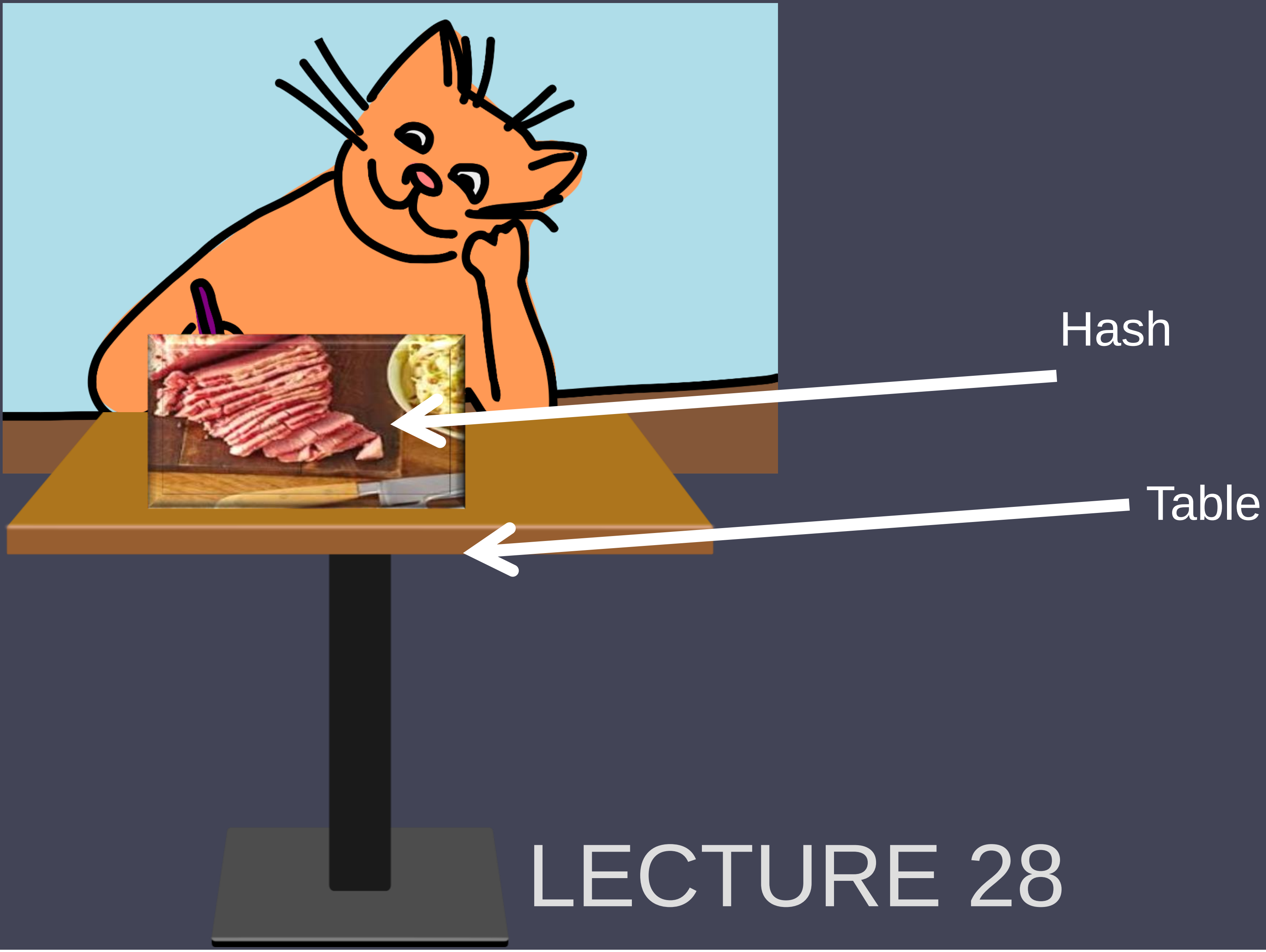


LECTURE 28

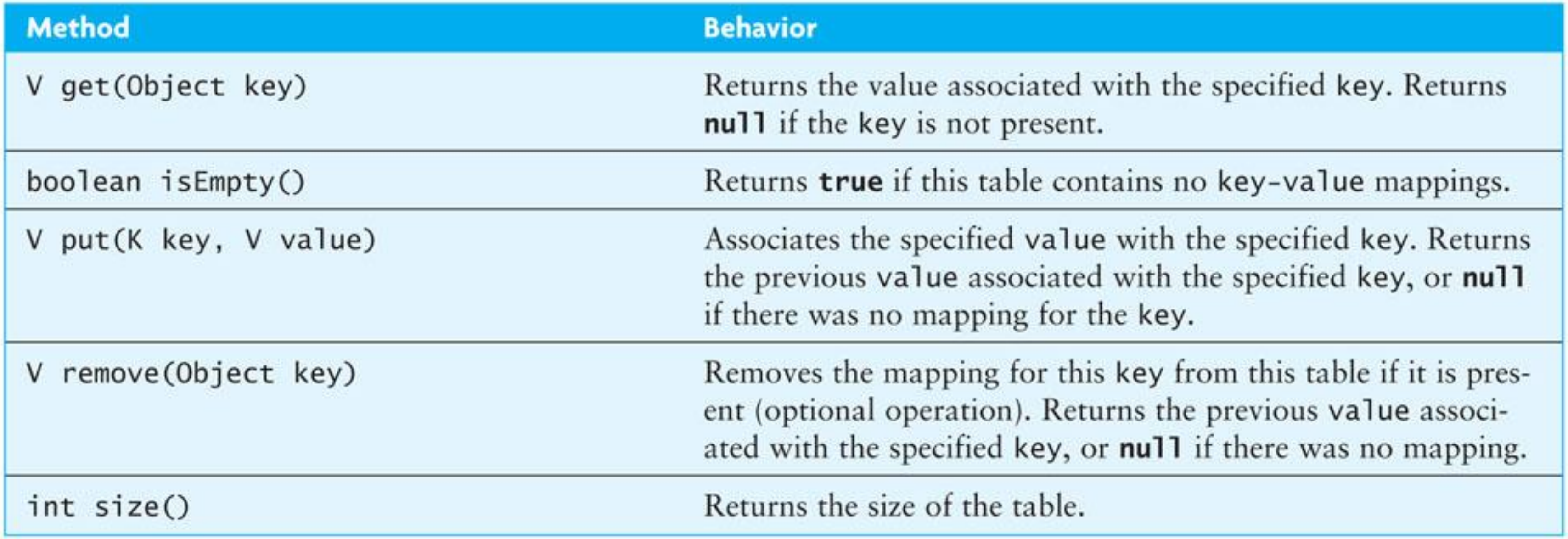
Implementing the Hash Table

Assignment

- Read section 7.4
  - Find out whether the statement in question on slide 7 makes sense and, if so, why

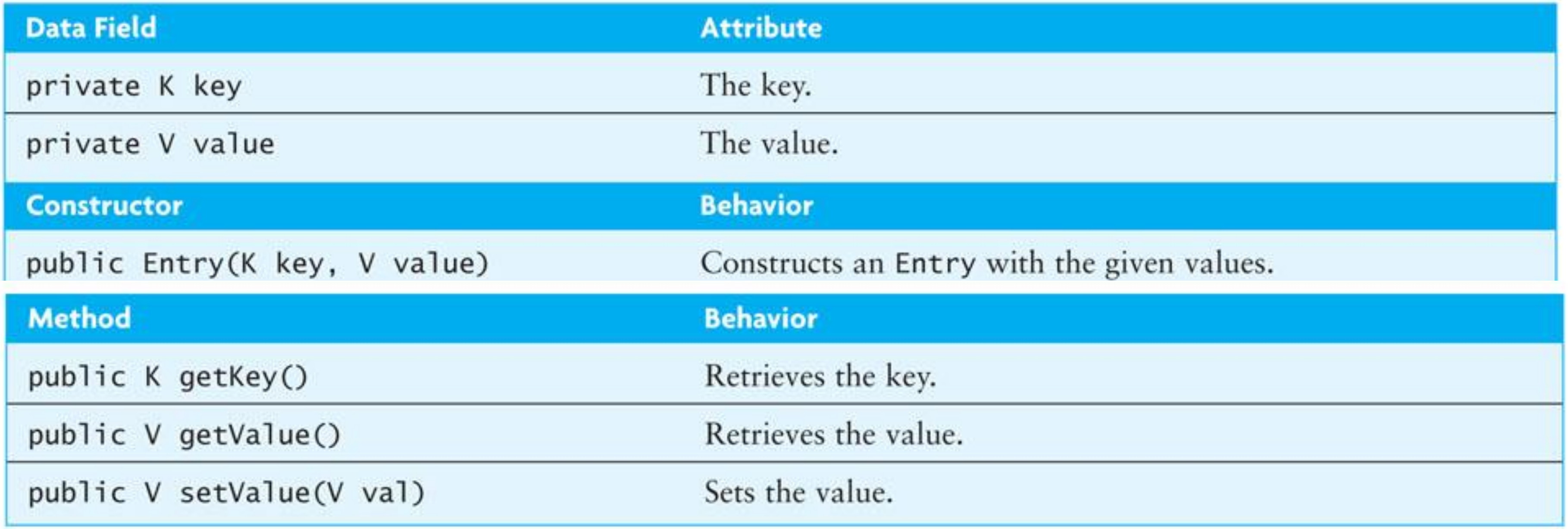
    (Hint: Check the *hashCode ()* method and find out whether it may return a negative number)
  - Study Listing 7.6 (Method `HashtableOpen.put`; page 389) and respond to the question on slide 11.
  - Perform the exercise on slide 23-26 (write the test cases and run them) by Friday.
  - Be ready to present answers in class on Wednesday

