Computing and Software Engineering

GET211

September 23, 2025

Data Structures
A data structure is a specialized format for organizing, processing, retrieving, and storing data. It defines how data is arranged in memory and how operations are performed on it.

Types of Data Structures

Linear Data Structure In linear data structures, elements are arranged in a sequential order. They are simple and easy to implement.

- **Arrays**: Collection of elements stored contiguously. Fixed size, direct indexing for fast access. Example: [10, 20, 30].
- **Linked Lists**: Nodes linked with pointers. Types: singly, doubly, circular. Dynamic size, efficient insertion and deletion.
- **Stacks**: Last In, First Out (LIFO) structure for parsing, recursion, undo mechanisms, and expression evaluation.
- **Queues**: First In, First Out (FIFO) structure for scheduling, buffers. Variants: Circular queue, deque (double-ended queue), priority queue.

Types of Data Structures

Non-Linear Data Structures In non-linear data structures, elements are arranged in a hierarchical manner or interlinked through relationships.

Trees

- Hierarchical structure of nodes with parent-child relationships.
- Examples: Binary Trees, Binary Search Trees, AVL Trees, Heaps, Tries.
- Applications: Searching, sorting, hierarchical data representation.

Graphs

- Nodes (vertices) connected by edges.
- Types: Directed, undirected, weighted, unweighted, cyclic, acyclic.
- Representations: Adjacency matrix, adjacency list.
- Applications: Networking, pathfinding, social networks.

Types of Data Structures

Abstract Data Types (ADTs):

- **List**: Ordered collection. Operations: insertion, deletion.
- **Set**: Unique elements. Operations: union, intersection.
- **Map**: Key-value pairs for fast retrieval.
- **Deque**: Double-ended queue for flexible insertions.

Advanced Data Structures:

- **Hash Tables**: Fast lookup and retrieval.
- **Heaps**: Priority queues for scheduling.
- **Tries**: String processing (autocomplete, prefix search).
- **Segment Trees**: Range queries and updates.
- **Disjoint Sets**: Union-Find for partitioned data.

Key Operations

Data structures need to perform some actions. Here are some of fundamentals operation taken by data structures:

- **Insertion**: Add an element.
- **Deletion**: Remove an element.
- **Traversal**: Visit elements sequentially.
- **Searching**: Finding a specific element.
- **Sorting**: Arranging elements in a specific order (e.g., ascending, descending).

Choosing the Right Data Structure

The choice of data structure depends on:

1. Data Characteristics: Size, type, and relationships among data.
2. Operations: Frequency and type of operations like searching, updating, or traversal.
3. Efficiency: Required time and space complexity for operations.
4. Scalability: Ability to handle growing data efficiently.

Best Practices

1. Understand the problem requirements before choosing a data structure.
2. Optimize memory usage and ensure efficient operations for large datasets.
3. Use existing libraries or frameworks for standard implementations.
4. Profile and test implementations for time and space efficiency.

Applications and Examples of Data Structures

Applications of Data Structures

1. Arrays: Used in matrix operations, image processing, and static datasets.
2. Linked Lists: Ideal for dynamic memory allocation and applications requiring frequent insertions/deletions.
3. Stacks: Used in parsing expressions, backtracking algorithms (e.g., maze solving), and function call management.
4. Queues: Essential for resource scheduling, buffering in data streams, and breadth-first search in graphs.
5. Trees: Critical for database indexing (e.g., B-trees), expression evaluation, and file system hierarchies.
6. Graphs: Solve problems in transportation networks, circuit design, and AI (e.g., shortest path algorithms).

Real-World Examples

1. Web Development: Hash tables for session management.
2. Game Development: Graphs for map navigation and AI behavior trees.
3. Machine Learning: Matrices and tensors for data representation.
4. Databases: B-trees and hashing for indexing and searching.
5. Networking: Graphs for routing protocols.

