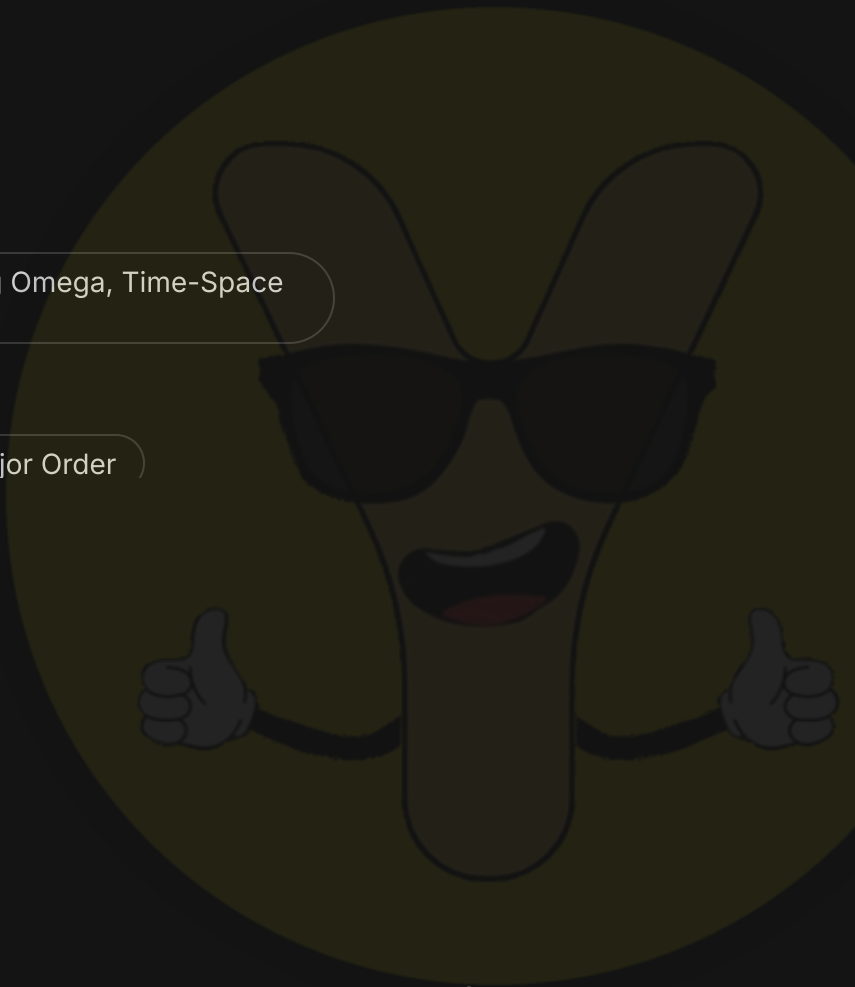
AKTU · 2ND YEAR · SEMESTER 3 · BCS301

Data Structure

Unit 1 — Introduction, Arrays, and Linked Lists

- Introduction: Basic Terminology, Elementary Data Organization, Built in Data Types in C
- Algorithm, Efficiency of an Algorithm, Time and Space Complexity, Asymptotic notations: Big Oh, Big Theta and Big Omega, Time-Space trade-off
- Abstract Data Types (ADT)
- Arrays: Definition, Single and Multidimensional Arrays, Representation of Arrays: Row Major Order, and Column Major Order



DECODED BY

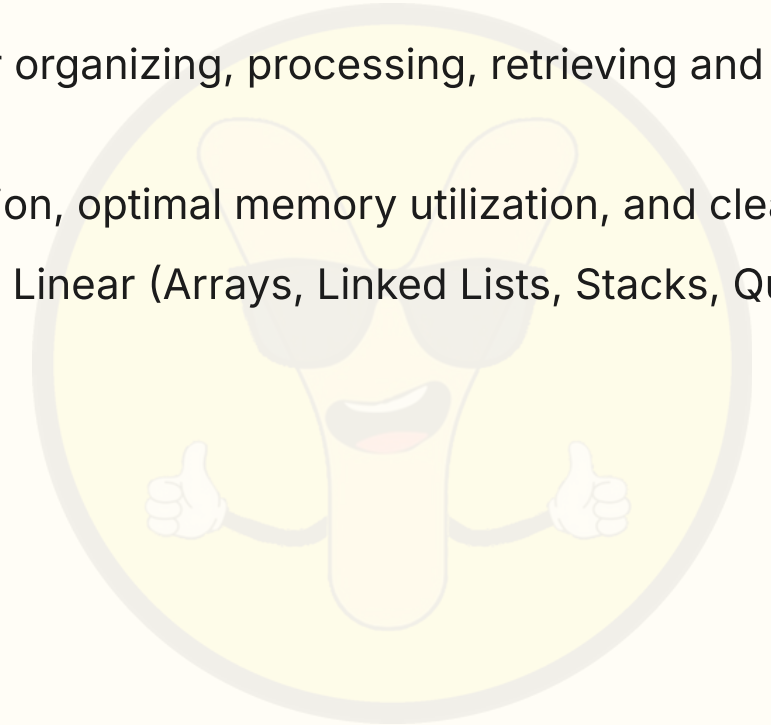
Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

Course Introduction: Data Structures

Fundamental building blocks of efficient software development and algorithm design.

