# CS 300 Project One Guidelines and Rubric

## Competencies

In this project, you will demonstrate your mastery of the following competencies:

* Apply non-coding development methodologies for outlining an algorithmic design
* Evaluate complex data structures that solve a given problem using advanced algorithmic designs

## Scenario

The academic advisors in the Computer Science department at ABCU are very happy with the pseudocode you completed. You are now prepared to expand the pseudocode to directly respond to the two items advising hopes to accomplish with this program. Remember, your program will need to do the following tasks:

1. Print a list of all the computer science courses in alphanumeric order.
2. For a given course, print out its title and prerequisites.

You will write pseudocode to address each advisor’s requirements. You will do so for each of the data structures you explored in the previous assignments, including vector, hash table, and tree. Then you will perform a runtime analysis to determine which data structure will be the best to use when you begin coding in the next project.

## Directions

In the milestones leading to this project, you wrote a description for the course object that will be stored in different data structures. To determine the running time of each of those data structures in this application, you must finish writing all the pseudocode for the rest of the code and perform a Big O analysis.

Specifically, you must address the following rubric criteria:

**Pseudocode**

1. **Resubmit pseudocode from previous pseudocode assignments and update as necessary**. In the previous assignments, you created pseudocode for each of the three data structures: vector, hash table, and tree. Be sure to resubmit the following pseudocode for each data structure:
   1. Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.
   2. Design pseudocode to show how to create course objects so that one course object holds data from a single line from the input file.
   3. Design pseudocode that will print out course information and prerequisites.

1. **Create pseudocode for a menu**. The menu will need to perform the following actions:
   1. Option 1: Load the file data into the data structure. Note that before you can print the course information or the sorted list of courses, you must load the data into the data structure.
   2. Option 2: Print an alphanumerically ordered list of all the courses in the Computer Science department.
   3. Option 3: Print the course title and the prerequisites for any individual course.
   4. Option 9: Exit the program.
2. **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.** Continue working with the Pseudocode Document linked in the Supporting Materials section. Note that you will design for the same three data structures that you have been using in your previous pseudocode milestones: vector, hash table, and tree. This time, you will create the final pieces of pseudocode that you will need for ABCU’s advising program. To complete this part of the process, do the following actions:
   1. Sort the course information by alphanumeric course number from lowest to highest.
   2. Print the sorted list to a display.

**Evaluation**

1. **Evaluate the run time and memory of data structures that could be used to address the requirements**. In previous assignments, you created pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors and to show how to create course objects so that one course object holds data from a single line from the input file.
   1. Using the pseudocode you wrote for the previous assignments, analyze the worst-case running time of each, reading the file and creating course objects, which will be the Big O value. This analysis should not include the pseudocode written for the menu or the search/print functions Print Course List (Option 2) above. To complete this part of the project, do the following actions:
      1. Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure.
      2. Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.
2. Based on the advisor’s requirements, analyze each of the vector, hash table, and tree data structures. **Explain the advantages and disadvantages of each structure in your evaluation.**
3. Now that you have analyzed all three data structures, **make a recommendation for which data structure you 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.

## Milestones

Resubmit your pseudocode for the Milestones One, Two, and Three assignments, and update it as necessary based on the feedback you received. In these assignments, you already created pseudocode for each of the three data structures:

* Milestone One: Vector
* Milestone Two: Hash Table
* Milestone Three: Binary Search Tree

Be sure to resubmit all three milestones that had the following components:

* Pseudocode designed to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors
* Pseudocode designed to show how to create course objects so that one course object holds data from a single line from the input file
* Pseudocode designed to find and print out course information and prerequisites for a single course

This submission is graded with the Final Project Part I Rubric below.

## What to Submit

To complete this project, you must submit the following item:

**Pseudocode and Runtime Analysis**  
Your submission should be formatted in a double-spaced 1- to 2-page Word document that includes your completed pseudocode, your runtime analysis in a chart, and your analysis of both the advantages and disadvantages of each structure.

