# CS 300 Pseudocode Document

Taylor Jones

12/8/2024

CS 300 – Project 1

//Print in Alphanumeric Order + Search

* **Vector Implementation**

void printOrderedCourses(courses) {

**IF courses is empty**

**PRINT “No courses loaded. Please load course file and try again.”**

**ELSE**

**SET orderedCourses = mergeSort(courses)**

**PRINT “Courses in Alphanumeric Order:”**

**FOR each course in courses**

**PRINT course.courseID + “: “ + course.courseTitle**

mergeSort(courses) {

**IF courses.length <= 1**

**RETURN courses**

**SET mid = LENGTH(courses) / 2**

**SET left = mergeSort(courses from first index to mid)**

**SET right = mergeSort(courses from mid to last index)**

**RETURN merge(left, right)**

merge(left, right)

**DECLARE mergeTemp as empty list**

**WHILE left and right are not empty:**

**IF left[0].courseID <= right[0].courseID**

**ADD left[0] to mergeTemp**

**REMOVE left[0] from left**

**ELSE**

**ADD right[0] to mergeTemp**

**REMOVE right[0] from right**

**mergeTemp = mergeTemp + left**

**mergeTemp = mergeTemp + right**

**RETURN mergeTemp**

}

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

**FOR all courses**

**IF the course is the same as courseID**

**PRINT out the course information**

**FOR each prerequisite of the course**

**PRINT the prerequisite course information**

}

* **Hash Table Implementation**

void printOrderedCourses(courses) {

SET node to address of element .at(i)

**PRINT “Courses in Alphanumeric Order:”**

FOR each node

WHILE node is not null AND the key is not the maximum

PRINT node.courseNumber + “: “ + node.courseTitle

node = next node

}

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

**unsigned int key = hash(courseNumber)**

**Node\* node = hashTable.table[key]**

**WHILE node != null**

**IF course number matches**

**RETURN node.course**

**MOVE to next node**

**PRINT “Not found”**

}

* **Binary Search Tree Implementation**

inOrderTraversal(Node\* node) {

**PRINT “Courses in Alphanumeric Order:”**  
IF node is null

RETURN

TRAVERSE left tree with inOrderTraversal(node->left)

PRINT node->course.courseNumber + “: “

PRINT node->course.courseTitle + endl;

TRAVERSE right tree with inOrderTraversal(node->right)

}

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

**SET current node to root**

**WHILE current course is NOT NULL loop tree**

**IF current course ID matches**

**RETURN node**

**ELSE IF course ID is < searched ID**

**SET node to left**

**ELSE IF course ID is > searched ID**

**SET node to right**

**RETURN node**

}

//Menu

displayMenu() {

PRINT “Menu:”

PRINT " 1. Load Bids"

PRINT " 2. Display Course List"

PRINT " 3. Display Course Details"

PRINT “ 9. Exit"

PRINT “Enter choice: “

GATHER input for choice

}

main() {

SET choice equal to 0

WHILE choice is not 9

CALL displayMenu()

SWITCH (choice)

CASE “1”:

CALL readFile() and lineParser() method

CASE “2”:

CALL printOrderedCourses()

CASE “3”:

CALL searchCourse()

CASE “9”:

PRINT “Exiting program.”  
 BREAK;

DEFAULT:

PRINT “Invalid choice. Try again.”

}

//Vector - Milestone 1

LOAD file parsing libraries and headers

DEFINE Struct Course {

**String courseID**

**String title**

**List of strings prerequisites**

}

void readFile(filePath) {

**DECLARE lines as empty list**

**TRY**

**OPEN file with filePath**

**WHILE file has lines to read**

**READ line from file with getline()**

**IF line is empty**

**CONTINUE**

**ADD line to lines**

**CLOSE file**

**CATCH error**

**PRINT file opening error**

**RETURN empty list**

**RETURN lines**

}

void lineParser(lines) {

**DECLARE courses as vector**

**FOR each line in lines**

**SPLIT line by ‘,’ into tokens**

**IF number of tokens < 2 then**

**PRINT error**

**CONTINUE**

**DECLARE course as new Course**

**SET course.courseID = token[0]**

**SET course.title = token[1]**

**SET course.prerequisites = empty list**

**FOR each token after index 1**

**ADD token to prerequisites**

**ADD course to courses vector**

**RETURN courses**

}

findCourse() {

PRINT “Enter courseID to search:”

