**Welcome to Data Structures Lab!**

This course, CS29001, is a B.Tech discipline with a 0-0-2-1 L-T-P-Cr structure, meaning it's a practical lab course with no lectures, 2 practical hours per week, and 1 credit2. Your instructor is Arghya Kundu3.

Data structures are essentially ways of organizing and storing data in a computer so that it can be accessed and modified efficiently. Algorithms are a set of step-by-step instructions to solve a problem. In this lab, you'll be implementing these concepts using code.

Let's go through the course content day by day, as if you know nothing about coding.

Lab-1 Assignments: Introduction (Structure, Pointer, Dynamic Memory Allocation and De-allocation) 444444444

This lab will introduce you to fundamental concepts in C programming that are crucial for understanding data structures.

1.1 Program to compare two numbers using "call by address" 5

* **What you need to know:**
  + **Variables:** Think of a variable as a named box in the computer's memory where you can store a value (like a number). For example, int x = 10; creates a box named x and puts the number 10 inside it.
  + **Functions:** Functions are blocks of code that perform a specific task. You can "call" a function to execute that task.
  + **Call by Value vs. Call by Address (Pointers):**
    - **Call by Value:** When you pass variables to a function by value, thefunction gets a *copy* of the variables. Any changes made to these copies inside the function *do not* affect the original variables outside the function.
    - **Call by Address (Pointers):** This is where it gets interesting! A **pointer** is a special type of variable that stores the *memory address* of another variable. Instead of passing the value itself, you pass the memory address of the variable to the function. This allows the function to directly access and modify the *original* variable's value.
      * Think of it like giving someone the actual street address of your house instead of a copy of your house. They can then go to your actual house and change things inside.
      * You'll use the & operator to get the address of a variable (e.g., &number) and the \* operator to access the value at a memory address (e.g., \*ptr).
* **What you'll do:** You'll write a program where a function compares two numbers, but instead of receiving the numbers directly, it will receive their memory addresses. This means the function can potentially modify the numbers, though in this specific problem, it's just for comparison.

1.2 Program to create a dynamic array and sum its prime elements 6

* **What you need to know:**
  + **Arrays:** An array is a collection of elements of the same data type stored in contiguous memory locations. Imagine a row of mailboxes, all holding letters.
  + **Static vs. Dynamic Memory Allocation:**
    - **Static:** When you declare a normal array like int arr[10];, its size is fixed at the time the program is compiled. You can't change it while the program is running.
    - **Dynamic:** This is super powerful! Dynamic memory allocation allows you to request memory space *while the program is running*. This is useful when you don't know the exact size of an array beforehand. You'll typically use functions like malloc() (memory allocate) and free() (to de-allocate, or release, the memory when you're done with it to prevent "memory leaks").
  + **Prime Numbers:** A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself (e.g., 2, 3, 5, 7, 11).
  + **Functions (again!):** You'll create a function to calculate the sum of prime elements.
* **What you'll do:** You'll create an array whose size is determined by the user during program execution. Then, you'll find all the prime numbers within that array and add them up. Finally, you'll release the memory you allocated.

1.3 Program to store employee information using "array of structures" 7

* **What you need to know:**
  + **Structures (Structs):** A structure is a way to group different types of related data under a single name. Think of it as a blueprint for a custom data type. For example, you might create an Employee structure that contains Emp-id (an integer), Name (a string of characters), Designation (another string), basic\_salary (a number), hra% (a percentage), and da% (another percentage).
  + **Array of Structures:** Just like you can have an array of integers, you can have an array of structures. This means you can store information for multiple employees in one organized unit.
  + **Gross Salary Calculation:** You'll need to know how to calculate gross salary (basic salary + HRA + DA).
* **What you'll do:** You'll define an Employee structure. Then, you'll create an array of these Employee structures to store information for several employees. You'll take input for each employee's details and then display their information along with their calculated gross salary.

