DATA STRUCTURES

Unit I — Data Structures & Algorithms: Searching, Sorting & Hashing

---

Topics Covered in this Unit

Introduction to Data Structures: Primitive vs. Non-Primitive, Linear vs. Non-Linear, Static vs. Dynamic Structures

Algorithm Analysis & Complexity: Space Complexity (Fixed & Variable parts), Time Complexity (Best, Average, Worst Case)

Asymptotic Notations: Mathematical Definitions & Graphs for Big-O, Big-Theta, Big-Omega, Little-o, and Little-omega

Step Count Method: Frequency Count Analysis, Steps per Execution (s/e), and Tabular Derivation of Complexity

Analysis of Programming Constructs: Constant O(1), Logarithmic O(log n), Linear O(n), Quadratic $O(n^2)$, Cubic $O(n^3)$, Exponential $O(2^n)$

Searching Algorithms: Sequential/Linear Search vs. Binary Search (Iterative & Recursive) with Traces & Complexities

Sorting Algorithms: Insertion Sort, Bubble Sort, Selection Sort, Merge Sort (Divide & Conquer), Quick Sort (Partitioning), Radix Sort

Comprehensive Sorting Comparison: Time Complexities, Space Overhead, In-Place Nature, and Algorithm Stability

Hashing & Hash Tables: Hash Functions (Division, Mid-Square, Folding, Multiplication) and Desirable Properties

Collision Resolution Techniques: Open Addressing (Linear Probing, Quadratic Probing, Double Hashing) vs. Separate Chaining

Hash Table Overflow & Rehashing: Load Factor Analysis ($\alpha = n/m$), Clustering Effects, and Dynamic Table Resizing

High-Yield University Exam Question Bank with Detailed Model Answers & Unit Summary

Compiled in a structured, easy-to-understand, textbook style format
Specially curated for B.Tech / B.E. / BCA / MCA / B.Sc Computer Science & Engineering Students

COURSE SYLLABUS & MAPPING

[UNIT I SYLLABUS — PRESCRIBED UNIVERSITY CURRICULUM]

This section contains the official syllabus for Unit I. Verify with your university curriculum mapping.

Unit I Syllabus Breakdown:

Unit I - Data Structures & Algorithms: Searching and Sorting: Introduction of Data Structures & types. Complexity of algorithm: Space complexity, Time complexity, Asymptotic notation- Big-O, Theta and Omega, finding complexity using step count method, Analysis of programming constructs-Linear, Quadratic, Cubic, Logarithmic. Searching: Sequential Search, Binary Search. Sorting: Insertion sort, Bubble Sort, Merge sort, Selection Sort, Quick sort, Radix sort. Hash: Hash Table, Hash Function, Collision Resolution Techniques in Hashing-Chaining, Open Addressing-Linear, Quadratic Probing and Double Hashing. Hash table overflow open addressing and chaining.

• Course Code: _____ • Semester / Year: _____

TABLE OF CONTENTS

1.1 Introduction to Data Structures & Classification

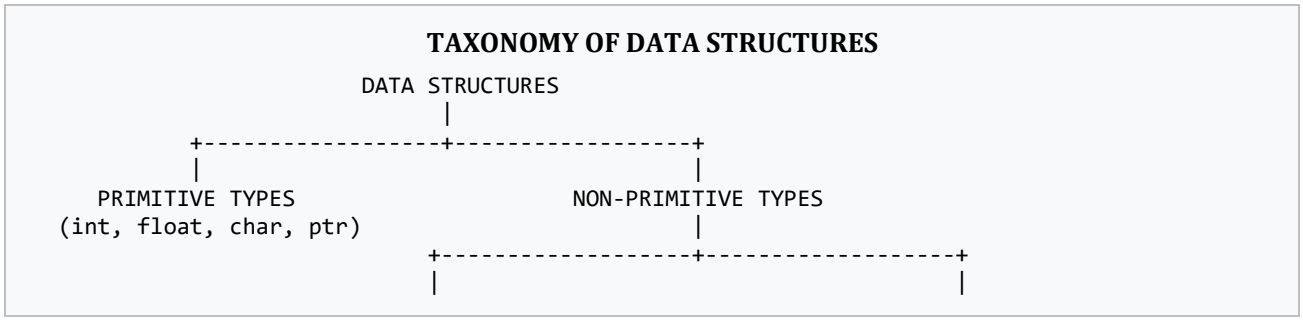
Definition: Data Structure

A Data Structure is a specialized format for organizing, processing, retrieving, and storing data in computer memory efficiently, such that operations (such as insertion, deletion, searching, traversal, and sorting) can be executed with optimal time and space resources.

