Hashing

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

Objectives

- To understand what ***hashing*** is and what hashing is used for
- To **obtain the hash code for an object and design the hash function to map a key to an index**
- To **handle collisions using open addressing**
  - To know the differences among ***linear probing***, ***quadratic probing***, and ***double hashing***
- To **handle collisions using separate chaining**
- To understand the **load factor** and the need for **rehashing**
- To implement **`MyHashMap`** using hashing
- To implement **`MyHashSet`** using hashing

Why hashing?

- Hashing is used for sets and maps:
  - A ***set*** is a container that contains unique elements (no order considered)
  - A ***map*** (***dictionary***, ***hash table***, or ***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 language dictionary can be stored in a map, where the words are the keys and the definitions of the words or the synonyms are the values
- **O(1)** time to search, insert, and delete an element in a set or a map vs. **O(log n)** time in a well-balanced search tree

How is Hashing implemented?

- Remember **Arrays**: If you know the index of an element in the array, you can retrieve/update the element using the **index** in **O(1)** time
- So, to implement a map, can we store the values in an array and use the key as the index to find/set the value**?**
  - The answer is yes if you can **map the 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 the technique that puts and retrieves the value using the index obtained from key in/from the hash table

Hash Functions and Hash Codes

- A typical ***hash function*** first converts a search key to an **integer** value called a ***hash code***, then **compresses** the hash code into an index to the hash table
- Examples of hash functions:
  - Java's root class **`Object`** has a **`hashCode`** method, which returns an integer hash code -- by default, the method returns the **memory address** for the object (this is the decimal value of what **`toString`** prints in hex)

```
Object o = new Object();
System.out.println(o.hashCode());
    366712642
System.out.println(o); // o.toString()
    java.lang.Object@15db9742
```

$$366712642 = (15db9742)_H$$

Hash Functions and Hash Codes

- You should override the **`hashCode`** method whenever the **`equals`** method is overridden to **ensure that two equal objects return the same hash code**

  - Two unequal objects may have the same hash code (called a *collision*), but you should implement the **`hashCode`** method to avoid too many such cases

Hash Codes for Primitive Types

- The types **`byte`**, **`short`**, **`char`** and **`int`** are simply casted into **`int`**
- For a search key of the type **`float`**, return an **`int`** value whose bit representation is the same as the bit representation for the floating number
  - uses **`Float.floatToIntBits(key)`** to get the hash code

```
Float f = new Float(1.23);
System.out.println(f.hashCode());
//1067282596
System.out.println(Float.floatToIntBits(1.23f));
//1067282596
```

Hash Codes for Primitive Types

- For a search key of the type **`long`**, simply casting it to **`int`** would not be a good choice, because all keys that differ in only the first 32 bits will have the same hash code

```
long key = 9876543210L;
Long l = new Long(key);
System.out.println((int)key);
    //1286608618
System.out.println((int)1286608618);
    //1286608618 <- not a good hash because collision
```

  - Divide the 64 bits into two halves and perform the **exclusive or** operation to combine the two halves (this process is called *folding*)

```
    hashCode = (int)(key ^ (key >> 32));
        for example: 1010110 ^ 0110111 yields 1100001
System.out.println( (int)(key ^ (key >> 32)) );
    // 1286608616
System.out.println(l.hashCode());
    // 1286608616
```

Hash Codes for Primitive Types

- For a key of the type **`double`**, first convert it to a **`long`** value using the **`Double.doubleToLongBits`** method, and then perform a folding:

```
double key = 1.23;
System.out.println((new Double(key)).hashCode());
      //1158867386
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
System.out.println(hashCode);
      //1158867386
```

Hash Codes for Strings

- An intuitive approach is to sum the Unicode of all characters as the hash code for the string
  - This approach may work if two search keys don't contain the same letters, but it fails otherwise (e.g., ***Fodor*** and ***Frodo)***
- A better approach is to generate a hash code that takes the position of characters into consideration:

$$s_0 * b^{(n-1)} + s_1 * b^{(n-2)} + \ldots + s_{n-1}$$

where s_i is **`s.charAt(i)`** and b is some positive integer

  - Using Horner's rule for polynomial evaluation, the hash code can be calculated efficiently as follows:

$$(\ldots((s_0 * b + s_1)b + s_2)b + \ldots + s_{n-2})b + s_{n-1}$$

  - This expression is a polynomial for some positive ***b***, so this is called a ***polynomial hash code***

Hash Codes for Strings

- The hash code computation can cause an overflow for long strings, but arithmetic overflows are ignored in Java

```
int i = 9999 * 9999 * 9999 * 9999 * 9999 * 9999 * 9999 * 9999 * 9999 * 9999;
System.out.println( i ); // 426346081
```

- Java needed to choose an appropriate value ***b*** to minimize collisions
  - Experiments show that good choices for b are **`31`**, **`33`**, **`37`**, **`39`**, and **`41`**
    - In the String class, the **`hashCode`** is overridden using the polynomial hash code with *b* being **`31`**
      - multiplications are also avoided because:

N * 31 = N * 32 - N = N<<5 - N

Hash Functions and Hash Codes

- During the execution of a program, invoking the **`hashCode`** method multiple times on the same object returns the same integer, provided that the object's data are not changed

```
ArrayList l = new ArrayList();
System.out.println(l.hashCode());
   // 1
System.out.println(l.hashCode());
   // 1
l.add(1);
System.out.println(l.hashCode());
   // 32
```

`ArrayList hashcode` changes when: the capacity changes, whenever you add or remove an element or mutate one of the elements in a way that changes its hashCode. Implementation:

```
public int hashCode() {
    int hashCode = 1;
    for (E e : this)
        hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
    return hashCode;
}
```

Compressing Hash Codes

- The hash code for a key can be a large integer that is out of the range for the hash-table index
  - we need to scale it down to fit in the index's range
- Assume the index for a hash table is between **0** and **N-1**

  -- the most common way to scale an integer is

  h(hashCode)=hashCode % N

Compressing Hash Codes

- In the Java API implementation for **`java.util.HashMap`**, **`N`** is set to a value of the power of **`2`** because then

```
h(hashCode) = hashCode % N
```

is the same as

```
h(hashCode) = hashCode & (N - 1)
```

Bitwise conjunction

`&` can be performed much faster than the **`%`** operator (*)

- For example, assume **`N = 4`** and **`hashCode = 11`**

```
h(hashCode) = hashCode % N
                    11 % 4 = 3
h(hashCode) = hashCode & (N - 1)
            = hashCode & 3
              (01011)2 & (00011)2 = (11)2 = 3
```

Note (*): The bitwise operations are much faster than the multiplication, division, and remainder operations -- you should replace these operations with the bitwise operations whenever possible

Compressing Hash Codes

- To ensure that the hashing is evenly distributed, a supplemental hash function is also used along with the primary hash function in the implementation of **`java.util.HashMap`**

```
private static int supplementalHash(int h) {
   h ^= (h >>> 20) ^ (h >>> 12);
   return h ^ (h >>> 7) ^ (h >>> 4);
}
```

- The complete hash function is defined as:

