L6: Hashing, HashMap, HashSet
Alex Steiger
CompSci 201, Spring 2024
1/31/24

Announcements, Coming up

• Today, Wednesday, 1/31
  • APT 2 due

• Monday, 2/5
  • Project 1: NBody due (future projects will be 2 week)
  • Project 2: Markov out (due in 2 weeks)

• Next Wednesday, 2/8
  • APT 3 due
Map pairs keys with values

- Like an **address book**, lookup the value (address) of a key (person). Like a dictionary in Python.

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>101 E. Main St.</td>
</tr>
<tr>
<td>Naomi</td>
<td>200 Broadway</td>
</tr>
<tr>
<td>Stavros</td>
<td>121 Durham Ave.</td>
</tr>
</tbody>
</table>

- Map is an interface, must have methods like:
  - `put(k, v)`: Associate value v with key k
  - `get(k)`: Return the value associated with key k
  - `containsKey(k)`: Return true if key k is in the Map

Implementations of Map

Two major implementations:

- **HashMap**: Very efficient `put`, `get`, `containsKey`
- **TreeMap**: Nearly as efficient, keeps keys sorted by their "natural ordering"

```java
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

Map<String, String> addressbook = new TreeMap<>();
addressbook.put("Bob", "101 E. Main St.");
addressbook.put("Naomi", "200 Broadway");
addressbook.put("Stavros", "121 Durham Ave.");
System.out.println(addressbook);
```

Check before you get

If you call `.get(key)` on a key not in the map, returns null, can cause program to crash.

```java
Map<String, Integer> myMap = new HashMap<>();
int val = myMap.get("hi");
```

Instead, check first with `.containsKey()`.

```java
Map<String, Integer> myMap = new HashMap<>();
if (myMap.containsKey("hi")) {
    int val = myMap.get("hi");
}
```
Adding “default” values

Often want a “default” value associated with new keys (examples: 0, empty list, etc.). Two options:
• \texttt{putIfAbsent(key, val)}
• Check if does not contain key before put

```
Map<String, Integer> myMap = new HashMap<>();
myMap.putIfAbsent("hi", 0);

// Equivalent to line 6 &
if (myMap.containsKey("hi")) {
    myMap.put("hi", 0);
}
```

Updating maps

<table>
<thead>
<tr>
<th>Immutable values:</th>
<th>Mutable values (e.g. collections)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{get()} returns a copy of the value.</td>
<td>\texttt{get()} returns reference to collection.</td>
</tr>
<tr>
<td>Must use \texttt{put()} again to update.</td>
<td>Update the collection directly.</td>
</tr>
</tbody>
</table>

```
Map<String, Integer> myMap = new HashMap<>();
myMap.put("hi", 0);
int currentVal = myMap.get("hi");
myMap.put("hi", currentVal + 1);
```

Counting with a Map

In this example we count how many of each character occur in message.

```
String message = "computer science is so much fun";
TreeMap<Character, Integer> charCounts = new TreeMap<>();
for (Char c : message.toCharArray()) {
    if (charCounts.containsKey(c)) {
        charCounts.put(c, charCounts.get(c) + 1);
    } else {
        int currentVal = charCounts.get(c);
        charCounts.put(c, currentVal + 1);
    }
}
```
HashSet/HashMap Implementation

HashSet/Map efficiency

public class HashSet<
    extends AbstractSet<
　Implements Iterable, Cloneable, Serializable>

This class implements the Set interface using a hash table (actually a HashMap instance). It makes no guarantees as to the insertion order of the set, in particular, it does not guarantee that the order will remain constant over time. This class stores the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size). Assuming the hash function disperses the elements properly among the buckets, inverting over this set requires O(n) time, proportional to the sum of the hashSet's iterators (size() of elements in the set). Thus, it's very important not to set the initial capacity too low, or else resizing will occur frequently.

Under assumptions...

Aside: Does constant time lookup (contains(), get(), etc.) matter?

• Social media: When you login, server needs to lookup to display the correct page for you.
  • Billions of accounts! Look it up in a List? NO! Constant time lookup with hashing.

• Routing/directions application: Need to lookup roads from a given intersection.
  • How many possible roads? Search through a list? NO! Constant time lookup with hashing.

• Could go on!
Big questions about hashing
Last class: Usage of API HashSet/HashMap.

Today:
1. How does a hash table work to implement HashMap/HashSet?
2. Why do .equals() and .hashCode() matter?
3. Why are the add(), contains(), put(), get(), and containsKey(), etc., all constant time (and under what assumptions)?

Hash Table Concept
- Implement HashMap with an Array also, storing <key, value> pairs
  - HashSet: A HashMap with only keys (no vals)
- Instead of always adding to next open spot (0, 1, 2, 3...)
- Big idea: Calculate hash (an int) of key to determine where to store & lookup
  - Java OOP: Will use the hashCode() method of the key to get the hash
- Same hash to put and get, no looping over list

HashMap methods at a high level
Always start by getting the hash = Math.abs(key.hashCode()) % list.size()
- put(key, value)
  - Add <key, value> to list at index hash
  - If key already there, update value
- get(key)
  - Return value paired with key at index hash position of list
- containsKey(key)
  - Check if key exists at index hash position of list
HashMap put/get example

- Suppose we have the <key, value> pair <“cs”, 201>.

