# CS 300 Pseudocode Document – Project 1

See highlighted sections for Project 1 entries. I also made several edits to the pseudocode based on assignment feedback.

### //Vector pseudocode

class Course:

public members:

string courseNumber

string courseName

vector<string> prereqCourses

vector<Course> courses; //this vector will eventually hold all the courses loaded from the file

void loadCoursesFromFile(string filepath):

try:

open an input filestream with the filepath argument

if the file could not be opened

print an error message

return

else:

string line

while there are still lines to be read from the file:

vector<string> courseAttributes

string attribute

read in the line using input string stream

while there are comma-separated attributes in the line:

split the attributes in the line on each comma

append each attribute to courseAttributes

//*end of while comma separated attributes*

if the size of courseAttributes is less than 2:

print error message that the course in question needs at least two parameters

Declare new Course object, called tempCourse

Fill in all public members of Course object to tempCourse based on attributes in courseAttributes vector

First attribute will be courseNumber

Second attribute will be courseName

Any additional attributes get appended to prereqCourses vector

//I’m thinking about “prereq existence validation” at the same step as we read in a file. Assuming the first class in the file does not have a prereq, then we can get away with searching as we go. This assumes the courses are ordered chronologically as the students would take them, and not alphabetically or something else. If the very first course in the file has a prereq, then in order to confirm that prereq’s existence in the file, we might have to search the entire file… and if we did that for every single course, for 1000’s of courses, that’s pretty inefficient.

For each prereq course that is required for tempCourse (i=2; i < prereqCourses size; ++i):

search if that prereq course already exists in the courses vector

if prereq is found

break (out of for loop)

if the prereq does not exist:

print error message

//*end of For loop*

Append tempCourse to courses vector

*//end of while loop (lines in file)*

Close the file

//end of try block

Except:

Catch the exception

Print the exception message

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 PrintAllCourses(Vector<Course> courses) {

}

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

for each course in the courses vector

if course.courseNumber is the same as courseNumber

print out the course number and course title

for each prerequisite of the course

//*updating based on feedback from hash table and BST pseudocode*

Implement find(vector.begin(), vector.end(), prereqCourseNumber) to determine whether the prereq exists in the vector

If the prereq exists

Print the prereq courseNumber

Else

Print “[courseNumber] requires prereq [prereqCourseNumber], but [prereqCourseNumber] was not found in the course list”

VECTOR SORT AND PRINTING

void PrintAllCourses(Vector<Course> courses) {

//1. Sort by alphanumeric course number (I'll use insertion sort)

//two parts: sorted and unsorted

//repeatedly insert the next value of the unsorted portion into the proper location in the sorted portion

n = size of courses vector

int i, j

course temp

for (i = 1; i < n; ++i) {

j = i

// Insert numbers[i] into sorted part

// stopping once numbers[i] in correct position

while (j > 0 && courses[j].courseNumber < courses[j - 1].courseNumber) {

//if a course on the left is "greater" than the course to the right,

//the right course gets held in "temp"

//the left course gets bumped to the right

//the "temp" course gets assigned to the former left

//congrats, we just sorted... now move on!

// Swap courses[j] and courses[j - 1]

temp = courses[j]

courses[j] = courses[j - 1]

courses[j - 1] = temp

--j

//decrementing j ensures that we are comparing the current course to all previously-sorted courses to the left

}

}

//2. Print the sorted course list

for course in courses:

print courseNumber

}

### // Hashtable pseudocode

#### class Course:

public members:

string courseNumber

string courseName

vector<string> prereqCourses

private:

Course()

//Populate with empty strings

#### Class HashTable: //this table will eventually hold all the courses loaded from the file

Public: HashTable()

//populate with default Course objects

Insert(Course course)

Remove(string courseID)

Search(string courseID)

PrintAll()

Private:

Hash()

Struct Node

Course course

Int key

Node \*next

Vector<Node> courses //each “Node” in “courses” represents one course, plus other “metadata” (which only applies to the vector/hashtable, not the course itself)

