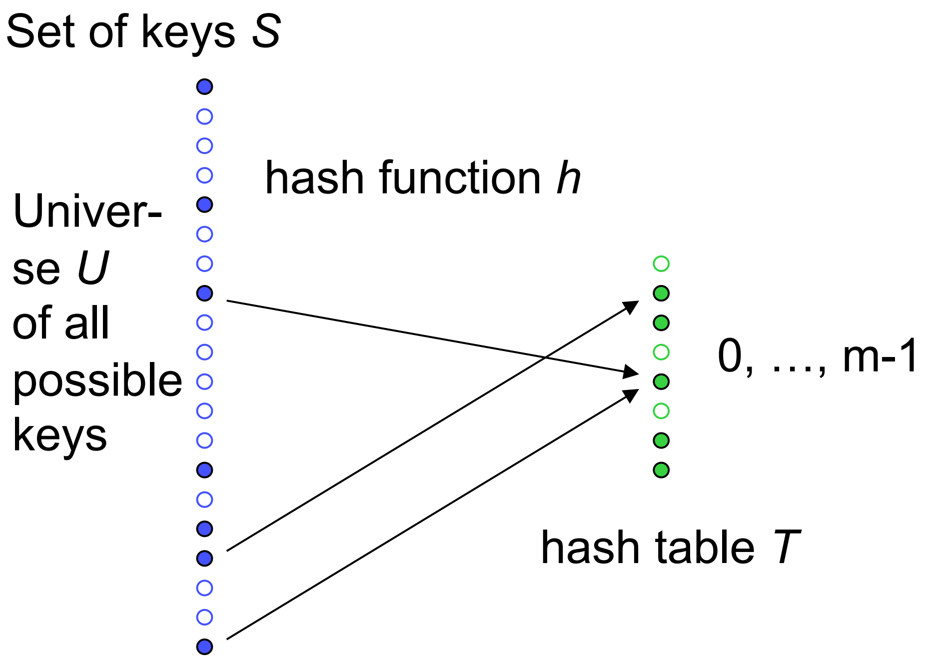
IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Theory I
Algorithm Design and Analysis

(7 Hashing: Open Addressing)

Prof. Th. Ottmann

Hashing: General Framework



$(H(U) \subseteq [-2^{31}, 2^{31} - 1])$

$h(s)$ = hash address

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

address collision

Possible ways of treating collisions

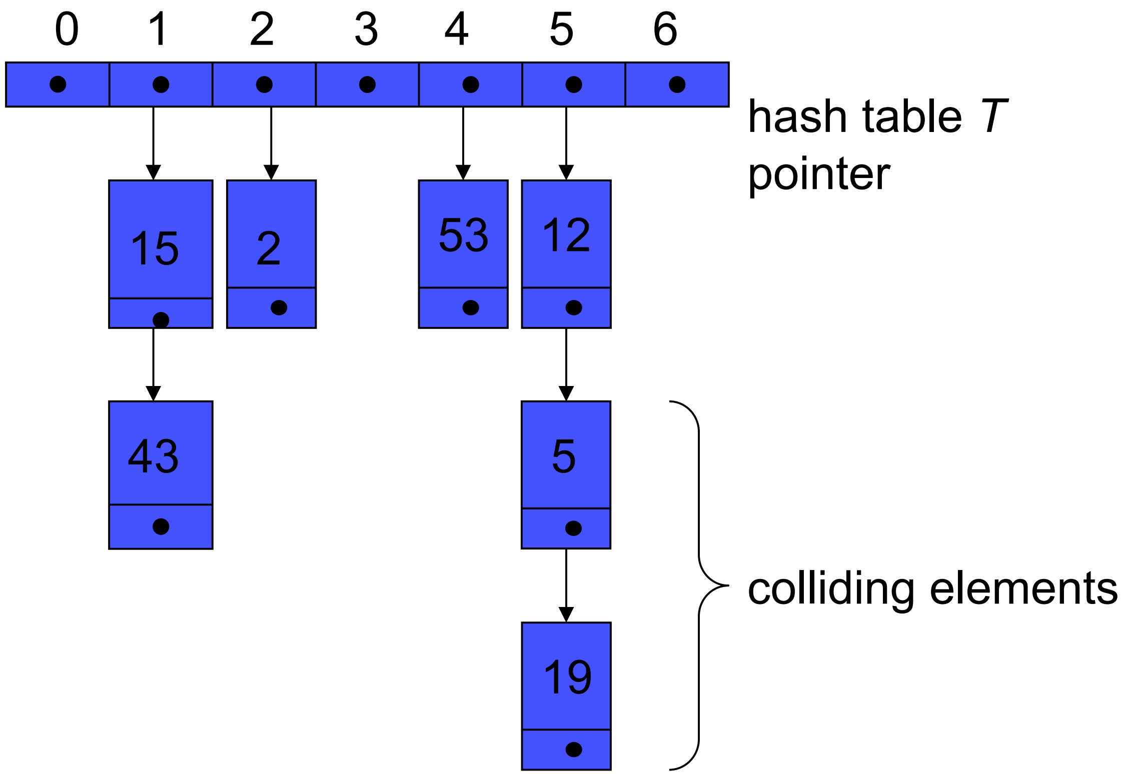
Treatment of collisions:

- 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

$h(k) = k \bmod 7$



This type of chaining is also known as direct chaining.

Open addressing

Idea:

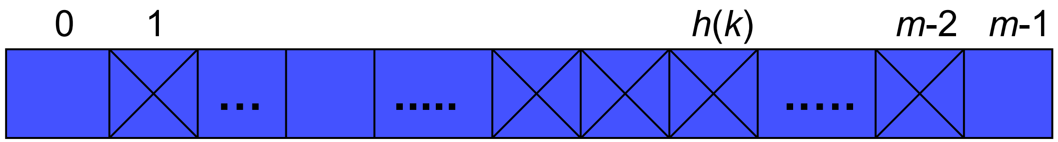
Store colliding elements in vacant (“open”) buckets of the hash table

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

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

$j = 0, \ldots, m\text{-}1$

Probe sequences

Even more general:

Consider the probe sequence

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

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

Properties of $s(j,k)$

Sequence

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

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

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

Example: Quadratic probing

0	1	2	3	4	5	6

$h(11) = 4$

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

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

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

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	-	-

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.

Improving successful search – motivation

Hash table of size 11; double hashing with

$h(k) = k \bmod 11$ and

$h'(k) = 1 + (k \bmod (11 - 2)) = 1 + (k \bmod 9)$

Already inserted: 22, 10, 37, 47, 17

Yet to be inserted: 6 and 30

$h(6) = 6$, $h'(6) = 1 + 6 = 7$

0	1	2	3	4	5	6	7	8	9	10
22			47	37		17				10

$h(30) = 8$, $h'(30) = 1 + 3 = 4$

0	1	2	3	4	5	6	7	8	9	10
22			47	37		6		17		10

Improving successful search

In general:

Insert:

- k collides with k_{old} in $T[i]$, i.e. $i = h(k) - s(j,k) = h(k_{old}) - s(j',k_{old})$
- k_{old} is already stored in $T[i]$

Idea:

Find a vacant bucket for k or k_{old}

Two options:

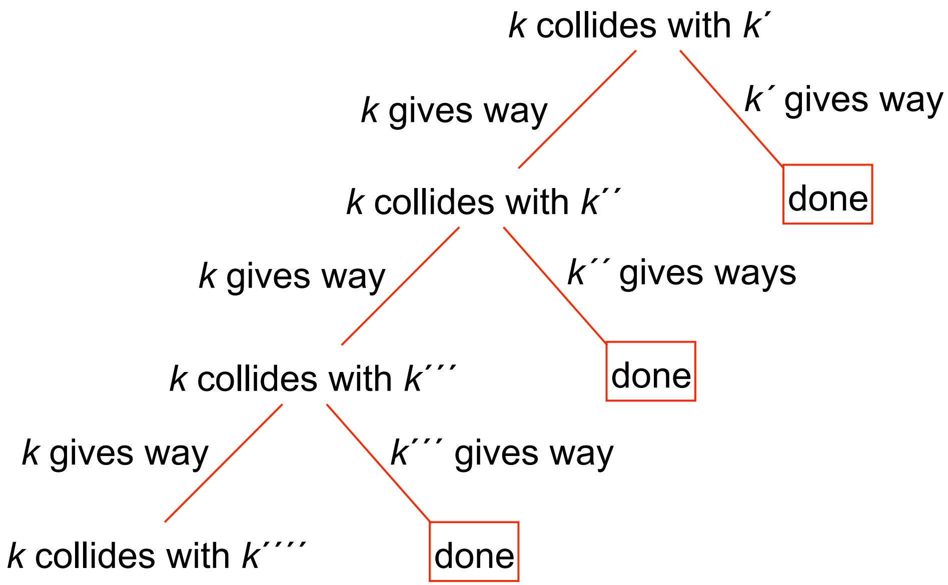
(O1) k_{old} remains in $T[i]$
consider new position $h(k) - s(j+1,k)$ for k

(O2) k replaces k_{old}
consider new position $h(k_{old}) - s(j'+1, k_{old})$ for k_{old}

if (O1) or (O2) finds a vacant bucket
then insert the respective key
done
else follow (O1) or (O2) further

Improving successful search

Brent’s method: only follow (O1)



Binary tree probing: follow (O1) and (O2)

Improving successful search

Problem: k_{old} replaced by k

→ next position in probe sequence for k_{old}?

Giving way is simple for k_{old} if:

$s(j, k_{old})$ - $s(j$ -1, $k_{old})$ = $s(1,k_{old})$

for all 1 ≤ j ≤ m -1.

This is, e.g., true for linear probing and double hashing.

$$C_n^{Brent} \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \ldots < 2.5$$

$$C_n' \approx \frac{1}{1-\alpha}$$

$$C_n^{Binarytree} \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \ldots < 2.2$$

Example

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

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

Insert sequence: 12, 53, 5, 15, 2, 19

0	1	2	3	4	5	6
				53	12	

$h(5) = 5$ occupied by $k' = 12$

Consider:

$h'(5) = 1 \rightarrow h(5) - 1 \cdot h'(5)$

→ 5 pushes 12 from its bucket

Improving unsuccessful search

Lookup *k*:

k´>k in probe sequence → lookup failed

Insert:

smaller keys push away greater keys

Invariant:

All keys in the probe sequence before *k* are smaller than *k*
(but not necessarily in ascending order)

Problems:

- The “pushing“ process may trigger a “chain reaction”
- *k´* pushed away by *k*: position of *k´* in probe sequence?

→ Required:

$$s(j,k) - s(j-1,k) = s(1,k), \quad 1 \le j \le m$$

Ordered hashing

Lookup

Input: key k

Output: Information about data set with key k, or *null*

Begin at $i \leftarrow h(k)$

while $T[i]$ not empty and $T[i].k < k$ do

$\quad i \leftarrow (i - s(1,k)) \bmod m$

end while;

if $T[i]$ occupied and $T[i].k = k$

$\quad$ then search successful

$\quad$ else search failed

Ordered hashing

Insert

Input: key k

```
Begin at i ← h(k)
while T[i] not empty and T[i] .k ≠ k do
    if k < T[i].k
        then if T[i] is removed
            then exit while-loop
            else  // k pushes away T[i].k
                swap T[i].k with k
    i = (i – s(1,k)) mod m
end while;
if T[i] is not occupied
    then insert k in T[i]
```