# CS 300 Pseudocode Document

**Project One**

**Steven Nave**

**V1.4**

**8/5/2025**

## Example Function Signatures

Below is an example of a function signature that you can use as a guide to help address the program requirements using each data structure for the milestones. The pseudocode for finding and printing course information is also given below and depicted in bold to help you get started. The provided pseudocode is for a vector data structure, so you may use this pseudocode in your first milestone as is. The hash table and tree structures are also shown below. But these structures are left for you to do in future milestones.

//Vector - Milestone 1

Struct Course {

String courseNumber;

String courseTitle;

Vector<string> prerequisites;

}

void validateCourses(Vector<Course> courses) {

// check that all prereqs exist in courses

FOR each course in courses {

FOR each prerequisite in course.prerequisites {

SET prereqFound = false

FOR each otherCourse in courses {

IF otherCourse.courseNumber == prerequisite {

SET prereqFound = true

BREAK

}

}

IF not prereqFound{

PRINT “Error: Prerequisite “ + prerequisite + “for course “ + course.courseNumber + “ does not exists”

}

}

}

}

Void loadCourses (string filename, Vector<Course> courses) {

Open file using filename

IF file fails to open {

Print “Error: Cannot open file”

return

}

WHILE lines to read from file {

Read line from file

Split line by comma into parts

If number parts < 2 {

Print “Error: Line missing parameter”

}

Course course;

Course.courseNumber = first part;

Course.courseTitle = second part;

For each remaining part {

Add part to course.prerequisites

}

Add course to courses

}

Close file

}

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**

}

Void printAllCourses (Vector<Course> courses) {

// Sort courses alphanumerically by course number

FOR i = 0 to courses.size() - 1 {

FOR j = i + 1 to courses.size() {

IF courses[i].courseNumber > courses[j].courseNumber{ SWAP courses[i] and courses[j]

}

}

}

FOR each course in courses {

PRINT course.courseNumber + “: “ courses.courseTitle

}

}

//Hash Table - Milestone 2

void validateCourses(Vector<Course> courses) {

// check that all prereqs exist in courses

FOR each course in courses {

FOR each prerequisite in course.prerequisites {

SET prereqFound = false

FOR each otherCourse in courses {

IF otherCourse.courseNumber == prerequisite {

SET prereqFound = true

BREAK

}

}

IF not prereqFound{

PRINT “Error: Prerequisite “ + prerequisite + “for course “ + course.courseNumber + “ does not exists”

}

}

}

}

void loadCourses(HashTable<Course> hashtable, String filename) {

ASSIGN courses as Vector<Course>

OPEN file using filename

IF file fails to open {

PRINT “Error: Cannot open file”

return

}

// Read and parse file

WHILE lines to read from file {

READ line from file

SPLIT line by comma into parts

IF number parts < 2 {

PRINT “Error: line missing parameter”

CONTINUE to next line

}

Course course;

Course.courseNumber = first part;

Course.courseTitle = second part;

FOR each remaining part {

ADD part to course.prerequisites

}

Add course to courses vector

}

CLOSE file

//validate all course have valid prereqs

validateCourses(courses)

//Load validated courses into hash table

FOR each course in courses {

// Calculate hash key from course number

SET key = hash (course.courseNumber)

// Get node at calculated position

SET node = hashTable.nodes[key]

// if position is empty

IF node.key is empty {

SET node.key = key

SET node.course = course

SET node.next = null

}

// If position is occupied, add to chain

ELSE {

CREATE new node with course and key

SET current = node

WHILE current.next != null {

SET current = current.next

}

SET current.next = new node

}

}

}

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

// Search through hashtable to find course

FOR each bucket in hashTable {

FOR each course in that bucket {

IF course.courseNumber == courseNumber {

// Print course information

PRINT course.courseNumber + “, “ + course.courseTitle

// Print prereqs

IF course.prerequisites is empty {

PRINT “Prerequisites: None”

} ELSE {

PRINT “Prerequisites: “

FOR each prerequisite in course.prerequisites {

PRINT prerequisite

IF not last prerequisite{

PRINT “, “

}

}

}

RETURN

}

}

}

//Course was not found

PRINT “Course “ + courseNumber + “ not found”

}

Void printAllCourses(HashTable hashtable) {

Vector<Course> allCourses;

//Collect all courses from hash table

FOR each bucket in hashtable.buckets {

SET current = bucket

WHILE current != NULL {

ADD current.course to allCourses

SET current = current.next

}

}

// Sort courses alphanumerically

FOR i = 0 to allCourses.size()-1 {

FOR j = i + 1 to allCourses.size() {

IF allCourses[i].courseNumber > allCourses[j].courseNumber {

SWAP allCourses[i] and allCourses[j]

}

}

}

//print sorted courses

FOR each course in allCourses {

PRINT course.courseNumber + “: “ + course.courseTitle

}

}

//Binary Search Tree – Milestone 3

Struct TreeNode {

Course course

TreeNode left

TreeNode right

}

Void insert (Course course) {

IF root IS null:

Root = new TreeNode(course)

ELSE:

InsertNode(root, course)

}

Void insertNode(TreeNode node, Course course) {

IF course.courseNumber < node.course.courseNumber

IF node.left is null:

node.left = new treeNode(course)

ELSE:

insertNode(node.left, course)

ELSE:

IF node.right is null

node.right = new TreeNode(course)

ELSE

insertNode(node.right, course)

