**Project One – Sylvia Trynkin**

1. **Pseudocode**
2. **Vector Data Structure 2**
3. **Hash Table Data Structure 8**
4. **Binary Search Tree Data Structure 13**
5. **Efficiency Analysis**
6. **Vector Data Structure 20**
7. **Hash Table Data Structure 21**
8. **Binary Search Tree Data Structure 22**
9. **Advantages**
10. **Vector Data Structure 23**
11. **Hash Table Data Structure 24**
12. **Binary Search Tree Data Structure 24**

**4. Disadvantages**

**a. Vector Data Structure 24**

**b. Hash Table Data Structure 24**

**c. Binary Search Tree Data Structure 24**

**5. Recommendation 24**

1. **References 25**

**1a. Pseudocode Vector Data Structure:**

*//Course object*

Vector<Course\*>CourseInformation(vector<string>details){

Initialize course vector<Course\*>

Initialize string token; //stores each value in line in a separate token

Initialize int count; //counts the number of tokens

Loop through file

For each new course{

Count equals 0;

While end of the line is not reached{

For every line in file{

If count equals 0{

Assign courseNumber to token;

Pushback to add into vector;

Increment count by 1;

}

If count equals 1{

Assign courseName to token;

Pushback to add into vector;

Increment by 1;

}

If count equals >= 2{

if coursenumber exists as first value in another line{

Assign PreReq to token;

Pushback to add into vector;

Display “Prerequisite required!”; Increment count by 1;

}

}

}

}

Return courseInformation;

*// Vector pseudocode*

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites{

add prerequisites of p to totalPrerequisites

}

print number of totalPrerequisites

}

*//searching for course in Data structure*

int courseSearch(vector<Course>courses, string courseNum) {

Get userInput //get course Number from user

for all Course in courses { //loop through vector

if userInput equals courseNum {

display course information;

for each PreReq of this course {

display PreReq course information;

}

}

}

return;

}

*//Open and read file*

initialize fstream to open and read file;

open file with instream; // use the filename to do so

if file is not found {

display “File not found!”;

}

//file is found

Else{

While end of line is not reached{

Read every line

If #values in per line < 2{

Display “File format Error!”

}

Else {

Read line;

}

//other values besides course number and course name (like PreReq courses)

If #values per line > 2 {

//PreReq course name or number is course name or number in another line

If this value equals the first value in another course;

Continue;

}

Else{

Display “File format Error!”

}

}

Close file

*//print course information*

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

get UserInput course//get user input course number

for all courses {

if the course is the same as courseNumber {

display the course information;

for each prerequisite of the course {

display the prerequisite course information

}

}

else{

Display “course not found!”;

}

}

*//menu*

Set userInput to 0;

Display menu options (1. Load Data Structure, 2. Print Course List, 3. Print Course, 4. Exit);

Get userInput;

While(userInput is not equal 4){

If(userInput = 1){

Load file data into data structure;

Open file;

Display content;

}

Else if(userInput = 2){

Display list of all courses in alphanumerical order;

}

Else if(userInput = 3){

Display course title and prerequisites for any individual course;

}

If (userInput = 4){

Exit program;

}

*//\*\*\*\*\*print list in alphanumerical order\*\*\*\*\**

*//create partition method*

set lowIndex to first item(1st course)

set highIndex to last item(last course)

choose the middle element as pivot:

midpoint = lowIndex + (highIndex - lowIndex)/2

set pivot to midpoint

while (!done){

while(lowIndex < pivot){

increment lowIndex;

}

while(pivot < highIndex){

decrement highIndex;

}

if(lowIndex >= highIndex){

return highIndex (done = true);

}

else{

swap lowIndex and highIndex;

set temp = lowIndex;

set lowIndex = highIndex;

set highIndex = temp;

update lowIndex += 1;

highIndex -= 1;

}

return highIndex

}

*//Create quicksort method*

Initialize integer i;

If(lowIndex >= highIndex){

Return;

}

Set lowEndIndex equal to partition(courses, lowIndex, highIndex);

Recursively sort low partition Quicksort(courses, lowIndex, lowEndIndex) and high partition Quicksort(courses, lowEndIndex +1, highIndex);

*//create print method for sorted courses*

Void printSortedCourses(Course, cours){

Loop:

For(i = 0; i < course.size(); i++){

printSortedCourses(course[i]);

}

**1b. Pseudocode Hash Table Data Structure**

*//Open and read file*

initialize fstream to open and read file;

open file with instream; // use the filename to do so

if file is not found {

display “File not found!”;

}

//file is found

Else{

While end of line is not reached{

Read every line

If #values in per line < 2{

Display “File format Error!”

