Project One

Thomas Dang

For the ABCU project I will examine three potential data structures: Vector, hash table, and binary search tree (BST). Each of these offer their own pros and cons in run time, memory, and implementation.

Beginning with the vector, some of the advantages are that C++ has built in functions for manipulating a vector such as sorting, and it will only utilize the amount of memory needed for the size of the vector. It is efficient when adding or removing items to or from the end of the vector. However, when inserting or removing from other points in the vector, the remaining elements will all need to shift with a potential O(n) runtime. Searching for an element in a vector also has a potential O(n) runtime.

The hash table in comparison has a very efficient search capability, often O(1). A worst case analysis can still result in O(log n) for a search however, due to potential hash collision. Unlike the vector a hash table can readily insert or remove elements from any position. The down side is that it is an unordered data structure, and will require additional logic to implement the ordered lists of courses requested in this project. Hash tables can also reserve more memory than necessary for the data being stored, although proper data management can mitigate this.

The BST will be in order by the nature of its data structure, and it can be readily traversed ascending or descending. While it does not have as good a search runtime as the hash table, it is more efficient than the vector. The worst performance of a search with this data structure is O(log n + 1), as it divides the branch each time it checks an entry. A downside is the need to navigate the tree each time a new entry is added. Likewise, removing an entry may require removing child and parent nodes adding complexity. The BST does not allocate more memory than it needs to store the assigned data.

The primary requirements of the ABCU project are to load all courses into a data structure, print a list of all data in ascending order, and search and print information about a selected course. For these purposes, I would recommend using a hash table. An average O(1) search time will be beneficial to the program. The hash table is efficient at loading new data in comparison to a binary search tree. It will require combining data structures to display the list in order, and I would recommend a sorted vector to do this. The vector’s entries can be called in order and searched for in the hash table when the user wishes to display the courses in order. Additionally, while the current plan for the program does not call for functions to add or remove courses, a hash table will be able to implement this more efficiently than the other data structures.

Vector

Course {

string courseId

string courseName

vector<string> prereqs

}

vector<Course> loadCourses(string fileName){

Initialize ifstream

Initialize currentLine

Initialize vector<Course> courses

Ifstream.open(fileName)

WHILE (not end of file)

WHILE getline ifstream, currentLine, \n

Initialize Course course

Initialize string value

Initialize vector<string> row

WHILE currentLine is not empty

IF find ‘,’ in currentLine

Value = currentLine substring to ‘,’

Erase currentLine substring to ‘,’

Add value to row

ELSE

Add currentLine to row

Set currentLine to empty

End while

Set course courseNumber to row[0]

Set course courseName to row[1]

FOR int i = 2, i < row size, i++

Add row[i] to course prereqs

End For

Add course to courses

End while

End while

Call courses.sort

Call validate(courses)

Return courses

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize ifstream** | 1 | 1 | 1 |
| **Initialize currentLine** | 1 | 1 | 1 |
| **Initialize courses** | 1 | 1 | 1 |
| WHILE (not end of file) | 1 | n | n |
| WHILE getline ifstream, currentLine, \n | 1 | n | n2 |
| Initialize course | 1 | n | n |
| Init string value | 1 | n | n |
| Init vector row | 1 | n | n |
| WHILE currentLine is not empty | 1 | n | n3 |
| IF find ‘,’ in currentLine | 1 | n | n |
| Value = currentLine substring to ‘,’ | 1 | 1 | 1 |
| Erase currentLine substring to ‘,’ | 1 | 1 | 1 |
| Add value to row | 1 | 1 | 1 |
| Add currentLine to row | 1 | 1 | 1 |
| Set currentLine to empty | 1 | 1 | 1 |
| Set course courseNumber to row[0] | 1 | 1 | 1 |
| Set course courseName to row[1] | 1 | 1 | 1 |
| FOR int i = 2, i < row size, i++ | 1 | n | n |
| Add row[i] to course prereqs | 1 | n | n |
| Add course to courses | 1 | 1 | 1 |
| Call courses.sort | 1 | 1 | O(n) |
| Call validate(courses) | 1 | 1 |  |
| Return courses | 1 | 1 | 1 |
| **Total Cost** | | | 9n + 12 |
| **Runtime** | | | O(n3) |

void validate(vector<Course> courses){

For each course of courses

Initialize vector<string> prereqs as course.prereqs

If prereqs size is greater than 0

For each prereq

Search for prereq in courses

If course is not found

Erase prereq from course prereqs

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **Initialize vector** | 1 | n | n |
| If prereqs size is greater than 0 | 1 | n | n |
| For each prereq | 1 | n2 | n2 |
| Search for prereq in courses | 1 | n | n |
| If course is not found | 1 | 1 | 1 |
| Erase prereq from course prereqs | 1 | 1 | 1 |
| **Total Cost** | | | 5n2 + 1 |
| **Runtime** | | | O(n2) |

