**Project One**

Thi Hong Van Nguyen

Department of Computer Science and Engineering, Southern New Hampshire University

CS-300 DSA: Analysis and Design

Michael Susalla

August 6, 2023

**Pseudocode**

**Setup: Define a structure to hold course information**

**struct Course {**

string courseNumber; // unique identifier

string title;

vector<string> prerequisites

**}**

**Function ValidateInputFile** (Common function for all data structures)

Purpose: The function opens the csv file, reads the course data from the file, parses each line, and checks for file format errors.

Pseudocode:

bool validateInputFile(string csvPath) {

INITIALIZE fstream inputFile to open file using csvPath

IF inputFile is not open

PRINT error message “File not found”  
 RETURN false  
 END

INITIALIZE an empty vector courseNumbers

INITIALIZE an empty vector prerequisites

WHILE not endOfFile inputFile

INITIALIZE string line to current line

SPLIT line using comma as the delimiter

IF line has less than 2 parameters

PRINT error message “Invalid format in line”

Return false

END

WHILE not end of line, starting from the third parameter

IF current Parameter not already in prerequisites

Push current Parameter to prerequisites

IF current Parameter not already in courseNumbers  
 Push current Parameter to courseNumbers

FOR each prerequisite p in prerequisites

IF p is not in courseNumbers:

PRINT error message “The file contains invalid prerequisite!”  
 Return false

END

Close the file

Return true

END

}

**Case 1: Vector**

***Function LoadCourses***

void LoadCourses (vector<Course>**&** courses) {

INITIALIZE fstream inputFile to open file using csvPath

IF inputFile is not open

PRINT error message “File not found”  
 RETURN  
 END

INITIALIZE an empty vector prerequisites

WHILE not endOfFile inputFile

INITIALIZE string line to current line

SPLIT line using comma as the delimiter

DECLARE Course course

SET course’s courseNumber to first parameter of the splitted line

SET course’s title to 2nd parameter of the splitted line

LOOP through the rest of the parameters

STORE each parameter in the declared prerequisites

SET course.prerequisites to prerequisites vector

APPEND course to the courses vector

Close the file  
END

}

***Function PrintCourseInformation***

Void printCourseInformation(vector<Course> courses, String courseNumber){

DECLARE foundCourse as false

FOR each course in courses

IF course’s number is equal to courseNumber

PRINT course’s number and title

FOR each prerequisite p in course’s prerequisites

PRINT p

SET foundCourse to true

IF foundCourse is equal to false

PRINT message “Course not found”

END

}

***Function Partition***

int partition(vector<Course>& courses, int begin, int end) {

INITIALIZE int low to begin

INITIALIZE int high to end

INITIALIZE int midpoint to low + (high - low) / 2

INITIALIZE Course pivot to the course at midpoint

INITIALIZE done to false

WHILE not done

WHILE course’s title at low is smaller than pivot’s title

INCREMENT low

WHILE pivot’s title is smaller than course’s title at high

DECREMENT high

IF low is greater than or equal to high

SET done to true

ELSE

SWAP course at low with course at high

INCREMENT low

DECREMENT high

RETURN high

END

}

***Function quickSort***

void quickSort(vector<Course>& courses, int begin, int end) {

INITIALIZE int mid to 0

IF begin is greater than or equal to end

RETURN

SET mid to partition(courses, begin, end)

RECURSE quickSort(courses, begin, mid)

RECURSE quickSort(courses, mid + 1, end)

END

}

***Function PrintCourses***

void PrintCourses (vector<Course>& courses) {

quickSort(courses, 0, size of courses - 1)

FOR course in courses vector

PRINT "Course Number: " + course's courseNumber

PRINT "Title: " + course's title

PRINT "Prerequisites: "

FOR prerequisite in course's prerequisites vector

PRINT "\t" + prerequisite

END

}

**Case 2: Hash Table**

**struct Node {**

Course course

unsigned int key

Node \*next;

// default constructor

Node() {

key = UINT\_MAX // To be decided

next = nullptr

}

// initialize with a course

Node(Course aCourse) : Node() {

course = aCourse

}

// initialize with a course and a key

Node(Course aCourse, unsigned int aKey) : Node(aCourse) {

key = aKey;

}

**}**

***Define a vector of nodes to store the data***

DECLARE vector<Node> nodes

***Function LoadCourses***

void loadCourses(string csvPath ) {

INITIALIZE fstream inputFile to open file using csvPath

IF inputFile is not open

PRINT error message “File not found”  
 END

ELSE

While not endOfFile of inputFile

INITIALIZE string line to current line

SPLIT line using comma as the delimiter

INITIALIZE string courseNumber to line’s first parameter

INITIALIZE string title to line’s second parameter

INITIALIZE vector<string> prerequisites to empty vector of strings

LOOP through the rest of the parameters

APPEND the parameter to prerequisites

CREATE a new Course object with courseNumber, title and prerequisites

Call Insert function with the object as parameter

Close the file  
END

}

***Function Insert***

Insert(Course course) {

INITIALIZE unsigned key to hash value of the course’s courseNumber

RETRIEVE node using key

IF no entry found for the key

ASSIGN node to the key position

ELSE IF node is not used

SET node.key to key

SET node.course to course

SET node.next to null pointer

ELSE

WHILE node.next is not null pointer

SET node to node.next

SET node.next to new Node constructed by course and key

}

***Function PrintCourseInformation***

Void printCourseInformation(String courseNumber){

INITIALIZE unsigned key to the hash value of the given courseNumber

GET the node from key position

IF no entry found for the key

PRINT message “Course not found! ”

IF entry found for the key

PRINT courseNumber, title

FOR each prerequisite p in prerequisites

PRINT p

ELSE

WHILE node not equal to null pointer

IF the current node matches

PRINT courseNumber, title

FOR each prerequisite p in prerequisites

PRINT p

SET node to node.next

END

}

***Function PrintCourses***

Void printCourses (){

DECLARE vector<Course> courses

FOR each iterator in nodes

IF iterator.key is not equal to UINT\_MAX

APPEND iterator to courses

INITIALIZE Node\* node to iterator.next

WHILE node is not equal to null pointer

APPEND node to courses

SET node to node.next

// Call quickSort function above

CALL function quickSort(courses, 0, size of courses - 1)

FOR each course in courses vector

PRINT "Course Number: " + course's courseNumber

PRINT "Title: " + course's title

PRINT "Prerequisites: "

FOR prerequisite in course's prerequisites vector

PRINT "\t" + prerequisite

END

}

**Case 3: Binary Tree**

**struct Node {**

Course course;

Node \*left;

Node \*right;

// default constructor

Node() {

left = nullptr;

right = nullptr;

}

// initialize with a course

Node(Course aCourse) :

Node() {

Course = aCourse;

}

**}**

***Function LoadCourses***

Void LoadCourses(string csvPath, Node\*& root) {

INITIALIZE fstream inputFile to open file using csvPath

IF inputFile is not open

PRINT error message “File not found”  
 RETURN  
 END

INITIALIZE an empty vector prerequisites

WHILE not endOfFile inputFile

INITIALIZE string line to current line

SPLIT line using comma as the delimiter

DECLARE Course course

SET course’s courseNumber to first parameter of the splitted line

SET course’s title to 2nd parameter of the splitted line

STORE the rest of the parameters in prerequisites

SET course’s prerequisites to prerequisites vector

CALL function insertCourse(root, course)

END while loop

CLOSE file

END

}

***Function insertCourse***

Purpose: Function to insert a course into the binary tree

void insertCourse(Node\*& root, Course course) {

IF root is null pointer

SET root to new Node formed by course

ELSE

IF course’s courseNumber is smaller than root.course.courseNumber

RECURSE insertCourse with root.left and course as arguments

ELSE

RECURSE insertCourse with root.right and course as arguments

END

}

***Function PrintCourseInformation***

Purpose: The function searches the data structure for a specific course and prints out course information and prerequisites. If the course is not found, a message is displayed to the user.

Void printCourseInformation(Node\* root, const string& courseNumber){

IF root is null pointer

PRINT “Course not found!”

END

IF root.course.courseNumber is equal to courseNumber

PRINT root.course.courseNumber and root.course.title

FOR prerequisite p in root.course.prerequisite

PRINT p

ELSE IF courseNumber is smaller than root.course.courseNumber

RECURSE printCourseInformation(root.left, courseNumber)

ELSE

RECURSE printCourseInformation(root.right, courseNumber)

END

}  
***Function PrintCourses***

Void PrintCourses(Node\* root) {

IF root is not null pointer

RECURSE PrintCourses(root.left)

PRINT "Course Number: " + root.course.courseNumber

PRINT "Title: " + root.course.title

PRINT "Prerequisites: "

FOR each prerequisite p in root.course.prerequisites

PRINT " - " + p

RECURSE PrintCourses(root.right)

END

}

**Menu**

int main(int argc, char\* argv[]) {

INITIALIZE choice to 0

WHILE choice is not equal to 9

PRINT “MENU:”

PRINT “1. Load Course”

