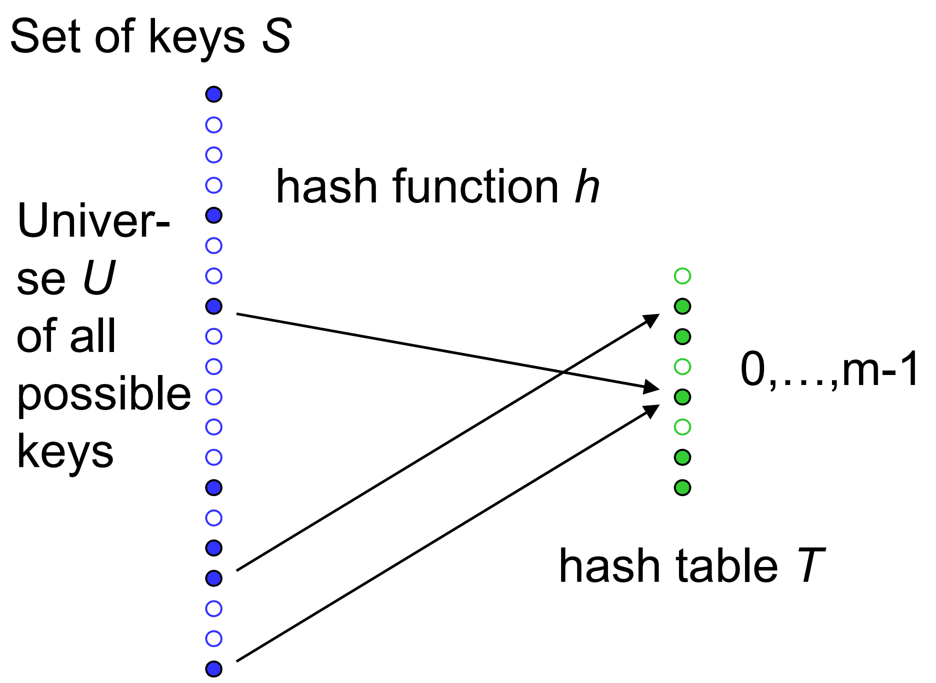
8 Hashing: Open addressing

Summer Term 2010

Robert Elsässer

Albert-Ludwigs-Universität Freiburg

Hashing: General Framework



$$\left(H(u) \subseteq [-2^{31}, 2^{31}]\right)$$

h(*s*) = hash address

h(*s*) = *h*(*s*´) $\Leftrightarrow$ *s* and *s*´ are synonyms with respect to *h*

