ICS 240
Introduction to Data Structures

Jessica Maistrovich

Metropolitan State University

Maps

New ADT

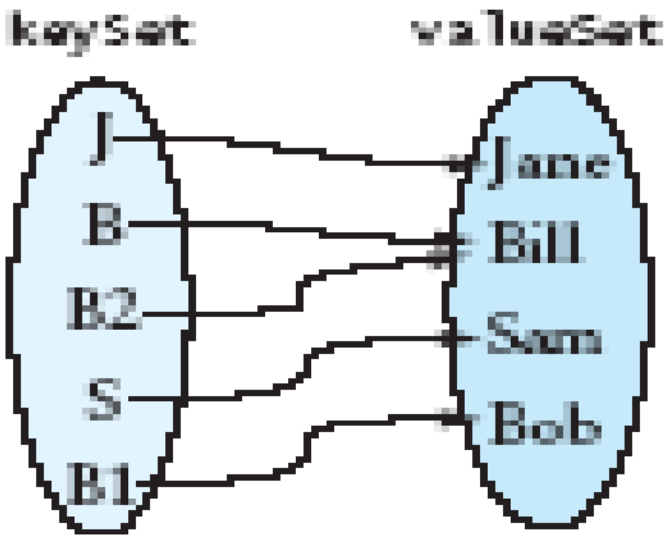
The `Set` Abstraction

- A ***set*** is a collection containing no duplicate elements
- Operations on sets include:
  - Testing for membership
  - Adding elements
  - Removing elements
  - Union
  - Intersection
  - Difference
  - Subset

Map Abstract Data Type

- A `Map` is a **set** of ordered pairs
- Ordered pair: (*key*, *value*)
  - there are no duplicate *keys*
  - values may appear more than once

FIGURE 9.2
Example of Mapping



- A `key` is basically a "mapping to" a particular `value`
- Maps support efficient organization of information in tables

Map ADT (continued)

- Maps are useful in situations where a `key` can be viewed as a unique identifier for the object
- The `key` is used to decide where to store the object in the structure
- In other words, the key associated with an object can be viewed as the **address** for the object
- Maps provide an alternative (more efficient) approach to searching

Why Do We Need the `Map` Data Structure?

- Assume you want to implement a language **dictionary** to store words and their definition. There are two main operations to be supported by the dictionary:
  - `insert:` insert words to the dictionary, and
  - `search:` retrieve the definition given a word
- The `Map` data structure can be used to store <word, definition of word> pairs where
  - `key` = word (note: words are unique)
  - `value` = definition of word
  - `get(word):` returns definition if word is in dictionary and returns null if word is not in dictionary
- What is the time needed to `insert` and `search` if the dictionary is implemented using any of the following data structures?
  - array □ `O(n)`
  - linked list □ `O(n)`
  - binary search tree □ `O(log n)` if balanced and `O(n)` otherwise.

Map Methods

- `size()`
- `isEmpty()`
- `get(k)`: “search” for a key k
  - if M contains an entry with key k, return it; else return null
- `put(k,v)`: insert a key k
  - if M does not have an entry with key k, add entry (k,v) and return null, else replace existing value of entry with v and return the old value
- `remove(k)`: delete a key k
  - remove entry (k,*) from M

Map Example

`(k,v)` key=integer, value=letter

M={}

- `put(5,A)` M={(5,A)} , return null
- `put(7,B)` M={(5,A), (7,B)} , return null
- `put(2,C)` M={(5,A), (7,B), (2,C)} , return null
- `put(8,D)` M={(5,A), (7,B), (2,C), (8,D)}, return null
- `put(2,E)` M={(5,A), (7,B), (2,E), (8,D)}, return C
- `get(4)` return null
- `get(7)` return B
- `get(2)` return E
- `remove(5)` M={(7,B), (2,E), (8,D)}
- `remove(2)` M={(7,B), (8,D)}
- `get(2)` return null

