# CS 300 Pseudocode Document

****

The course data is as follows:

CSCI100,Introduction to Computer Science

CSCI101,Introduction to Programming in C++,CSCI100

CSCI200,Data Structures,CSCI101

MATH201,Discrete Mathematics

CSCI300,Introduction to Algorithms,CSCI200,MATH201

CSCI301,Advanced Programming in C++,CSCI101

CSCI350,Operating Systems,CSCI300

CSCI400,Large Software Development,CSCI301,CSCI350

**1.) Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors**

//readFile- Milestone 1

Vector<Course> readFile(String fileName) {

**Create courseVec  
Create readVal   
try{  
 open file fileName  
 while not end of file  
 set readVal to file readline output**

**If Search readVal for a ‘,’ to validate, list > 0  
 courseVec appends return val from courseFactory(readVal)**

**Else**

**Print \*warning message\* on readVal (choose log level)  
 continue  
}catch{  
 print file could not open**

**}finally{  
 close file  
}**

**return courseVec**

}

**2.) Design pseudocode to show how to create course objects and store them in the appropriate data structure.**

//CourseFactory- Milestone 1

Course CourseFactory(String readVal) {

**Create new Course:**

**ID = ""**

**Name = ""**

**Prerequisites = empty vector**

**Split readVal by ',' into tokens**

**If tokens.length < 2:**

**Throw error or return null**

**Set Course.ID = tokens[0]**

**Set Course.Name = tokens[1]**

**For i = 2 to tokens.length - 1:**

**Append tokens[i] to Course.Prerequisites**

**return Course**

}

**3.) Design pseudocode that will search the data structure for a specific course and print out course information and prerequisites.**

//searchCourse - Milestone 1

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

**For each course in courses:**

**If course.ID == targetID:**

**Print course details // Exit function immediately after finding the match  
Return**

**Print "Course not found."**

}

**4.) Create pseudocode for a menu. The menu will need to perform the following actions:**

void main() {

**Declare filename as string**

**Declare myDataStructure as Vector<Course>**

**Declare sortedCoursesArray as Array<Course>**

**While true:**

**Print Menu:**

**1 - Load Data from File**

**2 - Sort and Display Courses**

**3 - Search for Course by ID**

**9 - Exit**

**Set choice equal to user input**

**SWITCH choice:**

**Case 1:**

**Set myDataStructure equal to readFile(filename)**

**Case 2:**

**Set sortedCoursesArray equal to mergeSort(myDataStructure, 0 , myDataStructures length -1)**

**Call printDataStructure(sortedCoursesArray)**

**Case 3:**

**Prompt for TargetID**

**Call searchCourse(myDataStructure, TargetID)**

**Case 9:**

**Exit**

**Default:**

**Print "Invalid option"**

}

**5) Sort the course information by alphanumeric course number from lowest to highest.**

// ascending ASCII order with merge

void mergeSort(Vector<Course>& courses, int left, int right) {

// Base case: 0 or 1 element in the segment -> already sorted

**IF (left >= right) THEN**

**return**

**END IF**

// Calculate midpoint of the current segment

**int mid = (left + right) / 2**

// Recursively sort the left half [left .. mid]

**mergeSort(courses, left, mid)**

// Recursively sort the right half [mid+1 .. right]

**mergeSort(courses, mid + 1, right)**

// Merge the two sorted halves [left .. mid] and [mid+1 .. right] into one sorted segment

**merge(courses, left, mid, right)**

}

**Continued below…**

**5) Continued**  
// Merges two sorted subarrays: [left..mid] and [mid+1..right] into one sorted section of courses

void merge(Vector<Course>& courses, int left, int mid, int right) {

// Temporary vector to store merged results (to avoid overwriting original data during merge)

**Vector<Course> temp**

**int i = left** // pointer for left subarray

**int j = mid + 1** // pointer for right subarray

// Traverse both subarrays and merge them in sorted order

**WHILE (i <= mid AND j <= right) DO**

// Compare course IDs lexicographically (ASCII comparison)

**IF (courses[i].courseNumber <= courses[j].courseNumber) THEN**

**ADD courses[i] to temp** // left element is smaller, add it

**i = i + 1** // move to next element in left subarray

**ELSE**

**ADD courses[j] to temp** // right element is smaller, add it

**j = j + 1** // move to next element in right subarray

**END IF**

**END WHILE**

// If there are remaining elements in the left subarray, add them to temp

**WHILE (i <= mid) DO**

**ADD courses[i] to temp**

**i = i + 1**

**END WHILE**

// If there are remaining elements in the right subarray, add them to temp

**WHILE (j <= right) DO**

**ADD courses[j] to temp**

**j = j + 1**

**END WHILE**

// Copy the merged sorted elements from temp back into the original courses vector

**FOR (int k = 0; k < temp.size(); k++) DO**

**courses[left + k] = temp[k]**

**END FOR**

}

**6) Print the sorted list to a display.**

Void printDataStructure( myDataStructure){  
 **for all courses**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

}

## Runtime Analysis for LinkedList

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **readFile:** *per line in file* | 3 | n | n |
| **courseFactory**: *prerequisites* | 1 | k | k |
| **Total Cost** | | | n(3+k) |
| **Runtime** | | | O(n) |

Reading and validating lines from a file generally involve operations like fetching the line and ensuring it's formatted correctly. Although in reality, fetching data from a file system might be more expensive due to I/O overhead, we simplify here for clarity and assume each operation as a constant cost. Parsing is straightforward—each course entry is split into tokens. The vector append operation is constant time amortized, making it efficient for linear operations. The overall runtime is linear, O(n), making this structure efficient for basic insertions but less ideal when searching or sorting frequently.