PRINT “2. Print All Courses”

PRINT “3. Print a specific Course”

PRINT “9. Exit”

PRINT “Enter choice:”

INPUT choice

IF choice is not equal to 1, 2, 3, or 9

PRINT “Invalid menu selection”

ELSE

IF choice is equal to 1

DECLARE string csvPath

PRINT “Enter file path”

INPUT csvPath

INITIALIZE boolean isValid

SET isValid to validateInputFile(csvPath)

IF isValid is true

LoadCourses() // Pass in appropriate parameter for selected data structure

IF choice is equal to 2

PrintCourses() // Pass in appropriate parameter for selected data structure

IF choice is equal to 3

PrintCourseInformation() // Pass in appropriate parameter for selected data structure

PRINT “Program terminated…”

END

}

**Evaluation**

**Vector data structure**

***LoadCourses***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE fstream inputFile to open file using csvPath | 1 | 1 | 1 |
| IF inputFile is not open | 1 | 1 | 1 |
| PRINT error message “File not found” | 1 | 1 | 1 |
| RETURN | 1 | 1 | 1 |
| INITIALIZE an empty vector prerequisites | 1 | 1 | 1 |
| WHILE not endOfFile inputFile | 1 | n | n |
| INITIALIZE string line to current line | 1 | n | n |
| SPLIT line using comma as the delimiter | 1 | n | n |
| DECLARE Course course | 1 | n | n |
| SET course's courseNumber to first parameter of the splitted line | 1 | n | n |
| SET course's title to 2nd parameter of the splitted line | 1 | n | n |
| LOOP through the rest of the parameters | 1 | n | n |
| STORE each parameter in the declared prerequisites | 1 | m\*n | m\*n |
| SET course.prerequisites to prerequisites vector | 1 | n | n |
| APPEND course to the courses vector | 1 | n | n |
| Close the file | 1 | 1 | 1 |
| **Total Cost** |  |  | n(m+9) + 6 |
| **Runtime** |  |  | O(n) |

Since both n and m are variables representing the number of courses and the number of prerequisites per course, respectively, we need to determine which one has a larger impact on the overall runtime. I would assume that m should always be much smaller than n. In that scenario, the runtime complexity can be approximated as O(n).

***Function PrintCourseInformation***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| FOR each course in courses | 1 | 1 | 1 |
| IF course’s number is equal to courseNumber | 1 | n | n |
| PRINT course’s number and title | 1 | n | n |
| FOR each prerequisite p in course’s prerequisites | 1 | n | n |
| PRINT p | 1 | m | m |
| SET foundCourse to true | 1 | 1 | 1 |
| IF foundCourse is equal to false | 1 | 1 | 1 |
| PRINT message “Course not found” | 1 | 1 | 1 |
| **Total Cost** |  |  | 3n + m + 4 |
| **Runtime** |  |  | O(n) |

***PrintCourses***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| quickSort(courses, 0, size of courses - 1) | 1 | n^2 | n^2 |
| FOR course in courses vector | 1 | n | n |
| PRINT "Course Number: " + course's courseNumber | 1 | n | n |
| PRINT "Title: " + course's title | 1 | n | n |
| FOR prerequisite in course's prerequisites vector | 1 | n | n |
| PRINT "\t" + prerequisite | 1 | m\*n | m\*n |
| **Total** |  |  | (m + 4)n + n^2 |
| **Runtime** |  |  | O(n^2) |

The time complexity of sorting a vector using quickSort in the worst case scenario is O(n^2) so line 1 cost would be n^2. To calculate the total cost and runtime, we need to consider the highest order of the terms in the costs. The highest order in this case is O(n^2) because it dominates the other terms. Therefore, the runtime of the function is O(n^2) in this case.

**Hash table**

***LoadCourses***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE fstream inputFile to open file using csvPath | 1 | 1 | 1 |
| IF inputFile is not open | 1 | 1 | 1 |
| PRINT error message "File not found" | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| While not endOfFile of inputFile | 1 | n | n |
| INITIALIZE string line to current line | 1 | n | n |
| SPLIT line using comma as the delimiter | 1 | n | n |
| INITIALIZE string courseNumber to line's first parameter | 1 | n | n |
| INITIALIZE string title to line's second parameter | 1 | n | n |
| INITIALIZE vector<string> prerequisites to empty vector of strings | 1 | n | n |
| LOOP through the rest of the parameters | 1 | n | n |
| APPEND the parameter to prerequisites | m | n | m\*n |
| CREATE a new Course object with courseNumber, title, and prerequisites | 1 | n | n |
| Call Insert function with the object as parameter | n | n | n^2 |
| Close the file | 1 | 1 | 1 |
| **Total Cost** |  |  | 5 + (8+m)n + n^2 |
| **Runtime** |  |  | O(n^2) |

The worst case of a hash table’s insertion is O(n) so for each course we have to call the Insert function once. Therefore the cost for “Call Insert function with the object as parameter” would be n\*n. To calculate the runtime, we need to consider the highest-order term in the total cost, which will be O(n^2).

