

Hashtables and their Applications

COMP2611: Data Structures

September 2019

Q: how would you build a (basic) search engine?

What's so Hard about Search Engines?

"The **Google** Search **index** contains **hundreds of billions of webpages** and is well over 100,000,000 gigabytes in **size**."

How Google Search Works | Crawling & Indexing

<https://www.google.com> › search › crawl...

About **40,600,000 results** (0.93 seconds)

 www.uwi.edu ▼

Home - The University of the West Indies

The University of the West Indies. ... Publications. Special Report; CHILL; MONA News; STAN; **UWI** Connect; University Reports; 70th Anniversary ...

[UWI St. Augustine](#) · [UWI Open Campus](#) · [myCampusNEWEE](#) · [UWI, Mona](#)

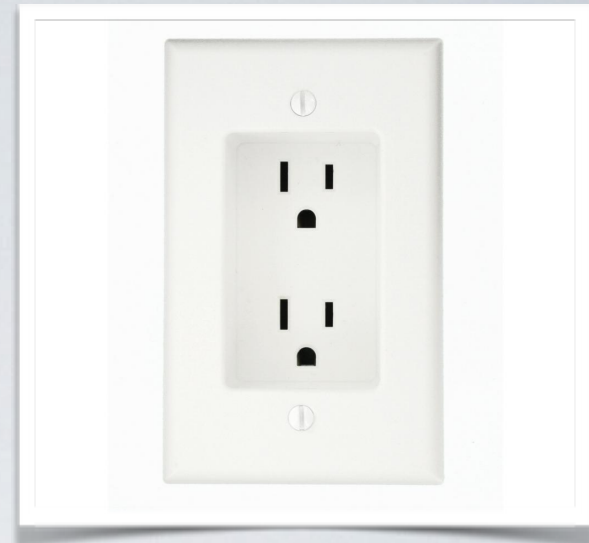
Search Through Each Page?

- ▶ Assume Google indexes **200,000,000,000** pages
- ▶ If we could scan **1** page in **1** microsecond
 - ▶ one search would take **55** hours
- ▶ How do we improve search time
 - ▶ when we have to look through billions of documents?

Outline

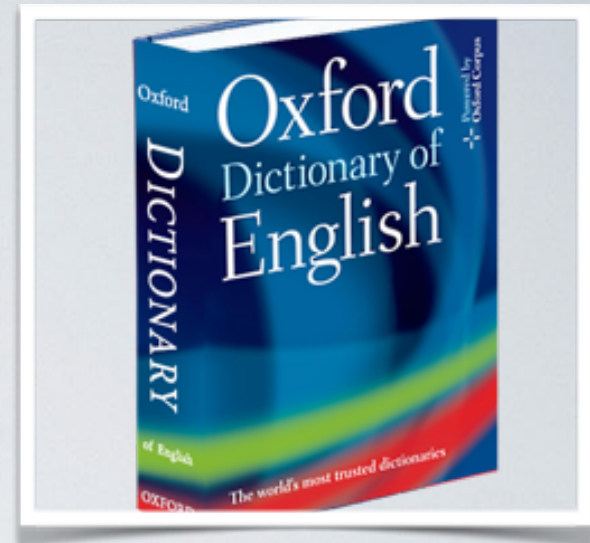
- ▶ Dictionary ADT
 - ▶ and its Naive Implementation
- ▶ Hashtables
 - ▶ Collisions and how to resolve them
 - ▶ Methods:
 - ▶ Chaining
 - ▶ Linear Probing
 - ▶ Quadrating Probing
 - ▶ Double Hashing
- ▶ Set ADT
 - ▶ Using Hashtables as sets

ADT - Abstract Data Type



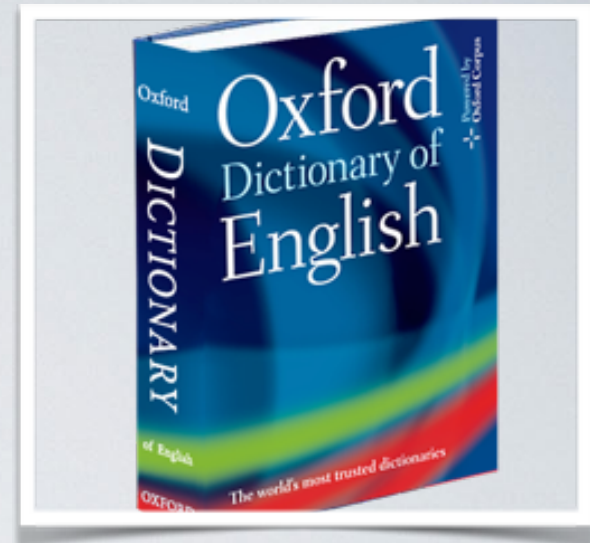
- ▶ Describes minimal operations that a data type must satisfy
- ▶ Think Interfaces from Java and C#
- ▶ Does not describe how to achieve functionality
 - ▶ Functionality achieved through data structures!
 - ▶ Different underlying data structures provide different performances

Dictionary ADT



- ▶ Collection of key/value pairs
 - ▶ Each key is unique
 - ▶ Keys are **unordered**
- ▶ Primarily concerned with lookup and storage by key
- ▶ Sometimes also called a **map**
 - ▶ maps keys to values
 - ▶ ex: student id → student record; word → definition
 - ▶ one-to-one mapping but can emulate one-to-many using lists (index)

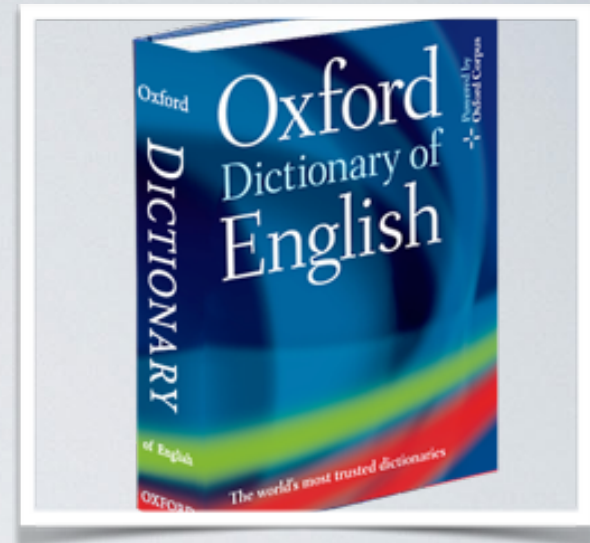
Dictionary ADT



- ▶ Few main operations
 - ▶ **insert**(key, value): inserts a key/value pair in the dictionary. If a value already exists for that key, it overwrites that value
 - ▶ **search**(key): searches the dictionary for a key and returns the associated value
 - ▶ **delete**(key): deletes this key and its associated value from the dictionary

Dictionary ADT

- ▶ Dictionaries are found everywhere!
 - ▶ Routing tables in Routers
 - ▶ Symbol Tables in Compilers and Interpreters
 - ▶ Indices in Databases (unordered access)
 - ▶ DNA Sequence analysis (Assignment #1)
 - ▶ NLP techniques
 - ▶ etc...



Dynamic Array Based Dictionary

- ▶ **insert**(key, value): scans the array. If a value for this key exists, then replace it, else insert at the end
- ▶ **search**(key): scans array until it finds the key in the array
- ▶ **remove**(key): scans array, finds key in array and deletes element, moving previous entries up

Dynamic Array Based Dictionary

- ▶ **insert**(key, value): **scans the array**. If a value for this key exists, then replace it, else insert at the end
- ▶ **search**(key): **scans array** until it finds the key in the array
- ▶ **remove**(key): **scans array**, finds key in array and deletes element, moving previous entries up

Dynamic Array Based Dictionary

- ▶ **insert**(key, value): **scans the array**. If a value for this key exists, then replace it, else insert at the end
- ▶ **search**(key): **scans array** until it finds the key in the array
- ▶ **remove**(key): **scans array**, finds key in array and deletes element, moving previous entries up

