Shayne Greene

Project One

08/14/2024

bool validateFile(string filename){

open the file….1

**IF** a valid file/able to open….1

**WHILE** not at the end of file….n

read next line….n

**IF** number of parameters >= 2 ….n

Check if last parameter exists in the file as a course….n

**IF** the prerequisite exists as a course….n

add the line to the courses vector….n

**ELSE**

Throw an invalid file format error ….n

**ENDIF**

**ELSE**

Throw an invalid file format error….n

**ENDIF**

**ENDWHILE**

**ELSE**

Throw unable to open file ….1

**ENDIF**

Return **TRUE**….1

}

Struct Course{

String courseNumber

String name

Vector<string> prerequisites

Course(){}

}

Void loadCourses(string inputFilePath, vector<Course>& courses){

Check if a valid file using “validateFile” method….n

**IF** a valid file….1

**WHILE** not at the end of file….n

Read the next file line….n

Create a new course object….n

Course courseNumber = first element ….n

Course name = second element….n

**WHILE** not at the end of the line….n

Add next element to prerequisite vector….n

**ENDWHILE**

Add course object to the Courses vector….n

**ENDWHILE**

**ELSE**

Throw invalid file format….1

**ENDIF**

}

Void searchCourse(vector<Course> courses, string courseNumber){

**For** the size of the vector

**IF** the course number matches

Print course elements

**For** prerequisite size

Print prerequisites

}

bool validateFile(string filename){

open the file

**IF** a valid file/able to open

**WHILE** not at the end of file

read next line

**IF** number of parameters >= 2

Check if last parameter exists in the file as a course

**IF** the prerequisite exists as a course

add the line to the courses vector

**ELSE**

Throw an invalid file format error

**ENDIF**

**ELSE**

Throw an invalid file format error

**ENDIF**

**ENDWHILE**

**ELSE**

Throw unable to open file

**ENDIF**

Return **TRUE**

}

Void loadCourses(string inputFilePath, HashTable<Course>& courses){

Check if a valid file using “validateFile” method

**IF** a valid file

**WHILE** not at the end of file

Read the next file line

Create a new course object

Course courseNumber = first element

Course name = second element

**WHILE** not at the end of the line

Add next element to prerequisite vector

**ENDWHILE**

Add course object to the Courses hash table

**ENDWHILE**

**ELSE**

Throw invalid file format

**ENDIF**

}

void searchCourse(HashTable<Course> courses, String courseNumber) {

Generate a key using the Hash function.

Create a currentnode based on generated key

**IF** the currentNode is not a nullptr and the currentNodes’ key doesn’t equal UINT\_MAX

**IF** the currentNodes courseNumber = the passed in courseNumber

Return the value of the currentNodes coursenumber

**ELSE**

**WHILE**

**IF** currentNodes coursenumber = passed in courseNumber

Return the value of the currentNodes coursenumber

**ELSE**

Set the currentNode to the currentNodes next

**ENDIF**

**ENDWHILE**

**ENDIF**

**ELSE**

Return an empty course object

**ENDIF**

}

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

Add a new node as the root

**WHILE** new node is not a nullptr

**IF** the current node holds the courseNumber

Return the courseNumber

**ENDIF**

**IF** the courseNumber is smaller than current node

Current node = current nodes left

**ELSE**

Current node = current nodes right

**ENDIF**

**ENDWHILE**

Return an empty bid

}

BinarySearchTree::InOrder(Node\* node){

**IF** the node is not a nullptr

Call InOrder with the nodes left until the leaf is reached

Print that nodes bid information until the leaf is reached

Call InOrder with the nodes right

**ENDIF**

}

<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

Menu

<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

Main(args){

Create a vector to hold Course objects called **courses**

Print “Option 1: Load Data”

Print “Option 2: Print Courses”

Print "Option 3: Print a Course”

Print "Option 9: Exit”

Print “Make a selection”

SWITCH (SELECTION)

Case 1:

**IF** validateFile(**dataFilename**) is true

loadCourses(**dataFilename**, **courses**)

**ENDIF**

Break

Case 2:

Sort the vector of objects by course ID

Print the courses

Break

Case 3:

Prompt for a course ID

Search for that course ID

**IF** the course ID is found

Display the values of the course struct

**ELSE**

Print that course was not found

**ENDIF**

Break

Case 9:

Clean up / destroy data structure

Exit the program

break

SWEND

}

VECTOR

Void loadCourses(string inputFilePath, vector<Course>& courses){

Check if a valid file using “validateFile” method….n

**IF** a valid file….1

**WHILE** not at the end of file….n^2

Read the next file line….n

Create a new course object….n

Course courseNumber = first element ….n

Course name = second element….n

**WHILE** not at the end of the line….n

Add next element to prerequisite vector….n

**ENDWHILE**

Add course object to the Courses vector….n

**ENDWHILE**

**ELSE**

Throw invalid file format….1

**ENDIF**

}

HASHTABLE

Void loadCourses(string inputFilePath, HashTable<Course>& courses){

Check if a valid file using “validateFile” method….1

**IF** a valid file….1

**WHILE** not at the end of file….n^2

Read the next file line….n

Create a new course object….n

Course courseNumber = first element ….n

Course name = second element….n

**WHILE** not at the end of the line….n

Add next element to prerequisite vector….n

**ENDWHILE**

Add course object to the Courses hash table….n

**ENDWHILE**

**ELSE**

Throw invalid file format….1

**ENDIF**

}

BST

BinarySearchTree::addNode(Node\* node, Bid bid){

**IF** the bidId is smaller than the root node…1

**IF** the nodes left is null …1

add new node…1

**ELSE**

Recusive call to addNode…n

**ENDIF**

**ELSE**

IF the nodes right is null…1

Add new node…1

**ELSE**

Recusive call to addNode…n

**ENDIF**

**ENDIF**

}

# Evaluation:

**Vector**:

Pros: faster loads

Easier to code / readability

Cons: Slower to add or remove elements

Slow sorting

**HashTable**

Pros: Highly efficient search

Cons: Hard to avoid collisions for large data sets

Difficult to retrieve in specific order

**BST**

Pros: Maintains the order of the data (great for fast retrieval)

Saves memory by growing with the data as needed

Cons: If not implemented correctly can become unbalanced, which makes searching take longer

Based on the computation time between all three data structures, a BST would be the best fit.