# CS 300 Pseudocode Document

## Function Signatures

**// Vector pseudocode**

void loadFile() {

CREATE variable to hold stream for open file

IF variable is -1

ERROR file not found

ELSE

WHILE not end of file

FOR EACH LINE

IF < 2 strings // meaning less than 2 parameters

RETURN error

CONTINUE

ELSE

CONTINUE

FOR EACH parameter after first 2 parameters

IF parameter has already been read

CONTINUE

ELSE

RETURN error

CLOSE file

}

void createCourseObjects(Vector<Course> courses) {

OPEN file

WHILE not end of file

CREATE empty Course object

FOR EACH line in file

SET course object’s number and title to first 2 strings in line

FOR EACH parameter after 2nd

ADD each to object using built in function

INSERT new course into Vector

CLOSE file

}

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

FOR EACH course in courses

IF courseNumber matches courseNumber (param)

PRINT course information

FOR EACH prerequisite of current course

printCourseInformation(courses, prerequisite courseNumber)

BREAK from function

}

int partition(vector<Course> courses, int begin, int end) {

CREATE 2 variables to hold high and low index

SET lowIndex to begin and highIndex to end

CREATE variable to hold midpoint of begin and end

CREATE course variable and set to the course at midpoint

CREATE bool variable and SET to false

WHILE !done

WHILE Courses[lowIndex]’s course # < pivot’s course #

INCREMENT lowIndex by 1

WHILE pivot’s course # < Courses[highIndex]’s course #

DECREMENT highIndex by 1

IF lowIndex >= highIndex

SET done = true

ELSE

SWAP the Courses[lowIndex] with Courses[highIndex]

INCREMENT lowIndex by 1

DECREMENT highIndex by 1

RETURN highIndex

}

void quickSort(vector<Course> courses, int begin, int end) {

CREATE 2 variables to hold high and low index

SET low index to begin and high index to end

IF low index is >= high index

RETURN

CREATE low end index variable

SET lowEndIndex = partition(courses, lowIndex, highIndex)

// Recursive call to sort low partition

quicksort(courses, lowIndex, lowEndIndex)

// Recursive call to sort high partition

quickSort(courses, lowEndIndex + 1, highIndex)

}

void printAll() {

quicksort(Courses, 0, Courses.size() - 1)

FOR each course in Courses

OUTPUT course information

}

**// Hashtable pseudocode**

void loadFile() {

CREATE variable to hold stream for open file

IF variable is -1

ERROR file not found

ELSE

WHILE not end of file

FOR EACH LINE

IF < 2 strings // meaning less than 2 parameters

RETURN error

CONTINUE

ELSE

CONTINUE

FOR EACH parameter after first 2 parameters

IF parameter has already been read

CONTINUE

ELSE

RETURN error

CLOSE file

}

void Insert(Hashtable<Course> courses, Course course) {

CREATE variable to hold key using course number

CREATE new Node pointer (oldNode) and set to node at key location

IF oldNode == nullptr

CREATE new Node pointer (newNode) with bid and key as arguments

INSERT newNode into courses using key location

ELSE

IF oldNode key == UINT\_MAX

SET oldNode’s key = key

SET oldNode’s course = course

SET oldNode’s next node = nullptr

ELSE

WHILE oldNode’s next node != nullptr

SET oldNode = oldNode’s next node

SET oldNode’s next node = new Node(course, key)

}

void createCourseObjects(Hashtable<Course> courses) {

OPEN file

WHILE not end of file

CREATE empty Course object (course)

FOR EACH line in file

SET course object’s number and title to first 2 strings in line

FOR EACH parameter after 2nd

ADD each to object using built in function

Insert(courses, course)

}

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

FOR EACH course in courses

IF courseNumber matches courseNumber (param)

PRINT course information

FOR EACH prerequisite of current course

printCourseInformation(courses, prerequisite courseNumber)

