**Experiment No. 9**

|  |  |
| --- | --- |
| **Name : Manjiri Chavande** | **DSE SY COMPS** |
| **Division: B batch-A** | **UID: 2023301003** |

|  |
| --- |
| * **Problem Statement** |

**Implement Hashing using Linear Probing**

|  |
| --- |
| * **Theory** |

Hashing is a technique that maps data to a fixed-size array, called a hash table, using a hash function. It enables quick data retrieval based on keys, reducing search time to nearly constant. Collision resolution methods handle situations where different keys produce the same hash value, such as using separate chaining (linked lists) or **open addressing** (probing techniques). **Separate chaining** links elements with the same hash to a linked list, while open addressing finds alternative locations for colliding elements. Common open addressing methods include **linear probing**, **quadratic probing**, **and double hashing**. These methods ensure efficient storage and retrieval of data, minimizing the impact of collisions on the overall performance of the hash table.

Linear probing is a collision resolution method that handles this by sequentially searching for the next available index in the array. Step-by-step explanation of linear probing:

1. Compute the hash value of the key using a hash function.
2. If the computed index is empty, store the key-value pair there.
3. If the index is occupied, search for the next available index by incrementing the index in a linear manner.
4. Continue this process until an empty index is found or the entire table is traversed.
5. Insert the key-value pair at the first empty index found.
6. During search, follow the same linear probing process until the key is found or an empty index is encountered.
7. For deletion, mark the key-value pair as deleted without removing it from the array.
8. Ensure the probing wraps around to the beginning of the array if the end is reached.
9. Monitor the load factor to avoid table overflow.
10. Handle resizing of the table if the load factor exceeds a certain threshold.

* **How to calculate the hash key?**

Let's take hash table size as 7.

size = 7

arr[size];

