|  |
| --- |
| SNHU CS 300 |
| Pseudocode & Runtime Analysis |
| Module Six: Project One |

|  |
| --- |
| Christopher Dewey  April 4, 2022 |

Contents

[Scenario Summary 3](#_Toc100419865)

[Pseudocode 3](#_Toc100419866)

[Menu 3](#_Toc100419867)

[Open, Read, Parse, and Check for Format Errors 4](#_Toc100419868)

[Validate Course Prerequisites 5](#_Toc100419869)

[Create Course 5](#_Toc100419870)

[Vector 5](#_Toc100419871)

[Insert Course 5](#_Toc100419872)

[Sort Courses 5](#_Toc100419873)

[Print Course List 6](#_Toc100419874)

[Print Courses with Prerequisites 6](#_Toc100419875)

[Print One Course with Prerequisites 6](#_Toc100419876)

[Hash Table 7](#_Toc100419877)

[Create HashTable Class 7](#_Toc100419878)

[Insert Course 8](#_Toc100419879)

[Print Course List 8](#_Toc100419880)

[Print Courses with Prerequisites 9](#_Toc100419881)

[Print One Course with Prerequisites 9](#_Toc100419882)

[Binary Search Tree 10](#_Toc100419883)

[Create Node 10](#_Toc100419884)

[Create BST Class 10](#_Toc100419885)

[Insert Course 10](#_Toc100419886)

[Print Course List 11](#_Toc100419887)

[Print Courses with Prerequisites 11](#_Toc100419888)

[Print One Course with Prerequisites 11](#_Toc100419889)

[Evaluation 12](#_Toc100419890)

[Shared Runtime Calculations 12](#_Toc100419891)

[Vector 13](#_Toc100419892)

[Vector Runtime Analysis 14](#_Toc100419893)

[Hash Table 15](#_Toc100419894)

[Hash Table Runtime Analysis 16](#_Toc100419895)

[Binary Search Tree 17](#_Toc100419896)

[Binary Search Tree Runtime Analysis 18](#_Toc100419897)

[Recommendation 20](#_Toc100419898)

# Scenario Summary

The academic advisors in the Computer Science department at ABCU need a way to catalog and index the courses offered in the Computer Science curriculum. Pseudocode has been developed to store and read course data for three data structure types, vectors, hash tables, and binary search trees. Each of these structures are evaluated for their implementation constraints and runtime complexity and a final recommendation will be made for the construction of this project.

# Pseudocode

## Menu

The base program is text based and will run in a command prompt or PowerShell environment without arguments. Users will interact with a menu outlined in this pseudocode:

In main:

Create string: *choice*

Set choice to empty value ( “” )

While the first letter in *choice* is not ‘q’ or ‘Q’,

Print menu:

“Menu:

[1] – Load Course Data

[2] – Print Course List

[3] – Print Courses with Prerequisites

[4] – Print One Course with Prerequisites

[Q] – Quit”

Create switch (takes: first letter of *choice*)

Case ‘1’:

Call *loadCourseData*(takes: string *path*)

Break

Case ‘2’:

Call *printCourseList*

Break

Case ‘3’:

Call *printCourses*

Break

Case ‘4’:

Create string: *searchId*

Print “Please enter a course: “

Set *searchId* to user entry

Call *printCourse* (takes: *searchId*)

Break

Case ‘q’ or ‘Q’:

RETURN: “Good Bye”

Default case:

Print “[ERROR] Invalid Entry”

## Open, Read, Parse, and Check for Format Errors

The file containing all course data will be read linearly. This data is temporarily stored in two separate vectors. This pseudocode serves as a precursor to fill any data structure. Since course data is read linearly, vectors hold all temp data.

Create *loadCourseData* method (takes: *path*)

Create a vector: *courseData*

Create a vector: *courseList*

Create ifstream: *inFile*

Connect *inFile* with *path*

If *inFile* has errors,

RETURN: “[ERROR]: File could not be read.”

While *inFile* is not at end of file,

Create vector: *tempCourse*

Create string: *currentLine*

Get next line from *inFile*, assign result to *currentLine*

Create string stream: *sStream* from *currentLine*

While *sStream* has no errors

Create string: *tempString*

Set *tempString* to output from *sStream* until comma reached

Add *tempString* to *tempCourse*

Close *sStream*

If *tempCourse* has less than two components,

RETURN: “[ERROR]: Invalid format. Course name and/or description missing.”

Add *tempCourse* to *courseData*

Add first entry in *tempCourse* to *courseList*

Close *inFile*

## Validate Course Prerequisites

If a course contains prerequisites, all prerequisites must be offered by the university. If any prerequisites are missing, an error must be thrown. This pseudocode is shared between all three data structures.

For each vector in *courseData*

Create vector: *prerequisites*

For each string in vector (skip first and second),

If *courseList* does not contain string,

RETURN: “[ERROR]: Prerequisite course does not exist.”

Add string to *prerequisites*

Call *insertCourse* (pass: new Course(vector[0], vector[1], *prerequisites*))

## Create Course

Define a struct or class: *Course*

Declare string: *courseNumber*

Declare string: *name*

Declare vector: *prerequisites*

Define default constructor

Create new empty *Course*

Define overloaded constructor (takes: *courseNumber*, *name*, and *prerequisites*)

Create new *Course* with given attributes

## Vector

In main: create a vector named *courses*

### Insert Course

Create *insertCourse* method (takes: Course *course*)

Add *course* to courses

### Sort Courses

Create *partition* method (takes: *courses* and two integers: *begin* and *end* | return: integer)

Create integer: *low*, set to *begin*

Create integer: *high*, set to *end*

Create integer: *pivot*, set to (*end* – *begin*) / 2

While true,

While *courseNumber* at index *low* < *courseNumber* at index *high*,

Increment *low* by 1

While *courseNumber* at index *pivot* < *courseNumber* at index *low*,

Decrement *high* by 1

If *low* >= *high*

RETURN *high*

Else,

Swap *low* and *high*

Decrement *high* by 1

Increment *low* by 1

RETURN *high*

Create *quicksort* method (takes: *courses* and two integers: *begin* and *end*)

Create integer: *mid*, set to 0

If *begin* >= *end*,

RETURN

Set *mid* to result of *partition* (pass: *courses*, *begin*, *end*)

Call (recursive) *quicksort* (pass *courses*, *begin*, *mid*)

Call (recursive) *quicksort* (pass *courses*, *mid* + 1, *end*)

### Print Course List

Create *printCourseList* method

Call *quicksort* (pass *courses*, 0, size of *courses* -1)

For each *course* in *courses*,

Print first entry from *course*

### Print Courses with Prerequisites

Create *printCourses* method

Call *quicksort* (pass: *courses*, 0, size of *courses* -1)

For each *course* in *courses*,

Print first and second entry from *course* (same line, separate by comma and space)

If *prerequisites* is not empty,

For each *prerequisite* in *prerequisites*,

Print “, “ and *prerequisite*

### Print One Course with Prerequisites

Create *printCourse* method (takes: *searchId*)

Call *quicksort* (pass *courses*, 0, size of *courses* -1)

Create integer: *mid*, set to 0

Create integer: *low*, set to 0

Create integer: *high*, set to size of *courses* -1

While *high* >= *low*,

Set *mid* to average of *low* and *high*

If *course* (at the mid index position in courses)->courseNumber < searchId,

Set low to mid +1

Else if *course* (*mid* index in *courses*)->*courseNumber* > *searchId*,

Set *high* to *mid* -1

Else,

Print *courseNumber* and *name* (same line, separate by comma and space)

If *prerequisites* is not empty,

For each *prerequisite* in *prerequisites*,

Print “, “ and *prerequisite*

RETURN

Print “[INFO]: {*searchId*} not found.”