**GATHER input**

**SET courseFound = false**

**FOR each course in Courses**

**IF input matches a courseID**

**PRINT course information**

**IF prerequisites is empty**

**PRINT “No prerequisites.”**

**ELSE**

**PRINT “Prereq: “ + course information**

**SET courseFound = true**

**BREAK**

**IF courseFound = false then**

**PRINT not found**

}

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

**for all courses**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

}

//Hash Table - Milestone 2

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

**unsigned int key = hash(courseNumber)**

**Node\* node = hashTable.table[key]**

**WHILE node != null**

**IF course number matches**

**RETURN node.course**

**MOVE to next node**

**PRINT “Not found”**

}

DEFINE Struct Course {

**Int courseNumber**

**String courseTitle**

**Vector<String> prereqs**

}

CLASS HashTable {

**-struct Node {**

**Course course**

**Unsigned int key**

**Node\* next**

**}**

**+HashTable()**

**+insert(Course course)**

**+printCourses()**

**+ Course searchCourse(String courseNumber)**

**-Vector<Node\*> table**

**-Unsigned int dataTableSize**

**-Unsigned int hash(String key)**

}

readFile() {

**GET CSV file path**

**IF no path submission**

**USE default file location**

**TRY**

**IF valid file input**

**CALL Parser() passing file path**

**CATCH**

**PRINT error**

}

lineParser(String csvPath, HashTable hashTable){

**OPEN file path using parsing libraries**

**DECLARE vector and call getHeader()**

**DISPLAY header row with “|” delimiter**

**TRY**

**FOR each line read until end of file**

**IF at least 2 parameters**

**CREATE course data structure**

**ASSIGN course.courseNumber**

**ASSIGN course.courseTitle**

**ASSIGN course.prereqs**

**INSERT course to end of hashTable**

**ELSE**

**PRINT “Wrong line format” and continue**

**CATCH**

**PRINT error**

}

printCourses() {

**FOR each node in HashTable**

**START at first course**

**WHILE current node != null AND current node.key != UINT\_MAX**

**PRINT course num, title, and preqs separated with “|”**

**MOVE to next node**

}

//Binary Search Tree – Milestone 3

readFile() {

**GET CSV file path from user**

**IF no path submission THEN**

**USE default file location**

**TRY**

**IF valid file input THEN**

**CALL Parser(file path, BinarySearchTree bst)**

**ELSE**

**PRINT “Invalid path.”**

**CATCH error**

**PRINT error**

}

DEFINE Struct Course {

**Int courseNumber**

**String courseTitle**

**Vector<String> prereqs**

}

DEFINE Struct Node {

Node()

