
Introduction to
Data Structures

Data Structures Definition

- **Definition:** Data Structure is a way to store or organize data in such way that it can be used / accessed efficiently.

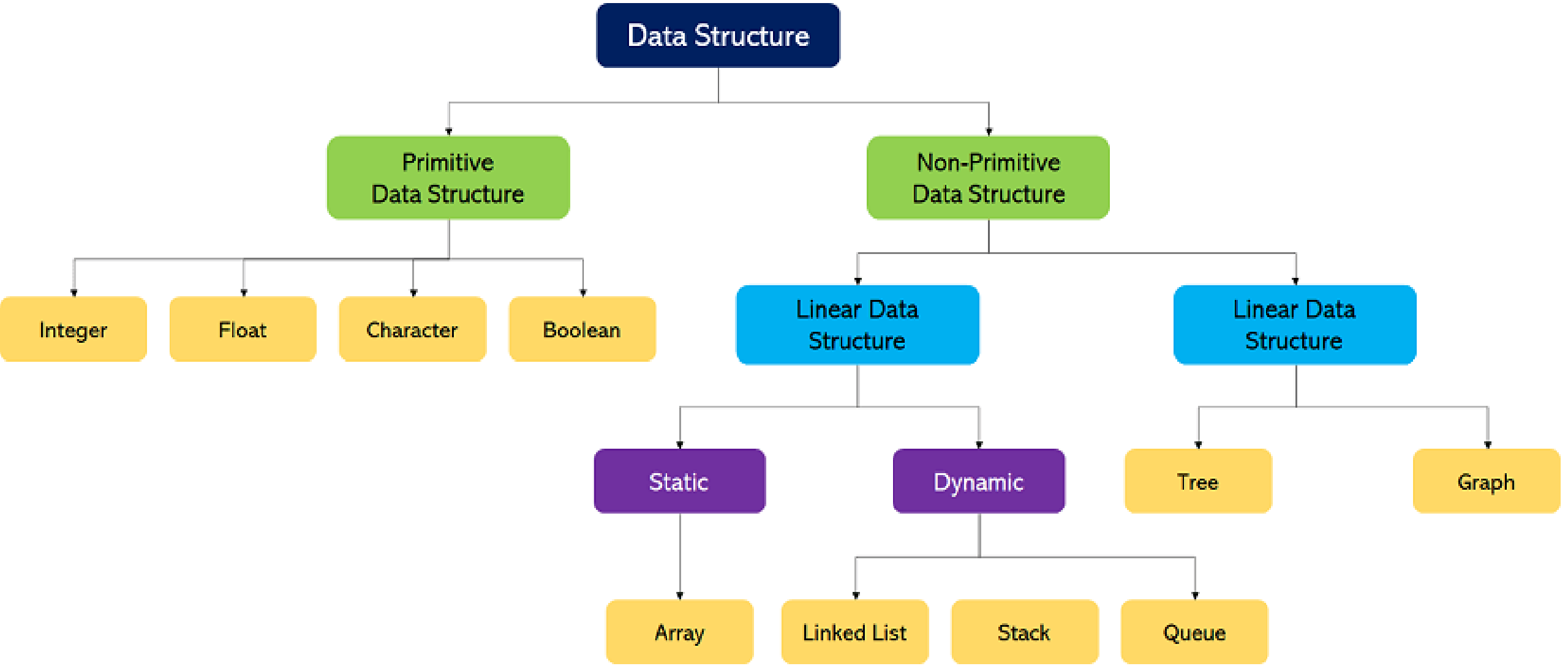
Classification of Data Structures

Data Structures are majorly classified into 2 types

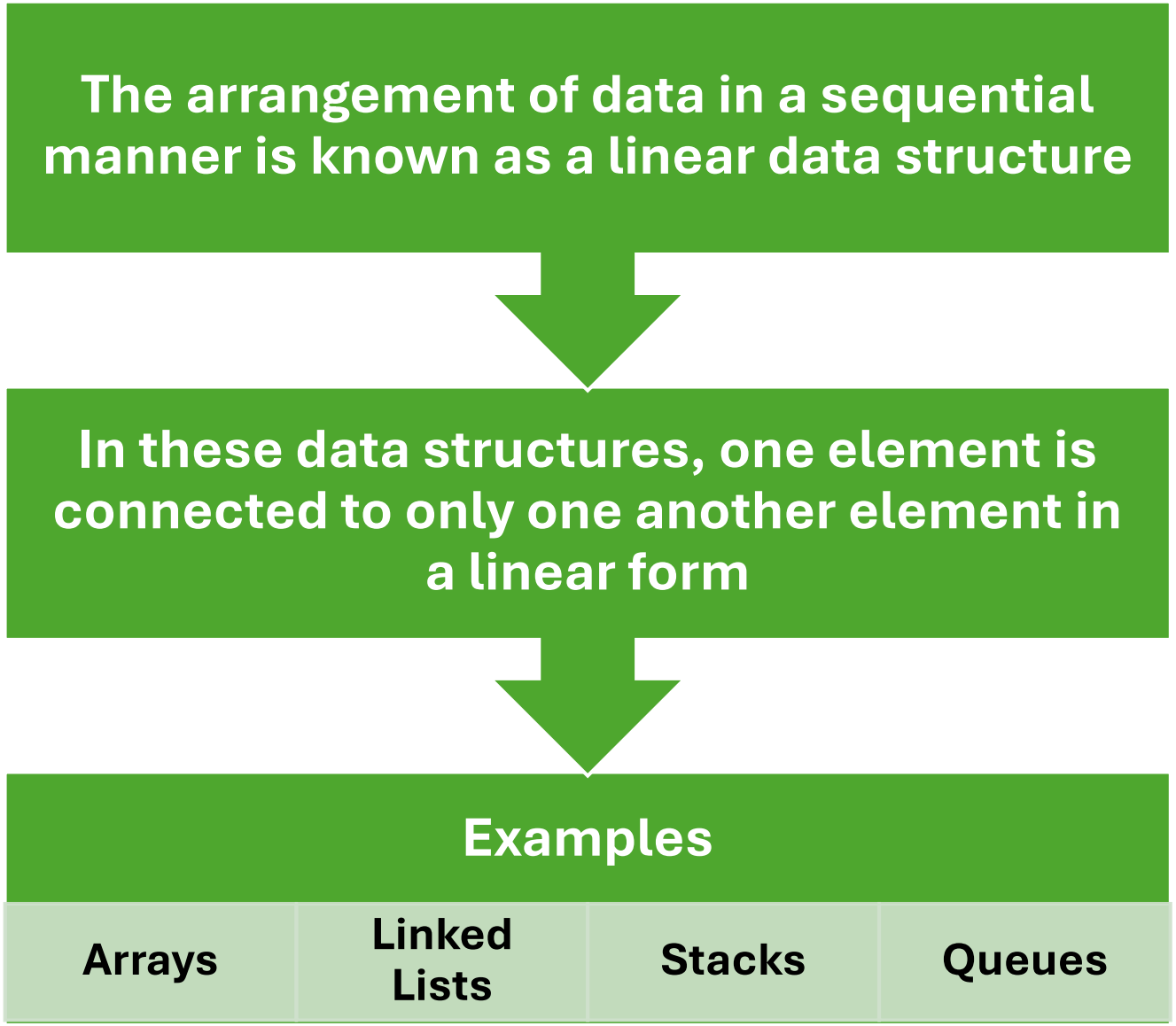
Linear Data Structures

Non-Linear Data Strctures

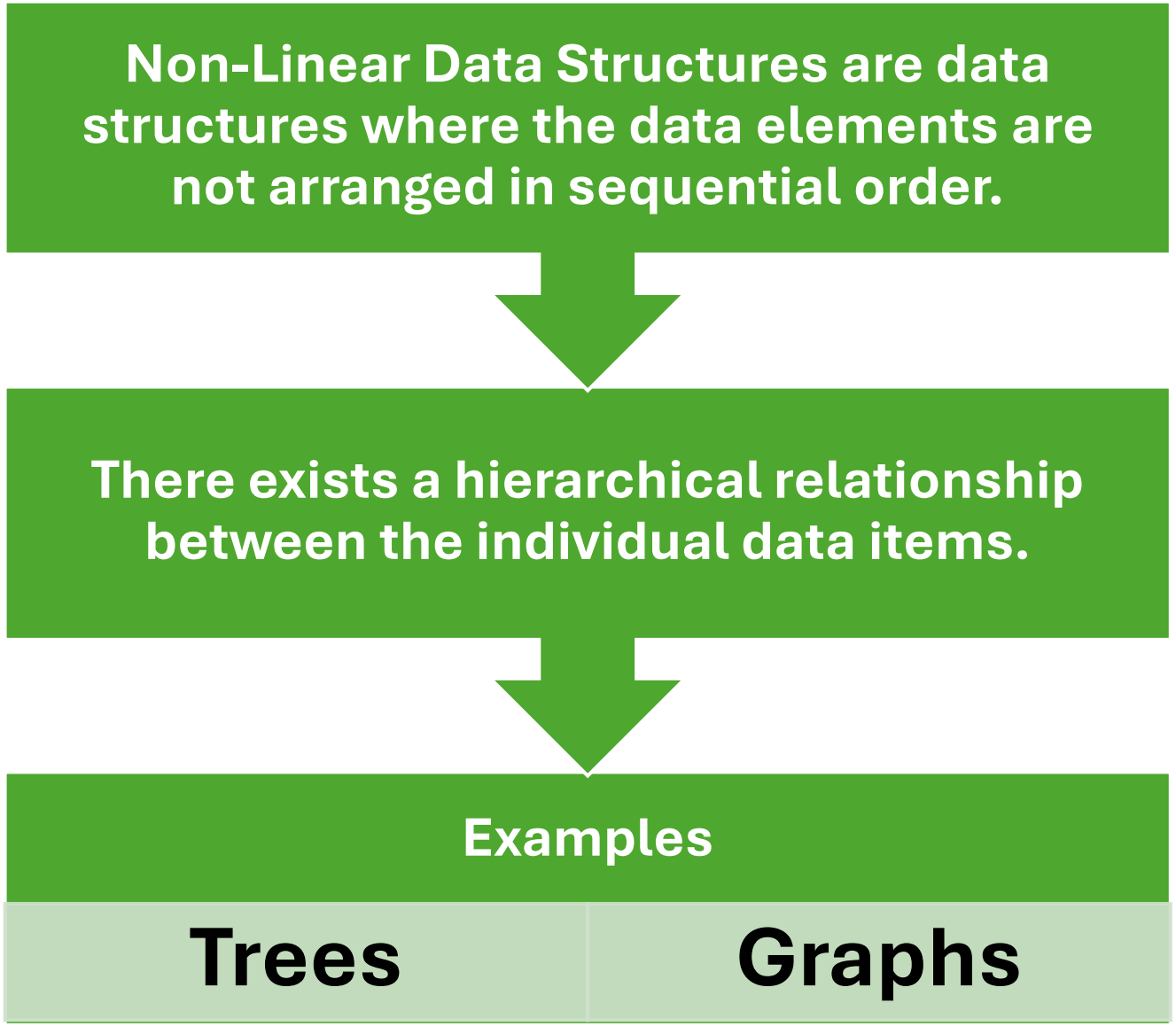
Detailed Classification



Linear Data Structures



Non Linear Data Structures



Let’s understand the importance of each **linear data structure** one by one.

Arrays

Arrays

Arrays are defined as the collection of similar types of data items stored at contiguous memory locations

Array elements are indexed (Starting from 0) and we can access elements using indices

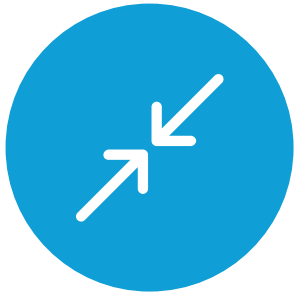
Advantages of Arrays

- Multiple values under a single name
- Traversing is easy (Contiguous memory locations)
- Accessing an element is very easy (Just use the index)

Disadvantages of Arrays

Homogeneous

Static memory allocation (If it’s a static array)

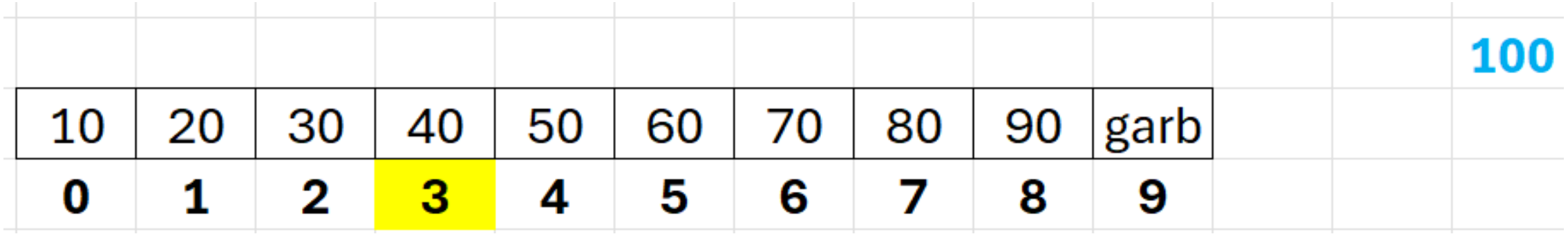


Wastage of Memory

Difficulty of insertion and deletion

Why deleting or inserting an element in an array is tedious?

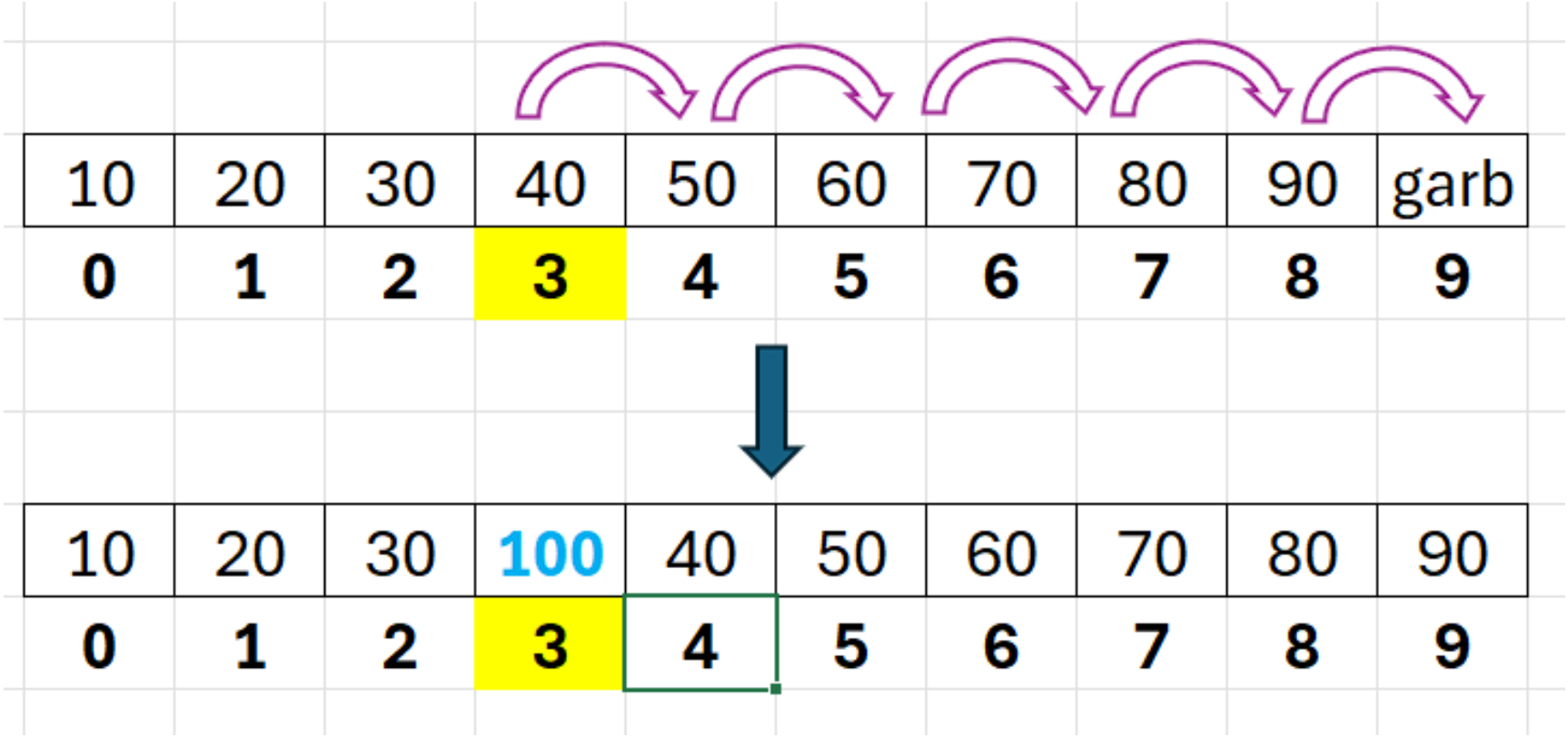
- Consider the following array of 10 elements where index 9 is empty (Currently a garbage value is sitting there.



- Technically the array only contains 9 elements and now you want to **insert a new element 100 at index 3**
- Remember **insertion doesn't mean replacement.**

Why deleting or inserting an element in an array is tedious?

- We should do the following to in-order to insert 100 at 3rd index



Why deleting or inserting an element in an array is tedious?

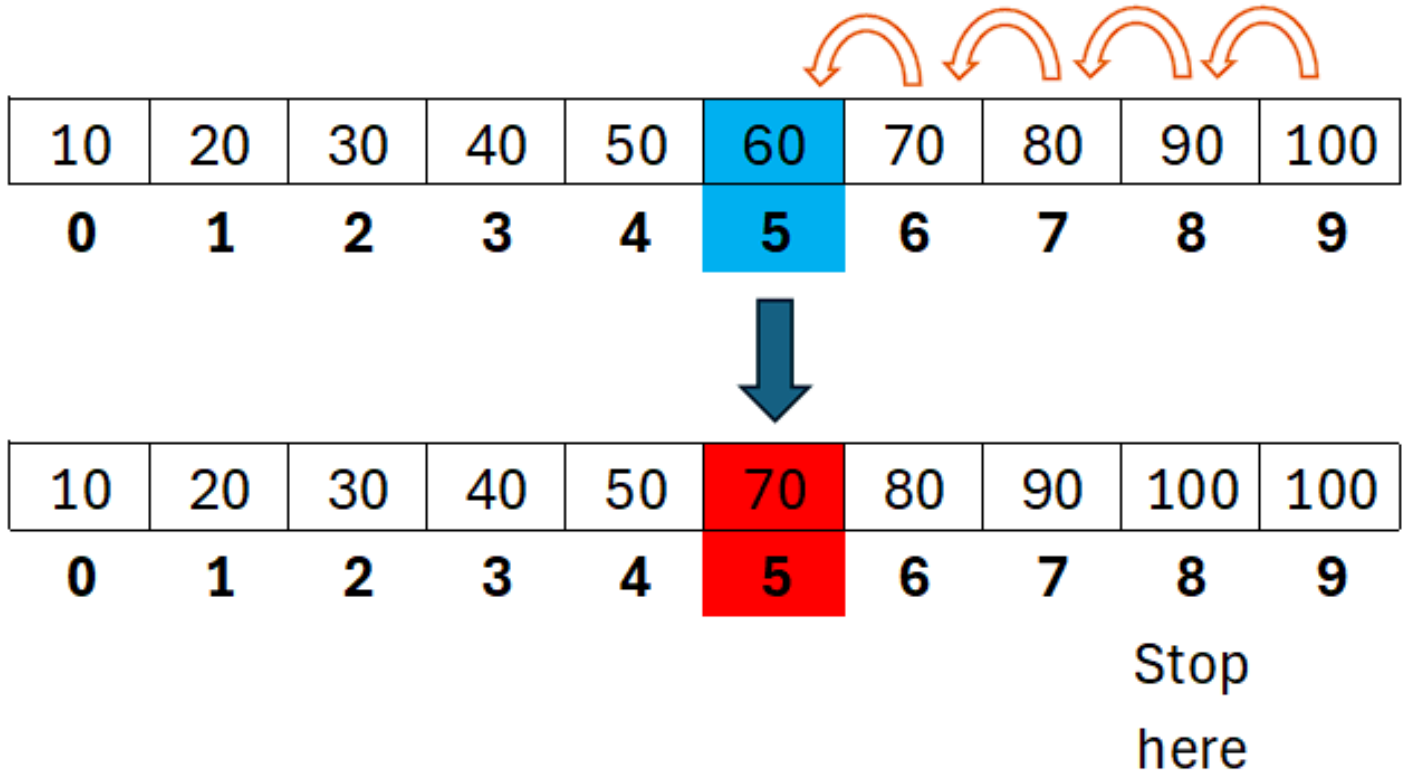
- Let’s see deleting an element from the below array now. Say we want to delete **60**

10	20	30	40	50	60	70	80	90	100
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

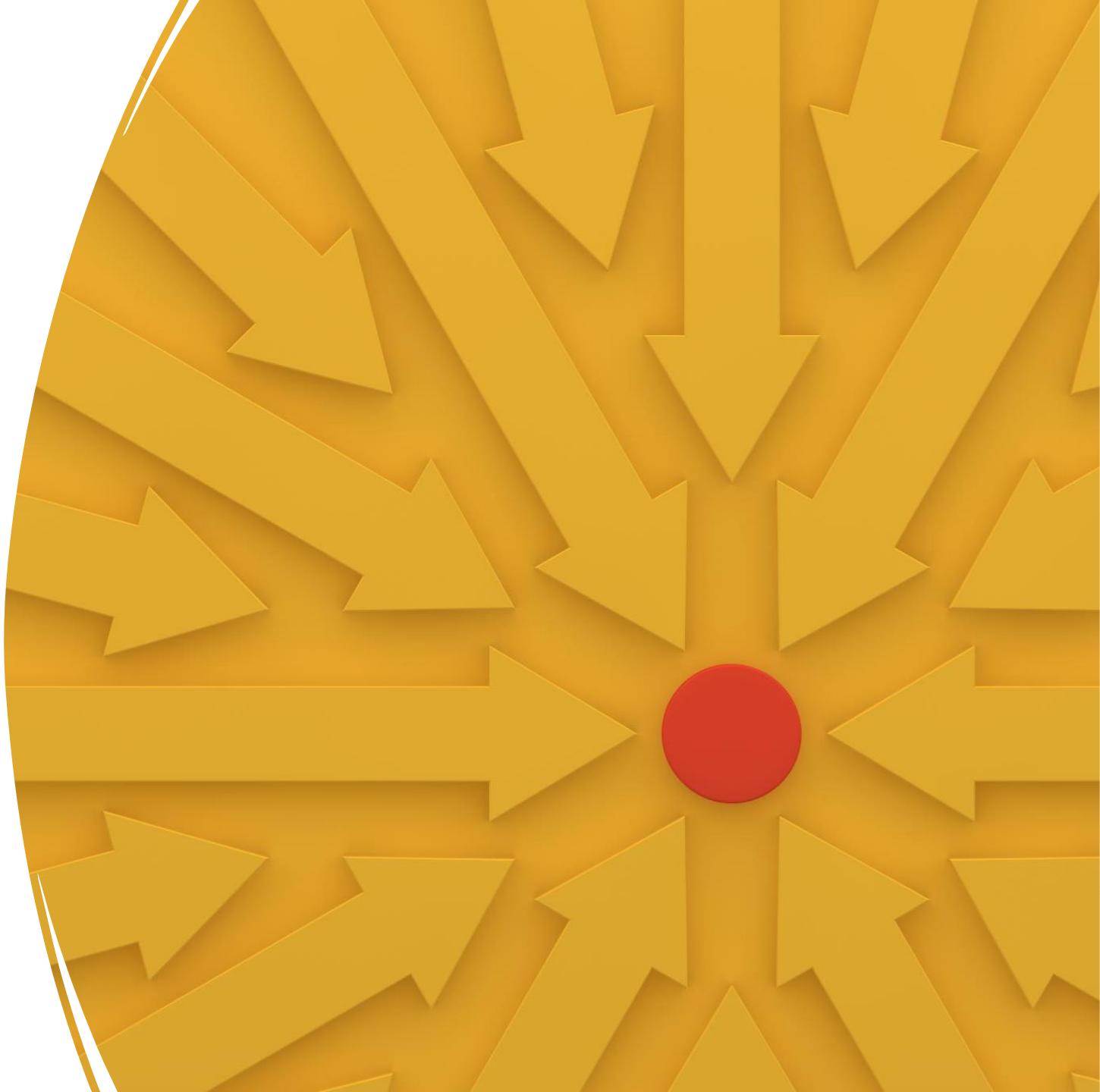
- Remember **after deleting we should be able to access 70 after 50**

Why deleting or inserting an element in an array is tedious?

- We should do the following to delete **60**



To deal with those disadvantages we have a concept called



Linked Lists

Linked Lists

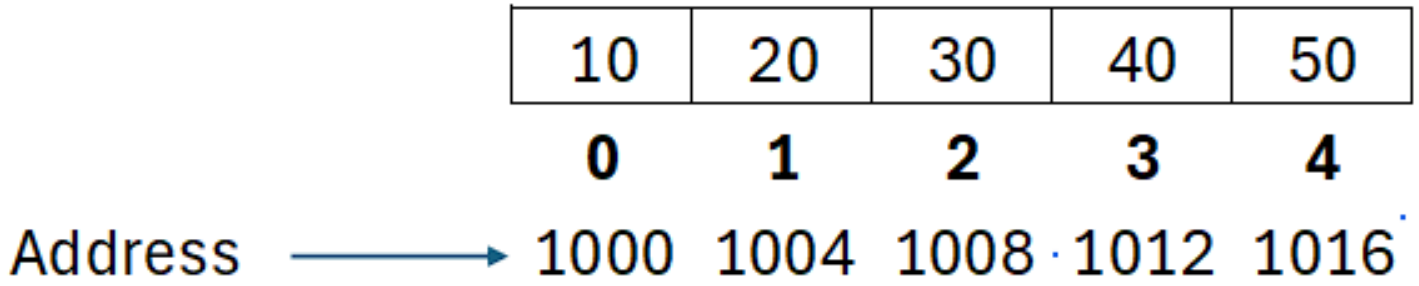
01

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory.

02

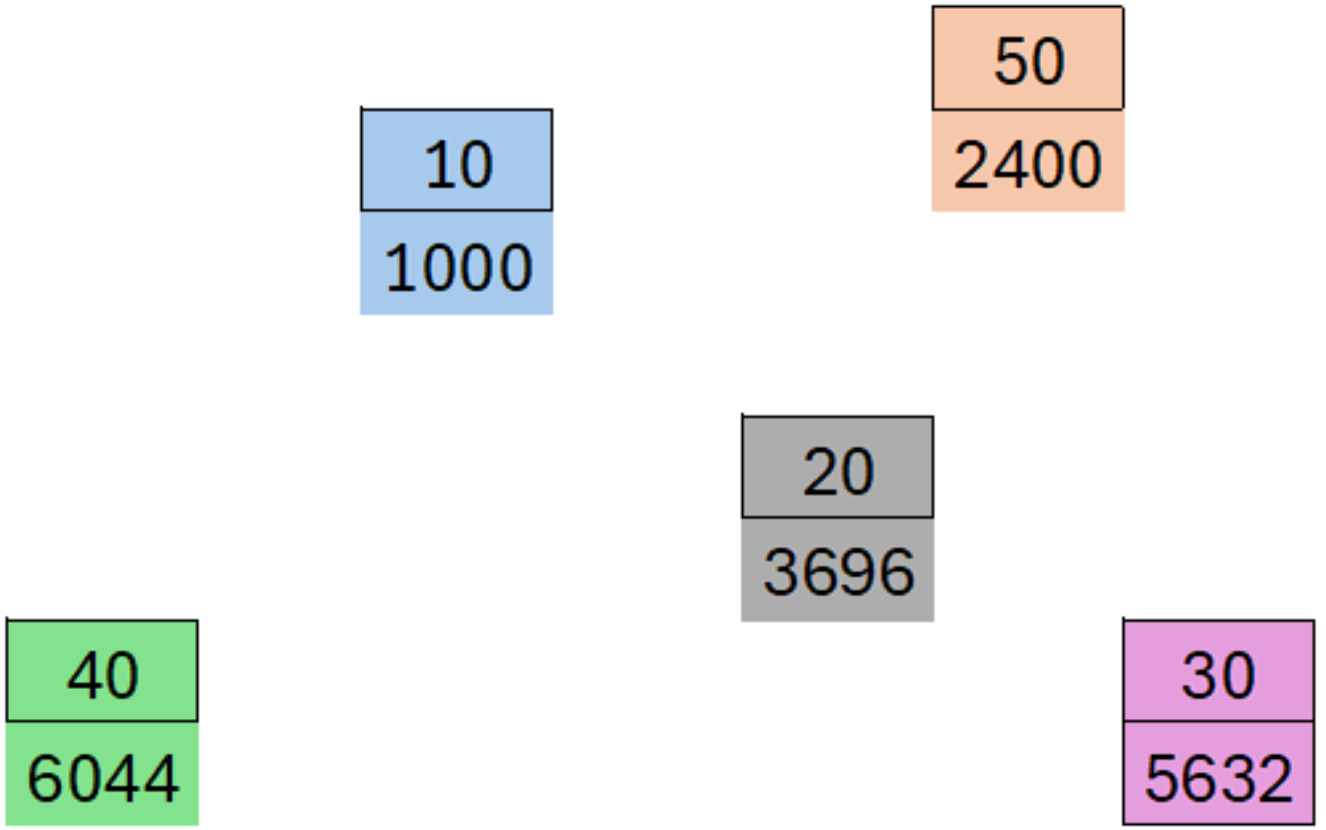
A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null.

Array (Elements at contiguous memory locations)



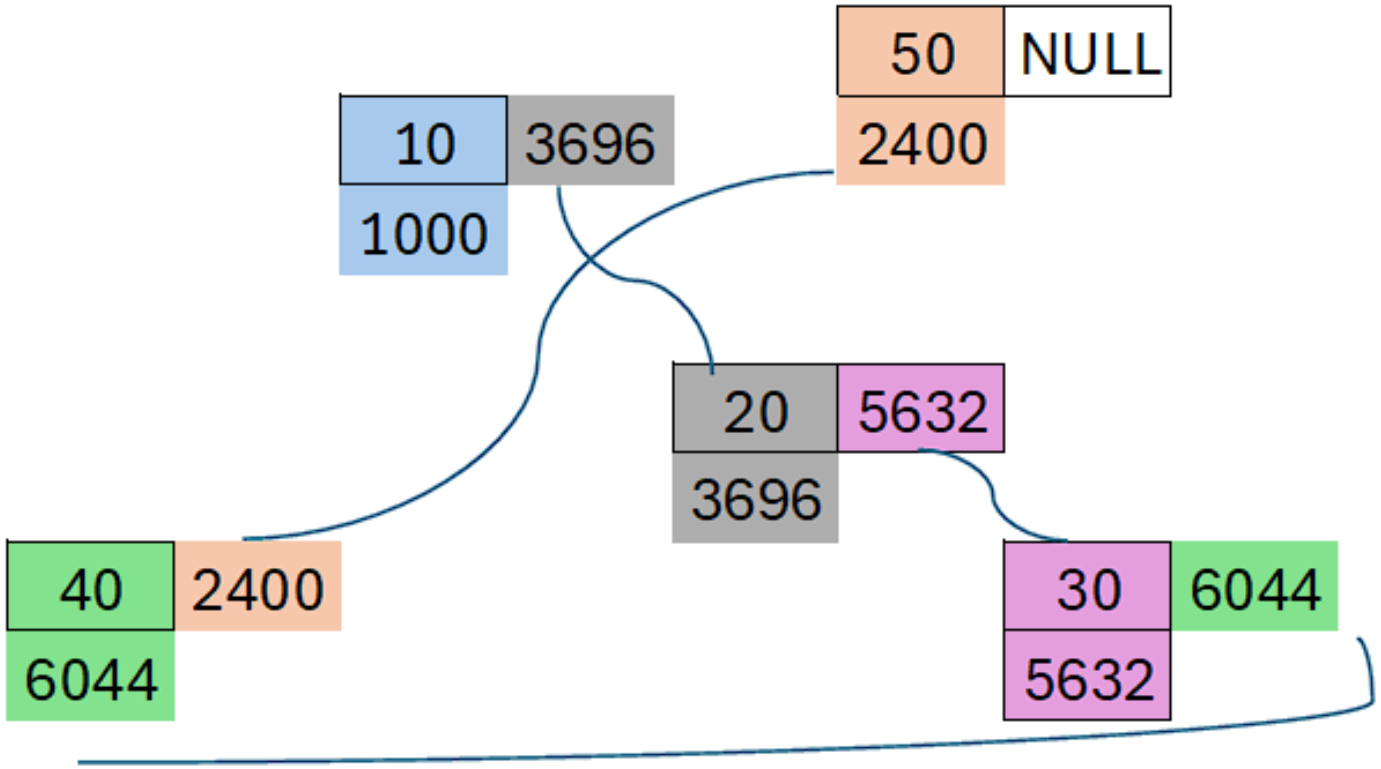
Linked List (Elements stored at random memory locations)

- Elements stored at different addresses. But **How to go from one element to another?**



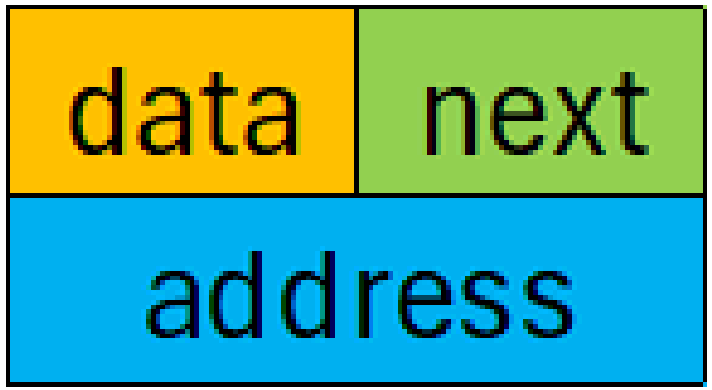
Linked List (Elements stored at random memory locations)

- The idea is to somehow store the address of the other element along side the value of current element.



