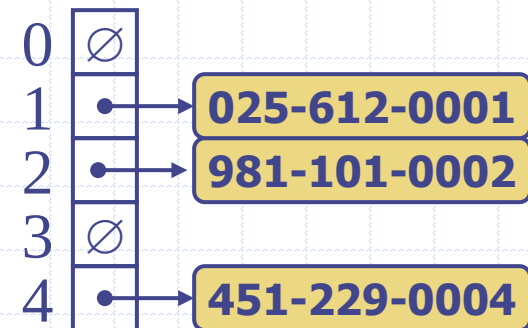


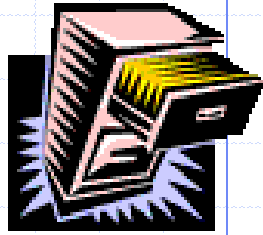
Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Hash Tables

In this section, we introduce one of the most efficient data structures for implementing a map, and the one that is used most in practice.

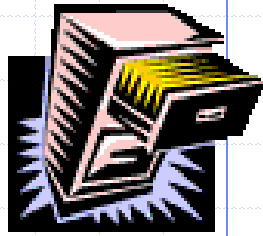
This structure is known as a hash table.





Recall the Map ADT

- ❑ **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

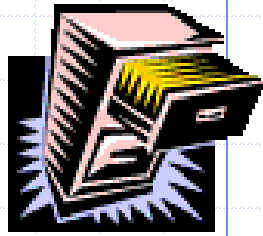


Intuitive Notion of a Map

- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as $M[k]$.
- Consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to $N - 1$, for some $N \geq n$.

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Figure 10.3: A lookup table with length 11 for a map containing entries $(1,D)$, $(3,Z)$, $(6,C)$, and $(7,Q)$.



Intuitive Notion of a Map

- In this representation, 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.

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Figure 10.3: A lookup table with length 11 for a map containing entries $(1,D)$, $(3,Z)$, $(6,C)$, and $(7,Q)$.

Intuitive Notion of a Map

- There are two challenges in extending this framework to the more general setting of a map.
 - First, we may not wish to devote an array of length N if it is the case that $N \gg n$.
 - Second, we do not require that a map's keys be integers.
- 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.
- Each bucket may manage a collection of entries that are sent to a specific index by the hash function.

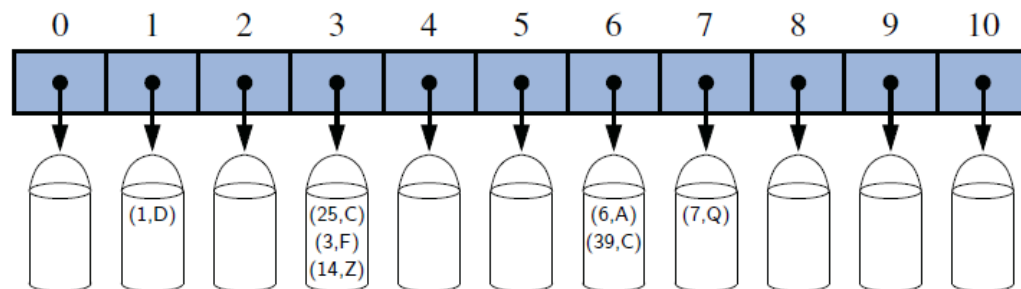
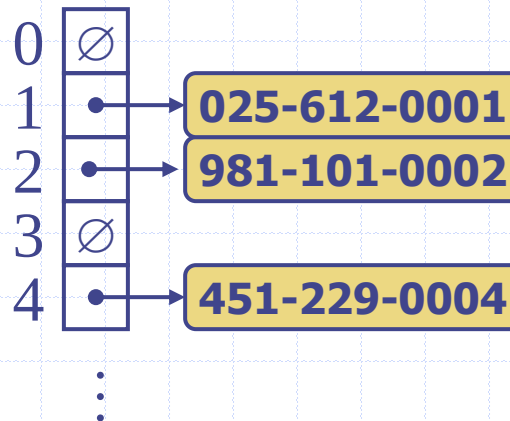


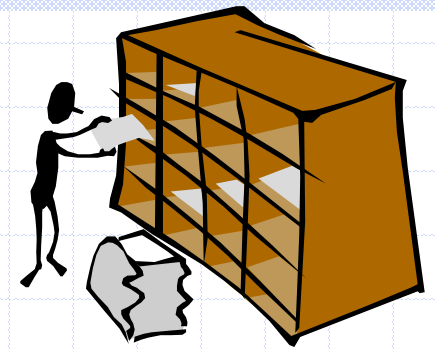
Figure 10.4: A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

More General Kinds of Keys

- 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 keys to corresponding indices in a table.
 - For instance, the last four digits of a Social Security number.



