Honors Data Structures

Lecture 13:
Hash tables I

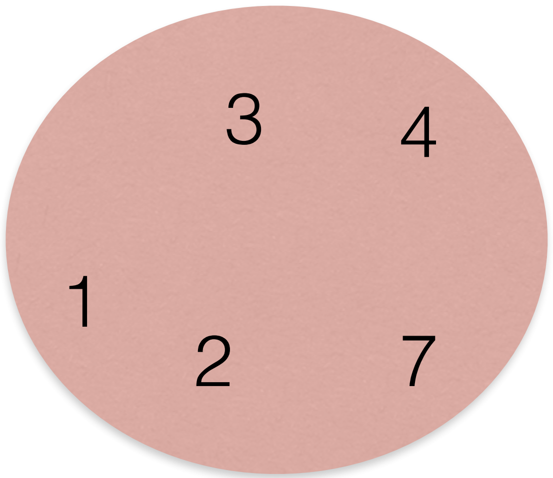
3/2/22

Daniel Bauer

Set ADT

- A Set is a collection of data items that does not allow duplicates.

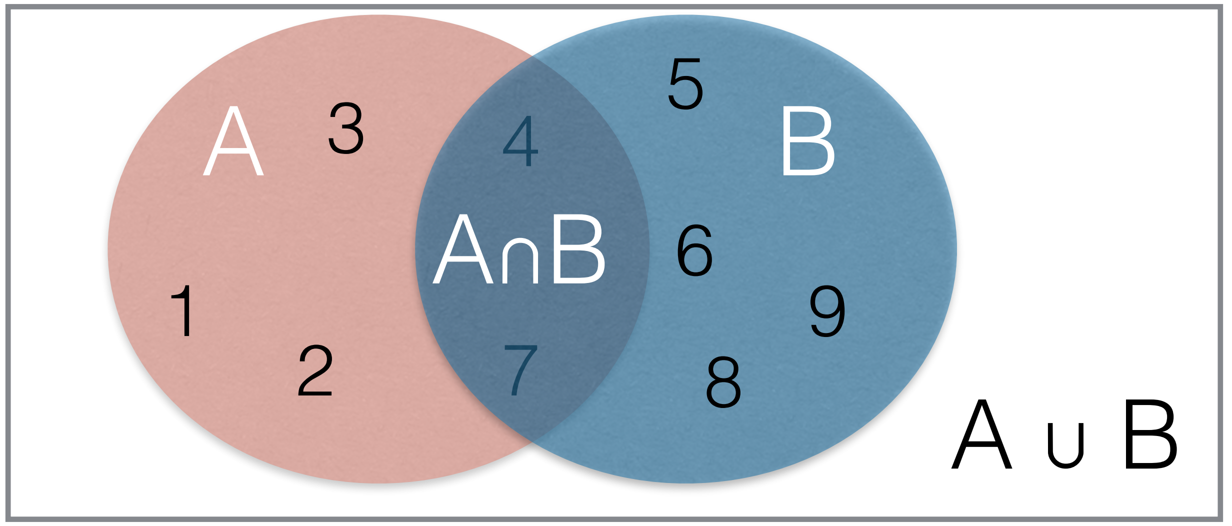
- Supported operations:
  - `insert(x)`
  - `remove(x)`
  - `contains(x)`
  - `isEmpty()`
  - `size()`



Set ADT

- A Set is a collection of data items that does not allow duplicates.

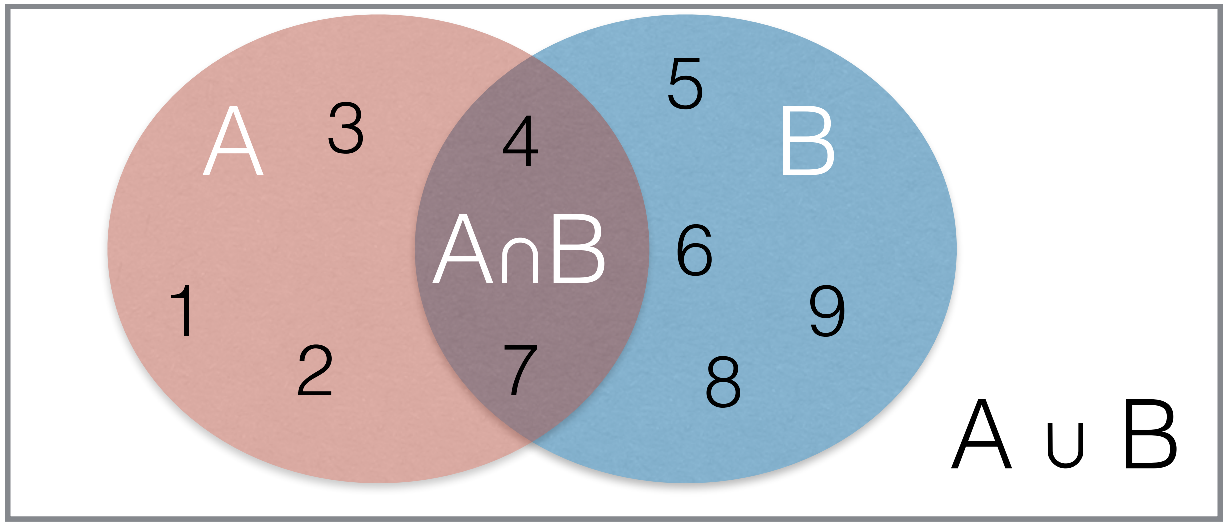
- Supported operations:
  - `insert(x)`
  - `remove(x)`
  - `contains(x)`
  - `isEmpty()`
  - `size()`
  - `addAll(s) / union(s)`
  - `removeAll(s)`
  - `retainAll(s) / intersection(s)`



