# CS 300 Pseudocode Document

## Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

// Vector pseudocode

This was wrong and need to return an int

idk if was suppose to fix it but I did

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

set<course> totaPreReq = c.getPrereq() **O(1)**

for each preReq p in totalPreReq

totalPreReq.addAAll(p.getprereq()); **O(K\*n) K=avg preReq**

end for each

return totalPreReq.size()

}

**TOTAL: O(K\*n +1)**

**RUN 0(n)**

void printSampleSchedule(Vector<Course> courses) {

for each Course in courses **O(n)**

if course is needed and open **O(1)**

print course Name endline

print cousee Time endline

end for each

}

**Total O(n) //iterates over all courses**

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

**for all courses O(n)**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course O(K)**

**print the prerequisite course information**

}

**TOTAL: O(n\*K) K= total number of PreReq**

**RUN: O(n)**

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

int sum = 0 **O(1)**

for each Couse Course in couses.vale(): **O(n)**

if course.isPrereq():

sum = sum +1

end if

return sum

}

**TOTAL: O(n +1)// it iterates over the whole hastable**

**RUN: O(n)**

void printSampleSchedule(Hashtable<Course> courses) {

for each Course in courses **O(n)**

if couse.isNeeded() and couse.isOpen()

node.printShedual()

end if

end for each

}

**TOTAL: 0(n)**

**RUN: O(n)**

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

Couse infoCourse – courses.get(courseNumber) **O(1) if key**

if infoCourse is not null **O(1)**

infoCourse.printInfo()

else

print “not found”

end if

}

**TOTAL: O(2) //assuming the string is a key for the hastable**

**RUN: 0(1)**

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {

root = course.getroot()

if root is null **O(1)**

return zero

int reReq equals zero **O(1)**

for each childNode in root.getChilder()

preeReq += numPrerequisiteCourses(childNode) **O(logk n)**

end for each

return preReq

}

**TOTAL: O(logk n+ 2) if tree is balanced**

**RUN: O(log n)**

**O(n) if unbalanced**

void printSampleSchedule(Tree<Course> courses) {

Node root = courses.getRoot() **O(1)**

if root is null:

print(“non”) **O(1)**

return

printSamplScheduleHelp(root)**O(logk N)**

}

**TOTAL: O (logk n + 1)**

**RUN: O(log n)**

void printSampleShcedleHelp( node)

if node is null **O(1)**

return

node.printcousename endline

node.printTimeDate endline

for each childNode in node.getChildern()**O(K) = k is number of childern**

printSampleSheculeHelp(childNode) **O (logk N)**

end for each

}

**TOTAL: O(log nk  + 1) if tree is balanced**

**RUN: O(log n)**

**O(n) if unbalanced**

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

Node root equals course.getroot() **O (1)**

if root is null

print “none” **O(1)**

return

Node targetNode = findCouseNodeBY Number(root, cousenumber) **O(logK N)**

if targetnode is not null

print course nameTimeDate

for each prereqistateNode in targetNode.getChildern **O(K) k=number of childern**

print targetnode.getcouseNumber

else

print “course not found”

}

**TOTAL O(logk n +1)**

**RUN: O(log n) if tree is balanced**

**O(n) if unbalanced**

TreeNode<Course> findCourseNodeByNumber(TreeNode<Course> node, String number)

if node is null **O(1)**

return null

if node.getNumber equals couseNumber **O(1)**

return node

for each childNode in node.getchildern() **O(K) k= number of childern**

TreeNode<couse> reqsualt – findCourseNodeByNumber(chilNode, couse) **O(logK n)**

if resualt is not null: **O(1)**

return result

end for each

return null

}

**TOTAL: O(logk n + 3) if tree is balanced**

**RUN: O(log n)**

**O(n) if unbalanced**

## Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the chart below to support your work. This 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.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 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) |

1 Resubmit pseudocode from previous pseudocode assignments and update as necessary.

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.

**Void procedureTry { CourseFile(String filename) {**

File path: new File—(filename)

Scanner scanner: brand-new File Scanner

CourseNumbers = new HashSet<>(); Set<String>

while (hasNextLine() in scanner) The string line is equal to scanner.nextLine().trim()

When line.isEmpty() is encountered, the following code is generated: print("Error: Empty line") continue

parts in String[] = line.split("\\s+")

if (length of sections < 2) { print("Error: Insufficient parameters") continue print("Error: Insufficient parameters")

courseNumber in the string = parts[0]

requirements = Arrays.asList(Arrays.copyOfRange(parts, 1, parts.length)) for List<String> prerequisites

If (courseNumbers.contains(courseNumber)) { print("Error: Duplicate course number") continue

for (String precondition: preconditions) { ....... if (!courseNumbers.contains(prerequisite)) { print("Error: Prerequisite course not found - " + prerequisite) continue print("Prerequisite course not found" + prerequisite)

**procedure courseNumbers.add(courseNumber)Prerequisites, courseNumber, courseData)**

try (FileNotFoundException e) { print("Error: File not found - " + filename) } catch (IOException e)` scanner.close() Printing "Error: IO exception occurred" is what { does.