**Course Structure**

DEFINE struct Course

Create a string called *id*

Create a string called *title*

Create a vector of strings called *preReqs*

END

**Parse a single line from the input file**

DEFINE a function GetCourseData(*lineOfData*) which returns a Course

IF *lineOfData* is blank

RETURN

Create a new Course called tempCourse

ASSIGN the string up to the first delimiter to tempCourse.id

IF end of *lineOfData* is reached

RETURN due to formatting error

ASSIGN the string up to the second delimiter to tempCourse.name

WHILE the end of the current line has not been reached

APPEND the string up to the next delimiter to tempCourse.preReqs

RETURN *tempCourse*

END

DEFINE a function called *PrintCourse(course)*

PRINT course id

PRINT course title

FOR each index of the *course* prerequisites

PRINT each prerequisite

END

**Main Menu**

DEFINE a function *DisplayMenu* which returns a boolean

PRINT menu option 1: Load file

PRINT menu option 2: Print all courses

PRINT menu option 3: Search for a course

PRINT menu option 9: Exit the program

END

**Quicksort**

DEFINE a function called *Quicksort*() which takes a *vector*, a *startIndex*, and an *endIndex*

IF the *startIndex s* course id is less than the *endIndex’s* course id

Create a course called *pivotIndex*

ASSIGN a CALL of *Partition* method to *pivot* passing in *vector, startIndex, endIndex*

Recursively CALL *Quicksort* using *vector, startIndex, pivotIndex*

Recursively CALL *Quicksort* using *vector, pivotIndex + 1, endIndex*

END

DEFINE a function called *Partition()* which takes a *vector*, a *startIndex*, and an *endIndex*

Create an integer called midpoint

ASSIGN the difference of *endIndex and startIndex* to *midpoint*

DIVIDE *midpoint* by 2

ADD the *startIndex* to midpoint

ASSIGN the *midpoint* of the vector to a variable called *pivot*

WHILE *running*

WHILE *startIndex’s* course id is less than the *pivot’s* course id

Increment *startIndex*

WHILE *pivot’s* course id is less than *endIndex’s* course id

Decrement *endIndex*

IF *endIndex* is greater than *startIndex*

ASSIGN *running* to false

ELSE

SWAP the courses at *startIndex* and *endIndex*

Increment *startIndex*

Decrement *endIndex*

RETURN *endIndex*

END

**Vector Methods**

DEFINE Parse(*filePath*) which returns a vector

OPEN the file at path *filePath*

Create a vector called *tempVector* to hold Course struct data

WHILE NOT end of file reachedCALL *GetCourseData* passing in the current line of the given file

APPEND *GetCourseData* return valueto *tempVector*

RETURN *tempVector*

END

DEFINE a method called *PrintSorted()*

CALL Quicksort using the courses vectorand it’s start and end indices

FOR each course in the *courses* vector

CALL *PrintCourse* passing in the current course as an argument

END

DEFINE CheckPreReqs(*coursesVector*)

FOR every course in *coursesVector*

IF *preReqs* is empty

CONTINUE to next iteration

FOR every index in *preReqs*

Search *coursesVector* to for an *id* that matches the value at *preReqs* current index

IF match found

CONTINUE with next *preReqs* index

ELSE

PRINT an error indicating a prerequisite course is not in  *coursesVector*

RETURN false

RETURN true

END

DEFINE Search(*course*)

FOR all courses

IF the current course is the same as *course*

CALL *PrintCourse*

END

**Hash Table Methods**

DEFINE a class *HashTable{}*  
 Create a private unsigned integer *tableSize*  
 DECLARE a private method *hash()* which takes an integer returns an unsigned integer  
 DEFINE a private struct *Node{}* Create Course *course*  
 Initialize unsigned integer *key* to NULL  
 Initialize Node pointer *next* to null pointer  
 DEFINE Node constructors for assigning relevant membersCreate a private vector of type Node called *hashTable*  
 DECLARE a public void method *Insert()*  
 DECLARE a public void method *PrintAll()*END

DEFINE Parse(*filePath*) which returns a *HashTable* object  
 OPEN the file at path *filePath*

