**CS 300 Pseudocode Document**

Asa Brown

SNHU | CS-300

Project One

26 Sept 2024

**Milestone One: Vector Data Structure**

Step 1 Read File and Check for Courses:

OPEN File at File Path to Read

If not found “File DNE”

Determine valid courses and store in a vector // Keep a list of format errors

For Each line in the file,

If line is empty, skip. Else continue

// Fields are comma separated

If fields < 2,

throw error else continue

CourseNumber = field[0]; CourseName = field[1]; prerequisites = field[2:];

Create course object (courseName, courseNumber, prerequisites)

ADD to courses vector

CLOSE file

Step 2 Course Objects and Functions:

Course Objects:

Variables – String courseNumber, courseName;

List prerequisites

Functions:

Constructor Course(courseName, courseNumber, prerequisites){

This.courseName = courseName;

This.courseNumber = courseNumber;

This.prerequisites = prerequisites;}

Void getCourseInfo()

Print CourseName

Print CourseNumber

If prerequites.size > 0

Print prerequisites

Step 3, Search for course info:

CourseSearch(courseNumber){

For Each CourseNumber in Courses

If course.courseNumber == courseNumber

Print Course Name}

**Milestone Two: Hash Tables**

Step 1 Read File and Check for Courses:

OPEN File at File Path to Read

If not found “File DNE”

Determine valid courses and store them in a vector

// Keep a list of format errors

For Each line in the file,

If line is empty, skip. Else continue

// Fields are comma-separated

If fields < 2,

throw error else continue

courseId = field[0]; courseName = field[1]; prerequisites = field[2:];

Create course struct{string courseId, string courseName, vector<string> prerequisites, Course\* next = nullptr;}

ADD to courses vector

CLOSE file

Step 2 Classes & Functions:

Class HashTable{

Constructors & Destructor

HashTable(), HashTable(unsigned int tableSize), ~HashTable()

Variables:

const unsigned int tableSize;

vector<Course> courses;

}

Unsigned Int Hash(Course course) – Get a hashed key for the hash table

// Will need to separate the courseId into parts

Declare two strings: string letters, numbers;

For each character in courseId

If char is equal to A through Z or a through z

Letters +=;

Else if char is equal to 0 through 9

Numbers +=; // Numbers will be used for hashed key

Throw exception if letters > 4 or numbers > 3

Return numbers % tableSize ;

Void Insert(Course course, int hashedKey) – Insert a course into the hash table

If courses[hashedKey] is empty

Courses[hashedkey] = course

Else

Course\* currentCourse = &courses[hashedKey]

Use while loop (current != nullptr) to iterate through the courses index to insert

Void Remove(Course course) – Remove course from hash table

If course is equal to the head at the hashedKey,

empty course node

Else

Iterate through until currentCourse->next.courseId is equal to courseId

Store next course in temp, set currentCourse->next to ->next->next;

Delete temp;

Void PrintAllCourseInfo(Course course) – Returns courseId, courseName, prerequisites

Nested loop

For loop – iterate while I = 0 and less than tableSize, increment by 1.

While loop – iterate while current->next != nullptr

Print out: courseId, CourseName

If prerequisites.size() > 0

Print prerequisites; // May require another loop

**Milestone Three: Tree Data Structure**

Step 1 Read File and Check for Courses:

OPEN File at File Path to Read

If not found “File DNE”

Determine valid courses and store them in a vector

// Keep a list of format errors

For Each line in the file,

If line is empty, skip. Else continue

// Fields are comma-separated

If fields < 2,

throw error else continue

courseId = field[0]; courseName = field[1]; prerequisites = field[2:];

struct course: courseId, courseName, <vector>Prerequisites

CLOSE file

Step 2 Classes, Functions, Variables:

Class: BinarySearchTree

Constructors & Destructors: BinarySearchTree(), ~BinarySearchTree()

Functions: insert(course), insertNode(currentNode, course), printCourseInfo(node)

Insert(course){

If root is nullptr

root = new Node(course)

Else, insertNode(root, course) } // End of function

insertNode(currentNode, course){

If course.courseId < currentNode->course.courseId{

If currentNode->left is nullptr

currentNode->left = new Node(course)

Else

insertNode(currentNode->left, course) }

Otherwise{

If currentNode->right is nullptr

currentNode->right = new Node(course)

Else

insertNode(currentNode->right, course) } // End of function

printCourseInfo(Node\* node){

If node != nullptr

printCourseInfo(node->left)

Print out CourseID

Print out CourseName

If node->course.prerequisites > 0

Print out Prerequisites

Else

Print “No prerequisites”

printCourseInfor(node->right) } // End of function

**Evaluation:**

Milestone One: Vector

* Reading the file
  + Cost per line: O(1) to check if the line is empty and create new course objects.
  + Total executions: n, one per course.
  + Total cost: O(n), Linear for reading and validating each line, file size may vary.
* Creating Course Objects
  + Cost per line: O(1) to store the course object into the vector
  + Total executions: n
  + Total cost: O(n)

Advantages:

* The size of vectors is dynamic and can be changed.
* Easier to understand

Disadvantages:

* It is difficult to insert new values since already present values need to be shifted.
* Searching for specific values requires each element to be looked through, which could take a lot of time depending on the vector’s size.

Milestone Two: Hash Table

* Reading the file
  + Cost per line: O(1)
  + Total executions: n, one per course
  + Total cost: O(n) for reading and checking each line
* Creating Course Objects
  + Cost per line: O(1) for adding the course to the hash table
  + Total execution: n
  + Toal cost: O(n)

Advantages:

* Insertions, deletions, and searching for specific values take less time with large file sizes.
* Searching for specific values requires less time in cases where there are few collisions.

Disadvantages:

* In scenarios with many collisions, the program could take longer to insert or search for values.

Milestone Three: Binary Search Tree

* Reading the file
  + Cost per line: O(1) for reading and checking each line
  + Total executions: n, one for course
  + Total cost: O(n) for validating each line
* Creating Course Objects
  + Cost per line: O(n) where n is the depth of the tree
  + Total executions: n, one per course
  + Total cost: O(n) in the case where the tree is unbalanced, O(log n) in a balanced tree

Advantages:

* Like vectors, the tree’s size can grow and shrink dynamically to account for more courses.
* Traversal through the tree’s nodes to search for a course is also efficient.

Disadvantages:

* Trees could end up being unbalanced, which hinders performance.

**Recommendation:**

Given the provided course information, I recommend using the Hash Table to sort and manage the course information. While vectors and Hash Tables have similar efficiency in processing information from my analysis, the hash table provides more flexibility when more courses need to be added. Additionally, since there aren’t many courses, the number of collisions can be mitigated to make it easier to parse through.