# Brent Artuch

CS-300-T4190

4/9/2023

# CS 300 Pseudocode Document

Struct Course

{

String courseNumber

String title

String prerequisite

}

Struct Node

{

Course course

}

**// Vector pseudocode**

Vector<Course> loadCourses(String data, Type dataType, String filePath)

{

Vector<Course> = Courses

if dataType is file and the file is open

for all rows in the file at filePath

create new Course

for each item

if the item count is less than 2

output error: “Not enough inputs”

if the item is not null and index less equals 2 and item 1 is courseNumber format

courseNumber = row item 1

title = row item 2

while next item not null

if item is in courses data

prerequisiteN = item at nth index

else

output error: “Prerequisite not met”

push course to the end of Courses

close file

if file size is 0

output error: “No data in file”

else

output error: “Failed to open file”

return Courses

}

int partition(Vector<Course> courses, int begin, int end)

{

int low = begin

int high = end

int midpoint = low+(high-low)/2

Course pivot = course at midpoint

bool done = false

while not done

while courseNumber of course at low less than at pivot

increment low

while pivot less than courseNumber of course at high

decrement high

if low greater equals high

done = true

else

swap low and high courses

increment low

decrement high

return high

}

void quickSort(Vector<Course> courses, int begin, int end)

{

int mid = 0

if begin greater equals end then return

mid = partition(courses, begin, end)

quickSort(courses, begin, mid)

quickSort(courses, mid+1, end)

}

void printSampleSchedule(Vector<Course> courses)

{

for each course in courses

output the course and all of its attributes

}

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

{

for all courses

if the course is the same as courseNumber

output the course information

for each prerequisite of the course

output prerequisite course information

else

print “Course not found.”

}

**// Hashtable pseudocode**

Vector<Node> nodes

Vector<Course> unsortedCourses

HashTable(int size)

resize to size vector courses count

HashKey(int key)

key modulus table size

void insert(course)

{

create key

retrieve node at key

if the node is null

create new node

insert new node with key

else

if node key is null

node key = key

node course = course

node next = null

else

while node next is not null

node = node next

node next = new node

}

void loadCourses(String data, Type dataType, String filePath, HashTable hashTable)

{

if dataType is file and the file is open

for all rows in the file at filePath

create new Course

for each item

if the item count is less than 2

output error: “Not enough inputs”

if the item is not null and index less equals 2 and item 1 is courseNumber format

courseNumber = row item 1

title = row item 2

while next item not null

if item is in courses data

prerequisiteN = item at nth index

else

output error: “Prerequisite not met”

insert course to end of hash table

close file

if file size is 0

output error: “No data in file”

else

output error: “Failed to open file”

}

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

{

Course course

key = hashKey

node = node at key

if node not null and key not null and node courseNumber matches

output the course in the node with matching courseNumber

if node is null or key is null

output “Course not found.”

while node not null

if current node matches return it

node becomes next node

output “Course not found.”

}

Vector<Course> createCourseList(Hashtable<Course> courses)

{

Vector<Course> = courses

for node in nodes

node = node at the current index

while node is not null and the key is set

push node course object to end of courses

move to the next course in the node

return courses

}

**// Tree pseudocode**

BinarySearchTree()

root = null

void insert(course)

{

if root is null

root becomes new course node

else

add node root and course

}

void addNode(node, course)

{

if node courseNumber is larger, add to left

if no left node

this node becomes left

else

go down the left node

addNode(node left, course)

else

if no right node

this node becomes right

else

go down the right node

addNode(node right, course)

}

void loadCourses(String data, Type dataType, String filePath, BinarySearchTree bst)

{

if dataType is file and the file is open

for all rows in the file at filePath

create new Course

for each item

if the item count is less than 2

output error: “Not enough inputs”

if the item is not null and index less equals 2 and item 1 is courseNumber format

courseNumber = row item 1

title = row item 2

while next item not null

if item is in courses data

prerequisiteN = item at nth index

else

output error: “Prerequisite not met”

insert course to end of binarySearchTree

close file

if file size is 0

output error: “No data in file”

else

return error: “Failed to open file”

}

void printSampleSchedule(Tree<Course> courses)

{

if course node not equal to null

printSampleSchedule(left node)

print course attributes

printSampleSchedule(right node)

}

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

{

current node is the root node

while the current node is not null

if matching courseNumber found

print course and its attributes

if courseNumber is smaller than current node then go left

else larger so travers right

if course not found print: “Course not found.”

