Bachelor of Computer Applications (BCA)

Data Structures Using 'C' Lab (OBCACO207P24)

Self-Learning Material (SEM -II)

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

COURSE INTRODUCTION

"Clean code always looks like it was written by someone who cares."
- Robert C. Martin

The "Data Structures Using C Lab" course offers an immersive, hands-on experience designed to complement theoretical knowledge of data structures with practical coding skills using the C programming language. This lab course focuses on applying fundamental data structures concepts through practical implementation, allowing students to develop, test, and refine their skills in a real-world programming environment.

Students will start by engaging with basic data structures, such as arrays and linked lists. They will write C programs to implement these structures, learning how to handle memory management and pointers effectively, which are crucial aspects of working with data structures in C. The lab sessions will involve creating dynamic data structures, handling insertion, deletion, and traversal operations, and understanding their underlying mechanisms.

As the course progresses, students will explore more advanced data structures, including stacks, queues, trees, and graphs. They will implement these structures in C, developing code to manage their operations and understand their use cases. Practical exercises will involve implementing stack and queue operations, designing binary trees and balanced trees, and working with various graph algorithms for traversal and searching.

The lab also emphasizes the implementation of algorithms associated with these data structures. Students will write and test algorithms for searching, sorting, and manipulating data, evaluating their performance and efficiency in different scenarios. They will gain hands-on experience in optimizing their code and understanding the impact of different data structures on algorithm performance.

Throughout the course, students will utilize debugging and performance analysis tools to test their implementations. They will learn to identify and resolve issues in their code, analyze the efficiency of their algorithms, and ensure correctness and robustness in their programs.

The collaborative nature of the lab encourages students to work together, share insights, and solve problems collectively. This environment fosters deeper understanding and provides a platform for discussing various approaches to implementing and managing data structures in C.

By the end of the course, students will have a thorough practical understanding of data structures and algorithms in C. They will be equipped with the skills to design, implement, and optimize data structures for a variety of programming tasks, and will be well-prepared for more advanced topics in computer science and software development.

Course Outcomes:

At the completion of the course, a student will be able to:

1. Recall how to analyze algorithms and estimate their worst-case and average-case behavior (in easy cases).
2. Illustrate a given problem and develop an algorithm to solve the problem
3. Determine the fundamental data structures and with the manner in which data structures can be best be implemented
4. Design the description of algorithms in both functional and procedural styles.
5. Implement theoretical knowledge in practice (via the practical component of the course).
6. Create algorithms or programs code or segment that contains iterative constructs and analyze the code segment

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.

Assignment 1: Implementing a Linked List

Program Statement: Write a C program to create a singly linked list of integers. The program should include functions to:

1. "Insert node at the beginning".
2. "Insert node at the end".
3. "Delete node from the beginning".
4. "Delete node from the end".
5. "Display the list".

Solution Description: To solve this problem, you will define a structure for the node, which contains an integer data field and a pointer to the next node. The insertion functions will handle the allocation of new nodes and adjusting the pointers accordingly. Deletion functions will free the appropriate nodes and update pointers to maintain the list integrity. The display function will traverse the list and print each node's data. This assignment helps in understanding dynamic memory allocation, pointer manipulation, and linked list operations.

Assignment 2: Implementing a Stack Using Arrays

Program Statement: Create a stack data structure using arrays. Implement the following operations:

1. Push "add an element to the stack".
2. Pop "remove an element from the stack".
3. Peek "get the top element of the stack without removing it".
4. Display "print all elements in the stack".

Solution Description: In this assignment, you'll use an array to store stack elements and an integer variable to track the top element's index. The push function will add an element to the top if the stack is not full, while the pop function will remove the top element if the stack is not empty. The peek function will return the top element, and the display function will iterate through the stack to print all elements. This exercise reinforces the concepts of stack operations and boundary conditions.

Assignment 3: Implementing a Queue Using Linked List

Program Statement: Write a C program to implement a queue using a linked list. Include functions for:

1. Enqueue “add an element to the queue”.
2. Dequeue “remove an element from the queue”.
3. Display “print all elements in the queue”.

Solution Description: A queue implemented using a linked list will involve a front and rear pointer. The enqueue function will add nodes at the rear end, while the dequeue function will remove nodes from the front end. The display function will traverse from the front to the rear of the queue, printing each element's data. This assignment helps in understanding the FIFO (First In First Out) principle and managing dynamic data structures.

