Raphael Coloma

CS-300-T2809 DSA: Analysis and Design

December 1, 2022

Project one – Run-Time Evaluation

## Vector Data Structure Pseudocode

Define Struct Course

string courseNumber

string courseName

vector string coursePrereq{}

**int partition(vector<Course>& courses, int begin, int end)** {

Initialize low and high variables to the begin and end indexes of the vector

Set string variable to the midpoint courseNumber between the high and low variables

While loop - low is less than high

While loop - the low courseNumber element is less than the midpoint string variable

Increment the low variable

While loop - the high courseNumber element is greater than the midpoint string variable

decrement the low variable

If statement – low is less than high

Call swap function to swap the courses at the low and high elements of the couse vector

Increment low

Decrement high

Return high variable

}

**void quickSort(vector<Course>& courses, int begin, int end)** {

Initialize integer variable to 0

If statement – vector beginning is greater than or equal to the end of the vector

return

Call partition function passing the couse vector, the start, and the end – assign it to the integer variable

Recursively call quickSort fuction passing the couse vector, the start, and the integer variable

Recursively call quickSort fuction passing the couse vector, the integer variable plus 1, and the end

}

**vector<Course> loadCourses(string csvPath)** {

Declare ifstream variable

Declare string variable

Declare string vector variable

ifstream open file

while loop to read entire line until the end of file

Create Course struct for line read

Call getline function passing ifstream variable and string variable

Pass read line to stringstream variable

While loop Calling getline function passing the stringstream variable, string variable, & delimiter

Push back string variables to string vector

Assign 0 element to Course struct courseNumber

Assign 1 element to Course struct courseName

If statement – string vector has less than 2 elements

Display error

Else

If statement – string vector has more than 2 elements

While loop starting at 3 element until size of vector

Push the ith element to the Course struct coursePrereq string vector

pushback course

close file

}

**void printCourse(Vector<Course> courses)** {

Display course information with necessary formatting

return

}

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

for loop - course vector size

if statement - compare courseNumber of i element in courses vector to passed courseNumber

print course information with necessary formatting

for loop – course prerequisite vector size of i element in courses vector

print prerequisite course information with necessary formatting

Return

}

**int main(int argc, char\* argv[])** {

Process commandline arguments

Define a Course vector to hold the courses

Initialize integer varible for user input

While loop to accept user input – Exit on 9

Display Program menu with description of choices 1-3 exit on 9

Store user input variable

Call switch function passing choice variable

Define case 1

Call method to loadCourses passing file variable

Display number of courses read

Break

Define case 2

Call sortCourses function passing the couse vector, the start, and the end of the vector

For Loop to cycle courses based on vector size

Call printCourse function passing course vector at element i

New line

Break

Define case 3

Print request for Course Number

Assign user input to string variable

Call to searchCourse function passing courses vector and course number string

Break

Return 0

}

## Hash Table Data Structure Pseudocode

**Define Struct Course**

string cNumber

string cName

int prereqCount

vector string cPrereqN

Establish tableSize

Initialize nodes to tableSize with resize function

Create/Initialize Hashtable

**unsigned int Hashtable::hash(int key)** {

Module courseNumber by tablesize

Return result

}

**void HashTable::Insert(Course course)** {

Create key variable and set it to the hash function call passing converted courseNumber c\_string

Create node pointer variable oldNode and set to address of node key

If oldNode equals nullptr

Set node pointer variable newNode to newly created node with bid and key

Call node insert function to set new node as beginning node of key

Else

If oldNode key is empty

Set oldNode corresponding pointers to key, bid, and nullptr

else

while loop oldNode next pointer is not nullptr

Advance oldNode

Set oldNode’s next pointer to newly created Node with bid and key

}

**vector<Course> loadCourses(string csvPath)** {

Declare ifstream variable

Declare string variable

Declare string vector variable

ifstream open file

while loop to read entire line until the end of file

Create Course struct for line read

Call getline function passing ifstream variable and string variable

Pass read line to stringstream variable

While loop Calling getline function passing the stringstream variable, string variable, & delimiter

Push back string variables to string vector

Assign 0 element to Course struct courseNumber

Assign 1 element to Course struct courseName

If statement – string vector has less than 2 elements

Display error

Else

If statement – string vector has more than 2 elements

While loop starting at 3 element until size of vector

Push the ith element to the Course struct coursePrereq string vector

insert course to hashTable

close file

}

**void HashTable::printCourse()** {

For loop interating through hash table - starting with the beginning node to the ending node

