DS - Seminar 1 - ADT Bag

- **Content**:
  - ADT Bag
  - ADT Iterator
  - Examples of representations
  - Python implementation

ADT Bag

Definition

When compared to a set:

- Unlike the case of a Set, the elements of a bag do not have to be unique
- Just as in the case of a Set, the order of the elements is not important

So, a Bag is a container in which the elements can repeat and their order is not relevant.

It is similar to a shopping cart/bag: We can buy multiple pieces of the same product, while the order in which we buy the items is unimportant. Since the order of the elements in a Bag is irrelevant:

- There are no positions in a Bag. Therefore, there are no operations taking positions as parameters or returning positions.
- The added elements are not necessarily stored in the order in which they were added (at least, there is no such guarantee; they can be stored in that order, but it is not necessarily to be store in that order).

  For example, if we add to an initially empty Bag the following elements in the following order: 1, 3, 2, 6, 2, 5, 2, if we print the content of the Bag, the elements can be printed in any order:
  - 1, 2, 2, 2, 3, 6, 5
  - 1, 3, 2, 6, 2, 5, 2
  - 1, 5, 6, 2, 3, 2, 2

    Etc

Specifications

1) Domain definition:

$\mathbf{B}$ = {b | b is a Bag with elements of the type TElem}

2) Interface (set of operations) specification:

init(b)

pre : true

post: b $\in \mathbf{B}$, b is an empty Bag

destroy(b)

pre: b $\in \mathbf{B}$

post: b has been destroyed

add(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ TElem

post: b' ∈ $\mathcal{B}$, b' = b $\oplus$ e (TElem *e* is added to the bag *b*)

remove(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ TElem

post: b' ∈ $\mathcal{B}$, b' = b - e (one occurrence of e has been removed from the Bag)

$$\text{remove} = \begin{cases} \text{true, if the element } e \text{ has been removed } (\text{size}(\text{b}') = \text{size}(b) - 1) \\ \text{false, if } e \text{ does not appear in b } (\text{size}(\text{b}') = \text{size}(\text{b})) \end{cases}$$

search(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ TElem

post: $search = \begin{cases} true, if\ e \in b \\ false, otherwise \end{cases}$

size(b)

pre: b ∈ $\mathcal{B}$

post: size $= the\ number\ of\ elements\ from\ b$

nrOccurrences(b, e)

pre: b ∈ $\mathcal{B}$, e ∈ TElem

post: $nrOccurrences = the\ number\ of\ occurrences\ of\ e\ in\ b$

iterator(b, i)

pre: b ∈ $\mathcal{B}$

post: i ∈ $\mathcal{I}$, i is an iterator over b

ADT Iterator

An iterator is a data type that is used to iterate through the elements of a container. Iterators are used to offer a common and generic way of traversing a container, independently of the representation of that container.

The iterator is a so-called light-weight data type, since it does not consume much memory.

In its representation, an iterator usually contains a reference to the container it iterates over and a so-called cursor, which is a reference to the current element in the container.

Iterating through the elements of the container actually means moving the cursor from one element to the next until the iterator becomes invalid, which means there are no more element to be iterated.

The exact way of representing the cursor of the iterator depends on the data structure used for representing the container.

As a consequence, if the representation of the container changes, we need to change the representation (implementation) of the iterator as well.

The Iterator is an ADT, but, unlike the Bag, it is not a container.

Specification

1) Domain definition:

$\mathcal{I}$ = {i | i is an iterator over b ∈ $\mathcal{B}$ }

2) Interface specification:

init(b, i)

pre: b ∈ $\boldsymbol{B}$

post: i ∈ I, i is an iterator over b, i refers to the first element of b or it is invalid if b is empty

getCurrent(i)

pre: i ∈ I, valid(i)

post: getCurrent = e ∈ TElem, e is the current element referred by i

Throws an exception if i is not valid

next(i)

pre: i ∈ I, valid(i)

post: i' ∈ I , the cursor of i' refers to the next element from the iterated bag

Throws an exception if i is not valid

valid(i)

pre: i ∈ I

post: $valid =$ $\begin{cases} true, \text{ if } i \text{ refers a valid element of the iterated bag} \\ false, otherwise \end{cases}$

first(i)

pre: i ∈ I

post: i' ∈ I , the cursor of i' refers to the first element from the iterated bag or i' is invalid if the bag is empty

The sub-algorithm for printing all the elements of a bag:

```
Subalgorithm printAllElements(b) is:
        iterator(b, it)
          While valid(it) execute:
                elem ← getCurrent(it)
                write(elem)
                next(it)
        end-while
End-subalgorithm
```

Examples of representation:

1. A first representation (**R1**)

   - An array of elements (*elems*), where every element can appear multiple times.
   - Besides the reference to the iterated bag (*bag*), the iterator will contain a current position (index) from the dynamic array (*current*)

1	3	2	6	2	5	2

Bag:

elems: TElem[1…n] - the array of elements
n: Integer - the number of elements in the array / the length of the array

IteratorBag:

bag: Bag - the reference to the iterated bag
current: Integer - the index of the current element in the iteration

2. A second representation (**R2**)

- A dynamic array of unique elements, and for each element its frequency (either two dynamic arrays, or one single dynamic array of pairs (element, frequency).
- In this case, it is not sufficient to have just the current index, but it is also necessary to have the current frequency for the element at the current index. So, the iterator will have in its representation a current position and a current frequency.

