6.0 - Week 6 - Workshop (MA)

Learning Objectives

- Understanding ADTs and containers.
- Understanding inheritance.
- Reinforcing your understanding of unit tests, exceptions, complexity, and bitwise operations.

Week 6 Padlet Discussion Board link: https://monashmalaysia.padlet.org/fermi/2022week6

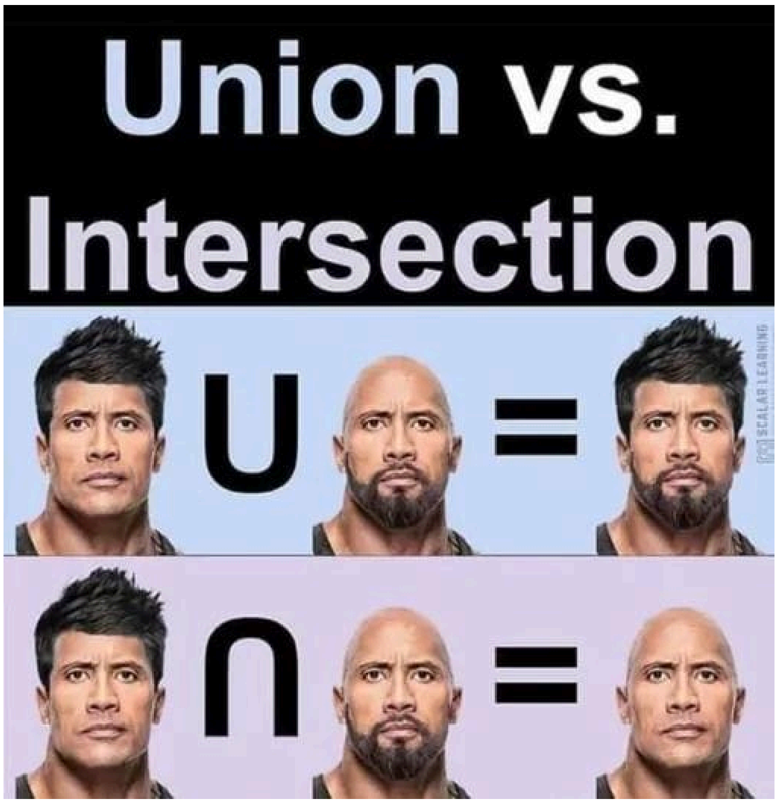
What is an ADT?

Question *Submitted Aug 29th 2022 at 10:05:17 am*

Do you know what ADT stands for?

- ○ No idea. Do you really expect me to have watched the Moodle lesson?
- ○ Some weird concept in computer science, who really cares?..
- ○ Sure, Google says it is *Australian Dance Theatre*!
- ● It is *Abstract Data Type*! In a sense, an ADT can be seen as a mathematical model of a data type defining its *behaviour (semantics)* but *ignoring* the implementation details.

Set ADT



In mathematics, a set is an ***unordered collection of distinct elements***. One can use the following operations on sets:

- **add** an element:

```
s = set([])  # create an empty set from an empty list
s.add(1)
print(s)
```

- **remove** an element:

```
s = {1, 2}  # another notation to create a set object
s.remove(1)
print(s)
```

- **union, intersection or difference** with another set:

Mathematically:
Union $S_3 = S_1 \cup S_2$ contains elements that are **either** in S_1 or S_2 .

Intersection $S_3 = S_1 \cap S_2$ contains elements that are **both** in S_1 and S_2.
Difference $S_3 = S_1 \setminus S_2$ contains elements that are **in** S_1 **but not in** S_2.

```
s1, s2 = {1, 2, 3}, {1, 4}
print("Union:", s1.union(s2))
print("Intersection:", s1.intersection(s2))
print("Difference:", s1.difference(s2))
```

- there are **more operations** on sets!

