# Cristian Chavez

10/7/22 - CS 300

# CS 300 Pseudocode Document

**Vector pseudocode**

void openFile {**//OPEN FILE**

open courseList

IF file opens successfully

Check for at least 2 comma separated values

Check that for a course number that isn’t at start of line, hasn’t been see already

ELSE

Print file opening error

}

int numPrerequisiteCourses(Vector<Course> courses, Course c) { **//Create Objects**

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

**//Print Course List Alphabetically**

sortedList = sort (courseList)

FOR iterate through sortedList

Print courses

}

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

**//Print Course information for chosen course number**

**for all courses**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

**Hashtable pseudocode**

void openFile{  
 open courseList

IF file opens successfully

Check for at least 2 comma separated values

Check that for a course number that isn’t at start of line, hasn’t been see already

ELSE

Print file opening error  
}

int numPrerequisiteCourses(Hashtable<Course> courses) {

key set equal to hashed courseNumber

oldNode is set equal to nodes at key

IF oldNode is equal to null

newNode created and takes courseNumber and key

insert newNode to key position

ELSE

IF oldNode equals end of table

oldNode->key is equal to key

oldNode->courseNumber is equal to courseNumber

oldNode->next is null

ELSE

WHILE oldNode->next is not null

oldNode is equal to oldNode->next

oldNode->next is equal to new node and takes courseNumber and key

}

void printSampleSchedule(Hashtable<Course> courses) {

}

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

key set equal to hashed courseNumber

Node is set equal to nodes at key

IF node is not null and course number is equal to course number

RETURN node->courseNumber

IF node is null or end of table

RETURN original courseNumber

WHILE node isn’t null

IF node->key is equal to current

RETURN node->courseNumber

Node is equal to node->next

Return courseNumber

**Tree pseudocode**

void openFile{

open courseList

IF file opens successfully

Check for at least 2 comma separated values

Check that for a course number that isn’t at start of line, hasn’t been see already

ELSE

Print file opening error

}

int numPrerequisiteCourses(Tree<Course> courses) {

IF tree -> root is null

root is set equal to node

node -> left is null

node -> right is null

ELSE

Current is equal to tree -> root

WHILE current is not null

IF node is smaller than current

IF current -> left is null

current -> left is equal to node

current is equal to null

ELSE

current equal to current -> left

ELSE

IF current -> right is null

current -> right is equal to node

current is equal to null

ELSE

current equal to current -> right

node->left is equal to null

node->right is equal to null

}

void printSampleSchedule(Tree<Course> courses) {

Create Function BST print, take Node as argument

IF node is null

RETURN

BST print (node -> left)

Print node

BST print (node -> right)

}

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

current is equal to tree -> root

WHILE current is not null

IF courseNumber is equal to current -> courseNumber

RETURN current

ELSE IF courseNumber is less than current -> courseNumber

current equals current -> left

ELSE

current equal current -> right

RETURN null if match not found

}

**Pseudocode Menu**

void menu{

load courseList  
 int choice

Take choice from user

SWITCH choice

CASE 1

Print course list in alphabetical order

CASE 2

Ask for course number

Call search function, pass course number

CASE 3

Exit Program

DEFAULT

Print invalid entry

}

## Runtime Analysis

Vector Alphabetical Order

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Sortedlist = sort (courseList) | 1 | 1 | 1 |
| FOR iterate through SortedList | 1 | n | n |
| Print course info | 1 | n | n |
|  |  |  |  |
|  |  |  |  |
| **Total Cost** | | | 2n+1 |
| **Runtime** | | | O(n) |

Vector Search and Print

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

Hash Table Search and Print

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| key set equal to hashed courseNumber | 1 | n | n |
| Node is set equal to nodes at key | 1 | n | n |
| IF node is not null and course number is equal to course number | 1 | n | n |
| RETURN node->courseNumber | 1 | 1 | 1 |
| IF node is null or end of table | 1 | n | n |
| RETURN original courseNumber | 1 | 1 | 1 |
| WHILE node isn’t null | 1 | n | n |
| IF node->key is equal to current |  |  |  |
| RETURN node->courseNumber | 1 | 1 | 1 |
| Node is equal to node->next | 1 | 1 |  |
| Return courseNumber | 1 | 1 | 1 |
| **Total Cost** | | | 5n+5 |
| **Runtime** | | | O(n) |

Tree Alphabetical Order

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Create Function BST print, take Node as argument | 1 | n | n |
| IF node is null | 1 | n | n |
| RETURN | 1 | 1 | 1 |
| BST print (node -> left) | n | n | n |
| Print node | 1 | n | n |
| BST print (node -> right) | n | n | n |
| **Total Cost** | | | 5n+1 |
| **Runtime** | | | O(n) |

Tree Search and Print

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| current is equal to tree -> root | 1 | n | n |
| WHILE current is not null | 1 | n | n |
| IF courseNumber is equal to current -> courseNumber | 1 | n | n |
| RETURN current | 1 | 1 | 1 |
| ELSE IF courseNumber is less than current -> courseNumber | 1 | n | n |
| current equals current -> left | 1 | n | n |
| ELSE | 1 | n | n |
| current equal current -> right | 1 | n | n |
| RETURN null if match not found  } | 1 | 1 | 1 |
| **Total Cost** | | | 7n + 2 |
| **Runtime** | | | O(n) |

**Data Structure Analysis**

Vector

Vectors are a data structure that are dynamic, so they are a great choice for data sets that would require constant removal or addition of elements. A vector will take up more memory as a result of this feature however. Vectors can be used by themselves or be implemented into a Hash Table or Linked List. There is also a large library that can be used with vectors such as sorting, which helps streamline code.

Hash Tables

Hash Tables are a data structure that holds data in an unsorted fashion. To access them a key-value system along with a hashing function is used. If it is a perfect hashing function, each bucket will hold one value and searching for a value is incredibly fast. When it is imperfect, techniques like chaining or open addressing are used to prevent value collisions. They are not always the most memory efficient however, since the amount of data held by the table might be less than what is allocated for it, leading to wasted space.

Binary Search Tree

Binary Search Trees are a data structure that by design holds data in a semi-sorted fashion. With the first piece of data acting as the root or starting point, all data to the left of the tree will be smaller than the root, and all the data to the right of the tree will be greater than the root. Compared to a Hash Table, searching a Binary Search Tree will be slower if the Hash Table is perfect, however the Binary Search Tree is more memory efficient since it doesn’t allocated more memory than it needs.

**Recommendation**

The data structure that I would recommend would be a vector, since c++ has library functions built in that will allow for efficient sorting. From the Big O analysis, the three data structures seem to about to equal, and because of the data size there the speed benefit of the Search Tree and the Hash Table wouldn’t be as large of a factor.