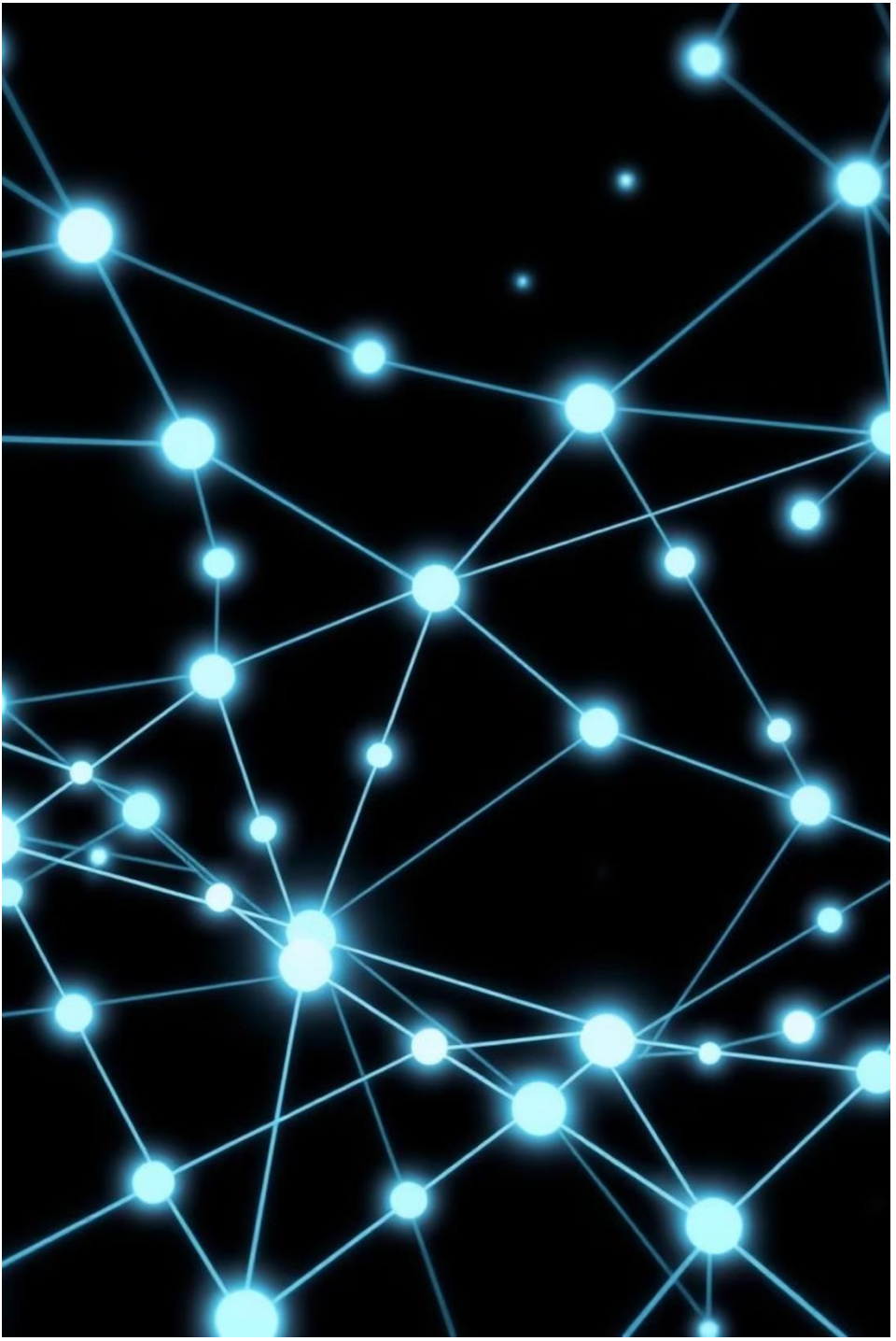


Data Structures

Lecture 2: Common DS

Instructor:
Md Samsuddoha
Assistant Professor
Dept of CSE, BU

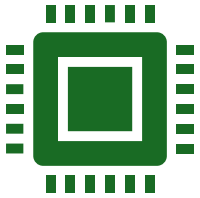
Contents

- Recap: DS
- Array
- Linked List
- Stack
- Queue
- Tree
- Graph
- Operations in DS

DS, why do they matter?

Organize and Store

Data structures are specialized formats for organizing and storing data in a computer so that it can be accessed and manipulated efficiently. They are fundamental to algorithms and programming.



