6-2 Project One

# Vector Data Structure Pseudocode

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

}

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

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

//pseudocode for loading data into the vector structure Ifstream myfile

Myfile.open(“name.csv”)

IF file open fail

print error

courseNames = second string of each line WHILE (myfile.good() and not end of file)

string line; getline(myfile, line) course(line)

//pseudocode for creating course objects and storing to vector Vector<course>courses

Course(line)

String name

Course \*courseName = new Course FOR each string

courseName-> number = first parsed string courseName-> name = second parsed string IF no number or name

Error return

FOR each remaining string

IF prerequisites name found in courseNames prerequisites = remaining parsed string courseName -> prerequisites = prerequisites prerequisites ++

Courses.push\_back(\*courseName)

//pseudocode for searching and printing a specific course and printing information Cin << search item

FOR each course.at(i) with i less than course length IF course name = search item

print course information

print prerequisites course information

**//print course list** selectionSort(courses) unsigned int min

FOR i<course.size() ++i min equals i

FOR j = i + 1 j<courses.size() ++j

IF courses.at j less than courses.at i

min equals j

IF min not equal i

swap courses at i, and min

FOR i<courses.size()

print all courses.at(i)

# //print courses alphanumerically

selectionSort(courses) WHILE i<courses.size()

print sorted courses.at(i)

# Vector Run Time Analysis (Reading File and Course Objects)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **#Times Executes** | **Total Cost** |
| Ifstream  myfile(“name.csv”) | 1 | 1 | 1 |
| If file open fail | 1 | n | n |
| return error | 1 | 1 | 1 |
| courseNames =  second string | 1 | 1 | 1 |
| while(myfile.good()) | 1 | n | n |
| String substr;  getlines(myfile, line) | 2 | n | n |
| Course(line) | 1 | n | n |
|  |  | **Total Cost** | **4n + 3** |
|  |  | **Run Time** | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **#Times Executes** | **Total Cost** |
| for each string | 1 | n | n |

|  |  |  |  |
| --- | --- | --- | --- |
| courseName-> num = first string  parsed | 1 | 1 | 1 |
| courseName-> name = second  string parsed | 1 | 1 | 1 |
| return error if no course name or  num | 1 | 1 | 1 |
| For each string remaining | 1 | n | n |
| If prerequisite name found in  courseNames | 1 | n | n |
| Prerequisites = remaining string  parsed | 1 | n | n |
| courseName-> prerequisite =  prerequisite | 1 | n | n |
| Prerequisites ++ Courses.push\_back(\*courseName  ) | 1 | n | n |
|  |  | **Total Cost** | **6n + 3** |
|  |  | **Runtime** | **O(n)** |

**Hash Table Pseudocode**

Struct Course

String courseName; String courseNumber;

vector<string>prerequisite;

Const unsigned int DEFAULT\_SIZE = 8; Class HashTable

Struct Node

Course\* course; Node\*next; Unsigned int key;

Node() ///default

Node(Course course, unsigned int key) //course and course key

Unsigned int size = DEFAULT\_SIZE; Vector<Node>nodes;

HashTable();

insert(Course course);

int numPrerequisiteCourses(Hashtable courses, Course c) {

totalPrerequisites = Hashtable[c]

for each prerequisite p in totalPrerequisites

add prerequisites in Hashtable[p] to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable courses) {

for all key, value pair in courses

print key course name

if value has prerequisites

for each prerequisites

print prerequisites

}

Int main()

HashTable\* table equal new HashTable() Vector<string>temp

String line

Ifstream infile(“file name”) WHILE(getline(infile, line))

Stringstream ss(line) WHILE(ss.good())

String substr getline(ss, substr, ‘,’)

temp.push\_back(substr)

table.insert(parseLine(temp)) temp clear()

# Hash Table Runtime Analysis (Reading File and Course Objects)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line cost** | **#Times Executed** | **Total Cost** |
| Create key by hashing  courseNumber | 1 | n | n |
| while node != nullptr | 1 | n | n |
| If node→course courseNumber !=  courseNumber | 1 | n | n |
| print course information | 1 | 1 | 1 |
| for each prerequisite of the  course | 1 | n | n |
| print prerequisite  information | 1 | n | n |
|  |  | **Total Cost** | **5n + 1** |
|  |  | **Runtime** | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line cost** | **#Times Executed** | **Total Cost** |
| HashTable\* table = new HashTable() Vector<string>  temp | 2 | 1 | 1 |
| Ifstream infile(“filestream”) | 1 | 1 | 1 |
| while(getline(infile, line)) | 1 | n | n |
| Stringstream ss(line) | 1 | n | n |
| while(ss.good()) | 1 | n | n |
| String substr; getline(ss, substr, ‘,’)  temp.push\_back(substr) | 3 | n | n |
| table.insert(parseLine(temp)); | 1 | n | n |
|  |  | **Total Cost** | **5n + 2** |
|  |  | **Runtime** | **O(n)** |

**Binary Search Tree Pseudocode**

Struct Course

String courseName String courseNum

Vector<String>Prerequisite Course()

Struct Node

Course course Node\* right Node\* left

Node()

Right equals left equals nullptr

Node(Course aCourse) : Node() this→course equals aCourse

class CourseBST Add

Remove Search Inorder

void CourseBST∷PrintCourseInfo(string courseNum) this→printCourseInfo(root, courseNum)

voidCourseBST∷addNode(Node\* node, Course course) IF(node→course courseNum compares (course.courseNum) > 0

