Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

MAPS

- A map is an abstract data type designed to efficiently store and retrieve values based upon a uniquely identifying search key for each
- Specifically, a map stores key-value pairs (k,v), which we call entries, where k is the key and v is its corresponding value
- Keys are required to be unique, so that the association of keys to values defines a mapping
- Maps are also known as associative arrays, because the entry's key serves somewhat like an index into the map, in that it assists the map in efficiently locating the associated entry

MAP Applications

- Common applications of maps include the following:
  - A university's information system relies on some form of a student ID as a key that is mapped to that student's associated record (such as the student's name, address, and course grades) serving as the value.
  - The domain-name system (DNS) maps a host name, such as www.wiley.com, to an Internet-Protocol (IP) address, such as 208.215.179.146.
  - A social media site typically relies on a (nonnumeric) username as a key that can be efficiently mapped to a particular user's associated information.
  - A company's customer base may be stored as a map, with a customer's account number or unique user ID as a key, and a record with the customer's information as a value. The map would allow a service representative to quickly access a customer's record, given the key.
  - A computer graphics system may map a color name, such as 'turquoise', to the triple of numbers that describes the color's RGB (red-green-blue) representation, such as (64, 224, 208).

The Map ADT

- Since a map stores a collection of objects, it should be viewed as a collection of key-value pairs
- As an ADT, a map M supports the following methods:
  - **size()**: Returns the number of entries in M.
  - **isEmpty()**: Returns a boolean indicating whether M is empty.
  - **get(k)**: Returns the value v associated with key k, if such an entry exists; otherwise returns null.
  - **put(k, v)**: If M does not have an entry with key equal to k, then adds entry (k,v) to M and returns null; else, replaces with v the existing value of the entry with key equal to k and returns the old value.
  - **remove(k)**: Removes from M the entry with key equal to k, and returns its value; if M has no such entry, then returns null.
  - **keySet()**: Returns an iterable collection containing all the keys stored inM.
  - **values()**: Returns an iterable collection containing all the values of entries stored in M (with repetition if multiple keys map to the same value).
  - **entrySet()**: Returns an iterable collection containing all the key-value entries in M.

MAP Operations

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()	$\{B,D\}$	$\{(7,B),(8,D)\}$

A Java Interface for the Map ADT

- A formal definition of a Java interface for our version of the map ADT
- It uses the generics framework with K designating the key type and V designating the value type

```
public interface Map<K,V> {
  int size( );
  boolean isEmpty( );
  V get(K key);
  V put(K key, V value);
  V remove(K key);
  Iterable<K> keySet( );
  Iterable<V> values( );
  Iterable<Entry<K,V>> entrySet( );
}
```

Abstract Map base class-1

```
public abstract class AbstractMap<K,V> implements Map<K,V> {
  public boolean isEmpty( ) { return size( ) == 0; }
  //---------------- nested MapEntry class ----------------
  protected static class MapEntry<K,V> implements Entry<K,V> {
    private K k;    // key
    private V v;    // value
    public MapEntry(K key, V value) {
      k = key;
      v = value;
    }
    // public methods of the Entry interface
    public K getKey( ) { return k; }
    public V getValue( ) { return v; }
    // utilities not exposed as part of the Entry interface
    protected void setKey(K key) { k = key; }
    protected V setValue(V value) {
      V old = v;
      v = value;
      return old;
    }
  } //---------- end of nested MapEntry class ----------
```

Abstract Map base class-2

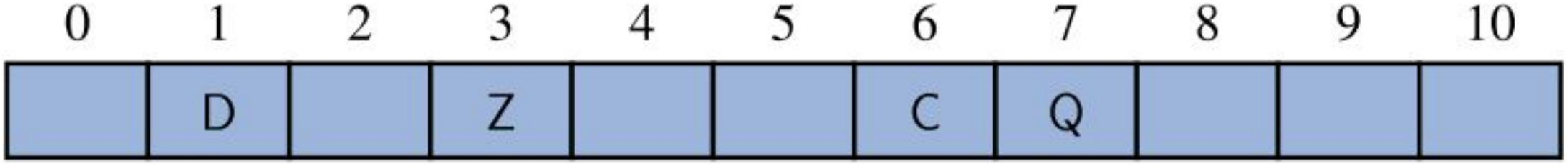
```
// Support for public keySet method...
private class KeyIterator implements Iterator<K> {
   private Iterator<Entry<K,V>> entries = entrySet().iterator(); // reuse entrySet
   public boolean hasNext() { return entries.hasNext(); }
   public K next() { return entries.next().getKey(); }                // return key!
   public void remove() { throw new UnsupportedOperationException(); }
}
private class KeyIterable implements Iterable<K> {
   public Iterator<K> iterator() { return new KeyIterator(); }
}
public Iterable<K> keySet() { return new KeyIterable(); }
```

