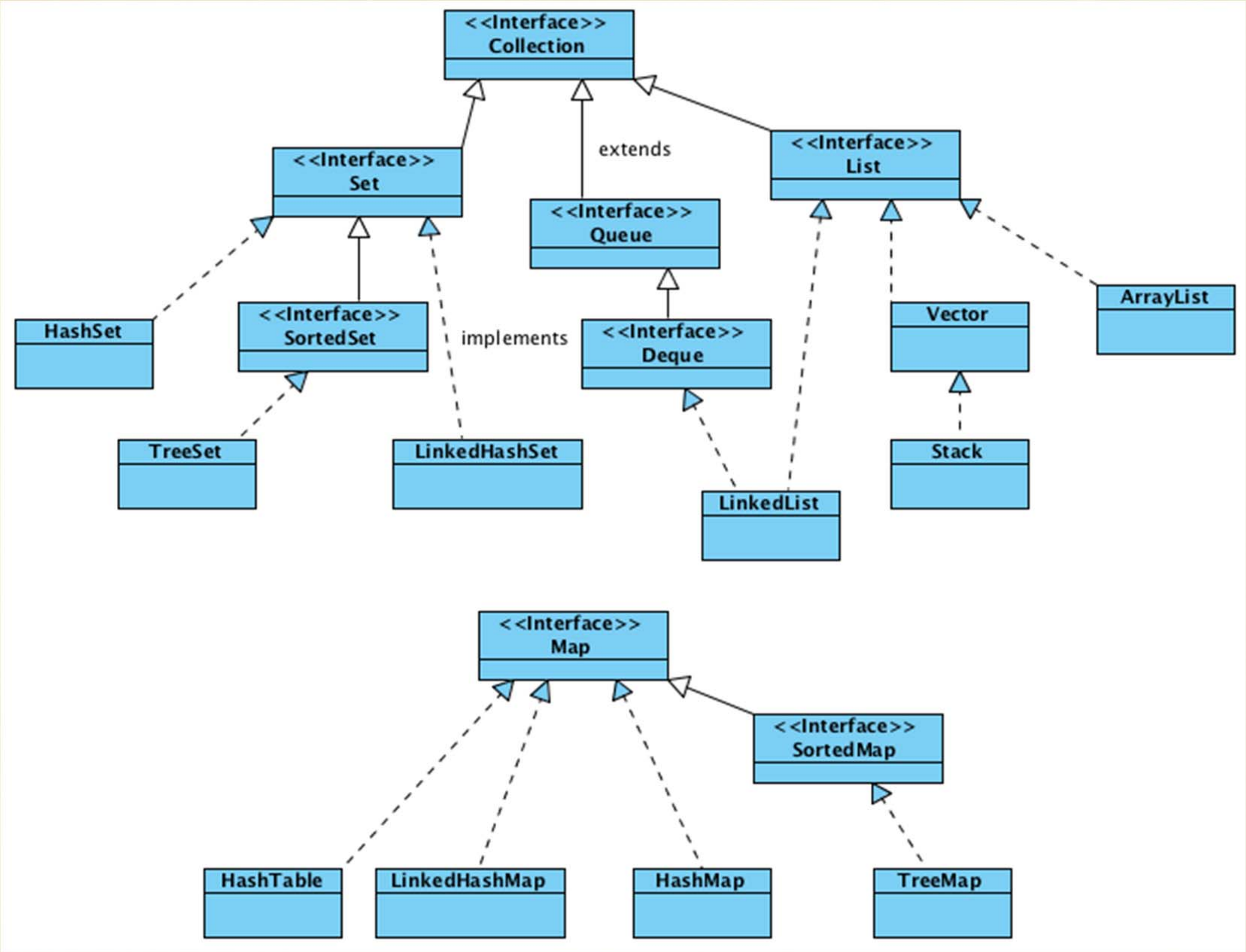
Data Structures and Object-Oriented Design VIII

