GeeksforGeeks

GATE फर्टे

CSE

DATA STRUCTURE

SHORT NOTES

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

Array

- **Array**: A collection of **homogeneous elements** stored in **contiguous memory**.
- **Indexing**: In C, starts from **0**.
  int arr[10]; → arr[0] is the first element, arr[9] is the last.
- **Types**:
  - 1D Array: Linear structure, e.g. int a [10];
  - 2D Array: Matrix-like, e.g. int a [2][3];
    int a[2][3] = { {0,0,0}, {1,1,1} };
  - Multi-dimensional: int a [3][2][4]; (3D array)

Memory & Access

- **Base Address**: Address of the first element.
- **Address Calculation**:
  - **1D**: a[k] = base + k * w
  - **2D Row-Major**: a[i][j] = base + ((i * n) + j) * w
  - **2D Column-Major**: a[i][j] = base + ((i) + j * m) * w
    Where:
  - m = rows, n = cols, w = size of element

Fixed size (static memory allocation)
Lower Bound (L.B): 0 in C
Upper Bound (U.B): n-1
Range: U.B - L.B + 1
operation on the 1d array.

Operation	Time Complexity	Explanation
Access	O (1)	Direct access using index: arr[i] → CPU calculates the address directly using formula Base + i * size
Insertion	O (n)	If insertion is at **beginning** or **middle**, all subsequent elements must be **shifted right**
Deletion	O (n)	If deleting from **start** or **middle**, elements must be **shifted left** to fill the gap

Sparse Matrix

A **sparse matrix** is a matrix in which **most of the elements are zero**.
If the number of zero elements > number of non-zero elements, the matrix is sparse.

Lower Triangular Matrix

A **lower triangular matrix** is a **square matrix** where all elements **above the main diagonal are zero**.
A[i][j] = 0 for all i < j
Must be **square** (n × n).
Non-zero elements are on or **below the main diagonal**.

Upper Triangular Matrix

An **upper triangular matrix** is a **square matrix** in which all elements **below the main diagonal are zero**.
A[i][j] = 0 for all i > j
Must be a **square matrix** (n × n)
Only elements **on or above** the main diagonal can be **non-zero**

Elements below the diagonal are always **zero**

Upper Triangular Matrix

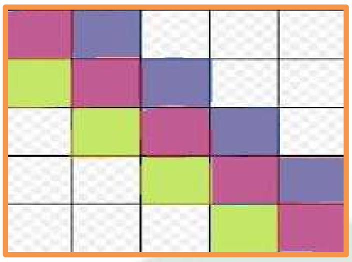
$$U = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix}_{4\times4}$$

Lower Triangular Matrix

$$L = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}_{4\times4}$$

Tridiagonal matrix

A **tridiagonal matrix** is a square matrix where **non-zero elements exist only on the main diagonal**, **just above it**, and **just below it**.
A[i][j] ≠ 0 only if i == j, i == j+1, or i == j–1
Else, A[i][j] = 0
Sum of all the element is **3n-2**.

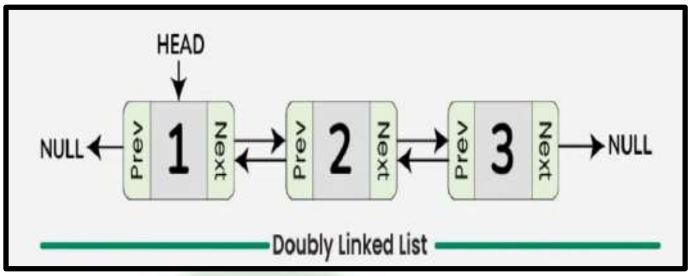


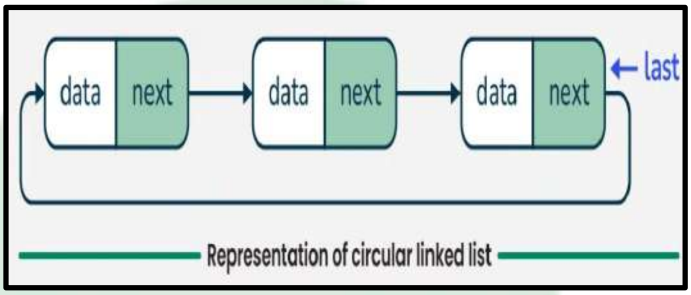
A **linked list** is a linear data structure where elements (called **nodes**) are stored in **non-contiguous** memory locations and connected using **pointers**.
Each node contains:

- **Data**
- **Pointer (next)** to the next node

Node structure in the C

```
struct Node {
  int data;
  struct Node* next;
                };
```





Advantages:

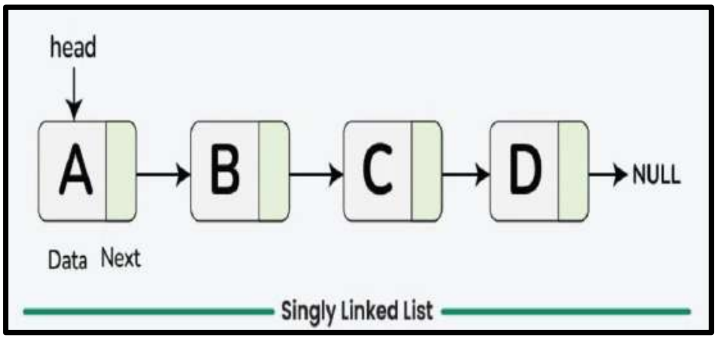
- Dynamic size (unlike arrays)
- Efficient insertions/deletions (at beginning/middle)

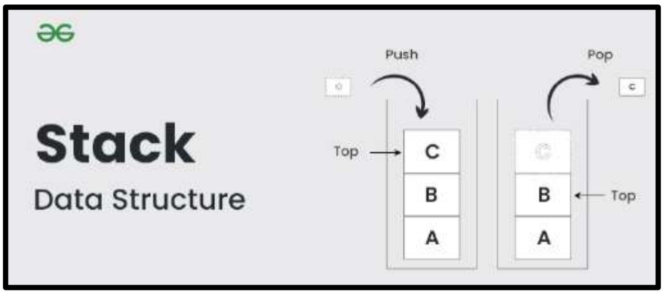
Disadvantages:

- No random access (O(n) access time)
- Extra memory for pointers

Stack

A **stack** is a linear data structure that follows the **LIFO** principle
Last In, First Out
The last inserted element is the first to be removed.





Operation	Description	Time Complexity
push(x)	Inserts element x at the top	O(1)
pop()	Removes and returns top element	O(1)
peek() / top()	Returns top element without removing	O(1)
isEmpty()	Checks if the stack is empty	O(1)

Implementation Methods:

1. **Using Array** (Fixed size, static memory)
2. **Using Linked List** (Dynamic size)

Applications of Stack:

- Expression Evaluation & Conversion

  (Infix ↔ Postfix)
- Balancing symbols (brackets, parentheses)
- Function call tracking (recursion)
- DFS traversal (graph)
- Undo functionality
- Backtracking (like maze, Sudoku)
- Number of possible stack permutations $= \frac{{}^{2n}C_n}{n+1}$

```
struct Stack {
    int arr [10];
    int top;
};
arr []: stores the stack elements
top: points to the topmost element (initially -1)
```

Queue

A **Queue** is a linear data structure that follows the **FIFO** principle:

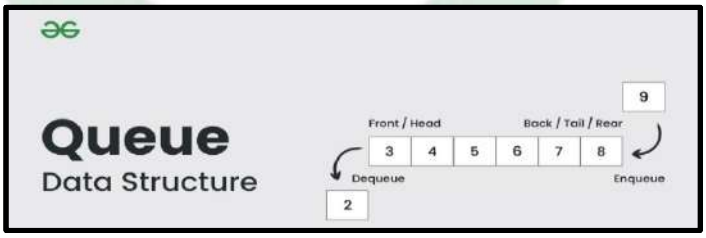
First In, First Out

The first element inserted is the first to be removed.

Real-World Examples:

- Ticket line
- Print queue

CPU task scheduling



Operation	Description	Time Complexity
enqueue(x)	Insert element x at the rear	O (1) (array or LL)
dequeue()	Remove and return element from front	O (1)

Type	Description
Simple Queue	Basic FIFO queue (insertion at rear, deletion at front)
Circular Queue	Last position connects back to first (solves overflow in array)
Deque (Double-Ended Queue)	Insertion/deletion possible from both ends
Priority Queue	Elements served based on priority, not position

In a **simple/linear queue using array**, after a few enqueue and dequeue operations:

- The **front** moves ahead
- But **rear** reaches the end of array
- Even though free space exists at the beginning, we can't use it

This leads to a **false overflow**.

