# Gabrielle Maitland

CS 300 Data Structures and Algorithms

# Project One

## Function Signatures

// Vector pseudocode

**Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.**

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

CALL filename.open

IF file is open

CALL getline to read data by each line

SPLIT line at comma

STORE each value into splitLine variable

**Your pseudocode should show how to create course objects, so that one course object holds data from a single line from the input file.**

INITIALIZE new courseObject (vector<course> courses)

IF splitLine length is equal to 2

INITIALIZE course objects for course name

create course.name

INITIALIZE course objects for course number

create course.number

IF split line length is greater than 2

INITIALIZE course objects for prereqs

course.prerequisites

ADD course objects to vector allCourses

IF there are less than two values from split line AND prerequisite is not in file

PRINT file format error

ELSE

IF file will not open

PRINT unable to open file

}

**Design pseudocode that will print out course information and prerequisites. Print Course: This will print the course title and the prerequisites for any individual course.**

void printCourseInformation(Vector<Course> courses, String courseNumber) {

FOR all courses

IF course is equal to course number

PRINT FOUND course.name and course.number

ELSE

PRINT course does not exist

FOR each prerequisite of course

IF course prerequisite length is greater than 0

PRINT course.prerequisites

}

**Sort the course information by alphanumeric course number from lowest to highest.**

void printInOrder(Vector<Course> courses, String courseNumber)

Partition bids into low and high

SET low = begin

SET high = end

SET pivot = low + (high-low) / 2

WHILE low < pivot

INCREMENT low

WHILE high > pivot

DECREMENT high

IF 0 or 1 elements remaining

all bids are partitioned.

ELSE swap low and high

Use quicksort to sort courses

INITIALIZE midpoint to 0

IF begin >= end

RETURN;

Set midpoint = partition into low and high

Recursively sort low partition begin to mid

CALL quicksort(bids, begin, midpoint)

CALL quicksort(bids, midpoint +1, end)

// Hashtable pseudocode

**Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.**

CALL filename.open

IF file is open

CALL getline to read data by each line

SPLIT line at comma

STORE each value into splitLine variable

* **Your pseudocode should show how to create course objects, so that one course object holds data from a single line from the input file.**

int numPrerequisiteCourses(Hashtable<Course> courses) {

INITIALIZE int key

INITIALIZE node \*Next

CONSTRUCT node with key as uint\_max and next as nullptr

SET tableSize as default\_size

Initialize node structure by resizing tableSize

Initialize logic to free storage when class is destroyed

IF splitLine length is greater or equal to 2

CREATE a key for given course with a hash function

IF no entry found for key, assign a new node

IF oldNode is null

new Node(course, key)

INSERT new node

ELSE if node is unoccupied

UPDATE key with new key

UPDATE course data and change next pointer to null

ELSE if there is a collision, use chaining to find open node

IF there are less than two values from split line AND prerequisite is not in file

PRINT file format error

ELSE

IF file will not open

PRINT unable to open file

}

**Design pseudocode that will print out course information and prerequisites. Print Course: This will print the course title and the prerequisites for any individual course.**

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

ITERATE through vector

IF key does not equal UINT\_Max

PRINT course name, number, and prerequisites

SET node equal to next node

WHILE node is not null

PRINT course name, number, and prerequisites

SET node equal to next node

}

**Sort the course information by alphanumeric course number from lowest to highest.** void sortInOrder(Hashtable<Course> courses, String courseNumber)

SORT keys from low to high

create min, current, and temp node variables to hold node info

SWAP nodes based on course number

PRINT courses by key

// Tree pseudocode

**Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.**

**Your pseudocode should show how to create course objects, so that one course object holds data from a single line from the input file.**

int numPrerequisiteCourses(Tree<Course> courses) {

INITIALIZE class Node for node structure

create data int

create node left

create node right

PASS val int into Node

SET data equal to val

SET left node as null

SET right node as null

CALL filename.open

IF file is open

CALL getline to read data by each line

SPLIT line at comma

STORE each value into splitLine variable

**Your pseudocode should show how to create course objects, so that one course object holds data from a single line from the input file.**

