# CS 300 Pseudocode Document

## Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

Ryan Hoskins

Feb. 18th, 2024

CS-300

// Vector pseudocode.

**Menu**

mainMenu():

Create a Vector structure to hold courses (courses)

Initialize “choice” variable to zero

While the user’s input does not equal 9,

Call displayMenu function (displayMenu())

Take user input

Set case 1   
 Call loadFile function to open CSV file, store into ‘courses variable

Set case 2

Call “print course list” function

Set case 3

Call searchCourse function

Set case 9

Close program

**Display Menu**

displayMenu()

Print “Main Menu”  
 Print “1. Load Courses”  
 Print “2. Print Courses List”  
 Print “3. Print Course Info”  
 Print “9. Exit program”

**Open and Load file**

Vector<Courses> loadFile(string fileName)

Create Vector structure to hold courses (Courses)

Initialize CSV parser to open at given path, store into ‘file’ variable

Try to open file,

If unable to open file,

Catch error, “Unable to open file”

For each row in the file,

Split the row data using commas as delimiter and store into courseData variable

Check if each row is valid (validateCourseData (courseData, courses))

Create object for course using courseData“ course = createCourseObject(courseData)

Push course to end

Catch errors if present

Return courses

**Valid Row**

validateCourseData( array courseData, vector<courses>)

if the length of courseData is less than two,

Raise error, “Error: Not enough parameters for course, please ensure all data is entered”

Return false

If courseData[0] is empty,

Raise error, “Error: Course Number is missing”

Return false

If courseData[1] is empty,

Raise error, “Error: Course Name is Missing”

Return false

For each prerequisite in courseData[2:],

If validCourse returns false,

Raise error, “Error: Prerequisite not found as valid course”

Return false

Return true

**Check Prerequisite Exists**

validCourse (string courseNumber,vector courses)

For each course in courses vector,

If course.courseNumber == courseNumber,

Return true

Return false

**Create Course Object**

createCourseObject (courseData)

Create new course object

Set course.courseNumber to courseData[0]

Set course.courseName to courseData[1]

Set course.prerequisites to courseData[2:]

Return course

**Searching and Printing A Specific Course**

menuFindACourse()

Take user input for courseNumber, set to courseNumber

foundCourse = findCourse(courses, courseNumber)

If course was found,

printCourseInfo(foundCourse)

else,

Display, “Course Not Found”

**Search for Course**

findCourse(vector<courses>, courseNumber)

For each course in ‘courses’ vector,

If course.courseNumber == courseNumber,

Return course

Return null

**Print Course Info**

printCourseInfo(course)

Display course number using passed course

Display course name used passed course

For prerequisites in passed course,

Display prerequisites

**Sort Courses**

sortCourses(vector<Course> &courses)

Implement selection sorting algorithm

**Print Sorted Course List**

menuPrintCourseList()

Call sortCourses function and set to sortedCourses variable

For each course in sortedCourses

Display course

sortedCourses = sortCourses (courses)

For course in sortedCourses:

Print course

// Hashtable pseudocode

**Course Structure**

Struct Course  
 define a string variable for courseNumber

Define a string variable for courseName

**HashTable class**

Class Courses

Define Node structure

Course course

Define key

Define node pointer (\*next)

Define default constructor (Node())

Set ‘key’ to UINT\_MAX

Set ‘next’ to null

Initialize constructor with course (Node(Course aCourse) : Node())

Set variable ‘course’ to ‘aCourse’

Initialize with course and key (Node(Course aCourse, unsigned\_int aKey) : Node(aCourse)

Set ‘key’ to ‘aKey’

Define Vector<Node> nodes

Define table size

Define hash with key

Define public

**Default Constructor**

Courses::Courses()

Resize table size for nodes

**Hashtable Destructor**

Courses::~Courses()

Erase beginning of nodes

**Calculate Hash Value**

Unsigned int Courses::hash(int key)

Initialized hash value

For each character in the key

Set hash to (hashvalue \* 31 + ch) % tableSize

Return hashvalue

**Menu**

mainMenu():

Create a Vector structure to hold courses (courses)

Initialize “choice” variable to zero

While the user’s input does not equal 9,

Call displayMenu function (displayMenu())

Take user input

Set case 1   
 Call loadFile function to open CSV file, store into ‘courses variable

Set case 2

Call “print course list” function

Set case 3

Call searchCourse function

Set case 9

Close program

**Display Menu**

displayMenu()

Print “Main Menu”  
 Print “1. Load Courses”  
 Print “2. Print Courses List”  
 Print “3. Print Course Info”  
 Print “9. Exit program”

**Open and Load file**

Void loadCourses(string fileName, HashTable\* hashTable)

Create empty hashable structure to hold courses (Courses)

Initialize CSV parser to open at given path, store into ‘file’ variable

Try to open file,

If unable to open file,

Catch error, “Unable to open file”

For each row in the file,

Split the row data using commas as delimiter and store into courseData variable

Check if each row is valid (validateCourseData (courseData, courses))

Create object for course using courseData“ course = createCourseObject(courseData)

Call insertCourse function to insert into hash table

Catch errors if present

**Valid Row**

validateCourseData( array courseData, Hashtable<courses>)

if the length of courseData is less than two,

Raise error, “Error: Not enough parameters for course, please ensure all data is entered”

Return false

If courseData[0] is empty,

Raise error, “Error: Course Number is missing”

Return false

If courseData[1] is empty,

Raise error, “Error: Course Name is Missing”

Return false

For each prerequisite in courseData[2:],

If validCourse returns false,

Raise error, “Error: Prerequisite not found as valid course”

Return false

Return true

**Check Prerequisite Exists**

validCourse (string courseNumber,HashTable<Course> courses)

Calculate hash value using courseNumber

Set current node to address of node located at hash value

While current node is not null,

If current node's key matches course number,

Return true

Continue to the next node

Return false

**Create Course Object**

createCourseObject (courseData)

Create new course object

Set course.courseNumber to courseData[0]

Set course.courseName to courseData[1]

Set course.prerequisites to courseData[2:]

Return course

**Insert Course**

void HashTable::Insert(Course course)

Calculate hash value for courseID attribute

Try and retrieve node as key's position

If there aren't any nodes at the key's position,

Create new node with course and key

Insert into ‘nodes’ array at the calculated index

Else,

If the node is empty,

Update attributes of existing node to match courseID and course, set pointer to null

Else,

Traverse linked nodes until the end,

Create new node with course and courseID (key)

**Searching and Printing A Specific Course**

menuFindACourse()

Take user input for courseId, set to courseId

foundCourse = Search(courseId)

If course was found,

printCourseInfo(foundCourse)

else,

Display, “Course Not Found”

**Search for Course**

Course HashTable::Search(string courseId)

Calculate hash for courseId attribute

Set current node

If the current node is empty or key unused

Return null course

If the first key matches and is not empty,

Return current course

While current is not null,

If the key is not empty and the courseId matches,

Return the current course object

Iterate through nodes

Return empty course object if not found

**Print Course Info**

printCourseInfo(course)

Display course number using passed course

Display course name used passed course

For prerequisites in passed course,

Display prerequisites

**Print Sorted Course List**

menuPrintCourseList()

Create empty array set to ‘courseList’

For each node in the hash table,

Set current node

While the current node is not null,

Append courseList with course

Move to next node

Sort courses

For each course in courseList,

Print course

// Tree pseudocode

**Binary Search Tree Class**

Class BinarySearchTree

Declare private members

Declare public members

**Constructor**

BinarySearchTree::BinarySearchTree

Set root to null pointer

**Default Destructor**

BinarySearchTree::~BinarySearchTree()

Call destroyTree with ‘root’ passed

**Destroy Tree**

void BinarySearchTree::destroyTree(Node\* node)

If the node is equal to null,

Call destroyTree with node -> left passed

Call destroyTree with node -> right passed.

Delete node

**Menu**

mainMenu():

Create a BinarySearchTree structure to hold courses (BinarySearchTree\* bst)

Set bst structure to new BinarySearchTree (bst = new BinarySearchTree())

Initialize “choice” variable to zero

While the user’s input does not equal 9,

Call displayMenu function (displayMenu())

Take user input

Set case 1   
 Call loadFile function to open CSV file, store into ‘courses variable

Set case 2

Access InOrder() function with ‘bst’ (bst->InOrder()

Call “print course list” function

Set case 3

Call searchCourse function

Set case 9

Close program

**Display Menu**

displayMenu()

Print “Main Menu”  
 Print “1. Load Courses”  
 Print “2. Print Courses List”  
 Print “3. Print Course Info”  
 Print “9. Exit program”

**Open and Load file**

Void loadFile (BinarySearchTree\* bst)

Initialize CSV parser to open at given path, store into ‘file’ variable

Try to open file,

If unable to open file,

Catch error, “Unable to open file”

For each row in the file,

Split the row data using commas as delimiter and store into courseData variable

***Check if each row is valid (validateCourseData (courseData, courses))***

Create object for course using courseData course = createCourseObject(courseData)

Call insertCourse function to insert into hash table

Catch errors if present

**Valid Row**

validateCourseData( array courseData)

If the length of courseData is less than two,

Raise error, “Error: Not enough parameters for course, please ensure all data is entered”

Return false

If courseData[0] is empty,

Raise error, “Error: Course Number is missing”

Return false

If courseData[1] is empty,

Raise error, “Error: Course Name is Missing”

Return false

For each prerequisite in courseData[2:],

If validCourse returns false,

Raise error, “Error: Prerequisite not found as valid course”

Return false

Return true

**Check Prerequisite Exists**

validCourse(courseNumber)

Start from the root node of the BST

While current node is not null,

If course number of current node matches the given courseNumber,

Return true (course exists)

If the given courseNumber is less than the course number of the current node,

Move to the left subtree

Otherwise, if courseNumber is greater than the course number of the current node,

Move to the right subtree

If the courseNumber is not found after traversing the entire BST,

Return false

**Create Course Object**

createCourseObject (courseData)

Create new course object

Set course.courseNumber to courseData[0]

Set course.courseName to courseData[1]

Set course.prerequisites to courseData[2:]

Return course

**Insert Course**

void BinarySearchTree::Insert(Course course)

If the tree is empty,

Create a new node (root = new Node(course)

Otherwise, if tree is not empty

Call addNode function and pass root and course

**Add Node**

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

If node is not null and existing courseId is larger than inserted,

If the left subtree is null,

Insert node with course (node->left = new Node(course))

Return

Else,

Call addNode to recurse down left subtree

Else, if existing courseId is smaller than inserted,

If the right subtree is null,

Insert node with course (node->right = new Node(course))

Return

Otherwise,

Call addNode to recurse down right subtree

**Searching and Printing A Specific Course**

menuFindACourse()

Take user input for courseId, set to courseId

foundCourse = Search(courseId)

If course was found,

printCourseInfo(foundCourse)

else,   
`

Display, “Course Not Found”

**Searching For Course**

Course BinarySearchTree::Search(string courseNumber)

Start from the root Node (Node\* current = root)

While the current isn’t null,

If courseNumber of current node matches entered courseNumber,

Return the course

Else, if entered courseNumber is less than current node’s courseNumber,

Move to the left (current = current->left)

Otherwise, if entered courseNumber is greater than current node’s courseNumber,

Move to the right (current = current->right)

Create empty course (Course course)

Return course

**Print Course Info**

void displayCourse(Course course)

Display course number using passed course

Display course name used passed course

For prerequisites in passed course,

Display prerequisites

**InOrder Function**

void BinarySearchTree::InOrder()

Call inOrder function with passed root (this->inOrder(root))

**inOrder Function**

void BinarySearchTree::inOrder(Node\* node)

If the node is empty,

Call inOrder function to traverse left subtree (inOrder(node->left))

Display information for each node

Call inOrder function to traverse right subtree (inOrder(node->right))

**Runtime Analysis**

**Vector – Open and Load File**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| Create Vector Structure | 1 | 1 | 1 |
| Initialize CSV Parser | 1 | 1 | 1 |
| Try to open file | 1 | 1 | 1 |
| For each row in file | 1 | n | n |
| Split row, store into courseData variable | 1 | n | n |
| validateCourseData | 1 | n | 3n+19 |
| Create course object | 5 | n | 5n |
| Push course to end | 1 | n | n |
|  |  | **Total Cost** | 11n + 22 |
|  |  | **Runtime** | O(n) |

**Hash Table - Open and Load File**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| Create hash table | 1 | 1 | 1 |
| Initialize CSV parser to open file | 1 | 1 | 1 |
| Try to open file | 1 | 1 | 1 |
| For each row in the file | 1 | n | n |
| Split data, store into courseData | 1 | n | n |
| validateCourseData | 1 | n | 3n |
| Create object for course | 5 | n | 5n |
| Call insertCourse | 11 | n | 11n |
| Catch errors | 1 | 1 | 1 |
|  |  | Total Cost | 19n+4 |
|  |  | Runtime | O(n) |

**Validate Course Data**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| If length of courseData is < 2 | **1** | **1** | **1** |
| Raise error | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| If courseData[0] is empty | 1 | 1 | 1 |
| Raise error | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| If courseData[1] is empty | 1 | 1 | 1 |
| Raise error | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| For each prerequisite | 1 | n | n |
| If validCourse is false | 7 | 2n | 2n+7 |
| Raise error | 1 | 1 | 1 |
| Return false | 1 | 1 | 1 |
| Return true | 1 | 1 | 1 |
|  |  | **Total Cost** | 3n+19 |

**Binary Search Tree – Open and Load File**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| Initialize CSV parser to open file | 1 | 1 | 1 |
| Try to open file | 1 | 1 | 1 |
| For each row in the file | 1 | n | n |
| Split data, store into courseData | 1 | n | n |
| validateCourseData | 1 | n | n |
| Create object for course | 5 | n | 5n |
| Call insertCourse | 1 | n | n |
| Catch errors | 1 | 1 | 1 |
|  |  | Total Cost | 9n+4 |
|  |  | Runtime | O(n) |

**Binary Search Tree – Insert Course**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| If the tree is empty | 1 | 1 | 1 |
| Create new node | 1 | 1 | 1 |
| Else, if tree is not empty | 1 | 1 | 1 |
| Call addNode function and pass root and course | 1 | 1 | 1 |
|  |  | **Total Cost** | 5 |
|  |  | **Runtime** | 5 |

**Binary Search Tree – Add Node**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Times Executed** | **Total Cost** |
| If node is not null and existing courseNumber is larger | 1 | 1 | 1 |
| If left subtree is null | 1 | n | n |
| Insert node with course | 1 | 1 | 1 |
| Else | 1 | n | n |
| Call addNode to recurse left subtree | 1 | n | n |
| Else if existing courseNumber is smaller | 1 | n | n |
| If the right subtree is null | 1 | n | n |
| Insert node with course into right | 1 | 1 | 1 |
| Otherwise | 1 | 1 | 1 |
| Call addNode to recurse down right subtree | 1 | n | n |
|  |  | **Total Cost** | 6n + 4 |
|  |  | **Runtime** | O(n) |

**Review**

Reviewing each of the three data structures reveals advantages and disadvantages for each. Where a Vector data structure excels, a Binary Search Tree may struggle. To start, a Vector data structure is fairly straight forward to implement. Due to the data being stored sequentially yet retaining flexibility, they provide a great option when deciding on a data structure. In addition to this, they are also dynamically sized, meaning there is not a set limit to the size of the structure. While this may not too necessary for the requirements of this assignment, it could be a factor when determining how often courses are updates, added, or removed from the school’s catalogue. With that said, Vector data structures to encounter some disadvantages as well. Regarding time complexity, the average runtime for standard procedures such as searching, insertion, or deletion, vector experience O(n) time complexity, making it fairly inefficient. Lastly, there aren’t many efficient methods of sorting elements within a vector data structure.

Hash Tables offer many appealing benefits when deciding on a data structure. Due to the near constant time-averages it can sustain, inserting, searching, and deleting procedures can be done on average with a time complexity of O(1), making this a very solid choice for this assignment. Hash Tables also provided the added benefit of handling larger datasets well and even more so when the maximum number of entries is known in advance – something that would be very fitting for the use of this application. Since a school’s course catalogue is known in advance and doesn’t experience a lot of frequent changing, a hash table can be a great option for a data structure used in this application. However, some of the advantages of a hash table can be affected, depending on the configuration or circumstances of its use. For example, collisions create a potential disadvantage, as they can happen with hash tables and require some sort of handling to ensure they can be appropriately resolved. The methods in which collisions can be handled, such as chaining, may result in decreased performance or increased memory usage. Additionally, hash tables do not have an inherent way to be sorted. While this can be addressed through additional steps, such as create an array or vector of Key values and sorting that accordingly, it does result in decreased performance and more maintenance, which could be a notable disadvantage that affects the decision.

Lastly, Binary Search Trees offer an interesting option for addressing the needs of a data structure within this assignment. One of the primary advantages of a binary search tree is the inherent ordering used in it. Since data is inserted through the creation of nodes and each node’s location is determined by some sort of comparison to a root node, elements can be ordered automatically. This provides a significant advantage for a use that prioritizes sorting content in a particular way. Additionally, Binary Search Trees typically have a time complexity of O(log(n)) for searching, inserting, and deleting. However, Binary Search Trees also introduce disadvantages that should be taken into consideration. Binary Search Trees do not automatically balance, meaning the tree could be severely uneven on one side or the other. This introduces complications for searching, deleting, and inserting operations that result in a time complexity of O(n). This could also result in potential crashing for larger datasets if a function recurses far too deeply due to a severely unbalanced tree data structure.

Considering the advantages and disadvantages of each data structure, I would recommend the application use a Hash Table. Hash Tables, as mentioned earlier, provide near constant time complexity for most operations, including searching, inserting, and deleting. They provide an incredible option to efficiently access courses inserted into the hash table, and although the method of sorting can be a bit of a round-about way to accommodate, the advantages for them outweigh the disadvantages.