Implementations

A Linked List Implementation of `Map`

- Store the (k,v) pairs in a linked list
  - Instance variables for a map node:
    - key
    - value
    - next
- `get(k)`:
  - hop through the list until finding the element with key k □ O(N)
- `put(k,v)`:
  - hop through the list until finding the node with key k □ O(N), store it in targetNode
  - if (targetNode != null), replace the value in targetNode with v
  - else create a new node(k,v) and add it at the front
- `remove(k)`:
  - hop through the list until finding the node with key k □ O(N), store it in targetNode
  - if (targetNode != null), remove node from list

- **Analysis:** insert, search, and delete require `O(n)` on a map with `n` elements

Map Implementations

- Linked-list:
  - search, insert, remove: $O(n)$
- Binary search trees:
  - search, insert, delete:
    - $O(n)$ if not balanced
    - $O(\log n)$ if balanced
- **Hash tables:**
  - search, insert, delete can be done in $O(1)$ – (under some assumptions)

Hashing

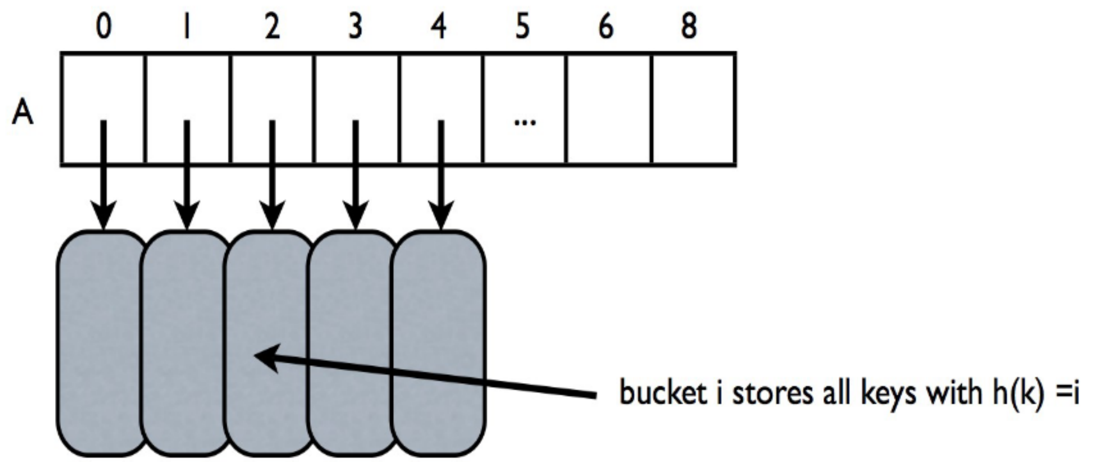
Hashing = Transforming a Key to an Integer

- Hashing is a completely different approach to searching from the comparison-based methods (binary search, binary search trees).
- Rather than navigating through a data structure comparing the search key with the elements, hashing tries to reference an element in a table directly based on its key.
- Hashing transforms a key into a table **address**.

Hashing: Direct Addressing

- Simplest approach
- Assume keys are integers in the range 0 to 99
- Values are stored in array A, where:
  - `A` initially empty
  - `put(key, value)`: stores `value` in `A[key]` ▭ `O(1)`
  - `get(key)`: returns `A[key]` ▭ `O(1)`
- Issues:
  - Keys need to be integers in a small range
  - Space may be wasted if A not full

Indirect Addressing



- Hashing has 2 components:
  - **The hash table - A:**
    - an array A of size N
    - each entry in the array is a bucket (a bucket array)
  - **The hash function – h ( ) :**
    - A function that maps each key to a bucket
    - h ( ) is a function : {all possible keys} ----> {0, 1, 2, ..., N-1}
    - key k is stored in bucket number h (k)
- The size of the table (N) and the hash function (h ( )) are decided by the user

Example

0	1	2	3	4	5	6	7	8	9
F		A	B		C			D	

- Keys: any integers
- Buckets of size 1
- `N = 10`
- `h(k) = k % 10`
  - [k % 10 is the remainder of k/10]

- add (2,A), (13,B), (15,C), (88,D), (2345,E), (100,F)