Abstract Map base class-3

```
  // Support for public values method...
  private class ValueIterator implements Iterator<V> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator(); // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public V next() { return entries.next().getValue(); }          // return value!
    public void remove() { throw new UnsupportedOperationException(); }
  }
  private class ValueIterable implements Iterable<V> {
    public Iterator<V> iterator() { return new ValueIterator(); }
  }
  public Iterable<V> values() { return new ValueIterable(); }
}
```

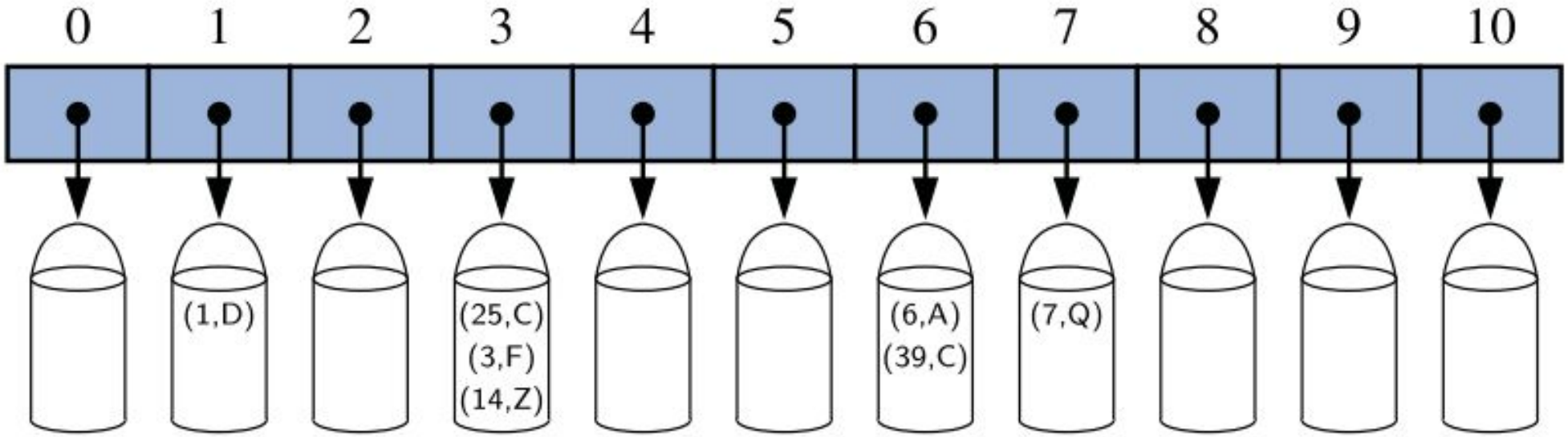
Hash Tables

- A hash table is a data structure that offers very fast insertion and searching
- Hash tables are significantly faster than trees
- Intuitively, a map M supports the abstraction of using keys as “addresses” that help locate an entry
- A map with n entries uses keys that are known to be integers in a range from 0 to N−1 for some N≥ n
- If we represent the map using a lookup table of length N
  - we store the value associated with key k at index k of the table
  - Basic map operations get, put, and remove can be implemented in O(1) worst-case time



Hash Tables

- In hash tables, the novel concept of a hash function is used to map general keys to corresponding indices in a table
- Ideally, keys will be well distributed in the range from 0 to N−1 by a hash function, but in practice there may be two or more distinct keys that get mapped to the same index
- As a result, we will conceptualize our table as a bucket array



Hash Functions

- The goal of a hash function, h, is to map each key k to an integer in the range [0,N−1], where N is the capacity of the bucket array for a hash table
- The main idea of this approach is to use the hash function value, h(k), as an index into our bucket array, A, instead of the key k
- That is, we store the entry (k,v) in the bucket A[h(k)]
- If there are two or more keys with the same hash value, then two different entries will be mapped to the same bucket in A
- In this case, we say that a collision has occurred
- A hash function is “good” if it maps the keys in our map so as to sufficiently minimize collisions