Hash Functions and Hash Tables

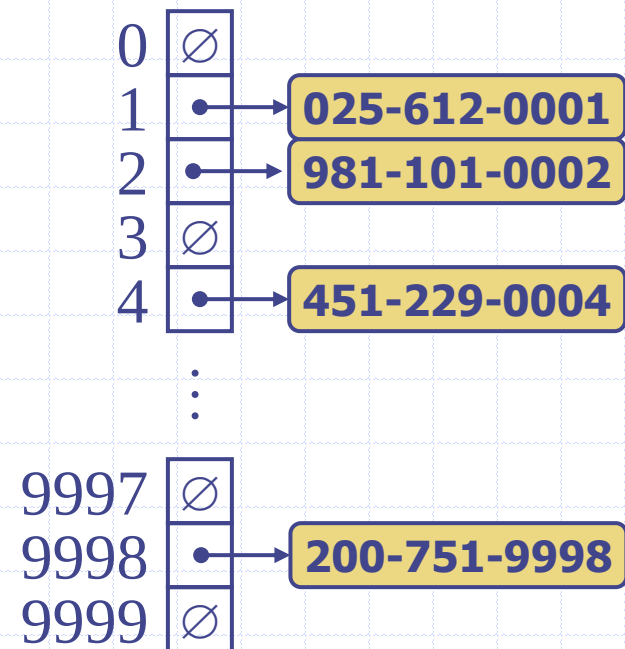


- A **hash function** h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:
$$h(x) = x \bmod N$$

is a hash function for integer keys
- The integer $h(x)$ is called the **hash value** of key x
- A **hash table** for a given key type consists of
 - Hash function h
 - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

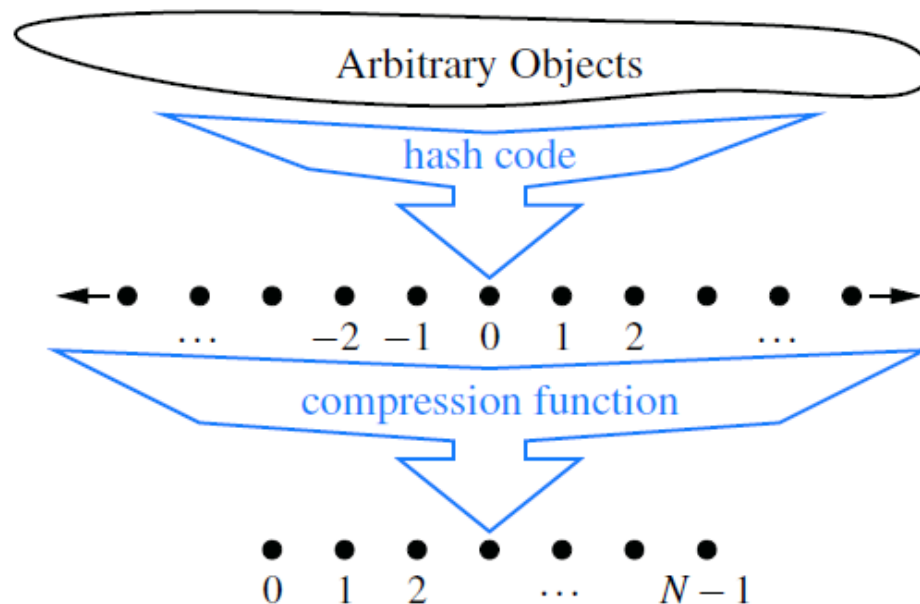
- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10,000$ and the hash function $h(x) = \text{last four digits of } x$



Hash Functions Components

A hash function, $h(k)$, consists 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



Hash Functions- Why Dividing Hash Function into 2 Components

- The advantage of separating the hash function into two components is:
 - The hash code portion is independent of a specific hash table size.
 - Only the compression function depends upon the table size.
- This allows the development of a general hash code for each object that can be used for a hash table of any size.
- Hence, underlying bucket array for a hash table may be dynamically resized, depending on the number of entries currently stored in the map.

