M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of CSE (AIML) and CSE (Cyber Security)

Course Name: Design and Analysis of Algorithms
Course Code: CY43
Credits: 3:0:0
UNIT 2

Reference:
Jon Kleinberg and Eva Tardos
Algorithm Design, Pearson (1st Edition),2013
Anany Levitin, Introduction to the Design and Analysis of Algorithms, Pearson (2017)

RAMAIAH
Institute of Technology

Unit II

Graphs – Basic Definitions, Graph Representations

Graph Connectivity and Graph Traversal

Breadth First Search

Exploring a Connected Component

Depth First Search

Implementing Graph Traversals using Queues and Stacks

Implementing BFS and DFS

An Application of BFS and DFS

Directed Acyclic Graphs (DAG)

Topological Ordering

Divide and Conquer Technique

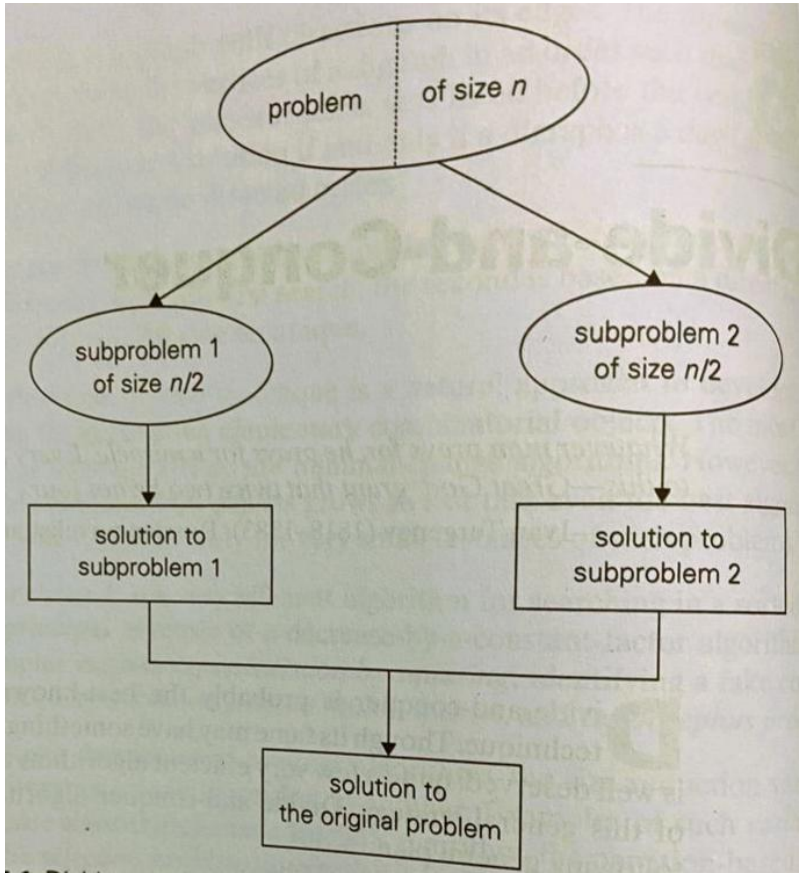
Master Theorem for recurrence relations

The Merge Sort Algorithm

Quick Sort Algorithm

Divide and Conquer Technique

- The 'Divide and Conquer' concept consists of three key principles:
  - ✓ Divide the problem into multiple subproblems or into several parts of the same type.
  - ✓ Solve the subproblems or each part independently and recursively.
  - ✓ Merge or Combine the solved subproblems to obtain the solution to the original problem.



RAMAIAH
Institute of Technology

Divide and Conquer Technique

- **Why Divide and Conquer Technique is applied?**
- Settings in which a divide and conquer approach is applied is that:
  - ✓ In a **brute force algorithm**, the ***running time of an algorithm is calculated in terms of a polynomial running time***.
  - ✓ Whereas, in a **divide and conquer strategy**, the technique mainly is focusing on ***reducing this running time to a lower polynomial***.

Divide and Conquer Technique

- **Merge Sort Algorithm:**
- **Algorithm:**

MERGE-SORT $A[1 \ldots n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \ldots \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \ldots n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**

