**Project One**

Stockton Toronto

Southern New Hampshire University

**Menu Pseudocode**

Main method for program, creates the user interface, menu():

Loop until user decides to end program:

Print menu options to console

Take user input, try to store it as an integer

If it isn’t an integer, prompt user to try again

Execute selected option based on input:

Case 1:

Load file, storing it in one of the designed data structures

Case 2:

Use data structure’s sort and printAll functions

Case 3:

Use data structure’s searching function to print specific item

Case 9:

Exit program by exiting loop

Default (if input is invalid):

Prompt user to try again

**Vector Pseudocode**

Creates a course object, constructor for course object, Course(num, name, prereq[]):

Course number equals num

Course name equals name

Prerequisites equals array of strings, prereq[]

Creates a vector from file, returns a vector<course>, coursesFromFile(file path):

Create empty vector<course> courses object

Open filestream using file path parameter

Loop while next line in filestream isn’t blank

Read next line and store to a string

Split string into an array of substrings using ‘,’ as a delimiter

If list of substrings is a size less than 2

Line is invalid, ignore input and continue loop

For each item in courses vector

If substring 1 is equal to course number

Repeated course num, line is invalid, ignore input and continue

For each index in substring array between 2 and size – 1 inclusive

Search courses vector for a course with an equivalent course Number

If no course has the substring as a course number is found

Line is invalid, ignore input and continue loop

Create new course object, Course(substring 1, substring 2, array of other strings)

Add course object to courses vector

Return courses vector

Prints out all courses course number and name, printAll(course&):

Loop through each item in vector:

Print course num and name

Prints out course information given course number, printCourse(courses&, num):

For each item in the courses vector

If course number equals num parameter

Print course number

Print course name

For each item in prerequisites[]

Print prerequisites

Return

Should loop exit without returning, print “course not found”

Sorts the courses into alphanumeric order, sort(course&, begin, end):

Set base case to return this function when there are 1 or zero items left to sort

Return if base case is true

Use the sortPartition function and obtain the last index of the lower partition

Recursively sort the list by calling sort on the lower partition

Call sort on the higher partition

Partions vector for use in sorting, sortPartition(course&, begin, end):

Set low equal to begin and high equal to end

Use high and low to calculate midpoint

Use the midpoint retrieve and store the pivot

Loop:

Loop through indices below pivot until a value greater than the pivot is found

Increment low index

Loop through indices above pivot until a value less than the pivot is found

Decrement high index

If the low index is greater than or equal to the high index

Exit loop, all courses have been partitioned

Else

Swap the values at the low and high indices

Increment low index

Decrement high index

Return high index, this value is now the final index in the lower partition

**HashTable Pseudocode**

Creates a hash key 1 from course num, Hash1(courseNum)

return courseNum modulo vector size

Creates a hash key 2 from course num, Hash2(courseNum)

Return constant minus (courseNum modulo constant)

Creates a course object, constructor for course object, Course(num, name, prereq[]):

Course number equals num

Course name equals name

Prerequisites equals array of strings, prereq[]

Prints out course information given course number, printCourse(courses&, num):

Loop until either course with matching num or spot that is “empty since start” is found

Find index using (Hash1(courseNum) + i \* Hash2(courseNum))

If course at index has num equal to the number passed to function

Print course number

Print course name

For each item in prerequisites[]

Print prerequisites

Return

Should loop exit without returning, print “course not found”

Creates a vector from file, returns a vector<course>, coursesFromFile(file path):

Create empty vector<course> courses object with a size 11

Open filestream using file path parameter

Loop while next line in filestream isn’t blank

Read next line and store to a string

Split string into an array of substrings using ‘,’ as a delimiter

If list of substrings is a size less than 2

Line is invalid, ignore input and continue loop

For each item at index (Hash1(courseNum) + i \* Hash2(courseNum)) until “empty since start”

If substring 1 is equal to course number

Repeated course num, line is invalid, ignore input and continue

For each index in substring array between 2 and size – 1 inclusive

Search courses vector for a course with an equivalent course Number

Loop until either course with matching num or spot that is “empty since start” is found

Find index using (Hash1(courseNum) + i \* Hash2(courseNum))

If course at index has num equal to the number in substring

Check next substring

If spot that is “empty since start” is found

Line is invalid, ignore input and do next course

Create new course object, Course(substring 1, substring 2, array of other strings)

Add course object to courses vector

Return courses vector

Adds course information to vector, addCourse(course)

Loop until an empty spot is found in the vector

index to place course equals (Hash1(courseNum) + i \* Hash2(courseNum)) modulo table size

if index is empty

insert course at index

if spot isn’t found within table size / 3 times,

resize vector to the next prime number after table size \* 2

increment i

creates an alphanumeric ordered list from hashtable and prints it out, printAll(courses&):

create new vector, sorted

Loop through hashtable:

Add each non-empty item to the sorted vector

Run the vector’s sort function passing the sorted vector

Run the vector’s printAll function passing the sorted vector

**BinarySearchTree Pseudocode**

Constructor for course object, Course(num, name, prereq[])

Set each member variable equal to relevant parameter

Create sorted BST from file, returns BinarySearchTree object, static BSTFromFile(file path)

Initialize BST with null root

Read file, for each line until one is empty:

Split line into substrings using ‘,’ as delimiter

If there is less than 2 substrings

Invalid line, move onto next line

Use Search function to check if substring 1 matches an existing node

If it does,

Invalid line, move onto next line

each substring past the 2nd are prerequisites

Search and check that all prerequisites exist in BST

If not,

Invalid line, move onto next line

Create new course object with substring 1 as Id, 2 as name, and 3+ as prereqs

Use AddNode function to add course to BST

Return courses BST

Add node to BST, AddNode(parentNode, course)