Assignment 4: Binary Search Tree Operations

Program Statement: Develop a C program to create and manage a binary search tree (BST). Implement functions to:

1. Insert a node into the BST.
2. Delete a node from the BST.
3. Perform in-order traversal.
4. Perform pre-order traversal.
5. Perform post-order traversal.

Solution Description: The BST will be defined using a node structure containing data and left and right child pointers. The insertion function will recursively place the new node in the correct position to maintain the BST properties. The deletion function will handle three cases: “deleting a leaf node”, “a node with one child”, and “a node with two children”. The traversal functions will recursively visit nodes in their respective orders. This assignment covers recursive algorithms and tree data structure manipulations.

Assignment 5: Circular Linked List

Program Statement: Write a C program to create a circular linked list of integers. Implement functions to:

1. "Insert node at the beginning".
2. "Insert node at the end".
3. "Delete node from the beginning".
4. "Delete node from the end".
5. "Display the list".

Solution Description: A circular linked list is a linked list where the last node points to the first node. The insertion and deletion operations need to handle the circular nature, ensuring the next pointers maintain the circle. The display function will start from the head node and continue until it loops back to the head. This assignment emphasizes understanding circular references and pointer adjustments in linked lists.

Assignment 6: Doubly Linked List

Program Statement: Create a doubly linked list in C with the following operations:

1. "Insert node at the beginning".
2. "Insert node at the end".
3. "Delete node from the beginning".
4. "Delete node from the end".
5. Display the list forward and backward.

Solution Description: A doubly linked list has nodes with pointers to both the next and previous nodes. This enables traversal in both directions. The insertion and deletion functions must update both the next and previous pointers accordingly. Display functions will show the list from head to tail and tail to head, demonstrating bidirectional traversal.

Assignment 7: Implementing a Stack Using Linked List

Program Statement: Implement a stack using a linked list in C. Include functions for:

1. Push “add an element to the stack”.
2. Pop “remove an element from the stack”.
3. Peek “get the top element of the stack without removing it”.
4. Display “print all elements in the stack”.

Solution Description: The stack will be implemented using a singly linked list where the head represents the top of the stack. The push function will insert nodes at the beginning, and the pop function will remove nodes from the beginning. The peek function returns the data of the head node. The display function traverses from head to the end, printing each node's data. This assignment helps understand the dynamic implementation of stack operations.

Assignment 8: Queue Using Arrays

Program Statement: Create a queue using arrays in C. Implement functions for:

1. Enqueue “add an element to the queue”.
2. Dequeue “remove an element from the queue”.
3. Peek “get the front element of the queue without removing it”.
4. Display the queue.

Solution Description: A queue implemented with an array involves maintaining two pointers, front and rear. The enqueue function adds elements at the rear, while the dequeue function removes elements from the front. The peek function returns the front element, and the display function iterates from front to rear to print all elements. This exercise focuses on understanding the circular nature of queues in array implementation.

Assignment 9: Priority Queue Using Linked List

Program Statement: Implement a priority queue using a linked list in C. The program should include:

1. Insertion of elements with a given priority.
2. Deletion of the highest priority element.
3. Display the queue.

Solution Description: A priority queue organizes elements based on their priority. Each node will have a priority value in addition to the data. The insertion function places nodes in the correct position to maintain order based on priority. The deletion function removes the node with the highest priority. The display function prints elements from the highest to lowest priority. This assignment helps understand priority-based data structures and their operations.

Assignment 10: Circular Queue Using Arrays

Program Statement: Implement a circular queue using arrays in C. Include functions for:

1. Enqueue "add an element to the queue".
2. Dequeue "remove an element from the queue".
3. Peek "get the front element of the queue without removing it".
4. Display the queue.

Solution Description: A circular queue allows the array to be used efficiently by connecting the end back to the beginning. The enqueue and dequeue functions will handle the circular nature by using modulo operations. The peek function returns the front element, and the display function prints elements from front to rear, wrapping around if necessary. This assignment emphasizes the implementation and benefits of circular queues.

Assignment 11: Implementing a Deque Using Arrays

Program Statement: Create a double-ended queue (deque) using arrays in C. Implement functions to:

1. Insert an element at the front.
2. Insert an element at the rear.
3. Delete an element from the front.
4. Delete an element from the rear.
5. Display the deque.

