# Philip Trinh

SNHU/CS300

04/06/2023

# CS 300 Pseudocode Document

// OPEN AND READ FILE, PARSES EACH LINE, AND CHECKS FOR FILE FORMAT ERRORS

Open file “File Name”

If file does not exist, print “file not found, please check name”

Else

While it is not end of file

For each line of the file

Strip white space from beginning and end of line

Split the line into parameters by the commas

If less than two parameters return from each line

Return ERROR, invalid format

Else

Format is valid, read the parameters

If the third or more parameters exist

Compare if it exist at the beginning of the line somewhere else

Continue

Else

Return Error, prerequisite not in list

Close file

//VECTOR

//CREATE COURSE OBJECTS AND STORE THEM IN THE APPROPRIATE DATA STRUCTURE

void courseObject(Vector<Course> courses, String courseNumber, String courseName, String <vector> prerequisite) {

Vector<Course> loadCourse (course)

Open course file “File Name”

Course course

for each line read

add courseNumber = first parameter

add courseName = second parameter

if there are more than two parameters

add to courseObject as prerequisites

append courseObjects to vector

}

return <vector> courses

//ORDEREDLIST QUICK SORT AND SELECTION SORT

Declare integer for partition (vector, begin, end)

Define variable for begin (low)

Define variable for end (high)

Set the middle element in the index as pivot point [low + (high – low) /2]

BOOL Done is FALSE

WHILE not done

Keep incrementing low index while bids[low] < bids[pivot]

Keep decrementing high index while bids[pivot] < bids[high]

If

there are zero or one elements remaining, Bool Done is TRUE

Return High

Else

Swap low and high bids and increment low while decrementing high

Void quickSort (vector, begin, end)

Set midpoint equal to zero

If

begin greater or equal to end

return

recursively sort low partition (begin to mid) and high partition (mid +1 to end)

void selectionSort(vector<Course> & course)

Define integer minimum

Define integer maximum equal to bids size

Set position within the bids that divides sorted/unsorted

For position equal 0, position is less than bids size, increment position.

Set minimum equal to position.

(Move to remainder position on the right by)

For position + 1, position is less than bids size - 1, increment position

If this element is less than the minimum title

Set minimum equal to this position.

Repeat this and swap the current minimum with smaller one found.

//VECTOR PRINT OUT COURSE INFORMATION AND PREREQUISITES

void printOrderedList(Vector<Course> courses)

Call quickSort method //or selectionSort method if prefer

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) {

for all courses

print courseNumber and CourseName

if course have prerequisites

for each prerequisite

print prerequisites

}

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

}

//HASHTABLE

//CREATE COURSE OBJECTS AND STORE THEM IN THE APPROPRIATE DATA STRUCTURE

Create Structures to hold course, struct Node {}

Course course

Unsigned int key

Node\*next

Create default constructor Node() {

Key = UINT\_MAX

Next = nullptr

Initialize with course and a key

Node (Course aCourse, unsigned int aKey) : Node (aCourse) {

Key = akey

}

Define tableSize

Unsigned int hash(int key)

Create default constructor for HashTable

Initialize node structure by resizing tableSize

Invoke local tableSize with this->

Resize node size

Create Destructor

Erase nodes beginning

Implement logic to calculate hash value by

Define Unsigned int hash(int key)

Return key % tableSize

Create insert method void HashTable

create the key for the given course, courseNumber, courseName, with the key value

Retrieve node using key

if no entry found for the key

assign this node to the key position

else if node is not used

assign old node key to UINT\_MAX, set to key, set old node to course, update with the actual course and old node to next nullptr

else if not found find the next open node available

add new newNode to end

void courseObject(Hashtable<Course> courses, String courseNumber, String courseName, String prerequisite) {

Hashtable<Course> loadCourse (course)

Open course file “File Name”

Course course

for each line read

add courseNumber = first parameter

add courseName = second parameter

if there are more than two parameters

add to courseObject as prerequisites

Hashtable -> Insert(courseObject)

