Seth Hamrick

Project 1

ABCU Pseudocode/Analysis

10/07/2022

**Course Object**

Class Course {

Declare public string, courseNum

Declare public string, title

Declare private empty Vector of strings, prereqs

Declare public method AddPrereq

}

Create default constructor for Course class{

Set default courseNumber to zero

Set default title to zero

}

int Course::AddPrereq(string t\_prereq) {

FOR each Course in courseList

IF Course matches t\_prereq

THEN add t\_prereq to Course vector prereqs

RETURN 1;

IF the end of the courseList is reached without finding a match

THEN RETURN 2;

}

Vector<string> Course::GetPrereq() {

RETURN prereqs

}

**Loading File**

void LoadFile(string fileName) {

Declare input file stream named, courseFS

Declare string variable named wholeLine (used to parse string)

Declare string variable named couseNum (local variable for each course number)

Declare string variable named courseTitle (local variable for each course title)

Declare empty vector of int type named currentString (used to parse string of “,”)

Open fileName in file stream, courseFS

IF courseFS is not open

THEN output the file was unable to be opened

RETURN 1

Declare Boolean variable named dataToRead set equal to true

While dataToRead is not equal to false

IF courseFS reaches the end of the file

THEN set dataToRead equal to false

BREAK;

Get line from courseFS and store in variable wholeLine

IF get line fails

Output to use unable to reach end of file loading aborted

BREAK;

Declare stringstream named newCourse input wholeLine

FOR each character in string wholeLine

Add each character to vector currentString

IF the character is a comma

Ignore the character

IF currentString size is less than 2

Output to user error loading courses file loading aborted

BREAK;

Create new Course named newCourse

Set newCourse’s courseNumber equal to index zero of vector currentString

Set newCourse’s courseTitle equal to index one of vector currentString

IF currentString size is greater than 2

FOR each string beyond index 1 in currentString

Add string to newCourse’s vector by calling Course method

IF Course method returns 2

THEN Output error loading prerequisites

BREAK;

ELSE

CONTINUE

Add Course to data structure by calling correct method()

Close file in courseFS

**}**

**Menu Display**

void DisplayMenu() {

OUTPUT menu greeting

OUTPUT menu option 1

OUTPUT menu option 2

OUTPUT menu option 3

OUTPUT menu option 4

}

**Vector**

**Adding Course to Vector**

Course may be appended to end of vector with vector library push\_back() function

**Search for Course**

Course SearchCourse(string t\_courseNumber) {

FOR each Course in courseList

IF courseNumber of Course matches t\_courseNumber

THEN RETURN Course

Move to next Course

RETURN empty Course

}

**Outputting Course Schedule**

***Sorting with QuickSort***

int Partition(Vector<Course>& courseList, int begin, int end) {

Declare int variable named low set equal to begin

Declare int variable named high set equal to end

Declare int variable named pivot set equal to the midpoint between begin and end

Declare Boolean variable named done set equal to false

WHILE done is not true

WHILE courseNum at index low within vector courseList is less than courseNum at index pivot within vector vector courseList

Increment low by 1

WHILE courseNum at index pivot within vector courseList is less than courseNum at index high within vector courseList

Decrement high by 1

IF low is greater than or equal to high

THEN set done equal to true

ELSE

Swap Course positions at index positions low and high in vector courseList

Increment low by 1

Decrement high by 1

RETURN high

}

void QuickSort(Vector<Course> courseList, int begin, int end) {

Declare int variable named mid set equal to 0

IF begin is greater than or equal to end

THEN RETURN

Set mid equal to result of partition performed on vector courseList

Recursively sort first part of vector begin to mid by calling QuickSort method

Recursively sort second part of vector mid plus one to end by calling QuickSort

method

}

***Printing Course Schedule via QuickSort Vector***

void PrintSchedule(Vector<Course> courseList) {

Pass courseList, 0, and last index of courseList as parameters in QuickSort() function

