Jonathan Miller

Prof. Michael J. Susalla

CS-300

April 9, 2023

Project One

BinarySearchTree Internal Data Structures:

```
DEFINE Course struct AS {string courseNumber;
                         string courseName;
                         vector<string> prerequisites; };

DEFINE Node struct AS{Course course;
                      Node *left;
                      Node *right;};
```

HashTable Internal Data Structures:

```
DEFINE Course struct as {string courseNumber;
                         string courseName;
                         vector<string> prerequisites; };

DEFINE Node struct as{Course course;
                      unsigned int key;
                      Node *next; };

DEFINE a nodes vector of Node structs
```

CourseVector Internal Data Structures:

```
DEFINE Course struct as {string courseNumber;
                         string courseName;
                         vector<string> prerequisites; };

DEFINE a courses vector of Course structs
```

```
// This will be defined in the main method, and it's used to store the raw strings from the csv file.
DEFINE csvLines vector of strings

// This function stores the raw strings from the file and returns whether the csv file is formatted correctly.
// This function is the same for all three data structure types!
bool checkFileFormat(string csvPath, vector<string>& csvLines) {
   DEFINE courseElement string
   DEFINE courseNumbers vector of strings
   DEFINE elementCount integer
   DEFINE csvFile ifstream

   OPEN csvFile WITH csvPath

   IF csvFile could not be opened
      OUTPUT an error string

   // Prepare for a new csvFile read.
   CALL clear() on the csvLines vector

   // Populate the csvLines and courseNumbers vectors.
   WHILE we can get a line from the csvFile
      APPEND the line TO csvLines
      SET the currentLineStream TO an istringstream of the line

      WHILE we can get courseElement from currentLineStream with a comma delimiter
         // Store all encountered course numbers.
         APPEND the first courseElement TO courseNumbers

         // Only get the first courseElement to store in courseNumbers.
         BREAK out of the inner loop

   // Check the format of the csv file.
   WHILE we can get a line from the csvFile
      SET the currentLineStream TO an istringstream of the line
      SET elementCount TO 0

      WHILE we can get courseElement from currentLineStream with a comma delimiter
         IF elementCount is GREATER THAN 2
            // If a prerequisite course number does not exist…
            IF courseElement IS NOT IN courseNumbers
               RETURN false

         INCREMENT elementCount

      // After parsing the currentLineStream.
      IF elementCount is LESS THAN 2
         RETURN false

   RETURN true
}
```

BinarySearchTree Course Loading:

```
// This function parses each raw string from the specified file and adds courses to the binary search tree.
void loadCourses(BinarySearchTree* bst, string csvPath, vector<string>& csvLines) {
    // Check the format of the csv file before continuing.
    IF checkFileFormat(csvPath) is EQUAL TO false
        OUTPUT a file format error string
        RETURN

    DEFINE courseElement string
    DEFINE elementIndex integer

    // Parse through each line from the csv file.
    FOR EACH courseString IN csvLines
        CREATE a newCourse struct
        SET courseStringStream TO an istringstream of courseString
        SET elementIndex TO 0

        // Parse the current course string.
        WHILE we can get courseElement from courseStringStream with a comma delimiter
            // Store the course number.
            IF elementIndex is EQUAL TO 0
                SET newCourse.courseNumber TO courseElement

            // Store the course name.
            IF elementIndex is EQUAL TO 1
                SET newCourse.courseName TO courseElement

            // Populate the prerequisite vector.
            IF elementIndex is GREATER THAN 1
                APPEND courseElement TO newCourse.prerequisites

            INCREMENT elementIndex

        // Recursively populate the BST if the root is not null; otherwise, set root to the course node.
        CALL Insert(newCourse) ON bst
}

// This is the public Insert method for the BinarySearchTree abstract data type.
// Worst-case runtime of O(n) for a completely unbalanced tree (basically a linked list at that point)
void BinarySearchTree::Insert(Course course) {
    // If the BST is empty...
    IF the root node IS a null pointer
        // Initialize the root node with the specified bid.
        SET the root node TO a new node with the given course
    ELSE
        CALL addNode on the root node with the given course
}
```

```
// This is the private method to add a node to the BinarySearchTree abstract data type.
void BinarySearchTree::addNode(Node* node, Course course) {
   IF the current node's course.courseNumber IS GREATER THAN the given course.courseNumber
      // If the current node does not have a left child...
      IF the current node’s left child IS a null pointer
         SET the current node’s left child TO a new node with the given course
      // If the current node has a left child...
      ELSE
         // Recursively traverse down the left sub-tree.
         CALL addNode ON the current node’s left child with the given course
   // If the current node's course.courseNumber IS LESS THAN OR EQUAL TO the given
   // course.courseNumber
   ELSE
      // If the current node does not have a right child...
      IF the current node’s right child IS a null pointer
         SET the current node’s left child TO a new node with the given course

      // If the current node has a right child...
      ELSE
         // Recursively traverse down the right sub-tree.
         CALL addNode ON the current node’s right child with the given course
}
```