**Node\* left = null**

**Node\* right = null**

**Node(Course aCourse)**

**course = aCourse**

}

CLASS BinarySearchTree {

**-Node root**

**-addNode(node, course)**

**-postOrder(node)**

**-inOrder(node)**

**-preOrder(node)**

**-Node removeNode(node, string course)**

**+BinarySearchTree()**

**+addNode()**

**+postOrder()**

**+inOrder(Course course)**

**+preOrder(string course)**

**+Course search(string course)**

}

lineParser(String csvPath, BinarySearchTree bst){

**OPEN file path using parsing libraries**

**DECLARE vector and call getHeader()**

**DISPLAY header row with “|” delimiter**

**TRY**

**FOR each line read until end of file**

**IF at least 2 parameters**

**CREATE course data structure**

**ASSIGN course.courseNumber**

**ASSIGN course.courseTitle**

**ASSIGN course.prereqs**

**INSERT course to node in BST**

**ELSE**

**PRINT “Wrong line format” and continue**

**CATCH**

**PRINT error**

}

Insert(Course courses){

**IF root is NULL**

**ASSIGN root to new node**

**ELSE**

**CALL addNode()**

**}**

printCourses() {

**GET course ID to search**

**IF root is NOT NULL**

**TRAVERSE tree left**

**IF found**

**OUTPUT node**

**IF not found**

**TRAVERSE tree right**

**IF found**

**OUTPUT node**

**IF not found**

**OUTPUT “Course ID not found.”**

}

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

**SET current node to root**

**WHILE current course is NOT NULL loop tree**

**IF current course ID matches**

**RETURN node**

**ELSE IF course ID is < searched ID**

**SET node to left**

**ELSE IF course ID is > searched ID**

**SET node to right**

**RETURN node**

}

**Vector:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **DECLARE courses as vector** | 1 | 1 | 1 |
| **FOR each line in lines** | 1 | n | n |
| **IF number of tokens < 2**  **PRINT error**  **CONTINUE** | 2 | n | n |
| **DECLARE course as new Course SET course.courseID = token[0] SET course.title = token[1] SET course.prerequisites = empty list** | 4 | n | 4n |
| **FOR each token after index 1 ADD token to prerequisites** | 1 | n | n |
| **ADD course to courses vector** | 1 | n | n |
| **RETURN courses** | 1 | 1 | 1 |
| **Total Cost** | | | 8n + 2 |
| **Runtime** | | | O(n) |

**Binary Search Tree:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **OPEN file path using parsing libraries** | 1 | 1 | 1 |
| **DECLARE vector** | 1 | 1 | 1 |
| **DISPLAY header row with “|” delimiter** | 1 | n | n |
| **FOR each line read until end of file** | 1 | n | n |
| **IF at least 2 parameters** | 1 | n | n |
| **CREATE course data structure ASSIGN course.courseNumber ASSIGN course.courseTitle ASSIGN course.prereqs** | 4 | n | 4n |
| **INSERT course to node in BST** | 1 | n | n |
| **PRINT “Wrong line format” and continue** | 1 | n | n |
| **CATCH PRINT error** | 1 | n | n |
| **Total Cost** | | | 10n + 2 |
| **Runtime** | | | O(n) |

**Hash Table:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **GET CSV file path** | 1 | n | n |
| **IF no path submission**  **USE default file location** | 1 | 1 | 1 |
| **IF valid file input CALL lineParser()** | 13 | 1 | 13n + 2 |
| lineParser() cost: | | | |
| **CREATE hash table** | 1 | 1 | 1 |
| **OPEN file path using parsing libraries** | 1 | n | n |
| **DECLARE vector** | 1 | 1 | 1 |
| **FOR each line read until end of file** | 1 | n | n |
| **IF at least 2 parameters** | 1 | n | n |
| **CREATE course data structure ASSIGN course.courseNumber ASSIGN course.courseTitle ASSIGN course.prereqs** | 4 | n | 4n |
| **INSERT course to end of hashTable** | 1 | n | n |
| **PRINT “Wrong line format” and continue** | 1 | n | n |
| **CATCH PRINT error** | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 14n + 3 |
| **Runtime** | | | O(n) |

**Advantages and Disadvantages**

As far as using a vector approach for this program, some advantages are that vectors are a very simple data structure to implement, we can achieve O(1) iteration times, and sorting with methods like merge sort can provide O(nlogn) sort times which offer a compromise on speed. The vector approach was also the fastest out of the three proposed methods based on the runtime and memory analysis with a total cost of 8n + 2 and total runtime of O(n). Some disadvantages to using a vector approach are that searching must be done linearly when a structure is not explicitly sorted which with O(n) time could take a very long time for larger data sets.

Major advantages for using a Hash Table are that hashing automatically organizes data for quick access, and with key-based access you can approach data retrieval in more intuitive ways when compared to a vector that uses sequential indices or a binary search tree that uses ordered relationships. Using a Hash Table also has the advantage of very fast (O(1)) searching, inserting, and deleting operations. Some disadvantages to using a hash table, particularly in this case, are that there is a lot of memory overhead, and due to the nature of hashing there is no inherent order of data.

Binary Search Trees comes with the advantage of easily sorting data through in-order traversal of the tree. BSTs allow for dynamic insertion and deletion while maintaining order. A disadvantage of using a BST is that when implemented improperly the trees can become unbalanced resulting in runtimes of O(n2). Based on the runtime and memory analysis, using a BST would be the costliest computationally.

**Recommendation**

Based on the analysis and advisor’s requirements I recommend choosing a Hash Table implementation. A Hash Table will provide great performance (O(1)) for searching, inserting, and deleting. Since the advisor will need to frequently search for data to retrieve information such as course details as well as routinely manage course information, I believe a hash table will offer the best end user experience. While hash tables lack inherent ordering, the compromise in slower sorting speeds is not a large factor in the decision since maintaining the order of the courses is not a requirement. Hash tables are a great choice when frequent lookups and updates are needed and are a data structure well-suited for housing course information.