File - /Users/john/Projects/code-catalog-python/catalog/suggested/maps/hash.py

```python
def hash_code(s):
    mask = (1 << 32) - 1  # limit to 32-bit integers
    h = 0
    for character in s:
        h = (h << 5 & mask) | (h >> 27)  # 5-bit cyclic shift of running sum
        h += ord(character)  # add in value of next character
    return h
```

File - /Users/john/Projects/code-catalog-python/catalog/suggested/maps/map_base.py

```python
from collections import MutableMapping


class MapBase(MutableMapping):
    """Our own abstract base class that includes a nonpublic _Item class."""

    # ------------------- nested _Item class -------------------
    class _Item:
        """Lightweight composite to store key-value pairs as map items."""
        __slots__ = '_key', '_value'

        def __init__(self, k, v):
            self._key = k
            self._value = v

        def __eq__(self, other):
            return self._key == other._key  # compare items based on their keys

        def __ne__(self, other):
            return not (self == other)  # opposite of __eq__

        def __lt__(self, other):
            return self._key < other._key  # compare items based on their keys
```

```python
class MultiMap:
    """
    A multimap class built upon use of an underlying map for storage.

    This uses dict for default storage.

    Subclasses can override class variable _MapType to change the default.
    That catalog class must have a default constructor that produces an empty map.
    As an example, one might define the following subclass to use a SortedTableMap

    class SortedTableMultimap(MultiMap):
      _MapType = SortedTableMap
    """
    _MapType = dict  # Map type; can be redefined by subclass

    def __init__(self):
        """Create a new empty multimap instance."""
        self._map = self._MapType()  # create map instance for storage
        self._n = 0

    def __len__(self):
        """Return number of (k,v) pairs in multimap."""
        return self._n

    def __iter__(self):
        """Iterate through all (k,v) pairs in multimap."""
        for k, secondary in self._map.items():
            for v in secondary:
                yield (k, v)

    def add(self, k, v):
        """Add pair (k,v) to multimap."""
        container = self._map.setdefault(k, [])  # create empty list, if needed
        container.append(v)
        self._n += 1

    def pop(self, k):
        """Remove and return arbitrary (k,v) pair with key k (or raise KeyError)."""
        secondary = self._map[k]  # may raise KeyError
        v = secondary.pop()
        if len(secondary) == 0:
            del self._map[k]  # no pairs left
        self._n -= 1
        return (k, v)

    def find(self, k):
        """Return arbitrary (k,v) pair with given key (or raise KeyError)."""
        secondary = self._map[k]  # may raise KeyError
        return (k, secondary[0])

    def find_all(self, k):
        """Generate iteration of all (k,v) pairs with given key."""
        secondary = self._map.get(k, [])  # empty list, by default
        for v in secondary:
            yield (k, v)
```

```python
from random import randrange  # used to pick MAD parameters

from .map_base import MapBase


class HashMapBase(MapBase):
    """Abstract base class for map using hash-table with MAD compression.

    Keys must be hashable and non-None.
    """

    def __init__(self, cap=11, p=109345121):
        """Create an empty hash-table map.

        cap     initial table size (default 11)
        p       positive prime used for MAD (default 109345121)
        """
        self._table = cap * [None]
        self._n = 0  # number of entries in the map
        self._prime = p  # prime for MAD compression
        self._scale = 1 + randrange(p - 1)  # scale from 1 to p-1 for MAD
        self._shift = randrange(p)  # shift from 0 to p-1 for MAD

    def _hash_function(self, k):
        return (hash(k) * self._scale + self._shift) % self._prime % len(self._table)

    def __len__(self):
        return self._n

    def __getitem__(self, k):
        j = self._hash_function(k)
        return self._bucket_getitem(j, k)  # may raise KeyError

    def __setitem__(self, k, v):
        j = self._hash_function(k)
        self._bucket_setitem(j, k, v)  # subroutine maintains self._n
        if self._n > len(self._table) // 2:  # keep load factor <= 0.5
            self._resize(2 * len(self._table) - 1)  # number 2^x - 1 is often prime

    def __delitem__(self, k):
        j = self._hash_function(k)
        self._bucket_delitem(j, k)  # may raise KeyError
        self._n -= 1

    def _resize(self, c):
        """Resize bucket array to capacity c and rehash all items."""
        old = list(self.items())  # use iteration to record existing items
        self._table = c * [None]  # then reset table to desired capacity
        self._n = 0  # n recomputed during subsequent adds
        for (k, v) in old:
            self[k] = v  # reinsert old key-value pair
```

File - /Users/john/Projects/code-catalog-python/catalog/suggested/maps/chain_hash_map.py

```python
from .hash_map_base import HashMapBase
from .unsorted_table_map import UnsortedTableMap


class ChainHashMap(HashMapBase):
    """Hash map implemented with separate chaining for collision resolution."""