Hash Functions

- Generaly, a hash function, h(k), as consisting of two portions—a hash code that maps a key k to an integer, and a compression function that maps the hash code to an integer within a range of indices, [0,N−1], for a bucket array
- The advantage of separating the hash function into two such components is that the hash code portion of that computation is independent of a specific hash table size; only the compression function depends upon the table size
- This is particularly convenient, because the underlying bucket array for a hash table may be dynamically resized, depending on the number of entries currently stored in the map

Hash Codes

- The first action that a hash function performs is to take an arbitrary key k in our map and compute an integer that is called the hash code for k; this integer need not be in the range [0,N−1], and may even be negative
- Practical implementations of hash codes are:
  - **Treating the Bit Representation as an Integer:** For any data type X that is represented using at most as many bits as our integer hash codes, we can simply take as a hash code for X an integer interpretation of its bits
  - **Polynomial Hash Codes:** this is simply a polynomial in a that takes the components (x0,x1, . . . ,xn−1) of an object x as its coefficients. This hash code is therefore called a polynomial hash code
  - **Cyclic-Shift Hash Codes:** A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits

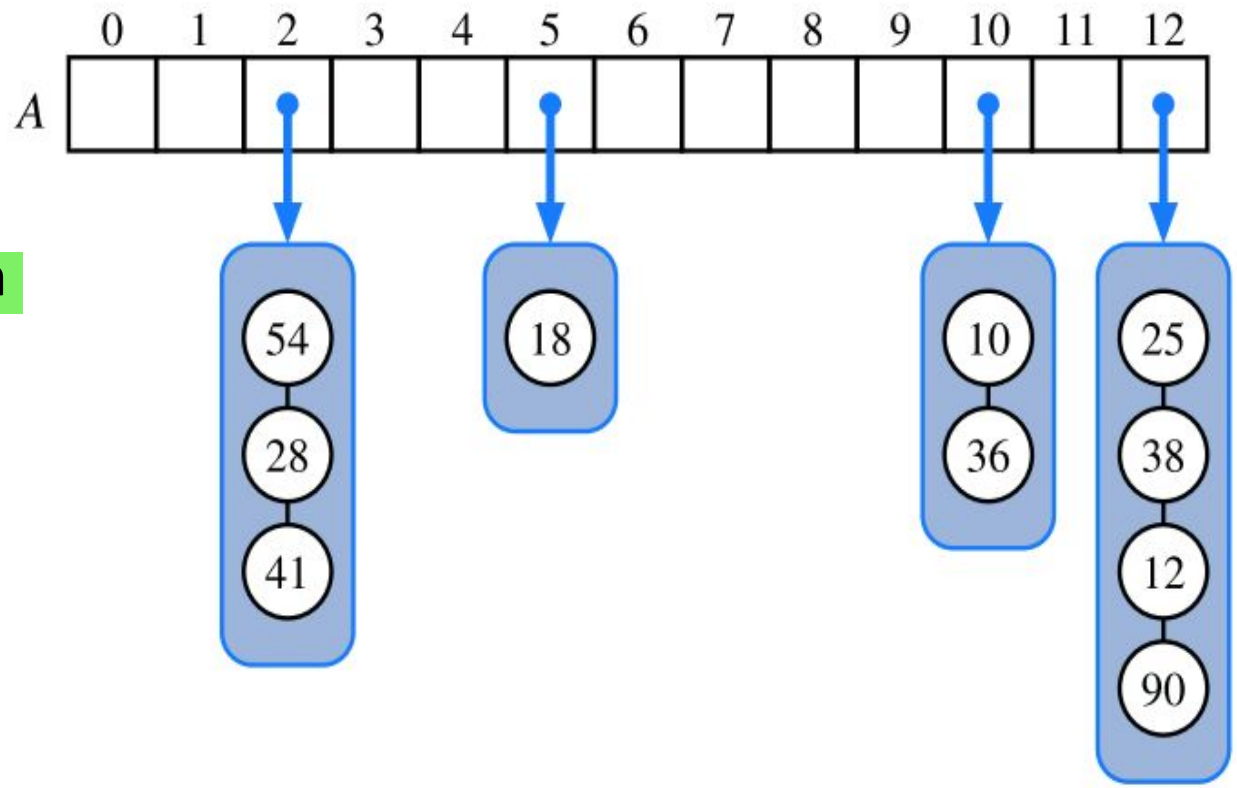
Compression Functions

- The hash code for a key k will typically not be suitable for immediate use with a bucket array, because the integer hash code may be negative or may exceed the capacity of the bucket array
- Thus, once we have determined an integer hash code for a key object k, then compression functions maps that integer into the range [0,N−1]
  - **The Division Method:** A simple compression function is the division method, which maps an integer i to $i \bmod N$, where N, the size of the bucket array, is a fixed positive integer
  - **The MAD Method:** It eliminates repeated patterns in a set of integer keys. This method maps an integer i to $[(ai+b) \bmod p] \bmod N$, where N is the size of the bucket array, p is a prime number larger than N, and a and b are integers chosen at random from the interval [0, p−1], with a > 0

Collision-Handling Schemes

