Hashtáblák

ADATSZERKEZETEK ÉS ALGORITMUSOK
11. GYAKORLAT

Motiváció

Nagyon sok adatban kell nagyon gyorsan keresni

O(log(n)) nem elég gyors

Háttértárat/memóriát használunk a processzoridő helyett

Konstans idejű

- Beszúrás
- Törlés
- Keresés

Az alapötlet

Az elem helyét *(indexét)* a kulcsából generáljuk

Az index független a többi értéktől, nincs összehasonlítás

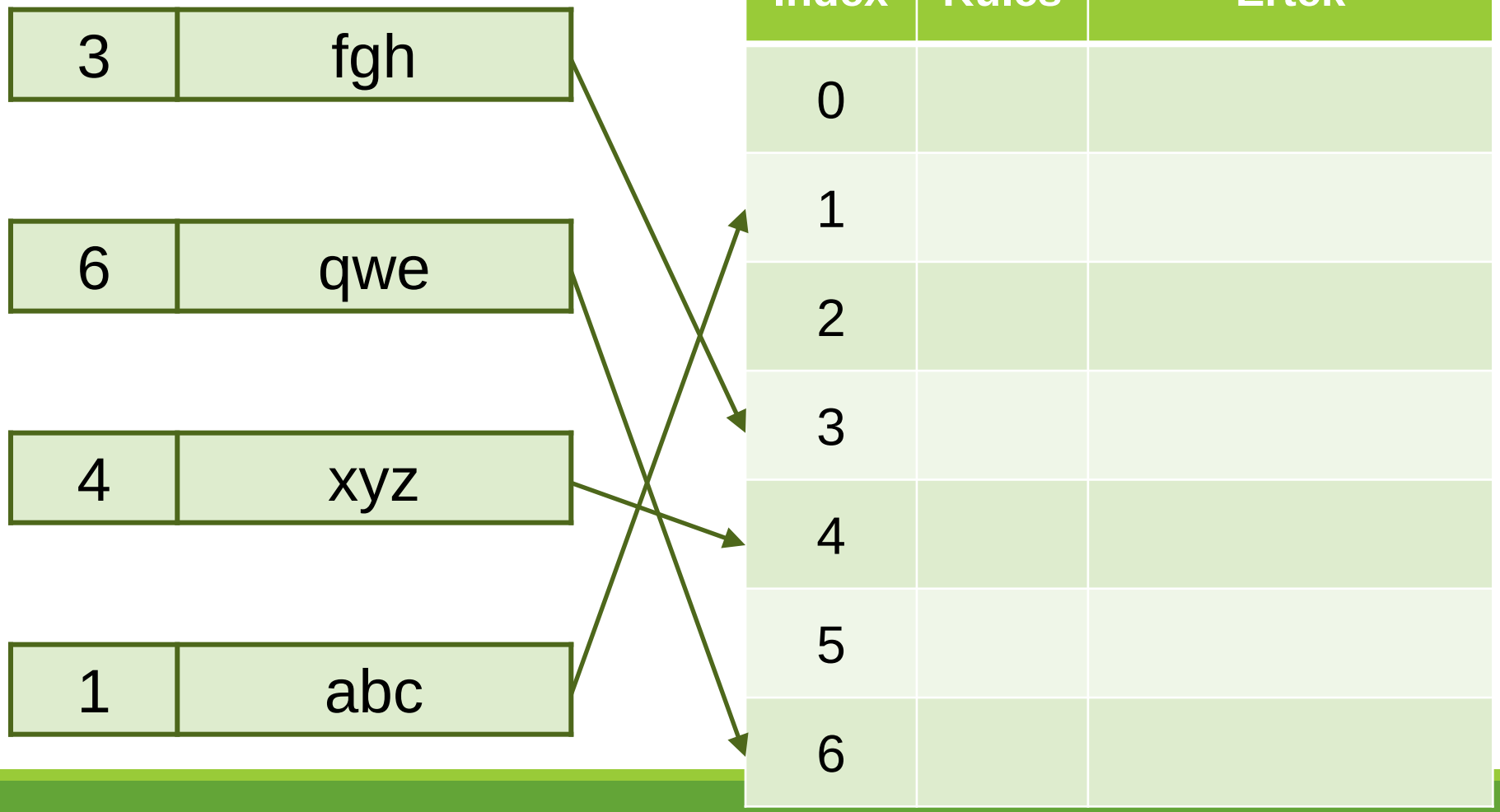
Konstans idő alatt generálható

Hogyan válasszunk indexet?