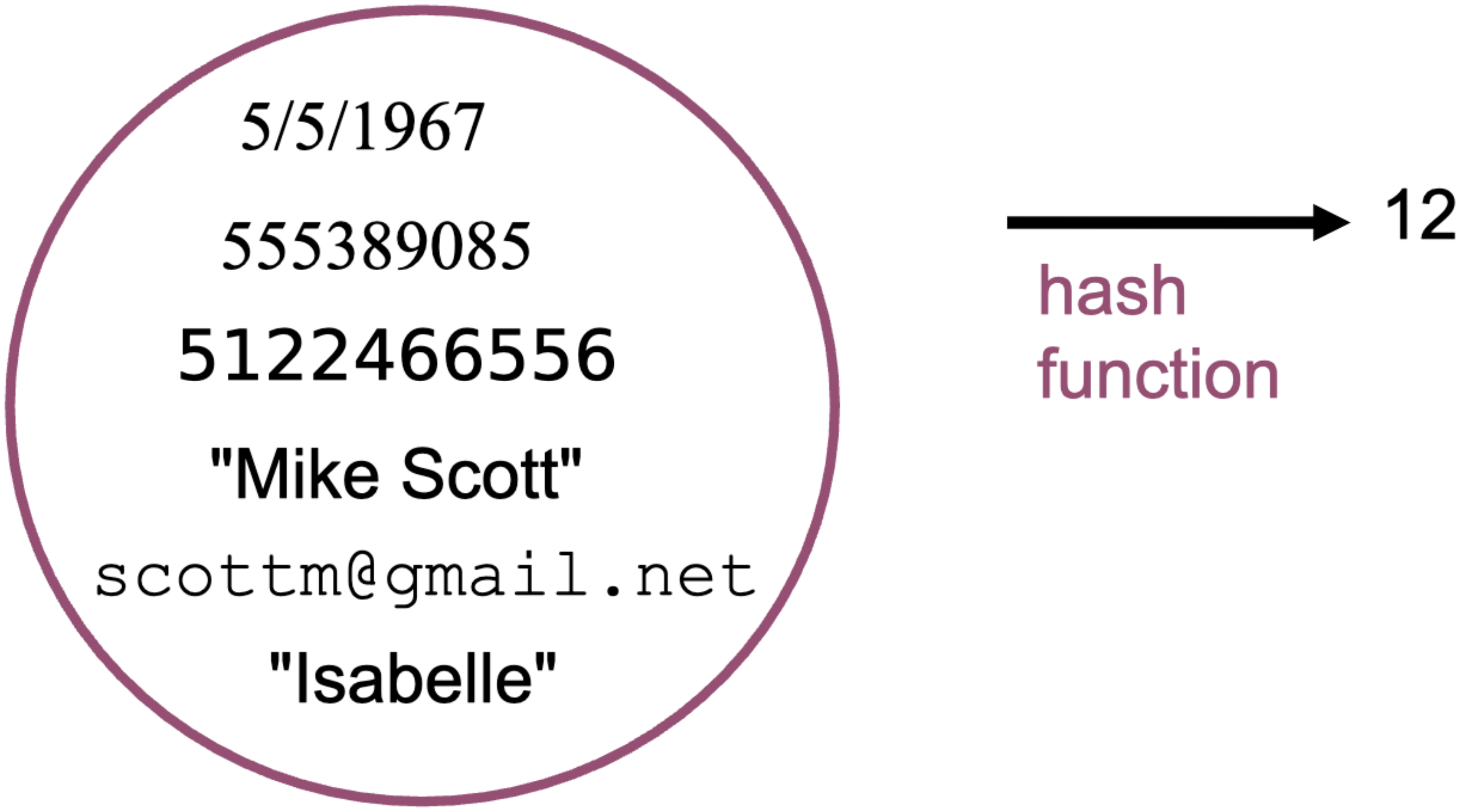
- **Collision:**
  - two keys that hash to the same value
  - e.g. 15, 2345 hash to slot 5
- Note: It is not feasible to have an array with one slot for each integer value

Hash functions

SHA-256

https://xorbin.com/tools/sha256-hash-calculator

Hash Functions Can Be Applied to Any Object



Notes about Hash Functions

- h( ) : {universe of all possible keys} ----> {0,1,2,…,N-1}
  - The universe of all possible keys need not be small
- The keys need not be integers
  - If keys are Strings then the hash function is defined to map strings to integers
  - Every class has a `hashCode()` method
    - Either implemented explicitly or inherited from Object
    - Returns an int
  - Example:

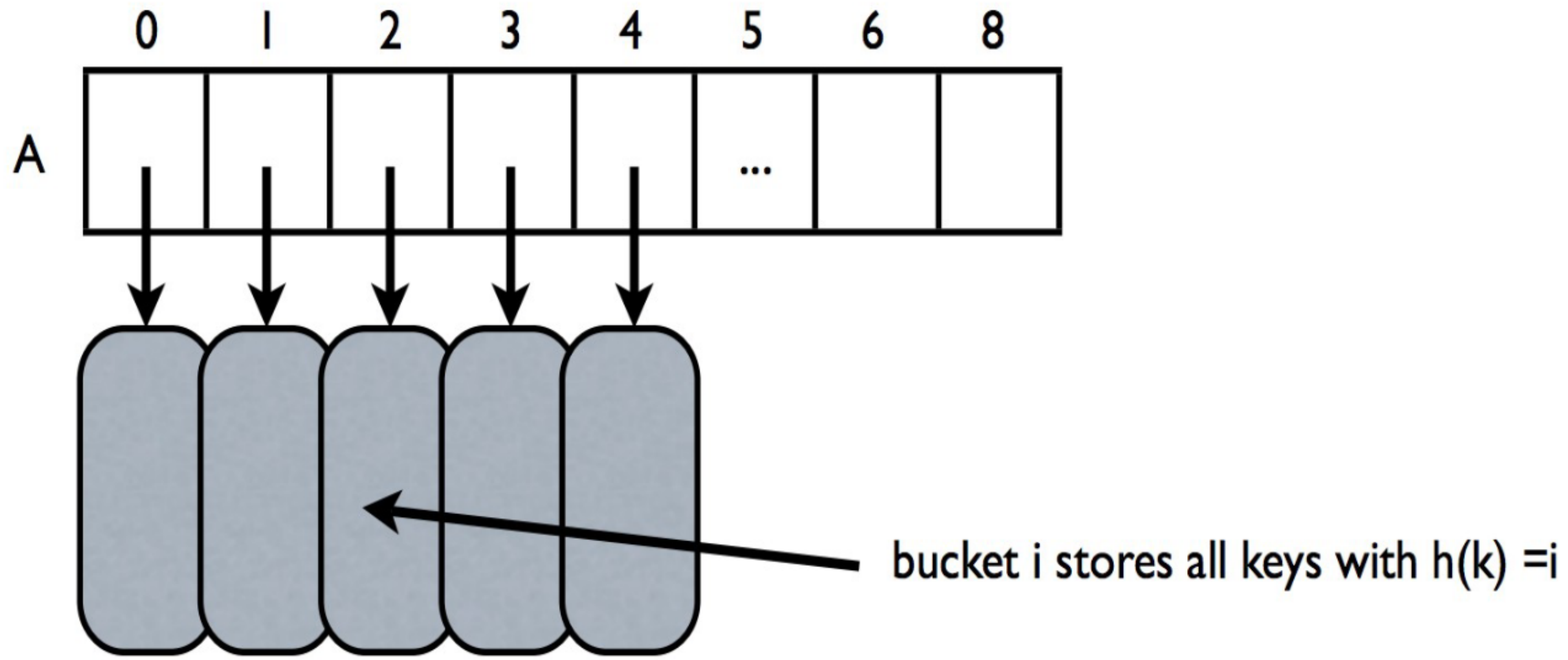
```
String str = "Hello";
System.out.println(str.hashCode());
```

- Hashing supports insert, delete and search in $O(1)$ time, with some assumptions

Choosing `h()` and `N`

- Notation:
  - `U` = universe of keys
  - `N` = hash table size
  - `n` = number of entries (i.e., data size)
  - Note that `n` may be unknown beforehand
- Goal of a hash function:
  - The probability of any two keys hashing to the same slot is `1/N`
  - Hash function throws the keys uniformly at random into the table
  - Load factor: If a hash function satisfies the uniform hashing property, then the expected number of elements that hash to the same entry is `n/N`
    - if n < N : `load factor <= 1` element per entry
    - if n >= N: `load factor ~= n/N` elements per entry

Load Factor > 1



What makes a good hash function?

