SUNWAY UNIVERSITY

School of Engineering and Technology

DEPARTMENT OF COMPUTING AND INFORMATION SYSTEMS

ASSIGNMENT FOR YEAR 2; BSC (HONS) COMPUTER SCIENCE, BSC (HONS) INFORMATION TECHNOLOGY, BSC (HONS) INFORMATION TECHNOLOGY (COMPUTER NETWORKING AND SECURITY) AND BACHELOR OF SOFTWARE ENGINEERING (HONS).

ACADEMIC SESSION: APRIL – JULY 2021

CSC2103: DATA STRUCTURES & ALGORITHMS

DEADLINE: 7 JULY 2021, 11.59pm, Wednesday

INSTRUCTIONS TO CANDIDATES

1. This assignment contributes **20%** to your final grade.
2. This is a group assignment of maximum Three members.
3. The coursework must be entirely your own work. Please make sure that you are aware of the rules concerning plagiarism.

IMPORTANT
The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.

- Coursework submitted after the deadline but within 1 week will be accepted for a maximum mark of 40%.
- Work handed in following the extension of 1 week after the original deadline will be regarded as a non-submission and marked zero.

Academic Honesty Acknowledgement

I Eltigani Hamadelniel Elfatih (name) 17087610 (ID), I Saurabh Varughese M. Kovoor (name) 20017604 (ID),

I Muhammad Mahad Niaz (name) 19053131 (ID) and I ……………………(name)……………………(ID).

verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment.

[illegible] 6/7/2021, SVMK 6/7/2021, mahad 6/7/2021, ……………………/
(Signature / Date)

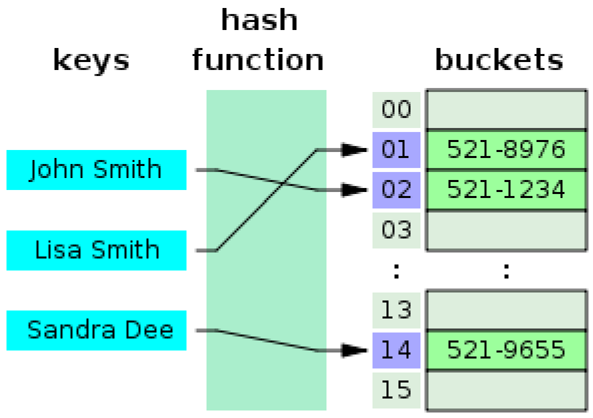
Table of Contents

Problem 1: Hashing

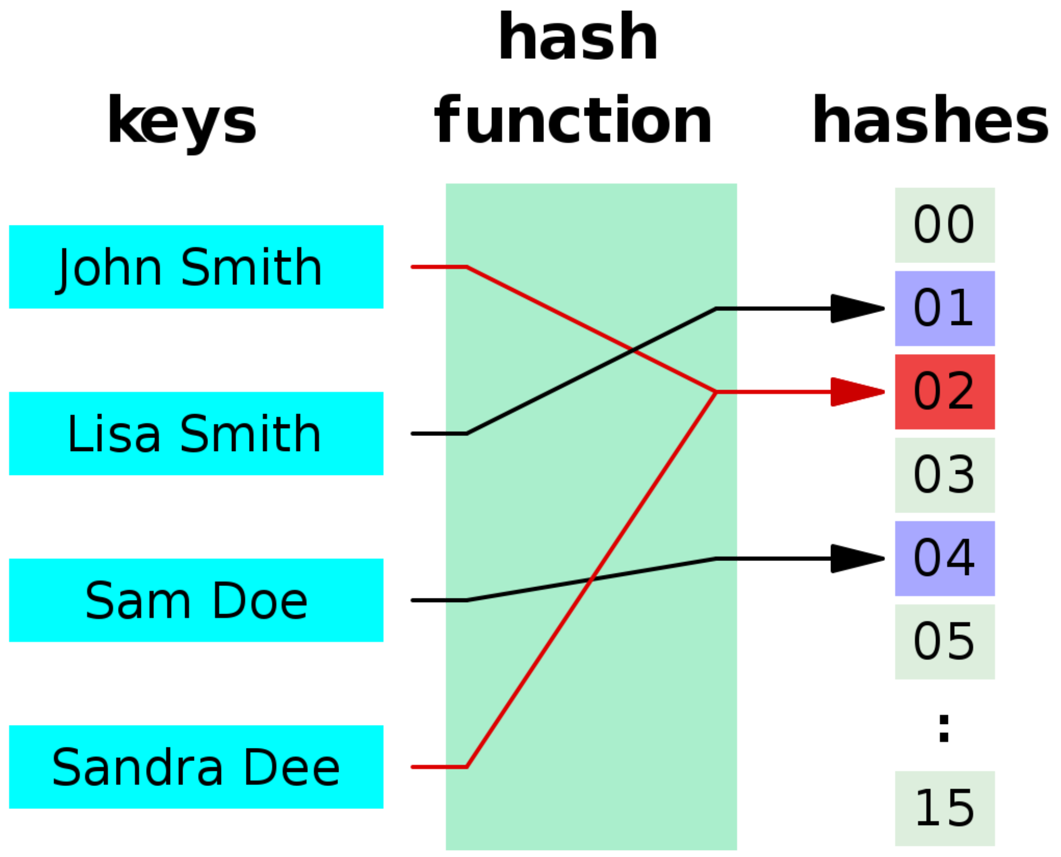
Introduction to Hashing

Overview of Hashing

Traditional data structures and algorithms with the purpose of storing and accessing data such as Arrays and linked lists become very inefficient when working with large datasets, this is because their processing time or time complexity exponentially increases with data size with a time complexity of O(log n) or O (n) [1]. Therefore a technique or algorithm is required in order to overcome these limitations. Hashing is an efficient algorithm or technique that allows us to insert, update, retrieve and delete data entries at a constant time. This means the time complexity or the running time of the algorithm is O(1) and does not depend on the data size n when performing operations [1]. Hashing comes in many forms and is applied across multiple different domains from data storage and retrieval to cryptocurrency to password encryption. While each hashing function differs according to the domain in which it is applied to, they all maintain the main principle of mapping a string of bits of arbitrary length to a fixed-length bit string through a function, known as the hashing function [2]. The mapping of these bits can be described as a relation between two sets forming a set of pairs, which each pair is of the form (key, value). This means for any given key we can find a value that is an output of some hash function that maps keys to values, these values are known as hash values or a hash [1]. Essentially we can think of an array as a Map where the hash value is the index that helps point to the value being stored at that index or looked up at that index.