***Function PrintCourseInformation***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE unsigned key to the hash value of the given courseNumber | 1 | 1 | 1 |
| GET the node from key position | 1 | 1 | 1 |
| IF no entry found for the key | 1 | 1 | 1 |
| PRINT message “Course not found! ” | 1 | 1 | 1 |
| IF entry found for the key | 1 | 1 | 1 |
| PRINT courseNumber, title | 1 | 1 | 1 |
| FOR each prerequisite p in prerequisites | 1 | 1 | 1 |
| PRINT p | 1 | m | m |
| ELSE | 1 | 1 | 1 |
| WHILE node not equal to null pointer | 1 | 1 | 1 |
| IF the current node matches | 1 | n | n |
| PRINT courseNumber, title | 1 | n | n |
| FOR each prerequisite p in prerequisites | 1 | n | n |
| PRINT p | m | n | m\*n |
| SET node to node.next | 1 | n | n |
| **Total Cost** |  |  | 9 + m + n(m + 3) |
| **Runtime** |  |  | O(m\*n) |

The highest order term is n(m + 3) so the runtime complexity of the function is O(m \* n).

***PrintCourses***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| DECLARE vector<Course> courses | 1 | 1 | 1 |
| FOR each iterator in nodes | 1 | n | n |
| IF iterator.key is not equal to UINT\_MAX | 1 | n | n |
| APPEND iterator to courses | 1 | n | n |
| INITIALIZE Node\* node to iterator.next | 1 | n | n |
| WHILE node is not equal to null pointer | 1 | n | n |
| APPEND node to courses | 1 | n | n |
| SET node to node.next | 1 | n | n |
| CALL function quickSort(courses, 0, size of courses - 1) | 1 | n^2 | n^2 |
| FOR each course in courses vector | 1 | n | n |
| PRINT "Course Number: " + course's courseNumber | 1 | n | n |
| PRINT "Title: " + course's title | 1 | n | n |
| PRINT "Prerequisites: " | 1 | n | n |
| FOR prerequisite in course's prerequisites vector | 1 | n | n |
| PRINT "\t" + prerequisite | m | n | m\*n |
| **Total Cost** |  |  | (12 + m)n + 1 + n^2 |
| **Runtime** |  |  | O(n^2) |

The highest order term is n^2, so the runtime complexity is O(n^2).

**Binary Tree**

***LoadCourses***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE fstream inputFile to open file using csvPath | 1 | 1 | 1 |
| IF inputFile is not open | 1 | 1 | 1 |
| PRINT error message “File not found” | 1 | 1 | 1 |
| RETURN | 1 | 1 | 1 |
| INITIALIZE an empty vector prerequisites | 1 | 1 | 1 |
| WHILE not endOfFile inputFile | 1 | 1 | 1 |
| INITIALIZE string line to current line | 1 | n | n |
| SPLIT line using comma as the delimiter | 1 | n | n |
| DECLARE Course course | 1 | n | n |
| SET course’s courseNumber to first parameter of the splitted line | 1 | n | n |
| SET course’s title to 2nd parameter of the splitted line | 1 | n | n |
| STORE the rest of the parameters in prerequisites | m | n | m\*n |
| SET course’s prerequisites to prerequisites vector | 1 | n | n |
| CALL function insertCourse(root, course) | n | n | n^2 |
| CLOSE file | 1 | 1 | 1 |
| **Total Cost** |  |  | (6 + m)n + 7 + n^2 |
| **Runtime** |  |  | O(n^2) |

The worst-case time complexity for insertion in a binary search tree (BST) is O(n), where n is the number of nodes in the tree. This occurs when the tree is unbalanced, and it essentially degenerates into a linked list. So the cost of the line “CALL function insertCourse(root, course)” is O(n).

