struct Course {

String courseNumber

String courseTitle

Vector<String> prerequisites

}

**// Vector**

Vector<Course> courses

void loadCourses(String filename) {

open the file with filename

if the file cannot be opened

print “Error when trying to open {file}”

return

for each line in the file

split the line by comma into tokens

if tokens count is less than 2

print “Error: Invalid format {line}”

continue

create a new course named Course

assign course’s courseNumber to the first token

assign course’s courseTitle to the second token

for each prerequisite

add the prerequisite to the course’s prerequisites

add the course to courses

close the file

}

bool validateCourses(Vector<Course> courses) {

assign isValid to true

for all courses

for each course prerequisite

if prerequisite is not in courses

print “Error: The prerequisite course was not found”

set isValid to false

return isValid

}

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

for all courses

if the course.courseNumber is the same as courseNumber

print “{courseNumber} {courseTitle}”

for each prerequisite of the course

print “{prerequisite}”

}

**// Vector Runtime Analysis**

void loadCourses(String filename)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| open the file with filename | 1 | 1 | 1 |
| for each line in the file | 1 | n | n |
| split the line by comma into tokens | 1 | n | n |
| if tokens count is less than 2 | 1 | n | n |
| print “Error: Invalid format {line}” | 1 | n | n |
| continue | 1 | n | n |
| create a new course named Course | 1 | n | n |
| assign course’s courseNumber to the first token | 1 | n | n |
| assign course’s courseTitle to the second token | 1 | n | n |
| for each prerequisite | 1 | m \* n | m \* n |
| add the prerequisite to the course’s prerequisites | 1 | m \* n | m \* n |
| add the course to courses | 1 | n | n |
| close the file | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 2m\*n + 2 |
| **Runtime** | | | O(n\*m) |

bool validateCourses(Vector<Course> courses)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| assign isValid to true | 1 | 1 | 1 |
| for all courses | 1 | n | n |
| for each course prerequisite | 1 | m\*n | m\*n |
| if prerequisite is not in courses | O(n) | m\*n | m\*n2 |
| Print “Error: The prerequisite course was not found” | 1 | m\*n | m\*n |
| set isValid to false | 1 | m\*n | m\*n |
| return isValid | 1 | 1 | 1 |
| **Total Cost** | | | n + 3m\*n + m\*n2 + 2 |
| **Runtime** | | | O(n2\*m) |

void searchCourse(Vector<Course> courses, String courseNumber)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| for all courses | 1 | n | n |
| If the course.courseNumber is the same as courseNumber | 1 | n | n |
| print “{courseNumber} {courseTitle}” | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | m | m |
| print “{prerequisite}” | 1 | m | m |
| **Total Cost** | | | 2n + 2m + 1 |
| **Runtime** | | | O(n + m) |

**// Hash Table**

struct Node {

String key

Course value

Node next

Node(key, value) {

assign self.key to the key

assign self.value to the value

}

}

struct HashTable {

Vector<Node> buckets

unsigned int tableSize

HashTable() {

resize buckets to tableSize

}

HashTable(unsigned int size) {

assign tableSize to size

resize buckets to size

}

unsigned int hash(int key) {

return key % tableSize

}

void insert(Course value) {

convert value.courseNumber to an int

hash the key using value.courseNumber

get the head node from the buckets vector using the key

if the head node is unused

set the head node to Node(key, value)

else

iterate to the end of the chain

set the next node to a new Node(key, value)

}

Course get(String courseNumber) {

convert courseNumber to an int

hash the key using courseNumber

assign current node to the head node from the buckets vector using the key

while current node is not null

if current.key is equal to key

return current.value

move to the next node

return Course()

}

bool containsKey(string courseNumber) {

convert courseNumber to an int

hash the key using courseNumber

assign current node to the head node from the buckets vector using the key

while current node is not null and key is not null

if current.key is equal to key

return true

move to the next node

return false

}

}

HashTable courseTable

void loadCourses(String filename) {

open the file with filename

if the file cannot be opened

print “Error when trying to open {file}”

return

for each line in the file

split the line by comma into tokens

if tokens count is less than 2

print “Error: Invalid format {line}”

continue

create a new Course named course

assign course’s courseNumber to the first token

assign course’s courseTitle to the second token

for each prerequisite

add the prerequisite to the course’s prerequisites

insert the course into the courseTable using insert(course)

close the file

}

bool validateCourses(HashTable courseTable) {

assign isValid to true

for each bucket in courseTable.buckets

for each node in the bucket

for each prerequisite in node.value.prerequisites

if courseTable.containsKey(prerequisite) is false

print “Error: The prerequisite was not found”

set isValid to false

return isValid

}

void searchCourse(HashTable courseTable, String courseNumber) {

if courseTable.containsKey(courseNumber) is false

print “The course number {courseNumber} was not found.”