- The main idea of a hash table is to take a bucket array, A, and a hash function, h, and use them to implement a map by storing each entry (k,v) in the "bucket" A[h(k)]
- When we have two distinct keys, k1 and k2, such that h(k1) = h(k2) then ***collisions*** prevents us from simply inserting a new entry (k,v) directly into the bucket A[h(k)]
- It also complicates our procedure for performing insertion, search, and deletion operations
- There are different ways to address collision
  - Separate Chaining
  - Open Addressing

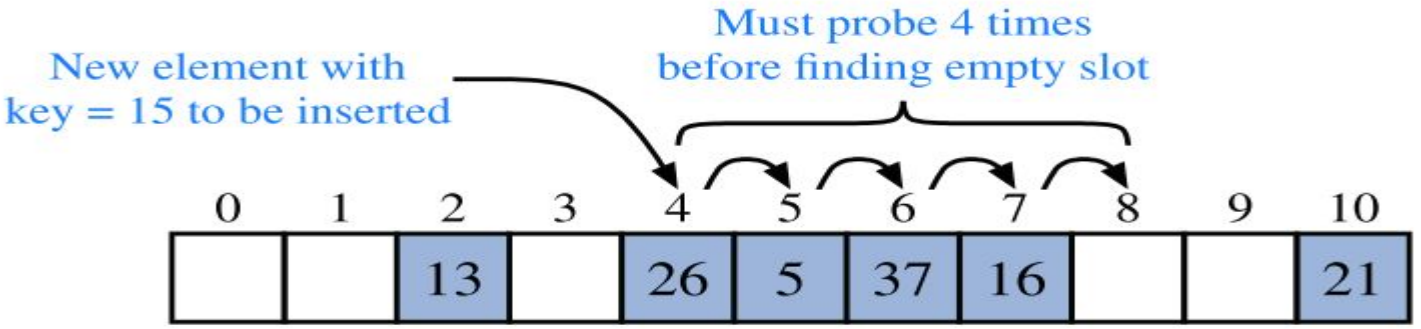
Separate Chaining



- A simple and efficient way for dealing with collisions is to have each bucket A[j] store its own secondary container, holding all entries (k,v) such that h(k) = j
- A natural choice for the secondary container is a small map instance implemented using an unordered list
- This collision resolution rule is known as separate chaining

- In the worst case, operations on an individual bucket take time proportional to the size of the bucket
- for n entries of our map in a bucket array of capacity N, the expected size of a bucket is n/N
- The ratio λ = n/N, called the load factor of the hash table, should be bounded by a small constant, preferably below 1
- As long as λ is O(1), the core operations on the hash table run in O(1) expected time

Open Addressing



- The separate chaining requires the use of an auxiliary data structure to hold entries with colliding keys
- If space is at a premium, then we can use the alternative approach of storing each entry directly in a table slot, known as ***Open Addressing***
- This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to properly handle collisions
- Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself
- A simple method for collision handling with open addressing is linear probing
- With this approach, ifwe try to insert an entry (k,v) into a bucket A[j] that is already occupied, where j = h(k), then we next try A[(j +1) mod N].
- If A[(j +1) mod N] is also occupied, then we try A[(j +2) mod N], and so on, until we find an empty bucket that can accept the new entry
- Once this bucket is located, we simply insert the entry there
- In particular, to attempt to locate an entry with key equal to k, we must examine consecutive slots, starting from A[h(k)], until we either find an entry with an equal key or we find an empty bucket