The concept of a hash table is a generalized idea of an array or any data structure in which the index of a value to be stored regardless of its type can be calculated through a hash function [1]. Furthermore, when the same hash function generates the same index for two separate objects it causes a clash, this clash is known as a collision. This collision may occur due to the similarity in nature of the objects for which a hash value is generated. It is essential that hashing functions are implemented with features or techniques that help to deal with occurrences of collisions. Examples of collision resolution techniques are: Linear probing, Quadratic probing, double hashing and separate chaining.



Collision occurs between hashes of Lisa smith and Sandra Dee

Data Structures, Tools, and Programming Language Features Used

In the spirit of simplicity in our implementation and demonstration of hashing we have decided to use arrays as our data structure of choice. In addition, we use some of the features of the java.util.array library as well as the (.hashCode) feature from the native java string library.

This report will be taking a deeper dive into the concept of hashing and providing a demonstration of its key concepts.

Implementation

For this Hashing program we created 2 java classes:

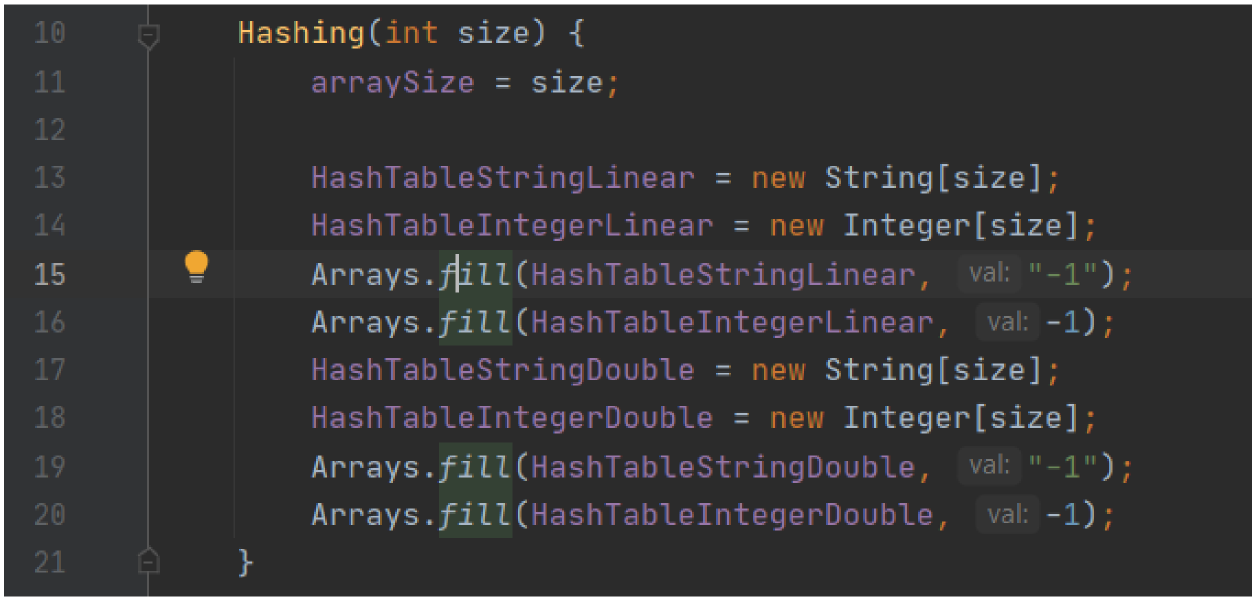
1. The main driver class, Main, instantiates the objects and calls the methods of the Hashing class.
2. The hashing class, Hashing, holds the constructors for creating the String or Integer array which represents the hashing table for the 2 separate data types, functions to insert data into the hashtable with solutions for hashtable collisions, namely linear probing and double hashing, function to search keys in the hash tables and display a representation or visualisation of the hashtables.

1. Generating Hash Tables

The first part of the program or the hashing class, is to instantiate the hash table which will hold the inserted keys at their respective hashing index or values. For this, we created 4 fixed-size arrays that signify the hash tables with a specific arraysize, for this case 40. The following are 4 types of hashtables we created:

1. To store integer values/entries/keys using the linear probing collision resolution.
2. To store string values using/entries/keys using the linear probing collision resolution.
3. To store integer values/entries/keys using the double hashing collision resolution.
4. To store string values using/entries/keys using the double hashing collision resolution.

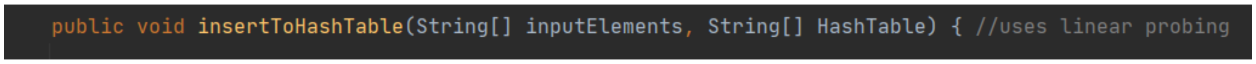
After instantiating the 4 arrays, initially, we fill all the cells of each table with the value -1, which signifies an empty hash table cell. For the Integer tables, this is in integer whereas for string tables this is written as a string or text. We opted for the -1, since we're accepting positive integer keys as input values, hence -1 wouldn't be inputted and hence it's an outlier and wouldn't interfere with the data insertion and searching process.



```java
    Hashing(int size) {
        arraySize = size;

        HashTableStringLinear = new String[size];
        HashTableIntegerLinear = new Integer[size];
        Arrays.fill(HashTableStringLinear, val: "-1");
        Arrays.fill(HashTableIntegerLinear, val: -1);
        HashTableStringDouble = new String[size];
        HashTableIntegerDouble = new Integer[size];
        Arrays.fill(HashTableStringDouble, val: "-1");
        Arrays.fill(HashTableIntegerDouble, val: -1);
    }
```

2. Inserting Keys

To insert the keys into the table, there are 4 different functions for each of the 4 different types of hash tables as outlined above. Although catering for different types of array value data types and collision resolutions, each of The functions have a similar flow. All the function accepts 2 parameters which includes an array of the input values and the array of the instantiated hashtable from before.