#### int numPrerequisiteCourses(Hashtable< Node > courses) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

return number of total Prerequisites

}

#### void printSampleSchedule (Hashtable< Node > courses) {

//I’m assuming “courses” table will be filled with default Course objects to start

//these vectors will help organize the available courses

Create vector coursesSansPrereqs<course>

Create vector coursesWithPrereqs<course>

Create vector coursesInSched<course>

for course in courses

if course is NOT default object //*Then it’s a Node containing a Course, key, and pointer called “next”. This pointer “next” is the stem for the linked list used to handle collision.*

while course.next is not null: //*this accounts for linked lists*

if course has prereqs

append to coursesWithPrereqs

else

append to coursesSansPrereqs

//*end of For loop (course in courses)*

*//start with courses that don’t have prerequisites*

for course in coursesSansPrereqs

print course information

append course to coursesInSched

*// Continue printing courses in an order where a course never precedes its prerequisites*

While coursesWithPrereqs is NOT empty:

For course in coursesWithPrereqs:

Bool prereqsMet = true;

//*check if all prereqs are in the schedule*

For prereq in course:

If prereq NOT in coursesInSched:

//*there exists a prereq that hasn’t been scheduled yet…*

prereqsMet = false;

break;

//*end of For loop (for each prereq in a course that requires prereq)*

If prereqsMet:

Print course information

Append course to coursesInSched

Remove course from coursesWithPrereqs

}

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

define hash key given courseNumber

access the course via courses[hash key]

traverse linked list, if necessary, to account for prior collisions.

See logic in loadCoursesFromFile (below) for checking for collisions

print course name

print course number

print “Prereqs: ”

bool prereqsExist = false

for each prereq of this course

prereqsExist = true

print prereq course number

//we’re stating “Prereqs” and only printing course number, to differentiate from the actual course.

if NOT prereqsExist:

print “None!”

}

#### void loadCoursesFromFile(string filepath):

try:

open an input filestream with the filepath argument

if the file could not be opened

print an error message

return

else:

string line

HashTable\* courses

while there are still lines to be read from the file:

vector<string> courseAttributes

string attribute

read in the line using input string stream

while there are comma-separated attributes in the line:

split the attributes in the line on each comma

append each attribute to courseAttributes

*//end of while (reading line attributes) loop*

if the size of courseAttributes is less than 2:

print error message that the course in question needs at least two parameters

Declare new Course object, called tempCourse

Fill in all public members of Course object to tempCourse based on attributes in courseAttributes vector

First attribute will be courseNumber

Second attribute will be courseName

Any additional attributes get appended to prereqCourses vector

For each prereq course that is required for tempCourse (i=2; i < prereqCourses size; ++i):

search if that prereq course already exists in the courses HashTable

if prereq is found

break (out of for loop)

if the prereq does not exist:

print error message that prereq does not exist

*//end of For loop*

*//Insert tempCourse into Hash Table called “courses”*

//*Get hash key based on course number*

hashKey = GetHashKey(tempCourse.courseNumber)

//*Check if node at hash key location contains default course (empty slot)*

if courses[hashKey] is defaultCourse:

//*Replace node with course information*

courses[hashKey] = tempCourse

//*If not default (non-empty slot, either this course already inserted OR something hashed to the same key, in which case handle the collision by linked list)*

Else:

//*Check if the existing course at that key location has the same exact course number (which indicates we are trying to upload duplicate courses)*

Course currentCourse, prevCourse

currentCourse = courses[hashKey] //*currentCourse is whatever the “head node” that is occupying the first slot in the hash cell.*

prevCourse = null

//*Traverse the linked list, looking for evidence that tempCourse has already been uploaded*

*//this while loop means “until we hit a null node, without encountering tempCourse along the way…*

while currentCourse is not null and currentCourse.courseNumber != tempCourse.courseNumber:

//*prevCourse will hold the last non-null Course*

prevCourse = currentCourse

currentCourse = currentCourse.next

//*end of while (linked list traversal)*

//*at this point, currentCourse either points to null (non-duplicate scenario), or points to tempCourse (previously uploaded/duplicate scenario)*

if currentCourse is null:

// *We’ve traversed the linked list at the hash key without finding tempCourse. “Append” tempCourse by updating the “next” pointer of the present tail node.*

prevCourse.next = tempCourse

else: //*currentCourse was discovered while traversing the linked list. We don’t want to add it again.*

Print "Error, course number is not unique, course already added to master list"

*//end of while loop (reading lines from the file)*

Close the file

*//end of try block*

Except:

Catch the exception

Print the exception message

HASH TABLE SORTING AND PRINTING

void PrintAllCourses(Hashtable< Node > courses) {

*//1. Sort by alphanumeric course number*

*//Insertion sort isn't well suited to hash tables, because hash tables aren't "ordered" in the way vectors are ordered*

*//It doesn't make as much sense to compare adjacent elements.*

*//Also, my hash table uses linked lists to handle collisions, so the sorting algo needs to account for linked list*

*//Come to think of it, hash tables really aren't designed to have ordered storage... they prioritize fast retrieval.*

*//I think I will extract the elements to a vector, then sort and print the vector*

*//reminder: the hash table is filled with Nodes.*

*//Each Node contains a Course, key, and pointer called "next" to an initially empty linked list (collision storage).*

*//Each Course contains a courseNumber, courseName, and possible a list of prereqCourses*

*//The hash table is accessed by "key", like courses[4] for example*

*//The "key" is obtained by hashing the courseNumber*

*//Step 0. Extract elements*

Declare a vector called "courseList" that contains strings (courseNumber)

for element in hashTable

if element->next is NULL (no linked list, no collisions)

append (element.courseNumber) to courseList

else

*//traverse the linked list until reaching the end, identified by NULL*

while element is NOT NULL

append (element.courseNumber) to courseList

*//might need to alter this code in practice to avoid dereferencing a null pointer...*

if element-> == NULL

break //*prevent dereferencing null pointer*

element = element->next

*//Step 1. Sort the vector*

*//can either use a built-in sort(), or re-use my code from the Vector data structure pseudocode*

*//built-in would be simple to sort in place*

*//sort() is a hybrid approach, blending elements of quicksort, heapsort, and insertion sort*

*//pretty sure it is faster, cleaner, and more efficient than anything I could code myself*

sort(courseList.begin(), courseList.end())

*//Step 2. print the sorted elements*

for courseNumber in courseList:

print courseNumber

}

### // BST pseudocode

class Course:

public members:

string courseNumber

string courseName

vector<string> prereqCourses

private:

Course()

//*Populate with empty strings*

Structure “Node”

Course

Left pointer

Right pointer

Node() *default constructor with null children and default Course*

Class BinarySearchTree: //*aka “BST”, this will be populated with “Nodes”. Each “Node” will contain a “Course”, and left/right pointers (branches) to other nodes in the tree.*

Private members

Node root

void addNode(Node\* node, const Course& course) : *Helper method to recursively add a course node*

void inOrder(Node\* node) : *Helper method to recursively traverse the tree in order*

void postOrder(Node\* node) : *Helper method to recursively traverse the tree in post-order*

void preOrder(Node\* node) : *Helper method to recursively traverse the tree in pre-order*

Node\* removeNode(Node\* node, const string& courseNumber) : *Helper method to recursively remove a course node via Remove method*

void removeSubtree(Node\* node) : *Helper method to recursively remove the entire subtree via Destructor*