Hash Functions



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

Hash code:

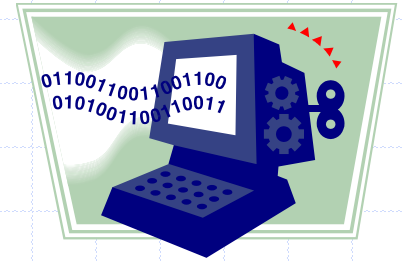
$h_1: \text{keys} \rightarrow \text{integers}$

Compression function:

$h_2: \text{integers} \rightarrow [0, N - 1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,
$$h(x) = h_2(h_1(x))$$
- The goal of the hash function is to “disperse” the keys in an apparently random way

Hash Codes



- ❑ **Memory address:**

- We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
- Good in general, except for numeric and string keys

- ❑ **Integer cast:**

- We reinterpret the bits of the key as an integer
- Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

- ❑ **Component sum:**

- We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
- Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.)

- **Polynomial accumulation:**

- We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

$$a_0 \ a_1 \ \dots \ a_{n-1}$$

- We evaluate the polynomial

$$p(z) = a_0 + a_1 z + a_2 z^2 + \dots \\ \dots + a_{n-1} z^{n-1}$$

at a fixed value z , ignoring overflows

- Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:

- The following polynomials are successively computed, each from the previous one in $O(1)$ time

$$p_0(z) = a_{n-1}$$

$$p_i(z) = a_{n-i-1} + zp_{i-1}(z) \\ (i = 1, 2, \dots, n-1)$$

- We have $p(z) = p_{n-1}(z)$

Cyclic-Shift Hash Codes

A variant of the polynomial hash code replaces multiplication by Z with a cyclic shift of a partial sum by a certain number of bits.

For example, a 5-bit cyclic shift of the 32-bit value:

001111011001011010101000101
01000

is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in:

101100101101010100010101000
00111.

An implementation of a cyclic-shift hash code computation for a character string in Java appears as follows:

```
static int hashCode(String s) {  
    int h=0;  
    for (int i=0; i<s.length( ); i++) {  
        h = (h << 5) | (h >>> 27);  
        // 5-bit cyclic shift of the  
        // running sum  
        h += (int) s.charAt(i);  
        // add in next character  
    }  
    return h;  
}
```

Hash Codes in Java

The notion of hash codes are an integral part of the Java language.

The Object class, which serves as an ancestor of all object types, includes a default `hashCode( )` method that returns a 32-bit integer of type `int`, which serves as an object's hash code.

The default version of `hashCode( )` provided by the Object class is often just an integer representation derived from the object's memory address.

Hash Codes in Java - Example

- To implement hashCode for a user-defined class, SinglyLinkedList.
- We can compute a robust hash code for a list by:
 - Taking the exclusive-or of its elements' hash codes,
 - Then, performing a cyclic shift.

```
1 public int hashCode() {  
2     int h = 0;  
3     for (Node walk=head; walk != null; walk = walk.getNext()) {  
4         h ^= walk.getElement().hashCode(); // bitwise exclusive-or with element's code  
5         h = (h << 5) | (h >>> 27);        // 5-bit cyclic shift of composite code  
6     }  
7     return h;  
8 }
```

Code Fragment 10.6: A robust implementation of the hashCode method for the SinglyLinkedList class from Chapter 3.