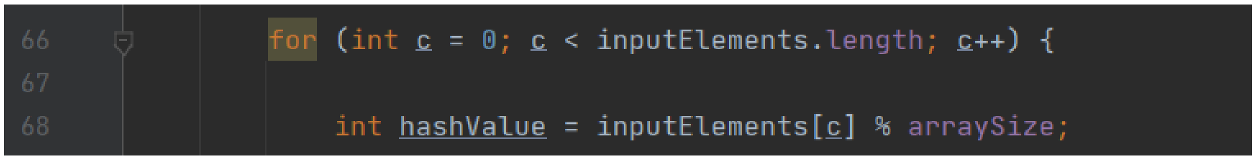


```java
public void insertToHashTable(String[] inputElements, String[] HashTable) { //uses linear probing
```

After that, we begin with the hashing function, where it differs for both integer and string hashing:

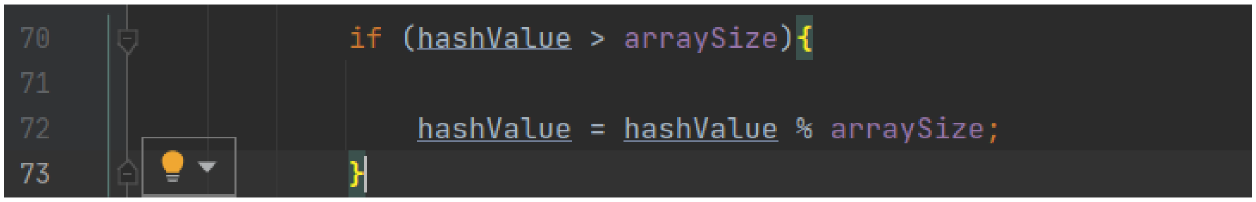
a) Inserting Integer Keys into a HashTable

For inserting keys with the integer data type, it's more straightforward than string hashing, so it was developed first. For this, we start with a for loop which loops through the inputted elements or integers, and using its hash function, which generates a hash value or index, by taking the remainder of the inputted integer and the size of the array or hashtable. Therefore, we use the typical modulo operator % or function to reduce the large input integer value into a range of values that's smaller and can be used as the hash table index.



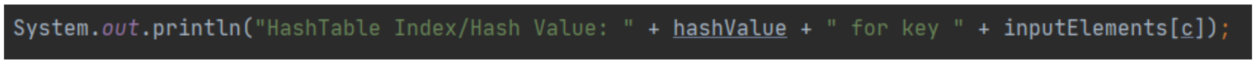
```java
        for (int c = 0; c < inputElements.length; c++) {

            int hashValue = inputElements[c] % arraySize;
```

We check to see if the calculated hash index value exceeds the size of the array or the hashtable, thus this could otherwise result in a “index out of bounds" error. Then we return the hash value to a smaller value by using the modulo operator.



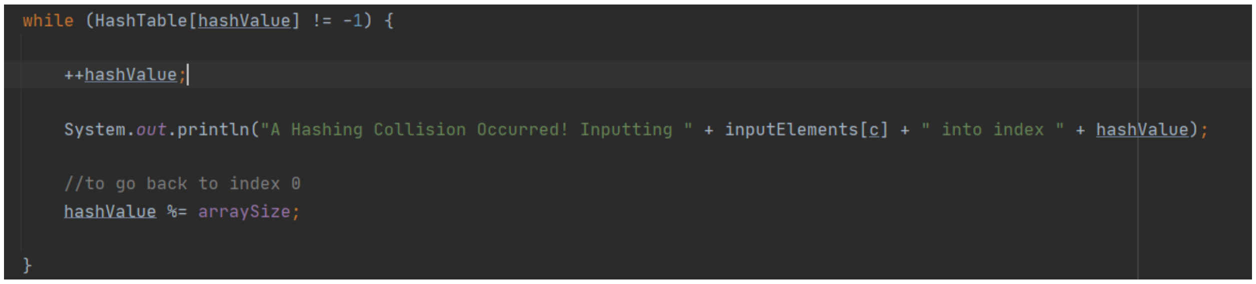
```
if (hashValue > arraySize){

    hashValue = hashValue % arraySize;
}
```

Then, the program will indicate to the user this key and its corresponding computed hash index.



```
System.out.println("HashTable Index/Hash Value: " + hashValue + " for key " + inputElements[c]);
```

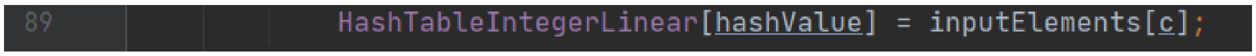
If a collision is detected the program will indicate it, and according to the type of insertion function and its collision resolution employed (linear probing/double hashing) it will indicate whether a collision has occurred, and for what key and where the program is inputting it next.



```
while (HashTable[hashValue] != -1) {

    ++hashValue;

    System.out.println("A Hashing Collision Occurred! Inputting " + inputElements[c] + " into index " + hashValue);

    //to go back to index 0
    hashValue %= arraySize;

}
```

We’ll explain more on this collision resolution in the following sections.