address collision

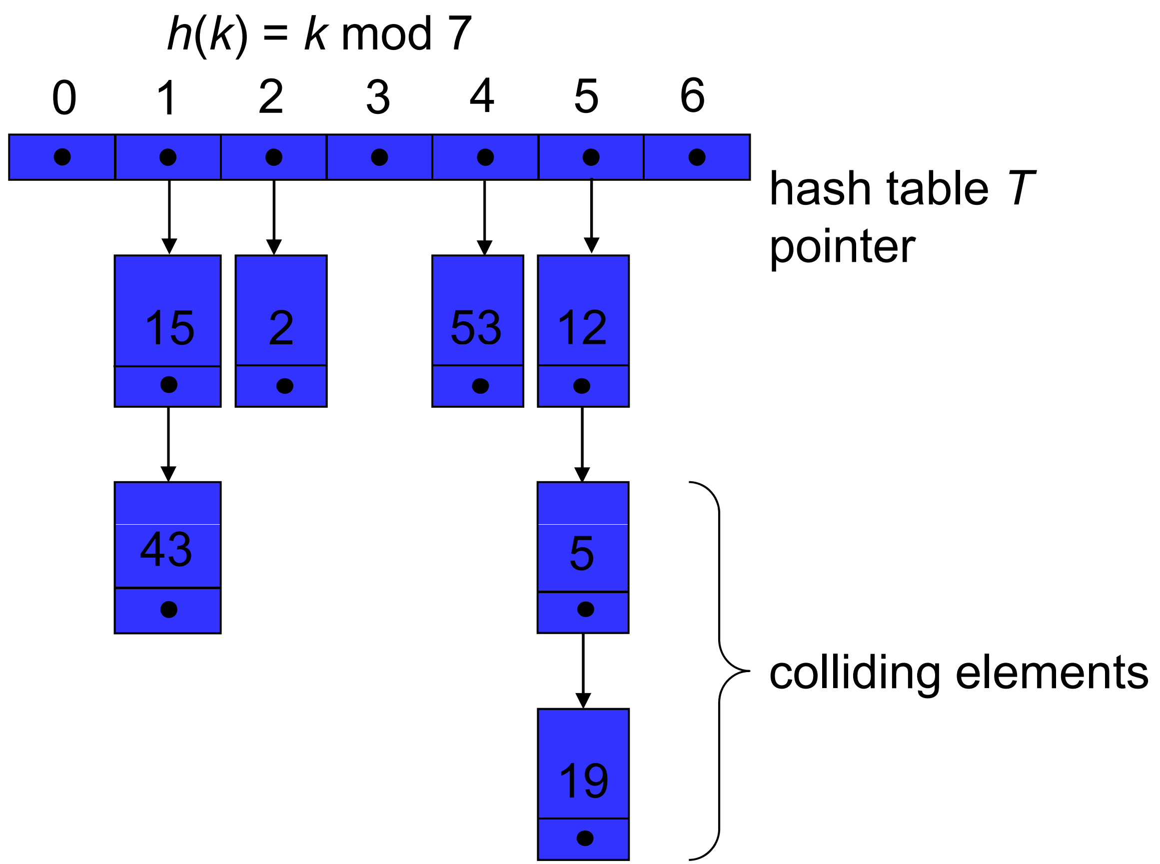
Possible ways of treating collisions

Treatment of collisions:

- Collisions are treated differently in different methods.
- A data set with key s is called a colliding element if bucket $B_{h(s)}$ is already taken by another data set.
- What can we do with colliding elements?
  1. Chaining: Implement the buckets as linked lists. Colliding elements are stored in these lists.
  2. Open Addressing: Colliding elements are stored in other vacant buckets. During storage and lookup, these are found through so-called probing.

Hashing by chaining

Keys are stored in overflow lists



This type of chaining is also known as direct chaining.

Open addressing

Idea:

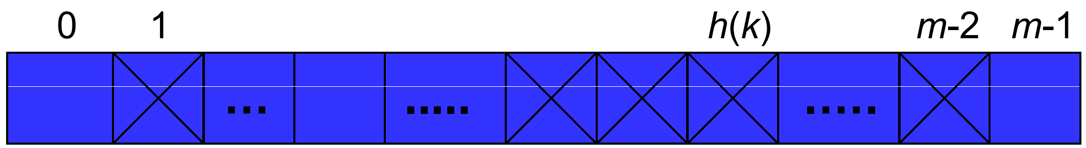
Store colliding elements in vacant ("open") buckets of the hash table

If $T[h(k)]$ is taken, find a different bucket for k according to a fixed rule

Example:

Consider the bucket with the next smaller index:

$$(h(k) - 1) \bmod m$$



General:

Consider the sequence

$$(h(k) - j) \bmod m$$

$j = 0, \ldots, m-1$

Probe sequence

Even more general:

Consider the probe sequence

$$(h(k) - s(j,k)) \bmod m$$

$j = 0, ..., m\text{-}1$, for a given function $s(j,k)$

Examples for the function

$s(j,k) = j$ (linear probing)

$s(j,k) = (-1)^j * \left\lceil \frac{j}{2} \right\rceil^2$ (quadratic probing)

