Sets and Maps

• Set Commands
• Set ADT
• Set ADT implementation
• Map ADT
• Map ADT implementation
Set

• The Set is a common container used in programming.

• Represents the mathematical set structure:
  • Stores *unordered* collection of object
  • Stores *unique* values.

• Has no index, sequence or keys.

• Fast Look Up – for checking membership not retrieval
Set

\[ A = \{2, 24, 17, 8\} \quad B = \{15, 24, 32\} \quad C = \{8, 24\} \]
Set: Equals

• Set A is equal to set B if both sets contain the same number of elements and both contain the exact same elements.

A == B?
Set: Subset of

- Set A is a subset of B if all elements of set A are subset of B.

\[ A \subseteq B? \]

\[ A = \{ 2, 24, 17, 8 \} \quad B = \{ 15, 24, 32 \} \quad C = \{ 8, 24 \} \]

- Exercises:
  - Is \( C \subseteq A? \)
  - Is \( C \subseteq B? \)
Set: Union

- The union of sets A and B contains all of the unique elements of both A and B.

\[ A \cup B? \]

\[ A = \{ 2, 24, 17, 8 \} \quad B = \{ 15, 24, 32 \} \quad C = \{ 8, 24 \} \]

\[ A \cup B = \{ 2, 8, 15, 17, 24, 32 \} \]
Set: Intersection

- The intersection of sets A and B contains only those elements that are in both A and B.

\[
A \cap B?
\]

\[
A = \{ 2, 24, 17, 8 \} \quad \text{B} = \{ 15, 24, 32 \} \quad \text{C} = \{ 8, 24 \}
\]

\[
A \cap B = \{ 24 \}
\]
Set: Difference

- The difference of sets A and B contains only those elements that are in A but not in B.

\[ A = \{ 2, 24, 17, 8 \} \quad B = \{ 15, 24, 32 \} \quad C = \{ 8, 24 \} \]

\[ A - B = \{ 2, 17, 8 \} \]
The Set ADT

- A set is a container that stores a collection of unique values over a given comparable domain.
- The stored values have no particular ordering.

<table>
<thead>
<tr>
<th>Set()</th>
<th>equals( setB )</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>isSubsetOf( setB )</td>
</tr>
<tr>
<td>contains( element )</td>
<td>union( setB )</td>
</tr>
<tr>
<td>add( element )</td>
<td>intersect( setB )</td>
</tr>
<tr>
<td>remove( element )</td>
<td>difference( setB )</td>
</tr>
<tr>
<td>iterator()</td>
<td>iterator()</td>
</tr>
</tbody>
</table>
Using the Set ADT

- Two sets containing the current courses for two students.

```python
smith = Set()
smith.add( "CSCI-112" )
smith.add( "MATH-121" )
smith.add( "HIST-340" )
smith.add( "ECON-101" )

roberts = Set()
roberts.add( "POL-101" )
roberts.add( "ANTH-230" )
roberts.add( "CSCI-112" )
roberts.add( "ECON-101" )
```
Using the Set ADT

- Determine if two students are taking:
  - all of the same courses or
  - any of the same courses.

```python
if smith == roberts :
    print( "Smith and Roberts are taking the same courses." )
else :
    sameCourses = smith.intersect( roberts )
    if len(sameCourses) == 0 :
        print( "Smith and Roberts are not taking any of "
                + "the same courses." )
    else :
        print( "Smith and Roberts are taking some of the "
               + "same courses:" )

    for course in sameCourses :
        print( course )
```
Using the Set ADT

- How can we determine which courses “Smith” is taking that “Roberts” is not taking?

  uniqueCourses = smith.difference( roberts )
Set: Which Data Structure?

- Evaluate each DS/ADT option:
  - dictionary
  - array
  - list

- Criteria:
  - storage requirements?
  - necessary access and manipulation functionality?
Set: Which Data Structure?

- Dictionary
  - stores unique elements (key/value pairs).
  - will waste space.

- Array
  - can store unique elements.
  - lacks the functionality.