FOR all the courses in courseList

OUTPUT Course information

Move to next Course

}

**Hash Table**

**Hash Table Creation**

Class HashTable {

Private:

Define structure named Node {

Declare Course object names course

Declare unsigned int key

Create default constructor for structure Node

Set key equal to UINT\_MAX

Set next pointer equal to null

Create constructor to initialize with Course object

Set course equal to this course

Create constructor to initialize with course and key

Set course equal to this course

Set key equal to this key

Declare empty vector of Nodes named nodes

Declare unsigned int named tableSize set equal to DEFAULT\_SIZE

Declare unsigned int equal to hash function of int key

Declare integer that is result of hash(int key)

Public:

HashTable()

HashTable(unsigned int size)

void Insert(Course course)

Course Search(string courseNum)

void PrintSchedule()

}

HashTable::HashTable() {

Resize vector nodes with tableSize

}

HashTable::HashTable(unsigned int size) {

Set this tableSize equal to size

Resize vector nodes to tableSize

}

Unsigned int HashTable::hash(int key) {

Set key equal to key modulo tableSize

Return key

}

**Inserting Course into Hash Table**

HashTable::Insert(Course course) {

Convert course’s courseNum to integer

Declare new integer named key set equal to hashed courseNum

Declare new Node named newNodeP set equal to new Node with courseNum and key

Declare integer variable named i

Declare new node pointer named currentNode set equal to node at index “key” in vector

Nodes

IF currentNode is null or currentNode’s key is UINT\_MAX

THEN set node at index “key” in vector nodes equal to newNode

ELSE

FOR all of the nodes in vector nodes beginning at index “key +1”

IF this node is null or has key UINT\_MAX

THEN set this node equal to newNode

BREAK;

IF this node’s key is equal to key

THEN OUTPUT hash table is full

Move to next node in vector

}

**Searching for Course**

Course Hashtable::Search(string courseNum) {

Declare new empty Course object

Convert string courseNum store as new integer named numToHash

Declare new integer named key set equal to hashed numToHash

Declare new node pointer named currentNode

Set currentNode equal to the node at index “key” in vector nodes

IF currentNode’s courseNum macthes courseNum being searched

THEN RETURN Course within currentNode

ELSE

FOR all of the nodes in vector nodes beginning at “key”

IF this node’s courseNum matches the courseNum being searched

THEN RETURN this nodes Course

IF this node’s key is equal to key

THEN RETURN empty course object

Move to next node in vector

}

**Printing Course Schedule**

void HashTable::PrintSchedule() {

FOR all the nodes in vector nodes

IF node’s key does not equal UINT\_MAX

THEN OUTPUT node’s Course information

IF node’s pointer is not null

THEN declare new node pointer named nextPrint

WHILE nextPrint is not null

OUPUT nextPrint’s Course information

Set nextPrint equal to next print’s next

}

**Binary Search Tree**

**Binary Search Tree Creation**

Define structure named Node {

Declare Course object named course

Declare Node pointer named left

Declare Node pointer named right

Create default constructor for Node() {

Set left equal to null

Set right equal to null

}

Create constructor to initialize with Course Node(Course t\_course) {

Call default constructor

Set course equal to t\_course

}

}

Class BinarySearchTree {

Private:

Declare Node pointer named root

Void addNode(Node\*, Course)

Node\* removeNode(Node\*, string courseNum)

Void inOrder(Node\*)

Public:

BinarySearchTree()

Void PrintInOrder()

Void Insert(Course)

Course Search(string courseNum)

}

BinarySearchTree::BinarySearchTree() {

Set root equal to null

}

**Inserting Course into Binary Search Tree**

void BinarySearchTree::Insert(Course t\_course) {

IF root is null (tree is empty)

THEN set root equal to new Node with t\_course

Set root’s left equal to null

Set root’s right equal to null

ELSE (tree is not empty)

