Nick Franklin

2 December 2022

CS300 DSA: Analysis and Design

Southern New Hampshire University

Project 1

**Pseudocode for Vector Data Structure**

**// Loading data into vector structure**

Bool Function readFile(courseFile, courseLines)

Declare vector courseNumbers

Declare vector courseTitles

Declare vector coursePrereq

Declare string variable line

While (not end of file)

Initialize vector courseData = getline(courseFile, line, separator = ‘,’ )

Append current variable line to courseData

If (size of courseData < 2)

Return False

End If

courseNumbers[i] = courseData[0]

courseTitles[i] = courseData[1]

Increment i

If (size of courseData > 2)

For (k = 2, k < courseData size, ++k)

coursePrereq[j] = courseData[k]

Increment j

End For

End If

End While

If Bool == True

For (i=0, i < coursePrereq size, ++i)

If coursePrereq[i] is not in courseNumbers

Return False

End If

End For

End If

Return Bool

**// Create course object**

Class Course

Declare string variable courseNum

Declare string variable courseTitle

Declare vector coursePrerequisites

Constructor Course(line)

courseNum = getline(line, separator = ‘,’)[0]

courseTitle = getline(line, separator = ‘,’)[1]

If (size of getline(line, separator = ‘,’) > 2 )

coursePrerequisites = getline(line, separator = ‘,’)[ 2 to size of getline]

End If

End Constructor

Return

Function createCourse(<Course> Courses, courseFile)

Initialize list courseLines

If readFile(courseFile, courseLines) == TRUE

For each line in courseLines

Append new Course object to Courses

End For

End IF

Else

Print("Error reading file")

End Else

Return

**// Print course list**

Function printCourseList

low = current Course

For all Courses

If next Course < low

low = next Course

Swap current Course and next Course

End if

End For

For each Course

printCourseInformation()

End for

return

**// Print searched course info**

Function printCourseInformation(<Course> Courses, string searchCourseNum) {

For (i = 0, i < Courses size, ++i)

If (Course[i] courseNum == searchCourseNum)

Print Course[i] courseNum

Print Course[i] courseTitle

For all Course[i] coursePrerequisites

Print coursePrerequisites

End For

End If

End For

Return

**// Main menu**

Int main()

Load default file path

Int choice = 0

While choice does not equal 9

Cout menu

If choice = 1

readFile()

If choice = 2

printCourseList()

If choice = 3

Cin searchCourseNum

printCourseInformation(searchCourseNum)

If choice = 9

Exit program

End while

Return

**Vector Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Read File same for all data structures** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Call function readFile | 1 | 1 | 1 |
| Declare vector courseNumbers | 1 | 1 | 1 |
| Declare vector courseTitles | 1 | 1 | 1 |
| Declare vector coursePrereq | 1 | 1 | 1 |
| Declare string variable line | 1 | 1 | 1 |
| Initialize vector courseData = getline(courseFile, line, separator = ‘,’ ) | 1 | n | n |
| Append line to courseData | 1 | n | n |
| If courseData < 2 return false | 1 | n | n |
| courseNumbers[i] = courseData[0] | 1 | n | n |
| courseTitles[i] = courseData[1] | 1 | n | n |
| For all prerequisites coursePrereq[j] = courseData[k] | 1 | n | n |
| For coursePrereq size if coursePreq[i] is not in courseNumbers return False | 1 | n | n |
|  |  | Total Cost | 7n + 5 |
|  |  | Runtime | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Create Course Object** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Declare string variable courseNum | 1 | 1 | 1 |
| Declare string variable courseTitle | 1 | 1 | 1 |
| Declare vector coursePrerequisites | 1 | 1 | 1 |
| Constructor Course(line) | 1 | n | n |
| courseNum = getline[0] | 1 | n | n |
| courseTitle = getline[1] | 1 | n | n |
| If size of getline > 2  coursePrerequisites = getline[2 to size of getline] | 1 | n | n |
| Initialize list courseLines | 1 | 1 | 1 |
| If readFile = True  Append new Course object to Courses | 1 | n | n |
| Else  Cout error | 1 | n | n |
|  |  | Total Cost | 6n + 4 |
|  |  | Runtime | O(n) |

Some advantages of using a vector data structure are firstly that it can be implemented rather easily. It also is not too time consuming to search as long as it has been sorted into order. However, sorting the vector data structure, unless done upon creation is time consuming due to requiring swapping and shifting. And adding entries to a vector is easy as long as they are appended but becomes more complicated and time consuming if it is necessary to insert the new entry in order.

