**Data Structures and Algorithms Design**

(S1-21\_DSECLZG519)

FIRST SEMESTER 2021-22



Assignment No.: 1

**PS2 - Vehicle Records**

SUBMITTED BY

Varinder Singh - 2021fc04070

Bandaru Raja Sekhar - 2021fc04074

Baskar K - 2021fc04066

**Table of Contents**

[1. Objective](#_heading=h.bacw9fer07q8) **3**

[2. Data Structure used - Binary Tree](#_heading=h.smmi906b8one) **3**

[2.1 Terminologies used in Binary Tree](#_heading=h.siix96s1wyxl) 3

[2.2 Node Structure of the Binary Tree](#_heading=h.yfxjqnpwbhva) 3

[2.3 Justification for using Binary Tree](#_heading=h.pgt4zzy662z2) 3

[3. Operation Details](#_heading=h.umc1xnwn188p) **4**

[4. An alternative way to implement the problem](#_heading=h.ka35rbkegbrs) **5**

# 

# **1. Objective**

To create a system using a binary tree for a warehouse that has a certain number of trucks that transport supplies in and out of the warehouse. Each truck has a unique identifier and a counter to keep a track of how many orders each truck has fulfilled. Whenever a truck moves in/out of the warehouse its unique ID is recorded. The warehouse manager uses the system to answer the below questions:

* Total vehicles that came to the warehouse for work?
* Check specific truck whereabouts
* Number of open, closed and yet to be fulfilled orders
* List of trucks that have moved in/out of the warehouse more than ‘z’ number of times.
* List and number of trucks that have completed their maximum deliveries for the day
* List and number of trucks that are currently in the warehouse and available to deliver supplies

# **2. Data Structure used - Binary Tree**

A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a binary tree has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes. There are three binary tree components. Every binary tree node has these three components associated with it.

* Data
* Pointer to the left child
* Pointer to the right child

## **2.1 Terminologies used in Binary Tree**

* Node– It represents a point in a tree.
* Root– A tree’s topmost node.
* Internal Node – Node having at least one child.
* Depth of a Tree– The number of edges from the tree’s node to the root is.
* Height of a Tree– It is the number of edges from the node to the deepest leaf. The tree height is also considered the root height.

## **2.2 Node Structure of the Binary Tree**

class TruckNode:

def \_\_init\_\_(self, Uid):

self.left = None

self.right = None

self.UId = int(Uid)

self.chkoutCtr = 0

## **2.3 Justification for using Binary Tree**

A binary Tree is implemented when data needs to be organized in hierarchical order. It is useful in any situation where the elements can be compared and organized as per the parent node. In our Vehicle Record system, vehicles moving in/out of the warehouse are organized in a binary tree according to their UID.

In a binary search tree, we start searching from the root of the tree with a specific search key(UID). If the search key is less than the index key of the element then go to the left child. If the search key is greater than the index key of the element then go to the right child. This process continues until the search key matches the desired key or until the end of the tree is found where the element does not exist. The binary tree significantly reduces the number of search operations compared to the same system implemented logically with the help of an array/list.

Therefore, searching is faster in the case of Binary trees. Also, the binary search tree does not require its elements to be stored in a contiguous block of memory. Inserting or deleting a node can be done by simply adjusting some pointers. As we know that each node(TruckNode) is a structure containing data and pointers to its left child and right child.

# **3. Operation Details**

All the operations are implemented by using recursion for vehicles moving in/out of the warehouse in a binary tree. Recursion works well for this type of structure because you can search multiple branching paths without having to include many different checks and conditions for every possibility.

\*\*N is the number of nodes present in the binary tree

\*\*H is the height of the binary tree

| **Name of the Operation** | **Time Complexity** | **The reason why the chosen operations are efficient for the given representation** |
| --- | --- | --- |
| readTruckRec | Average case - Θ(H)  Worst-case - O(N) | This recursive approach allows inserting the new node or updating the old node efficiently with fewer comparisons to reach the desired node. Recursion breaks down the problem into smaller, repetitive problems that have many possible branches and are too complex for an iterative approach. |
| printOrderStatus | O(N) | This method traverses the entire list to get the count of orders of each vehicle as it does not require the status of each vehicle counter. |
| availTrucks | O(N) | This recursive approach traverses the whole binary tree without using additional different checks and conditions. |
| checkTruckRec | Average case - Θ(H)  Worst-case - O(N) | This recursive operation traverses the tree to search nodes with UID with fewer comparisons required to reach the desired node. Recursion breaks down the problem into smaller, repetitive problems that have many possible branches and are too complex for an iterative approach. |
| updateTruckRec | Average case - Θ(H)  Worst-case - O(N) | This recursive approach allows updating a node efficiently with fewer comparisons to reach the desired node. Recursion breaks down the problem into smaller, repetitive problems that have many possible branches and are too complex for an iterative approach. |
| printTruckRec | O(N) | This operation iterates through a class level list populated during the readTruckRec() function whose size is equal to the number of vehicles in the system containing relevant data(chkoutCtr) to count the number of open orders, closed orders and yet to be fulfilled orders. |
| highFreqTrucks | O(N) | This recursive approach traverses the whole binary tree without using additional different checks and conditions. |
| maxDeliveries | O(N) | This recursive approach traverses the whole binary tree without using additional different checks and conditions. |

# **4. An alternative way to implement the problem**

# One alternate way of modelling the problem of Vehicle Records moving in and out of the warehouse would be using an array or a list data structure. In this case, a vehicle can be allocated to an array index position where its count of moving in and out can be updated accordingly.

One of the greatest advantages of using arrays is that element access(or searching an element) can be done in constant time. For example, if you want to get the fifth element from an array, you take the starting memory address and add (index \* element\_size). If our memory address starts at 0 and this is an array of 8-bit characters, the fifth element would be at memory address 0+(4\*8) = 32. It doesn’t matter how many elements are in the array, this access time remains constant. If we were to use a BST in the same scenario there would be no way to get to the fifth element without starting at the root node and traversing to the fifth element; which includes a number of comparisons from the root node to the desired node. Also, since the binary tree is not a single-dimensional construct, you would need to define exactly what you mean by the “fifth” element.

Also, if your data is already in array form (maybe it was returned from the database), it is possible that the time it would take to set up the binary tree; the first time may take longer than the worst-case search of that array. In addition, there is no problem with getting an “unbalanced tree” in the case of arrays.

However, if we need to insert a new vehicle within the existing array/list; in that case, it will increase the number of operations to carry out this insertion. Firstly, we need to increase the array size by 1 and insert the new vehicle at the desired location and streamline the other present vehicles in the array. The same can be said for the deletion of an old vehicle, where we need to delete the vehicle at the location and streamline the other vehicles in the array then decrease the array size.