Searching

Direct

..\files\11-01 Direct(indexed).pdf

Linear

..\files\11-02 linear.pdf

..\files\11-03 sentinel.pdf

Bisection

Hashing Method

podklady

Java pro začátečníky (20) - Hashovací tabulka

V předešlých dílech jsme mimo jiné vytvářeli kolekce – dynamické pole a spojový seznam. Jejich použití nám oproti práci s polem v mnohém usnadnilo práci, zejména pak pokud jde o ukládání dat předem neznámé délky. Jedna otázka ovšem zůstala otevřená – vyhledávání.

Vyhledávání

Seznam

Představme si, že vytváříme evidenci obyvatel nějakého státu. Prvním řešením, které nás napadne, je uložit všechny obyvatele do seznamu. Pokud budeme chtít zjistit, zda-li daný obyvatel existuje nebo žije, tak celý seznam jednoduše projdeme.

Toto řešení bude perfektně fungovat pro malé státy (Monaco, San Marino...), u těch středně velkých jako je Česká republika se již začne projevovat sekvenčnost prohledávání – v krajním případě nutnost projít každého jednotlivého obyvatele v evidenci. U velmi lidnatých států (Čína, Indie) již bude toto řešení zcela selhávat.

Pole

Alternativně bychom mohli všechny obyvatele umístit do pole, ve kterém bychom vyhledávali pomocí identifikačního čísla obyvatele (rodné číslo, číslo pojistky...). Pokud by identifikační číslo mělo 10 číslic, tak bychom vytvořili pole o velikosti 10^{10} (10 miliard). Jednotlivé prvky bychom potom vyhledávali přímo přes příslušný index (tj. s konstantní složitostí $O(1)$).

Nevýhoda tohoto přístupu je zřejmá – pro velké státy s miliardou obyvatel promrháme 90% prostoru pole, pro pro středně velké pak přes 99 procent, u malých států bychom si ani nevšimli, že tam nějaká data jsou...

Hashovací tabulka

Hashovací (rozptýlená) tabulka je struktura postavená nad polem, jež slouží k ukládání dvojic klíč-hodnota, a která kombinuje výhody obou předchozích přístupů – umožňuje

vyhledat prvek pomocí průměrně konstantního počtu operací a nevyžaduje velké množství dodatečné (nevyužité) paměti.

Hashovací funkce

Jádrem hashovací tabulky je využití tzv. hashovací funkce, která vypočte pro předaný klíč adresu v rozsahu pole, na níž bude příslušný prvek uložen.

Hashovací funkce má následující vlastnosti:

Konzistentně vrací pro stejné objekty (klíče) stejné adresy (slovo stejné typicky neznamená stejné instance, ale stejná data).

Nezaručuje, že pro dva různé objekty vrátí různou adresu.

Využívá celého prostoru adres se stejnou pravděpodobností.

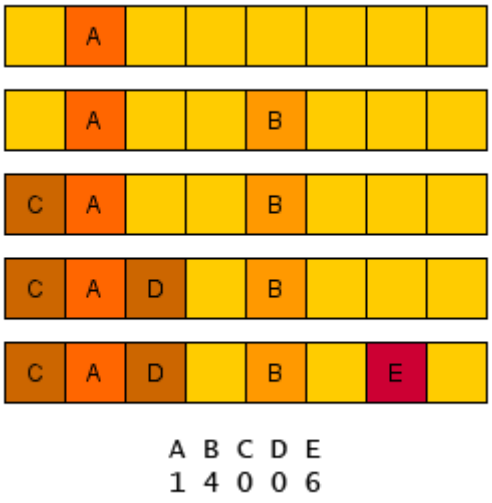
Výpočet adresy proběhne velmi rychle.

Jelikož funkce dle první vlastnosti vrací konzistentně pro stejný klíč stejnou adresu, lze uložený objekt vyhledat stejným postupem, jakým byl uložen.

Kolize

Dle druhé vlastnosti hashovacích funkcí může dojít k tomu, že dva různé objekty (klíče) dostanou stejnou adresu. Této situaci se říká kolize. Kolize se dá řešit mnoha způsoby, my si ale ukážeme jeden poměrně jednoduchý postup, který ale dosahuje na řídkých tabulkách velmi dobrého výkonu – linear probing.