Backbone of Software

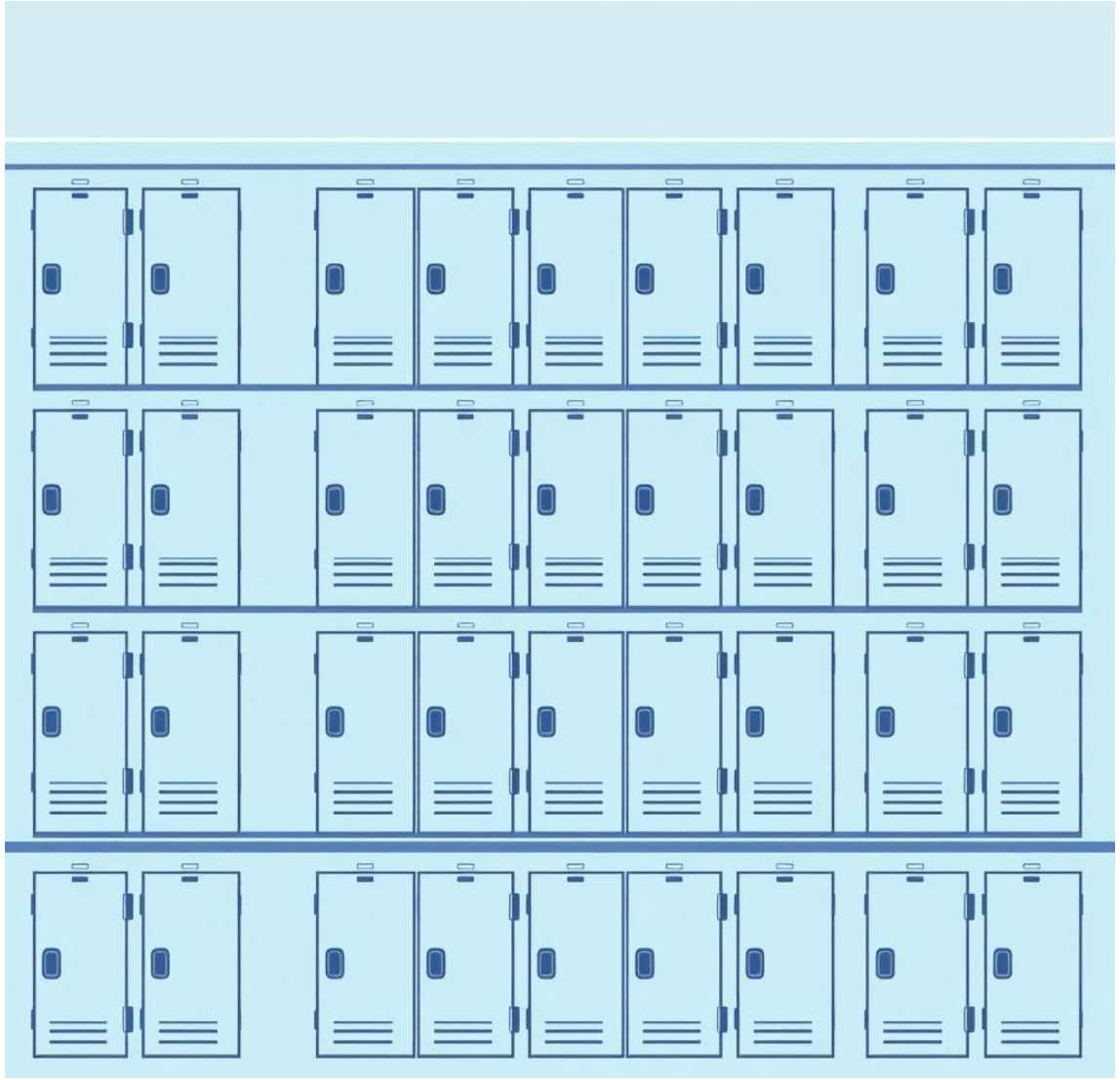
From operating systems to artificial intelligence, data structures underpin nearly every piece of software. They dictate how data flows and is processed, impacting performance and scalability.

Performance Critical

The choice of data structure directly impacts an algorithm's efficiency. An optimal structure can significantly **reduce processing time and memory usage**, leading to a faster and more responsive application.

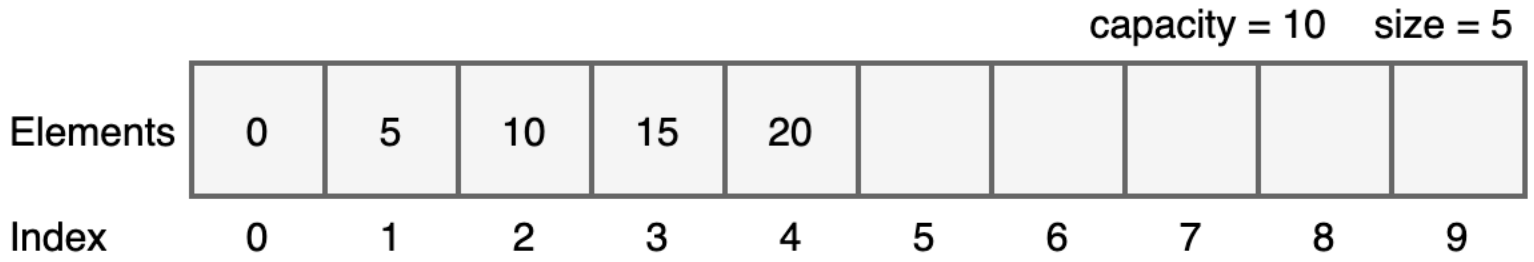
Arrays: The Contiguous Memory Workhorse

- **Fixed-Size, Contiguous Storage**
  - Arrays store elements in a fixed-size, contiguous block of memory.
  - This adjacency enables direct access to any element using its index in constant time (O(1)).
- **Efficiency Trade-offs**
  - While reads are lightning fast, insertions or deletions in the middle are costly (O(n)) as elements need to be shifted.
  - They are useful for lookup tables, image processing, and serve as the foundation for implementing other complex structures.