`h(hashCode) = supplementalHash(hashCode) % N`

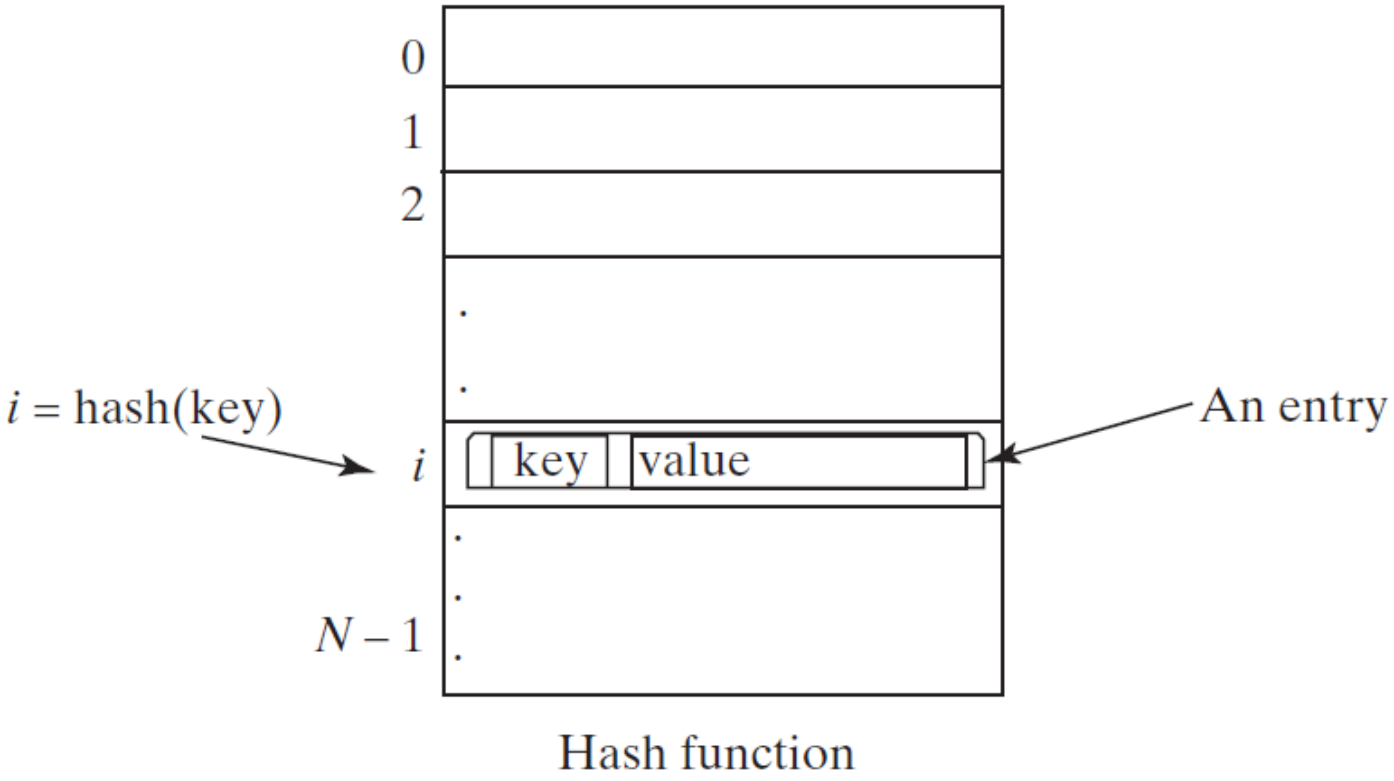
which is the same as

`h(hashCode) = supplementalHash(hashCode) & (N - 1)`

since **`N`** is a power of **`2`**

Hash Function and Hash Codes

- In summary: 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:

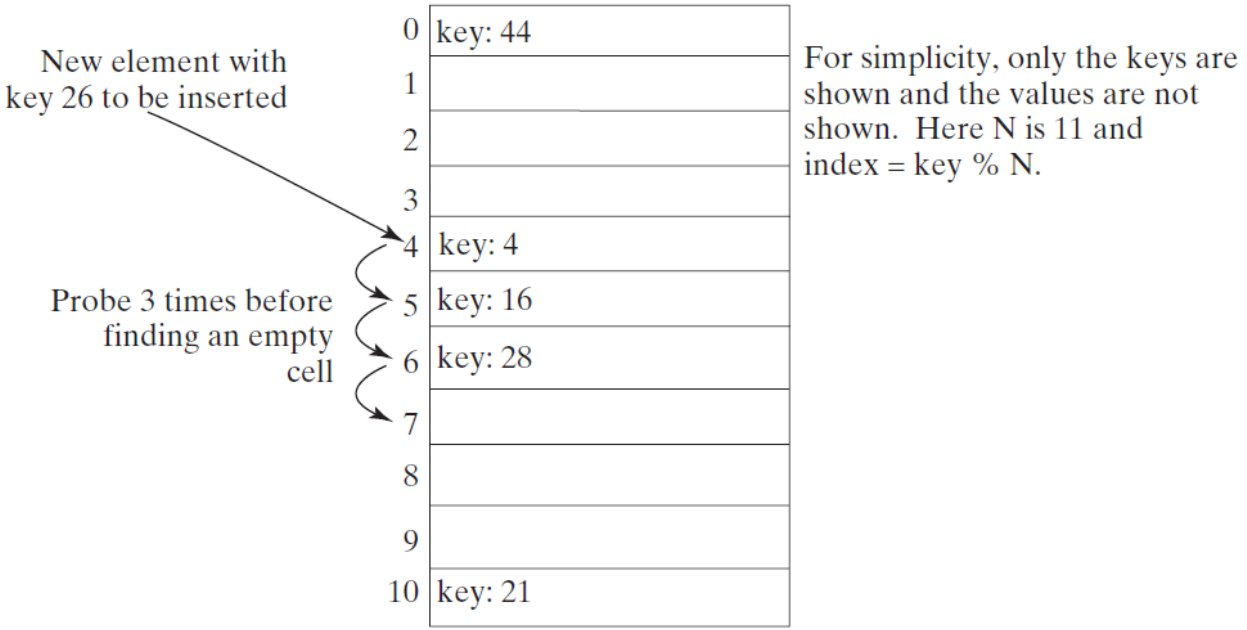


Handling Collisions

- A ***hashing collision*** (or *collision*) occurs when two different keys are mapped to the same index in a hash table
- There are two ways for handling collisions: ***open addressing*** and ***separate chaining***
  - ***Open addressing*** is the process of finding an open location in the hash table in the event of a collision
    - Open addressing has several variations: ***linear probing***, ***quadratic probing*** and ***double hashing***
  - ***Separate chaining*** places all entries with the same hash index into the same location in a list

Linear Probing

- When a collision occurs during the **insertion** of an entry to a hash table, linear probing finds the next available location sequentially
  - If a collision occurs at **`hashTable[key % N]`**, check whether **`hashTable[(key+1) % N]`** is available
  - If not, check **`hashTable[(key+2) % N]`** and so on, until an available cell is found