If statement to check if bucket is empty

Display bid with formatting

Node pointer is equal to next iteration

While loop to check chained bids in bucket

Display bid node from linked list with formatting

Assign node is equal to next node

}

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

Create key variable and set it to the hash function call passing converted courseNumber c\_string

Create node pointer variable node and set to address of node key

If node is not nullptr AND bucket is not empty AND course number matches passed variable

Print course information with necessary formating

If node equals nullptr OR bucket is empty

Print “course number not found”

While loop node is not null

If node course number matches passed courseNumber variable

Print course information with necessary formating

Advance node

}

**int main(int argc, char\* argv[])** {

Process command line arguments

Define a hash table to hold all the courses

Create Course Struct

Initialize new HashTable variable

Initialize user input

While loop to accept user input – Exit on 9

Display Program menu with description of choices 1-3 exit on 9

Store user input variable

Call switch function passing choice variable

Define case 1

Call method to loadCourses passing file variable and HashTable variable

Break

Define case 2

Call printCourse function through HashTable variable pointer

Break

Define case 3

Print request for Course Number

Assign user input to string variable

Call to searchCourse function passing courses vector and course number string through HashTable variable pointer

Break

Return 0

}

## Tree Data Structure Pseudocode

**Define Struct Course**

string cNumber

string cName

int prereqCount

vector string cPrereqN

**Define Struct Node**

Course course

Node pointer left

Node pointer right

Default constructor for Node()

left = nullptr

right = nullptr

Initialize Course with a course

Course = aCourse

**Define BinarySearchTree Class**

Node pointer root

BinarySearchTree function

Void inOrder function

Void Insert function pass Course course

Void printCourseInformation function pass Course courses and string cNumber

**BinarySearchTree::BinarySearchTree()** {

Root is equal to nullptr

}

**void BinarySearchTree::printCourse()** {

Call printCourse fuction and pass root

}

**void BinarySearchTree::Insert(Course course)** {

If statement to check if tree is empty

Sets root as new course node

Else - Tree is not empty

Call addNode function with THIS pointer and pass course to root

}

**void BinarySearchTree::addNode(Node\* node, Bid bid)** {

If statement – Call compare function to compare courseNumbers. Node is larger if returned value > 0

If statement to check if parent has a left child

Sets left child to new bid Node

Else - Parent has a left child

Recursive call to addNode to traverse down left child

Else - Node courseNumber is smaller

If statement to check if parent has a right child

Sets right child to new bid Node

Else - Parent has a right child

Recursive call to addNode to traverse down right child

}

**void BinarySearchTree::printCourse(Node\* node)** {

If statement node is not equal to null ptr

Recursive call to inOrder for the left child

Display course information with necessary formatting

Recursive call to inOrder for the right child

}

**vector<Course> loadCourses(string csvPath)** {

Declare ifstream variable

Declare string variable

Declare string vector variable

ifstream open file

while loop to read entire line until the end of file

Create Course struct for line read

Call getline function passing ifstream variable and string variable

Pass read line to stringstream variable

While loop Calling getline function passing the stringstream variable, string variable, & delimiter

Push back string variables to string vector

Assign 0 element to Course struct courseNumber

Assign 1 element to Course struct courseName

If statement – string vector has less than 2 elements

Display error

Else

If statement – string vector has more than 2 elements

While loop starting at 3 element until size of vector

Push the ith element to the Course struct coursePrereq string vector

push this course to the end with call to insert function passing course

close file

}

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

Create/set current node variable equal to root of tree

While loop to traverse tree - curNode will be null once it reaches the bottom

If statement to compare course numbers with current node – Match returns 0

Print course information with necessary formatting

Return

else if statement to compare course numbers with current node – Smaller negative

Set current node to current node’s left child for traversal

Else – course number is larger than current Node

Set current node to current node’s right child for traversal

Print “Course number not found”

Return

}

**int main(int argc, char\* argv[])** {

Process command line arguments

Initialize search course variable

Define a binary search tree to hold all courses

Create Course struct

Initialize user input variable

While loop to accept user input – Exit on 9

Display Program menu with description of choices 1-3 exit on 9

Store user input variable

Call switch function passing choice variable

Define case 1

Call method to loadCourses passing file variable and Binary tree variable

Break

Define case 2

Call printCourse function through Binary Tree variable pointer

Break

Define case 3

Print request for Course Number

Assign user input to string variable

Call to searchCourse function passing courses vector and course number string through Binary Tree variable pointer

Break

Return 0

}

## Creating Course Objects

The programs read input files the same way. After the user selects the option to load the Courses, the program calls the loadCourses function. The function loops through the file reading one line at a time. The line is passed to the stringstream and separated into strings with the commas serving as delimiters. Each string is pushed as elements in a string vector. The program then checks the number of elements in the string vector. If the number of elements is less than 2, an error message is displayed, and the line is skipped. If the number of elements is 2 or greater, the first two elements are assigned to the courseNumber and courseName string members of the Course struct. Each remaining element is pushed to the coursePrereq string vector member of the Course struct. Once the Course object is created, it is added to the programs corresponding data structure. This continues until the end of the file is reached.

