Project One Pseudocode and Runtime Analysis

SNHU CS-300

Kentrell Edwards

August 8, 2022

Struct Course

String courName

String courNum

Vector<String>Prereq

Struct Node

Course course

Create course(key)

IF(entry == key)

return node-> course

ENDIF

IF(entry != key)

return course

ENDIF

Class Tree

Private:

Node\* root

Void addNode(Node\* node, Course course)

Public:

BST()

Void InOrder()

Void Insert(Course course)

Course Search(string courNum)

BST::BST

root == nullptr

void BST::Insert(Course course)

IF(root == nullptr)

root = newNode(course)

ENDIF

ELSE

this->addNode(root, course)

ENDELSE

void BST::addNode(Node\* node, Course course)

IF(node != nullptr)

node->right equals new Node(course)

ENDIF

ELSE

this->addNode(node->left, course)

ENDELSE

ENDIF

Course BST::Search(string courNum)

Node\* curr == root

WHILE(curr != nullptr)

IF (courNum matches)

return curr->course

ENDIF

IF (current node compare to courNum is > 0)

current = current->left

ENDIF

ELSE

current = current->right

ENDELSE

ENDWHILE

Course course

return course

int totalPrereqs == 0

int numPrereqsCours(Tree<Course>, String courNum)

Course course = courses.search(courNum)

WHILE(course->Prereqs != 0)

FOR prereqs in course->Prereqs

Courses.search(course->prereqs->coursNum)

increment totalPrereqs

ENDFOR

ENDWHILE

void printCourseInfo(Tree<course> courses, String courNum)

Course course.search(courNum)

COUT course information

WHILE(course->Prereqs != 0)

FOR (each prereq in courses->Prereqs)

Courses.search(course->prereqs->courNum)

COUT prereq course information

ENDFOR

ENDWHILE

Course parseLine(vector<string> $line)

IF(line.size() equals 2)

Course newCourse

course.courName = line[0]

course.courNum = line[1]

set course prereq to an empty vector

return newCourse

ENDIF

ELSE

Vector<string>tmpPrereqs

FOR ( int i = 2; i < line.size(); i++)

tmpPrereqs.push\_back(line[i])

Course newCourse

course.courName = line[0]

course.courNum = line[1]

set course prereq to tmpPrereqs vector

return newCourse

int main()

Tree\* tree == l new Tree()

Vector<string>tmp

String line

IF stream infile("file name")

WHILE(getline(infile,line))

Stringstream ss(line)

WHILE(ss.good())

string substR)

getline(ss, substr, ',')

tmp.push\_back(substr)

ENDWHILE

tree.insert(parseLine(temp))

tmp.clear

ENDWHILE

ENDIF

Print Sorted List:

//Vector

printSrtd(courses)

int partition(vector& courses, int begin, int end)

Set lowIdx to first element, set highIdx to last element

Set midpoint to lowIdx + (highIdx - lowIdx) / 2

Set pivot to midpoint

Decrement highIdx

WHILE( pivot < highIdx)

Swap lower values to left of pivot, higher values to right of pivot

Set tmp to lowidx

Set lowidex to highidx

Set highidx to tmp

Void quickSrt(vector& courses, int begin, int end)

Set mid to 0, lowIdx to being, highIdx to end

IF (begin >= end)

return

Set lowEndIdx to partition (courses, lowIdx, highIdx)

quickSrt(courses, lowIdx, lowEndIdx)

quickSrt(courses, lowEndIdx + 1, highIdx)

ENDIF

Void displayCour(Course course)

COUT << course.courseId << ": " << course.name << " | " << course.prereq << endl

FOR (int i = 0; i < courses.size(); ++i)

displayCour(courses[i])

ENDFOR

Void BST::inOrder(Node\* node)

If (node != nullptr)

inOrder(node->left)

COUT << course.courId << ": " << course.name << " | " << course.prereq << endl

ENDIF

ELSE

inOrder(node->right)

COUT << course.courId << ": " << course.name << " | " << course.prereq << ENDELSE

Menu:

Set choice to 0

Menu choices (1. Load Course File, 2. Print Course List 3. Print Individual Course 4. Exit)

Case 1: loadCour(courFile, dataStruct)

Case 2: printSrtd(courses)

Case 3: printCourseInfo(courId)

Case 4: Terminate Program

Runtime Analysis

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create vector** | 1 | 1 | 1 |
| **Line in file** | 1 | n | n |
| **Course item** | 1 | n | n |
| **Create vector** | 1 | 1 | 1 |
| **While prerequisite exits** | 1 | n | n |
| **Append prerequisite** | 1 | n | n |
| **Pushback course item** | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

| **HashTable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create HashTable** | 1 | 1 | 1 |
| **Insert method** | 0 | 0 | 0 |
| **Key for course created** | 1 | n | n |
| **IF no key entry found** | 1 | n | n |
| **Node assigned to key** | 1 | n | n |
| **Else** | 1 | n | n |
| **Assign oldNode(key) to UNITMAX, set to key, set oldNode to course and oldNode next to nullptr** | 4 | N | 4n |
| **Else** | 1 | n | n |
| **Next open node** | 1 | n | n |
| **Add newNode** | 1 | n | n |
| **Line in file** | 1 | n | n |
| **Vector course item** | 1 | n | n |
| **While prerequisite exits** | 1 | n | n |
| **Append prerequisite** | 1 | n | n |
| **Insert course item** | 1 | n | n |
| **Total Cost** | | | 16n + 1 |
| **Runtime** | | | O(n) |

| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create Tree** | 1 | 1 | 1 |
| **AddNode method** | 0 | 0 | 0 |
| **IF root == null, addRoot** | 1 | 1 | 1 |
| **IF node < root, addLeft** | 1 | n | n |
| **IF no left node** | 1 | n | n |
| **Becomes left** | 1 | n | n |
| **IF node > root, addRight** | 1 | n | n |
| **For line in file** | 1 | n | n |
| **Vector course item created** |  |  |  |
| **While prerequisite exits** |  |  |  |
| **Append prerequisite** |  |  |  |
| **Insert course item** | 1 | n | n |
| **Total Cost** | 1 | n | n |
| **Runtime** | 1 | n | n |
|  | 1 | n | n |
|  | | | 11n + 2 |
|  | | | O(n) |

For the needs of the software, every data structure offers benefits and drawbacks. The benefit of the vector technique is that it reads the file and adds the course elements the quickest of all the methods. Every item is simply added to the end of a vector as the file is parsed in this extremely simple technique. Even though all three approaches used the same O(n) notation, the shortest runtime of the three was 5n+1. Searching the list for a particular course is a drawback of utilizing a vector. Until a match is found, the software must examine every element in the vector.

The ability to swiftly browse a list is one benefit of hash tables. By making a key, the locations of a particular course can be recognized and readily searched and printed. It takes longer to execute when building the initial list because a key must be made for every item and a place sought to put each course. Hash tables also don't work well for sorting. It is impossible to sort the table itself. Every value must be retrieved, sorted, and printed to provide an alpha-numeric list of all courses. Considering this, it is probably not the optimal data structure for this program.

The benefit of binary trees is that they can be searched more quickly than vectors. Knowing the search path makes it very simple to descend the tree until the value is discovered. It is speedier than a vector but not as simple as a hash table. In the worst situation, the tree would have to look through every component if it had no more leaves. In this case, the search time would be O(h), where h is the tree's height. Finally, for this job, I would advise a vector sort. I believe the client will value the ability to quickly sort and print the whole catalog. Additionally, the time lost during the search is not as awful as it could be given the benefit of the sort. The vector, in my opinion, is the finest choice overall.