Spring 2014

Carola Wenk

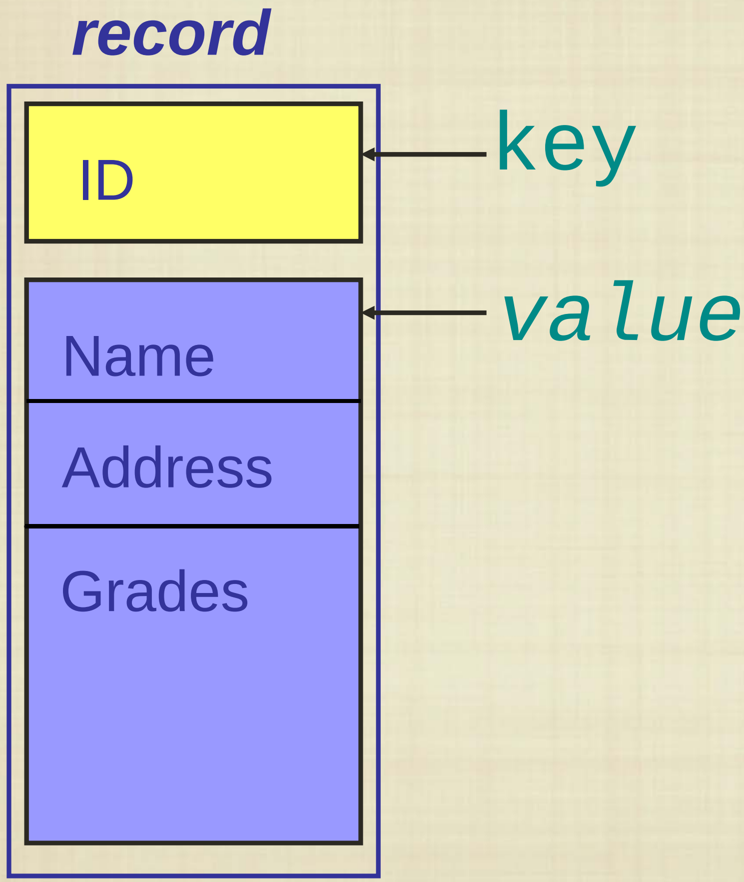
Collections and Maps

- The `Collection` interface is for storage and access, while a `Map` interface is geared towards associating keys with objects.



Student database problem

Tulane's student database D stores *n* ***records***:



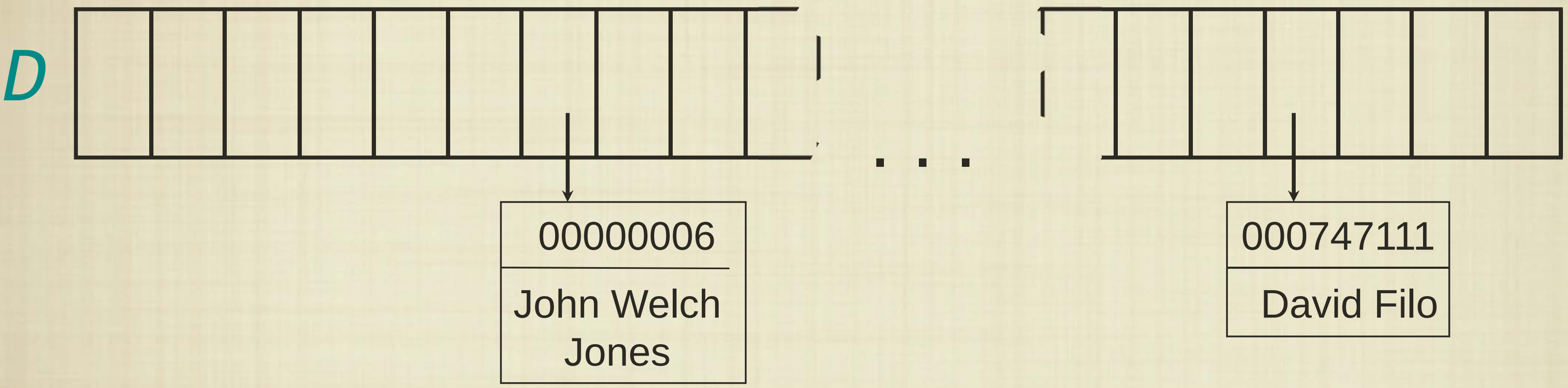
Operations on D:

- `D.put(key,value)` — "add"
- `D.get(key)` — "find"
- `D.remove(key)`

How should the data structure D be organized?

