# CS 300 Pseudocode Document

## Example Function Signatures

```
                 // Universal Code for All Storage Methods


// A Method for printing a Course
void printCourse(course) {
    print "Course Number: " + course.number
        print "Course Title: " + course.title

        // Check if there are any prerequisites to print
        if tree.course.prerequisites not empty {
            print "Prerequisite(s):"
            for each prerequisite in course.prerequisites {
                print "  " + prerequisite
            }
        }
        else {
            print "No Prerequisites"
        }
}

// Function to create a Course object
void Course(courseNumber, courseTitle, coursePrerequisites) {
    number = courseNumber
    title = courseTitle
    prerequisites = coursePrerequisites
}

// A Method for displaying the Menu for the user
void DisplayMenu() {
    // Prints the menus selections
    print("1: Load Bid Data")
    print("2: Print Courses Alphanumericly")
    print("3: Print Course Title with Prerequisites")
    print("9: Exit Program")
}


// The Main Method For BinarySearchTree
void main() {
    while (true) {
        DisplayMenu()

        inputVariable = input

        if (input = 1) {
            loadFileBST(filename)
        }
        else if (input = 2) {
            printAlphaNumBSTComputerScience(node)
        }
        else if (input = 3) {
            printCoursePrereqBST(courseNumber)
        }
        else if (input = 9) {
            return
        }
        else {
            print("Error: Not a valid input")
        }
    }
}

// Main Method For Vector
void main() {
    while (true) {
        DisplayMenu()

        inputVariable = input

        if (input = 1) {
            loadFileVector(filename)
        }
        else if (input = 2) {
            printAlphaNumVectorComputerScience()
        }
        else if (input = 3) {
            printCoursePrereqVector(courseNumber)
        }
        else if (input = 9) {
            return
        }
        else {
            print("Error: Not a valid input")
        }

    }
}

// Main Method for Hash Table
void main () {
    while (true) {
        DisplayMenu()

        inputVariable = input

        if (input = 1) {
            loadFileHash(filename)
        }
        else if (input = 2) {
            printAlphaNumHashComputerScience()
        }
        else if (input = 3) {
            printCoursePrereqHash(courseNumber)
        }
        else if (input = 9) {
            return
        }
        else {
            print("Error: Not a valid input")
        }

    }
}
```

<u>// Vector Code</u>

```
global variable Vector<Course> CourseList

void loadFileVector (String filePath) {

    attempt opening "filePath"

    if filePath not opened
        print "Error: File  could not be opened"
        return

    Array<String> courseNumbers // stores valid course numbers

    While not at the end of the file {
        read line from file
        split line by commas into an array of strings
        Array<String> tokens // stores the split string

        if number of strins < 2 {
            print "Error: Missing course Information"
        }
        else {
            String courseNumber = toekns[0]
            String courseTitle = tokens[1]
            Vector<String> prerequisites

            if tokens size > 2 {
                for each token after second {
                    if token not in courseNumbers
                        print "Error: prerequisite course does not exist for
course" + courseNumber
                     else
                        add token to prerequisite

                }
            }
            append courseNumber to courseNumbers

            Course newCourse = courseNumber, courseTitle, prerequisites
            Append newCourse to CourseList
        }
    }

    close "filePath"

}

// Method for printing all courses in order
void printAlphaNumVectorComputerScience (Vector<Course> courses) {
    // uses the in build vector selection sort
    courses.SelectionSort()
    // Prints the course in order
    for course in courses {
        printCourse(course)
    }

}

// Searches for an individual Course
Course searchCourseVector (Vector<Course> courses, String courseNumber) {


    for (i = 0; i < course.length; i++) {
        if(courses[i].courseNumber == courseNumber) {
            return courses[i]
        }
    }
    // If we reach here the Course hasn't been found
    print("Error: Course Not Found")
}

// Prints one distinct Course
void printCoursePrereqVector(String courseNum) {
    Course courseToPrint = searchCourseVector(courseNum)

    if (courseToPrint != null) {
        printCourse(courseToPrint)
    }

}
```

