Aral Müftüoğlu
22201566
Section - 3

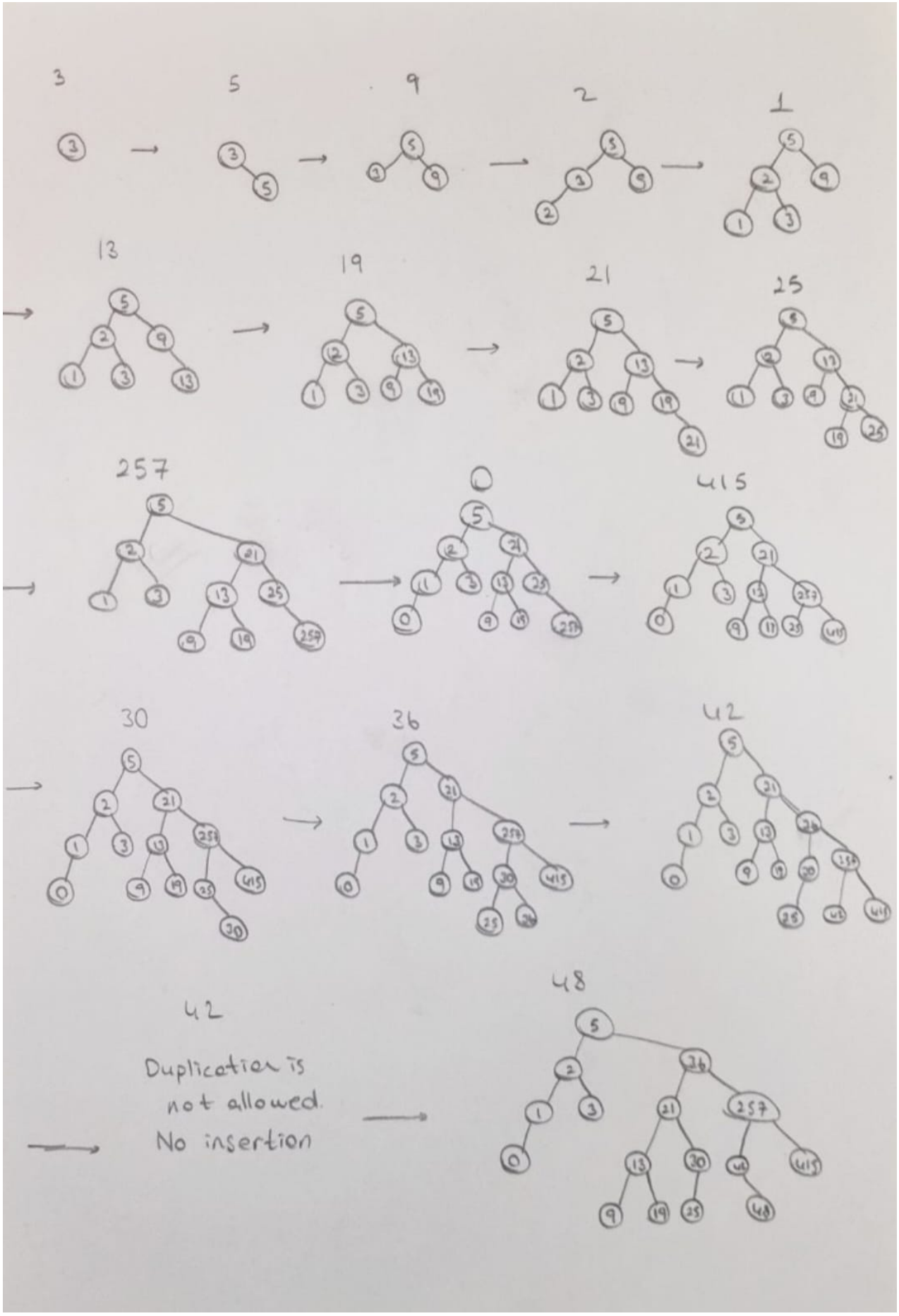
Q1)

Part A)