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

**Course Class String CourseThe number of List<String> requirements**

Course(List<String> requirements, String courseNumber) { this.prerequisites = prerequisites this.courseNumber = courseNumber } }

Function to import and export courses from a file

**courses = new Vector<>(); Vector<Course> createAndStoreCourses(String filename) {**

attempt { File file = new File—(filename)

New Scanner(file) = Scanner scanner

while (hasNextLine() in scanner) The string line is equal to scanner.nextLine().trim()

When line.isEmpty() is encountered, the following code is generated: print("Error: Empty line") continue

parts in String[] = line.split("\\s+")

if (length of sections < 2) { print("Error: Insufficient parameters") continue print("Error: Insufficient parameters")

courseNumber in the string = parts[0]

requirements = Arrays.asList(Arrays.copyOfRange(parts, 1, parts.length)) for List<String> prerequisites

course course = new Course(courseNumber, prerequisites) courses.add(course) course = new Course`

try (FileNotFoundException e) { print("Error: File not found - " + filename) } catch (IOException e)` scanner.close() Printing "Error: IO exception occurred" is what { does.

return classes

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

(Vector<Course> courses, String targetCourseNumber) void printCourseInfoAndPrerequisites { boolean courseFound = false;

for (Course course: courses) { print("Course Number: " + course.courseNumber) if (course.courseNumber.equals(targetCourseNumber))

print("Prerequisites: " + course.prerequisites) courseFound = true break courseFound = true }

print("Error: Course not found - " + targetCourseNumber) if (!courseFound) { }

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.

**The hash table got a full credit so I copy paste it here**

Object

**class Course:**

**function Course(course\_number, title, prerequisites)**

this.course\_number = course\_number

this.title = title

this.prereqisites = prerequisites

**function for the main program**

try

file = open(filename,’r’)

course\_table = HashTable()

for line in file

If isValidCourseFormat(couirse\_data)

course\_number, titke,prerequisites=course\_data

course =Course(course\_number,title,prereqisites)

if course\_table.contains(course\_number)

print(“Error: duplate” course\_number

throw runtime error

if len(prereqisites) < 2

print(“error not enough perams

throw new runtime error

for prereqiste in prereqisites

if not course\_table.contains(prereqiste

print “ prereqisite courseNumbeer not found”

throw runtime error

course\_table.insert(course\_number, course)

else

print “invalid format”

throw new runtime error

end for

file.close()

return course\_table

except Execption as e

print e

**function parseCourseLine(line)**

categorys = line.strip.split(“,”)

course\_number=categorys[0]

title=categorys[1]

prereqisites=componets[2:]

return course\_number title, prereqisites

**function isValidCourseFormat(course\_data)**

return len(course\_data) >=3

**function printCourseInfo(course\_table)**

for entry is course\_table.getAllEntries()

course\_numbers =entry.key

course =entry.value

print(courseNUmber + courseTitle)

if course.prereqisites

for prerequisites in course.prereqistes

print(prerequisite)

end for

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.

**BST CODE**

**Classof Study:**

Variables include: courseNumber of requirements; title; and prerequisites (List)

**ValidateFileFormat(file) is a function.**

For every line in the file:

Assign tokens to the line.

If there are less than two tokens:

"Insufficient parameters" is the print error.

Go back to False

If there are more tokens than two: concatenate the extra tokens to form the title.

For every prerequisite listed in the prerequisites:

If a requirement is absent from the tree:

"Prerequisite not found - " + prerequisite will print an error.

Go back to False

Go back to true.

**ParseFileAndCreateBST is a function.**

Make a binary search tree (BST) that is empty.

For every line in the file:

Assign tokens to the line.

If there are more than two tokens:

To make the title, concatenate several tokens.

With a courseNumber, title, and requirements, create a new course object.

To the BST, add the course object.

Go back to BST

**printCourseInfoAndPrerequisites() function (course):**

Print "Course Number: " + course.title + "Title: "

2 Create pseudocode for a menu.

**The function loadFileData(vector, hashtable, filename, BST):**

Open the file with the name filename.

In the event that the file cannot be opened:

Error message printed: "Failed to open file."

