Sage Reference Manual: Data Structures

Release 7.2

The Sage Development Team

May 15, 2016

CHAPTER

ONE

BINARY TREES

Implements a binary tree in Cython.

AUTHORS:

- Tom Boothby (2007-02-15). Initial version free for any use (public domain).

class sage.misc.binary_tree.**BinaryTree**
Bases: object

A simple binary tree with integer keys.

contains(*key*)
Returns True if a node with the given key exists in the tree, and False otherwise.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.contains(1)
False
sage: t.insert(1,1)
sage: t.contains(1)
True
```

delete(*key*)
Removes a the node corresponding to key, and returns the value associated with it.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(3,3)
sage: t.insert(1,1)
sage: t.insert(2,2)
sage: t.insert(0,0)
sage: t.insert(5,5)
sage: t.insert(6,6)
sage: t.insert(4,4)
sage: t.delete(0)
0
sage: t.delete(3)
3
sage: t.delete(5)
5
sage: t.delete(2)
2
sage: t.delete(6)
6
```

```
sage: t.delete(1)
1
sage: t.delete(0)
sage: t.get_max()
4
sage: t.get_min()
4
```

get(*key*)
Returns the value associated with the key given.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(0,Matrix([[0,0],[1,1]]))
sage: t.insert(0,1)
sage: t.get(0)
[0 0]
[1 1]
```

get_max()
Returns the value of the node with the maximal key value.

get_min()
Returns the value of the node with the minimal key value.

insert(*key*, *value=None*)
Inserts a key-value pair into the BinaryTree. Duplicate keys are ignored. The first parameter, key, should be an int, or coercible (one-to-one) into an int.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(1)
sage: t.insert(0)
sage: t.insert(2)
sage: t.insert(0,1)
sage: t.get(0)
0
```

is_empty()
Returns True if the tree has no nodes.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.is_empty()
True
sage: t.insert(0,0)
sage: t.is_empty()
False
```

keys(*order='inorder'*)
Returns the keys sorted according to "order" parameter, which can be one of "inorder", "preorder", or "postorder"

pop_max()
Returns the value of the node with the maximal key value, and removes that node from the tree.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(4,'e')
sage: t.insert(2,'c')
sage: t.insert(0,'a')
sage: t.insert(1,'b')
sage: t.insert(3,'d')
sage: t.insert(5,'f')
sage: while not t.is_empty():
...    print t.pop_max()
f
e
d
c
b
a
```

pop_min()

Returns the value of the node with the minimal key value, and removes that node from the tree.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(4,'e')
sage: t.insert(2,'c')
sage: t.insert(0,'a')
sage: t.insert(1,'b')
sage: t.insert(3,'d')
sage: t.insert(5,'f')
sage: while not t.is_empty():
...    print t.pop_min()
a
b
c
d
e
f
```

values(*order='inorder'*)

Returns the keys sorted according to "order" parameter, which can be one of "inorder", "preorder", or "postorder"

class sage.misc.binary_tree.**Test**

binary_tree(*values=100*, *cycles=100000*)

Performs a sequence of random operations, given random inputs to stress test the binary tree structure. This was useful during development to find memory leaks / segfaults. Cycles should be at least 100 times as large as values, or the delete, contains, and get methods won't hit very often.

INPUT:

- `values` – number of possible values to use
- `cycles` – number of operations to perform

TESTS:

```
sage: sage.misc.binary_tree.Test().random()
```

random()

CHAPTER

TWO

BITSETS

A Python interface to the fast bitsets in Sage. Bitsets are fast binary sets that store elements by toggling bits in an array of numbers. A bitset can store values between 0 and `capacity - 1`, inclusive (where `capacity` is finite, but arbitrary). The storage cost is linear in `capacity`.

Warning: This class is most likely to be useful as a way to store Cython bitsets in Python data structures, acting on them using the Cython inline functions. If you want to use these classes for a Python set type, the Python `set` or `frozenset` data types may be faster.

class `sage.data_structures.bitset.`**Bitset**

Bases: *sage.data_structures.bitset.FrozenBitset*

A bitset class which leverages inline Cython functions for creating and manipulating bitsets. See the class documentation of *FrozenBitset* for details on the parameters of the constructor and how to interpret the string representation of a *Bitset*.

A bitset can be thought of in two ways. First, as a set of elements from the universe of the n natural numbers $0, 1, \ldots, n-1$ (where the capacity n can be specified), with typical set operations such as intersection, union, symmetric difference, etc. Secondly, a bitset can be thought of as a binary vector with typical binary operations such as `and`, `or`, `xor`, etc. This class supports both interfaces.

The interface in this class mirrors the interface in the `set` data type of Python.

Warning: This class is most likely to be useful as a way to store Cython bitsets in Python data structures, acting on them using the Cython inline functions. If you want to use this class for a Python set type, the Python `set` data type may be faster.

See also:

- *FrozenBitset*
- Python's set types

EXAMPLES:

```
sage: a = Bitset('1101')
sage: loads(dumps(a)) == a
True
sage: a = Bitset('1101' * 32)
sage: loads(dumps(a)) == a
True
```

add(*n*)

Update the bitset by adding `n`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.add(5)
sage: a
110001
sage: a.add(100)
sage: sorted(list(a))
[0, 1, 5, 100]
sage: a.capacity()
101
```

TESTS:

The input `n` must be an integer.

```
sage: Bitset('110').add(None)
Traceback (most recent call last):
...
TypeError: an integer is required
```

clear()

Removes all elements from the bitset.

EXAMPLES:

```
sage: a = Bitset('011')
sage: a.clear()
sage: a
000
sage: a = Bitset('011' * 32)
sage: a.clear()
sage: set(a)
set()
```

difference_update(*other*)

Update the bitset to the difference of `self` and `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.difference_update(Bitset('0101'))
sage: a
1000
sage: a_set = set(a)
sage: a.difference_update(FrozenBitset('010101' * 10)); a
1000000000000000000000000000000000000000000000000000000000000000000000000000000
sage: a_set.difference_update(FrozenBitset('010101' * 10))
sage: a_set == set(a)
True
sage: a.difference_update(FrozenBitset('110'))
sage: a_set.difference_update(FrozenBitset('110'))
sage: a_set == set(a)
True
sage: a.difference_update(FrozenBitset('01010' * 20)); a
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
sage: a_set.difference_update(FrozenBitset('01010' * 20))
sage: a_set == set(a)
True
sage: b = Bitset('10101' * 20)
sage: b_set = set(b)
```

```
sage: b.difference_update(FrozenBitset('1' * 5)); b
000001010110101101011010110101101011010110101101011010110101101011010110101101011010110101101011010
sage: b_set.difference_update(FrozenBitset('1' * 5))
sage: b_set == set(b)
True
```

TESTS:

```
sage: Bitset('110').difference_update(None)
Traceback (most recent call last):
...
TypeError: other cannot be None
```

discard(*n*)

Update the bitset by removing `n`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.discard(1)
sage: a
100
sage: a.discard(2)
sage: a.discard(4)
sage: a
100
sage: a = Bitset('000001' * 15); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83, 89]
sage: a.discard(83); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
sage: a.discard(82); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
```

TESTS:

The input `n` must be an integer.

```
sage: Bitset('110').discard(None)
Traceback (most recent call last):
...
TypeError: an integer is required
```

intersection_update(*other*)

Update the bitset to the intersection of `self` and `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.intersection_update(Bitset('0101'))
sage: a
0100
sage: a_set = set(a)
sage: a.intersection_update(Bitset('0110' * 25))
sage: a
0100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
sage: a_set.intersection_update(Bitset('0110' * 25))
sage: set(a) == a_set
True
```

TESTS:

```
sage: Bitset('110').intersection_update(None)
Traceback (most recent call last):
...
TypeError: other cannot be None
```

pop()

Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

EXAMPLES:

```
sage: a = Bitset('011')
sage: a.pop()
1
sage: a
001
sage: a.pop()
2
sage: a
000
sage: a.pop()
Traceback (most recent call last):
...
KeyError: 'pop from an empty set'
sage: a = Bitset('0001'*32)
sage: a.pop()
3
sage: [a.pop() for _ in range(20)]
[7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63, 67, 71, 75, 79, 83]
```

remove(*n*)

Update the bitset by removing `n`. Raises `KeyError` if `n` is not contained in the bitset.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.remove(1)
sage: a
100
sage: a.remove(2)
Traceback (most recent call last):
...
KeyError: 2L
sage: a.remove(4)
Traceback (most recent call last):
...
KeyError: 4L
sage: a
100
sage: a = Bitset('000001' * 15); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83, 89]
sage: a.remove(83); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
```

TESTS:

The input `n` must be an integer.

```
sage: Bitset('110').remove(None)
Traceback (most recent call last):
...
```

```
TypeError: an integer is required
```

symmetric_difference_update(*other*)

Update the bitset to the symmetric difference of `self` and `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.symmetric_difference_update(Bitset('0101'))
sage: a
1001
sage: a_set = set(a)
sage: a.symmetric_difference_update(FrozenBitset('010101' * 10)); a
1100010101010101010101010101010101010101010101010101010101
sage: a_set.symmetric_difference_update(FrozenBitset('010101' * 10))
sage: a_set == set(a)
True
sage: a.symmetric_difference_update(FrozenBitset('01010' * 20)); a
1001011111000001111100000111110000011111000001111100000111110101001010010100101001010010100101001
sage: a_set.symmetric_difference_update(FrozenBitset('01010' * 20))
sage: a_set == set(a)
True
sage: b = Bitset('10101' * 20)
sage: b_set = set(b)
sage: b.symmetric_difference_update( FrozenBitset('1' * 5)); b
010101010110101101011010110101101011010110101101011010110101101011010110101101011010110101101011010110
sage: b_set.symmetric_difference_update( FrozenBitset('1' * 5))
sage: b_set == set(b)
True
```

TESTS:

```
sage: Bitset('110').symmetric_difference_update(None)
Traceback (most recent call last):
...
TypeError: other cannot be None
```

update(*other*)

Update the bitset to include items in `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.update(Bitset('0101'))
sage: a
1101
sage: a_set = set(a)
sage: a.update(Bitset('00011' * 25))
sage: a
110110001100011000110001100011000110001100011000110001100011000110001100011000110001100011000110001100
sage: a_set.update(Bitset('00011' * 25))
sage: set(a) == a_set
True
```

TESTS:

During update, `other` cannot be `None`.

```
sage: a = Bitset('1101')
sage: a.update(None)
Traceback (most recent call last):
```

```
        ...
        TypeError: other cannot be None
```

class sage.data_structures.bitset.**FrozenBitset**

Bases: object

A frozen bitset class which leverages inline Cython functions for creating and manipulating bitsets.

A bitset can be thought of in two ways. First, as a set of elements from the universe of the n natural numbers $0, 1, \ldots, n-1$ (where the capacity n can be specified), with typical set operations such as intersection, union, symmetric difference, etc. Secondly, a bitset can be thought of as a binary vector with typical binary operations such as `and`, `or`, `xor`, etc. This class supports both interfaces.

The interface in this class mirrors the interface in the `frozenset` data type of Python. See the Python documentation on set types for more details on Python's `set` and `frozenset` classes.

Warning: This class is most likely to be useful as a way to store Cython bitsets in Python data structures, acting on them using the Cython inline functions. If you want to use this class for a Python set type, the Python `frozenset` data type may be faster.