}

Else {

Read line;

}

//other values besides course number and course name (like PreReq courses)

If #values per line > 2 {

//PreReq course name or number is course name or number in another line

If this value equals the first value in another course;

Continue;

}

Else{

Display “File format Error!”

}

}

Close file

*//searching for course in Data structure*

int courseSearch(Hashtable<Course>courses, string courseNumber) {

Get userInput //get course Number from user

Assign userInput to key

for all Course in courses { //loop through vector

if key is found {

display course information;

for each PreReq of this course {

display PreReq course information;

}

}

}

return;

}

*//store code in hashtable/insert method*

Create key for the given courseNumber

Check if node is empty

If node is empty{

Insert course;

}

Else (node is not empty){

iterate until empty node is found;

insert course at empty node;

}

*//Hashtable Course object*

Initialize course vector<Node>nodes

Create HashTable class

Initialize string token; //stores each value in line in a separate token

Initialize int count; //counts the number of tokens

Loop through file

For each new course{

Count equals 0;

While end of the line is not reached{

For every line in file{

If count equals 0{

Assign courseNumber to token;

add into hashtable;

Increment count by 1;

}

If count equals 1{

Assign courseName to token;

add into hashtable;

Increment by 1;

}

If count equals >= 2{

if coursenumber exists as first value in another line{

Assign PreReq to token;

add into hashtable;

Display “Prerequisite required!”;

Increment count by 1;

}

}

}

}

Return courseInformation;

*// PreReq courses*

int numPrerequisiteCourses(Hashtable<Course> courses, string courseNumber) {

create key hashing the courseNumber

retrieve node using the key

set to new node variable

while node not equal to null pointer{

iterate until course found in Hashtable or empty node found

if node->course.courseNumber equal to courseNumber{

totalPrerequisites = node.Prerequisites.size();

for each prerequisite p in totalPrerequisites{

add prerequisites of p to totalPrerequisites

}

print number of totalPrerequisites;

}

}

}

*//print course information*

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

get UserInput course//get user input course number

create key by hashing the courseNumber

retrieve node by using key

set to new node variable

while node is not equal null pointer{

iterate until course is found in Hashtable or empty node is found

if node->course.courseNumber is equal to the courseNumber {

display the course information;

for each prerequisite of the course {

display the prerequisite course information

}

}

else{

Display “course not found!”;

}

}

//menu

Set userInput to 0;

Display menu options (1. Load Data Structure, 2. Print Course List, 3. Print Course, 4. Exit);

Get userInput;

While(userInput is not equal 4){

If(userInput = 1){

Load file data into data structure;

Open file;

Display content;

Close file (when done);

}

Else if(userInput = 2){

display list of all courses in alphanumerical order

}

Else if(userInput = 3){

Call printCourseInformation (displays course title and prerequisites for any individual courses);

}

If (userInput = 4){

Exit program;

}

}

*//print courses in alphabetical order*

*//According to my research it is not possible to sort a Hashtable as the data is stored in keys and not indexes.*

*//If we want to sort the data in the Hashtable, a sortable object is needed, for example an array or arrayList.*

create method to take Hashtable as parameter:

public void BuildTable(Hashtable coursesHashtable){

get count of hashtable;

create array on basis of value (as we want to sort the courses):

initiate int i and assign to 0;

for(each entry in hashtable){

coursesortedvalue[i] = parseCourse(de.Value.ToString());

increment i;

}

}

call Sort method to sort the array{

loop through all items in array{

display array;

}

}

**1c. Pseudocode Binary Search Tree Data Structure**

*//Open and read file*

initialize fstream to open and read file;

open file with instream; // use the filename to do so

if file is not found {

display “File not found!”;

}

//file is found

Else{

While end of line is not reached{

Read every line

If #values in per line < 2{

Display “File format Error!”

