Sanefa Amzad

CS-300-11105-M01 DSA: Analysis and Design 2024 C-3

Instructor: Chao Ling

10 June 2024

**6-2 Submit Project One**

**Milestone One: Vector Data Structure Pseudocode**

class Course

string number

string title

list<string> prerequisites

end class

function load\_courses\_from\_file(filename, courses)

open file filename for reading

if file is not open then

print "Error: Unable to open the file"

return

end if

while not end of file

read line from file

if is\_valid\_format(line) then

course\_data = split line by ','

course = create\_course(course\_data)

append course to courses

else

print "Error: Invalid file format"

end if

end while

close file

end function

function is\_valid\_format(line)

// Check if the line has at least two elements separated by ','

course\_data = split line by ','

return length(course\_data) >= 2

end function

function create\_course(course\_data)

course\_number = course\_data[0]

course\_title = course\_data[1]

prerequisites = empty list

for i from 2 to length(course\_data) - 1

append course\_data[i] to prerequisites

end for

declare new\_course as new Course

new\_course.number = course\_number

new\_course.title = course\_title

new\_course.prerequisites = prerequisites

return new\_course

end function

function print\_course\_info\_and\_prerequisites(course)

print "Course Number:", course.number

print "Course Title:", course.title

print "Prerequisites:"

if length(course.prerequisites) == 0 then

print "None"

else

for each prerequisite in course.prerequisites

print prerequisite

end for

end if

end function

function print\_alphabetically\_ordered\_courses(courses)

sort courses by course.number

for each course in courses

call print\_course\_info\_and\_prerequisites(course)

end for

end function

function main()

declare courses

declare user\_input

repeat

print "Menu:"

print "1. Load file data"

print "2. Print alphabetically ordered list of courses"

print "3. Print course information and prerequisites"

print "9. Exit"

read user\_input

switch user\_input

case 1:

call load\_courses\_from\_file("course\_data.txt", courses)

case 2:

call print\_alphabetically\_ordered\_courses(courses)

case 3:

print "Enter course number:"

read course\_number

course = find\_course\_by\_number(course\_number, courses)

if course is not found then

print "Course not found"

else

call print\_course\_info\_and\_prerequisites(course)

end if

case 9:

print "Exiting program"

default:

print "Invalid option"

end switch

until user\_input = 9

end function

call main()

**Runtime Analysis**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Operation** | **Time Complexity (worst case)** | |  |  | | --- | --- | |  | **Cost Per Line** |  |  | | --- | |  | | |  | | --- | | **# Times Executes** |  |  | | --- | |  | | |  | | --- | | **Total Cost** |  |  | | --- | |  | | **Explanation** |
|  | | | | | |
| Parsing each line | O(m) | 1 | n | n \* m | m is the number of elements in a line |
| Creating Course object | O(m) | 1 | n \* m | n \* m |  |
| Appending Course to vector | O(1) | 1 | n | n |  |
| |  | | --- | | Printing course info/prereqs | | |  | | --- | | Printing course info/prereqs | | |  | | --- | | Printing course info/prereqs | | |  | | --- | | Printing course info/prereqs | | |  | | --- | | Printing course info/prereqs | |  |
| |  | | --- | | Sorting courses | | |  | | --- | | Sorting courses | | |  | | --- | | Sorting courses | | |  | | --- | | Sorting courses | | |  | | --- | | Sorting courses | | |  | | --- | | Sorting courses | |

**Advantages and Disadvantages:**

**Advantages:**

* Simple implementation.
* Efficient for small datasets.
* Easy to understand and maintain.

**Disadvantages:**

* Linear search for course lookup.
* Sorting the entire list for alphabetical order can be inefficient for large datasets.
* Memory overhead due to storing all data in memory.

**Recommendation:**

The vector data structure is a good option for small- to medium-sized datasets when ease of use and simplicity are important factors. Hash tables and binary search trees are examples of alternate data structures that would be more suitable for larger datasets or situations where efficient searching and sorting are needed.

