Austin M. Mejias

CS 300 // SNHU

**Pseudocode and Runtime Analysis Advising**

**Thesis:**  
To develop an efficient program for ABCU’s Computer Science department that prints a list of all courses in alphanumeric order and displays specific course details, this project evaluates three data structures: vector, hash table, and binary search tree (BST). Through pseudocode design and Big O runtime analysis, this document aims to recommend the most suitable data structure for implementation based on the specific requirements of the advising program.

**Runtime Analysis**

| **Data Structure** | **Operation** | **Big O Notation** |
| --- | --- | --- |
| **Vector** | Reading and Insertion | O(n) |
|  | Sorting (std::sort) | O(n log n) |
|  | Search for a course | O(n) |
| **Hash Table** | Insertion | O(1) (average), O(n) (worst-case) |
|  | Search by key | O(1) (average), O(n) (worst-case) |
|  | Sorting keys | O(n log n) |
| **Binary Search Tree (BST)** | Insertion | O(log n) (average), O(n) (worst-case) |
|  | Search by key | O(log n) (average), O(n) (worst-case) |
|  | In-order traversal for sorting | O(n) |

**Advantages vs. Disadvantages**

Vectors are straightforward and efficient for accessing elements. They allow quick access to data using an index, making retrieval almost instant. They can also resize automatically, which is helpful when you don't know how many courses you'll need to store ahead of time. The data is stored in a continuous block of memory, which helps the computer process it faster. However, adding or removing items in a vector can be slow, especially if it requires shifting many elements around. If you need to sort the data, that can also be slow and resource intensive. Lastly, vectors might use more memory than necessary if they resize often.

Hash tables are very fast for looking up, adding, and deleting data, which is great for quickly finding courses by their ID. They work well with large amounts of data because they can retrieve information almost instantly. You can quickly access data using a key, which is often more intuitive than using an index. However, if the hash function isn't designed well, it can lead to collisions where multiple items are assigned the same spot, making operations slower. Hash tables don't keep data in any order, so you'll need to do extra work to sort it. They can also use a lot of memory because they store both keys and values and need extra space to handle collisions.

Binary search trees automatically keep data sorted, which is handy for printing courses in order. They offer efficient search, insertion, and deletion operations when balanced, typically handling these tasks quickly. BSTs are flexible and can handle changes in data size well. However, if the tree becomes unbalanced, it can slow down operations significantly. Implementing a self-balancing BST, like an AVL tree, adds complexity to your code. Additionally, BSTs might use more memory because each node stores not only data but also pointers to its child nodes.

**VECTOR PSEUDOCODE**

// START PROGRAM

// OPEN THE FILE

OPEN FILE "course\_data.txt" FOR READING

// INITIALIZE VECTOR

DECLARE vector AS COURSE\_VECTOR

// READ THE FILE LINE BY LINE

WHILE NOT EOF("course\_data.txt")

READ LINE

PARSE LINE INTO course\_id, course\_name, prerequisites

CREATE COURSE OBJECT WITH course\_id, course\_name, prerequisites

ADD COURSE OBJECT TO vector

END WHILE

// MENU OPTIONS

DISPLAY "1: Load Data"

DISPLAY "2: Print Alphanumeric List of Courses"

DISPLAY "3: Print Course Information"

DISPLAY "9: Exit"

// GET USER INPUT

INPUT user\_choice

// PERFORM ACTION BASED ON CHOICE

IF user\_choice = 1 THEN

// Load Data

CALL LoadData(vector)

ELSE IF user\_choice = 2 THEN

// Sort and Print Courses

CALL SortCourses(vector)

CALL PrintCourses(vector)

ELSE IF user\_choice = 3 THEN

// Print Course Information

INPUT course\_id

CALL PrintCourseInfo(vector, course\_id)

ELSE IF user\_choice = 9 THEN

// Exit Program

EXIT PROGRAM

END IF

**HASH TABLE PSEUDOCODE**

// START PROGRAM

// OPEN THE FILE

OPEN FILE "course\_data.txt" FOR READING

// INITIALIZE HASH TABLE

DECLARE hash\_table AS COURSE\_HASH\_TABLE

// READ THE FILE LINE BY LINE

WHILE NOT EOF("course\_data.txt")

READ LINE

PARSE LINE INTO course\_id, course\_name, prerequisites

CREATE COURSE OBJECT WITH course\_id, course\_name, prerequisites

ADD COURSE OBJECT TO hash\_table USING course\_id AS KEY

END WHILE

// MENU OPTIONS

DISPLAY "1: Load Data"

DISPLAY "2: Print Alphanumeric List of Courses"

DISPLAY "3: Print Course Information"

DISPLAY "9: Exit"

// GET USER INPUT

INPUT user\_choice

// PERFORM ACTION BASED ON CHOICE

IF user\_choice = 1 THEN

// Load Data

CALL LoadData(hash\_table)

ELSE IF user\_choice = 2 THEN

// Sort and Print Courses

CALL SortCourses(hash\_table)

CALL PrintCourses(hash\_table)

ELSE IF user\_choice = 3 THEN

// Print Course Information

INPUT course\_id

CALL PrintCourseInfo(hash\_table, course\_id)

ELSE IF user\_choice = 9 THEN

// Exit Program

EXIT PROGRAM

END IF

**BINARY SEARCH TREE PSEUDOCODE**

// START PROGRAM

// OPEN THE FILE

OPEN FILE "course\_data.txt" FOR READING

// INITIALIZE BINARY SEARCH TREE

DECLARE tree AS COURSE\_TREE

// READ THE FILE LINE BY LINE

WHILE NOT EOF("course\_data.txt")

READ LINE

PARSE LINE INTO course\_id, course\_name, prerequisites

CREATE COURSE OBJECT WITH course\_id, course\_name, prerequisites

INSERT COURSE OBJECT INTO tree

END WHILE

// MENU OPTIONS

DISPLAY "1: Load Data"

DISPLAY "2: Print Alphanumeric List of Courses"

DISPLAY "3: Print Course Information"

DISPLAY "9: Exit"

// GET USER INPUT

INPUT user\_choice

// PERFORM ACTION BASED ON CHOICE

IF user\_choice = 1 THEN

// Load Data

CALL LoadData(tree)

ELSE IF user\_choice = 2 THEN

// Sort and Print Courses

CALL SortCourses(tree)

CALL PrintCourses(tree)

ELSE IF user\_choice = 3 THEN

// Print Course Information

INPUT course\_id

CALL PrintCourseInfo(tree, course\_id)

ELSE IF user\_choice = 9 THEN

// Exit Program

EXIT PROGRAM

END IF