CS201
Data Structures and Algorithms
Revision Session 6

hashing

Hash Table

implementation:
definition
hash functions

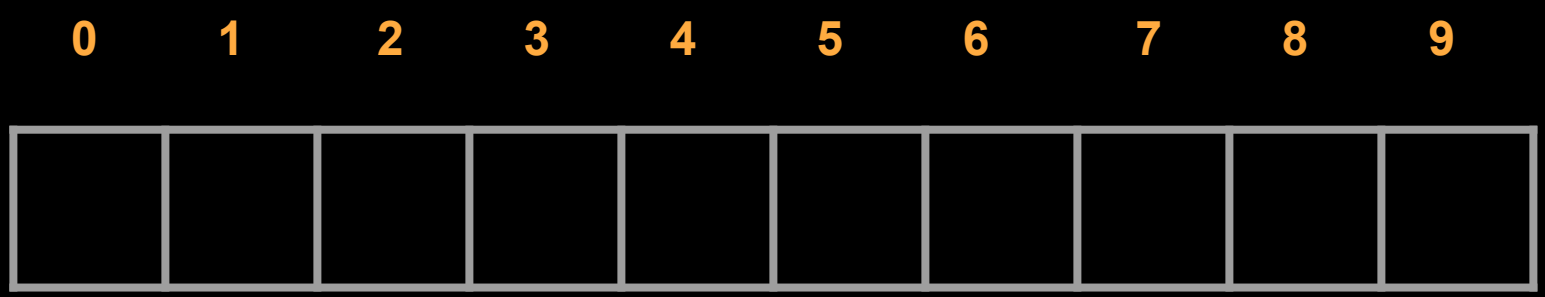
separate chaining:
search
insertion
deletion

