COMP 2140 – Data Structures: Analysis and Implementation

Course Description

Calendar entry

Introduction to the representation, implementation, and analysis of common data structures: stacks, queues, hash tables, binary and balanced trees. Algorithms for manipulating data structures will be analyzed using asymptotic notation. Prerequisites: COMP 1020 or COMP 1021.

General Course Description

Data structures form the backbone of every interesting application that's ever been built and that will ever be built. Being able to implement and analyze data structures is the first step to being able to implement and analyze real-world applications.

This course focuses the analysis and implementation of common abstract data types (ADTs) using fundamental data structures. The data structures include binary search trees, binary heaps, balanced search trees, and hash tables. The ADTs implemented include stacks, queues, priority queues, dictionaries, and graphs. For each ADT, multiple implementations are studied, as are algorithms to support each of the ADT's operations, and the corresponding running times and space requirements are analyzed.

Detailed Prerequisites

Before entering this course, a student should be able to:

- Design and implement iterative and recursive algorithms.
- Design and implement algorithms using arrays.
- Implement basic quadratic-time sorting algorithms.
- Write code that makes use of instantiation, classes, and referencing objects in memory.

Course Goals

By the end of this course students will:

- Express the worst-case cost of iterative algorithms as a function of input size.
- Formally show whether a function f is Big Oh, Theta, Omega, little oh, or little omega of a function g.
- Specify an abstract data type and its corresponding interface.

- Implement the common abstract data types: stack, queue, priority queue, dictionary, and graph, along with their associated operations, using various (appropriately selected) data structures.
- Implement common data structures and their associated operations, including a binary search tree, a binary heap, and a hash table.
- Analyze the running times and space requirements of every data structure/algorithm implemented.

Learning Outcomes

Abstract Data Types and Interfaces

Students should be able to:

1. Explain the difference between what types of data are stored and how they are stored (which data structure), and the difference between what operations are supported and how they are implemented (which algorithm).
2. Design an abstract data type for a given problem.
3. Write code for an interface that corresponds to a given abstract data type.

Algorithm Analysis

Students should be able to:

1. Compare the relative worst-case costs (e.g., worst-case running times) of two algorithms using more than a simple comparison of execution times on sample input.
2. Express the worst-case cost of a simple iterative algorithm as a function of its input size (under simplified assumptions for a unit of computation time, without formally defining a model of computation).
3. Explain the difference between best-case, worst-case, and average-case costs for a deterministic algorithm.
4. Explain the relative asymptotic rates of growth of common functions: constant, logarithmic, linear, quadratic, cubic, exponential, etc.
5. Explain why the largest-order term is responsible for the asymptotic growth, not constants nor lower-order terms.
6. Simplify an expression using asymptotic notation to identify the largest-order term.
7. Formally define Big Oh, Theta, Omega, little oh, and little omega notation.
8. For two polynomial or logarithmic functions f and g, formally show whether function f is Big Oh, Theta, Omega, little oh, or little omega of function g.
9. Appropriately use limits to show asymptotic relationships between two functions.

Stacks and Queues

Students should be able to:

1. Implement constant-time push and pop stack operations by writing appropriate code for an array data structure.
2. Implement constant-time push and pop stack operations by writing appropriate code for a linked list data structure.
3. Implement constant-time enqueue and dequeue queue operations by writing appropriate code for an array data structure.
4. Implement constant-time enqueue and dequeue queue operations by writing appropriate code for a linked list data structure.

Binary Search Trees

Students should be able to:

1. Identify the components of a binary tree: root, leaf node, internal node, left child, right child, left subtree, right subtree, parent node, sibling node, descendant nodes, ancestor nodes.
2. Identify properties of binary trees: height, depth of a node.
3. Implement a binary search tree data structure.
4. Write code that determines whether a tree is height balanced.
5. Write code that determines whether a binary tree is a binary search tree.
6. Write recursive code that traverses a binary tree: pre-order, in-order, and post-order traversals.
7. Implement linear-time (i.e., no tree rebalancing) insert, delete, search, and predecessor dictionary operations by writing appropriate code for a binary search tree data structure.

Balanced Search Trees

Students should be able to:

1. Compare and contrast a binary search tree (having linear-time operations) with a balanced search tree.
2. Understand the differences between weight-balanced trees and height-balanced trees. Express the balancing factor of a tree and determine whether a given tree is balanced or unbalanced.
3. Derive an upper bound on the height of a balanced tree as a function of the number of nodes.
4. Explain tree rotations, when a double rotation is necessary, the cost of a rotation, and which rotations are necessary after insertion and deletion into a balanced search tree.
5. Analyze the worst-case cost of insertion and deletion in a balanced search tree.
6. Implement search and insert dictionary operations by writing appropriate code for a balanced search tree.

Heaps

Students should be able to:

1. Implement an array-based binary tree data structure.
2. Write code that determines whether a binary tree is a heap.
3. Implement logarithmic-time insert and extract priority queue operations by writing appropriate code for an array-based data structure.
4. Write code that reorders an array into a heap-ordered binary tree in-place.

Hash Tables

Students should be able to:

1. Design simple hash functions using modular arithmetic (the division method) or the multiplication method.
2. Implement search, insert, and delete dictionary operations by writing appropriate code for an array-based hash table with open addressing to resolve collisions.
3. Implement search, insert, and delete dictionary operations by writing appropriate code for an array-based hash table with chaining to resolve collisions.
4. Analyze the space and worst-case time costs for the search, insert, and delete operations on a hash table.

Graphs

Students should be able to:

1. Identify the components of a graph: vertex, edge, degree, directed graph, undirected graph.
2. Write code to implement an adjacency list and an adjacency matrix.
3. Analyze the space requirements of an adjacency list and an adjacency matrix as a function of the number of vertices and the number of edges in the graph.
4. Implement depth-first and breadth-first search operations by writing appropriate code that uses stacks and queues.