RAMAIAH
Institute of Technology

Divide and Conquer Technique

- **Merge Sort Algorithm:**
  - **Algorithm**

Algorithm:
step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.
```
if left > right
    return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

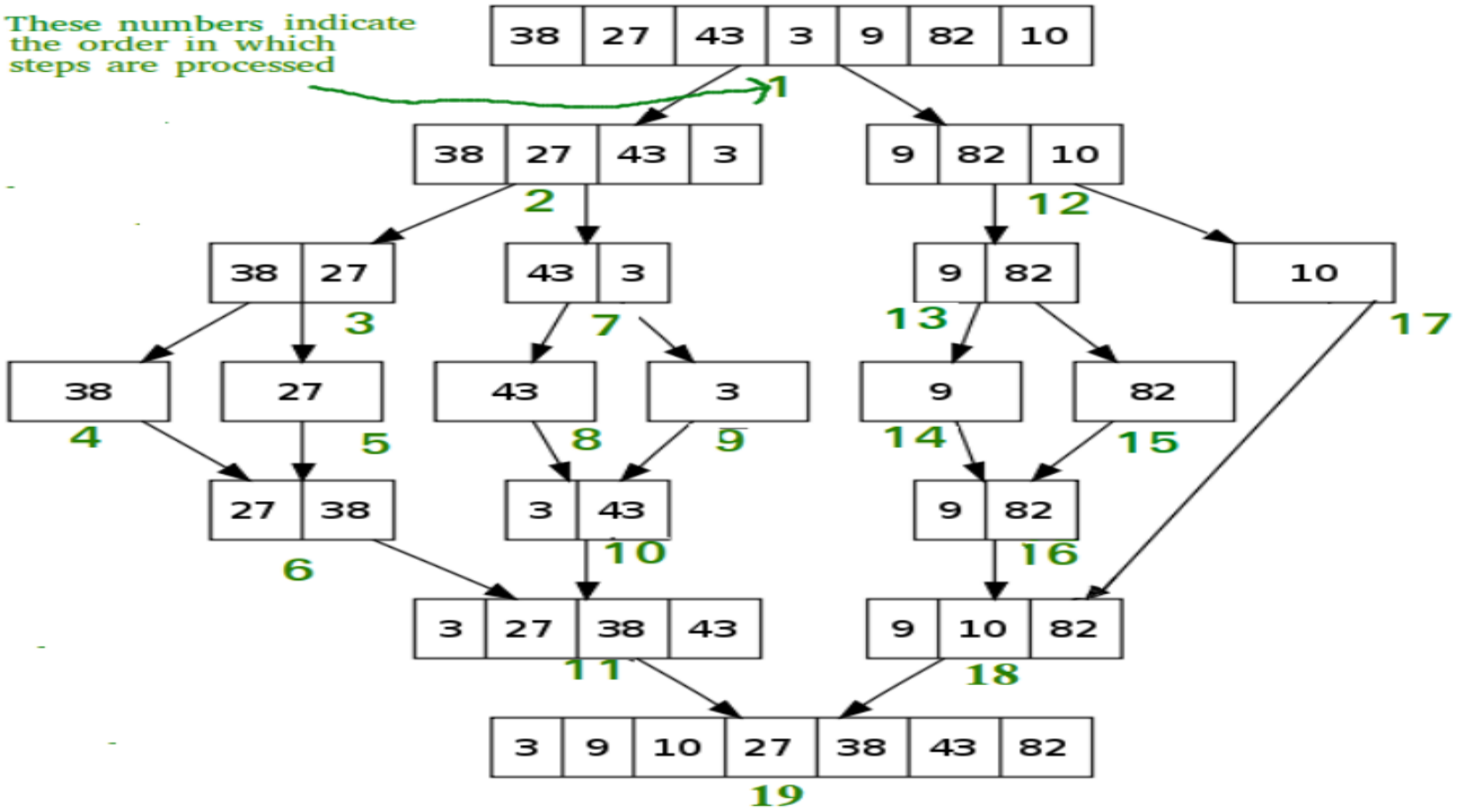
Divide and Conquer Technique

- **Merge Sort Algorithm:**
  - **Algorithm Explanation:**
    1. An unordered list is **divided in half until** lists containing **only a single element** is formed.
    2. The elements of lists containing only a **single element** are then **compared** with the **elements** with which they were **divided**.
    3. Finally, they are **merged** until a **single ordered list** is formed.

RAMAIAH
Institute of Technology

Divide and Conquer Technique – Mergesort

Example Problem



Divide and Conquer Technique

- **Recurrence relations:**
- **Analyzing** the running time of a divide and conquer algorithm involves **solving a recurrence relation**.
- Recurrence relations *bounds or limits the running time recursively in terms of the running time on smaller instances* - *a recurrence relation* ***expresses the running time of an algorithm as a function of its running time on smaller subproblems***
- There are many techniques that can be used to **bound the running time of an algorithm**. Using these methods, we can find an upper bound (worst-case time complexity) for the algorithm.
    1. Recursion Tree
    2. Substitution Method
    3. **Masters Theorem (Explain proof)**

Divide and Conquer Technique

- **Masters Theorem**
  - The Master Theorem is a powerful tool used to solve recurrence relations that arise in the analysis of divide-and-conquer algorithms. It provides a systematic way to determine the time complexity of recursive algorithms of the form:
  - We use the Master Theorem to quickly find the time complexity of divide-and-conquer problems. It applies to recurrence relations of this type:

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

n = Input size
a = Number of subproblems
b = How much each subproblem reduces (factor)
f(n) = Work done outside recursion (splitting/merging)

Divide and Conquer Technique

- **Masters Theorem**

- We compare f(n) with n^(log_b a):
  - If f(n) grows slower than n^(log_b a) → Time is O(n^(log_b a))
  - If f(n) grows at the same speed as n^(log_b a) → Time is O(n^(log_b a) log n)
  - If f(n) grows faster than n^(log_b a) → Time is O(f(n))

Divide and Conquer Technique

- **Masters Theorem**

Examples (Step-by-Step Solution)

Example 1: $T(n) = 4T(n/2) + n$

- $a = 4$, $b = 2$, $f(n) = n$
- $\log_2 4 = 2$ (because $2^2 = 4$)
- Compare $f(n) = n$ with n^2
- Since n grows slower than $n^2 \rightarrow$ Answer = $O(n^2)$

Example 2: $T(n) = 2T(n/3) + n$

- $a = 2$, $b = 3$, $f(n) = n$
- $\log_3 2 \approx 0.63$
- Compare $f(n) = n$ with $n^{0.63}$
- Since n grows faster than $n^{0.63} \rightarrow$ Answer = $O(n)$

Divide and Conquer Technique

