

## Hashovací tabulka = Hash table

Hashovací tabulka (hash table, rozptýlená tabulka, hešovací tabulka)

- je datová struktura, která slouží k ukládání dvojic klíč-hodnota.
- kombinuje výhody vyhledávání pomocí indexu (složitost  $O(1)$ ) a procházení seznamu (nízké nároky na paměť).
- konstantní složitost
- rozloží prvky rovnoměrně

**POZOR klíč NENÍ index (adresa), ale podle klíče se hodnoty ukládají na indexy (adresy).**

### Podporované funkce

- vkládání
- mazání
- vyhledávání
- modulo = typická hash fce  $h(k) = k \bmod m$

### Možnosti ukládání dvojic klíč-hodnota

#### Pole

Složitost  $O(1)$ , protože je zde přímé adresování prvku.

Polem můžeme implementovat rychle, ale nevýhodou je prostorová náročnost.

Uvažujme, že chceme ukládat dvojice klíč-hodnota a vyhledávat v nich. Jednou z možností by pak bylo jako klíč zvolit celé číslo (nebo nějaké mapování klíče na celé číslo) a hodnoty ukládat do pole. Tímto bychom si sice zajistili, že prvek nalezneme s konstantní paměťovou složitostí (jednoduše bychom jej vyžádali pomocí indexu), ale na druhou stranu by docházelo k obrovskému mrhání paměti.

Na *heš* se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejně“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

### Spojový seznam

Opačným přístupem by bylo využití seznamu, ve kterém bychom vyhledávali sekvenčně. Zcela zřejmě by tímto odpadl problém s nevyužitými klíči a pamětí (nevyužití klíče by vůbec neexistovaly).

Nevýhoda je ovšem zcela zřejmá – výkonnost.

V okamžiku, kdy bychom se neptali jen tisícovky respondentů (lokální výzkum), ale uspořádali bychom globální anketu s miliony dotazovaných, tak by v těchto datech již nešlo rozumně vyhledávat (pro nalezení jednoho záznamu bychom spotřebovali  $O(n)$  kroků), protože musí prohledat vše od prvního.

## Hashovací (rozptýlená) tabulka

Hashovací (rozptýlená) tabulka je struktura, jež je postavena nad polem omezené velikosti  $n$  (tzn. pole nepopisuje celý stavový prostor klíče), a která pro adresaci využívá hashovací funkci. Nalezení prvku pro daný klíč zabere průměrně  $O(1)$  operací.

## Hashovací (rozptylovací) funkce

Hashovací funkce má následující vlastnosti:

- nezaručuje, že pro dva různé objekty vrátí různou adresu = vzniká kolize – lze řešit několika způsoby (viz dále)
- využívá celého prostoru adres se stejnou pravděpodobností, tzn. rovnoměrné rozložení prvků.
- výpočet adresy proběhne velmi rychle.
- pomocí rozptylovací fce zredukujeme potřebné místo.
- průměrná složitost je konstantní  $O(1)$
- $h(k) = k \bmod m$  ( $k$  je klíč,  $m$  je velikost pole)

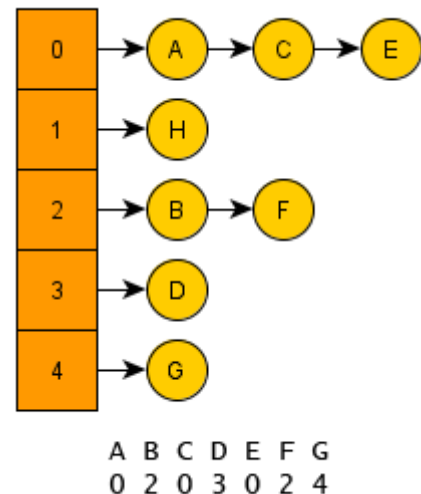
**Synonymum** = prvky mají stejný výsledek funkce  $h_1(k)$ .

## Kolize

Jednou z uvedených vlastností hashovací funkce je, že nezaručuje, že dvěma různým objektům nepřihadí stejné adresy. Situaci, kdy chceme uložit na stejnou adresu více objektů, se říká kolize. Kolize = dva nebo více klíčů (synonum) se zobrazují na tutéž pozici v tabulce.

