Project One

**Pseudocode**

//Updated code to reflect template in pseudocode document

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

**The following code is shared by all three data structures:**

//Returns true if prerequisite already exists in courseArrayVect

bool checkPrereq(vector<array> courseArrayVect, string t\_prereq) {

//Initialize prereqFound to false

bool prereqFound = false

//Check if prereq exists in courseArrayVect

For all arrays in courseArrayVect {

array tempArray = courseArrayVect.at(i)

//Prereq found

if (tempArray[0] == t\_prereq) {

prereqFound = true

}

}

return prereqFound

}

//Opens and reads each line to an array of strings. Uses checkPrereq() method to verify prerequisite requirements have been met

void readFile(string filePath) {

string tempLine

array lineArray

vector<array> tempCourseList

open filestream

if file not open

print “error: unable to open file”

while not end of file

getline into tempLine

//reset lineArray

lineArray = []

while not end of stringstream split tempLine into substr

lineArray.push(substr)

//Check if courseId and courseName data is valid

if (lineArray[0].length() == 7 && lineArray[1] != null) {

//If prerequisites exist

if (lineArray.size() > 2) {

//Verify prereqs already exist in tempCourseList

bool allPrereqsMet = false;

for (i = 3; i < lineArray.size(); i++) {

if(checkPrereq(tempCourseList, lineArray[i]) {

allPrereqsMet = true

} else {

allPrereqsMet = false

print lineArray[i] does not meet prerequisite requirement

}

}

//If all prerequisites were found, add lineArray to tempCourseList

if(allPrereqsMet) {

tempCourseList.push\_back(lineArray)

} else {

print “error: prerequisite requirement not met”

}

}

//No prerequisites. CourseId and CourseName valid. Add to list.

tempCourseList.push\_back(lineArray)

}

//courseId or courseName not valid

print “error: invalid course ID or course Name data. skipping line”

}

**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 course class

class Course {

public:

//Define variables

string courseId = “noCourseId”

string courseName = “noCourseName”

vector<string> prereqs

//Setters

void SetCourseId(string t\_courseId) {

courseId = t\_courseId

}

void SetCourseName(string t\_courseName) {

courseName = t\_courseName

}

void AddPrereq(string t\_prereq) {

prereqs.push\_back(t\_prereq)

}

//Getters

string GetCourseId() {

return courseId

}

string GetCourseName() {

return courseName

}

vector<string> GetPrereqVect() {

return prereqs

}

void PrintPrereqs() {

for (int i = 0; i < prereqs.size(); i++) {

print “Prereq[“ + i + “]:” + prereqs[i] + “ “

}

}

**Storing objects in vector:**

// Method to create a vector of courses from vector of arrays

void CreateCoursevector(vector<array> courseArrayVect) {

//Initialize vector of courses

vector<Course> courseVect

For all arrays in courseArrayVect {

Course tempCourse

array tempArray = courseArrayVect.at(i)

//All arrays have courseId and courseName

tempCourse.SetCourseId(tempArray[0])

tempCourse.SetCourseName(tempArray[1])

//Prerequisites

if (tempArray.size() > 2) {

for all prerequisites

AddPrereq(t\_prereq)

}

//Add tempCourse to courseVect

courseVect.push\_back(tempCourse)

}

}

**Storing objects in Hash Table:**

// Create HashTable class

class HashTable {

private:

struct Node {

Course course

int key

Node \*next

//default constructor

Node() {

key = UINT\_MAX

next = nullptr

}

Node(Course aCourse) : Node() {

course = aCourse

}

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

key = aKey

}

};

vector<Node> nodes

int tableSize = size

int hash(string courseId)

public:

HashTable()

void Insert(Course course)

void PrintAll()

}

//Default constructor

HashTable::HashTable() {

nodes.resize(tableSize)

}

//Hash algorithm to convert courseId (CSCI100) to integer

int HashTable::hash(string courseId) {

int temp = convert string to integer

key = temp % tableSize

return key

}

// Insert a course

void HashTable::Insert(Course course) {

key = hash(course.GetCourseId())

Node oldNode = node at key

if oldNode equals nullptr

Node newNode = new Node(course, key)

insert newNode at key position

else

if oldNode key equals UINT\_MAX

oldNode key equals key

oldNode course equals course

oldNode next equals nullptr

else

while oldNode next doesn’t equal nullptr

oldNode equals oldNode next

oldNode next equals new Node(course, key)