Direct-Access Table (array)

- Suppose every key is a different number: $K \subseteq \{0, 1, \ldots, m-1\}$
- Set up an array $D[0 \,..\, m-1]$ such that D[*key*] = *value* for every record, and D[*key*]=*null* for keys without records.



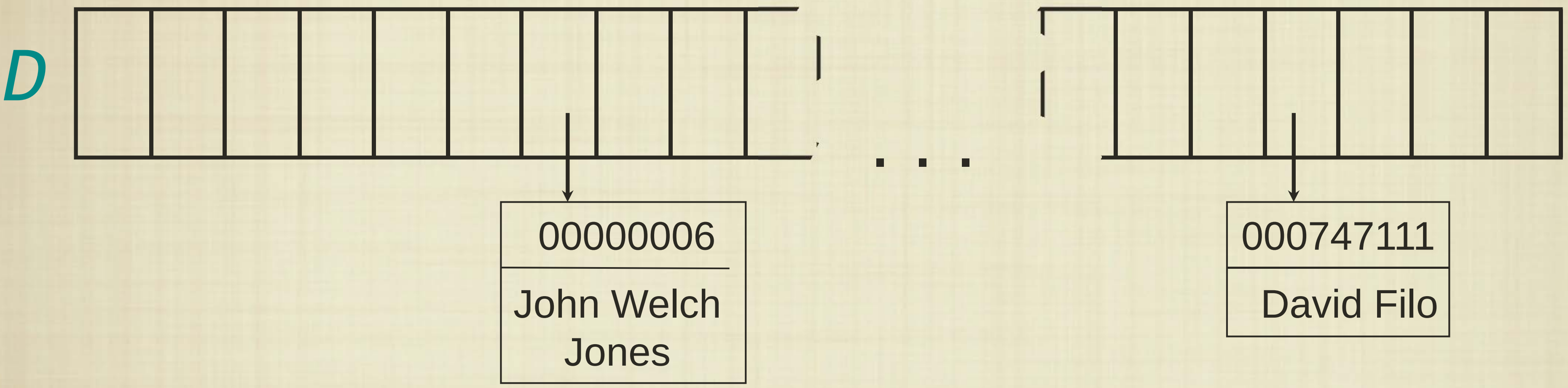
Direct-Access Table (array)

```
class DirectAccessTable{
   MyObject[] dataTable = null;

   DirectAccessTable(int n){
      dataTable = new MyObject[n];
      for (int i = 0; i < n; i++)
         dataTable[i] = null;
   }

   void add(MyObject x){
      dataTable[x.key] = x;
   }

   boolean find(int key){
      if (dataTable[key] != null)
         return true;
      else
         return false;
   }
}
```

We can use the key itself to index into the data being stored.

Direct-Access Table (array)

