CS A200: DATA STRUCTURES

Item	Value
Curriculum Committee Approval Date	12/06/2023
Top Code	070600 - Computer Science (Transfer)
Units	4 Total Units
Hours	90 Total Hours (Lecture Hours 63; Lab Hours 27)
Total Outside of Class Hours	0
Course Credit Status	Credit: Degree Applicable (D)
Material Fee	Yes
Basic Skills	Not Basic Skills (N)
Repeatable	No
Grading Policy	Standard Letter (S), • Pass/No Pass (B)

Course Description

A study of data abstraction and algorithm analysis. Data structures include lists, stacks, queues, trees, tables, and graphs. Algorithms include searching, sorting, pattern-matching, tree traversal, and balancing. This is a core course for students who want to study advanced programming, computer science, or engineering. PREREQUISITE: CS A150 and CS A250. Transfer Credit: CSU; UC.

Course Level Student Learning Outcome(s)

1. Define the structural implementation of stacks, queues, trees, and graphs.
2. Evaluate sorting and searching algorithms, including Merge sort, Quick sort, and binary search.

Course Objectives

- 1. Apply object-oriented programming techniques to create abstract data types.
- 2. Differentiate between the interface and implementation of a data structure. Judge which implementation is better for a specific problem.
- 3. Understand the foundation structures of lists.
- 4. Apply the concept of data abstraction to write stack and queue implementations.
- 5. Apply different types of nonlinear data structures, such as trees and graphs.
- 6. Explain the different type of trees (binary search trees, balance trees and AVL trees).
- 7. Produce programs for creating, inserting, deleting, balancing, traversing, and searching for elements in a tree both iteratively and recursively.
- 8. Apply hashing techniques on tables.
- 9. Apply the concepts of searching and sorting to various programming problems and compare between sorting algorithms.
- 10. Apply the concepts of recursion and how to think recursively, distinguish between recursive and iterative solutions, and solve problems using recursive backtracking algorithms.
- 11. Produce programs using heaps, hash-tables, and graphs.
- 12. Design and implement graph traversal algorithms
- 13. Perform basic execution analysis of an algorithm and differentiate between basic efficiency measures.
- 14. Use dynamic memory management and allocation to construct dynamic as well as static data structures.
- 15. Implement abstract data structures.

Lecture Content

Principles of programming and software engineering Problem solving and software engineering What is problem solving The life cycle of software What is a good solution Achieving modular design Abstraction and information hiding Object-oriented design Top-down design General design guidelines Key issues in programming Modularity Modifiability Ease of use Fail-safe programming Style Debugging Recursion Recursive solutions A recursive valued function A recursive “Void” function Counting recursively Searching recursively Finding largest item in an array Binary search Finding the kth smallest item in an array Organizing data Towers of Hanoi Recursion and efficiency Data Abstraction Abstract Data Types (ADT) Specifying ADTs ADT List ADT Sorted list Designing ADT Axioms Implementing ADTs An array-based implementation of ADT List Linked Lists Preliminaries Pointers Dynamic allocation of arrays Pointer-based linked lists Programming with linked lists Displaying contents of a linked list Deleting a node from a linked list Inserting a node into a linked list Pointer-based implementation of ADT List Comparing array-based pointer-based implementations Saving restoring lists by using files Passing a list to a function Processing linked lists recursively Variations of linked lists Ci rcular linked lists Dummy-head nodes Doubly-linked lists Stacks ADT Stack Stack applications Checking balanced braces Recognizing strings in a language Implementation of ADT Stack Array-based implementation Pointer-based implementation Implementation using ADT List Comparing the implementations Application: Algebraic Expressions Evaluating postfix expressions Converting infix into postfix Application: A search problem Non-recursive solution Recursive solution Relationship between stack recursion Queues ADT Queue Simple applications of ADT Queue Reading a string of characters Recognizing palindromes Implementation of ADT Queue Array-based implementation Pointer-based implementation Implementation using ADT List Comparing the implementations Summary of Position-oriented ADTs Algorithm efficiency and sorting Measuring efficiency of an algorithm Execution time Growth rate Order of magnitude and Big-O notation Efficiency of searching algorithms Sorting algorithms their efficiency Selection sort Bubble sort Insertion sort Merge-sort Quick-sort Radix sort Comparison of sorting algorithms Trees Terminology ADT Binary Tree (BT) Traversal of a BT Possible representation of BT Pointer-based implementation of ADT BT Efficiency of ADT BT ADT Binary Search Tree (BST) > Algorithms for ADT Tree Operations Pointer-based implementation of ADT BST Efficiency of ADT BST Tree-sort Saving BST in a file General Trees Tables and Priority Queues ADT Table Selecting an implementation Sorted array-based implementation of ADT Table BST implementation of ADT Table ADT Priority Queue Heaps Heap implementation of ADT Priority Queue Heap-sort Advanced implementation of Tables Balanced Search Trees 2-3 Trees 2-3-4 Trees Red-black Trees AVL Trees Hashing Hash functions Resolving collisions Efficiency of hashing Good hashing functions Table traversal Data with multiple organizations Graphs Terminology Graph ADT Implementing graphs Graph traversal Depth-first Breadth-first Applications of graphs Topological sorting Spanning trees Minimum spanning trees Shortest paths External methods Introduction to external

