Quinlin MacKenzie

Chao Ling

SNHU-CS300

4/17/24

**Project One**

Open file, read data, parse each line, check for formatting errors;

**Vector Course Object: Applicable for all data structures**

Define Course class {

String course number

String course title

Vector<String> prerequisites

Course(number, title) {

constructor for course object initialization

}

...

}

**Vector File Read and Sort**

Open File

if File is not null

Vector<Course> courses

while not end of File

String line = read Line

if line is not empty

String[] tokens = split line into tokens

if length of tokens >= 2

String course Number = tokens[0]

String course Title = tokens[1]

Course course = new Course(course Number, course Title)

if length of tokens > 2

Store string of prerequisites from tokens[2 to end]

for each prerequisite in prerequisites

course.addPrerequisite(prerequisite)

courses.add(course)

else

print "Error: not enough parameters"

else

print "Error: empty line"

close File

**Vector: Print Course Information**

printCourseInfo(Vector<Course> courses)

if courses is empty

print "No courses found"

else

for each course in courses

print "Course Number: " + course.getNumber()

print "Course Title: " + course.getTitle()

if course has prerequisites

for each prerequisite in course.getPrerequisites()

print prerequisite

else

print "No prerequisites"

print "" // Blank line for clarity between courses

**Hashtable File Read and Sort**

Open File

if File is not null

Hashtable<String, Course> coursesHashtable // Using course number as key

while not end of File

String line = read Line

if line is not empty

String[] tokens = split line into tokens

if length of tokens >= 2

String courseNumber = tokens[0]

String courseTitle = tokens[1]

Course course = new Course(courseNumber, courseTitle)

if length of tokens > 2

Store string of prerequisites from tokens[2 to end]

for each prerequisite in prerequisites

course.addPrerequisite(prerequisite)

coursesHashtable.put(courseNumber, course) // Store course in hashtable

else

print "Error: not enough parameters"

else

print "Error: empty line"

close File

**Hashtable: Print Course Information**

printCourseInfo(Hashtable<String, Course> coursesHashtable)

if coursesHashtable is empty

print "No courses found"

else

for each key in coursesHashtable.keys()

Course course = coursesHashtable.get(key)

print "Course Number: " + course.getNumber()

print "Course Title: " + course.getTitle()

if course has prerequisites

for each prerequisite in course.getPrerequisites()

print prerequisite

else

print "No prerequisites"

print "" // Blank line for clarity between courses

**Define Tree Node class for BST nodes**

class Tree Node {

Course course

Tree Node left Child

Tree Node right Child

Tree Node(Course course) {

this.course = course

this.leftChild = null

this.rightChild = null

}

}

**Define BinarySearchTree class for managing courses**

class BinarySearchTree

Tree Node root

BinarySearchTree()

this.root = null

void insert(Course new course)

node = new node(new course)

if root is null

new course becomes the root

else

Current node = root

while not inserted

if new course < current course

if left child is null

Insert as left child

else

Move to left child

else if new course > current course

if right child is null

Insert as right child

else

Move to right child

else

Course with the same course number already exists

**BST File Read and Sort**

Open File

if File is not null

BinarySearchTree<String, Course> coursesBST // Using course number as key

while not end of File

String line = read Line

if line is not empty

String[] tokens = split line into tokens

if length of tokens >= 2

String courseNumber = tokens[0]

String courseTitle = tokens[1]

Course course = new Course(courseNumber, courseTitle)

if length of tokens > 2

Store string of prerequisites from tokens[2 to end]

for each prerequisite in prerequisites

course.addPrerequisite(prerequisite)

Insert course into BST

else

print "Error: not enough parameters"

else

print "Error: empty line"

close File

**BST: Print Course Information**

printCourseInfo(BinarySearchTree<String, Course> coursesBST)

if coursesBST is empty

print "No courses found"

else

coursesBST.inOrderTraversal(printCourseDetails)

Function printCourseDetails(key, value)

Course course = value

print "Course Number: " + course.getNumber()

print "Course Title: " + course.getTitle()

if course has prerequisites

for each prerequisite in course.getPrerequisites()

print prerequisite

else

print "No prerequisites"

print blank line for clarity between courses

**In Order Traversal Functions Needed for full list printing**

void inOrderTraversal(Function printFunction) {

inOrderTraversalHelper(root, printFunction)

}

void inOrderTraversalHelper(TreeNode node, Function printFunction) {

if node is not null {

inOrderTraversalHelper(node.leftChild, printFunction)

printFunction(node.course)

inOrderTraversalHelper(node.rightChild, printFunction)

}

}

}

// Since the BST is sorted as it is stored, a simple in order traversal should print all the course information in alphanumeric order based on the course number. Assuming we are using a similar menu loop for managing courses as we did with managing bid information; specifically loading the data, printing the whole list, and printing specific courses with their prerequisites, we can have the following main function...

**Define main menu loop function**

boolean running = true

BinarySearchTree coursesBST = new BinarySearchTree()

while running {

print "Menu Options:"

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "4. Exit"

print "Enter your choice:"

int choice = read user input

switch choice {

case 1:

loadCourses(coursesBST, "data.txt") // Assuming "data.txt" is the file containing course information

break

case 2:

printCourseList(coursesBST)

break

case 3:

print "Enter course number:"

String courseNumber = read user input

printCourse(coursesBST, courseNumber)

break

case 4:

running = false

break

default:

print "Invalid choice. Please try again."

}

}

**Run-time and Memory Analysis:**

Vector Data Structure:

Reading the file and creating course objects:

Cost per line: 1

Number of lines executed: n (assuming n courses)

Big O: O(n)

Advantages: Simple to implement, maintains insertion order.

Disadvantages: Linear search for course information can be slow for large datasets.

Hashtable Data Structure:

Reading the file and creating course objects:

Cost per line: 1 (assuming constant time for hash table operations)

Number of lines executed: n (assuming n courses)

Big O: O(n)

Advantages: Fast retrieval (assuming good hash function), efficient for large datasets.

Disadvantages: Potential collisions.

Binary Search Tree (BST) Data Structure:

Reading the file and creating course objects:

Cost per line: Log(n) (assuming balanced tree)

Number of lines executed: n (assuming n courses)

Big O: O(n \* log(n)) (due to log(n) insertion time for each course)

Advantages: Fast search and retrieval (logarithmic time complexity), maintains sorted order.

Disadvantages: Requires balancing for optimal performance, memory overhead for tree structure.

**Conclusion:**  
Based on the Big O analysis and the specific requirements of fast retrieval and efficient handling of large datasets, the Hashtable Data Structure emerges as the most suitable choice for managing course information in this context. Hashtable operations offer an average-case time complexity of O(1) for search, insert, and delete operations, ensuring fast retrieval times (assuming a good hash function). While there may be some overhead in hash table maintenance and potential collisions, these drawbacks are outweighed by the Hashtable's ability to handle large datasets efficiently and scale well. Overall, the Hashtable provides the best balance of performance and scalability for the given requirements.