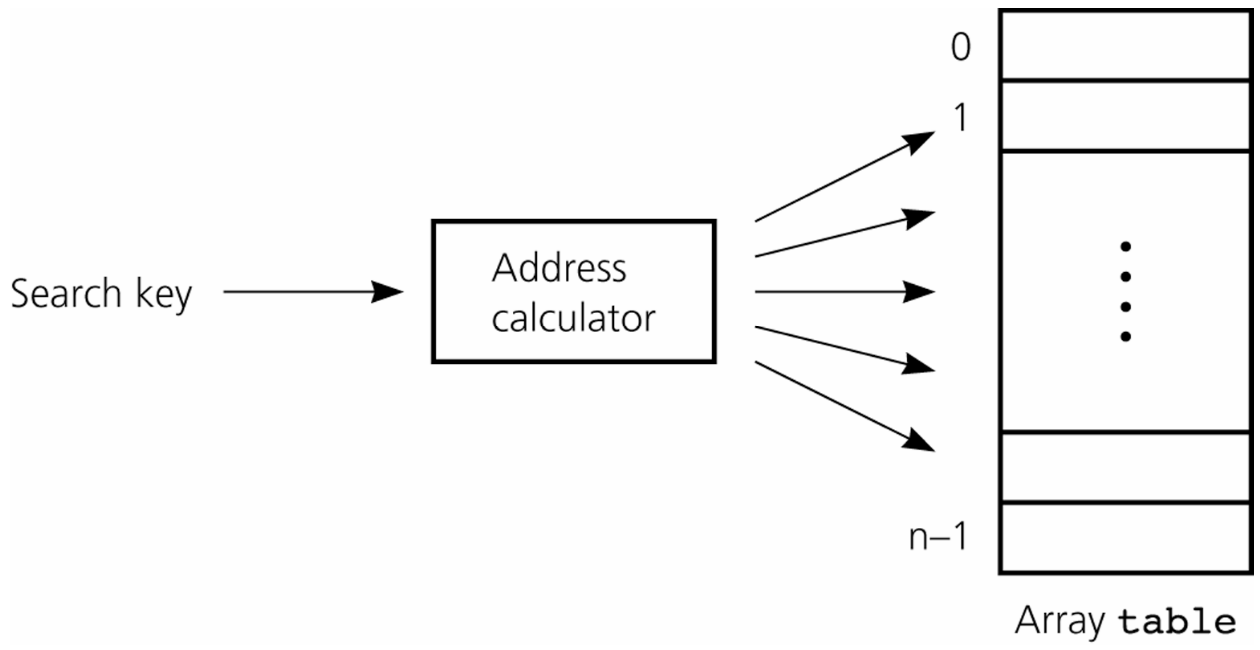
Using arrays – Example 2: names as keys

- How do we map strings to integers?
  - One way is to convert each letter to a number, either by mapping them to 0-25 or their ASCII characters or some other method and concatenating those numbers.
  - So how many possible arrangements of letters are there for names with a maximum of (say) 10 characters? The first letter can be one of 26 letters. Then for each of these 26 possible first letters there are 26 possible second letters (for a total of 26* 26 or 26^2 arrangements). With ten letters there are 26^{10} possible strings, i.e. 141,167,095,653,376 possible keys!

Hash function: mapping key to index

So far this approach (of converting the key to an integer which is then used as an index to an array whose size is equal to the largest possible integer key) is not looking very promising. What would be more useful would be to pick the size of the array we were going to use (based on how many customers, or citizens, or items we think we want to keep track of) and then somehow map the keys to the array indices (which would range from 0 to our array size-1). This map is called a **hash function**.



Hash Table

- A hash table consists of
  - an array to store data in and
  - a hash function to map a key an array index.
- We can assume that the array will contain references to objects of some data structure. This data structure will contain a number of attributes, one of which must be a *key value* used as an index into the hash table. We'll assume that we can *convert the key to an integer* in some way. We can map that to an array index using the modulo (or remainder) function:
- **Simple hash function:** *h(key)* = *key* % *array_size* where *h(key)* is the hash value (array index) and % is the modulo operator.
- **Example:** using a customer phone number as the key, assume that there are 500 customer records and that we store them in an array of size 1,000. A record with a phone number of 604-555-1987 would be mapped to array element 987 (6,045,551,987 % 1,000 = 987).

