```
displacement( x, loc ):  // Calculate displacement of x from its ideal location of h( x ).
    return loc ≥ h( x ) ?  loc − h( x )  :  table.length + loc − h( x )
```

```
Robin Hood hashing add( x ):
    loc ← h( x );   d ← 0
    loop forever:
        if table [ loc ] = x  or table [ loc ] is free or deleted then
            table[ loc ] ← x;   return
        else if displacement( table[ loc ], loc ) ≥ d then
            d ← d + 1;   loc ← (loc + 1) % table.length
        else  // x has bigger displacement than element at loc, so replace it
            Exchange table [ loc ] ⇔ x
            loc ← (loc + 1) % table.length
            d ← displacement( x, loc )
```

```
Hopscotch hashing add( x ):
    loc ← h( x )
    while table[ loc ] is occupied do   // find loc closest to h( x ) that is available for x
        loc ← (loc + 1) % table.length
    while displacement( x, loc ) > d do
        failed ← true
        for  j ← 1 to d    // seek a volunteer to move down
            volunteerLoc ← loc − j < 0 ? loc − j + table.length : loc − j
            if displacement( table[ volunteerLoc ], loc ) ≤ d  then
                table[ loc ] ← table[ volunteerLoc ]
                Mark table[ volunteerLoc ] as deleted
                loc ← volunteerLoc
                failed ← false
                break
        if failed then   // no solution with displacement d; increase d, or rehash with new hash function
            d ← d + 1
    table[ loc ] ← x
```

**Applications of hashing**

1. Dictionaries with only add/contains/remove operations, associative arrays (maps)
2. Remove duplicates (especially during database query processing)
3. Cryptographic applications: confirmation numbers, preventing accidental access/update of wrong records, digital certificates, passwords, surrogate key generation, data transfer, bittorrent
4. Find duplicate web pages (in web crawlers)
5. Bloom filters (for malicious URL lookups in browsers):
   Detecting membership in a set S; use k hash functions $h_1 \cdots h_k$, and a bit array table[ 0..n − 1].
   for each $x \in S$, set table[ $h_i$( x ) ] ← 1, for $1 \leq i \leq k$.
   For a given y, if table[ $h_i$( y ) ] $\neq$ 1 for any $1 \leq i \leq k$, then y is not in S.
   Otherwise, y may be in S (false positive). A Bloom filter uses n = O( |S| ) and k = O( log n ).

**Multi-dimensional search**:

Suppose we have a dictionary of <Key, Value> pairs, where the keys are derived from a totally ordered set (i.e., elements are comparable). Then, storing elements in a balanced binary search tree (TreeMap), allows efficient implementation of the following operations: get, put, min, max, floor, ceiling, iteration of elements in sorted order of their keys.

What can be done, if in addition to the above operations, the following operations are also needed?
findValue( v ): find all keys whose associated value is equal to v.
removeValue( v ): remove all entries whose value field is equal to v.

If the operations are rare, an O( n ) algorithm that traverses the tree, looking for entries with value field equal to v, can be used. If these operations are frequent, then a better solution can be obtained by combining a binary search tree based on keys, and a hash table based on values.

```
Solution using TreeMap< Key, Value > tree  +  HashMap< Value, TreeSet<Key> > table:

add( key, value ):
    if tree has entry with key then
        reject add operation
        // Otherwise, to replace existing element, execute remove( key ) + add( key, value ).
    else
        tree.put( key, value )
        set ← table.get( value )
        if set is null then
            table.put( value, a new tree set containing key )
        else
            set.add( key )

remove( key ):
    value ← tree.remove( key )
    if value ≠ null then
        set ← table.get( value )
        if set.size( ) > 1 then
            set.remove( key )
        else
            table.remove( value )

findValue( value ):
    return table.get( value )

removeValue( value )
    set ← table.remove( value )
    if set ≠ null then
        for key in set do
            tree.remove( key )
```

## Bloom Filters

Problem: Set $S$.

with 99% certainty

Query: $x \in S$ ?
- Yes — Maybe (False positives allowed)
- No — Certain $x \notin S$

Challenge: Minimize size of database used.

Bloom Filters: $O(1)$ bits per element of $S$.
$\approx 8-10$

Application: Malicious URL lookup.

---

$k$ hash functions $h_1, h_2, \ldots h_k$ (independent)

$k \approx \log |S|$. Bit array table $[0..n-1]$

$n \approx c \cdot |S|$ $\quad c \approx 8-10$

for $x \in S$ do
  for $i \leftarrow 1$ to $k$ do
    table$[h_i(x)] \leftarrow 1$

Other entries of table are 0.

Send table$[0..n-1]$, $h_1, \ldots h_k$ to client

Bloom filter

---

client side: $x \in S$?

table $[h_i(x)] \neq 1$ for any $1 \le i \le k$

~~cat~~ check table $[h_i(x)]$
$\Rightarrow x \notin S$

If all entries are 1 $\longrightarrow$ check with google to see if $x \in S$.

---

```
add (key, value):
    if key exists in tree then
        reject add operation
        or remove(key), add(key,value)
    else
        tree.put(key, value)
        set ← table.get(value)
        if set = null then
            table.put(value, {key})
        else  set.add(key)
```

---

```
remove (key):
    value ← tree.remove(key)
    if value ≠ null then
        set ← table.get(value)
        if set.size() > 1 then
            set.remove(key)
        else  table.remove(value)
```

---

```
findValue(value):  return table.get(value)
```

---

```
removeValue(value):
    set ← table.~~get~~ remove(value)
    if set ≠ null then
        for key: set do
            tree.remove(key)
```