Data Structures
COMP242

Ala' Hasheesh
ahashesh@birzeit.edu

Hashing

BIRZEIT UNIVERSITY

Hashing (Properties)

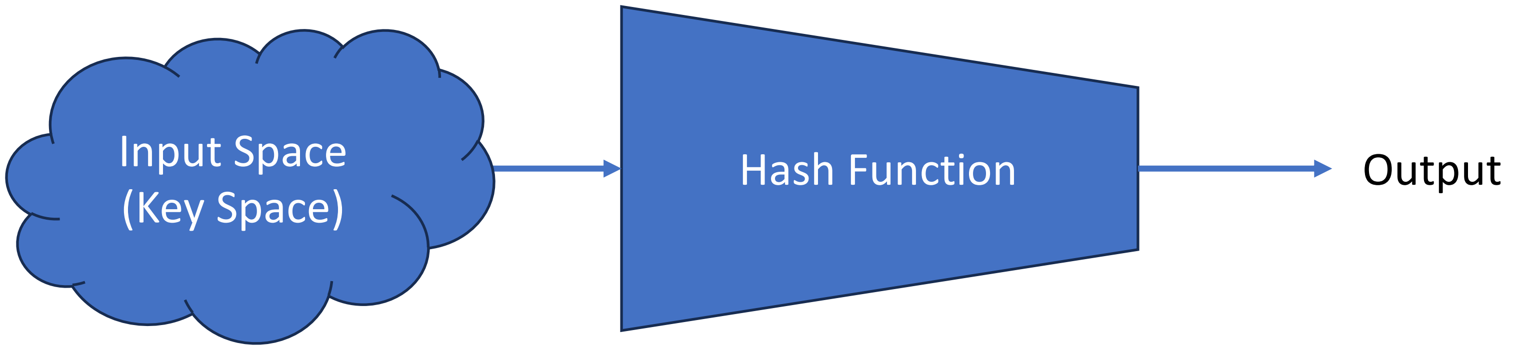
1. Constant time access O(1). Or almost constant depending on the implementation! Depending on the implementation this can grow to O(logn) or O(n)!

2. Hash Table: is an array of fixed size n (usually n is a prime).

3. General Idea is to map a key (i.e., our object) to an index and insert it into the table!

Hashing (Example)

Assume Table size is 7.