- Definition: A specialized format for organizing, processing, retrieving and storing data in a computer memory.
- Importance: Enables faster execution, optimal memory utilization, and clean software architecture.
- Classification: Broadly divided into Linear (Arrays, Linked Lists, Stacks, Queues) and Non-Linear (Trees, Graphs).

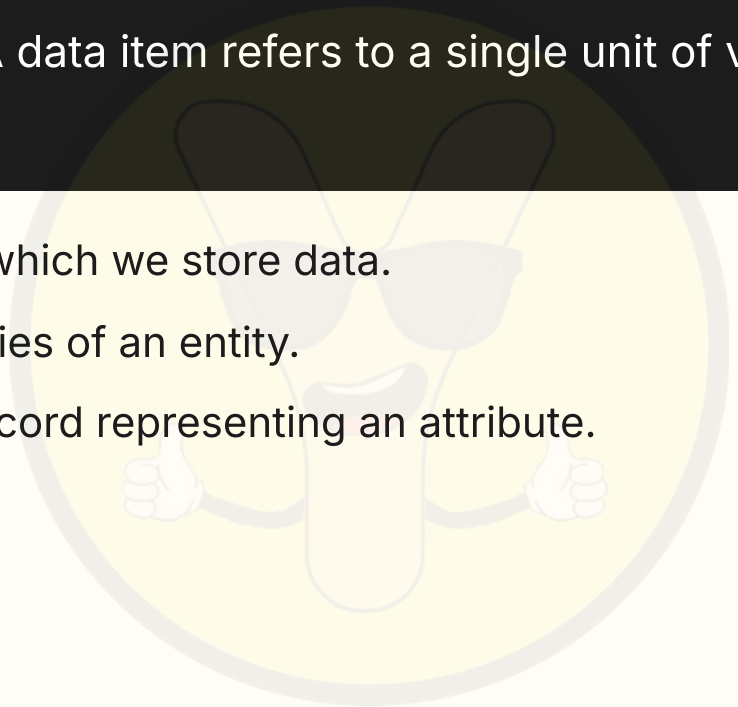


Basic Terminology

DATA, DATA ITEM, AND RECORD

Data are raw, unorganized facts. A data item refers to a single unit of value. A record is a collection of related data items.

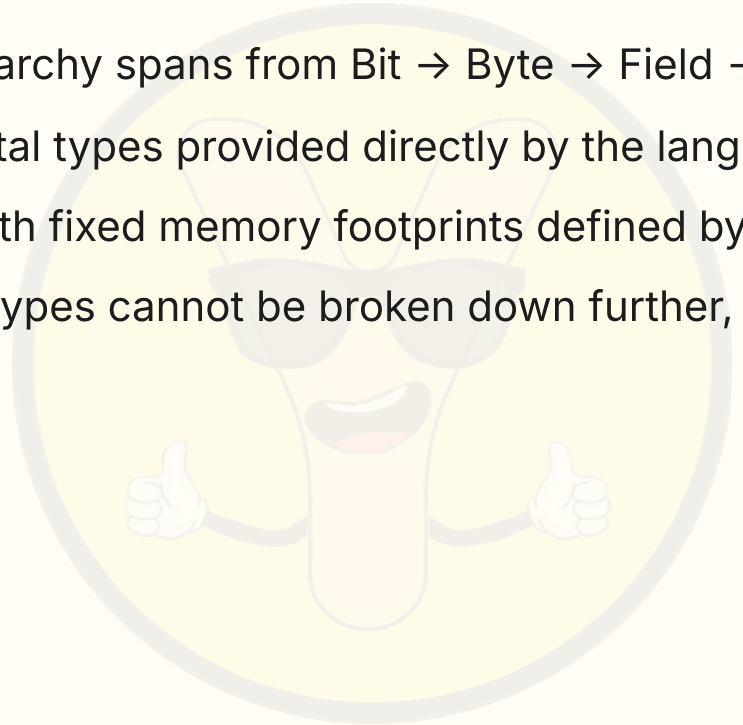
- Entity: An object or concept about which we store data.
- Attribute: Characteristics or properties of an entity.
- Field: Single elementary unit of a record representing an attribute.



Elementary Data Organization & Built-in Types

How data is arranged from primitive bits up to complex structures.

- Elementary Data Organization: Hierarchy spans from Bit → Byte → Field → Record → File → Database.
- Built-in Data Types in C: Fundamental types provided directly by the language compiler.
- Examples: int, float, char, double with fixed memory footprints defined by system architecture.
- Primitive vs Non-Primitive: Built-in types cannot be broken down further, whereas data structures are composed of primitive types.