open addressing:
linear probing
quadratic probing
double hashing

rehashing

hash tables

N = 10



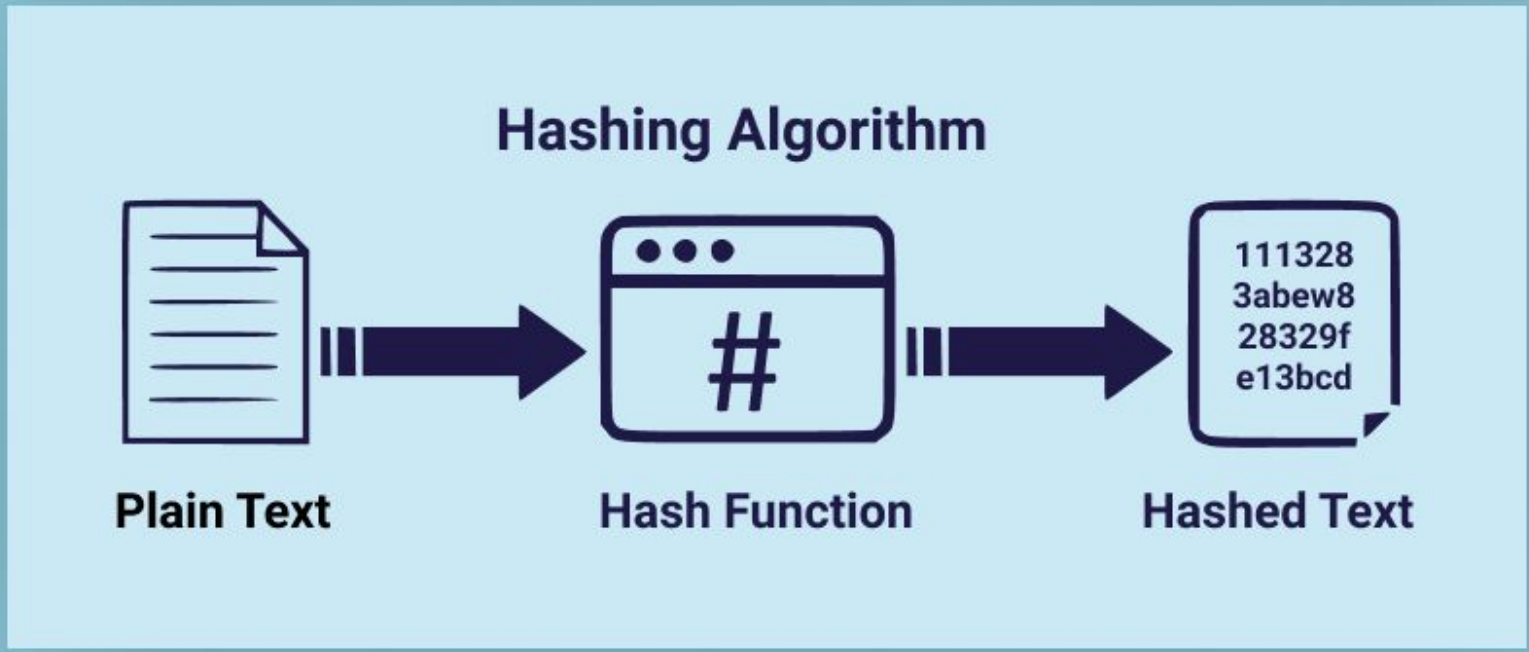
key-value pairs

the keys in the array implementation are the indexes

similar to a dictionary

the intrinsic quality of the data determines its placement

hash functions


Hashing Algorithm
#
111328
3abew8
28329f
e13bcd
Plain Text
Hash Function
Hashed Text

- deterministic: same input, same hash value
- even distribution across available slots
- quick computation
- rare or no production of the same hash value

```java
public int hashFunction(String stringValue){
    int i;
    int position = 0;

    for (i = 0; i < stringValue.length(); i++){
        position = 36 * position + stringValue.charAt(i);
    }
    position = position % N;
    return position;
}
```

Hash function for strings that takes the ASCII code of each character, modifies it and adds them up.

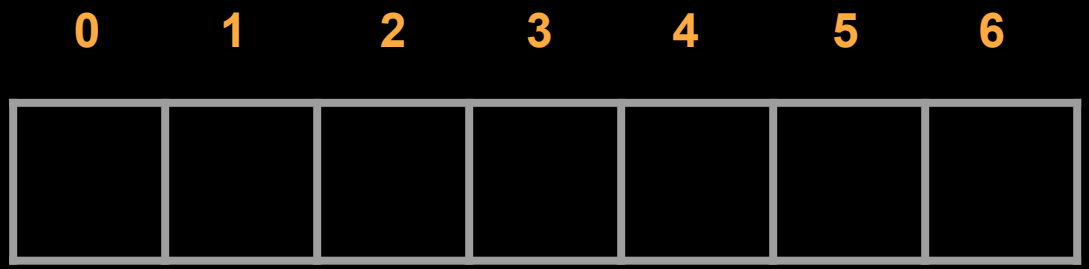
Takes the modulus according to N and returns hash value.

```java
public int hashFunction(int value){
    return value % N;
}
```

Hash function for integers, using the modulus operation with N.

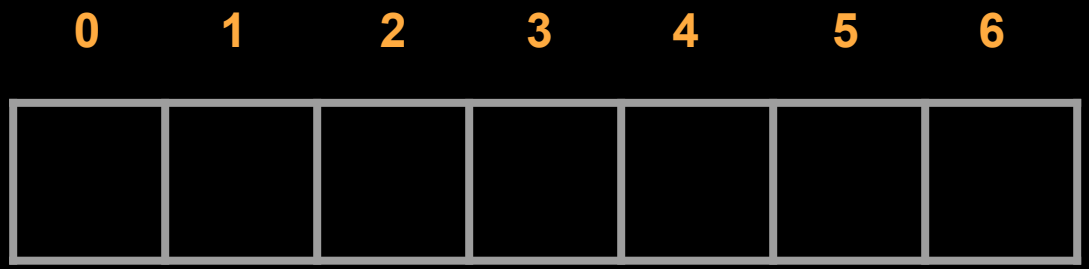
When you multiply by 36 and add the character's value, you're essentially treating the string as a base-36 number. Each character represents a digit in this base-36 system. If you're primarily hashing alphanumeric strings, this works well because each character maps directly to a value within the base.

