1 Introduction

This coursework aims to design and implement a tracking system for vehicles in a parking lot using C++. The whole system is broken down into 2 different main subsystems. The first one is to track vehicles entering and exiting the car park. For this subsystem, a balanced binary search tree was used called the Red-Black Tree. On the other hand, to produce a report for a given date along with the vehicle's plate number, entering and exiting time and the parking price, a hash table was used, with a customed hash function and separate chaining to solve collisions in the hash table. To justify the choices of the above data structures, these methods have been analysed. Showing why they are appropriate for the vehicle tracking system. Now, I will explain the different components of the report.

In this section, a brief description of the project was explained. In section 2, the design of the system will be discussed. The reason behind the choice of Red-Black Tree and Separate Chaining will be justified. In addition, section 3, will also contain the analysis of the algorithm's key functionalities using pseudo code. Furthermore, in section 4, the report will be concluded. It will contain a summary of the work done, the limitation of the project and a critical reflection upon the project and how a similar project will be approached in the future. In the final part of the report, section 5 will contain a reference list of the references used to support my arguments in the project.

2.1 Design

The tracking system for a parking lot requires fast adding and removing of vehicles when they are entering and exiting the parking lot and fast adding and getting of vehicles from a relatively large amount of records. According to the Guinness world record (Guinnessworldrecords.com, 2022), the largest car park's capacity is 20,000. So, I am assuming that the data structure to store the vehicles' details in the parking lot should be able to do fast adding and removing for a dataset of a minimum of 20,000 vehicles.

When a vehicle enters the parking lot, it will be added at the end of the vector. So, insertion happens at the end. According to Cay H. and Timothy A. (2009), when using a vector as the data structure and adding to its end is not so costly. That procedure is carried out by the push back member function. This is an O(1) operation for the worst-case scenario. However, when a vehicle exits the parking lot, it needs to be removed from the data structure. It can be at the start, middle or end of the data structure. For vectors, elements with higher index values must be moved to delete an element. Assuming deletion will happen at a random location, on average, each deletion advances n/2 elements, where n is the size of the vector. As a result, vector removal is an O(n) operation which is not ideal for large data sets (Cay H. and Timothy A., 2009).

According to Horstmann Cay S. (2017), although linked lists allow for quick insertion and removal O(1), element access can be slow. The operation of locating the kth element is O(k) which is performing a sequential search is not ideal to check whether a vehicle is in the parking or not when validating an entry in the system for a large data set.

Robert S. and Kevin W. (2011) defined a binary search tree (BST) as a binary tree in which each node has a comparable key and an associated value and the key in any node is greater than the keys in all nodes in that node's left subtree but smaller than the keys in all nodes in that node's right subtree. The vehicle's plate number will be used as the comparable key and its associated value will be the time the latter entered the parking lot.

The execution times of algorithms on binary search trees are determined by the shapes of the trees, which are determined by the order in which keys are placed shown by Robert S. and Kevin W. (2011). A tree with N nodes could be perfectly balanced in the best case, with lg(N) nodes between the root and each null link. In the worst-case scenario, O(N) nodes could be on the search path. In this case, a balanced search tree will be used to make sure that the tree is perfectly balanced to avoid the worst-case scenario.

Horstmann Cay S. (2017) confirmed that in a binary search tree, insertion and removal are O(log(n)) operations if the tree is balanced. They guaranteed that with Red-Black Trees, we can achieve O(log(n)). Since, insertion, searching and removal are important elements in the tracking system, Red-Black Tree will be the chosen algorithm to keep a record of the current vehicle in the parking lot.

In addition, the tracking system also requires fast adding and getting of records from a history data structure. Whenever a car exits the car park, its parking charges will be calculated and then added to the history data structure. The report generated will be for a given date specified in the command line by the user. Assuming that the history data structure should contain data for several years and a minimum of 20,000 vehicles entries per day.

