Runtime and Memory analysis of the source code

Using vector data structure

Read File function:

Runtime cost: 6 + 1 + (n2) (parse function) + 1 + 1 = O(n2) Worst case scenario

Memory cost: O(n) because of vector with n courses

Parse function:

Runtime cost: 1 + (n \* k) + 2 + (n \* n) = (n \* k) + (n \* n) ~ k2 = O(n2) is worst case scenario

k = number of courseNames

n = number of prerequisites

Memory cost = O(1) for input line and course reference

Print Course Information function:

Runtime cost: n \* 1 + 2 + (n \* 1) + 1 = n \* k ~ O(n2) worst case scenario

n = number of courses in paraCourses

k = number of prerequisites in course.prerequisites

Using Hashtable data structure

Hash function:

Runtime: O(1)

Memory: O(1)

Figuring put the number of courses function:

Runtime: O(number of lines in file)

Memory: O(1)

Read File function:

Runtime cost: Opening the file O(1) + Reading the file line by line O(n) + Creating and inserting into the hashtable O(n) + Other operations O(1) => worst case scenario is O(n2) since there is prerequisite error checking

Memory cost: For courseListFromFile = O(n), for storing the hashtable = O(n), for other operations each = O(1), worst case scenario = O(n)

Print Course Information:

Runtime: O(1) for the course best case, O(n) worst (low prob), O(p) for the prerequisites ~ O(p) => O(p) best case, O(n \* p) ~ O(n2) worst case

Memory: O(1) for the course

Using Binary Search Tree

Insert function:

Runtime: empty tree = O(1) best case scenario, balanced nonempty tree = O(height) = O(logn), imbalanced nonempty tree = O(n) worst case scenario

Memory cost : O(1)

addCourse function:

Runtime: Recursion => O(height) = O(log n) best case

Worst case = O(n) runtime

Memory cost: O(1)

Read file function:

RUNTIME:

Reading the file line by line = O(n)

Looping over prerequisites = O(p)

Checking for courseName match = O(n)

Inserting prerequisites = O(log p) best case, O(p) worse case

Inserting courses into BST = O(log n) best case, O(n) worst case

All other operations = O(1)

Overall worst case runtime = O(n \* p + n + p) ~ O(n \* p) ~ O(n2)

MEMORY:

courseListFromFile = O(n)

PrerequisiteBST = O(p)

CourseBST = O(n)

Overall worst case = O(n + p) ~ O(n)

Print Course Information function:

RUNTIME:  
 Traversing CourseBST = O(height) best case if balanced tree, O(n) worst case

Travesring prerequisites = O(height of prerequisite tree) best case, O(p) worst case

Every other operation = O(1)

Overall best case = O(log n \* log p)

Overall worst case = O(n \* p) ~ O(n2)

Memory:

O(1) for every line

The working code that will sort and print out a list of the courses in the Computer Science program in alphanumeric order:

// prints the course number and name alphanumerically

void BSTPrintInorder(Node\* node) {

if (node) {

BSTPrintInorder(node->left);

cout << node->course->courseNumber << ", " << node->course->courseName << endl;

BSTPrintInorder(node->right);

}

}

The following will be written within a switch case

case 3:

{

if (coursesInFile == nullptr) {

cout << "\nThere is no data structure loaded into the program\n";

break;

}

cout << "What course do you want to know about? (Enter the course Id)\n";

cin >> wanted\_course;

printACourseInformation(coursesInFile, to\_uppercase(wanted\_course));

break;

}

printACourseInformation function is defined as:

// prints a selected course and its information after traversing/searching aTree

void printACourseInformation(BST\* aTree, string aCourseNumber) {

Node\* curr;

curr = aTree->root;

while (curr != nullptr)

{

if (curr->course->courseNumber.compare(aCourseNumber) == 0) {

cout << "\n" << curr->course->courseNumber << ", ";

cout << curr->course->courseName << endl;

cout << "Prerequisites: ";

if (curr->course->prerequisites.size() == 0) {

cout << "None\n";

}

for (int i = 0; i < curr->course->prerequisites.size(); i++) {

cout << curr->course->prerequisites[i];

if (i == curr->course->prerequisites.size() - 1) {

cout << "\n\n";

}

else {cout << ", ";}

};

return;

}

else if (curr->course->courseNumber.compare(aCourseNumber) > 0) {

curr = curr->left;

}

else if (curr->course->courseNumber.compare(aCourseNumber) < 0) {

curr = curr->right;

}

}

cout << "\n" << aCourseNumber << " could not be found in the data structure\n" << endl;

return;

};