# CS 300 Pseudocode Document

Danielle Williams

SNHU

Document Outline:  
This document provides pseudocode for three milestones, focusing on vectors, hash tables, binary search trees; the last portion covers runtime analysis.

**Vectors - Milestone 1**

• Displays menu options for loading data and printing courses.

• Functions include:

• `displayMenu()`

• `printCourseList(Vector<Course> courses)`

• `searchCourse(Vector<Course> courses, String courseNumber)`

• `getCourse(Vector<Course> courses, String courseNumber)`

• `OpenReadFile(string filename))`

• `ParseCourseLines (Vector<Course> courseLines)`

**Hash Table - Milestone 2**

• Search and retrieve course information using a hash table.

• Key functions:

• `getCourse(HashTable<Course> table, String courseNumber)`

• `printCourseInformation(Course course)`

**Binary Search Tree - Milestone 3**

• Implements in-order traversal for course listing and sorting.

• Key functions include:

• `printCourseList(Tree<Course> courses)`

• `searchCourse(Tree<Course> courses, String courseNumber)`

• `inOrder(Tree<Course> node)`

**Runtime Analysis**

• Compares efficiency of each data structure:

• Vector: O(n)

• Hash Table: O(1)

• Tree: O(log n)

• Highlights implementation challenges and expected performance.

**//Vector - Milestone 1**

void displayMenu() {

loop until user exits

print menu:

1. Load Data

2. Print Course List

3. Print Course(s)

9. Exit program

read user input and store in variable

switch(variable)

case 1:

get and read filename

courseLines = OpenReadFile(filename)

courses = ParseCourseLines(courseLines)

case 2:

get courseNumber

read courseNumber

searchCourse(courses, courseNumber)

case 3:

printCourseList(courses)

case 9:

exit the loop

default:

print "Invalid input"

}

void printCourseList(Vector<Course> courses) {

sort courses by courseNumber (asc)

for each course in courses

print course.courseNumber and course.name

}

void searchCourse(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

}

struct Course {

String courseNumber

String name

Vector<String> prerequisites

}

void searchCourse(HashTable<Course> courses, String courseNumber) {

course = getCourse(courses, courseNumber)

if course exists

printCourseInformation(course)

for each prerequisite in course.prerequisites

prerequisiteCourse = getCourse(courses, prerequisite)

if prerequisiteCourse exists

printCourseInformation(prerequisiteCourse)

}

void getCourse(HashTable<Course> table, String courseNumber) {

index = hash(courseNumber)

for each course in table[index]

if course.courseNumber == courseNumber

return course

return null

}

Vector<string> OpenReadFile(string filename) {

initialize Vector<string> courseLines

initialize string line

create ifstream instream

open file using instream with filename

if file is not open

output "message”

return empty courseLines

while instream has lines to read

read line into variable 'line'

push line to back of courseLines

close file

return courseLines

}

Vector<Course> ParseCourseLines(Vector<string> courseLines) {

initialize Vector<Course> courses

for each line in courseLines

split line by comma into parts

if parts.length < 2

print "Error message”

continue to next line

courseNumber = parts[0]

courseName = parts[1]

prerequisites = parts[2 to end]

create new Course with courseNumber, courseName, prerequisites

add Course to courses vector

for each course in courses

for each prereq in course.prerequisites

if prereq is not found in courses

print "Error message”

return courses

}

void main() {

courseLines = OpenReadFile("file.txt")

courses = ParseCourseLines(courseLines)

prompt user for courseNumber

read courseNumber input

searchCourse(courses, courseNumber)

}

void printCourseInformation(Course course) {

print course.courseNumber and course.name

if course has prerequisites

print Prerequisites

for each prerequisite in course.prerequisites

print prerequisite

else

print "nothing found message"

}

**//Hash Table - Milestone 2**

void searchCourse(HashTable <Course> courses, String courseNumber) {

for each course in courses

if course.courseNumber == courseNumber

printCourseInformation(course)

for each prerequisite in course.prerequisites

prerequisiteCourse = getCourse(courses, prerequisite)

if prerequisiteCourse exists

printCourseInformation(prerequisiteCourse)

}

Course getCourse(HashTable<Course> table, String courseNumber) {

index = hash(courseNumber)

for each course in table[index]

if course.courseNumber == courseNumber

return course

return null

}

void printCourseInformation(Course course) {

print course.courseNumber and course.name

if course has prerequisites

print Prerequisites

for each prereq in course.prerequisites

print prereq

else

print "nothing found message"

}

**//Binary Search Tree – Milestone 3**

void printCourseList(Tree<Course> courses) {

inOrderTraversal(courses.root)

}

void inOrder (Tree<Course> node) {

if node is not null

inOrder (node.left)

print node.course.courseNumber + node.course.name

inOrder (node.right)

}

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

course = getCourse(courses, courseNumber)

if course exsists

print course information printCourseInformation

start for loop: for each prerequisite in courses prerequisite

prerequisite = get prerequisite for course

if prerequisite exsists

print prerequisite information printCourseInformation

}

Course getCourse(Tree<Course> tree, String courseNumber) {

if tree is empty

return null

if courseNumber == tree.root.courseNumber

return tree.root

else if courseNumber < tree.root.courseNumber

return getCourse(tree.left, courseNumber)

else

return getCourse(tree.right, courseNumber)

}

void printCourseInformation(Course course) {

print course.courseNumber and course.name

if course has prerequisites

print Prerequisites

for each prerequisite in course.prerequisites

print prerequisite

else

print "nothing found message"

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Vector - Vectors are slower when searching through larger datasets due to their linear search method. This means that as the number of courses increases, the time taken to find a specific course also increases proportionally. However, vectors are simpler and easier to implement compared to the other options, making them a good choice for smaller datasets or when ease of use is a priority.** | | | O(n) |
| **Hash – Hash tables offer fast access times, often achieving constant time complexity for search operations. This speed is beneficial when dealing with large amounts of data, as it allows for quick lookups. However, hash tables can encounter potential issues related to logic and structure, particularly if the implementation is not carefully managed. Problems such as collisions can arise if multiple keys hash to the same index, which may complicate retrieval and storage.** | | | 0(1) |
| **Tree – Trees provide an average speed of searching while maintaining a well-organized structure. Their inherent ability to keep data sorted makes searching more efficient, as the tree’s branching structure allows for logarithmic search times. Automatic sorting is a significant advantage, especially in scenarios where data is frequently updated or accessed in a sorted manner.** | | | O(log n) |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Out of vector, hash, and tree, I prefer hash tables for several reasons. Although many might opt for trees due to their speed and structured organization, my experience shows that I can implement hash tables more effectively. I have a solid understanding of mapping, which is crucial for using hash tables. These tables efficiently map each course number to a specific index, allowing for quick data retrieval.

With a well-planned implementation, hash tables can achieve constant time complexity for search operations, making them fast and reliable. In contrast, while trees can offer better organization, they may introduce additional overhead with their balancing mechanisms. Ultimately, my familiarity with hash tables not only boosts my confidence in their implementation but also ensures that users can search and display course data seamlessly.