Patrick D. Pattison

CS-300: Data Structures and Algorithms

Southern New Hampshire University

5 December 2021

**Issue 1:**

**Reading and parsing course file; vector data structure**

function splitLine(string line, string delimiter) returns Vector<string>

Vector<string> return\_value

std::stringstream stream

stream = std::stringstream(line)

std::string token

while std::getline(stream, token, delimiter)

return\_value.push\_back(token)

function parseCourses(string filename) returns Vector<Course>

Vector<Course> return\_value

std::ifstream input\_file

input\_file = std::ifstream(filename)

if input\_file.is\_open()

std::string input\_line

**while std::getline(input\_file, input\_line)**

Vector<std::string> tokens

**tokens = splitLine(input\_line)**

// A valid course entry should have two or more fields

if tokens.size() >= 2

// tokens[0] is the current course ID

// tokens[1] is the current course title

// If tokens contains more than 2 elements, all additional

// elements are prerequisite course IDs

Course new\_course

new\_course.courseId = tokens[0]

new\_course.courseTitle = tokens[1]

for integer index = 2; index < tokens.size(); index = index + 1

new\_course.prerequisites.push\_back(tokens[index])

**Reading and parsing course file; Hash Table data structure**

function CourseTable::splitLine(string line, string delimiter = “ “) returns Vector<string>

Vector<string> return\_value

std::stringstream stream

stream = std::stringstream(line)

std::string token

while std::getline(stream, token, delimiter)

return\_value.push\_back(token)

return return\_value

function CourseTable::populate(std::string filename) returns unsigned integer

std::ifstream input\_file

input\_file = std::ifstream(filename)

unsigned integer count = 0

if input\_file.is\_open()

std::string input\_line

while std::getline(input\_file, input\_line)

Vector<std::string> tokens

tokens = this->splitLine(input\_line)

// A valid course entry should have two or more fields

if tokens.size() >= 2

// tokens[0] is the current course ID

// tokens[1] is the current course title

// If tokens contains more than 2 elements, all additional

// elements are prerequisite course IDs

Course new\_course

new\_course.courseId = tokens[0]

new\_course.courseTitle = tokens[1]

for integer index = 2; index < tokens.size(); index = index + 1

new\_course.prerequisites.push\_back(tokens[index])

this->insert(new\_course)

count = count + 1

return count

**Reading and parsing course file; Binary Search Tree data structure**

function CourseTree::splitLine(string line, string delimiter = “ “) returns Vector<string>

Vector<string> return\_value

std::stringstream stream

stream = std::stringstream(line)

std::string token

while std::getline(stream, token, delimiter)

return\_value.push\_back(token)

return return\_value

function CourseTree::populate(std::string filename) returns unsigned integer

std::ifstream input\_file

input\_file = std::ifstream(filename)

unsigned integer count = 0

if input\_file.is\_open()

std::string input\_line

while std::getline(input\_file, input\_line)

Vector<std::string> tokens

tokens = this->splitLine(input\_line)

// A valid course entry should have two or more fields

if tokens.size() >= 2

// tokens[0] is the current course ID

// tokens[1] is the current course title

// If tokens contains more than 2 elements, all additional

// elements are prerequisite course IDs

Course new\_course

new\_course.courseId = tokens[0]

new\_course.courseTitle = tokens[1]

for integer index = 2; index < tokens.size(); index = index + 1

new\_course.prerequisites.push\_back(tokens[index])

this->insert(new\_course)

**Issue 2**

**Menu Pseudocode**

String input\_filename

Parse input\_filename from parameters passed to executable

Boolean user\_quit = false

//

// Variable course\_list will be of the appropriate type.

ApproriateType course\_list

While ! user\_quit

Put “Main Menu” to output

Put “ 1. Load Data” to output

Put “ 2. Print Course List” to output

Put “ 3. Print a Course” to output

Put “ Q. Quit” to output

Get user\_choice from input

If user\_choice == “1”

Course\_list.populate(file\_name)

Else if user\_choice == “2”

Course\_list.printAll()

Else if user\_choice == “3”

Put “Enter course ID to print: “ to output

Get course\_id from input

Course\_list.print(course\_id)

Else if user\_choice == “Q”

User\_quit = true

**Issue 3**

**Print all; vector data structure**

**NOTE: The vector is unsorted so a comparison function is required to use the std::sort function**

Function lessThan(Course left\_hand, Course right\_hand) returns boolean

Return left\_hand.courseId < right\_hand.courseId

Function printAll(vector<Course> courses)

Std::sort(courses.begin(), courses.end(), &lessThan)

For int index = 0; index < courses.size(); index = index + 1

Put “Course ID: “ + courses.at(index).courseId to output

Put “Course Title: “ + courses.at(index)..courseTitle to output

If courses.at(index).prerequisites.size() > 0

Put “Prerequisites:” to output

For int pre\_index = 0; pre\_index < courses.at(index).prerequisites.size(); pre\_index = pre\_index + 1

Put courses.at(index).prerequisites.at(pre\_index)

// The following line is to put a blank line between courses.