Solution:

In a **circular queue**, we connect the **rear** back to **front**, forming a **circle**.

Condition	Formula
Empty	front == -1
Full	(rear + 1) % SIZE == front
Enqueue	rear = (rear + 1) % SIZE
Dequeue	front = (front + 1) % SIZE

Tree Type	Description
Binary Tree	Each node has ≤ 2 children
Full Binary Tree	All non-leaf nodes have 2 children
Complete Binary Tree	All levels filled except possibly the last (left to right)
Perfect Binary Tree	All internal nodes have 2 children & all leaves are at the same level
Balanced Tree	Height ≈ log(n), e.g. AVL tree
Binary Search Tree (BST)	Left < root < right
AVL Tree	Height-balanced BST
Heap	Complete binary tree (used in PQ)
B-Tree / B+ Tree	Multi-way trees for disk-based search

Priority queue

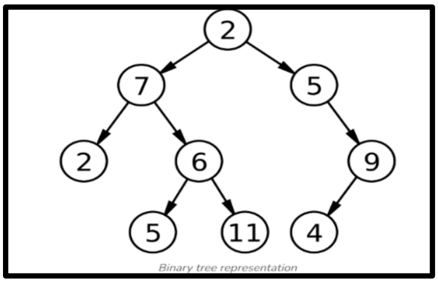
A **priority queue** is a type of **abstract data structure** in which each element is associated with a **priority**, and elements are served **based on their priority**, **not just insertion order**.

Tree data Structure

A **tree** is a **non-linear**, **hierarchical** data structure consisting of **nodes**, with a single **root node** and zero or more **child nodes**, forming a **parent-child** relationship.

Term	Meaning
Root	Topmost node (no parent)
Node	Element in the tree
Edge	Connection between nodes
Parent	Node with children
Child	Node with a parent
Leaf	Node with no children
Degree	Number of children of a node
Height	Max level from root to leaf
Depth	Distance from root to a node
Subtree	Tree formed from any node and its descendants

Binary tree



Binary tree representation

- Maximum Nodes in a Binary Tree of Height 'h' = $2^{h+1} - 1$
- Minimum nodes in a Binary tree= h+1
- The minimum possible height for **N** nodes is $\lfloor \log_2 N \rfloor$
- Total number of unlabelled binary tree with n node = $\frac{{}^{2n}C_n}{n+1}$
- Total number of labelled binary tree = $\frac{{}^{2n}C_n}{n+1} \times n!$
- Total number of the binary tree with given inorder/ preorder/ postorder = $\frac{{}^{2n}C_n}{n+1}$
- Number of the tree with inorder+ preorder =1, this is unique
- Number of the tree with inorder + postorder = 1, this is unique
- Number of the tree with preorder + postorder = many possible

K-ary Tree:

A **K-ary tree** is a tree in which every internal node has either **0 or exactly K children**.

Let:

- **N** = Total number of nodes
- **L** = Number of leaf nodes
- **I** = Number of internal nodes
- **K** = Maximum number of children per internal node

Relationship:

Each internal node has exactly K children:

N=K·I+1

Also, total number of nodes is the sum of internal and leaf nodes:

N=L+ I

So, equating both expressions for N

L+I=K·I+1

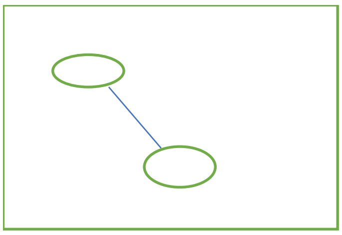
Complete Binary tree

This is the binary tree which is filled at second last level, and the insertion happens from the left to right.

For complete binary tree

- Minimum number of nodes is 2^h
- Maximum number of nodes is = $2^{h+1} - 1$

This is not the CBT, because it doesn't follow the properties.