`OrderedSet` ADT

- A set with a total order defined on the items (all pairs of items are in a ‘>’ or ‘<‘ relation to each other).

- Supported operations: all `Set` operations and
  - `findMin()`
  - `findMax()`



Implementing Sets

Implementing Sets

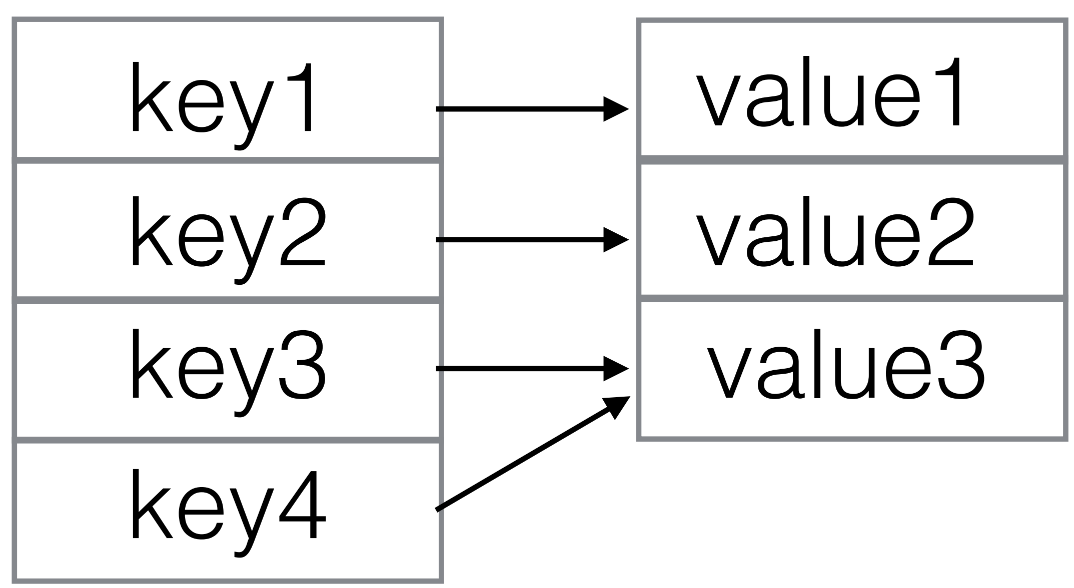
- Naive implementation: LinkedList, ArrayList (bad!)
    - Need to be able to check for item equality.
    - Running time of all operations at least O(N), because we need to check for membership first.

Implementing Sets

- Naive implementation: LinkedList, ArrayList (bad!)
  - Need to be able to check for item equality.
  - Running time of all operations at least O(N), because we need to check for membership first.
- Better: implement ordered sets as search trees.
  - With balanced search trees:
    O(log N) for insert, remove, contains.
  - Need to be able to compare every pair of items.
    Implement the `Comparable` interface.

Map ADT

- A *map* is collection of *(key, value)* pairs.
- Keys are unique, values need not be (keys are a Set!).
- Two operations:
  - `get(key)` returns the value associated with this key
  - `put(key, value)` (overwrites existing keys)



Arrays as Maps

- When keys are integers, arrays provide a convenient way of implementing maps.