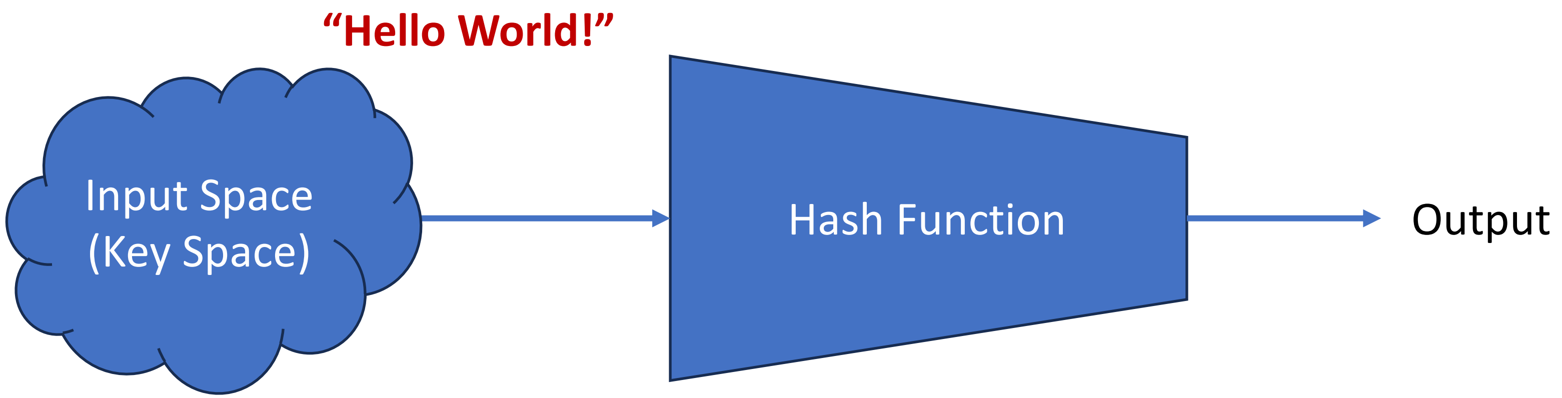


0	
1	
2	
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

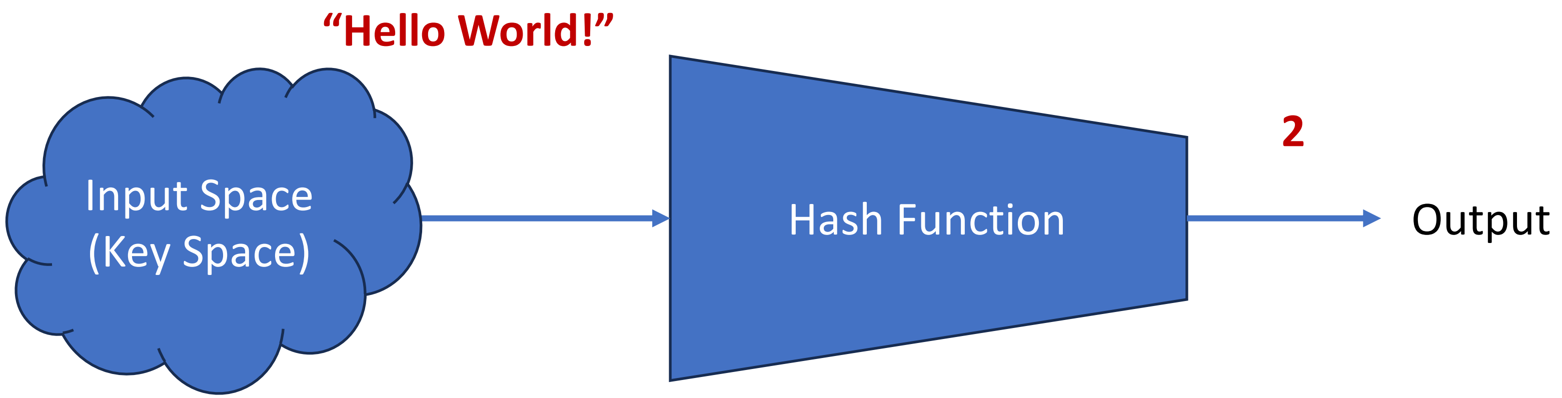


0	
1	
2	
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