}

Else {

Read line;

}

//other values besides course number and course name (like PreReq courses)

If #values per line > 2 {

//PreReq course name or number is course name or number in another line

If this value equals the first value in another course;

Continue;

}

Else{

Display “File format Error!”

}

}

Close file

*//searching for course in Data structure*

Get userInput

userInput equals the courseNumber we are searching

assign current node with null

while current node != null{

if (current node is matching the courseNumber){

return node;

}

else if courseNumber is less{

assign current node to left;

}

else if courseNumber is more{

assign current node to the right;

}

else if matching courseNumber is not found{

return null;

}

*//store code in Tree/insert method*

If (root equals null){

Assign current course to root;

Assign left child to null;

Assign right child to null;

}

Else if (courseNumber is less than root && left child = null){

Assign left child with courseNumber;

}

Else if (courseNumber is greater than root && right child =null){

Assign right child with courseNumber;

}

*//Tree Course object*

Initialize course vector<Node>nodes

CreateTree class

Initialize string token; //stores each value in line in a separate token

Initialize int count; //counts the number of tokens

Loop through file

For each new course{

Count equals 0;

While end of the line is not reached{

For every line in file{

If count equals 0{

Assign courseNumber to token;

add into Tree:

if courseNumber is less than leaf{

add as left child;

}

if courseNumber is greater than leaf{

add as right child;

}

Increment count by 1;

}

If count equals 1{

Assign courseName to token;

add into Tree:

if courseName is less than leaf{

add as left child;

}

if courseName is greater than leaf{

add as right child;

}

Increment by 1;

}

If count equals >= 2{

if coursenumber exists as first value in another line{

Assign PreReq to token;

add into Tree:

if PreReq is less than leaf{

add as left child;

}

if PreReq is greater than leaf{

add as right child;

}

Display “Prerequisite required!”;

Increment count by 1;

}

}

}

}

Return courseInformation;

*// PreReq courses*

int numPrerequisiteCourses(Tree<Course> courses, string courseNumber) {

create key hashing the courseNumber

retrieve node using the key

set to new node variable

while node not equal to null pointer{

iterate until course found in or empty node found

if node->course.courseNumber equal to courseNumber{

totalPrerequisites = node.Prerequisites.size();

for each prerequisite p in totalPrerequisites{

add prerequisites of p to totalPrerequisites

}

print number of totalPrerequisites;

}

}

}

*//print course information*

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

get UserInput course//get user input course number

while root is not equal null{

traverse left side

if found{

display the course information;

}

Else{

Traverse right;

display the course information;

}

display the course information;

If not found{

Display “course not found!”;

}

}

}

*//menu*

Set userInput to 0;

Display menu options (1. Load Data Structure, 2. Print Course List, 3. Print Course, 4. Exit);

Get userInput;

While(userInput is not equal 4){

If(userInput = 1){

Load file data into data structure;

Open file;

Display content;

Close file (when done);

}

Else if(userInput = 2){

Call inOrder method (displays list of all courses in alphanumerical order);

}

Else if(userInput = 3){

Call printCourseInformation (displays course title and prerequisites for any individual courses);

}

If (userInput = 4){

Exit program;

}

*//\*\*\*\*print courses in alphanimerical order\*\*\*\**

*//create an inOrder method and print*

If(node equals null){

Return;

}

If(node does not equal null){

//Check left node

inOrder(node->left);

display course information (courseNumber, courseName, Prerequisites);

//next check right node

inOrder(node->right);

display course information (courseNumber, courseName, Prerequisites);

