maps & hash tables

Maps

- Designed to efficiently store and retrieve values based on a unique **search key**.
- A map stores *key value* pairs.
    - Called entries
    - k is key
    - v is value
- Keys are unique so that we can use them for *mapping*
- Maps are also known as **associative arrays**
    - Keys are the index into the map.
    - Key can be something other than a number.
        - does not have to be numeric
    - It does not designate a position within the structure.

Maps

- Examples
    - DNS maps a host name to an IP address
    - Websites use username as a key to map it to user information
    - Companies use customer account id
    - A computer graphic system maps a color name to RGB values

Map ADT

- size()
    - Returns the number of entries
- isEmpty()
    - Returns a boolean indicating whether M is empty
- get(k)
    - Returns value **v** associated with key **k**. Returns null if empty.
- put(k,v)
    - If M does not have an entry with key **k**, adds (k,v) to M and returns null
    - Else, replaces **v** with the existing value and returns the old value
- remove(k)
    - Removes from M the entry with key **k**, returns is value. If no entry, null.

Map ADT

- keySet()
  - Returns an iterable connection containing all keys stored in M.
- values()
  - Returns an iterable collection containing all the **values** of entries stored in M.
- entrySet()
  - Returns an iterable collection containing all the key value entries in M.

Method	**Return Value**	**Map**
isEmpty()	true	$\{\}$
put$(5,A)$	null	$\{(5,A)\}$
put$(7,B)$	null	$\{(5,A),(7,B)\}$
put$(2,C)$	null	$\{(5,A),(7,B),(2,C)\}$
put$(8,D)$	null	$\{(5,A),(7,B),(2,C),(8,D)\}$
put$(2,E)$	C	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(7)	B	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(4)	null	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(2)	E	$\{(5,A),(7,B),(2,E),(8,D)\}$
size()	4	$\{(5,A),(7,B),(2,E),(8,D)\}$
remove(5)	A	$\{(7,B),(2,E),(8,D)\}$
remove(2)	E	$\{(7,B),(8,D)\}$
get(2)	null	$\{(7,B),(8,D)\}$
remove(2)	null	$\{(7,B),(8,D)\}$
isEmpty()	false	$\{(7,B),(8,D)\}$
entrySet()	$\{(7,B),(8,D)\}$	$\{(7,B),(8,D)\}$
keySet()	$\{7,8\}$	$\{(7,B),(8,D)\}$
values()	$\{B,D\}$	$\{(7,B),(8,D)\}$

Hash table

- A look up table
- When designed well, has nearly O(1) average running time to find or insert.
- A hash table is an array of fixed size containing data items with unique keys, together with a function called hash function
  - Maps keys to indexes in table/array.

Example

- If keys are integers and hash table is an array of size 127 then;
  - hash(key) = key % 127
- This **maps** numbers to their modulus in the finite field of size 127.
  - For each key there is only one possible value of *hash(key)*
  - Multiple keys may have the same hash.
    - For example 10, 137 and 264 all map to same array location because:
    - 10 % 127 = 137 % 127 = 264 % 127 = 10
- A hash table is a very general structure:
  - It is a table H containing a collection of (key, value) pairs with the property that H may be indexed by the *key* itself.
  - We reference an element of an array *A* by writing *A[i]* using an integer index i.
  - With a hash table, we replace **i** by the **key** contained in location **i**.

- H contains the set of pairs
- We call it by
  - H[“Italy”]
  - It will print Rome.
- As long as we know the **key** associated with data item we can access it in the table in O(1) time.
- When we want to add something, we determine the key and add it.

```
("Italy", "Rome")
("Japan", "Nagano")
("Canada", "Banff")
("France", "Paris")
("Belgium", "Bruges")
("Hungary", "Budapest")
("Portugal", "Porto")
```

Properties of a good hash function

- Supposed to chop up its argument and construct a value out of the chopped up little pieces.
- Good hash fn make the original key hard to reconstruct from the computed hash.
- To be good, it should
  - be easy to compute (for speed)
  - repeatable
  - randomly disperse keys evenly throughout the table
    - making sure that no two keys map to the same index.

- Easy to compute
  - Function is an o(1) operation.
  - Independent of the input size and hash table size.
  - If the function tried to find all the prime factors of a given number in order to compute the hash function, it won’t be easy to compute.
- Repeatable
  - Making two calls to a hash function with the same argument should give the same output.
  - Would make it easy to compute values quickly.
- Dispersion
  - There is as much distance between successive pairs of keys as possible.
  - If hash table is of size 1000 and there are 200 keys in it, they should each be about five addresses apart from their neighbors.

- In principle, if the set of keys is finite and known in advance, one can construct a perfect hash function.
    - Where every key is mapped to unique index.
- However, in practice the set of keys is rarely known at the time of writing the program and may not be finite.

Example

- Lets say we have integer keys: 112, 46, 75, 515
- We want a function to map them to numbers 0,1,2,3.
- Suppose that *hash(key)* is a function that returns the sum of the decimal digits in the key and if that sum has more than one digit itself, we add them together again.
- hash(112) = 1 + 1 + 2 = 4
- hash(46) = hash(4+6) = hash(10) = 1
- hash(75) = h(7+5) = h(12) = 3
- hash(515) = h(11) = 2
- This here is a perfect hash function for these keys.
    - But very poor for larger number of keys.