INPUT:

- `iter` – initialization parameter (default: `None`). Valid input are:
  - *Bitset* and *FrozenBitset* – If this is a *Bitset* or *FrozenBitset*, then it is copied.
  - `None` – If `None`, then the bitset is set to the empty set.
  - string – If a nonempty string, then the bitset is initialized by including an element if the index of the string is `1`. If the string is empty, then raise a `ValueError`.
  - iterable – If an iterable, then it is assumed to contain a list of nonnegative integers and those integers are placed in the set.
- `capacity` – (default: `None`) The maximum capacity of the bitset. If this is not specified, then it is automatically calculated from the passed iterable. It must be at least one.

OUTPUT:

- None.

The string representation of a *FrozenBitset* `FB` can be understood as follows. Let $B = b_0 b_1 b_2 \cdots b_k$ be the string representation of the bitset `FB`, where each $b_i \in \{0, 1\}$. We read the b_i from left to right. If $b_i = 1$, then the nonnegative integer i is in the bitset `FB`. Similarly, if $b_i = 0$, then i is not in `FB`. In other words, `FB` is a subset of $\{0, 1, 2, \ldots, k\}$ and the membership in `FB` of each i is determined by the binary value b_i.

See also:

- *Bitset*
- Python's set types

EXAMPLES:

The default bitset, which has capacity 1:

```
sage: FrozenBitset()
0
sage: FrozenBitset(None)
0
```

Trying to create an empty bitset fails:

```
sage: FrozenBitset([])
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset(list())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset(())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset(tuple())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset("")
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
```

We can create the all-zero bitset as follows:

```
sage: FrozenBitset(capacity=10)
0000000000
sage: FrozenBitset([], capacity=10)
0000000000
```

We can initialize a *FrozenBitset* with a *Bitset* or another *FrozenBitset*, and compare them for equality. As they are logically the same bitset, the equality test should return `True`. Furthermore, each bitset is a subset of the other.

```
sage: def bitcmp(a, b, c):  # custom function for comparing bitsets
....:     print(a == b == c)
....:     print(a <= b, b <= c, a <= c)
....:     print(a >= b, b >= c, a >= c)
....:     print(a != b, b != c, a != c)
sage: a = Bitset("1010110"); b = FrozenBitset(a); c = FrozenBitset(b)
sage: a; b; c
1010110
1010110
1010110
sage: a < b, b < c, a < c
(False, False, False)
sage: a > b, b > c, a > c
(False, False, False)
sage: bitcmp(a, b, c)
True
(True, True, True)
(True, True, True)
(False, False, False)
```

Try a random bitset:

```
sage: a = Bitset(randint(0, 1) for n in range(1, randint(1, 10^4)))
sage: b = FrozenBitset(a); c = FrozenBitset(b)
sage: bitcmp(a, b, c)
True
(True, True, True)
```

```
(True, True, True)
(False, False, False)
```

A bitset with a hard-coded bitstring:

```
sage: FrozenBitset('101')
101
```

For a string, only those positions with `1` would be initialized to `1` in the corresponding position in the bitset. All other characters in the string, including `0`, are set to `0` in the resulting bitset.

```
sage: FrozenBitset('a')
0
sage: FrozenBitset('abc')
000
sage: FrozenBitset('abc1')
0001
sage: FrozenBitset('0abc1')
00001
sage: FrozenBitset('0abc10')
000010
sage: FrozenBitset('0a*c10')
000010
```

Represent the first 10 primes as a bitset. The primes are stored as a list and as a tuple. We then recover the primes from its bitset representation, and query the bitset for its length (how many elements it contains) and whether an element is in the bitset. Note that the length of a bitset is different from its capacity. The length counts the number of elements currently in the bitset, while the capacity is the number of elements that the bitset can hold.

```
sage: p = primes_first_n(10); p
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: tuple(p)
(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
sage: F = FrozenBitset(p); F; FrozenBitset(tuple(p))
0011010100010100010100010000001
0011010100010100010100010000001
```

Recover the primes from the bitset:

```
sage: for b in F:
....:     print b,
2 3 5 7 11 13 17 19 23 29
sage: list(F)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Query the bitset:

```
sage: len(F)
10
sage: len(list(F))
10
sage: F.capacity()
30
sage: s = str(F); len(s)
30
sage: 2 in F
True
sage: 1 in F
False
```

A random iterable, with all duplicate elements removed:

```
sage: L = [randint(0, 100) for n in range(1, randint(1, 10^4))]
sage: FrozenBitset(L) == FrozenBitset(list(set(L)))
True
sage: FrozenBitset(tuple(L)) == FrozenBitset(tuple(set(L)))
True
```

TESTS:

Loading and dumping objects:

```
sage: a = FrozenBitset('1101')
sage: loads(dumps(a)) == a
True
sage: a = FrozenBitset('1101' * 64)
sage: loads(dumps(a)) == a
True
```

If `iter` is a nonempty string and `capacity` is specified, then `capacity` must match the number of elements in `iter`:

```
sage: FrozenBitset("110110", capacity=3)
Traceback (most recent call last):
...
ValueError: bitset capacity does not match passed string
sage: FrozenBitset("110110", capacity=100)
Traceback (most recent call last):
...
ValueError: bitset capacity does not match passed string
```

The parameter `capacity` must be positive:

```
sage: FrozenBitset("110110", capacity=0)
Traceback (most recent call last):
...
ValueError: bitset capacity must be greater than 0
sage: FrozenBitset("110110", capacity=-2)
Traceback (most recent call last):
...
OverflowError: can't convert negative value to mp_bitcnt_t
```

capacity()

Return the size of the underlying bitset.

The maximum value that can be stored in the current underlying bitset is `self.capacity() - 1`.

EXAMPLES:

```
sage: FrozenBitset('11000').capacity()
5
sage: FrozenBitset('110' * 32).capacity()
96
sage: FrozenBitset(range(20), capacity=450).capacity()
450
```

complement()

Return the complement of self.

EXAMPLES:

```
sage: ~FrozenBitset('10101')
01010
sage: ~FrozenBitset('11111'*10)
00000000000000000000000000000000000000000000000000
sage: x = FrozenBitset('10'*40)
sage: x == ~x
False
sage: x == ~~x
True
sage: x|(~x) == FrozenBitset('11'*40)
True
sage: ~x == FrozenBitset('01'*40)
True
```

difference(*other*)

Return the difference of `self` and `other`.

EXAMPLES:

```
sage: FrozenBitset('10101').difference(FrozenBitset('11100'))
00001
sage: FrozenBitset('11111' * 10).difference(FrozenBitset('010101' * 10))
1010101010101010101010101010101010101010101010101010100000000000
```

TESTS:

```
sage: set(FrozenBitset('11111' * 10).difference(FrozenBitset('010101' * 10))) == set(FrozenB
True
sage: set(FrozenBitset('1' * 5).difference(FrozenBitset('01010' * 20))) == set(FrozenBitset(
True
sage: set(FrozenBitset('10101' * 20).difference(FrozenBitset('1' * 5))) == set(FrozenBitset(
True
sage: FrozenBitset('10101').difference(None)
Traceback (most recent call last):
...
ValueError: other cannot be None
```

intersection(*other*)

Return the intersection of `self` and `other`.

EXAMPLES:

```
sage: FrozenBitset('10101').intersection(FrozenBitset('11100'))
10100
sage: FrozenBitset('11111' * 10).intersection(FrozenBitset('010101' * 10))
0101010101010101010101010101010101010101010101010101000000000000
```

TESTS:

```
sage: set(FrozenBitset('11111' * 10).intersection(FrozenBitset('010101' * 10))) == set(froze
True
sage: set(FrozenBitset('1' * 5).intersection(FrozenBitset('01010' * 20))) == set(FrozenBitse
True
sage: set(FrozenBitset('10101' * 20).intersection(FrozenBitset('1' * 5))) == set(FrozenBitse
True
sage: FrozenBitset("101011").intersection(None)
Traceback (most recent call last):
...
ValueError: other cannot be None
```

isdisjoint(*other*)

Test to see if `self` is disjoint from `other`.

EXAMPLES:

```
sage: FrozenBitset('11').isdisjoint(FrozenBitset('01'))
False
sage: FrozenBitset('01').isdisjoint(FrozenBitset('001'))
True
sage: FrozenBitset('00101').isdisjoint(FrozenBitset('110' * 35))
False
```

TESTS:

```
sage: FrozenBitset('11').isdisjoint(None)
Traceback (most recent call last):
...
ValueError: other cannot be None
```

isempty()

Test if the bitset is empty.

INPUT:

- None.

OUTPUT:

- `True` if the bitset is empty; `False` otherwise.

EXAMPLES:

```
sage: FrozenBitset().isempty()
True
sage: FrozenBitset([1]).isempty()
False
sage: FrozenBitset([], capacity=110).isempty()
True
sage: FrozenBitset(range(99)).isempty()
False
```

issubset(*other*)

Test to see if `self` is a subset of `other`.

EXAMPLES:

```
sage: FrozenBitset('11').issubset(FrozenBitset('01'))
False
sage: FrozenBitset('01').issubset(FrozenBitset('11'))
True
sage: FrozenBitset('01').issubset(FrozenBitset('01' * 45))
True
```

TESTS:

```
sage: FrozenBitset('11').issubset(None)
Traceback (most recent call last):
...
ValueError: other cannot be None
```

issuperset(*other*)

Test to see if `self` is a superset of `other`.

EXAMPLES:

```
sage: FrozenBitset('11').issuperset(FrozenBitset('01'))
True
sage: FrozenBitset('01').issuperset(FrozenBitset('11'))
False
sage: FrozenBitset('01').issuperset(FrozenBitset('10' * 45))
False
```

TESTS:

```
sage: FrozenBitset('11').issuperset(None)
Traceback (most recent call last):
...
ValueError: other cannot be None
```

symmetric_difference(*other*)

Return the symmetric difference of `self` and `other`.

EXAMPLES:

```
sage: FrozenBitset('10101').symmetric_difference(FrozenBitset('11100'))
01001
sage: FrozenBitset('11111' * 10).symmetric_difference(FrozenBitset('010101' * 10))
1010101010101010101010101010101010101010101010101010100101010101
```

TESTS:

```
sage: set(FrozenBitset('11111' * 10).symmetric_difference(FrozenBitset('010101' * 10))) == s
True
sage: set(FrozenBitset('1' * 5).symmetric_difference(FrozenBitset('01010' * 20))) == set(Fro
True
sage: set(FrozenBitset('10101' * 20).symmetric_difference(FrozenBitset('1' * 5))) == set(Fro
True
sage: FrozenBitset('11111' * 10).symmetric_difference(None)
Traceback (most recent call last):
...
ValueError: other cannot be None
```

union(*other*)

Return the union of `self` and `other`.

EXAMPLES:

```
sage: FrozenBitset('10101').union(FrozenBitset('11100'))
11101
sage: FrozenBitset('10101' * 10).union(FrozenBitset('01010' * 10))
11111111111111111111111111111111111111111111111111
```

TESTS:

```
sage: set(FrozenBitset('10101' * 10).union(FrozenBitset('01010' * 10))) == set(FrozenBitset(
True
sage: set(FrozenBitset('10101').union(FrozenBitset('01010' * 20))) == set(FrozenBitset('1010
True
sage: set(FrozenBitset('10101' * 20).union(FrozenBitset('01010'))) == set(FrozenBitset('1010
True
sage: FrozenBitset('10101' * 10).union(None)
Traceback (most recent call last):
...
ValueError: other cannot be None
```

sage.data_structures.bitset.**test_bitset**(*py_a*, *py_b*, *n*)

Test the Cython bitset functions so we can have some relevant doctests.

TESTS:

```
sage: from sage.data_structures.bitset import test_bitset
sage: test_bitset('00101', '01110', 4)
a 00101
list a [2, 4]
a.size 5
len(a) 2
a.limbs 1
b 01110
a.in(n)   True
a.not_in(n)   False
a.add(n)     00101
a.discard(n)   00100
a.set_to(n)  00101
a.flip(n)    00100
a.set_first_n(n)    11110
a.first_in_complement()    4
a.isempty()  False
a.eq(b)      False
a.cmp(b)     1
a.lex_cmp(b) -1
a.issubset(b) False
a.issuperset(b) False
a.copy()     00101
r.clear()     00000
complement a        11010
a intersect b      00100
a union b       01111
a minus b      00001
a symmetric_difference b      01011
a.rshift(n)  10000
a.lshift(n)  00000
a.first()           2
a.next(n)           4
a.first_diff(b)     1
a.next_diff(b, n)   4
a.hamming_weight()  2
a.map(m)  10100
a == loads(dumps(a))  True
reallocating a      00101
to size 4          0010
to size 8          00100000
to original size    00100
```

```
sage: test_bitset('11101', '11001', 2)
a 11101
list a [0, 1, 2, 4]
a.size 5
len(a) 4
a.limbs 1
b 11001
a.in(n)   True
a.not_in(n)   False
a.add(n)     11101
a.discard(n)   11001
a.set_to(n)  11101
a.flip(n)    11001
```

```
    a.set_first_n(n)    11000
    a.first_in_complement()    2
    a.isempty()  False
    a.eq(b)      False
    a.cmp(b)     1
    a.lex_cmp(b) 1
    a.issubset(b) False
    a.issuperset(b) True
    a.copy()     11101
    r.clear()     00000
    complement a        00010
    a intersect b      11001
    a union b       11101
    a minus b      00100
    a symmetric_difference b      00100
    a.rshift(n)  10100
    a.lshift(n)  00111
    a.first()           0
    a.next(n)           2
    a.first_diff(b)     2
    a.next_diff(b, n)   2
    a.hamming_weight()  4
    a.map(m)  10111
    a == loads(dumps(a))  True
    reallocating a      11101
    to size 2          11
    to size 4          1100
    to original size    11000
```

Test a corner-case: a bitset that is a multiple of words:

```
    sage: test_bitset('00'*64, '01'*64, 127)
    a 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    list a []
    a.size 128
    len(a) 0
    a.limbs ...
    b 01010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
    a.in(n)   False
    a.not_in(n)   True
    a.add(n)     00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    a.discard(n)   00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    a.set_to(n)  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    a.flip(n)    00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    a.set_first_n(n)    11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
    a.first_in_complement()    127
    a.isempty()  True
    a.eq(b)      False
    a.cmp(b)     -1
    a.lex_cmp(b) -1
    a.issubset(b) True
    a.issuperset(b) False
    a.copy()     00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    r.clear()     00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    complement a        11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
    a intersect b      00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    a union b       01010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
    a minus b      00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
a symmetric_difference b      01010101010101010101010101010101010101010101010101010101010101010101
a.rshift(n)  00000000000000000000000000000000000000000000000000000000000000000000
a.lshift(n)  00000000000000000000000000000000000000000000000000000000000000000000
a.first()           -1
a.next(n)           -1
a.first_diff(b)     1
a.next_diff(b, n)   127
a.hamming_weight()  0
a.map(m)  00000000000000000000000000000000000000000000000000000000000000000000
a == loads(dumps(a))  True
rshifts add  True
lshifts add  True
intersection commutes True
union commutes  True
not not = id True
flipped bit  127
add bit      127
discard bit    127
lshift add unset ok True
rshift set unset ok True
reallocating a      00000000000000000000000000000000000000000000000000000000000000000000
to size 127          00000000000000000000000000000000000000000000000000000000000000000000
to size 254          00000000000000000000000000000000000000000000000000000000000000000000
to original size    00000000000000000000000000000000000000000000000000000000000000000000
```

Large enough to span multiple limbs. We don't explicitly check the number of limbs below because it will be different in the 32 bit versus 64 bit cases:

```
sage: test_bitset('111001'*25, RealField(151)(pi).str(2)[2:], 69)
a 11100111100111100111100111100111100111100111100111100111100111100111100111100111100111100111110
list a [0, 1, 2, 5, 6, 7, 8, 11, 12, 13, 14, 17, 18, 19, 20, 23, 24, 25, 26, 29, 30, 31, 32, 35,
a.size 150
len(a) 100
a.limbs ...
b 00010010000111111011010101000100010000101101000110000100011010011000100110001100110001010001010
a.in(n)   False
a.not_in(n)   True
a.add(n)     11100111100111100111100111100111100111100111100111100111100111100111100111100111100111101111100111100
a.discard(n)   11100111100111100111100111100111100111100111100111100111100111100111100111100111100111101001111001111
a.set_to(n)  11100111100111100111100111100111100111100111100111100111100111100111100111100111100111101111100111100
a.flip(n)    11100111100111100111100111100111100111100111100111100111100111100111100111100111100111101111100111100
a.set_first_n(n)    11111111111111111111111111111111111111111111111111111111111111111111111111111111111111110000000
a.first_in_complement()    69
a.isempty()  False
a.eq(b)      False
a.cmp(b)     -1
a.lex_cmp(b) 1
a.issubset(b) False
a.issuperset(b) False
a.copy()     11100111100111100111100111100111100111100111100111100111100111100111100111100111100111101111100111100
r.clear()     00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
complement a        00011000011000011000011000011000011000011000011000011000011000011000011000011000011000010000100001
a intersect b      00000010000111100011000101000100000000100101000110000100000010000000100110000
a union b       11110111100111111111110111100111111011111011111100111100111111111111111111001111011111
a minus b      11100101100000000100100010100011100111000010100001100011100101100111000001100011000
a symmetric_difference b      11110101100000011100110010100011110111001010100001100011111101111100
a.rshift(n)  00111100111100111100111100111100111100111100111100111100111100111100111100111100111100111100100
a.lshift(n)  00000000000000000000000000000000000000000000000000000000000000000000000000000000000000011100111100111
```

```
    a.first()           0
    a.next(n)           71
    a.first_diff(b)     0
    a.next_diff(b, n)   73
    a.hamming_weight()  100
    a.map(m)  1001111001111001111001111001111001111001111001111001111001111001111001111001111001111001111001111
    a == loads(dumps(a))  True
    rshifts add  True
    lshifts add  True
    intersection commutes True
    union commutes  True
    not not = id True
    flipped bit  69
    add bit      69
    discard bit    69
    lshift add unset ok True
    rshift set unset ok True
    reallocating a      1110011110011110011110011110011110011110011110011110011110011110011110011110011110011110011110
    to size 69          111001111001111001111001111001111001111001111001111001111001111001111001111
    to size 138          11100111100111100111100111100111100111100111100111100111100111100111100111100000000
    to original size    11100111100111100111100111100111100111100111100111100111100111100111100111100000000
```

sage.data_structures.bitset.**test_bitset_pop**(*py_a*)

Tests for the bitset_pop function.

TESTS:

```
    sage: from sage.data_structures.bitset import test_bitset_pop
    sage: test_bitset_pop('0101')
    a.pop()   1
    new set:  0001
    sage: test_bitset_pop('0000')
    Traceback (most recent call last):
    ...
    KeyError: 'pop from an empty set'
```

sage.data_structures.bitset.**test_bitset_remove**(*py_a*, *n*)

Test the bitset_remove function.

TESTS:

```
    sage: from sage.data_structures.bitset import test_bitset_remove
    sage: test_bitset_remove('01', 0)
    Traceback (most recent call last):
    ...
    KeyError: 0L
    sage: test_bitset_remove('01', 1)
    a 01
    a.size 2
    a.limbs 1
    n 1
    a.remove(n)   00
```

sage.data_structures.bitset.**test_bitset_set_first_n**(*py_a*, *n*)

Test the bitset function set_first_n.

TESTS:

```
    sage: from sage.data_structures.bitset import test_bitset_set_first_n
    sage: test_bitset_set_first_n('00'*64, 128)
```

```
    a.set_first_n(n)    11111111111111111111111111111111111111111111111111111111111111111111111111111
```

sage.data_structures.bitset.**test_bitset_unpickle**(*data*)

This (artificially) tests pickling of bitsets across systems.

INPUT:

- `data` – A tuple of data as would be produced by the internal, Cython-only, method `bitset_pickle`.

OUTPUT:

A list form of the bitset corresponding to the pickled data.

EXAMPLES:

We compare 64-bit and 32-bit encoding. Both should unpickle on any system:

```
    sage: from sage.data_structures.bitset import test_bitset_unpickle
    sage: test_bitset_unpickle((0, 100, 2, 8, (33, 6001)))
    [0, 5, 64, 68, 69, 70, 72, 73, 74, 76]
    sage: test_bitset_unpickle((0, 100, 4, 4, (33, 0, 6001, 0)))
    [0, 5, 64, 68, 69, 70, 72, 73, 74, 76]
```

CHAPTER

THREE

SEQUENCES OF BOUNDED INTEGERS

This module provides *BoundedIntegerSequence*, which implements sequences of bounded integers and is for many (but not all) operations faster than representing the same sequence as a Python `tuple`.

The underlying data structure is similar to `Bitset`, which means that certain operations are implemented by using fast shift operations from MPIR. The following boilerplate functions can be cimported in Cython modules:

- `cdef bint biseq_init(biseq_t R, mp_size_t l, mp_size_t itemsize) except -1`

  Allocate memory for a bounded integer sequence of length `l` with items fitting in `itemsize` bits.

- `cdef inline void biseq_dealloc(biseq_t S)`

  Deallocate the memory used by `S`.

- `cdef bint biseq_init_copy(biseq_t R, biseq_t S)`

  Initialize `R` as a copy of `S`.

- `cdef tuple biseq_pickle(biseq_t S)`

  Return a triple `(bitset_data, itembitsize, length)` defining `S`.

- `cdef bint biseq_unpickle(biseq_t R, tuple bitset_data, mp_bitcnt_t itembitsize, mp_size_t length) except -1`

  Initialise `R` from data returned by `biseq_pickle`.

- `cdef bint biseq_init_list(biseq_t R, list data, size_t bound) except -1`

  Convert a list to a bounded integer sequence, which must not be allocated.

- `cdef inline Py_hash_t biseq_hash(biseq_t S)`

  Hash value for `S`.

- `cdef inline int biseq_cmp(biseq_t S1, biseq_t S2)`

  Comparison of `S1` and `S2`. This takes into account the bound, the length, and the list of items of the two sequences.

- `cdef bint biseq_init_concat(biseq_t R, biseq_t S1, biseq_t S2) except -1`

  Concatenate `S1` and `S2` and write the result to `R`. Does not test whether the sequences have the same bound!

- `cdef inline bint biseq_startswith(biseq_t S1, biseq_t S2)`

  Is `S1=S2+something`? Does not check whether the sequences have the same bound!

- `cdef mp_size_t biseq_contains(biseq_t S1, biseq_t S2, mp_size_t start) except -2`

Return the position in `S1` of `S2` as a subsequence of `S1[start:]`, or `-1` if `S2` is not a subsequence. Does not check whether the sequences have the same bound!

- `cdef mp_size_t biseq_starswith_tail(biseq_t S1, biseq_t S2, mp_size_t start) except -2:`

  Return the smallest number `i` such that the bounded integer sequence `S1` starts with the sequence `S2[i:]`, where `start <= i < S1.length`, or return `-1` if no such `i` exists.

- `cdef mp_size_t biseq_index(biseq_t S, size_t item, mp_size_t start) except -2`

  Return the position in `S` of the item in `S[start:]`, or `-1` if `S[start:]` does not contain the item.

- `cdef size_t biseq_getitem(biseq_t S, mp_size_t index)`

  Return `S[index]`, without checking margins.

- `cdef size_t biseq_getitem_py(biseq_t S, mp_size_t index)`

  Return `S[index]` as Python `int` or `long`, without checking margins.

- `cdef biseq_inititem(biseq_t S, mp_size_t index, size_t item)`

  Set `S[index] = item`, without checking margins and assuming that `S[index]` has previously been zero.

- `cdef inline void biseq_clearitem(biseq_t S, mp_size_t index)`

  Set `S[index] = 0`, without checking margins.

- `cdef bint biseq_init_slice(biseq_t R, biseq_t S, mp_size_t start, mp_size_t stop, mp_size_t step) except -1`

  Initialise `R` with `S[start:stop:step]`.

AUTHORS:

- Simon King, Jeroen Demeyer (2014-10): initial version (trac ticket #15820)

class `sage.data_structures.bounded_integer_sequences.`**`BoundedIntegerSequence`**
Bases: `object`

A sequence of non-negative uniformely bounded integers.

INPUT:

- `bound` – non-negative integer. When zero, a `ValueError` will be raised. Otherwise, the given bound is replaced by the power of two that is at least the given bound.
- `data` – a list of integers.

EXAMPLES:

We showcase the similarities and differences between bounded integer sequences and lists respectively tuples.

To distinguish from tuples or lists, we use pointed brackets for the string representation of bounded integer sequences:

```
sage: from sage.data_structures.bounded_integer_sequences import BoundedIntegerSequence
sage: S = BoundedIntegerSequence(21, [2, 7, 20]); S
<2, 7, 20>
```

Each bounded integer sequence has a bound that is a power of two, such that all its item are less than this bound:

```
sage: S.bound()
32
sage: BoundedIntegerSequence(16, [2, 7, 20])
Traceback (most recent call last):
```

```
...
OverflowError: list item 20 larger than 15
```

Bounded integer sequences are iterable, and we see that we can recover the originally given list:

```
sage: L = [randint(0,31) for i in range(5000)]
sage: S = BoundedIntegerSequence(32, L)
sage: list(L) == L
True
```

Getting items and slicing works in the same way as for lists:

```
sage: n = randint(0,4999)
sage: S[n] == L[n]
True
sage: m = randint(0,1000)
sage: n = randint(3000,4500)
sage: s = randint(1, 7)
sage: list(S[m:n:s]) == L[m:n:s]
True
sage: list(S[n:m:-s]) == L[n:m:-s]
True
```

The *index()* method works different for bounded integer sequences and tuples or lists. If one asks for the index of an item, the behaviour is the same. But we can also ask for the index of a sub-sequence:

```
sage: L.index(L[200]) == S.index(L[200])
True
sage: S.index(S[100:2000])     # random
100
```

Similarly, containment tests work for both items and sub-sequences:

```
sage: S[200] in S
True
sage: S[200:400] in S
True
sage: S[200]+S.bound() in S
False
```

Bounded integer sequences are immutable, and thus copies are identical. This is the same for tuples, but of course not for lists:

```
sage: T = tuple(S)
sage: copy(T) is T
True
sage: copy(S) is S
True
sage: copy(L) is L
False
```

Concatenation works in the same way for lists, tuples and bounded integer sequences:

```
sage: M = [randint(0,31) for i in range(5000)]
sage: T = BoundedIntegerSequence(32, M)
sage: list(S+T)==L+M
True
sage: list(T+S)==M+L
True
sage: (T+S == S+T) == (M+L == L+M)
True
```

However, comparison works different for lists and bounded integer sequences. Bounded integer sequences are first compared by bound, then by length, and eventually by *reverse* lexicographical ordering:

```
sage: S = BoundedIntegerSequence(21, [4,1,6,2,7,20,9])
sage: T = BoundedIntegerSequence(51, [4,1,6,2,7,20])
sage: S < T   # compare by bound, not length
True
sage: T < S
False
sage: S.bound() < T.bound()
True
sage: len(S) > len(T)
True
```

```
sage: T = BoundedIntegerSequence(21, [0,0,0,0,0,0,0,0])
sage: S < T    # compare by length, not lexicographically
True
sage: T < S
False
sage: list(T) < list(S)
True
sage: len(T) > len(S)
True
```

```
sage: T = BoundedIntegerSequence(21, [4,1,5,2,8,20,9])
sage: T > S   # compare by reverse lexicographic ordering...
True
sage: S > T
False
sage: len(S) == len(T)
True
sage: list(S) > list(T) # direct lexicographic ordering is different
True
```

TESTS:

We test against various corner cases:

```
sage: BoundedIntegerSequence(16, [2, 7, -20])
Traceback (most recent call last):
...
OverflowError: can't convert negative value to size_t
sage: BoundedIntegerSequence(1, [0, 0, 0])
<0, 0, 0>
sage: BoundedIntegerSequence(1, [0, 1, 0])
Traceback (most recent call last):
...
OverflowError: list item 1 larger than 0
sage: BoundedIntegerSequence(0, [0, 1, 0])
Traceback (most recent call last):
...
ValueError: positive bound expected
sage: BoundedIntegerSequence(2, [])
<>
sage: BoundedIntegerSequence(2, []) == BoundedIntegerSequence(4, []) # The bounds differ
False
sage: BoundedIntegerSequence(16, [2, 7, 4])[1:1]
<>
```

bound()

Return the bound of this bounded integer sequence.

All items of this sequence are non-negative integers less than the returned bound. The bound is a power of two.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import BoundedIntegerSequence
sage: S = BoundedIntegerSequence(21, [4,1,6,2,7,20,9])
sage: T = BoundedIntegerSequence(51, [4,1,6,2,7,20,9])
sage: S.bound()
32
sage: T.bound()
64
```

index(*other*)

The index of a given item or sub-sequence of `self`

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import BoundedIntegerSequence
sage: S = BoundedIntegerSequence(21, [4,1,6,2,6,20,9,0])
sage: S.index(6)
2
sage: S.index(5)
Traceback (most recent call last):
...
ValueError: 5 is not in sequence
sage: S.index(BoundedIntegerSequence(21, [6, 2, 6]))
2
sage: S.index(BoundedIntegerSequence(21, [6, 2, 7]))
Traceback (most recent call last):
...
ValueError: not a sub-sequence
```

The bound of (sub-)sequences matters:

```
sage: S.index(BoundedIntegerSequence(51, [6, 2, 6]))
Traceback (most recent call last):
...
ValueError: not a sub-sequence
sage: S.index(0)
7
sage: S.index(S.bound())
Traceback (most recent call last):
...
ValueError: 32 is not in sequence
```

TESTS:

```
sage: S = BoundedIntegerSequence(10^9, [2, 2, 2, 1, 2, 4, 3, 3, 3, 2, 2, 0])
sage: S[11]
0
sage: S.index(0)
11
```

```
sage: S.index(-3)
Traceback (most recent call last):
...
ValueError: -3 is not in sequence
```

```
sage: S.index(2^100)
Traceback (most recent call last):
...
ValueError: 1267650600228229401496703205376 is not in sequence
sage: S.index("hello")
Traceback (most recent call last):
...
TypeError: an integer is required
```

list()

Converts this bounded integer sequence to a list

NOTE:

A conversion to a list is also possible by iterating over the sequence.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import BoundedIntegerSequence
sage: L = [randint(0,26) for i in range(5000)]
sage: S = BoundedIntegerSequence(32, L)
sage: S.list() == list(S) == L
True
```

The discussion at trac ticket #15820 explains why the following is a good test:

```
sage: (BoundedIntegerSequence(21, [0,0]) + BoundedIntegerSequence(21, [0,0])).list()
[0, 0, 0, 0]
```

maximal_overlap(*other*)

Returns `self`'s maximal trailing sub-sequence that `other` starts with.

Returns `None` if there is no overlap

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import BoundedIntegerSequence
sage: X = BoundedIntegerSequence(21, [4,1,6,2,7,2,3])
sage: S = BoundedIntegerSequence(21, [0,0,0,0,0,0,0])
sage: T = BoundedIntegerSequence(21, [2,7,2,3,0,0,0,0,0,0,0,1])
sage: (X+S).maximal_overlap(T)
<2, 7, 2, 3, 0, 0, 0, 0, 0, 0, 0>
sage: print (X+S).maximal_overlap(BoundedIntegerSequence(21, [2,7,2,3,0,0,0,0,0,1]))
None
sage: (X+S).maximal_overlap(BoundedIntegerSequence(21, [0,0]))
<0, 0>
sage: B1 = BoundedIntegerSequence(4,[1,2,3,2,3,2,3])
sage: B2 = BoundedIntegerSequence(4,[2,3,2,3,2,3,1])
sage: B1.maximal_overlap(B2)
<2, 3, 2, 3, 2, 3>
```

startswith(*other*)

Tells whether `self` starts with a given bounded integer sequence

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import BoundedIntegerSequence
sage: L = [randint(0,26) for i in range(5000)]
sage: S = BoundedIntegerSequence(27, L)
sage: L0 = L[:1000]
sage: T = BoundedIntegerSequence(27, L0)
```

```
sage: S.startswith(T)
True
sage: L0[-1] += 1
sage: T = BoundedIntegerSequence(27, L0)
sage: S.startswith(T)
False
sage: L0[-1] -= 1
sage: L0[0] += 1
sage: T = BoundedIntegerSequence(27, L0)
sage: S.startswith(T)
False
sage: L0[0] -= 1
```

The bounds of the sequences must be compatible, or *startswith()* returns `False`:

```
sage: T = BoundedIntegerSequence(51, L0)
sage: S.startswith(T)
False
```

sage.data_structures.bounded_integer_sequences.**NewBISEQ**(*bitset_data*, *itembitsize*, *length*)

Helper function for unpickling of *BoundedIntegerSequence*.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import BoundedIntegerSequence
sage: L = [randint(0,26) for i in range(5000)]
sage: S = BoundedIntegerSequence(32, L)
sage: loads(dumps(S)) == S     # indirect doctest
True
```

TESTS:

We test a corner case:

```
sage: S = BoundedIntegerSequence(8,[])
sage: S
<>
sage: loads(dumps(S)) == S
True
```

And another one:

```
sage: S = BoundedIntegerSequence(2*sys.maxsize, [8, 8, 26, 18, 18, 8, 22, 4, 17, 22, 22, 7, 12,
sage: loads(dumps(S))
<8, 8, 26, 18, 18, 8, 22, 4, 17, 22, 22, 7, 12, 4, 1, 7, 21, 7, 10, 10>
```

CHAPTER

FOUR

MUTABLE POSET

This module provides a class representing a finite partially ordered set (poset) for the purpose of being used as a data structure. Thus the posets introduced in this module are mutable, i.e., elements can be added and removed from a poset at any time.

To get in touch with Sage's "usual" posets, start with the page `Posets` in the reference manual.

4.1 Examples

4.1.1 First Steps

We start by creating an empty poset. This is simply done by

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P
poset()
```

A poset should contain elements, thus let us add them with

```
sage: P.add(42)
sage: P.add(7)
sage: P.add(13)
sage: P.add(3)
```

Let us look at the poset again:

```
sage: P
poset(3, 7, 13, 42)
```

We see that they elements are sorted using $\leq$ which exists on the integers $\mathbf{Z}$. Since this is even a total order, we could have used a more efficient data structure. Alternativly, we can write

```
sage: MP([42, 7, 13, 3])
poset(3, 7, 13, 42)
```

to add several elements at once on construction.