Linear probing



Linear probing

Linear probing řeší koliza následovně. Je-li adresa (vypočtené místo) obsazená, tak jednoduše zkusí následující pozici. Takto postupuje tak dlouho, dokud se mu položku nepodaří uložit (tj. nenarazí na prázdnou buňku). Z jednoduchosti postupu ale plyne také jedna nevýhoda – shlukování (clustering).

Shlukování

Shlukování znamená, že v tabulce vznikají dlouhé jednolité sekvence uložených prvků, v rámci kterých mohou být vypočtené klíče různě pomíchané. Pro nalezení (nebo vyloučení existence) prvku je pak často nutné projít velké množství buněk, což notně degraduje výkon. Naštěstí bylo dokázáno, že k největší degradaci dochází až při zaplnění tabulky z více než 75%. Stačí tedy tabulku včas natáhnout (a prvky do nové tabulky znovu uložit).

Mazání prvků

Vzhledem k promíchanosti shluků nelze mazaný prvek pouze jednoduše odstranit (nahradit ukazatelem na null), protože by se při vyhledávání stal zbytek shluku nedostupným (vyhledávání je ukončeno při nalezení prvního volného místa). Stejně tak nelze zbytek shluku pouze posunout o jedno místo doleva, protože ve zbytku shluku může existovat prvek, který leží právě na své správné adrese a tento by byl po přesunutí nenávratně ztracen.

Při mazání prvků můžeme postupovat následujícími dvěma způsoby. Tím prvním je zbytek shluku odstranit a znovu uložit. Druhým způsobem je vložit místo smazaného prvku speciální objekt hlídky (sentinel), který je při vyhledávání ignorován (přeskočen), ale při vkládání nového prvku je považován za prázdné místo (a je nahrazen).

Hashovací funkce

Každý javovský objekt obsahuje metodu hashCode (dokumentace), která vrací, v souladu s vlasnostmi hashovací funkce uvedenými výše, celé číslo – hash.

Tuto funkci jsme zmínili již v sedmém dílu, ve kterém jsme překrývali metodu equals. Řekli jsme si tehdy, že mezi těmito dvěma metodami existuje užší vazba. Tato vazba je specifikována pomocí dokumentačního komentáře v třídě Object, v níž jsou obě zmíněné metody deklarovány, a uvaluje funci hashCode následující omezení:

Za předpokladu, že nebyla v objektu změněna žádná informace využívaná metodou equals, tak volání funkce hashCode v rámci jednoho běhu aplikace vrátí vždy stejný výsledek. Tento výsledek se může lišit v rámci jednotlivých běhů aplikace.

Pokud jsou dva objekty totožné dle metody equals, tak hashCode musí pro oba vrátit stejný výsledek.

Pokud dva objekty nejsou totožné, tak hashCode nemusí vrátit různý výsledek.

Příklad

V sedmém dílu jsme vytvářeli třídu zvířátka, které mělo druh a vydávalo různé zvuky. Dva objekty reprezentovaly to samé zvířátko právě tehdy, měly-li uvedené stejný druh a produkovaný zvuky.

Java

view source

print?

```
01./**
02. * Zviratko (v kodu jsou uvedeny pouze relevantni pasaze, zbytek je vyteckovan)
03. * @author Pavel Micka
```

```
04. */
05.public class Animal {
06.    /**
07.     * Druh zviratka
08.     */
09.    private String kind;
10.    /**
11.     * Zvuk, ktery zviratko vydava
12.     */
13.    private String sound;
14.    .
15.    .
16.    .
17.    @Override
18.    public boolean equals(Object obj) {
19.        if(obj instanceof Animal){ //je stejneho typu
20.            Animal animal = (Animal) obj; //muzeme tedy pretypovat
21.            if(this.kind.equals(animal.kind) && this.sound.equals(animal.sound)){
22.                return true; //je stejneho druhu a vydava stejny zvuk
23.            }
24.        }
25.        return false;
26.    }
```

```
27.    @Override
28.    public int hashCode() {
29.        int hash = 5;
30.        hash = 79 * hash + (this.kind != null ? this.kind.hashCode() : 0);
31.        hash = 79 * hash + (this.sound != null ? this.sound.hashCode() : 0);
32.        return hash;
33.    }
34.}
```

Metoda equals

Metoda equals nejprve kontroluje, zda-li je porovnávaný objekt typu Animal. Pokud ano, tak jej přetypuje a porovná druh a zvuk. Jsou-li obě tyto vlastnosti totožné, pak objekty reprezentují stejné zvířátko, v opačném případě tomu tak není.