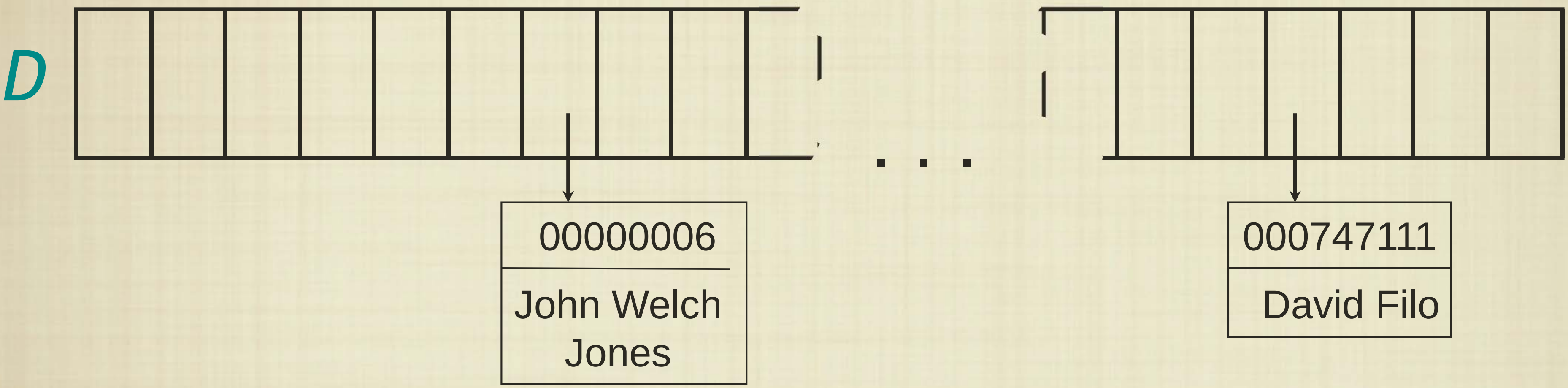
- Suppose every key is a different number: $K \subseteq \{0, 1, \ldots, m-1\}$
- Set up an array $D[0\ ..\ m-1]$ such that `D`[*key*] = *value* for every record, and `D`[*key*]=*null* for keys without records.



`add`, `find`, `remove` take $O(1)$ time.

Direct-Access Table (array)

- Suppose every key is a different number: $K \subseteq \{0, 1, \ldots, m-1\}$
- Set up an array $D[0\,..\,m-1]$ such that $D[key] = value$ for every record, and $D[key]=null$ for keys without records.