}

Void loadCourses(Tree<Course> courses, String fileName) {

ASSIGN courses as Vector<Course>

OPEN file using filename

IF file fails to open {

PRINT “Error: Cannot open file”

return

}

// Read and parse file

WHILE lines to read from file {

READ line from file

SPLIT line by comma into parts

IF number parts < 2 {

PRINT “Error: line missing parameter”

CONTINUE to next line

}

Course course;

Course.courseNumber = first part;

Course.courseTitle = second part;

FOR each remaining part {

ADD part to course.prerequisites

}

Add course to courses vector

}

CLOSE file

//validate all course have valid prereqs

validateCourses(courses)

//Insert all validated courses into tree

FOR each course in courseList {

Courses.insert(course)

}

}

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

// start at the root of tree

TreeNode<Course> current = courses.root

// search for the course in BST

WHILE current IS NOT null:

IF current.course.courseNumber == courseNumber:

// Print main course info

Print current.course.courseNumber and current.course.courseName

//check if course has prereqs

IF current.course.prerequisites is empty:

Print “Prerequisites: None”

ELSE:

PRINT “Prerequisites:”

// loop through each prereq course number

FOR each prereqNumber in current.course.prerequisites:

//Search tree for this prereq

TreeNode<Course> prereqNode = courses.root

WHILE prereqNode IS NOT null:

IF prereqNode.course.courseNumber == prereqNumber:

PRINT “prereqNode.course.courseNumber : prereqNode.course.courseName

Break

ELSE IF prereqNumber < prereqNode.course.courseNumber:

PrereqNode = prereqNode.left

ELSE:

PrereqNode = prereqNode.right

//If prereq was not found in tree

IF prereqNode IS null:

Print prereqNumber Not found

RETURN

ELSE IF courseNumber < current.course.courseNumber:

Current = current.left

ELSE

Current = current.right

Print Course with courseNumber not found

}

Void printAllCourses(Tree<Course> courses) {

PrintInOrder(courses.root)

}

Void printInOrder(TreeNode<Course> node) {

IF node IS NOT null:

printInOrder(node.left)

PRINT node.course.courseNumber : node.course.courseTitle

printInOrder(node.right)

}

Void displayMenu() {

PRINT “ABCU Course Management System”

PRINT “1. Load Data Structure”

PRINT “2. Print Course List”

PRINT “3. Print Course”

Print “9. Exit”

PRINT “Make your selection...”

}

Int main() {

Int choice = 0;

String courseNumber;

String filename = “courses.txt”

Bool dataLoaded = false; // track if data has been loaded

WHILE choice != 9 {

DisplayMenu()

INPUT choice

SWITCH choice {

CASE 1:

LoadCourses(courses, filename)

SET dataLoaded = true;

PRINT “Courses Loaded successfully”

BREAK

CASE 2:

IF not dataLoaded {

PRINT “Please load courses first”

} ELSE {

PrintAllCourses(courses)

}

BREAK

CASE 3:

IF not dataLoaded {

PRINT “Please load courses first”

} ELSE {

PRINT “Enter course number: “

INPUT courseNumber

SearchCourse(courses, courseNumber)

}

BREAK

CASE 9:

PRINT “Goodbye!”

BREAK

DEFAULT:

PRINT “Invalid option. Please try again.”

BREAK

}

}

RETURN 0;

}

## Example Runtime Analysis

When you are ready to analyze the runtime for the Project One data structures for which you created the pseudocode, use the example chart below to support your work. This particular 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. The example only covers the search function for the vector structure. You do not have to complete your runtime analysis until Project One. However, working on your analysis now may help you understand the changes as you complete the milestones. Don’t forget to include your charts in Project One. You will submit Project One in Module Six.

| **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 |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

**Evaluation**

Vector:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Line Cost** | **Times Executed** | **Total Cost** |
| File operations | 1 | 1 | 1 |
| Read Each line | 1 | n | n |
| Parse line | 1 | n | n |
| Create Course object | 1 | n | n |
| Add to vector | 1 | n | n |
| Validate courses | N^2 | 1 | N^2 |

Total Cost: O(n^2)

Pros:

* Simple to implement
* Predictable memory usage

Cons:

* Linear search time O(n)
* Sorting required for this implementation is O(n^2)

HashTable:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Line Cost | Times Executed | Total Cost |
| File operations | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Parse line | 1 | n | n |
| Create Course object | 1 | n | n |
| Hash and insert | 1 | n | N |
| Validate courses | N^2 | 1 | N^2 |

Total Cost: O(n^2)

Pros:

* Fast average search
* Good for frequent lookups

Cons:

* Requires sorting for ordered output
* Higher memory usage overhead

BinarySearchTree:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Line Cost | Times Executed | Total Cost |
| File Operations | 1 | 1 | 1 |
| Read each line | 1 | n | n |
| Parse line | 1 | n | n |
| Create Course object | 1 | n | n |
| Insert into BST | Log n | n | N log n |
| Validate courses | N^2 | 1 | N^2 |

Total cost: O(n^2)

Pros:

* Efficient search time
* Naturally sorted in order
* Balanced time and space complexity

Cons:

* More complex to implement
* Recursive implementation uses stack memory

Data Structure Recommendation:

The binary search tree would be the best data structure to use for the ABCU CS department course program, as it requires no extra code for the sorting of courses when loading them, and will always have an efficient search time due to the binary search capabilities, making it O(log n) searchable.