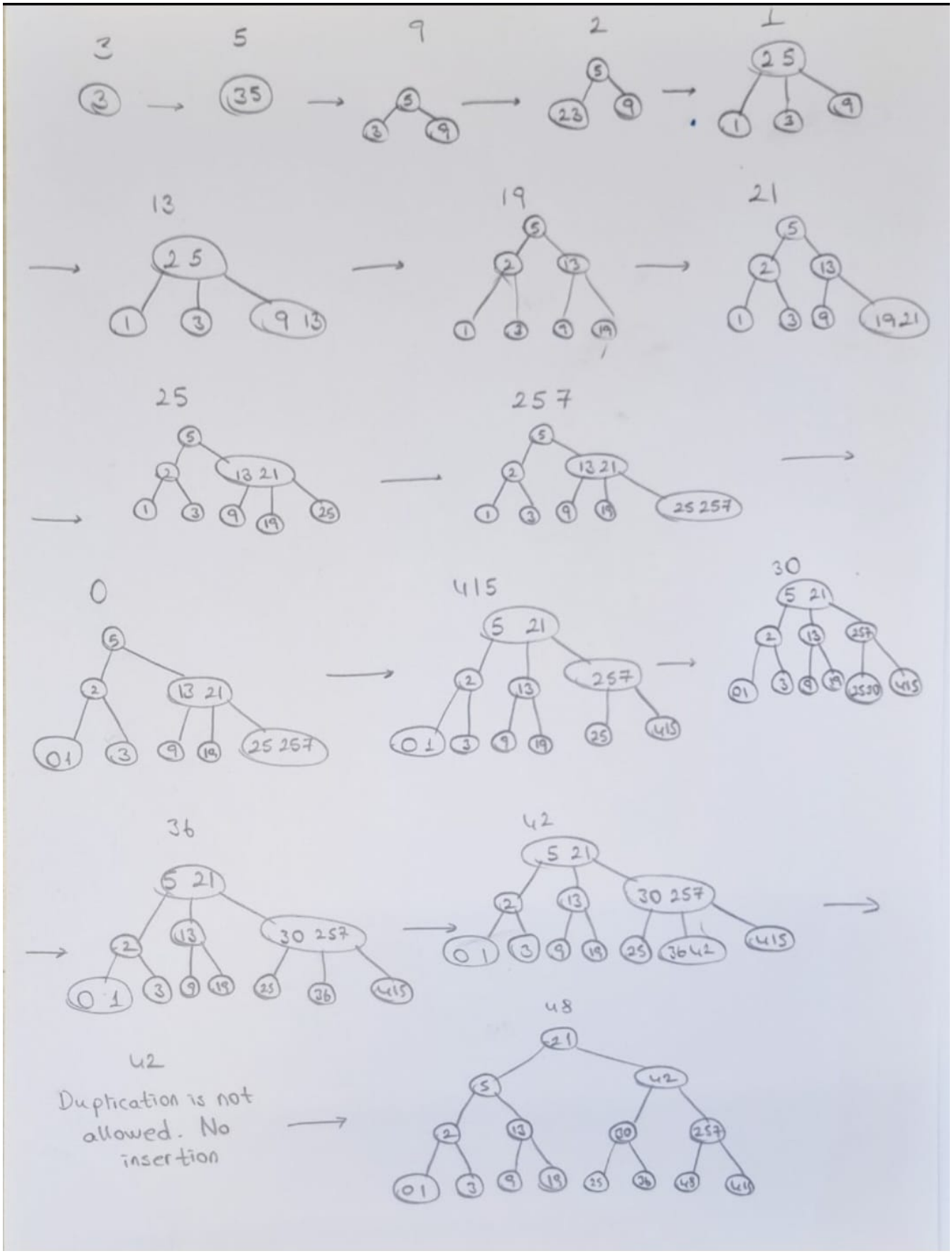
In order to store coordinates consisting of integers in a good way, it is necessary to use self-balancing trees. For this, AVL tree, 2-3 tree or red-black tree can be used. These trees continue to be balanced automatically after each insertion or deletion. If self-balancing tree is not used, the tree can behave like a linked list or array list in some cases, which can lead to slow and troublesome situations. Therefore, using such trees will help to store coordinates both safely and efficiently.

Part B)

AVL trees are particularly suitable for this job because they have a strong balancing mechanism. For each element, it has a feature called balance factor, which gives the height difference between the left and right subtrees. After each insertion and deletion, the balance factor is expected to be equal to -1, 0 or 1. In cases where it is not equal, various rotations are made to preserve the balanced structure of the tree and to ensure that the height of the tree is kept at a minimum. Thus, fast insertion and deletion operations can be achieved. Here, cyberspace defense is a must to maintain the balanced structure and the features of AVL trees distinguish it from other self-balancing trees. Therefore, AVL trees are suitable for this job.


3
5
9
2
1
13
19
21
25
257
0
415
30
36
42
42
Duplication is
not allowed.
No insertion
48

Q2)


3
5
9
2
1
13
19
21
25
257
0
415
30
36
42
48
42
Duplication is not allowed. No insertion

Q3)

Subtask-1

For example assume we insert "NeuralLink", "PowerCore" and "FlexCable", when we insert them to find the index we will sum their chars ASCII values.

NeuralLink → N e u r a l L i n k
78 + 101 + 117 + 114 + 97 + 108 + 76 + 105 + 110 + 107 = 920

with the function that used for hashing index = 920 mod 211
= 76

PowerCore → P o w e r C o r e
80 + 111 + 119 + 101 + 114 + 67 + 111 + 114 + 101 = 920
index = 920 mod 211
= 76

FlexCable → F l e x C a b l e
70 + 108 + 101 + 120 + 67 + 97 + 98 + 108 + 101 = 920
index = 920 mod 211
= 76

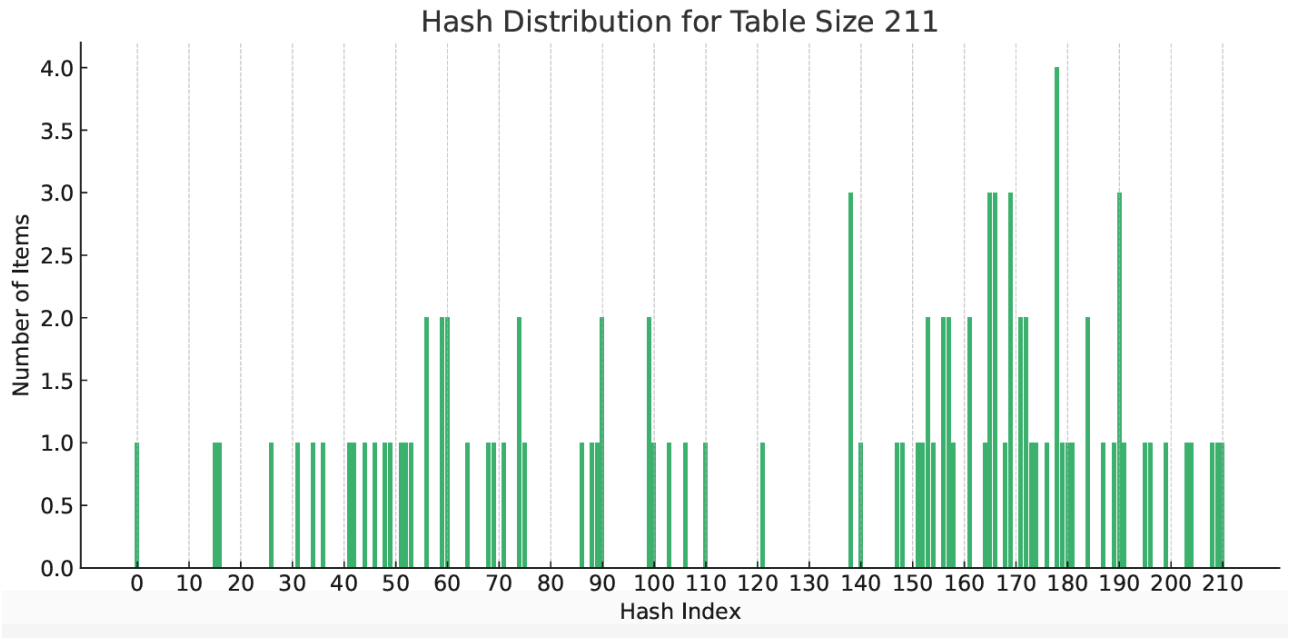
Since; three of them have the same result for the hash function, initiall they are wanted to be stored at same slot of hashtable. In short, even they are different they have the same hash value. Therefore, this situation creates a false positive condition which table gives false positive.