Solution Description: A deque allows insertion and deletion at both ends. Using an array, maintain front and rear pointers and adjust them for operations. The insertions and deletions will handle both ends, and the display function will print elements considering the circular nature if implemented. This assignment helps in understanding the flexibility and operations of deques.

Assignment 12: Binary Search Tree - Find Minimum and Maximum

Program Statement: Extend your binary search tree (BST) implementation to include functions for:

1. Finding the minimum value node.
2. Finding the maximum value node.

Solution Description: To find the minimum value node in a BST, traverse the leftmost path starting from the root. For the maximum value node, traverse the rightmost path. These functions will return the nodes with the respective minimum and maximum values. This assignment reinforces understanding of the BST properties and traversal techniques.

Assignment 13: AVL Tree Implementation

Program Statement: Implement an AVL tree in C, which is a self-balancing binary search tree. Include functions for:

1. Inserting a node.
2. Deleting a node.
3. Performing in-order traversal.

Solution Description: An AVL tree maintains balance by ensuring the height difference between left and right subtrees of any node is at most one. The insertion and deletion functions will include rotations (single and double) to maintain balance after each operation. The in-order traversal will display the elements in sorted order. This assignment covers advanced tree operations and self-balancing mechanisms.

Assignment 14: Graph Representation Using Adjacency Matrix

Program Statement: Write a C program to represent a graph using an adjacency matrix. Include functions for:

1. Adding an edge.
2. Removing an edge.
3. Displaying the adjacency matrix.

Solution Description: An adjacency matrix is a 2D array used to represent a graph, where each cell (i, j) indicates the presence or absence of an edge between vertices i and j. The add edge function will set the appropriate matrix cell to 1, and the remove edge function will reset it to 0. The display function will print the matrix. This assignment helps understand graph representations and matrix operations.

Assignment 15: Depth-First Search (DFS) in Graph

Program Statement: Implement the depth-first search (DFS) algorithm for a graph represented using an adjacency list in C. Include a function to perform DFS starting from a given vertex.

Solution Description: DFS explores as far as possible along each branch before backtracking. Using an adjacency list representation, the DFS function will use a stack (either explicitly or through recursion) to explore vertices, marking them as visited. This assignment emphasizes understanding graph traversal and recursion techniques.

Assignment 16: Breadth-First Search (BFS) in Graph

Program Statement: Implement the breadth-first search (BFS) algorithm for a graph represented using an adjacency list in C. Include a function to perform BFS starting from a given vertex.

Solution Description: BFS explores all neighbors at the present depth level before moving on to nodes at the next depth level. Using an adjacency list, the BFS function will use a queue to manage the traversal order. This assignment covers graph traversal using BFS and queue operations.

Assignment 17: Implementing a Hash Table Using Open Addressing

Program Statement: Create a hash table in C using open addressing with linear probing for collision resolution. Include functions for:

1. Inserting a key-value pair.
2. Searching for a key.
3. Deleting a key.
4. Displaying the hash table.

Solution Description: Open addressing handles collisions by probing for the next available slot. Linear probing checks subsequent slots until an empty one is found. The insert, search, and delete functions will use this strategy to manage collisions. The display function will print the hash table. This assignment emphasizes hash table operations and collision resolution techniques.

Assignment 18: Implementing a Trie

Program Statement: Write a C program to implement a trie (prefix tree) for storing strings. Include functions for:

1. Inserting a string.
2. Searching for a string.
3. Deleting a string.
4. Displaying the trie.

Solution Description: A trie is a tree-like data structure used for efficient storage and retrieval of strings. Each node represents a character, and paths down the tree represent strings. The insert function will add nodes for each character of the string. The search function will traverse the nodes to find the string. The delete function will remove nodes if the string exists. The display function will print all strings in the trie. This assignment helps understand trie operations and string manipulations.