BinarySearchTree Print Course Information:

```
// This public method prints the specified course information from the binary search tree.
void BinarySearchTree::PrintCourseInformation (string courseNumber) {
   SET the currentNode TO the root node of the BST

   // Loop down the BST.
   WHILE the currentNode IS NOT null
      // If the currentNode contains the specified course.
      IF the specified courseNumber EQUALS the currentNode’s course.courseNumber
         PRINT the currentNode’s course.courseNumber
         PRINT the currentNode’s course.courseName

         // If the currentNode’s course has any prerequisites.
         IF the currentNode’s course.prerequisites size IS GREATER THAN 0
            FOR EACH courseNumber IN the course’s prerequisites vector
               PRINT the prerequisite’s courseNumber

         BREAK out of the while loop if the course is found

      ELSE IF the specified courseNumber IS LESS THAN the currentNode’s courseNumber
         SET the currentNode TO the currentNode’s left child

      ELSE
         SET the currentNode TO the currentNode’s right child
}
```

BinarySearchTree Print All In Order:

```
// This public method prints all courses in the binary search tree in alphanumeric order.
// Worst-case runtime is O(n) since every node is visited.
void BinarySearchTree::PrintCoursesInOrder() {
   CALL printInOrder(node)
}

// This is the recursive private method to print each node in the BST in order.
void BinarySearchTree::printInOrder(Node* node) {
   IF the current node IS NOT null
      // Recursively print out the left sub-tree.
      CALL printInOrder ON the current node’s left child

      PRINT the current node's course.courseNumber
      PRINT the current node’s course.courseName

      IF the current node’s course.prerequisites vector size IS GREATER THAN 0
         // Loop through all elements in the course.prerequisites vector and print each course number.
         FOR EACH prerequisite IN the current node’s course.prerequisites vector
            PRINT the prerequisite courseNumber

      // Recursively print out the right sub-tree.
      CALL printInOrder ON the current node’s right child
}
```

