Master of Computer Applications (MCA)

Advance Data Structure and Algorithm Lab

(OMCASE110P24)

Self-Learning Material (SEM 1)

Jaipur National University
Centre for Distance and Online Education

Established by Government of Rajasthan
Approved by UGC under Sec 2(f) of UGC ACT 1956
&
NAAC A+ Accredited

TABLE OF CONTENTS

EXPERT COMMITTEE

Prof. Sunil Gupta
(Department of Computer and Systems Sciences, JNU Jaipur)

Dr. Satish Pandey
(Department of Computer and Systems Sciences, JNU Jaipur)

Dr. Shalini Rajawat
(Department of Computer and Systems Sciences, JNU Jaipur)

COURSE COORDINATOR

Mr. Hitendra Agrawal
(Department of Computer and Systems Sciences, JNU Jaipur)

UNIT PREPARATION

Unit Writer(s)	**Assisting & Proofreading**	**Unit Editor**
Mr. Hitendra Agrawal (Department of Computer and Systems Sciences, JNU Jaipur)	Ms. Heena Shrimali (Department of Computer and Systems Sciences, JNU Jaipur)	Mr. Ramlal Yadav (Department of Computer and Systems Sciences, JNU Jaipur)

Secretarial Assistance
Mr. Mukesh Sharma

COURSE INTRODUCTION

The Advanced Data Structures Laboratory course for students is an intensive, hands-on module designed to build upon the foundational knowledge of data structures and algorithms. This course focuses on the practical implementation and analysis of complex data structures, enhancing the students' ability to solve sophisticated computational problems efficiently. By the end of the course, students will have a deep understanding of various advanced data structures and their applications in real-world scenarios.

The course begins with an exploration of **advanced tree data structures**. Students will delve into AVL trees, Red-Black trees, and B-trees, understanding their balancing mechanisms and how these structures optimize search, insertion, and deletion operations. Segment trees and interval trees will also be covered, focusing on their applications in range query problems.

Following the study of trees , the course transitions into **graph algorithms**. Students will learn different graph representation techniques, such as adjacency matrices and adjacency lists, and implement fundamental graph traversal algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS). The course will then cover algorithms for finding minimum spanning trees, including Kruskal's and Prim's algorithms, as well as shortest path algorithms like Dijkstra's, Bellman-Ford, and Floyd-Warshall. Additionally, students will explore network flow algorithms, such as Ford-Fulkerson and Edmonds-Karp, which are essential for solving various flow problems in graphs.

The course also addresses **advanced sorting and searching techniques**. Students will implement sorting algorithms like radix sort and understand their applications in specific scenarios. They will also learn about searching techniques such as quickselect, which is used to find the k-th smallest element in an unordered list, and external sorting methods for handling large datasets that do not fit into memory.

Finally, the course introduces students to **amortized analysis** and **persistent data structures**. Amortized analysis helps in understanding the average time complexity of operations over a sequence of operations, while persistent data structures enable access to previous versions of the data, which is useful in applications requiring historical data tracking.

Throughout the course, students will engage in weekly programming assignments, comprehensive projects, and regular lab sessions. These activities are designed to reinforce theoretical concepts through practical implementation and to develop problem-solving skills. The course also emphasizes collaboration, effective communication, and the application of advanced data structures and algorithms in professional and research contexts.

This Advanced Data Structures Laboratory course equips MCA students with the skills and knowledge necessary to tackle complex computational problems, preparing them for advanced studies and professional roles in the field of computer science.

This lab course is designed to complement the theoretical knowledge gained in advanced data structures and algorithms. Through hands-on programming assignments and projects, students will develop a deeper understanding of complex data structures and their applications, as well as enhance their problem-solving skills and algorithmic thinking.

Course Outcomes:

After completion of the course, the students will be able to:

1. Identify and recall various advanced data structures, including AVL trees, Red-Black trees, B-trees, segment trees, interval trees, and their key properties.
2. Explain the functioning and applications of advanced data structures and algorithms, such as graph algorithms, heaps, priority queues, and hashing techniques.
3. Implement advanced data structures and algorithms in a programming language to solve specific computational problems.
4. Utilize appropriate data structures to optimize the performance of applications involving large datasets and complex operations.
5. Evaluate and analyze the time and space complexities of different data structures and algorithms.
6. Critically assess the efficiency and effectiveness of implemented data structures and algorithms through rigorous testing and debugging.
7. Justify the choice of specific data structures and algorithms for solving real-world problems, based on their performance and suitability.
8. Design and develop novel solutions to complex computational problems by combining various advanced data structures and algorithms.

