**Project One: Pseudocode and Evaluation**

CS 300: Data Structures and Algorithms

Dr. Aline Yurik

October 8, 2022

1. **Read, parse input file, create course objects and print course information**

**// Used by all three storage methods**

Struct Course {

Declare string courseNumber

Declare string courseName

Create Vector<string> prerequisites

}

//Open the course file and store each line in a vector

Include fstream

Include string

vector<string> courseFile(string file) {

initialize ifstream newFile

newfile.open(filename, ::in) to open file

IF newFile is\_open() {

declare string line to hold each line

create vector<string> to hold each read line variable

WHILE there are lines available in file using getline()

Push\_back individual string line to vector<string>

ENDWHILE at end of file

ELSE

File did not open return empty vector with error message

ENDIF

CLOSE input stream

RETURN vector<string>

}

**//Vector Data structure Insert Created object**

Include sstream

Vector<Course> createCourse(vector<string> from courseFile) {

Create vector<Course> courses

Initialize lineStream from sstream

Declare string token to store each string from the line input

Declare integer count to track the tokens for formatting

FOR EACH line from the vector<string> from courseFile

Set the count to 0

WHILE iterating through each string to the comma ‘,’ delimiter

IF count is EQUAL to 0

Set Course couresNumber to token

Increment count to get to next string

ELSEIF count is EQUAL to 1

Set courseName of course object to token

Increment count to get to next string

ELSE

IF the current token is a CourseNumber that exists in file

Push\_back token to vector<strings> prerequisites

Increment count in case there are more prereqs

ENDIF

ENDIF

ENDWHILE when end of line is reached “\n”

//Capture formatting errors

IF count is LESS than 2 at the end of input stream

Output formatting error, each course needs name and course number

Empty linestream

Push\_back Course object to vector<Course>

RETURN vector<Course>

}

//Print Vector

void searchAndPrint(vector<Course> courses, userinput courseNumber) {

FOR EACH course in courses

If courseNumber from course Object is EQUAL to userinput courseNumber

Print the course object with prerequisites

FOR EACH prerequisite of the course

Print the prerequisite course information

ENDFOR

ELSE

Print course is unavailable

ENDIF

ENDFOR

**//Hashtable data Structure insert Created object**

// HashTable structure to hold bids

Class HashTable {

STRUCT Node {

Initialize bid

Initialize integer key

Initialize Node as next

Create node constructors-default and overloaded with course and a key to hold Course

}

Create a vector to hold each Node object

Initialize the hashtable size

Initialize hash function with default key

Create public methods for implementation

}

Implement Hashtable() {

Initialize node structure by resizing table

}

Implement destructor Hashtable() {

Node.erase to free storage when class is destroyed\

}

METHOD to calculate hash value:: hash(integer key) {

RETURN the modulo result of key % hashtable size

}

void createCourse(vector<string> from courseFile) {

Create Course course

Initialize lineStream from sstream

Declare string token to store each string from the line input

Declare integer count to track the tokens for formatting

FOR EACH line from the vector<string> from courseFile

Set the count to 0

WHILE iterating through each string to the comma ‘,’ delimiter

IF count is EQUAL to 0

Set Course courseNumber to token

Increment count to get to next string

ELSEIF count is EQUAL to 1

Set courseName of course object to token

Increment count to get to next string

ELSE

IF the current token is a CourseNumber that exists in file

Push\_back token to vector<strings> prerequisites

Increment count in case there are more prereqs

ENDIF

ENDIF

ENDWHILE when end of line is reached “\n”

//Capture formatting errors

IF count is LESS than 2 at the end of input stream

Output formatting error, each course needs name and course number

Empty linestream

HashTable :: INSERT () new Course Object

}

METHOD to calculate hash value:: hash(integer key) {

RETURN the modulo result of key % hashtable size

}

METHOD Insert(Course course) to insert created courses into hashtable {

Convert courseNumber to Integer value

Create key for the given course by calling hash(courseNumber)

SET node at the key position

IF node at position is null

Create a new node and insert into table at position

ELSE IF the node at the position is UINT\_MAX

SET previous node key and course object to position and point next to null pointer

ELSE

WHILE items still exist at node where next is not null

ITERATE to next position until NULL

END WHILE

APPEND new node to key position for object chaining

ENDIF

}

//Print course information based on search

void searchAndPrint(Hashtable<Course> courses, userinput string courseNumber) {

CONVERT courseNumber to Integer value

CREATE key using Hash(courseNumber)

WHILE node at key position is not equal to null

IF courseNumber is EQUAL to userinput courseNumber

Print the course object with prerequisites

FOR EACH prerequisite of the course

ADD number of prereqs

Print the prerequisite course information

ENDFOR

Print total number of prereqs

ELSE

MOVE to next node

ENDIF

Print course was not found

ENDFOR

**//Binary Search Tree Data Structure Insert and create object**

Struct Node {

Course course

Node \*left;