Hash:

```java
jshell> Math.abs("cs").hashCode() % 8
8 == 0
```

- put("cs", 201) in position 0
- get("cs") by looking up position 0, returning the value

Collisions

- Suppose now we want to put <“fain”, 104>.

Hash:

```java
jshell> Math.abs("fain").hashCode() % 8
7 == 0
```

- put("fain", 104) in position 0
- But "cs", 201 is already stored at position 0! Call this a collision.

Dealing with collisions: concepts

- Think of the hash table as an Array of "buckets".
- Each bucket can store multiple <key, value> pairs.

- **put(key, value)**
  - Add to hash index bucket
  - Update value if key already in bucket

- **get(key)**
  - Loop over keys in hash index bucket
  - Return value of one that equals(key)
Dealing with collisions: details

- Bucket is really another list
- Hash table is really an array of lists of <key, value> pairs.
- We call this technique for dealing with collisions chaining.
Where does equals() come in?

• If multiple <key, value> pairs in same bucket, need to know which to get() or update on a put() call.

• Always the pair where the key in the bucket equals() the key we put() or get().

• Need equals() to work correctly for the key type
  • String keys? Integer? Already implemented for you
  • Storing objects of a class you write? Need to override and implement equals().

What happens without equals()?

Hashing cats

```java
public class Cat {
    String name;
    int age;

    // Override
    public int hashCode() {
        return 8;
    }
}
```

Even though all cat objects have the same hashCode() of 0 and so go to the same bucket:

```java
public static void main(String[] args) {
    Set<Cat> myCats = new HashSet<>();
    myCats.add(new Cat("kirk", 22));
    myCats.add(new Cat("kirk", 22));
    System.out.println(myCats.size());
}
```

And these 2 Cat objects have the same values:

```java```
```java```

Points 2: cannot detect duplicates without equals()
hashCode Correctness

- Need `hashCode()` to work correctly for the key type.
  - String keys? Already implemented for you.
  - Storing objects of classes you write? Need to override and implement `hashCode()`.

- What makes a `hashCode()` "correct" (not necessarily efficient)?
  - Any two objects that are `equals()` should have the same `hashCode()`.

What happens without `hashCode()`? Hashing more cats

```java
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        return other.getName().equals(this.name) && other.getAge() == this.age;
    }

    public void print() {
        System.out.println(name + " - " + age);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Cat myCat1 = new Cat("Meow", 1);
        Cat myCat2 = new Cat("Meow", 1);
        System.out.println(myCat1.equals(myCat2));
    }
}
```

Still prints 2

Cat with `equals()` and `hashCode()`

```java
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        return other.getName().equals(this.name) && other.getAge() == this.age;
    }

    @Override
    public int hashCode() {
        return name + Integer.tostring(age).hashCode();
    }
}
```
Aside: toString()

Don't need for hashing, but toString() method allows "nice" printing.

```java
public class Cat {
  String name;

  @Override
  public String toString() {
    return name;
  }
}
```

toString() method used for printing, including inside a Collection

Prints Kirk instead of Cat...

What is the String hashCode()?  

Remember how hashCode() is used to get the bucket index.

Interprets each character as an int, does arithmetic.

Revisiting Hashing Efficiency

- Runtime of get(), put(), and containsKey():
  - Time to get the hash
  - Time to search "bucket", calling .equals() on everything in the bucket

⇒ HashMaps are faster with more buckets
“correct” but inefficient `hashCode`

Correctness requirement: Any `.equals()` keys should have the same `hashCode()`.

```
@Override
public int hashCode() {
    return 0;
}
```

Still satisfies, but not good...
Stores everything in the first bucket!
No more efficient than a list!

Correct and efficient `hashCode()`

From the Java 17 API documentation:

- **Correctness**: "If two objects are equal...`hashCode()` must produce the same integer result."
- **Efficiency**: "...producing distinct integer results for unequal objects may improve the performance of hash tables."

```
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (o instanceof Cat) {
            Cat other = (Cat) o;
            if (other.name == this.name && other.age == this.age) {
                return true;
            }
            return false;
        }
    }

    @Override
    public int hashCode() {
        return (name + Integer.toString(age)).hashCode();
    }
}
```

Cat `hashCode()` revisited

```
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (o instanceof Cat) {
            Cat other = (Cat) o;
            if (other.name == this.name && other.age == this.age) {
                return true;
            }
            return false;
        }
    }

    @Override
    public int hashCode() {
        return (name + Integer.toString(age)).hashCode();
    }
}
```
Simple uniform hashing assumption (SUHA)

- Suppose we hash N pairs to M buckets.

**Simple uniform hashing assumption:**

- Any element (i.e., key for HashMap, value for HashSet) is equally likely to hash into any bucket, independently of where any other element hashes to. ([CLRS])

- Probability any two unequal elements hash into the same bucket: 1/M
  - Spread of pairs to buckets looks random (but is not).
  - Ways to design such hash functions, not today
  - We make this assumption to analyze efficiency in theory, can verify runtime performance in practice

Implications of SUHA

- Expected number of pairs per bucket under SUHA? N/M [N pairs, M buckets].

- Stronger statements are true: Very high probability that any bucket has approximately N/M pairs.

- Runtime implication?
  - Time to get the hash
  - Time to search over the hash index "bucket"
  - Calling .equals() on everything in the bucket

  Constant, does not depend on N or M

  Expect ~ N/M pairs to search