# 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

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) {

for each course c in courses

for each prerequisite p of course c

print p

print c

}

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**

}

// Hashtable pseudocode

int numPrerequisiteCourses(Hashtable<Course> courses) {

for each course c in courses

for each prerequisite p in course c.prerequisites:

numPrerequisites++

return numPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

for each course c in courses

for each prerequisite p of course c

print p

print c

}

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

for each course c in courses

if courseNumber is the same as c.courseNumber

print course information

for each prerequisite p of c.prerequisites

print prerequisite course information

}

// Tree pseudocode

// Print the number of prerequisites for a given course

int numPrerequisiteCourses(Tree<Course> courses, string courseNumber, Node\* node) {

if node is null

return

int prerequisites = 0

if node.courseNumber == courseNumber

for each prerequisite p in node->course

prerequisites++

return prerequisites

if courseNumber < node->course.courseNumber

numPrerequisiteCourses(courses, courseNumber, node->left)

else if courseNumber > node->course.courseNumber

numPrerequisiteCourses(courses, courseNumber, node->right)

}

// Prints the required prerequisites, then the course as a schedule // for a given course

void printSampleSchedule(Tree<Course> courses, string courseNumber, Node\* node) {

if node is null

return

if node.courseNumber == courseNumber

for each prerequisite p in node->course

print p

print node.courseNumber, node.courseName

return

if courseNumber < node->course.courseNumber

numPrerequisiteCourses(courses, courseNumber, node->left)

else if courseNumber > node->course.courseNumber

numPrerequisiteCourses(courses, courseNumber, node->right)

}

// Prints the course number, name, and prerequisites of a given course

void printCourseInformation(Tree<Course> courses, String courseNumber, Node\* node) {

if node is null

return

if node.courseNumber == courseNumber

print node.courseNumber, node.courseName

for each prerequisite p in node->course

print p

return

if courseNumber < node->course.courseNumber

numPrerequisiteCourses(courses, courseNumber, node->left)

else if courseNumber > node->course.courseNumber

numPrerequisiteCourses(courses, courseNumber, node->right)

}

**Pseudocode**

**Course Object**

Class course:

String courseNumber

String courseName

Vector<String> prerequisites

**Determining if a course number is a valid course**

INITIALIZE new Vector<String> for valid course numbers

For each line in input file

Split line by “,”

INSERT string before first comma into course numbers vector, with whitespace stripped

**Loading into Vector**

INITIALIZE new Vector for courses

FOR EACH line in input file:

IF number of parameters in line >= 2:

INITIALIZE new Course object

SET course.courseNumber to String before first comma

SET course.courseName to String after first comma before second comma

IF number of parameters > 2:

SPLIT rest of line by “,”

FOR EACH prerequisite:

IF current prerequisite is in courseNumbers vector:

APPEND current prerequisite to course.prerequisites vector

ELSE:

THROW new error and notify user of invalid prerequisite

continue

INSERT course into courses Vector

ELSE:

THROW new error and notify user of invalid data

**Loading into HashTable**

INITIALIZE new HashTable for courses

FOR EACH line in input file:

IF number of parameters in line >= 2:

INITIALIZE new Course object

SET course.courseNumber to String before first comma

SET course.courseName to String after first comma before second comma

IF number of parameters > 2:

SPLIT rest of line by “,”

FOR EACH prerequisite:

IF current prerequisite is in courseNumbers vector:

APPEND current prerequisite to course.prerequisites vector

ELSE:

THROW new error and notify user of invalid prerequisite

continue

INSERT KEY=courseNumber, VALUE=Course object into courses HashTable

ELSE:

THROW new error and notify user of invalid data

**Loading into BST**

INITIALIZE new BST for courses

FOR EACH line in input file:

IF number of parameters in line >= 2:

INITIALIZE new Course object

SET course.courseNumber to String before first comma

SET course.courseName to String after first comma before second comma

IF number of parameters > 2:

SPLIT rest of line by “,”

FOR EACH prerequisite:

IF current prerequisite is in courseNumbers vector:

APPEND current prerequisite to course.prerequisites vector

ELSE:

THROW new error and notify user of invalid prerequisite

continue

INSERT Node into BST using recursive addNode function

ELSE:

THROW new error and notify user of invalid data

**Menu**

OUTPUT menu as follows:

1. Load course data
2. Print course list
3. Print course
4. Exit

INPUT user choice

SWITCH user choice

Case 1:

Load course data into data structure using above code

Case 2:

Print course list using above code

Case 3:

OUTPUT “Enter course number: “

INPUT user course number

Print given course

Case 4:

Exit program

**Sorting and Printing Vector**

**Quick Sort**

SET partition index to 0

IF begin parameter is greater than end parameter, return. There are no bids to sort

PARTITION bids from begin parameter to end parameter

Recursively call Quick Sort on low partition

Recursively call Quick Sort on high partition

**Partition**

SET low integer variable to begin parameter

SET high integer variable to end parameter

SET middle string variable to middle of current partition

WHILE low is less than high:

WHILE bids[low].title is less than bids[pivot]:

INCREMENT low variable

END WHILE

WHILE bids[high].title is greater than bids[pivot]:

DECREMENT high variable

END WHILE

IF low is less than high:

SWAP bids[low] and bids[high]

INCREMENT low variable

DECREMENT high variable

END WHILE

Return high variable

For each course c in courses vector

Print c.courseNumber, c.courseName

For each prerequisite p in c.prerequisites

Print p

**Sorting and Printing HashTable**

// HashTables are inherently unsorted data structures, so we place all

// elements into a vector and sort that

INITIALIZE Vector<course> courses

For each node in HashTable’s Vector of nodes

WHILE node->next != nullptr

INSERT node->course into courses vector

Sort courses vector using QuickSort

For each course c in courses vector

Print c.courseNumber, c.courseName

For each prerequisite p in c.prerequisites

Print p

**Sorting BST**

By definition, Binary Search Trees are in sorted order and can be traversed in sorted order using an InOrder Traversal on the root of the tree.

**InOrder Traversal (parameter: Node\* node)**

WHILE node != nullptr

inOrder(node->left)

OUTPUT node->course.courseNumber, node->course.courseName

For each prerequisites p in node->course.prerequisites

Print p

inOrder(node->right)

**Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE new Vector for courses | 1 | 1 | 1 |
| FOR EACH line in input file | 1 | n | n |
| IF number of parameters in line >= 2: | 1 | n | n |
| INITIALIZE new Course object | 1 | n | n |
| SET course.courseNumber to String before first comma | 1 | n | n |
| IF number of parameters > 2 | 1 | n | n |
| SPLIT rest of line by “,” | 1 | n | n |
| FOR EACH prerequisite | 1 | n \* m | n \* m |
| IF current prerequisite is in courseNumbers vector | 1 | m | m |
| APPEND current prerequisite to course.prerequisites vector | 1 | m | m |
| INSERT course into courses Vector | n | n | 2n |
| **Total Cost** | | | n \* m + 8n + 2m + 1 |
| **Runtime** | | | O(n \* m) |

**Loading Into HashTable**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE new HashTable for courses | 1 | 1 | 1 |
| FOR EACH line in input file | 1 | n | n |
| IF number of parameters in line >= 2: | 1 | n | n |
| INITIALIZE new Course object | 1 | n | n |
| SET course.courseNumber to String before first comma | 1 | n | n |
| IF number of parameters > 2 | 1 | n | n |
| SPLIT rest of line by “,” | 1 | n | n |
| FOR EACH prerequisite | 1 | n \* m | n \* m |
| IF current prerequisite is in courseNumbers vector | 1 | m | m |
| APPEND current prerequisite to course.prerequisites vector | 1 | m | m |
| INSERT KEY=courseNumber, VALUE=Course object into courses HashTable | n | n | 2n |
| **Total Cost** | | | n \* m + 8n + 2m + 1 |
| **Runtime** | | | O(n \* m) |

**Loading Into BST**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| INITIALIZE new BST for courses | 1 | 1 | 1 |
| FOR EACH line in input file | 1 | n | n |
| IF number of parameters in line >= 2: | 1 | n | n |
| INITIALIZE new Course object | 1 | n | n |
| SET course.courseNumber to String before first comma | 1 | n | n |
| IF number of parameters > 2 | 1 | n | n |
| SPLIT rest of line by “,” | 1 | n | n |
| FOR EACH prerequisite | 1 | n \* m | n \* m |
| IF current prerequisite is in courseNumbers vector | 1 | m | m |
| APPEND current prerequisite to course.prerequisites vector | 1 | m | m |
| INSERT Node into BST using recursive addNode function | n | n | 2n |
| **Total Cost** | | | n \* m + 8n + 2m + 1 |
| **Runtime** | | | O(n \* m) |

**Advantages and Disadvantages of Each Data Structure in the Context of Inserting and Sorting**

Vectors are simple and efficient data structures for inserting and sorting data. They offer dynamic resizing to allow for an unknown number of courses to be inserted into the vector. There are also many well-known and efficient sorting algorithms to perform on a vector. In the worst case, inserting an element into a vector could cost O(n) operations if the vector needs to be resized and all elements copied into a new larger memory space. QuickSort achieves a worst-case time complexity of O(n^2). Overall, a vector is a good choice for sorting an ordered collection of elements with little memory.

HashTables offer incredibly efficient lookup of elements. Well-implemented HashTables can have constant time lookup for elements. However, if hash collisions occur, worst case lookup and insertion would still be O(n). A proper hashing algorithm is crucial for HashTables to maintain their efficiency. HashTables are unfortunately unordered data structures by definition, making sorting them a hassle. The elements of a HashTable can be copied into a vector and then sorted in order to output sorted elements. HashTables also require more memory than vectors, so if memory consumption is a large concern, vectors may be a better choice.

Binary search trees are very good data structures for maintaining a sorted collection of elements. When inserting into the binary search tree, elements are inserted in sorted order as long as the correct traversal algorithm is used. This also comes with the disadvantage of not having constant time accessing of elements because the tree must be traversed in order to search for an element. If a tree becomes unbalanced, it can largely defeat the point of the tree essentially becoming a linked list. Keeping a BST balanced can be very important for the efficiency of the data structure. BSTs also require more memory consumption than vectors.

**Data Structure Recommendation**

Due to the fact that Binary Search Trees maintain sorted order, I would recommend this data structure for loading, storing, and printing course objects in sorted order. When adding a node to a BST, it is automatically added in a sorted order, so no sorting operations are needed. This handles two of the program requirements in one solution (adding and sorting). Memory consumption does not seem to be an issue as most modern machines come with plenty of memory, so a more complex data structure such as BSTs can be implemented.