Interface `KWHashMap`



Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this table contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key`. Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this table if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping.
`int size()`	Returns the size of the table.

Class `Entry`

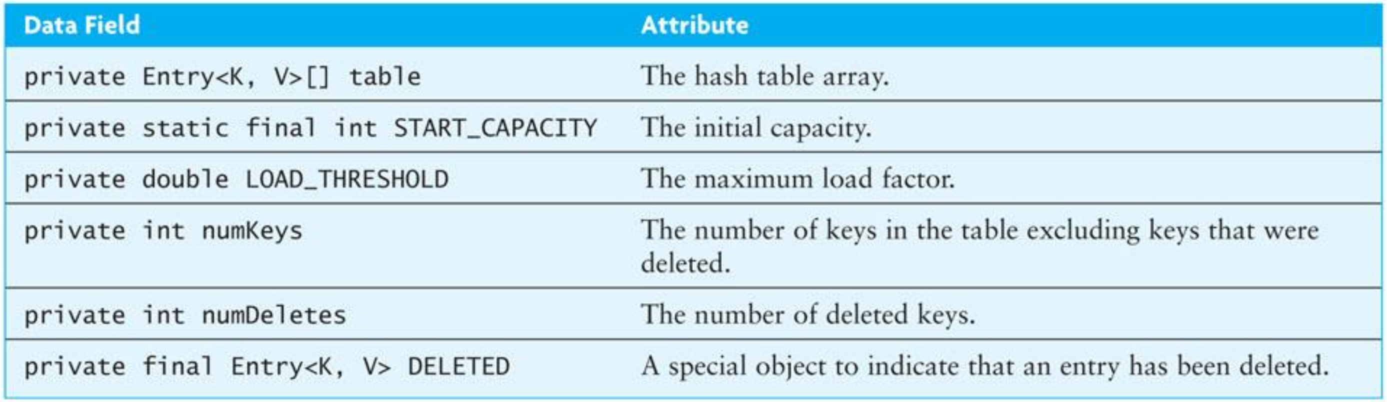


Data Field	Attribute
`private K key`	The key.
`private V value`	The value.
Constructor	**Behavior**
`public Entry(K key, V value)`	Constructs an `Entry` with the given values.
Method	**Behavior**
`public K getKey()`	Retrieves the key.
`public V getValue()`	Retrieves the value.
`public V setValue(V val)`	Sets the value.

Class `Entry` (cont.)

- Listing 7.3 (Inner Class `Entry`; page 385)

Class HashTableOpen



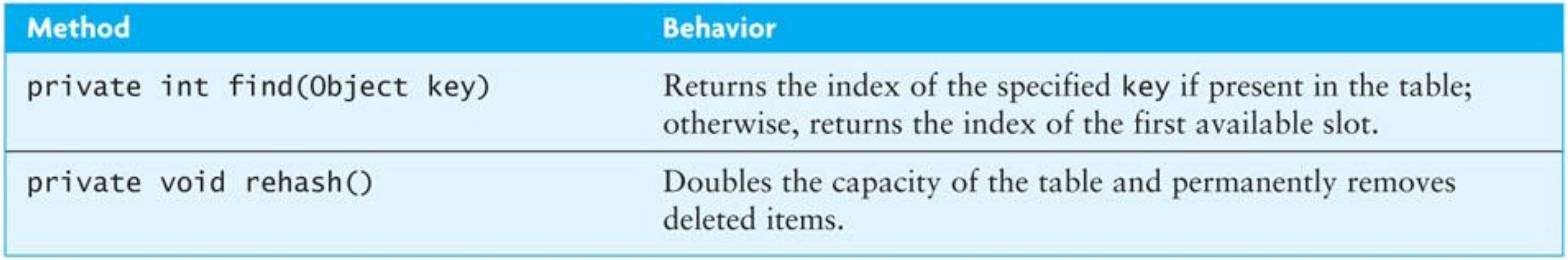
Data Field	Attribute
`private Entry<K, V>[] table`	The hash table array.
`private static final int START_CAPACITY`	The initial capacity.
`private double LOAD_THRESHOLD`	The maximum load factor.
`private int numKeys`	The number of keys in the table excluding keys that were deleted.
`private int numDeletes`	The number of deleted keys.
`private final Entry<K, V> DELETED`	A special object to indicate that an entry has been deleted.

```
/** Hash table implementation using open addressing. */
public class HashtableOpen<K, V> implements KWHashMap<K, V> {
    // Data Fields
    private Entry<K, V>[] table;
    private static final int START_CAPACITY = 101;
    private double LOAD_THRESHOLD = 0.75;
    private int numKeys;
    private int numDeletes;
    private final Entry<K, V> DELETED =
            new Entry<K, V>(null, null);

    // Constructor
    public HashTableOpen() {
        table = new Entry[START_CAPACITY];
    }

    // Insert inner class Entry<K, V> here.
    . . .
```