***Function PrintCourseInformation***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| IF root is null pointer | 1 | 1 | 1 |
| PRINT “Course not found!” | 1 | 1 | 1 |
| IF root.course.courseNumber is equal to courseNumber | 1 | 1 | 1 |
| PRINT root.course.courseNumber and root.course.title | 1 | 1 | 1 |
| FOR prerequisite p in root.course.prerequisite | 1 | 1 | 1 |
| PRINT p | m | m | m |
| ELSE IF courseNumber is smaller than root.course.courseNumber | 1 | 1 | 1 |
| RECURSE printCourseInformation(root.left, courseNumber) | 1 | n | n |
| ELSE | 1 | 1 | 1 |
| RECURSE printCourseInformation(root.right, courseNumber) | 1 | n | n |
| **Total Cost** |  |  | 7 + m + 2n |
| **Runtime** |  |  | O(n) |

The worst-case time complexity for searching in a binary search tree is O(n) when the tree is unbalanced, so the runtime would be O(n).

***PrintCourses***

| **Line** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| IF root is not null pointer | 1 | 1 | 1 |
| RECURSE PrintCourses(root.left) | 1 | n | n |
| PRINT "Course Number: " + root.course.courseNumber | 1 | 1 | 1 |
| PRINT "Title: " + root.course.title | 1 | 1 | 1 |
| PRINT "Prerequisites: " | 1 | 1 | 1 |
| FOR each prerequisite p in root.course.prerequisites | 1 | 1 | 1 |
| PRINT " - " + p | m | 1 | m |
| RECURSE PrintCourses(root.right) | 1 | n | n |
| **Total Cost** |  |  | 5 + m + n |
| **Runtime** |  |  | O(n) |

The recursive calls to PrintCourses(root.left) and PrintCourses(root.right) can be O(n) each in the worst case if the binary search tree is skewed. Therefore, in the worst case, the runtime can be O(n + m) if the tree is skewed.

**Advantages and disadvantages**

***Vector***

Vectors are dynamic arrays that provide a simple and straightforward data structure to store and manage course information. In the context of our analysis, we find that the LoadCourses function exhibits a runtime of O(n), which means it is relatively efficient in loading courses from a CSV file into the vector. This linear time complexity allows for quick data insertion, making it suitable for applications with a moderate number of courses.

Furthermore, the PrintCourseInformation function also demonstrates a runtime of O(n). It efficiently searches and prints course information based on the given course number. This linear time complexity ensures swift data retrieval and presentation, which is advantageous for smaller datasets.

However, when it comes to the PrintCourses function, the runtime becomes O(n^2). This inefficiency arises from the sorting operation required to print all courses in alphabetical order. For larger datasets, this can lead to slower performance, making it a drawback for applications with extensive course catalogs.

***Hash Table***

Hash tables provide an excellent solution for efficient data retrieval and storage. The LoadCourses function in this case exhibits a runtime of O(n^2). While not as fast as the vector implementation, it is still reasonable for loading courses into a hash table. The hashing technique allows for relatively fast insertion times.

The PrintCourseInformation function demonstrates a runtime of O(m\*n), where m is the average number of prerequisites for each course. This runtime indicates that the hash table can efficiently search and print course information along with its prerequisites. For applications where prerequisites are a critical factor, this data structure can be advantageous.

Nonetheless, the PrintCourses function faces the same O(n^2) runtime issue, primarily due to the sorting requirement to print courses in alphabetical order. The hash table itself does not guarantee sorting, necessitating additional steps that can impact performance.

***Binary Tree***

Binary trees offer an organized hierarchical structure for storing course information. The LoadCourses function presents a runtime of O(n^2), which is reasonable for loading courses into a binary tree. The binary tree's balanced structure can provide relatively fast insertion times.

The PrintCourseInformation function demonstrates a runtime of O(n), which is efficient for searching and printing course information based on course numbers. The binary tree's structural advantages enable quick searches, making it an excellent choice for applications that require frequent data retrieval.

The PrintCourses function showcases a runtime of O(n), indicating efficient course printing, with sorting.

**Recommendation**

Each data structure offers a unique set of advantages and disadvantages when it comes to managing course information. The vector's simplicity and linear time complexity make it ideal for smaller datasets and straightforward operations. Hash tables are excellent for fast data retrieval and storage, making them ideal for applications prioritizing quick searches. However, hash tables with chaining might be less efficient compared to vectors and binary trees in terms of searching. Lastly, binary trees' hierarchical structure enables quick searches and efficient course printing, but the need for additional sorting may hamper performance.

Based on the analyzed runtime complexities and considering the efficiency of printing all courses while maintaining a straightforward loadCourses implementation, I would recommend using the binary tree data structure for managing course information in the ABCU system. The binary tree provides a more efficient runtime of O(n) for printing all courses compared to the hash table and vector’s O(n^2). Additionally, the binary tree offers the advantage of presenting courses in alphabetical order, which can be beneficial for better organization and readability.