**CS 300 Project One: Pseudocode and Runtime Analysis**

**Note:** *Pseudocode simplified to accommodate report length*

**Pseudocode for Vector Data Structure**

| Function LoadCourses\_Vector(filename)  open file  while not end of file  read line  parse courseNum, courseName, prerequisites  create course object  append to vector  close file  Function PrintCourse\_Vector(courseNum)  for all courses  if courseNum matches  print course name  for each prerequisite  print prerequisite  Function PrintAllCourses\_Vector()  sort vector by courseNum  for all courses  print courseNum and courseName |
| --- |

**Pseudocode for Hash Table**

| Function LoadCourses\_Hash(filename)  open file  while not end of file  read line  parse courseNum, courseName, prerequisites  create course object  insert into hashTable  close file  Function PrintCourse\_Hash(courseNum)  get course from hashTable  if course exists  print course  for each prerequisite  print prerequisite  Function PrintAllCourses\_Hash()  collect all values from hashTable  sort by courseNum  for all courses  print courseNum and courseName |
| --- |

**Pseudocode for Binary Search Tree**

| Function LoadCourses\_BST(filename)  open file  while not end of file  read line  parse courseNum, courseName, prerequisites  create course object  insert into BST  close file  Function PrintCourse\_BST(courseNum)  search BST for courseNum  if found  print course and prerequisites  Function PrintAllCourses\_BST()  perform in-order traversal  print each course |
| --- |

**Runtime Analysis Tables**

Vector Runtime

| **Code Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for all courses | 1 | n | n |
| if courseNum matches | 1 | n | n |
| for each prerequisite of the course | 1 | 1 | 1 |
| print the prerequisite course info | 1 | n | n |
| **Total Cost** |  |  | 3n + 1 |
| Runtime |  |  | O(n) |

Hash Table Runtime

| **Code Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| get course from hash table | 1 | 1 | 1 |
| if course exists | 1 | 1 | 1 |
| for each prerequisite | 1 | p | p |
| print prerequisite info | 1 | p | p |
| **Total Cost** |  |  | 2p + 2 |
| **Runtime** |  |  | **O(1)** for lookup, **O(p)** for printing |

BST Runtime

| **Code Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| search BST for courseNum | log n | 1 | log n |
| if course found | 1 | 1 | 1 |
| for each prerequisite | 1 | p | p |
| print prerequisite info | 1 | p | p |
| **Total Cost** |  |  | log n + 2p + 1 |
| **Runtime** |  |  | **O(log n + p)** |

**Analysis and Recommendation**

In this project, I experimented with three data structures, vector, hash table, and binary search tree (BST), to hold and retrieve course information in an advising system. All the structures were tried on whether they could load information, print all course names alphabetically, and retrieve detailed information for a particular course along with its prerequisites. The vector-based solution is natural and simple but inefficient. Inserting data is efficient (O(1)), but retrieving a specific course requires an O(n) linear time search since the entire list has to be scanned. Furthermore, printing a sorted list of courses requires an explicit sort each time (O(n log n)) and hence creates additional overhead.

Hash table performs well in direct access by course number with constant-time search efficiency (O(1)). This makes it ideal for fast lookups of course data. Hash tables, though, have no inherent ordering. Therefore, for printing the courses in sorted form, the information first must be looked up and sorted, paying an additional O(n log n) cost. The combination makes the hash table very efficient as a search mechanism but somewhat poor when sorted output is often necessary. The binary search tree achieves a balanced trade-off. With the tree balanced, it gives O(log n) search and insert performance. Additionally, its in-order traversal gives a sorted list of courses automatically in O(n) time without the need to sort again. Such dual capability makes the BST ideally suited to the advisor's requirements: quick lookup and sorted output. While insertion is slightly slower than in a hash table, the advantage of structured order and equal performance across operations is enormous.

Overall, taking into account the runtime analysis and implementation, the binary search tree (BST) is optimal data structure for this advising system for this class. It delivers solid performance both for sorted listing and for the retrieval of specific courses, two of the essential requirements of the system. The BST strikes an ideal balance between functionality and efficiency and is the overall best choice.