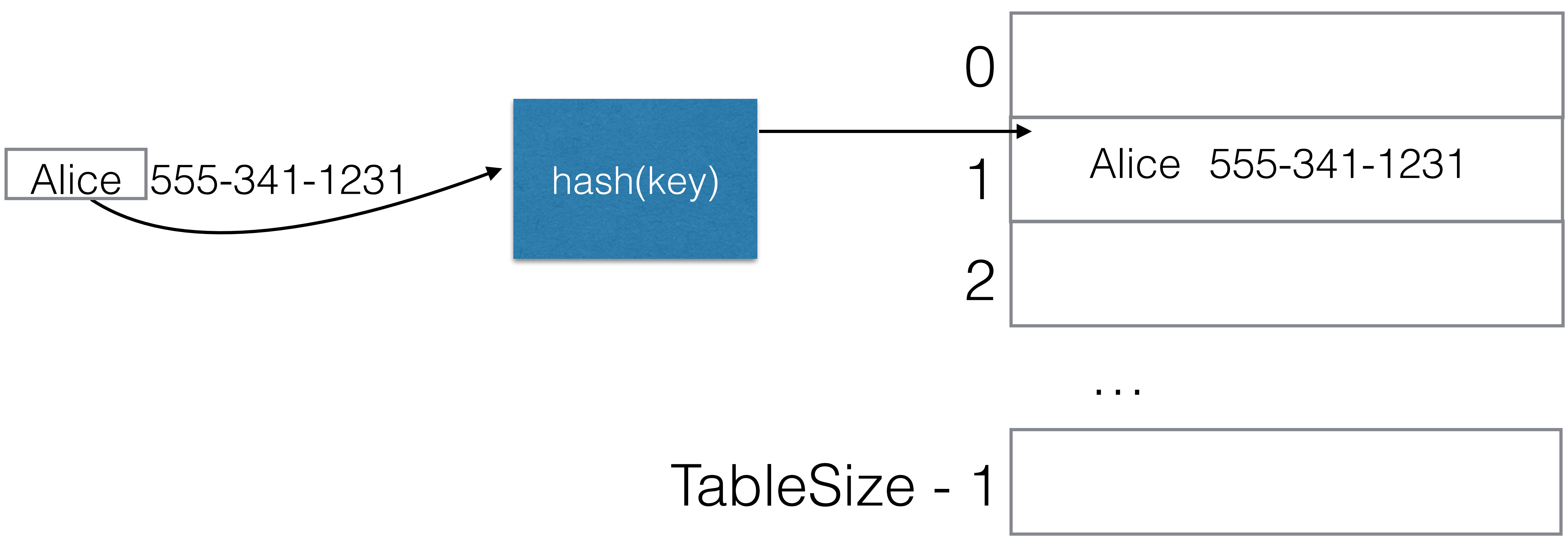
- Time for `get` and `put` is O(1).

A		B		D	C	
0	1	2	3	4	5	6

- What if we don't have integer keys? Any other potential issues?

Hash Tables

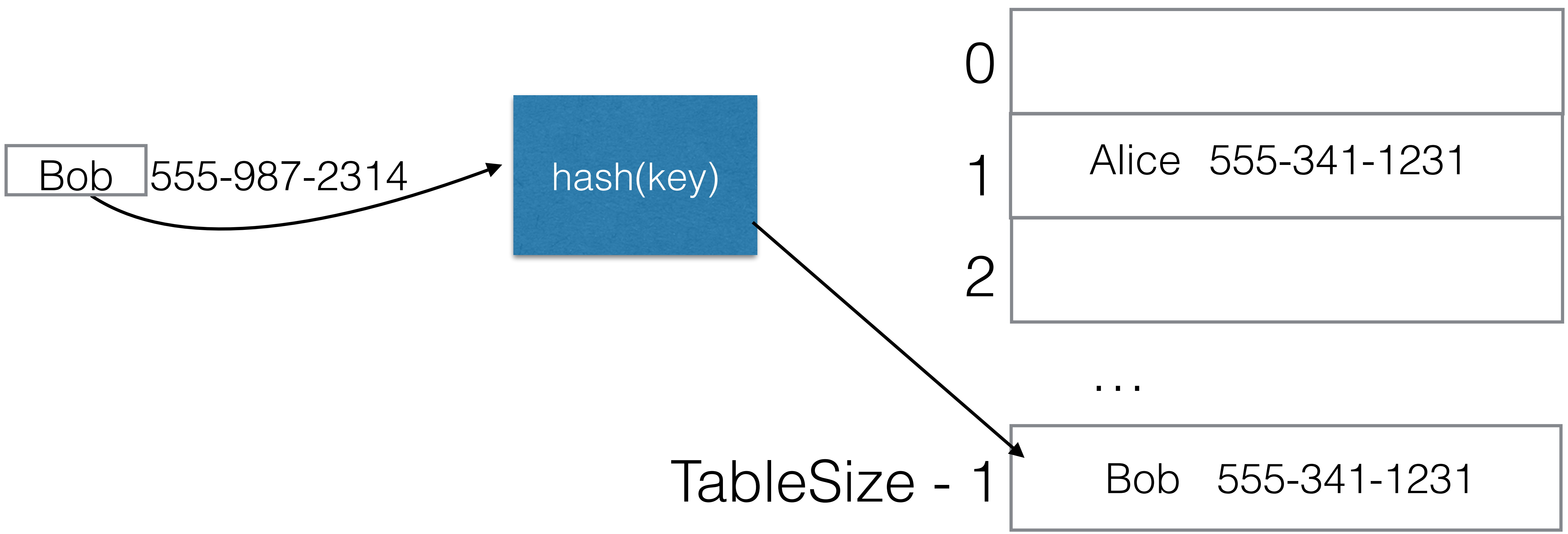
- Define a table (an array) of some length *TableSize.*
- Define a function `hash(key)` that maps key objects to an integer index in the range *0 … TableSize -1*