Assignment 19: Implementing a Heap (Max-Heap)

Program Statement: Create a max-heap data structure using arrays in C. Implement functions for:

1. Inserting an element.
2. Deleting the maximum element.
3. Displaying the heap.

Solution Description: A max-heap is a complete binary tree where each node is greater###

Assignment 19: Implementing a Heap (Max-Heap)

Program Statement: Create a max-heap data structure using arrays in C. Implement functions for:

1. Inserting an element.
2. Deleting the maximum element.
3. Displaying the heap.

Solution Description: A max-heap is a complete binary tree where each node is greater than or equal to its children. It is usually implemented using an array where for a node at index **i**, its children are at indices **2i+1** and **2i+2**, and its parent is at **(i-1)/2**. The insert function will add the new element at the end of the array and "heapify" up to maintain the heap property. The delete function will remove the root (maximum element), replace it with the last element, and "heapify" down to restore the heap property. The display function will print the array representation of the heap. This assignment covers tree-based data structures and array manipulations.

Assignment 20: Implementing a Min-Heap

Program Statement: Create a min-heap data structure using arrays in C. Implement functions for:

1. Inserting an element.
2. Deleting the minimum element.
3. Displaying the heap.

Solution Description: A min-heap is similar to a max-heap but each node is less than or equal to its children. The array-based implementation follows the same indexing rules. The insert function will add the element and "heapify" up to maintain the heap property. The delete function will remove the root (minimum element), replace it with the last element, and "heapify" down. The display function will print the array representation of the heap. This assignment helps understand heap structures and their applications.

Assignment 21: Implementing a Hash Table Using Chaining

Program Statement: Create a hash table in C using chaining for collision resolution. Include functions for:

1. Inserting a key-value pair.
2. Searching for a key.
3. Deleting a key.
4. Displaying the hash table.

Solution Description: Chaining resolves collisions by maintaining a list of all elements that hash to the same index. Each array index points to a linked list of elements. The insert function adds elements to the appropriate list, the search function traverses the list to find the key, and the delete function removes the key from the list. The display function prints all elements in the hash table. This assignment emphasizes understanding hash functions and collision handling through chaining.

Assignment 22: Implementing a Binary Heap Sort

Program Statement: Write a C program to sort an array of integers using heap sort. Implement the heap sort algorithm which involves building a max-heap and then sorting the array.

Solution Description: Heap sort is an efficient sorting algorithm that uses a binary heap. First, build a max-heap from the input array. Then repeatedly remove the maximum element (root of the heap) and rebuild the heap with the remaining elements. The process continues until all elements are sorted. This assignment covers the concepts of heap construction and sorting algorithms.

Assignment 23: Implementing a Red-Black Tree

Program Statement: Create a red-black tree in C. Include functions for:

1. Inserting a node.
2. Deleting a node.
3. Searching for a node.
4. Performing in-order traversal.

Solution Description: A red-black tree is a self-balancing binary search tree where each node has an extra bit for color (red or black). It ensures balance through rotations and color changes during insertions and deletions. The insert and delete functions maintain the tree properties by performing necessary rotations and color adjustments. The search function finds nodes based on key values. The in-order traversal function prints nodes in ascending order. This assignment covers advanced tree operations and self-balancing techniques.

Assignment 24: Implementing a Fibonacci Heap

Program Statement: Implement a Fibonacci heap in C. Include functions for:

1. Inserting an element.
2. Extracting the minimum element.
3. Decreasing a key.
4. Displaying the heap.

Solution Description: A Fibonacci heap is a collection of heap-ordered trees. It supports insertions, deletions, and decreases key operations efficiently. The insert function adds a new node, the extract-min function removes the minimum node and consolidates the trees, and the decrease key function decreases the key value of a node and possibly performs a cut operation. The display function will show the structure of the heap. This assignment focuses on advanced heap data structures and their operations.

Assignment 25: Implementing a Splay Tree

Program Statement: Create a splay tree in C. Include functions for:

1. Inserting a node.
2. Searching for a node.
3. Deleting a node.
4. Performing in-order traversal.

Solution Description: A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. The insert, search, and delete functions perform rotations to splay the accessed node to the root. The in-order traversal function will display the elements in ascending order. This assignment covers self-adjusting trees and their advantages.

Assignment 26: Implementing an Adjacency List for Graphs

Program Statement: Implement a graph using an adjacency list in C. Include functions for:

1. Adding an edge.
2. Removing an edge.
3. Displaying the adjacency list.

Solution Description: An adjacency list represents a graph where each vertex has a list of adjacent vertices. The add edge function updates the lists for the involved vertices, while the remove edge function deletes the corresponding entries from the lists. The display function prints each vertex and its adjacent vertices. This assignment focuses on efficient graph representations and operations.