N = 7



hash function : value % N

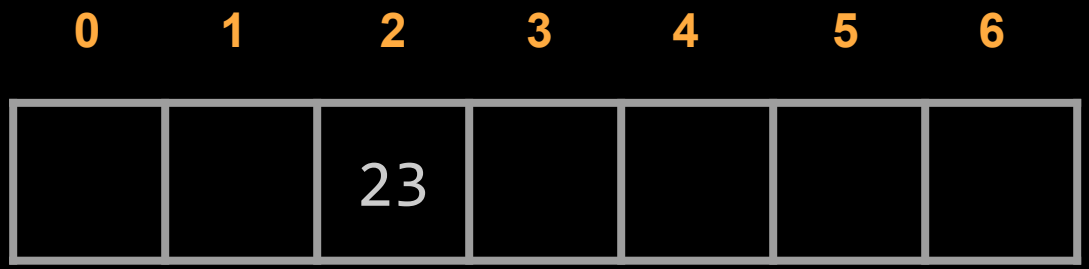
N = 7



hash function : value % N

23

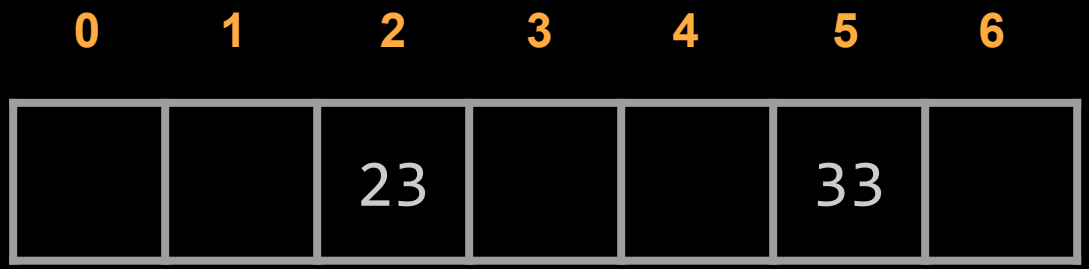
N = 7



hash function : value % N → 2

23

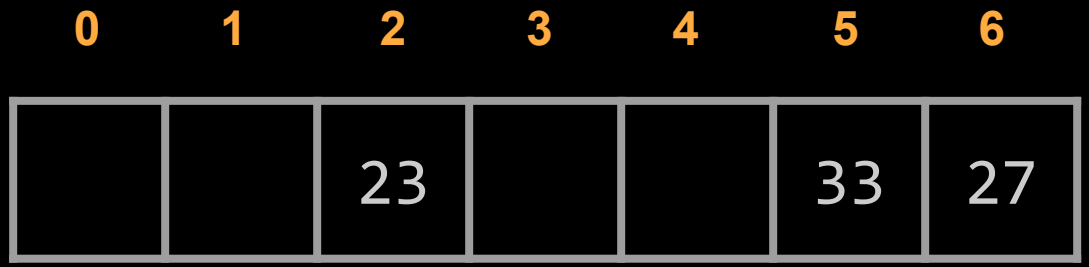
N = 7



33

hash function : value % N → 5

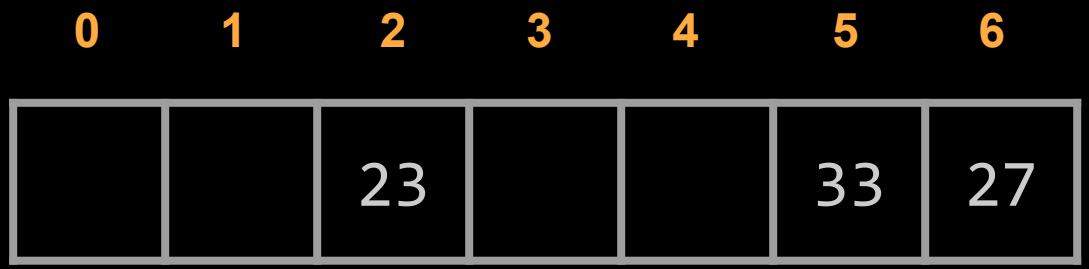
N = 7



27

hash function : value % N → 6

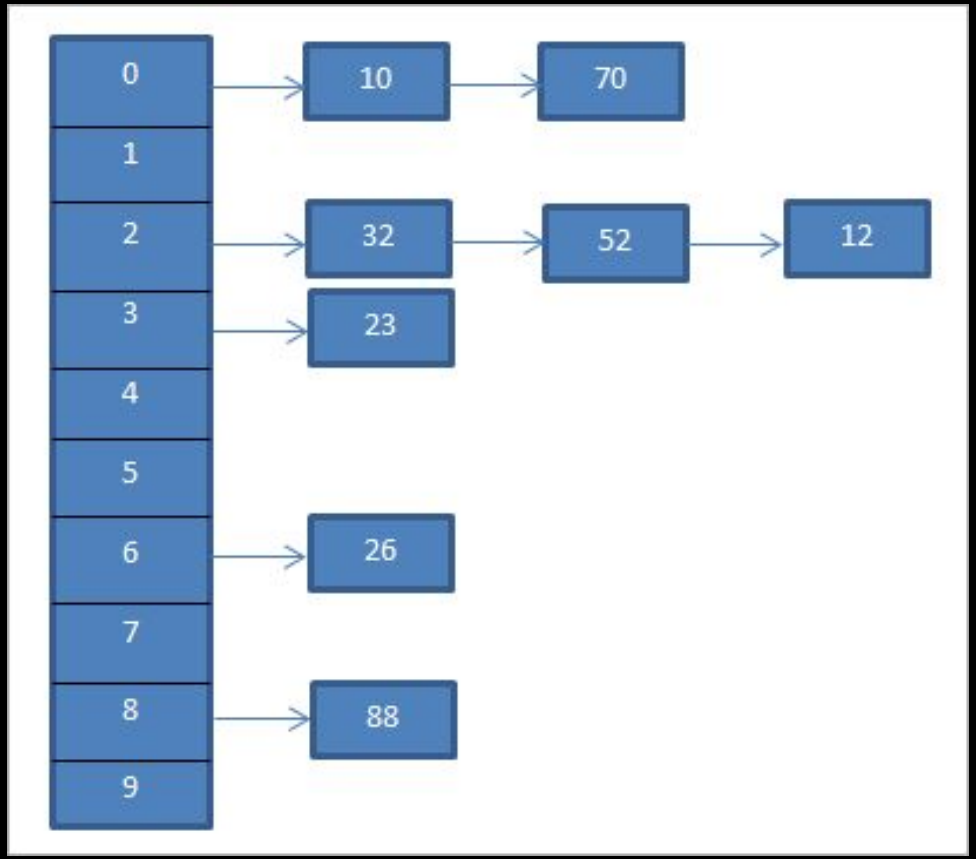
N = 7



51

hash function : value % N → 2

separate chaining
(linked list implementation)


0
1
2
3
4
5
6
7
8
9
10
70
32
52
12
23
26
88

```java
public class LinkedHashMap {
    6 usages
    LinkedList[] table;
    3 usages
    int N;

    no usages  neslihancesurr
    public LinkedHashMap(int N){
        table = new LinkedList[N];
        this.N = N;
        for (int i = 0; i < table.length; i++){
            table[i] = new LinkedList();
        }
    }
```

search

```java
public Node search(int searched){
    int index = hashFunction(searched);
    return table[index].searchNode(searched);
}
```

```java
public Node searchNode(int i){
    Node tmp = head;

    while (tmp != null){
        if (tmp.data == i){
            return tmp;
        }
        tmp = tmp.next;
    }
    return null;
}
```

Find the hash value of the search item.

Go to that index in the array and search the linked list with the linked list ***searchNode*** function.

The ***searchNode*** method will return the node (if it exists) and the return key of hashmap method will return the same node back to main method.

insertion

```java
public void insertion(Node inserted){
    int index = hashFunction(inserted.data);
    table[index].insertLast(inserted);
}
```

Find the hash value of the item to be inserted, reaching its value with *.data*.

