# CS 300 Pseudocode Document

## 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

return length of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

sort courses alphanumerically by course number

for each course in courses

print course details (course number, title)

}

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

for each course in courses

if course's number equals courseNumber

print course details (course number, title)

if course has prerequisites

for each prerequisite of the course

print the prerequisite course details (course number, title)

}

## Vector Runtime Analysis

| **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** | 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) |

## Vector Advantages/Disadvantages

* **Advantages:**
  + Simplicity: Vectors (or dynamic arrays) are straightforward to implement and understand.
  + Contiguous Memory Allocation: This allows faster access in many cases due to better cache locality.
  + Direct Access: O(1) time complexity for accessing an element if the index is known.
* **Disadvantages:**
  + Insertion/Deletion: Inserting or deleting elements (especially in the middle) can be costly - O(n) in the worst case.
  + Resizing: If the vector runs out of allocated space, it needs to resize, which might involve copying all elements to a new location.
  + Search: To find an element or check for its existence, O(n) time is required in the worst case if the position is not known.

## Hashtable 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

return length of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

sort courses alphanumerically by course number

for each course in courses

print course details (course number, title)

}

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

for each course in courses

if course's number equals courseNumber

print course details (course number, title)

if course has prerequisites

for each prerequisite of the course

print the prerequisite course details (course number, title)

}

int numPrerequisiteCourses(Hashtable<Course> courses, String courseNumber) {

if courses contains courseNumber

course = courses.get(courseNumber)

return length of course's prerequisites

else

return 0 // or some flag value indicating course not found

}

void printSampleSchedule(Hashtable<Course> courses) {

allCourseNumbers = get all keys from courses

sort allCourseNumbers alphanumerically

for each courseNumber in allCourseNumbers

course = courses.get(courseNumber)

print course details (course number, title)

}

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

if courses contains courseNumber

course = courses.get(courseNumber)

print course details (course number, title)

if course has prerequisites

for each prerequisiteNumber in course's prerequisites

prerequisiteCourse = courses.get(prerequisiteNumber)

print prerequisiteCourse details (course number, title)

else

print "Course not found"

}

## Hashtable Runtime Analysis

| **Hashtable Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **get course using courseNumber** | 1 | 1 | 1 |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | 2 (max) | 2 |
| **get prerequisite course using courseNumber** | 1 | 2 (max) | 2 |
| **print the prerequisite course information** | 1 | 2 (max) | 2 |
| **Total Cost** |  |  | 8 |
| **Runtime** | | | O(1) |

## HashTable Advantages/Disadvantages

* **Advantages:**
  + Fast Access: Provides O(1) average time complexity for search, insert, and delete operations.
  + Scalability: Can handle a large number of entries efficiently.
* **Disadvantages:**
  + Space Overhead: Extra space is needed for keys, hash values, and sometimes for handling collisions.
  + Complexity: More complex to implement than vectors or trees.
  + Performance Variation: Performance depends on the quality of the hash function. Poor hash functions can lead to collisions, reducing efficiency.
  + No Order: The hash table doesn't maintain any order of the elements.

## Tree Pseudocode

int numPrerequisiteCourses(Tree<Course> courses, String courseNumber) {

course = searchBST(courses.root, courseNumber)

if course is not null

return length of course's prerequisites

else

return 0 // or some flag value indicating course not found

}

void printSampleSchedule(Tree<Course> courses) {

inOrderTraversal(courses.root)

}

void inOrderTraversal(Node node) {

if node is not null

inOrderTraversal(node.left)

print node.course details (course number, title)

inOrderTraversal(node.right)

}

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

course = searchBST(courses.root, courseNumber)

if course is not null

print course details (course number, title)

if course has prerequisites

for each prerequisiteNumber in course's prerequisites

prerequisiteCourse = searchBST(courses.root, prerequisiteNumber)

print prerequisiteCourse details (course number, title)

else

print "Course not found"

}

Course searchBST(Node node, String courseNumber) {

if node is null or node.course's number is courseNumber

return node.course

if courseNumber < node.course's number

return searchBST(node.left, courseNumber)

return searchBST(node.right, courseNumber)

