1. INTRODUCTION

The second coursework of this CST2550 Software Engineering Management and Development, aimed at the designing and implementation of a parking management system in C++. Since the system ought to allow the user to store the vehicle registration number, the date and entry time, a hash table data structure having a specific function to deal with possible collisions, was used. Additionally, since the operator of the program should be able to access a report history of the vehicles that parked on a particular date with their entry and exit times as well as charged price, a sorted single linked list data structure was implemented. The choice of these two data structures is justified in the ensuing sections of this report as an analysis of their algorithms is done.

The report of this coursework in divided in different sections. The first section is an introduction to the coursework 2. The justification of the choice of the appropriate data structures and analysis of the data structures and their algorithms are explored in section 2. In section 3, the pseudocode is inspected, and the time complexity of the functions are calculated. Section 4 will conclude the report, giving a summary before reflecting on the limitations and further improvements that could be brought to this program. Finally, the last section will include the references used when researching the appropriate data structure.

2. APPROPRIATE SELECTION OF DATA STRUCTURE AND ANALYSIS OF DATA STRUCTURE AND ALGORITHMS

In this case scenario, we must choose an appropriate data structure to store vehicles in a car park system. In a parking lot, cars come and go at different times and remain parked for different periods of times. Moreover, a parking lot has limited space for the number of vehicles it can house. Thus, the data structure I opted for the storing of vehicles is a Hash Table.

This data structure allows the user to enter and store a vehicle based on the vehicle registration number. It also enables the user to store the date and entry time of the vehicle. This data structure is ideal for this system as although searching for an element in a hash table can take as long as searching for an element in a linked list— O(n) time in the worst case— in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is O(1) (Cormen, Thomas H., et al., pg 253).

Moreover, when the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is computed from the key (Cormen, Thomas H., et al. pg253). In a hash table, the memory it occupies increases as the number of keys increase which would render using direct addressing impractical.

To store for the history, on the other hand, a linked list data structure is used. Since the report for the history should allow the user to retrieve all the vehicle information for a specific date, a sorted data structure ought to be used. Since a sorted linked list data structure is one which stores its elements linearly, it will be easier to retrieve information by the user. Also, since the nodes are all interconnected, they will store vehicle data on the same date in grouped blocks.

3. Pseudocode of Program

3.1 Hash Table

The figure below demonstrates the time complexity of insertion, searching and deletion of items in a hash table:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

FIGURE 1 - TIME COMPLEXITY OF HASH TABLE

Hashing Function

The calculation of the worse complexity is carried out as follows;

line	function	cost	frequency
	hash()		
	{		
1	for (int i = 0; i < tableSize; i++)	C_1	$n + 1$
	{		
2	HashTable[i] = new vehicle;	C_2	n
3	HashTable[i]->date = "empty";	C_3	n
4	HashTable[i]->VRN = "empty";	C_4	n
5	HashTable[i]->time_in = "empty";	C_5	n
6	HashTable[i]->next = NULL;	C_6	n
	}		
	}		

```
FUNCTION Hash(key)
{
    DECLARE key[i]: INTEGER
    DECLARE hash: INTEGER
    DECLARE index: INTEGER
    hash ← 0

    FOR (unsigned i ← 0; i LESS THAN key.length(); i++)
    {
        hash ← hash + key[i];
    }

    index ← hash % tableSize;

    RETURN index;
}
```

FIGURE 2 - PSEUDOCODE OF HASH FUNCTION

Time Complexity: $(n + 1) + n + n + n + n + n$

$$= 1 + 6n$$

Therefore, degree of n is 1, thus time complexity if big $O(n)$

Adding to Hash Table

line	function	cost	frequency
	addVehicle(std::string date, std::string VRN, std::string time_in)		
	{		
1	int index = Hash(VRN);	C_1	1
2	if (HashTable[index]->VRN == "empty")	C_2	n
	{		
3	HashTable[index]->VRN = VRN;	C_3	n
4	HashTable[index]->date = date;	C_4	n
5	HashTable[index]->time_in = time_in;	C_5	n
	}		
6	else	C_6	n
	{		
7	vehicle *Ptr = HashTable[index];	C_7	n
8	vehicle *n = new vehicle;	C_8	n
9	n->VRN = VRN;	C_9	n
10	n->date = date;	C_{10}	n
11	n->time_in = time_in;	C_{11}	n
12	n->next = NULL;	C_{12}	n
13	while (Ptr->next != NULL)	C_{13}	$n + 1$
	{		
14	Ptr = Ptr->next;	C_{14}	n
	}		
15	Ptr->next = n;	C_{15}	n
	}		
16	std::cout << "\tVehicle having registration number " << VRN << " added to parking lot." << std::endl;	C_{16}	n
	}		

```
FUNCTION addVehicle(date, VRN, time_in)
{
    int index ← Hash(VRN);

    IF (HashTable[index]->VRN EQUAL TO "empty")
    {
        HashTable[index]->VRN ← VRN;
        HashTable[index]->date ← date;
        HashTable[index]->time_in ← time_in;
    }
    ELSE
    {
        vehicle POINTER Ptr ← HashTable[index];
        vehicle POINTER n ← new vehicle;
        n->VRN ← VRN;
        n->date ← date;
        n->time_in ← time_in;
        n->next ← NULL;
        WHILE (POINTER TO NEXT ITEM NOT NULL)
        {
            Ptr ← Ptr->next;
        }
        Ptr->next ← n;
    }
    OUTPUT "Vehicle having registration number " << VRN << " added to parking lot."
}
```

