Hashing

CSE219, Computer Science III

Stony Brook University

http://www.cs.stonybrook.edu/~cse219

Hashing and Maps

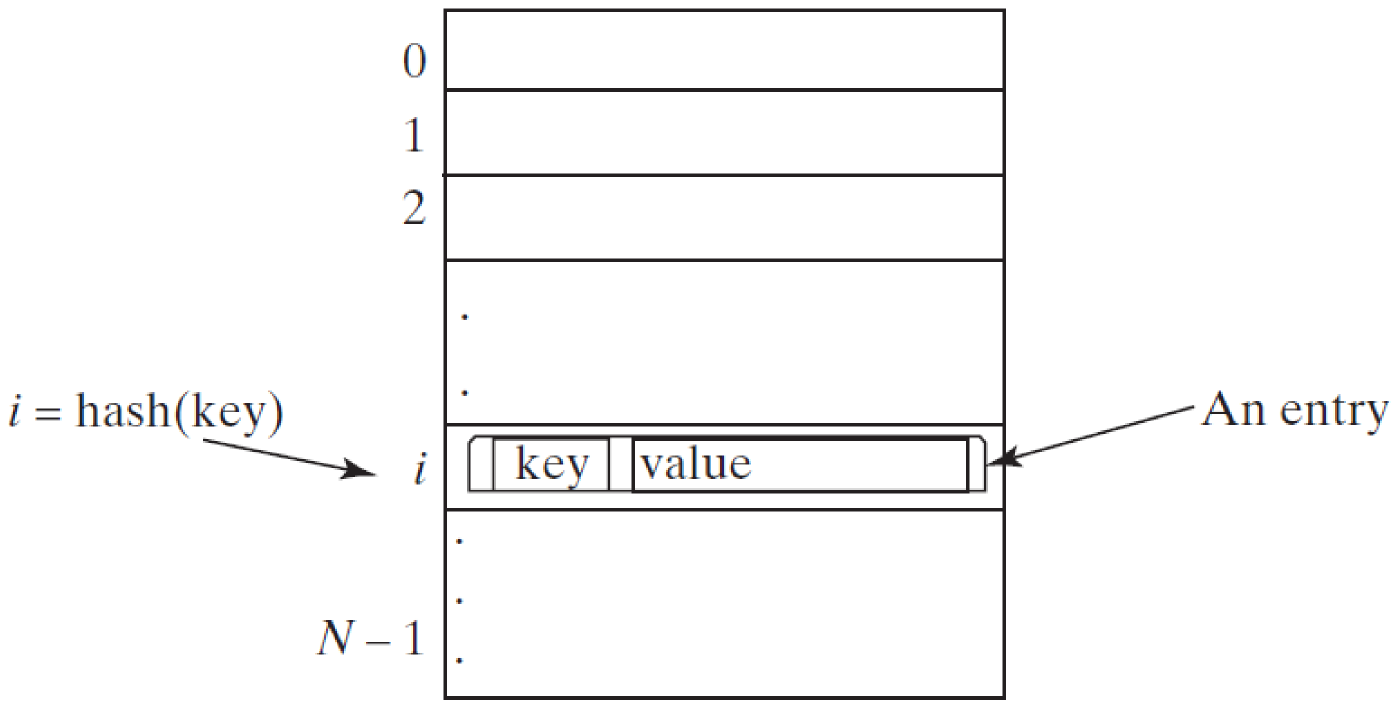
- Why hashing?
  - O(1) time to search, insert, and delete an element in a map or a set vs. O(logn) time in a well-balanced search tree.
- A *map* (aka. *dictionary*, a *hash table*, or an *associative array*) is a data structure that stores entries, where each entry contains two parts: **key** (also called a search key) and **value**.
  - The key is used to search for the corresponding value.
  - For example, a dictionary can be stored in a map, where the words are the keys and the definitions of the words are the values.

Hashing

- If you know the index of an element in the array, you can retrieve the element using the index in O(1) time.
  - So, can we store the values in an array and use the key as the index to find the value?
  - The answer is yes if you can map a key to an index.
- The array that stores the values is called a ***hash table***.
- The function that maps a key to an index in the hash table is called a ***hash function***.
- ***Hashing*** is a technique that retrieves the value using the index obtained from key without performing a search.