}

## Tree Runtime Analysis

| **Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **search course using courseNumber (BST traversal**) | 1 | log n (avg) | Log n (avg) |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | 2 (max) | 2 |
| **search prerequisite using courseNumber** | 1 | 2 log n (avg) | 2 log n (avg |
| **print the prerequisite course information** | 1 | 2 (max) | 2 |
| **Total Cost** | | | 3 log n + 5 |
| **Runtime** | | | O(log n) |

## Tree Advantages/Disadvantages

* **Advantages:**
  + Logarithmic Access: Provides O(log n) time complexity for search, insert, and delete in average scenarios (for balanced trees).
  + Maintains Order: Elements are always in a specific order, facilitating operations like in-order traversal.
  + Flexibility: Allows efficient range queries and can be self-balancing (like AVL trees or Red-Black Trees).
* **Disadvantages:**
  + Overhead: Each entry requires additional memory for two pointers (left and right child).
  + Complexity: More complex than vectors, especially self-balancing trees.
  + Worst-case Time: If not balanced, the tree can degrade to a linked list, making the worst-case time complexity O(n).

## File Handling & Parsing Pseudocode

Function loadCoursesFromFile(filename) -> List<Course>

Declare List<Course> courses = empty list

Declare File file = open filename in read mode

if file doesn't exist

print "File not found."

return courses

while file has more lines

Declare String line = read line from file

// Parsing the line

Declare String[] tokens = split line by ","

// Check for formatting errors (Assuming a format: courseNumber,title,prerequisite1,prerequisite2)

if length of tokens is not between 2 and 4

print "Error in line format:", line

continue to next iteration

Declare Course course = new Course

course.number = tokens[0]

course.title = tokens[1]

if length of tokens > 2

course.addPrerequisite(tokens[2])

if length of tokens > 3

course.addPrerequisite(tokens[3])

courses.add(course)

close file

return courses

End Function

## Menu System Pseudocode

Function mainMenu()

Declare Vector<Course> courses

Declare Hashtable<Course> coursesHashtable

Declare Tree<Course> coursesTree

Declare currentDataStructure = "none" // initially no data structure is loaded

while true

print "Select an option:"

print "1. Load Data into Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "4. Exit"

Declare int choice = read user input

switch choice

case 1:

print "Select the data structure to load into:"

print "a. Vector"

print "b. Hashtable"

print "c. Tree"

Declare char dsChoice = read user input

switch dsChoice

case 'a':

print "Loading data into Vector..."

courses = loadCoursesFromFile("filename.txt")

currentDataStructure = "vector"

case 'b':

print "Loading data into Hashtable..."

coursesHashtable = loadIntoHashtable(loadCoursesFromFile("filename.txt"))

currentDataStructure = "hashtable"

case 'c':

print "Loading data into Tree..."

coursesTree = loadIntoTree(loadCoursesFromFile("filename.txt"))

currentDataStructure = "tree"

default:

print "Invalid data structure choice."

case 2:

if currentDataStructure == "none"

print "No data structure loaded. Load data first."

else

print "Printing course list from", currentDataStructure, "..."

if currentDataStructure == "vector"

printSampleSchedule(courses)

else if currentDataStructure == "hashtable"

printSampleSchedule(coursesHashtable)

else if currentDataStructure == "tree"

printSampleSchedule(coursesTree)

case 3:

if currentDataStructure == "none"

print "No data structure loaded. Load data first."

else

print "Enter course number:"

Declare String courseNumber = read user input

if currentDataStructure == "vector"

printCourseInformation(courses, courseNumber)

else if currentDataStructure == "hashtable"

printCourseInformation(coursesHashtable, courseNumber)

else if currentDataStructure == "tree"

printCourseInformation(coursesTree, courseNumber)

case 4:

print "Exiting..."

exit program

default:

print "Invalid main menu choice. Please select again."

End Function

## Recommendations

Considering the data (courses) is likely not changed frequently but queried often, Hash Tables would generally be recommended for faster individual look-ups (O(1)). However, printing all the courses in alphanumeric order becomes a bit more complicated and less efficient with hash tables.

My recommendation would be a balanced Binary Search Tree (BST). The BST can provide efficient search capabilities while maintaining the order of elements, making it easier to print all courses in order. We also ensure that the tree remains relatively balanced, providing good average-case time complexities (O(log n)).

It can find course details quickly and can list them in order easily. It's like having a well-organized bookshelf where you can quickly find the book you want and also see all the books in order from A to Z!