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
Finishing Maps API
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<KEY_TYPE, VALUE_TYPE> addressBook = new TreeMap<>();
addressBook.put("Bob", "101 E. Main St.");
addressBook.put("Naomi", "200 Broadway");
addressBook.put("Xi", "121 Durham Ave.");
System.out.println(addressBook);

{Bob=101 E. Main St., Naomi=200 Broadway, Xi=121 Durham Ave.}
```
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");
```

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.lang.Integer.intValue()" because the return value of "java.util.Map.get(Object)" is null

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:

• .putIfAbsent(key, val)
• Check if does not contain key before put

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

// Equivalent to line 8
if (!myMap.containsKey("hi")) {
    myMap.put("hi", 0);
}
```
Updating maps

Immutable values:
• `.get()` returns a copy of the value.
• Must use `.put()` again to update.

Mutable values (e.g. collections)
• `.get()` returns reference to collection.
• Update the collection directly.

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

```
14   Map<String, List<Integer>> otherMap = new HashMap<>();
15   otherMap.put("hi", new ArrayList<>());
16   otherMap.get("hi").add(0);
```
In this example we count how many of each character occur in message.

```java
String message = "computer science is so much fun";
char[] messageCharArray = message.toCharArray();
TreeMap<Character, Integer> charCounts = new TreeMap<>();
for (char c : messageCharArray) {
    if (!charCounts.containsKey(c)) {
        charCounts.put(c, 1);
    } else {
        int currentValue = charCounts.get(c);
        charCounts.put(c, currentValue + 1);
    }
}
System.out.println(charCounts);
{ =5, c=4, e=3, f=1, h=1, i=2, m=2, n=2, o=2, p=1, r=1, s=3, t=1, u=3}
```
HashSet/HashMap Implementation
HashSet/Map efficiency

```java
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, Serializable;
```

This class implements the Set interface backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits 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.** Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Java API documentation

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>&lt;“hi”, 5&gt;</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;“ok”, 3&gt;</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Suppose we have the <key, value> pair <“cs”, 201>.

**Hash:**

```
[jshell> Math.abs("cs".hashCode()) % 8
$7 ==>> 0
```

- put(“cs”, 201) in position 0
- get(“cs”) by looking up position 0, returning the value returns 201
Collisions

- Suppose now we want to put `<"fain", 104>`.

  **Hash:**

  ```java
  jshell> Math.abs("fain".hashCode()) % 8
  $11 ==> 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;“cs”, 201&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;“fain”, 104&gt;</td>
</tr>
<tr>
<td>1</td>
<td>&lt;“hi”, 5&gt;</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;“ok”, 3&gt;</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
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**.

Illustration credit: By Jorge Stolfi - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6471915
2. HashSet and HashMap have constant time add, contains, put, get, and containsKey operations. That means that these methods... *

- Take the same amount of time to run
- Have the same number of operations

☑️ Runtimes do not depend on number of elements of the Set/Map
3. What is stored in each "bucket" in a HashMap? *

- A <key, value> pair
- A list of keys
- A list of values
- A list of <key, value> pairs

* Correct answer: A list of <key, value> pairs
4. Suppose we want to put \(<s, 1>\) into a HashMap where \(s\.hashCode() = 12\). If hash table has 4 buckets, in which bucket will we store \(<s, 1>\)?

Select your answer **Answer: 0**
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
  - 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 0;
    }
}
```

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

And these 2 Cat objects have the same values

```
public static void main(String[] args) {
    Set<Cat> myCats = new HashSet<>();
    myCats.add(new Cat("kirk", 2));
    myCats.add(new Cat("kirk", 2));
    System.out.println(myCats.size());
}
```

Prints 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

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

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

Fixed `equals()` but removed `hashCode()`, using default

```
public static void main(String[] args) {
    Set<Cat> myCats = new HashSet<>();
    myCats.add(new Cat("kirk", 2));
    myCats.add(new Cat("kirk", 2));
    System.out.println(myCats.size());
}
```

Still prints 2!
Cat with `equals()` and `hashCode()`

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

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

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

- `equals()` if have same name and age
- `hashCode()` of name concat with age, if `equals()` will have same `hashCode()`
Aside: `toString()`

Don’t need for hashing, but `toString()` method allows “nice” printing.

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

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

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

```java
public static void main(String[] args) {
    Set<Cat> myCats = new HashSet<>();
    myCats.add(new Cat("kirk", 2));
    System.out.println(myCats);
}
```

Prints `kirk` instead of `Cat@...`
What is the String `hashCode()`?

```java
private int getBucket(String s) {
    int val = Math.abs(s.hashCode()) % myTable.size();
    return val;
}
```

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

Interprets each character as an int, does arithmetic.

Java API String documentation

<table>
<thead>
<tr>
<th>String</th>
<th>HashCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;hello&quot;</td>
<td>99162322</td>
</tr>
<tr>
<td>&quot;hellop&quot;</td>
<td>99162323</td>
</tr>
<tr>
<td>&quot;what&quot;</td>
<td>3648196</td>
</tr>
</tbody>
</table>
Revisiting Hashing Efficiency

• Runtime of `get()`, `put()`, and `containsKey()`
  \[\text{= Time to get the hash} \]
  \[\text{+ Time to search “bucket”, calling \textit{.equals()} on everything in the bucket} \]

⇒ HashMaps are faster with more buckets
“correct” but inefficient \texttt{hashCode()}

Correctness requirement: Any \texttt{.equals()} keys should have the same \texttt{hashCode()}. 

\begin{verbatim}
@Override
public int hashCode() {
    return 0;
}
\end{verbatim}

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](https://docs.oracle.com/en/java/javase/17/docs/api/jdk/):

- **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.”

- **String `hashCode()`** satisfies both

  ```java
  jshell> "hello".hashCode();
  $4 ==> 99162322
  ```

  ```java
  jshell> "hello".hashCode();
  $5 ==> 99162323
  ```

  ```java
  jshell> "what".hashCode();
  $6 ==> 3648196
  ```
Cat 

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

    @Override
    public boolean equals(Object o) {
        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

Expect ~ N/M pairs to search

Constant, does not depend on N or M.