A problem – collisions

- Let’s assume that we make the array size (roughly) double the number of values to be stored in it.
  - This a common approach (as it can be shown that this size is minimal possible for which hashing will work efficiently).
- We now have a way of mapping a numeric key to the range of array indices.

- However, there is no guarantee that two records (with different keys) won’t map to the same array element (consider the phone number 512-555-7987 in the previous example). When this happens it is termed a **collision**.
- There are *two issues* to consider concerning collisions:
  - *how to minimize the chances of collisions occurring and*
  - *what to do about them when they do occur*.

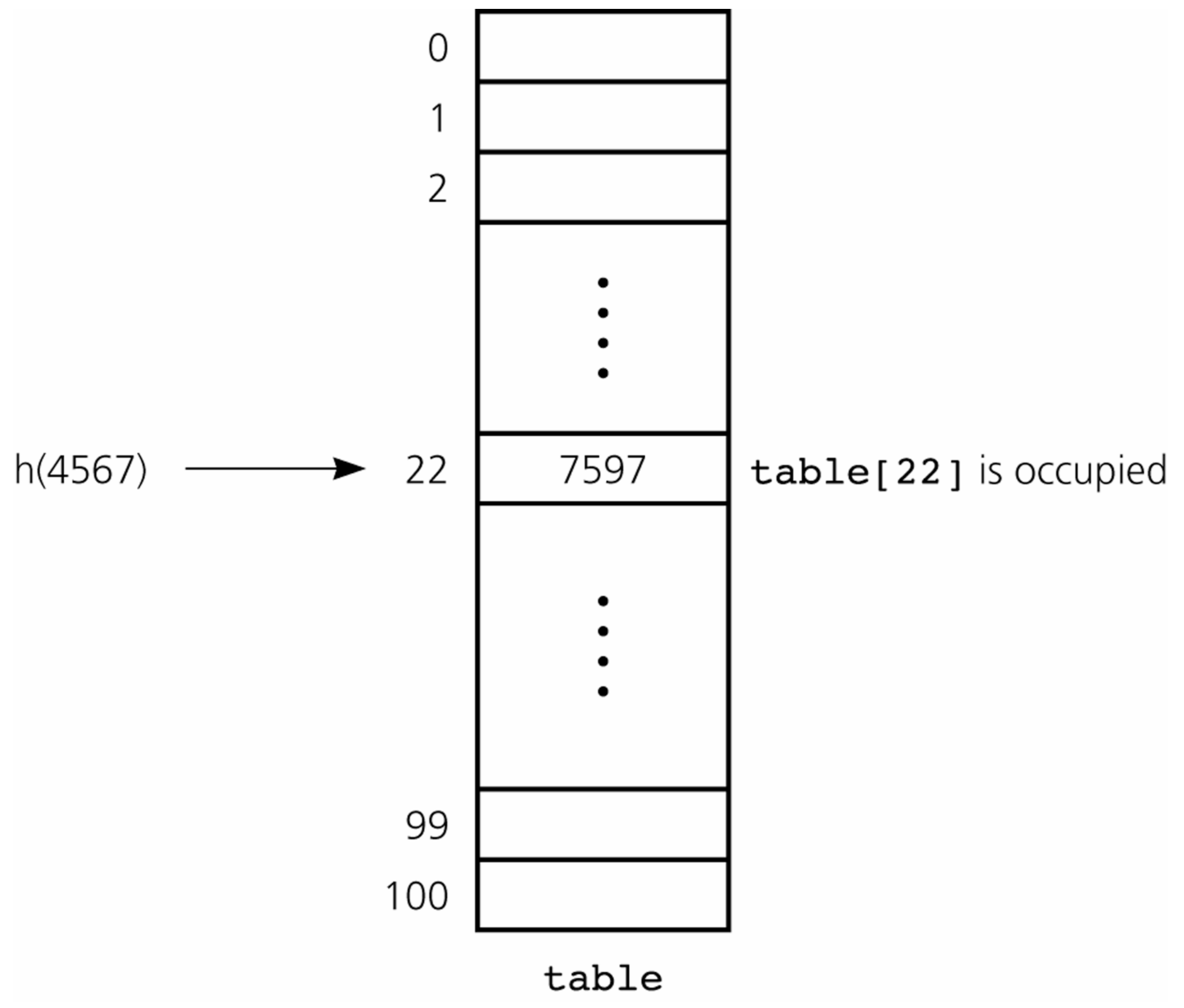


Figure: A collision

Minimizing Collisions by Determining a Good Hash Function

- A good hash function will *reduce the probability of collisions* occurring, while a bad hash function will increase it. Let's look at an example of a bad hash function first to illustrate some of the issues.
- **Example:** Suppose I want to store a few hundred English words in a hash table. I create an array of 26^2 (676) and map the words based on the first two letters in the word. So, for example the word "earth" might map to index 104 (e=4, a=0; 4*26 + 0 = 104) and a word beginning with "zz" would map to index 675 (z = 25*26 + 25 = 675).
  - **Problem:** The flaw with this scheme is that the universe of possible English words is not uniformly distributed across the array. There are many more words beginning with "ea" or "th" than there are with "hh" or "zz". So this scheme would probably generate many collisions while some positions in the array would be never used.
- Remember this is an example of a bad hash function!

A Good Hash Function

- First, it should be fast to compute.
- A good hash function should result in each key being equally likely to hash to any of the array elements. Or other way round: each index in the array should have same probability to be mapped an item (considering the distribution of possible datas).
  - Well, the best function would be a random function, but that **doesn’t work:** we would be not able to find an element once we store it in the table, i.e.
    the function has to return the same index each time it is a called on the same key.
- To achieve this it is usual to determine the hash value so that it is independent of any patterns that exist in the data. In the example above the hash value *is* dependent on patterns in the data, hence the problem.

A Good Hash Function

- Independent hash function:
  - Express the key as an integer (if it isn't already one), called **hash value** or **hash code**. When doing so remove any non-data (e.g. for a hash table of part numbers where all part numbers begin with 'P', there is don't to include the 'P' as part of the key), otherwise **base the integer on the entire key**.
  - Use a **prime** number as the size of the array (independent from any constants occurring in data).