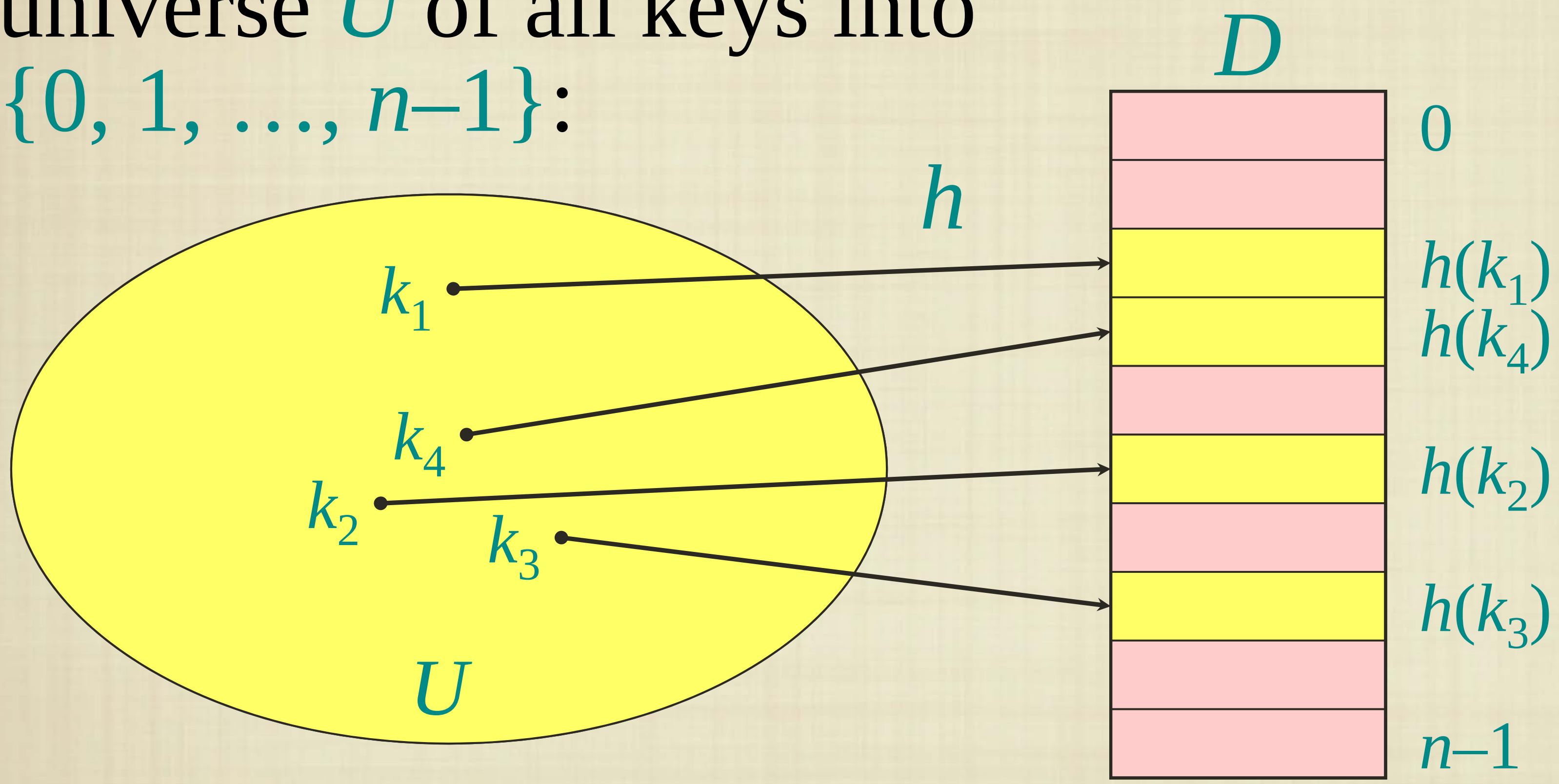


Problem: The range of keys can be large:

- 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
- Character strings (even larger!).

Hash functions

Solution: Use a ***hash function*** h to map the universe U of all keys into $\{0, 1, \ldots, n\text{–}1\}$:



As each key is inserted, h maps it to a slot of D.

Hash functions: Examples

Can be any number; preferably a prime number.

- If key is a number:
  $h_1(key) = key \% p$, for example key % 13
- If key is a string:
  $h_2(c_{n-1}\ldots c_1 c_0) = (c_0*31^{n-1} + c_1*31^{n-2} + \ldots + c_{n-1}) \% p$
- Java classes have a `hashCode()` method

(most of which do not have meaningful implementations. The String class has the above implementation.)