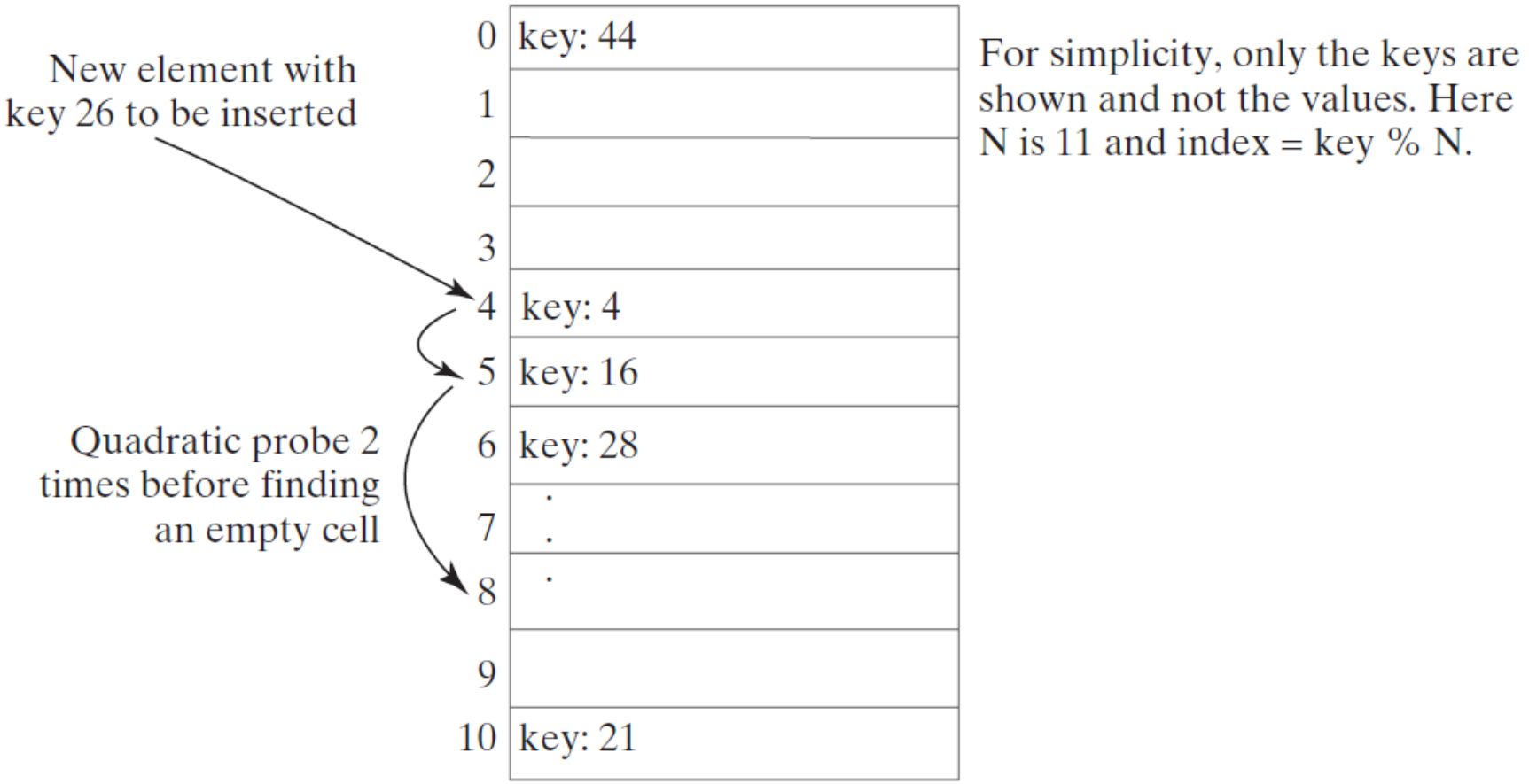


Linear Probing

- To **remove** an entry from the hash table, search the entry that matches the key
  - If the entry is found, place a special marker ***marked*** to denote that the entry is deleted, but available for insertion of other values
    - Each cell in the hash table has three possible states: ***occupied***, ***marked***, or ***empty***
      - a marked cell is also available for insertion
- Linear probing tends to cause groups of consecutive cells in the hash table to be occupied -- each group is called a ***cluster***
  - Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry
  - As clusters grow in size, they may merge into even larger clusters, further slowing down the search time
  - This is a big disadvantage of linear probing

Quadratic Probing

- Quadratic probing looks at the cells at indices **`(key+j²)%N`** for **`j>=0`**, that is, **`key%N`**, **`(key+1)%N`**, **`(key+4)%N`**, **`(key+9)%N`**, **`(key+16)%N`**, and so on



Quadratic Probing

- Quadratic probing works in the same way as linear probing for retrieval, insertion and deletion, except for the change in the search sequence
- ***Quadratic probing*** can avoid the clustering problem in linear probing **for consecutive keys**
  - It still has its own clustering problem, called ***secondary clustering*** for **keys that collide with the occupied entry** using the same probe sequence
- Linear probing guarantees that an available cell can be found for insertion as long as the table is not full
  - there is no such guarantee for quadratic probing (it can have a cycle that skips the same positions)

Double Hashing

- Both linear probing and quadratic probing add an increment to the index **`key`**: **`1`** for linear probing and **`j`**^{2} for quadratic probing **independent of the keys**
  - ***Double hashing*** uses a secondary hash function **`h'(key)`** on the keys to **determine the increments to avoid the clustering** problem
    - Double hashing looks at the cells at indices **`(key+j*h'(key))%N`** for **`j>=0`**, that is,