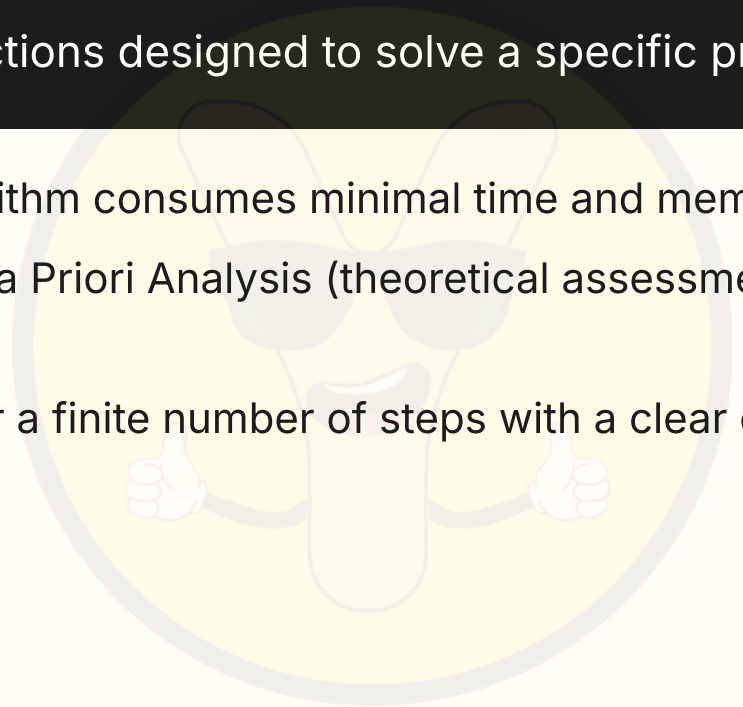


Algorithm and Efficiency

ALGORITHM

A step-by-step finite set of instructions designed to solve a specific problem.

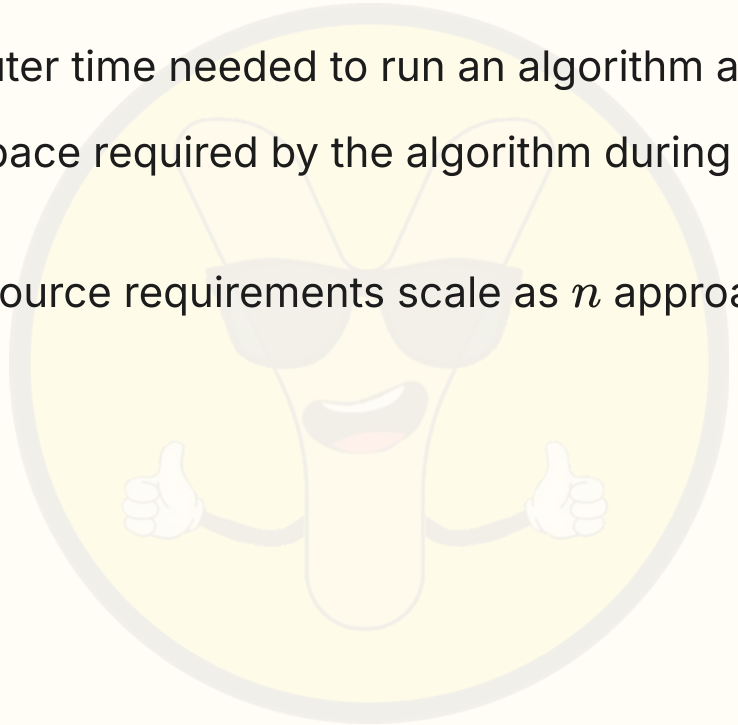
- Efficiency Criteria: An optimal algorithm consumes minimal time and memory space.
- Performance Analysis: Evaluated via Priori Analysis (theoretical assessment independent of hardware) and Posteriori Testing.
- Key constraint: Must terminate after a finite number of steps with a clear output.



Time and Space Complexity

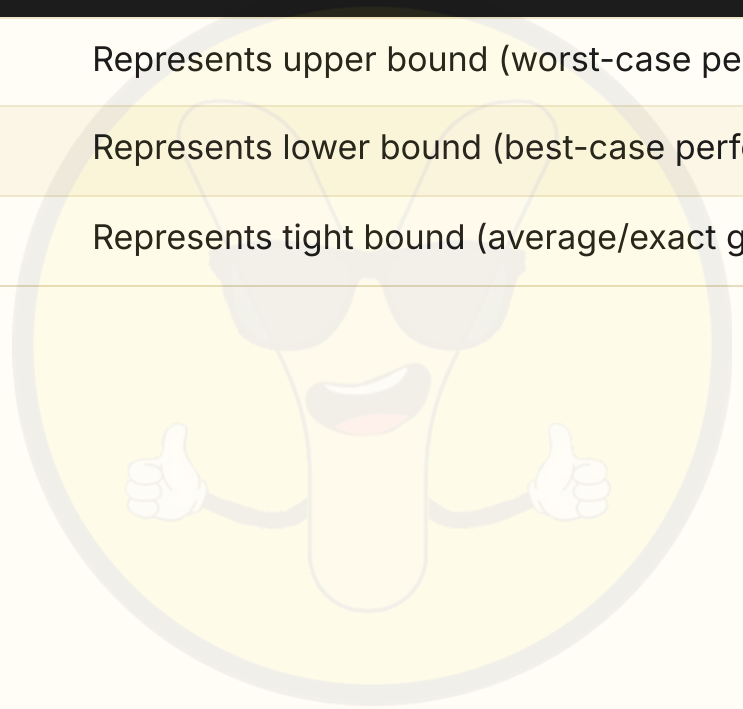
Quantifying resource utilization of an algorithm.

