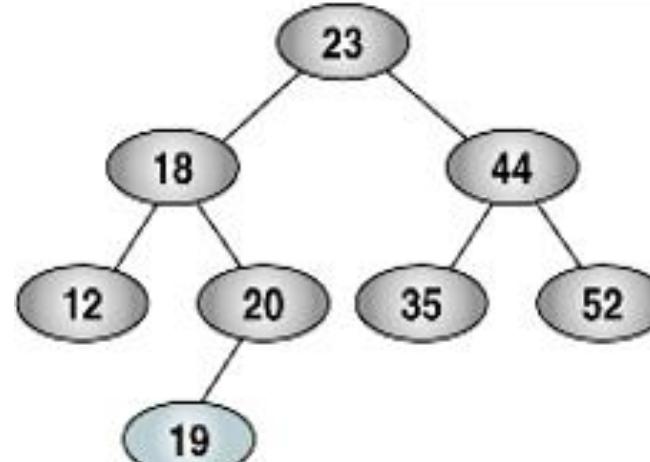
All BST insertions take place at a leaf or a leaflike node.



Inserting 19

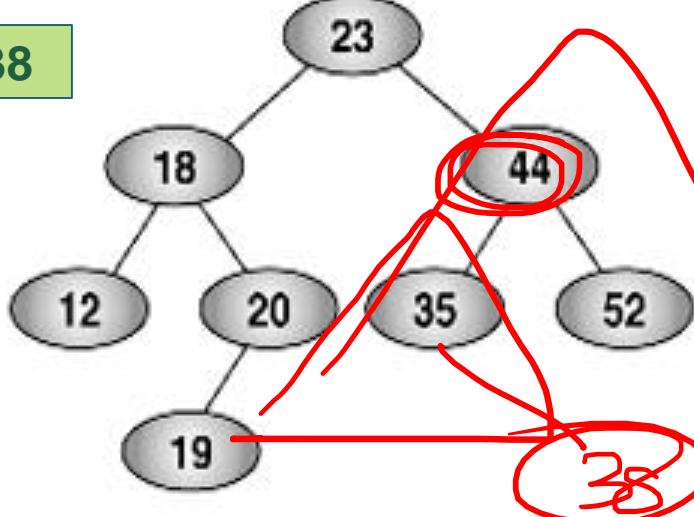


(a) Before inserting 19

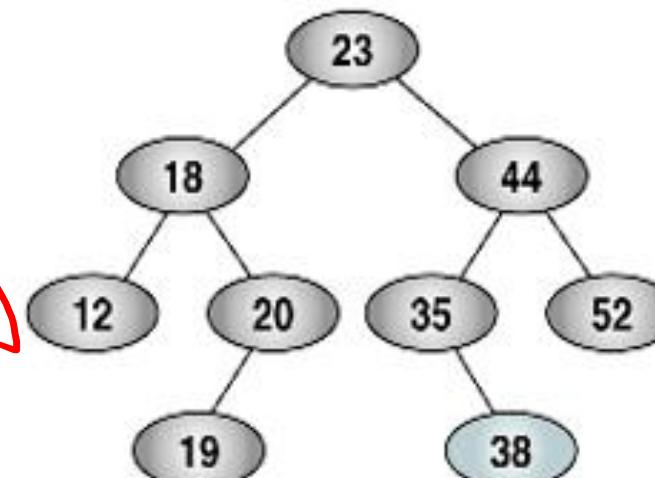


(b) After inserting 19

Inserting 38



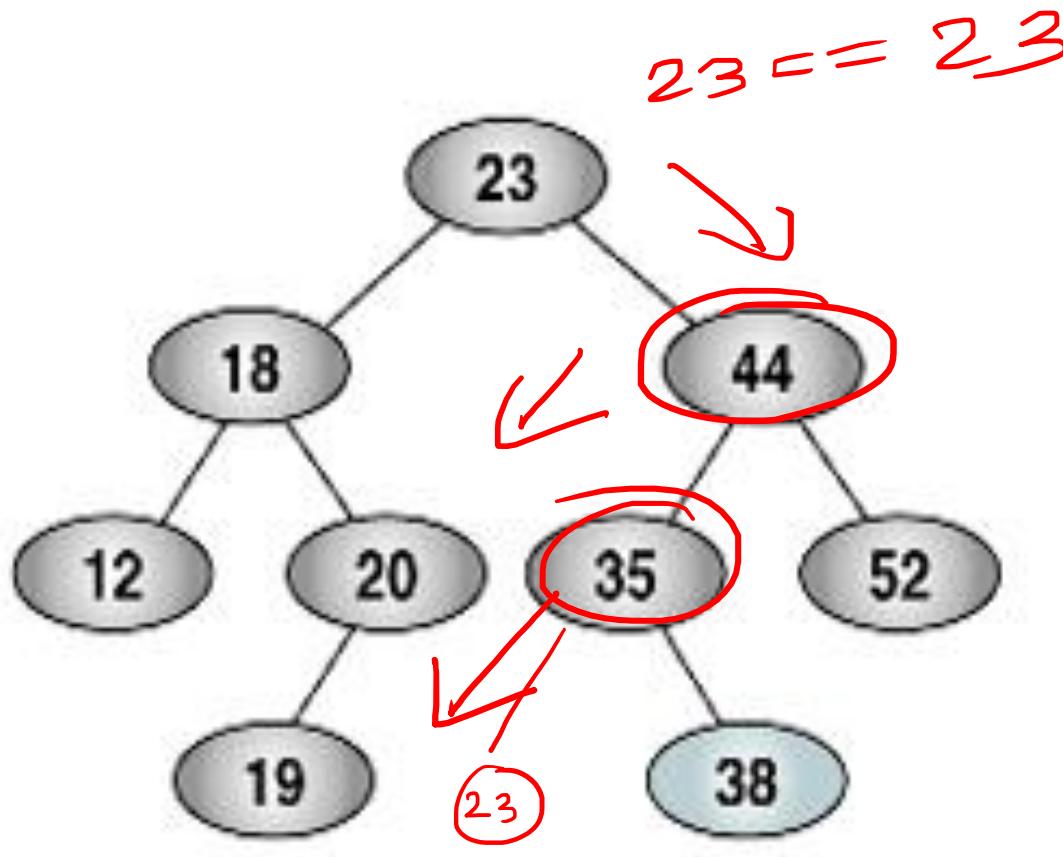
(c) Before inserting 38



(d) After inserting 38

FIGURE 7-8 BST Insertion

Inserting 23 ?



(d) After inserting 38

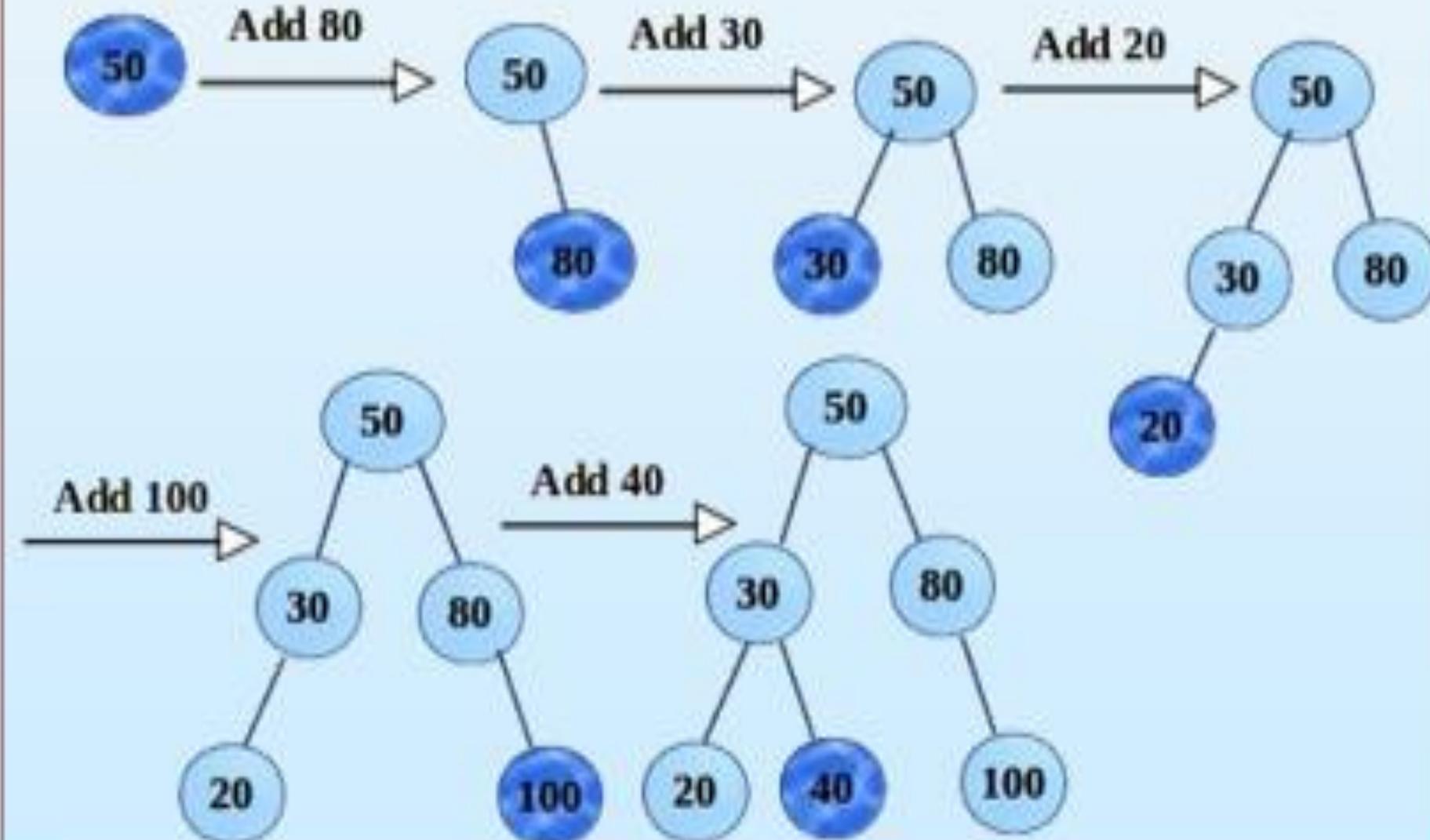
then insert 23

FIGURE 7-8 BST Insertion



50 ✓
80 ✓
30 ✓
20 ✓
100 ✓
40 ✓

example 2



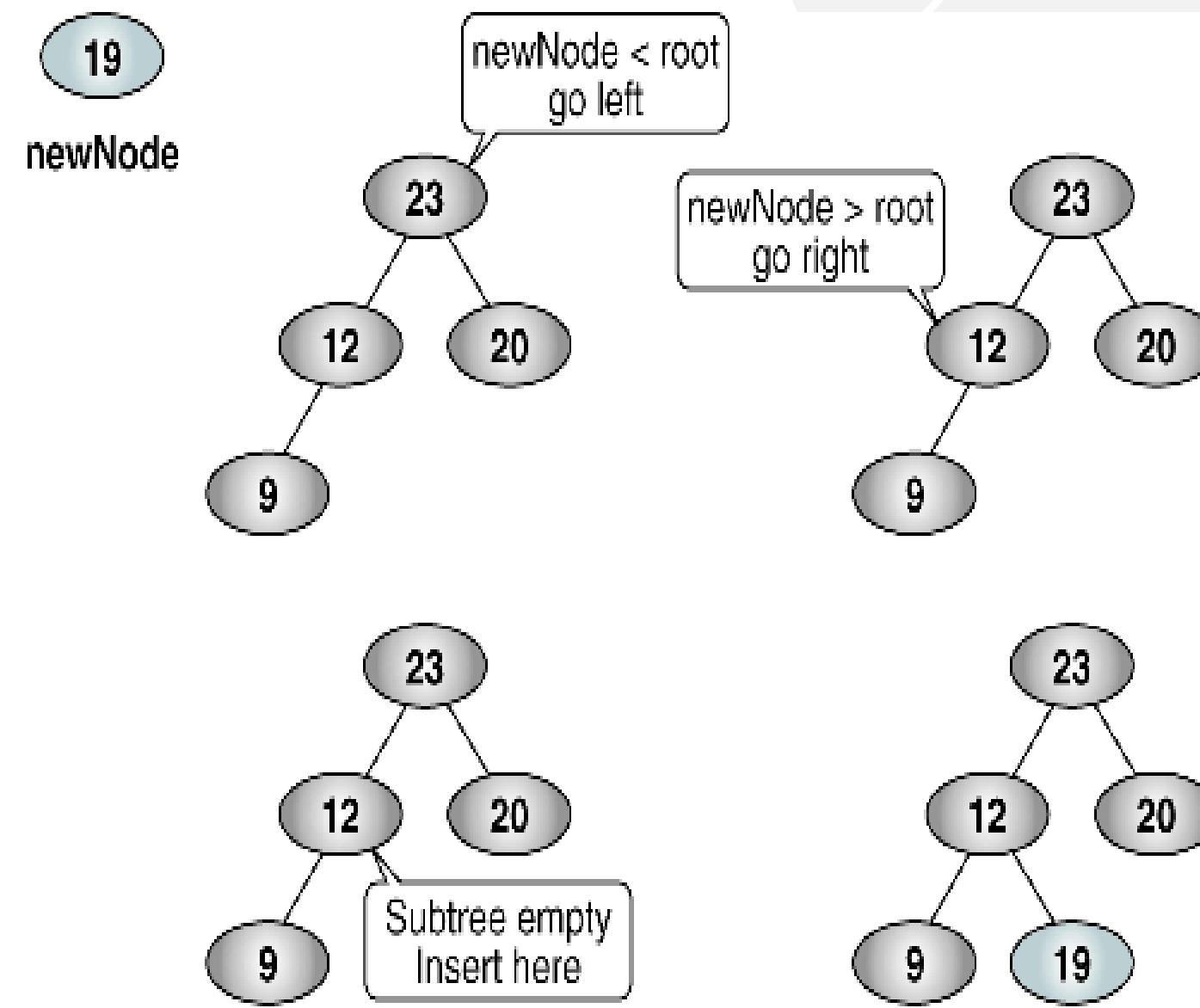


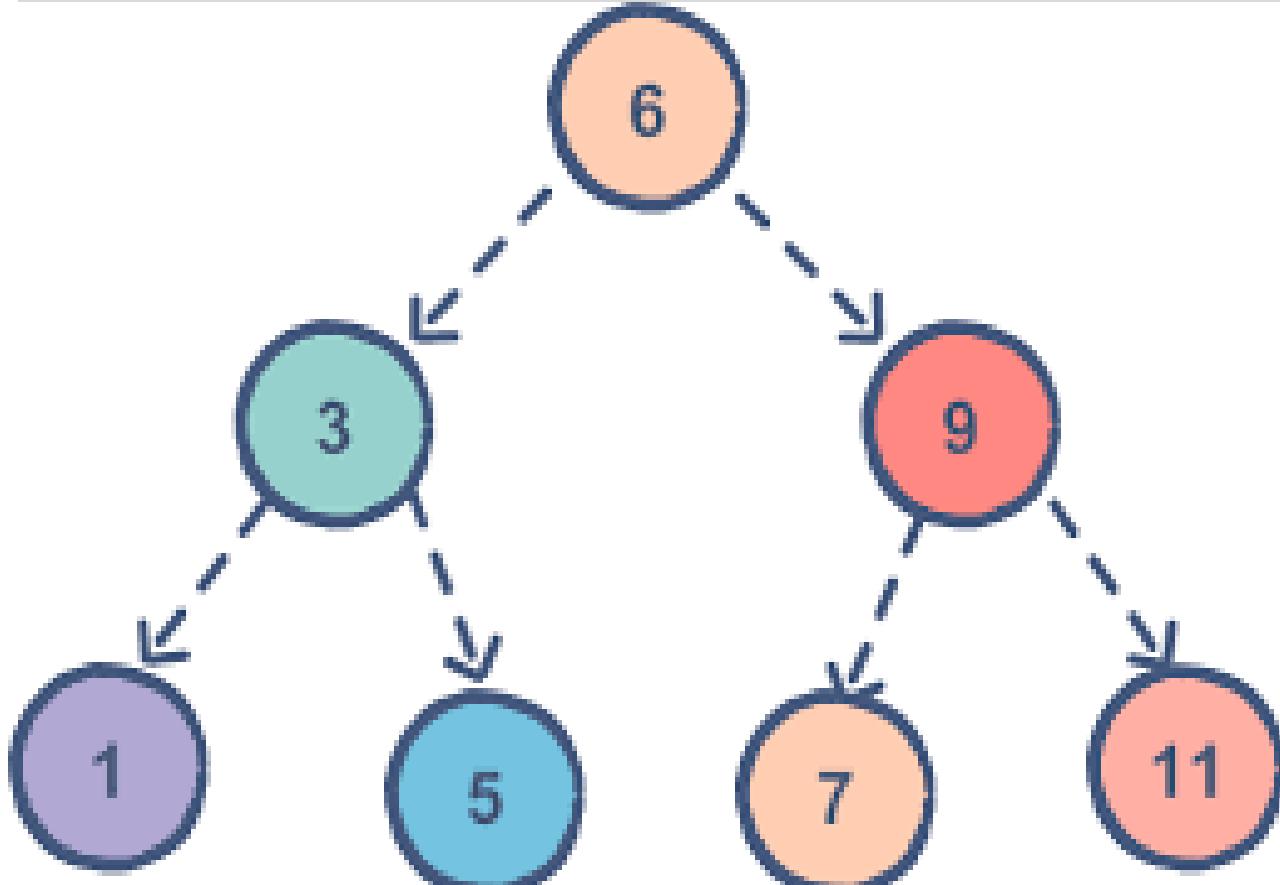
FIGURE 7-9 Trace of Recursive BST Insert

ALGORITHM 7-4 Add Node to BST

```
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
```

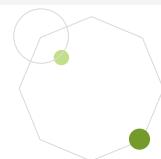
ALGORITHM 7-4 Add Node to BST (continued)

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



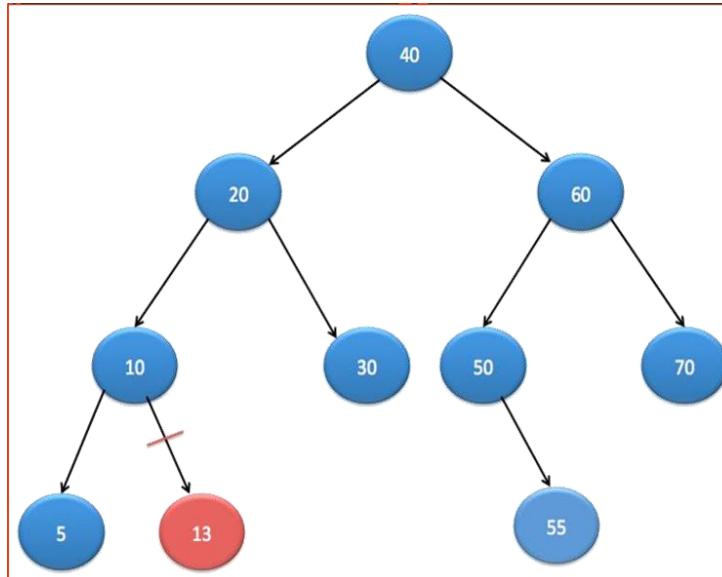
An example of a binary search tree

Break.

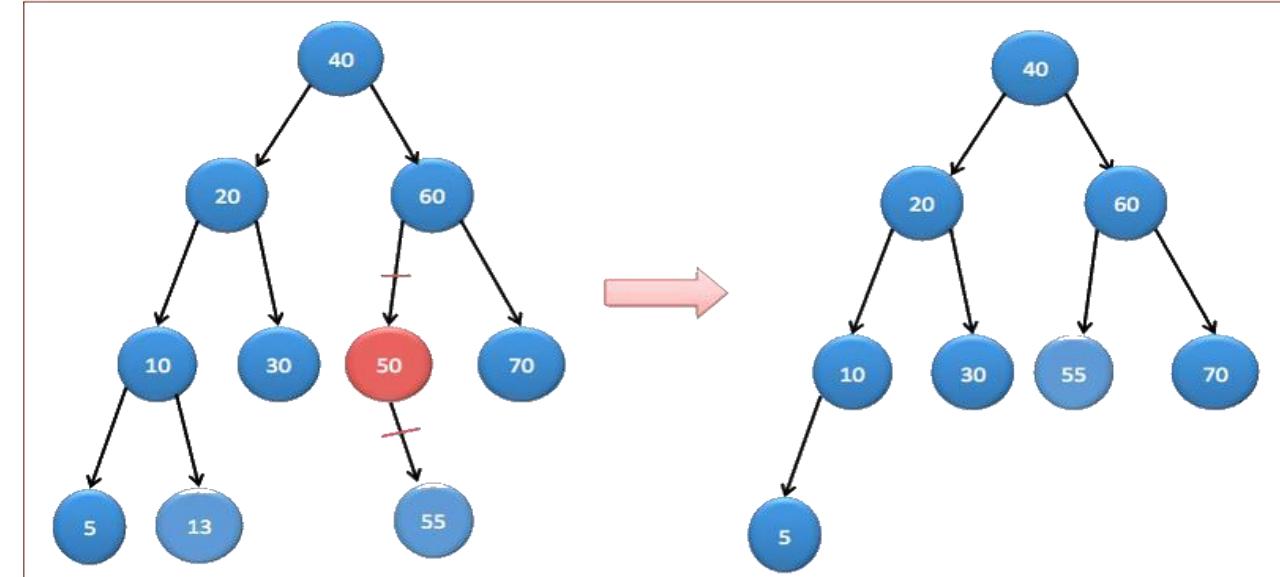


Deletion.

- 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.



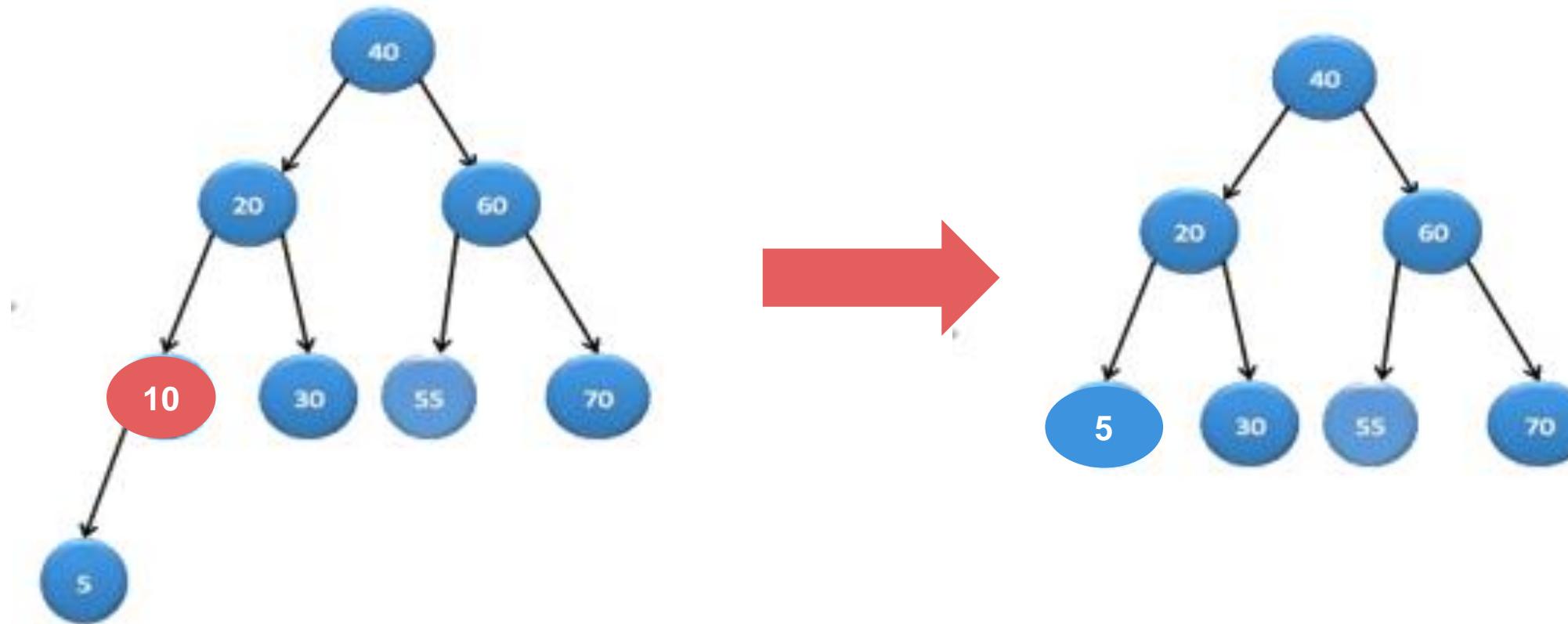
No children



Node with right subtree

Deletion.

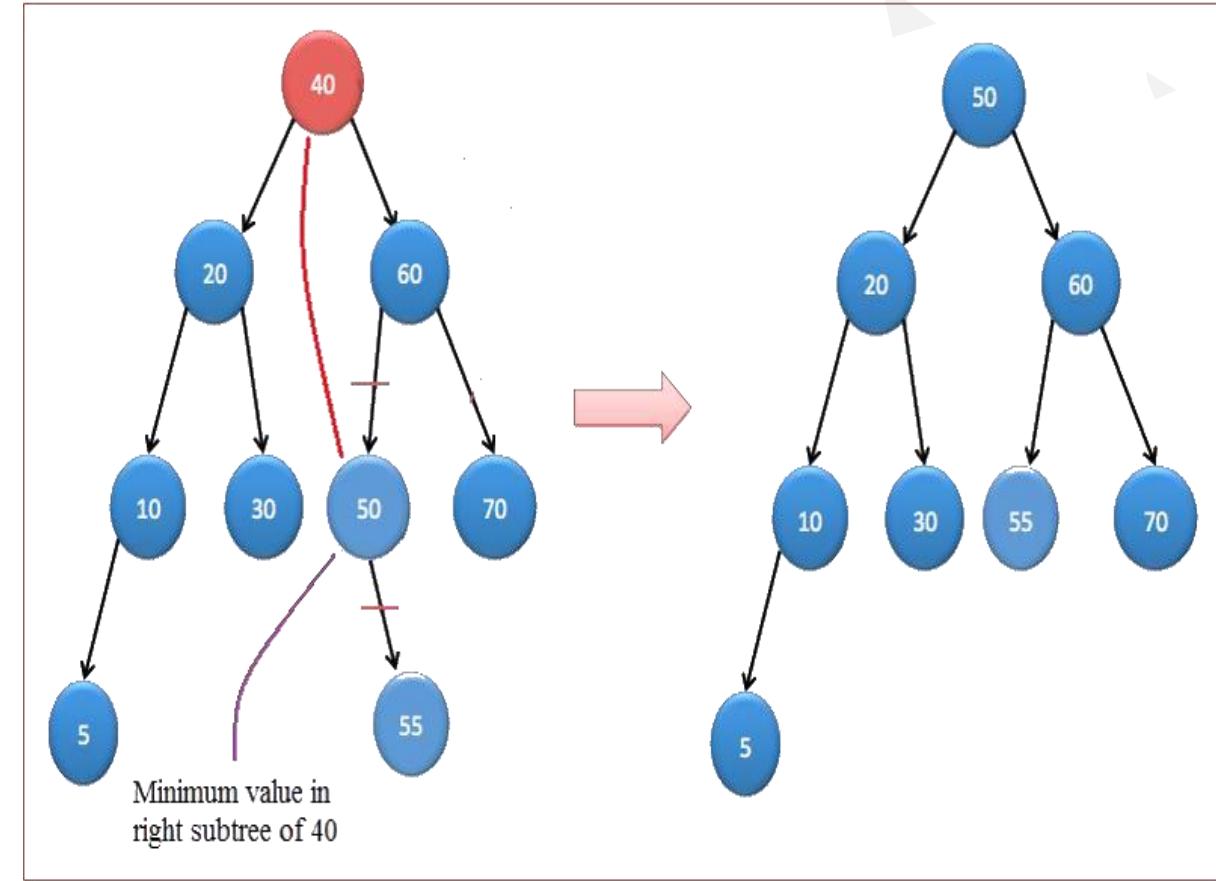
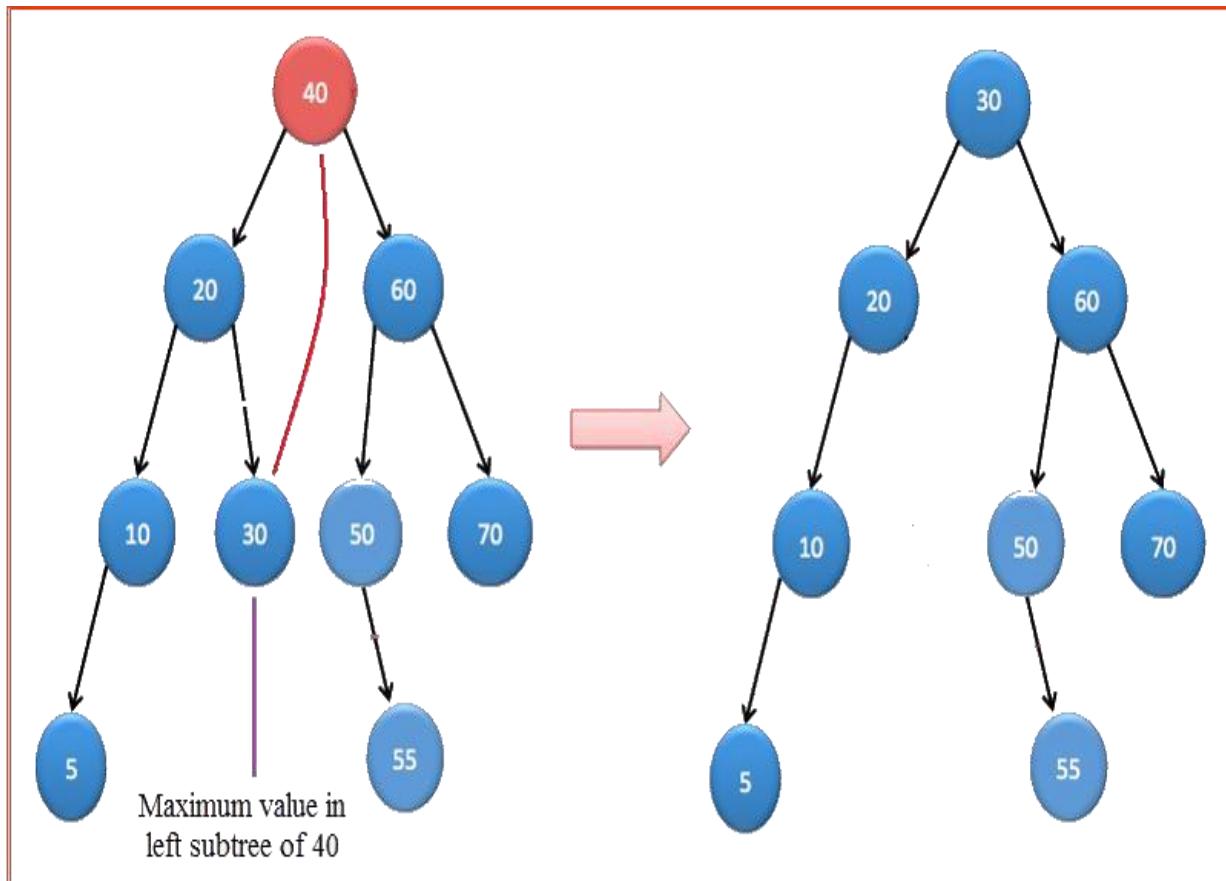
3. The node to be deleted has **only a left subtree**. Delete the node and **attach the left subtree** to the deleted node's parent.



Node with left subtree

Deletion.

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



Node with two subtrees



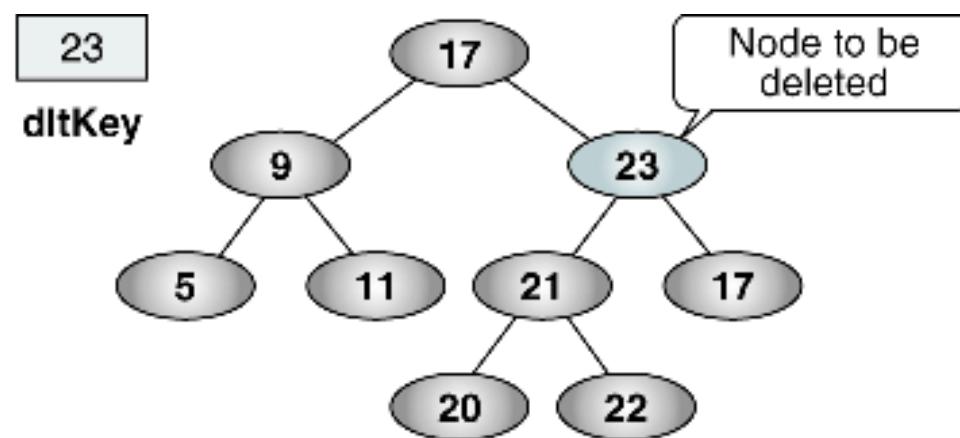
Deletion of node with two subtrees – two ways

- 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.

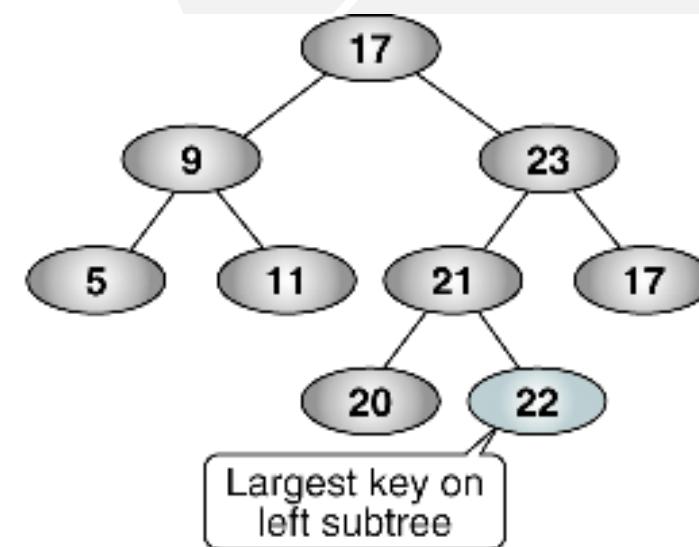
Or

- 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.

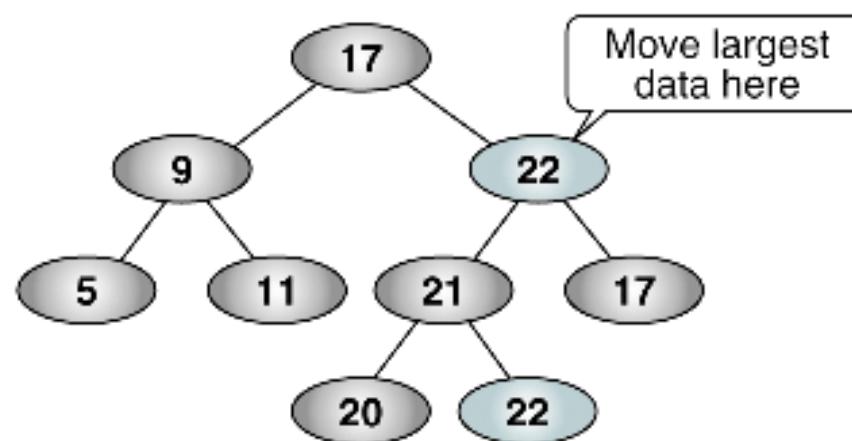




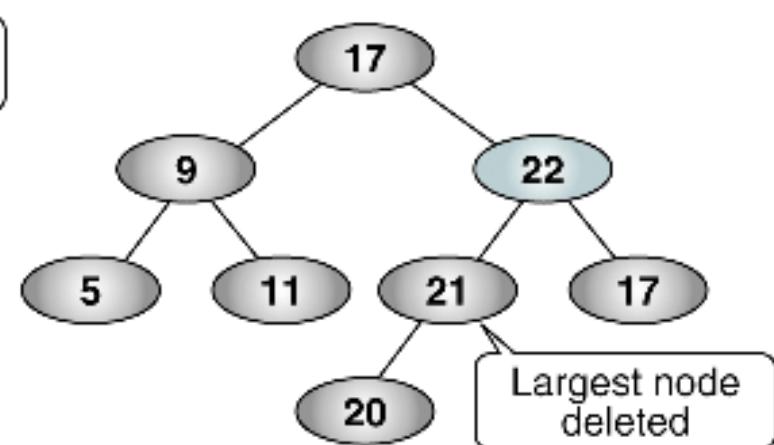
(a) Find dltKey



(b) Find largest



(c) Move largest data

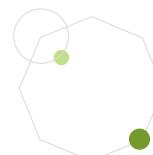


(d) Delete largest node

FIGURE 7-10 Delete BST Test Cases

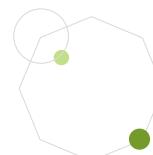
ALGORITHM 7-5 Delete Node from BST

```
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
```



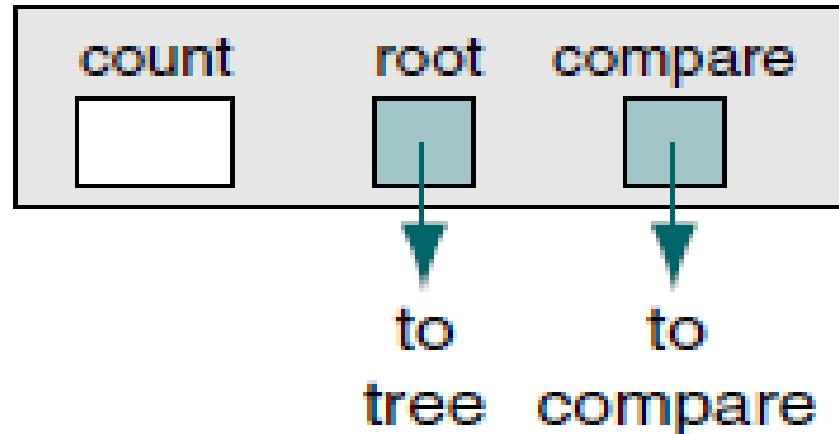
Binary Search Tree ADT.

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

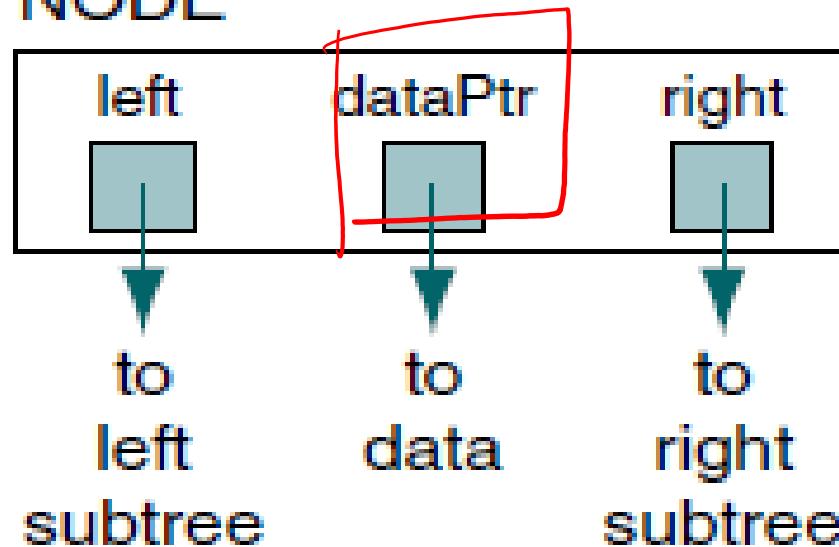
Data Structure

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

BST_TREE



NODE

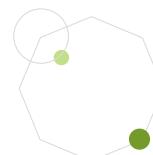


BST APPLICATIONS.



Integer Application

Student List Application



Integer Application:

