Unit-I

Short Answer Questions:

Q. Write types of Data Structure. (Nov22)
Ans. Data structures are broadly classified into linear (e.g., arrays, stacks, queues) and non-linear (e.g., trees, graphs) types based on data organization.

Q. What are advantages of recursion? (Nov22)
Ans. Recursion simplifies code for problems that have repetitive sub-problems, like tree traversals and factorials, and replaces complex looping logic.

Q. Give concept of strings. (Nov22),(Nov23)
Ans. A string is a sequence of characters terminated by a null character \0 in C/C++; it is used to store and manipulate text data.

Q. What is dynamic memory?(Nov22),(Nov23)
Ans. Dynamic memory allocation allows memory to be allocated at runtime using pointers, enhancing flexibility and efficient memory use.

Q. Flowchart (Nov23),(Nov24)
Ans. A flowchart is a visual representation of an algorithm or process using symbols like arrows, ovals, and rectangles to show flow of control.

Q. Array vs Pointers (Nov24)
Ans. Arrays are fixed-size collections of elements, while pointers are variables that store memory addresses and provide dynamic data access.

Long Answer Questions:

Q. Discuss the limitations of Arrays. Write the steps to find out largest element of an array. (Nov22)
Ans. **Limitations of Arrays:** Arrays, while fundamental in programming, have several limitations. One major limitation is **fixed size**: once an array is declared, its size cannot be changed during runtime. This can lead to inefficient memory use if the array is underutilized or insufficient space if it's overfilled. Arrays also suffer from **inefficient insertion and deletion** operations, especially in the middle of the array, because shifting of elements is required. Furthermore, **only elements of the same data type** can be stored in an array, limiting flexibility. Arrays do not provide built-in methods for dynamic resizing or complex operations, which are available in data structures like lists or vectors.
Steps to Find the Largest Element of an Array:

1. **Initialize** a variable max with the value of the first element of the array.
2. **Traverse** the array from the second element to the last.
3. For each element, compare it with the current value of max.
4. If the current element is greater than max, update max with that value.
5. After the loop ends, max will hold the largest element.

Python Example:
```
arr = [12, 45, 23, 67, 34]
max_val = arr[0]
for num in arr[1:]:
   if num>max_val:
max_val = num
print("Largest element is:", max_val)
```
This approach ensures a simple and efficient way to find the largest element.

Q. a) What do you mean by data structure? Explain complexity of an algorithm. (Nov23)
Q. b) What do you mean by an Array? How a multi-dimensional array differs from one dimensional

array? (Nov23)

Ans. **(a) What do you mean by data structure? Explain complexity of an algorithm.**

A **data structure** is a specialized format for organizing, processing, and storing data efficiently. Common data structures include arrays, stacks, queues, linked lists, trees, and graphs. Each is suited for particular kinds of applications and helps in efficient data management and processing.

The **complexity of an algorithm** refers to the amount of time and/or space resources required for its execution. It is measured in terms of **time complexity** (how fast it runs) and **space complexity** (how much memory it uses). Big O notation (e.g., O(n), O(log n)) is commonly used to describe these complexities, helping to compare the performance of algorithms independently of hardware.

(b) What do you mean by an Array? How a multi-dimensional array differs from one-dimensional array?

An **array** is a collection of elements stored in contiguous memory locations, where each element can be accessed using an index. A **one-dimensional array** is a linear list of elements, while a **multi-dimensional array** (like a 2D array) is an array of arrays, used to represent matrices or tables.

For example:

```
# 1D array
arr = [10, 20, 30]

# 2D array
matrix = [[1, 2], [3, 4]]
```

Multi-dimensional arrays require multiple indices for access, e.g., matrix[1][0] = 3, while 1D arrays use a single index, e.g., arr[2] = 30.

Q. Differentiate between arrays and linked list. (Nov24)

Ans. **Arrays vs Linked Lists**

An **array** is a collection of elements stored in contiguous memory locations. It allows fast access to elements using indexing (e.g., arr[3] directly accesses the 4th element). Arrays are of fixed size, meaning the size must be defined at the time of declaration and cannot be changed dynamically. They offer efficient read operations (O(1)) but are inefficient for insertions or deletions in the middle, as shifting is required.

A **linked list**, on the other hand, is a dynamic data structure consisting of nodes. Each node contains data and a pointer (or reference) to the next node. Linked lists do not require contiguous memory and can grow or shrink at runtime, making them flexible in memory usage. However, accessing elements is slower (O(n)) since traversal is needed from the head to the desired node.