Put “” to output

**Print all; Hash Table data structure**

**NOTE: Hash Tables are by default un-sortable, so in order to print in sorted order we will first extract a vector<Course> of all courses, sort that list, and then iterate through the sorted list printing out the courses in order.**

Function CourseTable::print(Course course)

Put “Course ID: “ + course.courseId to output

Put “Course Title: “ + course.courseTitle to output

If course.prerequisites.size() > 0

Put “Prerequisites:” to output

For integer index = 0; index < course.prerequisites.size(); index = index + 1

Put course.prerequisites.at(index) to output

Function lessThan(Course left\_hand, Course right\_hand) returns Boolean

Return left\_hand.courseId < right\_hand.courseId

Function CourseTable::printAll()

Vector<Course> sorted\_courses

Node\* node

For integer index = 0; index < this->courses.size(); index = index + 1

Node = &(this->courses.at(index))

While ((node != nullptr) && (node->key != UINT\_MAX))

Sorted\_courses.append(node->course)

node = node->nextNode

std::sort(courseIds.being(), courseIds.end(), &lessThan)

for integer index = 0; index < sorted\_courses.size(); index = index + 1

this->print(sorted\_courses.at(index))

**Print all; BST data structure**

function CourseTree::printAll(Node\* node = root)

if node != nullptr

printAll(node->left)

put node->course.courseId + “: “ + node->course.courseTitle to output

if node->course.prerequisites.size() > 0

Put “Prerequisites:” to output

For integer index = 0; index < node->course.prerequisites.size(); index = index + 1

Put node->prerequisites.at(index) to output

printAll(node->right)

**Issue 4**

**Evaluation Reading and Parsing; vector data structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| **function splitLine(string line, string delimiter) returns Vector<string>** | **N/A** | **N/A** | **N/A** |
| Vector<string> return\_value | 1 | 1 | 1 |
| std::stringstream stream | 1 | 1 | 1 |
| stream = std::stringstream(line) | 1 | 1 | 1 |
| std::string token | 1 | 1 | 1 |
| while std::getline(stream, token, delimiter) | 1 | n | n |
| return\_value.push\_back(token) | 1 | n | n |
|  |  |  |  |
| **function parseCourses(string filename) returns Vector<Course>** | **N/A** | **N/A** | **N/A** |
| Vector<Course> return\_value | 1 | 1 | 1 |
| std::ifstream input\_file | 1 | 1 | 1 |
| input\_file = std::ifstream(filename) | 1 | 1 | 1 |
| if input\_file.is\_open() | 1 | 1 | 1 |
| std::string input\_line | 1 | 1 | 1 |
| while std::getline(input\_file, input\_line) | 1 | n | n |
| Vector<std::string> tokens | 1 | 1 | 1 |
| tokens = splitLine(input\_line) | 1 | n | n |
| if tokens.size() >= 2 | 1 | 1 | 1 |
| Course new\_course | 1 | 1 | 1 |
| new\_course.courseId = tokens[0] | 1 | 1 | 1 |
| new\_course.courseTitle = tokens[1] | 1 | 1 | 1 |
| for integer index = 2; index < tokens.size(); index = index + 1 | 1 | n | n |
| new\_course.prerequisites.push\_back(tokens[index]) | 1 | n | n |
| **Total Cost** | | | 6n+14 |
| **Runtime** | | | O(n) |

