-.//Vector - Milestone 1

Struct Course(courseNumber, CourseName, preReq) {

courseNumber string

courseName string

preReq linked list

}

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**

}

bool validateDocument(String fileName) {

**open document filename**

**create temporary vector of courses**

**create bool variable valid and initialize to true**

**for all lines in the document**

**if there are less than two parameters on the line**

**set valid to false**

**exit method**

**for every prerequisite**

**if prerequisite is not in temporary vector of course**

**set valid to false**

**exit method**

**create a vector for the course data**

**add course data vector to vector of courses**

**move to the next line of the document**

}

Vector<Course> createCourseVector(String fileName){

**open document fileName**

**create an empty vector of courses**

**if the document is valid**

**while there is a next line in document**

**create a vector for the course data**

**add course data vector to vector of courses**

**move to the next line of the document**

**return the completed vector of courses**

}

void sortCourses(Vector<Course> courses){

int min

int i

int j

int size\_t = length of courses – 1;

for(i = 0 I < size\_t -1 i++){

min = i

for(j = i+1; j < size\_t; j++){

if (courses j is less than courses min) {

min = j

}

}

Swap courses I with courses j

}

}

Void printCourses(Vector<Course> courses){

For(I = 0 I < length of courses i++){

Output courses I course ID Course name.

While(prereq is not null){

Output prereq

}

}

}

//Hash Table - Milestone 2

Struct Course(courseNumber, CourseName, preReq) {

courseNumber string

courseName string

preReq linked list

}

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

**create hash key from courseNumber**

**get the node for the key**

**if there is an entry for the key and it matches**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

**else if the node has never been used**

**return**

**else**

**while there is a next node**

**if there is a match**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

**move to the next node**

}

bool validateDocument(String fileName) {

**open document filename**

**create temporary vector of courses**

**create bool variable valid and initialize to true**

**for all lines in the document**

**if there are less than two parameters on the line**

**set valid to false**

**exit method**

**for every prerequisite**

**if prerequisite is not in temporary vector of course**

**set valid to false**

**exit method**

**create a vector for the course data**

**add course data vector to vector of courses**

**move to the next line of the document**

}

void loadCourse(HashTable<Course> courses, String fileName){

**open document fileName**

**create an empty vector of courses**

**if the document is valid**

**while there is a next line in document**

**string courseID**

**String courseName**

**LinkedList preReq**

**courseID equals the next token before comma**

**courseName equals the next token before comma**

**If not at end of line**

**Add next token to preReq**

**Create course from courseID, courseName, preReq**

**Create key from courseID**

**Get node for the key**

**If there is no entry**

**Add a new node with the course**

**Else if the node is not used**

**Add node with the course**

**Else**

**While next node is not null**

**Move to the next node**

**Add node with the course**

}

void sortCourses(HashTable<Course> courses){

Vector<Course> SortedCourses

For(I = first node; I < last node; i++){

If(key is not empty){

Add course to SortedCourses

Move to the next node in the key

While(node is not null)

{

Add course to SortedCourses

Node = next node

}

}

}

int min

int i

int j

int size\_t = length of SortedCourses – 1;

for(i = 0 I < size\_t -1 i++){

min = i

for(j = i+1; j < size\_t; j++){

if (SortedCourses j is less than SortedCourses min) {

min = j

}

}

Swap SortedCourses I with SortedCourses j

}

}

Void printCourses(HashTable<Course> courses){

For(I = first node; I < last node; i++){

If(key is not empty){

Output courses I course ID Course name.

While(prereq is not null){

Output prereq

}

Move to the next node in the key

While(node is not null)

{

Output courses I course ID Course name.

While(prereq is not null){

Output prereq

}

Node = next node

}

}

}

}

//Binary Tree - Milestone 3

Struct Course(courseNumber, CourseName, preReq) {

courseNumber string

courseName string

preReq linked list

}

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

**Set currentNode to root node**

**While currentNode isn’t empty**

**If courseNumber matches the courseNumber from the currentNode**

**Return the courseNumber from the currentNode**

**If the courseNumber is smaller than the bid of the currentNode**

**Set currentNode to the left node**

**Else**

**Set currentNode to the right node**

**Create an empty courseNumber**

**Return empty courseNumber**

}

bool validateDocument(String fileName) {

**open document filename**

**create temporary vector of courses**

**create bool variable valid and initialize to true**

**for all lines in the document**

**if there are less than two parameters on the line**

**set valid to false**

**exit method**

**for every prerequisite**

**if prerequisite is not in temporary vector of course**

**set valid to false**

**exit method**

**create a vector for the course data**

**add course data vector to vector of courses**

**move to the next line of the document**

}

void loadCourse(Tree<Course> courses, String fileName){

**open document fileName**

**create an empty vector of courses**

**if the document is valid**

**while there is a next line in document**

**string courseID**

**String courseName**

**LinkedList preReq**

**courseID equals the next token before comma**

**courseName equals the next token before comma**

**If not at end of line**

**Add next token to preReq**

**Create course from courseID, courseName, preReq**

**If root of courses is empty**

**Set root to new node with course**

**Else**

**addCourse with root and course**

}

void addCourse(Node\* node, Course course){

if node is larger than the course

if there is no left node

add course to the left node

else

call addNode with left node and course

else

if there is no right node

add course to the right node

else

call addNode with right node and course

**}**

void sortCourses(Tree<Course> courses){

\*\*\*Binary Tree is already sorted\*\*

}

Void printCourses(Tree<Course> courses){

inOrder(root)