- **<u>Analyzing the Mergesort Algorithm using Recurrence relation:</u>**

  Let T(n) = denotes worst-case running time for a merge sort algorithm

  n = input size

  Say, for example, n is even, the algorithm spends O(n) time to divide the input into 2 pieces of size (n/2) each.

  It then spends T(n/2) to solve each piece – considered the worst-case running time for an input size n/2

  Now, finally combining the solutions of 2 pieces together, it spends O(n) time.

- Hence T(n) satisfies the following **recurrence relation**:

  For some constant c,

$T(n) <= 2T(n/2) + cn$	where $n > 2$ and $T(2) <= c$.

Divide and Conquer Technique

- **Analyzing the Mergesort Algorithm:**
- To analyze the Mergesort algorithm or any similar divide and conquer technique algorithms, a base case for recursion should be considered.
- The recursive function must stop calling itself when it reaches a certain smallest possible input size (constant size)

- Example in Mergesort algorithm,
  - when the input is reduced to size 2, recursion stops. [O(logn)]
  - Sort the elements by comparing them to each other [O(n)]
  - Hence worst-case analysis can be told as [O(n*logn)]

- **Best Case**: O(nlogn): **Array is already sorted**. Merge Sort still divides the array recursively into halves and merges them back. Even if merging is trivial (elements already sorted), it still requires O(n) work per level.
- **Worst Case**: O(nlogn): **Array is in completely reverse order**.
- **Average Case**: O(nlogn): Since this case considers **all possible ways the input elements can be arranged.**

Divide and Conquer Technique

- **Analyzing the Mergesort Algorithm using Masters Theorem:**

  The standard recurrence formula is:

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

  For Merge Sort:

  - a = 2 (two subproblems)
  - b = 2 (each subproblem is half of n)
  - f(n) = O(n) (merging step)

  Now, check log_b a: log2 2=1

  Since f(n) = O(n) = O(n^{log_2 2}), we apply Case 2 of the Master Theorem.

  Final Answer: Merge Sort takes O(n log n) time

Divide and Conquer Technique

- **Quick Sort Algorithm:**
- **Concept:**

To efficiently sort an array, quicksort uses three key pointers:

1. **Pivot**: The pivot is the element around which the array is divided.
2. **Index Pointer**: This pointer keeps track of the *boundary where elements smaller than the pivot* should be placed.
3. **Walker Pointer**: This pointer scans the array to *identify elements smaller than or equal to the pivot*, which are then *swapped* with *elements at the index pointer*.

When the walker pointer reaches the pivot, the array is divided at that point, and the process repeats on the left and right subarrays. This continues recursively until the entire array is sorted.

Divide and Conquer Technique

- **Quick Sort Algorithm:**
- **Algorithm:**
  1. **Pivot Selection:** Choose a pivot element from the array. This pivot is used to divide the array into two parts.