Public members

BinarySearchTree() : *Constructor, populates a single node with root nullptr*

~BinarySearchTree() : *Destructor*

void InOrder() : *Traverse the tree in-order*

void PostOrder() : *Traverse the tree in post-order*

void PreOrder() : *Traverse the tree in pre-order*

void Insert(Course course) : *Insert a course into the tree*

void Remove(string courseNumber) : *Remove a course from the tree*

Course Search(string courseNumber) : *Search for a course in the tree*

*//File Input*

void loadCoursesFromFile(string filepath):

try:

open an input filestream with the filepath argument

if the file could not be opened

print an error message

return

else:

string line

BinarySearchTree \***courses** = new BinarySearchTree()

while there are still lines to be read from the file:

vector<string> courseAttributes

string attribute

read in a single line using input string stream

while there are comma-separated attributes in the line:

split the attributes in the line on each comma

append each attribute to courseAttributes

*//end of while (comma-separated) loop*

if the size of courseAttributes is less than 2:

print error message that the course in question needs at least two parameters

//*Create course objects and store them in the BST*

Declare new Course object, called tempCourse

*//Fill in all public members of Course object to tempCourse based on attributes in courseAttributes vector*

First attribute will be courseNumber

Second attribute will be courseName

Any additional attributes get appended to prereqCourses vector

For each prereq course that is required for tempCourse (i=2; i < prereqCourses size; ++i):

Search if that prereq course already exists in the courses BinarySearchTree

//*I will re-use the Search() method from Mod 5*

Start at root node

* + - * While current node is not null
        + Return current node if it contains matching prereq courseNumber
        + Else traverse left (less) or right (greater)
      * If entire BST traversed and matching node not returned, return a default (empty) Course
    - If non-null Course Node is returned (pre-req found)
      * Pass (move on to next pre-req in the required list)
    - Else (null Course Node returned, pre—req not found)
      * print error message that prereq does not exist
      * Break out of For loop

//*end of For loop*

Insert tempCourse into courses BinarySearchTree by calling courses.Insert(tempCourse)

// *I will re-use my code from Mod 5 here… but for review:*

Insert() calls addNode() if BST is not null

addNode() performs string comparison of courseNumber in the current node (starting at the root) and courseNumber in tempCourse.

If it is less than the current node, recursively call addNode() to the left child.

If it is greater, recursively call addNode() to right child.

If at any time the courseNumber of tempCourse is identical to the courseNumber of the current node,

Print “Error, course number is not unique, course “ [courseNumber] “ already exists in the master list”.

Return //*it’s a void function, but this is a way to back out after encountering user/data error.*

When encountering a null child, assign tempCourse to that child node.

*//End result: BST will be “ordered” alphabetically by courseNumber*

*//I considered alternate ordering, where courses with 0 prerequisites are pushed left, and courses that have prereqs which themselves have prereqs are pushed right… but the logic got a bit messy, and it doesn’t actually improve computational time.*

//*end of while loop*

Close the file

*//end of try block*

Except:

Catch the exception

Print the exception message

//Binary Search Tree version SORT AND PRINT

*//helper function: inOrderTraversalVector…*

void BinarySearchTree::inOrderTraversalVector(Node\* node, vector<string>& courseList)

*//if node is not equal to null ptr*

if (node != nullptr) {

recursively call inOrderTraverselVector for the left node

*//push courseNumber to the vector*

courseList.push(node->course.courseNumber)

recursively call inOrderTraverselVector for the right node

void PrintAllCourses(BinarySearchTree<Node> courses)

*//reminder: The BST is filled with Nodes.*

*//Each Node contains a Course object, and left and right pointers*

*//the BinarySearchTree Class has the following public methods (defined in previous assignment)*

/\*\*\*

BinarySearchTree() : Constructor, populates a single node with root nullptr

~BinarySearchTree() : Destructor

void InOrder() : Traverse the tree in-order

void PostOrder() : Traverse the tree in post-order

void PreOrder() : Traverse the tree in pre-order

void Insert(Course course) : Insert a course into the tree

void Remove(string courseNumber) : Remove a course from the tree

Course Search(string courseNumber) : Search for a course in the tree

\*\*\*/

*//Same principle as Hash Table... I will traverse the BST, extract results to a vector, then sort and print the vector*

