**6-2 Project One:**

**Vector Pseudocode:**

void coursePrerequisites(Vector<Course> courses, Course c) {

Open File using fstream

If file failed to open

Output an error message

Else, the file opened

While the file is not at the end

loadCourses();

Close File

}

Vector<Course> loadCourses(string csvPath) {

Declare vector to hold courses vector<Course> courses

Initialize CSV Parser with path

For loop to read rows until end of CSV File

Course course

Course.courseNumber = file[i][0]

Course.courseName = file[i][1]

If more than two values on a line

If file[i][i] is equal to a courseName

Course.prerequisite = file [i][i]

Output course number, course name, and any prerequisites

Push back course to end of vector list

Output error message if CSV File wasn’t opened

Return Course

Struct Course {

string courseNumber

string courseName;

string prerequisite;

};

void printSampleSchedule(Vector<Course> courses) {

courses = loadCourses(csvPath);

For loop from beginning of csv to end of file

Output course number, name, and any prerequisite for course at i index

}

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

For all courses

If the course is the same as courseNumber

Output the course information

For each prerequisite of the course

Output the prerequisite course information

}

**Hash Table Pseudocode:**

Void coursePrerequisites(Hashtable<Course> courses) {

Open File using fstream

If file failed to open

Output an error message

Else, the file opened

While the file is not at the end

loadCourses();

Close File

}

Void loadCourses(string vscPath, HashTable\* hashTable) {

Load CSV File

Initialize CSV Parser with path

For loop from beginning of file to end

Course course;

Course.courseNumber = file[i][0];

Course.courseName = file[i][1];

If more than two values on a line

If file[i][i] is equal to a courseName

Course.prerequisite = file [i][i];

Output courseNumber, courseName, and any prerequisite;

Push course to end of binary search tree;

Output error if file wasn’t opened;

}

struct Course {

string courseNumber;

string courseName;

string prerequisite;

};

struct Node {

Course course;

unsigned int key;

Node \*next;

// default constructor

Node() {

key = UINT\_MAX;

next = nullptr;

}

void printSampleSchedule(Hashtable<Course> courses) {

For loop from beginning node until the end

If key is not equal to UINT\_MAX

Output iterator key, course number, course name, and prerequisite

Node is assigned to the next iterator

While node is not equal to null

Output node key, course number, course name, and prerequisite

Set node equal to the next node

}

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

Course course;

Assign integer key to hash

Node\* node is assigned to key node

If node is not null and key is not UNIT\_MAX and node and searched course number are the same

Return node course

If node is null and node key is equal to UINT\_MAX

Return course

While node is not null

If current node is not UINT\_MAX and searched and current course Number are the same

Return node bid

Node is assigned to the next node

Return course;

}

**Tree Pseudocode:**

Void coursePrerequisites (Tree<Course> courses) {

Open File using fstream

If file failed to open

Output an error message

Else, the file opened

While the file is not at the end

loadCourses();

Close File

}

void loadCourses(string csvPath, BinarySearchTree\* bst) {

Initialize CSV Parser with Path

For loop to read rows of the CSV file

Course course

Course.courseNumber = file[i][0];

Course.courseName = file[i][1]

If more than two values on a line

If file[i][i] is equal to a courseName

Course.prerequisite = file [i][i]

Output courseNumber, courseName, and any prerequisite.

Push course to end of binary search tree

Output error if file wasn’t opened

}

struct Course {

string courseNumber

string courseName;

string prerequisite;

};

struct Node {

Course course;

Node \*left;

Node \*right;

// default constructor

Node() {

Assign left to nullptr

Assign right to nullptr

}

};

void printSampleSchedule(Tree<Course> courses) {

if root is equal to nullptr {

Root is equal to new course

}

else root is not null {

Add Node root and course

}

};

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

Set current node equal to the root

While current node is not null

If current courseNumber is the same as specified courseNumber

Print current courseNumber

If specified courseNumber is less than current courseNumber

Traverse left current is assigned to current left

Else if specified cNumber is greater than current cNumber

Traverse right current is assigned to current right

Course courseNumber

Return courseNumber

}

**Menu Pseudocode:**

Output menu options

Initialize choice to 0

While choice is not equal to 9

If user chooses 1

Use loadCourse(csvPath) function to load courses

If user chooses 2

For loop from start to size of courses list

Use orderedCourses() function to display all courses

If user chooses 3

Ask user for courseNumber

Return corresponding CourseName and any prereuqiusites

If user chooses 9

Display Goodbye message

return 0 and exit

**Alphanumeric Order Vector Pseudocode (Selection Sort):**

orderedCourses(course, size)

Initialize i, j, indexSmallest, and temp to 0

For loop from i = 0 to i is less than size -1

Initialize indexSmallest to i

For loop from j = i + 1 to j is less than size

If course[j] is less than course[indexSmallest]

Assign indexSmallest to J

Temp is assigned to course[i]

course[i] is assigned to course[indexSmallest]

course[indexSmallest] is assigned to temp

temp = course[i]

course[i] = course[indexSmallest]

course[indexSmallest] = temp

}

Main() {

For loop from i=0 to i is less than size

Output course[i]

}

**Alphanumeric Order Hash Table Pseudocode (Array):**

Create string array orderedCourses and set to length of hash table

For each course in Course

OrderedCourses[i] is equal to key as a string

Iterate i

Sort orderedCourses

For each course in Course

Print course

**Alphanumeric Order Tree Pseudocode (InOrder):**

orderedCourses(node) {

If node is not null

orderedCourses(node -> left)