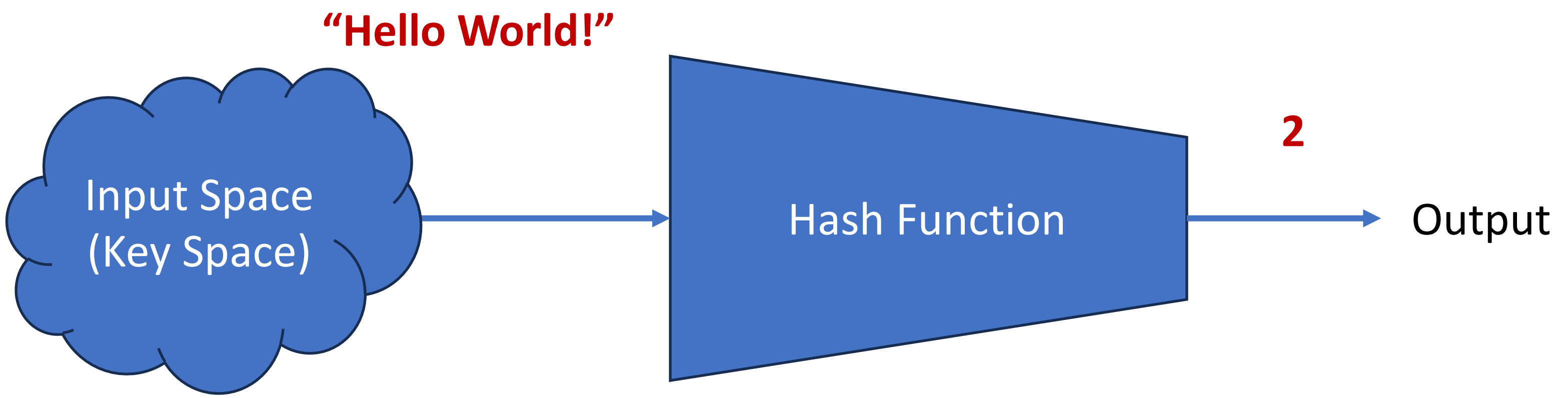


0	
1	
2	
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

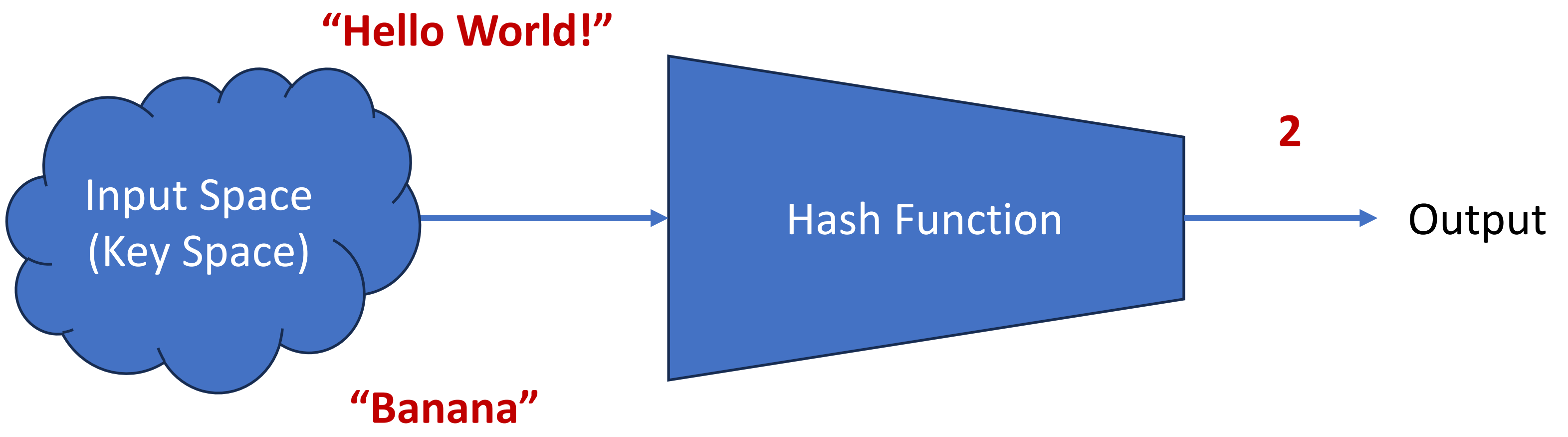


0	
1	
2	Hello World!
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.

