Hash tables using separate chaining (e.g. Java's HashSet/HashMap)

Dictionary with n entries

Hash table array with l lists.

Load factor $= \lambda = \frac{n}{l}$ = Average # of entries per list.

Well defined hash functions:

For any x, y: $Pr\{h(x) = h(y)\} = \frac{1}{l}$
(Probability)

Consider a dictionary with no correlation:

n entries are randomly, independently distributed;

Expected size of a list $= Pr\{h(x) \text{ lands in that list}\} * n$

$= \frac{n}{l} = \lambda$

Therefore if $\lambda = O(1)$:

expected RT of all hash table operations (add / contains / remove) $= O(1)$.

RT of iterator ops (hasNext(), next()) $= O(1)$, amortized

Note that worst case RT per op $= O(n)$.

Practical: Valid range for λ: $\lambda_{min} \leq \lambda \leq \lambda_{max}$

e.g. $\lambda_{min} = 0.2$ $\lambda_{max} = 0.6$

when $\lambda > \lambda_{max} \rightarrow$ rehash into bigger table | $\lambda < \lambda_{min} \rightarrow$ rehash into smaller table

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[$h_i(x)$] $\leftarrow 1$, for $1 \le i \le k$.

   For a given y, if table[$h_i(y)$] $\neq 1$ for any $1 \le i \le k$, then y is not in S.
   Otherwise, y may be in S (false positive). A Bloom filter uses $n = O(|S|)$ and $k = O(\log n)$.

<u>Multi-dimensional search</u>:

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 )
```