*//Step 0: Extraction, featuring Chris Helmsworth*

*//I will modify the in-order method to push to a vector... see above*

vector<string> courseList;

inOrderTraversalVector(courses, courseList)

*//Step 1: Sort the vector*

sort(courseList.begin(), courseList.end())

//*Step 2. print the sorted elements*

for courseNumber in courseList:

print courseNumber

*//Print out course information and prereqs*

void printCourseInformation(BinarySearchTree<Node> courses, String courseNumber) {

//*First, find the node that contains the relevant course.*

Node\* courseNode = courses.Search(courseNumber)

// *If the courseNode is null, that indicates that the courseNumber was not found in any Node.*

if (courseNode != nullptr): (if the course was successfully found)

print courseName

print.courseNumber

print "Prereqs: "

bool prereqsExist = false

// *Check if there are any prerequisites*

for each prereq of this course

prereqsExist = true

//*Check for prereq existence before printing to improve user-friendliness, per HashTable assignment feedback*

Node\* prereqFinder = courses.Search(prereq courseNumber)

If (prereqFinder!= nullptr):

print prereq course number

//*we’re stating “Prereqs” and only printing course number, to differentiate from the actual course.*

Else:

Print “Course [courseNumber] lists [prereq courseNumber] as a prerequisite, but [prereq courseNumber] was not found in the master course list”

if NOT prereqsExist:

print “None!”

//*end of For loop*

else:

Print an error message if the course is not found

## Menu pseudocode

//*this will be implemented in main()*

//*process command line arguments using a switch statement*

Declare string variables csvPath, courseNumber;

Create new switch statement that takes in int argc

Process case 2, csvPath provided but default courseNumber

Process case 3, csvPath and courseNumber provided

Process default case, csvPath and courseNumber use default values

Declare new instance of chosen DataStructure //*vector, hash table, or BST*

Declare new instance of Course object “course”

Declare int variable “choice”, initialize to 0

Declare int variable “EXIT\_CUE”, initialize to 9

While “choice” is not EXIT\_CUE

Print “Menu” and options on susbsequent lines:

1. Load Data Structure
2. Print Course List
3. Print Course Title and Prerequisites
4. Exit

Input “choice” from keyboard

Create switch statement for “choice”

Case 1

Call loadDataStructure(), pass csvPath and data structure pointer

Case 2

Call DataStructure->PrintAllCourses() to print an alphanumerically ordered list of all courses

Case 3

Assign “course” with the course object that gets returned by DataStructure->Search(courseNumber)

If the returned course is empty/default

Print a message that the course could not be found

Else

call printCourseInformation() and pass courseNumber to print the course title and prerequisites for the individual course

//*end of switch “choice”*

//*if we got here, it means the user typed “9” into the menu, cue to exit*

Print message that program is ending

Return 0 to terminate main

# 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) |

Task: analyze worst-case running time of reading the file and creating course objects for Vector, Hash Table, and BST. Line cost of running a function is the run time of that function

Vector:

| **Code** | **Line Cost** | **# Times Exe** | **Total Cost** |
| --- | --- | --- | --- |
| While lines to read in file: | 1 | n | n |
| Declare vector courseAttributes | 1 | 1 | n |
| Declare string attribute | 1 | 1 | 1 |
| While there are comma sep. att’s | 1 | 4 | 4 |
| Append each attribute to courseAttributes | 1 | 1 | 1 |
| If size courseAtt < 2, print error msg | 1 | n | n |
| Declare tempCourse | 1 | n | n |
| Populate tempCourse with courseNumber | 1 | n | n |
| Populate tempCourse with courseName | 1 | n | n |
| Populate tempCourse with prereqCourses vector | 1 | n | n |
| For each prereq course in prereqCourses vector: | 1 | 2 | 2 |
| Search() Courses for prereq course, | n | n | n2 |
| Print error message if prereq not found by end | 1 | 1 | 1 |
| Append tempCourse to Courses | 1 | n | n |
| Close the file | 1 | 1 | 1 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | n2 + 8n + 10 |
| **Runtime** | | | O(n2) |

**Advantages of Vector:**

Vector data structures are fairly simple to code, and can implement many built-in methods in C++. They are also very efficient at accessing elements. Looking up elements by a known index yields constant-time random access. Also, as the size of the input grows over time, vectors can be dynamically resized to accommodate the additional data. This data is all stored in contiguous memory and is not prone to memory leaks or inappropriate lookups.

