1	2	3	∑

Algorithmen, Datenstrukturen und Datenabstraktion, WiSe 17/18

Tutor_in: William Surau, Tutorium 4

Übung 5

Merlin Joseph & Anton Oehler

8. Dezember 2017

Aufgabe 1: Hashing: Implementierung

a)
- Wird `hashcode()` mehrfach auf einem Objekt aufgerufen (ohne dass sich das Objekt gegenüber `equals()` ändert) so bleibt sein hashcode gleich.
- Sind zwei Objekte nach `equals()` gleich, so müssen auch ihre hashcodes gleich sein.
- Eigene Implementierungen von `equals()` (die u.U. effizienter sind, den Daten entsprechen, ...) können es notwendig machen, `hashcode()` auch zu überschreiben, um die in `hashcode()` beschriebenen Regeln einzuhalten.

Listing 1: Hashtable

b)

```java
import java.util.*;
import java.util.Random;

public class Hashtable<K, V> {
 public static void main(String[] args) {
  Hashtable<Byte, String> h = new Hashtable<>();
  run(h);
 }
 public static void run(Hashtable<Byte, String> h) {
  System.err.println("Usage:");
  System.err.println("\tp     - print hashtable");
  System.err.println("\ta [k] - add new entry [with key k]");
  System.err.println("\tc  k  - check whether key k is in the hashtable");
  System.err.println("\tr  k  - delete key k from hashtable");
  System.err.println("\th  k  - print hash for input k");
  System.err.println("\texit  - exit program");
  System.err.println();

  Random r = new Random();
  byte k;
  byte c = 0;
  for (String in = "";; in = System.console().readLine().trim()) {
   String[] input = in.split(" ");
   switch (input[0]) {
    case "print":
    case "p":
     System.out.println("size/capacity: " + h.size() + "/" + h.capacity());
     System.out.println(h);
     break;
    case "add":
    case "a":
     if (input.length > 1) {
      k = Byte.parseByte(input[1]);
     } else {
      k = c;
     c += 1;
    }
    String v = "" + r.nextInt();
    System.out.println("adding key + random value: " + k + "; " + v);
    h.put(k, v);
    break;
   case "contains":
   case "c":
    if (input.length <= 1) {
     System.err.println("expected 2 args");
     break;
    }
    k = Byte.parseByte(input[1]);
    System.out.println("Key " + k + " in HT? " + h.contains(k));
    break;
   case "remove":
   case "r":
    if (input.length <= 1) {
     System.err.println("expected 2 args");
     break;
    }
    k = Byte.parseByte(input[1]);
    System.out.println("deleted " + k + " : " + h.remove(k));
    break;
   case "capacity":
    System.out.println("capacity: " + h.capacity());
    break;
   case "size":
    System.out.println("size: " + h.size());
    break;
   case "hash":
   case "h":
    if (input.length <= 1) {
     System.err.println("expected 2 args");
     break;
    }
    k = Byte.parseByte(input[1]);
    System.out.println("hash(" + k + ") = " + h.hash(k));
    break;
   case "exit":
    return;
  }
 }
}

protected class Entry {
 public K key;
 public V val;
 public Entry(K key, V val) {
  this.key = key;
  this.val = val;
 }
 public K getKey() { return key; }
 public V getVal() { return val; }
}

protected Object[] table; // ArrayList<Entry>[]
protected int size = 0;

public Hashtable() {
 this(10);
}
public Hashtable(int size) {
 table = new Object[size];
}

protected int genIndex(K key) {
 // hashCode() of an integer maps just to its value,
 // which results in a poor distribution of hashes,
 // so key is casted to a String, concatenated with "_"
 // and with itself again, to gain somewhat of a better
 // distribution
 int hash = (key + "_" + key).hashCode();
 // modulo the hashcode to fit it into the array
 int hc = hash % capacity();
 if (hc < 0) {
  return -hc;
 } else {
  return hc;
 }
}
protected String hash(K key) {
 return ""+((key + "_" + key).hashCode());
}
public void put(K key, V val) {
 _put(key, val);
 check_rehash();
}
protected void _put(K key, V val) {
 int hash = genIndex(key);
 if (table[hash] == null) {
  table[hash] = new ArrayList<Entry>();
 }
 @SuppressWarnings("unchecked")
 ArrayList<Entry> cell = (ArrayList<Entry>) table[hash];
 cell.add(new Entry(key,val));
 table[hash] = cell;
 size += 1;
}
public boolean contains(K key) {
 int hash = genIndex(key);
 if (table[hash] == null) return false;

 @SuppressWarnings("unchecked")
 ArrayList<Entry> cell = (ArrayList<Entry>) table[hash];
 for (Entry e : cell) {
  if (e.getKey() == key) {
   return true;
  }
 }
 return false;
}
public V remove(K key) {
 int hash = genIndex(key);
 if (table[hash] == null) return null;

 @SuppressWarnings("unchecked")
 ArrayList<Entry> cell = (ArrayList<Entry>) table[hash];
 for (int z = 0; z < cell.size(); z++) {
  if (cell.get(z).getKey() == key) {
   V tmp = cell.remove(z).getVal();
   size -= 1;
   check_rehash();
   return tmp;
  }
 }
 return null;
}

protected void check_rehash() {
 int old_size = capacity();
 int new_size;

 if (size() > capacity()) {
  new_size = 2 * old_size;
 } else if (2 * size() < capacity() && size() > 10) {
  new_size = old_size / 2;
 } else {
  // no need to rehash
  return;
 }
 System.out.println("rehash: " + old_size + " -> " + new_size);

 Object[] old_table = table;

 table = new Object[new_size];
 size = 0;

 for (int i = 0; i < old_size; i++) {
  if (old_table[i] == null) {
   continue;
  }

  @SuppressWarnings("unchecked")
  ArrayList<Entry> cell = (ArrayList<Entry>) old_table[i];
  for (Entry e : cell) {
   this._put(e.getKey(), e.getVal());
  }
 }
}
public String toString() {
 StringBuilder sb = new StringBuilder();
 for (int i = 0; i < capacity(); i++) {
  sb.append(i).append(": ");
  if (table[i] == null) {
   sb.append("null\n");
   continue;
  }

  @SuppressWarnings("unchecked")
  ArrayList<Entry> cell = (ArrayList<Entry>) table[i];
  if (cell.isEmpty()) {
   sb.append("null\n");
   continue;
  }
  sb.append("[");
  for (Entry e : cell) {
   sb.append("(").append(e.getKey()).append(",").append(e.getVal()).append("),");
  }
  sb.deleteCharAt(sb.length() - 1).append("]\n");
 }
 return sb.toString();
}

protected int capacity() {
 return table.length;
}
public int size() {
 return size;
}
}
```