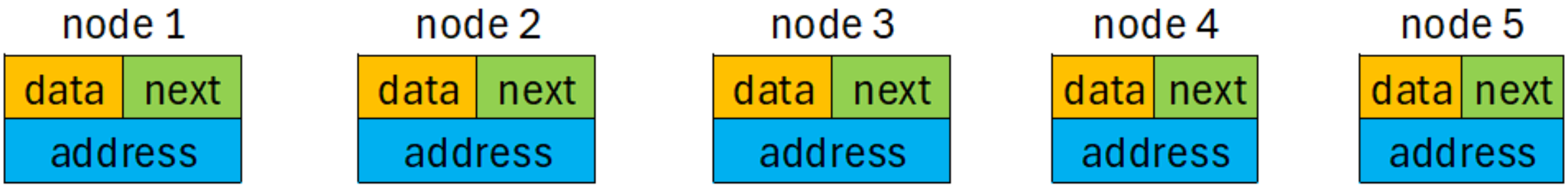
Linked List Nodes

- Linked list node contains **2 parts**
  - **Data Part** (To store the value)
  - **Next Part** (To store the address of the next element)
- Here's how a linked list node will look like



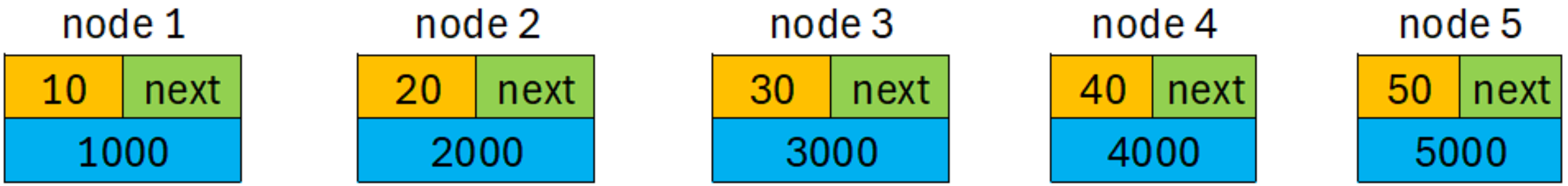
Linked List Nodes

- This is how a linked list representation of 5 integers look like



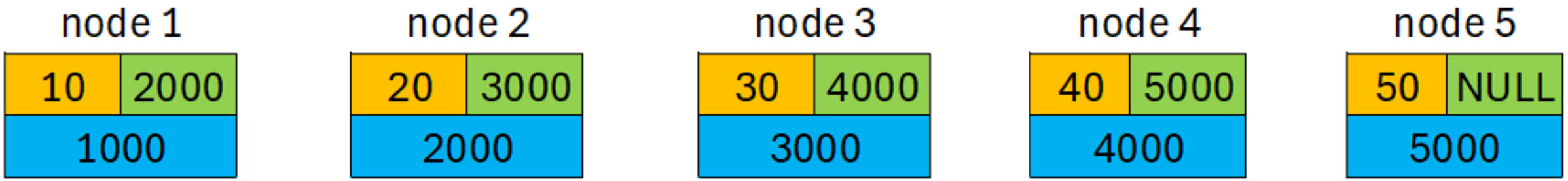
Linked List Nodes

- Every node contains its own address and can have some value at data part.



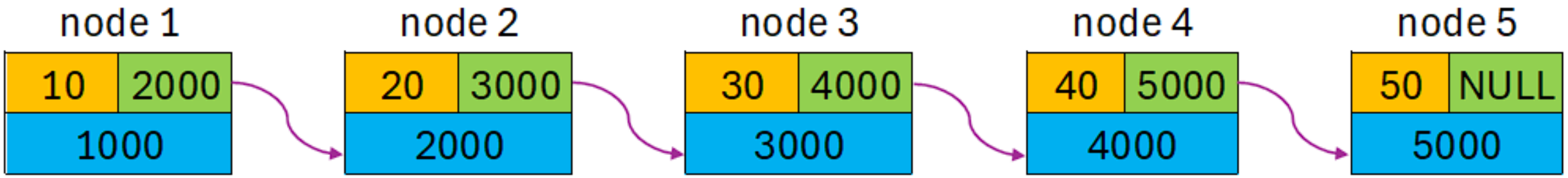
Linked List Nodes

- Now comes the linking part. The idea is to **store the address of NEXT NODE in PREVIOUS NODE to maintain connectivity.**

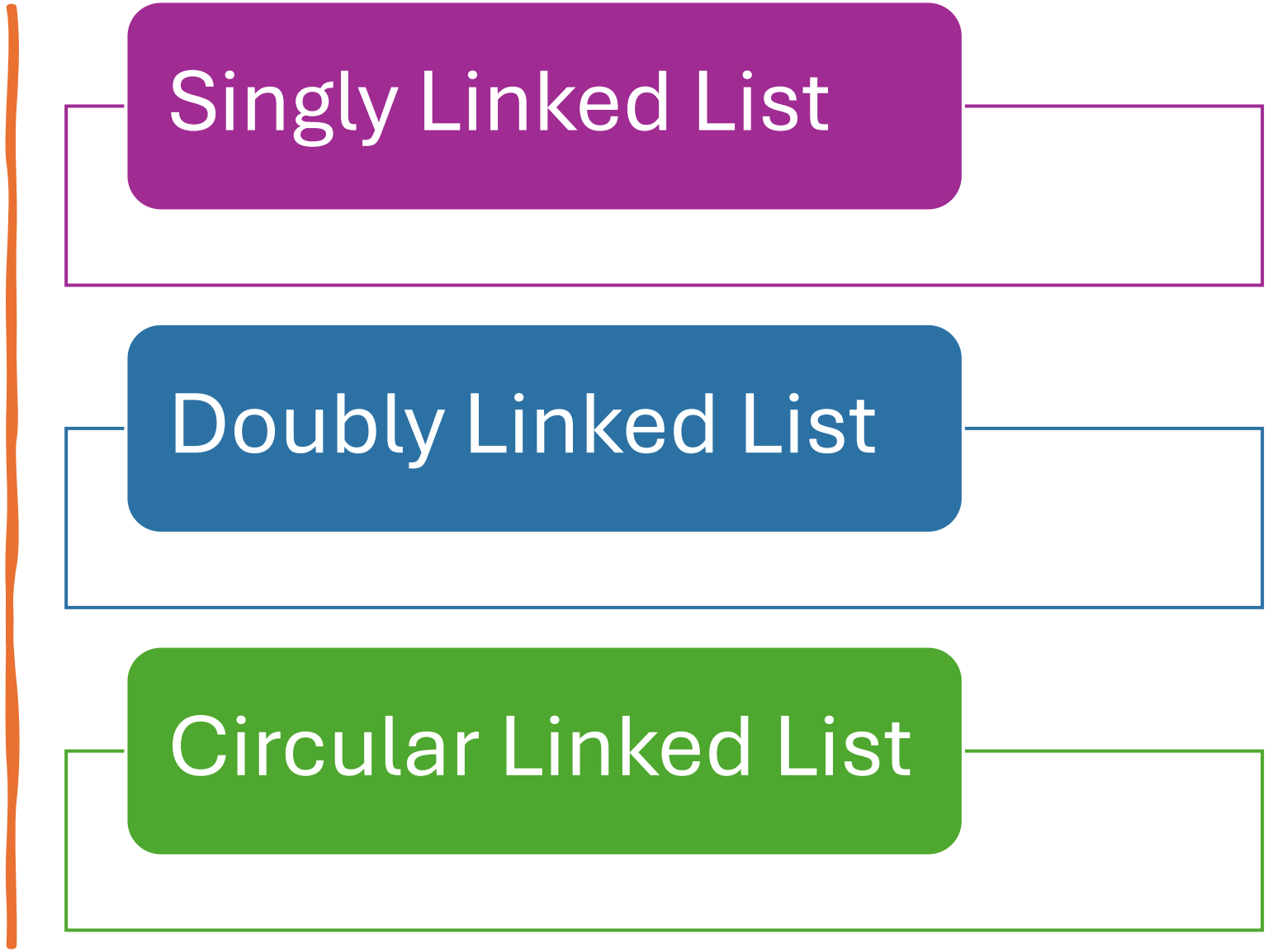


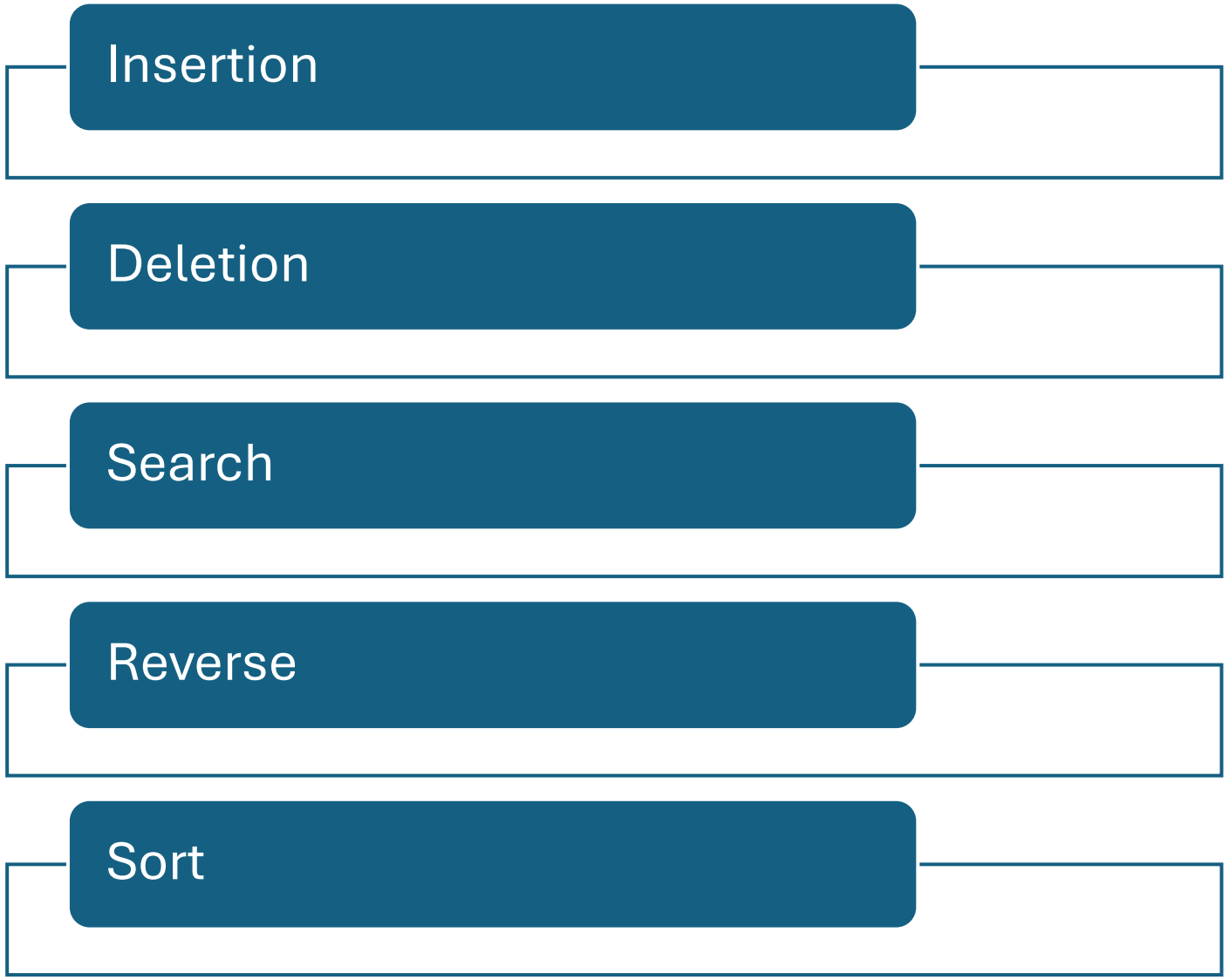
Linked List Nodes

- By doing this we give connection from one node to another. **Last node is connected to nothing else so we store NULL.**



Types of Linked Lists





Linked List Operations

Advantages of Linked Lists

Dynamic and Memory Efficient

Linked List doesn’t have a fixed size.

Based on the requirement, **N** number of nodes can be added.

The size of linked list can grow or shrink according to the requirement. So there won’t be any scope for wasting of memory.

Insertions & Deletions are relatively easy

Unlike arrays, insertion, and deletion in linked list is easier.

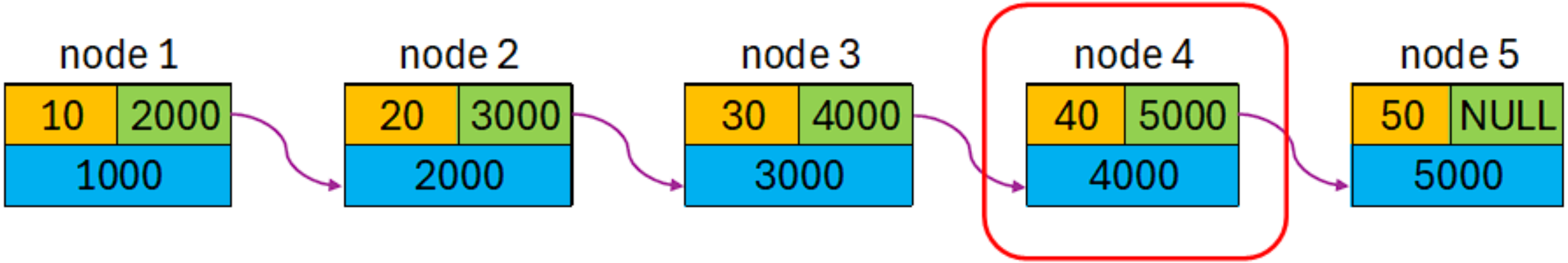
Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location.

To insert or delete an element in an array, we must shift the elements for creating the space.

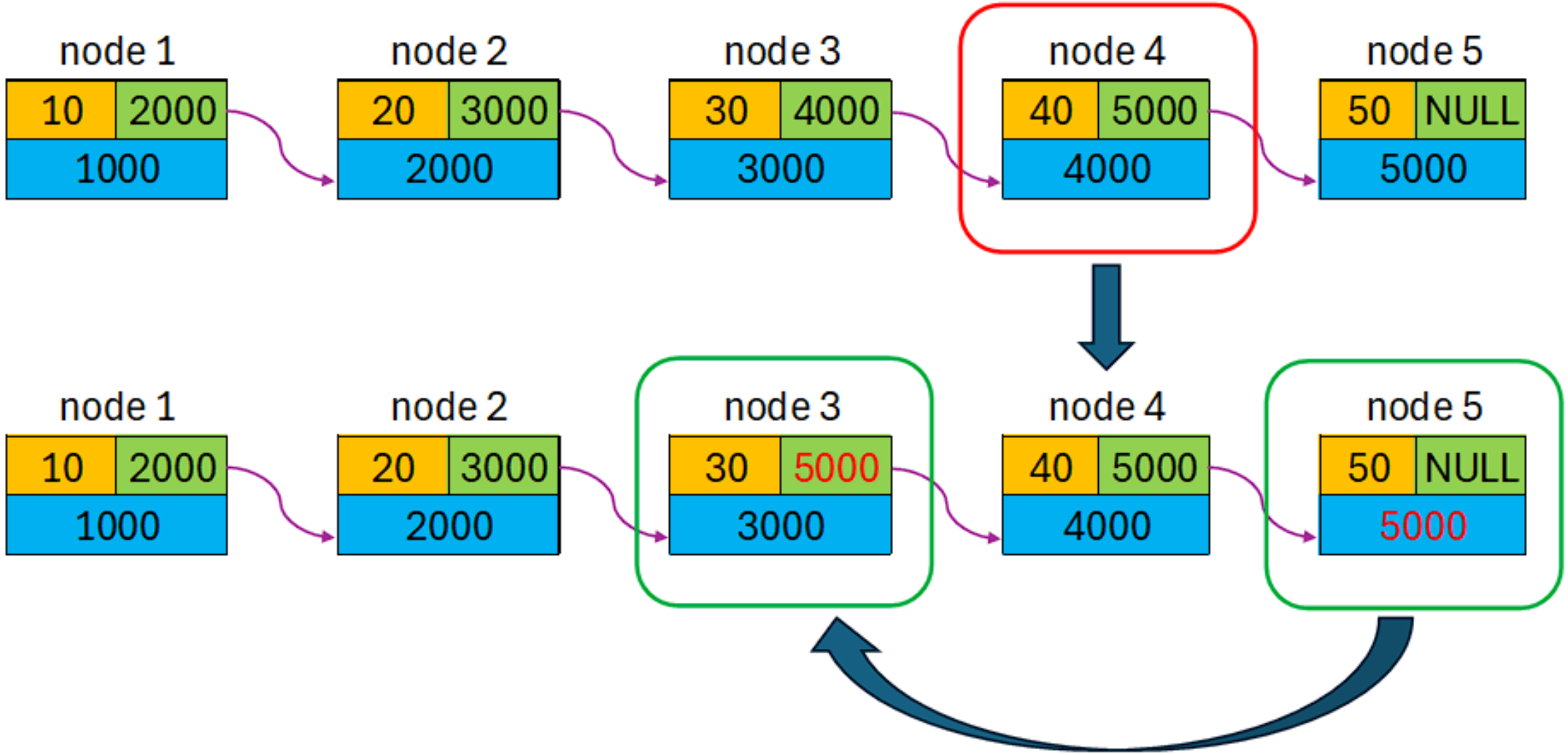
Whereas, in linked list, instead of shifting, we can just update the address of the pointer of the node.

Deleting an element from a linked list

- Say we want to delete the 4th node from the below linked list



Deleting an element from a linked list



Disadvantages of linked lists

Memory usage - In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.

Traversal - Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.

Reverse traversing - Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of linked lists

- **Polynomial Representation**
- **Sparse Matrix Representation**
- **Implementations of other data structures like stacks and queues**
- **Adjacency List representation for Graphs**

Where are linked lists being used?

- Image viewers
- Previous and next pages in web browsers
- Music player
- Forward and backward in File explorers and many more

Stacks

Stack

(Last In First Out (LIFO) or
(First In Last Out (FILO)) Structures

01

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle

02

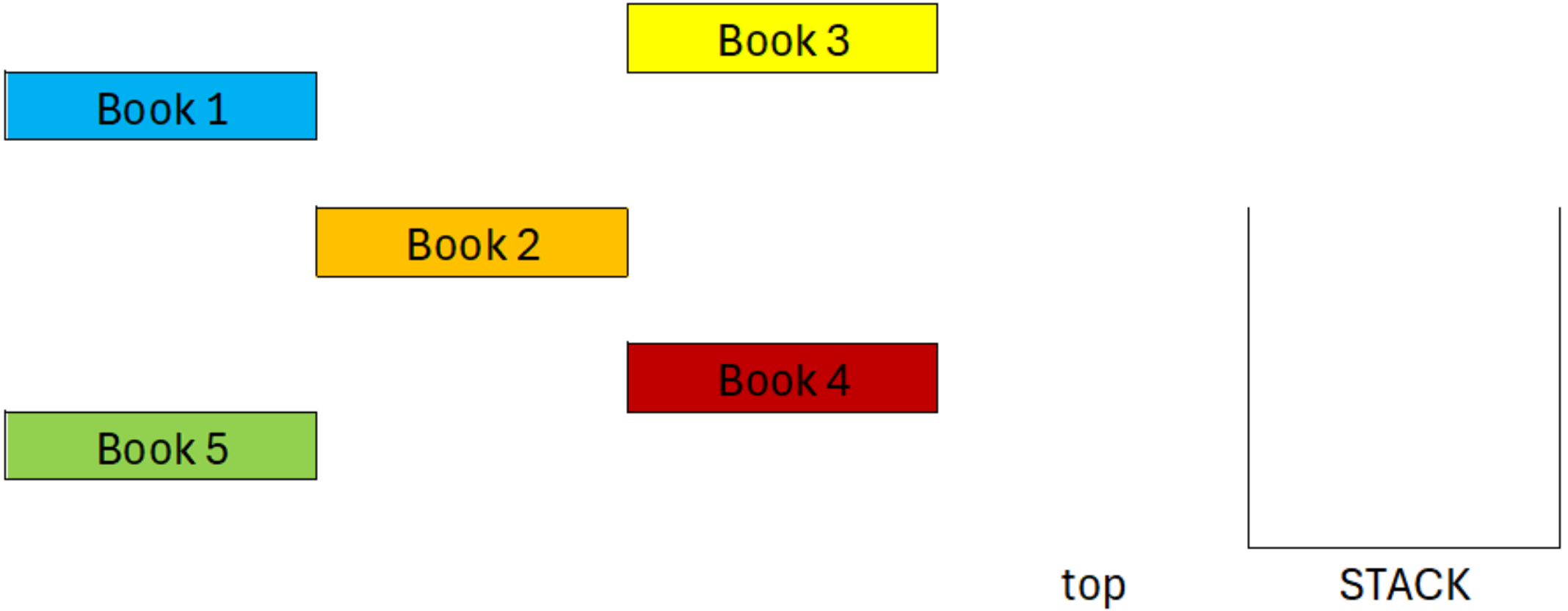
The element inserted last will be the first element to be taken out.

03

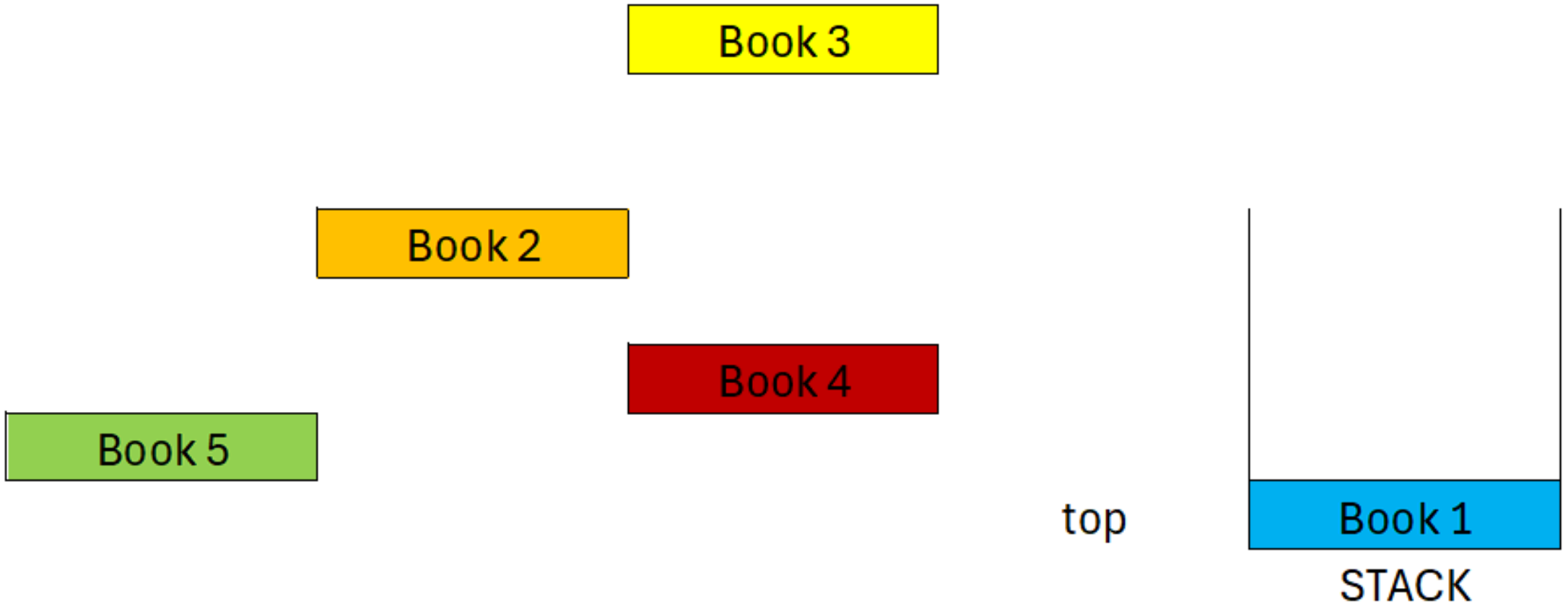
Operations on stack will be done from only ONE END called

top

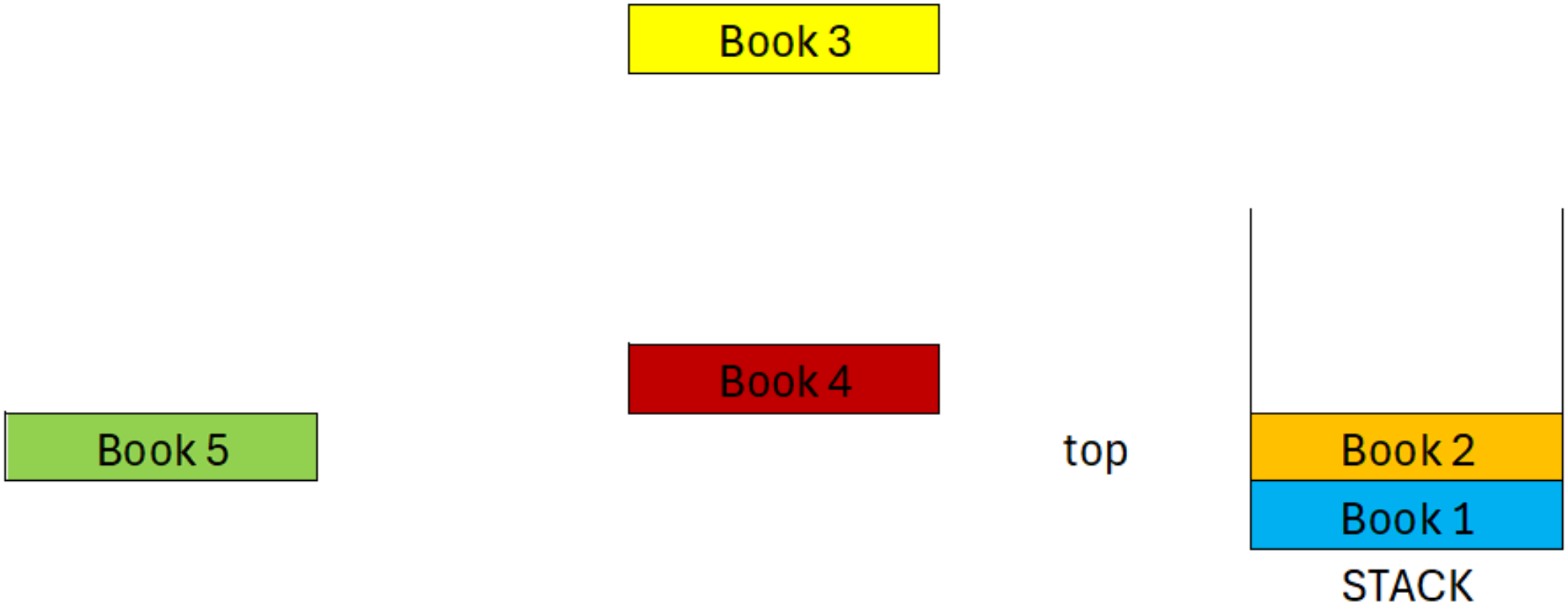
Here’s an example (Organizing Books)



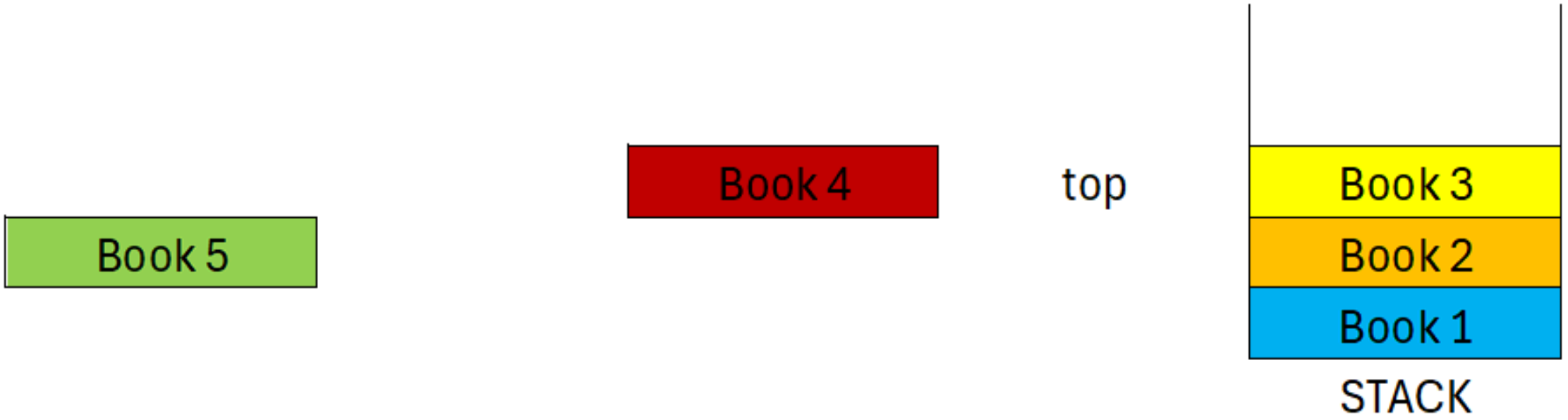
Inserting 1st Book



Inserting 2nd Book

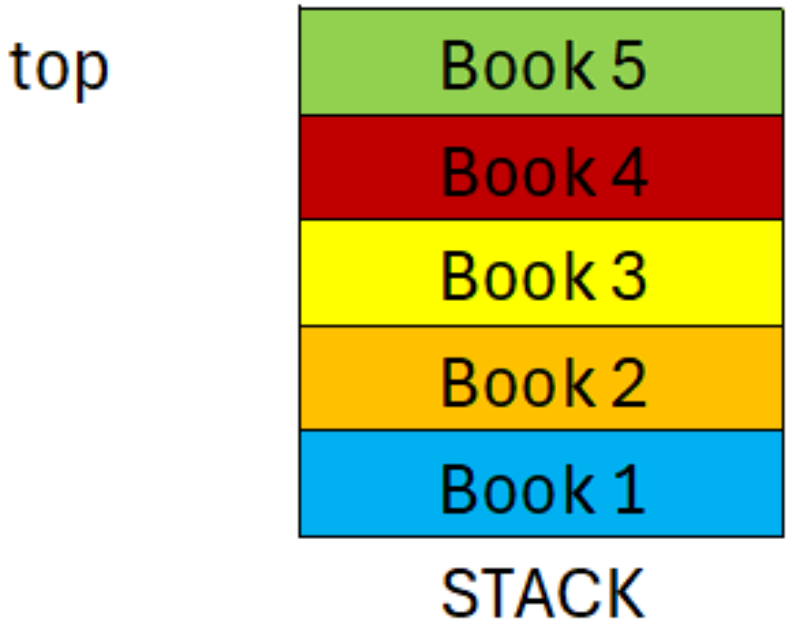


Inserting 3rd Book

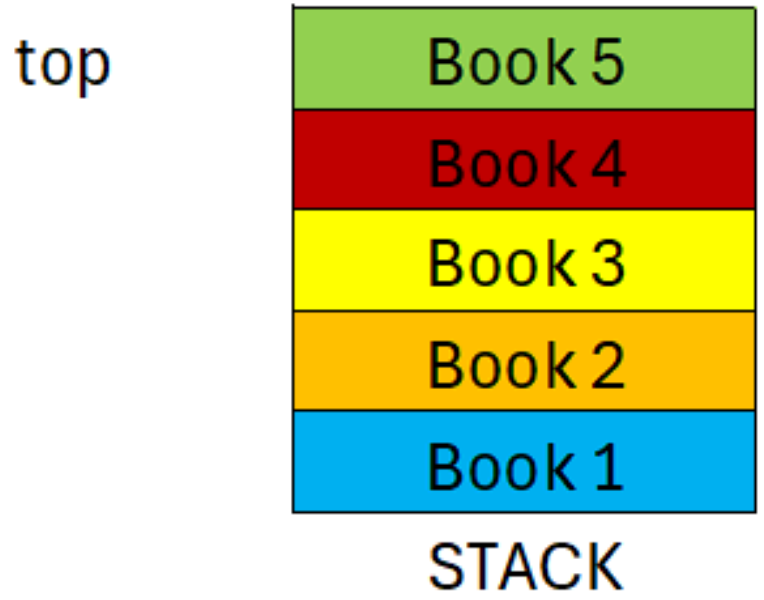


Inserting 4th Book

Inserting 5th Book

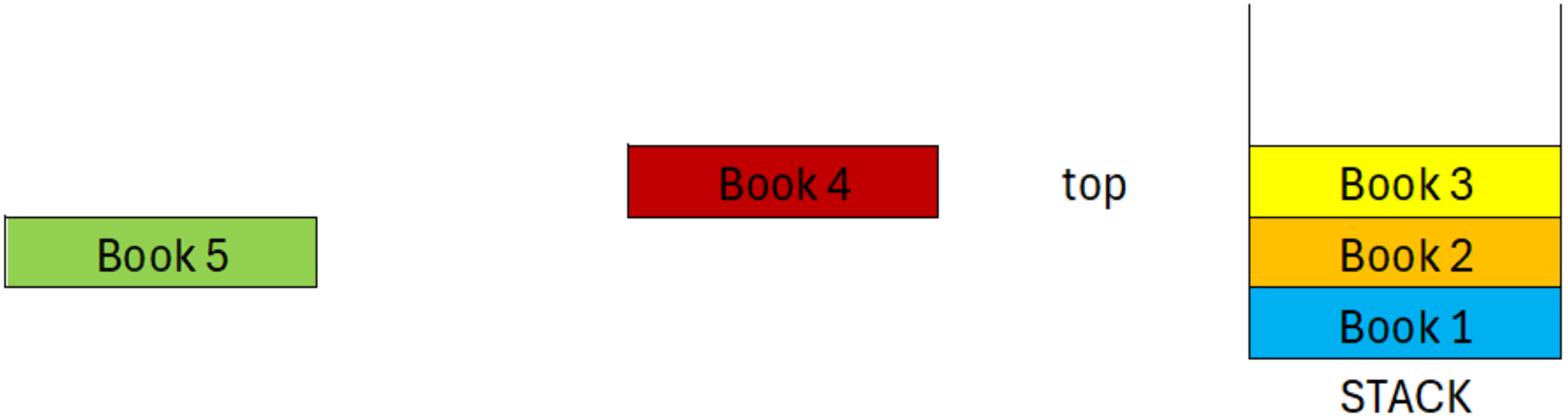


Now deleting the books inserted

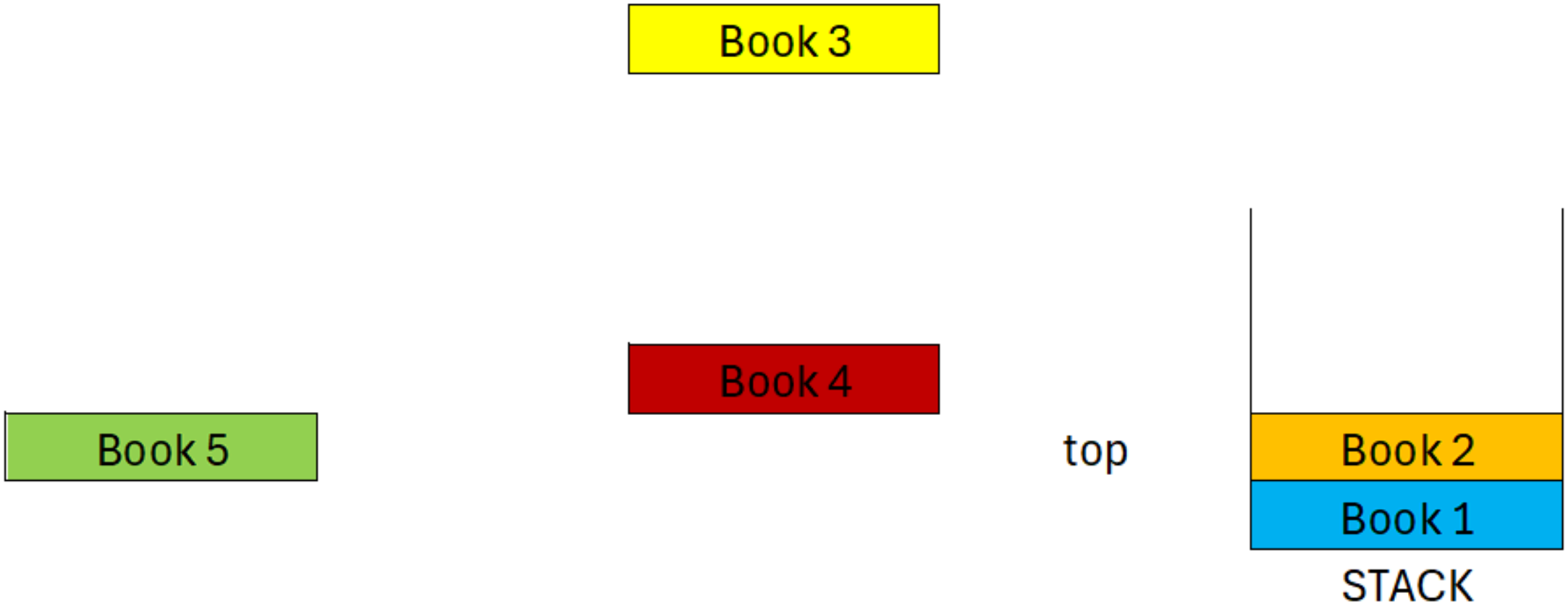


Deleting Book 5 (Which was inserted as the LAST element into the stack)