Odstranění kolizí

- zřetěžené hashování = řetězení klíčů (synonum) do pomocných datových struktur (spojový seznam, vyhledávací strom)
- otevřené adresování = ukládání synonym (klíčů) přímo na indexy tabulky, které jsou vypočteny pomocí další funkce (rozptylovací, hash fce)
  - lineární probing (lineární prohledávání)
  - lineární prohledávání s krokem
  - double hashing
  - kvadratické hashování



## Zřetěžené hashování

- nejjednodušší způsob odstranění kolizí
- hodnoty ukládáme do spojových seznamů
- nastane-li kolize, prvek se pouze přidá na konec adresovaného spojového seznamu
- **nevýhoda** – při velkém zaplnění tabulky dojde k rychlé degradaci výkonu kvůli sekvenčnímu prohledávání příslušných seznamů.

## Otevřené hashování (rozptylování, adresování)

- hodnoty jsou ukládány přímo do pole.
- klíče jsou unikátní.
- má dva parametry – klíč a číslo pokusu
- kolidující prvky se neřetězí, ale ukládají se do další volné pozice v hlavním poli.
- při mazání je potřeba pozici prvku označit značkou (např. DELETED nebo "null") nebo zbytek "řetězce" posunout dopředu (vlevo).
- u VKLÁDÁNÍ – lineární složitost, musí se nejprve projít všechny "chlívečky"
- značku DELETED fce prohledávání ignoruje, fce VKLÁDÁNÍ přepisuje

**PŘ** - vkládám hodnoty 1, 3, 9, 11 a  $m = 4$  (= jsou 4 indexy → mod 4)

Zřetěžené hashování

0	
1	→ 1, 9
2	
3	→ 3, 11

Otevřené hashování

0	1	2	3
↓	↓	↓	↓
11	1	9	3

Existují tři základní strategie, pomocí nichž se tabulka vypořádává s kolizemi – **linear probing**, **linear probing s krokem** a **double hashing**.

### Linear probing (otevřené hashování s lineárním prohledáváním)

- nejprve vypočteme adresu, na kterou daný prvek uložíme
- je-li adresa obsazená, tak se posuneme o jedno místo dál a zkusíme prvek uložit znovu
- postup se opakuje dlouho, dokud se nám prvek nepodaří uložit.

Ukládací schéma lze charakterizovat funkcí ( $i \in \{0, 1, \dots, n/1\}$ ):  $adresa = h(k, m) + i \bmod m$

**PŘ** – mod 13, bude pole o indexech 0 – 12

0	1	2	3	4	5	6	7	8	9	10	11	12
				↓	↓	↓	↓					
				4	17	19	5					

- vkládám 4 = na  $i = 4$ , protože  $4 \bmod 13 = 0$ , zbyde 4
- vkládám 19 = na  $i = 6$ , protože  $19 \bmod 13 = 1$ , zbyde 6
- vkládám 17 = na  $i = 4$ , protože  $17 \bmod 13 = 1$ , zbyde 4 – KOLIZE!!! vložím na nejbližší volný  $i = 5$
- vkládám 5 = na  $i = 5$ , protože  $5 \bmod 13 = 0$ , zbyde 5 – KOLIZE!!! vložím na nejbližší volný  $i = 7$

Vyhledávání prázdného indexu – hledám např. 17 (= 13 + 4).

Začnu hledat od první možné pozice  $i = 4$  a hledám postupně směrem vpravo, dokud nenajdu prázdný index.

### Linear probing s krokem

- zvolíme si hodnotu pevného kroku
- hodnota kroku musí být nesoudělná s  $m$  (velikost pole)

### Shlukování (= clustering)

Velkou nevýhodou linear probing je shlukování. Vzhledem k principu ukládání objektů dochází ke vzniku shluků objektů, jež mají blízkou nebo totožnou adresu. Tyto shluky je pak při vyhledávání nutné sekvencně procházet, což má dopad na výkon. Ten je ještě vyšší než u zřetězeného rozptylování, protože shluky mohou obsahovat prvky odpovídající více klíčům.

### Mazání prvků

- mazaný prvek se označí značkou (DELETE, "null" apod.)
- tím vznikne objekt, který značí, že je dané místo prázdné
- operace vyhledávání null objekt přeskakuje
- operace vkládání null objekt přepíše, tzn. uloží na toto místo nový prvek.
- je při mazání možné prvek odstranit
- všechny prvky ve zbylé části shluku (tj. po nejbližší null) je ale nutno re-uložít, tzn. re-hashovat.
- není možné prvky pouze "setřepat", protože bychom mohli ztratit hodnoty pro další klíče, které se v daném shluku vyskytují (pokud bychom z tabulky na obrázku smazali prvek C a zbytek shluku setřepali, tak již nikdy nenalezneme prvek A)
- POZOR – nutno nejprve zjistit, co patří do shluku!!!