//HASHTABLE PRINT OUT COURSE INFORMATION AND PREREQUISITES

int numPrerequisiteCourses(Hashtable<Course> courses) {

Take input for courseNumber = couse c

Assign key = courseNumber

Assign node to the node at key

currentList = Hashtable of item pointing to key

If current node pointing to currentList match course c inputted and not null

Return prerequisites of course c

Else

Return null

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

for all courses from node begin to end

print courseNumber and CourseName

if course have prerequisites

for each prerequisite

print prerequisites

node is equal to next node

}

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

for all courses

get course at table index of hashed courseNumber

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

get course at table index of Hashtable prerequisites (courseObject)

print the prerequisite course information

}

//BinarySearchTree

//CREATE COURSE OBJECTS AND STORE THEM IN THE APPROPRIATE DATA STRUCTURE

create struct Course {

string courseNumber

string courseName

string prerequisite

}

create struct Node {

Course course

Node left

Node right

}

Reference tree: insert (Course course)

If course is empty

Root = add new node

Else

This -> addNode root and course

BinarySearchTree <course> addNode(Node type node, Course course) {

Implement inserting course into BinarySearchTree by

If node is larger than root then add to the left

If no left node

This node become left node

Else recurse down the left node

Else if node is less than root then add to right

if no right node

this node become right

Else recurse down the right node

//ORDEREDLIST INORDER

Tree in inOrder(Node type node)

If node is not equal to nullptr

inOrder node point to left

output courseNumber, courseName, prerequisite

inOrder node point to right

void courseObject(BinarySearchTree <Course> courses, String courseNumber, String courseName, String prerequisite) {

BinarySearchTree <Course> loadCourse (course)

Open course file “File Name”

Course course

for each line read

add courseNumber = first parameter

add courseName = second parameter

if there are more than two parameters

add to courseObject as prerequisites

BinarySearchTree -> addNode (courseObject)

// BINARYSEARCHTREE PRINT OUT COURSE INFORMATION AND PREREQUISITES

void printOrderedList(BinarySearchTree <Course> courses)

call inOrder method

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrerequisites = combine of left and right children of courses

for each prerequisite p in totalPrerequisites

add left and right nodes to the totalPrerequisites

print totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

for all Nodes as course

print courseName

if there is left node or right node

print node as prerequisites

}

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

for all nodes

if the course is same as courseNumber

print out the course information

if the course has left or right node

print as prerequisite information of that course

}

//DISPLAY COURSE

Void displayCourse (Course course) {

cout << course.courseNumber << ": " << course.courseName << " | " << course.prerequisites<< endl; }

return;

Loop through Vector/BinarTree to display courses

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

displayCourse(Course course)

//CREATE MENU

VOID MENU

Int choice = 0

While choice not equal to 0

Print “Menu”

“1. Load Course File”

“2. Print Ordered List of Courses”

“3. Print Individual Course”

“4. Exit”

switch(choice){

Case 1: call loadCourse (File hold course) //choose a dataStructure

Implement ticks = clock – ticks

Print time of clock ticks

Print time after convert ticks to clock

break

Case 2: printOrderedList(courses)

call method printOrderedList from either Vector or BinarySearchTree dataStructure

break

Case 3: printCourseInformation

Call method printCourseInformation(courseNumber) from chosen dataStruct

break

}

Print “Goodbye” //this will automatically exit(break)if option other than 1-3 choices are chosen.

return 0;

//RUNTIME ANALYSIS

| **OPEN FILE**, **READS THE DATA FROM THE FILE, PARSES EACH LINE, AND CHECKS FOR FORMATTING ERRORS** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| OPEN FILE | 1 | 1 | 1 |
| For each line of the file | 1 | n | n |
| Strip white space from beginning and end of line | 1 | n | n |
| Split the line into parameters by the commas | 1 | n | n |
| If less than two parameters return from each line | 1 | n | n |
| Return ERROR, invalid format | 1 | 1 | 1 |
| Else | 1 | n | n |
| Format is valid, read the parameters | 1 | n | n |
| If the third or more parameters exist | 1 | n | n |
| Compare if it exist at the beginning of the line somewhere else | 1 | n | 1 |
| Continue | 1 | n | n |
| Else | 1 | n | n |
| Return Error, prerequisite not in list | 1 | 1 | 1 |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 5 |
| **Runtime** | | | O(n) |

| **VECTOR, READ FILE & CREATE COURSE OBJECTS, ONE COURSE OBJECT HOLDS DATA FROM A SINGLE LINE FROM THE INPUT FILE.** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open course file “File Name” | 1 | 1 | 1 |
| Create course object | 1 | 1 | 1 |
| For each line read | 1 | n | n |
| Add courseNumber = first parameter | 1 | n | n |
| Add courseName = second parameter | 1 | n | n |
| If there are more than two parameters | 1 | n | n |
| Add to courseObject as prerequisites | 1 | n | n |
| Append courseObjects to vector | 1 | n | n |
| Return <vector> courses | 1 | 1 | 1 |
| **Total Cost** | | | 6n + 3 |
| **Runtime** | | | O(n) |

