# CS 300 Pseudocode Document

Pseduocode

// vector

vector<Course> loadCourses(string csvPath)

for (int i = 0; i < file.rowCount(); i++) {

Create data structureadd to collection

Course course;

course.courseId = file[i][1];course.name = file[i][0];

while not end of line

course.prereq. = file[i][8];

courses.push\_back(course);

// HashTables

Create Hashtable

Create Node struct

Course course

Unsigned int key Vector<Node> nodes

Define tableSize

Unsigned int has(int key)

Create insert method void HashTable::Insert(Course course)

create the key for the given course, search for node with the key value

if no entry found for the key

assign this node to the key position

else if node is used

old node key to UNIT\_MAX, set to key,

else find the next open node add new newNode to end

void loadCourses(string csvPath, HashTable\* hashTable) loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

Create data structure add to collection Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line course.prereq. = file[i][8];

hashTable->Insert(course);

//tree

Define a binary search tree to hold all courses

BinarySearchTree\* bst;

bst = new BinarySearchTree();

Course course;

If root is null, add root

if node is less than

then add to left

if no left node

node is left

if node is greater than root add right

this node becomes right

void loadCourses(string csvPath, BinarySearchTree\* bst) loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

Create a data structure add to collection of Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line course.prereq. = file[i][8];

bst->Insert(course);

Printing courses info and prereq

// Vector

Create method void printCourseInformation(Vector<Course> courses, String courseId)

Get input for courseId

While vector is not empty

if the input is the same as courseId

output course.courseId << output course.name

while (prereq = true)

output course.prereq

// Hashtable

Create method void printCourseInformation(Hashtable<Course> courses, String courseId)

Get input for courseId

Assign key = courseId

Assign node to the node.at(key)

if current node matches key

Return course, displayCourse(nodes[key].course)

If node points to null, return null

Else while the node is not Null, check against the key

If the key matches the couseId, Return course, displayCourse(nodes[key].course)

Point to next node

// Tree

Create method void printCourseInformation(Tree<Course> courses, String courseId)

Get input for courseId

Assign current node to root

While current is not NULL

If course.courseId matches current

Return current, output course.courseId << output course.name

while (prereq = true)

out put course.prereq

If courseIid is less than root

Set current to left

Else set current to right

Print sort list:

//Vector

Create sorted print method printSorted(courses)

Create partition method int partition(vector<Course>& courses, int begin, int end)

Set lowIndex to first element, set highIndex to last element

Set midpoint to lowIndex + (highIndex - lowIndex) / 2

Set pivot to midpoint

Decrement highIndex while pivot is less than highIndex

Swap lower values to left of pivot, higher values to right of pivot

Set temp value to low index

Set low index to high index

Set high index to temp

Create quicksort method void quickSort(vector<Course>& courses, int begin, int end)

Set mid to 0, lowIndex to being, highIndex to end

If begin >= end, return

Set lowEndIndex to partition(courses, lowIndex, highIndex)

Make recursive call to quicksort quicksort

(courses, lowIndex, lowEndIndex);

quickSort(courses, lowEndIndex + 1, highIndex)

Create display course method void displayCourse(Course course) {

cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

Loop through vector to display courses

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

displayCourse(courses[i])//Tree

Create inOrder method void BinarySearchTree::inOrder(Node\* node)

If (node != Nul)

Check most left side first

inOrder(node->left)

cout << course.courseId << ": " << course.name << " | " << course.prereq << endl; check next right leaf

inOrder(node->right)

cout << course.courseId << ": " << course.name << " | " << course.prereq << endl;

Menu print :

Set to 0

loop menu if not equal to 4

menu choices (1. Load Course, 2. Print sort class 3. Print Course info 4.End)

1: loadCoursesbinary search tree

2: function to print sorted class list

3: printCourseInformation

4: End Program

Runtime

| **Vector Code** | **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** | 2 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 2 | n | n |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | 1(n) |
| **\** | | |  |

| **Hashtable Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 2 | 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** | 2 | n | n |
| **print the prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

| **Tree Code** | **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** | 2 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 8n + 1 |
| **Runtime** | | | O(n) |

Pros and cons of each structure

Vector:

Pros- constant inseration

* Easy to put in

Cons- have to be sorted before using

* May be using more space than is needed

HashTable:

Pros- Fast when used correctly

* Delete or insert

Cons- Takes up space

* Could make long delays from random stores

Tree:

Pros – speed

* Allowed to get items in order

Cons- first item determines the shape

* Needs balance for efficient output

I personally use the binary search tree for holding course objects. When displaying courses in alphabetic order, the binary search tree does better than the other data structures. The search tree does not need any sorting since it goes in order. Vectors and hash tables don’t have sorting abilities to display courses in alphabetic order.