- Time Complexity: Amount of computer time needed to run an algorithm as a function of input size n.
- Space Complexity: Total memory space required by the algorithm during its execution, including inputs and auxiliary variables.
- Order of Growth: Evaluates how resource requirements scale as n approaches infinity, dropping constants and lower-order terms.



Asymptotic Notations

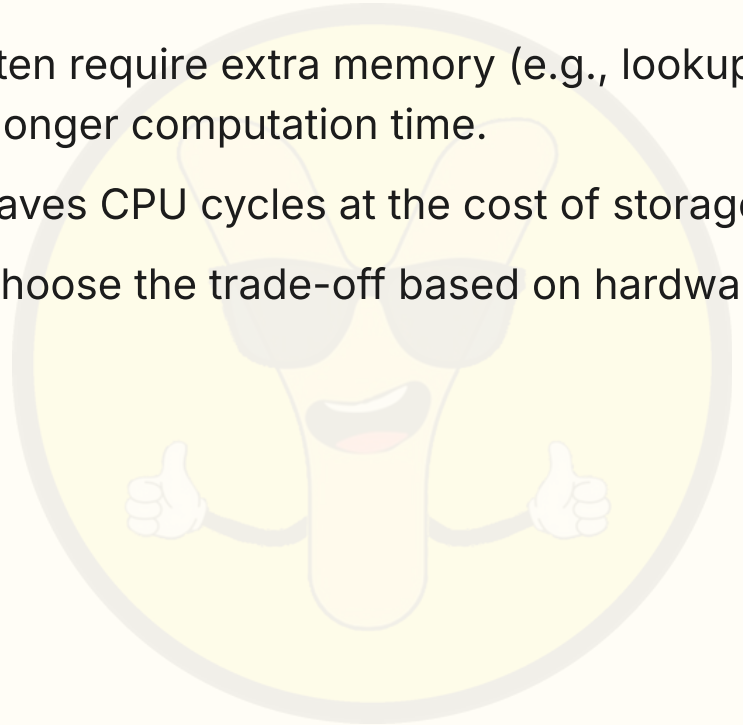
Notation	Mathematical Name	Role in Analysis
$O(g(n))$	Big Oh	Represents upper bound (worst-case performance analysis)
$\Omega(g(n))$	Big Omega	Represents lower bound (best-case performance analysis)
$\Theta(g(n))$	Big Theta	Represents tight bound (average/exact growth rate)



Time-Space Trade-Off

Balancing execution speed against memory consumption.

- Core Dilemma: Faster algorithms often require extra memory (e.g., lookup tables, memoization), while memory-efficient approaches take longer computation time.
- Example: Precomputing factorials saves CPU cycles at the cost of storage.
- Engineering Decision: Developers choose the trade-off based on hardware constraints and specific application requirements.

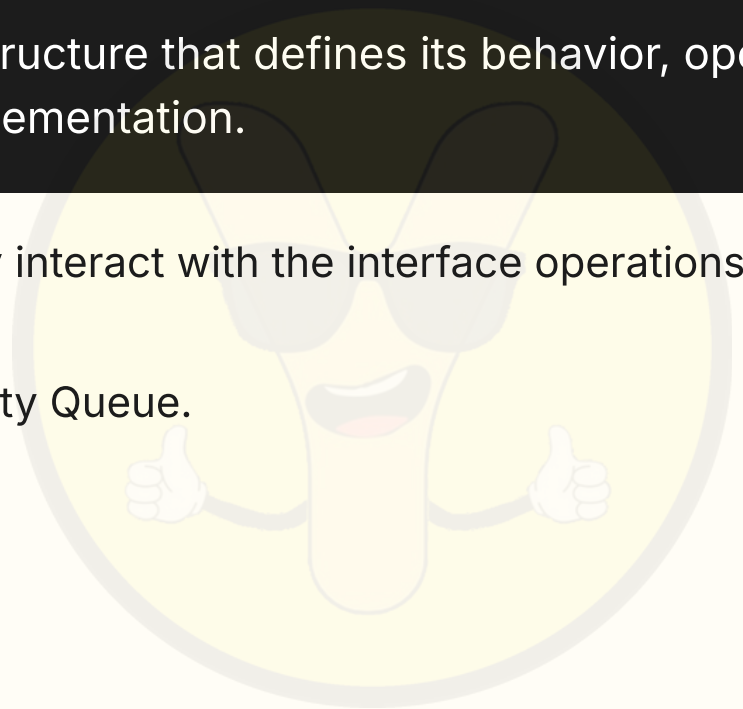


Abstract Data Type (ADT)

ABSTRACT DATA TYPE

A mathematical model of a data structure that defines its behavior, operations, and constraints without specifying its internal implementation.

- Separation of Concerns: Users only interact with the interface operations (e.g., push, pop), ignoring underlying memory mechanics.
- Examples: Stack, Queue, List, Priority Queue.