Create a new HashTable object called *tempTable*

WHILE NOT end of file reachedCALL *GetCourseData* passing in the current line of the given file CALL *HashTable* *Insert* method to add *GetCourseData’s* return value to the *tempTable*

RETURN *tempTable*END

DEFINE CheckPreReqs(*hashTable*) that returns a boolean  
 FOR every node in the *hashTable* IF *preReqs* is empty  
 CONTINUE to next iteration  
 FOR every index in *preReqs* Search *hashTable* for an *id* that matches the value at *preReqs* current index  
 IF match found  
 CONTINUE with next *preReqs* index  
ELSE  
 PRINT an error indicating a prerequisite course is not in *courses* RETURN false  
 RETURN true  
END

DEFINE the hash table method *hash(key)*  
 Calculate and RETURN the hash value of the given *key*END

DEFINE the hash table method *Insert(course)*  
 Create an integer *hashKey* to hold the return value of *hash(course)*  
 Create a Node pointer *currentNode* that points to the *hashKey* location in the *hashTable* IF the *currentNode* has no valid *key*  
 Create a new node using the *course* and *hashKey*  
 ASSIGN the new node to the *hashKey* location in the *hashTable*  
 ELSE there is a collision  
 WHILE *currentNode’s* next pointer is not a null pointer  
 Move the *currentNode* to the next node  
 Create a new node using the *course* and *hashKey*  
 ASSIGN the new node to the *hashKey* location in the *hashTable*  
END

DEFINE the hash table method *PrintAll()*  
 FOR the length of the hash table  
 Create a node pointer *currentNode* to point to the current index of the hash table  
 IF the *currentNode* has a valid *key* PRINTthe currentnode’s *key*

CALL *PrintCourse* passing in the course stored in the current node  
 WHILE the *currentNode* has a *next* pointer  
 Move to the next node  
 PRINT the current node’s key

CALL *PrintCourse* passing in the course stored in the current node  
END

DEFINE the hash table method *PrintSorted()*

Create a node pointer *currentNode* to point to the head of the hash table

Create a vector called *tableAsVector* with a size of the hash table

FOR the length of the hash table

IF the *currentNode* has a valid key

APPEND the current node’s course into *tableAsVector*

WHILE the *currentNode* has a next pointer

Move to the next node

APPEND the current node’s key into *tableAsVector*

CALL Quicksort using *tableAsVector* and it’s start and end indices

FOR each course in *tableAsVector*

CALL *PrintCourse* passing in the current course as an argument

END

DEFINE a method called *Search(courseId)*

CALL *hash* using the *courseId* as an argument

IF the entry in *courses* vector is empty

RETURN

IF course is found

RETURN the course

WHILE the found node has a next pointer

iterate through the linked list

IF course is found

RETURN the course

END

**Binary Search Tree Methods**

DEFINE a class *BinarySearchTree* DECLARE a void method *addNode* which takes a node and a course as input  
 DECLARE a void method *Insert* which takes a course as input  
 DECLARE a void method *PrintAll*­ which takes a node as input  
END

DEFINE Parse(*filePath*) which returns a *BinarySearchTree* object  
 OPEN the file at path *filePath*

Create a new *BinarySearchTree* object called *tempBST*  
WHILE NOT end of file reached

Create a new Node called tempNodeASSIGN a CALL of *GetCourseData(currentLine)* to *tempNode* Call the *BinarySearchTree* method *Insert* to add *tempNode* to the tree

RETURN *tempBST*End

DEFINE a void method *Insert* which takes a course as input  
 Create a new node and point it to the root  
 IF the root node is null  
 ASSIGN a new node containing the given course to the root  
 ELSE  
 CALL *addNode* passing in the newly created node and the given course  
END

DEFINE a void method *addNode* which takes a node and a course as input  
 IF the given course’s id is less than the given node’s course id  
 IF the node’s left child is null give it a new node containing the given course  
 ELSE recursively CALL *addNode* using the left child node and the given course  
 ELSE  
 IF the node’s right child is null give it a new node containing the given course  
 ELSE recursively CALL *addNode* using the right child node and the given course  
END

