# CS 300 Pseudocode Document

// Menu pseudocode

Void Main() {

//dataStructure below would depend on what was used, vector, //hashtable or tree

dataStructure courseData

int selectedOption

while (selectedOption != 9){

print “1) Load Data”

print “2) Print course list”

print “3) Print course”

Print “9) Exit”)

Input selectedOption

Switch (selectedOption) {

Case 1:

couseData = loadCourseData(“filename”)

break

Case 2:

printCourseList(courseData)

break

case 3:

input selectedCourseName

PrintCourse(courseData, selectedCourseName)

break;

}

}

}

// Load courses from file pseudocode

// Vector version

Vector<Course> loadCourseData(String fileName) {

courseFileStream = open(filename)

if CourseFileStream not open {

Print error

Return

}

Vector<Course> courses

while no error reading from file stream {

thisLine = read next line

courseNumber = parse (beginning of string, first comma)

courseName = parse (first comma, second comma or end of string)

while not end of thisLine

prerequisite = parse (current comma, next comma or end of string)

// assumes course are in order where prerequisites are always listed before

// courses that need them

foreach prerequisite in prerequisites {

search courses vector to ensure course exists

if requisite not in courses

Print error with courseNumber

}

Add new Course(courseNumber, courseName, prerequisites) to courses Vector

}

return courses

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Opening file, initializing vector | 3 | 1 | 3 |
| Reading each course line | 4 | n | 4n |
| Loading prerequisites | 2 | n | 2n |
| Add course to vector | 1 | n | n |
| Return courses | 1 | 1 | 1 |
| Total Cost | | | 7n + 4 |
| Runtime | | | O(n) |

Course UML diagram

A picture containing text

Description automatically generated

Creating a Course object:

Courses can either use the default constructor, making a Course object that contains no data (but it can be set afterwards), or use the overloaded constructor. This passes the course number and name (both as Strings), and the prerequisites, which are an array of strings, each one of which is a course number String. This array could have 0, 1 or 2 entries.

// Load courses from file pseudocode

// Hashtable version

Hashtable loadCourseData(String fileName) {

courseFileStream = open(filename)

if CourseFileStream not open

Print error

Return

Hashtable courses

while no error reading from file stream {

thisLine = read next line

courseNumber = parse (beginning of string, first comma)

courseName = parse (first comma, second comma or end of string)

while not end of thisLine

prerequisite = parse (current comma, next comma or end of string)

// assumes course are in order where prerequisites are always listed before

// courses that need them

foreach prerequisite in prerequisites {

search courses vector to ensure course exists

if requisite not in courses

Print error with courseNumber

}

Create new Course(courseNumber, courseName, prerequisites)

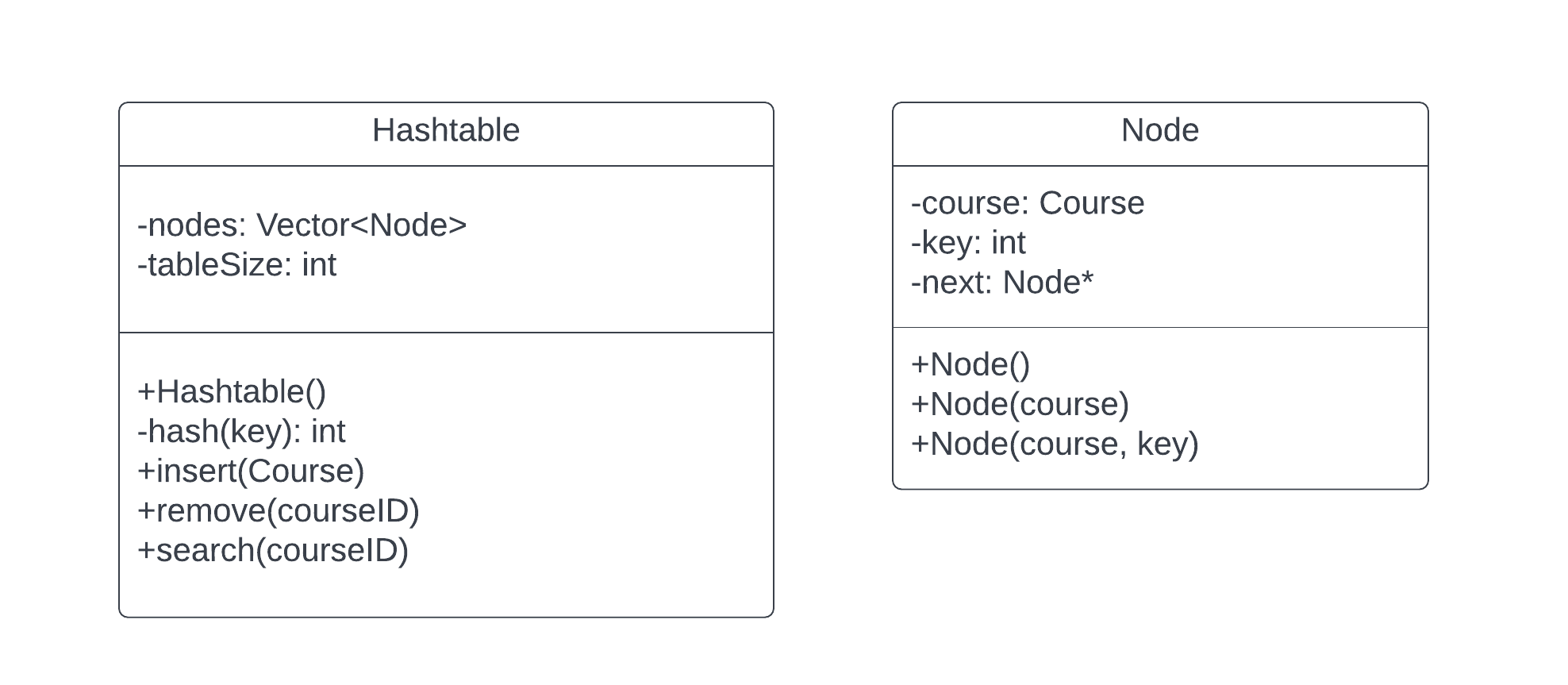
Insert newly created course into courses hashtable

