C S 3C: ADVANCED DATA STRUCTURES & ALGORITHMS IN PYTHON

Foothill College Course Outline of Record

Heading	Value
Effective Term:	Summer 2023
Units:	4.5
Hours:	4 lecture, 2 laboratory per week (72 total per quarter)
Prerequisite:	C S 3B.
Advisory:	Demonstrated proficiency in English by placement via multiple measures OR through an equivalent placement process OR completion of ESLL 125 & ESLL 249.
Degree & Credit Status:	Degree-Applicable Credit Course
Foothill GE:	Non-GE
Transferable:	CSU/UC
Grade Type:	Letter Grade (Request for Pass/No Pass)
Repeatability:	Not Repeatable

Student Learning Outcomes

- Analyze the time complexity of a variety of algorithms and data structure access techniques and choose the best algorithm and/or data structure for the project at hand.
- implement and analyze advanced sorting algorithms such as Shellsort and Quicksort.
- write code for balanced trees, hash tables, directed graphs and priority queues and incorporate these structures into larger applications.

Description

A systematic treatment of advanced data structures, algorithm analysis, and abstract data types in the Python programming language, intended for computer science majors as well as non-majors and professionals seeking advanced Python experience. Coding topics include large program software engineering design, multi-dimensional arrays, string processing, primitives, compound types, and allocation of instance and static data. Data structure concept topics include dynamic memory, inheritance, polymorphism, hierarchies, recursion, linked-lists, stacks, queues, trees, hash tables, and graphs. Algorithm concept topics include searching, big-O time complexity, analysis of all major sorting techniques, top down splaying, AVL tree balancing, shortest path algorithms, minimum spanning trees, and maximum flow graphs.

Course Objectives

The student will be able to:

1. Demonstrate a working knowledge of data abstraction through various data types, data structures, and built-in Python classes, and choose the appropriate data structure to model a given problem
2. Design, implement, test, and debug intermediate-level Python programs that use each of the following fundamental programming constructs: user interaction and communication, string processing, numeric computation, simple I/O, arrays, and the Python built-in classes
3. Analyze the basic algorithms of a general tree ADT
4. Use object-oriented programming (OOP) to create alternative implementations of binary search trees in Python, and verify or compare the logN behavior of each
5. Compute the big-O, little-o, omega, and theta time complexity of selected algorithms
6. Define asymptotic behavior and perform empirical benchmarks to compare brute-force techniques with divide-and-conquer strategies
7. Analyze the basic algorithms of a general tree ADT
8. Use object-oriented programming (OOP) to implement a binary search tree in Python and verify its logN behavior
9. Describe the advantages of balanced trees and analyze the performance of AVL trees
10. Write code that realizes a Splay Tree using either top-down or bottom-up splaying
11. Define linear probing, quadratic probing, and open addressing as used in the hash tables ADT
12. Design a priority queue using heaps in Python
13. Implement a Quicksort and at least one other NlogN sort and compare the results as the number of data items approaches infinity
14. Define indirect sorting and explain when it is needed
15. Create a graph data and implement shortest path, minimum spanning tree, and maximum flow problems

Course Content

1. The notion of run-time complexity
   a. Exhaustive search vs. Heuristic search
   b. Search space pruning
   c. Analysis of recursive solutions which are not appropriate due to exponential time complexity vs. iterative polynomial complexity
2. Formal treatment of time complexity
   a. O(n) order of magnitude
   b. o(n) order of magnitude
   c. theta(n) order of magnitude
   d. omega(n) order of magnitude
   e. Constant, polynomial, logarithmic, and exponential time complexity
   f. NlogN time complexity
   g. Improper use of recursion leading to exponential time complexity
3. Measuring asymptotic behavior
   a. Empirical methods of measurement (benchmarking)
   b. Brute-force vs. strategies that use branching or divide-and-conquer algorithms
   c. Timing greedy and depth-first algorithms in graphs
4. General trees
   a. Tree nodes, roots, leaves, children, and siblings
   b. Binary node implementation of a general tree
   c. Insertion and deletion in general trees
   d. Traversal with recursion
5. Searching and Binary Search Trees (BSTs)

   a. Ordering condition and structure condition
   b. Object-oriented programming (OOP) implementation
   c. Time complexity consequence of the divide-and-conquer algorithm in of BSTs
6. Searching and BSTs
   a. Ordering condition and structure condition
   b. OOP implementation
   c. Time complexity of BSTs
   d. Lazy deletion of tree nodes
   e. Threaded trees
   f. Bin heaps
7. Balanced BSTs 1: AVL trees
   a. Tree height and rebalancing
   b. Single and double rotations as the fundamental rebalancing tools
   c. Implementing AVL trees by inheriting from a generic BST
8. Balanced BSTs 2: Splay trees
   a. Splaying
   b. Top down vs. bottom-up splaying
   c. Implementing splay trees using Python
9. Hashing
   a. Hashing functions
   b. Separate chaining
   c. Linear and quadratic probing
10. Priority queues
   a. The percolate down operation
   b. Bin heap implementation of priority queues
   c. Heap sort
11. Non-NlogN sorts
   a. Insertion sort
   b. Shellsort
   c. Benchmarking non-NlogN sorts compared with standard library sort
12. Indirect sorting
   a. What it is
   b. When it is needed
13. Graph theory
   a. Structures of a graph
   b. Nodes, edges, and adjacency tables
   c. Shortest path algorithms and Dijkstra
   d. Minimal spanning tree algorithms and Kruskal
   e. Maximum flow graphs and their algorithms