Class HashTableOpen (cont.)



Method	Behavior
private int find(Object key)	Returns the index of the specified key if present in the table; otherwise, returns the index of the first available slot.
private void rehash()	Doubles the capacity of the table and permanently removes deleted items.

Algorithm for HashtableOpen.find(Object key)

```
1. Set index to key.hashCode() % table.length.
2. if index is negative, add table.length. ???
3. while table[index] is not empty and the key is not at table[index]
4.         increment index.
5.         if index is greater than or equal to table.length
6.                 Set index to 0.
7. Return the index.
```

Class `HashTableOpen` (cont.)

- Listing 7.4 (Method `HashtableOpen.find`; page 387)

Class `HashTableOpen` (cont.)

Algorithm for `get(Object key)`

```
1. Find the first table element that is empty or the table element that contains the key.
2. if the table element found contains the key
           return the value at this table element.
3. else
4.         return null.
```

Class `HashTableOpen` (cont.)

- Listing 7.5 (Method `HashtableOpen.get`; page 388)

Class HashTableOpen (cont.)

Algorithm for HashtableOpen.put(K key, V value)

```
1. Find the first table element that is empty or the table element that contains
   the key.
2. if an empty element was found
3.         insert the new item and increment numKeys.
4.         check for need to rehash.
5.         return null.
6. The key was found. Replace the value associated with this table element and
   return the old value.
```

How?

Class `HashTableOpen` (cont.)

- Listing 7.6 (Method `HashtableOpen.put`; page 389)

Class `HashTableOpen` (cont.)

Algorithm for `remove(Object key)`

```
1. Find the first table element that is empty or the table element that contains
   the key.
2. if an empty element was found
3.        return null.
4. Key was found. Remove this table element by setting it to reference DELETED,
   increment numDeletes, and decrement numKeys.
5. Return the value associated with this key.
```

Class `HashTableOpen` (cont.)

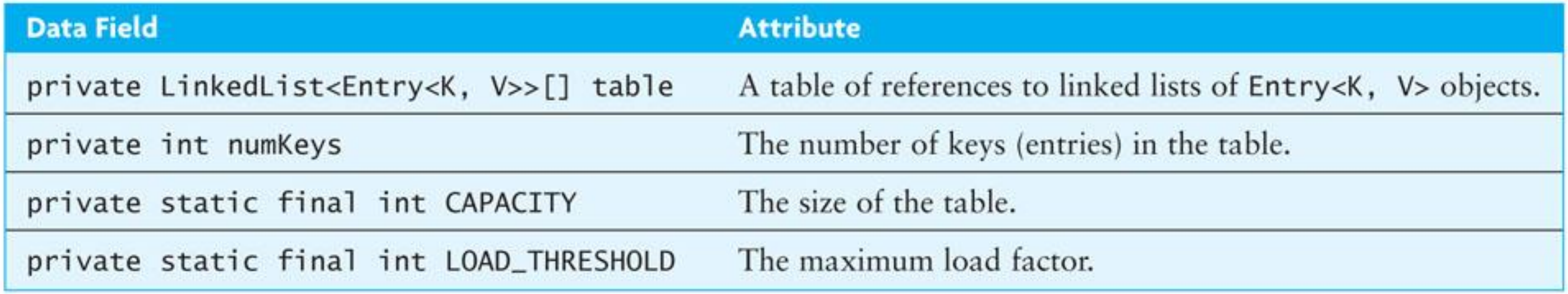
Algorithm for `HashtableOpen.rehash`

1. Allocate a new hash table that is at least double the size and has ~~an odd~~ prime length.
2. Reset the number of keys and number of deletions to 0.
3. Reinsert each table entry that has not been deleted in the new hash table.