Aufgabe 4: Kryptographische Hashfunktionen

a) Kryptographische Hashfunktionen sind Hashfunktionen, die möglichst kollisionsresistens sind und sich nicht invertieren lassen (Es ist (quasi) unmöglich, aus einem Hash die ursprüngliche Eingabe oder eine Eingabe, die zum gleichen Hash führt zu berechnen)
Beispiele: MD5 (unsicher), SHA-256, Whirlpool

b) Das package `java.security` enthält die Klasse `MessageDigest`, welche mindestens die Hash-Funktionen MD5, SHA-1 und SHA-256 bereitstellen muss (jedoch je nach Implementierung noch mehr bereitstellen kann)

Listing 2: Hashtable mit kryptografischer Hashfunktion

c)

```java
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Formatter;

public class HashtableSHA<K extends Byte, V> extends Hashtable<K, V> {
 public static void main(String[] args) {
  HashtableSHA<Byte, String> h = new HashtableSHA<>();
  run(h);
 }

 private MessageDigest md;

 public HashtableSHA() {
  this(10);
 }
 public HashtableSHA(int capacity) {
  super(capacity);
  try {
   md = MessageDigest.getInstance("SHA-256");
  } catch (NoSuchAlgorithmException e) {
   System.err.println(e);
  }
 }

 @Override
 protected int genIndex(K key) {
  md.reset();
  md.update(key);
  byte[] hash = md.digest();
  int hc = hash[0] % capacity();
  if (hc < 0) {
   return -hc;
  } else {
   return hc;
  }
 }
 @Override
 protected String hash(K key) {
  md.reset();
  md.update(key);
  byte[] hash = md.digest();

  Formatter form = new Formatter();
     for (int i = 0; i < hash.length; i++) {
      form.format("%02x", hash[i]);
     }
     return form.toString();
 }
}
```