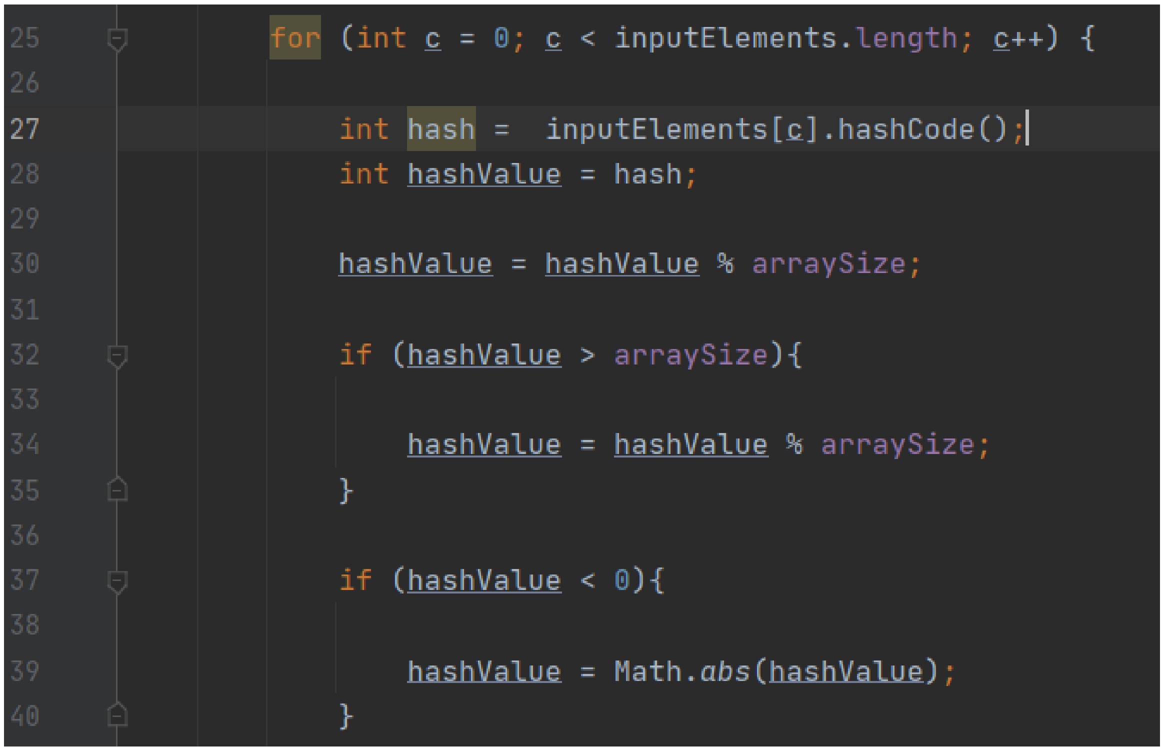
If there’s no existing keys in that hashtable index, then the program will assign the input element key to that hash value index of the hash table.



```
HashTableIntegerLinear[hashValue] = inputElements[c];
```

b) Inserting String Keys into a HashTable
For inserting keys which have a Strings datatype into the hashtable, we’ll first need to convert it into an integer data type first so that it can undergo hashing and be assigned to a hash index which is an integer data type.

For this, the method we employed is by taking the hashcode value of the string input, with the hasCode() function which returns the integer value of the string. Then, like for the hashing insertion function for integers, we take the remainder of this hashcode value and the array size as the string key’s hash value or index. We check to see if the calculated hash value is larger than the arraysize to prevent any “index out of bounds“ error, and if so, we perform the hash function again where we take the remainder of the hashValue and the array size. Also the hash index can’t be a negative integer or less than 0, so we check for that, and if so, we use a modulus or absolute function (Math.abs()) to take its positive value as the hash index.



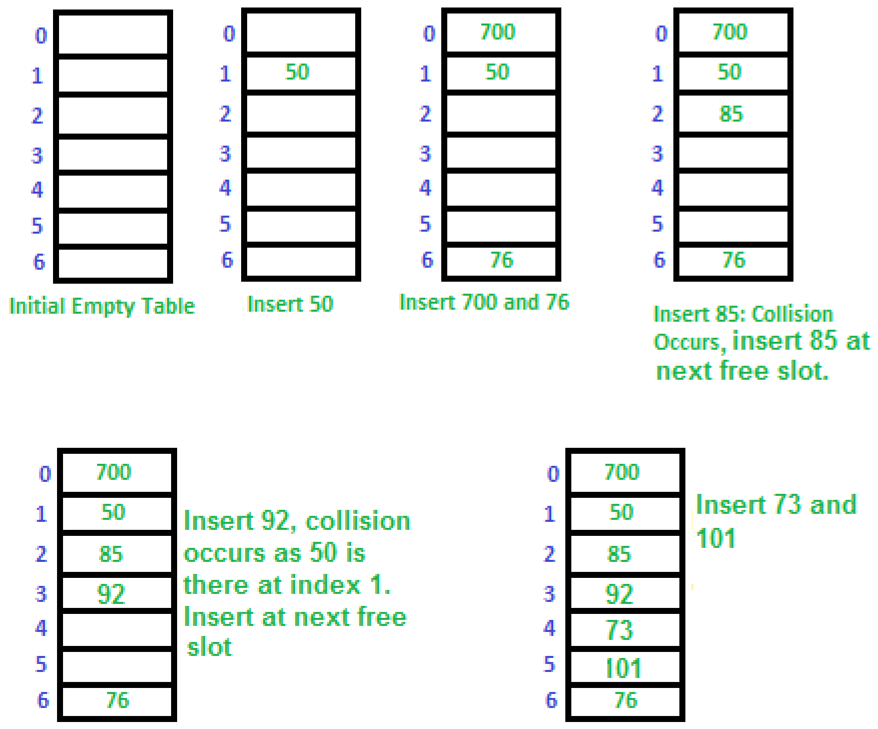
```java
        for (int c = 0; c < inputElements.length; c++) {

            int hash =  inputElements[c].hashCode();
            int hashValue = hash;

            hashValue = hashValue % arraySize;

            if (hashValue > arraySize){

                hashValue = hashValue % arraySize;
            }

            if (hashValue < 0){

                hashValue = Math.abs(hashValue);
            }
```

Then it'll check if there's an existing element in that particular array index, if so, it'll find another index to insert that key according to the collision resolution.

After that, if no collisions are found the program will assign the input element key to that hash value index of the hash table.