}

**// Main (menu)**

int main(int argc, char argv)

{

Course course

Vector<Course> courses

courseTable = new HashTable

bst = new BinarySearchTree

dst = input data structure type

int choice = 0

while(choice not 9)

Output:

Menu:

1. Load Data Structure
2. Print Course List
3. Print Course

9. Exit

output “Enter choice: ”

input choice

switch(choice)

case 1:

if dst isVector

loadCourses(data, file, filepath)

else if dst is Hash Table

loadCourses(data, file, filepath, courseTable)

else

loadCourses(data, file, filepath, bst)

break

case 2:

if dst is Vector

quickSort(courses, 0, courses size-1)

printSampleSchedule(courses)

else if dst is Hash Table

createCourseList(courseTable)

quickSort(courses)

printSampleSchedule(courses)

else

// sorted via In Order Traversal

printSampleSchedule(bst)

break

case 3:

if dst is Vector

printCourseInformation(courses, courseNumber)

else if dst is Hash Table

printCourseInformation(courseTable, courseNumber)

else

printCourseInformation(bst, courseNumber)

break

output “Goodbye.”

}

## 

## 

## 

## File Parsing and Course Object Creation Runtime Analysis: Vector

| **loadCourses() All Data Structures** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| if dataType is file and the file is open | 1 | 1 | 1 |
| for all rows in the file at filePath | 1 | n | n |
| create new Course | 1 | 1 | 1 |
| for each item | 1 | n | n |
| if the item count is less than 2 | 1 | 1 | 1 |
| output error: “Not enough inputs” | 1 | 1 | 1 |
| if the item is not null and index less equals 2 | 1 | 1 | 1 |
| courseNumber = row item 1 | 1 | 1 | 1 |
| title = row item 2 | 1 | 1 | 1 |
| while next item not null | 1 | n | n |
| if item is in courses data | 1 | 1 | 1 |
| prerequisiteN = item at nth index | 1 | n | n |
| else | 1 | 1 | 1 |
| output error: “Prerequisite not met” | 1 | 1 | 1 |
| push course to end of courses (Vector) | 1 | n | n |
| close file | 1 | 1 | 1 |
| if file size is 0 | 1 | 1 | 1 |
| output error: “No data in file” | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| output error: “Failed to open file” | 1 | 1 | 1 |
| **Worst Case Vector Total Cost** | | | 3n^3 + 11 |
| **Runtime** | | | O(n^3) |
| **Space** | | | O(n) |

Additional Runtime Notes:

* It is also important to note that sorting the vector alpha numerically requires a quickSort algorithm at **O(n\*log(n))**.
* Printing a specific course and its prerequisites uses an **O(n)** method.
* Printing the vector is an **O(n)** loop.

## File Parsing and Course Object Creation Runtime Analysis: Hash Table

| **loadCourses() HashTable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| … |  |  |  |
| insert(course) to end of hash table (HashTable) | 2n+12 | n | n(2n+12) |
| … |  |  |  |
| insert(course) |  |  | 2n+12 |
| create key | 1 | 1 | 1 |
| retrieve node at key | 1 | 1 | 1 |
| if the node is null | 1 | 1 | 1 |
| create new node | 1 | 1 | 1 |
| insert new node with key | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| if node key is null | 1 | 1 | 1 |
| node key = key | 1 | 1 | 1 |
| node course = course | 1 | 1 | 1 |
| node next = null | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| while node next not null | 1 | n | n |
| node = node next | 1 | n | n |
| node next = new node | 1 | 1 | 1 |
| **Worst Case HashTable Total Cost** | | | 3n^3+2n^2 + 12n + 11 |
| **Runtime** | | | O(n^3) |
| **Space** | | | O(n) |

Additional Runtime Notes:

* It is also important to note that sorting the hash table alpha numerically requires the creation of a courses vector at **O(n^2)** and a quickSort algorithm at **O(n\*log(n))**.
* Uses the vector method to print the sorted courses at **O(n)**.
* Printing a specific course is **O(1)**.