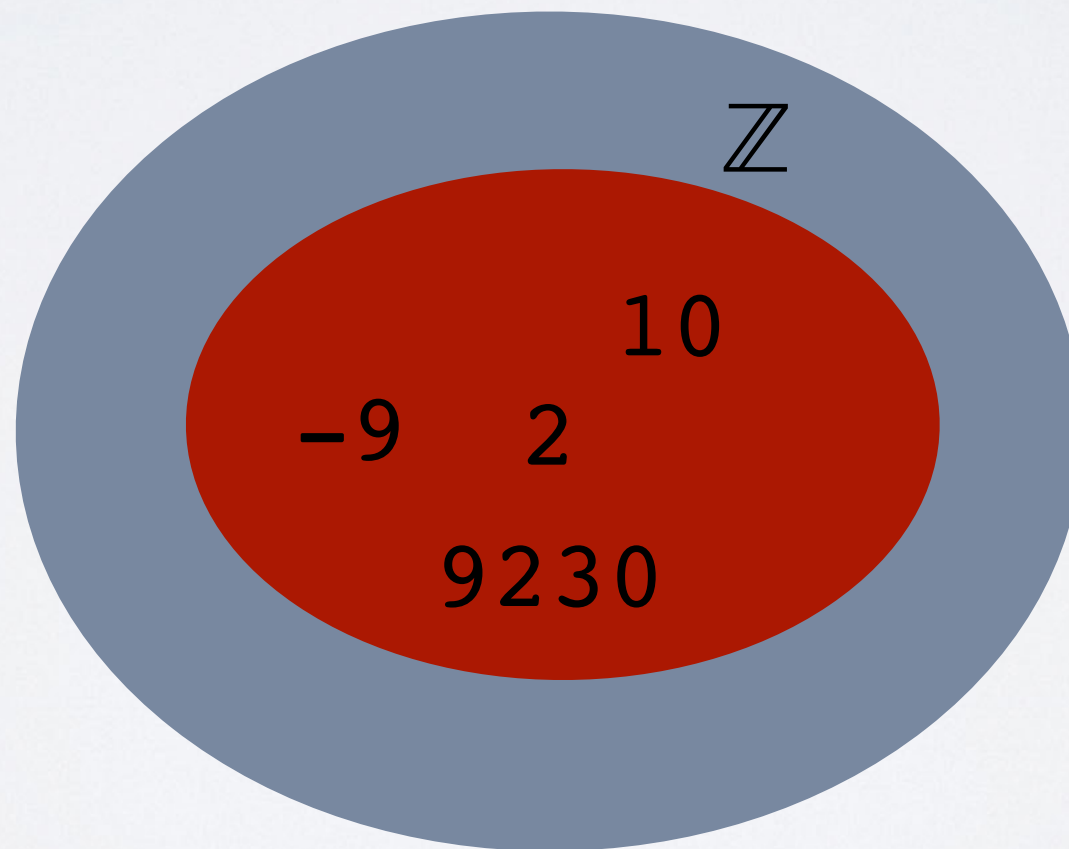
This is worst-case **$O(n)$**

Solution: Hashtables

- ▶ Dictionaries serve as the workhorse behind a lot of other algorithms
- ▶ They need to be efficient
- ▶ Array's are like dictionaries
 - ▶ keys are limited to $\{0, 1, \dots, (n - 1)\}$
- ▶ Can we generalise arrays for natural keys outside of range
 - ▶ What about keys that are not numbers?

Solution: Hashtables

- ▶ Core insight: Even though we can have a large universe of keys, we will only deal with a finite subset of them in practice!



Solution: Hashtables

- ▶ Core insight: We can “shrink” a larger universe of keys to a smaller one
- ▶ Core insight: We can project keys from a larger universe of keys to the set of indices for an array!
- ▶ If we have an array index, then we can insert, search, and delete data using random access

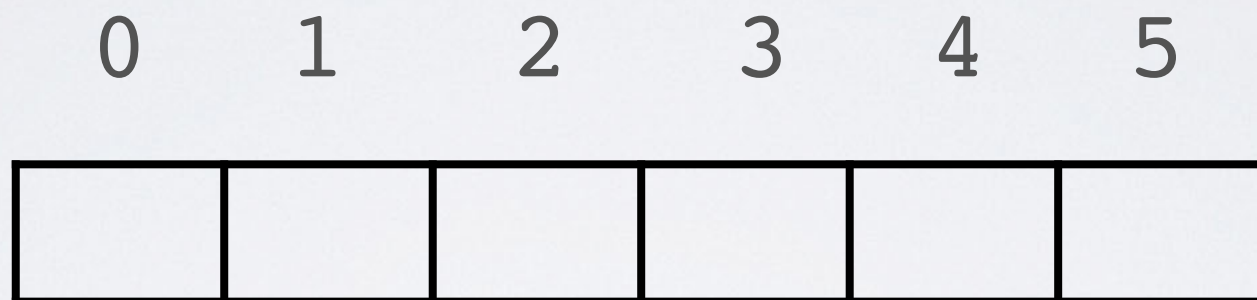
Random access on array indices is $O(1)$!

Hashtables

- ▶ Consider that we have a universe of keys \mathbf{X}
- ▶ Hashtables:
 - ▶ An array (may or may not be static)
 - ▶ Each space in the array is called a bucket
 - ▶ A hash function, $\mathbf{h: X \longrightarrow Y}$
 - ▶ Here $\mathbf{Y = \{0, 1, \dots n-1\}}$ where \mathbf{n} is the length of the array
 - ▶ Applying a hash function is called hashing
 - ▶ The result of the application of a hash function is called the digest or hash value
 - ▶ In practice, hash functions used in **hash tables** also take the length of the array
 - ▶ Other uses of hashing!

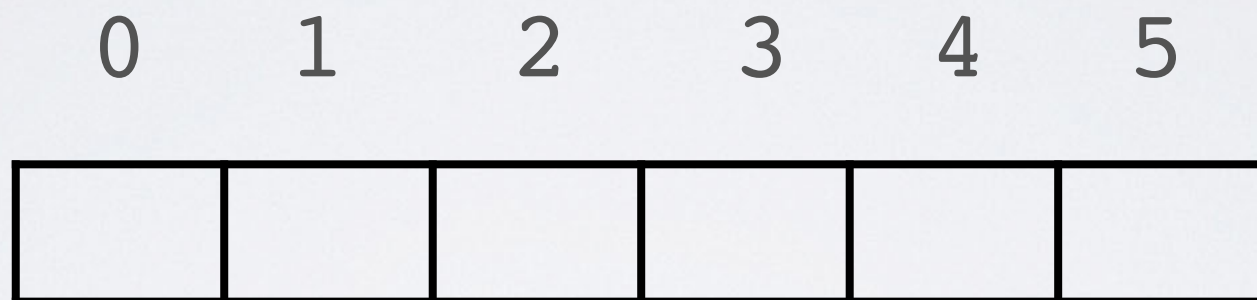
Hashtables

Suppose that we need
to store the pair (k, v)



Hashtables

Suppose that we need
to store the pair (k, v)



$$k \longrightarrow h(x) \longrightarrow 2$$

Hashtables

Suppose that we need
to store the pair (k, v)

0	1	2	3	4	5
		(k, v)			

$$k \longrightarrow h(x) \longrightarrow 2$$

```
function insert(table, h, k, v)
    index = h(k, length(table))
    arr[index] = (k, v)
```

$O(1)$

```
function search(table, h, k)
    index = h(k, length(table))
    // arr is initially filled with NIL
    return arr[index]
```

$O(1)$

```
function delete(table, h, k)
    index = h(k, length(table))
    // arr is initially filled with NIL
    arr[index] = NIL
```

$O(1)$

Assuming that **h** is $O(1)$



Hashtables: Trouble in Paradise



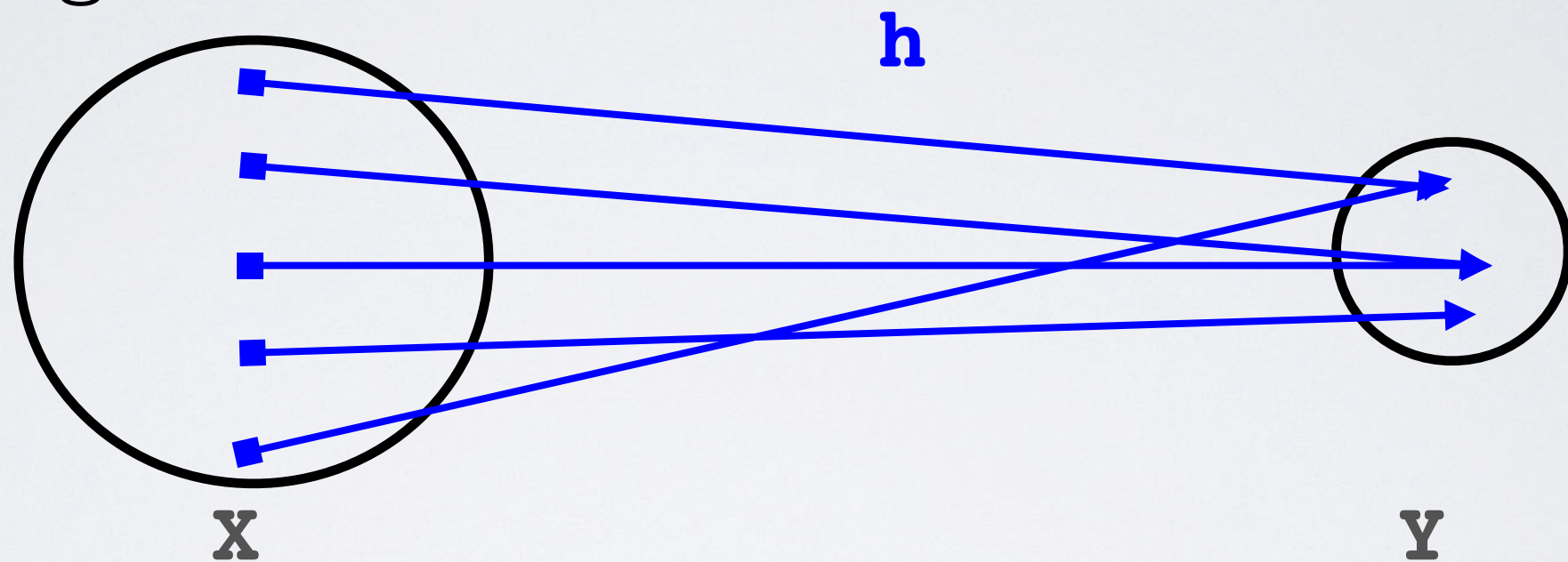
- ▶ Hashtables are wonderful, but there are challenges
- ▶ $|X| > |Y|$
 - ▶ When shrinking universe of keys, will have multiple keys hashing to the same hash value. (Pigeon-hole principle)
 - ▶ This is called a collision
- ▶ The concept of a hashtable is easy!
- ▶ Handling these collisions is the hard part!