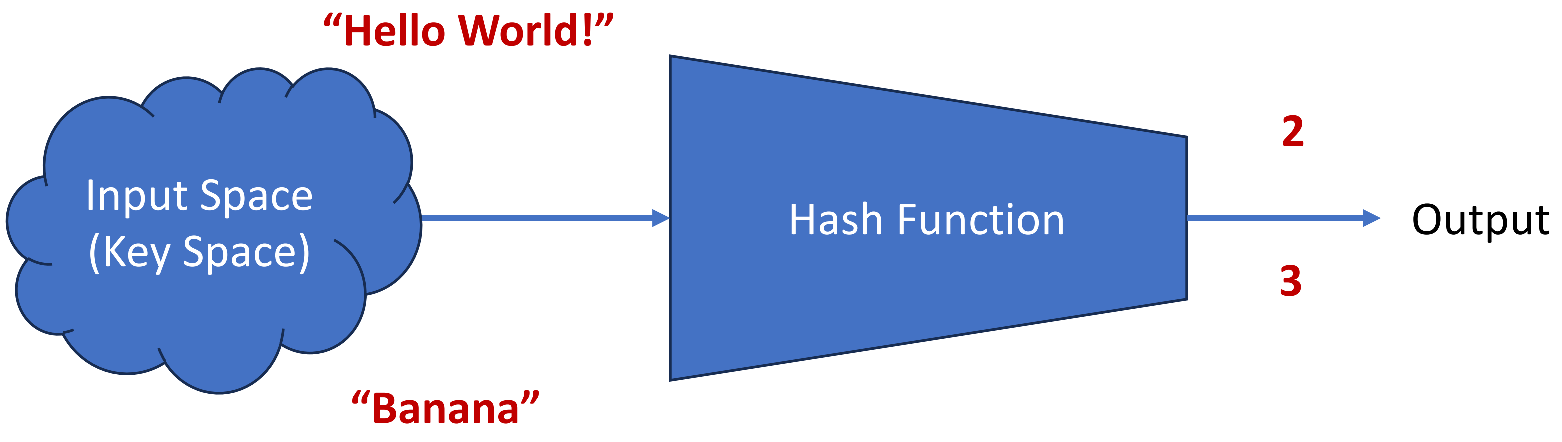


0	
1	
2	Hello World!
3	
4	
5	
6	

Hash Table

Hashing (Example)

Assume Table size is 7.



0	
1	
2	Hello World!
3	Banana
4	
5	
6	

Hash Table

Hashing (Definitions)

1. Constant time access O(1). Or almost constant depending on the implementation! Depending on the implementation this can grow to O(logn) or O(n)!

2. **Hash Table**: is an array of fixed size n (usually n is a prime).

3. General Idea is to map a key to an index and insert it into the table!

4. **Hash Function**: is a special function than maps our element to another value. In a hash table, hash function will map the input into a value in the range [0 – n). From 0 to (n - 1)

5. Hash Function in general maps values from one range (big) to another (small). One way to bound its values into [0 - n-1] is to use mod!

0	
1	
⋮	
n-1	

Hash Table

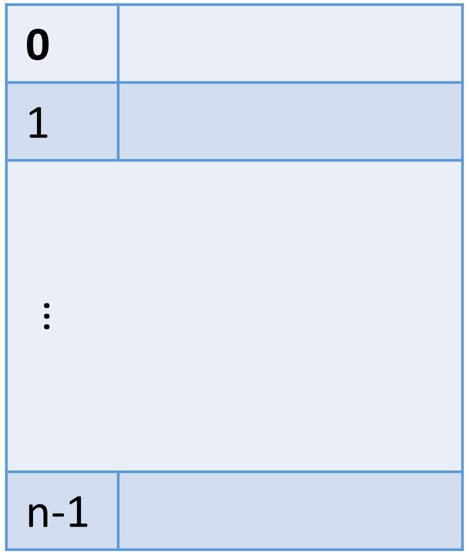
x % n will always produce values in the range [0 - n-1]

Hashing (Hash Function)

1. Don’t use the element directly (even if it’s an integer).

2. Map the element using the hash function into the range [0 - n-1]

3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!



Hash Table

Hashing (Hash Function)

1. Don't use the element directly (even if it's an integer).

2. Map the element using the hash function into the range [0 - n-1]

3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!

4. Mapping is not unique and different inputs can map to the same index!

0	
1	
⋮	
n-1	

Hash Table

Hashing (Hash Function)

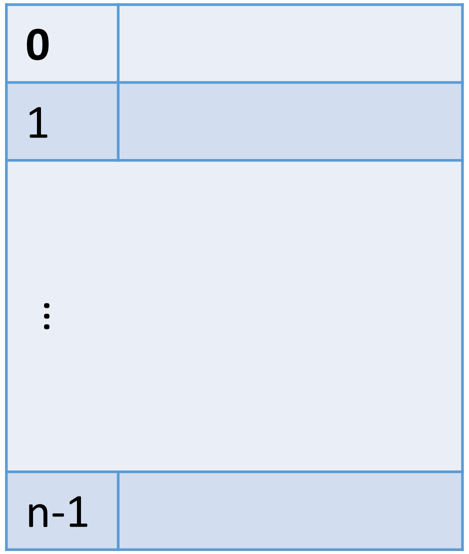
1. Don’t use the element directly (even if it’s an integer).
2. Map the element using the hash function into the range [0 - n-1]
3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!
4. Mapping is not unique and different inputs can map to the same index!
5. Good hash functions produce unique output (most of the time!).

