Hashing: subset of dictionary operations: add, contains, remove.
A function h, known as hash function, maps elements to non-negative integers in [0, n-1] where the table size is chosen to be n. Then x will be placed in table[h(x)], if possible.

Design goals:

1. Choose n proportional to number of elements in dictionary: λ = size / n, the load factor, is O(1).
2. For any two keys x and y, Pr{ h(x) = h(y) } = 1/n.
3. Pseudorandom function: h(1), h(2), h(3), $\cdots$ should be indistinguishable from a random sequence.
4. Deterministic, and easy to compute.

Implementation sketch:

add(x): Place x in table[h(x)]	**contains**(x): Is x in table[h(x)]?	**remove**(x): remove x from table[h(x)]

Collision resolution: What do you do if add(x) finds table[h(x)] is already occupied by another element?
(1) Separate chaining (known as open hashing): each entry of the hash table is a linked list of elements.
(2) Open addressing (closed hashing): each entry of the hash table can store only one element (or a small, fixed number of elements). Many schemes are available for collision resolution.

Java: Hash tables use separate chaining. Hash function is called hashCode(), and h(x) is a function of x.hashCode() and n. Table size is automatically adjusted based on load factor, and system tries to keep the load factor to be less than 0.5. In the base class of the object hierarchy, Object, hashCode is defined to be the address of the object. This is not a good hash function. Wrapper classes override it. User-defined classes that need to be used as keys in hashing should implement hashCode( ) and equals( ) methods.

The lengths of Java's hash tables are powers of 2 to simplify calculations. Bit operations are used to mangle the integer given by hashCode( ) to avoid problems created by poorly defined hash functions.

```
// Code extracted from Java's HashMap:
static int hash(int h) {
   // This function ensures that hashCodes that differ only by
   // constant multiples at each bit position have a bounded
   // number of collisions (approximately 8 at default load factor).
   h ^= (h >>> 20) ^ (h >>> 12);
   return h ^ (h >>> 7) ^ (h >>> 4);
}
static int indexFor(int h, int length) {      // length = table.length is a power of 2
   return h & (length-1);
}
// Key x is stored at table[ hash( x.hashCode( ) ) & ( table.length − 1 ) ].
```

Java hash tables: HashSet, HashMap, LinkedHashSet, ConcurrentHashMap, HashTable.
HashSet: implementation of Set interface. Main operations: add, contains, remove, iterator. add(x) is rejected if x is already in the set. HashSet is implemented using HashMap.
HashMap: Implementation of Map interface (key/value pairs). Main ops: : get, put, containsKey, remove, iterator. put operation replaces value if key already exists in map. get returns null if key does not exist.
LinkedHashSet: like HashSet, but iterator goes through elements in order of add.
ConcurrentHashMap, **HashTable**: synchronized, suitable for multi-threaded applications.

<u>Open addressing collision resolution schemes</u>: Each entry of the hash table can store a fixed number of elements. The algorithms use a sequence of probes at indexes $i_0, i_1, \cdots, i_k$. Probing stops when table[i_k] contains x, or, it is free. When an element is removed, that element of the table is marked as "deleted". The table is periodically reorganized when the load factor crosses a threshold (say, 0.5), or when a probing sequence is longer than some prescribed value. Elements are rehashed into the table, possibly with new hash functions, and deleted entries are marked as "free".

Linear probing: $i_k = h(x) + k \pmod{n}$. Advantage: simple algorithm. Disadvantage: clustering of nodes.
Quadratic probing: $i_k = h(x) + k^2 \pmod{n}$. Better than linear probing, but elements with h(x) = h(y) have the same probing sequence, and this leads to secondary clustering.
Double-hashing: a second hash function (g) is used to determine step length: $i_k = h(x) + k * g(x)$.

<u>Advanced hashing techniques</u>:
k-choice hashing: used with separate chaining. Choose k hash functions, $h_1, \cdots, h_k$. During add, the sizes of table[$h_i(x)$], $i = 1 \cdots k$ are examined, and the element is inserted into the list with the fewest entries. When searching for x, all k lists must be searched. The value of k is usually chosen to be 2, and this technique has much better worst-case performance than using just one hash function.

Cuckoo hashing: Choose k hash functions, $h_0, \cdots, h_{k-1}$. Element x will be placed in $table_i$ [h_i] for some i in [0, k-1]. Running time of contains and delete are O(k) in the worst case. Usually, k = 2 or 3. Add operation is more complex. Expected running time of add is O(1), with a threshold value of $\log n$. Scheme can be modified to have just one hash table and/or multiple elements stored at each hash table entry. A well designed hash table using cuckoo hashing can reach load factors of more than 90%, with worst case running times of O(1) for contains and remove, and expected times of O(1) for add.

```
add( x ): // Version of cuckoo hashing with k tables, each with its own hash function
    if table_i[ h_i( x ) ] = x, or table_i[ h_i( x ) ] is free or deleted, for some 0 ≤ i < k then
        table_i[ h_i( x ) ] ← x;  return
    i ← 0;   count ← 0
    while count++ < threshold do
        loc ← h_i( x )
        if table_i[ loc ] is free or deleted then
            table_i[ loc ] ← x;   return
        else
            Exchange table_i[ loc ] ⇔ x
            i ← ( i + 1 ) % k
    Too many steps (possible infinite loop).  Rebuild hash table with new hash functions.
```

Robin Hood hashing: an improvement over elementary methods like linear probing, quadratic probing, and double-hashing. Each element in the table also store the number of probes used during insertion of that element. A new element being inserted with a larger probe count can displace an existing element in the table with a smaller probe count. Variance in the number of probes to find elements is decreased.

Hopscotch hashing: combines ideas from linear probing and cuckoo hashing. An element x is placed within a distance of d from h(x). Elements are moved down to make place for new elements, provided there are vacant spots available within a distance of d from their hash index.

Hashing

insert / find / delete ops
add / contains / remove

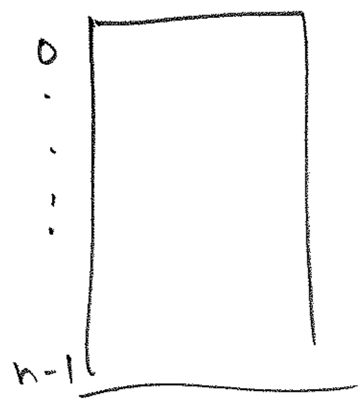
No min/max/floor/ceiling etc.

No ordering is assumed.

> Design Goal:
> RT of each operation $= O(1)$ (expected), independent of size of set.

set of elements S

$$h : S \longrightarrow [0 .. n-1]$$



array — hash table

Use $h(x)$ as index into table where x is stored.

Collision: $h(x) = h(y)$ for $x, y \in S$.

Open hashing
- each table element is a linked list.
(Separate Chaining) — Java's hash tables

Closed hashing
(also known as "open addressing")
Each row of table can store a fixed no. of elements say 1.

Hashing examples

Elements:

x	h(x)	g(x)
12497	14	5
18608	7	5
28754	7	2
34678	3	7
45500	14	7
56699	3	1
67891	4	2
70011	15	3
81209	3	5
99194	14	3

	Separate chaining	Linear probing	Quad probing	Double hashing
0		70011	70011	
1		99194	~~99194~~	99194
2			99194	
3	34678 → 56699 → 81209	34678	34678	34678
4	67891	~~34678~~ 56699	56699	56699
5		67891	67891	45500
6		81209		67891
7	18608 → 28754	18608	18608	18608
8		28754	28754	81209
9				28754
10				
11				
12			81209	
13				
14	12497 → 45500 → 99194	12497	12497	12497 +7
15	70011	45500	45500	70011

Elements:

x	h(x)
12497	14
18608	7
28754	7
34678	3
45500	14
56699	3
67891	4
70011	15
81209	3
99194	14
X	3

Robin Hood hashing:

0	~~70011 (1)~~ 99194 (2)
1	70011 (2)
2	
3	34678 (0)
4	56699 (1)
5	~~67891 (1)~~ 81209 (2)
6	~~67891 (2)~~ X (3)
7	~~18608 (0)~~ 67891 (3)
8	28754 (1)
9	18608 (2)
10	
11	
12	
13	
14	12497 (0)
15	45500 (1)

Limits variance of find.

Hopscotch hashing:

0	~~70011~~ (1) 99194 (2)
1	70011 (2)
2	
3	34678 (0)
4	56699 (1)
5	~~67891~~ (1) 81209 (2)
6	~~67891 (2)~~ X (3)
7	~~18608 (0)~~ 67891 (3)
8	~~28754 (1)~~ 18608 (1)
9	28754 (2)
10	
11	
12	
13	
14	12497 (0)
15	45500 (1)

Ø X ⊬ 3

Limits worst case.

Elements:

x	$h_0(x)$	$h_1(x)$
12497	14	15
18608	7	5
28754	7	2
34678	3	7
45500	14	8
56699	3	1
67891	4	12
70011	15	3
81209	3	0
99194	1	3

2-choice hashing:

0	81209
1	56699 → 99194
2	28754
3	34678
4	67891
5	
6	
7	18608
8	45500
9	
10	
11	
12	
13	
14	12497
15	70011

Cuckoo hashing:

0	81209
1	~~56699~~ 99194
2	28754
3	~~34678~~ 56699
4	67891
5	18608
6	
7	~~18608~~ 34678
8	45500
9	
10	
11	
12	
13	
14	12497
15	70011

Infinite loop is possible
RT: Contains/remove O(1) worst
add: O(1) expected