}

**2a. Efficiency Analysis Vector Data Structure**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Vector Data Structure*** | Line cost: | how many times executed: | total cost: |
| open and read file: | 0 | 0 | 0 |
| initialize fstream | 1 | 1 | 1 |
| open file with instream | 1 | 1 | 1 |
| if file is not found | 1 | 1 | 1 |
| display “File not found!” | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| While end of line is not reached | 1 | n | n |
| Read every line | 1 | n | n |
| If #values in per line < 2 | 1 | n | n |
| Display “File format Error!” | 1 | n | n |
| Else | 1 | n | n |
| Read line | 1 | n | n |
| If #values per line > 2 | 1 | n | n |
| If this value equals the first value in another course; | 1 | n | n |
| Continue; | 1 | n | n |
| Else | 1 | n | n |
| Display “File format Error!” | 1 | n | n |
| Close file | 1 | 1 | 1 |
|  |  |  |  |
| create vector<Course\*>CourseInformtaion(vector<string>details) | 1 | 1 | 1 |
| initialize course vector <Course\*> | 1 | 1 | 1 |
| initialize string token | 1 | 1 | 1 |
| initialize int count | 1 | 1 | 1 |
| for each new course | 1 | n | n |
| set count = 0 | 1 | n | n |
| while end of line not reached | 1 | n | n |
| for every line in file | 1 | n\*n | n\*n |
| if count = 0 | 1 | 1 | 1 |
| assign courseNumber to token | 1 | 1 | 1 |
| pushback to add into vector | 1 | 1 | 1 |
| increment count by 1 | 1 | 1 | 1 |
| if count = 1 | 1 | 1 | 1 |
| assign courseName to token | 1 | 1 | 1 |
| pushback to add into vector | 1 | 1 | 1 |
| incement by 1 | 1 | 1 | 1 |
| if count >= 2 | 1 | n | n |
| if courseNumber exists as first value in another line | 1 | n | n |
| assign PreReq to token | 1 | n | n |
| pushback to add into vector | 1 | n | n |
| display "Prerequisite required" | 1 | n | n |
| increment count by 1 | 1 | n | n |
| return courseinformation | 1 | 1 | 1 |
| total cost: |  |  | 21n\*n+ 19 |
| runtime: |  |  | O(n²) |

**2b. Efficiency Analysis Hash Table Data Structure**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Hash Table Data Structure*** | Line cost: | how many times executed: | total cost: |
| open and read file: | 0 | 0 | 0 |
| initialize fstream | 1 | 1 | 1 |
| open file with instream | 1 | 1 | 1 |
| if file is not found | 1 | 1 | 1 |
| display “File not found!” | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| While end of line is not reached | 1 | n | n |
| Read every line | 1 | n | n |
| If #values in per line < 2 | 1 | n | n |
| Display “File format Error!” | 1 | n | n |
| Else | 1 | 1 | 1 |
| Read line | 1 | 1 | 1 |
| If #values per line > 2 | 1 | n | n |
| If this value equals the first value in another course; | 1 | n | n |
| Continue; | 1 | n | n |
| Else | 1 | 1 | 1 |
| Display “File format Error!” | 1 | 1 | 1 |
| Close file | 1 | 1 | 1 |
|  |  |  |  |
| create HashTable class | 1 | 1 | 1 |
| insert method: | 0 | 0 | 0 |
| create key for given courseNumber | 1 | 1 | 1 |
| check if node is empty | 1 | 1 | 1 |
| if node is empty | 1 | 1 | 1 |
| insert course | 1 | 1 | 1 |
| else if node is not empty | 1 | n | n |
| iterate until empty node is found | 1 | n | n |
| insert course at empty node | 1 | 1 | 1 |
| hashtable course object: | 0 | 0 | 0 |
| initialize string token | 1 | 1 | 1 |
| initialize int count | 1 | 1 | 1 |
| for each new course | 1 | n | n |
| set count = 0 | 1 | 1 | 1 |
| while end of line not reached | 1 | n | n |
| for every line in file | 1 | n\*n | n\*n |
| if count = 0 | 1 | 1 | 1 |
| assign courseNumber to token | 1 | 1 | 1 |
| add into hashtable | 1 | 1 | 1 |
| increment count by 1 | 1 | 1 | 1 |
| if count = 1 | 1 | 1 | 1 |
| assign courseName to token | 1 | 1 | 1 |
| add into hashtable | 1 | 1 | 1 |
| incement by 1 | 1 | 1 | 1 |
| if count >= 2 | 1 | n | n |
| if courseNumber exists as first value in another line | 1 | n | n |
| assign PreReq to token | 1 | n | n |
| add into hashtable | 1 | n | n |
| display "Prerequisite required" | 1 | n | n |
| increment count by 1 | 1 | n | n |
| return courseinformation | 1 | 1 | 1 |
| total cost: |  |  | 28 + 18n\*n |
| runtime: |  |  | O(n²) |

