# Anthony Spedaliere

CS-300-T6612 Analysis and Design

# CS 300 Pseudocode Document

## Function Signatures

// Vector pseudocode

struct Course {

**string courseNumber;**

**string courseName;**

**vector<string> prerequisite;**

};

void loadCourseData(Vector<Course> courses, string fileName) {

**instantiate file stream**

**open fileName**

**error handle if file does not exist**

**read data**

**while not end of document**

**for line in fileName**

**if line contains >= two elements delimited by comma**

**while end of line not reached**

**instantiate new Course object**

**add 1st element to courseNumber(comma reached)** **add 2nd element to courseName (comma reached)** **add rest of elements to prerequisite vector**

**append Course object to vector**

**close the open file**

**compare all Course prerequisites in courses to courseNumber**

**if all match**

**cout “Courses successfully saved.”**

**return**

**else cout “file format is not correct – one or more course prerequisites do not exist.”**

}

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

**totalPrerequisites = prerequisites of course c**

**for each prerequisite p in totalPrerequisites**

**add prerequisites of p to totalPrerequisites**

**print number of totalPrerequisites**

}

void printSampleSchedule(Vector<Course> courses) {

**sortAlphanumerically(courses)**

}

void printCourseInformation(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**

}

Vector<Courses> sortAlphanumerically(Vector<Course> courses) {

**sort courses->courseNumber alphanumerically**

**for all courses**

**print course information**

**end line**

}

// Hashtable pseudocode

struct Course {

**string courseNumber;**

**string courseName;**

**vector<string> prerequisite;**

};

void loadCourseData(Hashtable<Course> courses, string fileName) {

**instantiate file stream**

**open fileName**

**error handle if file does not exist**

**read data**

**while not end of document**

**for line in fileName**

**if line contains >= two elements delimited by comma**

**while end of line not reached**

**instantiate new Course object**

**add 1st element to courseNumber(comma reached)** **add 2nd element to courseName (comma reached)** **add rest of elements to prerequisite vector**

**call Insert(Course)to insert object to hashtable**

**close the open file**

**compare all Course prerequisites in courses to courseNumber**

**if all match**

**cout “Courses successfully saved.”**

**return**

**else cout “file format is not correct – one or more course prerequisites do not exist.”**

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

**int sum = 0;**

**for each key in courses**

**sum += prerequisites.size();**

**return sum;**

}

void printSampleSchedule(Hashtable<Course> courses) {

**for each key in courses**

**print the courseNumber, courseNumber, and prerequisite**

**if collision exists**

**for each course in list print description**

}

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

**key = courseNumber**

**for each course in courses**

**if key matches**

**print out course information**

}

// Tree pseudocode

struct Course {

**string courseNumber;**

**string courseName;**

**vector<string> prerequisite;**

};

void loadCourseData(Tree<Course> courses, string fileName) {

**instantiate file stream**

**open fileName**

**error handle if file does not exist**

**read data**

**while not end of document**

**for line in fileName**

**if line contains >= two elements delimited by comma**

**while end of line not reached**

**instantiate new Course object**

**add 1st element to courseNumber(comma reached)** **add 2nd element to courseName (comma reached)** **add rest of elements to prerequisite vector**

