Sets and Maps

Based on Chapter 9 of
Koffmann and Wolfgang

Chapter Outline

- The **`Map`** and **`Set`** interfaces and how to use them
- *Hash coding* and its use in efficient search & retrieval
- Two forms of hash tables:
  - *Open addressing*
  - *Chaining*
  - Their relative benefits and performance tradeoffs
- *Implementing* both hash table forms
- Introduction to implementation of Maps and Sets
- Applying Maps and Sets to previous problems

Sets and the Set Interface

- This part of the Collection hierarchy includes 3 interfaces, 2 abstract classes, and 2 actual classes

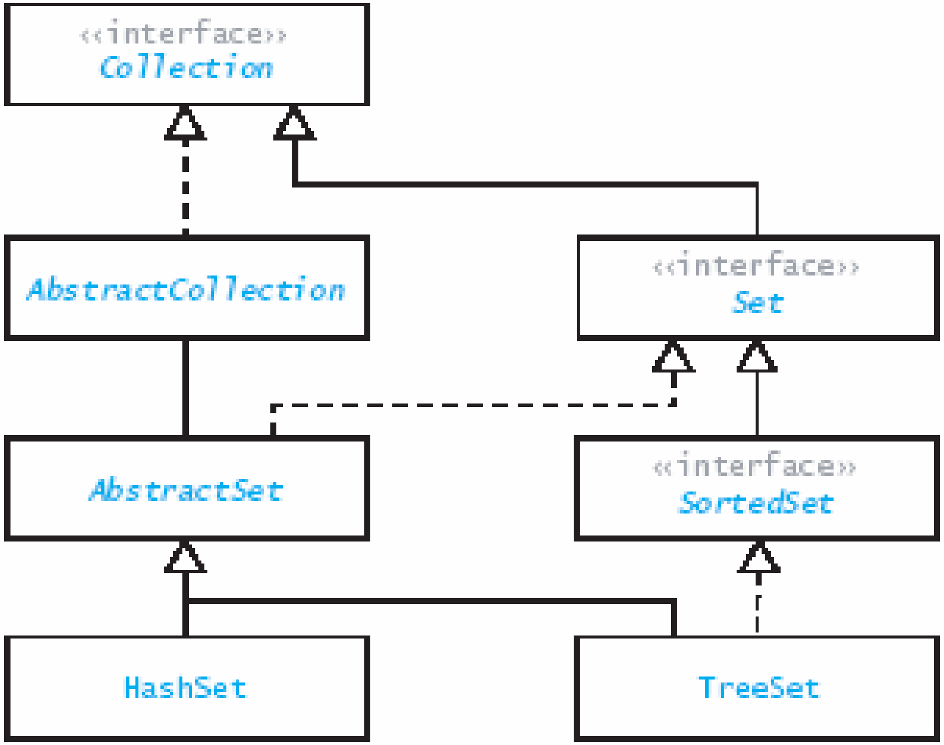


FIGURE 9.1
The Set Interface Part of the Collection Hierarchy

The Set Abstraction

- A *set* is a collection containing no duplicate elements
- Operations on sets include:
  - Testing for membership
  - Adding elements
  - Removing elements
  - Union
  - Intersection
  - Difference
  - Subset

The Set Interface and Methods

```
// element oriented methods

boolean contains (E e) // member test
boolean add (E e)      // enforces no-dups
boolean remove (Object o)

boolean isEmpty ()
int size ()

Iterator<E> iterator ()
```

The Set Interface and Methods (2)

```
// Set/Collection oriented methods
boolean containsAll (Collection<E> c)
  // subset test
boolean addAll    (Collection<E> c)
  // set union
boolean removeAll (Collection<E> c)
  // set difference
boolean retainAll (Collection<E> c)
  // set intersection
```

The Set Interface and Methods (3)

- Constructors enforce the “no duplicates” criterion
- Add methods do not allow duplicates either
- Certain methods are *optional*
  - add, addAll, remove, removeAll, retainAll

Set Example

```
String[] aA = {“Ann”,”Sal”,”Jill”,”Sal”};
String[] aB = {“Bob”,”Bill”,”Ann”,”Jill”};
Set<String> sA = new HashSet<String>();
                 // HashSet implements Set
Set<String> sA2 = new HashSet<String>();
Set<String> sB = new HashSet<String>();
for (String s : aA) {
  sA.add(s); sA2.add(s);
}
for (String s : aB) {
  sB.add(s);
}
```