  2. **Partitioning**: Rearrange the array so that elements smaller than the pivot are on the left side and elements greater than the pivot are on the right side.
  3. **Recursion:** Repeat the process on the left and right subarrays until they contain only one element each (which are inherently sorted).

What is the advantage of the Quick Sort Algorithm over the other divide and conquer algorithms (Merge Sort Algorithm)?

Uses the concept of **In-Place Sorting:** Quicksort sorts the array without creating any additional data structures. Meaning Unlike Merge Sort, which creates temporary subarrays for merging, Quicksort rearranges elements within the same array using swaps.

RAMAIAH Institute of Technology

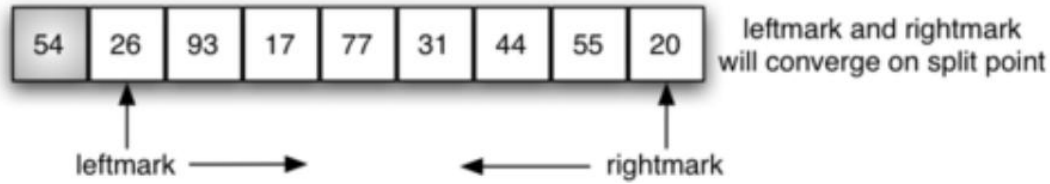
Divide and Conquer Technique – Quick Sort Example Problem

- Consider an array of elements given below:

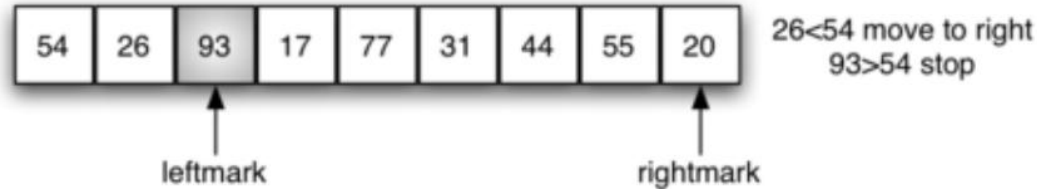
- Here we simply choose the first element as the **first pivot value**.

- The next step is to **partition the array elements**. To do this we need to find the **split point -** used to divide the list.
- Partitioning begins by **locating two position markers**— a *leftmark* and a *rightmark*—at the beginning and end of the remaining items in the list (in this case, position 1 and position 8)

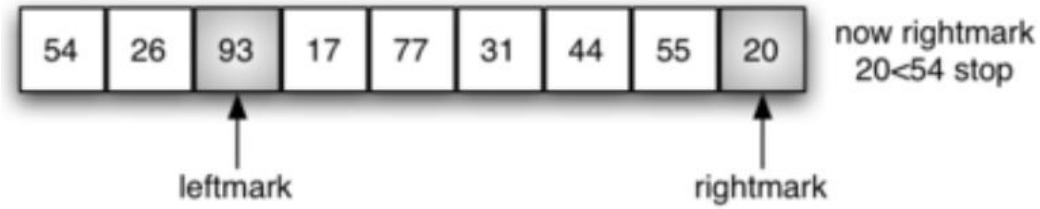
Divide and Conquer Technique – Quick Sort Example Problem



- Now, begin by **incrementing leftmark** until we locate a value that is greater than the pivot value

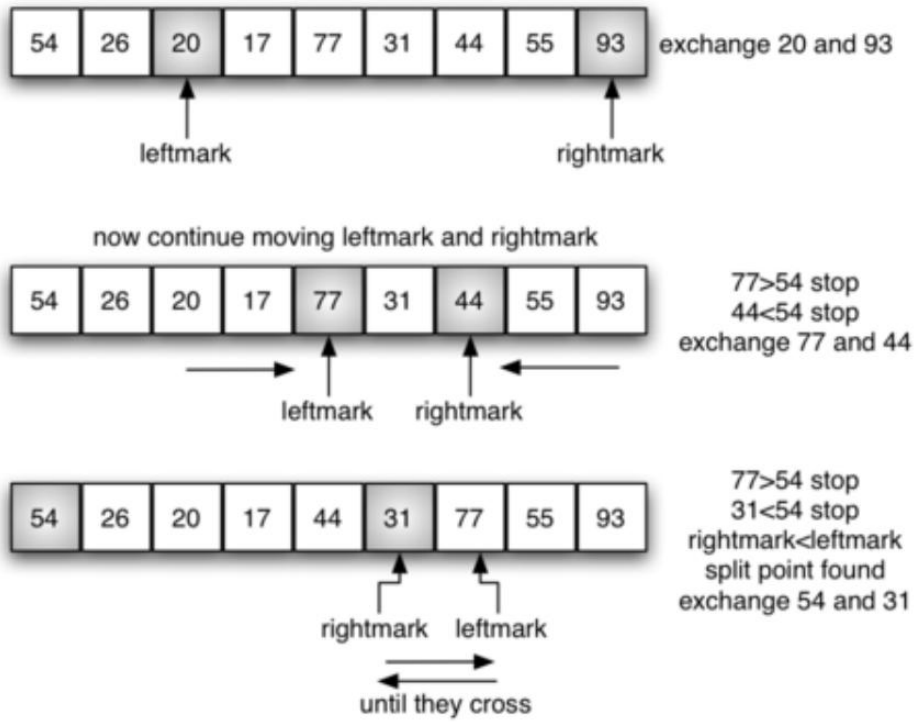


- Now, go to the rightmark. **Decrement rightmark** until we find a value that is less than the pivot value



Divide and Conquer Technique – Quick Sort Example Problem

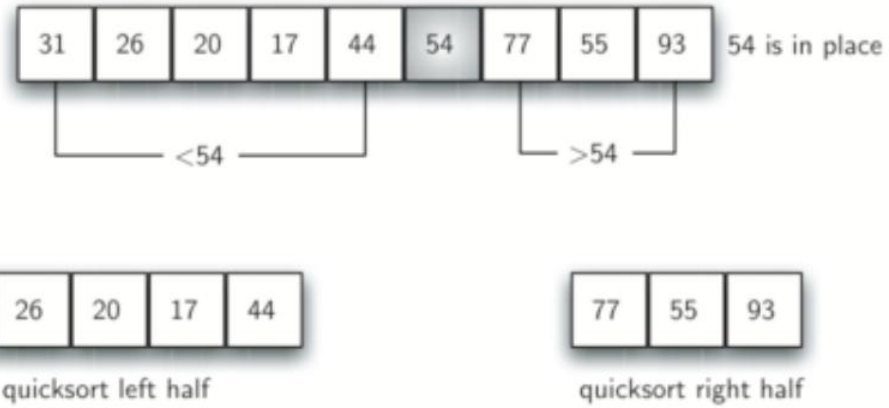
- So as of now here, two values are out of place (93 and 20). Exchange these two items and then repeat the process again.



RAMAIAH Institute of Technology

Divide and Conquer Technique – Quick Sort Example Problem

- At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point (i.e., at 31).
- Next step, the pivot value (54 in this case) can be exchanged with the contents of the split point (31 in this case). Thus, the pivot value is now in place.



- Hence items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value.
- The quick sort is invoked recursively on the two halves.

RAMAIAH
Institute of Technology

Divide and Conquer Technique – Quick Sort Example Problem

- **Time Complexity Analysis for Quick Sort Algorithm**

✓ Best Case: O(nlogn): The pivot is **always** the **median** (or close to it).

✓ Worst Case: $O(n^2)$: This is when the **split points may not be in the middle and can be very skewed to the left or the right**, leaving a very uneven division.

✓ Average Case: O(nlogn): If partition occurs in the middle of the list, there will again be "log n" divisions. In order to find the split point, each of the *n* items needs to be checked against the pivot value, hence n log n.