1.4 Menu-driven program for complex number operations using structures 8

* **What you need to know:**
  + **Complex Numbers:** A complex number is a number that can be expressed in the form a+bi, where a and b are real numbers, and i is the imaginary unit, satisfying i2=−1.
  + **Structures:** You'll define a structure to represent a complex number, likely with two members: one for the real part and one for the imaginary part.
  + **Menu-Driven Program:** This means your program will display options to the user (like "1. Addition", "2. Multiplication") and perform the chosen operation based on the user's input.
  + **Function Calls (again!):**
    - **Addition (call by value):** You'll pass copies of the complex numbers to the addition function.
    - **Multiplication (call by address):** You'll pass the memory addresses of the complex numbers to the multiplication function, allowing direct modification if needed (though for multiplication, you'd likely return a new complex number).
* **What you'll do:** You'll create a Complex structure. Your program will present a menu. Based on the user's choice, it will perform either addition (passing by value) or multiplication (passing by address) of two complex numbers and display the result.

Lab-2 Assignments: Arrays 9

This lab focuses on performing various operations on arrays.

2.1 Menu-based operations on a 1-D array 10

* **What you need to know:**
  + **1-D Array:** A simple list of elements.
  + **Menu-Driven Program:** As before, you'll present options to the user.
  + **Array Operations:**
    - **Insert:** Adding an element at a specific position in the array. This usually requires shifting existing elements to make space.
    - **Delete:** Removing an element from a specific position. This usually involves shifting subsequent elements to fill the gap.
    - **Linear Search:** A simple search algorithm that checks each element in the array sequentially until a match is found or the end of the array is reached.
    - **Traversal:** Visiting each element in the array, typically to display it.
* **What you'll do:** You'll create a 1-D array and then implement functions for inserting an element, deleting an element, searching for an element using linear search, and displaying all elements in the array. The user will choose which operation to perform from a menu.

2.2 Operations on a square matrix 11

* **What you need to know:**
  + **Matrix (2-D Array):** A rectangular array of numbers, organized in rows and columns. A *square matrix* has the same number of rows and columns.
  + **Non-zero elements:** Simply counting how many elements in the matrix are not zero.
  + **Upper Triangular Matrix:** In a square matrix, an upper triangular matrix has all elements *below* the main diagonal equal to zero. The main diagonal runs from the top-left corner to the bottom-right corner.
  + **Elements above and below the main diagonal:** Identifying and displaying elements that are immediately adjacent to the main diagonal.
* **What you'll do:** You'll write a program that takes a square matrix as input and performs these operations:
  + Counts and displays the number of non-zero elements.
  + Displays the upper triangular matrix (showing only the relevant elements).
  + Displays the elements directly above and below the main diagonal.

2.3 Representing a sparse matrix in 3-tuple format 12

* **What you need to know:**
  + **Sparse Matrix:** A matrix where most of the elements are zero. Storing all those zeros is inefficient.
  + **3-tuple format:** A common way to represent a sparse matrix efficiently. Instead of storing all elements, you only store the non-zero elements along with their row, column, and value. So, each non-zero element is represented as a "tuple" (row, column, value). The first row of the 3-tuple format usually stores the total number of rows, columns, and non-zero elements in the original sparse matrix.
* **What you'll do:** You'll take a sparse matrix as input and convert it into its 3-tuple representation, displaying the result.

Lab-3 Assignments: Sparse Matrix & Polynomial Representation 13

This lab continues with sparse matrices and introduces polynomial representation.

3.1 Transpose of a sparse matrix in 3-tuple format 14

* **What you need to know:**
  + **Transpose of a Matrix:** Swapping the rows and columns of a matrix. If the original matrix has an element at (row, column), its transpose will have that element at (column, row).
  + **3-tuple format:** You'll be working with the 3-tuple representation from the previous lab. To transpose it, you essentially swap the row and column values in each tuple.
* **What you'll do:** You'll take a sparse matrix in 3-tuple format as input and then produce its transpose, also in 3-tuple format.