**Disadvantages of Vector:**

My method for validating prerequisites is very costly. As the input size gets large, I would be searching a growing database once for each prereq listed for a course. Based on the data we were given, I assumed the max number of prereqs for a course was 2. However, I think it represent O(n2) behavior. For each course we add to the input of size n, we would have to make up to n comparisons to validate the existence of a single prerequisite course. As the input size grows, the number of comparisons that must be made for each course also grows at the same rate, thus exponential growth. In fact, if every course had, say, 4 prerequisites, it would more accurately be 4n2. I know this would still be categorized as O(n2), but in practice it would execute much more slowly than other O(n2) family programs. Other, more general disadvantages of Vectors are that it is inefficient to remove elements in the middle of the vector, because all other elements must be bumped down one to preserve the continuous index.

**Hash Table:**

| **Code** | **Line Cost** | **# Times Exe** | **Total Cost** |
| --- | --- | --- | --- |
| While lines to read in file: | 1 | n | n |
| Declare vector courseAttributes | 1 | 1 | n |
| Declare string attribute | 1 | 1 | 1 |
| While there are comma sep. att’s | 1 | 4 | 4 |
| Append each attribute to courseAttributes | 1 | 1 | 1 |
| If size courseAtt < 2, print error msg | 1 | n | n |
| Declare tempCourse | 1 | n | n |
| Populate tempCourse with courseNumber | 1 | n | n |
| Populate tempCourse with courseName | 1 | n | n |
| Populate tempCourse with prereqCourses vector | 1 | n | n |
| For each prereq course in prereqCourses vector: | 1 | 2 | 2 |
| Search() the Hash Table by key for prereq course | 2 | n | 2n |
| Print error message if prereq not found by end | 1 | 1 | 1 |
| Get hashKey based on courseNumber | 1 | n | n |
| Check if node at hashKey location is default | 1 | n | n |
| If default, replace with tempCourse | 1 | n | n |
| If not default, (either already inserted, or collision): | 1 | n | n |
| Declare currentCourse, prevCourse | 1 | n | n |
| Access headnode at currentCourse | 1 | n | n |
| While current !null AND currentNum != tempNum: | 1 | n | n |
| Traverse the linked list, update current and prev ptrs | 1 | n | n |
| Either append tempCourse to final node, or print error that tempCourse was already added | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 18n + 9 |
| **Runtime** | | | O(n) |

**Advantages of Hash Table:** The greatest advantage of hash tables is probably the fast lookup time, with constant-time lookup on average. Also, thanks to their ability to dynamically resize, hash tables are very adaptable to different applications. One of the most common applications is password storage, which enables servers to accurately assess whether a user has entered the correct password without actually storing all those passwords in a single location. Such a cache would be a great target for criminal hackers. Hashing the passwords allows administrators to verify that the user typed the correct password without knowing the literal password. Hash functions are also used in cryptography and cryptocurrency like Bitcoin. They enable computers to very quickly verify the integrity of a certain calculation, without having to expend all of the computer power to actually find the solution to that calculation.

**Disadvantages of Hash Table:**

For my code, the worst case scenario is that the HashTable only contains one bucket, and thus will essentially behave as a linked list when it comes to analyzing run times. That’s why I assigned time costs of “n” for traversing the linked list that handles collision in a given bucket. In practice, selecting an appropriate number of buckets based on the input size will greatly improve the run times, because it will quickly distribute that one monster linked list into many shorter linked lists in each bucket. More generally, Hash Tables do not lend themselves well to sorting and ordering. The bucket that elements are stored in are based on a hash function, which could be as simple as modulo division. Furthermore, the number on which the hash is performed might be seem random, like the middle bunch of characters in a part’s serial number, and thus the bucket in which the element lands in might seem random. Poorly designed hash functions can lead to more collisions, which both slows down future operations and does not take advantage of all the memory afforded to the hash table’s other buckets. Finally, hash tables are more complex to implement compared to vectors. Attempting to parse collisions can easily lead to memory leaks, dereferenced null pointers, or memory access violations.

