Algorithms and Constructs

Hashing and Maps

Amilcar Aponte

amilcar@cct.ie

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database

Methods of a Map

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Performance of a List-Based Map

- **Performance:**
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

Hashing - Motivation

- Let’s think for a second about an ordinary array
- All the elements would be in a specific position in the array.
- What’s the complexity of indexing an element in the array when we know its position? – O(1)
- What’s the complexity of looking for a value in the array when we don’t know its position? – Depending on the algorithm, the best we’ve seen is O(log n)

What is Hashing?

- If you know the index of an element in the array, you can retrieve the element using the index in O(1) time. So, can we store the values in an array and use the key as the index to find the value? The answer is yes if you can map a key to an index.
- The array that stores the values is called a hash table. The function that maps a key-value pair to an index in the hash table is called a hash function.
- Hashing is a technique that retrieves the value using the index obtained from key without performing a search.

Hash Functions and Hash Tables

- A hash function $\boldsymbol{h}$ maps keys of a given type to integers in a fixed interval $[0, \boldsymbol{N} - 1]$
  - N is the size of the array that will store the data
- Example:

  $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{x} \bmod \boldsymbol{N}$

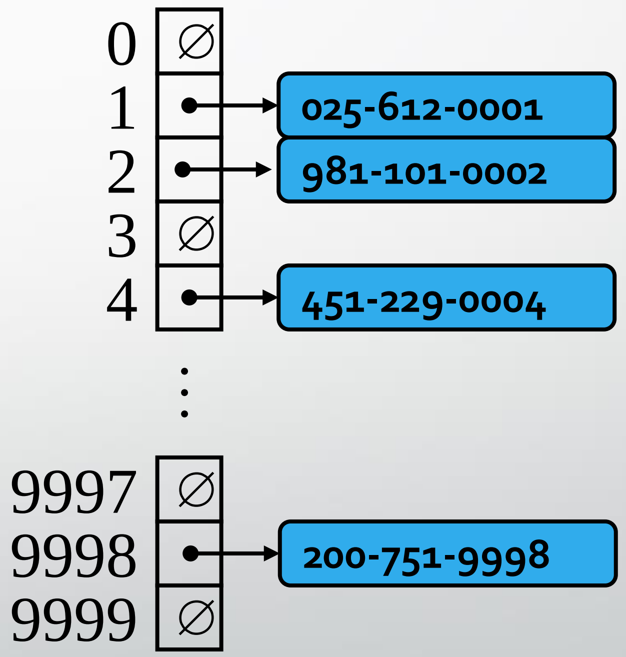
  is a hash function for integer data
- The integer $\boldsymbol{h}(\boldsymbol{x})$ is called the hash value of the key $\boldsymbol{x}$

Hash Functions and Hash Tables

- A hash table for a given key type consists of
  - Hash function $\boldsymbol{h}$
  - Array (called table) of size $\boldsymbol{N}$
- When implementing a map with a hash table, the goal is to store the key-value pair at index $\boldsymbol{i} = \boldsymbol{h}(\boldsymbol{k})$

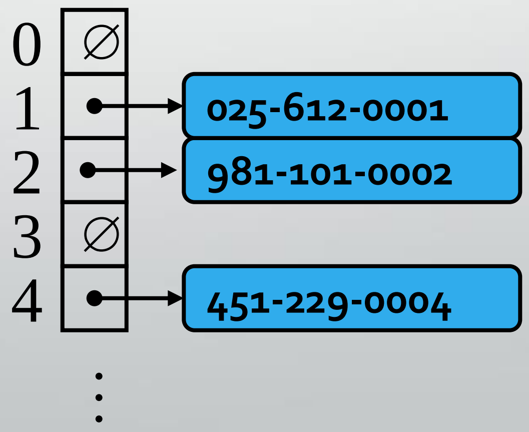
Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) = $ last four digits of x



More General Kinds of Data

- But what should we do if our keys are not integers in the range from 0 to N – 1?
    - Use a **hash function** to map general data to corresponding indices in a table.
    - For instance, the last four digits of a Social Security number.



