**Vector CSV Ingest and ADT loading Pseudocode:**

Struct course:

Int course\_num;

String course\_title;

Vector\* prerequisites = NULL; // Vector of course\_num’s representing the prereqs

Vector data\_in;

File in\_file;

While not in\_file.EOF:

Read a line in from the CSV file

Send that line to the Parse\_line function along with our data\_in vector

Parse\_line( string line, vector\* array ):

course t\_course;

Assert there are at least two params to parse

Load t\_course with data from parsing the line

Array->push\_back( t\_course );

**Vector Search and print pseudocode:**

Search(vector course\_vector, int course\_id):

For each course in course\_vector:

If course.course\_id == course\_id:

Print( “Course ID: “ + course.course\_id)

Print( “Course Title: “ + course.course\_title)

If course.prerequisites != NULL:

For each prereq in course.prerequisites:

Print( “Prerequisite: “ + prereq)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each course** | 1 | n | n |
| **if course.course\_id == course\_id** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **if prereqs exist** | 1 | 1 | 1 |
| **print each prerequisites** | 2 | n | n |
| **Total Cost** | | | 4n + 2 |
| **Worst-case Runtime** | | | O(n) |

**Vector Pseudocode:**

class Vector {

public:

Vector(type t) {

size = 5;

length = 0;

array = new t[size];

}

t at(int idx) {

assert idx < length;

return array[idx];

}

void push\_back(type t) {

if length+1 > size {

resize(5);

}

Array[length++] = t;

}

private:

int size;

int length;

std::unique\_ptr\* array;

resize(int amt) {

t temp\_Arr = new t[size+amt];

copy arr to temp\_Arr;

delete array;

array = temp\_Arr;

}

}

**Hashtable CSV Ingest and ADT loading Pseudocode:**

Struct course:

Int course\_num;

String course\_title;

Vector\* prerequisites = NULL; // Vector of course\_num’s representing the prereqs

Hashtable data\_in;

File in\_file;

While not in\_file.EOF:

Read a line in from the CSV file

Send that line to the Parse\_line function along with our data\_in hashtable

Parse\_line( string line, hashtable\* tbl ):

course t\_course;

Assert there are at least two params to parse

Load t\_course with data from parsing the line

Tbl.insert(t\_course);

**Hashtable Search and print pseudocode:**

Search(hashtable course\_table, int course\_id):

Int key = hash(course\_id)

// iterate through linked list

For each course in course\_table.at(key):

If course.course\_id == course\_id:

Print( “Course ID: “ + course.course\_id)

Print( “Course Title: “ + course.course\_title)

If course.prerequisites != NULL:

For each prereq in course.prerequisites:

Print( “Prerequisite: “ + prereq)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Key = hash(course\_id** | 1 | 1 | n |
| **For each course in course\_table.at(key)** | 1 | n | n |
| If course.course\_id == course\_id | 1 | n | 1 |
| **Print course info** | 1 | 1 | n |
| **If prereqs exist** | 1 | 1 | n |
| **Print each prereq** | 1 | n | n |
| **Total Cost** | | | 3n + 3 |
| **Worst-case Runtime** | | | O(n) |

\*Note that n here is a subset of the superset

**Hashtable Pseudocode:**

Class hashtable:

Struct Node:

Node\* next = null;

Int value = -1;

Unsigned int key = uint\_max;

vector<node> nodes;

unsigned int tablesize;

Hashtable\_Constructor(unsigned int size):

tablesize = size;

Nodes.resize(tablesize);

Hashtable\_Hash(int key):

Key % tablesize;

Hashtable\_insert(Node some\_node):

Int key = Hashtable\_Hash(some\_node.value);

If nodes.at(key).value != -1:

Node\* open\_node = Nodes.find\_next\_open();

Open\_node = some\_node;

Hashtable\_search(int value):

Int key = Hashtable\_Hash(value);