Hash Function and Hash Codes

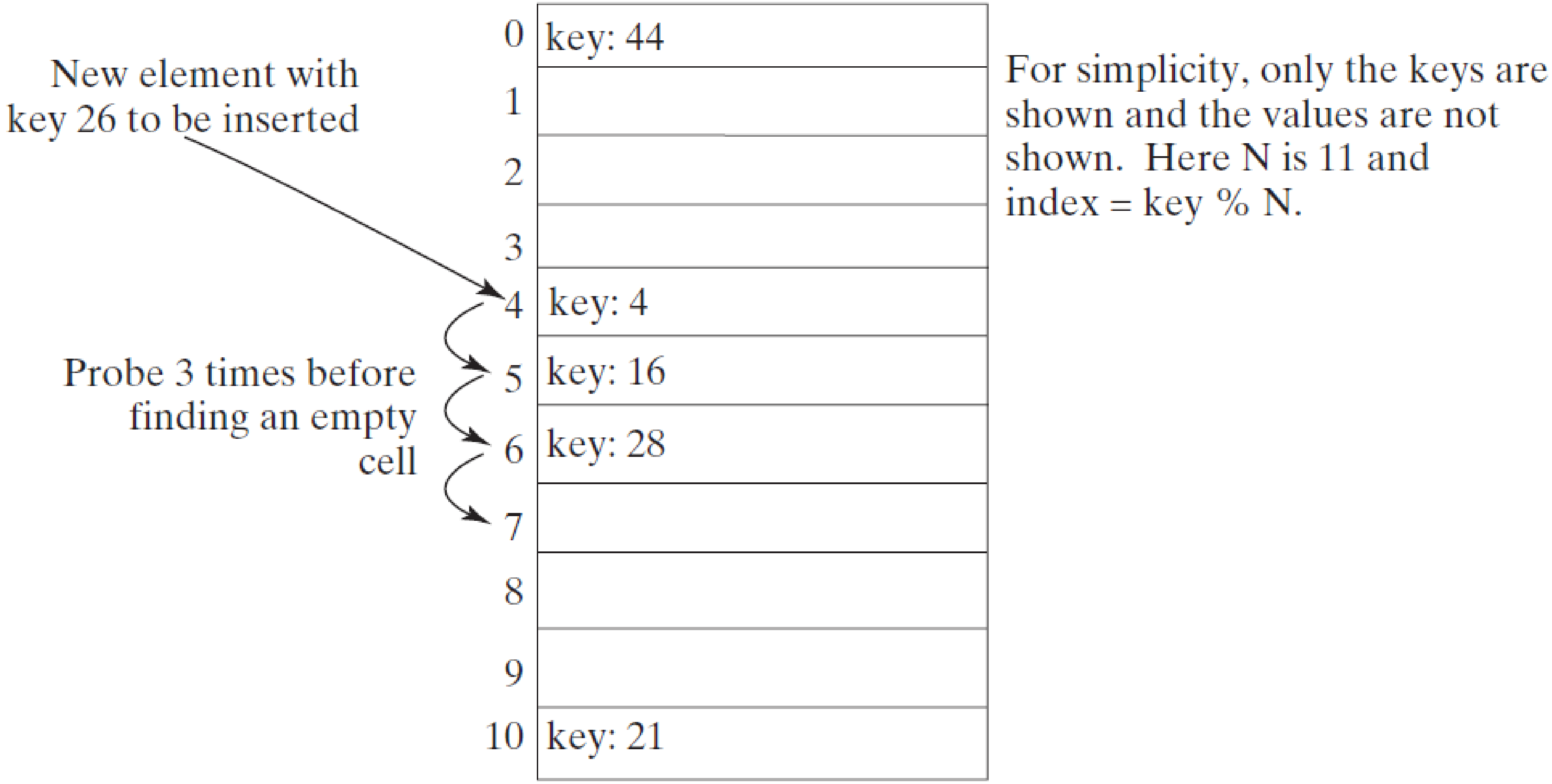
- A typical hash function first converts a search key to an integer value called a ***hash code***, and then compresses the hash code into an index to the hash table.