Hash Tables

- Define a table (an array) of some length *TableSize.*

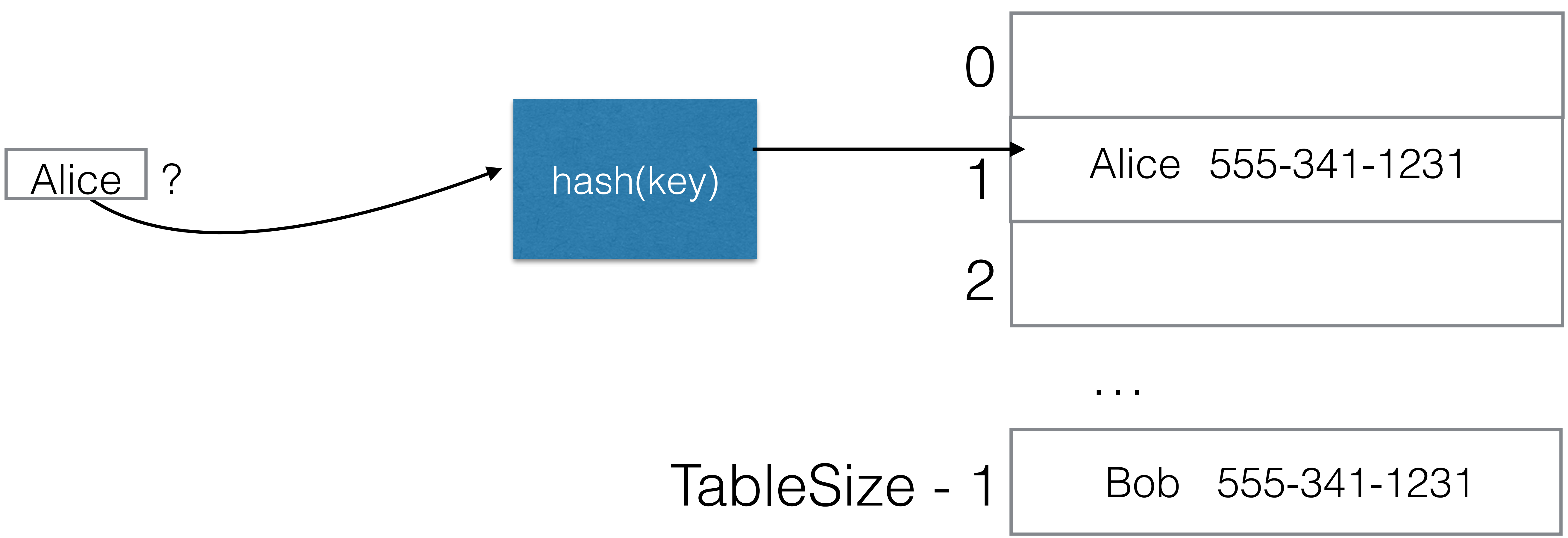
- Define a function `hash(key)` that maps key objects to an integer index in the range *0 … TableSize -1*



Hash Tables

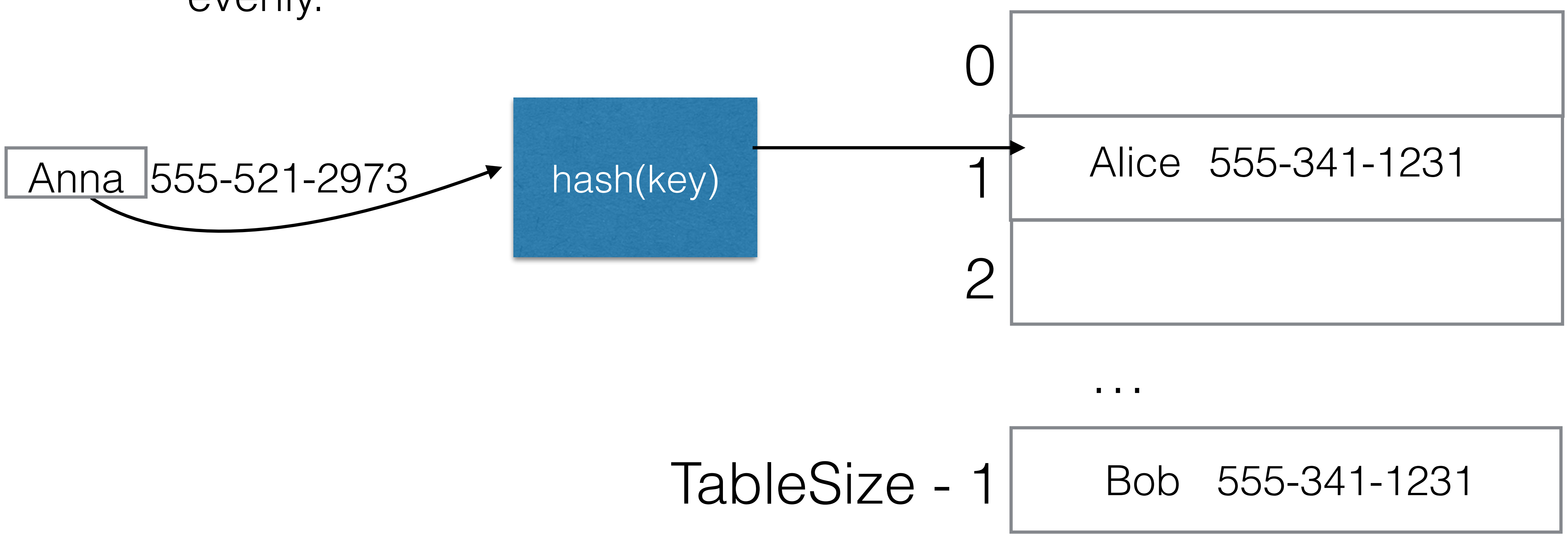
- Lookup/get: Just hash the key to find the index.

