Angie Zanolini

Doctor Pollack

CS-300-T1009

6.2 Project One

**Linked List Pseudocode**

struct COURSE

Course code

Course title

Vector prereqs = null

vector<string> SPLIT (line, delimiter)

WHILE not newline

vector push\_back = items in between delimiter

RETURN items

APPEND(course)

Node\* newNode = new Node(course)

IF (list->head is null) // List empty

list->head = newNode

list->tail = newNode

ELSE

list->tail->next = newNode

list->tail = newNode

vector <course> GET\_COURSES

IF file is open

vector<courses> courses

FOR line in input document lines

vector splitLine = SPLIT(line, delimiter)

//delimiter = “,” to split entries by comma

IF(splitLine.length < 2)

CONTINUE

Course course = Course()

course.code = splitLine[0]

course.title = splitLine[1]

IF(splitLine.length > 2)

vector course.prereqs = new vector

FOR(int i = 2; i < splitLine.length; i++)

course.prereq[i-2] = splitLine[i];

courses.push\_back(course)

FOR course in courses

vector validPrereqs = new vector

FOR prereq in course.prereqs

FOR course in courses

IF(course.code == prereq)

validPrereqs push\_back (prereq)

BREAK

course.prereqs = validPrereqs

APPEND(course)

PRINT COURSE INFORMATION(list, input)

FOR course in courses

IF course.code = input

PRINT code, title

IF course.prereqs is not null

PRINT ("Prerequisites: ")

FOR prereq in course.prereqs

PRINT(prereq)

RETURN

PRINT "Invalid course code"

PREPEND(list, newNode)

newNode->next = list->head

list->head = newNode

INSERT AFTER(list, curNode, newNode) {

IF (curNode == list->tail) // Insert after tail

list->tail->next = newNode

list->tail = newNode

ELSE

newNode->next = curNode->next

curNode->next = newNode

REMOVE AFTER(list, curNode)

// Special case, remove head

IF (curNode is null && list->head is not null)

sucNode = list->head->next

list->head = sucNode

IF (sucNode is null) // Removed last item

list->tail = null

ELSE IF (curNode->next is not null)

sucNode = curNode->next->next

curNode->next = sucNode

IF (sucNode is null) // Removed tail

list->tail = curNode

FIND POSITION(list, data)

curNodeA = null

curNodeB = list->head

WHILE (curNodeB is not null && data > curNodeB->data)

curNodeA = curNodeB

curNodeB = curNodeB->next

return curNodeA

SORT(list)

beforeCurrent = list->head

curNode = list->head->next

WHILE (curNode is not null)

next = curNode->next

position = FIND POSITION(list, curNode->data)

IF (position == beforeCurrent)

beforeCurrent = curNode

ELSE

REMOVE AFTER(list, beforeCurrent)

IF (position is null)

PREPEND(list, curNode)

ELSE

INSERT AFTER(list, position, curNode)

curNode = next

PRINT SAMPLE SCHEDULE(list)

SORT(list)

FOR course in courses

PRINT course.code

void MAIN

HashTable\* table{}; //new empty hash table

WHILE (input is not 4)

PRINT "Main Menu"

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course Information"

PRINT "4. Exit"

TAKE input

IF (input is 1)

GET COURSES()

ELSE IF (input is 2)

PRINT SAMPLE SCHEDULE(list)

ELSE IF (input is 3)

PRINT "Enter a course code"

TAKE input

PRINT COURSE INFORMATION(list, input)

ELSE

PRINT "Invalid input"

TAKE input

**Linked List Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| IF file is open | 1 | 1 | 1 |
| vector<courses> courses | 1 | 1 | 1 |
| FOR line in input document lines | 1 | n | n( |
| Vector splitLine = SPLIT(line, delimiter) | 1 | n | n |
| IF(splitLine.length < 2) CONTINUE | 1 | n | n |
| Course course = Course() | 1 | n | n |
| course.code = splitLine[0] | 1 | n | n |
| course.title = splitLine[1] | 1 | n | n |
| IF(splitLine.length > 2) | 1 | n | n |
| vector course.prereqs = new vector | 1 | n | n |
| FOR(int i = 2; i < splitLine.length; i++) | 1 | n | n |
| course.prereq[i-2] = splitLine[i]; | 1 | n | n |
| courses.push\_back(course) | 1 | n | n) |
| FOR course in courses | 1 | n | n( |
| vector validPrereqs = new vector | 1 | n | n |
| FOR prereq in course.prereqs | 1 | n | n( |
| FOR course in courses | 1 | n | n( |
| IF(course.code == prereq) | 1 | n | n |
| validPrereqs push\_back (prereq); BREAK | 1 | 1 | 1)) |
| course.prereqs = validPrereqs | 1 | 1 | 1) |
| APPEND(course) | 1 | n | n |
| **Total cost** | | | 2+2n²+n³ |
| **Run time** | | | O(n³) |