$s(j,k) = j * h'(k)$ (double hashing)

Probe sequence

Properties of $s(j,k)$

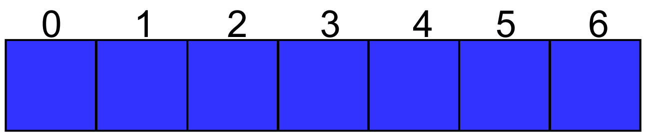
Sequence

$$(h(k) - s(0,k)) \bmod m,$$
$$(h(k) - s(1,k)) \bmod m,$$

$$(h(k) - s(m\text{-}2,k)) \bmod m,$$
$$(h(k) - s(m\text{-}1,k)) \bmod m$$

should result in a permutation of 0, ..., m-1.

Example: Quadratic probing



$h(11) = 4$

$s(j,k)$ = -1,1,-4,4,-9,9

Critical:

Deletion of data sets → mark as deleted

(Insert 4, 18, 25; delete 4; lookup 18, 25)

Open addressing

```
class OpenHashTable extends HashTable {
        // in HashTable: TableEntry [] T;
        private int [] tag;
        static final int EMPTY = 0;
        static final int OCCUPIED = 1;
        static final int DELETED = 2;
        // Constructor
        OpenHashTable (int capacity) {
            super(capacity);
            tag = new int [capacity];
            for (int i = 0; i < capacity; i++) {
                tag[i] = EMPTY;
            }
        }
        // The hash function
        protected int h (Object key) {...}
        // Function s for probe sequence
        protected int s (int j, Object key) {
            // quadratic probing
            if (j % 2 == 0)
                return ((j + 1) / 2) * ((j + 1) / 2);
            else
                return -((j + 1) / 2) * ((j + 1) / 2);
            }
```

Open addressing - lookup

```
public int searchIndex (Object key) {
        /* searches for an entry with the given key in the hash table and
           returns the respective index or -1 */
        int i = h(key);
        int j = 1;  // next index of probing sequence
        while (tag[i] != EMPTY &&!key.equals(T[i].key)){
            // Next entry in probing sequence
            i = (h(key) - s(j++, key)) % capacity;
            if (i < 0)
                i = i + capacity;
        }
        if (key.equals(T[i].key) && tag[i] == OCCUPIED)
            return i;
        else
            return -1;
    }
    public Object search (Object key) {
        /* searches for an entry with the given key in the hash table and
           returns the respective value or NULL */
        int i = searchIndex (key);
        if (i >= 0)
            return T[i].value;
        else
            return null;
    }
```

Open addressing - insert

```
public void insert (Object key, Object value) {
        // inserts an entry with the given key and value
        int j = 1;  // next index of probing sequence
        int i = h(key);
        while (tag[i] == OCCUPIED) {
            i = (h(key) - s(j++, key)) % capacity;
            if (i < 0)
                i = i + capacity;
        }
        T[i] = new TableEntry(key, value);
        tag[i] = OCCUPIED;
    }
```

Open addressing - delete

```
public void delete (Object key) {
        // deletes entry with given key from the hash table
        int i = searchIndex(key);
        if (i >= 0) {
            // Successful search
            tag[i] = DELETED;
         }
    }
```

Test program

```
public class OpenHashingTest {
        public static void main(String args[]) {
            Integer[] t= new Integer[args.length];
            for (int i = 0; i < args.length; i++)
                t[i] = Integer.valueOf(args[i]);
            OpenHashTable h = new OpenHashTable (7);
            for (int i = 0; i <= t.length - 1; i++) {
                h.insert(t[i], null);#
                h.printTable ();
            }
            h.delete(t[0]); h.delete(t[1]);
            h.delete(t[6]); h.printTable();
        }
    }
```

