# Jordan Walker

CS-300

James Webb

February 15th, 2025

**Pseudocode Implementation**

**Main Function (Menu Loop):**

Function main()  
 Set filename = "course\_data.txt"  
   
 Print "Select a data structure to store course information:"  
 Print "1. Vector"  
 Print "2. Hash Table"  
 Print "3. Binary Search Tree"  
 Print "9. Exit"  
   
 Read user input and store in dataChoice  
   
 If dataChoice is "1"  
 Load courses into Vector  
 Else If dataChoice is "2"  
 Load courses into Hash Table  
 Else If dataChoice is "3"  
 Load courses into Binary Search Tree  
 Else If dataChoice is "9"  
 Print "Goodbye!"  
 Exit the program  
 Else  
 Print "Invalid choice, please enter 1, 2, 3, or 9."  
   
 While user does not choose "9"  
 Print "Select an action:"  
 Print "1. Validate Course List"  
 Print "2. Search for a Course"  
 Print "3. Print All Courses"  
 Print "9. Exit"  
  
 Read user input and store in menuChoice  
   
 If menuChoice is "1"  
 Validate the selected data structure  
 Else If menuChoice is "2"  
 Prompt user for course ID  
 Search for course in the selected data structure  
 Else If menuChoice is "3"  
 Print all courses in alphabetical order  
 Else If menuChoice is "9"  
 Print "Goodbye!"  
 Exit the program  
 Else  
 Print "Invalid choice, please try again."

**Vector Implementation:**

Function loadCourses(filename: String) -> Vector<Course>  
 Open file named filename  
 If file cannot be opened  
 Print "Error: Unable to open file."  
 Return an empty Vector<Course>  
  
 Create an empty Vector<Course>  
  
 For each line in the file  
 Split line by commas  
 If line has fewer than 2 items  
 Print "Invalid data format."  
 Continue to next line  
   
 Assign courseNumber and courseName  
 Create an empty list for prerequisites  
   
 If additional items exist in the line  
 Add them to the prerequisites list  
   
 Create a Course object  
 Append to Vector<Course>  
  
 Close file  
 Return Vector<Course>

**Hash Table Implementation:**

Function loadCourses(filename: String) -> HashTable<String, Course>  
 Open file named filename  
 If file cannot be opened  
 Print "Error: Unable to open file."  
 Return an empty HashTable  
  
 Create an empty HashTable<String, Course>  
  
 For each line in the file  
 Split line by commas  
 If line has fewer than 2 items  
 Print "Invalid data format."  
 Continue to next line  
   
 Assign courseNumber and courseName  
 Create an empty list for prerequisites  
   
 If additional items exist in the line  
 Add them to the prerequisites list  
   
 Create a Course object  
 Store Course object in HashTable with courseNumber as key  
  
 Close file  
 Return HashTable

**Binary Search Tree Implementation:**

Function loadCourses(filename: String) -> BinarySearchTree  
 Open file named filename  
 If file cannot be opened  
 Print "Error: Unable to open file."  
 Return an empty BinarySearchTree  
  
 Create an empty BinarySearchTree  
  
 For each line in the file  
 Split line by commas  
 If line has fewer than 2 items  
 Print "Invalid data format."  
 Continue to next line  
   
 Assign courseNumber and courseName  
 Create an empty list for prerequisites  
   
 If additional items exist in the line  
 Add them to the prerequisites list  
   
 Create a Course object  
 Insert Course object into BinarySearchTree  
  
 Close file  
 Return BinarySearchTree

**Run Time Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Binary Search Tree** |
| Insertion | O(1) (unsorted), O(n) (sorted) | O(1) (average), O(n) (worst-case due to collisions) | O(log n) (balanced), O(n) (unbalanced) |
| Search | O(n) (linear search) | O(1) (average), O(n) (worst-case) | |  | | --- | |  |  |  | | --- | | O(log n) (balanced), O(n) (unbalanced) | |
| Sort & Print | O(n log n) (QuickSort) | O(n) (if already stored in sorted order) | O(n) (in-order traversal) |

**Advantage Analysis:**

A vector provides a simple and easy-to-implement solution for storing courses. Since courses are stored sequentially, iterating through the data is efficient. However, searching for a specific course requires O(n) time because a linear search is needed. Sorting a vector requires O(n log n) time, making it the least efficient choice for frequent lookups.

The hash table is the fastest for searching because it offers O(1) average lookup time. This makes it the best choice when frequent retrieval of course information is required. However, if a poor hash function is used or the table size is too small, collisions can occur, degrading performance to O(n) in the worst case. The trade-off is that hash tables require more memory due to key-value pairs.

A binary search tree maintains sorted order, allowing for O(log n) search and insert time if the tree remains balanced. This makes it efficient for searching compared to a vector. However, a BST can become unbalanced, leading to O(n) worst-case time complexity. If the course data is inserted in sorted order without balancing, performance significantly declines.

**Recommendation:**

For this project, the most effective data structure depends on the frequency of data retrieval and the need for efficient lookups. Since searching for courses will likely be the most common operation, a hash table is the best choice due to its average O(1) lookup time. This ensures that course information can be accessed quickly without the delays associated with linear or logarithmic searches. However, to maintain optimal performance, the hash table must be designed with a well-distributed hashing function and an appropriately sized table to minimize collisions. While other structures like binary search trees provide ordered data retrieval, they introduce additional complexity and potential performance degradation if unbalanced. Given the need for quick searches and minimal computational overhead, implementing a hash table offers the best balance of speed and efficiency for this application.