4.1.2 A less boring Example

Let us continue with a less boring example. We define the class

```
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
```

It is equipped with a $\leq$-operation such that $a \leq b$ if all entries of a are at most the corresponding entry of b. For example, we have

```
sage: a = T((1,1))
sage: b = T((2,1))
sage: c = T((1,2))
sage: a <= b, a <= c, b <= c
(True, True, False)
```

The last comparison gives `False`, since the comparison of the first component checks whether $2 \leq 1$.

Now, let us add such elements to a poset:

```
sage: Q = MP([T((1, 1)), T((3, 3)), T((4, 1)),
....:         T((3, 2)), T((2, 3)), T((2, 2))]); Q
poset((1, 1), (2, 2), (2, 3), (3, 2), (3, 3), (4, 1))
```

In the representation above, the elements are sorted topologically, smallest first. This does not (directly) show more structural information. We can overcome this and display a "wiring layout" by typing:

```
sage: print Q.repr_full(reverse=True)
poset((3, 3), (2, 3), (3, 2), (2, 2), (4, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors:   (3, 3), (4, 1)
+-- (3, 3)
|   +-- successors:   oo
|   +-- predecessors:   (2, 3), (3, 2)
+-- (2, 3)
|   +-- successors:   (3, 3)
|   +-- predecessors:   (2, 2)
+-- (3, 2)
|   +-- successors:   (3, 3)
|   +-- predecessors:   (2, 2)
+-- (2, 2)
|   +-- successors:   (2, 3), (3, 2)
|   +-- predecessors:   (1, 1)
+-- (4, 1)
|   +-- successors:   oo
|   +-- predecessors:   (1, 1)
+-- (1, 1)
|   +-- successors:   (2, 2), (4, 1)
|   +-- predecessors:   null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

Note that we use `reverse=True` to let the elements appear from largest (on the top) to smallest (on the bottom).

If you look at the output above, you'll see two additional elements, namely `oo` (∞) and `null` ($\emptyset$). So what are these strange animals? The answer is simple and maybe you can guess it already. The ∞-element is larger than every other element, therefore a successor of the maximal elements in the poset. Similarly, the $\emptyset$-element is smaller than any other element, therefore a predecessor of the poset's minimal elements. Both do not have to scare us; they are just there and sometimes useful.

AUTHORS:

- Daniel Krenn (2015)

ACKNOWLEDGEMENT:

- Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26.

4.2 Classes and their Methods

class sage.data_structures.mutable_poset.**MutablePoset**(*data=None*, *key=None*, *merge=None*, *can_merge=None*)

Bases: sage.structure.sage_object.SageObject

A data structure that models a mutable poset (partially ordered set).

INPUT:

- data – data from which to construct the poset. It can be any of the following:
  1. None (default), in which case an empty poset is created,
  2. a *MutablePoset*, which will be copied during creation,
  3. an iterable, whose elements will be in the poset.
- key – a function which maps elements to keys. If None (default), this is the identity, i.e., keys are equal to their elements.

  Two elements with the same keys are considered as equal; so only one of these two elements can be in the poset.

  This key is not used for sorting (in contrast to sorting-functions, e.g. sorted).
- merge – a function which merges its second argument (an element) to its first (again an element) and returns the result (as an element). If the return value is None, the element is removed from the poset.

  This hook is called by *merge()*. Moreover it is used during *add()* when an element (more precisely its key) is already in this poset.

  merge is None (default) is equivalent to merge returning its first argument. Note that it is not allowed that the key of the returning element differs from the key of the first input parameter. This means merge must not change the position of the element in the poset.
- can_merge – a function which checks whether its second argument can be merged to its first.

  This hook is called by *merge()*. Moreover it is used during *add()* when an element (more precisely its key) is already in this poset.

  can_merge is None (default) is equivalent to can_merge returning True in all cases.

OUTPUT:

A mutable poset.

You can find a short introduction and examples *here*.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
```

We illustrate the different input formats

1.No input:

```
sage: A = MP(); A
poset()
```

2.A *MutablePoset*:

```
sage: B = MP(A); B
poset()
sage: B.add(42)
sage: C = MP(B); C
poset(42)
```

3.An iterable:

```
sage: C = MP([5, 3, 11]); C
poset(3, 5, 11)
```

See also:

MutablePosetShell.

add(*element*)

Add the given object as element to the poset.

INPUT:

- `element` – an object (hashable and supporting comparison with the operator <=).

OUTPUT:

Nothing.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2))])
sage: print P.repr_full(reverse=True)
poset((4, 4), (1, 3), (1, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 1)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (1, 2)
|   +-- successors:   (1, 3)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 1)
```

```
        +-- (1, 1)
        |   +-- successors:   (1, 2), (2, 1)
        |   +-- predecessors: null
        +-- null
        |   +-- successors:   (1, 1)
        |   +-- no predecessors
        sage: P.add(T((2, 2)))
        sage: reprP = P.repr_full(reverse=True); print reprP
        poset((4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1))
        +-- oo
        |   +-- no successors
        |   +-- predecessors: (4, 4)
        +-- (4, 4)
        |   +-- successors:   oo
        |   +-- predecessors: (1, 3), (2, 2)
        +-- (1, 3)
        |   +-- successors:   (4, 4)
        |   +-- predecessors: (1, 2)
        +-- (2, 2)
        |   +-- successors:   (4, 4)
        |   +-- predecessors: (1, 2), (2, 1)
        +-- (1, 2)
        |   +-- successors:   (1, 3), (2, 2)
        |   +-- predecessors: (1, 1)
        +-- (2, 1)
        |   +-- successors:   (2, 2)
        |   +-- predecessors: (1, 1)
        +-- (1, 1)
        |   +-- successors:   (1, 2), (2, 1)
        |   +-- predecessors: null
        +-- null
        |   +-- successors:   (1, 1)
        |   +-- no predecessors
```

When adding an element which is already in the poset, nothing happens:

```
        sage: e = T((2, 2))
        sage: P.add(e)
        sage: P.repr_full(reverse=True) == reprP
        True
```

We can influence the behavior when an element with existing key is to be inserted in the poset. For example, we can perform an addition on some argument of the elements:

```
        sage: def add(left, right):
        ....:     return (left[0], ''.join(sorted(left[1] + right[1])))
        sage: A = MP(key=lambda k: k[0], merge=add)
        sage: A.add((3, 'a'))
        sage: A
        poset((3, 'a'))
        sage: A.add((3, 'b'))
        sage: A
        poset((3, 'ab'))
```

We can also deal with cancellations. If the return value of our hook-function is `None`, then the element is removed out of the poset:

```
        sage: def add_None(left, right):
        ....:     s = left[1] + right[1]
```

```
....:     if s == 0:
....:         return None
....:     return (left[0], s)
sage: B = MP(key=lambda k: k[0],
....:        merge=add_None)
sage: B.add((7, 42))
sage: B.add((7, -42))
sage: B
poset()
```

See also:

discard(), *pop()*, *remove()*.

TESTS:

```
sage: R = MP([(1, 1, 42), (1, 3, 42), (2, 1, 7),
....:         (4, 4, 42), (1, 2, 7), (2, 2, 7)],
....:        key=lambda k: T(k[2:3]))
sage: print R.repr_full(reverse=True)
poset((1, 1, 42), (2, 1, 7))
+-- oo
|   +-- no successors
|   +-- predecessors: (1, 1, 42)
+-- (1, 1, 42)
|   +-- successors:   oo
|   +-- predecessors: (2, 1, 7)
+-- (2, 1, 7)
|   +-- successors:   (1, 1, 42)
|   +-- predecessors: null
+-- null
|   +-- successors:   (2, 1, 7)
|   +-- no predecessors
```

```
sage: P = MP()
sage: P.add(None)
Traceback (most recent call last):
...
ValueError: None is not an allowed element.
```

clear()

Remove all elements from this poset.

INPUT:

Nothing.

OUTPUT:

Nothing.

See also:

discard(), *pop()*, *remove()*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(42); P
poset(42)
sage: P.clear()
```

```
sage: print P.repr_full()
poset()
+-- null
|   +-- no predecessors
|   +-- successors:   oo
+-- oo
|   +-- predecessors:   null
|   +-- no successors
```

contains(*key*)

Test whether `key` is encapsulated by one of the poset's elements.

INPUT:

- `key` – an object.

OUTPUT:

`True` or `False`.

See also:

shells(), *elements()*, *keys()*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP()
sage: P.add(T((1, 1)))
sage: T((1, 1)) in P  # indirect doctest
True
sage: T((1, 2)) in P  # indirect doctest
False
```

copy(*mapping=None*)

Create a shallow copy.

INPUT:

- `mapping` – a function which is applied on each of the elements.

OUTPUT:

A poset with the same content as `self`.

See also:

map(), *mapped()*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2))])
sage: Q = copy(P)  # indirect doctest
sage: P.repr_full() == Q.repr_full()
True
```

difference(**other*)

Return a new poset where all elements of this poset, which are contained in one of the other given posets, are removed.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT:

A poset.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.difference(Q)
poset(3, 7)
```

See also:

union(), *union_update()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

TESTS:

```
sage: P.difference(Q, Q)
poset(3, 7)
sage: P.difference(P)
poset()
sage: P.difference(Q, P)
poset()
```

difference_update(**other*)

Remove all elements of another poset from this poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT:

Nothing.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
```

```
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.difference_update(Q)
sage: P
poset(3, 7)
```

See also:

union(), *union_update()*, *difference()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

discard(*key*, *raise_key_error=False*)

Remove the given object from the poset.

INPUT:

- `key` – the key of an object.
- `raise_key_error` – (default: `False`) switch raising `KeyError` on and off.

OUTPUT:

Nothing.

If the element is not a member and `raise_key_error` is set (not default), raise a `KeyError`.

Note: As with Python's `set`, the methods *remove()* and *discard()* only differ in their behavior when an element is not contained in the poset: *remove()* raises a `KeyError` whereas *discard()* does not raise any exception.

This default behavior can be overridden with the `raise_key_error` parameter.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: P.discard(T((1, 2)))
sage: P.remove(T((1, 2)))
Traceback (most recent call last):
...
KeyError: 'Key (1, 2) is not contained in this poset.'
sage: P.discard(T((1, 2)))
```

See also:

add(), *clear()*, *remove()*, *pop()*.

element(*key*)

Return the element corresponding to `key`.

INPUT:

`key` – the key of an object.

OUTPUT:

An object.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(42)
sage: e = P.element(42); e
42
sage: type(e)
<type 'sage.rings.integer.Integer'>
```

See also:

shell(), *get_key()*.

elements(***kwargs*)

Return an iterator over all elements.

INPUT:

- `kwargs` – arguments are passed to *shells()*.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7])
sage: [(v, type(v)) for v in sorted(P.elements())]
[(3, <type 'sage.rings.integer.Integer'>),
 (7, <type 'sage.rings.integer.Integer'>),
 (42, <type 'sage.rings.integer.Integer'>)]
```

Note that

```
sage: it = iter(P)
sage: sorted(it)
[3, 7, 42]
```

returns all elements as well.

See also:

shells(), *shells_topological()*, *elements_topological()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

elements_topological(***kwargs*)

Return an iterator over all elements in topological order.

INPUT:

- `kwargs` – arguments are passed to *shells_topological()*.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
```

```
....:             return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: [(v, type(v)) for v in P.elements_topological()]
[((1, 1), <class '__main__.T'>),
 ((1, 2), <class '__main__.T'>),
 ((1, 3), <class '__main__.T'>),
 ((2, 1), <class '__main__.T'>),
 ((2, 2), <class '__main__.T'>),
 ((4, 4), <class '__main__.T'>)]
```

See also:

shells(), *shells_topological()*, *elements()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

get_key(*element*)

Return the key corresponding to the given element.