Node \*right;

Node constructor

Left = nullptr

Right = nullptr

INITIALIZE node with a course object

}

Class BinarySearchTree {

Private:

SET root NODE

Void addNode(Node node, Course course)

Void searchAndPrint (Tree<Course> courses, String courseNumber)

Public:

BinarySearchTree()

Void Print()

Void Insert(Course course)

}

Implement method BinarySearchTree() {

SET root value to nullptr

}

Implement destructor BinarySearchTree() {

Recurse from root and delete each node

}

METHOD to print courses Print{

CALL private print function, this enhances data encapsulation

}

METHOD Insert(Course course) to insert created courses into BinarySearchTree {

IF root value in tree node is Null

SET root value to node with Course Object

ELSE

RECURSIVELY call addNode input (node, course Object) until Node is null to input new node

ENDIF

}

METHOD addNode to insert new node into the tree {

IF input courseNumber is less than current Node courseNumber

IF node to the left is Null

Add new Node to left of current Node

ELSE

RECURSIVELY call addNode to move to next Node position

ENDIF

ELSE courseNumber is greater than current Node courseNumber

IF node to the right is pointing Null

ADD new Node to the right of current Node

ELSE

RECURSIVELY call addNode to move to next Node position

ENDIF

ENDIF

}

//Create each course based on vector line input and check for formatting

Include sstream

void createCourse(vector<string> from courseFile) {

Create Course course

Initialize lineStream from sstream

Declare string token to store each string from the line input

Declare integer count to track the tokens for formatting

FOR EACH line from the vector<string> from courseFile

Set the count to 0

WHILE iterating through each string to the comma ‘,’ delimiter

IF count is EQUAL to 0

Set Course courseNumber to token

Increment count to get to next string

ELSEIF count is EQUAL to 1

Set courseName of course object to token

Increment count to get to next string

ELSE

IF the current token is a CourseNumber that exists in file

Push\_back token to vector<strings> prerequisites

Increment count in case there are more prereqs

ENDIF

ENDIF

ENDWHILE when end of line is reached “\n”

//Capture formatting errors

IF count is LESS than 2 at the end of input stream

Output formatting error, each course needs name and course number

Empty linestream

BinarySearchTree :: INSERT (new Course Object)

}

//Print course information based on search

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

SET current Node to the root Node

WHILE current node is not pointing to NULL

IF current node course is equal to inputCourseNumber

PRINT CourseInformation including prerequisites

ENDIF

IF input courseNumber is less than current node CourseNumber

Move current node to the next node to the left

ELSE

Move current node to the next node on the right

ENDIF

ENDWHILE when CourseNumber is found

PRINT Course information

RECURSIVELY call searchAndPrint for EACH prerequisite Course number if it exists

1. **Create Pseudocode for a menu**

INITITALIZE data structure

CREATE new instance of data structure

INITIALIZE Object

INITIALZE user choice to 0

While user choice is NOT EQUAL to 4, display menu

PRINT “MENU:”

PRINT choice 1 as “1. Load courses”

PRINT choice 2 as “2. Print Course List” – alphanumeric list

PRINT choice 3 as “2. Print Course”- individual course with prereqs

PRINT choice 4 as “Exit” to leave application

INPUT user choice value 1 to 4

SWITCH user choice input

CASE 1

LOAD courses into data structure

CASE 2

PRINT alphanumeric list of all courses available

CASE 3

USE respective Data structure Search method to find course

IF course is found

PRINT course information and prereqs

ELSE

PRINT “Course not found.”

ENDIF

CASE 4

BREAK the while loop and exit app

1. **Print in alphanumeric order**

**//Vector – using quick sort applied to course list**

METHOD partition(Vector<Courses>, Integer begin, integer end)

SET pivot to middle index of vector

WHILE bid partition elements are GREATER THAN 1

WHILE bids starting at begin index are less than the value at the pivot

INCREMENT begin index

ENDWHILE

WHILE bids starting at pivot index are less than the end index

DECRIMENT end index

ENDWHILE

IF begin index is GREATER OR EQUAL to high index

ENDWHILE – bid partition element is LESS THAN 1

ELSE

SWAP bids at begin index and bids at high index

INCREMENT begin

DECREMENT high

ENDIF

RETURN high index

METHOD quicksort(Vector<Courses>, Integer begin, integer end)

IF begin GREATER OR EQUAL to end

RETURN -partition is sorted

ENDIF

SET middle element to partition()

quickSort(bids, begin, middle element)

quickSort(bids, middle element +1, end)

