// definition of a course

struct Course {

string number,

string name,

vector<string> prerequisites

};

// a string representation of the menu options

Const char \*MENU { “Menu:\n\t1. Load Courses\n\t2. Print Courses in Order\n\t3. Find and Print Course\n\t9. Exit” };

int main(int argc, char\* argv[]) {

std::string = argc > 0 ? argv[0] : “path/to/default/file”

std::vector<Courses> data;

int choice = 0;

if (ValidateFile(csv\_path) < 0) {

std::cout << “Could not load data from file: ” << csv\_path << std::endl;

return -1;

}

while (choice != 9) {

std::cout << MENU << std::endl;

std::cin >> choice;

switch (choice) {

case 1:

data = LoadCourses(csv\_path, &data);

break;  
 case 2:

QuickSort(data, 0, data.size() - 1);

for (Course course : data)

PrintCourse(course);

break;

case 3:

std::string course\_id;

std::cout << “Course ID: ”;

cin >> course\_id;

Course course = data.find(course\_id);

std::cout << “Title: ” << course.title << “\nPrerequisites: \n”;

for (std::string prerequisite : data.prerequisites)

std::cout << ‘\t’ << prerequisite << std::endl;

break;

default:

std::cout << “Menu option unknown, please select a valid menu option” << std::endl;

break;

}

}

}

// returns 0 for valid file, -1 for invalid file

int ValidateFile(String file\_path) {

File file = File(file\_path) // allocate memory

Set<String> prerequisites // storage for prerequisite courses

Set<String> courses // storage for course numbers

// parse each row

for String row in file {

int count = 0

int start = 0

int end = row.find(‘,’)

if end == 0:

Print “Rows should not begin with a comma”

return -1

// while end exists

while end != -1 {

String found = row.substr(start, end)

++count

if count == 1:

courses.insert(found)

else if count > 2:

prerequisites.insert(found)

// move indices forward

start = end + 1

// if there is more data following the previous comma

if start < row.length():

row = row.substr(start, row.length())

end = row.find(‘,’)

else:

Print “Rows should not end on a comma”

return -1

}

// the last value in the row

// value must exist since we know the row did not end on a comma, but a comma wasn’t found

String found = row.substr(start, row.length())

++count

if count > 2:

prerequisites.insert(found)

else if count < 2:

Print “There must be at minimum a course number and name, but only {count} values were found”

return -1

}

// make sure all prerequisites have a corresponding data entry

for (String prerequisite in prerequisites) {

if courses does not contain prerequisite:

Print “No entry found for listed prerequisite: {prerequisite}”

return -1

}

return 0

}

// Function should not be used before validating csv file

DataStruct<Courses> LoadCourses(std::string data\_path) {

File file = File(data\_path) // allocate memory

DataStruct<Course> courses // will eventually be returned

for (std::string row : file) {

// get course number

int start = 0

int end = row.find(‘,’)

String course\_number = row.substr(start, end)

// get course name

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

String course\_name = row.substr(start, end)

// get all prerequisites

Vector<String> prerequisites // storage for prerequisites

start = end + 1

// if no prerequisites: add course and continue

if start > row.length() {

Course course { course\_number, course\_name, prerequisites };

courses.append(course)

continue

}

row = row.substr(start, row.length())

end = row.find(‘,’)

while end != -1 {

String prerequisite = row.substr(start, end)

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

prerequisites.append(prerequisite)

}

// the last prerequisite (rows cannot end on commas)

prerequisites.append(row.substr(start, row.length())

Course course { course\_number, course\_name, prerequisites }

// using .add here because data structure is unknown

courses.add(course)

return courses;