Compression Functions

□ Division:

- $h_2(y) = y \bmod N$
- The size N of the hash table is usually chosen to be a prime
- The reason has to do with number theory and is beyond the scope of this course

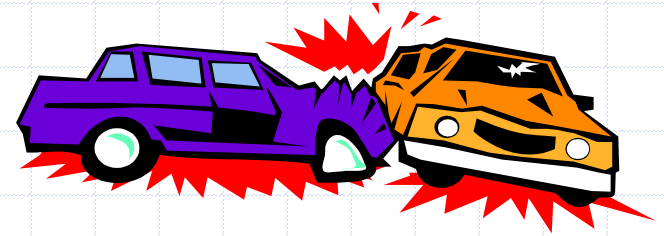
□ Multiply, Add and Divide (MAD):

- $h_2(y) = (ay + b) \bmod N$
- a and b are nonnegative integers such that
$$a \bmod N \neq 0$$
- Otherwise, every integer would map to the same value b

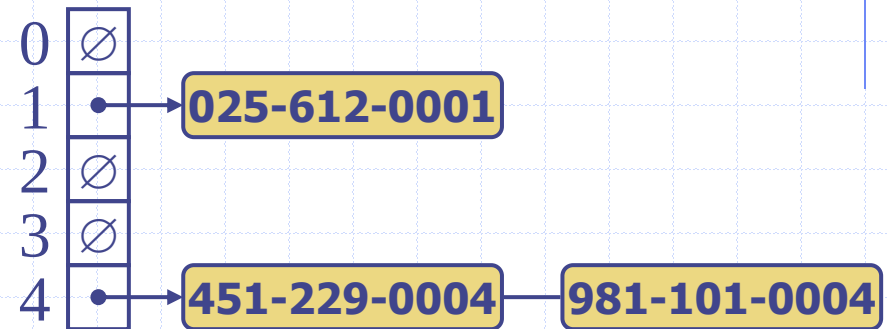
Collision-Handling Schemes

- 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” $A[h(k)]$.
- This simple idea is challenged, when we have two distinct keys, k_1 and k_2 , such that $h(k_1) = h(k_2)$.
- The existence of such 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.

Collision Handling



- ❑ Collisions occur when different elements are mapped to the same cell



- ❑ **Separate Chaining:** let each cell in the table point to a linked list of entries that map there
- ❑ Separate chaining is simple, but requires additional memory outside the table

