CSE 373
Data Structures and Algorithms

Lecture 18: Hashing III

Runtime of hashing

- the load factor λ is the fraction of the table that is full
  - λ = 0 (empty) λ = 0.5 (half full) λ = 1 (full table)
- Linear probing:
  - If hash function is fair and λ < 0.5 - 0.6, then hashtable operations are all O(1)
- Double hashing:
  - If hash function is fair and λ < 0.9 - 0.95, then hashtable operations are all O(1)

Rehashing

- **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?
  - when table is half full
  - when an insertion fails
  - when load reaches a certain level (best option)

Rehashing (cont'd)

- What is the cost (Big-Oh) of rehashing?
  - O(n). Isn't that bad?

- How much bigger should a hash table get when it grows?
  - What is a good hash table array size?
    - Find next prime that is at least twice the current table's size

Hashing practice problem

- Draw a diagram of the state of a hash table of size 10, initially empty, after adding the following elements.
  - $h(x) = x \bmod 10$ 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.

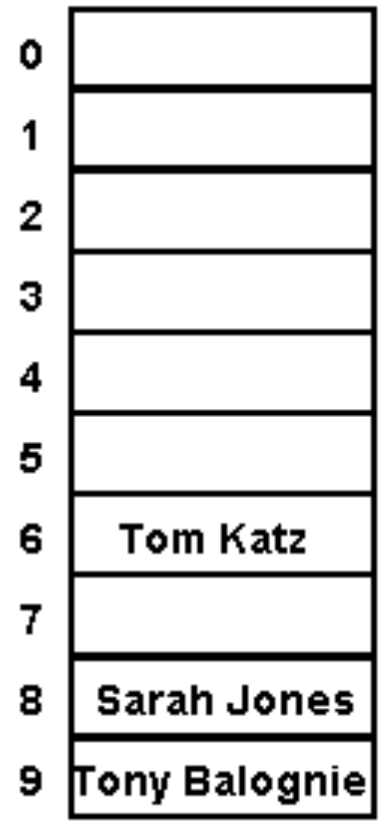
How do we hash different objects in Java?

- Every object that will be hashed should define a reasonably unique *hash code*
  - `public int hashCode()` in class `Object`

- Hash tables will index elements in array by `hashCode()` value
  - If using separate chaining, 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 Balognie".hashCode() % 10 == 9
```

* Assuming chains are not too long



Error: not overriding equals

```
public class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    // No equals!
}
```

- The following code prints `false`!

```
ArrayList<Point> p = new ArrayList<Point>();
p.add(new Point(7, 11));
System.out.println(p.contains(new Point(7, 11)));
```

Membership testing in `ArrayList` in Java

- When searching for a given object (`contains`):
  - Java compares the given object with objects in the `ArrayList` using the object's `equals` method

- Override the `Employee`'s `equals` method.

Error: overriding equals but not hashCode

```
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 following code prints `false`!

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

Membership testing in `HashSet` in Java

- When searching for a given object (`contains`):
  - The set computes the `hashCode` for the given object
  - It looks in the chain at that index of the `HashSet`'s internal array
  - Java compares the given object with objects in the `HashSet` using the object's `equals` method

- **General contract:** if `equals` is overridden, `hashCode` should be overridden also; equal objects must have equal hash codes

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 not?

- 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

```
public int hashCode() {
    int result = 37 * x;
    result = result + y;
    return result;
}
```

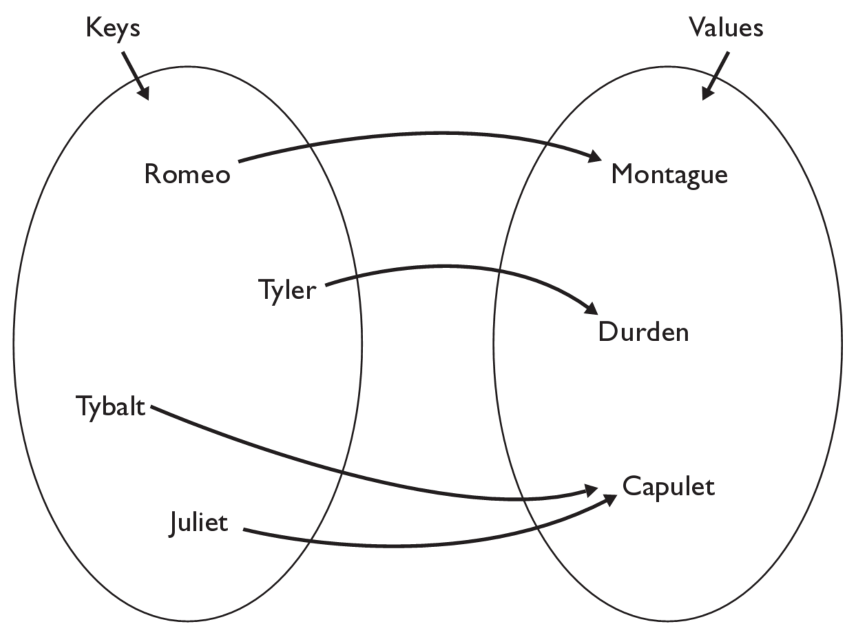
Overriding hashCode, cont'd.

- 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
  - Try to ensure that the hash code depends on every piece of state that is used in `equals`
    - What if you don't?
      - Strings prior to Java 1.2 only considered the first 16 letters. What is wrong with this?
  - Preferably, weight the pieces so that different objects won't happen to add up to the same hash code

- Override the `Employee`'s `hashCode` method.

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
  - File systems (file name → location)
- Real world Examples
  - Names to phone numbers
  - URLs to IP addresses
  - Student ID to student information

Using Maps

- In Java, maps are represented by the `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>();
```

Map methods

`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**key**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`
`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

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

```
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()) {          // Daisy -> 1
    int age = ages.get(name);                // Kona -> 3
    System.out.println(name + " -> " + age); // 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 key type (`null` is a valid key value)
    - 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.

```
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();
}
```

HashMapEntry

```
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

```
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++;
}
```