**2c. Efficiency Analysis Binary Search Tree Data Structure**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Binary Search Tree Data Structure*** | Line cost: | how many times executed: | total cost: |
| open and read file: |  |  |  |
| initialize fstream | 1 | 1 | 1 |
| open file with instream | 1 | 1 | 1 |
| if file is not found | 1 | 1 | 1 |
| display “File not found!” | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| While end of line is not reached | 1 | n | n |
| Read every line | 1 | n | n |
| If #values in per line < 2 | 1 | n | n |
| Display “File format Error!” | 1 | n | n |
| Else | 1 | 1 | 1 |
| Read line | 1 | 1 | 1 |
| If #values per line > 2 | 1 | n | n |
| If this value equals the first value in another course; | 1 | n | n |
| Continue; | 1 | n | n |
| Else | 1 | 1 | 1 |
| Display “File format Error!” | 1 | 1 | 1 |
| Close file | 1 | 1 | 1 |
|  |  |  |  |
| create Tree class | 1 | 1 | 1 |
| insert method: | 0 | 0 | 0 |
| if root = null | 1 | 1 | 1 |
| assign current course to root | 1 | 1 | 1 |
| assign left child to null | 1 | 1 | 1 |
| assign right child to null | 1 | 1 | 1 |
| elseif courseNumber < root && left child = null | 1 | n | n |
| assign left child with courseNumber | 1 | n | n |
| elseif courseNumber > root && right child = null | 1 | n | n |
| assign right child with courseNumber | 1 | n | n |
| Tree course object: | 0 | 0 | 0 |
| initialize string token | 1 | 1 | 1 |
| initialize int count | 1 | 1 | 1 |
| for each new course | 1 | n | n |
| set count = 0 | 1 | 1 | 1 |
| while end of line not reached | 1 | n | n |
| for every line in file | 1 | n\*n | n\*n |
| if count = 0 | 1 | 1 | 1 |
| assign courseNumber to token | 1 | 1 | 1 |
| add into tree: | 0 | 0 | 0 |
| if CourseNumber < leaf | 1 | 1 | 1 |
| add as left child | 1 | 1 | 1 |
| if courseNumber > leaf | 1 | 1 | 1 |
| add as right child | 1 | 1 | 1 |
| increment count by 1 | 1 | 1 | 1 |
| if count = 1 | 1 | 1 | 1 |
| assign courseName to token | 1 | 1 | 1 |
| add into tree: | 0 | 0 | 0 |
| if CourseNumber < leaf | 1 | 1 | 1 |
| add as left child | 1 | 1 | 1 |
| if courseNumber > leaf | 1 | 1 | 1 |
| add as right child | 1 | 1 | 1 |
| incement by 1 | 1 | 1 | 1 |
| if count >= 2 | 1 | 1 | 1 |
| if courseNumber exists as first value in another line | 1 | 1 | 1 |
| assign PreReq to token | 1 | 1 | 1 |
| add into tree: | 0 | 0 | 0 |
| if PreReq < leaf | 1 | 1 | 1 |
| add as left child | 1 | 1 | 1 |
| if PreReq > leaf | 1 | 1 | 1 |
| add as right child | 1 | 1 | 1 |
| display "Prerequisite required" | 1 | 1 | 1 |
| increment count by 1 | 1 | 1 | 1 |
| return courseinformation | 1 | 1 | 1 |
| totel cost: |  |  | 42 + 14n\*n |
| runtime: |  |  | O(n²) |

**3a. Advantage Vector Data Structure:**

Vectors are easy to implement and maintain because of their simple structure and can be adjusted in their size according to the user’s needs.