}

return courses

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Opening file, initializing vector | 3 | 1 | 3 |
| Reading each course line | 4 | n | 4n |
| Loading prerequisites | 2 | n | 2n |
| Add course to courses | 1 | n | n |
| Return courses | 1 | 1 | 1 |
| Total Cost | | | 7n + 4 |
| Runtime | | | O(n) |

In addition to the previously mentioned Course class, the hashtable version would also require a Hashtable class, as well as a node class:

Generally a Course object is first created, then inserted into the Hashtable using the insert function. The course ID can be used as the key to the hash.

//Pseudocode for inserting into hashtable:

Hashtable::insert (Course course) {

Node newNode (course, course.getCourseID())

hashKey = hash(course.getCourseID())

if nodes.at(hashkey) is empty

nodes.at(hashkey) = course

else

add node at end of list of nodes in chain in nodes.at(hashkey)

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Create node, hash, if | 3 | 1 | 3 |
| Worst case if all nodes in one bucket | 1 | n | n |
| Total Cost | | | n + 3 |
| Runtime | | | O(n) |

//Pseudocode for searching hashtable

Hashtable::search(coursedID) {

hashkey = hash(coursedID)

if nodes.at(hashkey) isn’t empty

loop through nodes in location

if node.key equal coursed

return node.course

return empty course (not found)

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Hash | 1 | 1 | 1 |
| Worst case if all nodes in one bucket | 1 | n | n |
| Total Cost | | | n + 1 |
| Runtime | | | O(n) |

// Load courses from file pseudocode

// Binary tree version

BinarySearchTree loadCourseData(String fileName) {

courseFileStream = open(filename)

if CourseFileStream not open

Print error

Return

BinarySearchTree courses

while no error reading from file stream {

thisLine = read next line

courseNumber = parse (beginning of string, first comma)

courseName = parse (first comma, second comma or end of string)

while not end of thisLine

prerequisite = parse (current comma, next comma or end of string)

// assumes course are in order where prerequisites are always listed before

// courses that need them

foreach prerequisite in prerequisites {

search courses vector to ensure course exists

if requisite not in courses

Print error with courseNumber

}

Create new Course(courseNumber, courseName, prerequisites)

Insert newly created course into courses binary tree

}

return courses

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Opening file, BinarySearchTree | 3 | 1 | 3 |
| Reading each course line | 4 | n | 4n |
| Loading prerequisites | 2 | n | 2n |
| Add course to tree (see insert into tree below) | 1 | n+3 | n+3 |
| Return courses | 1 | 1 | 1 |
| Total Cost | | | 7n + 7 |
| Runtime | | | O(n) |

In addition to the previously mentioned Course class, the binary tree version would also require a BinarySearchTree class, as well as a node class:

Text

Description automatically generated with low confidence

Generally a Course object is first created, then inserted into the binary tree using the insert function.