Hashtables: Fixing the problems

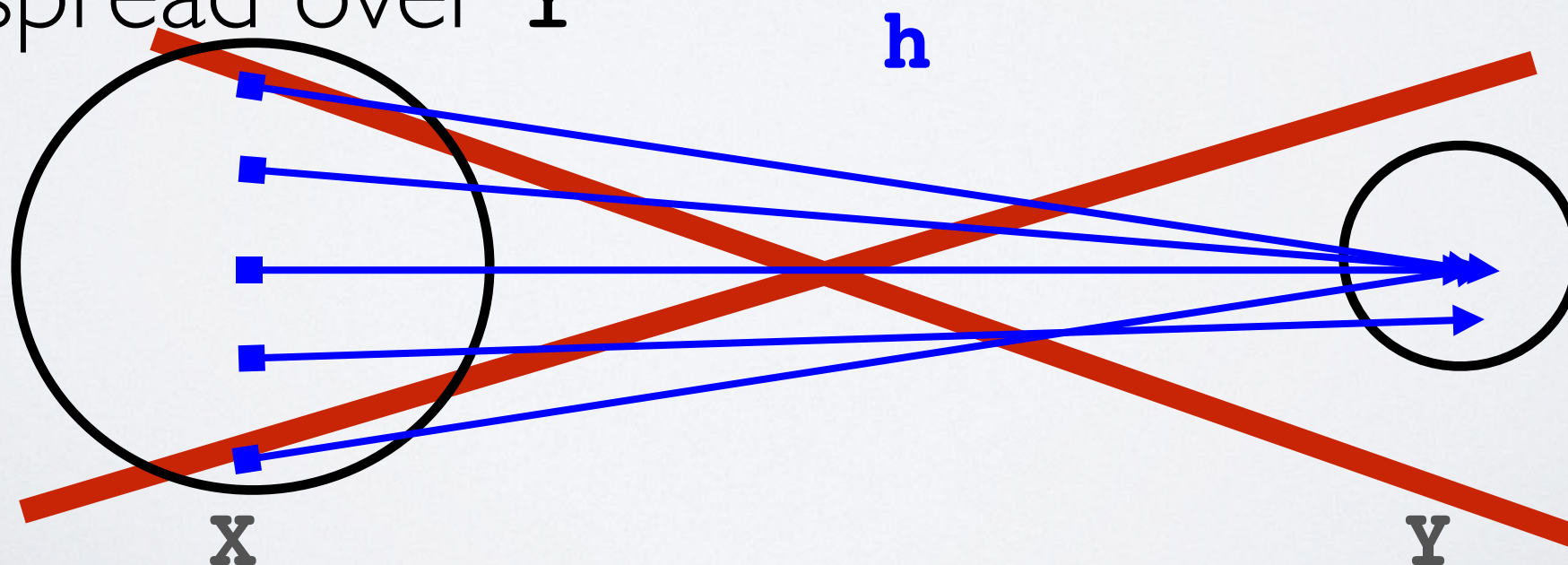
- ▶ We have control over **h**
 - ▶ We can design a good **h**
 - ▶ What are the properties of a good hash function?

Properties of a Good Hash Function

- ▶ Shrinking



- ▶ Well-spread over Y



Properties of a Good Hash Function

- ▶ A good hash function should consider all of the data available in the key
 - ▶ Consider the strings “abc” and “abd”.
 - ▶ One character different, but this would cause them to map to different digests
- ▶ A good hash function should depend solely on the the input alone.
 - ▶ External data might inadvertently increase collisions

Properties of a Good Hash Function

1. **Well-spread**: a good hash function is uniformly spread across the set of hash digests
2. **Deterministic**: considers only its input
3. **Comprehensive**: considers all of its input

Most hash functions we would use in this course will be of the form

$$(af(x) + b) \bmod n$$

where

n is the length of the array

$a = 0, b = 0$ for the most part

f is a function that interprets x as an integer

Interpreting as an Integer

- ▶ We can also the case of non-integer keys
- ▶ To use the hash function on the last slide, we need to interpret or cast the input to an integer
- ▶ Suppose that we had the string “foobar”?
 - ▶ Operate on character byte representation to generate hash code, then use hash code to generate digest for the hashtable
 - ▶ In Python, we can do this using the (poorly named) *hash* function

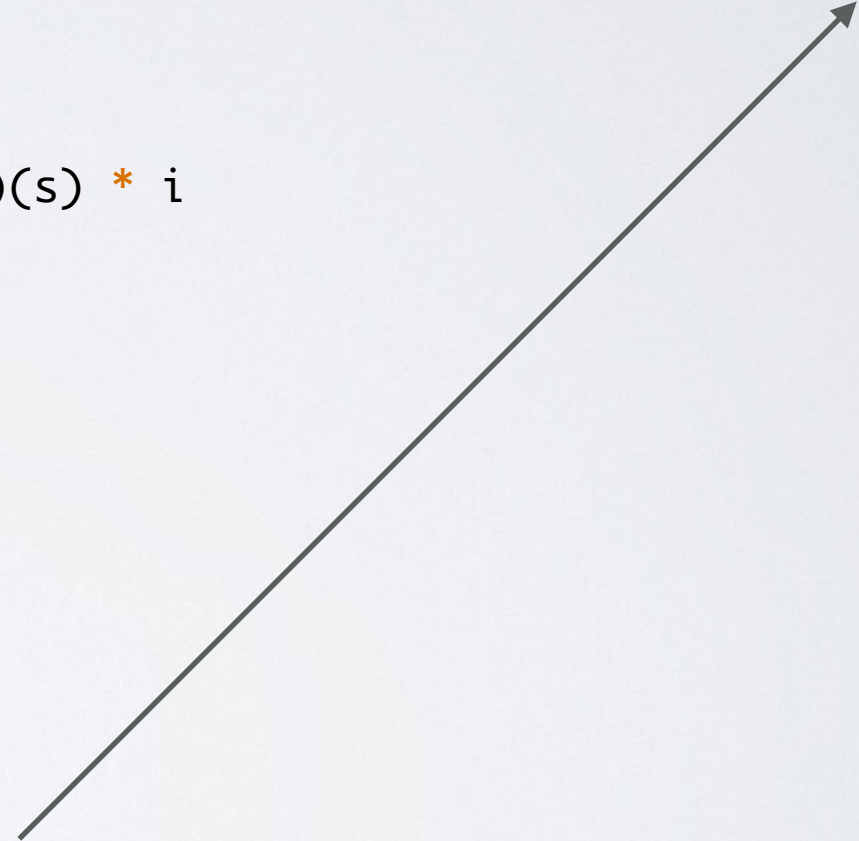
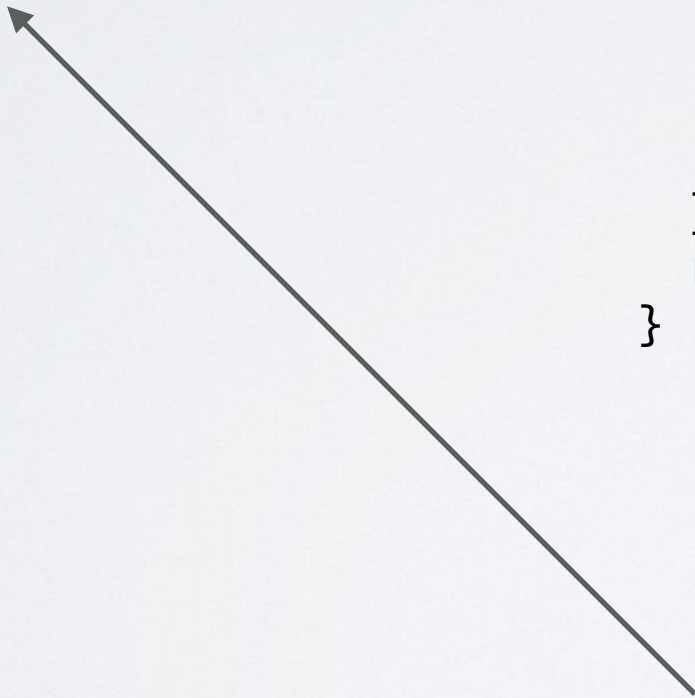
Code to do this C/C++



```
s = 'foobar'
i = hash(s)
print(i)
```

```
int string_hash(char* s)
{
    long code = 0
    long i = 1
    while(*s)
    {
        code += (long)(s) * i
        i += 1
        s += 1
    }
    return code
}
```

```
String s = "foobar";
Integer i = s.hashCode()
System.out.println(i)
```