Class `HashTableOpen` (cont.)

- Listing 7.7 (Method `HashtableOpen.rehash`; page 390)

Class `HashTableChain`



Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.

- Listing 7.8 (Data Fields and Constructor for `HashtableChain.java`; page 391)

Class HashTableChain (cont.)

Algorithm for HashtableChain.get(Object key)

```
1. Set index to key.hashCode() % table.length.
2. if index is negative
3.         add table.length.
4. if table[index] is null
5.         key is not in the table; return null.
6. For each element in the list at table[index]
7.         if that element's key matches the search key
8.                 return that element's value.
9. key is not in the table; return null.
```

Class `HashTableChain` (cont.)

- Listing 7.9 (Method `HashtableChain.get`; page 392)

Class HashTableChain (cont.)

Algorithm for HashtableChain.put(K key, V value)

```
1. Set index to key.hashCode() % table.length.
2. if index is negative, add table.length.
3. if table[index] is null
4.         create a new linked list at table[index];
   else N.B.!  (compare the book, p. 392)
5.         Search the list at table[index] to find the key.
6.         if the search is successful
7.                 replace the value associated with this key.
8.                 return the old value.
9.         else
10.                insert the new key-value pair in the linked list located at
                   table[index].
11.                increment numKeys.
12.                if the load factor exceeds the LOAD_THRESHOLD
13.                 Rehash.
14.                return null.
```

Class `HashTableChain` (cont.)

- Listing 7.10 (Method `HashtableChain.put`; page 393)

Class HashTableChain (cont.)

Algorithm for HashtableChain.remove(Object key)

```
1. Set index to key.hashCode() % table.length.
2. if index is negative, add table.length.
3. if table[index] is null
4. key is not in the table; return null.  /* Same problem */
5.  else
5.    Search the list at table[index] to find the key.
6.     if the search is successful
7.          remove the entry with this key and decrement numKeys.
8.          if the list at table[index] is empty
9.                Set table[index] to null.
10.         return the value associated with this key.
11.   else

11.      The key is not in the table; return null.
```

Testing the Hash Table Implementation

- Write a method to
  - create a file of key-value pairs
  - read each key-value pair and insert it in the hash table
  - observe how the hash table is filled
- Implementation
  - Write a `toString` method that captures the index of each non-`null` table element and the contents of the table element
  - For open addressing, the contents is the string representation of the key-value pair
  - For chaining, a list iterator can traverse at the table element and append each key-value pair to the resulting string

Testing the Hash Table Implementation (cont.)

- Cases to examine:
  - Does the array index wrap around as it should?
  - Are collisions resolved correctly?
  - Are duplicate keys handled appropriately? Is the new value retrieved instead of the original value?
  - Are deleted keys retained in the table but no longer accessible via a `get`?
  - Does rehashing occur when the load factor reaches 0.75 (3.0 for chaining)?
- Step through the `get` and `put` methods to
  - observe how the table is probed
  - examine the search chain followed to access or retrieve a key

Testing the Hash Table Implementation (cont.)

- Alternatively, insert randomly generated integers in the hash table to create a large table with O(n) effort

```
for (int i = 0; i < SIZE; i++) {
    Integer nextInt = (int) (32000 * Math.random());
    hashTable.put(nextInt, nextInt):
}
```

Testing the Hash Table Implementation

- Insertion of randomly generated integers into a table allows testing of tables of very large sizes, but is less helpful for testing for collisions
- You can add code to count the number of items probed each time an insertion is made—these can be totaled to determine the average search chain length

Implementation Considerations for Maps and Sets

Methods `hashCode` and `equals`

- □ Class `Object` implements methods `hashCode` and `equals`, so every class can access these methods unless it overrides them, but they are likely to be useless!
- □ `Object.equals`, as we know, compares two objects based on their addresses, not their contents
- □ Most predefined classes override method `equals` and compare objects based on content
- □ If you want to compare two objects (whose classes you've written) for equality of content, you need to override the `equals` method

Methods `hashCode` and `equals`

- □ `Object.hashCode` calculates an object's hash code based on its address, not its contents
- □ Most predefined classes also override method `hashCode`
- □ Java recommends that if you override the `equals` method, then you should also override the `hashCode` method
- □ Otherwise, you violate the following rule:

If obj1.equals(obj2) is true, then
obj1.hashCode = obj2.hashCode