Reading integers (values) from the keyboard and constructing BST

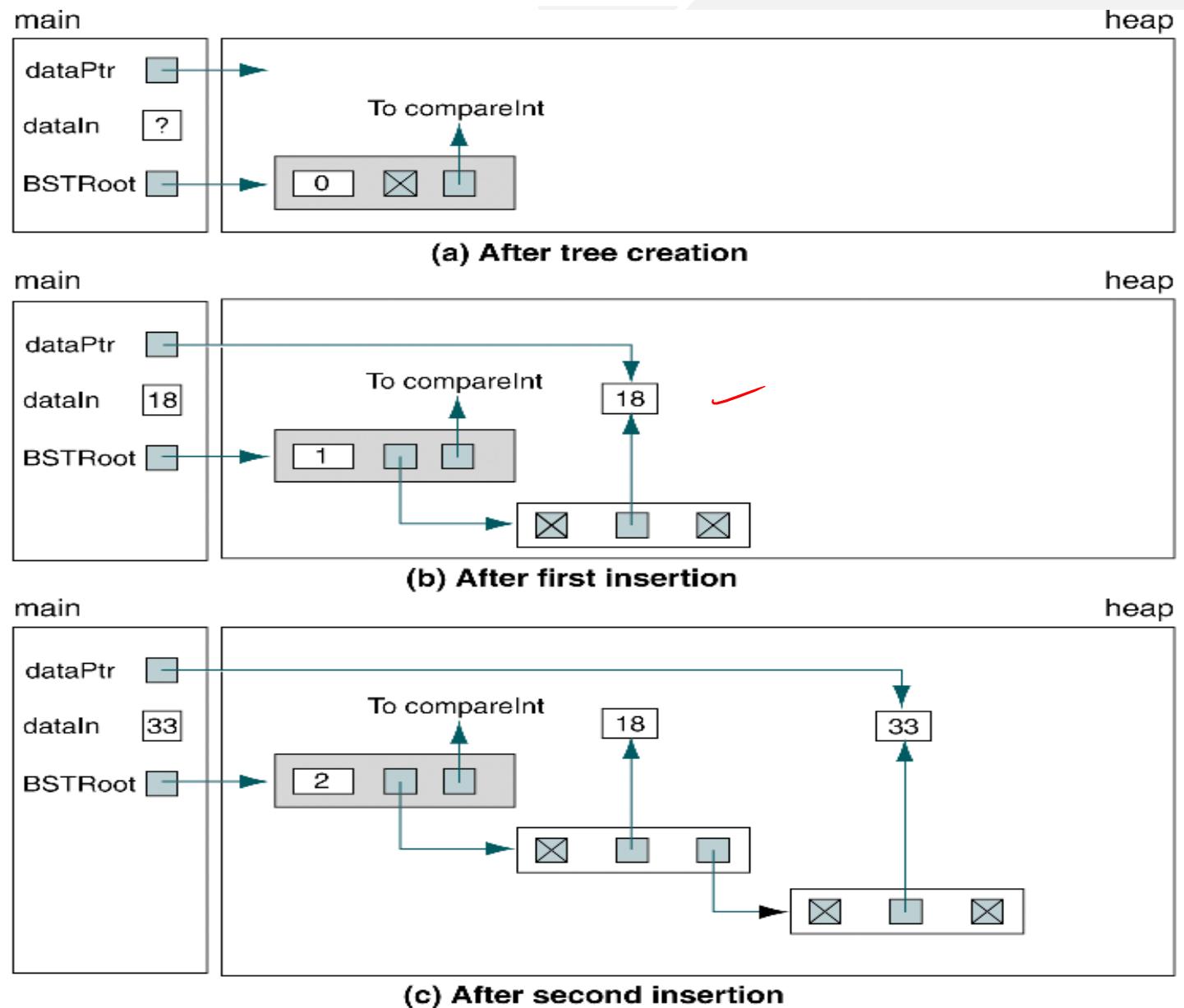


FIGURE 7-13 Insertions into a BST

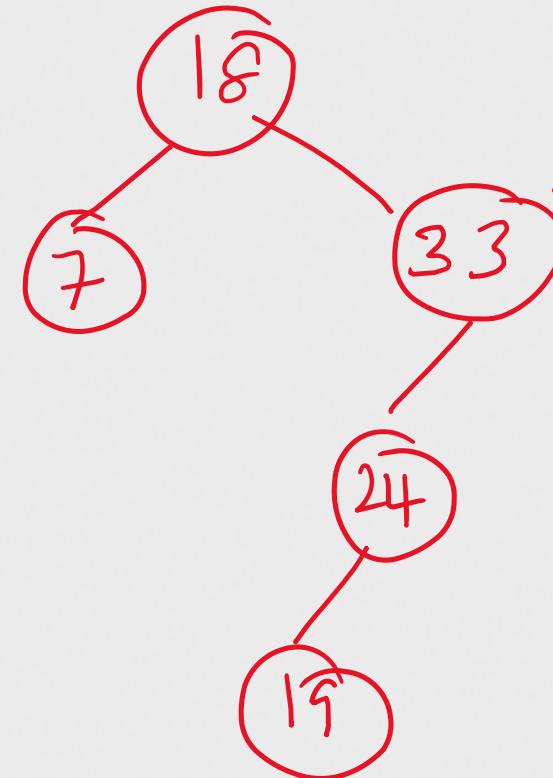
PROGRAM 7-16

```
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

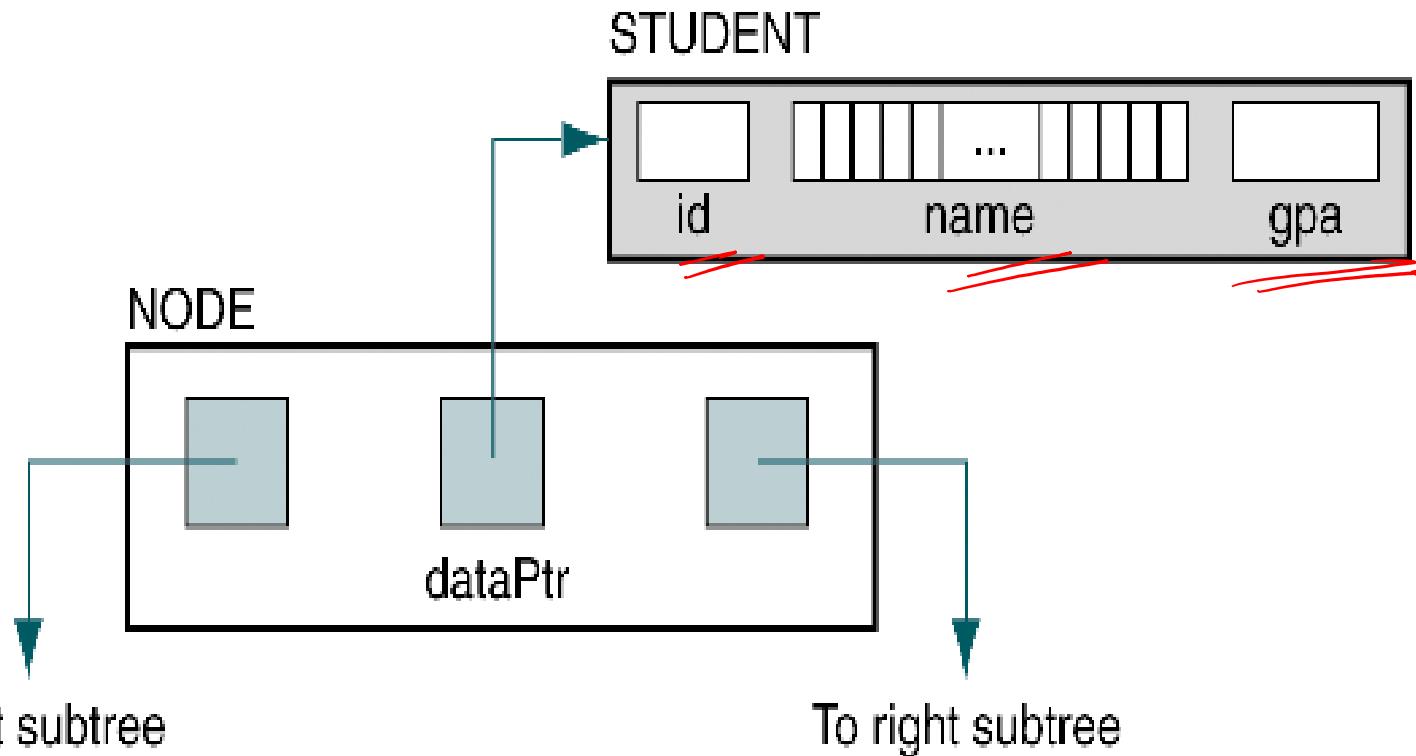


FIGURE 7-14 Student Data in ADT

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.



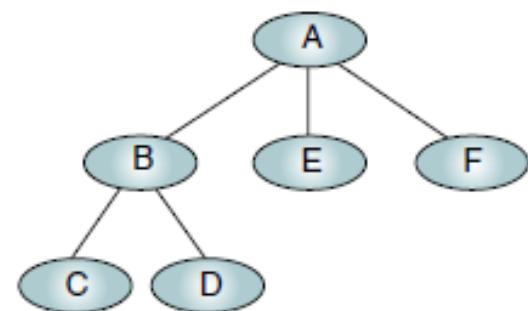
INSERTION INTO GENERAL TREES.

- 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

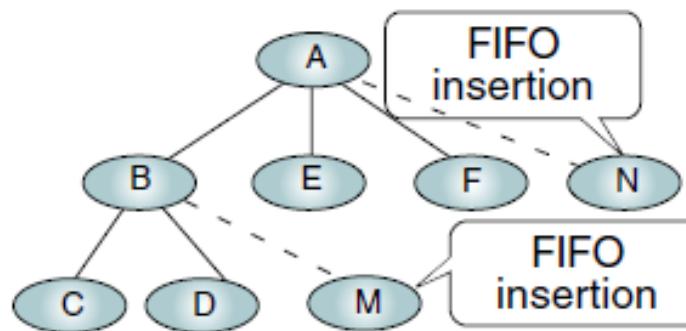


FIFO INSERTION.

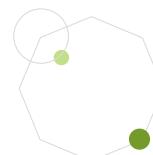
- When using FIFO insertion, **Insert the nodes at the end of the sibling list**, much as we insert a new node at the **rear of a queue**.
- When the list is then processed, the siblings are processed in FIFO order. FIFO order is used when the application requires that the data be processed in the order 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

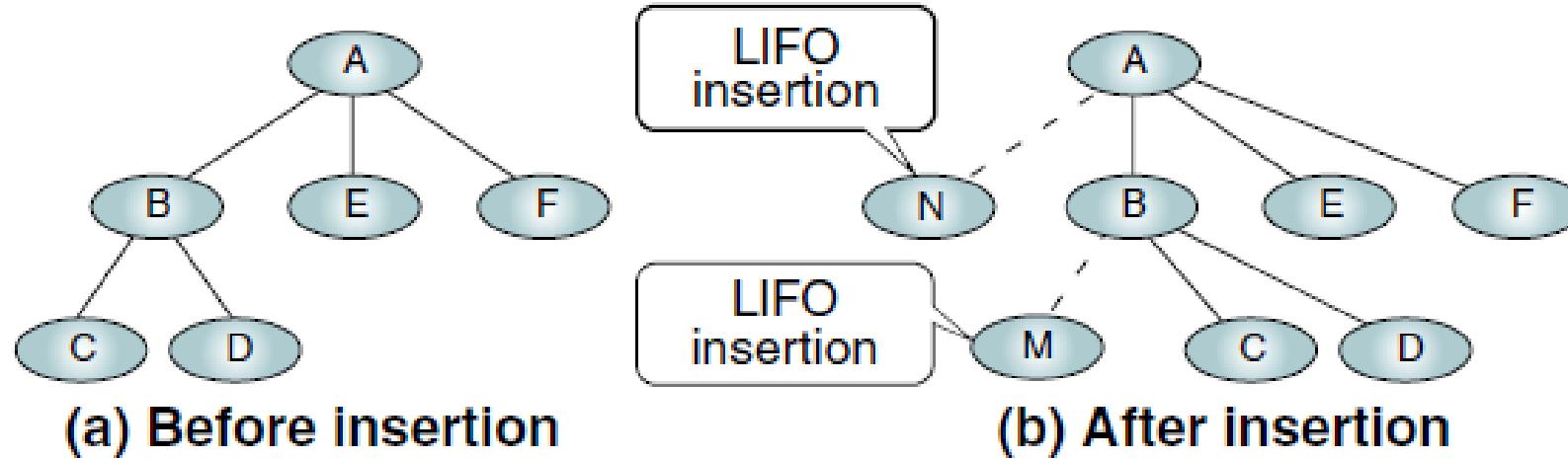


(b) After insertion



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.

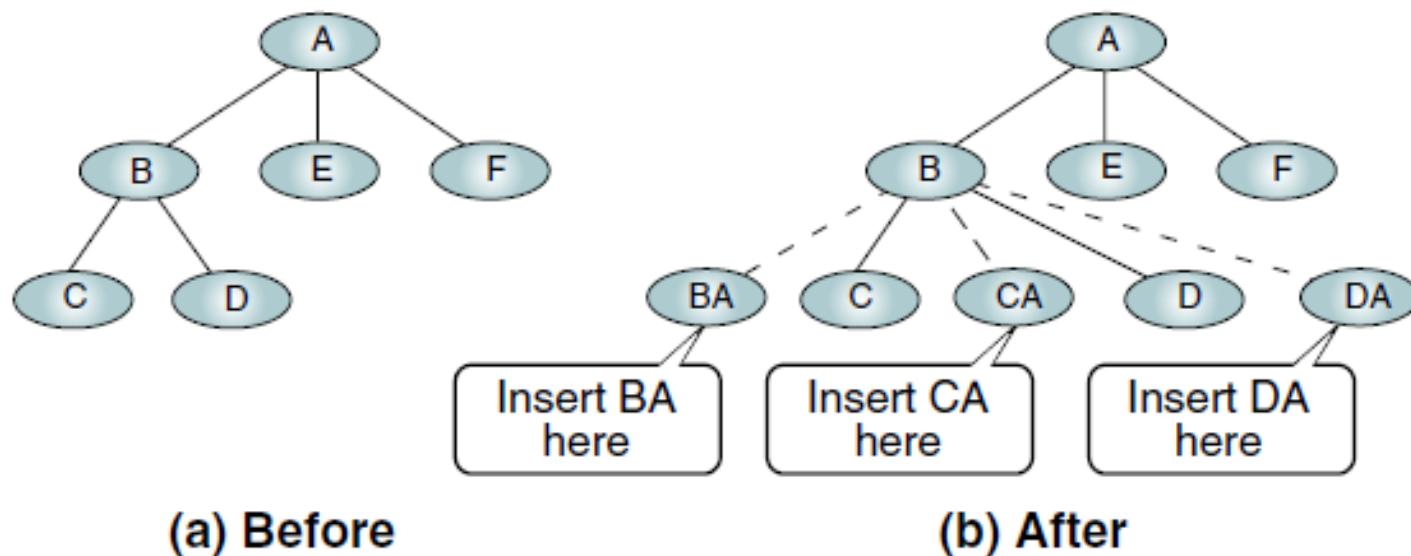


LIFO Insertion into General Trees



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.

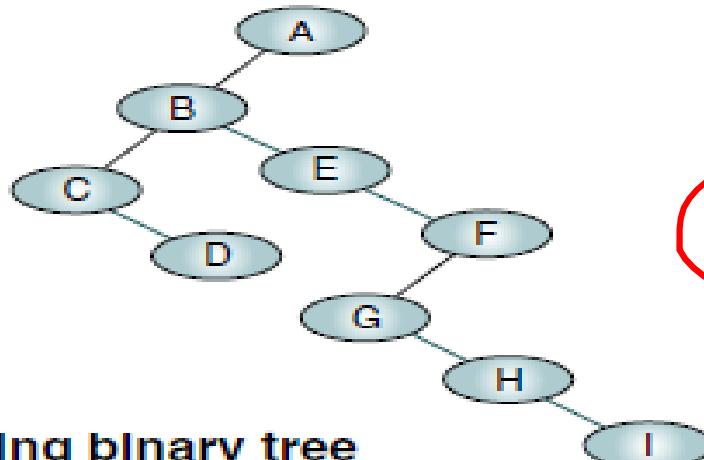
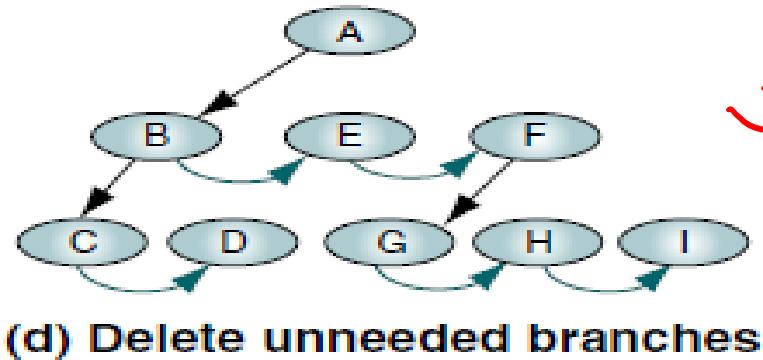
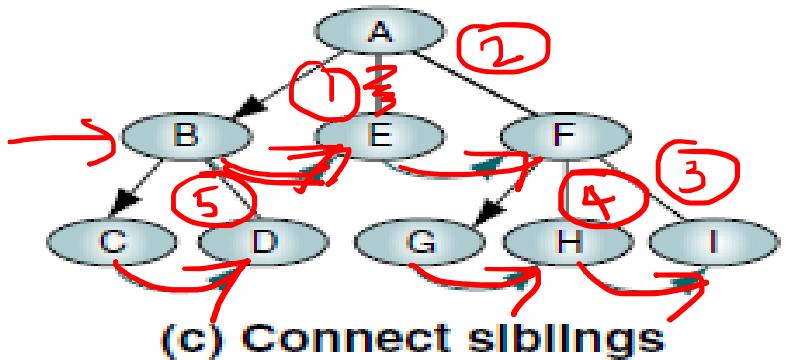
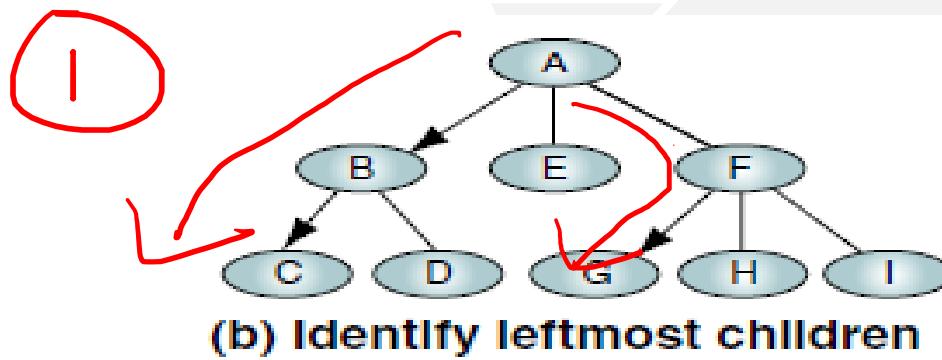
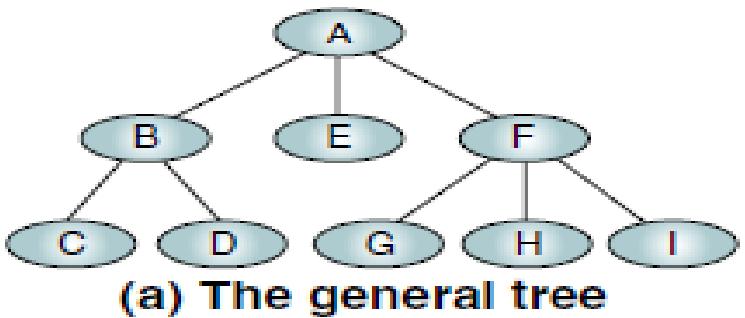
- No standard rules for general tree insertions,
- BUT **standard deletion rules are:**
 1. **A node may be deleted only if it is a leaf.** In the general tree, this means a node cannot be deleted if it has any children. If the user tries to delete a node that has children, the program provides an error 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.

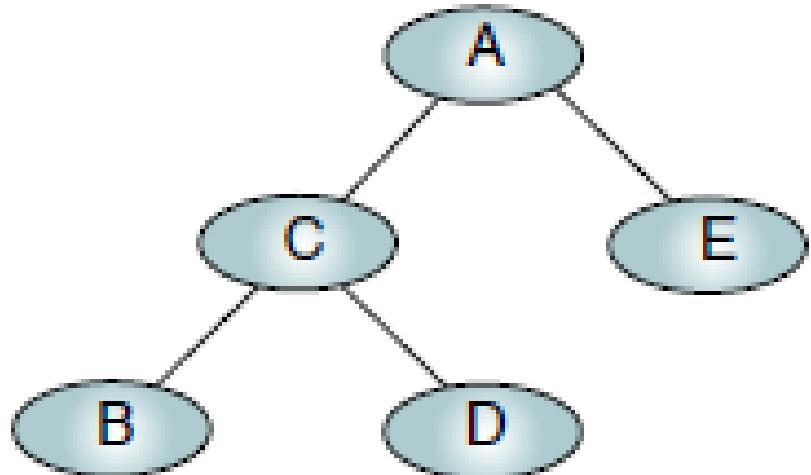


THREADED TREES.

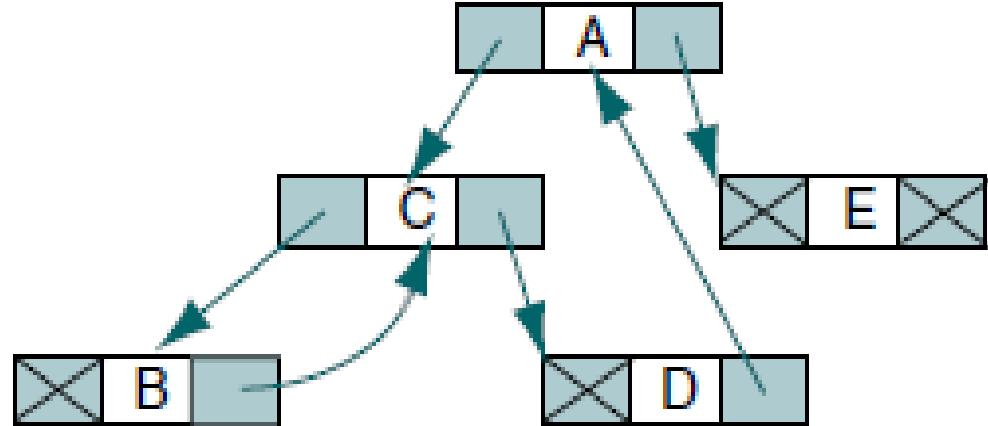
- Binary tree traversal algorithms are written using either recursion or programmer-written stacks. If the tree must be traversed frequently, **using stacks** rather than recursion may be more **efficient**.
- A third alternative is a ***threaded tree***. In a threaded tree, **null pointers are replaced 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.

THREADED TREES.

Inorder traversal (LNR)

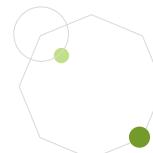


(a) Binary tree



(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

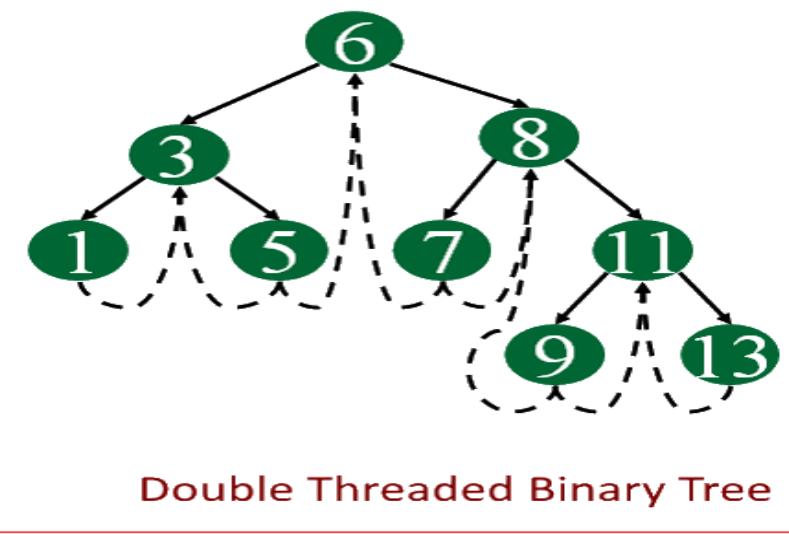
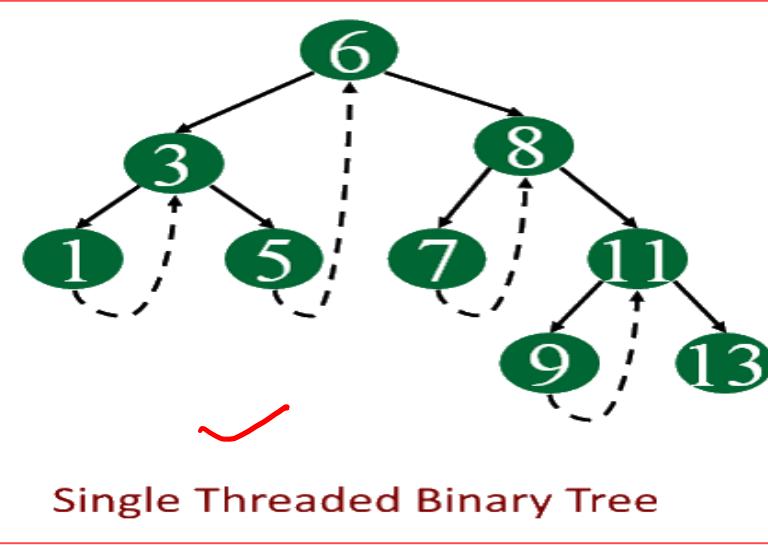
- A binary tree with n nodes has $n + 1$ null pointers

- Waste of space due to null pointers

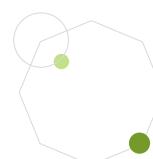
- Replace by threads pointing to inorder successor and/or inorder predecessor (if any)

INORDER TRAVERSAL IS:-

1 3 5 6 7 8 9 11 13



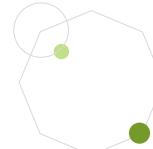
- Single threaded: Makes inorder traversal easier
- Double threaded: Makes inorder and postorder easier



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)**

**0: Even
High (EH)**

**+1: Left
High (LH)**

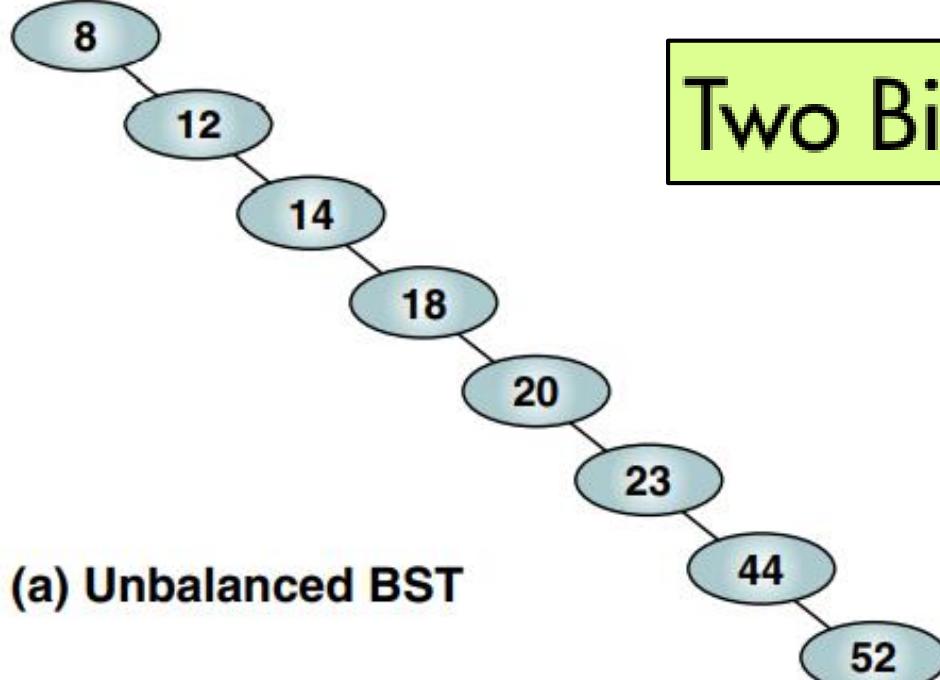
LST is shorter than RST

LST is equal to RST

LST is longer than RST

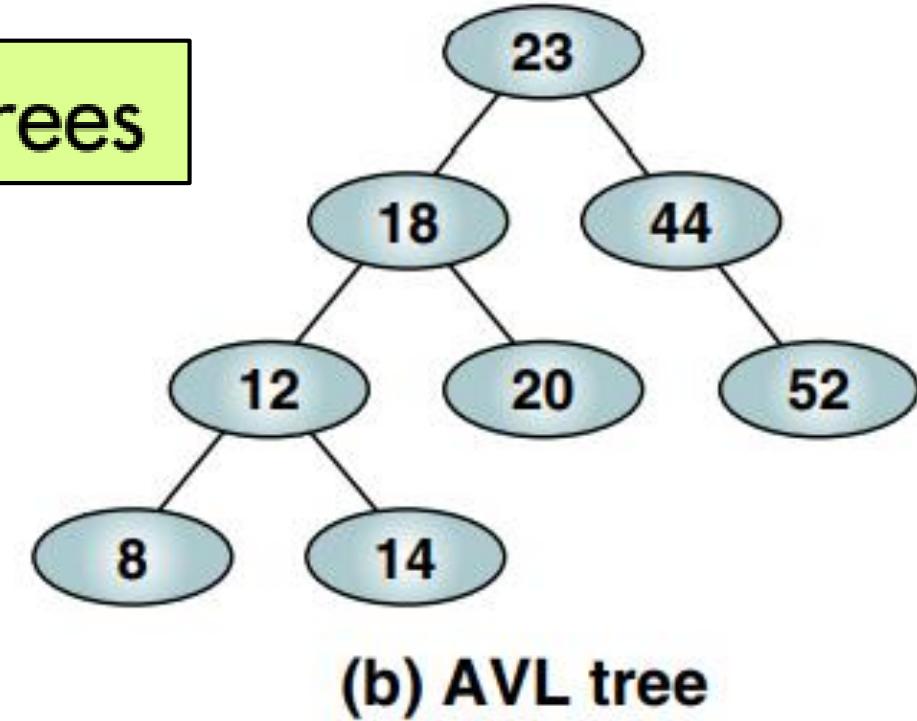


It takes **2** tests to locate 12.
It takes **3** tests to locate 14.
It takes **8** tests to locate 52.



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.

Two Binary Trees



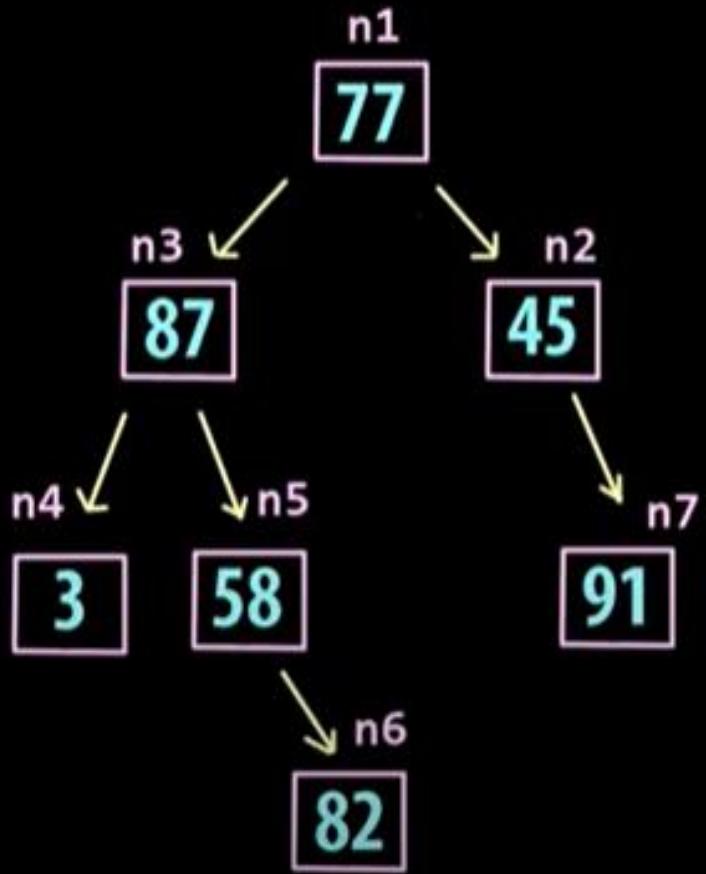
Search effort for this
binary search tree is $O(n)$

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

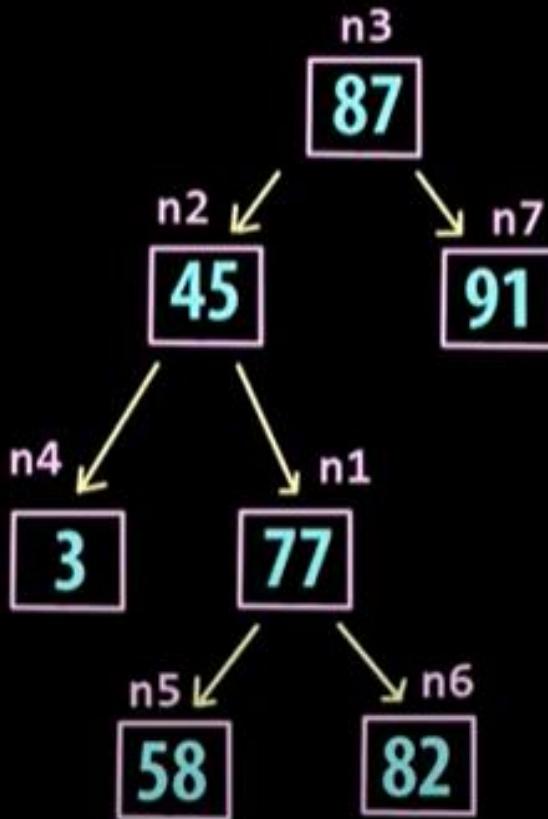
Examples for AVL Trees.

An AVL tree is a height-balanced binary search tree

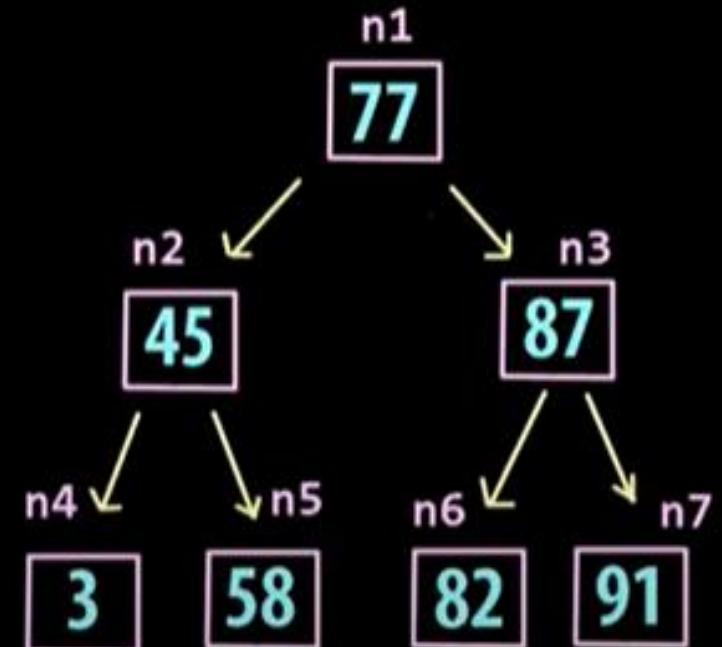
Binary Tree



Binary SEARCH Tree



AVL Tree



AVL SEARCH TREES – Definition.

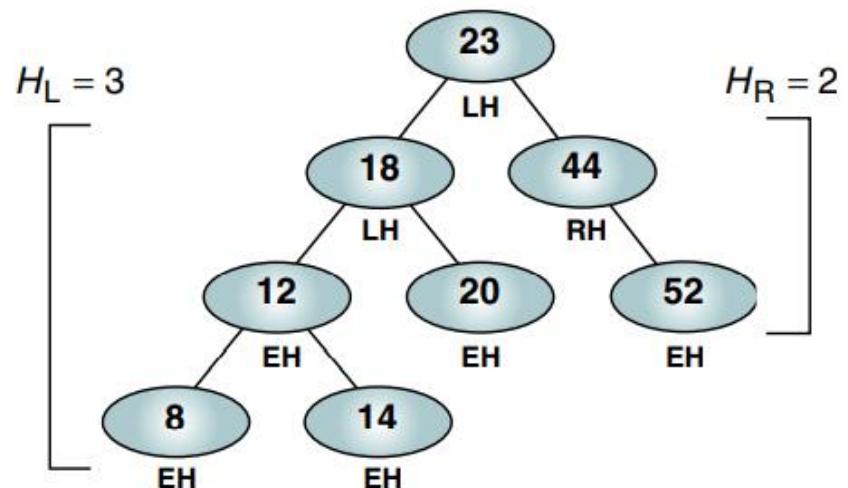
An AVL tree is a binary tree that either is empty or consists of two AVL subtrees, TL, and TR, whose heights differ by no more than 1.

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

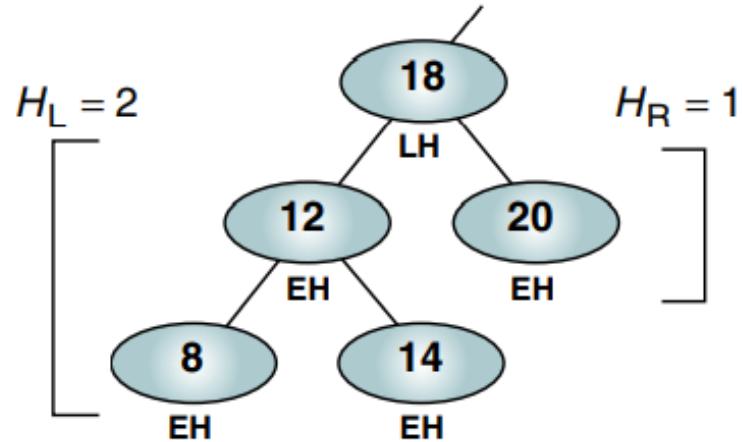
H_L is the *height of the left subtree* and H_R is the *height of the right subtree*.

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

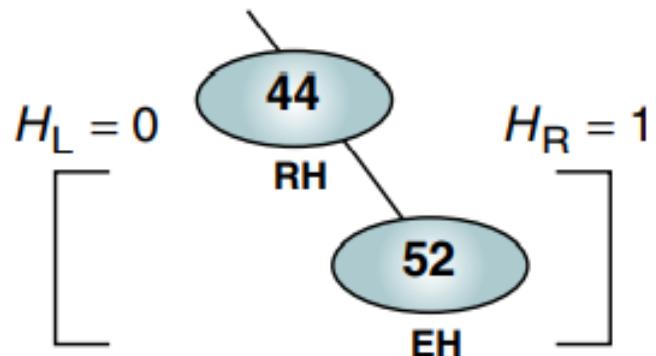
BALANCE FACTORS of AVL Trees.



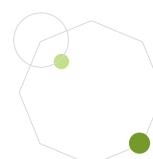
(a) Tree 23 appears balanced: $H_L - H_R = 1$



(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.**

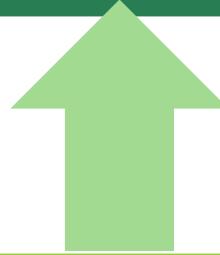


Consider the basic balancing algorithms for four cases of unbalancing:

UNBALANCED

Right of right

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

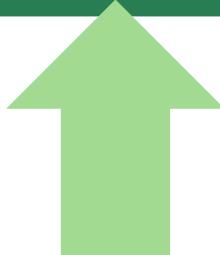


Left Rotation

Single Rotation
Left

Left of left

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



Right Rotation

Single Rotation
Right

Right of left:

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



Left Right Rotation

Double Rotation

Left of right:

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

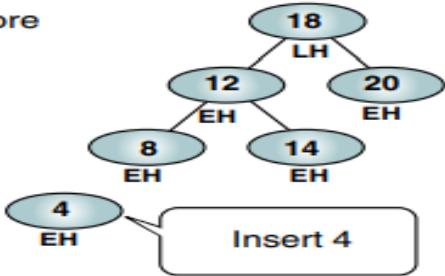


Right Left Rotation

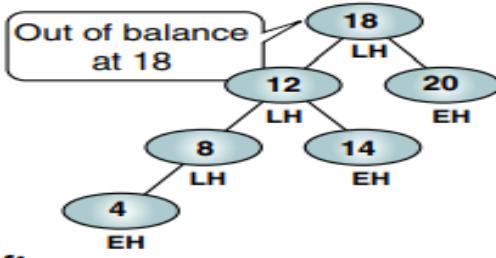
Double Rotation



Before

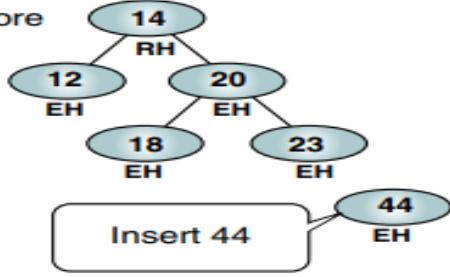


After

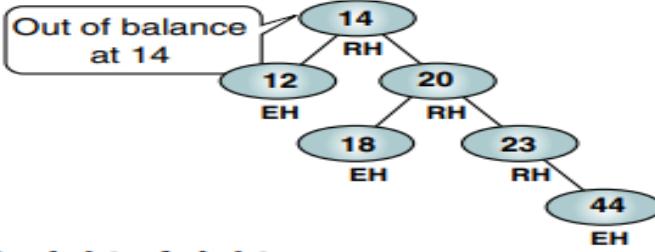


(a) Case 1: left of left

Before

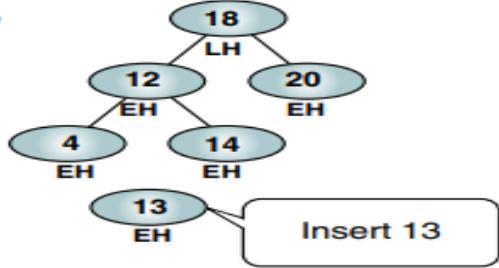


After

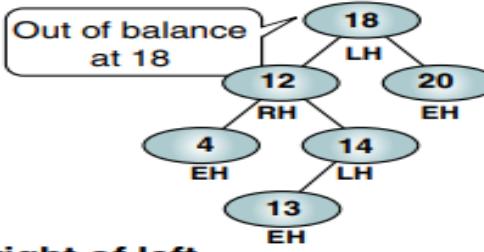


(b) Case 2: right of right

Before

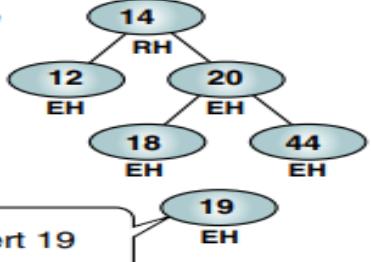


After

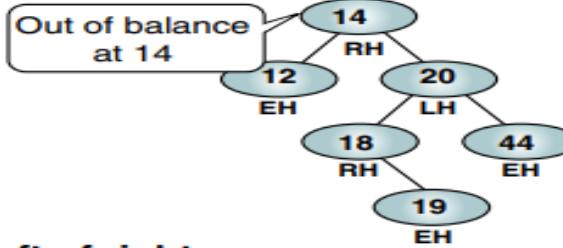


(c) Case 3: right of left

Before



After



(d) Case 4: left of right

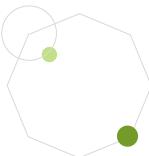
Out-of-balance AVL Trees



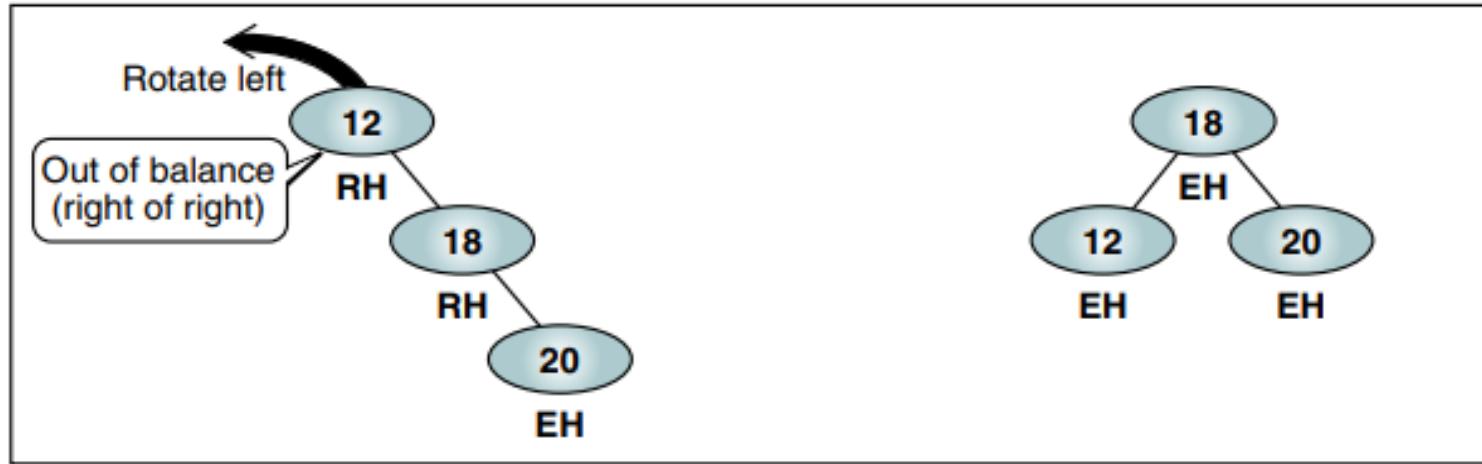
Case 1: Left Rotation or Single Rotation Left



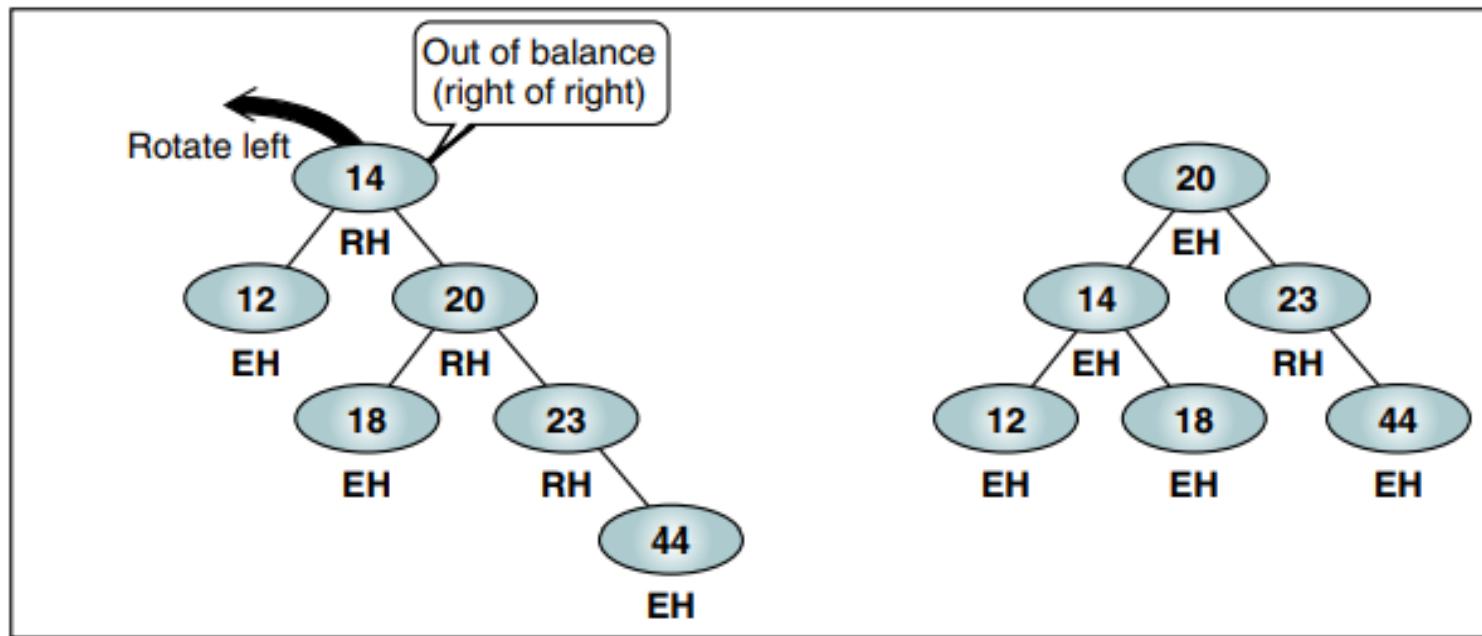
Insertion is Right of Right



Case 1: Left Rotation or Single Rotation Left



(a) Simple left rotation



(b) Complex left rotation

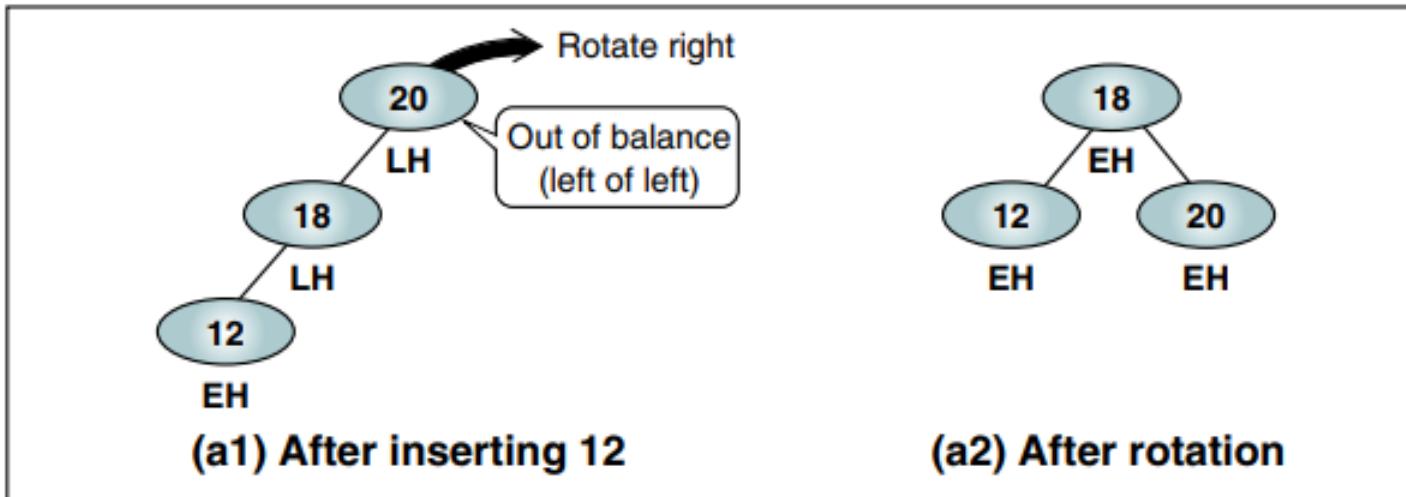
Case 2: Right Rotation or Single Rotation Right



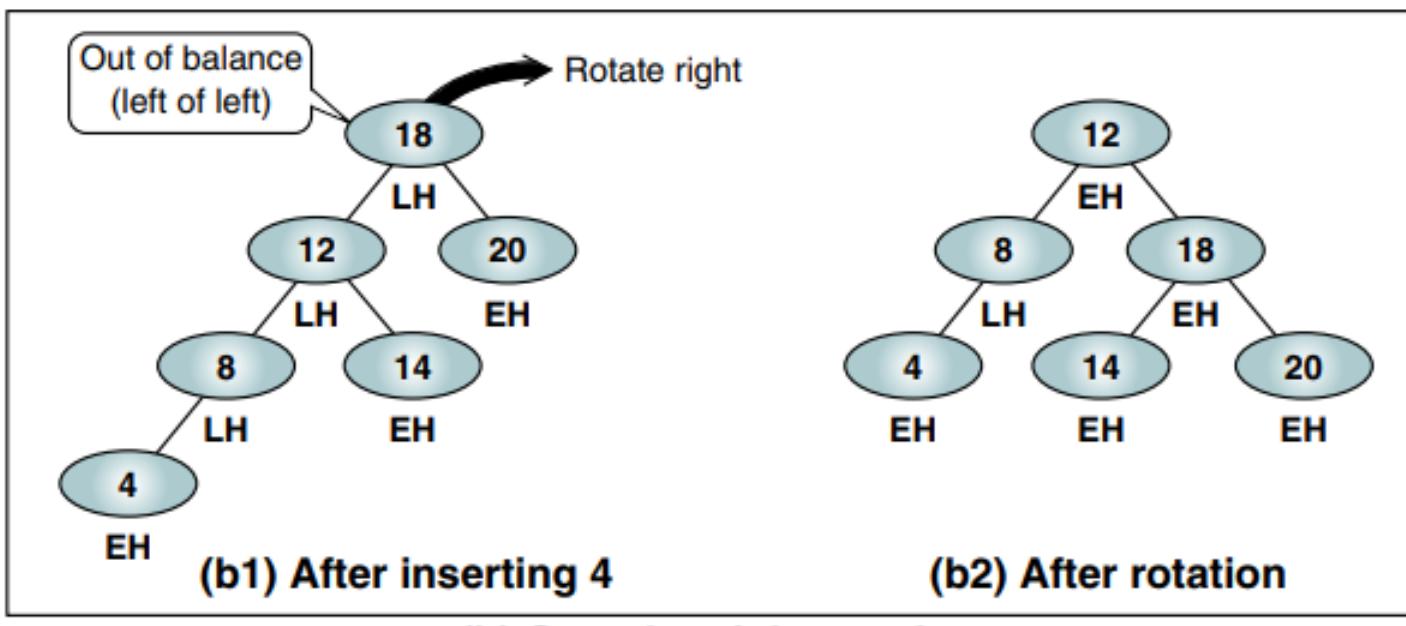
Insertion is Left of Left



Case 2: Right Rotation or Single Rotation Right



(a) Simple right rotation



(b) Complex right rotation

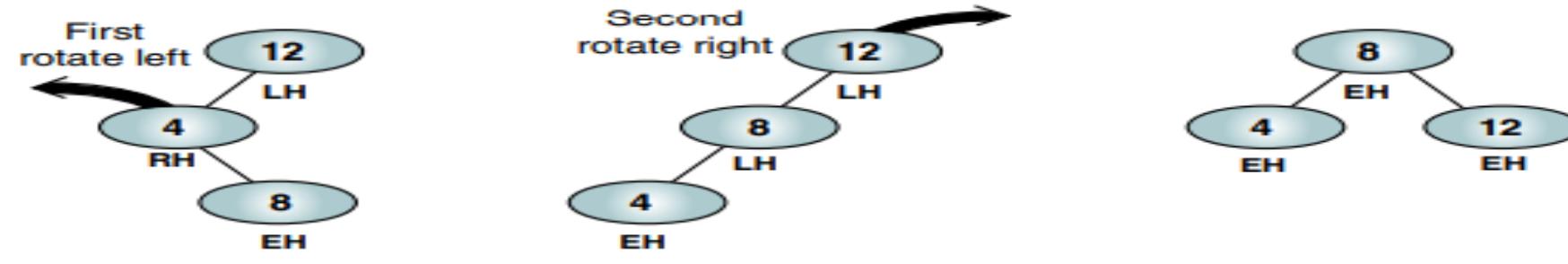
Case 3: Left - Right Rotation or Double Rotation



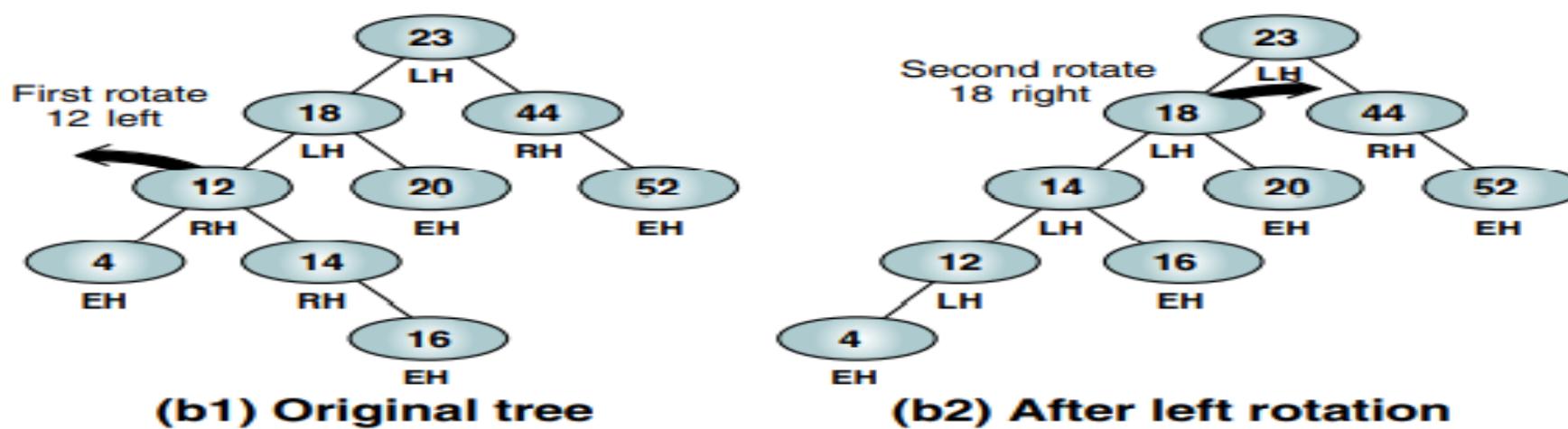
Insertion is Left and then Right (Right of left Insertion).



Case 3: Left - Right Rotation or Double Rotation Left



(a) Simple double rotation right



(b1) Original tree

(b2) After left rotation

(b3) After right rotation

(b) Complex double rotation right

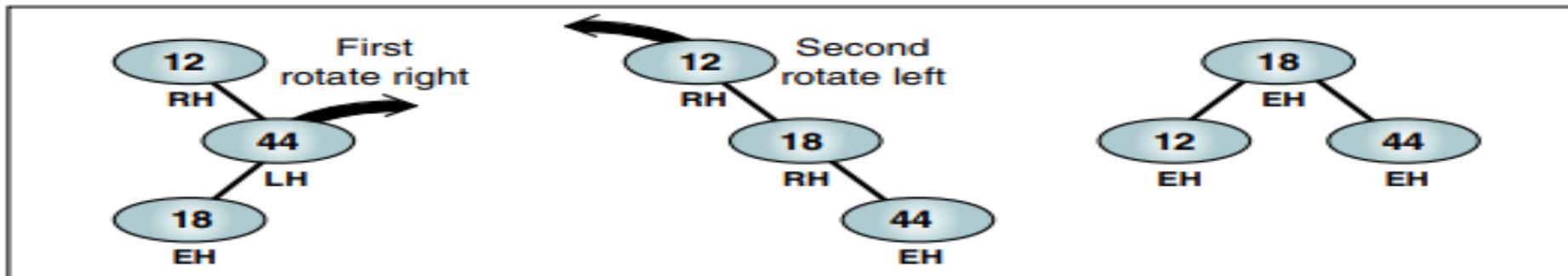
Case 4: Right-Left Rotation or Double Rotation



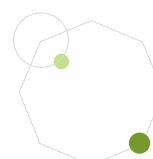
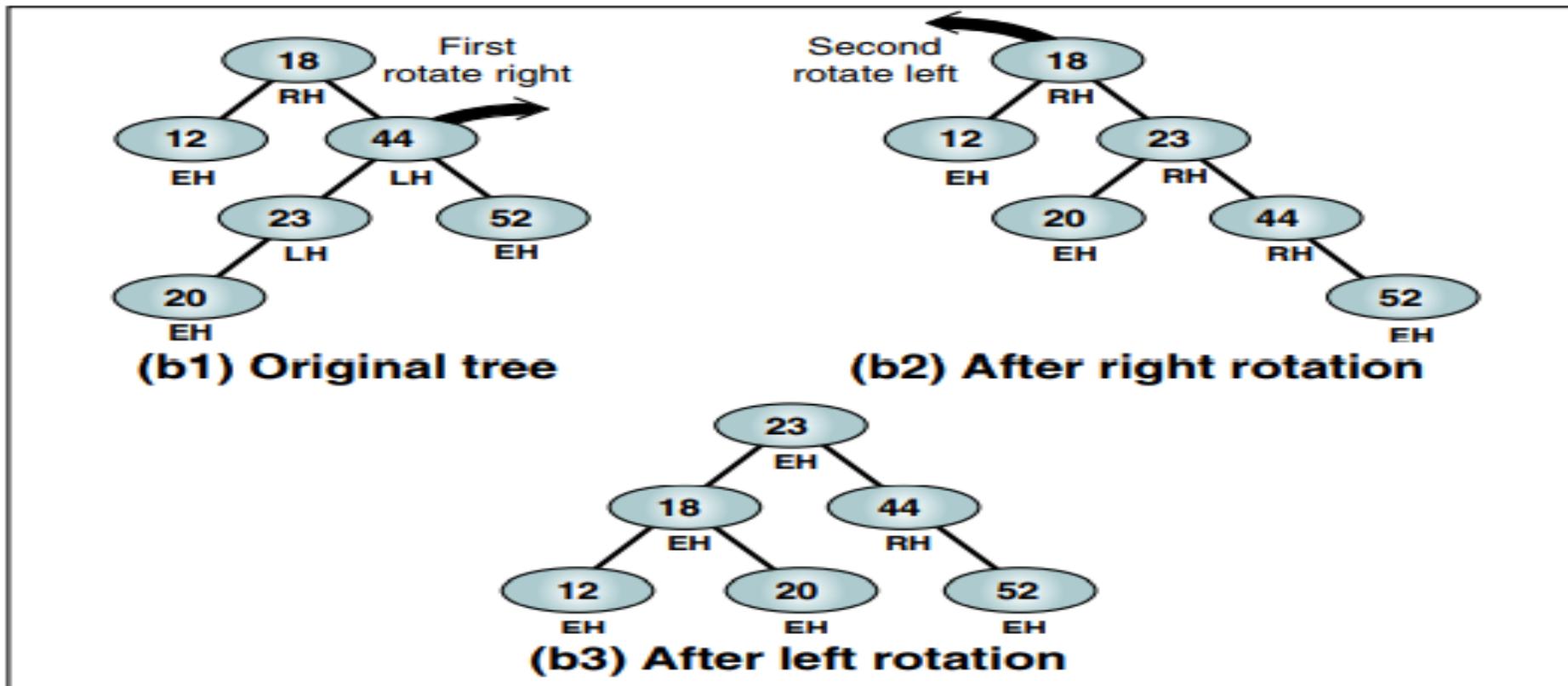
Insertion is Right and then left (**Left of Right Insertion**).



Case 4: Right-Left Rotation or Double Rotation



(a) Simple double rotation right



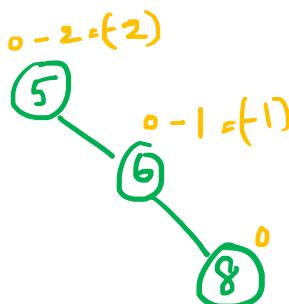
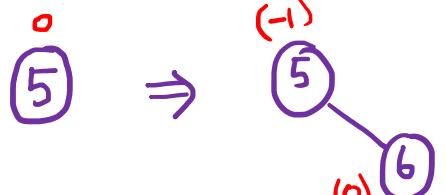
Insertion Operation on AVL trees



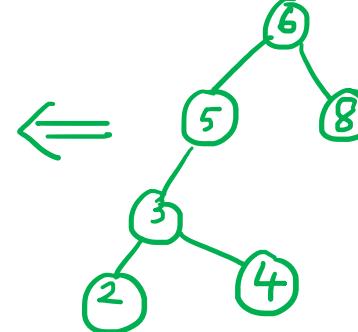
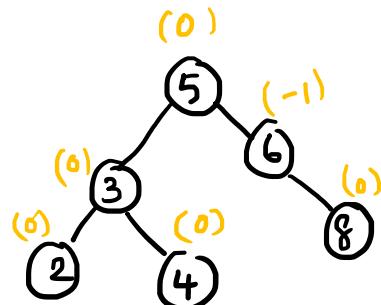
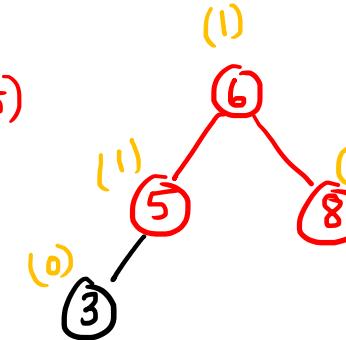
Insertion

Construction of AVL Trees:

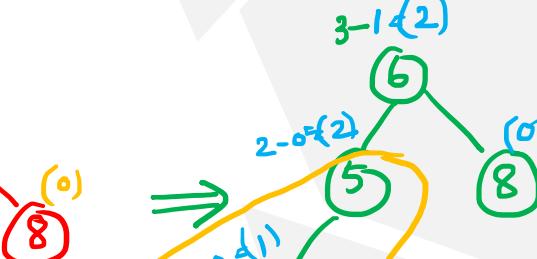
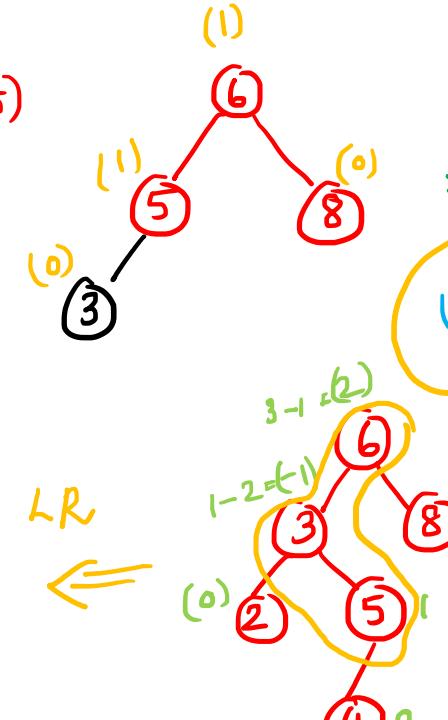
5, 6, 8, 3, 2, 4, 7



L(5)

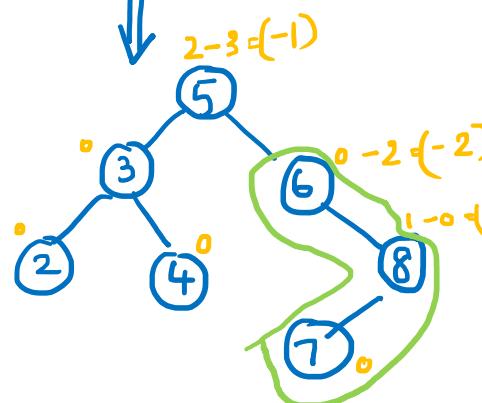


LR

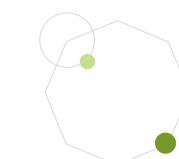
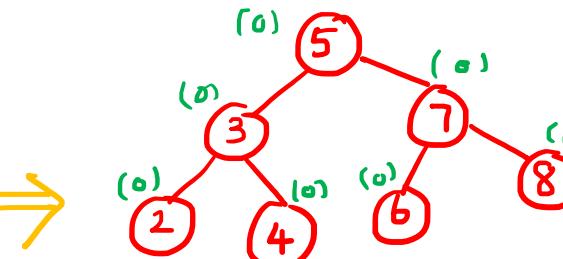
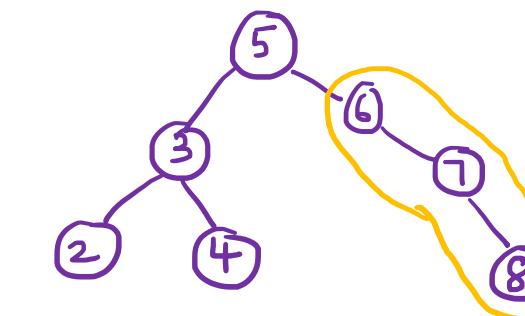


R(5)

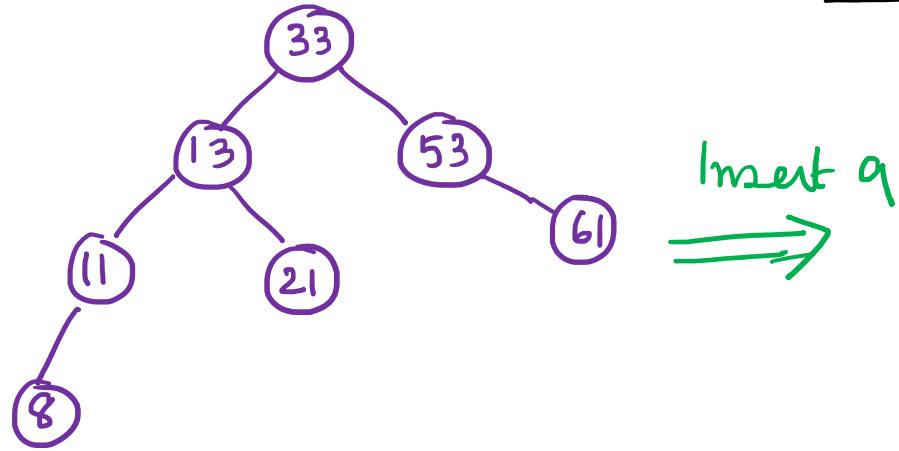
Insert 7



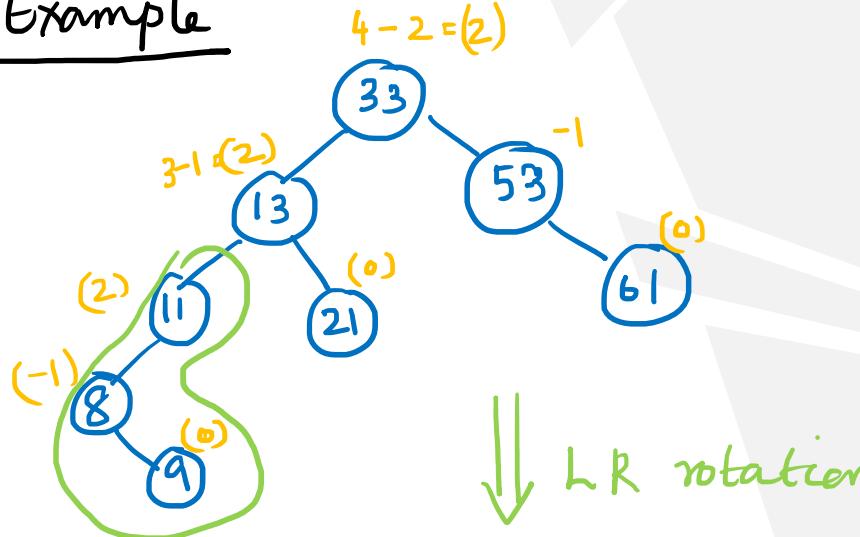
RL



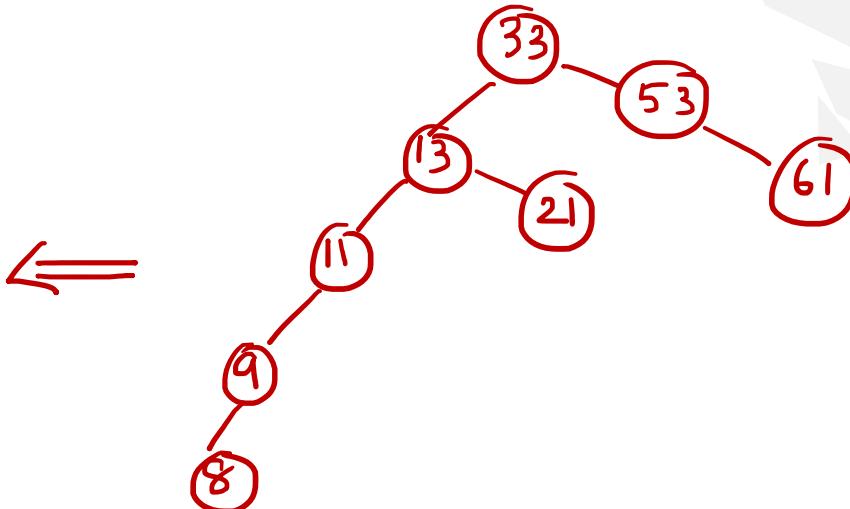
Insertion Example



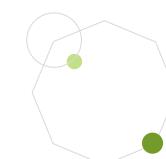
Insert 9



↓ LR rotation



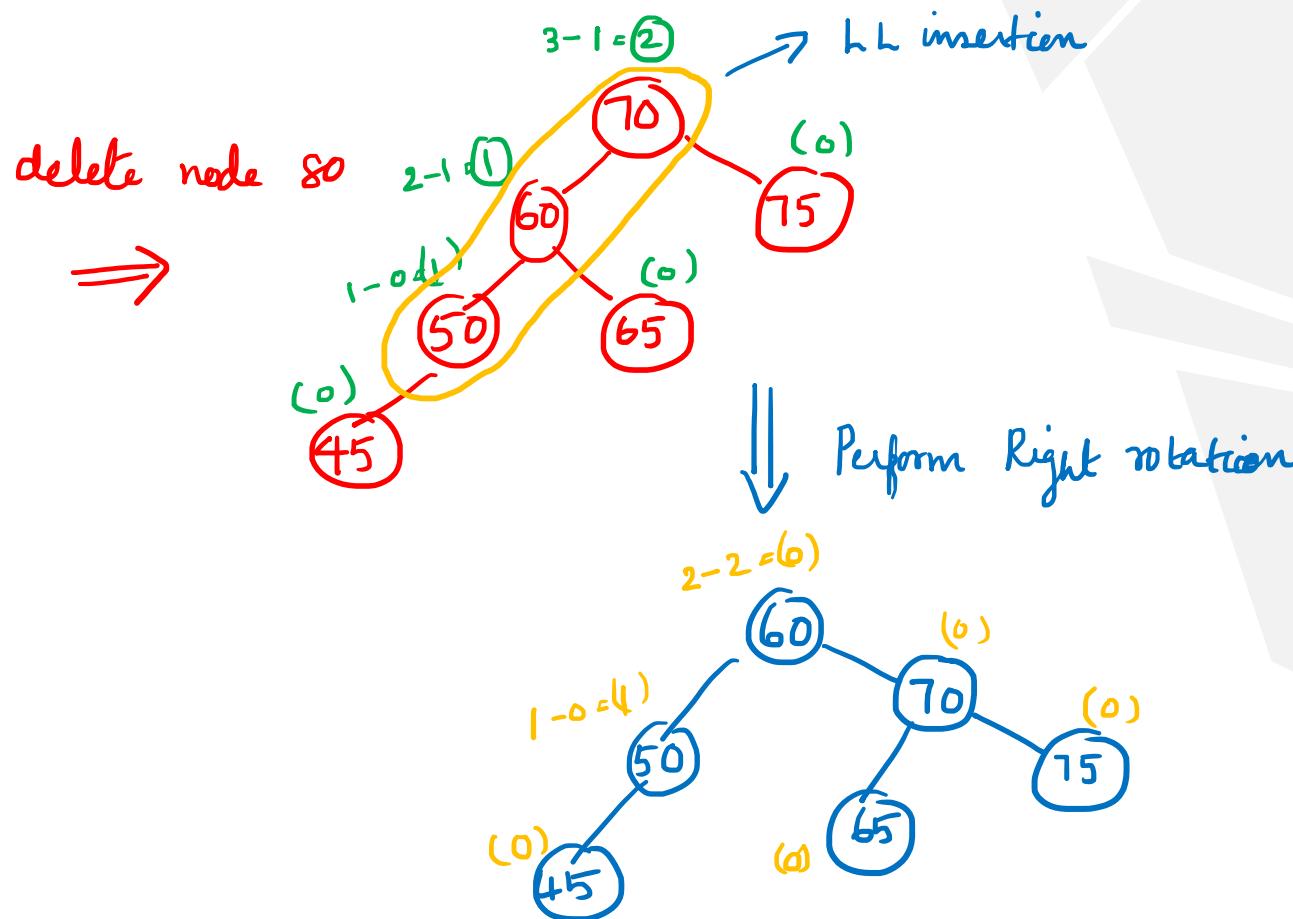
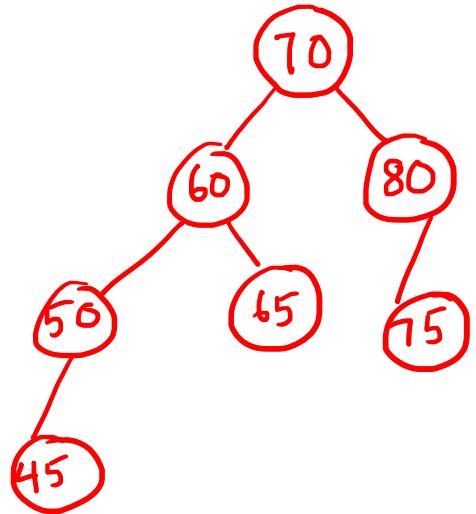
Resultant Tree AVL



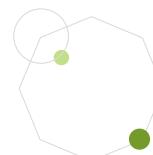
DELETION OPERATION ON AVL TREES



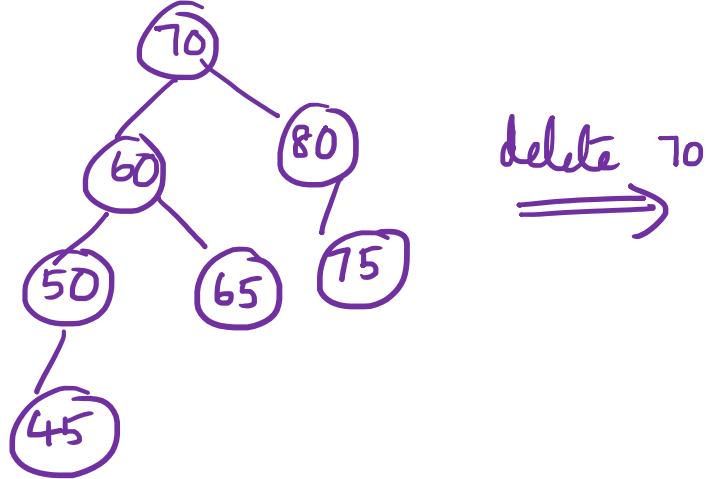
DELETION



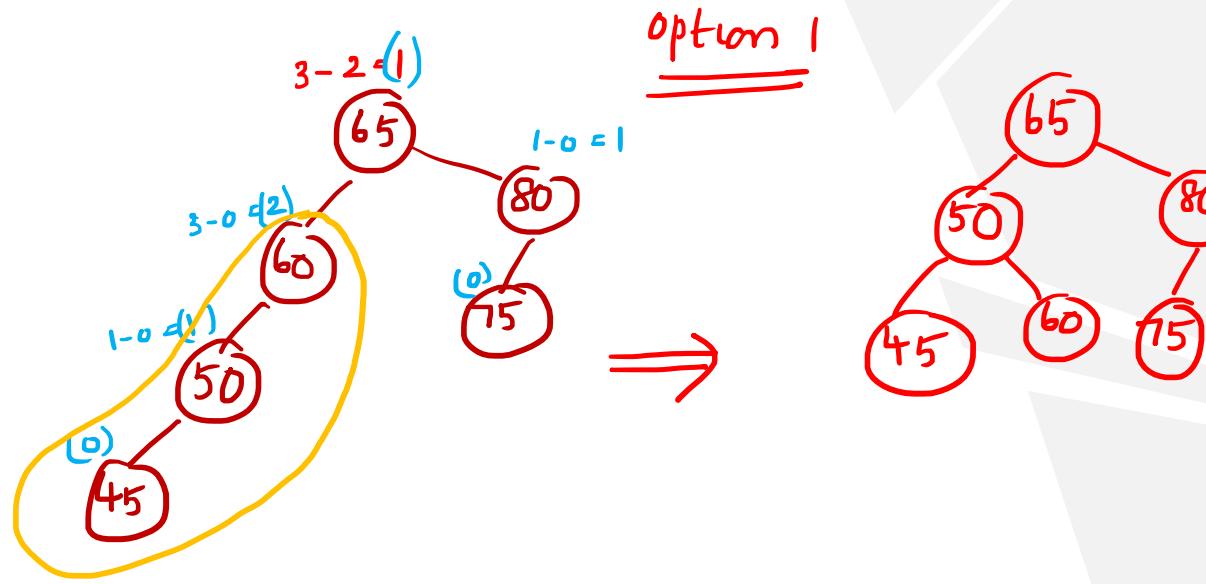
Resultant tree is
AVL tree
(balanced)



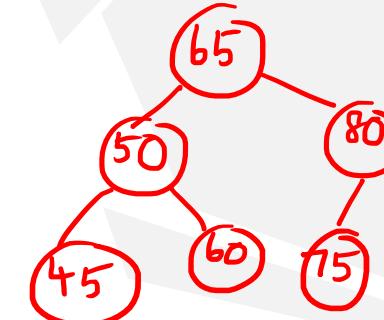
Example 2



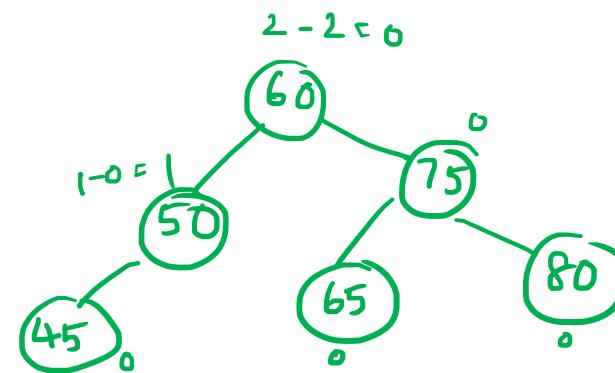
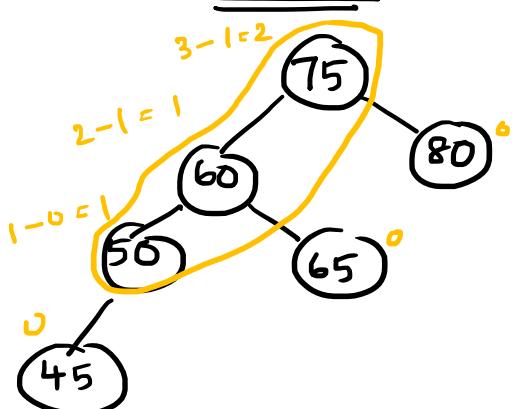
delete 70



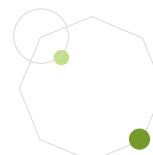
option 1



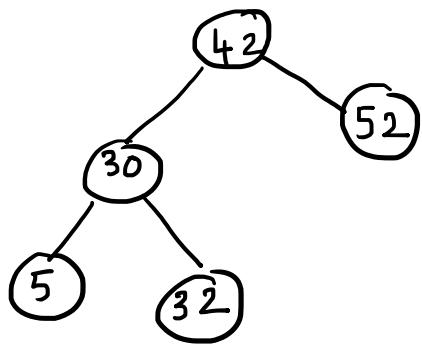
option 2



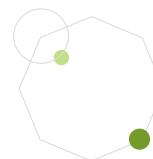
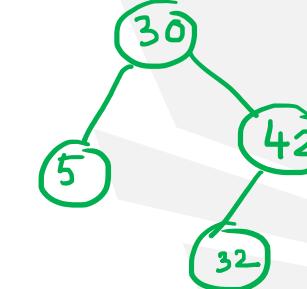
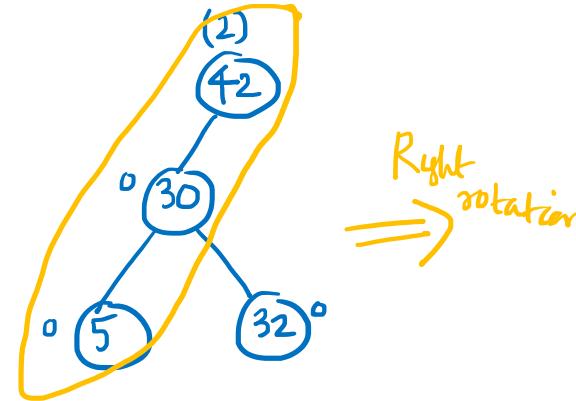
Resultant tree AVL [Balanced tree]



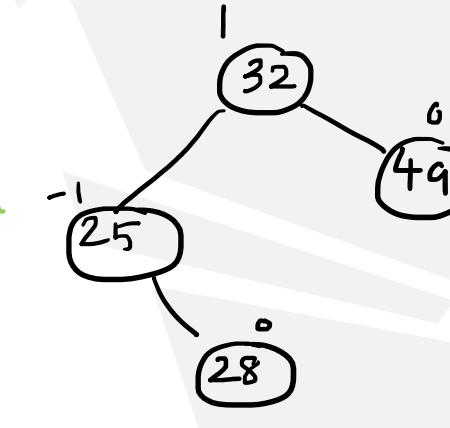
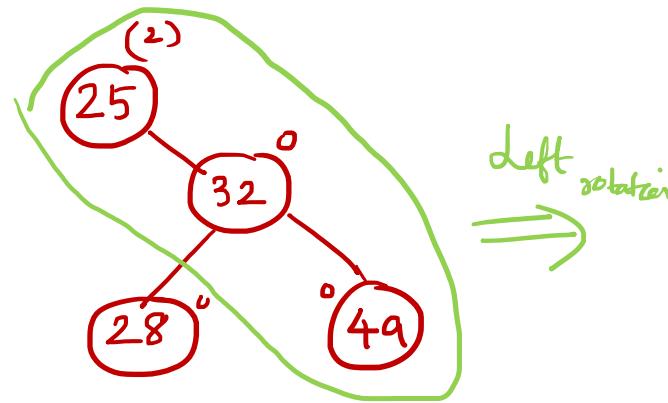
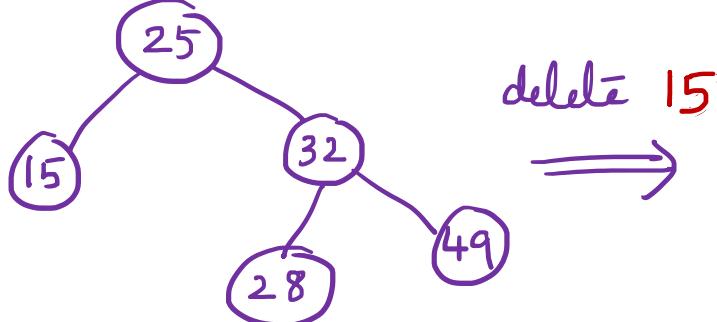
Example 3



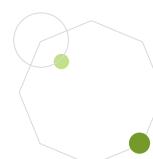
⇒ delete 52



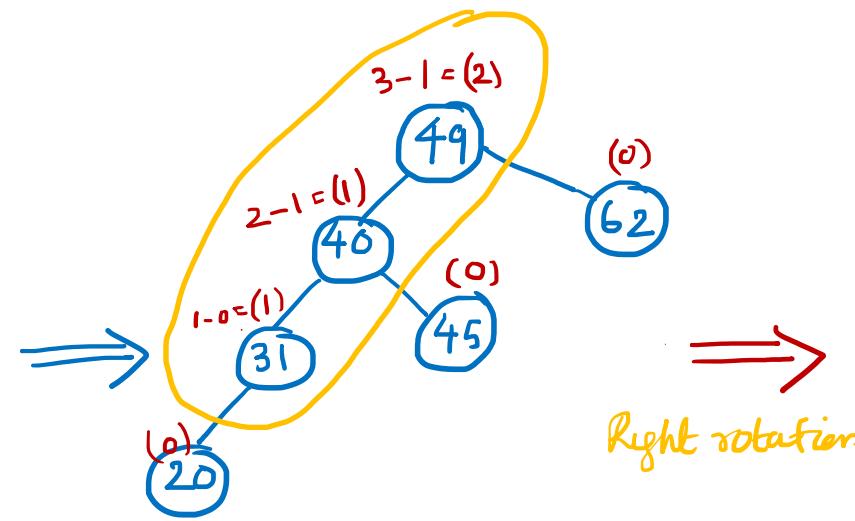
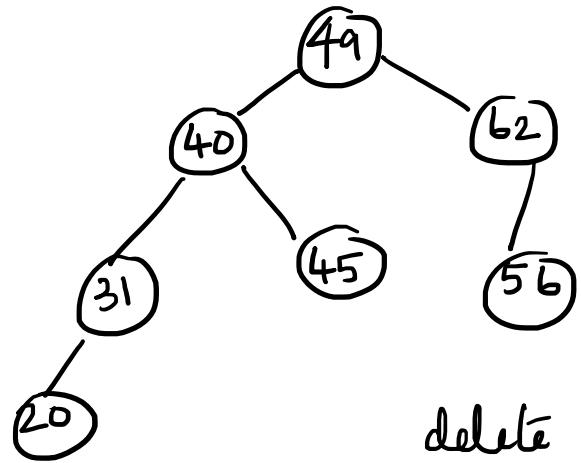
Example 4



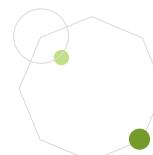
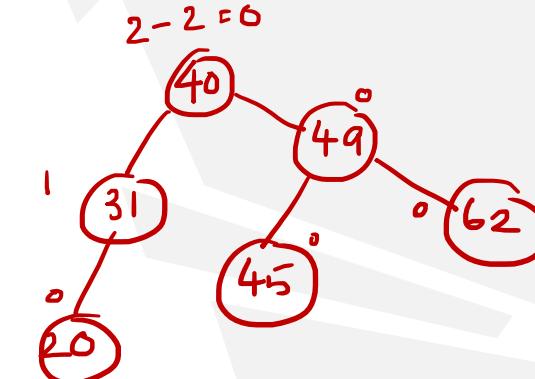
AVL tree



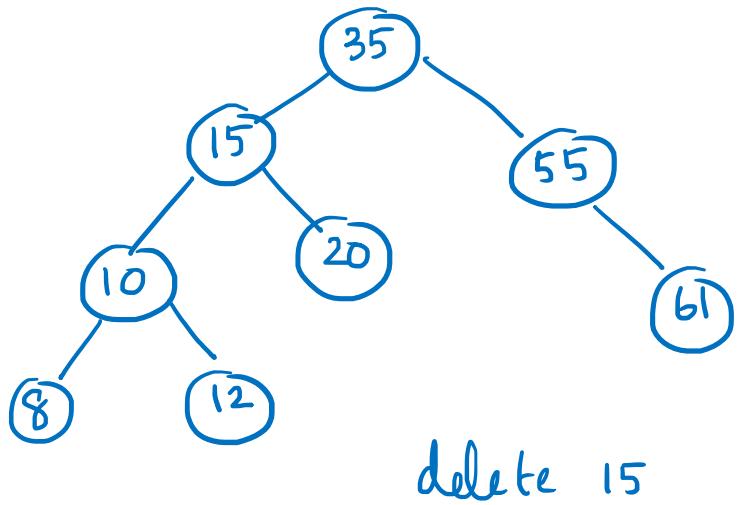
Example 5



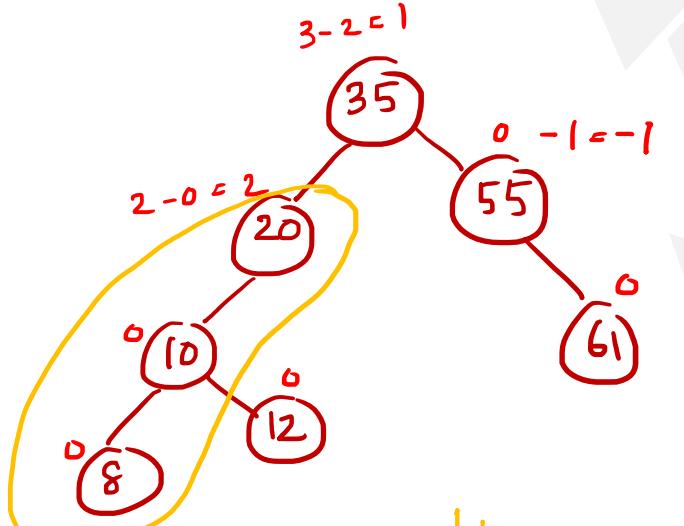
Right rotation



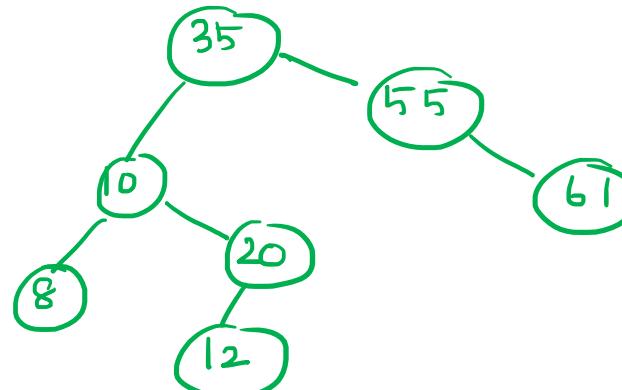
Example 6



delete 15



↓ Right rotation