Classification of Data Structures

Data structures are broadly categorized based on their organization, memory allocation, and relationship among elements:

- **1. Primitive Data Structures:** Basic atomic data types directly supported by hardware and machine-level instructions. Examples include Integer (int), Floating-point (float), Character (char), Boolean, and Pointers.
- **2. Non-Primitive Data Structures:** More complex data structures derived from primitive data types. They emphasize grouping homogeneous or heterogeneous data items into structured collections.
  - **a. Linear Data Structures:** Elements form a sequential succession where every element (except first and last) has a unique predecessor and successor. Traversal is sequential (single-run). Examples: Arrays, Linked Lists, Stacks, Queues.
  - **b. Non-Linear Data Structures:** Elements are arranged hierarchically or interconnected arbitrarily (multi-level relationships). An element can connect to multiple elements. Examples: Trees (hierarchical parent-child), Graphs (many-to-many networks).
- **3. Static vs. Dynamic Data Structures:** Static structures (e.g., Arrays) have fixed memory allocated at compile time. Dynamic structures (e.g., Linked Lists, Trees) grow or shrink dynamically during execution using heap memory.
- **4. Homogeneous vs. Heterogeneous Data Structures:** Homogeneous structures store elements of the same data type (e.g., Arrays), whereas heterogeneous structures store mixed types (e.g., Structures, Classes).



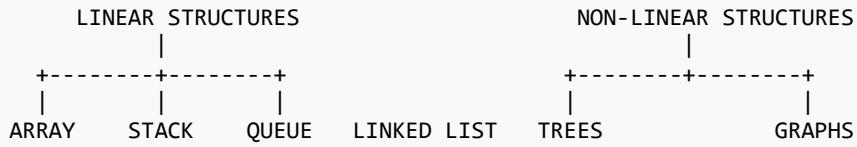


Figure: Comprehensive hierarchical classification of primitive and non-primitive data structures

1.2 Algorithm Complexity & Asymptotic Notations

Definition: Algorithm Complexity

Algorithm Complexity is a formal measure of the computational resources (Time and Space/Memory) required by an algorithm as a function of the input size (n).

- *Space Complexity S(P) = c + S_p(I): Sum of fixed part c (instruction space, simple variables, constants) and variable instance part S_p(I) (dynamically allocated memory, recursion call stack).*
- *Time Complexity T(n): Total number of fundamental operations executed by the algorithm as input size n approaches infinity.*

Asymptotic Notations (Formal Mathematical Definitions)

Asymptotic notations describe the limiting behavior of execution time or memory when input size n increases indefinitely:

Notation	Formal Definition	Mathematical Meaning	Graphical Interpretation
Big-O Notation (O)	$f(n) = O(g(n))$ iff $\exists$ constants $c > 0$, $n_0 \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$	Asymptotic Upper Bound (Worst-Case Guarantee)	f(n) never grows faster than $c \cdot g(n)$ beyond n_0.
Big-Omega (Ω)	$f(n) = \Omega(g(n))$ iff $\exists$ constants $c > 0$, $n_0 \geq 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$	Asymptotic Lower Bound (Best-Case Guarantee)	f(n) grows at least as fast as $c \cdot g(n)$ beyond n_0.
Big-Theta (Θ)	$f(n) = \Theta(g(n))$ iff $\exists$ constants $c_1, c_2 > 0$, $n_0 \geq 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$	Asymptotic Tight Bound (Exact Growth Rate)	f(n) is sandwiched between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$.
Little-o Notation (o)	$f(n) = o(g(n))$ iff $\lim (n \to \infty) [f(n) / g(n)] = 0$	Strict Upper Bound (Non-tight upper bound)	f(n) grows strictly slower than g(n).
Little-omega (ω)	$f(n) = \omega(g(n))$ iff $\lim (n \to \infty) [f(n) / g(n)] = \infty$	Strict Lower Bound (Non-tight lower bound)	f(n) grows strictly faster than g(n).