**Hash Table Pseudocode**

struct COURSE

unsigned int key

Node\* next

Course code

Course title

Vector prereqs = null

vector<string> SPLIT (line, delimiter)

WHILE not newline

vector push\_back = items in between delimiter

RETURN items

unsigned int HASH (key)

RETURN (key % hash table size)

vector <course> GET COURSES

IF file is open

vector<courses> courses

FOR line in input document lines

Vector splitLine = SPLIT(line, delimiter)

//delimiter = “,” to split entries by comma

IF(splitLine.length < 2)

CONTINUE

Course course = Course()

course.code = splitLine[0]

course.title = splitLine[1]

key = HASH(atoi course code)

node = &(courses at key)

IF(splitLine.length > 2)

vector course.prereqs = new vector

FOR(int i = 2; i < splitLine.length; i++)

course.prereq[i-2] = splitLine[i]

IF (node is null)

new pointer = new Course(course, key)

courses.insert(courses.begin() + key, \*pointer)

ELSE

IF (node key = default value)

node key = key

node course = course

next node = null

ELSE

WHILE (next node is not null)

node = next node

next node = new Course (course, key)

FOR course in courses

vector validPrereqs = new vector

FOR prereq in course.prereqs

FOR course in courses

IF(course.code == prereq)

validPrereqs push\_back (prereq)

BREAK

course.prereqs = validPrereqs

void PRINT ALL

FOR (courses begin to end)

pointer = &(courses current index)

IF (pointer key not default value) //contains course entry

PRINT course code, title, prereqs

pointer = next pointer

WHILE (pointer not null) //iterate through chain

PRINT course code, title, prereqs

Pointer = next pointer

void PRINT SAMPLE SCHEDULE(table)

FOR (courses begin to end)

pointer = &(courses current index)

IF (pointer key not default value) //contains course entry

PRINT course code

pointer = next pointer

WHILE (pointer not null) //iterate through chain

PRINT course code

Pointer = next pointer

void PRINT COURSE INFORMATION(table, course)

key = HASH(atoi course)

pointer = &(table item at key)

IF (pointer course code == course)

PRINT course code, title

IF (couse.prereqs not null)

PRINT "Prerequisites: "

FOR prereq in prereqs

PRINT prereq

RETURN

WHILE (pointer is not null)

IF (pointer course code == course)

PRINT course code, title

IF (couse.prereqs not null)

PRINT "Prerequisites: "

FOR prereq in prereqs

PRINT prereq

RETURN

Pointer = next pointer

PRINT "Course code does not exist"

void MAIN

HashTable\* table{}; //new empty hash table

WHILE (input is not 4)

PRINT "Main Menu"

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course Information"

PRINT "4. Exit"

TAKE input

IF (input is 1)

GET COURSES()

ELSE IF (input is 2)

PRINT SAMPLE SCHEDULE(table)

ELSE IF (input is 3)

PRINT "Enter a course code"

TAKE input

PRINT COURSE INFORMATION(table, input)

ELSE

PRINT "Invalid input"