Go back

For every line in the document:

Create a course object after parsing the line.

Include the course object in the hashtable, vector, and BST.

Shut down the document.

"File data loaded into data structures successfully" is printed.

**Use the printCourseList(BST) function to traverse the BST in sequence.**

Print each course's title and course number.

**The function printCourse(hashtable, BST, courseNumber):**

If the hashtable contains no courseNumber, then:

Error message printed: "Course not found."

Go back

Take the course object out of the hashtable.

Print the title of the course.

If there are prerequisites for the course:

Print requirements

**The function displayMenu() function prints the menu selections.**

1. Import Data Structure

2. Download the Course Catalog

3. Course Print

4. Go out

**3 Design pseudocode that will print out the list of the courses**

**The printCourse function (courseNumber, BST):**

// To locate the node holding course information, use the search function.

CourseNode node = searchBST(BST, courseNumber)

// No course detected if courseNode is null.

Should courseNode be empty:

Error message printed: "Course not found."

Otherwise:

// Output "Course Title:" + courseNode.title; print the course title

// If there are prerequisites for the course

In the event that courseNode.prerequisites is not null:

"Prerequisites:" should be printed.

// Go through the list of requirements and print each one.

In courseNode.prerequisites, for every prerequisite:

Print requirement

**Function searchBST(courseNumber, node):** // Return the node if the node is null or if the course number corresponds to the node's course number

if there is a node or null.courseNumber and courseNumber are equal:

Node of return

// Search in the left subtree if the course number is less than the course number of the node.

Otherwise, if node.courseNumber < courseNumber:

If the course number is bigger, return searchBST(courseNumber, node.left)

// Print the courses' ordered list.

For every classIn sortedCourseNumbers, number:

Copy the course number.

**Function in sequenceTraversal(node):** // Create a blank list at the beginning to hold course numbers

CourseNumbers in List<String> = new List<String>()

// Carry out recursive inorder traverse.

Should node not be null:

// traverse the left subtree courseNumbers.addAll(inorderTraversal(node.left))

// Go to the active node and add the course number to the roster.

add(node.courseNumber) to courseNumbers

// Move through the right subtreeNumbers.addAll(node.right) inorderTraversal)

Course numbers returned

// Presuming that course information has previously been added to BST // Example use:

// The binary search tree that holds course data is called bst.

Printing Courses in Alpha Numeric Order (bst)

**4 Evaluation**

Let's look at the data structures involved in order to assess the program's runtime and memory usage for opening the file, reading the data, parsing each line, and ensuring that there are no formatting problems.

Vector: A vector has dynamic behavior similar to that of an array, making it possible to insert and access elements quickly. Nevertheless, there may be additional runtime overhead if the vector needs to be resized when it hits capacity.

Hashtable: Hashtables allow for insertion, deletion, and average-case lookups in constant time. On the other hand, collisions could occur, resulting in higher lookup times and possibly expanding the hashtable.

Binary Search Tree (BST): The height of the tree determines how long an operation will take to complete. The height of a balanced BST is logarithmic in relation to the number of elements, resulting in effective operations.

**Runtime**

Opening the file: In comparison to other operations, this one usually takes very little duration.

Reading data from the file: The read operation, which has a linear runtime complexity of O(n), where n is the number of lines in the file, entails reading each line from the file.

Splitting a line into tokens is the first step in the parsing process, which has a linear runtime complexity of O(m), where m is the line length.

Examining each token in the parsed line iteratively to look for formatting mistakes is a process with a linear runtime complexity of O(m), where m is the number of tokens in the line.

Taking into account these runtime complications:

Vector: The runtime complexity of reading data into a vector is O(n) since it requires linear-time operations.

Hashtable: O(n) runtime complexity is achieved by reading data into a hashtable, which also requires linear-time operations.

BST: Inserting elements one at a time is required when reading data into a BST. O(n log n) is the runtime complexity in the average situation (balanced tree), where inserting each member takes logarithmic time. On the other hand, O(n^2) runtime complexity could arise from inserting each node in a linear amount of time in the worst-case situation (an unbalanced tree).

**Memory**

Vector: The amount of memory used by a vector depends on how many elements it contains. As a result, the memory use is O(n), where n is the file's line count.

Hashtable: The amount of memory used by a hashtable depends on how many key-value pairs it stores. The starting capacity and load factor are two examples of the elements that affect memory consumption. Memory utilization is typically O(n), where n is the file's line count.

BST: The amount of memory used by a BST varies with the number of nodes it has. O(n), where n is the number of lines in the file, is also the memory utilization.