HashTable Course Loading:

```
// This function parses each raw string from the specified file and inserts courses into the hashTable.
void loadCourses(HashTable* hashTable, string csvPath, vector<string>& csvLines) {
    // Check the format of the csv file before continuing.
    IF checkFileFormat(csvPath) is EQUAL TO false
        OUTPUT a file format error string
        RETURN

    DEFINE courseElement string
    DEFINE elementIndex integer

    // Parse through each line from the csv file.
    FOR EACH courseString IN csvLines
        CREATE a newCourse struct
        SET courseStringStream TO an istringstream of courseString
        SET elementIndex TO 0

        // Parse the current course string.
        WHILE we can get courseElement from courseStringStream with a comma delimiter
            // Store the course number.
            IF elementIndex is EQUAL TO 0
                SET newCourse.courseNumber TO courseElement

            // Store the course name.
            IF elementIndex is EQUAL TO 1
                SET newCourse.courseName TO courseElement

            // Populate the prerequisite vector.
            IF elementIndex is GREATER THAN 1
                APPEND courseElement TO newCourse.prerequisites

            INCREMENT elementIndex

        CALL Insert(newCourse) ON hashTable
}
```

```
// This is the public Insert method for the HashTable abstract data type.
// Worst-case runtime of O(n) if all elements are in a single bucket.
void HashTable::Insert(Course course) {
    SET key TO bernsteinHash(course.courseNumber)
    SET headNode TO the head node at the calculated key index in the hash table nodes vector

    // If the head node is not initialized...
    IF headNode IS a null pointer
        CREATE a new nodeToInsert for the given course and calculated key

        INSERT the nodeToInsert into the hash table’s nodes vector at the key index
    ELSE
        // If the bucket is empty...
        IF the headNode’s key IS EQUAL TO UINT_MAX
            // Update the head node with the given course and calculated key.
            SET the headNode’s course TO the given course
            SET the headNode’s key TO the calculated key

            // Make sure the next node points to null.
            SET the headNode’s next node TO a null pointer

        // Otherwise, if there is a collision...
        ELSE
            // Set the starting node in the list traversal to the head node.
            SET currentNode TO the headNode

            // Traverse the list until the tail node is reached.
            WHILE the currentNode’s next node IS NOT a null pointer
                // Update the current node in the list traversal.
                SET currentNode TO the currentNode’s next node

            // Append a new node with the given course and calculated key to the end of the list.
            SET the currentNode’s next node TO a new node with the given course and calculated key
}
```

HashTable Print Course Information:

```
// This public method prints the specified course information from the hash table.
void HashTable::PrintCourseInformation(string courseNumber) {
    // Calculate the key from the hash function.
    SET key TO bernsteinHash(courseNumber)

    // Get the head node at the calculated key.
    SET headNode TO the node at the key index of the hash table

    IF (the head node IS null OR the bucket IS empty)
        PRINT a courseNumber not found message

    ELSE IF (the headNode IS NOT null AND the bucket IS NOT empty
                AND the headNode CONTAINS the specified course)
        PRINT the headNode's course.courseNumber and course.courseName

        // If the course has any prerequisites…
        IF the headNode's course.prerequisites size IS GREATER THAN 0
            FOR EACH prerequisite IN the course.prerequisites vector
                PRINT the course number

    ELSE IF (the list has more than one node
                AND the headNode DOES NOT CONTAIN the specified course)
        // Start the list traversal at the next node.
        SET the currentNode TO the headNode's next node

        // Traverse the remaining nodes in the list.
        WHILE the currentNode IS NOT null
            IF the currentNode CONTAINS the specified courseNumber
                PRINT the currentNode's course.courseNumber and course.courseName

                // If the currentNode's course has any prerequisites…
                IF the currentNode's course.prerequisites size IS GREATER THAN 0
                    FOR EACH prerequisite IN the course.prerequisites vector
                        PRINT the course number

                BREAK out of the while loop once the courseNumber is found
}
```

