Paulina Weaver

Professor Jamaliannasrabadi

CS-300

02/18/2024

Project 1

**Vector Pseudocode**

**File Input**

FUNCTION loadCourses(file\_path):

Define a vector to hold all the courses: vector<Course> courses;

TRY:

OPEN file at file\_path for reading

IF file not found:

PRINT "Error: File not found at path: " + file\_path

RETURN null

ENDIF

WHILE not end of file:

line = read a line from the file

data = split line using comma as the delimiter

IF line has less than two parameters:

PRINT "Error: Insufficient parameters in line: " + line

RETURN null

ELSE:

Create a data structure and add to the collection of courses: Course course

course.courseNumber = data[0]

course.courseTitle = data[1]

course.coursePrerequisites = empty list

IF length of data > 2:

FOR i = 2 to length of data:

prerequisite = data[i]

IF (!course\_exists(prerequisite)):

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

RETURN null

ENDIF

coursePrerequisites.push\_back(prerequisite)

ENDFOR

ENDIF

push this course to the end: courses.push\_back(course);

ENDIF

CATCH FileNotFound:

PRINT "Error: File not found at path: " + file\_path

RETURN null

END TRY

CLOSE the file

RETURN courses

END FUNCTION

FUNCTION course\_exists(courseNumber):

FOR each course in courses:

IF course.courseNumber == courseNumber:

RETURN true

RETURN false

END FUNCTION

**Course Object Pseudocode**

Create structure named Course to hold course information

String courseNumber

String courseTitle

String coursePrerequisites

ENDSTRUCT

Define a vector to hold all the courses: vector<Course> courses;

**Print Course Information Pseudocode**

FUNCTION printCourseInfo(courseNumber, courses):

foundCourse = null

FOR course in courses:

IF course.courseNumber == courseNumber:

foundCourse = course

BREAK

ENDIF

ENDFOR

IF foundCourse is not null:

PRINT("Course Number:", foundCourse.courseNumber)

PRINT("Title:", foundCourse.courseTitle)

IF length of foundCourse.coursePrerequisites > 0:

PRINT("Prerequisites:", ", ".join(foundCourse.coursePrerequisites))

ELSE:

print("No prerequisites.")

ENDIF

ELSE:

PRINT("Course not found.")

ENDIF

END function

**Sort and print courses in alphanumeric order**

FUNCTION PARTITION:

SET low = begin

SET high = end

BOOL done = false

SET pivot = middle element

WHILE not done THEN:

WHILE low is less than pivot THEN:

INCREMENT low

ENDWHILE

WHILE high is more than pivot THEN:

DECREMENT high

ENDWHILE

IF low is more or equal to high THEN:

SET done = true

ELSE THEN:

SWAP low and high

INCREMENT low

DECREMENT high

ENDIF

ENDWHILE

RETURN high

END function

FUNCTION QUICKSORT:

SET mid = 0

IF begin is more or equal to end THEN:

RETURN

ENDIF

SET midpoint to location of last element in low

CALL quicksort function to recursively sort low partition (begin to mid)

CALL quicksort function to recursively sort high partition (mid+1 to end)

END function

FUNCTION printSortedCourseList(courses)

CALL quicksort(courses, 0, courses.size() - 1)

FOR i = 0 to courses.size() - 1 :

PRINT("Course Number:", courses[i].courseNumber)

PRINT("Title:", courses[i].courseTitle)

IF length of courses[i].coursePrerequisites > 0 THEN:

PRINT("Prerequisites:", join(courses[i].coursePrerequisites, ", ")) ELSE:

PRINT("No prerequisites.")

ENDIF

ENDFOR

END function

**Menu**

FUNCTION menu():

DECLARE choice as integer

DECLARE string course\_number

WHILE choice is NOT 4:

PRINT "Menu:"

PRINT "1. Load Vector"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "4. Exit"

choice = READ user input as integer

SWITCH choice:

CASE "1":

courses = loadCourses(file\_path)

BREAK

CASE "2":

printSortedCourseList(courses)

BREAK

CASE "3":

PRINT "Enter course number: "

course\_number = READ course number from user

CALL printCourseInfo(course\_number, courses)

BREAK

CASE 4:

PRINT "Exiting program."