}

Void inOrder(Node\* node){

if(node != null){

inOrder(left node)

Output courses I course ID Course name.

While(prereq is not null){

Output prereq

}

inOrder(right node)

}

}

**MENU**

**<Vector> OR <HashTable> OR <Tree> courses**

**While(option choosen is not 9){**

**Output “Option 1: Load Course Data.”**

**Output “Option 2: Print Ordered List.”**

**Output “Option 3: Print Course Data.”**

**Output “Option 9: Exit Program”**

**Switch Case:**

**Case 1:**

**loadCourse(courses, filename)**

**Case 2:**

**Sort(courses)**

**Print(courses)**

**Case 3:**

**courseNumber = Input**

**searchCourse(courses, courseNumber)**

**CreateCourseVector Run Time Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open document fileName** | 1 | 1 | 1 |
| **create an empty vector of courses** | 1 | 1 | 1 |
| **if the document is valid** | 1 | n | n |
| **while there is a next line in document** | 1 | n | n |
| **create a vector for the course data** | 1 | n | n |
| **add course data vector to vector of courses** | 1 | 1 | 1 |
| **move to the next line of the document** | 1 | 1 | 1 |
| **return the completed vector of courses** | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 5 |
| **Runtime** | | | O(n) |

**HashTable Run Time Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open document fileName** | 1 | 1 | 1 |
| **create an empty vector of courses** | 1 | 1 | 1 |
| **if the document is valid** | 1 | n | n |
| **while there is a next line in document** | 1 | n | n |
| **String course ID** | 1 | 1 | 1 |
| **String Course Name** | 1 | 1 | 1 |
| **LinkedList preReq** | 1 | 1 | 1 |
| **CourseID equals next token** | 1 | 1 | 1 |
| **courseName equals** | 1 | 1 | 1 |
| **If not at the end of line** | 1 | N | N |
| **Create course** | 1 | 1 | 1 |
| **Create Key** | 1 | 1 | 1 |
| **If there is no entry** | 1 | 1 | 1 |
| **Else if node is not used** | 1 | 1 | 1 |
| **Else while next note** | 1 | N | N |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | 4n + 11 |
| **Runtime** | | | O(n) |

Binary Tree Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open document fileName** | 1 | 1 | 1 |
| **create an empty vector of courses** | 1 | 1 | 1 |
| **if the document is valid** | 1 | n | n |
| **while there is a next line in document** | 1 | n | n |
| **String course ID** | 1 | 1 | 1 |
| **String Course Name** | 1 | 1 | 1 |
| **LinkedList preReq** | 1 | 1 | 1 |
| **CourseID equals next token** | 1 | 1 | 1 |
| **courseName equals** | 1 | 1 | 1 |
| **If not at the end of line** | 1 | N | N |
| **Create course** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **AddCourse** | 1 | N | N |
| **Total Cost** | | | 4n + 9 |
| **Runtime** | | | O(n) |

Analysis

For all three data structures, vector, hash table, and binary tree, there are advantages and disadvantages. Vectors are simple and straightforward. They are easy to implement and access data from. If we are accessing data sequentially, they can be very fast. Adding to the end of the vector is also very quick. However, if we are searching, it can be inefficient, possibly O(n).

For hash tables, some benefits are fast lookups, insertion, and deletions, as once we generate a key, we can typically go right to the bucket that stores the item. We don’t have to traverse as many items as only if the bucket has existing items would we have to start traversing. It excels at tasks where quick retrieval is important. On the downside, a lot of these benefits are lost if there is a high collision rate. If all classes end up in the same bucket, it would effectively be a vector. We can end up with a worst-case time complexity of O(N).

For binary search trees, we are storing the items in a sorted order already; this is helpful when we want to print all items in order. We get fast lookup, insertion, and deletion in a well-balanced tree, O(logN). However, if the tree is unbalanced, we come back to O(N) as a worst-case scenario. It is the most complicated to implement of the three. If we wanted to maintain performance we may need to rebalance the tree.

Given the project I think binary search trees would be a good fit. For outputting an alphanumerically ordered list of all the courses it would be easy to implement as the tree will already be sorted. Looking up courses would be fairly quick as well.