3. Collision Resolution 1, Linear Probing

Linear probing is the first form of open addressing collision resolution we employed and it resolves a hashing collision by searching sequentially for the next vacant cell. So, the step size is 1 where if one cell is occupied the program will check the next cell if it's empty and so on, till it finds an empty index that it can assign the input element or key.

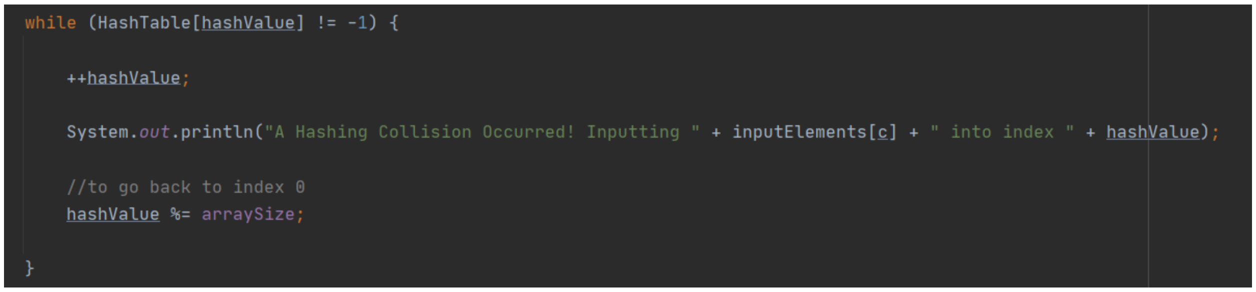

Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, insert 85 at next free slot.
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
Insert 73 and 101

Example of linear probing as a collision resolution

The linear probing algorithm used for both the integer and string hashing are similar.

Using a while loop, when the program is attempting to assign a key to a particular hashing table index it detects whether the cell is empty and not occupied by another key. An empty cell is represented by -1, so the program uses this as the checking condition for the while loop.

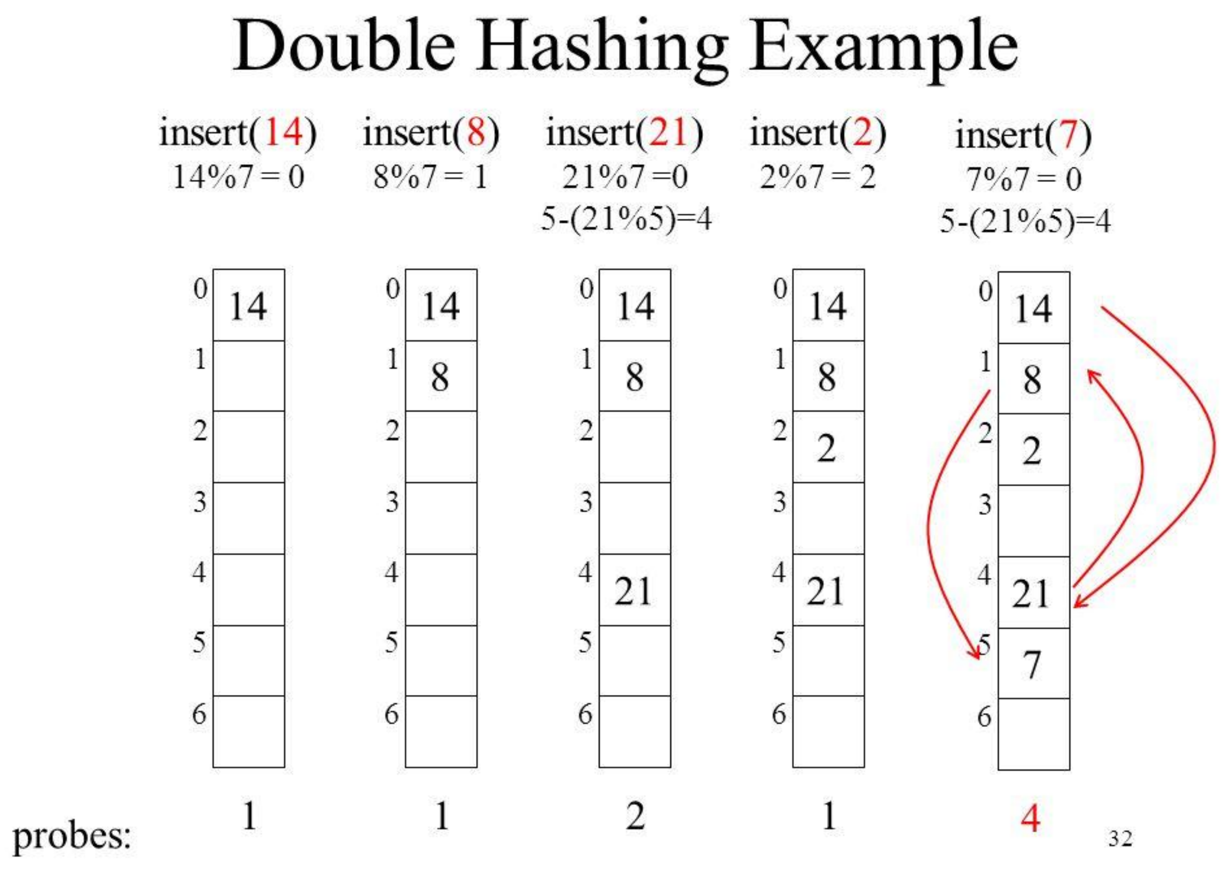
If a particular cell has an existing key (!= -1), then the next course of action is to increment the hash value index by 1 and try to insert the key at the next index. The system will also prompt the user that a hashing collision has occurred, indicating that it's attempting to insert the particular element to the next index. The system will show the user which particular key it is, and which index it's trying to insert at. Through this, the system can demonstrate the hashing action as well as the collision resolution action more clearly.


```
while (HashTable[hashValue] != -1) {

    ++hashValue;

    System.out.println("A Hashing Collision Occurred! Inputting " + inputElements[c] + " into index " + hashValue);

    //to go back to index 0
    hashValue %= arraySize;

}
```

This process is iterative until it finds an empty cell in the hash table to insert the key.

