Marysha Brown

CS-300 DSA: Analysis and Design

10/4/2022

Project One Pseudocode, Evaluation, and Analysis

# Linked List Pseudocode

//Menu

Main function()

//openFile

Initialize string fileName

Initialize int userChoice

WHILE (userChoie != 9):

OUTPUT "Course Program Menu"

OUTPUT "Options:"

OUTPUT "1. Load Courses"

OUTPUT "2. Print Course List"

OUTPUT "3. Print Course"

OUTPUT "9. Exit Program"

OUTPUT "Enter choice: "

GET userChoice from user input

switch(usersChoice):

case 1:

OUTPUT "Enter file name: "

GET filename from user input and SET to fileName

CALL loadCourses with fileName

break

case 2:

CALL printList function for the courseList

break

case 3:

OUTPUT "Enter course number to print course:"

GET courseNumber from user input

CALL printLinkedListCourse function for courseNumber

break

end switch

end while

OUTPUT "Thank you, heave a great day!"

RETURN 0

//Load data into Linked List

loadCourses(filename)

fin.open(fileName)

IF (inputFile open failed):

display error

endif

//read data

WHILE (inputFile is open):

declare fileLine variable of a string type

WHILE getline(inputFile, fileLine):

Split line by commas

//load data into vector

IF (num of tokens >= 2):

SET token[0] equal to courseNumber

SET token[1] equal to courseName

FOR tokens[2] and more:

coursePrereq.push\_back prereq

end for

endif

end while

end while

inputFile.close

//sort while in vector of objects

CALL sort(vector.begin(), vector.end(), sortcol)

//validate prerequisites

FOR data in vector:

newcourse.setName(token[0])

newcourse.setTitle(token[1])

FOR tokens[2] and more:

FOR each token:

IF token[i] matches token[0] in file:

newcourse.setPrereq(token[i])

endif

end for

end for

CALL Append newcourse to Course Linked List

end for

//Sort vector function

sortcol(v1, v2)

return v1[0] < v2[0]

//load into linked list

Append(course)

Create new node with the course

IF (head node == nullptr):

SET new node to head and tail

end if

ELSE:

SET current tail node to new node

SET tail to new node

end else

//Display all course information from Linked List (unordered without storting function)

printList()

SET current node to head node

WHILE(current):

OUTPUT course information: courseNumber and courseName

IF (more than 0 coursePrereqs):

FOR loop to iterate through the coursePrereqs with iterator:

OUPUT coursePrereqs

end for

endif

OUTPUT blank line between courses

SET current node to the next node

end while

//Display course information from Linked List

printLinkedListCourse(string courseNumber)

FOR (start (integer i=0) to length of vector of courses, increment i++):

IF (courseNumber[i] == courseNumber):

OUTPUT courseNumber, courseName, coursePrereq information

endif

end for

# Hash Table Pseudocode

//Menu

Main function()

//openFile

Initialize string fileName

Initialize int userChoice

WHILE (userChoie != 9):

OUTPUT "Course Program Menu"

OUTPUT "Options:"

OUTPUT "1. Load Courses"

OUTPUT "2. Print Course List"

OUTPUT "3. Print Course"

OUTPUT "9. Exit Program"

OUTPUT "Enter choice: "

GET userChoice from user input

switch(usersChoice)

case 1:

OUTPUT "Enter file name: "

GET filename from user input and SET to fileName

CALL loadCourses with fileName

break

case 2:

CALL printList function for the courseList

break

case 3:

OUTPUT "Enter course number to print course:"

GET courseNumber from user input

CALL printHashTableCourse function for courseNumber

break

end switch

end while

OUTPUT "Thank you, heave a great day!"

RETURN 0

//Load data into Hash Table

loadCourses(filename)

fin.open(fileName)

IF (inputFile open failed):

display error

endif

//read data

WHILE (inputFile is open):

declare fileLine variable of a string type

WHILE getline(inputFile, fileLine):

Split line by commas

//load data into vector

IF (num of tokens >= 2):

SET token[0] equal to courseNumber

SET token[1] equal to courseName

FOR tokens[2] and more:

coursePrereq.push\_back prereq

end for

endif

end while

end while

inputFile.close

//sort while in vector of objects

CALL sort(vector.begin(), vector.end(), sortcol)