Conen et al (2009) explained that direct-address search, insert and delete operations take only O(1) time which is beneficial for this system. However, they also emphasize the fact that O(1) only happens for a reasonably small dataset which is not the case for this system. In addition, the keys must be integers which is not possible for this scenario as the keys will be the date in the format, DD/MM/YYYY. Hash tables aim to solve these problems. Hashing is a perfect example of a time-space tradeoff (Conen et al, 2009).

According to Robert S. and Kevin W. (2011), hashing-based search algorithms are divided into two components. The first step is to create a hash function that converts the date which is the search key into an array index. Various keys should ideally map to different indices representing the vehicles' details such as the number plate, enter/exit time and the parking price. Because this ideal is generally out of reach, we must accept the chance that two or more different keys hash to the same array index. As a result, the second component of a hashing search is a collision-resolution process that handles this issue. Two different collision resolution approaches will be considered: Linear Probing and Separate chaining.

Linear probing is the simplest open-addressing method that uses associative arrays according to Robert S. and Kevin W. (2011). Horstmann Cay S. (2017), explained that the date will be used to compute the hash code and proceed through the probing sequence until we find a match or an empty slot. This is still an O(1) process as long as the hash table is not too big, but it may need more comparisons than separate chaining. If there are no empty spaces in the probing sequence, the table must be reallocated to a bigger size. On the other hand, only elements with the same hash code are compared when using separate chaining.

In separate chaining, the hashing function will be used to determine all colliding elements and they are then collected in a linked list of elements with the same position value. Such a list is called a "bucket". (Horstmann Cay S., 2017). The worst-case scenario of the linked list, that is the sequential search having an O(n) will not happen since the only operation of the separate chaining algorithm has to add and get a date using the hashing function. Computing the hash table index takes O(1). So, a separate chaining algorithm will be the other data structure to be used in the tracking system.

A little modification will be done to the separate chaining, instead of using a linked list, a vector will be used to store the different vehicles' details to that specific date. When a vehicle exits the parking lot, its exit time will be generated by the system and then, the date will be used to generate the hash table index value to know to which vector, the vehicle should be appended. The latter will be appended to the end of the vector. As per Horstmann Cay S. (2017), adding an element to a vector is not costly: when the insertion happens at the end. This is an O(1) operation.

3. Analysis

3.1 Red-Black Tree

3.1.1 Left Rotate

The arrangement of the nodes on the right is changed into the organization of the nodes on the left in the left rotation. (Programiz.com, 2022)

The worst-case scenario will happen when y has a left subtree. When the latter is not null, x will be assigned to the parent of the left subtree of y. This means that the worst case is 12. Therefore the function leftRotate(T,x) has a **O(1)** operation.

Function leftRotate(T,x)	Cost	Times
y ← x.right	C1	1
x.right ← y.left	C2	1
if y.left **is not equal to** T.Nil	C3	1
y.left.p ← x	C4	[0,1]
y.p ← x.p	C5	1
if x.p **equals** T.nil	C6	1
T.root ← y	C7	[0,1]
elseif x **equals** x.p.left	C8	1
x.p.left ← y	C9	[0,1]
else x.p.right ← y	C10	1
y.left ← x	C11	1
x.p ← y	C12	1
end function		
Worst Case **T(n)** = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8 + C9 + C10 + C11 + C12 = **12**		

Table 1: Left Rotate

3.1.2 Right Rotate

Function rightRotate(T,x)	Cost	Times
y ← x.left	C1	1
x.left ← y.right	C2	1
if y.right **is not equal to** T.Nil	C3	1
y.right.p ← x	C4	[0,1]
y.p ← x.p	C5	1
if x.p **equals** T.nil	C6	1
T.root ← y	C7	[0,1]
elseif x **equals** x.p.right	C8	1
x.p.right ← y	C9	[0,1]
else x.p.left ← y	C10	1
y.right ← x	C11	1
x.p ← y	C12	1
end function		
Worst Case **T(n)** = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8 + C9 + C10 + C11 + C12 = **12**		

Table 2: Right Rotate

The arrangement of the nodes on the left is changed into the arrangement of the nodes on the right in the right rotation. (Programiz.com, 2022)

The worst-case scenario will happen when y has a right subtree. When the latter is not null, x is assigned to the parent of the right subtree of y. This means that the worst case is 12. Therefore the function rightRotate(T,x) has a **O(1)** operation.

