# Kolekce 10

Doposud, pokud jsme chtěli pracovat s nějakou kolekcí objektů, museli jsme použít pole nebo si implementovat vlastní datovou strukturu (třídu), která tyto objekty byla schopna pojmout. Velká výhoda polí je, že jsou efektivně implementována běhovým prostředím a je možné s nimi pracovat velmi rychle, avšak mají pevnou velikost, což komplikuje jejich použití v řadě běžných situací. Implementace vlastních datových struktur pak zdržuje od problémů, které má programátor řešit.

Avšak máme-li k dispozici tak silné a obecné nástroje jako jsou polymorfismus, dědičnost a generické třídy, dává smysl, aby někdo naprogramoval obecné třídy pro práci se soubory objektů a my je mohli rovnou používat, aniž bychom si je museli sami naprogramovat. Tyto třídy se souhrně označují jako kolekce a ve standardní knihovně jazyka Java (balíček `java.util`) je k dispozici několik základní typy kolekcí, např. *seznam*, *slovník*, *množina* nebo *fronta*. Všechny tyto kolekce podporují operace jako je vložení objektu, odstranění objektu z kolekce, ověření, zda-li je objekt součástí kolekce, a mnohé další.

**Poznámka 1** *Kolekce patří k základní výbavě programátora a jejich použití při programování je tak běžné a samozřejmé jako použití základních konstrukcí typu* `if` *nebo* `for`*. Avšak pro jejich správné pochopení, a tedy i použití, je důležité osvojit si polymorfismus a práci s generickými třídami, případně výjimkami. A právě proto, jsme se k tak základní věci (z pohledu programátora) dostali až v tomto semináři.*

## 1 Práce s kolekcemi

Kolekce v Javě jsou navrženy tak, že pro každý druh kolekce máme definované rozhraní (např. `List<T>` pro seznam) a pak existuje jedna nebo více tříd (např. `ArrayList<T>`), které toto rozhraní implementují. Při programování postupujeme tak, že u argumentů metody nebo pro návratový typ používáme rozhraní dané kolekce. Například následovně:

```
List<T> removeDuplicates(List<T> list) { /* ... */}
```

Díky takto použitým typům, můžeme do metody předat libovolný seznam, bez ohledu na to, o jakou třídu implementující seznam se jedná. A v případě hodnoty, kterou tato metoda vrací, můžeme volit libovolnou třídu, která implementuje rozhraní seznam a případně ji podle potřeby vyměnit.

**Poznámka 2** *Vedle rozhraní pro každý specifický druh kolekce existuje ještě zastřešující rozhraní* `Collection<T>`*, které zahrnuje obecné metody pro práci s kolekcemi, např.* `size()`*, pro zjištění počtu prvků v kolekci,* `add(T)` *a* `addAll(Collection<T>)` *pro vložení prvku do kolekce,* `contains(T)`*, pro test přítomnosti prvku v kolekci, a další.*

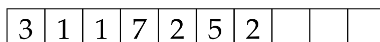
Nyní se zaměříme na práci s nejčastěji používánými kolekcemi a na jejich vlastnosti.

## 1.1 Seznam

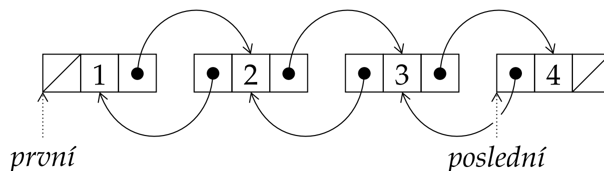
*Seznam* je kolekce, která je charakteristická tím, že jednotlivé hodnoty jsou uloženy v řadě za sebou, typicky tak, jak byly vloženy, a můžeme k těmto hodnotám přistupovat pomocí indexu. Ve srovnání s polem je práce se seznamem mnohem pružnější. Nejen že je „nafukovací”, můžeme navíc vložit prvek doprostřed seznamu a prvky napravo od něj se automaticky posunou, podobně je tomu i při odstraňování prvků.

Práci se seznamy popisuje rozhraní List<T>,[1] které má dvě prominentní implementace ArrayList<T>[2] a LinkedList<T>,[3] přičemž jejich typový parametr T udává typ hodnot, které bude možné do seznamu vložit.