BREAK from function

}

void PrintAll() {

FOR each node in nodes

IF node’s key != UINT\_MAX

OUTPUT node’s course information

CREATE tempNode = node’s next node

WHILE tempNode != nullptr

OUTPUT tempNode’s information

SET tempNode = tempNode’s next node

}

**// Tree pseudocode**

void loadFile() {

CREATE variable to hold stream for open file

IF variable is -1

ERROR file not found

ELSE

WHILE not end of file

FOR EACH LINE

IF < 2 strings // meaning less than 2 parameters

RETURN error

CONTINUE

ELSE

CONTINUE

FOR EACH parameter after first 2 parameters

IF parameter has already been read

CONTINUE

ELSE

RETURN error

CLOSE file

}

void Insert(Tree<Course> courses, Course course) {

CREATE new node with course as the data

IF root == nullptr

SET root == newNode

SET newNode’s left & right node = nullptr

ELSE

CREATE new node currNode

WHILE currNode != nullptr

IF newNode’s bidId < currNode’s bidId

IF currNode’s left node == nullptr

SET currNode’s left node = newNode

SET currNode = nullptr

ELSE

SET currNode = currNode’s left node

ELSE

IF currNode’s right node == nullptr

SET currNode’s right node = newNode

SET currNode = nullptr

ELSE

SET currNode = currNode’s right node

SET newNode’s left and right node = nullptr

}

void createCourseObjects(Tree<Course> courses) {

OPEN file

WHILE not end of file

CREATE empty Course object

FOR EACH line in file

SET course object’s number and title to first 2 strings in line

FOR EACH parameter after 2nd

ADD each to object using built in function

Insert(courses, course)

}

void <Tree>::InOrder() {

this->inOrder(root)

}

Void inOrder(Node\* node) {

IF node == nullptr

Return

inOrder(node->left)

OUTPUT node course information

inOrder(node->right)

}

**// Menu pseudocode**

void showMenu() {

int choice = 0;

WHILE choice != 9   
 CLEAR screen

OUTPUT select 1 to load data

OUTPUT select 2 to display all courses

OUTPUT select 3 to display specific course

OUTPUT select 9 to exit

GET user input for choice

SWITCH based on variable choice

CASE 1

loadFile()

BREAK

CASE 2

CALL function to display all courses

BREAK

CASE 3

CREATE variable to hold integer

GET user input

printCourseInformation(<data structure variable>, <user input>)

BREAK

END SWITCH

END WHILE

EXIT call

}

**Runtime Analysis**

| **Vector loadFile()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create variable for file stream and check if == -1** | 1 | 1 | 1 |
| **WHILE not end of file** | 1 | n | N |
| **FOR EACH line** | 1 | n | N |
| **Check amount of strings returning error if < 2** | 1 | n | N |
| **FOR each string after 2nd string** | 1 | n | N |
| **Check if string has already been read** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | n(n(n + 1) + 1) |
| **Runtime** | | | O() |

| **Vector createCourseObjects()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **WHILE not end of file** | 1 | n | N |
| **CREATE empty object** | 1 | 1 | 1 |
| **FOR each line in file** | 1 | n | N |
| **Set empty objects variables** | 1 | 1 | N |
| **FOR each string after 2nd** | 1 | n | N |
| **ADD string object to empty object** | 1 | 1 | 1 |
| **INSERT new course into vector data structure** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | n(n(n + 1) + 1) |
| **Runtime** | | | O() |

| **Hashtable loadFile()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create variable for file stream and check if == -1** | 1 | 1 | 1 |
| **WHILE not end of file** | 1 | n | N |
| **FOR EACH line** | 1 | n | N |
| **Check amount of strings returning error if < 2** | 1 | n | N |
| **FOR each string after 2nd string** | 1 | n | N |
| **Check if string has already been read** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | n(n(n + 1) + 1) |
| **Runtime** | | | O() |

| **Hashtable Insert()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **CREATE variable to hold key and another to hold a pointer** | 1 | 1 | 1 |
| **Check if node pointer is null** | 1 | 1 | 1 |
| **CREATE new node pointer** | 1 | 1 | 1 |
| **INSERT new node into courses table at key location** | 1 | 1 | 1 |
| **ELSE** | 1 | 1 | 1 |
| **IF key == UINT\_MAX** | 1 | 1 | 1 |
| **SET old node’s key, course, and next node to appropriate values** | 1 | 1 | 1 |
| **ELSE** | 1 | 1 | 1 |
| **WHILE old node’s next node != nullptr** | 1 | n | N |
| **SET old node’s next node = next node** | 1 | 1 | 1 |
| **SET old node’s next node = new node with course and key as args** | 1 | 1 | 1 |
| **Total Cost** | | | n + 10 |
| **Runtime** | | | O(n) |

