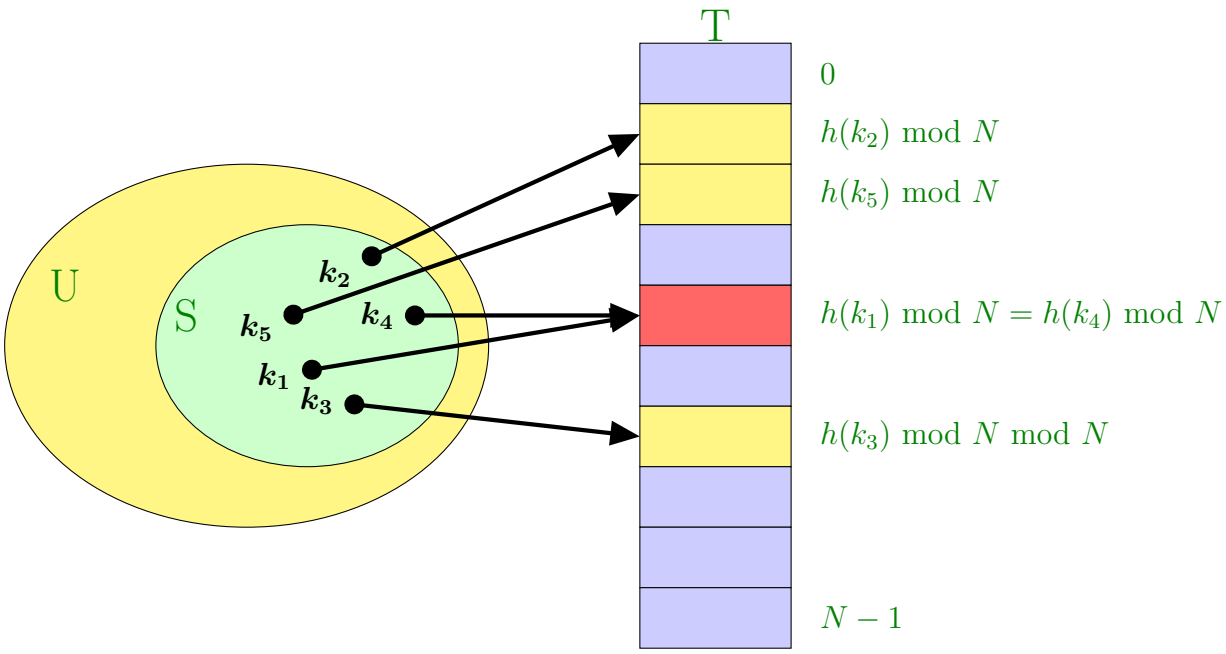
# Algoritmusok és adatszerkezetek gyakorlat – 08