INPUT:

- `element` – an object.

OUTPUT:

An object (the key of `element`).

See also:

element(), *shell()*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.get_key(None) is None
True
sage: P.get_key((1, 2))
(1, 2)
sage: Q = MP(key=lambda k: k[0])
sage: Q.get_key((1, 2))
1
```

intersection(**other*)

Return the intersection of the given posets as a new poset

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT:

A poset.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.intersection(Q)
poset(42)
```

See also:

union(), *union_update()*, *difference()*, *difference_update()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

TESTS:

```
sage: P.intersection(P, Q, Q, P)
poset(42)
```

intersection_update(**other*)

Update this poset with the intersection of itself and another poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT:

Nothing.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal; `A.intersection_update(B)` and `B.intersection_update(A)` might result in different posets.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.intersection_update(Q)
sage: P
poset(42)
```

See also:

union(), *union_update()*, *difference()*, *difference_update()*, *intersection()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

is_disjoint(*other*)

Return whether another poset is disjoint to this poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset.

OUTPUT:

Nothing.

Note: If this poset uses a `key`-function, then all comparisons are performed on the keys of the elements (and not on the elements themselves).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.is_disjoint(Q)
False
sage: P.is_disjoint(Q.difference(P))
True
```

See also:

is_subset(), *is_superset()*, *union()*, *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*.

is_subset(*other*)

Return whether another poset contains this poset, i.e., whether this poset is a subset of the other poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset.

OUTPUT:

Nothing.

Note: If this poset uses a `key`-function, then all comparisons are performed on the keys of the elements (and not on the elements themselves).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.is_subset(Q)
False
sage: Q.is_subset(P)
False
sage: P.is_subset(P)
True
sage: P.is_subset(P.union(Q))
True
```

See also:

is_disjoint(), *is_superset()*, *union()*, *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*.

is_superset(*other*)
Return whether this poset contains another poset, i.e., whether this poset is a superset of the other poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset.

OUTPUT:

Nothing.

Note: If this poset uses a `key`-function, then all comparisons are performed on the keys of the elements (and not on the elements themselves).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.is_superset(Q)
False
sage: Q.is_superset(P)
False
sage: P.is_superset(P)
True
sage: P.union(Q).is_superset(P)
True
```

See also:

is_disjoint(), *is_subset()*, *union()*, *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*.

isdisjoint(*other*)
Alias of *is_disjoint()*.

issubset(*other*)
Alias of *is_subset()*.

issuperset(*other*)
Alias of *is_superset()*.

keys(***kwargs*)
Return an iterator over all keys of the elements.

INPUT:

- `kwargs` – arguments are passed to *shells()*.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7], key=lambda c: -c)
sage: [(v, type(v)) for v in sorted(P.keys())]
[(-42, <type 'sage.rings.integer.Integer'>),
 (-7, <type 'sage.rings.integer.Integer'>),
```

```
 (-3, <type 'sage.rings.integer.Integer'>)]

sage: [(v, type(v)) for v in sorted(P.elements())]
[(3, <type 'sage.rings.integer.Integer'>),
 (7, <type 'sage.rings.integer.Integer'>),
 (42, <type 'sage.rings.integer.Integer'>)]

sage: [(v, type(v)) for v in sorted(P.shells(),
....:                               key=lambda c: c.element)]
[(3, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 (7, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 (42, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>)]
```

See also:

shells(), *shells_topological()*, *elements()*, *elements_topological()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

keys_topological(***kwargs*)

Return an iterator over all keys of the elements in topological order.

INPUT:

- `kwargs` – arguments are passed to *shells_topological()*.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([(1, 1), (2, 1), (4, 4)],
....:        key=lambda c: c[0])
sage: [(v, type(v)) for v in P.keys_topological()]
[(1, <type 'sage.rings.integer.Integer'>),
 (2, <type 'sage.rings.integer.Integer'>),
 (4, <type 'sage.rings.integer.Integer'>)]
sage: [(v, type(v)) for v in P.elements_topological()]
[((1, 1), <type 'tuple'>),
 ((2, 1), <type 'tuple'>),
 ((4, 4), <type 'tuple'>)]
sage: [(v, type(v)) for v in P.shells_topological()]
[((1, 1), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 ((2, 1), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 ((4, 4), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>)]
```

See also:

shells(), *shells_topological()*, *elements()*, *elements_topological()*, *keys()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

map(*function*, *topological=False*, *reverse=False*)

Apply the given `function` to each element of this poset.

INPUT:

- `function` – a function mapping an existing element to a new element.
- `topological` – (default: `False`) if set, then the mapping is done in topological order, otherwise unordered.

- `reverse` – is passed on to topological ordering.

OUTPUT:

Nothing.

Note: Since this method works inplace, it is not allowed that `function` alters the key of an element.

Note: If `function` returns `None`, then the element is removed.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))],
....:        key=lambda e: e[:2])
sage: P.map(lambda e: e + (sum(e),))
sage: P
poset((1, 2, 3), (1, 3, 4), (2, 1, 3), (2, 2, 4), (4, 4, 8))
```

TESTS:

```
sage: P.map(lambda e: e if e[2] != 4 else None); P
poset((1, 2, 3), (2, 1, 3), (4, 4, 8))
```

See also:

copy(), *mapped()*.

mapped (*function*)

Return a poset where on each element the given `function` was applied.

INPUT:

- `function` – a function mapping an existing element to a new element.
- `topological` – (default: `False`) if set, then the mapping is done in topological order, otherwise unordered.
- `reverse` – is passed on to topological ordering.

OUTPUT:

A *MutablePoset*.

Note: `function` is not allowed to change the order of the keys, but changing the keys themselves is allowed (in contrast to *map()*).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
```

```
sage: P.mapped(lambda e: str(e))
poset('(1, 2)', '(1, 3)', '(2, 1)', '(2, 2)', '(4, 4)')
```

See also:

copy(), *map()*.

maximal_elements()

Return an iterator over the maximal elements of this poset.

INPUT:

Nothing.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((1, 2)), T((2, 2))])
sage: list(P.maximal_elements())
[(1, 3), (2, 2)]
```

See also:

minimal_elements()

merge(*key=None*, *reverse=False*)

Merge the given element with its successors/predecessors.

INPUT:

- `key` – the key specifying an element or `None` (default), in which case this method is called on each element in this poset.
- `reverse` – (default: `False`) specifies which direction to go first: `False` searches towards `'oo'` and `True` searches towards `'null'`. When `key=None`, then this also specifies which elements are merged first.

OUTPUT:

Nothing.

This method tests all (not necessarily direct) successors and predecessors of the given element whether they can be merged with the element itself. This is done by the `can_merge`-function of *MutablePoset*. If this merge is possible, then it is performed by calling *MutablePoset*'s `merge`-function and the corresponding successor/predecessor is removed from the poset.

Note: `can_merge` is applied in the sense of the condition of depth first iteration, i.e., once `can_merge` fails, the successors/predecessors are no longer tested.

Note: The motivation for such a merge behavior comes from asymptotic expansions: $O(n^3)$ merges with, for example, $3n^2$ or $O(n)$ to $O(n^3)$ (as n tends to ∞; see Wikipedia article Big_O_notation).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: key = lambda t: T(t[0:2])
sage: def add(left, right):
....:     return (left[0], left[1],
....:             ''.join(sorted(left[2] + right[2])))
sage: def can_add(left, right):
....:     return key(left) >= key(right)
sage: P = MP([(1, 1, 'a'), (1, 3, 'b'), (2, 1, 'c'),
....:         (4, 4, 'd'), (1, 2, 'e'), (2, 2, 'f')],
....:        key=key, merge=add, can_merge=can_add)
sage: Q = copy(P)
sage: Q.merge(T((1, 3)))
sage: print Q.repr_full(reverse=True)
poset((4, 4, 'd'), (1, 3, 'abe'), (2, 2, 'f'), (2, 1, 'c'))
+-- oo
|   +-- no successors
|   +-- predecessors:   (4, 4, 'd')
+-- (4, 4, 'd')
|   +-- successors:   oo
|   +-- predecessors:   (1, 3, 'abe'), (2, 2, 'f')
+-- (1, 3, 'abe')
|   +-- successors:   (4, 4, 'd')
|   +-- predecessors:   null
+-- (2, 2, 'f')
|   +-- successors:   (4, 4, 'd')
|   +-- predecessors:   (2, 1, 'c')
+-- (2, 1, 'c')
|   +-- successors:   (2, 2, 'f')
|   +-- predecessors:   null
+-- null
|   +-- successors:   (1, 3, 'abe'), (2, 1, 'c')
|   +-- no predecessors
sage: for k in P.keys():
....:     Q = copy(P)
....:     Q.merge(k)
....:     print 'merging %s: %s' % (k, Q)
merging (1, 2): poset((1, 2, 'ae'), (1, 3, 'b'),
                      (2, 1, 'c'), (2, 2, 'f'), (4, 4, 'd'))
merging (1, 3): poset((1, 3, 'abe'), (2, 1, 'c'),
                      (2, 2, 'f'), (4, 4, 'd'))
merging (4, 4): poset((4, 4, 'abcdef'))
merging (2, 1): poset((1, 2, 'e'), (1, 3, 'b'),
                      (2, 1, 'ac'), (2, 2, 'f'), (4, 4, 'd'))
merging (2, 2): poset((1, 3, 'b'), (2, 2, 'acef'), (4, 4, 'd'))
merging (1, 1): poset((1, 1, 'a'), (1, 2, 'e'), (1, 3, 'b'),
                      (2, 1, 'c'), (2, 2, 'f'), (4, 4, 'd'))
sage: Q = copy(P)
sage: Q.merge(); Q
poset((4, 4, 'abcdef'))
```

See also:

MutablePosetShell.merge()

TESTS:

```
sage: copy(P).merge(reverse=False) == copy(P).merge(reverse=True)
True
```

```
sage: P = MP(srange(4),
....:        merge=lambda l, r: l, can_merge=lambda l, r: l >= r); P
poset(0, 1, 2, 3)
sage: Q = P.copy()
sage: Q.merge(reverse=True); Q
poset(3)
sage: R = P.mapped(lambda x: x+1)
sage: R.merge(reverse=True); R
poset(4)
```

```
sage: P = MP(srange(4),
....:        merge=lambda l, r: r, can_merge=lambda l, r: l < r)
sage: P.merge()
Traceback (most recent call last):
...
RuntimeError: Stopping merge before started;
the can_merge-function is not reflexive.
```

minimal_elements()

Return an iterator over the minimal elements of this poset.

INPUT:

Nothing.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: list(P.minimal_elements())
[(1, 2), (2, 1)]
```

See also:

maximal_elements()

null

The shell $\emptyset$ whose element is smaller than any other element.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: z = P.null; z
null
sage: z.is_null()
True
```

See also:

oo(), *MutablePosetShell.is_null()*, *MutablePosetShell.is_special()*.

oo

The shell ∞ whose element is larger than any other element.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: oo = P.oo; oo
oo
sage: oo.is_oo()
True
```

See also:

null(), *MutablePosetShell.is_oo()*, *MutablePosetShell.is_special()*.

pop(***kwargs*)

Remove and return an arbitrary poset element.

INPUT:

- `kwargs` – arguments are passed to *shells_topological()*.

OUTPUT:

An object.

Note: The special elements `'null'` and `'oo'` cannot be popped.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(3)
sage: P
poset(3)
sage: P.pop()
3
sage: P
poset()
sage: P.pop()
Traceback (most recent call last):
...
KeyError: 'pop from an empty poset'
```

See also:

add(), *clear()*, *discard()*, *remove()*.

remove(*key*, *raise_key_error=True*)

Remove the given object from the poset.