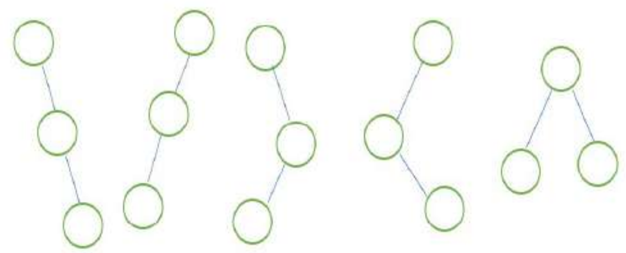


Binary Search Tree

Left < root < right

Every structure of the n node has unique binary search tree.

i.e. if we have 3 keys, then then we have the 5 structures



This are the structure with five nodes. Now we have the only 1 binary search tree with every structure.

So total binary search tree with n keys = $\frac{^{2n}C_n}{n+1}$

! Binary search tree is not the complete binary tree.

Insertion in a BST

Average case, O(logn)

Worst case, O(n)

Three Cases of Deletion:

1. **Case 1: Node has no children (Leaf Node)**
   - Simply **remove the node.**
   - No tree structure change.
2. **Case 2: Node has one child**
   - **Replace** the node with its **only child.**
   - Maintain the link with the parent.
3. **Case 3: Node has two children**
   - Replace the node with its:
     - **Inorder Successor** (smallest in right subtree) **or**
     - **Inorder Predecessor** (largest in left subtree)
   - Then **delete** the successor/predecessor recursively.

Time Complexity:

- **Best/Average Case**: O(logn)— for balanced BST
- **Worst Case**: O(n) — for skewed BST

AVL Tree

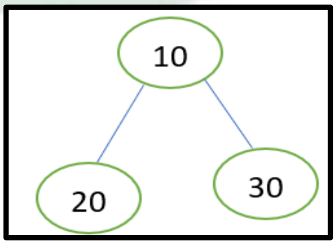
- A **self-balancing binary search tree (BST)** where the **difference in heights** of the left and right subtrees (called **balance factor**) of every node is **-1, 0, or +1**.

Balance Factor:
Balance Factor (BF)=Height of Left Subtree−Height of Right Subtree

- Valid values: **-1, 0, +1**
- If the balance factor becomes less than -1 or more than +1, **rotation** is needed to restore balance.

Properties:
Height of AVL Tree: O(logn)
Search/Insert/Delete Time Complexity: O(logn)
space Complexity: O(n)
minimum number of the nodes in the AVL tree **n(h) = n(h-1) + n(h-2) +1**
where the n(h)= number of the node at height h
LL, RR are single rotations
LR, RL are double rotations.
We check the balance factor from **bottom to top**, if we find any node not following the properties then we do the rotations.

Tree traversal

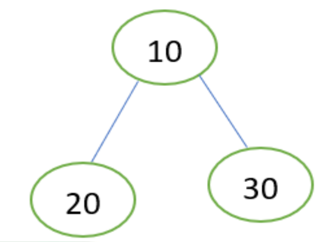


Inorder

Left Subtree → Node → Right Subtree
In **BST**, it gives **sorted order** of elements.

```
inorder(node) {
    if (node == NULL) return;
    inorder(node->left);
    visit(node);
    inorder(node->right);
}
```

Inorder: 20 10 30

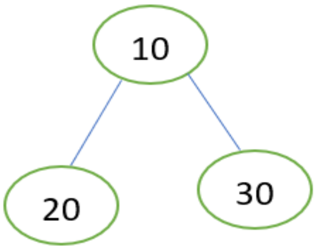


Preorder

Node → Left Subtree → Right Subtree

```
preorder(node) {
    if (node == NULL) return;
    visit(node);
    preorder(node->left);
    preorder(node->right);
}
```

Preorder: 10 20 30

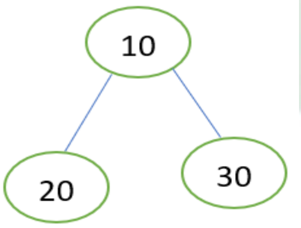


Postorder

Left Subtree → Right Subtree → Node

```
postorder(node) {
    if (node == NULL) return;
    postorder(node->left);
    postorder(node->right);
    visit(node);
}
```

