Matthew Muller

**Pseudocode**

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

Create file object of type ifstream

Open “chosenFile” using file object

Declare String Array courseCheck

WHILE end of file is not reached

Get line from file

Split line

Append first String from line into courseCheck Array

WHILE end of file is not reached

Get line from file

Array courseInfo equals split line

IF courseInfo does not have at least 2 elements

Print “Incorrect file format”

FOR each element of courseInfo after the 2nd

IF element is not in courseCheck Array

Print “Non-existent Prerequisite”

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

**Pseudocode for all three data structures begins with:**

Create Course CLASS

Declare courseNumber variable of type String

Declare courseName variable of type String

Declare prerequisites Vector with String data type

Open file for reading

WHILE end of file is not reached

Get line from file

Split line at commas

Create new Course object course

courseNumber equals first data value from line

courseName equals second data value from line

FOR each remaining data value on line

Append Course object’s prerequisites Vector with courseNumber data value

**Pseudocode differs after this point**

**Vector:**

Append newly created Course object to courses Vector

**HashTable:**

Declare internal structure for node

Course object course

Pointer to next node

Default constructor

Initialize with course

Add Course object to hash table

Bucket equals courseNumber % 100

Create new node and initialize it with Course object

node->next = null

Append new node to bucket’s linked list

**Tree:**

Declare internal structure for node

Course object course

Pointers to children nodes

Default constructor

Initialize with course

Add Course object to tree data structure

Create new node and initialize it with Course object

If root is null

Root equals new node

Else

currentNode = root

While currentNode is not nullptr

If object’s courseNumber < currentNode courseNumber

If currentNode->left is nullptr

currentNode->left equals new node

currentNode = null

else

currentNode = currentNode->left

Else

If currentNode->right is nullptr

currentNode->right equals new node

currentNode = null

else

currentNode = currentNode->right

1. **Design pseudocode that will print out course information and prerequisites.**

**Vector:**

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

FOR each course in courses

IF course’s courseNumber equals courseNumber parameter

Print courseNumber and name

FOR each prerequisite of the course

Print prerequisite’s courseNumber

}

**HashTable:**

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

Create key by hashing courseNumber

Retrieve node with key and set new node variable to it

While node is not nullptr

If node’s course’s courseNumber equals courseNumber parameter

Print course information

For each prerequisite of course

Print prerequisite’s course information

Else

node = node->next

}

**Tree:**

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

Set current node equal to root

While current is not nullptr

If courseNumber equals current->course.courseNumber

Print current node’s object’s courseNumber and name

FOR each prerequisite of the course

Print prerequisite’s courseNumber

Else if courseNumber < current->course.courseNumber

current = current->left

Else

current = current->right

}

**Runtime Analysis**

| **Vector** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Create vector | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Create course item | 1 | n | n |
| While prerequisite exists | 1 | n | n |
| Append prerequisite to vector | 1 | n | n |
| Pushback course item | 1 | n | n |
|  |  | Total Cost | 5n + 1 |
|  |  | Runtime | O(n) |

| **HashTable** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Create hash table | 1 | 1 | 1 |
| Insert course method | 0 | 0 | 0 |
| Create key for course | 1 | n | n |
| If no entry found for key | 1 | n | n |
| Assign node to key | 1 | n | n |
| Else | 1 | n | n |
| Assign old node key to UNIT\_MAX, set to key, set old node to course and old node next to null pointer | 4 | n | 4n |
| Else | 1 | n | n |
| Find next open node | 1 | n | n |
| Append new node | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Insert course item | 1 | n | n |
|  |  | Total Cost | 16n + 1 |
|  |  | Runtime | O(n) |

| **Tree** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Create tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| If root is nullptr, add root | 1 | 1 | 1 |
| If node is less than root, proceed to left | 1 | n | n |
| If no left node | 1 | n | n |
| This node becomes left | 1 | n | n |
| If node is greater than root, proceed to right | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prerequisite exists | 1 | n | n |
| Append prerequisite | 1 | n | n |
| Insert course item | 1 | n | n |
|  |  |  | 11n + 2 |
|  |  |  | O(n) |

**Advantages and Disadvantages**

**Vector:**

The main advantage of the vector data structure when it comes to this program is its speed at adding and removing course items. It has the lowest total runtime cost at 5n+1, although all three share the same O(n) notation. The main disadvantage of the vector data structure for this program is its speed when it comes to searching for a specific course. Unlike the other two data structures, in order to find a course, each item in the vector must be checked in order until a match is found.

**HashTable:**

The main advantage of the hashtable data structure when it comes to this program is its speed at searching for a specific course. Hashing a course’s node using its key allows for extremely fast searching as the program simply has to check the bucket's list after being mapped to it by the key. The main disadvantage of the hashtable data structure for this program is its inability to sort itself. In order to print all of the courses in alphanumeric order, the course objects must be extracted from the table, separately sorted, and then printed.

**Tree:**

Like the hashtable data structure, the main advantage of the tree data structure when it comes to this program is its speed at searching for a specific course. Unlike a vector, a tree does not have to check each item in order to find a specific course and can instead implement a binary search. Trees are also very useful for sorting the course objects as a binary search tree essentially has the items already in order. The biggest disadvantage of the tree data structure for this project is the complexity of adding and deleting courses.

**Recommendation**

I would recommend that ABCU goes with a tree data structure for this program. Implementing a binary search tree will make it easy to keep the courses in alphanumeric order and it will provide a good level of speed for searching for specific courses. The only disadvantage of the tree data structure for this program is the complexity of adding and deleting nodes. However, this is likely an action that will not be utilized as frequently as the search or print functions of the program. For these reasons, I believe that a binary search tree is the most appropriate data structure for this system.