**Menu Pseudocode:**

#include <iostream>

#include <string>

using namespace std:

struct Course

string coursed

string courseTitle

vector<string> coursePrerequisites

void loadData(string csvPath, data structure)

print “loading csv file”

try

for each line in the file

Course course

Course.courseId = found data from line

Course.courseTitle = found data

While there are still prerequisites

Course.prerequisit.append(data from line)

Data structure -> insert(course)

Catch

Error occurred while loading data

void printMenu(int& choice)

print “1: Load file data”

print “2: Print Course list”

print “3: Print course”

print “9: Exit”

cin >> choice

int main()

int choice = 0

define needed data structure here (vector, hash table, BST)

Course course

While (choice !=9)

printMenu()

switch(choice)

case 1:

loadData(csvPath, data structure)

break

case 2:

print all() //use specific data structure method to print A-Z alnum

break

case 3:

print “Enter coursed”

get coursed

search(data structure, courseId)

if found

print course title and prerequisites

else

course not found

break

case 9:

print “goodbye”

default:

print “not a valid choice”

return 0;

**Old Milestone Pseudocode with Time Complexity Breakdown:**

**Vector:**

#include <iostream>

#include <vector>

#include <stream>

#include <string>

using namespace std;

class Course

public:

String course\_id O(1)

String course\_title O(1)

Vector<string> prerequisites O(1)

Course ()

course\_id = “none” O(1)

course\_title = “none” O(1)

prerequisites = empty vector

Course (course \_id, course\_title, prerequisites)

this.course\_id = course\_id O(1)

this.course\_title = course\_title O(1)

this.prerequisites = prerequisites O(1)

void OpenParseAndAddCourses(Vector<Course> & Courses, String filepath,) O(n)

Try and open the Course Information csv from filepath

if no errors opening file

continue

else

display error opening file

return

while next line is not null {

Read a line from the file

Parse the line into course number, title, and prerequisites

if course\_id is string, course\_title is string, and prerequisites are string

if course has prerequisites, validate it exists as another course

Create a Course object with the parsed course information using constructor

Add the Course object to the vector

else

Display an error message indicating a formatting error

}

Close the file

Void SearchCourses(vector<Course> Courses, String find\_course) O(n)

for all courses in Courses

If find\_course is equal to course.course\_id

Print the Course Id

Print the course title

For p in course [2] // prints all prerequisites in the vector

Print p

int Main()

filepath = “inserfilename.csv” O(1)

Create vector to store Course objects O(1)

Call the OpenParseAndAddCourses(filepath, Courses) with the correct filepath O(n)

Prompt user for class they would like to search for via course\_id O(1)

Find\_course = input O(1)

SearchCourse(Courses, findCourse) O(n)

Return 0 O(1)

Total time complexity is 0(n)

**Hash Table:**

#include <iostream>

#include <vector>

#include <stream>

#include <string>

using namespace std;

class Course

public:

String course\_id

String course\_title

Vector<string> prerequisites

Course ()

course\_id = “none”

course\_title = “none”

prerequisites = empty vector

Course (course \_id, course\_title, prerequisites)

this.course\_id = course\_id

this.course\_title = course\_title

this.prerequisites = prerequisites

Class Node

Course course

Node next

Node(course)

This.course = course

This.next = null

getLen() O(n)

start = 0

count = 0

Node\* currNode = start

While (currNode is not null)

Count ++

currNode = currNode.next

return count

class HashTable

vector<Node> courses O(1)

HashedTable(size) O(1)

Set initial size with size of nodes using getLen()

Insert(course) O(1)

Key = hash(course.course\_id)

If key is null

Key = new Node

If not open

Add the course to the end of the linked list under that key

CourseSearch(courses, find\_course) O(n)

Key = hash(find\_course.id)

For all courses in course vector

If course.id == find\_course

Return course

Return default course

Int HashCourse(course) O(1)

Chose best hash function given the datatypes of the course\_id

Return key

void OpenParseAndAddCourses(Vector<Course> & Courses, String filepath,) O(n)

Try and open the Course Information csv from filepath

if no errors opening file

continue

else

display error opening file

return

while next line is not null { O(n)

Read a line from the file

Parse the line into course number, title, and prerequisites

if course\_id is string, course\_title is string, and prerequisites are string

if course has prerequisites, validate it exists as another course

Create a Course object with the parsed course information using constructor

insert(course)

else

Display an error message indicating a formatting error

}

Close the file

Void SearchCourses(vector<Course> Courses, String find\_course) O(1)

Key = hash(find\_course)

for all courses in Courses

if key is equal to hash(course.course\_id)

if course.course\_id = find\_course

Print the Course Id

Print the course title

For p in course [2] // prints all prerequisites in the vector

int Main()

filepath = “inserfilename.csv” O(1)

Create vector to store Course objects O(1)

Call the OpenParseAndAddCourses(filepath, Courses) with the correct filepath O(n)

Prompt user for class they would like to search for via course\_id O(1)

Find\_course = input O(1)

SearchCourse(Courses, findCourse) O(1)

Return 0 O(1)

Total time complexity is O(n)

**Binary Search tree:**

#include <iostream>

#include <stream>

#include <string>

using namespace std;

class Course

public:

String course\_id O(1)

String course\_title O(1)

Vector<string> prerequisites O(1)

Course () O(1)

course\_id = “none”

course\_title = “none”

prerequisites = empty vector

Course (course \_id, course\_title, prerequisites) O(1)

this.course\_id = course\_id

this.course\_title = course\_title

this.prerequisites = prerequisites

Class Node

Course course

Node\* next

Node(course) O(1)

This.course = course

This.next = null

class BinaraySearchTree

node\* root

BinarySearchTree() O(1)

Root = nullptr

Insert(Course courseId) O(log n)

If tree empty

New node(course) = root

Else

While cur node != null

If current courseId> new courseId

Move left

If we move all the way left and left is empty

Add new node and course here

Else

Move right

If we move all the way right and right is empty

Add new node and course here

void SearchCourses(string CourseId) O(log n)

Cur = root node

While cur is not null

If match

Return node

If courseId is smaller than current

Move left

Else

Move right

Return empty course aka courseId not found

void OpenParseAndAddCourses(String filepath) O(n)

Try and open the Course Information csv from filepath

if no errors opening file

continue

else

display error opening file

return

while next line is not null {

Read a line from the file

Parse the line into course number, title, and prerequisites

if course\_id is string, course\_title is string, and prerequisites are string

if course has prerequisites, validate it exists as another course

Create a Course object with the parsed course information using constructor

insert(course)

else

Display an error message indicating a formatting error

}

Close the file

int Main()

filepath = “inserfilename.csv” O(1)

Create vector to store Course objects O(1)

Call the OpenParseAndAddCourses(filepath) with the correct filepath O(n)

Prompt user for class they would like to search for via course\_id O(1)

Find\_course = input O(1)

SearchCourse(Courses, findCourse) O(log n)

Return 0 O(1)

Total time complexity is O(n)

**Evaluation:**

The 3 structures above (vector, hash table, and binary search tree) all work to load and store the course data, but the efficiencies in doing so vary. The OpenAndParseData() function will be O(n) across the board as it reads n lines and stores them in the data structures so in comparison I will leave this step out as they all share is common function.

Starting with the vector, for the simple load and search function, the loading of the course into the data structure has a time complexity of O(n) where n is the number of courses being added. This is because each course is simply added onto the end of the vector. However, the time complexity is only 0(n) if there is sufficient space for all new courses. If we must resize the vector to fit data as we add it that process also takes O(n) as we must copy all elements into a new vector and if it occurs inside the O(n) function the time now can become O (n^2). This is not ideal. While searching is still O(n) as the worst case we search the whole list, having an insert time complexity of O(n^2) when resizing is needed is something to be cautious of. The advantage of using this structure is it’s easy to use and can be indexed but it is slower than other structures.

For the hash table, inserting without collisions would be O (1) but with the use of chaining and inserting colliding elements into a linked list, the time becomes O(n) because in the worst case, all elements could be at the same key and part of that list. The same thing can be said for searching. If no collisions occur and each key has one bucket, the key and value can be found in constant time, however if the bucket is a linked list, we must traverse the list to find the value meaning worst case again could be O(n). So, while on average the hash table could be fast its worst case is O(n), better than the Vector but can be beaten. The hash table can be fast especially when using open indexing and other methods but is harder to use than a vector due to the element of hashing but could keep data more secure if that was a needed element of the program.

The Binary Search Tree, for a balanced tree can insert in O(logn) as each level gets broken down until the item is inserted. The same goes for searching. At each level the number of items to search is cut in half making the worst-case search time O (log n). I believe that the binary search tree is the most ideal data structure for the program as it has the fastest insert and search compared to the other two. Another advantage would be sorting. The other two structures would need to be sorted while the bst can simply be traverse using an in-order operation to print sorted data. The bst may be more complex than the other two methods but its performance and overall ability to complete the needed task outweighs that challenge.

For all the reasons stated above, the bst would be the best data structure to use for efficiency moving forward. Without counting the OpenAndParseData() function that all the structures share, the vector has worst case runtime of O(n^2), hash table O(n), and bst of O(log n).

|  |  |
| --- | --- |
| Data Structure | Time complexity |
| Vector | O(n^2) |
| Hash table | O(n) |
| Binary Search Tree | O(log n) |

**Print CourseId Sorted Pseudocode:**

**Vector:**

Uses quicksort the sort the CourseId values of the courses vector and print the sorted solution. Since the courseId are not only digits the use of the compareString operation will ensure that they are sorted intentionally by ACII and by digits.

Bool compareString(string a, string b)

mininumLength – min of a and b

for char in minimumLength – 1

if a[i] and b[i] are not equal

if a is digit and b is digit

return a[i] < b[i]

else

return a[i] < b[i]

Int partition(vector<Course>& courses, int low, int high)

String pivot = courses[high].courseId

Int i = low – 1

For (int j = low; j < high; j++) //loops while low is less than high

If courses[j].courseId <= pivot

i++

temp = course[i]

course[i] = course[j]

course[j] = temp

temp = course[i+1]

course[i+1] = course[high]

course[high] = temp

return i+1

void quicksort(vector<Course>& courses, int low, int high)

if low < high

int pivot\_index = partition(courses, low, high)

quicksort(courses, low, pivot\_index – 1)

quicksort(courses, pivot\_index +1, high)

void printAll(vector<Course> courses)

quicksort(courses, 0, course.size() – 1)

for every course in courses

print “courseId”

**HashTable:**

Uses a method to obtain the keys from the Nodes and then sorts the vector of keys. Once the keys are sorted using quicksort, we can access them in that order to get them in the correct order. Using the custom operation for < we can make digits and letters are being sorted correctly.

Bool compareString(string a, string b)

mininumLength – min of a and b

for char in minimumLength – 1

if a[i] and b[i] are not equal

if a is digit and b is digit

return a[i] < b[i]

else

return a[i] < b[i]

Vector getKeys(vector<Node>& nodes)

Vector<string> courseKeys

For all Node& currNode in nodes //for each course node that exists in the node vector

If chaining leads to linked list

For all elements of the linked list

Coursekeys.pushback(currNode->courseId)

else

Coursekeys.pushback(currNode->courseId)

Return courseKeys

Int partition(vector<string>& courseKeys, int low, int high)

String pivot = courseKeys[high]

Int i = low – 1

For (int j = low; j < high; j++) //loops while low is less than high

If courseKeys[j] <= pivot

i++

temp = courseKeys[i]

courseKeys[i] = course[j]

courseKeys[j] = temp

temp = courseKeys[i+1]

courseKeys[i+1] = courseKeys[high]

courseKeys[high] = temp

return i+1

void quicksort(vector<string> courseKeys, int low, int high)

if low < high

int pivot\_index = partition(courses, low, high)

quicksort(coursekeys, low, pivot\_index – 1)

quicksort(courseKeys, pivot\_index +1, high)

void printall()

vector<string> sortedKeys = getKeys(vector<Node>& nodes)

quicksort(sortedKeys, 0, sortedkeys.size() – 1)

for i in courseKeys

int key = hash(courseKey) //find the hashed location of the key

if more than one value is stored

for j in the linked list

traverse until found //we know it exists because we have found the Id already

print “courseId, coursetitle, and prerequisites”

break

else

print”courseId, courseTitle, prerequisites”

**BinarySearchTree:**

Use to traverse and print each course in order. Since comparisons are done in the inert part of the bst, we can just traverse to get them in order, no comparisons needed.

Void inOrder(Node\* node)

If (node != nullptr)

inOrder(node->left)

Print “courseId, courseTitle, prerequisites”

Else

inOrder(node->right)

Void printall()

inOrder(root)