(1,1)	(3,1)	(2,3)	(5,1)	(6,1)

Pair:

e: TElem - the element
f: Integer - the frequency (number of occurrences) of the element

Bag:

pairs: Pair[1…n] - the array of pairs
n: Integer - the number of pairs in the array / the length of the array

IteratorBag:

b: Bag - the reference to the iterated bag
current: Integer - the index of the current element in the iteration
f: Integer - the index of the current element in the iteration

Please note that the algorithm *printAllElements* for printing all the elements of a bag is the same irrespective to the representation chosen for the Bag (R1 or R2 or any other representation).
This is due to the abstraction principle and is one of the advantages of working with Abstract Data Types.

Python implementation

- **Note:** Python lists are actually Dynamic Arrays

- Python implementation for the first representation (**R1**)

```
class Bag:
    def __init__(self):
        self.__elems = []
    def add(self, e):
        self.__elems.append(e)
    def search(self, e):
        return e in self.__elems
    def remove(self, e):
        return self.__elems.remove(e)
    def size(self):
        return len(self.__elems)
    def iterator(self):
        return Iterator(self)
    def nrOccurrences(self, e):
        n = 0
        for elem in self.__elems:
            if elem == e:
                n = n + 1
        return n

class Iterator:
    def __init__(self,c):
        self.__bag = c
        self.__current = 0
    def getCurrent(self):
        return self.__bag._Bag__elems[self.__current]
    def next(self):
        self.__current = self.__current+1
    def valid(self):
        return self.__current < self.__bag.size()
    def first(self):
        self.__current = 0

def populateIntBag(c):
    c.add(1)
    c.add(2)
    c.add(3)
    c.add(2)
    c.remove(2)
    c.add(2)

def printBag(c):
    it = c.iterator()
    while it.valid():
        print(it.getCurrent())
        it.next()
    print("Over. Let's print the bag content again.")
    it.first()
    while it.valid():
        print(it.getCurrent())
        it.next()
```

```
def main():
    b = Bag()
    populateIntBag(b)
    print("Number of occurrences for 2: ", b.nrOccurrences(2))
    printBag(b)

main()
```

- Python implementation for the second representation (**R2**)

```
class PairElementFrequency:
    def __init__(self,e,f):
        self.__e = e
        self.__f = f
    def setElement(self, e):
        self.__e = e
    def setFrequency(self, f):
        self.__f = f
    def getElement(self):
        return self.__e
    def getFrequency(self):
        return self.__f
    def __eq__(self, other):
        return (self.__e == other.__e)

class BagElementFrequency:

    def __init__(self):
        self.__pairs = []

    def add(self, e):
    # We distinguish two cases: the element already appears in the bag or it
    # does not appear in the bag yet
    # If the element already appears in the bag, we have to increment its
    # frequency
    # If the element does not appear in the bag yet, then we add it to the
    # bag with frequency 1
        found = False
        for p in self.__pairs:
            if p.getElement()== e:
                p.setFrequency(p.getFrequency()+1)
                found = True
        if not(found):
            p = PairElementFrequency(e,1)
            self.__pairs.append(p)

    def search(self, e):
         for p in self.__pairs:
            if p.getElement() == e:
                return True
         return False

    def remove(self, e):
    # We distinguish three cases: the element does not appear in the bag, the
    # element appears only once in the bag or the element appears in the bag
    # more than once
    # If the element does not appear in the bag, than the bag remains
    # unmodified
    # If the element appears in the bag only once, then it will be deleted
    # If the element appears in the bag more than once, then its frequency
    # will be decremented
```

```
        for p in self.__pairs:
            if p.getElement() == e:
                if p.getFrequency() > 1:
                    p.setFrequency(p.getFrequency()-1)
                else:
                    self.__pairs.remove(p)
            return True
        return False


    def size(self):
        s = 0
        for p in self.__pairs:
            s = s + p.getFrequency()
        return s

    def iterator(self):
        return IteratorBagFrequency(self)

    def nrOccurrences(self, e):
        for p in self.__pairs:
            if p.getElement() == e:
                return p.getFrequency()
        return 0


class IteratorBagFrequency:
    def __init__(self, b):
        self.__bag = b
        self.__current = 0
        self.__f = 1

    def valid(self):
        return self.__current < len(self.__bag._BagElementFrequency__pairs)

    def getCurrent(self):
        return
self.__bag._BagElementFrequency__pairs[self.__current].getElement()

    def next(self):
    # We distinguish two cases: the current frequency for the current element is
    #(1) strictly less than the frequency of the current element in and bag or
    #(2) equal to the frequency of the current element in the bag
        # If the current frequency for the current element is strictly less that
        # the frequency of the element in the bag, then we just increment the
        # current frequency
        # If the current frequency of the current element is equal to the
        # frequency of the element in the bag, then we have to pass to the next
        # element by incrementing the current index and reinitializing the
        # current frequency with 1
        if self.__f <
self.__bag._BagElementFrequency__pairs[self.__current].getFrequency():
            self.__f = self.__f+1
        else:
            self.__current = self.__current+1
            self.__f=1

    def first(self):
        self.__current = 0
        self.__f = 1

def populateIntBag(c):
```

```
    c.add(1)
    c.add(2)
    c.add(3)
    c.add(2)
    c.remove(2)
    c.add(2)

def printBag(c):
    it = c.iterator()
    while it.valid():
        print(it.getCurrent())
        it.next()
    print("Over. Let's print the bag content again.")
    it.first()
    while it.valid():
        print(it.getCurrent())
        it.next()

def main():
    b = BagElementFrequency()
    populateIntBag(b)
    print("Number of occurrences for 2: ", b.nrOccurrences(2))
    printBag(b)

main()
```