# CS 300 Pseudocode Project 6-2

//Vector - Milestone 1

struct Course {

String courseNumber

String name

String prerequisites

}

Void displayCourse (Course course) {

Print out “CourseID: ” courseNumber new line<< “Course Name: “ name new line<< “Prerequisites: “ prerequisites endl;

Return;

}

Vector<Course> loadCourses (string csvPath){

Initialize empty course list

cout << "Loading CSV file " << csvPath << endl;

Open file

IF file does not open

Return error

WHILE statement to read until end of file

IF row is empty go to next

Separate each row into fields by commas

courseNumber = field[0]

name = field[1]

prerequisites = any additional fields

Create Course with courseNumber, name, and prerequisites

Add Course to courses Vector

Return

End

}

void searchCourse(Vector<Course> courses, String courseNumber) {

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

void print sortedCourses

sort courseNumber in alphanumeric ascending order

FOR each course

print out the course information

FOR each prerequisite of the course

print the prerequisite course information

}

//Hash Table - Milestone 2

Class Course {

String courseNumber

String name

String prerequisite

}

Create Hash Table

Initialize empty table to hold bids

ReSize Hash Table

Create table size

Resize

Adjust table according to needed size

Rehash all entries to new table

Hash Table Insert

Calculate hash index for courseNumber

IF slot at index is empty

Insert into slot

ELSE

Resolve collision

Add Course to list

IF exceeds threshold

Resize hash table

Open/Read File {

Initialize empty course list

Open file

IF file does not open

Return error

WHILE statement to read until end of file

IF row is empty go to next

Separate each row into fields by commas

courseNumber = field[0]

name = field[1]

prerequisites = any additional fields

Create Course with courseNumber, name, and prerequisites

Add Course to courses Hash Table

Return

End

}

void searchCourse(HashTable<Course> courses, String courseNumber) {

IF courseNumber is found

Print courseNumber, name, prerequisites

FOR prerequisites

IF found, print prerequisites

ELSE, print “none”

ELSE print not found

}

void print sortedCourses

extract course list

sort list in alphanumeric ascending order

FOR each course

print out the course information

FOR each prerequisite of the course

print the prerequisite course information

//Binary Search Tree – Milestone 3

Create left and right nodes

Create Tree

Set root to nullptr

Create order methods(insert, search)

Insert{

IF root is null

Return root

IF current < root

Insert left

ELSE IF current > root

Insert Right

}

Search{

Start at root

WHILE current is not null

IF match found, return match

ELSE look left then right

Return

}

Load courses from file (same logic as open/read above)

Use insert to sort by courseNumber

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

Search for courseNumber

IF found, print courseNumber, name, prerequisites

ELSE print error

Void print sorted courses

List sort courses in-order traversal

FOR each course

print out the course information

FOR each prerequisite of the course

print the prerequisite course information

List

New List

If node is found

Traverse left, add nodes

Traverse right, add nodes

return

}

Menu Functions

void menu(){

initialize data structure

WHILE (choice !=9)

Print:

Menu:

1: Load the file data into the data structure.

2: Print ordered list of courses

3: Print the course title and prerequisites for any individual course.

9: Exit

Enter choice:

SWITCH choice

CASE 1:

Load data into data structure

Print “Data loaded”

BREAK

CASE 2:

IF:

*Vector/Hashtable*

Sort alphanumerically by courseNumber

Print (courseNumber, name, prerequisites)

*Binary Search Tree*

Traverse tree to get courses in order

Print (courseNumber, name, prerequisites)

ELSE Print “No data loaded”

CASE 3:

Print “Enter Course Number”

Read user input

IF course is found

Print (courseNumber, name, prerequisites)

ELSE

Print “not found”

CASE 9:

EXIT program

CATCH invalid entry

Display menu

}

| **Vector Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize empty course list** | 1 | 1 | 1 |
| **Open file** | 1 | 1 | 1 |
| **IF file does not open** | 1 | 1 | 1 |
| **Return error** | 1 | 1 | 1 |
| **WHILE statement to read until end of file** | 1 | n | n |
| **IF row is empty go to next** | 1 | n | n |
| **Separate each row into fields by commas** | 1 | n | n |
| **courseNumber = field[0]** | 1 | n | n |
| **name = field[1]** | 1 | n | n |
| **Prerequisites = any additional fields** | 1 | n | n |
| **Create new Course object** | 1 | n | n |
| **Add Course to courses vector** | 1 | n | n |
| **Return** | 1 | 1 | 1 |
| **Total Cost** | | | 8n + 5 |
| **Runtime** | | | O(n) |

| **Hash Table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Hash Table Creation** | 1 | 1 | 1 |
| **Resize Hash Table** | 1 | n | n |
| **Calculate hash index** | 1 | n | n |
| **Insert into slot** | 1 | n | n |
| **Resolve collision** | 1 | n | n |
| **Resize if exceeds threshold** | 1 | n | n |
| **Initialize empty course list** | 1 | 1 | 1 |
| **Open file** | 1 | 1 | 1 |
| **IF file does not open** | 1 | 1 | 1 |
| **Return error** | 1 | 1 | 1 |
| **WHILE statement to read until end of file** | 1 | n | n |
| **IF row is empty go to next** | 1 | n | n |
| **Separate each row into fields by commas** | 1 | n | n |
| **courseNumber = field[0]** | 1 | n | n |
| **name = field[1]** | 1 | n | n |
| **Prerequisites = any additional fields** | 1 | n | n |
| **Create new Course object** | 1 | n | n |
| **Add Course to Hash Table** | 1 | n | n |
| **Return** | 1 | 1 | 1 |
| **Total Cost** | | | 13n + 6 |
| **Runtime** | | | O(n) |

| **Binary Search Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Node** | 1 | n | n |
| **Create Tree** | 1 | 1 | 1 |
| **IF root is null** | 1 | n | n |
| **Return root** | 1 | n | n |
| **IF current < root** | 1 | n | n |
| **Insert Left** | 1 | n | n |
| **ELSE if current > root** | 1 | n | n |
| **Insert Right** | 1 | n | n |
| **Open file** | 1 | 1 | 1 |
| **IF file does not open** | 1 | 1 | 1 |
| **Return error** | 1 | 1 | 1 |
| **WHILE statement to read until end of file** | 1 | n | n |
| **IF row is empty go to next** | 1 | n | n |
| **Separate each row into fields by commas** | 1 | n | n |
| **courseNumber = field[0]** | 1 | n | n |
| **name = field[1]** | 1 | n | n |
| **Prerequisites = any additional fields** | 1 | n | n |
| **Create new Course object** | 1 | n | n |
| **Add Course to courses tree** | 1 | n | n |
| **Total Cost** | | | 15n + 4 |
| **Runtime** | | | O(n) |

**Advantages/Disadvantages**

When analyzing each of these data structures, there are many advantages as well as some disadvantages. Vectors are incredibly simple and have relatively fast access. The disadvantages also include that it is simple and might require more work to maintain if it was handling complex data that is changing regularly. A hash table can handle insertions much better than a vector but will likely require more memory. It also does not handle sorting data as well as the other data structures we are reviewing. Finally, a binary search tree is going to be the most efficient at sorting, storing, and inserting data. It will use less memory due to the way it searches for data.

**Recommendation**

There are a few factors that come into play when making a final selection for the data structure that will be implemented in this software. The first is what are the needs of ABCU and the second is which data structure will fit those needs. ABCU is specifically looking for software that will help its computer science advisors access course information for students. Since this data is fairly static, we must look for a data structure that can support this, efficiently sort the data, and use minimal memory. For the reasons above, I suggest using the Vector data structure. While it might not be the most efficient at searching data, there are not many courses to search through, and the data will change so infrequently that it will be easy to maintain.