//SNHU

//Student name: THOMAS SEIBERT

//Mod 6: Project One

**---------------------------------------------------------------------------------------------------------------------**

1. **VECTOR (PSEUDOCODE)**

//loads data from file into a vector of Course structs

//each struct contains the course number, name, and a vector of pre-reqs

//after loading data, it validates that each pre-req listed for a course corresponds to a course in the file

//finally, it searches for a specific course and prints out the course information and pre-reqs

Struct Course {

String courseNumber

String courseName

Vector prerequisites

}

Vector Course courses

loadDataFromFile(String filename) {

Open file with filename

IF file is not open {

Print "Error opening file"

Exit

}

WHILE not end of file {

Read line from file

Split line into parameters using space as delimiter

IF parameters size is less than 2 {

Print "Invalid file format"

Exit

}

Create new Course object

Set courseNumber of Course object to first parameter

Set courseName of Course object to second parameter

FOR i from 3 to parameters size {

Add parameter[i] to prerequisites of Course object

}

Add Course object to courses vector

}

Close file

}

validateCourses() {

FOR each Course in courses {

FOR each prerequisite in Course prerequisites {

IF not exists course with courseNumber equals to prerequisite in courses {

Print "Invalid prerequisite: " + prerequisite

Exit

}

}

}

}

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

}

main() {

LoadDataFromFile("text file")

ValidateCourses()

searchCourse("course id")

}

1. **HASH TABLE (PSEUDOCODE)**

// Function to load data into hash table

loadDataIntoHashTable(HashTable courses, String fileName):

Open file with fileName

IF file is not open:

Print "Error opening file"

Exit

End IF

WHILE not end of file:

Read line from file

Split line into parts using space as delimiter

IF parts size is less than 2:

Print "Invalid line format"

Continue to next line

End IF

Create a new Course object

Set Course number to parts[0]

Set Course title to parts[1]

IF parts size is more than 2:

FOR i from 2 to parts size - 1:

Add parts[i] to Course prerequisites

End FOR loop

End IF

Add Course to courses hash table with Course number as key

End WHILE loop

Close file

// Function to search and print course info

searchCourse(HashTable Course courses, String courseNumber):

IF courseNumber exists in courses:

Course course = courses[courseNumber]

Print "Course Number: " + course.number

Print "Course Title: " + course.title

IF course prerequisites not empty:

Print "Prerequisites:"

For each prerequisite in course prerequisites:

Print prerequisite

End For

ELSE:

Print "No prerequisites"

End IF

ELSE:

PRINT "Course not found"

End IF

//Function to validate file format

validateFileFormat(HashTable courses, String fileName):

Open file with fileName

IF file is not open:

PRINT "Error opening file"

Exit

End IF

WHILE not end of file:

Read line from file

Split line into parts using space as delimiter

IF parts size is less than 2:

PRINT "Invalid line format"

Continue to next line

End IF

FOR i from 2 to parts size - 1:

IF parts[i] does not exist in courses:

PRINT "Invalid prerequisite: " + parts[i]

End IF

End FOR loop

End WHILE loop

Close file

1. **BINARY TREE SEARCH (PSEUDOCODE)**

//reads file lines

//parses lines into course object

//inserts course object into tree data structure

//with searchCourse function to print information

1. Define Struct Course:

String courseNumber

String courseTitle

Vector prerequisites

2. Define Class Tree:

Course root

Tree left

Tree right

3. Define main method:

Declare Tree courses

Declare File courseFile = open("courseFile.txt")

WHILE not end of courseFile:

Declare String line = readLine(courseFile)

Declare Course newCourse = parseLine(line)

IF newCourse is not null:

courses = insertCourse(courses, newCourse)

End IF loop

End WHILE loop

close(courseFile)

4. Define parseLine(String line) method:

Declare Array tokens = split(line, ",")

IF length of tokens < 2:

print("Invalid line format")

return null

End IF

Declare Course course

course.courseNumber = tokens[0]

course.courseTitle = tokens[1]

FOR i from 2 to length of tokens:

add tokens[i] to course.prerequisites

End FOR loop

return course

5. Define insertCourse(Tree courses, Course newCourse) method:

IF courses is null:

return new Tree with newCourse as root

ELSE IF newCourse.courseNumber < courses.root.courseNumber:

courses.left = insertCourse(courses.left, newCourse)

ELSE IF newCourse.courseNumber > courses.root.courseNumber:

courses.right = insertCourse(courses.right, newCourse)

End IF loop

return courses

6. Define searchCourse(Tree<Course> courses, String courseNumber) method:

IF courses is null:

PRINT ("Course not found")

ELSE IF courseNumber == courses.root.courseNumber:

PRINT ("Course Number: " + courses.root.courseNumber)

PRINT ("Course Title: " + courses.root.courseTitle)

PRINT ("Prerequisites: " + courses.root.prerequisites)

ELSE IF courseNumber < courses.root.courseNumber:

searchCourse(courses.left, courseNumber)

ELSE IF courseNumber > courses.root.courseNumber:

searchCourse(courses.right, courseNumber)

End IF loop

1. **MENU (PSEUDOCODE)**

//this code assumes that certain functions already exist

//to read a file

//to parse file data into data structure

//and sort the courses

Initialize data\_structure as empty

Function main\_menu:

WHILE True:

Print "1. Load data"

Print "2. Print ordered list of courses"

Print "3. Print course details"

Print "9. Exit"

user\_input = Get user input

IF user\_input == 1:

data\_structure = load\_data()

ELSE IF user\_input == 2:

IF data\_structure is not empty:

print\_ordered\_courses(data\_structure)

ELSE:

PRINT "Please load the data first."

ELSE IF user\_input == 3:

IF data\_structure is not empty:

course\_name = Get user input for course name

print\_course\_details(data\_structure, course\_name)

ELSE:

PRINT "Please load the data first."

ELSE IF user\_input == 9:

Exit program

ELSE:

Print "Invalid option. Please try again."

Function load\_data:

file\_data = Read file

IF file\_data is not empty:

data\_structure = Parse file\_data into data structure

Return data\_structure

Function print\_ordered\_courses(data\_structure):

ordered\_courses = Sort courses in data\_structure alphanumerically

FOR each course in ordered\_courses:

Print course

Function print\_course\_details(data\_structure, course\_name):

IF course\_name exists in data\_structure:

PRINT "Course Title: ", course\_name

PRINT "Prerequisites: ", prerequisites of course\_name in data\_structure

ELSE:

PRINT "Course not found."

1. **ALPHANUMERIC SORT (PSEUDOCODE)**

//init empty list

//split lines with commas

//extract course numbers

//sort the list

//print the sorted list

Initialize an empty list called 'courses'

FOR each line in the course information:

Split the line by commas

Extract the course number (the first element of the split line)

Add the course number to the 'courses' list

Sort the 'courses' list in alphanumeric order

FOR each course in the 'courses' list:

PRINT the course

1. **EVALUATION**

*// Vector*

|  |  |  |
| --- | --- | --- |
| **Line of Code** | **Number of Executions** | **Cost** |
| Open file with filename | 1 | 1 |
| If file is not open | 1 | 1 |
| While not end of file | N+1 | N+1 |
| Read line from file | N | N |
| Split line into parameters | N | N |
| If parameters size < 2 | N | n |
| Create new Course object | N | N |
| Set courseNumber and courseName | 2n | 2n |
| For I from 3 to parameters size | N\*(m-2) | N\*(m-2) |
| Add course object to courses vector | N | N |
| Close file | 1 | 1 |
| For each Course in courses | N | N |
| For each prereq in Course prereqs | N\*m | N\*m |
| If not exists course with courseNumber equals to prereq in courses | N\*m | N\*m |
| searchCourse | 1 | n |

|  |  |
| --- | --- |
| **Function** | **Big-O value** |
| loadDataFromFile | O(n\*m) |
| validateCourses | O(n\*m) |
| searchCourse | O(n) |

// Vector: advantages and disadvantages

Advantages:

1. Vectors can change size dynamically, which may come in handy when any number of elements is not yet known.
2. Vectors provide random access to any element.
3. Vectors provide efficient methods to insert and delete elements from the end.

Disadvantages:

1. Vectors do not provide effective methods to insert and delete elements from the beginning or middle as they require shifting of elements.
2. Vectors need extra memory to account for future elements and thus can waste memory.

*// Hash Table*

|  |  |  |
| --- | --- | --- |
| **Line of Code** | **Number of Executions** | **Cost** |
| Open file | 1 | 1 |
| If file not open | 1 | 1 |
| While not end file | N+1 | N+1 |
| Read line from file | N | N |
| Split line into parts | N | n |
| If parts size is less than 2 | N | n |
| Create new Course object | N | N |
| Set course number and title | N | n |
| If parts size > 2 | N | n |
| For I from 2 to parts size – 1 | N\*(parts size – 2) | N\*(parts size – 2) |
| Add course to hash table | N | N |
| Close file | 1 | 1 |
| If courseNumber in courses | 1 | 1 |
| Course course = courses[courseNumber] | 1 | 1 |
| Print course details | 1 | 1 |
| If course prerequisites not empty | 1 | 1 |
| For each prerequisite in course prerequisites | Prerequisites size | Prerequisites size |
| Open file | 1 | 1 |
| If file not open | 1 | 1 |
| While not end file | N+1 | N+1 |
| Read line from file | N | N |
| Split in into parts | N | N |
| If parts size < 2 | N | n |
| For I from 2 to parts size – 1 | N\*(parts size – 2) | N\*(parts size – 2) |

|  |  |
| --- | --- |
| **Function** | **Big-O value** |
| loadDataIntoHashTable | O(n \* (parts size – 2)) |
| searchCourse | O(1) |
| validateFileFormat | O(1) |