storage Sorting data in an external file External tables Indexing Hashing Trees Traversal Multiple indexing

Lab Content

The following programming labs are designed to help students master the topics learned through hands-on practice using linear and nonlinear data structures, iterative and recursive approaches, and object-oriented programming techniques: Represent graphs using adjacency lists and matrices, integrating Dijkstras shortest path, Kruskals minimum spanning tree, depth-first, and breadth-first traversals. Implement a Binary Search Tree (BST) class and extend it to include derived classes for balanced trees such as AVL trees, red-black trees, and 2-3 trees. Represent heaps using a dynamic array class, emphasizing their role in algorithms like heapsort. Implement circular queues and priority queues to explore practical applications of data structures. Write stack implementations using linked lists and dynamic arrays. Implement hash tables, covering hash functions, collision resolution, and the efficiency of hashing in data retrieval. Apply sorting concepts to various programming problems, comparing algorithms such as selection sort, insertion sort, merge sort, quicksort, bucket sort, and radix sort. Additional lab exercises should provide practice on all algorithms and data structures covered each week. This includes activities like tracing code, illustrating scheduling techniques, evaluating time complexity by analyzing the execution of algorithms, drawing trees and graphs using different traversal and balancing techniques.

Method(s) of Instruction

- Lecture (02)
- DE Live Online Lecture (02S)
- DE Online Lecture (02X)
- Lab (04)
- DE Live Online Lab (04S)
- DE Online Lab (04X)

Instructional Techniques

Lecture; PowerPoint presentations; problem solving exercises; discussion

Reading Assignments

Students will spend a minimum of 4 hours per week reading the textbook and/or other reading material assigned. Students will be expected to follow along with the exercises in the reading material.

Writing Assignments

Students will spend a minimum of 6 hours per week writing code.

Out-of-class Assignments

Students will spend a minimum of 6 hours per week completing weekly programming assignments.

Demonstration of Critical Thinking

Written tests and quizzes; homework and in-class assignments.

Required Writing, Problem Solving, Skills Demonstration

Successful performance of the assignments and project presentations.

Eligible Disciplines

Computer science: Masters degree in computer science or computer engineering OR bachelors degree in either of the above AND masters degree in mathematics, cybernetics, business administration, accounting or engineering OR bachelors degree in engineering AND masters degree in cybernetics, engineering mathematics, or business administration OR bachelors degree in mathematics AND masters degree in cybernetics, engineering mathematics, or business administration OR bachelors degree in any of the above AND a masters degree in information science, computer information systems, or information systems OR the equivalent. Note: Courses in the use of computer programs for application to a particular discipline may be classified, for the minimum qualification purposes, under the discipline of the application. Masters degree required.

Textbooks Resources

1. Required Cormen, T.H.. Introduction to Algorithms, 4th ed. The MIT Press, 2022