Randy Phimmasen

CS 300

Professor Oluwayomi Adamo

August 3rd, 2024

Project One

**//Vector - Milestone 1**

**Opening Files, Reading Data, and Checking for Format Errors:**

void loadCourses(String filename) {

Vector<Course> courses

open file with filename

if file is not open

print "Error: Cannot open file"

exit

while not end of file

read line from file

split line by commas into tokens

if tokens.size() < 2

print "Error: Invalid file format"

continue

String courseNumber = tokens[0]

String courseTitle = tokens[1]

Vector<String> prerequisites

for i from 2 to tokens.size()

prerequisites.add(tokens[i])

Course course = new Course(courseNumber, courseTitle, prerequisites)

courses.add(course)

close file

return courses

}

**Creating Course Objects and Storing Them in Vector Data Structure:**

struct Course {

String courseNumber

String courseTitle

Vector<String> prerequisites

}

Vector<Course> loadCourses(String filename) {

Vector<Course> courses

open file with filename

if file is not open

print "Error: Cannot open file"

exit

while not end of file

read line from file

split line by commas into tokens

if tokens.size() < 2

print "Error: Invalid file format"

continue

String courseNumber = tokens[0]

String courseTitle = tokens[1]

Vector<String> prerequisites

for i from 2 to tokens.size()

prerequisites.add(tokens[i])

Course course = new Course(courseNumber, courseTitle, prerequisites)

courses.add(course)

close file

return courses

}

**Searching Data Structure for Specific Course and Printing Course Info and Prerequisites:**

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

for Course course in courses

if course.courseNumber equals courseNumber

print "Course Number: " + course.courseNumber

print "Title: " + course.title

if course.prerequisites is empty

print "Prerequisites: None"

else

print "Prerequisites: "

for prerequisite in course.prerequisites

print prerequisite

return

print "Error: Course not found"

}

**Menu:**

void menu() {

while true

print "1. Load Courses"

print "2. Print All Courses"

print "3. Print Course Info"

print "9. Exit"

int choice

input choice

if choice equals 1

courses = loadCourses(filename)

else if choice equals 2

printAllCourses(courses)

else if choice equals 3

print "Enter Course Number:"

String courseNumber

input courseNumber

searchCourse(courses, courseNumber)

else if choice equals 9

exit

else

print "Invalid choice, please try again."

}

**Printing All Courses in Alphanumeric Order:**

void printAllCourses(Vector<Course> courses) {

sort courses by courseNumber

for Course course in courses

print "Course Number: " + course.courseNumber + ", Title: " + course.title

}

**//Hash Table- Milestone 2**

**Opening Files, Reading Data, and Checking for Format Errors:**

HashTable loadCourses(String filePath) {

HashTable hashTable

Set<String> allCourses

open file with filePath

if file cannot be opened

print "Error: Cannot open file"

return hashTable

while not end of file

read line from file

split line by commas into tokens

if tokens.size() < 2

print "Error: Incorrect file format in line: " + line

return hashTable

allCourses.add(tokens[0])

set file to beginning

while not end of file

read line from file

split line by commas into tokens

String courseNumber = tokens[0]

String courseTitle = tokens[1]

Vector<String> prerequisites

for i from 2 to tokens.size()

if tokens[i] not in allCourses

print "Error: Prerequisite " + tokens[i] + " not found for course " + courseNumber

return hashTable

prerequisites.add(tokens[i])

Course course = new Course(courseNumber, courseTitle, prerequisites)

hashTable.insert(courseNumber, course)

close file

return hashTable

}

**Creating Course Objects and Storing Them in Hash Table:**

struct Course {

String courseNumber

String courseTitle

Vector<String> prerequisites

}

Course createCourse(String courseNumber, String courseTitle, Vector<String> prerequisites) {

Course course

course.courseNumber = courseNumber

course.courseTitle = courseTitle

course.prerequisites = prerequisites

return course

}

void insertCourseIntoHashTable(HashTable hashTable, Course course) {

hashTable.insert(course.courseNumber, course)

}

**Searching Data Structure for Specific Course and Printing Course Info and Prerequisites:**

void searchCourse(HashTable hashTable, String courseNumber) {

Course course = hashTable.get(courseNumber)

if course is not null

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

if course.prerequisites is not empty

print "Prerequisites: " + join(course.prerequisites, ' ')

else

print "Prerequisites: None"

else

print "Error: Course not found"

}

**Menu:**

void menu() {

while true

print "1. Load Courses"

print "2. Print All Courses"

print "3. Print Course Info"

print "9. Exit"

int choice

input choice

if choice equals 1

hashTable = loadCourses(filePath)

else if choice equals 2

printAllCourses(hashTable)

else if choice equals 3

print "Enter Course Number:"

String courseNumber

input courseNumber

searchCourse(hashTable, courseNumber)

else if choice equals 9

exit

else

print "Invalid choice, please try again."

