Project One

Steve Arevalo

CS 300

Southern New Hampshire University

Professor Nathan Lebel

8/10/2025

**Vector**

**Open File, Reading, Parsing, and Validating File Format**

Open the file for reading

Create an empty list to hold all lines of the file (to check prerequisites later)

While there are still lines in the file:

Read one line from the file

Trim whitespace from the line

If the line is empty:

Skip to the next line

Split the line into parts using a separator (like comma or tab)

If the number of parts is less than 2:

Print error "Invalid line format: Less than two parameters"

Stop processing

Extract course number from the first part

Extract course title from the second part

Extract prerequisites from the remaining parts (if any)

Add the course number to the list of all courses

Store the full line (or parts) temporarily for next processing

Close the file

For each stored line in the temporary list:

For each prerequisite in the line:

If prerequisite does NOT exist in the list of all courses:

Print error "Prerequisite course does not exist: [prerequisite]"

Stop processing

If no errors found:

Proceed to create course objects from the stored lines

**Creating Course Objects and Storing in a Vector**

Define a Course struct with:

- courseNumber (string)

- courseTitle (string)

- prerequisites (list of strings)

Create an empty vector called coursesVector

For each stored line:

Split the line into parts again (course number, title, prerequisites)

Create a new Course object

Set courseNumber to the first part

Set courseTitle to the second part

Set prerequisites to the remaining parts (if any, else empty list)

Add the Course object to coursesVector

**Searching a Course and Printing Its Information and Prerequisites**

Function searchCourse(coursesVector, targetCourseNumber):

For each course in coursesVector:

If course.courseNumber equals targetCourseNumber:

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.courseTitle

If course.prerequisites is empty:

Print "No prerequisites"

Else:

Print "Prerequisites:"

For each prereq in course.prerequisites:

For each course in coursesVector:

If course.courseNumber equals prereq:

Print "- " + course.courseNumber + ": " + course.courseTitle

Return // stop after finding the course

Print "Course not found: " + targetCourseNumber

**Hash Table**

**Open File, Read, Parse, and Validate**

Open the file for reading

Create an empty list called allCourseNumbers to track all courses read

Create an empty temporary list called rawCourseData to store each parsed line

WHILE not end of file:

Read one line from the file

Trim whitespace

IF line is empty:

Continue to next iteration

Split the line by separator (e.g., comma) into parts

IF number of parts < 2:

Print "Error: Line does not have at least two parameters"

Stop processing

Extract courseNumber from parts[0]

Extract courseTitle from parts[1]

Extract prerequisites from parts[2 ... end] (can be zero or more)

Add courseNumber to allCourseNumbers list

Add the parsed parts as a record to rawCourseData

Close the file

FOR each record in rawCourseData:

FOR each prerequisite in that record:

IF prerequisite not in allCourseNumbers:

Print "Error: Prerequisite '" + prerequisite + "' does not exist in file"

Stop processing

If no errors found, proceed to create course objects

**Create course objects and store them in the appropriate data structure**

Define a struct Course with:

- courseNumber (string)

- courseTitle (string)

- prerequisites (list of strings)

Create an empty HashTable called courseHashTable

FOR each record in rawCourseData:

Create a new Course object called newCourse

Set newCourse.courseNumber = record[0]

Set newCourse.courseTitle = record[1]

Set newCourse.prerequisites = list of strings from record[2 ... end] (empty list if none)

Insert newCourse into courseHashTable using courseNumber as the key

**Searching and Printing Course Information from Hash Table**

Function printCourseInfo(courseHashTable, targetCourseNumber):

course = courseHashTable.lookup(targetCourseNumber)

IF course is NULL:

Print "Course not found: " + targetCourseNumber

Return

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.courseTitle

IF course.prerequisites is empty:

Print "No prerequisites"

ELSE:

Print "Prerequisites:"

FOR each prereqCourseNumber in course.prerequisites:

prereqCourse = courseHashTable.lookup(prereqCourseNumber)