Formula to calculate key is,

key = element % size

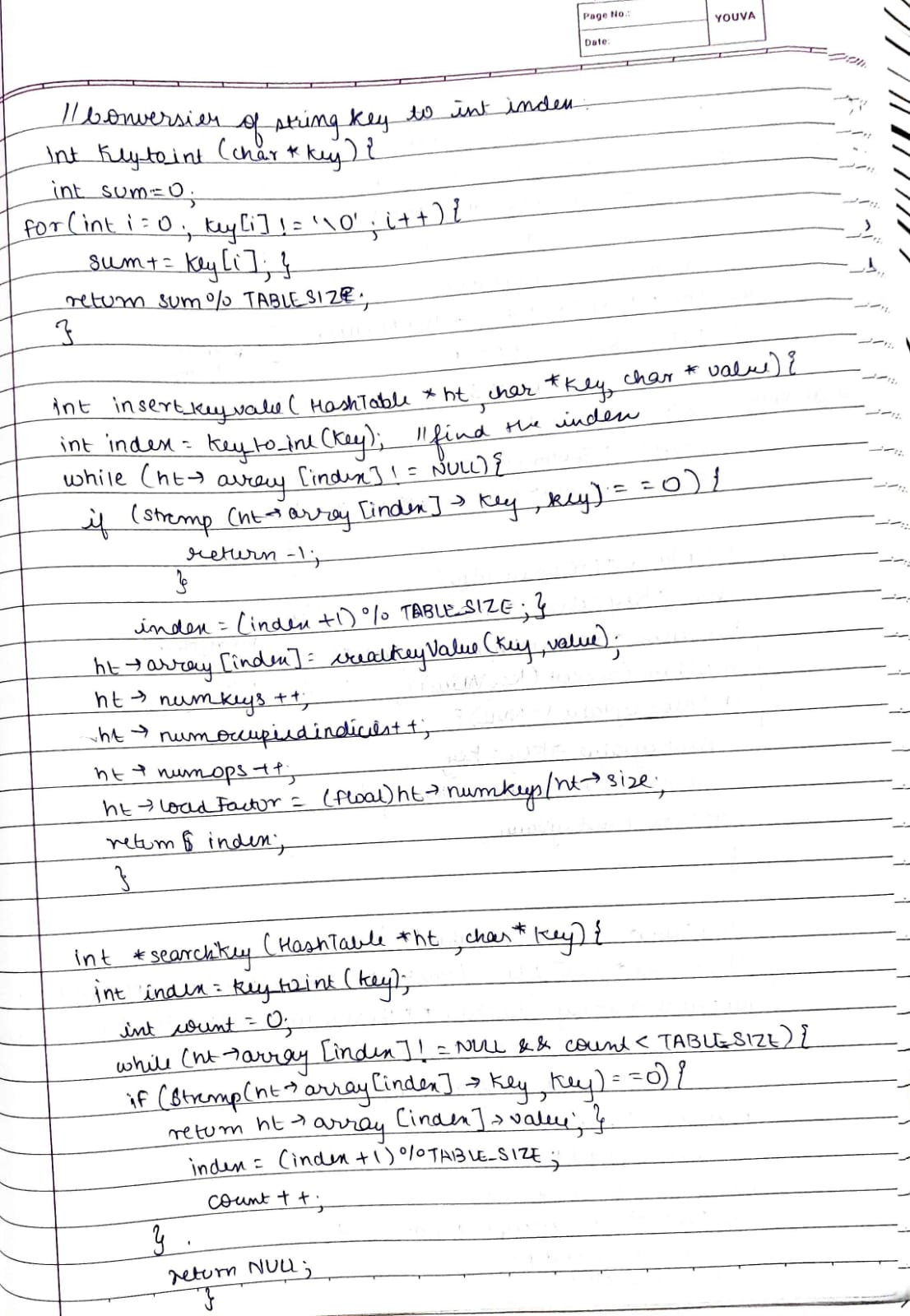
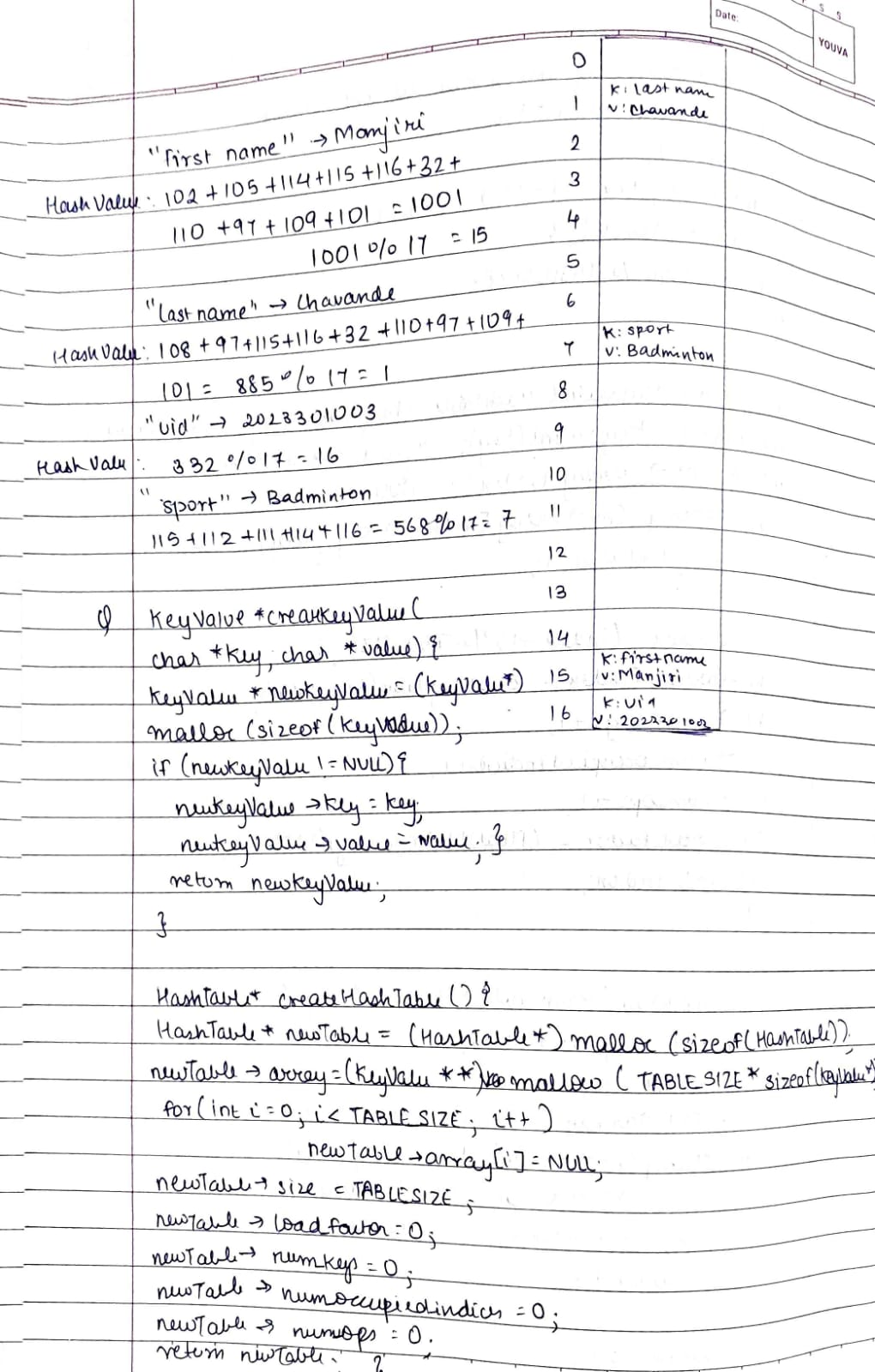
If we take modulo of number with N, the remainder will always be 0 to N - 1.