FIGURE 3 - PSEUDOCODE OF FUNCTION ADDING ITEM TO HASH TABLE

Time Complexity: $1 + n + n + n + n + n + n + n + n + n + n + n + (n + 1) + n + n + n$

$$= 2 + 15n$$

Therefore, degree of n is 1, thus time complexity if big $O(n)$

SEARCHING FROM HASH TABLE

line	function	cost	frequency
	findVehicle(std::string VRN)		
	{		
1	int index = Hash(VRN);	C_1	1
2	bool foundVRN = false;	C_2	1
3	std::string date;	C_3	1
4	std::string time_in;	C_4	1
5	vehicle *Ptr = HashTable[index];	C_5	1
6	while (Ptr != NULL)	C_6	$n + 1$
	{		
7	if (Ptr->VRN == VRN)	C7	n
	{		
8	foundVRN = true;	C8	n
9	date = Ptr->date;	C9	n
10	time_in = Ptr->time_in;	C10	n
	}		
11	Ptr = Ptr->next;	C11	n
	}		
12	if (foundVRN == true)	C12	n
	{		
13	std::cout << "\tVehicle having registration number " << VRN << " parked on the " << date << " at " << time_in << std::endl;	C13	n
	}		
14	else	C14	n
	{		
15	std::cout << "\tNo vehicle with this registration number currently parked here." << std::endl;	C15	n
	}		
	}		

```
FUNCTION findVehicle(VRN)
{
    int index ← Hash(VRN);

    bool foundVRN ← false;
    DECLARE date: STRING
    DECLARE time_in: STRING

    vehicle Pointer Ptr ← HashTable[index];

    WHILE (Ptr NOT EQUAL TO NULL)
    {
        IF (Ptr->VRN EQUAL VRN)
        {
            foundVRN ← true;
            date ← Ptr->date;
            time_in ← Ptr->time_in;
        }
        Ptr ← Ptr->next;
    }
    IF (foundVRN EQUAL TO true)
    {
        OUTPUT "Vehicle having registration number ", VRN," parked on the ",
date, " at ", time_in
    }
    ELSE
    {
        OUTPUT "No vehicle with this registration number currently parked
here."
    }
}
```

FIGURE 4 - PSEUDOCODE OF FUNCTION TO SEARCH ITEM FROM THE HASH TABLE

Time Complexity: $1 + 1 + 1 + 1 + 1 + (n\ +\ 1) +\ n + n + n + n + n + n + n + n + n$