RAMAIAH
Institute of Technology

How to solve recurrence relations using substitution method

- The substitution method is a technique used to solve recurrence relations by
    i. guessing a solution and then
    ii. using mathematical induction to prove it correct.

Graphs – Basic Definitions

- A graph is a way of encoding **pairwise relationships among a set of objects**.
- It consists of a **collection V of nodes or vertices (vertex)** and a **collection E of edges**.
- The collection E of edges "joins" two nodes of the graph.
- Thus, an edge of a graph ($e \in E$) is represented as a two-element subset of V: $e = \{u,v\}$ for some $u, v \in V$, where u and v are the ends of e.
- **Symmetric relationships in graphs**: Indicated by an **UNDIRECTED GRAPH** where the edges in the graph indicates a symmetric relationship between their ends.

Graphs – Basic Definitions

- **Asymmetric relationships in graphs**: Edges indicating a **one-way relationship** and is represented by a **DIRECTED GRAPH G'** that consists of **a set of nodes V** and **a set of directed edges E'**.
- Each **e' ∈ E' is an ordered pair (u,v)** where
  1. the positions of u and v are not interchangeable.
  2. u is called the tail of the edge and v is called the head of the edge.
  3. Also it means, **edge e'** leaves **node u** and enters **node v**.

Graphs – Representations (Adjacency Matrix and Adjacency List

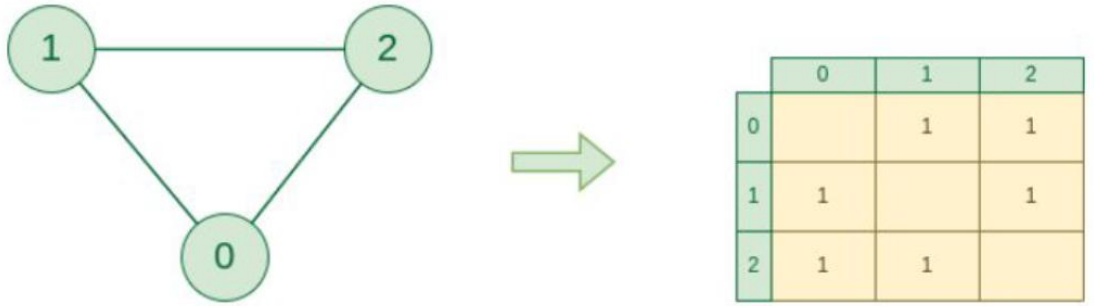
- Graphs for computer algorithms are usually represented in one of two ways: the adjacency matrix and adjacency lists.
- The adjacency matrix of a graph with n vertices is an n × n boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge.

Graphs – Representations (Adjacency Matrix and Adjacency List

- **Adjacency Matrix Representation**
  - An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)
  - Let's assume there are n vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.
    - If there is an edge from vertex i to j, mark **adjMat[i][j]** as 1.
    - If there is no edge from vertex i to j, mark **adjMat[i][j]** as 0.

Graphs – Representations (Adjacency Matrix and Adjacency List

- **Representation of Undirected Graph as Adjacency Matrix**
- The below figure shows an undirected graph. Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 to both cases (**adjMat[destination] and adjMat[destination]**) because we can go either way.



Undirected Graph

Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

Graphs – Representations (Adjacency List)

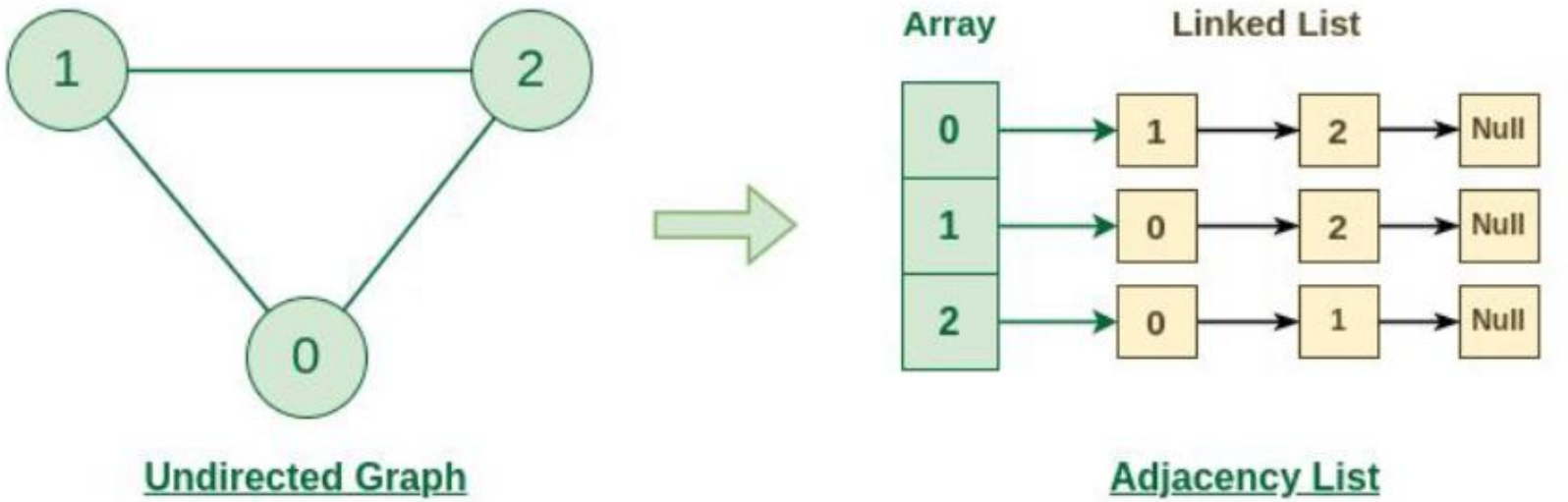
- **Adjacency List Representation**
  - An array of Lists is used to store edges between two vertices.
  - The size of array is equal to the number of vertices (i.e, n).
  - Each index in this array represents a specific vertex in the graph.
  - The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i.
  - Assume there are n vertices in graph, create an array of list of size n as **adjList[n]**.