}

Partition(std::vector<Course> courses, lowIndex, highIndex) {

// Pick middle element as pivot

midpoint = lowIndex + (highIndex - lowIndex) / 2

pivot = courses.at(midpoint).id

done = false

while (!done) {

// Increment lowIndex while courses.at(lowIndex) < pivot

while (courses.at(lowIndex).compare(pivot) < 0) {

lowIndex += 1

}

// Decrement highIndex while pivot < courses.at(highIndex)

while (pivot.compare(courses.at(highIndex).id) < 0) {

highIndex -= 1

}

// If zero or one elements remain, then all numbers are partitioned. retun highIndex

if (lowIndex >= highIndex) {

done = true

} else {

// Swap low and high

temp = courses.at(lowIndex)

courses.at(lowIndex) = courses.at(highIndex)

courses.at(highIndex) = temp

// Update lowIndex and highIndex

lowIndex += 1

highIndex -= 1

}

}

return highIndex

}

Quicksort(std::vector<Courses> courses, lowIndex, highIndex) {

// Base case: if the partition size is 1 or zero elements, then the partition is already sorted

if (lowIndex >= highIndex) {

return

}

// Partition the data within the array. Value lowEndIndex returned from partitioning is the index of hte low partition's last element.

lowEndIndex = Partition(courses, lowIndex, highIndex)

// Recursively sort low partition (lowIndex to lowEndIndex)

// and high partition (lowEndIndex + 1 to highIndex)

Quicksort(courses, lowIndex, lowEndIndex)

Quicksort(courses, lowEndIndex + 1, highIndex)

}

void PrintCourse(Course course) {

int SIZE = 3

string descriptors[SIZE] {

“Number: ”,

“Name: ”,

“Prerequisites: ”

};

string prerequisites;

// if there are prerequisites

if (course.prerequisites.size() != 0) {

for (string prerequisite : course.prerequisites) {

// if this is the last one, don’t add a comma and leave loop

if (prerequisite == course.prerequisites.back()) {

prerequisites += prerequisite

break

}

// not the last one, add comma and keep going

prerequisites += prerequisite + “, ”

}

} else {

prerequisites = “None”

}

string vals[SIZE] {

course.number,

course.name,

prerequisites

};

for (int index = 0; index < SIZE; ++index) {

cout << descriptors[index] << vals[index] << endl;

}

}

// returns 0 for valid file, -1 for invalid file

int ValidateFile(String file\_path) {

File file = File(file\_path) // allocate memory

Set<String> prerequisites // storage for prerequisite courses

Set<String> courses // storage for course numbers

// parse each row

for String row in file {

int count = 0

int start = 0

int end = row.find(‘,’)

if end == 0:

Print “Rows should not begin with a comma”

return -1

// while end exists

while end != -1 {

String found = row.substr(start, end)

++count

if count == 1:

courses.insert(found)

else if count > 2:

prerequisites.insert(found)

// move indices forward

start = end + 1

// if there is more data following the previous comma

if start < row.length():

row = row.substr(start, row.length())

end = row.find(‘,’)

else:

Print “Rows should not end on a comma”

return -1

}

// the last value in the row

// value must exist since we know the row did not end on a comma, but a comma wasn’t found

String found = row.substr(start, row.length())

++count

if count > 2:

prerequisites.insert(found)

else if count < 2:

Print “There must be at minimum a course number and name, but only {count} values were found”

return -1

}

// make sure all prerequisites have a corresponding data entry

for (String prerequisite in prerequisites) {

if courses does not contain prerequisite:

Print “No entry found for listed prerequisite: {prerequisite}”