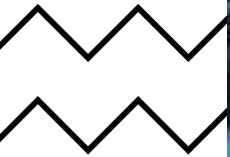
Resultant tree - AVL



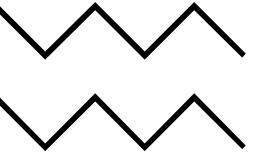


Thank You.

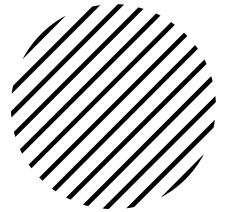




HEAPS.
ALSO, A TREE.



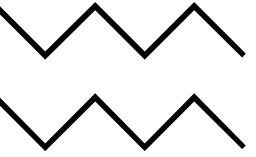
Terminology.



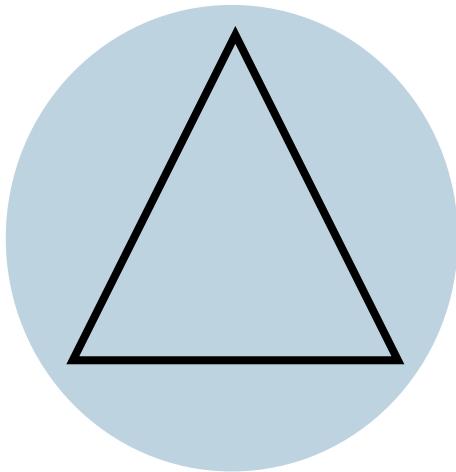
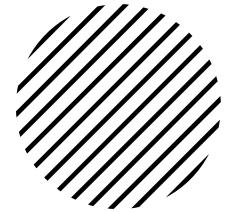
Full Binary Tree.

Complete Binary Tree.



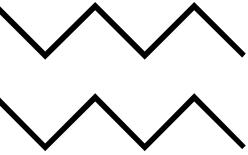


Full Binary Tree.

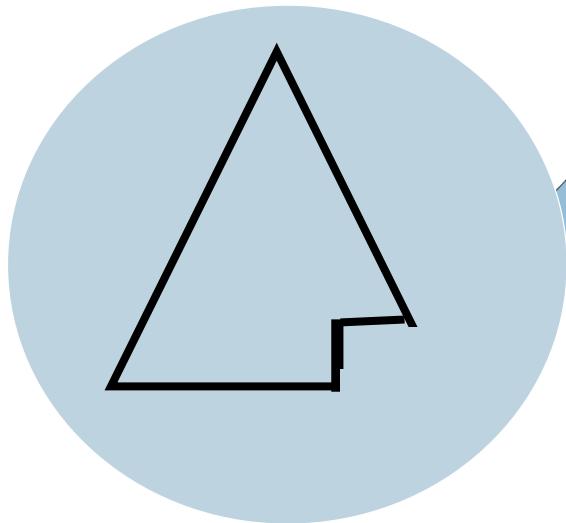
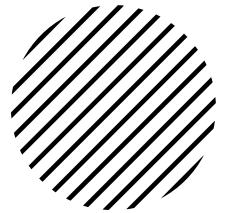


Every non-leaf node has two children
All the leaves are on the same level



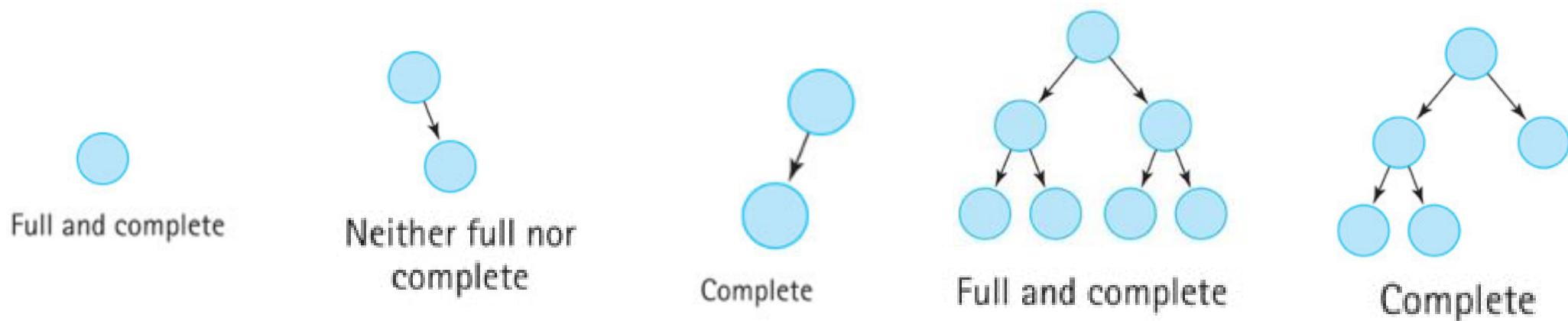


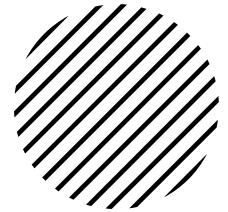
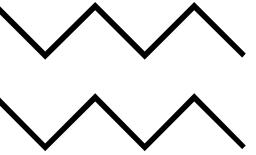
Complete Binary Tree.



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)

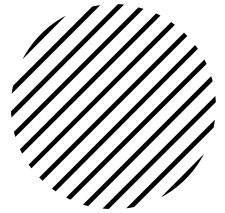
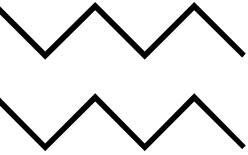




Heaps.

- ↳ 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.

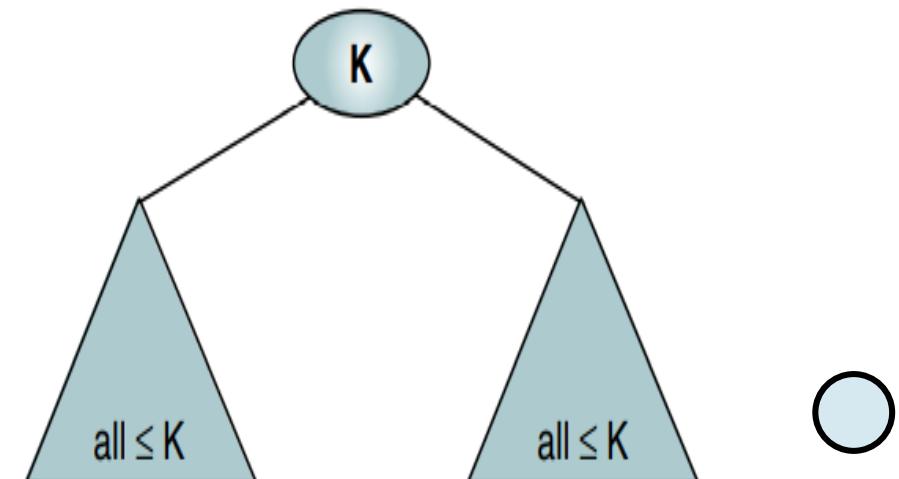


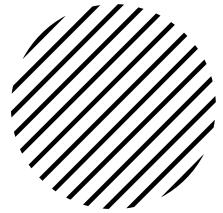
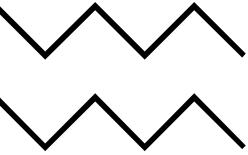


Basic Concepts.

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.

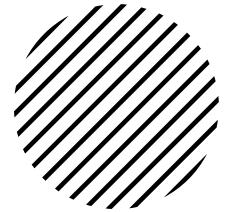
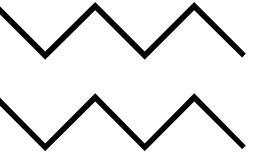




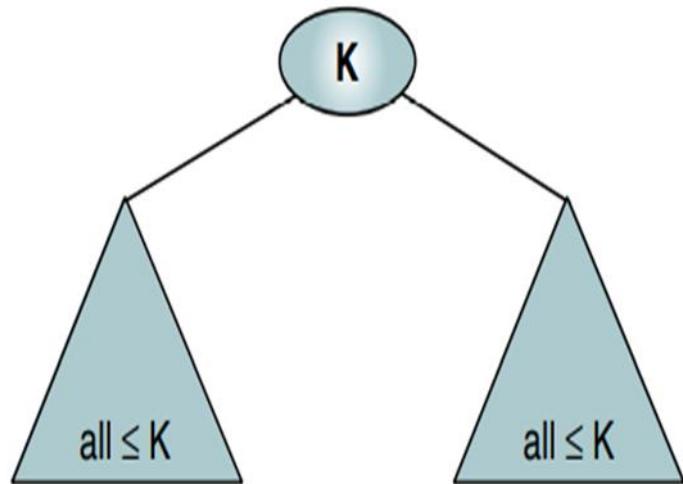
Basic Concepts - Definition.

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.



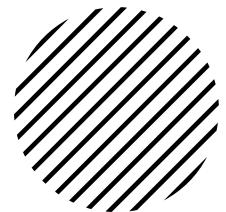
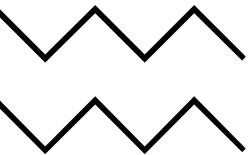


Heaps.

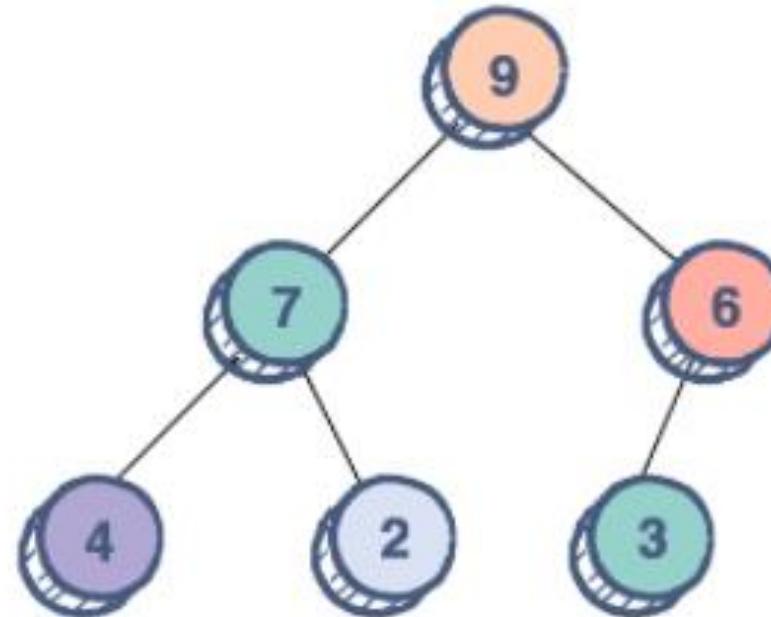


- ↳ Sometimes this structure is called a **max-heap**.
- ↳ The second property of a heap “key value is greater than the keys of the subtrees” can be reversed to create a min-heap.
- ↳ **Min heap:** *Key value in a node is less than the key values in all of its subtrees.*
- ↳ Generally speaking, whenever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



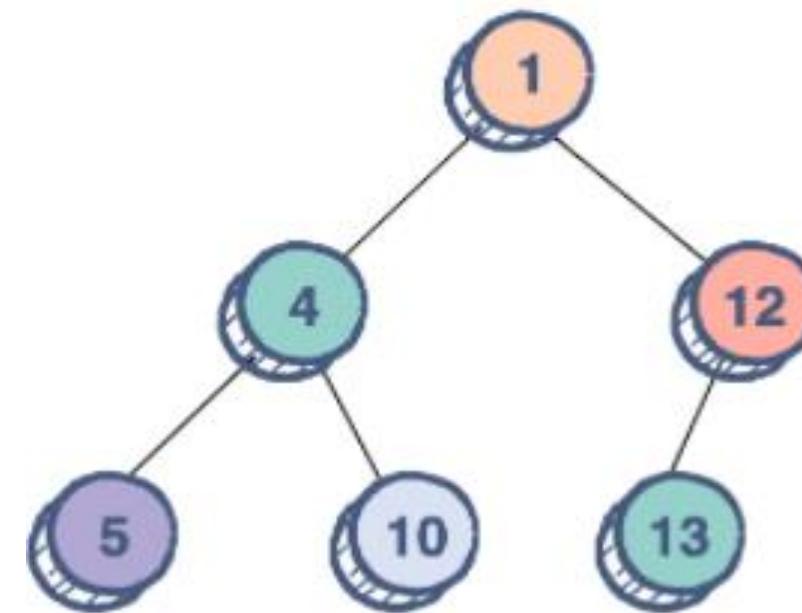


Max Heap and Min Heap.



If Node A has a child node B,
then,

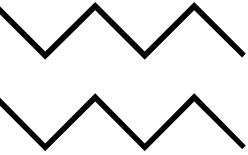
$$\text{key}(A) \geq \text{key}(B)$$



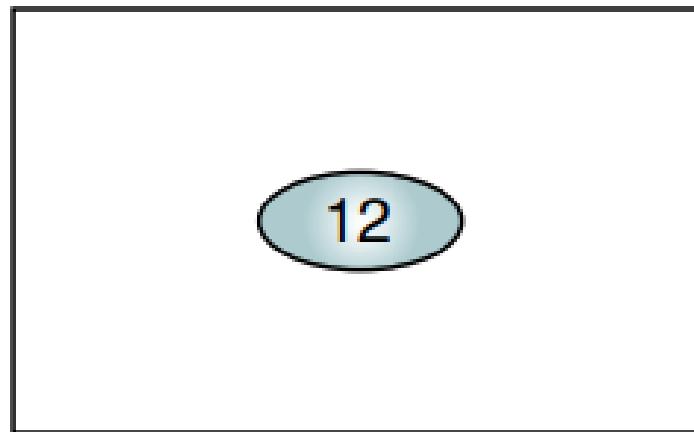
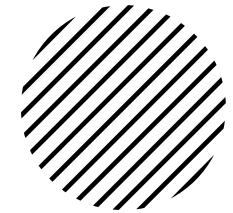
If Node A has a child node B,
then,

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

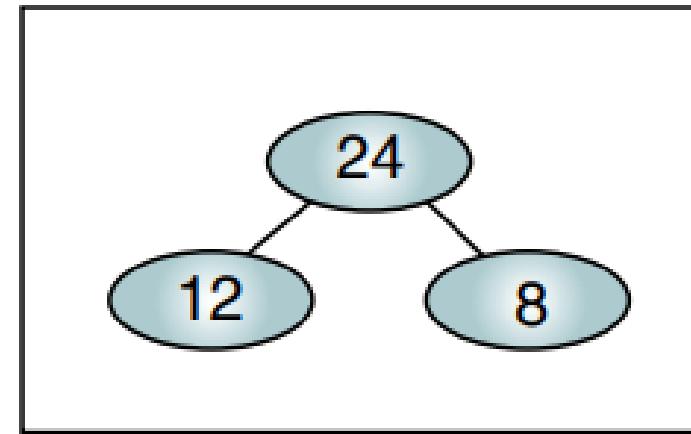




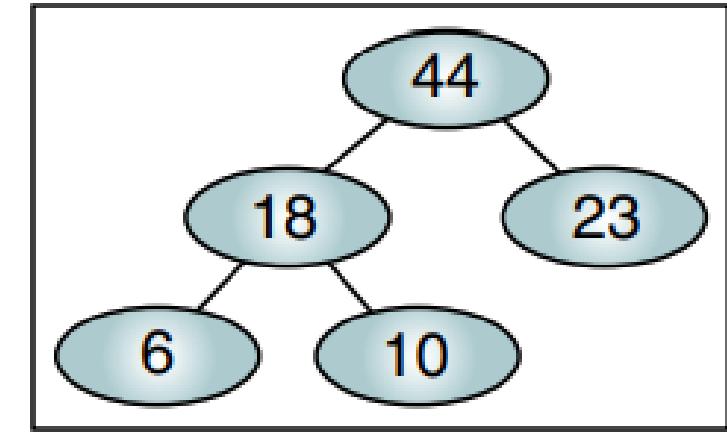
Examples of *Valid* Heap trees.



(a) Root-only heap

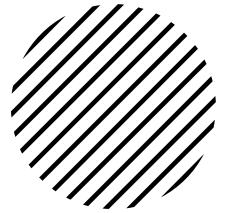
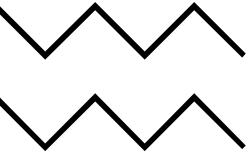


(b) Two-level heap

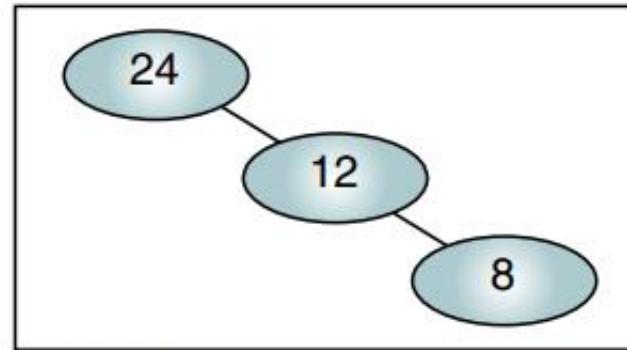


(c) Three-level heap

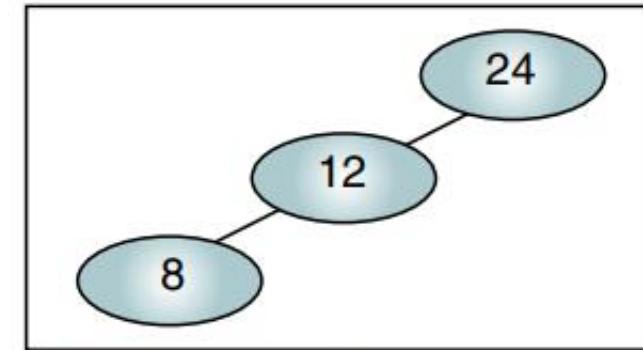




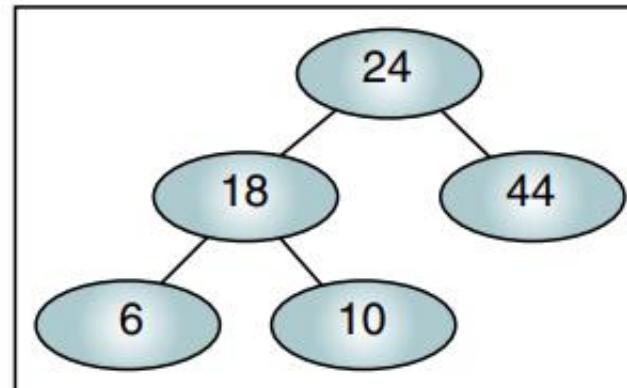
Examples of *Invalid* Heap trees.



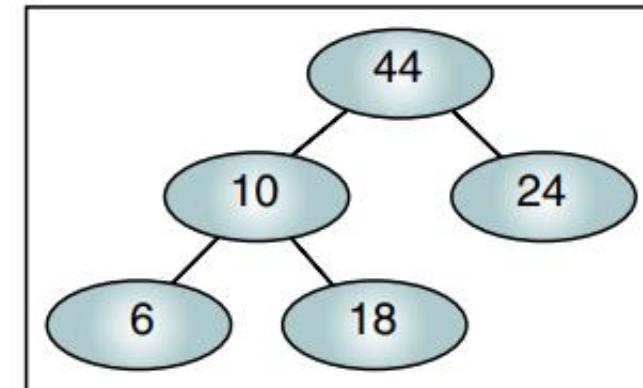
**(a) Not nearly complete
(rule 1)**



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

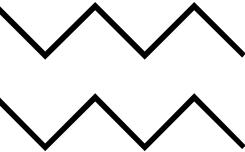


**(c) Root not largest
(rule 2)**



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



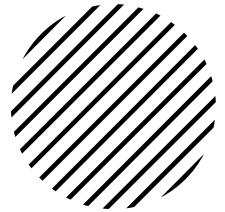
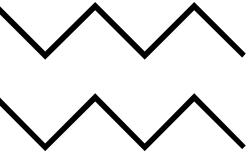


Some more terms.

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

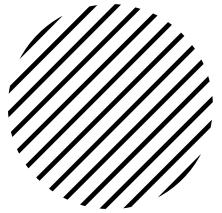
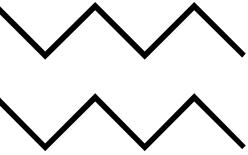
Heapify is the process of converting a binary tree into a Heap data structure



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

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.



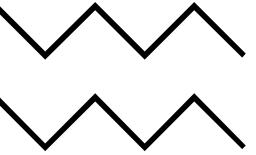
Heap Implementation.

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

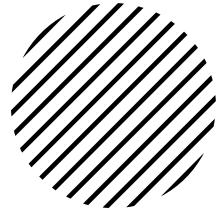
💡 How ?

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.



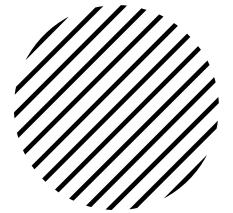
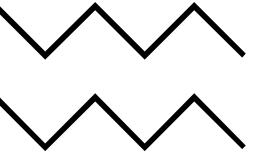


Heap Implementation Relationships.

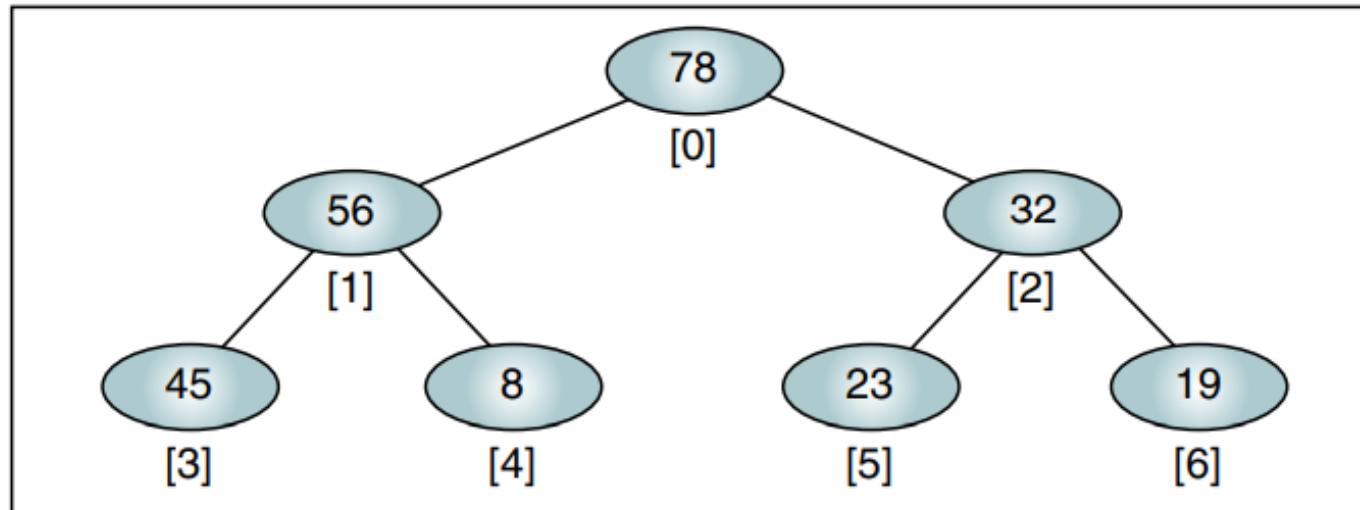


1. For a node located at index i , its children are found at:
 - a. Left child: $2i + 1$
 - b. Right child: $2i + 2$
2. The parent of a node located at index i is located at $\lfloor (i - 1) / 2 \rfloor$.
3. Given the index for a left child, j , its right sibling, if any, is found at $j + 1$. Conversely, 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.

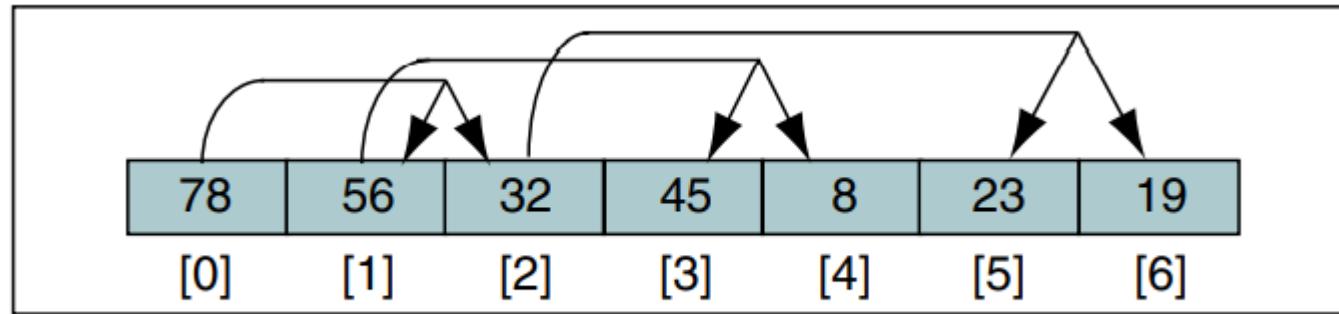




Heaps in Arrays.

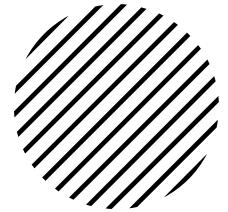
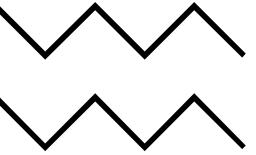


(a) Heap in its logical form



(b) Heap in an array

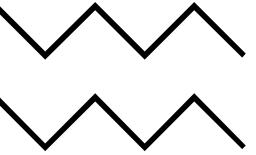




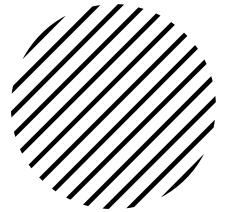
Heaps in Arrays.

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.



Insertion

Deletion





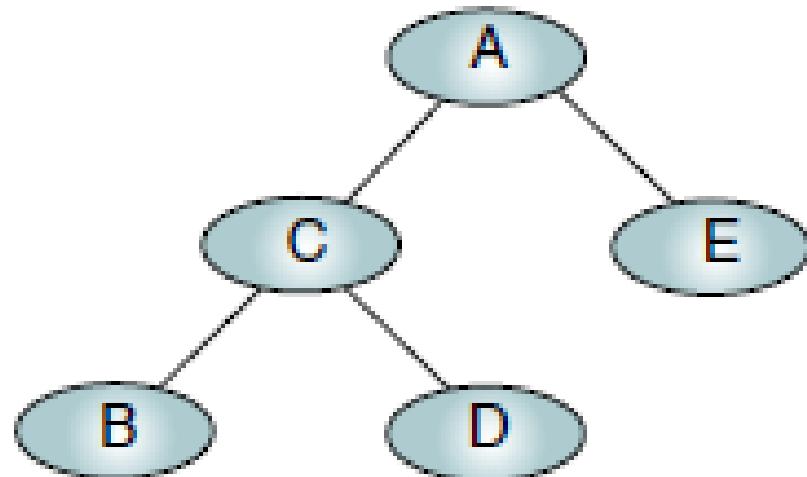
THREADED TREES.

THREADED TREES.

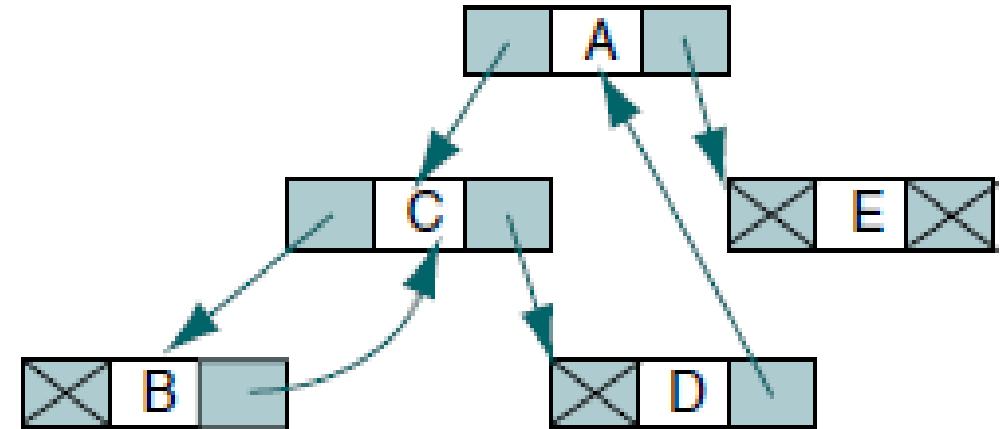
- Binary tree traversal algorithms are written using either recursion or programmer-written stacks. If the tree must be traversed frequently, **using stacks** rather than recursion may be more **efficient**.
- A third alternative is a ***threaded tree***. In a threaded tree, **null pointers are replaced 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

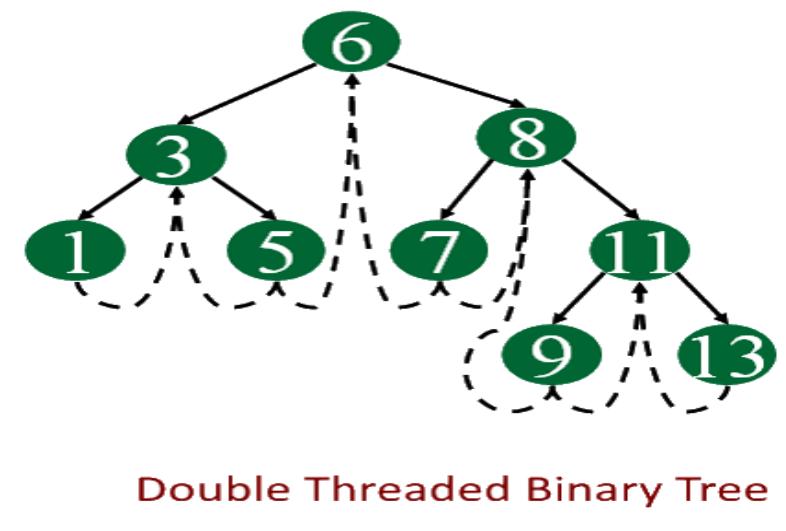
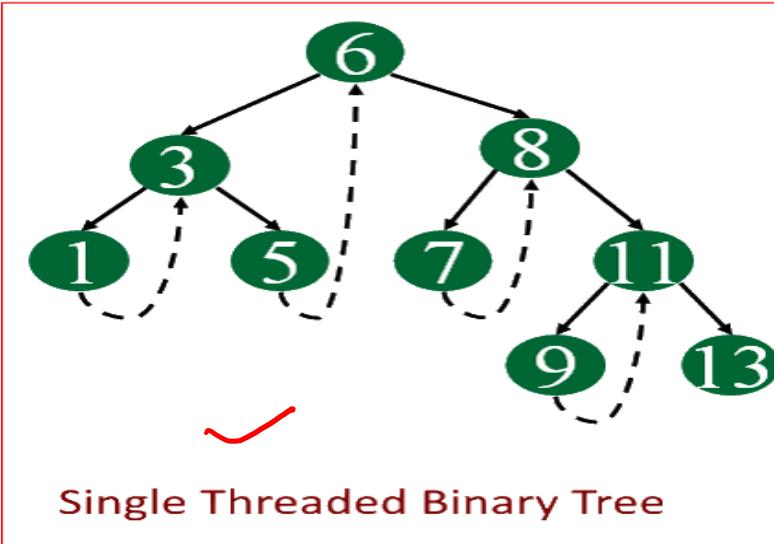
- A binary tree with n nodes has $n + 1$ null pointers

- 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

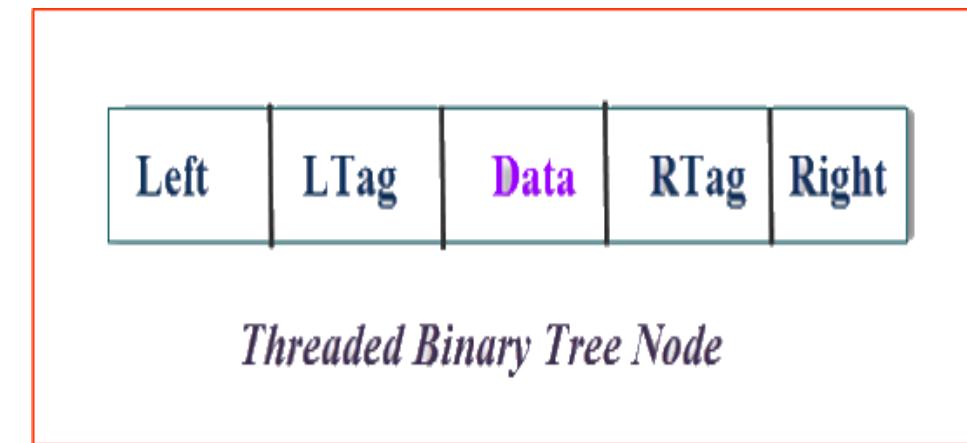


- Single threaded: Makes inorder traversal easier
- Double threaded: Makes inorder and postorder easier

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

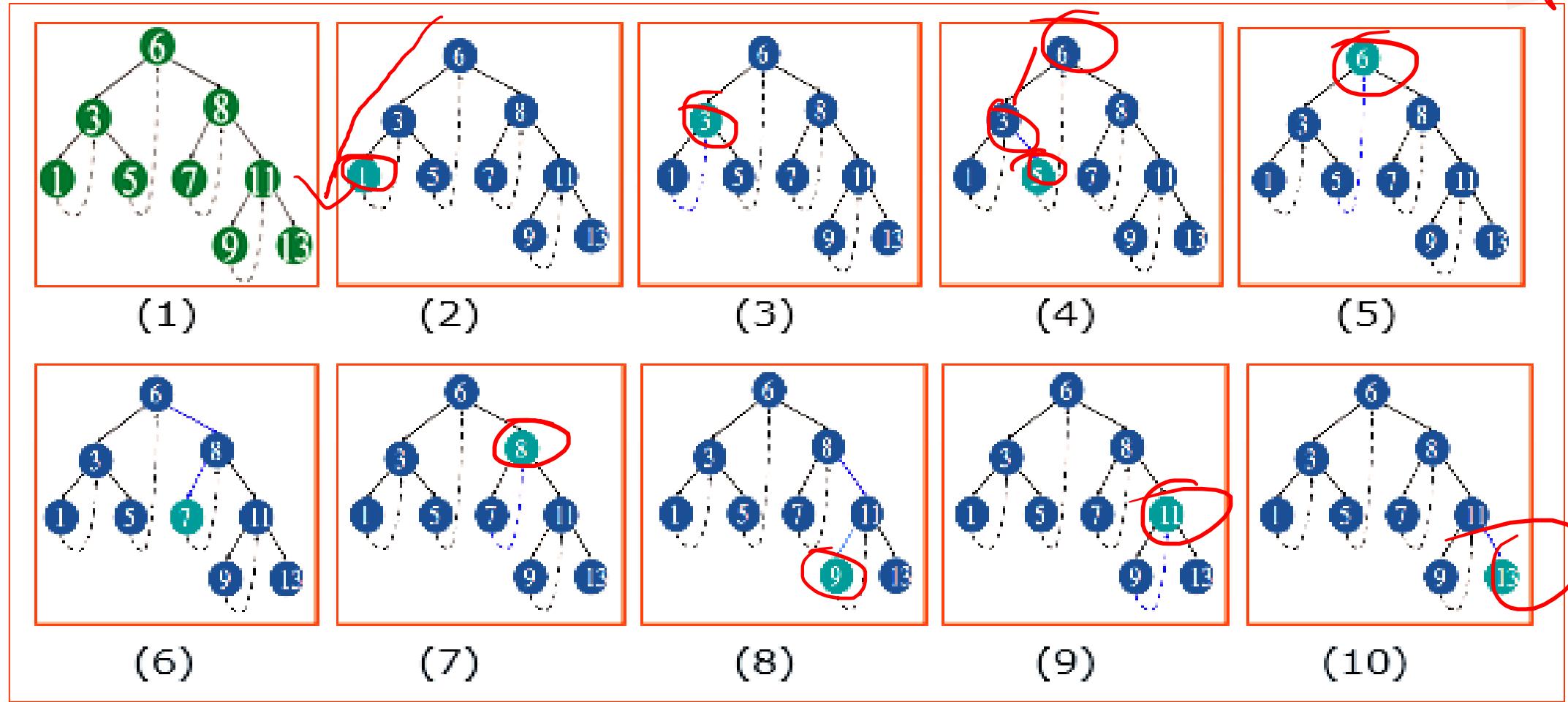
INORDER TRAVERSAL IS:-

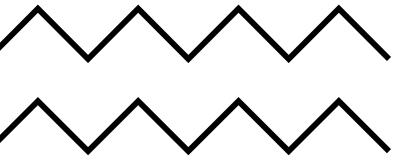
■ Threaded Binary Tree (Continued...)

- Example (for inorder traversal)



1 3 5 6 7 8 9 11 13

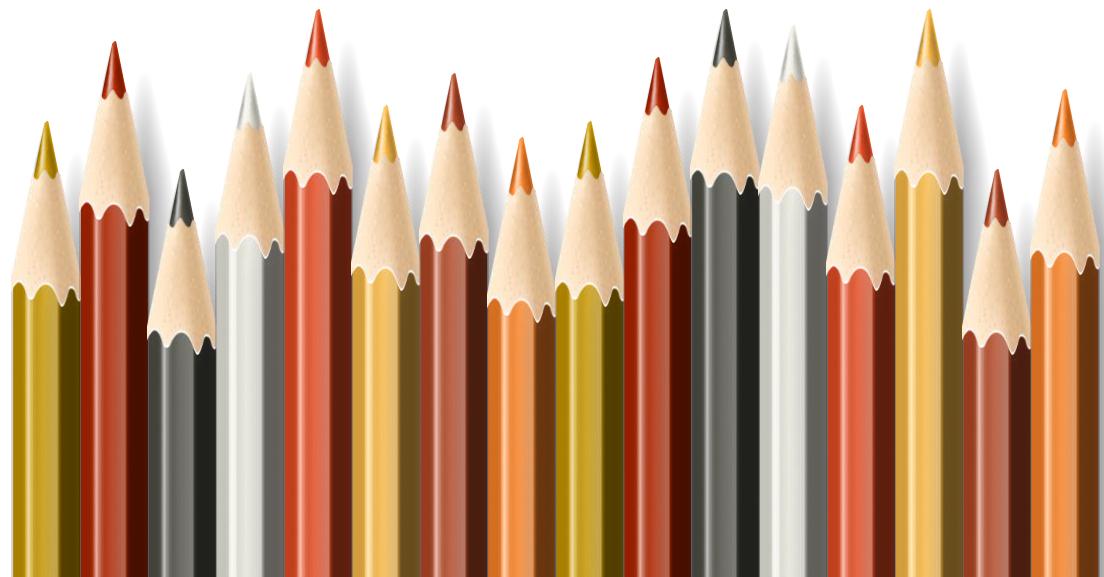




THANK
YOU

SORTING & SEARCHING.

Looking for data.



SORTING.

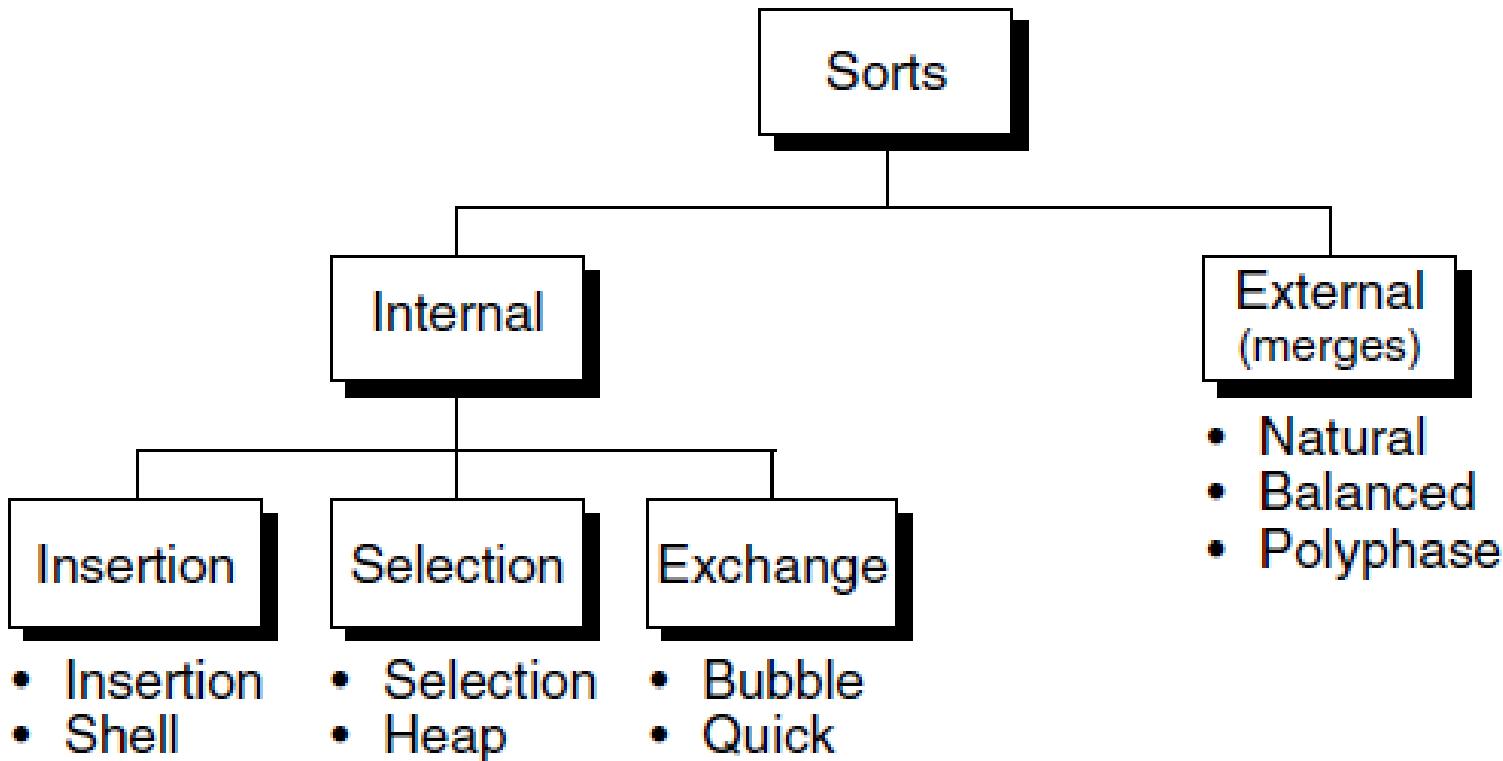
One of the most common data-processing applications.

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

(c) Unstable sort

output

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.

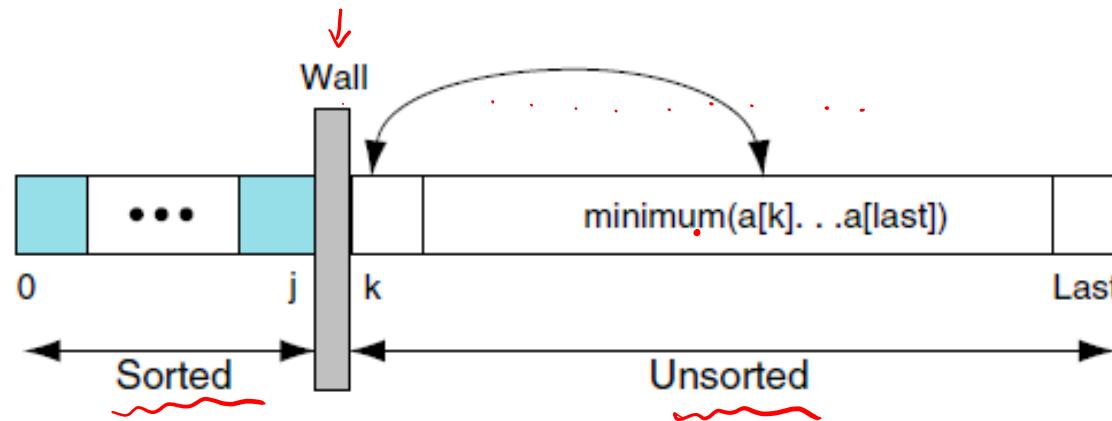
During the sorting process, the data are traversed many times. Each traversal of the data is referred to as a **sort pass**.

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.

- 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.
- If there is a list of n elements, therefore, $n - 1$ passes are needed to completely rearrange the data.



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

$n=6$

I Pass

$i \min j \ j < n \ a[j] < a[\min] \text{ exchange}$

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)

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

II Pass

1 1 2 $2 < 6$ $45 < 78$ T

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

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

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

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

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

wall

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

sorted

III Pass

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

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

4 5 $5 < 6$ $56 < 32$ F, 6 →

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

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)

$i++$

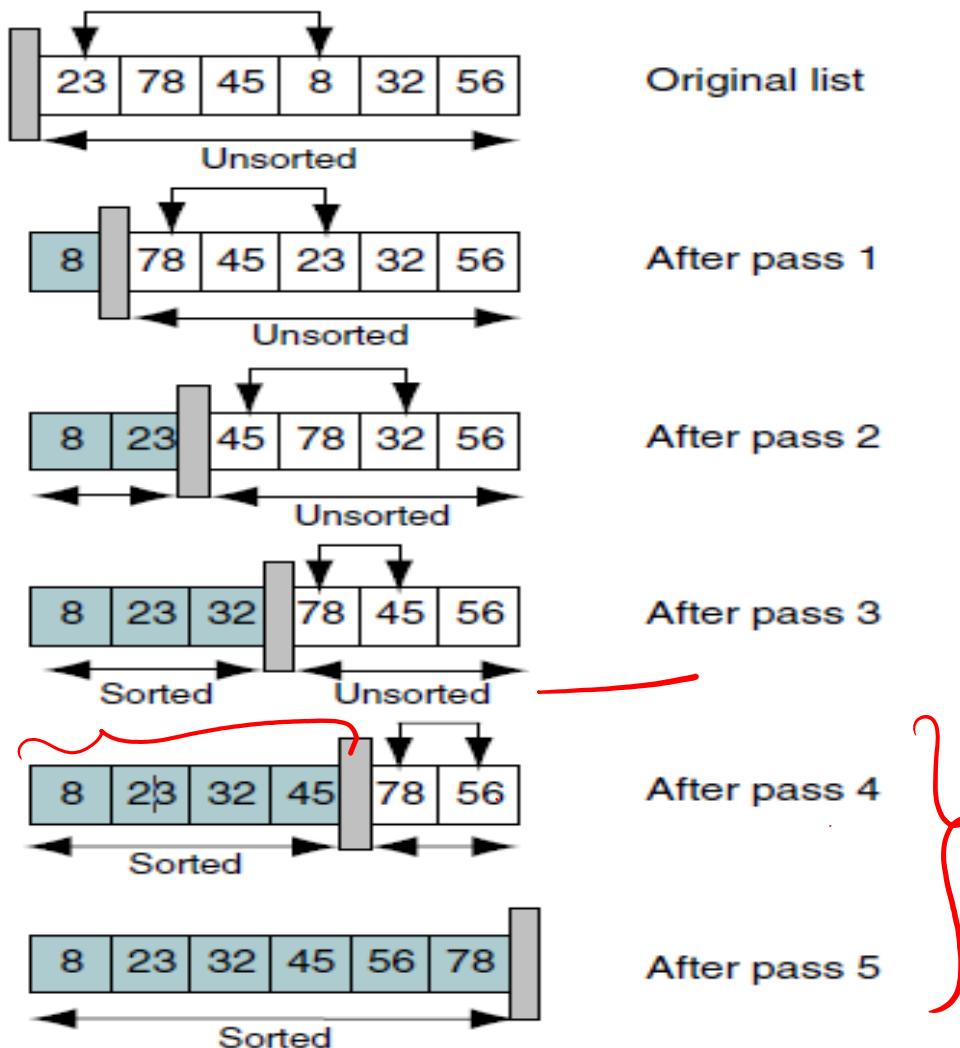
}

8 23 32 45 56 78

SELECTION SORT

5 3 4 1 2

Example 2

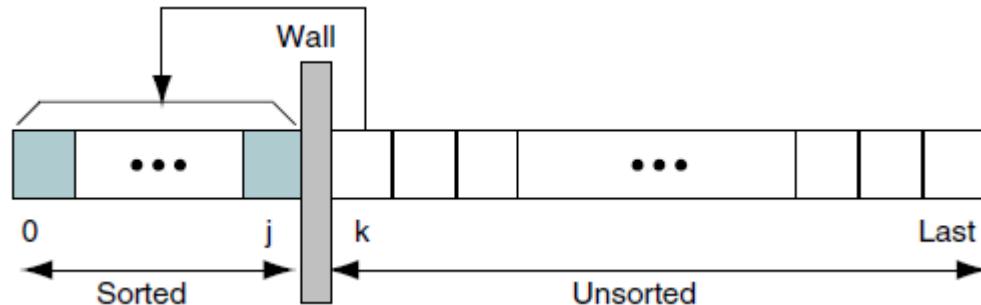


Selection Sort Efficiency.

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

INSERTION SORT.

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



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

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

$i \leq a[i] \neq t$ $j \geq 0$ & $t < a[j]$ $i=2$

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 $0 \geq 0$ & $8 < 23$ T

4 32 32 3 $3 \geq 0$ & $32 < 78$ T

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

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

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

0	1	2	3	4	5
23	45	78	8	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

SORTED

Algorithm

$i=1$ { 2nd element }

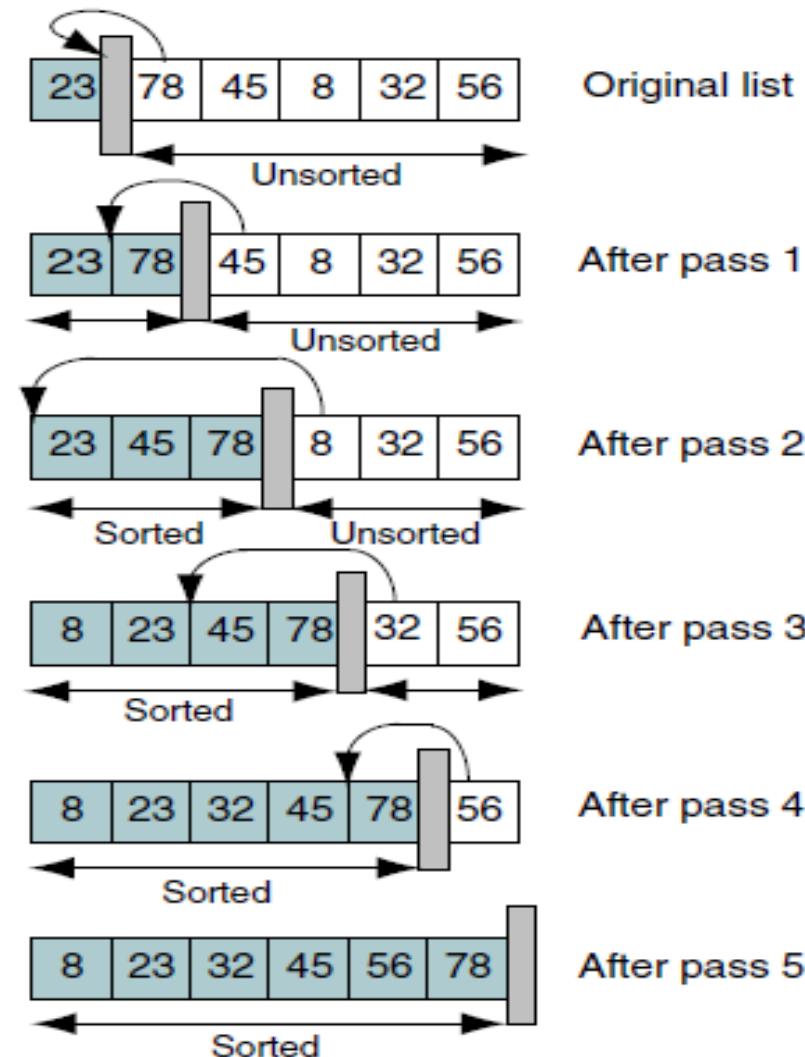
loop until last ele sorted
{ $t \leftarrow a[i]$
 $j \leftarrow i-1$

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

$a[j+1] = t$ // Always
 $i++$
}

8 23 32 45 56 78

INSERTION SORT.



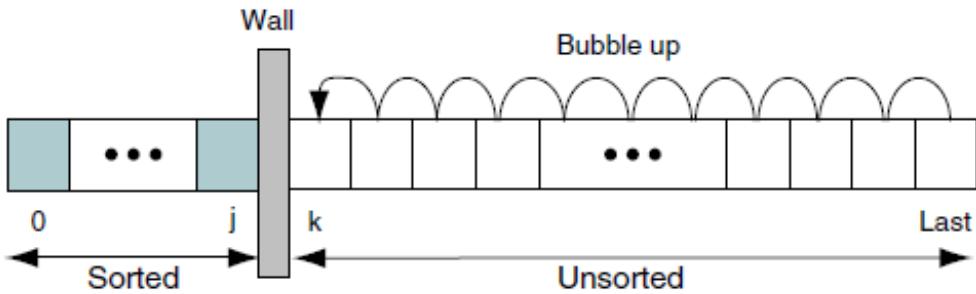
INSERTION SORT EFFICIENCY.

- The outer loop executes $n - 1$ times, from 1 through the last element in the list.
- 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.
- 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

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

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

BUBBLE SORT.

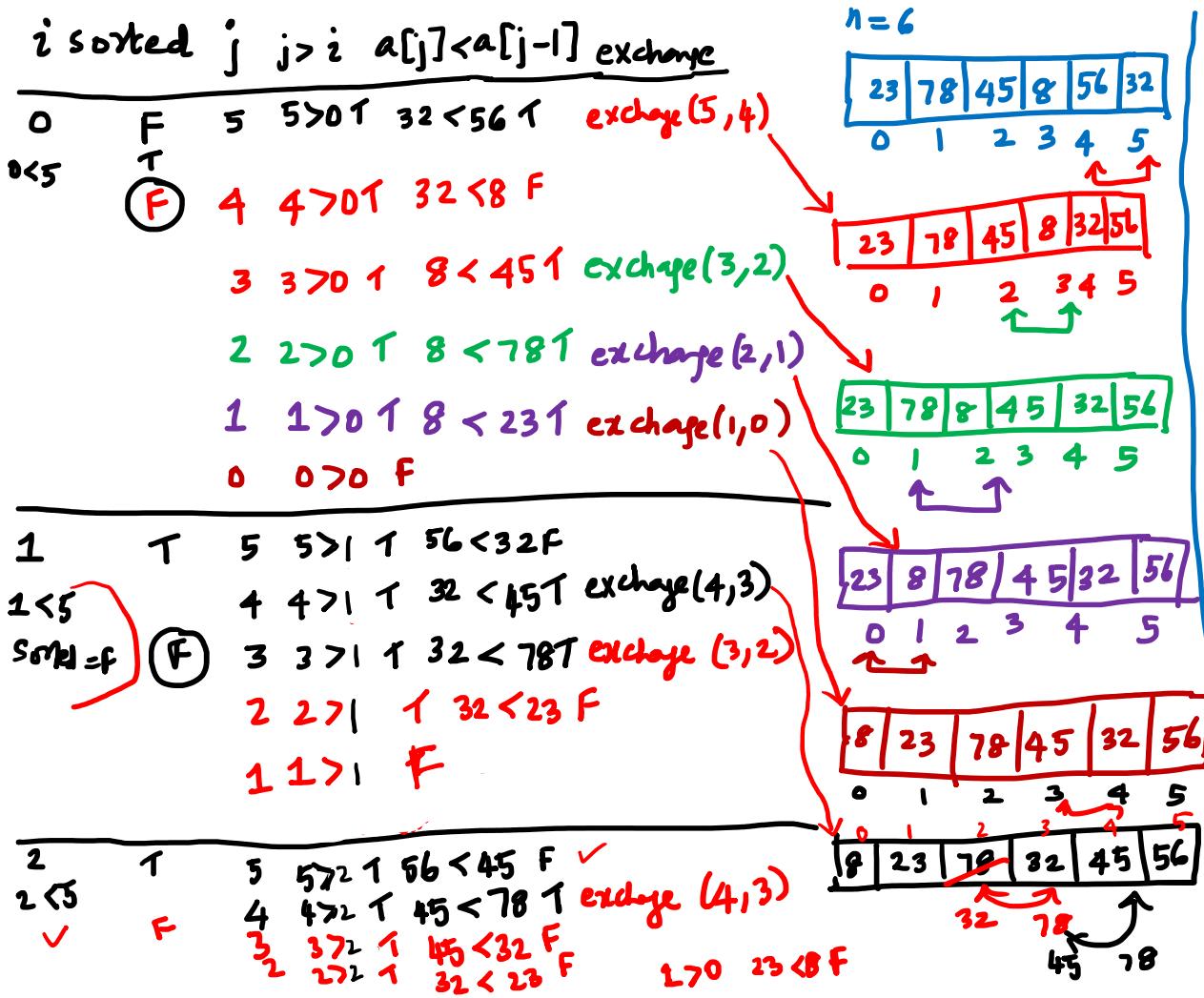


- The list is divided into two sublists: sorted and unsorted.
- The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist.
- After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the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