return

Course course = courseTable.get(courseNumber)

print “{course.courseNumber} {course.courseTitle}”

for each prerequisite

print “{prerequisite}”

}

**// HashTable Runtime Analysis**

unsigned int hash(int key)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| return key % tableSize | 1 | 1 | 1 |
| **Total Cost** | | | 1 |
| **Runtime** | | | O(1) |

void insert(Course value)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| convert value.courseNumber to an int | 1 | 1 | 1 |
| hash the key using value.courseNumber | O(1) | 1 | 1 |
| get the head node from the buckets vector using the key | 1 | 1 | 1 |
| if the head node is unused | 1 | 1 | 1 |
| set the head node to Node(key, value) | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| iterate to the end of the chain | 1 | n | n |
| set the next node to a new Node(key, value) | 1 | 1 | 1 |
| Total Cost | | | n + 7 |
| Runtime | | | O(n) |

void loadCourses(String filename)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| open the file with filename | 1 | 1 | 1 |
| for each line in the file | 1 | n | n |
| split the line by comma into tokens | 1 | n | n |
| if tokens count is less than 2 | 1 | n | n |
| print “Error: Invalid format {line}” | 1 | n | n |
| continue | 1 | n | n |
| create a new Course named course | 1 | n | n |
| assign course’s courseNumber to the first token | 1 | n | n |
| assign course’s courseTitle to the second token | 1 | n | n |
| for each prerequisite | 1 | m\*n | m\*n |
| add the prerequisite to the course’s prerequisites | 1 | m\*n | m\*n |
| insert the course into the courseTable using insert(course) | O(n) | n | n2 |
| close the file | 1 | 1 | 1 |
| **Total Cost** | | | 8n + 2m\*n + n2 + 2 |
| **Runtime** | | | O(n2) |

bool containsKey(string courseNumber)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| convert courseNumber to an int | 1 | 1 | 1 |
| hash the key using courseNumber | O(1) | 1 | 1 |
| assign current node to the head node from the buckets vector using the key | 1 | 1 | 1 |
| while current node is not null and key is not null | 1 | n | n |
| if current.key is equal to key | 1 | n | n |
| move to the next node | 1 | n | n |
| return false | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 4 |
| **Runtime** | | | O(n) |

bool validateCourses(HashTable courseTable)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| assign isValid to true | 1 | 1 | 1 |
| for each bucket in courseTable buckets | 1 | n | n |
| for each node in the bucket | 1 | m\*n | m\*n |
| for each prerequisite in node.value.prerequisites | 1 | l\*m\*n | l\*m\*n |
| if courseTable.containsKey(prerequisite) is false | O(n) | l\*m\*n | l\*m\*n2 |
| print “Error: The prerequisite was not found” | 1 | l\*m\*n | l\*m\*n |
| set isValid to false | 1 | l\*m\*n | l\*m\*n |
| return isValid | 1 | 1 | 1 |
| **Total Cost** | | | n + m\*n + 3l\*m\*n + l\*m\*n2 + 1 |
| **Runtime** | | | O(l\*m\*n2) |

Course get(String courseNumber)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| convert courseNumber to an int | 1 | 1 | 1 |
| hash the key using courseNumber | O(1) | 1 | 1 |
| assign current node to the head node from the buckets vector using the key | 1 | 1 | 1 |
| while current node is not null | 1 | n | n |
| if current.key is equal to key | 1 | n | n |
| move to the next node | 1 | n | n |
| return false | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 4 |
| **Runtime** | | | O(n) |

void searchCourse(HashTable courseTable, String courseNumber)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| if courseTable.containsKey(courseNumber) is false | O(n) | 1 | n |
| Course course = courseTable.get(courseNumber) | O(n) | 1 | n |
| print “{course.courseNumber} {course.courseTitle}” | 1 | 1 | 1 |
| for each prerequisite | 1 | m | m |
| print “{prerequisite}” | 1 | m | m |
| **Total Cost** | | | 2n + 2m + 1 |
| **Runtime** | | | O(n + m) |

**// Binary Search Tree**

struct Node {

Course course

Node\* left

Node\* right

Node(Course aCourse) {

course = aCourse

}

}

class Tree {

private:

Node\* root

void addNode(Node\* node, Course course)

void preOrder(Node\* node)

void inOrder(Node\* node)

void postOrder(Node\* node)

public:

Tree()

void Insert(Course course)

void preOrder()

void inOrder()

void postOrder()

Course Search(string courseNumber)

}

Tree() {

root = nullptr

}

void addNode(Node\* node, Course course) {

if course.courseNumber is less than the node’s course.courseNumber

if the node’s left pointer is equal to nullptr

node->left = new Node(course)

else

addNode(node->left, course)

else

if the node’s right pointer is equal to nullptr

node->right = new Node(course)

else

addNode(node->right, course)

}

void preOrder(Node\* node) {

print “{courseNumber} {courseTitle}”

for each prerequisite of the course

print “{prerequisite}”

preOrder(node->left)

preOrder(node->right)

}

void inOrder(Node\* node) {

inOrder(node->left)

print “{courseNumber} {courseTitle}”

for each prerequisite of the course

print “{prerequisite}”

inOrder(node->right)

}

void postOrder(Node\* node) {

postOrder(node->left)

postOrder(node->right)

print “{courseNumber} {courseTitle}”

for each prerequisite of the course

print “{prerequisite}”

}

void Insert(Course course) {

if root is equal to nullptr

root = new Node(course)

else

addNode(root, course)

}

void preOrder() {

preOrder(root)

}

void inOrder() {

inOrder(root)

}

void postOrder() {

postOrder(root)

}

Course Search(string courseNumber) {

Node\* curr\_node = root

while curr\_node is not equal to nullptr

if curr\_node->course.courseNumber is equal to courseNumber

return curr\_node->course

if courseNumber is less than curr\_node->course.courseNumber

curr\_node = curr\_node->left

else

curr\_node = curr\_node->right

return Course()

}

Tree<Course> courses

void loadCourses(String filename) {

try to open file with filename

if the file could not be opened

print “Error while trying to open {filename}”

return

for each line in the file

split the line by comma into tokens

if tokens count is less than 2

print “Error: Invalid format {line}”

continue

create a new Course named course

assign course’s courseNumber to the first token

assign course’s courseTitle to the second token

for each token > 2

append the prerequisite to course’s prerequisites

insert the course into the Tree using Insert(course)

close the file

}

bool verifyCourses(Node\* node, Tree<Course> courses, bool& verified) {

if node is equal to nullptr

return

verifyCourses(node->left, courses, verified)

for each prerequisite of node->course

Course course = courses.Search(prerequisite)

if course.courseNumber is empty

verified = false

print “{prerequisite} not found for {node->course.courseNumber}”

verifyCourses(node->right, courses, verified)

}

bool verifyCourses(Tree<Course> courses) {

bool verified = true

verifyCourses(courses.root, courses, verified)

return verified

}

void searchCourse(Tree<Course> courses, String courseNumber) {

Course course = courses.Search(courseNumber)

if course.courseNumber is empty

print “The course number {courseNumber} was not found.”

else

print “{course.courseNumber} {course.courseTitle}”

for each prerequisite of the course

print “{prerequisite}”

}

**// BST Runtime Analysis**

void addNode(Node\* node, Course course)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| if course.courseNumber is less than the node’s course.courseNumber | 1 | n | n |
| if the node’s (left or right) pointer is equal to nullptr | 1 | n | n |
| (node->left or node->right) = new Node(course) | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

void Insert(Course course)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| if root is equal to nullptr | 1 | 1 | 1 |
| addNode(course) | O(n) | 1 | n |
| **Total Cost** | | | n + 1 |
| **Runtime** | | | O(n) |

void loadCourses(String filename)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| try to open file with filename | 1 | 1 | 1 |
| for each line in the file | 1 | n | n |
| split the line by comma into tokens | 1 | n | n |
| create a new Course named course | 1 | n | n |
| assign course’s courseNumber to the first token | 1 | n | n |
| assign course’s courseTitle to the second token | 1 | n | n |
| for each token > 2 | 1 | m\*n | m\*n |
| append the prerequisite to course’s prerequisites | 1 | m\*n | m\*n |
| insert the course into the Tree using Insert(course) | O(n) | n | n2 |
| close the file | 1 | 1 | 1 |
| **Total Cost** | | | 5n + 2m\*n + n2 + 2 |
| **Runtime** | | | O(n2) |

Course Search(string courseNumber)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Node\* curr\_node = root | 1 | 1 | 1 |
| while curr\_node is not equal to nullptr | 1 | n | n |
| if curr\_node->course.courseNumber is equal to courseNumber | 1 | n | n |
| if courseNumber is less than curr\_node->course.courseNumber | 1 | n | n |
| curr\_node = curr\_node->left | 1 | n | n |
| return Course() | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 2 |
| **Runtime** | | | O(n) |

void searchCourse(Tree<Course> courses, String courseNumber)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Course course = courses.Search(courseNumber) | O(n) | 1 | n |
| if course.courseNumber is empty | 1 | 1 | 1 |
| print “{course.courseNumber} {course.courseTitle}” | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | m | m |
| print “{prerequisite}” | 1 | m | m |
| **Total Cost** | | | n + 2m + 2 |
| **Runtime** | | | O(n + m) |

**// Menu**

void menu() {

(Vector<Course> or HashTable or Tree<Course>) dataStruct

isLoaded = false

while true

print “1. Load course data from file”

print “2. Print all courses (alphanumerically)”

print “3. Print a course with prerequisites”

print “9. Exit”

get user input

if input is equal to 1

loadCourses(String filename)

if input is equal to 2

if isLoaded is true

printAlphanumerically(dataStruct)

else

print “Please load the course data”

if input is equal to 3

if isLoaded is true

get courseNumber from the user

searchCourse(dataStruct, courseNumber)

else

print “Please load the course data”

if input is equal to 9

return

}

**// Alphanumeric Print**

void printAlphanumerically(Vector<Course> courses) {

sort courses

for each course in courses

printCourse(course)

}

void printAlphanumerically(HashTable courseTable) {

create an empty list called courseList

for each bucket in courseTable.buckets

for each node in the bucket

append node.value to courseList

sort courseList

for each course in courseList

printCourse(course)

}

void printAlphanumerically(Tree<Course> courses) {

courses.inOrder()

}

void printCourse(Course course) {

print “{course.courseNumber} {course.courseTitle}”

for each prerequisite

print “{prerequisite}”

}

**// Advantages and Disadvantages**

A vector is a very simple data structure in terms of implementation. Simply declaring the vector and then appending additional values to the end makes the vector very user friendly. However, when traversing a vector, every item up to the item searched (or until the end of the vector, if the item does not exist within the vector) is visited. This greatly increases the overall runtime, with worst-case scenarios taking O(n + m) time, where n is the number of courses and m is the number of prerequisites, as can be seen in the runtime analyses. This worst-case runtime further increases when considering prerequisites, such as when validating the courses, resulting in time complexities of O(n2 + m). These inefficiencies become increasingly problematic as the vector size increases. As such, the vector is best used for small to medium data sets.

The hash table provides excellent average look up times, typically O(1) in best case scenarios and O(n) in the worst. However, as can be seen in validateCourses(), having to take into account the number of buckets, chained nodes, and collision resolution, worst case scenarios can become very complex, along the lines of O(l\*m\*n2). It must also be noted that directly printing the hash table in alphanumerical order is a bit more complex, as can be seen in the pseudocode. First each node must be extracted into a sortable data structure and then processed.

A binary search tree is a very complex data structure to implement due to pointer management and recursive calls. However, a binary search tree provides one major benefit over the other two data structures: natural alphanumeric sorting. Best case scenarios when working with binary search trees, such as searching and insertion, typically fall in O(log n) time, with worst-case scenarios averaging O(n) time if the tree becomes unbalanced, with nodes falling entirely along the left or right edge of the root node, as demonstrated in the binary search tree analyses.

**// Recommendation**

Due to ABCU’s needs, the data structure recommended is the binary search tree. The binary search tree naturally provides data in an alphanumerical order through in-order traversal, making it ideal for printing the full course catalog. Course insertion and searching is also performed very reasonably in a well-balanced binary search tree, with typical times taking O(log n) time.

While the binary search tree is more complex to implement than the vector or hash table data structures, the benefits it provides, namely natural sorting and time efficiency, make it the optimal choice when taking into account ABCU’s requirements of loading, validating, printing, and searching course data.