    - adjList[0] will have all the nodes which are connected (neighbour) to vertex 0.
    - adjList[1] will have all the nodes which are connected (neighbour) to vertex 1 and so on.

Graphs – Representations (Adjacency List)

- **Representation of Undirected Graph as Adjacency list:**
  - The below undirected graph has 3 vertices.
  - So, an array of list will be created of size 3, where each indices represent the vertices.
  - Now, vertex 0 has two neighbours (i.e, 1 and 2).
  - So, insert vertex 1 and 2 at indices 0 of array.
  - Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array.
  - Similarly, for vertex 2, insert its neighbours in array of list

Graphs – Representations (Adjacency List)

- **Representation of Undirected Graph as Adjacency list:**



Graph Representation of Undirected graph to Adjacency List

Graphs – Representations (Adjacency List)

- The adjacency lists of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex
- To put it another way, adjacency lists indicate columns of the adjacency matrix that, for a given vertex, contain 1's.

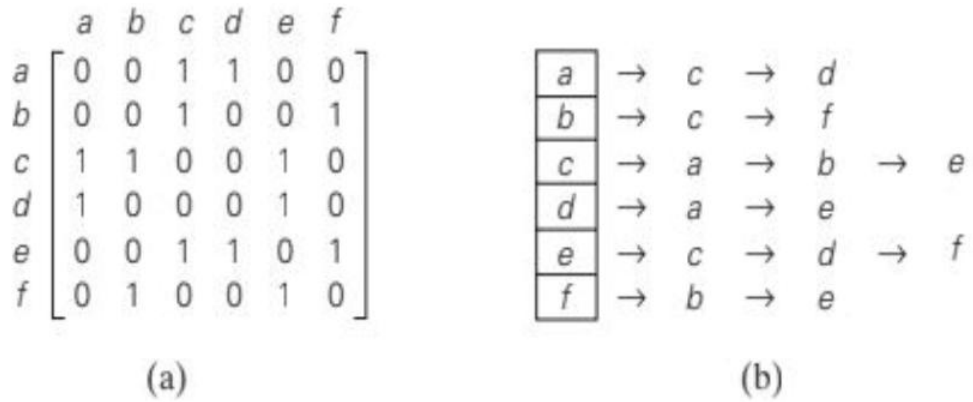


FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a.

Graphs – Examples of Graphs

- Transportation Networks
- Communication Networks
- Information Networks
- Social Networks
- Dependency Networks

Graphs – Paths and Connectivity

- **Fundamental operation in a graph** – Traversing a sequence of nodes connected by edges.
- **PATH (P)** in a graph – A path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, v_3,\ldots,v_{k-1},v_k$
- **Properties of PATH**
  - Each consecutive pair v_i v_{i+1} is joined by an edge in G. So, P is called as a path from v_1 to v_k
  - A path is called simple if all its vertices are distinct from one another.
- **CYCLE** in a graph – A cycle is a path $v_1, v_2, v_3,\ldots,v_{k-1},v_k$ in which the **sequence of nodes "cycles back" to where it began**.

Graphs – Paths and Connectivity

- **DIRECTED PATH OR CYCLE** in a graph –

  1. Each pair of consecutive nodes has the property that (v_i v_{i+1}) is an edge.
  2. The sequence of nodes in the path or cycle must have a directionality of edges.

- **CONNECTIVITY OF GRAPHS -**
- **CONNECTED UNDIRECTED GRAPH** – An undirected graph is connected, if, for every pair of nodes u and v there is a path from u to v.
- **CONNECTED DIRECTED GRAPH** – A directed graph is a strongly connected, if, for every two nodes u and v, there is a path from u to v and a path from v to u.

Graphs – Paths and Connectivity

- **SHORTEST PATH** in a graph – The distance between the two nodes u and v is said to be a short path when there is a **minimum number of edges in the u-v path**.
- The distance between the nodes or vertices that are not connected by a path in a graph can be denoted with a symbol ∞
- **TREES -** An undirected graph is said to be a tree, if,
  i. **It is connected**
  ii. **Does not contain a CYCLE**
