# CS 300 Pseudocode Document

## Example Function Signatures

//Vector - Milestone 1

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

**Create Course Array/list**

GET file path from user OPEN CSV file

DEFINE Class Course

Private courseId

Private courseName

Private coursePrerequisite1

Private coursePrerequisite2

Public setCourseId()

Public getCourseId()

Public setCourseName()

Public getCourseName()

Public setCoursePrequiste1()

Public getCousePrequiste1()

Public setCoursePrequiste2()

Public getCousePrequiste2()

CourseConstructor(courseId, courseName, coursePrerequiste1, coursePrerequiste2)

FOR Each line IN file:

fields = SPLIT(string, ‘, ’)

IF parameters are less than two Throw “Missing Parameter” error

IF parameters three and four do not exist set to “none”

CREATE Course with CourseConstructor()

ADD course to CoursesList

END

**Print All Courses**

FOR Each element IN CourseList:

PRINT GetCourseId

PRINT GetCourseName

PRINT GetCoursePrequiste1

PRINT GetCoursePrequiste2

END

**Search for Specific Course**

GET user input for courseSearch

FOR Each element in CourseList:

IF courseSearch equals courseID

PRINT Course Data

ELSE IF courseSearch equals courseName

PRINT Course Data

ELSE INCREMENT

END

}

//Hash Table - Milestone 2

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

**Parse CSV file**

GET file path from user OPEN CSV file

DEFINE Class Course

Private courseId

Private courseName

Private coursePrerequisite1

Private coursePrerequisite2

Public setCourseId()

Public getCourseId()

Public setCourseName()

Public getCourseName()

Public setCoursePrequiste1()

Public getCousePrequiste1()

Public setCoursePrequiste2()

Public getCousePrequiste2()

CourseConstructor(courseId, courseName, coursePrerequiste1, coursePrerequiste2)

FOR Each line IN file:

fields = SPLIT(string, ‘, ’)

IF parameters are less than two Throw “Missing Parameter” error

IF parameters three and four do not exist set to “none”

CREATE Course with CourseConstructor()

ADD course to CoursesList

END

**CONSTRUCT HASH TABLE**

DEFINE int TABLE\_Size;

DEFINE Class Course

String courseNumber;

String courseName;

String prerequisite1;

String prerequisite2;

courseNumber\* next;

DEFINE HashTable

Course\* table[TABLE\_SIZE]

DEFINE CourseTable()

FOR (int i=0; i < TABLE\_SIZE; i++)

Course[i] = nullptr;

CONVERT courseName to hashFunction (courseNumber)

courseNameAscii

RETURN courseNameAscii % TABLE\_SIZE

**INSERT NEW BUCKET**

int index = hashFunction(courseName)

course\* newCourse = new Course

newCourse->courseNumber = courseNumber

newCourse->courseName = courseName

newCourse->prerequiste1 = prerequiste1

newCourse->prerequiste2 = prerequiste2

newCourse->next = nullptr

IF (table[index] == nullptr)

table[index] = newCourse

ELSE {

Course\* current = table[index]

WHILE (current->courseNumber != nullptr)

current = current->next

current->next = newCourse

**PRINT HASH TABLE**

Index = hashFunctions(courseNumber)

Course\* current = table[index];

WHILE (current != courseNumber)

IF (current->courseNumber == courseNumber)

PRINT current->course

}

//Binary Search Tree – Milestone 3

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

**Parse CSV file**

GET file path from user OPEN CSV file

DEFINE Class Course

Private courseId

Private courseName

Private coursePrerequisite1

Private coursePrerequisite2

Public setCourseId()

Public getCourseId()

Public setCourseName()

Public getCourseName()

Public setCoursePrequiste1()

Public getCousePrequiste1()

Public setCoursePrequiste2()

Public getCousePrequiste2()

CourseConstructor(courseId, courseName, coursePrerequiste1, coursePrerequiste2)

FOR Each line IN file:

fields = SPLIT(string, ‘, ’)

IF parameters are less than two Throw “Missing Parameter” error

IF parameters three and four do not exist set to “none”

CREATE Course with CourseConstructor()

ADD Course to CourseList

END

**CONSTRUCT TREE**

DEFINE TreeNode(course)

root = course

left = nullptr

right = nullptr

return newNode

INSERT(Node, root)

FOR each item in

IF root is null

Node = createNode(root)

Else if root treeeNode.Root

Node.left = insert(Node.left, root)

Else

Node.right = insert(Node.right, root)

**PRINT TREE**

IF root is not null

Print(node.left)

Print(node.root)

Print(node.right)

END

**MENU**

WHILE selection DOES NOT equal 9

PRINT menu options

“1. Load CSV File”

“2. Print Courses Alphanumerical Order”

“3. Print Course title and the prerequisites for any individual course”

“9. Exit the program”

GET user selection

TRY selection is 1, 2,3,9

CATCH

PRINT “Error Invalid Selection”

IF “1” Load CSV into Data Structure

IF “2” PRINT Courses in Alphanumeric Order

IF “3” PRINT Courses with prerequisites

IF “9” PRINT “Goodbye” then RETURN

END

**PRINT ALPAHNUMERIC ORDER**

Quicksort(CourseList)

WHILE array length is greater than one

FOR each object in CourseList

SET pivot to array length / 2

LEFT is each element < pivot

MIDDLE is each element == to pivot

RIGHT is each element > pivot

RETURN quicksort(left) + middle + quicksort(right)

## Runtime Analysis

**VECTOR**

FOR Each element in CourseList:

IF courseSearch equals courseID

PRINT Course Data

ELSE INCREMENT

END

|  | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For Each element in CourseList** | 1 | n | n |
| IF courseSearch Equals courseID | 1 | n | n |
| **Print Course Data** | 1 | 1 | 1 |
| ELSE INCREMENT | 1 | n | 1 |
| **Total Cost** | | | 3n + 1 |
| **Runtime** | | | O(n) |

**HASHTABLE**

**Function Search(courseIndex)**

PRINT data from Index location

|  | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| PRINT Data from Index location | 1 | 1 | 1 |
| **Total Cost** | | | 1 |
|  | | | O(1) |

**BINARY SEARCH TREE**

**Function search(root, key)**

**IF root is null or root.key is equal to key**

**RETURN root**

**IF key is > the root key search right**

**Return root.right, key**

**IF key < root key then search left**

**Return root.left, key**

|  | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If root = null or root.key = to key** | 1 | n | n |
| Return root | 1 | 1 | 1 |
| **IF key > root search right** | 1 | n | n |
| Return root.right, key | 1 | n | n |
| **IF key < root key then search left** | 1 | n | n |
| **Return root.left,key** | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 2 |
| **Runtime** | | | O(n) |

**EVALUATION**

For the advisors I am going to recommend that we use a hash table. I feel that this will best suit their needs. A hash table provides fast look up with runtime O(1). It may take them longer to add or delete courses, but this probably would not be a daily occurrence. The vector would give them quick access to add and delete courses, but you to search the for data it would take some time to find. I would not select a binary search tree because if the tree is not balanced the runtimes for searches could get higher. While adding and deleting could be potentially faster with O(log(n)) in a worst case scenario it is O(n).