- There are other ways of computing hash functions, and much work has been done on this subject, which is beyond the scope of this course.

How do we map string key to hash code?

- How do we map strings to integers?
  - Convert each letter to a number, either by mapping them to 0-25 or their ASCII characters or some other method.
  - Concatenating those values to one huge integer is not very efficient (or if the values are let to overflow, most likely we would just ignore the most of the string).
  - Summing the number doesn't work well either ('stop', 'tops', 'pots', 'spot')
  - Use polynomial hash codes:
    $x_0a^{k-1}+x_1a^{k-2}+\ldots+x_{k-2}a+x_{k-1}$,
    where a is a prime number (33,37,39,41 works best for English words) [remark: and let it overflow]

Hashing summary

- Determine the size *m* of the hash table's underlying array. The size should be:
  - approximately **twice the size** of the expected number of records and
  - a **prime number**, to evenly distribute items over the table.
- Express the key as the integer such that it depends on the entire key.
- Map the key to the hash table index by calculating the remainder of the key, *k*, divided by the size of the hash table *m*: $h(k) = k \bmod m$.

Dealing with collisions

- Even though we can reduce collisions by using a good hash function they will still occur.
- There are two main approaches of dealing with collisions:
  - The first is to find somewhere else to insert an item that has collided **(open addressing)**;
  - the second is to make the hash table an array of linked lists **(separate chaining)**.

Open Addressing

- The idea behind open addressing is that when a *collision occurs* the new value is inserted in a *different index* in the array.
- This has to be done in a way that allows the value to be found again.
- We'll look at three separate versions of open addressing. In each of these versions, the **"step" value** is a distance from the original index calculated by the hash function.
  - The original index plus the step gives the new index to insert a record at if a collision occurs.

Open addressing: Linear Probing

- the simplest method
- In linear probing the step increases **by one** each time an insertion fails to find space to insert a record:
  - So, when a record is inserted in the hash table, if the array element that it is mapped to is occupied we look at the next element. If that element is occupied we look at the next one, and so on.
- **Disadvantage of this method:** sequences of occupied elements build up making the step values larger (and insertion less efficient); this problem is referred to as ***primary clustering*** ***("The rich gets richer")***.
- Clustering tends to get worse as the hash table fills up (has many elements – more than ½ full). This means that more comparisons (or probes) are required to look up items, or to insert and delete items, reducing the efficiency of the hash table.