Collision-Handling Schemes- Separate Chaining

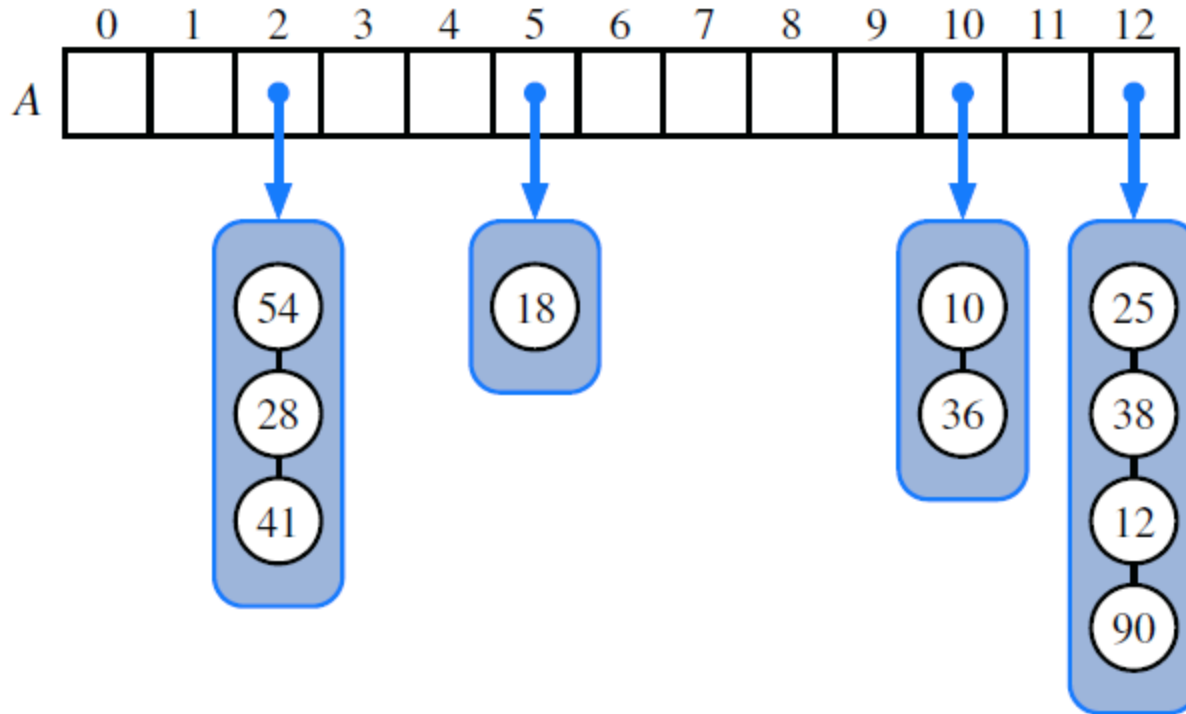


Figure 10.6: A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

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

0	1	2	3	4	5	6	7	8	9	10	11	12

↓

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

- Simple method for collision handling with open addressing.
- With this approach, if we try to insert an entry (k,v) into a bucket $A[j]$ that is already occupied, where $j = h(k)$, then we next try $A[(j+1) \bmod N]$.
- If $A[(j+1) \bmod N]$ is also occupied, then we try $A[(j+2) \bmod N]$, 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.

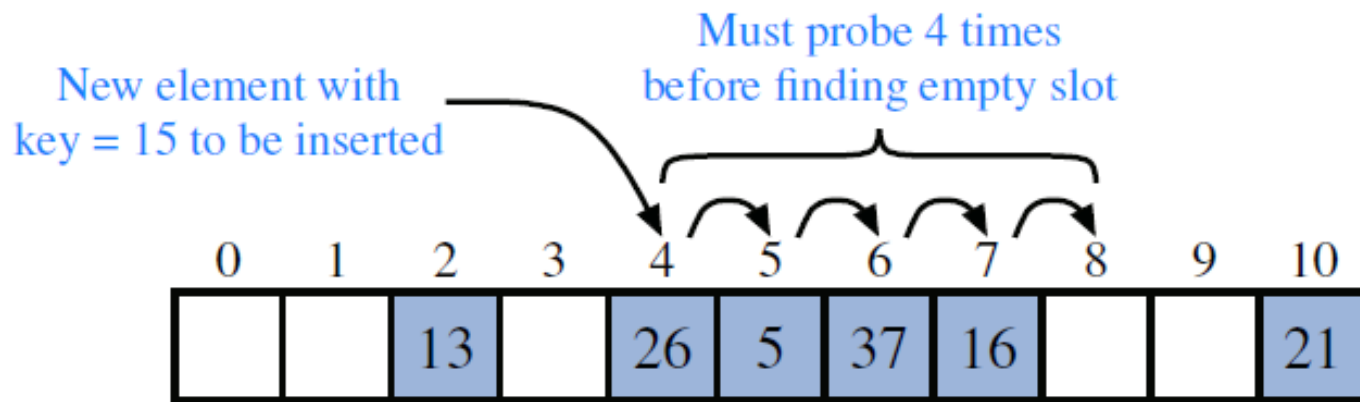


Figure 10.7: Insertion into a hash table with integer keys using linear probing. The hash function is $h(k) = k \bmod 11$. Values associated with keys are not shown.

Linear Probing- Search

This collision resolution strategy requires that we change the implementation when searching for an existing key—the first step of all get, put, or remove operations.

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.

Linear Probing- Delete

To implement a deletion, we cannot simply remove a found entry from its slot in the array.

For example, after the insertion of key 15 portrayed in Figure 10.7, if the entry with key 37 were trivially deleted, a subsequent search for 15 would fail because that search would start by probing at index 4, then index 5, and then index 6, at which an empty cell is found.

A typical way to get around this difficulty is to replace a deleted entry with a special “defunct” sentinel object.

With this special marker possibly occupying spaces in our hash table, we modify our search algorithm so that the search for a key k will skip over cells containing the defunct sentinel and continue probing until reaching the desired entry or an empty bucket (or returning back to where we started from).

Additionally, our algorithm for put should remember a defunct location encountered during the search for k , since this is a valid place to put a new entry (k,v) , if no existing entry is found beyond it.

Java Hash Table Implementation

We develop two implementations of a hash table, one using separate chaining and the other using open addressing with linear probing.

While these approaches to collision resolution are quite different, there are many higher level commonalities to the two hashing algorithms.