	A						
	A		B				
C	A		B				
C	A		D	B			
C	A		D	B		E	

	A	B	C	D	E
	1	4	0	0	3
	3	3	3	3	3

### Double hashing

- eliminuje shlukování díky využití dvojice rozptylovacích funkcí  $h_1(k)$  a  $h_2(k)$
- klasická fce  $h_1(k) = k \bmod m$  vypočte iniciální adresu stejným způsobem jako u linear probing nebo zřetězeného rozptylování
- fce  $h_2(k)$  určí velikost kroku  $h_2(k) = k \bmod m + 1$  a její výsledek musí být nesoudělný s "m"
- funkce  $h_2(k)$  nastoupí, pokud je dané místo již obsazené, a vypočte posun
- pokud je  $i$  nové místo plné, dojde opět k posunu na základě funkce  $h_2(k)$
- POZOR u fce  $h_2(k)$  nutno zajistit nesoudělnost hodnoty  $m$  a velikost kroku
- **adresa =  $h_1(k) + h_2(k)$**

PŘ

0	1	2	3	4	5	6	7	8	9	10
					↓	↓			↓	
					5	16			6	

mod 11 = máme 11 indexů  
 $h_1(k) = k \bmod 11$  (=  $k \% 11$ )  
 $h_2(k) = k \bmod 4 + 1$

hash 5, 16, 6

### Mazání prvků

Při mazání prvků si tentokrát již nemůžeme pomoci novým uložením zbytku shluku, protože double hashing shluky netvoří, resp. tvoří je výrazně méně.

Musíme proto striktně využívat již zmíněného null objektu, kterým nahradíme mazaný objekt.

### Hashovací tabulka

```
@param <KEY> typovy parametr klice
@param <VALUE> typovy parametr hodnoty
public class HashTable<KEY, VALUE> {

    //Pomer zaplneni pri kterem dojde k vytvoreni nove (vetsi) tabulky
    private final float LOAD_FACTOR = 0.75f;

    //Pomer zaplneni pri kterem dojde k vytvoreni nove (mensi tabulky)
    private final float COLLAPSE_RATIO = 0.1f;

    //Hodnota, pod kterou nikdy neklesne velikost tabulky
    private final int INITIAL_CAPACITY;

    //Pocet ulozenych prvku
    private int size = 0;
    private Entry<KEY, VALUE>[] table;

    Zkonstruuje hashovací tabulku s vchozi kapacitou 10
    public HashTable() {
        this(10); //vychozi kapacita
    }

    Zkonstruuje hashovací tabulku
    @param initialCapacity kapacita, pod kterou nikdy tabulka neklesne
    public HashTable(int initialCapacity) {
        if (initialCapacity <= 0) {
            throw new IllegalArgumentException("Kapacita nesmi byt nulova nebo
            zaporna");
        }
        this.INITIAL_CAPACITY = initialCapacity;
        this.table = new Entry[initialCapacity];
    }

    Vlozi prvek do tabulky, pokud jiz prvek s danym klicem existuje, tak bude
    nahrazen
    @param key klic
    @param value hodnota
    @return @null pokud klic v tabulce neexistuje, v opacnem pripade
    nahrazena hodnota
    @throws IllegalArgumentException pokud je klic @null
    public VALUE put(KEY key, VALUE value) {
        if (key == null) {
            throw new IllegalArgumentException("Klic nesmi byt null");
        }
        VALUE val = performPut(key, value);
    }
}
```

```

    if (val == null) {
        size++;
        resize();
    }
    return val;
}

```

**Odstrani prvek odpovídající danému klíci**

```

    @param key klíč
    @return @null pokud klíč v tabulce neexistuje, v opačném případě
    odstraněná hodnota
public VALUE remove(KEY key) {
    Entry<KEY, VALUE> e = getEntry(key);
    if (e == null) { //prvek neexistuje
        return null;
    }
    VALUE val = e.value;
    e.key = null; //prvek je nyní sentinelem
    e.value = null; //odstraníme referenci na hodnotu, aby GC mohl
    cinit svou práci

    size--;
    resize();
    return val; //vratíme původní hodnotu
}

```

**Navrátí hodnotu asociovanou s daným klíčem**

```

    @param key klíč
    @return hodnota, @null pokud není v tabulce obsazena
public VALUE get(KEY key) {
    Entry<KEY, VALUE> e = getEntry(key);
    if (e == null) {
        return null;
    }
    return e.value;
}

```

**Dotaz na přítomnost prvku s daným klíčem**

```

    @param key klíč
    @return @true, pokud tabulka obsahuje hodnotu asociovanou s daným
    klíčem, @false v opačném případě
public boolean contains(KEY key) {
    return getEntry(key) != null;
}