Třída ArrayList<T> data uchovává ve stejném duchu jako seznam, který jsme si ukázali v předchozím semináři, tj. vnitřně obsahuje pole, které je podle potřeb zvětšeno. Tento způsob uložení ilustruje následující schéma představující uložení sedmi celočíselných hodnot do pole o deseti prvcích.




Třída LinkedList<T> ukládá hodnoty ve formě tzv. (dvousměrného) spojového seznamu. To znamená, že pro každou hodnotu, která je v seznamu uložena, si udržuje odkaz na předchozí a na následující hodnotu, jak demonstruje následující obrázek.




Objekty této třídy si navíc udržují ukazatel na začátek a konec seznamu, což umožňuje efektivně implementovat operace *vložení na začátek/konec seznamu*.

Když se podíváme na uspořádání hodnot, je patrné, že efektivita jednotlivých operací se bude u těchto dvou tříd lišit. Například metoda `ArrayList.get(int)`, která vrací hodnotu na dané pozici, bude rychlejší než `LinkedList.get(int)`, protože ta musí projít postupně jednotlivé prvky seznamu než se dostane na kýženou pozici. Naopak operace vložení prvku na začátek seznamu je v případě třídy `LinkedList` rychlejší, protože stačí vhodně přeskládat odkazy a není potřeba přesouvat všechny prvky v rámci pole, jak se to děje u třídy `ArrayList`.

Pro vložení a odstranění prvku ze seznamu máme k dispozici metody `add` a `remove`.

[1] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html

[2] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html

[3] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html

```java
List<String> names = new ArrayList<>();

// vlozi hodnotu na konec seznamu
names.add("alice");
names.add("bozena");
names.add("cyril");
names.add("david");
// [alice, bozena, cyril, david]

names.remove("bozena");
// [alice, cyril, david]

// vlozi hodnotu na pozici 1
names.add(1, "benedikt");
// [alice, benedikt, cyril, david]
```

Lze také získávat hodnoty a informace o hodnotách uložených v seznamu.

```java
names.size()             // 4
names.get(2)             // "cyril"
names.indexOf("david")   // 3
names.contains("david"); // true
names.contains("eva");   // false
```

**Poznámka 3** *Metody* `contains`, `indexOf` *nebo* `remove` *používají metodu* `equals` *k nalezení příslušné hodnoty v seznamu. Proto by objekty, které vkládáme do seznamu, měly mít vhodně definovanou metodu* `equals`.

Jednotlivé prvky seznamu lze procházet také pomocí konstrukce `for`-each stejně jako prvky pole.

```java
for (String name : names) {
  System.out.println(name);
}
```

Běžně se stává, že potřebujeme vytvořit seznam s pevně danými prvky. Pro tyto účely má rozhraní `List`, statickou metodu, která vytvoří neměnný seznam ze zadaných hodnot.

```java
List<Integer> numbers = List.of(1, 2, 3);
List<String> strings = List.of("abc", "cde");
List<Object> objects = List.of("abc", 123, true);
```

## 1.2 Slovník

Dalším často používaným typem kolekce je *slovník*. Slovníky uchovávají data jako dvojice *klíč-hodnota*, přičemž jeden klíč se může v kolekci vyskytovat nanejvýš jednou. Typickou operací je vložení dvojice *klíč-hodnota*, vrácení hodnoty odpovídající danému klíči nebo odstranění hodnoty s daným klíčem.

Slovníky popisuje rozhraní `Map<K,V>`,[4] které má dvě velmi používané implementace – `HashMap<K,V>`[5] a `TreeMap<K,V>`.[6] Ve všech případech typový parametr `K` udává typ hodnot, jež budou použity jako klíč a `V` udává typ hodnot.

