Brian Engel

Project One

Vector

Struct Course {

String courseNumber

String name

Vector<string> prerequisites

}

Vector<Course> readFile(string fileName) { Cost Times run

Vector<Course> courseList | 1 | 1

Ifstream file(fileName) | 1 | 1

If file didn’t open { | 1 | 1

output “could not open file” | 1 | 1

Return courseList | 1 | 1

}

String lineInFile | 1 | 1

While getline(file, lineInFile) { | 1 | n + 1

Course course | 1 | n

stringstream ss(lineInFile) | 1 | n

getline(ss, course.courseNumber, ‘,’) | 1 | n

if course.courseNumber is empty { | 1 | n

output invalid coursenumber | 1 | 1

Return courseList | 1 | 1

}

getline(ss, course.name, ‘,’) | 1 | n

if course.name is empty { | 1 | n

output invalid course name | 1 | 1

return courseList | 1 | 1

}

string prereq | 1 | n

while getline(ss, prereq, ‘,’) { | 1 + n – 1 (could technically have all courses before as prerequisites | n

if prereq in courseList.courseNumber { | n^2 (last in list) | n^3

course.prerequisite.push\_back(prereq) | 1 | n^3

}

Else {

output invalid prerequisite | 1 | 1

return courseList | 1 | 1

}

}

courseList.push\_back(course) | 1 | n

}

File.close() | 1 | 1

Return courseList | 1 | 1

}

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

}

}

}

}

Void printInOrder(Vector<Course> courses) {

Sort the vector

For each in vector {

Print the course number

Print the name

For each in prerequisite {

Print the prerequisite

}

}

}

Hash Table

Class HashTable {

Structure Node to hold a course, a key, and a pointer to the next node

A vector of nodes to hold the hashtable

Node\* InsertCourse(Course course, Node\* previousNodePtr) Function to insert a course into the hashtable: at the start of the key if the list is empty or at the end of the key if it isn’t. {

hash the course id to make a key

create a node and assign its key to the hashed key and course

if the vector of nodes at the key is empty {

assign the node to the vector at the key

}

else {

add the node to the end of the vector

}

previousNodePtr points to this node

return this nodes pointer

}

}

Struct Course {

String courseNumber

String name

Vector<string> prerequisites

}

HashTable readFile(string fileName) { Cost Times run

HashTable courseList | 1 | 1

Ifstream file(fileName) | 1 | 1

If file didn’t open { | 1 | 1

output “could not open file” | 1 | 1

Return courseList | 1 | 1

}

String lineInFile | 1 | 1

While getline(file, lineInFile) { | 1 | n + 1

Course course | 1 | n

stringstream ss(lineInFile) | 1 | n

getline(ss, course.courseNumber, ‘,’) | 1 | n

if course.courseNumber is empty { | 1 | n

output invalid coursenumber | 1 | 1

Return courseList | 1 | 1

}

getline(ss, course.name, ‘,’) | 1 | n

if course.name is empty { | 1 | n

output invalid course name | 1 | 1

return courseList | 1 | 1

}

string prereq | 1 | n

while getline(ss, prereq, ‘,’) { | 1 + n – 1 (could technically have all courses before as prerequisites | n^2

if prereq in courseList.courseNumber { | n^2 (all courses are in same key) | n^3

course.prerequisite.push\_back(prereq) | 1 | n^3

}

Else {

output invalid prerequisite | 1 | 1

return courseList | 1 | 1

}

}

InsertCourse(course, previousNodePtr) | n (all courses are in same key) | n

}

File.close() | 1 | 1

Return courseList | 1 | 1

}

void HashTable::printCourseInformation(String courseNumber) {

For all keys in the HashTable {

For all nodes in the key {

if the course is the same as courseNumber {

print out the course information

for each prerequisite of the course {

print the prerequisite course information

}

}

}

}

}

Void HashTable::printInOrder() {

Vector<courses> sortCourses

For all keys in hashtable {

For each course in the key {

SortCourses.push\_back(course)

}

}

Sort the vector

For each in vector {

Print the course number

Print the name

For each in prerequisite {

Print the prerequisite

}

}

}

Binary Search Tree

Class BST {

Tree root pointer

Structure Node to hold a course and a pointer to the left and right node

Void InsertCourse(Course course) Function to insert a course into the tree {

Set node pointer current to tree root

If current is null {

set root to new node(course)

return

}

While current is not null {

If courseId < current courseId {

If current left = null {

Current left = new node(course)

Current = null

}

Else {

Current = current left

}

}

Else {

If current right = null {

Current right = new node(course)

