1 Class HashMap<K,V>

1.1 javaDoc

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values

Hash table based implementation of the **Map** interface. This implementation provides all of the optional map operations, and permits **null** values and the **null** key. (The **HashMap** class is roughly equivalent to **Hashtable**, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (**get** and **put**), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the **HashMap** instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of **HashMap** has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to **get** before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the **HashMap** class, including **get** and **put**). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a **HashMap** instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same **hashCode()** is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.

Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

The iterators returned by all of this class's "collection view methods" are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw **ConcurrentModificationException** on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

This class is a member of the Java Collections Framework.

1.2 Implementation notes.

This map usually acts as a binned (bucketed) hash table, but when bins get too large, they are transformed into bins of TreeNodes, each structured similarly to those in java.util.TreeMap. Most methods try to use normal bins, but relay to TreeNode methods when applicable (simply by checking instanceof a node). Bins of TreeNodes may be traversed and used like any others, but additionally support faster lookup when overpopulated. However, since the vast majority of bins in normal use are not overpopulated, checking for existence of tree bins may be delayed in the course of table methods.

Tree bins (i.e., bins whose elements are all TreeNodes) are ordered primarily by hashCode, but in the case of ties, if two elements are of the same "class C implements Comparable<C>", type then their compareTo method is used for ordering. (We conservatively check generic types via reflection to validate this – see method comparableClassFor). The added complexity of tree bins is worthwhile in providing worst-case O(log n) operations when keys either have distinct hashes or are orderable, Thus, performance degrades gracefully under accidental or malicious usages in which hashCode() methods return values that are poorly distributed, as well as those in which many keys share a hashCode, so long as they are also Comparable. (If neither of these apply, we may waste about a factor of two in time and space compared to taking no precautions. But the only known cases stem from poor user programming practices that are already so slow that this makes little difference.)

Because TreeNodes are about twice the size of regular nodes, we use them only when bins contain enough nodes to warrant use (see $TREEIFY\_THRESHOLD$). And when they become too small (due to removal or resizing) they are converted back to plain bins. In usages with well-distributed user hashCodes, tree bins are rarely used. Ideally, under random hashCodes, the frequency of nodes in bins follows a Poisson distribution (http://en.wikipedia.org/wiki/Poisson_distribution) with a parameter of about 0.5 on average for the default resizing threshold of 0.75, although with a large variance because of resizing granularity. Ignoring variance, the expected occurrences of list size k are $(exp(-0.5) * pow(0.5, k)/factorial(k))$. The first values are:

0: 0.60653066
1: 0.30326533
2: 0.07581633
3: 0.01263606
4: 0.00157952
5: 0.00015795

6: 0.00001316
7: 0.00000094
8: 0.00000006
more: less than 1 in ten million

The root of a tree bin is normally its first node. However, sometimes (currently only upon Iterator.remove), the root might be elsewhere, but can be recovered following parent links (method TreeNode.root()).

All applicable internal methods accept a hash code as an argument (as normally supplied from a public method), allowing them to call each other without recomputing user hashCodes. Most internal methods also accept a ”tab” argument, that is normally the current table, but may be a new or old one when resizing or converting.

When bin lists are treeified, split, or untreeified, we keep them in the same relative access/traversal order (i.e., field Node.next) to better preserve locality, and to slightly simplify handling of splits and traversals that invoke iterator.remove. When using comparators on insertion, to keep a total ordering (or as close as is required here) across rebalancings, we compare classes and identityHashCodes as tie-breakers.

The use and transitions among plain vs tree modes is complicated by the existence of subclass LinkedHashMap. See below for hook methods defined to be invoked upon insertion, removal and access that allow LinkedHashMap internals to otherwise remain independent of these mechanics. (This also requires that a map instance be passed to some utility methods that may create new nodes.)

The concurrent-programming-like SSA-based coding style helps avoid aliasing errors amid all of the twisty pointer operations.

1.3 source code understanding