COMP 2140 – Data Structures and Algorithms

Course Description

Calendar entry

Introduction to the representation and manipulation of data structures. Topics will include lists, stacks, queues, trees, and graphs. May not be held with COMP 2061. 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 on implementing common abstract data types (ADTs) using fundamental data structures. The data structures include arrays, linked lists, binary search trees, binary heaps, hash tables, adjacency lists, and adjacency matrices. 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.

Additional topics include merge sort and quicksort, their running time analysis, as well as lower bounds on the worst-case time for comparison-based sorting; a selection of concepts in programming relevant to ADTs; and discussions of recursive algorithms, which are used extensively.

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:

- 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 doubly linked list, a binary search tree, a binary heap, and a hash table.
- Implement merge sort and quicksort algorithms.
- Analyze the running times and space requirements of every data structure/algorithm implemented.

Learning Outcomes

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.

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.

Sorting Algorithms

Students should be able to:

1. Write code that implements quicksort and merge sort, recursively, for the array data structure.
2. Explain the strategy of divide-and-conquer algorithms.
3. Analyze the worst-case runtimes of merge sort and quicksort.
4. Explain why the worst-case time of quicksort differs from its expected-case time.
5. Write code that implements heapsort using a heap data structure.
6. Implement a counting sort and analyze its running time.

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 graph search operations by writing appropriate code that uses stacks and queues.

Introduction to 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, either a B-tree or 2-3 tree.
2. Express lower and upper bounds on the height of a balanced search tree, either a B-tree or 2-3 tree, as a function of the number of keys stored in the tree.
3. Describe the split and merge operations on a balanced search tree, either a B-tree or 2-3 tree.
4. Implement search and insert dictionary operations by writing appropriate code for a balanced search tree, either a B-tree or 2-3 tree.