Cody Faircloth

Michael Susalla

CS 300

February 11, 2023

Project One: Pseudocode and Evaluation

**Pseudocode:**

**Vector Data Type:**

//Menu

Void printMenu() {

Create vector empty vector named courses

String userCourseNum

Int userChoice = -1

While (userChoice does not equal 9)

Output “1) Load Courses endl; 2) Display Courses endl; 3) Find Course endl; 9) Exit

Assign userChoice to input

Switch (userChoice)

Case 1:

courses = storeData(loadCourses(fileName))

break

Case 2:

printSampleSchedule(courses)

break

Case 3:

Output “Enter course number:”

Assign userCourseNum to user input

printCourseInformation(courses, userCourseNum)

break

Case 9:

Output “Exiting program”

break

Default:

“Invalid input.”

}

//Alphanumeric sorting and printing

Void printSampleSchedule(Vector<Course> courses) {

Create vector from passed vector called sortedCourses

Course temp

//Sorting

For ( i = 1; i < size of sortedCourses vector; i++)

Int j = 1

While (j > 0 and sortedCourses[j].courseNumber < sortedCourses[j – 1].courseNumber)

temp = sortedCourses[j]

sortedCourses[j] = sortedCourses[j – 1]

sortedCourses[j – 1] = temp

--j

//Printing

For (i = 0; i < size of sorted Courses vector, i++)

Output sortedCourses[i]’s courseNumber, courseName

If (sortedCourses[i]’s prerequisite vector is not empty)

For (each element in sortedCourses[i]’s prerequisite vector)

Output prerequisite element

**Hash Table Data Type:**

// Menu

Void printMenu() {

Create empty hash table named courses

String userCourseNum

Int userChoice = -1

While (userChoice does not equal 9)

Output “1) Load Courses endl; 2) Display Courses endl; 3) Find Course endl; 9) Exit

Assign userChoice to input

Switch (userChoice)

Case 1:

courses = storeData(loadCourses(fileName))

break

Case 2:

courses->printSampleSchedule(root)

break

Case 3:

Output “Enter course number:”

Assign userCourseNum to user input

printCourseInformation(courses, userCourseNum)

break

Case 9:

Output “Exiting program”

break

Default:

“Invalid input.”

}

//Alphanumeric sorting and printing

Void printSampleSchedule() {

For (each node in courses hash table)

If (courses[i].key does not equal UINT\_MAX)

Output courses[i]’s courseNum, courseName

If (courses[i]’s prerequisite vector is not empty)

For (each element in prerequisite vector)

Output prerequisite element

**Binary Search Tree Data Type:**

//Menu

Void printMenu() {

Create empty BST named courses

String userCourseNum

Int userChoice = -1

While (userChoice does not equal 9)

Output “1) Load Courses endl; 2) Display Courses endl; 3) Find Course endl; 9) Exit

Assign userChoice to input

Switch (userChoice)

Case 1:

courses = storeData(loadCourses(fileName))

break

Case 2:

courses->printSampleSchedule(rot)

break

Case 3:

Output “Enter course number:”

Assign userCourseNum to user input

printCourseInformation(courses, userCourseNum)

break

Case 9:

Output “Exiting program”

break

Default:

“Invalid input.”

}

//Alphanumeric sorting and printing

Void printSampleSchedule(node) {

If (node is null)

Return

Recursively call printSampleSchedule(node’s left child node)

Output node’s course object information (courseNumber, courseName, and prerequisites if applicable)

Recursively call printSampleSchedule(node’s right child node)

**Evaluation:**

For all three data types I used the same function to read the data from the file and assign each line to an object. This function is storeData(). The analysis of this function is shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Create fileData vector | 1 | 1 | 1 |
| Create string variable currentLine | 1 | 1 | 1 |
| Open file using fileName | 1 | 1 | 1 |
| While fileName is open | 1 | 1 | 1 |
| For (Each Line) | 1 | n | n |
| Create new course object currentCourse | 1 | 1 | 1 |
| If(Line has two or more parameters) | 1 | n | n |
| Assign first parameter to currentCourse courseNumber | 1 | 1 | 1 |
| Assign Second parameter to currentCourse courseName | 1 | 1 | 1 |
| If(Line has three or more parameters) | 1 | n | n |
| If(Additional parameters are in fileData) | 1 | n | n |
| For(each additional parameter) | 1 | n | n |
| Add current parameter to currentCourse prerequisites | 1 | 1 | 1 |
| Add currentCourse to fileData | 1 | 1 | 1 |
| Else | 1 | n | n |
| Go to next line | 1 | 1 | 1 |
| Close file using fileName | 1 | 1 | 1 |
| Return fileData | 1 | 1 | 1 |
| Total Cost: | | 6n + 12 | |
| Runtime: | | O(n) | |

Since all the data structures use the same loading and object creation function the worst case runtime will be O(n) for all cases; however, other functions associated with these data structures vary by structure. Vector search, insertion, and deletion functions have an average and worst case time complexity of O(n) each with a worst case space complexity of O(n). Hash table search, insertion, and deletion functions have an average time complexity of O(1) and a worst case time complexity of O(n). Hash table worst case space complexity is O(n). Binary search tree search, insertion, and deletion functions have an average time complexity of O(log(n)) and a worst case of O(n). Binary search tree worst case space complexity is O(n). It is important to time and space complexity of these additional functions in order to understand the advantages and disadvantages of each data type for this scenario.

The vector data structure is great for small data sets. The advantage of a vector is the space complexity as well as the speed at which data can be entered into the data structure. The main disadvantage is the slowness of searching for an entry as well as having to sort the vector outside of the structure. For example, to print all entries of a vector in ascending order, the vector must first be sorted and then printed. Depending on the type of sort this may increase runtime drastically or increase space complexity. For this scenario, with the number of courses being unknown, the disadvantages significantly outweigh the advantages.

The hash table data structure does an initial sort automatically when entered into the table. The insertion runtime is slightly slower than a vector since an additional hash function is ran to determine the bucket to place the data into. The main strength of this data structure is the speed of search, insertion, and deletion. This significantly decreases time complexity. Space complexity is slightly larger than the vector as additional space is designated for buckets that may or may not contain data. For this scenario, the hash table is a better option than the vector; however, a better choice exists.

The binary search tree is the happy medium of the two data structures above. The binary search tree has lower time complexity for search, insertion, and deletion than the vector but a larger time complexity than the hash table. The tree makes up for this increased time complexity with a lower space complexity than a hash table. Additionally, the tree sorts data as entries are added allowing for quicker runtimes when printing all data.

In this scenario, the binary search tree will be the best data structure to use. The number of courses is unknown, therefore, we will assume there will be many. This automatically eliminates the vector as it is wildly inefficient for large data sets. We know that the courses will be sorted based on alphanumeric course number. This would be slightly more difficult to implement in a hash table format, this eliminates the hash table. We can further justify the tree by the average runtime for search, insertion, and deletion functions of O(log(n)). This data structure boasts good space complexity as well. Ultimately, based on the known requirements, the binary search tree is the best option.