Node\* curr\_node = nodes.at(key);

While curr\_node.next != null and curr\_node.value != value:

Curr\_node = curr\_node.next;

Return curr\_node;

Hashtable\_remove(int value):

Int key = Hashtable\_Hash(value);

Node\* curr\_node = nodes.at(key);

While curr\_node.next != null and curr\_node.value != value:

Curr\_node = curr\_node.next;

If curr\_node.value == value:

Delete curr\_node;

**BST CSV Ingest and ADT loading Pseudocode:**

Struct course:

Int course\_num;

String course\_title;

Vector\* prerequisites = NULL; // Vector of course\_num’s representing the prereqs

bst data\_in;

File in\_file;

While not in\_file.EOF:

Read a line in from the CSV file

Send that line to the Parse\_line function along with our data\_in tree

Parse\_line( string line, bst\* tree ):

course t\_course;

Assert there are at least two params to parse

Load t\_course with data from parsing the line

Tree.insert( t\_course );

**BST pseudocode:**

Search(bst course\_tree, int course\_id):

For each course in course\_tree:

If course.course\_id == course\_id:

Print( “Course ID: “ + course.course\_id)

Print( “Course Title: “ + course.course\_title)

If course.prerequisites != NULL:

For each prereq in course.prerequisites:

Print( “Prerequisite: “ + prereq)

Else If course.course\_id > course\_id:

Search(course.left)

Else If course.course\_id < course\_id:

Search(course.right)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each course in course\_tree** | 1 | Log(n) | Log(n) |
| **if course.course\_id == course\_id** | 1 | Log(n) | Log(n) |
| **print out the course information** | 1 | 1 | 1 |
| **If prereqs exist** | 1 | 1 | 1 |
| **print the prereqs** | 1 | Log(n) | Log(n) |
| **Else if course.course\_id > course\_id** | 1 | Log(n) | Log(n) |
| **Search left** | 1 | Log(n) | Log(n) |
| **Else if course.course\_id < course\_id** | 1 | Log(n) | Log(n) |
| **Search right** | 1 | Log(n) | Log(n) |
| **Total Cost** | | | 7n + 2 |
| **Worst-case Runtime** | | | O(log(n)) |

Insert(course t\_course):

If there is no root node:

Root = new Node(t\_course)

Return

Else:

While not on a leaf with a null ptr:

If t\_course.course\_id is less, traverse left

If t\_course.course\_id is more, traverse right

Set leaf’s right to a new node with t\_course if more

Set leaf’s left to a new node with t\_course if less

PrintInOrder(Node\* node):

If node is null:

Return

PrintInOrder(node->left)

Print(node)

PrintInOrder(node->right)

Menu():

Input choice

Switch choice

Case ‘load courses’:

Figure out type to use (vector/hashtable/bst)

Route to CSV parser for that datatype

Case ‘print courses’:

Route to print function of datatype

Case ‘find course’:

Route to search function of datatype

Case ‘exit’:

Exit program

Analysis:

A hashtable is by far the most efficient data structure for searching, however storage can be computationally more expensive and unused allocated space in the table costs memory.

A vector is a nice and simple data structure that fits this assignment nicely due to the limited size of possible classes, targeted architecture and infrastructure which has plenty of compute and memory for our needs, and its multitude of search strategies we could utilize. The disadvantages here would most likely be the hits we would take for the resizes done on the backend of the vector class if the course count ever got up high. The search time is also not the best it could be, but, for the purposes of this assignment, would be negligible at the targeted scale.

A binary search tree doesn’t fit this use-case too well, as there is no algorithmic drive to have the data in a tree structure. Searching would be slower than a hashtable, though slightly more space-efficient. Typically things that benefit from a tree would have a natural relationship that could be mapped out into a tree, such as a network map, company directory, or filesystem. Advantages would be that it probably has the best compromise between search speed ( O(log(n)) ) and memory footprint.