Organizing Data: Hash Tables

Recall Arrays and Linked Lists

	Advantages	Disadvantages
Arrays	Constant-time lookup (search) if you know the index	Requires a contiguous block of memory
Linked Lists	Flexible memory usage	Linear-time lookup (search)

How can we exploit the advantages of arrays and linked lists to improve search time in dynamic data sets?

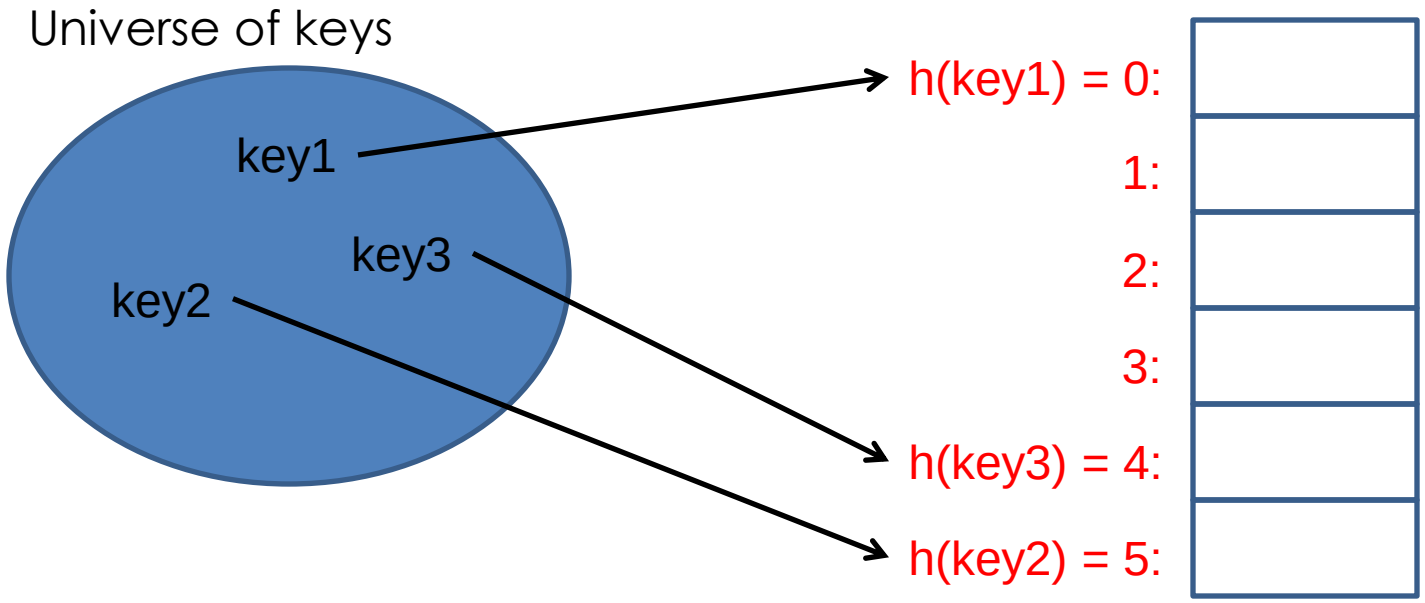
Key-Value Pairs

- Many applications require a flexible way to look up information.
  - For example, we may want to retrieve information (value) about employees based on their SSN (key).
- **Associative arrays**: collection of (key, value pairs)
  - Key-value pair examples: name-phone number, username-password pairs, zipcode-shipping costs
- If we could represent the key as an integer and store all data in an array we would have constant look up time. Can we always do that?

No, memory is a bounded resource.

Hashing

- A "hash function" `h(key)` that maps a key to an array index in 0..*k*-1.
- To search the array `table` for that key, look in `table[h(key)]`



A hash function `h` is used to map keys to hash-table (array) slots. Table is an array bounded in size. The size of the universe for keys may be larger than the array size. We call the table slots buckets.