TAKE input

**Hash Table Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| IF file is open | 1 | 1 | 1 |
| vector<courses> courses | 1 | 1 | 1 |
| FOR line in input document lines | 1 | n | n( |
| Vector splitLine = SPLIT(line, delimiter) | 1 | n | n |
| IF(splitLine.length < 2) CONTINUE | 1 | n | n |
| Course course = Course() | 1 | n | n |
| course.code = splitLine[0] | 1 | n | n |
| course.title = splitLine[1] | 1 | n | n |
| key = HASH(atoi course code) | 1 | n | n |
| node = &(courses at key) | 1 | n | n |
| IF(splitLine.length > 2) | 1 | n | n |
| vector course.prereqs = new vector | 1 | n | n |
| FOR(int i = 2; i < splitLine.length; i++) | 1 | n | n |
| course.prereq[i-2] = splitLine[i]; | 1 | n | n |
| IF (node is null) | 1 | n | n |
| new pointer = new Course(course, key) | 1 | n | n |
| courses.insert(courses.begin() + key, \*pointer) | 1 | n | n |
| ELSE | 1 | n | n |
| IF (node key = default value) | 1 | n | n |
| node key = key | 1 | n | n |
| node course = course | 1 | n | n |
| next node = null | 1 | n | n |
| ELSE | 1 | n | n |
| WHILE (next node is not null) | 1 | n | n |
| node = next node | 1 | n | n |
| next node = new Course (course, key) | 1 | n | n) |
| FOR course in courses | 1 | n | n( |
| vector validPrereqs = new vector | 1 | n | n |
| FOR prereq in course.prereqs | 1 | n | n( |
| FOR course in courses | 1 | n | n( |
| IF(course.code == prereq) | 1 | n | n |
| validPrereqs push\_back (prereq); BREAK | 1 | 1 | 1)) |
| course.prereqs = validPrereqs | 1 | 1 | 1) |
| **Total cost** | | | 2+n+2n²+n³ |
| **Run time** | | | O(n³) |

**Binary Search Tree Pseudocode**

struct COURSE

Course code

Course title

Vector prereqs = null

struct NODE

Node\* left

Node\* right

vector<string> SPLIT (line, delimiter)

WHILE not newline

vector push\_back = items in between delimiter

RETURN items

void INSERT (course)

IF (root is null)

root = NEW NODE (course)

ELSE

ADD NODE (root, course)

void ADD NODE (node, course)

IF (node->atoi course code compare (course.code) > 0)

IF (node->left is null)

node->left = NEW NODE (course)

ELSE

ADD NODE(node->left, course) //recursively go down tree

ELSE

IF (node->right is null)

node->right = NEW NODE (course)

ELSE

ADD NODE(node->right, course)

void IN ORDER (node)

IF (node is not null)

IN ORDER (node->left)

PRINT node->course info

IN ORDER(node->right)

void GET COURSES (tree)

IF file is open

FOR line in input document lines

Vector splitLine = SPLIT(line, delimiter)

//delimiter = “,” to split entries by comma

IF(splitLine.length < 2)

CONTINUE

Course course = Course()

course.code = splitLine[0]

course.title = splitLine[1]

IF(splitLine.length > 2)

vector course.prereqs = new vector

FOR(int i = 2; i < splitLine.length; i++)

course.prereq[i-2] = splitLine[i]

courses.push\_back(course)

FOR course in courses

vector validPrereqs = new vector

FOR prereq in course.prereqs

FOR course in courses

IF(course.code == prereq)

validPrereqs push\_back (prereq)

BREAK

course.prereqs = validPrereqs

FOR course in courses

INSERT (course)

void PRINT ALL(Tree courses)

IN ORDER(root)

void PRINT SAMPLE SCHEDULE(Tree courses)

Node\* node = root

IF (node is not null)

IN ORDER (node->left)

PRINT node->course code

IN ORDER(node->right)

void PRINT COURSE INFORMATION(Tree courses, string courseNumber)

Node\* current = root;

WHILE (current is not null) {

IF (current->course code compare(courseNumber) == 0)

PRINT course code, course title

IF course.prereqs is not null

PRINT "Prerequisites: "

FOR prereq in course prereqs

PRINT prereq

RETURN

IF (current->course code compare(courseNumber) > 0)

current = current->left

ELSE

current = current->right

PRINT "course code does not exist"

void MAIN

BinarySearchTree\* tree{}; //new empty tree

WHILE (input is not 4)

PRINT "Main Menu"

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course Information"

PRINT "4. Exit"

TAKE input

IF (input is 1)

GET COURSES(tree)

ELSE IF (input is 2)

PRINT SAMPLE SCHEDULE(tree)

ELSE IF (input is 3)

PRINT "Enter a course code"

TAKE input

PRINT COURSE INFORMATION(tree, input)

ELSE

PRINT "Invalid input"