HashTable Print All In Order:

```
// Load all courses from the hash table into a vector, sort the vector, and then print the sorted vector.
void HashTable::PrintCoursesInOrder() {
    // Temporary vector to hold courses from the hash table for sorting purposes.
    DEFINE courses vector of Course structs

    // Loop through each head node of the hash table.
    FOR EACH index of the nodes vector
        IF the bucket IS NOT empty
            APPEND the headNode.course TO the courses vector

            SET currentNode TO the next node

            // Visit each remaining node in the current hash table bucket.
            WHILE the currentNode IS NOT a null pointer
                APPEND the currentNode.course TO the courses vector

                SET the currentNode TO the next node

    // Sort the courses from the hash table.
    CALL selectionSort on the courses vector

    // Print the course information for each course.
    FOR EACH course IN the courses vector
        PRINT course.courseNumber
        PRINT course.courseName

        // If the course has any prerequisites…
        IF the course.prerequisites size IS GREATER THAN 0
            FOR EACH prerequisite IN the course.prerequisites vector
                PRINT the course number
}

// Use the selection sort algorithm to sort a vector of Course structs. Worst-case runtime of O(n^2).
void selectionSort(vector<Course>& courses) {
    SET the dividing point between sorted and unsorted partitions, i, TO 0.
    SET the current element in the unsorted partition, j, TO 0.
    SET the index for the current minimum course.courseNumber, indexMin, TO 0.
    GET the size of the courses vector and store it in vectorSize.

    // Loop through the courses vector by the dividing point until there is only one element left.
    FOR i = 0 TO (vectorSize – 2), INCREMENT i.
        SET indexMin TO i.

        // Loop through the unsorted partition.
        FOR j = (i + 1) TO (vectorSize – 1), INCREMENT j.
            IF the current element at index j is < the element at indexMin THEN indexMin = j.

        SWAP the element at index i with the element at indexMin.
}
```

CourseVector Course Loading:

```
// This function parses each raw string from the specified file and appends courses to the courses vector.
void loadCourses(CourseVector* courses, string csvPath, vector<string>& csvLines) {
   // Check the format of the csv file before continuing.
   IF checkFileFormat(csvPath) is EQUAL TO false
      OUTPUT a file format error string
      RETURN

   DEFINE courseElement string
   DEFINE elementIndex integer

   // Parse through each line in the csv file.
   FOR EACH courseString IN csvLines
      CREATE a newCourse struct
      SET courseStringStream TO an istringstream of courseString
      SET elementIndex TO 0

      // Parse the current course string.
      WHILE we can get courseElement from courseStringStream with a comma delimiter
         // Store the course number.
         IF elementIndex is EQUAL TO 0
            SET newCourse.courseNumber TO courseElement

         // Store the course name.
         IF elementIndex is EQUAL TO 1
            SET newCourse.courseName TO courseElement

         // Populate the prerequisite vector.
         IF elementIndex is GREATER THAN OR EQUAL TO 2
            APPEND courseElement TO newCourse.prerequisites

         INCREMENT elementIndex

      // After parsing the course string stream…
      CALL Append(newCourse) ON courses
}

// This is the public Append function for the CourseVector abstract data type.
// push_back has an amortized runtime of O(1) if no underlying reallocation happens.
void CourseVector::Append(Course course) {
   CALL push_back(course) ON the courses vector
}
```

CourseVector Print Course Information:

```
// This public method prints the specified course information from the vector of courses.
void CourseVector::PrintCourseInformation(string courseNumber) {
    // Visit each course element in the courses vector.
    FOR EACH course IN the courses vector
        // If the course is found…
        IF the course.courseNumber is EQUAL TO courseNumber
            PRINT the course.courseNumber and course.courseName

            // If the course has any prerequisites…
            IF the course.prerequisites size IS GREATER THAN 0
                FOR EACH prerequisite IN the course.prerequisites vector
                    PRINT the course number

        BREAK out of the outer loop once the course is found
}
```

CourseVector Print All In Order:

```
// Sort the courses vector, and then print all courses in the sorted vector.
void CourseVector::PrintCoursesInOrder() {
    // Sort the vector or courses.
    CALL selectionSort ON the courses vector

    // Visit each course element in the courses vector.
    FOR EACH course IN the courses vector
        PRINT the course.courseNumber and the course.courseName

        // If the course has any prerequisites…
        IF the course.prerequisites size IS GREATER THAN 0
            FOR EACH prerequisite IN the course.prerequisites vector
                PRINT the course number
}
```