Data Structures in MATLAB

MATLAB data structures enable efficient data storage and manipulation for numerical, textual, and categorical data. Choose the right data structure based on the type and size of data and the operations needed.
MATLAB provides built-in functions to simplify operations on data structures.
Some of the types of the data structures:
- Arrays
- Cell Arrays
- Structures
- Tables
- Categorical Arrays
- Timetables
- Maps (Containers.Map)
- Sparse Matrices
- Graphs
- Custom Classes and Objects

Arrays

- Arrays are the foundational data structure in MATLAB for storing and manipulating data.
- MATLAB arrays can store numerical, logical, or character data.
- Common types:
  - Numeric Arrays
  - Logical Arrays
  - Character Arrays and Strings
  - Cell Arrays
  - Multidimensional Arrays

Examples of Arrays in MATLAB

1. Numeric Arrays:

```
% Row vector
rowVector = [1, 2, 3];

% Column vector
colVector = [1; 2; 3];

% Matrix
matrix = [1, 2, 3; 4, 5, 6; 7, 8, 9];
```

2. Logical Arrays:

```
logicalArray = [true, false, true];
isGreater = matrix > 5; % Returns a logical array
```

Common Operations in Array Manipulation

Array Creation:

```
A = [10, 20, 30, 40];
Z = zeros(3, 3); % 3x3 matrix of zeros
I = eye(3,3) % 3x3 Identity matrix
L = linspace(0, 10, 5); % 5 points between 0 and 10
K = logspace(1, 3, 4); % 10^1 to 10^3
```

Modifying Elements

```
A(1) = 100; % Modify 1st element
A(2:3) = [200, 300]; % Modify elements
```

Accessing Elements

```
A = [10, 20, 30, 40];
value = A(2); % Access 2nd element
subset = A(2:4); % Elements 2 to 4
logicalIndex = A > 20;
filteredValues = A(logicalIndex);
```

Common Operations in Array Manipulation

Using Random Number Generators

```
% Uniformly distributed random array
A = rand(3, 3);

% Normally distributed random array
B = randn(4, 1);

% Random integers
C = randi([1, 10], 2, 2);
```

Using Colon Operator

```
% Create a sequence
A = 1:5; % [1, 2, 3, 4, 5]

% Specify step size
B = 1:2:9; % [1, 3, 5, 7, 9]
```

Advanced Initialization

```
% Diagonal matrix
A = diag([1, 2, 3]);
% Repeated values
B = repmat(5, 3, 2);
% Magic matrix
C = magic(4);
```

Common Operations in Array Manipulation

Size

```
dims = size(A); % Returns the shape
```

Dimensions

```
ndims(A); % Returns 3
```

Length

```
len=length(A); % Returns 3
```

Flatten array

```
C = A(:); % Convert to 1D vector
```

Reshaping

```
A = [1, 2, 3, 4];
B = reshape(A, [2, 2]); % Reshape
```

Common Operations in Array Manipulation

Adding/Removing Elements

```
A = [10, 20, 30];
A = [A, 40]; % Append 40
A(2) = []; % Remove 2nd element
```

Transposing

```
A = [1, 2, 3; 4, 5, 6];
B = A.'; % Transpose matrix
```

Concatenation

```
A = [1, 2]; B = [3, 4];
C = [A, B]; % Horizontal
D = [A; B]; % Vertical
```

Sorting

```
sortedA = sort(A); % Ascending
sortedA = sort(A, 'descend'); % Desc.
```

Searching

```
indices = find(A > 20); % Get indices
isPresent = ismember(25, A);
```

Common Operations in Array Manipulation

Element-wise Operations

```
B = A + 10; % Add 10 to
    each element
C = A .* 2; % Multiply
    element-wise
result = A .* B; %
    Element-wise
    multiplication
```

Matrix Multiplication

```
C = A * B; % Matrix
    product
```

Aggregation

```
total = sum(A); % Sum
avg = mean(A); % Mean
maxValue = max(A); %
    Maximum
```

Cumulative Operations

```
cumulativeSum =
    cumsum(A); %
    Cumulative sum
```

Multidimensional Arrays

Multidimensional Arrays are arrays with more than two dimensions.

