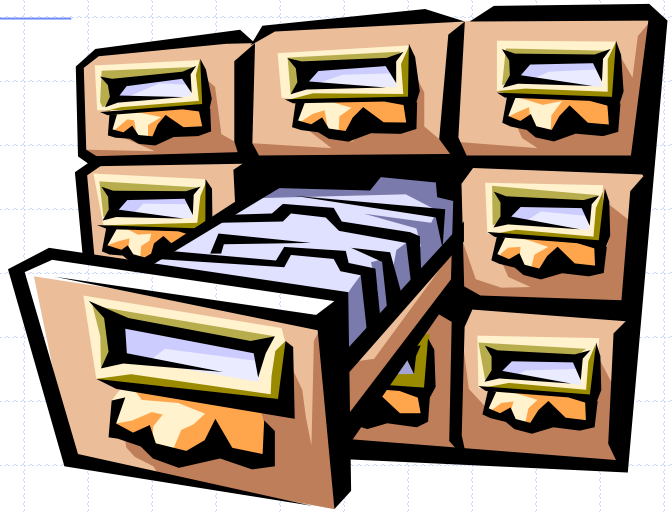
Modified version of the presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Maps



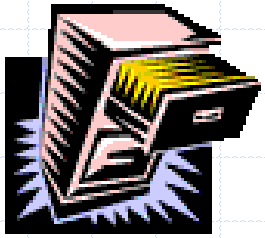
Application Example: Document Classification

- You are an algorithmic trader and want to classify news about companies as good or bad to make buy/sell decisions
- You already have examples of good and bad documents and want to train a classifier
- How do you represent all these news documents such that they can be used in machine learning?
- An idea is to use the frequency of words in these documents
- This is the basis of the term frequency-inverse document frequency statistic
- How do you count the word frequency?

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
- Note that a total order relation is not needed

The Map ADT



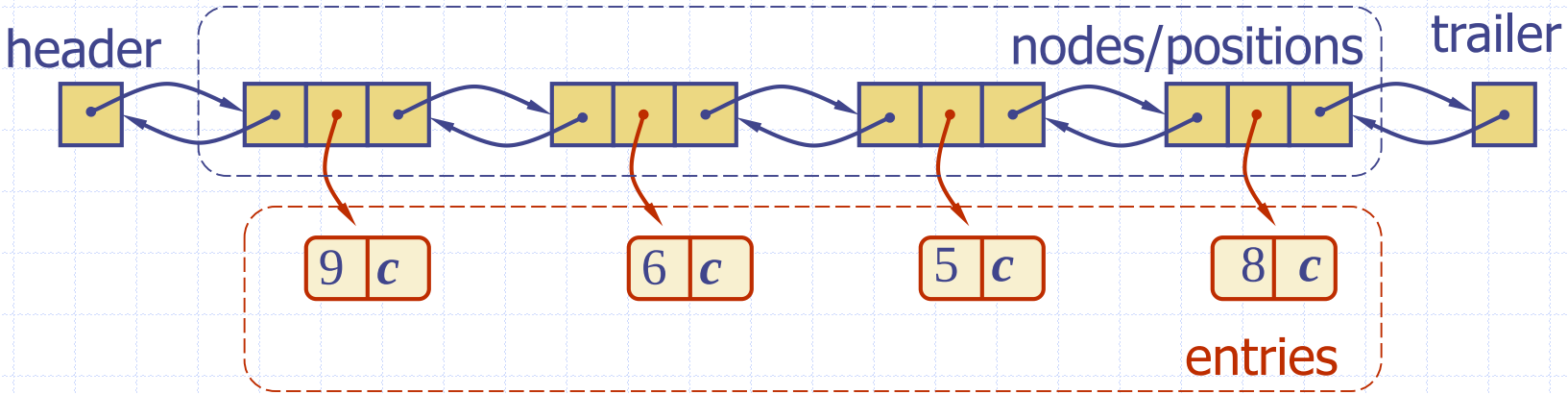
- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M 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

Example

Operation	*Output*	*Map*
isEmpty()	**true**	Ø
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*)
isEmpty()	**false**	(7,*B*),(8,*D*)

A Simple List-Based Map

- We can implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doublylinked list), in arbitrary order
  - Similar to what you had in lab 1