Integer computed
using underlying bytes

Collision Resolution

- ▶ A good hash function would minimise the probability of collisions
 - ▶ But these theoretical guarantees are hard to come by and prove in practice
- ▶ Still need to handle collisions when they (inevitably) happen

Collision Resolution Strategies

- ▶ Chaining
- ▶ Linear Probing
- ▶ Quadrating Probing
- ▶ Double Hashing
- ▶ Hopscotch Hashing
- ▶ Cuckoo Hashing

Collision Resolution Strategies

- ▶ Chaining
- ▶ Linear Probing
- ▶ Quadrating Probing
- ▶ Double Hashing
- ▶ Hopscotch Hashing
- ▶ Cuckoo Hashing



Open Addressing

Collision Resolution Strategies

- ▶ **Chaining**
- ▶ **Linear Probing**
- ▶ **Quadrating Probing**
- ▶ **Double Hashing**
- ▶ Hopscotch Hashing
- ▶ Cuckoo Hashing



Open Addressing

Chaining

- ▶ Core idea: Allow each **bucket** to hold more than one item
- ▶ Uses additional data structure to store multiple key/value pairs. Notice each value still maps to **single** value
- ▶ Hashtable becomes array of this data structure
- ▶ Data Structures used:
 - ▶ Linked Lists
 - ▶ Binary Search Trees
 - ▶ Dynamic Arrays
 - ▶ Another Hashtable

Chaining

- ▶ Core idea: Allow each **bucket** to hold more than one item
- ▶ Uses additional data structure to store multiple key/value pairs. Notice each value still maps to **single** value
- ▶ Hashtable becomes array of this data structure
- ▶ Data Structures used:
 - ▶ Linked Lists
 - ▶ Binary Search Trees
 - ▶ **Dynamic Arrays**
 - ▶ Another Hashtable

Will use Dynamic Arrays
But result generalise!

Hashtables: Trouble in Paradise

- ▶ Hashtables are wonderful, but there are challenges
- ▶ $|X| > |Y|$
 - ▶ When shrinking universe of keys, will have multiple keys hashing to the same hash value. (Pigeon-hole principle)
 - ▶ This is called a collision
- ▶ The concept of a hashtable is easy!
- ▶ Handling these collisions is the hard part!

Chaining

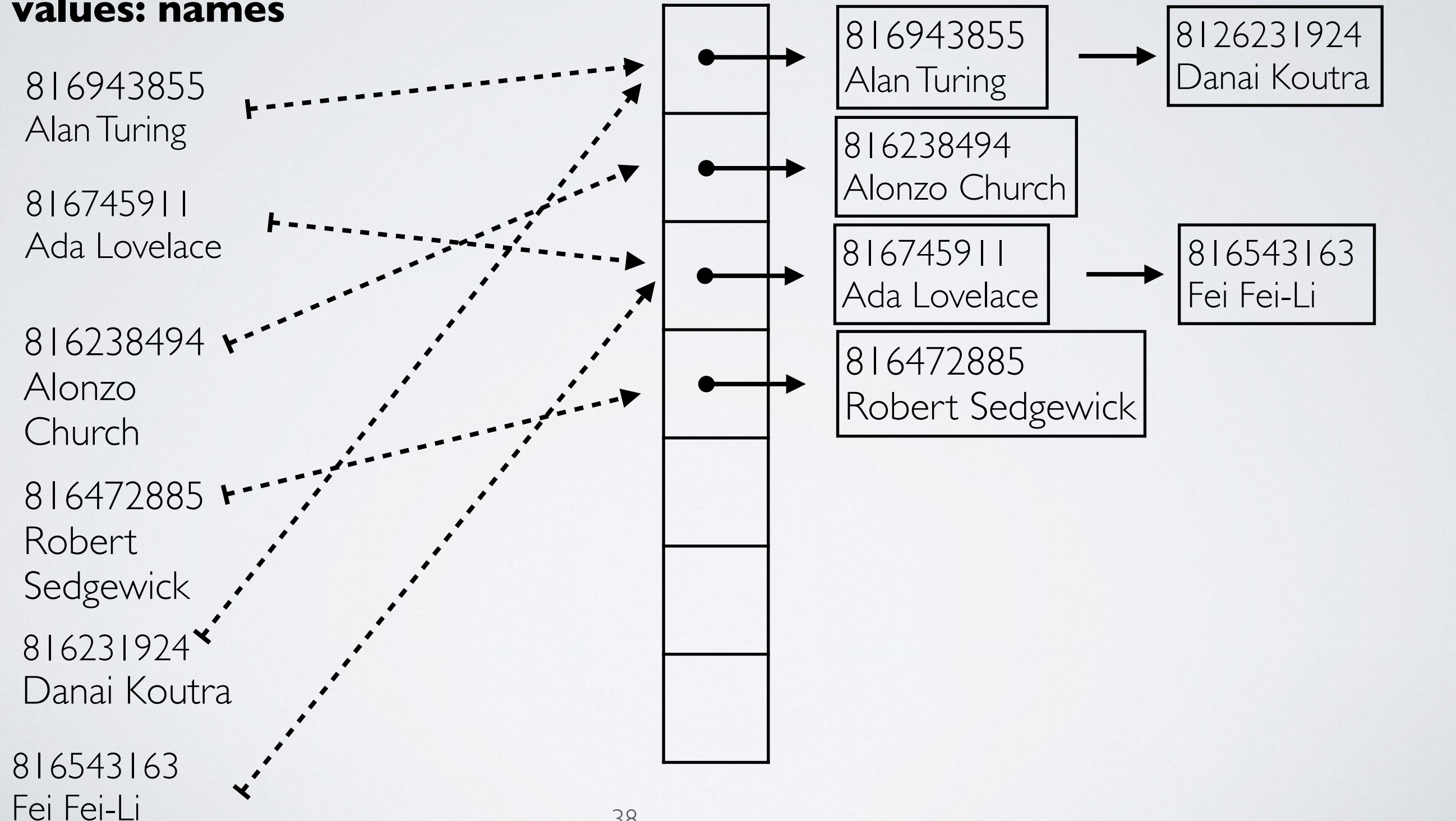
- ▶ Let's consider a practical example
 - ▶ Assume student IDs are integers (note that is not the best type for them!)
 - ▶ Why?
 - ▶ Assume a hashtable has 7 buckets
 - ▶ $h(x) = x \bmod 7$
 - ▶ Will guarantee that every student ID maps to a single bucket
 - ▶ Why?

Chaining

- ▶ Let's consider a practical example
 - ▶ Assume student IDs are integers (note that is not the best type for them!)
 - ▶ Student IDs can start with 0 (and that is semantically relevant)
 - ▶ We don't use arithmetic operations on student IDs!
 - ▶ We will consider how to handle strings later (focus on resolution for now)
 - ▶ Assume a hashtable has 7 buckets
 - ▶ $h(x) = x \bmod 7$
 - ▶ Will guarantee that every student ID maps to a single bucket
 - ▶ Taking modulo of an integer with 7 finds the remainder after dividing by 7.
 - ▶ This is bounded between 0 (inclusive) and 6 (inclusive)

