Sets and Maps

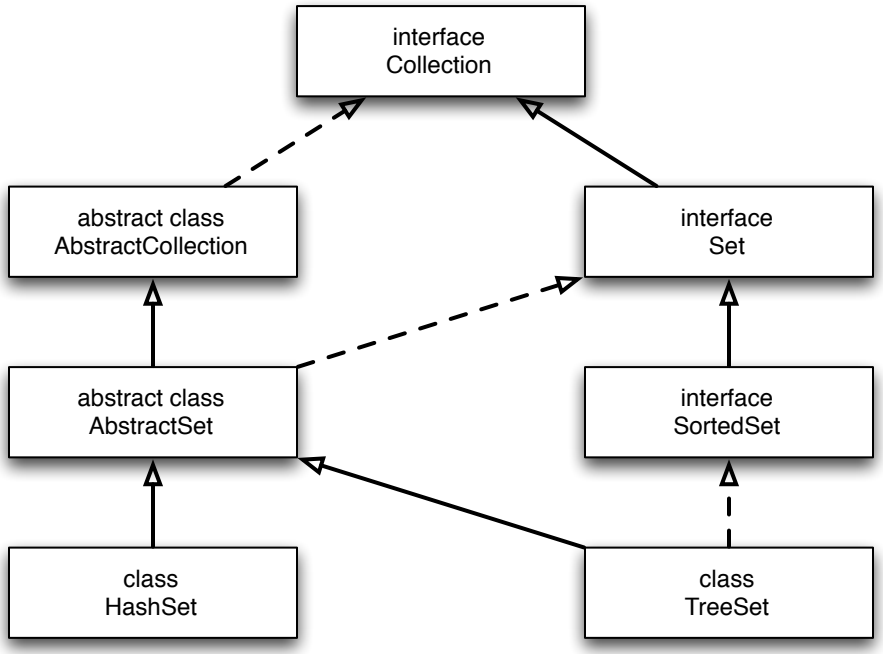
Chapter 7

Chapter Objectives

- To understand and learn the use of
  - The Java Map interface
  - The Java Set interfaces
- To learn about hash coding and its use to facilitate efficient search and retrieval
- To study and implement two forms of hash table:
  - Open addressing
  - Chaining
- To learn implementation of the Map interface

Sets and the Set Interface

- The set family belongs to the Collection hierarchy. They consist of three interfaces, two abstract classes, and two actual classes



The Set Abstraction

- A set is a collection of elements without duplication (one with duplication is called a multiset).
- Set operations include:
  - Operations with an element:
    - Testing for membership
    - Adding an element
    - Removing an element
  - Operations with a set:
    - Union
    - Intersection
    - Difference
    - Subset

The Set Interface and Methods

- The Set interface has methods required for:
  - Testing set membership
  - Testing for an empty set
  - Determining set size
  - Creating an iterator over the set
- Two optional methods (defined to throw “Unsupported Operation” exception unless overridden) for:
  - Adding an element and removing an element
- Constructors enforce the no-duplicate-members criterion
  - Add method does not allow duplicate items to be inserted

Some Methods in java.util.Set<E>

Methods	Action
boolean add(E obj)	Adds obj to the set. Returns **true** if obj was not in the set and **false** o.w.
boolean addAll(Collection<E> coll)	Adds all the elements in coll. Returns **true** if the set is modified and **false** o.w.
boolean contains(E obj)	Returns **true** if the set contains obj and **false** o.w.
boolean containsAll(Collection<E> coll)	Returns **true** if the set contains all the elements of coll and **false** o.w.
boolean isEmpty()	Returns **true** is the set has no elements and **false** o.w.
Iterator<E> iterator()	Returns an iterator over the set.
boolean remove(E obj)	Attempts to remove obj. Returns **true** if obj was in the set and thus was removed and **false** o.w.
boolean removeAll(Collection<E> coll)	Attempts to remove all objects in coll. Returns **true** if the set is modified and **false** o.w.
boolean retainAll(Collection<E> coll)	Retain only the elements in coll. Returns **true** if the set is modified and **false** o.w.
int size()	Returns the number of elements in the set.

Maps and the Map Interface

- A map is a set of ordered pairs (key,value).
- In a map, the keys are unique but values may not be unique.

Some of the java.util.Map<K,V> Methods

Methods	Action
V get(Object key)	Returns the value associated with *key*. Returns null if there is no pair with *key* in the map.
boolean isEmpty()	Returns **true** if the map has no pairs and **false** o.w.
V put(K key, V value)	Assigns *value* to *key* in the map. If the key had a value associated with it, returns the value; o.w., return null.
V remove(Object key)	Removes the mapping of *key* in the map. If there was a value associated with key returns the value, otherwise returns null.
int size()	Returns the number of elements in the map.

Hash Tables

- The goal behind the hash table is to provide an instant access to many elements through their keys, without having to search for their keys in an array or in a tree.
  - With proper arrangement, constant-time access can be provided to every element on average.
- A hash table is implemented using an array, where an element is assigned to a slot in an array by a map that assigns an index value to each key.