Course searchCourses(Vector<Course> courses, string courseId){

Initialize Course returnCourse

FOR each course of courses

IF course courseId is same as courseId

SET returnCourse equal course

break

endfor

RETURN returnCourse

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize returnCourse** | 1 | 1 | 1 |
| **For each course of courses** | 1 | n | n |
| IF course courseId is same as courseId | 1 | n | n |
| **Set returnCourse to course** | 1 | 1 | 1 |
| **break** | 1 | 1 | 1 |
| **Return returnCourse** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 4 |
| **Runtime** | | | O(n) |

void printCourse(Course course){

cout course.courseNumber “ | “ course.courseName endl

IF course.prereqs is not empty

cout “Prerequisites”

FOR each string prereq of course.prereqs

cout prereq “, “

endfor

cout endl

end if

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| cout course.courseNumber “ | “ course.courseName endl | 1 | 1 | 1 |
| IF course.prereqs is not empty | 1 | 1 | 1 |
| cout “Prerequisites” | 1 | 1 | 1 |
| FOR each string prereq of course.prereqs | 1 | n | n |
| cout prereq “, “ | 1 | n | n |
| **Total Cost** | | | 2n + 3 |
| **Runtime** | | | O(n) |

void printCourses(vector<Course> courses){

Foreach Course course of courses

Cout course.courseNumber “, “ course.courseName endl

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Foreach Course course of courses | 1 | n | n |
| Cout course.courseNumber “, “ course.courseName endl | 1 | n | n |
| **Total Cost** | | | 2n |
| **Runtime** | | | O(n) |

Hashtable

Course {

string courseId

string courseName

vector<string> prereqs

}

HashTable{

Node{

Course course

Unsigned int key

Node next

Node(){

Key = max

Next = null

}

Node(Course newCourse) {

course = newCourse

}

Node(Course newCourse, unsigned int newKey){

key = newKey

}

}

vector<Node> nodes

vector<string> courseNumbers

unsigned int size = default size

unsigned int hash(int key)

HashTable()

HashTable(int size)

void insert(Course course)

void printCourses()

Course searchCourse()

}

void HashTable::insert(Course course){

Initialize int key

Initialize Node node

Set key from hash of course.courseNumber

Set node to node key position of nodes

If node does not exist

Create newNode(course, key)

Insert newNode into nodes

End if

Else if node key is UINT\_MAX, node is empty

Set node.key to key

Set node.course to course

Set node.next to null

End else if

Else

While node.next is not null

Set node to node.next

End while

Set node.key to key

Set node.course to course

Set node.next to null

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize int key** | 1 | 1 | 1 |
| **Initialize Node node** | 1 | 1 | 1 |
| Set key from hash of course.courseNumber | 1 | 1 | 1 |
| Set node to node key position of nodes | 1 | 1 | 1 |
| If node does not exist | 1 | 1 | 1 |
| Create newNode(course, key) | 1 | 1 | 1 |
| Insert newNode into nodes | 1 | 1 | 1 |
| Else if node key is UINT\_MAX, node is empty | 1 | 1 | 1 |
| Set node.key to key | 1 | 1 | 1 |
| Set node.course to course | 1 | 1 | 1 |
| Set node.next to null | 1 | 1 | 1 |
| Else While node.next is not null | 1 | n | N |
| Set node to node.next | 1 | n | N |
| Set node.key to key | 1 | 1 | 1 |
| Set node.course to course | 1 | 1 | 1 |
| Set node.next to null | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 14 |
| **Runtime** | | | O(n) |

void FileInput(string filename, HashTable hashtable){

Initialize ifstream

Initialize currentLine

Ifstream.open(fileName)

WHILE (not end of file)

WHILE getline ifstream, currentLine, \n

Initialize Course course

Initialize string value

Initialize vector<string> row

WHILE currentLine is not empty

IF find ‘,’ in currentLine

Value = currentLine substring to ‘,’

Erase currentLine substring to ‘,’

Add value to row

ELSE

Add currentLine to row

Set currentLine to empty

End while

Set course courseNumber to row[0]

Add courseNumber to courseNumbers

Set course courseName to row[1]