Hash function

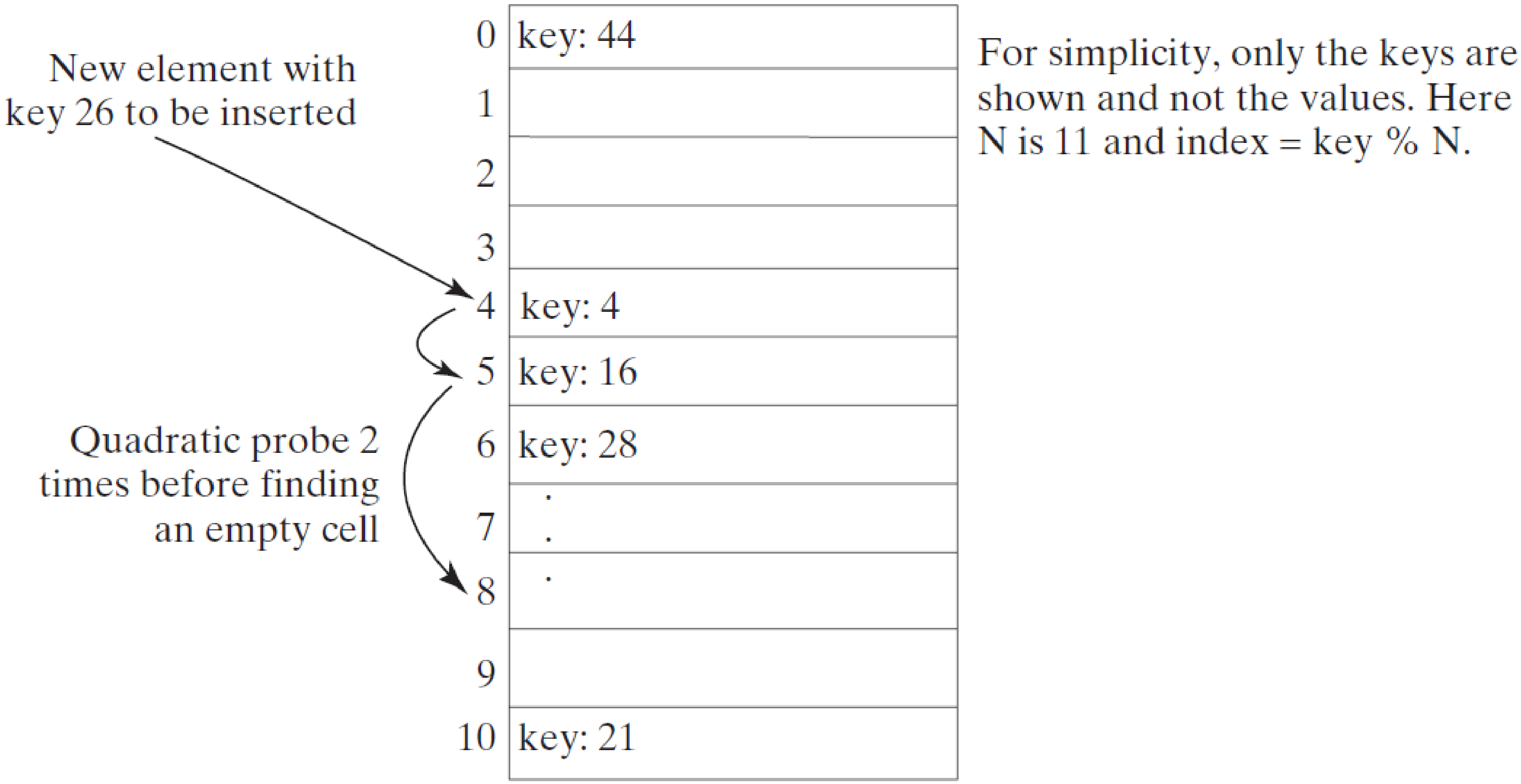
Linear Probing

- If a index position is already occupied, *linear probing* algorithm looks at the consecutive cells beginning at index k



Quadratic Probing

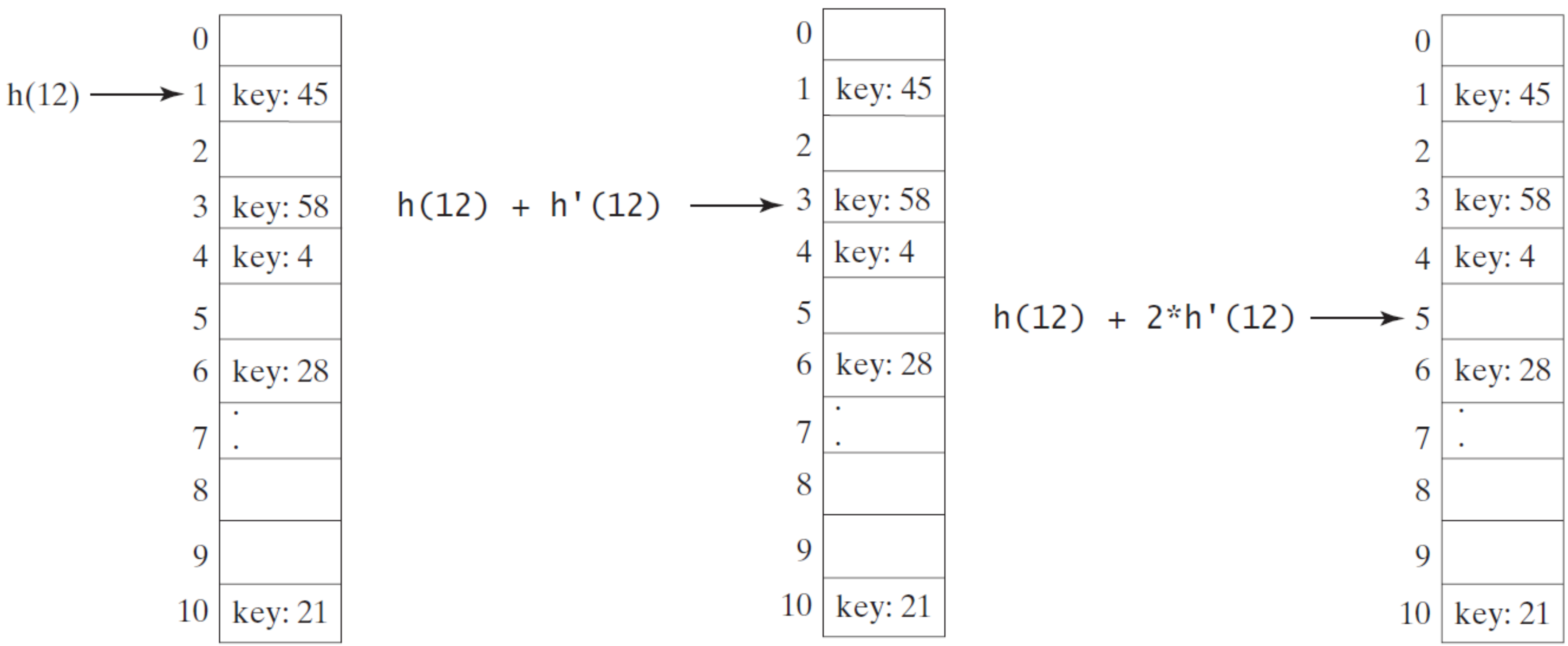
- *Quadratic probing* can avoid the clustering problem in linear probing:



Double Hashing

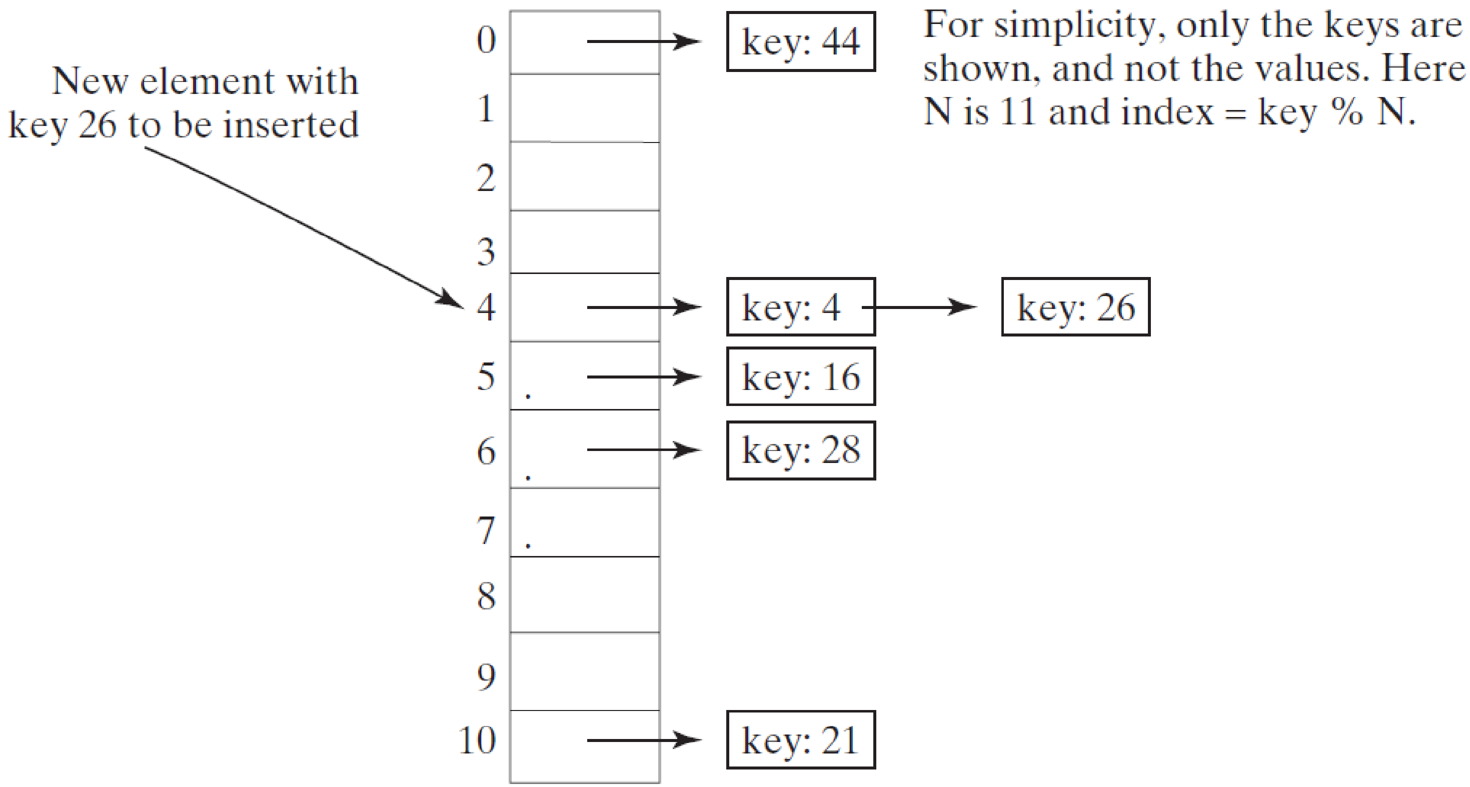
- *Double hashing* uses a secondary hash function on the keys to determine the increments to avoid the clustering problem

$$h'(k) = 7 - k \% 7;$$



Handling Collisions Using Separate Chaining

- The *separate chaining scheme* places all entries with the same hash index into the same location, rather than finding new locations.
  - Each location in the separate chaining scheme is called a *bucket*.
  - A bucket is a container that holds multiple entries.



```java
import java.util.HashMap;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> map;
        map = new HashMap<>();
        map.put("Smith", 30);
        map.put("Anderson", 31);
        map.put("Lewis", 29);
        map.put("Cook", 29);
        map.put("Smith", 65);

        System.out.println("Entries in map: " + map);

        System.out.println("The age for " + "Lewis is "
                + map.get("Lewis"));

        System.out.println("Is Smith in the map? "
                + map.containsKey("Smith"));
        System.out.println("Is age 33 in the map? "
                + map.containsValue(33));

        map.remove("Smith");
        System.out.println("Entries in map: " + map);

        map.clear();
        System.out.println("Entries in map: " + map);
    }
}
```

