

Pseudocode run-time analysis

Caleb Partain



June 18, 2023

SNHU

Caleb Partain

Table of Contents

[VECTOR PSEUDOCODE: 2](#_Toc137946770)

[HASH TABLE PSEUDOCODE: 3](#_Toc137946771)

[BINARY SEARCH TREE PSEUDOCODE: 5](#_Toc137946772)

[MENU PSEUDOCODE: 7](#_Toc137946773)

[BIG O NOTATION: 9](#_Toc137946774)

[Vector: 9](#_Toc137946775)

[Hash table: 9](#_Toc137946776)

[Binary search treev: 9](#_Toc137946777)

[ADVANTAGES AND DISADVANTAGES: 10](#_Toc137946778)

[Vector: 10](#_Toc137946779)

[Hash Table: 10](#_Toc137946780)

[Binary Search Tree: 10](#_Toc137946781)

[RECOMMENDATION: 11](#_Toc137946782)

# VECTOR PSEUDOCODE:

function readCourseData(file):

courseList = create an empty list

open the file

for each line in the file:

courseData = split the line by ","

course = create a new Course with courseData[0] and courseData[1:] as title and prereqs

add course to the courseList

function printCourseList(courseList):

sortedCourses = sort the courseList by number

for each course in sortedCourses:

print the course title and prerequisites

function printCourse(courseList, courseNum):

for each course in the courseList:

if course number is equal to courseNum:

print the course title

print the course prereqs

break

# HASH TABLE PSEUDOCODE:

function insert(courseTable, courseNum, course):

add an entry in courseTable with key courseNum and value course

function get(courseTable, courseNum):

retrieve the value associated with the key courseNum in courseTable

function getKeys(courseTable):

get all the keys from courseTable and sort them

function readCourseData(file):

courseTable = create an empty course table

open the file

for each line in the file:

courseData = split the line by ","

course = create a new Course with courseData[0] and courseData[1:] as title an prereqs

insert course into courseTable with course number as the key

return courseTable

function printCourseList(courseTable):

courseList = getKeys(courseTable)

for each courseNum in courseList:

print the courseNum

function printCourse(courseTable, courseNum):

course = get the value associated with courseNum in courseTable

if course exists:

print the course title

print the course prereqs

# BINARY SEARCH TREE PSEUDOCODE:

function insertCourse(node, course):

if node is null:

create a new tree node with course as the course

assign the new node to node

else if course number is less than node course number:

recursively call insertCourse with node's leftChild and course

else:

recursively call insertCourse with node's rightChild and course

return node

function readCourseData(file):

assign null to root

open the file

for each line in the file:

courseData = split the line by ","

course = create a new Course with courseData[0]and courseData[1:] as title and prereqs

insertCourse with root and course

function printCourseList():

inOrderTraversal with root

function inOrderTraversal(node):

if node is not null:

recursively call inOrderTraversal with node's leftChild

print node course number

recursively call inOrderTraversal with node's rightChild

function printCourse(courseNum):

node = findNode with root and courseNum

if node is not null:

print node course title

print node course prereqs

function findNode(node, courseNum):

if node is null or node course number is equal to courseNum:

return node

else if courseNum is less than node course number:

recursively call findNode with with node's leftChild and courseNum

else:

recursively call findNode with node's rightChild and courseNum

# MENU PSEUDOCODE:

Function sortCourses():

sortedCourses = copy of courses

n = size of sortedCourses

for i from 0 to n-1:

for j from 0 to n-i-1:

if sortedCourses[j] > sortedCourses[j+1]:

swap sortedCourses[j] and sortedCourses[j+1]

for each course in sortedCourses:

print course

Function main() :

Define a timer variableclock\_t ticks

Define a binary search tree to hold all courses

choice = 0

while (choice not equal to 9) :

print "Menu:"

print "1. Load Courses"

print "2. Display All Courses"

print "3. Find Course"

print "9. Exit"

print "Enter choice: "

cin >> choice

switch (choice) :

case 1:

Initialize a timer variable before loading courses

Complete the method call to load the courses loadCourses(csvPath, bst)

ticks = current clock ticks minus starting clock ticks

print clock ticks

print time it took

break

case 2:

Complete the method call to order the courses

break

case 3:

ticks = clock()

course = bst->Search(courseKey)

ticks = current clock ticks minus starting clock ticks

if (course id not empty):

displayCourse(course)

else:

prtin “course not found”

print clock ticks

print time it took

break

Print "Good bye."

return 0

# BIG O NOTATION:

## Vector:

| **Code Snippet** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| readCourseData(file) | 1 | n | n |
| printCourseList(courseList) | 1 | n | n |
| printCourse(courseList, courseNum) | 4 | n | 4n |
| Total Runtime | 6n | Big O Notation | O(n) |

## Hash table:

| **Code Snippet** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| readCourseData(file) | 1 | n | n |
| printCourseList(courseTable) | 1 | n | n |
| printCourse(courseTable, courseNum) | 4 | 1 | 4 |
| Total Runtime | 2n+4 | Big O Notation | O(n) |

## Binary search treev:

| **Code Snippet** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| readCourseData(file) | 1 | n | n |
| printCourseList() | 1 | n | n |
| inOrderTraversal(node) | 2 | n | 2n |
| printCourse(courseNum) | 4 | 1 | 4 |
| findNode(node, courseNum) | 4 | log n | 4 log n |
| Total Runtime | 4n + 4 + 4log(n) | Big O Notation | O(n Log(n)) |

# ADVANTAGES AND DISADVANTAGES:

## Vector:

Just the same as any other selection in day-to-day life, data structures come with advantages and disadvantages. The vector data structure is among the simpler data structures and can be very easy to implement into your database. Among the strengths of the vector data structure is its efficient memory utilization. Unlike other data structures, a vector only required contiguous memory allocation. While it is incredible for applications where the order of elements is important, this can come at a cost. Vectors are not optimized for fast lookup with keys like hash tables. For this reason, searching for a specific element can be rather time consuming. This will also domino and create issues when attempting to add or delete elements in certain positions.

## Hash Table:

The hash table is an appealing option for many applications, boasting a multitude of advantages over its counterparts. As mentioned before, hash tables are optimized for quick lookup. This is because they hold keys for their elements to be searched by. This fast retrieval of elements makes it a strong choice for application with a heavy use of insertion, deletion, or lookup. While hash tables can be fast when done correctly, collisions can drastically slow the process down. I have taken a small amount of time to research methods such as chaining and open addressing which can help. Hash tables are also quite a bit more complex than vectors and will require much higher memory utilization. Due to the order of elements not being guaranteed, hash tables are out of the question when order of elements matters.

## Binary Search Tree:

Another option for when the order of elements matters is the binary search tree. A binary search tree is kind of the best of both worlds when it comes to hash tables and vectors. You can keep the order of elements, but also search elements based on their keys. It can even be quite efficient when it comes to average-case time complexity. That being said, the wors-case time complexity for searching, inserting and deleting can be much slower than both the vector and hash table. This is because after any insertion or deletion, the tree may require balancing to maintain efficiency. As you can imagine, a data structure this complex can be difficult to implement. BSTs also require more in terms of memory due to the necessary left and right child pointers.

# RECOMMENDATION:

While the vector can be a tempting choice in terms of efficiency for its incredibly allocation of memory, looking back at the Big O notation charts there was a clear winner. Both the vector and hash table have a time complexity of O(n) for reading the course data and printing the course list. This means they will scale linearly with the number of courses. The difference is when we look at the search functions. As mentioned before, hash tables allow the use of keys for quick searching. The time complexity for printing a single course will stay constant no matter the number of courses. The same can be said for binary search trees (BST), however, the use of a binary search tree over a hash table requires added complexity and memory usage. While a BTS does have impressive average-case time complexity, the worst-case time complexity far outweighs the benefits when compared to the hash table. In summary, due to the efficiency, fast look up, and flexibility of a hash table, I believe it would be the best data structure for the universities course list. The benefits of the hash table are only highlighted when considering the possibility of course numbers being used for keys.