For the Vector Data Structure program, the Course object is appended to the end of the Course vector, courses.

In the Hash Table Data Structure program, the hashTable created is compiled of link list buckets. The program maps the Course object to a bucket using a key. The program calls the insert function passing the Course object. The Course object is assigned a key by converting its courseNumber to an integer and applying the modulo operation by the tableSize. If the Course object is the first element for the assigned bucket, the Course object will be set as the beginning of the linked list for the respective bucket. If a collision occurs and the Course object is not the first element for the assigned bucket, the program will traverse to the last node of the link list for the respective bucket and point to the Course object as the next node.

The Tree Data Structure program assigns the Course objects as nodes on the binary search tree (BST). Starting from the BST root, each node can have up to 2 children: a left child and a right child. The Tree Data Structure program calls the insert function and passes the course. If the tree is empty, the program will assign the Course object as the root. If the tree is not empty, the addNode function is called passing the root and the Course object to determine where the Course object’s assignment. In a BST, the placement of the child nodes to the parent is based on their value. The left child is less than the parent and the right child is greater than the parent. If the courseNumber of the Course object being added is less than the courseNumber of the node already on the tree, the program will check if the node has a left child. If not, a new node is created, passing the Course object, as the existing node’s left child. If the node does have a left child, the program will make a recursive call to the addNode function to traverse down the left child and make the comparison again with the left child. This is the same concept with the right child, but if the courseNumber of the Course object being added is greater than the courseNumber of the node already on the tree. If the node does not have a right child, it will create a new node, passing the Course object, and assign it as the right child. If the node does have a right child, the program will make a recursive call to the addNode function to traverse down the right child and make the comparison again with the right child.

## Run-Time Evaluation Tables

**Vector Data Structure – O(N^2)**

| **partition Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| While loop - low is less than high | 1 | N | N |
| While loop - the low is less than the midpoint string variable | 1 | N | N |
| Increment the low variable | 1 | 1 | 1 |
| While loop - the high is greater than the midpoint string variable | 1 | N | N |
| Decrement the low variable | 1 | 1 | 1 |
| If statement – low is less than high | 1 | N | N |
| Call swap function to swap the low and high elements | 1 | N | N |
| Increment low | 1 | 1 | 1 |
| Decrement high | 1 | 1 | 1 |
| **Total Cost** | | | **5N + 5** |
| **Runtime** | | | **O(N)** |

| **quickSort Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If statement – beginning is greater than or equal to the end of the vector | 1 | N | N |
| Return | 1 | 1 | 1 |
| Call partition function – assign it to mid | 1 | N | N |
| Recursively call quickSort function – start to mid | 1 | N\*N | N\*N |
| Recursively call quickSort function – mid to end | 1 | N\*N | N\*N |
| **Total Cost** | | | **2N^2+2N+1** |
| **Runtime** | | | **O(N^2)** |

| **loadCourses Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| While loop to read entire line until the end of file | 1 | N | N |
| Create Course struct for line read | 1 | 1 | 1 |
| Call getline function passing ifstream variable and string variable | 1 | N | N |
| Pass read line to stringstream variable | 1 | 1 | 1 |
| While loop Calling getline function | 1 | N | N |
| Push back string variables to string vector | 1 | N | N |
| Assign 0 element to Course struct courseNumber | 1 | 1 | 1 |
| Assign 1 element to Course struct courseName | 1 | 1 | 1 |
| If statement – string vector has less than 2 elements | 1 | N | N |
| Display error message | 1 | 1 | 1 |
| Else - If statement – string vector has more than 2 elements | 1 | N | N |
| While loop starting at 3 element until size of vector | 1 | N | N |
| Push the ith element to the Course struct coursePrereq | 1 | N | N |
| pushback course | 1 | N | N |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | **9N+10** |
| **Runtime** | | | **O(N)** |

| **searchCourse Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For loop - course vector size | 1 | N | N |
| if statement - compare courseNumbers | 1 | N | N |
| print course information with necessary formatting | 1 | 1 | 1 |
| for loop – course prerequisite vector size | 1 | N | N |
| print prerequisite course information | 1 | N | N |
| Return | 1 | 1 | 1 |
| **Total Cost** | | | **4N+2** |
| **Runtime** | | | **O(N)** |