$$= 6 + 10n$$

Therefore, degree of n is 1, thus time complexity if big $O(n)$

REMOVING FROM HASH TABLE

line	function	cost	frequency
	removeVehicle(std::string VRN)		
	{		
1	int index = Hash(VRN);	C_1	1
2	vehicle *delPtr;	C_2	1
3	vehicle *P1;	C_3	1
	vehicle *P2;		
4	if (HashTable[index]->VRN == "empty" && HashTable[index]->date == "empty")	C_4	n
	{		
5	std::cout << "\tVehicle having registration number " << VRN << " was not found in the parking lot." << std::endl;	C_5	n
	}		
6	else if (HashTable[index]->VRN == VRN && HashTable[index]->next == NULL)	C_6	n
	{		
7	HashTable[index]->VRN = "empty";	C_7	n
8	HashTable[index]->date = "empty";	C_8	n
9	HashTable[index]->time_in = "empty";	C_9	n
10	std::cout << "\tVehicle having registration number " << VRN << " has successfully exited the parking lot." << std::endl;	C_{10}	n
	}		
11	else if (HashTable[index]->VRN == VRN)	C_{11}	n
	{		
12	delPtr = HashTable[index];	C_{12}	n
13	HashTable[index] = HashTable[index]->next;	C_{13}	n
14	delete delPtr;	C_{14}	n
15	std::cout << "\tVehicle having registration number " << VRN << " has successfully exited the parking lot." << std::endl;	C_{15}	n
	}		
16	else	C_{16}	n
	{		
17	P1 = HashTable[index]->next;	C_{17}	n
18	P2 = HashTable[index];	C_{18}	n
19	while (P1 != NULL && P1->VRN != VRN)	C_{19}	$n + 1$
	{		
20	P2 = P1;	C_{20}	n
21	P1 = P1->next;	C_{21}	n
	}		
22	if (P1 == NULL)	C_{22}	n
	{		
23	std::cout << "\tVehicle having registration number " << VRN << " was not found in the parking lot." << std::endl;	C_{23}	n
	}		
24	else	C_{24}	n
	{		
25	delPtr = P1;	C_{25}	n
26	P1 = P1->next;	C_{26}	n
27	P2->next = P1;	C_{27}	n
28	delete delPtr;	C_{28}	n
29	std::cout << "\tVehicle having registration number " << VRN << " has successfully exited the parking lot." << std::endl;	C_{29}	n
	}		
	}		
	}		

```
Function removeVehicle(VRN)
{
    Declare index: INTEGER
    index ← Hash(VRN);

    vehicle POINTER delPtr;
    vehicle POINTER P1;
    vehicle POINTER P2;

    IF (HashTable[index]->VRN == "empty" AND HashTable[index]->date EQUAL TO
"empty")
    {
        OUTPUT "Vehicle having registration number ", VRN," was not found in
the parking lot."
    }
    ELSE IF (HashTable[index]->VRN EQUAL TO VRN AND HashTable[index]->next
EQUAL TO NULL)
    {
        HashTable[index]->VRN ← "empty";
        HashTable[index]->date ← "empty";
        HashTable[index]->time_in ← "empty";

        OUTPUT "Vehicle having registration number ", VRN, " has successfully
exited the parking lot."
    }
    ELSE IF (HashTable[index]->VRN == VRN)
    {
        delPtr ← HashTable[index];
        HashTable[index] ← HashTable[index]->next;
        DELETE delPtr;

        OUTPUT "Vehicle having registration number ", VRN, " has successfully
exited the parking lot."
    }
    ELSE
    {
        P1 ← HashTable[index]->next;
        P2 ← HashTable[index];

        WHILE (P1 NOT EQUAL TO NULL AND P1->VRN NOT EQUAL TO VRN)
        {
            P2 ← P1;
            P1 ← P1->next;
        }
        IF (P1 EQUAL TO NULL)
        {
            OUTPUT "Vehicle having registration number ", VRN, " was not found
in the parking lot."
        }
        ELSE
        {
            delPtr ← P1;
            P1 ← P1->next;
            P2->next ← P1;

            DELETE delPtr;
            OUTPUT "Vehicle having registration number ", VRN, " has
successfully exited the parking lot."
        }
    }
}
```


FIGURE 5 - PSEUDOCODE TO REMOVE ITEM FROM HASH TABLE

Time Complexity: $1 + 1 + 1 + (n + 1) + n + n + n + n + n + n + n + n + n + n + n + n + n + n + (n + 1) + n + n + n + n + n + n + n + n + n$

$$= 5 + 25n$$

Therefore, degree of n is 1, thus time complexity if big $O(n)$

3.2 Linked List

The figure below demonstrates the time complexity of insertion, searching and deletion of items in a linked list:

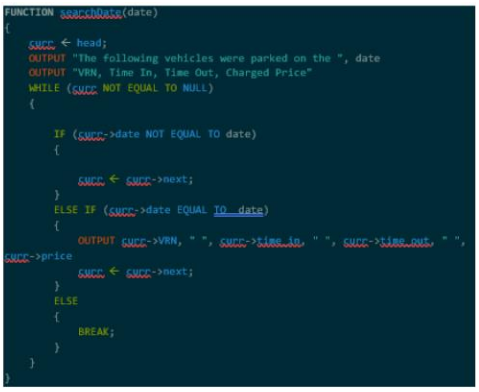
SINGLY LINKED LIST OPERATION	REAL TIME COMPLEXITY	ASSUMED TIME COMPLEXITY
Access i-th element	O(√N * N)	O(N)
Traverse all elements	O(√N * N)	O(N)
Insert element E at current point	O(1)	O(1)
Delete current element	O(1)	O(1)
Insert element E at front	O(1)	O(1)
Insert element E at end	O(√N * N)	O(N)