IF prereqCourse is NOT NULL:

Print "- " + prereqCourse.courseNumber + ": " + prereqCourse.courseTitle

ELSE:

Print "- " + prereqCourseNumber + ": Course information not found"

**Binary Seach Tree**

**Open File, Read, Parse, and Validate**

Open the file for reading

Create an empty list allCourseNumbers to track all course numbers

Create an empty temporary list rawCourseData to hold parsed lines

WHILE not end of file:

Read one line from the file

Trim whitespace

IF line is empty:

Continue to next iteration

Split the line by separator (e.g., comma) into parts

IF number of parts < 2:

Print "Error: Line does not have at least two parameters"

Stop processing

Extract courseNumber = parts[0]

Extract courseTitle = parts[1]

Extract prerequisites = parts[2 ... end] (zero or more)

Add courseNumber to allCourseNumbers

Add parsed parts to rawCourseData

Close the file

FOR each record in rawCourseData:

FOR each prereq in record's prerequisites:

IF prereq NOT in allCourseNumbers:

Print "Error: Prerequisite '" + prereq + "' does not exist in file"

Stop processing

If no errors, proceed to create course objects

**Create Course Objects and Insert into Binary Search Tree**

Define struct Course:

courseNumber (string)

courseTitle (string)

prerequisites (list of strings)

Define struct TreeNode:

course (Course)

left (TreeNode pointer)

right (TreeNode pointer)

Define function insertBST(root, course):

IF root is NULL:

Create new TreeNode with course

RETURN new TreeNode

IF course.courseNumber < root.course.courseNumber:

root.left = insertBST(root.left, course)

ELSE IF course.courseNumber > root.course.courseNumber:

root.right = insertBST(root.right, course)

ELSE:

Print "Warning: Duplicate course number found: " + course.courseNumber

// Decide how to handle duplicates (skip or update)

RETURN root

Initialize root of BST as NULL

FOR each record in rawCourseData:

Create Course object newCourse

Set newCourse.courseNumber = record[0]

Set newCourse.courseTitle = record[1]

Set newCourse.prerequisites = list from record[2 ... end] or empty list

root = insertBST(root, newCourse)

**Search BST and Print Course Information**

Define function searchBST(root, targetCourseNumber):

IF root is NULL:

RETURN NULL

IF targetCourseNumber == root.course.courseNumber:

RETURN root.course

ELSE IF targetCourseNumber < root.course.courseNumber:

RETURN searchBST(root.left, targetCourseNumber)

ELSE:

RETURN searchBST(root.right, targetCourseNumber)

Define function printCourseInfo(root, targetCourseNumber):

course = searchBST(root, targetCourseNumber)

IF course is NULL:

Print "Course not found: " + targetCourseNumber

RETURN

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.courseTitle

IF course.prerequisites is empty:

Print "No prerequisites"

ELSE:

Print "Prerequisites:"

FOR each prereqCourseNumber in course.prerequisites:

prereqCourse = searchBST(root, prereqCourseNumber)

IF prereqCourse is NOT NULL:

Print "- " + prereqCourse.courseNumber + ": " + prereqCourse.courseTitle

ELSE:

Print "- " + prereqCourseNumber + ": Course information not found"

**Pseudocode For Menu**

Start the program

Set "isDataLoaded" to FALSE

Set "root" (the course data structure) to empty

Repeat the following steps until the user chooses to exit:

Show this menu:

1. Load course data from file

2. Show a list of all courses in order

3. Show details and prerequisites for a course

9. Exit the program

Ask the user to enter a choice

Read what the user enters

If the user chooses 1:

Load all the course data from the file

Save the data into the tree structure

Mark that data has been successfully loaded (set isDataLoaded to TRUE)

Tell the user: "Course data successfully loaded"

If the user chooses 2:

If the data has NOT been loaded yet:

Tell the user: "Please load the data file first using option 1"

Otherwise:

Go through the course list in order

For each course, show its course number and title

If the user chooses 3:

If the data has NOT been loaded yet:

Tell the user: "Please load the data file first using option 1"

Otherwise:

Ask the user to enter a course number (like CS300)

Search the course list for that course

If found, show the course number, title, and any prerequisites

If not found, tell the user the course wasn't found

If the user chooses 9:

Tell the user: "Goodbye"

End the program

If the user enters anything else:

Tell the user: "Invalid choice. Please enter 1, 2, 3, or 9."