**Binary Search Tree**

| **Code** | **Line Cost** | **# Times Exe** | **Total Cost** |
| --- | --- | --- | --- |
| While lines to read in file: | 1 | n | n |
| Declare vector courseAttributes | 1 | 1 | n |
| Declare string attribute | 1 | 1 | 1 |
| While there are comma sep. att’s | 1 | 4 | 4 |
| Append each attribute to courseAttributes | 1 | 1 | 1 |
| If size courseAtt < 2, print error msg | 1 | n | n |
| Declare tempCourse | 1 | n | n |
| Populate tempCourse with courseNumber | 1 | n | n |
| Populate tempCourse with courseName | 1 | n | n |
| Populate tempCourse with prereqCourses vector | 1 | n | n |
| For each prereq course in prereqCourses vector: | 1 | 2 | 2 |
| Search() the BST using Search() from assignment 5 | 2 | log(n) | 2log(n) |
| Print error message if prereq not found by end | 1 | 1 | 1 |
| Insert(tempCourse) into BST | 1 | n | n |
| Insert() calls addNode() if BST is not null | 1 | n | n |
| addNode() recursively traverses the BST until finding appropriately-located null node… continually halves the amount of data it has to parse | 1 | log(n) | log(n) |
| Assign tempCourse into that null node | 1 | n | n |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | 3 log(n) + 10n + 9 |
| **Runtime** | | | O(n) |

**Advantages of BST:** Because the BST uses recursion for many common functions like Search(), it’s runtime should be faster than Vector and HashTable. The runtime can be thought of as O(n + log(n)), but evaluates to O(n) since the linear side will grow much faster than the logarithmic operations as the input size grows. Binary Search Trees facilitate sorted traversals, since they are sorted while being added to the tree. This makes it much easier to deal with the sorting and printing task compared to the Hash Table structure. Also, BST can be accessed in a variety of ways, such as in-order, pre-order, and post-order, depending on the desired data access and context. Adding or removing a node from the BST is much easier, and less of a performance penalty, than other data structures. I think that my BST program will be easier to program than Hash Tables, and faster than both Hash Tables and Vectors.

**Disadvantages of BST:**

One of the obvious disadvantages of BSTs is that unlike Vectors and Hash Tables, we do not have the luxury of constant time lookup. Searching for a particular node will take log(n) time, since we potentially have to traverse all the way to a leaf node. Hash Tables and Vectors both allow fast access via a known position (index, or hash key), but BST is structured in such a way that does not allow quick indexing. Something we could look into in order to improve the performance of the BST is to balance the tree, ensuring that the heights of child subtrees stay relatively equal. Otherwise, we could theoretically have a tree that consisted of a single linear branch. Searching that branch would take O(n) time, since we don’t have the benefit of halving the data size as we go. Thus we would have the drawback of Vector runtime without the simplicity of Vector location lookup.

**Conclusion:**

In a perfect world, I think a directed graph would be the most suitable structure for this assignment. However, given the three choices, BST seems to be the clear winner. Although my assessment of O(n) runtime matches the Hash Table time, I think that in practice the BST will allow much faster access and will be closer to log(n) on average. O(n) represents the worst-case scenarios, as discussed above. The fact that the flowchart for this data already resembles a “tree” actually made me think that the BST would be the most suitable data structure from the start. The course diagram seems to flow from root nodes (having no prerequisites) to leaf nodes (the final courses, with no follow-on activities). It feels intuitive to translate this structure into a BST in a coding environment.