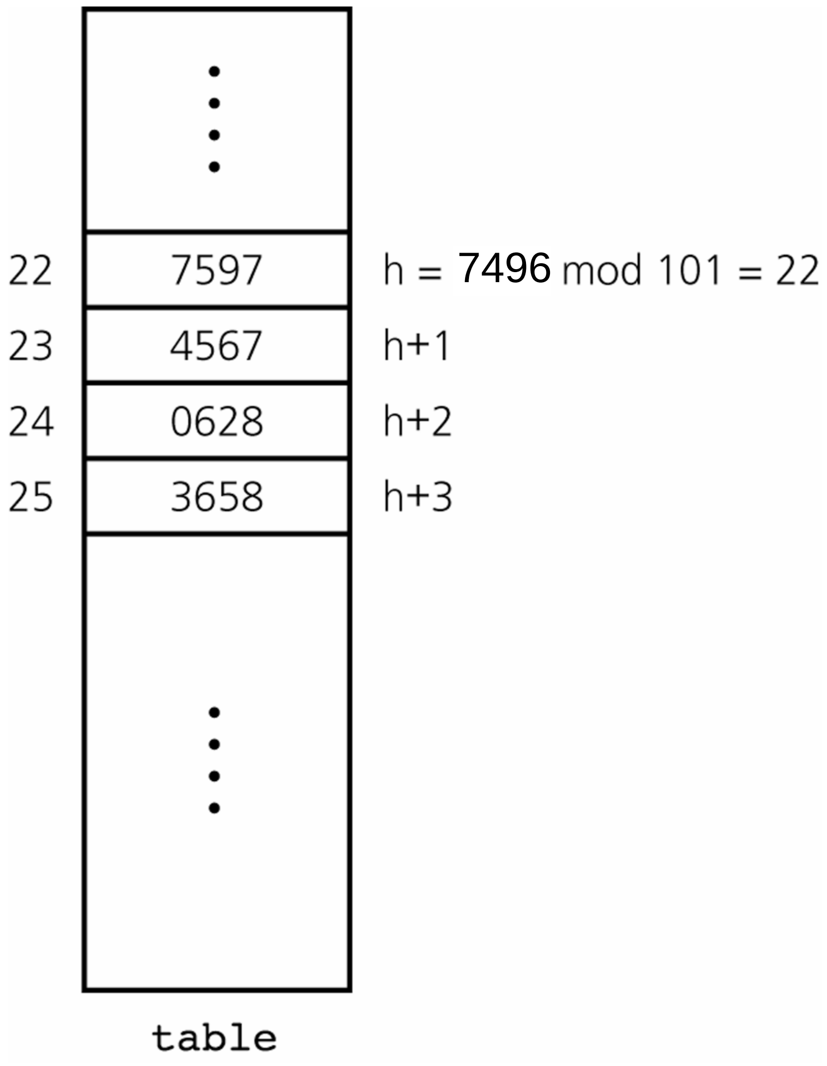


Figure: Linear probing with $h(x) = x \bmod 101$

Implementation

- **Insertion:** described on previous slides
- **Searching:** it’s not enough to look in the hash array at index where the key (hash code) was mapped, but we have to continue “probing” until we find either the element with the searched key or an empty spot (“not found”)
- **Deleting:** We cannot just make a spot empty, as we could interrupt a probe sequence. Instead we mark it AVAILABLE, to indicate that the spot can be used for insertion, but searching should continue when AVAILABLE spot is encountered.

Implementation

- Interface:

```
public interface HashTableInterface<T extends KeyedItem> {
  public void insert(T item) throws HashTableFullException;
  // PRE: item.getKey()!=0
  public T find(long key);
  // PRE: item.getKey()!=0
  // return null if the item with key 'key' was not found
  public T delete(long key);
  // PRE: item.getKey()!=0
  // return null if the item with key 'key' was not found
}
```

Implementation

- Data members and helping methods:

```
public class HashTable<T extends KeyedItem>
implements HashTableInterface<T>
{
  private KeyedItem[] table;
  // special values: null = EMPTY, T with key=0 = AVAILABLE
  private static KeyedItem AVAILABLE = new KeyedItem(0);

  private int h(long key) // hash function
  // return index
  { return (int)(key % table.length); }// typecast to int

  private int step(int k) // step function
  { return k; } // linear probing

  public HashTable(int size)
  { table = new KeyedItem[size]; }
```

Implementation

- Insertion:

```
public void insert(T item) throws HashTableFullException
 {
   int index = h(item.getKey());
   int probe = index;
   int k = 1; // probe number
   do {
     if (table[probe]==null || table[probe]==AVAILABLE) {
       // this slot is available
       table[probe] = item;
       return;
     }
     probe = (index + step(k)) % table.length; // check next slot
     k++;
   } while (probe!=index);
   throw new HashTableFullException("Hash table is full.");
 }
```