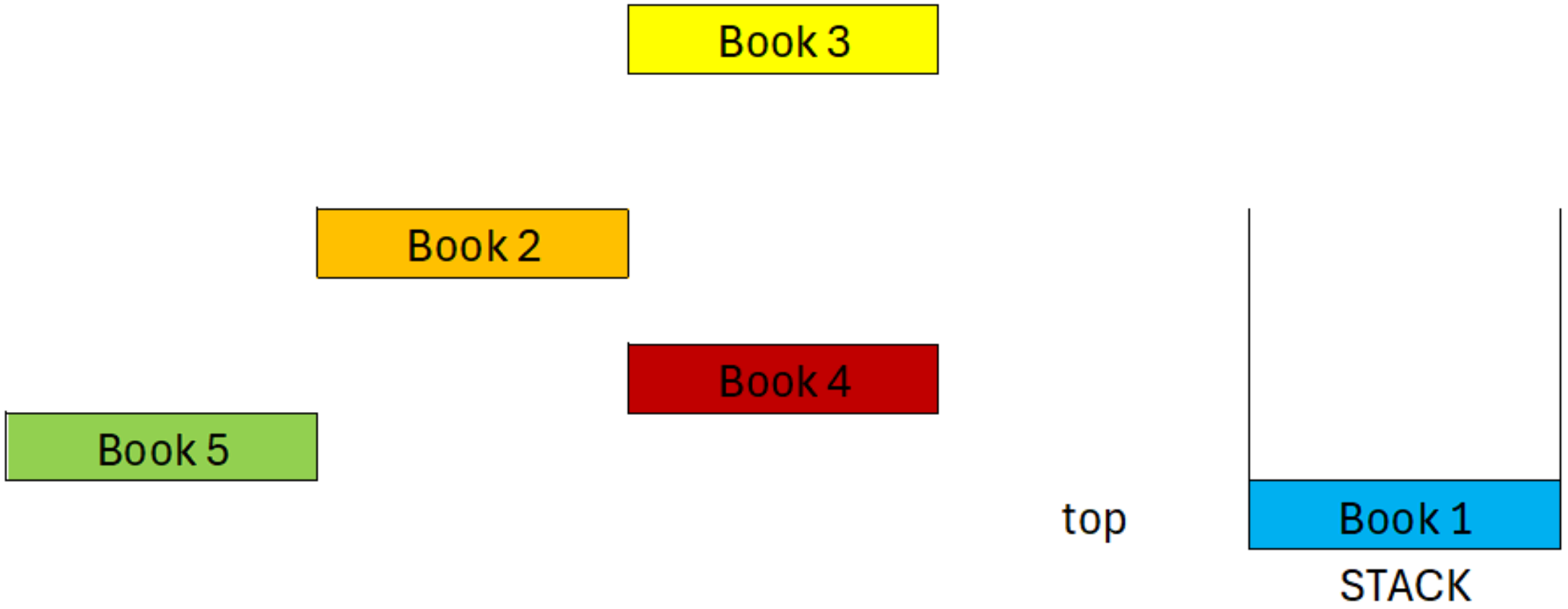
Deleting Book 4



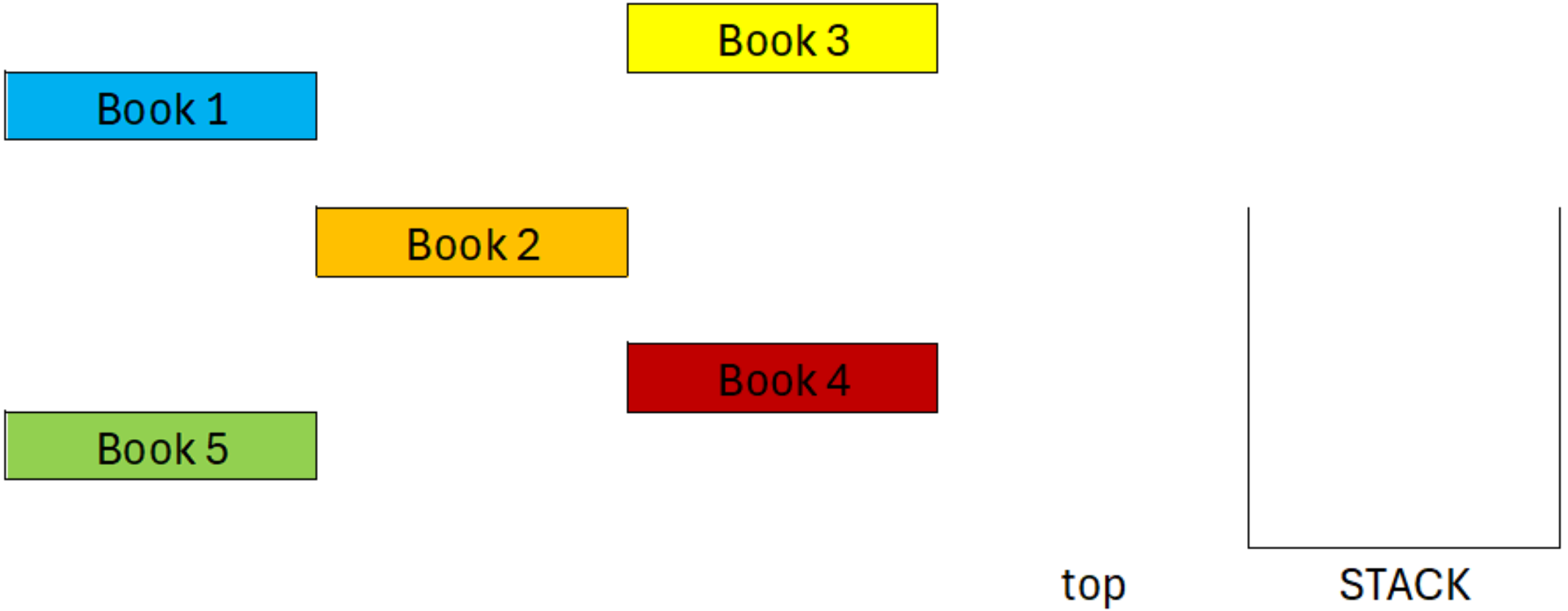
Deleting Book 3



Deleting Book 2



Deleting Book 1 (Thus by making Stack empty)



Stack Operations

Push(): Inserting a new element into the stack

Pop(): Deleting an element from the stack

isEmpty(): Checking if stack is empty (Underflow Condition)

isFull(): Checking if stack is full of elements (Overflow Condition)

peek(): Having a look at the element currently on **top**

display(): Printing all the stack elements.

Stacks can be implemented using

An Array

A Linked List

Applications of Stack

- Arithmetic Expression Evaluations
- Balancing parentheses
- Recursion
- Backtracking
- Reversal Operations
- Undo / Redo Operations in Text Editors

Queue

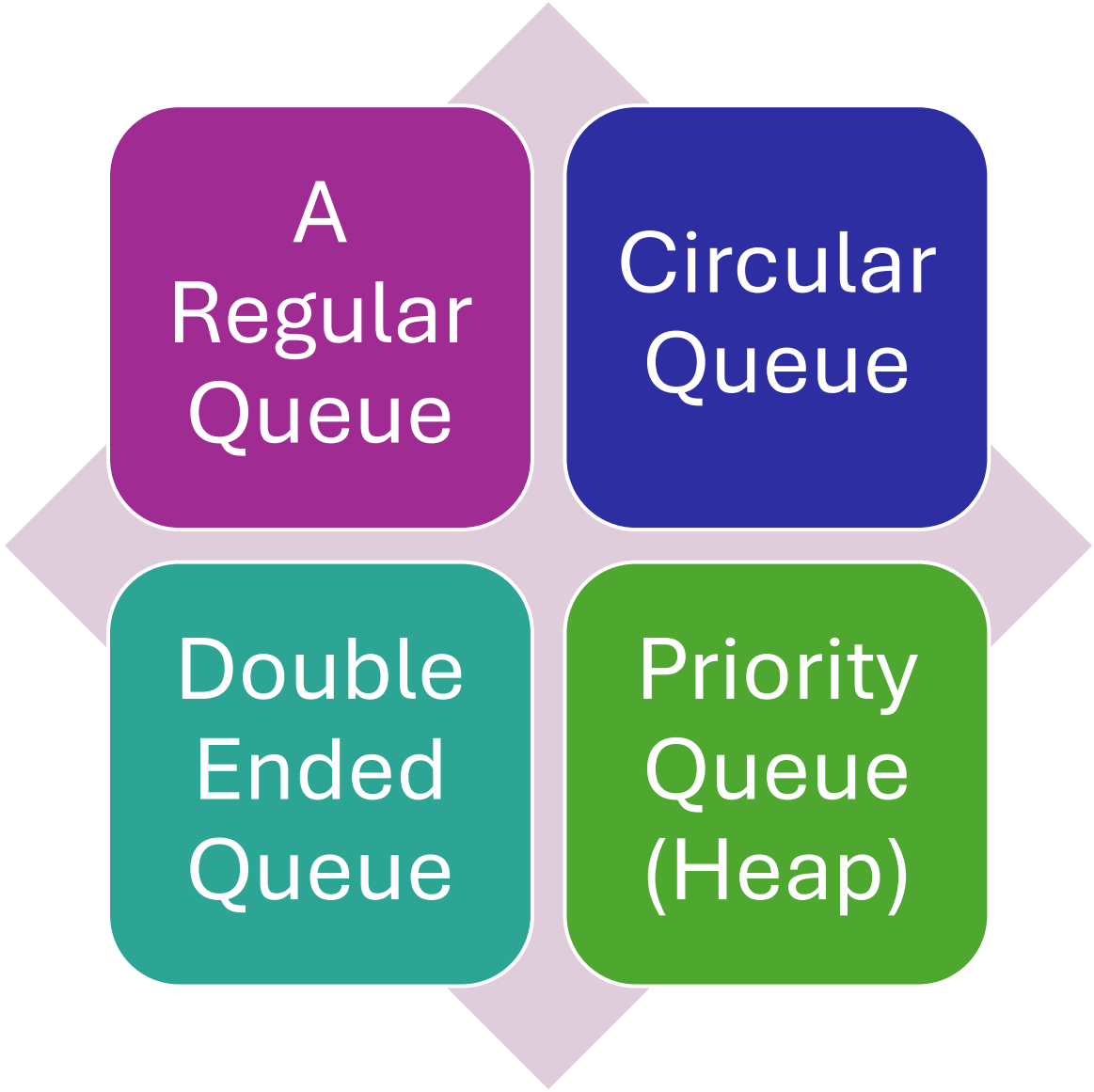
Queue Data Structure

Also called as **First In First Out (FIFO)** structure

Data is organized in **FIRST COME FIRST SERVE (FCFS)** nature.

The element that enters first will be the first one to be processed.

Types of Queues



Operations on Queues

Enqueue

- Inserting a new element into the queue

Dequeue

- Removing an element from the queue

Peek

- Accessing the element to be processed (without removing it)

Operations on Queues

Display

- Looking at all the elements in the queue

IsEmpty

- Checking if the queue is empty

IsFull

- Checking if the queue is full

Queues can be implemented using

An Array

A Linked List

Common terms in the implementation of Queues

Front

- Used to perform deletions (Dequeue Operation)

Rear

- Used to perform insertions (Enqueue Operation)

Simulation of a Queue Data Structure

An Empty Queue (size – 5)

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🧛)

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🧛)

🧛				

Front

Rear

Simulation of a Queue Data Structure

Enqueue ()

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🧟)

🧛				

Front

Rear

Simulation of a Queue Data Structure

Enqueue ()

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🧟)

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🦹)

🧛	🧟			

Front

Rear

Simulation of a Queue Data Structure

Enqueue ()

Front

Rear

Simulation of a Queue Data Structure

Enqueue ()

Front

Rear

Simulation of a Queue Data Structure

Enqueue ()

Front

Rear

Simulation of a Queue Data Structure

Enqueue ()

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🤱)

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🤱)

Front

Rear

Simulation of a Queue Data Structure

Enqueue (🦕)

Front

Rear

Simulation of a Queue Data Structure

Dequeue()

Front

Rear

Simulation of a Queue Data Structure

Dequeue()  -> Processed

Front

Rear

Simulation of a Queue Data Structure

Dequeue()

Front

Rear

Simulation of a Queue Data Structure

Dequeue()  -> Processed

Front

Rear

Simulation of a Queue Data Structure

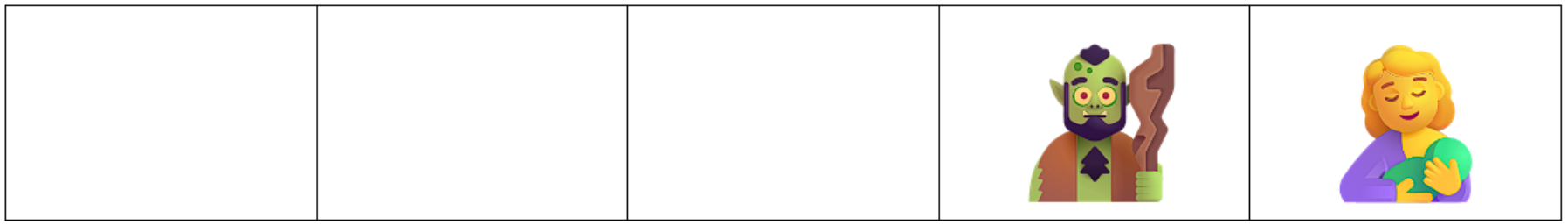
Dequeue()

Front

Rear

Simulation of a Queue Data Structure

Dequeue()  -> Processed



Front

Rear

Simulation of a Queue Data Structure

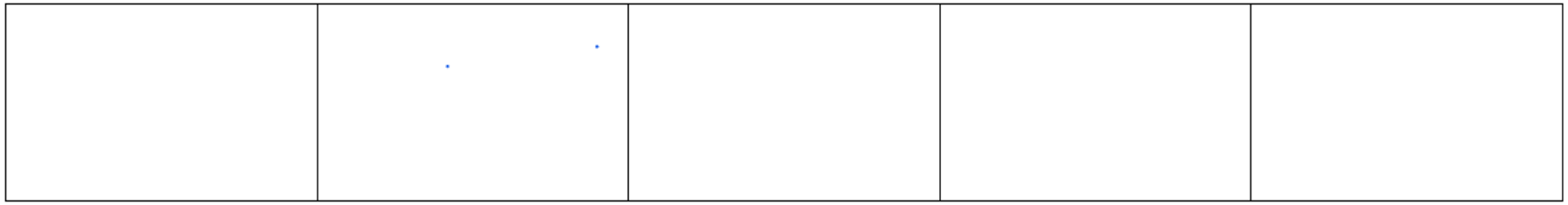
Dequeue()  -> Processed

Front

Rear

Simulation of a Queue Data Structure

Dequeue()  -> Processed



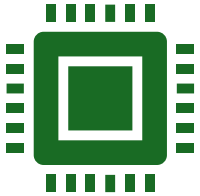
Front

Rear

Applications of Queues

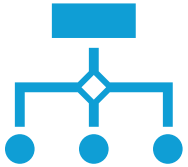
Task Scheduling

Queues can be used to schedule tasks based on priority or the order in which they were received



Resource Allocation

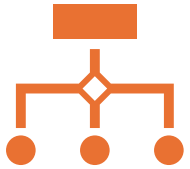
Queues can be used to manage and allocate resources, such as printers or CPU processing time



Message Buffering

Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks

Applications of Queues



Event Handling

Queues can be used to handle events in event-driven systems, such as GUI applications

Printer Queues

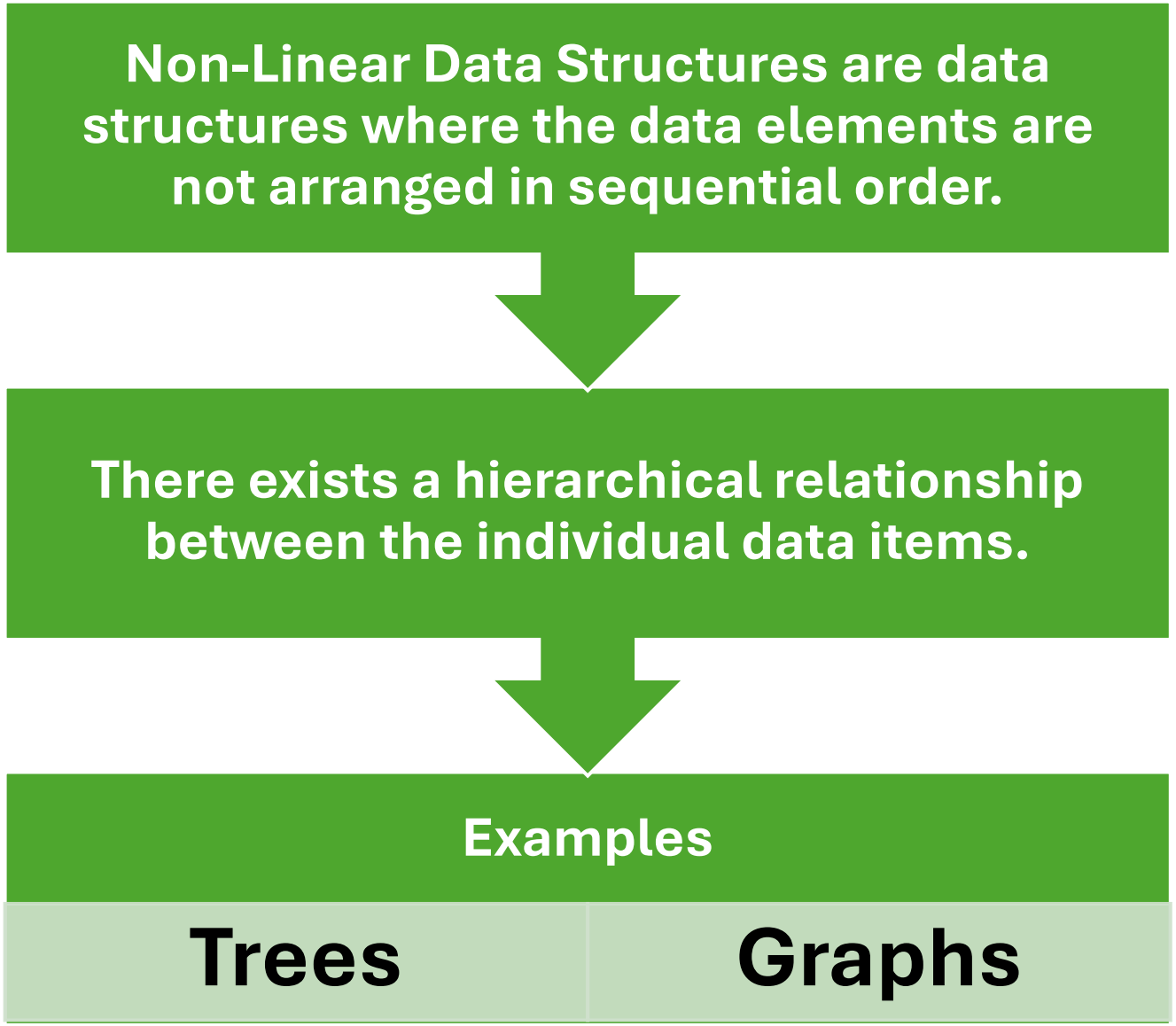
In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.

Breadth-first search algorithm

The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level

Non - Linear Data Structures

Non Linear Data Structures

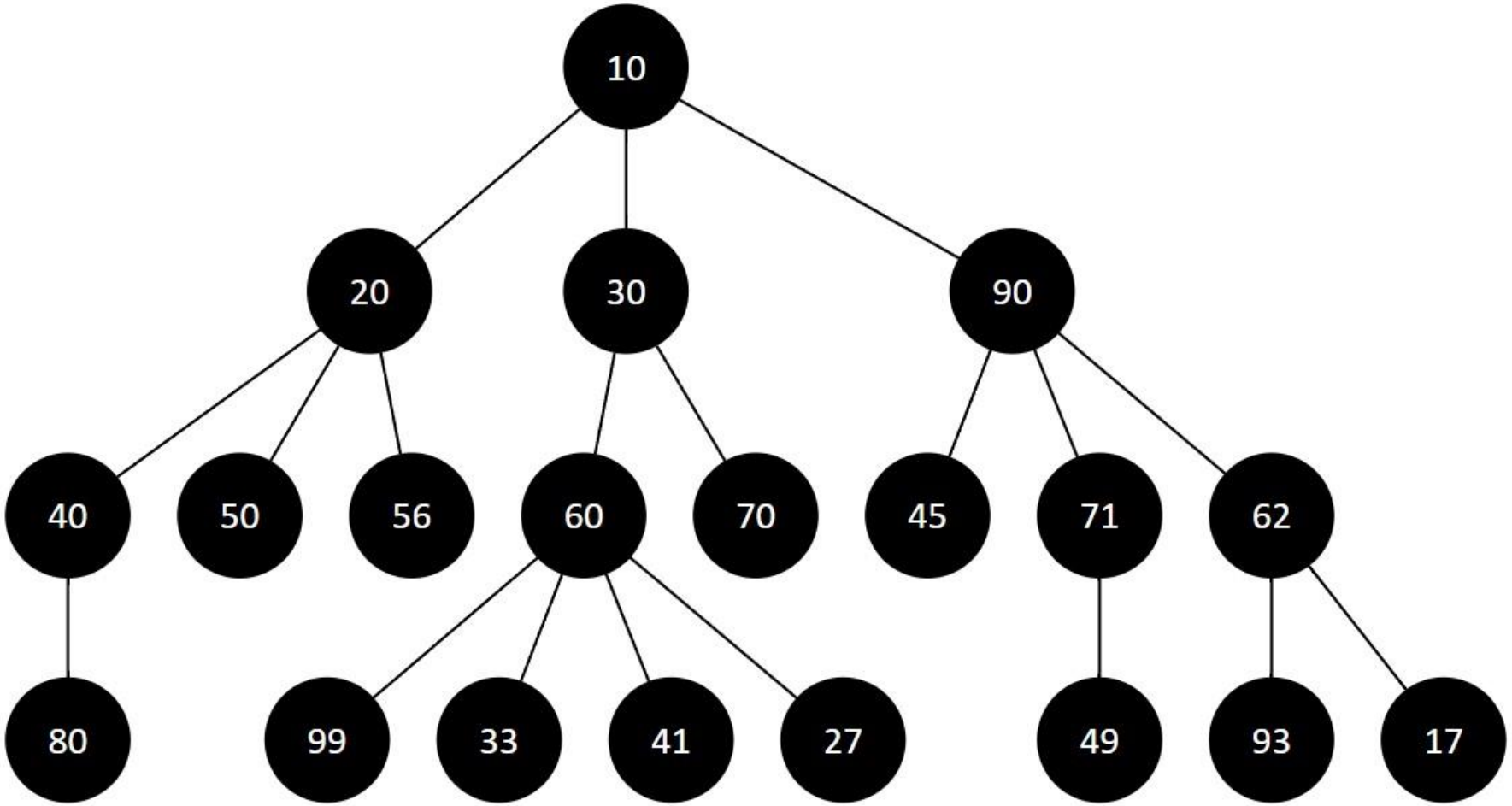


Tree Data Strcture

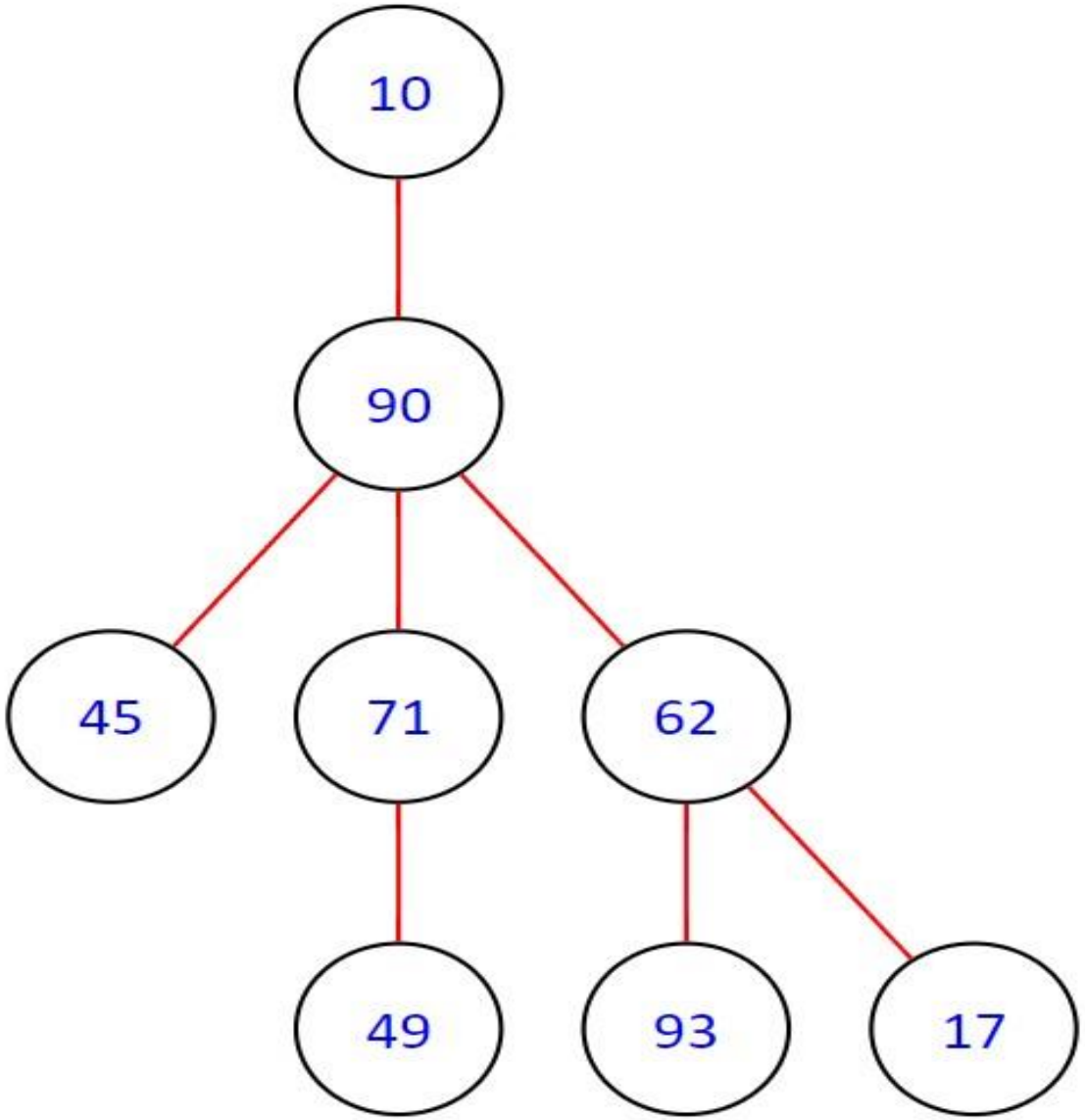
Definition

- A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the “children”).