**Milestone Two: Hash Table Data Structure Pseudocode**

STRUCT Course

STRING number

STRING title

LIST<String> prerequisites

END STRUCT

STRUCT HashTable

ARRAY OF LIST<Course> table

INTEGER size

END STRUCT

FUNCTION createHashTable(size)

DECLARE hashTable AS new HashTable

hashTable.size = size

FOR i FROM 0 TO size - 1

hashTable.table[i] = EMPTY LIST

END FOR

RETURN hashTable

END FUNCTION

FUNCTION hash(courseNumber, size)

// Hash function to map course number to an index in the table

RETURN courseNumber MOD size

END FUNCTION

FUNCTION loadCoursesFromFile(filename, hashTable)

OPEN file filename FOR reading

IF file IS NOT OPEN THEN

PRINT "Error: Unable to open the file"

RETURN

END IF

WHILE NOT end of file

READ line FROM file

IF isValidFormat(line) THEN

courseData = SPLIT line BY ','

course = createCourse(courseData)

index = hash(course.number, hashTable.size)

APPEND course TO hashTable.table[index]

ELSE

PRINT "Error: Invalid file format"

END IF

END WHILE

CLOSE file

END FUNCTION

FUNCTION printCourseInfoAndPrerequisites(course)

PRINT "Course Number:", course.number

PRINT "Course Title:", course.title

PRINT "Prerequisites:"

IF LENGTH(course.prerequisites) == 0 THEN

PRINT "None"

ELSE

FOR EACH prerequisite IN course.prerequisites

PRINT prerequisite

END FOR

END IF

END FUNCTION

FUNCTION printAlphabeticallyOrderedCourses(hashTable)

FOR i FROM 0 TO hashTable.size - 1

FOR EACH course IN hashTable.table[i]

CALL printCourseInfoAndPrerequisites(course)

END FOR

END FOR

END FUNCTION

FUNCTION main()

DECLARE hashTable

DECLARE userInput

REPEAT

PRINT "Menu:"

PRINT "1. Load file data"

PRINT "2. Print list of courses"

PRINT "3. Print course information and prerequisites"

PRINT "9. Exit"

READ userInput

SWITCH userInput

CASE 1:

PRINT "Enter hash table size:"

READ size

hashTable = createHashTable(size)

CALL loadCoursesFromFile("course\_data.txt", hashTable)

CASE 2:

CALL printAlphabeticallyOrderedCourses(hashTable)

CASE 3:

PRINT "Enter course number:"

READ courseNumber

index = hash(courseNumber, hashTable.size)

course = findCourseByNumber(courseNumber, hashTable.table[index])

IF course IS NOT FOUND THEN

PRINT "Course not found"

ELSE

CALL printCourseInfoAndPrerequisites(course)

END IF

CASE 9:

PRINT "Exiting program"

DEFAULT:

PRINT "Invalid option"

END SWITCH

UNTIL userInput = 9

END FUNCTION

CALL main()

**Runtime Analysis**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Operation** | **Time Complexity (worst case)** | |  |  | | --- | --- | |  | **Cost Per Line** |  |  |  | | --- | --- | |  |  | | |  | | --- | | **# Times Executes** |  |  | | --- | |  | | |  | | --- | | **Total Cost** |  |  | | --- | |  | |  | | **Explanation** |
|  | | | | | |
| Opening file | O(1) | 1 | 1 | 1 | Assuming constant time file opening |
| Reading each line | O(n) | 1 | n | n | n is the number of lines in the file |
| Parsing each line | O(m) | 1 | n | n \* m | m is the number of elements in a line |
| Creating Course object | O(m) | 1 | n \* m | n \* m |  |
| Appending Course to hash table | O(1) | 1 | n | n |  |
| Printing course info/prereqs | O(m) | 1 | n | n \* m |  |

**Advantages and Disadvantages:**

**Advantages:**

* Efficient for large datasets.
* Constant time lookup for each course (on average).
* No need to sort the data.

**Disadvantages:**

* Possibility of collisions affecting performance.
* Requires a good hash function for even distribution of data.
* Memory overhead due to the hash table size.

**Recommendation:**

The hash table data structure is a good option for medium- to large-sized collections where effective searching and constant-time lookup are crucial. To achieve best speed, nevertheless, rigorous hash function construction and collision management are needed.