Pass root and t\_course as parameters in addNode() function

}

**Searching Binary Search Tree**

Course BinarySearchTree::Search(string t\_courseNum) {

Declare Course names empty

Declare Node pointer named current set equal to root

WHILE current is not equal to null

IF current’s courseNum is equal to t\_courseNum

RETURN current’s Course

ELSE IF current’s courseNum is less than t\_courseNum

Set current equal to current’s right

ELSE

Set current equal to current’s left

RETURN empty

}

**Printing Course Schedule**

void BinarySearchTree::PrintInOrder() {

Pass root through inOrder() function

}

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

IF node is not null

THEN pass node’s left as parameter in inOrder() function

OUTPUT node’s data

Pass node’s right as parameter in inOrder() function

}

**Main (menu function)**

void main() {

Declare character variable named userKey

Declare new Boolean variable named running set equal to true

WHILE running is not equal to false

call displayMenu() function

GET input from user store as userKey

IF userKey is equal to 1

THEN call loadFile() function

ELSE IF userKey is equal to 2

THEN call appropriate PrintSchedule() function

ELSE IF userKey is equal to 3

Declare new string named findCourse

OUTPUT “Enter Course Number”

GET input from user store as findCourse

Pass findCourse as parameter in appropriate Search() function

ELSE IF userKey is equal to 4

Set running equal to false

BREAK

ELSE

OUTPUT “Invalid entry, please enter menu option”

CONTINUE

**RunTime Analysis**

**Vector Analysis**

***Inserting Course into data structure (vector)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executes** | **Total Cost** |
| add newCourse to vector courseList | 1 | 1 | 1 |

**Total Cost: 1**

**Runtime: O(1)**

***Searching for Course in data structure (vector)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executes** | **Total Cost** |
| For each course in courseList | 1 | N | N |
| IF the courseNumbers match | 1 | N | N |
| THEN RETURN Course | 1 | 1 | 1 |
| Move to next course | 1 | N | N |
| RETURN empty course | 1 | 1 | 1 |

**Total Cost: 2 + 3N**

**Runtime: O(N)**

***Printing Course Schedule (vector)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Time Executes** | **Total Cost** |
| Declare int variable named low set equal to begin | 1 | 1 | 1 |
| Declare int variable named high set equal to end | 1 | 1 | 1 |
| Declare in variable named pivot set equal to midpoint | 1 | 1 | 1 |
| Declare Boolean variable named done set equal to false | 1 | 1 | 1 |
| WHILE done is not true | 1 | N | N |
| WHILE courseNum at index is less than courseNumat pivot | 1 | N | N-1 |
| Increment low by 1 | 1 | 1 | 1 |
| WHILE courseNum at pivot is less than courseNum at high | 1 | N | N |
| Decrement high by 1 | 1 | 1 | 1 |
| IF low is greater than or equal to high | 1 | N | N |
| THEN set done equal to true | 1 | 1 | 1 |
| ELSE | 1 | N | N |
| Swap Course positions low and high in vector courseList | 1 | N | N |
| Increment low by 1 | 1 | 1 | 1 |
| Increment high by 1 | 1 | 1 | 1 |
| RETURN high | 1 | 1 | 1 |
| Declare int variable named mid set equal to 0 | 1 | 1 | 1 |
| IF begin is greater than or equal to 0 | 1 | 1 | 1 |
| THEN RETURN | 1 | 1 | 1 |
| Set mid equal to partition performed on vector courseList | 1 | N | N |
| Recursively sort first half of vector | 1 | N-1 | N-1 |
| Recursively sort second half of vector | 1 | N-1 | N-1 |
| FOR all the Courses in courseList | 1 | N | N |
| OUPUT Course information | 1 | N | N |
| Move to next course | 1 | N-1 | N-1 |