**Evaluation Reading and Parsing; Hash Table data structure**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| **function CourseTable::splitLine(string line, string delimiter = “ “) returns Vector<string>** | **N/A** | **N/A** | **N/A** |
| Vector<string> return\_value | 1 | 1 | 1 |
| std::stringstream stream | 1 | 1 | 1 |
| stream = std::stringstream(line) | 1 | 1 | 1 |
| std::string token | 1 | 1 | 1 |
| while std::getline(stream, token, delimiter) | 1 | n | n |
| return\_value.push\_back(token) | 1 | n | n |
| return return\_value | 1 | 1 | 1 |
|  |  |  |  |
| **function CourseTable::populate(std::string filename) returns unsigned integer** | **N/A** | **N/A** | **N/A** |
| std::ifstream input\_file | 1 | 1 | 1 |
| input\_file = std::ifstream(filename) | 1 | 1 | 1 |
| unsigned integer count = 0 | 1 | 1 | 1 |
| if input\_file.is\_open() | 1 | 1 | 1 |
| std::string input\_line | 1 | 1 | 1 |
| while std::getline(input\_file, input\_line) | 1 | n | n |
| Vector<std::string> tokens | 1 | n | n |
| tokens = this->splitLine(input\_line) | 1 | n | n |
| if tokens.size() >= 2 | 1 | n | n |
| Course new\_course | 1 | n | n |
| new\_course.courseId = tokens[0] | 1 | n | n |
| new\_course.courseTitle = tokens[1] | 1 | n | n |
| for integer index = 2; index < tokens.size(); index = index + 1 | 1 | n | n |
| new\_course.prerequisites.push\_back(tokens[index]) | 1 | n | n |
| this->insert(new\_course) | 1 | n | n |
| count = count + 1 | 1 | n | n |
| return count | 1 | 1 | 1 |
| **Total Cost** | | | 13n + 11 |
| **Runtime** | | | O(n) |

**Evaluation Reading and Parsing; BST data structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| **function CourseTree::splitLine(string line, string delimiter = “ “) returns Vector<string>** | **N/A** | **N/A** | **N/A** |
| Vector<string> return\_value | 1 | 1 | 1 |
| std::stringstream stream | 1 | 1 | 1 |
| stream = std::stringstream(line) | 1 | 1 | 1 |
| std::string token | 1 | 1 | 1 |
| while std::getline(stream, token, delimiter) | 1 | n | n |
| return\_value.push\_back(token) | 1 | n | n |
| return return\_value | 1 | 1 | 1 |
|  |  |  |  |
| **function CourseTree::populate(std::string filename) returns unsigned integer** | **N/A** | **N/A** | **N/A** |
| std::ifstream input\_file | 1 | 1 | 1 |
| input\_file = std::ifstream(filename) | 1 | 1 | 1 |
| unsigned integer count = 0 | 1 | 1 | 1 |
| if input\_file.is\_open() | 1 | 1 | 1 |
| std::string input\_line | 1 | 1 | 1 |
| while std::getline(input\_file, input\_line) | 1 | n | n |
| Vector<std::string> tokens | 1 | n | n |
| tokens = this->splitLine(input\_line) | 1 | n | n |
| if tokens.size() >= 2 | 1 | n | n |
| Course new\_course | 1 | n | n |
| new\_course.courseId = tokens[0] | 1 | n | n |
| new\_course.courseTitle = tokens[1] | 1 | n | n |
| for integer index = 2; index < tokens.size(); index = index + 1 | 1 | n | n |
| new\_course.prerequisites.push\_back(tokens[index]) | 1 | n | n |
| this->insert(new\_course) | 1 | n | n |
| **Total Cost** | | | 12n + 10 |
| **Runtime** | | | O(n) |

**Evaluation Printing All; Vector data structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| **Function lessThan(Course left\_hand, Course right\_hand) returns boolean** | **N/A** | **N/A** | **N/A** |
| Return left\_hand.courseId < right\_hand.courseId | 1 | 1 | 1 |
|  |  |  |  |
| **Function printAll(vector<Course> courses)** | **N/A** | **N/A** | **N/A** |
| Std::sort(courses.begin(), courses.end(), &lessThan) | n log n | 1 | n log n |
| For int index = 0; index < courses.size(); index = index + 1 | 1 | n | n |
| Put “Course ID: “ + courses.at(index).courseId to output | 1 | n | n |
| Put “Course Title: “ + courses.at(index)..courseTitle to output | 1 | n | n |
| If courses.at(index).prerequisites.size() > 0 | 1 | n | n |
| Put “Prerequisites:” to output | 1 | n | n |
| For int pre\_index = 0; pre\_index < courses.at(index).prerequisites.size(); pre\_index = pre\_index + 1 | 1 | n | n |
| Put courses.at(index).prerequisites.at(pre\_index) | 1 | n | n |
| Put “” to output | 1 | 1 | 1 |
| **Total Cost** | | | n log n + 7n + 2 |
| **Runtime** | | | O(n log n) |