**Poznámka 4** *Původ označení* `Map` *pochází z anglického „map" nebo „mapping" označující matematický pojem, kterému v češtině říkáme „zobrazení". Můžeme na to nahlížet tak, že objekty typu* `Map` *přirazují k jednotlivým klíčům hodnoty, resp. představují zobrazení z množiny klíčů do množiny hodnoty, přičemž každému klíči je přiřazena nejvýše jedna hodnota. Z tohoto důvodu, i když to k tomu může svádět, není dobré objektům typu* `Map` *říkat „mapy", protože tento typ nemá s mapami mnoho společného.*

### 1.2.1 Slovník jako hash tabulka

Třída `HashMap` uchovává hodnoty v tzv. *hash tabulce*. Hash tabulka je tabulka pevné velikosti, v níž jsou hodnoty uspořádány podle tzv. *hashovací funkce*. Tato funkce vezme atributy daného objektu a vrátí celé číslo. Toto číslo nemá žádný konkrétní význam ve vztahu k objektu. Jen je nutné, aby hashovací funkce pro dva stejné objekty (ve smyslu rovnosti atributů, resp. metody `equals`) vždy vrátila stejné číslo. A dále je žádoucí, aby hashovací funkce pro objekty s různými hodnotami atributů vracela různá čísla. Jak takové hodnoty vypadají například pro řetězce, ukazuje Tabulka 1, konkrétně její druhý sloupec.

| řetězec (s) | `s.hashCode()` | řádek v tabulce |
|---|---:|---:|
| `"methan"` | -1077555399 | 1 |
| `"ethan"` | 96834438 | 6 |
| `"propan"` | -979803440 | 0 |
| `"butan"` | 94105198 | 6 |

Tabulka 1: Příklad použití hashovací funkce

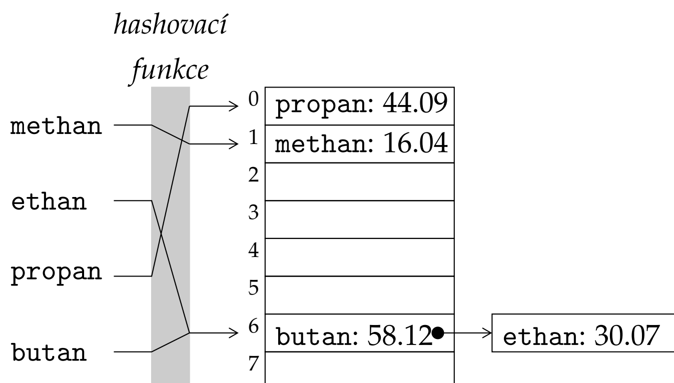
Jak je patrné, hodnoty hashovací funkce jsou vybírány z velkého rozsahu. Proto jsou následně tyto hodnoty přepočítány na řádky v tabulce. To lze elegantně udělat tak, že vydělíme hodnotu hashovací funkce počtem řádků tabulky a zbytek po celočíselném dělení nám dá příslušný řádek, kam uložit data. Přepočet na řádek v hash tabulce o velikosti osm řádků lze vidět ve třetím sloupci Tabulky 1.

Může se stát, že pro dva různé objekty vyjde, že mají být uloženy do stejného řádku tabulky. V našem příkladu to vyšlo pro řetězce `"ethan"` a `"butan"`. S touto eventualitou se počítá. V případě takové kolize se

[4] `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Map.html`
[5] `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html`
[6] `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeMap.html`

hodnoty „zřetězí" za sebe pomocí odkazů a s hodnotami se pracuje jako by byly uloženy v seznamu (nebo podobné vhodné struktuře) za sebou.

Jak by vypadala struktura slovníku, který je implementován pomocí hash tabulky, kde klíčem jsou řetězce představující názvy uhlovodíků a hodnotami jsou jejich molární hmotnosti (g/mol) ukazuje následující schéma.




Operace s hash tabulkou jsou rychlé, protože stačí z hashovací funkce odvodit řádek tabulky a pomocí indexu se na hodnotu v tabulce dotázat. Nevýhodou je, že s rostoucím zaplněním tabulky dochází k častějším kolizím a hash tabulka přestává být efektivní. Aby se tomuto problému předešlo, každý objekt třídy `HashMap` sleduje míru svého zaplnění, a pokud tato míra překročí určitou hranici, je hash tabulka zvětšena tak, aby měla více řádků.