Call:

```
java OpenHashingTest 12 53 5 15 2 19 43
```

Output (quadratic probing):

```
[ ]  [ ]  [ ]  [ ]  [ ]  (12) [ ]
[ ]  [ ]  [ ]  [ ]  (53) (12) [ ]
[ ]  [ ]  [ ]  [ ]  (53) (12) (5)
[ ]  (15) [ ]  [ ]  (53) (12) (5)
[ ]  (15) (2)  [ ]  (53) (12) (5)
(19) (15) (2)  [ ]  (53) (12) (5)
(19) (15) (2)  (43) (53) (12) (5)

(19) (15) (2)  {43} {53} {12} (5)
```

Probe sequences – linear probing

$s(j,k) = j$

Probe sequence for k:

$h(k), h(k)-1, \ldots, 0, m-1, \ldots, h(k)+1,$

Problem:

"primary clustering"

0	1	2	3	4	5	6
			5	53	12	

Pr (next object ends at position 2) = 4/7

Pr (next object ends at position 1) = 1/7

Long chains are extended with higher probability than short ones.

Efficiency of linear probing

Successful search:

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

Failed search:

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

α	C_n (successful)	C'_n (failed)
0.50	1.5	2.5
0.90	5.5	50.5
0.95	10.5	200.5
1.00	-	-

Efficiency of linear probing decreases drastically as soon as the load factor α gets close to the value 1.

Quadratic probing

$$s(j,k) = (-1)^j \ast \lceil \tfrac{j}{2} \rceil^2$$

Probe sequence for *k*:

h(*k*), *h*(*k*)+1, *h*(*k*)-1, *h*(*k*)+4, ...

Permutation, if $m = 4l + 3$ is prime.

Problem: secondary clustering, i.e. two synonyms *k* and *k´* always run through the same probe sequence.

Efficiency of quadratic probing

Successful search:

$$C_n \approx 1 - \frac{\alpha}{2} + ln\left(\frac{1}{(1-\alpha)}\right)$$

Failed search:

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + ln\left(\frac{1}{(1-\alpha)}\right)$$

α	C_n (successful)	C'_n (failed)
0.50	1.44	2.19
0.90	2.85	11.40
0.95	3.52	22.05
1.00	-	-

Uniform probing

$s(j,k) = \pi_k(j)$

π_k one of the m! permutations of {0, ..., m-1}

- only depends on k
- same probability for each permutation

$$C'_n \leq \frac{1}{1-\alpha}$$

$$C_n \approx \frac{1}{\alpha} * ln\left(\frac{1}{(1-\alpha)}\right)$$

α	C_n (successful)	C'_n(failed)
0.50	1.39	2
0.90	2.56	10
0.95	3.15	20
1.00	-	-

Random probing

Realization of uniform probing is very complex.

Alternative:

Random probing

$s(j,k)$ = random number depending on k

$s(j,k) = s(j',k)$ possible, but improbable

Double hashing

Idea: Choose another hash function h'

$$s(j,k) = j \cdot h'(k)$$

Probe sequence for k:

$$h(k),\ h(k)-h'(k),\ h(k)-2h'(k),\ \ldots$$

Requirement:

Probing sequence must correspond to a permutation of the hash addresses.

Hence:

$h'(k) \neq 0$ and $h'(k)$ no factor of m, i.e. $h'(k)$ does not divide m.

Example:

$h'(k) = 1 + (k \bmod (m-2))$

Example

Hash functions: $h(k) = k \bmod 7$

$h'(k) = 1 + k \bmod 5$

Insert sequence: 15, 22, 1, 29, 26

0	1	2	3	4	5	6	
	15						$h'(22) = 3$

0	1	2	3	4	5	6	
	15				22		$h'(1) = 2$

0	1	2	3	4	5	6	
	15				22	1	$h'(29) = 5$

0	1	2	3	4	5	6	
	15		29		22	1	$h'(26) = 2$

In this example we can do with a single probing step almost every time.

- Double hashing is as efficient as uniform probing.
- Double hashing is simpler to implement.