**6-2 Project One**

Ryan Rieth

Prof. Burns — CS 300 — SNHU

Open & Read a File

Procedure Vector OpenFile(String csvPath)

DECLARE an empty vector called courseList.

DECLARE an empty Course object called course.

DECLARE a file pointer.

OPEN the file with this pointer via the csvPath string.

DECLARE a vector of type String called “row”.

DECLARE Strings “word” and “temp”.

DECLARE String stream “ssLine”.

INSTANTIATE loop index “i” as zero.

WHILE the file pointer stores data from file into temp…

BEGIN

READ entire line in file and store it into ssLine.

WHILE ssLine can be broken up via ‘,’ char and stored into word…

BEGIN

ADD word to row.

END while

IF row[0] is not empty…

BEGIN

SET course.Num as row[0].

IF row[1] is not empty…

SET course.Id as row[1].

END if

IF row[2] is not empty…

SET course.prereq1 as row[2].

END if

IF row[3] is not empty…

SET course.prereq2 as row[3].

END if

END if

ADD this course to courseList vector.

END while

RETURN courseList vector.

END procedure

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

Create Course Objects

Default Constructor CreateCourse()

DECLARE courseNum.

DECLARE courseId.

DECLARE prereq1.

DECLARE prereq2.

INSTANTIATE courseNum as empty.

INSTANTIATE courseId as empty.

INSTANTIATE prereq1 as empty.

INSTANTIATE prereq2 as empty.

END constructor

Procedure Vector CoursesVector(Vector<Course> courseList)

RETURN. // Do nothing since courseList is already a vector.

END procedure

Procedure HashTable CoursesHashTable(Vector<Course> courseList)

CREATE a null HashTable called “courses” of Course type, with a number of buckets equal to size of courseList.

INSTANTIATE “N” as size of courseList.

DECLARE a null “bucket”.

INSTANTIATE “bucketsProbed” as zero.

INSTANTIATE loop index “i” as zero.

FOR every course in courseList…

BEGIN

SET bucket equal to Hash(course.courseNum) modulo N.

WHILE bucketsProbed is less than N…

BEGIN

IF HashTable[bucket] is empty…

BEGIN

HashTable[bucket] = course.

BREAK.

END if

INCREMENT i.

SET bucket equal to Hash(course.courseId) + i + i \* i % N.

INCREMENT bucketsProbed.

END while

END for

RETURN courses hash table.

END procedure

Procedure Tree CoursesTree(Vector<Course> courseList)

CREATE a null Tree called “courses” of Course type.

CREATE a null node called “current”.

FOR every “current” course in courseList…

BEGIN

SET current’s data as course.

CALL procedure InsertCourse(courses, current)[[1]](#footnote-1).

END for loop

RETURN courses Tree.

END procedure

Procedure InsertCourse(courses, nodeToInsert)

INSTANTIATE null node “current”.

IF courses->root is null…

BEGIN

SET courses->root equal to nodeToInsert.

SET node->left equal to null.

SET node->right equal to null.

ELSE

SET current equal to courses->root.

WHILE current is not null…

BEGIN

IF nodeToInsert->courseId is less than current->courseId…

BEGIN

IF current->left is null…

BEGIN

SET current->left as nodeToInsert.

SET current equal to null.

ELSE

SET current equal to current->left.

END if-else

ELSE

IF current->right is null…

BEGIN

SET current->right equal to nodeToInsert.

SET current equal to null.

ELSE

SET current equal to current->right.

END if-else

END if-else

END while

SET node->left equal to null.

SET node->right equal to null.

END if-else

END procedure

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

Print Course Information

Procedure DisplayCourse(Course course)

PRINT course.Num.

PRINT course.Id.

PRINT course.prereq1.

PRINT course.prereq2.

END procedure

Procedure PrintVecCourse(Vector<Course> courseList, String courseNum)

INSTANTIATE loop index “i” as zero.

WHILE i is less than size of courseList…

BEGIN

IF the courseNum of the course at index i in courseList equals courseNum…

BEGIN

CALL DisplayCourse(course).

RETURN.

END if

INCREMENT i.

END while

DISPLAY message of nonexistence.

END procedure

Procedure PrintHashCourse(HashTable<Course> courseList, String courseNum)

INSTANTIATE “bucket” as Hash(intCourseNum).

INSTANTIATE “bucketsProbed” as zero.

INSTANTIATE “N” as size of courseList.

WHILE HashTable[bucket] has not always been empty and bucketsProbed is less than N…

BEGIN

IF HashTable[bucket] is not empty and HashTable[bucket].courseNum equals courseNum…

BEGIN

CALL DisplayCourse(HashTable[bucket]).

RETURN.

END if

SET bucket equal to (bucket + 1) % N.

INCREMENT bucketsProbed.

END while

DISPLAY message of nonexistence.

END procedure

Procedure PrintTreeCourse(Tree<Course> courseList->root, String courseNum)

INSTANTIATE “current” node equal to courseList->root.