4. Collision Resolution 2, Double Hashing

Similar to linear probing, double hashing is another collision resolution technique we have demonstrated in our implementation of hashing. Double hashing revolves around the idea of generating a new hash value or index for the object we are trying to store within the hashtable. The introduction of a second hashing function allows us to overcome any collision occurrences during insertions of values into a hashtable. It is essential however to ensure that both hash functions are completely different such that after the second hashing function is used no collision occurs. In our implementation of double hashing, the first hashing function is a modulus hashing function in the format of (key % array size) if this hash function generates a hash value that causes a collision then our algorithm performs a while with the condition if the index generated by the hash value is occupied generate a new hash value using our second hash function. Our second function uses the prime number 7 in order to calculate a number of steps that should be added to the hash value to help overcome the collision and this calculation is done through the formula of 7 - (hash value of the first function modulus of 7). Number of steps are then added to the hash value generated from the previous function to generate a new hash value. Once the new hash value is generated it goes through the condition check once again if that hash value is occupied this process is repeated, if it is not occupied then the object is inserted at that index. This approach to double hashing is taken for both string and integer value where the only difference is the prime numbers being used to calculate the number of steps to be added, namely 7 for integer values and 5 for string values.



Example of Double Hashing

5. Retrieving Keys

To further demonstrate the efficiency of hashing operations, we have implemented a function that retrieves data entries from the hash table. In order to maintain the time complexity of O(1) when retrieving keys the retrieve function must access the key index directly rather than iterating through the array and comparing the value at each index during each iteration. To do this the search function must be able to generate the same hash value for the key as the hash function. Such that it can retrieve the value at that specific index directly. In our implementation we adopted the same modulus function in our search key function as in our hash key function. Furthermore, we have also take into account the impact the collision resolution technique implemented might have on the hash value and therefore implemented a while loop with the base condition checking if the index contains the value of the key if it not it increases the index value by 1 each time and performs a check, this is technique is take into account the linear probing collision resolution technique and its impact on the hashkey generation.

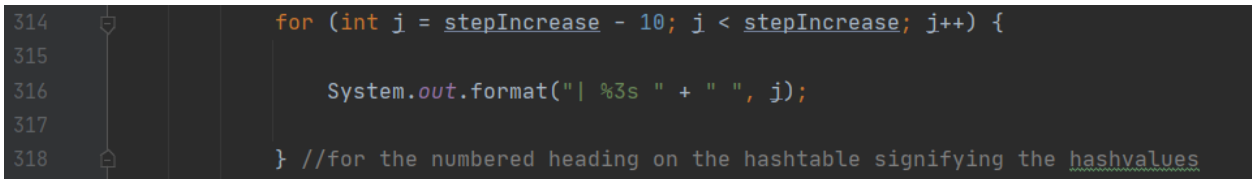
6. Displaying the Hash Table

In order to further demonstrate the hashing process and the final layout or diagram of the hashing output we can display the hashtable and its contents according to the corresponding indexes. Thus, further enhancing the program as an interactive and representative visualisation or simulation tool of hashing integer and string keys as well as the collision resolution mechanisms, linear probing and double hashing.

For this purpose, we created 2 functions to display the hashtable, one for the hash tables that store integer and string keys respectively.

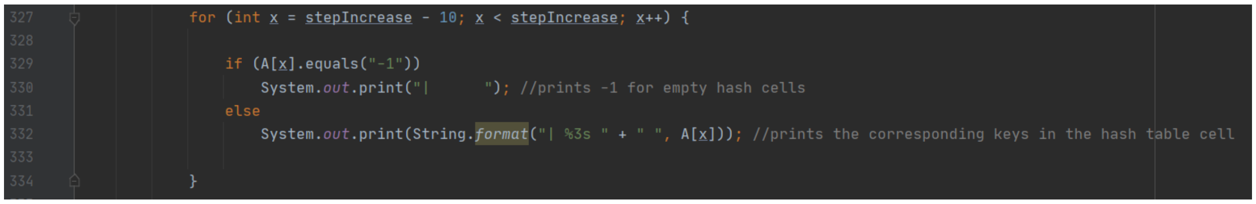
The functionality of both are similar, where they both display hash tables with 10 columns per row. So, if there are 40 indexes or values to hash there are 4 rows.

For this, there's the main for loop which iterates for 4 times or the number of index values/array size divided by 10. Then, the program prints a numbered header of the hash tables which indicate the hashtable indexes from 0-9 for the first row and 10-19 for the following and so on.



```java
for (int j = stepIncrease - 10; j < stepIncrease; j++) {

    System.out.format("| %3s " + " ", j);

} //for the numbered heading on the hashtable signifying the hashvalues
```

It iterates through the hash table and if it detects a stored key, it'll print its value at the corresponding hash index. If not, it'll maintain the original -1 value (for integer hash tables) or blank value (for string hash tables.)



```java
for (int x = stepIncrease - 10; x < stepIncrease; x++) {

    if (A[x].equals("-1"))
        System.out.print("|      "); //prints -1 for empty hash cells
    else
        System.out.print(String.format("| %3s " + " ", A[x])); //prints the corresponding keys in the hash table cell

}
```

Limitations

Limitation 1: The program uses relatively-small fixed size arrays to represent hashtables
For our implementation, at its basic application, we set its array size to a relatively small, 40. Thus, by having a fixed size, the limitation is that it is difficult to expand the size of these arrays

to accommodate for more values and can only be achieved through hardcoding. This is because arrays are a static data structure in that their size is fixed at instantiation and cannot be expanded or contracted. Plus, as more values are inserted and the array is filled up, the efficiency and performance of the hash table as the rate and speed of insertion decreases because the table has become too full, resulting in more collisions during data insertion. A way we can overcome this is to instantiate the hashtable at a larger array size to accommodate for more key insertions. Additionally, since we're using Java for the Hashing implementation, we can use java ArrayList from the java.util package, which is a dynamic array that expands as more values/keys are inserted and contracts as values/keys are removed.

