Scott Cain

CS 300 Project One

12/27024

**Evaluation of Data Structures**

**Vector:**

**Reading Data from File**: Reading each line from the file and processing it takes linear time. This means that the time taken to read the file increases directly with the number of courses in the file. This time complexity would be shown as O(n).

**Creating Course Objects:** This has a constant and does not depend on the number of courses. This time complexity is O(1) for each course.

**Inserting into Vector:** This operation takes a constant amount of time. This time complexity is O (1) for each insertion.

**Sorting Courses:** Sorting the vector of courses to display time complexity will be O(n log n) time. This means the time taken increases slightly more than linearly with the number of courses.

**Hash Table:**

**Reading Data from File**: Reading each line from the file and processing it takes linear time. This means that the time taken to read the file increases directly with the number of courses in the file. This time complexity would be shown as O(n).

**Creating Course Objects:** This has a constant and does not depend on the number of courses. This time complexity is O (1) for each course.

**Inserting into Hash Table:** This operation takes a constant amount of time. This time complexity is O (1) for each insertion.

**Sorting Courses:** The sorting needed here requires comparing and arranging the keys. Which. This time complexity will be O (n log n) time.

**Binary Search Tree:**

**Reading Data from File**: Reading each line from the file and processing it takes linear time. This means that the time taken to read the file increases directly with the number of courses in the file. This time complexity would be shown as O(n).

**Creating Course Objects:** This has a constant and does not depend on the number of courses. This time complexity is O (1) for each course.

**Inserting into Binary Search Tree**: Inserting for BST involves comparing keys and placing the course in the correct position. For a balanced tree, it would be O (n log n) time and for an unbalanced tree linear time or O (n).

**Sorting Courses:** Displaying courses in alphanumeric order involves an in-order traversal of the binary search tree, which takes linear time, O(n). This means the time taken increases directly with the number of courses.

**Advantages and Disadvantages**

**Vector:**

**Advantages:** Loading data into a vector using an append method is quick and easy. It's a simple data structure that's easy to implement and understand.

**Disadvantages:** Sorting the vector after it has been entered can become a time-consuming process, especially as the number of courses increases. Inserting and deleting items has the chance of becoming costly.

**Hash Table:**

**Advantages:** Hash tables offer quick insertion, deletion, and lookup on average, making them the best for quick access. They can handle data efficiently with minimal collisions.

**Disadvantages:** Can be slowed down due to collisions. Collisions need to be maintained for good performance. Sorting data in a hash table requires extracting and sorting keys which will not be as easy as Vector.

**Binary Search Tree:**

**Advantages:** BSTs provide ordered data and allow for efficient in-order traversal, making it easy to sort and access data in sequence.

**Disadvantages:** If a BST becomes unbalanced, its performance can degrade quickly and dramatically. Implementing and maintaining a balanced BST is more complex compared to vectors and hash tables.

**Recommendation**

Taking into account that the data will be loaded occasionally, printed less frequently, but searched often, the Hash Table stands out as the preferable choice. To optimize performance, the hash function should be well-designed, and the table size appropriately set to reduce collisions, maintaining efficiency close to O(1).

**Pseudocode (didn’t adjust since it was good)**

**Vector**

Opening and reading data

FUNCTION loadCourses(string csvPath, dataStructure)

OPEN file csvPath FOR reading

IF file cannot be opened THEN

PRINT "Error: File not found"

RETURN

END IF

WHILE not at end of file (EOF)

READ line from file

SPLIT line by comma into tokens

IF number of tokens < 2 THEN

PRINT "Error: Invalid line format"

CONTINUE

END IF

SET courseNumber = tokens[0]

SET courseName = tokens[1]

CREATE vector<String> prerequisites

FOR i = 2 to (number of tokens - 1)

ADD tokens[i] to prerequisites

END FOR

CREATE Course object with courseNumber, courseName, prerequisites

ADD Course object to dataStructure

END WHILE

CLOSE file

END FUNCTION

END

Validating the file format

FUNCTION validateFile(Vector<Course> courses)

CREATE set<String> courseNumbers

FOR each course IN courses

ADD course.courseNumber TO courseNumbers

END FOR

FOR each course IN courses

FOR each prerequisite IN course.prerequisites

IF prerequisite NOT IN courseNumbers THEN

PRINT "Error: Prerequisite " + prerequisite + " does not exist"

RETURN FALSE

END IF

END FOR

END FOR

RETURN TRUE

END FUNCTION

END

Course Objects and Storing

STRUCT Course

String courseID

String courseName

Vector<String> prerequisites

END STRUCT

FUNCTION createCourse(String courseID, String courseName, Vector<String> prerequisites)

CREATE Course object

SET object.courseID = courseID

SET object.courseName = courseName

SET object.prerequisites = prerequisites

RETURN object

END FUNCTION

FOR each line in file

SPLIT line by comma INTO tokens

SET courseID = tokens[0]

SET courseName = tokens[1]

CREATE Vector<String> prerequisites

FOR i = 2 to (number of tokens - 1)

ADD tokens[i] TO prerequisites

END FOR

CREATE course object USING createCourse(courseID, courseName, prerequisites)

ADD course object TO courses vector

END FOR

END

Searching for Specific Courses and Print

FUNCTION searchCourse(Vector<Course> courses, String courseID)

FOR each course IN courses

IF course.courseID == courseID THEN

PRINT "Course ID: " + course.courseID

PRINT "Course Name: " + course.courseName

PRINT "Prerequisites: "

