Project One: ABC University Evaluation

Tatiana Case

June 11th, 2023

CS-300

**System Pseudocode Using Vector:**

FUNCTION fileRead {

OPEN file

IF file is open {

SET fileRead = OPEN file

IF fileRead is null

OUTPUT “ERROR”

EXIT

}

WHILE not at end of file {

READ file by parsing each line

CREATE vector courseInfo to hold course info

CREATE string courseData

GETLINE from file and save into courseData

WHILE courseData length > 0 {

IF courseData >= two parameters AND prerequisite(s) exists {

ADD courseData into vector courseInfo

ELSE {

OUTPUT “Error: insufficient course data”

}

}

}

ELSE {

OUTPUT “Error Opening File”

EXIT

}

}

courseSearch {

OPEN AND READ file

WHILE not at end of file {

IF courseNum and courseName == user input

PRINT course info

FOR each prereq of the course {

PRINT prereq info

}

ELSE {

OUTPUT “Course Not Found”

EXIT

}

}

}

WHILE userSelection NOT EQUAL to 9

COUT menu options

COUT “Select a menu option”

CIN userSelection

SWITCH userSelection

CASE 1:

SET Courses equal to loadFile

CASE 2:

CIN searchNum

CALL searchCourses passing in course number and courses vector

CASE 3:

CALL printAllCourses passing in courses vector

CASE 9:

COUT “Good Bye”

EXIT PROGRAM

RETURN 0

**Vector Evaluation:**

At worst, the normal runtime complexity is directly proportional to the number of elements in the set. This means conducting all the operations for the vector would be O(n). To insert a course from the file, the file must be parsed line by line. A nested while loop is used to perform the parsing process. This makes the parsing operation runtime is O(n2) because each line in the file and elements on the line are parsed. The searching and deletion operations are linear. So, the worst case for those processes is O(n) runtime. This is because each operation requires looping over all elements at least once to find and or delete the element. Printing requires O(n2) because of the nested for loop required for printing the prerequisites, if any. This loop is required because courses can contain any number of prerequisites. Each of the above functions do not require additional space to work because they utilize the already allocated memory for the vector. The vector data structure has a space complexity of O(n) because it is completely dependent on the input size.

**Run Time Analysis Chart for Vector:**

|  |  |  |  |
| --- | --- | --- | --- |
| **CODE** | **LINE COST** | **# OF EXECUTIONS** | **TOTAL COST** |
| FOR ALL COURSES | 1 | n | n |
| PRINTING COURSE INFORMATION | 1 | n | n |
| FOR EACH PREREQ | 1 | n | n\*n |
| PRINT PREREQ INFO | 1 | n | n\*n |
| TOTAL COST | 4 |  | n\*n |
|  | | RUNTIME | 4n\*n |
| O(n^2) |

**System Pseudocode Using Hashtable:**

FUNCTION fileRead {

OPEN file

IF file is open {

SET fileRead = OPEN file

IF fileRead is null

OUTPUT “ERROR”

EXIT

}

WHILE not at end of file {

READ file by parsing each line

CREATE vector courseInfo to hold course info

CREATE string courseData

GETLINE from file and save into courseData

WHILE courseData length > 0 {

IF courseData >= two parameters AND prerequisite(s) exists {

ADD courseData into vector courseInfo

ELSE {

OUTPUT “Error: insufficient course data”

}

}

}

ELSE {

OUTPUT “Error Opening File”

EXIT

}

}

courseSearch {

OPEN AND READ file

WHILE not at end of file {

IF courseNum and courseName == user input

PRINT course info

FOR each prereq of the course {

PRINT prereq info

}

ELSE {

OUTPUT “Course Not Found”

EXIT

}

}

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrerequisites = Hashtable[c]

for each prerequisite p in totalPrerequisites

add prerequisites in Hashtable[p] to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

FOR all key, value pair in courses

PRINT key course name

IF value has prerequisites

FOR each prereq

PRINT prereqs

}

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

FOR all courses

IF the course == courseNumber

PRINT courseInfo

FOR each prereq in Hashtable[course]

PRINT prereq course information

}

WHILE userSelection NOT EQUAL to 9

COUT menu options

COUT “Select a menu option”

CIN userSelection

SWITCH userSelection

CASE 1:

CALL LoadFile passing fileName and courseTable as parameters

CASE 2:

CALL courseTable PrintAll

CASE 9:

COUT “Good Bye”

EXIT PROGRAM

**Hashtable Evaluation:**

The hash table is excellent for quick insertion and deletion. Even though hash tables use the same parsing method as the vector, the insertion function is faster because it has a time complexity of O(1) on average. The worst-case runtime for a hash table is O(n). This can happen if many elements of the input are hashed to the same bucket. Still, with a functioning hash function, collisions can be avoided and the time complexity can be on average O(1) or constant time complexity which is perfect for a large inputs. Searching the hash table would also be O(1), O(n) in worst-case because of collisions. The time complexity for printing a single entry from a hash table would be O(1) because the hash function is computed and the value corresponding to a given key is printed out. In order to output the full hash table, the time complexity would be O(n2 ) because it requires a nested while loop to ensure that each node is reached, along with each subsequent pointed node. The space complexity of the hash table is O(n) because the size is dependent on the size of the input. The space complexity does not require any additional memory other than the space already allocated memory to the return value.