Limitation 2: The linear probing algorithm experiences primary clustering
Primary clustering is a typical problem with the linear probing algorithm where the keys that hash to a particular index of the hashtable will follow the same sequence in finding the next vacant cell in the hashtable. Thus, this sequence is in increments of 1 from the previous hash index. Even if we were to introduce quadratic probing where the sequence of steps is the square of the step number this results in another limitation known as the secondary clustering. Here, again, entries colliding with a certain filled hash index will follow the same sequence of steps from the initial hash index. Thus, as there are many and close to unlimited collisions as the hash table nears full capacity, this drastically reduces the efficiency of the linear probing hashing function we've developed.

One way we've overcome this is by introducing a second collision resolution mechanism known as Double Hashing. Here, there's a second hash function that hashes the entry or key a second time using a different function. Thus, the result is a different step size. Therefore, when a collision occurs, the chances there's another subsequent collision is lower.

Limitation 3: Double hashing experiences many collisions and becomes inefficient when hash table is nearing full capacity
As more inputs or keys are inputted into the hashtable it grows closer to full capacity. Thus, the probability of collisions increases as well. Therefore, a malicious user with knowledge of the specific hash function can take advantage of this to input the worst-case scenario entries or keys, exploiting the vulnerabilities of the system and resulting in excessive collisions. As a result, the performance of the hash function will be affected as it'll need to handle all the additional collision resolutions.

Hence, this renders this hashing algorithm less suited to be applied in real-world applications, like critical applications and login systems, as better hashing algorithms and data structures with better worst-case guarantees are more preferable. For instance, in that case, we can utilise a universal hashing algorithm or a more widely-accepted and sophisticated hashing algorithm like MD5 and SHA. These algorithms have far less vulnerabilities and are prone to lesser exploits.

Problem 2: Dijkstra Algorithm

Introduction to Dijkstra Algorithm

Overview of Dijkstra Algorithm

Dijsktra's algorithm is primarily used to find the shortest or lowest weighted path from a given source vertex to all other vertices in the graph [3]. Thus, it's considered a minimum spanning tree (MST) and functions similarly to its counterpart the Prim's algorithm. So, given a set of vertices and its corresponding length to certain nodes, we use this to find the lowest total weight between two particular nodes [3].

From looking at it's operation, we can see that the Dijkstra algorithm has many applications in real-world scenarios. This mainly includes for road networks and routing protocols, where the nodes signify objects that transport or transfer information and goods whereas the edges that connect these nodes, display routes it can follow.

Essentially how a Dijkstra algorithm works is to maintain 2 sets of numbers, where one contains vertices in the shortest path tree, whereas another has vertices yet to be included in the shortest path tree [3]. So, in every stage of the algorithm's operation, we aim to find the vertex in the set that's yet to be included, and calculate the minimum distance to it from the source.

For our implementation of the Dijkstra algorithm we use the same principle to find the distance from a source node or start location to the end node or destination. And a requirement of it that we've followed is that the weight or distance between 2 vertices is a non-negative value.

Data Structures, Tools, and Programming Language Features Used

The data structures implemented in the code are queue and breadth first search. The programming features used in the code are python dictionary data type, python list data type, python list manipulation and python sort function.

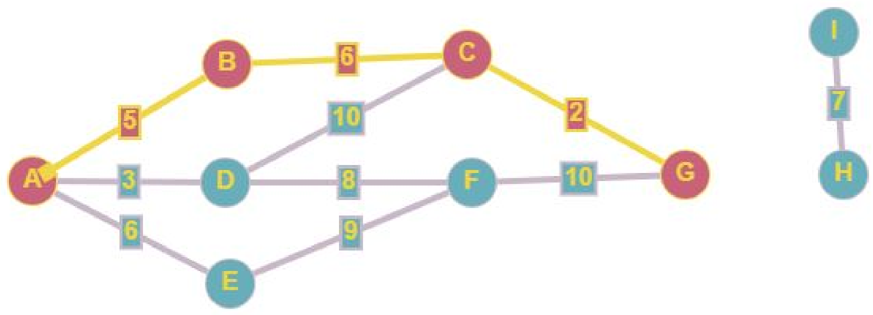
Implementation

The Dijkstra algorithm takes two nodes one is the initial node and the other is the destination node. The algorithm then computes the shortest distance between the two nodes. There are 6 major steps in the algorithm

Step 1: Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set. In our algorithm we use the python list data type to create this universal set.

Step 2: Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current. For our algorithm we use 100 instead of infinity and use tuples to store the node and tentative distance value.

Step 3: For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has length 2, then the distance to B through A will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, the current value will be kept. In our algorithm we make use of two helper functions that give us the nearest neighbouring node with its distance.



Step 4: When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again. We use if statements in our code to check for this and use the python list remove function to remove the visited node.

Step 5: If the destination node has been marked visited then stop. The algorithm has computed the distance successfully. We use another check statement and if it's true then the loop breaks.

Step 6: Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3. However, if the loop does not break, we continue our algorithm with the nearest neighbouring node.

After the algorithm has computed the shortest distance from the initial node to all the nodes. We get a list as an output of our Dijkstra function. The list contains the minimum distance it takes to reach each node from the initial node. We then use this list to compute our shortest path and distance to the destination node. The getshortestpath function in our code computes the distance using a backtracking list. It backtracks from the destination node and follows the path to the initial node. Afterwards we use the backtracking list to display the correct shortest path from initial node to destination node.

Limitations

The Dijkstra algorithm that we have produced is designed in a way that we have to input the graph of the map, as a dictionary data type, by ourselves into the code. The limitation of this approach is that the code will not work unless it is provided with a graph dictionary prior to executing the code. Moreover, the input dictionary needs to follow a certain structure. For example, the keys of the dictionary are the names of the locations and the value is a list of neighbouring locations. The value list contains all the neighbours of the key as a tuple where the first element of the tuple is the name of the neighbouring location and second element of the tuple is the cost of going to this location. The above information can be written like this as an example:

{“location A”: [(“neighbour B”, 5) , (“neighbour C”, 10)],
“location B”: [(“neighbour C”, 7)]}.