Set Example (2)

```
...
System.out.println(“The two sets are:\n” +
  sA + “\n” + sB);

sA.addAll(sB);       // union
sA2.retainAll(sB);  // intersection
System.out.println(“Union: “, sA);
System.out.println(“Intersection: “, sA2);
```

Lists vs. Sets

- Sets allow no duplicates
- Sets do not have *positions*, so no **`get`** or **`set`** method
- Set iterator can produce elements in any order

Maps and the `Map` Interface

- Map is related to Set: it is a set of ordered pairs
- Ordered pair: (*key*, *value*)
  - In a given **`Map`**, there are no duplicate *keys*
  - Values may appear more than once
- Can think of key as “mapping to” a particular value
- Maps support efficient organization of information in tables
- Mathematically, these maps are:
  - Many-to-one (not necessarily one-to-one)
  - Onto (every value in the map has a key)

Map Picture

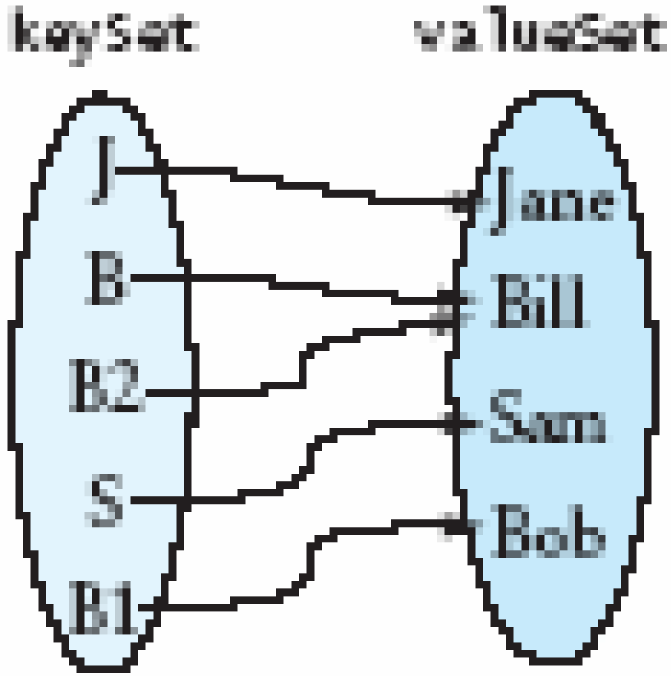


FIGURE 9.2
Example of Mapping

The Map Interface

```
// some methods of java.util.Map<K, V>
//   K is the key type
//   V is the value type
V get (Object key)
  // may return null
V put (K key, V value)
  // returns previous value or null
V remove (Object key)
  // returns previous value or null

boolean isEmpty ()
int size ()
```

Map Example

```
// this builds Map in previous picture
Map<String, String> m =
  new HashMap<String, String>();
  // HashMap is an implementation of Map
m.put(“J” , “Jane”);
m.put(“B” , “Bill”);
m.put(“S” , “Sam” );
m.put(“B1”, “Bob” );
m.put(“B2”, “Bill”);
//
System.out.println(“B1->“ + m.get(“B1”));
System.out.println(“Bill->”+m.get(“Bill));
```

Word Index Revisited

```
// Idea: enter word once
//   with list of lines on which it occurs
... inner loop: word has the word ...
  // get list of lines for this word
  ArrayList<Integer> lines =
    index.get(word);
  if (lines == null) {
    lines = new ArrayList<Integer>();
    index.put(word, lines);
  }
  lines.add(lineNum);
...
```

Hash Tables

- **Goal:** access item given its *key* (not its *position*)
- Therefore, want to *locate it directly from the key*
- In other words, we wish to avoid much searching

- ***Hash tables*** provide this capability
  - ***Constant time*** in the average case! O(1)
  - Linear time in the worst case O(n)

- Searching an array: O(n) Searching BST: O(log n)

Hash Codes

- Suppose we have a table of size N
- A *hash code* is:
  - A number in the range 0 to N-1
  - We compute the hash code from the key
  - You can think of this as a “default position” when inserting, or a “position hint” when looking up
- A *hash function* is a way of computing a hash code
- **Desire:** The set of keys should spread evenly over the N values
- When two keys have the same hash code: *collision*