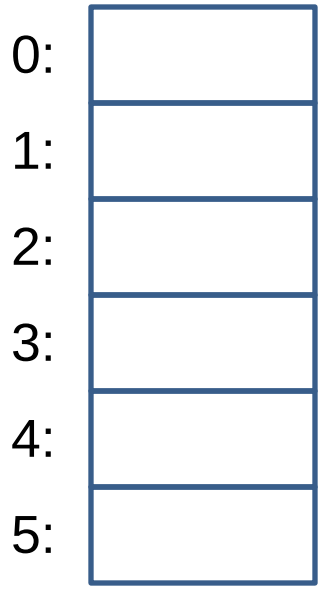
Example: Hash function

- Suppose we have (key,value) pairs where the key is a string such as (name, phone number) pairs and we want to store these key value pairs in an array.
- We could pick the array position where each string is stored based on the first letter of the string using this hash function:

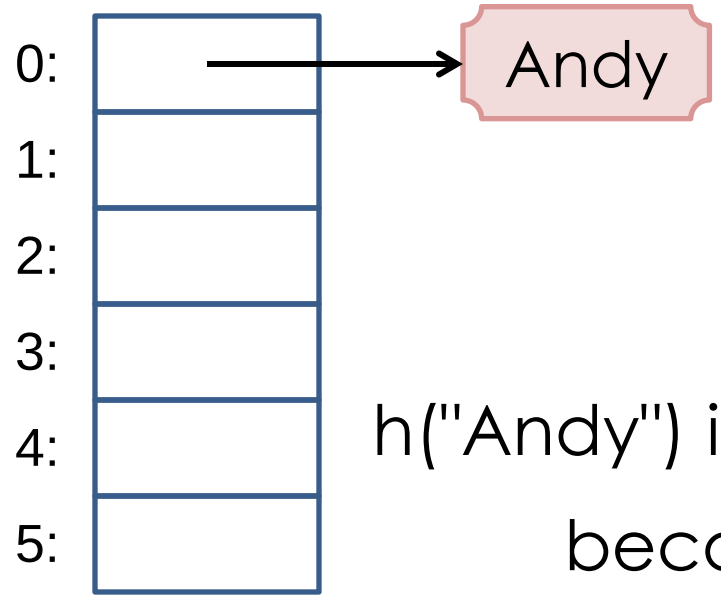
```
def h(str):
    return (ord(str[0]) – 65) % 6
```

Note ord('A') = 65

An Empty Hash Table



Add Element "Andy"



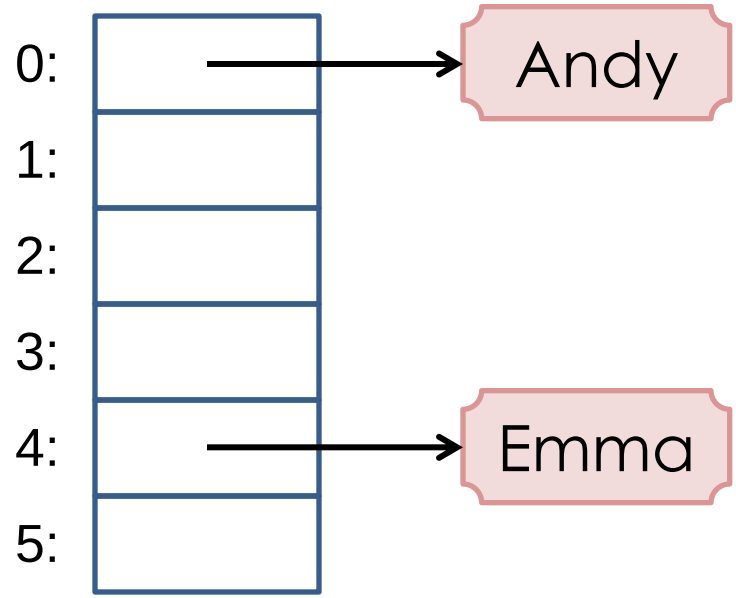
h("Andy") is 0

because ord("A") = 65

and (65-65) % 6 = 0.

Suppose we use the function h from the previous slide.