Why?

- Because it matches all keys to one digit indexes of an array.
    - What if we have 250 keys?
- Completely ignores information about position of digits in the key so 155, 515, 551 will be assigned to same index.

Good hash functions

- Depends on what it is used for and what is the size of the table.
- Java has a method called hashCode() for all its classes that can be used as keys into hash tables.
-

Collision resolution

- What to do when two keys are hashed to the same location in the hash table?
  - Called a **collision**
- Open addressing (closed hashing)
  - Finds an alternative location for the k,v pair, if the first location is occupied
- Closed addressing (open hashing)
  - Allows multiple k,v pairs to be stored in a single array location.

open addressing (probing)

- All data elements are in the hash table.
- When collision occurs, algorithm probes(looks for) for an alternative empty slot within the table to place the element.
- Generally simpler to implement
  - No need for a separate data structure
  - Can handle dynamic changes in the table.
- Performance can degrade as the table fills up
  - Since it is probing, it will lead to longer search times
- Clustering can occur where collisions create *clumps* of elements in the table.
  - Decrease performance.

closed addressing (chaining)

- Each array entry corresponds to a **bucket** containing a mutable set of elements.
  - Typically, implemented as a linked list. So each array entry contains a pointer to the head of the linked list.
- To check whether an element is in hash table, key is first hashed to find the correct bucket to look in.
- Then the linked list is scanned to see if the desired element is present.
  - If the list is short, it will be quick.
- An element is added or removed by hashing it to find the correct bucket.
- Bucket is checked to see if the element is there, and finally it is added or removed.

Bucket array

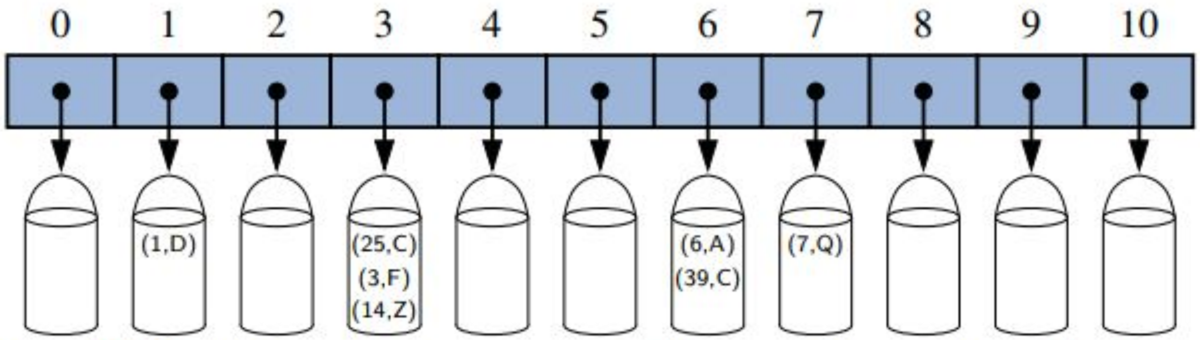

Figure 10.4: A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

Example

- Suppose that we have a hash table with 5 buckets (indexes 0 to 4)
- We will use a very simple hash function
  - hash(key) = length(key) % 5
- Each bucket will contain a linkedlist to handle collisions.
- Our k,v pairs are
  - (apple, 5)
  - (banana, 2)
  - (grape, 1)
  - (orange, 4)
  - (pear, 3)

- Lets apply the hash functions
    - len(apple) = 5 → 5%5 = 0. So it goes to bucket 0.
    - len(banana) = 6 → 6%5 = 1. It goes to bucket 1.
    - grape to 0
    - orange to 1
    - pear to 4.
- Our hash table is
    - bucket 0: linked list with (apple, 5) and (grape, 1)
    - bucket 1: LL with (banana,2) and (orange, 4)
    - bucket 2,3 : empty
    - bucket 4: linked list with entry (pear,3)

- Lets say we want to find the value for grape
- We apply hash function
  - len(grape) = 5. 5%5 = 0. Means look in bucket 0
  - In bucket 0, we have a linked list.
  - Traverse through the linked list. Find grape. Get the value.

Java

- We can use **HashMap** or **HashTable**
- HashMap is more modern and preferred.
  - Uses a different hashing algorithm leading to potentially faster lookup
  - Allows null keys
  - Not synchronized by default

```java
import java.util.HashMap;

// Create an empty hash map
HashMap<String, String> countries = new HashMap<>();

// Add key-value pairs
countries.put("US", "United States");
countries.put("UK", "United Kingdom");
countries.put("FR", "France");

// Get the value for a key
String franceName = countries.get("FR");

// Check if a key exists
boolean hasIndia = countries.containsKey("IN");

// Iterate over the key-value pairs
for (String key : countries.keySet()) {
  String value = countries.get(key);
  System.out.println(key + ": " + value);
}
```