**call Insert(Course)to insert object to tree**

**close the open file**

**compare all Course prerequisites in courses to courseNumber**

**if all match**

**cout “Courses successfully saved.”**

**return**

**else cout “file format is not correct – one or more course prerequisites do not exist.”**

}

int numPrerequisiteCourses(Tree<Course> courses) {

**sum = 0**

**Course\* current = root**

**if current not nullptr**

**traverse through tree recursively**

**sum += Course.prerequisite.length**

**return sum**

}

void printSampleSchedule(Tree<Course> courses) {

**Course\* current = root**

**if Course->next not nullptr**

**traverse through tree recursively**

**print course number and name of current**

}

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

**Course\* current = root**

**while current != nullptr**

**traverse through tree**

**if current == courseNumber**

**print course number and name of current**

**return**

}

// Menu pseudocode

{

**int choice = 0;**

**while choice != 9**

**print menu options:**

**1. Load Data Structure**

**2. Print Course List**

**3. Print Course**

**9. Exit**

**get user input --> choice**

**switch statement (choice)**

**case 1:**

**Load data structure**

**break**

**case 2:**

**Print course list**

**break**

**case 3:**

**Print course**

**break**

**default:**

**print “Invalid option. Try again.”**

}

## Runtime Analysis

// Vector – Reading File and Creating Course Object – void loadCourseData(Vector<Course> courses, string fileName)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **instantiate file stream** | 1 | 1 | 1 |
| **open fileName** | 1 | 1 | 1 |
| **error handle if file does not exist** | 1 | 1 | 1 |
| **read data** | 1 | n | n |
| **while not end of document** | 1 | n | n |
| **for line in fileName** | 1 | n | n |
| **if line contains >= two elements delimited by comma** | 1 | n | n |
| **while end of line not reached** | 1 | n | n |
| **instantiate new Course object** | 1 | n | n |
| **add 1st element to courseNumber(comma reached)** | 1 | n | n |
| **add 2nd element to courseName (comma reached)** | 1 | n | n |
| **add rest of elements to prerequisite vector** | 1 | n | n |
| **append Course object to vector** | 1 | n | n |
| **close the open file** | 1 | 1 | 1 |
| **compare all Course prerequisites in courses to courseNumber** | 1 | n | n |
| **cout “Courses successfully saved.”**  **return** | 1 | 1 | 1 |
| **else cout “file format is not correct – one or more course prerequisites do not exist.”** | 1 | 1 | 1 |
| **Total Cost** | | | 6+11n |
| **Runtime** | | | n |

// Tree – Reading File and Creating Course Object – void loadCourseData(Hashtable<Course> courses, string fileName)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **instantiate file stream** | 1 | 1 | 1 |
| **open fileName** | 1 | 1 | 1 |
| **error handle if file does not exist** | 1 | 1 | 1 |
| **read data** | 1 | n | n |
| **while not end of document** | 1 | n | n |
| **for line in fileName** | 1 | n | n |
| **if line contains >= two elements delimited by comma** | 1 | n | n |
| **while end of line not reached** | 1 | n | n |
| **instantiate new Course object** | 1 | n | n |
| **add 1st element to courseNumber(comma reached)** | 1 | n | n |
| **add 2nd element to courseName (comma reached)** | 1 | n | n |
| **add rest of elements to prerequisite vector** | 1 | n | n |
| **call Insert(Course)to insert object to hashtable** | 1 | log n | log n |
| **close the open file** | 1 | 1 | 1 |
| **compare all Course prerequisites in courses to courseNumber** | 1 | n | n |
| **cout “Courses successfully saved.”**  **return** | 1 | 1 | 1 |
| **else cout “file format is not correct – one or more course prerequisites do not exist.”** | 1 | 1 | 1 |
| **Total Cost** | | | 6+10n + log n |
| **Runtime** | | | n log n |

// HashTable – Reading File and Creating Course Object – void loadCourseData(Tree<Course> courses, string fileName)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **instantiate file stream** | 1 | 1 | 1 |
| **open fileName** | 1 | 1 | 1 |
| **error handle if file does not exist** | 1 | 1 | 1 |
| **read data** | 1 | n | n |
| **while not end of document** | 1 | n | n |
| **for line in fileName** | 1 | n | n |
| **if line contains >= two elements delimited by comma** | 1 | n | n |
| **while end of line not reached** | 1 | n | n |
| **instantiate new Course object** | 1 | n | n |
| **add 1st element to courseNumber(comma reached)** | 1 | n | n |
| **add 2nd element to courseName (comma reached)** | 1 | n | n |
| **add rest of elements to prerequisite vector** | 1 | n | n |
| **call Insert(Course)to insert object to tree** | 1 | log n | log n |
| **close the open file** | 1 | 1 | 1 |
| **compare all Course prerequisites in courses to courseNumber** | 1 | n | n |
| **cout “Courses successfully saved.”**  **return** | 1 | 1 | 1 |
| **else cout “file format is not correct – one or more course prerequisites do not exist.”** | 1 | 1 | 1 |
| **Total Cost** | | | 6+10n + log n |
| **Runtime** | | | n log n |

When choosing an algorithm, various qualities of the algorithm should be considered including but not limited to runtime complexity, spacetime complexity, code readability, and network connection. Runtime complexity is a function of constant time operations for an algorithm given an input size n. This program is to be used with a small input size. Theoretically, if we were to increase that input size to encompass a class schedule for the largest school in the country the size of n would still be relatively small, considering a school can only have a limited number of classes. Therefore, it is safe to assume that n will never be more than 1x10^5. The small value of n will factor into the decision of which algorithm to use.

Spacetime complexity is a function of memory units used by the algorithm for a given input size n. Memory usage increases as the data size increases and the data size increases according to the number of classes in the data set. As previously noted, n is limited due to a school only being able to offer a limited number of courses. Even if this actual number of classes were artificially increased it would still be small. Of the courses offered by a school a limited number of them will be CS courses. Therefore, the number of memory units used should be within reasonable limits even in the most extreme of cases.

Assuming network capabilities remain constant, that leaves code readability. Code readability is desirable but not essential. It is also relative to the programmer or institution. However, there is a general average consensus on what is “readable” and what is considered less readable. For example, the use of recursion adds a layer of complexity to code that makes it slightly more complex. On average, this makes the code more abstract, and abstraction could cause code to be less intuitive.

Considering, runtime complexity and spacetime complexities are a function of n and n is relatively small it would be easy to consider these both constants that are not impactful. We already noted that we would assume network capabilities are constant. Therefore, I would recommend the use of a vector.