Print node information (name, number, prerequisites)

orderedCourses(node⇢right)

}

**Big-O Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create vector** | 1 | 1 | 1 |
| **Read in file** | 1 | 1 | 1 |
| **For each line** | 1 | n | n |
| **Create course object** | 1 | n | n |
| **For each prerequisite** | 1 | n | n |
| **Create prerequisite object** | 1 | n | n |
| **Total Cost** | | | 4n+2 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create Hash Table** | 1 | 1 | 1 |
| **Read in file** | 1 | 1 | 1 |
| **Create key for courses** | 1 | n | n |
| **Create key hash** | 1 | n | n |
| **Assign key to UINT\_MAX** | 1 | n | n |
| **Assign key to a node** | 1 | n | n |
| **For all nodes** | 1 | n | n |
| **If key isn’t UINT\_MAX** | 1 | n | n |
| **Output key and course info** | `1 | n | n |
| **Assign node to next it** | 1 | n | n |
| **While node is not null** | 1 | n | n |
| **Output node and course info** | 1 | n | n |
| **Assign node to next node** | 1 | n | n |
| **Total Cost** | | | 11n + 2 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Binary Search Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Create Tree** | 1 | 1 | 1 |
| **Read in file** | 1 | 1 | 1 |
| **If root is null** | 1 | n | n |
| **Root is assigned to new course** | 1 | n | n |
| **If root isn’t null** | 1 | n | n |
| **Add node root and course** | 1 | n | n |
| **Set current node to root** | 1 | n | n |
| **While current is not null** | 1 | n | n |
| **If current course# is same as specified course#** | 1 | n | n |
| **Print current course#** | 1 | n | n |
| **If specified course# is less than current course#** | 1 | n | n |
| **Traverse left** | 1 | n | n |
| **If spec course# is greater than current course#** | 1 | n | n |
| **Traverse Right** | 1 | n | n |
| **Total Cost** | | | 12n + 2 |
| **Runtime** | | | O(n) |

**Advantages and Disadvantages of Each Structure:**

There are advantages and disadvantages of each data structure. One advantage of vectors is that it is easy to insert or remove items into it. In addition, you are able to store many items in a vector, which is good for the ABC University application, which will need to store and access a lot of information in whichever ADT is chosen to be used. In addition, vectors can easily be copied, which can help aid if a vector is being edited or transferred. There are, however, some disadvantages of the vector abstract data type. They tend to run slower than other data structures and use up more memory compared to hash tables and binary search trees. ABC University will need to access this information frequently and have a lot of data to store, meaning these two issues will be important to their overall efficiency. Thankfully, however, their slower speed is almost negligible compared to other structures. This means it is still a valid option for ABC University’s computer science department.

There are also distinct advantages and disadvantages of using a hash table for holding our data. One advantage is that a hash table can store a lot of information and can be accessed quickly and easily. Therefore, hash tables are considered a more efficient data structure when compared to vectors or binary search trees. One major disadvantage, however, is that hash tables are prone to collisions, which can slow down and ultimately destroy or corrupt the information included within. The bigger the data set, the more likely it is to have a collision. Since ABC University’s computer science department will house a large amount of their data in this data structure, it is important that it is secure and free from flaws. Therefore, hash tables may not be the safest option for this case.

Binary search trees also come with their own unique advantages and disadvantages. One benefit of using a search tree is that elements are stored in a sorted fashion, making searching for an item quick and easy. In addition, it is easy to insert or remove an item from a search tree. Lastly, binary search trees are considered very efficient in regards to the time and space they take up. There are some disadvantages, though. In an unbalanced binary search tree, the information becomes skewed, which causes efficiency to decrease. This would be considered the worst case and would have a very slow time complexity compared to other data structures. Since binary search trees can easily become unbalanced, it is often best practice to not use them when you have a large amount of data that needs to be stored. Since ABC University’s computer science department will be storing a large amount of information in the data structure chosen, I can assume that a binary search tree is not the most practical or appropriate choice for the given circumstance.

**Final Recommendation:**

Overall, it is my recommendation to use a vector data structure to house the information provided by ABC University’s computer science department. I came to this conclusion after analyzing the pseudocode, the big O analysis, and advantages and disadvantages of each of the three abstract data types. The Big-O analysis time cost for a vector is 4n+2. This is relatively much smaller than compared to the time costs for the hash table and binary search table, which were 11n + 2 and 12n + 2, respectively. The data structure with the lowest total time cost will be the one that runs the most efficiently. This is an indication that for our particular use, a vector would be the most suitable and appropriate option for storing the data. In addition, when considering the pros and cons of all three data structures, one could see why vectors are the most appropriate for this scenario. One of the main reasons why a vector is suitable for the particular needs of ABC University is that a vector abstract data type can easily hold a large amount of information. We know there will be a lot of information needed to be held in this structure, including all the course names, numbers, and prerequisites for each course in the computer science department. It is important to utilize a data structure that will store the information in a secure and efficient way. Due to a vector’s ability to store large amounts of data while keeping it secure and efficient to alter or access, as well as having the lowest time costs of all three potential data structures, I believe that the vector data structure is the best option for the ABC University’s computer science department and their needs.