Metoda hashCode

Metoda hashCode musí reflektovat třídní proměnné použité v operaci equals. Z tohoto důvodu smísí výsledek hashovací funkce pro oba použité řetězce.

Zde využívaná hashovací funkce pro řetězce funguje téměř totožně jako naše implementace. Jediným rozdílem je, že se pro výpočet využívá číselných hodnot jednotlivých znaků (a nedochází již k dalšímu volání hashCode).

Z implementačního hlediska je poměrně důležité, aby multiplikativní konstanty byly prvočíselné – minimalizuje se tak počet kolizí. Samotná operace násobení pak společně s modulární povahou primitivního typu integer zaručí, že dojde dobrému rozptýlení výsledných hodnot.

Hashovací tabulka

Nyní již k samotné implementaci tabulky. Budeme ji vytvářet s využitím metody linear probing. Prvky budeme mazat za využití hlídky (sentinelu).

Rozhraní

První věcí, kterou si musíme rozmyslet je rozhraní typu tabulka. Jeho specifikace nám umožní bezbolestně vyměnit implementaci v případě, že s tou stávající nebudeme z nějakého důvodu spokojeni.

Java

view source

print?

```
01./**
02. * Rozhrani tabulky
03. * @author Pavel Micka
04. * @param <KEY> typovy parametr klice
05. * @param <VALUE> typovy parametr hodnoty
06. */
07.public interface TableIface<KEY, VALUE> {
08.    /**
09.     * Vlozi prvek do tabulky, pokud jiz prvek s danym klicem existuje, tak bude nahrazen
10.     * @param key klic
11.     * @param value hodnota
12.     * @return @null pokud klic v tabulce neexistuje, v opacnem pripade nahrazena hodnota
13.     * @throws IllegalArgumentException pokud je klic @null
14.     */
15.    public VALUE put(KEY key, VALUE value);
16.    /**
```

```
17.     * Odstrani prvek odpovidajici danemu klici
18.     * @param key klic
19.     * @return @null pokud klic v tabulce neexistuje, v opacnem pripade odstranena hodnota
20.     */
21.    public VALUE remove(KEY key);
22.    /**
23.     * Navrati hodnotu asociovanou s danym klicem
24.     * @param key klic
25.     * @return hodnota, @null pokud neni v tabulce obsazena
26.     */
27.    public VALUE get(KEY key);
28.    /**
29.     * Dotaz na pritomnost prvku s danym klicem
30.     * @param key klic
31.     * @return @true, pokud tabulka obsahuje hodnotu asociovanou s danym klicem, @false v opacnem pripade
32.     */
33.    public boolean contains(KEY key);
34.    /**
35.     * Dotaz na pocet ulozenych prvku
36.     * @return pocet ulozenych prvku
37.     */
```

```
38.    public int size();
39.    /**
40.     * Kolekce vsech ulozenych hodnot
41.     * @return kolekce ulozenych hodnot (poradi neni zadnym zpusobem zaruceno)
42.     */
43.    public Collection<VALUE> values();
44.    /**
45.     * Kolekce vsech klicu
46.     * @return kolekce vsech klicu, ktere se vyskytuji v tabulce
47.     */
48.    public Collection<KEY> keys();
49.}
```

Rozhraní je typováno pomocí dvojice generických parametrů – klíče a hodnoty. Toto nám umožní ukládat do tabulky rozdílné typy, aniž bychom ztratili silnou typovost. Metody keys a values vrací object typu Collection (dokumentace). Collection je rozhraní, jež rozšiřuje interface Iterable, a které je implementováno standardními kolekcemi Javy. Díky této abstrakci můžeme vrátit kontejner, který nejvíce odpovídá našim představám (seznam, strom, tabulku, množinu...).