DEFINE *CheckPreReqs(tree)* that returns a boolean  
 FOR every node in the *tree* IF *preReqs* is empty  
 CONTINUE to next iteration  
 FOR every index in *preReqs* Search *tree* for an *id* that matches the value at *preReqs* current index  
 IF match found  
 CONTINUE with next *preReqs* index  
ELSE  
 PRINT an error indicating a prerequisite course is not in *courses* RETURN false  
 RETURN true  
END

DEFINE a void method *PrintSorted* which takes a node as input  
 IF the given node is null then RETURN  
 Recursively call *PrintSorted* passing in the node’s left child  
 PRINT the current node’s details  
 Recursively call *PrintSorted* passing in the node’s right child  
END

DEFINE a method called *Search(courseTitle)*

ASSIGN null to a variable *tempCourse*  
 IF the root node is null  
 RETURN null  
 ELSE  
 CALL to *searchRecursive* passing in the current node and the *courseId*

ASSIGN the *searchRecursive* return value to *tempCourse*

RETURN *tempCourse*   
END

DEFINE a method *searchRecursive* which takes a node and a *courseId* as input and returns a course  
 IF the given *courseId* is less than the given node’s course id  
 IF the node’s left child is null give it a new node containing the given course  
 ELSE recursively CALL *searchRecursive* with the left child and the given *courseId*  
 ELSE  
 IF the node’s right child is null give it a new node containing the given course  
 ELSE recursively CALL *searchRecursive* with the right child and the given *courseId*

RETURN the current node’s course  
END

**Main**

DEFINE main()

Create a boolean called *running* and set it’s value to *true*

Create a container to hold parsed course data

WHILE *running* is true

CALL *DisplayMenu*

GET input from user

CASE 1: Load File

GET the file path from the user

CALL the container’s *Parse* function passing in the user’s file path

IF container’s *CheckPreReqs* function returns false

PRINT an error message

CONTINUE the *running* loop from the beginning

CASE 2: Print all courses

CALL the container’s *PrintSorted* function

CASE 3: Search for a course

CALL container’s *Search* function passing in the course id

CALL *PrintCourse* using the return value of *Search*

CASE 9: Exit program

ASSIGN *running* with a value of false

DEFAULT CASE

PRINT a message reporting invalid input

**Runtime Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GLOBAL** | **GetCourseData**(*lineOfData*) | | | |
| **Line** | | **Line Cost** | **Executions** | **Total Cost** |
| IF lineOfData is blank | | 1 | 1 | 1 |
| RETURN | | 1 | 1 | 1 |
| Create a new Course called tempCourse | | 1 | 1 | 1 |
| ASSIGN the string up to the first delimiter to tempCourse.id | | k | 1 | k |
| IF end of lineOfData is reached | | 1 | 1 | 1 |
| RETURN due to formatting error | | 1 | 1 | 1 |
| ASSIGN the string up to the second delimiter to tempCourse.name | | k | 1 | k |
| WHILE the end of the current line has not been reached | | 1 | p | p |
| APPEND the string up to the next delimiter to tempCourse.preReqs | | k | 1 | k |
| RETURN tempCourse | | 1 | 1 | 1 |
| **Total Cost** | | | | 3kp+8 |
| **Runtime** | | | | O(k) |

**Note:** k = characters read during string separation; p = number of delimiter separated prerequisites

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Vector** | **Parse**(*filePath*) | | | |
| **Line** | | **Line Cost** | **Executions** | **Total Cost** |
| OPEN the file at path filePath | | 1 | 1 | 1 |
| Create a vector called tempVector to hold Course struct data | | 1 | 1 | 1 |
| WHILE NOT end of file reached | | 1 | n | n |
| CALL GetCourseData passing in the current line of the given file | | k | n | nk |
| APPEND GetCourseData return value to tempVector | | 1 | n | n |
| RETURN tempVector | | 1 | 1 | 1 |
| **Total Cost** | | | | 2n+ nk+3 |
| **Runtime** | | | | O(nk) |