For that reason, we extend the `AbstractMap` class (from Code Fragment 10.3) to define a new `AbstractHashMap` class (see Code Fragment 10.7), which provides much of the functionality common to our two hash table implementations.

We will begin by discussing what this abstract class does not do—it does not provide any concrete representation of a table of “buckets.” With separate chaining, each bucket will be a secondary map.

With open addressing, however, there is no tangible container for each bucket; the “buckets” are effectively interleaved due to the probing sequences.

Java Hash Table Implementation

- In our design, the `AbstractHashMap` class presumes the following to be abstract methods—to be implemented by each concrete subclass:

`createTable()`: This method should create an initially empty table having size equal to a designated capacity instance variable.

`bucketGet(h, k)`: This method should mimic the semantics of the public `get` method, but for a key *k* that is known to hash to bucket *h*.

`bucketPut(h, k, v)`: This method should mimic the semantics of the public `put` method, but for a key *k* that is known to hash to bucket *h*.

`bucketRemove(h, k)`: This method should mimic the semantics of the public `remove` method, but for a key *k* known to hash to bucket *h*.

`entrySet()`: This standard map method iterates through *all* entries of the map. We do not delegate this on a per-bucket basis because “buckets” in open addressing are not inherently disjoint.

Java Hash Table Implementation

- What the AbstractHashMap class does provide is mathematical support in the form of a hash compression function using a randomized Multiply-Add-and- Divide (MAD) formula, and support for automatically resizing the underlying hash table when the load factor reaches a certain threshold.
- The hashCode method relies on an original key's hash code, as returned by its hashCode() method, followed by MAD compression based on a prime number and the scale and shift parameters that are randomly chosen in the constructor.
- To manage the load factor, the AbstractHashMap class declares a protected member, n, which should equal the current number of entries in the map; however, it must rely on the subclasses to update this field from within methods bucketPut and bucketRemove.
- If the load factor of the table increases beyond 0.5, we request a bigger table (using the createTable method) and reinsert all entries into the new table. (For simplicity, this implementation uses tables of size $2k + 1$, even though these are not generally prime.)

Abstract Hash Map in Java

```
1 public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
2     protected int n = 0;                // number of entries in the dictionary
3     protected int capacity;             // length of the table
4     private int prime;                   // prime factor
5     private long scale, shift;           // the shift and scaling factors
6     public AbstractHashMap(int cap, int p) {
7         prime = p;
8         capacity = cap;
9         Random rand = new Random();
10        scale = rand.nextInt(prime-1) + 1;
11        shift = rand.nextInt(prime);
12        createTable();
13    }
14    public AbstractHashMap(int cap) { this(cap, 109345121); } // default prime
15    public AbstractHashMap() { this(17); } // default capacity
16    // public methods
17    public int size() { return n; }
18    public V get(K key) { return bucketGet(hashValue(key), key); }
19    public V remove(K key) { return bucketRemove(hashValue(key), key); }
20    public V put(K key, V value) {
21        V answer = bucketPut(hashValue(key), key, value);
22        if (n > capacity / 2) // keep load factor <= 0.5
23            resize(2 * capacity - 1); // (or find a nearby prime)
24        return answer;
25    }
26 }
```

Abstract Hash Map in Java, 2

```
26 // private utilities
27 private int hashCode(K key) {
28     return (int) ((Math.abs(key.hashCode())*scale + shift) % prime) % capacity);
29 }
30 private void resize(int newCap) {
31     ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
32     for (Entry<K,V> e : entrySet())
33         buffer.add(e);
34     capacity = newCap;
35     createTable(); // based on updated capacity
36     n = 0; // will be recomputed while reinserting entries
37     for (Entry<K,V> e : buffer)
38         put(e.getKey(), e.getValue());
39 }
40 // protected abstract methods to be implemented by subclasses
41 protected abstract void createTable();
42 protected abstract V bucketGet(int h, K k);
43 protected abstract V bucketPut(int h, K k, V v);
44 protected abstract V bucketRemove(int h, K k);
45 }
```

Code Fragment 10.7: A base class for our hash table implementations, extending the AbstractMap class from Code Fragment 10.3.

