Liu, Kevin

14 June 2024

Professor Nathan Lebel

CS-300-11426-M01 DSA: Analysis and Design 2024

**Pseudocode**

**Vector Data Structure**

Load necessary text parsing libraries and headers

Define a struct to hold course data

struct Course

string courseID

string courseName

int preCount

string prelist

Course

courseID = "";

courseName = "";

preCount = 0;

preList = "";

Main Function

Create new List of the struct-type Course

Get file from user

Insert error handling if no input found

Validate list

if Valid

Get user input

Print result

else

Print error

Function to parse a CSV file and return a list of courses

Open file and parse data

Loop through rows

Extract course information

Add course to tempList

return tempList

Function to search for a course in the list

Loop through list

If courseId matches, assign course to tempCourse

Function to print course information recursively

Print courseID and courseName

Loop through prerequisites

Recursively call PrintCourse for each prerequisite

Function to validate the course list

Loop through courseList

If any course's prerequisites are invalid, set isValid to false and break

**Hash Table**

Function parseLine(line):

Parses each line into course number, title, and prerequisites.

Checks for the minimum required tokens (course number and title).

Function validatePrerequisites(courses):

Check that every prerequisite listed for each course exists as a course in the file.

Validates prerequisites to ensure all listed prerequisites exist.

For each course\_number in courses

For each prerequisite in courses.prerequisites

If prerequisite not in courses

Raise format error

Processes each line, creates Course objects, and stores them in a hash table.

Function processCourses:

Initialize an empty hash table

Create a new course object

Store course in hash table

Validate prerequisites

Prints the course information, including prerequisites, in a readable format.

**Binary Search Tree**

Open the file containing course data. If the file cannot be opened, print an error message and exit.

Create an empty dictionary to store course information

While not the end of file

Read each line of the file, splitting it into tokens.

Check if the line has at least two tokens (course number and title)

Create a course object with the parsed data and add it to a dictionary.

Validate that each prerequisite exists as a course in the file

If a prerequisite is missing

print an error message and remove the course from the dictionary.

Create Course Objects and Store in Binary Search Tree

Create a new binary search tree

Insert each course object from the dictionary into the binary search tree

Define a function to print course information using in-order traversal of the binary search tree

Print course information for each course, including course number, title, and prerequisites.

**Pseudocode for a menu**

Create a load data function that reads course data from a file and populates the three data structures. It adds parsed course details to the vector, hash table, and tree.

Create a sort courses function that sorts the courses in the vector alphanumerically by course number.

Create a print courses function that prints the course number and title for each course in the given list.

Create a print course details function that prints the title and prerequisites for a specific course using the hash table for quick access.

Create a function for the main menu allows the user to select actions. It handles loading data, printing the sorted list of courses, printing course details, and exiting the program.

Declare a loop that while True

Menu along with choices

Menu

1: Load file data

2: Print alphanumerically ordered list of courses

3 : Print course title and prerequisites

9 : Exit Program

**Pseudocode that will print out the list of the courses in the** **Computer Science program in alphanumeric order**

Create Traversal function to collect courses in sorted order

Create a sort courses function that sorts the courses alphanumerically

Use Traversal function to get sorted lists of courses

Print Computer Science program in alphanumeric order

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N)  *\*depends on if there are collisions* | O(log N) |
| **Search** | O(n) | O(1) – O(N)  *\*depends on if there are collisions* | O(log N) – O(N)  *\*depends on balance of the tree* |
| **Sort/Print** | O(N log N) *\*using quick sort* | O(N)  *\*assumes the table is created in order* | O(N)  *\*in order traversal* |

**Evaluations**

**Analysis**

Analyzing the vector, hash table, and tree data structures based on the advisor’s requirements reveals distinct advantages and disadvantages for each. Vectors are simple to implement and provide O(1) time complexity for accessing elements by index, making them ideal for scenarios requiring frequent access to data without many insertions or deletions. However, their performance degrades with costly O(n) insertions and deletions, particularly when modifying the dataset frequently.

Hash tables offer fast O(1) average-case time complexity for insertions, deletions, and lookups, making them suitable for scenarios requiring quick access by key. However, they do not maintain any order among elements and can experience degraded performance in the worst-case scenario due to hash collisions. Additionally, hash tables come with extra memory overhead for storing keys and managing collisions.

On the other hand, balanced binary search trees maintain elements in sorted order and provide balanced O(log n) performance for insertions, deletions, and lookups, which is beneficial for handling dynamic datasets. While they are more complex to implement and have slightly higher overhead for maintaining balance, their ability to efficiently handle ordered data and range queries makes them highly suitable for applications requiring ordered data access. Considering these factors, balanced binary search trees emerge as the most suitable choice for the advisor’s requirements, offering both efficient performance and the necessary functionality to manage and print an alphanumerically ordered list of courses.

**Recommendation**

Based on the advisor’s requirements, I recommend using a balanced binary search tree for the data structure. This recommendation stems from the need to print an alphanumerically ordered list of courses, which a balanced binary search tree naturally supports by maintaining sorted order. The tree also provides balanced O(log n) performance for insertions, deletions, and lookups, ensuring efficiency even as the dataset changes. Although implementing a balanced binary search tree is more complex than a vector or hash table, its ability to handle ordered data and range queries efficiently outweighs this complexity. Furthermore, while vectors offer simple implementation and fast access, they struggle with costly insertions and deletions, and hash tables, though excellent for fast lookups, do not maintain order. Therefore, the balanced binary search tree presents the best trade-off, offering both efficient performance and the required functionality, making it the most suitable choice for the advisor’s program.