Algorithms and Data Structures

Hash Table

Robert Horvick

www.pluralsight.com

Outline

- **Hash Table overview**
  - Associative Array
- **Hashing overview**
- **Add items**
- **Remove items**
- **Search for items**
- **Enumerate Items**

Hash Table Overview

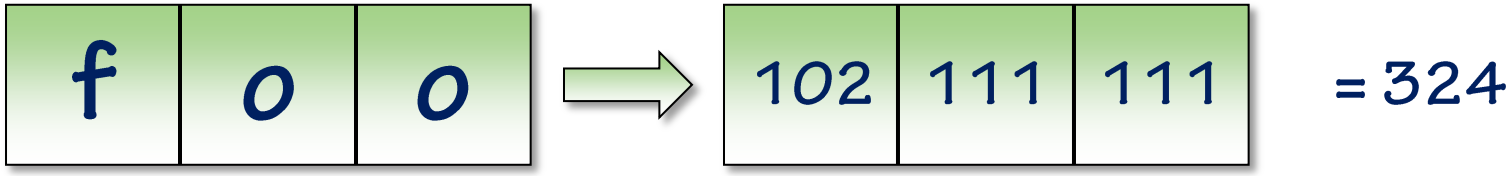
- **Associative Array**
  - Storage of Key/Value Pairs
- **Each key is unique**
- **The key type is mapped to an index**
- **Adding Jane**
  - int index = GetIndex(Jane.Name);
  - _array[index] = Jane;

Hashing Overview

- **Hashing derives a fixed size result from an input**
- **Stable**
  - The same input generates the same output every time
- **Uniform**
  - The hash value should be uniformly distributed through available space
- **Efficient**
  - The cost of generating a hash must be balanced with application needs
- **Secure**
  - The cost of finding data that produces a given hash is prohibitive

Hashing a String

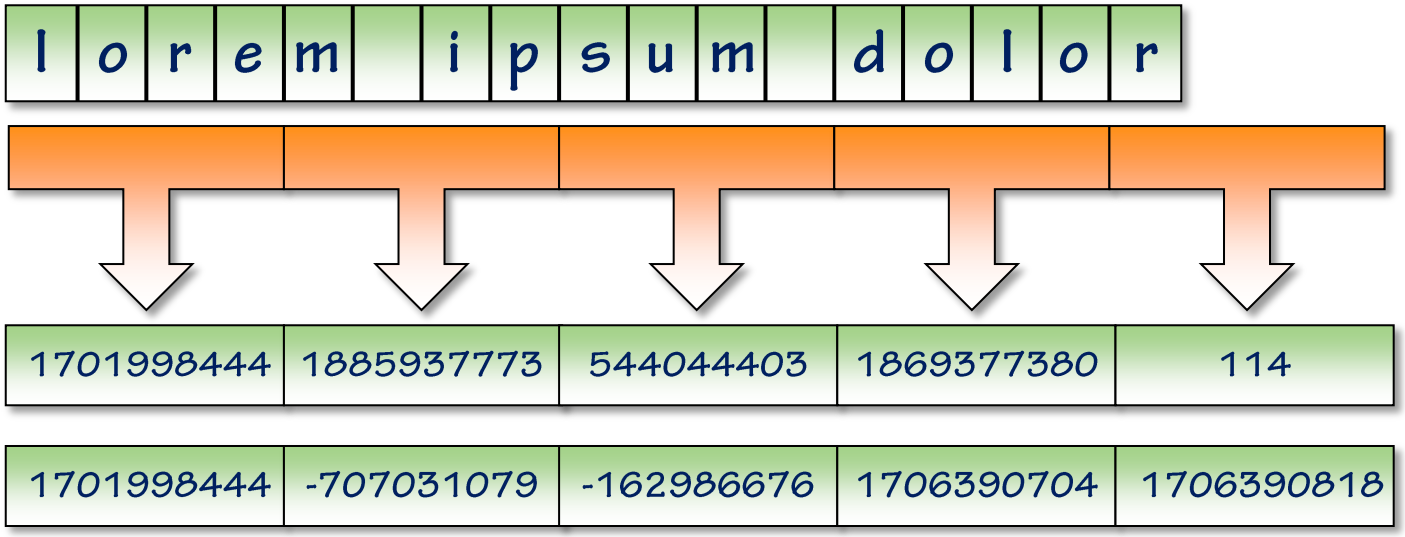
- **Naïve implementation**
  - Summing the ASCII value of each character



- **Pros**
  - Stable
  - Efficient
- **Cons**
  - Not Uniform
    - AdditiveHash("foo") equals AdditiveHash("oof")
  - Not Secure