Key Differences:

- **Memory Allocation:** Arrays use contiguous memory; linked lists use non-contiguous memory.
- **Size:** Arrays have a fixed size; linked lists can grow/shrink dynamically.
- **Access Time:** Arrays allow random access; linked lists require sequential access.
- **Insertion/Deletion:** Expensive in arrays due to shifting; efficient in linked lists using pointers.
- **Memory Usage:** Arrays may waste space; linked lists use extra memory for pointers.

In summary, arrays are preferred for fast access and fixed-size data, while linked lists are ideal for dynamic data with frequent insertions or deletions.

Unit-II
Short Answer Questions:
Q. Discuss Circular Queue. (Nov22)
Ans. A circular queue connects the rear end to the front, allowing efficient utilization of space by reusing freed-up memory positions.

Q. Multiple stack (Nov23)
Ans. Multiple stacks use a single array to implement two or more stacks, optimizing memory in stack-based applications.

Q. Priority queue (Nov23)
Ans. A priority queue arranges elements based on priority rather than insertion order, where higher-priority elements are served before lower ones.

Q. De-queue (Nov24)
Ans. A Dequeue (Double-Ended Queue) allows insertion and deletion of elements from both the front and rear ends of the queue.

Long Answer Questions:

Q. Define Stack. What operations are performed on a stack? Write applications of a stack. (Nov22), (Nov24), (Nov23)
Ans. A **Stack** is a linear data structure that follows the **Last-In, First-Out (LIFO)** principle. This means the last element added (pushed) to the stack is the first one to be removed (popped). It can be visualized like a pile of plates—only the top plate can be accessed or removed at a time.
Operations Performed on a Stack:
1. **Push:** Adds an element to the top of the stack.
2. **Pop:** Removes the top element from the stack.
3. **Peek/Top:** Returns the top element without removing it.
4. **IsEmpty:** Checks if the stack is empty.
5. **IsFull:** Checks if the stack is full (in case of fixed size).

Applications of Stack:
1. **Expression Evaluation and Conversion:** Used in evaluating arithmetic expressions (postfix, prefix) and converting between notations.
2. **Function Call Management:** Manages function calls and returns in programming via the call stack.
3. **Undo Mechanism:** Used in applications like text editors to store history for undo/redo functionality.
4. **Syntax Parsing:** Used in compilers to parse code and check for balanced symbols like parentheses.
5. **Backtracking Algorithms:** Helps in solving mazes, puzzles, and pathfinding by storing previous states.

Stacks are fundamental in both software applications and system-level programming.

Q. What is Queue? Write an algorithm to insert an element from a simple Queue. (Nov22)
Ans. A **Queue** is a linear data structure that follows the **First-In, First-Out (FIFO)** principle. This means the element inserted first is the one to be removed first, similar to a line of people waiting for a service.
Basic Queue Operations:
1. **Enqueue (Insert):** Adds an element to the rear of the queue.
2. **Dequeue (Delete):** Removes an element from the front of the queue.
3. **IsEmpty:** Checks if the queue is empty.
4. **IsFull:** Checks if the queue is full (in case of a fixed-size array).

5. **Peek/Front:** Returns the front element without removing it.

Algorithm to Insert (Enqueue) an Element in a Simple Queue:
Assumptions:

- QUEUE[] is an array of size MAX.
- FRONT and REAR are pointers initialized to -1.

```
Algorithm Enqueue(QUEUE, ITEM)
1. IF REAR == MAX - 1
2.     PRINT "Queue Overflow"
3.     EXIT
4. ELSE IF FRONT == -1 AND REAR == -1
5.     SET FRONT ← 0, REAR ← 0
6. ELSE
7.     SET REAR ← REAR + 1
8. SET QUEUE[REAR] ← ITEM
9. PRINT "Item Inserted Successfully"
```

This algorithm checks for overflow and inserts the element at the correct position. Queues are widely used in scheduling, buffering, and resource management.

Q. Write algorithm to insert and delete a node in circular queue. (Nov24)
Ans. A **Circular Queue** is a linear data structure that connects the end of the queue back to the front, forming a circle. It efficiently utilizes memory by reusing empty spaces left by deletions. It uses two pointers: **FRONT** and **REAR**.