Current = null

}

Else {

Current = current right

}

}

}

}

}

Struct Course {

String courseNumber

String name

Vector<string> prerequisites

}

BST readFile(string fileName) { Cost Times run

BST courseList | 1 | 1

Ifstream file(fileName) | 1 | 1

If file didn’t open { | 1 | 1

output “could not open file” | 1 | 1

Return courseList | 1 | 1

}

String lineInFile | 1 | 1

While getline(file, lineInFile) { | 1 | n + 1

Course course | 1 | n

Stringstream ss(lineInFile) | 1 | n

getline(ss, course.courseNumber, ‘,’) | 1 | n

If course.courseNumber is empty { | 1 | n

output invalid coursenumber | 1 | 1

Return courseList | 1 | 1

}

getline(ss, course.name, ‘,’) | 1 | n

If course.name is empty { | 1 | n

output invalid course name | 1 | 1

return courseList | 1 | 1

}

string prereq | 1 | n

while getline(ss, prereq, ‘,’) { | 1 + n – 1 (could technically have all courses before as prerequisites | n^2

if prereq in courseList.courseNumber { | n^2 (if list is already sorted) | n^3

course.prerequisite.push\_back(prereq) | 1 | n^3

}

else {

output invalid prerequisite | 1 | 1

return courseList | 1 | 1

}

}

courseList.InsertCourse(course) | n (if list is already sorted) | n

}

File.close() | 1 | 1

Return courseList | 1 | 1

}

void BST::printCourseInformation(String courseNumber) {

current = root

while (courseNumber != current courseNumber) {

if courseNumber < current courseNumber {

current = current left

}

else {

current = current right

}

}

print out the course information

for each prerequisite of the course {

print the prerequisite course information

}

}

Void BST::printOrder(pointer node){

If node != null

PrintOrder(node left)

Print the course number

Print the name

For each in prerequisite {

Print the prerequisite

}

PrintOrder(node right)

}

Void BST::printInOrder() {

PrintOrder(root)

}

Menu Pseudocode

Void main () {

choice = 0

Get file name for data source

While choice is not 9 {

Print 1. load data

Print 2. print course list

Print 3. print course

Print 9. exit

Get choice

Switch for choice

Case 1:

Depending on which data structure you wanted to use you would only have one of the three below.

(Vector)

Vector<course> courseList = readFile(file name)

(Hash Table)

HashTable courseList.readFile(file name)

(Binary Search Tree)

BST courseList.readFile(file name)

Case 2:

Depending on which data structure you wanted to use you would only have one of the three below.

(Vector)

printInOrder(courseList)

(Hash Table)

CourseList.printInOrder()

(Binary Search Tree)

CourseList.printInOrder()

Case 3:

Get the course number to print

Depending on which data structure you wanted to use you would only have one of the three below.

(Vector)

printCourseInformation(courseList, courseNumber)

(Hash Table)

CourseList.printCourseInformation(courseNumber)

(Binary Search Tree)

CourseList.printCourseInformation(courseNumber)

}

}

There are advantages and disadvantages of using a vector for a data structure. A vector is easy to access elements by an index, so using a for loop on them is ideal. They are also easy to resize, because when you push or push\_back an element in a vector, it is automatically resized larger. A disadvantage is when you use pop or pop\_back though, because it does not automatically decrease the size of the vector, and you might end up wasting a lot of memory with a larger than needed vector. Another disadvantage is that since it is stored in contiguous memory locations, adding or deleting from anywhere but the end of the vector is slow, since it must move all the elements after the insert location. While the average case scenario for inserting in a vector is O(1) (at the end), the worst case is O(n) (at the beginning). The average case for a search is O(n/2) (in the middle) and the worst case is O(n) (at the end).

Hash tables also have several advantages and disadvantages. A hash table is easy to search, insert and delete, as long as a properly hashed key is used. It makes these functions much faster than a vector for the most part since it basically breaks the vector up into very small parts if it uses chaining to handle collisions. The main disadvantage is they can be unpredictable if you don’t know what types of values you are hashing. This can lead to a lot of collisions and an ineffective and clustered hash table. The average insert for a hash table is O(1) (no collision) and the worst case is O(n) (all values in 1 key). The average search for a hash table is O(1) (only element in key) and worst case is O(n) (all values in the key).

Binary search trees have advantages and disadvantages as well. They are very easy to search, insert, and delete nodes from, and don’t waste memory. The main disadvantage is in the creation of them. If the list is already in order, or close to it, the height of the tree ends up being the number of elements in the list, or very unbalanced. This causes the search, insert, and delete to be much less efficient. The average insert for a binary search tree is O(log n) (balanced tree) and the worst case is O(n) (list in order). The average search in a binary search tree is also O(log n) and the worst case is also O(n).

After analyzing all three of these data structures, it is a lot easier to pick the one that I would use in this instance. First, I would not use the vector, since it can eat up memory and the average runtime would be slower than the other data structures even though the worst case is the same. I also wouldn’t use the binary search tree in this instance, even though it is the one that I would have tried using before the analysis. While the average insert and search are excellent, the list is already in order which means we would be using the worst-case scenario, which is not as fast. This means that the hash table is the data structure I would use in this scenario, since we can hash the course Id easily to get unique keys and stay away from the worst-case scenario runtime.