Sage Reference Manual: Data Structures

Release 6.8

The Sage Development Team

July 29, 2015

CONTENTS

CHAPTER

ONE

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
100000000000000000000000000000000000000000000000000000000000
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
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
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
00000101011010110101101011010110101101011010110101101011010110101101011010110101101011010110
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
01000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
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
110001010101010101010101010101010101010101010101010101010101
sage: a_set.symmetric_difference_update(FrozenBitset('010101' * 10))
sage: a_set == set(a)
True
sage: a.symmetric_difference_update(FrozenBitset('01010' * 20)); a
10010111110000011111000001111100000111110000011111010100101001010010100101001010010100101001
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
01010101011010110101101011010110101101011010110101101011010110101101011010110101101011010110
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
11011000110001100011000110001100011000110001100011000110001100011000110001100011000110001100
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 $0, 1, \dots, n-1$ (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 $\{0, 1, 2, \dots, k\}$ 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))
001101010001010001010001000001
001101010001010001010001000001
```

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))
101010101010101010101010101010101010101010000000000
```

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))
010101010101010101010101010101010101010100000000000
```

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))
10101010101010101010101010101010101010101010100101010101
```

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 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
list a []
a.size 128
len(a) 0
a.limbs ...
b 010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
a.in(n)   False
a.not_in(n)   True
a.add(n)     0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
a.discard(n)   00000000000000000000000000000000000000000000000000000000000000000000000000000000000
a.set_to(n)  0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
a.flip(n)    0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
a.set_first_n(n)    111111111111111111111111111111111111111111111111111111111111111111111111111111
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()     0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
r.clear()     000000000000000000000000000000000000000000000000000000000000000000000000000000000000
complement a        111111111111111111111111111111111111111111111111111111111111111111111111111111
a intersect b      0000000000000000000000000000000000000000000000000000000000000000000000000000000
a union b       0101010101010101010101010101010101010101010101010101010101010101010101010101010101
a minus b      00000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
a symmetric_difference b      010101010101010101010101010101010101010101010101010101010101010101
a.rshift(n)  000000000000000000000000000000000000000000000000000000000000000000000000000000000000
a.lshift(n)  000000000000000000000000000000000000000000000000000000000000000000000000000000000000
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)  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
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      00000000000000000000000000000000000000000000000000000000000000000000000000000
to size 127          0000000000000000000000000000000000000000000000000000000000000000000000000000
to size 254          0000000000000000000000000000000000000000000000000000000000000000000000000000
to original size    00000000000000000000000000000000000000000000000000000000000000000000000000000
```

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 11100111100111100111100111100111100111100111100111100111100111100111100111100111100111100111100
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 0001001000011111101101010100010001000010110100011000010001101001100010011000110011000101000101
a.in(n)   False
a.not_in(n)   True
a.add(n)     11100111100111100111100111100111100111100111100111100111100111100111101111001111100
a.discard(n)   111001111001111001111001111001111001111001111001111001111001111001111001111001111
a.set_to(n)  11100111100111100111100111100111100111100111100111100111100111100111101111001111100
a.flip(n)    11100111100111100111100111100111100111100111100111100111100111100111101111001111100
a.set_first_n(n)    11111111111111111111111111111111111111111111111111111111111111111111100000000
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()     11100111100111100111100111100111100111100111100111100111100111100111100111100111100
r.clear()     00000000000000000000000000000000000000000000000000000000000000000000000000000000000
complement a        0001100001100001100001100001100001100001100001100001100001100001100001100001
a intersect b      000000100001111000110001010001000000000100101000110000100000010000000100110000
a union b      1111011110011111111111011110011111011110111110011110011111111111111111100111101111
a minus b     11100101100000000100100010100011100111000010100001100011100101100111000001100011
a symmetric_difference b      1111010110000001110011001010001111011100101010000110001111110111111
a.rshift(n)  00111100111100111100111100111100111100111100111100111100111100111100111100111100100
a.lshift(n)  00000000000000000000000000000000000000000000000000000000000000000000011100111100111
```

```
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)  10011110011110011110011110011110011110011110011110011110011110011110011110011110011110
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      1110011110011110011110011110011110011110011110011110011110011110011110011110
to size 69          111001111001111001111001111001111001111001111001111001111001111001111
to size 138          111001111001111001111001111001111001111001111001111001111001111001111000000
to original size    11100111100111100111100111100111100111100111100111100111100111100111100000000
```

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

INDICES AND TABLES

- Index
- Module Index
- Search Page

d

m

A

B

C

D

F

G

I

K

L

M

N

P

R

S

T

U

V