Konstanty, třídní proměnné, vnitřní třída Entry a konstruktory třídy tabulky

Java

view source

print?

```
01./**
02. * Hashovaci tabulka
03. * @author Pavel Micka
```

```
04. * @param <KEY> typovy parametr klice
05. * @param <VALUE> typovy parametr hodnoty
06. */
07.public class HashTable<KEY, VALUE> implements TableIface<KEY, VALUE> {
08.
09.    /**
10.     * Pomer zaplneni pri kterem dojde k vytvoreni nove (vetsi) tabulky
11.     */
12.    private final float LOAD_FACTOR = 0.75f;
13.    /**
14.     * Pomer zaplneni pri kterem dojde k vytvoreni nove (mensi tabulky)
15.     */
16.    private final float COLLAPSE_RATIO = 0.1f;
17.    /**
18.     * Hodnota, pod kterou nikdy neklesne velikost tabulky
19.     */
20.    private final int INITIAL_CAPACITY;
21.    /**
22.     * Pocet ulozenych prvku
23.     */
24.    private int size = 0;
25.    private Entry<KEY, VALUE>[] table;
26.
```

```
27.  /**
28.   * Zkonstruuje hashovaci tabulku s vychozi kapacitou 10
29.   */
30.  public HashTable() {
31.      this(10); //vychozi kapacita
32.  }
33.
34.  /**
35.   * Zkonstruuje hashovaci tabulku
36.   * @param initialCapacity kapacita, pod kterou nikdy tabulka neklesne
37.   */
38.  public HashTable(int initialCapacity) {
39.      if (initialCapacity <= 0) {
40.          throw new IllegalArgumentException("Kapacita nesmi byt nulova nebo
zaporna");
41.      }
42.      this.INITIAL_CAPACITY = initialCapacity;
43.      this.table = new Entry[initialCapacity];
44.  }
45.
46.  /**
47.   * Vnitrni trida reprezentujici zaznam tabulky
48.   */
```

```
49.    private class Entry<KEY, VALUE> {
50.
51.        /**
52.         * Klic, @null == prvek je sentinel
53.         */
54.        private KEY key;
55.        /**
56.         * Hodnota
57.         */
58.        private VALUE value;
59.    }
60.}
```

Deklarace a konstanty

V rámci deklarace musíme nejprve předat získané typové parametry výše definovanému rozhraní. Základem definice pak jsou konstanty LOAD_FACTOR - poměr zaplnění, při kterém bude zalokována nová větší tabulka, COLLAPSE_RATIO - poměr zaplnění, při kterém bude zalokována nová menší tabulka a INITIAL_CAPACITY - hodnota, pod kterou nikdy neklesne velikost tabulky.

Třídní proměnné, vnitřní třída Entry a konstruktory

Tabulka obsahuje dvě třídní proměnné - počet uložených prvků (size) a pole záznamů (table). Tyto položky, a již uvedená konstanta INITIAL_CAPACITY, jsou inicializovány v rámci konstruktorů, z nichž jeden je bezparametrický a druhý umožňuje uživateli nastavit výchozí velikost tabulky.

Jednotlivé záznamy jsou reprezentovány soukromou vnitřní třídou Entry, která má dvě proměnné klíč (key) a hodnotu (value). Pokud má záznam nullový klíč, tak je považován za hlídku (sentinel).

Pomocné metody

Nejprve vytvoříme pomocné metody, které nám usnadní manipululaci s tabulkou samotnou.