FOR int i = 2, i < row size, i++

Add row[i] to course prereqs

End For

Insert course into hashtable

End while

End while

Call sort function on courseNumbers

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize ifstream** | 1 | 1 | 1 |
| **Initialize currentLine** | 1 | 1 | 1 |
| Ifstream.open(fileName) | 1 | 1 | 1 |
| WHILE (not end of file) | 1 | n | N |
| WHILE getline ifstream, currentLine, \n | 1 | n2 | n2 |
| Initialize Course course | 1 | 1 | 1 |
| Initialize string value | 1 | 1 | 1 |
| Initialize vector<string> row | 1 | 1 | 1 |
| WHILE currentLine is not empty | 1 | n3 | n3 |
| IF find ‘,’ in currentLine | 1 | 1 | 1 |
| Value = currentLine substring to ‘,’ | 1 | 1 | 1 |
| Erase currentLine substring to ‘,’ | 1 | 1 | 1 |
| Add value to row | 1 | 1 | 1 |
| ELSE Add currentLine to row | 1 | 1 | 1 |
| Set currentLine to empty | 1 | 1 | 1 |
| Set course courseNumber to row[0] | 1 | 1 | 1 |
| Add courseNumber to courseNumbers | 1 | 1 | 1 |
| Set course courseName to row[1] | 1 | 1 | 1 |
| FOR int i = 2, i < row size, i++ | 1 | n | n |
| Add row[i] to course prereqs | 1 | 1 | 1 |
| Insert course into hashtable | 1 | 1 | 1 |
| Call sort function on courseNumbers | 1 | n | n |
| **Total Cost** | | | 2n + n3 + 17 |
| **Runtime** | | | O(n3) |

Course HashTable::searchCourse(string courseNumber){  
 Initialize Course course

Get key from by hash of courseNumber

Create node equal to nodes at key

If node is null

Return course

While node is not null

If node.course.courseNumber is equal to courseNumber

Return node.course

Endif

Set node = node.next

Endwhile

Return course

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize course** | 1 | 1 | 1 |
| Get key from by hash of coursNumber | 1 | 1 | 1 |
| Create node equal to nodes at key | 1 | 1 | 1 |
| If node is null | 1 | 1 | 1 |
| **Return course** | 1 | 1 | 1 |
| While node is not null | 1 | n | n |
| If node.course.courseNumber is equal to courseNumber | 1 | n | n |
| Return node.course | 1 | 1 | 1 |
| Set node = node.next | 1 | n | n |
| Return course | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 7 |
| **Runtime** | | | O(n) |

Void printCourseInfo(string courseNumber){

Course course = searchCourse(courseNumber)

Print course.courseNumber course.courseName

If course prereqs size > 0

For each string prereq of course prereqs

Print prereq

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Course course = searchCourse(courseNumber) | 1 | 1 | n |
| Print course.courseNumber course.courseName | 1 | 1 | 1 |
| If course prereqs size > 0 | 1 | n | n |
| For each string prereq of course prereqs | 1 | n | n |
| Print prereq | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

Void HashTable::printCourses(){  
 For each string courseNumber of courseNumbers

New Course course = Search(courseNumber)

If course found and not empty

Print course.courseNumber course.courseName

Endif

endfor

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For each string courseNumber of courseNumbers | 1 | n | n |
| New Course course = Search(courseNumber) | 1 | n | n |
| If course found and not empty | 1 | n | n |
| Print course.courseNumber course.courseName | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 1 |
| **Runtime** | | | O(n) |

Binary Search Tree

Course {

string courseId

string courseName

vector<string> prereqs

}

Node {

Cousre course

Node left

Node right

Node(){

Left = right = null

}

Node(Course newCourse){

course = newCourse

}

}

BinarySearchTree{

Node root

BinarySearchTree()

void addNode(Node node, Course course)

void printOrderedCourses(Node node)

void insert(Course course)

Course search(string courseNumber)

}

BinarySearchTree::BinarySearchTree(){

Root = null

}

BinarySearchTree::addNode(Node node, Course course){

If node.course.courseNumber is greater than course.courseNumber

If node.left is null

Set node left = new node(course)

Else

addNode(node.left, course)

Else

If node right is null

Set node right = new node(course)

Else

addNode(node.right, course)

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If node.course.courseNumber is greater than course.courseNumber | 1 | 1 | 1 |
| If node.left is null | 1 | 1 | 1 |
| Set node left = new node(course) | 1 | 1 | 1 |
| Else addNode(node.left, course) | 1 | n/2 | n/2 |
| Else If node right is null | 1 | 1 | 1 |
| Set node right = new node(course) | 1 | 1 | 1 |
| Else addNode(node.right, course) | 1 | n/2 | n/2 |
| **Total Cost** | | | T(n/2) + 4 |
| **Runtime** | | | O(log n) |

BinarySearchTree::void printOrderedCourses(Node node){

If node is not null

printOrderedCourses(node.left)

Print courseNumber courseName

printOrderedCourses(node.right)

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If node is not null | 1 | 1 | 1 |
| printOrderedCourses(node.left) | 1 | n | n |
| Print courseNumber courseName | 1 | 1 | 1 |
| printOrderedCourses(node.right) | 1 | n | n |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