BREAK

DEFAULT:

PRINT "Invalid choice. Please try again."

BREAK

END SWITCH

ENDWHILE

PRINT "Goodbye”

END FUNCTION

**Hash Table Pseudocode**

**File Input**

FUNCTION loadCourses(file\_path: string, hashTable: HashTable\*)

TRY:

OPEN file at file\_path for reading

IF file not found:

PRINT "Error: File not found at path'"

RETURN null

ENDIF

WHILE not end of file:

line = read a line from the file

data = split line using comma as the delimiter

IF line has less than two parameters:

print "Error: Insufficient parameters in line " line

return null

ELSE

courseNumber = data[0]

courseTitle = data[1]

coursePrerequisites = empty list

IF length of data > 2:

FOR i = 2 to length of data:

prerequisite = data[i]

IF not course\_exists(courseNumber, hashTable):

PRINT "Error: Prerequisite does not exist for course '"

RETURN null

ENDIF

coursePrerequisites.push\_back(prerequisite)

ENDFOR

ENDIF

new\_course = create Course(courseNumber, courseTitle, coursePrerequisites)

hashTable->InsertCourses(courseNumber, new\_course)

ENDIF

ENDWHILE

CLOSE the file

RETURN hashTable

EXCEPTION FileNotFound:

PRINT "Error: File not found at path'"

RETURN null

END TRY

END FUNCTION

FUNCTION course\_exists(courseNumber: string, hashTable: HashTable\*)

RETURN hashTable->Contains(courseNumber)

END FUNCTION

**Course Object Pseudocode**

CREATE structure named Course to hold course information

String courseNumber

String courseTitle

List<String> coursePrerequisites

CREATE class HashTable {

DEFINE private members:

Struct Node{

Course course

Integer key

Node\* next

default constructor Node()

key = UINT\_MAX;

next = nullptr;

INITIALIZE with a course: Node(Course aCourse) : Node() {

course = aCourse;

INITIALIZE with a course and a key: Node(Course aCourse, unsigned int aKey) : Node(aCourse) {

key = aKey;

vector<Node> nodes;

unsigned int tableSize = DEFAULT\_SIZE;

unsigned int hash(int key);

DEFINE public members:

HashTable();

HashTable(unsigned int size);

void InsertCourses(Course course);

Course SearchAndPrint(string courseNumber);

void loadCourses(string file\_path);

END

Default constructor: HashTable::HashTable() {

RESIZE tableSize to initialize node structure

END

Constructor for specifying size of the table: HashTable::HashTable(unsigned int size) {

INVOKE local tableSize to size by using this->

RESIZE nodes size

END

Implement logic to calculate a hash value: unsigned int HashTable::hash(int key) {

RETURN key tableSize

END

FUNCTION to insert course: HashTable::InsertCourses(Course course) {

GENERATE key for given courseNumber using hash(course.courseNumber)

RETRIEVE node using key: Node\* node = nodes[key]

IF (no entry is found)

ASSIGN node to the key position

ELSE (if node is not used)

IF the key of the old node is uninitialized

ASSIGN old node key to UNIT\_MAX, set to key, set old node to course and old node next to null pointer

ELSE (Collision occurred, append the new node to the end of the linked list)

WHILE oldNode.next is not null:

oldNode = oldNode.next

ENDWHILE

ADD a new node to the end of the linked list

ENDIF

ENDIF

END

**Print Course Information Pseudocode**

FUNCTION Course HashTable::SearchAndPrint(string courseNumber) {

Course course

CREATE the key for the given courseNumber

IF key out of vector bounds

PRINT (“Course not found”)

RETURN

ENDIF

Node\* node = table[key];

// RETRIEVE node using key

WHILE (node does not equal nullptr)

IF the current node has a valid key and courseNumber matches

PRINT ("Course Number: " << node->course.courseNumber)

PRINT ("Course Title: " << node->course.courseTitle)

IF prerequisites not empty

PRINT ("Prerequisites: ")

FOR each prerequisite

PRINT (prerequisite << ", ")

ELSE:

PRINT("No prerequisites.")