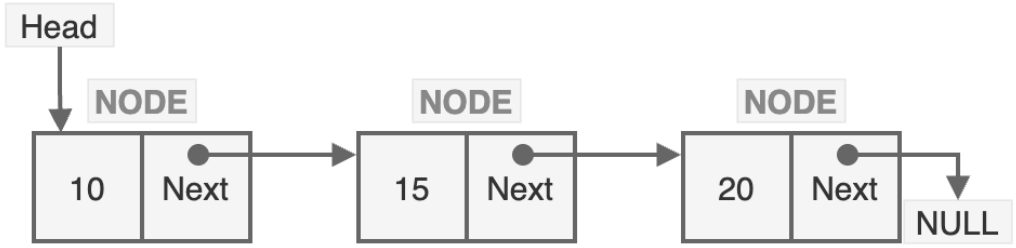


Array

- A linear data structure that stores elements **in contiguous memory locations**.
- Each element is identified by an **index** (starting from 0).
- All elements in an array are of the **same data type**.
- Almost all the programs use the Array in some places.
- Supported operations are Insertion, Deletion, Search, and Access by index.



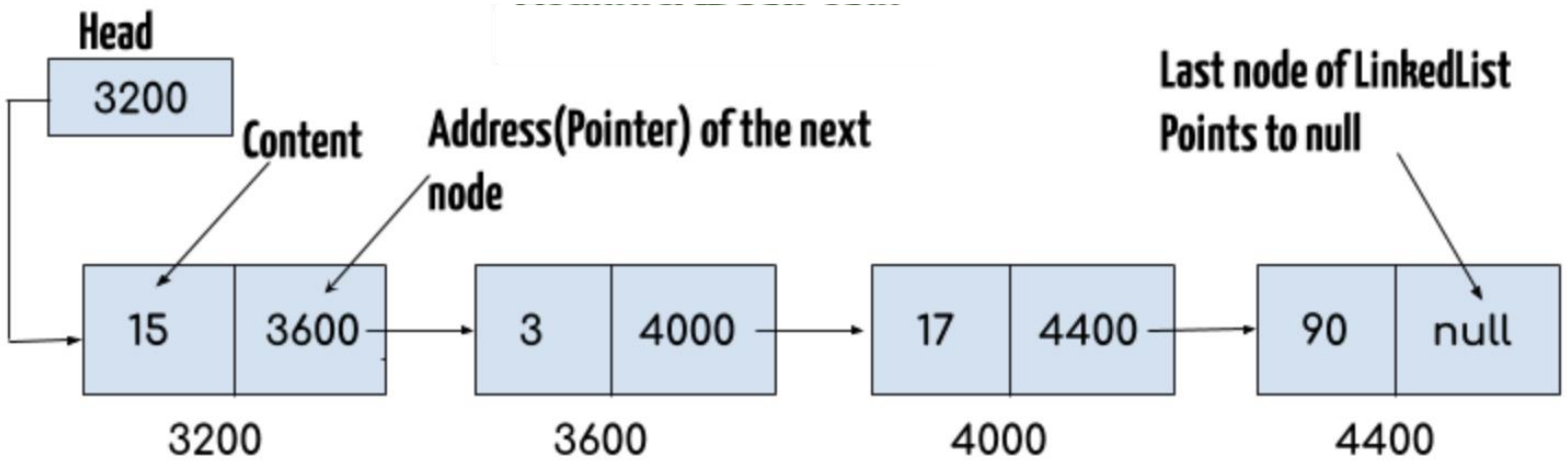
Linked Lists: Flexible Chains of Nodes



- A linked list is a collection of linear items. In contrast to the array, a linked list stores the elements in **non-contiguous memory** locations.
- Linked lists consist of nodes, each containing data and a pointer (or reference) to the next node in the sequence.
- Linked List is best used for storing dynamic data that requires frequent insertion and deletion operations.
- Supported operations are Insertion, Deletion, and Search.