3.1.3 Insert

When a new node is inserted, it is always inserted as a RED node. This is because inserting a red node does not break a red-black tree's depth property. If you join a red node to another red node, the rule is broken, but it is easier to fix than the problem caused by breaking the depth attribute. (Programiz.com, 2022)

The worst-case scenario will happen when the while loop runs that is when x is not equal to null. The while loop will keep dividing the BST in half each time to know which subtree (right or left) to choose. This means that the worst case is 10 + 4log(n). Therefore it is an **O(log(n))** operation.

If the tree violates the properties of the red-black tree after the insertion of a new node, we do the following activities. (Programiz.com, 2022)

1 Recolour
2 Rotation

Function insert(T,z)	Cost	Times
y ← T.nil	C1	1
x ← T.root	C2	1
while x **is not equal to** T.nil	C3	[0, log(n)]
y ← x	C4	[0, log(n)-1]
if z.key **is less than**	C5	[0, log(n)-1]
x.key	C6	[0,1]
x ← x.left	C7	[0, log(n)-1]
else x ← x.right	C8	1
z.p ← y	C9	1
if y **equals to** T.nil	C10	[0,1]
T.root ← z	C11	1
elseif z.key **is less than** y.key	C12	[0,1]
y.left ← z	C13	[0,1]
else y.right ← z	C14	1
z.left ← T.nil	C15	1
z.right ← T.nil	C16	1
z.color ← RED	C17	1
InsertFixup(T,z)		
end function		
Worst Case **T(n)** = C1 + C2 + C3log(n) + C4(log(n)-1) + C5(log(n)-1) + C6 + C7(log(n)-1) + C8 + C9 + C10 + C11 + C12 + C13 + C14 + C15 + C16 + C17 = 1 + 1 + log(n) + log(n) – 1 + log(n) – 1 + 1 + log(n) – 1 + 1 + 1 + 1 + 1 +1 + 1 + 1 + 1 + 1 + 1 = **10 + 4log(n)**		

Table 3: Insert in Red-Black Tree

3.1.4 Search

This function is implemented to search for a specific key and when found, return it.

The worst-case scenario will happen when the while loop runs that is when the root is not equal to null. The while loop will keep dividing the BST in half each time to know which subtree (right or left) to choose. This means that the worst case is 4log(n). Therefore it is an **O(log(n))** operation.

Function search(T, key)	Cost	Times
root ← T.root	C1	1
while root **is not equals to** T.Nil	C2	[0, log(n)-1]
if root.key **is equals to** key	C3	[0, log(n)-1]
z ← root	C4	[0,1]
if root.key **is less than** key	C5	[0, log(n)-1]
root ← root.right	C6	[0,1]
else root ← root.left	C7	[0, log(n)-1]
return z	C8	1
end function		
Worst Case **T(n)** = C1 + C2(log(n)-1) + C3(log(n)-1) + C4 + C5(log(n)-1) + C6 + C7(log(n)-1) + C8 = 1 + log(n) – 1 + log(n) – 1 + 1 + log(n) – 1 + 1 + log(n) – 1 + 1 = **4 log(n)**		

Table 4: Search in Red-Black Tree

3.1.5 Delete

This operation is used to delete a node from the tree. The red-black property is restored after deleting a node. (Programiz.com, 2022)

The worst-case scenario will happen when the while loop runs that is when the root is not equal to null. The while loop will keep dividing the BST in half each time to know which subtree (right or left) to choose. This means that the worst case is 4log(n) + 16. Therefore it is an **O(log(n))** operation.