      `key%N`, **`(key+h'(key))%N`**, **`(key+2* h'(key))%N`**, **`(key+3*h'(key))%N`**, and so on

Double Hashing

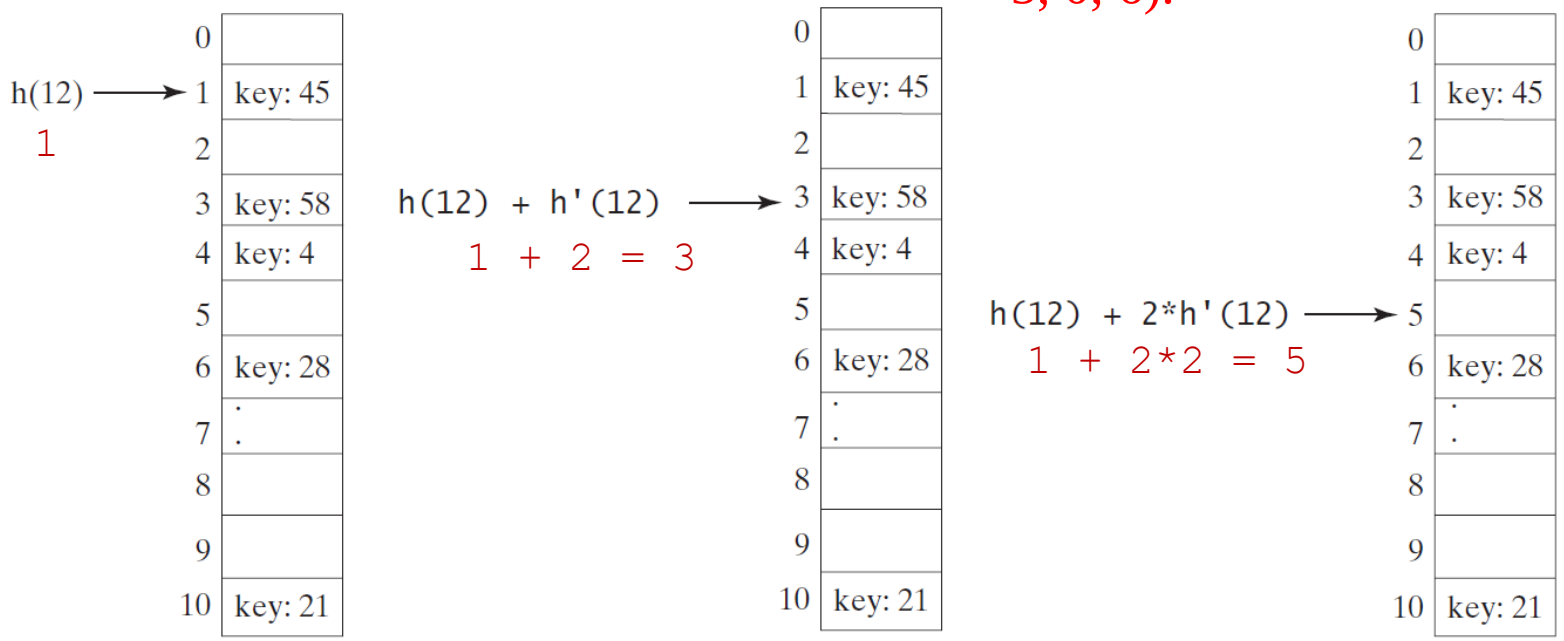
- For example, let the primary hash function **h** and secondary hash function **h'** on a hash table of size **11** be defined as follows:

```
h(key) = key % 11;
h'(key) = 7 - key % 7;
```