```
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
```



Algorithm

```

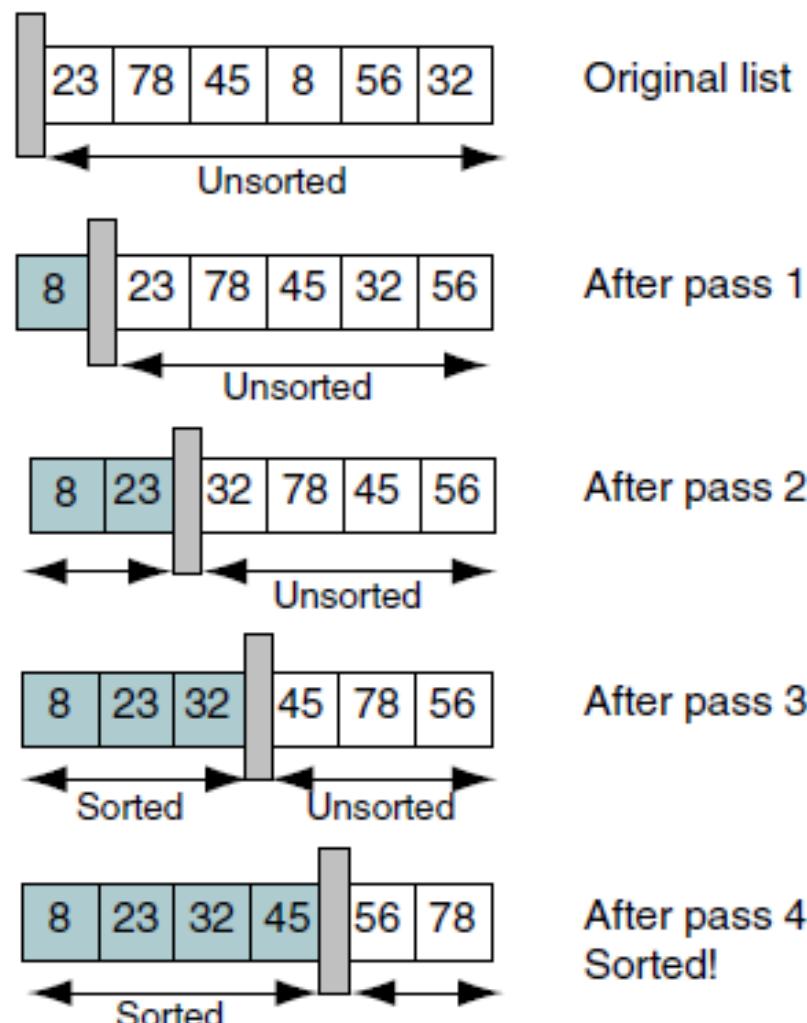
i < 0
sorted < false
Loop (i <= n-1 & & sorted == false)
{
    j < n-1
    sorted < true
    loop (j > i)
    {
        if ( $a[j] < a[j-1]$ )
        {
            sorted < false
            exchange( $a[j], a[j-1]$ )
        }
        j--
    }
    i++
}
    
```

$n=6$

$n-1 = 5$

BUBBLE SORT.

6 5 3 1 8 7 2 4



BUBBLE SORT EFFICIENCY.

- Uses two loops to sort the data.
- The outer loop tests two conditions: the current index and a sorted flag.
- 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 $n\left(\frac{n+1}{2}\right)$
- **The bubble sort efficiency is $O(n^2)$.**

QUICK SORT.

- ✖ In **Quick sort**, also an exchange sort method, developed by **C. A. R. Hoare in 1962**.
- ✖ **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.**
- ✖ **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.

- ✖ Also called **partition-exchange sort**.
- ✖ 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**.
- ✖ Pivot element can be **any element from the array**, it can be the **first** element, the **last** element or any **random** element.
- ✖ Approach is **recursive**.

LOGIC OF PARTITION.

- In the array **{52, 37, 63, 14, 17, 8, 6, 25}** , 25 is taken as **pivot**.
- **First pass:** **{6, 8, 17, 14, 25, 63, 37, 52}**
- 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**.
- Next, **{6 8 17 14}** and **{63 37 52}** are considered as two separate subarrays
- 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 it

Left partition $<$ pivot, Right partition \geq pivot

Repeat recursively for Left and Right partitions

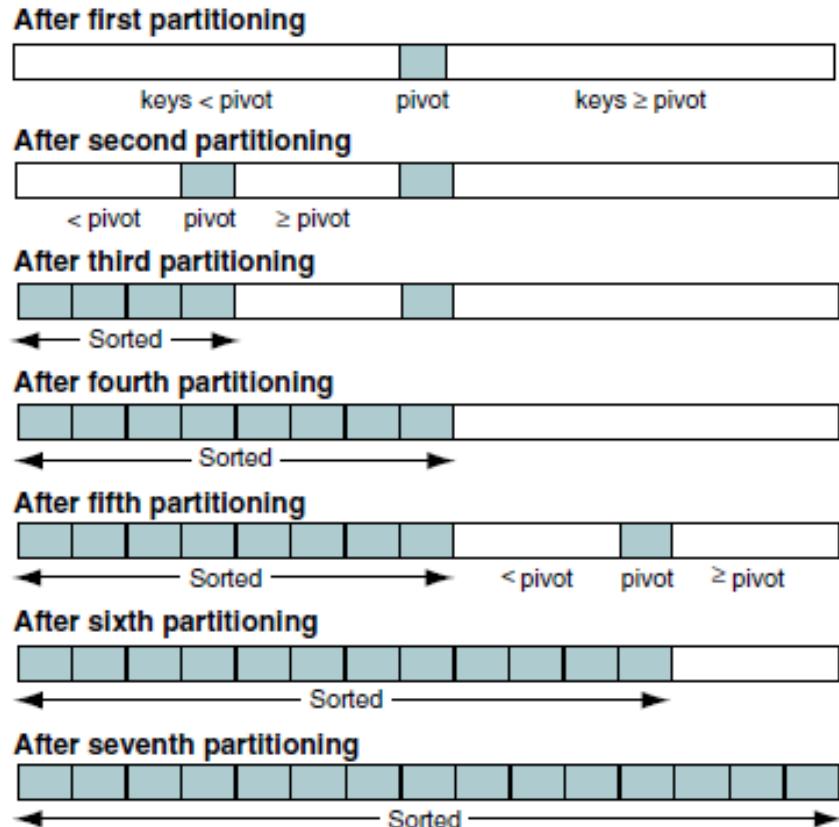
Can be used for finding the k^{th} smallest OR largest element in an array without completely sorting the array of size n .

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.

- ✖ Two Algorithms:
 - Quick Sort Recursive
 - Partition



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
---	---	---	---	---	---	----	----	----	----	----	----

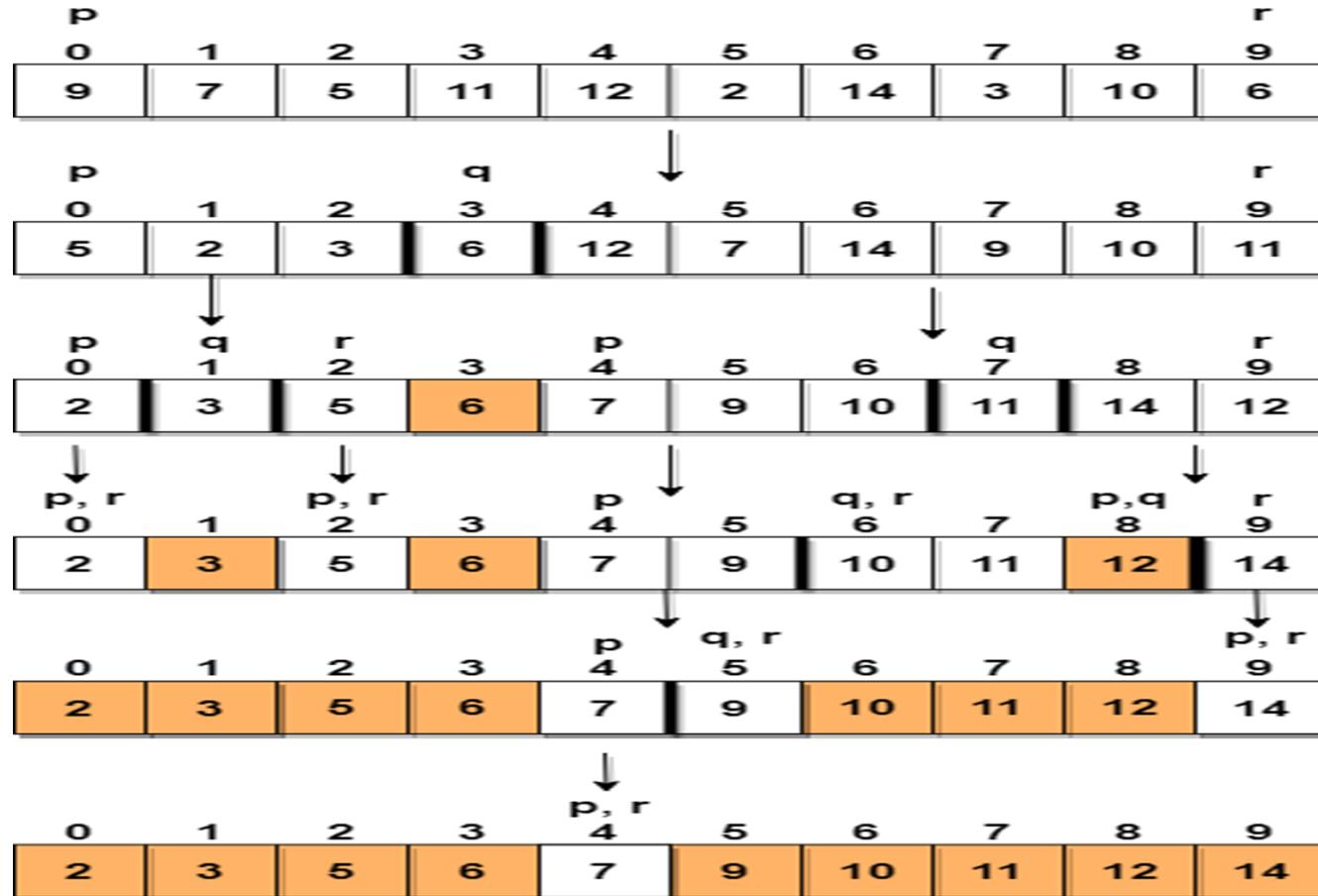
4	3	3	5	8	9	12	15	18	21	25	28
---	---	---	---	---	---	----	----	----	----	----	----

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 (Low < High) Then
 1. Set Mid = Partition (Array, Low, High)
 2. Quicksort (Array, Low, Mid – 1)
 3. Quicksort (Array, Mid + 1, 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 (KEY = 42)
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

while ($i < 9$ && $42 > a[i]$) ✓ $i = 2$
 $2 < 9 \&\& 42 > a[2]$ ✓ $i = 3$
 $3 < 9 \&\& 42 > a[3]$ ✗

while $42 < a[9]$ ✓ $j = 8$
 $42 < a[8]$ ✓ $j = 7$
 $42 < a[7]$ ✗

if $3 < 7$ ✓ swap($a[3], a[7]$)
i.e.; swap(98, 10)

42	37	11	10	36	72	65	98	88	78
0	1	2	3	4	5	6	7	8	9

while $3 < 9 \&\& 42 > a[3]$ ✓ $i = 4$
 $4 < 9 \&\& 42 > a[4]$ ✓ $i = 5$
 $5 < 9 \&\& 42 > a[5]$ ✗

while $42 < a[7]$ ✓ $j = 6$
 $42 < a[6]$ ✓ $j = 5$
 $42 < a[5]$ ✓ $j = 4$
 $42 < a[4]$ ✗

$n = 10$, $low = 0$, $high = 9$
key = $a[0] = 42$, $i = 1$, $j = 9$

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 ($a, 5, 9$)

Set Key = Array[low], I = Low + 1, J = High
Repeat Steps A through C
A. while ($I < High \& \& Key \geq Array[I]$)
i++
B. while ($Key < Array[j]$) j -
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

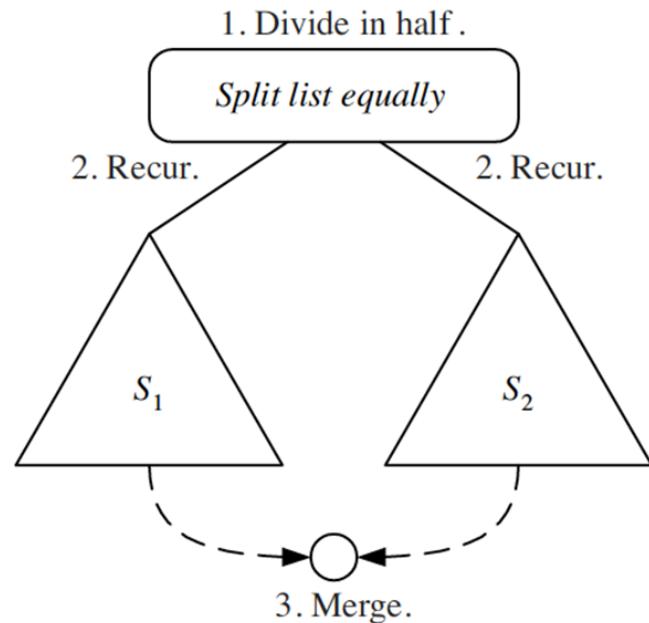
Set Mid = Partition (Array, Low, High)
Quicksort (Array, Low, Mid - 1)
Quicksort (Array, Mid + 1, High)
End

QUICK SORT EFFICIENCY.

- Quick sort is considered the best general-purpose sort known today.
- 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.
- The number of comparisons to sort the left subarray is $f(i)$, and the number of comparisons to sort the right subarray is $f(\underline{n - 1 - i})$, where i can be between 0 to $n - 1$.
- To continuously divide the array into subarrays, n comparisons needed.
- The efficiency of quick sort is **O($n \log n$)**.

DIVIDE & CONQUER.

- Divide-and conquer is a general algorithm design paradigm.
- **Divide:** divide the input data S in two disjoint subsets S_1 and S_2 .
- **Recur:** solve the subproblems associated with S_1 and S_2 .
- **Conquer:** combine the solutions for S_1 and S_2 into a solution for S .
- Base case for recursion are subproblems of size 0 or 1.



MERGE SORT.

- ✖ Invented by ***John von Neumann*** in 1945, example of Divide & Conquer.
- ✖ Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.
- ✖ **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.
- ✖ Often implemented **recursively**
- ✖ Runtime: **O(n log n).**

MERGE SORT ALGORITHM.

Algorithm **mergeSort(S)**

Input: Sequence S with n elements

Output: Sequence S sorted

if $S.size() > 1$

$(S1, S2) \leftarrow \text{partition}(S, n/2)$

mergeSort(S1)

mergeSort(S2)

$S \leftarrow \text{merge}(S1, S2)$

Algorithm **merge(S1, S2, S)**

Input: Two arrays $S1$ & $S2$ of size $n1$ and $n2$ sorted in non decreasing order and an empty array S of size at least $(n1+n2)$

Output: S containing elements from $S1$ & $S2$ in sorted order.

$i \leftarrow 1$

$j \leftarrow 1$

while ($i \leq n1$) and ($j \leq n2$)

 if $S1[i] \leq S2[j]$ then

$S[i+j-1] \leftarrow S1[i]$

$i \leftarrow i + 1$

 else

$S[i+j-1] \leftarrow S2[j]$

$j \leftarrow j + 1$

while ($(i \leq n1)$) do

$S[i+j-1] \leftarrow S1[i]$

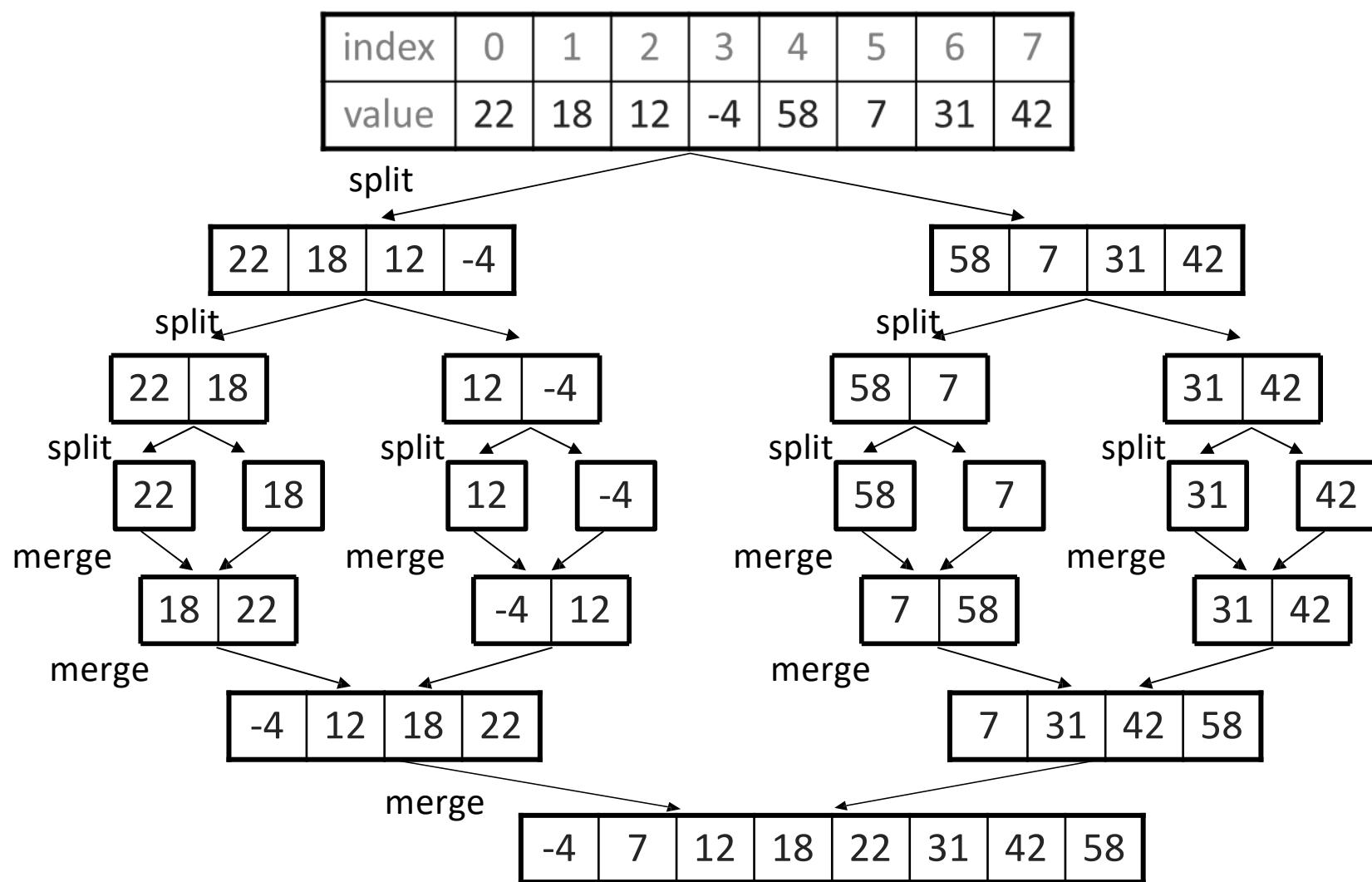
$i \leftarrow i + 1$

while ($(j \leq n2)$)

$S[i+j-1] \leftarrow S2[j]$

$j \leftarrow j + 1$

Merge Sort example.



Merge Sort example

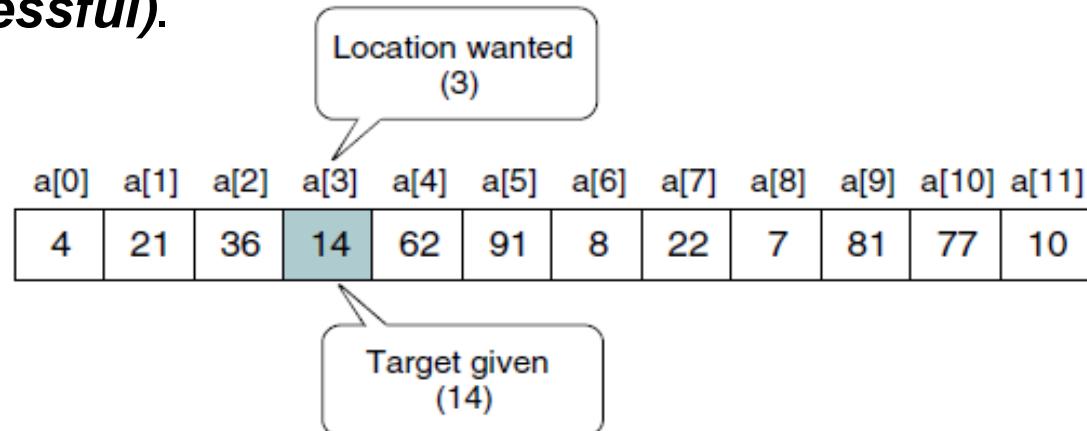
Subarrays	Next include	Merged array
0 1 2 3 14 32 67 76 i1	0 1 2 3 23 41 58 85 i2	14 from left 14
14 32 67 76 i1	23 41 58 85 i2	23 from right 14 23
14 32 67 76 i1	23 41 58 85 i2	32 from left 14 23 32
14 32 67 76 i1	23 41 58 85 i2	41 from right 14 23 32 41
14 32 67 76 i1	23 41 58 85 i2	58 from right 14 23 32 41 58
14 32 67 76 i1	23 41 58 85 i2	67 from left 14 23 32 41 58 67
14 32 67 76 i1	23 41 58 85 i2	76 from left 14 23 32 41 58 67 76
14 32 67 76 i1	23 41 58 85 i2	85 from right 14 23 32 41 58 67 76 85

SEARCHING.

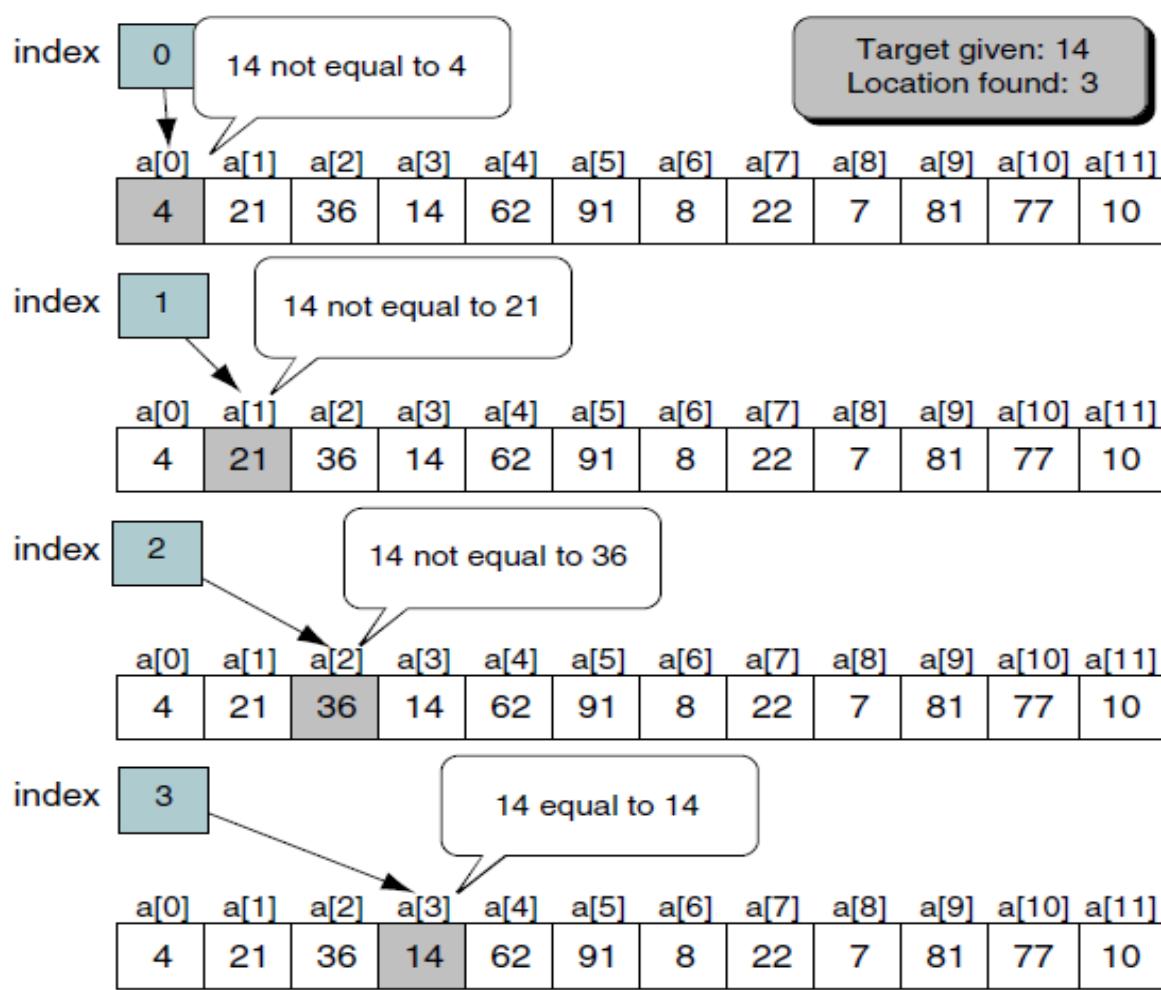
- ✖ One MORE common and time-consuming operations in computer science is searching
- ✖ The process used to find the location of a target among a list of objects.
- ✖ The two basic search algorithms:
- ✖ **Sequential search** including three interesting variations and,
- ✖ **Binary search.**

SEQUENTIAL SEARCH.

- Used whenever the list is not ordered.
- Generally, technique used only for small lists or lists that are not searched often.
- Process: Start searching for the target at the beginning of the list and continue until target found or it is not in the list.
- This approach has two possibilities: Find element (***successful***) or reach end of list (***unsuccessful***).

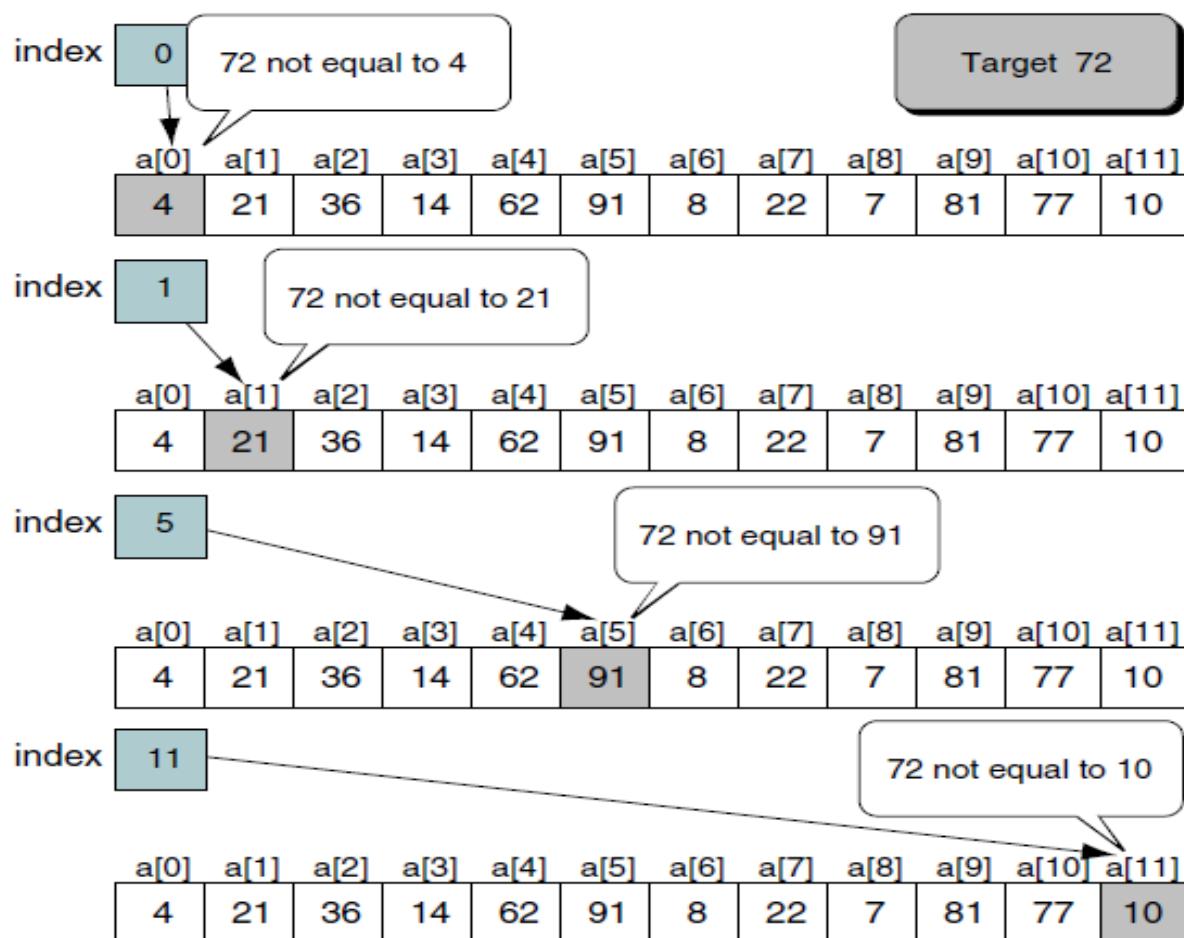


Sequential Search Example for Successful search.



Successful Search of an Unordered List

Sequential Search Example for Unsuccessful search.



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 i < n & key != a[i]

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

Efficiency of the sequential search is O(n).

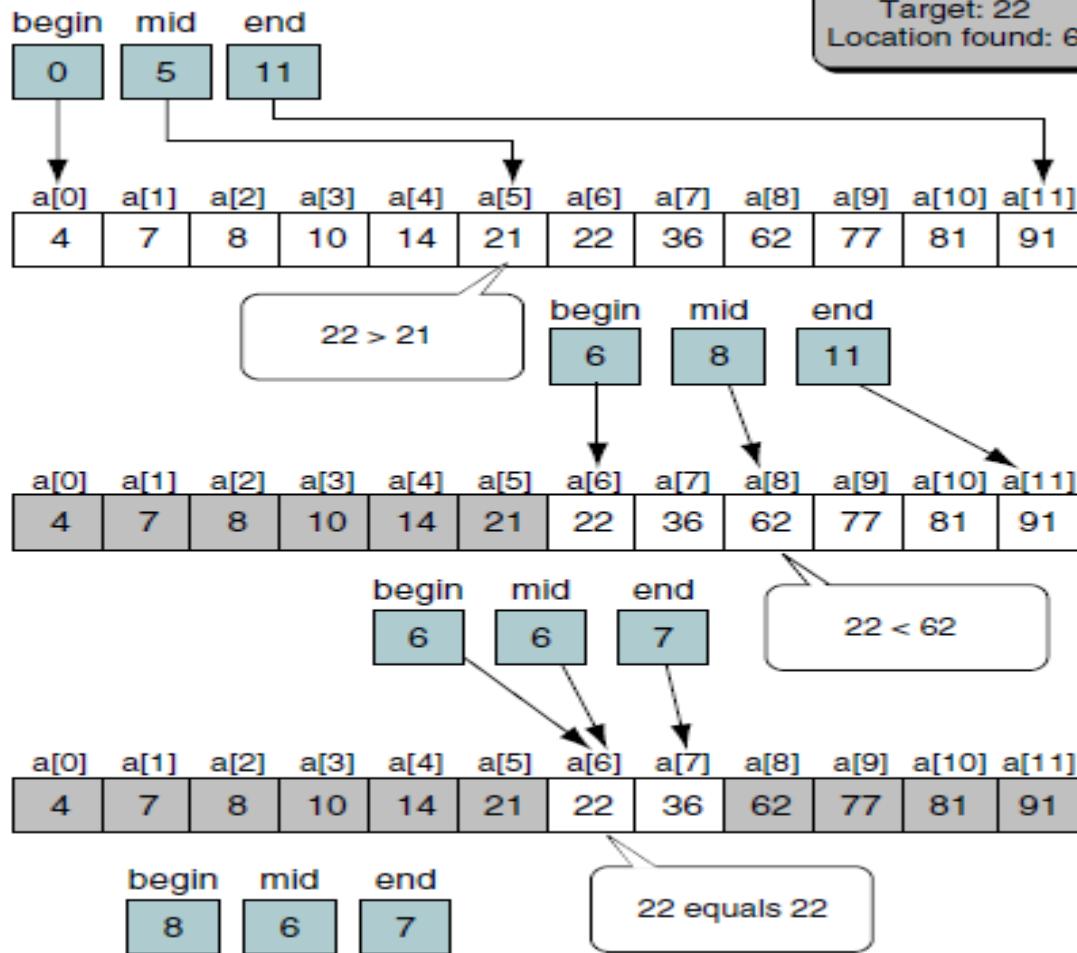
BINARY SEARCH.

- ✖ Sequential search algorithm is very slow. If an array of 1000 elements, exists, 1000 comparisons are made in worst case.
- ✖ If the array is not sorted, the sequential search is the only solution.
- ✖ However, if the array is sorted, we can use a more efficient algorithm called ***binary search***.
- ✖ Generally speaking, Binary search used whenever the list starts to become large.

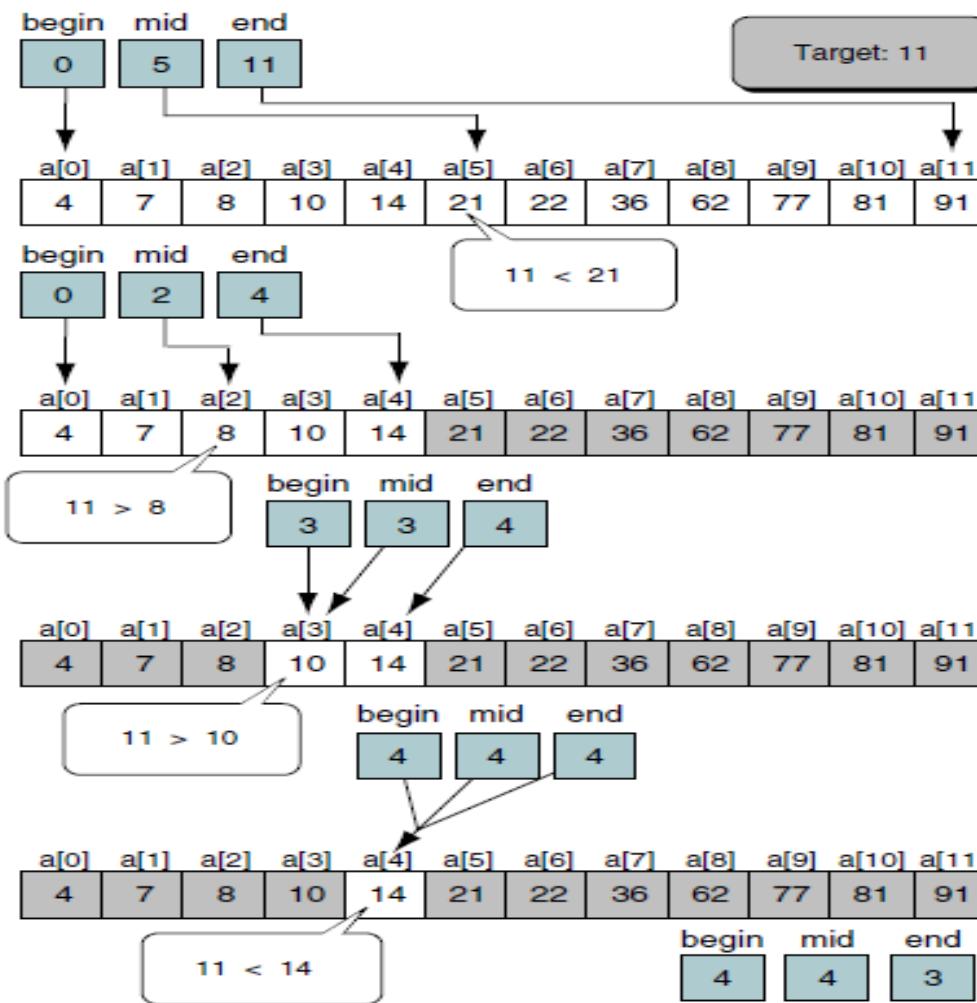
BINARY SEARCH.

- 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.***
- If **target in first half**, there is NO need to check the second half.
- If **target in second half**, NO need to test the first half.
- In other words, half the list is eliminated from further consideration with just one comparison.
- This process repeated, eliminating half of the remaining list with each test, until target is found or does not exist in the list.
- 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.



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

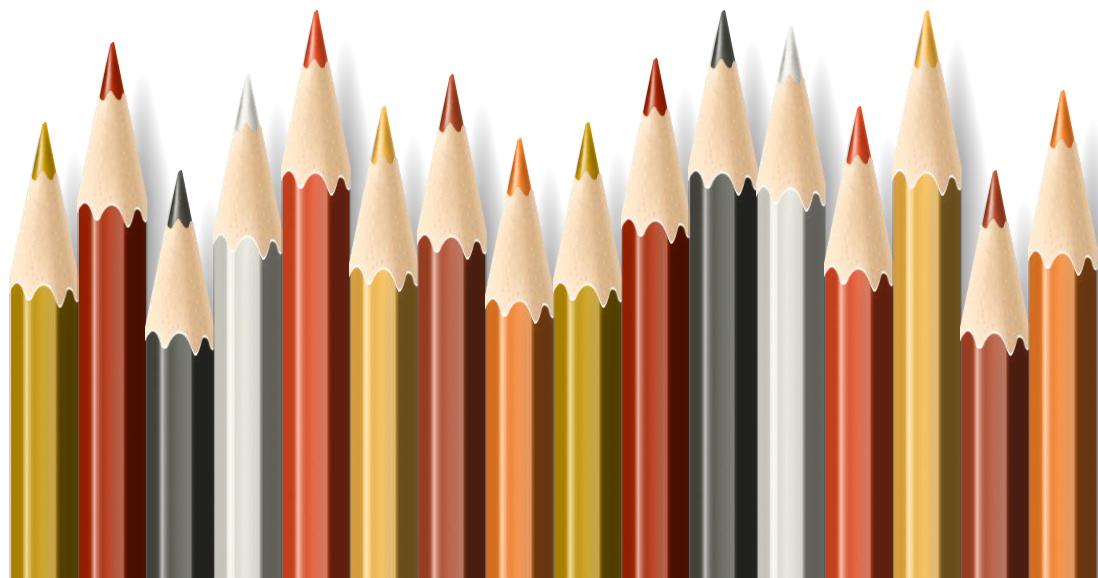
```

Efficiency of the
binary search is
O(log n).



X End for today.

THE END FOR TODAY



MULTIWAY TREES. M-WAY SEARCH TREES.



Types of Trees

Binary Trees

Binary
Search Trees

Heap Trees

Multiway
Search Trees

B 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.

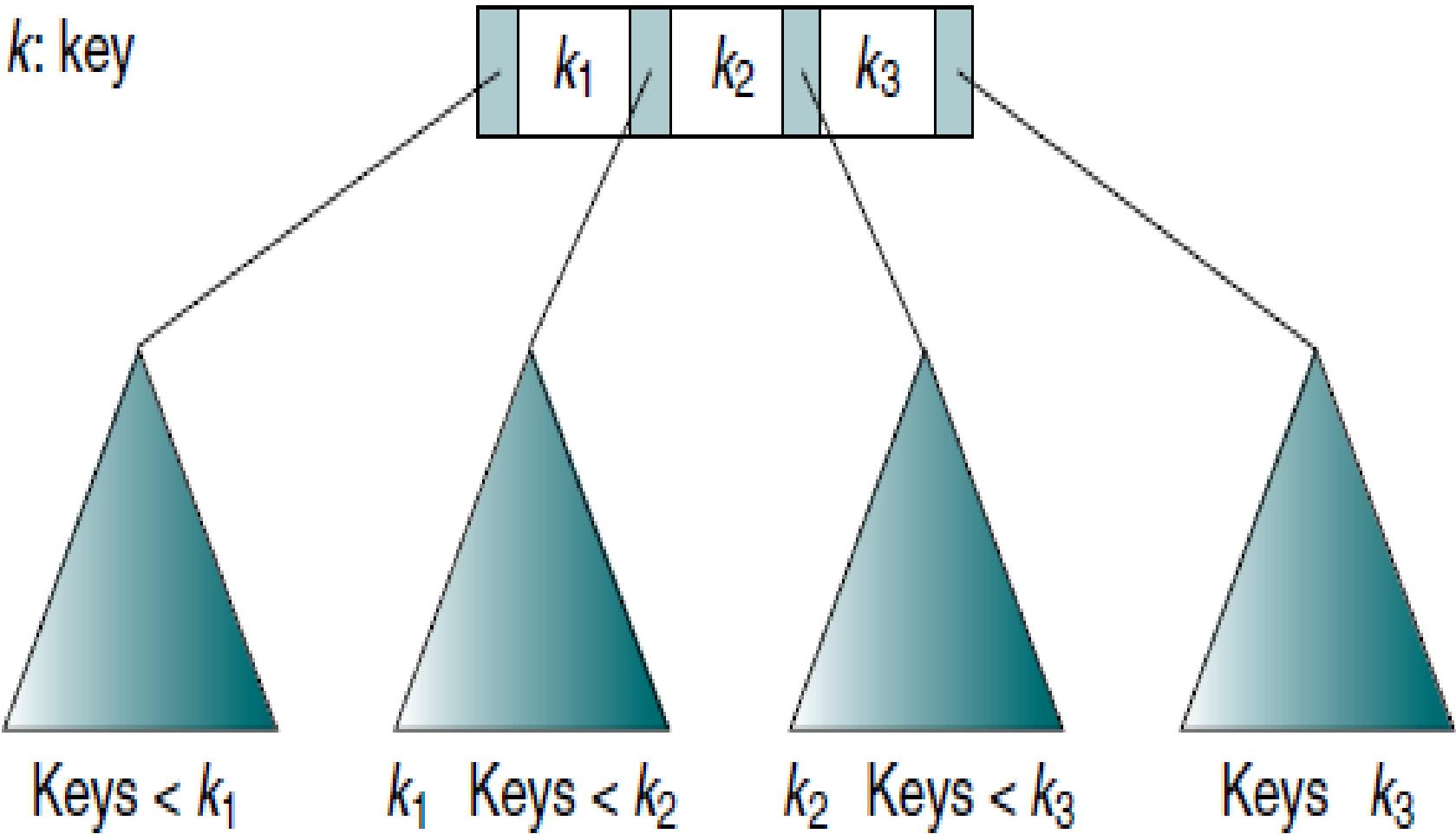
A node with
 $k < m$ subtrees
contains
k subtrees
and
 $k - 1$
data entries.

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.

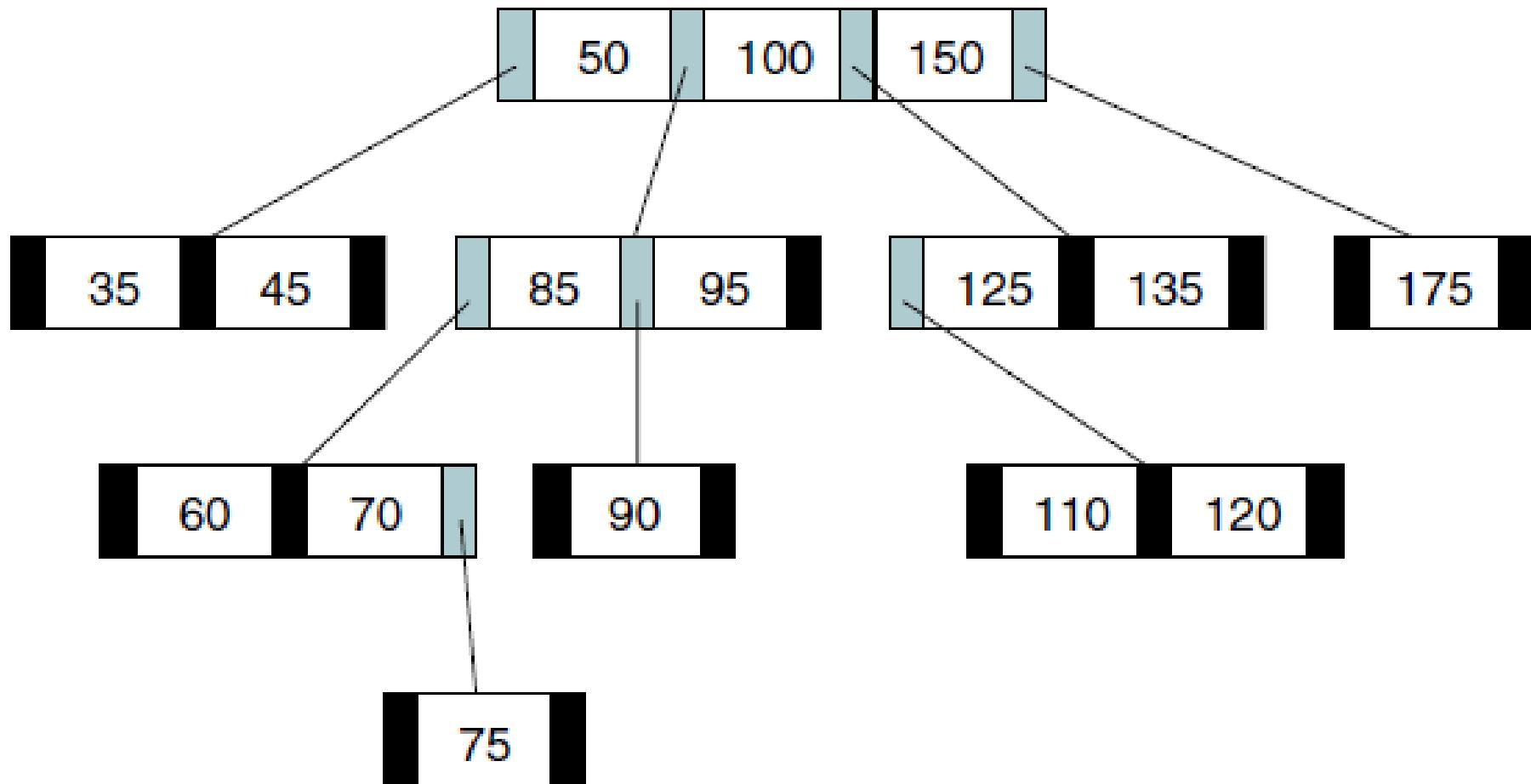
Keys of the data entries are ordered
 $\text{key}_1 \leq \text{key}_2 \leq \dots \leq \text{key}_k$.

All subtrees are themselves multiway trees.

k: key

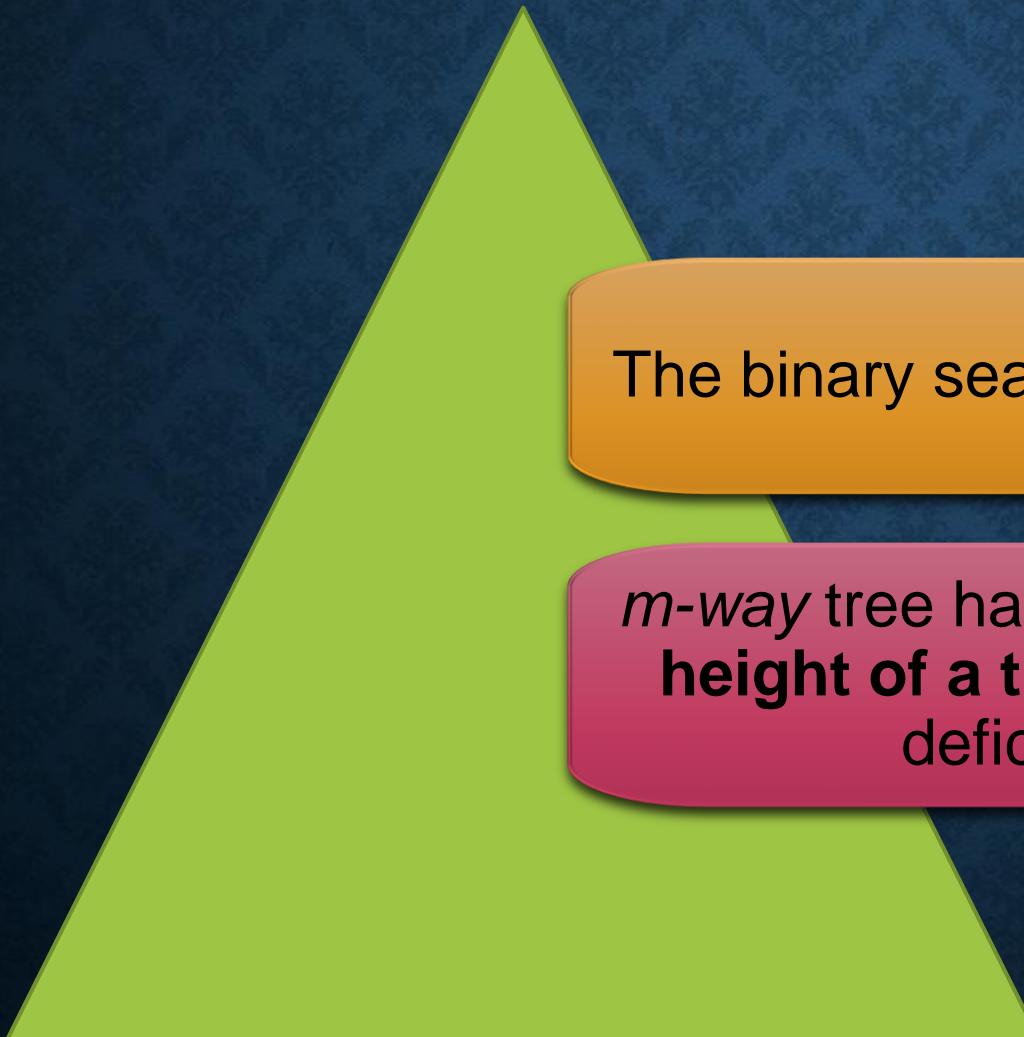


M-way Tree of Order 4



Four-way Tree

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)

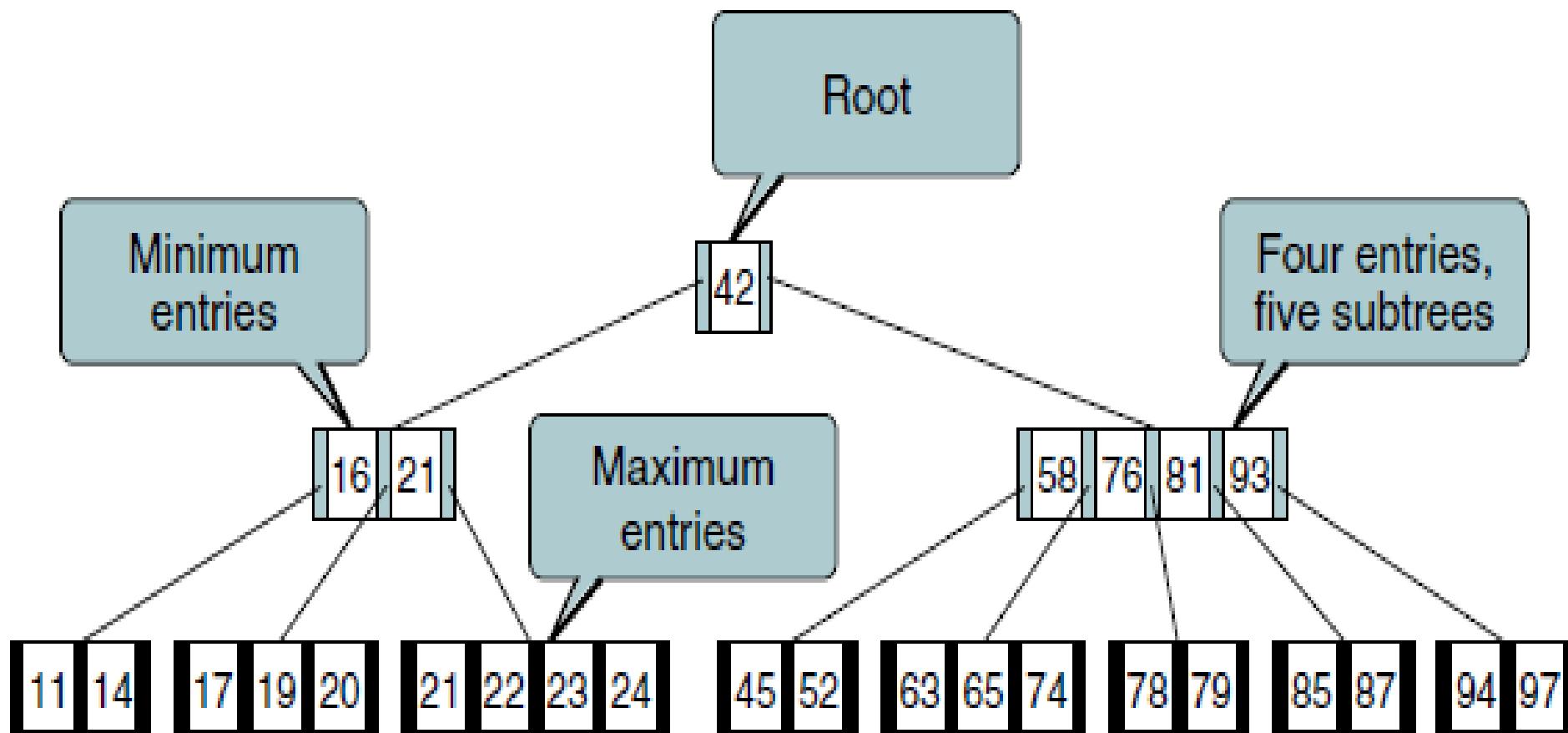
A B-tree is a perfectly balanced m-way tree in which each node, with the possible exception of the root, is at least half full.

**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.

11

(a) Insert 11

11 21

(b) Insert 21

11 14 21

(c) Insert 14

11 14 21 78

(d) Insert 78

11 14 21 78

Full node

(e) Ready to insert 97

11 14 21

Original node

97

New data

78 97

New node

(f) Create new right subtree

21

11 14

Original node

78 97

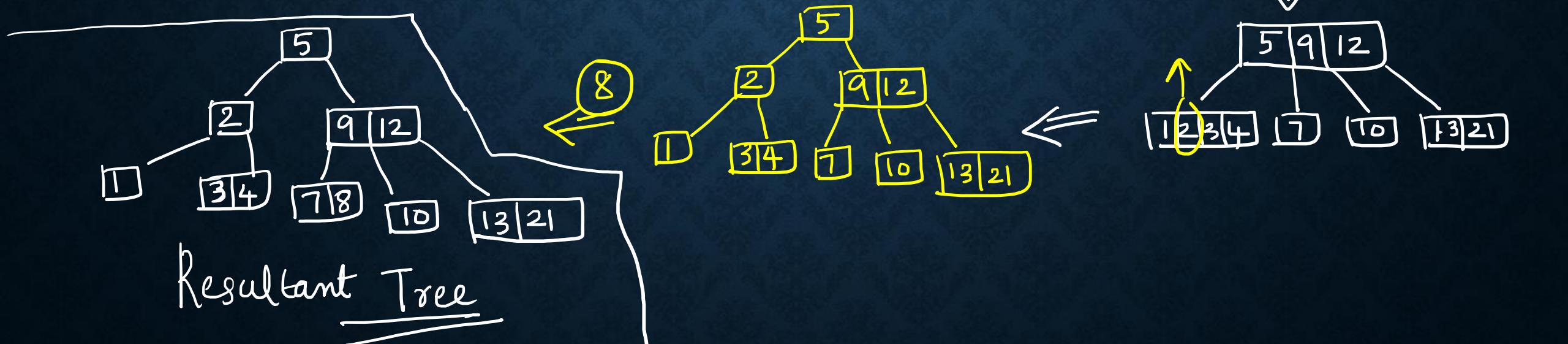
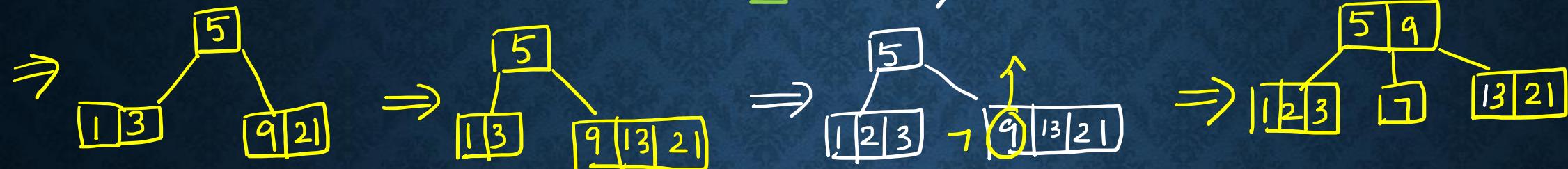
New node

(g) Insert median into parent (new root)

$5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8$ (order 4)

max subtree = 4
max data = 3

$5 \Rightarrow [3|5] \Rightarrow [3|5|21] \Rightarrow [3|5|9|21] \Rightarrow$



78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19

Order 5

52, 30, 21

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.



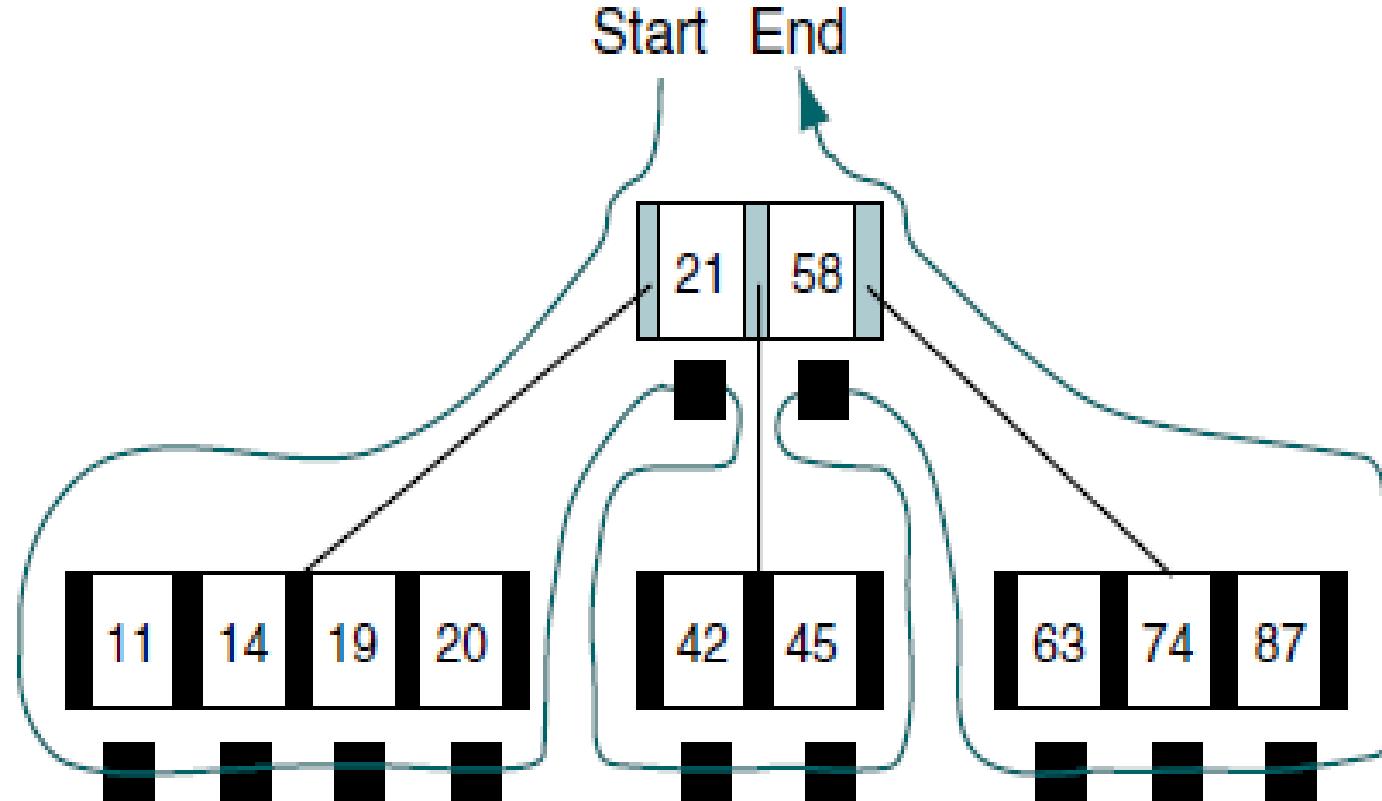
When we can't balance, we must combine nodes.

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.

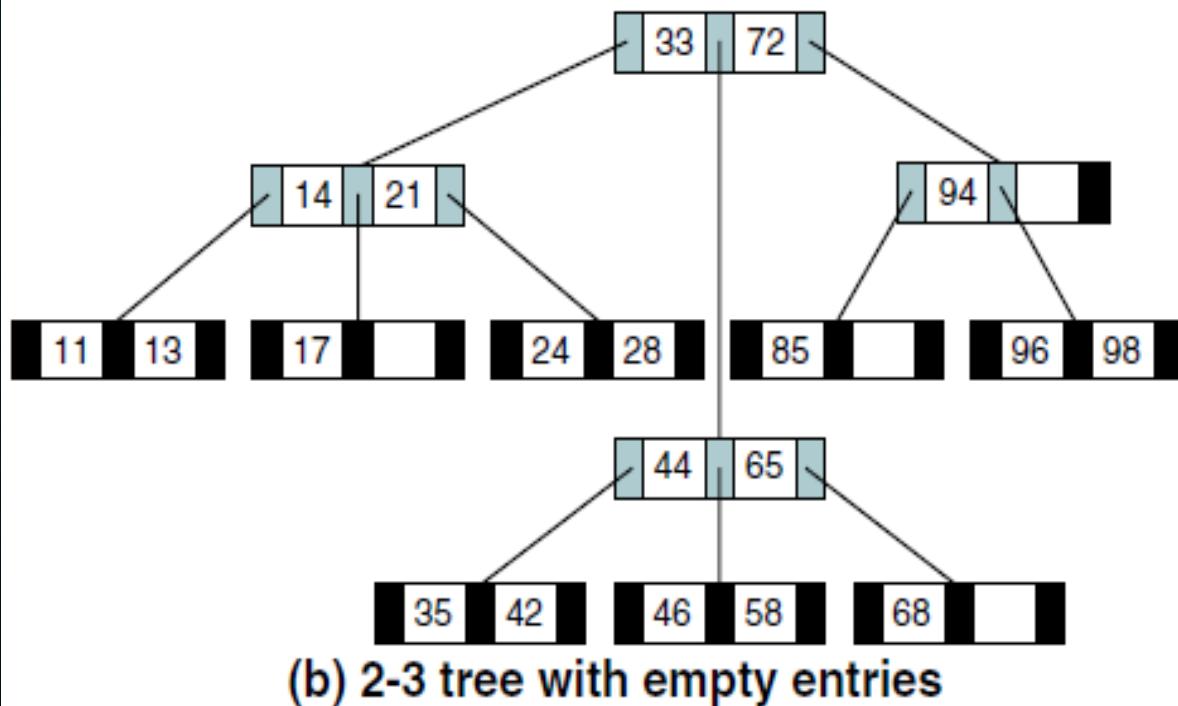
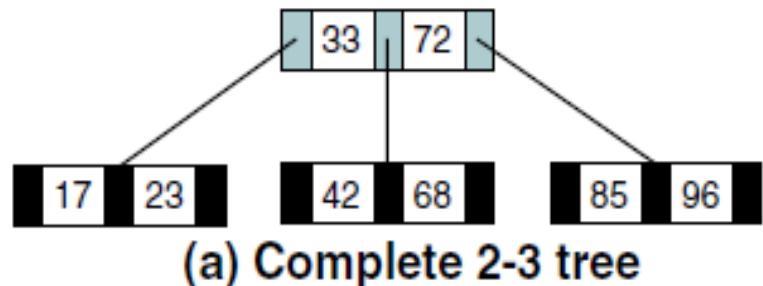
The major difference, however, is that with the ***exception of leaf nodes***, we **don't process all of the data in a node** at the same time. Therefore, we must remember which entry was processed whenever we return to a node and continue from that point.

B-TREE TRAVERSAL.



Basic B-tree Traversal

2-3 TREES.



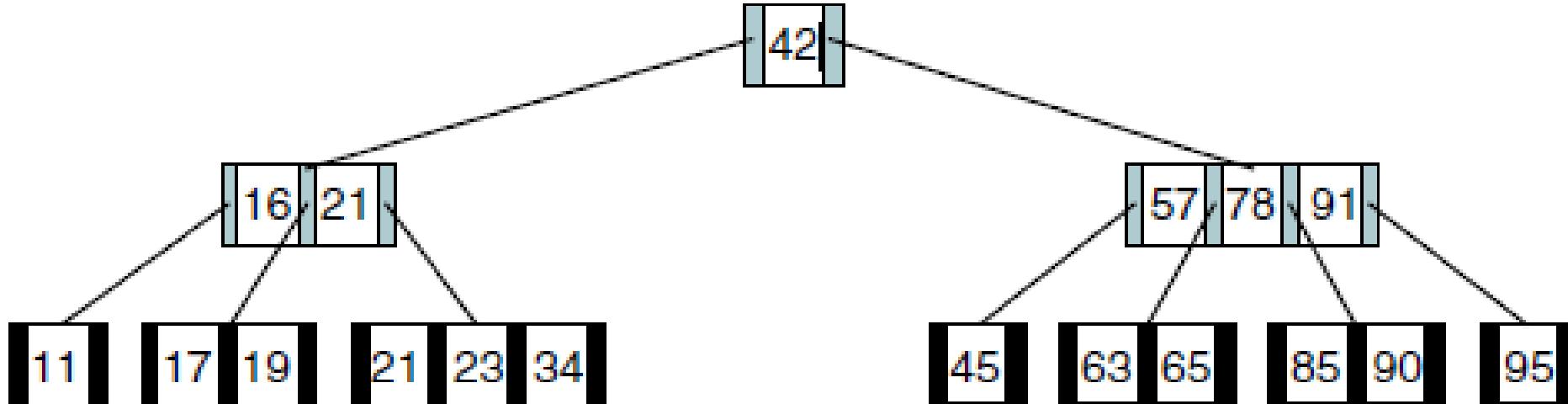
The 2-3 tree is a B-tree of order 3.

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

2-3-4 TREES.



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

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

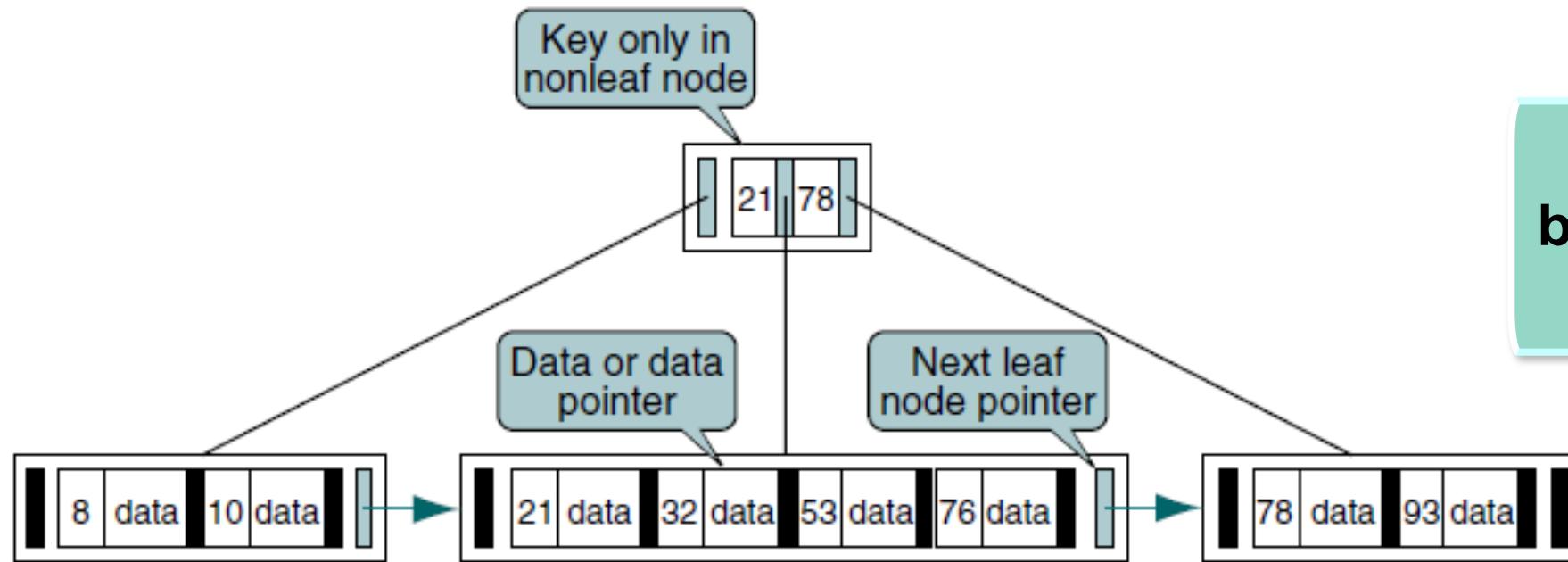
B-TREE VARIATIONS.

B* Tree

B+Tree

Lexical
Search
Tree

B+TREE.

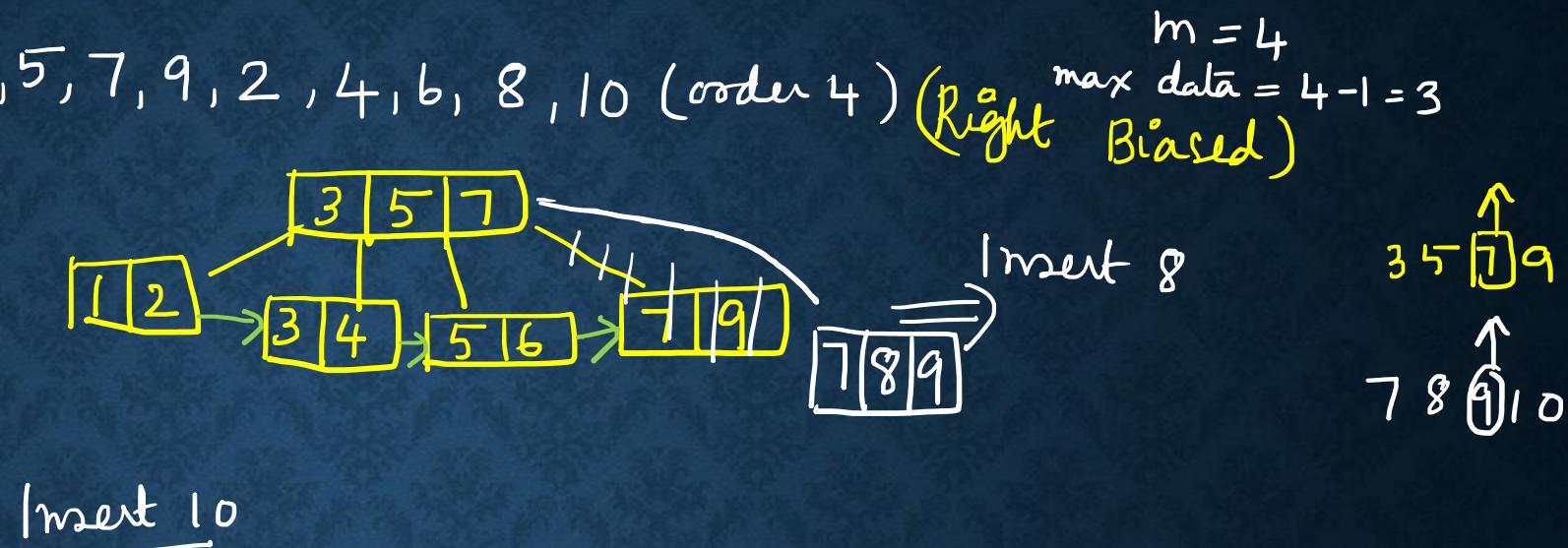
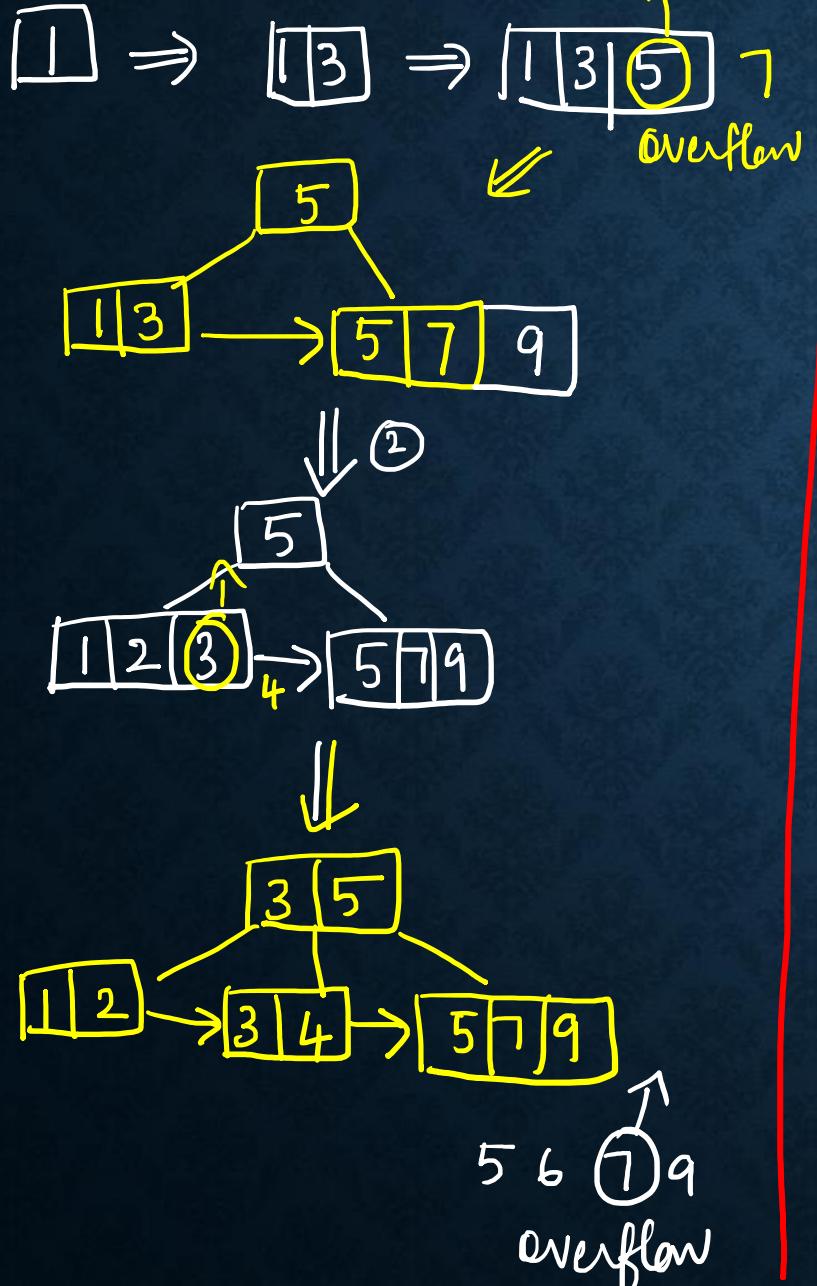


Two differences between the B-tree and the B+tree:

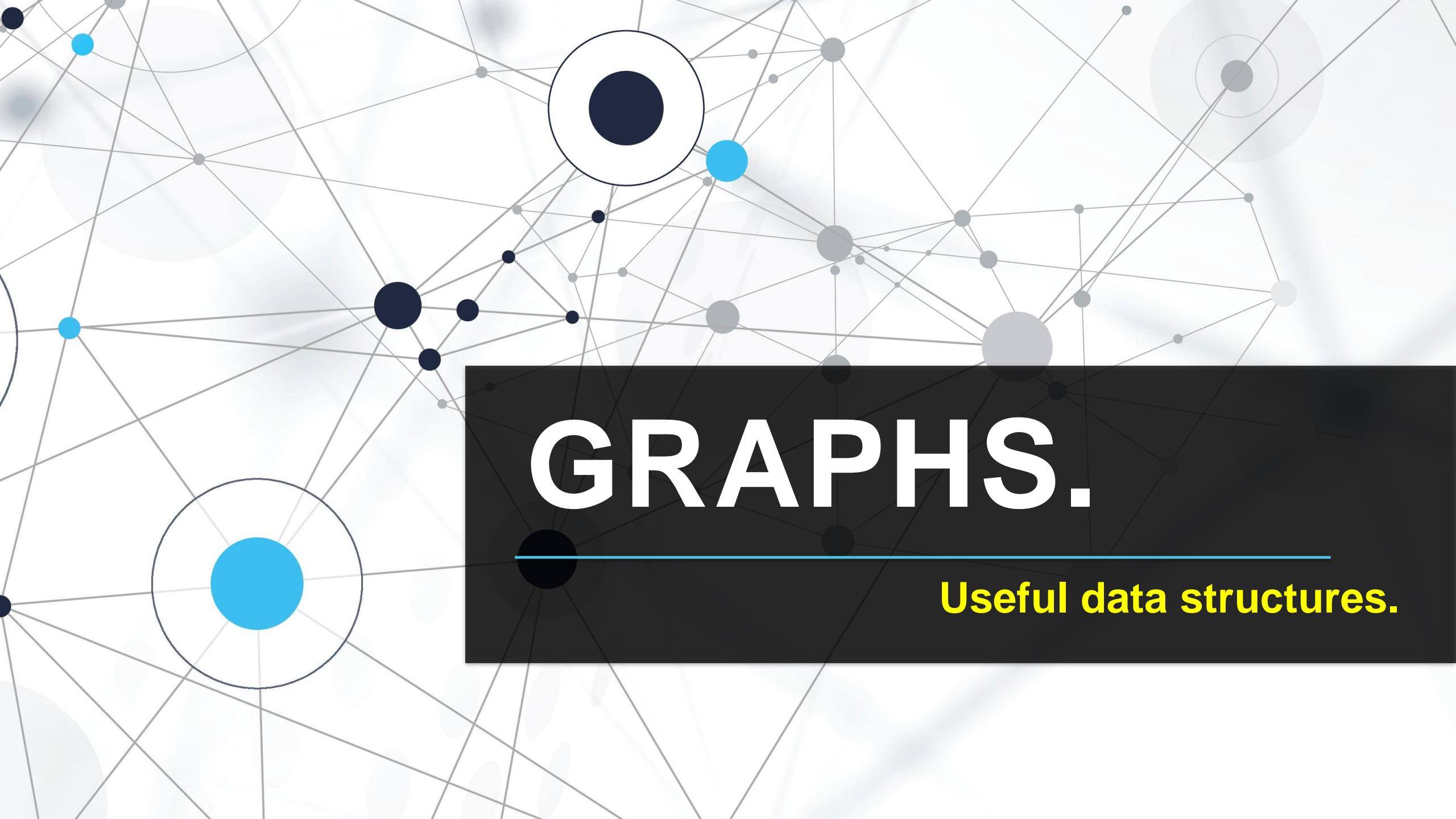
Each data entry must be represented at the leaf level, even though there may be internal nodes with the same keys. Because the internal nodes are used only for searching, they generally do not contain data.

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

Insertion in B + tree :- 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 (order 4) (Right Biased)



Note :- KEYS should not be repeated in internal nodes.



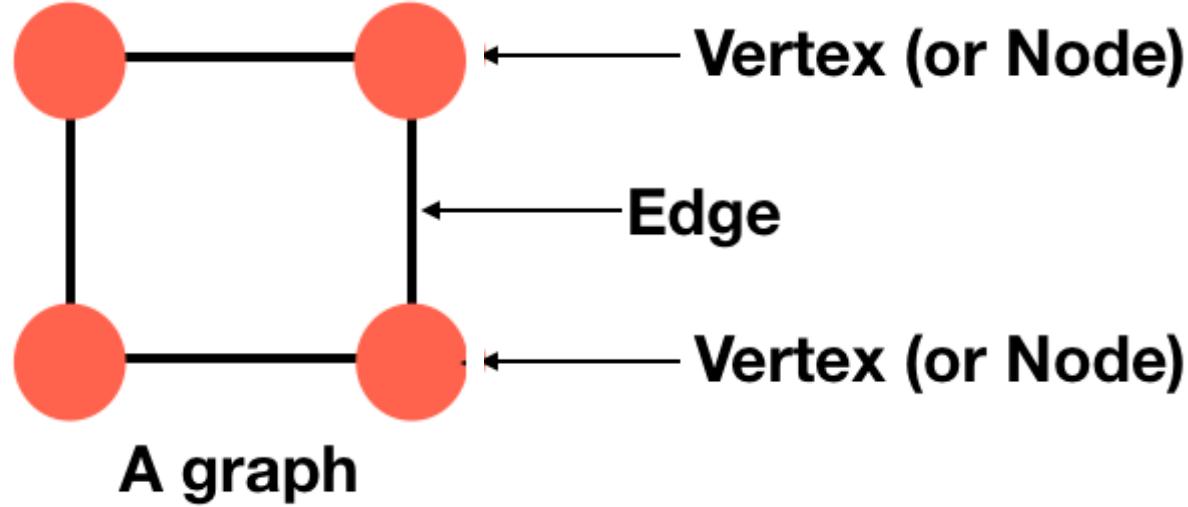
GRAPHS.

Useful data structures.

WHAT WE KNOW ALREADY.

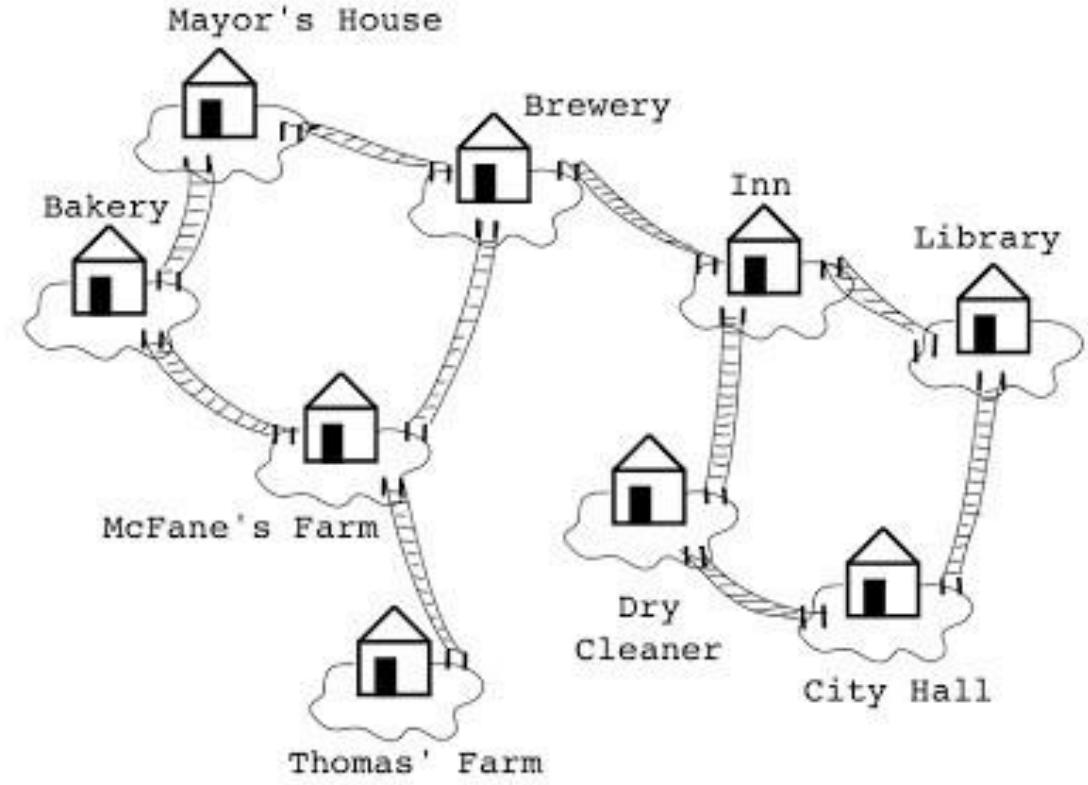
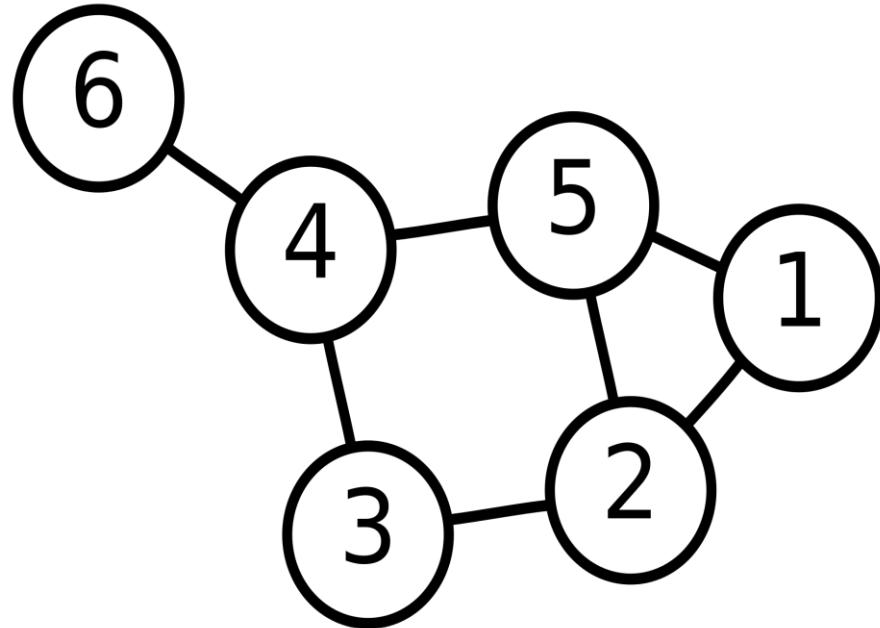
- 👉 Need for ADTs and data structures:
 - 👉 Regular Arrays (with dynamic sizing)
 - 👉 Linked Lists
 - 👉 Stacks, Queues
 - 👉 Heaps
 - 👉 Unbalanced and Balanced Search Trees, B-tree, B+-tree
- 👉 Some algorithms like Tree traversals

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 $V = \{v_1, v_2, \dots, v_n\}$

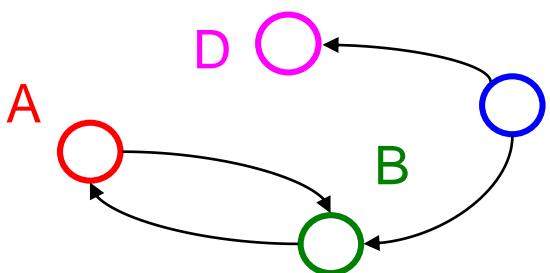
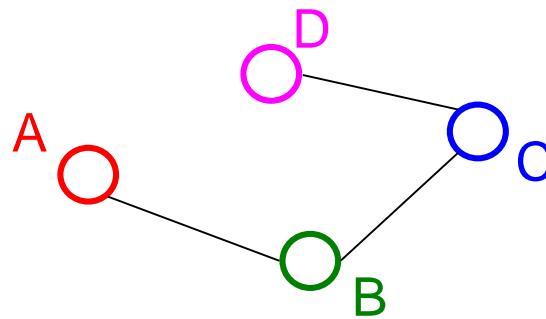
$E(G)$: a set of edges (pairs of vertices) $E = \{e_1, e_2, \dots, e_m\}$

An edge "connects" the vertices

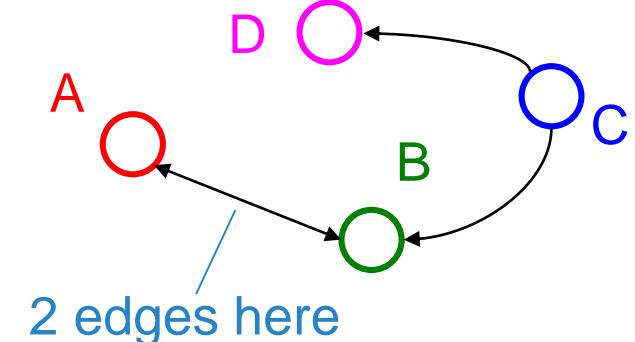
Graphs can be directed or undirected

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.