Subtask-2

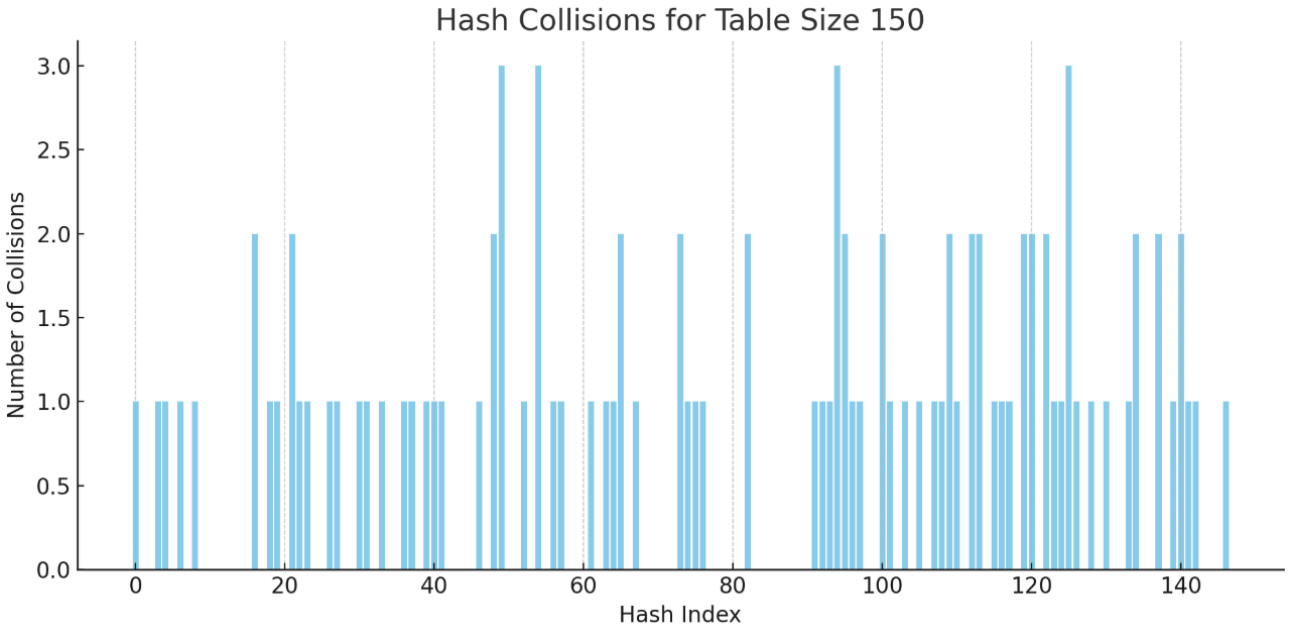
In my HashTableImproved version, I used a technique called separate chaining to fix collisions. A collision occurs when two or more items want to be added to the same index in the hash table. Instead of looking for an empty spot somewhere else in the table, I add a linked-list attached to each slot. This way, each cell has the ability to store multiple items in a chain. When I add a new item, I first add the ASCII values of its characters, and then calculate its position using a hash function that finds the remainder when divided by 211 (the prime number I chose). If the cell at the calculated index is empty, the item is added immediately. If not, I check the linked list at that index to see if the string I want to add already exists, to prevent duplication. If it is not in the linked list, I add the new item to the beginning of the linked list. This allows multiple items that are expected to be at the same index in the hash table to be added to the hash table without colliding. I have included a screenshot of the insert function to show how this chaining method works.

```
void HashTableImproved::insert(const string& item) {

    //calculating the sum of ascii values for every char
    int sumOfASCIIvalues = 0;
    for (int i = 0; i < item.length(); i++) {
        sumOfASCIIvalues = sumOfASCIIvalues + item[i];
    }

    //finding the initial index
    int initialIndex = sumOfASCIIvalues % size;

    Node* current = table[initialIndex];

    //check if the item is already in the chain to avoid duplication
    while (current != nullptr) {
        if ((*current).data == item) {
            cout << item << " is already registered as a weak spot." << endl;
            return;
        }
        current = (*current).next;
    }

    //if everything is okey, insert at the beginning of the chain
    Node* newNode = new Node;
    (*newNode).data = item;
    (*newNode).next = table[initialIndex];
    table[initialIndex] = newNode;

    cout << item << " is registered as a weak spot." << endl;
}
```