A Simple Hash Function

- Want to count occurrences of each Character in a file
- There are 2^{16} possible characters, but ...
  - Maybe only 100 or so occur in a given file
- Approach: *hash* character to range 0-199
  - That is, use a hash table of size 200
- A possible *hash function* for this example:

```
int hash = unicodeChar % 200;
```

- Collisions are certainly possible (see later)

Devising Hash Functions

- Simple functions often produce many collisions
  - ... but complex functions may not be good either!
- It is often an empirical process
  - Adding letter values in a string: same hash for strings with same letters in different order
  - Better approach:

```
int hash = 0;
for (int i = 0; i < s.length(); ++i)
  hash = hash * 31 + s.charAt(i);
```

  - This is the hash function used for String in Java

Devising Hash Functions (2)

- The **`String`** hash is good in that:
    - Every letter affects the value
    - The order of the letters affects the value
    - The values tend to be spread well over the integers
- Table size should not be a multiple of 31:
    - Calculate index: **`int index = hash % size;`**
    - For short strings, index depends heavily on the last one or two characters of the string
    - They chose 31 because it is prime, and this is less likely to happen

Devising Hash Functions (3)

- Guidelines for good hash functions:
  - *Spread values evenly*: as if “random”
  - *Cheap to compute*
- Generally, number of possible values >> table size

Open Addressing

- Will consider two ways to organize hash tables
  - Open addressing
  - Chaining
- Open addressing:
  - Hashed items are in a single array
  - Hash code gives position “hint”
  - Handle collisions by checking multiple positions
  - Each check is called a *probe* of the table

Linear Probing

- Probe by <u>*incrementing the index*</u>
- If "fall off end", wrap around to the beginning
  - Take care not to cycle forever!

1. Compute **index** as **`hashCode() % table.length`**
2. if table[index] == null, item is not in the table
3. if table[index] matches item, found item (done)
4. Increment index circularly and go to 2

- Why must we probe repeatedly?
  - hashCode may produce collisions
  - remainder by table.length may produce collisions

Search Termination

Ways to obtain proper termination

- Stop when you come back to your starting point
- Stop after probing N slots, where N is table size
- Stop when you reach the bottom the second time
- Ensure table never full
    - Reallocate when occupancy exceeds threshold

Hash Table Considerations

- Cannot *traverse* a hash table
  - Order of stored values is arbitrary
  - Can use an *iterator* to produce in arbitrary order
- When item is deleted, cannot just set its entry to null
  - Doing so would break probing
  - Must store a “dummy value” instead
  - Deleted items waste space and reduce efficiency
- Use prime number for table size: reduces collisions
- Higher occupancy causes makes for collisions

Hash Table Example

- Table of strings, initial size 5
- Add “Tom”, hash 84274 → 4 Slot 4
- Add “Dick”, hash 2129869 → 4 Slot 0 (wraps)
- Add “Harry”, hash 69496448 → 3 Slot 3
- Add “Sam”, hash 82879 → 4 Slot 1 (wraps)
- Add “Pete”, hash 2484038 → 3 Slot 2 (wraps)

- Note: many lookups will probe a lot!
- Size 11 gives these slots: 3, 5, 10, 5→6, 7

Reducing Collisions By Growing

- Choose a new larger size, e.g., doubling
- (Re)insert non-deleted items into new array
- Install the new array and drop the old

- Similar to reallocating an ArrayList, etc.
  - *But*, elements can move around in reinsertion
  - Hope: rehashing distributes items at least as well

Quadratic Probing

- Linear probing
  - Tends to form long *clusters* of keys in the table
  - This causes longer search chains
- *Quadratic probing* can reduce the effect of clustering
  - Index increments form a *quadratic series*
  - Direct calculation involves multiply, add, remainder
    - Incremental calculation better (in a moment)
  - Probe sequence may not produce all table slots

Quadratic Probing (2)

- Generating the quadratic sequence

  Want: s, s+1^2, s+2^2, s+3^2, s+4^2, etc. (all % length)

  “Trick” to calculate incrementally:

  Initially:

```
int index = ... 1st probe slot ...
int k = -1;
```

  At each iteration:

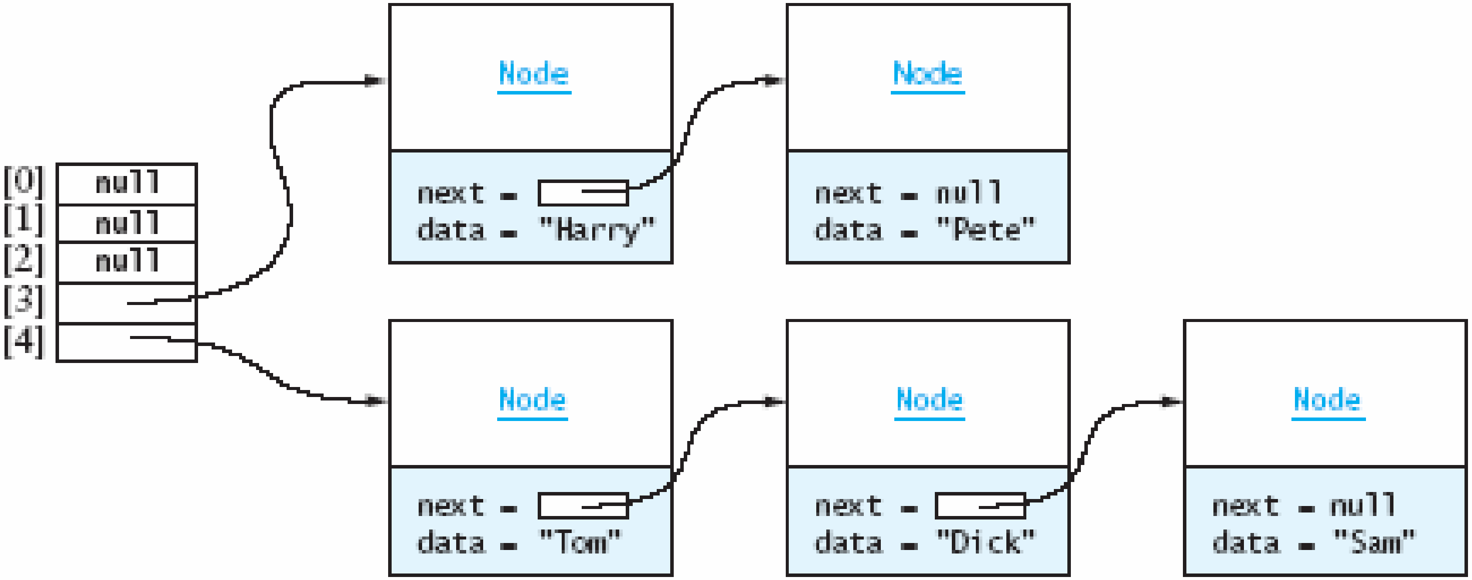
```
k += 2;
index = (index + k) % table.length;
```

Chaining

- *Alternative* to open addressing
- Each table slot references a *linked list*
  - List contains all items that hash to that slot
  - The linked list is often called a *bucket*
  - So sometimes called bucket hashing
- Examines only items with same hash code
- Insertion about as complex
- Deletion is simpler
- Linked list can become long → rehash

Chaining Picture

FIGURE 9.7
Example of Chaining



Two items hashed to bucket 3

Three items hashed to bucket 4

Performance of Hash Tables

- *Load factor* = # filled cells / table size
  - Between 0 and 1
- Load factor has greatest effect on performance
- Lower load factor → better performance
  - Reduce collisions in sparsely populated tables
- Knuth gives expected # probes p for open addressing, linear probing, load factor L: $p = \frac{1}{2}(1 + 1/(1-L))$
  - As L approaches 1, this zooms up
- For chaining, $p = 1 + (L/2)$
  - Note: Here L can be greater than 1!