IF current is not null…

BEGIN

IF courseNum equals current node’s data…

BEGIN

CALL DisplayCourse(current->course).

RETURN.

ELSE IF courseNum is less than current’s data…

CALL PrintTreeCourse(current->left, courseNum).

RETURN.

ELSE IF courseNum is greater than current’s data…

CALL PrintTreeCourse(current->right, courseNum).

RETURN.

END if-else

END if

PRINT message of nonexistence.

END procedure

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

Display a Menu

/\*

\* There exists a previously declared empty vector “courseList” as well as

\* a String “csvPath”.

\*/

Procedure void DisplayMenu(int choice)

DECLARE adtType as empty.

DECLARE desiredCourse as empty.

INPUT String from user as adtType.

IF choice is equal to 1…

BEGIN

SET courseList as result of OpenFile(csvPath).

IF adtType is vector…

BEGIN

CALL CoursesVector(courseList).

ELSE IF adtType is hashTable…

DECLARE empty hash table “hashCourses”.

INSTANTIATE hashCourses as CoursesHashTable(courseList).

ELSE IF adtType is tree…

DECLARE empty tree “treeCourses”.

INSTANTIATE hashTree as CoursesTree(courseList).

END if-else

ELSE IF choice is equal to 2…

IF adtType is vector…

BEGIN

CALL PrintAllVector(courseList).

ELSE IF adtType is hashTable…

CALL PrintAllHash(hashCourses).

ELSE IF adtType is tree…

CALL PrintAllTree(treeCourses).

END if-else

ELSE IF choice is equal to 3…

INPUT string from user as desiredCourse.

IF adtType is vector…

BEGIN

CALL PrintVecCourse(courseList, desiredCourse).

ELSE IF adtType is hashTable…

Call PrintHashCourse(courseList, desiredCourse).

ELSE IF adtType is tree…

CALL PrintTreeCourse(courseList, desiredCourse).

END if-else

ELSE IF choice is equal to 4…

RETURN. // Not just any ol’ return: the BIG one.

END if-else

DISPLAY menu options.

END procedure

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

List Courses Alphanumerically

Procedure void PrintAllVector(vector<Course> vecList)

DECLARE empty vector called “coursesCopy”.

SET coursesCopy equal to vecList.

SORT coursesCopy.

FOR every course in coursesCopy…

BEGIN

CALL DisplayCourse(course).

END for

CLEAR coursesCopy.

END procedure

Procedure void PrintAllHash(hashTable<Course> hashList)

INSTANTIATE temporary String “tempCourseId” as 50.

INSTANTIATE empty vector “courseIds”.

INSTANTIATE int numBuckets as size of hash table.

INSTANTIATE loop index “i” as zero.

WHILE courseIds size is less than numBuckets…

BEGIN

FOR every bucket in hash table…

BEGIN

IF tempCourseId is greater than hashList[bucket]’s courseId…

BEGIN

SET tempCourseId equal to hashList[bucket]’s courseId.

END if

END for

IF tempCourseId does not equal courseId at index I…

BEGIN

INSERT tempCourseId at end of courseIds vector.

INCREMENT loop index i.

END if

END while

FOR every String “course” in courseIds…

BEGIN

CALL PrintHashCourse(hashList, course).

END for

END procedure

Procedure void PrintAllTree(Node node)

IF node is null…

BEGIN

RETURN.

END if

CALL PrintAllTree(node->left).

CALL DisplayCourse(node->course.courseId).

CALL PrintAllTree(node->right).

END procedure

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

Runtime Analyses

Each of the three data structures will be analyzed for runtime complexity regarding their various uses, from creation to printing information. Each line of code will cost *1 unit* and will be surmised together for a general idea of cost per aspect of each data structure. An important detail to note: these are calculations based on the scripting done for this particular project — different implementations could result in better or worse analyses. Following are the results of this project’s calculations:

| Vector | | |
| --- | --- | --- |
| Procedure | Runtime Complexity | Cost |
| Creating a Course List | O(1) | 1 |
| Printing a Specific Course | O(N) | 7 |
| Printing All Courses | O(N) | 9 |
|  |  | Total Cost: *17 units* |

| Hash Table | | |
| --- | --- | --- |
| Procedure | Runtime Complexity | Cost |
| Creating a Course List | O(N2) | 15 |
| Printing a Specific Course | O(N) | 13 |
| Printing All Courses | O(N2) | 13 |
|  |  | Total Cost: *41 units* |

| Tree | | |
| --- | --- | --- |
| Procedure | Runtime Complexity | Cost |
| Creating a Course List | O(N) | 23 |
| Printing a Specific Course | O(N/2) | 30 |
| Printing All Courses | O(N2) | 26 |
|  |  | Total Cost: *79 units* |

Advantages & Disadvantages

Dealing with multiple data types, the pros and cons of each one seem to surface when least expected — or when least wanted. Vectors, Hash Tables, and Trees are all three data structures that can accomplish the same goal but require, at times, vastly different methods of meeting this demand. Although all three data structures possess the same mindset of storing objects [in this case] into a list of a sort, each one has its own strengths and weaknesses when it comes to such tasks as perusing, inserting, and even creating as a general concept. Data used in my pro-con analysis includes per-line total unit costs as well as runtime complexities[[2]](#footnote-2).

