CS 300

Professor Michael Susalla

Dennis James Stelmach

11 June 2023

1. **Previous pseudocode**
   1. **Vector**

struct Course {

string courseNumber

string title

Vector<string> prerequisites

}

Course searchById(Vector<Course> courses, string courseNumber) {

for each course in courses {

if courseNumber of course is the same as courseNumber {

return course

}

}

}

void loadCourses(Vector<Course> courses, string filePath) {

dataFile = open file at filePath

coursesBuffer = empty Course Vector

for each line in dataFile {

courseNumber = the zeroth element after splitting the line by ‘,’

title = the first element after splitting the line by ‘,’

if courseNumber or title are not present {

return early as the input file is not valid

}

prerequisites = a vector of the rest of the optional elements

readCourse = instantiate Course struct with courseNumber, title, prerequisites

append readCourse to coursesBuffer

}

for each Course c in coursesBuffer {

for each prerequisite p of Course c {

if searchById(coursesBuffer, p) is not valid {

return early as the input file is not valid

}

}

}

clear courses

copy contents of coursesBuffer to courses

}

void sortAsc(Vector<Course> courses) {

n = length of courses

for i = 0 to n - 1 {

for j = 0 to n - i - 1 {

if courseNumber of courses[j] is greater than courseNumber of courses[j + 1] {

// Swap courses[j] and courses[j + 1]

temp = courses[j]

courses[j] = courses[j + 1]

courses[j + 1] = temp

}

}

}

}

void printAll(Vector<Course> courses) {

for Course c in courses {

print course information of c

}

}

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

foundCourse = searchById(courseNumber)

if foundCourse is valid {

print out the course information

for each prerequisite of the course {

print the prerequisite course information

}

}

}

* 1. **HashTable**

struct Course {

string courseNumber

string title

HashTable<string> prerequisites

}

Course searchById(HashTable<Course> courses, string courseNumber) {

foundCourse = course from HashTable courses at key courseNumber

while courseNumber of foundCourse is not given courseNumber or foundCourse is not valid {

foundCourse = next node of foundCourse

}

return foundCourse

}

void loadCourses(HashTable<Course> courses, string filePath) {

dataFile = open file at filePath

coursesBuffer = empty Course HashTable

for each line in dataFile

courseNumber = the zeroth element after splitting the line by ‘,’

title = the first element after splitting the line by ‘,’

if courseNumber or title are not present

return early as the input file is not valid

prerequisites = a hash table of the rest of the optional elements

readCourse = instantiate Course struct with courseNumber, title, prerequisites

insert readCourse into coursesBuffer at key courseNumber

for each key k in coursesBuffer

Course c = Course in coursesBuffer at key k

for each key m of prerequisites of Course c

prerequisite = Course in c.prerequisites at key m

if searchById(coursesBuffer, prerequisite) is not valid

return early as the input file is not valid

clear courses

copy contents of coursesBuffer to courses

}

void sortAsc(HashTable<Course> courses) {

sortedCourses = empty Course HashTable

courseKeys = get all keys from courses

sort courseKeys in ascending alphanumeric order (Vector::sortAsc)

for each key in courseKeys {

Course c = Course in courses at key

insert c into sortedCourses at key

}

clear courses

copy contents of sortedCourses to courses

}

void printAll(HashTable<Course> courses) {

for Node n in courses {

while key of n is not invalid {

print course information of course of n

if n’s next node is not invalid {

n = n’s next node

}

else {

break from while loop

}

}

}

}

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

foundCourse = searchById(courseNumber)

print out the course information

for each prerequisite of the course {

print the prerequisite course information

}

}

* 1. **Tree**

struct Course {

string courseNumber

string title

Tree<string> prerequisites

}

Course searchById(Tree<Course> courses, string courseNumber) {

if courses is empty {

return null

}

current = root of courses

while current is not null {

if courseNumber of current is the same as courseNumber {

return current

}

else if courseNumber is less than courseNumber of current {

current = left child of current

}

else {

current = right child of current

}

return null

}

void loadCourses(Tree<Course> courses, string filePath) {

dataFile = open file at filePath

coursesBuffer = empty Tree<Course>

for each line in dataFile {

courseNumber = the zeroth element after splitting the line by ','

title = the first element after splitting the line by ','

if courseNumber or title are not present {

throw error or return early as the input file is not valid

}

prerequisites = a Tree of the rest of the optional elements

readCourse = instantiate Course struct with courseNumber, title, prerequisites

insert readCourse into coursesBuffer

}

for each Course c in coursesBuffer {

for each prerequisite p of Course c {

if searchById(coursesBuffer, p) is null {

throw error or return early as the input file is not valid

}

}

}

clear courses

copy contents of coursesBuffer to courses (typically using pre-order traversal)

}

// Print all courses by courseNumber in ascending alphanumeric order

void inOrder(Tree<Course> courses) {

if courses is not empty{

inOrder(left subtree of courses)

print course information of root of courses

inOrder(right subtree of courses)

}

}

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

foundCourse = searchById(courses, courseNumber)

if foundCourse is null {

print "Course not found"

return

}

print out the course information

for each prerequisite of foundCourse {

print the prerequisite course information

}

}

1. **Pseudocode for menu**

void menu(string courseFilePath, string courseNumberToFind) {

courses = new ADT of choice (Vector, HashTable, or Tree)

choice = 0

display menu items (

1. Load Courses

2. Print Course List

3. Print Course

9. Exit

)

while choice is not equal to 9 {

if choice is equal to 1 {

loadBids(courses, courseFilePath)

}

else if choice is equal to 2 {

if courses ADT is Vector or HashTable {

sortAsc(courses)

printAll(courses)

}

if courses ADT is Tree {

courses.inOrder()

}

}

else if choice is equal to 3 {

printCourseInformation(courses, courseNumberToFind)

}

else if choice is not equal to 9 {

print to alert user the choice is not valid

}

}

print Goodbye message

}

1. **Print all courses in alphanumeric order**

given ADT called courses

if the ADT is a Vector or HashTable {

sourtAsc(courses)

printAll(courses)

}

if the ADT is a Tree {

inOrder(courses)

}

1. **ADT runtime evaluation**
   1. **Vector**
      1. **searchById**

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| for each course in courses | 1 | n | n |
| If courseNumber of course is the same as courseNumber | 1 | n | n |
| return course | 1 | 1 | 1 |
|  |  | Total Cost | 2n + 1 |
|  |  | Runtime | O(n) |

* + 1. **loadCourses**

Given m is a constant representing the number of prerequisites of a course

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| dataFile = open file at filePath | 1 | 1 | 1 |
| coursesBuffer = empty Course Vector | 1 | 1 | 1 |
| for each line in dataFile | 1 | n | n |
| courseNumber = zeroth element after splitting line by ‘,’ | 1 | n | n |
| title = first element after splitting line by ‘,’ | 1 | n | n |
| if courseNumber or title are not present | 1 | n | n |
| return early as the input file is not valid | 1 | 1 | 1 |
| prerequisites = a vector of the rest of the optional elements | 1 | m | m |
| readCourse = instantiate Course struct with courseNumber, title, prerequisites | 1 | n | n |
| append readCourse to coursesBuffer | 1 | n | n |
| for each Course c in coursesBuffer | 1 | n | n |
| for each prerequisite p of Course c | 1 | n \* m | n \* m |
| if searchById(coursesBuffer, p) is not valid | n | n \* m | n2 \* m |
| return early as the input file is not valid | 1 | 1 | 1 |
| clear courses | 1 | 1 | 1 |
| copy contents of coursesBuffer to courses | n | 1 | n |
|  |  | Total Cost | (n2 \* m) + (n \* m) + 7n + m + 2 |
|  |  | Runtime | O(n2) |

* + 1. **printCourseInformation**

Given m is a constant representing the number of prerequisites of a course

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| foundCourse = searchById(courseNumber) | n | 1 | n |
| if foundCourse is valid | 1 | 1 | 1 |
| print course information | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | m | m |
| print the prerequisite course information | 1 | m | m |
|  |  | Total Cost | n + 2m + 2 |
|  |  | Runtime | O(n) |

* 1. **HashTable**
     1. **searchById**

Given m is a constant representing the length of the hash table bucket node chain

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| foundCourse = course from HashTable courses at key courseNumber | 1 | 1 | 1 |
| while courseNumber of foundCourse is not given courseNumber or foundCourse is not valid | 1 | m | m |
| foundCourse = next node of foundCourse | 1 | m | m |
| return foundCourse | 1 | 1 | 1 |
|  |  | Total Cost | 2m + 2 |
|  |  | Runtime | O(1) |

* + 1. **loadCourses**

Given m is a constant representing the number of prerequisites of a course

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| dataFile = open file at filePath | 1 | 1 | 1 |
| coursesBuffer = empty Course Vector | 1 | 1 | 1 |
| for each line in dataFile | 1 | n | n |
| courseNumber = zeroth element after splitting line by ‘,’ | 1 | n | n |
| title = first element after splitting line by ‘,’ | 1 | n | n |
| if courseNumber or title are not present | 1 | n | n |
| return early as the input file is not valid | 1 | 1 | 1 |
| prerequisites = a hash table of the rest of the optional elements | 1 | m | m |
| readCourse = instantiate Course struct with courseNumber, title, prerequisites | 1 | n | n |
| insert readCourse into coursesBuffer at key courseNumber | 1 | n | n |
| for each key k in coursesBuffer | 1 | n | n |
| Course c = Course in coursesBuffer at key k | 1 | n | n |
| for each key m of prerequisites of Course c | 1 | n \* m | n \* m |
| prerequisite = Course in c.prerequisites at key m | 1 | n \* m | n \* m |
| if searchById(coursesBuffer, p) is not valid | 1 | n \* m | n \* m |
| return early as the input file is not valid | 1 | 1 | 1 |
| clear courses | 1 | 1 | 1 |
| copy contents of coursesBuffer to courses | n | 1 | n |
|  |  | Total Cost | 3(n \* m) + 7n + m + 2 |
|  |  | Runtime | O(n) |

* + 1. **printCourseInformation**

Given m is a constant representing the number of prerequisites of a course

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| foundCourse = searchById(courseNumber) | 1 | 1 | 1 |
| if foundCourse is valid | 1 | 1 | 1 |
| print course information | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | m | m |
| print the prerequisite course information | 1 | m | m |
|  |  | Total Cost | 2m + 3 |
|  |  | Runtime | O(1) |

* 1. **Tree**
     1. **searchById**

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| if courses is empty | 1 | 1 | 1 |
| return null | 1 | 1 | 1 |
| current = root of courses | 1 | 1 | 1 |
| while current is not null | 1 | log(n) | log(n) |
| if courseNumber of current is the same as courseNumber | 1 | log(n) | log(n) |
| return current | 1 | 1 | 1 |
| else if courseNumber is less than courseNumber of current | 1 | log(n) | log(n) |
| current = left child of current | 1 | log(n) | log(n) |
| else | 1 | log(n) | log(n) |
| current = right child of current | 1 | log(n) | log(n) |
| return null | 1 | 1 | 1 |
|  |  | Total Cost | 6log(n) + 5 |
|  |  | Runtime | O(log(n)) |