IF(node→left equals nullptr)

node left equals new Node(course) ELSE

this→addNode(node→left, course)

ELSE IF(node→right equals nullptr) node→right equals new Node(courses) ELSE

this→addNode(node→right, course)

void CourseBST∷printSampleSchedule(Node\* node) IF(node not equal nullptr)

printSampleSchedule(node→left)

print node→course.courseNum, node→course,courseName printSampleSchedule(node→right)

return

void CourseBST∷printCourseInfo(Node\* current, string courseNum) WHILE (current not equal nullptr)

IF (current→course.coureNum compares(courseNum) equals 0 print current course.courseNum, current course.courseName

unsigned int size equals NumPrerequisitesCourses(current→course) print prerequisites

unsigned int i equals 0

FOR (i equals 0, I <size i++)

print current→course.preRequisites.at(i) IF( i not equal size – 1)

print “, “ IF (i equals 0)

print no prerequisites needed return

ELSE IF (courseNum compare(current→course. courseNum) < 0 current equals current→left

ELSE

current equals current→right

int main()

Tree\* tree equal new Tree() Vector<string>temp

String line

Ifstream infile(“file name”) WHILE(getline(infile,line))

Stringstream ss(line) WHILE(ss.good())

string substr getline(ss, substr, ‘,’)

temp.push\_back(substr)

tree.insert(parseLine(temp)) temp.clear

# //prints course list

inOrder (Node\* node)

IF node not equal nullptr inOrder→node.left print course information inOrder→node.right

# //print alphanumerically

Bst→inOrder() print inOrder

# Binary Search Tree Runtime Analysis (Reading File and Course Objects)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **#Times Executed** | **Total Cost** |
| If(node courseNum  compares courseNum | 1 | 1 | 1 |
| If(node left equals nullptr) | 1 | 1 | 1 |
| add new node to left  subtree | 1 | 1 | 1 |
| else(traverse nodes left  subtree) | 1 | n | n |
| If(node right equals  nullptr) | 1 | 1 | 1 |
| add new node to right  subtree | 1 | 1 | 1 |
| else(traverse node right  subtree) | 1 | n | n |
|  |  | **Total Cost** | **2n + 5** |
|  |  | **Runtime** | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line cost** | **#Times Executed** | **Total Cost** |
| Tree\* tree = new Tree() | 2 | 1 | 1 |
| Ifstream infile(“filestream”) | 1 | 1 | 1 |
| while(getline(infile, line)) | 1 | n | n |
| Stringstream ss(line) | 1 | n | n |
| while(ss.good()) | 1 | n | n |
| String substr; getline(ss, substr, ‘,’)  temp.push\_back(substr) | 3 | n | n |
| tree.insert(parseLine(temp))  ; | 1 | n | n |
|  |  | **Total Cost** | **4n + 2** |
|  |  | **Runtime** | **O(n)** |

**Menu**

CourseBST\* coursesBST equal nullptr print Welcome this is your course planner. String option equal 0

int userOption equals option[0] – ‘0’ WHILE (userOption not equal 9)

print 1. Load Data Structure print 2. Print Course List print 3. Print Course

print 9. Exit

print Please select an option cin ≫ option

IF (userOption.length() equals 1)

userOption = option[0] – ‘0’ ELSE

userOption equal 0 bool success equal 1 switch(userOption)

case 1:

IF(courseBST equal nullptr)

courseBST equal new CourseBST() IF (csv.Path.length() equal 0)

print Please enter the file:

cin >> csvPath

correct equals loadCourses(csvPath, coursesBST) IF(correct)

print Courses have been loaded ELSE

print Error, no courses found csvPath equal “ “

break case 2:

IF(courseBST not equal nullptr and correct) print Sample course schedule coursesBST->PrintSampleSchedule()

ELSE

print loaded courses break

case 3:

IF(coursesBST not equal nullptr and correct) IF(courseId.length() equal 0)

print Please select a course for course information cin>> coursed

FOR(auto& userOption : coursed) userOption equals userOption coursesBST->PrintCourseInfo(courseId)

ELSE

print loaded course coursed equals “ “ break

default:

IF(userOption not equal 9) print option not valid break

print Thank you, Goodbye!

# Evaluation and Recommendation

Vectors have the advantage of adding an index by increasing the array size, which increases the data structure. Vectors can also be implemented easily. You can easily add items to the back of the vector. It takes time to add to the front of a vector due to the need to shift. Because of the way the algorithm loops, it may also result in slower run times. Using hash tables, you can search for matching keys quickly and have direct access to data. Regardless of the size of the table, insertions and deletions are in constant time. After each bucket is filled, searching becomes more difficult. The search for specific keys would take longer if there is more data. A hash table could consume more space than necessary. The items in a binary search tree are retrieved in order. A balanced tree is optimal for performance. The binary tree traverses in order, preorder, or postorder and inserts and deletes in O(logn). To perform well, the tree must be balanced.

There is a similar complexity and space for all the data structures. Vectors would not perform as fast as other structures. Hash tables don't provide ordered lists of data values and must be sorted linearly. This assignment requires that you display lists in alphanumeric order, so the binary search tree is the best approach. Compared to vectors or hash tables, trees make traversing course lists easier. The binary tree does not require sorting and can store custom classes with multiple fields or just one.