Java Hash Table Implementation- Separate Chaining

Map with Separate Chaining

Delegate operations to a list-based map at each cell:

Algorithm `get(k)`:
return `A[h(k)].get(k)`

Algorithm `put(k,v)`:
`t = A[h(k)].put(k,v)`
if `t = null` **then**
 `n = n + 1`
return `t`

{k is a new key}

Algorithm `remove(k)`:
`t = A[h(k)].remove(k)`
if `t ≠ null` **then**
 `n = n - 1`
return `t`

{k was found}

Java Hash Table Implementation- Separate Chaining

See code next slides

```

1  public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
2      // a fixed capacity array of UnsortedTableMap that serve as buckets
3      private UnsortedTableMap<K,V>[ ] table;    // initialized within createTable
4      public ChainHashMap() { super(); }
5      public ChainHashMap(int cap) { super(cap); }
6      public ChainHashMap(int cap, int p) { super(cap, p); }
7      /** Creates an empty table having length equal to current capacity. */
8      protected void createTable() {
9          table = (UnsortedTableMap<K,V>[ ]) new UnsortedTableMap[capacity];
10     }
11     /** Returns value associated with key k in bucket with hash value h, or else null. */
12     protected V bucketGet(int h, K k) {
13         UnsortedTableMap<K,V> bucket = table[h];
14         if (bucket == null) return null;
15         return bucket.get(k);
16     }
17     /** Associates key k with value v in bucket with hash value h; returns old value. */
18     protected V bucketPut(int h, K k, V v) {
19         UnsortedTableMap<K,V> bucket = table[h];
20         if (bucket == null)
21             bucket = table[h] = new UnsortedTableMap<>();
22         int oldSize = bucket.size();
23         V answer = bucket.put(k,v);
24         n += (bucket.size() - oldSize);    // size may have increased
25         return answer;
26     }

```

Hash Table with Chaining

```

27  /** Removes entry having key k from bucket with hash value h (if any). */
28  protected V bucketRemove(int h, K k) {
29      UnsortedTableMap<K,V> bucket = table[h];
30      if (bucket == null) return null;
31      int oldSize = bucket.size();
32      V answer = bucket.remove(k);
33      n -= (oldSize - bucket.size());    // size may have decreased
34      return answer;
35  }
36  /** Returns an iterable collection of all key-value entries of the map. */
37  public Iterable<Entry<K,V>> entrySet() {
38      ArrayList<Entry<K,V>> buffer = new ArrayList<>();
39      for (int h=0; h < capacity; h++)
40          if (table[h] != null)
41              for (Entry<K,V> entry : table[h].entrySet())
42                  buffer.add(entry);
43      return buffer;
44  }
45  }

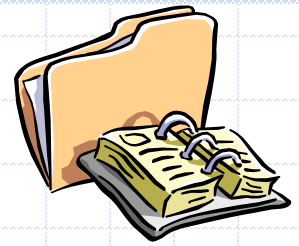
```

Hash Table with Chaining, 2

Java Hash Table Implementation- Linear Probing

See code next slides

Search with Linear Probing



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

Algorithm *get(k)*

$i \leftarrow h(k)$

$p \leftarrow 0$

repeat

$c \leftarrow A[i]$

if $c = \emptyset$

return *null*

else if $c.getKey() = k$

return $c.getValue()$

else

$i \leftarrow (i + 1) \bmod N$

$p \leftarrow p + 1$

until $p = N$

return *null*

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

Probe Hash Map in Java

```
1 public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
2     private MapEntry<K,V>[ ] table;           // a fixed array of entries (all initially null)
3     private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null); //sentinel
4     public ProbeHashMap() { super(); }
5     public ProbeHashMap(int cap) { super(cap); }
6     public ProbeHashMap(int cap, int p) { super(cap, p); }
7     /** Creates an empty table having length equal to current capacity. */
8     protected void createTable() {
9         table = (MapEntry<K,V>[ ]) new MapEntry[capacity]; // safe cast
10    }
11    /** Returns true if location is either empty or the "defunct" sentinel. */
12    private boolean isAvailable(int j) {
13        return (table[j] == null || table[j] == DEFUNCT);
14    }
```