Hash Code Generation

- In most applications, a key is a series of digits or a series of characters.
- The number of possible key values is much larger than the table size, but the number of elements to be stored in the table is expected to be significantly smaller than the table size.
- Generating good hash codes is somewhat of an experimental process.
- The code generation process must be relatively simple.

A Simplistic Hash Code

- Treat each element of the series as a number (in the case of a string), and sum the numbers.
  - Pro: easy to calculate
  - Con: order insensitive
    - "sing" and "sign" are mapped to the same value
- The Java String.hashCode() returns the integer calculated by the formula:
    - $s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1}$

  where s_i is the *i*th character of the string, and *n* is the length of the string
    - "Cat" will have a hash code of: 'C' x 31^2 + 'a' x 31 + 't'
    - 31 is a prime number that generates relatively few collisions

Open Addressing and Chaining

- Consider two ways to organize hash tables when collisions occur
  - Open addressing
  - Chaining
- Linear probing can be used to access an item in a hash table
  - If that element contains an item with a different key, increment the index by one
  - Keep incrementing until the key is found or a null entry is encounter

Searching for an Item in Open Indexing

1. Compute the index by taking the item's hashCode() % table.length.
2. **if** table[index] is null
3. The item is not in the table.
4. **else if** table[index] is equal to the item
5. The item is in the table.
6. **else** (A case of collision)
7. Repeat the following:
8. Update the index by adding one and then taking
9. the modulo (i.e., %) by table.length
10. Stop if either the item is found or a null is found.

Avoid an Infinite Loop

- If the item is not in the table and the table is full, possibly enter an infinite loop.
- Solve the problem by:
  - Stopping when the index becomes the start index
  - Ensuring that the table is never full by enlarging the table whenever the occupancy rate exceeds a certain value, for example 50%.

Issues

- No meaningful way to enumerate the elements in the table.
- When open indexing is used, can't remove element because the slot may need to be recognized as occupied to find another element.
  - A dummy value must be stored instead and so deleted items waste storage space and reduce search efficiency
  - Recovery can be made when table size is changed.
- Use a prime number for the size of the table so as to reduce collisions
- A fuller table will result in increased collisions

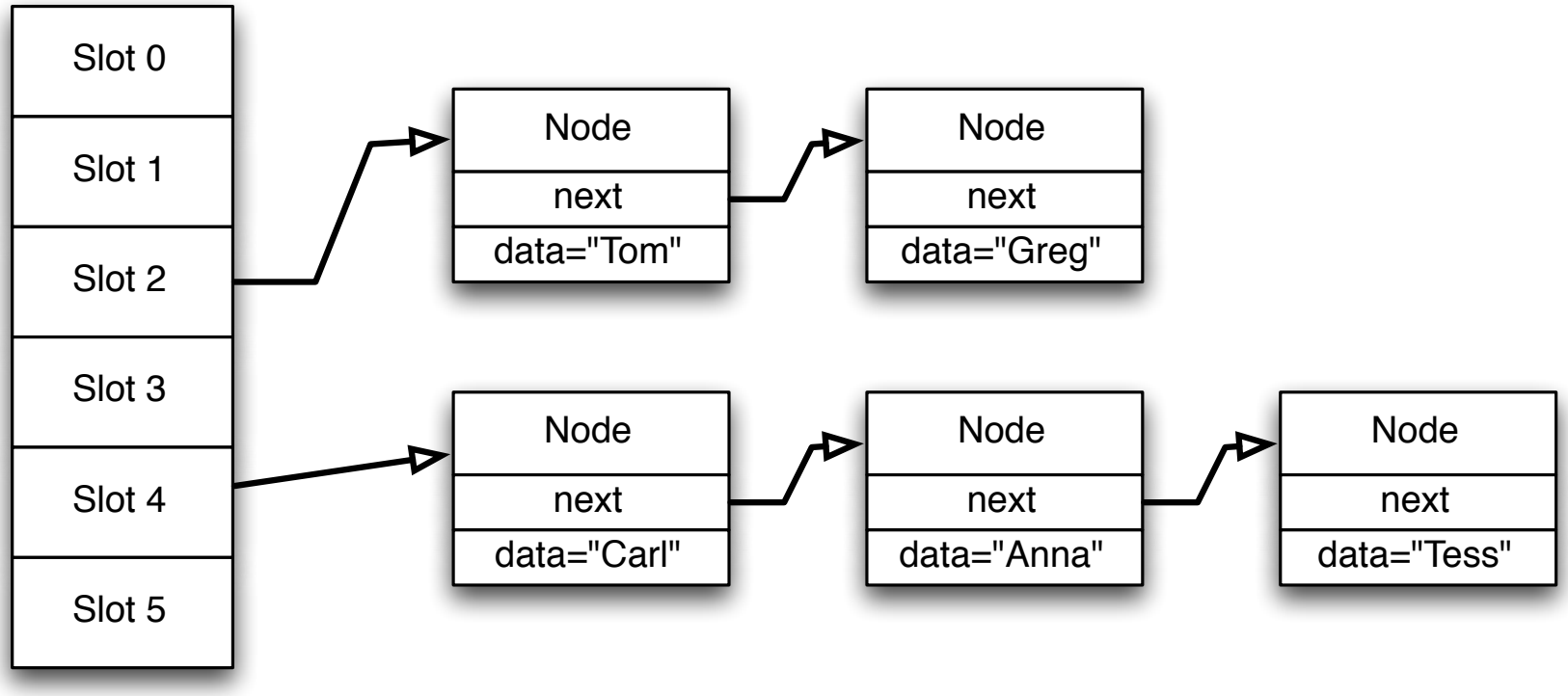
Quadratic Probing

- Linear probing tends to form clusters of keys in the table, causing longer search chains. This is because consecutive indices are examined for an open slot.
- Quadratic probing can reduce the effect of clustering
  - Increments form a quadratic series. The *i*-th probing examines the slot:
    - $h(k,i) = (h_0(k) + c_0 i + c_1 i^2)$ % table_size
  - The next index calculation is time consuming as it involves some additions and a modulo division
  - Not all table elements are examined when looking for an insertion index

Chaining

- Chaining is an alternative to open addressing
- Each table element references a linked list that contains all of the items that hash to the same table index
  - The linked list is often called a bucket
  - The approach sometimes called bucket hashing
- Only items that have the same value for their hash codes will be examined when looking for an object

Chaining (continued)



Performance of Hash Tables

- Load factor is the number of filled cells divided by the table size
- Load factor has the greatest effect on hash table performance
- The lower the load factor, the better the performance as there is a lesser chance of collision when a table is sparsely populated

Mini-Implementation of a Hash Table KWHashMap<K,V>

Methods	Action
V get(Object key)	Returns the value associated with *key*. Returns null if there is no pair associated with *key* in the map.
boolean isEmpty()	Returns **true** is the map has no pairs and **false** o.w.
V put(K key, V value)	Assigns *value* to *key* in the map. Returns the value the key had an associated value and null o.w.
V remove(Object key)	Removes the mapping of key in the map. If there was a value associated with key returns the value, otherwise returns null.
int size()	Returns the number of elements in the set.

Internal Class Entry<K, V>

Data Field	Attributes
private E key	The key
private V value	The value
Constructor	**Action**
public Entry(E key, V Value)	Construct an entry with the key and the value
Methods	**Action**
public K getKey()	Returns the key of the entry.
public V getValue()	Returns the value of the entry.
public setValue(V val)	Sets the value of the entry.

Implementation of Hash Tables
HashTableOpen

Data Field	Attributes
private Entry<E,V>[] table	The hash table realized as an array of Entry type objects
private static final int START_CAPACITY	The initial capacity (the initial table size)
private double LOAD_THRESHOLD	The maximum load capacity
private int numKeys	The number of keys currently stored in the table, excluding the keys that were deleted
private final Entry<E, V> DELETED	An Entry type object used in place of a deleted entry.
Private Methods	**Action**
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.

Implementation of Hash Tables
HashTableChain

Data Field	Attributes
private LinkedList<Entry <E,V>>[] table	The hash table realized as an array of LinkedList type objects that link Entry type objects
private static final int CAPACITY	The initial capacity (the initial table size)
private static final int LOAD_THRESHOLD	The maximum load capacity
private int numKeys	The number of keys currently stored in the table, excluding the keys that were deleted

Implementation Considerations for Maps and Sets

- Class Object implements methods *hashCode()* and *equals(Object o)*, so every class can access these methods unless it overrides them
  - *Object.equals(Object o)* compares *this* and *o* based on their addresses, not their contents
  - Object.hashCode calculates the hash code of *this* based on its address, not based on its contents
- Java recommends that overriding of the equals method should be accompanied with overriding of the hashCode method

Implementing HashSetOpen

Map Methods	Set Methods
Object get(Object key)	boolean contains(Object key)
Object put(Object key, Object value)	boolean add(Object key)
Object remove(Object key)	boolean remove(Object key)

Implementing the Java Map and Set Interfaces

- The Java API uses a hash table to implement both the Map and Set interfaces
- The task of implementing the two interfaces is simplified by the inclusion of abstract classes AbstractMap and AbstractSet in the Collection hierarchy

Nested Interface Map.Entry

- One requirement on the key-value pairs for a Map object is that they implement the interface Map.Entry<K, V>, which is an inner interface of interface Map
  - An implementer of the Map interface must contain an inner class that provides code for the methods in the table below

Methods	Action
public K getKey()	Returns the key of the entry.
public V getValue()	Returns the value of the entry.
public setValue(V val)	Sets the value of the entry.

Other Applications of Maps

- Phone Directory can be viewed as a Map of a name to a number
- Huffman Coding Problem
  - Use a map for creating an array of elements and replacing each input character by its bit string code in the output file
  - Frequency table
    - The key will be the input character
    - The value is the character code string