**Hash Table Data Structure – O(N)**

| **loadCourses Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| While loop to read entire line until the end of file | 1 | N | N |
| Create Course struct for line read | 1 | 1 | 1 |
| Call getline function passing ifstream variable and string variable | 1 | N | N |
| Pass read line to stringstream variable | 1 | 1 | 1 |
| While loop Calling getline function | 1 | N | N |
| Push back string variables to string vector | 1 | N | N |
| Assign 0 element to Course struct courseNumber | 1 | 1 | 1 |
| Assign 1 element to Course struct courseName | 1 | 1 | 1 |
| If statement – string vector has less than 2 elements | 1 | N | N |
| Display error message | 1 | 1 | 1 |
| Else - If statement – string vector has more than 2 elements | 1 | N | N |
| While loop starting at 3 element until size of vector | 1 | N | N |
| Push the ith element to the Course struct coursePrereq | 1 | N | N |
| insert course to hashTable | 1 | N | N |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | **9N+10** |
| **Runtime** | | | **O(N)** |

| **insert Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create key variable and set it to the hash function call | 1 | 1 | 1 |
| Create node pointer variable and set to address of node key | 1 | 1 | 1 |
| If oldNode equals nullptr | 1 | N | N |
| Set node pointer newNode to newly created node | 1 | 1 | 1 |
| Call node insert function set new node as beginning | 1 | N | N |
| Else - If oldNode key is empty | 1 | N | N |
| Set oldNode corresponding pointers to key, bid, and nullptr | 1 | 1 | 1 |
| Else - while loop oldNode next pointer is not nullptr | 1 | N | N |
| Advance oldNode | 1 | 1 | 1 |
| Set oldNode’s next pointer to newly created Node | 1 | 1 | 1 |
| **Total Cost** | | | **4N+6** |
| **Runtime** | | | **O(N)** |

| **searchCourse Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create key variable and set it to the hash function call | 1 | 1 | 1 |
| Create node pointer variable and set to address of node key | 1 | 1 | 1 |
| If node is not nullptr+bucket is not empty+course number matches | 1 | N | N |
| Print course information with necessary formatting | 1 | N | N |
| If node equals nullptr OR bucket is empty | 1 | N | N |
| Print “course number not found” | 1 | 1 | 1 |
| While loop node is not null | 1 | N | N |
| If node course number matches passed courseNumber | 1 | N | N |
| Print course information with necessary formating | 1 | 1 | 1 |
| Advance node | 1 | 1 | 1 |
| **Total Cost** | | | **5N+5** |
| **Runtime** | | | **O(N)** |

| **printCourse Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For loop interating through hash table | 1 | N | N |
| If statement to check if bucket is empty | 1 | N | N |
| Display bid with formatting | 1 | 1 | 1 |
| Node pointer is equal to next iteration | 1 | 1 | 1 |
| While loop to check chained bids in bucket | 1 | N | N |
| Display bid node from linked list with formatting | 1 | 1 | 1 |
| Assign node is equal to next node | 1 | 1 | 1 |
| **Total Cost** | | | **3N+4** |
| **Runtime** | | | **O(N)** |

**Tree Data Structure – O(logN)**

| **loadCourses Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| While loop to read entire line until the end of file | 1 | N | N |
| Create Course struct for line read | 1 | 1 | 1 |
| Call getline function passing ifstream variable and string variable | 1 | N | N |
| Pass read line to stringstream variable | 1 | 1 | 1 |
| While loop Calling getline function | 1 | N | N |
| Push back string variables to string vector | 1 | N | N |
| Assign 0 element to Course struct courseNumber | 1 | 1 | 1 |
| Assign 1 element to Course struct courseName | 1 | 1 | 1 |
| If statement – string vector has less than 2 elements | 1 | N | N |
| Display error message | 1 | 1 | 1 |
| Else - If statement – string vector has more than 2 elements | 1 | N | N |
| While loop starting at 3 element until size of vector | 1 | N | N |
| Push the ith element to the Course struct coursePrereq | 1 | N | N |
| push this course to the end with call to insert function passing course | 1 | N | N |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | **9N+10** |
| **Runtime** | | | **O(N)** |

| **insert Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If statement to check if tree is empty | 1 | N | N |
| Sets root as new course node | 1 | 1 | 1 |
| Else - Tree is not empty | 1 | N | N |
| Call addNode function with THIS pointer and pass course to root | 1 | N | N |
| **Total Cost** | | | **3N+1** |
| **Runtime** | | | **O(N)** |