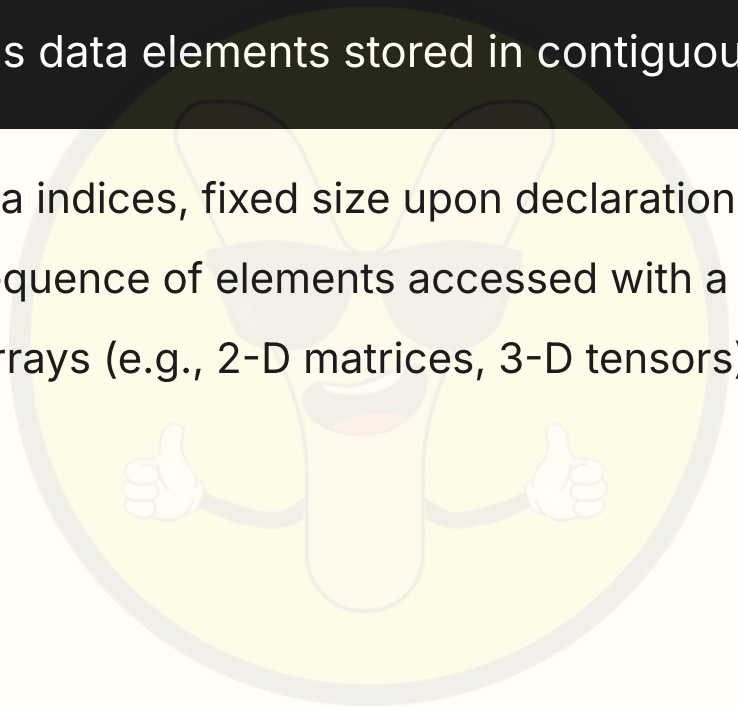


Arrays: Definition and Representation

ARRAY

A collection of finite, homogeneous data elements stored in contiguous memory locations.

- Properties: Direct random access via indices, fixed size upon declaration.
- Single-Dimensional Array: Linear sequence of elements accessed with a single index.
- Multidimensional Array: Arrays of arrays (e.g., 2-D matrices, 3-D tensors).



Array Memory Mapping: Row Major & Column Major

How multi-dimensional arrays are flattened into linear RAM addresses.

- Row Major Order: Elements of a row are stored contiguously in memory. Used by C, C++, and Java.
- Column Major Order: Elements of a column are stored contiguously in memory. Used by Fortran and MATLAB.
- Base Address B: Starting memory address of the array, denoted as \(&A[0]\).

Index Formula Derivation: 1-D and 2-D Arrays

Derive address formulas for 1-D and 2-D Row Major arrays given base address B and element size W.

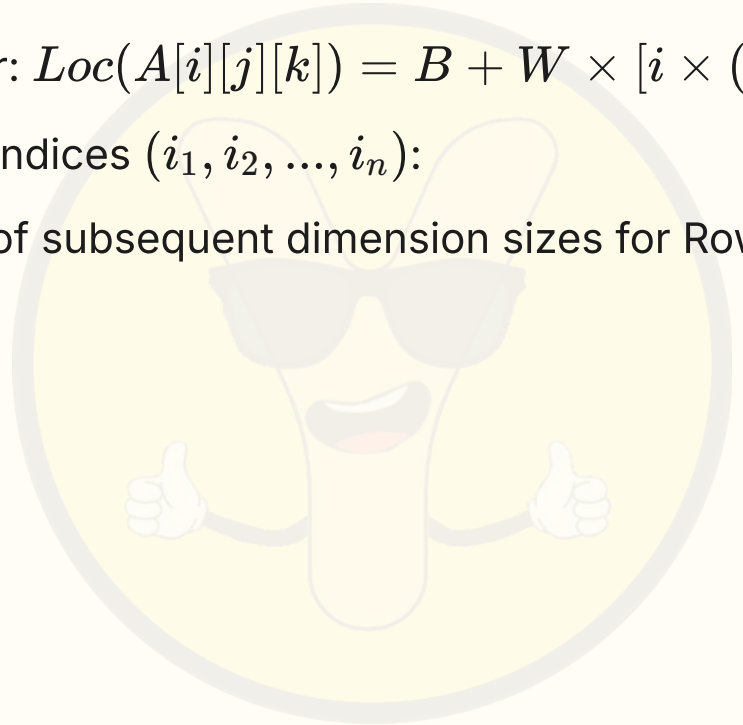
1. 1-D Array Address: $Loc(A[i]) = B + W \times (i - lower_bound)$
2. 2-D Row Major $A[R][C]$: $Loc(A[i][j]) = B + W \times [(i - r_0) \times \text{Total Columns} + (j - c_0)]$
3. Assuming zero-indexed $(r_0 = 0, c_0 = 0)$: $Loc(A[i][j]) = B + W \times (i \times N + j)$ where N is number of columns.

Formulas allow $O(1)$ direct calculation of any element memory location.

Index Formulae for 3-D and n-D Arrays

Extending mapping principles to higher dimensions.

