Kaitlyn Lush

CS 300

Project One

**Vector Pseudocode:**

*(Read File)*

BEGIN

FUNCTION readFile

OPEN file

IF file opened

SET readFile = open file

IF readFile cannot be read

OUTPUT “Error”

EXIT

CREATE vector courseInfo

CREATE string courseRequirements

WHILE not at the end of file

READ file by parsing each line

GETLINE from file into courseReq

WHILE not at the end of file

IF courseReq has less than 2 elements

OUTPUT “incorrect format”

ELSE

ADD courseReq into courseInfo

ELSE

OUTPUT “Could not open file”

EXIT

END

*(Create course object and store)*

CREATE course class

DECLARE string courseNum

DECLARE string courseName

DECLARE vector preReq string type

CREATE course vector

OPEN file to read

WHILE not at the end of the file

SET courseNum = first data value

SET courseName = second data value

APPEND preReq vector with courseNum data value

STORE into course vector

CLOSE file

END

*(Print course info and prerequisites)*

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

FOR each course in courses

IF courseNum = courseNum parameter

PRINT courseNum and courseName

FOR each preReq of the course

PRINT preReq’s courseNum

END

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times executed** | **Total Cost** |
| Create vector | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| Create vector | 1 | 1 | 1 |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Push back course item | 1 | n | n |
|  |  | Total Cost | 5n+1 |
|  |  |  | O(n) |

**Hash Table:**

*(Read File)*

BEGIN

FUNCTION readFile

OPEN file

IF file opened

SET readFile = open file

IF readFile cannot be read

OUTPUT “Error”

EXIT

CREATE vector courseInfo

CREATE string courseRequirements

WHILE not at the end of file

READ file by parsing each line

GETLINE from file into courseReq

WHILE not at the end of file

IF courseReq has less than 2 elements

OUTPUT “incorrect format”

ELSE

ADD courseReq into courseInfo

ELSE

OUTPUT “Could not open file”

EXIT

END

*(Create course object and store)*

START

DECLARE string courseNum

DECLARE string courseName

DECLARE vector for string prerequisites

CREATE hashTable class

CREATE Node structure

DECLARE Course course

DECLARE unsigned int key

DECLARE Node next pointer

DECLARE Node Constructor

SET key to UNIT\_MAX

SET next pointer to nullptr

DEFINE Node Constructor w course parameters

DEFINE Node Constructor w course, unsigned int parameters course, key

DECLARE vector of Node nodes

DECLARE unsigned in size = 101

DECLARE unsigned int CALL hash pass key

DECLARE HashTable

DECLARE HashTable with Unsigned int size

DEFINE HashTable

RESIZE nodes vector to size

ERASE nodes vector from beginning

CREATE insert method HashTable::(Course course)

Load Courses (string csvPath, Hashtable)

Loop to read CSV file rows

DECLARE string hasNum

DECLARE int temp = 0

FOR int i = 0 while I < courseNum size INCREMENT i

IF i < 4

SET temp to CAST INT of courseNum at i

APPEND to string temp to hashNum

ELSE

APPEND to string courseNum at i

DECLARE Signed key set to temp using c\_str() func

DECLARE node set to the node at key

IF node not nullptr

DECLARE newNode set to new node passing parameters course and key

INSERT newNode to node at beginning plus key

ELSE

IF node key = UNIT\_MAX

SET node key to key

SET node course to course

SET node next pointer to nullptr

ELSE

WHILE node next pointer not nullptr

SET node to next pointer

SET node next pointer to new node passing parameters course and key

END

*(Print course info and prerequisites)*

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

FOR each course in courses

IF courseNum = courseNum parameter

PRINT courseNum and courseName

FOR each preReq of the course

PRINT preReq’s courseNum

END

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table** | **Line Cost** | **# Times executed** | **Total Cost** |
| Create hash table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| Create key for course | 1 | n | n |
| If no entry found for key | 1 | n | n |
| Assign node to key | 1 | n | n |
| Else | 1 | n | n |
| assign old node to UNIT\_MAX set to key, set old not to course and next to null pointer | 4 | n | 4n |
| Else | 1 | n | n |
| Find next open node | 1 | n | n |
| Add new node to end | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Insert course item | 1 | n | n |
|  |  | Total Cost | 16n+1 |
|  |  |  | O(n) |

**Binary Search Tree:**

*(Read File)*

BEGIN

FUNCTION readFile

OPEN file

IF file opened

SET readFile = open file

IF readFile cannot be read

OUTPUT “Error”

EXIT

CREATE vector courseInfo

CREATE string courseRequirements

WHILE not at the end of file

READ file by parsing each line

GETLINE from file into courseReq

WHILE not at the end of file

IF courseReq has less than 2 elements

OUTPUT “incorrect format”

ELSE

ADD courseReq into courseInfo

ELSE

OUTPUT “Could not open file”