Algorithm to Insert (Enqueue) in Circular Queue

```
Algorithm Enqueue(CQ, ITEM)
1. IF (FRONT == 0 AND REAR == MAX-1) OR (FRONT == REAR + 1)
2.     PRINT "Queue Overflow"
3.     EXIT
4. IF FRONT == -1
5.     SET FRONT ← 0, REAR ← 0
6. ELSE IF REAR == MAX-1
7.     SET REAR ← 0
8. ELSE
9.     SET REAR ← REAR + 1
10. SET CQ[REAR] ← ITEM
11. PRINT "Item Inserted"
```

Algorithm to Delete (Dequeue) in Circular Queue

```
Algorithm Dequeue(CQ)
1. IF FRONT == -1
2.     PRINT "Queue Underflow"
3.     EXIT
4. SET ITEM ← CQ[FRONT]
5. IF FRONT == REAR
6.     SET FRONT ← -1, REAR ← -1
7. ELSE IF FRONT == MAX-1
8.     SET FRONT ← 0
9. ELSE
10.    SET FRONT ← FRONT + 1
11. PRINT "Deleted ITEM"
```

These algorithms ensure efficient circular movement of FRONT and REAR pointers, avoiding wasted space. Circular queues are commonly used in memory management, buffering, and scheduling tasks.

Q. How to convert in-fix notation into post-fix notation? (Nov24)

Ans. **Infix to Postfix Conversion** involves rearranging an arithmetic expression so that **operators come after operands**, removing the need for parentheses and respecting operator precedence. Postfix notation (also called **Reverse Polish Notation**) is ideal for computation using stacks.

Steps to Convert Infix to Postfix:

1. **Initialize an empty stack** for operators and an empty list for the output.
2. **Scan the infix expression** from left to right.
3. **If the symbol is an operand**, add it directly to the output.
4. **If the symbol is an operator**:
   - While the stack is not empty and the **top of the stack has higher or equal precedence**, **pop from the stack and add to output**.
   - Push the current operator onto the stack.
5. **If the symbol is '('**, push it onto the stack.
6. **If the symbol is ')'**, pop and output from the stack until '(' is encountered. Discard both parentheses.
7. **After the expression ends**, pop and output all remaining operators in the stack.

Example:

Infix: A+B*C

Postfix: ABC*+ +

Explanation: Multiplication has higher precedence, so it comes before addition.

This method ensures that the expression can be evaluated using a stack without ambiguity.

Unit-III
Short Answer Questions:
Q. Discuss AVL trees. (Nov22) (Nov23) (Nov24)
Ans. An AVL tree is a self-balancing binary search tree where the height difference of left and right subtrees (balance factor) is at most one.

Q. Circular linked list (Nov23)
Ans. In a circular linked list, the last node points back to the first node, forming a circle, which allows continuous traversal from any point.

Q. Out-degree (Nov24)
Ans. The out-degree of a vertex in a graph is the number of edges leaving that vertex, representing direct connections to other vertices.

Q. Complete binary tree (Nov24)
Ans. A complete binary tree is a binary tree in which all levels are completely filled except possibly the last, which is filled from left to right.

Long Answer Questions:

Q. What is linked list? Discuss the various operations on linked list. How single linked list is different from doubly linked list? (Nov24),(Nov23)
Ans. A **linked list** is a linear data structure where each element, called a **node**, contains two parts: the data and a pointer (or reference) to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory locations, making them efficient for dynamic memory allocation.
Operations on Linked List:
1. **Insertion** – Add a new node at the beginning, end, or a specified position.
2. **Deletion** – Remove a node from the beginning, end, or a specified position.
3. **Traversal** – Visit each node in the list to display or process data.
4. **Searching** – Find whether a particular value exists in the list.
5. **Updation** – Modify the data in a specific node.

Singly Linked List vs Doubly Linked List:

Feature	**Singly Linked List**	**Doubly Linked List**
Pointers per node	One (next)	Two (next and previous)
Navigation	One direction (forward only)	Two directions (forward and backward)
Memory usage	Less (one pointer)	More (two pointers per node)
Insertion/Deletion	Less efficient (no backward pointer)	More efficient in both directions

In summary, linked lists are versatile for dynamic data structures. **Singly linked lists** are simpler and use less memory, while **doubly linked lists** offer easier navigation and flexibility at the cost of extra memory.

Q. What is Doubly Linked List? Write an algorithm to insert and delete a node in Doubly Linked List. (Nov22)
Ans. A **Doubly Linked List (DLL)** is a type of linked list where each node contains **three parts**:
1. Data – The actual value stored.
2. Prev – Pointer to the previous node.
3. Next – Pointer to the next node.