My favorite so far has been the tree (more specifically, the binary search tree). I genuinely cannot express exactly *why* it is my favorite, but the core functionality of its parent-child layout appeals to me. For a task such as storing courses into a course list, it seems immediately to be the best choice as it features linked nodes that rely on some sort of lesser or greater dependency. The idea having classes with prerequisites fits this idea — at least until certain classes have prerequisites that you wouldn’t expect, which has been the case throughout this term. A binary search tree does not allow one “course” to be the prerequisite of two separate courses (i.e., a child cannot have two parents). Unless the course list were to change, this ultimately renders a binary search tree as the *worst* choice, even though finding and removing pieces of the tree are easier when compared to doing so for hash tables or vectors. In my implementation, the tree has a nice balance of linear, exponential, and logarithmic runtime complexities, but with the most sizable code cost of the three ADTs at *79 units*.

Vector is one of the most “basic” ideas of utilizing a list type of functionality in coding. It is easy to use and navigate, keeps track of its size while maintaining an easy way to resize (albeit memory-intensive), and allows data accessing on an incredibly simple level. Where vectors lose their luster is in its one-track-mindset; in other words, a vector is perused linearly and only linearly unless you know the *exact* index data is stored at. If you want to search for a particular data piece, you must begin at the first piece in the vector and peruse each one — even if the one you desire is the very last one. The same reality is true for perusal in reverse, even if yours is the first one. This coupled with vector’s sometimes memory-intensive resizing knocks this data type down a few pegs. The vector, however, is saved by its constant and linear runtime complexities as well as its unsurprising low code cost of *17 units*.

Much like a vector, a hash table finds itself in the middle when it comes to pros and cons. A hash table is essentially a grouping of key-value pairs where the key is mathematically deduced by hash functions, cementing a mindset of unique key-value pairs. Accessing a data piece can be incredibly quick if done right, and deleting a piece or adding a new one is just as simple. However, hash tables are complex and require a deeper understanding of mathematics than the regular data structure, especially when trying to figure out hash functions that minimize or eliminate collisions — which, to add, are a problem trees and vectors do not have to worry about. A hash table can also be very taxing on memory if it is not implemented properly — it’s linear mixed with exponential runtime complexities can attest to this idea. While it is a pro that a hash table has different manners of utilization, learning each one is daunting and opens the doors to more syntax errors than the other data structures. Not to mention: a hash table finds itself with relatively expensive code at *41 units*.

Recommendation

All evidence, analyses, and calculations presented thus far have given very complete ideas of the vector, hash table, and tree data structures. Each one on paper seems capable of serving the same purpose this project has detailed while also maintaining individuality in its pros and cons. These advantages and disadvantages are what distinguish and divide, and this particular application idea of creating a course list, searching through it, printing a specific course, and printing all courses in an alphanumeric order can best be served by only one of the three ADTs.

It is not necessarily justified to pick the ADT with the lowest coding cost solely because its cost is low; the same catch stands for picking an ADT based on its low worst-case runtime complexity. Generally speaking, I believe the **tree** would be the best *onset* choice for this type of project — its cost is high and its runtime complexity can run a bit hot, but a course list where a course may have one or two prerequisites fits the foundational tree concept of “parent” (course) nodes having “child” (prerequisite) nodes. However, the courses would have to follow a stringent prerequisite listing; in other words, no course can be a prerequisite for two separate courses, which *is* the case with this project’s example criteria. With this reality taken into consideration, a hash table would present itself as a viable option with its lower cost and equal-in-intensity overall runtime complexity. We arrive at an issue, however, when we remember *collisions*, or — in this example — courses being stored in the same “bucket” based on their course number. Each bucket in the hash table could, of course, but turned into its own list where several objects can be stored, but that requires much more coding, further understanding, and deeper delving into code that opens the door for much more syntax error. Thus, my firm recommendation for this project having taken all pieces into consideration is **the vector**. The vector offers a simple way to store objects — such as courses — into a list and, coincidentally, offers the lowest coding cost as well as the lowest overall worst-case runtime complexities. It is true that sorting a vector is more difficult than not and its usually linear runtimes could be lapped by, say, a tree’s in the right circumstances, but the balance is more attainable and acceptable than with the other ADTs. It is with these thoughts that I will proceed further into this class utilizing a **vector** for this course application example.

1. See next page. [↑](#footnote-ref-1)
2. Refer to previous page for data review. [↑](#footnote-ref-2)