Postorder: 20 30 10



Heap

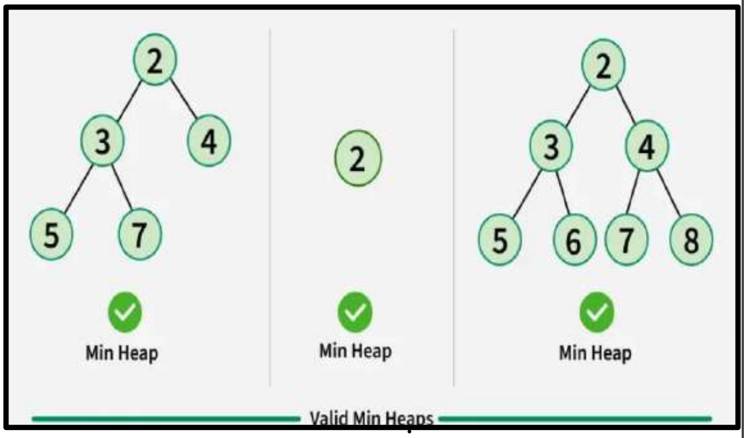
A **Heap** is a special **Complete Binary Tree** where every level is completely filled except possibly the last level, and nodes are as far left as possible.

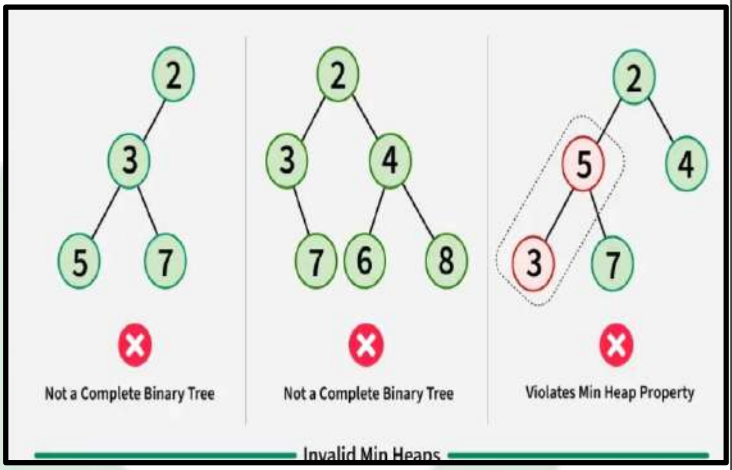
Min Heap

The value of each node is **less than or equal** to its children.