**Pseudocode for Hash Table Data Structure**

**// Read File and check errors**

FUNCTION Bool openFile with parameters: (courseInfoFile, fileLines[])

Declare vector storage courseNumbers[]

Declare vector storage courseTitles[]

Declare vector storage coursePrereq[]

Declare string variable line

LOOP While not the end of the file

Initialize vector storage courseData[] = get line (courseInfoFile, line, separator = “,”)

Append the current line to fileLines[]

If the size of courseData[] is less than 2

Return False

End If

courseNumbers[i] = courseData[0]

courseTitles[i] = courseData[1]

Increment i

If the size of courseData[] is greater than 2

LOOP: For each k >= 2 in course data

coursePrereq[j] = courseData[k]

Increment j

End While

If Bool is True

LOOP For all i in coursePrereq[]

If coursePrereq[i] is not in courseNumbers

Bool is False

Return Bool

**// Create Course Objects and Store into Hash Table Data Structure**

STRUCTURE Courses

Declare String variable courseNumber

Declare String variable courseTitle

Declare vector storage coursePrerequisite[]

CLASS Course hash table

PRIVATE:

STRUCTURE: Node

Courses course

Unsigned int key

Node \*next

DEFAULT CONSTRUCTOR: Node()

key = UINT\_MAX

next = nullptr

// constructor for initializing with a course

CONSTRUCTOR: Node(Courses aCourse) : Node()

course = aCourse

// constructor for initializing with a course and key

CONSTRUCTOR: Node(Courses aCourse, unsigned int akey) : Node(aCourse)

key = akey

Initialize vector<Node> nodes

Initialize int variable tableSize = default

Initialize unsigned int hash(int key)

PUBLIC:

Void PrintAll()

Declare variable String cNum

Declare variable String cTitle

Declare vector storage cPrerequisites[]

CONSTRUCTOR: Course(line)

cNum = getline (line, separator = “,”)[0]

cTitle = getline (line, separator = “,”)[1]

If the size of getline is greater than 2 then it has prerquisites

cPrerequisites[] = getline (line, separator = “,”) [2 to size of getline]

End If

End Class

FUNCTION int Course: :hash(int key)

Return key % tableSize

FUNCTION createCourse(<Node> nodes, courseInfoFile)

Initialize vector courseLines[]

If openFile(courseInfoFile, courseLines) Bool = True

For every line in courseLines

Get key from hash of cNum

Add new Node at nodes key position

End For

End If

Else

Print “Problem while reading file”

End Else

Return

**// Print Course List**

FUNCTION Course: :PrintAll()

Auto iterateNode = nodes.begin()

LOOP For all iterateNode until end of nodes

If current iterateNode key is not UINT\_MAX

Print all current iterateNode data

Iterate iterateNode to next node

LOOP While iterateNode is not null

Print all current iterateNode data

Iterate iterateNode to next node

Return

**// Print course searched from key**

Courses Course:: printCourseInfo(courseID)

Courses course

unsigned key = bidID key

Node\* searchNode = node at key

If entry found for the key

Return key course info

If no entry found for the key

Return error

While node not equal to nullptr

If current node matches key

Return key course info

searchNode = next node

**// Main menu**

Int main()

Load default file path

Int choice = 0

While choice does not equal 9

Cout menu

If choice = 1

openFile()

If choice = 2

printAll()

If choice = 3

Cin courseID

printCourseInfo(courseID)

If choice = 9

Exit program

**Hash Table Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Create Course Object** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Declare String variable courseNumber | 1 | 1 | 1 |
| Declare String variable courseTitle | 1 | 1 | 1 |
| Declare vector storage coursePrerequisite[] | 1 | 1 | 1 |
| cNum = getline[0] | 1 | n | n |
| cTitle = getline[1] | 1 | n | n |
| If size of getline > 2 | 1 | n | n |
| cPrerequisites[] = getline[2 to end] | 1 | n | n |
| Course:: hash(int key)  Return key % tableSize | 1 | n | n |
| Initialize vector courseLines[] | 1 | 1 | 1 |
| If openFile(courseInfoFile, courseLines) Bool = True | 1 | 1 | 1 |
| For every line in courseLines | 1 | n | n |
| Add new Node at key position | 1 | n | n |
|  |  | **Total Cost** | 7n + 5 |
|  |  | **Runtime** | O(n) |

A great advantage of a hash table data structure is that each entry has a key so searching for a specific entry is rather quick and easy. This also applies for deleting entries as well since it does not need to be sorted but rather a few pointers only need to be changed. A problem with the hash table though is that, unless the perfect size table is used, they can take up unnecessary space.