Set Abstract Class

Here is the abstract class `Set`, which we will use next to develop two implementations of the Set ADT:

```
"""
    Set ADT. Defines a generic abstract set with the usual methods.
"""

from __future__ import annotations

__author__ = "Alexey Ignatiev"
__docformat__ = 'reStructuredText'

from abc import ABC, abstractmethod
from typing import TypeVar, Generic
T = TypeVar('T')

class Set(ABC, Generic[T]):
    """ Abstract class for a generic Set. """

    def __init__(self) -> None:
        """ Initialization. """
        self.clear()

    @abstractmethod
    def __len__(self) -> int:
        """ Returns the number of elements in the set. """
        pass

    @abstractmethod
    def is_empty(self) -> bool:
        """ True if the set is empty. """
        pass

    @abstractmethod
    def clear(self) -> None:
        """ Makes the set empty. """
        pass

    @abstractmethod
    def __contains__(self, item: T) -> bool:
        """ True if the set contains the item. """
        pass

    @abstractmethod
    def add(self, item: T) -> None:
        """ Adds an element to the set. Note that the element already
        present in the set should not be added.
        """
        pass

    @abstractmethod
    def remove(self, item: T) -> None:
        """ Removes an element from the set. An exception should be
        raised if the element to remove is not present in the set.
        """
        pass

    @abstractmethod
    def union(self, other: Set[T]) -> Set[T]:
        """ Makes a union of the set with another set. """
        pass

    @abstractmethod
    def intersection(self, other: Set[T]) -> Set[T]:
        """ Makes an intersection of the set with another set. """
        pass

    @abstractmethod
    def difference(self, other: Set[T]) -> Set[T]:
        """ Creates a difference of the set with another set. """
        pass
```

An array-based implementation of Set

The goal of this activity is to:

Given the class `ASet`, which is an implementation of the class `Set` that uses an (unordered) array to store the elements of the set, implement two missing methods:

- `self.add(item)`
- `self.intersection(other)`

Note that both the `Set` abstract class and the `ArrayR` class are provided in the scaffold.

Complexity of ASet operations

Denote n the size of `self` and m the size of `other`.

Suppose that the complexity of `add`, `remove` and `__contains__` is $O(k)$ if k is the size of the `ASet`.

Question *Submitted Aug 29th 2022 at 11:23:50 am*

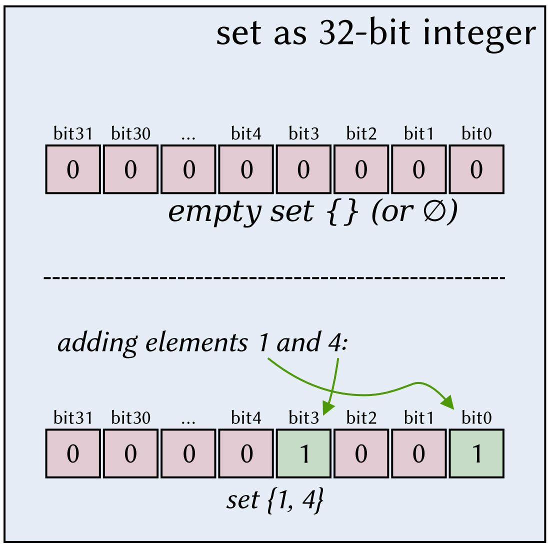
What is the worst-case time complexity of this function?

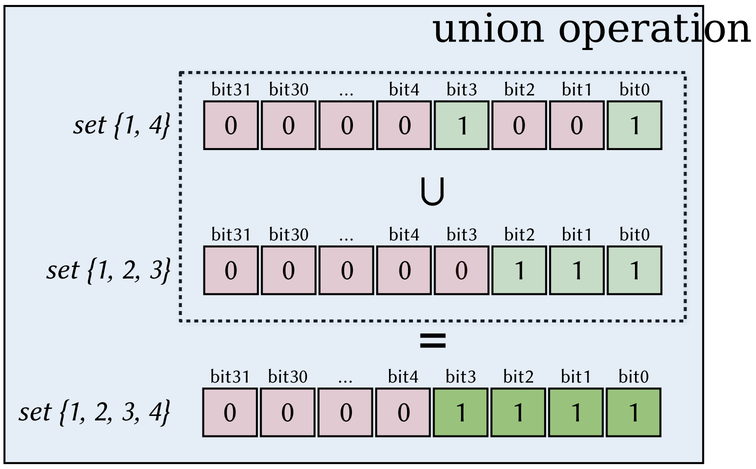
```
def union(self, other: ASet[T]) -> ASet[T]:
    """ Creates a new set equal to the union with another one. """
    maxcapacity = len(self.array) + len(other.array)
    res = ASet(maxcapacity)
    for the_set in [self, other]:
        for i in range(len(the_set)):
            res.add(the_set.array[i])
    return res
```

