CSE 373: Data Structures and Algorithms

Lecture 16: Hashing III
Hashing practice problem

• Draw a diagram of the state of a hash table of size 10, initially empty, after adding the following elements.
  – \( h(x) = x \mod 10 \) as the hash function.
  – Assume that the hash table uses linear probing.

  7, 84, 31, 57, 44, 19, 27, 14, and 64
Removal with probing hash tables

- **lazy removal**: instead of actually removing elements, replace them with a special REMOVED value
  - avoids expensive re-shuffling of elements on remove
  - example: remove 18, search for 57
  - lookup algorithm becomes slightly modified
    - what should we do when we hit a slot containing the REMOVED value?
      - keep going
      - add algorithm becomes slightly modified
        - what should we do when we hit a slot containing the REMOVED value?
          - use that slot, replace REMOVED with the new value
          - add(17) --> slot 8
Analysis of linear probing

- The load factor $\lambda$ is the fraction of the table that is full:
  - Empty table $\lambda = 0$
  - Half full table $\lambda = 0.5$
  - Full table $\lambda = 1$

- Assuming a reasonably large table, the average number of buckets examined per insertion (taking clustering into account) is roughly $\frac{1 + 1/(1-\lambda)^2}{2}$:
  - Empty table: $\frac{1 + 1/(1-0)^2}{2} = 1$
  - Half full: $\frac{1 + 1/(1-.5)^2}{2} = 2.5$
  - 3/4 full: $\frac{1 + 1/(1-.75)^2}{2} = 8.5$
  - 9/10 full: $\frac{1 + 1/(1-.9)^2}{2} = 50.5$

- As long as the hash function is fair and the table is not too full, then inserting, deleting, and searching are all $O(1)$ operations.
Rehashing, hash table size

- **rehash**: increasing the size of a hash table's array, and re-storing all of the items into the array using the hash function
  - can we just copy the old contents to the larger array?

  - When should we rehash? Some options:
    - when load reaches a certain level (e.g., $\lambda = 0.5$)
    - when an insertion fails

- What is the cost (Big-Oh) of rehashing?
- what is a good hash table array size?
  - how much bigger should a hash table get when it grows?
Hashing practice problem

• Draw a diagram of the state of a hash table of size 10, initially empty, after adding the following elements.
  – \( h(x) = x \mod 10 \) as the hash function.
  – Assume that the hash table uses linear probing.
  – Assume that rehashing occurs at the start of an add where the load factor is 0.5.

  7, 84, 31, 57, 44, 19, 27, 14, and 64

• Repeat the problem above using quadratic probing.
How does Java's HashSet work?

- It stores *Objects*; every object has a reasonably-unique *hashCode*
  - public int hashCode() in class Object

- HashSet stores elements in array by hashCode() value
  - searching for this element later, we just have to check that one index to see if it's there (O(1))
    - "Tom Katz".hashCode() % 10 == 6
    - "Sarah Jones".hashCode() % 10 == 8
    - "Tony Balogne".hashCode() % 10 == 9
**Membership testing in HashSets**

- When searching a `HashSet` for a given object (`contains`):
  - the set computes the `hashCode` for the given object
  - it looks in that index of the `HashSet`'s internal array
    - Java compares the given object with the object in the `HashSet`'s array using `equals`; if they are equal, returns true

- Hence, an object will be considered to be in the set only if both:
  - It has the same `hashCode` as an element in the set, and
  - The `equals` comparison returns true

- General contract: if `equals` is overridden, `hashCode` should be overridden also; equal objects must have equal hash codes
Common Error: overriding `equals` but not `hashCode`

```java
public class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x;  this.y = y;
    }
    public boolean equals(Object o) {
        if (o == this) { return true; }
        if (!(o instanceof Point)) { return false; }
        Point p = (Point)o;
        return p.x == this.x && p.y == this.y;
    }
    // No hashCode!
}
```

- The follow code would surprisingly print `false`!

```java
HashSet<Point> p = new HashSet<Point>();
p.add(new Point(7, 11));
System.out.println(p.contains(new Point(7, 11)));
```
Overriding `hashCode`

- **Conditions for overriding `hashCode`:**
  - return same value for object whose state hasn’t changed since last call
  - `if x.equals(y), then x.hashCode() == y.hashCode()`
  - `(if !x.equals(y), it is not necessary that x.hashCode() != y.hashCode() ... why?)`

- **Advantages of overriding `hashCode`**
  - your objects will store themselves correctly in a hash table
  - distributing the hash codes will keep the hash balanced: no one bucket will contain too much data compared to others

```java
public int hashCode() {
    int result = 37 * x;
    result = result + y;
    return result;
}
```
Things to do in a good `hashCode` implementation