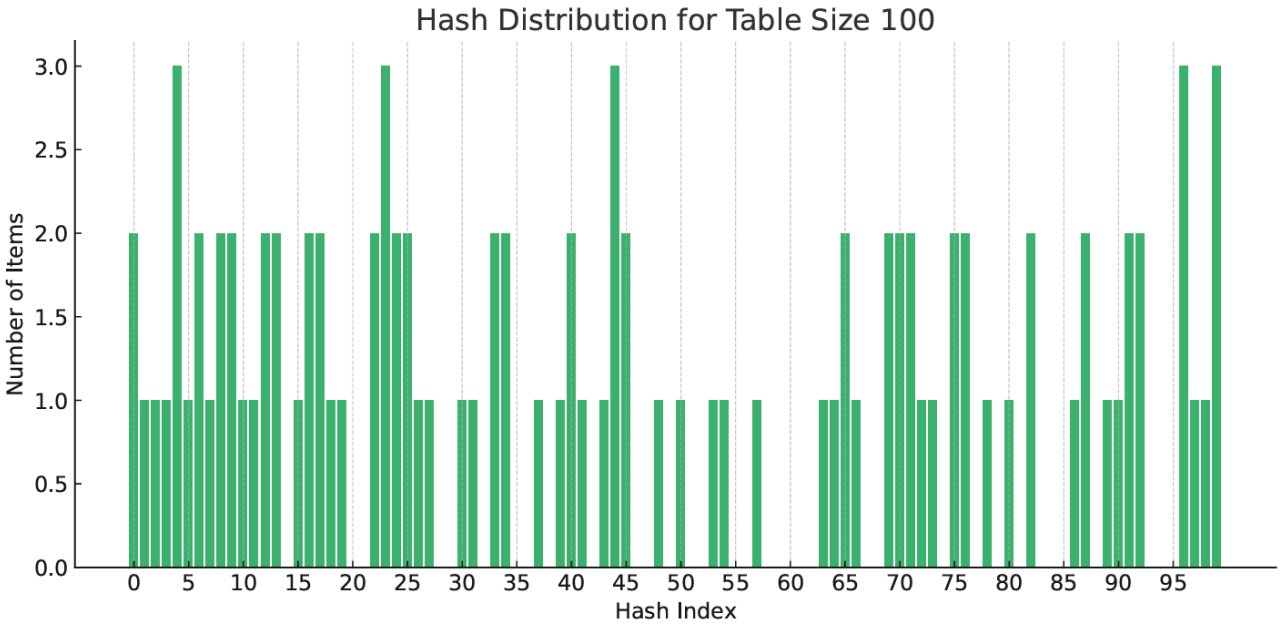
For table size 211



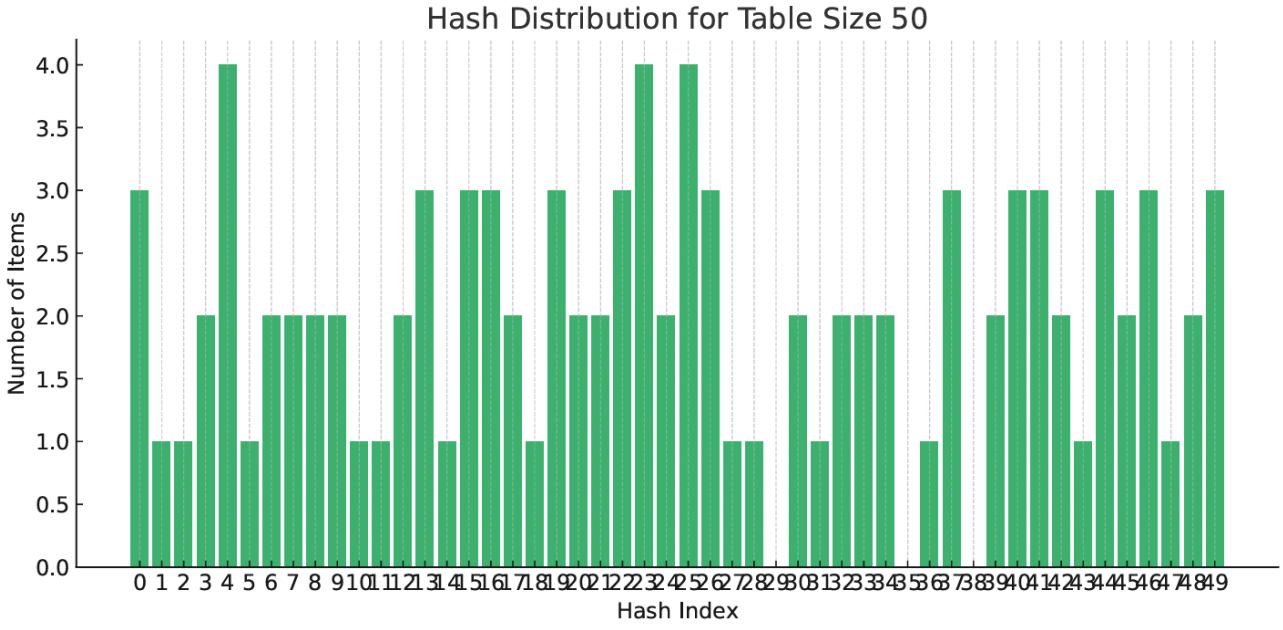
For table size 150



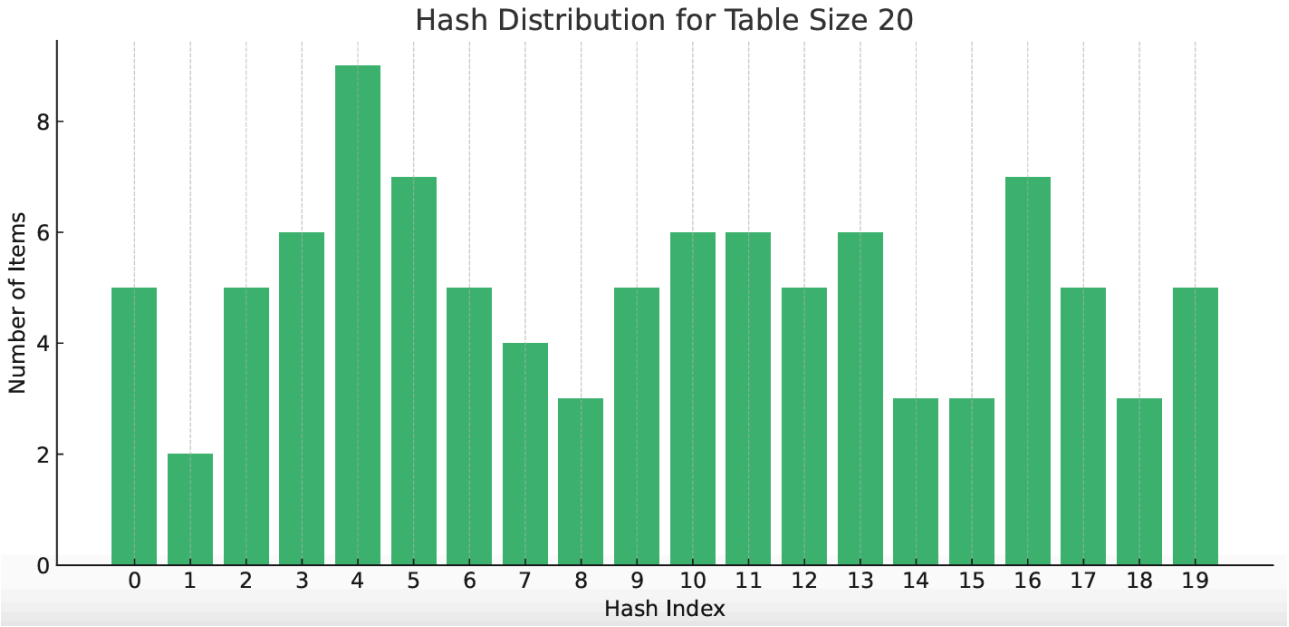
For table size 100



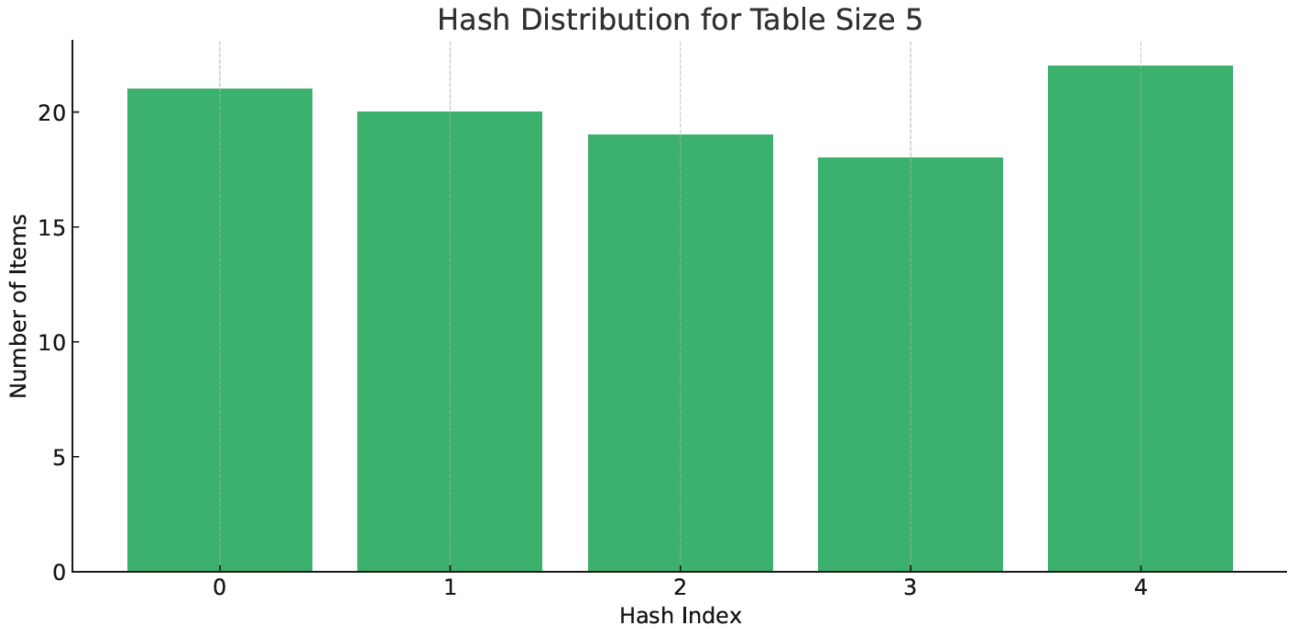
For table size 50



For table size 20



For table size 5



As the table size increases, the remainders change and as a result, the variety that can be used in the hash function also increases, thus increasing the variety. As can be seen in the graphs, as the number of elements per index increases, the length of the linked lists increases, which together with longer linked lists, creates less efficient, long-running functions (insert, remove, search). In addition, when a prime number such as 211 is used, the mod results that can be obtained increase, so the number of collisions decreases and the length of the linked list shortens. As a result, efficiency increases and running times shorten. But it should not be forgotten that while the effectiveness increases as the table size increases, the number of empty slots increases, which decreases efficiency. Therefore, in order to get the best efficiency and performance, a choice should be made between the two and a table size with a prime number should be selected.