Maps and Dictionaries

Maps

- A **map** is a searchable collection of items that are key-value pairs
- The main operations of a map are for searching, inserting, and deleting items
- Multiple items with the same key are not allowed
- Applications:
  - address book
  - student-record database

Dictionaries

- Python's **dict** class is arguably the most significant data structure in the language.
  - It represents an abstraction known as a **dictionary** in which unique **keys** are mapped to associated **values.**
- Here, we use the term "dictionary" when specifically discussing Python's dict class, and the term "map" when discussing the more general notion of the abstract data type.

The Map ADT (Using **dict** Syntax)

M[k]:	Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError. In Python, this is implemented with the special method __getitem__.
M[k] = v:	Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. In Python, this is implemented with the special method __setitem__.
del M[k]:	Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError. In Python, this is implemented with the special method __delitem__.
len(M):	Return the number of items in map M. In Python, this is implemented with the special method __len__.
iter(M):	The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method __iter__, and it allows loops of the form, **for** k **in** M.

More Map Operations

k in M:	Return True if the map contains an item with key k. In Python, this is implemented with the special __contains__ method.
M.get(k, d=None):	Return M[k] if key k exists in the map; otherwise return default value d. This provides a form to query M[k] without risk of a KeyError.
M.setdefault(k, d):	If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.
M.pop(k, d=None):	Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d (or raise KeyError if parameter d is None).

A Few More Map Operations

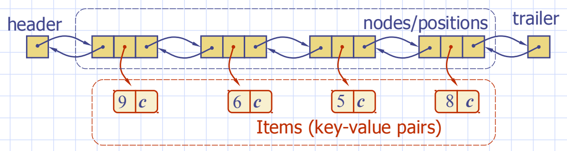
M.popitem():	Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.
M.clear():	Remove all key-value pairs from the map.
M.keys():	Return a set-like view of all keys of M.
M.values():	Return a set-like view of all values of M.
M.items():	Return a set-like view of (k,v) tuples for all entries of M.
M.update(M2):	Assign M[k] = v for every (k,v) pair in map M2.
M == M2:	Return True if maps M and M2 have identical key-value associations.
M != M2:	Return True if maps M and M2 do not have identical key-value associations.

Example

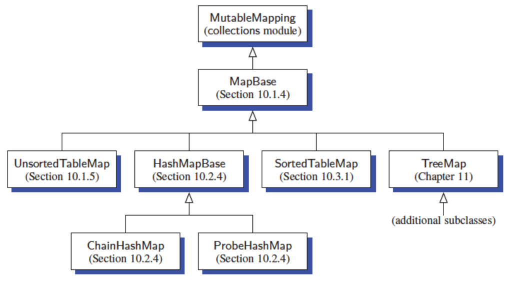
Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Our MapBase Class



The MapBase Abstract Class

```
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
```

An Unsorted List Implementation

```
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
```

Performance of a List-Based Map

- Performance:
  - Inserting an item takes $O(1)$ time since we can insert the new item at the beginning or at the end of the unsorted list
  - Searching for or removing an item takes $O(n)$ time, since in the worst case (the item is not found) we traverse the entire listto look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)