Metoda calculateRequiredTableSize

Java

view source

print?

```
01./**
02. * Vypocte velikost, jakou by mela tabulka mit
03. * @return velikost, jakou by mela tabulka mit
04. */
05.private int calculateRequiredTableSize() {
06.    if (this.size() / (double) table.length >= LOAD_FACTOR) { //tabulka je preplnena
07.        return table.length * 2;
08.    } else if (this.size() / (double) table.length <= COLLAPSE_RATIO) {
09.        //vratime vetsi z hodnot SOUCASNA_VELIKOST/2 a INITIAL_CAPACITY
10.        return Math.max(this.INITIAL_CAPACITY, table.length / 2);
11.    } else {
12.        return table.length; //tabulka ma spravnou velikost
13.    }
14.}
```

Metoda calculateRequiredTableSize vypočte velikost, jakou by měla tabulka mít vzhledem ke svému obsahu. K tomuto využívá trojice konstant LOAD_FACTOR, COLLAPSE_RATIO a INITIAL_CAPACITY. V případě, že je tabulka příliš krátká, doporučí

metoda její zvětšení na dvojnásobnou kapacitu, je-li tabulka příliš dlouhá, tak naopak doporučí poloviční velikost. Má-li tabulka ideální délku, tak tuto hodnotu navrátí (tj. doporučí neměnit velikost).

Metoda getEntry

Java

view source

print?

```
01./**
02. * Vrati zaznam, ktery odpovida danemu klice
03. * @return
04. */
05.private Entry getEntry(KEY key) {
06.    int index = key.hashCode() % table.length;
07.    //dokud nenarazime na volne misto, existujici zaznam pro klic nebo sentinel
08.    while (table[index] != null) {
09.        if (key.equals(table[index].key)) { //zaznam existuje
10.            return table[index];
11.        }
12.        index = (index + 1) % table.length; //prejdeme na dalsi adresu
13.    }
14.    return null; //nenalezen
15.}
```

Metoda getEntry vrátí záznam, který odpovídá předanému klíči. Nejprve proto vypočítá základní adresu, na které buď může být prázdné místo (null) nebo shluk prvků. Metoda

poté prověří všechny prvky případného shluku, zda-li nemají hledaný klíč. V případě shody odpovídající prvek vrátí. V případě, že shluk hledaný prvek neobsahuje (nebo je základní adresa prázdná), vrátí metoda null.

Metoda performPut

Java

view source

print?

```
01./**
02. * Provede samotnou operace vlozeni, aniz by jakkoliv menil informaci o velikosti
03. * pole (size), pripadne menil velikost pole (resize)
04. *
05. * Je-li klic jiz v tabulce obsazen, tak bude asociovana hodnota nahrazena
06. *
07. * @param key klic
08. * @param value hodnota
09. * @return byl-li klic v tabulce jiz obsazen, tak nahrazena hodnota, v opacnem pripade @null
10. */
11.private VALUE performPut(KEY key, VALUE value) {
12.    Entry<KEY, VALUE> e = getEntry(key);
13.    if (e != null) {
14.        //prvek je v tabulce
15.        VALUE val = e.value;
16.        e.value = value; //zamenime hodnoty
```

```
17.       return val;
18.   }
19.   int index = key.hashCode() % table.length;
20.   while (table[index] != null && table[index].key != null) { //dokud nenarazime na prazdne misto nebo sentinel
21.       index = (index + 1) % table.length; //posuneme se o adresu dal
22.   }
23.   if (table[index] == null) { //prazdne misto
24.       table[index] = new Entry<KEY, VALUE>();
25.   }
26.   table[index].key = key;
27.   table[index].value = value;
28.
29.   return null;
30.}
```

Metoda performPut provede prosté vložení prvku do tabulky bez jakýchloliv vedlejších akcí (změna velikosti tabulky, změna počtu prvků v tabulce). Této metody totiž využijeme při dvou příležitostech - při vložení prvku (tehdy si obalující metoda zajistí případné dodatečné akce) a při změně velikosti tabulky, kdy je třeba všechny prvky znovu uložit (v tomto případě jsou jakékoliv další akce nežádoucí, protože nedochází ke vložení nových prvků, ale již těch existujících a navíc má tabulka zaručenou ideální délku).

Nyní již k samotné implementaci této metody. Metoda nejprve zjistí pomocí volání getEntry, jestli tabulka již neobsahuje záznam pro hledaný klíč. Pokud ano, tak nahradí asociovanou hodnotu za novou a vrátí tu původní. V opačném případě musí metoda najít volné místo (nebo sentinel), na které vloží nový záznam. Jelikož se jedná o čisté vložení, tak v souladu s kontaktem metoda vrátí null.