## // Hashing Pseudocode

```
// Opens the file and validates its contents
void loadFileHash(String filename) {
    file = open(filename)
    if file cannot be opened {
        print "Error: cannot open file."
        return
    }
    print "File opened successfully."

    // Initialize the data structures
    List<String> listofCourses
    Set<String> validCourseNumbers

    while not end of file {
        // Read and split the line into parts
        line = read(file)
        courseNumber, courseTitle, coursePrerequites = split(line, ',')

        // Validation of the course data
        if courseNumber or courseTitle is missing {
            print "Error: Missing course number or title on line", line
            continue
        }
        // Store valid course numbers and add line to list
        validCourseNumbers.add(courseNumber)
        listofCourses.add(line)
    }
    close(file)
    return listofCourses, validCourseNumbers
}

// Parsing, Validating, and Storing Data

void validateAndStoreCourses(List<String> listOfCourses, Set<String>
validCourseNumbers) {
    // Initialize the hash table probably made in the class instatiation
    HashTable<String, Course> hashTable

    for each line in listOfCourses {
        // Split the line into components
        courseNumber, courseTitle, coursePrerequisites = split(line, ',')

        // Validate prerequisites
        if coursePrerequisites exist {
            for each prerequisite in coursePrerequisites {
                if prerequisite not in validCourseNumbers {
                    print "Error: Invalid prerequisite", prerequisite, "for
course", courseNumber
                    continue
                }
            }
        }

        // Create the course object and store it in the hash table
        Course course
        course.number = courseNumber
        course.title = courseTitle
        course.prerequisites = coursePrerequisites

        hashTable.put(courseNumber, courseObject)
    }
    return hashTable
}

// Finding a specific Course

Course searchCourseHash(HashTable<String, Course> courses, String
courseNumber) {
    // Retrieve course from the hash table
    Course course = courses.get(courseNumber)

    if course is null {
        print "Error: Course not found"
        return null
    }

    return couse
}

// Method to print a specific Course
```

```
void printCoursePrereqHash(String courseNumber) {
    Course courseToPrint = searchCourseHash(courseNumber)

    if (courseToPrint != null) {
        printCourse(courseToPrint)
    }
}

// A method to print all the Courses alpha numerically
// For this I will assume that a vector exists in the data
// structure for all the keys called validCourseNumbers
void printAlphaNumHashComputerScience() {
    // Sorts the course numbers
    validCourseNumbers.Sort()

    for each courseNum in validCourseNumbers {
        printCourse(searchCourseHash(courseNum))
    }

}
```

<u>// Binary Search Tree Pseudocode</u>

```
// Function to load courses from a file into the binary search tree
void loadFileBST(filename) {
    open the file for reading
    if file not able to open {
        print: "Error: Cannot open file"
        return
    }

    // Reading the file line by line
    while there are still more lines to read {
        line = read next line
        if line is not empty {
            courseData = split line by commas
            if number of elements in courseData < 2 {
                print: "Error: Invalid course data format"
                return
            }

            courseNumber = pop courseData
            courseTitle = pop courseData
            if courseData still has elements {
                for each element in courseData {
                    prerequisites push element
                }
            }

            // Validate prerequisites
            for each prerequisite in prerequisites {
                if prerequisite not in courseTree {
                    print: "Error: Prerequisite course not found"
                    return
                }
            }
            // Create a course object
            course = new Course(courseNumber, courseTitle, prerequisites)
            // Add Course to BinarySearchTree
            addCourseToTree(root, course)
        }
    }

    close the file
}

// Function to create a Course object
void Course(courseNumber, courseTitle, coursePrerequisites) {
    number = courseNumber
    title = courseTitle
    prerequisites = coursePrerequisites
}

// Function to add the course to the binary search tree
void addCourseToTree(node, course) {
    if node is empty {
        node = new Node(course)
    }
    else {
        if couse.number < node.course.number {
            addCourseToTree(node.left, course)j
        }
        else {
            addCourseToTree(node.right, course)
        }
    }
}

// Function to print all courses and prerequisite information.
// For this we are using an in-order traversal
void printAlphaNumBSTComputerScience(node) {
    if node not null {
        // Traverse the left sub-tree first
        printAllCourses(node.left)

        // Print the current course information
        printCourse(node.course)

        // Traverse the right subtree
        printAllCourses(node.right)
    }

}

// Function to search for a specific Course
Couse searchCourseBST(courseNumber, node) {

    if (courseNumber == currentNode.course.courseNumber) {
        return currentNode.course
    }

    if (courseNumber < currentNode.course.courseNumber) {
        searchCourseBST(currentNode.leftNode)
    }
    else {
        searchCourseBST(currentNode.rightNode)
    }

}

// Function to print a specific course
void printAlphaNumBSTComputerScience(String courseNumber) {
    course courseToPrint = searchCourseBST(courseNumber, root)

    if (courseToPrint != null) {
        printCourse(courseToPrint)
    }
    else {
        print("Error: Course Not Found")
    }
}
```