**3b. Advantage Hash Table Data Structure:**

As Hash tables stores data in buckets, it is faster and easier to search with the use of keys with the function of an index. This saves a lot of time especially with large data sets, as not every item needs to be examined in order to find the correct item as long as you know the key. When several items are supposed to go in the same bucket there are different ways to handle those situations. Those so called “collisions” can be handled with a linked list chain, linear probing or quadratic probing. Another advantage is that inserting, searching and deleting can be done in constant time.

**3c. Advantage Binary Search Tree Data Structure:**

Binary Search Trees are easy to visualize and show relationships between different kinds of data. They are very helpful in routing systems and whenever you sort by “more than” and “less than” method (for example sorting in alphabetical order). In our project we use this method to parse through the lines for our insert method to make sure courseNumber and courseName are stored correctly and if PreReq courses are required. A BST can also scale automatically and has a good time complexity.

**4a. Disadvantage Vector Data Structure**

One of the disadvantages of the vector data structure is that it is linear and when an item is searched, this is done sequentially which takes more time. Therefore, it is not suitable for big datasets. Also, operations like inserting or removing an item in a vector is more cumbersome than with the other two methods.

**4b. Disadvantage Hash Table Data Structure**

Hash Tables in general are very useful but there are some scenarios where hash tables are not the first choice. For example, when the project requires that only one item is located in every bin. This can be very complex if the data set to be used is very large. That would require a big hash table with many bins and therefore high memory usage. Another issue is, “if we want to find the **predecessor or successor** of a node in a hash table, we have to maintain a parent of every node and then traverse to all those nodes one by one which will take more time”. Furthermore, hash tables have “lesser security in smaller architectures as the hash function is susceptible to manipulation”. (Kumar, 2020)

**4c. Disadvantage Binary Search Tree**

The major disadvantage especially for large BST is the time factor, especially for searching and deleting as it can be very complex if the tree is large. Depending on the Data set to be used it might not be suitable to use a BST as data needs to be sorted in the tree which again adds to the time necessary.

**5. Recommendation**

As easy it would be to use a vector data structure it is just not suitable for our project. It would be too slow to process. Especially when considering that this data structure is part of the Universities data base and many students and potential students are accessing it, it must be reliably fast.

Even though a hash table is suitable for large data sets and operations like searching, inserting and deleting courses can be done in constant time, I would recommend against it because with a hash table we would run into the issue of collision which would need to be handled. That would increase time and memory and decrease attractivity for our project.

A binary search tree, especially as our pool of offered courses is not too big, is suitable for our project. We have to link the courses in a way to show relationship with the required PreReq courses, which could be achieved with a BST. As already mentioned in the advantages it can scale automatically and has a good time complexity.

After comparing the Big-O analysis and runtime all 3 data structures for the required part of the program have a Big-O of O(n²) as we can ignore the constants. But when having a closer look at the individual total costs, the BST requires less time than the other two data structures (for the analyzed part of the program at least). We also know that binary search results in O(log n) which would be sufficiently quick. Finally, knowing that we have to create a relationship between the courses strengthens my recommendation.

**6. References**

Kumar, M. (2020, November 13). Advantages of Binary Search Tree over Hash Table. CodingNinjas.

<https://www.codingninjas.com/blog/2020/11/13/advantages-of-binary-search-tree-over-hash-table/>

Vahid, F., Lysecky, S., Wheatland, N., Siu, R., Lysecky, R., Edgcomb, A., & Yuen, J.(2019, February). CS 300: Data Structures and Algorthims. Zybook. Retireved October 04, 2022

<https://learn.zybooks.com/zybook/CS-300-H7431-OL-TRAD-UG.22EW1/chapter/3/section/18>

<https://learn.zybooks.com/zybook/CS-300-H7431-OL-TRAD-UG.22EW1/chapter/3/section/7>

Prasoon (2022, Aug 24). Hashtable Sorting. C-Sharpcorner.

<https://www.c-sharpcorner.com/uploadfile/presoonk/hashtable-sorting/>