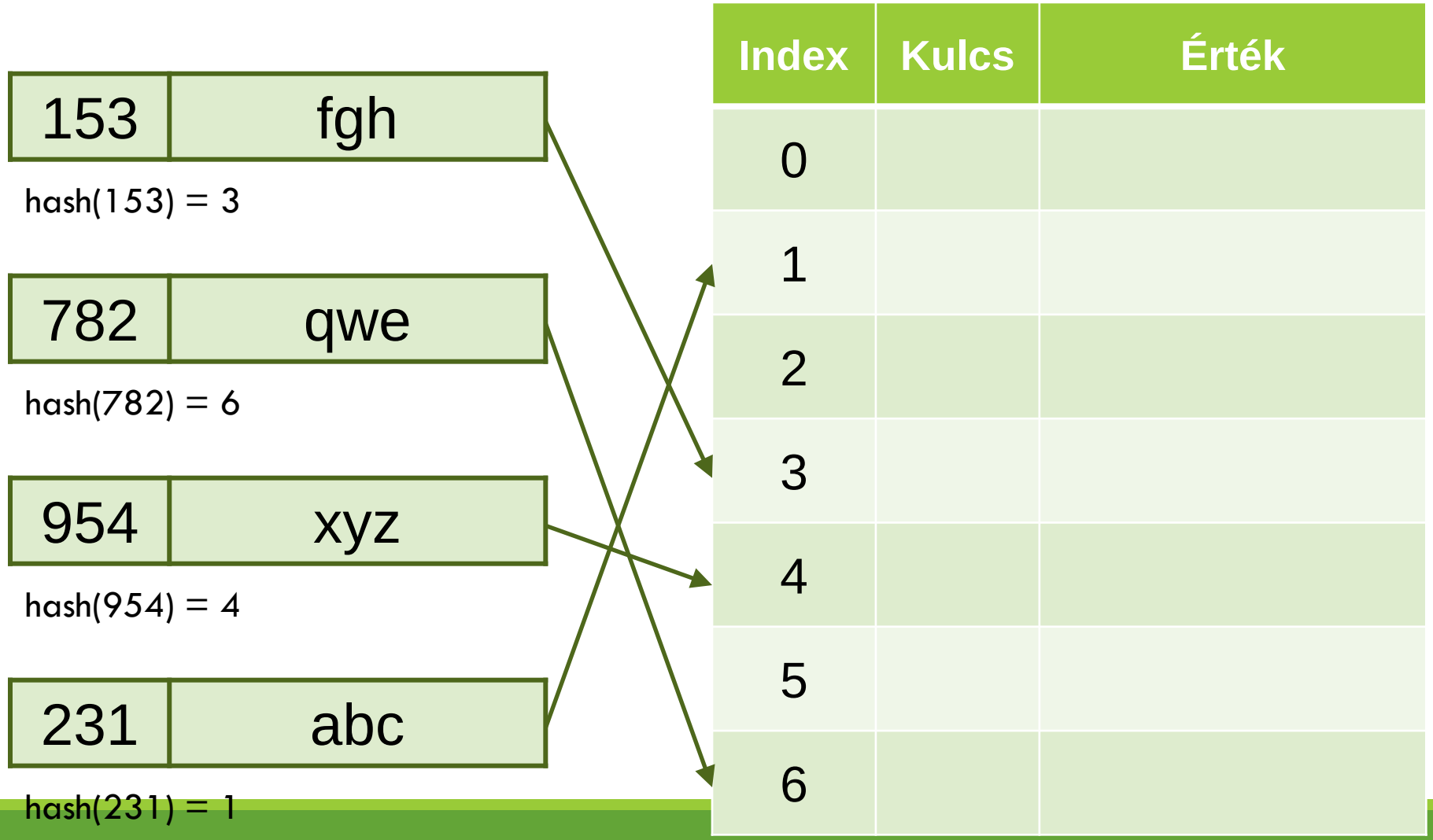
Ha a kulcs rövid intervallumon vesz fel értékeket (mindegyiknek tudunk helyet foglalni), akkor azt választjuk indexnek.

Ha az intervallum széles, akkor ez túl nagy méretű tömböt indexelne. Ekkor a kulcsot egy ún. *hashfüggvénnyel* szűkebb tartományra vetítjük.

Kulcs mint index



Szűkített (hashelt) index



A hashfüggvény

Kritériumok:

- Determinisztikus – ugyanazt a kulcsot mindig ugyanarra az indexre képezi le
- Egyenletes eloszlású kimenet – nem képez le sok elemet ugyanoda
- Konstans idejű

Egy egyszerű hashfüggvény

Kulcsok halmaza: 32 bites előjeles számok
(-2 147 483 648 … 2 147 483 647)

Indexek halmaza: 0…n-1, ahol n a hashtáblánk mérete

Modulo n

Kulcsütközések

Még a legjobb hashfüggvénynél is előfordulhat, hogy két különböző kulcsot azonos indexre képez le.