- make sure the hash code is same for equal objects
- try to ensure that the hash code will be different for different objects
- ensure that the hash code depends on every piece of state that is important to the object (every piece of state that is used in `equals`)
- preferably, weight the pieces so that different objects won’t happen to add up to the same hash code

```java
public class Employee {
    ...
    public int hashCode() {
        int result = 17;
        result = 37 * result + this.name.hashCode();
        result = 37 * result + new Double(this.salary).hashCode();
        return 37 * result + this.employeeID;
    }
}
```
The Map ADT

- **map**: Holds a set of unique keys and a collection of values, where each key is associated with one value
  - a.k.a. "dictionary", "associative array", "hash"

- **basic map operations**:
  - **put(key, value)**: Adds a mapping from a key to a value.
  - **get(key)**: Retrieves the value mapped to the key.
  - **remove(key)**: Removes the given key and its mapped value.

```
myMap.get("Juliet") returns "Capulet"
```
Maps in computer science

• Compilers
  – symbol table

• Operating Systems
  – page tables
  – file systems (file name → location)

• Real world Examples
  – names to phone numbers
  – student ID to student information
Using Maps

• in Java, maps are represented by Map interface in java.util

• Map is implemented by the HashMap and TreeMap classes
  – HashMap: implemented with hash table; uses separate chaining extremely fast: $O(1)$; keys are stored in unpredictable order
  – TreeMap: implemented with balanced binary search tree; very fast: $O(\log N)$; keys are stored in sorted order

  – A map requires 2 type parameters: one for keys, one for values.

// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>put(key, value)</strong></td>
<td>adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one</td>
</tr>
<tr>
<td><strong>get(key)</strong></td>
<td>returns the value mapped to the given key (null if not found)</td>
</tr>
<tr>
<td><strong>containsKey(key)</strong></td>
<td>returns true if the map contains a mapping for the given key</td>
</tr>
<tr>
<td><strong>remove(key)</strong></td>
<td>removes any existing mapping for the given key</td>
</tr>
<tr>
<td><strong>clear()</strong></td>
<td>removes all key/value pairs from the map</td>
</tr>
<tr>
<td><strong>size()</strong></td>
<td>returns the number of key/value pairs in the map</td>
</tr>
<tr>
<td><strong>isEmpty()</strong></td>
<td>returns true if the map's size is 0</td>
</tr>
<tr>
<td><strong>toString()</strong></td>
<td>returns a string such as &quot;{a=90, d=60, c=70}&quot;</td>
</tr>
<tr>
<td><strong>keySet()</strong></td>
<td>returns a set of all keys in the map</td>
</tr>
<tr>
<td><strong>values()</strong></td>
<td>returns a collection of all values in the map</td>
</tr>
<tr>
<td><strong>putAll(map)</strong></td>
<td>adds all key/value pairs from the given map to this map</td>
</tr>
<tr>
<td><strong>equals(map)</strong></td>
<td>returns true if given map has the same mappings as this one</td>
</tr>
</tbody>
</table>
keySet and values

• **keySet()** returns a `Set` of all keys in the map
  – can loop over the keys in a foreach loop
  – can get each key's associated value by calling `get` on the map

```java
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Meghan", 29);
ages.put("Kona", 3);  // ages.keySet() returns Set<String>
ages.put("Daisy", 1);
for (String name : ages.keySet()) {
    int age = ages.get(name);  // Daisy -> 1
    System.out.println(name + " -> " + age);  // Kona -> 3
    // Meghan -> 29
}
```

• **values()** returns a collection of values in the map
  – can loop over the values in a foreach loop
  – no easy way to get from a value to its associated key(s)
Implementing Map with Hash Table

• Each map entry adds a new key → value pair to the map
  – entry contains:
    • key element of given type
    • value element of given value type
    • additional information needed to maintain hash table

• Organized for super quick access to keys
  – the keys are what we will be hashing on
Implementing Map with Hash Table, cont.

```java
public interface Map<K, V> {
    public boolean containsKey(K key);
    
    public V get(K key);
    
    public void print();
    
    public void put(K key, V value);
    
    public V remove(K key);
    
    public int size();
}
```
public class HashMapEntry<K, V> {
    public K key;
    public V value;
    public HashMapEntry<K, V> next;

    public HashMapEntry(K key, V value) {
        this(key, value, null);
    }

    public HashMapEntry(K key, V value, HashMapEntry<K, V> next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }
}
Map implementation: put

• Similar to our Set implementation's add method
  – figure out where key would be in the map
  – if it is already there replace the existing value with the new value
  – if the key is not in the map, insert the key, value pair into the map as a new map entry
Map implementation: put

```java
public void put(K key, V value) {
    int keyBucket = hash(key);

    HashMapEntry<K, V> temp = table[keyBucket];
    while (temp != null) {
        if ((temp.key == null && key == null)
            || (temp.key != null && temp.key.equals(key))) {
            temp.value = value;
            return;
        }
        temp = temp.next;
    }

    table[keyBucket] =
        new HashMapEntry<K, V>(key, value, table[keyBucket]);
    size++;
}
```