RETURN

ENDIF

MOVE to the next node in the linked list

ENDWHILE

IF no entry found for the key

PRINT("Course not found”)

ENDIF

END

**Sort and print courses in alphanumeric order**

FUNCTION SortCoursesByNumber(hashTable: HashTable\*)

Initialize an empty array to store course objects: coursesAlphanumeric = []

FOR each key in hashTable->nodes

node = hashTable->nodes[key]

WHILE node is not null

Append course to the array: coursesAlphanumeric.push(node->course)

node = node->next

ENDWHILE

ENDFOR

SORT array by alphanumeric course number: coursesAlphanumeric.sort\_by(course => course.courseNumber)

FOR each course in coursesAlphanumeric

PRINT("Course Number: " + course.courseNumber)

PRINT("Course Title: " + course.courseTitle)

IF course.coursePrerequisites is not empty

PRINT("Prerequisites: " + course.coursePrerequisites.join(", "))

RETURN

ELSE

PRINT("No prerequisites.")

ENDIF

ENDFOR

END

**Menu**

FUNCTION menu():

DECLARE choice as integer

DECLARE string course\_number

WHILE choice is NOT 4:

PRINT "Menu:"

PRINT "1. Load Hash Table"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "4. Exit"

choice = READ user input as integer

SWITCH choice:

CASE "1":

loadCourses(file\_path, courseTable)

BREAK

CASE "2":

SortCoursesByNumber(hashTable)

BREAK

CASE "3":

PRINT "Enter course number: "

course\_number = READ course number from user

hashTable->SearchAndPrint(course\_number)

BREAK

CASE 4:

PRINT "Exiting program."

BREAK

DEFAULT:

PRINT "Invalid choice. Please try again."

BREAK

END SWITCH

ENDWHILE

PRINT "Goodbye”

END FUNCTION

**Tree Pseudocode**

**File Input**

FUNCTION loadCourses(string file\_path, BinarySearchTree\* bst):

TRY:

OPEN file at file\_path for reading

IF file not found:

PRINT "Error: File not found at path: " + file\_path

RETURN null

ENDIF

WHILE not end of file:

line = read a line from the file

data = split line using comma as the delimiter

IF line has less than two parameters:

PRINT "Error: Insufficient parameters in line: " + line

RETURN null

ELSE:

courseNumber = data[0]

courseTitle = data[1]

coursePrerequisites = empty list

IF length of data > 2:

FOR i = 2 to length of data:

prerequisite = data[i]

IF (!course\_exists(prerequisite, bst)):

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

RETURN null

ENDIF

coursePrerequisites.push\_back(prerequisite)

ENDFOR

ENDIF

CREATE course object: Course course(courseNumber, courseTitle, coursePrerequisites);

INSERT into binary search tree: bst->Insert(course);

ENDIF

ENDWHILE

CLOSE the file

CATCH FileNotFound:

PRINT "Error: File not found at path: " + file\_path

RETURN null

END TRY

END FUNCTION

FUNCTION BinarySearchTree::courseExists (string courseNumber, BinarySearchTree\* bst):

RETURN bst->Search(courseNumber) != nullptr

END FUNCTION

FUNCTION to insert a course BinarySearchTree::Insert(Course course) {

IF root is equal to null pointer THEN

SET root equal to new node course

ELSE

CALL addNode function and pass root and courseNumber

ENDIF

END FUNCTION

FUNCTION to add a course to a node recursively BinarySearchTree::addNode(Node\* node, Course course):

IF node is not null pointer and is node’s course is larger THEN:

IF left node equals nullptr THEN:

SET left node to new node

RETURN

ELSE:

RECURSE down the left node

ENDIF

ELSE IF node is not null pointer and is node’s course is smaller THEN:

IF right node equals nullptr THEN:

SET right node to new node

RETURN

ELSE :

RECURSE down the right node

ENDIF

ENDIF

END

**Create course objects:**

STRUCT Course to hold course information:

String courseNumber

String courseTitle

List<String> coursePrerequisites

END STRUCT

STRUCT Node:

Course course

Node\* left

Node\* right

Default constructor Node():

left <- nullptr

right <- nullptr

Constructor with a course Node(Course aCourse):

CALL Node() // Call the default constructor

course <- aCourse

END STRUCT

CLASS BinarySearchTree:

DEFINE private members:

Node\* root

FUNCTION addNode(Node\* node, Course course)