DeletefixUp(x) is used when a black node is destroyed because it violates the red-black tree's black depth property. This violation is fixed by assuming that node x (which is at the original position of node y) has an extra black. As a result, node x is neither red nor black. It comes in either double black or black-and-red. (Programiz.com, 2022)

Function delete(T, z)	Cost	Times
root ← T.root	C1	1
while root **is not equals to** T.Nil	C2	[0, log(n)-1]
if root.key **equals to** z	C3	[0, log(n)-1]
z ← root	C4	[0,1]
if root.key **less or equals to** z	C5	[0, log(n)-1]
root ← root.right	C6	[0,1]
else root ← root.left	C7	[0, log(n)-1]
color ← z.color	C8	1
if z.left **equals to** T.Nil	C9	1
x ← z.right	C10	[0,1]
transplant(z,x)	C11	[0,1]
elseif z.right **equals to** T.Nil	C12	1
x ← z.left	C13	[0,1]
transplant(z, x)	C14	[0,1]
else		
y ← minimum(z.right)	C15	[0,1]
color ← y.color	C16	[0,1]
x ← y.right	C17	[0,1]
if y.p **equals to** z	C18	1
x.p ← y	C19	[0,1]
else transplant(y, y.right)	C20	[0,1]
transplant(z, y)	C21	[0,1]
y.color ← color	C22	[0,1]
if color **equals to** BLACK	C23	1
deleteFixup(x)	C24	[0,1]
end function		
Worst Case **T(n)** = C1 + C2(log(n)-1) + C3(log(n)-1) + C4 + C5(log(n)-1) + C6 + C7(log(n)-1) + C8 + C9 + C10 + C11 + C12 + C13 + C14 + C15 + C16 + C17 + C18 + C19 + C20 + C21 + C22 + C23 + C24 = 1 + log(n) – 1 + log(n) – 1 + 1 + log(n) – 1 + 1 + log(n) – 1 + 17 = **4log(n)+16**		

Table 5: Delete in Red-Black Tree

3.2 Separate Chaining

3.2.1 Hash Analysis

The hash function is determined by the key type. Strictly speaking, each key requires a different hash function. Since the key will be the date in a string data type, a strong hashing function will be used based on the 32-bit djb2 hash by Daniel J. Bernstein. On a sliding window of size 7, a 32-bit djb2 hash is computed. At each phase, the least significant t bits of the hash (the trigger) are evaluated, and if they are all set to 1, a context discovery is announced; this is the critical parameter that distinguishes the various levels of resolution. The default value for the lowest level 0 is 8. (Roussev, V., Richard, G.G. and Marziale, L., 2007). The 32-bit djb2 hash is the most commonly used hashing function and hence it will help to generate a more unique hash index, which will eventually help to increase the performance and efficiency of the separate chaining data structure.

The hash function will loop through all the letters in the key(date) and calculate the hash code. The hashCode will be then modded by the size of the table to make sure the hash index falls between 0 and M, where M is the size of the hash table. The Hash() function will be an **O(n)** operation.

Function hash(key)	Cost	Times
hashCode ← 5381	C1	1
For each x **in** key	C2	n
hashCode ← ((hashCode << 5) + hashCode) + x	C3	n – 1
return hashCode **mod** size	C4	1
end function		
Worst Case		
T(n) = C1 + C2(n) + C3(n-1) + C4 = 1 + n + n – 1 + 1 = **2n + 1**		

Table 6: Hash Function

3.1.2 Hash Quality Analysis

In the diagram below, figure 1 shows the frequency of hash values based on the djb2 hashing function. This table's dataset was generated from a novel called War and Peace. A function loops through each line in the text file and makes use of regex to split the line into words. Then the word is passed to the djb2 hashing function to generate the hash index. The hash index is then compared to check if already been generated before, if yes, the hash index count is incremented, else, one is assigned to the hash index count.