- List
  - can store unique elements.
  - provides the functionality.
Set: List Implementation

```
Set
  theElements

0   “CSCI-112”
1   “MATH-121”
2   “HIST-340”
3   “ECON-101”

Set
  theElements

0   “POL-101”
1   “ANTH-230”
2   “CSCI-112”
3   “ECON-101”
```
class Set:
    def __init__(self):
        self._theElements = list()

    def __len__(self):
        return len(self._theElements)

    def __iter__(self):
        return _SetIterator(self._theElements)
The Set Class

class Set:
    # ...
    def __contains__(self, element):
        return element in self._theElements
The Set Class

class Set :
    # ...
    def add(self, element):
        if element not in self:
            self._theElements.append(element)
The Set Class

class Set :
    # ...
    def remove( self, element ):
        assert element in self, "The element must be in the set."
        self._theElements.remove( item )
The Set Class

class Set :
    # ...
    def isSubsetOf( self, setB ) :
        for element in self :
            if element not in setB :
                return False

    return True
class Set:
# ...
def __eq__(self, setB):
    if len(self) != len(setB):
        return False
    else:
        return self.isSubsetOf(setB)
The Set Class

class Set:
    # ...
    def union(self, setB):
        newSet = Set()
        newSet._theElements.extend(self._theElements)
        for element in setB:
            if element not in self:
                newSet._theElements.append(element)
        return newSet
Maps

- Stores a collection of records.
  - A unique `key` identifies each record.
  - Records are selected by key value.
- aka `dictionary`.
  - Python provides a built-in dictionary.
  - Great example for comparing different implementations.
Example Use

- Collection of student records managed by a university registrar:
The Map ADT

- A *map* is a container for storing a collection of data records:
  - Each record is associated with a unique key.
  - Key components must be comparable.

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map()</td>
</tr>
<tr>
<td>length()</td>
</tr>
<tr>
<td>contains( key )</td>
</tr>
<tr>
<td>add( key, value )</td>
</tr>
<tr>
<td>remove( key )</td>
</tr>
<tr>
<td>valueOf( key )</td>
</tr>
<tr>
<td>iterator()</td>
</tr>
</tbody>
</table>
Map: Which Data Structure?

- Evaluate each DS/ADT option:
  - dictionary
  - array
  - list
  - set
Map: Which Data Structure?

- **Array**
  - can store key/value pairs.
  - lacks the functionality.

- **List**
  - can store key/value pairs
  - provides the functionality.

- **Set**
  - same as a list.
  - not a simple implementation.
  - may not be efficient.
Map: List Implementation

- At this point, the list is the best choice.
- How should the data be stored and organized within the list?
Map: Using Dual Lists

- Use 2 lists in parallel:
  - one for the keys
  - one for the records.
Map: Using a Single List

- Use 2 lists in parallel:
  - must store both key and value at the same position.
  - use a storage class.
The MapEntry Class

# Storage object private to the module.

class _MapEntry :
    def __init__( self, key, value ):
        self.key = key
        self.value = value
The Map Class

class Map :
    # ...

    # Find and return the index position of a key # or return None.
    def _findPosition( self, key ):
        for i in range( len(self) ) :
            if self._entryList[i].key == key :
                return i
        return None
The Map Class

```python
class Map:
    def __init__(self):
        self._entryList = list()

    def __len__(self):
        return len(self._entryList)

    def __contains__(self, key):
        ndx = self._findPosition(key)
        return ndx is not None

    def __iter__(self):
        return _MapIterator(self._entryList)

# ... remaining Map methods go here ...
```

linearmap.py
class Map:
# ...

def add(self, key, value):
    ndx = self._findPosition(key)
    if ndx is not None:
        self._entryList[ndx].value = value
        return False
    else:
        entry = _MapEntry(key, value)
        self._entryList.append(entry)
        return True
The Map Class

class Map :
    # ...
    def valueOf( self, key ):
        ndx = self._findPosition( key )
        assert ndx is not None, "Invalid map key."
        return self._entryList[ndx].value

    def remove( self, key ):
        ndx = self._findPosition( key )
        assert ndx is not None, "Invalid map key."
        self._entryList.pop( ndx )