DEFINE public members:

FUNCTION BinarySearchTree():

SET root equal to nullptr

FUNCTION Insert(Course course)

FUNCTION Course Search(string courseNumber)

bool course\_exists(string courseNumber) to check if a course exists in the BST

FUNCTION SearchAndPrintCourse(string courseNumber)

END CLASS

**Print out course information and prerequisites:**

Search for a course: Course BinarySearchTree::Search(string courseNumber):

SET current node equal to root

WHILE current node is not null pointer THEN:

IF current node's courseNumber is equal to search courseNumber THEN:

RETURN current node's course

ELSE IF search courseNumber is smaller than current node's courseNumber THEN

TRAVERSE down the left node

ELSE

TRAVERSE down the right node

ENDIF

ENDWHILE

PRINT "Course with number " + courseNumber + " not found."

RETURN course

END

FUNCTION BinarySearchTree::searchAndPrintCourse (string courseNumber):

Call the Search function to find the course with the specified number: Course foundCourse = Search(courseNumber)

IF foundCourse is not null THEN:

PRINT "Course Number: " << foundCourse.courseNumber

PRINT "Course Title: " << foundCourse.courseTitle

IF foundCourse.coursePrerequisites is not empty THEN:

PRINT "Prerequisites: "

FOR each prerequisite IN foundCourse.coursePrerequisites:

PRINT prerequisite + ", "

ENDFOR

ELSE:

PRINT "No prerequisites."

ENDIF

ENDIF

END FUNCTION

**Sort and print courses in alphanumeric order**

Traverse the tree in order:

void BinarySearchTree::InOrder() {

CALL inOrder fuction and pass root

END

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

IF node is not equal to null pointer THEN

CALL inOrder(node->left) to traverse left subtree

PRINT "Course Number: " << node->course.courseNumber

PRINT "Course Title: " << node->course.courseTitle

IF node->course.coursePrerequisites is not empty THEN:

PRINT "Prerequisites: "

FOR each prerequisite IN node->course.coursePrerequisites:

PRINT prerequisite + ", "

ENDFOR

ELSE:

PRINT "No prerequisites."

ENDIF

CALL inOrder(node->right) to traverse right subtree

ENDIF

END

**Menu**

FUNCTION menu():

DECLARE integer choice = 0

DECLARE string course\_number

WHILE choice is NOT 4:

PRINT "Menu:"

PRINT "1. Load Binary Search Tree "

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "4. Exit"

choice = READ user input as integer

SWITCH choice:

CASE "1":

loadCourses(file\_path, courseBst)

BREAK

CASE "2":

CALL courseBst->InOrder()

BREAK

CASE "3":

PRINT "Enter course number: "

course\_number = READ course number from user

CALL courseBst-> searchAndPrintCourse (course\_number)

BREAK

CASE 4:

PRINT "Exiting program."

BREAK

DEFAULT:

PRINT "Invalid choice. Please try again."

BREAK

END SWITCH

ENDWHILE

PRINT "Goodbye”

END FUNCTION

**Run-time and Memory**

**Vector**

**Worst-case Runtime** **Complexity**

1. Opening and reading the file: O(n)

Opening the file and checking for errors: O(1)

Reading each line from the file: O(n), where n is the number of lines in the file

2. Parsing each line: O(n \* m)

Splitting the line using a comma delimiter: O(1)

Checking if the line has sufficient parameters: O(1)

Creating a data structure for each course: O(1)

Parsing course number and title: O(1)

Parsing course prerequisites: O(m), where m is the number of prerequisites for each course

3. Checking if a course exists: O(n^2)

Looping through the vector of courses: O(n), where n is the number of courses

4. Adding courses to the vector: O(n)

Pushing a course to the end of the vector: O(1)

5. Closing the file: O(1)

Overall worst-case running time for reading the file and creating course objects is approximately O(n^2) due to the nested loop for checking if a course exists.