3.2 Addition of two sparse matrices in 3-tuple format 15

* **What you need to know:**
  + **Matrix Addition:** To add two matrices, you add their corresponding elements. For sparse matrices in 3-tuple format, this means efficiently combining the non-zero elements. If two matrices have non-zero elements at the same (row, column) position, their values are added. If an element exists in one matrix but not the other at a particular position, it's carried over to the result.
* **What you'll do:** You'll take two sparse matrices (in 3-tuple format) as input and calculate their sum, displaying the resultant sparse matrix in 3-tuple format.

3.3 Polynomial representation using 1-D array and addition 16

* **What you need to know:**
  + **Polynomial:** An expression consisting of variables and coefficients, involving only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables (e.g., 3x2+2x+4).
  + **Representation using 1-D Array:** You can represent a polynomial using a 1-D array where each index corresponds to the exponent of the variable, and the value at that index is the coefficient. For example, for 4+2x+3x2, an array might be [4, 2, 3] where index 0 is for x0, index 1 for x1, and so on.
  + **Polynomial Addition:** To add two polynomials, you add the coefficients of terms with the same exponent.
* **What you'll do:** You'll represent polynomials using 1-D arrays. You'll take two polynomial equations as input and then implement a function to add them, displaying the resultant polynomial.

Lab-4 Assignments: Single Linked List 17

This lab introduces you to one of the most fundamental dynamic data structures: the single linked list.

* **What you need to know:**
  + **Linked List:** Unlike arrays, linked lists store elements at non-contiguous memory locations. Instead, each element (called a "node") contains two parts: the actual data and a "pointer" (or reference) to the next node in the sequence.
  + **Single Linked List:** Each node points only to the *next* node. The last node points to NULL (indicating the end of the list).
  + **Node Structure:** You'll define a structure for a node, typically containing data and a next pointer (which points to another node structure).
  + **Head Pointer:** A special pointer that always points to the first node in the linked list. If the list is empty, the head pointer is NULL.

4.1 Menu-based operations on a single linked list 18

* **What you'll do:** You'll implement a menu-driven program to perform the following operations on a single linked list:
  + **Insert a node at a specific position:** This involves creating a new node and correctly adjusting the pointers of the surrounding nodes.
  + **Delete an element from a specific position:** This involves finding the node to be deleted and adjusting the pointers of the preceding and succeeding nodes to bypass it, then freeing the deleted node's memory.
  + **Count nodes:** Simply iterating through the list and incrementing a counter for each node.
  + **Traverse the linked list:** Visiting each node in the list, typically to display its data.

4.2 Advanced operations on a single linked list 19

* **What you'll do:** In addition to the operations from 4.1, you'll implement:
  + **Search an element:** Iterating through the list to find a specific data value.
  + **Sort the list in ascending order:** This can be done by various sorting algorithms adapted for linked lists (e.g., bubble sort, insertion sort).
  + **Reverse the list:** Changing the next pointers of each node so that the list order is reversed. This typically involves iterating through the list and adjusting pointers.

4.3 Polynomial equation representation and addition using single linked list 20

* **What you need to know:**
  + **Polynomial Representation with Linked List:** Each node in the linked list can represent a term in the polynomial, storing its coefficient and exponent. For example, a node could be (coefficient, exponent). The nodes would be linked in descending or ascending order of exponents.
* **What you'll do:** You'll represent polynomial equations using single linked lists. Each node will hold a coefficient and its corresponding exponent. You will then implement a function to add two such polynomial equations and display the resulting polynomial.

Lab-5 Assignments: Double and Circular Linked List 21

This lab introduces variations of linked lists.