Using Built-in Functions

- `zeros`, `ones`, `rand`, etc.:

```
A = zeros(3, 3, 2); % 3x3x2 array
B = rand(4, 2, 5);  % 4x2x5 array
```

Manual Initialization

- Using concatenation:

```
A(:, :, 1) = [1 2; 3 4];
A(:, :, 2) = [5 6; 7 8];
```

Reshaping Existing Arrays

- `reshape` function:

```
A = reshape(1:24, [4, 3, 2]); % 4x3x2 array
```

Accessing Multidimensional Arrays

Indexing

- Access individual elements:

```
value = A(2, 3, 1); % Element at (2,3,1)
```

- Slice across dimensions:

```
slice = A(:, :, 1); % First layer
```

Manipulating Elements

- Assign values:

```
A(1, 1, 2) = 99; % Update value
```

- Use logical indexing:

```
A(A > 10) = 0; % Set elements >10 to 0
```

Best Practices for Arrays

- Preallocate memory to improve performance:

```
A = zeros(1, 1000); % Preallocate
```

- Use vectorized operations instead of loops:

```
% Inefficient
for i = 1:100
    A(i) = i^2;
end

% Efficient
A = (1:100).^2;
```

- Leverage MATLAB's built-in functions for optimization.

Cell Arrays

- Containers for holding values of different types and sizes.
- Features:
  - Heterogeneous storage.
  - Dynamic resizing.
  - Flexible access: Use {} for content, `()` for the cell itself.

Cell Arrays in MATLAB

Creating Cell Arrays

- Direct Initialization:

```
C = {1, 'data'; 3.14,
    [1 2 3]};
```

- Using `cell` Function:

```
C = cell(2, 3); % 2x3
    empty cell array
```

- Converting Data:

```
A = [1, 2; 3, 4];
C = num2cell(A); %
    Convert to cell
    array
```

Accessing Elements

- Access cell contents:

```
value = C{1, 2};
```

- Access the cell itself:

```
subCell = C(1, :);
```

Modifying Cells

- Update content:

```
C{2, 1} = 'new text';
```

- Dynamic resizing:

```
C{3, 1} = pi;
```

Manipulating Cell Arrays

Operations on Cell Arrays

- Concatenation:

```
C1 = {1, 2}; C2 =
    {'a', 'b'};
C = [C1; C2];
```

- Extracting elements:

```
values = C{:, 2};
```

Applications of Cell Arrays

- Storing mixed data types:

```
patientData =
    {'Name', 'Age';
    'Alice', 30;
    'Bob', 25};
```

- Managing uneven data sizes:

```
unevenData = {1:3,
    4:8, rand(2)};
```

- Iterative processing:

```
for i = 1:numel(C)
    disp(C{i});
end
```

Categorical Arrays in MATLAB

Categorical Arrays

- Data type for storing categorical data.
- Values are represented as categories (labels) instead of raw data.
- Reduces memory usage for repeated labels.

Use Cases

- Storing non-numeric data (e.g., gender, region).
- Grouping data for statistical analysis.
- Efficient data handling for repeated labels.

Creating Categorical Arrays

From Cell Arrays

```
data = {'red', 'blue',
    'red', 'green'};
catArray =
    categorical(data);
```

Specifying Categories

```
catArray =
    categorical({'high',
    'low',
    'medium'},{'low',
    'medium', 'high'},
    'Ordinal', true);
```

Remove unused categories

```
catArray =
    removecats(catArray);
```

Converting Arrays

```
numericData = [1 2 1 3];
catArray =
    categorical(numericData,
    [1 2 3], {'low',
    'medium', 'high'});
```

Common Operations

- Get Categories:

```
categories(catArray);
```

- Count Occurrences:

```
counts =
    countcats(catArray);
```

Working with Categorical Arrays

Manipulating Categories

- Add Categories:

```
catArray =
    addcats(catArray,
    'extra');
```

- Merge Categories:

```
catArray =
    mergecats(catArray,
    {'red', 'blue'},
    'primary');
```

Practical Applications

- Data summarization and grouping:

```
grpstats(data,
    catArray, {'mean',
    'std'});
```

- Visualization:

```
bar(categories(catArray),
    counts);
```

- Statistical Analysis:

```
anova1(response,
    catArray);
```

Structures in MATLAB

Structures are data types in MATLAB that group related data using named fields, which can hold different types of data.