The get(k) Algorithm

```
Algorithm get(k):
    B = S.positions() {B is an iterator of the positions in S}
    while B.hasNext() do
        p = B.next()   { the next position in B }
        if p.element().getKey() = k        then
            return p.element().getValue()
    return null {there is no entry with key equal to k}
```

The put(k,v) Algorithm

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
          t = p.element().getValue()
          S.set(p,(k,v))
          return t          {return the old value}
S.addLast((k,v))
n = n + 1          {increment variable storing number of entries}
return null     { there was no entry with key equal to k }
```

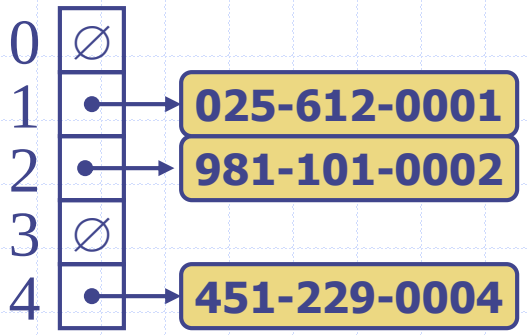
The remove(k) Algorithm

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
        t = p.element().getValue()
        S.remove(p)
        n = n – 1      {decrement number of entries}
        return t       {return the removed value}
return null            {there is no entry with key equal to k}
```

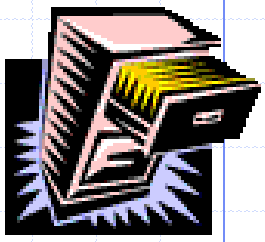
Performance of a List-Based Map

- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence (unless we check for an existing key which make this $O(n)$)
  - 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
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

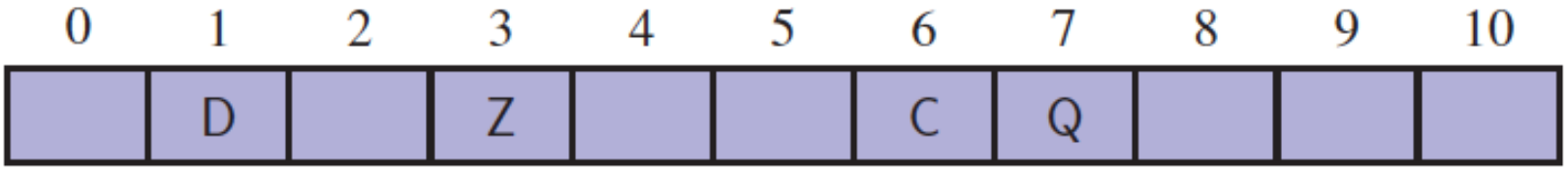
Hash Tables



Intuitive Notion of a Map

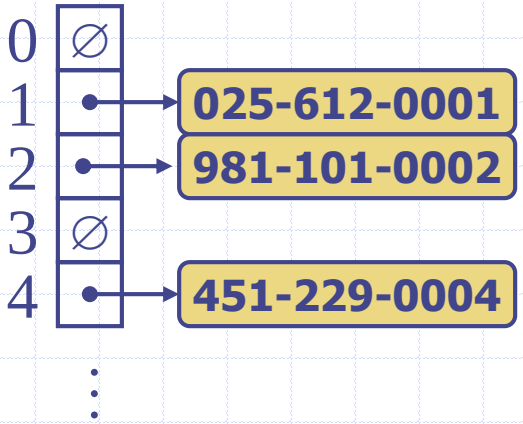


- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].
- As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to N − 1, for some N ≥ n - Arrays

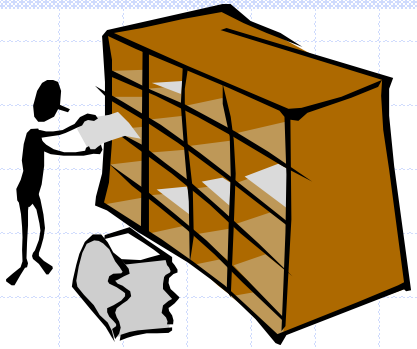


More General Kinds of Keys

- 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:

  $$h(x) = x \bmod N$$

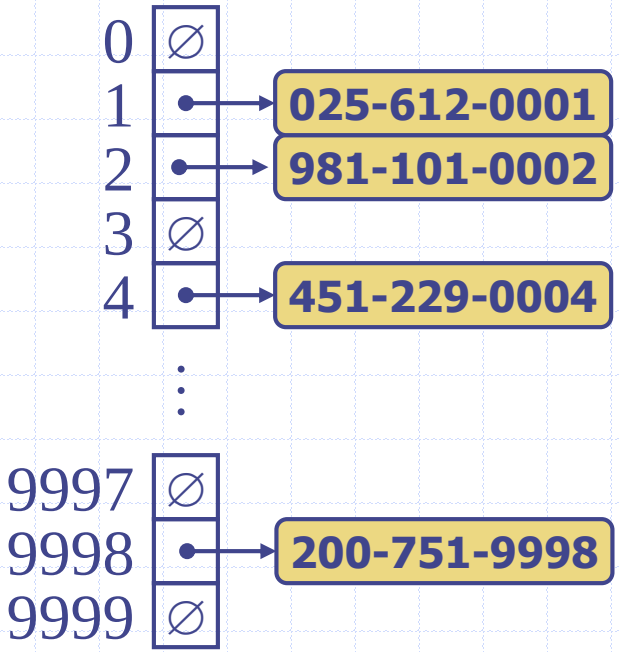
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Some Definitions

- Bucket: A single fast-access location in the underlying container (e.g. an array element)
- Capacity: The size of the underlying container i.e. the number of buckets. Can be dynamic.
- Load Factor: The ration of the number of full buckets to the capacity i.e. $\alpha = n/N$ where n is the number of full buckets and N is the number of buckets

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



Hash Functions

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

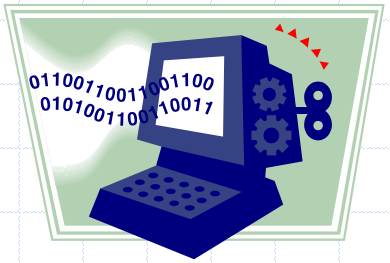
  h_2: integers $\rightarrow [0, N-1]$

- 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)

    $$a_0 a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial

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

    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

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

    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

Java's Object.hashCode()

- The base class *Object* in java has a method called *hashCode*
- This returns a hash code, as an *int*, for the object
- Default implementation is JVM specific
- The *String* class returns the same *hashCode* for the same strings even if they refer to different objects

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):
  - $h_1(y) = (ay + b) \bmod P$
  - $h_2(y) = (h_1(y)) \bmod N$
  - a and b are nonnegative integers such that $a \bmod P \neq 0$
  - Otherwise, every integer would map to the same value $b \bmod P$
  - This approach, along with a with a prime number results in more uniform distribution of the keys

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   }
```

Abstract Hash Map in Java, 2

```
26    // private utilities
27    private int hashValue(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  }
```

The MAD method of calculating. The absolute value is needed to avoid negative due to overflow.

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                    {k is a new key}
        n = n + 1
    return t

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

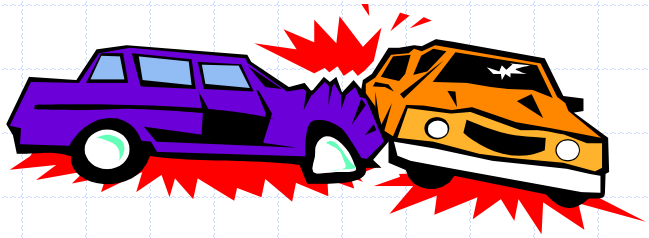
Hash Table with Chaining

```
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   }
```

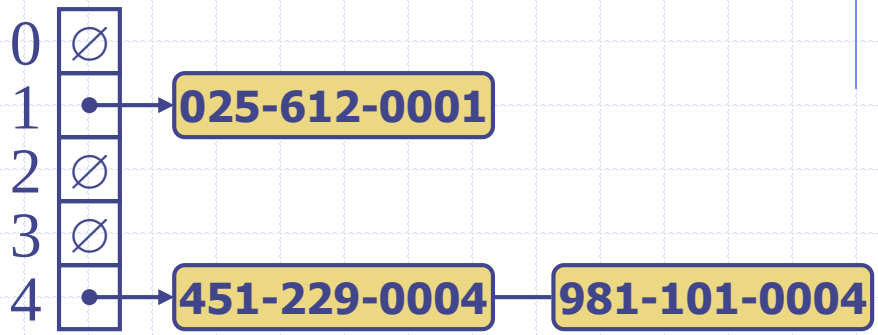
Hash Table with Chaining, 2