5.1 Menu-based operations on a double linked list 22

* **What you need to know:**
  + **Double Linked List:** Each node in a double linked list has three parts: data, a next pointer (to the next node), and a prev (previous) pointer (to the preceding node). This allows for traversal in both forward and backward directions.
* **What you'll do:** You'll implement a menu-driven program for a double linked list, including:
  + **Insert an element at a specific position:** This involves adjusting four pointers (the next and prev pointers of the new node, and the next and prev pointers of its surrounding nodes).
  + **Delete an element from a specific position:** Similar to insertion, but you'll adjust the pointers of the surrounding nodes to bypass the deleted node, and then free its memory.
  + **Traverse the list:** Displaying elements, potentially in both forward and backward directions.

5.2 Create and display a circular linked list 23

* **What you need to know:**
  + **Circular Linked List:** In a circular linked list, the next pointer of the last node points back to the first node (head) of the list, forming a circle. This means you can traverse the entire list starting from any node.
* **What you'll do:** You'll write a program to create a circular linked list and then display its elements.

5.3 Sparse matrix representation using header single linked list 24

* **What you need to know:**
  + **Header Single Linked List:** This is a variation where the first node (header node) doesn't store actual data but contains information about the list itself (e.g., number of rows, columns, non-zero elements in a sparse matrix context). Subsequent nodes store the actual sparse matrix elements. This can simplify some operations.
* **What you'll do:** You'll represent a given sparse matrix using a header single linked list and display its contents. This would be a more complex representation than the 3-tuple format, involving nodes for rows and then linked lists for non-zero elements within each row.

Lab-6 Assignments: Stacks 25

This lab introduces stacks, a Last-In, First-Out (LIFO) data structure.

* **What you need to know:**
  + **Stack:** Imagine a stack of plates. You can only add a plate to the top, and you can only remove a plate from the top. The last plate you put on is the first one you take off.
  + **LIFO (Last-In, First-Out):** The last element added to the stack is the first one to be removed.
  + **Key Stack Operations:**
    - **Push:** Adding an element to the top of the stack.
    - **Pop:** Removing the top element from the stack.
    - **IsEmpty:** Checking if the stack has any elements.
    - **IsFull:** Checking if the stack has reached its maximum capacity (relevant when using an array-based implementation).
    - **Display/Traverse:** Showing the elements in the stack.

6.1 Menu-driven program: Stack using Array 26

* **What you'll do:** You'll implement a stack using a fixed-size array. You'll create a menu to perform:
  + Push an element.
  + Pop an element.
  + Check if the stack IsEmpty.
  + Check if the stack IsFull.
  + Display the stack elements.

6.2 Menu-driven program: Stack using Linked List 27

* **What you'll do:** You'll implement a stack using a single linked list. This implementation is more flexible as it can grow dynamically. You'll perform:
  + Push an element.
  + Pop an element.
  + Check if the stack IsEmpty.
  + Display the stack elements.

6.3 Infix to postfix expression conversion using stack 28

* **What you need to know:**
  + **Infix Expression:** The way we normally write mathematical expressions (e.g., A + B).
  + **Postfix Expression (Reverse Polish Notation):** Operators come *after* their operands (e.g., AB+). This notation is useful because it doesn't require parentheses and is easier for computers to evaluate.
  + **Algorithm for Infix to Postfix:** This involves using a stack to handle operators and parentheses based on their precedence.
* **What you'll do:** You'll write a program that takes an infix expression as input and converts it into its equivalent postfix expression using a stack.

Lab-7 Assignments: Linear Queues & Circular Queue 29

This lab introduces queues, a First-In, First-Out (FIFO) data structure.

* **What you need to know:**
  + **Queue:** Imagine a line at a ticket counter. The first person in line is the first one to get served.
  + **FIFO (First-In, First-Out):** The first element added to the queue is the first one to be removed.
  + **Key Queue Operations:**
    - **Enqueue:** Adding an element to the rear (back) of the queue.
    - **Dequeue:** Removing an element from the front of the queue.
    - **IsEmpty:** Checking if the queue has any elements.
    - **IsFull:** Checking if the queue has reached its maximum capacity (relevant for array-based implementations).
    - **Traverse:** Displaying the elements in the queue.