Add Element “Emma”

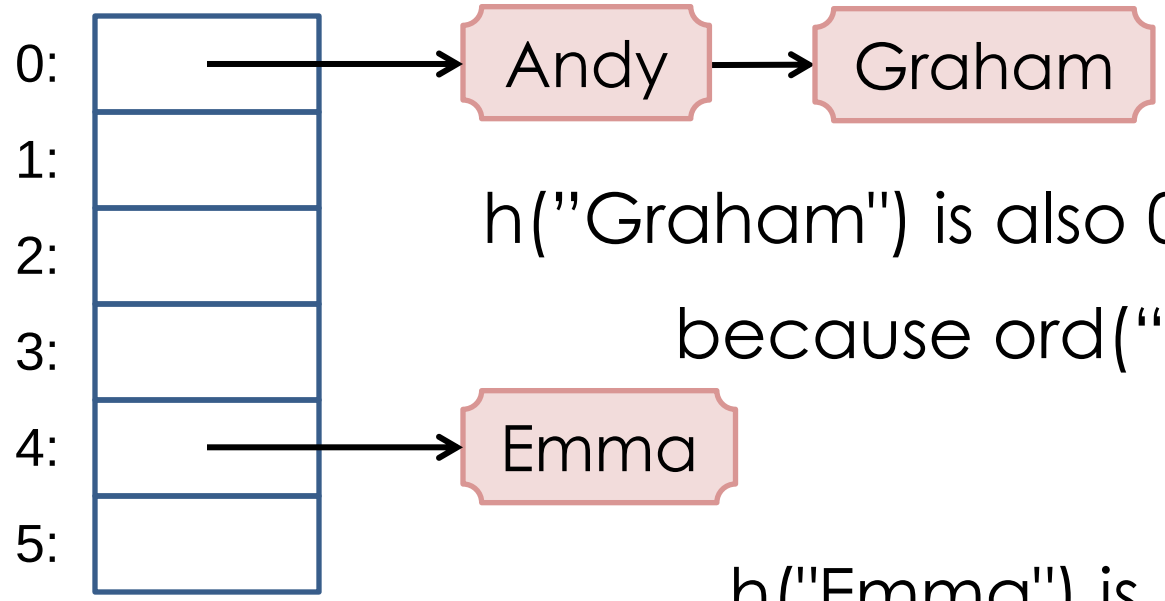


h(“Emma") is 4

because ord(“E") = 69

and (69 – 65) % 6 = 4.

Add Element "Graham"



h("Graham") is also 0

because ord("G") is 71.

h("Emma") is 4

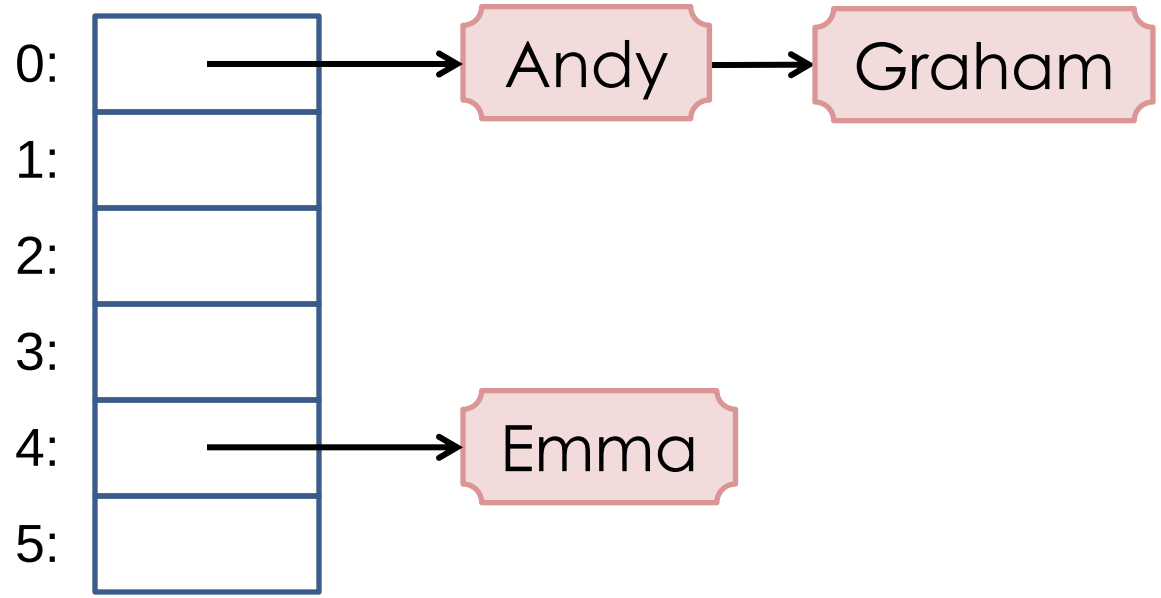
In order to add Graham's information to the table we had to form a linked list for bucket 0.