Linked List

Node→ Data + Pointer

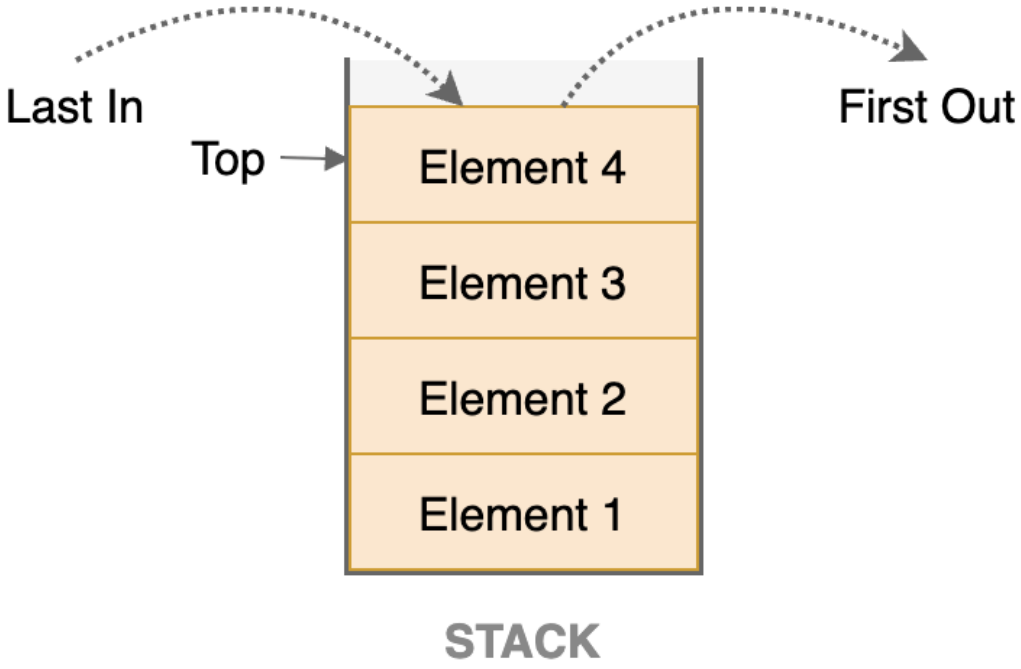


Linked List

- **Types**
  - Singly Linked List
  - Doubly Linked List
  - Circular Linked List
  - Doubly Circular Linked List
- **Applications**
  - Music playlist navigation
  - Web browser back/forward history
  - Undo functionality in software
  - Operating system's memory management

Stacks: Last In, First Out (LIFO) Collections

- **The Plate Analogy**
  - Imagine a stack of plates: you can only add a new plate to the top, and you can only take the top plate off. This "Last In, First Out" (LIFO) principle defines stack behavior.
- **Key Operations**
  - **Push:** Adds an element to the top of the stack.
  - **Pop:** Removes the top element from the stack.
  - **Peek:** Views the top element without removing it.
- **Ubiquitous Applications**
  - Stacks are critical for managing function calls in programming, implementing "undo" mechanisms in software, and evaluating mathematical expressions. They can be efficiently built using either arrays or linked lists.



Properties of Stack

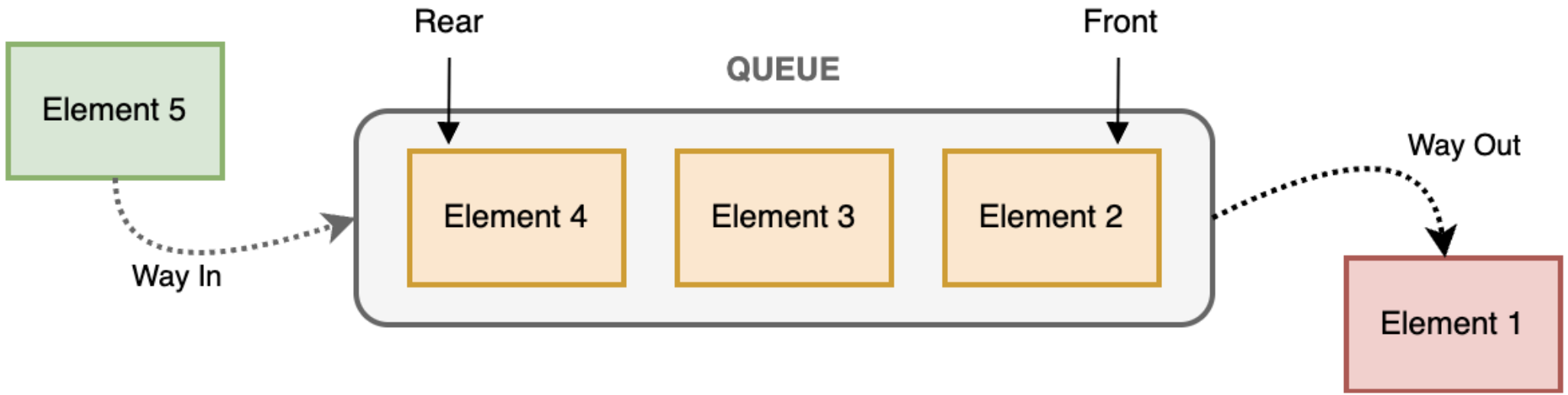
- **Top**: The top of the stack
- **Element**: The actual data
- **Push**: A new element is inserted on Top of the stack
- **Pop**: Element is removed from the Top of the stack
- **Underflow**: Stack is empty, but requested for pop
- **Overflow**: Stack reaches its capacity, but requested for push (applicable only for the array-based implementation of the stack)