Create root node

Create courseName string to hold course name

For each line

IF splitLine length is equal to 2

SET root as course number

SET courseName as course name

IF split line length is greater than 2

SET root as course number

Set root->left node equal to smallest prerequisite

Set root->right node equal to largest prerequisite

IF there are less than two values from split line AND prerequisite is not in file

PRINT file format error

ELSE

IF file will not open

PRINT unable to open file

}

**Design pseudocode that will print out course information and prerequisites. Print Course: This will print the course title and the prerequisites for any individual course.**

void printCourseInformation(Tree<Course> courses, String courseNumber) {

FOR all courses

IF course number entered is equal to course number in tree

PRINT FOUND course number

PRINT course number, course name, and prerequisites (parent and children)

ELSE

PRINT course does not exist

}

**Sort the course information by alphanumeric course number from lowest to highest.**

void inOrder(Tree<Course> courses, String courseNumber)

Use inOrder call

IF node is not null

CALL inOrder with left node

Print course name and number

Recursively CALL inOrder with right node

// Vector Menu Pseudocode

While userInput != 9

print “Menu”

PRINT “1. Load Course Data

2. Print Course List

3. Print Course

9. Exit”

PRINT “Enter choice”

Accept userInput

Switch (userInput)

Case 1:

Complete call to load course data

loadData()

Break

Case 2:

Call InOrder to print all courses in order alphanumerically

InOrder()

Break

Case 3:

Call method to print all courses with associated prerequisites

Loop to display data read

For int I = 0, with I less than data size, ++I

displayCourse(courses[I]);

Break;

Case 4:

PRINT “Menu exited, goodbye.”

// Hash Table Menu Pseudocode

While userInput != 9

print “Menu”

PRINT “1. Load Course Data

2. Print Course List

3. Print Course

9. Exit”

PRINT “Enter choice”

Accept userInput

Switch (userInput)

Case 1:

Complete call to load course data

loadData(csvPath, courseTable)

Break;

Case 2:

Call sortInOrder to sort all courses in order alphanumerically

bidTable->sortInOrder()

Bidtable->PrintCourseInformation()

Break;

Case 3:

Call method to print all courses with associated prerequisites

courseTable->PrintCourseInformation()

Break;

Case 4:

PRINT “Menu exited, goodbye.”

Break;

// Binary Tree Menu Pseudocode

While userInput != 9

print “Menu”

PRINT “1. Load Course Data

2. Print Course List

3. Print Course

9. Exit”

PRINT “Enter choice”

Accept userInput

Switch (userInput)

Case 1:

Complete call to load course data

loadData(csvPath, bst)

Break;

Case 2:

Call InOrder to print all courses in order alphanumerically

Bst->InOrder

Break;

Case 3:

Call method to print all courses with associated prerequisites

Bst->PrintCourseInformation

Break;

Case 4:

PRINT “Menu exited, goodbye.”

Break;

Return 0;

## Example Runtime Analysis

| **Code**  **Reading File Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| CALL filename.open | 1 | 1 | 1 |
| IF file is open | 1 | 1 | 1 |
| CALL getline to read data by each line | 1 | n | n |
| SPLIT line at comma | 1 | n | n |
| STORE each value into splitLine variable | 1 | n | n |
| IF there are less than two values from split line AND prerequisite is not in file | 1 | n | n |
| PRINT file format error | 1 | 1 | 1 |
| ELSE IF file will not open | 1 | n | n |
| PRINT unable to open file | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 4 |
| **Runtime** | | | O(n) |

| **Code**  **Creating Objects Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE new course object (vector<course> courses) | 1 | 1 | 1 |
| IF splitLine length is equal to 2 | 1 | n | n |
| create course.name | 1 | n | n |
| create course.number | 1 | n | n |
| IF split line length is greater than 2 | 1 | n | n |
| Create course.prerequisites | 1 | n | n |
| ADD course objects to vector allCourses | 1 | n | n |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | O(n) |

| **Code**  **Reading File Hash** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| CALL filename.open | 1 | 1 | 1 |
| IF file is open | 1 | 1 | 1 |
| CALL getline to read data by each line | 1 | n | n |
| SPLIT line at comma | 1 | n | n |
| STORE each value into splitLine variable | 1 | n | n |
| IF there are less than two values from split line AND prerequisite is not in file | 1 | n | n |
| PRINT file format error | 1 | 1 | 1 |
| ELSE IF file will not open | 1 | n | n |
| PRINT unable to open file | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 4 |
| **Runtime** | | |  |