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:
  $\boldsymbol{h}_1$: keys $\rightarrow$ integers

  Compression function:
  $\boldsymbol{h}_2$: integers $\rightarrow [0, \boldsymbol{N} - 1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,
- $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$
- The goal of the hash function is to "disperse" the keys in an apparently random way

Example

- Imagine we have a list of words that we want to sort on a hash table.
    - Cat
    - Dog
    - Mouse
    - Ear

- We’ll assign a number for each letter of the alphabet
    - A = 1
    - B = 2
    - And so on…

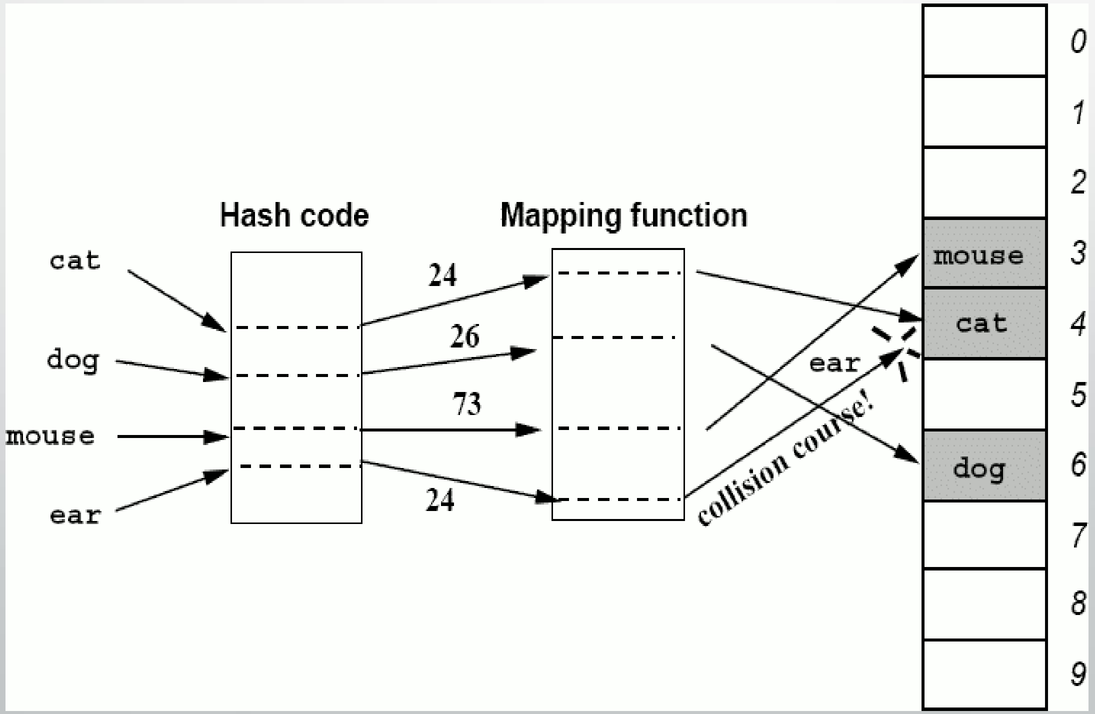
Hashing

- The hash function will be sum of the values of individual letters in the word

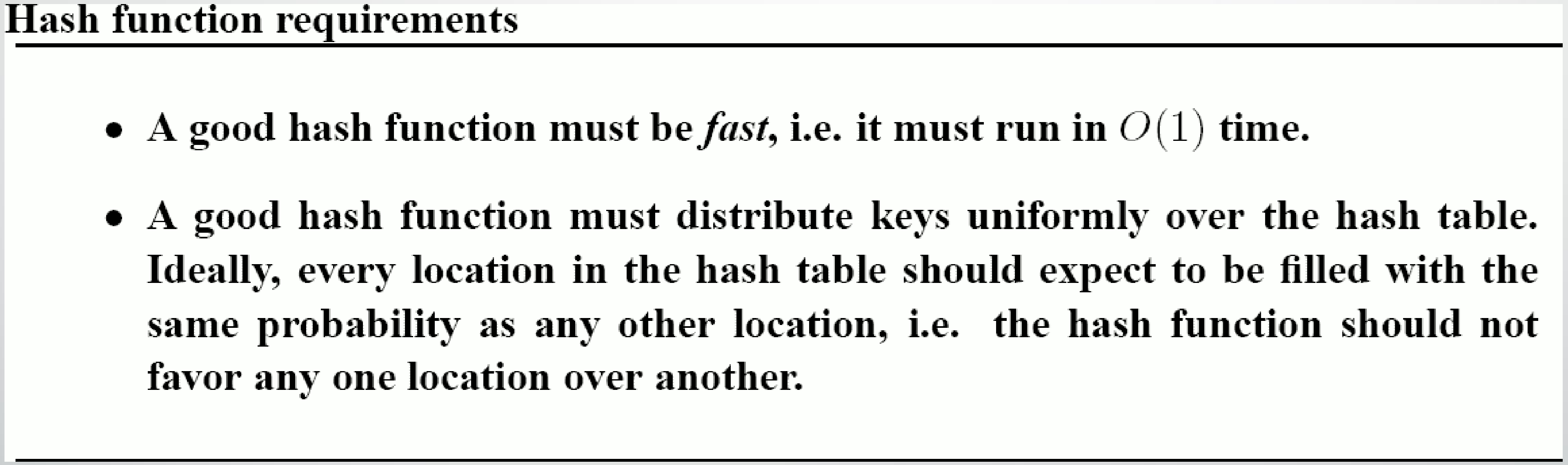
$$\text{cat} \equiv 3 + 1 + 20 = 24$$
$$\text{dog} \equiv 4 + 15 + 7 = 26$$
$$\text{mouse} \equiv 13 + 15 + 21 + 19 +$$
$$5 = 73$$
$$\text{ear} \equiv 5 + 1 + 18 = 24$$