Performance of Hash Tables (2)

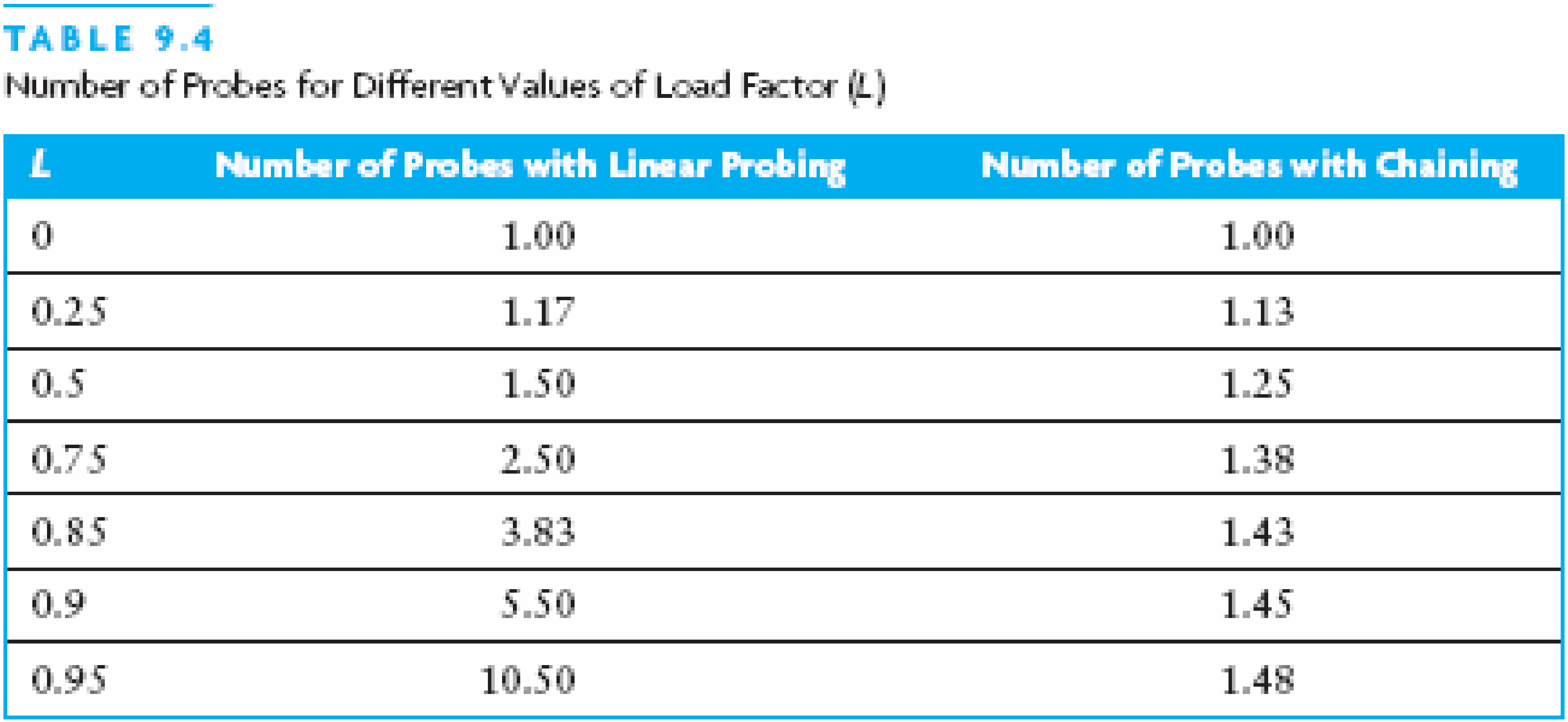


TABLE 9.4

Number of Probes for Different Values of Load Factor (*L*)

L	Number of Probes with Linear Probing	Number of Probes with Chaining
0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Performance of Hash Tables (3)

- Hash table:
  - Insert: average O(1)
  - Search: average O(1)
- Sorted array:
  - Insert: average O(n)
  - Search: average O(log n)
- Binary Search Tree:
  - Insert: average O(log n)
  - Search: average O(log n)
- But balanced trees can *guarantee* O(log n)

Performance of Hash Tables (3)

- Hash table:
  - Open addressing space: n/L   e.g., 1.5 to 2 x n
  - Chaining: assuming 4 words per list node (2 header, 1 next, 1 data): n(1+4L)
- Sorted array:
  - Space: n
- Binary Search Tree:
  - Space: 5n (5 words per tree node: 2 header, 1 left, 1 right, 1 data)

Implementing Hash Tables

- Interface **`HashMap`**: used for both implementations
- Class **`Entry`**: simple class for (key, value) pairs
- Class **`HTOpen`**: implements open addressing
- Class **`HTChain`**: implements chaining
- Further implementation concerns

