# CS 300 Pseudocode Document

## 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

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**

}

//Hash Table - Milestone 2

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

}

//Binary Search Tree – Milestone 3

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

## 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) |

Menu Pseudocode:

FUNCTION DisplayMenu()

OUTPUT “ABCU Course Advising Menu”

OUTPUT option 1 “Load course data”

OUTPUT option 2 “Print Computer Science Course List”

OUTPUT option 3 “Print Specific Course Information”

OUTPUT option 9 “Exit program”

END FUNCTION

FUNCTION Main()

INT userInput = next input

BOOL loadedData = false

VECTOR courseVector

HASHTABLE courseHashTable

TREE courseTree

REPEAT

DisplayMenu()

OUTPUT “Choose an option”

INPUT userInput

SWITCH userInput

CASE 1:

LoadCourseData (courseVector, courseHashTable, courseTree)

SET loadedData = true

OUTPUT “Course data loaded successfully

BREAK

CASE 2:

IF loadedData == False Then

OUTPUT “Error, need course data”

ELSE

OUTPUT Computer Science course data

ENDIF

BREAK

CASE 3:

IF loadedData == False Then

OUTPUT “Error, need course data”

ELSE

OUTPUT “Enter a course number”

INPUT courseNumber

OUTPUT corresponding course data

ENDIF

BREAK

CASE 9:

OUTPUT exiting program

BREAK

DEFAULT:

OUTPUT “Option invalid, choose a valid option”

END SWITCH

UNTIL userChoice == 9

END FUNCTION

Computer Science program output in alphanumeric order pseudocode:

FUNCTION PrintSortedCompSciData(courseVector, courseHashTable, courseTree)

OUTPUT “Computer Science Courses (Sorted)”

DECLARE tempVector as Vector

FOR each course in courseVector

IF courseNumber starts with “CSC”

ADD course to tempVector

ENDIF

ENDFOR

SORT tempVector by courseNumber (alphanumeric ascending order)

FOR each course in tempVector

OUTPUT course’s courseNumber, courseTitle

END FOR

DECLARE tempList as List

FOR each key in courseHashTable

IF key starts with “CSC”

ADD courseHashTable[key] to tempList

ENDIF

END FOR

SORT tempList by courseNumber

FOR each course in tempList

OUTPUT course’s courseNumber, courseTitle

ENDFOR

Call InOrderTraversal(courseTree.root)

FUNCTION InOrderTraversal(node)

IF node is NULL

RETURN

ENDIF

CALL InOrderTraversal(node.left)

IF node courseNumber starts with “CSC”

OUTPUT node’s courseNumber, courseTitle

ENDIF

CALL InOrderTraversal(node.right)

END FUNCTON

END FUNCTION

FUNCTION PrintCourseInfo(courseNumber, courseHashTable)

IF courseNumber exists in courseHashTable

Course = courseHashTable[courseNumber]

OUTPUT “Course “ + course’s courseNumber + “ – “ + course’s courseTitle

IF course’s prerequisites is empty

OUTPUT “no prerequisites”

ELSE

OUTPUT “Prerequisites “ + Join(course’s prerequisites, “, “)

ENDIF

ELSE

OUTPUT “Course not found”

ENDIF

END FUNCTION

| **Code for Load Course Data from File Milestone 1 (Vector)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **OPEN FILE File Name** | 1 | 1 | 1 |
| **IF file doesn’t open** | 1 | 1 | 1 |
| **VECTOR Course courses** | 1 | 1 | 1 |
| **FOR each line of file** | 1 | ~n | 1 |
| **REMOVE trailing and leading spaces** | 1 | n | n |
| **IF line is empty - continue** | 1 | n | n |
| **SPLIT line by commas** | 1 | Up to n | n |
| **IF length of list is < 2 – print error** | 1 | Up to n | n |
| **DEFINE Course newCourse** | 1 | Up to n | n |
| **SET newCourse.courseNumber = list[0]** | 1 | Up to n | n |
| **SET newCourse.title = list[1]** | 1 | Up to n | n |
| **IF length > 2 – FOR each prerequisite** | 1 | Up to n | n |
| **FOR each prerequisite ADD to newCourse.prerequisite** | 1 | =< p \* n | O(pn) |
| **ADD newCourse to courses** | 1 | Up to n | n |
| **CLOSE FILE** | 1 | 1 | 1 |
| **RETURN courses** | 1 | 1 | 1 |
| **Total Cost** | | | 13n+pn+5 |
| **Runtime** | | | O(n\*p) |

Advantages of vectors:

* Easy to use
* Simple to loop through for validation and to display
* Maintains insertion order

Disadvantages of vectors:

* Runtime for searching is not efficient for large amounts of data
* To validate the prerequisites, you’d have to search through the entire list
* Not optimal to use for fast lookups or data insertion

| **Code for Load Course Data from File Milestone 2 (Hash Table)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Create empty dictionary courseTable** | 1 | 1 | 1 |
| **Loop over each line in file** | 1 | n | n |
| **Split line into elements by comma** | 1 | n | n |
| **If length < 2 - error** | 1 | n | n |
| **Assign CourseNumber = first element** | 1 | n | n |
| **Assign Prerequisites = elements[…]** | 1 | n | n |
| **Call constructor function** | 1 | n | n |
| **Constructor function** | 1 | 3 per call | 3n |
| **Insert course object into dict** | 1 | n | n |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | 13n+3 |
| **Runtime** | | | O(n) |

Advantages of Hash tables:

* Fast lookup
* Efficient validation using key checking
* Simple to implement
* Supports quick checking for the existence of something

Disadvantages:

* Structure is not ordered, have to use sort keys for ordered printing
* Cannot naturally traverse in course order

| **Code for Load Course Data from File Milestone 3 (Binary Search Tree)** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define maps** | 1 | 3 | 3 |
| **Open input file** | 1 | 1 | 1 |
| **If file fails print + exit** | 1 | 1 | 1 |
| **While not end of file – read line by line** | 1 | n | n |
| **If line is not empty, append** | 1 | =< n | =< n |
| **Close file** | 1 | 1 | 1 |
| **For each line in rawLines** | 1 | n | n |
| **Split into elements** | 1 | n | n |
| **If < 2 elements print error + exit** | 1 | n | n |
| **Assign courseNumber, courseTitle** | 1 each | n | 2n |
| **Add courseNumber to courseList** | 1 | n | n |
| **If elements > 2 extract data and add to map or empty list** | 1 | n | n |
| **Total Cost** | | | 10n+5 |
| **Runtime** | | | O(n) |

Advantages of Binary Search Trees

* Prints courses in order automatically using in-order traversal
* Sorted structure
* Better than using a list because of in-order logic and range searches

Disadvantages:

* Long runtime for insertions if tree is unbalanced
* Validation uses list-based search
* More difficult to implement than the other two structures
* Can be slower if tree is unbalanced

My personal recommendation for the data structure I plan to use in my code is a Hash Table. The hash table provides the best Big O performance and is easy to implement. The hash table also provides the fastest lookup and by using a key value, courses are organized naturally via their course number. This also allows for quick retrieval of course info, quick updates, easy deletions, and simple validation, all because of the use of a key value. The code needed is also not very complex for what we’re using the hash table for and the hash table is much more scalable when handling large datasets, like that of the course information we’re using here. Simply put, the hash table will provide results instantly when being utilized because of the associated key value. If the key value is known, you can instantly access whatever data is tied to that key value and if that key value doesn’t exist, you’ll know instantly because nothing will be returned (or some sort of error code could potentially be utilized IE: “item does not exist”)