Alex Crosswhite

SNHU

CS 300

DSA: Analysis and Design

**Project 1: Sorting Pseudocode**

**Menu Pseudocode**

SET userChoice to 0

WHILE (userChoice is not equal to 9)

PRINT menu options

1. Load Data Structure
2. Print Course List

3. Print Course

9. Exit

Switch(choice)

case 1:

courses = LoadDataStructure()

BREAK

case 2:

printCourseList(courses)

BREAK

case 3:

searchCourse(courses)

BREAK

END WHILE

Return 0

**Print List Pseudocode**

printCourseList(vector<Course> courses)

DECLARE n = length of courses

DECLARE i

DECLARE j

FOR (i = 0, i < n - 1)

FOR (j = 0, j < n – 1 – 1)

If (courses[j]->courseNumber > courses[j + 1]->courseNumber)

Swap courses [j + 1] and courses[j]

INCREMENT j

END FOR

INCREMENT i

END FOR

FOR (I = 0, I < n)

printCourse(courses[i])

END FORS

**Runtime Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **readFile Function** | **Line Cost** | **Number of Times Executed** | **Total Cost** | |
| FUNCTION readFile(File f, lines) | 1 | 1 | 1 | |
| DECLARE courseNumber, courseTitle, prerequisite, line: String | 1 | 1 | 1 | |
| DECLARE I = 0, j = 0: Int | 1 | 1 | 1 | |
| DECLARE bool = TRUE: Bool | 1 | n | n | |
| WHILE (not at end of file) | 1 | n | n | |
| courseInfo = SPLIT (READLINE(f, line),  DELIMETER = ,) | 1 | n | n | |
| APPEND line to lines | 1 | n | n | |
| IF length of courseInfo is less than 2 | 1 | n | n | |
| bool = FALSE | 1 | 1 | 1 | |
| BREAK | 1 | 1 | 1 | |
| courseNumber[i] = courseInfo[0] | 1 | n | n | |
| courseTitle[i] = courseInfo[1] | 1 | n | n | |
| INCREMENT i | 1 | n | n | |
| IF length of courseInfo is equal to or greater  than 2 | 1 | n | n | |
| FOR k = 2 for courseInfo length | 1 | n | n | |
| Prerequisite[j] = courseInfo[k] | 1 | n | n | |
| INCREMENT j | 1 | n | n | |
| IF bool is equal to TRUE | 1 | 1 | 1 | |
| FOR each p in prerequisite | 1 | n | n | |
| IF p is NOT in courseNumber | 1 | n | n | |
| bool is equal to FALSE | 1 | 1 | 1 | |
| BREAK | 1 | 1 | 1 | |
| RETURN | 1 | 1 | 1 | |
| Total Cost | | | | 14n+9 |
| Runtime | | | | O(n) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Vector to Create Course Objects** | **Line Cost** | **Number of Times Executed** | **Total Cost** | |
| FUNCTION for createObject (Courses <Course>, File f) | 1 | 1 | 1 | |
| Lines[] = “ “ | 1 | 1 | 1 | |
| IF readFile (f, Lines) = TRUE | 1 | n | n | |
| APPEND new course (line) TO Courses | 1 | n | n | |
| ELSE PRINT “Error file cannot be read” | 1 | 1 | 1 | |
|  |  |  |  | |
| FUNCTION for dataStructure | 1 | 1 | 1 | |
| INITIATE CLASS Course | 1 | 1 | 1 | |
| DECLARE number, title, prerequisites: String | 1 | 1 | 1 | |
| number = SPLIT (line, DELIMETER = ,) [0] | 1 | 1 | 1 | |
| title = SPLIT (line, DELIMETER = ,) [1] | 1 | 1 | 1 | |
| IF LENGTH of the SPLIT is greater than 2 | 1 | 1 | 1 | |
| Prerequisite = SPLIT (line)[2 to LENGTH of SPLIT (line,  DELIMETER = ,)] | 1 | 1 | 1 | |
| Total Cost | | | | 2n+11 |
| Runtime | | | | O(n) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hash Table to Create Course Objects** | **Line Cost** | **Number of Times Executed** | **Total Cost** | |
| FUNCTION for createObject (Courses <Course>, File f) | 1 | 1 | 1 | |
| Lines[] = “ “ | 1 | 1 | 1 | |
| IF readFile (f, Lines) = TRUE | 1 | n | n | |
| APPEND new course (line) TO Courses | 1 | n | n | |
| ELSE PRINT “Error file cannot be read” | 1 | 1 | 1 | |
|  |  |  |  | |
| Function for dataStructure | 1 | 1 | 1 | |
| INITITATE CLASS Course | 1 | 1 | 1 | |
| DECLARE Number, Title, Prerequisite: String | 1 | 1 | 1 | |
| Number = SPLIT(line, DELIMETER = ,) [0] | 1 | 1 | 1 | |
| Title = SPLIT(line, DELIMETER = ,) [1] | 1 | 1 | 1 | |
| IF (LENGTH of SPLIT(line, DELIMETER = ,) > 2 | 1 | 1 | 1 | |
| Prerequisites = SPLIT(line)[ 3 to LENGTH of SPLIT(line,  DELIMETER = ,)] | 1 | 1 | 1 | |
| Total Cost | | | | 2n+10 |
| Runtime | | | | O(n) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Binary Search Tree to Create Course Objects** | **Line Cost** | **Number of Times Executed** | **Total Cost** | |
| Struct Course | 1 | 1 | 1 | |
| DECLARE courseName, courseNumber: String | 1 | 1 | 1 | |
| DECLARE Vector<String> Prerequisite | 1 | 1 | 1 | |
| Struct Node | 1 | 1 | 1 | |
| SET Course course | 1 | 1 | 1 | |
| SET key for course | 1 | 1 | 1 | |
| IF (element exists for the given key) | 1 | 1 | 1 | |
| RETURN node-> course | 1 | 1 | 1 | |
| IF (no element exists for the given key) | 1 | 1 | 1 | |
| RETURN course | 1 | 1 | 1 | |
| INTIATE CLASS Tree | 1 | 1 | 1 | |
| SET Node root | 1 | 1 | 1 | |
| Void Insert(Course course) | 1 | 1 | 1 | |
| IF (root is null) | 1 | 1 | 1 | |
| root equals newNode(course) | 1 | 1 | 1 | |
| ELSE this->addNode(root, course) | 4 | n | 4n | |
| | addNode(Node\* node, Course course) | | 1 | 1 |  | |
| | IF (node->bidId > 0) | | 1 | 1 |  | |
| | node->right equals new Node(course) | | 1 | 1 |  | |
| | ELSE this->addNode(node->right, course) | | 1 | 1 |  | |
| Total Cost | | | | 4n+15 |
| Runtime | | | | O(n) |