Interface `HashMap<K,V>`

Note: Java API version has many more operations!

```
V get (Object key)
  // may return null
V put (K key, V value)
  // returns previous value; null if none
V remove (Object key)
  // returns previous value; null if none

boolean isEmpty ()
int size ()
```

Class Entry

```
private static class Entry<K, V> {
  private K key;
  private V value;
  public Entry (K key, V value) {
    this.key = key; this.value = value;
  }
  public K getKey () { return key; }
  public V getValue () { return value; }
  public V setValue (V newVal) {
    V oldVal = value;
    value = newVal;
    return oldVal;
  }    }
```

Class HTOpen<K,V>

```
public class HTOpen<K, V>
    implements HashMap<K, V> {
  private Entry<K, V>[] table;
  private static final int INIT_CAP = 101;
  private double LOAD_THRESHOLD = 0.75;
  private int numKeys;
  private int numDeletes;
  // special “marker” Entry
  private final Entry<K, V> DELETED =
    new Entry<K, V>(null, null);
  public HTOpen () {
    table = new Entry[INIT_CAP];
  }
...  // inner class Entry can go here
```

Class HTOpen<K,V> (2)

```
private int find (Object key) {
  int hash = key.hashCode();
  int idx = hash % table.length;
  if (idx < 0) idx += table.length;
  while ((table[idx] != null) &&
         (!key.equals(table[idx].key))) {
    idx++;
    if (idx >= table.length)
      idx = 0;
    // could do above 3 lines as:
    // idx = (idx + 1) % table.length;
  }
  return idx;
}
```

Class `HTOpen<K,V>` (3)

```
public V get (Object key) {
  int idx = find(key);
  if (table[idx] != null)
    return table[idx].value;
  else
    return null;
}
```

Class HTOpen<K,V> (4)

```
public V put (K key, V val) {
  int idx = find(key);
  if (table[idx] == null) {
    table[idx] = new Entry<K,V>(key,val);
    numKeys++;
    double ldFact =   // NOT int divide!
      (double)(numKeys+numDeletes) /
      table.length;
    if (ldFact > LOAD_THRESHOLD) rehash();
    return null;
  }
  V oldVal = table[idx].value;
  table[idx].value = val;
  return oldVal;  }
```

Class HTOpen<K, V> (5)

```
private void rehash () {
  Entry<K, V>[] oldTab = table;
  table = new Entry[2*oldTab.length + 1];
    // the + 1 keeps length odd
  numKeys = 0;
  numDeletes = 0;
  for (int i = 0; i < oldTab.length; ++i){
    if ((oldTab[i] != null) &&
        (oldTab[i] != DELETED)) {
      put(OldTab[i].key, oldTab[i].value);
    }
  }
}
// The remove operation is an exercise
```

Class HTChain<K,V>

```
public class HTChain<K, V>
    implements HashMap<K, V> {
  private LinkedList<Entry<K, V>>[] table;
  private int numKeys;
  private static final int CAPACITY = 101;
  private static final double
    LOAD_THRESHOLD = 3.0;
  // put inner class Entry here
  public HTChain () {
    table = new LinkedList[CAPACITY];
  }
  ...
}
```

Class `HTChain<K,V>` (2)

```
public V get (Object key) {
  int hash = key.hashCode();
  int idx = hash % table.length;
  if (idx < 0) idx += table.length;
  if (table[idx] == null) return null;
  for (Entry<K, V> item : table[idx]) {
    if (item.key.equals(key))
      return item.value;
  }
  return null;
}
```

Class HTChain<K,V> (3)

```
public V put (K key, V val) {
  int hash = key.hashCode();
  int idx = hash % table.length;
  if (idx < 0) idx += table.length;
  if (table[idx] == null)
    table[idx] =
      new LinkedList<Entry<K, V>>();
  for (Entry<K, V> item : table[idx]) {
    if (item.key.equals(key)) {
      V oldVal = item.value;
      item.value = val;
      return oldVal;
    }
  }  // more ....
```

Class `HTChain<K,V>` (4)

```
  // rest of put: “not found” case
  table[idx].addFirst(
    new Entry<K, V>(key, val));
  numKeys++;
  if (numKeys >
       (LOAD_THRESHOLD * table.length))
    rehash();
  return null;
}
// remove and rehash left as exercises
```