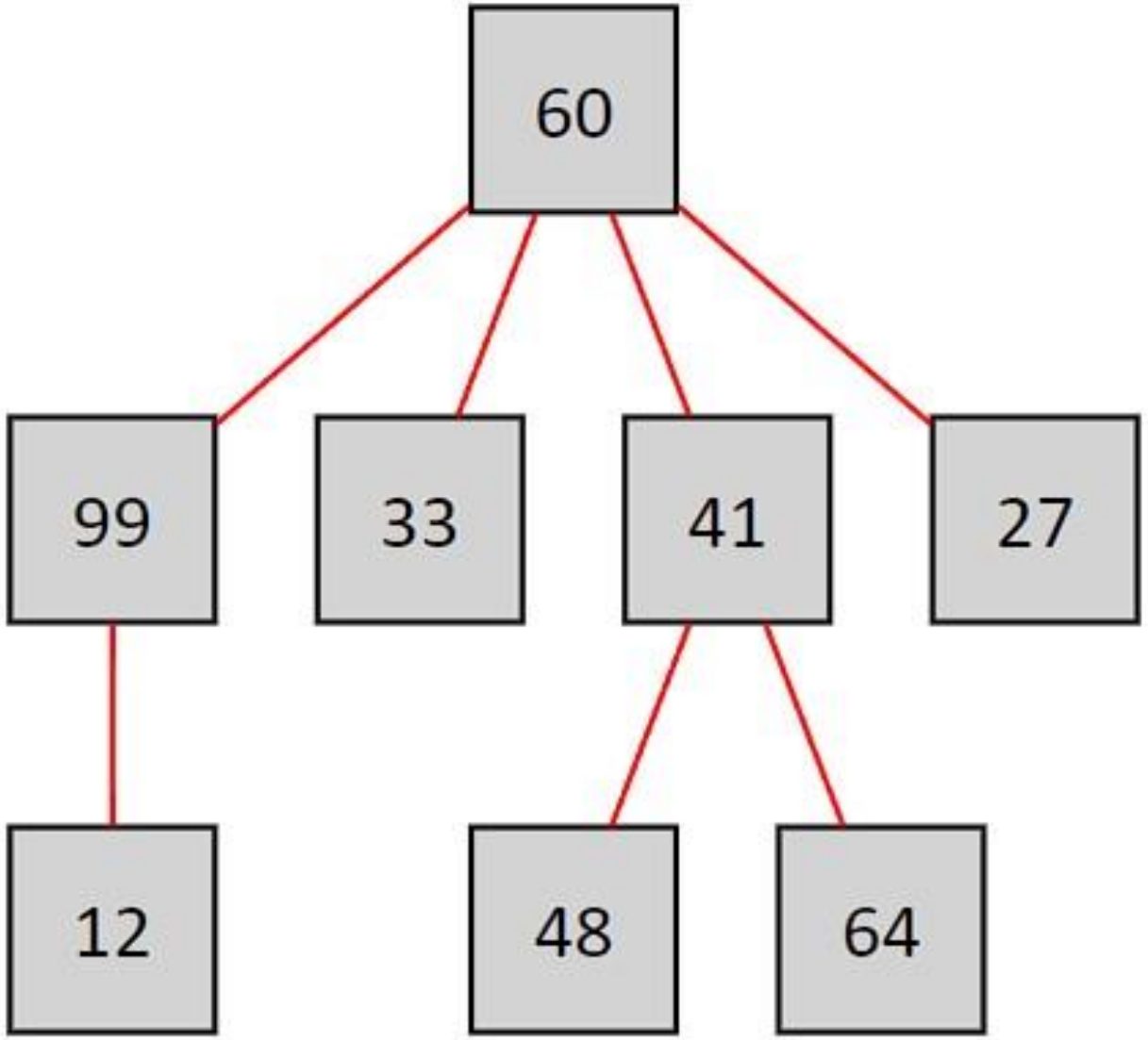
Examples of some trees



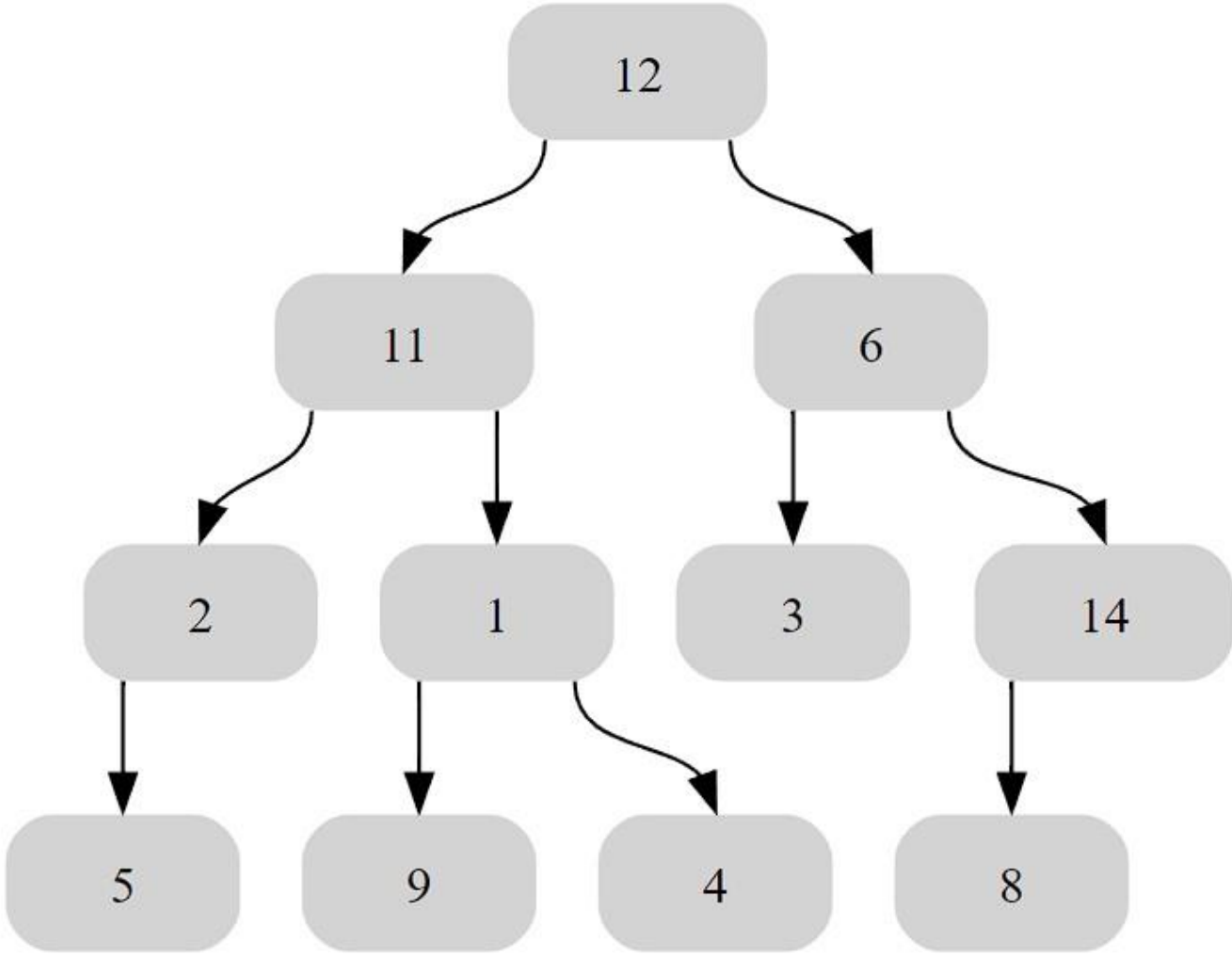
Examples of some trees



Examples of some trees



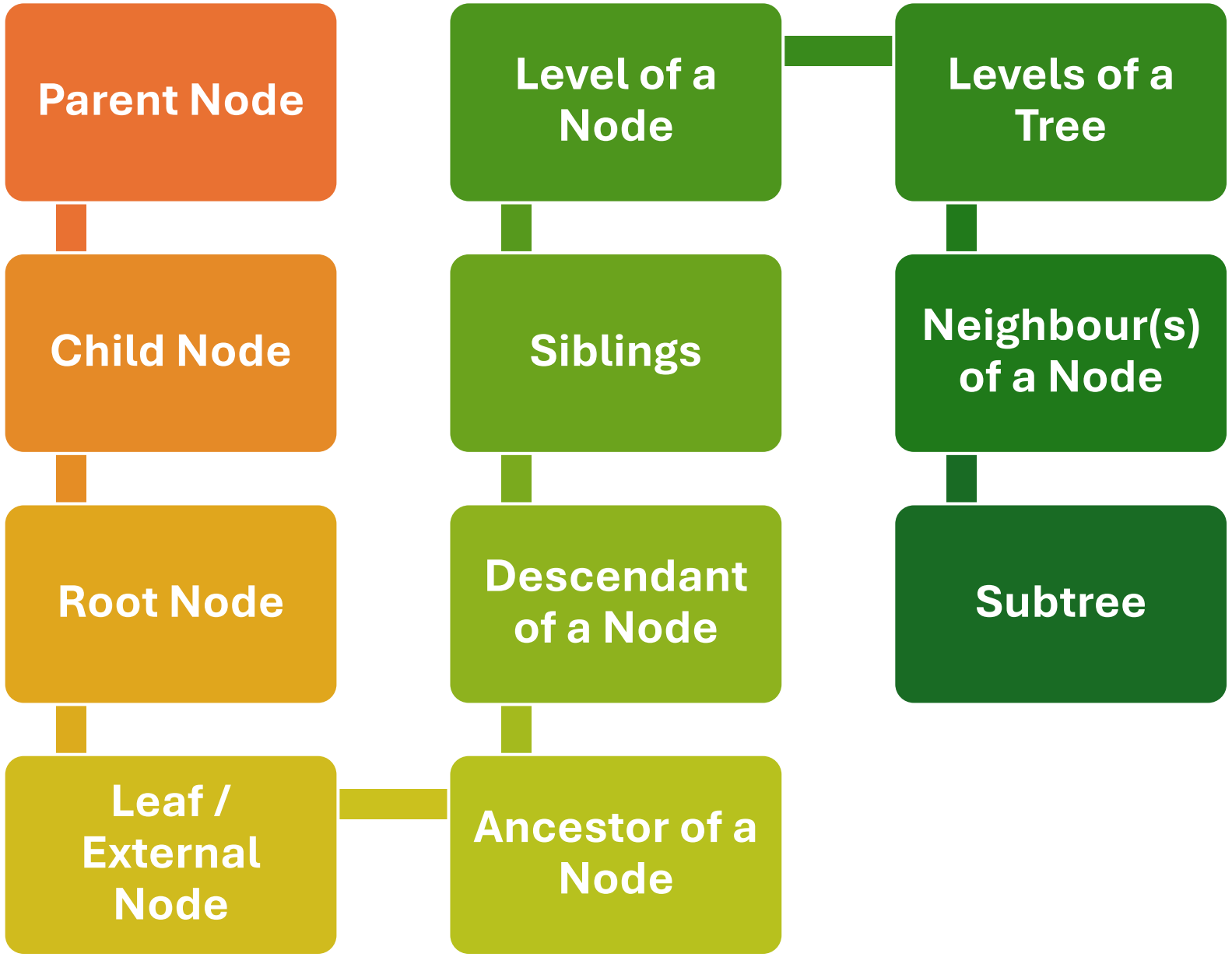
Examples of some trees



Why are Trees considered as Non-Linear Data Structure ??

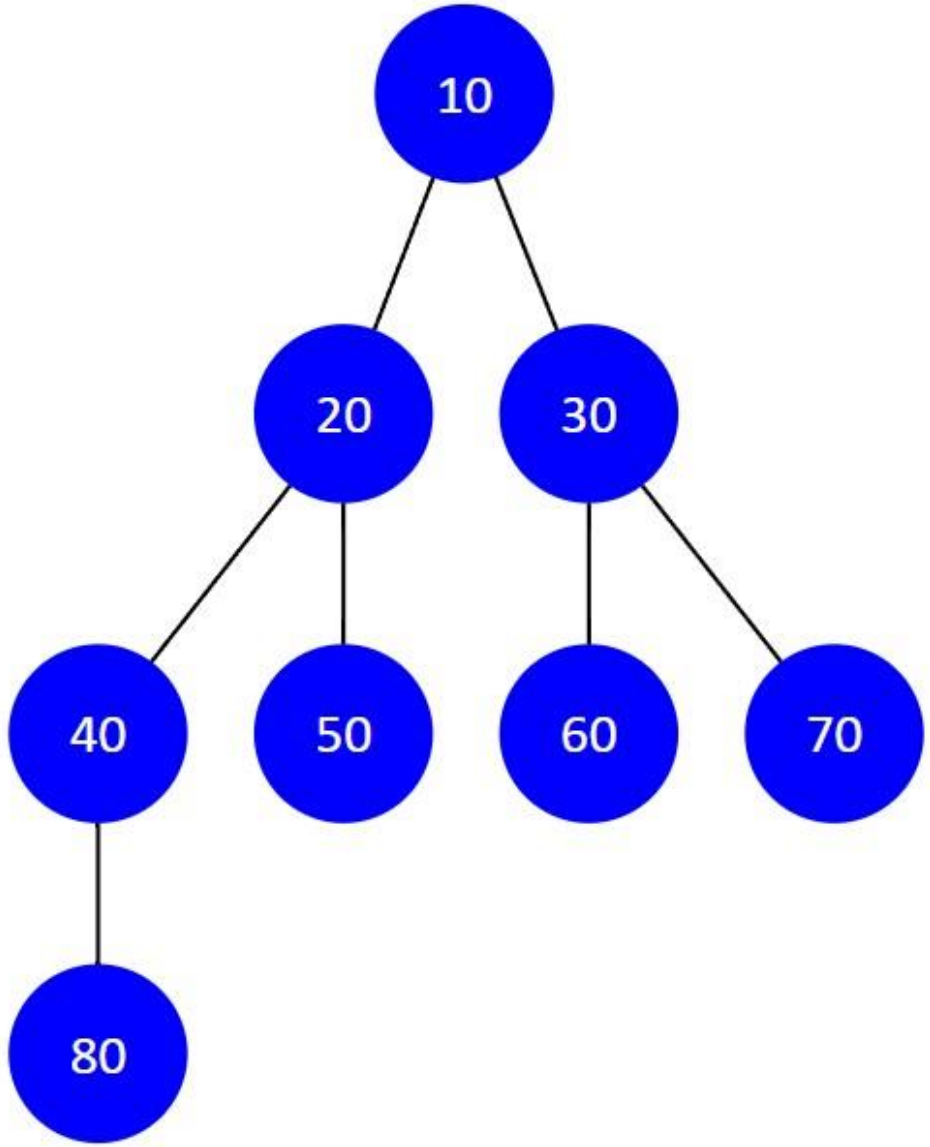
- The data in a tree are not stored in a sequential manner i.e, they are not stored linearly. Instead, they are arranged on multiple levels, or we can say it is a hierarchical structure. For this reason, the tree is a non-linear data structure.

Terminology



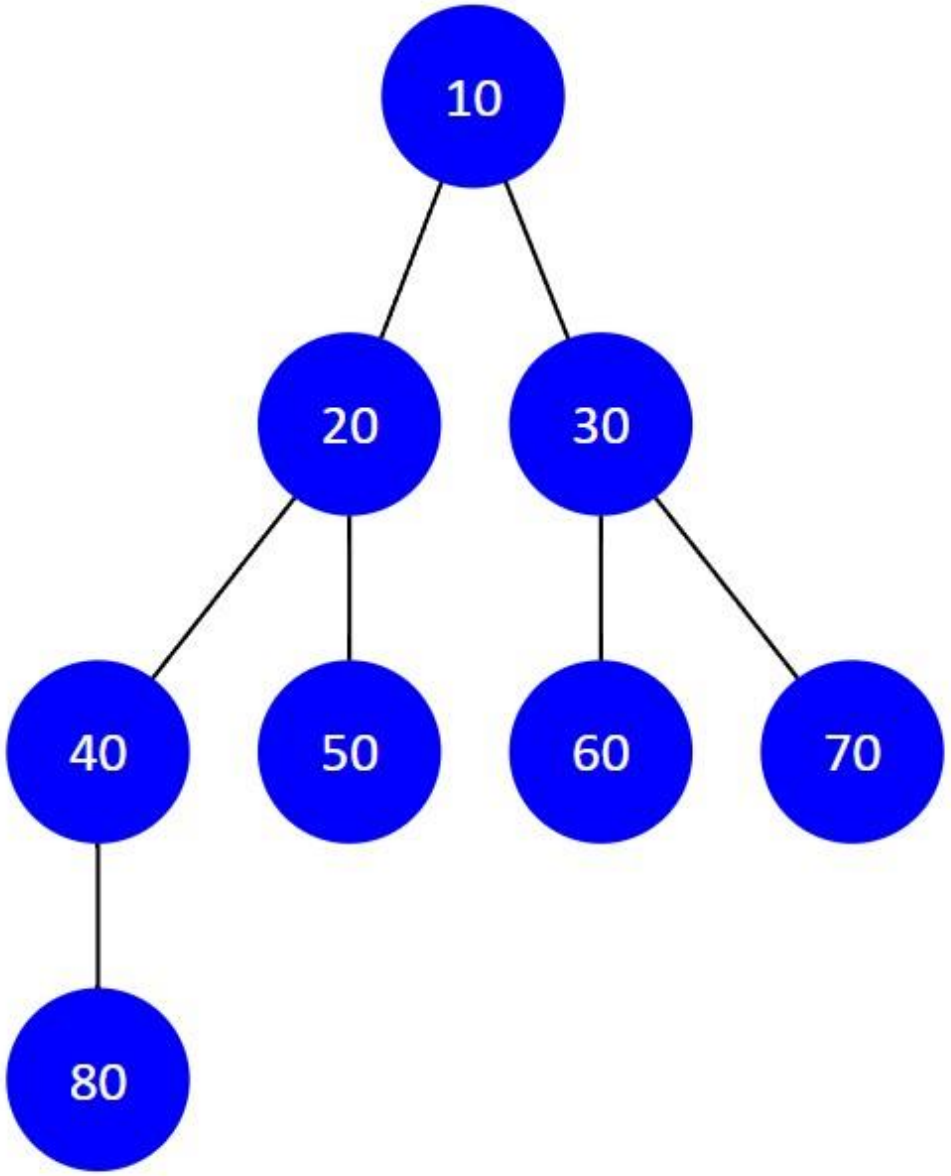
Parent Node

- The node which is an immediate predecessor of another node is called the parent node of that node.
- In the given example
  - **10 is parent node of both 20 and 30**
  - **20 is parent node of both 40 and 50**
  - **30 is parent node of both 60 and 70**
  - **40 is parent node of 80**



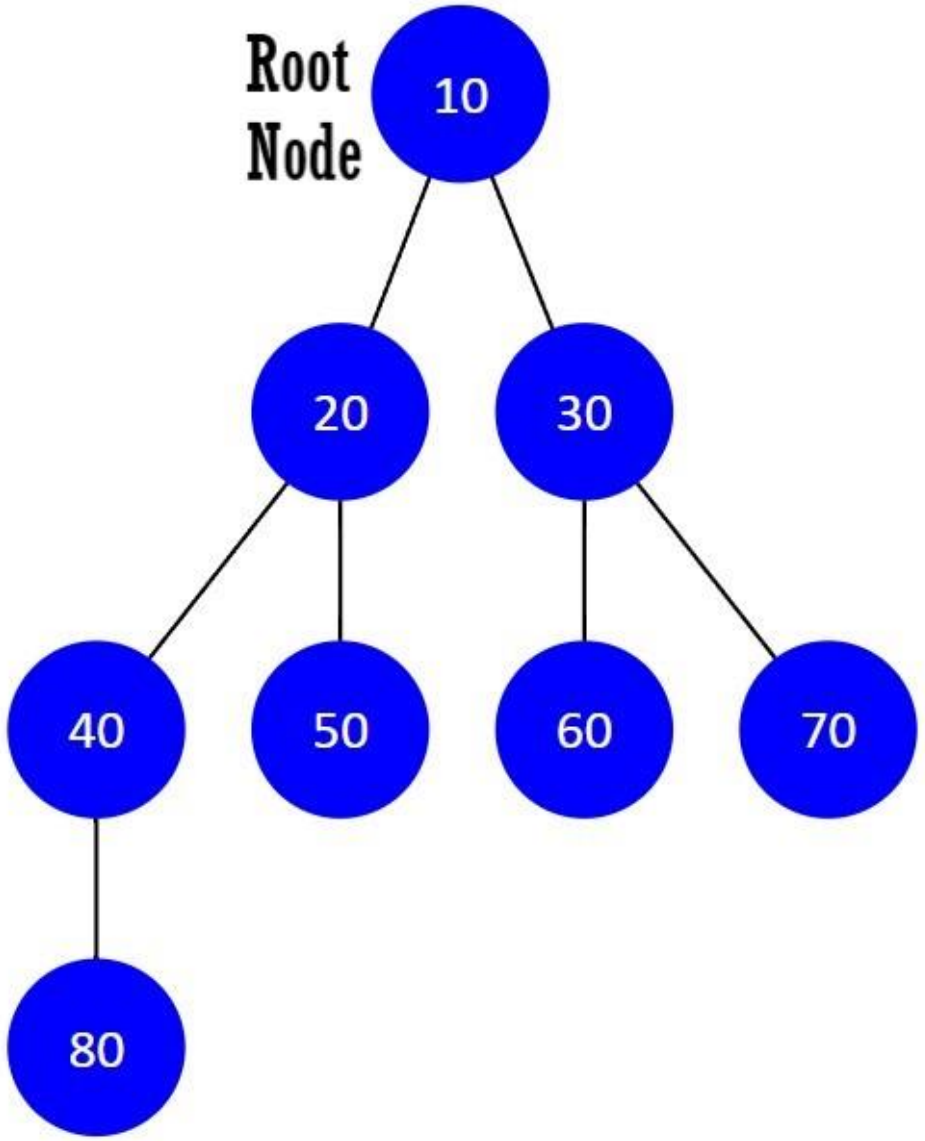
Child Node

- The node which is the **immediate successor** of a node is called the child node of that node.
- In the given example
  - {20, 30} are the child nodes of node 10
  - {60, 70} are the child nodes of 30
  - {80} is the child node of 40



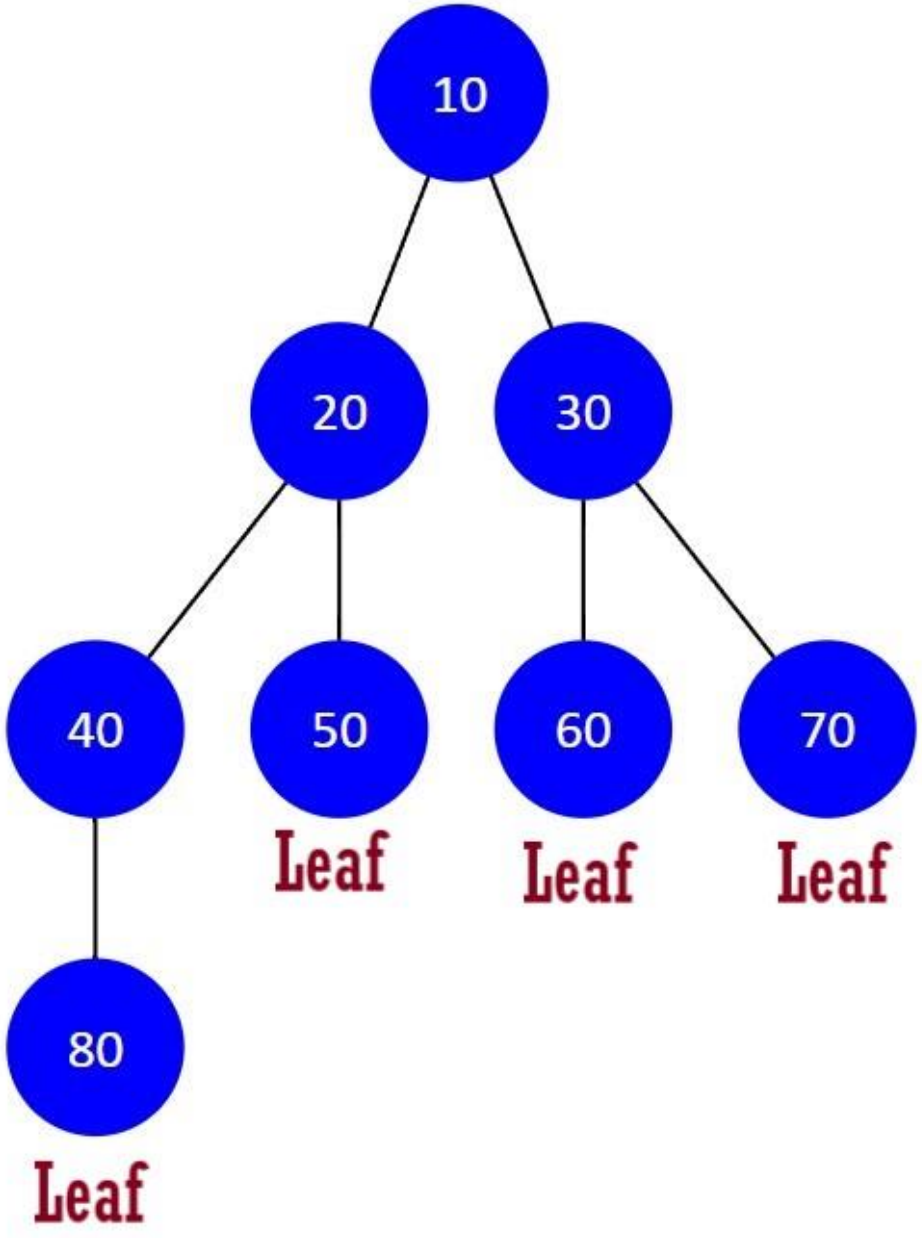
Root Node

- The topmost node of a tree or the node which does not have any parent node is called the root node.
- A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- In the given example
  - **10 is the root node of entire tree**



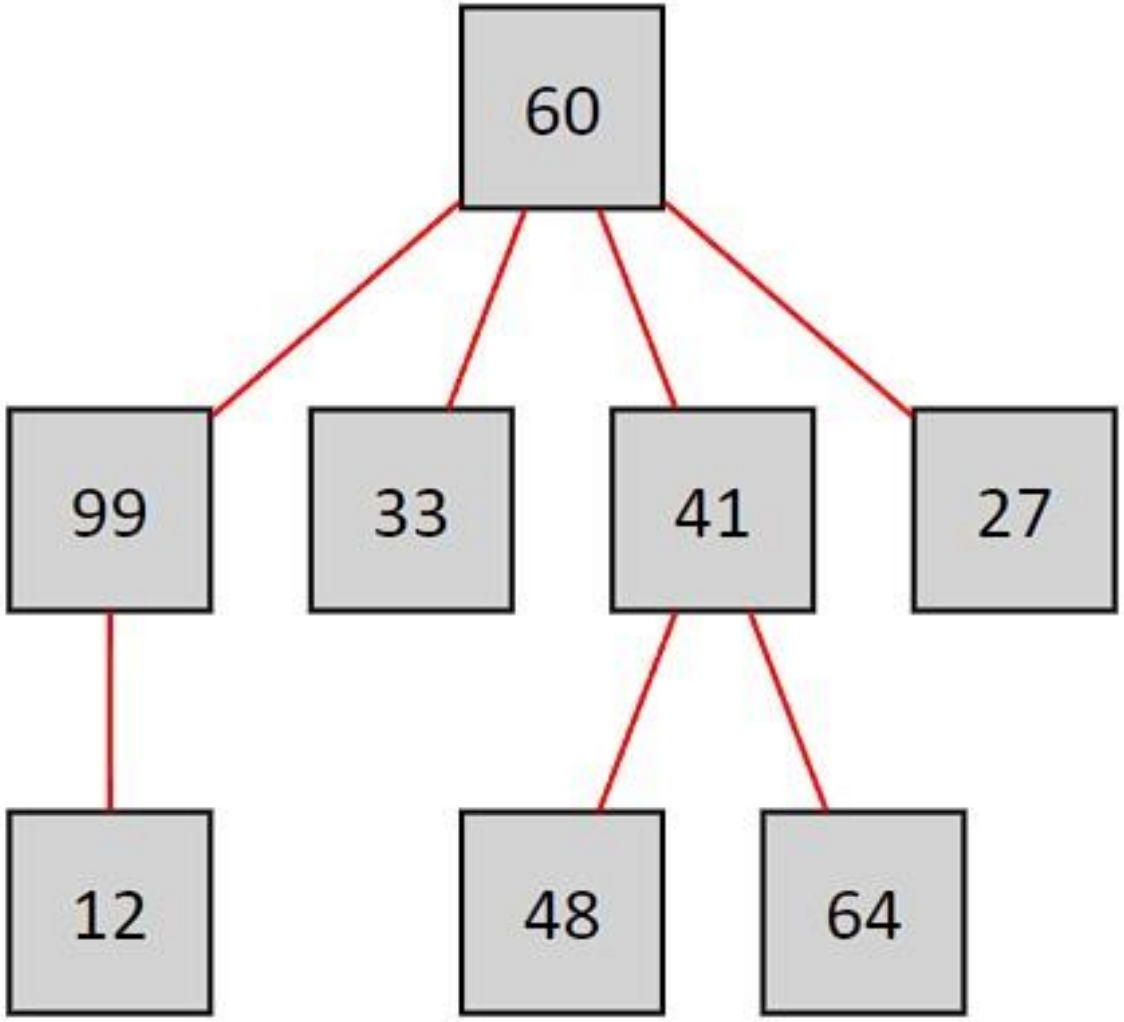
Leaf / External Node

- The nodes which do not have any child nodes are called leaf nodes.
- In the given example
  - {50, 60, 70, 80} are the leaf nodes



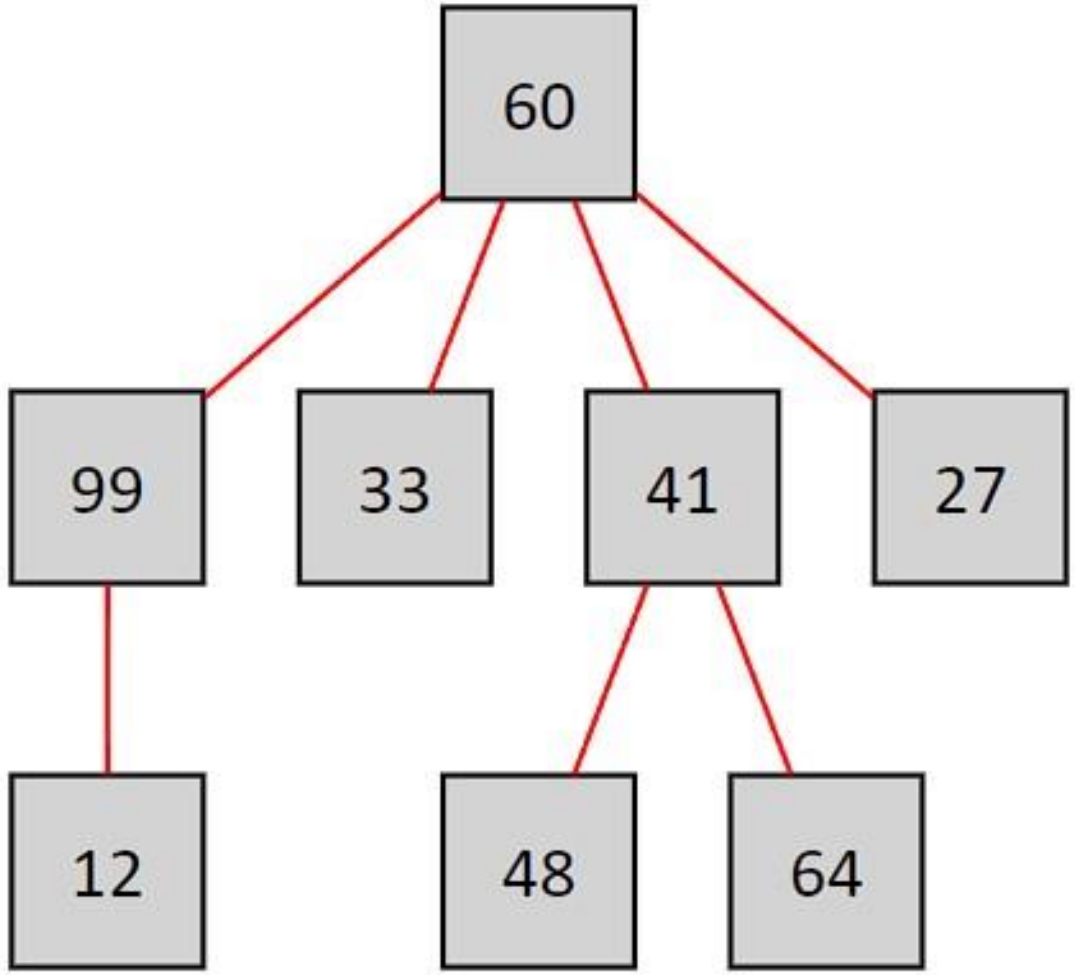
Ancestor of a Node

- Any predecessor nodes on the path of the root to that node are called Ancestors of that node.
- In the given example
  - **99 and 60 are the ancestors of 12**
  - **41 and 60 are the ancestors of 48 and 64**
  - **60 is the ancestor of {99, 33, 41, 27}**



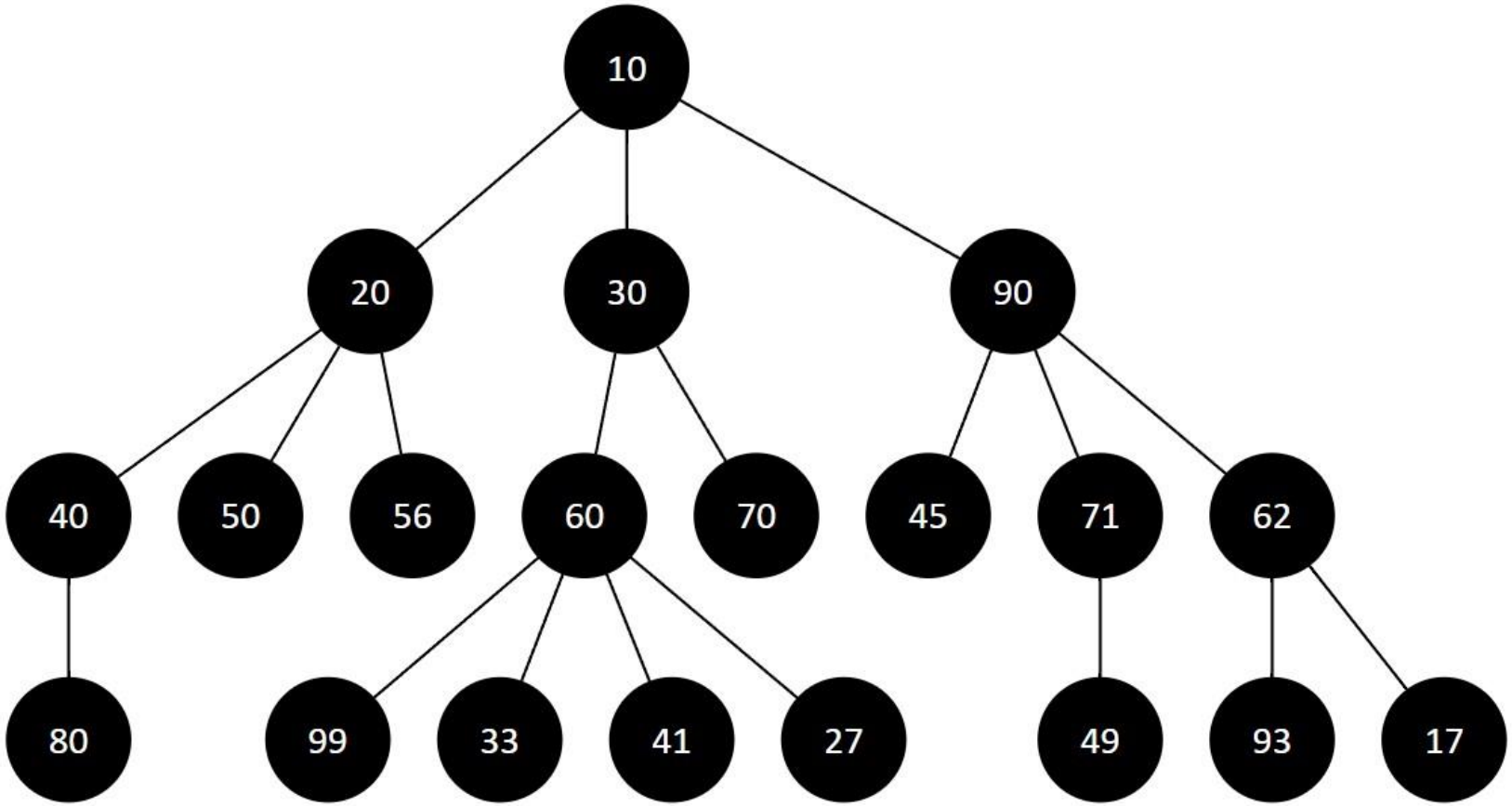
Descendants of a Node

- Any successor nodes on the path from the leaf node to that node.
- In the given example
  - Descendants of **60 are {99, 12, 33, 41, 48, 64, 27}**
  - Descendant of **99 is {12}**
  - Descendants of **41 are {48, 64}**



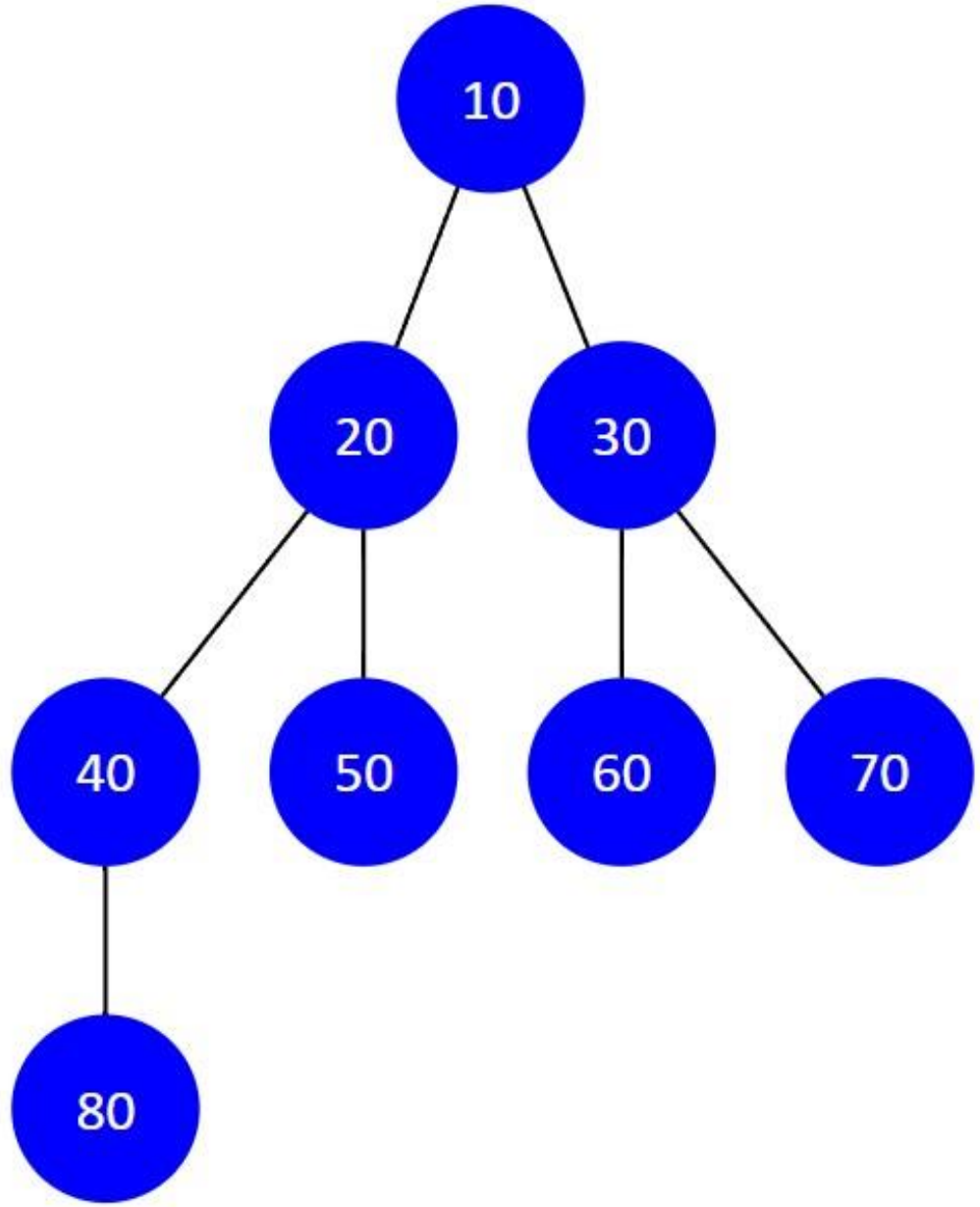
Siblings

- Children of the same parent node are called siblings.
- In the given example
  - Siblings of 30 are {20, 90}
  - Siblings of 56 {?}
  - Siblings of 71 {?}
  - Siblings of 93 {?}