7.1 Menu-driven program: Linear Queue using Array 30

* **What you'll do:** You'll implement a linear queue using a fixed-size array. You'll manage front and rear pointers/indices. You'll provide a menu for:
  + Enqueue.
  + Dequeue.
  + IsEmpty.
  + IsFull.
  + Traverse.

7.2 Menu-driven program: Linear Queue using Single Linked List 31

* **What you'll do:** You'll implement a linear queue using a single linked list. This allows for dynamic resizing. You'll provide a menu for:
  + Enqueue.
  + Dequeue.
  + IsEmpty.
  + Traverse.

7.3 Menu-driven program: Circular Queue using Array 32

* **What you need to know:**
  + **Circular Queue:** A linear queue implemented in a circular fashion within an array. This overcomes the problem of wasted space in a linear array-based queue when dequeue operations occur frequently. When the rear reaches the end of the array, it "wraps around" to the beginning if there's space.
* **What you'll do:** You'll implement a circular queue using an array. You'll manage front and rear pointers that wrap around. You'll provide a menu for:
  + Enqueue.
  + Dequeue.
  + IsEmpty.
  + IsFull.
  + Traverse.

Lab-8 Assignments: Deques & Priority Queue 33

This lab introduces more specialized queue types.

8.1 Menu-driven program: Deques (Input/Output restricted) using static array 34

* **What you need to know:**
  + **Deque (Double-Ended Queue):** A queue that allows insertion and deletion from *both* ends (front and rear).
  + **Input-Restricted Deque:** Elements can only be inserted at one end (e.g., rear), but can be deleted from both ends (front or rear).
  + **Output-Restricted Deque:** Elements can be inserted at both ends (front or rear), but can only be deleted from one end (e.g., front).
  + **Static Array:** You'll implement this using a fixed-size array.
  + **Peek:** An operation to look at the front or rear element without removing it.
* **What you'll do:** You'll implement a menu-driven program that demonstrates both input-restricted and output-restricted Deques using a static array. You'll support operations like Enqueue (insert), Dequeue (delete), Peek, IsEmpty, and IsFull.

8.2 Menu-driven program: Priority Queue using linked list 35

* **What you need to know:**
  + **Priority Queue:** A special type of queue where each element has a "priority." Elements with higher priority are served before elements with lower priority. If elements have the same priority, they are served based on their order of arrival (FIFO).
  + **Linked List Implementation:** You'll typically insert elements into the linked list in sorted order of their priority.
* **What you'll do:** You'll implement a menu-driven program for a priority queue using a linked list. You'll support:
  + Enqueue (insert an element with a given priority).
  + Dequeue (remove the element with the highest priority).
  + Traverse (display the elements in priority order).

Lab-9 Assignments: Trees (Binary Search Tree) 36

This lab introduces trees, particularly binary search trees.

* **What you need to know:**
  + **Tree (Data Structure):** A hierarchical data structure consisting of nodes connected by edges. Think of an upside-down tree with a root at the top and branches extending downwards.
  + **Root Node:** The topmost node in a tree.
  + **Child Node:** A node directly connected to another node (its parent) one level below it.
  + **Parent Node:** A node that has one or more child nodes.
  + **Leaf Node:** A node that has no children.
  + **Binary Tree:** A tree data structure in which each node has at most two children, referred to as the left child and the right child.
  + **Binary Search Tree (BST):** A special type of binary tree where for every node:
    - All values in its left subtree are *less than* the node's value.
    - All values in its right subtree are *greater than* the node's value.
    - This property makes searching, insertion, and deletion very efficient.
  + **Linked List Implementation of a BST:** Each node will contain data, a pointer to its left child, and a pointer to its right child.

9.1 Menu-driven program: BST operations (traversals, search) 37

* **What you'll do:** You'll create a BST using a linked list and implement a menu-driven program for:
  + **Preorder Traversal:** Visit the root, then traverse the left subtree, then traverse the right subtree. (Root -> Left -> Right)
  + **Postorder Traversal:** Traverse the left subtree, then traverse the right subtree, then visit the root. (Left -> Right -> Root)
  + **Inorder Traversal:** Traverse the left subtree, then visit the root, then traverse the right subtree. For a BST, inorder traversal always gives you the elements in sorted (ascending) order. (Left -> Root -> Right)
  + **Search an element:** Efficiently find if a given element exists in the BST, using the BST property.

9.2 More BST operations (insert, largest/smallest, height, leaf count) 38

* **What you'll do:** In addition to the above, you'll implement:
  + **Insert an element:** Add a new node to the BST while maintaining its properties.
  + **Display the largest element:** This will be the rightmost node in the BST.
  + **Display the smallest element:** This will be the leftmost node in the BST.
  + **Height of a node:** The length of the longest path from the node to a leaf.
  + **Count number of leaf nodes:** Recursively counting nodes with no children.

9.3 Deletion of an element in BST 39

* **What you'll do:** You'll implement the crucial and often tricky operation of deleting a node from a BST while maintaining the BST properties. This involves different cases depending on whether the node to be deleted has 0, 1, or 2 children.

Lab-10 Assignments: Graph (Un-directed Graph) 40

This lab introduces graphs, a powerful data structure for representing relationships.

* **What you need to know:**
  + **Graph:** A non-linear data structure consisting of a set of "vertices" (or nodes) and a set of "edges" that connect pairs of vertices.
  + **Un-directed Graph:** Edges have no direction (if A is connected to B, then B is connected to A).
  + **Adjacency Matrix:** A common way to represent a graph. It's a square matrix where rows and columns represent vertices. An entry A[i][j] is 1 (or some other value) if there's an edge between vertex i and vertex j, and 0 otherwise.
  + **Degree of a Vertex:** The number of edges connected to a vertex. In an undirected graph, it's simply the count of 1s in its corresponding row/column in the adjacency matrix.

10.1 Create un-directed graph using Adjacency Matrix and display degree of each vertex 41

* **What you'll do:** You'll take input to create an undirected graph represented by an adjacency matrix. Then, you'll calculate and display the degree of each vertex.

10.2 DFS traversal sequence of undirected graph 42

* **What you need to know:**
  + **Depth-First Search (DFS):** An algorithm for traversing or searching tree or graph data structures. It starts at a chosen root (selecting some arbitrary node as a root in the case of a graph) and explores as far as possible along each branch before backtracking. It typically uses a stack (or recursion, which uses the call stack).
* **What you'll do:** Given an undirected graph (likely using the adjacency matrix from 10.1), you'll implement DFS and display the traversal sequence.

10.3 BFS traversal sequence of undirected graph 43

* **What you need to know:**
  + **Breadth-First Search (BFS):** An algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth leve44l. It typically uses a queue.
* **What you'll do:** Given an undirected graph, you'll implement BFS and display the traversal sequence.

10.4 Create directed graph using Adjacency Matrix and display degrees 45

* **What you need to know:**
  + **Directed Graph:** Edges have a direction (if A is connected to B, it doesn't necessarily mean B is connected to A).
  + **In-Degree:** The number of edges pointing *to* a vertex.
  + **Out-Degree:** The number of edges pointing *from* a vertex.
  + **Total Degree:** Sum of in-degree and out-degree.
* **What you'll do:** You'll create a directed graph using an adjacency matrix. For each vertex, you'll calculate and display its in-degree, out-degree, and total degree.

Lab-11 Assignments: Sorting 46464646

This lab focuses on classic sorting algorithms.