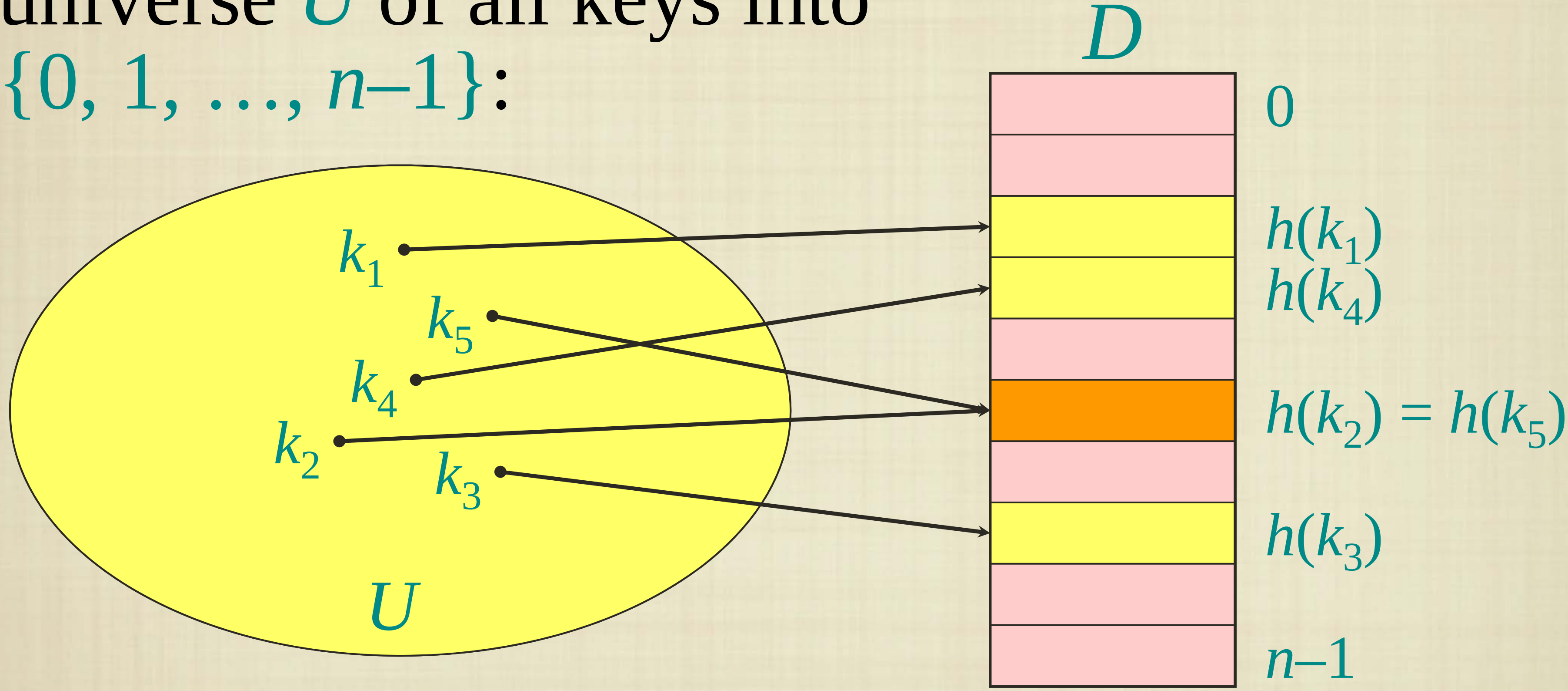
A Hash Table for Strings

```
class StringHashTable {
    String[] dataTable = null;

    StringHashTable(int n) {
        dataTable = new String[n];
        for (int i = 0; i < n; i++)
            dataTable[i] = null;
    }

    private int hashCode(String S) {
        return Math.abs(S.hashCode())%dataTable.length;
    }

    public void add(String S) {
        dataTable[hashCode(S)] = S;
    }

    public boolean find(String S) {
        if (dataTable[hashCode(S)] != null)
            return true;
        else
            return false;
    }
}
```

Assumes a perfect hash function.