}

// Method to load all courses from array and insert to HashTable

//create HashTable to store courses

HashTable\* courseTable = new HashTable();

void LoadCoursesToHashTable(vector<array> courseArrayVect) {

For all arrays in courseArrayVect {

Course tempCourse

array tempArray = courseArrayVect.at(i)

//All arrays have courseId and courseName

tempCourse.SetCourseId(tempArray[0])

tempCourse.SetCourseName(tempArray[1])

if tempArray.size() > 2

for all prerequisites

AddPrereq(t\_prereq)

courseTable.Insert(tempCourse)

}

**Storing objects in Binary Tree:**

// Create BinarySearchTree class

class BinarySearchTree {

private:

Node\* root

void addNode(Node\* node, Course course)

void inOrder(Node\* node)

void preOrder(Node\* node)

void postOrder(Node\* node)

public:

BinarySearchTree()

void InOrder()

void PreOrder()

void PostOrder()

void Insert(Course course)

}

// Default constructor

BinarySearchTree::BinarySearchTree() {

root = nullptr

}

BinarySearchTree::Insert(Course course) {

if root equals nullptr

root equals new Node(course)

else

this addNode(root, course)

}

void BinarySearchTree::addNode(Node\* node, Course course) {

if node courseId > course courseId

if node left equals nullptr

node left equals new Node(course)

else

this addNode(node left, course)

else

if node right equals nullptr

node right equals new Node(course)

else

this addNode(node right, course)

}

// Method to load all courses from array and insert into BinarySearchTree

//Create BinarySearchTree to store courses

BinarySearchTree\* bst = new BinarySearchTree();

void LoadCoursesToBST(vector<array> courseArrayVect) {

For all arrays in courseArrayVect {

Course tempCourse

array tempArray = courseArrayVect.at(i)

//All arrays have courseId and courseName

tempCourse.SetCourseId(tempArray[0])

tempCourse.SetCourseName(tempArray[1])

if tempArray.size() > 2

for all prerequisites

AddPrereq(t\_prereq)

bst.Insert(tempCourse)

}

**Print objects from Vector:**

void PrintAll(vector<Course> courseVect) {

Course tempCourse

for all courses in courseVector

tempCourse = courseVector.at(i)

print tempCourse.GetCourseId() tempCourse.GetCourseName() tempCourse.PrintPrereqs()

}

**Print objects from Hash Table:**

void HashTable::PrintAll() {

Node node

Course course

for all nodes

node = node at iterator

//print first course in bucket

if node key doesn’t equal UINT\_MAX

print course.courseId course.courseName course.PrintPrereqs()

//print any chained courses

while node next doesn’t equal nullptr

course = node next course

print course.courseId course.courseName course.PrintPrereqs()

node = node next

}

**Print objects from Binary Tree:**

void BinarySearchTree::InOrder() {

this inOrder(root)

}

void BinarySearchTree::inOrder(Node\* node) {

if node not equal to nullptr

inOrder(node left)

print node course.courseId node course.courseName node course.PrintPrereqs()

inOrder(node right)

}

void BinarySearchTree::PreOrder() {

this preOrder(root)

}

void BinarySearchTree::preOrder(Node\* node) {

if node not equal to nullptr

print node course.courseId node course.courseName node course.PrintPrereqs()

preOrder(node left)

preOrder(node right)

}

void BinarySearchTree::PostOrder() {

this postOrder(root)

}

void BinarySearchTree::postOrder(Node\* node) {

if node not equal to nullptr

postOrder(node left)

postOrder(node right)

print node course.courseId node course.courseName node course.PrintPrereqs()

}

**Pseudocode for menu for all three data structures:**

bool exitMenu equals false

while (!exitMenu)

print 1. Load Courses (Vector)

print 2. Load Courses (Hash Table)

print 3. Load Courses (Binary Search Tree)

print 4. Print All Courses (Vector)

print 5. Print All Courses (Hash Table)

print 6. Print All Courses (Binary Search Tree)

print 7. Print Course Info (Vector)

print 8. Print Course Info (Hash Table)

print 9. Print Course Info (Binary Search Tree)

print 0. Exit

String menuEntry = next input

char menuSingleChar = menuEntry.at(0)

switch (menuSingleChar)

case ‘1’

vector<Course> courses

courses equals LoadBids(csvPath)

break

case ‘2’

courseTable equals new HashTable();

courseTable equals loadBids(csvPath, bidTable)

break

case ‘3’

bst equals new BinarySearchTree()

bst equals loadBids(csvPath, bst)

break

case ‘4’

for all courses

print course[i].courseId course[i].courseName course[i].PrintPrereqs()

break

case ‘5’

courseTable->PrintAll()

break

case ‘6’

bst->InOrder()

break

case ‘7’

print “Enter course number to print details”

string search equals next input

PrintCourseVect(courses, search)

break

case ‘8’

print “Enter course number to print details”

string search equals next input

courseTable->PrintCourseHT(search)

break

case ‘9’

print “Enter course number to print details”

string search equals next input

bst->PrintCourseBST(search)

break

case ‘0’

print “Exiting.”