//validate prerequisites

FOR data in vector:

newcourse.setName(token[0])

newcourse.setTitle(token[1])

FOR tokens[2] and more:

FOR each token:

IF token[i] matches token[0] in file:

newcourse.setPrereq(token[i])

endif

end for

end for

CALL Insert newcourse to Course Hash Table

end for

//Sort vector function

sortcol(v1, v2)

return v1[0] < v2[0]

//load into hash table

Insert(course)

CALCULATE key for course

GET node from key

IF (current == nullptr):

SET new node to the position of the course with key

endif

ELSE if node is found:

IF (current->key == UINT\_MAX):

SET the old node to the key

SET the old node to the course

SET the old node pointing to null

endif

ELSE:

WHILE (current->next != nullptr):

SET node to next node

end while

SET old node point to the new node with the course and key

end else

end else

//Display all course information from Hash Table

printList()

FOR loop to iterate through the table with iterator:

OUTPUT courseNumber and courseName

IF (more than 0 coursePrereqs):

FOR loop to iterate through the coursePrereqs with iterator:

OUPUT coursePrereq information

end for

end if

SET node to the next iteration

//traversing lists in the table

WHILE (node != nullptr):

OUTPUT courseNumber and courseName

IF (more than 0 coursePrereqs):

FOR loop to iterate through the coursePrereqs with iterator:

OUPUT coursePrereq information

end for

end if

SET node to next node

end while

OUTPUT blank line between courses

end for

//Display course information from Hash Table

printHashTableCourse(string courseNumber)

CALCULATE key for courseNumber

GET node using key

IF (nothing found for the key):

OUPUT "Course not found"

endif

IF (something is found for the key):

OUTPUT courseNumber, courseName, coursePrereq information

endif

WHILE (node != nullptr):

IF (current node matches the node):

OUTPUT courseNumber, courseName, coursePrereq information

endif

SET node to next node

end while

# Binary Search Tree Pseudocode

//Menu

Main function()

//openFile

Initialize string fileName

Initialize int userChoice

WHILE (userChoie != 9):

OUTPUT "Course Program Menu"

OUTPUT "Options:"

OUTPUT "1. Load Courses"

OUTPUT "2. Print Course List"

OUTPUT "3. Print Course"

OUTPUT "9. Exit Program"

OUTPUT "Enter choice: "

GET userChoice from user input

switch(usersChoice)

case 1:

OUTPUT "Enter file name: "

GET filename from user input and SET to fileName

CALL loadCourses with fileName

break

case 2:

CALL printList function for the courseList

break

case 3:

OUTPUT "Enter course number to print course:"

GET courseNumber from user input

CALL printBtsCourse function for courseNumber

break

end switch end while

OUTPUT "Thank you, heave a great day!"

RETURN 0

//Load data into Binary Tree

loadCourses(filename)

fin.open(fileName)

IF (inputFile open failed):

display error

endif

//read data

WHILE (inputFile is open):

declare fileLine variable of a string type

WHILE getline(inputFile, fileLine):

Split line by commas

//load tokens into vector

IF (num of tokens >= 2):

SET token[0] equal to courseNumber

SET token[1] equal to courseName

FOR tokens[2] and more:

coursePrereq.push\_back prereq

end for

endif

end while

end while

inputFile.close

//validate prerequisites

Create new Course

FOR data in vector:

newcourse.setName(token[0])

newcourse.setTitle(token[1])

FOR tokens[2] and more:

FOR each token:

IF token[i] matches token[0] in file:

newcourse.setPrereq(token[i])

endif

end for

end for

CALL Insert newcourse to Course binary tree

end for

//Insert into binary tree

Insert (current node, course)

IF (current == nullptr):

SET root equal to new node course

endif

ELSE IF (courseNumber is smaller than current courseNumber):

Recursively call Insert(current->left, course)

end else if

ELSE:

Recursively call Insert(current->right, course)

end else

//Display all course information from Binary Tree in order

inOrder(Node\* node)

IF (node != nullptr):

inOrder(node->left)

OUTPUT courseNumber, courseName, coursePrereq information

inOrder(node->right)

endif

//Display course information from Binary Tree

printBtsCourse(string courseNumber)

SET current node equal to root

WHILE(current != nullptr):

IF (current courseNumber matches courseNumber):

OUTPUT courseNumber, courseName, coursePrereq information

endif

ELSE If (courseNumber is smaller than current courseNumber):