//Pseudocode for inserting into hashtable:

BinarySearchTree::insert (Course course) {

Node newNode (course, course.getCourseID())

If root is nullptr {

root = course

return;

}

Node curNode = root;

Loop {

If curNode.key > newNode.key {

If curNode.left is nullptr {

curNode.left = newNode

return

}

else

curNode = curNode.left

}

Else {

If curNode.right is nullptr {

curNode.right = newNode

return

}

Else

curNode = curNode.right

}

}

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Create node, basic code | 3 | 1 | 3 |
| While, worst case if all nodes on one side | 1 | n | N |
| Total Cost | | | n + 3 |
| Runtime | | | O(n) |

//Pseudocode for searching binary tree

BinarySearchTree::search(courseID) {

Node curNode = root

While curNode != nullptr {

If curNode.key equals couseID

Return curNode.course

If curNode.key > courseID

curNode = curNode.left

else

curNode = curNode.right

}

return empty course (not found)

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Basic code | 1 | 1 | 1 |
| While, worst case if all nodes on one side | 1 | n | n |
| Total Cost | | | n + 1 |
| Runtime | | | O(n) |

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

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

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Add number of prereqs for course c | 1 | 1 | 1 |
| Loop – worse case 2 prereqs, which recurses n times (all courses are prerequisites) | 2 | n | 2n |
| Total Cost | | | 2n + 1 |
| Runtime | | | O(n) |

void printSampleSchedule(Vector<Course> courses) {

Sort course vector

for each Course c in courses {

print course

}

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Sort vector | 1 | n log n | n log n |
| for all courses | 1 | n | N |
| print out the course information | 1 | 1 | 1 |
| Total Cost | | | n log n + n + 1 |
| Runtime | | | O(n log n) |

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

for each course in courses {

if course.getCourseNumber equals courseNumber{

print course.getCourseNumber

print course.getCourseName

for each prerequisite in course.getPrerequisites

print prerequisite

}

}

}

| 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 | 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) |

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

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 each node in courses.nodes {

if (node.course != nullptr)

print node.course

while (node.next != nullptr)

node = node.next

print node.course

}

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| For each node | 1 | n | 1 |
| Print first course info | 2 | 1 | 2 |
| Loop through nodes.next (worst case, with all course in one node) | 2 | n | 2n |
| Total Cost | | | 3n +2 |
| Runtime | | | O(n) |

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

Course theCourse equals courses.search(courseNumber)

If course found {

print course.getCourseNumber

print course.getCourseName

for each prerequisite in course.getPrerequisites

print prerequisite

}

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| courses.search (see Hashtable search function pseudocode above) | n+1 | 1 | n+1 |
| 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 | | | 3n + 2 |
| Runtime | | | O(n) |

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in prerequisites

add prerequisites of p to totalPrerequisites

return number of totalPrerequisites

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Add number of prereqs for course c | 1 | 1 | 1 |
| Loop – worse case 2 prereqs, which recurses n times (all courses are prerequisites) | 2 | n | 2n |
| Total Cost | | | 2n + 1 |
| Runtime | | | O(n) |

void printSampleSchedule(Tree<Course> courses) {

//Call recursive function starting from root

PrintCoursesInOrder(courses.root)

}

Void PrintCoursesInOrder(Node\* node) {

If node equals nullptr

Return

PrintCoursesInOrder(node.left)

Print course information

PrintCoursesInOrder(node.right)

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Call to recursive function in main function | 1 | 1 | 1 |
| Call to recursive function within recursive function | 2 | Log n | 2log n |
| Other code in recursive function (if, print) | 2 | Log n | 2log n |
| Total Cost | | | 4log n +1 |
| Runtime | | | O(log n) |

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

Course theCourse equals courses.search(courseNumber)

If course found {

print course.getCourseNumber

print course.getCourseName

for each prerequisite in course.getPrerequisites

print prerequisite

}

}

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| courses.search (see tree search function pseudocode above) | n+1 | 1 | n+1 |
| 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 | | | 3n + 2 |
| Runtime | | | O(n) |

Evaluation:

Each structure could be used to store course data, but each has advantages and disadvantages. Using a vector structure is the simplest, in terms of code complexity. The downside is that searching for a particular course is comparatively slow at O(n). Before being able to print a list of courses in order, it would be necessary to sort the array, also a fairly slow operation at O(n log n). Inserting and deleting courses is also comparatively slow, but adding courses only happens when loading the data file and deleting courses generally would not happen.

Using a hashtable structure would generally be faster to search for a particular course – O(1) in the best case, O(n) as a worst case. Printing the courses in order would be a fairly slow operation, as sorting them would require checking all the buckets and then, if using chaining, looking through all the nodes in each bucket. Inserting data would be faster than using a Vector, between O(1) to O(n) at worst case.

A binary tree is somewhere between the two previous structures in performance: searching for a particular course is faster than the Vector option at O(log n) to O(n) worst case, but not as fast as a hashtable. Sorting/printing a list of courses in order would be faster than both the vector and the hashtable at O(log n) to O(n). Insertion would be faster than the Vector at O(log n) to O(n), but slower than the hashtable.

Determining the best structure depends on usage. Given the fairly small number of courses that the program needs to deal with, the vector option appears to be more than sufficient, and benefits by being easier to code and test. If there were significantly more courses, this option would eventually become slow enough to become a problem, but with the current requirements it is likely the best choice.