This structure allows traversal in **both directions**, making insertion and deletion operations more flexible compared to a singly linked list.

Algorithm to Insert a Node at the End:
Step 1: Create newNode

```
Step 2: newNode.data = value
Step 3: newNode.next = NULL
Step 4: If head is NULL:
        head = newNode
newNode.prev = NULL
      Else:
        temp = head
        While temp.next != NULL:
           temp = temp.next
temp.next = newNode
newNode.prev = temp
```

Algorithm to Delete a Node from the End:

```
Step 1: If head == NULL:
        Print "List is empty"
      Else if head.next == NULL:
        Free head
        head = NULL
      Else:
        temp = head
        While temp.next != NULL:
           temp = temp.next
temp.prev.next = NULL
        Free temp
```

Use Case: DLL is ideal when frequent insertions/deletions at both ends are required. Its two-way traversal advantage outweighs the extra memory used by the additional pointer.

Q. Explain Inorder, Preorder and Postorder Traversal operation. (Nov22)

Ans. In binary trees, **traversal** refers to visiting each node in a specific order. The three main depth-first traversal methods are:

1. Inorder Traversal (Left, Root, Right)

- Visit the **left subtree**, then **root**, then **right subtree**.
- **Used to retrieve nodes in sorted order** for Binary Search Trees (BST).

Example:

For the tree:

```
    A
   / \
  B   C
```

Inorder: B → A → C

2. Preorder Traversal (Root, Left, Right)

- Visit the **root**, then **left subtree**, then **right subtree**.
- **Used to create a copy** of the tree or prefix notation.

Example:

Preorder: A → B → C

3. Postorder Traversal (Left, Right, Root)

- Visit the **left subtree**, then **right subtree**, then **root**.
- **Useful for deleting** a tree or postfix expression.

Example:

Postorder: B → C → A

Recursive Algorithm (Generic):

```
def inorder(node):
   if node:
inorder(node.left)
      print(node.data)
inorder(node.right)
```

Each traversal method serves different purposes in applications like expression trees, tree reconstruction, and compiler design.

Q. What is a binary tree? How it is traversed? How a binary search tree is different from a binary tree? (Nov22),(Nov23)

Ans. A **binary tree** is a hierarchical data structure in which each node has at most **two children**, referred to as the **left** and **right** child. The topmost node is called the **root**, and the nodes without children are called **leaves**. Binary trees are widely used for searching, sorting, and hierarchical data representation.

Tree Traversal Methods:

Traversal means visiting all nodes in a specific order:

1. **Inorder (Left → Root → Right)** – Retrieves nodes in ascending order for Binary Search Trees.
2. **Preorder (Root → Left → Right)** – Useful for tree copying or prefix expression.
3. **Postorder (Left → Right → Root)** – Useful for deleting or evaluating expressions.
4. **Level-order (Breadth-first)** – Visits nodes level by level using a queue.

Binary Tree vs Binary Search Tree (BST):

Feature	**Binary Tree**	**Binary Search Tree (BST)**
Node Order	No specific order among nodes	Left < Root < Right
Purpose	General tree representation	Efficient searching and sorting
Duplicate Values	Allowed	Generally not allowed (varies by implementation)
Time Complexity	Not optimized for search	O(log n) for balanced BST

A BST is a **specialized binary tree** that maintains an ordered structure, enabling faster search, insertion, and deletion operations.

Q. What is BST? Explain its traversals with an example. (Nov24)

Ans. A **Binary Search Tree (BST)** is a special type of binary tree where each node contains a key, and it follows a specific ordering property:

- All keys in the **left subtree** are **less than** the root node's key.
- All keys in the **right subtree** are **greater than** the root node's key.
- This property holds true **recursively** for all nodes in the tree.

BSTs allow **efficient searching, insertion, and deletion**, typically in **O(log n)** time for balanced trees.

BST Traversals:

1. **Inorder Traversal (Left → Root → Right)**
   - ○ Produces the elements in **sorted ascending order**.
   - ○ Example:

```
  50
 / \
30   70
/\  /\
    20 40 60 80
```

InorderOutput: 20, 30, 40, 50, 60, 70, 80

2. **Preorder Traversal (Root → Left → Right)**
   - ○ Used to create a **copy of the tree** or for prefix expression.

3. **Postorder Traversal (Left → Right → Root)**
    - ○ Useful for **deleting nodes** or evaluating postfix expressions.