End repeat

**Pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order**

**Vector**

Function printCourseList(Vector<Course> courses):

Sort the vector "courses" by course.courseNumber in alphanumeric order

For each course in courses:

Print course.courseNumber + ": " + course.courseTitle

**Hash Table**

Function printCourseList(HashTable<Course> courseTable):

Create an empty list called courseList

For each course in courseTable:

Add course to courseList

Sort courseList by course.courseNumber in alphanumeric order

For each course in courseList:

Print course.courseNumber + ": " + course.courseTitle

**Binary Seach Tree**

Function printCourseList(TreeNode root):

If root is NULL:

Return

// In-order traversal: left, current, right

printCourseList(root.left)

Print root.course.courseNumber + ": " + root.course.courseTitle

printCourseList(root.right)

**Runtime and Memory Analysis by Data Structure**

**Vector**

After parsing, inserting each course object into a vector is fast and simple. Appending an item to the end of a vector is done in constant time. The overall time to insert n courses is O(n). However, searching for a course or sorting the list later, will take additional time O(n) for a search and O(n log n) for sorting.

| **Code Step** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Create course object | 1 | n | n |
| Add course object to end of vector | 1 | n | n |

Total insertion cost: 2n → O(n)

**Advantages:**  
Vectors are easy to use and offer quick appends. They're a good choice for simple storage when order or frequent lookups aren't necessary.

**Disadvantages:**  
Searching is slow, and sorting must be done manually. Not efficient for lookup-heavy tasks.

**Hash Table**

Hash tables allow for average-case constant time insertions and lookups. After reading the file, each course is added using its course number as the key. Copying values for sorting or printing must be done separately. The worst-case scenario, due to hash collisions, is O(n²), but with a good hash function and load factor, the average case is O(n).

| **Code Step** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Create course object | 1 | n | n |
| Insert course into hash table | O(1) | n | n (avg) |

Total insertion cost: 2n → O(n) average, O(n²) worst

**Advantages:**  
Extremely fast lookups and inserts in average cases. Ideal for checking prerequisites or retrieving specific courses quickly.

**Disadvantages:**  
Unordered structure means additional work for sorting. Worst-case performance can degrade without good hash functions.

**Binary Search Tree (BST)**

BSTs store data in a way that naturally maintains order based on course number. Insertions take O(log n) in the average case, but O(n) in the worst case if the tree becomes unbalanced. Once built, an in-order traversal of the tree prints all courses in alphanumeric order without extra sorting steps.

| **Code Step** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Create course object: | 1 | n | n |
| Insert course into BST: | O(log n) average | n | n log n (avg) |

Total insertion cost: **O(n log n)** average, **O(n²)** worst

**Advantages:**  
Perfect for ordered output (A-Z). Good balance between search and sort efficiency.

**Disadvantages:**  
Can become unbalanced and slow if not implemented with self-balancing. Slightly more complex to manage.

**Recommendation**

After evaluating the three structures, I recommend using the Binary Search Tree (BST) for this project. Although hash tables are faster for direct access, they lack natural ordering, which is a key requirement for this advisor system. The vector is simple but performs poorly for search operations and requires sorting as a separate step.

The BST offers the best balance: it supports fast search and insert operations ,especially if balanced, and naturally maintains sorted order. This makes it ideal for printing the course list in alphanumeric order and searching for course details efficiently. With its structured nature and flexibility, the BST is the most appropriate choice for ABCU's advising program.