Metoda resize

Java

view source

print?

```
01./**
02. * Zmeni velikost tabulky, je-li to nutne
03. */
04.private void resize() {
05.    int requiredTableSize = calculateRequiredTableSize();
06.    if (requiredTableSize != table.length) { //pokud je treba zmenit velikost tabulky
07.        Entry<KEY, VALUE>[] oldTable = table;
08.        table = new Entry[requiredTableSize]; //tak vytvorime novou tabulku
09.        for (int i = 0; i < oldTable.length; i++) {
10.            if (oldTable[i] != null && oldTable[i].key != null) {
11.                this.performPut(oldTable[i].key, oldTable[i].value); //a hodnoty do ni ulozime
12.            }
13.        }
14.    }
15.}
```

Metoda resize zajistí, že má tabulka správnou délku. Nejprve se zeptá metody calculateRequiredTableSize na ideální délku. Odpovídá-li navrácená hodnota současné délce tabulky, tak metoda terminuje. V opačném případě metoda vytvoří novou tabulku ideální délky a nechá do ní uložit všechny prvky stávající tabulky prostřednictvím metody performPut.

Metoda put

Java

view source

print?

```
01.public VALUE put(KEY key, VALUE value) {
02.    if (key == null) {
03.        throw new IllegalArgumentException("Klic nesmi byt null");
04.    }
05.    VALUE val = performPut(key, value);
06.    if(val == null){
07.        size++;
08.        resize();
09.    }
10.    return val;
11.}
```

Metoda put vloží do tabulky nový záznam. K tomuto využije pomocné metody performPut. Pokud její volání vrátí null, tak to znamená, že byl vložen nový záznam a je zapotřebí inkrementovat počet obsažených prvků a případně změnit velikost tabulky. V opačném případě došlo pouze k náhradě hodnoty u již existujícího záznamu, počet prvků se proto nemění. V každém případě metoda vrátí návratou hodnotu volání performPut (tj. případnou původní hodnotu záznamu).

Metoda remove

Java

view source

print?

```
01.public VALUE remove(KEY key) {
```

```
02.    Entry<KEY, VALUE> e = getEntry(key);
03.    if (e == null) { //prvek neexistuje
04.        return null;
05.    }
06.
07.    VALUE val = e.value;
08.    e.key = null; //prvek je nyni sentinelem
09.    e.value = null; //odstranime referenci na hodnotu, aby GC mohl cinit svou praci
10.
11.    size--;
12.    resize();
13.
14.    return val; //vratime puvodni hodnotu
15.}
```

Metoda remove odstraní prvek odpovídající danému klíči. Nejprve proto zjistí prostřednictvím volání getEntry, jestli záznam vůbec existuje. Pokud ne, tak metoda terminuje a vrací null.

Pokud záznam existuje, tak nastaví jeho klíč na null, čímž se z něj stane sentinel a zresetuje hodnotu na null, což umožní její sebrání garbage collectorem. Dále metoda sníží dekrementuje třídní proměnnou size, zajistí možnou změnu veliksoti tabulky a nakonec vrátí hodnotu, která byla uložena ve smazaném záznamu.

Metody get a contains

Java

view source

print?

```
01.public VALUE get(KEY key) {
02.    Entry<KEY, VALUE> e = getEntry(key);
03.    if (e == null) {
04.        return null;
05.    }
06.    return e.value;
07.}
08.
09.public boolean contains(KEY key) {
10.    return getEntry(key) != null;
11.}
```

Metoda get vrátí hodnotu asociovanou s předaným klíčem. Pro toto stačí získat odpovídající záznam voláním getEntry. V případě jeho existence pak metoda vrátí jeho hodnotu (value), v opačném případě null.

Implementace metody contains (dotaz na přítomnost) je pak ještě jednodušší. Jedná se pouze o kontrolu nenullovosti návratové hodnoty volání getEntry.