- An ideal hash function approximates a random function:
  - For each input element, every output should be in some sense equally likely
  - Impossible to guarantee
- Every hash function has a worst-case scenario where all elements map to the same entry
- However, there exists a set of good heuristics that can be followed to choose a good hashing function

Devising Hash Functions

- Simple functions often produce many collisions
  - But complex functions may not be good either!
- It is often a trial and error process
  - Adding letter values in a string:

    ```
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
            hash = hash + s.charAt(i);
    ```

    - Same hash for strings with same letters in different order
  - Better approach (the hash function used for `String` in Java):

    ```
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
            hash = hash * 31 + s.charAt(i);
    ```

Devising Hash Functions (2)

- The **`String`** hash is good in that:
  - Every letter affects the value
  - The order of the letters affects the value
  - The values tend to be spread well over the set of integers
- Inserting strings in an array:
  - Calculate index: **`int index = hash % size;`**

Hash Table Example

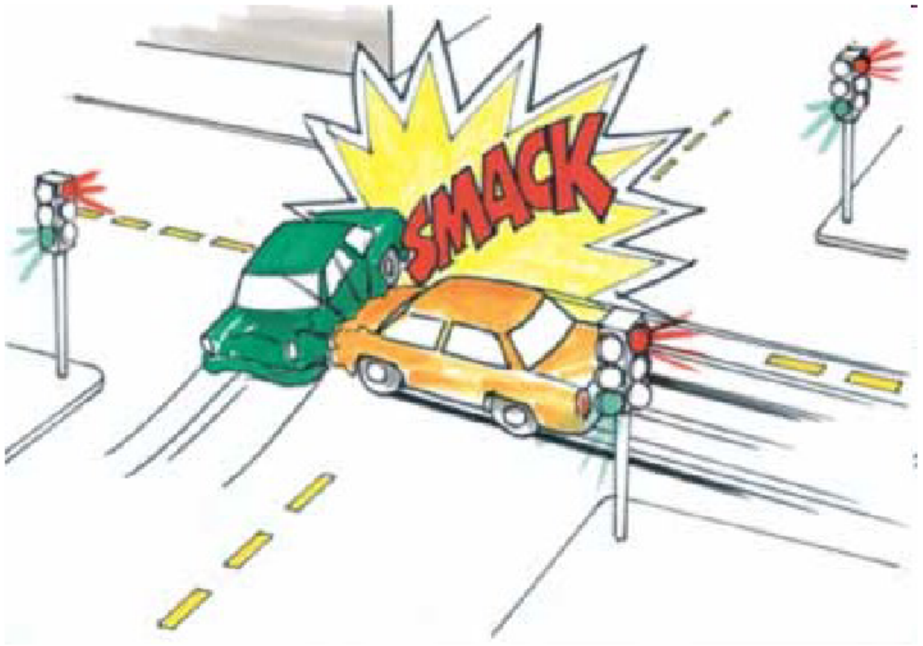
- Table of strings, initial size 5
  - Add “Tom”, hash 84274 → **4**
  - Add “Dick”, hash 2129869 → **4**
  - Add “Harry”, hash 69496448 → 3
  - Add “Sam”, hash 82879 → **4**
  - Add “Pete”, hash 2484038 → 3

- If table size is 11:
  - Add “Tom”, hash 84274 → 3
  - Add “Dick”, hash 2129869 → **5**
  - Add “Harry”, hash 69496448 → 10
  - Add “Sam”, hash 82879 → **5**
  - Add “Pete”, hash 2484038 → 7

Collisions

Handling Collisions

- What to do when inserting an element and something is already present?



Collision Resolution Strategies

- Open addressing - find the “next” available slot
  - Linear probing – h(key) + 1
  - Quadratic probing – $h(key)+1^2$, $h(key)+2^2$, $h(key)+3^2$, etc
- Chaining
  - Put all elements that hash to the same location in a linked list (the buckets)
- Double Hashing
  - Hash a second time
  - h (h (key))