**Project One**

Kaylin Hooper

CS-300

Prof. Sathish Gopalakrishnan

August 10, 2025

**Pseudocode**

The pseudocode from previous milestones have been updated/changed to have less technical language. Each milestone for the separate data structures had pseudocode that covered opening and reading a file, creating course objects, and printing out course information.

**Milestone One: Vector**

**Pseudocode to Open the File, Read the Data, Parse each Line, and Check for File Format Errors.**

Parser(string csv file path)

GET csv file path

IF file path is empty

SET file path to default

OPEN csv file at file path

IF file not found

DISPLAY error message

RETURN 0

ELSE if file found

DEFINE new tempList with Course struct to hold input data in each line from file

WHILE not end of file

READ every line of file

FORMAT each line

IF less than two strings in a line

RETURN error message

ELSE

ADD string 1 to struct courseNumber

ADD string 2 to struct courseName

IF more than two strings in a line

IF third or more

IF value of courseNumber exists

INCREMENT prereqCount

ADD string 3 to prereqList

ELSE

DISPLAY invalid file message

CLOSE file

RETURN tempList

END

**Pseudocode for Course Object**

DEFINE Course structure AS {

string courseNumber;

string courseName;

int prereqCount;

list<string> prereqList;

CONSTRUCTOR Course() {

INITIALIZE Course with default values

}

}

**Pseudocode for Search for Specific Course, Print Course Information and Prerequisites**

searchCourse()

READ every course

IF course is equal to courseNumber

PRINT course information

IF prerequisite exists

PRINT prerequisite course information

END

**Milestone Two: Hash Table**

**Pseudocode to Open the File, Read the Data, Parse each Line, and Check for File Format Errors.**

Parser(string csv file path)

GET csv file path

IF file path is empty

SET file path to default

ENDIF

OPEN csv file at file path

IF file is open

WHILE file has data

DEFINE HashTable structure courses to hold each line

READ every line until end of file

PARSE line into courseNumber, title, prerequisites

IF courseNumber OR title is empty

PRINT “Improper file format.”

CONTINUE to read next line

IF courseNumber AND title are not empty

IF prerequisites exist

INCREMENT prereqCount

ADD prerequisites to prereqList

ELSE

PRINT “Improper file format.”

CLOSE file

IF file does not have data

PRINT error message

ELSE

PRINT “Failed to open file”

END

**Pseudocode for Course Object**

DEFINE class HashTable AS {

DEFINE Course structure AS {

string courseNumber;

string courseName;

int prereqCount;

list<string> prereqList;

unsigned int key;

Course\* next;

CONSTRUCTOR Course() {

INITIALIZE Course with default values

}

HashTable<Course> courses

}

addCourse()

CREATE key by converting number string to integer using atoi

CREATE oldNode node using the key

IF node is equal to nullptr (meaning it is null)

CREATE new node for course, SET equal to the key position

ELSE

WHILE the next node is not equal to nullptr

SET oldNode equal to oldNode->next to find open node

SET next open node equal to new Node

END

**Pseudocode for Search for Specific Course, Print Course Information and Prerequisites**

searchCourse(HashTable<Course> courses, String courseNumber)

READ every course

IF course is equal to courseNumber

PRINT course information

IF prerequisite exists

PRINT prerequisite course information

END

**Milestone Three: Binary Search Tree**

**Pseudocode to Open the File, Read the Data, Parse each Line, and Check for File Format Errors.**

Parser(string csv file path)

GET csv file path

IF file path is empty

SET file path to default

OPEN csv file at file path

IF file not found

DISPLAY error message

RETURN 0

ELSE if file found

CREATE new tempList with Course struct to hold input data in each line from file

WHILE not end of file

READ every line of file

FORMAT each line

IF less than two strings in a line

RETURN error message

ELSE

ADD string 1 to struct courseNumber

ADD string 2 to struct courseName

IF more than two strings in a line

IF third or more

IF value of courseNumber exists

INCREMENT prereqCount

ADD string 3 to prereqList

ELSE

DISPLAY invalid file message

CLOSE file

RETURN tempList

END

**Pseudocode to Create Course Objects and Store in Appropriate Data Structure**

DEFINE Course structure AS {

string courseNumber;

string courseName;

int prereqCount;

list<string> prereqList;

unsigned int key;

Course\* next;

CONSTRUCTOR Course() {

INITIALIZE Course with default values

}

}

DEFINE Class BinarySearchTree

NODE root

void insert(Course course)

DEFINE Node internal tree node structure AS {

Course course;

Node\* left;

Node\* right;

CONSTRUCTOR Node() {

left = nullptr

right = nullptr;

}

INITIALIZE Node(Course aCourse):

Node() {

course = aCourse;

}

}

void BinarySearchTree::Insert(Course course)

IF root is equal to nullptr, meaning the tree is empty

SET root equal to new Node(course), constructing root from the course

ELSE

SET this node to CALL addNode, constructing with root and course

END

void BinarySearchTree::addNode(Node\* node, Course, course)

Starting at root node,

IF course value is less than the node’s

TRAVERSE left

IF left node does not exist

CREATE left node

ELSE a left node does exist

TRAVERSE down

ELSE

IF no right node exists

CREATE right node

ELSE a right node does exist

TRAVERSE down

**Pseudocode to Print out Course Information and Prerequisites**

void BinarySearchTree::InOrder()

CALL inOrder function and pass root

END

void BinarySearchTree::inOrder(Node\* node)

IF node is not equal to nullptr, meaning it exists

TRAVERSE inOrder left side

PRINT Course info at node

TRAVERSE inOrder right side

END

**Pseudocode for a Menu**

void mainMenu()

DISPLAY main menu screen, listing options:  
 OPTION 1: Load the File Data into the data structure

OPTION 2: Print an alphanumerically ordered list of all the courses in the CS department

OPTION 3: Print the course title and prerequisites for any individual course

OPTION 9: Exit the program

INPUT user choice

CASE OF user choice

CASE 1:

LOAD CSV file data into specified data structure

PRINT amount courses read

BREAK;

CASE 2:

CALL sortCourses()

DISPLAY all courses of CS Department in ordered list

BREAK;

CASE 3:

CALL specified data structure SearchCourse()

BREAK;

CASE 9:

PRINT “Good bye.”

EXIT program

END

**Pseudocode for Courses Sorted in Alphanumeric Order**

**Vector Design**

partition()

SET low and high index equal to begin and end

CALCULATE middlePoint

SET pivot as middlePoint element

SET done = FALSE

WHILE done = FALSE

WHILE low index are less than pivot

INCREMENT low index

WHILE pivot is less than high index

DECREMENT high index

IF there are zero or one elements remaining

SET done = TRUE

ELSE

SWAP the low and high bids

MOVE low and high closer

ENDIF

RETURN high;

sortCourse()

quickSort method

IF there are one or zero courses to sort

RETURN;

CALL partition()

SET value of lowEndIndex to value returned by partition(), assigned to “mid”

CALL quickSort recursively to sort low partition

CALL quickSort recursively to sort high partition

END

**Hash Table Design**

sortCourse()

FOR nodes, begin to end iterate

IF the key is not equal to UINT\_MAX

PRINT the course information

SET node equal to next iteration

WHILE node is not equal to nodeptr

PRINT the course information

SET node equal to next node

END

**Binary Search Tree Design**

Already sorted from previous milestone.

sortCourse()

inOrder traversal

inOrder: traverse left, visit root, then traverse right

IF node is not equal to nullptr, meaning while it exists

CALL inOrder function, TRAVERSE left

PRINT Course

CALL inOrder function, TRAVERSE right

END

**Evaluation**

**Runtime Analysis**

Worst-case running time for reading the file and creating course objects.

| **Vector Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Get file path | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| Define course structure | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read every line | 1 | n | n |
| Format each line | 1 | n | n |
| Initialize course structure | 1 | n | n |
| Append course data into structure | 1 | n | n |
| **Total Cost** | | | 5n + 3 |
| **Runtime** | | | O(n) |

| **Hash Table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Get file path | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| Define course structure | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read every line | 1 | n | n |
| Parse each line | 1 | n | n |
| Initialize course structure | 1 | n | n |
| Create key | 1 | 1 | 1 |
| Create oldNode | 1 | 1 | 1 |
| Else node is not equal to nullptr | 1 | n | n |
| Create new node for course | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 6 |
| **Runtime** | | | O(n) |

| **Binary Search Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Get file path | 1 | 1 | n |
| Open file | 1 | 1 | n |
| Define course structure | 1 | 1 | 1 |
| While not end of file | 1 | n | n |
| Read every line | 1 | n | n |
| Format each line | 1 | n | n |
| Initialize course structure | 1 | n | n |
| If root is not equal to nullptr | 1 | 1 | 1 |
| If course value is greater than current node | 1 | logn | logn |
| If right node does not exist | 1 | 1 | 1 |
| Create right node | 1 | 1 | 1 |
| **Total Cost** | | | 6n + logn + 4 |
| **Runtime** | | | O(logn) |

**Advantages and Disadvantages of each Structure**

Starting with the vector, the initial time it takes to load the data into an unsorted vector and appending it is fast, but does not take into consideration how much longer it takes to sort that data afterwards. Insertion/deletion is only fast on the end, while the middle would require the elements to all be shifted, leading to further a linear (O(N)) runtime complexity.

Hash tables are not reliant on the order of the data, as the arrays of buckets store key-value pairs. If it were able to prevent collisions, the hash table could offer an average constant runtime complexity (O(1)). But that is not realistic, and the worst-case runtime could degrade the hash table to O(N). The performance relies on how collisions will be resolved with the hash functions.

The binary tree is able to maintain its sorted order, which allows traversals to be rather quick and efficient. It is typically running at O(logn) which allows for predictable runtime complexity. However, the performance is only reliable/predictable if the sorted data is not skewed/unbalanced. An unbalanced BST will not guarantee a better performance, and could actually lead to O(N).

**Recommendation**

The recommendation was stuck between the Hash Table or the BST. Ultimately, it comes down to how important it is to maintain a sorted order, as well as a consistent time. The data itself does not seem to be changed too often, and being sorted (when printing all) also shouldn’t be a frequent task. Since the order of the courses is not crucial to the data, a hash table may be the better option, as it handles searching large datasets very well. It allows average-case constant runtime complexity, as long as the table is created with the goal of minimizing collisions and handling them when they occur.