| **HASHTABLE – INSERT METHOD** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| CREATE HASHTABLE | 1 | 1 | 1 |
| Create insert method void HashTable | 1 | 1 | 1 |
| Create the key for the given course, courseNumber, courseName, with the key value | 1 | n | n |
| Retrieve node using key | 1 | n | n |
| If no entry found for the key | 1 | n | n |
| Assign this node to the key position | 1 | n | n |
| Else if node not used | 1 | n | n |
| Assign old node key to UINT\_MAX, set to key, set old node to course, update with the actual course and old node to next nullptr | 4 | n | 4n |
| Else, if not found find the next open node available | 2 | 2n | 2n |
| Add new newNode to end | 1 | n | n |
| **Total Cost** | | | 12n+ 2 |
| **Runtime** | | | O(n) |

| **HASHTABLE READ FILE & CREATE COURSE OBJECTS, ONE COURSE OBJECT HOLDS DATA FROM A SINGLE LINE FROM THE INPUT FILE.** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| open courses file “fileName” | 1 | 1 | 1 |
| Create course object | 1 | 1 | 1 |
| for each line read | 1 | n | n |
| Add courseNumber = first parameter | 1 | n | n |
| Add courseName = second parameter | 1 | n | n |
| If there are more than two parameters | 1 | n | n |
| Add to courseObject as prerequisites | 1 | n | n |
| Hashtable -> Insert(courseObject) | 12n+2 | n | 13n+2 |
| **Total Cost** | | | 18n+ 4 |
| **Runtime** | | | O(n) |

| **BINARY SEARCH TREE ADDNODE** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create BinarySearchTree addNode method | 1 | 1 | 1 |
| If root is empty, add root | 1 | n | n |
| Else | 1 | n | n |
| If node is larger than root, then add to the left | 2 | n | n |
| If no left node | 1 | n | n |
| This node become left node | 1 | n | n |
| Else recurse down the left node | 1 | n | n |
| Else if node is less than root, then add to right | 2 | n | n |
| if no right node | 1 | n | n |
| this node become right | 1 | n | n |
| Else recurse down the right node | 1 | n | n |
| **Total Cost** | | | 10n+ 1 |
| **Runtime** | | | O(n) |

//ADVANTAGES AND DISADVANTAGES OF EACH STRUCTURE

According to the time analysis using big O notation above. The run time for all three (vector, hash table, and binary search tree) are the same O(n). However, the vector method will be slightly faster in comparison due to less operation needed to add course into vector and load it the total cost estimated is 6n + 3. A disadvantage with vector is printing out specific course information. The vector needs to go through each list to find the course, resulting in slower search time than hash table or binary search tree.

Hash tables’ disadvantage is it cannot be used to sort the list. Hash table put the object into empty slot that corresponding to the hash value logic. Another disadvantage is that the hash table is slowest in comparison to the other two due to longer check of logic to ensure key are assigned to each element while creating the listing, resulting in total cost 18n + 4. However, this disadvantage is also one benefit of using a hash table since each of the assigned keys helped to locate the item at constant-time without relying on another form of search algorithm which makes it the fastest in searching out of the three structures.

Binary search tree’s advantage is that it uses a system of elimination which categorizes the search at greater or lesser than the root. This helps it to be able to do specific search for course faster than vector. However, a disadvantage of binary search tree would be when more nodes are added or removed, the structure becomes unbalance and some of the branches get deeper than others which will likely to encounter worst-case search time and become much less efficient.

#RECOMMENDATION FOR WHICH DATA STRUCTURE PLAN TO USE FOR CODING

The client wanted to sort and print an alphanumerically ordered list of all the courses, so Hash Table won’t meet this requirement. Also Hash Table is the slowest out of the three on adding course object into hash table, so I don’t want that.

Vector is great for the addition of courses with the fastest time but will not sustain once the array gets bigger and more complex. If there is frequently search or data in sorted manner, vector will not be nearly as efficient as binary search tree.

In conclusion, I planned on using binary search tree for my coding project. Not only that it is great in storing and sorting elements, but it is also quite efficient in searching for elements. Although a bit slower than vector in adding elements, it is a very organized structure and would be best fit for the requirements of the Computer Science department at ABCU.