**Runtime Analysis**

- **Vector Data Structure:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
|---|---|---|---|
| `Opening the file` | 1 | n | n |
| `Reading each line` | 1 | n | n |
| `Parsing each line` | 1 | n | n |
| `Checking for Errors` | 1 | n | n |
| `Creating a Course and Append Vector` | 2 | n^2 | 2n^2 |
| **Total Cost** | | | 2n^2 + 4n |
| **Runtime** | | | O(n^2) |

-

    - The overall time complexity is n^2 due to the fact that during insertions the vector might need to be resized. However, Vectors use contiguous memory with the exception of resizing which can lead to reallocating memory
    - **Advantages:**
        - Allows for quick lookups for reasonably sized projects and
        - Very simple to implement
    - **Disadvantages:**
        - Expensive resizing as we can see by the possible resizing of the vector.
        - Along with resizing comes higher memory usage.

| Code | Line Cost | # Times Executes | Total Cost |
|---|---|---|---|
| `Opening the file` | 1 | n | n |
| `Reading each line` | 1 | n | n |
| `Parsing each line` | 1 | n | n |
| `Checking for Errors` | 1 | n | n |
| `Creating a Course, Storing the Key, and Hashing to location` | 3 | n^2 | 3n^2 |
| **Total Cost** | | | 3n^2 + 4n |
| **Runtime** | | | O(n^2) |

- **Hash Table:**

  - The overall time complexity is O(n^2) in the worst case although with a correctly sized table for the data it is actually closer to O(n). Hash tables will typically use more memory due to the hashing function and overhead for storing nodes.
  - **Advantages:**
    - On average insertions and lookups are constant time
    - Efficient search and insertion for large datasets
  - **Disadvantages:**
    - Collisions can cause the worst case insertions of n^2
    - Has a higher memory usage for storing keys values and handling collisions.

| Code | Line Cost | # Times Executes | Total Cost |
|---|---|---|---|
| `Opening the file` | 1 | n | n |
| `Reading each line` | 1 | n | n |
| `Parsing each line` | 1 | n | n |
| `Checking for Errors` | 1 | n | n |
| `Creating a Course and Inserting into the Tree` | 2 | n^2 | 2n^2 |

| Code | Line Cost | # Times Executes | Total Cost |
|---|---|---|---|
| `Opening the file` | 1 | n | n |
| `Reading each line` | 1 | n | n |
| `Parsing each line` | 1 | n | n |
| `Checking for Errors` | 1 | n | n |
| **Total Cost** | | | 2n^2 + 4n |
| **Runtime** | | | O(n^2) |

- **Binary Search Tree (BST):**

  - Over all the time complexity is O(nlog(n)) in the average case, and O(n^2) in the worst case. BST's use pointers to store the tree structure which results in a higher memory overhead compared to vectors but less than hash tables.
  - **Advantages:**
    - Since a BST maintains a sorted order, it is ideal for tasks that require sorted data.
    - As long as the tree is balanced search, insertion, and deletion are efficient.
  - **Disadvantages:**
    - If the tree becomes unbalanced, insertion and search can degrade to linear time
    - Large memory usage.

**Recommendation:**

Based on the analysis of the data structures, I choose the Binary Search Tree. The BST offers an ideal balance between memory usage and efficient searching, insertion and sorting. It also maintains data in a pre sorted format, which is essential for this project. We will have to take steps to ensure that the table remains balanced. I think the sacrifice we make with more memory usage by storing pointers is manageable and offset by the quick access time and sorted nature of this data structure.