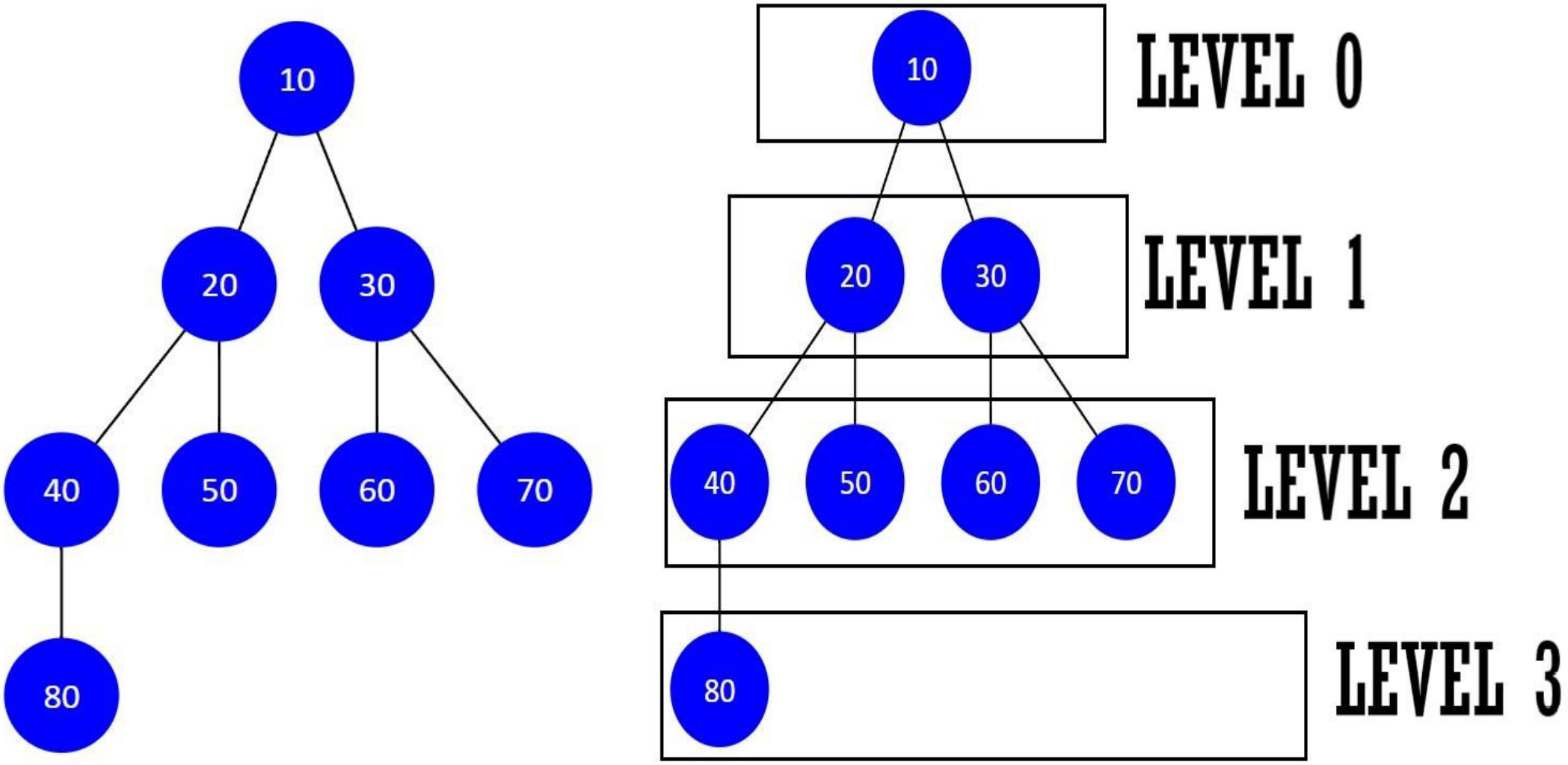


Level of a Node

- The count of edges on the path from the root node to that node. The root node has level 0.
- In the given example
  - Level of node 80 is 3, as there are 3 edges (lines) from root to that node
  - Level of 30 is 1
  - Level of 10 is 0
  - Level of 70 is 2

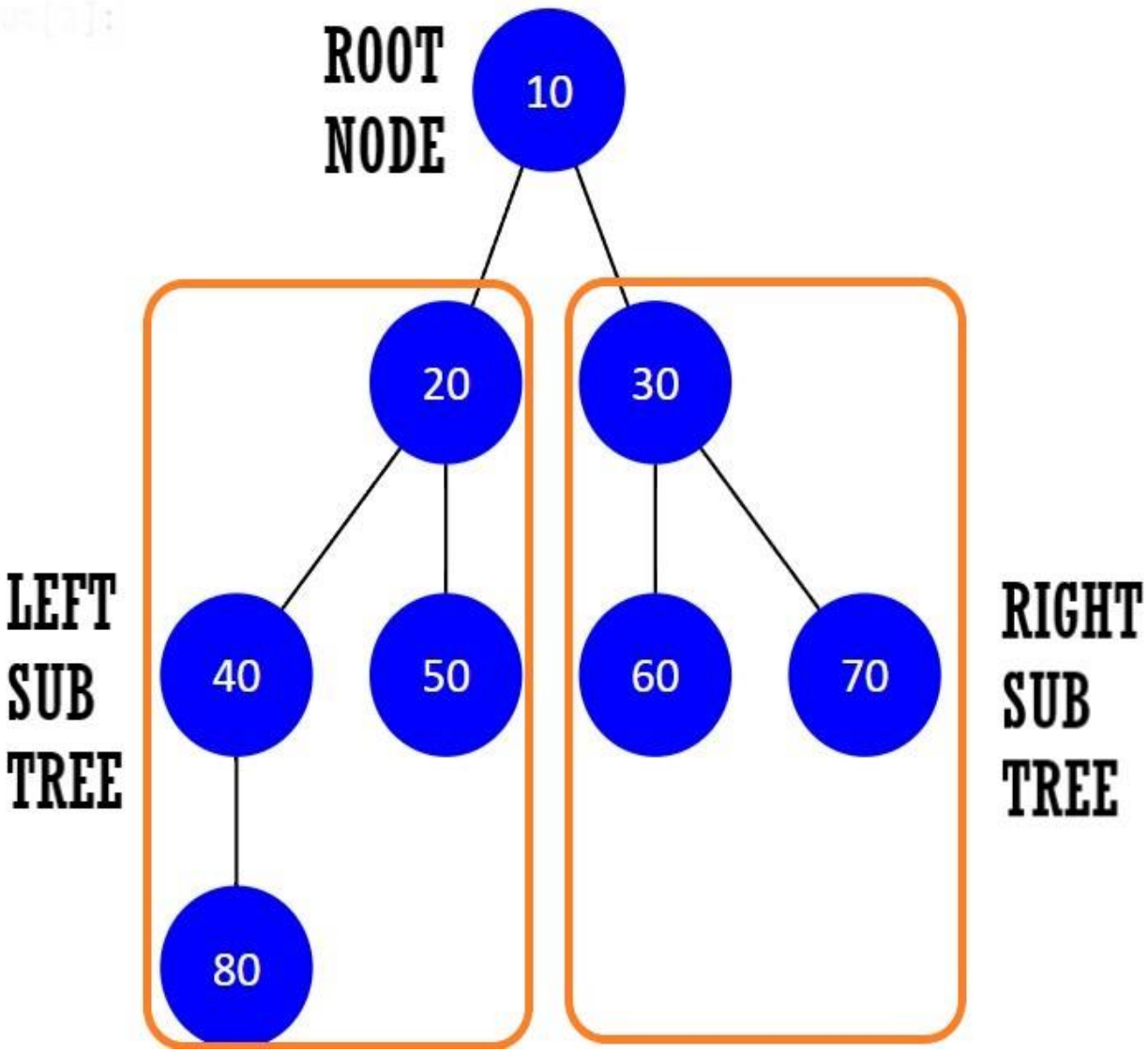


Levels of a Tree



Subtree

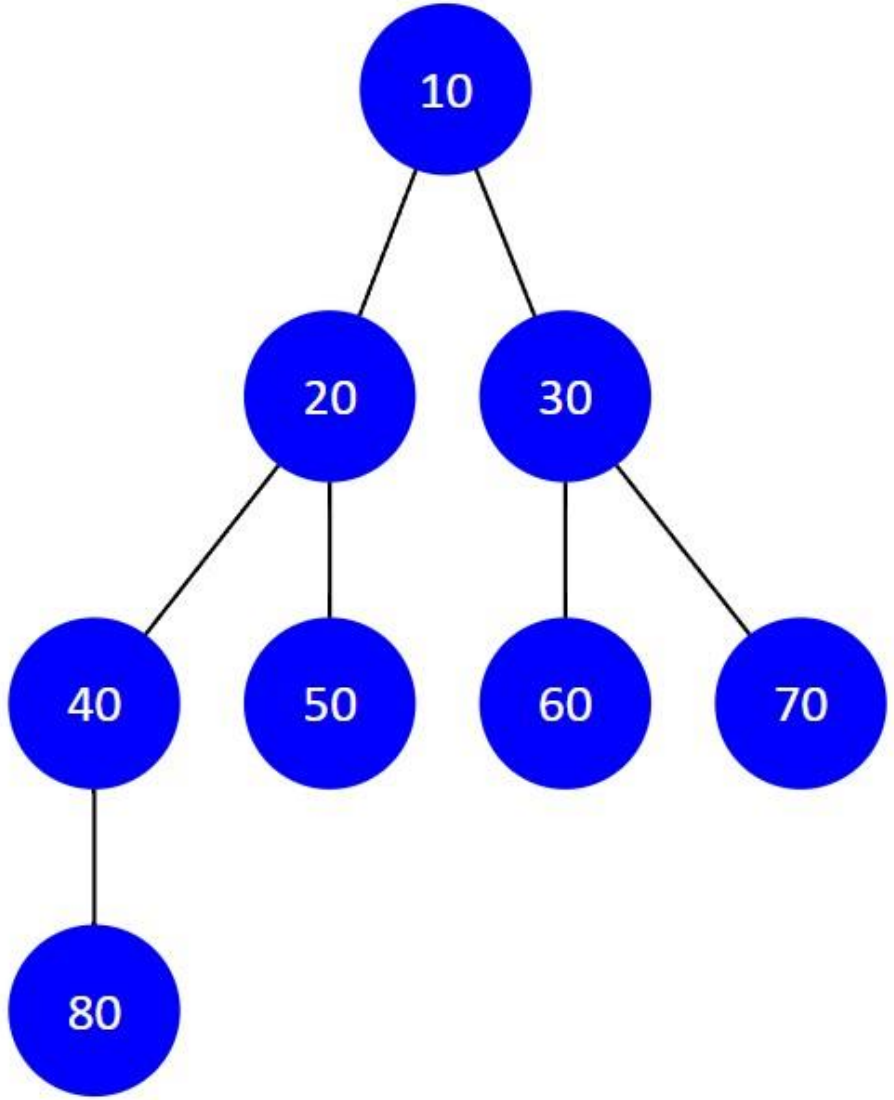
- Any node of the tree along with its descendants is called as a subtree.
- Trees are recursive structures in nature.
- That is, if you take any random node in a tree, the nodes it contain along with it can form a separate tree.



Properties of a Tree

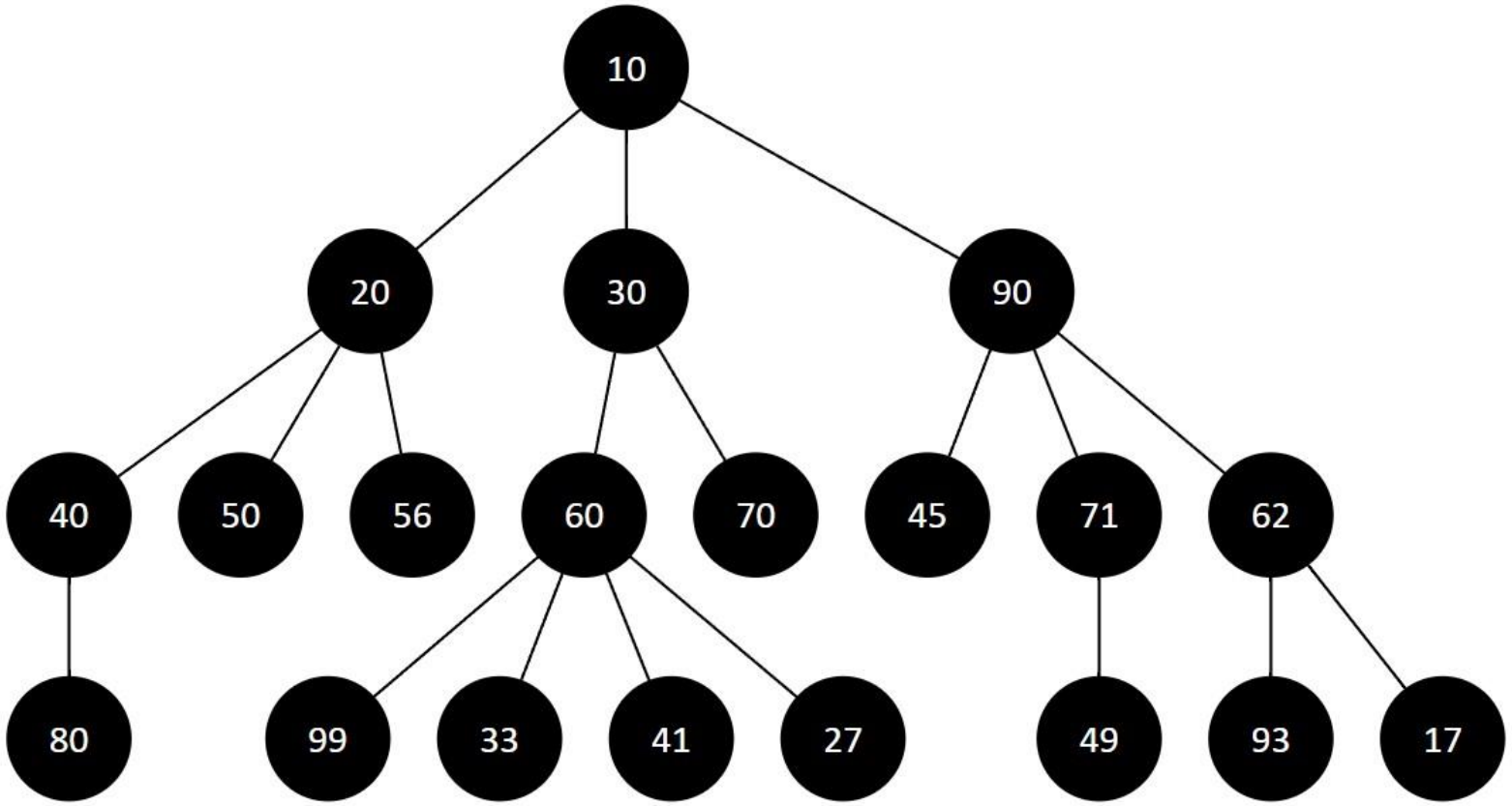
Number of Edges

- An edge can be defined as the connection between two nodes.
- If a tree has N nodes, then it will have (N-1) edges.



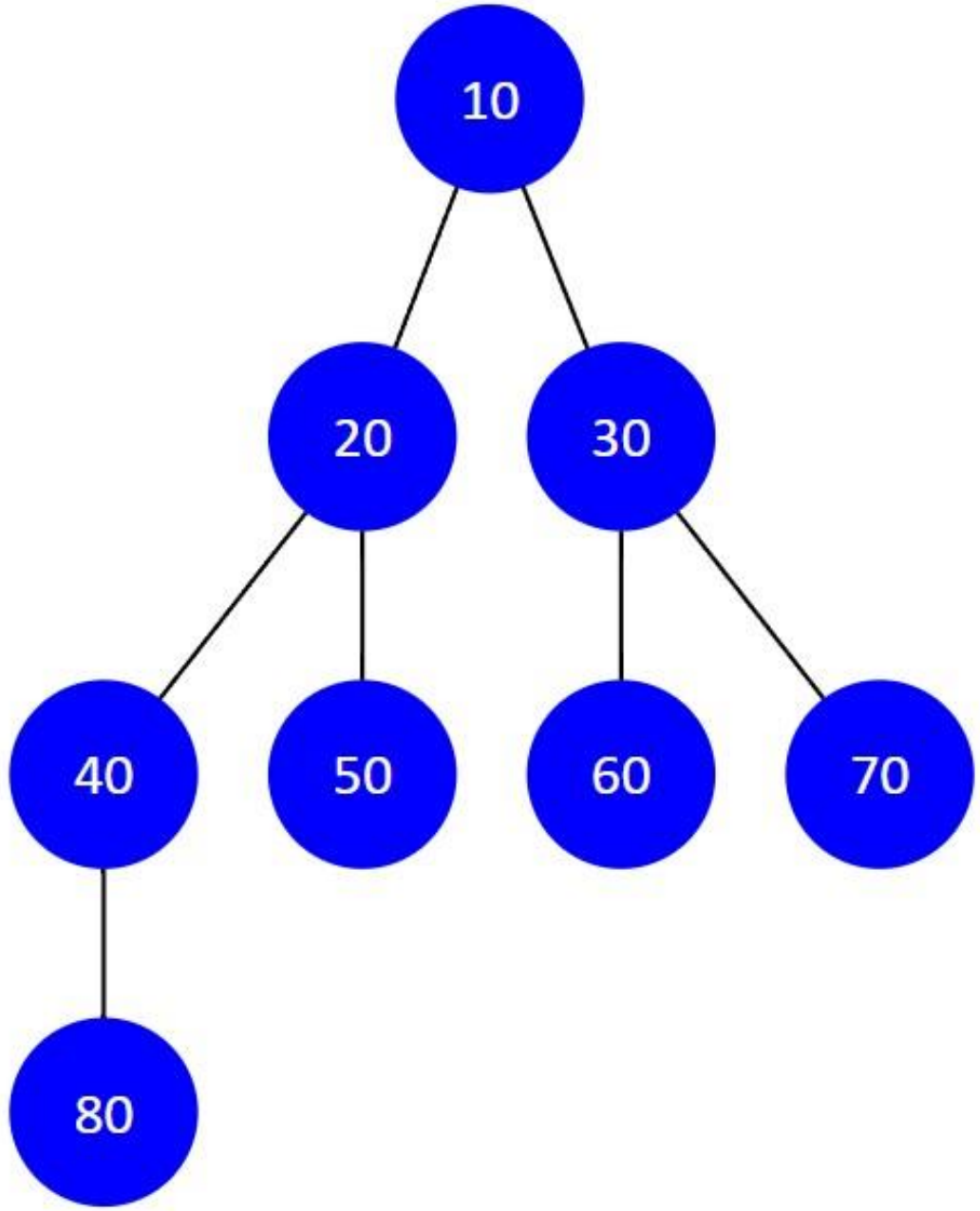
Number of Edges

- An edge can be defined as the connection between two nodes.
- If a tree has N nodes, then it will have (N-1) edges.



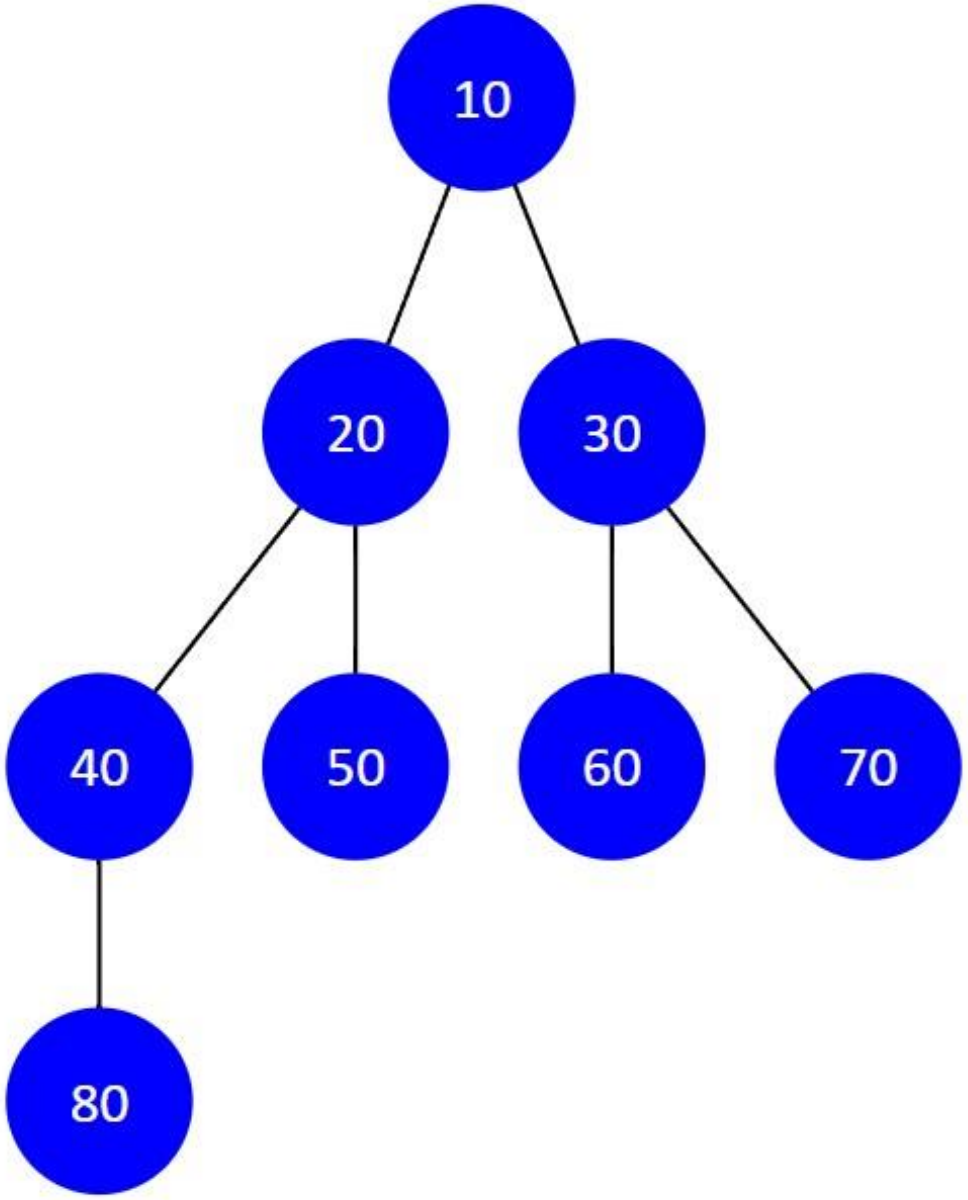
Depth of a Node

- The depth of a node is defined as the length of the path from the root to that node.
- Each edge adds 1 unit of length to the path.
- So, it can also be defined as the number of edges in the path from the root of the tree to the node.



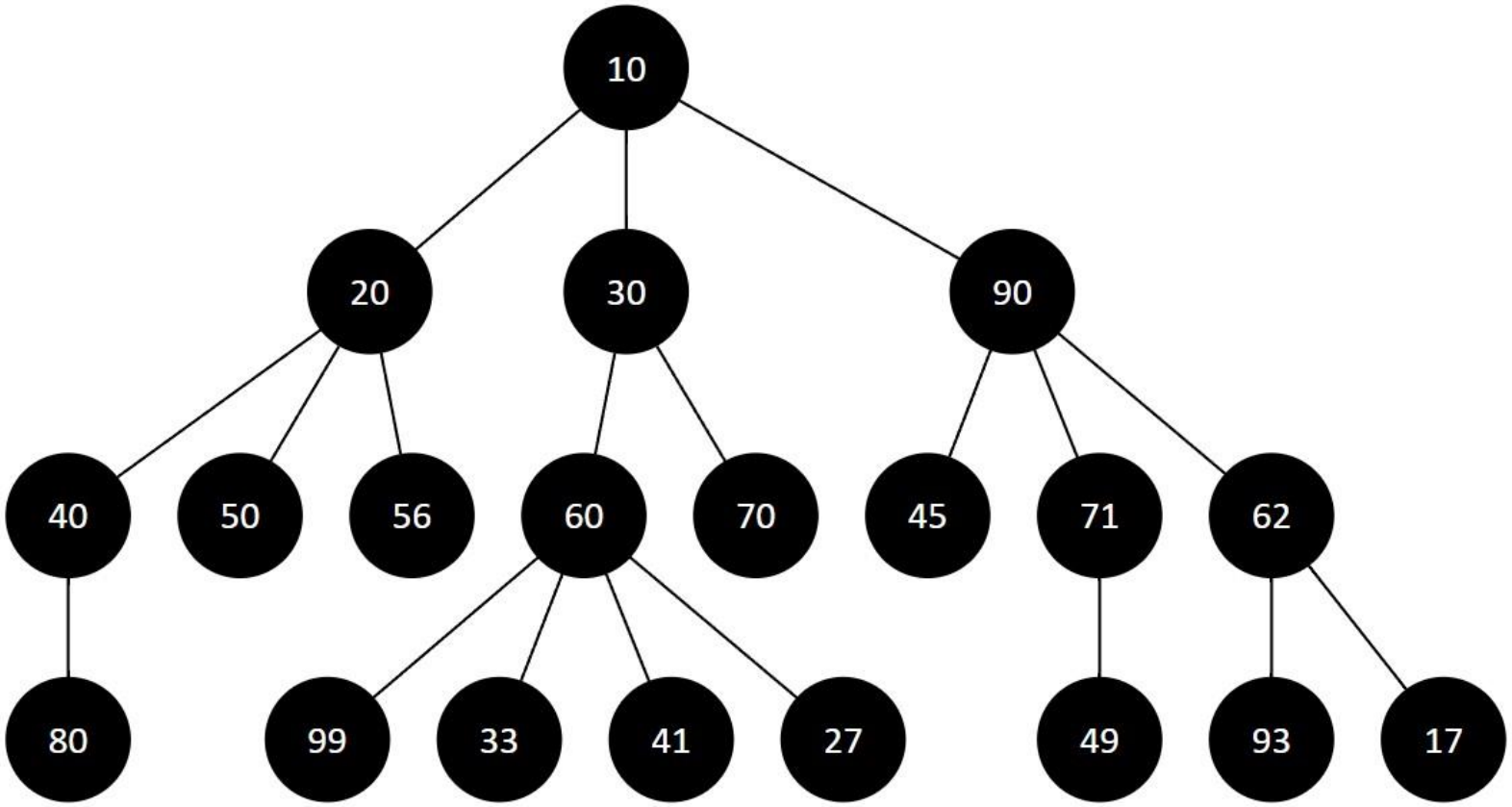
Height of a Node

- The height of a node in a tree is the number of edges on the longest downward path between that node and a leaf.
- In simpler terms, the height of a node is the length of the longest path from that node to a leaf node.
- Height of Node 20 is 2.
- Height of Node 10 is 3.
- Height of Node 30 is 1.



Degree of a Node

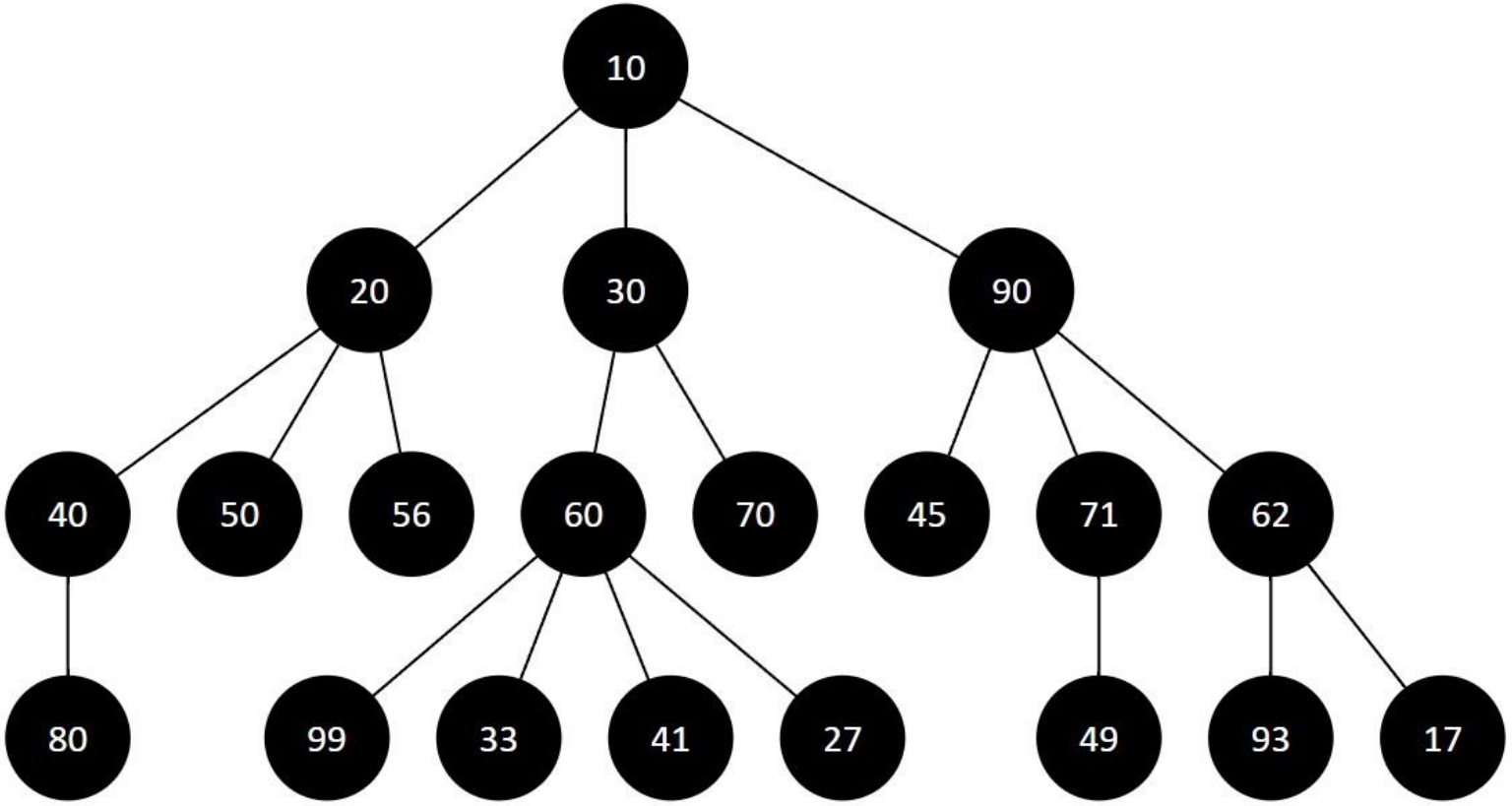
- The degree of a node can be defined as number of children the node is having.
- Leaf nodes' degree will be 0.
- Degree of 10: 3
- Degree of 30: 2
- Degree of 60: 4
- Degree of 50: 0



Types of Trees

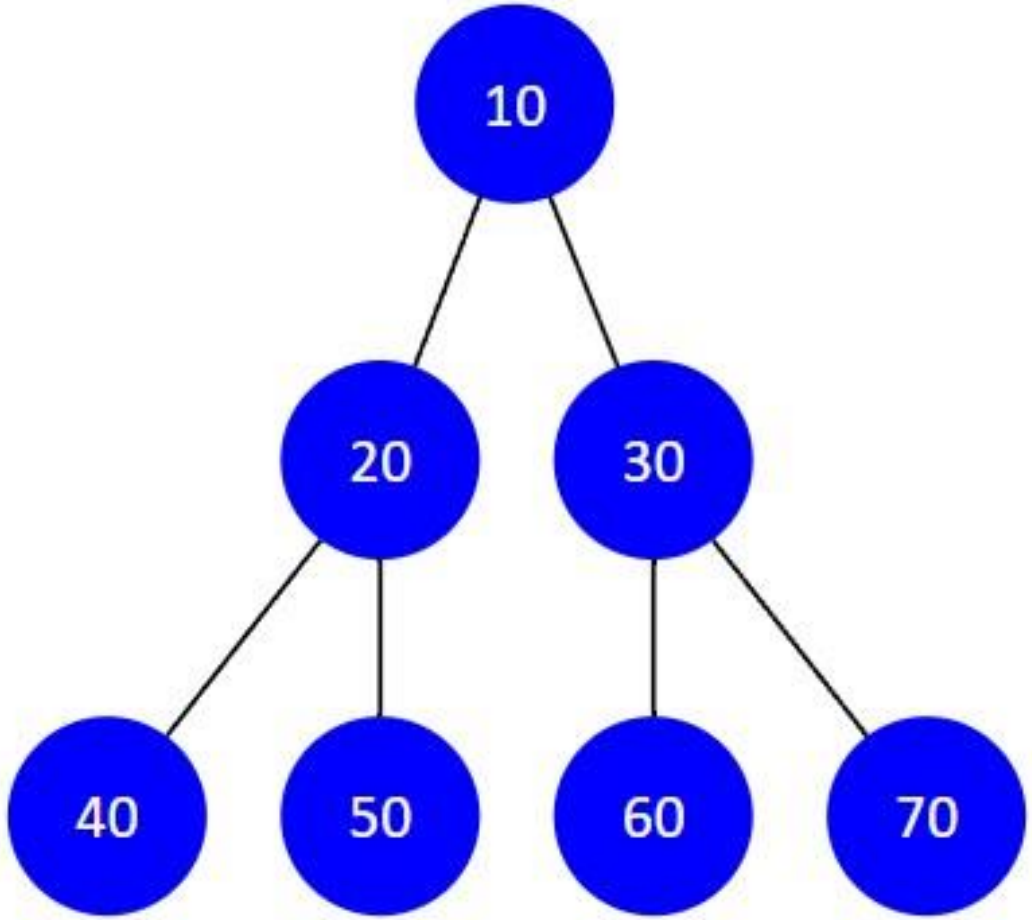
Types of Trees – General (N-ary) Tree

- General tree (N-ary) has no restriction on the number of nodes.
- It means that a parent node can have any number of child nodes.