$|a_k|$ < | left tree, right tree|

Root = Minimum element



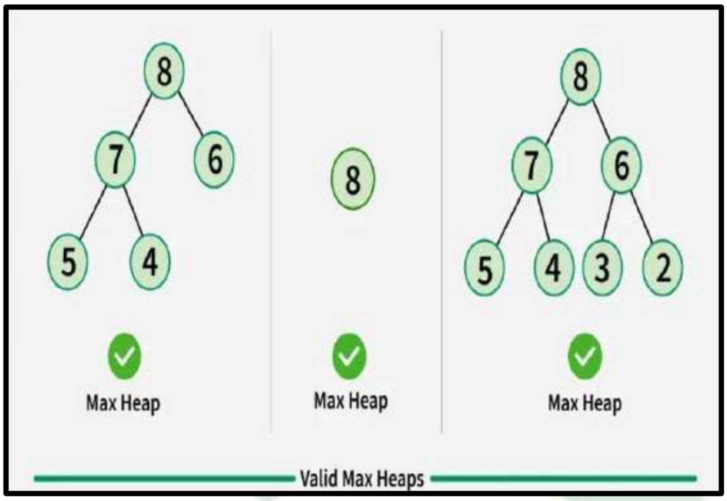


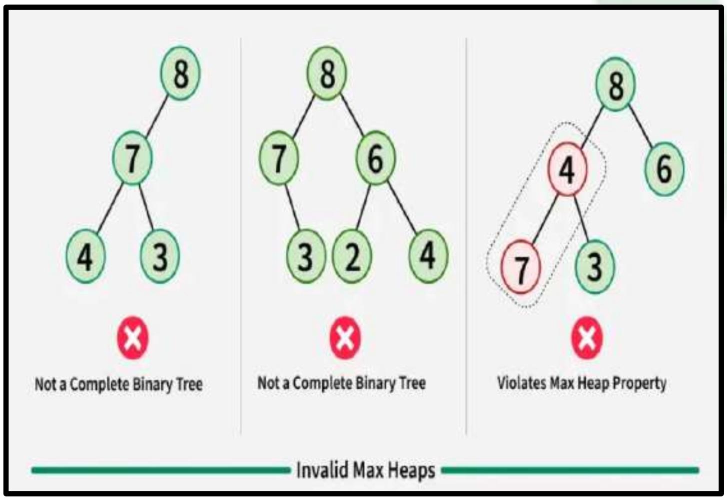
Max Heap

The value of each node is **greater than or equal** to its children.

$|a_k|$ > | left tree, right tree|

Root = Maximum element





Properties:

- Implemented using arrays.
- For node at index i:
  - Left child = 2i + 1
  - Right child = 2i + 2
  - Parent = (i - 1) / 2

Applications:

- Priority Queue
- Heap Sort
- Scheduling algorithms
- Graph algorithms

- Access root (min/max): **O(1)**
- Inserting an element into the heap requires **O(log n)** time.
- If we insert **n** elements one by one, the total time will be **O(n log n)**.
- If we build a heap and then heapify it, the total time complexity is **O(n)**.
- **Deletion** of an element from the heap takes **O(log n)** time.
- **Searching** in a heap takes **O(n)** time, as it is not a sorted structure.
- In a **Min Heap**, the **maximum element** will be found in the **leaf nodes**, so the time complexity to find it is **O(n)**.
- The number of **leaf nodes** in a heap is **ceil(n/2)**.
- The **minimum element** in a **Max Heap** will be found in the leaf nodes, so the time complexity is also **O(n)**.
- **Heap Sort** repeatedly deletes the root element and re-heapifies the remaining heap, so the total time complexity is **O(n log n)**.
- The number of **distinct binary heaps** (Min or Max) with n **distinct elements** is:
  T(n) = T(k). T(n-k-1). $n-1_{C_k}$
  k is the number of the element in left subtree.
  T(n) = number of the heap with n nodes.

Deletion is mostly done at the **root node** (i.e., the max in Max Heap or min in Min Heap).

Steps:

1. **Remove the root node** (i.e., element at index 0 in array representation).
2. **Replace it with the last element** in the heap.
3. **Reduce heap size by 1.**
4. **Heapify (percolate down)** from the root to restore the heap property.

Time Complexity: O(log n)

Heap Sort uses a **Max Heap** (for ascending order sorting).

Steps:

1. **Build a Max Heap** from the input array (takes O(n) time).
2. Repeat until heap size > 1:
   - **Swap** the root (maximum) with the last element.
   - **Reduce the heap size by 1.**
   - **Heapify** the root element to restore the max heap.
3. The array will be sorted in **ascending order**.

Time Complexity:

- Build Heap: O(n)
- Heapify (n times): O(n log n)
- **Overall:** O(n log n)

Heap Sort uses a **Max Heap** (for ascending order sorting).

Steps:

4. **Build a Max Heap** from the input array (takes O(n) time).
5. Repeat until heap size > 1:
   - **Swap** the root (maximum) with the last element.
   - **Reduce the heap size by 1.**
   - **Heapify** the root element to restore the max heap.
6. The array will be sorted in **ascending order**.

Time Complexity:

- Build Heap: O(n)
- Heapify (n times): O(n log n)
- **Overall:** O(n log n)

Graph

A **graph** is a collection of **vertices (nodes)** and **edges (connections)** that represent relationships between pairs of objects.