## 

## 

## 

## File Parsing and Course Object Creation Runtime Analysis: Binary Search Tree

| **loadCourses() BinarySearchTree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| … |  |  |  |
| insert(course) to end of bst (Binary Search Tree) | n^2+4n+3 | n | n(n^2+4n+3) |
| … |  |  |  |
| insert(course) |  |  | n^2+4n+3 |
| if root is null | 1 | 1 | 1 |
| root becomes new course node | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| addNode(root, course) | n+4 | n | n(n+4) |
|  |  |  |  |
| addNode(node, course) |  |  | n+4 |
| if node courseNumber is larger, add to left | 1 | 1 | 1 |
| if no left node | 1 | 1 | 1 |
| this node becomes left | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| addNode(node left, course) | n | n | n |
| else | 1 | 1 | 1 |
| if no right node | 1 | 1 | 1 |
| this node becomes right | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| addNode(node right, course) | 1 | n | n |
| **Worst Case BST Total Cost** | | | 4n^3 + 4n^2 + 3n + 11 |
| **Runtime** | | | O(n^3) |
| **Space** | | | O(n) |

Additional Runtime Notes:

* Alphanumeric sorting and printing is accomplished with In Order Traversal at **O(n)**.
* Printing a specific course is **O(log(n))**.

**Advantages and Disadvantages**

*Vector*: First, it is the fastest at loading the course objects from the file because it doesn’t need to create additional nodes for the courses to be tied too. Second, it is the middle choice in terms of outputting all of the courses in alphanumeric order by courseNumber, because it has to be sorted using quick sort and then output using a standard iteration loop. Finally, it is the worst choice of the three in terms of searching for a specific course due to the necessity of a nested loop to print the courseNumber and name along with all of the prerequisites. Also, due to the nature of vectors being dependent on indices, it will take longer to find the desired course if its index is towards the end of the vector.

*Hash Table*: The Hash Table is just a bit slower than the vector in terms of loading the course objects due to the iteration through the nodes array to bind a course to a node based on the calculated key value. It is the worst choice for printing all of the courses in alphanumeric order by courseNumber, because the Hash Table needs to be converted to a vector so that it can then be sorted using the quickSort method and printed using a loop. However, the Hash Table is efficient when searching for an individual course as it calculates the key for the desired course based on the given course number and quickly references the corresponding node if it exists.

*Binary Search Tree*: This data structure is a bit slower than the other because of the recursive addNode method tied to the insert method for loading the course objects from the file. It is the best choice for sorting the course alphanumerically by course number because the In Order Traversal sorts and outputs the courses in one recursive method. This data structure is also efficient for searching for a specific course, because the method will traverse left or right down the tree based on the value of the desired courseNumber against the current node courseNumber which quickly eliminates the courses that don’t match until the desired course is found.

**Recommendation**

Based on the analysis of the three structures, I will utilize the Binary Search Tree data structure for this application. The first reason I believe this to be the best choice, is because the course objects will be sorted by course number automatically upon creation of the tree which makes up for the slightly bulkier runtime for creating a bst. So, when the In Order Traversal output method is called, the application will quickly and efficiently print all of the course information by starting at the node with the node with the smallest courseNumber and working its way across. Second, the binary search tree isn’t superior to the hash table in terms of searching for a specific course, but will have overall better runtime performance across all methods because it won’t need to be converted to a vector and sorted for the printAll function. The manner in which the bst traverses down the tree is also far more efficient than the vector in terms of searching for a specific course, because it doesn’t need to touch every course before finding the desired one.

Another reason that I would choose the Binary Search Tree data structure is that it is memory efficient. The hash table creates quite a few nodes that oftentimes go unused because none of the provided ID numbers create a key with the modulus operator that will match to certain nodes. The bst will only create nodes as needed and will bind only one course object to each node as opposed to the hash table who will sometimes need to create a linked list within a node if there is a key collision. Finally, the bst is highly scalable and new courses could be easily inserted or irrelevant courses could be easily deleted because the node left, node right, and root pointers can be quickly referenced and changed as needed should that functionality be desired later on.