Main Menu Pseudocode:

```
int main(int argc, char* argv[]) {
   SET csvPath TO the default path for the courses csv file
   SET courseNumber TO the default courseNumber of the course to find
   DEFINE a csvLines vector of strings

   // Process command line arguments.
   SWITCH over command line arguments
      CASE for one command line argument:
         SET csvPath TO the first command line argument
         BREAK out of the first switch case

      CASE for two command line arguments:
         SET csvPath TO the first command line argument
         SET courseNumber TO the second command line argument
         BREAK out of the second switch case

   // Define the data structure that will be used to hold all courses.
   DEFINE the dataStructure

   SET option TO 0
   SET coursesLoaded boolean TO false

   WHILE the option IS NOT 4
      PRINT the menu title
      PRINT the load data structure option
      PRINT the print course list option
      PRINT the print course option
      PRINT the exit option

      PRINT a message asking the user for the option number

      GET the option number from the user

      SWITCH over the option number from the user
         CASE for option one:
            SET coursesLoaded TO true

            // Complete the method call to load the courses into the specified data structure.
            CALL loadCourses(dataStructure, csvPath, csvLines)
            BREAK out of the switch case for option one

         CASE for option two:
            IF coursesLoaded IS false
               PRINT a message asking the user to load the courses first
            ELSE
               CALL PrintCoursesInOrder() ON the dataStructure
               BREAK out of the switch case for option two
```

```
        CASE for option three:
            IF coursesLoaded IS false
                PRINT a message asking the user to load the courses first
            ELSE
                CALL **PrintCourseInformation**(courseNumber) ON the dataStructure
                BREAK out of the switch case for option three

         CASE for option four:
             PRINT an end program message
             BREAK out of the switch case for option four

    RETURN 0
}
```

checkFileFormat Runtime Analysis:

Code	Line Cost	Times Executed	Total Cost
DEFINE courseElement string	1	1	1
DEFINE courseNumbers vector of strings	1	1	1
DEFINE elementCount integer	1	1	1
DEFINE csvFile ifstream	1	1	1
OPEN csvFile WITH csvPath	1	1	1
IF csvFile could not be opened	1	1	1
OUTPUT an error string	1	1	1
CALL clear() on the csvLines vector	n	1	n
WHILE we can get a line from the csvFile	1	n	n
APPEND the line TO csvLines	1	n	n
SET the currentLineStream TO an istringstream of the line	1	n	n
WHILE we can get courseElement from currentLineStream with a comma delimiter	1	n^2	n^2
APPEND the first courseElement TO courseNumbers	1	n^2	n^2
BREAK out of the inner loop	1	n	n
WHILE we can get a line from the csvFile	1	n	n
SET the currentLineStream TO an istringstream of the line	1	n	n
SET elementCount TO 0	1	n	n
WHILE we can get courseElement from currentLineStream with a comma delimiter	1	n^2	n^2
IF elementCount is GREATER THAN 2	1	n^2	n^2
IF courseElement IS NOT IN courseNumbers	n	1	n
RETURN false	1	1	1
INCREMENT elementCount	1	n	n
IF elementCount is LESS THAN 2	1	1	1
RETURN false	1	1	1
RETURN true	1	1	1
Total Cost	4n^2 + 10n + 11		
Worst-case Runtime	O(n^2)		

loadCourses Runtime Analysis for BinarySearchTree:

Code	Line Cost	Times Executed	Total Cost
IF checkFileFormat(csvPath) is EQUAL TO false	n^2	1	n^2
OUTPUT a file format error string	1	1	1
RETURN	1	1	1
DEFINE courseElement string	1	1	1
DEFINE elementIndex integer	1	1	1
FOR EACH courseString IN csvLines	1	n	n
CREATE a newCourse struct	1	n	n
SET courseStringStream TO an istringstream of courseString	1	n	n
SET elementIndex TO 0	1	n	n
WHILE we can get courseElement from courseStringStream with a comma delimiter	1	n^2	n^2
IF elementIndex is EQUAL TO 0	1	n^2	n^2
SET newCourse.courseNumber TO courseElement	1	n^2	n^2
IF elementIndex is EQUAL TO 1	1	n^2	n^2
IF elementIndex is GREATER THAN 1	1	n^2	n^2
APPEND courseElement TO newCourse.prerequisites	1	n^2	n^2
INCREMENT elementIndex	1	n^2	n^2
CALL Insert(newCourse) ON bst	n	n	n^2
Total Cost	9n^2 + 4n + 4		
Worst-case Runtime	O(n^2)		