ASYMPTOTIC BOUNDS VISUALIZATION

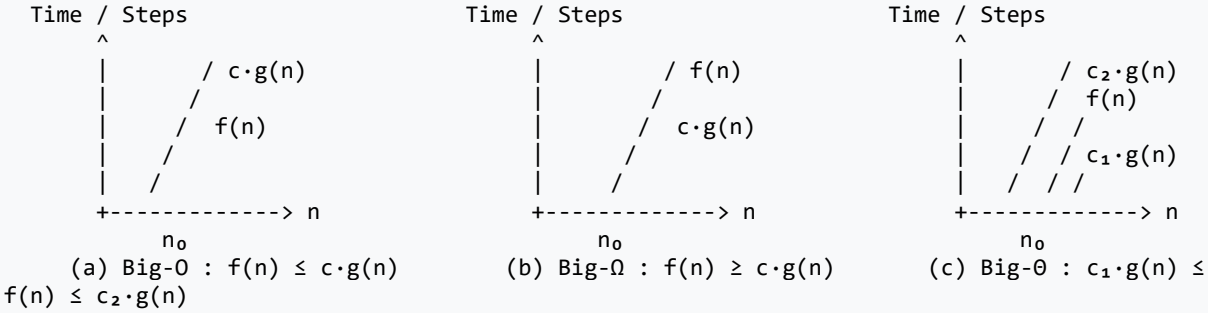


Figure: Visual representation of Upper Bound (Big-O), Lower Bound (Big-Omega), and Tight Bound (Big-Theta)

1.3 Step Count Method & Analysis of Programming Constructs

The Step Count Method determines time complexity by counting the number of executable steps (statements) performed. Each statement has a Step per Execution (s/e) and a Frequency Count (the number of times it executes).

Step Count Table: Matrix Multiplication Algorithm

Consider Matrix Multiplication C[n][n] = A[n][n] × B[n][n]:

Statement	*s/e*	*Frequency*	*Total Steps*
for (int i = 0; i < n; i++)	*1*	*n + 1*	*n + 1*
for (int j = 0; j < n; j++)	*1*	*n(n + 1)*	$n^2 + n$
C[i][j] = 0;	*1*	n^2	n^2
for (int k = 0; k < n; k++)	*1*	$n^2(n + 1)$	$n^3 + n^2$
*C[i][j] += A[i][k] * B[k][j];*	*1*	n^3	n^3

Total Step Count $T(n) = 2n^3 + 3n^2 + 2n + 1 \Rightarrow T(n) = O(n^3)$

Analysis of Standard Programming Constructs

- **1. Constant Time O(1):** Sequential assignment, arithmetic operations, array element access by index. Execution time is independent of input size n.
- **2. Logarithmic Time O(log n):** Loop counter multiplied or divided by a constant factor (e.g., for (i = 1; i <= n; i *= 2)). Halves problem size at each step (e.g., Binary Search).

- **3. Linear Time O(n):** Single loop executing n times (e.g., for (i = 0; i < n; i++)). Traverses every element once (e.g., Linear Search).
- **4. Linearithmic Time O(n log n):** Divide-and-conquer algorithms that divide problem into subproblems and combine in linear time (e.g., Merge Sort, Quick Sort average).
- **5. Quadratic Time $O(n^2)$:** Two nested loops each running up to n (e.g., Bubble Sort, Insertion Sort, Selection Sort).
- **6. Cubic Time $O(n^3)$:** Three nested loops (e.g., Naive Matrix Multiplication, Floyd-Warshall All-Pairs Shortest Path).
- **7. Exponential Time $O(2^n)$:** Algorithms generating all subsets/combinations (e.g., Recursive Fibonacci, 0/1 Knapsack brute force, Tower of Hanoi).

1.4 Searching Algorithms: Sequential vs. Binary Search

1. Sequential Search (Linear Search)

Sequential search examines each array element from index 0 to n-1 sequentially until the target element is found or the end of the array is reached. Applicable to both sorted and unsorted data structures.

- **Best-Case Complexity:** O(1) — Target key is present at the very first index (arr[0]).
- **Worst-Case Complexity:** O(n) — Target key is at the last index (arr[n-1]) or not present in the array.
- **Average-Case Complexity:** O(n) — Average comparisons = (1 + 2 + ... + n)/n = (n + 1)/2 ≈ O(n).