Hashing

- The compression function will be
  - h(k) = k mod 10
- Where k is the hashed value obtained from the hash function (hash code)



Hash Function Requirements



Hash function requirements

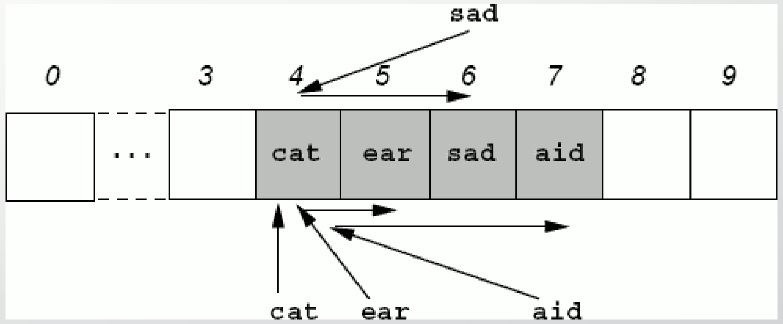
- **A good hash function must be *fast*, i.e. it must run in $O(1)$ time.**
- **A good hash function must distribute keys uniformly over the hash table. Ideally, every location in the hash table should expect to be filled with the same probability as any other location, i.e. the hash function should not favor any one location over another.**

Collision Resolution

- In the **open addressing** method, we call a collision when a key two keys have the same hash value.
- The collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed
- The simplest method is **linear probing**, for which $p(i) = i$, and for the ith probe, the position to be tried is $(h(K) + i)$ mod $TSize$

Linear Probing

- As more and more entries are hashed into the table, they tend to form clusters that get bigger and bigger.
- The number of probes on collisions gradually increases, thus slowing down the hash time to a crawl.
  - Insert "cat", "ear", "sad", and "aid"

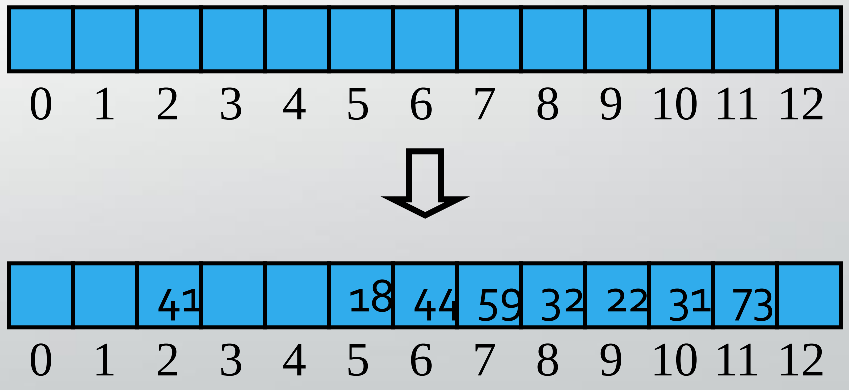


- Clustering is the downfall of linear probing, so we need to look to another method of collision resolution that avoids clustering.

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- **Example:**
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Put with Linear Probing

- put($\boldsymbol{k}, \boldsymbol{o}$)
  - We throw an exception if the table is full
  - We start at cell $\boldsymbol{h}(\boldsymbol{k})$
  - We probe consecutive cells until one of the following occurs
    - A cell $\boldsymbol{i}$ is found that is either empty, or
    - $\boldsymbol{N}$ cells have been unsuccessfully probed
  - We store ($\boldsymbol{k}, \boldsymbol{o}$) in cell $\boldsymbol{i}$

Removes with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***DEFUNCT***, which replaces deleted elements
- remove(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k, o***) is found, we replace it with the special item ***DEFUNCT*** and we return element ***o***
  - Else, we return ***null***