return -1

}

return 0

}

|  |  |  |  |
| --- | --- | --- | --- |
| **ValidateFile** | **Line Cost** | **# Times Executes** | **Total Cost** |
| File file = File(file\_path) | 1 | 1 | 1 |
| Set<String> prerequisites | 1 | 1 | 1 |
| Set<String> courses | 1 | 1 | 1 |
| for String row in file | 1 | N | 1 |
| int count = 0 | 1 | N | 1 |
| int start = 0 | 1 | N | 1 |
| if end == 0: | 1 | N | 1 |
| Print “Rows should not begin with a comma” | 1 | 1 | 1 |
| return -1 | 1 | 1 | 1 |
| while end != -1 { | 1 | N | 1 |
| String found = row.substr(start, end) | 1 | N | 1 |
| ++count | 1 | N | 1 |
| if count == 1: | 1 | N | 1 |
| courses.insert(found) | 1 | N | 1 |
| else if count > 2: | 1 | N | 1 |
| prerequisites.insert(found) | 1 | N | 1 |
| start = end + 1 | 1 | N | 1 |
| if start < row.length(): | 1 | N | 1 |
| row = row.substr(start, row.length()) | 1 | N | 1 |
| end = row.find(‘,’) | 1 | N | 1 |
| else: | 1 | N | 1 |
| Print “Rows should not end on a comma” | 1 | 1 | 1 |
| return -1 | 1 | 1 | 1 |
| String found = row.substr(start, row.length()) | 1 | N | 1 |
| ++count | 1 | N | 1 |
| if count > 2: | 1 | N | 1 |
| prerequisites.insert(found) | 1 | N | 1 |
| else if count < 2: | 1 | N | 1 |
| Print “There must be at minimum a course number and name, but only {count} values were found” | 1 | 1 | 1 |
| return -1 | 1 | 1 | 1 |
| for (String prerequisite in prerequisites) { | 1 | N | 1 |
| if courses does not contain prerequisite: | 1 | N | 1 |
| Print “No entry found for listed prerequisite: {prerequisite}” | 1 | 1 | 1 |
| return -1 | 1 | 1 | 1 |
| return 0 | 1 | 1 | 1 |
| **Total Cost:** | | | 23N + 12 |
| **Runtime:** | | | O(N) |

// definition of a course

struct Course {

string number,

string name,

vector<string> prerequisites

};

// print course information and prerequisites

void PrintCourse(Course course) {

int SIZE = 3

string descriptors[SIZE] {“Number: ”, “Name: ”, “Prerequisites: ”};

string prerequisites;

// if there are prerequisites

if (course.prerequisites.size() != 0) {

for (string prerequisite : course.prerequisites) {

// if this is the last one, don’t add a comma and leave loop

if (prerequisite == course.prerequisites.back()) {

prerequisites += prerequisite

break

}

// not the last one, add comma and keep going

prerequisites += prerequisite + “, ”

}

} else {

prerequisites = “None”

}

string vals[SIZE] {course.number, course.name, prerequisites};

for (int index = 0; index < SIZE; ++index) {

cout << descriptors[index] << vals[index] << endl;

}

}

|  |  |  |  |
| --- | --- | --- | --- |
| **PrintCourse** | **Line Cost** | **# Times Executes** | **Total Cost** |
| int SIZE = 3 | 1 | 1 | 1 |
| string descriptors[SIZE] {“Number: ”, “Name: ”, “Prerequisites: ”}; | 1 | 1 | 1 |
| string prerequisites; | 1 | 1 | 1 |
| if (course.prerequisites.size() != 0) { | 1 | 1 | 1 |
| for (string prerequisite : course.prerequisites) { | 1 | N | N |
| if (prerequisite == course.prerequisites.back()) { | 1 | N | N |
| prerequisites += prerequisite | 1 | N | N |
| break | 1 | N | N |
| prerequisites += prerequisite + “, ” | 1 | N | N |
| } else { | 1 | N | N |
| prerequisites = “None” | 1 | N | N |
| string vals[SIZE] {course.number, course.name, prerequisites}; | 1 | 1 | 1 |
| for (int index = 0; index < SIZE; ++index) { | 1 | N | N |

# Vector

// because psuedocode exists for validating the file, there will not be any error handling in this function

// returns all courses found in .csv file as a vector of Courses

Vector<Course> loadCourses(String file\_path) {

File file = File(file\_path) // allocate memory

Vector<Course> courses // will eventually be returned

for String row in file {

// get course number

int start = 0

int end = row.find(‘,’)

String course\_number = row.substr(start, end)

// get course name

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

String course\_name = row.substr(start, end)

// get all prerequisites

Vector<String> prerequisites // storage for prerequisites

start = end + 1

// if no prerequisites: add course and continue

if start > row.length():

Course course {course\_number, course\_name, prerequisites }

courses.append(course)

continue

row = row.substr(start, row.length())

end = row.find(‘,’)

while end != -1 {

String prerequisite = row.substr(start, end)

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

prerequisites.append(prerequisite)

}

// the last prerequisite (rows cannot end on commas)

prerequisites.append(row.substr(start, row.length())

Course course {course\_number,course\_name,prerequisites}

courses.append(course)