- Assuming hash(key) takes constant time, get and put run in O(1).



Hash Table Collisions

- Problem: There is an infinite number of keys, but only *TableSize* entries in the array.
    - How do we deal with collisions? (new item hashes to an array cell that is already occupied)
    - Need to find a hash function that distributes items in the array evenly.



Choosing a Hash Function

- Hash functions depends on: type of keys we expect (Strings, Integers…) and *TableSize*.
- Hash functions needs to:
  - Spread out the keys as much as possible in the table (ideal: uniform distribution).
  - Make sure that all table cells can be reached.

Choosing a Hash Function: Integers

- If the keys are integers, it is often okay to assume that the possible keys are distributed evenly.

hash(x) = x % TableSize

```
public static int hash( Integer key, int tableSize ) {
    return key % tableSize;
}
```

e.g. TableSize = 5
hash(0) = 0, hash(1) = 1,
hash(5) = 0, hash(6) = 1

Choosing a Hash Function: Strings - Idea 1

- Idea 1: Sum up the ASCII (or Unicode) values of all characters in the String.

```
public static int hash( String key, int tableSize ) {
        int hashVal = 0;

        for( int i = 0; i < key.length( ); i++ )
            hashVal = hashVal + key.charAt( i );

        return hashVal % tableSize;
}
```

e.g. “Anna” → $65 + 2 \cdot 110 + 97 = 382$

A → 65, n → 110, a → 97

Choosing a Hash Function: Strings - Problems with Idea 1

- Idea 1 doesn't work for large table sizes:
  - Assume *TableSize = 10,007*
  - Every character has a value in the range 0 and 127.
  - Assume keys are at most 8 chars long:
    - hash(key) is in the range 0 and $127 \cdot 8 = 1016$.
    - Only the first 1017 cells of the array will be used!

Choosing a Hash Function: Strings - Problems with Idea 1

- Idea 1 doesn't work for large table sizes:
  - Assume *TableSize = 10,007*
  - Every character has a value in the range 0 and 127.
  - Assume keys are at most 8 chars long:
    - hash(key) is in the range 0 and $127 \cdot 8 = 1016$.
    - Only the first 1017 cells of the array will be used!
- All anagrams will produce collisions: "rescued", "secured","seducer"

Choosing a Hash Function: Strings - Idea 2

- Idea 2: Only look at prefix.
  Spread out the value for each character.

```
public static int hash( Integer key, int tableSize ) {
    return (key.charAt(0) +
            27 * key.charAt(1) +
        27 * 27 * key.charAt(2)) % tableSize;
}
```

Choosing a Hash Function: Strings - Idea 2

- Idea 2: Only look at prefix.
  Spread out the value for each character.

```
public static int hash( Integer key, int tableSize ) {
    return (key.charAt(0) +
            27 * key.charAt(1) +
        27 * 27 * key.charAt(2)) % tableSize;
}
```

- Problem: assumes that the all three letter combinations (*trigrams*) are equally likely at the beginning of a string.

  - This is not the case for natural language
    - some letters are more frequent than others
    - some trigrams (e.g. “xvz”) don’t occur at all.

Choosing a Hash Function: Strings - Idea 3

```
public static int hash( String key, int tableSize ) {
    int hashVal = 0;

    for( int i = key.length()-1; i >= 0; i-- )
        hashVal = 37 * hashVal + key.charAt( i );

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;

    return hashVal;
}
```

$$key[N-1] \cdot 37^{N} + key[N-2] \cdot 37^{N-1} + \cdots + key[1] \cdot 37 + key[0]$$

This is what Java Strings use; works well, but slow for large strings.

Combining Hash Functions

- In practice, we often write hash functions for some container class:
  - Assume all member variables have a hash function (Integers, Strings…).
  - Multiply the hash of each member variable with some distinct, prime number.
  - Then sum them all up.

Combining Hash Functions, Example

```
public class Person {
    public String firstName;
    public String lastName;
    public Integer age;
}
```

Combining Hash Functions, Example

```
public class Person {
    public String firstName;
    public String lastName;
    public Integer age;
}
```

```
public static int hash( Person key, int tableSize ) {
    int hashVal =  hash(key.firstName, tableSize) * 127 +
                   hash(key.lastName, tableSize) * 1901 +
                   hash(key.age, tableSize) * 4591;
    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;
}
```

Table Size and Hash Functions

Table Size and Hash Functions

- Good practices:
  - Keep *TableSize* a **prime number.**
  - When combining hash values, make the factors **prime numbers.**
  - This minimizes collisions.