```

**Dotaz na počet uložených prvků**

```

    @return počet uložených prvků
public int size() {
    return size;
}

```

**Kolekce všech uložených hodnot**

```

    @return kolekce uložených hodnot (poradí není žádným způsobem
    zaručeno)
public Collection<VALUE> values() {
    List<VALUE> values = new ArrayList<VALUE>(size);
    for (int i = 0; i < table.length; i++) {
        if (table[i] != null && table[i].key != null) {
            values.add(table[i].value);
        }
    }
    return values;
}

```

**Kolekce všech klíčů**

```

    @return kolekce všech klíčů, které se vyskytují v tabulce
public Collection<KEY> keys() {
    List<KEY> keys = new ArrayList<KEY>(size);
}

```

```

    for (int i = 0; i < table.length; i++) {
        if (table[i] != null && table[i].key != null) {
            keys.add(table[i].key);
        }
    }
    return keys;
}

```

**Vrati zaznam, ktory odpovida danemu klice**

```

    @return
private Entry getEntry(KEY key) {
    int index = key.hashCode() % table.length;
        //dokud nenarazime na volne misto, existujici zaznam pro klic nebo
        sentinel
    while (table[index] != null) {
        if (key.equals(table[index].key)) {                //zaznam existuje
            return table[index];
        }
        index = (index + 1) % table.length;                //prejdeme na dalsi adresu
    }
    return null;                //nenalezen
}

```

**Provede samotnou operace vlozeni, aniz by jakkoliv menil informaci o velikosti pole (size), pripadne menil velikost pole (resize)**

**Je-li klic jiz v tabulce obsazen, tak bude asociovana hodnota nahrazena**

```

    @param key klic
    @param value hodnota
    @return byl-li klic v tabulce jiz obsazen, tak nahrazena hodnota, v
    opacnem pripade @null
private VALUE performPut(KEY key, VALUE value) {
    Entry<KEY, VALUE> e = getEntry(key);
    if (e != null) {                //prvek je v tabulce
        VALUE val = e.value;
        e.value = value;                //zamenime hodnoty
        return val;
    }
    int index = key.hashCode() % table.length;
    while (table[index] != null && table[index].key != null) {
        //dokud nenarazime na prazdne misto nebo sentinel
        index = (index + 1) % table.length;                //posuneme se o adresu dal
    }
    if (table[index] == null) {                //prazdne misto
        table[index] = new Entry<KEY, VALUE>();
    }
    table[index].key = key;
    table[index].value = value;
    return null;
}

```

**Vypocte velikost, jakou by mela tabulka mit**

```

    @return velikost, jakou by mela tabulka mit
private int calculateRequiredTableSize() {

    if (this.size() / (double) table.length >= LOAD_FACTOR) {
        //tabulka je preplnena
        return table.length * 2;
    } else if (this.size() / (double) table.length <= COLLAPSE_RATIO) {
        //vratime vetsi z hodnot SOUCASNA VELIKOST/2 a INITIAL_CAPACITY
        return Math.max(this.INITIAL_CAPACITY, table.length / 2);
    } else {
        return table.length; //tabulka ma spravnou velikost
    }
}

```

**Vypocte velikost, jakou by mela tabulka mit**

```

    @return velikost, jakou by mela tabulka mit
private int calculateRequiredTableSize() {

    if (this.size() / (double) table.length >= LOAD_FACTOR) {
        //tabulka je preplnena
        return table.length * 2;
    } else if (this.size() / (double) table.length <= COLLAPSE_RATIO) {
        //vratime vetsi z hodnot SOUCASNA VELIKOST/2 a INITIAL_CAPACITY
        return Math.max(this.INITIAL_CAPACITY, table.length / 2);
    } else {
        return table.length; //tabulka ma spravnou velikost
    }
}

```

**Zmeni velikost tabulky, je-li to nutne**

```
private void resize() {
    int requiredTableSize = calculateRequiredTableSize();
    if (requiredTableSize != table.length) {
        //pokud je treba zmenit velikost tabulky
        Entry<KEY, VALUE>[] oldTable = table;
        table = new Entry[requiredTableSize];
        //tak vytvorime novou tabulku
        for (int i = 0; i < oldTable.length; i++) {
            if (oldTable[i] != null && oldTable[i].key != null) {
                this.performPut(oldTable[i].key, oldTable[i].value);
                //a hodnoty do ni ulozieme
            }
        }
    }
}
```

**Vnitřní třída reprezentující záznam tabulky**

```
private class Entry<KEY, VALUE> {
    //Klic, @null == prvek je sentinel
    private KEY key;

    //Hodnota
    private VALUE value;
}
```