2. Binary Search

Binary Search is an efficient searching algorithm based on the Divide and Conquer strategy. Prerequisite: The array MUST be sorted. It compares target key with middle element arr[mid]:

- • **If key == arr[mid]:** Search successful; return index mid.
- • **If key < arr[mid]:** Search continues in left subarray (high = mid - 1).
- • **If key > arr[mid]:** Search continues in right subarray (low = mid + 1).

Binary Search Recurrence & Derivation

Recurrence Relation: T(n) = T(n/2) + c

Applying Master Theorem or Back-Substitution:

T(n) = T(n/4) + 2c = T(n/8) + 3c = ... = T(n/2^k) + k·c

Let n/2^k = 1 ==> k = $log_2(n)$

Therefore: T(n) = T(1) + c·$log_2(n)$ ==> Time Complexity = O(log n).

Space Complexity: O(1) for Iterative; O(log n) for Recursive due to call stack.

Comparison Parameter	Sequential (Linear) Search	Binary Search
Data Precondition	None; works on unsorted and sorted collections.	Array MUST be sorted in ascending/descending order.
Access Mechanism	Sequential / Random access (Arrays & Linked Lists).	Direct random access required (Arrays only).
Best-Case Time	O(1) (First element match)	O(1) (Middle element match)
Average & Worst-Case Time	O(n)	O(log n)
Auxiliary Space	O(1)	O(1) Iterative, O(log n) Recursive
Efficiency on Large Data (n = 10^6)	Up to 1,000,000 comparisons	At most 20 comparisons ($\log_2 10^6 \approx 20$)

1.5 Sorting Algorithms: Principles, Traces & Analysis

1. Bubble Sort

Repeatedly compares adjacent elements and swaps them if they are in the wrong order. After pass i, the i-th largest element bubbles up to its correct final position at the end of the array.

- **Optimized Flag:** An isSwapped boolean flag halts the algorithm in pass 1 if no swaps occur, achieving O(n) best-case time for already-sorted arrays.

2. Selection Sort

Divides array into sorted (left) and unsorted (right) subarrays. In each pass, finds the minimum element in the unsorted subarray and swaps it with the first element of the unsorted subarray.

- **Key Characteristic:** Performs minimal memory writes (at most n-1 swaps), but always requires $O(n^2)$ comparisons irrespective of initial order. Not stable in standard array implementation.

3. Insertion Sort

Builds the sorted array one element at a time by picking the next element (key) and shifting larger elements in the sorted portion one position to the right to make room.

- **Adaptive & Online:** Highly efficient for nearly sorted data (O(n) comparisons) and small datasets (n < 30). Used as the base case in hybrid algorithms like TimSort.

4. Merge Sort (Divide and Conquer)

Recursively divides the array into two halves until single-element subarrays remain, then merges sorted halves using auxiliary memory.

- **Recurrence:** $T(n) = 2T(n/2) + \Theta(n)$ ==> By Master Theorem Case 2: $T(n) = \Theta(n \log n)$ in all cases (Best, Average, Worst).
- **Properties:** Stable sorting algorithm. Requires O(n) auxiliary memory. Ideal for sorting Linked Lists and External Sorting.

5. Quick Sort (Divide and Conquer via Partitioning)

Selects a 'pivot' element and partitions the array such that all elements smaller than pivot are moved to its left, and all elements larger are moved to its right. Recursively sorts subarrays.

- **Best & Average Case:** O(n log n) when pivot consistently divides array into roughly equal halves.
- **Worst Case $O(n^2)$:** Occurs when array is already sorted/reverse-sorted and pivot is chosen as first or last element (unbalanced partitions of sizes 0 and n-1).
- **Mitigation:** Randomized pivot selection or Median-of-Three pivot strategy prevents worst-case degradation.

6. Radix Sort (Non-Comparative Sorting)

Non-comparison integer sorting algorithm that processes numbers digit by digit, from Least Significant Digit (LSD) to Most Significant Digit (MSD), using a stable sub-sorting subroutine (such as Counting Sort with base b = 10).

- **Time Complexity:** $O(d \cdot (n + b))$, where d is the number of digits in maximum key, n is element count, and b is the base (radix). For fixed d, achieves linear time O(n).