Real Life Application of Queues

- **Undo/Redo in Editors**
  - Example: MS Word, Photoshop
  - Most recent action is undone first → **LIFO**
- **Browser Back Button**
  - When navigating webpages, previous pages are pushed to stack.
  - Pressing "Back" pops the last visited page.
- **Call Stack in Programming**
  - Tracks function calls.
  - When a function is called, it's pushed to the stack.
  - When it finishes, it's popped.
- **Expression Evaluation & Syntax Parsing:** Used in compilers to evaluate expressions like ((a+b)*c).
- **Reversing Text:** Pushing characters to a stack and popping them gives reversed order. (e.g abc→cba)

Queues: First In, First Out (FIFO) Lines

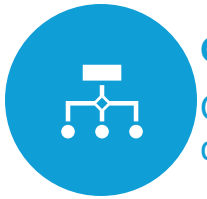
- **The Line Analogy**
  - Think of a queue like a traditional line at a store: the first person to enter the line is the first one to be served. This "First In, First Out" (FIFO) principle governs queue operations.
- **Fundamental Operations**
  - **Enqueue:** Adds an element to the rear of the queue.
  - **Dequeue:** Removes the element from the front of the queue.
- **Diverse Applications**
  - Queues are vital in various computing scenarios, including **breadth-first search (BFS)** algorithms, **managing tasks in operating systems**, handling **shared resources** (like printer queues), and buffering data in streams.


Rear
Front
QUEUE
Element 5
Element 4
Element 3
Element 2
Way In
Way Out
Element 1

Real Life Application of Queue

Ticket Booking System:
People are served in the order they arrive.

Printer Queue:
Documents sent to printer are processed in order.



CPU Task Scheduling:
OS schedules processes using queues (e.g., round robin).

Call Center Systems:
Incoming calls are queued and answered one by one.

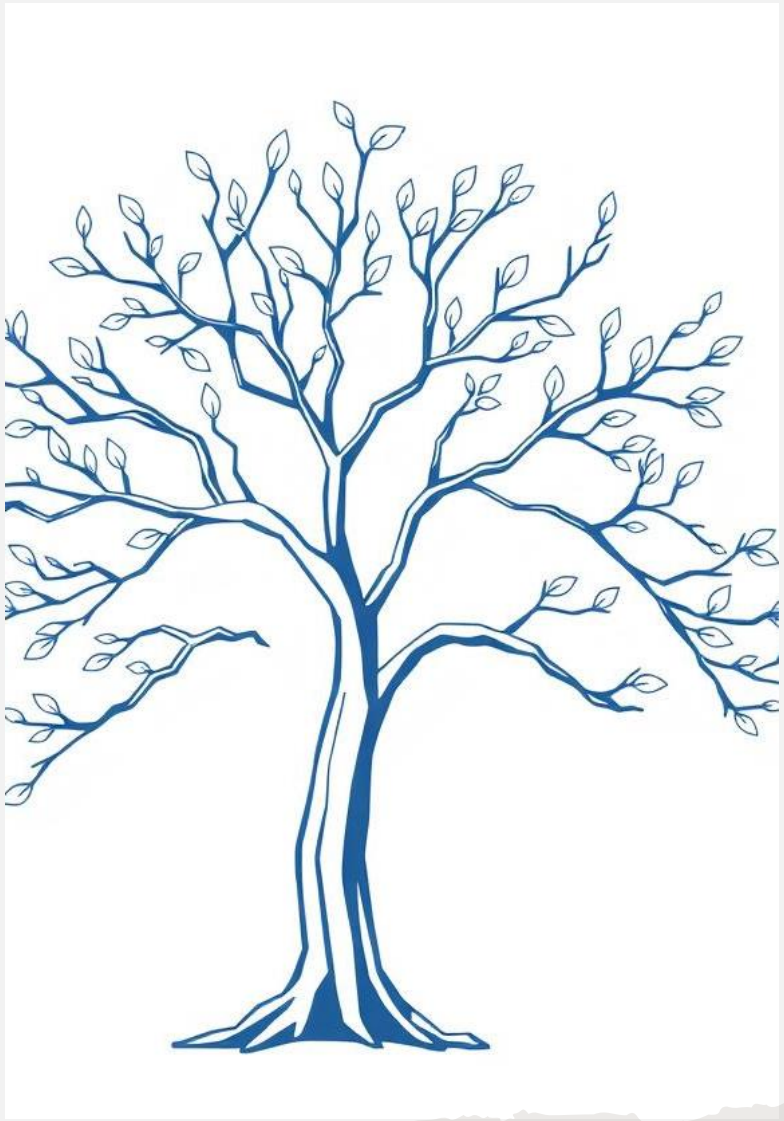
Customer Service:
Service counters like banks, hospitals, and help desks.