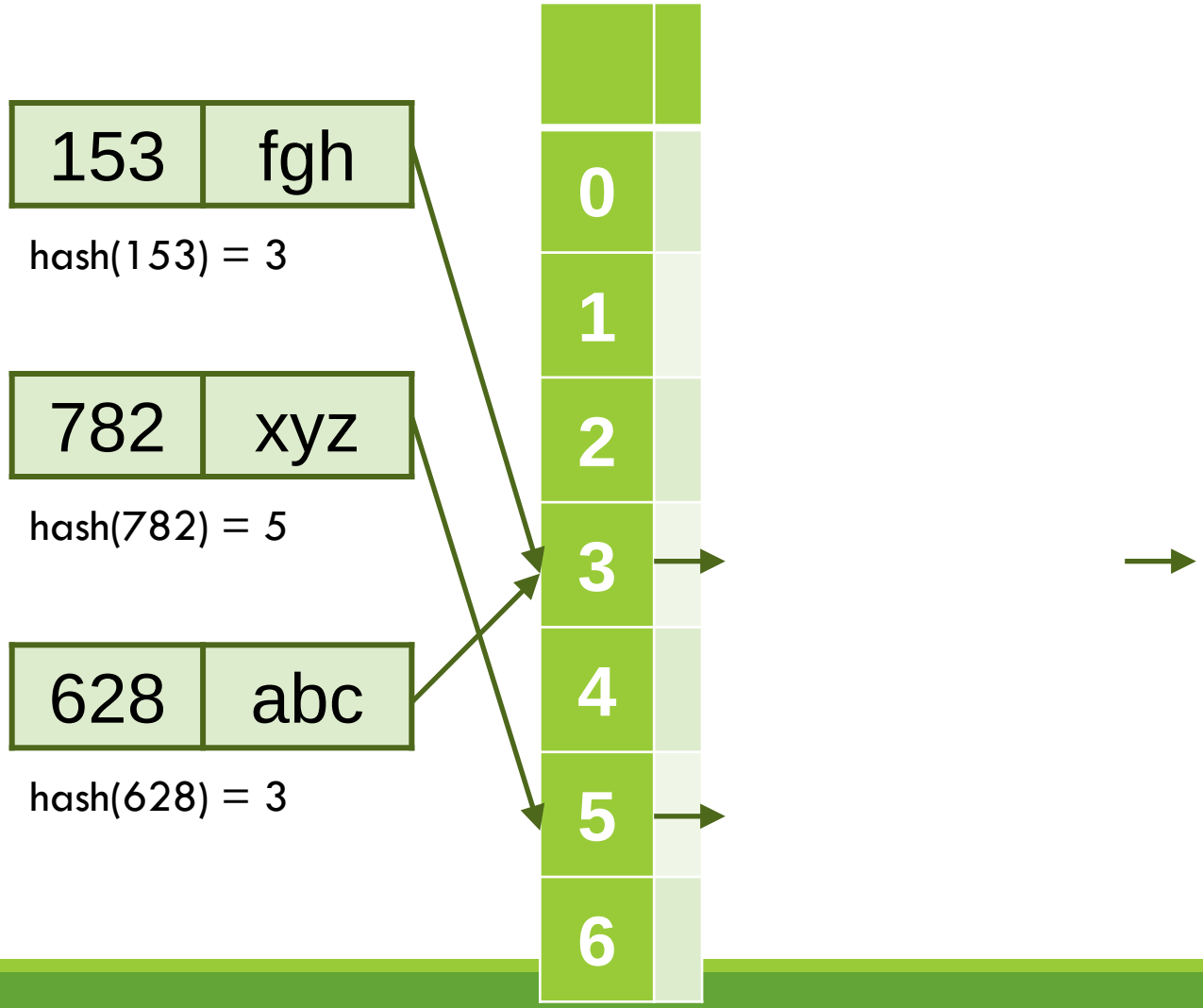
Ez elkerülhetetlen, mivel egy szélesebb intervallumról egy szűkebbre vetítünk.

Kulcsütközések feloldása

Sokféle módszer létezik. Néhány példa:

- **Láncolt lista:** A tábla mezői nem egyetlen elemet, hanem egy listát tartalmaznak.
- **Túlcsordulási terület:** A lista egy külön speciális területen található.
- **Újrahashelés (nyílt címzés):** Speciális hashfüggvényt használunk, ütközés esetén újrageneráljuk az indexet.

Listás ütközésfeloldás



Feladat

Írjátok meg a kiadott kódban a hiányzó részeket!

- Láncolt listás ütközésfeloldást használjatok!
- Használjátok az std::list-et!
- Kulcsok 32 bites int-ek

Kulcsütközések feloldása

Sokféle módszer létezik. Néhány példa:

- **Láncolt lista:** A tábla mezői nem egyetlen elemet, hanem egy listát tartalmaznak.
- **Túlcsordulási terület:** A lista egy külön speciális területen található.
- **Újrahashelés (nyílt címzés):** Speciális hashfüggvényt használunk, ütközés esetén újrageneráljuk az indexet.

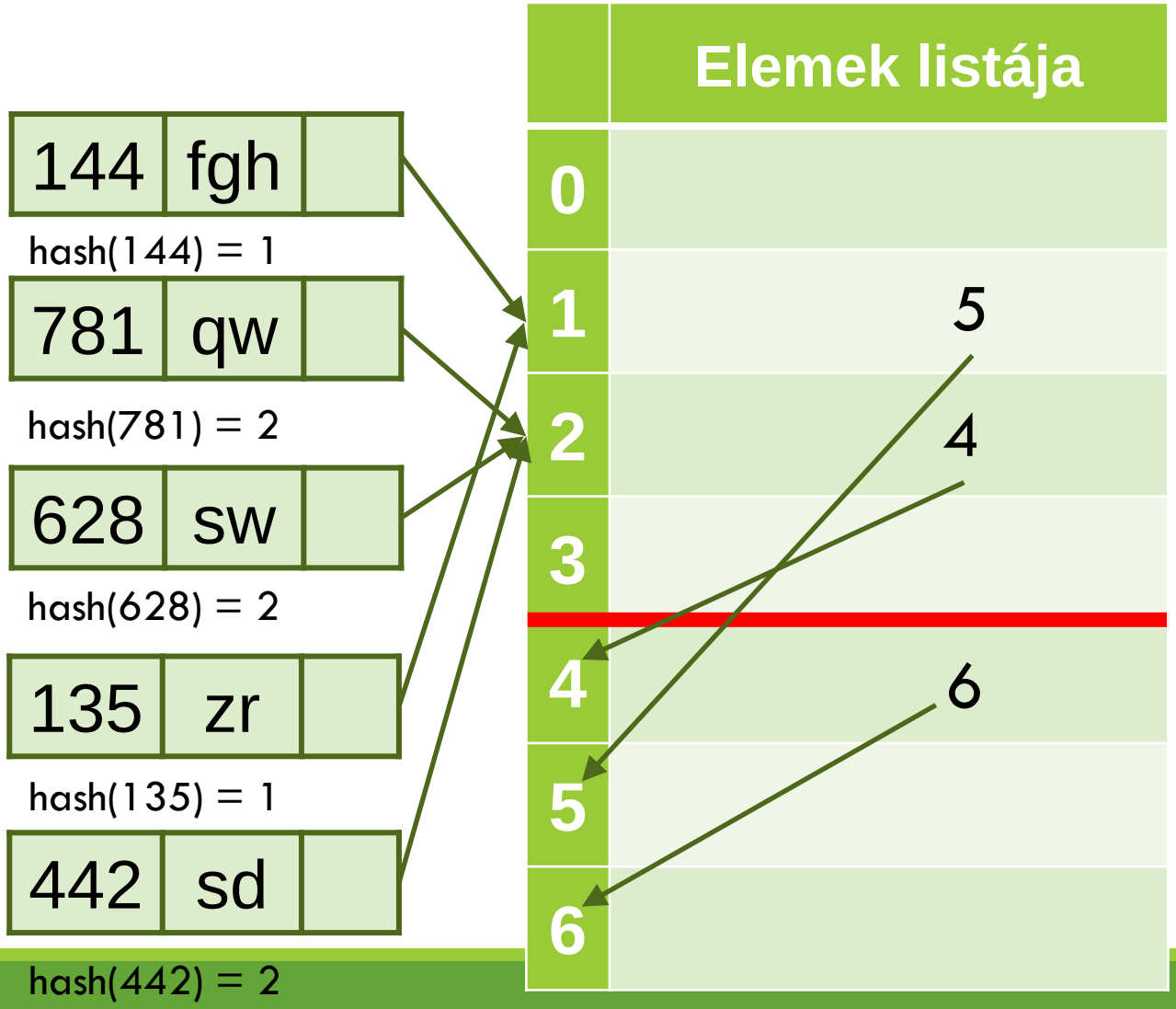
Túlcsordulási terület

A láncolt listát egy speciális területen hozzuk létre

A táblában nem elférő elemeket ide tároljuk le

Rámutatunk az előző listaelemből, ezzel létrehozva a láncolást

Túlcsordulási terület – beszúrás



Túlcsordulási terület – törlés

Ha a táblában található az elem

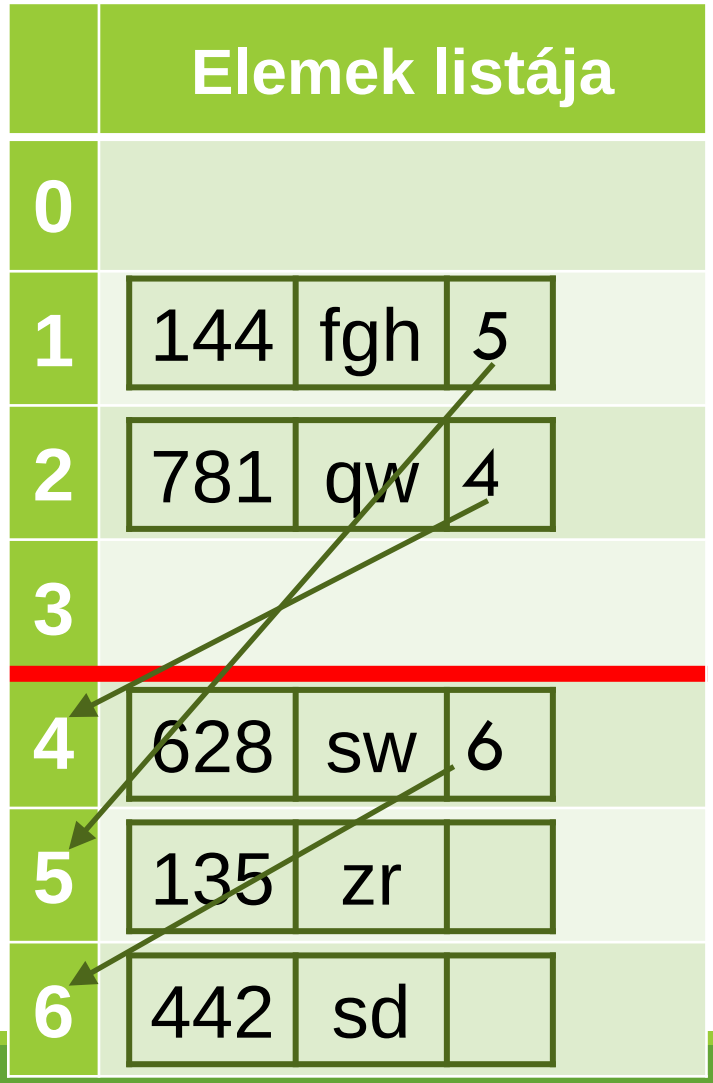
- Kitöröljük, és ha van ahhoz a hashértékhez elem a túlcsordulási területen, akkor onnan a lista első elemét feltesszük a táblába

Ha a túlcsordulási területen található az elem

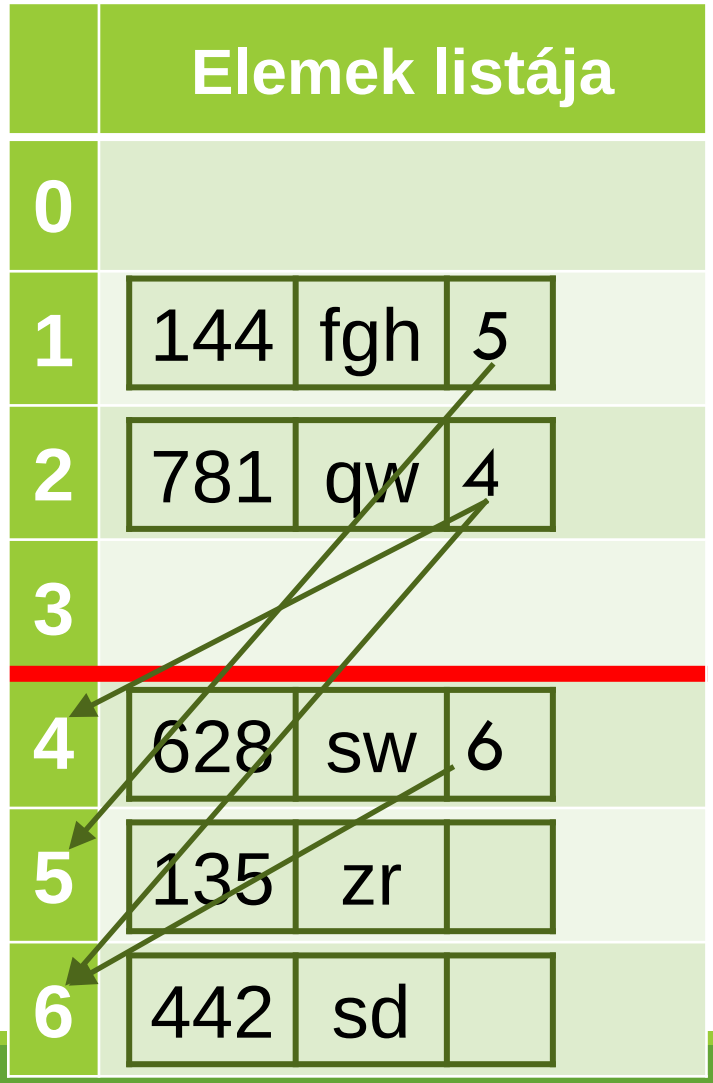
- Kitöröljük és átláncoljuk

Ez nem teljesen ugyanaz, mint ami az előadáson szerepelt!

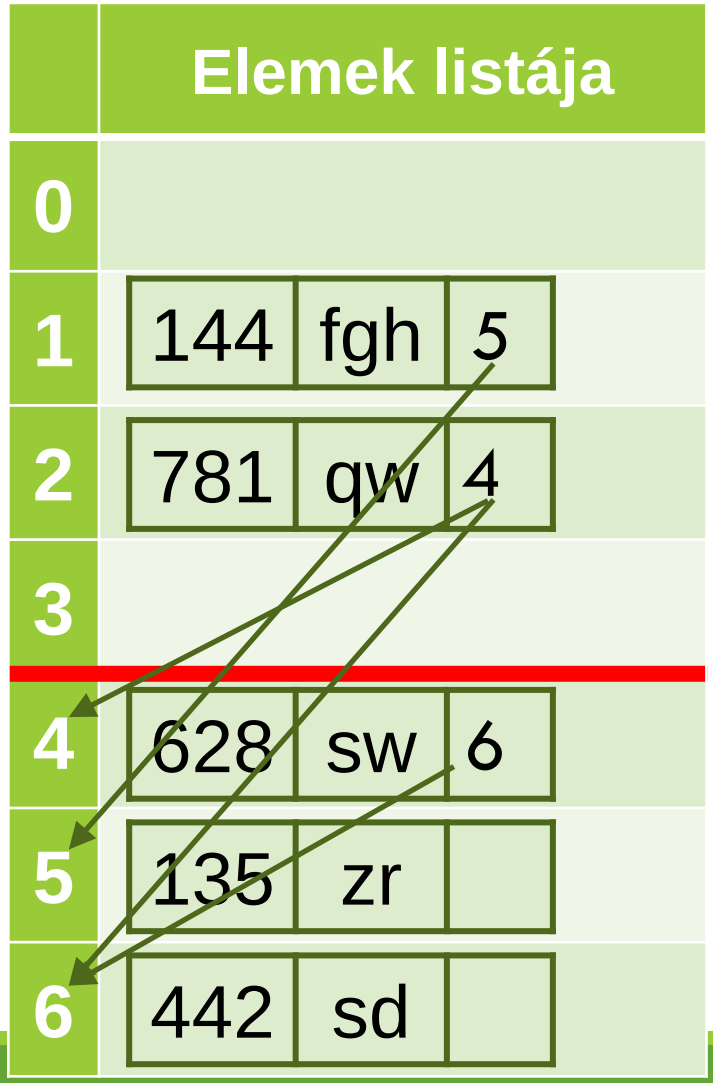
Túlcsordulási terület – törlés



Túlcsordulási terület – törlés



Túlcsordulási terület – törlés



Feladat

Nézzétek meg a kiadott kód működését!

- A kód túlcsordulási területes ütközésfeloldást használ
- Kulcsok 32 bites int-ek

Ujjlenyomat-készítés

Mi van, ha nem csak int típusú kulcsaink vannak?

Más-más hashfüggvény minden típushoz?

Hogy ne kelljen minden típusra külön hasító függvényt írni, a hasító függvényt két függvényre bontjuk:

- Ujjlenyomat-képező függvény
- Hasító függvény

Ujjlenyomat-készítés

Minden típusnak saját ujjlenyomat-készítő függvény, ez ugyanolyan ujjlenyomatot állít elő minden típusra

A hashfüggvény az immár egységes ujjlenyomatból fog indexet készíteni

Az ujjlenyomat méretét általában érdemes 32 bites egésznek (int) választani

- 32 bit a legtöbb esetben elég, ezt könnyű kezelni
- Ha esetleg ez nem elég, akkor 64 bites (long long)
  - Nagyon nagy tárhelyigény (32 bittel 16 GB-os méret még kezelhető, 64 bitnél 32 GB-ról indulunk)

Ujjlenyomat, `sizeof(T) <= sizeof(int)`

sizeof(T): a T típus ábrázolásához szükséges memóriaterület

Egész típusok (char, short, int, enum, …)
– (érték szerinti) típuskonverzió
- **`x` → `int(x)`, `static_cast<int>(x)`**
- ‘k’ (char) → 107 (int)
- 1337 (short) → 1337 (int)

Egyéb típusok
– bitminta újraértelmezése
- 3.14 (float) → **`4048F5C3`** → 1078523331 (int)

Ujjlenyomat, `sizeof(T) > sizeof(int)`

String

- Egyszerű megoldások
  - Karakterek összege mod 2^{32}
  - Utolsó 4 karakter intként értelmezve

Ujjlenyomat, `sizeof(T) > sizeof(int)`

Vegyük észre, hogy a stringeket értelmezhetjük úgy, mint egy nagy egész számot.

- Pl. egy 12 bájt hosszú string tekinthető egy 96 bites egész számnak.

Másik értelmezés: egy string tekinthető egy 256-os számrendszerben felírt számnak.

$$'abc' = 'a' \cdot 256^2 + 'b' \cdot 256 + 'c'$$

$$= 97 \cdot 256^2 + 98 \cdot 256 + 99 = 6382179$$

Ujjlenyomat, `sizeof(T) > sizeof(int)`

Általánosítás: egy string tekinthető egy *q*-s számrendszerben felírt számnak.

$$'abc' = 'a' \cdot q^2 + 'b' \cdot q + 'c'$$

Hasító függvény: tekintsük a stringet számnak, és határozzuk meg a *p* prímszám szerinti maradékát.

Polinom forma: hatékonyan kiértékelhető Horner sémával. Pszeudokód:

```
h = 0

a string minden c karakterére

 h = ( q * h + c ) modulo p
```

Ujjlenyomat, `sizeof(T) > sizeof(int)`

Paraméterválasztás: a pszeudokód akkor működik jól, ha kiértékelés során nem történik túlcsordulás.

Ezért a p és q értékét okosan kell megválasztani.

String esetén q = 256 = 2^8

Ahhoz, hogy q-val szorozva és c-t hozzáadva ne legyen túlcsordulás, h max 2^{24}-1 lehet

Ennél kisebb prímszám a 2^{24}-3=16 777 213

```
h = 0

a string minden c karakterére

 h = ( q * h + c ) modulo p
```

Egy kicsit bonyolultabb ujjlenyomat

SHA1

- 40 karakter (20 byte, 160 bit)
- Pl. „PPKE-ITK” = dc4e2182030b2597ec98894268559bf9b53d8aac
- Mennyire sok ez?
  - Jelenleg nem ismert két azonos hasht elérő bemenet
  - De ez nem jó nekünk. Miért?
  - Kicsit lassabb számolni – itt nem a teljesítmény a lényeg, hanem hogy nehéz legyen kulcsütközést elérni
  - Ha ekkora hashtáblát akarunk előállítani, akkor 2^160 sorra lenne szükségünk
  - Ez mennyi?
    - Atomok száma a világegyetemben: ~2^80
    - Minden atom tartalmazná a világegyetemet, és annak minden atomja egy értéket, akkor tudnánk eltárolni
  - De vehetjük mondjuk az utolsó 8 karakterét számként (32 bit), már ekkor is nagyon ritka lesz az ütközés

Hasító függvények

Most konkrétan azokra a függvényekre gondolunk, amelyek az ujjlenyomatot képezik le a tábla méretének megfelelő intervallumra.

Osztó módszer: **modulo** *m*

- Jól működik, ha m prímszám.
- Tehát a módszer használható, ha garantálni tudjuk, hogy a tábla mérete prímszám.

Feladat

Alakítsátok át a láncolt listás kódot úgy, hogy többféle típusra is működjön!