Each traversal serves a unique purpose, and the choice depends on the operation being performed (e.g., printing, copying, or evaluating the tree structure).

Q. Explain Inorder, Preorder and Postorder Traversal operation. (Nov22)
Ans. **Inorder, Preorder, and Postorder** are three fundamental types of **depth-first traversal techniques** used in binary trees to visit every node in a specific sequence.

1. Inorder Traversal (Left → Root → Right):
- This traversal visits the **left subtree first**, then the **root**, and finally the **right subtree**.
- In a **Binary Search Tree (BST)**, inorder traversal returns the nodes in **ascending sorted order**.
- **Example:**
  For a tree:

```
  10
 / \
5  15
```

Output: 5, 10, 15

2. Preorder Traversal (Root → Left → Right):
- This traversal visits the **root first**, followed by the **left** and then the **right subtree**.
- It is useful for **creating a copy** of the tree or generating prefix expressions.
- **Example Output:** 10, 5, 15

3. Postorder Traversal (Left → Right → Root):
- This traversal visits the **left subtree**, then the **right subtree**, and finally the **root**.
- Often used to **delete a tree** from bottom up or evaluate postfix expressions.
- **Example Output:** 5, 15, 10

Each traversal method provides a unique way to process the nodes of a binary tree depending on the use case (e.g., sorting, copying, or deletion).

Unit-IV
Short Answer Questions:
Q. Explain selection sort. (Nov22)
Ans. Selection sort repeatedly selects the smallest (or largest) element from the unsorted portion and places it at the correct position in the sorted portion.

Q. Write concept of Hash Function. (Nov22)
Ans. A hash function maps data of arbitrary size to fixed-size values, used for efficient data retrieval in hash tables.

Q. Graph (Nov23)
Ans. A graph is a non-linear data structure consisting of nodes (vertices) connected by edges, used to represent networks.

Q. Quick sort (Nov23)
Ans. Quick sort is a divide-and-conquer algorithm that partitions the array around a pivot and recursively sorts the subarrays.

Q. Big 'O' Notation (Nov24)
Ans. Big O notation describes the upper bound of an algorithm’s runtime or space complexity, indicating the worst-case performance.

Q. Recursion,(Nov24)
Ans. Recursion is a method where a function calls itself to solve a problem by breaking it into smaller subproblems.

Long Answer Questions:

Q. Explain depth first search and breadth first search in graphs.(Nov22) (Nov24)
Ans. **Depth First Search (DFS)** and **Breadth First Search (BFS)** are two fundamental algorithms used for traversing or searching in graphs and trees.

Depth First Search (DFS):
- DFS explores as far as possible along each branch before backtracking.
- It uses a **stack** data structure (either explicitly or via recursion).
- Starting from a source node, DFS visits a node, marks it as visited, and recursively visits its unvisited adjacent nodes.
- DFS is useful for detecting cycles, pathfinding, and solving puzzles like mazes.
- **Example Order:** If A is connected to B and C, and B to D, it will visit: A → B → D → C

Breadth First Search (BFS):
- BFS explores all neighbors of a node before moving to the next level of nodes.
- It uses a **queue** data structure.
- Starting from a source node, BFS visits all its adjacent nodes, then their adjacent nodes, and so on.
- BFS is ideal for finding the shortest path in an unweighted graph.
- **Example Order:** A → B → C → D

Comparison:
- **DFS:** Deep exploration, uses stack, can go deep before exploring siblings.
- **BFS:** Level-by-level exploration, uses queue, good for shortest path.

Both algorithms have time complexity **O(V + E)**, where V is vertices and E is edges.

Q. Perform bubble and selection sort on given array: (Nov24)

43, 57, 66, 11, 9, 96, 23, 37, 72, 88

Ans. Let’s perform **Bubble Sort** and **Selection Sort** step-by-step on the array:
Given Array:
43, 57, 66, 11, 9, 96, 23, 37, 72, 88

Bubble Sort (Ascending Order):
Pass 1:
43, **57**, 66, 11, 9, 96, 23, 37, 72, 88
→ No swap (57 < 66)
43, 57, **66**, 11, 9, 96, 23, 37, 72, 88
→ Swap 66 & 11 → 43, 57, 11, 66...
→ Continue swapping...
After Pass 1:
43, 11, 9, 57, 66, 23, 37, 72, 88, 96

Pass 2:
11, 9, 43, 23, 37, 57, 66, 72, 88, 96

Pass 3:
9, 11, 23, 37, 43, 57, 66, 72, 88, 96
Now sorted.