G= (V, E)

Where:

- V = set of vertices
- E = set of edges (unordered pair for undirected, ordered for directed)

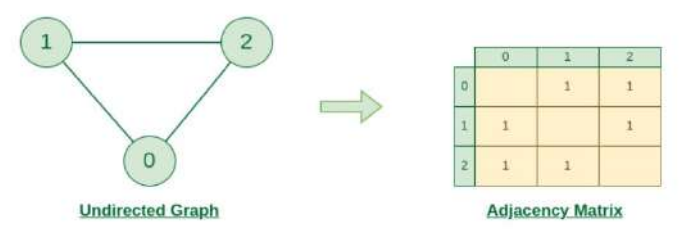
Graph Representations

Adjacency Matrix

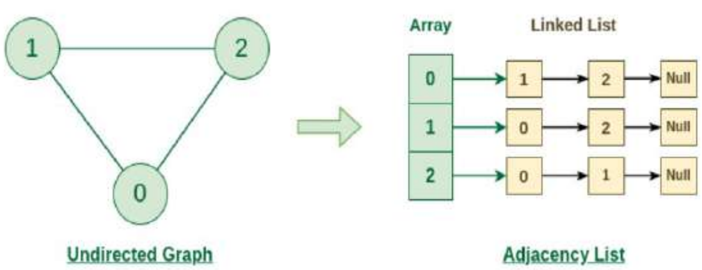
- 2D array G[V][V]
- G[i][j] = 1 if edge exists, else 0
- Space: $O(V^2)$

Adjacency List

- Array of lists
- Each list contains neighbours of a vertex
- Space: O (V + E)
- Preferred for **sparse graphs**



Graph Representation of Undirected graph to Adjacency Matrix



Graph Representation of Undirected graph to Adjacency List

Graph Traversals

Graph traversal refers to visiting all the vertices (and optionally edges) of a graph in a **systematic way**.

- **BFS** (Queue): Level-order, shortest path in unweighted graph
- **DFS** (Stack/Recursion): Deepest first, used in cycle detection, topological sort

Breadth-First Search (BFS)

Level-wise traversal

Uses a **queue (FIFO)** data structure

Visits all immediate neighbours before going deeper

Algorithm

- Mark the starting node as visited
- Enqueue it
- While queue not empty:
- Dequeue node
- Visit all **unvisited neighbours**
- Mark them visited and enqueue

Time Complexity:

- O (V + E) (V = vertices, E = edges)

Applications:

- Shortest path in **unweighted graphs**
- Bipartite graph check
- Connected components in undirected graph

Depth-First Search (DFS)

- **Explores as deep as possible**
- Uses **stack (explicit or recursion)**
- Backtracks when no unvisited neighbours

Algorithm:

1. Mark current node visited
2. Recursively visit all unvisited neighbours

Time Complexity:

- O (V + E)

Applications:

- Topological sorting (DAG)
- Cycle detection
- Strongly Connected Components
- Maze/path solving

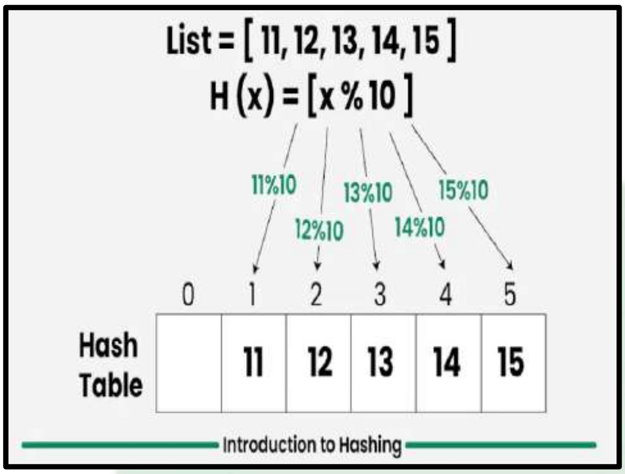
Hashing

Hashing is a technique to **map data of arbitrary size to fixed-size values** using a **hash function**, for efficient **search**, **insert**, and **delete** operations.

Hash Table:

- Stores key-value pairs
- Access time is **O(1)** (average case), **O(n)** (worst case due to collisions)

Properties:



- Should be **fast**, **uniform**, and **deterministic**
- Maps key k to an index:

h(k)=k mod n

where n is the table size

Good n:

- Should be a **prime number** to reduce collisions

Collisions

- **Open Addressing** (Store in same array)

Collision Resolution Techniques

Technique	Rule
Linear Probing	Try next slot: (h(k) + i) % m
Quadratic Probing	$(h(k) + i^2) \% m$
Double Hashing	$(h_1(k) + i \times h_2(k)) \% m$