**Memory Complexity**

Worst-case: O(n)

1. Memory usage is directly proportional to the number of elements stored in the vector.

2. There's no additional overhead beyond what's required to store the actual data elements.

**Hash Table**

**Worst-case Runtime** **Complexity**

1. Opening and reading the file: O(n)

Opening the file and checking for errors: O(1)

Reading each line from the file: O(n), where n is the number of lines in the file

2. Parsing each line: O(n)

Splitting the line using a comma delimiter: O(1)

Checking if the line has sufficient parameters: O(1)

Creating a new course object: O(1)

Checking if a course exists in the hash table: O(1) average case, O(n) worst-case if there are hash collisions

3. Inserting courses into the hash table: O(n)

Generating the hash key: O(1)

Retrieving the node from the hash table: O(1) average case, O(n) worst-case due to collisions

Inserting the course into the hash table: O(1) average case, O(n) worst-case if there are hash collisions

4. Closing the file: O(1)

Overall worst-case running time for loading courses from the file and creating course objects using a hash table approach is approximately O(n^2) due to the worst-case scenario of hash table collisions. However, the actual runtime might vary depending on factors such as the efficiency of the hash function and the load factor of the hash table.

**Memory Complexity**

Worst-case: O(n + m)

1. Memory usage depends on both the number of elements stored in the hash table (n) and the number of buckets (m).
2. Additional memory may be required for pointers, bucket nodes, and other internal structures.
3. In cases of hash table resizing or rehashing, additional memory may be temporarily allocated.

**Binary Tree**

**Worst-case Runtime** **Complexity**

1. Opening and reading the file: O(n)

Opening the file and checking for errors: O(1)

Reading each line from the file: O(n), where n is the number of lines in the file

2. Parsing each line: O(n)

Splitting the line using a comma delimiter: O(1)

Checking if the line has sufficient parameters: O(1)

Creating a new course object: O(1)

Checking if a course exists in the BST: O(log n) average case, O(n) worst-case if the tree is unbalanced

3. Inserting a course into the BST: O(log n) average case, O(n) worst-case if the tree is unbalanced

4. Closing the file: O(1)

Overall worst-case running time for loading courses from the file and creating course objects using a binary search tree approach is approximately O(n^2) due to the worst-case scenario of the tree becoming highly unbalanced. However, in practice, the average-case performance is often better, closer to O(n log n), assuming the tree remains balanced.

**Memory Complexity**

Worst-case: O(n)

1. Memory usage depends solely on the number of elements stored in the binary search tree.
2. Additional memory may be required for tree nodes and pointers.
3. In cases where the tree becomes unbalanced, memory usage may increase due to the deeper tree structure required to store the same number of elements.

The vector data structure is efficient for scenarios where sequential access to elements is common. Its contiguous memory allocation facilitates fast iteration, making it ideal for tasks like linear scans or processing elements in order. Vectors are straightforward to implement and understand, making them popular for basic use cases. However, their efficiency decreases for operations like arbitrary insertions or deletions, as these actions may require resizing the vector and shifting elements, resulting in O(n) worst-case time complexity. Additionally, searching for elements within a vector necessitates a linear scan, which may be inefficient for large datasets.

Hash tables provide quick search and insertion operations, typically with constant-time complexity on average (O(1)). This makes them suitable for scenarios requiring rapid data access, such as dictionary implementations or caching. Hash tables are versatile in handling key-value pairs, offering a convenient way to store and retrieve associated data. However, they are vulnerable to hash collisions, where multiple keys map to the same index, leading to performance degradation and potentially O(n) worst-case time complexity for search and insertion. Additionally, hash tables may incur additional memory overhead due to bucket nodes, pointers, and resizing operations.

Binary search trees maintain elements in sorted order, making them suitable for applications requiring ordered data. They offer efficient search operations with O(log n) time complexity on average, leveraging the binary search algorithm. However, inefficient insertion or deletion operations may lead to unbalanced trees, resulting in degraded performance with O(n) worst-case time complexity. Additionally, binary search trees may have additional memory overhead due to the need for tree nodes and pointers, especially in complex implementations.

For the given scenario where the program needs to store and sort courses based on alphanumeric course numbers, as well as search for specific courses by their course number, a binary search tree is the most suitable data structure. Binary search trees naturally maintain elements in sorted order based on their keys (course numbers) and support efficient searching with an average-case time complexity of O(log n). This makes them suitable for quickly finding courses by their course number and printing them in sorted order. While binary search trees may have additional memory overhead compared to vectors or hash tables, this overhead is generally manageable for moderate-sized datasets.