**Evaluation Printing All; Hash Table data structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| **Function CourseTable::print(Course course)** | **N/A** | **N/A** | **N/A** |
| Put “Course ID: “ + course.courseId to output | 1 | 1 | 1 |
| Put “Course Title: “ + course.courseTitle to output | 1 | 1 | 1 |
| If course.prerequisites.size() > 0 | 1 | 1 | 1 |
| Put “Prerequisites:” to output | 1 | 1 | 1 |
| For integer index = 0; index < course.prerequisites.size(); index = index + 1 | 1 | 1 | 1 |
| Put course.prerequisites.at(index) to output | 1 | 1 | 1 |
|  |  |  |  |
| **Function lessThan(Course left\_hand, Course right\_hand) returns Boolean** | **N/A** | **N/A** | **N/A** |
| Return left\_hand.courseId < right\_hand.courseId | 1 | 1 | 1 |
|  |  |  |  |
| **Function CourseTable::printAll()** | **N/A** | **N/A** | **N/A** |
| Vector<Course> sorted\_courses | 1 | 1 | 1 |
| Node\* node | 1 | 1 | 1 |
| For integer index = 0; index < this->courses.size(); index = index + 1 | 1 | n | n |
| Node = &(this->courses.at(index)) | 1 | n | n |
| While ((node != nullptr) && (node->key != UINT\_MAX)) | 1 | n | n |
| Sorted\_courses.append(node->course) | 1 | n | n |
| node = node->nextNode | 1 | n | n |
| std::sort(courseIds.being(), courseIds.end(), &lessThan) | n log n | 1 | n log n |
| for integer index = 0; index < sorted\_courses.size(); index = index + 1 | 1 | n | n |
| this->print(sorted\_courses.at(index)) | 1 | n | n |
| **Total Cost** | | | n log n + 7n + 9 |
| **Runtime** | | | O(n log n) |

**Evaluation Printing All; BST data structure:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| **function CourseTree::printAll(Node\* node = root)** | **N/A** | **N/A** | **N/A** |
| if node != nullptr | 1 | 1 | 1 |
| printAll(node->left) | 1 | n / 2 | n / 2 |
| put node->course.courseId + “: “ + node->course.courseTitle to output | 1 | 1 | 1 |
| if node->course.prerequisites.size() > 0 | 1 | 1 | 1 |
| Put “Prerequisites:” to output | 1 | 1 | 1 |
| For integer index = 0; index < node->course.prerequisites.size(); index = index + 1 | 1 | n | n |
| Put node->prerequisites.at(index) to output | 1 | 1 | 1 |
| printAll(node->right) | 1 | n / 2 | n / 2 |
| **Total Cost** | | | 2n + 5 |
| **Runtime** | | | O(n) |

**Issue 5**

**Advantages and Disadvantages of vector data structure:**

The vector data structure is a fairly straight forward data structure to implement. For storing and retrieving a value is a constant time operation. One should note that during the insertion operation, should a resize of the vector be required, the resize would be a O(n) operation. This makes the overall parsing and storing the courses O(n) in time and space complexity. The disadvantage of the vector data structure is that it is by default un-sorted, so in order to print the courses in sorted order, they must first be sorted. According to the ISO C++ standard the std::sort function will have “Complexity: O(N log(N)) (where N == last – first) comparisons.” (ISO/IEC, 2013) The space complexity is just O(n).

**Advantages and Disadvantages of hash table data structure:**

The hash table data structure is a slightly more complex data structure to implement. For storing and retrieving a value is near constant time. This makes the overall parsing and storing the courses O(n). One should note that during the insertion operation, should a resize of the hash table be required, the resize would be a O(n) operation. The disadvantage of the hash table data structure is that it is by default un-sorted, so in order to print the courses in sorted order, they must first be extracted out of the hash table, sorted, and then printed. This makes the time complexity O(n log n) thanks to the std::sort function, the space complexity would be 2n, because we are extracting out the courses into a second vector to sort, however that simplifies into a space complexity of O(n).

**Advantages and Disadvantages of BST data structure:**

The BST data structure is the most complex of the three to implement. The time and space complexity for parsing and storing the courses are O(n). The main advantage of the BST is that by default walking the tree is done in sorted order, so unlike the other two data structures, printing the BST in sorted order is done in O(n) time, with no additional space overhead required.

**Issue 6**

**Recommendation:**

Based on the Big O time complexities of the three data structures, my recommendation for the data structure I plan to use in my code is the Binary Search Tree (BST). The complexity to implement is not much more than the other two, and the time required to search the tree, or even traverse the entire tree more than makes up for any implementation complexity.

**References**

International Organization for Standards, International Electrotechnical Commission. (2013). Programming Languages – C++. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf