CS-300

6/14/2024

Justin Phillips

Southern New Hampshire University

**Binary Search Tree Pseudocode**

Struct Course {

String courseNumber // The course identifier, i.e. CSCI100. This acts as the unique identifier.

String name // The name of the course

Vector <courses> // The prerequisites are arguments [2]->...[-1], the third argument to the last. This vector’s size is [0..\*]. This courses are saved as prerequisites courseNumber strings.

}

/////////////////////////////////BINARY TREE DATA STRUCTURE///////////////////////////////////////////

Class BinarySearchTree {

Struct Node {

Node Ptr left

Node Ptr Right

Course Course

}

Node Root

Vector SortedCoursesToPrint = {} // ToPrint vector

SortAlphanumeric() // Run the sorter

Constructor BinarySearchTree (courseNumber, courseName, courses[0..\*]) {

If courseNumber < currentNode->courseNumber

If left occupied

Recurse left

Else

Insert left

Else If courseNumber > currentNode->courseNumber

If right occupied

Recurse right

Else

Insert right

Course(courseName, courses[0..\*])

Insert courseNumber, Course

Function FindSpecificCourse (courseNumber)

If courseNumber < currentNode->courseNumber

Recurse left

Else If courseNumber > currentNode->courseNumber

Recurse right

Else

Return currentNode

::CourseInvalidation() { // Snuff out invalid prerequisites.

For course in courses {

For prereq in Course.courses {

If search prereq == null

Remove(prereq)

}

}

}

CourseInvalidation() // Run the invalidator to remove unlisted courses from the prereqs for each of the courses.

}

}

::Insert(String courseNumber, Course course) { // Insert the course at the natural hash of the courseNumber

Int index = hash(courseNumber) // Get location hash

Location = courses[index] // Get the location for the course

Course\* newCourse = new Course; // Create a new course to insert

newCourse.courseNumber = courseNumber

newCourse.course = course

While location != nullptr { // If the target insertion is occupied,

Location = location.next // Progress the search for an insertion point towards the greater end of the vector

}

if location == nullptr { // Once a free insertion point is found

newCourse.next = location.next // Insert it by assigning its next and previous to be the current memory location

newCourse.previous = location.previous

return newCourse // Return the course object as a token of completion

}

Return null // A location is never found, so a null should be returned to flag incompletion

}

::PrintCourse(String courseNumber) {

Print “ID: “ course.courseNumber “ | “ course.courseName

If course.courses[0] != null { // Determine if has prerequisites

Print “ | Prerequisites: “

While I = 0; I < course.courses.length; I++ { // Proper pluralization

Print courses “, “

}

If I == course.courses.length {

Print courses

}

}

Print “\n” // Print a newline at the end

}

::Search(String courseNumber, Boolean print=false) {

Int index = hash(courseNumber)

Course\* course = courses[index]

While (course.next != UINTMAX) { // While there is still a value or a nullptr assigned to the next location

If course.CourseNumber == courseNumber { // If the course is found

If print { // If want the searched for course printed

PrintCourse(courseNumber)

}

Return course.course // Return it

} // Otherwise progress the search through the list

course = courses[index.next]

} // Once every location is checked and all do not trigger the return,

Return null

}

::PrintAll() {

For course in SortedCoursesToPrint {

PrintCourse(course.courseNumber)

}

}

::SortAlphanumeric(Index = root) { // Cycle through the BST and use the left/right to write a sorted list out

While index.left and index.left !memberOf(SortedCoursesToPrint) { // So long as there is a route left (lesser)

::SortAlphanumeric(index.left) // Recurse down that left path

}

While index.right and index.right !memberOf(SortedCoursesToPrint) { // So long as there is a route right (greater)

::SortAlphanumeric(index.right) // Recurse down that left path

}

SortedCoursesToPrint.append(index)

If root !memberOf(SortedCoursesToPrint) {

::SortAlphanumeric(root)

} else {

Return

}

}

////////////////////////////////////FILE HANDLING///////////////////////////////////////////////////////////////////

Function txtParse(BinarySearchTree BST, String path) // Parse the text file from the input path

File = Open(String path)

If path[:length(“Appropriate path”)] != “Appropriate path” { // For security, validate the authenticity of the path

Print “Bad path!”

Return Error // Do not continue running; Could be a vulnerability under attack

}

Lines = split(file.strip(), “\n”) // Split the file into lines

For line in lines {

If line != EOF { // As long as the file is continuing

Tokens = line.split(“,”) // Split each line into its comma separated values

If Tokens.length >= 2 { // As long as the appropriate number of tokens, one courseNumber, one name, and potentially any prerequisite courses exist, process the line as a new course object.

newCourse = new Course

newCourse.courseNumber = tokens[0] // Assign the number string

newCourse.name = tokens[1] // Assing the name string

for prerequisite in tokens[2:] { // Assign each of the prerequisites strings starting from token 2 (third location)

newCourse.courses.append(tokens[prerequisite])

}

BST.insert(newCourse) // Add the new course to its location in the BST

} else { // Otherwise, the line is bad; Print an error message, but don’t invoke a crash or stop running, in case it is a very minor error.

Print “Bad line at “ path “ line “ line

}

}

}

}

/////////////////////////////////////////////////////////MAIN///////////////////////////////////////////////////////////////////////

Init::{

BST = new BinarySearchTree

txtParse(BST, “ourSecurePath/file/courses.txt”)

int input

while input != 9 {

input = cin

switch input:

1. Load
2. Print Alphanumerically Sorted List
3. Search and Print Specific Course

9. Quit

}

Print “Quitting gracefully”

Return 0

}

**Hash Table Pseudocode**

Struct Course {

String courseNumber // The course identifier, i.e. CSCI100. This acts as the unique identifier.

String name // The name of the course

Vector <courses> // The prerequisites are arguments [2]->...[-1], the third argument to the last. This vector’s size is [0..\*]. This courses are saved as prerequisites courseNumber strings.

}

//////////////////////////////////HASHTABLE DATA STRUCTURE////////////////////////////////////////////

Struct Hashtable {

String hashKey // The hash key for the hash table.

Int size // Size of the hashtable

Vector <courses[Course]> // A vector of courses saved to the hash table.

}

::CourseInvalidation() { // Snuff out invalid prerequisites.

For course in courses {

For prereq in Course.courses {

If search prereq == null

Remove(prereq)

}

}

}

::HashTable(int initsize=0) { // A constructor

Size = initsize

Vector = new Vector[size] // Create a new vector to store the contents of the hashtable up to its capacity.

For (int I; I < size; I++) { // For every potential entry in the vector of the size of the hashtable

Vector[I] = nullptr // Fill the entry in the vector with a nullptr

}

CourseInvalidation() // Run the invalidator to remove unlisted courses from the prereqs for each of the courses.

SortedCourses = SortAlphanumeric(vector)

}

::Hash(string courseNumber) { // Using the courseNumber, convert it to an int then use modulo in respect to size to calculate the hashvalue for which the course will be indexed.

Return atoi(courseNumber) % size;

}

::Insert(String courseNumber, Course course) { // Insert the course at the natural hash of the courseNumber

Int index = hash(courseNumber) // Get location hash

Location = courses[index] // Get the location for the course

Course\* newCourse = new Course; // Create a new course to insert

newCourse.courseNumber = courseNumber

newCourse.course = course

While location != nullptr { // If the target insertion is occupied,

Location = location.next // Progress the search for an insertion point towards the greater end of the vector

}

if location == nullptr { // Once a free insertion point is found

newCourse.next = location.next // Insert it by assigning its next and previous to be the current memory location

newCourse.previous = location.previous

return newCourse // Return the course object as a token of completion

}

Return null // A location is never found, so a null should be returned to flag incompletion

}

::PrintCourse(String courseNumber) {

Print “ID: “ course.courseNumber “ | “ course.courseName

If course.courses[0] != null { // Determine if has prerequisites

Print “ | Prerequisites: “

While I = 0; I < course.courses.length; I++ { // Proper pluralization

Print courses “, “

}

If I == course.courses.length {

Print courses

}

}

Print “\n” // Print a newline at the end

}

::Search(String courseNumber, Boolean print=false) {

Int index = hash(courseNumber)

Course\* course = courses[index]

While (course.next != UINTMAX) { // While there is still a value or a nullptr assigned to the next location

If course.CourseNumber == courseNumber { // If the course is found

If print { // If want the searched for course printed

PrintCourse(courseNumber)

}

Return course.course // Return it

} // Otherwise progress the search through the list

course = courses[index.next]

} // Once every location is checked and all do not trigger the return,

Return null

}

::PrintAll() {

For course in sortedCourses {

PrintCourse(course)

}

}

::SortAlphanumeric(coursesVector) {

Vector sorted = {} // Temporary vector to add sorted course pointers

For course in coursesVector { // For each course

If coursesVector[course.next] { // So long as a next course exists

If Compare (courses[course], courses[course.next]) { // If the current index is in a higher place alphanumerically than its successor

sorted.append(Courses[course]) // Add the index pointer to the end of the sorted vector

} Else { // Otherwise

I = 1 // Initialize a temp variable for precursors to compare the index to

While !compare(courses[course.prev [i], courses[course.next]) { // While the precursor is of higher location than the index

I++ // Further search the precursors

} else { // When the while terminates naturally

Sorted.insert(courses[course.next] after courses[course.prev[i]) // Insert the index after the precursor

}

}

} else { // Once there are no more values to compare

Return sorted // Return the sorted vector.

}

}

}

::Compare (index, comparison) { // Compare alphanumeric values between the index and comparison

if index > comparison {

return true // If the index is greater return true

} else {

Return false // If the index is smaller return false

}

}

////////////////////////////////////FILE HANDLING///////////////////////////////////////////////////////////////////

Function txtParse(hashtable HashTable, String path) // Parse the text file from the input path

File = Open(String path)

If path[:length(“Appropriate path”)] != “Appropriate path” { // For security, validate the authenticity of the path

Print “Bad path!”

Return Error // Do not continue running; Could be a vulnerability under attack

}

Lines = split(file.strip(), “\n”) // Split the file into lines

For line in lines {

If line != EOF { // As long as the file is continuing

Tokens = line.split(“,”) // Split each line into its comma separated values

If Tokens.length >= 2 { // As long as the appropriate number of tokens, one courseNumber, one name, and potentially any prerequisite courses exist, process the line as a new course object.

newCourse = new Course

newCourse.courseNumber = tokens[0] // Assign the number string

newCourse.name = tokens[1] // Assing the name string

for prerequisite in tokens[2:] { // Assign each of the prerequisites strings starting from token 2 (third location)

newCourse.courses.append(tokens[prerequisite])

}

HashTable.insert(newCourse) // Add the new course to its location in the hash table

} else { // Otherwise, the line is bad; Print an error message, but don’t invoke a crash or stop running, in case it is a very minor error.

Print “Bad line at “ path “ line “ line

}

}

}

}

/////////////////////////////////////////////////////////MAIN///////////////////////////////////////////////////////////////////////

Init::{

coursesHashtable = new hashtable

txtParse(courseHashtable, “ourSecurePath/file/courses.txt”)

int input

while input != 9 {

input = cin

switch input:

1. Load
2. Print Alphanumerically Sorted List
3. Search and Print Specific Course

9. Quit

}

Print “Quitting gracefully”

Return 0

}

**Vector Pseudocode**

Struct Course

Int Course.courseNumber

String Course.title

Vector Prereqs

End struct

Vector sortedCourses = {}

Vector courses[course]

PrintCourse(String courseNumber) {

Print “ID: “ course.courseNumber “ | “ course.courseName

If course.courses[0] != null { // Determine if has prerequisites

Print “ | Prerequisites: “

While I = 0; I < course.courses.length; I++ { // Proper pluralization

Print courses “, “

}

If I == course.courses.length {

Print courses

}

}

Print “\n” // Print a newline at the end

}

SortCourses() {

For course in courses {

For sortedCourse in sortedCourses {

If course < sortedCourse {

SortedCourses.insert(before SortedCourse, course)

break

}

}

}

}

PrintSorted() {

For SortedCourse in SortedCourses {

PrintCourse(SortedCourse)

}

}

Search(courseNumber, print = false) {

For course in courses {

If course.courseNumber == courseNumber

If print {

PrintCourse(course)

}

Return course

}

}

}

////////////////////////////////////FILE HANDLING///////////////////////////////////////////////////////////////////

Function txtParse(hashtable HashTable, String path) // Parse the text file from the input path

File = Open(String path)

If path[:length(“Appropriate path”)] != “Appropriate path” { // For security, validate the authenticity of the path

Print “Bad path!”

Return Error // Do not continue running; Could be a vulnerability under attack

}

Lines = split(file.strip(), “\n”) // Split the file into lines

For line in lines {

If line != EOF { // As long as the file is continuing

Tokens = line.split(“,”) // Split each line into its comma separated values

If Tokens.length >= 2 { // As long as the appropriate number of tokens, one courseNumber, one name, and potentially any prerequisite courses exist, process the line as a new course object.

newCourse = new Course

newCourse.courseNumber = tokens[0] // Assign the number string

newCourse.name = tokens[1] // Assing the name string

for prerequisite in tokens[2:] { // Assign each of the prerequisites strings starting from token 2 (third location)

newCourse.courses.append(tokens[prerequisite])

}

Courses.append(newCourse) // Add the new course to its location in the hash table

} else { // Otherwise, the line is bad; Print an error message, but don’t invoke a crash or stop running, in case it is a very minor error.

