**Vincent Messina**

**CS 300**

**2/12/2023**

**VECTOR DATA STRUCTURE PSUEDOCODE**

CLASS Course {

INT courseNumber;

STRING courseName;

vector<string> PreReqs;

}

Course Search (vector <Course> courses, string courseNum) {

CREATE empty course;

FOR (each Course in courses) {

IF (current courseNumber = = courseNum) {

RETURN course;

}

}

RETURN empty;

}

//Open file and store information in vector

Vector <string> OpenReadFile (string filename) {

INITIALIZE vector <string>;

INITIALIZE string line to hold single line;

INITIALIZE ifstream instream to get contents of file;

OPEN file with instream using file name;

IF (unable to open file) [

OUTPUT “Unable to open file.”;

}

pull line from instream until all information is received;

push line to back;

close file;

RETURN information;

}

//Store information from file and create a course for each line and store in vector

vector <Course\*> CreateSchedule (vector<string> contents) {

INITIALIZE vector <Course\*>;

INITIALIZE stringstream linestream;

INITIALIZE string token to store each word from line ;

INITIALIZE int count to track tokens per line;

FOR (each new course) {

set count = 0;

CREATE new Course\* course;

fill lineStream with line contents;

pull token from lineStream till end of line;

IF (count = = 0) {

course courseNumber = token;

count++;

IF (count = = 1) {

courseName = token;

count++;

}

ELSE {

}

IF (token = courseNumber already in courses) {

push token to back of course preReqs;

}

ELSE {

output “Prerequisite must be a previous course.”;

count++;

}

IF (count < 2){

OUTPUT “Error with file format. Check for course number and name.”;

empty lineStream;

push course to back of courses;

}

RETURN courses;

}

void Print (vector <Course> courses, string courseNum) {

CREATE course object;

IF (course returned is empty) {

OUTPUT “Course not in schedule.”;

RETURN;

}

OUTPUT course courseNumber and courseName;

FOR (preReq in preReqs){

OUTPUT preReq;

}

}

**HASH TABLE PSUEDOCODE**

CREATE public hash table method

DEFINE insert;

DEFINE print;

DEFINE remove;

DEFINE search;

RESIZE tableSize;

CREATE key relating to tableSize;

CREATE insert

ASSIGN current bid node to key;

SEARCH for a node using the key;

IF (no node exists) {

ASSIGN to current key;

}

ELSE {

WHILE loop to find next open node;

}

CREATE print

FOR (all values in table) {

CALL display method and display each value;

}

CREATE erase

ASSIGN user entry to current key;

ERASE node that matches the key;

CREATE search

ASSIGN user entry to current key;

IF (list is not empty) {

Point to current key;

}

Iterate through table until key is found

Return value;

}

**TREE PSUEDOCODE**

Root is equal to nullptr;

CALL inOrder function and pass root;

CALL postOrder function and pass root;

CALL preOrder function and pass root;

IF (root == nullptr) {

Root is equal to the new node bid;

}

ELSE {

Add node root and bid;

}

REMOVE node root bidId

SEARCH for a bid {

Node equal to root;

WHILE (current node = root) {

IF (current bidId == 0) {

RETURN current bid;

}

IF (current bidId < 0) {

Current = current->left;

}

ELSE {

Current = current->right;

}

}

RETURN bid;

}

ADD a bid to some node {

IF ( bidId > 0) {

IF ( no left node) {

This node becomes left node;

}

ELSE {

Recurse down left node;

}

}

ELSE {

IF (no right node) {

This node becomes right node;

}

ELSE {

Recurse down the left node;

}

}

}

inOrder {

IF (node != nullptr) {

inOrder left;

OUTPUT bidID, title, amount, and fund;

inOrder right;

}

}

postOrder {

IF (node != nullptr) {

postOrder left;

postOrder right;

OUTPUT bidID, title, amount, and fund;

}

}

preOrder {

IF (node != nullptr) {

OUTPUT bidID, title, amount, and fund;

preOrder left;

postOrder;

}

}

**PSUEDOCODE FOR A MENU**

BEGIN

WHILE (input is not 4) {

DISPLAY: “Select a menu option”

“1. Load Data Structure”

“2. Print Course List”

“3. Print Course”

“4. Exit”

IF (input is 1) {

FOR (i = 0; while i < files row count; i++) {

CourseNumber = file[i][1];

CourseNumber = file[i][0];

}

IF (more info is available after number and name) {

prerequisiteCourse = course number in spot;

}

IF (more info is still available) {

prerequisiteCourseTwo = course number in spot;