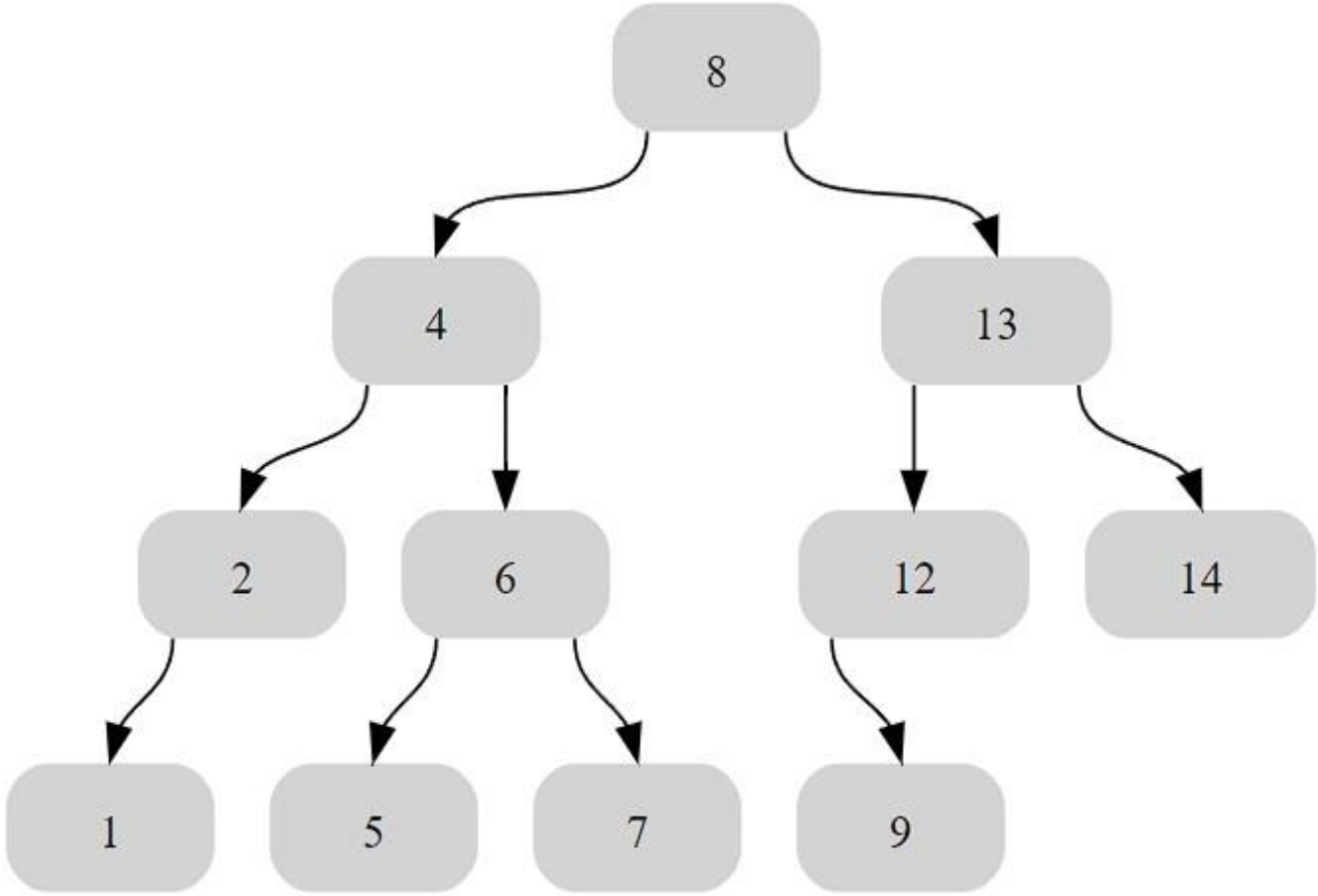
Types of Trees – Binary Tree

- A node of a binary tree can have a maximum of two child nodes.
- Binary Tree is a Tree in which, each node will contain at most 2 nodes.
- As there will be at most 2 children for any given parent, we will call them as
  - Left Child
  - Right Child



Types of Trees – Binary Search Tree (BST)

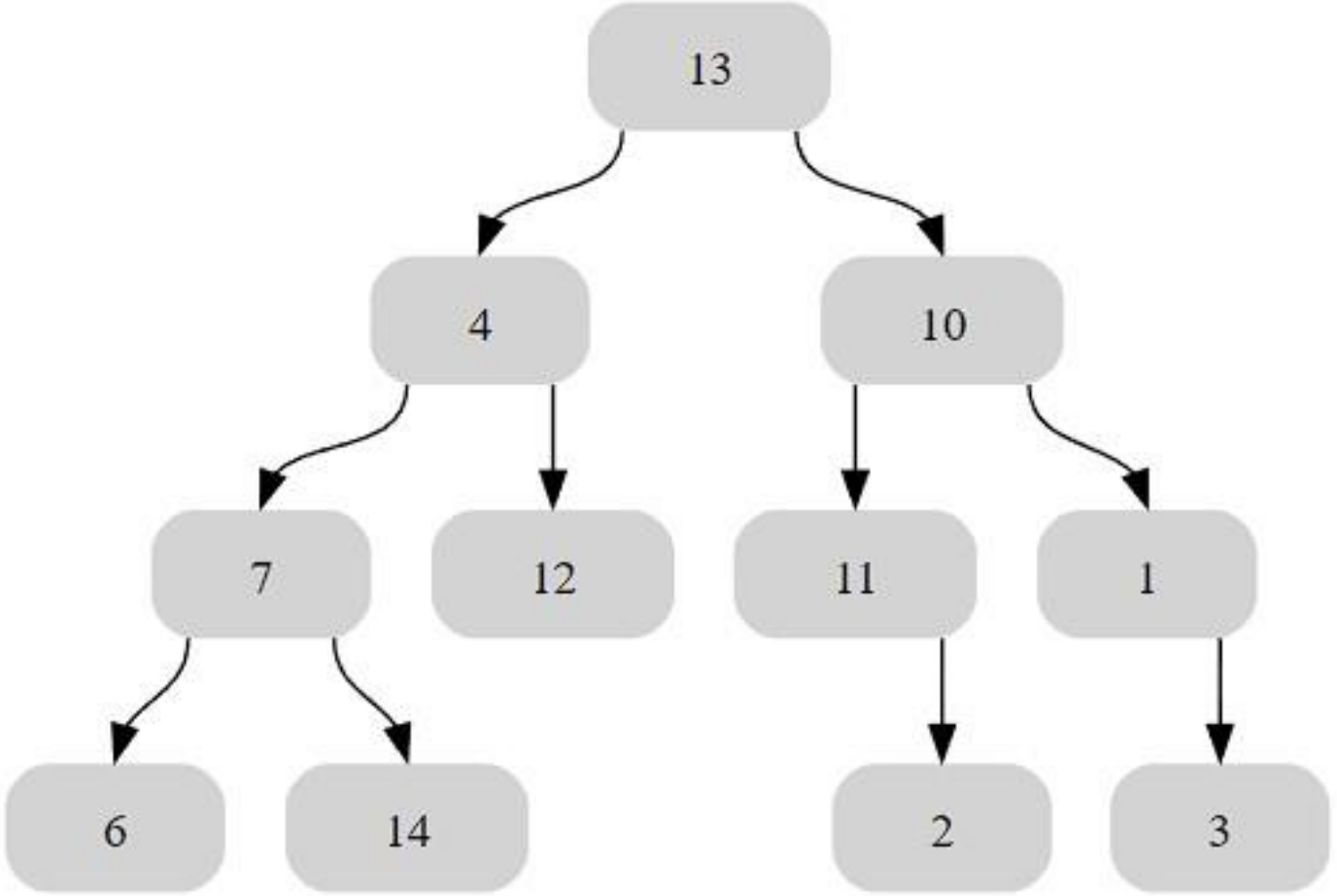
- In a BST, every node's left part contains the nodes with value less than that of the node
- And right part contains the nodes with value greater than that of the node
- In other words
  - In a BST, if you consider a node N with value V, its left part contains nodes with value less than V
  - Right part contain nodes with value greater than V.



Binary Trees

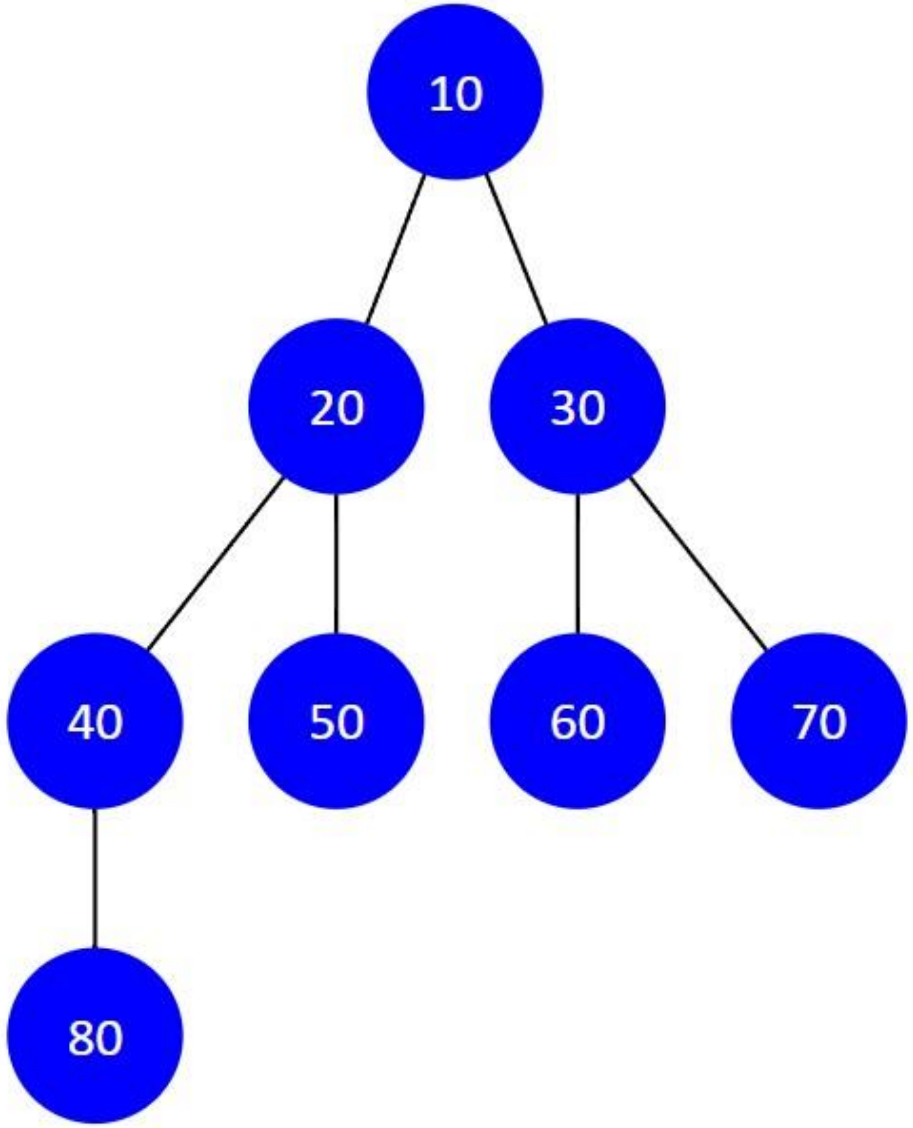
Binary Tree - Definition

- A tree whose nodes have at most 2 children is called a binary tree.
- Since each parent in a binary tree can have only 2 children, we typically name them as
  - left child
  - right child.
- A Tree node contain the following parts
  - Data
  - Pointer to the left child
  - Pointer to the right child



Binary Tree - Example

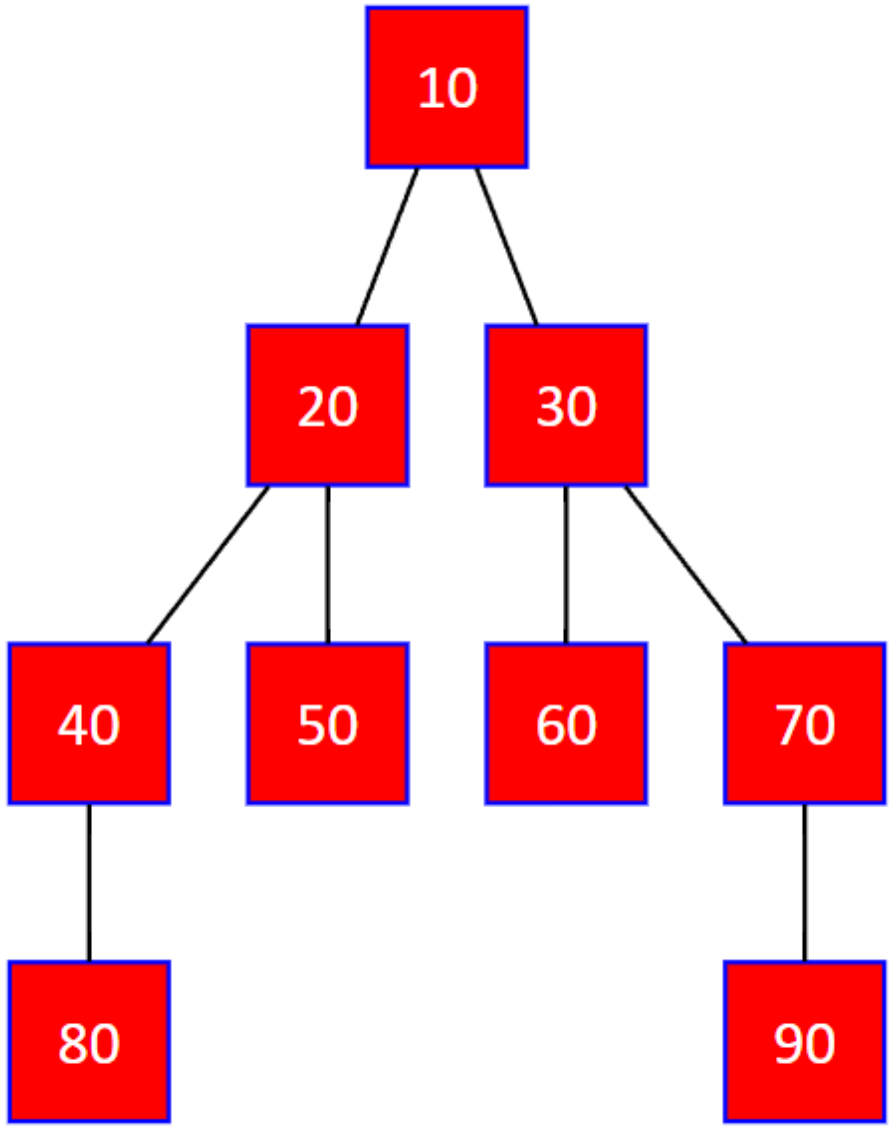
- Consider the tree on right side
- In it every parent has at most 2 children
  - node 10 has 2 children {20, 30}
  - node 20 has 2 children {40, 50}
  - node 30 has 2 children {60, 70}
  - node 40 has 1 child {80}
  - nodes 50, 60 70 and 80 have 0 children, so they are leaf nodes



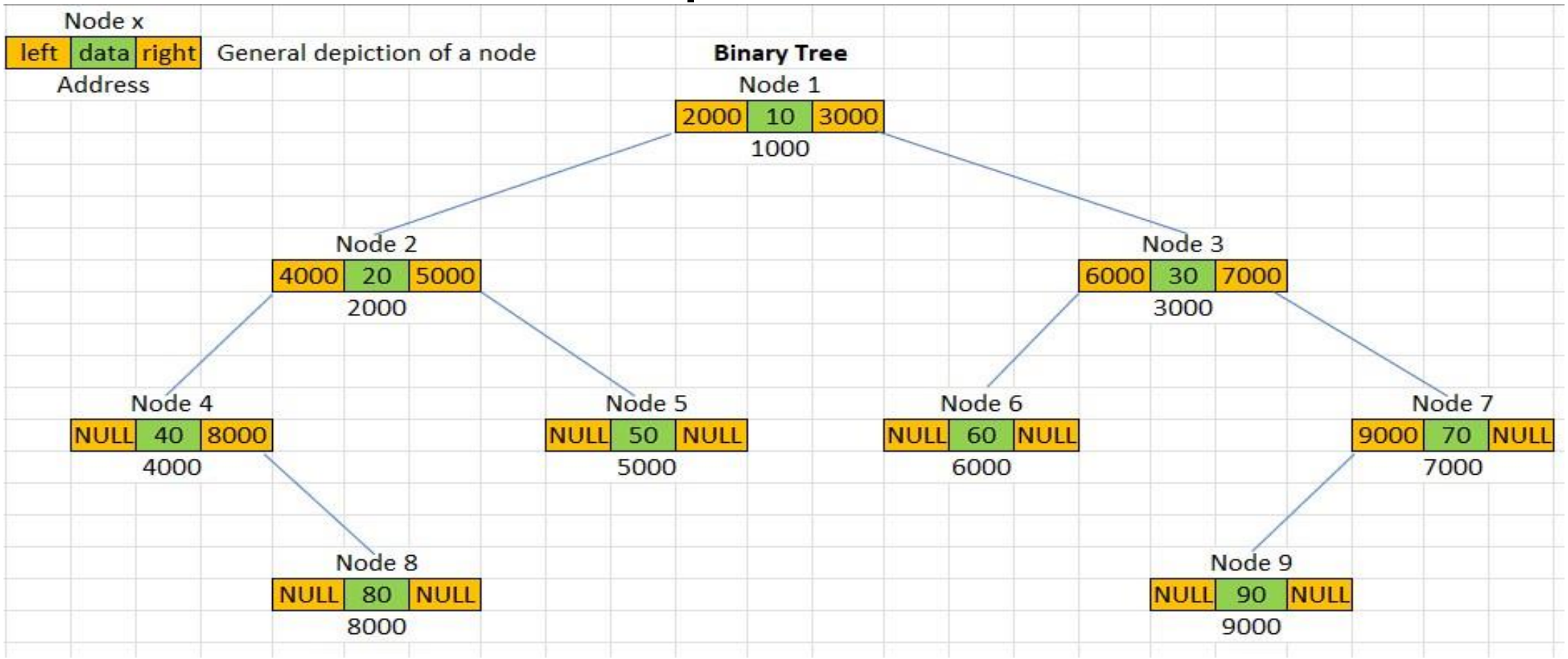
Memory Representation of Binary Tree

–

Consider this binary tree



This is how it will be represented



Binary Tree Traversals

Traversing a way to visit all the nodes (vertices) in a tree. It is of 2 types

Depth First Traversal (DFT)

- Pre-Order Traversal
- In-Order Traversal
- Post-Order Traversal

Breadth First Traversal (BFT)

- Level-Order Traversal

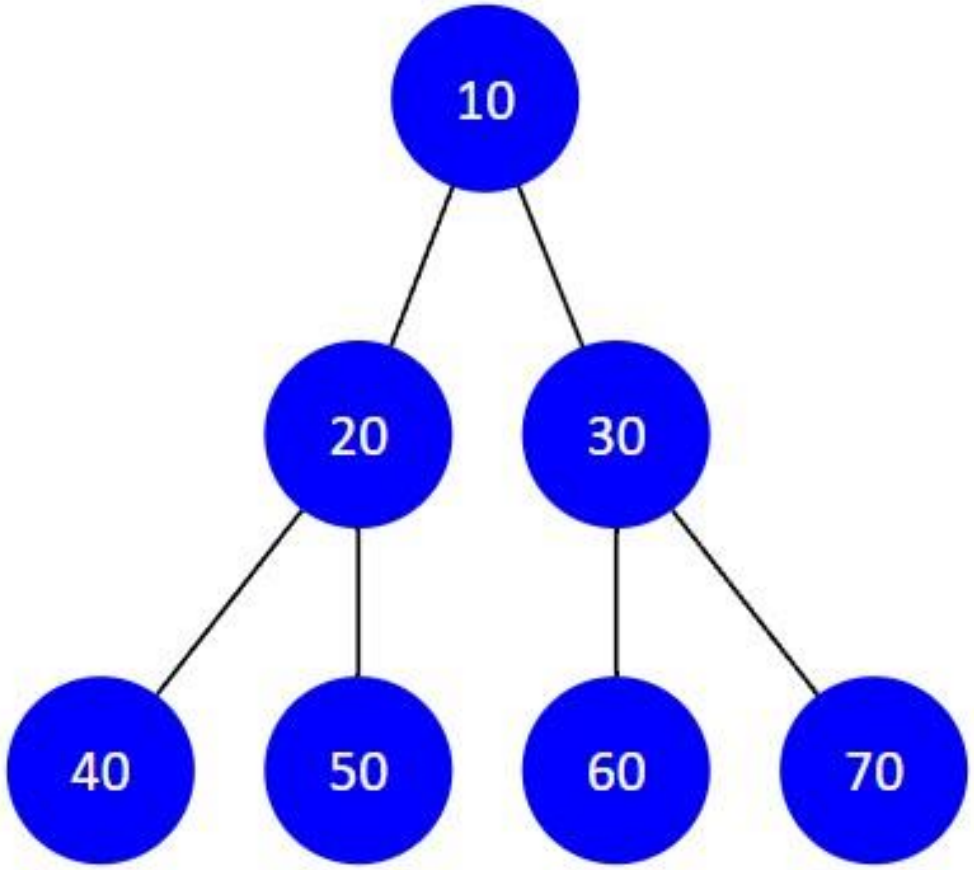
Binary Tree Traversals - DFT

Pre-Order Traversal	In-Order Traversal	Post-Order Traversal
• Root Node • Left Child • Right Child	• Left Child • Root Node • Right Child	• Left Child • Right Child • Root Node

Binary Tree Traversals: Pre-Order Traversal

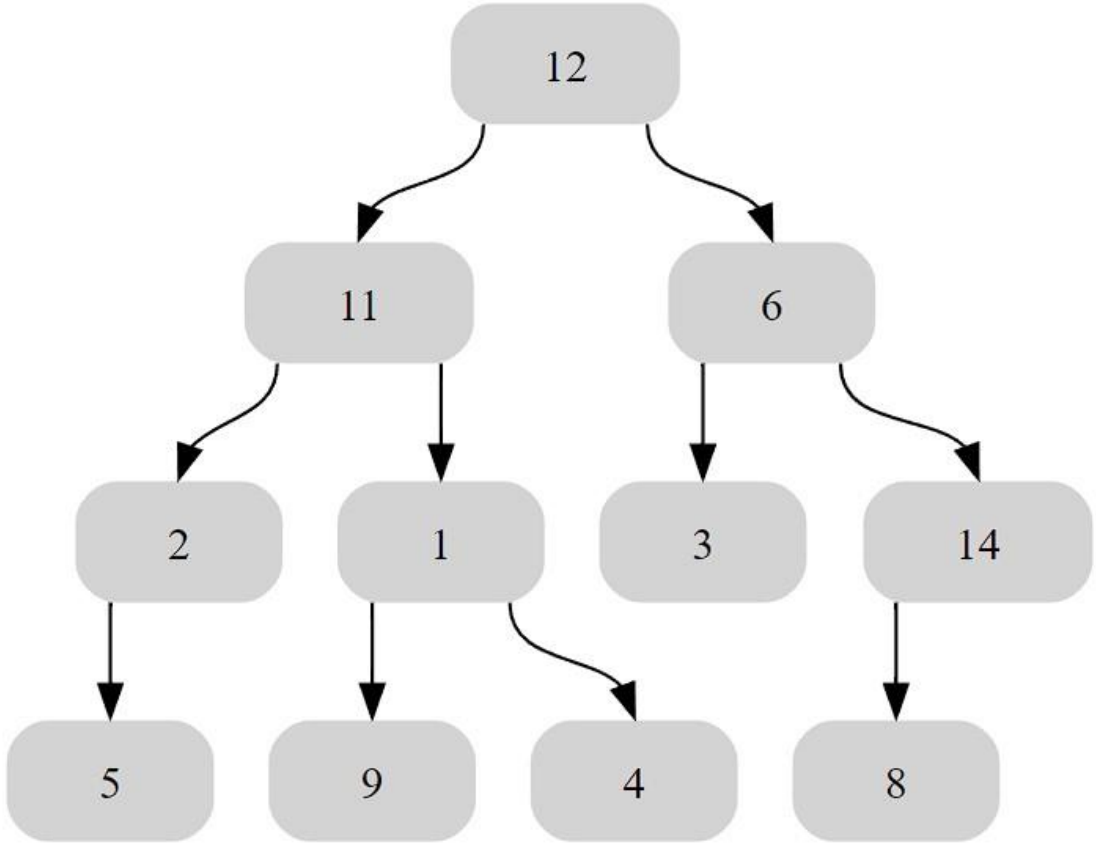
- In Pre-Order Traversal we give preference to
  - Root Node
  - Left Child
  - Right Child
- Pre-Order Traversal of given Binary Tree is

10 20 40 50 30 60 70



Binary Tree Traversals: Pre-Order Traversal

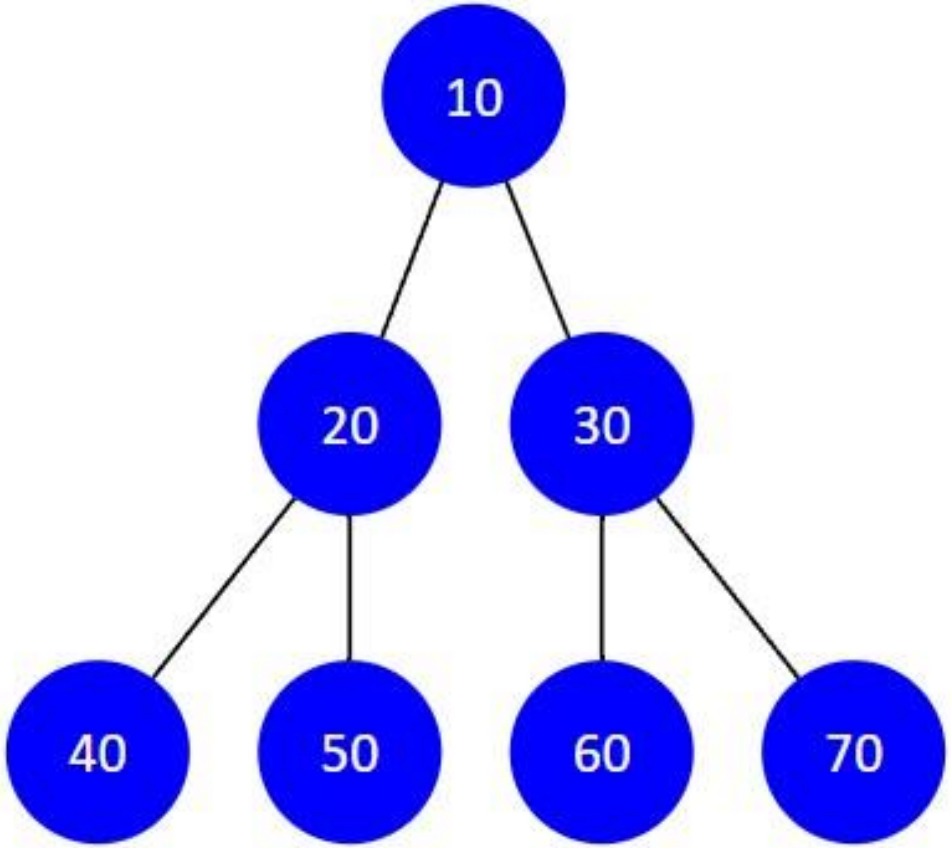
- Root – Left – Right
- Pre-Order Traversal of given Binary Tree is??
- **12 11 2 5 1 9 4 6 3 14 8**



Binary Tree Traversals: In-Order Traversal

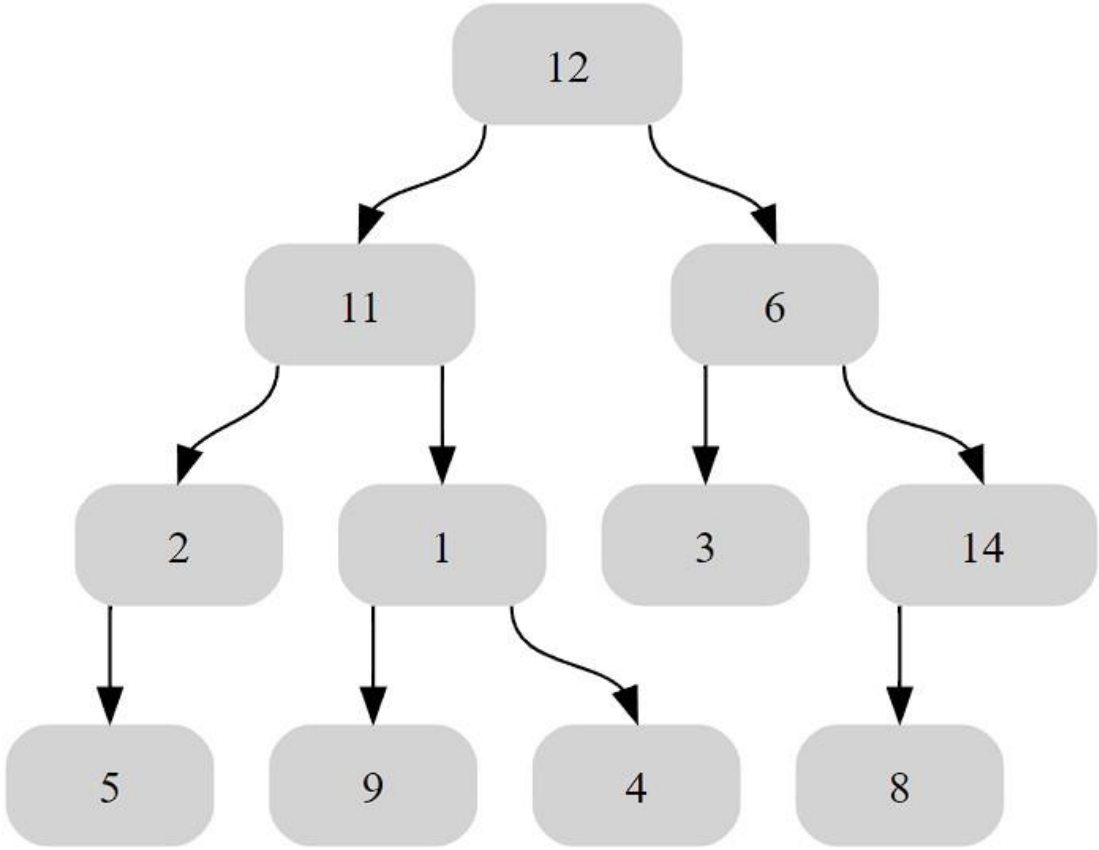
- In In-Order Traversal we give preference to
  - Left Child
  - Root Node
  - Right Child
- In-Order Traversal of given Binary Tree is

  40 20 50 10 60 30 70



Binary Tree Traversals: In-Order Traversal

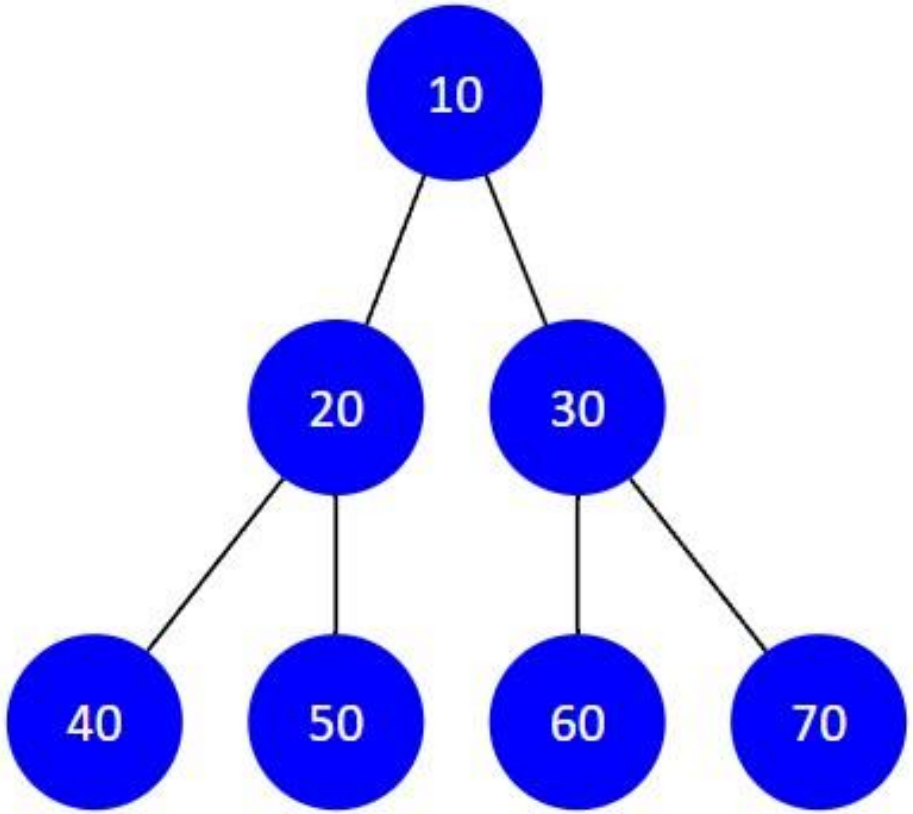
- Left – Root – Right
- In-Order Traversal of given Binary Tree is??
- **5 2 11 9 1 4 12 3 6 8 14**



Binary Tree Traversals: Post-Order Traversal

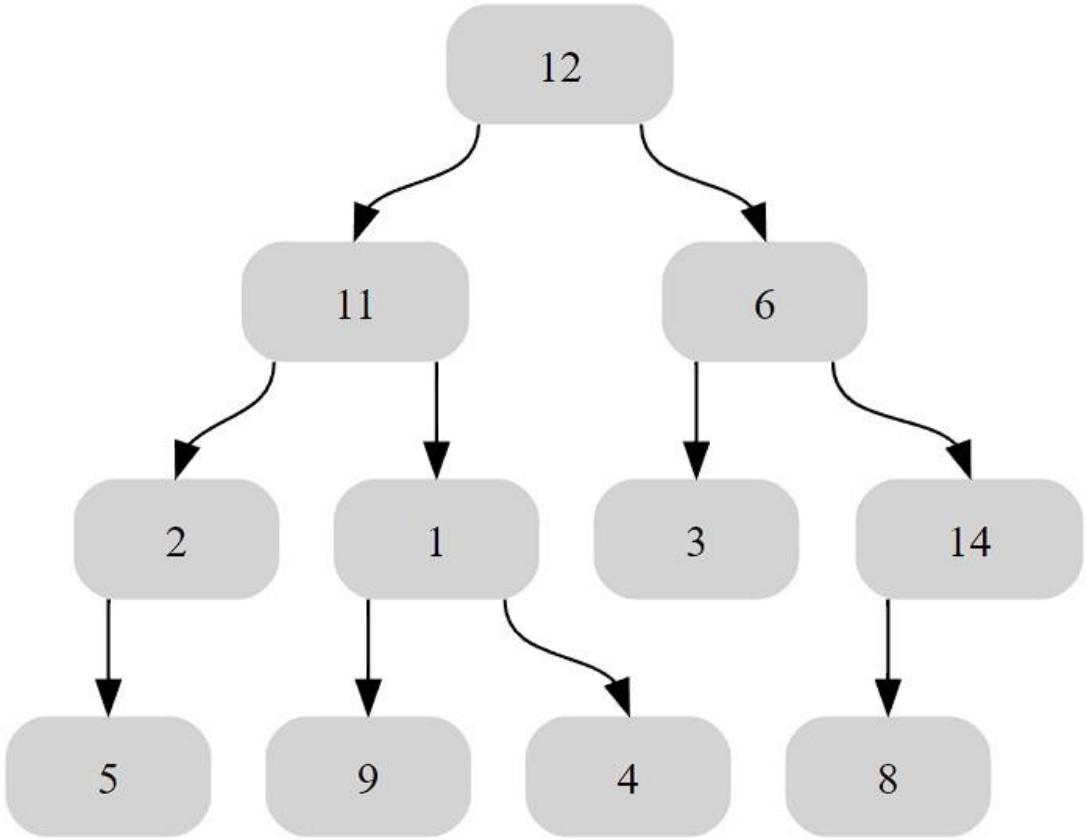
- In Post-Order Traversal we give preference to
  - Left Child
  - Right Child
  - Root Node
- Post-Order Traversal of given Binary Tree is