Go to that index in the array and insert the node at the end of linked list with the ***insertLast*** method in LinkedList class.

deletion

```java
public void deletion(Node deleted){
    int index = hashFunction(deleted.data);
    LinkedList list = table[index];
    Node deletedNode = list.searchNode(deleted.data);

    if (deletedNode != null){
        if (deletedNode == list.head){
            list.deleteFirst();
        } else if (deletedNode == list.tail){
            list.deleteLast();
        } else {
            list.deleteMiddle(deletedNode);
        }
    }
}
```

Find the hash value of the item to be deleted, reaching its value with *.data*.

Save the linked list found at this index in a new variable.

Using the same ***searchNode*** method, find the node to be deleted.

If the node to be deleted is the head of the linked list, call the ***deleteFirst*** method. If it's the tail, call ***deleteLast*** and else call ***deleteMiddle***.

open addressing
(array implementation)

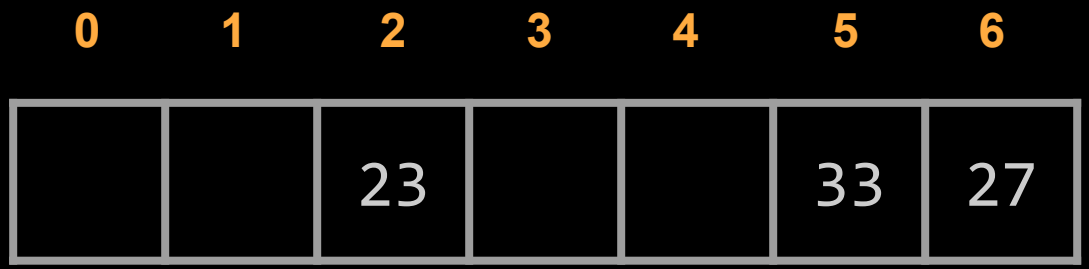
```java
public class HashMap {
    22 usages
    Element[] table;
    14 usages
    boolean[] deleted;
    21 usages
    int N;
    5 usages
    int currentSize;

    2 usages  neslihancesurr
    public HashMap(int N){
        table = new Element[N];
        deleted = new boolean[N];
        this.N = N;
        currentSize = 0;
    }
```

This implementation makes use of a static Element array.

As the array is immutable, another boolean array is used to mark which elements are deleted.

N is the size of the array. It is preferable for N to be a prime number.

N = 7



51

hash function : value % N → 2

linear probing

In linear probing, ***f*(i) = 1**:

hashValue + 1 % N → 2 + 1 % 7 = 3

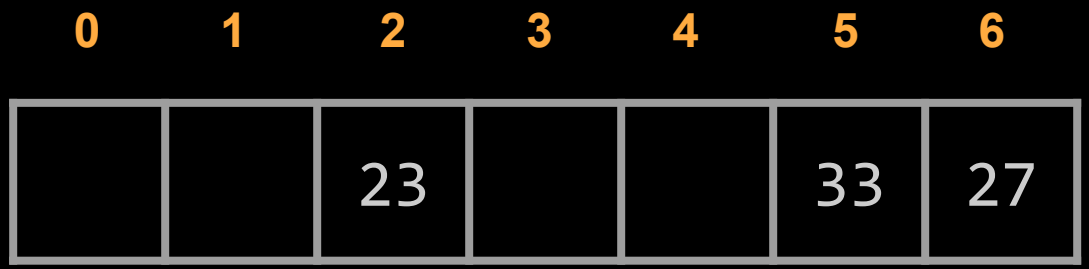
hashValue + 1 % N → 3 + 1 % 7 = 4

hashValue + 1 % N → 4 + 1 % 7 = 5

… until we find an empty space.

Function *f* is the collision resolution strategy.