Metody keys a values

Java

view source

print?

```
01.public Collection<KEY> keys() {
02.    List<KEY> keys = new ArrayList<KEY>(size);
03.    for (int i = 0; i < table.length; i++) {
04.        if (table[i] != null && table[i].key != null) {
```

```
05.            keys.add(table[i].key);
06.        }
07.    }
08.    return keys;
09.}
10.
11.public Collection<VALUE> values() {
12.    List<VALUE> values = new ArrayList<VALUE>(size);
13.    for (int i = 0; i < table.length; i++) {
14.        if (table[i] != null && table[i].key != null) {
15.            values.add(table[i].value);
16.        }
17.    }
18.    return values;
19.}
```

Metody keys a values vytvoří dynamické pole (ArrayList) klíčů (nebo hodnot) a projdou tabulku. Ze záznamů, které nejsou sentinely pak vyberou odpovídající položku. Nakonec tento seznam vrátí.

Metoda size

Java

view source

print?

```
1.public int size() {
```

```
2.    return size;
3.}
```

Metoda size pouze vrátí odpovídající třídní proměnnou.

Výsledný kód

Kompletní kód hashovací tabulky:

Java

view source

print?

```
001./**
002. * Hashovaci tabulka
003. * @author Pavel Micka
004. * @param <KEY> typovy parametr klice
005. * @param <VALUE> typovy parametr hodnoty
006. */
007.public class HashTable<KEY, VALUE> implements TableIface<KEY, VALUE> {
008.
009.    /**
010.     * Pomer zaplneni pri kterem dojde k vytvoreni nove (vetsi) tabulky
011.     */
012.    private final float LOAD_FACTOR = 0.75f;
013.    /**
014.     * Pomer zaplneni pri kterem dojde k vytvoreni nove (mensi tabulky)
015.     */
```

```
016.    private final float COLLAPSE_RATIO = 0.1f;
017.    /**
018.     * Hodnota, pod kterou nikdy neklesne velikost tabulky
019.     */
020.    private final int INITIAL_CAPACITY;
021.    /**
022.     * Pocet ulozenych prvku
023.     */
024.    private int size = 0;
025.    private Entry<KEY, VALUE>[] table;
026.
027.    /**
028.     * Zkonstruuje hashovaci tabulku s vychozi kapacitou 10
029.     */
030.    public HashTable() {
031.        this(10); //vychozi kapacita
032.    }
033.
034.    /**
035.     * Zkonstruuje hashovaci tabulku
036.     * @param initialCapacity kapacita, pod kterou nikdy tabulka neklesne
037.     */
038.    public HashTable(int initialCapacity) {
```

```
039.        if (initialCapacity <= 0) {
040.            throw new IllegalArgumentException("Kapacita nesmi byt nulova nebo
zaporna");
041.        }
042.        this.INITIAL_CAPACITY = initialCapacity;
043.        this.table = new Entry[initialCapacity];
044.    }
045.
046.    public VALUE put(KEY key, VALUE value) {
047.        if (key == null) {
048.            throw new IllegalArgumentException("Klic nesmi byt null");
049.        }
050.        VALUE val = performPut(key, value);
051.        if(val == null){
052.            size++;
053.            resize();
054.        }
055.        return val;
056.    }
057.
058.    public VALUE remove(KEY key) {
059.        Entry<KEY, VALUE> e = getEntry(key);
060.        if (e == null) { //prvek neexistuje
```

```
061.            return null;
062.        }
063.
064.        VALUE val = e.value;
065.        e.key = null; //prvek je nyni sentinelem
066.        e.value = null; //odstranime referenci na hodnotu, aby GC mohl cinit svou praci
067.
068.        size--;
069.        resize();
070.
071.        return val; //vratime puvodni hodnotu
072.    }
073.
074.    public VALUE get(KEY key) {
075.        Entry<KEY, VALUE> e = getEntry(key);
076.        if (e == null) {
077.            return null;
078.        }
079.        return e.value;
080.    }
081.
082.    public boolean contains(KEY key) {
083.        return getEntry(key) != null;
```

```
084.    }
085.
086.    public int size() {
087.        return size;
088.    }
089.
090.    public Collection<VALUE> values() {
091.        List<VALUE> values = new ArrayList<VALUE>(size);
092.        for (int i = 0; i < table.length; i++) {
093.            if (table[i] != null && table[i].key != null) {
094.                values.add(table[i].value);