IF course.prerequisites.size() == 0 THEN

PRINT "None"

ELSE

FOR each prerequisite IN course.prerequisites

PRINT prerequisite

END FOR

END IF

RETURN

END IF

END FOR

PRINT "Course not found"

END FUNCTION

END

FUNCTION loadCourses(string csvPath)

CREATE vector<Course> courses

CALL loadCourses(csvPath, courses)

RETURN courses

END FUNCTION

END

**Hash Table**

STRUCT Node

Course course

UNSIGNED INT key

Node\* next

END STRUCT

FUNCTION loadCourses(string csvPath)

CREATE hashTable with size of your choice

CALL loadCourses(csvPath, hashTable)

RETURN hashTable

END FUNCTION

END

Hash Table Data Structure

Struct to hold course data:

struct Course {

string courseID

string courseName

int preCount

list<string> preList

Course() {

courseID = ""

courseName = ""

preCount = 0

preList = "" }}

Class HashTable

class HashTable {

struct bucket {

Course course

unsigned int key

bucket\* next

}

unsigned int hash(string key)

void printAll()

vector<bucket> hashTable}

Main Function:

Function Main()

Create a new vector named courseList of type Course

Prompt user for the CSV file path

If no path is provided, use the default location

Call txtParser with the CSV file path

Call validateList with courseList

Prompt user to enter a course number to search for and store in userSearch

Call printCourse with userSearch

Txt Parser Function

Function txtParser(string filePath)

Open the file at filePath using parsing libraries

While not end of file (eof)

Read line from file

If line has at least two parts

Call hash function with the first part (courseID)

Insert into the hash table at the hashed index

Create a new Course object

Set Course.courseID to the first part

Set Course.courseName to the second part

While more columns exist in the line

Increment preCount for each prerequisite found

Append prerequisite to preList

Add preCount to the Course object

Add preList to the Course object

Return temporary list of courses

Search List Function:

Function searchList(string courseID)

Create a temporary bucket

Set the temporary bucket to the location in the hash table based on hashed courseID

For each course in the list

If courseID matches the course's courseID

Set the temporary bucket to the course found

Return the temporary bucket

Print Course Function:

Function printCourse(string courseID)

Create a temporary bucket

Set the temporary bucket to the location in the hash table based on hashed courseID

While the temporary bucket is not null

Print courseID and courseName from the Course struct in the temporary bucket

For each prerequisite in preList

Call printCourse with each prerequisite

Validate list Function:

Function validateList()

Create a temporary bucket

Set valid to True

For each course in the list

If valid is False, break

While temporary bucket.next is not null

For each prerequisite in preList

Set the temporary bucket to result of searchList with the prerequisite

If the temporary bucket's courseID is empty, set valid to False

Return valid

**Tree Data Structure**

Struct to hold the course data

struct Course {

string courseID

string courseName

list<string> preList

Course() {

courseID = ""

courseName = ""

preList = list<string>() }

}

Class Binary Search Tree

class BinarySearchTree {

struct Node {

Course course

Node\* left

Node\* right

Node(Course course) {

this->course = course

left = nullptr

right = nullptr

}

}

Node\* root

BinarySearchTree() {

root = nullptr

}

~BinarySearchTree() {

destroyNode(root)

}

void destroyNode(Node\* node) {

if (node != nullptr) {

destroyNode(node->left)

destroyNode(node->right)

delete node

}

}

void insert(Course course)

Node\* insertNode(Node\* node, Course course)

Course search(string courseID)

Node\* removeNode(Node\* node, string courseID)

void inOrder(Node\* node)

void printAll()

}

Main Function

Function Main()

courses = ReadCourseFile("course\_data.csv")

If courses is not empty:

For each course in courses:

Insert course into bst

DisplayCourseInfo(bst)

Main()

Txt Parser Function

Function ReadCourseFile(fileName)

Open file with name fileName for reading

Create an empty list named courses

For each line in file:

Split the line by commas into tokens

If number of tokens < 2:

Print "Invalid line format"

Continue

Assign courseID = tokens[0]

Assign courseName = tokens[1]

Assign preList = tokens[2:] (if any)

Create a new Course object with courseID, courseName, preList

Add the Course object to the courses list

Insert Node Function

Function insertNode(Node\* node, Course course)

If node is None

Return new Node(course)

If course.courseID < node.course.courseID

node.left = insertNode(node.left, course)

Else

node.right = insertNode(node.right, course)

Return node

Search Function

Function search(string courseID)

Create a Node\* current and set it to root

While current is not None

If current.course.courseID == courseID

Return current.course

Else If courseID < current.course.courseID

current = current.left

Else

current = current.right

Return an empty Course object

Remove Node Function

Function removeNode(Node\* node, string courseID)

If node is None

Return node

If courseID < node.course.courseID

node.left = removeNode(node.left, courseID)

Else If courseID > node.course.courseID

node.right = removeNode(node.right, courseID)

Else

If node.left is None

Node\* temp = node.right

delete node

Return temp

Else If node.right is None

Node\* temp = node.left

delete node

Return temp

Node\* temp = findMin(node.right)

node.course = temp.course

node.right = removeNode(node.right, temp.course.courseID)

Return node

In Order Traversal Function

Function inOrder(Node\* node)

If node is not None

inOrder(node.left)

Print "CourseID: " + node.course.courseID + ", CourseName: " + node.course.courseName

If node.course.preList is not empty

Print "Prerequisites: " + ", ".join(node.course.preList)

inOrder(node.right)

Print

Function printAll()

Call inOrder with root