}

**Printing All Courses in Alphanumeric Order:**

void printAllCourses(HashTable hashTable) {

sort hashTable by courseNumber

for key in hashTable

Course course = hashTable.get(key)

print "Course Number: " + course.courseNumber + ", Title: " + course.title

}

**//Binary Search Tree- Milestone 3**

**Opening Files, Reading Data, and Checking for Format Errors:**

Vector<Course> loadCourses(String filePath) {

Vector<Course> listOfCourses

open file 'courses.txt' for reading

if file not found

print "Error: File not found."

return listOfCourses

while not end of file

read line from file

split line into tokens using comma as delimiter

if tokens.size() < 2

print "Error: Incorrect format in line: " + line

continue

String courseNumber = tokens[0]

String courseTitle = tokens[1]

Vector<String> prerequisites

for i from 2 to tokens.size()

prerequisites.add(tokens[i])

Course course = new Course(courseNumber, courseTitle, prerequisites)

listOfCourses.add(course)

close file

return listOfCourses

}

**Creating Course Objects and Storing in the BST:**

BinarySearchTree bst

for each Course course in listOfCourses

bst.insert(course)

**Searching Data Structure for Specific Course and Printing Course Info and Prerequisites:**

void printCourseInfo(Course course) {

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

if course.prerequisites is empty

print "Prerequisites: None"

else

print "Prerequisites: " + join(course.prerequisites, ' ')

}

void searchCourse(BinarySearchTree bst, String courseNumber) {

Node node = bst.search(courseNumber)

if node is not null

printCourseInfo(node.course)

else

print "Error: Course not found"

}

**Menu:**

void menu() {

while true

print "1. Load Courses"

print "2. Print All Courses"

print "3. Print Course Info"

print "9. Exit"

int choice

input choice

if choice equals 1

bst = new BinarySearchTree()

Vector<Course> listOfCourses = loadCourses(filePath)

for each Course course in listOfCourses

bst.insert(course)

else if choice equals 2

printAllCourses(bst)

else if choice equals 3

print "Enter Course Number:"

String courseNumber

input courseNumber

searchCourse(bst, courseNumber)

else if choice equals 9

exit

else

print "Invalid choice, please try again."

}

**Printing All Courses in Alphanumeric Order:**

void printAllCourses(BinarySearchTree bst) {

bst.inOrderTraversal()

}

**Runtime Analysis**

**Vector**

Using a Vector is straightforward and easy to implement. The time complexity for searching and inserting elements in a Vector is O(n), where n is the number of courses. This linear complexity means that in the worst case, we may need to scan through all elements to find or insert a course. While Vectors are simple and efficient for small datasets, they can become slow as the dataset grows due to this linear search time.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| for Course course in courses | 1 | n | n |
| if course.courseNumber equals courseNumber | 1 | n | n |
| print "Course Number" | 1 | 1 | 1 |
| print "Title" | 1 | 1 | 1 |
| if course.prerequisites is empty | 1 | 1 | 1 |
| print "Prerequisites: None" | 1 | 1 (if no prerequisites) | 1 |
| print "Prerequisites: " | 1 | 1 (if there are prerequisites) | 1 |
| for prerequisite in course.prerequisites | 1 | m (number of prerequisites) | m |
| print prerequisite | 1 | m (number of prerequisites) | m |
| print "Error: Course not found" | 1 | 1 (if course not found) | 1 |
| Total Cost | - | - | 3n + 3 + 2m |

**Total Cost:** 3n + 3 + 2m  
**Runtime:** O(n)

**Hash Table**

Hash Tables are efficient for average-case performance. Both insertion and search operations typically have a constant time complexity of O(1). This efficiency is due to the hash function, which quickly maps course numbers to their slots in the table. However, in the rare worst case scenario, hash collisions can cause the time complexity to degrade to O(n), like a Vector. The primary drawback of Hash Tables is their potentially high memory usage and the need for a good hash function to minimize collisions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| get course from hashTable | 1 | 1 | 1 |
| if course is not null | 1 | 1 | 1 |
| print "Course Number: " | 1 | 1 | 1 |
| print "Course Title: " | 1 | 1 | 1 |
| if course.prerequisites is not empty | 1 | 1 | 1 |
| print "Prerequisites: " | 1 | m | m |
| print "Prerequisites: None" | 1 | 1 | 1 |
| print "Error: Course not found" | 1 | 1 | 1 |
| Total Cost | - | - | 6 + m |

**Total Cost:** 6 + m (m is the number of prerequisites)  
**Runtime:** O(1) (average case), O(n) (worst case due to collisions)

**Binary Search Tree**

Binary Search Trees offer a balanced approach, with a time complexity of O(log n) for search and insertion operations, assuming the tree remains balanced. This logarithmic complexity is much better than the linear complexity of Vectors, especially for large datasets. However, if the tree becomes unbalanced, the time complexity can degrade to O(n). To prevent this, self-balancing trees like AVL trees or Red-Black trees can be used to maintain logarithmic performance.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| search the BST | 1 | log(n) | log(n) |
| if node is not null | 1 | 1 | 1 |
| printCourseInfo(node.course) | 1 | 1 | 1 |
| print "Course Number" | 1 | 1 | 1 |
| print "Course Title" | 1 | 1 | 1 |
| if course.prerequisites is empty | 1 | 1 | 1 |
| print "Prerequisites: None" | 1 | 1 (if no prerequisites) | 1 |
| print "Prerequisites: " | 1 | 1 (if there are prerequisites) | 1 |
| print "Prerequisites: " | 1 | m (number of prerequisites) | m |
| print "Error: Course not found" | 1 | 1 (if course not found) | 1 |
| Total Cost | - | - | log(n) + 6 + m |

**Total Cost:** log(n) + 6 + m  
**Runtime:** O(log n) (average case, balanced tree), O(n) (worst case, unbalanced tree)

**Recommendation**

After analyzing the three data structures, the Hash Table stands out for its average-case performance. It provides the fastest access times for both insertion and search operations, making it the best choice for scenarios where quick access to data is essential. However, if memory usage is a concern or if the dataset frequently changes and could cause collisions, the Binary Search Tree is a reliable alternative because of its balanced performance. Vectors, while easy to implement, are less suitable for large datasets due to their linear time complexity.

Therefore, it is recommended to use a Hash Table. It offers the best average case performance and scalability, making it well-suited for handling many courses efficiently.