- For a search key of 12, we have

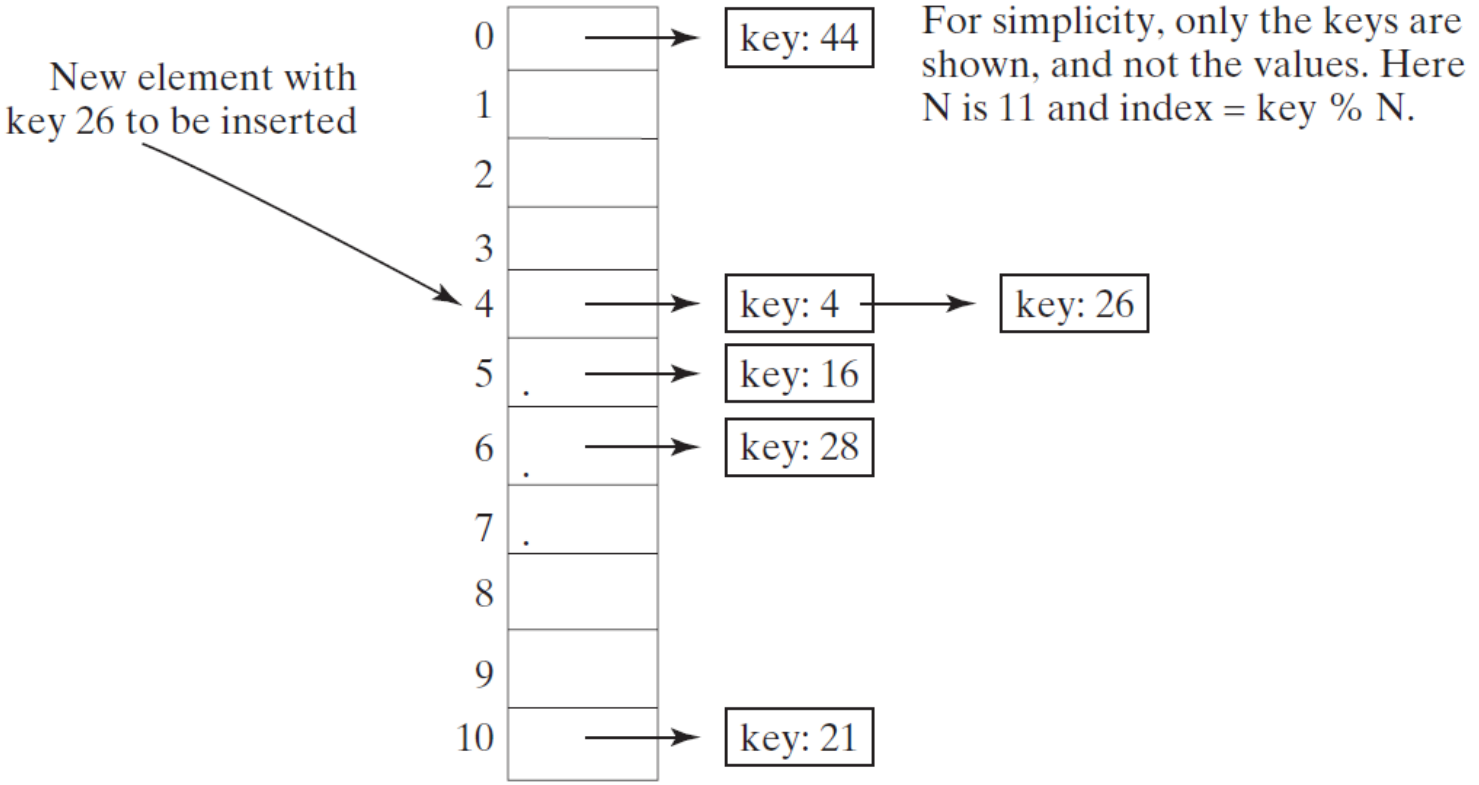
```
h(12) = 12 % 11 = 1;
h'(12) = 7 - 12 % 7 = 2;
```

The indices of this probe sequence are as follows: 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10.
For another key (e.g., 1), the skips will be different (1,7, 2, 8, 3, 10, 5, 0, 6).



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:



Handling Collisions Using Separate Chaining

- Collision in **`java.util.HashMap`** is possible because hash function uses **`hashCode`**() of key object which doesn't guarantee different hash codes for different objects.
- Up to Java 8, **`HashMap`** handles collision by using a linked list to store map entries ended up in same array location or bucket location.
  - From Java 8 onwards, **`HashMap`** uses a balanced tree in place of linked list to handle frequently hash collisions.
    - By switching from linked list to balanced tree for handling collision, the iteration order of **`HashMap`** changed, which is Ok because **`HashMap`** doesn't provide any guarantee on iteration order.
    - Legacy class **`Hashtable`** which exists in JDK from Java 1 does not use the balanced binary tree to handle frequent hash collision to keep its iteration order intact. This was decided to avoid breaking many legacy Java application which depends upon iteration order of **`Hashtable`**.

Load Factor and Rehashing

- Load factor λ (lambda) is the ratio of the number of elements to the size of the hash table