}

return courses;

}

|  |  |  |  |
| --- | --- | --- | --- |
| **loadCourses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| File file = File(file\_path) | 1 | 1 | 1 |
| Vector<Course> courses | 1 | 1 | 1 |
| for String row in file | 1 | N | N |
| int start = 0 | 1 | N | N |
| int end = row.find(‘,’) | 1 | N | N |
| String course\_number = row.substr(start, end) | 1 | N | N |
| start = end + 1 | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| String course\_name = row.substr(start, end) | 1 | N | N |
| Vector<String> prerequisites // storage for prerequisites | 1 | N | N |
| start = end + 1 | 1 | N | N |
| if start > row.length(): | 1 | N | N |
| Course course {course\_number, course\_name, prerequisites } | 1 | N | N |
| courses.append(course) | 1 | N | N |
| continue | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| while end != -1 { | 1 | N | N |
| String prerequisite = row.substr(start, end) | 1 | N | N |
| start = end + 1 | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| prerequisites.append(prerequisite) | 1 | N | N |
| prerequisites.append(row.substr(start, row.length()) | 1 | N | N |
| Course course {course\_number,course\_name,prerequisites} | 1 | N | N |
| courses.append(course) | N | N | N^2 |
| return courses; | 1 | N | N |
| **Total Cost:** | | | N^2 + 26N + 2 |
| **Runtime:** | | | O(N^2) |

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

for Course course in courses:

if course.number == courseNumber:

PrintCourse(course);

return

}

|  |  |  |  |
| --- | --- | --- | --- |
| **searchCourse** | **Line Cost** | **# Times Executes** | **Total Cost** |
| for Course course in courses: | 1 | N | N |
| if course.number == courseNumber: | 1 | N | N |
| Print(course) | N | 1 | N |
| **Total Cost:** | | | 3N |
| **Runtime:** | | | O(N) |

# Hash Table

// because psuedocode exists for validating the file, there will not be any error handling in this function

// returns all courses found in .csv file as a HashTable of Courses

HashTable<Course> loadCourses(String file\_path) {

File file = File(file\_path) // allocate memory

HashTable<Course> courses // will eventually be returned

for String row in file {

// get course number

int start = 0

int end = row.find(‘,’)

String course\_number = row.substr(start, end)

// get course name

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

String course\_name = row.substr(start, end)

// get all prerequisites

Vector<String> prerequisites // storage for prerequisites

start = end + 1

// if no prerequisites: add course and continue

if start > row.length():

Course course {course\_number, course\_name, prerequisites}

courses.insert(course)

continue

row = row.substr(start, row.length())

end = row.find(‘,’)

while end != -1 {

String prerequisite = row.substr(start, end)

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

prerequisites.append(prerequisite)

}

// the last prerequisite (rows cannot end on commas)

prerequisites.append(row.substr(start, row.length())

Course course {course\_number, course\_name, prerequisites}

courses.insert(course)

}

return courses;

}

|  |  |  |  |
| --- | --- | --- | --- |
| **loadCourses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| File file = File(file\_path); | 1 | 1 | 1 |
| HashTable<Course> courses; | 1 | 1 | 1 |
| for String row in file { | 1 | N | N |
| int start = 0 | 1 | N | N |
| int end = row.find(‘,’) | 1 | N | N |
| String course\_number = row.substr(start, end) | 1 | N | N |
| start = end + 1 | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| String course\_name = row.substr(start, end) | 1 | N | N |
| Vector<String> prerequisites | 1 | N | N |
| start = end + 1 | 1 | N | N |
| if start > row.length(): | 1 | N | N |
| Course course {course\_number, course\_name, prerequisites} | 1 | N | N |
| courses.Insert(course) | 1 | N | N |
| continue | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| while end != -1 { | 1 | N | N |
| String prerequisite = row.substr(start, end) | 1 | N | N |
| start = end + 1 | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| prerequisites.append(prerequisite) | 1 | N | N |
| prerequisites.append(row.substr(start, row.length()) | 1 | N | N |
| Course course {course\_number,course\_name,prerequisites} | 1 | N | N |
| courses.insert(course) | 1 | N | N |
| return courses; | 1 | 1 | 1 |
| **Total Cost:** | | | 25N + 3 |
| **Runtime:** | | | O(N) |