Hash functions

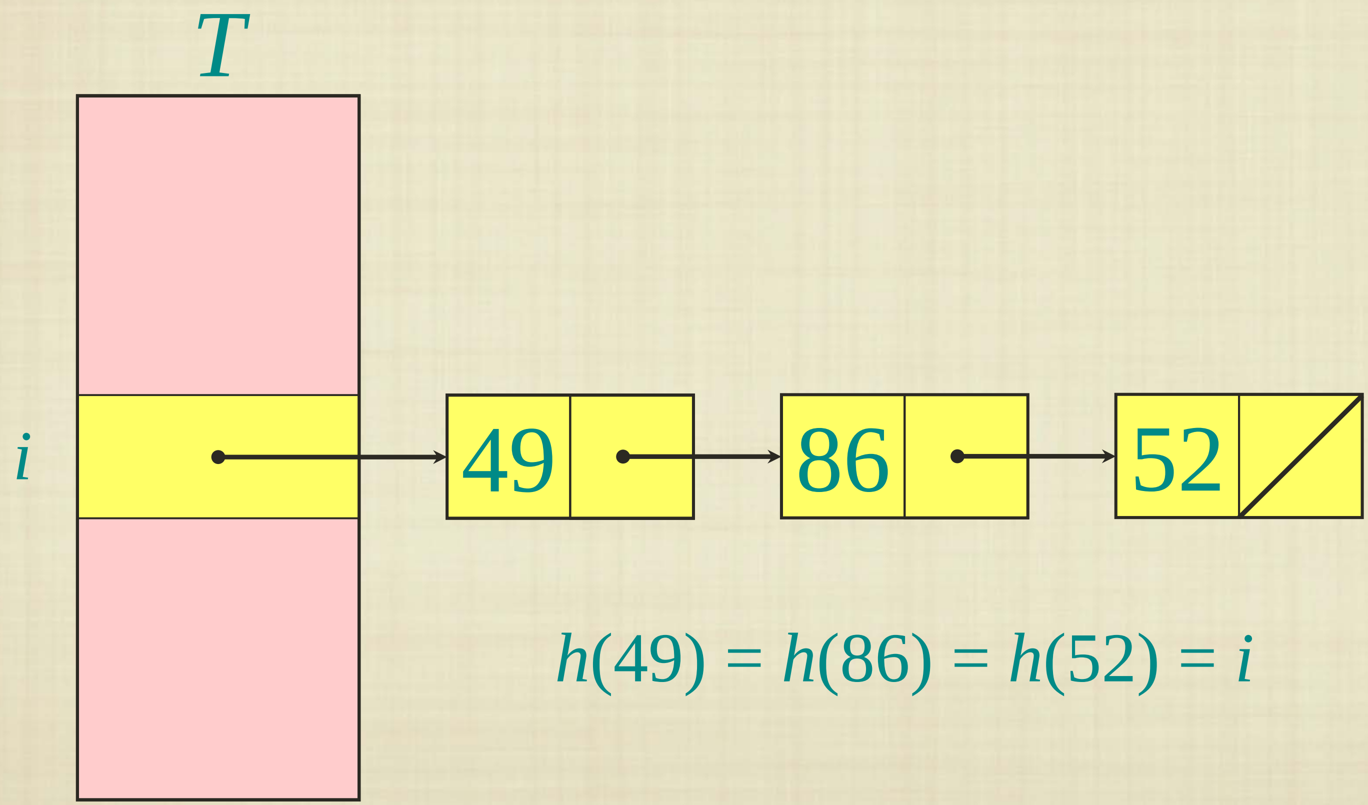
Solution: Use a ***hash function*** h to map the universe U of all keys into $\{0, 1, \ldots, n–1\}$:



When a record to be inserted maps to an already occupied slot in D, a ***collision*** occurs.

Resolving collisions by chaining

- Records in the same slot are linked into a list.



$h(49) = h(86) = h(52) = i$

Resolving collisions by open addressing (probing)

No storage is used outside of the hash table itself.

- Insertion *systematically* probes the table until an empty slot is found:
  - **Linear probing:** Try the next, the 2nd next, the 3rd next, the 4th next, … slot
  - **Quadratic probing:** Try the next, the 4th next, the 9th next, the 16th next,… slot
  - **Rehashing:** Repeatedly apply another hash function to find a sequence of slots

Resolving collisions by open addressing

- Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.

- The table may fill up, and deletion is difficult (but not impossible; usually deleted slots are not deleted but only marked as “deleted”).

Probing

```
class StringHashTable {
    ...
    static final int a = 1;
    static final int b = 0;

    private int probe(int h, int i){
      return (h + (a*i + b)) % dataTable.length;
    }

    public void add(String S){
      int h = hashCode(S);
      int i=1;
      int current = h;
      while(dataTable[current]!=null){
        current = probe(h,i);
        i++;
      }
      dataTable[current] = S;
    }
}
```

This is known as a “linear” probe.

Probing

This is known as a "quadratic" probe.

```
class StringHashTable {
    ...
    static final int a = 1;
    static final int b = 0;
    Static final int c = 0;

    private int probe(int h, int i){
       return (h + (a*i*i +b*i + c)) % dataTable.length;
    }

    public void add(String S){
       int h = hashCode(S);
       int i=1;
       int current = h;
       while(dataTable[current]!=null){
         current = probe(h,i);
         i++;
       }
       dataTable[current] = S;
    }
}
```

What happens if the data table is "full"?