Hashing a String

- **Somewhat better**
  - "Folds" bytes of every four characters into an integer



- **Stable, Efficient and better Uniformity**
- **Not secure**

Hashing Functions

- Don't write your own hashing algorithm!
- Pick the right hash for the job at hand

Adding Items

- **Adding Values**
  - int arrayLength = 9;
  - int hashCode = GetHashCode(Jane.Name);
  - int index = hashCode % arrayLength;
  - _array[index] = Jane;

Handling Collisions

- **Two distinct items have same hash value**
  - Items are assigned to the same index in the hash table
- **Two common strategies**
  - Open Addressing
    - Moving to next index in table
  - Chaining
    - Storing items in a linked list

Open Addressing

```
int arrayLength = 9;

int hashCode = GetHashCode(Person.Name);

int index = hashCode % arrayLength;

while(_array[index] != null)

        Index++

_array[index] = Person;
```

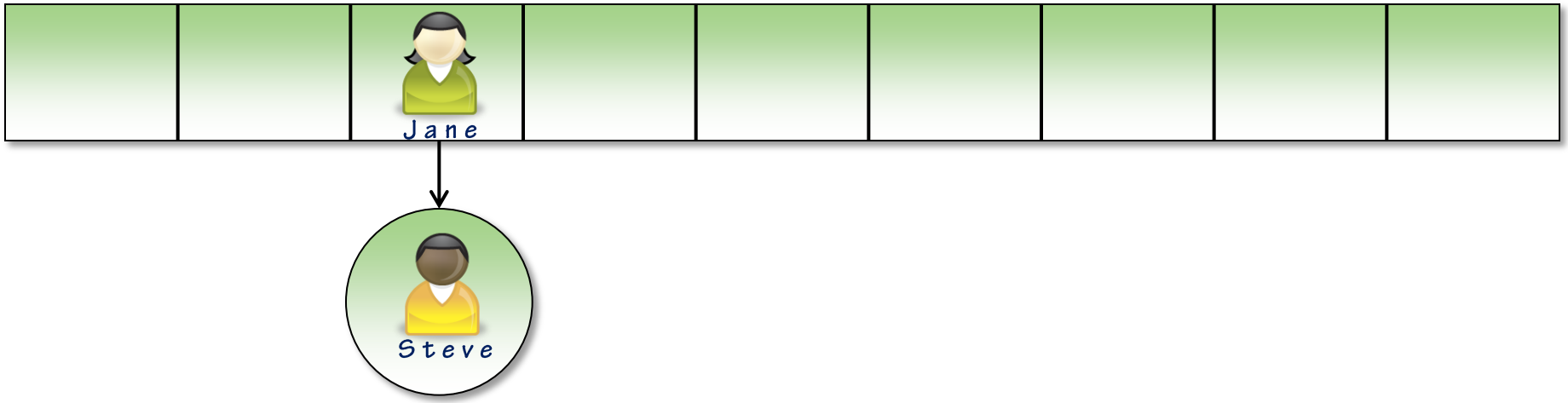
Chaining

```
int arrayLength = 9;

int hashCode = GetHashCode(Person.Name);

int index = hashCode % arrayLength;

_array[index].AddFirst(Person);
```



Growing the Collection

- **Load Factor**
  - The ratio of filled hash table array slots
  - Also known as “fill factor”
- **Add(item)**

```
if ( fillFactor >= maxFillFactor ) {
        _newArray = new ArrayType[_array.Length * 2];
        foreach ( item in _array ) {
                AddItemToHashTable(_newArray, item);
        }
        _array = _newArray;
}
AddItemToHashTable(_array, item);
```

Removing Items

- **Items are removed by key**
  - HashTable.Remove("Jane");
- **Open Addressing**
  - Get the index of the key
  - If the value at the index is non-null
    - If the keys match, remove
    - If the keys don't match, check the next index
- **Chaining**
  - Get the index of the key
  - Remove the item from the linked list

Finding Items

- **Items are found by key**
  - Person p = HashTable.Find("Jane");
- **Open Addressing**
  - Get the index of the key
  - If the value at the index is non-null
    - If the keys match, return the value
    - If the keys don't match, check the next index
- **Chaining**
  - Get the index of the key
  - Find the item in the list

Enumerating Keys and Values

- **Open Addressing**

```
foreach ( item in _array ) {
        if (item != null )
                return item;
}
```

- **Chaining**

```
foreach ( list in _array ) {
        if (list != null ) {
                foreach ( item in list ) {
                        return item;
                }
        }
}
```

Summary

- **Hash table overview**
  - Associative array
- **Hashing overview**
  - Stable, Uniform, Efficient, Secure
- **Add items**
  - Open Addressing
  - Chaining
- **Remove, Search and Enumerate**
  - Depend largely on the collision handling