## Hash táblák

### Hash táblák

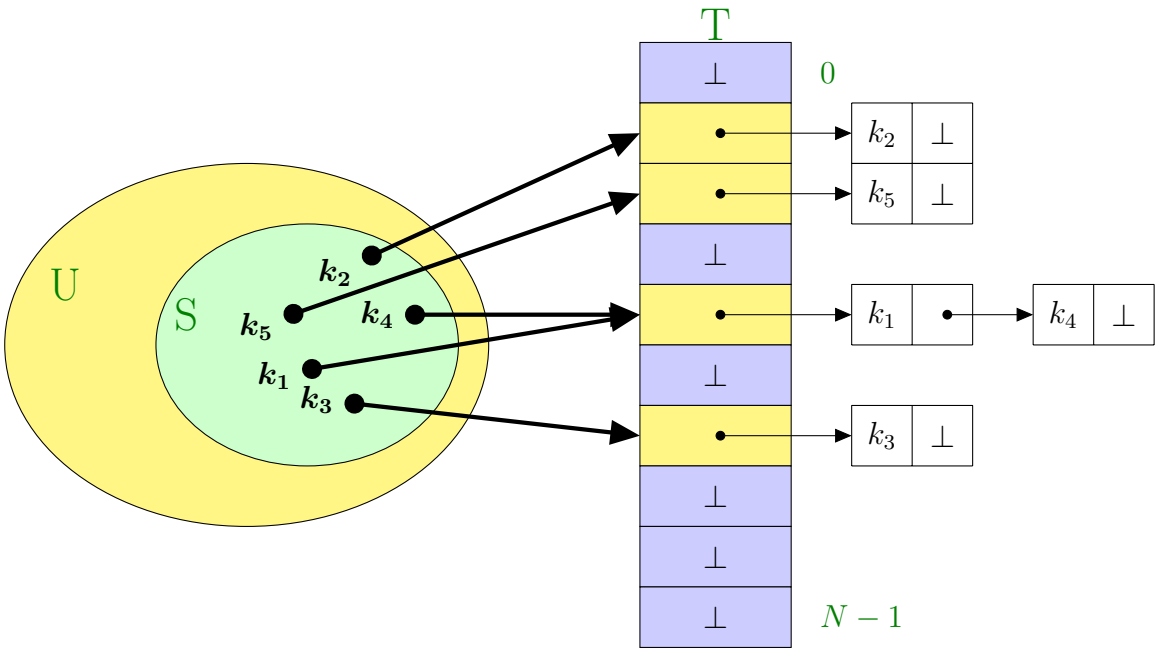
- **Cél:** értékek egy halmazát eltárolni úgy, hogy $O(1)$ időben támogatjuk a következő műveleteket:
  - insert($i$): hozzáad egy $i$ értéket a halmazhoz
  - delete($i$): törli az $i$ értéket a halmazból
  - contains($i$): megadja, hogy az $i$ érték benne van-e a halmazban
- **Hash függvény:** egy objektumhoz rendeljünk egy intet úgy, hogy különböző objektumoknak általában (jó eséllyel) különböző legyen a hash kódja. Elvárás, hogy gyorsan ($O(1)$ időben) számítható legyen
- **Hash tábla:** egy tömb. Ha $N$ elemű, akkor a kulcsok halmaza $0, \ldots, N-1$. Mérete futás közben változtatható
- Legegyszerűbb módszer a kulcsok $[0, N-1]$ halmazban tartására: a hash függvény készít egy intet és ezt mod $N$ vesszük.



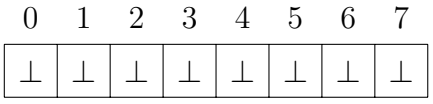

- Amikor egy elemet egy már foglalt cellába kellene rakjunk a hash kódja alapján, ütközésről beszélünk. Ezeket valahogyan fel kell oldjuk.

## Chaining

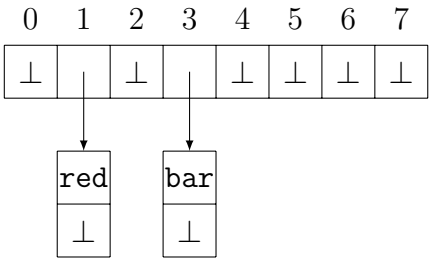
- A tömb egy cellájában egy pointer van egy láncolt listára, ami az oda eső kulcsokat tárolja



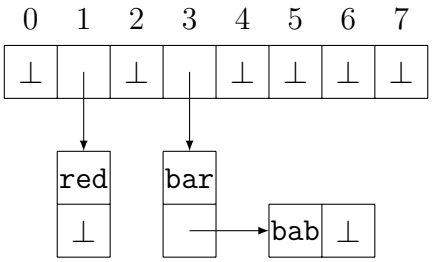

- Példa (a hash kód a Java String osztályának `hashCode()` metódusával készült):




- `insert("bar")` (hash: `"bar".hashCode()=97299`, `97299%8 = 3`) és `insert("red")`, hash: `"red".hashCode()=112785` ⇒ `1`




- `insert("bab")`, hash: `97283` ⇒ `3`



- Jó, ha rövidek a láncok
- Load factor: vödrök száma / elemek száma
- Ha a load factor túl nagy ⇒ több vödör és hash-eljük újra az összes eddigi elemet
- Kétszer (vagy konstansszor) annyi vödröt éri meg létrehozni
- Akkor éri meg, ha a rehash nem túl drága

## Amikor a több vödör nem segít

Ha ugyanaz a hash code

```
println( "AaBBAa".hashCode() ) //1952508096
println( "BBAaAa".hashCode() ) //1952508096
println( "BBBBAa".hashCode() ) //1952508096
println( "AaAaAa".hashCode() ) //1952508096
println( "AaBBBB".hashCode() ) //1952508096
println( "BBAaBB".hashCode() ) //1952508096
println( "BBBBBB".hashCode() ) //1952508096
println( "AaAaBB".hashCode() ) //1952508096
```

Java-ban a `String.hashCode()`:

$$s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + ... + s[n-1]$$

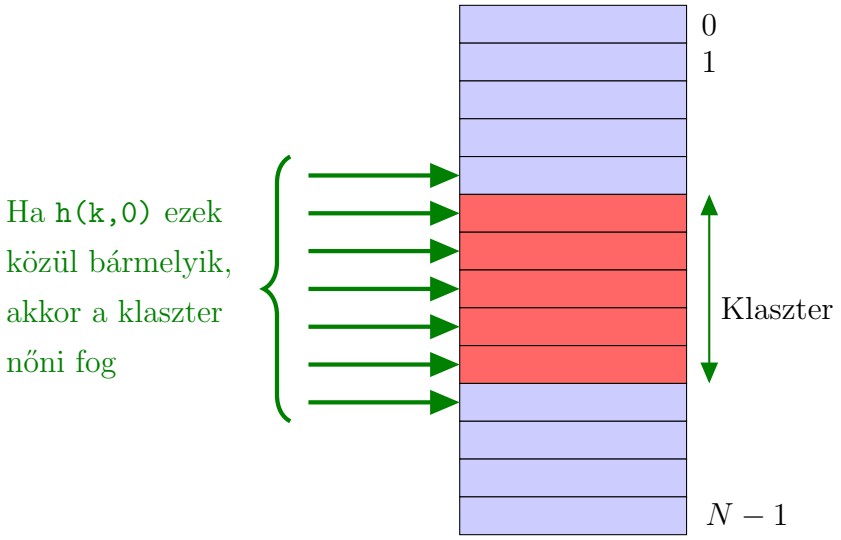
$$\texttt{"Aa".hashCode()} == 31 * 65 + 97 == 2112$$
$$== 31 * 66 + 66 == \texttt{"BB".hashCode()}$$

## Open adressing

- Minden vödörbe max egy kulcsot teszünk
- A hash kód alapján számolt index csak kiindulópont
- Ha foglalt, akkor valamilyen módszerrel végigpróbálunk másokat.
  Jelölés: `probe(h(k),i)`: a `h(k)` kulcs $i$. próbája
  - **linear probing**: `probe( h(k), i ) = (h(k) + i) mod N`
  - **quadratic probing**: $\texttt{probe( h(k), i ) = h(k) + c}_1 \cdot \texttt{i} + \texttt{c}_2 \cdot \texttt{i}^2 \texttt{ mod N}$
  - **double hashing**: $\texttt{probe( h(k), i ) = h}_1\texttt{(k) + i} \cdot \texttt{h}_2\texttt{(k) mod N}$
- A linear probingnél a ,,clustering" probléma, quadraticnál kevésbé, double hashingnál még kevésbé
- Ha a load factor kb. 0.7-re növekszik, mindegyik módszer meglassul

**Clastering (linear probing esetében)**

Az egymás követő foglalt cellák csoportjai, melyek egyre nagyobbak lesznek. És minél nagyobbak, annál nagyobb eséllyel fognak tovább nőni. Vegyük a következő példát:




Ha egy ilyen klaszterbe beletalál egy hash kód, akkor végig kell próbálni a klaszter utolsó cellájáig a lehetséges helyeket. A klaszter vége után be fogjuk szúrni az elemet, így nőni fog a klaszter mérete, ami miatt pedig még nagyobb esélyünk lesz eltalálni a klaszter által fedett intervallumot.