```
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  }
```

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

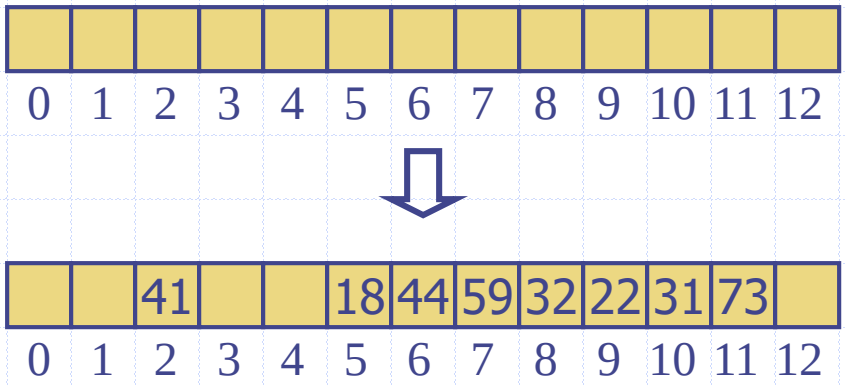


- Separate chaining is simple, but requires additional memory outside the table

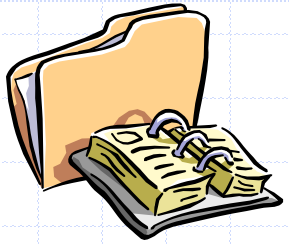
Linear Probing

- 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



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

Why do we need the $DEFUNCT$ object?

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   }
```

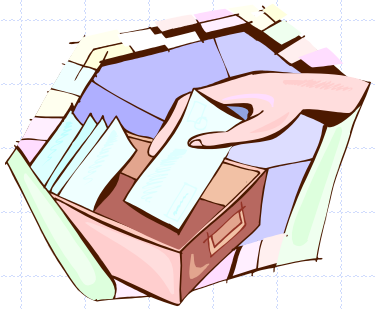
Probe Hash Map in Java, 2

```
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    }
```

Probe Hash Map in Java, 3

```
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



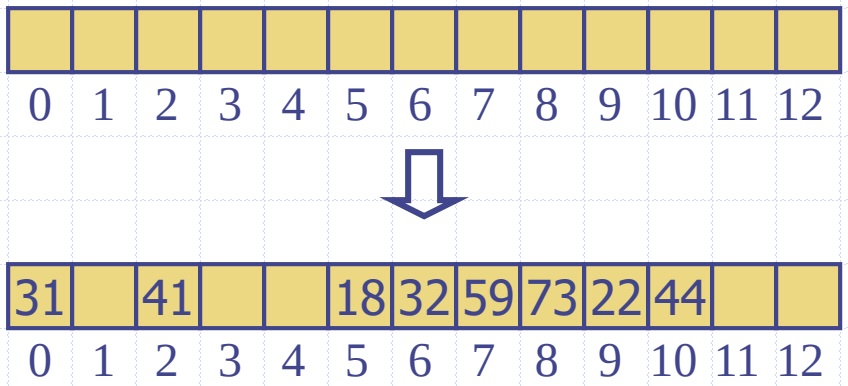
- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(i + jd(k)) \bmod N$$
for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing 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
$$1, 2, \ldots, q$$

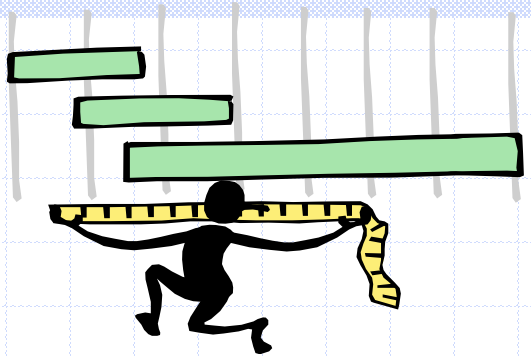
Example of Double Hashing

- 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	
41	2	1	
22	9	6	
44	5	5	
59	7	4	
32	6	3	
31	5	4	
73	8	4	



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