- 3-D Array $A[L][R][C]$ in Row Major: $Loc(A[i][j][k]) = B + W \times [i \times (R \times C) + j \times C + k]$
- General n-D Array $A[d_1][d_2]...[d_n]$ indices $(i_1, i_2, ..., i_n)$:
- Formula uses cumulative products of subsequent dimension sizes for Row Major order.

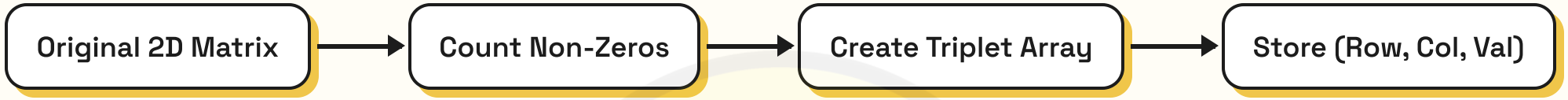


Applications of Arrays & Sparse Matrices

Where arrays excel and how they handle sparse data.

- Applications: Implementing matrices, lookup tables, sorting algorithms, and serving as backing structures for stacks and queues.
- Sparse Matrix: A matrix with a high proportion of zero elements.
- Inefficiency: Storing all zeros wastes valuable RAM.
- Representation Solution: Store only non-zero triplets $(Row, Column, Value)$ using 2-D arrays or linked lists.

Sparse Matrix Representation



Compression pipeline for sparse matrices to save memory.

Linked Lists: Array vs Pointer Implementation

LINKED LIST

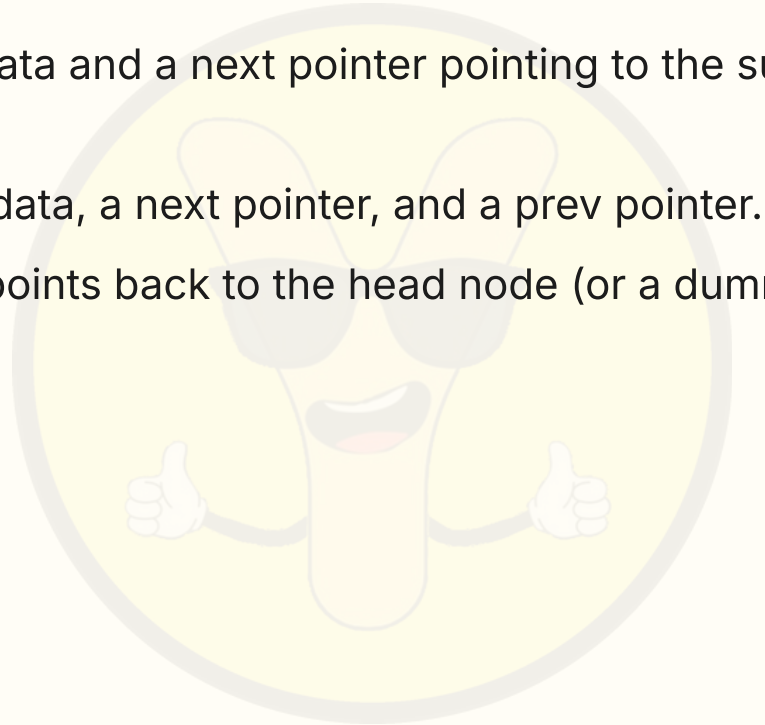
A linear data structure where elements (nodes) are not stored at contiguous memory locations, linked together via pointers.

- Array Implementation: Uses parallel arrays (info array and next array) to simulate pointers using integer indices. Useful in languages without pointers.
- Pointer Implementation: Uses dynamic memory allocation ($malloc/new$) where each node contains data and self-referential pointers.

Variants of Linked Lists

Singly, Doubly, and Circular linked lists explained.

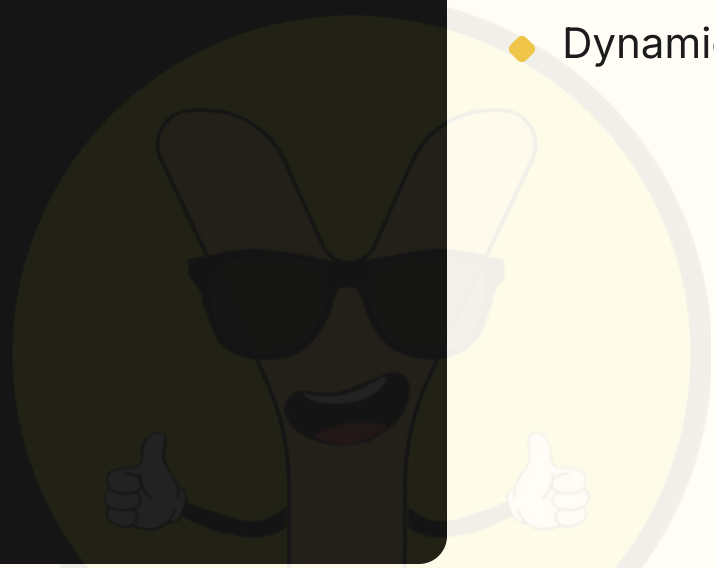
- Singly Linked List: Each node has data and a next pointer pointing to the successor. Traversal is unidirectional.
- Doubly Linked List: Each node has data, a next pointer, and a prev pointer. Allows bidirectional traversal.
- Circular Linked List: The last node points back to the head node (or a dummy node), forming a continuous loop.