The orange line shows n/m (**load factor**), which is an acceptable number for the average frequency for each hash value in general. When utilizing the djb2 hash function, the majority of the results are near to the load factor, as seen in Figure 1. To avoid collisions, a balance between the value (hash table size) and the n value must be struck (number of entries).

A good hash function will have lower frequencies for each hash value, reducing the frequency of collisions and aiding in the preservation of **0(1)** complexity.

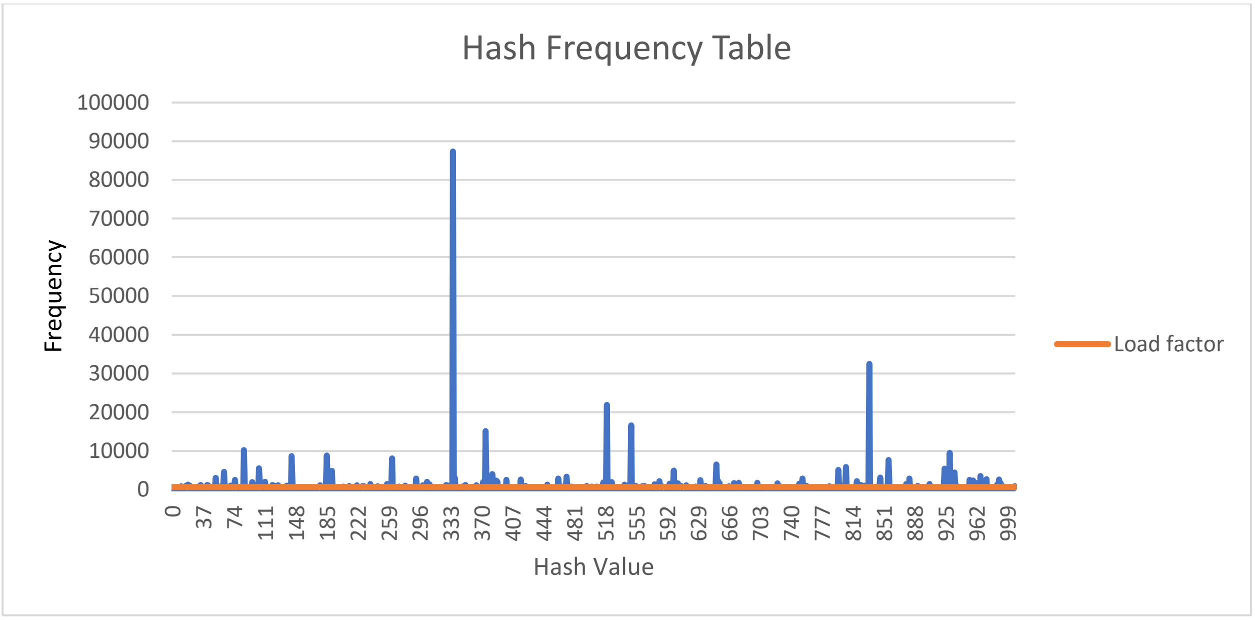

Figure 1: Hash Frequency Table

3.1.3 Insert

The insert function will be used when storing the history of an exiting vehicle from the parking lot. The hash index will be determined first bypassing the key which is the date to the hash function. Then, the generated hash index will be the key of the hash table to which vector the vehicle will be appended to. For this function, both the best case and the worst case will have a complexity of 1. Since the worst case is more important, insert(x, key) will have an **O(1)** operation.

Function insert(x, key)	Cost	Times
index ← hash(key)	C1	1
list[index].push_back(x)	C2	1
Worst Case		
T(n) = C1 + C2 = 1 + 1 = **2**		

Table 7: Insert in Separate Chaining

3.1.4 Search

The search function will be executed when the user is trying to find the list of vehicles for a specific date. The worst-case scenario will happen when the list is not empty, and it will loop through the vector containing the list of vehicles. It will be an **O(n)** operation.