**Total Cost: 13 + 8N + 4(N-1) 🡪 N + (N-1) + N-2) …+3 + 2+ 1**

**Runtime: O(N^2)**

**Hash Table Analysis**

***Inserting Course into data structure (hash table) Direct Hashing***

|  |  |  |  |
| --- | --- | --- | --- |
| **Cost** | **Line Cost** | **# of Times Executes** | **Total Cost** |
| Convert courseNum to integer | 1 | 1 | 1 |
| Declare integer set equal to hashed courseNum | 1 | 1 | 1 |
| Declare Node set equal to the new Node | 1 | 1 | 1 |
| Declare integer named “i” | 1 | 1 | 1 |
| Declare new pointer set equal to node at key | 1 | 1 | 1 |
| IF current pointer is null or key is UINT\_MAX | 1 | 1 | 1 |
| THEN set node at index “key” equal to new node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| FOR all the nodes in the vector | 1 | N | N |
| IF this node is null or key is UINT\_MAX | 1 | N | N |
| THEN set this node equal to new node | 1 | 1 | 1 |
| ELSE IF this node’s key is equal to key | 1 | N | N |
| THEN OUTPUT table is full | 1 | 1 | 1 |
| ELSE | 1 | N | N |
| Move to next node | 1 | N | N |

**Total Cost: 10 + 5N**

**Runtime: O(N)**

***Searching for course in data structure (hash table)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Cost** | **Line Cost** | **# of Times Executes** | **Total Cost** |
| Declare new empty Course object | 1 | 1 | 1 |
| Convert courseNum from string to int | 1 | 1 | 1 |
| Declare integer names key set equal to hashed courseNum | 1 | 1 | 1 |
| Declare new node pointer set equal to key | 1 | 1 | 1 |
| IF pointer courseNum matches | 1 | 1 | 1 |
| THEN RETURN Course | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| FOR all of the nodes | 1 | N | N |
| IF this node courseNum matches | 1 | N | N |
| THEN RETURN Course | 1 | 1 | 1 |
| ELSE IF this node’s key is equal to key | 1 | N | N |
| THEN RETURN empty Course | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| Move to next node | 1 | N | N |

**Total Cost: 10 + 4N**

**Runtime: O(N)**

***Printing Course Schedule (hash table)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executes** | **Total Cost** |
| FOR all the nodes in vector nodes | 1 | N | N |
| IF node key does not equal UINT\_MAX | 1 | N | N |
| THEN OUTPUT nodes Course information | 1 | N | N |

**Total Cost: 3N**

**Runtime: O(N)**

**Binary Search Tree Analysis**

***Inserting course into data structure (binary search tree)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Cost** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| IF root is null | 1 | 1 | 1 |
| THEN set root equal to new Node with course | 1 | 1 | 1 |
| Set roots left equal to null | 1 | 1 | 1 |
| Set roots right equal to null | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| Pass root and course parameters in addNode() | 1 | 1 | 1 |
| IF Course courseNum is less than nodes courseNum | 1 | N | N |
| IF node’s left is null | 1 | N | N |
| THEN set node’s left equal to new node with Course | 1 | 1 | 1 |
| ELSE | 1 | N | N |
| Recursively pass node’s left and course | 1 | N/2 | N/2 |
| ELSE | 1 | N | N |
| IF nodes right is null | 1 | N | N |
| THEN set nodes right equal to new node with Course | 1 | 1 | 1 |
| ELSE | 1 | N | N |
| Recursively pass node’s right and course | 1 | N/2 | N/2 |

**Total Cost: 8 + 6N + 2(N/2)**

**Runtime: O(N)**

***Searching for course in data structure (binary search tree)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Cost** | **Line Cost** | **# of Times Executes** | **Total Cost** |
| Declare new empty Course | 1 | 1 | 1 |
| Declare new pointer set equal to root | 1 | 1 | 1 |
| WHILE pointer is not null | 1 | N | N |
| IF pointers courseNum is equal to courseNum | 1 | N | N |
| RETURN pointers Course | 1 | 1 | 1 |
| ELSE IF pointer’s courseNum is less than courseNum | 1 | N | N |
| Move to the right | 1 | N | N |
| ELSE | 1 | N | N |
| Move to left | 1 | N | N |
| RETURN empty Course | 1 | 1 | 1 |