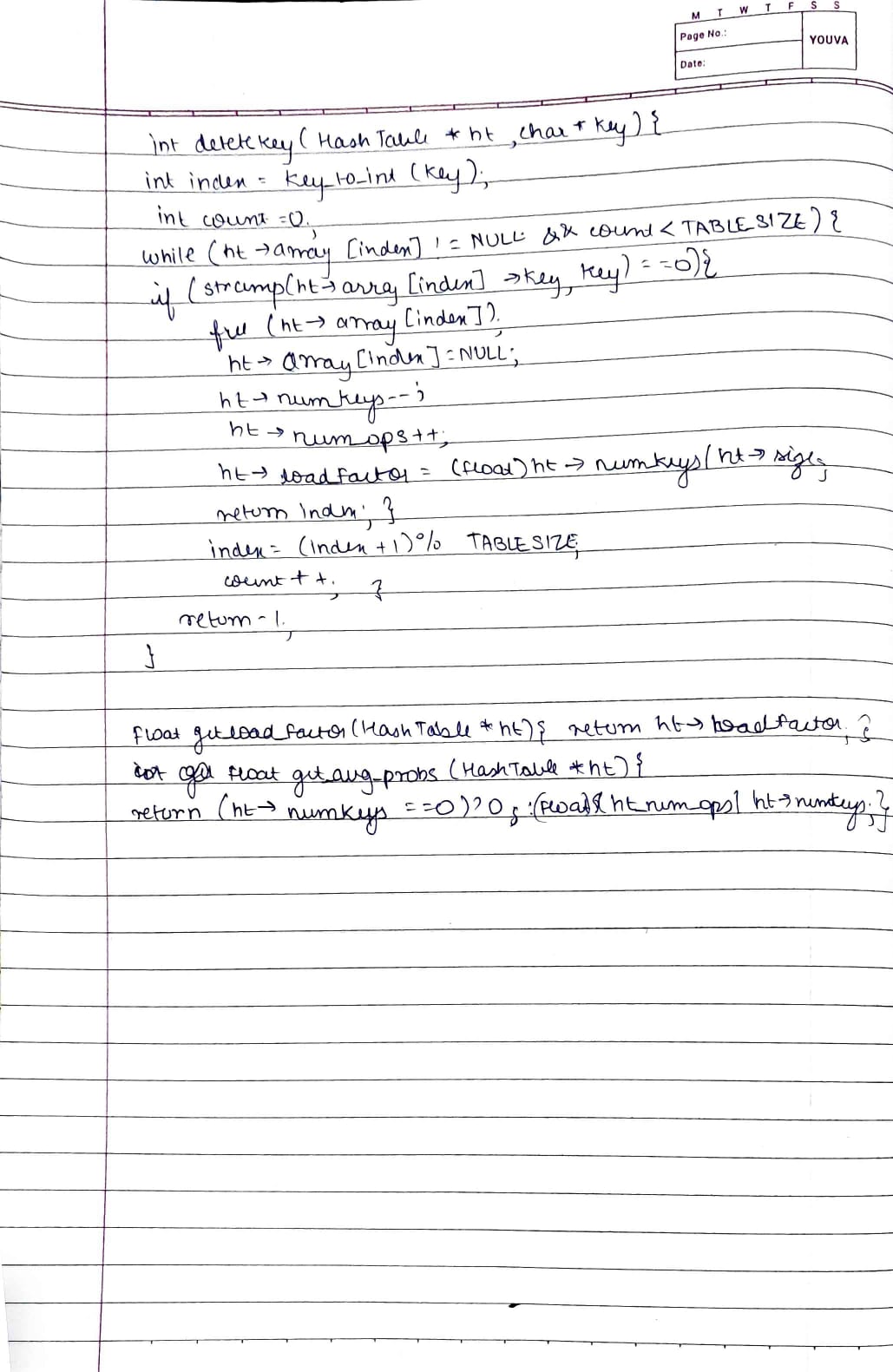
Exactly array index also starts from 0 and ends with index N -1. So we can easily store elements in array index.