TAKE input

**Binary Search Tree Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| IF file is open | 1 | 1 | 1 |
| vector<courses> courses | 1 | 1 | 1 |
| FOR line in input document lines | 1 | n | n( |
| Vector splitLine = SPLIT(line, delimiter) | 1 | n | n |
| IF(splitLine.length < 2) CONTINUE | 1 | n | n |
| Course course = Course() | 1 | n | n |
| course.code = splitLine[0] | 1 | n | n |
| course.title = splitLine[1] | 1 | n | n |
| IF(splitLine.length > 2) | 1 | n | n |
| vector course.prereqs = new vector | 1 | n | n |
| FOR(int i = 2; i < splitLine.length; i++) | 1 | n | n |
| course.prereq[i-2] = splitLine[i]; | 1 | n | n |
| courses.push\_back(course) | 1 | n | n) |
| FOR course in courses | 1 | n | n( |
| vector validPrereqs = new vector | 1 | n | n |
| FOR prereq in course.prereqs | 1 | n | n( |
| FOR course in courses | 1 | n | n( |
| IF(course.code == prereq) | 1 | n | n |
| validPrereqs push\_back (prereq); BREAK | 1 | 1 | 1)) |
| course.prereqs = validPrereqs | 1 | 1 | 1) |
| FOR course in courses | 1 | n | n( |
| INSERT(course) | 1 | n | n) |
| **Total cost** | | | 2+n+3n²+n³ |
| **Run time** | | | O(n³) |

**Analysis**

Linked lists, hash tables, and trees all have advantages and disadvantages. Linked lists are simpler to assemble, but require several function calls and nested loops to sort their content. Hash tables require more steps when creating objects, but hashing data to determine location helps to sort objects as a process of adding them to the table. Sorting while creating new objects is also a feature of trees. However, a tree’s efficiency can vary greatly based on the order in which objects are added, which can affect a tree’s height and the number of recursive calls required to traverse it.

All three data structures have a runtime of O(n³) because of the nested loops required for parsing through the input document, verifying each line contains at least a course code and title, and confirming that any prerequisites that are listed exist as a course elsewhere in the document.

For the purpose of reading and verifying course input, and outputting a sorted list of courses or a specific course’s details, a linked list makes for simple object creation but requires sort, find position, remove after, prepend, and insert after functions to sort the list in order of their course codes. This also means that the print course information function has to traverse the entire list in order to find the particular entry to output.

A benefit to hash tables is, because object data is hashed to a key that determines the location of where the object will be stored in the structure, hash tables can sort objects as they are added. By converting the course code string into a unique ASCII value and using modulo division with the table’s size, each course should be mapped to locations which will allow the print sample schedule function to go in ascending order. As a backup, hash tables can also chain values that are mapped to the same key. Overall, hash tables take more steps to create keys and pointers for objects, but allow for faster traversal later.

Trees also offer the opportunity to sort objects as they are added to the structure and can feature much faster runtime when traversing a tree’s height over a linked list’s order. A binary search tree with N nodes can have a height as low as log₂N and as high as N-1. Trees may offer opportunities to illustrate hierarchies, but unfortunately do not lend themselves to our input course structure. If the tree were to use CSCI100 as the root, MATH201 is not a “child” of CSCI100 in terms of prerequisites, but is a “parent” of CSCI300. CSCI400 has two “parents” in terms of prerequisites—CSCI301 and CSCI350. If the tree were to be sorted by course number alone, the tree’s height could potentially impact the runtime with more recursion to print the courses in order, depending on the order in which the courses were added to the tree.

For the needs of our program, I am planning to use a hash table for my code. For the same runtime to create, verify, and add objects to the data structures, I looked at each print sample schedule function to analyze runtime. Because the linked list must first be sorted, the runtime comes to O(n²). For a hash table, if hashing course ASCII values to unique keys means there are no chained values, the runtime for printing a sample schedule is O(n). Similarly, while a binary search tree may require several recursive calls, printing a sample schedule would have a runtime of O(n) to visit each node. Next, looking at the print course information function, a hash table with no chaining will have a runtime of O(1) because each course will be mapped to its own key and can be found immediately. A binary search tree would have a runtime of O(n) because it must traverse the tree to find the specific course. For all the requirements of our program, a hash table data structure will offer the most benefits with the fewest drawbacks.