Function search(date)	Cost	Times
index ← hash(date)	C1	1
for each i **in** list[start] : list[index].size()	C2	n
returnString = list[index][i]	C3	n – 1
return returnString	C4	1
Worst Case		
T(n) = C1 + C2n + C3(n – 1) + C4 = 1 + n + n – 1 + 1 = **2n + 1**		

Table 8: Search in Separate Chaining

4 Conclusion

In this coursework, a tracking system for a parking lot was implemented. The whole system depends on two data structures. The red-black tree was used to store the vehicle which is currently in the parking lot. The other data structure is a separate chaining hash table. A little modification was done to the separate chaining, instead of a linked list, a vector is used. This data structure will store the details of all exit vehicles. When adding to this data structure, the date will be used by the hashing function to generate the hash table index value. When the user chooses a specific date, the hashing function will determine the index by hashing the date and then it will loop through the vector to display the vehicle's details for that particular date. These two data structures were chosen after thorough research and analysis to make sure they aimed to provide fast and efficient functionality for the tracking system. The design and the analysis of the data structures were explained in this report.

4.1 Limitations and critical reflection

One limitation of this design is that the separate chaining data structure will only store the details of exit vehicles. When the software starts execution, it will read through the input text file. Every line which does not end with a "-" will be added to the separate chaining data structure. In addition, only when the user enters the exiting time for a specific vehicle, the latter will be removed from the red-black tree data structure and appended to the separate chaining data structure. So, the user will not be able to view a report about vehicles currently in the parking lot.

Another limitation is that the hashing function will use the date which is a string to generate the hash table index value. I had to use a quite large range (1009) to make sure that there is no duplicate value for two different dates. So when the djb2 hashing function generates a value between 0 and 1009, I have to create a hash table of size 1009 which takes up spaces in the memory since there will be some empty locations. It should be noted that the default size is 1009, and at the start, there will be a function to determine the capacity of the hash table depending on the input file and if it is greater than 1009, the next prime number relative to the capacity generated will be used.

4.2 Future project approach

The first limitation is that the user is not able to view reports about current vehicles in the car park. This can be avoided in future projects by being more interactive with the client whether he/she needs to view the details about current vehicles or not. This will enable the development of the software according to the need of the client and not add features that are not required that can take up memory unnecessary.

A more efficient hashing function can be used to generate the hash index for the date and a function to resize the hash table can be added to prevent empty locations in the hash table from taking up unnecessary memory locations. In addition, the separate chaining hash table's capacity can be generated by making use of the linear probing to store the date and the number of times that particular date happened when looping through the file at the start. Once we have the exact total number of dates, we just add 1 to it to store the details for today's date. Then we just use the capacity generated to construct the separate chaining hash table with more exact size.

5 Reference List

Cay H. and Timothy B. (2009) *Big C++.* 2nd edition. United States: Wiley.

Guinnessworldrecords.com. (2022). [online] Available at: https://www.guinnessworldrecords.com/world-records/largest-car-park [Accessed 5 Apr. 2022].

Horstmann, Cay S. (2017). *Big C++, Enhanced eText.* 3rd edition. United States: Wiley.

Linkedin.com. (2022). [online] Available at: https://www.linkedin.com/learning/programming-foundations-data-structures-2/understand-data-structures?autoplay=true&u=42408908 [Accessed 13 Apr. 2022].

Programiz.com. (2022). *Red-Black Tree*. [online] Available at: https://www.programiz.com/dsa/red-black-tree [Accessed 9 Apr. 2022].

Robert S. and Kevin W. (2011) *Algorithms.* 4th edition. Westford: Courier

Roussev, V., Richard, G.G. and Marziale, L. (2007) 'Multi-resolution similarity hashing', *Digital Investigation,* 4, pp. 105-113. doi: https://doi.org/10.1016/j.diin.2007.06.011.