Entries in map:
{Lewis=29, Smith=65,
Cook=29, Anderson=31}

The age for Lewis is 29

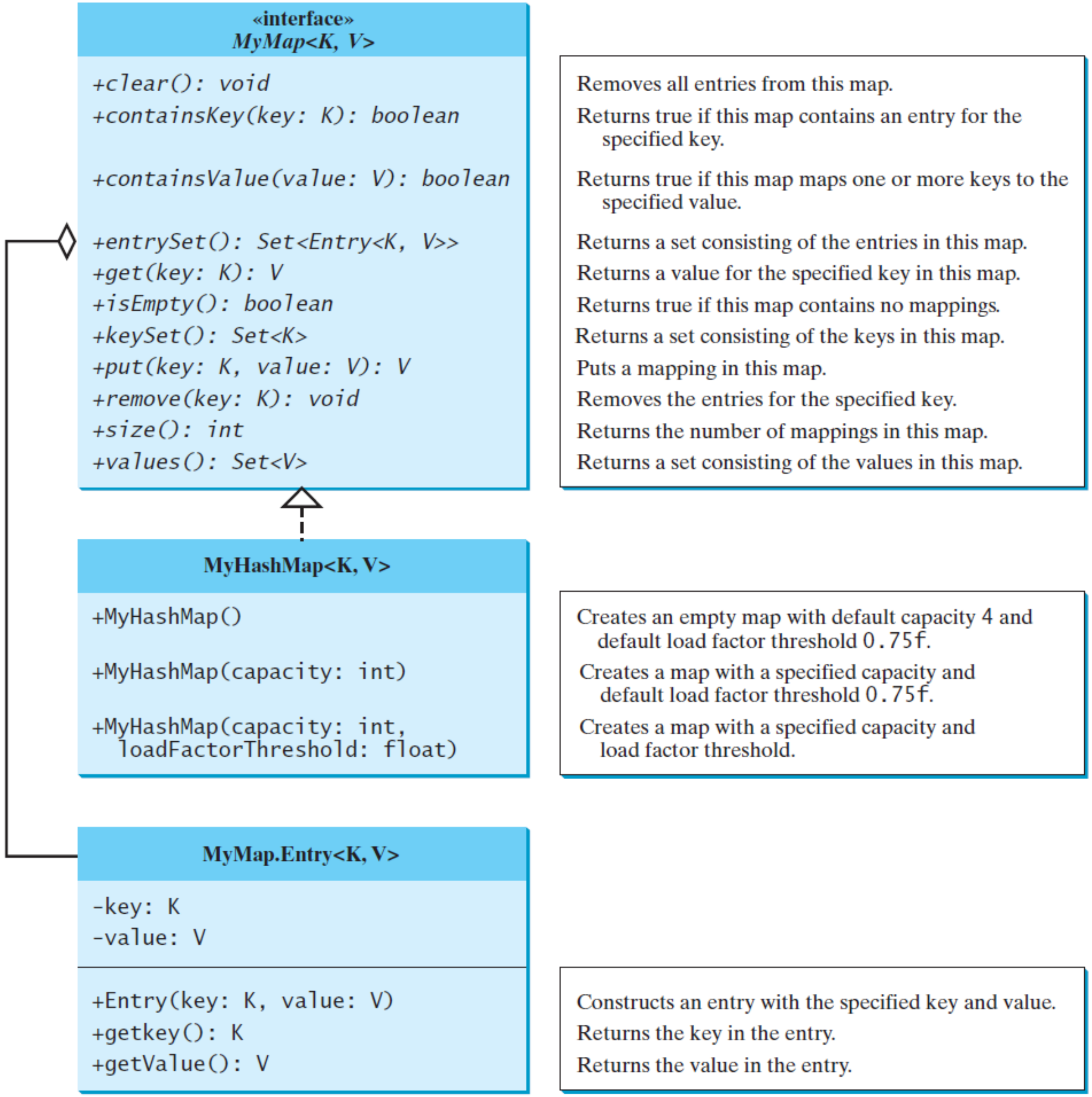
Is Smith in the map? true

Is age 33 in the map? false

Entries in map:
{Lewis=29, Cook=29,
Anderson=31}

Entries in map: {}

Best Way to Learn: do it yourself!



```
public interface MyMap<K, V> {
  /** Remove all of the entries from this map */
  public void clear();

  /** Return true if the specified key is in the map */
  public boolean containsKey(K key);

  /** Return true if this map contains the specified value */
  public boolean containsValue(V value);

  /** Return a set of entries in the map */
  public java.util.Set<Entry<K, V>> entrySet();

  /** Return the first value that matches the specified key */
  public V get(K key);

  /** Return true if this map contains no entries */
  public boolean isEmpty();

  /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet();

  /** Add an entry (key, value) into the map */
  public V put(K key, V value);

  /** Remove the entries for the specified key */
  public void remove(K key);
```

```
  /** Return the number of mappings in this map */
  public int size();

  /** Return a set consisting of the values in this map */
  public java.util.Set<V> values();

  /** Define inner class for Entry */
  public static class Entry<K, V> {
    K key;
    V value;

    public Entry(K key, V value) {
      this.key = key;
      this.value = value;
    }

    public K getKey() {
      return key;
    }

    public V getValue() {
      return value;
    }

    @Override
    public String toString() {
      return "[" + key + ", " + value + "]";
    }
  }
}
```

```java
import java.util.LinkedList;

public class MyHashMap<K, V> implements MyMap<K, V> {
  // Define the default hash table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load factor used in the hash table
  private float loadFactorThreshold;

  // The number of entries in the map
  private int size = 0;

  // Hash table is an array with each cell that is a linked list
  LinkedList<MyMap.Entry<K,V>>[] table;

  /** Construct a map with the default capacity and load factor */
  public MyHashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }
```

```java
/** Construct a map with the specified initial capacity and
 * default load factor */
public MyHashMap(int initialCapacity) {
  this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a map with the specified initial capacity
 * and load factor */
public MyHashMap(int initialCapacity, float loadFactorThreshold) {
  if (initialCapacity > MAXIMUM_CAPACITY)
