# CS 300 Pseudocode Document

**Andrei Kourouchin**

## Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

CS 300 Pseudocode Document

Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

// Vector pseudocode

struct Course{

string courseid

string coursename

vector<string> totalPrequisites

}

vector<Course> courses

void loadcourses(string filepath){

use csv parser to open filepath

for each line in the csv

if the line has two elements or less

print "Line ", line number, " has less than two elements and is invalid"

else

create a new Course object

Course course

course.courseid will be first element

course.coursename will be second element

while there is another element in the line

add element to the end of the totalPrerequisite vector

pushback to the courses vector

for each object in courses

for each item in totalPrerequisites

if item is not in courses.courseid

print item is invalid because the prerequisite does not exist.

}

create partition function that takes a course vector, begin integer and end integer

set low and high equal to begin and end

create a pivot string

pick the middle element as the pivot point

while not done

increment low index while course index is lower than course pivot

increment high index while ocurse index is higher than course pivot

if there are 0 or one elmeents remaining

all bids are partitioned

return high

else

swap the low and high courses

decrement high

increment low

return high

create quicksort function that takes a course vector, begin integer and end integer

set mid equal to 0

if there are 1 or 0 bids to sort partition is already sorted

otherwise if begin is greater than or equal to end

then return

partition bids into low and high

recursively sort low poartition

recursively sort high partition

create print all function

run quicksort function if it has not been run

for all items in sorted courses

print course information

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 "Course id:", courseid, " | ", "Course Name: ", coursename, " | "

for each item in totalPrerequsites

print item ", "

print new line

}

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 menu {

print out option menu for 4 choices below

create a switch case

case 1:

call load data function

break

case 2:

call course list function

break

case 3:

print "What course would you like to search for?"

course number = ask for input for course user wishes to search for

call course print function and pass course number

break

case 4:

exit the program

}

Main {

call menu function

}

// Hashtable pseudocode

struct Course{

string courseid

string coursename

vector<string> totalPrerequisites

}

class HashTable{

private:

struct Node{

Course course

unsigned int key

Node\* next

Create a Node constructor

key is default UNIT\_MAX

next is nullptr

Initialize with a course

public:

Insert(Course course)

unsigned int tableSize = 500

}

}

vector<Course> courses

unsigned int hash function takes int key as parameter{

return key % tablesize

void HashTable Insert(Course course){

create a key for the given bid using hashtable

retrieve node using key

if no entry found for the key

insert at the key position

else

find the next open node

insert new node to the end

}

create search function that takes courseid as parameter{

Course course

hash the courseid to create key

create a node for key node

if entry found for the key

return node course

if no entry found for key

return course

while node not equal to nullptr

if the current node matches

return current node

node is equal to next node

return course

}

void loadcourses(string filepath){

create key by hashing our courseNumber

retrieve node using key

use csv parser to open filepath

for each line in the csv

if the line has two elements or less

print "Line ", line number, " has less than two elements and is invalid"

else

create a new Course object

Course course

course.courseid will be first element

course.coursename will be second element

while there is another element in the line

add element to the end of the totalPrerequisite vector

pushback to the courses vector

for each object in courses

for each item in totalPrerequisites

if item is not in courses.courseid

print item is invalid because the prerequisite does not exist.

insert course into hash table

create function PrintAll

create a new node

create a new course

for loop to iterate beginning to end of course vector

find memory address of index node

if the key is not equal to default value

output course information

while node is not equal to null pointer

set the course pointer to the next node pointer

print out course information

set node to be equal to the next node

int numPrerequisiteCourses(Hashtable<Course> courses) {

integer numPrerequisiteCourses = 0

for item in courses

if item is in the hashtable

if totalPrerequisites is not empty

numPrerequisiteCourses++

return numPrerequisiteCourses

}

void printSampleSchedule(Hashtable<Course> courses) {

for all courses

print "Course id:", courseid, " | ", "Course Name: ", coursename, " | "

for each item in totalPrerequsites

print item ", "

print new line

}

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

create key by hashing our courseNumber

for item in hashtable

if entry is found for key

return key

if no entry is found for key

return bid

while node is not equal to null pointer

if the current node matches,

return it

node is equal to next node

}

void menu {

print out option menu for 4 choices below

create a switch case

case 1:

call load data function

break

case 2:

call PrintAll function

break

case 3:

print "What course would you like to search for?"

course number = ask for input for course user wishes to search for

call course print function and pass course number

break

case 4:

exit the program

}

Main {

call menu function

}

// Tree pseudocode

struct Course{

string courseid

string coursename

vector<string> totalPrequisites

}

struct Node{

Course course

Create pointer for values < parent

Create pointer for values > parent

Create default constructer

left = right = nullptr

}

Class BST{

create private functions

Node\* root

void insert

void search

create public functions

void BSTsearch

void BSTinsert(Course course)

}

void BSTinsert{

if root is equal to null pointer

root is equal to new node course

else

add node root and course

}

}

Create inorder function that takes a node as a parameter

if node is not equal to null pointer

recursive call to inorder function with left node

print out course informatoin

recursive call to in order function with right node

void loadcourses(string filepath){

BinarySearchTree\* Tree

Tree tree

tree = new BinarySearchTree

use csv parser to open filepath

for each line in the csv

if the line has two elements or less

print "Line ", line number, " has less than two elements and is invalid"

else

create a new Course object

Course course

course.courseid will be first element

course.coursename will be second element

while there is another element in the line

add element to the end of the totalPrerequisite vector

pushback to the courses vector

for each object in courses

for each item in totalPrerequisites

if item is not in courses.courseid

print item is invalid because the prerequisite does not exist.