    def _bucket_getitem(self, j, k):
        bucket = self._table[j]
        if bucket is None:
            raise KeyError('Key Error: ' + repr(k))  # no match found
        return bucket[k]  # may raise KeyError

    def _bucket_setitem(self, j, k, v):
        if self._table[j] is None:
            self._table[j] = UnsortedTableMap()  # bucket is new to the table
        oldsize = len(self._table[j])
        self._table[j][k] = v
        if len(self._table[j]) > oldsize:  # key was new to the table
            self._n += 1  # increase overall map size

    def _bucket_delitem(self, j, k):
        bucket = self._table[j]
        if bucket is None:
            raise KeyError('Key Error: ' + repr(k))  # no match found
        del bucket[k]  # may raise KeyError

    def __iter__(self):
        for bucket in self._table:
            if bucket is not None:  # a nonempty slot
                for key in bucket:
                    yield key
```

```python
from .hash_map_base import HashMapBase


class ProbeHashMap(HashMapBase):
    """Hash map implemented with linear probing for collision resolution."""
    _AVAIL = object()  # sentinal marks locations of previous deletions

    def _is_available(self, j):
        """Return True if index j is available in table."""
        return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

    def _find_slot(self, j, k):
        """Search for key k in bucket at index j.

        Return (success, index) tuple, described as follows:
        If match was found, success is True and index denotes its location.
        If no match found, success is False and index denotes first available slot.
        """
        firstAvail = None
        while True:
            if self._is_available(j):
                if firstAvail is None:
                    firstAvail = j  # mark this as first avail
                if self._table[j] is None:
                    return (False, firstAvail)  # search has failed
            elif k == self._table[j]._key:
                return (True, j)  # found a match
            j = (j + 1) % len(self._table)  # keep looking (cyclically)

    def _bucket_getitem(self, j, k):
        found, s = self._find_slot(j, k)
        if not found:
            raise KeyError('Key Error: ' + repr(k))  # no match found
        return self._table[s]._value

    def _bucket_setitem(self, j, k, v):
        found, s = self._find_slot(j, k)
        if not found:
            self._table[s] = self._Item(k, v)  # insert new item
            self._n += 1  # size has increased
        else:
            self._table[s]._value = v  # overwrite existing

    def _bucket_delitem(self, j, k):
        found, s = self._find_slot(j, k)
        if not found:
            raise KeyError('Key Error: ' + repr(k))  # no match found
        self._table[s] = ProbeHashMap._AVAIL  # mark as vacated

    def __iter__(self):
        for j in range(len(self._table)):  # scan entire table
            if not self._is_available(j):
                yield self._table[j]._key
```

```python
from .sorted_table_map import SortedTableMap


class CostPerformanceDatabase:
    """Maintain a database of maximal (cost,performance) pairs."""

    def __init__(self):
        """Create an empty database."""
        self._M = SortedTableMap()  # or a more efficient sorted map

    def best(self, c):
        """Return (cost,performance) pair with largest cost not exceeding c.

        Return None if there is no such pair.
        """
        return self._M.find_le(c)

    def add(self, c, p):
        """Add new entry with cost c and performance p."""
        # determine if (c,p) is dominated by an existing pair
        other = self._M.find_le(c)  # other is at least as cheap as c
        if other is not None and other[1] >= p:  # if its performance is as good,
            return  # (c,p) is dominated, so ignore
        self._M[c] = p  # else, add (c,p) to database
        # and now remove any pairs that are dominated by (c,p)
        other = self._M.find_gt(c)  # other more expensive than c
        while other is not None and other[1] <= p:
            del self._M[other[0]]
            other = self._M.find_gt(c)
```

```python
from .map_base import MapBase


class SortedTableMap(MapBase):
    """Map implementation using a sorted table."""

    # ---------------- nonpublic behaviors ----------------
    def _find_index(self, k, low, high):
        """Return index of the leftmost item with key greater than or equal to k.

        Return high + 1 if no such item qualifies.

        That is, j will be returned such that:
           all items of slice table[low:j] have key < k
           all items of slice table[j:high+1] have key >= k
        """
        if high < low:
            return high + 1  # no element qualifies
        else:
            mid = (low + high) // 2
            if k == self._table[mid]._key:
                return mid  # found exact match
            elif k < self._table[mid]._key:
                return self._find_index(k, low, mid - 1)  # Note: may return mid
            else:
                return self._find_index(k, mid + 1, high)  # answer is right of mid

    # ---------------- public behaviors ----------------
    def __init__(self):
        """Create an empty map."""
        self._table = []

    def __len__(self):
        """Return number of items in the map."""
        return len(self._table)

    def __getitem__(self, k):
        """Return value associated with key k (raise KeyError if not found)."""