095.            }
096.        }
097.        return values;
098.    }
099.
100.   public Collection<KEY> keys() {
101.       List<KEY> keys = new ArrayList<KEY>(size);
102.       for (int i = 0; i < table.length; i++) {
103.           if (table[i] != null && table[i].key != null) {
104.               keys.add(table[i].key);
105.           }
106.       }
```

```
107.        return keys;
108.    }
109.
110.    /**
111.     * Vrati zaznam, ktery odpovida danemu klice
112.     * @return
113.     */
114.    private Entry getEntry(KEY key) {
115.        int index = key.hashCode() % table.length;
116.        //dokud nenarazime na volne misto, existujici zaznam pro klic nebo sentinel
117.        while (table[index] != null) {
118.            if (key.equals(table[index].key)) { //zaznam existuje
119.                return table[index];
120.            }
121.            index = (index + 1) % table.length; //prejdeme na dalsi adresu
122.        }
123.        return null; //nenalezen
124.    }
125.
126.    /**
127.     * Provede samotnou operace vlozeni, aniz by jakkoliv menil informaci o velikosti
128.     * pole (size), pripadne menil velikost pole (resize)
129.     *
```

```
130.     * Je-li klic jiz v tabulce obsazen, tak bude asociovana hodnota nahrazena
131.     *
132.     * @param key klic
133.     * @param value hodnota
134.     * @return byl-li klic v tabulce jiz obsazen, tak nahrazena hodnota, v opacnem pripade @null
135.     */
136.    private VALUE performPut(KEY key, VALUE value) {
137.        Entry<KEY, VALUE> e = getEntry(key);
138.        if (e != null) {
139.            //prvek je v tabulce
140.            VALUE val = e.value;
141.            e.value = value; //zamenime hodnoty
142.            return val;
143.        }
144.        int index = key.hashCode() % table.length;
145.        while (table[index] != null && table[index].key != null) { //dokud nenarazime na prazdne misto nebo sentinel
146.            index = (index + 1) % table.length; //posuneme se o adresu dal
147.        }
148.        if (table[index] == null) { //prazdne misto
149.            table[index] = new Entry<KEY, VALUE>();
150.        }
```

```
151.        table[index].key = key;
152.        table[index].value = value;
153.
154.        return null;
155.    }
156.
157.    /**
158.     * Vypocte velikost, jakou by mela tabulka mit
159.     * @return velikost, jakou by mela tabulka mit
160.     */
161.    private int calculateRequiredTableSize() {
162.        if (this.size() / (double) table.length >= LOAD_FACTOR) { //tabulka je preplnena
163.            return table.length * 2;
164.        } else if (this.size() / (double) table.length <= COLLAPSE_RATIO) {
165.            //vratime vetsi z hodnot SOUCASNA_VELIKOST/2 a INITIAL_CAPACITY
166.            return Math.max(this.INITIAL_CAPACITY, table.length / 2);
167.        } else {
168.            return table.length; //tabulka ma spravnou velikost
169.        }
170.    }
171.
172.    /**
173.     * Zmeni velikost tabulky, je-li to nutne
```

```
174.     */
175.    private void resize() {
176.        int requiredTableSize = calculateRequiredTableSize();
177.        if (requiredTableSize != table.length) { //pokud je treba zmenit velikost tabulky
178.            Entry<KEY, VALUE>[] oldTable = table;
179.            table = new Entry[requiredTableSize]; //tak vytvorime novou tabulku
180.            for (int i = 0; i < oldTable.length; i++) {
181.                if (oldTable[i] != null && oldTable[i].key != null) {
182.                    this.performPut(oldTable[i].key, oldTable[i].value); //a hodnoty do ni
ulozime
183.                }
184.            }
185.        }
186.    }
187.
188.    /**
189.     * Vnitrni trida reprezentujici zaznam tabulky
190.     */
191.    private class Entry<KEY, VALUE> {
192.
193.        /**
194.         * Klic, @null == prvek je sentinel
195.         */
```

```
196.        private KEY key;
197.        /**
198.         * Hodnota
199.         */
200.        private VALUE value;
201.    }
202.}
```