What Objects Can be Keys?

- Anything can be a key, we just need to find a good hash function.
- Need to make sure that objects that are used as keys cannot be changed at runtime (they are **immutable**)

What Objects Can be Keys?

- Anything can be a key, we just need to find a good hash function.

- Need to make sure that objects that are used as keys cannot be changed at runtime (they are **immutable**)

    - Otherwise, if their content changes their hash value should change too!

What Objects Can be Keys?

- Anything can be a key, we just need to find a good hash function.

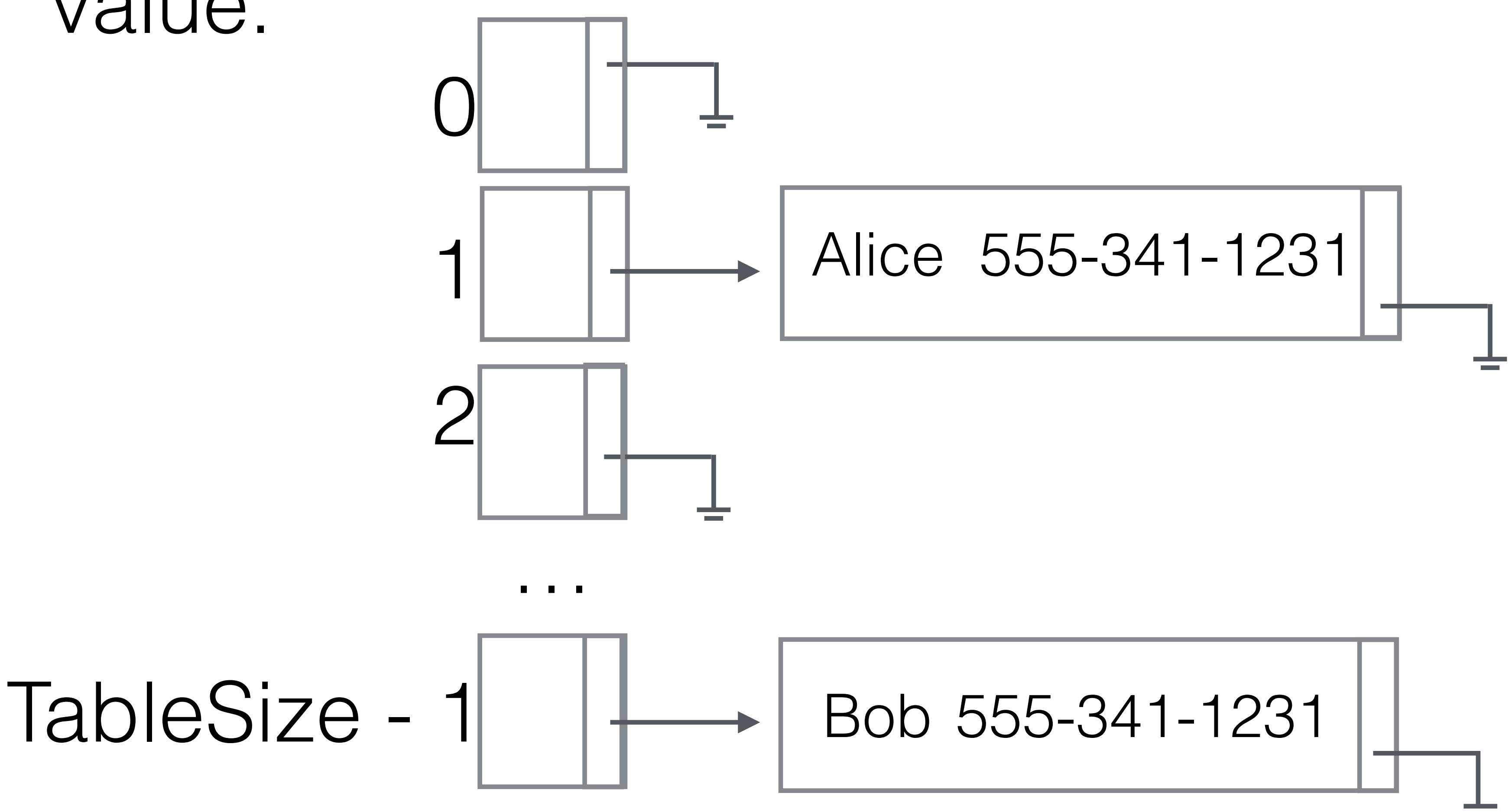
- Need to make sure that objects that are used as keys cannot be changed at runtime (they are **immutable**)

    - Otherwise, if their content changes their hash value should change too!

- How would you compute the hash value for a LinkedList or a Binary Tree?