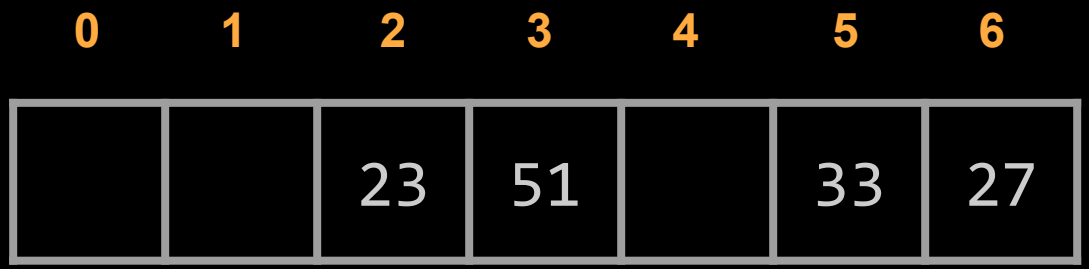
N = 7



51

hash function : value % N → 2

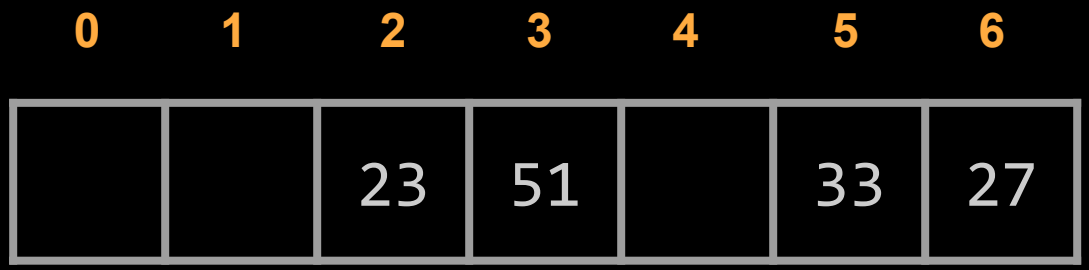
N = 7



hashValue + 1 % N → 2 + 1 % 7 = 3

51

N = 7



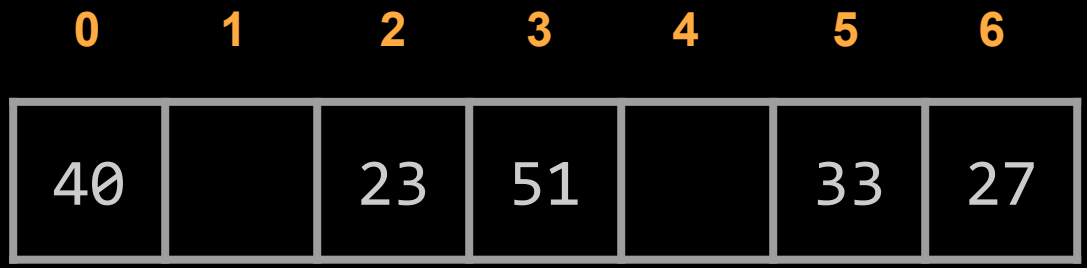
40 % 7 = 5 → 5 is full

40

hashValue + 1 % N → 5 + 1 % 7 = 6 — 6 is full

hashValue + 1 % N → 6 + 1 % 7 = 0 — 0 is empty

N = 7



40 % 7 = 5 → 5 is full

40

hashValue + 1 % N → 5 + 1 % 7 = 6 6 is full

hashValue + 1 % N → 6 + 1 % 7 = 0 0 is empty

linear search

```java
public Element searchLinear(int searched){
    int address = hashFunction(searched);

    while (table[address] != null){
        if (!deleted[address] && table[address].data == searched){
            break;
        }
        address = (address + 1) % N;
    }
    return table[address];
}
```

Find the hash value for the searched integer.

While the index is not null keep looking for a place where value is not deleted and the data matches the searched item. If found, break. If not, modify the index with linear probing and keep going.

When we break from the loop, return the found element (can be null here).

linear insertion

```java
public void insertLinear(Element inserted){
    if (currentSize == N){
        System.out.println("The table is full. Rehashing...");
        rehash();
    }
    int index = hashFunction(inserted.data);

    while (table[index] != null){
        if (deleted[index]){
            deleted[index] = false;
            break;
        }
        index = (index + 1) % N;
    }
    table[index] = inserted;
    currentSize++;
}
```

If table is full, rehash.