Acknowledgements:

The content we have utilized is solely educational in nature. The copyright proprietors of the materials reproduced in this book have been tracked down as much as possible. The editors apologize for any violation that may have happened, and they will be happy to rectify any such material in later versions of this book.

Question 1: Dynamic Order Statistics

Program Statement:

Implement a data structure to maintain a dynamic set of numbers which allows the operations insert, delete, and find the it smallest number efficiently.

Solution Hints:

- Consider using an Order Statistic Tree, which is a variant of the red-black tree that maintains size information with each node.
- Ensure that each node additionally stores the count of nodes in its left sub tree, which enables quick access to order statistics.
- Implement tree rotations to maintain balanced tree properties after insertions and deletions.

Question 2: Persistent Segment Tree

Program Statement:

Construct a persistent segment tree for range sum queries that allows access to historical data after updates.

Solution Hints:

- Use a segment tree where each update creates a new version of the tree, preserving the old versions.
- Implement node copying during updates to ensure that previous versions remain accessible, resulting in a partially persistent data structure.
- Focus on efficient memory usage by sharing unchanged parts between versions.

Question 3: Van Emden Boas Tree

Program Statement:

Implement a van emde boas tree tree to support search, insert, delete, and predecessor and successor operations in O(log log M) time, where M is the universe size.

Solution Hints:

- Understand the recursive structure of the Van Emde Boas tree, which involves a main tree and several sub-trees.
- Implement auxiliary structures to handle minimum and maximum values specially.
- Carefully manage base cases for recursion, particularly when the tree size becomes small.

Question 4: K Dimensional Tree

Program Statement:
Construct a K-D tree for storing points in a k-dimensional space and implement search and insert operations.

Solution Hints:

- Each node in the K-D tree represents a k-dimensional point. Alternate between dimensions at each level of the tree to partition the space.
- Implement search by recursively navigating the tree based on dimension-wise comparisons with the target point.
- Handle insertions by maintaining the balance of the tree through subtree rotations if necessary.

Question 5: Fibonacci Heap

Program Statement:
Implement a Fibonacci heap to support operations such as insert, extract-min, decrease-key, and merge heaps in amortized constant time.

Solution Hints:

- Focus on the lazy consolidation strategy that delays tree consolidation until necessary during extract-min operations.
- Implement marked nodes and cascading cuts for the decrease-key operation to maintain the heap's properties.
- Use pointers efficiently to manage parent-child and sibling relationships.

Question 6: Treap Implementation

Program Statement:
Create a Treap (Tree Heap) that maintains binary search tree properties with the addition of heap priorities.

Solution Hints:

- Nodes in the treap should contain a key (for the BST property) and a priority (for the heap property).
- Implement rotations to maintain the heap property during insertions and deletions while ensuring the BST properties are not violated.
- Randomly assign priorities to nodes upon insertion to maintain balance probabilistically.

Question 7: Suffix Array and LCP Array Construction

Program Statement:
Construct the suffix array for a given text and use it to build an LCP (Longest Common Prefix) array efficiently.

Solution Hints:

- Use sorting algorithms to build the suffix array based on lexicographical order of suffixes.
- Implement Kasai's algorithm for efficient LCP array construction once the suffix array is built.
- Focus on understanding the relationship between suffix ranks and their LCP values with previous suffixes.

Question 8: Link/Cut Trees

Program Statement:
Implement a link/cut tree to dynamically manage a forest of trees and support operations like link, cut, and path queries.

Solution Hints:

- Use a splay tree to represent each tree in the forest.

- Implement link and cut operations to modify the forest structure, ensuring that the represented trees remain valid.
- Support path queries (such as finding the maximum or sum over a path) by maintaining appropriate aggregates at each node.

Question 9: Heavy-Light Decomposition

Program Statement:
Apply heavy-light decomposition to a given tree to support path queries and updates efficiently.

Solution Hints:

- Decompose the tree into heavy and light edges, ensuring that the path from the root to any node crosses a logarithmic number of light edges.
- Use segment trees or binary indexed trees along heavy paths to support efficient queries and updates.
- Handle queries by combining results from segment trees corresponding to each heavy path involved.