**Run Time Analysis Chart for Hashtable:**

|  |  |  |  |
| --- | --- | --- | --- |
| **CODE** | **LINE COST** | **# OF EXECUTIONS** | **TOTAL COST** |
| FOR ALL COURSES | 1 | n | n |
| IF NOT UINT MAX | 1 | n | n |
| PRINT COURSE INFO | 1 | n | n |
| FOR EACH PREREQ | 1 | n\*n | n\*n |
| PRINT PREREQ INFO | 1 | n\*n | n\*n |
| SET NODE TO NEXT PTR | 1 | n | n |
| WHILE NODE NOT NULLPTR | 1 | n\*n | n\*n |
| PRINT COURSE INFO | 1 | n\*n | n\*n |
| TOTAL COST | 8 |  |  |
|  | | RUNTIME | 8n\*n |
| O(n^2) |

**Binary Search Tree Pseudocode:**

FUNCTION fileRead {

OPEN file

IF file is open {

SET fileRead = OPEN file

IF fileRead is null

OUTPUT “ERROR”

EXIT

}

WHILE not at end of file {

READ file by parsing each line

CREATE vector courseInfo to hold course info

CREATE string courseData

GETLINE from file and save into courseData

WHILE courseData length > 0 {

IF courseData >= two parameters AND prerequisite(s) exists {

ADD courseData into vector courseInfo

ELSE {

OUTPUT “Error: insufficient course data”

}

}

}

ELSE {

OUTPUT “Error Opening File”

EXIT

}

}

courseSearch {

OPEN AND READ file

WHILE not at end of file {

IF courseNum and courseName == user input

PRINT course info

FOR each prereq of the course {

PRINT prereq info

}

ELSE {

OUTPUT “Course Not Found”

EXIT

}

}

}

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrerequisites = left and right child of Node c

FOR each prerequisite p in totalPrerequisites

ADD left and right Nodes of node p to totalPrerequisites

PRINT number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

FOR all Nodes as courses

PRINT courseName

IF course has left node

PRINT left node as prereq

IF course contains right node

PRINT right node as prereq

}

void printCourseInformation(Tree<Course> courses, String courseNumber) {

FOR all Nodes

IF course is == courseNumber

PRINT node info

IF course contains left node

PRINT left node as prereq course info

IF course contains right node

PRINT right node as prereq course infor

END

ELSE

IF course contains left node

GO TO left node

IF course contains right node

GOTO right node

}

WHILE userSelection NOT EQUAL to 9

COUT menu options

COUT “Select a menu option”

CIN userSelection

SWITCH userSelection

CASE 1:

CALL LoadFile passing fileName and courseTable as parameters

CASE 2:

CALL courseTable PrintAll

CASE 9:

COUT “Good Bye”

EXIT PROGRAM

**Binary Search Tree Evaluation:**

The binary search tree is beneficial for quick insertion, searching, deletion and printing. The time complexity for a binary search tree is on average O(logN). The binary search tree benefits from making only as many comparisons as the height of the tree. The worst-case time complexity for the binary search tree is O(n). This means all the nodes are on the same branch. So, the height of the tree correlates to the number of elements in the input. The worst-case occurs if the binary search tree is unbalanced. Searching for a result only requires the algorithm to make as many comparisons as the height of the tree, So, a balanced tree will result in fast search results. When inserting, the algorithm will search for an empty position O(logN) times until it locates a new node. Printing out each of the nodes could result in a worst-case time complexity of O(n) because the node to be printed could have a prerequisite causing the loop to be run a few times. When printing the entire tree in order, the traversal time is would also be O(n2 ), The space complexity of all binary search tree functions is O(n) for each node.

|  |  |  |  |
| --- | --- | --- | --- |
| **CODE** | **LINE COST** | **# OF EXECUTIONS** | **TOTAL COST** |
| IF NODE NOT NULLPTR | 1 | n | n |
| RECURSIVE CALL WITH LEFT PTR | 1 | n | n |
| PRINT COURSE INFO | 1 | n | n |
| FOR EACH PREREQ | 1 | n | n |
| PRINT PREREQ INFO | 1 | n\*n | n\*n |
| RECURSIVE CALL WITH RIGHT PTR | 1 | n | n |
| TOTAL COST | 8 |  |  |
|  | | RUNTIME | 6n\*n |
| O(n^2) |

**Advantages and Disadvantages:**

Each data structure has its own advantages and disadvantages. The vector data structure is inherently fast, but considerably slower in comparison to a hash table or a binary search tree. However, vectors are considerably easier to implement in code. In regard to speed, the hash table is the fastest of the three with a time complexity of O(1) while performing all functions. One disadvantage of a hash table is the probability of implementing a hash table that will have collisions. If the hash table collides, it can lead to the run time increasing dramatically. So, if the key is unknown, it might be beneficial to the coder and the client to choose a different data structure.

**RECOMMENDATION:**

Due to its average speed and efficiency, my recommendation would have to be the hash table data structure. Because the key is known in this scenario (course number), it would be the most efficient algorithm to store the large data set that is required. Since hash tables are easily adaptable, it will continue to function as ABC University increases their course catalogue. A functioning hash table is extremely effective for inserting, searching, and deleting elements. The space required for each of the data structures is the same at O(n). So, the decision comes down to the time each of the data structures requires. This is ultimately why I fully recommend the hash table data structure.