Looking Up Values in Hash Tables

- Convert a key (e.g. string) to an integer and convert this integer to a bucket number using % (hash function)

- Get this bucket to from the table and then traverse it to find the key we want

Add Element "Graham"



"Andy" and "Graham" end up in the same bucket.

These are **collisions** in a hash table.

Why do we want to minimize collisions?

Minimizing Collisions

- The more collisions you have in a bucket, the more you have to search in the bucket to find the desired element.

- We want to try to minimize the collisions by creating a hash function that distribute the keys (strings) into different buckets as evenly as possible.

Requirements for the Hash Function h(x)

- Must be fast: O(1)
- Must distribute items roughly uniformly throughout the array, so everything doesn't end up in the same bucket.

What's A Good Hash Function?

- For strings:
  - Treat the characters in the string like digits in a base-256 number.
  - Divide this quantity by the number of buckets, *k*.
  - Take the remainder, which will be an integer in the range 0..*k*-1.

Fancier Hash Functions

- How would you hash an integer i?
  - Perhaps i % k would work well.
- How would you hash a list?
  - Sum the hashes of the list elements.
- How would you hash a floating point number?
  - Maybe look at its binary representation and treat that as an integer?

Efficiency

- If the keys (strings) are distributed well throughout the table, then each bucket will only have a few keys and the search should take O(1) time.
- Example:
  If we have a table of size 1000 and we hash 4000 keys into the table and each bucket has approximately the same number of keys (approx. 4), then a search will only require us to look at approx. 4 keys => O(1)
  - But, the distribution of keys is dependent on the keys and the hash function we use!

Summary of Search Techniques

Technique	Setup Cost	Search Cost
Linear search	0, since we're given the list	O(n)
Binary search	O(n log n) to sort the list	O(log n)
Hash table	O(n) to fill the buckets	O(1)

Associative Arrays

- Hashing is a method for implementing associative arrays. Some languages such as Python have associate arrays (**mapping** between keys and values) as a built-in data type.

- Examples:
  - Name in contacts list => Phone number
  - User name => Password
  - Product => Price

Dictionary Type in Python

- This example maps car brands (*keys*) to prices (*values*).

```
>>> cars = { "Mercedes": 55000,
             "Bentley": 120000,
             "BMW": 90000}
>>> cars["Mercedes"]
55000
```

- Dictionary keys can be of any immutable data type.

Iteration with Dictionaries

```
>>> for i in cars:
        print(i)

BMW
Mercedes
Bentley
```

Think what the loop variable is bound to in each case.

```
>>> for i in cars.items():
        print(i)

("BMW", 90000)
("Mercedes", 55000)
("Bentley", 120000)
```

```
>>> for k,v in cars.items():
        print(k, ":", v )

BMW : 90000
Mercedes : 55000
Bentley : 120000
```

Dictionaries have no notion of order: No first element, second element in a dictionary.

Some Dictionary Operations

- `d[key] = value -- Set d[key] to value.`
- `del d[key] -- Remove d[key] from d. Raises a an error if key is not in the map.`
- `key in d -- Return True if d has a key key, else False.`
- `items() -- Return a new view of the dictionary's items ((key, value) pairs).`
- `keys() -- Return a new view of the dictionary's keys.`
- `pop(key[, default]) If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, an error is raised.`

Source: https://docs.python.org/

Next

- Finish data structures unit with trees and graphs
- New unit in data representation