



Pázmány Péter Katolikus Egyetem  
Információs Technológiai és Bionikai Kar

# OOP és Verem

2018. szeptember 17.



Pázmány Péter Katolikus Egyetem  
Információs Technológiai és Bionikai Kar

# OOP Összefoglaló II.



# Polimorfizmus

- Polimorfizmus (többalakúság): az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat.
  - Statikus típus: a deklaráció során kapja.
  - Dinamikus típus: run-time éppen milyen típusú objektumra hivatkozik = a statikus típus, vagy annak leszármazottja.
- Aki Manager, az egy (is-a) Employee is.
- A Háromszög az egy Alakzat.



# C++

- Tekintsük az alábbi kódot

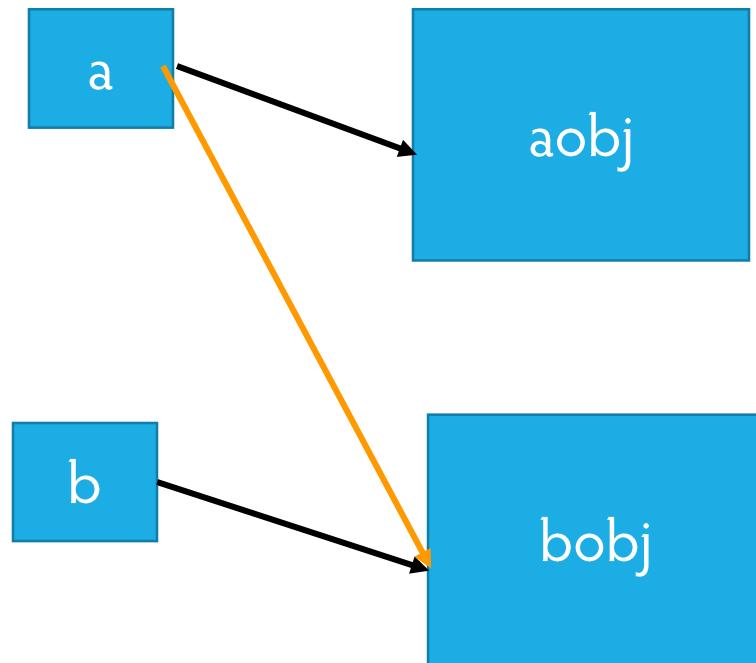
```
Employee emp ("Jozsef", "Fekete", 3);  
Manager m ("Kati", "Kovacs", 3, 2);  
emp = m;
```

- Megengedett

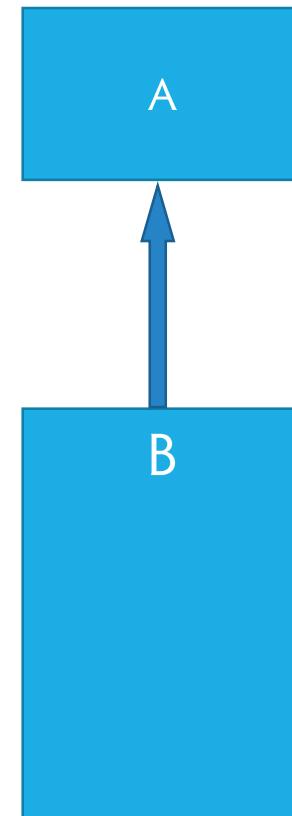
```
emp.print();  
m.print();
```

- Mi történik?

# Hogy lenne jó?



$a := b$





# Altípus

- B típus altípusa A típusnak, amennyiben a B használható (legalább) ott, ahol az A típus használható.
  - C++ esetén a public öröklődéssel létrejött leszármazottak altípusai az ősnek.



# Altípusos polimorfizmus

```
Employee* empp=new Employee ("Istvan", "Nagy", 5);
```

...

```
Manager* mp=new Manager("Laszlo", "Hajto", 2, 3);
```

...

```
empp = mp;
```



# Altípusos polimorfizmus

```
Shape *s;
```

```
...
```

```
s = new Triangle (...);
```

```
...
```

```
s = new Rectangle (...);
```

```
...
```

- Ha B (pl. Triangle, Rectangle) altípusa az A (pl. Shape) típusnak, akkor B objektumainak referenciái értékül adhatók az A típus referenciainak.



# Altípusos polimorfizmus

Shape \*a;

```
Triangle* h= new Triangle (...);
```

```
Rectangle *t= new Rectangle (...);
```

```
a = h; a->draw(); ...
```

```
a = t; a->draw(); ...
```

Melyik draw( )?



# Altípusos polimorfizmus

```
Employee* empp=new Employee ("Istvan", "Nagy",5);
```

...

```
Manager* mp=new Manager("Laszlo", "Hajto",2,3);
```

...

```
empp = mp;
```

...

```
empp->print();
```

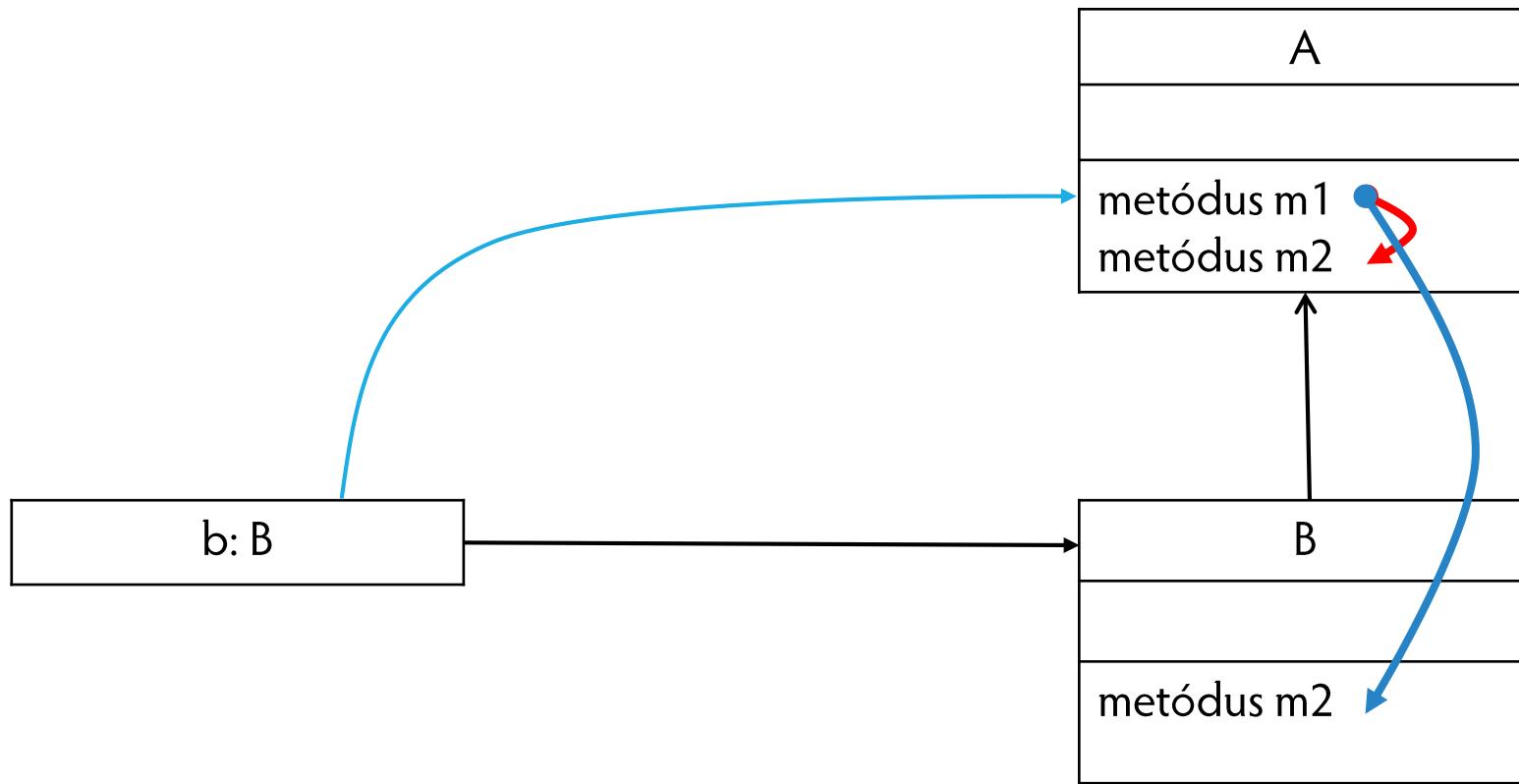
Melyik print?



# Dinamikus összekapcsolás

- Run-time fogalom
  - Az a jelenség, hogy a változó éppen aktuális dinamikus típusának megfelelő metódus implementáció hajtódik végre.
- Előző példák
  - A háromszög kirajzolása
  - A manager kinyomtatása

# Dinamikus összekapcsolás





# Dinamikus összekapcsolás – C++-ban

```
class Employee {  
    ...  
public:  
    ...  
    virtual void print() const {  
        cout << "A vezeteknev:" << name() << '\n';  
        ...  
    }  
    ...  
}
```



# Mi az objektumorientált programozás?

- A programozó definiálhat altípus kapcsolatokat
- A típusszabályok megengedik, hogy az altípus használható legyen a szupertípus helyén (altípusos polimorfizmus)
- Típus-vezérelt metódus elérés (dinamikus kötés)
- Implementáció megosztása (öröklődés)



# Típus-vezérelt metódus elérés

```
s: Shape := new Triangle (3, 4, 5);  
s.draw();
```

Statikus elérés:

A Shape draw metódusát  
hívja

Dinamikus elérés:

A Triangle draw metódusát hívja



# Elérési döntések

- C++
  - Az őstípus virtual-nak deklarálja a metódust, amire megengedi a felüldefiniálást
- Más programozási nyelveknél ez másképp lehet
  - Erről a programozási nyelvek és módszerek tárgyon beszélünk



# C++ példa (folyt.)

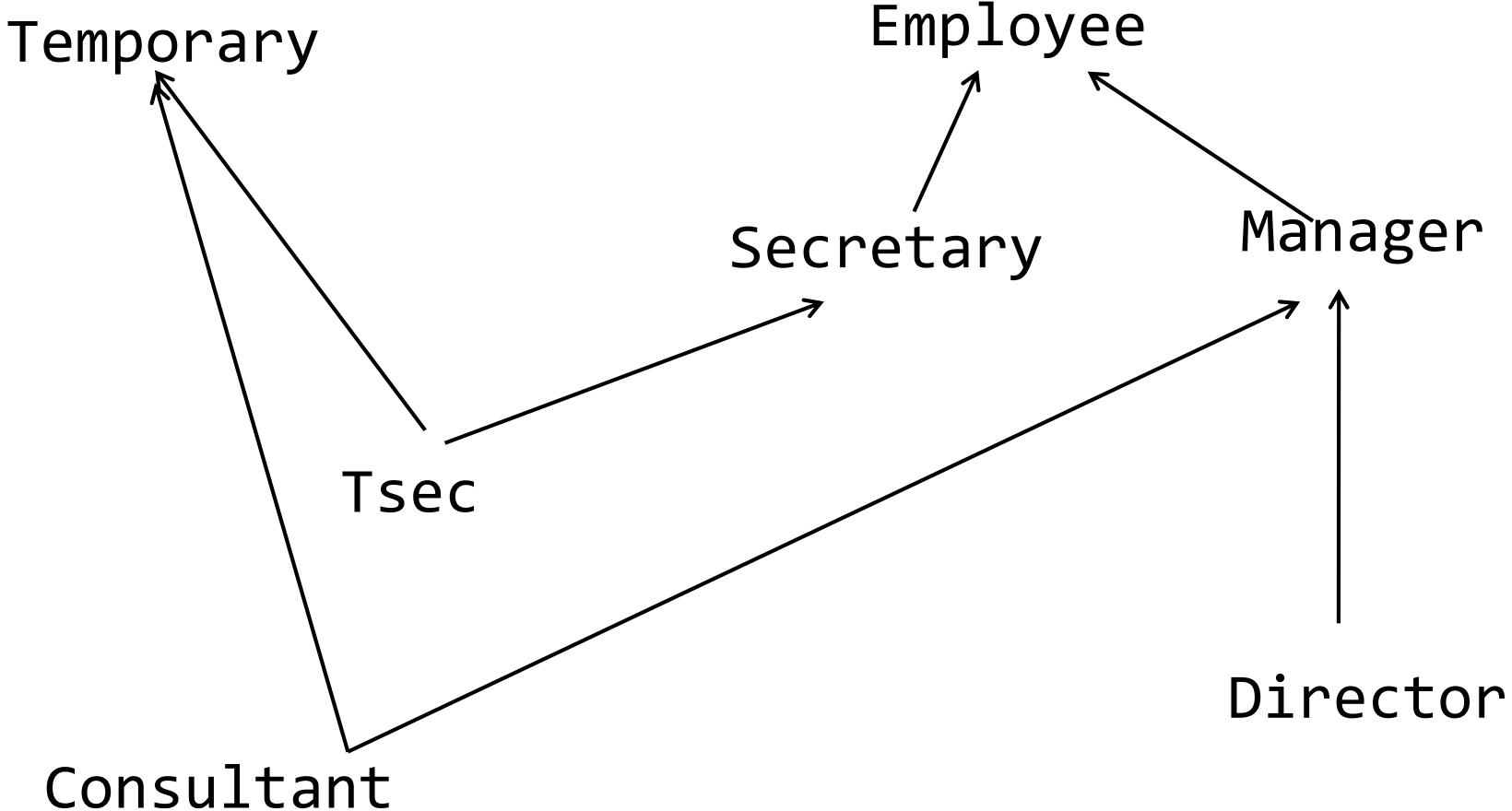
- egy leszármazott lehetős is

```
class Employee { ... };  
class Manager : public Employee {...};  
class Director: public Manager {...};
```

- Az osztályhierarchia lehet fa, de lehet általánosabb gráf is:

```
class Temporary { ... };  
class Secretary: public Employee { ... };  
class Tsec: public Temporary, public Secretary{ ... };  
class Consultant: public Temporary, public Manager { ... };
```

# Példa

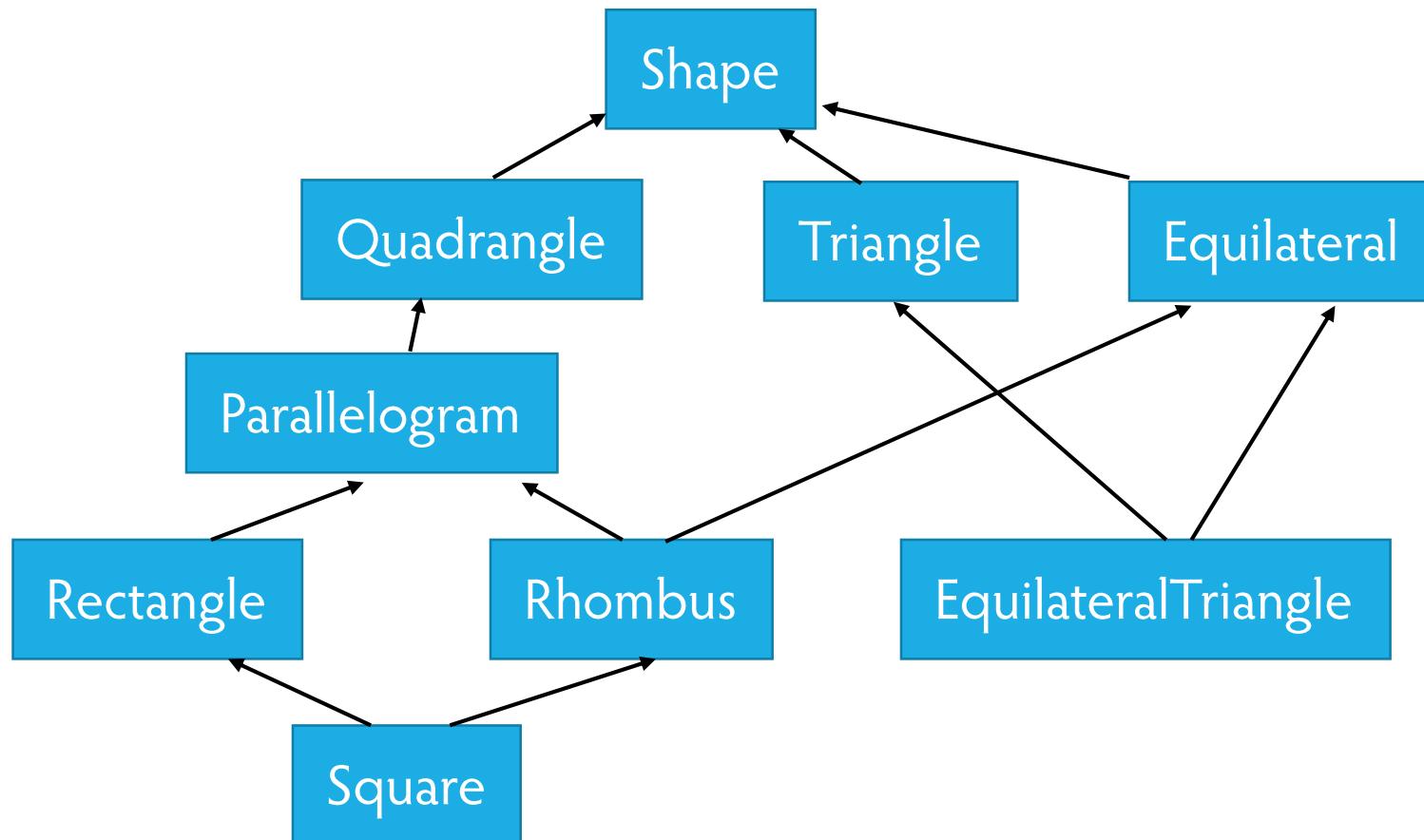




# Implementáció újrahasznosítás: alosztályképzés

- Használd egy típus implementációját egy másik típus implementálására!
- Gyakran használjuk az ōstípus implementációját az altípus implementálására
- A gyakran használt OO programozási nyelvek keverik az altípus és alosztály fogalmakat:
  - C++ - implementációs öröklés altípus nélkül:
    - private, protected öröklés

# Egy típus- és osztályhierarchia





# Adjunk hozzá egy attribútumot!

- Az alakzatoknak legyen színe – `color` – és egy `set_color` metódusa
  - Változtassuk meg a Shape, Quadrangle, Parallelogram, Triangle, Equilateral, EquilateralTriangle, Rhombus, Rectangle, Square stb. Típusokat
  - Változtassuk meg a Shape-t, a többiek öröklik az új attribútumot és metódust automatikusan

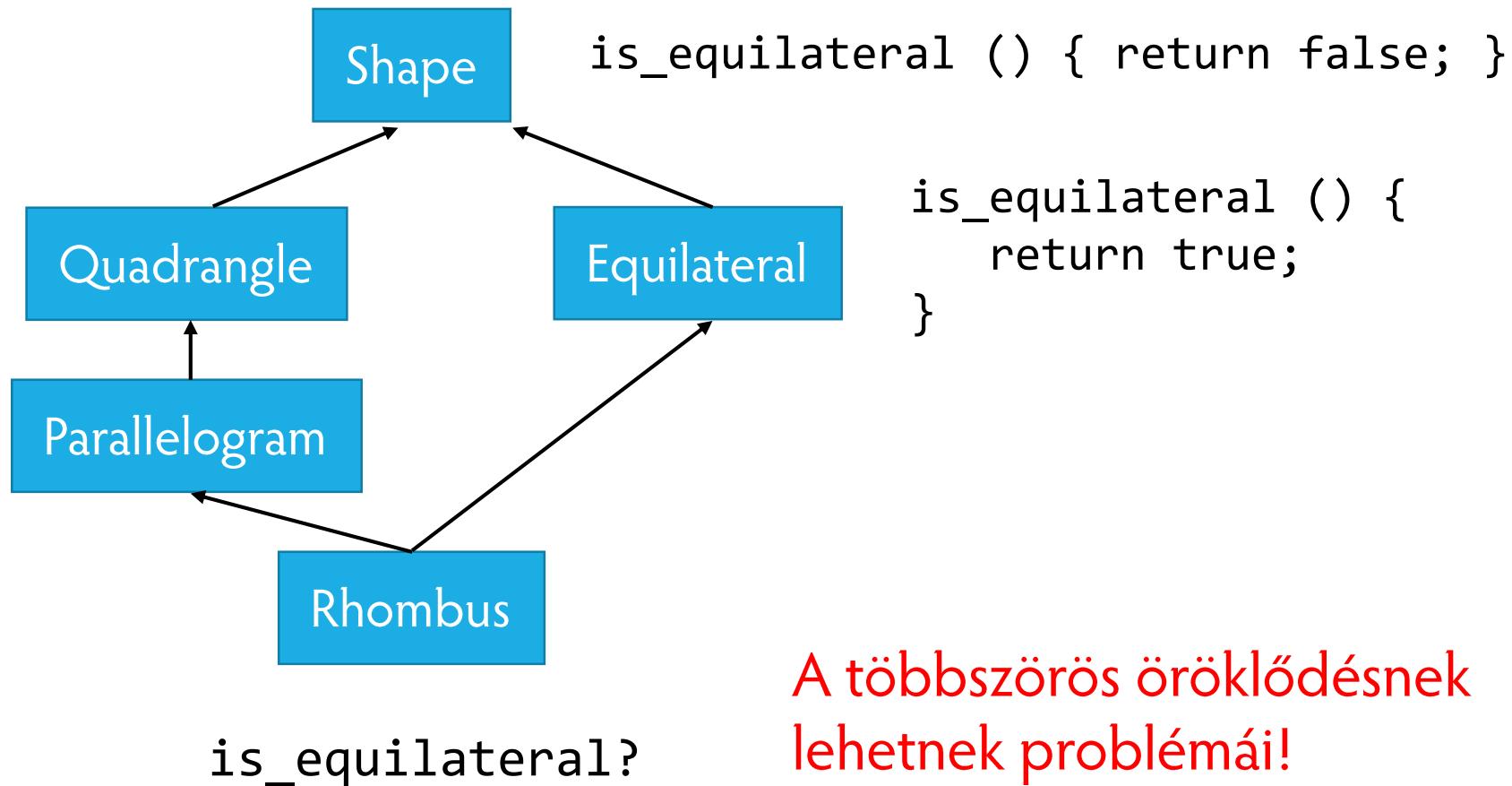


# Adjuk hozzá az `is_equilateral`-t!

```
bool Shape::is_equilateral () {  
    return false;  
}
```

```
bool Equilateral::is_equilateral () {  
    return true;  
}
```

# Egy Rhombus egyenlőoldalú?

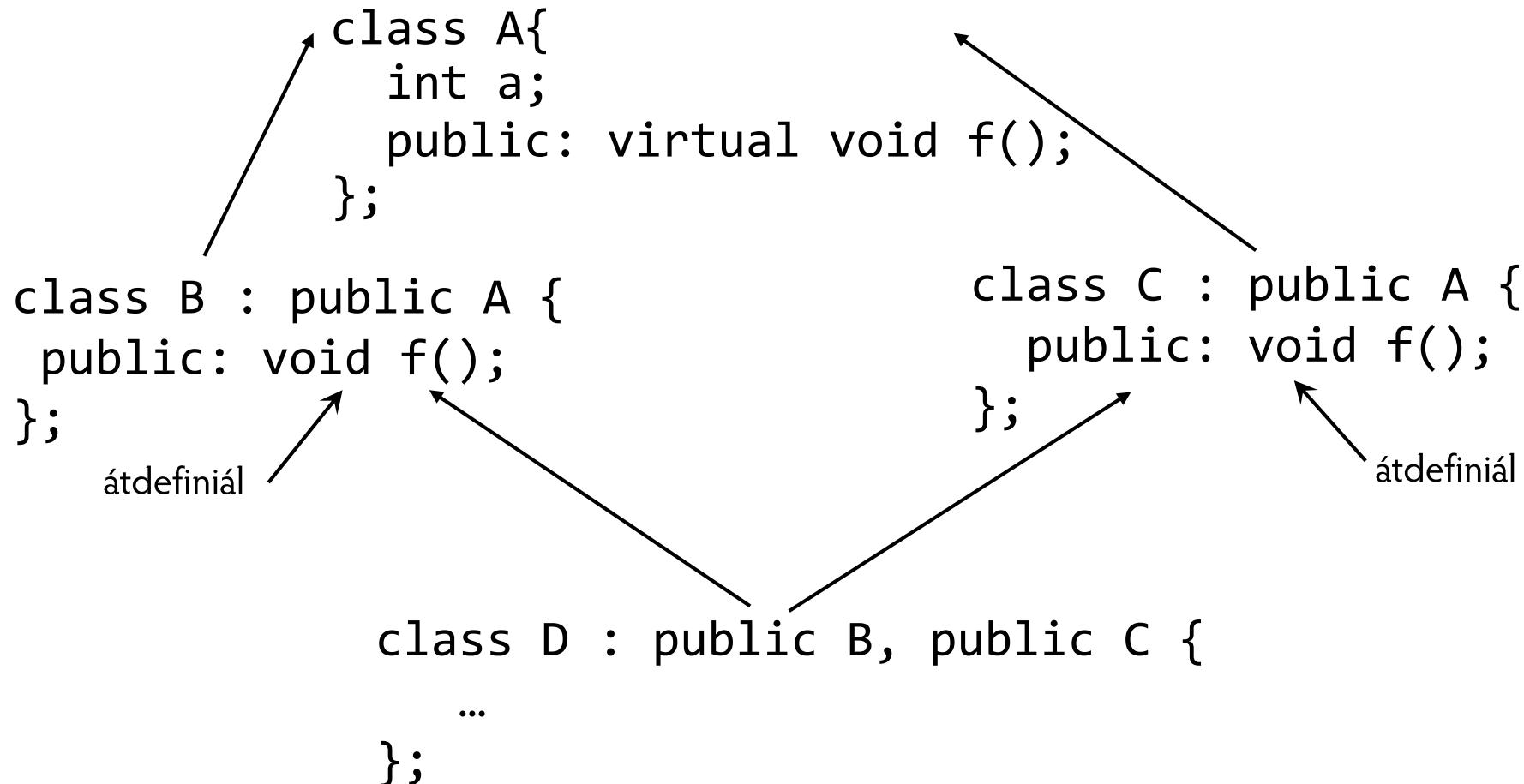




# Többszörös öröklődés

- Egy osztálynak egynél több közvetlen őse lehet
- Problémák:
  - Adattagok hányszor?
  - Melyik metódus?

# A többszörös öröklődés problémái:





# A többszörös öröklődés problémái:

- Ha egy D-beli „f”-re (felüldefiniáltuk B-ben és/vagy C-ben) hivatkozunk, akkor az melyiket jelentse?
  - A ?
  - B ?
  - C ?
- Az „a” attribútum hány példányban jelenjen meg D-ben?
- A két kérdés lényegében ugyanazt a problémát veti fel:  
ha kétértelműség van, hogyan válasszunk?



# Megoldási lehetőségek

- A legtöbb esetben az ilyen kódot nem lehet lefordítani, a fordító, vagy a futtató környezet kétértelműségre (ambiguous) hivatkozva hibajelzéssel leáll.
- Az œsosztály mondja meg, hogy mit szeretne tenni ilyen esetben.
- A származtatott osztály mondja meg, hogy melyiket szeretné használni.



# C++ – többszörös öröklődés

```
class Animal {  
    public: virtual void eat();  
};  
class Mammal : public Animal {  
public: virtual Color getHairColor();  
};  
class WingedAnimal : public Animal {  
public: virtual void flap();  
};  
// A bat is a winged mammal  
class Bat : public Mammal, public WingedAnimal {  
};  
Bat bat;
```

Hogyan eszik?



# C++ – többszörös örökłódés

```
class Mammal : public virtual Animal {  
    public: virtual Color getHairColor();  
    ...  
};  
  
class WingedAnimal : public virtual Animal {  
    public: virtual void flap();  
    ...  
};  
  
// A bat is still a winged mammal  
class Bat : public Mammal, public WingedAnimal {  
    ...  
};
```



# Absztrakt osztály

```
class Alakzat{
public:
    virtual void kirajzol()=0;
    virtual bool zart()=0;
...
}
```



# Absztrakt osztály

- Tervezés eszköze
- Egy felső szinten összefogja a közös tulajdonságokat
- A metódusok között van olyan, aminek csak specifikációja van, törzse nincs
  - Nem hozható létre példánya
  - A leszármazott teszi konkréttá



# C++ - Absztrakt osztály

```
class Sikidom {  
protected:  
    int szelesseg, magassag;  
public:  
    virtual int terulet()=0;  
    void beallit_ertekek(int a, int b)  
        {szelesség = a; magassag = b;}  
};  
  
class Teglalap: public Sikidom{  
public:  
    int terulet()  {return (szelesseg * magassag);} ...  
};  
  
class Haromszog: public Sikidom{  
public:  
    int terulet()  {return (szelesseg * magassag /2);} ...  
};
```



# C++

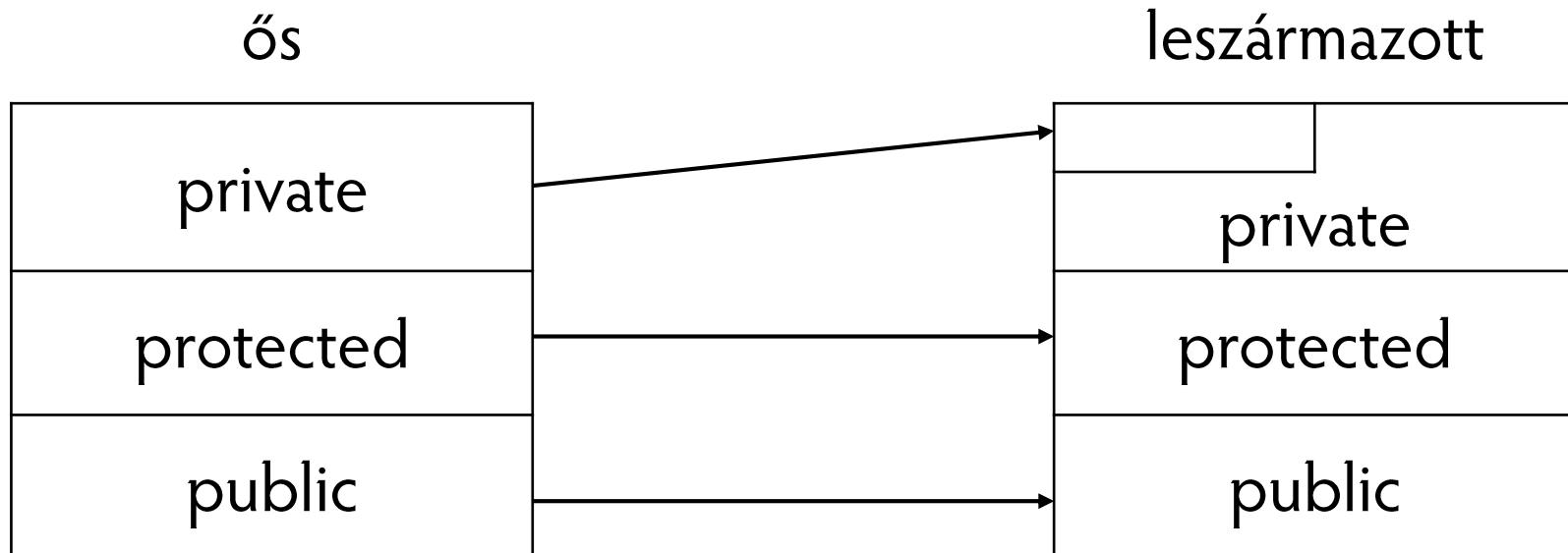
- Nem lehet:  
`Sikidom a;`
- Lehet:  
`Teglalap t;`  
`Haromszog h;`

`Sikidom* a2 = &t;`  
`Sikidom* a3 = &h;`

`a2->terulet(); //Teglalap::terulet()`  
`a3->terulet(); //Haromszog::terulet()`

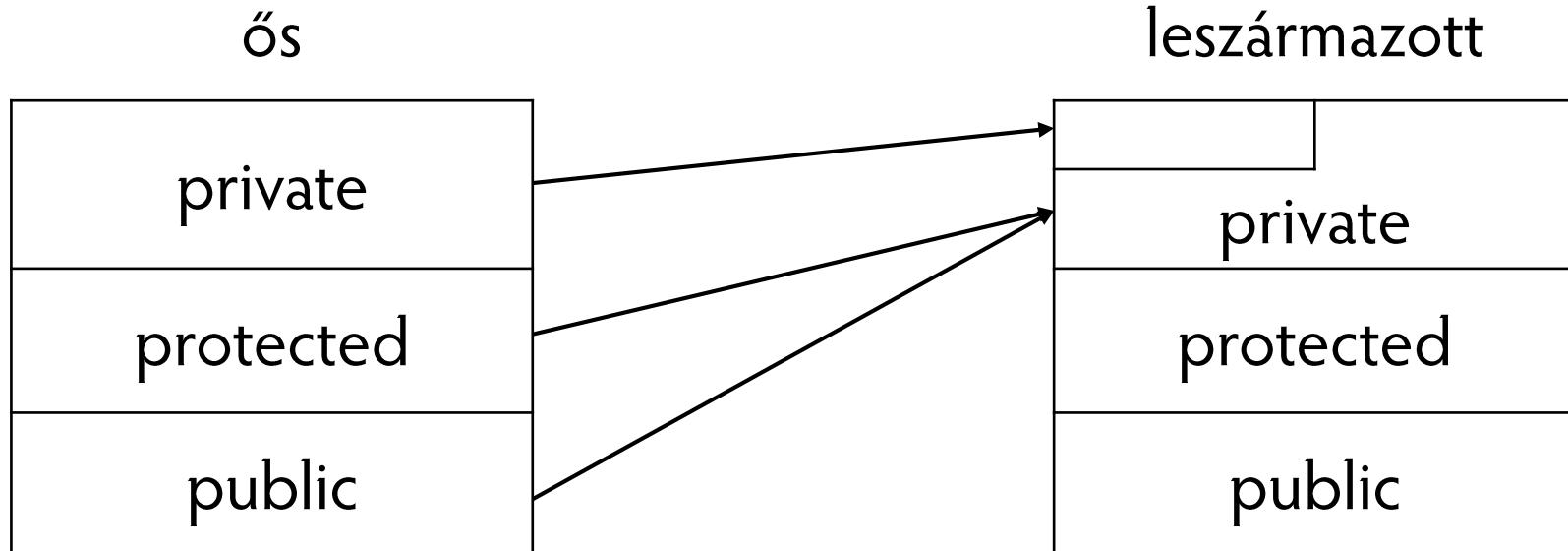
# Láthatóságok öröklődésnél

- public öröklődés:



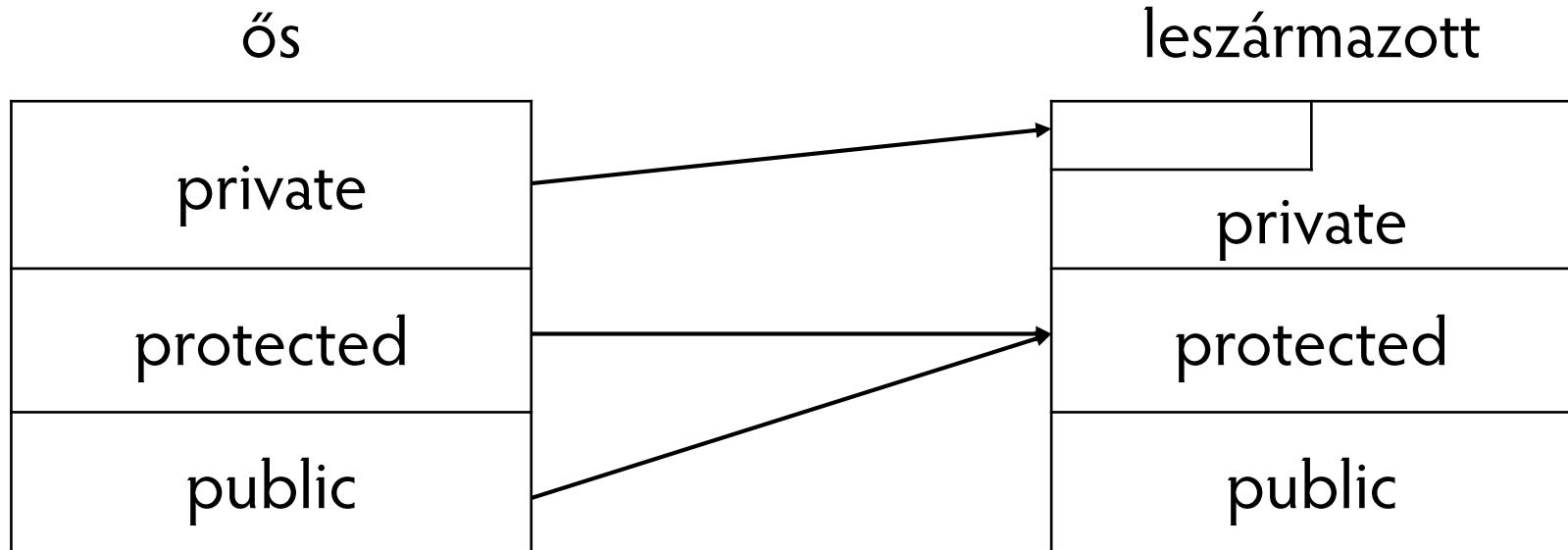
# Láthatóságok öröklődésnél

- private öröklődés:



# Láthatóságok öröklődésnél

- protected öröklődés:





# Öröklés

- Mit örököl a leszármazott?
  - Adattagokat
  - Metódusokat
- Mit nem örököl a leszármazott?
  - Ősosztály konstruktörait, destruktörét
    - Ám tudja használni / meghívni / delegálni
  - Ősosztály értékkadás operátorát
  - Ősosztály barátait



# Öröklés

- Mit vezethet be a leszármazott osztály?

- Új adattagokat
- Új metódusokat
- Felüldefiniálhat már meglévőket
- Új konstruktorkat és destruktort
- Új barátokat



# Friend

- Egy általános metódus deklarációja a következőket jelenti:
  1. A metódus elérheti a privát mezőket is
  2. Az osztály scope-ját használja
  3. A metódus egy konkrét objektumra hívódik meg, ezért eléri a 'this' pointert
- Statikus metódus esetén a 3. pont nem érvényes
- friend függvény esetén csak az 1. jogunk lesz



# Operátorok – Friend

```
typedef double Angle;
class Complex {
public:
    Complex(double r=0, double i=0){ R = r; I = i;}
    Complex operator =(Complex z){R = z.R; I = z.I; return *this; }
    Complex operator +(Complex z) {return Complex(R+z.R,I+z.I);}
    Complex operator +(double x) { return Complex(R+x,I);}
    Complex operator *(Complex);
    Complex operator *(double);
    Complex operator -(Complex);
    Complex operator -(double);
    Complex operator /(Complex);
    Complex operator /(double);
    double Re(); double Im();
    double Abs(); Angle Phi();
private:
    double R;  double I;
};
```



# Operátorok – Friend

```
Complex operator + (double x, Complex z){ return z+x; }
```

- Vagy: osztály belsejében:

```
friend Complex operator+(double, Complex);  
Complex operator+(double p1, Complex p2)  
{  
    Complex temp;  
    temp.R = p1+p2.R;  
    temp.I = p2.I;  
    return (temp);  
}
```



# Friend

- Még egy (tipikus) friend példa

```
class Point {  
    friend ostream &operator<<( ostream &, const Point &);  
public:  
    Point( int = 0, int = 0 );           // default constructor  
    void setPoint( int, int );         // set coordinates  
    int getX() const { return x; }     // get x coordinate  
    int getY() const { return y; }     // get y coordinate  
protected:                      // accessible by derived classes  
    int x, y;                      // x and y coordinates of the Point  
}; // end class Point
```



# Verem – LIFO



Pázmány Péter Katolikus Egyetem  
Információs Technológiai és Bionikai Kar

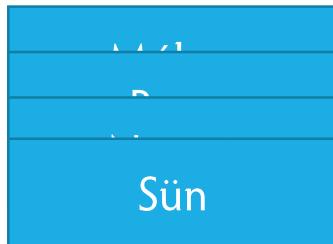
# Verem

Stack, LIFO

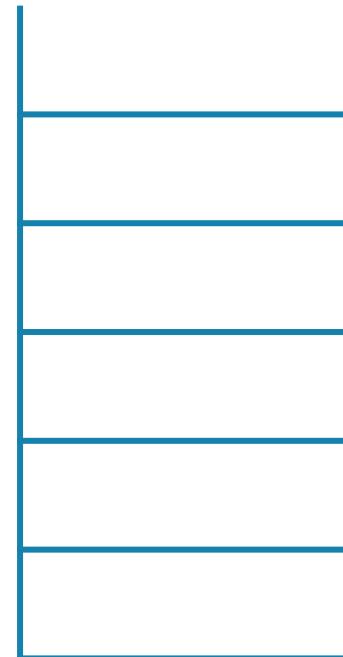


# VEREM

- LIFO: last-in, first-out
- Köznapi fogalma



- Kimászás sorrendje





# A verem ADT axiomatikus leírása

## E alaptípus feletti V verem típus jellemzése

- Műveletek

- empty  $\rightarrow V$
- isempty  $V \rightarrow L$
- push  $V \times E \rightarrow V$
- pop  $V \rightarrow V \times E$
- top  $V \rightarrow E$

- Megszorítások:

- pop és top értelmezési tartománya
  - $V \setminus \{\text{empty}\}$

- Műveletek jelentése

- Üres verem létrehozása
- Üres a verem?
- Elem betétele a verembe
- Elem kivétele a veremből
- Felső elem lekérdezése

- Figyelem!

- A műveletek között nem szerepel „isfull” művelet!



# A verem ADT axiomatikus leírása

- Axiómák
  - $\text{isempty}(\text{empty})$ 
    - Vagy:  $v = \text{empty} \rightarrow \text{isempty}(v)$
  - $\text{isempty}(v) \rightarrow v = \text{empty}$
  - $\neg\text{isempty}(\text{push}(v, e))$
  - $\text{pop}(\text{push}(v, e)) = (v, e)$
  - $\text{push}(\text{pop}(v)) = v$
  - $\text{top}(\text{push}(v, e)) = e$



# A verem ADT funkcionális leírása

- Matematikai reprezentáció
  - a verem rendezett párok halmaza $v = \{(e_1, t_1), \dots (e_i, t_i) | a t_j \text{ komponensek különbözőek}\}$ 
    - 1.komponens: a veremben elhelyezett (push) érték
    - 2.komponens: a verembe helyezés (push) időpontja
- Megszorítás (invariáns)
  - az idő értékek különbözők
- A valóságban nem így implementáljuk!



# A verem ADT funkcionális leírása

- A „pop” specifikációja:
  - Állapottér
    - $V \times E$  (Állapottér típusai:  $V = \{(e_i, t_i), \dots\}$ )
    - $v \quad e$  (Állapottér változói)
  - Paramétertér
    - $V$  és  $v'$
  - Előfeltétel és utófeltétel:
  - Ef = ( $v = v' \wedge v' \neq \emptyset$ )
  - Uf =  $\left( (v = v' \setminus \{(e_j, t_j)\}) \wedge (e = e_j) \wedge ((e_j, t_j) \in v') \wedge (\forall i ((e_i, t_i) \in v' \wedge i \neq j) : t_j > t_i) \right)$

# Verem

## Műveletek jelölése

$\text{push}(v, e)$



procedurális szemlélet

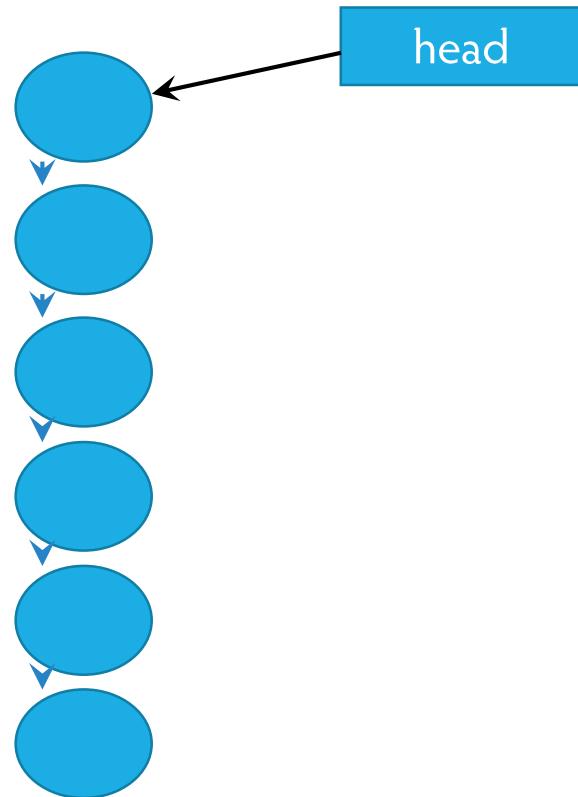
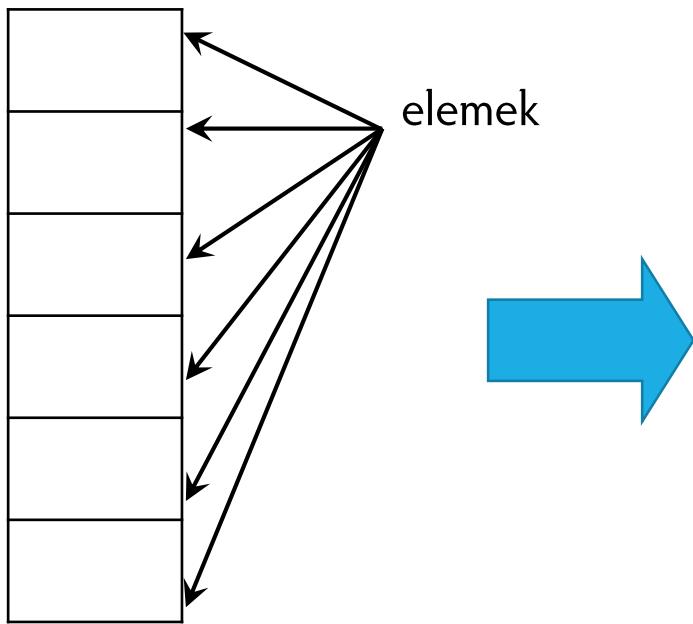
$v.\text{push}(e)$



objektum-elvű szemlélet

# Verem – ADS

- Lineáris adatszerkezet



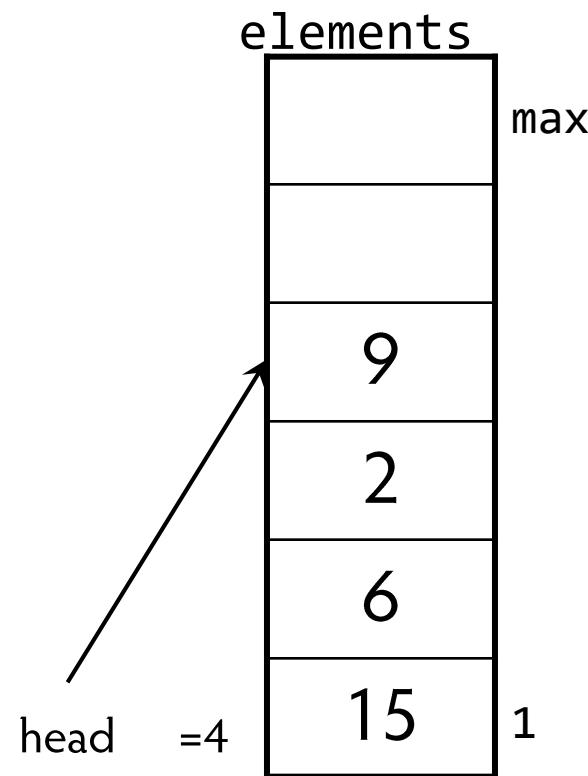


# Elemek száma

- Statikus vs. Dinamikus
  - Az adatszerkezet elemeinek száma a feldolgozás során rögzített vagy változtatható
  - A rögzített nem jelenti, hogy a tárolt adatok nem megváltoztathatók
- Kapacitás: fix vs. változó
  - Fix: A tárolható adatelemek számának felső korlátja a létrehozáskor (esetleg fordítási időben) rögzített.
  - Változó: A memória mérete (illetve kapcsolódó technikai korlátok) szab határt az adatelemek számánának

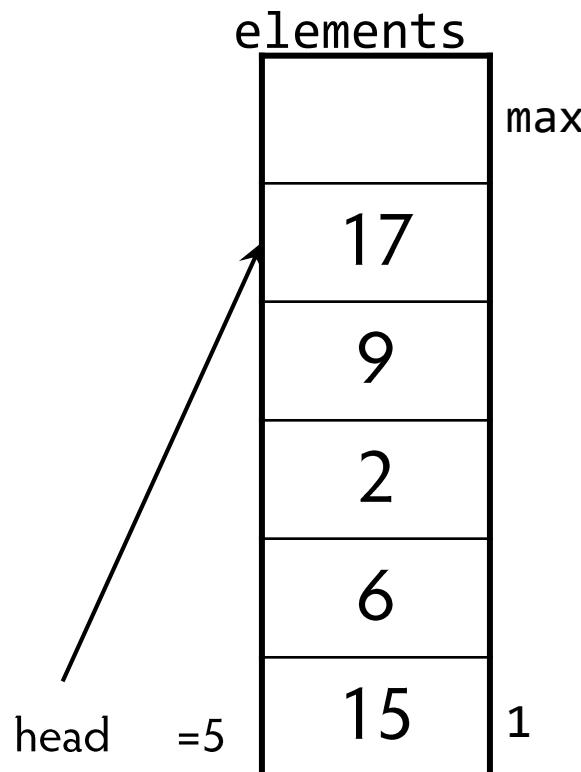
# Reprezentáció

- Aritmetikai ábrázolás:
  - egy max hosszú vektor (ez az elemek tömbje)  
`elements[1..max]`
  - a verem tetejének mutatója  
`head ∈ [0, max]`  
 $\text{head}=0 \Leftrightarrow$  üres a verem
  - Választási lehetőség, hogy hova mutat a head
    - Az első szabad helyre
    - Az utolsó elfoglalt helyre



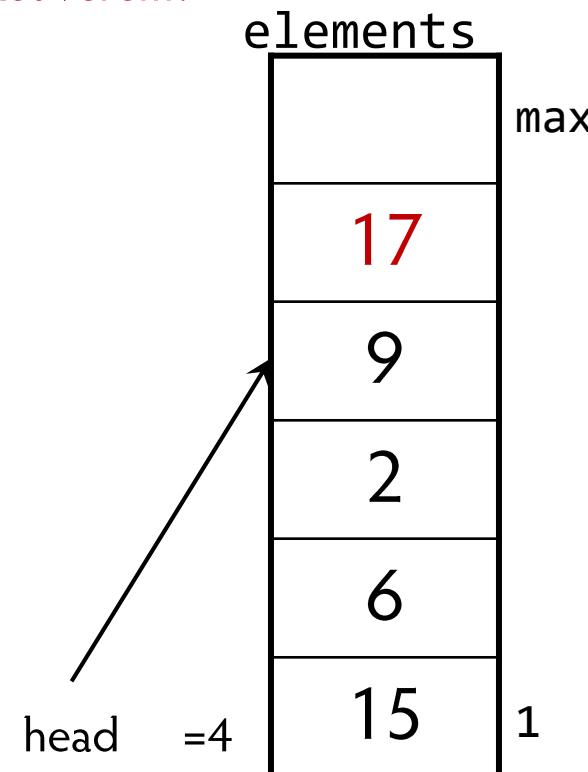
# Reprezentáció

**v.push(17) után**



**v.pop() után**

Egyenlő a két verem?





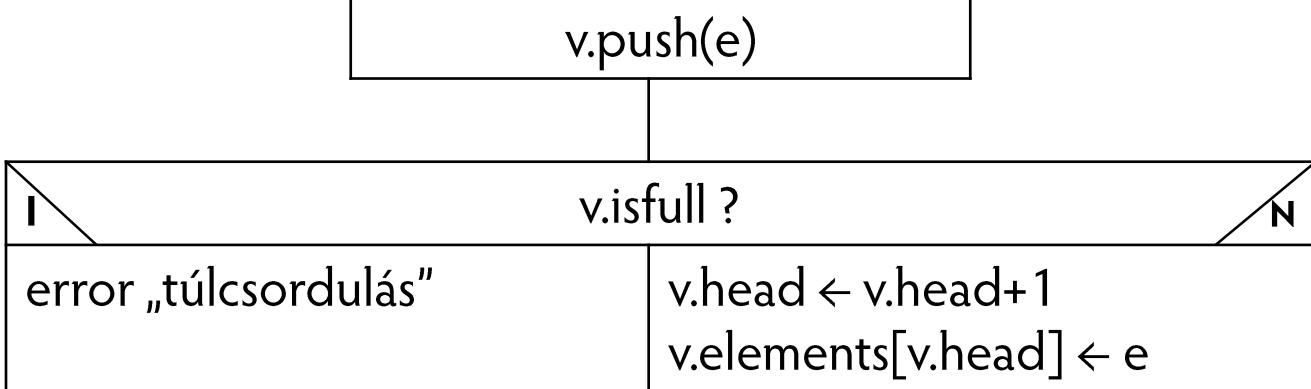
# Implementáció

- Műveletek pszeudokódja/struktogramja:
  - **v.empty**
    - üresre állítja a vermet
    - `v.head ← 0`
  - **visempty**
    - Üres a verem? - Logikai értéket ad vissza
    - `return (v.head=0)`
  - **v.isfull**
    - Tele van a verem? - Logikai értéket ad vissza
    - `return (v.head=max)`

# Implementáció

```
• v.push(e)
  -- e-t beteszi a v verem tetejére
if v.isfull
  then error „túlcsordulás”
else v.head ← v.head +1
  v.elements[v.head] ← e
end if
```

v.push(e)





# Implementáció

```
• v.pop
  -- kiveszi a legfelső elemet és visszaadja
if v.isempty
  then error „alulcsordulás”
else v.head ← v.head -1
  return v.elements[v.head+1]
end if
```

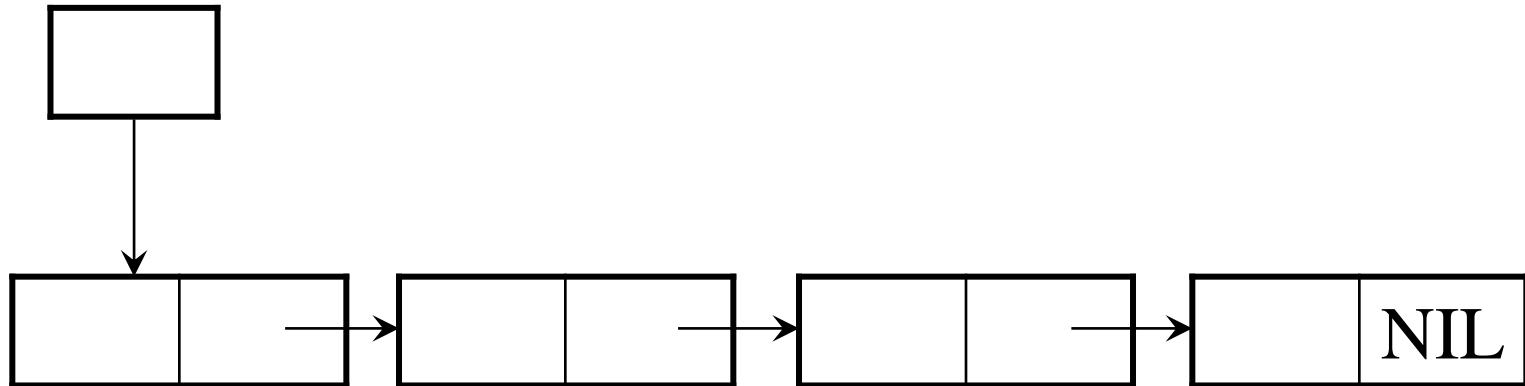


# Implementáció

```
• v.top
  -- lekérdezi a legfelső elemet
if v.isempty
  then error „alulcsordulás”
  else return v.elements[v.head]
end if
```

# Reprezentáció

- Lesz még láncolt ábrázolás is! (Gyakorlaton)





# Adatszerkezetek elméleti alapjai



# Típusok

## Kezdetben

- Egy adat típusán csak az adat által felvehető értékek halmazát értik.

## Ma

- A típus egy adat által felvehető lehetséges értékek halmazán kívül megadja az adaton értelmezett műveleteket is.



# A típus-absztrakció szintjei

- Absztrakt adattípus (ADT)
  - semmit nem feltételez a belső szerkezetről
  - enkapszuláció
- Absztrakt adatszerkezet (ADS)
  - Absztrakt szerkezet
  - irányított gráf mutatja a rákövetkezéseket:
    - csúcsok – adatelemek
    - élek – rákövetkezések



# A típus-absztrakció szintjei

- Reprezentáció
  - ADS gráf az absztrakt memóriában
  - Fontos, hogy a rákövetkezések megmaradjanak!
    - Láncolt vagy aritmetikai ábrázolás
- Implementáció
  - programnyelven
- Fizikai ábrázolás
  - az illúzió vége: bitek



# Típus-specifikáció

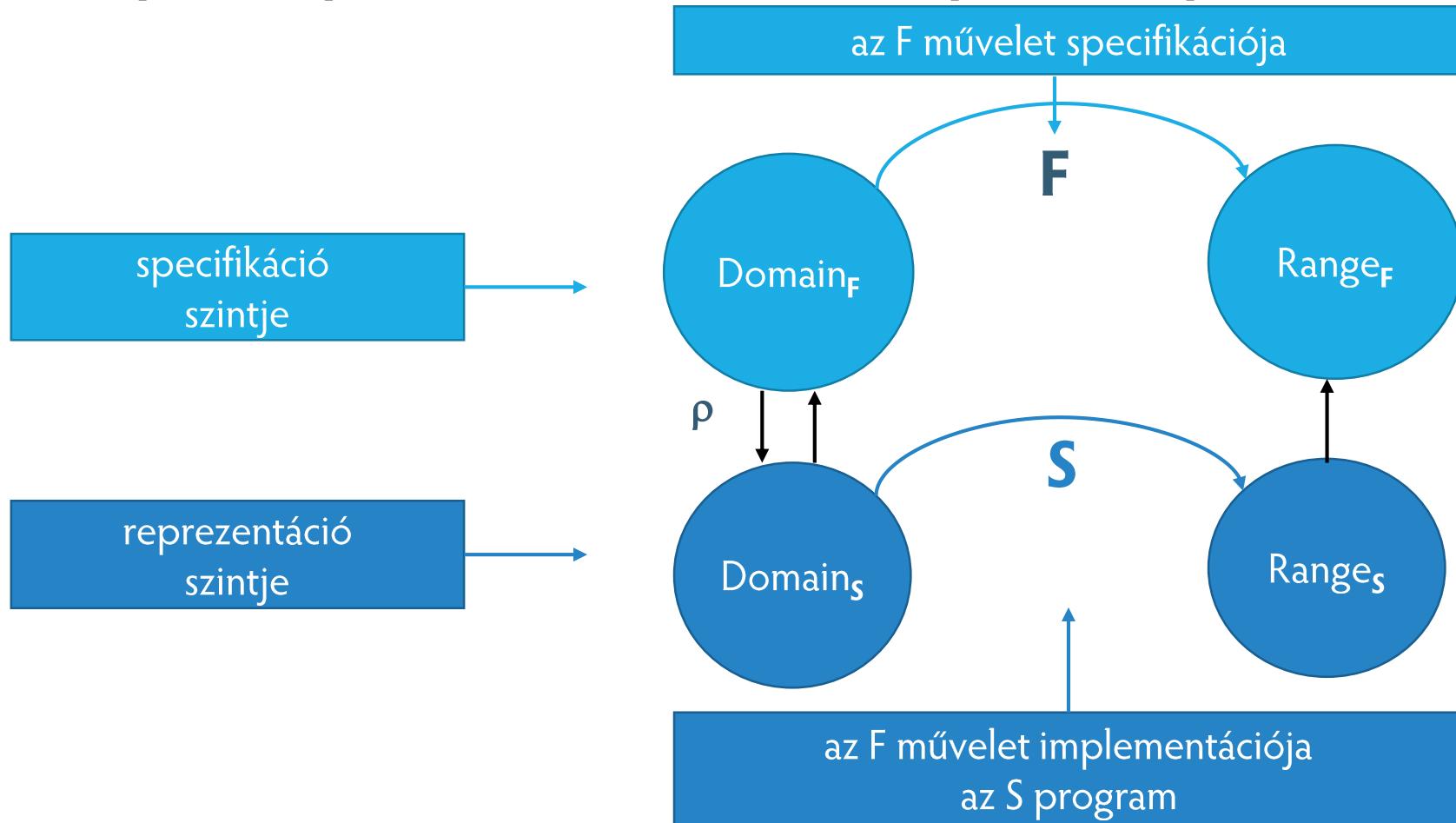
- Egy adat külső jellemzésére szolgál (interfész)
  - típusérték-halmaz
    - Az adat által felvehető értékek  $T$  halmaza.
  - típusműveletek
    - $T$ -n értelmezett feladatok.
- Önálló szerepet játszik a programtervezésben
- Megadására többféle lehetőség van:
  - algebrai specifikáció (axiómák megadásával)
  - funkcionális specifikáció (elő- és utófeltételekkel)



# Típus

- A típus-reprezentáció (típusértékek ábrázolása)
  - Ábrázoló elemek H halmaza. (típus-szerkezet)
  - Az ábrázoló elemek és a típusértékek kapcsolatát leíró leképezés:  
 $\rho : H \rightarrow T, \rho \subseteq H \times T$
  - A típus-invariáns kiválasztja a hasznos ábrázoló elemeket:  $I : H \rightarrow L, [I]$
- A típus-implementáció (műveletek helyettesítése)
  - Nem a típusértékekkel, hanem az azokat ábrázoló elemekkel működő programok

# A típus specifikáció és a típus kapcsolata





# Absztrakt adattípus

- A típus-specifikáció (közvetett) megadására szolgál
  - Nem szükséges, hogy egy konkrét programozási környezetben ábrázoljuk a típusértékeket.
  - Elég a műveletek programjainak csak a hatását ismerni.
- Absztrakt a programozási környezet számára és a megoldandó feladat számára.
- Szükség van egy őt kiváltó (konkrét) típusra.
  - Részfeladatokra bontás eszköze.



# Absztrakt adattípus

- A típus szemléletének ez a legmagasabb szintje
- Semmilyen feltételezéssel nem élünk a típus szerkezetéről, megvalósításáról!
- A specifikációban csak tisztán matematikai fogalmakat használhatunk
- Ez a szint nem a formalizálás mértékétől absztrakt; lehet informálisan is gondolkodni, beszélni ADT szinten!



# Az ADT algebrai specifikációja

- Részei:

- típusérték halmaz
- műveletek (mint leképezések)
- megszorítások (értelmezési tartományok)
- axiómák

- Kérdések:

- helyesség (ellentmondásmentesség)
- teljesség /nehéz
- redundancia /nem fontos



# Az ADT funkcionális specifikációja

- A típus matematikai reprezentációját használjuk
- Ez semmilyen módon nem kell, hogy utaljon a típus ábrázolási módjának megválasztására a megvalósítás során!
- Részei:
  - típusérték halmaz
  - műveletek
  - állapottér
  - paramétertér
  - előfeltétel
  - utófeltétel



# Algoritmusok ADT szinten

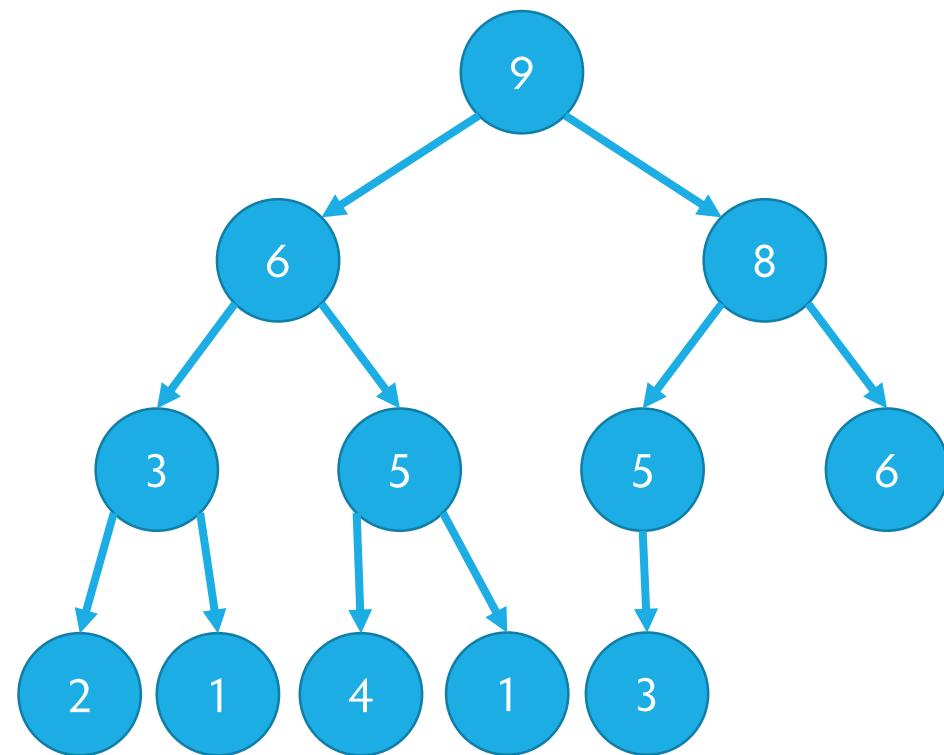
- A típus értékeit csak a típusműveletek segítségével lehet változtatni
- Azt nem tudjuk, hogy a műveletek hogyan működnek
- A típus „fekete doboz”
  - semmilyen megkötés nincs a szerkezetéről



# Absztrakt adatszerkezet (ADS)

- Széles körben használják az oktatásban, könyvekben
- A típus alapvető - absztrakt - szerkezetét egy irányított gráffal ábrázoljuk
  - A gráf jelentése:
  - csúcsok: adataelemek
  - élek: rákövetkezési reláció
- A műveletek ezen a szinten is jelen vannak (ha egy típus ADS-éről van szó)
- A műveletek hatása szemléltethető az ADS-gráf változásaival

# Példa



- Kupac

- A prioritásos sor szokásos reprezentációja
- bináris fa
- majdnem teljes
- balra tömörített
- a szülő csúcsokban nagyobb értékek vannak, mint a gyerek csúcs(ok)ban

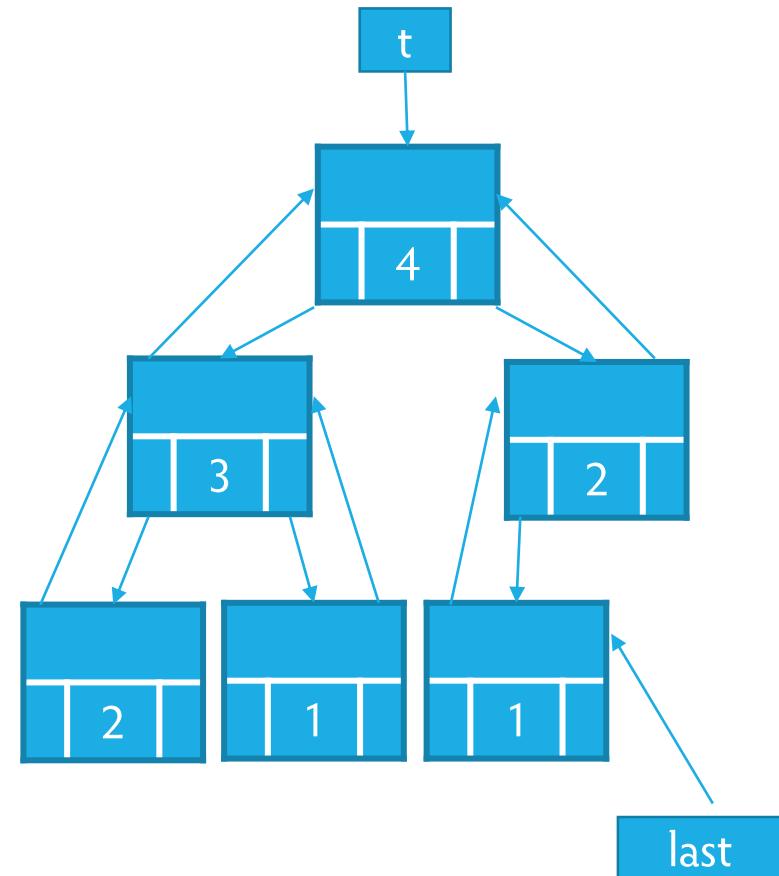


# Reprezentáció

- Absztrakt reprezentáció – absztrakt memóriában
- Az absztrakt szerkezet ábrázolható
  - pointerekkel (láncolt ábrázolás), vagy
  - cím-/indexfüggvényel (aritmetikai reprezentáció)
  - (vegyes ábrázolás lehetséges)
- Mindkét reprezentáció megadja az adatelemek közötti rákövetkezési relációt
- További rákövetkezések is megadhatók pointerekkel, illetve kiolvashatók az index-függvényből, mint amelyet az ADS leírt!

# Pointeres ábrázolás (láncolás)

- Példa: kupac láncolt ábrázolása
- Az ADS-gráf éleit pointerekkel ábrázoljuk
- A műveletek algoritmusait itt már meg kell adni
- A feladatban bevezetett függvények kiszámító algoritmusait is meg kell adni (szemben az ADS-sel)
- Következmény: egy feladat megoldása gazdagabb reprezentációt igényelhet, mint maga az ADS



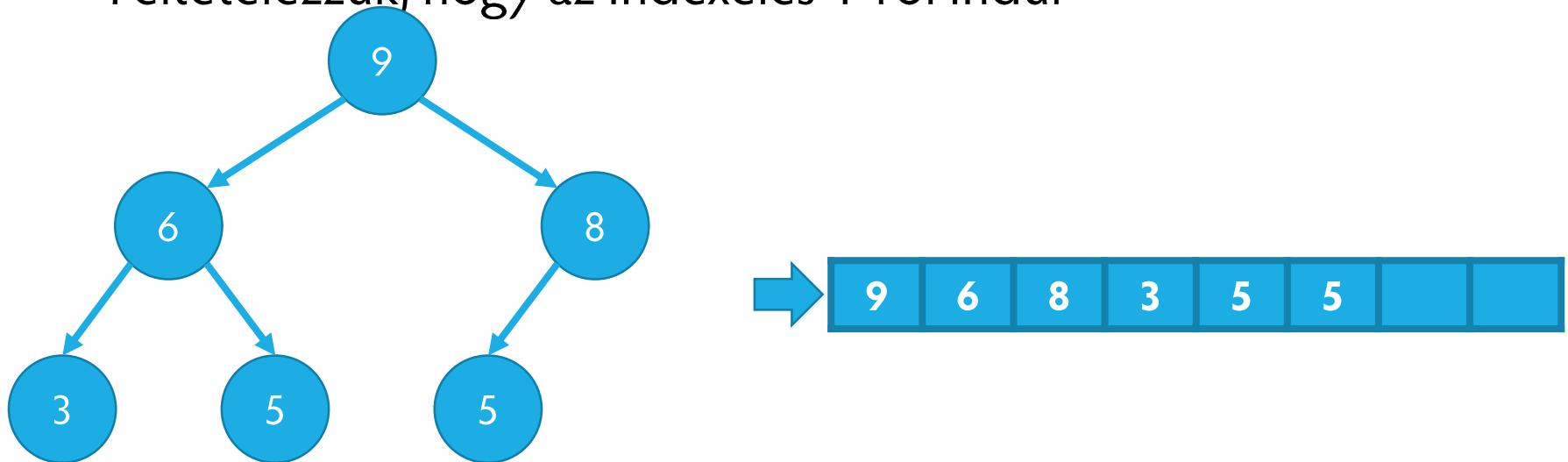


# Aritmetikai reprezentáció

- Index- és címfüggvények megadása
  - Az adatalemekeket folyamatosan elhelyezzük az absztrakt memóriában/ egy ugyanilyen alaptípusú vektorban
  - Az elemek közötti rákövetkezési relációt egy cím-/ indexfüggvénnyel adjuk meg
  - A címfüggvényből további rákövetkezések is kiolvashatók, nem csak az ADS-beli
  - A műveletek és a feladat-specifikus függvények algoritmusait meg kell adni

# Példa

- (Majdnem) teljes fa indexelése tömbben:
  - Szintfolytonosan:
    - $\text{index}(\text{bal}(a)) = 2 * \text{index}(a)$
    - $\text{index}(\text{jobb}(a)) = 2 * \text{index}(a) + 1$
  - Feltételezzük, hogy az indexelés 1-ről indul





# Ami még hiányzik

- Az implementáció szintje egy programnyelv, illetve fejlesztő környezet megválasztását jelenti
  - gyakorlatokon lesz
- A fizikai ábrázolás szintjén azt vizsgáljuk, hogy az adatszerkezet hogyan képeződik le a memória bájtjaira
  - ezzel ebben a tárgyban alapvetően nem foglalkozunk
  - kicsit lesz róla szó a gyakorlatokon



# Az adatszerkezetek osztályozása

- Az adatszerkezet egy  $\langle A, R \rangle$  rendezett pár, ahol
  - $A$  : az adataelemek véges halmaza
  - $R$  : az  $A$  halmazon értelmezett valamelyen reláció
  - $R \subseteq (A \times A)$



# Az adatszerkezetek osztályozása

- Az **adatelemek típusa** szerint
  - Homogén
    - Az adatszerkezet valamennyi eleme azonos típusú.
  - Heterogén
    - Az adatszerkezet elemei különböző típusúak lehetnek.

