<u>Hashing</u>: 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: $\lambda = \text{size} / \text{n}$, 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):	contains(x):	remove(x):
Place x in table[h(x)]	Is x in table[h(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.

y Java HoshMap h(x) = index for (hash (hash (ode (x)), fable · length)table length is a power of 2. Hashtables in Javan 1. HashSet: HashSet (Ventex) set = new HashSet ()(); Operations: add (x), contains(x), remove (x),
iterator() - no order is guaranteed Key, Value païrs. 2. Hash Map: 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 dd value associated
iteratur: Map. Entry (x, v). for (Map Entry < Vertex, Integer) ent:

Vertex a = ent. get Key ();

Vertex a = ent. get Key (); Integer val = ent. get Value ():

Typical application: Given an array of integers, return a list of its distinct elements. List (Integer) dichnot Flement, (int [] avr) } Set (Integer) set = new Hach Set ()(); expedition - for (Integer e: arr) { set.add(e); } List (Integer) result = new ArrayList(>(); n - for (Integer e: set) { result. add(e);}
return result; D(n) expected the Generalize integers -> arbitrary classes. It user-defined class C is used as a Hach Set/major Key, then C should implement: public mt hash Code () - s hash code of object. public boolean equals (Object otter) !.. }

- whether this object is equal.

Implementation of Hash Set/Hish May in Java: of Separate charing. - each table location is a lonked list of elements. List (T) [] table; Hash Begt: if (table[h(x)]. contains (x))
return false; add(x): else { table [h(x)]. add(x);
return the;
} contains (x): return table [L(x)]. contains (x). remove (x): retur remove table[1(x)]. remove(x); chained iterator of the lists' iterators. iterator(): La Iberator object: intindex;
Therator (T) iter; iterator of table [index] Reality: It is not really a list.

Tree Map hacked to have compareto()

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 \leftarrow 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 \leftarrow 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 \leftarrow find(x)
  if table [loc] = x then return true
  else return false
add(x):
  loc \leftarrow find(x)
  if table [loc] = x then return false
  else { table [ loc ] \leftarrow x; return true }
remove(x):
  loc \leftarrow find(x)
  if table [loc] = x then
        result ← table [ loc ]
        mark table [ loc ] as deleted
        return result
  else return null
```

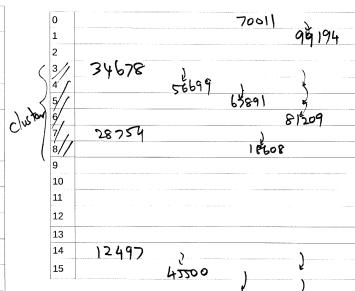
Chaining

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 -5699 -81209 67891
4	67891
5	
6	
7	28754 -> 18608
8	
9	
10	
11	**************************************
12	
13	
14	12497 -> 4500 -> 99194
15	70011

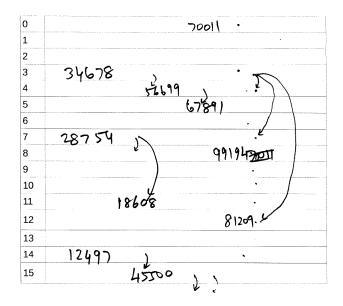
Linear probing

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



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	



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