    this.capacity = MAXIMUM_CAPACITY;
  else
    this.capacity = trimToPowerOf2(initialCapacity);

  this.loadFactorThreshold = loadFactorThreshold;
  table = new LinkedList[capacity];
}

@Override /** Remove all of the entries from this map */
public void clear() {
  size = 0;
  removeEntries();
}

@Override /** Return true if the specified key is in the map */
public boolean containsKey(K key) {
  return get(key) != null;
}
```

```
@Override /** Return true if this map contains the value */
public boolean containsValue(V value) {
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        if (entry.getValue().equals(value))
          return true;
    }
  }
  return false;
}

@Override /** Return a set of entries in the map */
public java.util.Set<MyMap.Entry<K,V>> entrySet() {
  java.util.Set<MyMap.Entry<K, V>> set =
    new java.util.HashSet<MyMap.Entry<K, V>>();

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry);
    }
  }
  return set;
}
```

```
@Override /** Return the value that matches the specified key */
public V get(K key) {
  int bucketIndex = hash(key.hashCode());
  if (table[bucketIndex] != null) {
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key))
        return entry.getValue();
  }
  return null;
}

@Override /** Return true if this map contains no entries */
public boolean isEmpty() {
  return size == 0;
}

@Override /** Return a set consisting of the keys in this map */
public java.util.Set<K> keySet() {
  java.util.Set<K> set = new java.util.HashSet<K>();

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry.getKey());
    }
  }
  return set;
}
```

```
@Override /** Add an entry (key, value) into the map */
public V put(K key, V value) {
  if (get(key) != null) { // The key is already in the map
    int bucketIndex = hash(key.hashCode());
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
        V oldValue = entry.getValue();
        // Replace old value with new value
        entry.value = value;
        // Return the old value for the key
        return oldValue;
      }
  }
  // Check load factor
  if (size >= capacity * loadFactorThreshold) {
    if (capacity == MAXIMUM_CAPACITY)
      throw new RuntimeException("Exceeding maximum capacity");
    rehash();
  }
  int bucketIndex = hash(key.hashCode());
  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] == null) {
    table[bucketIndex] = new LinkedList<Entry<K, V>>();
  }
  // Add a new entry (key, value) to hashTable[index]
  table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));
  size++; // Increase size
  return value;
}
```