If BST is empty, set root to the new node containing course

If course number is less than parent node’s

If parent node’s left child is null, set left child equal to new node with course

Else, recursive call AddNode passing left child and course

Else,

If parent node’s right child is null, set right child equal to new node with course

Else, recursive call AddNode passing right child and course

Search BST for node with course num, returns course object, Search(courseNum)

Start at root

Loop while current node is not null

If current node is a match, return that node’s course

Else, if current node course number is greater, set current to left child

Else, set current to right child

Return empty course, as the loop ended without finding a match

Print course info from BST, PrintCourse(courseNum)

Use Search function, passing course num, to get the relevant course object

Print course number and title

Print each item in the course’s prereq[] array

Prints out BST in order from left to right, printAll(node\* initially the root):

If node isn’t null:

Call printAll on left child

Display current node’s info

Call printAll on right child

**Runtime Evaluations**

Creating vector:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file stream | 1 | 1 | 1 |
| Create courses vector | 1 | 1 | 1 |
| Loop until file is empty | 1 | N | N |
| Store line as string | 1 | N | N |
| Split line into substrings | 1 | N | N |
| If there is less than 2  substrings | 1 | N | N |
| Ignore line, continue  Upper loop | 1 | N | N |
| For each item in vector | 1 | N2 | N2 |
| If IDs are equal | 1 | N2 | N2 |
| Ignore line, continue  Upper loop | 1 | N2 | N2 |
| For each prerequisite | 1 | N | N |
| Search vector for  prerequisite ID | 3 | N2 | N2 |
| If prerequisite isn’t  found | 1 | N | N |
| Ignore line, continue  upper loop | 1 | N | N |
| Create new course object | 3 | N | N |
| Add course to vector | 1 | N | N |
| Return courses vector | 1 | 1 | 1 |
| **Total Cost** | | | 4N2+10N+3 |
| **Runtime** | | | O(N2) |

Creating hashtable:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file stream | 1 | 1 | 1 |
| Create courses vector | 1 | 1 | 1 |
| Loop until file is empty | 1 | N | N |
| Store line as string | 1 | N | N |
| Split line into substrings | 1 | N | N |
| If there is less than 2  substrings | 1 | N | N |
| Ignore line, continue  Upper loop | 1 | N | N |
| For each item in vector at  index calculated using hash  functions | 3 | N2 | N2 |
| If IDs are equal | 1 | N2 | N2 |
| Ignore line, continue  Upper loop | 1 | N2 | N2 |
| For each prerequisite | 1 | N | N |
| Search vector for  prerequisite ID | 3 | N2 | N2 |
| If prerequisite isn’t  Found before “empty since  start” | 3 | N | N |
| Ignore line, continue  upper loop | 1 | N | N |
| Create new course object | 3 | N | N |
| Add course to vector | 1 | N | N |
| Return courses vector | 1 | 1 | 1 |
| **Total Cost** | | | 4N2+10N+3 |
| **Runtime** | | | O(N2) |

Creating BinarySearchTree:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file stream | 1 | 1 | 1 |
| Initialize BST with root | 1 | 1 | 1 |
| Loop until file is empty | 1 | N | N |
| Store line as string | 1 | N | N |
| Split line into substrings | 1 | N | N |
| If there is less than 2  substrings | 1 | N | N |
| Ignore line, continue  Upper loop | 1 | N | N |
| Use search function to check  For repeated course number | 6 | Nlog(N) | Nlog(N) |
| Ignore repeated entry | 1 | N | N |
| For each prerequisite | 1 | N | N |
| If search for prerequisite  returns empty | 6 | Nlog(N) | Nlog(N) |
| Ignore entry | 1 | N | N |
| Create course object | 3 | N | N |
| Use add node to add course to  tree | 7 | Nlog(N) | Nlog(N) |
| Return BinarySearchTree | 1 | 1 | 1 |
| **Total Cost** | | | 3Nlog(N)+9N+3 |
| **Runtime** | | | O(Nlog(N)) |

**Advantages, Disadvantages and Recommendation**

The vector data structure is the most basic and malleable of the three structures designed here. The order of the elements in the vector has no bearing on functionality, meaning items in the vector can be easily reordered. Because of this, vectors only need to be sorted once as long as the desired order is the same. The downside of the vector is that it suffers heavily when searching for specific values, require any number of comparisons between one and the size of the vector. Also, Vectors cannot be resized directly, instead a new vector is created, and all elements are moved to the new one.

The hash table is definitely the more abstract structure and can be implemented in many different ways. In this paper, I implemented the hash table using my previously created vectors and made use of two hash functions. Hash tables are designed to mitigate the number of comparisons needed when searching for an element in the table. The order of elements in a hash table is intrinsically tied to its functionality, elements cannot be reordered without drastically damaging the structure. Moreover, the order of the elements is rarely useful outside of implementing the hash table itself. “Sorting” a hash table often means creating a second data structure to temporarily hold the table elements and then sorting that. This means that the sorted order needs to be recalculated every time a sorted list is desired, even though the desired order hasn’t changed.

The binary search tree is the fastest of these structures to create and search through. Like the hash table, the order of elements in a binary search tree is integral to its function and cannot easily be changed after creation; however, this order is alphanumeric, making the tree sorted upon its creation. The problem with the tree is that its efficiency is dependent on its shape. Without code to restructure the tree under certain condition, there will be a drastic decrease in efficiency.

Out of these three data structures, or at least my implementation of them, I would recommend using the binary search tree to store course information. It is faster to create and search, it is also sorted by design and doesn’t require its own sorting algorithm. That being said, an additional function may need to be created to ensure that the tree has a favorable node structure.