40 50 20 60 70 30 10



Binary Tree Traversals: Post-Order Traversal

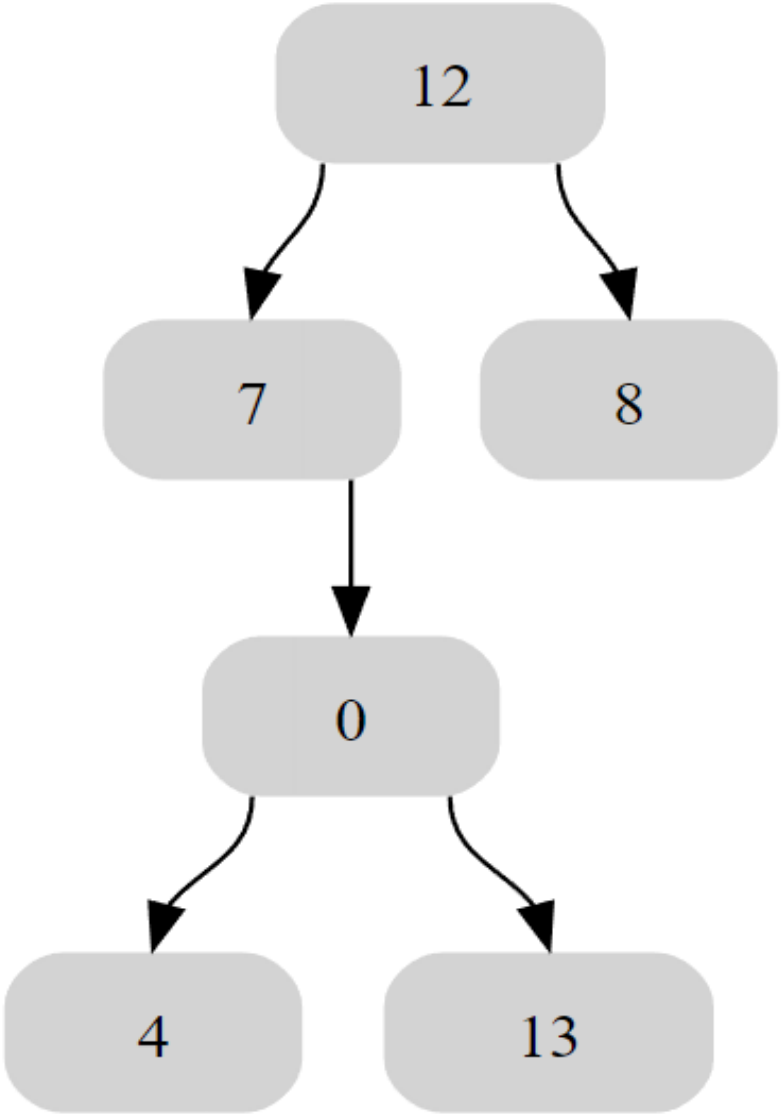
- Left – Right – Root
- Post-Order Traversal of given Binary Tree is??
- **5 2 9 4 1 11 3 8 14 6 12**



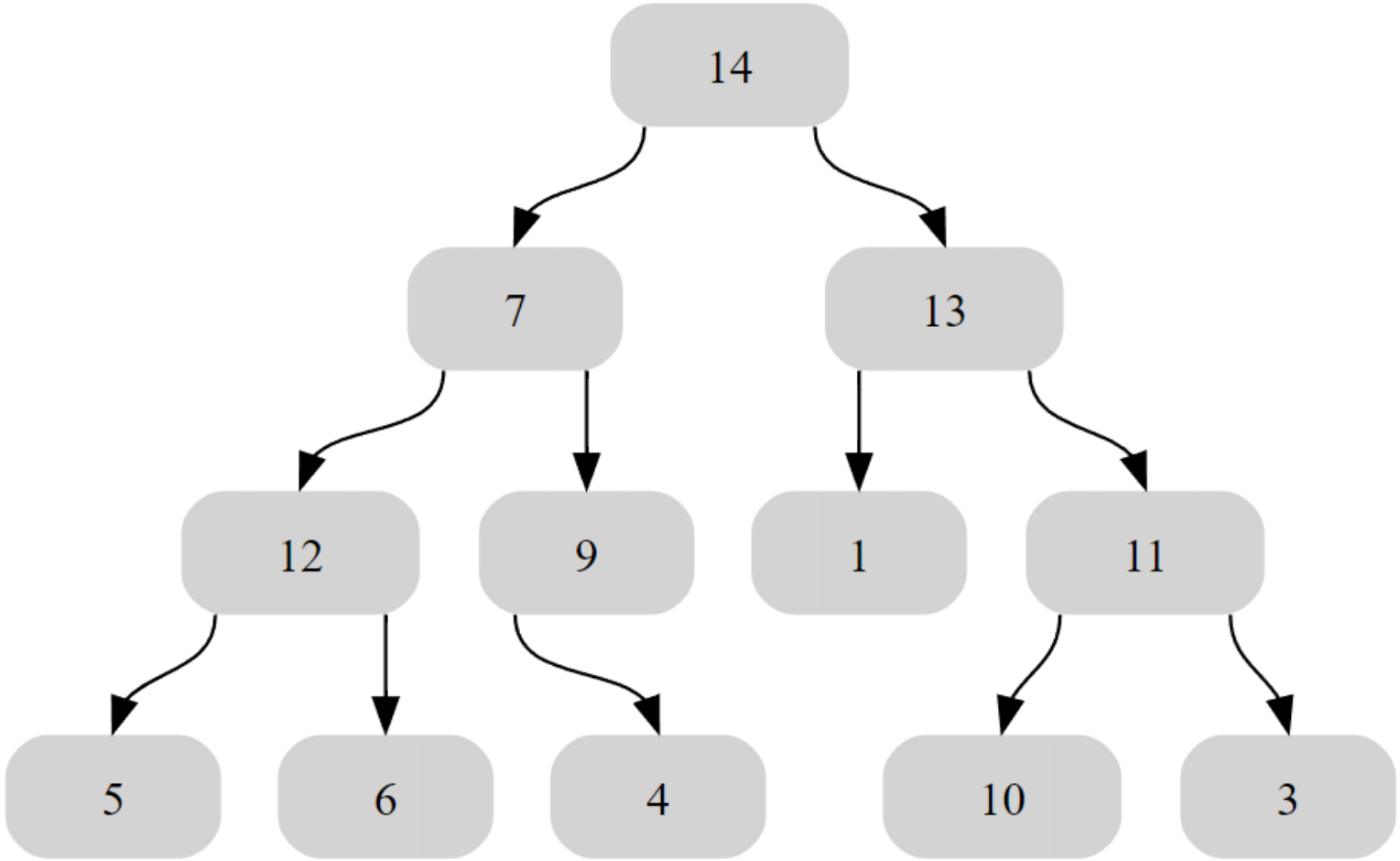
Exercise

- Write the
  - Pre-Order
  - In-Order
  - Post-Order traversals of given binary trees

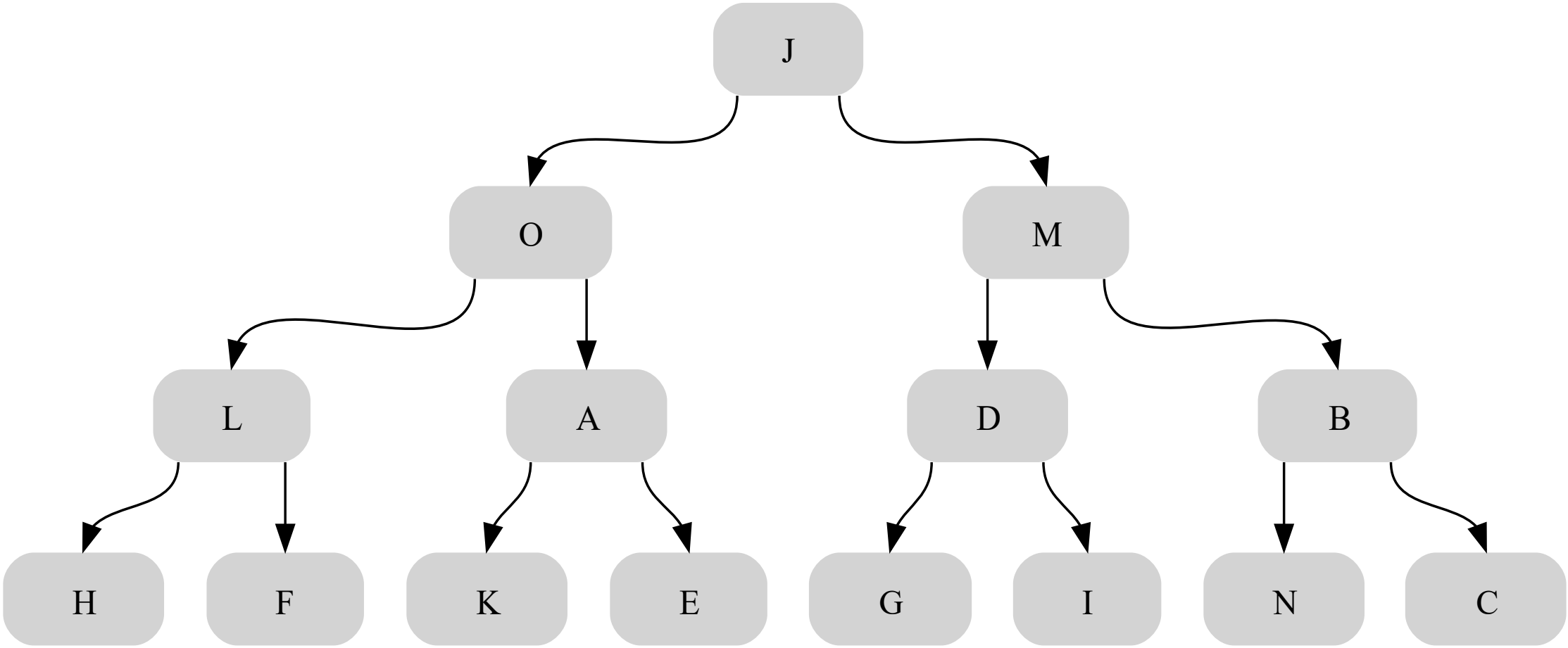
Binary Tree 1



Binary Tree 2



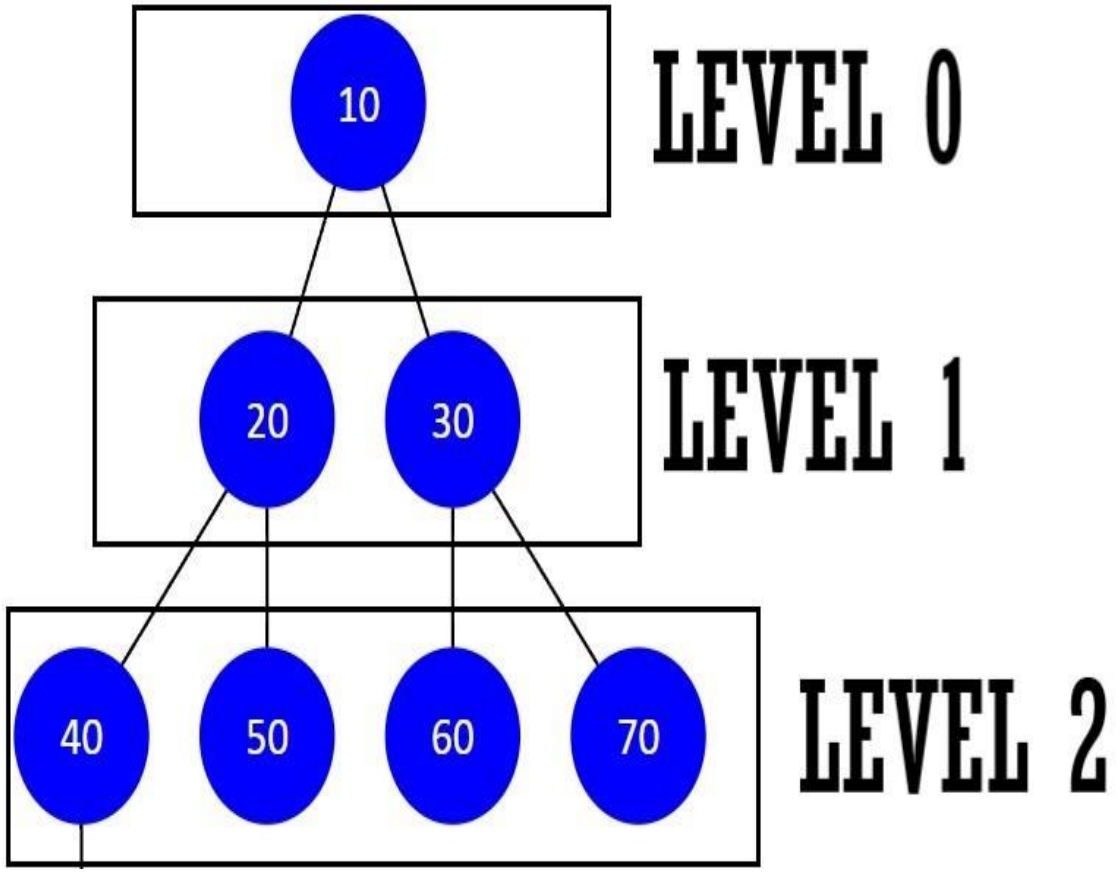
Binary Tree 3



Binary Tree Traversals

(Breadth First Traversal): Level Order Traversal

- In Level-Order Traversal, we traverse through the tree level by level.
- We will process all the nodes in one level before moving on to another level.
- A Queue data structure can be used for Level-Order Traversal.
- Level Order Traversal of given tree is
  - **10 20 30 40 50 60 70**



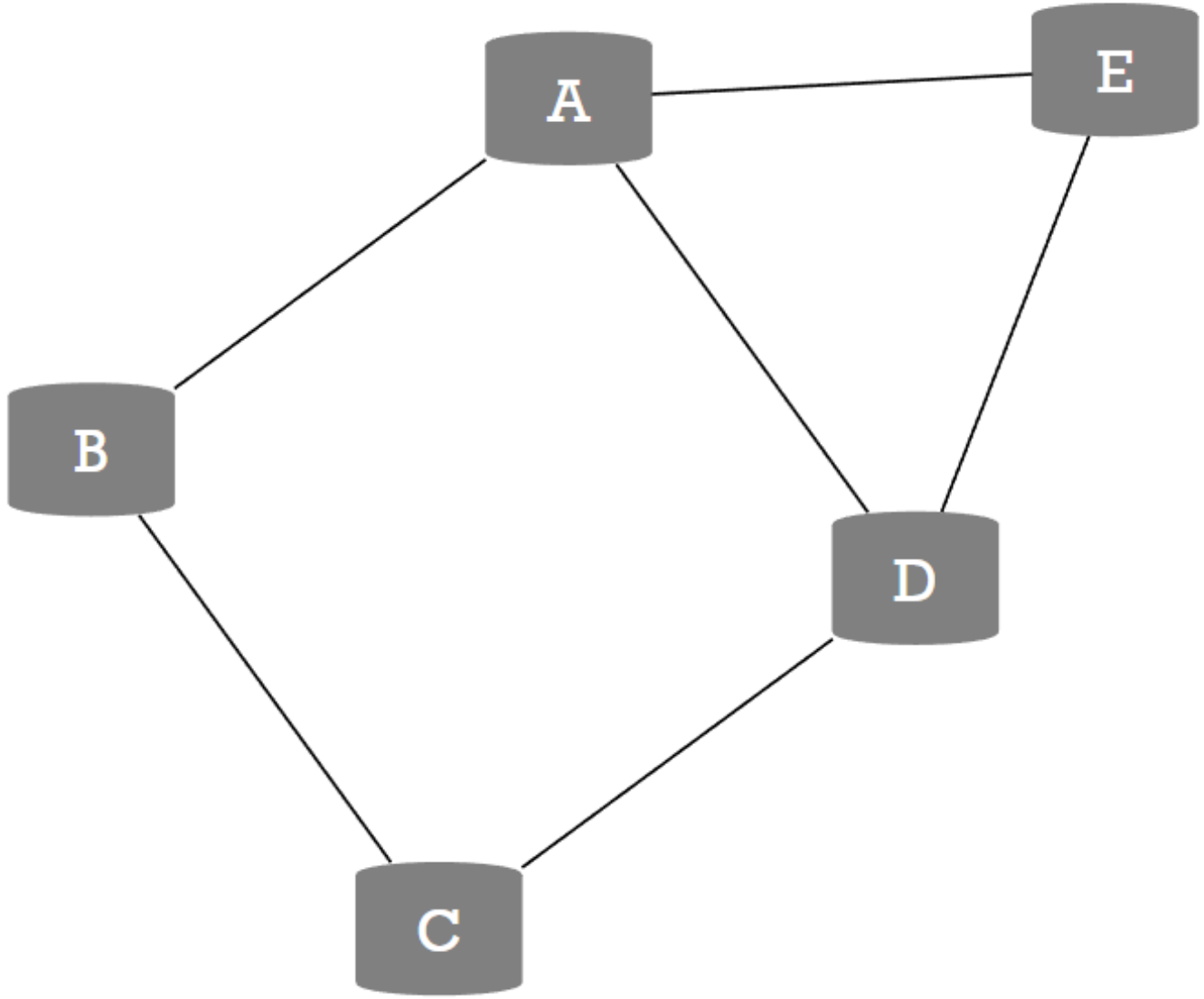
Graphs

Graph – Definition

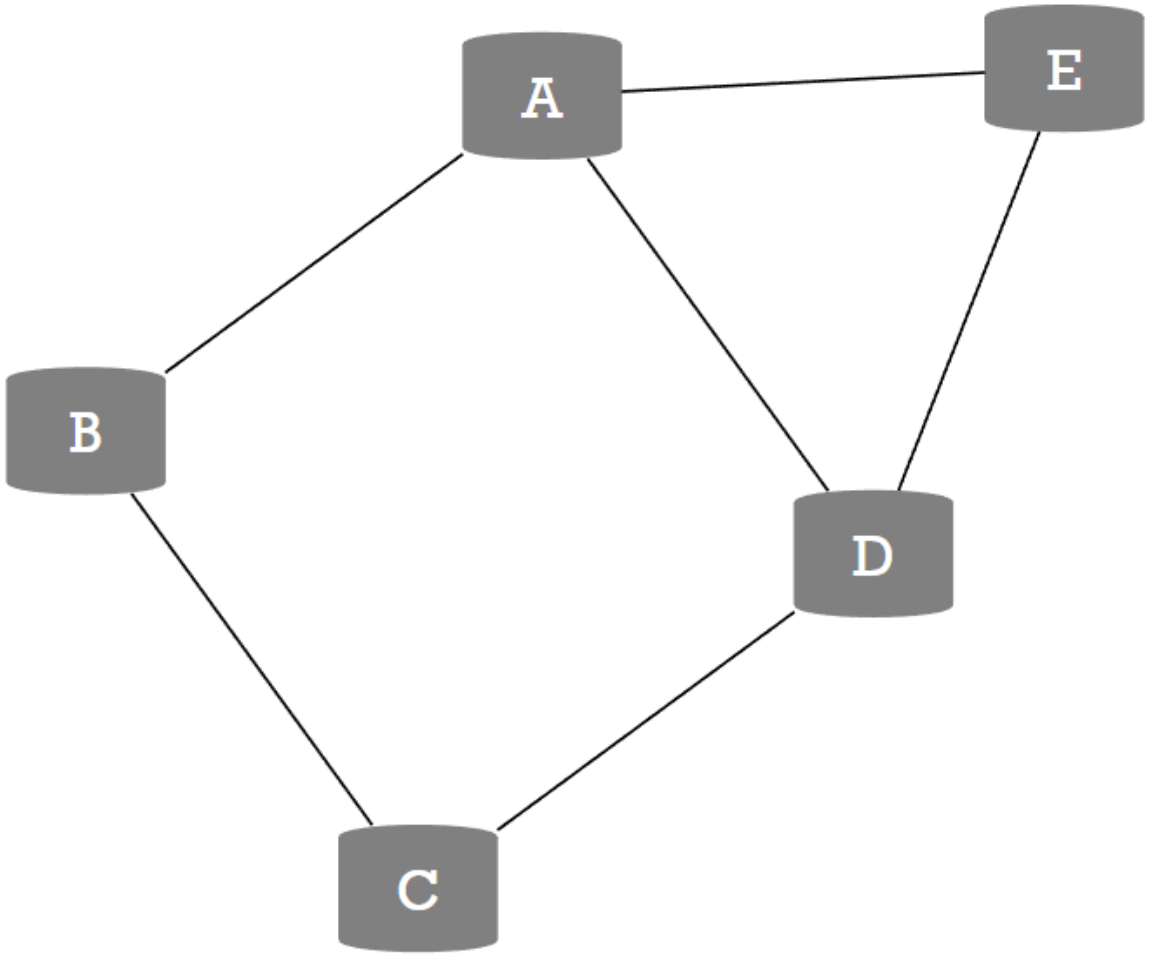
Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.

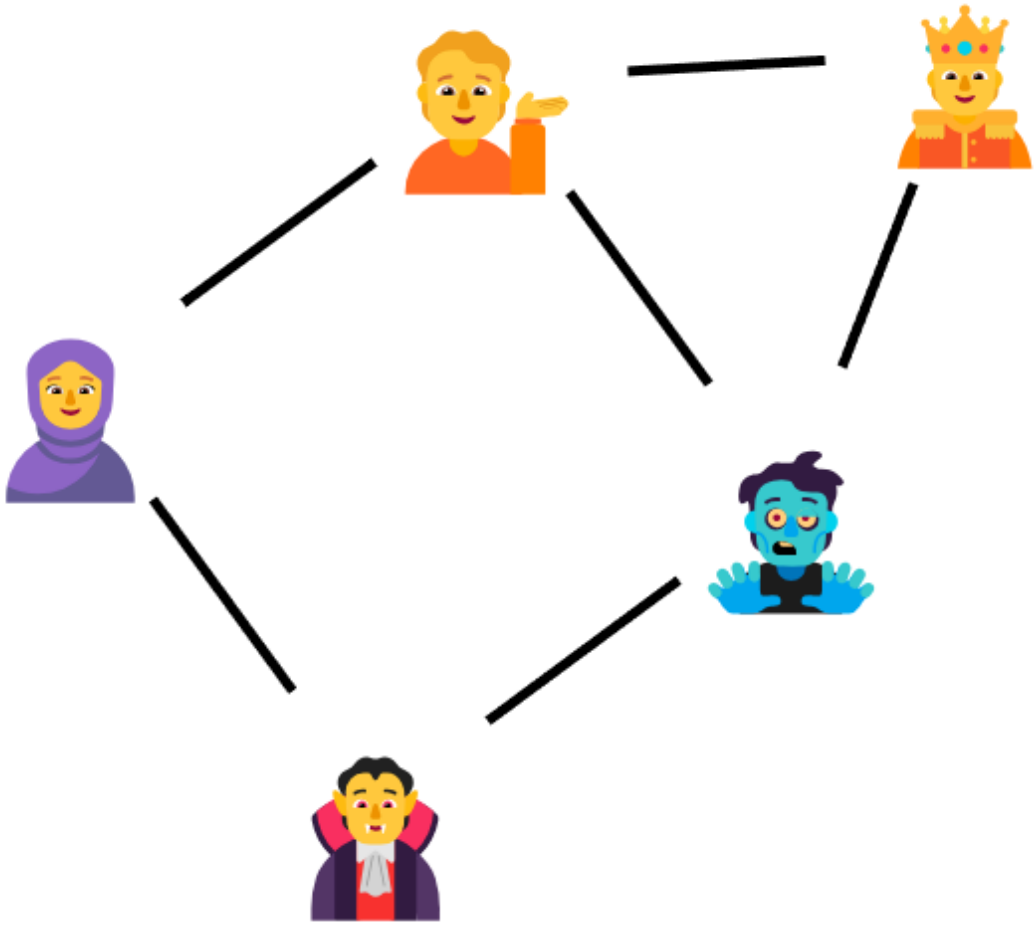
More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E).**

Here is an example of a Graph

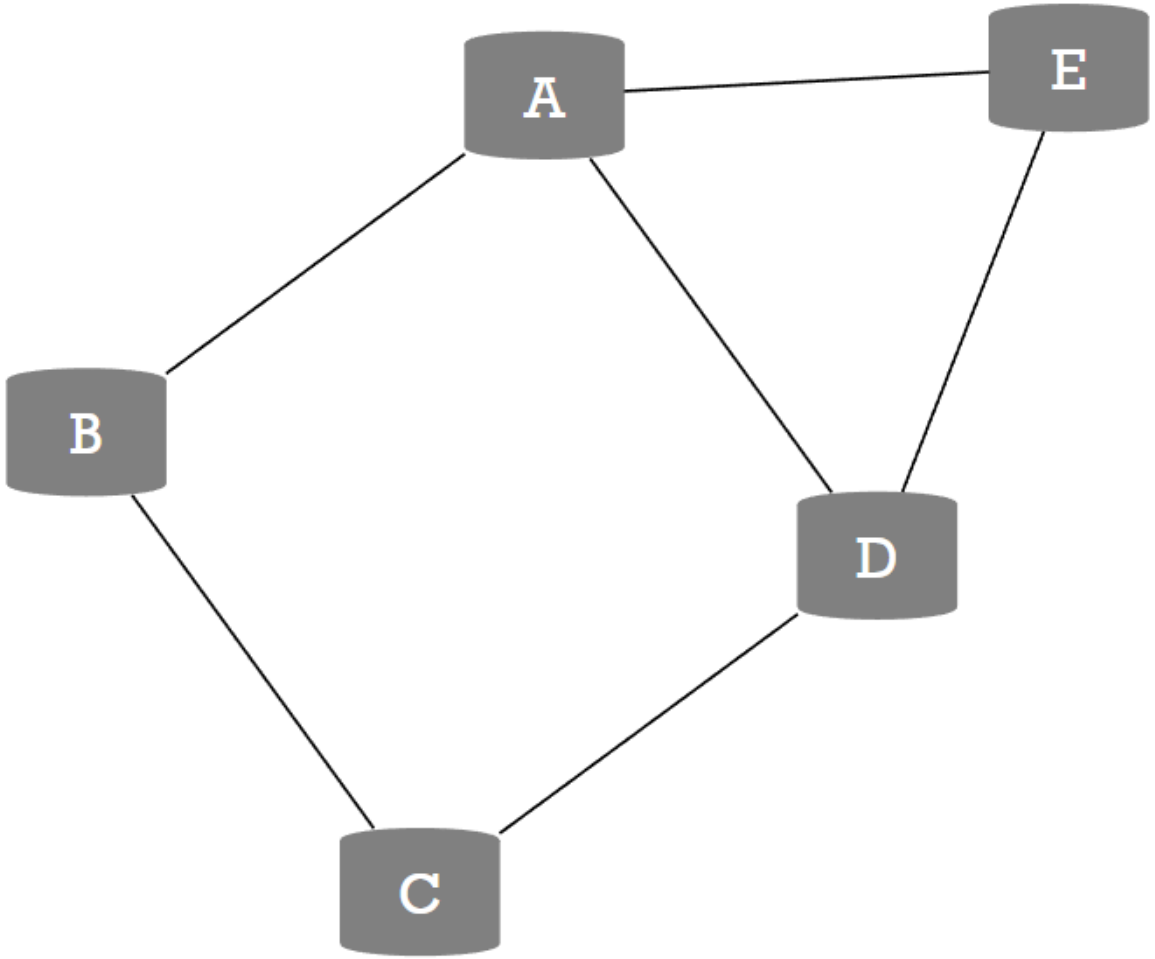


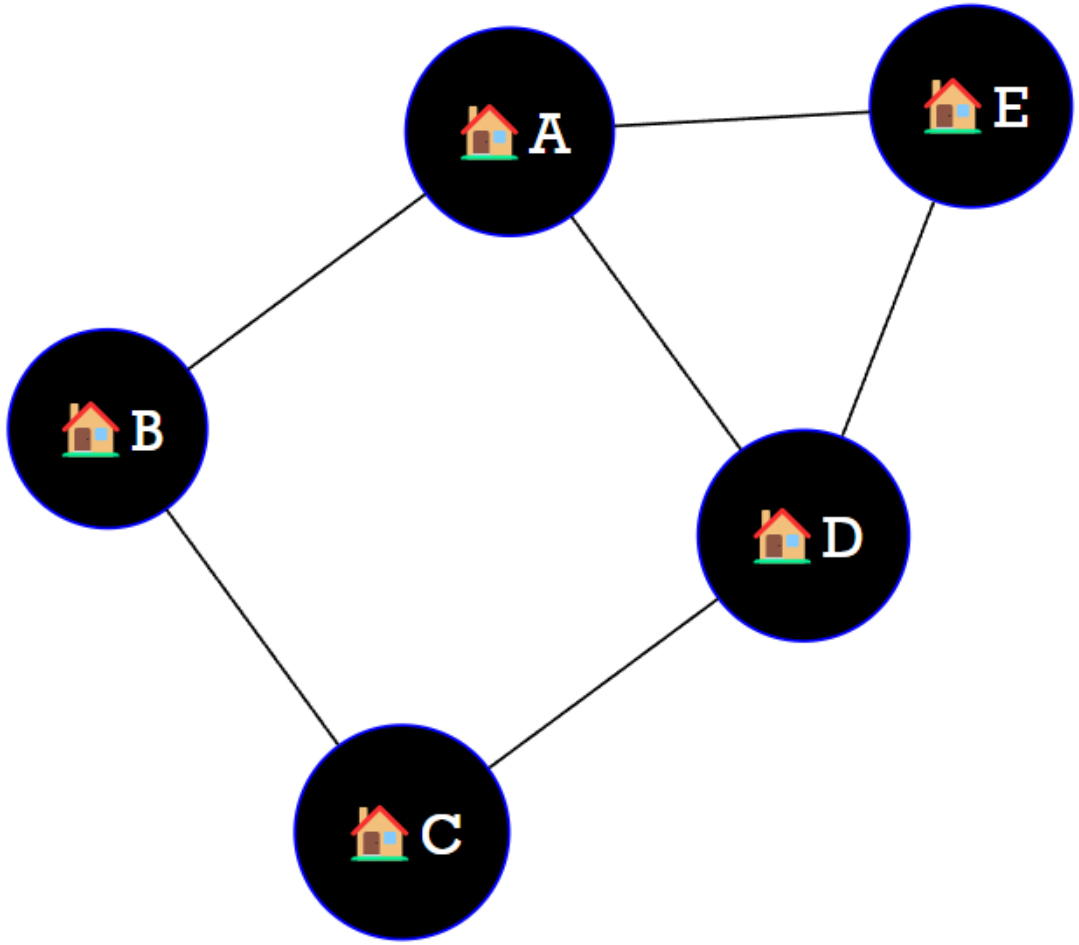
A better real-world example of a Graph - **A Social Network**





A better real-world example of a Graph - **Houses Connected by Roads**





Types of Graphs

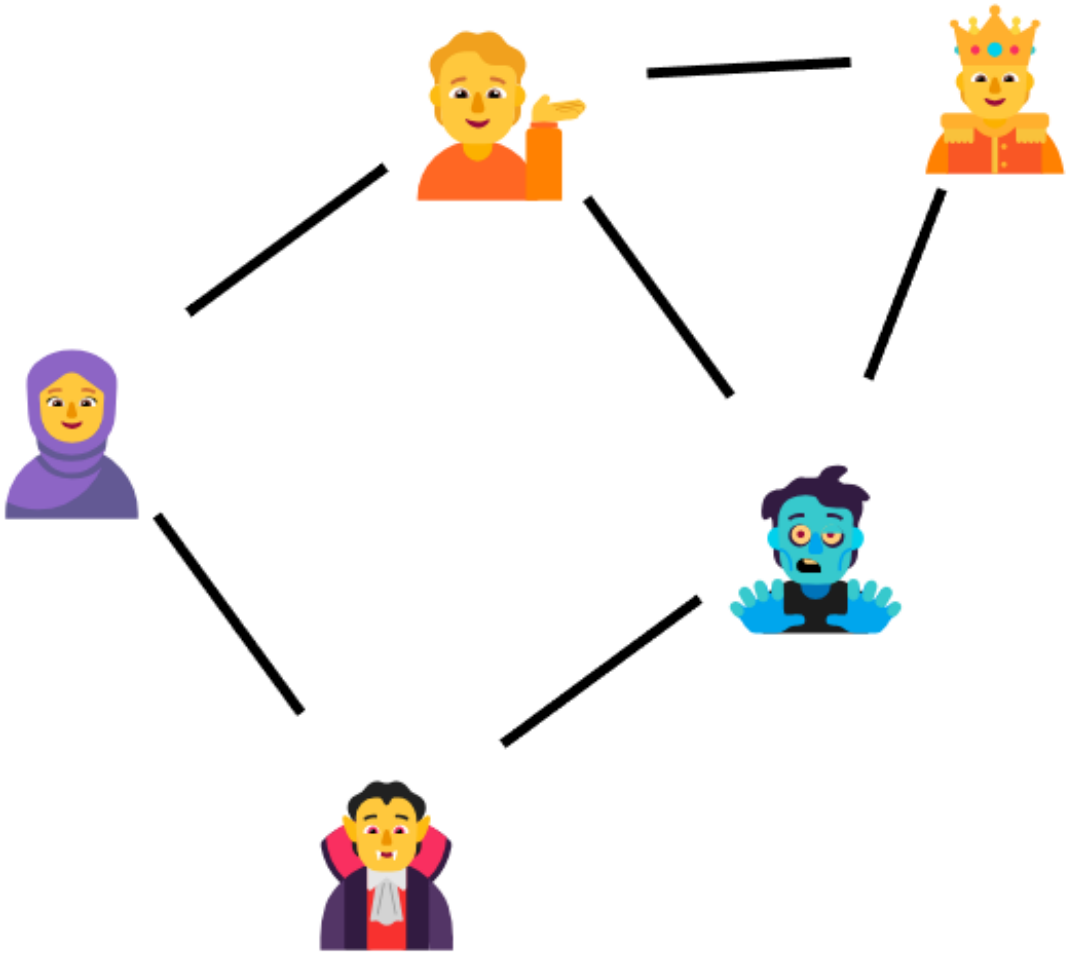
- Undirected Graphs
- Directed Graphs
- Unweighted Graphs
- Weighted Graphs

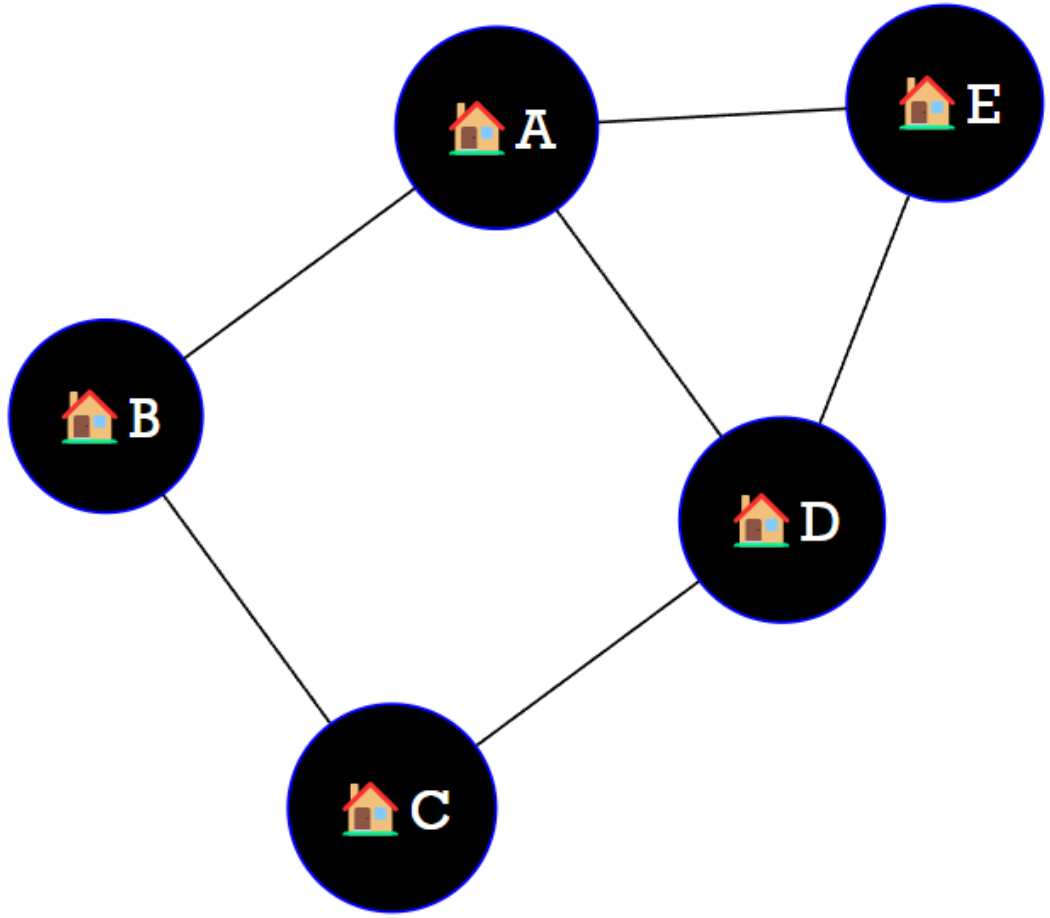
Undirected Graphs

A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal.