Question 10: 2-3-4 Tree Implementation

Program Statement:
Implement a 2-3-4 tree that supports search, insert, and delete operations while maintaining balanced tree properties.

Solution Hints:

- Understand the properties of 2-3-4 trees where each node can have 2, 3, or 4 children.
- Focus on splitting and merging nodes during insertions and deletions to maintain the tree balance.
- Implement a recursive descent with backtracking to handle the restructuring needed after modifications.

Question 11: Hop croft–Karp Algorithm for Bipartite Matching

Program Statement:
Implement the Hop croft–Karp algorithm to find maximum matching in a bipartite graph.

Solution Hints:

- Use BFS to find an augmenting path and DFS to verify the path and augment the matching.
- Efficiently manage the graph representation to support iterative exploration of edges.
- Focus on updating matching's and unmatched vertices dynamically during each iteration.

Question 12: Geometry Convex Hull Algorithm

Program Statement:
Implement the algorithm to compute the convex hull of a set of points in 2D space.

Solution Hints:

- Consider using Andrew's monotone chain algorithm, which sorts the points and constructs upper and lower hulls.
- Handle edge cases such as collinear points and duplicate points in the input set.
- Ensure the implementation can handle large datasets efficiently.

Question 13: Range Minimum Query with Sparse Table

Program Statement:
Implement a Sparse Table to answer range minimum queries (RMQ) in constant time after pre-processing.

Solution Hints:

- Build the sparse table using dynamic programming to store minimums for overlapping intervals of varying lengths.
- Pre-process the input array to fill the sparse table with appropriate range minimums.
- Handle queries by accessing the recomputed minimums without further computation.

Question 14: Biconnected Components in Graphs

Program Statement:
Identify and extract biconnected components in a graph using Tarjan's algorithm.

Solution Hints:

- Use DFS to find articulation points and bridges.
- Maintain a stack to keep track of visited edges and nodes, and pop from it to form a disconnected component when a back edge is found.
- Implement careful handling of recursion stack and discovery times.

Question 15: Segment Tree with Lazy Propagation

Program Statement:
Implement a segment tree that supports range updates and queries with lazy propagation.

Solution Hints:

- Build the segment tree to store sums or minimums over ranges of an array.
- Use lazy propagation to defer updates to segments of the tree, improving update operations' efficiency.
- Ensure that the lazy values are properly propagated before any node values are accessed or modified.

Question 16: **Aho-Corasick** Algorithm for Pattern Searching

Program Statement:
Implement the Aho-Corasick algorithm for multi-pattern search to efficiently search for multiple patterns simultaneously in a given text.

Solution Hints:

- Construct a tire of all patterns and a failure function for fallbacks similar to the KMP pre-processing.
- Simulate the FSM built from the tree over the text to find occurrences of the patterns.
- Manage the output function to collect entries where patterns match in the text.

Question 17: Dynic's Algorithm for Maximum Flow

Program Statement:
Implement Dynic's algorithm to compute the maximum flow in a flow network.

Solution Hints:

- Use BFS to construct a level graph and DFS to find blocking flows.
- Maintain a residual graph to track capacities after each flow adjustment.
- Iterate the process until no augmenting path can be found in the level graph.

Question 18: Centroid Decomposition of Trees

Program Statement:
Implement centroid decomposition on a tree to solve problems related to path queries and updates efficiently.

Solution Hints:

- Identify the centroid of the tree recursively and decompose the tree around it.
- Use a combination of data structures like segment trees or BITs to manage path queries in each sub tree.
- Understand the properties of centroids to optimize query and update operations across the decomposed tree.

Question 19: Palindrome Tree

Program Statement:
Build a palindrome tree that efficiently counts different palindromes in a string and answers queries about them.

Solution Hints:

- Initialize nodes corresponding to the imaginary palindromes **-1** and **0**.
- Extend the tree with each new character by considering the longest suffix palindrome of the current string.

- Maintain links to manage the tree structure and support efficient palindrome counting and querying.

Question 20: Integer Factorization with Pollard's Rho Algorithm

Program Statement:
Implement Pollard's Rho algorithm for integer factorization, aimed at finding non-trivial factors of a composite number.

Solution Hints:

- Use the cycle-finding Floyd's Tortoise and Hare approach to detect cycles.
- Apply the factorization function iteratively to attempt to identify a divisor of the number.
- Handle cases where the algorithm fails to find a factor by switching parameters and retrying.