        j = self._find_index(k, 0, len(self._table) - 1)
        if j == len(self._table) or self._table[j]._key != k:
            raise KeyError('Key Error: ' + repr(k))
        return self._table[j]._value

    def __setitem__(self, k, v):
        """Assign value v to key k, overwriting existing value if present."""
        j = self._find_index(k, 0, len(self._table) - 1)
        if j < len(self._table) and self._table[j]._key == k:
            self._table[j]._value = v  # reassign value
        else:
            self._table.insert(j, self._Item(k, v))  # adds new item

    def __delitem__(self, k):
        """Remove item associated with key k (raise KeyError if not found)."""
        j = self._find_index(k, 0, len(self._table) - 1)
        if j == len(self._table) or self._table[j]._key != k:
            raise KeyError('Key Error: ' + repr(k))
        self._table.pop(j)  # delete item

    def __iter__(self):
        """Generate keys of the map ordered from minimum to maximum."""
        for item in self._table:
            yield item._key

    def __reversed__(self):
        """Generate keys of the map ordered from maximum to minimum."""
        for item in reversed(self._table):
            yield item._key

    def find_min(self):
        """Return (key,value) pair with minimum key (or None if empty)."""
        if len(self._table) > 0:
            return (self._table[0]._key, self._table[0]._value)
        else:
            return None

    def find_max(self):
```

```
        """Return (key,value) pair with maximum key (or None if empty)."""
        if len(self._table) > 0:
            return (self._table[-1]._key, self._table[-1]._value)
        else:
            return None

    def find_le(self, k):
        """Return (key,value) pair with greatest key less than or equal to k.

        Return None if there does not exist such a key.
        """
        j = self._find_index(k, 0, len(self._table) - 1)  # j's key >= k
        if j < len(self._table) and self._table[j]._key == k:
            return (self._table[j]._key, self._table[j]._value)  # exact match
        elif j > 0:
            return (self._table[j - 1]._key, self._table[j - 1]._value)  # Note use of j
-1
        else:
            return None

    def find_ge(self, k):
        """Return (key,value) pair with least key greater than or equal to k.

        Return None if there does not exist such a key.
        """
        j = self._find_index(k, 0, len(self._table) - 1)  # j's key >= k
        if j < len(self._table):
            return (self._table[j]._key, self._table[j]._value)
        else:
            return None

    def find_lt(self, k):
        """Return (key,value) pair with greatest key strictly less than k.

        Return None if there does not exist such a key.
        """
        j = self._find_index(k, 0, len(self._table) - 1)  # j's key >= k
        if j > 0:
            return (self._table[j - 1]._key, self._table[j - 1]._value)  # Note use of j
-1
        else:
            return None

    def find_gt(self, k):
        """Return (key,value) pair with least key strictly greater than k.

        Return None if there does not exist such a key.
        """
        j = self._find_index(k, 0, len(self._table) - 1)  # j's key >= k
        if j < len(self._table) and self._table[j]._key == k:
            j += 1  # advanced past match
        if j < len(self._table):
            return (self._table[j]._key, self._table[j]._value)
        else:
            return None

    def find_range(self, start, stop):
        """Iterate all (key,value) pairs such that start <= key < stop.

        If start is None, iteration begins with minimum key of map.
        If stop is None, iteration continues through the maximum key of map.
        """
        if start is None:
            j = 0
        else:
            j = self._find_index(start, 0, len(self._table) - 1)  # find first result
        while j < len(self._table) and (stop is None or self._table[j]._key < stop):
            yield (self._table[j]._key, self._table[j]._value)
            j += 1
```

File - /Users/john/Projects/code-catalog-python/catalog/suggested/maps/unsorted_table_map.py

```python
from .map_base import MapBase


class UnsortedTableMap(MapBase):`
    """Map implementation using an unordered list."""

    def __init__(self):
        """Create an empty map."""
        self._table = []  # list of _Item's

    def __getitem__(self, k):
        """Return value associated with key k (raise KeyError if not found)."""
        for item in self._table:
            if k == item._key:
                return item._value
        raise KeyError('Key Error: ' + repr(k))

    def __setitem__(self, k, v):
        """Assign value v to key k, overwriting existing value if present."""
        for item in self._table:
            if k == item._key:  # Found a match:
                item._value = v  # reassign value
                return  # and quit
        # did not find match for key
        self._table.append(self._Item(k, v))

    def __delitem__(self, k):
        """Remove item associated with key k (raise KeyError if not found)."""
        for j in range(len(self._table)):
            if k == self._table[j]._key:  # Found a match:
                self._table.pop(j)  # remove item
                return  # and quit
        raise KeyError('Key Error: ' + repr(k))

    def __len__(self):
        """Return number of items in the map."""
        return len(self._table)

    def __iter__(self):
        """Generate iteration of the map's keys."""
        for item in self._table:
            yield item._key  # yield the KEY
```