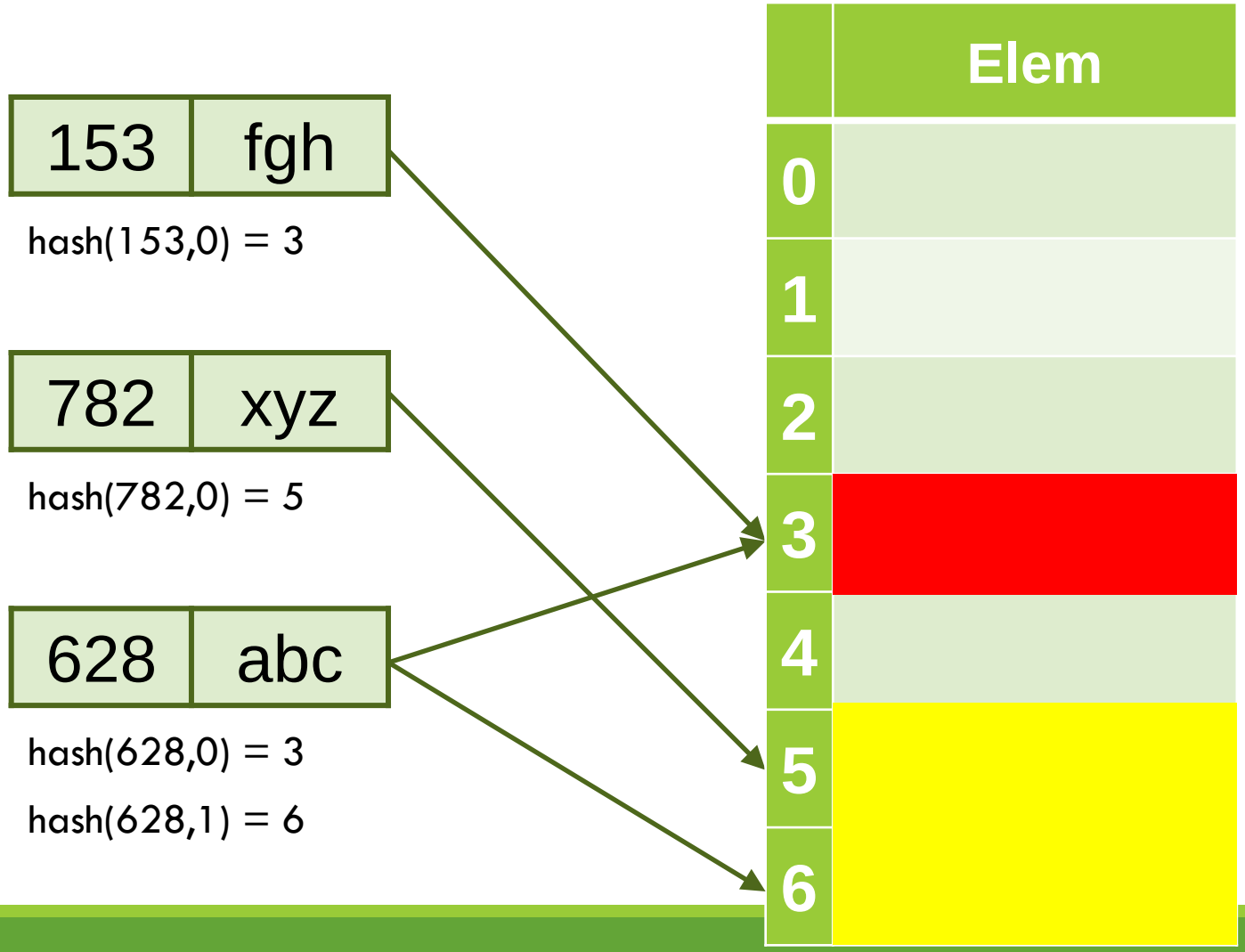
- int
- string
- telefonszám (pl. "3618864700")

Kulcsütközések feloldása

Sokféle módszer létezik. Néhány példa:

- **Láncolt lista:** A tábla mezői nem egyetlen elemet, hanem egy listát tartalmaznak.
- **Túlcsordulási terület:** A lista egy külön speciális területen található.
- **Újrahashelés (nyílt címzés):** Speciális hashfüggvényt használunk, ütközés esetén újrageneráljuk az indexet.

Újrahasheléses ütközésfeloldás



Újrahashelés – beszúrás

Az ötlet az, hogy a hashfüggvény egy kulcshoz ne csak egy indexet rendeljen, hanem egy indexsorozatot.

Ehhez kiegészítjük a függvényt még egy paraméterrel, amely kiválaszt a sorozatból egy konkrét indexet.

Például ha **k** egy kulcs, és a **hash(k,0)** egy foglalt hely indexét adja vissza, akkor a **hash(k,1)**-gyel próbálkozunk. Ha ez is foglalt, akkor **hash(k,2)** jön, és így tovább.

Újrahashelés – keresés

Korábban, az elem beszúrásakor lehet, hogy több helyet is kipróbáltunk, mielőtt betettük volna a táblába.

Ezért kereséskor is addig kell próbálkoznunk az indexekkel, amíg vagy meg nem találjuk a keresett elemet, vagy üres helyre nem érünk.

Tehát ha a **hash(k,0)** által kijelölt helyen nem egy k kulcsú elem van, akkor meg kell nézni a **hash(k,1)**-edik helyet is, ami esetleg szintén hamis találat, és így tovább.

Az is előfordulhat, hogy egyszer csak egy üres táblaelemet találunk, ami azt jelenti, hogy a keresett elem nincs a táblában.

Újrahashelés – törlés

Mi történik, ha a „hagyományos” módszerrel törlünk?

Keressük a 458-as indexű elemet, de előtte töröltük a 785-öset.

A kereső algoritmus megáll a 785-ös helyénél!

Iteráció	Index	Elem	
0	5	128	abc
1	10	854	xyz
2	3	~~785~~	~~mno~~
3	2	544	bcd
4	9	276	fgh
5	13	**458**	**qwe**

Újrahashelés – törlés

A megoldás:
törléskor egy „törölt” bejegyzést helyezünk el a táblában.

Ez megváltoztatja a beszúrást is: nem csak üres helyre, hanem „törölt” jelzésűre is tehetünk új elemet.

A kereső algoritmus tudni fogja, hogy a „törölt” bejegyzésen nem kell megállnia.

Iteráció	Index	Elem	
0	5	128	abc
1	10	854	xyz
2	3	[törölt elem]	
3	2	544	bcd
4	9	276	fgh
5	13	**458**	**qwe**

Újrahashelés – hashfüggvény

Újrahashelésnél a hashfüggvény tulajdonságainak a 2. paraméter növelésekor is teljesülniük kell!

Azaz a **(hash(k,0),hash(k,1),...)** sorozat is:

- pszeudorandom
- korlátos intervallumon vesz fel értékeket
  - ugyanazon, amin **(hash(a,0),hash(b,0),...)**
- az intervallumon belül minden értéket felvesz
- minden értéket egyforma valószínűséggel vesz fel

Ez azért fontos, hogy az újrapróbálkozásoknál legvégső esetben minden mezőt kipróbáljon.

Újrahashelés – hashfüggvény

Szokásos módszerek újrahashelésre:

- Lineáris kipróbálás

$$h'(k,i) = \left(h(k) + i\right) \bmod m$$

- Négyzetes kipróbálás

$$h'(k,i) = \left(h(k) + c_1 \cdot i + c_1 \cdot i^2\right) \bmod m$$

- Kettős hashelés

$$h'(k,i) = \left(h_1(k) + i \cdot h_2(k)\right) \bmod m$$

Részletek az előadáson voltak.