Print “Bad line at “ path “ line “ line

}

}

}

}

/////////////////////////////////////////////////////////MAIN///////////////////////////////////////////////////////////////////////

Init::{

txtParse(courses, “ourSecurePath/file/courses.txt”)

int input

while input != 9 {

input = cin

switch input:

1. Load
2. Print Alphanumerically Sorted List
3. Search and Print Specific Course

9. Quit

}

Print “Quitting gracefully”

Return 0

}

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Search Tree** |
| **Load** | O(n \* m) | O(n \* m) | O(n \* (m + log n)) |
| **Search** | O(n) | O(n) | O(n/2) |
| **Sort** | O(n^3) | O(n^3) | O(n^3) |
| **Print** | O(n) + O(m) | O(n) + O(m) | O(n) + O(m) |

In the above table, *n* represents the primary data unit; In the load function, this refers to the lines in the document loaded. Otherwise, it refers to the nodes (courses) being processed in the function. Subsequent variables represent subsequent data handled. In loading, for instance, *m* refers to the tokens per line processed. The table also assumes a worst use case per function.

The **Vector** data structure uses a potentially exponential amount of time to load its members, reflective of the number of lines (for line) and tokens (for token). Its search is a linear search through a straightforward vector (for member of vector) and a comparison (if, constant time use). Its sort is reflective of an outer loop, an inner loop, and an insert function who may involve shifting the list, meaning that its sort is among its most computationally expensive components.

The **Hash Table** data structure’s load function operates similarly to the vector’s, and has a similar computation time. The search function is composed of a constant hash() function, a constant comparison, and a while loop, meaning it is fairly inexpensive. The sort function is a constant if comparison, a for loop, a constant append function, a while loop, and an insert, meaning two constants and three functions of O(n) complexity.

The **Binary Search Tree** data structure’s load function is more computationally expensive than the others, as it integrates sorting into itself, featuring a comparison, for lines, for tokens, and an insert function which is logarithmic in nature. The sort function recurses down the left until it runs out of left-sided unvisited nodes, then recurses right, each taking a time complexity reflective of the members of the path O(n), an append function O(n), and a search for membership, O(n). Finally, the search function is a recurse down each branch by a single less or greater comparison, which yields worst case O(n/2).

Among all data structures, the expensive sort() function precedes the cheaper print() function, whose role is to iterate through a singular for and run a constant cout function, meaning that the computation time is essentially a 1:1 of the size of the course list, then print the information which is constant and then each pre-requisite, which can be O(m).

In comparison to each other, a vector is relatively straightforward and simple to use, understand, and implement. It would save development time drastically, but is inefficient for insertion, deletion, popping, or any function that alters its contents, having to iterate all its contents every time. Its search function’s worst scenario is a linear progression through the contents of the vector. I then would not recommend the use of a vector in this program.

A hash table uses integer keys generated by a hash function to make lookups and storage more efficient, and keeps the expected rates of alterations for its contents low because refactoring is less intensive. Hash functions, however, may lead to extra computational complexity, and collisions may happen, leading to some iteration through the list to find a free slot or, in a search, find a member in a location that is different from the one anticipated by the hash method. Overall a hash table is an improvement over the vector layout.

The binary search tree offers itself in-built sorting, which makes lookups easier and faster. It also offers performant refactoring through easily modified directional pointers in either the left or right dimension (less than or greater than the key, respectively). Its performance, however, depends to a great degree on the loading of the data file. My tree implementation does not involve a randomization of line reading order while the text is parsed, and there is the probability that the trees branches are balanced in a way that hinders performance. Overall, I would expect greater computational efficiency from the BST and recommend it as the data structure for our application. The O-notation data suggests a logarithmic load time and a relatively swift look-up time, as there are two dimensions of travel per rung at most for a search, and the comparison computation is extremely inexpensive.

Our application will likely need occasional modifications to the data. Course requirements change, courses get added, courses get removed, and so we need a fast implementation of inserting, removing, sorting, validation of pre-requisites, and conversion of CSV to loaded program. My choice for this category of requirements is the binary search tree, whose implementation of loading CSV files is O(n \* (m + log n)), computationally cheaper than O(n\*m) of the vector and hashtable methods. We will also need fast searches; Because the BST sorts itself as it loads nodes, it has a radically fast lookup technique for individual courses, O(n/2), as well as a quick sorting method which relies on a constant comparison greater or less than the value of the current node. Therefore, the BST is the best data structure for this application’s use case.