- **TREES** are the simplest kind of connected graphs. Deleting any edge from a tree will disconnect the tree.

Graphs – Paths and Connectivity

- **PROPERTIES OF TREES**
  - Every n-node tree has exactly n-1 edges
  - If G is an undirected graph on n nodes, then any of the following two statements implies the third statement
    i. G is connected
    ii. G does not contain a CYCLE
    iii. G has n-1 edges

Graph Connectivity and Graph Traversal

- **Node-to-node Connectivity:**

  - **Assumption** - Suppose there is a graph G = (V, E) with 2 specific nodes or vertices s and t,
  - **Goal** - To find an efficient algorithm that answers the below question:

    "Is there a path from s to t in G?"

  This is called THE **PROBLEM OF DETERMINING THE s-t CONNECTIVITY or MAZE SOLVING PROBLEM**

  - **Problem Statement –** To start at the node s and find the way to another designated node t. To design efficient algorithms for this kind of problem statements we have 2 natural algorithms
  - ✓ **Breadth First Search (BFS) and Depth First Search (DFS) –** Here the input to the algorithm is a graph

Breadth First Search (BFS)

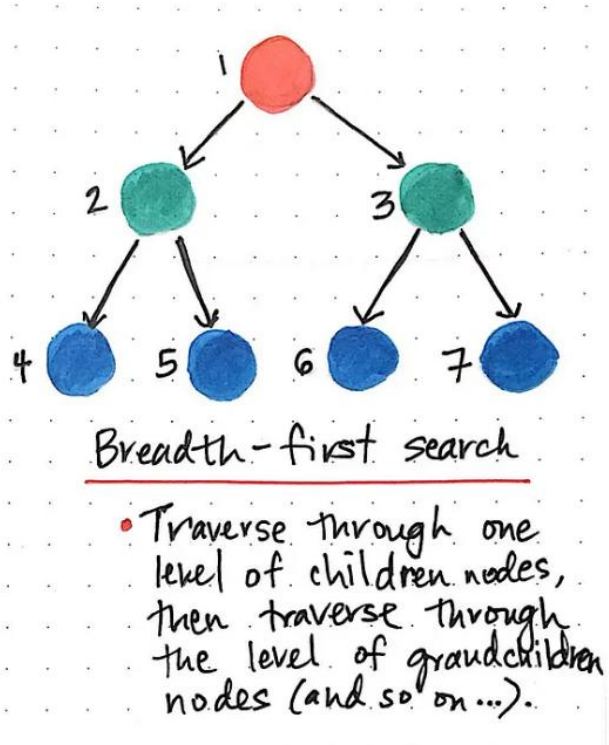
- **Breadth-First Search (BFS): Exploring a Connected Component**
  - Breadth First Search (BFS) is a graph traversal algorithm used to explore nodes and edges of a graph or tree level by level.
  - Use case: To find a shortest path
  - BFS explores all nodes at the present depth before moving deeper. It uses a queue (FIFO) structure.
  - **BFS Concept:**
    - The ‘Breadth First Search’ concept consists of the following key principles:

Breadth First Search (BFS)

- **Breadth-First Search (BFS): Exploring a Connected Component**
  - **BFS Concept:**
  - **Explore all the neighbours of a node before moving to the next level of neighbours** - It starts from a **source node** and explores all its neighbouring nodes first, before moving on to the next level of neighbours.
  - Visit nodes in **increasing order of distance** from the start node.
  - Use a **queue** to manage the order of exploration.
  - Track **visited nodes** to avoid cycles and redundant work.

Breadth First Search (BFS)

- **Breadth-First Search (BFS): Exploring a Connected Component**
  - **Example of BFS**



Breadth First Search (BFS) - Algorithm

1. **Initialize** an empty set or list **visited** to keep track of visited nodes.
2. **Initialize** an empty queue Q.
3. **Enqueue** the start node into the queue.
4. **Mark** the start node as visited.
5. **While** the queue is not empty:
    a. Dequeue a node *current* from the front of the queue.
    b. Print the current node.
    c. For each **neighbour node** of the current node:
        i. If the neighbour is **not visited**:
            1. Mark it as visited.
            2. Enqueue the neighbour into the queue.
6. Repeat until all reachable nodes are visited.

Displaying the BFS traversal order: To **display the BFS traversal order**, you just need to **print each node** as you visit it during the BFS traversal.