Feladat

Alakítsd át a hasító tábla osztályt, hogy nyílt címzést használjon, lineáris kipróbálással!

Hashtáblák az STL-ben

C++11 óta:

- `std::unordered_set`
- `std::unordered_multiset`
- `std::unordered_map`
- `std::unordered_multimap`
- std::hash

STL – ami még hasznos lehet

Sorbarendezés:

```
std::sort(numbers.begin(), numbers.end()) ;
```

- Nem stabil, C++11 óta garantált O(n*log(n))

```
std::stable_sort(numbers.begin(), numbers.end()) ;
```

- Stabil, garantált O(n*log^2(n)), elég memória esetén O(n*log(n))

Komparátorral:

- `std::sort(numbers.begin(), numbers.end(), comp) ;`
- `std::stable_sort(numbers.begin(), numbers.end(), comp) ;`

C++11 óta szintaktikai lehetőségek:

```
std::vector<int> numbers = { 3, 4, 5 } ;

for (int num : numbers) {

}
```

Feladat

Hasonlítsd össze az std::set, az std::unordered_set és a saját hashtáblánk futási idejét!

Hogyan lehetne gyorsítani?

Amit még nézzetek át a ZH-ra

STL-es adatszerkezetek:

- Lesz még szó róluk előadáson. Használjátok nyugodtan őket a zh-ban is, sokszor gyorsabb megírni így, mintha a saját megoldásainkat használnánk.

Futási idők:

- Tudjátok, hogy mi mennyi idő alatt fut le (O szempontból; pl. beszúrás, törlés, rendezés így/úgy/amúgy, iterálás tömbön/fán/hashtáblán), hogy minél jobb megoldást tudjatok adni a feladatokban.
- Gyakran, ha meg van mondva, akkor segítség a futási idő abban, hogy milyen megoldást érdemes választani (lásd pl. a gyakorló feladatokat vagy a tavalyi zh-t).

Gyakorló feladat

Készíts programot, ami egy txt-ben szereplő könyv szavait beolvassa, és azokból gyakoriságot számol. Az írásjeleket vegyétek ki az egyes szavakból, illetve a kis- és nagybetűk között ne legyen különbség. A program beolvasás (O(n)) után tudja a következő funkciókat:

- Megadja, hogy hány különböző szó található a könyvben. O(1)
- Megadja a leggyakrabban használt szavakat. O(k)
- Egy adott szóra rákeresve megadja, hogy az hányszor szerepel. (O(1))

Ahol a hatékonyságnál n az összes szó száma a szövegben, k pedig a különböző szavak száma.

A teszteléshez txt fájlokat pl. innen állíthattok elő:

- http://mek.oszk.hu/00600/00656/html/

Gyakorló feladat

Az 1. közepes háziban főleg saját adatszerkezeteket használtatok a feladat megoldására.

Írjátok át ezeket STL-es verziójukra!

Adjátok meg minden parancsra, hogy annak mekkora a futási ideje (O(?))!

Gyorsítsátok fel a feladatot az azóta megtanult algoritmusok és adatszerkezetek használatával!

Gyakorló feladat

B-fa: bár idén gyakorlaton nem szerepelt, gyakorlásnak, illetve haladóbbaknak mindenképp ajánlott lehet ezeket átnézni, megírni.

Ehhez segítségként elérhető anyagok:

- Korábbi diasor: https://wiki.itk.ppke.hu/twiki/pub/PPKE/AdatAlg201314/09_btree.pdf
- Korábbi órai kód: https://wiki.itk.ppke.hu/twiki/pub/PPKE/AdatAlg201314/09_btree.zip

Egészítsétek ki az órai kódot, nézzétek át, hogy hogyan működik.

Az ezzel és az önálló adatszerkezet-írással szerzett tapasztalat hasznos lehet pl. későbbi állásinterjúk esetén is.

Házi feladat

Csináljátok meg a tavalyi 2. ZH második feladatát!

Wikin elérhető:
https://wiki.itk.ppke.hu/twiki/pub/PPKE/AdatAlg201415/ZH2feladat.pdf

A következő dián megtalálhatók az osztály, aminek a függvényeit ki kell egészíteni.

Keretrendszer most nincs; írjatok egy főprogramot, ami ennek az osztálynak a metódusait meghívja és értelmes módon teszteli, a szükséges adatokat pedig a következő fájlokból olvassa be:

- https://users.itk.ppke.hu/~hunma/adatszerk/minorplanet.txt
- https://users.itk.ppke.hu/~hunma/adatszerk/hu_cities_en.txt

(A beugró feladatot is javasolt gyakorolni, de azt házi feladatként nem fogjuk pontozni.)

Házi feladat: a megírandó fájl váza

```
class ZHStudentSolution {
public:
	ZHStudentSolution() {};
	virtual ~ZHStudentSolution() {};
	virtual DayOfYear task1(const std::vector<MinorPlanet>& data);
	virtual void prepare2_3_4(const std::vector<MinorPlanet>& data);
	virtual std::string task2();
	virtual std::vector<std::string> task3();
	virtual std::vector<std::string> task4(std::string place);
	virtual std::vector<std::string> task5(const
std::vector<MinorPlanet>& data, const std::vector<std::string>&
cityData);
	virtual std::vector<int> task6(const std::vector<MinorPlanet>&
data);
	virtual std::vector<int> task7(const std::vector<MinorPlanet>&
data);

};
```