**Advantages and Disadvantages of Data Structures**

**Vector:**

**Advantages –**

Vectors offer a simple structure that is easy to visualize when traversing, searching, sorting, or adding and removing elements. Elements are stored in consecutive index and can be called as needed by their index.

**Disadvantages –**

The method for adding, removing, searching, sorting, and traversing requires many more direct comparisons than other methods. In large vectors this can be more intensive and take quite a lot of time. When adding or removing an element, each element after the new element must be shifted individually.

**Hash Table:**

**Advantages –**

Hash tables can be searched very quickly using a hash key. As elements are added or removed the table can be resized continue to provide quick access to items.

**Disadvantages –**

The process for increasing the hash tables size can lead to space being taken up that isn’t being utilized. Unless the tables size is known and unchanging hash tables can lead to suboptimal use of space as they add or remove elements. Another common issue for hash tables are collisions. Improperly assigning more than one item to a bucket can result in unintended problems. Care must be taken to deal with collisions when using hash tables.

**Binary Search Tree:**

**Advantages –**

Accessing items in a binary search tree can be done quickly, particularly if they need to be accessed in ascending or descending order. Items can be efficiently added or removed in the proper order. Binary search trees also offer a wide variety of very practical uses in day-to-day computer systems like creating file systems.

**Disadvantages –**

The tree’s shape is dependent on which item makes up the root node. This can determine the number of comparisons that might be needed for any given search, it also affects the balance of the tree. An effective tree is properly balanced in order to provide the most even distribution of items. A balanced tree can be searched and accessed much more quickly.

**Recommended Data Structure**

I recommend the use of a binary search tree for this project. While a vector may be able to print in order, it must first be sorted. A binary search tree has the advantage of being sorted during its creation and as each addition or removal is completed. A hash table could provide a quicker option for searching, the disadvantages present in potential collisions and the unnecessary use of space make it a less desirable option than a binary search tree.