Singly Linked List Node Declaration in C

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

struct Node* createNode(int val) {
    struct Node* newNode = (struct
Node*)malloc(sizeof(struct Node));
    newNode→data = val;
    newNode→next = NULL;
    return newNode;
}
```

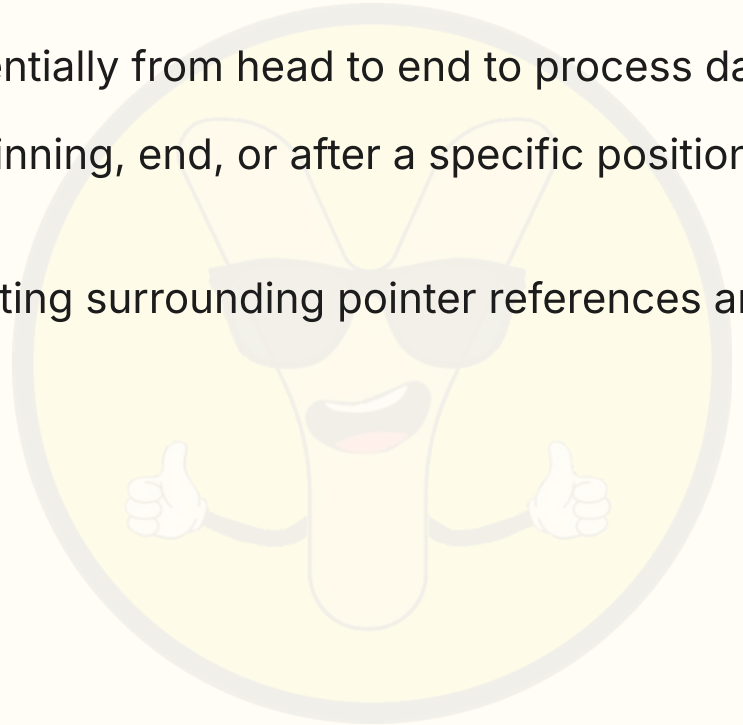


- Defines structural self-reference.
- Dynamically allocates memory using malloc.

Operations on a Linked List

Core manipulations performed on dynamic lists.

- Traversal: Visiting each node sequentially from head to end to process data $O(n)$.
- Insertion: Adding a node at the beginning, end, or after a specific position $O(1)$ at head, $O(n)$ at tail if no tail pointer.
- Deletion: Removing a node by updating surrounding pointer references and freeing memory $O(1)$ at head, $O(n)$ to search.

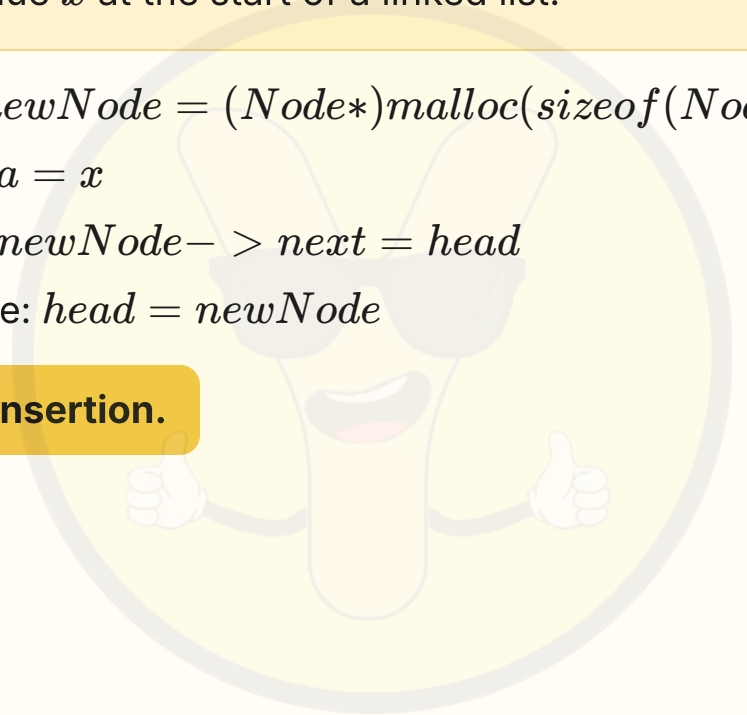


Algorithm: Insertion at Beginning of Singly Linked List

Write steps to insert a new node with value x at the start of a linked list.

1. Step 1: Allocate memory for new node: $newNode = (Node*)malloc(sizeof(Node))$
2. Step 2: Assign data: $newNode->data = x$
3. Step 3: Point new node to current head: $newNode->next = head$
4. Step 4: Update head to point to new node: $head = newNode$

Time Complexity: $O(1)$. Constant time insertion.

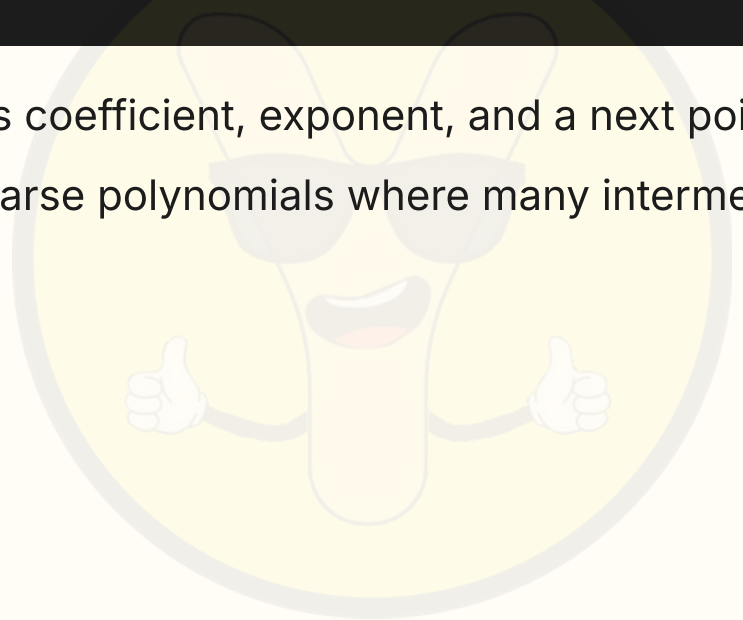


Polynomial Representation

POLYNOMIAL DATA STRUCTURE

Representing algebraic expressions $P(x) = a_n x^n + ... + a_0$ using linked lists.

- Node Structure: Each node contains coefficient, exponent, and a next pointer.
- Advantage: Memory efficient for sparse polynomials where many intermediate degree coefficients are zero.