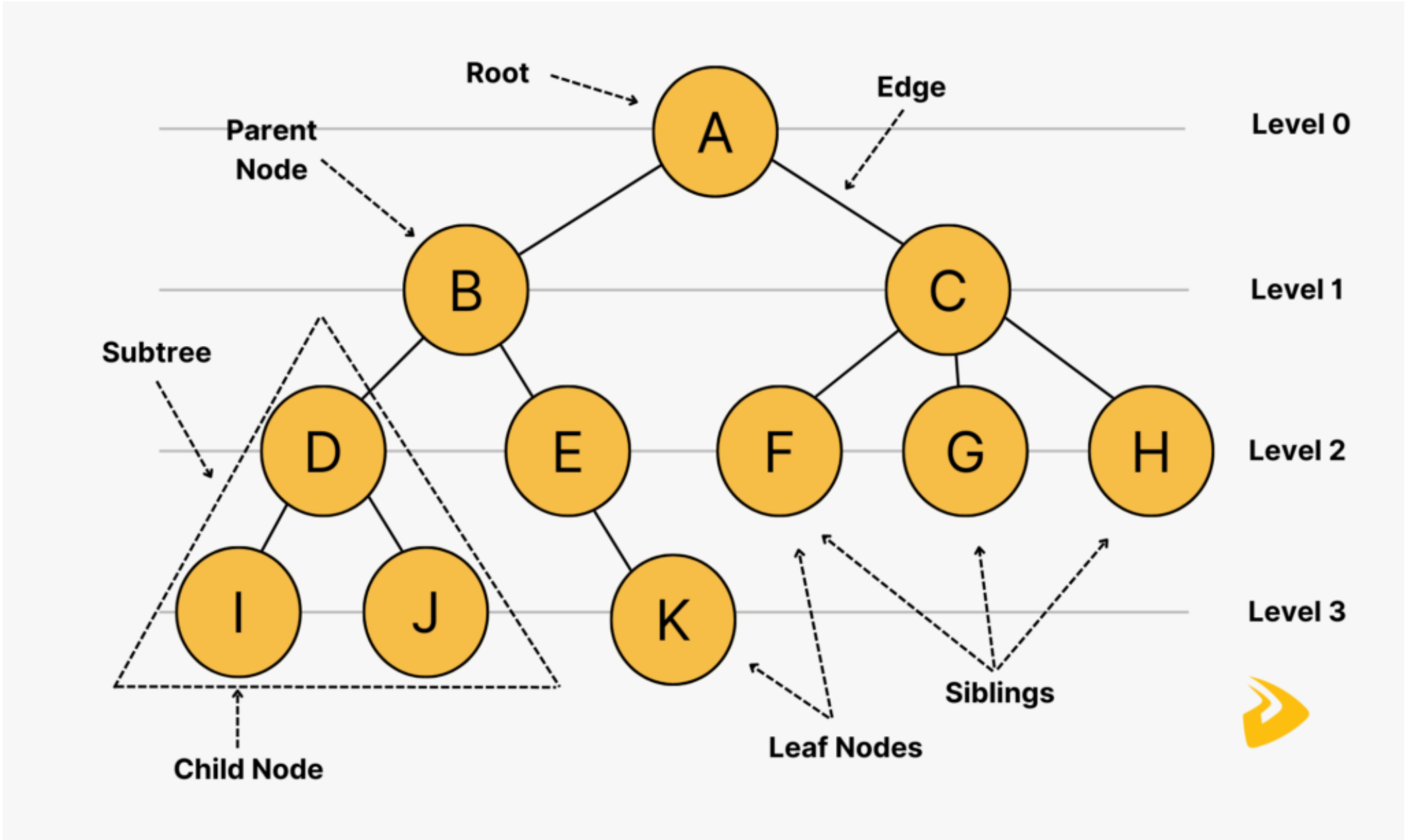
Messaging Systems:
Emails, SMS, or WhatsApp messages use queues to deliver messages in order.

Traffic Systems:
Vehicles waiting at a red light form a queue.

Trees: Hierarchical Data Structures

- Trees organize data in a **hierarchical structure**, with nodes connected in parent-child relationships. They start with a single root node and branch downwards, with no cycles, ensuring a clear path from root to any node.
- There are many specialized tree types, such as **binary trees** (each node has at most two children), **binary search trees** (ordered nodes for efficient searching), **heaps** (used in priority queues), and **tries** (for string retrieval).
- Trees are extensively used in **file systems** to represent directories and files, in **databases for indexing**, in parsing expressions (abstract syntax trees), and for efficiently organizing and searching hierarchical data like the Document Object Model (DOM) in web browsers.