// Hash Table: advantages and disadvantages

Advantages:

1. Hash Tables allow for O(1) average time complexity for searching, inserting, and deleting key value pairs. In other words, fast lookup and insertion.
2. Hash Tables store data in an array, which increases memory efficiency compared to trees, for example.
3. Hash Tables offer good cache locally. When accessing a hash table, a CPU can prefetch adjacent elements.

Disadvantages:

1. A poor hash function can lead to clustering and collisions.
2. When a hash table changes size, the whole table may need rehashing.
3. A good hash function is essential for a hash table’s performance. Without this, a hash table can be rendered useless altogether.

*// Binary Tree Search*

|  |  |  |
| --- | --- | --- |
| **Line of Code** | **Number of Executions** | **Cost** |
| Declare Tree courses | 1 | 1 |
| Declare File courseFile = open(“courseFile.txt”) | 1 | 1 |
| While not end courseFile | N+1 | N+1 |
| Declare string line = readline(coursefile) | N | N |
| Declare course newcourse = parsline(line) | N | N \* parseline cost |
| If newcourse is not null | N | n |
| Courses = insertcourse(courses, newcourse) | N | N \* insertcourse cost |
| End if | N | N |
| End while | N | n |
| Close(coursefile) | 1 | 1 |
| Declare array tokens = split(line “,” | 1 | 1 |
| If length of tokens < 2 | 1 | 1 |
| Print(“Invalid”) | 1 | 1 |
| Return null | 1 | 1 |
| Declare course course | 1 | 1 |
| Course.courseNumber = tokens[0] | 1 | 1 |
| Course.courseTitle = tokens[1] | 1 | 1 |
| For I from 2 to length of tokens: | N | n |
| Add tokens[i] to course.prerequisites | N | N |
| End for | N | N |
| Return course | 1 | 1 |
| If courses is null | 1 | 1 |
| Return new tree with newcourse as root | 1 | 1 |
| Courses.left = insertCourse(courses.left, newCourse) | N | N \* insertCourse cost |
| Courses.right = insertCourse(courses.right, newCourse) | N | N \* insertCourse cost |
| End if | 1 | 1 |
| Return courses | 1 | 1 |
| If courses is null: | 1 | 1 |
| Print(“Course not found”) | 1 | 1 |
| Else if courseNumber == courses.root.courseNumber | 1 | 1 |
| PRINT ("Course Number: " + courses.root.courseNumber) | 1 | 1 |
| PRINT ("Course Title: " + courses.root.courseTitle) | 1 | 1 |
| PRINT ("Prerequisites: " + courses.root.prerequisites) | 1 | 1 |
| ELSE IF courseNumber < courses.root.courseNumber: | 1 | 1 |
| searchCourse(courses.left, courseNumber) | N | N \* searchCoursecost |
| ELSE IF courseNumber > courses.root.courseNumber: | 1 | 1 |
| searchCourse(courses.right, courseNumber) | N | N \* searchCoursecost |
| End if | 1 | 1 |

**Big-O Value: O(n)**

// Binary Tree Search: advantages and disadvantages

Advantages:

1. BST's allow for efficient logarithmic based time searching, making them ideal for large datasets.
2. BST's maintain balance between left and right subtrees. This allows for efficient insert and delete methods.
3. BST's can be compactly represented compared to other search data structures, such as hash tables.

Disadvantages:

1. While this data structure does maintain balance, insert and delete methods may still be complex in cases where the tree is not balanced, which might lead to poor performance.
2. BST's cannot contain duplicate values by default.
3. BST's are optimized for exact match queries, which limits their query types.
4. **RECOMMENDATION (HASH TABLE)**

According to my research and understanding, the best data structure for this application would be a Hash Table, for the reasons I listed above in "advantages and disadvantages".

It provides fast access time for search, insert, and delete operations in the average case, regardless of the size of the data. This means if the university wanted to add courses to the list, they could do so without effecting the time taken to perform operations with the application.

With a hash table, each course can be associated with a unique key (course ID, for example), which isn't possible with vectors or BST's. Hash tables also do not require data to be stored in a pre-sorted manner, unlike BST's.