SET current equal to current->left

end else if

ELSE:

SET current equal to curreent->right

end else

end while

# Linked List Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| **loadCourses(filename)** | | | |
| fin.open(fileName) | 1 | 1 | 1 |
| IF (inputFile open failed): | 1 | 1 | 1 |
| display error | 1 | 1 | 1 |
| WHILE (inputFile is open): | 1 | n | n |
| declare fileLine variable of a string type | 1 | 1 | 1 |
| WHILE getline(inputFile, fileLine): | 1 | n | n |
| Split line by commas | 1 | 1 | 1 |
| IF (num of tokens >= 2): | 1 | n | n |
| SET token[0] equal to courseNumber | 1 | n | n |
| SET token[1] equal to courseName | 1 | n | n |
| FOR tokens[2] and more: | 1 | n | n |
| coursePrereq.push\_back prereq | 1 | n | n |
| inputFile.close | 1 | 1 | 1 |
| CALL sort(vector.begin(), vector.end(), sortcol) | 1 | 1 | 1 |
| Create new Course | 1 | 1 | 1 |
| FOR data in vector: | 1 | n | n |
| newcourse.setName(token[0]) | 1 | n | n |
| newcourse.setTitle(token[1]) | 1 | n | n |
| FOR tokens[2] and more: | 1 | n | n |
| FOR each token: | 1 | n | n |
| IF token[i] matches token[0] in file: | 1 | n | n |
| newcourse.setPrereq(token[i]) | 1 | 1 | 1 |
| CALL Append newcourse to Course Linked List | 1 | 1 | 1 |
| Total Cost | | | 13n + 10 |
| Runtime | | | O(n) |
| **sortcol(v1 , v2)** | | | |
| return v1[0] < v2[0] | 1 | 1 | 1 |
| Total Cost | | | 1 | |
| Runtime | | | O(1) | |
| **Append(course)** | | | | |
| Create new node with the course | 1 | 1 | 1 | |
| IF (head node == nullptr): | 1 | 1 | 1 | |
| SET new node to head and tail | 1 | 1 | 1 | |
| ELSE: | 1 | 1 | 1 | |
| SET current tail node to new node | 1 | 1 | 1 | |
| SET tail to new node | 1 | 1 | 1 | |
| Total Cost | | | O(1) | |
| Runtime | | |  | |

# Hash Table Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| **loadCourses(filename)** | | | |
| fin.open(fileName) | 1 | 1 | 1 |
| IF (inputFile open failed): | 1 | 1 | 1 |
| display error | 1 | 1 | 1 |
| WHILE (inputFile is open): | 1 | n | n |
| declare fileLine variable of a string type | 1 | 1 | 1 |
| WHILE getline(inputFile, fileLine): | 1 | n | n |
| Split line by commas | 1 | 1 | 1 |
| IF (num of tokens >= 2): | 1 | n | n |
| SET token[0] equal to courseNumber | 1 | n | n |
| SET token[1] equal to courseName | 1 | n | n |
| FOR tokens[2] and more: | 1 | n | n |
| coursePrereq.push\_back prereq | 1 | n | n |
| inputFile.close | 1 | 1 | 1 |
| CALL sort(vector.begin(), vector.end(), sortcol) | 1 | 1 | 1 |
| Create new Course | 1 | 1 | 1 |
| FOR data in vector: | 1 | n | n |
| newcourse.setName(token[0]) | 1 | n | n |
| newcourse.setTitle(token[1]) | 1 | n | n |
| FOR tokens[2] and more: | 1 | n | n |
| FOR each token: | 1 | n | n |
| IF token[i] matches token[0] in file: | 1 | n | n |
| newcourse.setPrereq(token[i]) | 1 | 1 | 1 |
| CALL Insert newcourse to Course Hash Table | n | n | n2 |
| Total Cost | | | n2 + 13n + 9 |
| Runtime | | | O(n2) |
| **sortcol(v1 , v2)** | | | |
| return v1[0] < v2[0] | 1 | 1 | 1 |
| Total Cost | | | 1 | |
| Runtime | | | O(1) | |
| **Insert(course)** | | | | |
| CALCULATE key for course | 1 | 1 | 1 | |
| GET node from key | 1 | 1 | 1 | |
| IF (current == nullptr): | 1 | 1 | 1 | |
| SET new node to the position of the course with key | 1 | 1 | 1 | |
| ELSE if node is found: | 1 | n | n | |
| IF (current->key == UINT\_MAX): | 1 | n | n | |
| SET the old node to the key | 1 | 1 | 1 | |
| SET the old node to the course | 1 | 1 | 1 | |
| SET the old node pointing to null | 1 | 1 | 1 | |
| ELSE: | 1 | n | n | |
| WHILE (current->next != nullptr): | 1 | n | n | |
| SET node to next node | 1 | n | n | |
| SET old node point to the new node with the course and key | 1 | n | n | |
| Total Cost | | | 6n + 7 | |
| Runtime | | | O(n) | |