Polynomial Arithmetic: Addition & Subtraction

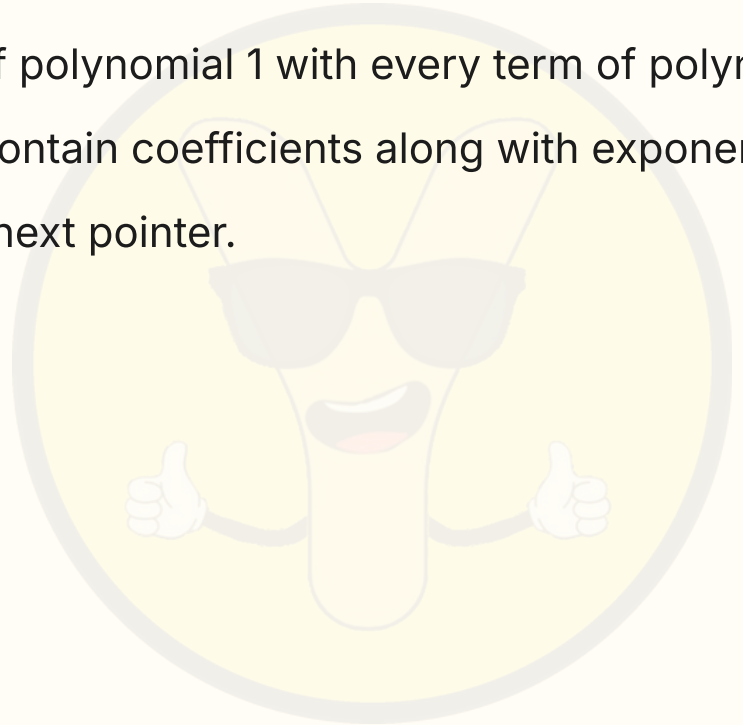
Combining algebraic polynomials using linked lists.

- Algorithm for Addition: Traverse both polynomial lists simultaneously using two pointers p_1 and p_2.
- Case 1 (Exponents equal): Add coefficients, create result node, advance both pointers.
- Case 2 (Exponent $p_1 > p_2$): Copy p_1 node to result, advance p_1.
- Case 3 (Exponent $p_2 > p_1$): Copy p_2 node to result, advance p_2.

Polynomial Multiplication & Two-Variable Polynomials

Advanced polynomial manipulations.

- Multiplication: Multiply each term of polynomial 1 with every term of polynomial 2, then add like terms.
- Two-Variable Polynomials: Nodes contain coefficients along with exponents for two variables (e.g., $x^a y^b$).
- Node Structure: coeff, expX, expY, next pointer.



Exam corner

- **Q1** Define Data Structure. Differentiate between primitive and non-primitive data structures with examples.
- **Q2** Explain Asymptotic Notations (Big-Oh, Big-Omega, Big-Theta) with mathematical definitions and graphical representation.
- **Q3** Derive the address calculation formula for a 2-D array in Row Major Order and Column Major Order.
- **Q4** What is a Sparse Matrix? Explain its efficient representation using triplet form.
- **Q5** Write a complete C function to insert a node at any given position in a Singly Linked List.
- **Q6** Write an algorithm to add two polynomials represented using linked lists.
- **Q7** Compare Arrays and Linked Lists with respect to memory allocation, insertion/deletion, and access time.

Unit 1 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 2 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