exitMenu equals true

break

default

print “Please enter a valid menu selection”

break

**Sorting Vector:**

void InsertionSortVector(vector<Course> courseVect) {

// Create copy of vector passed as argument to sort

vector<Course> sortedCourseVect = courseVect

Course tempCourse

int i = 0

int j = 0

for (i = 1; i < sortedCourseVect.size(); ++i)

j = i

while (j > 0 && sortedCourseVect.at(j).getCourseId().compare(sortedCourseVect.at(j - 1.GetCourseId())) < 0)

tempCourse = sortedCourseVect.at(j)

sortedCourseVect.at(j) = sortedCourseVect.at(j - 1)

sortedCourseVect.at(j - 1) = tempCourse

--j

**Sorting Hash Table:**

**//** Hash tables work by mapping keys to values and part of what makes it a hash table is that the //keys aren’t sorted or stored in any order.

**Sorting a Binary Tree:**

// Use an InOrder traversal of the BST to print sorted results

void BinarySearchTree::InOrder() {

this inOrder(root)

}

void BinarySearchTree::inOrder(Node\* node) {

if node not equal to nullptr

inOrder(node left)

print node course.courseId node course.courseName node course.PrintPrereqs()

inOrder(node right)

}

**Evaluation**

**Evaluate the run-time and memory of data structures that could be used to address the requirements.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Load n-items | Sort | PrintSorte | Search |
| Vector | O(n) | O() | O() | O(n) |
| Hash Table | O(n) | X | X | O(1) |
| Binary Tree | O(n) | Completed during load | O(n) | O(log n) |

**Explain the advantages and disadvantages of each structure in your evaluation.**

As can be seen from the table each of the data structures require O(n) for loading n items. The vector is slightly more efficient as it pushes each line of the csv. file to the back of the vector.

Hash Tables require a hashing algorithm, making the code for each item longer by a line or two.

Binary Trees require traversal of the existing tree with each insert, which means the bigger the tree becomes the need from more lines of code (recursive calls) for each item inserted increases.

Hash Tables, as is seen from the table, are the least useful data structure. Because items are hashed and assigned a key for placement in the data structure with no relation to sorted order.

Vectors can use several algorithms for its sorting needs, with some choices being more efficient and speedier.

The insertion sort algorithm has a worst case runtime of O(), that the vector sort starts from a reverse sorted state. The Binary Tree has the best data structure for sorting as each item is loaded already in a sorted position upon creation.

When it comes to printing a sorted list of all items the vector must use a sorting algorithm, then print each item. Since the insertion sort has a worst case of O() and the printing is O(n), when combined this reduced to O(). As Hash Tables aren’t suited for sorting this means they aren’t suited for printing a list of sorted items. This means that, again, the Binary Tree is the most efficient at printing a list of sorted items. The Binary Tree requires printing each node plus traversing each edge connecting the item in the tree; meaning it is more efficient than a vector.

When it comes to searching, if an item being searched for is last on a Vector then that means that every single item would have to be checked from first to last. This can be improved is a Vector is sorted with algorithms like a Binary Tree. Searching a Hash Table is very efficient for specific entries, as the item to be searched is hashed in the same way as the other hashed items entered in the table; which means only one key need be checked. However, if a Hash Table only contains one bucket, meaning all items are stored under the same key, then this greatly reduces the efficiency of a Hash Table. Sufficient buckets and a proper hashing algorithm is important for the benefits of a Hash Table. The Binary Tree is the stronger option when it comes to searching. A Binary Tree is slowed if O(n) occurs due to items being inserted in a sorted order. However, when items are inserted at random, the tree will be more balanced and thus more efficient.

**Make a recommendation for which data structure you will plan to use in your code.**

My recommendation for ABCU and their requested course information system, I would recommend a Binary Search Tree data structure. For their application it is important that they be able to print a sorted list of all the courses that have been loaded. The Binary Search Tree would also allow the program to search for a course so that the information can be printed. The Binary Search Tree has been shown to excel at both these requested requirements.