**Milestone Three: Binary Search Tree Data Structure Pseudocode**

STRUCT Course

STRING number

STRING title

LIST<String> prerequisites

END STRUCT

STRUCT Node

Course course

Node leftChild

Node rightChild

END STRUCT

CLASS BinarySearchTree

Node root

METHOD addCourse(course)

// Implementation to add a course to the binary search tree

root = insert(root, course)

END METHOD

METHOD insert(node, course)

IF node IS NULL THEN

DECLARE newNode AS new Node

newNode.course = course

RETURN newNode

END IF

IF course.number < node.course.number THEN

node.leftChild = insert(node.leftChild, course)

ELSE IF course.number > node.course.number THEN

node.rightChild = insert(node.rightChild, course)

END IF

RETURN node

END METHOD

METHOD printInOrder(node)

IF node IS NOT NULL THEN

CALL printInOrder(node.leftChild)

CALL printCourseInfoAndPrerequisites(node.course)

CALL printInOrder(node.rightChild)

END IF

END METHOD

END CLASS

FUNCTION loadCoursesFromFile(filename, tree)

OPEN file filename FOR reading

IF file IS NOT OPEN THEN

PRINT "Error: Unable to open the file"

RETURN

END IF

WHILE NOT end of file

READ line FROM file

IF isValidFormat(line) THEN

courseData = SPLIT line BY ','

course = createCourse(courseData)

CALL tree.addCourse(course)

ELSE

PRINT "Error: Invalid file format"

END IF

END WHILE

CLOSE file

END FUNCTION

FUNCTION printCourseInfoAndPrerequisites(course)

PRINT "Course Number:", course.number

PRINT "Course Title:", course.title

PRINT "Prerequisites:"

IF LENGTH(course.prerequisites) == 0 THEN

PRINT "None"

ELSE

FOR EACH prerequisite IN course.prerequisites

PRINT prerequisite

END FOR

END IF

END FUNCTION

FUNCTION main()

DECLARE tree AS new BinarySearchTree

DECLARE userInput

REPEAT

PRINT "Menu:"

PRINT "1. Load file data"

PRINT "2. Print list of courses"

PRINT "3. Print course information and prerequisites"

PRINT "9. Exit"

READ userInput

SWITCH userInput

CASE 1:

CALL loadCoursesFromFile("course\_data.txt", tree)

CASE 2:

CALL tree.printInOrder(tree.root)

CASE 3:

PRINT "Enter course number:"

READ courseNumber

course = findCourseByNumber(courseNumber, tree.root)

IF course IS NULL THEN

PRINT "Course not found"

ELSE

CALL printCourseInfoAndPrerequisites(course)

END IF

CASE 9:

PRINT "Exiting program"

DEFAULT:

PRINT "Invalid option"

END SWITCH

UNTIL userInput = 9

END FUNCTION

CALL main()

**Runtime Analysis**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Operation** | **Time Complexity (worst case)** | |  |  | | --- | --- | |  | **Cost Per Line** |  |  | | --- | |  | | |  | | --- | | **# Times Executes** |  |  | | --- | |  | | |  | | --- | | **Total Cost** | |  |  |  | | --- | |  | | **Explanation** |
|  | | | | | |
| Opening file | O(1) | 1 | 1 | 1 | Assuming constant time file opening |
| Reading each line | O(n) | 1 | n | n | n is the number of lines in the file |
| Parsing each line | O(m) | 1 | n | n \* m | m is the number of elements in a line |
| Creating Course object | O(m) | 1 | n \* m | n \* m |  |
| Inserting course into BST | O(log n) | log n | n | n \* log n | n is the number of courses already inserted into the tree |
| Printing course info/prereqs | O(n) | 1 | n | n |  |

**Advantages and Disadvantages**

**Advantages:**

* Efficient searching (logarithmic time complexity).
* Sorted order traversal makes it easy to print courses in alphabetical order.
* Balanced trees ensure optimal performance.

**Disadvantages:**

* Requires additional memory for storing left and right child pointers.
* Slower insertion and deletion compared to hash tables.
* Complexity in balancing the tree to maintain optimal performance.

**Recommendation:**

The binary search tree data structure is a good option for medium- to large-sized datasets where sorted traversal and effective searching are needed. The search operations are balanced and have a logarithmic time complexity. For some operations, including insertion and deletion, it might be less effective than hash tables and needs to be carefully balanced to ensure performance.

**Make a recommendation for which data structure you plan to use in your code**

Based on the Big O analysis results and the characteristics of the three data structures presented, the Hash Table data structure seems to be the best option when considering the specifications and features. For the purpose of quickly retrieving course information, it offers effective lookup procedures with constant time complexity. Hash tables provide great performance for handling huge datasets effectively, but they may require careful consideration of collision resolution mechanisms and hash function design. Thus, employing a hash table would be the suggested method for a project where scalability and performance are important considerations.