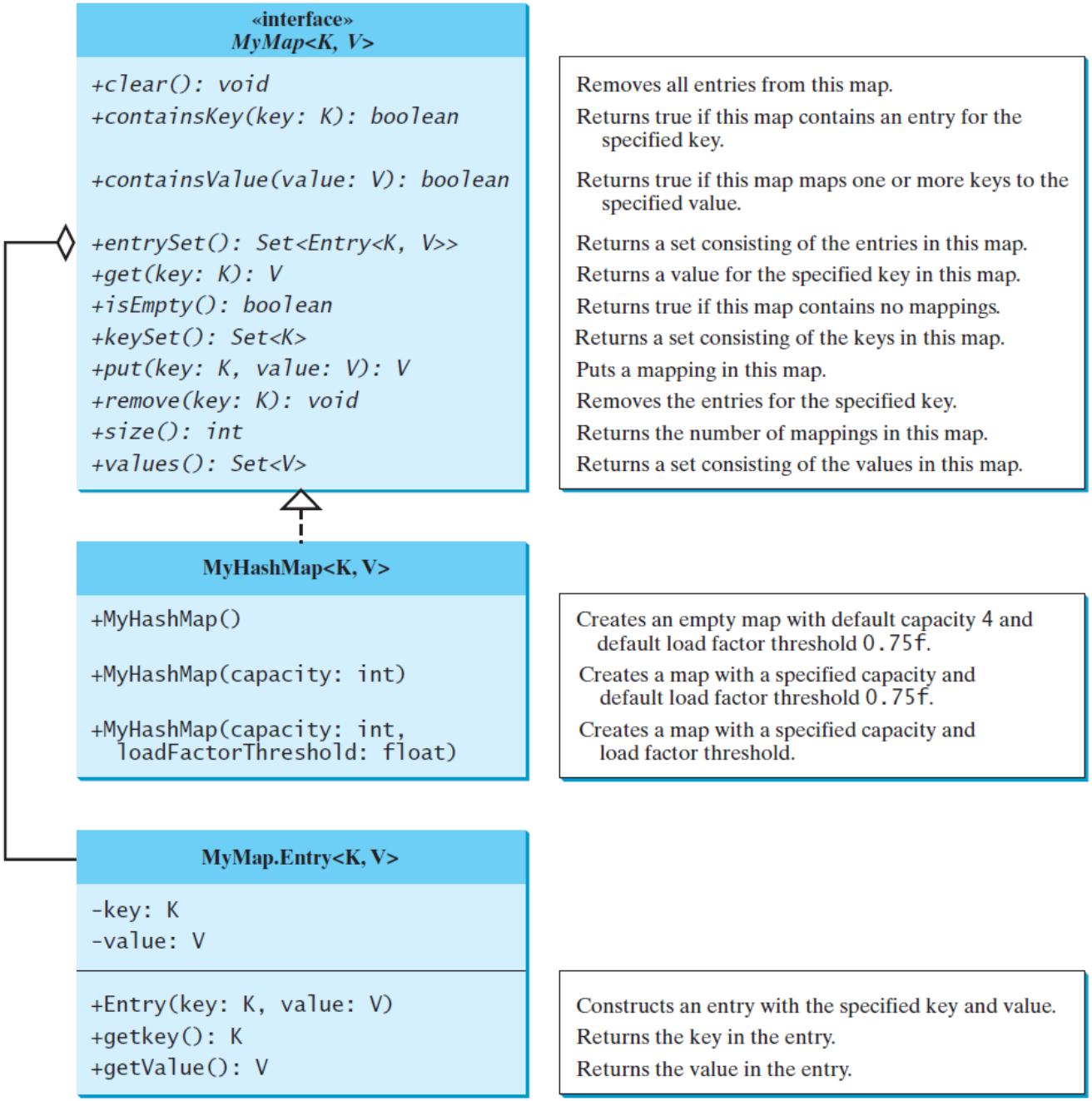
$$\lambda = n/N$$

where **n** denotes the number of elements and **N** the number of locations in the hash table

  - For the open addressing scheme, λ is between **0** and **1**
    - If the hash table is empty, then $\lambda = 0$
    - If the hash table is full, then $\lambda = 1$
  - For the separate chaining scheme, λ can be any value

Load Factor and Rehashing

- As λ increases, the probability of a collision increases
  - Because the ***load factor*** measures how full a hash table is
  - You should maintain the load factor under `0.5` for the open addressing schemes and under `0.9` for the separate chaining scheme
    - In the implementation of the **`java.util.HashMap`** class in the Java API, the threshold `0.75` is used
- If the load factor is **exceeded**, we **increase** the hash-table size and **reload** the entries into a new larger hash table -> this is called ***rehashing***
  - We need to change the hash functions, since the hash-table size has been changed
  - To reduce the likelihood of rehashing, since it is costly, you should at least double the hash-table size



```
public interface MyMap<K, V> {
  /** Add an entry (key, value) into the map */
  public V put(K key, V value);

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

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

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

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

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

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

  /** Remove all of the entries from this map */
  public void clear();

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

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

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

  /** 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 + "]";
    }
  }
}
```

```
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;

  // Hash table is an array with each cell that is a linked list
  // we will implement the separate chaining scheme
  LinkedList<MyMap.Entry<K,V>>[] table;

  // 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;
```

```
/** 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);
  table = new LinkedList[capacity];
  this.loadFactorThreshold = loadFactorThreshold;
}

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

/** 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 default capacity and load factor */
public MyHashMap() {
  this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
}
```

```
@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;
}
```

```
/** 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();
}
```

```
@Override /** Return the value that matches the specified key */
public V get(K key) {
  // we implement the separate chaining scheme
  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 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 /** Return true if this map contains no entries */
public boolean isEmpty() {
  return size == 0;
}
```

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

```
@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 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;
}
```