Master Comparison Table of Sorting Algorithms

Algorithm	Best Time	Average Time	Worst Time	Space Complexity	Stable?	In-Place?
Bubble Sort	O(n) (Optimized)	$O(n^2)$	$O(n^2)$	O(1)	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No	Yes
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	No
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n) stack	No	Yes
Radix Sort	O(d · (n + b))	O(d · (n + b))	O(d · (n + b))	O(n + b)	Yes	No

1.6 Hashing: Hash Tables, Functions & Collision Resolution

Definition: Hashing & Hash Table

Hashing is a technique that maps large keys of arbitrary size into fixed-size integer indices (hash codes) in a table called a Hash Table, enabling average-case O(1) time complexity for search, insert, and delete operations.

- *Hash Function: h(k) maps key k into index range [0, m-1], where m is table size.*
- *Load Factor (α): Ratio α = n / m (where n = number of stored keys, m = total table slots). Measures fullness.*

Popular Hash Functions

- **1. Division Remainder Method:** h(k) = k mod m. To minimize collisions, table size m should be a Prime Number not close to powers of 2 or 10.
- **2. Mid-Square Method:** Square the key (k^2) and extract r middle digits to form the hash index. For table size m = 100, r = 2 digits. Uniformly distributes keys since all digits contribute to the square.
- **3. Folding Method:** Divide key k into equal-sized parts (except possibly the last), add the parts together, and take mod m. Variations: Fold-shifting vs. Fold-boundary (reversing alternate segments).
- **4. Multiplication Method:** h(k) = floor(m · (k · A mod 1)), where 0 < A < 1 is a constant (Knuth suggests golden ratio fractional part A ≈ (√5 - 1)/2 ≈ 0.618033).

Collision Resolution Techniques

A Collision occurs when two distinct keys $k_1 \neq k_2$ yield the same hash value $h(k_1) = h(k_2)$. Collision resolution methods are categorized into Chaining (Open Hashing) and Open Addressing (Closed Hashing):

- **A. Separate Chaining (Open Hashing):** Each slot in the hash table points to a linked list of records that hash to the same slot.
  - • Advantages: Handles unlimited collisions; load factor α can exceed 1.0; simple deletion.
  - • Disadvantages: Pointer overhead; poor cache locality; worst-case lookup degrades to O(n) if all keys hash to a single slot.

- **B. Open Addressing (Closed Hashing):** All keys are stored directly in the hash table array ($m \geq n$). When collision occurs, systematically probe alternative slots:
  - **1. Linear Probing:** Probe sequence: $h(k, i) = (h'(k) + i) \bmod m$, for $i = 0, 1, ..., m-1$. Suffers from Primary Clustering (long contiguous blocks of occupied slots build up, drastically increasing average probe count).
  - **2. Quadratic Probing:** Probe sequence: $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$. Eliminates primary clustering but suffers from Secondary Clustering (keys with same initial hash follow identical probe paths).
  - **3. Double Hashing:** Probe sequence: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$, where $h_2(k)$ is a secondary hash function (must return a value relatively prime to m, e.g., $h_2(k) = 1 + (k \bmod m')$). Completely eliminates primary and secondary clustering.

COLLISION RESOLUTION COMPARISON

```
SEPARATE CHAINING (OPEN HASHING)          OPEN ADDRESSING (LINEAR PROBING)
Slot                                      Slot   Array Content
[0] -> [ 20 ] -> [ 50 ] -> NULL           [0]  |  20  |
[1] -> NULL                               [1]  |  50  | <-- Collision at 0 resolved by
[2] -> [ 12 ] -> NULL                     [2]  |  12  |     probing slot 1 linearly
[3] -> [ 33 ] -> [ 73 ] -> NULL           [3]  |  33  |
[4] -> NULL                               [4]  |  73  | <-- Collision at 3 placed at 4
```

Figure: Structural comparison between Separate Chaining (linked lists) and Open Addressing (array probing)

Hash Table Overflow & Rehashing

- **Table Overflow:** In Open Addressing, when the table becomes full ($\alpha \geq 0.7$ to 0.8), probe lengths spike exponentially and insertions fail.
- **Rehashing Process:** When load factor α exceeds a defined threshold (typically $\alpha > 0.75$), allocate a new hash table of roughly DOUBLE size (next prime number ≈ 2m), re-calculate hash indices for all existing elements using the new modulus, and insert them into the new table. Amortized cost per operation remains O(1).