Chaining - Insertion

keys: student IDs $h(\text{key}) = \text{key} \% 7$
values: names

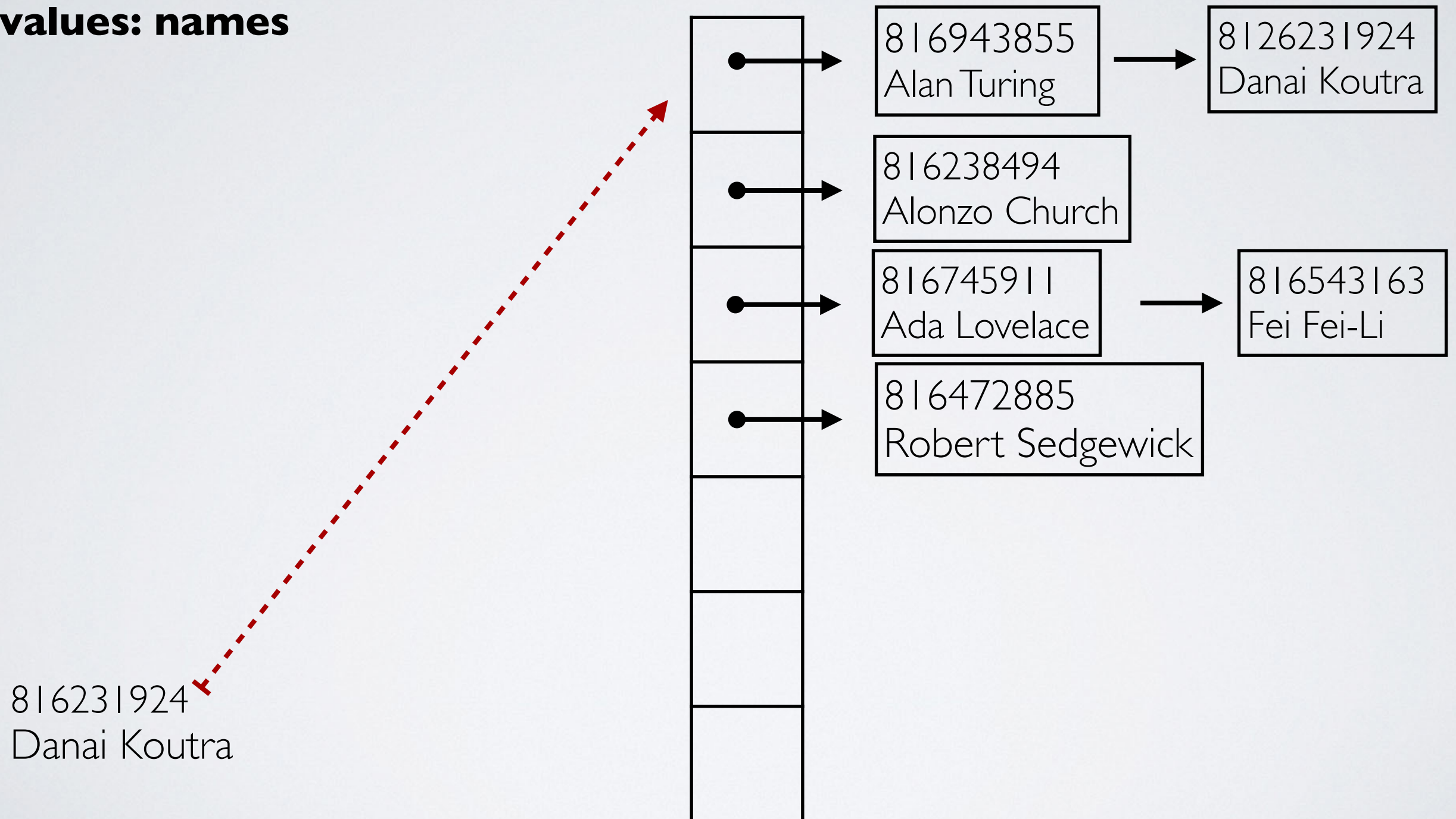


Chaining - Searching

keys: student IDs
values: names

$$h(\text{key}) = \text{key} \% 7$$

**Array of buckets w/
key/value pairs**

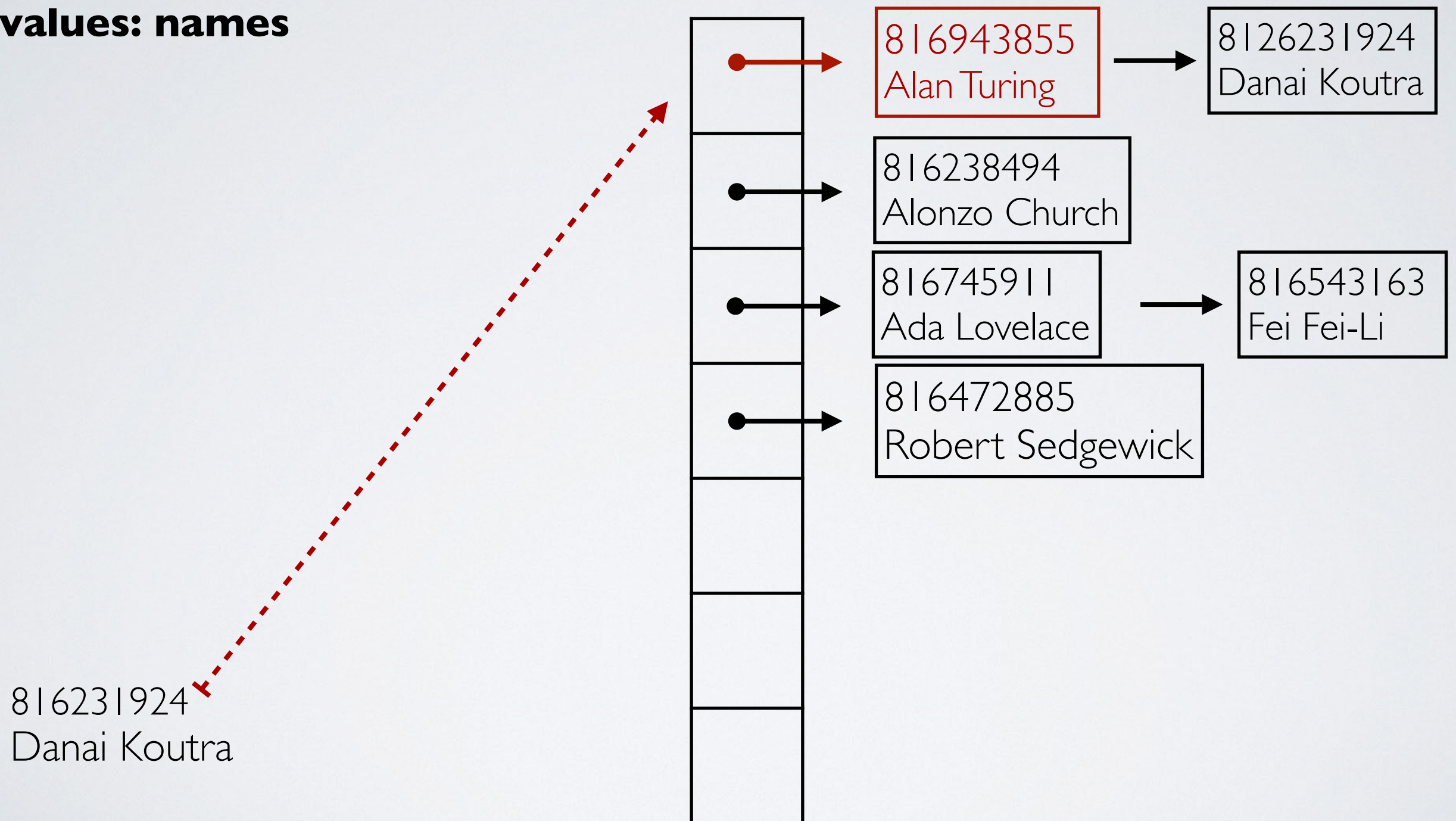


Chaining - Searching

keys: student IDs
values: names

$$h(\text{key}) = \text{key} \% 7$$

**Array of buckets w/
key/value pairs**

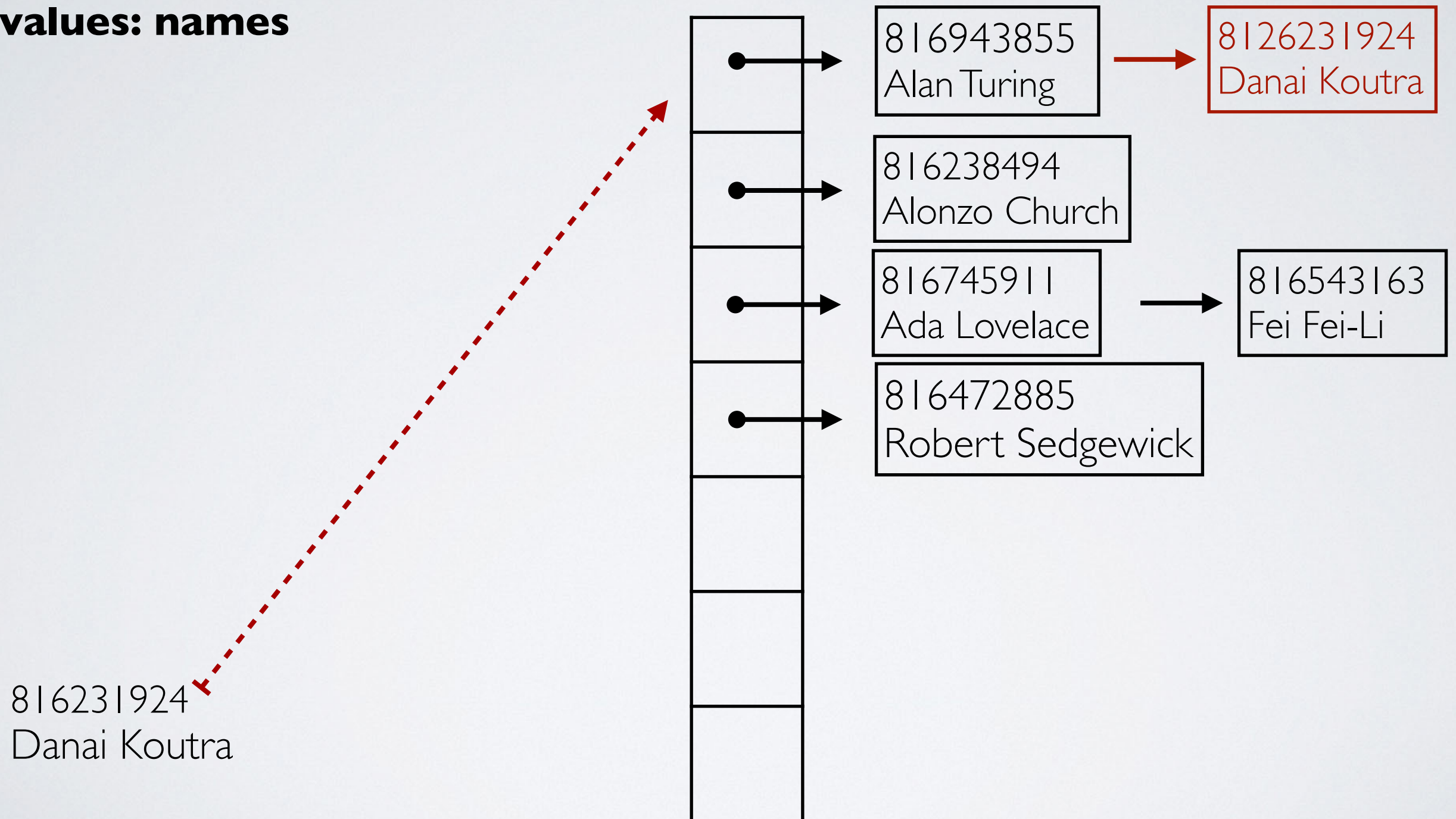


Chaining - Searching

keys: student IDs
values: names

$$h(\text{key}) = \text{key} \% 7$$

**Array of buckets w/
key/value pairs**



Chaining - Deletion

- ▶ Deletion just involves locating the key/value pair in the appropriate location and removing it

Chaining - Pseudocode

Pseudocode has redundancy (Try to eliminate it)

```
function insert(table, h, k, v)
    index = h(k, length(table))
    // if the bucket is empty
    if length(table[index]) == 0:
        table[index].append((k, v))
    else:
        inserted = False
        for j, (k_prime, v_prime) in table[index]:
            // update value if key exists
            if k == k_prime:
                table[index][j] = (k, v)
                inserted = True
        // insert if it doesn't
        if inserted == False:
            table[index].append((k, v))
```

Chaining - Pseudocode

```
function search(table, h, k)
    index = h(k, length(table))
    // if the bucket is empty
    if length(table[index]) == 0:
        return NIL
    else:
        // search bucket for (key, value) pair
        for (k_prime, v_prime) in table[index]:
            if k_prime == k:
                return v_prime
        // if didn't return, key not found
        return NIL
```


Chaining - Pseudocode

```
function delete(table, h, k)
    index = h(k, length(table))
    if length(arr[index]) > 0:
        for i, (kp, vp) in enumerate(table[index]):
            if k == kp:
                delete(table[index], i)
```

```

class ChainingHashTable:
    def __init__(self, n, h):
        # create blank list with n elements
        self.table = [[]] * n
        self.n = n

    def insert(self, key, value):
        i = hash(key)
        i = i % self.n
        if self.table[i] is None:
            self.table[i] = [(key, value)]
        else:
            inserted = False
            for j, (k_prime, v_prime) in enumerate(self.table[i]):
                if key == k_prime:
                    self.table[i][j] = value
                    inserted = True
            if inserted == False:
                self.table[i].append((key, value))

    def search(self, key, value):
        i = hash(key)
        i = i % self.n
        if self.table[i] is None:
            return None
        for (k_prime, v_prime) in self.table[i]:
            if key == k_prime:
                return value
        return None

    def delete(self, key):
        i = hash(key)
        i = i % self.n
        if self.table[i] is not None:
            for j, (k_prime, v_prime) in enumerate(self.table[i]):
                if key == k_prime:
                    # Python remove element from index j
                    # moves all other elements up accordingly
                    self.table[i].pop(j)

```

Chaining

- ▶ Advantages:
 - ▶ Very Easy to implement
 - ▶ Can have more entries than buckets without expanding the number of buckets
 - ▶ Deletion is trivial in Chaining. Not so much in open addressing
- ▶ Disadvantages
 - ▶ Large number of collisions lead to large buckets!
 - ▶ Has worst case $O(n)$
 - ▶ Amortized and average case is better
 - ▶ But we want **stronger** guarantees

Open Addressing

- ▶ In Open Addressing collision resolution schemes, each bucket contains only a single element
- ▶ In Open Addressing, we access key/value pairs directly by an index
- ▶ Underlying generalisation: if a bucket is filled, we compute new bucket indices in a **deterministic** way until we find a free space

Linear Probing

- ▶ Uses number of misses in computing the new space.
 - ▶ misses = # of buckets that we've seen already filled during our search for a free space
- ▶ If a bucket is filled, we move in increments of **1** until we find a free bucket
 - ▶ Consider (k, v) . Let $i = h(k)$. If we find a place after m misses, we put the data into index $(i + m) \bmod n$ where n is the number buckets in the hashtable
 - ▶ We use $\bmod n$ to ensure that we stay in the bounds of the table

Double Hashing

- ▶ Double Hashing is a generalisation of linear probing where the fixed increment is a function of the input
 - ▶ Primary hash function computes initial index, $h_1(x)$
 - ▶ Use secondary hash function $h_2(x)$ to compute increment
 - ▶ Consider (k, v) . Let $i = h_1(x)$, $j = h_2(x)$. If we find a place after m misses, we put the data into index $(i + m * j) \bmod n$ where n is the number buckets in the hashtable
- ▶ Think of linear probing as double hashing where $h_2(x)$ always returns a 1

Quadratic Probing

- ▶ Uses number of misses in computing the new space.
- ▶ increment is the square of the number of misses!
- ▶ Consider (k, v) . Let $i = h(k)$. If we find a place after m misses, we put the data into index $(i + m^2) \bmod n$ where n is the number buckets in the hashtable
- ▶ Will will cover Linear Probing pseudocode in lecture
- ▶ Quadratic Probing and Double Hashing in labs!
 - ▶ Make sure that you know the pseudocode for all, can implement all of them, and can trace through all collision resolution methods
- ▶ Unifying principle: move in increments

Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names

816943855
Alan Turing

816745911
Ada Lovelace

816238494
Alonzo
Church

816472885
Robert
Sedgewick

816231924
Danai Koutra

816943855 Alan Turing
816238494 Alonzo Church
816745911 Ada Lovelace
816472885 Robert Sedgewick

Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names

816943855
Alan Turing

816745911
Ada Lovelace

816238494
Alonzo Church

816472885
Robert Sedgewick

816231924
Danai Koutra

816943855 Alan Turing
816238494 Alonzo Church
816745911 Ada Lovelace
816472885 Robert Sedgewick

Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names

816943855
Alan Turing

816745911
Ada Lovelace

816238494
Alonzo Church

816472885
Robert Sedgewick

816231924
Danai Koutra

816943855 Alan Turing
816238494 Alonzo Church
816745911 Ada Lovelace
816472885 Robert Sedgewick

Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names

816943855
Alan Turing

816745911
Ada Lovelace

816238494
Alonzo
Church

816472885
Robert
Sedgewick

816231924
Danai Koutra

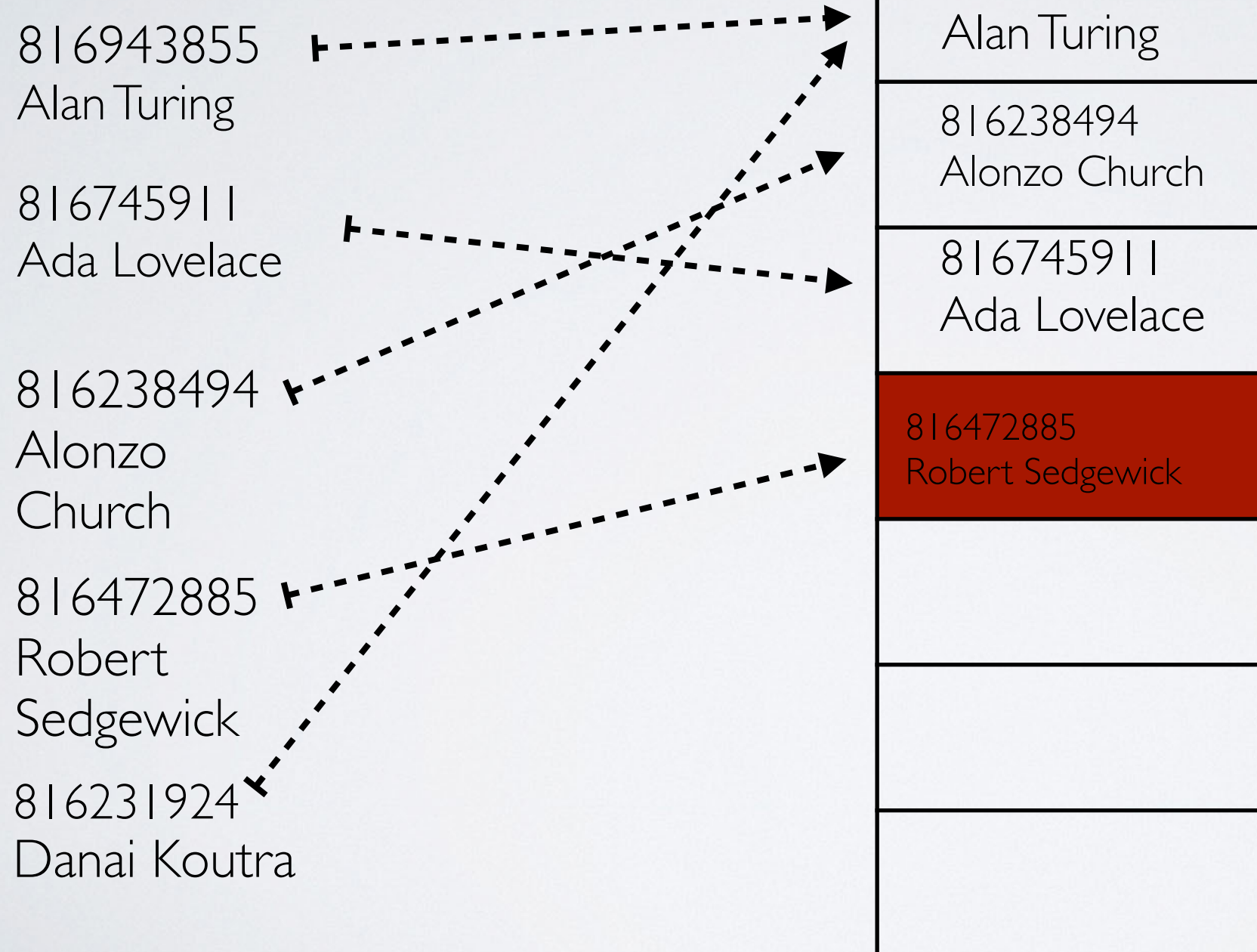
816943855 Alan Turing
816238494 Alonzo Church
816745911 Ada Lovelace
816472885 Robert Sedgewick

Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names



Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names

816943855
Alan Turing

816745911
Ada Lovelace

816238494
Alonzo Church

816472885
Robert Sedgewick

816231924
Danai Koutra

816943855 Alan Turing
816238494 Alonzo Church
816745911 Ada Lovelace
816472885 Robert Sedgewick

Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names

816943855
Alan Turing

816745911
Ada Lovelace

816238494
Alonzo
Church

816472885
Robert
Sedgewick

816231924
Danai Koutra

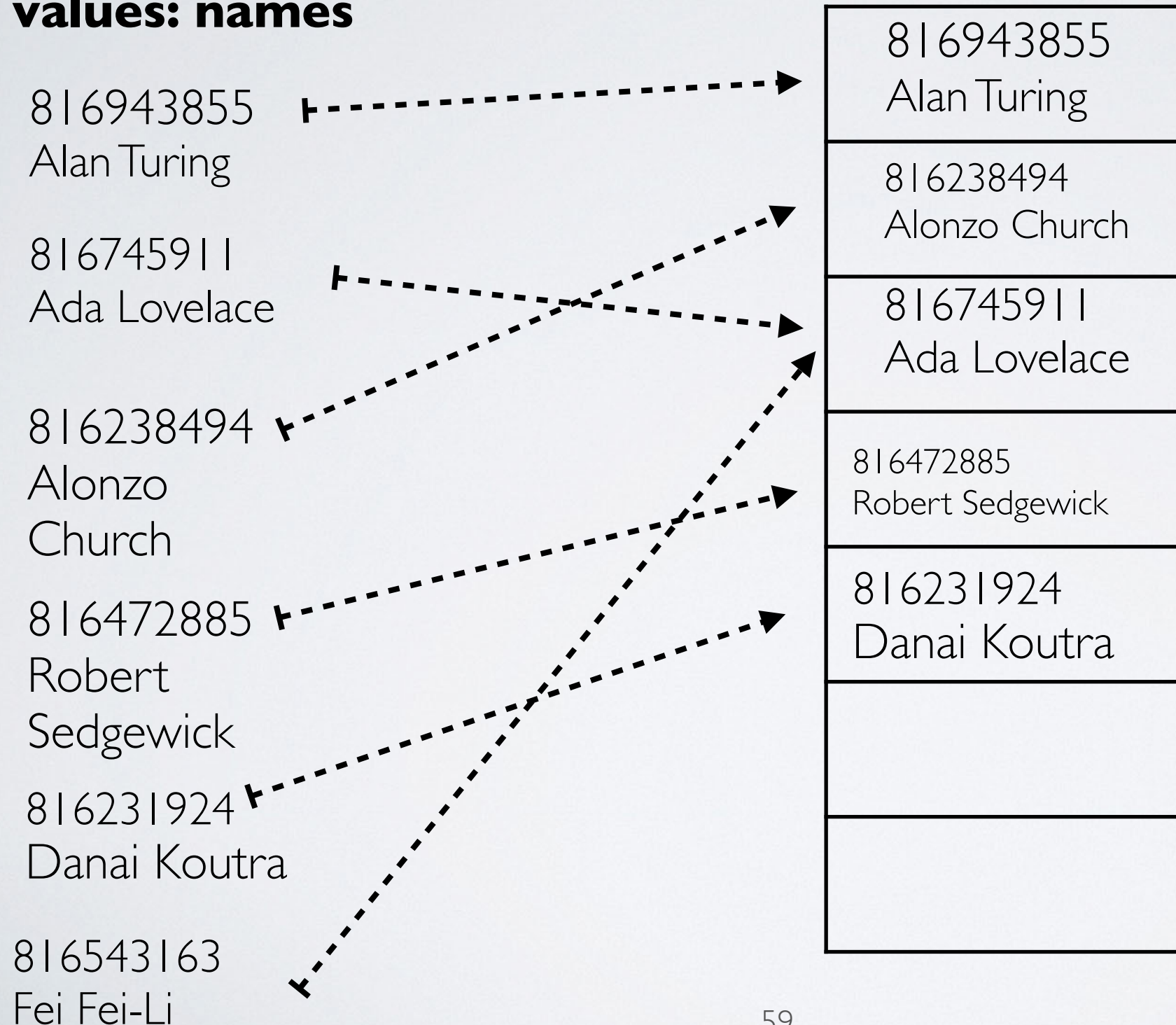
816943855 Alan Turing
816238494 Alonzo Church
816745911 Ada Lovelace
816472885 Robert Sedgewick
816231924 Danai Koutra

Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names

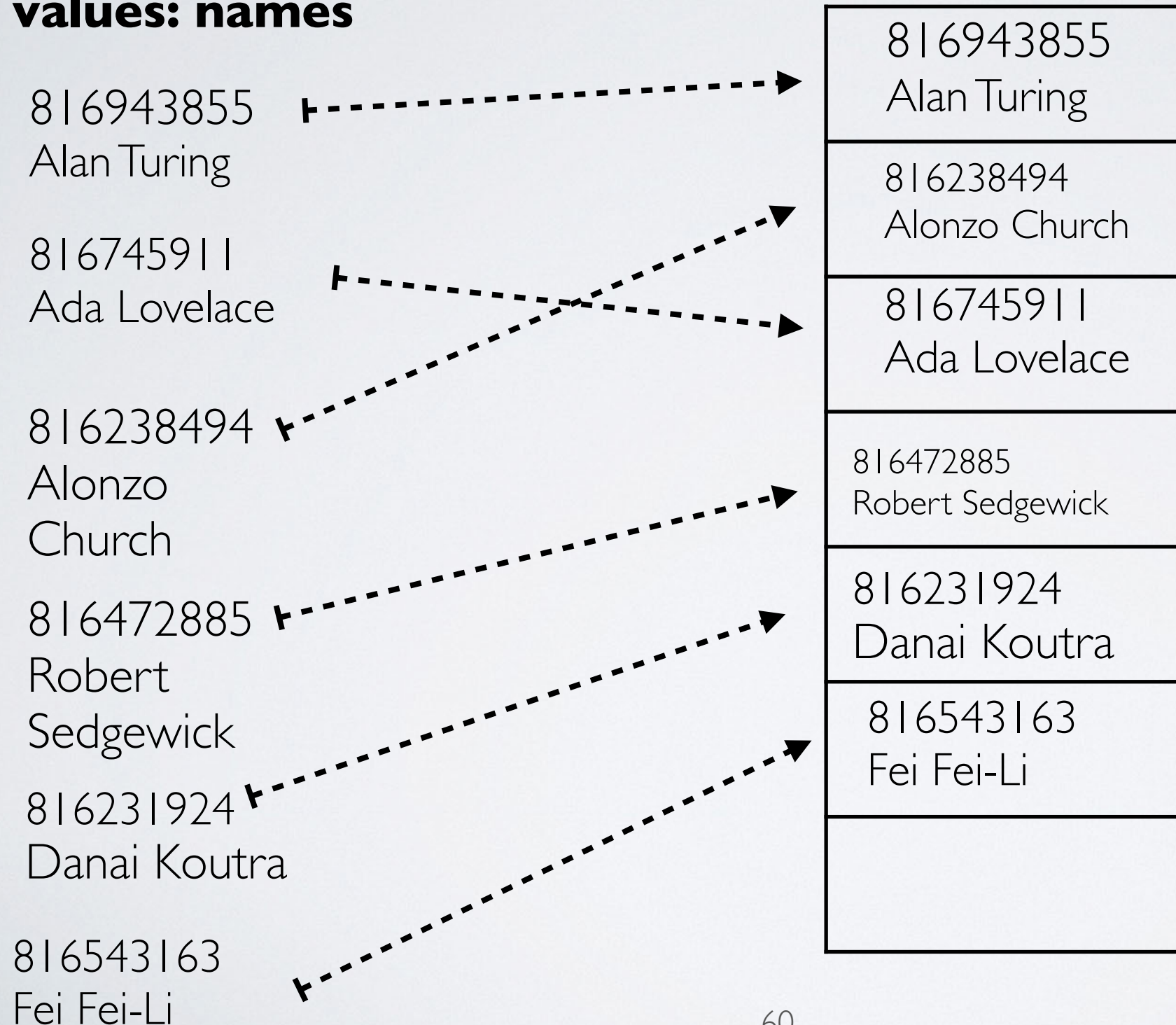


Linear Probing- Insertion

**Array of buckets w/
key/value pairs**

keys: student IDs $h(\text{key}) = \text{key} \% 7$

values: names



Linear Probing - Pseudocode

```
function insert(table, h, k, v)
    index = h(k, length(table))
    // if the bucket is empty
    if arr[index] is NIL:
        table[index] = (k, v)
    else:
        n = length(table)
        misses = 1
        new_index = (index + misses) mod n
        while table[new_index] is not NIL:
            misses += 1
            new_index = (index + misses) mod n
        table[new_index] = (k, v)
```

Open Addressing Searching

- ▶ Recall that we insert key/value pair into the first free bucket
- ▶ Hence if are searching for a key....
- ▶ And come across an empty bucket....
- ▶ This means that

Open Addressing Searching

- ▶ Recall that we insert key/value pair into the first free bucket
- ▶ Hence if are searching for a key....
- ▶ And come across an empty bucket....
- ▶ This means that
- ▶ Key is not in the hashtable

Linear Probing - Pseudocode

```
function search(table, h, k)
    index = h(k, length(table))
    misses = 0
    n = length(table)
    new_index = (index + misses) mod n
    while (table[index] is not NIL):
        key_prime, value_prime = table[index]
        if key_prime == k:
            return value_prime
        misses += 1
        new_index = (index + misses) mod n
    return NIL
```

Open Addressing Deletion

- ▶ Deletion is more complicated when using open addressing
- ▶ If searching assumes that seeing an empty space means the key is not present
- ▶ Simply setting the bucket to NIL will not work!
- ▶ We need to move content of other buckets backwards

Linear Probing - Pseudocode

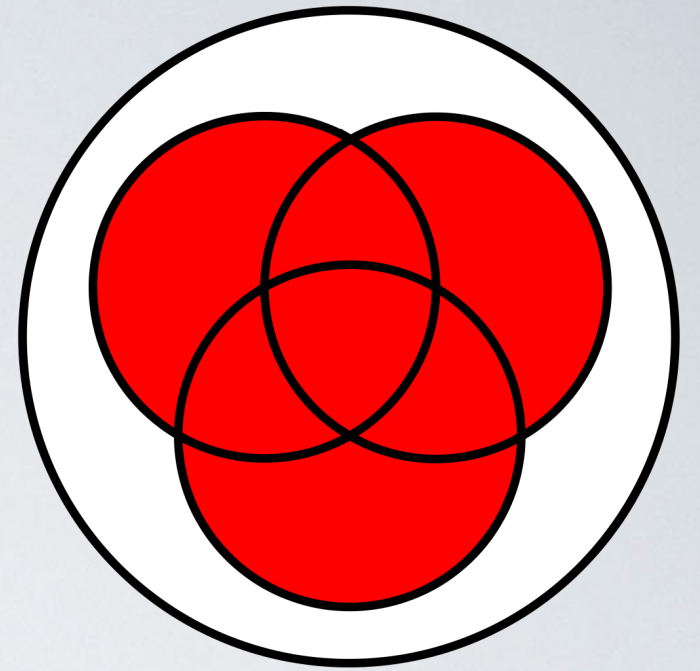
```
function delete(table, h, k)
  index = h(k, length(table))
  misses = 0
  n = length(table)
  new_index = (index + misses) mod n
  while (table[index] is not NIL):
    key_prime, value_prime = table[index]
    if key_prime == k:
      table[new_index] = NIL
      prev_index = new_index
      curr = misses + 1
      new_index = (index + curr) mod n
      while (table[new_index] is not NIL):
        table[prev_index] = table[new_index]
        prev_index = new_index
        curr += 1
        new_index = (index + curr) mod n
      break // exit of the main while loop
    misses += 1
  new_index = (index + misses) mod n
```

Linear Probing Problems

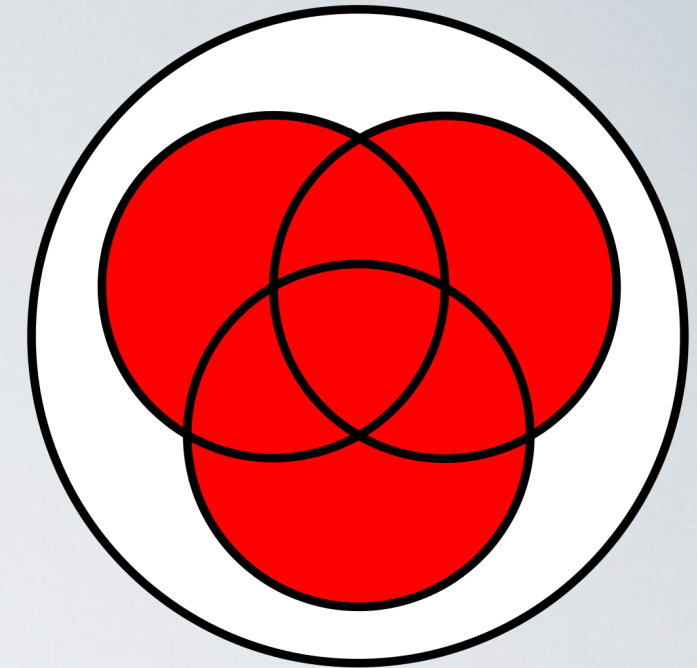
- ▶ Finding a free space by incrementing by **1** is not very efficient
- ▶ Using quadratic probing or double hashing are better as you hit fewer filled buckets on insertion
- ▶ Deletion is slightly more cumbersome though

Sets

- ▶ Collection of elements that are
 - ▶ distinct
 - ▶ unordered (unlike lists or arrays)



Set ADT



- ▶ **add**(object):
 - ▶ adds object to set if not there
- ▶ **remove**(object):
 - ▶ removes object from set if there
- ▶ boolean **contains**(object):
 - ▶ checks if object is in set

Set Use Cases

- ▶ Any application that needs to check if something is unique or is in a particular collection of items
 - ▶ Blacklists or whitelists on proxy servers
 - ▶ Privileged users in a system
 - ▶ Nodes already visited in a graph or network
 - ▶ Anti-viruses (Assignment #1)
 - ▶ etc

Sets as special cases of Dictionaries

- ▶ If you have a type implementing the Dictionary ADT, you can use that to implement Set ADT
- ▶ **add**(object) = **insert**(object, object)
- ▶ **remove**(object) = **delete**(object)
- ▶ **contains**(object) = **search**(object) **!=** NIL

References

- ▶ Some examples taken from Brown's CS6 I
- ▶ Template adapted from CS6 I
- ▶ Wikipedia
- ▶ Sedgewick
- ▶ Kalicharan