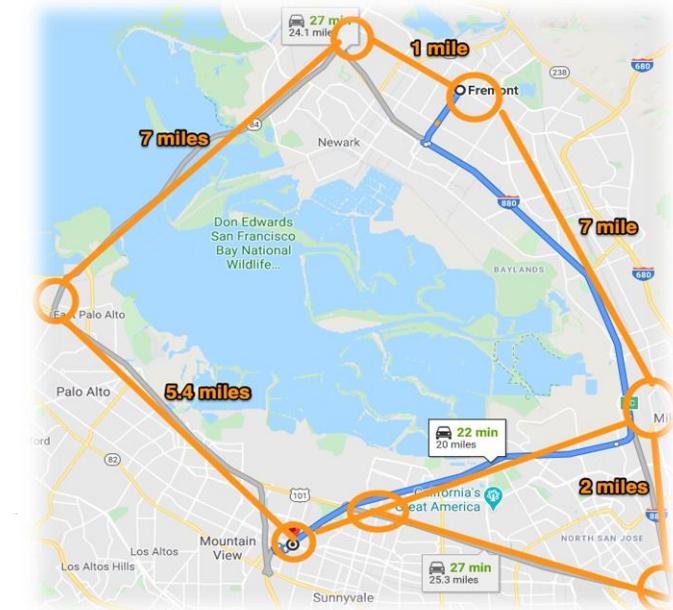
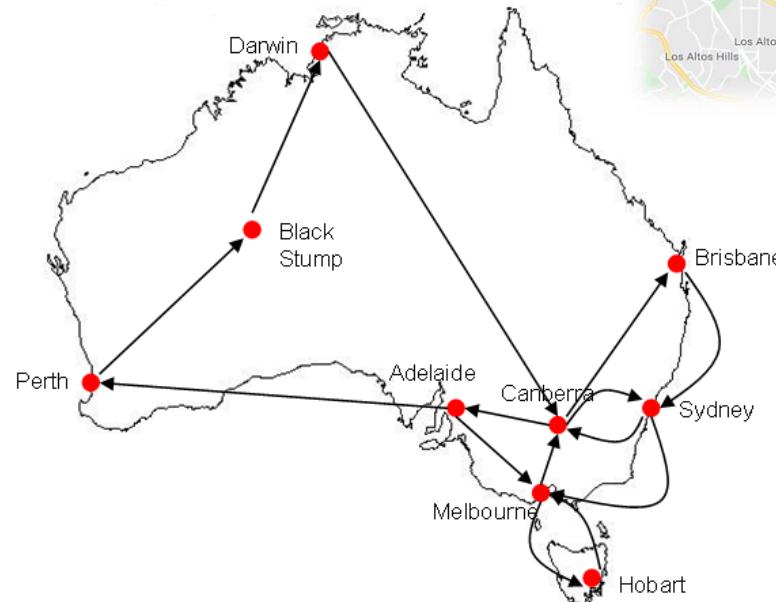
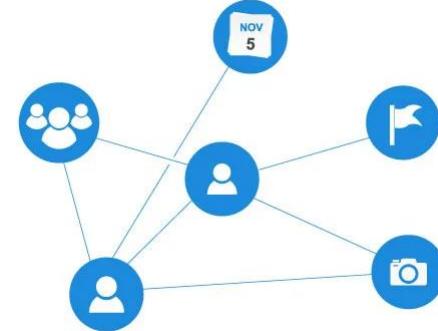
or



- ↳ **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*.

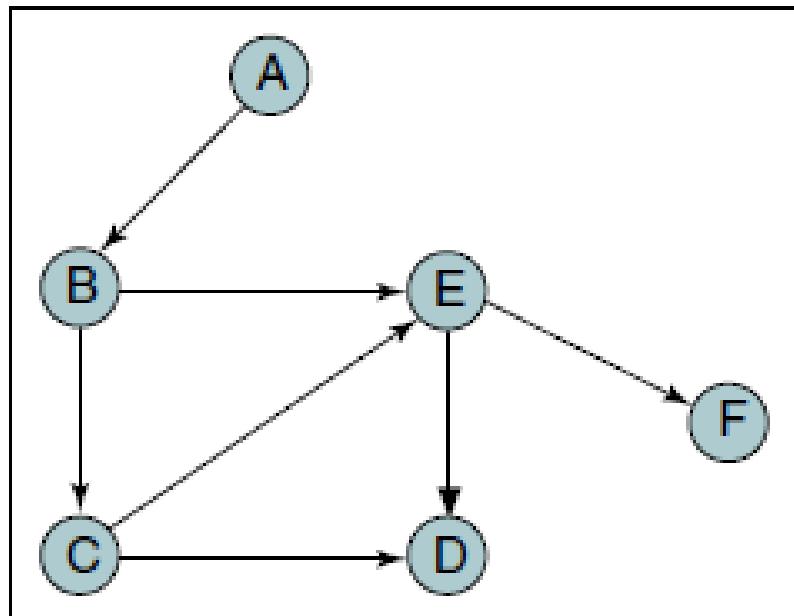


FIGURE 11-1 (a) **Directed graph**

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.

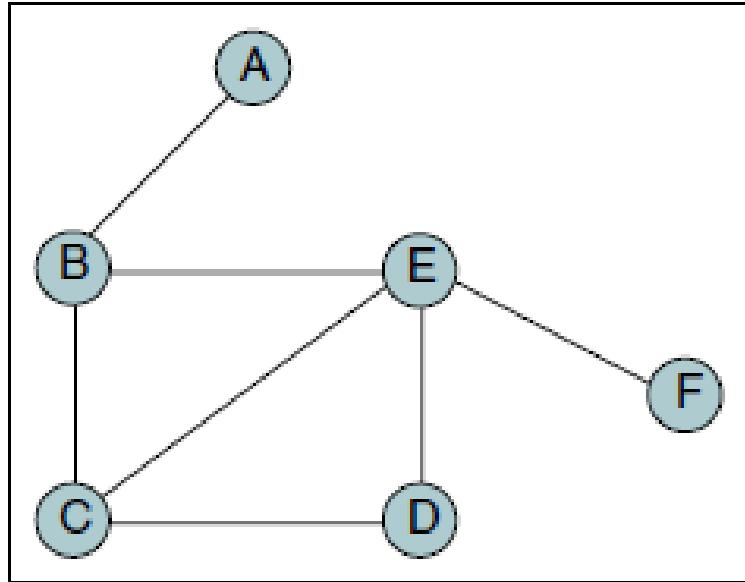


FIGURE 11-1 (b) Undirected graph

$\{A, B, C, E\}$ 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.

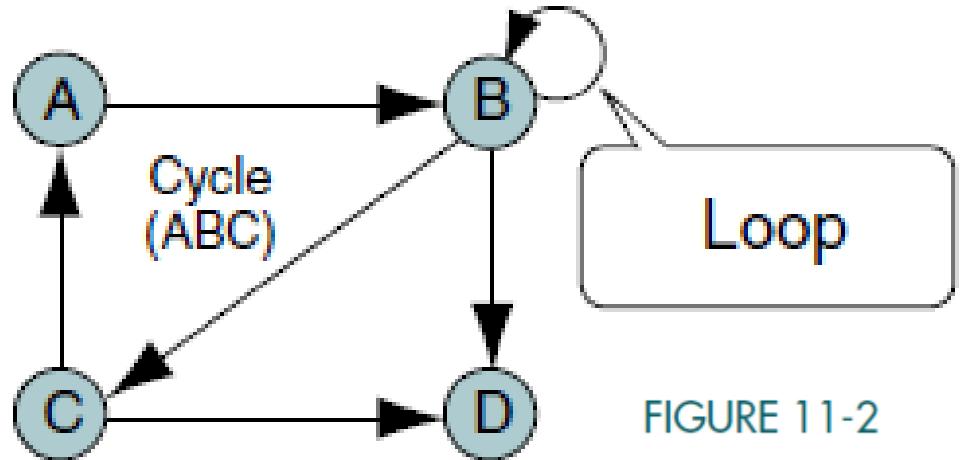
A path **is a sequence of vertices** in which each vertex is adjacent to the next one.

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.

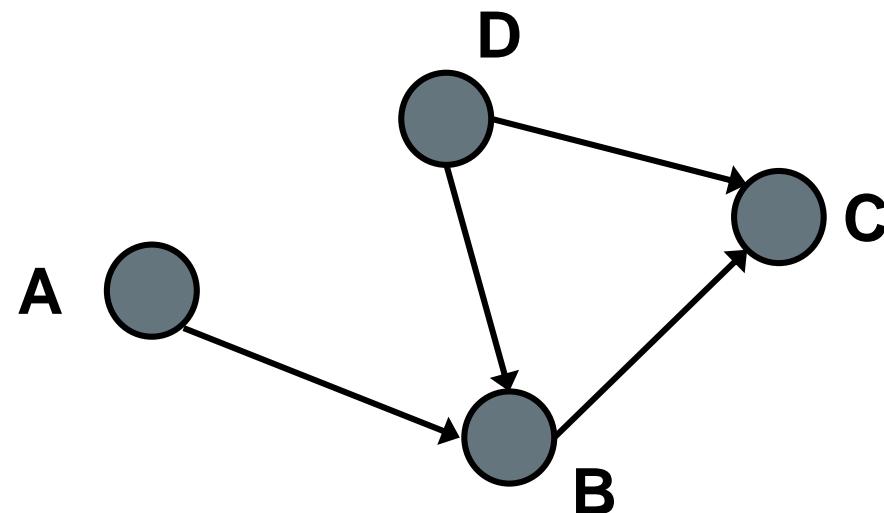
A **cycle** is a path consisting of at least three vertices that starts and ends with the same vertex.

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

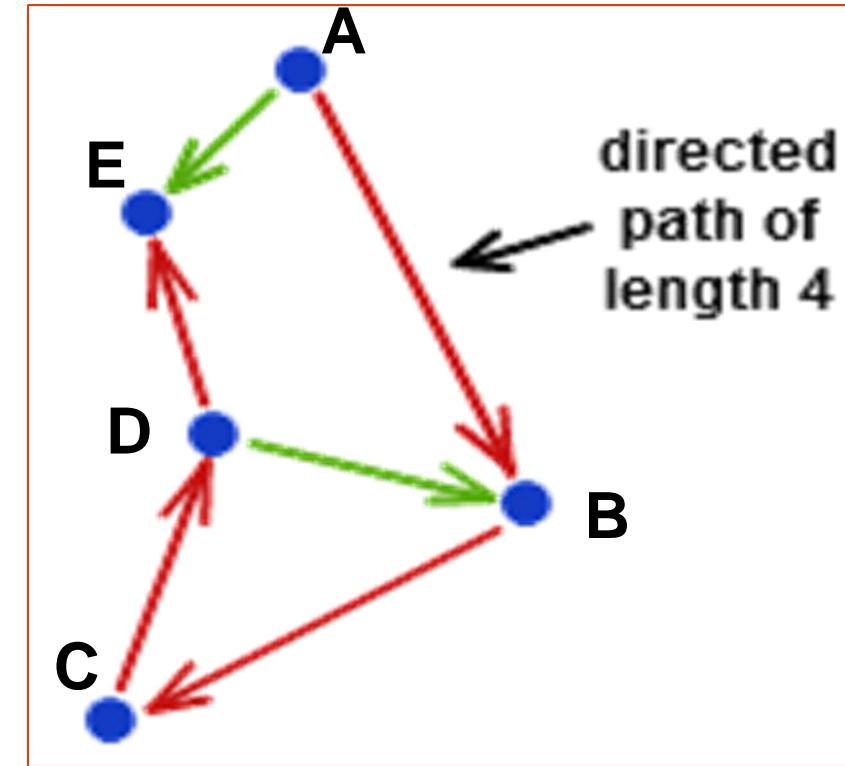


Is there a path from A to D?

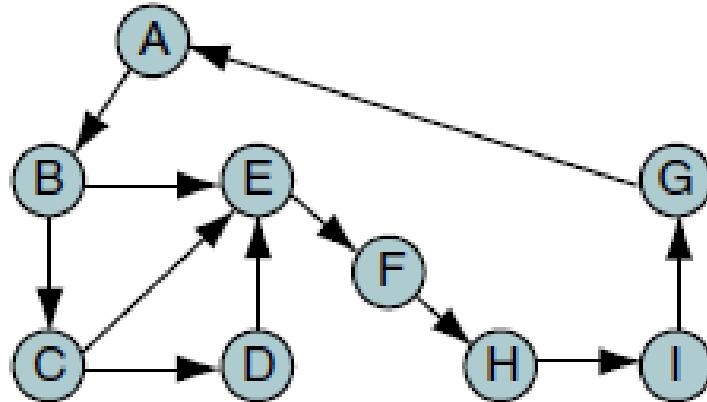
No

Does the graph contain any cycles?

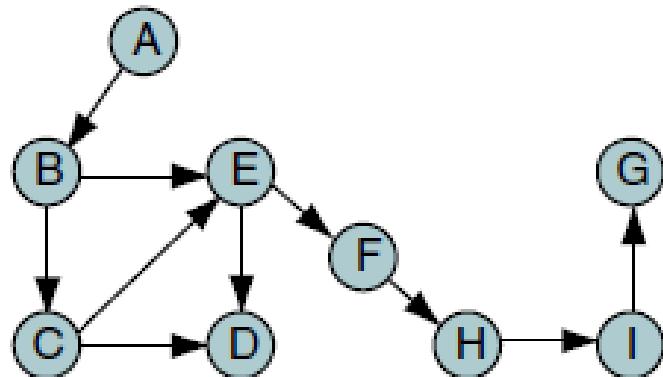
No



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.

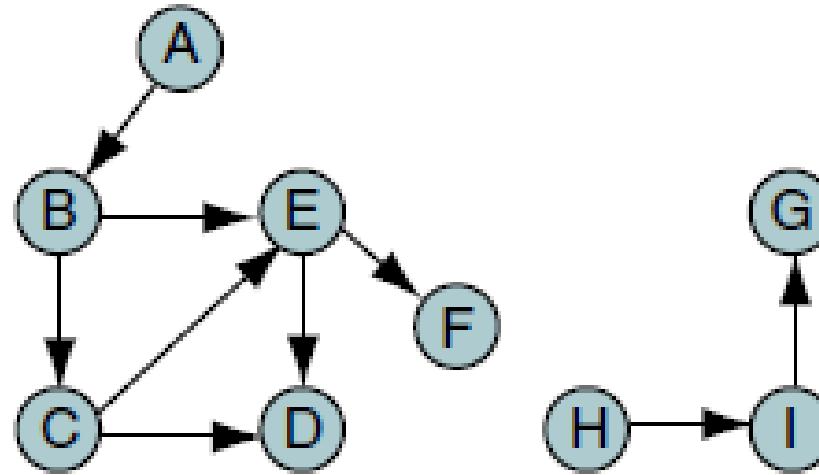


FIGURE 11-3

(c) Disjoint graph

A graph is a disjoint graph if it is not connected.

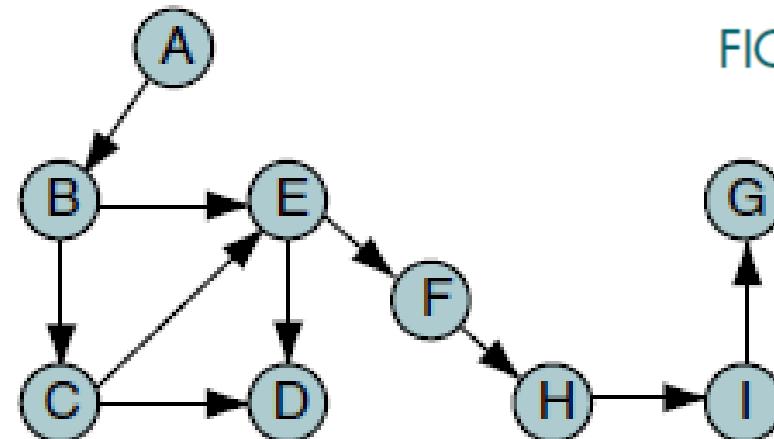
DEGREE OF A VERTEX.

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

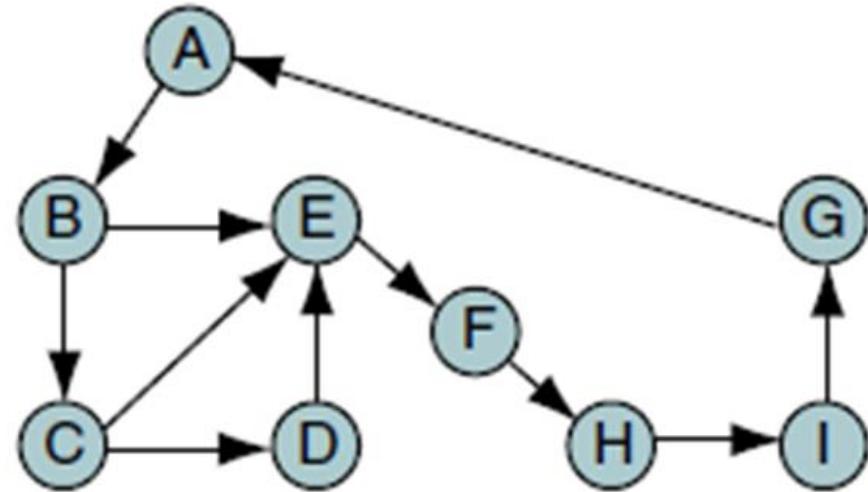
DEGREE OF A VERTEX.

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.

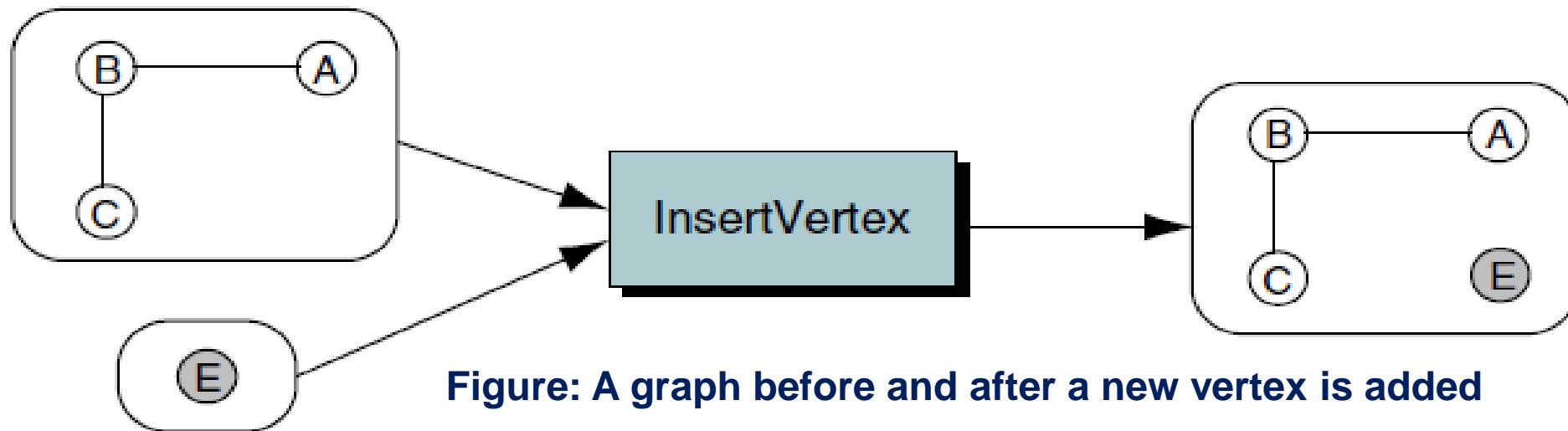
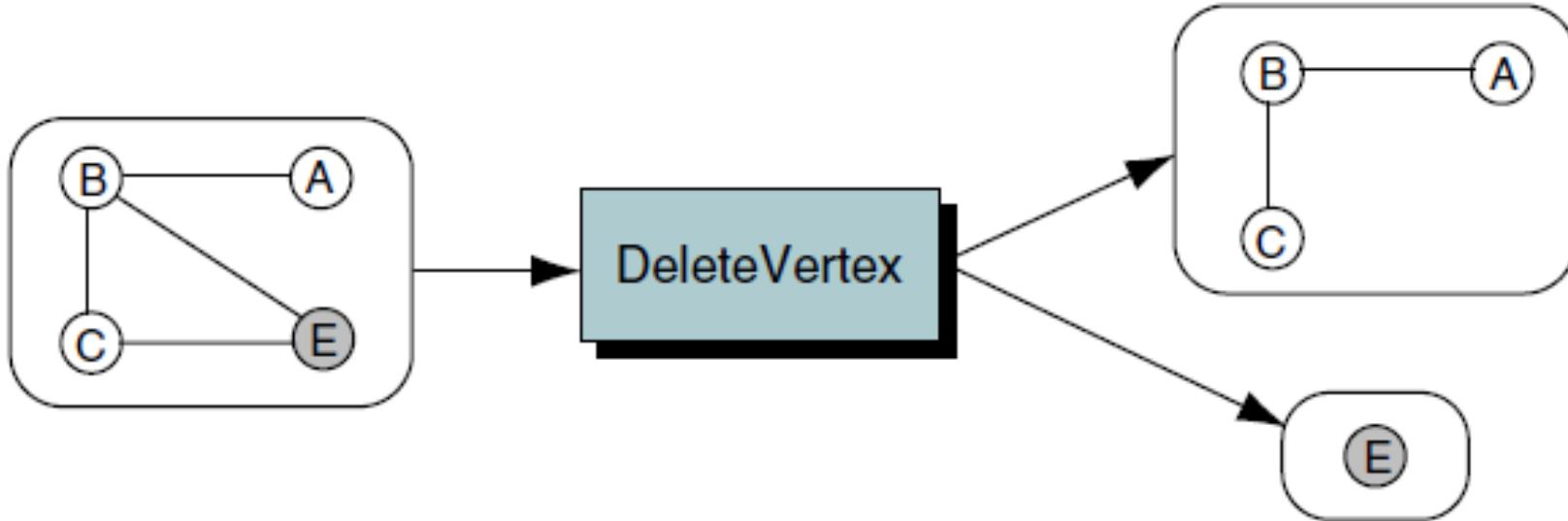
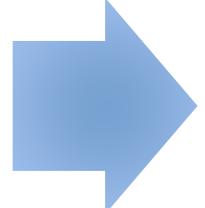


Figure: A graph before and after a new vertex is added

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.

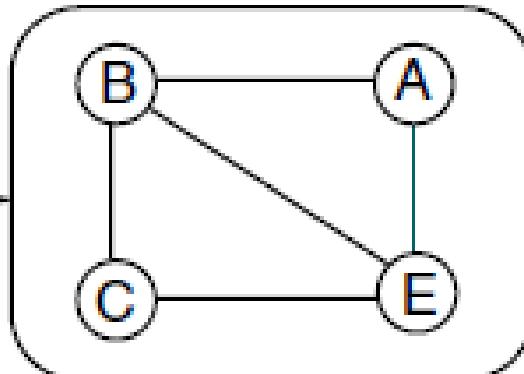
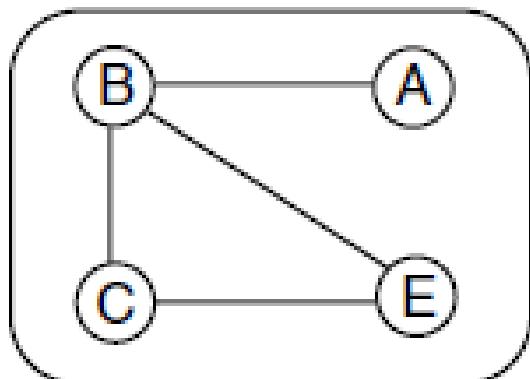


Figure: Adding an edge, {A, E}, to the graph.

DELETE EDGE.

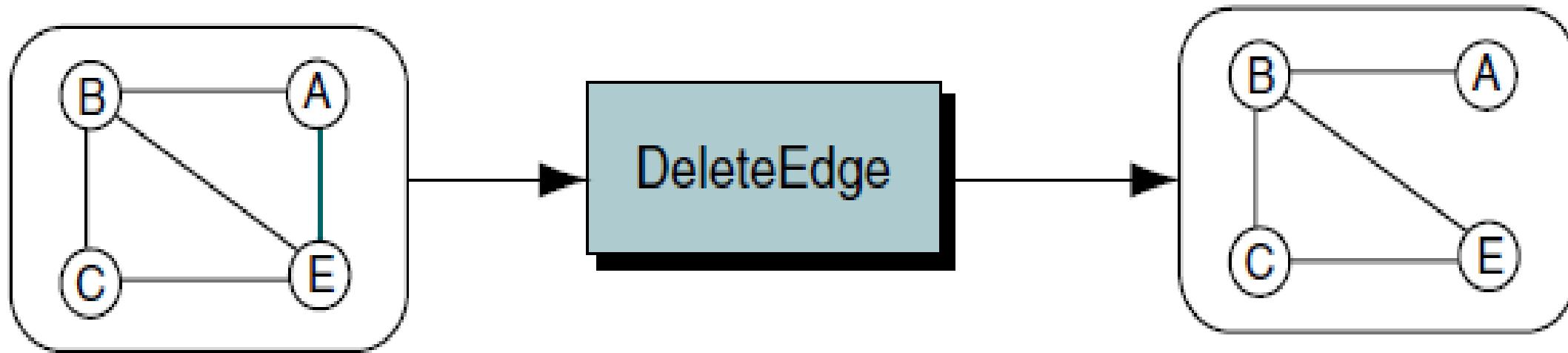


Figure: Deleting the edge, {A, E}, to the graph.

Delete edge removes one edge from a graph.

FIND VERTEX.

Find vertex traverses a graph, looking for a specified vertex.

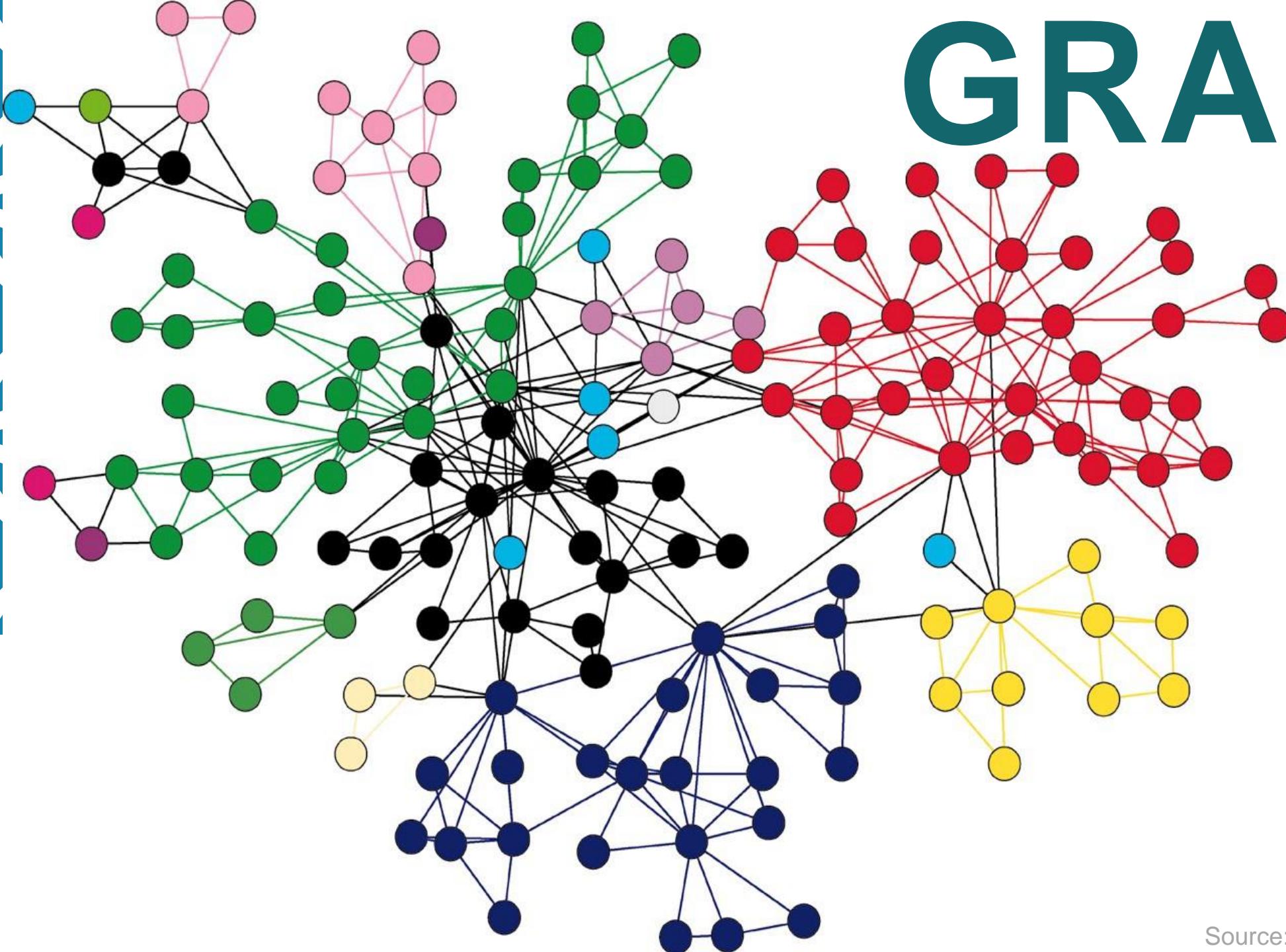
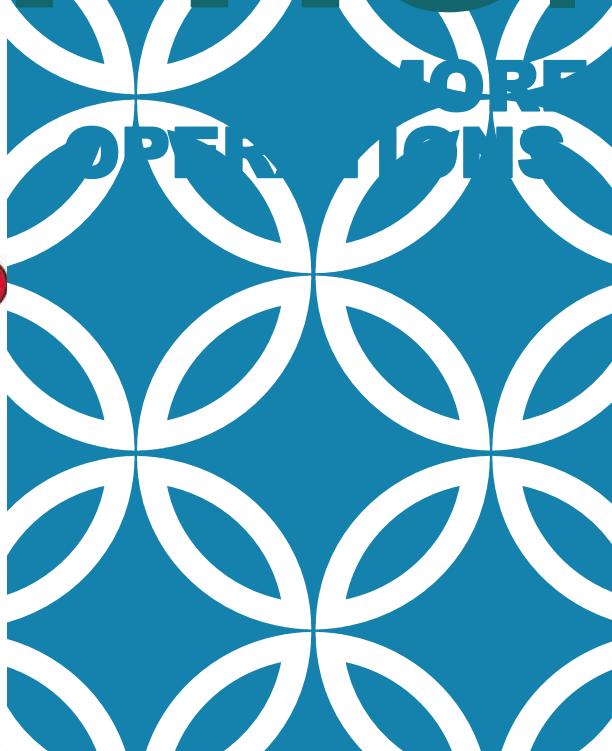
If the vertex is found, its data are returned. If it is not found, an error is indicated.



Figure: Find vertex traverses the graph, looking for vertex C..

END.

GRAPHS



WE WILL LOOK INTO:

Graph Traversal

Graph Storage Structures

- Adjacency Matrix.
- Adjacency List.

Graph Algorithms.

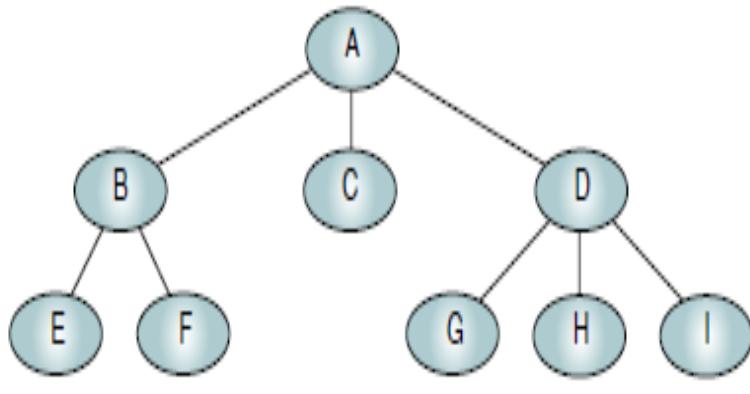
Networks

- City Network.
- Minimum Spanning Tree
- Shortest Path Algorithm

GRAPH TRAVERSAL

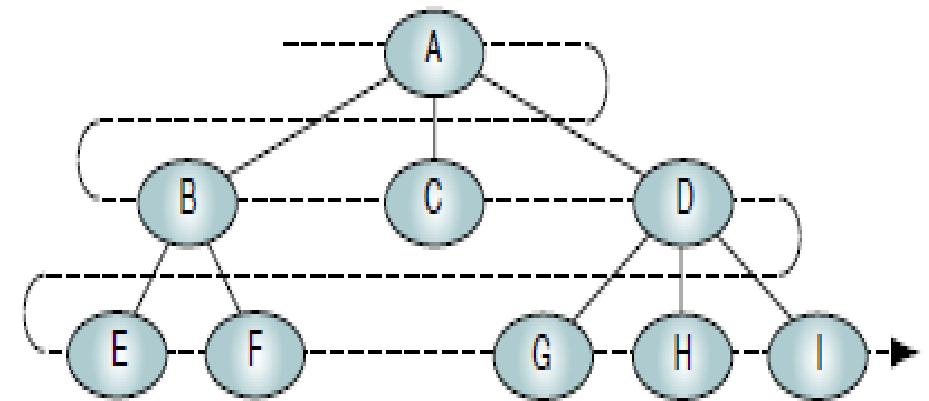
Two Commonly used Traversal Techniques are

- **Depth First Search (DFS)**
 - Process all of a vertex's descendants before moving to an adjacent vertex.
- **Breadth First Search (BFS)**



Depth-first traversal: A B E F C D G H I

j to the



Breadth-first traversal: A B C D E F G H I

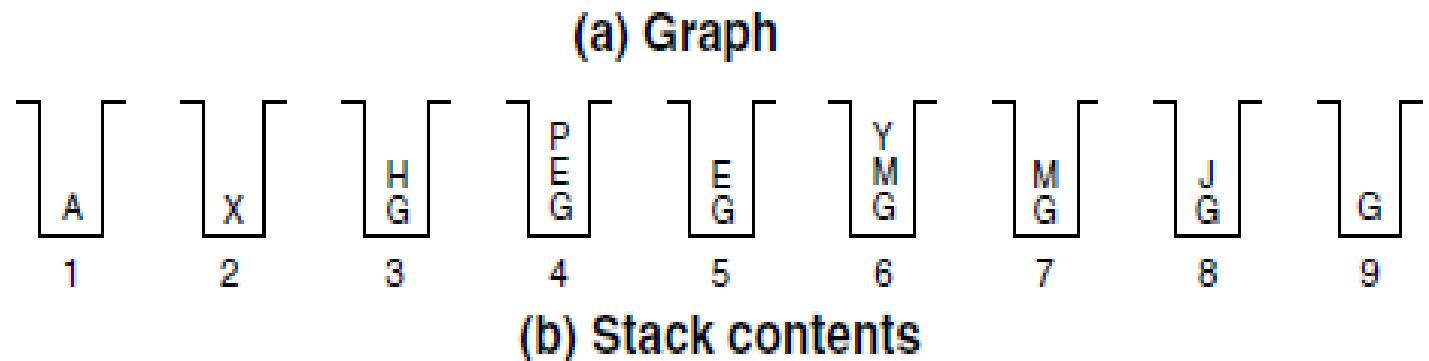
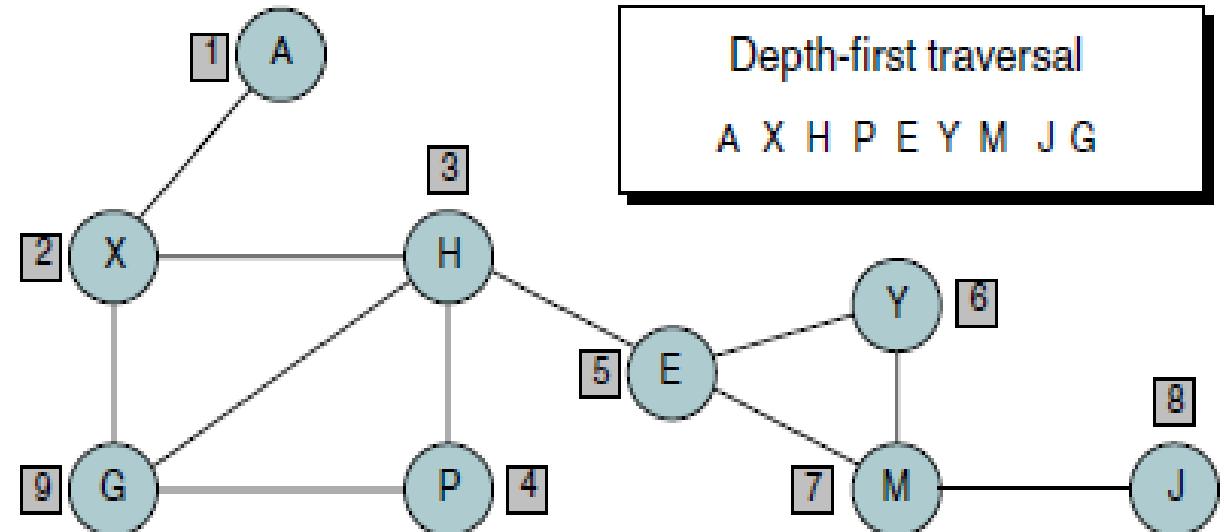
Depth-first Traversal of a Tree

Breadth-first Traversal of a Tree

DEPTH FIRST SEARCH (DFS)

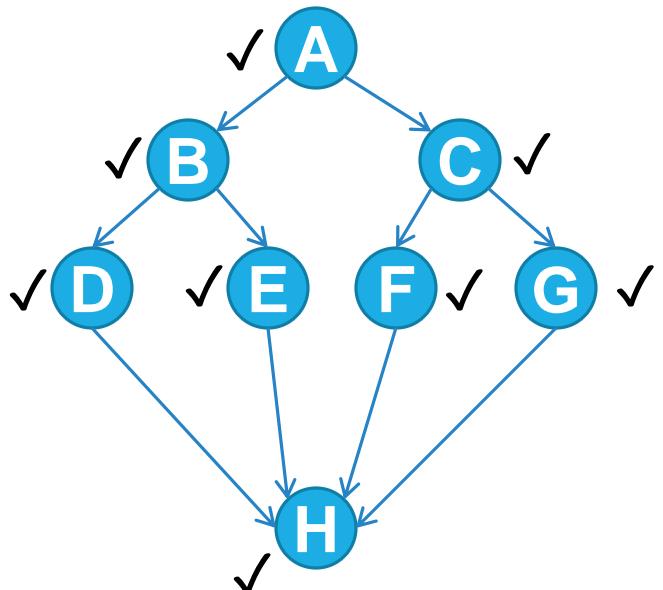
Steps:

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.

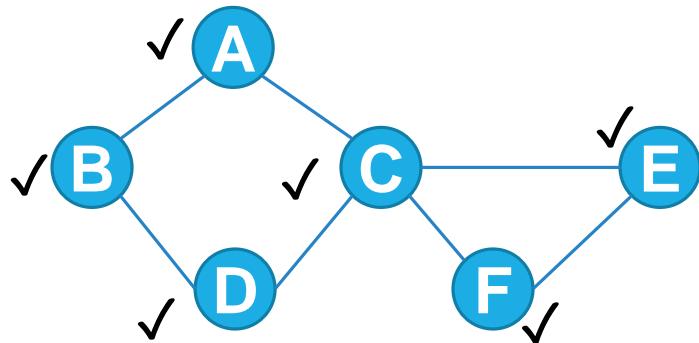
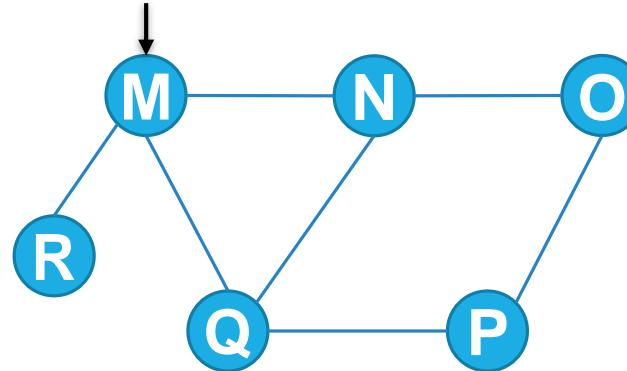


Depth-first Traversal of a Graph

DEPTH FIRST SEARCH (DFS)



ABDHECFG

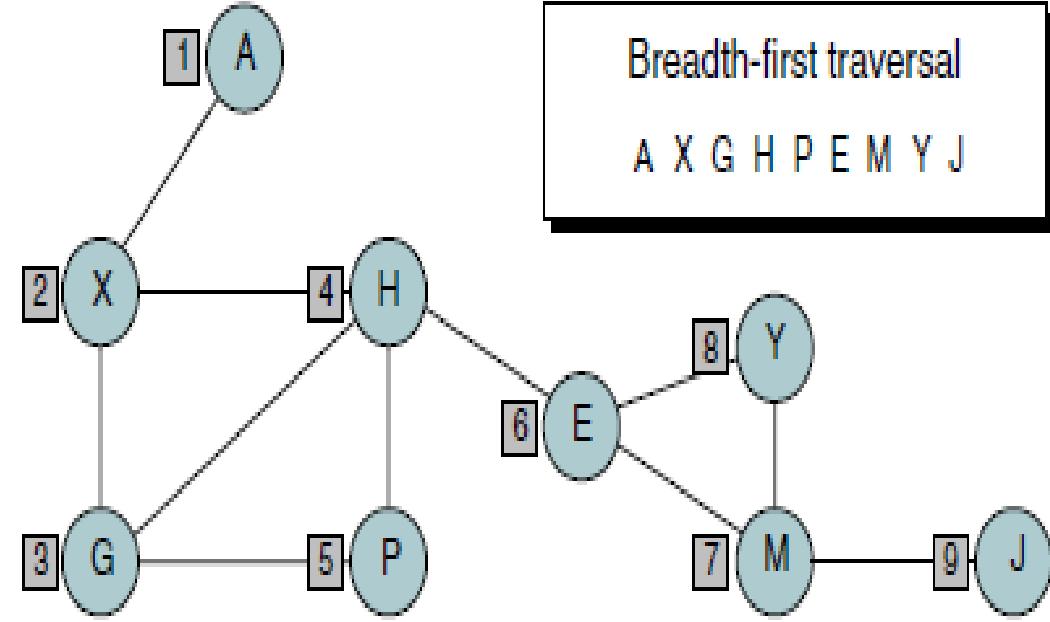


A B D C F E

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**.



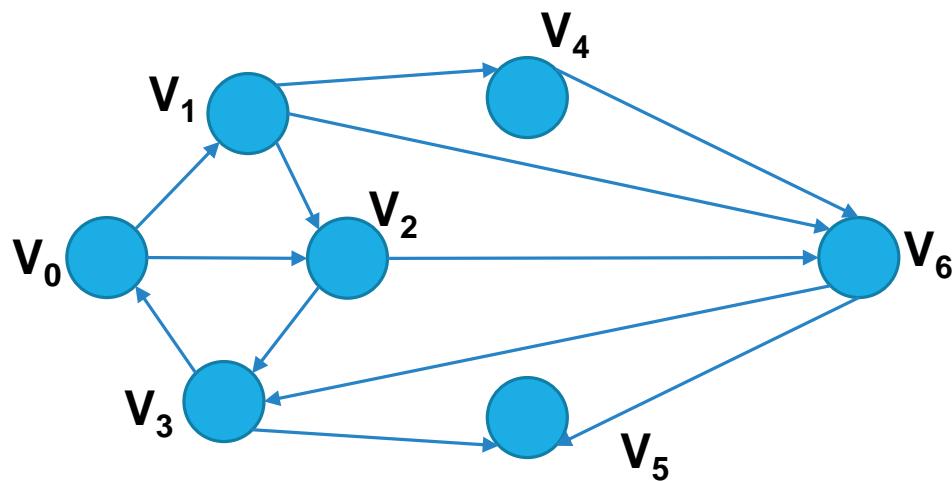
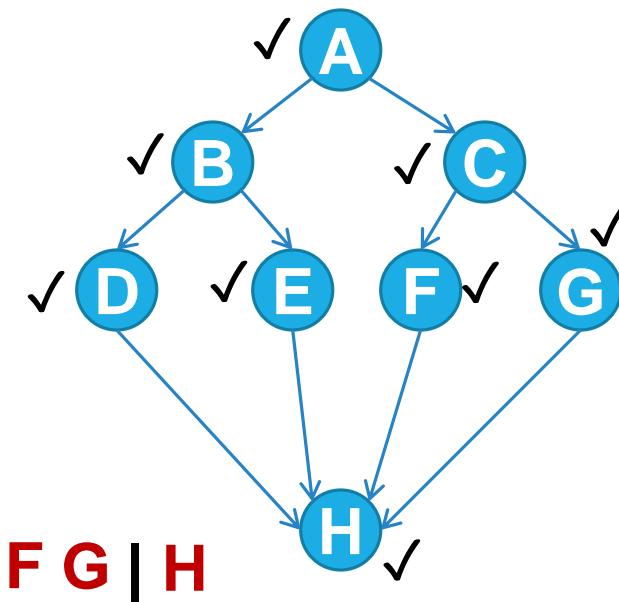
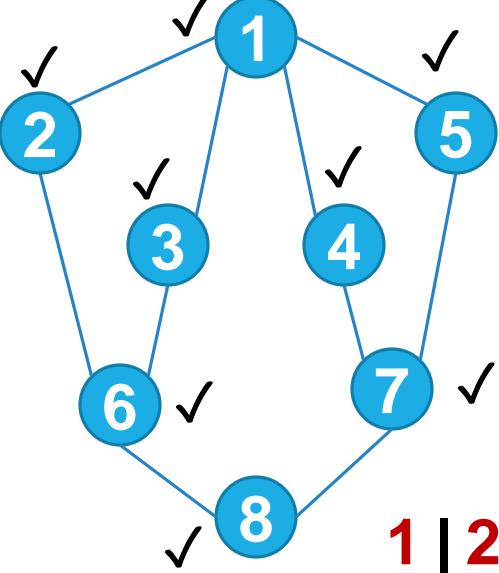
(a) Graph



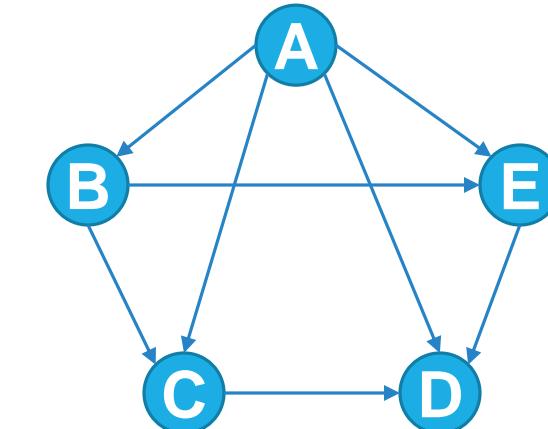
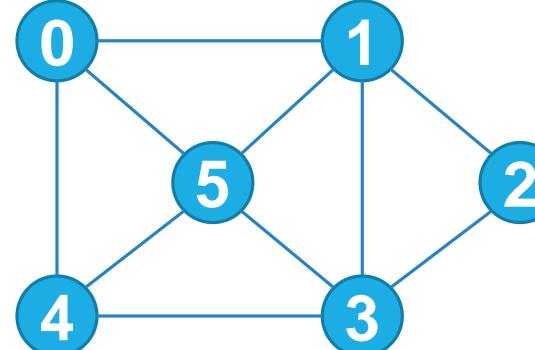
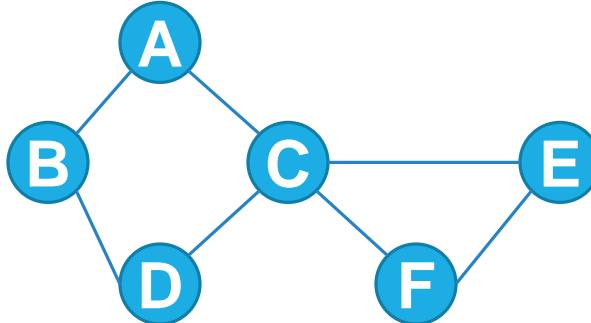
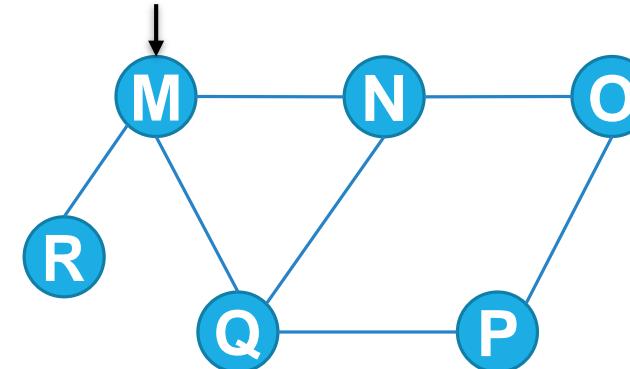
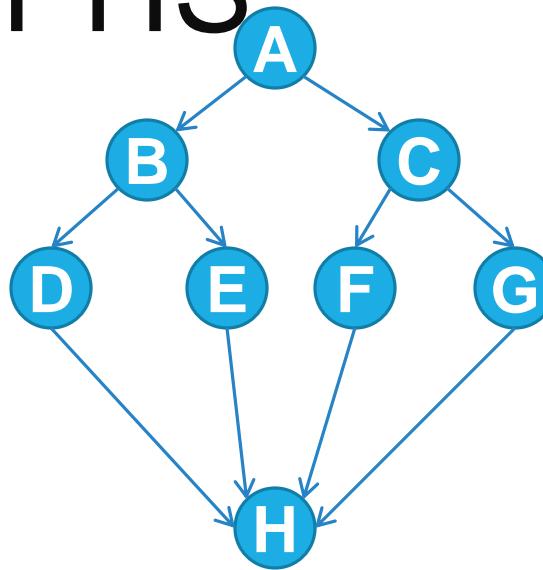
(b) Queue contents

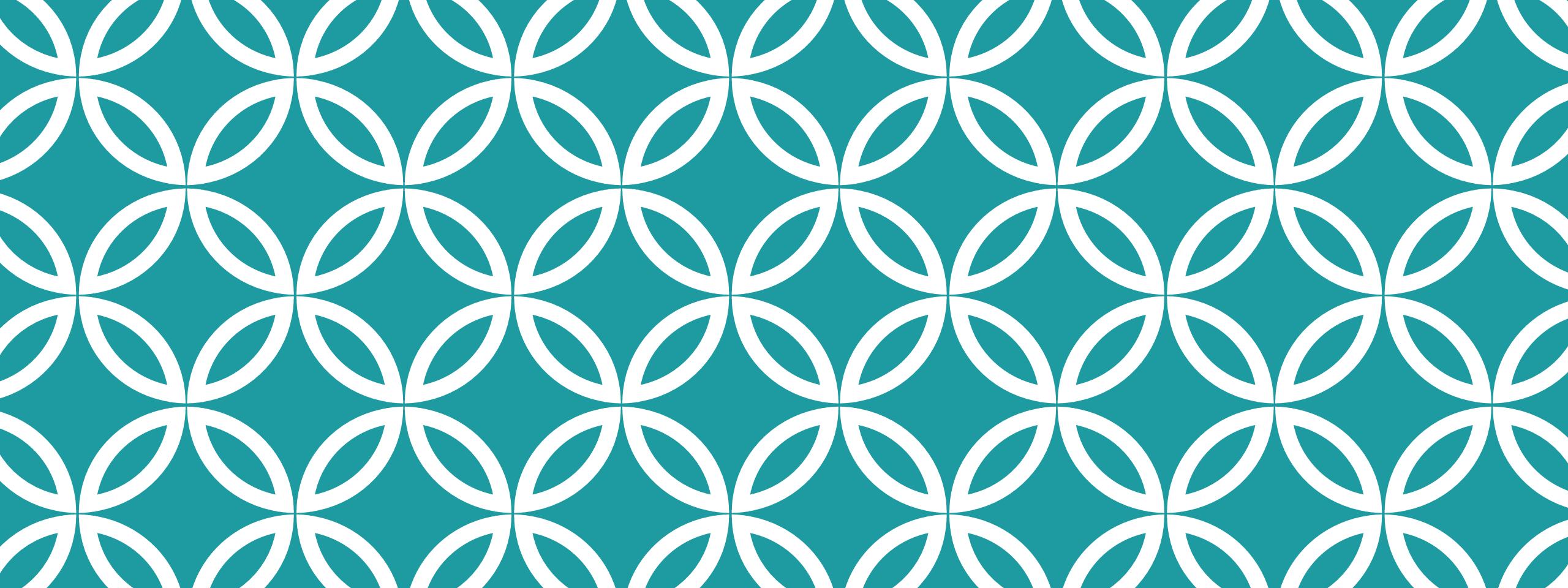
Breadth-first Traversal of a Graph

Breadth First Search (BFS)

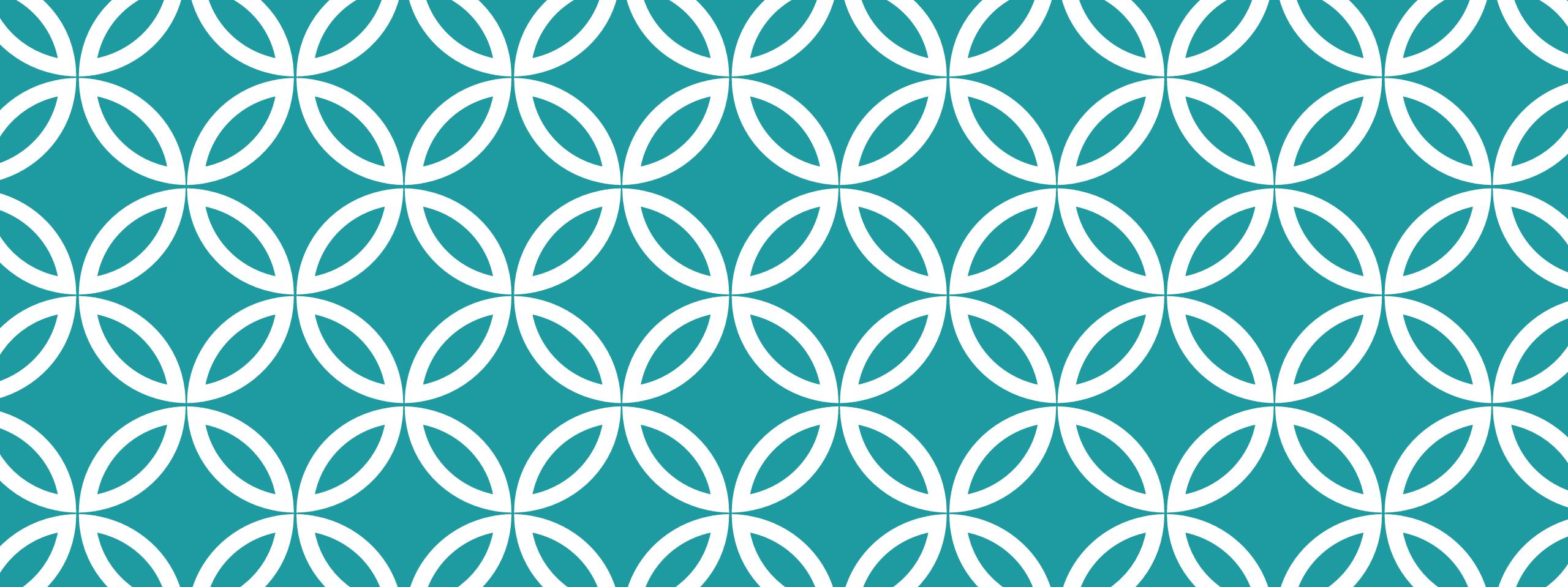


WRITE DFS & BFS OF FOLLOWING GRAPHS





END.



GRAPH STORAGE STRUCTURES.

Ref. Books: Corman & Forouzan

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,

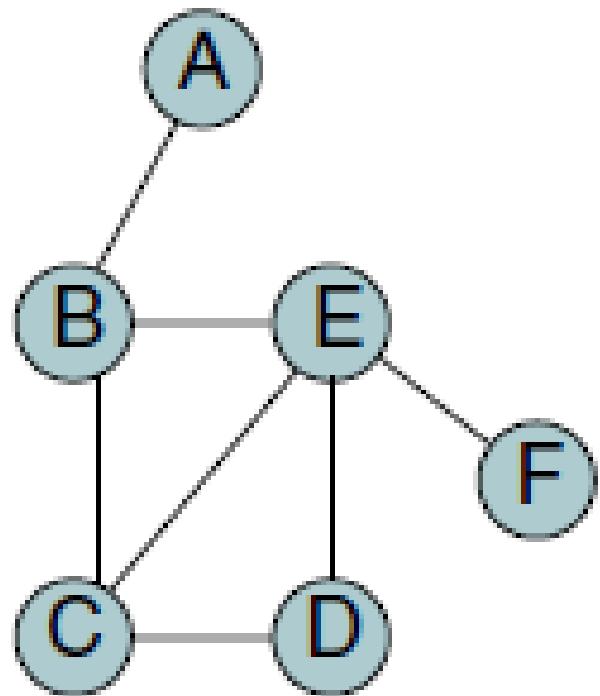
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E ; \\ 0 & \text{otherwise} \end{cases}$$

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.