**Pseudocode for Binary Tree Data Structure**

**// Open File and check it**

FUNCTION BOOL loadCourses(string file, BinarySearchTree\* bst)

Initialize string variable line

While not end of file:

// Initialize file parser using file path

Initialize fileData[] = getline(file, line, separator = “,”)

If size of fileData < 2

Return False

End If

Course course

course.courseNumber = fileData[i][1]

course.courseTitle = fileData[i][2]

increment i

If size of fileData[] > 2

For (k = 2; k < fileData size; ++k)

course.coursePrerequisite[j] = fileData[k]

increment j

bst->Insert(course)

End While

If Bool == True

For (i = 0; i < course.coursePrerequisite size; ++i)

If course.coursePrerequisite[i] is not in courseNumber

Return False

Return Bool

**// Create course objects and store in tree**

STRUCTURE Course

Declare String variable courseNumber

Declare String variable courseTitle

Declare vector storage coursePrerequisite[]

Course()

STRUCTURE Node

Course course

Node \*left

Node \*right

Default Constructor: Node()

left = nullptr

right = nullptr

INITIALIZED with a course: Node(Course aCourse)

course = aCourse

CLASS BinarySearchTree

PRIVATE:

Node\* root

Void addNode(Node\* node, Course course)

Void inOrder(Node\* node)

PUBLIC:

BinarySearchTree()

Void InOrder()

Void Insert(Course course)

BinarySearchTree::BinarySearchTree()

Root = nullptr

void Binary SearchTree::Insert(Course course)

if (root == nullptr)

root = new Course(course)

End If

Else

this->addNode(root, course)

End Else

Void BinarySearchTree::addNode(Node\* node, Course course)

If (node larger then go left)

If (no left node)

add this node to left

Else

Recurse left

Else

If (no right node)

Add this node to right

Else

Recurse right

**// Print Course list**

Void BinarySearchTree::InOrder()

this->inOrder(root)

Void BinarySearchTree::inOrder(Node\* node)

If (node not nullptr)

inOrder(left)

cout Node info

inOrder(right)

**// Print Course from search course number**

Course BinarySearchTree::Search(string courseNumber)

Node\* current = root

While current does not equal nullptr

If current equals search

Print current course info

If search number is smaller than current

Traverse tree left

Else if search number is larger than current

Travers tree right

Course course

Return course info

**// Main menu**

Int main()

Load default file path

Define Binary Search tree to hold courses

Int choice = 0

While choice does not equal 9

Display menu

If choice = 1

loadCourses()

If choice = 2

InOrder()

If choice = 3

Cin courseNumber

Search(courseNumber)

If choice = 9

Exit program

**Binary Tree Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Create Course Object** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **If root equals nullptr add new Course(course)** | **1** | **1** | **1** |
| **Else this->addNode(root, course)** | **1** | **n** | **n** |
| **If new node larger than current then traverse left** | **1** | **n** | **n** |
| **If no left node then add node to left** | **1** | **n** | **n** |
| **Else traverse left** | **1** | **n** | **n** |
| **Else if no right node add node to right** | **1** | **n** | **n** |
| **Else travers right** | **1** | **n** | **n** |
|  |  | **Total Cost** | **6n + 1** |
|  |  | **Runtime** | **O(n)** |

One of the greatest perks of using a binary tree data structure is that it is easy to order and it is very easily searched, sorted and added to as long as it is balanced. However, if the tree is not balanced then performance problems can occur. The first entry is very important when considering a binary tree’s balance and so storing data in a binary tree must be done more carefully than with the other data structures.

**Recommendation**

Although all of the data structures have their strengths and weaknesses and are all useful in certain situations, for this assignment, I would go with the binary tree. Since the data needs to be easily accessed in alphanumerical order and more courses may be added, it just makes the most sense to use the binary tree which inserts in order easily as compared to inserting in order with a vector. It is also easy to retrieve in order, especially compared to a hash table which is not likely to have its data stored in order. Searching for a specific entry would hypothetically be a bit more efficient in the hash table but because its data is not necessarily stored in order, when trying to retrieve the entire data list, it would have to be sorted every time. Considering all this, I think the binary tree is the strongest choice for this assignment and the hash table would be my second choice.