**1. Feladat** Szúrjuk be egy 10 hosszú hash táblába a következő objektumokat chaining, linear probing, quadratic probing ($c_1 = 0, c_2 = 1$) és double hashing módszerekkel. Az elemek két hash függvény (`h1`,`h2`) által adott hash kódjai a következők:

- `h1("cím") = 210`, `h2("cím") = 567`
- `h1("foo") = 130`, `h2("foo") = 132`
- `h1("bar") = 65`, `h2("foo") = 324`
- `h1("fák") = 188`, `h2("foo") = 111`
- `h1("bin") = 785`, `h2("foo") = 176`
- `h1("one") = 960`, `h2("foo") = 608`

**Megoldás**

Megoldás chaining használatával:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |

Megoldás linear probinggal:

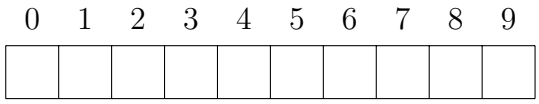
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

A clusteringet itt láthatjuk ,,akcióban", ahogy a két részen elkezdenek ,,csomósodni" az elemek és ahányszor csak beletalál egy hash egy ilyen csomóba, akkor a csomó szélét fogja eggyel növelni, miután sok idő alatt kimegy a csomó széléig.

Megoldás quadratic probing-gal:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

Megoldás double hashing módszerrel:




**2. Feladat** Az input egy hosszú szöveget szavanként tartalmazó String tömb. A feladat az, hogy adjuk meg a benne található összes szó előfordulásainak számát. Írjunk rá pszeudokódot!

### Megoldás

Minden szóhoz el kell tárolnunk, hogy hányszor szerepelt a szövegben. Ahhoz, hogy az algoritmusunk hatékony legyen, olyan adatszerkezetre van szükség, amiben gyorsan rá tudunk keresni a szavakra és eltárolhatunk minden szóhoz egy gyakorisági értéket. Erre a legalkalmasabb adatszerkezet egy hash alapú asszociatív tömb.

```
wordFrequencies(words[])
  freqs = dict()
  for word in words
    n = 1
    if freqs.contains(word)
      n = freqs[word] + 1
    freqs[word] = n
```

Egy lehetséges megvalósítás Java-ban:

```
public static Map<String, Integer> wordFrequencies(String[] words) {
  Map<String, Integer> result = new HashMap<>();
  for (String word : words) {
    int n = 1;
    if (result.containsKey(word)) {
      n = result.get(word) + 1;
    }
    result.put(word, n);
  }
  return result;
}
```

**3. Feladat** Nézzük meg az alábbi kódot. Mi a hiba, hogyan javítanánk?

```
public class Coord {
  int x;
  int y;

  public Coord(int x, int y) {
    this.x = x;
   this.y = y;
  }
}

public static void main(String[] args) {
  Set<Coord> s = new HashSet<Coord>();

  Coord p = new Coord(100, 100);
  s.add(p);
  System.out.println(s.contains(p)); //true

  Coord q = new Coord(100,100);
  System.out.println(s.contains(q)); //false

  Coord r = p;
  System.out.println(s.contains(r)); //true
}
```

### Megoldás

Az elvárás az lenne, hogy ugyanazon értékű adattagokkal rendelkező objektum ugyanazt a hash kódot kapja attól függetlenül, hogy a memóriában nem ugyanott helyezkedik el. Itt látható, hogy a csak azok az objektum számítanak egyenlőnek, amik a memóriában ugyanott vannak. Javítás: definiáljuk felül az `equals` metódust!

```
public class Coord {
  int x;
  int y;
  public Coord(int x, int y) {
   this.x = x;
   this.y = y;
  }
```

```
  public boolean equals(Object o) {
    if (o == null) return false;
    if (!(o instanceof Coord)) return false;
    Coord oc = (Coord)o;
    return x == oc.x && y == oc.y;
  }
}


public static void main(String[] args) {
Set<Coord> s = new HashSet<Coord>();

Coord p = new Coord(100, 100);
s.add(p);
System.out.println(s.contains(p)); //true

Coord q = new Coord(100,100);
System.out.println(s.contains(q)); //false

Coord r = p;
System.out.println(s.contains(r)); //true
}
```