| **addNode Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If statement – Call compare function to compare courseNumbers | 1 | N | N |
| If statement to check if parent has a left child | 1 | N | N |
| Sets left child to new bid Node | 1 | 1 | 1 |
| Else - Parent has a left child | 1 | N | N |
| Recursive call to addNode to traverse down left child | 1 | logN | logN |
| Else - Node courseNumber is smaller | 1 | N | N |
| If statement to check if parent has a right child | 1 | N | N |
| Sets right child to new bid Node | 1 | 1 | 1 |
| Else - Parent has a right child | 1 | N | N |
| Recursive call to addNode to traverse down right child | 1 | logN | logN |
| **Total Cost** | | | **2logN+6N+8** |
| **Runtime** | | | **O(logN)** |

| **printCourse Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If statement node is not equal to null ptr | 1 | N | N |
| Recursive call to inOrder for the left child | 1 | logN | logN |
| Display course information with necessary formatting | 1 | 1 | 1 |
| Recursive call to inOrder for the right child | 1 | logN | logN |
| **Total Cost** | | | **2logN+N+2** |
| **Runtime** | | | **O(logN)** |

| **searchCourse Function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create/set current node variable equal to root of tree | 1 | 1 | 1 |
| While loop to traverse tree - curNode will be null once it reaches the bottom | 1 | N | N |
| If statement to compare course numbers with current node | 1 | N | N |
| Print course information with necessary formatting | 1 | 1 | 1 |
| Return | 1 | 1 | 1 |
| Else if statement to compare course numbers with current node | 1 | N | N |
| Set current node to current node’s left child for traversal | 1 | 1 | 1 |
| Else – course number is larger than current Node | 1 | N | N |
| Set current node to current node’s right child for traversal | 1 | 1 | 1 |
| Print “Course number not found” | 1 | 1 | 1 |
| Return | 1 | 1 | 1 |
| **Total Cost** | | | **4N+7** |
| **Runtime** | | | **O(N)** |

## Advantages and Disadvantages

Vectors are simple to implement and are easy to work with. Vectors are dynamic and use functions like push and pop to effortlessly add and remove elements. They can also be initialized and resized in a single line of code. Because vector indexes are expressions, programs can easily lookup and alter the Nth item in a list. This coupled with their size function makes vectors work well with loops and iterations. What holds vectors back is their inefficiency when sorting their values and searching within the vector. Because the data is essentially held in the vector as a list, operations may be applied to each element given the scenario. Sorting/searching a vector is the least efficient in terms of run-time of the three data structures analyzed.

Hash Tables have the potential to search in constant time. Hash tables store unordered items by mapping each item to a location in an array called a bucket. Hash tables use keys to map items to indexes. Ideally, every key is unique to the items stored in the hash table allowing searches in constant time. Although, this is not always the case. Collisions can occur when the algorithm assigning item keys maps two or more items to the same bucket. This can be handled in several different ways but will affect the run-time efficiency of the data structure and are practically unavoidable with large data sets. Hash tables insertion algorithms can be complicated and confusing if a constant run-time is to be maintained. Since hash tables assign keys to items to be stored in the hash table, it is difficult to sort the data is a specific order once it is assigned to bucket. It would be better to have the data already sorted before inserting it into a hash table. Hash tables are the least memory efficient of the three data structures.

Binary trees allow for quick data insertion and searching. Binary trees consist of link list nodes that have up to two children: a left child and/or right child. Within a Binary Search Tree (BST), the assignment of the left and right child is based on their values relative to their parent. The left child is less than the parent and the right child is greater than the parent. Because of this ordering property, data is sorted once it is assigned. BST’s have an at worst run-time of O(logn). This is due to the paths created when the data was inserted onto the BST. These paths cut down the number of items to be compared allowing efficient insertion, deletion, and searching. BSTs’ biggest issue deals with height and insertion of data. Height is the maximum number of edges from the root of the tree to any leaf. When inserting randomly ordered data, the tree heigh stay near the minimum of the tree. If data is inserted already sorted, the height of the tree yields reaches its max and leaves the tree unbalanced. At this point the tree becomes a single link list because no other paths are created.

## Recommendation

After analyzing the three data structures and factoring in the requirements of the Computer Science department at ABCU, I have determined that the best data structure to use is a Binary Search Tree (BST). A vector would be the easiest to code but would not be efficient in terms of run-time when handling large amounts of data. A hash table seemed like a good idea, but given the use case of courses at the university, it is possible for multiple collisions due to the structure of the course Ids i.e. CSCI300, CSCI100, CSCI400, etc. Also, sorting the courses would prove to be more complicated without much benefit. The Binary Search Tree still allows for efficient searching of course Ids and will have the courses sorted on insertion to the BST.