## Hash Table

### Create HashTable Class

Create unsigned integer: *tableSize*

Create vector: *courses*

Define class: *HashTable*

Define internal struct named *Node*

Create Course: *course*

Create unsigned integer: *key*

Create Node\*: *next*

Define default constructor

Initialize *courses* with *tableSize*

Define overloaded constructor (takes: unsigned integer: *size*)

Set *tableSize* to *size*

Initialize *courses* with *tableSize*

Create *hash* method (takes: *courseNumber* | return: unsigned integer)

Create unsigned integer: *converted*

Set *converted* to integer value of *courseNumber*

RETURN: *converted* % *tableSize*

### Insert Course

Create *insertCourse* method (takes: Course *course*)

Create unsigned integer: *key*

Set *key* to result of *hash* (pass: *course*->*courseNumber*)

Create Node\*: *oldNode*

Set *oldNode* to address of *courses* at *key*

If *oldNode* points to null,

Create Node\*: *newNode*

Insert *newNode* (call Node constructor: pass *course*, *key*) at *oldNode*->*key*

Else,

If *oldNode*->*key* contains a *node* but no *entry*,

Set *oldNode*->*course* to *course*

Set *oldNode*->*key* to *key*

Else,

While *oldNode* has an entry,

If *oldNode*->*courseNumber* < *course*->*courseNumber*

Set *oldNode* to *oldNode*->*next*

Else,

Create Node\*: *temp* to store *oldNode*

Set *oldNode*->*course* to *course*

Set *oldNode*->*key* to *key*

Set *oldNode*->*next* to *temp*

RETURN

Set *oldNode*->*next* to new Node (call Node constructor: pass *course*, *key*)

### Print Course List

Create *printCourseList* method

Iterate through *courses*

Create unsigned integer: *key*

Set *key* to *hash* (pass: *course*->*courseNumber*)

Create Node\*: *node*

Set *node* to address of *courses* at *key*

If *courses* contains a node at *key*,

While *node*->*next* does not point to null,

If the node contains an entry,

Print *course*->*courseNumber*

Else,

Set *node* to *node*->*next*

### Print Courses with Prerequisites

Create *printCourses* method

Iterate through *courses*

Create unsigned integer: *key*

Set *key* to *hash* (pass: *course*->*courseNumber*)

Create Node\*: *node*

Set *node* to address of *courses* at *key*

If *courses* contains a node at *key*,

While *node*->*next* does not point to null,

If *node* contains an entry,

Print *node*->*course*->*courseNumber* followed by a comma

Print *node*->*course*->*name*

If *node*->*course*->*prerequisites* is not null,

For each *prerequisite*,

Print “, “ and *prerequisite*

Else,

Set *node* to *node*->*next*

### Print One Course with Prerequisites

Create *printCourse* method (takes: *searchId*)

Create unsigned integer: *key*

Set *key* to hash(*searchId*)

Create Node\*: *node*

Set *node* to address of *courses* at *key*

If *courses* contains a node at *key* and *node* contains an entry,

If *node*->*course*->*courseNumber* matches *searchId*,

Print *node*->*course*->*courseNumber* followed by a comma

Print *node*->*course*->*name*

If *node*->*course*->*prerequisites* is not null,

For each *prerequisite*,

Print “, “ and prerequisite

Else,

While *node*->*next* does not point to null,

Set *node* to *node*->*next*

If *node*->*course*->*courseNumber* matches *searchId*,

Print *node*->*course*->*courseNumber* followed by a comma

Print *node*->*course*->*name*

If *node*->*course*->*prerequisites* is not null,

For each *prerequisite*,

Print “, “ and *prerequisite*

## Binary Search Tree

### Create Node

Create class or struct: *Node*

Create Course: course

Create Node\*: *left*

Create Node\*: *right*

Create default constructor

Set *left* to nullptr

Set *right* to nullptr

Create overloaded constructor (takes: *course*)

Set *this*->*course* with *course*

### Create BST Class

Create class: *BST*

Declare private Node\*: *root*

Declare private *insertCourse* method signature (takes: Node\* *node*, Course *course*)

Declare private printCourseList method signature (takes: Node\* *node*)

Declare private printCourses method signature (takes: Node\* *node*)

Declare constructor

Declare public *insertCourse* method signature (takes: Course *course*)

Declare public *printCourseList* method signature

Declare public *printCourses* method signature

Declare *printCourse* method signature (takes: *searchId*)

Define BST constructor

Set *root* to nullptr

### Insert Course

Define public *insertCourse* method (takes: *node*, *course*)

If *root* is null,

Set *root* to new *Node* (call *Node* constructor: pass *course*)

Else,

Call (private) *insertCourse* (pass: *root,* *course*)

Define private *insertCourse* method (takes: *node*, *course*)