Search with Linear Probing

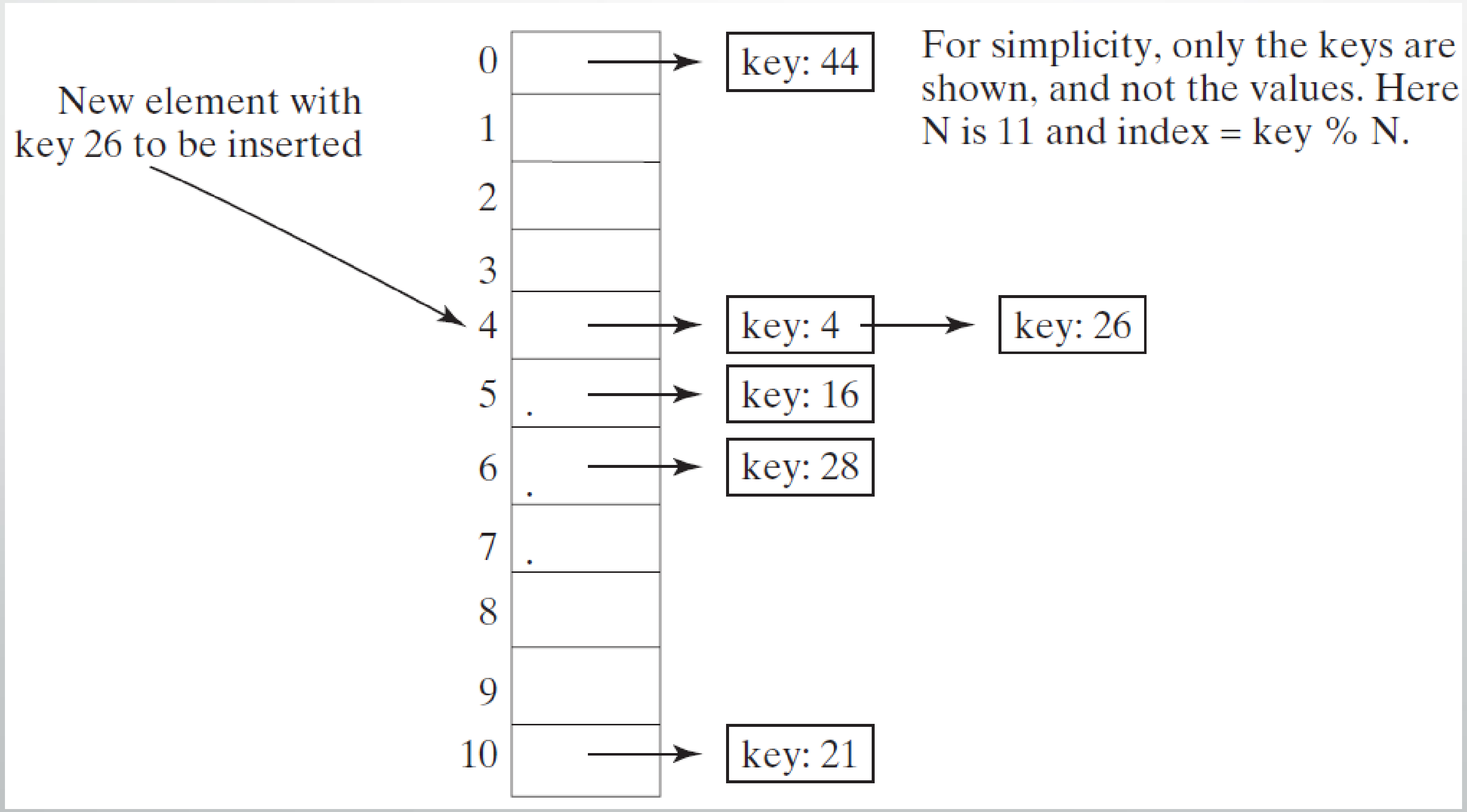
- Consider a hash table $\boldsymbol{A}$ that uses linear probing
- get($\boldsymbol{k}$)
  - We start at cell $\boldsymbol{h}(\boldsymbol{k})$
  - We probe consecutive locations until one of the following occurs
    - An item with key $\boldsymbol{k}$ is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Handling Collisions Using Separate Chaining

- There are other ways of handling collusion in the hash table such as
  - Quadratic Probing
  - Chaining

Handling Collisions Using Separate Chaining

- The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a bucket. A bucket is a container that holds multiple entries.


New element with key 26 to be inserted
0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 26
key: 16
key: 28
key: 21
For simplicity, only the keys are shown, and not the values. Here N is 11 and index = key % N.

The *java.util.HashMap* Class

- Java includes an implementation of a HashMap that you've probably have used before.
- The ***Map*** interface also provides operations to enumerate all the keys, enumerate all the values, get the size of the dictionary, check whether the dictionary is empty, and so on.
- The ***java.util.HashMap*** implements the dictionary abstraction as specified by the ***java.util.Map*** interface. It resolves collisions using chaining.

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $\boldsymbol{O(n)}$ time
- The worst case occurs when all the keys inserted into the map collide
- The expected running time of all the dictionary ADT operations in a hash table is $\boldsymbol{O}(1)$

That’s all folks

- Any questions?