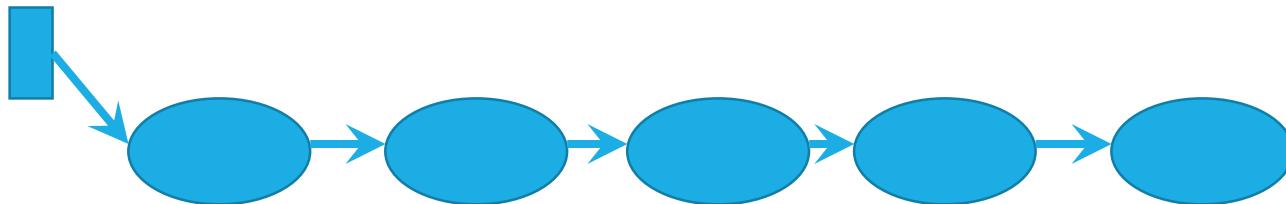


# Az adatszerkezetek osztályozása

- Az elemek közti ***R*** reláció szerint
  - Struktúra nélküli
    - Az egyes adatelemek között nincs kapcsolat. Nem beszélhetünk az elemek sorrendjéről (Pl. halmaz).
  - Asszociatív címzésű
    - Az adatelemek között lényegi kapcsolat nincs.
    - Az adatszerkezet elemei tartalmuk alapján címezhetők.
  - Szekvenciális
    - Szekvenciális adatszerkezetben az egyes adatelemek egymás után helyezkednek el.
    - Az adatok között egy-egy jellegű a kapcsolat: minden adatelem csak egy helyről érhető el és az adott elemről csak egy másik látható.
    - Két kitüntetett elem az első és az utolsó.

# Az adatszerkezetek osztályozása

- Szekvenciális
  - Intuitív ADT és ADS szint



Végigmehetünk az elemeken egymás után.  
Lehetőség van a módosítás, törlés, beszúrás műveletekre.

- Definíció: A szekvenciális adatszerkezet olyan  $\langle A, R \rangle$  rendezett pár amelynél az  $R \subseteq (A \times A)$  reláció tranzitív lezártja teljes rendezési reláció
- Az  $R \subseteq (A \times A)$  reláció tranzitív lezártja az a reláció, mely tranzitív, tartalmazza  $R$ -et, és a lehető legkevesebb elemet tartalmazza
- Megadása:
  1.  $R' = R \cup (R \circ R)$
  2. Ha  $R \neq R'$ , akkor folyt. 1.-nél,  
különben  $R' = R_T$ , a tranzitív lezárt

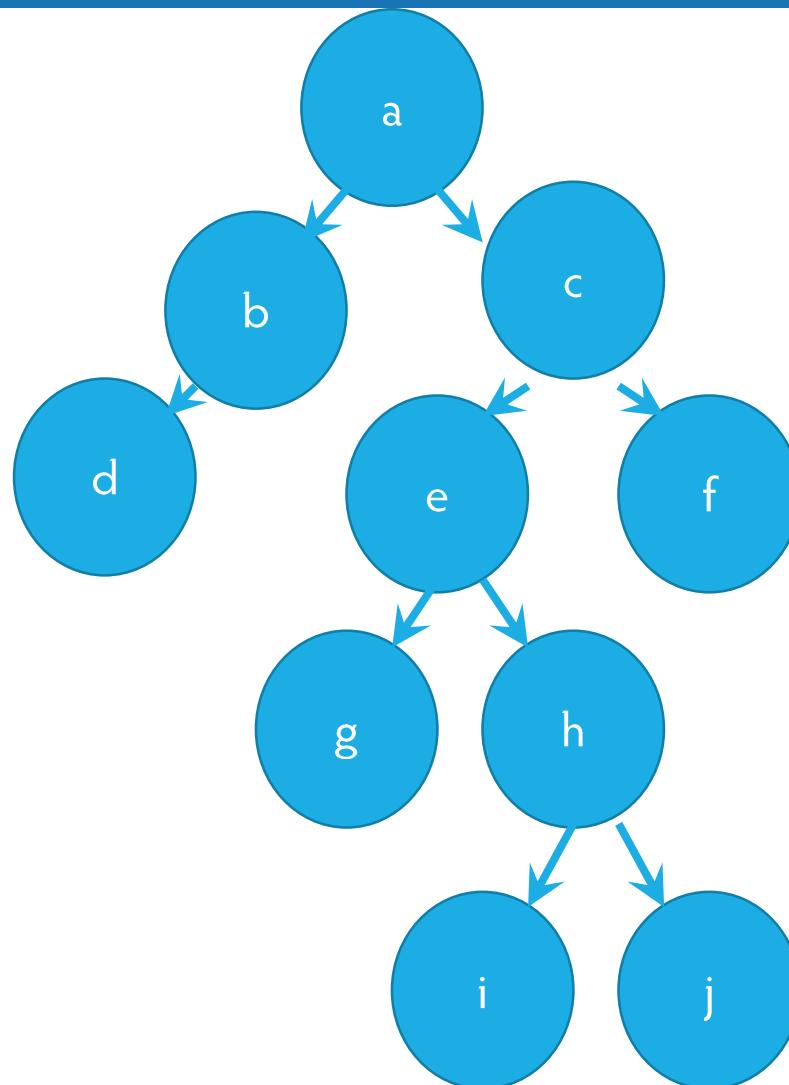


# Az adatszerkezetek osztályozása

- Hierarchikus

- A hierarchikus adatszerkezet olyan  $\langle A, R \rangle$  rendezett pár, amelynél van egy kitüntetett  $r$  elem, ez a gyökérelem, úgy, hogy:
  - $r$  nem lehet végpont
  - $\forall a \in A \setminus \{r\}$  elem egyszer és csak egyszer végpont
  - $\forall a \in A \setminus \{r\}$  elem  $r$ -ből elérhető
- Az adataelemek között egy-sok jellegű kapcsolat áll fenn.
- minden adataelem csak egy helyről érhető el, de egy adott elemből akárhány adataelem látható
  - Például fa, összetett lista, B-fa

# Bináris fa





# Az adatszerkezetek osztályozása

- Hálós

- A hálós adatszerkezet olyan  $\langle A, R \rangle$  rendezett pár, amelynél az  $R$  relációra semmilyen kikötés nincs
- Az adatalemek között a kapcsolat sok-sok jellegű: bármelyik adatalemhez több helyről is eljuthatunk, és bármelyik adatalemtől elvileg több irányban is mehetünk tovább
  - Például gráf, irányított gráf



# Az adatszerkezetek osztályozása

- Az **adatelemek száma** szerint
  - Statikus
    - Egy statikus adatszerkezetet rögzített számú adatelem alkot.
    - A feldolgozás folyamán az adatelemek csak értéküket változtathatják, de maga a szerkezet, az abban szereplő elemek száma változatlan
    - Következésképpen az adatszerkezetnek a memóriában elfoglalt helye változatlan a feldolgozás során
  - Dinamikus
    - Egy dinamikus adatszerkezetben az adatelemek száma egy adott pillanatban véges ugyan, de a feldolgozás során tetszőlegesen változhat
    - Dinamikus adatszerkezetek lehetnek rekurzív vagy nem-rekurzív, lineáris vagy nem-lineáris struktúrák



# Az adatszerkezetek osztályozása

- Az **adatelemek száma** szerint
  - Dinamikus
    - Egy adatszerkezet rekurzív, ha definíciója saját magára való hivatkozást tartalmaz.
      - Ha egyetlen ilyen hivatkozás van, akkor lineáris a struktúra, ha több, akkor nem-lineáris
    - A dinamikus adatszerkezetek feldolgozása során az adatelemek száma változik így egy-egy elemnek területet kell allokálnunk, illetve a lefoglalt területeket fel kell szabadítanunk
      - Ezzel felvetődik a tárolóhely újrahasznosításának problémája



# Az adatszerkezetek osztályozása

- Reprezentáció szerint
  - Az egyes adatszerkezetek tárolhatók
    - folytonosan
    - szétszórt módon
  - A leképezés és a műveletek megvalósítása annál egyszerűbb, minél jobban illeszkedik az adatszerkezet a tárolási szerkezetre.
    - Az asszociatív és a string szerkezetek nagyon jól tárolhatók folytonosan
    - A hierarchikus és hálós szerkezetek elsősorban szétszórt tárolással kezelhetők
    - De például a verem és a sor minden két módon tárolható hatékonyan



# Az adatszerkezetek osztályozása

- Reprezentáció szerint
  - Folytonos ábrázolású
    - A központi tárban a tárelemek egymás után helyezkednek el.
    - Az adattételek tárolási jellemzői (típus, ábrázolási forma, méret) azonosak.
    - Ismert az első elem címe, ehhez képest bármely elem címe számítható.
      - Legyen minden elem hossza  $H$  byte és jelölje  $\text{loc}(a1)$  az első adatelem címét.
      - Ekkor  $\text{loc}(aN) = \text{loc}(a1) + (N - 1) * H$
  - Szétszórt ábrázolású
    - A tárelemek véletlenszerűen helyezkednek el
    - Közöttük a kapcsolatot az teremti meg, hogy minden elem tartalmaz más elemek elhelyezkedésére vonatkozó információt
      - az elemek címét



Pázmány Péter Katolikus Egyetem  
Információs Technológiai és Bionikai Kar

# Sor, Lista, Vektor

Következő alkalommal