Building Java Programs

Generics, hashing

reading: 18.1
Hashing

- **hash**: To map a value to an integer index.
  - **hash table**: An array that stores elements via hashing.

- **hash function**: An algorithm that maps values to indexes.
  - One possible hash function for integers: \( HF(I) = I \mod \text{length} \)

```java
set.add(11);  // 11 % 10 == 1
set.add(49);  // 49 % 10 == 9
set.add(24);  // 24 % 10 == 4
set.add(7);   // 7 % 10 == 7
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>
public static int hashFunction(int i) {
    return Math.abs(i) % elementData.length;
}

- Add: set elementData[HF(i)] = i;
- Search: check if elementData[HF(i)] == i
- Remove: set elementData[HF(i)] = 0;

- What is the runtime of add, contains, and remove?
  - $O(1)$

- Are there any problems with this approach?
Hash Functions

- Maps an object to a number
  - result should be constrained to some range
  - passing in the same object should always give the same result

- Results from a hash function should be distributed over a range
  - very bad if everything hashes to 1!
  - should "look random"

- How would we write a hash function for String objects?
Hashing objects

- It is easy to hash an integer I (use index $I \mod length$).
  - How can we hash other types of values (such as objects)?

- All Java objects contain the following method:

  ```java
  public int hashCode()
  ```

  Returns an integer hash code for this object.

  - We can call `hashCode` on any object to find its preferred index.

- **How is `hashCode` implemented?**
  - Depends on the type of object and its state.
    - Example: a String's `hashCode` adds the ASCII values of its letters.
  - You can write your own `hashCode` methods in classes you write.
    - All classes come with a default version based on memory address.
Hash function for objects

```java
public static int hashFunction(E e) {
    return Math.abs(e.hashCode()) % elements.length;
}
```

- **Add:** set `elements[HF(o)] = o;`
- **Search:** check if `elements[HF(o)].equals(o)`
- **Remove:** set `elements[HF(o)] = null;`
String's hashCode

- The `hashCode` function inside `String` objects looks like this:

```java
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

- As with any general hashing function, collisions are possible.
  - Example: "Ea" and "FB" have the same hash value.
  - Early versions of Java examined only the first 16 characters. For some common data this led to poor hash table performance.
Implementing generics

// a parameterized (generic) class
class name<TypeParameter> {
    ...
}

- Forces any client that constructs your object to supply a type.
  - Don't write an actual type such as String; the client does that.
  - Instead, write a type variable name such as \( E \) (for "element") or \( T \) (for "type").

- You can require multiple type parameters separated by commas.

- The rest of your class's code can refer to that type by name.
Collisions

- **collision**: When hash function maps 2 values to same index.

```java
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(7);
set.add(54);  // collides with 24!
```

- **collision resolution**: An algorithm for fixing collisions.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>54</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>
Probing

- **probing**: Resolving a collision by moving to another index.
  - **linear probing**: Moves to the next index.

```java
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);
set.add(54); // collides with 24; must probe
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>54</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>

- Is this a good approach?
  - variation: **quadratic probing** moves increasingly far away
Clustering

- **clustering**: Clumps of elements at neighboring indexes.
  - slows down the hash table lookup; you must loop through them.

```java
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);
set.add(14);
set.add(86);
```

```
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>54</td>
<td>14</td>
<td>7</td>
<td>86</td>
<td>49</td>
</tr>
</tbody>
</table>
```

- How many indexes must a lookup for 94 visit?
Chaining

- **chaining**: Resolving collisions by storing a list at each index
  - add/search/remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes, unlike with probing
public class HashIntSet {
    private static final int CAPACITY = 137;
    private List<Integer>[] elements;

    // constructs new empty set
    public HashSet() {
        elements = (List<Integer>[])(new List[CAPACITY]);
    }

    // adds the given value to this hash set
    public void add(int value) {
        int index = hashFunction(value);
        if (elements[index] == null) {
            elements[index] = new LinkedList<Integer>();
        }
        elements[index].add(value);
    }

    // hashing function to convert objects to indexes
    private int hashFunction(int value) {
        return Math.abs(value) % elements.length;
    }
    ...
}
...  
// Returns true if this set contains the given value.
public boolean contains(int value) {
    int index = hashFunction(value);
    return elements[index] != null &&
           elements[index].contains(value);
}

// Removes the given value from the set, if it exists.
public void remove(int value) {
    int index = hashFunction(value);
    if (elements[index] != null) {
        elements[index].remove(value);
    }
}
Rehashing

- **rehash**: Growing to a larger array when the table is too full.
  - Cannot simply copy the old array to a new one. (Why not?)

- **load factor**: ratio of (# of elements) / (hash table length)
  - many collections rehash when load factor ≈ .75
  - can use big prime numbers as hash table sizes to reduce collisions
... // Grows hash array to twice its original size.
private void rehash() {
    List<Integer>[] oldElements = elements;
    elements = (List<Integer>[])
            new List[2 * elements.length];
    for (List<Integer> list : oldElements) {
        if (list != null) {
            for (int element : list) {
                add(element);
            }
        }
    }
}
Other questions

- How would we implement `toString` on a `HashSet`?
Implementing a hash map

- A hash map is just a set where the lists store key/value pairs:

```java
//       key    value
map.put("Marty", 14);
map.put("Jeff",  21);
map.put("Kasey", 20);
map.put("Stef",  35);
```

- Instead of a `List<Integer>`, write an inner `Entry` node class with `key` and `value` fields; the map stores a `List<Entry>`