# Binary Tree

// public definition of a BinarySearchTree

class BinarySearchTree {

void InOrder();

void PostOrder();

void PreOrder();

void Insert(Bid bid);

void Remove(string bidId);

Bid Search(string bidId);

};

// because psuedocode exists for validating the file, there will not be any error handling in this function

// returns all courses found in .csv file as a HashTable of Courses

BinarySearchTree loadCourses(String file\_path) {

File file = File(file\_path); // allocate memory

BinarySearchTree courses; // will eventually be returned

for String row in file {

// get course number

int start = 0

int end = row.find(‘,’)

String course\_number = row.substr(start, end)

// get course name

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

String course\_name = row.substr(start, end)

// get all prerequisites

Vector<String> prerequisites // storage for prerequisites

start = end + 1

// if no prerequisites: add course and continue

if start > row.length():

Course course {course\_number, course\_name, prerequisites}

courses.Insert(course)

continue

row = row.substr(start, row.length())

end = row.find(‘,’)

while end != -1 {

String prerequisite = row.substr(start, end)

start = end + 1

row = row.substr(start, row.length())

end = row.find(‘,’)

prerequisites.append(prerequisite)

}

// the last prerequisite (rows cannot end on commas)

prerequisites.append(row.substr(start, row.length())

Course course {course\_number,course\_name,prerequisites}

courses.Insert(course)

}

return courses;

}

|  |  |  |  |
| --- | --- | --- | --- |
| **loadCourses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| File file = File(file\_path); | 1 | 1 | 1 |
| BinarySearchTree courses; | 1 | 1 | 1 |
| for String row in file { | 1 | N | N |
| int start = 0 | 1 | N | N |
| int end = row.find(‘,’) | 1 | N | N |
| String course\_number = row.substr(start, end) | 1 | N | N |
| start = end + 1 | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| String course\_name = row.substr(start, end) | 1 | N | N |
| Vector<String> prerequisites | 1 | N | N |
| start = end + 1 | 1 | N | N |
| if start > row.length(): | 1 | N | N |
| Course course {course\_number, course\_name, prerequisites} | 1 | N | N |
| courses.Insert(course) | 1 | N | N |
| continue | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| while end != -1 { | 1 | N | N |
| String prerequisite = row.substr(start, end) | 1 | N | N |
| start = end + 1 | 1 | N | N |
| row = row.substr(start, row.length()) | 1 | N | N |
| end = row.find(‘,’) | 1 | N | N |
| prerequisites.append(prerequisite) | 1 | N | N |
| prerequisites.append(row.substr(start, row.length()) | 1 | N | N |
| Course course {course\_number,course\_name,prerequisites} | 1 | N | N |
| courses.Insert(course) | Log n | N | Nlog(N) |
| return courses; | 1 | 1 | 1 |
| **Total Cost:** | | | Nlog(N) + 25N + 3 |
| **Runtime:** | | | O(N) |

# Data Structure Examination

There are pros and cons to every data structure. Since data commonly has the four Create, Read, Update, and Delete operations, a data structure is generally chosen based on the efficiency of the most used operations for the given data structure. Of these operations, Create (.append/.insert) is the only one that is used within this psuedocode, so I’ll be focusing on that.

The Create operation:

* Because a single vector append operation has linear complexity, appending N elements leads to O(N^2) complexity.
* Because a single hash table insert operation has constant complexity, inserting N elements leads to O(N) complexity.
* While a single binary tree insert operation has log(N) complexity, inserting N elements still leads to O(N) complexity due to the other calls made during loadCourses().

In my opinion, a has table would be best. By implementing the courses as a hash table, constant time insertion can be guaranteed (zero conflicts) since the number of courses offered by the institution should be known well in advance. While a binary tree is also efficient, it creates a complex graph of nodes which may be more difficult to maintain than a simple hash table. If a hash table is chosen in the end, consider writing an administrator tool to re-hash all of the course IDs for the eventuality of courses being added/removed to/from the curriculum.