**Total Cost: 4 + 6N**

**Runtime: O(N)**

***Printing Course Schedule (binary search tree)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executes** | **Total Cost** |
| Pass root through inOrder() function | 1 | 1 | 1 |
| IF node is not null | 1 | N | N |
| THEN pass nodes left as parameter inOrder() function | 1 | N/2 | N/2 |
| OUPUT nodes data | 1 | N | N |
| Pass nodes right as parameter inOrder() function | 1 | N/2 | N/2 |

**Total Cost: 1 + 2N + 2(N/2)**

**Runtime: O(N)**

**Recommendation**

The data requirements currently requested by ABCU are not particularly vast. The provided course list is short, and the desired functions are not complex. It is for these reasons I see advantageous to most of the data structures tested. We will now begin to analyze each of these data structures in closer detail to form our recommendation.

We will begin with ruling out any of the data structures do not seem to fit our needs. The data we need to store includes courses that will primarily be identified by their corresponding course number. There are currently 8 courses provided by ABCU all of which are primarily identifies by their corresponding course numbers. All the course numbers are between the values of 100-400 and end in one of the three following combinations 00, 50, or 01. If a hash table were to be used with the current data, I would recommend direct hashing due to the limited size. Additionally, I believe this method would also reduce collision time as the course numbers are so like one another. With this design in mind, we find the hash table benefits very similar to those of a vector. However, hash table implementation is much more complex than the use of a vector would be. As a result, I do not believe that a hash table is the best option for this project.

This leaves us with two data structures, a vector, and a binary search tree, that we must now select between. The vector is the easiest of the data structures to implement and this is a convenience in design but is not the main advantage we are seeking. The largest benefit of the binary search tree is likely its’ inherit ability to sort the courses. The use of a vector requires the use of a sorting algorithm to produce our course schedule. In this analysis we utilized the quicksort algorithm which is commonly accepted as having a typical runtime of O (N log N). Our recommended design selects the mid-point of the data increasing the likelihood that we receive this typical runtime. However, as noted in the analysis above the worst-case runtime or big O analysis indicates a runtime of O(N^2). This may be compared to the production of a course schedule with the use of the binary search tree which has a worst-case runtime of O(N) for printing the course schedule. Though the course list is currently small, and runtimes may be little difference, this may be drastically noted if the course list were to increase. Thus, if we use a binary search tree and the worst-case scenario occurs where all the courses are on one side of the root, we still find a runtime complexity than that of a quicksorting a vector in the worst-case scenario. Furthermore, searching a vector is often less efficient than searching a binary search tree due to its linear algorithm design. Additionally, the worst-case scenario for both data structures is the same for searching. A vector provides a constant runtime for insertion which is advantageous, but I believe the sorting benefit of the binary search tree outweighs this. Finally insertion into the binary search tree provides acceptable runtimes that are primarily dictated by the trees height.

Due to the analysis and the above description, I recommend the use of a binary search tree for this project. I see few downsides to the use of a binary search tree in this case. The removal of an object from a binary search tree may be tricky to implement but is not required in this program. Furthermore, a binary search tree provides us many upsides. The runtime analysis indicated very little difference between the data structures. Hash tables are complicated to implement and provide more benefits for larger samples of data with more greatly varying indicators. Vectors require the use of a sorting algorithm to produce our course schedules which provide the worst-case runtime scenario of any of the data structures and functions analyzed. Meanwhile, the binary search tree is simple to implement, serves to sort the courses inherently, and provides the best or matching runtime analysis examined. Thus, I believe that a binary search tree and vector are the best options for this project, with the slight edge going to the binary search tree as it stores the objects in a sorted manner.