- Fields act as named containers for data.
- Allow organization of complex or heterogeneous data.
- Useful for encapsulating data and its associated metadata.

Key Features:

- Flexible: Each field can contain data of different types or sizes.
- Easily accessible using dot notation.
- Support dynamic creation and manipulation.

Creating Structures in MATLAB

Manual Creation:

```
% Create a structure
student.name = 'Alice';
student.age = 23;
student.scores = [90, 85,
    88];
```

Using the struct Function:

```
% Create using struct
student = struct('name',
    'Alice', 'age', 23,
    'scores', [90, 85,
    88]);
```

Preallocating Structures:

```
students(3) = struct('name', '', 'age', 0, 'scores',
    []);
students(1).name = 'Alice'; students(1).age = 23;
    students(1).scores = [90, 85, 88];
students(2).name = 'Bob'; students(2).age = 25;
    students(2).scores = [80, 75, 78];
students(3).name = 'Charlie'; students(3).age = 22;
    students(3).scores = [88, 90, 92];
```

Accessing and Modifying Structures

Accessing Fields:

```
% Access a single field
name = student.name;

% Access a field in an
    array of structures
ages = [students.age];
```

Modifying Fields:

```
% Update a field
student.age = 24;

% Add a new field
student.grade = 'A';
```

Removing Fields:

```
% Remove a field
student =
    rmfield(student,
    'grade');
```

Field Names and Manipulation:

```
% List all field names
fields =
    fieldnames(student);

% Check if a field exists
isFieldPresent =
    isfield(student,
    'age');
```

Operations on Structures

Merging Structures:

```
% Merge two structures
studentExtra = struct('hobbies', {'reading',
    'swimming'});
combinedStudent = structmerge(student, studentExtra);
```

Converting Structures:

```
% Convert structure to cell array
cellArray = struct2cell(student);

% Convert structure to table
tableData = struct2table(students);
```

Tables in MATLAB

Tables in MATLAB are data types used for organizing, storing, and processing heterogeneous data. They are well-suited for handling tabular data where rows represent observations and columns represent variables.

- Support for different data types in each column.
- Allow metadata like variable names and row names.
- Enable indexing, manipulation, and integration with visualization tools.

Operations on Tables

Creating Tables From Arrays:

```
% Create a table from arrays
Name = {'Alice'; 'Bob'; 'Charlie'};
Age = [25; 30; 35];
Height = [5.5; 6.1; 5.8];
T = table(Name, Age, Height)
```

Importing Tables:

```
% Import from a CSV file
T = readtable('data.csv');
```

Grouping and Aggregation:

```
G = groupsummary(T, 'Age', 'mean');
```

Adding Metadata:

```
% Add description and variable names
T.Properties.Description = 'Sample Participant Data';
T.Properties.VariableNames = {'Name', 'Age', 'Height'};
```

Summary Statistics:

```
summary(T)
```

Sorting Rows:

```
T = sortrows(T, 'Age');
```

Operations on Tables

Accessing Data:

```
% Access by variable name
T.Age

% Access by row index
T(2, :)

% Access by logical
    indexing
T(T.Age > 30, :)
```

Joining Tables:

```
T2 = table({'Alice';
    'Bob'}, [100; 200],
    'VariableNames',
    {'Name', 'Score'});
TJoined = join(T, T2,
    'Keys', 'Name');
```

Modifying Data:

```
% Add a new variable
T.Weight = [60; 75; 70];

% Rename a variable
T.Properties.VariableNames{2
    = 'Years';
```

Adding/Removing Rows:

```
% Append a new row
newRow = {'David', 28,
    5.9};
T = [T; newRow];

% Remove the third row
T(3, :) = [];
```

Maps in MATLAB

Maps in MATLAB are data structures that store data in key-value pairs. They are ideal for situations where you need to access elements efficiently using unique keys.