# Binary Search Tree Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| **loadCourses(filename)** | | | |
| fin.open(fileName) | 1 | 1 | 1 |
| IF (inputFile open failed): | 1 | 1 | 1 |
| display error | 1 | 1 | 1 |
| WHILE (inputFile is open): | 1 | n | n |
| declare fileLine variable of a string type | 1 | n | n |
| WHILE getline(inputFile, fileLine): | 1 | n | n |
| Split line by commas | 1 | n | n |
| IF (num of tokens >= 2): | 1 | n | n |
| SET token[0] equal to courseNumber | 1 | n | n |
| SET token[1] equal to courseName | 1 | n | n |
| FOR tokens[2] and more: | 1 | n | n |
| coursePrereq.push\_back prereq | 1 | n | n |
| inputFile.close | 1 | 1 | 1 |
| Create new Course | 1 | 1 | 1 |
| FOR data in vector: | 1 | n | n |
| newcourse.setName(token[0]) | 1 | n | n |
| newcourse.setTitle(token[1]) | 1 | n | n |
| FOR tokens[2] and more: | 1 | n | n |
| FOR each token: | 1 | n | n |
| IF token[i] matches token[0] in file: | 1 | n | n |
| newcourse.setPrereq(token[i]) | 1 | 1 | 1 |
| Insert newcourse to Course binary tree | 1 | n | n |
| Total Cost | | | 16n + 6 |
| Runtime | | | O(n) |
| **Insert (current node, course)** | | | |
| IF (current == nullptr): | 1 | 1 | 1 |
| SET root equal to new node course | 1 | 1 | 1 |
| ELSE IF (courseNumber is smaller than current courseNumber): | 1 | n | n |
| Recursively call Insert(current->left, course) | 1 | n | n |
| ELSE: | 1 | n | n |
| Recursively call Insert(current->right, course) | 1 | n | n |
| Total Cost | | | 4n + 2 | |
| Runtime | | | O(n) | |

# Analysis

The insertion of the courses into the binary search tree could give us a better runtime complexity if the tree is balanced since we are calculating the worst case. Insertions into the binary search tree and search of a balanced tree could be O(logN) in a best-case scenario. A binary search tree would be beneficial when searching for a course to print since the search path can be shorter due to the tree structure compared to the linked list, which I also calculated as O(n) for the worst case. The binary search tree also sorts itself when populating, so printing the list of courses can easily be done in order from lowest to highest.

There are two options for handling sort and search algorithms in a linked list. The first option is to sort the courses before they are loaded into the structure to use a search algorithm. The other option is to create the linked list from whatever order the courses are in and traverse the entire linked list looking for a specific course. The sorting algorithm can give a worst-case of O(NlogN). While a linked list can be a good data structure if the sorting is not a requirement since you can insert it into a linked list with O(1), it is not best suited for this program.

As for a hash table, while an insertion and search function would be fast with O(1), the courses would also not be ordered unless there is code to sort the courses before they are added to the hash table. I calculated the insertion worst case as O(n). The program is to print a list of ordered courses, which is an essential requirement when deciding on the best data structure. Building the hash table structure is also complex and takes more work than the other two structures since hash tables are built knowing the size of the hash table, and chaining needs to be implemented to account for any collisions that may happen.

Based on the Big O analysis, I would use a binary search tree for my data structure of the course program since the analysis is O(n) for the worst case. I would not need to implement any sorting function with a binary search tree, saving space and time while keeping the program's functionality. Even if the binary search tree were not balanced, it would still be O(n), and I think the trade-off is better than a linked list. A hash table would need even more space to implement correctly; therefore, I would choose a binary search tree as the data structure for the course program.