METHOD void printCourses(Vector<Courses>)

quicksort(Vector<Courses>, begin, end)

FOR EACH object in courses list

PRINT Course

**//HASHTABLE**

//There is no sorting method for a hashtable, so easiest method may be to convert to vector

METHOD Vector<Courses> tableToVector(HashTable<Courses>)

INITIALIZE vector variable to size of HashTable

INITIALIZE vector iterator value (i) to 0

FOR EACH node in Hashtable

IF current node is NOT EQUAL NULL

SET Vector[i] to current node value

INCREMENT vector iterator

ELSE

MOVE to next Node

ENDIF

ENDFOR

RETURN Vector<Courses>

//print courses

void printCourses(Vector<Courses>)

quicksort(Vector<Courses>, begin, end)

FOR EACH object in courses list

PRINT Course

**//TREE – inOrder print**

METHOD inOrder(Node node)

IF node is not equal to null

RECURSIVE call inOrder with left node input

Print node course information from left Subtree

RECURSIVE call inOrder with right node input

Print remaining course information from right subtree

1. **RUN-TIME Evaluation**

VECTOR

| **VECTOR READ AND CREATE COURSE** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **CREATE VECTOR<COURSES> courses** | 1 | 1 | 1 |
| **INITIALIZE linestream** | 1 | 1 | 1 |
| **DECLARE STRING TOKEN** | 1 | 1 | 1 |
| **DECLARE INTEGER COUNT** | 1 | 1 | 1 |
| **FOR EACH LINE FROM vector<string>** | 1 | n | n |
| **SET COUNT TO 0** | 1 | n | n |
| **WHILE ITERATING THROUGH EACH STRING** | n | n2 | n2 |
| **IF COUNT EQUAL TO 0** | 1 | n | n |
| **SET Course CourseNumber to token** | 1 | n | n |
| **INCREMENT COUNT** | 1 | n | n |
| **ELSEIF COUNT EQUAL TO 1** | 1 | n | n |
| **SET courseName of course to token** | 1 | n | n |
| **INCREMENT COUNT** | 1 | n | n |
| **ELSE**  **IF current token exists** | 1 | n | n |
| **PUSH\_BACK token to vector<string> prereqs** | 1 | n | n |
| **INCREMENT COUNT if more prereqs** | 1 | n | n |
| **IF count less than 2**  **OUTPUT FORMAT ERROR** | 1 | 1 | 1 |
| **EMPTY Linestream** | 1 | n | n |
| **INSERT Course into vector<Course>** | 1 | n | n |
| **RETURN vector<Course>** | 1 | 1 | 1 |
| **Total Cost** | | | n2+13n + 6 |
| **Runtime** | | | O(n2) |

HASHTABLE

| **HASHTABLE READ AND CREATE COURSE** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **CREATE Course course** | 1 | 1 | 1 |
| **INITIALIZE linestream** | 1 | 1 | 1 |
| **DECLARE STRING TOKEN** | 1 | 1 | 1 |
| **DECLARE INTEGER COUNT** | 1 | 1 | 1 |
| **FOR EACH LINE FROM vector<string>** | 1 | n | n |
| **SET COUNT TO 0** | 1 | n | n |
| **WHILE ITERATING THROUGH EACH STRING** | n | n2 | n2 |
| **IF COUNT EQUAL TO 0** | 1 | n | n |
| **SET Course CourseNumber to token** | 1 | n | n |
| **INCREMENT COUNT** | 1 | n | n |
| **ELSEIF COUNT EQUAL TO 1** | 1 | n | n |
| **SET Course courseName of course to token** | 1 | n | n |
| **INCREMENT COUNT** | 1 | n | n |
| **ELSE**  **IF current token exists** | 1 | n | n |
| **PUSH\_BACK token to vector<string> prereqs** | 1 | n | n |
| **INCREMENT COUNT if more prereqs** | 1 | n | n |
| **IF count less than 2**  **OUTPUT FORMAT ERROR** | 1 | 1 | 1 |
| **EMPTY Linestream** | 1 | n | n |
| **HASHTABLE insert(Course course)** | n | n | n |
| **Total Cost** | | | n2+13n + 5 |
| **Runtime** | | | O(n2) |

| **HASHTABLE INSERT COURSE** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **CONVERT courseNumber to Integer** | 1 | 1 | 1 |
| **CREATE key by calling hash(courseNumber)** | 1 | 1 | 1 |
| **SET NODE at KEY position** | 1 | 1 | 1 |
| **IF node at position is null** | 1 | 1 | 1 |
| **CREATE NEW NODE AND INSERT** | 1 | 1 | 1 |
| **ELSE IF node is UINT\_MAX**  **SET previous node key and course object to position and point next to null** | 1 | 1 | 1 |
| **ELSE**  **WHILE items still exist at node** | 1 | n | n |
| **ITERATE TO NEXT POSITION until null** | 1 | n | n |
| **APPEND new node for object chaining** | 1 | n | n |
| **Total Cost** | | | 3n + 6 |
| **Runtime** | | | O(n) |