insert course into binary tree

int numPrerequisiteCourses(Tree<Course> courses) {

integer numPrerequisiteCourses = 0

for item in courses

if item is in courses

if totalPrerequisites is not empty

numPrerequisiteCourses++

return numPrerequisiteCourses

}

void printSampleSchedule(Tree<Course> courses) {

if node is not equal to null pointer

recursive traverse left branch

print course information

recursive traverse right branch

}

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

set current node equal to root

loop through the binary tree until bottom reached or matching courseid found

if match found

return current course

if course is smaller than current node

recursive BSTsearch call traverse left

else course is larger

recursive BSTsearch call traverse right

}

void menu {

print out option menu for 4 choices below

create a switch case

case 1:

call load data function

break

case 2:

call in order function

break

case 3:

print "What course would you like to search for?"

course number = ask for input for course user wishes to search for

call course print function and pass course number

break

case 4:

exit the program

}

Main {

call menu function

}

## Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document.

| **Vector algorithm** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Set high and low equal to beg and end | 1 | 1 | 1 |
| Create a pivot string | 1 | 1 | 1 |
| **Pick middle element** | 1 | 1 | 1 |
| While not done | 1 | n | N |
| Increment low index while course index is lower than course pivot | 1 | 1 | 1 |
| Increment high index while course index is higher than course pivot | 1 | 1 | 1 |
| **If there are 0 or 1 elements remaining** | 1 | N | n |
| **Return high** | 1 | 1 | n |
| **Else** | 1 | n | N |
| **Swap low and high courses** | 1 | 1 | 1 |
| **Decrement high** | 1 | 1 | 1 |
| **Increment low** | 1 | 1 | 1 |
| **Return high** | 1 | 1 | 1 |
| **Set mid equal to 0** | 1 | 1 | 1 |
| **If there are 1 or 0 bids to sort partition is already sorted** | 1 | n | n |
| **Else if begin is greater than or equal to end** | 1 | n | N |
| **Return** | 1 | 1 | 1 |
| **Partition bids into low and high** | 1 | 1 | 1 |
| **Recursively sort low partition** | 1 | 1 | 1 |
| **Recursively sort high partition** | 1 | 1 | 1 |
| **Run quicksort function if it has not run** | 1 | 1 | 1 |
| **For all items in sorted courses** | 1 | n | N |
| **Print course information** | 1 | 1 | 1 |
| **Total Cost** | | | 7n + 16 |
| **Runtime** | | | O(16+7n) |

| **BST algorithm** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create BST Insert function | 1 | 1 | 1 |
| If root is equal to null pointer | 1 | n | n |
| **Root is equal to new node course** | 1 | 1 | 1 |
| Else | 1 | n | n |
| Add node root and course | 1 | 1 | 1 |
| Create in order function that takes node as a parameter | 1 | 1 | 1 |
| **If node is not equal to null pointer** | 1 | n | n |
| **Recursive call to in order function with left node** | 1 | N | N |
| **Print out course information** | 1 | 1 | 1 |
| **Recursive call to inorder function with right node** | 1 | n | n |
| **Total Cost** | | | 5n+5 |
| **Runtime** | | | O(5n+5) |

| **HashTable algorithm** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create hash function that takes key as parameter | 1 | 1 | 1 |
| Return key % tablesize | 1 | 1 | 1 |
| Create hashtable insert function that takes course as parameter | 1 | 1 | 1 |
| Create a key for the given bid using hash function | 1 | 1 | 1 |
| **Retrieve node using key** | 1 | 1 | 1 |
| **Create search function that takes coursed as parameter** | 1 | 1 | 1 |
| **Course course** | 1 | 1 | 1 |
| **Hash the coursed to create key** | 1 | 1 | 1 |
| **Create a node for key node** | 1 | 1 | 1 |
| **If entry found for key** | 1 | n | N |
| **Return node course** | 1 | 1 | 1 |
| **If no entry found for key** | 1 | n | N |
| **Return course** | 1 | 1 | 1 |
| **While node not equal to nullptr** | 1 | n | N |
| **If current node matches** | 1 | n | N |
| **Return current node** | 1 | 1 | 1 |
| **Node is equal to next node** | 1 | 1 | 1 |
| **Return course** | 1 | 1 | 1 |
| **Total Cost** | | | 4n+14 |
| **Runtime** | | | O(4n+14) |

| **Sample** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Reflection:

We have used three data structures, a vector, hash table and tree in this exercise. There are a few distinct advantages and disadvantages of each structure. I think starting off with a vector, we have a large disadvantage in that it requires every item to be looked at as we sort it. This results in an inefficient algorithm that performs poorly. This is evidenced by the runtime analysis chart that we created above. In the case of being as efficient as possible, the winner of that contest would be the Hash table. That is also one of the main advantages of the Hash table which is that it is very quick to both search as well as manipulate the data. The hash table takes a first place compared to the BST. Hash tables are very simple and are relatively easy to implement, which makes them an ideal candidate for key-to-value lookup. Their operations are also very efficient typically taking constant time. Unfortunately, operations in a hash table can degrade as the number of collisions take place, so careful programming is a necessity. BSTs are very efficient in insertion and deletion when they’re balanced, and their code is very efficient. However searching takes a long time.

I will recommend that we use a hash table for this software as we are not too worried about collisions for keys. The main disadvantage of a hash table only comes into play when we have lots of similar values, but since we don’t have those in our data set, we do not have to worry about that as much and can get the most value out of our hash table. The efficiency and relative simplicity of a hash table makes it a very attractive data structure for our purpose.