**Note**: k = characters read during string separation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **HashTable** | **Parse**(*filePath*) | | | |
| **Line** | | **Line Cost** | **Executions** | **Total Cost** |
| OPEN the file at path filePath | | 1 | 1 | 1 |
| Create a new HashTable object called tempTable | | 1 | 1 | 1 |
| WHILE NOT end of file reached | | 1 | n | n |
| CALL GetCourseData passing in the current line of the given file | | k | n | nk |
| CALL HashTable Insert method to add GetCourseData’s return value to the tempTable | | c | n | nc |
| RETURN tempTable | | 1 | 1 | 1 |
| **Total Cost** | | | | nc+nk+n+3 |
| **Runtime** | | | | O(nc) |

**Note**: k = characters read during string separation; c = chained collisions in the hash table index

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BinarySearchTree** | **Parse**(*filePath*) | | | |
| **Line** | | **Line Cost** | **Executions** | **Total Cost** |
| OPEN the file at path filePath | | 1 | 1 | 1 |
| Create a new BinarySearchTree object called tempBST | | 1 | 1 | 1 |
| WHILE NOT end of file reached | | 1 | n | n |
| Create a new Node called tempNode | | 1 | n | n |
| ASSIGN a CALL of GetCourseData(currentLine) to tempNode | | k | n | nk |
| Call the BinarySearchTree method Insert to add tempNode to the tree | | n | n | n2 |
| RETURN tempBST | | 1 | 1 | 1 |
| **Total Cost** | | | | n2+nk+2n+1 |
| **Runtime** | | | | O(n2) |

**Note**: k = characters read during string separation

**Structure Analysis: advantages and disadvantages**

**Vector**

* A vector has the advantage of small space complexity as it doesn’t require the use of nodes or an additional data structure for chaining. It is very simple to implement and can append in constant time. That said, vectors are slow if searching for an entry when the index is not known and will need to be resized as the number of entries grows. It also doesn’t handle arbitrary insertion or deletion well.

**Hash** **Table**

* When paired with a well-constructed hashing formula and an appropriate means of handling collisions, a hash table can handle building and inserting quickly. If collisions are stored using a linked list whose head contains a pointer to the tail, then the need to iterate to the tail from the head is eliminated which is how rapid build/insertion times can be achieved. Unlike a vector, hash tables do not require frequent resizing as the dataset grows because the table is generally maintained at a fixed size and defers data into a linked list via chaining. Unfortunately, hash tables are only effective if they are evenly distributed and require an equally effective hashing formula that ensures even distribution. Further, while hashing an input allows constant time lookup of the hash table location, collisions still require iteration, meaning the look-up runtime is dependent on the length of the chain at a given hash table index.

**Binary** **Search** **Tree**

* In general, binary search trees have an advantage when it comes to performing searches and insertions, particularly when building from an unordered dataset. If, however, the dataset already ordered, the tree becomes linear due to many nodes having only one child. Also, determining the location of each entry in the tree requires navigating from the root to its destination meaning that if the tree is not balanced and is instead entirely linear, the worst-case build time becomes quadratic. On the other hand, the tree is built in sorted order meaning no additional work is needed to sort the input data prior to printing all the data in alphanumeric order.

**Recommendation**

For this project I would recommend using a hash table for storing course data. This is because the size of the data input is unknown, as is it’s ordering in the input file. This does mean that the data will have to be sorted when printing the entire dataset from the hash table, however a vector would require the same treatment anyway. While a binary search tree wouldn’t require additional sorting for the sake of printing, the worst-case quadratic build time if the input data is already sorted (which the example file suggests may be the case) makes this an unattractive option. The functionality of this application suggests that the option that is likely to performed most frequently by the user is a search for an individual course, meaning that the look-up runtime plays an important role for the end user. For a vector and a BST, the look-up time could be linear. This is also true of a hash table; however, this can be addressed by simply adjusting the hash table size and the hashing function. In addition, a modified linked list for handling collisions may reduce worst case look-ups even further.