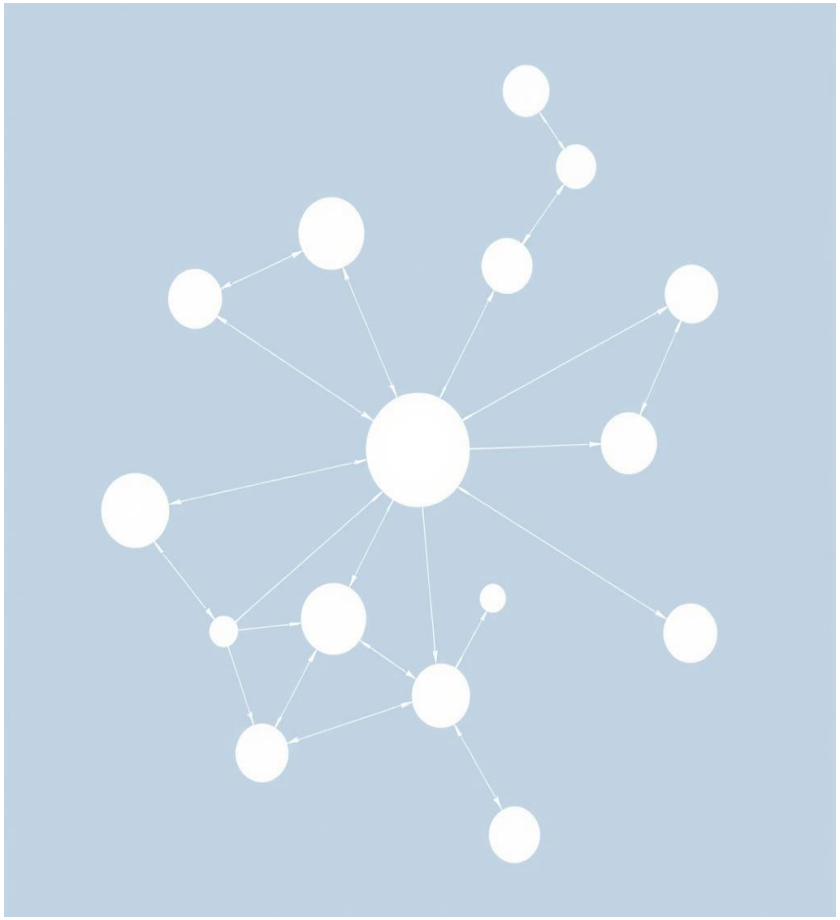

Root
Edge
Parent
Node
A
B
C
Level 0
Level 1
Subtree
D
E
F
G
H
Level 2
I
J
K
Level 3
Siblings
Leaf Nodes
Child Node