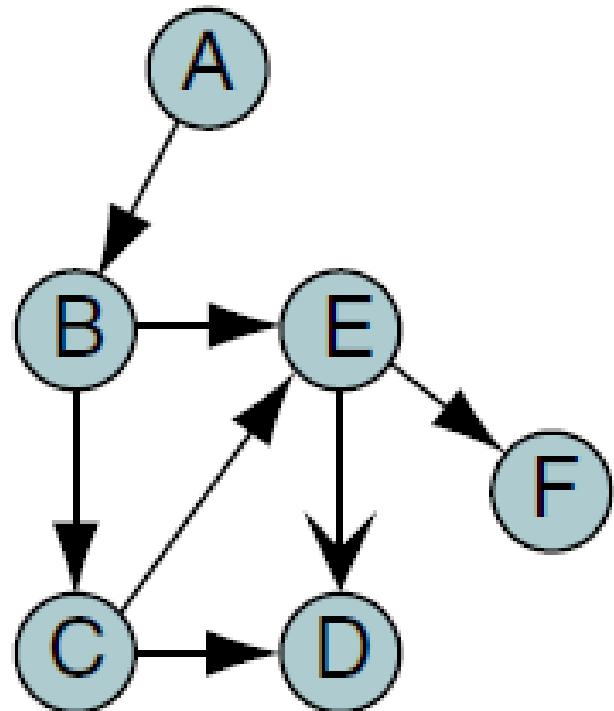
A
B
C
D
E
F

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***.

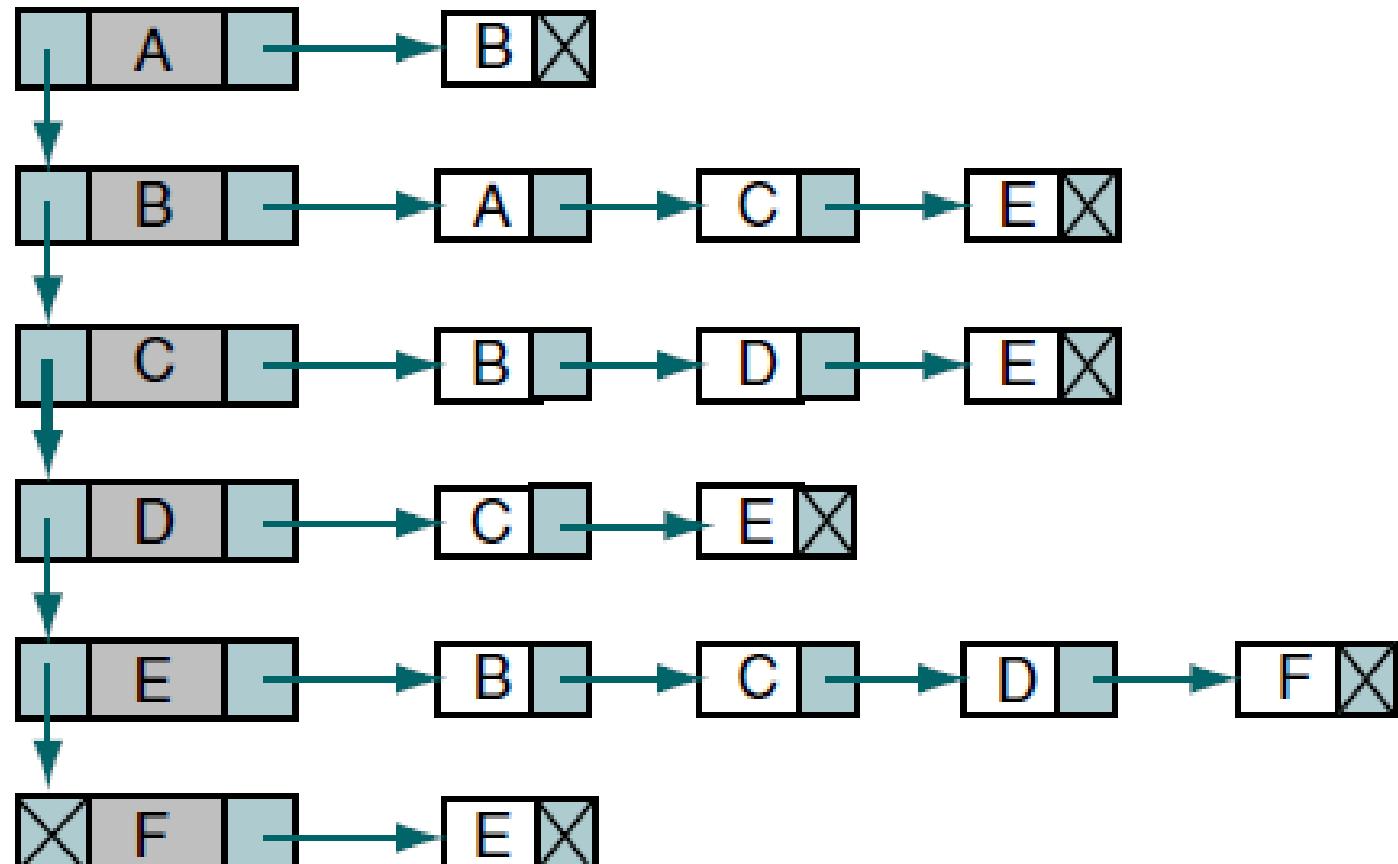
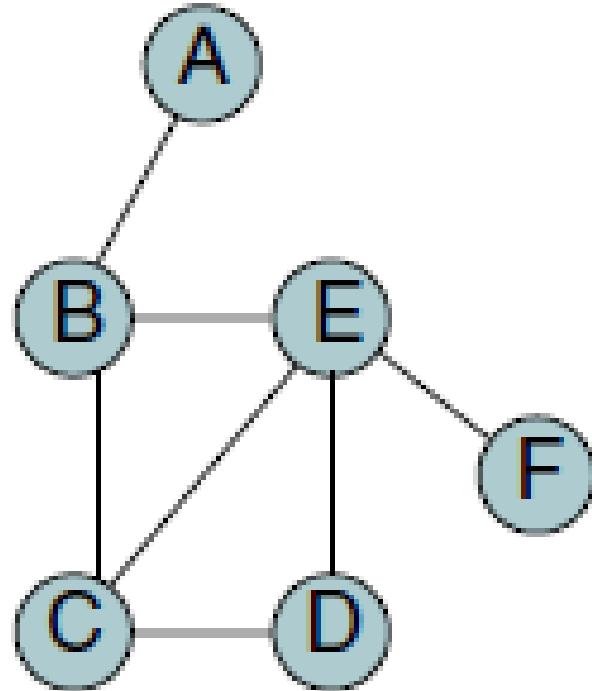
The **vertex list** is a **singly linked list** of the vertices in the list.

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**.

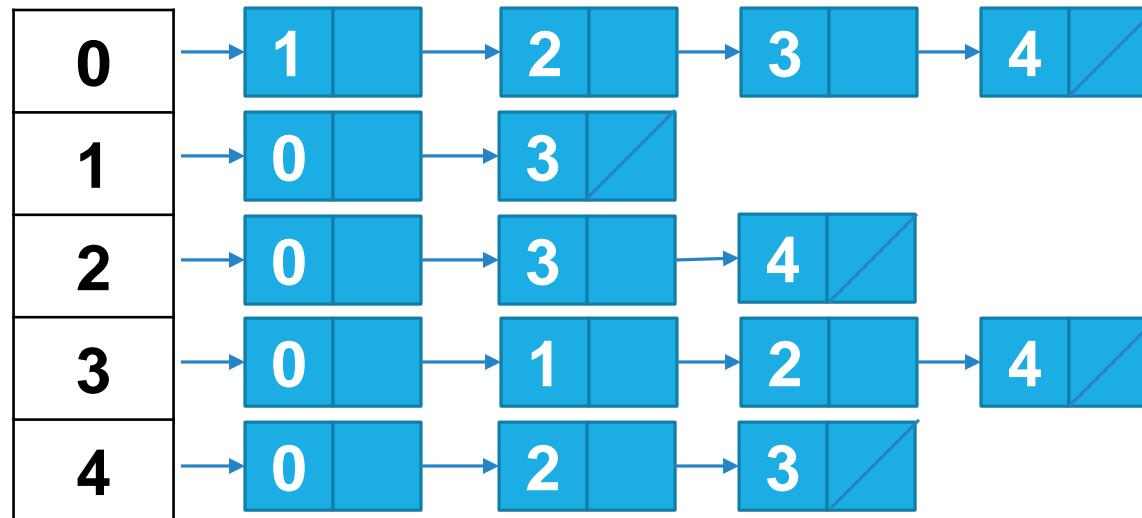
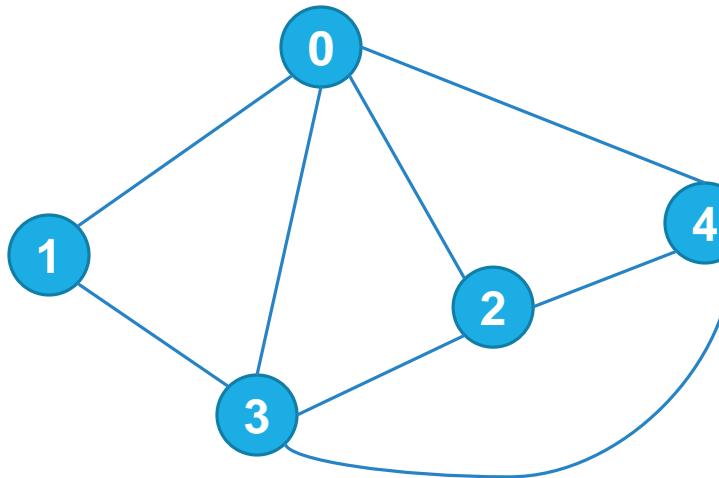
ADJACENCY LIST FOR UNDIRECTED GRAPH.



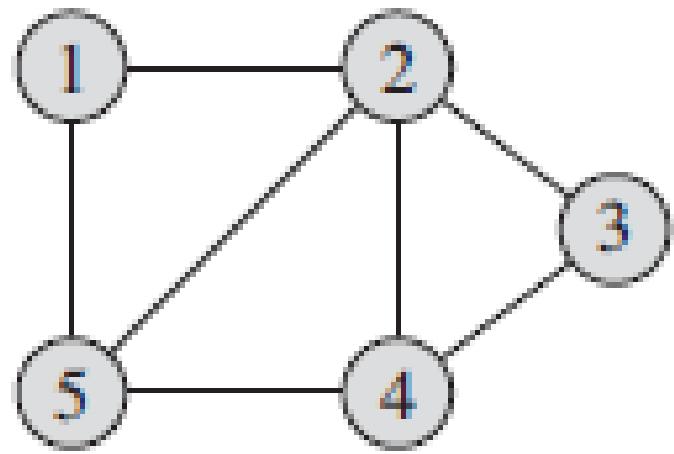
Vertex list

Adjacency list

ADJACENCY LIST: ONE MORE EXAMPLE.

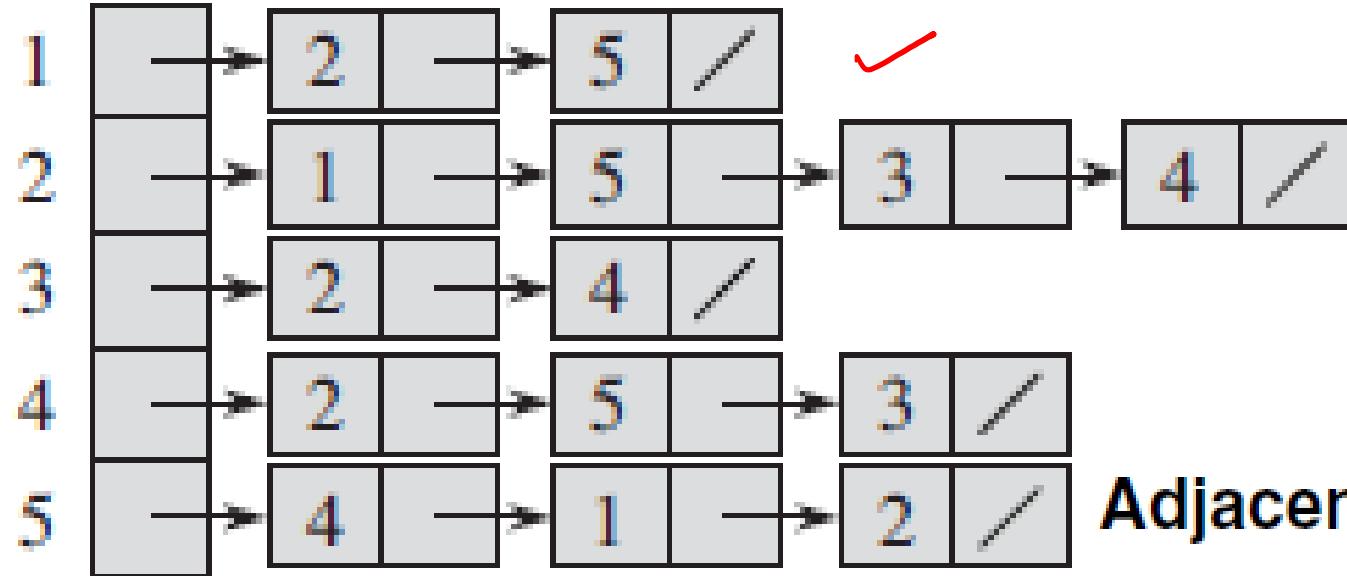


REPRESENTATIONS FOR UNDIRECTED GRAPH: EXAMPLE 1.



1
2
3
4
5

Vertex vector

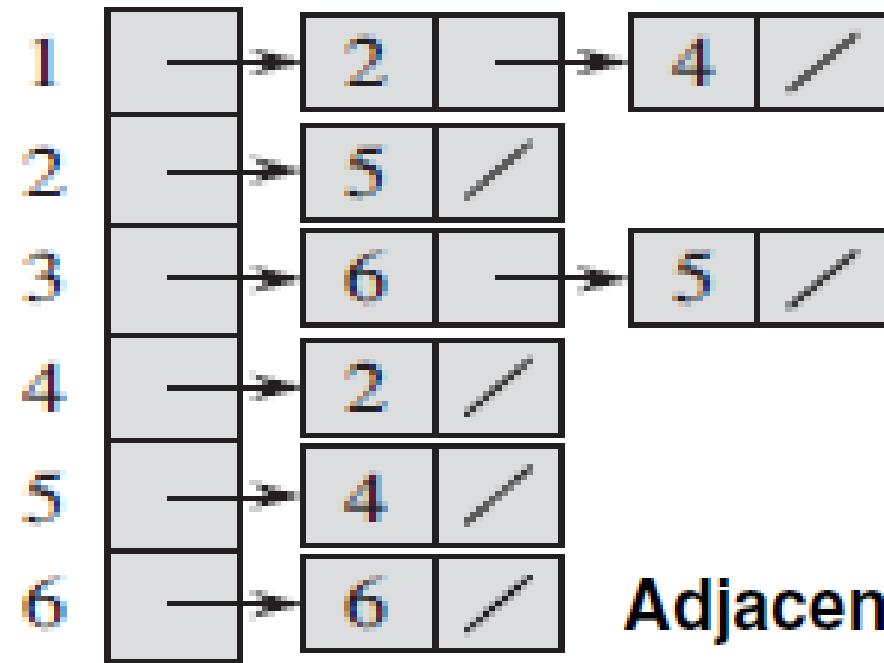
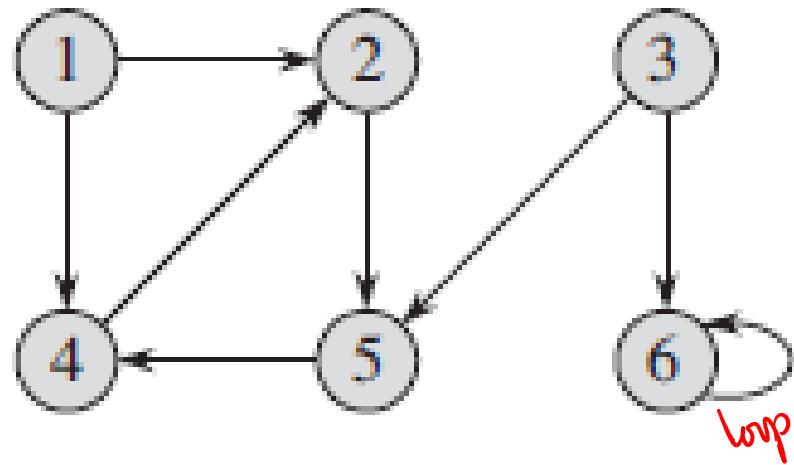


Adjacency list

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

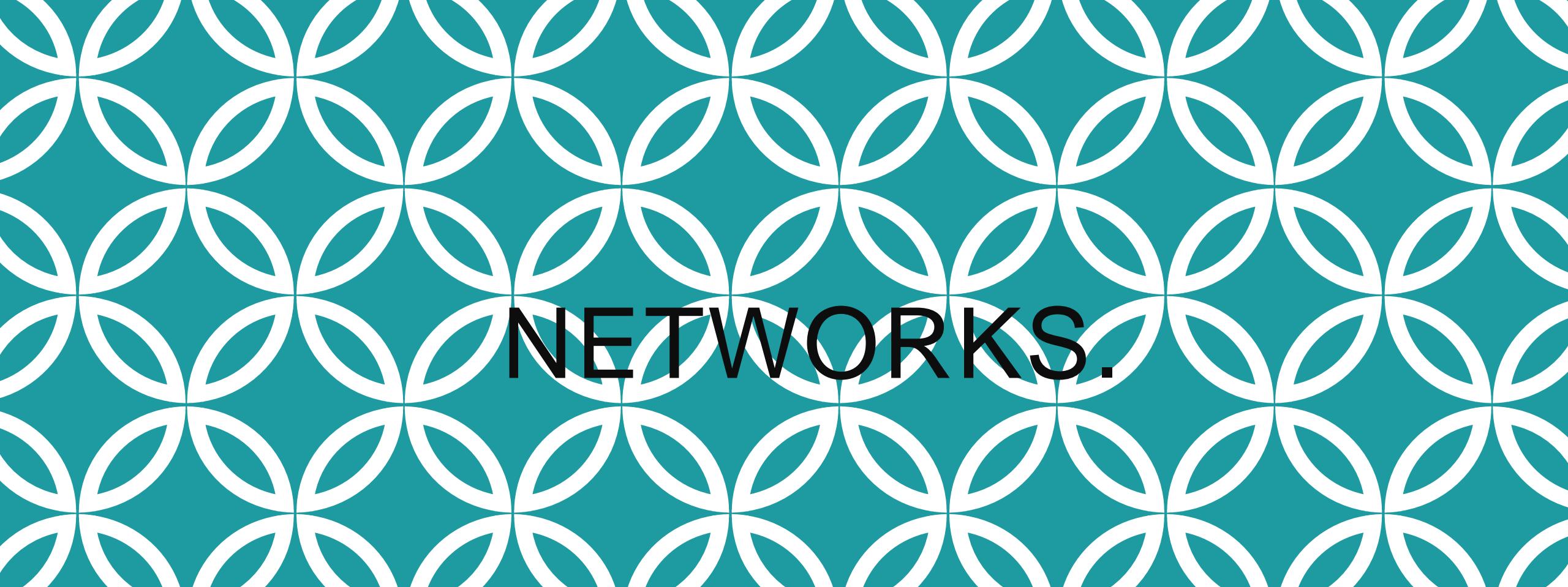
1
2
3
4
5
6

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.

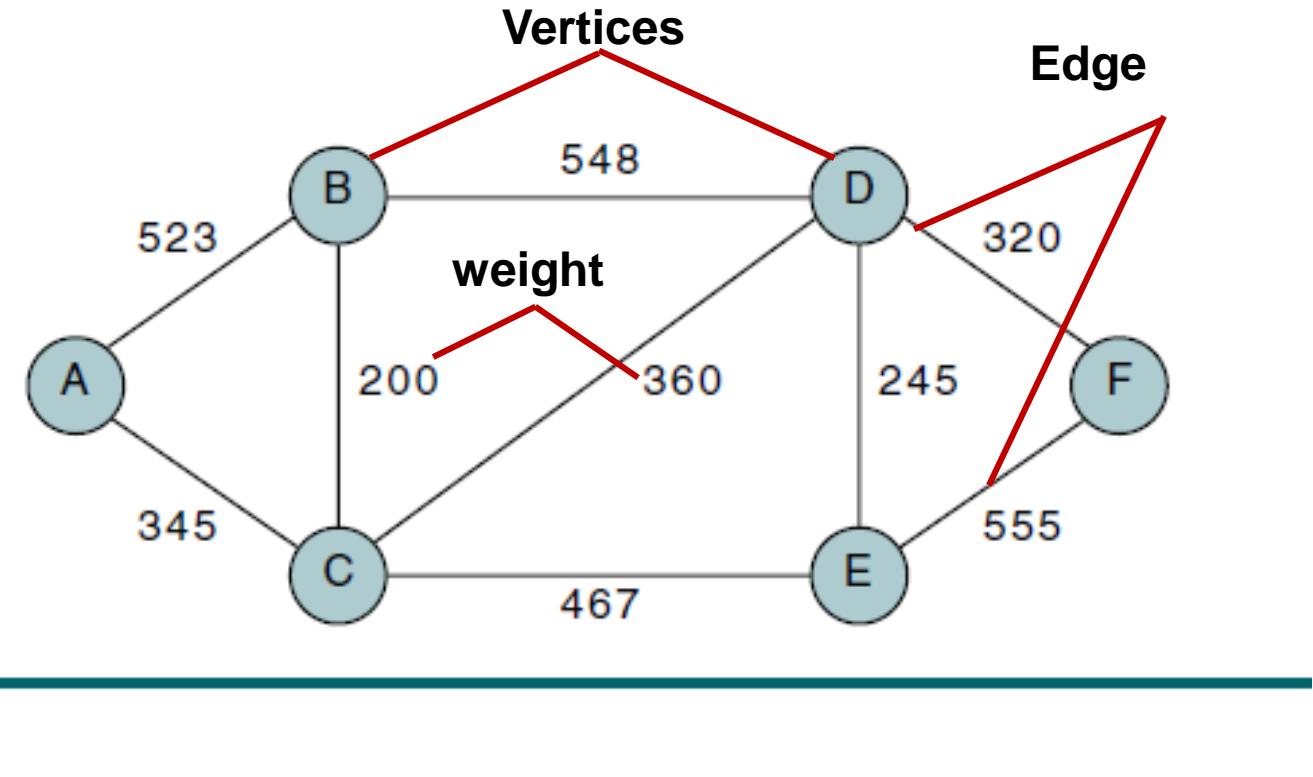
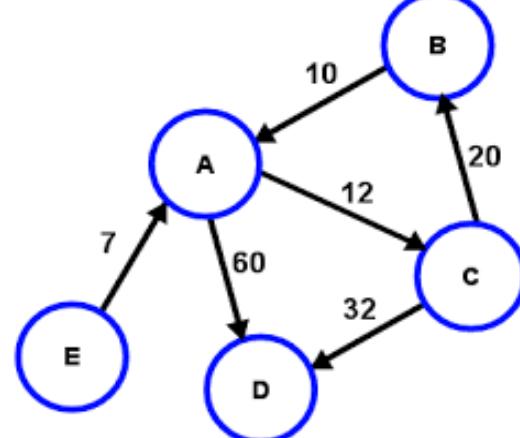


NETWORKS.

A **network** is a **graph** whose **lines** are **weighted** or called **weighted graph**.

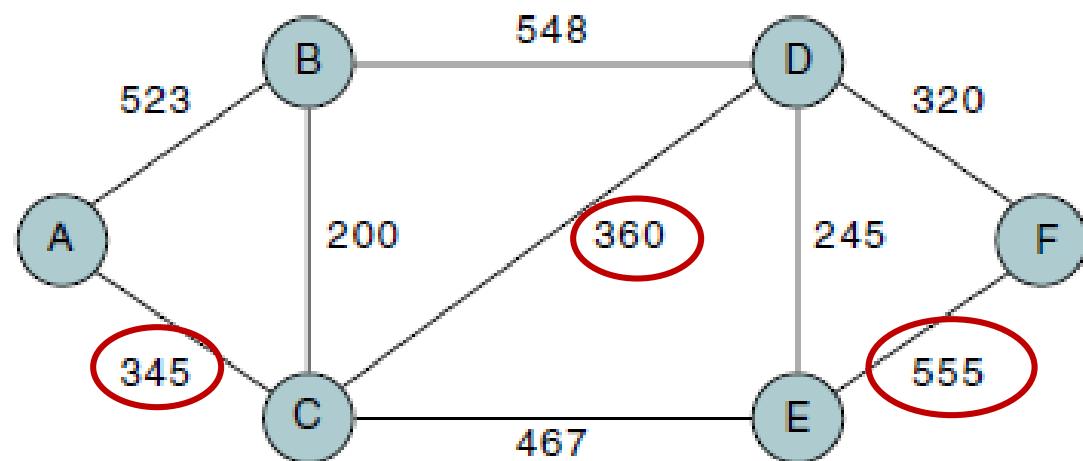
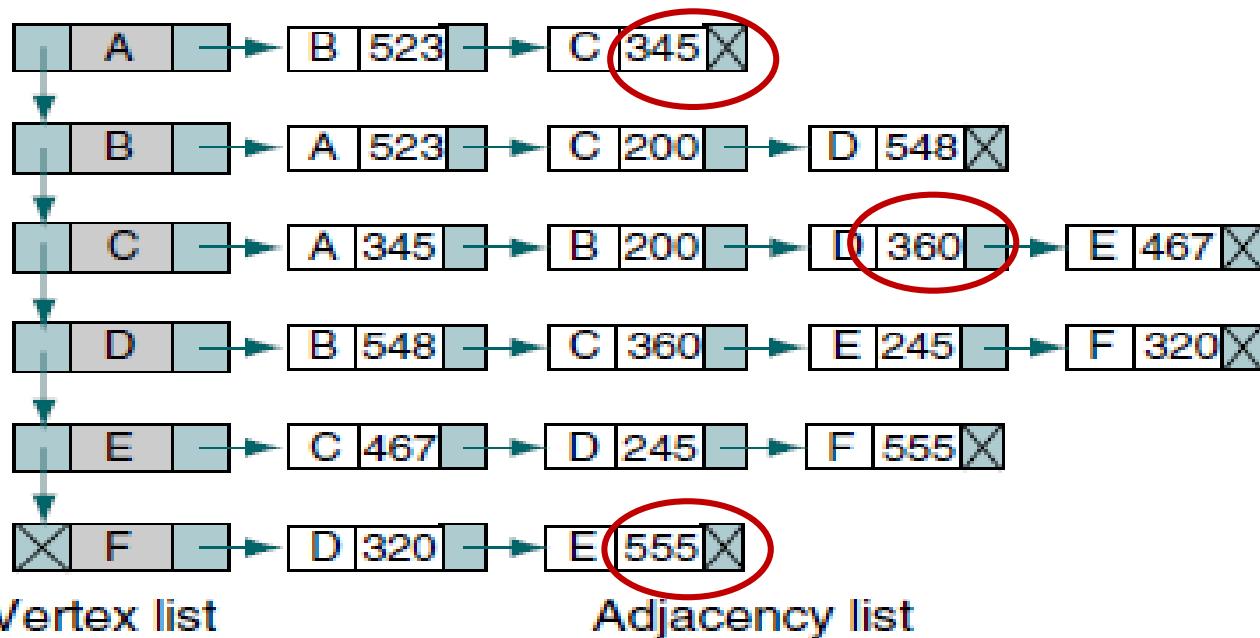
Meaning of the weights depends on the application. For example, an airline might use a graph to represent the routes between cities that it serves. Here,

- **Vertices** represent **cities**, and
- **Edge** is a **route** between 2 cities.
- Edge's **weight** could be:
 - Miles between the 2 cities
 - Price of the flight



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.



Vertex vector

	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

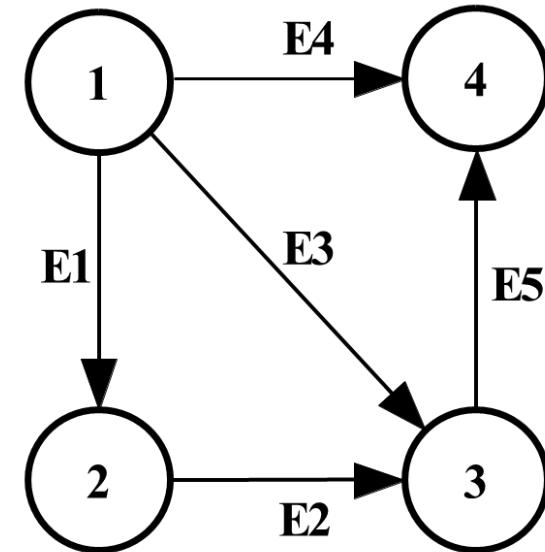
Adjacency matrix

INCIDENCE MATRIX OF A DIGRAPH.

Incidence matrix is a $|E| \times |V|$ matrix $B = (b_{ij})$ such that,

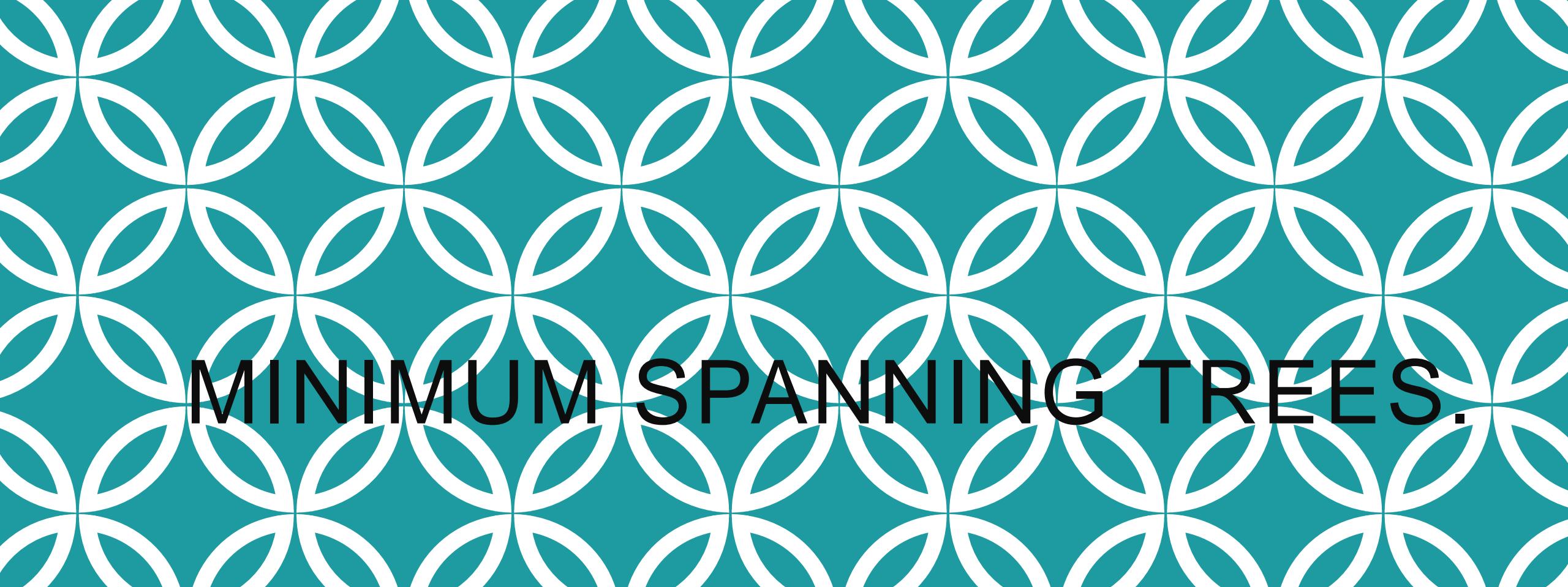
	1	2	3	4
E1	-1	1	0	0
E2	0	-1	1	0
E3	-1	0	1	0
E4	-1	0	0	1
E5	0	0	-1	1

$$b_{ij} = \begin{cases} -1 & \text{if edge } i \text{ leaves vertex } j \\ 1 & \text{if edge } i \text{ enters vertex } j \\ 0 & \text{otherwise} \end{cases}$$



With no self loops

It has one column for each vertex and one row for each edge

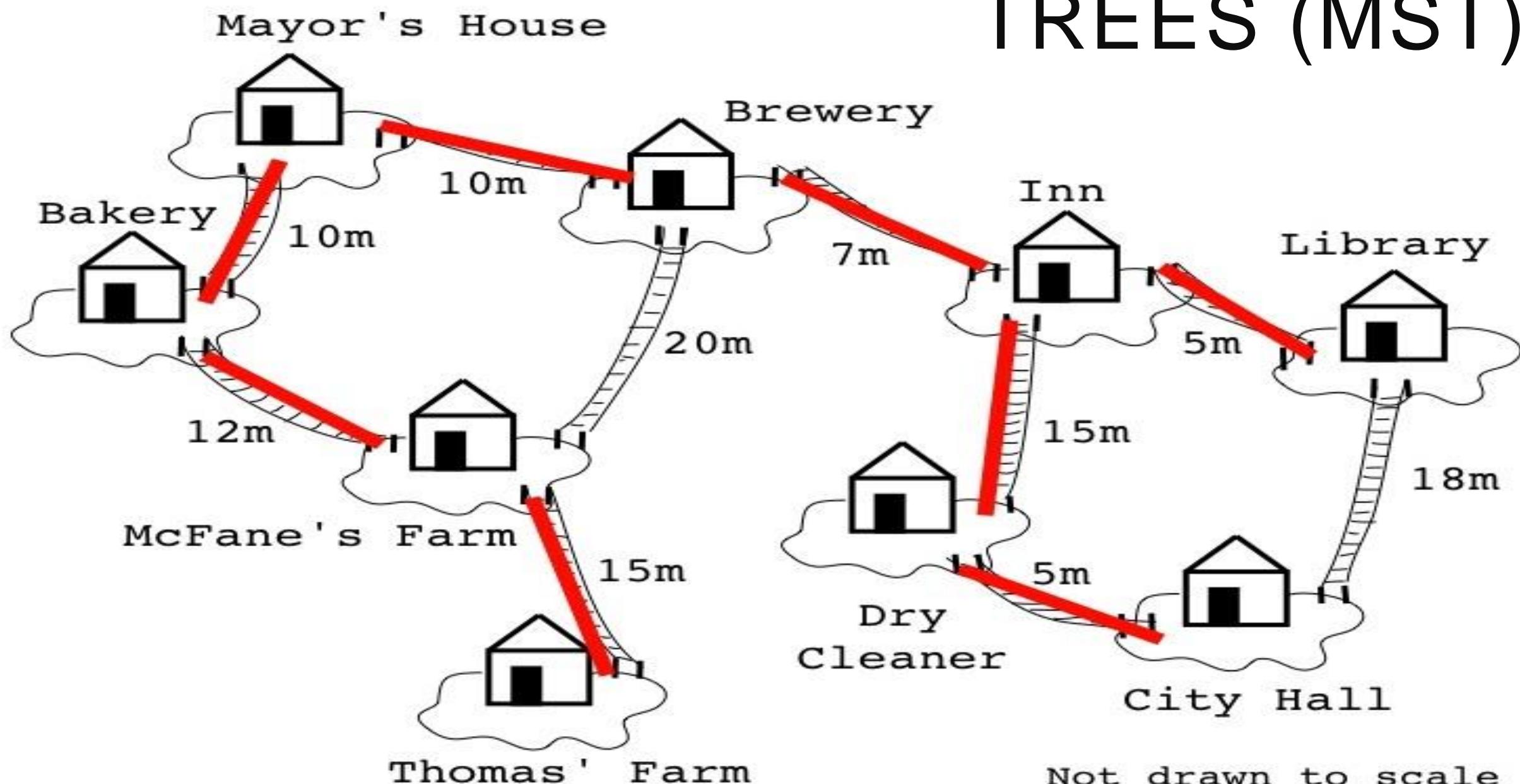


MINIMUM SPANNING TREES.





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.

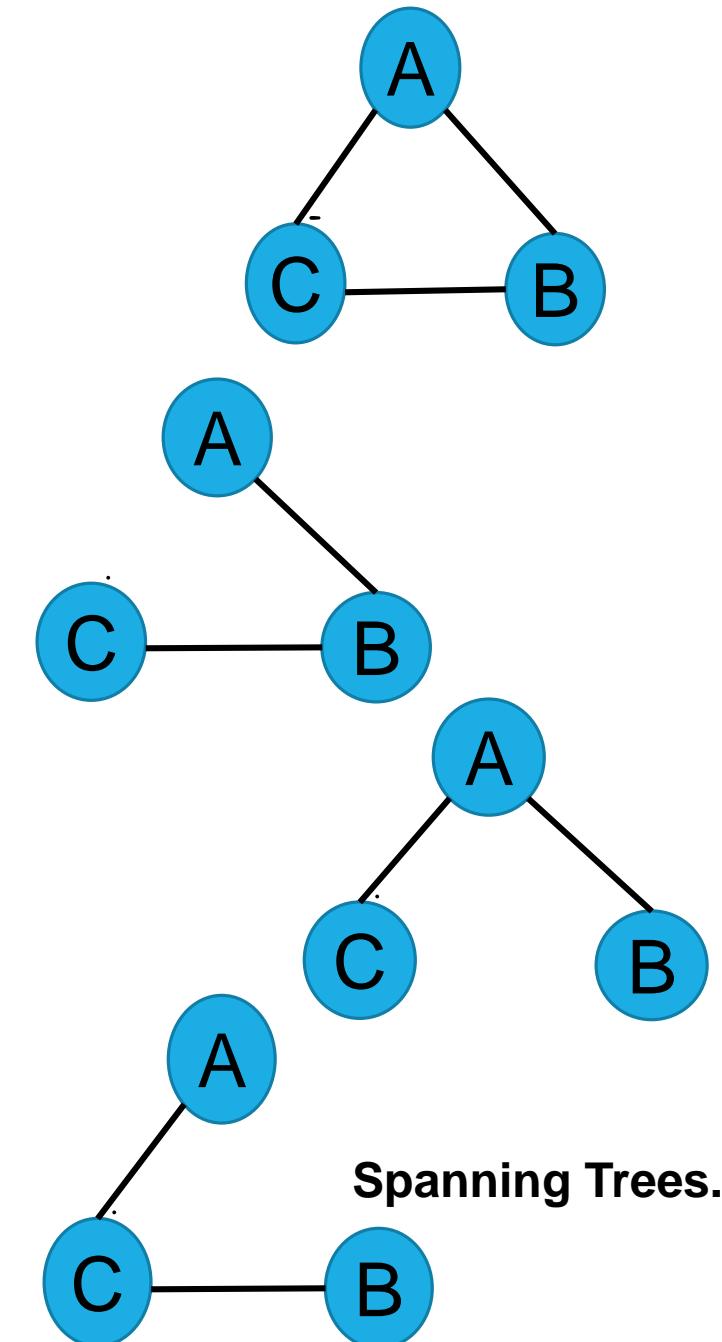
A spanning tree has **$n-1$ edges**; where **n** is the number of nodes (vertices).

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.

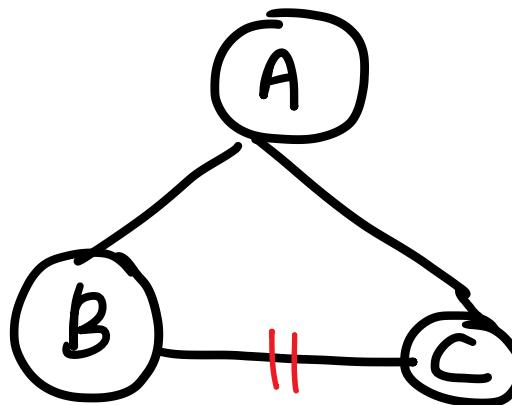
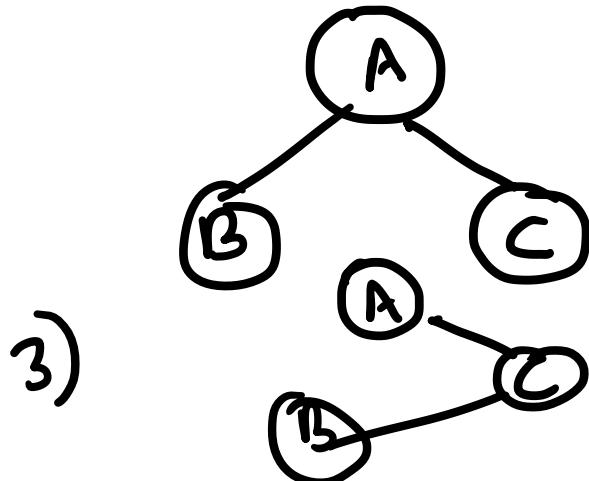
SPANNING TREE.

A graph can be represented by $G(V, E)$ where V is vertex set and E is edge set.

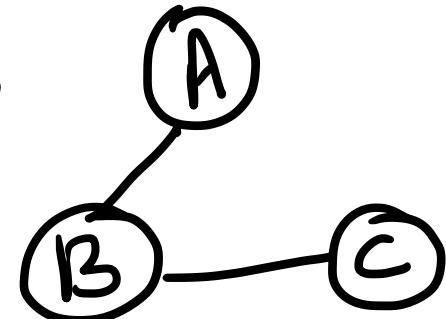
Spanning Tree of this graph $G'(V', E')$ where $V' = V$ and E' is a subset of E ;

Also, $E' = |V| - 1$

1) Remove one edge



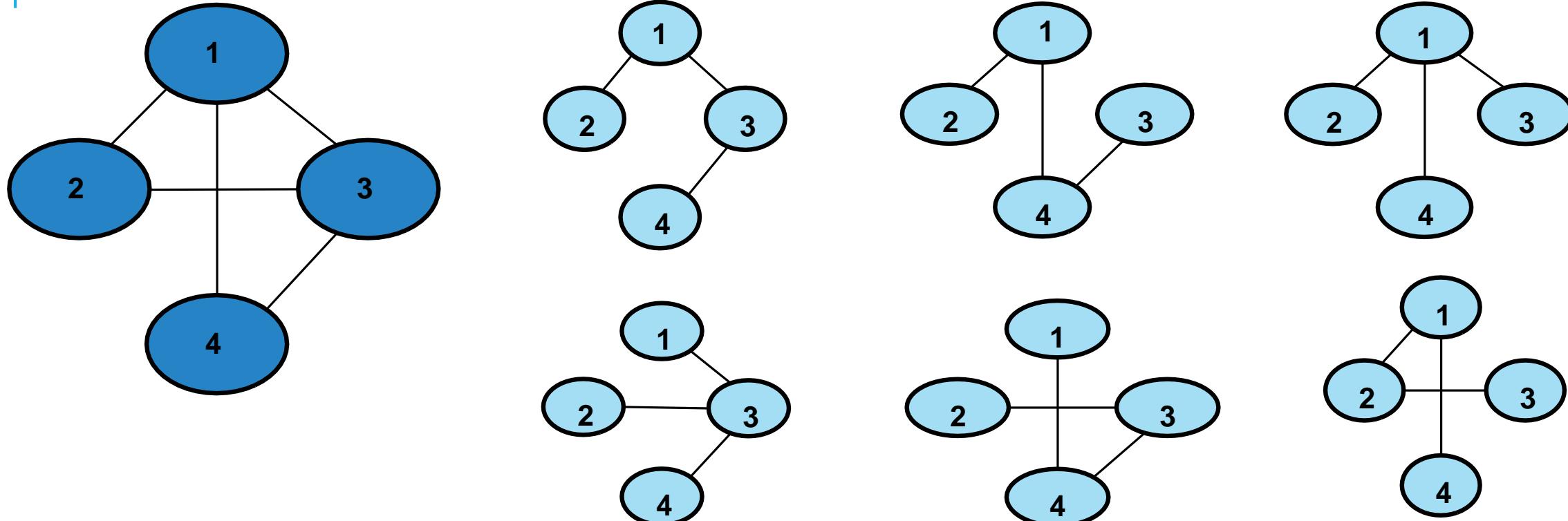
2) Remove another edge



$$n = 3$$

$$\begin{aligned} \text{No. of spanning trees} &= 3^{3-2} \\ &= 3^1 = 3 \end{aligned}$$

SPANNING TREE – CONNECTED.



All edges are assumed to have **same weight** (cost)

MINIMUM SPANNING TREE.

If the edges have different (non-negative) weight (cost), **a minimum-cost spanning tree** can be constructed.

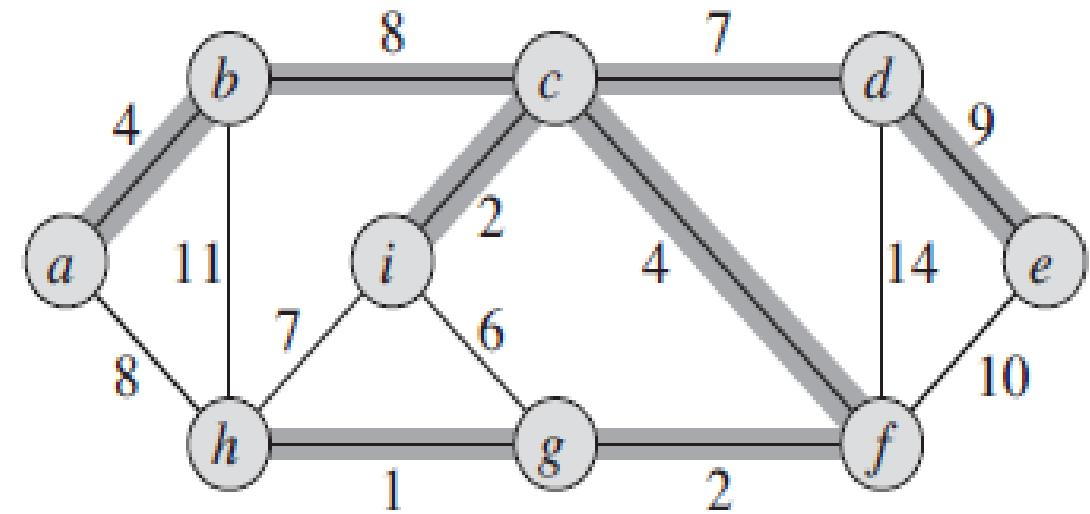
Let $G = (V, E)$ be an weighted, connected and undirected graph. For each edge $(u, v) \in E$, the weight $w(u, v)$ specifies the cost to connect u to v .

The acyclic subset (tree) $T \in E$ 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.

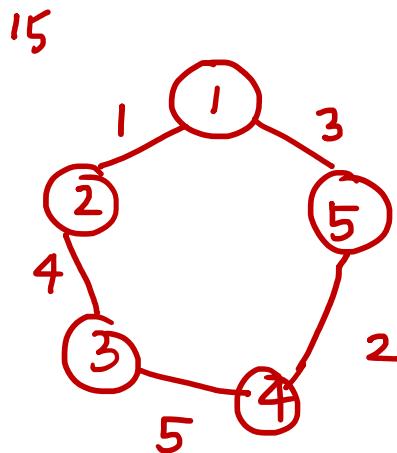
The weights on edges are shown, and the edges in a minimum spanning tree are shaded.



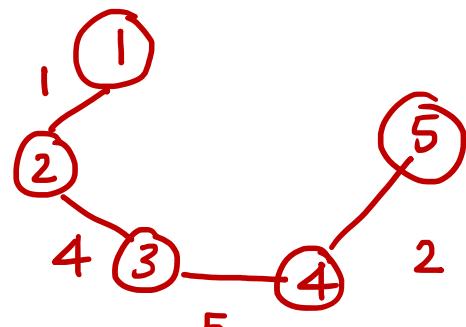
The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

MINIMUM SPANNING TREE OR MST.

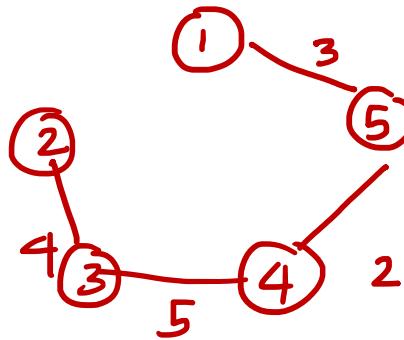
If the edges have different (non-negative) weight (cost), a minimum-cost spanning tree can be constructed.



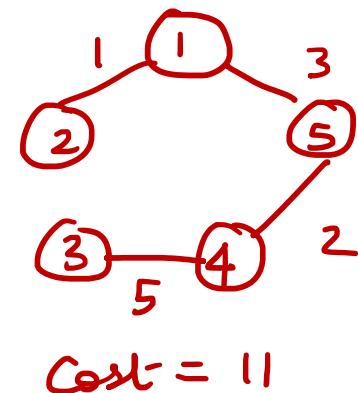
Start at any vertex, remove one edge



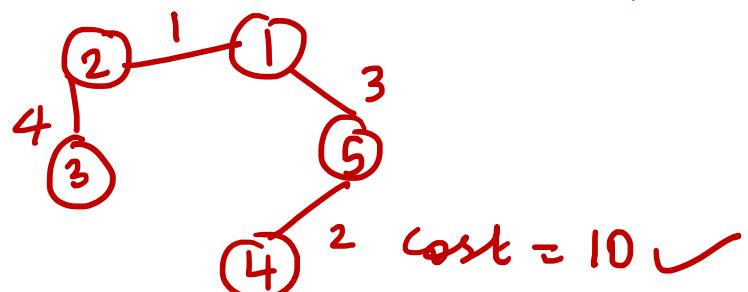
Cost = 12



Cost = 14 ✓



Cost = 11

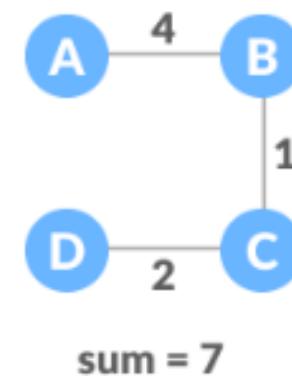
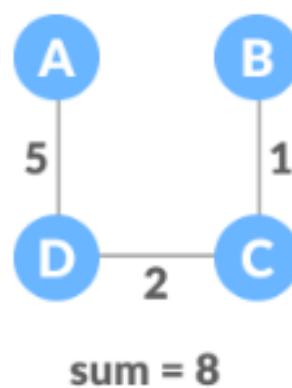
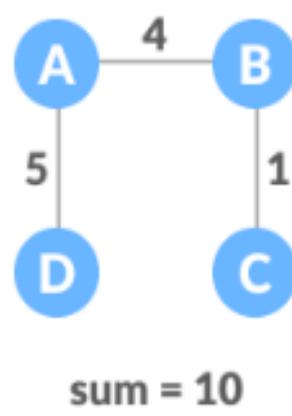
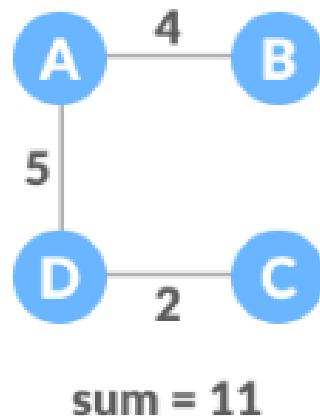
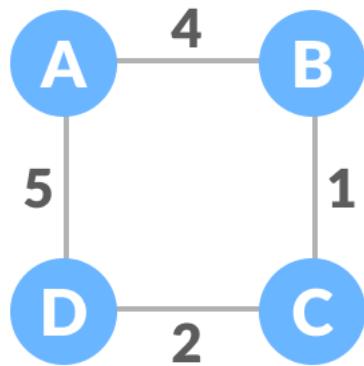


MST

Cost = 10 ✓

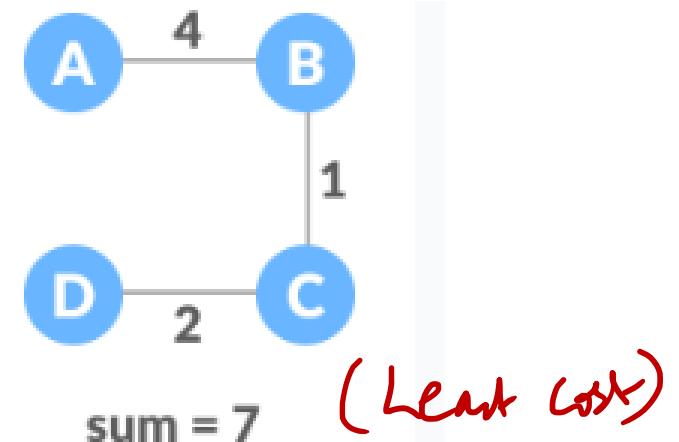
MST WITH COSTS.

The cost of the spanning tree is the sum of the weights of all the edges in the tree.

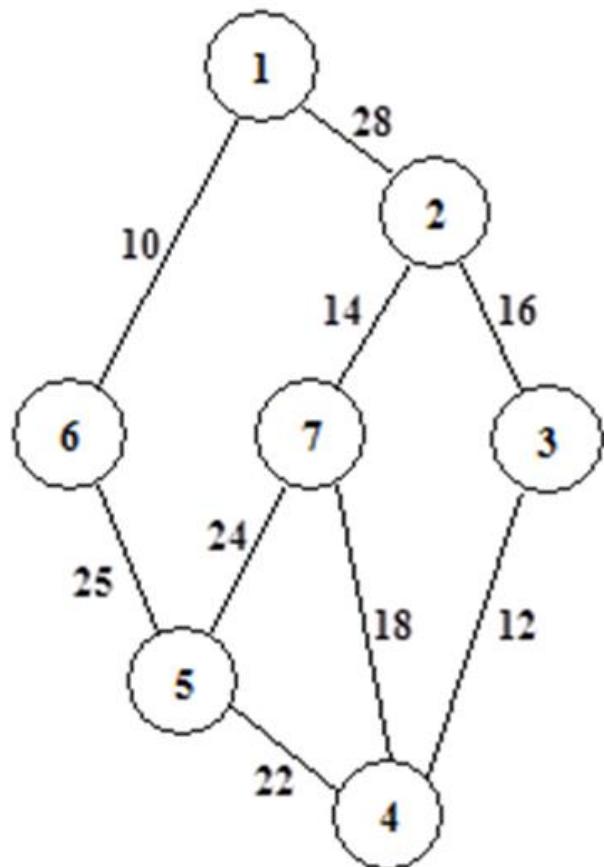


12

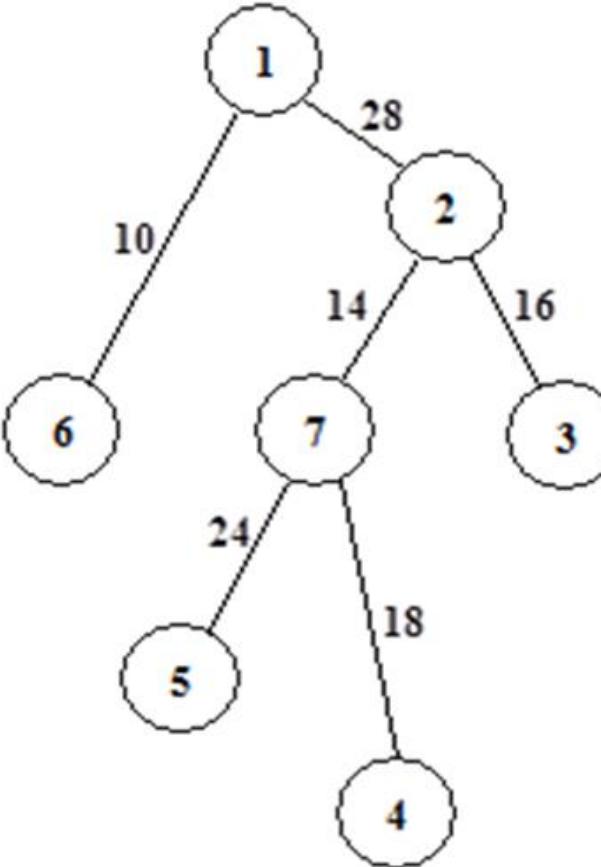
MST from the above spanning trees is:



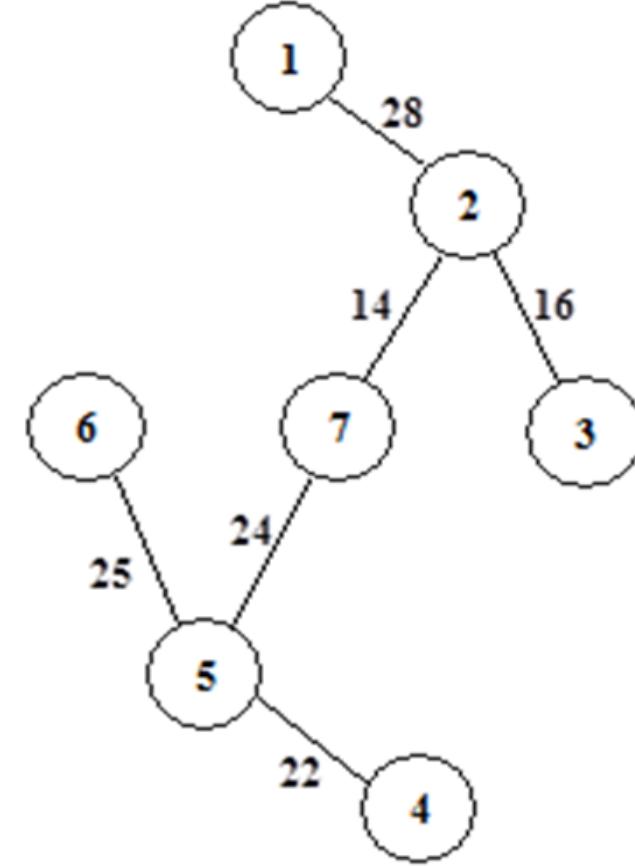
MINIMUM SPANNING TREE OR MST.



