DSA Hash Sets

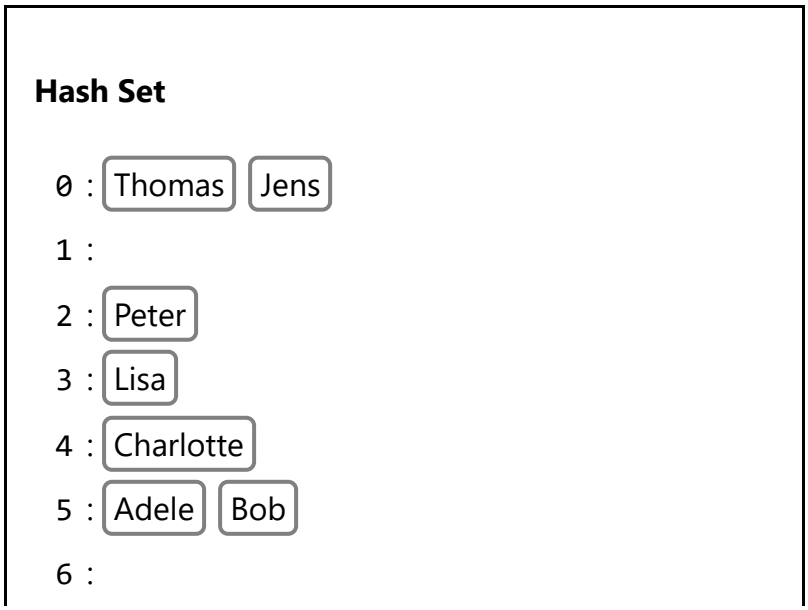
‹ Previous Next ›

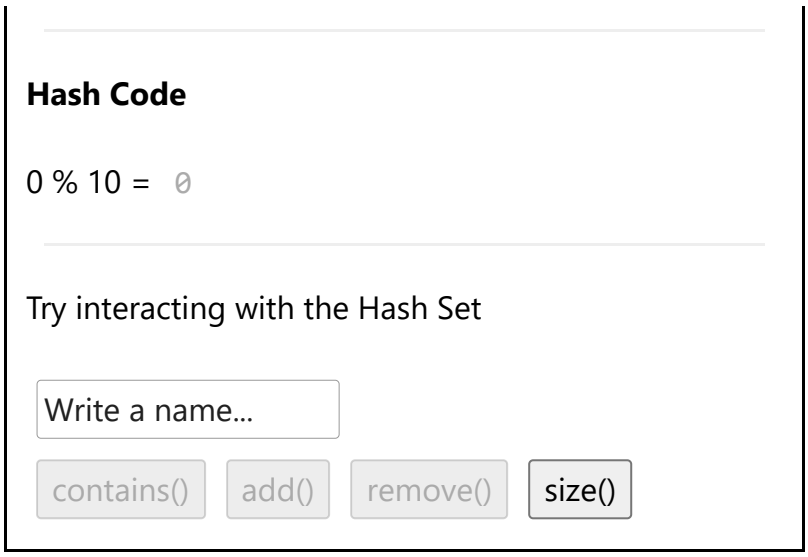
Hash Sets

A Hash Set is a form of Hash Table data structure that usually holds a large number of elements.

Using a Hash Set we can search, add, and remove elements really fast.

Hash Sets are used for lookup, to check if an element is part of a set.





A Hash Set stores unique elements in buckets according to the element's hash code.

- **Hash code:** A number generated from an element's unique value (key), to determine what bucket that Hash Set element belongs to.
- **Unique elements:** A Hash Set cannot have more than one element with the same value.
- **Bucket:** A Hash Set consists of many such buckets, or containers, to store elements. If two elements have the same hash code, they belong to the same bucket. The buckets are therefore often implemented as arrays or linked lists, because a bucket needs to be able to hold more than one element.

Finding The Hash Code

A hash code is generated by a **hash function**.

The hash function in the animation above takes the name written in the input, and sums up the Unicode code points for every character in that name.

After that, the hash function does a modulo 10 operation (`% 10`) on the sum of characters to get the hash code as a number from 0 to 9.

This means that a name is put into one of ten possible buckets in the Hash Set, according to the hash code of that name. The same hash code is generated and used when we want to search for or remove a name from the Hash Set.

The Hash Code gives us instant access as long as there is just one name in the corresponding bucket.

has Unicode code point `65`. Just try it in the simulation above. See this page for more information about how characters are represented as numbers.

Modulo: A mathematical operation, written as `%` in most programming languages (or mod in mathematics). A modulo operation divides a number with another number, and gives us the resulting remainder. So for example, `7 % 3` will give us the remainder `1`. (Dividing 7 apples between 3 people, means that each person gets 2 apples, with 1 apple to spare.)

Direct Access in Hash Sets

Searching for `Peter` in the Hash Set above, means that the hash code `2` is generated (`512 % 10`), and that directs us right to the bucket `Peter` is in. If that is the only name in that bucket, we will find `Peter` right away.

In cases like this we say that the Hash Set has constant time $O(1)$ for searching, adding, and removing elements, which is really fast.

But, if we search for `Jens`, we need to search through the other names in that bucket before we find `Jens`. In a worst case scenario, all names end up in the same bucket, and the name we are searching for is the last one. In such a worst case scenario the Hash Set has time complexity $O(n)$, which is the same time complexity as arrays and linked lists.

To keep Hash Sets fast, it is therefore important to have a hash function that will distribute the elements evenly between the buckets, and to have around as many buckets as Hash Set elements.

Having a lot more buckets than Hash Set elements is a waste of memory, and having a lot less buckets than Hash Set elements is a waste of time.

Hash Set Implementation

Hash Sets in Python are typically done by using Python's own `set` data type, but to get a better understanding of how Hash Sets work we will not use that here.

To implement a Hash Set in Python we create a class `SimpleHashSet`.

We also create a method `print_set` to better see how the Hash Set looks like.

Example

```
class SimpleHashSet:
    def __init__(self, size=100):
        self.size = size
        self.buckets = [[] for _ in range(size)]  # A list of bucke

    def hash_function(self, value):
        # Simple hash function: sum of character codes modulo the n
        return sum(ord(char) for char in value) % self.size

    def add(self, value):
        # Add a value if it's not already present
        index = self.hash_function(value)
        bucket = self.buckets[index]
        if value not in bucket:
            bucket.append(value)

    def contains(self, value):
        # Check if a value exists in the set
        index = self.hash_function(value)
        bucket = self.buckets[index]
        return value in bucket

    def remove(self, value):
        # Remove a value
        index = self.hash_function(value)
        bucket = self.buckets[index]
        if value in bucket:
            bucket.remove(value)

    def print_set(self):
        # Print all elements in the hash set
        print("Hash Set Contents:")
```

Using the `SimpleHashSet` class we can create the same Hash Set as in the top of this page:

Example

```
            print(f"Bucket {index}: {bucket}")

# Creating the Hash Set from the simulation
hash_set = SimpleHashSet(size=10)

hash_set.add("Charlotte")
hash_set.add("Thomas")
hash_set.add("Jens")
hash_set.add("Peter")
hash_set.add("Lisa")
hash_set.add("Adele")
hash_set.add("Michaela")
hash_set.add("Bob")

hash_set.print_set()

print("\n'Peter' is in the set:",hash_set.contains('Peter'))
print("Removing 'Peter'")
hash_set.remove('Peter')
print("'Peter' is in the set:",hash_set.contains('Peter'))
print("'Adele' has hash code:",hash_set.hash_function('Adele'))
```

Try it Yourself »

COLOR PICKER

PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate