CSC 143 Java

Maps & Hashing

Map

- Recall the Map interface (subset):

```
public interface Map<K, V> {
    // Return the value associated with the given key
    public V get(Object key);

    // Associate the given key with the given value
    public V put(K key, V value);

    // Remove the key, and its associated value from the table
    public V remove(Object key);
}
```


Using a Map

- Adding/Retrieving associations:

```
HashMap<String, Book> books = new HashMap<String, Book> ( );
Book book1 = new Book("The Java Programming Language",
                      "Arnold, Gosling, Holmes", "A-W", 2000);
Book book2 = new Book ("Java in a Nutshell", "Flanagan", "O'Reilly",
1999);
books.put("0-201-70433-1", book1);
books.put("0-201-65432-1"  book2);
Book text = books.get("0-201-70433-1");
```

What about:

```
Book otherther = books.get("0-201-11111-1");
```


Map Entries

- A map is a collection of key-value pairs

```
class MapEntry<K, V> {          // One entry in the map
    K key;                 //   Entry key
    V value;               //   Entry value
     MapEntry(K key, V value) { ... }
}
```

(java.util.Map.Entry is the actual interface used in the Java collection classes)

- Main restriction: The set of key objects in a map may not contain duplicates. (No such restriction on values)

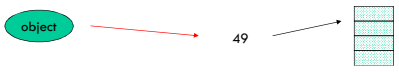
Implementation Choices

- Options?
  - array, linked list, List, binary tree?
  - Thoughts?
- Efficiencies?

How can we get the efficiency of direct access with an array, without having to search?

Hashing

- What if we had a method that could *convert each possible element value into its own unique integer?*
  - Takes an element, returns an integer
  - a ***perfect hash function***
- Then we could store the map entries in an array, with each entry stored at an index equal to the key's hash code



- Array access is constant time – very fast: O(1)
- If computing the hash value is also O(1), lookup is O(1)

Ex: hashing a String. Any ideas? What about the capacity of the array?

Hash Codes in Your Own Classes

- Class Object defines a method hashCode( ) which returns an integer code for an object
- Subclasses should override hashCode( ) if a more suitable hash function is appropriate for instances
- Key rule: if o1 and o2 are different objects, then if
  `o1.equals(o2) == true`
  it must also be true that
  `o1.hashCode( ) == o2.hashCode( )`
- Corollary: If you override either hashCode( ) or equals(…) in a class, you probably should override the other one to be consistent. Base each method on the same instance variables.
- Danger: The Java system cannot enforce these rules. A well-designed ("proper") class will follow them as a matter of good practice

Collisions

What happens if two keys map to the same index?

Several possible solutions

- **Open addressing**: probe into the array for an open index
  - ✓ Linear
  - ✓ Quadratic
  - ✓ Rehash
- **Separate chaining**: each index refers to a list of all key/value pairs that hash to that index (we'll look at this one)

Solution: Buckets

- Instead of each array position containing the map entry...
  - it can contain a *list* of elements that all share the same hash code
  - This list is called a *bucket*
  - Unlike ordinary buckets, this kind can never be full!
- To test whether an element is in the map:
  - Use the hash code to find the correct bucket
  - Search that bucket's list for the element
- Add works similarly

More about Buckets

- If hash function is good, then most elements will be in different buckets, and each bucket will be short
  - Most of the time, contains( ) and add( ) will be fast!
- There will probably be unused buckets – particularly at first
  - No data value happens to hash to a particular bucket
- Tradeoff:
  - more buckets: shorter linked lists, more unused space
  - fewer buckets: longer linked lists, less unused space

HashMap Representation

```
class HashMap<K, V> implements Map<K, V> {
  private static final nBuckets =101; // # of buckets

  private List<MapEntry>[ ] bucket;
                    // bucket[k] is a list of <key, value> pairs
                    //  key.hashCode( ) % nBuckets == k
                    // bucket[k]==null if no keys map to k
  public HashMap( ) {
     bucket = new List <MapEntry>[ nBuckets ];
  }
```

Other class methods?

Analysis

- Parameters
  - *n* number of items stored in the HashMap
  - *b* number of buckets
- Load factor: *n*/*b* – ratio of # entries to # buckets
- Cost of key lookup ~ O(1) provided that
  - Hash function is good – distributes keys evenly throughout buckets
    Ensures that buckets are all about the same size; no really long buckets
  - Load factor is small
    Don't have to search too far in any one bucket