Sorting and adding to linked list

```
FUNCTION addNode (date, VRN, time_in, time_out, price)
{
    DECLARE flag: BOOLEAN
    nodePtr n ← new node;
    n->next ← NULL;
    n->date ← date;
    n->VRN ← VRN;
    n->time_in ← time_in;
    n->time_out ← time_out;
    n->price ← price;

    if (head NOT EQUAL TO NULL)
    {
        curr ← head;

        flag ← sameDates(n->date, curr->date); //calls function to compare
dates for sorted insertion into linked list

        WHILE (curr->next NOT EQUAL TO NULL && flag EQUAL TO FALSE)
        {
            curr ← curr->next;
        }
        n->next ← curr->next;
        curr->next ← n;
    }
    ELSE
    {
        head ← n;
    }
}
```

FIGURE 6 - FUNCTION SORTING ITEM BEFORE ADDING TO THE LINKED LIST

Searching from hash table



```
FUNCTION searchDate(date)
{
    curr ← head;
    OUTPUT "The following vehicles were parked on the ", date
    OUTPUT "VRN, Time In, Time Out, Charged Price"
    WHILE (curr NOT EQUAL TO NULL)
    {

        IF (curr->date NOT EQUAL TO date)
        {

            curr ← curr->next;
        }
        ELSE IF (curr->date EQUAL TO  date)
        {
            OUTPUT curr->VRN, " ", curr->time_in, " ", curr->time_out, " ",
curr->price
            curr ← curr->next;
        }
        ELSE
        {
            BREAK;
        }
    }
}
```

FIGURE 7 - FUNCTION TO SEARCH ITEM FROM LINKED LIST

REMOVING FROM HASH TABLE

```
FUNCTION deleteNode(delData)
{

    nodePtr delPtr ← NULL;
    temp ← head;
    curr ← head;
    WHILE (curr NOT EQUAL TO NULL AND curr->VRN NOT EQUAL TO delData)
    {
        temp ← curr;
        curr ← curr->next;
    }
    IF (curr EQUAL TO NULL)
    {
        OUTPUT delData, " was not in the list"
        DELETE delPtr
    }
    ELSE
    {
        delPtr ← curr;
        curr ← curr->next;
        temp->next ← curr;
        DELETE delPtr;
        OUTPUT " The vehicle having registration number ", delData "was
deleted from history."
    }
}
```

FIGURE 8 - FUNCTION TO REMOVE ITEM FROM LINKED LIST

4. CONCLUSION

In this project, I made use of a hash table to store vehicles as they entered the parking lot. Their information, specifically their entry date and time along with the vehicle registration number is stored in this data structure. The user can retrieve a vehicle's information based on its vehicle registration number and remove the vehicle from the hash table once the vehicle exits the parking lot.

Once the vehicle exits the parking lot, it is removed from the hash table and inserted into a sorted singly linked list. The linked list was sorted in order for the proper and fast retrieval of all the information of a vehicle on a specific date. This successfully offers a report history of vehicles and their registration number, entry and exit time as well the price payable for the length of time spent in the parking lot.

Other improvements which would be brought to this program include a better user interface which would displace the parking spaces. If the spaces were taken by vehicles or if they were free, they would be displayed. Moreover, the choice of parking spaces could be offered.

The use of hash table is effective as in the best-case scenario it offers a time complexity of O(1) for the insertion, retrieval and deletion of an item from the table. However, hash tables have certain drawbacks such as its static size which is hard coded and risks of collision. If ever the number of items stored in the hash table exceed the declared size of the hash table, resizing might be necessary. Since every element in the current table must be re-hashed back into the database after the resize, it can be an expensive procedure. (Richter et al, pg. 98) To avoid resizing, the hash table's starting size should be large enough to prevent unwanted resizing. Hence, for this project, the parking lot size is assumed to be 50.

As for the linked list, the time complexity is O(n), as it needs to traverse through every node of the list in order to retrieve information for dates that are at the end of the list. A linked list is used since the hashing for dates would render similar values which would cause collision. Double hashing could have been implemented to attend to this problem.

5. References

1. Cormen, Thomas H., et al. Introduction to Algorithms, Third Edition, MIT Press, 2009. ProQuest Ebook Central, Available at: https://ebookcentral.proquest.com/lib/mdx/detail.action?docID=3339142 [Accessed 5 May 2022].
2. Horstmann, C., n.d. *Big C++*. 3rd ed. Wiley[Accessed 5 May 2022]..
3. GeeksforGeeks. 2022. *Hashing Data Structure - GeeksforGeeks*. [online] Available at:

   <https://www.geeksforgeeks.org/hashing-data-structure/> [Accessed 5 May 2022].

4. OpenGenus IQ: Computing Expertise & Legacy. 2022. *Time Complexity Analysis of Linked List*. [online] Available at: <https://iq.opengenus.org/time-complexity-of-linked-list/> [Accessed 5 May 2022].