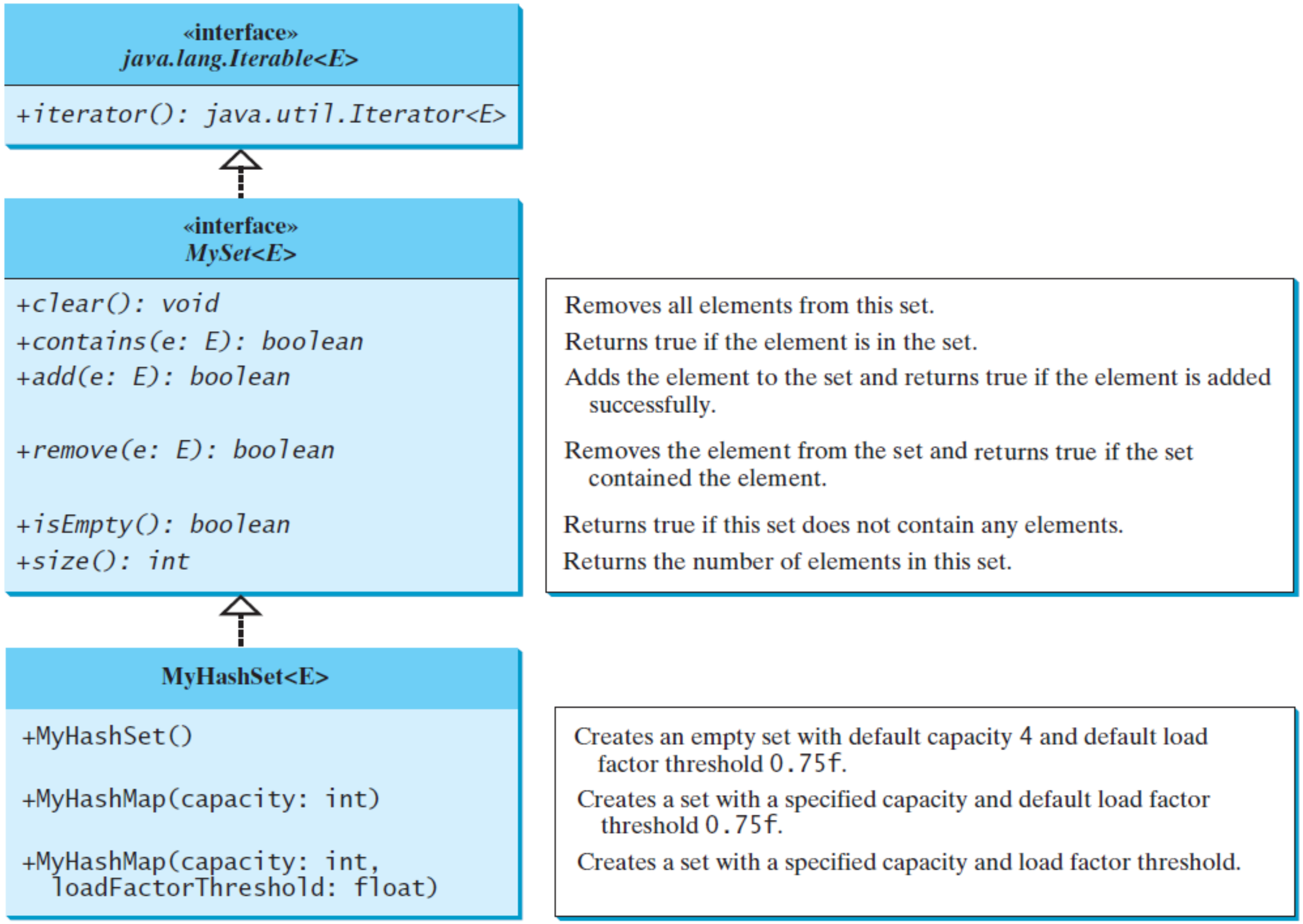
```
@Override /** Remove the entries for the specified key */
public void remove(K key) {
  int bucketIndex = hash(key.hashCode());
  // Remove the first entry that matches the key from a bucket
  if (table[bucketIndex] != null) {
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
        bucket.remove(entry);
        size--; // Decrease size
        break; // Remove just one entry that matches the key
      }
  }
}

@Override /** Return the number of entries in this map */
public int size() {
  return size;
}

@Override /** Return a set consisting of the values in this map */
public java.util.Set<V> values() {
  java.util.Set<V> set = new java.util.HashSet<V>();
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry.getValue());
    }
  }
  return set; }
```

```java
/** Hash function */
private int hash(int hashCode) {
  return supplementalHash(hashCode) & (capacity - 1);
}

/** Ensure the hashing is evenly distributed */
private static int supplementalHash(int h) {
  h ^= (h >>> 20) ^ (h >>> 12);
  return h ^ (h >>> 7) ^ (h >>> 4);
}

/** Return a power of 2 for initialCapacity */
private int trimToPowerOf2(int initialCapacity) {
  int capacity = 1;
  while (capacity < initialCapacity) {
    capacity <<= 1;
  }
  return capacity;
}

/** Remove all entries from each bucket */
private void removeEntries() {
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      table[i].clear();
    }
  }
}
```

```java
  /** Rehash the map */
  private void rehash() {
    java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
    capacity <<= 1; // Double capacity
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0; // Reset size to 0
    for (Entry<K, V> entry: set) {
      put(entry.getKey(), entry.getValue()); // Store to new table
    }
  }

  @Override
  public String toString() {
    StringBuilder builder = new StringBuilder("[");

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null && table[i].size() > 0)
        for (Entry<K, V> entry: table[i])
          builder.append(entry);
    }
    builder.append("]");
    return builder.toString();
  }
}
```

```java
public class TestMyHashMap {
  public static void main(String[] args) {
    // Create a map
    MyMap<String, Integer> map = new MyHashMap<String, Integer>();
    map.put("Smith", 30);
    map.put("Anderson", 31);
    map.put("Lewis", 29);
    map.put("Cook", 29);
    map.put("Smith", 65);

    System.out.println("Entries in map: " + map);

    System.out.println("The age for " + "Lewis is " +
      map.get("Lewis"));

    System.out.println("Is Smith in the map? " +
      map.containsKey("Smith"));
    System.out.println("Is age 33 in the map? " +
      map.containsValue(33));

    map.remove("Smith");
    System.out.println("Entries in map: " + map);

    map.clear();
    System.out.println("Entries in map: " + map);
  }
}
```