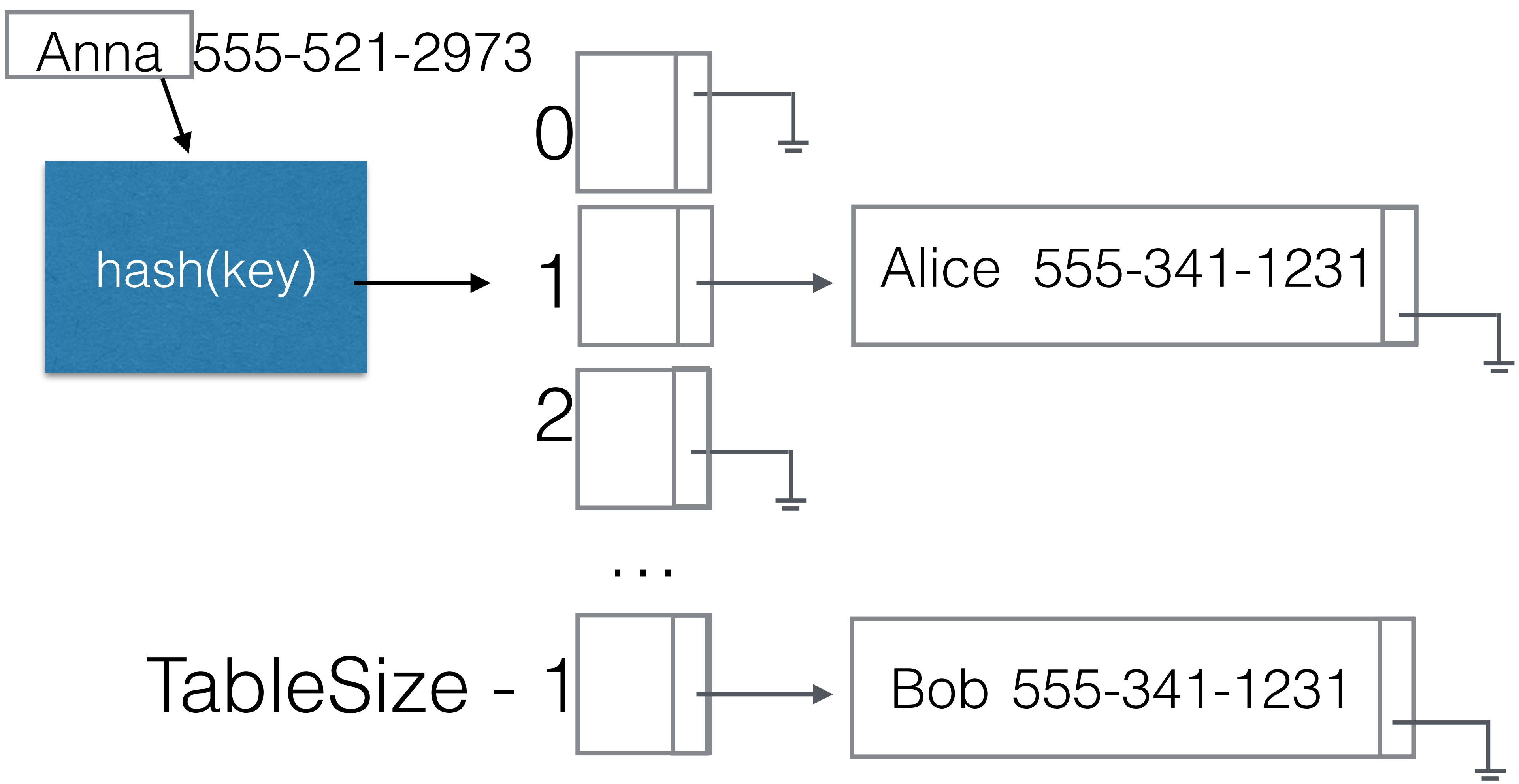
Dealing with Collisions: Separate Chaining

- Keep all items whose key hashes to the same value in a list.
- Can think of each list as a *bucket* defined by the hash value.