Final Bubble Sort Result:
✅ 9, 11, 23, 37, 43, 57, 66, 72, 88, 96

✅ **Selection Sort (Ascending Order):**
Step 1: Find min (9) → Swap with 43
9, 57, 66, 11, 43, 96, 23, 37, 72, 88
Step 2: Find next min (11) → Swap with 57
9, 11, 66, 57, 43, 96, 23, 37, 72, 88
Continue...

Final Selection Sort Result:
✅ 9, 11, 23, 37, 43, 57, 66, 72, 88, 96

✅**Both sorting methods result in:**
9, 11, 23, 37, 43, 57, 66, 72, 88, 96

Q. Write an algorithm to sort an array of integers in the descending order using quick sort. (Nov22)
Ans. **Quick Sort Algorithm for Sorting in Descending Order**
Quick sort is a divide-and-conquer algorithm. To sort an array in **descending order**, we simply reverse the comparison in the partition step.

Algorithm: QuickSortDescending(arr, low, high)

```
1. If low < high:
    o Partition the array using partitionDescending(arr, low, high)
    o Let pi be the partition index returned
    o Recursively sort the sub-arrays:
        ▪ QuickSortDescending(arr, low, pi - 1)
        ▪ QuickSortDescending(arr, pi + 1, high)
```

partitionDescending(arr, low, high)

```
1. Set pivot = arr[high]
```

```
2. Initialize i = low - 1
3. For j = low to high - 1:
      o If arr[j] > pivot: (Note: Greater than for descending)
            ▪ i = i + 1
            ▪ Swap arr[i] and arr[j]
4. Swap arr[i + 1] and arr[high]
5. Return i + 1 (Partition index)
```

Example:
Given arr = [34, 7, 23, 32, 5]
After applying quick sort in descending order:
Output: [34, 32, 23, 7, 5]

Time Complexity:

- Best/Average: O(n log n)
- Worst: $O(n^2)$

Quick sort is preferred for its efficiency and in-place sorting.

Q. Write short notes on following: (Nov23)
a) Adjacency matrix
b) Linear search
Ans. **a) Adjacency Matrix:** An adjacency matrix is a 2D array used to represent a finite graph. For a graph with **n vertices**, it uses an **n × n** matrix where each cell **(i, j)** indicates the presence (usually marked as 1) or absence (marked as 0) of an edge between vertex **i** and vertex **j**. For **weighted graphs**, the matrix holds the weight of the edge instead of just 1. It is widely used for **dense graphs** due to its quick lookup time, but it is space-inefficient for sparse graphs, as it uses **$O(n^2)$** space regardless of the number of edges.

b) Linear Search: Linear search is a simple searching algorithm used to find an element in a list or array. It checks each element of the list sequentially until the target element is found or the end of the list is reached. It works on both sorted and unsorted arrays and is easy to implement. However, its time complexity is **O(n)** in the worst case, making it inefficient for large datasets. Linear search is suitable when the list is small or unsorted, and no additional data structures are available to improve search efficiency.

Q. Explain the concept of Hashing and Hashing function. What are various collision resolution techniques? (Nov23),(Nov23)
Ans. **Hashing** is a technique used to map data of arbitrary size to fixed-size values, known as **hash values** or **hash codes**. This process allows for **quick data retrieval**, especially in **hash tables**, where each data element is stored at a unique index determined by a **hash function**. Hashing is widely used in applications such as **databases**, **symbol tables**, and **caching** systems.
A **hash function** takes an input (or key) and returns an integer value, which determines the index in the hash table. A good hash function distributes the keys uniformly and minimizes collisions. For example, a simple hash function could be:
hash(key) = key % table_size

Collision Resolution Techniques:
Since different keys can hash to the same index, **collisions** must be handled effectively:

1. **Chaining**: Uses a linked list to store multiple elements at the same hash index.
2. **Open Addressing**: Finds another open slot in the hash table when a collision occurs.
    - **Linear Probing**: Checks the next slot sequentially.
    - **Quadratic Probing**: Jumps in quadratic increments.
    - **Double Hashing**: Uses a second hash function to decide the jump.
3. **Rehashing**: Expands the table size and recalculates all positions using a new hash function.

Effective collision resolution ensures efficient lookup, insertion, and deletion in hash-based data structures.