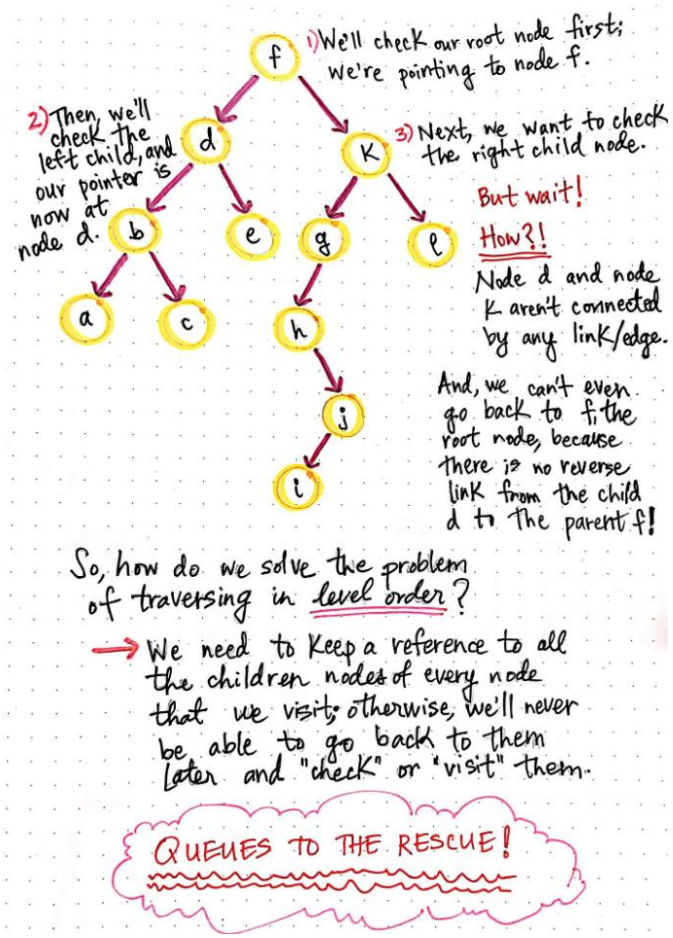
Breadth First Search (BFS) – Implementation using Queue

- BFS is implemented using a Queue.
- Because the Queue follows a FIFO approach i.e., a queue processes the items in the order they were added. The first element added is the first to be removed.
- Similarly, in BFS, we start with node 1, we enqueue node 1, then we dequeue 1, mark it as visited, and enqueue its neighbours. If there are no neighbours, we move to the next level, enqueue 2, mark it as visited and enqueue its neighbour 3.
- Now since the queue is FIFO, 2 is visited before 3, and 3 is visited before 4, 4 is visited before 5, 6, and 7.

Thus, ensuring that we explore all nodes at the current level before going deeper – LEVEL-BY-LEVEL TRAVERSAL.

RAMAIAH Institute of Technology

Breadth First Search (BFS) – Implementation using Queue



Breadth First Search (BFS) – Applications of BFS

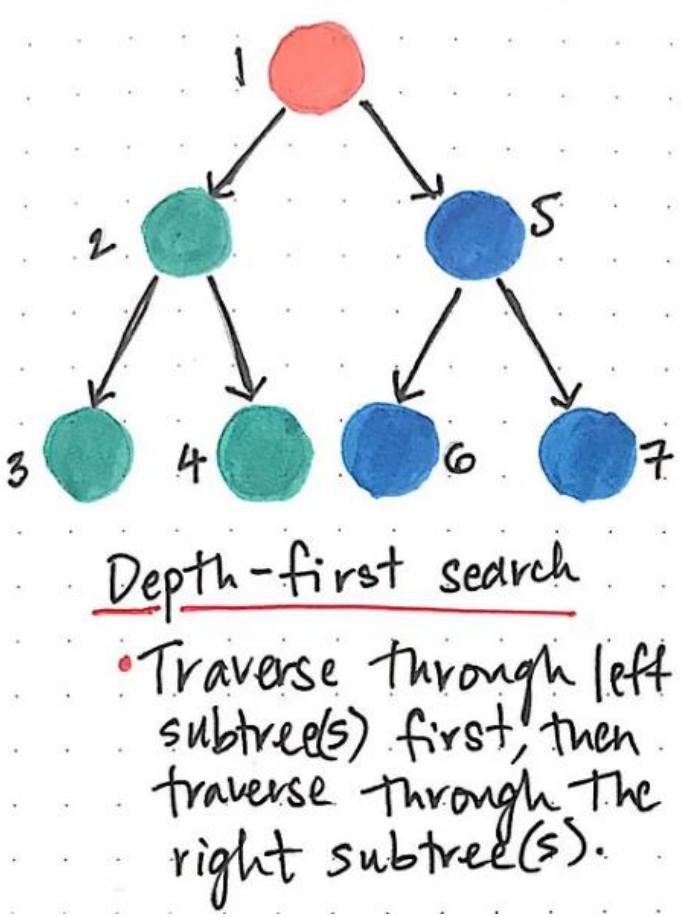
- Finding the shortest path in an unweighted graph.
- Checking graph connectivity.
- Solving mazes and puzzles

Depth First Search (DFS)

- **DFS** is a graph traversal algorithm that explores **as deep as possible along each branch** before backtracking.
- It uses a **stack (LIFO)** or recursion.
- **Usecase –** To check if there is a path or to check if the graph is connected or not
- **DFS Concept:**
- The 'Depth First Search' concept consists of the following key principles:
  - **Go deep into one path until you can't go further, then backtrack and explore the next path**
  - In DFS, we can determine whether two nodes x and y have a path between them by looking at the children of the starting node and recursively determining if a path exists.

Depth First Search (DFS)

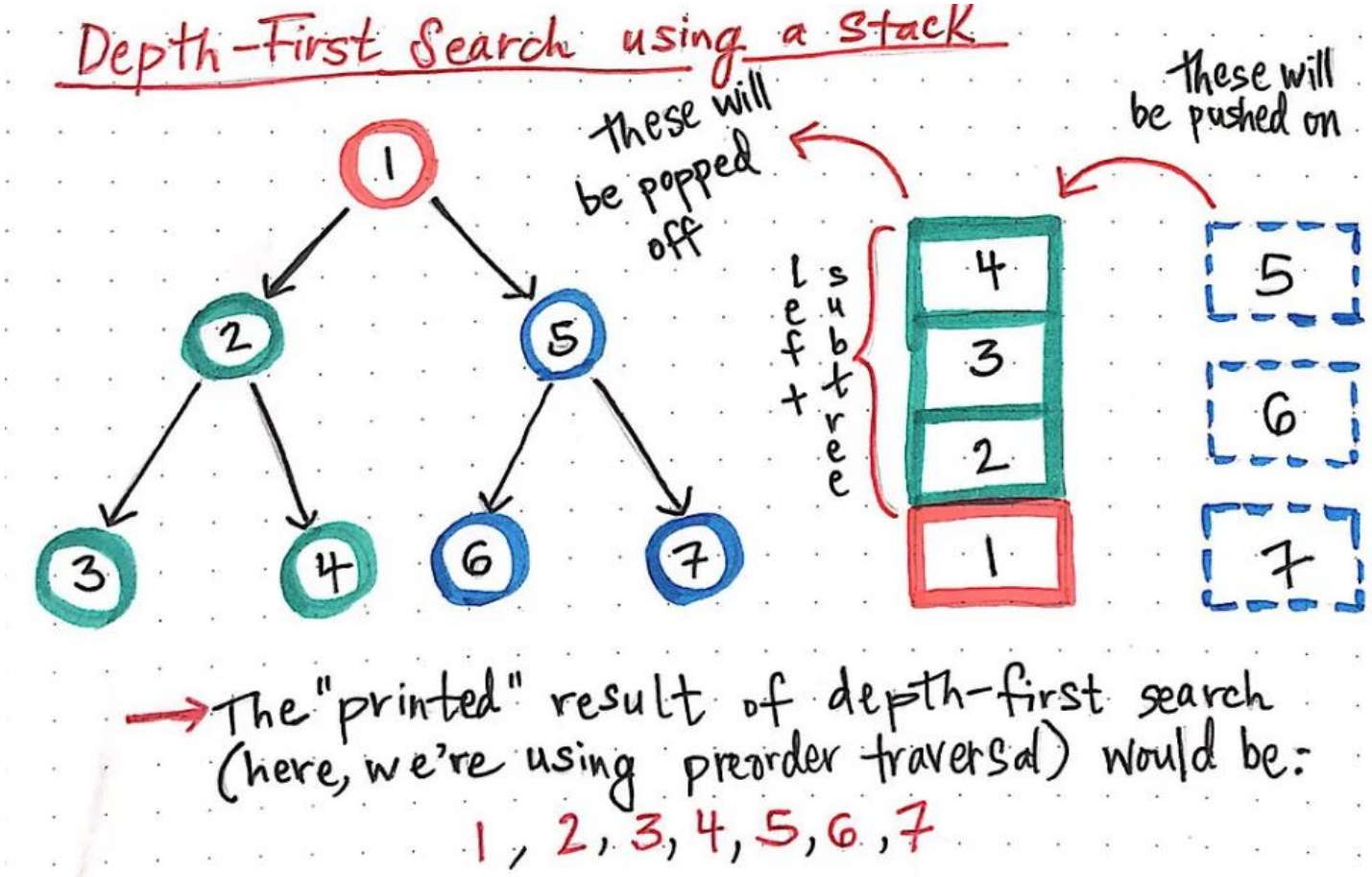
- **DFS Example:**



Depth First Search (DFS) - Algorithm

1. Start from a source node (or any unvisited node in a disconnected graph).
2. Mark the current node as visited.
3. Recursively visit all unvisited adjacent nodes.
4. Backtrack when no unvisited adjacent nodes remain.

Depth First Search (DFS) – Implementation using Stack



Depth First Search (DFS) – Application of DFS

1. Topological sorting.
2. Detecting cycles in graphs.
3. Finding strongly connected components

- To summarize, graph connectivity refers to whether all vertices (nodes) in a graph are connected in some way. A graph can be categorized into:
  1. **Connected Graph:** A graph is connected if there is a path between any two vertices.
  2. **Disconnected Graph:** A graph is disconnected if at least one pair of vertices has no path between them.
  3. **Strongly Connected (for Directed Graphs):** A directed graph is strongly connected if there is a directed path from any vertex to every other vertex.
  4. **Weakly Connected (for Directed Graphs):** A directed graph is weakly connected if replacing all directed edges with undirected edges results in a connected graph