INPUT:

- `key` – the key of an object.
- `raise_key_error` – (default: `True`) switch raising `KeyError` on and off.

OUTPUT:

Nothing.

If the element is not a member and `raise_key_error` is set (default), raise a `KeyError`.

Note: As with Python's `set`, the methods *`remove()`* and *`discard()`* only differ in their behavior when an element is not contained in the poset: *`remove()`* raises a `KeyError` whereas *`discard()`* does not raise any exception.

This default behavior can be overridden with the `raise_key_error` parameter.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: print P.repr_full(reverse=True)
poset((4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2), (2, 1)
+-- (1, 2)
|   +-- successors:   (1, 3), (2, 2)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 2), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
sage: P.remove(T((1, 2)))
sage: print P.repr_full(reverse=True)
poset((4, 4), (1, 3), (2, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 1)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (2, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
```

```
+-- (1, 1)
|   +-- successors:   (1, 3), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

See also:

add(), *clear()*, *discard()*, *pop()*.

TESTS:

```
sage: Q = MP([(1, 1, 42), (1, 3, 42), (2, 1, 7),
....:         (4, 4, 42), (1, 2, 7), (2, 2, 7)],
....:        key=lambda k: T(k[0:2]))
sage: print Q.repr_full(reverse=True)
poset((4, 4, 42), (1, 3, 42), (2, 2, 7),
      (1, 2, 7), (2, 1, 7), (1, 1, 42))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4, 42)
+-- (4, 4, 42)
|   +-- successors:   oo
|   +-- predecessors: (1, 3, 42), (2, 2, 7)
+-- (1, 3, 42)
|   +-- successors:   (4, 4, 42)
|   +-- predecessors: (1, 2, 7)
+-- (2, 2, 7)
|   +-- successors:   (4, 4, 42)
|   +-- predecessors: (1, 2, 7), (2, 1, 7)
+-- (1, 2, 7)
|   +-- successors:   (1, 3, 42), (2, 2, 7)
|   +-- predecessors: (1, 1, 42)
+-- (2, 1, 7)
|   +-- successors:   (2, 2, 7)
|   +-- predecessors: (1, 1, 42)
+-- (1, 1, 42)
|   +-- successors:   (1, 2, 7), (2, 1, 7)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1, 42)
|   +-- no predecessors
sage: Q.remove((1,1))
sage: print Q.repr_full(reverse=True)
poset((4, 4, 42), (1, 3, 42), (2, 2, 7), (1, 2, 7), (2, 1, 7))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4, 42)
+-- (4, 4, 42)
|   +-- successors:   oo
|   +-- predecessors: (1, 3, 42), (2, 2, 7)
+-- (1, 3, 42)
|   +-- successors:   (4, 4, 42)
|   +-- predecessors: (1, 2, 7)
+-- (2, 2, 7)
|   +-- successors:   (4, 4, 42)
|   +-- predecessors: (1, 2, 7), (2, 1, 7)
+-- (1, 2, 7)
```

```
|   +-- successors:   (1, 3, 42), (2, 2, 7)
|   +-- predecessors: null
+-- (2, 1, 7)
|   +-- successors:   (2, 2, 7)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 2, 7), (2, 1, 7)
|   +-- no predecessors
```

```
sage: P = MP()
sage: P.remove(None)
Traceback (most recent call last):
...
ValueError: None is not an allowed key.
```

repr(*include_special=False*, *reverse=False*)

Return a representation of the poset.

INPUT:

- `include_special` – (default: `False`) a boolean indicating whether to include the special elements `'null'` and `'oo'` or not.
- `reverse` – (default: `False`) a boolean. If set, then largest elements are displayed first.

OUTPUT:

A string.

See also:

repr_full()

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: print MP().repr()
poset()
```

repr_full(*reverse=False*)

Return a representation with ordering details of the poset.

INPUT:

- `reverse` – (default: `False`) a boolean. If set, then largest elements are displayed first.

OUTPUT:

A string.

See also:

repr()

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: print MP().repr_full(reverse=True)
poset()
+-- oo
|   +-- no successors
|   +-- predecessors: null
+-- null
```

```
|   +-- successors:   oo
|   +-- no predecessors
```

shell(*key*)

Return the shell of the element corresponding to `key`.

INPUT:

`key` – the key of an object.

OUTPUT:

An instance of *MutablePosetShell*.

Note: Each element is contained/encapsulated in a shell inside the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(42)
sage: e = P.shell(42); e
42
sage: type(e)
<class 'sage.data_structures.mutable_poset.MutablePosetShell'>
```

See also:

element(), *get_key()*.

shells(*include_special=False*)

Return an iterator over all shells.

INPUT:

- `include_special` – (default: `False`) if set, then including shells containing a smallest element ($\emptyset$) and a largest element (∞).

OUTPUT:

An iterator.

Note: Each element is contained/encapsulated in a shell inside the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: tuple(P.shells())
()
sage: tuple(P.shells(include_special=True))
(null, oo)
```

See also:

shells_topological(), *elements()*, *elements_topological()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

shells_topological(*include_special=False*, *reverse=False*, *key=None*)
Return an iterator over all shells in topological order.

INPUT:

- `include_special` – (default: `False`) if set, then including shells containing a smallest element ($\emptyset$) and a largest element (∞).
- `reverse` – (default: `False`) – if set, reverses the order, i.e., `False` gives smallest elements first, `True` gives largest first.
- `key` – (default: `None`) a function used for sorting the direct successors of a shell (used in case of a tie). If this is `None`, then the successors are sorted according to their representation strings.

OUTPUT:

An iterator.

Note: Each element is contained/encapsulated in a shell inside the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: list(P.shells_topological())
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4)]
sage: list(P.shells_topological(reverse=True))
[(4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1)]
sage: list(P.shells_topological(include_special=True))
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4), oo]
sage: list(P.shells_topological(
....:     include_special=True, reverse=True))
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1), null]
```

See also:

shells(), *elements()*, *elements_topological()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

symmetric_difference(*other*)
Return the symmetric difference of two posets as a new poset.

INPUT:

- `other` – a poset.

OUTPUT:

A poset.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.symmetric_difference(Q)
poset(3, 4, 7, 8)
```

See also:

union(), *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

symmetric_difference_update(*other*)

Update this poset with the symmetric difference of itself and another poset.

INPUT:

- `other` – a poset.

OUTPUT:

Nothing.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal; `A.symmetric_difference_update(B)` and `B.symmetric_difference_update(A)` might result in different posets.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.symmetric_difference_update(Q)
sage: P
poset(3, 4, 7, 8)
```

See also:

union(), *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

union(**other*)

Return the union of the given posets as a new poset

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT:

A poset.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal.

Due to keys and a `merge` function (see *MutablePoset*) this operation might not be commutative.

Todo

Use the already existing information in the other poset to speed up this function. (At the moment each element of the other poset is inserted one by one and without using this information.)

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.union(Q)
poset(3, 4, 7, 8, 42)
```

See also:

union_update(), *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

TESTS:

```
sage: P.union(P, Q, Q, P)
poset(3, 4, 7, 8, 42)
```

union_update(**other*)

Update this poset with the union of itself and another poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT:

Nothing.

Note: The key of an element is used for comparison. Thus elements with the same key are considered as equal; `A.union_update(B)` and `B.union_update(A)` might result in different posets.

Todo

Use the already existing information in the other poset to speed up this function. (At the moment each element of the other poset is inserted one by one and without using this information.)

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
```

```
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.union_update(Q)
sage: P
poset(3, 4, 7, 8, 42)
```

See also:

union(), *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

TESTS:

```
sage: Q.update(P)
sage: Q
poset(3, 4, 7, 8, 42)
```

update(**other*)
Alias of *union_update()*.

class sage.data_structures.mutable_poset.**MutablePosetShell**(*poset*, *element*)
Bases: sage.structure.sage_object.SageObject

A shell for an element of a *mutable poset*.

INPUT:

- poset – the poset to which this shell belongs.
- element – the element which should be contained/encapsulated in this shell.

OUTPUT:

A shell for the given element.

Note: If the *element()* of a shell is None, then this element is considered as "special" (see *is_special()*). There are two special elements, namely

- a 'null' (an element smaller than each other element; it has no predecessors) and
- an 'oo' (an element larger than each other element; it has no successors).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(66)
sage: P
poset(66)
sage: s = P.shell(66)
sage: type(s)
<class 'sage.data_structures.mutable_poset.MutablePosetShell'>
```

See also:

MutablePoset

element
The element contained in this shell.

See also:

key(), *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: from sage.data_structures.mutable_poset import MutablePosetShell
sage: e = MutablePosetShell(P, (1, 2))
sage: e.element
(1, 2)
```

eq(*other*)

Return whether this shell is equal to `other`.

INPUT:

- `other` – a shell.

OUTPUT:

`True` or `False`.

Note: This method compares the keys of the elements contained in the (non-special) shells. In particular, elements/shells with the same key are considered as equal.

See also:

le(), *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: from sage.data_structures.mutable_poset import MutablePosetShell
sage: e = MutablePosetShell(P, (1, 2))
sage: f = MutablePosetShell(P, (2, 1))
sage: z = P.null
sage: oo = P.oo
sage: z == z
True
sage: oo == oo
True
sage: e == e
True
sage: e == f
False
sage: z == e
False
sage: e == oo
False
sage: oo == z
False
```

Comparing elements in different mutable posets is possible; their shells are equal if their elements are:

```
sage: S = MP([42]); s = S.shell(42)
sage: T = MP([42]); t = T.shell(42)
sage: s == t
True
sage: S.oo == T.oo
True
```

is_null()

Return whether this shell contains the null-element, i.e., the element smaller than any possible other element.

OUTPUT:

`True` or `False`.

See also:

is_special(), *is_oo()*, *MutablePoset.null()*, *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.null.is_null()
True
sage: P.oo.is_null()
False
```

is_oo()

Return whether this shell contains the infinity-element, i.e., the element larger than any possible other element.

OUTPUT:

`True` or `False`.

See also:

is_null(), *is_special()*, *MutablePoset.oo()*, *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.null.is_oo()
False
sage: P.oo.is_oo()
True
```

is_special()

Return whether this shell contains either the null-element, i.e., the element smaller than any possible other element or the infinity-element, i.e., the element larger than any possible other element.

INPUT:

Nothing.

OUTPUT:

`True` or `False`.

See also:

is_null(), *is_oo()*, *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.null.is_special()
True
sage: P.oo.is_special()
True
```

iter_depth_first(*reverse=False*, *key=None*, *condition=None*)
Iterate over all shells in depth first order.

INPUT:

- `reverse` – (default: `False`) if set, reverses the order, i.e., `False` searches towards `'oo'` and `True` searches towards `'null'`.
- `key` – (default: `None`) a function used for sorting the direct successors of a shell (used in case of a tie). If this is `None`, no sorting occurs.
- `condition` – (default: `None`) a function mapping a shell to `True` (include in iteration) or `False` (do not include). `None` is equivalent to a function returning always `True`. Note that the iteration does not go beyond a not included shell.

OUTPUT:

An iterator.

Note: The depth first search starts at this (`self`) shell. Thus only this shell and shells greater than (in case of `reverse=False`) this shell are visited.

ALGORITHM:

See Wikipedia article Depth-first_search.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: list(P.null.iter_depth_first(reverse=False, key=repr))
[null, (1, 1), (1, 2), (1, 3), (4, 4), oo, (2, 2), (2, 1)]
sage: list(P.oo.iter_depth_first(reverse=True, key=repr))
[oo, (4, 4), (1, 3), (1, 2), (1, 1), null, (2, 2), (2, 1)]
sage: list(P.null.iter_depth_first(
....:     condition=lambda s: s.element[0] == 1))
[null, (1, 1), (1, 2), (1, 3)]
```

See also:

iter_topological(), *MutablePoset*.

iter_topological(*reverse=False*, *key=None*, *condition=None*)
Iterate over all shells in topological order.

INPUT:

- `reverse` – (default: `False`) if set, reverses the order, i.e., `False` searches towards `'oo'` and `True` searches towards `'null'`.
- `key` – (default: `None`) a function used for sorting the direct predeccessors of a shell (used in case of a tie). If this is `None`, no sorting occurs.
- `condition` – (default: `None`) a function mapping a shell to `True` (include in iteration) or `False` (do not include). `None` is equivalent to a function returning always `True`. Note that the iteration does not go beyond a not included shell.

OUTPUT:

An iterator.

Note: The topological search will only find shells smaller than (in case of `reverse=False`) or equal to this (`self`) shell. This is in contrast to *iter_depth_first()*.

ALGORITHM:

Here a simplified version of the algorithm found in *[T1976]* and *[CLRS2001]* is used. See also Wikipedia article Topological_sorting.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
```

```
sage: for e in P.shells_topological(include_special=True,
....:                               reverse=True):
....:     print e
....:     print list(e.iter_topological(reverse=True, key=repr))
oo
[oo]
(4, 4)
[oo, (4, 4)]
(1, 3)
[oo, (4, 4), (1, 3)]
(2, 2)
[oo, (4, 4), (2, 2)]
(1, 2)
[oo, (4, 4), (1, 3), (2, 2), (1, 2)]
(2, 1)
[oo, (4, 4), (2, 2), (2, 1)]
(1, 1)
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1)]
null
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1), null]
```

```
sage: for e in P.shells_topological(include_special=True,
....:                               reverse=True):
....:     print e
....:     print list(e.iter_topological(reverse=False, key=repr))
oo
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4), oo]
(4, 4)
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4)]
(1, 3)
[null, (1, 1), (1, 2), (1, 3)]
(2, 2)
[null, (1, 1), (1, 2), (2, 1), (2, 2)]
(1, 2)
[null, (1, 1), (1, 2)]
(2, 1)
[null, (1, 1), (2, 1)]
(1, 1)
```

```
        [null, (1, 1)]
        null
        [null]
```

```
        sage: list(P.null.iter_topological(
        ....:     reverse=True, condition=lambda s: s.element[0] == 1))
        [(1, 3), (1, 2), (1, 1), null]
```

See also:

iter_depth_first(), *MutablePoset.shells_topological()*, *MutablePoset.elements_topological()*, *MutablePoset.keys_topological()*, *MutablePoset*.

key

The key of the element contained in this shell.

The key of an element is determined by the mutable poset (the parent) via the `key`-function (see construction of a *MutablePoset*).

See also:

element(), *MutablePoset*.

TESTS:

```
        sage: from sage.data_structures.mutable_poset import MutablePoset as MP
        sage: from sage.data_structures.mutable_poset import MutablePosetShell
        sage: P = MP()
        sage: e = MutablePosetShell(P, (1, 2))
        sage: e.key
        (1, 2)
        sage: Q = MP(key=lambda k: k[0])
        sage: f = MutablePosetShell(Q, (1, 2))
        sage: f.key
        1
```

Test the caching of the key:

```
        sage: def k(k):
        ....:     print 'key %s' % (k,)
        ....:     return k
        sage: R = MP(key=k)
        sage: h = MutablePosetShell(R, (1, 2))
        key (1, 2)
        sage: h.key; h.key
        (1, 2)
        (1, 2)
```

le(*other*, *reverse=False*)

Return whether this shell is less than or equal to `other`.

INPUT:

- `other` – a shell.
- `reverse` – (default: `False`) if set, then return whether this shell is greater than or equal to `other`.

OUTPUT:

`True` or `False`.

Note: The comparison of the shells is based on the comparison of the keys of the elements contained in the shells, except for special shells (see *MutablePosetShell*).

See also:

eq(), *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: from sage.data_structures.mutable_poset import MutablePosetShell
sage: e = MutablePosetShell(P, (1, 2))
sage: z = P.null
sage: oo = P.oo
sage: z <= e  # indirect doctest
True
sage: e <= oo  # indirect doctest
True
sage: z <= oo  # indirect doctest
True
sage: oo <= z  # indirect doctest
False
sage: oo <= e  # indirect doctest
False
sage: e <= z  # indirect doctest
False
sage: z <= z  # indirect doctest
True
sage: oo <= oo  # indirect doctest
True
sage: e <= e  # indirect doctest
True
```

```
sage: z.le(e, reverse=True)
False
sage: e.le(oo, reverse=True)
False
sage: z.le(oo, reverse=True)
False
sage: oo.le(z, reverse=True)
True
sage: oo.le(e, reverse=True)
True
sage: e.le(z, reverse=True)
True
sage: z.le(z, reverse=True)
True
sage: oo.le(oo, reverse=True)
True
sage: e.le(e, reverse=True)
True
```

lower_covers (*shell*, *reverse=False*)

Return the lower covers of the specified `shell`; the search is started at this (`self`) shell.

A lower cover of x is an element y of the poset such that $y < x$ and there is no element z of the poset so that $y < z < x$.

INPUT:

- `shell` – the shell for which to find the covering shells. There is no restriction of `shell` being contained in the poset. If `shell` is contained in the poset, then use the more efficient methods *predecessors()* and *successors()*.
- `reverse` – (default: `False`) if set, then find the upper covers (see also *upper_covers()*) instead of the lower covers.

OUTPUT:

A set of *shells*.

Note: Suppose `reverse` is `False`. This method starts at the calling shell (`self`) and searches towards `'oo'`. Thus, only shells which are (not necessarily direct) successors of this shell are considered.

If `reverse` is `True`, then the reverse direction is taken.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: e = P.shell(T((2, 2))); e
(2, 2)
sage: sorted(P.null.lower_covers(e),
....:        key=lambda c: repr(c.element))
[(1, 2), (2, 1)]
sage: set(_) == e.predecessors()
True
sage: sorted(P.oo.upper_covers(e),
....:        key=lambda c: repr(c.element))
[(4, 4)]
sage: set(_) == e.successors()
True
```

```
sage: Q = MP([T((3, 2))])
sage: f = next(Q.shells())
sage: sorted(P.null.lower_covers(f),
....:        key=lambda c: repr(c.element))
[(2, 2)]
sage: sorted(P.oo.upper_covers(f),
....:        key=lambda c: repr(c.element))
[(4, 4)]
```

See also:

upper_covers(), *predecessors()*, *successors()*, *MutablePoset*.

merge(*element*, *check=True*, *delete=True*)

Merge the given element with the element contained in this shell.

INPUT:

- `element` – an element (of the poset).
- `check` – (default: `True`) if set, then the `can_merge`-function of *MutablePoset* determines whether the merge is possible. `can_merge` is `None` means that this check is always passed.

- `delete` – (default: `True`) if set, then `element` is removed from the poset after the merge.

OUTPUT:

Nothing.

Note: This operation depends on the parameters `merge` and `can_merge` of the *MutablePoset* this shell is contained in. These parameters are defined when the poset is constructed.

Note: If the `merge` function returns `None`, then this shell is removed from the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: def add(left, right):
....:     return (left[0], ''.join(sorted(left[1] + right[1])))
sage: def can_add(left, right):
....:     return left[0] <= right[0]
sage: P = MP([(1, 'a'), (3, 'b'), (2, 'c'), (4, 'd')],
....:        key=lambda c: c[0], merge=add, can_merge=can_add)
sage: P
poset((1, 'a'), (2, 'c'), (3, 'b'), (4, 'd'))
sage: P.shell(2).merge((3, 'b'))
sage: P
poset((1, 'a'), (2, 'bc'), (4, 'd'))
```

See also:

MutablePoset.merge(), *MutablePoset*.

TESTS:

```
sage: MP([2], merge=operator.add,
....:    can_merge=lambda _, __: False).shell(2).merge(1)
Traceback (most recent call last):
...
RuntimeError: Cannot merge 2 with 1.
```

poset

The poset to which this shell belongs.

See also:

MutablePoset

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: from sage.data_structures.mutable_poset import MutablePosetShell
sage: e = MutablePosetShell(P, (1, 2))
sage: e.poset is P
True
```

predecessors(*reverse=False*)

Return the predecessors of this shell.

INPUT:

- `reverse` – (default: `False`) if set, then return successors instead.

OUTPUT:

A set.

See also:

successors(), *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: from sage.data_structures.mutable_poset import MutablePosetShell
sage: e = MutablePosetShell(P, (1, 2))
sage: e.predecessors()
set()
```

successors(*reverse=False*)

Return the successors of this shell.

INPUT:

- `reverse` – (default: `False`) if set, then return predecessors instead.

OUTPUT:

A set.

See also:

predecessors(), *MutablePoset*.

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: from sage.data_structures.mutable_poset import MutablePosetShell
sage: e = MutablePosetShell(P, (1, 2))
sage: e.successors()
set()
```

upper_covers(*shell*, *reverse=False*)

Return the upper covers of the specified `shell`; the search is started at this (`self`) shell.

An upper cover of x is an element y of the poset such that $x < y$ and there is no element z of the poset so that $x < z < y$.

INPUT:

- `shell` – the shell for which to find the covering shells. There is no restriction of `shell` being contained in the poset. If `shell` is contained in the poset, then use the more efficient methods *predecessors()* and *successors()*.
- `reverse` – (default: `False`) if set, then find the lower covers (see also *lower_covers()*) instead of the upper covers.

OUTPUT:

A set of *shells*.

Note: Suppose `reverse` is `False`. This method starts at the calling shell (`self`) and searches towards `'null'`. Thus, only shells which are (not necessarily direct) predecessors of this shell are considered.

If `reverse` is `True`, then the reverse direction is taken.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: e = P.shell(T((2, 2))); e
(2, 2)
sage: sorted(P.null.lower_covers(e),
....:        key=lambda c: repr(c.element))
[(1, 2), (2, 1)]
sage: set(_) == e.predecessors()
True
sage: sorted(P.oo.upper_covers(e),
....:        key=lambda c: repr(c.element))
[(4, 4)]
sage: set(_) == e.successors()
True
```

```
sage: Q = MP([T((3, 2))])
sage: f = next(Q.shells())
sage: sorted(P.null.lower_covers(f),
....:        key=lambda c: repr(c.element))
[(2, 2)]
sage: sorted(P.oo.upper_covers(f),
....:        key=lambda c: repr(c.element))
[(4, 4)]
```

See also:

predecessors(), *successors()*, *MutablePoset*.

sage.data_structures.mutable_poset.**is_MutablePoset**(*P*)

Test whether P inherits from *MutablePoset*.

See also:

MutablePoset

TESTS:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: from sage.data_structures.mutable_poset import is_MutablePoset
sage: P = MP()
sage: is_MutablePoset(P)
True
```

CHAPTER

FIVE

INDICES AND TABLES

- Index
- Module Index
- Search Page

[T1976] Robert E. Tarjan, *Edge-disjoint spanning trees and depth-first search*, Acta Informatica 6 (2), 1976, 171-185, doi:10.1007/BF00268499.

[CLRS2001] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Section 22.4: Topological sort*, Introduction to Algorithms (2nd ed.), MIT Press and McGraw-Hill, 2001, 549-552, ISBN 0-262-03293-7.

d

m

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V