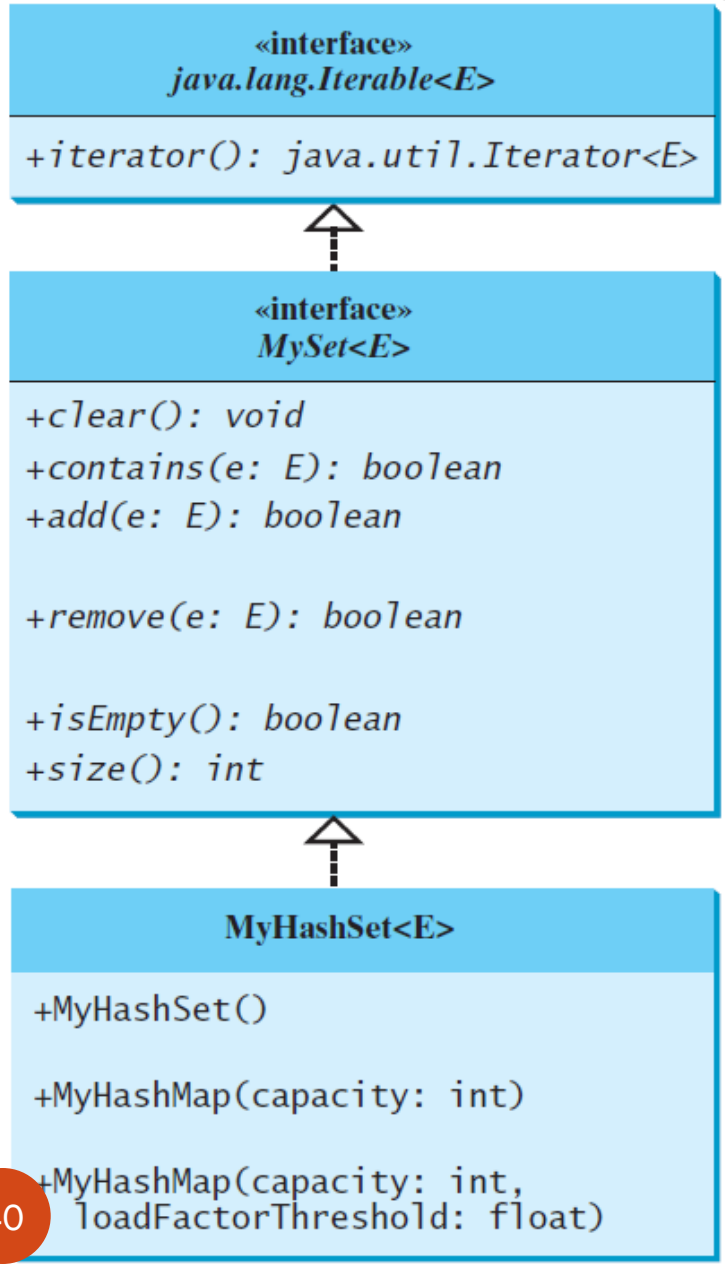
```
  /** Hash function */
  private int hash(int hashCode) {
    return supplementalHash(hashCode) & (capacity - 1);
  }

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

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

  @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;
  }
}
```

```
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);
  }
}
```

Output:

```
Entries in map: [[Anderson, 31][Smith, 65][Lewis, 29][Cook, 29]]
The age for Lewis is 29
Is Smith in the map? true
Is age 33 in the map? false
Entries in map: [[Anderson, 31][Lewis, 29][Cook, 29]]
Entries in map: []
```

Implementing Set Using Hashing



Removes all elements from this set.

Returns true if the element is in the set.

Adds the element to the set and returns true if the element is added successfully.

Removes the element from the set and returns true if the set contained the element.

Returns true if this set does not contain any elements.

Returns the number of elements in this set.

Creates an empty set with default capacity 4 and default load factor threshold `0.75f`.

Creates a set with a specified capacity and default load factor threshold `0.75f`.

Creates a set with a specified capacity and load factor threshold.

```
public interface MySet<E> extends java.lang.Iterable<E>{

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

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

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

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

  /** Remove all elements from this set */
  public void clear();

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

```
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;

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

  // 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;

  /** 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 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];
}

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

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

```
@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 /** 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 /** 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 /** Remove the element from the set */
public boolean remove(E e) {
  if (!contains(e))
    return false;
  int bucketIndex = hash(e.hashCode());
  LinkedList<E> bucket = table[bucketIndex];
    for (E element: bucket)
      if (e.equals(element)) {
        bucket.remove(element);
        break;
      }
  size--; // Decrease size
  return true;
}

/** 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 /** 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
  }
}
```

```
/** 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
  }

 @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();
  }
}
```

```
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);
  }
}
```

Output:

```
Elements in set: [Anderson, Smith, Lewis, Cook]
Number of elements in set: 4
Is Smith in set? true
Names in set in uppercase are ANDERSON LEWIS COOK
Elements in set: []
```