11.1 Insertion sort (ascending and descending) 47

* **What you need to know:**
  + **Sorting:** Arranging elements in a specific order (ascending or descending).
  + **Insertion Sort:** A simple sorting algorithm that builds the final sorted array (or list) one item at a time. It iterates through the input elements and inserts each element into its correct position in the already sorted part of the array. Imagine sorting a hand of cards.
* **What you'll do:** You'll implement insertion sort to arrange an array of elements in both ascending and descending order.

11.2 Selection sort (ascending and descending) 48

* **What you need to know:**
  + **Selection Sort:** Another simple sorting algorithm that divides the input list into two parts: a sorted sublist and the unsorted sublist. It repeatedly finds the minimum element from the unsorted sublist and puts it at the beginning of the sorted sublist.
* **What you'll do:** You'll implement selection sort to arrange an array of elements in both ascending and descending order.

11.3 Quick sort (ascending) 49

* **What you need to know:**
  + **Quick Sort:** A highly efficient, comparison-based sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.
* **What you'll do:** You'll implement quick sort to arrange an array of elements in ascending order.

Lab-12 Assignments: Sorting & Searching 50

This final lab covers more sorting algorithms and binary search.

12.1 Merge sort (ascending) 51

* **What you need to know:**
  + **Merge Sort:** A divide-and-conquer algorithm. It recursively divides an array into two halves, sorts each half, and then merges the two sorted halves back into a single sorted array.
* **What you'll do:** You'll implement merge sort to arrange an array of elements in ascending order.

12.2 Heap sort (ascending) 52

* **What you need to know:**
  + **Heap Sort:** A comparison-based sorting technique based on the Binary Heap data structure. It's similar to selection sort where we first find the maximum element and place it at the end. It builds a max-heap (where the parent node is always greater than its children) from the input data.
* **What you'll do:** You'll implement heap sort to arrange an array of elements in ascending order.

12.3 Binary search 53

* **What you need to know:**
  + **Binary Search:** An efficient algorithm for finding an item from a *sorted* list of items. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, 54you narrow the interval to the lower half. Otherwise, you narrow it to the upper half.
  + **Crucial Requirement:** Binary search *only works on sorted data*.
* **What you'll do:** You'll implement binary search to find a given element within a sorted array.

Grading Policies 55

Your overall grade will be determined by:

* **Continuous Evaluation (60%):**
  + Lab record evaluation: 10 marks 56
  + Quizzes: 10 marks (2 quizzes, 5 marks each) 57
  + Viva (oral examination): 10 marks (2 vivas, 5 marks each) 58
  + Program Execution: 20 marks (10 evaluations, 2 marks each) 59
  + Class Participation: 10 marks (10 evaluations, 1 mark each) 60
* **End Semester Evaluation (40%):**
  + Program Execution: 15 marks 61
  + Viva: 10 marks 62
  + Quiz: 15 marks 63

This means regular attendance, active participation, keeping a good lab record, and performing well in quizzes and vivas are all important, in addition to successfully executing your programs.

Reference Materials 64

You have excellent resources at your disposal:

**Reference Books:**

* *Data Structures, Schaum's OutLines* by Seymour Lipschutz 65
* *Data Structures using C* by Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein 66
* *Data Structures A Pseudocode Approach with C* by Richard F. Gilberg, Behrouz Forouzan 67
* *Data Structures Using C* by Reema Thereja 68
* *Data Structures and Algorithm Analysis in C* by Mark Allen Weiss 69

**Reference Sites:**

* NPTEL:

<https://onlinecourses.nptel.ac.in/explorer> 70

* Tutorials Point:

<https://www.tutorialspoint.com/data_structures_algorithms/> 71

* Geeks for Geeks:

<http://www.geeksforgeeks.org/> 72

These resources will be incredibly helpful for understanding the theoretical concepts and seeing example implementations.

Good luck with your Data Structures Lab! It's a challenging but very rewarding subject that forms the backbone of efficient programming.