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.

Hashing

Java HashMap

$h(x) =$ indexFor (hash (hashCode (x)), table.length)

table.length is a power of 2.

Hashtables in Java

1. HashSet:

```
HashSet <Vertex> set = new HashSet<>();
```

Operations: add(x), contains(x), remove(x),
iterator() – no order is guaranteed

2. HashMap: Key, Value pairs.

```
HashMap<Vertex, Integer> map = new HashMap<>();
```

Operations: get(x), – value associated with key x.
(null if no such key exists)

put(x, value) – associate a new value with key x.
put returns old value associated with x.

iterator: Map.Entry <K,V>.

```
for ( Map.Entry < Vertex, Integer> ent : map.entrySet()) {
    Vertex u = ent.getKey();
    Integer val = ent.getValue();
    :
}
```

Typical application: Given an array of integers, return a list of its distinct elements.

List <Integer> distinct ~~unique~~ Elements (int [] arr) {

1 — Set <Integer> set = new HashSet <>();

expected n — for (Integer e : arr) { set.add(e); }

1 — List <Integer> result = new ArrayList <>();

n — for (Integer e : set) { result.add(e); }

1 — return result;

}

$O(n)$ expected time

Generalize integers $\longrightarrow$ arbitrary classes.

If user-defined class C is used as a HashSet/Map key, then C should implement:

public int hashCode ( ) $\longrightarrow$ hash code of object.

public boolean equals (Object other) {..}

– whether this object is equal to other object.

Implementation of HashSet/HashMap in Java:

≈ separate chaining.

- each table location is a linked list of elements.

Hash~~Map~~Set: List<T>[ ] table;

add(x): if (table[h(x)].contains(x))
 return false;
 else { table[h(x)].add(x);
 return true;
 }

contains(x): return table[h(x)].contains(x);

remove(x): return ~~remove~~ table[h(x)].remove(x);

iterator(): chained iterator of the lists' iterators.

↳ Iterator object: int index;
 Iterator<T> iter;

iterator of table[index]

Reality: It is not really a list.
 → TreeMap hacked to have compareTo( )
 and allow equal keys.

Tables are automatically resized when
 load factor exceeds some threshold.

Open addressing collision resolution schemes: Each entry of the hash table can store a fixed number of elements. The algorithms use a sequence of probes at indexes $i_0, i_1, ...i_k$. 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: $i_k = (h(x) + k)\ \%\ n$. Advantage: simple algorithm. Disadvantage: clustering of nodes.
Quadratic probing: $i_k = (h(x) + k^2)\ \%\ n$. 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 (h_2) is used to determine step length: $i_k = (h(x) + k * h_2(x))\ \%\ n$.

```
find( x ): // search for x and return index of x. If x is not found, return index where x can be added.
   k ← 0
   while true do
         if table [ i_k ] = x or table [ i_k ] is free then return i_k
         else if table [ i_k ] is deleted then break
         else k++
   xspot ← i_k
   while true do
         k++
         if table [ i_k ] = x then return i_k
         if table [ i_k ] is free then return xspot

contains( x ):
   loc ← find( x )
   if table [ loc ] = x then return true
   else return false

add( x ):
   loc ← find( x )
   if table [ loc ] = x then return false
   else { table [ loc ] ← x;  return true }

remove( x ):
   loc ← find( x )
   if table [ loc ] = x then
         result ← table [ loc ]
         mark table [ loc ] as deleted
         return result
   else return null
```

Chaining

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

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

Linear probing

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

cluster

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

Quadratic probing

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

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

Double hashing

Key	$h_1(x)$	$h_2(x)$
12497	14	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
18608	7	5

0		
1	99194	
2		
3	34678	
4	56699	
5	45500	← 14+7 mod 16 = 5
6	67891	
7	28754	
8	81209	
9		
10		
11		
12	18608	
13		
14	12497	
15	70011	