1.7 High-Yield University Exam Question Bank

Part A: Short Answer Questions (2 to 5 Marks)

Q1. Differentiate between Linear and Non-Linear data structures with two examples each. [3 Marks]

- **Answer:** In Linear Data Structures, elements are arranged sequentially in a single tier where every element has unique predecessors and successors (e.g., Arrays, Linked Lists). In Non-Linear Data Structures, elements form multi-level hierarchical or interconnected relationships (e.g., Trees, Graphs). Traversal in linear structures requires single-pass, while non-linear requires recursive or multi-path exploration.

Q2. Define Big-O, Big-Omega, and Big-Theta notations with their formal mathematical equations. [4 Marks]

- **Answer:** 1. Big-O: $f(n) = O(g(n))$ iff $\exists\ c > 0, n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$ (Upper bound). 2. Big-Omega: $f(n) = \Omega(g(n))$ iff $\exists\ c > 0, n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)\ \forall\ n \geq n_0$ (Lower bound). 3. Big-Theta: $f(n) = \Theta(g(n))$ iff $\exists\ c_1, c_2 > 0, n_0 \geq 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \forall\ n \geq n_0$ (Tight bound).

Q3. What is Primary Clustering in Linear Probing, and how does Double Hashing resolve it? [3 Marks]

- **Answer:** Primary clustering is the phenomenon where occupied slots form large contiguous clusters in linear probing because probe step is constant (+1). Any key hashing into the cluster extends it further. Double Hashing resolves this by computing step size using a second hash function $h_2(k)$, ensuring different keys have distinct probe sequences even if they have the same initial hash.

Q4. Why is Quick Sort faster than Merge Sort in practice despite having worst-case $O(n^2)$? [3 Marks]

- **Answer:** Quick Sort operates in-place with O(1) auxiliary space (excluding recursion stack), exhibiting superior CPU cache locality and lower constant factors. Merge Sort requires O(n) auxiliary memory copying. With randomized/median pivot, Quick Sort's worst case is virtually eliminated.

Part B: Long Answer Questions (10 to 15 Marks Outline Guides)

- **Q5. Explain the Step Count Method. Calculate total step count and asymptotic time complexity for Matrix Multiplication and Recursive Fibonacci algorithms. [12 Marks]**
- **Q6. Write complete algorithms for Binary Search (iterative and recursive). Trace Binary Search on array [11, 22, 33, 44, 55, 66, 77, 88, 99] for key = 66. Derive its time complexity recurrence relation. [10 Marks]**

- **Q7. Explain Quick Sort algorithm with Lomuto/Hoare partitioning. Trace Quick Sort for array [38, 27, 43, 3, 9, 82, 10]. Analyze its best, average, and worst-case time complexity. [12 Marks]**
- **Q8. What is Collision in Hash Tables? Compare Chaining vs. Open Addressing (Linear Probing, Quadratic Probing, Double Hashing). Insert keys {25, 42, 96, 101, 102, 162, 197} into hash table of size 10 using h(k) = k mod 10 with Linear Probing and Separate Chaining. [14 Marks]**

1.8 Unit Summary & Key Takeaways

- Data structures are organized formats for storing and manipulating data, categorized into primitive vs. non-primitive and linear (Array, List, Stack, Queue) vs. non-linear (Tree, Graph).
- Space complexity combines fixed overhead with instance variables; Time complexity measures operations executed as a function of input size n.
- Asymptotic notations mathematically bound algorithms: Big-O (upper bound), Big-Omega (lower bound), and Big-Theta (tight bound).
- Binary Search requires sorted data and executes in O(log n) time, outperforming Linear Search's O(n) for large datasets.
- Sorting algorithms offer trade-offs: Bubble/Insertion/Selection are $O(n^2)$ simple sorts; Merge Sort guarantees O(n log n) with O(n) space; Quick Sort provides in-place O(n log n) average performance; Radix Sort achieves O(d·n) linear sorting.
- Hashing maps keys to indices for average O(1) operations. Collisions are resolved via Separate Chaining or Open Addressing (Linear/Quadratic probing and Double Hashing).
- Rehashing dynamically doubles table size and re-indexes elements when load factor exceeds threshold $\alpha \approx 0.75$.