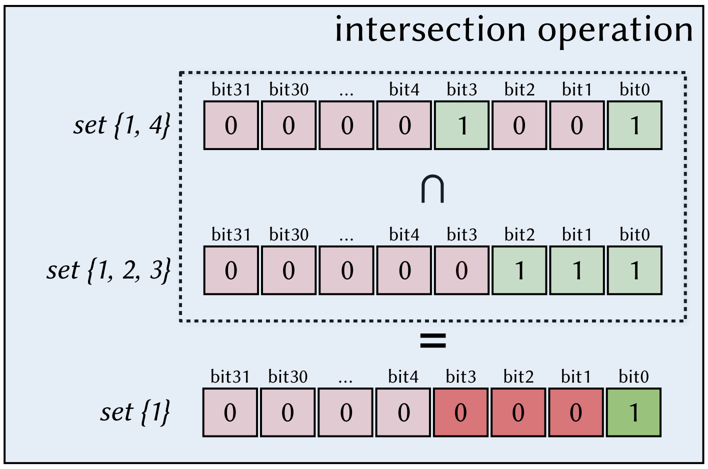
- $\mathcal{O}(1)$
- $\mathcal{O}(n)$
- $\mathcal{O}(m)$
- $\mathcal{O}(n+m)$
- $\mathcal{O}(nm)$
- [✓ None of the above]

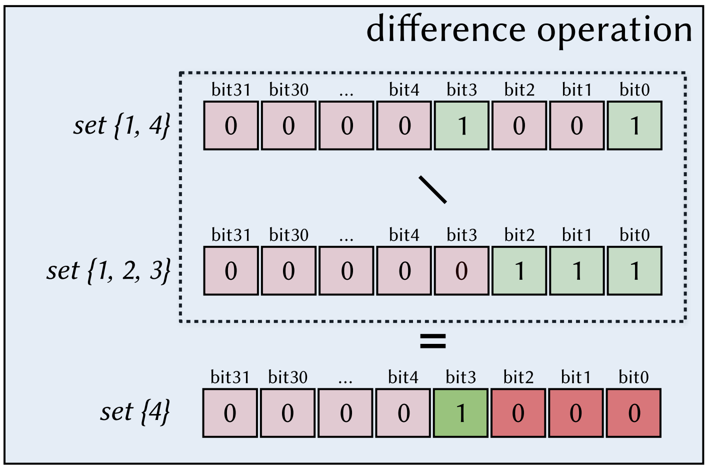
Set as Integer (Bit Vector, or Bit Array)

A natural representation of a *set of integers* is a bit vector (or a *bit array*, or *bit string*, or simply an *integer*).









A bitvector-based implementation of Set

The goal of this activity is to:

Given the class `BSet`, which is an implementation the set ADT using an integer number (bit-vector) to store the elements of the set, implement three missing methods:

- `self.is_empty()`
- `self.intersection(other)`
- `self.difference(other)`

Note that the `Set` abstract class is provided in the scaffold.

Complexity of BSet Operations

Question 1 *Submitted Aug 29th 2022 at 11:24:40 am*

Let us denote the number of elements in `self` and `other` by `n` and `m`, respectively. Also, assume that bitwise operations are constant-time, what is the worst-case complexity of the `union()` operation for `BSet`?

```
def union(self, other: BSet[int]) -> BSet[int]:
    """ Creates a new set equal to the union with another one,
    i.e. the result set should contains the elements of self and other.
    """
    res = BSet()
    res.elems = self.elems | other.elems
    return res
```

- [✓ $\mathcal{O}(1)$]
- $\mathcal{O}(n)$
- $\mathcal{O}(m)$
- $\mathcal{O}(n + m)$
- $\mathcal{O}(n \times m)$
- None of the above

Question 2 *Submitted Aug 29th 2022 at 11:24:48 am*

Real-world computers perform constant-time operations with "machine words" of a constant size, e.g. with 8-, 16-, 32-, and 64-bit `int` types. This also holds for bitwise operations that are constant-time only for constant-size integers.

Python, however, is a high-level programming language, which can work with integers of an arbitrary length by representing them as chunks of machine words. For instance, assuming that the size of each chunk is 32 bits, an integer `2 ** 128 - 1` could be represented using 4 chunks. Does this knowledge change our complexity analysis for the `union()` operation on `BSet`?

○ No, it does not.

[✓ Yes, the complexity becomes $\mathcal{O}(n)$.]

○ Yes, the complexity becomes $\mathcal{O}(n^2)$.

○ None of the above

Question 3 *Submitted Aug 29th 2022 at 11:25:04 am*

Denote the number of set elements by `n`. Note that `int.bit_length(K)` returns the number of bits `n` in the binary representation for integer `K`.