EXIT

END

*(Create course object and store)*

START

DEFINE Course struct

DECLARE string courseNum

DECLARE string courseName

DECLARE vector for string prerequisites

DECLARE Course

DEFINE Node struct

DECLARE Node left

DECLARE Node right

DECLARE Node constructor

DEFINE BinarySearchTree class

DECLARE Node root

DECLARE addNode

DECLARE BinarySearchTree constructor

DECLARE BinarySearchTree destructor

SET root to nullptr

CREATE insert method BinarySearchTree::addNode(Node node, Course course)

IF root = null

THEN current course = root

ELSE IF course number is less than root

ADD course to left side of tree

IF left side = null

ADD course number

ELSE IF course number is less than leaf

ADD to the left

ELSE IF course number is greater than leaf

ADD to the right

ELSE IF course number is greater than root

ADD course to right side of tree

IF right side = null

ADD course number

ELSE IF course number is less than leaf

ADD to the left

ELSE IF course number is greater than leaf

ADD to the right

END

*(Print course info and prerequisites)*

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

SET current node = root

GET user INPUT

WHILE current course number does not equal null

IF current course number = input

PRINT course information

ELSE IF current course number is less than 0

TRAVERSE left

ELSE IF current course number is greater than 0

TRAVERSE right

LOOP until found

ELSE

PRINT “Course not found”

END

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Tree** | **Line cost** | **# Times Executed** | **Total Cost** |
| Create tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| If root is null add root | 1 | 1 | 1 |
| If node < root add left | 1 | n | n |
| if no left node | 1 | n | n |
| this node becomes left | 1 | n | n |
| if node > root add right | 1 | n | n |
| if no right node | 1 | n | n |
| this node becomes right | 1 | n | n |
| For each line in file | 1 | n | n |
| create vector course item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Insert course item | 1 | n | n |
|  |  | Total Cost | 11n+2 |
|  |  |  | O(n) |

**Menu:**

START

CREATE while loop for menu

WHILE choice is not equal to 4

OUTPUT “Select Menu Option: “

“1. Load Data Structure”

“2. Print Course List”

“3. Print Individual course”

“4. Exit”

OUTPUT “Enter menu choice: “

DECLARE switch

CREATE case for each input:

CASE 1

CALL load function

CASE 2

CALL Print sorted list function

CASE 3

CALL Print course information

CASE 4

BREAK

**Print Sorted List:**

CREATE sorted print method

CREATE partition method int partition(vector<Course>&courses, int begin, int end)

SET lowIndex to first element

SET highIndex to last element

SET midpoint to pivot

WHILE pivot is less then highIndex

DECREMENT highIndex

SWAP lower values left of pivot

SWAP higher values right of pivot

SET temp value to low index

SET low index to high index

SET high index to temp

CREATE quicksort method quicksort(vector<Course>&courses, int begin, int end)

SET mid to 0

SET lowIndex to begin

SET highIndex to end

IF being >= end

RETURN

SET lowEndIndex to partition(courses, lowIndex,highIndex)

Make CALL to quicksort

Quicksort(courses, lowIndex, lowEndIndex)

Quicksort(courses, lowEndIndex +1, highIndex)

CREATE display course method

OUTPUT course.courseId, “:”, course.name, “ | ”, course.prereq, endl

LOOP through vector to get courses

Each data structure that we have looked at comes with benefits and drawbacks. The vector data structure has advantages to its use as well as disadvantages. One advantage to using the vector data structure is how fast it can read the file as well as add the course objects. It is a fairly simple structure where each line of the file is parsed and each item from each line is then placed at the end of the vector. This method showed the shortest runtime between the three data structures at 5n + 1. A vector would be the simplest data structure to implement as well as the most efficient to quickly add items to the list. The disadvantage that comes with a vector would be the search factor. Using a vector with a long list will slow the run time down since it has to search through every item on the list.

A hash table has the advantage of being quicker than a vector when it comes to the search function. However, hash tables are more complicated and harder to implement. Since hash tables rely on a key to search through the list it allows the program to search through the list of items a lot quicker than with a vector. A hash table will also be slower when implementing an initial list as a key has to be created for each item within the list. The hash table would be more complex to program and if one mistake is made within the program it could cause problems with the outputs of the program.

A binary tree gives the advantage of being quicker than a vector as well. The search function for a binary tree would be quicker on a long list of items than a vector would be. The binary tree runtime for the search function directly coincides with the height of the tree. The binary tree search factor would go between the vector and the hash table as its quicker than the vector but not as quick as a hash table. It does have the disadvantage of being slower than a hash table as it has to go down the tree branches to find the correct item.

Based on my evaluations I would recommend going with a binary tree. I think that the binary tree provides a happy medium between the vector and the hash table. It allows for a quicker search method than a vector would. However if implemented correctly it would function fairly quick and would meet the necessary requirements.