Práci se slovníkem ilustruje následující kód.

```
Map<String, Double> g = new HashMap<>();
g.put("methan", 16.04);
g.put("ethan", 30.07);
g.put("propan", 44.09);
g.put("butan", 58.12);

g.get("ethan")          // ==> 30.07
g.get("hexan")          // ==> null (nepřítomen)
g.containsKey("butan") // ==> true
g.remove("methan");
```

Za povšimnutí stojí, že v tomto kódu nikde nefiguruje hashovací funkce. Odkud tedy objekty třídy `HashMap` berou hodnoty hashovací funkce? Slouží k tomu metoda `hashCode()` z třídy `Object`, má ji tedy každý objekt.

Třídy, které jsou běžně součástí Javy, mají metodu `hashCode()` dobře definovanou již v základu, není proto problém použít například hodnoty typu `String` nebo `Integer` pro klíče ve slovníku typu `HashMap`. Pokud bychom však chtěli jako klíč použít objekt vlastní třídy, musíme metodu `hashCode()` předefinovat. Pro naši třídu `Car` by metoda mohla vypadat třeba následovně.

```java
public int hashCode() {
  return plateNo.hashCode() ^ color.hashCode() ^ (7 * speed);
}
```

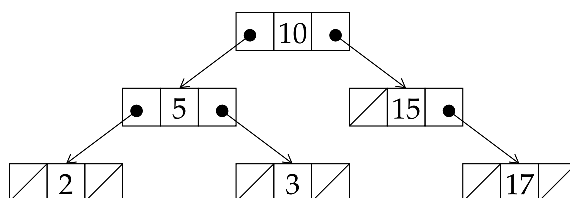
Z kódu je patrné, že hodnota hashovací funkce opravdu nemá žádný jasně popsatelný význam. Cílem je, aby pro různé hodnoty atributů byly výsledky co nejrůznější. Proto se v hashovací funkci často kombinují hodnoty hashovacích funkcí jednotlivých atributů, které jsou spojeny buď operací XOR nebo různými kombinacemi násobení a sčítání.

**Poznámka 5** *Abychom objekt mohli použít jako klíč v hash tabulce, je nutné, aby vedle metody* `hashCode()` *byla i dobře definována metoda* `equals`*. Jelikož vytvoření těchto dvou metod má svá úskalí a je to relativně nezáživná činnost, mají vývojová prostředí volby, které automaticky vygenerují tyto dvě metody. U tříd typu záznam (*`Record`*) jsou metody* `equals` *a* `hashCode` *vygenerovány automaticky.*

**Poznámka 6** *Aby se objekty třídy slovník chovali korektně, je nutné, aby objekty reprezentující klíče byly neměnné, nebo se alespoň neměnili. Je zde totiž implicitní předpoklad, že daný klíč tabulky bude mít vždy stejný hash. Pokud se změní atribut, změní se i hash objektu a nebude možné objekt dohledat.*

### 1.2.2 Slovník jako vyhledávací strom

Druhou často používanou třídou implementující rozhraní `Map<K,V>` je třída `TreeMap<K,V>`. Objekty této třídy uspořádávají dvojice *klíč-hodnota* do tzv. *binárního vyhledávacího stromu*. Binární vyhledávací strom si můžeme zjednodušeně představit jako soubor buněk (podobných těm, které používá `LinkedList`) s tím rozdílem, že levý odkaz ukazuje na buňky s menšími hodnotami a pravý odkaz ukazuje na buňky s většími hodnotami. Toto uspořádání hodnot ilustruje následující schéma.




Naše definice binárního vyhledávacího stromu není příliš přesná, podrobnější informace jsou třeba na Wikipedii.[7] Důležité je, abychom si utvořili představu, jak jsou záznamy uspořádány. Toto uspořádání má své výhody. Nalezení hodnoty podle klíče je efektivní a navíc, je-li to potřeba, lze si nechat jednoduše vypsat všechny hodnoty vzestupně seřazené.

Protože objekty třídy `TreeMap` mají stejné rozhraní jako objekty třídy `HashMap`, práce s nimi je velmi podobná. Avšak, aby bylo možné hodnoty uvnitř kolekce uspořádat do binárního vyhledávacího stromu, musíme uvést, jak jednotlivé klíče mezi sebou porovnávat. Slouží k tomu rozhraní `Comparable<T>`, které by měly objekty,

[7] `https://cs.wikipedia.org/wiki/Binrn_vyhledvac_strom`

které chceme použít jako klíče, implementovat.[8] Toto rozhraní má jedinou metodu `int compareTo(T o)`, která vrací

- záporné číslo, pokud je objekt *menší* než objekt `o`,
- kladné číslo, pokud je objekt *větší* než objekt `o`,
- 0, pokud je objekt roven objektu `o`.