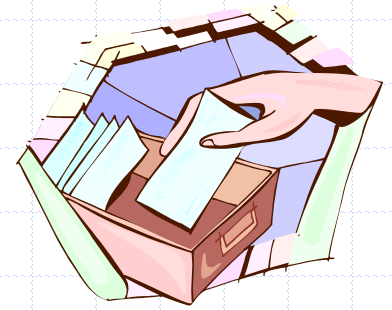
Probe Hash Map in Java, 2

```
15  /** Returns index with key k, or  $-(a+1)$  such that k could be added at index a. */
16  private int findSlot(int h, K k) {
17      int avail = -1; // no slot available (thus far)
18      int j = h; // index while scanning table
19      do {
20          if (isAvailable(j)) { // may be either empty or defunct
21              if (avail == -1) avail = j; // this is the first available slot!
22              if (table[j] == null) break; // if empty, search fails immediately
23          } else if (table[j].getKey().equals(k))
24              return j; // successful match
25          j = (j+1) % capacity; // keep looking (cyclically)
26      } while (j != h); // stop if we return to the start
27      return -(avail + 1); // search has failed
28  }
29  /** Returns value associated with key k in bucket with hash value h, or else null. */
30  protected V bucketGet(int h, K k) {
31      int j = findSlot(h, k);
32      if (j < 0) return null; // no match found
33      return table[j].getValue();
34  }
```

```

35  /** Associates key k with value v in bucket with hash value h; returns old value. */
36  protected V bucketPut(int h, K k, V v) {
37      int j = findSlot(h, k);
38      if (j >= 0)                                // this key has an existing entry
39          return table[j].setValue(v);
40      table[-(j+1)] = new MapEntry<>(k, v);    // convert to proper index
41      n++;
42      return null;
43  }
44  /** Removes entry having key k from bucket with hash value h (if any). */
45  protected V bucketRemove(int h, K k) {
46      int j = findSlot(h, k);
47      if (j < 0) return null;                    // nothing to remove
48      V answer = table[j].getValue();
49      table[j] = DEFUNCT;                      // mark this slot as deactivated
50      n--;
51      return answer;
52  }
53  /** Returns an iterable collection of all key-value entries of the map. */
54  public Iterable<Entry<K,V>> entrySet() {
55      ArrayList<Entry<K,V>> buffer = new ArrayList<>();
56      for (int h=0; h < capacity; h++)
57          if (!isAvailable(h)) buffer.add(table[h]);
58      return buffer;
59  }
60  }

```



Double Hashing

- ❑ Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(i + jd(k)) \bmod N$$
for $j = 0, 1, \dots, N - 1$
- ❑ The secondary hash function $d(k)$ cannot have zero values
- ❑ The table size N must be a prime to allow probing of all the cells
- ❑ Common choice of compression function for the secondary hash function:
$$d_2(k) = q - k \bmod q$$
where
 - $q < N$
 - q is a prime
- ❑ The possible values for $d_2(k)$ are
$$1, 2, \dots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
 - $h(k) = k \bmod 13$
 - $d(k) = 7 - k \bmod 7$

- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

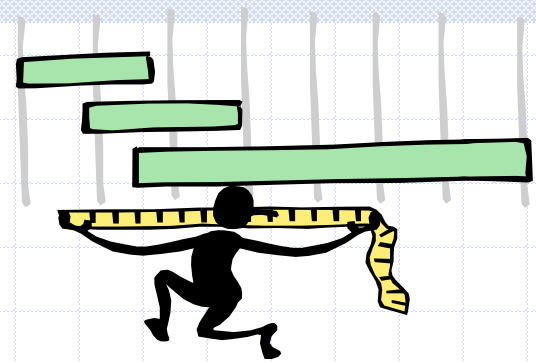
k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	
32	6	3	6	
31	5	4	5	9 0
73	8	4	8	

0	1	2	3	4	5	6	7	8	9	10	11	12



31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance of Hashing



- ❑ In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- ❑ The worst case occurs when all the keys inserted into the map collide
- ❑ The load factor $\alpha = n/N$ affects the performance of a hash table
- ❑ Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is
$$1 / (1 - \alpha)$$
- ❑ The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- ❑ In practice, hashing is very fast provided the load factor is not close to 100%
- ❑ Applications of hash tables:
 - small databases
 - compilers
 - browser caches