Implementation Considerations for Maps and Sets

- Class **`Object`** implements **`hashCode`** and **`equals`**
  - Every class has these methods
  - One may override them when it makes sense to
- **`Object.equals`** compares addresses, not contents
- **`Object.hashCode`** based on address, not contents
- Java recommendation:
  - If you override **`equals`**, then
  - you should also override **`hashCode`**

Example of `equals` and `hashCode`

- Consider class **`Person`** with field **`IDNum`**

```
public boolean equals (Object o) {
  if (!(o instanceof Person))
    return false;
  return IDNum.equals(((Person)o).IDNum);
}
```

- Demands a matching **`hashCode`** method:

```
public int hashCode () {
  // equal objects will have equal hashes
  return IDNum.hashCode();
}
```

Implementing HashSetOpen

- Can use **`HashMap<E,E>`** and pairs (key,key)
  - This is an *adapter class*
- Can use an **`Entry<E>`** inner class
- Can implement with an **`E`** array
- In each case, can code open addressing and chaining
- The coding of each method is analogous to what we saw with **`HashMap`**

Implementing the Java Map and Set Interfaces

- The Java API uses a hash table to implement both the **`Map`** and **`Set`** interfaces
- Implementing them is aided by abstract classes **`AbstractMap`** and **`AbstractSet`** in the **`Collection`** hierarchy
- Interface **`Map`** requires nested type **`Map.Entry<K,V>`**
- Interface **`Map`** also requires support for viewing it as a **`Set`** of **`Entry`** objects

Applying Maps: Phone Directory

```
public String addOrChangeEntry (
    String name, String newNum) {
  String oldNum = dir.put(name, newNum);
  modified = true;
  return oldNum;
}
public String lookupEntry (String name) {
  return dir.get(name);
}
public String removeEntry (String name) {
  String ret = dir.remove(name);
  if (ret != null) modified = true;
  return ret; }
```

Applying Maps: Phone Directory (2)

```
// in loadData:
while ((name = ins.readLine()) != null) {
  if ((number = ins.readLine()) == null)
    break;
  dir.put(name, number);
}

// saving
for (Map.Entry<String,String> curr :
     dir.entrySet()) {
  outs.println(curr.getKey());
  outs.println(curr.getValue());
}
```

Applying Maps: Huffman Coding

```
// First, want to build frequency table
//   for a given input file
public static HuffData[] buildFreqTable (
    BufferedRead ins) {
  Map<Character, Integer> freqs =
    new HashMap<Character, Integer>();
  try {
    ... process each character ...
  } catch (IOException ex) {
    ex.printStackTrace();
  }
  ... build array from map ...
```

Applying Maps: Huffman Coding (2)

```
// process each character
int next;
while ((next = ins.read()) != -1) {
  Integer count = freqs.get((char) next);
  if (count == null)
    count = 1;
  else
    ++count;
  freqs.put((char)next, count);
}
ins.close();
```

Applying Maps: Huffman Coding (3)

```
// build array from map
HuffData[] freqTab =
  new HuffData[freqs.size()];
int i = 0;
for (Map.Entry<Character,Integer> entry :
     freqs.entrySet()) {
  freqTab[i++] =
    new HuffData(
      entry.getValue().doubleValue(),
      entry.getKey());
}
return freqTab;
```

Applying Maps: Huffman Coding (4)

```
// build ENCODING table
public void buildCodeTab () {
  codeMap =
    new HashMap<Character,BitString>();
  buildCodeTab(huffTree, new BitString());
}
```

Applying Maps: Huffman Coding (5)

```
public void buildCodeTab (
    BinaryTree<HuffData> tree,
    BitString code) {
  HuffData datum = tree.getData();
  if (datum.symbol != null)
    codeMap.put(datum.symbol, code);
  else {
    BitString l = (BitString)code.clone();
    l.append(false);
    buildCodeTab(tree.left() , l);
    BitString r = (BitString)code.clone();
    r.append(true);
    buildCodeTab(tree.right(), r); } }
```

Applying Maps: Huffman Coding (6)

```
public void encode (BufferedReader ins,
    ObjectObjectStream outs) {
  BitString res = new BitString();
  try {
    int next;
    while ((next = ins.read()) != -1) {
      Character nxt = (char)next;
      BitString nextChunk =
        codeMap.get(nxt);
      res.append(nextChunk);
    }
    res.trimCapacity();   ins.close();
    outs.writeObject(res);outs.close();...
```