Ez még mindig nem lesz elég. A két objektum hash kódja mégis más lesz, mert Javában a hashkódot alapértelmezés szerint az objektum kezdő memóriacíméből számítjuk, ami a két objektumnál más lett természetesen. Javítás: definiáljuk felül a `hashCode`-ot is!

```
public class Coord {
  int x;
  int y;

  public Coord(int x, int y) {
   this.x = x;
    this.y = y;
  }

  public boolean equals(Object o) {
    if (o == null) return false;
    if (!(o instanceof Coord)) return false;
    Coord oc = (Coord)o;
```

```
    return x == oc.x && y == oc.y;
  }

  public int hashCode() {
    return x ^ y;
  }
}

public static void main(String[] args) {
  Set<Coord> s = new HashSet<Coord>();

  Coord p = new Coord(100, 100);
  s.add(p);
  System.out.println(s.contains(p)); //true

  Coord q = new Coord(100,100);
  System.out.println(s.contains(q)); //true :)

  Coord r = p;
  System.out.println(s.contains(r)); //true
}
```

Az osztályunkban felüldefiniáltuk az `equals` és `hashCode` metódusokat. Akkor itt mégis hol a hiba?

```
public static void main(String[] args) {
  Set<Coord> s = new HashSet<Coord>();

  Coord p = new Coord(100, 100);
  s.add(p);
  System.out.println(s.contains(p)); //true

  p.x = 50;
  System.out.println(s.contains(p)); //false

  Coord q = new Coord(50,100);
  System.out.println(s.contains(q)); //false

  Coord r = new Coord(100,100);
  System.out.println(s.contains(r)); //false wtf
}
```

**Tanulságok:**

- A `hashCode` és az `equals` metódusokat mindig párban kell felülírni
- A `hashCode` soha ne változhasson az objektum életciklusa alatt
- Ha két objektum lehet `equals` bármikor élettartamuk során, akkor a `hashCode`-juk ugyanaz kell legyen

Tehát `hashCode`ot csak immutable mezőkből (olyan adattagok, amiken nem tudunk változtatni) szabad számítani, különben ez (↑) lesz

**4. Feladat** Mi történik, ha ugyanezt az osztályt `HashSet` helyett `TreeSet`-ben szeretnénk eltárolni?

```
public static void main(String[] args) {
  Set<Coord> s = new TreeSet<Coord>();

  Coord p = new Coord(100, 100);
  s.add(p);
}
```

**Megoldás**

Ezt a hibaüzenetet kapjuk:

Exception in thread "main" java.lang.ClassCastException: Coord cannot be cast to java.lang.Comparable

A hibaüzenet arról szól, hogy a `TreeSet`-be csak olyan elemeket rakhatunk, amelyek egy **teljesen rendezett** (mindenki mindenkivel összehasonlítható) univerzumból kerültek ki. A koordinátákat tároló osztályunkhoz nem definiáltuk még ezt a relációt, így nem is tudja bináris fába szervezni azokat a Java. A javítás módja tehát az, hogy megmondjuk, mi ez a reláció, amivel össze tudjuk hasonlítani az elemeket (megjegyzés: itt nem elég az `equals`, a $>$ és $<$ relációt is el kell intézni az $=$ mellett egy metódusban). Ehhez jelezzük, hogy az osztályunk egy ilyen teljesen rendezett halmazba rakható elemeket tartalmaz és definiáljuk felül a vonatkozó metódust:

```
public class Coord implements Comparable<Coord> {
  int x;
  int y;
```

```
  public Coord(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object o) {
    if (o == null) return false;
    if (!(o instanceof Coord)) return false;
    Coord oc = (Coord)o;
    return x == oc.x && y == oc.y;
  }

  public int hashCode() {
    return x ^ y;
  }

  // Sorfolytonos osszehasonlitas
  public int compareTo(Coord o) {
    int comp_y = Integer.compare(y, o.y);
    if (comp_y == 0) {
      return Integer.compare(x, o.x);
    }
    return comp_y;
  }
}
```

**Keresőfa vs Hasítótábla**

**Keresőfa**

- Keresés: $O(\log n)$
- Beszúrás: $O(\log n)$
- Rendezés: van (A kulcsok egy teljesen rendezett halmazból **KELL** jöjjenek)
- Megelőző/rákövetkező elem keresése: gyors, van rá metódus

**Hasítótábla**

- Keresés: $O(1)$
- Beszúrás: $O(1)$
- Rendezés: nincs
- Megelőző/rákövetkező elem keresése: nem lehetséges