* + 1. **loadCourses**

Given m is a constant representing the number of prerequisites of a course

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| dataFile = open file at filePath | 1 | 1 | 1 |
| coursesBuffer = empty Tree<Course> | 1 | 1 | 1 |
| for each line in dataFile | 1 | n | n |
| courseNumber = zeroth element after splitting line by ‘,’ | 1 | n | n |
| title = first element after splitting line by ‘,’ | 1 | n | n |
| if courseNumber or title are not present | 1 | n | n |
| return early as the input file is not valid | 1 | 1 | 1 |
| prerequisites = a Tree of the rest of the optional elements | 1 | m | m |
| readCourse = instantiate Course struct with courseNumber, title, prerequisites | 1 | n | n |
| insert readCourse into coursesBuffer | log(n) | n | n \* log(n) |
| for each Course c in coursesBuffer | 1 | n | n |
| for each prerequisite p of Course c | 1 | n \* m | n \* m |
| if searchById(coursesBuffer, p) is null | log(n) | n \* m | log(n) \* n \* m |
| throw error or return early as the input file is not valid | 1 | 1 | 1 |
| clear courses | 1 | 1 | 1 |
| copy contents of coursesBuffer to courses (typically using pre-order traversal) | n | 1 | n |
|  |  | Total Cost | (n \* log(n)) + 7n + (n \* m) + m + 5 |
|  |  | Runtime | O(n \* log(n)) |

* + 1. **printCourseInformation**

Given m is a constant representing the number of prerequisites of a course

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| foundCourse = searchById(courseNumber) | log(n) | 1 | log(n) |
| if foundCourse is null | 1 | 1 | 1 |
| print “Course not found” | 1 | 1 | 1 |
| return | 1 | 1 | 1 |
| print course information | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | m | m |
| print the prerequisite course information | 1 | m | m |
|  |  | Total Cost | log(n) + 2m + 4 |
|  |  | Runtime | O(log(n)) |

1. **Advantages and disadvantages of each structure**
   1. **Vector**
      1. Advantages
         1. Random access: Courses in a Vector can be accessed using their indices, allowing for constant-time access
         2. Simple implementation: Vectors are straightforward to understand and implement
         3. Preserves insertion order: Courses are stored in the order they were inserted
      2. Disadvantages
         1. Inefficient search: searchById for a Vector has a linear runtime complexity of O(n) because it must iterate over each course in the Vector without additional adjustments
         2. Costly insertion and deletion: Courses in a Vector may need to be shifted to account for an insertion or deletion, which introduces overhead and results in a higher time complexity
   2. **Hash Table**
      1. Advantages
         1. Fast search: searchById for a Hash Table has an constant runtime complexity of O(1) on average
         2. Efficient insertion and deletion: Inserting and deleting Courses from a Hash Table can be done in constant time
         3. Flexible key-value storage: Hash Tables allow for key-value pairs, making it easy mapping associations between courses and their prerequisites
      2. Disadvantages
         1. No inherent order: The elements in a hash table are not stored in a specific order
         2. High space complexity: To gain efficiency in time complexity, space complexity is partially sacrificed
   3. **Tree**
      1. Advantages
         1. Efficient search: searchById for a Tree has a logarithmic time complexity of O(log(n)) in a balanced tree
         2. Ordered traversal: Tree structures allow for traversing Courses in a specific order such in-order, pre-order, or post-order
         3. Dynamic structure: Trees can dynamically adjust their structure to maintain balance to ensure efficient search, insertions, and deletions
      2. Disadvantages
         1. Complex implementation: Tree structures can be more complex to implement and understand compared to Vectors or Hash Tables
         2. No random access: Trees do not provide direct random access to elements based on indices like Vectors and Hash Tables and must be traversed.
2. **Recommended data structure for future work**

I plan to use a Hash Table structure for storing courses and their prerequisites as they offer the most efficient retrieval and modification functions time-complexity-wise compared to Vectors and Trees. The fact that Hash Tables do not have an inherit order is not a negative in this situation, as course ordering can be determined from the data itself if needed. A Tree structure also seems to be a suitable structure, but the fact that courses are semi-randomly accessed and Trees do not support this, and the greater implementation complexity does not suit this situation.