BinarySearchTree::void insert(Course course){

If root is null

Set root equal to new Node(course)

Else

addNode(root, course)

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If root is null | 1 | 1 | 1 |
| Set root equal to new Node(course) | 1 | 1 | 1 |
| Else addNode(root, course) | 1 | 1 | Log n |
| **Total Cost** | | | Log n + 2 |
| **Runtime** | | | O(log n) |

BinarySearchTree::Course search(string courseNumber){

Initialize Node current

While current is not null

If current.course.courseNumber is equal to courseNumber

Return current.course

If current.course.courseNumber is less than courseNumber

current = current.left

Else

current = current.right

Initialize Course empty

return empty

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Initialize Node current | 1 | 1 | 1 |
| While current is not null | 1 | n | n |
| If current.course.courseNumber is equal to courseNumber | 1 | n | n |
| Return current.course | 1 | 1 | 1 |
| If current.course.courseNumber is less than courseNumber | 1 | n | n |
| current = current.left | 1 | 1 | 1 |
| Else current = current.right | 1 | n | n |
| Initialize Course empty | 1 | 1 | 1 |
| return empty | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 5 |
| **Runtime** | | | O(n) |

void loadCourses(string filename, BinarySearchTable courses){

Initialize ifstream

Initialize currentLine

Ifstream.open(fileName)

WHILE (not end of file)

WHILE getline ifstream, currentLine, \n

Initialize Course course

Initialize string value

Initialize vector<string> row

WHILE currentLine is not empty

IF find ‘,’ in currentLine

Value = currentLine substring to ‘,’

Erase currentLine substring to ‘,’

Add value to row

ELSE

Add currentLine to row

Set currentLine to empty

End while

Set course courseNumber to row[0]

Set course courseName to row[1]

FOR int i = 2, i < row size, i++

Add row[i] to course prereqs

End For

Insert course to courses

End while

End while

Return courses

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize ifstream** | 1 | 1 | 1 |
| **Initialize currentLine** | 1 | 1 | 1 |
| Ifstream.open(fileName) | 1 | 1 | 1 |
| WHILE (not end of file) | 1 | n | N |
| WHILE getline ifstream, currentLine, \n | 1 | n2 | n2 |
| Initialize Course course | 1 | 1 | 1 |
| Initialize string value | 1 | 1 | 1 |
| Initialize vector<string> row | 1 | 1 | 1 |
| WHILE currentLine is not empty | 1 | n3 | n3 |
| IF find ‘,’ in currentLine | 1 | 1 | 1 |
| Value = currentLine substring to ‘,’ | 1 | 1 | 1 |
| Erase currentLine substring to ‘,’ | 1 | 1 | 1 |
| Add value to row | 1 | 1 | 1 |
| ELSE Add currentLine to row | 1 | 1 | 1 |
| Set currentLine to empty | 1 | 1 | 1 |
| Set course courseNumber to row[0] | 1 | 1 | 1 |
| Add courseNumber to courseNumbers | 1 | 1 | 1 |
| Set course courseName to row[1] | 1 | 1 | 1 |
| FOR int i = 2, i < row size, i++ | 1 | n | n |
| Add row[i] to course prereqs | 1 | 1 | 1 |
| Insert course into courses | 1 | 1 | Log n |
| **Total Cost** | | | Log n + n3 + 16 |
| **Runtime** | | | O(n3) |

void printCourse(string courseNumber){

Initialize Course course

Set course = searchCourse(courseNumber)

cout course.courseNumber “ | “ course.courseName endl

IF course.prereqs is not empty

cout “Prerequisites”

FOR each string prereq of course.prereqs

Print prereq info

endfor

cout endl

end if

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Initialize Course course | 1 | 1 | 1 |
| Set course = searchCourse(courseNumber) | 1 | 1 | n |
| cout course.courseNumber “ | “ course.courseName endl | 1 | 1 | 1 |
| IF course.prereqs is not empty | 1 | 1 | 1 |
| cout “Prerequisites” | 1 | 1 | 1 |
| FOR each string prereq of course.prereqs | 1 | n | n |
| Print prereq info | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 3n + 4 |
| **Runtime** | | | O(n) |

Menu

void printMenu(){

Print welcome to course planner

Print 1. Load data structure

Print 2. Print course list

Print 3. Print course

Print 9. Exit

Print What would you like to do?

}

Int main(){

Int choice

String course number

While choice is not 9

printMenu()

Set choice equal to user input

Switch(choice)

Case 1

loadData

break

Case 2

printCourses

break

Case 3

Print What course do you want to know about?

Set coursenumber to user input

printCourse(courseNumber)

break

Case 9

Print exit

Break

Default

Print Please enter a valid choice

End switch

Return 0

}