Find the hash value for the element to be inserted (*.data*).

While the index is not empty, keep looking for a place where deleted table says true. If found, change the deleted to false and break. If not, modify the index with linear probing and **keep looking for either and empty or deleted space.**

When we break from the loop, add the new element to the found index and increase current size.

linear deletion

```
public void deleteLinear(Element tobeDeleted){
    int index = hashFunction(tobeDeleted.data);

    while (table[index] != null) {
        if (!deleted[index] && table[index] == tobeDeleted){
            break;
        }
            index = (index + 1) % N;
    }
    deleted[index] = true;
}
```

Find the hash value for the element to be deleted (*.data*).

Until you reach the next empty space, keep looking for a place where the value is not deleted and the element matches the deleted element. If found, break. If not, modify the index with linear probing and keep looking for the next one.

When we break from the loop, make the index at the deleted array true to indicate the value there is deleted.

As long as the hash table is large enough, we can find an empty position with linear probing.

But linear probing can spend much more time to find this empty position.

Worse than that, if the table is empty, blocks of full positions (primary clusters) starts forming.

When the hash function maps a value to a position in such a cluster, more attempts are required to resolve that collusion, which takes time.

quadratic probing

In linear probing, ***f*(i) = i^2** :

hashFunction(x) + f(1) % N → 2 + 1 % 7 = 3

hashFunction(x) + f(2) % N → 3 + 4 % 7 = 5

hashFunction(x) + f(3) % N → 5 + 9 % 7 = 0

… until we find an empty space.

Function *f* is the collision resolution strategy.

```java
public Element searchQuadratic(int searched){
    int index = hashFunction(searched);
    int i = 0;

    while (table[index] != null){
        if (!deleted[index] && table[index].data == searched){
            break;
        }
        i++;
        index = (index + i * i) % N;
    }
    return table[index];
}
```

This code does not work properly. It is only to show how the index is updated. Might get into an infinite loop and go to negative integers if the searched value is not in the table.

Quadratic Probing and Table Size:

- When the table size is **a prime number** and the table is **at most half full**, quadratic probing guarantees that you can always find an empty spot for a new element. This is a mathematical property of quadratic probing and prime table sizes.

- However, if the table is **more than half full**, there’s a (rare) chance that **inserting a new element becomes impossible**, even if there are still some empty spots. This happens because the probing sequence might not be able to reach all empty spots due to the quadratic pattern.

Clustering Problems:

- Quadratic probing **solves the primary clustering problem**, which occurs in linear probing when many elements group together in long runs, making collisions more likely in those regions.

- However, it introduces **secondary clustering**, meaning that elements that hash to the same initial position will follow the exact same sequence of alternative positions. This keeps collisions localized in certain patterns.

double hashing

```java
public Element searchDouble(int value){
    int hashValue1 = hashFunction(value);
    int hashValue2 = hashFunctionDouble(value);

    int i = 1;
    while (table[hashValue1] != null){
        if(!deleted[hashValue1] && table[hashValue1].data == value){
            break;
        }
        hashValue1 = (hashValue1 + i * hashValue2) % N;
    }
    return table[hashValue1];
}
```

This code is also incomplete and has problems for certain cases. It is only to show how to use two hash functions in the method.

```java
public int hashFunctionDouble(int value){
    return 1 + (value % (N - 1));
}
```

rehashing

```java
public void rehash(){
    Element[] oldTable = table;
    boolean[] oldDeleted = deleted;
    int oldN = N;

    this.N = this.N * 2;
    this.table = new Element[N];
    this.deleted = new boolean[N];

    for (int i = 0; i < oldN; i++){
        if (!oldDeleted[i] && oldTable[i] != null){
            insertLinear(oldTable[i]);
        }
    }
}
```

Keep the old two tables and N in new variables.

Double the N.

Reinitialize the class fields (table and deleted) with new N.

If the element is not deleted or the index is not null, insert old information to the new array.