Example: A social network graph where friendships are not directional.

Examples – Undirected Graphs



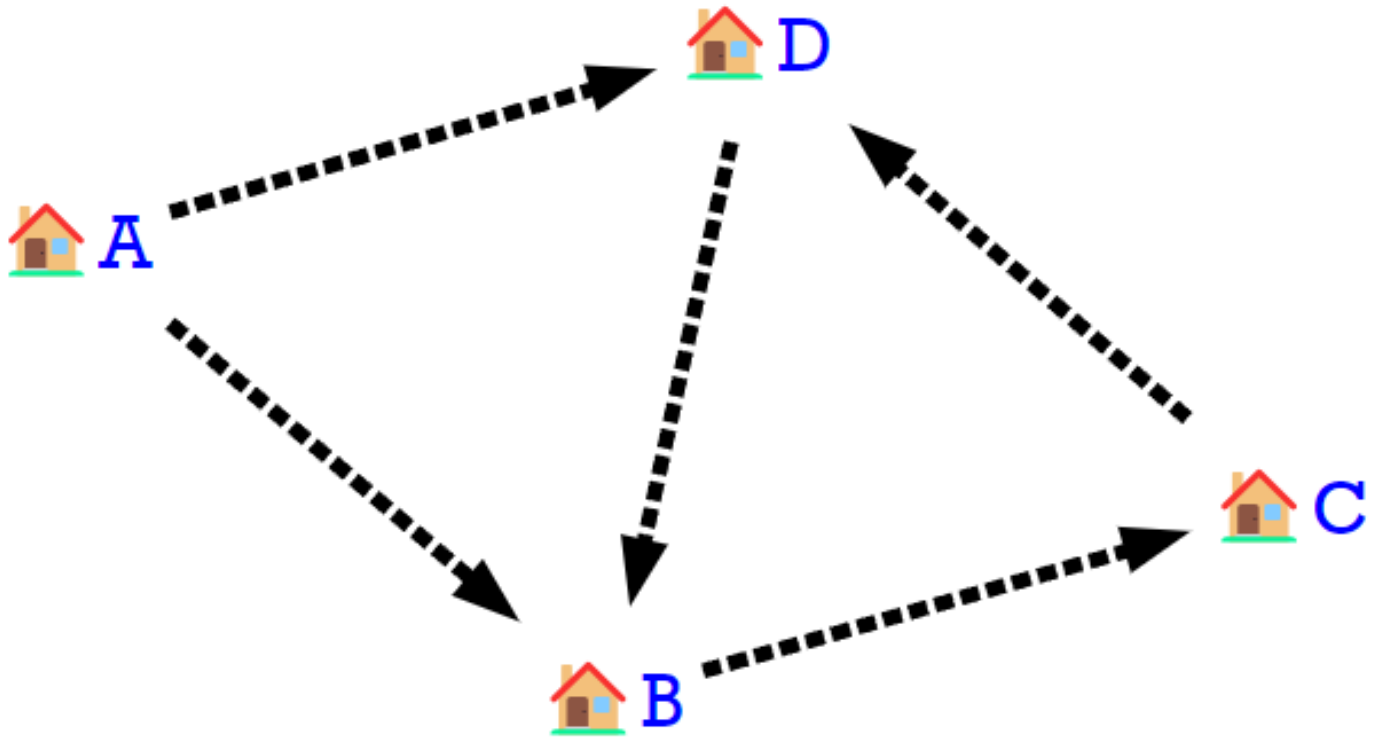


Directed Graphs

A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal.

Example: A web page graph where links between pages are directional.

Example – Directed Graphs



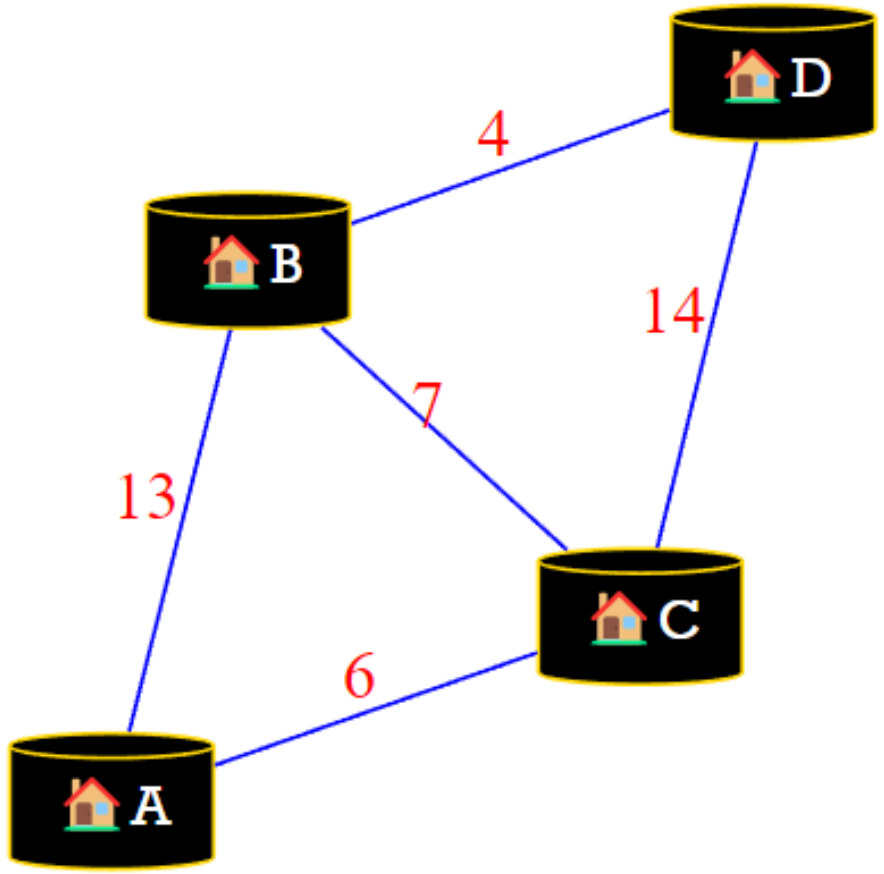
A Directed Graph
Houses Connected by roads Uni-Directionally

Weighted Graph

A graph in which edges have weights or costs associated with them.

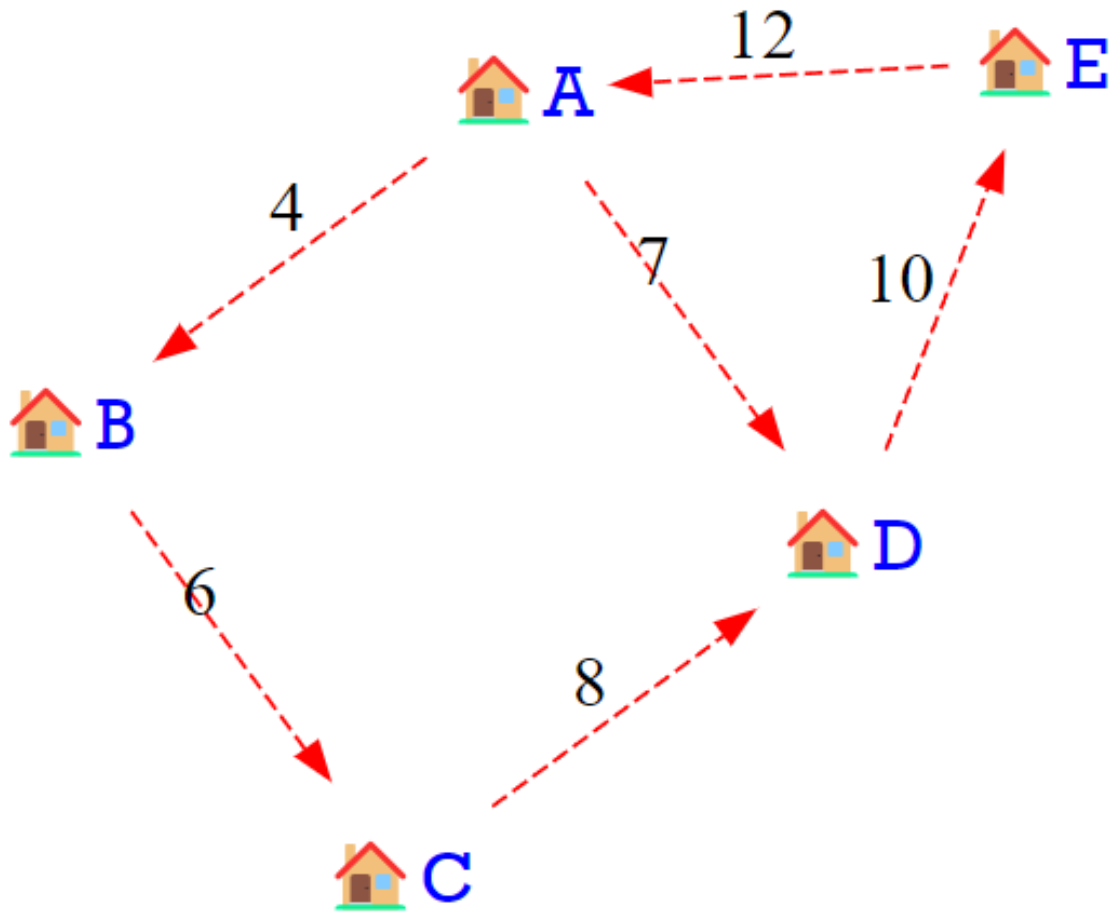
Example: A road network graph where the weights can represent the distance between two cities.

Example – Weighted (Undirected) Graph



An undirected Weighted Graph - Houses Connected with roads
Where each edge weight
being the length of road in KM

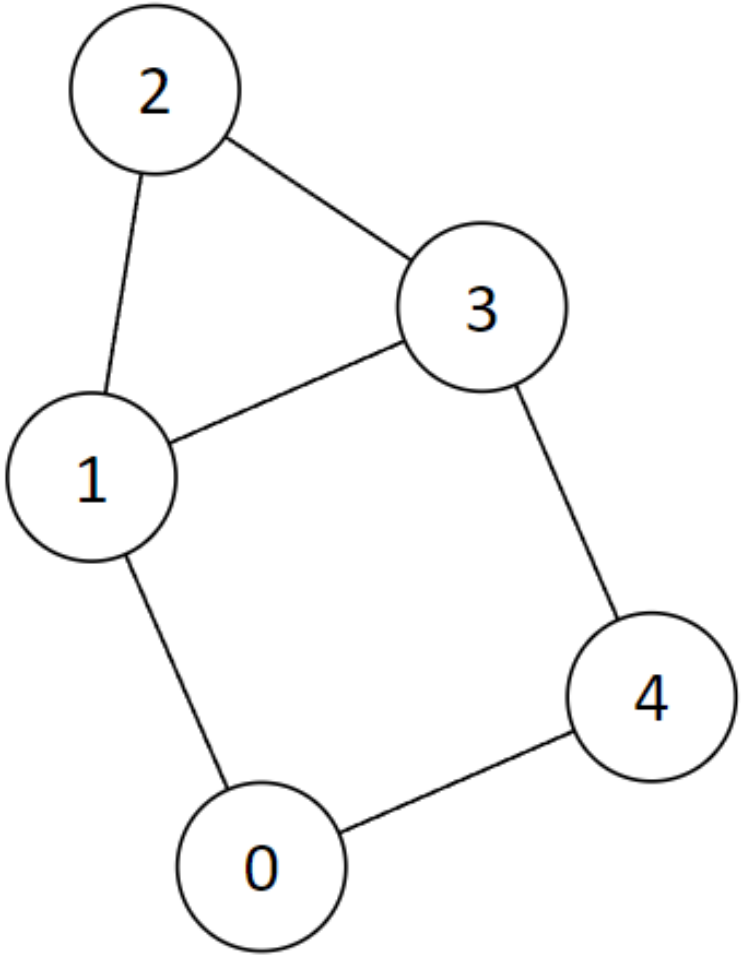
Example – Weighted (Directed) Graph



A Directed Graph
Houses Connected by roads Uni-Directionally

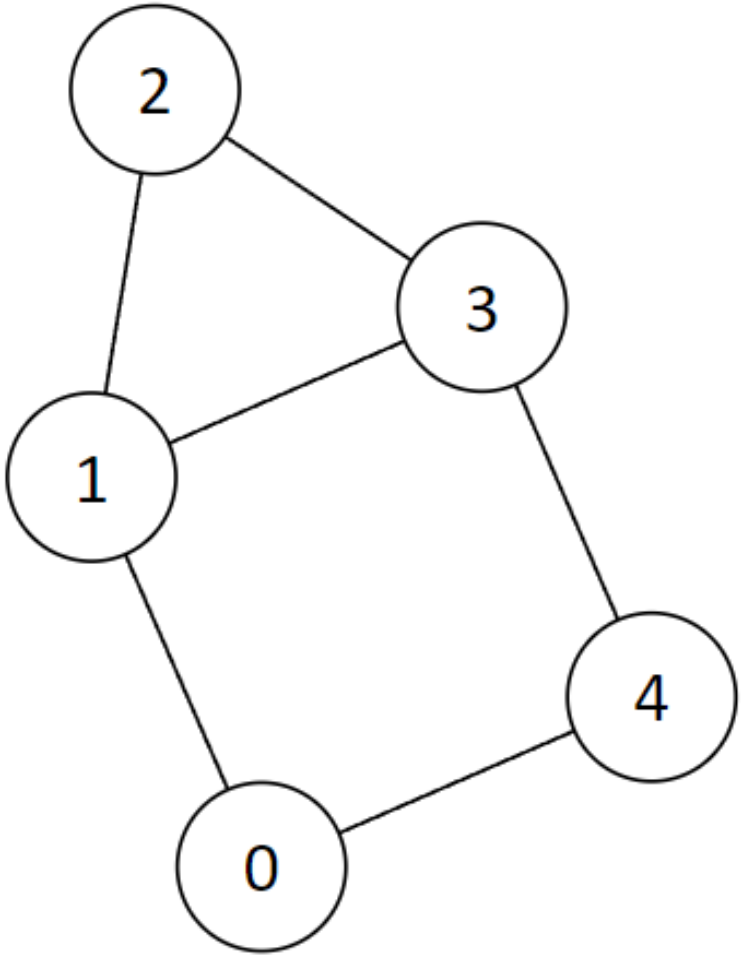
Graph Inputs

An un-directed graph as input



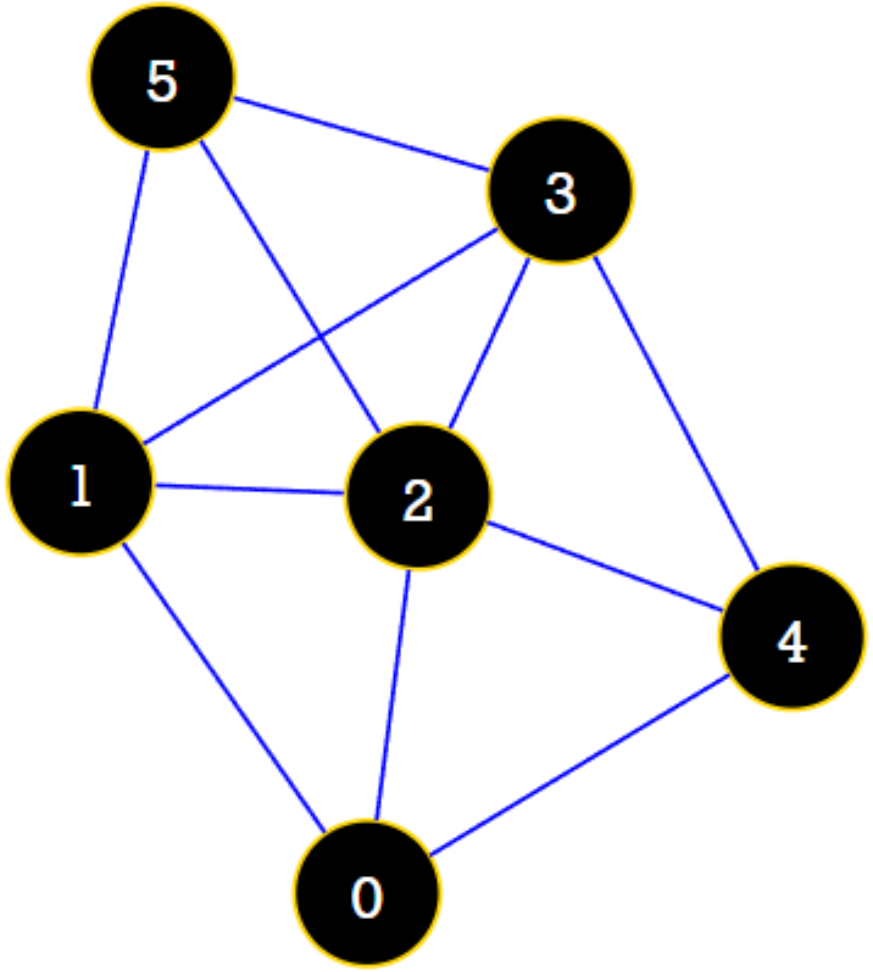
- 5
- 6
- 0 1
- 0 4
- 1 2
- 1 3
- 2 3
- 3 4

An un-directed graph as input



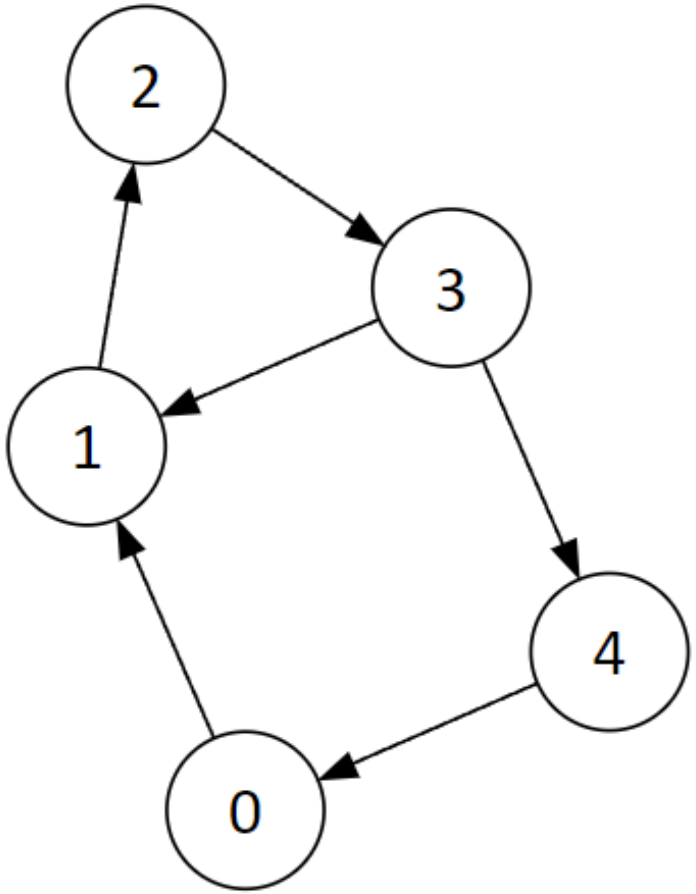
- 5 -> No. of Nodes
- 6 -> No. of Edges
- 0 1 -> Edge 1
- 0 4 -> Edge 2
- 1 2 -> Edge 3
- 1 3 -> Edge 4
- 2 3 -> Edge 5
- 3 4 -> Edge 5

Guess the input for the following undirected graph



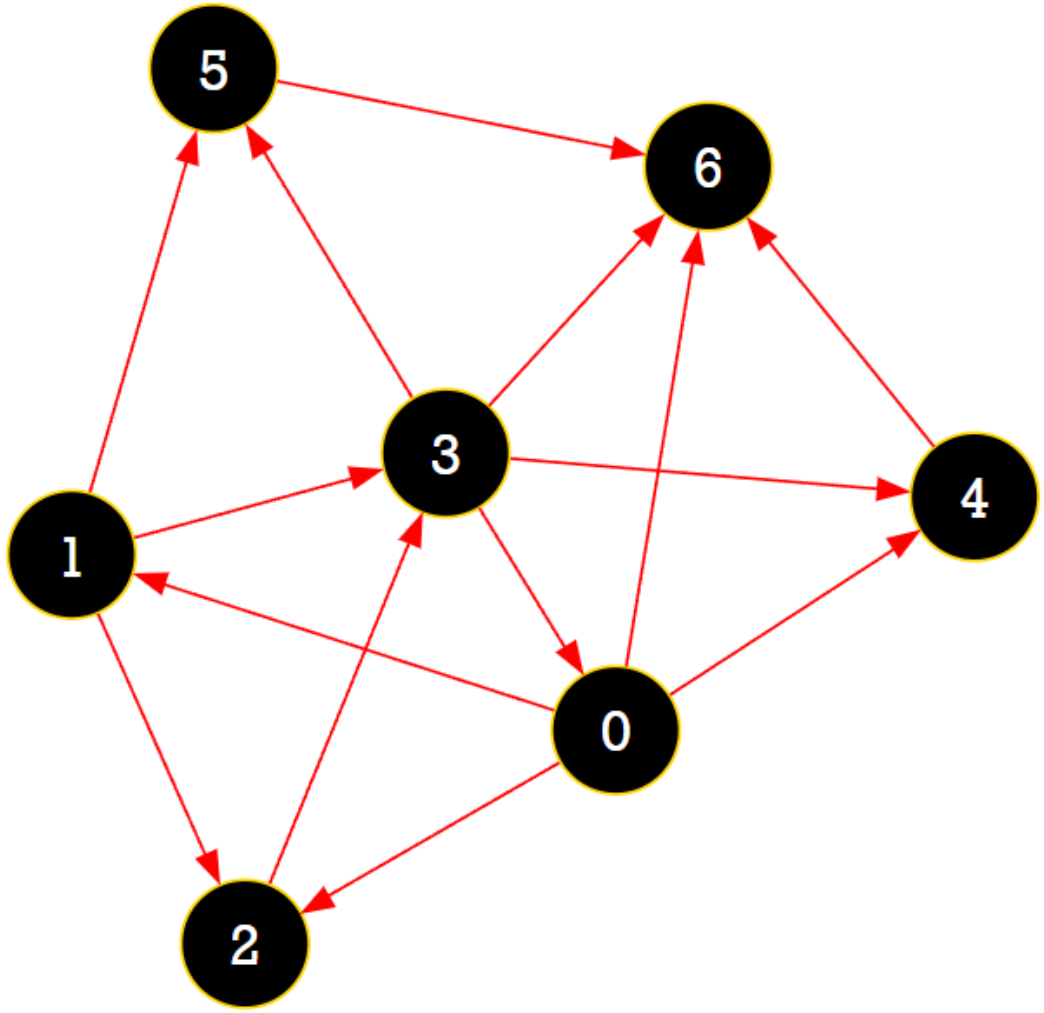
- 6 (Six - Nodes)
- 11 (Eleven - Edges)
- 0 1
- 0 2
- 0 4
- 1 2
- 1 3
- 1 5
- 2 3
- 2 4
- 2 5
- 3 4
- 3 5

A Directed graph as input



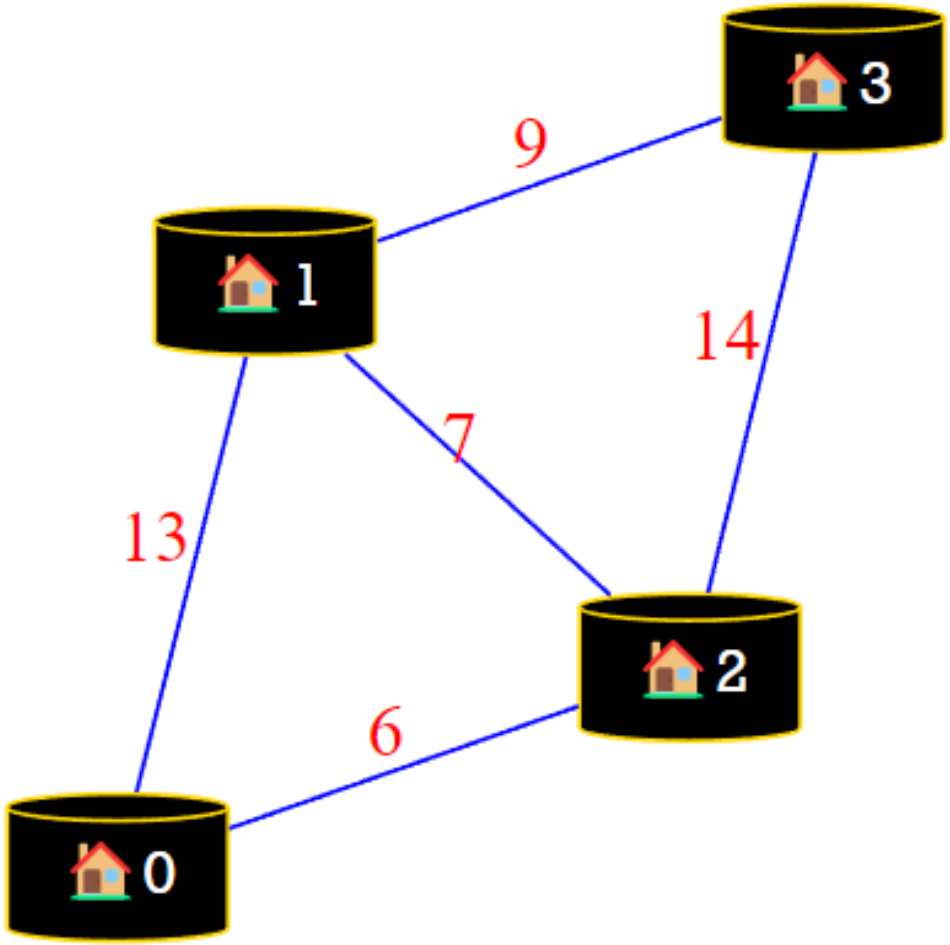
- 5
- 6
- 0 1
- 1 2
- 2 3
- 3 1
- 3 4
- 4 0

Guess the input for the following directed graph



- 7 (Seven - Nodes)
- 13 (Thirteen - Edges)
- 0 2
- 0 4
- 0 1
- 0 6
- 1 2
- 1 3
- 1 5
- 2 3
- 3 0
- 3 5
- 3 6
- 4 6
- 5 6

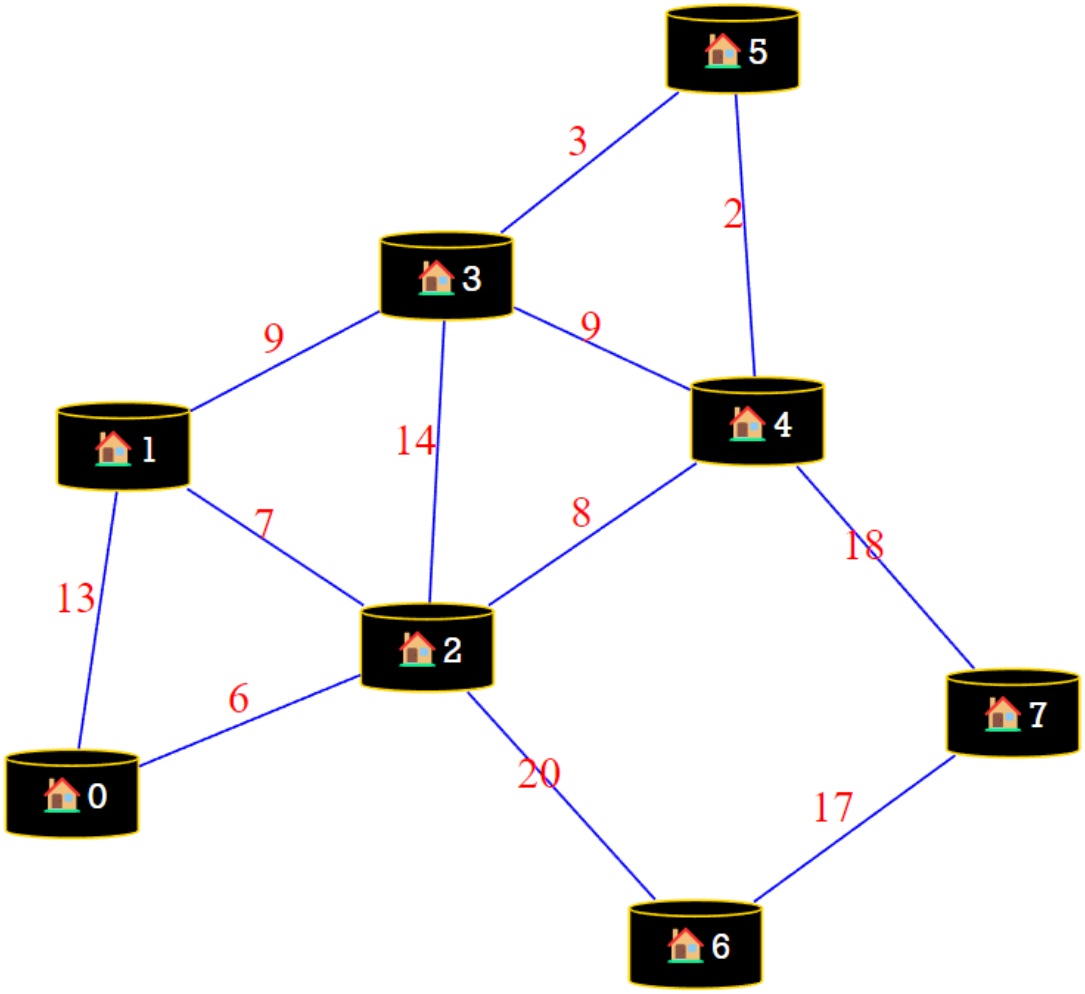
An un-directed weighted graph as input



An undirected Weighted Graph

- 4
- 5
- 0 1 13
- 0 2 6
- 1 2 7
- 1 3 9
- 2 3 14

Guess the input for the following undirected weighted graph



An Undirected Weighted Graph

Graph Representation

Adjacency Matrix

Adjacency List

Construct Graphs from Inputs

Construct an Undirected Graph from the below input

	No. of Nodes	8
	No. of Edges	15
	Start	**End**
Edge 1	0	1
Edge 2	0	2
Edge 3	0	3
Edge 4	1	2
Edge 5	1	3
Edge 6	2	3
Edge 7	2	4
Edge 8	2	5
Edge 9	3	5
Edge 10	3	7
Edge 11	4	5
Edge 12	5	6
Edge 13	5	6
Edge 14	5	7
Edge 15	6	7

Construct a Directed Graph from the below input

	No. of Nodes	8
	No. of Edges	14
	From	**To**
Edge 1	0	1
Edge 2	1	2
Edge 3	1	6
Edge 4	2	3
Edge 5	3	1
Edge 6	3	4
Edge 7	4	0
Edge 8	4	5
Edge 9	5	0
Edge 10	5	7
Edge 11	6	3
Edge 12	6	2
Edge 13	6	7
Edge 14	7	1

Construct a Weighted (Undirected) Graph from the below input

	No. of Nodes	5	
	No. of Edges	10	
	From	**To**	**Edge Weight**
Edge 1	0	1	9
Edge 2	0	2	11
Edge 3	0	3	7
Edge 4	0	4	12
Edge 5	1	2	6
Edge 6	1	3	5
Edge 7	1	4	16
Edge 8	2	3	8
Edge 9	2	4	19
Edge 10	3	4	10