0	
1	
⋮	
n-1	

Hash Table

Hashing (Hash Function)

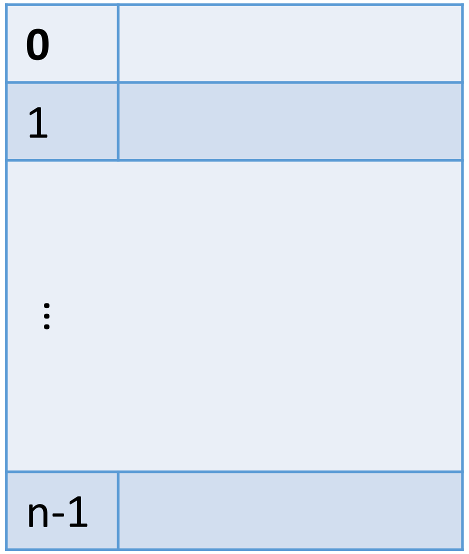
1. Don’t use the element directly (even if it’s an integer).
2. Map the element using the hash function into the range [0 - n-1]
3. In other words, given an array of size **n**, we use a hash function **h(k)** to map Input **x** into some index in the array!
4. Mapping is not unique and different inputs can map to the same index!
5. Good hash functions produce unique output (most of the time!).
6. When two inputs are mapped into the same output (same index) we call that a **Collision**.



Hash Table

Hashing

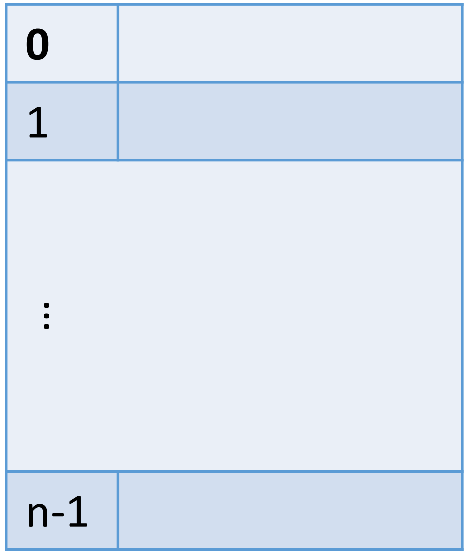
- Hash an input (any input of any type) and produce an index as an output!



Hash Table

Hashing

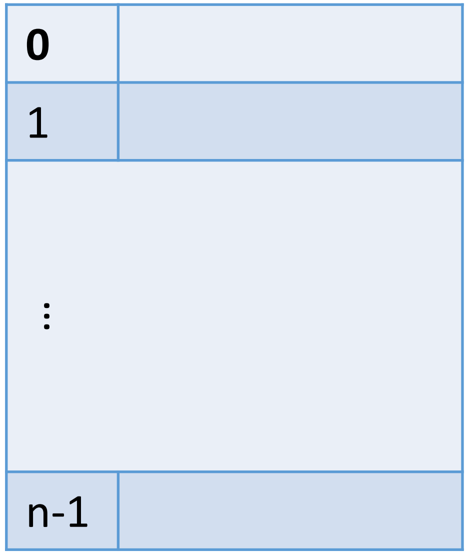
- Hash an input (any input of any type) and produce an index as an output!
- If the hash function is good with unique output this will give **O(1)** access time. Which is a great improvement over **O(logn)**.



Hash Table

Hashing

- Hash an input (any input of any type) and produce an index as an output!
- If the hash function is good with unique output this will give **O(1)** access time. Which is a great improvement over **O(logn)**.
- The main challenge is to find a good hashing function and storage strategy.