- Similar to dictionaries in other programming languages (e.g., Python).
- Keys can be of various data types (e.g., numbers, strings, objects).
- Values can store any MATLAB data type (arrays, cell arrays, structures, etc.).

Applications:

- Efficient lookup and storage of data.
- Associative arrays for mapping relationships.
- Data indexing with non-numeric keys.

Creating Maps in MATLAB

Manual Creation:

```
% Create a map with string keys and numeric values
mapExample = containers.Map({'key1', 'key2', 'key3'},
    [10, 20, 30]);
```

Dynamic Creation:

```
% Create an empty map and populate it
myMap = containers.Map();
myMap('A') = 1;
myMap('B') = 2;
myMap('C') = 3;
```

Using Complex Keys:

```
% Use numeric or mixed keys
mapMixed = containers.Map({'January', 'February'},
    {31, 28});
mapNumbers = containers.Map([1, 2, 3], {'One', 'Two',
    'Three'});
```

Accessing and Modifying Maps

Accessing Values:

```
% Retrieve a value using a key
value = mapExample('key1');
```

Adding and Removing Key-Value Pairs:

```
% Add a new key-value pair
myMap('D') = 4;

% Remove a key-value pair
remove(myMap, 'B');
```

Checking for Existence:

```
% Check if a key exists
isKey(myMap, 'A')   % Returns true
isKey(myMap, 'Z')   % Returns false
```

Keys and Values:

```
% Retrieve all keys or values
keysList = keys(myMap);
valuesList = values(myMap);
```

Iterating Through Maps:

```
% Loop through keys and values
keySet = keys(myMap);
for i = 1:length(keySet)
    disp(['Key: ', keySet{i}, ', Value: ',
        num2str(myMap(keySet{i}))]);
end
```

Combining Maps:

```
% Merge two maps
map1 = containers.Map({'A', 'B'}, [1, 2]);
map2 = containers.Map({'C', 'D'}, [3, 4]);
combinedMap = [map1; map2]; % Custom handling
    required for conflicts
```

Clearing All Entries:

```
% Clear map
clearMap = containers.Map();
clearMap('A') = 10;
clearMap.remove(keys(clearMap)); % Remove all entries
```

Exercise

Generate the following Matrix: $X = \begin{bmatrix} 4 & 16 & 6 \\ 5 & 9 & 17 \\ 8 & 21 & 33 \end{bmatrix}$

- Extract the sub-matrix in rows 2 to 3 and columns 1 to 2 of the matrix X
- Extract the second column of the matrix X.
- Extract the first row of the matrix
- Extract the element in row 1 and column 3 of the matrix X

Exercise

- Generate a vector containing the first 20 even numbers.
- Reverse the elements of a given vector.
- Find the maximum, minimum, and average values within a vector.

Exercise

The table shows the hourly cost of four types of manufacturing processes. It also shows the number of hours required of each process to produce three different products. Use table data structure and MATLAB to solve the following.
(a) Determine the cost of each process to produce 1 unit of product 1.
(b) Determine the cost to make 1 unit of each product.
(c) Suppose we produce 10 units of product 1, 5 units of product 2, and 7 units of product 3. Compute the total cost for each process.

		Hours required to produce one unit		
Process	**Hourly cost ($)**	**Product 1**	**Product 2**	**Product 3**
Lathe	10	6	5	4
Grinding	12	2	3	1
Milling	14	3	2	5
Welding	9	4	0	3

Exercise

Create a cell array to store the following information for three students: Name, Age, and Grade. Perform the following operations:
- Display the name of the student with the highest grade.
- Add a new student to the cell array.

Hint

```
 Input
 students = Ali, 20, 85; Bimbo, 22, 90; Chukwuma,
    19, 78;

Add the student:
Muhammed, 18 , 75

Output:
Top student is Bimbo

Display updated list
```

Exercise

Create a map where keys are city names and values are their respective populations.
- Add five cities and their populations.
- Write a script to find the city with the highest population.