Furthermore, the code only works for all the locations listed in the input dictionary only. If the user enters a location that is not found in the input dictionary the code will not work and it will give an error to the user.

Individual Contribution

ELTIGANI HAMADELNIEL 17087610

In the implementation and demonstration of Hashing for our first problem, I contributed to the development of the implementation of double hashing as collision resolution for both the integer and string double hashing insertion functions. In addition, I cooperated with Saurabh in the implementation of our search key demonstration function through which we retrieved data entries. Moreover, I have also cooperated with Saurabh in the brainstorming and implementation of the hashing technique for string values and how linear probing can be introduced to help overcome collisions. Furthermore, I contributed in writing and describing an overview of hashing as a whole within the report. Overall completing this project has helped me gain a fundamental understanding of hashing its key concepts and applications across different domains in the field. Furthermore, completing this project has allowed me and my team mates to come together for a brainstorming session to come up with the best ways to help achieve our goal which is to visualize and demonstrate hashing and its concepts in an easy to digest manner.

SAURABH VARUGHESE M. KOVOOR 20017604

Through this project, I gained a more practical understanding of hashing and inserting values into a hash table, whereas how and why Dijkstra's algorithm is implemented. Essentially, for this project, I primarily worked together with my teammates to perform extensive preliminary research into hashing tables and ways we can implement and demonstrate it through a simple representation using java. From there, I was tasked to work together with my teammates to develop the code to instantiate the hash tables and develop the algorithm for inserting values, and resolve the resulting hashing collisions using linear probing. Along with that, I also developed the showHashTable() function to display the outputs of the hashing function and the hashtable in a clear, visible and easy to understand manner. Overall throughout the project our goal was to make a simple demonstration or visualisation tool for hashing and the linear probing and double hashing collision resolutions. Also, I helped perform research and writing to describe the motive for the solution to the second problem, which is Dijsktra's algorithm. All in all, this has been a great-learning experience and a stepping stone about a highly-used data structure in various real-world applications, which is hashing.

MUHAMMAD MAHAD NIAZ 1905313

Firstly, I worked with my teammates in the preliminary research to understand Dijkstra's algorithm. After we had done our research and understood the flow of the algorithm, I primarily worked on developing the code to find the shortest path between two locations using Dijkstra's algorithm. Writing the code made me understand the inner logics of the algorithm and increased my knowledge on the topic. Moreover, during the coding process I had to understand and use different data structures to help me create the algorithm. Doing so helped me understand the need of data structures and how using data structures can help produce better code. Secondly, I worked together with my teammates to write the report for the project. Writing the report helped me gain fundamental knowledge on the two topics. Working together with my teammates helped with my communication and practical skills. The cooperation from my teammates and I helped us gain knowledge outside the typical study environment and helped us visualize two of the most important core topics in data structures and algorithms. The assignment has been a great learning experience for me as it helped me gain knowledge outside the books and study material and helped me gain practical skills which will be applied by me in the real world.

Challenges

Hashing

One of the challenges we faced while implementing our algorithm, was finding a way to convert string inputs into variable sized integers that can be used to generate hash values. In order overcome this challenge, we research multiple libraries with functions that can help convert string values into their unicode or integer values, in this process we found the .hashCode() function that is a feature of the java.lang.string library that helps convert string objects or variables into a 128 bits integer values.

Another challenge that we faced during implementation was a bug in the code within the search function for string values, where the code would not print out the output or cause index out of bounds exceptions. We found that the error was caused because the hash value produced during insertion process was impacted by collision and resolved through our collision resolution technique but that was not the case for the hash function in the search function. To solve this we applied the same conditional while loops in our insertion algorithms to the searching algorithms in order to ensure that the same hash value was produced and therefore data retrieval was accurate and bug free.

Finally, the last challenge we faced during implementation was a bug in the code within the double hashing functions where if a collision occurred the value would be placed into two indexes and stored twice in the hashtable causing a duplication. We found that this problem was caused by the second hashing function that would be executed if a collision would occur and would set the key into two indexes rather than one. To solve this we change the second hashing function to produce a number of steps to be added to the original hash value rather than creating a completely new hash value.

Dijkstra Algorithm

One of the challenges we faced while coding the algorithm was that the code was giving an error if we wanted to go from location A to H but there was no path to link the two locations together. So, to solve the issue we set up a variable named cost which would have the total cost of the trip from start location to end location. Since we know that in our algorithm if there is no path between two locations then their cost would be 100 so after that we wrote an if statement which would check for the cost. If the cost is 100 then the code would display that there is no path between these two locations else the code would display the shortest path computed by the algorithm.

Conclusion

Overall, through this project we aimed to develop a user-friendly program to demonstrate the application of hashing to store integer and string values as well as the application of Dijkstra's algorithm. Along with that, both programs were designed to display, simulate and show the users how a hashing algorithm and it's collision resolution mechanisms take place in addition to the calculation of the short path from one node to the other nodes given a weighted graph. Whereas for the Dijkstra's algorithm implementation, the program was intended to showcase how the algorithm works and calculate the shortest distance from 2 given points that the user inputs according to the predefined weighted graph. In conclusion, through the implementation of our algorithms and datastructures we were able to gain a better understanding of hashing and dijkstra's algorithms key features and concepts. Furthermore, giving us a broader understanding of how these algorithms are applied across multiple domains within the tech industry.

References

[1] “Introduction to Hashing,” *Cmu.edu*. [Online]. Available: https://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture17.pdf. [Accessed: 06-Jul-2021].

[2] P. M. de Araújo, “An overview about hash functions: Theory and Security,” *Medium*, 22-Nov-2018. [Online]. Available: https://pemtajo.medium.com/an-overview-about-hash-functions-theory-and-security-21e52ddc9993. [Accessed: 06-Jul-2021].

[3] “Dijsktra’s algorithm,” *Geeksforgeeks.org*, 25-Nov-2012. [Online]. Available: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. [Accessed: 06-Jul-2021].