| **Hashtable createCourseObjects()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **OPEN file** | 1 | 1 | 1 |
| **WHILE not end of file** | 1 | n | N |
| **FOR EACH line** | 1 | n | N |
| **SET course objects number and title** | 1 | 1 | 1 |
| **FOR each string after 2nd string** | 1 | n | N |
| **Add to course object** | 1 | 1 | 1 |
| **Call insert function from above using courses table and new course as args** | 1 | 1 | N |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | n(n(n + 1 + n) + 1) |
| **Runtime** | | | O() |

| **Tree loadFile()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create variable for file stream and check if == -1** | 1 | 1 | 1 |
| **WHILE not end of file** | 1 | n | N |
| **FOR EACH line** | 1 | n | N |
| **Check amount of strings returning error if < 2** | 1 | n | N |
| **FOR each string after 2nd string** | 1 | n | N |
| **Check if string has already been read** | 1 | 1 | 1 |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | n(n(n + 1) + 1) |
| **Runtime** | | | O() |

| **Tree Insert()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **CREATE new node variable to hold course data from args** | 1 | 1 | 1 |
| **Check if root is null and set new node to root is so** | 1 | 1 | 1 |
| **ELSE create new current node** | 1 | 1 | 1 |
| **WHILE current node is not null, go down tree left if course # is lower that current node and right if higher until null node is found** | 1 | n | N |
| **Total Cost** | | | N + 1 + 1 + 1 |
| **Runtime** | | | O(n) |

| **Tree createCourseObjects()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **OPEN file** | 1 | 1 | 1 |
| **WHILE not end of file** | 1 | n | N |
| **FOR EACH line** | 1 | n | N |
| **SET course objects number and title** | 1 | 1 | 1 |
| **FOR each string after 2nd string** | 1 | n | N |
| **Add to course object** | 1 | 1 | 1 |
| **Call insert function from above using courses table and new course as args** | 1 | 1 | N |
| **Close file** | 1 | 1 | 1 |
| **Total Cost** | | | n(n(n + 1 + n) + 1) |
| **Runtime** | | | O() |

**Evaluation Questions**

**Based on the advisor’s requirements, analyze each data structure (vector, hash table, and tree). Explain the advantages and disadvantages of each structure in your evaluation.**

When comparing each of the data structures, the loadFile function for each is the same. The loadFile function is just a function to go though and determine if the file and its contents are applicable when creating an object. Due to its relatively compatible functionality, it has the same runtime complexity for all 3 data structures. The next main function that was analyzed was the function required to create the course objects and then add them to the list of other objects. Now, the vector function for this had a runtime complexity of O(), whereas both the hashtable and tree data structure had a runtime complexity of O(). This is due to the hashtable and tree structure requiring a separate insert function to add the node data into their respective spot We can however sort the vector using a separate quicksort function that splits the vector recursively and swaps the nodes if they are in the wrong spot. A problem with vector structures is that with a lot of data, finding the lowest value or a node with a certain name, this can take quite some time since you must iterate through all nodes until the correct one is found. Also, due to how hashtable elements are placed in a key value pair based on the key’s hashed value, there is no real sorting. This makes hashtables more viables for retrieving data in large quantities. On the other hand, tree data structures are great at having data organized by a common factor. However, when parsing the data into the structure, the root node would be the first node that is parsed in. This can create a problem down the line as this is the first node that determines which way the next nodes go, left or right. In the case that this node is either one of the higher or lower value nodes, most of the nodes that follow will only go to one side since they will mostly be higher or lower than the root if the root is mainly on one end. This means that if most nodes are to one side of the branch, it will take much longer to iterate one branch than the other, increasing runtimes.

**Now that you have analyzed all three data structures, make a recommendation for which data structure you will plan to use in your code. Provide justification for your recommendation, based on the Big O analysis results and your analysis of the three data structures.**

The data structure that I plan on using in my code will be the tree structure. As stated previously, this structure I feel is relatively easier to work with due to its ability to keep the tree sorted and will be easier to iterate through since each branch that you go down also gets rid of total amount that still needs to be iterated. Since this data will need to be read out in the order of the course number and can be search rather quickly for a specific course, I will be using the tree data structure in my code.