Hash Table

Hashing (Key, Value)

In most hashing data structure, we use two storing strategies

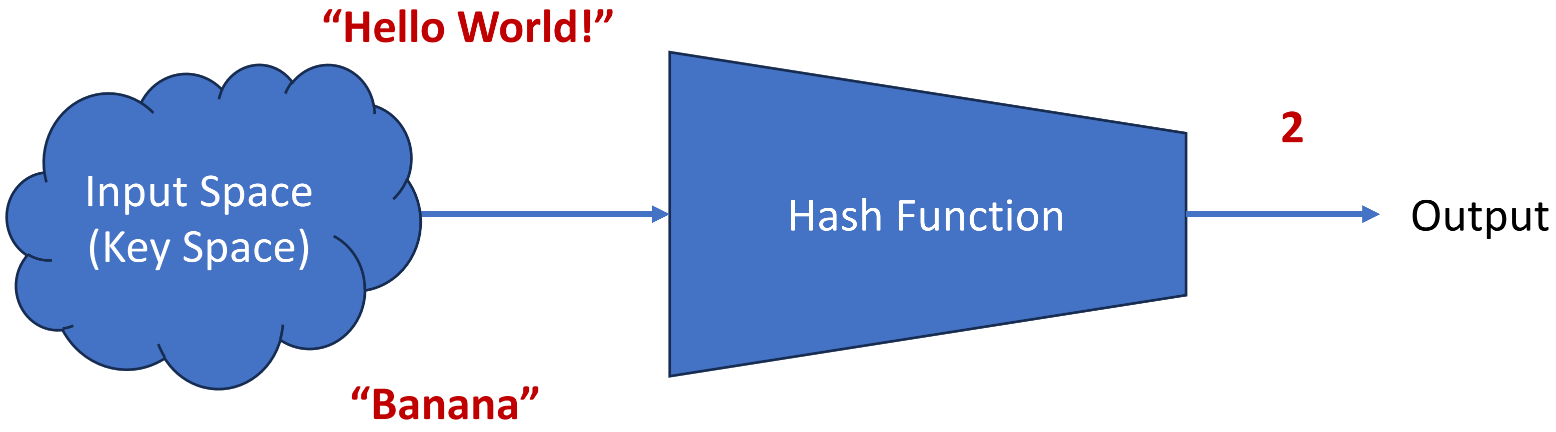
1. (Key, Value) pair where we feed the key to the hash function and store the value.

Hashing (Key, Value)

In most hashing data structure, we use two storing strategies

1. (Key, Value) pair where we feed the key to the hash function and store the value.

   Example: input is (“Hello World!”, “2023-01-14”)



“1210000”, “Ahmad”
“1210001”, “Ahmad”

0	
1	
2	“2023-01-14”
3	
4	
5	
6	

Hash Table

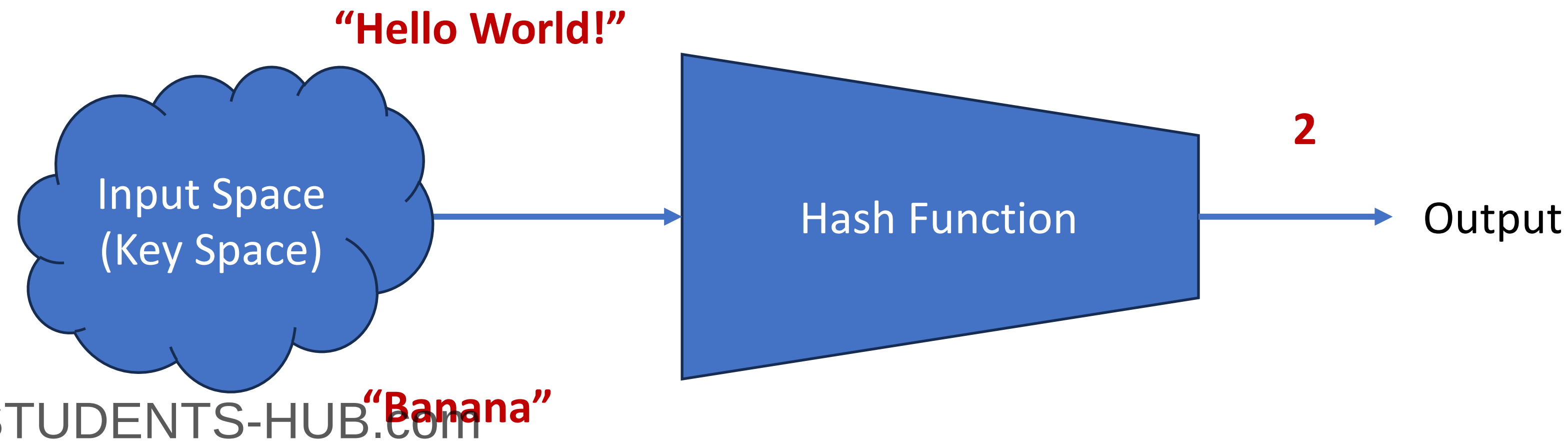
Hashing (Key, Value)

In most hashing data structure, we use two storing strategies

1. (Key, Value) pair where we feed the key to the hash function and store the value.

   Example: input is (“Hello World!”, “2023-01-14”)

   This way we can store any kind of value and build our hashing around **string** keys for example!



0	
1	
2	“2023-01-14”
3	
4	
5	
6	

Hash Table

Hashing (Set)

In most hashing data structure, we use two storing strategies

1. (Key, Value) pair where we feed the key to the hash function and store the value.

2. Just use the value directly to compute the hash and store the value! Usually, we call this a **HashSet**!

Hashing (Set)

In most hashing data structure, we use two storing strategies

1. (Key, Value) pair where we feed the key to the hash function and store the value.

2. Just use the value directly to compute the hash and store the value! Usually, we call this a **Set**!

Hashing (Set)

In most hashing data structure, we use two storing strategies

1. (Key, Value) pair where we feed the key to the hash function and store the value.

2. Just use the value directly to compute the hash and store the value! Usually, we call this a **Set**!

- Duplicate values are usually not allowed in a hash (We can allow them if we use key, value pair), but even then duplicate keys are not allowed!

Hashing (Set)

In most hashing data structure, we use two storing strategies

1. (Key, Value) pair where we feed the key to the hash function and store the value.

2. Just use the value directly to compute the hash and store the value!
   Usually, we call this a **Set**!

- Duplicate values are usually not allowed in a hash (We can allow them if we use key, value pair), but even then duplicate keys are not allowed!

- Since duplicates are not allowed, we can use HashTables to remove them in O(n)!

Hashing (Implementation)

```
public class HashEntry<V> {
    String key;

    V value;

    public HashEntry(String key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public String toString() {
        return String.format("(%s, %s)", key, value);
    }
}
```

Hashing (Implementation)

```
public class HashEntry<V> {
    String key;

    V value;

    public HashEntry(String key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public String toString() {
        return String.format("(%s, %s)", key, value);
    }
}
```

We will talk about how we can make HashEntry accept a generic key during the lab!

Hashing (Implementation)

```
public interface Hashable<V> {
    int getHash(String k);

    void insert(String key, V value);

    boolean contains(String key);

    V find(String key);

    int size();

    boolean isEmpty();
}
```

Hashing (Implementation)

```
public class HashTable<V> {
    HashEntry<V>[] data;

    int size;

    public HashTable() {
        this(10);
    }

    public HashTable(int capacity) {
        data = (HashEntry<V>[]) new Object[capacity];
        size = 0;
    }
}
```

Hashing (Implementation)

```
public void insert(String key, V value) {
    if (shouldRehash()) {
        rehash();
    }

    int hash = getHash(key);

    // Perform quadratic probing
    int i = 1;

    data[hash] = new HashEntry<>(key, value);
}
```

Hashing (Implementation)

We will do the rest during the lab!