Implementing Set Using Hashing



```
public interface MySet<E> extends java.lang.Iterable<E> {
  /** Remove all elements from this set */
  public void clear();

  /** Return true if the element is in the set */
  public boolean contains(E e);

  /** Add an element to the set */
  public boolean add(E e);

  /** Remove the element from the set */
  public boolean remove(E e);

  /** Return true if the set contains no elements */
  public boolean isEmpty();

  /** Return the number of elements in the set */
  public int size();
}
```

```java
import java.util.LinkedList;

public class MyHashSet<E> implements MySet<E> {
  // Define the default hash table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load factor threshold used in the hash table
  private float loadFactorThreshold;

  // The number of elements in the set
  private int size = 0;

  // Hash table is an array with each cell that is a linked list
  private LinkedList<E>[] table;

  /** Construct a set with the default capacity and load factor */
  public MyHashSet() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }
```

```
/** Construct a set with the specified initial capacity and
 * default load factor */
public MyHashSet(int initialCapacity) {
  this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a set with the specified initial capacity
 * and load factor */
public MyHashSet(int initialCapacity, float loadFactorThreshold) {
  if (initialCapacity > MAXIMUM_CAPACITY)
    this.capacity = MAXIMUM_CAPACITY;
  else
    this.capacity = trimToPowerOf2(initialCapacity);

  this.loadFactorThreshold = loadFactorThreshold;
  table = new LinkedList[capacity];
}

@Override /** Remove all elements from this set */
public void clear() {
  size = 0;
  removeElements();
}

@Override /** Return true if the element is in the set */
public boolean contains(E e) {
  int bucketIndex = hash(e.hashCode());
  if (table[bucketIndex] != null) {
    LinkedList<E> bucket = table[bucketIndex];
    for (E element: bucket)
```

```
      if (element.equals(e))
        return true;
    }
    return false;
  }

  @Override /** Add an element to the set */
  public boolean add(E e) {
    if (contains(e)) // Duplicate element not stored
      return false;
    if (size + 1> capacity * loadFactorThreshold) {
      if (capacity == MAXIMUM_CAPACITY)
        throw new RuntimeException("Exceeding maximum capacity");
      rehash();
    }
    int bucketIndex = hash(e.hashCode());
    // Create a linked list for the bucket if it is not created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<E>();
    }
    // Add e to hashTable[index]
    table[bucketIndex].add(e);
    size++; // Increase size
    return true;
  }

  @Override /** Remove the element from the set */
  public boolean remove(E e) {
    if (!contains(e))
      return false;
```

```java
    int bucketIndex = hash(e.hashCode());
    // Create a linked list for the bucket if it is not created
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      for (E element: bucket)
        if (e.equals(element)) {
          bucket.remove(element);
          break;
        }
    }
    size--; // Decrease size
    return true;
  }

  @Override /** Return true if the set contains no elements */
  public boolean isEmpty() {
    return size == 0;
  }

  @Override /** Return the number of elements in the set */
  public int size() {
    return size;
  }

  @Override /** Return an iterator for the elements in this set */
  public java.util.Iterator<E> iterator() {
    return new MyHashSetIterator(this);
  }
```

```
/** Inner class for iterator */
private class MyHashSetIterator implements java.util.Iterator<E> {
  // Store the elements in a list
  private java.util.ArrayList<E> list;
  private int current = 0; // Point to the current element in list
  private MyHashSet<E> set;

  /** Create a list from the set */
  public MyHashSetIterator(MyHashSet<E> set) {
    this.set = set;
    list = setToList();
  }

  @Override /** Next element for traversing? */
  public boolean hasNext() {
    if (current < list.size())
      return true;
    return false;
  }

  @Override /** Get current element and move cursor to the next */
  public E next() {
    return list.get(current++);
  }

  @Override /** Remove the current element and refresh the list */
  public void remove() {
    // Delete the current element from the hash set
    set.remove(list.get(current));
    list.remove(current); // Remove current element from the list
  }
}
```

```java
/** Hash function */
private int hash(int hashCode) {
  return supplementalHash(hashCode) & (capacity - 1);
}

/** Ensure the hashing is evenly distributed */
private static int supplementalHash(int h) {
  h ^= (h >>> 20) ^ (h >>> 12);
  return h ^ (h >>> 7) ^ (h >>> 4);
}

/** Return a power of 2 for initialCapacity */
private int trimToPowerOf2(int initialCapacity) {
  int capacity = 1;
  while (capacity < initialCapacity) {
    capacity <<= 1;
  }
  return capacity;
}

/** Remove all e from each bucket */
private void removeElements() {
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      table[i].clear();
    }
  }
}
```

```
/** Rehash the set */
private void rehash() {
  java.util.ArrayList<E> list = setToList(); // Copy to a list
  capacity <<= 1; // Double capacity
  table = new LinkedList[capacity]; // Create a new hash table
  size = 0; // Reset size

  for (E element: list) {
    add(element); // Add from the old table to the new table
  }
}

/** Copy elements in the hash set to an array list */
private java.util.ArrayList<E> setToList() {
  java.util.ArrayList<E> list = new java.util.ArrayList<E>();
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      for (E e: table[i]) {
        list.add(e);
      }
    }
  }
  return list;
}
```

```
 @Override
  public String toString() {
    java.util.ArrayList<E> list = setToList();
    StringBuilder builder = new StringBuilder("[");

    // Add the elements except the last one to the string builder
    for (int i = 0; i < list.size() - 1; i++) {
      builder.append(list.get(i) + ", ");
    }

    // Add the last element in the list to the string builder
    if (list.size() == 0)
      builder.append("]");
    else
      builder.append(list.get(list.size() - 1) + "]");

    return builder.toString();
  }
}
```

```java
public class TestMyHashSet {
  public static void main(String[] args) {
    // Create a MyHashSet
    MySet<String> set = new MyHashSet<String>();
    set.add("Smith");
    set.add("Anderson");
    set.add("Lewis");
    set.add("Anderson");
    set.add("Cook");
    set.add("Smith");
    set.add("Cook");
    set.add("Smith");

    System.out.println("Elements in set: " + set);
    System.out.println("Number of elements in set: " + set.size());
    System.out.println("Is Smith in set? " + set.contains("Smith"));

    set.remove("Smith");
    System.out.print("Names in set in uppercase are ");
    for (String s: set)
      System.out.print(s.toUpperCase() + " ");

    set.clear();
    System.out.println("\nElements in set: " + set);
  }
}
```