Also, assume that bitwise operations are constant-time, what is the worst-case complexity of the `__len__()` operation for `BSet`? Assume here that the complexity of `__contains__()` is $\mathcal{O}(1)$.

```
def __len__(self) -> int:
    """ Size computation. """
    res = 0
    for item in range(1, int.bit_length(self.elems) + 1):
        if item in self:
            res += 1
    return res
```

○ $\mathcal{O}(1)$

[✓ $\mathcal{O}(n)$]

○ $\mathcal{O}(n^2)$

○ None of the above

Reference on Bitwise Operations

Bitwise operations are performed on binary (two's complement) representation of integers. Given integers `x`, `y`, and `z`, the operations are as follows:

- `z = x << y` - **logical left shift** - `z` equals to `x` with the bits shifted to the left by `y` places

```
def bits(x, n):
    """ Return binary representation of x with at least n bits. """
    s = bin(x & int('1' * n, 2))[2:]
    return ('0b{0:0>%s}' % (n)).format(s)

x = int(input('Enter x: '))
y = int(input('Enter y: '))

# left shift
print('{0} = {1} << {2}'.format(bits(x << y, int.bit_length(x) + y), bits(x, int.bit_length(x)), y)
```

- `z = x >> y` - **logical right shift** - `z` equals to `x` with the bits shifted to the right by `y` places

```
def bits(x, n):
    """ Return binary representation of x with at least n bits. """
    s = bin(x & int('1' * n, 2))[2:]
    return ('0b{0:0>%s}' % (n)).format(s)

x = int(input('Enter x: '))
y = int(input('Enter y: '))

# right shift
print('{0} = {1} >> {2}'.format(bits(x >> y, int.bit_length(x >> y)), bits(x, int.bit_length(x)), y
```

- `z = x & y` - **bitwise "and"** - each bit of `z`'s is 1 if the corresponding bit of `x` AND of `y` is 1

```
def bits(x, n):
    """ Return binary representation of x with at least n bits. """
    s = bin(x & int('1' * n, 2))[2:]
    return ('0b{0:0>%s}' % (n)).format(s)

x = int(input('Enter x: '))
y = int(input('Enter y: '))
l = max(int.bit_length(x), int.bit_length(y))  # maximal length
```

```
# bitwise "and"
print('{0} = {1} & {2}'.format(bits(x & y, l), bits(x, l), bits(y, l)))
```

- `z = x | y` - **bitwise "or"** - each bit of `z`'s is 1 if the corresponding bit of either `x` OR of `y` is 1

```
def bits(x, n):
    """ Return binary representation of x with at least n bits. """
    s = bin(x & int('1' * n, 2))[2:]
    return ('0b{0:0>%s}' % (n)).format(s)

x = int(input('Enter x: '))
y = int(input('Enter y: '))
l = max(int.bit_length(x), int.bit_length(y))  # maximal length

# bitwise or
print('{0} = {1} | {2}'.format(bits(x | y, l), bits(x, l), bits(y, l)))
```

- `z = ~x` - **complement** - the bits of `z` are the "flipped" bits of `x`

```
def bits(x, n):
    """ Return binary representation of x with at least n bits. """
    s = bin(x & int('1' * n, 2))[2:]
    return ('0b{0:0>%s}' % (n)).format(s)

x = int(input('Enter x: '))

# complement
print('{0} = ~{1}'.format(bits(~x, int.bit_length(x)), bits(x, int.bit_length(x))))
```

- `z = x ^ y` - **bitwise "xor"** - each bit of `z`'s is 1 if the corresponding bit of either `x` OR of `y` is 1 (**not both!**)

```
def bits(x, n):
    """ Return binary representation of x with at least n bits. """
    s = bin(x & int('1' * n, 2))[2:]
    return ('0b{0:0>%s}' % (n)).format(s)

x = int(input('Enter x: '))
y = int(input('Enter y: '))
l = max(int.bit_length(x), int.bit_length(y))  # maximal length

# bitwise xor
print('{0} = {1} ^ {2}'.format(bits(x ^ y, l), bits(x, l), bits(y, l)))
```

By the way, how do we set the i's bit of a number?

```
x = 0
y = int(input('Which bit? '))

x |= 1 << y  # here we apply two bitwise operations ("lshift" + "or")
print(bin(x))
```

Feedback Form

Weekly Workshop Feedback Form

Question 1 *Submitted Sep 4th 2022 at 9:49:07 pm*

I am enrolled in:

- ▯▯ Australia
- [✓ ▯▯ Malaysia]

Question 2

What needs improvement?

No response

Question 3

What worked best?

No response

Question 4

How engaged were you by the workshop?

- ▯▯▯ Very engaged
- ▯▯▯ Engaged
- ▯😐▯ Not impressed
- 😕😴▯ Lost