Graph G



Spanning Tree (Cost =110)



Spanning Tree (Cost =129)

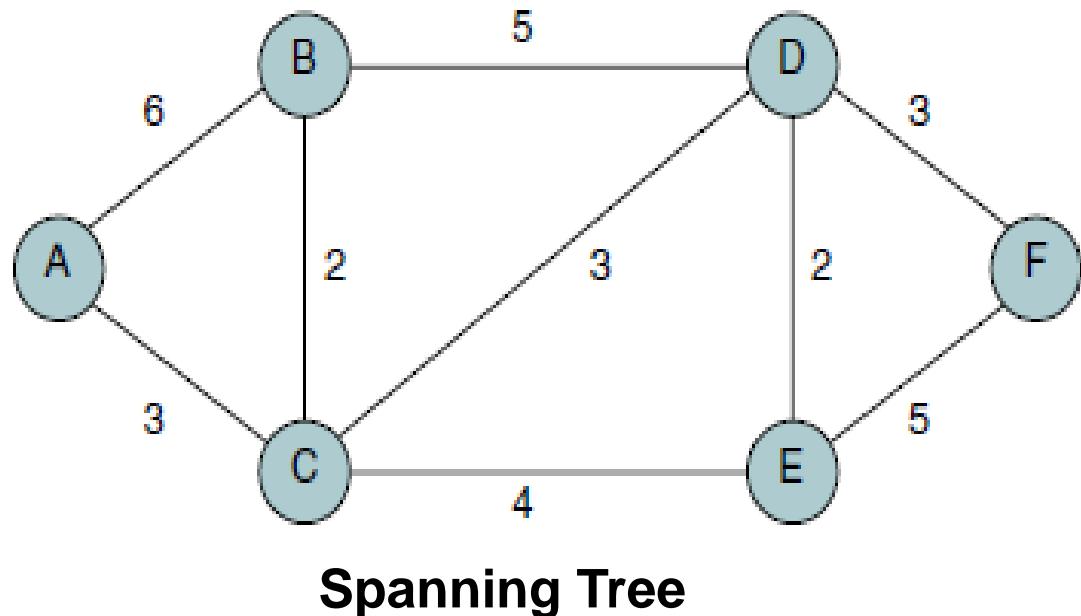
MINIMUM SPANNING TREE APPLICATIONS.

1. Given a ***network of computers***, a 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.
2. ***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.
3. 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.

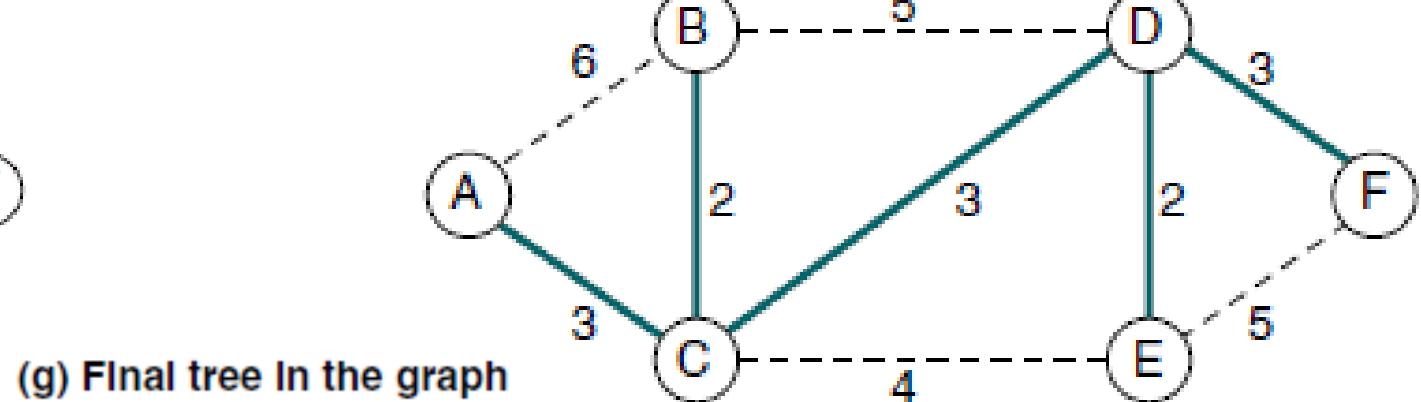
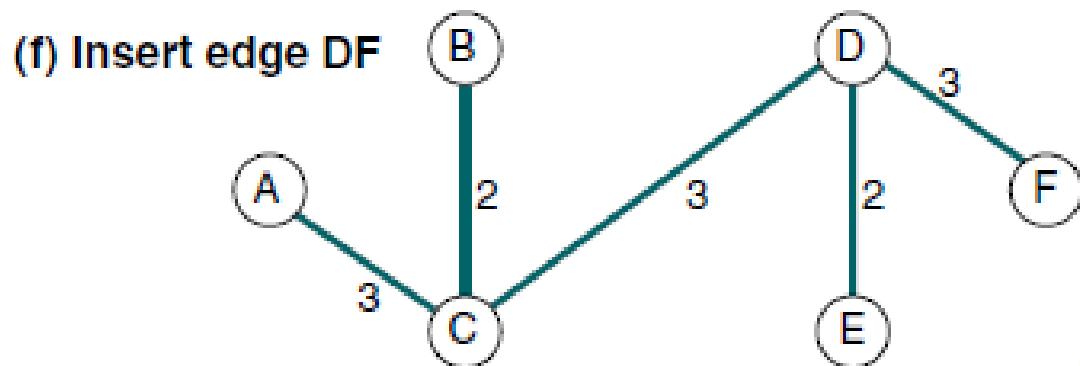
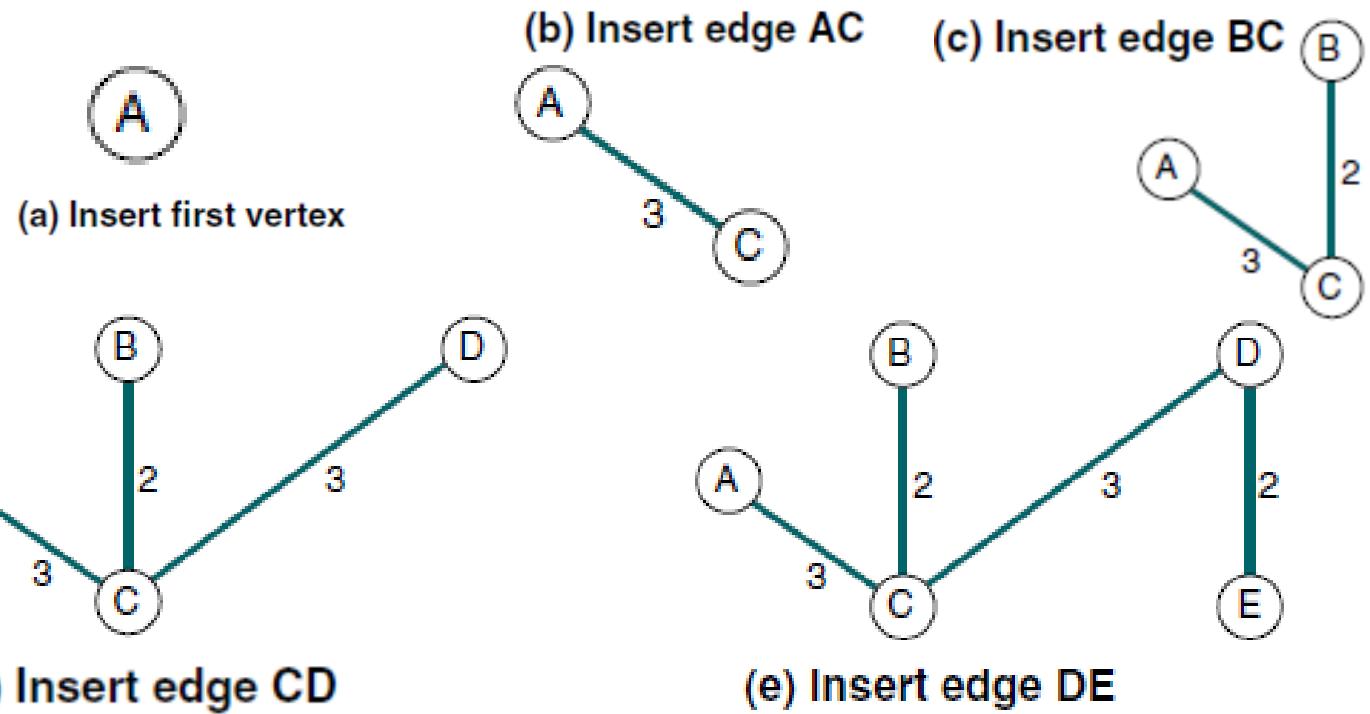
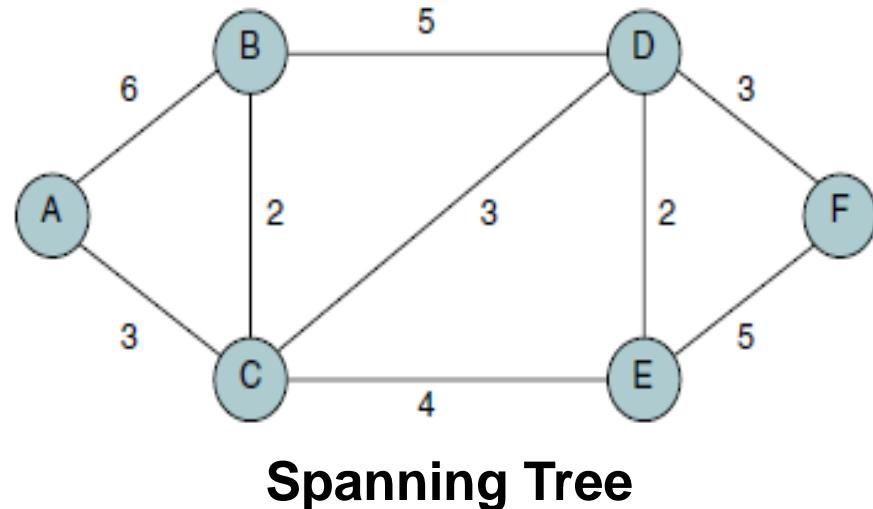


Rules:

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

SIMPLE STEPS FOR IMPLEMENTING KRUSKAL'S ALGORITHM.

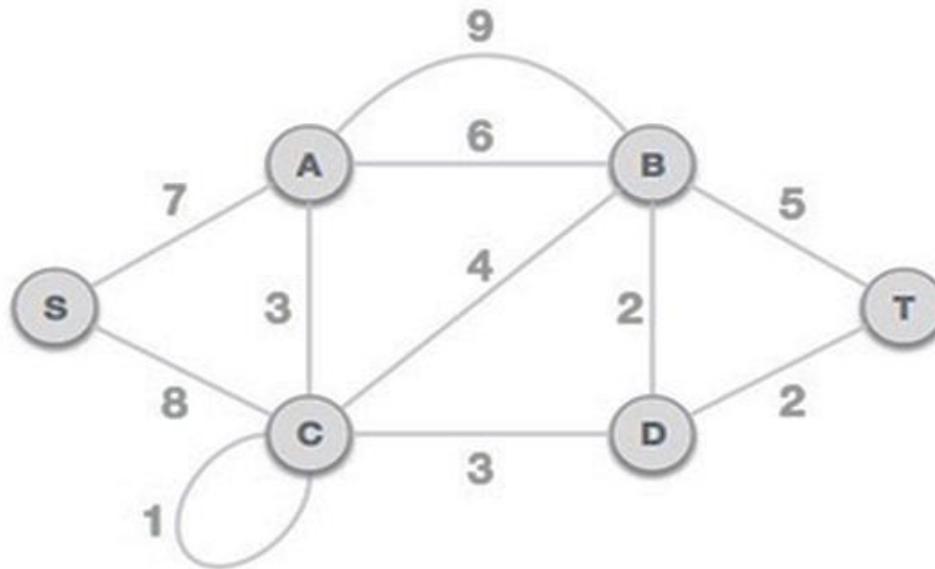
Sort all the edges from low weight to high

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.

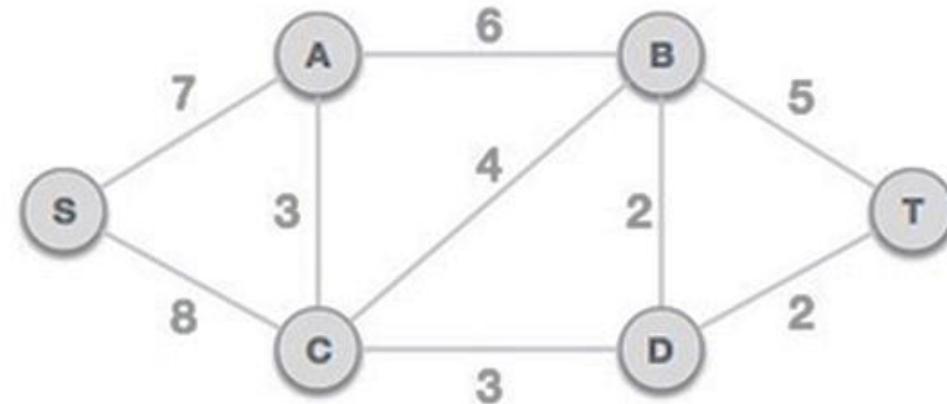
Keep adding edges until we reach all vertices.

KRUSKAL'S ALGORITHM EXAMPLE.

Consider the graph G:-



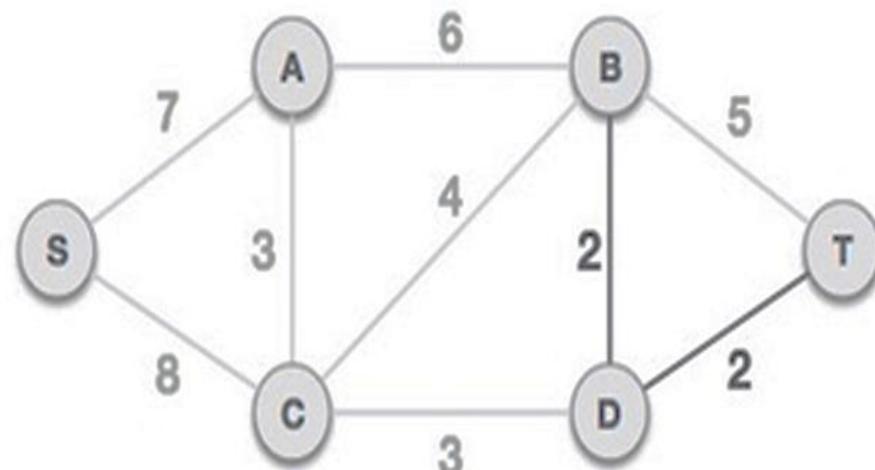
Step 1: Remove all loops and Parallel Edges



Step 3 - Add the edge which has the least weightage

Step 2: Arrange all edges in increasing order of weights.

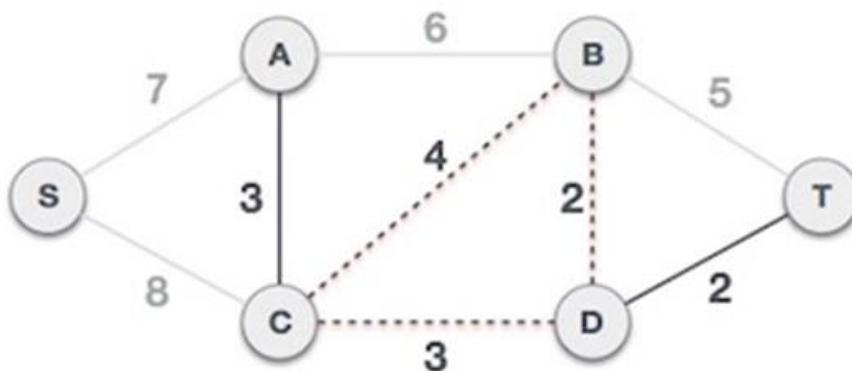
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8



When edges are selected, edges that create cycles have to be avoided. If there is a cycle present, ignore the edge.

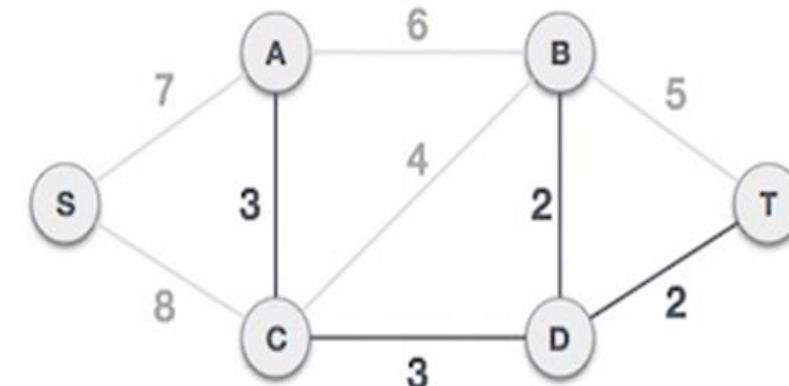
KRUSKAL'S ALGORITHM EXAMPLE.

Step 4: Next cost is 3, and associated edges are A,C and C,D

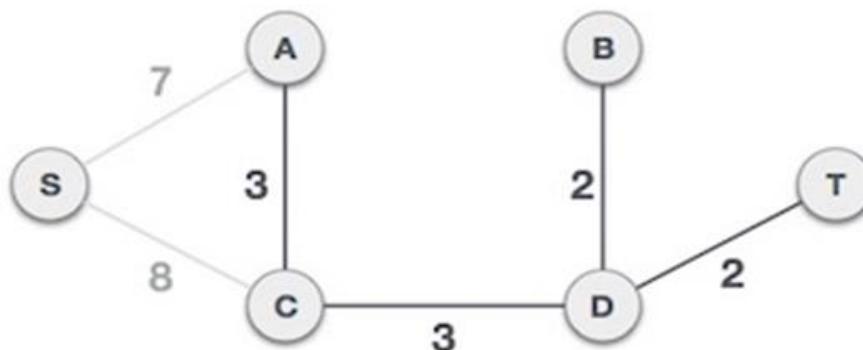


If a graph has "n" vertices, , then the MST must contain n vertices and (n-1) edges

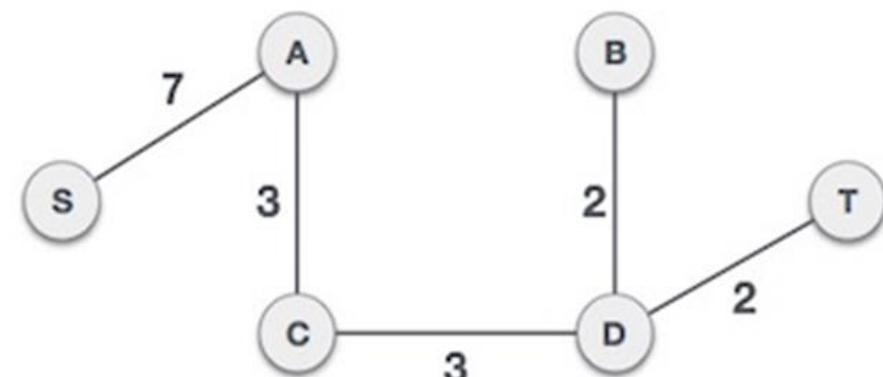
Step 5: Next cost is 4, and it will create a circuit in the graph.



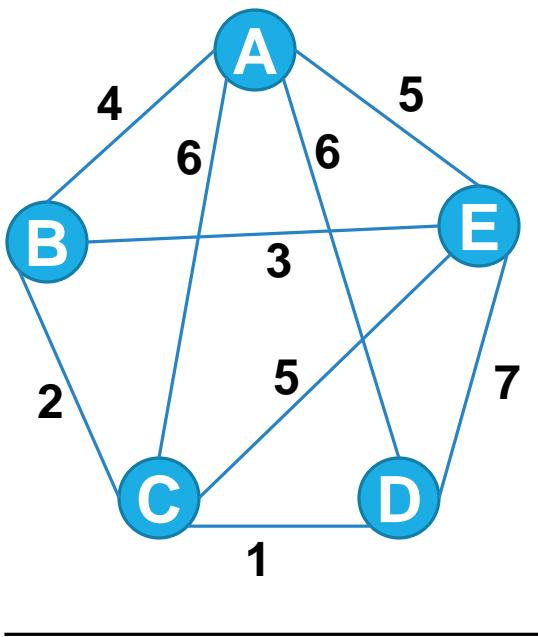
Step 6: Between the two least cost edges 7 and 8, add the edge with cost 7.



Step 7: All vertices of the graph have included and a minimum cost spanning tree created. Therefore, **Cost of MST=17**



KRUSKAL'S ALGORITHM

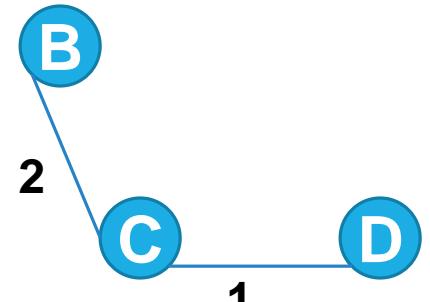


Step 1: Taking min edge (C,D)

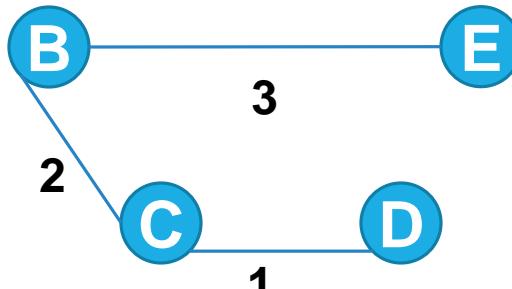


C, D	B, C	B, E	A, B	E, C	A, E	C, A	A, D	E, D
1	2	3	4	5	5	6	6	7

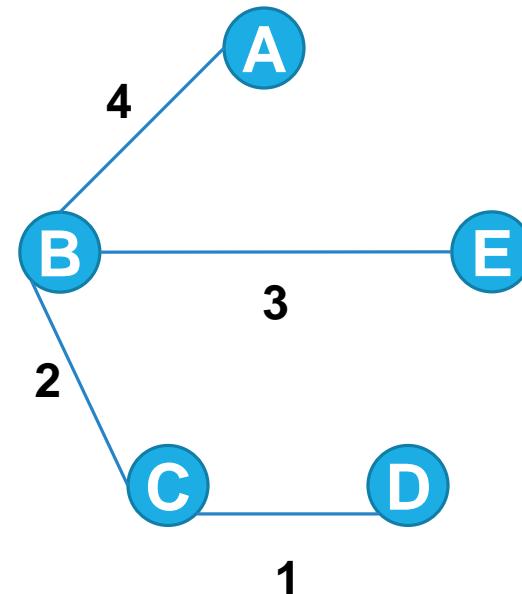
Step 2: Taking next min edge (B,C)



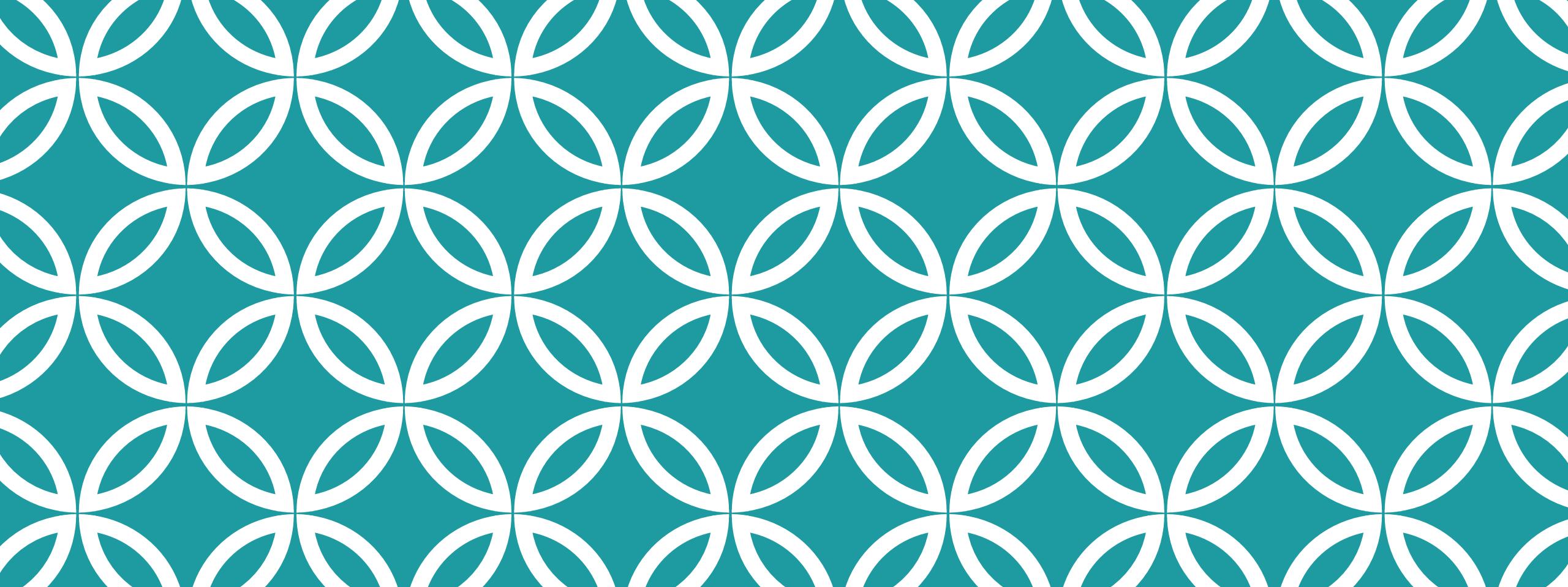
Step 3: Taking next min edge (B,E)



Step 4: Taking next min edge (A,B)



so we obtained minimum spanning tree of cost:
 $4 + 2 + 1 + 3 = 10$



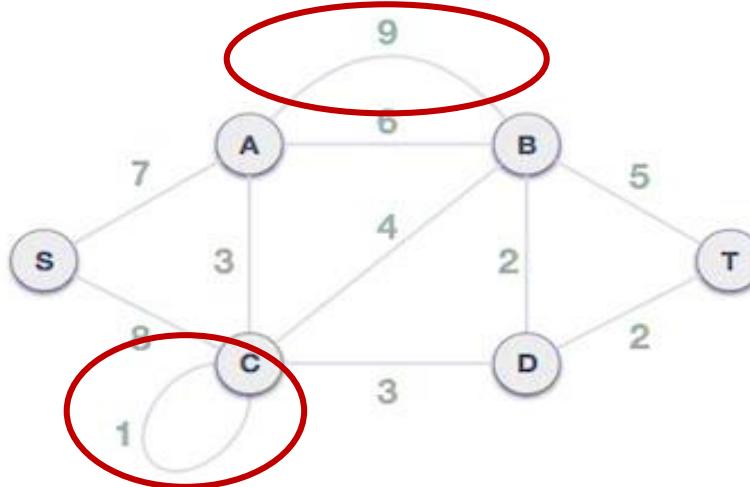
PRIM'S ALGORITHM.

PRIM'S ALGORITHM STEPS.

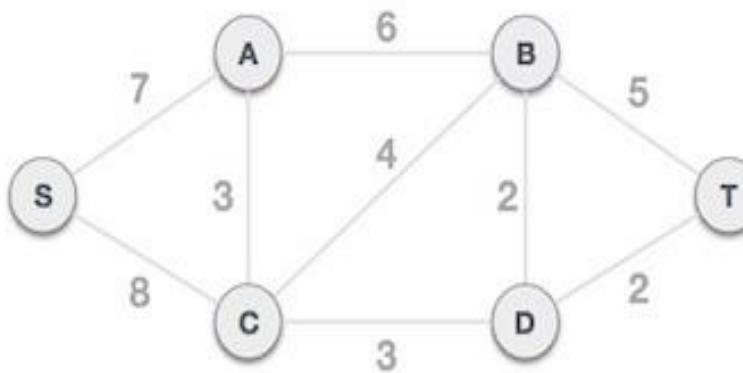
1. Keep a track of all the vertices that have been visited and added to the spanning tree.
2. Initially the spanning tree is empty.
3. Choose a random vertex, and add it to the spanning tree. This is **the root node**.
4. Add a new vertex, say **vx**, such that **vx** is not in the already built spanning 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:-



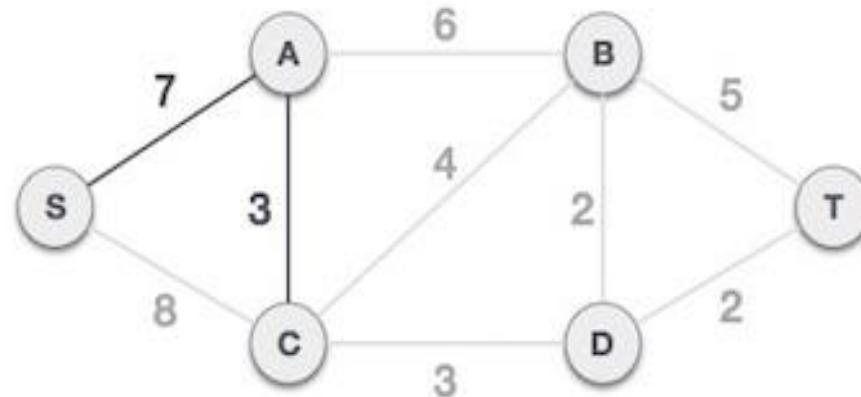
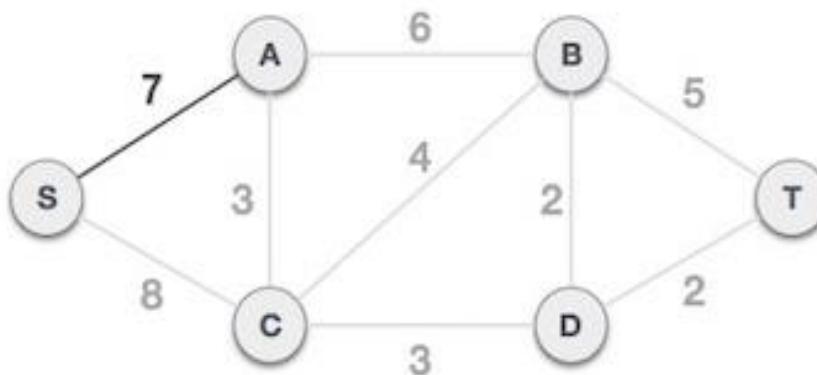
Step 1: Remove all loops and Parallel Edges



Step 2: Choose any arbitrary node as root node
Here choose S node as the root node

Step 3: Check outgoing edges and select the one with less cost. Choose the edge S,A. Why?

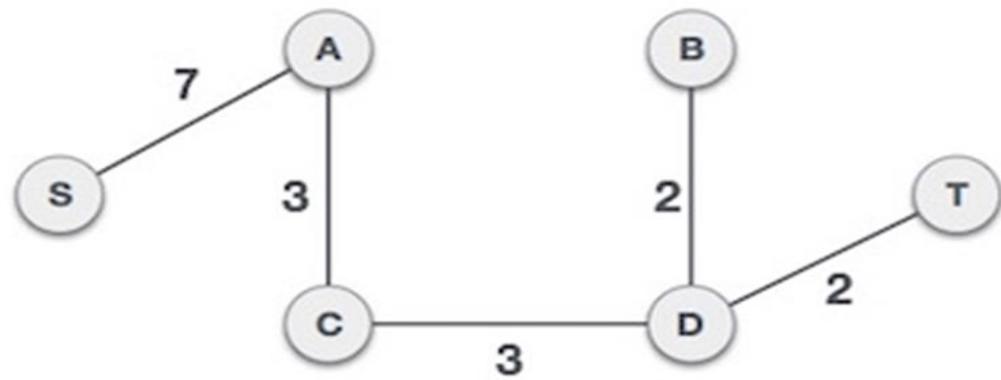
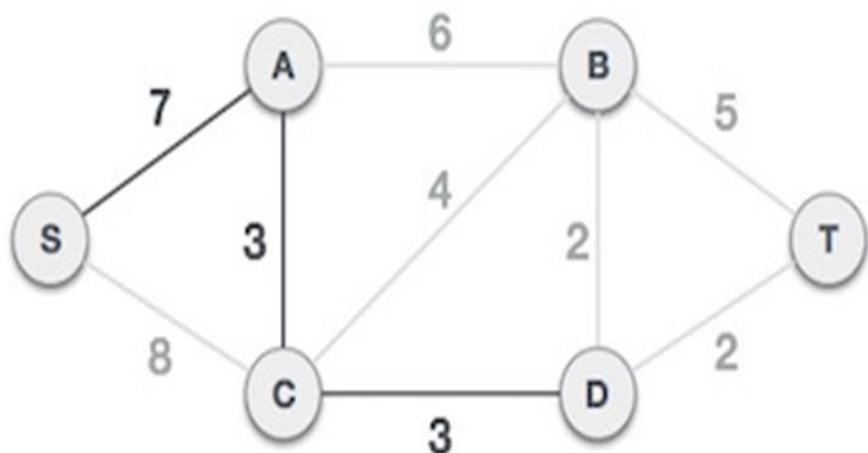
Step 4: The tree S-7-A is treated as one node. Check for all outgoing edges. Select the one with lowest cost



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.

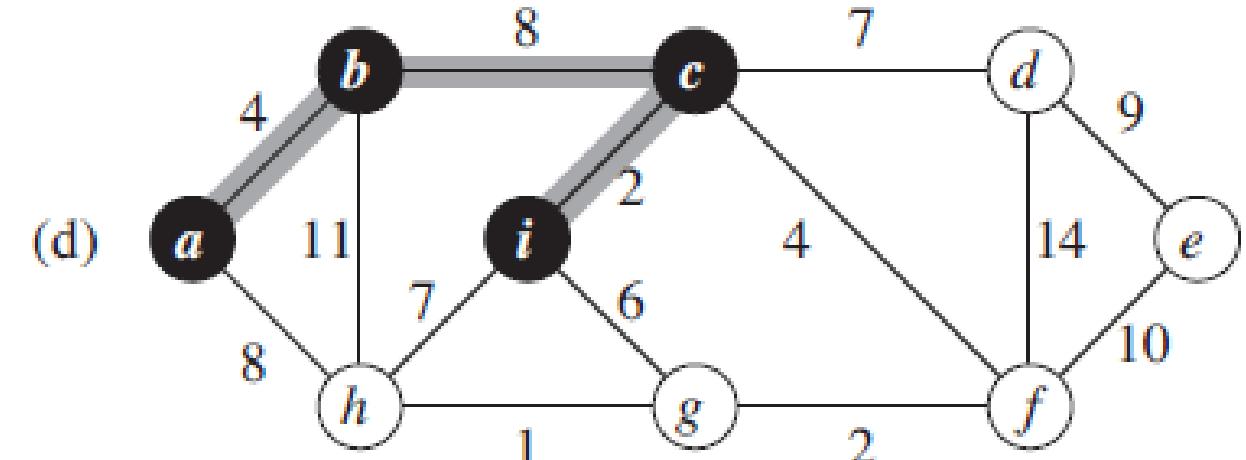
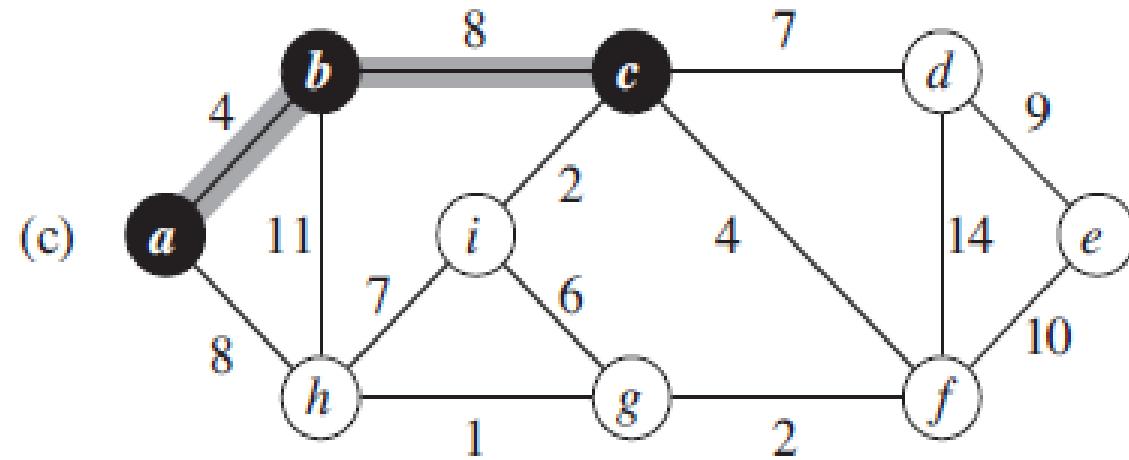
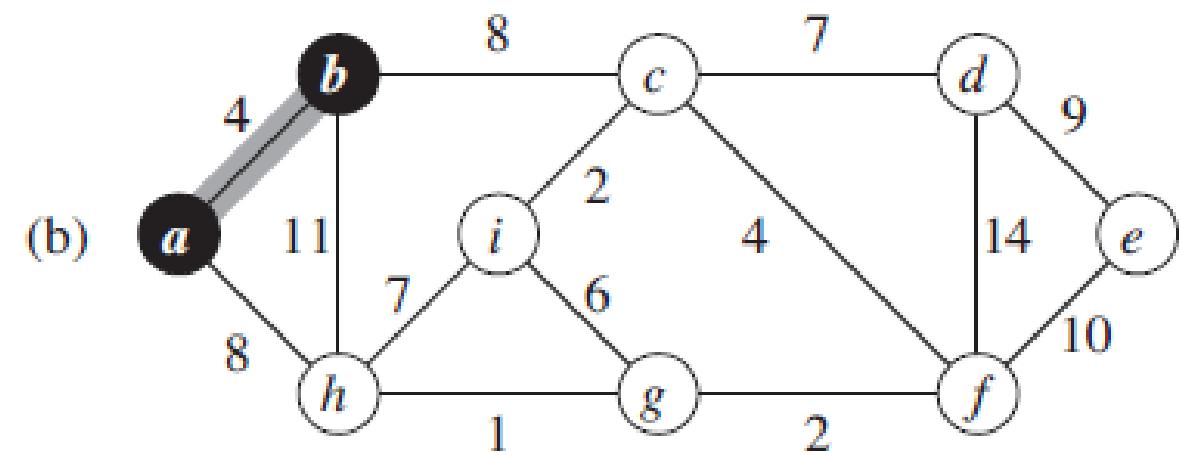
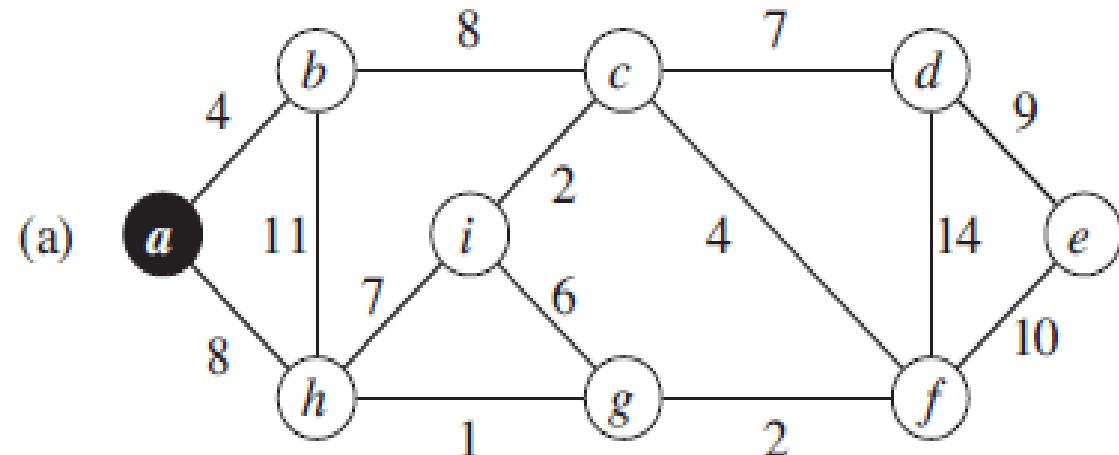
Step 6: All vertices of the graph have been included in the MST, which means the total edge weight or cost is minimum.



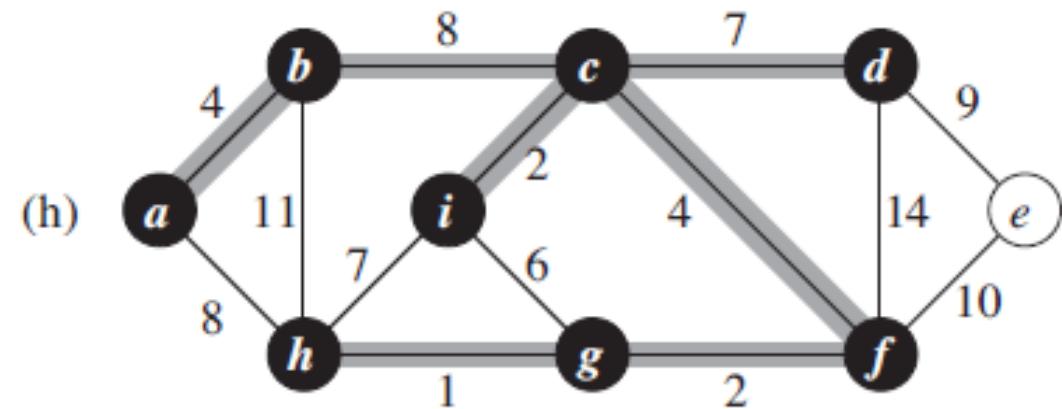
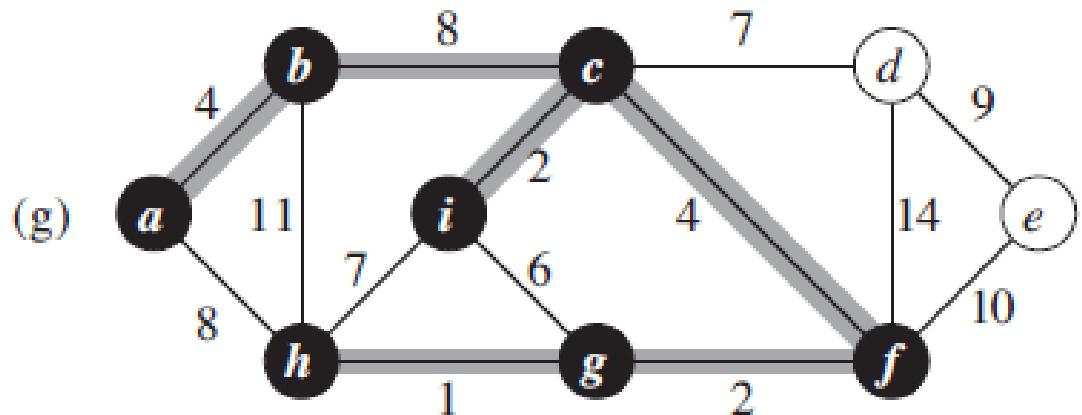
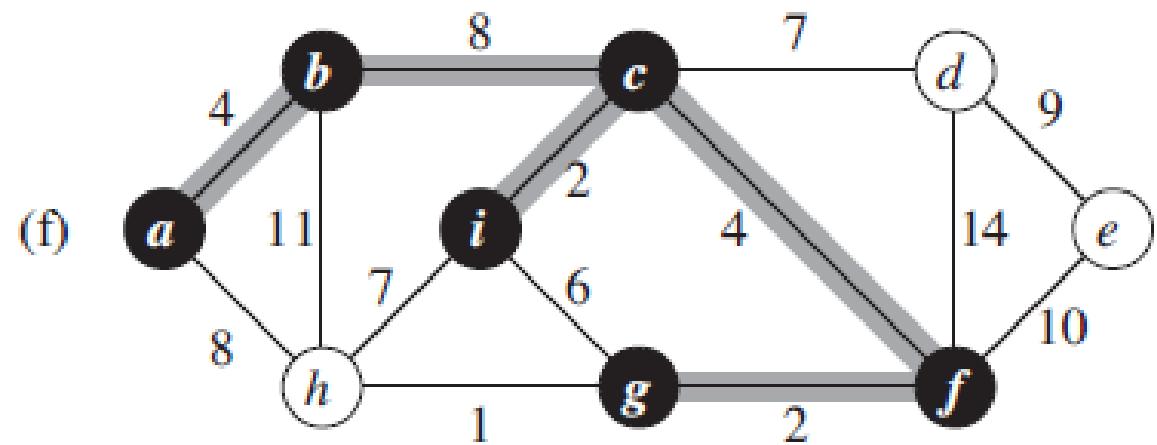
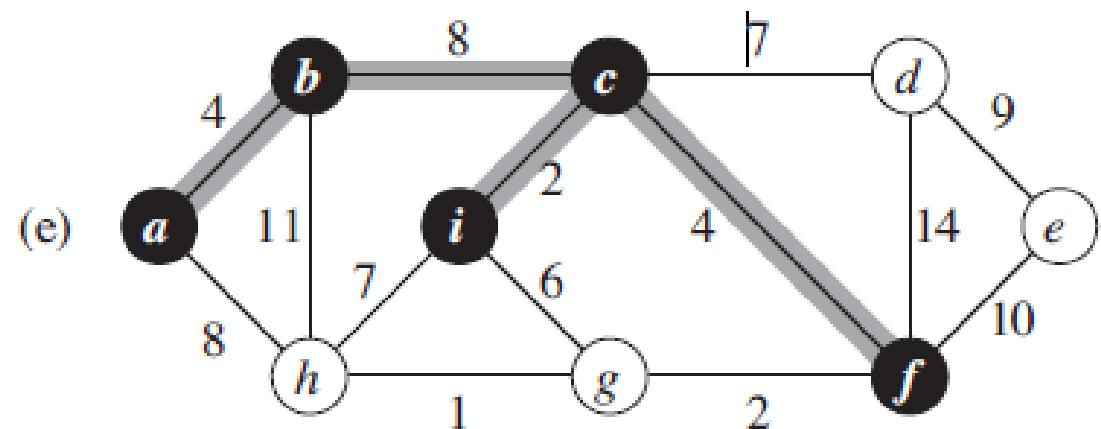
Cost of min. spanning tree

$$7+3+3+2+2 = 17$$

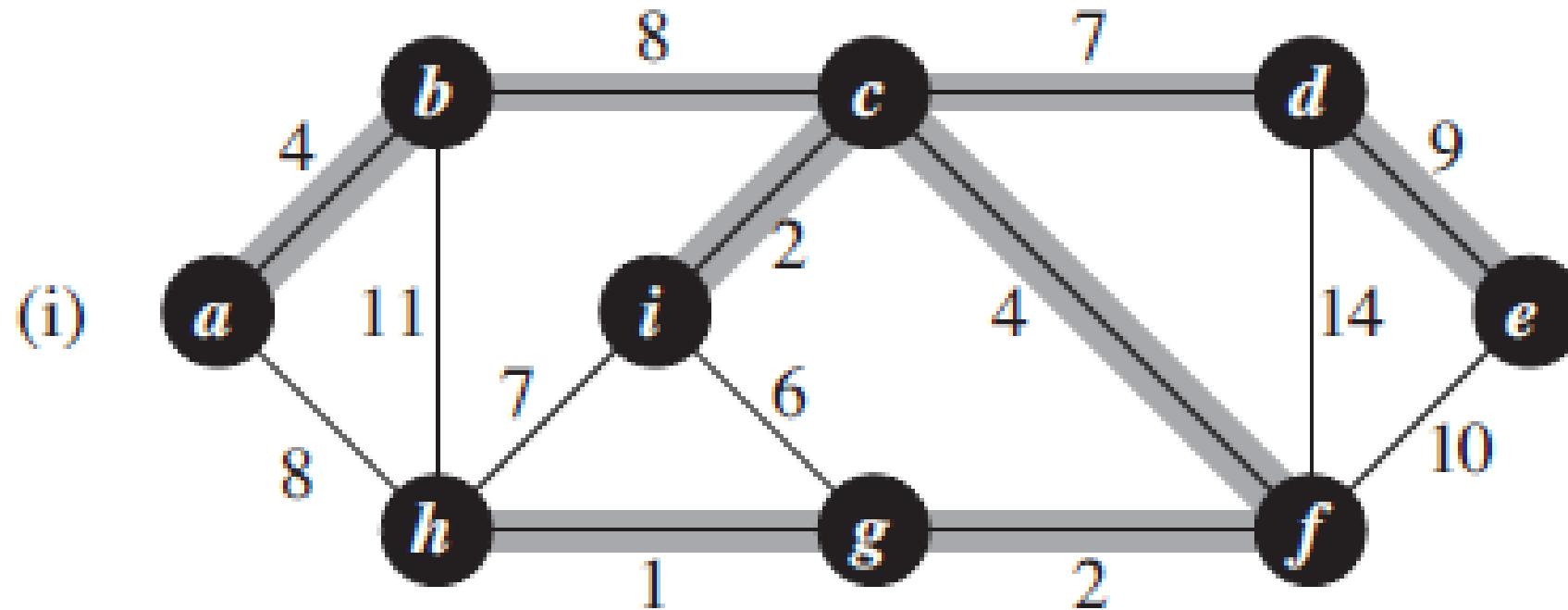
PRIM'S ALGORITHM EXAMPLE 2-1.



PRIM'S ALGORITHM EXAMPLE 2-2.



PRIM'S ALGORITHM EXAMPLE 2-3.

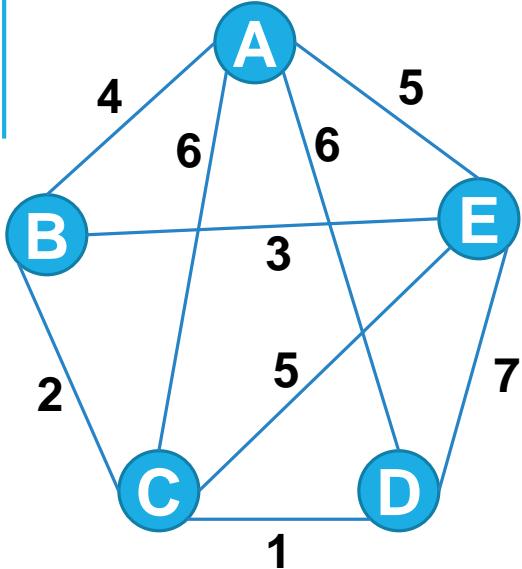


The above segments showed the execution of Prim's algorithm on the graph. The root vertex is **a**. Shaded edges are in the tree being grown, and black vertices are in the tree.

Cost of min. spanning tree

$$4+8+2+4+2+1+7+9 = 37$$

PRIMS ALGORITHM

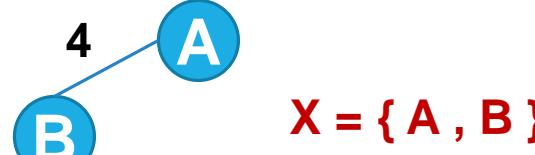


A - B 4	A - D 6	C - E 5
A - E 5	B - E 3	C - D 1
A - C 6	B - C 2	D - E 7

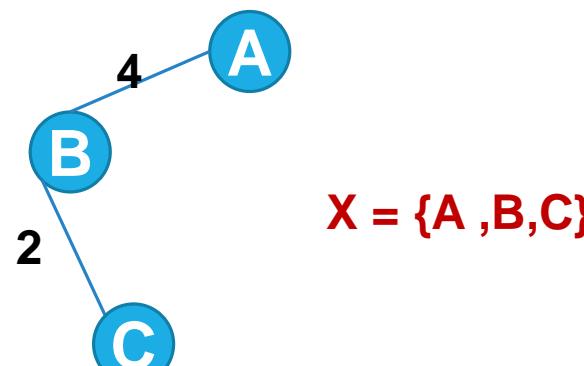
Let X be the set of nodes explored, initially $X = \{ A \}$



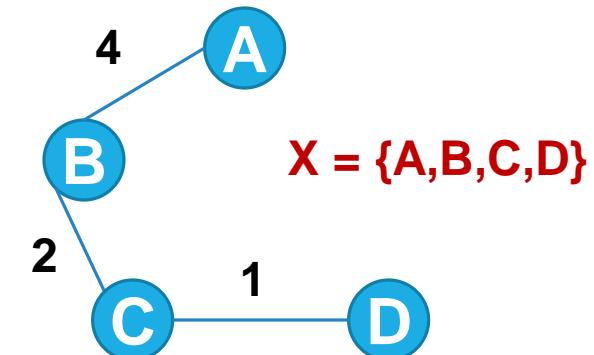
Step 1: Taking minimum Weight edge of all Adjacent edges of $X = \{ A \}$



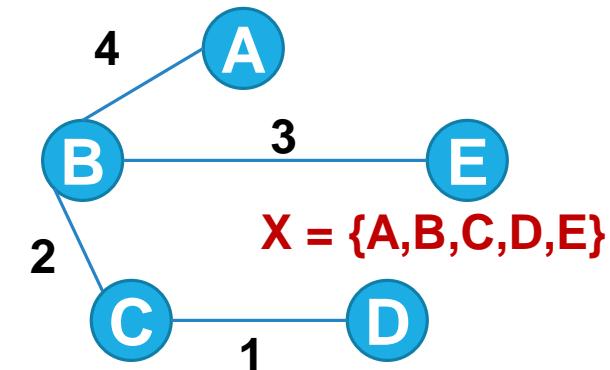
Step 2: Taking minimum weight edge of all Adjacent edges of $X = \{ A, B \}$



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 $X = \{ A, B, C, D \}$



We obtained minimum spanning tree of cost:
 $4 + 2 + 1 + 3 = 10$

THANK YOU