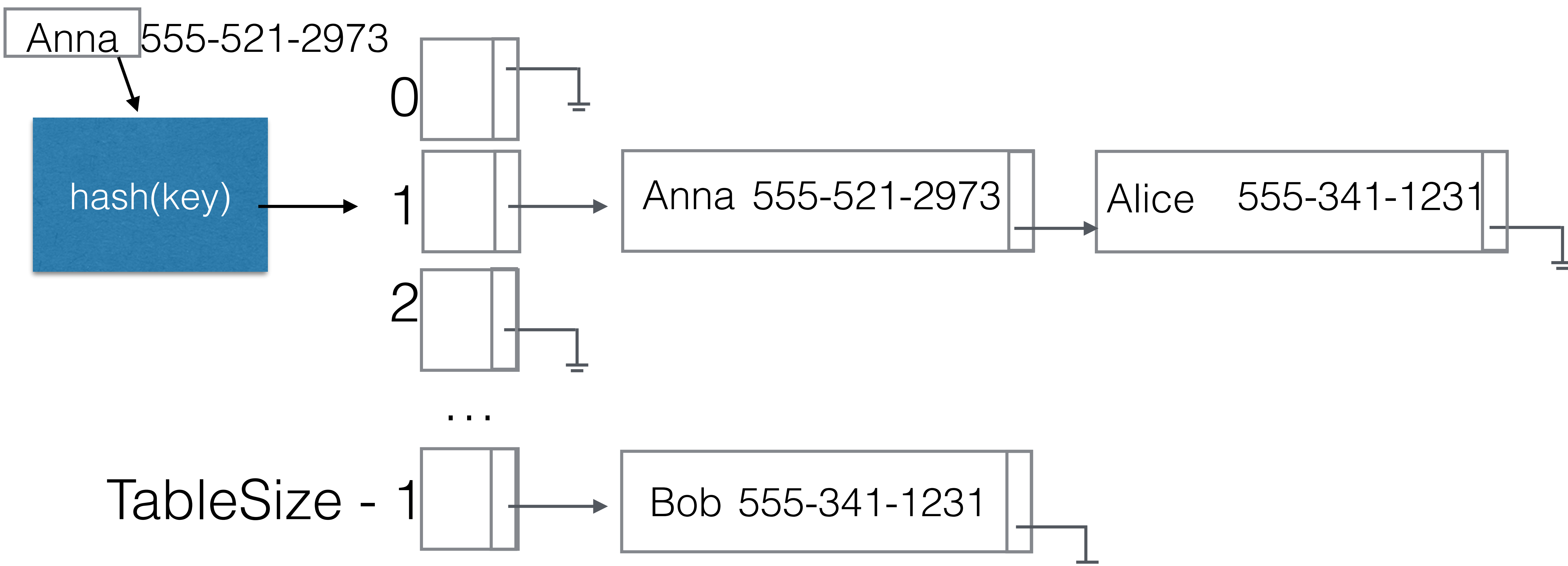
Dealing with Collisions: Separate Chaining

- To insert a new key in cell that's already occupied prepend to the list.



Dealing with Collisions: Separate Chaining

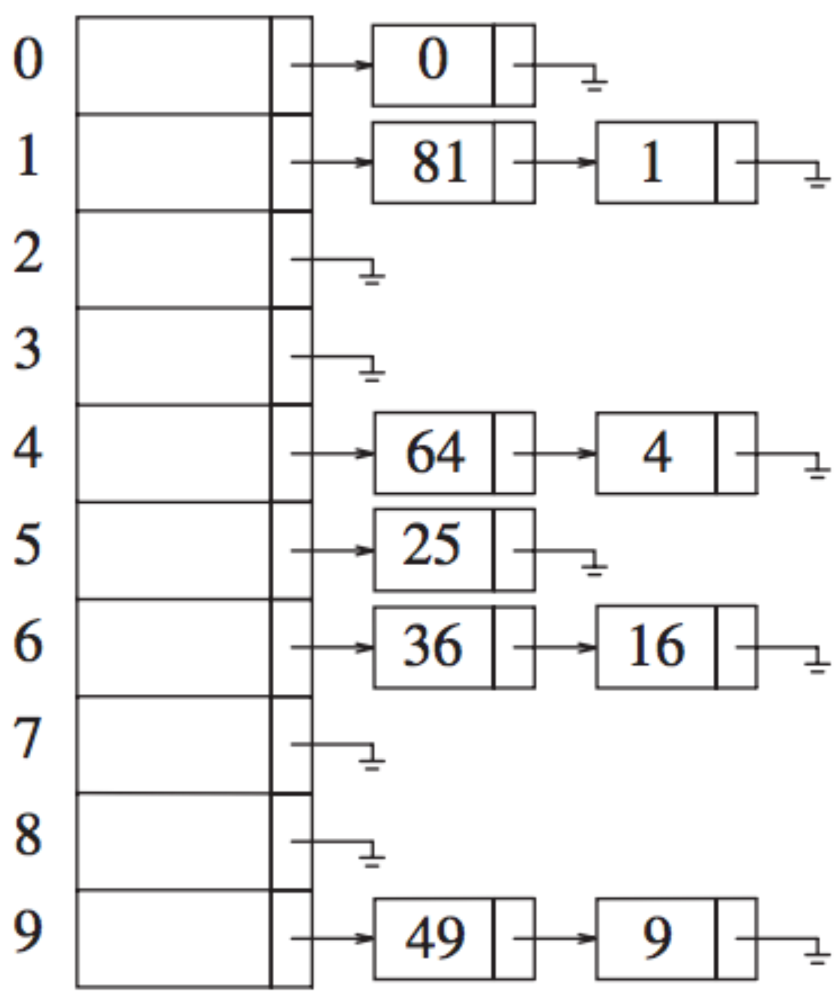
- To insert a new key in cell that's already occupied prepend to the list.



Analyzing Running Time for Separate Chaining (1)

- Time to find a key = time to compute hash function
  + time to traverse the linked list.
- Assume hash functions computed in O(1).
- How many elements do we expect in a list on average?

Load Factor



- Let N be the number of keys in the table.
- Define the load factor as

$$\lambda = \frac{N}{TableSize}$$

- The average length of a list is λ .