Hash Functions

- Really, hashing just a “trick” that makes use of key values being in a small range. When can we use this trick?

- Let $\mathcal{U}$ be our elements of a particular data type, and let n be the size of our table. We need a mapping from elements to table indices.

- We want the hash function to have the following properties:

$$h : \mathcal{U} \rightarrow \{0, 1, \ldots, n-1\}$$

$$x = y \Rightarrow h(x) = h(y)$$

Choosing a hash function

- Theoretically, it is possible to devise a “perfect” hash function, but these solutions are not often used in practice.

- Hash functions are typically “engineered” to work well in practice for particular data types (e.g. `String`).

- Finding a good practical hash function is an ongoing research topic.

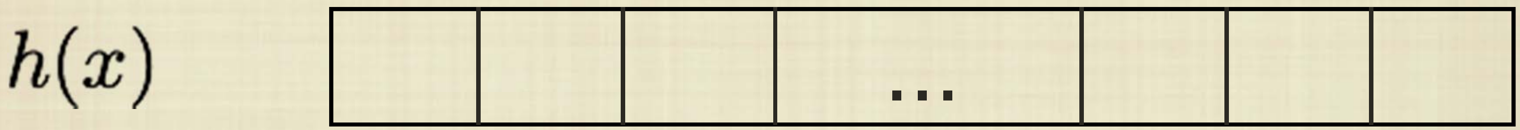
- Runtime depends on the

$$\text{load factor} = \frac{\text{number of keys stored in table}}{\text{number of slots in table}}$$

- For good hash functions, few collisions occur and the runtime is close to O(1)

Hash Tables

A hash table is defined by a hash function and the policy by which we resolve collisions.

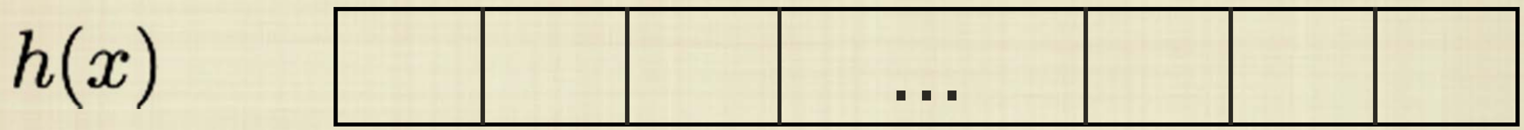


	Add	Find
Probing:		
Chaining:		

What is the absolute worst-case performance of a hash table under either collision policy?

Hash Tables

A hash table is defined by a hash function and the policy by which we resolve collisions.



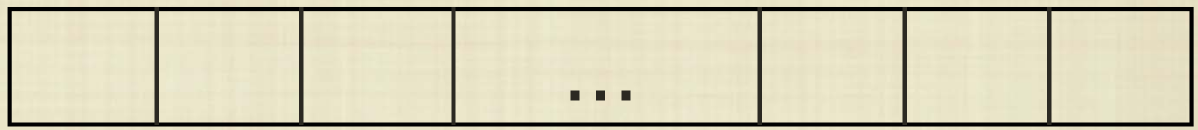
	Add	Find
Probing:	$O(n)$	$O(n)$
Chaining:	$O(n)$	$O(n)$

What is the absolute worst-case performance of a hash table under either collision policy?

Hash Tables

A hash table is defined by a hash function and the policy by which we resolve collisions.

$h(x)$



	Add	Find
Probing:	$\approx O(1)$	$\approx O(1)$
Chaining:	$\approx O(1)$	$\approx O(1)$

Hashing is a black art - we strive to choose a table size and hashing function that gives good performance.

Collections and Maps

- The `Collection` interfaces is for storage and access, while a `Map` interface is geared towards associating keys with objects.