TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Christoph Anneser (anneser@in.tum.de)
http://db.in.tum.de/teaching/ss21/ei2/

Blatt Nr. 8
Dieses Blatt wird am Montag, den 14. Juni 2021 besprochen.

Aufgabe 1: Hashing mit Linear Probing

Veranschaulichen Sie Hashing mit Linear Probing.

Die Größe der Hashtabelle ist dabei jeweils $m = 13$. Führen Sie die folgenden Operationen aus:

`insert`	$16, 3, 12, 17, 29, 10, 24$
`delete`	16
`insert`	$5, 1, 15$
`delete`	10
`insert`	14
`delete`	1

Verwenden Sie die Hashfunktion

$$h(x) = 3x \bmod 13.$$

Bei dieser Aufgabe sind die Schlüssel der Elemente die Elemente selbst.

Beim **Löschen** soll die Wiederherstellung der folgenden Invariante erfolgen: *Für jedes Element e in der Hashtabelle mit Schlüssel $k(e)$, aktueller Position j und optimaler Position $i = h(k(e))$ sind alle Positionen $i, (i+1) \bmod m, (i+2) \bmod m, \ldots, j$ der Hashtabelle belegt.* Überlegen Sie sich eine effiziente Strategie, die diese Invariante widerherstellt. Bei dieser Aufgabe soll keine dynamische Größenanpassung der Hashtabelle stattfinden.

1. Operation: `insert`(16) mit opt. Position: 9

0	1	2	3	4	5	6	7	8	9	10	11	12
									16			

2. Operation: `insert`(3) mit opt. Position: 9

0	1	2	3	4	5	6	7	8	9	10	11	12
									16	3		

3. Operation: `insert`(12) mit opt. Position: 10

0	1	2	3	4	5	6	7	8	9	10	11	12
									16	3	12	

4. Operation: `insert`(17) mit opt. Position: 12

0	1	2	3	4	5	6	7	8	9	10	11	12
									16	3	12	17

5. Operation: `insert`(29) mit opt. Position: 9

0	1	2	3	4	5	6	7	8	9	10	11	12
29									16	3	12	17

6. Operation: `insert`(10) mit opt. Position: 4

0	1	2	3	4	5	6	7	8	9	10	11	12
29				10					16	3	12	17

7. Operation: `insert`(24) mit opt. Position: 7

0	1	2	3	4	5	6	7	8	9	10	11	12
29				10			24		16	3	12	17

8. Operation: `delete`(16) mit opt. Position: 9

0	1	2	3	4	5	6	7	8	9	10	11	12
				10			24		3	12	29	17

9. Operation: `insert`(5) mit opt. Position: 2

0	1	2	3	4	5	6	7	8	9	10	11	12
		5		10			24		3	12	29	17

10. Operation: `insert`(1) mit opt. Position: 3

0	1	2	3	4	5	6	7	8	9	10	11	12
		5	1	10			24		3	12	29	17

11. Operation: `insert`(15) mit opt. Position: 6

0	1	2	3	4	5	6	7	8	9	10	11	12
		5	1	10		15	24		3	12	29	17

12. Operation: `delete`(10) mit opt. Position: 4

0	1	2	3	4	5	6	7	8	9	10	11	12
		5	1			15	24		3	12	29	17

13. Operation: `insert`(14) mit opt. Position: 3

0	1	2	3	4	5	6	7	8	9	10	11	12
		5	1	14		15	24		3	12	29	17

14. Operation: `delete`(1) mit opt. Position: 3

0	1	2	3	4	5	6	7	8	9	10	11	12
		5	14			15	24		3	12	29	17

Aufgabe 2: Hashing mit Chaining

Veranschaulichen Sie Hashing mit Chaining. Die Größe m der Hash-Tabelle ist in den folgenden Beispielen jeweils die Primzahl 11. Die folgenden Operationen sollen nacheinander ausgeführt werden.

`insert`	$3, 11, 9, 7, 14, 56, 4, 12, 15, 8, 1$
`delete`	56
`insert`	25

Der Einfachheit halber sollen die Schlüssel der Elemente die Elemente selbst sein.

Verwenden Sie zunächst die Hashfunktion

$$g(x) = 5x \mod m.$$

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

1. Operation: `insert`(3):

0	1	2	3	4	5	6	7	8	9	10
				3						

2. Operation: `insert`(11):

0	1	2	3	4	5	6	7	8	9	10
11				3						

3. Operation: `insert`(9):

0	1	2	3	4	5	6	7	8	9	10
11	9			3						

4. Operation: `insert`(7):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3						

5. Operation: `insert`(14):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3						
				14						

6. Operation: `insert`(56):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56					
				14						

7. Operation: `insert`(4):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56				4	
				14						

8. Operation: `insert`(12):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56				4	
				14	12					

9. Operation: `insert`(15):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56				4	
				14	12				15	

10. Operation: `insert`(8):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56		8		4	
				14	12				15	

11. Operation: `insert`(1):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56		8		4	
				14	12				15	
					1					

12. Operation: `delete`(56):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	12		8		4	
				14	1				15	

13. Operation: `insert`(25):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	12		8		4	
				14	1				15	
				25						

Aufgabe 3: Hashing mit Linear Chaining - Implementierung

Implementieren Sie in Java eine Hashtabelle die "linear chaining" zur Kollisionsbehandlung verwendet: Jede Position in der Hashtabelle speichert eine Liste von Elementen mit diesem Hashwert. Bonus: Implementieren Sie einen Mechanismus um die Hashtabelle wachsen zu lassen.

```java
import java.util.ArrayList;
import java.util.List;

public class MyHashTable<T> {
    private static final float maxLoadFactor = 0.75f;
    private int numBuckets; // initial number of buckets; usually a power of 2
    private int size = 0; // number of elements currently saved in hashtable
    private ArrayList<ArrayList<T>> buckets;

    MyHashTable() {
        this(2);
    }

    MyHashTable(int numBuckets) {
        this.numBuckets = numBuckets;
        this.buckets = new ArrayList<>(numBuckets); // (optional) specify
        ↪ initial capacity for no reallocations
        for (int i = 0; i < numBuckets; i++) { // fill array with empty lists
            this.buckets.add(new ArrayList<>());
        }
    }

    public void add(T value) {
        // Add to appropriate bucket
        int hash = calculateBucketOffset(value);
        List<T> bucket = buckets.get(hash);
        for (T key : bucket) {
            if (key == value) { // Avoid duplicates
                return;
            }
        }
        bucket.add(value);
```

```
32          size++;
33
34          // Rehash ?
35          if ((1.0f * size) / numBuckets >= maxLoadFactor) {
36              rehash(numBuckets * 2);
37          }
38      }
39
40      public void remove(T value) {
41          int hash = calculateBucketOffset(value);
42          List<T> bucket = buckets.get(hash);
43
44          for (int i = 0; i < bucket.size(); i++) {
45              if (bucket.get(i) == value) {
46                  size--;
47                  bucket.remove(i);
48                  return;
49              }
50          }
51      }
52
53      private int calculateBucketOffset(T value) {
54          return Math.abs(value.hashCode() % numBuckets); // save hash, absolute
            ↪ value for negative hashCode
55      }
56
57      private void rehash(int newNumBuckets) {
58          MyHashTable<T> largerHashTable = new MyHashTable<>(newNumBuckets);
59          for (List<T> bucket : buckets) {
60              for (T value : bucket) {
61                  largerHashTable.add(value);
62              }
63          }
64          numBuckets = newNumBuckets;
65          buckets = largerHashTable.buckets;
66      }
67  }
```