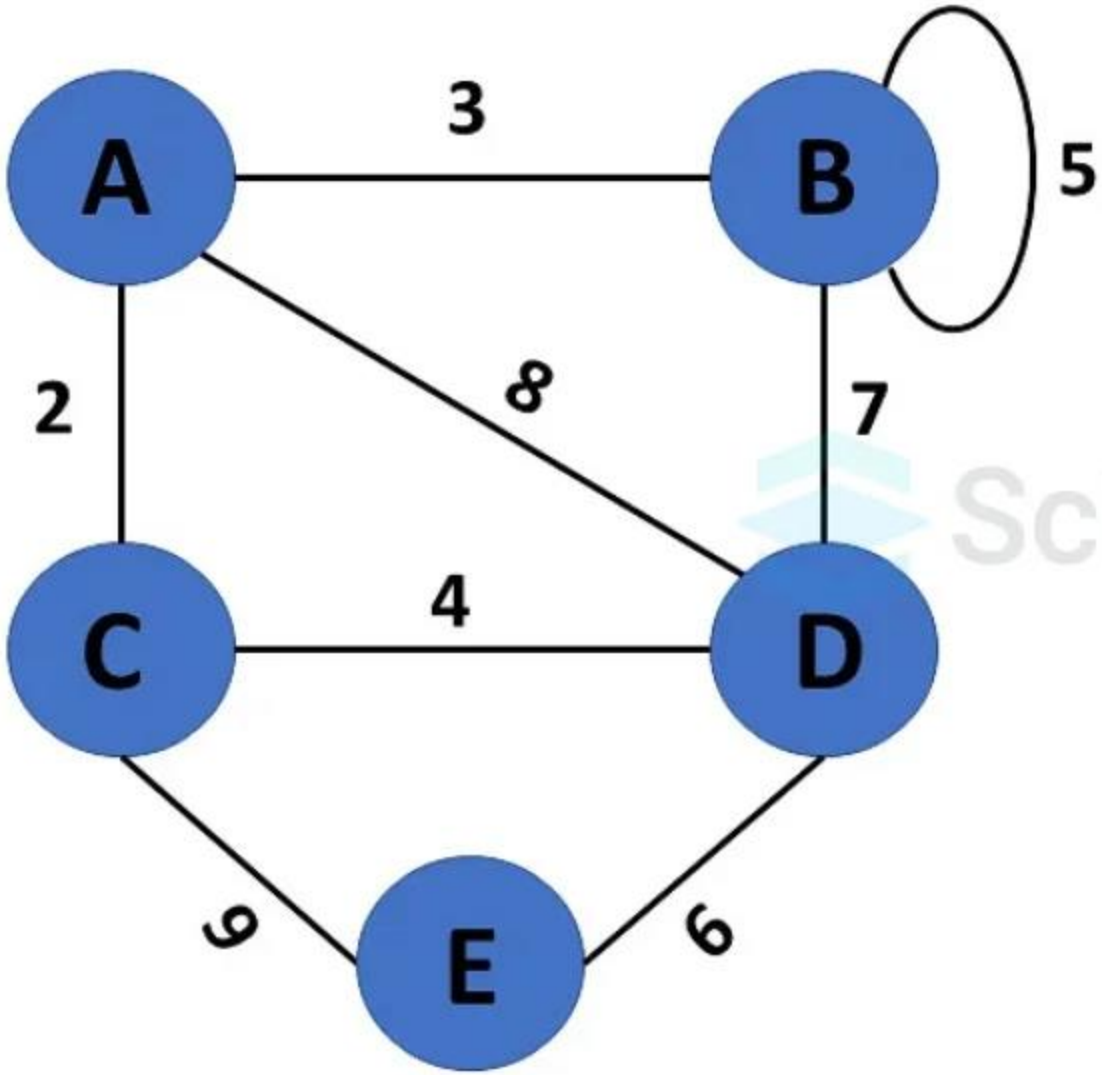
Properties of Trees

- **Root**: The topmost node.
- **Node**: Node consists of data, a pointer to the left child, and a pointer to the right child.
- **Parent**: A node directly above is called its parent.
- **Children**: All the nodes directly under it are called children.
- **Leaves**: All the nodes that have no children are called leaves.

Graphs: Complex Networks of Nodes and Edges

- **Nodes and Connections**
  - Graphs are highly versatile data structures that consist of a set of **nodes (or vertices) connected by edges**. These edges can be directed (one-way relationship) or undirected (two-way relationship), allowing for complex representations of real-world systems.
- **Modeling Real-World Systems**
  - Unlike trees, graphs can have cycles and nodes can have multiple parents, making them ideal for modeling intricate relationships such as **social networks, transportation maps, and web page links**. Each node represents an entity, and each edge represents a connection or interaction between them.
- **Powerful Algorithms**
  - Numerous powerful algorithms operate on graphs, solving problems like finding the **shortest path** between two points (e.g., GPS navigation), determining connectivity within a network, or optimizing resource flow. Understanding graphs opens up solutions to some of the most challenging computational problems.





	A	B	C	D	E
A	0	3	2	8	0
B	3	5	0	7	0
C	2	0	0	4	9
D	8	7	4	0	6
E	0	0	9	6	0

Properties of Graph

Term	Explanation	Example
Vertex	Point/node in a graph	A, B, C
Edge	Connection between two vertices	A — B
Degree	Number of edges connected to a vertex	A has 2 edges
Path	Sequence of connected vertices	A → B → C
Cycle	Path that ends at the starting vertex	A → B → C → A
Connected Graph	Every vertex is reachable	All nodes connected
Component	Independent connected subgraph	{A, B}, {C, D}

Real life applications of Graphs

- **Social Networks** (users = nodes, friendships = edges)
- **Maps and GPS** (places = nodes, roads = edges)
- **Computer Networks** (devices = nodes, connections = edges)
- **Web Pages** (pages = nodes, links = edges)

Data Structure Operations

- **Traversing:** It is the process of accessing each record of a data structure exactly once.
- **Searching:** It is the process of finding the location of a given value within a data structure.
- **Inserting:** It is the process of adding a new record into a data structure.
- **Deleting :**It is the process of removing an existing record from a data structure.

Apart from these typical data structure operations, there are some other important operations associated with data structures, such as:

- **Sorting:** It is the process of arranging the records in a particular order, such as alphabetical, ascending, or descending.
- **Merging:** It is the process of combining the records of two different sorted data structures to produce a single sorted data set.

Comparing Data Structures: When to Use What?

Structure	Access Time	Insertion/ Deletion	Memory Use	Use Case Examples
Array	O(1)	O(n)	Low	Static lists, lookup tables, fixed-size datasets
Linked List	O(n)	O(1)	Higher	Dynamic lists, implementing stacks/queues, memory-efficient insertions
Stack	O(1)	O(1)	Low	Function call management, undo/redo features, expression parsing
Queue	O(1)	O(1)	Low	Task scheduling, breadth-first search (BFS), buffering data
Tree	O(log n)*	O(log n)*	Moderate	Hierarchical data (file systems), databases (B-trees), efficient searching
Graph	Varies	Varies	High	Social networks, mapping/routing, dependency management, complex relationships

References

- **Chapter 3:**
  - **Data Structures using C** by E. Balagurusamy

Thank You