| **Code**  **Creating Objects Hash** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| int numPrerequisiteCourses(Hashtable<Course> courses) | 1 | 1 | 1 |
| INITIALIZE int key | 1 | 1 | 1 |
| INITIALIZE node \*Next | 1 | 1 | 1 |
| CONSTRUCT node with key as uint\_max and next as nullptr | 1 | 1 | 1 |
| SET tableSize as default\_size | 1 | 1 | 1 |
| Initialize node structure by resizing tableSize | 1 | 1 | 1 |
| Initialize logic to free storage when class is destroyed | 1 | 1 | 1 |
| IF splitLine length is greater or equal to 2 | 1 | n | n |
| CREATE a key for given course with a hash function | 1 | n | n |
| IF no entry found for key, assign a new node | 1 | n | n |
| IF oldNode is null | 1 | n | n |
| new Node(course, key) | 1 | n | n |
| INSERT new node | 1 | n | n |
| ELSE if node is unoccupied | 1 | n | n |
| UPDATE key with new key | 1 | n | n |
| UPDATE course data and change next pointer to null | 1 | n | n |
| ELSE if there is a collision, use chaining to find open node | 1 | n | n |
| **Total Cost** | | | 10n + 7 |
| **Runtime** | | | O(n) |

| **Code**  **Reading File Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| CALL filename.open | 1 | 1 | 1 |
| IF file is open | 1 | 1 | 1 |
| CALL getline to read data by each line | 1 | n | n |
| SPLIT line at comma | 1 | n | n |
| STORE each value into splitLine variable | 1 | n | n |
| IF there are less than two values from split line AND prerequisite is not in file | 1 | n | n |
| PRINT file format error | 1 | 1 | 1 |
| ELSE IF file will not open | 1 | n | n |
| PRINT unable to open file | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 4 |
| **Runtime** | | | O(n) |

| **Code**  **Creating Objects Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE class Node for node structure | 1 | 1 | 1 |
| create data int | 1 | 1 | 1 |
| create node left | 1 | 1 | 1 |
| create node right | 1 | 1 | 1 |
| PASS val int into Node | 1 | 1 | 1 |
| SET data equal to val | 1 | 1 | 1 |
| SET left node as null | 1 | 1 | 1 |
| SET right node as null | 1 | 1 | 1 |
| Create root node | 1 | 1 | 1 |
| Create courseName string to hold course name | 1 | 1 | 1 |
| For each line | 1 | n | n |
| IF splitLine length is equal to 2 | 1 | n | n |
| SET root as course number | 1 | n | n |
| SET courseName as course name | 1 | n | n |
| IF split line length is greater than 2 | 1 | n | n |
| SET root as course number | 1 | n | n |
| Set root->left node equal to smallest prerequisite | 1 | n | n |
| Set root->right node equal to largest prerequisite | 1 | n | n |
| **Total Cost** | | | 8n + 10 |
| **Runtime** | | | O(n) |

**Evaluation and Recommendation**

The advantages of using a vector would be a shorter total cost for creating objects and easy implementation of algorithms like quick sort. However, the disadvantages are also linked to its algorithms, since it can slow down run times.

The advantages of using a hash table would be its efficiency in holding data, since each bucket is assigned with a key. A huge disadvantage of using a hash table would be the lack of sorting support. There are ways that you can ‘sort’ a hash table using workarounds, however these methods are expected to be more complex and complicated.

The biggest advantage of a tree is that it is already sorted from highest to lowest since lower values are on the left side of the tree and higher values are on the right side. This directly coincides with the use cases brought up by ABCU. However, a tree is disadvantageous because it needs to be properly balanced to yield the best results.

I would recommend that a data structure of a search tree is used for ABCU’s requirements. Search trees prove to be best because it is naturally sorted, which already serves a purpose for ABCU. Search trees are also sufficient due to the ease of inserting and deleting items, which is helpful if there are any changes to the course catalog. Hash tables cannot sort and vectors require other algorithms to sort in order. This avoids the obstacle of implementing additional slow algorithms and convoluted solutions.