Linear probing

Problem: Causes **primary clustering**

→ Consecutive blocks of filled slots form, making collisions more frequent

Pros: Simple implementation

Cons: Slower as clustering grows

Quadratic Probing

Solves primary clustering

Problem: May lead to **secondary clustering**

Can fail to insert even when space exists if not carefully designed

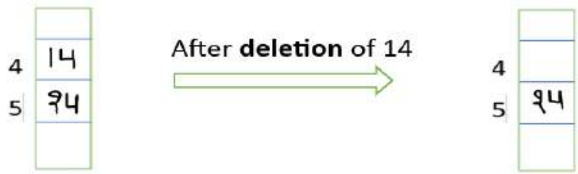
In double hashing h2(k) should not be 0, otherwise it becomes the linear probing.

Deletion Problem in Open Addressing (Linear, Quadratic, Double Hashing)

In **open addressing**, when you delete an element, simply marking the slot as empty can **break the search chain** for other elements inserted due to collisions.

Why It's a Problem:

- Open addressing relies on **probing sequences**.
- In open addressing (e.g., linear probing), suppose n = 10, and we insert 14 and 24. Both hash to index 4, so 14 goes to 4, and 24 goes to 5.
- If we delete 14, index 4 becomes empty. Now, searching for 24 starts at 4 and stops there, thinking it's not present, even though 24 is at index 5.
- This happens because deletion breaks the probing chain, causing search failure.
- This may require the rehashing.
-

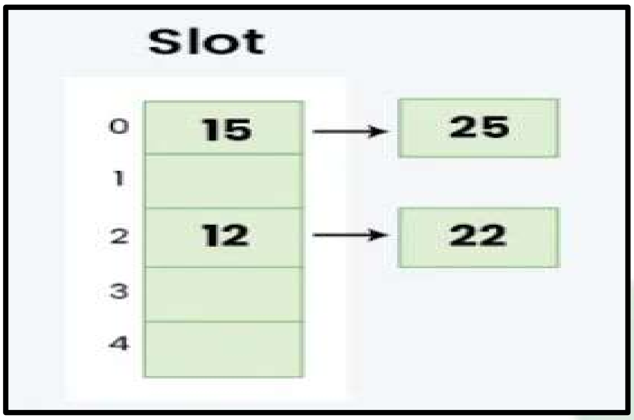


Separate chaining

A collision resolution technique where each slot in the hash table stores a **linked list** (or chain) of elements.

Insertion: Insert the element at the head (or tail) of the linked list at the hashed index.

Search: Hash the key to find the index, then linearly search the linked list at that index.
Deletion: Hash the key, search the linked list, and remove the node if found.
No Clustering: Since elements are in separate lists, primary/secondary clustering does not occur.
Load Factor (λ): $\lambda = n / m$
(n = total elements, m = table size)
Performance depends on λ.



If **load factor $\lambda \leq 1$**, then **open addressing** methods (like linear probing, quadratic probing, or double hashing) are efficient.
If **$\lambda > 1$**, then **separate chaining** is preferred, since open addressing works best only when the table is sparsely filled.

STAR MENTOR CS/DA

KHALEEL SIR
ALGORITHM & OS
29 YEARS OF TEACHING EXPERIENCE

SATISH SIR
DISCRETE MATHEMATICS
BE in IT from MUMBAI UNIVERSITY

VIJAY SIR
DBMS & COA
M. TECH FROM NIT
14+ YEARS EXPERIENCE

SAKSHI MA'AM
ENGINEERING MATHEMATICS
IIT ROORKEE ALUMNUS

AVINASH SIR
APTITUDE
10+ YEARS OF TEACHING EXPERIENCE

CHANDAN SIR
DIGITAL LOGIC
GATE AIR 23 & 26 / EX-ISRO

MALLESHAM SIR
M.TECH FROM IIT BOMBAY
AIR – 114, 119, 210 in GATE
(CRACKED GATE 8 TIMES)
14+ YEARS EXPERIENCE

PARTH SIR
DA
IIIT BANGALORE ALUMNUS
FORMER ASSISTANT PROFESSOR

SHAILENDER SIR
C PROGRAMMING & DATA STRUCTURE
M.TECH in Computer Science
15+ YEARS EXPERIENCE

AJAY SIR
PH.D. IN COMPUTER SCIENCE
12+ YEARS EXPERIENCE