BST

| **BINARY SEARCH TREE READ AND CREATE COURSE** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **CREATE Course course** | 1 | 1 | 1 |
| **INITIALIZE linestream** | 1 | 1 | 1 |
| **DECLARE STRING TOKEN** | 1 | 1 | 1 |
| **DECLARE INTEGER COUNT** | 1 | 1 | 1 |
| **FOR EACH LINE FROM vector<string>** | 1 | n | n |
| **SET COUNT TO 0** | 1 | n | n |
| **WHILE ITERATING THROUGH EACH STRING** | n | n2 | n2 |
| **IF COUNT EQUAL TO 0** | 1 | n | n |
| **SET Course CourseNumber to token** | 1 | n | n |
| **INCREMENT COUNT** | 1 | n | n |
| **ELSEIF COUNT EQUAL TO 1** | 1 | n | n |
| **SET Course courseName of course to token** | 1 | n | n |
| **INCREMENT COUNT** | 1 | n | n |
| **ELSE**  **IF current token exists** | 1 | n | n |
| **PUSH\_BACK token to vector<string> prereqs** | 1 | n | n |
| **INCREMENT COUNT if more prereqs** | 1 | n | n |
| **IF count less than 2**  **OUTPUT FORMAT ERROR** | 1 | 1 | 1 |
| **EMPTY Linestream** | 1 | n | n |
| **BinarySearchtree insert(Course course)** | n | n | n |
| **Total Cost** | | | n2+13n + 5 |
| **Runtime** | | | O(n2) |

| **BST INSERT COURSE** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **IF root value of tree is NULL** | 1 | 1 | 1 |
| **SET root value to node with course** | 1 | 1 | 1 |
| **ELSE**  **CALL addNode until node is null to input new node** | n | n | n |
| **Total Cost** | | | n + 2 |
| **Runtime** | | | O(n) |

| **BST addNode** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **IF input courseNumber < current node courseNumber** | 1 | 1 | 1 |
| **IF node to left is Null**  **Add new Node** | 1 | 1 | 1 |
| **ELSE**  **Call addNode to move to next position** | n | n | n |
| **ELSE**  **IF Node to the right is NULL** | 1 | 1 | 1 |
| **Add new node** | 1 | 1 | 1 |
| **ELSE**  **Call addNode to move position** | n | n | n |
| **Total Cost** | | | 2n + 4 |
| **Runtime** | | | O(n) |

1. **Advantages and disadvantages**

Some advantages of using a vector are when sorting the courses alphanumerically as we can use useful methods such as quick sort or selection sort. Another benefit is that extra time is not wasted when trying to store the course when compared to the other methods. The course can be read and created from the file and simply appended to a vector list. There are no other method calls necessary to perform the operation. Some disadvantages are when it comes to searching for a course in the vector list. Most often the index of the course will not be known. Therefore, other methods will need to be created for searching, such as a binary search algorithm or a simple loop iteration. Both of which require more time than other possibilities.

Searching for items is where the hash table would shine. The best-case scenario for implementing this method is a search in constant time. The value of the course number is used as a key in the table, making the course easy to be found. However, some time constraints can be found in the search because of chaining due to collisions. One major disadvantage applying this structure is that hash tables cannot be sorted. This provides major problems when the student needs to print the courses alphanumerically. The program would have to extract the values into another data structure such as a vector then sort and display the values. This would add unnecessary time and memory constraints to the program.

The binary search tree really shines when it comes to inserting and deleting items in the tree. Often, the best-case scenario here is O(log n) which is not possible with the other structures discussed. Another advantage is that printing the sorted list can be implemented easily as the values are stored in values from lowest (left values) to the highest (right values). Some disadvantages to the data structure are that it is best used if it is a completely balanced tree. This can cause certain complications when traversing the tree. Also, searching may not be as efficient with the other implemented methods, especially if other search methods are implemented on the vector data structure.

1. **RECOMMENDATION**

Based on the overall analysis of each data structure, I would suggest a vector would be sufficient for this program. The insertion of the course values can be inserted as we input the line streams a little easier than implementing the hash table or BST methods. When considering sorting the items, it does a sufficient job by applying quick sort in this instance which has Big O (log n) complexity. Finally, in terms of searching for specific course values we can implement a binary search algorithm to give a search result on par with that of BST with a complexity of Big O(log n) as well. The worst case here could be O(n2), however the file is small enough that I do not foresee this being the case.