**CS300 Analysis and Design: Project One**

Daniel J. Vidmar

Southern New Hampshire University

CS300: Analysis and Design

Jimmy Farley

# Print and Menu Pseudocode

# // Assumed Global variables depending on the type

# Declare Vector<Course> courseVector

# Declare HashTable<string, Course> courseHashTable

# Declare BinaryTree<Course> courseTree

# // Function to display the menu and handle user input

# Function displayMenu()

# While True

# Print "Menu:"

# Print "1. Load course data"

# Print "2. Print alphanumeric list of all courses"

# Print "3. Print course information and prerequisites"

# Print "9. Exit program"

# Print "Enter your choice: "

# Declare int choice = Read user input

# If choice == 1

# loadCourseData()

# Else If choice == 2

# printAlphanumericCourseList()

# Else If choice == 3

# printCourseInformation()

# Else If choice == 9

# Exit

# Else

# Print "Invalid choice. Please try again."

# // Function to load course data from a file into all data structures

# Function loadCourseData()

# Declare string filename = "courses.txt"

# Declare courses = loadCoursesFromFile(filename)

# // Load into vector

# courseVector = courses

# // Load into hash table

# For each course in courses

# Insert (course.courseNumber, course) into courseHashTable

# // Load into binary tree

# For each course in courses

# Insert course into courseTree

# Print "Course data loaded successfully."

# // Function to print an alphanumerically ordered list of all courses

# Function printAlphanumericCourseList()

# // if vector Sort and print using vector

# Sort courseVector by courseNumber

# For each course in courseVector

# Print course.courseNumber + ": " + course.name

# // if hashtable Print using hash table (sorted by keys)

# Declare Vector<string> keys = getKeys(courseHashTable)

# Sort keys alphanumerically

# For each key in keys

# Course course = courseHashTable[key]

# Print course.courseNumber + ": " + course.name

# //if binary tree Print using binary tree (in-order traversal)

# inOrderTraversal(courseTree)

# // Function to perform in-order traversal of binary tree and print courses

# Function inOrderTraversal(Course tree)

# If tree is not empty

# inOrderTraversal(tree.left)

# Course course = tree.value

# Print course.courseNumber + ": " + course.name

# inOrderTraversal(tree.right)

# // Main function to start the program

# Function main()

# displayMenu()

# Vector

**Load Courses**

1. Open the file using fstream.
2. If the file cannot be opened, display an error message indicating the file is not found.
3. Otherwise, proceed to read the file.
4. While the end of the file is not reached:
   * Read each line from the file.
   * If the line contains fewer than two values, display an error.
   * Otherwise, extract the parameters from the line.
   * If a parameter appears as the first value elsewhere, continue to the next line.
   * Otherwise, display an error.
5. Close the file.

**Create a Course**

1. Initialize a vector to store Course objects.
2. Loop through the file's content again.
3. While the end of the file is not reached:
   * For each line, extract the first and second values.
   * Use push\_back to add these values to the vector.
   * If there is a third value, continue using push\_back until a newline is encountered.

**Search and Print**

1. Prompt the user for a course number.
2. Loop through the vector:
   * If the input matches a course number in the vector, display the course information.
   * For each prerequisite of the course, display the prerequisite's information.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost Per Op | # of Executions | Total Cost |
| Open File | 1 | 1 | 1 |
| Read Lines | 1 | N | n |
| Validate Line | 1 | N | N |
| Initialize Vector | 1 | 1 | 1 |
| Loop lines | 1 | N | n |
| Add to vector | 1 | N | N |
| Pushback | 1 | N | n |
|  |  | Total Cost | 7n+2 |
|  |  | Runtime | O(n) |

# Hash Table

**Load Courses**

1. Open the file using fstream.
2. If the file cannot be opened, display an error message.
3. Otherwise, while the file is not at the end:
   * Read each line.
   * If a line contains fewer than two values, display an error.
   * Otherwise, extract parameters from the line.
   * If there are three or more parameters, continue to the next line.
4. Close the file.

**Create a Course**

1. Create a Hash Table.
2. Insert data into the hash table.
3. Loop through the file:
   * While the end of the file is not reached, read each line.
   * For each line, extract the first and second values.
   * Create a temporary item to store these values.
   * If there is a third value, add it to the current item.

**Search for Course and Print:**

1. Prompt the user for input.
2. Assign the input to a key.
3. If the key is found in the hash table:
   * Display the course information.
   * For each prerequisite of the course, display the prerequisite's information.
4. If the key is not found, display an error message.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost Per Op | # of Executions | Total Cost |
| Open File | 1 | 1 | 1 |
| Read Lines | 1 | N | n |
| Validate Line | 1 | N | N |
| Extract Parameters | 1 | N | n |
| Create HashTable | 1 | 1 | 1 |
| Insert Course | 1 | N | N |
| Loop through file | 1 | N | n |
| For each line | 1 | N | n |
| Create temp item | 1 | N | n |
| For 1st and 2nd, add id and course | 1 | N | n |
| Has 3rd? add preqs | 1 | N | N |
|  |  | Total Cost | 11n+2 |
|  |  | Runtime | O(n) |

# Binary Tree

**Load Courses**

1. Open the file using fstream.
2. If the file cannot be opened, display an error message indicating the file is not found.
3. Otherwise, while the file is not at the end:
   * Read each line.
   * If a line contains fewer than two values, display an error.
   * Otherwise, extract parameters from the line.
   * If there are three or more parameters, continue to the next line.
4. Close the file.

**Create a Course**

1. Create a Binary Search Tree.
2. Insert data into the tree.
3. While the end of the file is not reached:
   * For each line, extract the first and second values.
   * Add the course ID and course name to the tree.
   * If a third value exists, add it as a prerequisite until a newline is found.

**Search for Course and Print**

1. Prompt the user for input.
2. Create a search and print method:
   * If the tree is not empty, start at the root.
   * Traverse left if necessary.
   * If a node matches the course ID, display the course information.
   * For each prerequisite of the course, display the prerequisite's information.
   * Otherwise, traverse right and repeat the search process.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost Per Op | # of Executions | Total Cost |
| Open File | 1 | 1 | 1 |
| Read Lines | 1 | N | n |
| Validate Line | 1 | N | N |
| Extract Parameters | 1 | N | n |
| Create tree | 1 | 1 | 1 |
| Insert Course | 1 | N | N |
| Loop through file | 1 | N | n |
| For each line | 1 | N | n |
| For 1st and 2nd, add id and course | 1 | N | n |
| Has 3rd? add preqs | 1 | N | N |
|  |  | Total Cost | 10n+2 |
|  |  | Runtime | O(n) |

# Comparisons

## Vector

Vectors excel in quickly reading files and adding course objects due to their efficient memory management and contiguous storage. However, they are less efficient for searching specific courses because each element must be examined sequentially, leading to slower search times.

## Hash Table

Hash tables provide rapid search capabilities by using keys to directly access data, making them highly efficient for lookups. The downside is that hash tables do not maintain order, and additional processing is required to sort courses in alphanumeric order.

## Binary Search Tree

Binary Search Trees (BSTs) efficiently organize or sort items, facilitating easier searches compared to vectors. Although slower than hash tables for searches, BSTs offer a good balance between sorting and searching efficiency. A drawback of BSTs is the longer time required for modifications, such as insertions or deletions.

# Recommendation

Considering the strengths and weaknesses of each data structure, I recommend using vectors for this project. Vectors offer a lower runtime complexity of 7n+2, making them highly effective for reading files and adding objects quickly. Their speed in handling these operations helps offset the slower search and print times compared to other data structures.