|  |
| --- |
| * **Algorithm** |

1. **CreateKey values function:**
   1. **Dynamically allocate memory**
   2. **Assign newValue->key = key; and newValueKey->value =value**
   3. **Return the created newKeyValue pair**
2. **Create hashTable function:** 
   1. **Dynamically allocate memory for newHashTable and the array inside it**
   2. **Interate through array and set all the values as NULL ;**
   3. **Set all the properties to 0;**
   4. **Return the created hash table.**
3. **convert a string key to an integer index**
   1. **iterate through the key until ‘\0’ end of string is met**
   2. **Sum up the ASCII values of characters in the key**
   3. **Return the Modulo operation to ensure the index is within the table size**
4. **Insert a value**
   1. **Goto step 3 get the index**
   2. **Iterate thorught the array till it is not null**
   3. **Compare the current key and the key previously present at that particular index ,if it is true that means key alraey exists return -1;**
   4. **find the next available slot**
   5. **hkey = key% TABLE\_SIZE Increment all the other properties , calculate the load factor**
   6. **Return the index.**
5. **Seraching**
   1. **Hashtable is an array of size = TABLE\_SIZEStep**
   2. **1: Read the value to be searched, keyStep**
   3. **2: let i = 0**
   4. **(key+1)% TABLE\_SIZE get next index**
   5. **compute the index at which the key can be found index = (hkey+ i) % TABLE\_SIZE**
   6. **if the element at that index is same as the search value then print element found and STOP**
   7. **else step 4: i = i+1**
   8. **if i < TABLE\_SIZE then go to step 4**
6. **Delete**
   1. **Goto step 3 get the index , count = 0**
   2. **While the array is not null and count <17**
   3. **Check of the key exists if true free the index and decrement the num\_keys and increment the operation by 1;**
   4. **Return the index**
   5. **Index = (key+1)% TABLE\_SIZE**
   6. **Count++**

|  |
| --- |
| * **Solution** |





|  |
| --- |
| * **Output** |

|  |  |
| --- | --- |
| **When all the values are inserted , and deletion of holiday key:**  **Size of the array is 17** |  |
| **When the value searching does not exists** |  |
|  |  |

|  |
| --- |
| * **Test Case** |

|  |  |
| --- | --- |
| Testing for different values  Array size :23 |  |

|  |
| --- |
| * **Conclusion** |

Thus we have successfully implemented linear probing.