All three data structures (vector, hashtable, and BST) can open the file, read the contents, parse each line, and check for formatting issues when runtime and memory usage are taken into account. The decision, nevertheless, might also be influenced by other elements including the application's particular requirements and the anticipated size of the dataset. Even though we didn't discuss it, I would definitely use a red-black tree if I didn't know anything.

**5 Explain the advantages and disadvantages of each structure in your evaluation.**

**BST, or binary search tree:**

**Benefits**

Ordered Access: BSTs save data sorted according to course numbers, which serves as a key. This makes it possible to efficiently access elements in sorted order.

Effective Search: Using a BST to find a certain course number has an average time complexity of O(log n) (where n is the number of elements), which makes it appropriate for fast searches.

Memory Efficiency: Compared to structures that need extra metadata, BSTs can be more memory-efficient because they just need memory to store the data and pointers to the left and right child nodes.

**Drawbacks:**

Unbalanced Trees: In the worst-case situation, the tree may degenerate into a linked list if the BST is unbalanced (for example, if members are placed in sorted order). This would result in decreased performance with linear time complexity for operations such as search, insertion, and deletion.

Dynamic Structure: In order to sustain peak performance, BSTs need to be balanced properly. Certain procedures can become ineffective without balancing, particularly for skewed or unbalanced trees.

Complexity of Operations: In a BST, operations like as insertion and deletion could necessitate rebalancing and tree rearrangement, which could result in increased runtime cost and complicated implementation.

**Vector**

**Benefits**

Sequential Access: Vectors enable effective sequential access and traversal by storing elements in adjacent memory regions.

Dynamic Size: Without the need for manual resizing or reallocation, vectors can dynamically resize to accommodate more elements, providing flexible memory management.

Cache friendliness: Vectors have high cache locality, which can improve performance by lowering the number of cache misses that occur when accessing data sequentially.

**Drawbacks:**

Expensive Insertions and Deletions: Changing items at random points in a vector can necessitate expensive operations, particularly when dealing with huge vectors.

Inefficient Lookup: Although lookup operations (such as looking for a specific element) require linear search through the vector, they have an O(n) time complexity. This is in contrast to vectors, which offer efficient sequential access.

Memory Fragmentation: Regular insertions and deletions can cause memory fragmentation, which over time can affect performance and memory use.

**Hashtable:**

**Benefits**

Quick Lookups: Hashtable is effective for search operations because it offers constant-time average-case lookup (O(1)) for adding, removing, and retrieving elements.

Hashtables enable the storage of key-value pairs, which offers flexibility in the representation and retrieval of data.

Dynamic Resizing: A lot of hashtable implementations allow for dynamic resizing, which ensures effective memory use by letting the table change size dynamically based on the number of items.

**Drawbacks:**

Hash Collisions: When several keys hash to the same index, or when there are hash collisions, lookup times can decrease to O(n), particularly when collision resolution strategies like chaining are employed.

Deterministic Performance: Average-case performance is constant-time, but worst-case performance can deteriorate (for example, in the event of repeated collisions), which in some circumstances can result in unexpected performance.

Memory Overhead: When compared to more straightforward data structures, hashtables may demand more memory to maintain the hash table structure and any collision resolution procedures.

**6 Recommendation**

For the stated task of opening a file, reading data, parsing each line, and checking for formatting issues, I would advise utilizing a hashtable based on the findings of the Big O study and the analysis of the three data structures (Hashtable, Vector, and BST).

**Reasoning**

Efficiency of Lookups: Inserting, removing, and retrieving elements using a hashtable requires an average-case lookup to take constant time (O(1)). This guarantees effective retrieval of course information based on course numbers, which is beneficial for rapidly checking for formatting issues.

Dynamic Resizing: The ability to dynamically expand or contract a table according to its element count is provided by hashtables. This adaptability guarantees effective memory management and efficiency, particularly when handling a file containing a varied number of courses.

Memory Efficiency: The amount of memory needed by hashtables depends on how many key-value pairs they hold. This implies that, in contrast to data structures like vectors, which could demand memory for unused capacity, memory consumption scales with the amount of the incoming data, enabling efficient memory usage.

Versatility: Course information can be stored using course numbers as keys and course details as values. Hashtables offer versatility in encoding data as key-value pairs. This makes it possible to quickly retrieve and modify course data based on course numbers.

Balanced Performance: The overall trade-off between runtime efficiency, memory utilization, and ease of implementation makes hashtables the better choice for this particular task, even though vectors and BSTs may be advantageous in some situations, such as sequential access or ordered traversal. Hashtables are an excellent choice for meeting the demands of efficiently extracting course data from a file and checking for formatting issues because they offer a strong mix between performance characteristics and versatility.