}

}

IF (input is 2) {  
 START at head node;

WHILE (currentNode is not nullptr) {

IF (node > currentNode) {

Move to front;

}

IF (node < currentNode) {

Keep behind;

}

}

PRINT list from head tail;

}

IF (input is 3) {

INPUT courseNumber;

FOR (all courses) {  
 IF (course is the same as courseNumber) {  
 PRINT course information;

FOR (each prerequisite of the course) {  
 PRINT prerequisite information

}

}

}

}

IF (input is 4) {

PRINT “Goodbye”;

}

END

}

**PSUEDOCODE TO PRINT AN ALPHANUMERIC ORDER OF COURSES**

START at head node;

WHILE (currentNode is not nullptr) {  
 IF (node > currentNode) {  
 move in front;

}

IF (node < currentNode) {

Keep behind;

}

}

PRINT list from head tail;

**EVALUATION**

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each course in courses** | 1 | n | n |
| **if current course number == course number** | 1 | n | n |
| **Return course** | 1 | 1 | 1 |
| **For each new course** | 1 | n | n |
| **Set count = 0** | 1 | 1 | 1 |
| **If count = 0** | 1 | n | n |
| **Course courseNumber = token** | 1 | 1 | 1 |
| **Count++** | 1 | 1 | 1 |
| **If count == 1** | 1 | n | n |
| **Course name = token** | 1 | 1 | 1 |
| **Count++** | 1 | 1 | 1 |
| **If token = courseNumber already in course** | 1 | n | n |
| **Push token back to course prerequisites** | 1 | 1 | 1 |
| **Else** | 1 | n | n |
| **Output “prerequisite must be a previous course”** | 1 | 1 | 1 |
| **Count++** | 1 | 1 | 1 |
| **If count < 2** | 1 | n | n |
| **Output “error with file format”** | 1 | 1 | 1 |
| **Count++** | 1 | 1 | 1 |
| **If course returned is empty** | 1 | n | n |
| **Output “course not in schedule** | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 12 |
| **Runtime** | | | O(n) |
|  | | |  |

| **Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If no node exists** | 1 | n | n |
| **Assign to current key** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **While loop to find next open node** | 1 | n | n |
| **For all values in table** | 1 | n | n |
| **Call display method and display each value** | 1 | 1 | 1 |
| **If list Is not empty** | 1 | n | n |
| **Point to current key** | 1 | 1 | 1 |
| **Iterate through table until key is found** | 1 | n | n |
| **Return value** | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 5 |
| **Runtime** | | | O(n) |
|  | | |  |

| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If root = nullptr** | 1 | n | n |
| **Root = new node bid** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Add node root and bid** | 1 | 1 | 1 |
| **While current node = root** | 1 | n | n |
| **If current bidId == 0** | 1 | n | n |
| **Return current bid** | 1 | 1 | 1 |
| **If current bidId < 0** | 1 | n | n |
| **Current = current->left** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Current = current->right** | 1 | 1 | 1 |
| **Return bid** | 1 | 1 | 1 |
| **If bidId > 0** | 1 | n | n |
| **If no left node** | 1 | n | n |
| **This node becomes left node** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Recurse down left node** | 1 | 1 | 1 |
| **If node != nullptr** | 1 | n | n |
| **inOrder left** | 1 | 1 | 1 |
| **Output bidId, title, amount, and fund** | 1 | 1 | 1 |
| **inOrder right** | 1 | 1 | 1 |
|  |  |  |  |
| **Total Cost** | | | 7n + 14 |
| **Runtime** | | | O(n) |

All three of my psuedocodes have a Linear runtime complexity of O(N). They all loop through the file linearly as many times as needed. The runtime complexities are all the same because each item is created one at a time. Each of those items are searched with a linear runtime complexity of O(N) as well.

Vectors can insert items at O(1) to add items quick and efficiently. Vectors are also the easiest to code compared to the hash table and trees. Vectors are also scalable, easy to edit, and generally have small file sizes. However, when using is a long list, a vector will take a long time to search through that list. Hash tables are much quicker than vectors and trees. Hash tables provide constant time for searching, insertion, and deletion operations. One disadvantage of a hash table is that it is difficult to code. These are much more complex compared to vectors. Using trees can be easier than using a hash table. Trees are also much quicker than vectors, but the hash table is still faster.

My recommendation would be to use a hash table. Hash tables are the most efficient algorithm of the three. If the hash table is too difficult to implement, I would choose the tree over the vector. Vectors are the worst algorithm of the three.