Například, chtěli-li bychom jako klíče použít automobily z našeho příkladu a ve stromu je uspořádat podle rychlosti a v případě shodné rychlosti podle SPZ, použili bychom rozhraní `Comparable` následovně.

```
public class Car implements Comparable<Car> {
  /* ... */
  public int compareTo(Car o) {
    if (this.speed < o.speed) return -1;
    if (this.speed > o.speed) return 1;
    // jedou stejne rychle, proto porovname SPZ
    return this.plateNo.compareTo(o.plateNo);
  }
}
```

**Poznámka 7** *V tomto příkladu jsme využili toho, že třída* `String`*, stejně jako další třídy např.* `Integer`, `Double`*, implementuje rozhraní* `Comparable`*, a to tak, aby odpovídalo přirozenému uspořádání hodnot. U čísel je to uspořádání od nejmenšího čísla po největší, řetězce jsou řazeny lexikograficky (podle abecedy).*

U všech operací, které jsme si dosud uvedli, se uspořádání hodnot uvnitř objektu `TreeMap` nijak neprojeví a porovnání hodnot se používá primárně pro efektivní uložení hodnot v kolekci. Jednou z metod, kde se vnitřní uspořádání projeví, je metoda `keySet`, která vrací množinu všech klíčů použitých ve slovníku. S touto množinou lze dále pracovat, například procházet všemi prvky pomocí konstrukce `for`-each.

```
Map<String, Double> g = /* .... */;
for (String k : g.keySet()) {
  System.out.println(k + " => " + g.get(k));
}
```

Pokud je jako kolekce použit objekt třídy `HashMap`, nebudou klíče nijak uspořádány.

```
ethan => 30.07
butan => 58.12
propan => 44.09
methan => 16.04
```

[8] Alternativně je možné v konstruktoru dodat objekt implementující rozhraní `Comparator<T>`, který slouží k porovnávání jednotlivých klíčů.

V případě objektů třídy `TreeMap` však klíče budou uspořádány vzestupně.

```
butan => 58.12
ethan => 30.07
methan => 16.04
propan => 44.09
```

**Poznámka 8** *Pokud bychom chtěli mít kolekci typu slovník, která se chová jako hash tabulka, avšak zachovává pořadí prvků, jak byly vloženy do tabulky, nabízí standardní knihovna jazyka Java třídu* `LinkedHashMap<K, V>`*, která je však mírně pomalejší než* `HashMap<K,V>`*.*

Může se stát, že objekty, které chceme použít jako klíč, nemají rozhraní `Comparable`, nebo chceme klíče uspořádat jiným způsobem. Pro tyto účely je možné do konstruktoru předat objekt implementující rozhraní `Comparator<T>`, který má za úkol porovnávat objekty mezi sebou stejným způsobem jako to dělá rozhraní `Comparable<T>`. Ukažme si to na příkladu objektu, který bude porovnávat řetezce ne podle abecedy, ale podle jejich délky.

```
public class StringLengthComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    if (s1.length() < s2.length()) return -1;
    if (s1.length() > s2.length()) return 1;

    // jsou stejne dlouhe, proto je porovname
    // prirozene podle abecedy
    return s1.compareTo(s2);
  }
}
```

Tento objekt můžeme objektu třídy `TreeMap` předat v konstruktoru.

```
Map<String, Integer> m = new TreeMap<>(new StringLengthComparator());
m.put("abc", 10);
m.put("a", 2);
m.put("b", 3);
m.put("de", 15);

m.get("a") // 10
```

**Poznámka 9** *Třída* `TreeMap` *nabízí metody, které u obecného slovníku nedávají smysl, např.* `firstKey()` *a* `lastKey()` *vracející první a poslední klíč slovníku. Abychom se mohli v deklaraci metod vyhnout použití typu* `TreeMap` *máme k dispozici rozhraní* `SortedMap<K, V>`*, který tento typ tříd zastřešuje.*