If *node*->*course*->*courseNumber* > *course*->*courseNumber*

If *node*->*left* is points to null,

Set *node*->*left* to new *Node* (call *Node* constructor: pass *course*)

RETURN

Else,

(recursive) Call *insertCourse* (pass: *node*->*left*, *course*)

Else,

If *node*->*right* points to null,

Set *node*->*right* to new *Node* (call *Node* constructor: pass *course*)

Else,

(recursive) Call *insertCourse* (pass: *node*->*right*, *course*)

### Print Course List

Define public *printCourseList* method

Call (private) *printCourseList* method (pass: *root*)

Define private *printCourseList* method (takes: *node*)

If *node* does not point to null,

(recursive) Call *printCourseList* (pass: *node*->*left*)

Print *course*->*courseNumber*

(recursive) Call *printCourseList* (pass: *node*->*right*)

### Print Courses with Prerequisites

Define public *printCourses* method

Call (private) *printCourses* method (pass: *root*)

Define private *printCourses* method (takes: *node*)

If *node* does not point to null,

(recursive) Call *printCourses* (pass: *node*->*left*)

Print *course*->*courseNumber* followed by a comma

Print *course*->*name*

If *course*->*prerequisites* is not null,

For each *prerequisite*,

Print “, “ and *prerequisite*

(recursive) Call *printCourses* (pass: *node*->*right*)

### Print One Course with Prerequisites

Define *printCourse* method (takes: *searchId*)

Create Node\*: *current*

Set *current* to *root*

While *current* is not equal to nullptr,

If *current*->*course.courseNumber* equals *searchId*,

Print *current*->*course*.*courseNumber* followed by a comma

Print *current*->*course.name*

If current->*course.prerequisites* is not empty,

For each *prerequisite*,

Print “, “ and *prerequisite*

RETURN

If *current*->*course*.*courseNumber*  > *searchId*,

Set *current* to *current*->*left*

Else,

Set *current* to *current*->*right*

# Evaluation

## Shared Runtime Calculations

All data structures share these runtimes:

* Each non-function line of code is assumed to cost 1 unit of time.
* Number of courses = N
* Assume each course has 1 courseNumber, 1 name, and 2 prerequisites
* Lines are added based on longest solution when an if statement is present
* Where applicable, Big O Runtime Complexity is extracted from our class text by Lysecky and Vahid (2020)

Open, Read, Parse, Check Format (runs 1 time)

6 lines = 6 units

Outer Loop (runs N times):

8 lines = 8 units

Inner Loop (runs 4 times):

2 lines = 2 units

Inner Loop = 8 units

Outer Loop = 8 units + Inner Loop = 16 units

Other Lines = 6 units

Total Time Complexity = 16N + 6

Big O Time Complexity = O(

Validate Prerequisites (contained in Open, Read, Parse, Check Format: runs 1 time)

Note: The call to *insertCourse* is considered 1 unit. Additional runtime is accounted for in each data structure.

Outer Loop (runs N times):

2 lines = 2 units

Inner Loop (runs 2 times):

3 lines = 3 units

Inner Loop = 6 units

Outer Loop = 2 units + Inner Loop = 8 units

Total Time Complexity = 8N

Big O Time Complexity =

Create a Course Object

A course object is created N times

5 lines = 5 units

Total Time Complexity = 5N

Big O Time Complexity =

## Vector

Vectors are extremely easy to implement and use. C++ comes with standard libraries to perform insertion, deletion, swaps, creation, sorting, sizing, and more without requiring custom code or third-party libraries. Vectors store elements by index, numbering each element incrementally from 0, 1, 2, … , N. Accessing an element in a vector when the index is known is extremely efficient and fast, O(1). On the other hand, searching for an element in a vector has a worst-case time complexity of O(N), if the element is found at the end of the vector.

Implementing vectors to read and parse a file is optimal because vectors are extremely efficient when read from and written to linearly. Since built-in libraries are available, vectors will require less debugging of a custom implementation and greater reduction in errors.

Since typical universities offer a limited number of courses, typically a few thousand unique courses within the entire academic catalog, Computer Science is likely to have less than 500 unique courses. Although most modern computers would have no trouble using a vector for this purpose, searching the vector for a unique entry ranks slowest in worst-case time complexity.

### Vector Runtime Analysis

Note: Runtime calculations exclude code shared between all three data structures to more accurately gauge the impact of individual data structures.

Insertion (runs once for each course)

Built-in C++ function: push back =

Sort Courses (runs once when called from each print method)

Partition (called each time quick sort is called)

4 lines = 4 units

Outer Loop (runs up to N/2 times):

6 lines = 6 units

Inner Loop 1 (runs up to N times):

2 lines = 2 units

Inner Loop 2 (runs up to N/2 times):

2 lines = 2 units

Inner Loop 1 = 2N units

Inner Loop 2 = N units

Outer Loop = 3N units

Other Lines = 4 units

Total Time Complexity =

Quicksort (called recursively – up to times)

5 lines = 5 units

Partition = units

Total Time Complexity =

Big O Time Complexity =

Print Course List(runs once when called from menu)

Quicksort =

Loop (N times)

1 line = 1 unit

Total Time Complexity =

Big O Time Complexity =

Print One Course w/ Prerequisites (runs once when called from menu)

Quicksort =

5 lines = 5 units

Outer Loop (N/2 times max)

8 lines = 8 units

Inner loop (runs 2 times before ending outer loop)

1 line = 1 unit

Inner Loop = 2 units

Outer Loop = 4N units

Other lines = 5 units

Total Time Complexity =

Big O Time Complexity =

Total:

Big O Time Complexity =

## Hash Table

Even though hash tables also use vectors to some degree in their implementation, hash table construction relies on a hash method to index and key elements. The hash function can be called to generate a key during search, insertion, and deletion to find a specific element much quicker than traditional vectors. Average time complexity when searching, deleting, and inserting is O(1), but is based on collision handing and hash table sizing. If implemented poorly, a hash table ties with vectors for worst case time complexity.

On the other hand, if a hash table is implemented effectively, time complexity reaches O(1). Hash table implementation, since C++ does not have inbuilt libraries for them, requires careful construction and custom code. If the hash function or the table size results in constant collisions, a verbose adaptation of a vector can result. Most of the courses in the Computer Science curriculum share prefixes, followed by similar three-digit numbers, therefore, the hash function requires careful consideration to prevent frequent collision events, worsening Big O time complexity.

Hash tables are unordered. The order of input ultimately determines element placement, similar to vectors. When printing a hash table in order, the hash table must be sorted first. Fast sort algorithms, like the quick sort algorithm in the above pseudocode can be used to optimize time complexity, but this addition makes the use of hash tables somewhat less attractive for the university’s use case.

While the use of a hash table is a viable solution for its speed benefits and read memory, write memory is often higher than that of a vector due to added support structures. Assuming the university’s application will always read, parse, and fill the hash table each time the program is accessed, the number of courses offered is known, and the hash function separates keys effectively, this data structure is an acceptable option for all but printing a sorted catalog.

### Hash Table Runtime Analysis

Note: Worst case runtime for a hash table is when all entries result in collision (this can be forced by setting the table size to 1)

Class Instantiation (only called once)

2 lines = 2 units

Node Instantiation (called once during each insertion)

5 lines = 5 units

Hash Method (called once during each insertion and print)

3 lines = 3 units

Insertion (called N times)

Create course = 5 units

Create Node = 5 units

Hash = 3 units

5 lines maximum = 5 units

Loop (runs when collision occurs: worst case N times)

7 lines = 7 units

Total Time Complexity =

Big O Time Complexity =

Print Course List / Print Courses w/ Prerequisites (called by user in menu)

Outer Loop (runs N times):

5 lines = 5 units

Hash = 3 units

Create Node = 5 units

Inner Loop (runs on chains: worst case N times) :

6 lines = 6 units

Second Inner Loop (for prerequisites: runs 2 times):

1 line = 1 unit

Total Time Complexity =

Big O Time Complexity =

Print One Course w/ Prerequisites (called by user in menu)

5 lines = 5 units

Hash = 3 units

Create Node = 5 units

Outer Loop (runs on chains: worst case N times):

4 lines = 4 units

Inner Loop (for prerequisites: runs 2 times):

1 line = 1 unit

Total Time Complexity =

Big O Time Complexity =

Total Runtime for All Methods

Total Time Complexity =

Big O Total Time Complexity =

## Binary Search Tree

Binary search trees store data in tiers. A root node, consisting of one element, begins the tree. Subsequent elements are placed in additional nodes consisting of up to two elements per node on either the left or right side below a previous node. Inserting elements into a binary search tree starts at the root and traverses the tree until an open spot is available. Traversal direction is dependent on the value of a key within the element to be inserted and how it compares to the value of the current node’s element key. Insertion of an element into a binary search tree can be longer than hash tables and vectors and is dependent on the number of nodes traversed. Best-case runtime complexity is O(logN) and worst-case O(N) with N being the number of nodes visited.

Elements are naturally ordered by the insertion process, but can be problematic if the source data is already sorted. Inserting ordered data results in a complicated vector, similar to a hash table with constant collisions. Inserting randomized data into a binary search tree provides near-best-case results.

Searching a binary search tree for a specific element is where this data structure really shines. The runtime complexity for a search is based on the number of tiers/levels of the tree “N”. If data is inserted such that all tree levels are filled perfectly, search runtime complexity is O(logN).

Printing sorted (in order) data from binary search tree, does not require data to be reorganized like in hash tables, and instead traverses the tree in a particular manner (left, root, right). Traversals of a binary search tree, whether for insertion or printing, often utilize either while loops or recursion, blurring runtime complexity calculations. On a rudimentary level, O(N) can be assumed when traversing all elements in order due to each node being visited. Realistically, the number of nodes and vertices must be considered, resulting in approximately O(V+E). In this case, “V” is the number of vertices and “E” is the number of nodes.

Assuming the university’s course data is inserted in random order, a binary search tree would be an excellent choice, considering moderately fast insertion and very fast searching. On the other hand, if the data is sorted prior to insertion, a binary search tree would be slower than a vector.

### Binary Search Tree Runtime Analysis

Create Node (runs once when called)

6 lines = 6 units

Instantiate Class (runs once)

12 lines = 12 units

Insertion (runs once per course inserted: N times)

Public (called once per insertion)

Create course = 5 units

Create Node = 6 units

5 lines = 5 units

Private (called recursively traversing tiers: worst case times)

4 lines = 4 units

Total Time Complexity =

Big O Time Complexity =

Print Course List / Courses w/ Prerequisites

Public(called by user in menu)

1 line = 1 unit

Private (called recursively: N times)

3 lines = 3 units

Loop (runs 2 times)

1 line = 1 unit

Total Time Complexity = 5N + 1

Big O Time Complexity =

Print One Course w/ Prerequisites

Public (called by user in menu)

1 line = 1 unit

Private (runs recursively: worst case N times)

5 lines = 5 units

Loop (runs 2 times once)

1 line = 1 unit

Total Time Complexity = 5N + 3

Big O Time Complexity =

Total Runtime for All Methods

Total Time Complexity =

Big O Total Time Complexity =

# Recommendation

It is not unrealistic to use any of these three data structures for relatively small datasets, like university course lists. A vector is easiest to implement, but slowest to execute in all but insertion. If courses were inserted into the vector already sorted, no additional methods to sort would be needed, making full linear traversal as fast as O(N). Searching for an entry can also be sped up numerous search algorithms that can be implemented with custom code or use of the built-in C++ libraries.

Since the number of courses is known, a hash table’s table size could be set to the number of courses. As long as the hash method was designed such that all entries were placed in their own bucket, eliminating all chaining, this data structure would be the fastest to implement overall, but creating a hash function capable of this could be time restrictive.

Binary search trees are easier to implement than hash tables and yield similar results with respect to speed. While it is possible to plan data insertion to result in a perfect binary tree, adding extra functions to complete this would be unnecessary, especially considering a small dataset and the possibility of using a database instead.

Ideally, the university should employ a database (such as SQL) to make queries easy, fast, and maintain data after manipulation for future use. If one of these data structures is to be used instead, as long as data is randomized prior to insertion, a binary search tree is the optimal choice.

Works Cited

Lysecky, R. L., & Vahid, F. V. (2020). *CS 300: Data structures and algorithms* (SNHU CS 300 ed., Vol. 1). Zyante Inc. (zyBooks.com). https://learn.zybooks.com/zybook/SNHUCS300v1