

Objektumelvű programozás

Gregorics Tibor

gt@inf.elte.hu

<http://people.inf.elte.hu/gt/oep>

Procedurális vs. objektumelvű programozás

- ❑ **Procedurális program:** a feladatot megoldó résztevékenységeket önálló egységekbe, procedurákba (részprogram, makró, eljárás, függvény) szervezzük. A megoldás folyamatát ezen procedurák közötti vezérlés-átadások (eljárások, függvények esetében hívások) jelöli ki.
- ❑ **Objektumelvű program :** a feladat megoldáshoz szükséges adatoknak egy-egy részét az azokkal kapcsolatos tevékenységekkel (az ún. metódusokkal) együtt zárjuk egységekbe, az objektumokba. A megoldás folyamatát ezen objektumok metódusai közötti vezérlés-átadások (hívások vagy üzenet-váltások) jelöli ki.

Feladat

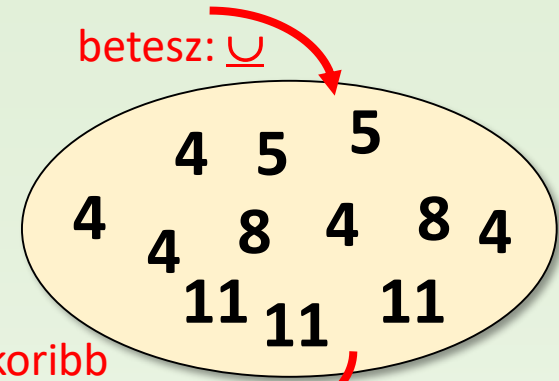
Egy számsorozatban **0 és m közé eső természetes számok** találhatóak. Adjuk meg, hogy melyik szám fordul elő a sorozatban legtöbbször!

□ Ezt a feladatot többféleképpen is meg lehet oldani:

- **Procedurális megoldás:** maximum kiválasztásba ágyazott számlálás,
 - amikor a $0..m$ intervallum elemeinek a megadott számsorozatban való előfordulásainak darabszámai között kell a legnagyobbat megtalálni,
 - amikor a megadott számsorozat elemeinek a számsorozatban való előfordulásainak darabszámai között kell a legnagyobbat megtalálni.
- **Objektumelvű megoldás:** olyan tároló objektumban (gyűjteményben) helyezzük el a számsorozat elemeit, amelynél egy elem behelyezése és a leggyakoribb elem lekérdezése gyors.
 - $0..m$ közé eső számokat tároló zsák (multiplicitásos halmaz)
 - tetszőleges természetes számokat tároló zsák

Elemzés és Tervezés

b:Zsák



leggyakoribb
elem kiválasztása

a feladat **változói**, amelyek egyben a megoldó program változói is lesznek.

$A : m:\mathbb{N}, x:\mathbb{N}^*, \text{elem}:\mathbb{N}$

m kezdőértéke: m_0
 x kezdőértéke: x_0

x nem üres,
 x elemei 0 és m közé esnek

$Ef : m = m_0 \wedge x = x_0 \wedge |x| \geq 1 \wedge \forall i \in [1 .. |x|] : x[i] \in [0 .. m]$

változók kezdőértékét jellemzi

$Uf : m = m_0 \wedge x = x_0 \wedge b:\text{Zsák} \wedge b = \bigcup_{i=1}^{|x|} \{x[i]\} \wedge \text{elem} = \text{leggyakoribb}(b)$

változók végértékét jellemzi

végrehajtható specifikáció

a bemenő változók megőrzik kezdőértékeiket

Összegzés:

\sim bezsákolás

$s = \sum_{i=m..n} f(i) \sim b = \bigcup_{i=1..|x|} \{x[i]\}$

felsorolt indexek: $i \in [m .. n] \sim i \in [1 .. |x|]$

felsorolt értékek: $f(i) \sim \{x[i]\}$

eredmény: $s \sim b$

összegzés művelete: $H, +, 0 \sim \text{Zsák}, \cup, \emptyset$

$b := \emptyset$

$i = 1 .. |x|$

$b := b \cup \{x[i]\}$

$\text{elem} := \text{leggyakoribb}(b)$

Tesztelési stratégiák

❑ **Fekete doboz:** a feladat (specifikációja) alapján felírt tesztesetek.

- az előfeltételt megszegő ún. érvénytelen tesztesetek
- az utófeltétel eseteinek vizsgálata
- ...

❑ **Fehér doboz:** a kód alapján felírt tesztesetek.

- minden utasítás lefedése
- minden elágazási csomópont lefedése
- ...

❑ **Szürke doboz:** végrehajtható specifikáció által előrevetített algoritmusok működését ellenőrző tesztesetek.

- Ha a specifikáció programozási tételekre utal, akkor a programozási tételekre általában jellemző teszteseteket érdemes megvizsgálni.

Tesztelés

Összegzés tesztesetei		tesztadatok (x sorozat elemeinek bezsákolása után melyik a zsák leggyakoribb eleme)		
intervallum szerint	hossza: 0	$m = 0$	$x = \langle \rangle$	→ érvénytelen
	hossza: 1	$m = 2$	$x = \langle 2 \rangle$	→ $e=2$
	hossza: több	$m = 5$	$x = \langle 3, 5, 5, 1 \rangle$	→ $e=5$
	eleje	$m = 2$	$x = \langle 1, 2 \rangle$	→ $e=1$
		$m = 2$	$x = \langle 1, 1, 2 \rangle$	→ $e=1$
	vége	$m = 2$	$x = \langle 1, 2, 2 \rangle$	→ $e=2$
		$m = 2$	$x = \langle 1, 2 \rangle$	→ $e=2$
tétel szerint	terhelés	$m = 2$	$x = \langle 2, 2, \dots, 2 \rangle$	→ $e=2$

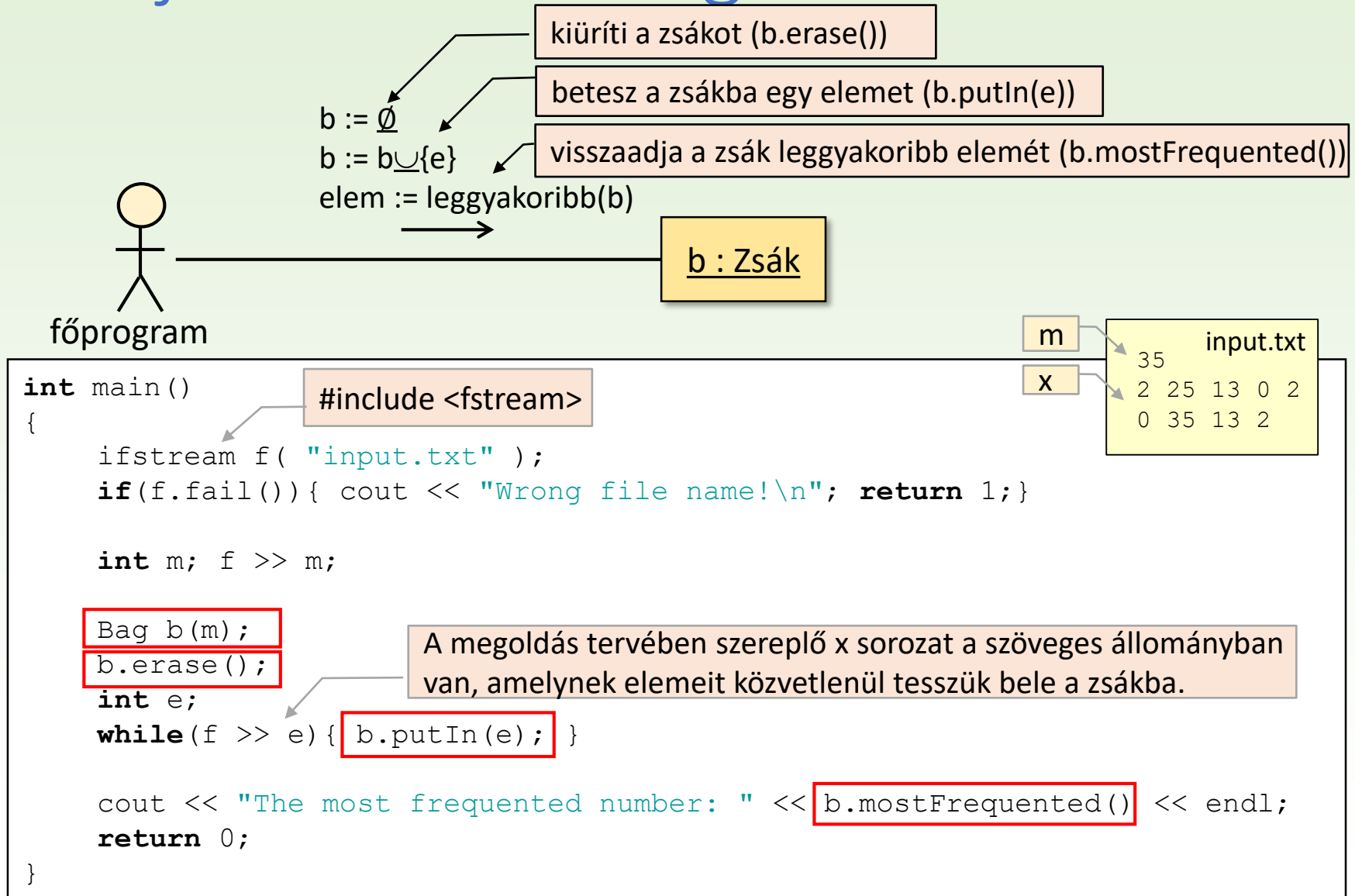
ha $x = \langle 1, 2 \rangle \rightarrow e=1$

ha $x = \langle 1, 2 \rangle \rightarrow e=2$

ha $x = \langle 1, 2 \rangle \rightarrow e=1$

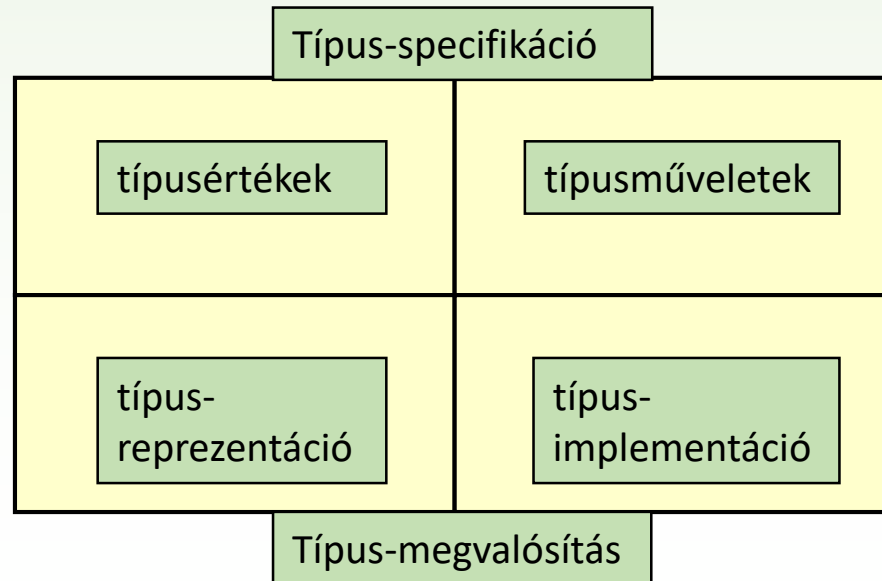
ha $x = \langle 1, 2 \rangle \rightarrow e=2$

Objektumelvű megvalósítás



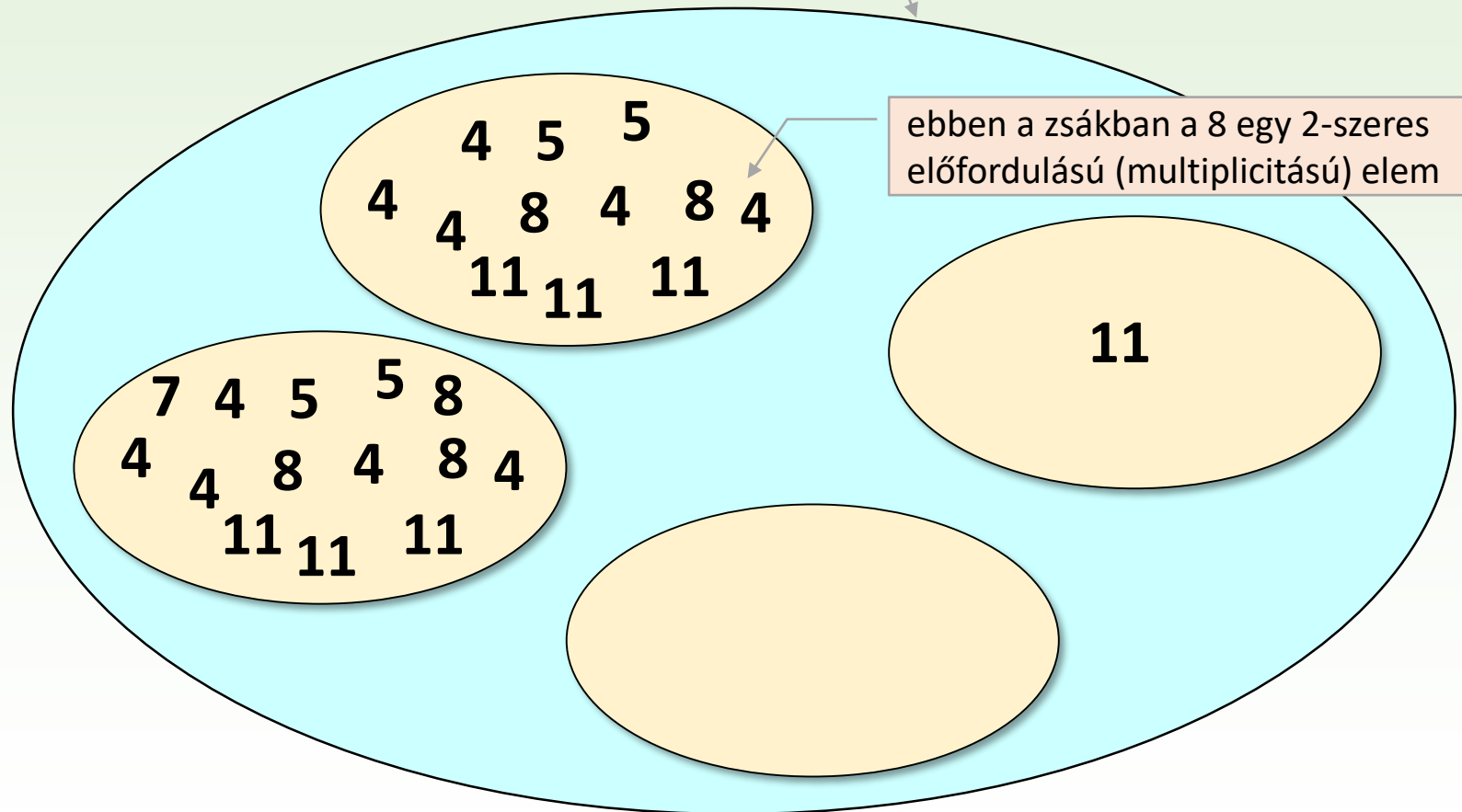
Adattípus

- ❑ Egy adat típusát az adat által felvehető **értékek** és az azokkal végezhető **műveletek** mutatják (**specifikálják**).
- ❑ Bizonyos esetekben nekünk kell megadni azt is, hogy hogyan ábrázoljuk a típusértékeket a számítógépen (**reprezentáció**), és ennek ismeretében milyen programokkal oldjuk meg a típusműveleteket (**implementáció**). Erre a két lépésre együttesen azt mondjuk, hogy **megvalósítjuk** a típust.



Zsák típus típusérték-halmaza

Egy zsák típusú adat által felvehető értékek zsákok, amelyek a zsák típus típusérték-halmazát alkotják.



Zsák típus műveletei

Kiüríti a zsákot (erase):

$b := \underline{\emptyset}$

$b: \text{Zsák}$

üres zsák jele

Betesz egy elemet a zsákba (putIn):

$b := b \underline{\cup} \{e\}$

$b: \text{Zsák}, e: \mathbb{N}$

adjon hibajelzést,
ha $e \notin [0 .. m]$

„zsákba betesz” művelet jele

Zsák leggyakoribb eleme (mostFrequented) :

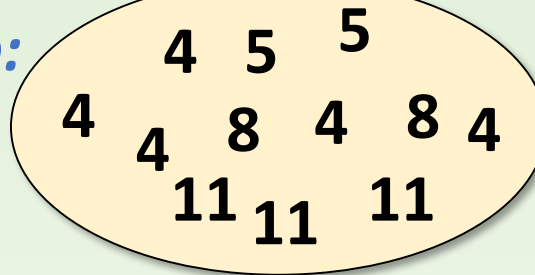
$e := \text{leggyakoribb}(b)$ $b: \text{Zsák}, e: \mathbb{N}$

adjon hibajelzést,
ha a zsák üres

zsák leggyakoribb elemét
kiválasztó művelet jele

Zsák típus reprezentációja

b:



Itt használjuk ki, hogy a zsákba csak $0..m$ közé eső természetes számok kerülhetnek.

vec:

0	0	0	0	5	2	0	0	2	0	0	3	...
---	---	---	---	---	---	---	---	---	---	---	---	-----

 : $\mathbb{N}^{0..m}$

max:

4

 : \mathbb{N}

külön nyilvántartjuk a leggyakoribb elemet

típusinvariáns:

$\max \in [0..m] \wedge$

$\vec{vec}[\max] = \max \bigwedge_{i=0}^m \vec{vec}[i]$

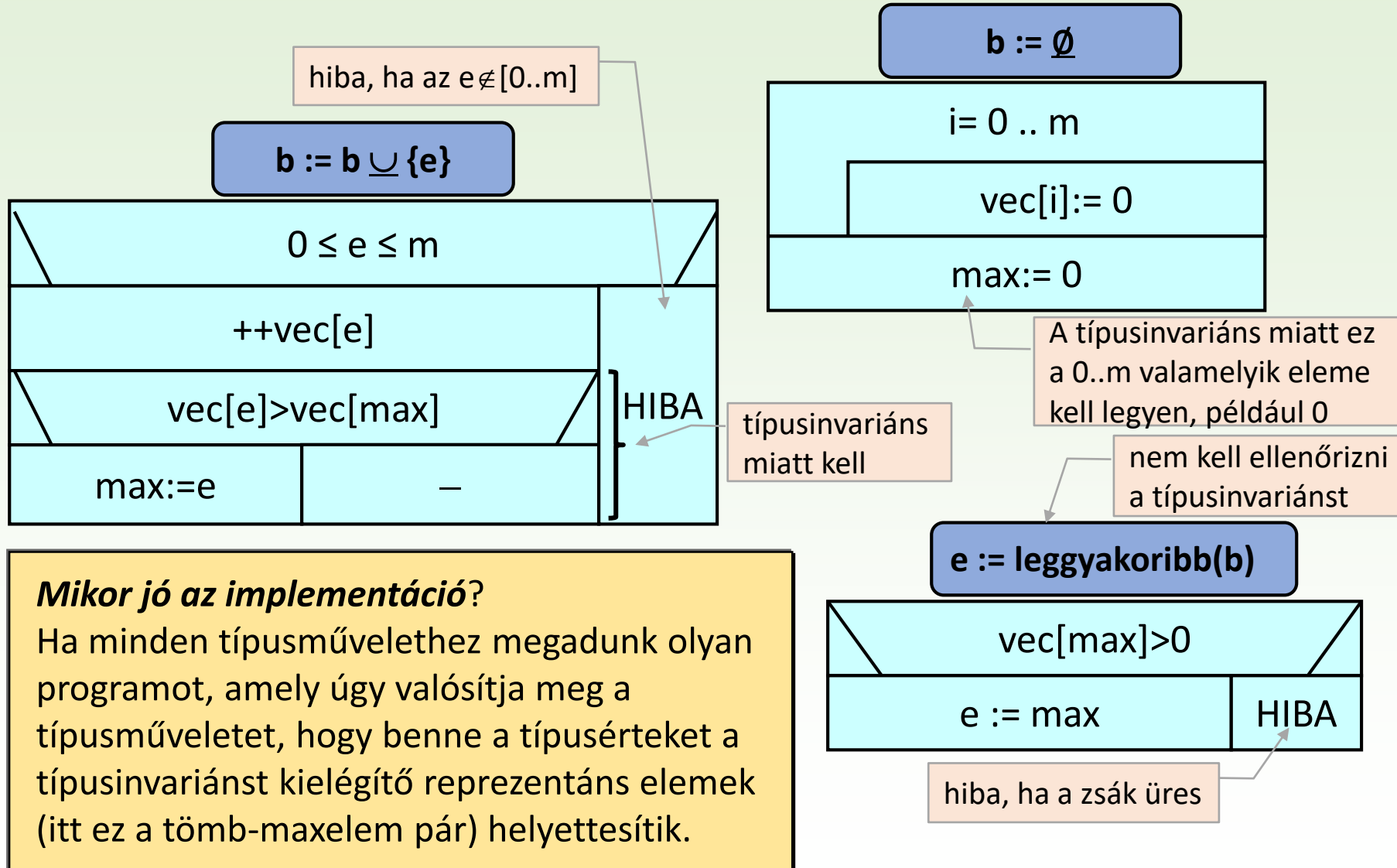
reprezentáló elemek tulajdonságai és a közöttük levő kapcsolat

$\vec{vec}[\max] = 0$
esetén a zsák üres

Mikor jó egy reprezentáció?

Ha minden típusértéknek (zsáknak) van típusinvariánst kielégítő reprezentánsa (itt ez egy tömb-maxelem pár), és minden típusinvariánst kielégítő reprezentáns (azaz tömb-maxelem pár) egy típusértéket (zsákot) helyettesít.

Zsák típus implementációja



Zsák típus tesztelése

Zsák típus tesztje

`erase()` metódus tesztje:

- üres zsák jön-e létre: `b.mostFrequented()==0`

`putIn()` metódus tesztje:

- elem betevése üres zsákba
- elem betevése olyan nem üres zsákba ahol az elem még szerepel
- elem betevése olyan zsákba ahol az elem már szerepel
- olyan elem betevése, amitől a max értékének változnia kell
- a 0 illetve az m elem betevése
- illegális elem betevése

Minden metódusnál ellenőrizni kell, hogy az invariáns megmarad-e.

`mostFrequented()` metódus tesztje: (mintha maximum kiválasztás lenne)

- üres zsák esetén
- egy elemű zsák esetén
- több elemű zsák esetén, amelyben a leggyakoribb elem egyértelmű
- több elemű zsák, amelyben a leggyakoribb elem nem egyértelmű
- amikor a 0 illetve az m elem a leggyakoribb

Integrációs teszt:

- `b.erase()` utána sokszor lefut a `b:=b.putIn(e)`
- `b.erase()` utána rögtön az `e:=b.mostFrequented()`

Zsák típus tesztesetei

erase()		Tesztadatok			
üres zsák létrehozása	m = 0	→ vec = < 0 >			
	m = 1	→ vec = < 0, 0 >		max = 0 ∨ 1	
	m = 4	→ vec = < 0, 0, 0, 0, 0 >		max = 0	

mostFrequented()		Tesztadatok			
üres zsák		m = 1	vec = < 0, 0 >	→ hiba	
egy elemű		m = 1	vec = < 1, 0 >	→ 0	
egyértelmű		m = 1	vec = < 2, 1 >	→ 0	
többértelmű		m = 1	vec = < 2, 2 >	→ 0 ∨ 1	
max = 0		m = 1	vec = < 1, 0 >	→ 0	
max = m		m = 1	vec = < 0, 1 >	→ 1	

putIn()	Tesztadatok					
első	m = 1	vec = < 0, 0 >	max=0	putIn(0)	→ vec = < 1, 0 >	max = 0
új	m = 1	vec = < 1, 0 >	max=0	putIn(1)	→ vec = < 1, 1 >	max = 0
létező	m = 1	vec = < 1, 1 >	max=0	putIn(1)	→ vec = < 1, 2 >	max = 1
létező	m = 1	vec = < 2, 1 >	max=0	putIn(1)	→ vec = < 1, 2 >	max = 0
0	m = 1	vec = < 1, 1 >	max=0	putIn(0)	→ vec = < 2, 1 >	max = 0
m	m = 1	vec = < 1, 1 >	max=0	putIn(1)	→ vec = < 1, 2 >	max = 1
illegális	m = 1	vec = < 1, 2 >	max=0	putIn(2)	→ hiba	

Alapfogalmak

például egy zsák

- Az objektum (*object*) a feladat **egy adott részéért felelős egység**, amely tartalmazza az ezen részhez tartozó **adatokat** és az ezekkel kapcsolatos **műveleteket**.

erase, putIn, mostFrequented

a zsákot reprezentáló *vec* és *max*

- Az osztály (*class*) egy **objektum szerkezetének és viselkedésének mintáját** adja meg, azaz

- felsorolja az objektum **adattagjait** (azok nevét és típusát)
- megadja az objektumra meghívható **metódusokat**

a *vec* egy tömb, a *max* egy egész

- Az osztály lényegében az objektum típusa: az objektumot az osztálya alapján hozzuk létre, azaz **példányosítjuk**.

- Egy osztály alapján több objektum is példányosítható.

erase() : void,
putIn(int) : void,
mostFrequented() : int

1

Az objektum-orientáltság lényeges ismérve az **egységbezárás**: egy adott tárgykör megvalósításához szükséges adatokat és az azokat manipuláló programrészeket egy egységként a program többi részétől elkülönítve adjuk meg.

Osztály UML jelölése

□ Egy osztály leírásához szükséges

- a **neve**, **adattagjai**, **metódusai**, illetve az adattagjainak és metódusainak **láthatósági** megszorítása, amely lehet
 - kívülről is látható, azaz publikus (*public* +)
 - külvilág előtt rejtett: privát (*private* -) vagy védett (*protected* #).

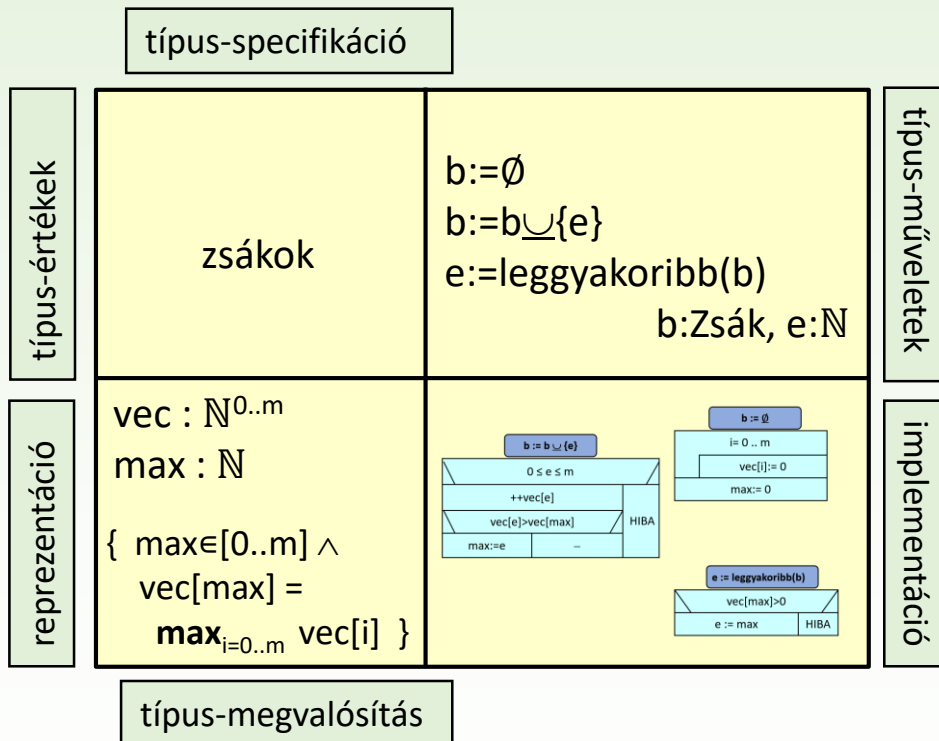
<osztálynév>
<+ - #> <adattagnév> : <típus>
...
<+ - #> <metódusnév>(<paraméterek>) : <típus>
...

Zsák
- vec : int[0..m]
- max : int
+ erase() : void
+ putIn(e:int) : void
+ mostFrequented() : int

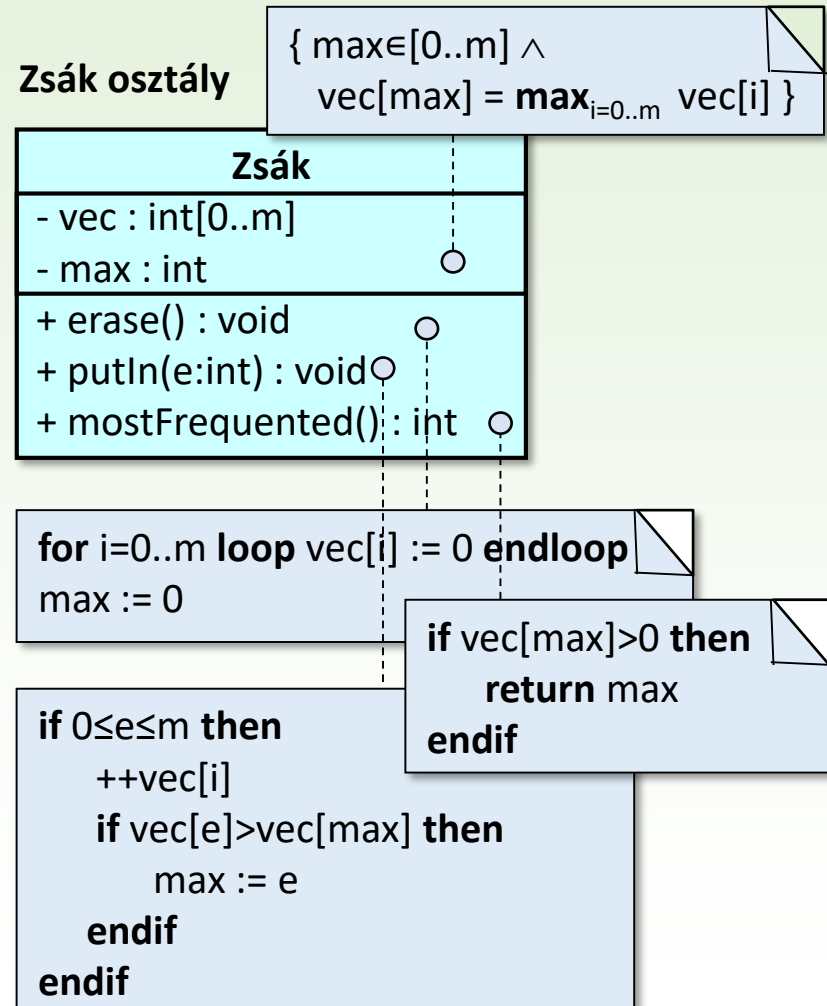
Típus és Osztály

- Az objektumelvű tervezés során egy típust osztályként adhatunk meg.

Zsák típus



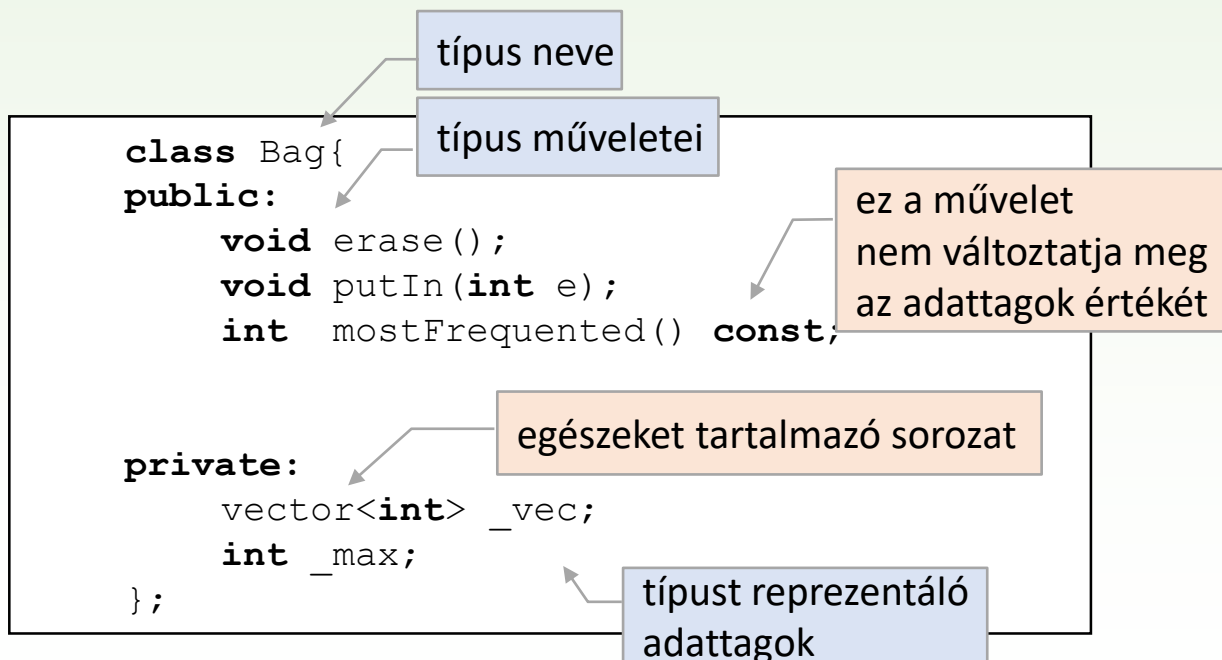
Zsák osztály



Felhasználói típus kódja

- Az objektum-orientált programozási nyelvekben egy **felhasználói típus** (*custom type*) leírását az osztály (*class*) nyelvi elemmel adhatjuk meg.
- az osztály **neve** azonos a típus nevével
 - a típus reprezentációjához használt elemek az osztály **adattagjai** (van nevük és típusuk)
 - az osztály **metódusai** a típusműveletek

Zsák
- vec : int[0..m]
- max : int
+ erase() : void
+ putIn(e:int) : void
+ mostFrequented() : int



Láthatóság

```
class Bag{
public:
    void erase();
    void putIn(int e);
    int mostFrequented() const;
private:
    vector<int> _vec;
    int _max;
};

int main()
{
    Bag b;
    b.erase();
    b.putIn(5);
    int a = b.mostFrequented();

    b._vec[5]++;
}
```

Egy objektum metódusait mindig az objektumra hívjuk meg (ez az objektum a metódusának kitüntetett adata lesz). A metódus törzsében az objektum külön feltüntetése nélkül hivatkozhatunk ennek az objektumnak adattagjaira, és hívhatjuk más metódusait.

Az objektum osztályán kívül a privát tagokra nem tudunk közvetlenül hivatkozni.

2

Az objektum orientáltság fontos ismérve az **elrejtés**: az egységbe zárt elemek láthatóságát korlátozzuk. (Általában az adattagok rejtettek, azok értékéhez csak közvetetten, a publikus metódusokkal férünk hozzá.)

Zsák típus osztályának metódusai

```
void Bag::erase() {  
    for(unsigned int i=0; i<_vec.size(); ++i) _vec[i] = 0;  
    _max = 0;  
}  
  
void Bag::putIn(int e) {  
    if( e<0 || e>=int(_vec.size()) ) return; // hibakezelés kell  
    if( ++_vec[e] > _vec[_max] ) _max = e;  
}  
  
int Bag::mostFrequented() const {  
    if( 0 ==_vec[_max] ) return -1; // hibakezelés kell  
    return _max;  
}
```

a Bag osztályhoz tartozó

sorozat hossza

castolás int-re

Konstruktor

Egy objektum létrehozásakor (**példányosításakor**) egy speciális metódust, a **konstruktort** hívjuk meg, amely többek között lefuttatja az objektum adattagjainak konstruktorait is.

Minden objektumnak – feltéve, hogy más konstruktort nem adunk meg – van egy ún. **üres konstruktora**, amely nem vár paramétert, csak meghívja az adattagok üres konstruktorait. A *Bag b* deklaráció ezért egy nulla hosszú vektort és egy integert hoz létre. De ez nekünk most nem elég!

```
class Bag{
public:
    Bag(int m);
    ...
private:
    vector<int> _vec;
    int _max;
};
```

Kell egy olyan konstruktor, amely segítségével megadhatjuk, hogy milyen hosszú tömbre lesz szükségünk a reprezentációban. Ezt a konstruktort például a *Bag b(21)* utasítással hívhatjuk meg.

a konstruktor törzse átméretezi (*resize*) a nulla hosszú vektort, és lefuttatja az *erase()*-t.

```
Bag::Bag(int m) { _vec.resize(m+1); erase(); }
```

elvégzi azt is, amit az *erase()*

```
Bag::Bag(int m) { _vec.resize(m+1, 0); _max = 0; }
```

vector-nak, int-nek is vannak nem üres konstruktorai

```
Bag::Bag(int m) : _vec(m+1, 0), _max(0) { }
```

Zsák osztálya pontosabban

fejállományok (header fájlok) elejére

```
#pragma once  
#include <vector>
```

```
class Bag{  
public:  
    enum Errors{EmptyBag, WrongInput};
```

hibaesetek:

az **enum** egy olyan új típust definiál,
amelynek csak típusértékei vannak

```
    Bag(int m);  
    void erase();  
    void putIn(int e);  
    int mostFrequented() const;
```

```
private:  
    std::vector<int> _vec;  
    int _max;  
};
```

külön fejállomány

bag.h

A header fájlokban nem szokás a *using namespace std*, ezért külön jelezzük, hogy a *vector* definíciója az **std névtérben** található.

Zsák osztályának metódusai

Bag osztály definíció bemásolása

```
#include "bag.h"
```

```
Bag::Bag(int m) : _vec(m+1,0), _max(0) { }
```

```
void Bag::erase() {  
    for(unsigned int i=0; i<_vec.size(); ++i) _vec[i] = 0;  
    _max = 0;  
}
```

```
void Bag::putIn(int e) {  
    if( e<0 || e>=int(_vec.size()) ) throw WrongInput;  
    if( ++_vec[e] > _vec[_max] ) _max = e;  
}
```

```
int Bag::mostFrequented() const {  
    if( 0 == _vec[_max] ) throw EmptyBag;  
    return _max;  
}
```

kivétel kiváltása:
jelzi a hibát, de nem kezeli le

kivétel kiváltása

külön fordítási egység

bag.cpp

Főprogram

```
#include <iostream>
#include <fstream>
#include "bag.h"
using namespace std;

int main()
{
    ifstream f( "input.txt" );
    if(f.fail()){ cout << "Wrong file name!\n"; return 1;}
    int m; f >> m;
    if(m<0){
        cout << "Upper limit of natural numbers cannot be negative!\n";
        return 1;
    }
    try{
        Bag b(m);
        int e;
        while(f >> e) { b.putIn(e); }
        cout << "The most frequented element: " << b.mostFrequented() << endl;
    } catch(Bag::Errors ex){
        if (ex==Bag::WrongInput){ cout << "Illegal integer!\n"; }
        else if(ex==Bag::EmptyBag) { cout << "No input no maximum!\n"; }
    }
    return 0;
}
```

Bag osztály definíció bemásolása

kivétel figyelése

kivétel elkapása és lekezelése

main.cpp

Automatikus tesztelés


```
#include <iostream>
#include <fstream>
#include "bag.h"
using namespace std;

#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("empty sequence", "[sum]") {
    ifstream f( "input1.txt" );
    int m; f >> m;
    Bag b(m); b.erase();
    int e;
    while(f >> e){ b.putIn(e); }
    CHECK_THROWS(b.mostFrequented());
}

TEST_CASE("one element in the file", "[sum]") {
    ifstream f( "input2.txt" );
    int m; f >> m;
    Bag b(m); b.erase();
    int e;
    while(f >> e){ b.putIn(e); }

    CHECK(b.mostFrequented()==2);
}
...
```



```
REQUIRE(b.mostFrequented()==2)
```

összegzés
intervallum hossza: 0
 $f = < 15 > (m = 15)$
 $b = \{ \} \rightarrow$ nincs leggyakoribb
throw Bag::EmptyBag

összegzés
intervallum hossza: 1
 $f = < 15, 2 > (m = 15)$
 $b = \{ 2(1) \} \rightarrow$ leggyakoribb = 2

Automatikus tesztelés

```
TEST_CASE("creation of an empty bag", "[bag]")
```

```
{  
    int m = 0;  
    Bag b(m);  
    vector<int> v = { 0 };  
    CHECK(v == b.getArray());  
}
```

bag()

üres zsák létrehozása:

$m = 0 \rightarrow _vec = < 0 >$

```
TEST_CASE("new element into empty bag", "[putIn]")
```

```
{  
    Bag b(1);  
    b.putIn(0);  
    vector<int> v = { 1, 0 };  
    CHECK(v == b.getArray());  
}
```

jól jönne ez a `getArray()`
metódus a teszteléshez

putIn()

üres zsákba új elem:

$m = 1, e = 0 \rightarrow _vec = < 1, 0 >$

```
class Bag {
```

```
private:
```

```
    std::vector<int> _vec;
```

```
    int _max;
```

```
public:
```

```
    ...
```

```
    const std::vector<int>& getArray() const {return _vec;}
```

```
};
```

Ez nem szép megoldás, hiszen így „beleszemeteltünk” a Bag osztály kódjába. Elegánsabb lenne készíteni egy új Bag_Test osztályt, amely átvinné (örökölné) a Bag osztály minden tulajdonságát, és kiegészítené azokat a `getArray()` metódussal. A tesztesetek pedig a Bag osztály helyett ezt a Bag_Test osztályt használnák.

„inline” definíció