Pokud bychom chtěli vytvořit slovník s pevně danými hodnotami, má rozhraní `Map` statickou metodu `of`, která pro zadané dvojice hodnoty vytvoří neměnný slovník, například následovně.

```
Map<Integer, String> numbers = Map.of(1, "one", 2, "two", 3, "three");
Map<String, Integer> romanNumerals = Map.of("i", 1, "v", 5, "x", 10, "l", 50, "c", 100, "d",
↪  500, "m", 1000);
```

### 1.3 Množina

Posledním typem kolekce, který si ukážeme, je *množina*. Pro množiny je charakteristické, že (na rozdíl od seznamu) nezachovávají pořadí vložených prvků a každý objekt se může v množině nacházet nanejvýš jednou. To je zcela v souladu s vlastnostmi množin, jak se používají v matematice. Práce s množinami je popsána rozhraním `Set<T>`.[9] Pro toto rozhraní existují dvě implementace `HashSet<T>`[10] a `TreeSet<T>`.[11] Tyto třídy přímo vycházejí z tříd `HashMap` a `TreeMap`, proto pro hodnoty, které lze vložit do množin typu `HashSet` a `TreeSet`, platí stejná pravidla jako pro klíče těchto slovníků. Pokud chceme vložit objekt do množiny typu `HashSet`, musí mít tento objekt korektně definované metody `hashCode` a `equals`. V případě třídy `TreeSet` musí prvky této množiny implementovat rozhraní `Comparable` nebo při vytváření množiny musíme dodat objekt s rozhraním `Comparator<T>`, který bude porovnávat jednotlivé prvky množiny. Práce s objekty třídy `TreeSet` je trochu pomalejší, ale výhodou je, že chceme-li prvky vypsat, či s nimi pracovat podobným způsobem, máme je již setříděny.

V matematice se obvykle pro množiny definují operace průnik, sjednocení a množinový rozdíl. V Javě, aby rozhraní pro práci s množinami bylo co nejpodobnější ostatním operacím s kolekcemi, mají metody odpovídající těmto operacím trochu jiné názvy. Metoda

- `addAll(s)` – přidá do množiny objekty z kolekce `s` (odpovídá sjednocení množin),
- `retainAll(s)` – ponechá v množině objekty, které jsou i v kolekci `s` (odpovídá průniku množin),
- `removeAll(s)` – odstraní z množiny objekty, které jsou v kolekci `s` (odpovídá rozdílu množin).

Hodnoty v množině lze také procházet pomocí konstrukce `for`-each stejně jako pole nebo seznamy.

```
Set<Integer> nums = new HashSet<>();
for (int i = 0; i < 10; i++)
  nums.add(i * 10);

for (int i : nums)
  System.out.print(i + " ");
// 0 80 50 20 70 40 10 90 60 30
```

[9] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Set.html

[10] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html

[11] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html

V tomto případě nejsou hodnoty na výstupu nijak uspořádány. Je to dáno tím, jak funguje hash tabulka, která je použita pro uložení prvků množiny. Pokud bychom použili `TreeSet`, budou hodnoty přirozeně uspořádány.

**Poznámka 10** *Alternativně bychom mohli použít třídu* `LinkedHashSet`*, kdy je pro reprezentaci množin použita hash tabulka, která zachovává pořadí vložených hodnot.*

Analogicky, jako v případě ostatních kolekcí, má rozhraní `Set` statickou metodu, která vytvoří neměnnou množinu na základě výčtu zadaných hodnot.

```
Set<Integer> numbers = Set.of(1, 2, 3);
Set<String> strings = Set.of("abc", "cde");
Set<Object> objects = Set.of("abc", 123, true);
```

### Komentář závěrem

Tento výčet tříd, které je možné pro práci s kolekcemi použít, není ani v nejmenším úplný a nečiní si na to ani nárok. Například jsme opomenuli frontu (rozhraní `Queue`) nebo oboustrannou frontu `Deque` a jejich implementace. Podobně jsme se nevěnovali všem metodám, které jednotlivé rozhraní a třídy nabízí. V principu se jedná o běžnou práci s objekty, a proto je to ponecháno p.t. čtenáři k samostudiu a k objevování při řešení praktických úkolů.