PrintCourseInformation Runtime Analysis for BinarySearchTree:

Code	Line Cost	Times Executed	Total Cost
SET the currentNode TO the root node of the BST	1	1	1
WHILE the currentNode IS NOT null	1	n	n
IF the specified courseNumber EQUALS the currentNode’s course.courseNumber	1	n	n
PRINT the currentNode’s course.courseNumber	1	1	1
PRINT the currentNode’s course.courseName	1	1	1
IF the currentNode’s course.prerequisites size IS GREATER THAN 0	1	1	1
FOR EACH courseNumber IN the course’s prerequisites vector	1	n	n
PRINT the prerequisite’s courseNumber	1	n	n
BREAK out of the while loop if the course is found	1	1	1
ELSE IF the specified courseNumber IS LESS THAN the currentNode’s courseNumber	1	n	n
SET the currentNode TO the currentNode’s left child	1	n	n
ELSE	1	n	n
SET the currentNode TO the currentNode’s right child	1	n	n
Total Cost	8n + 5		
Worst-case Runtime	O(n)		

PrintCoursesInOrder Runtime Analysis for BinarySearchTree:

Code	Line Cost	Times Executed	Total Cost
CALL printInOrder(node)	T(n)	1	T(n)
IF the current node IS NOT null	1	n	n
CALL printInOrder ON the current node’s left child	T(n/2)	1	T(n/2)
PRINT the current node's course.courseNumber	1	n	n
PRINT the current node’s course.courseName	1	n	n
IF the current node’s course.prerequisites vector size IS GREATER THAN 0	1	n	n
FOR EACH prerequisite IN the current node’s course.prerequisites vector	1	n^2	n^2
PRINT the prerequisite courseNumber	1	n^2	n^2
CALL printInOrder ON the current node’s right child	T(n/2)	1	T(n/2)
Total Cost	2n^2 + 4n + 2T(n/2)		
Worst-case Runtime	O(n^2)		

loadCourses Runtime Analysis for HashTable:

Code	Line Cost	Times Executed	Total Cost
IF checkFileFormat(csvPath) is EQUAL TO false	n^2	1	n^2
OUTPUT a file format error string	1	1	1
RETURN	1	1	1
DEFINE courseElement string	1	1	1
DEFINE elementIndex integer	1	1	1
FOR EACH courseString IN csvLines	1	n	n
CREATE a newCourse struct	1	n	n
SET courseStringStream TO an istringstream of courseString	1	n	n
SET elementIndex TO 0	1	n	n
WHILE we can get courseElement from courseStringStream with a comma delimiter	1	n^2	n^2
IF elementIndex is EQUAL TO 0	1	n^2	n^2
SET newCourse.courseNumber TO courseElement	1	n^2	n^2
IF elementIndex is EQUAL TO 1	1	n^2	n^2
IF elementIndex is GREATER THAN 1	1	n^2	n^2
APPEND courseElement TO newCourse.prerequisites	1	n^2	n^2
INCREMENT elementIndex	1	n^2	n^2
CALL Insert(newCourse) ON hashTable	n	n	n^2
Total Cost	9n^2 + 4n + 4		
Worst-case Runtime	O(n^2)		

PrintCourseInformation Runtime Analysis for HashTable:

Code	Line Cost	Times Executed	Total Cost
SET key TO bernsteinHash(courseNumber)	1	1	1
SET headNode TO the node at the key index of the hash table	1	1	1
IF (the head node IS null OR the bucket IS empty)	1	1	1
PRINT a courseNumber not found message	1	1	1
ELSE IF (the headNode IS NOT null AND the bucket IS NOT empty AND the headNode CONTAINS the specified course)	1	1	1
PRINT the headNode's course.courseNumber and course.courseName	1	1	1
IF the headNode's course.prerequisites size IS GREATER THAN 0	1	1	1
FOR EACH prerequisite IN the course.prerequisites vector	1	n	n
PRINT the course number	1	n	n
ELSE IF (the list has more than one node AND the headNode DOES NOT CONTAIN the specified course)	1	1	1
SET the currentNode TO the headNode's next node	1	1	1
WHILE the currentNode IS NOT null	1	n	n
IF the currentNode CONTAINS the specified courseNumber	1	n	n
PRINT the currentNode's course.courseNumber and course.courseName	1	1	1
IF the currentNode's course.prerequisites size IS GREATER THAN 0	1	1	1
FOR EACH prerequisite IN the course.prerequisites vector	1	n	n
PRINT the course number	1	n	n
BREAK out of the while loop once the courseNumber is found	1	1	1
Total Cost	6n + 12		
Worst-case Runtime	O(n)		

PrintCoursesInOrder Runtime Analysis for HashTable:

Code	Line Cost	Times Executed	Total Cost
DEFINE courses vector of Course structs	1	1	1
FOR EACH index of the nodes vector	1	n	n
IF the bucket IS NOT empty	1	n	n
APPEND the headNode.course TO the courses vector	1	n	n
SET currentNode TO the next node	1	n	n
WHILE the currentNode IS NOT a null pointer	1	n	n
APPEND the currentNode.course TO the courses vector	1	n	n
SET the currentNode TO the next node	1	n	n
CALL selectionSort on the courses vector	n^2	1	n^2
FOR EACH course IN the courses vector	1	n	n
PRINT course.courseNumber	1	n	n
PRINT course.courseName	1	n	n
IF the course.prerequisites size IS GREATER THAN 0	1	n	n
FOR EACH prerequisite IN the course.prerequisites vector	1	n^2	n^2
PRINT the course number	1	n^2	n^2
Total Cost	3n^2 + 11n + 1		
Worst-case Runtime	O(n^2)		

loadCourses Runtime Analysis for CourseVector:

Code	Line Cost	Times Executed	Total Cost
IF checkFileFormat(csvPath) is EQUAL TO false	n^2	1	n^2
OUTPUT a file format error string	1	1	1
RETURN	1	1	1
DEFINE courseElement string	1	1	1
DEFINE elementIndex integer	1	1	1
FOR EACH courseString IN csvLines	1	n	n
CREATE a newCourse struct	1	n	n
SET courseStringStream TO an istringstream of courseString	1	n	n
SET elementIndex TO 0	1	n	n
WHILE we can get courseElement from courseStringStream with a comma delimiter	1	n^2	n^2
IF elementIndex is EQUAL TO 0	1	n^2	n^2
SET newCourse.courseNumber TO courseElement	1	n^2	n^2
IF elementIndex is EQUAL TO 1	1	n^2	n^2
IF elementIndex is GREATER THAN 1	1	n^2	n^2
APPEND courseElement TO newCourse.prerequisites	1	n^2	n^2
INCREMENT elementIndex	1	n^2	n^2
CALL Append(newCourse) ON courses	1	n	n
Total Cost	8n^2 + 5n + 4		
Worst-case Runtime	O(n^2)		

PrintCourseInformation Runtime Analysis for CourseVector:

Code	Line Cost	Times Executed	Total Cost
FOR EACH course IN the courses vector	1	n	n
IF the course.courseNumber is EQUAL TO courseNumber	1	n	n
PRINT the course.courseNumber and course.courseName	1	1	1
IF the course.prerequisites size IS GREATER THAN 0	1	1	1
FOR EACH prerequisite IN the course.prerequisites vector	1	n	n
PRINT the course number	1	n	n
BREAK out of the outer loop once the course is found	1	1	1
Total Cost	4n + 3		
Worst-case Runtime	O(n)		

PrintCoursesInOrder Runtime Analysis for CourseVector:

Code	Line Cost	Times Executed	Total Cost
CALL selectionSort ON the courses vector	n^2	1	n^2
FOR EACH course IN the courses vector	1	n	n
PRINT the course.courseNumber and course.courseName	1	n	n
IF the course.prerequisites size IS GREATER THAN 0	1	n	n
FOR EACH prerequisite IN the course.prerequisites vector	1	n^2	n^2
PRINT the course number	1	n^2	n^2
Total Cost	3n^2 + 3n		
Worst-case Runtime	O(n^2)		

Advantages and Disadvantages of BinarySearchTree

The binary search tree data structure has some advantages. One such advantage is that the data structure is already sorted because elements are inserted in a sorted order. Therefore, an external sorting algorithm is not needed to print a sorted list of courses. The tree just needs to be traversed in a sorted order while printing each node. This increases the complexity of node insertions but decreases the complexity of printing all courses. Searching for a course in a binary search tree can either be good or bad. If the courses are loaded in a csv file that is not sorted to begin with, then the binary search tree would be rather balanced, and the height would be roughly log(n). This would make searching for a specific course take O(log(n)) time. However, if the csv file of courses is already sorted to begin with, then the binary search tree would end up completely unbalanced, and the binary search tree would essentially just be a linked list. In that case, then searching for a specific course would take O(n) time. Another disadvantage could be the complexity of the algorithms for inserting, traversing, and searching in the binary search tree. These algorithms include recursion, which can be difficult to grasp and implement.

Advantages and Disadvantages of HashTable

The hash table data structure has some advantages for inserting and searching for a specific element. If the hash function does a good job at generating and dispersing keys across the hash table, then insertion will take less than O(n) time. If the hash function generates the same key for each element, then all of the elements would be placed in the same bucket. This would make the hash map behave like a linked list with extra time needed for hashing, so the time complexity for inserting would be O(n) in this case. This bad hash function could also make searching for a specific course take longer because all elements could be in a single bucket. This would make searching take O(n) time. However, searching would take less than O(n) time if the courses are well dispersed across the hash table. A disadvantage of the hash table is that it is not a sorted data structure, and it cannot be sorted due to the nature of hashing into buckets. This would mean that all of the elements from the hash table would need to be added to a sortable data structure and then sorted in that structure. This adds complexity to printing all of the courses in a sorted order with a hash table.

Advantages and Disadvantages of CourseVector

The vector data structure is great for inserting elements. A simple push_back method would need to be called on the vector. This mostly takes O(1) time; however, this could take up to O(n) if the underlying structure needs to reallocate space for the push_back operation. Searching for a course in a vector will always have a worst-case time of O(n) because the element to find could be the last element in the vector. When it comes to sorting the vector, there are various algorithms that can be used to sort the vector in place without needed additional vectors. There are more complex sorting algorithms that take less time in the worst-case, but they require a higher space complexity. The algorithm that I chose was the selection sort algorithm because it does in-place sorting and has a small space complexity. This sorting algorithm takes O(n^2), which means that printing each course in sorted order would be rather slow at O(n^2) since the vector will need to be sorted first.

Recommended Data Structure

As it turns out, the worst-case runtime complexity of loadCourses, PrintCourseInformation, and PrintCoursesInOrder is the same for each data structure due to the way they are implemented. This makes choosing the best data structure a bit more difficult. However, I do not believe that a hash map should be used due to the simple fact that another temporary data structure is needed to print all of the courses in sorted order. The vector data structure would be a good choice because it has the lowest space complexity, but it also needs an additional sorting algorithm function to print all of the courses from the vector in order. Searching in a vector is also always guaranteed to have a worst-case time complexity of O(n), which is not the case for a binary search tree. Therefore, I believe the binary search tree would be the best data structure for the processing of the csv file of courses. The binary search tree will have a faster search time if the csv file is not pre-sorted, and the binary search tree is sorted upon insertion which negates the need to sort all of the courses before printing them. Inserting into the binary search tree will usually take less than O(n) as well.