Lab Content

1. Implementation of time-intensive algorithms on various data types
   a. Design and implement an algorithm whose execution time and/or memory requirements grow significantly when data size increases
   b. Use generics (a.k.a. templates), which are a universal tool in advanced data structures, in some aspect of the algorithm
   c. Demonstrate that the algorithm adapts correctly when the generic that you use is applied to at least two distinct underlying data types
   d. Document the results of the algorithm when the program is applied to different sized data and different underlying data types
2. Using linked-list ADTs to optimize for size-varying or space-sensitive data types
   a. Demonstrate the ability to use programming language-supplied linked-list structures in a problem that is not easily solved using fixed-size ADTs, such as arrays
   b. Incorporate generics so as to allow the algorithm to work on various underlying data types
   c. Try different sized data for the linked-list and demonstrate that it handles growth properly
   d. Summarize the results along with sample program runs
3. Analyzing time complexity in the lab
   a. Implement an assigned algorithm after first predicting its time complexity (linear, quadratic, NlogN, etc.)
   b. Run the algorithm on various sized data sets, recording times
   c. Describe the largest size data set that the computer can handle without running out of memory or taking an unreasonable amount of time
   d. Compare the expected growth rate with the observed growth rate
4. Demonstrating competence with binary search trees
   a. Implement a binary search tree (BST) from scratch, or make significant assigned adjustments to an existing BST data structure supplied by your instructor
   b. Use recursion as appropriate for some of the BST methods
   c. Demonstrate that the class works on various underlying base type by use of generic specialization
   d. Supply runs and report on expected vs. observed time complexity
5. Demonstrating competence with balanced trees
   a. Implement an assigned balanced tree algorithm (such as AVL, splay, or red-black) from scratch, or make significant adjustments to an existing balanced tree algorithm supplied by your instructor
   b. Use recursion as appropriate for some of the balanced tree methods
   c. Demonstrate that the class works on various underlying base type by use of generic specialization
   d. Supply runs and report difference between balanced tree times and simple BST times
6. Incorporating hash tables into programs
   a. Produce a lab that creates or modifies a hash table and hashing function
   b. Write a client that tests out the hash table on various data
   c. Using a large data set, demonstrate that near-constant time access is produced by the hashing function and hash table
   d. Supply runs and report results with varying sized data sets
7. Analysis of a single sort algorithm
   a. Implement a single sort algorithm as directed by the instructor
   b. Experiment with coding adjustments to try to improve the performance
   c. Compare the known time complexity of that algorithm with what you observe using increasingly larger data sets
   d. Attempt to explain any discrepancies in the expected vs. observed growth rate of the sort algorithm
8. Analysis of multiple sort algorithms
   a. Implement multiple sort algorithms, at least two of which involve Shell sort and quicksort
   b. Experiment with coding adjustments to try to improve the performance on any one of them to see if you can beat the fastest of the algorithms

   c. Time the algorithms on very small to very large data sets
   d. Report on which algorithms work better on small sets, and which on large sets
9. Writing projects that use graph theory
   a. Implement a generic (a.k.a. template) that will realize/specialize a graph of any underlying data type, either from scratch or using code provided by your instructor
   b. Write one of the common algorithms for graphs: shortest path, maximum flow, or minimum spanning tree
   c. Discuss the problems that arise when debugging labs which involve data structures as complex as graph theoretic algorithms
   d. Devise a reasonable output for displaying graphs and supply samples with your program runs

Special Facilities and/or Equipment

1. The college will provide access to a computer laboratory with Python interpreters.
2. The college will provide a website or course management system with an assignment posting component (through which all lab assignments are to be submitted) and a forum component (where students can discuss course material and receive help from the instructor). This applies to all sections, including on-campus (i.e., face-to-face) offerings.
3. When taught via Foothill Global Access on the internet, the college will provide a fully functional and maintained course management system through which the instructor and students can interact.
4. When taught via Foothill Global Access on the internet, students must have currently existing email accounts and ongoing access to computers with internet capabilities.

Method(s) of Evaluation

Methods of Evaluation may include but are not limited to the following:

Tests and quizzes
Written laboratory assignments which include source code, sample runs, and documentation
Final examination

Method(s) of Instruction

Methods of Instruction may include but are not limited to the following:

Lectures which include motivation for syntax and use of the Python language and OOP concepts, example programs, and analysis of these programs
Online labs (for all sections, including those meeting face-to-face/on campus), consisting of:
1. A programming assignment webpage located on a college-hosted course management system or other department-approved internet environment. Here, the students will review the specification of each programming assignment and submit their completed lab work
2. A discussion webpage located on a college-hosted course management system or other department-approved internet environment. Here, students can request assistance from the instructor and interact publicly with other class members
Detailed review of programming assignments which includes model solutions and specific comments on the student submissions
In-person or online discussion which engages students and instructor in an ongoing dialog pertaining to all aspects of designing, implementing, and analyzing programs

When course is taught fully online:
1. Instructor-authored lecture materials, handouts, syllabus, assignments, tests, and other relevant course material will be delivered through a college-hosted course management system or other department-approved internet environment
2. Additional instructional guidelines for this course are listed in the attached addendum of CS department online practices

Representative Text(s) and Other Materials

Cormen, Thomas H.. Introduction to Algorithms, 4th ed.. 2022.

Types and/or Examples of Required Reading, Writing, and Outside of Class Assignments

1. Reading
   a. Textbook assigned reading averaging 30 pages per week
   b. Reading the supplied handouts and modules averaging 10 pages per week
   c. Reading online resources as directed by instructor though links pertinent to programming
   d. Reading library and reference material directed by instructor through course handouts
2. Writing
   a. Writing technical prose documentation that supports and describes the programs that are submitted for grades
   b. Writing specifications using prose to connect natural English language to the formulaic programming languages

Discipline(s)

Computer Science