Základní datové struktury

Důležitá poznámka

Jazyk Java má implementováno mnoho datových struktur, jako jsou zásobník, fronta, seznamy, mapy a mnoho dalších. Obvykle je používáme hotové, nepotřebujeme je znovu vymýšlet. To, že v následujícím textu jsou uvedeny některé implementace (ostatně, maximálně zjednodušené), je jen pro snadnější pochopení principu, nikoliv pro praktické použití.

Přehled

Existuje ještě mnoho důležitých a zajímavých datových struktur. Jejich popis však vesměs překračuje rámec této stati. S některými se seznámíme na jiných místech, jak je uvedeno dále.

Některé datové struktury jsou popsány v tomto dokumentu. Jsou to:

- Dynamické pole
- Fronta
- Kruhová fronta (kruhový buffer)
- Zásobník

Další datové struktury jsou vysvětleny na jiných místech. Jedná se o následující struktury:

- Binární halda – viz kapitola Kolekce v tématu Java2,
- Binomiální halda – viz kapitola Kolekce v tématu Java2,
- Disjoint-set – viz kapitola Kolekce v tématu Java2,
- D-regulární halda – viz kapitola Kolekce v tématu Java2,
- Graf – viz téma Grafy,
- Hashovací tabulka – viz téma Hledání,
- Množina – viz kapitola Kolekce v tématu Java2,
- Multimnožina – viz kapitola Kolekce v tématu Java2,
- Spojový seznam – viz téma Seznamy,
- Strom – viz téma Stromy.
-

Dynamická změna velikosti pole

Dynamická změna velikosti pole není žádná datová struktura. Ale je to technologie, kterou při práci s datovými strukturami často používáme, a proto ji vysvětlím hned na začátku.

Dynamická změna velikosti je přípustná v jazyce Java. Mnoho jiných programovacích jazyků (např. Pascal) ji nedovoluje.

Dynamické pole umožňuje přidávat libovolný počet prvků, čímž eliminuje hlavní nevýhodu klasického pole – fixní velikost.

Dynamické pole ukládá své prvky do vnitřního pole fixní délky, v okamžiku, kdy je kapacita pole vyčerpána, dojde k alokaci nového většího pole (obvykle 2x většího z důvodu amortizace), všechny prvky původního pole se překopírují a staré pole se dealokuje.

Obdobným způsobem se kontejner zachová, je-li příliš prázdný (poměr neobsazenosti překročí určitou mez) - zaalokuje se přiměřeně menší pole a opět se do něj hodnoty překopírují.

Typické operace

Vkládání

Vkládání: Na první pohled je postup při vkládání prvku neefektivní, protože v okamžiku, kdy dojde místo, musí *array* veškerá data překopírovat. Asymptotická složitost vkládání prvku je proto O(n).

Pokud program běží dostatečně dlouho, tak se tato složitost vkládání amortizuje. To znamená, že se náklady na provedení této operace určitým způsobem v čase rozloží.

Díky amorizované složitosti je struktura dynamického pole velmi rychlá, ač by tomu asymptotická složitost jejích operací nenapovídala.

Na druhou stranu není její použití vhodné v *real-time* systémech, které musí garantovat určitou odezvu, protože vždy může dojít k onomu jednotlivému nepříznivému případu.

Mazání

Mazání: Složitost operace mazání prvku závisí na konkrétní implementaci. Pokud mazaný prvek pouze prohodíme s posledním prvkem a místo na konci kontejneru uvolníme (prohlásíme za neobsazené), tak asymptotická složitost bude O(1), ale dojde ke

zpřeházení prvků. Pokud odstranění provedeme tak, že všechny další prvky posuneme o jedno místo, bude asymptotická složitost O(n), ale pořadí prvků zůstane zachováno.

Čtení

Čtení: Čtení prvku na indexu *i* proběhne v O(1), protože pole umožňuje náhodný přístup.

Zásobník

Význam

Zásobník (*stack*) je jednou ze základních datových struktur, která se využívá například pro dočasné ukládání dat v průběhu výpočtu.

Zásobník data ukládá způsobem, kterému se říká **LIFO** - last in, first out - čili poslední vložený prvek jde na výstup jako první, předposlední jako druhý a tak dále.

Poznámka: opačným způsobem funguje datový typ fronta - FIFO - first in, first out.

Zásobník se v informatice používá zejména pro ukládání stavu algoritmů a programů. Je použit v prohledávání do hloubky a implicitně ve všech rekurzivních algoritmech.

Na zásobníkové architektuře jsou postaveny virtuální stroje pro jazyky Java a Lisp.

Základní operace

Abstraktní datový typ **zásobník** specifikuje tyto operace:

- push - vloží prvek na vrch zásobníku
- pop - odstraní vrchol zásobníku
- top - dotaz na vrchol zásobníku
- isEmpty - dotaz na prázdnost zásobníku (nebo size - dotaz na velikost zásobníku)

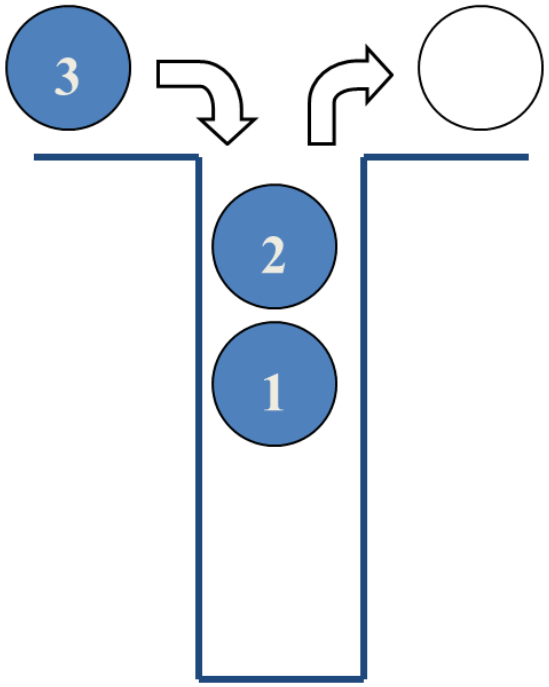


Figure 1 Stack

Pro manipulaci s uloženými datovými položkami se udržuje tzv. ukazatel zásobníku, který udává relativní adresu poslední přidané položky, tzv. **vrchol zásobníku**.

Ve většině vyšších programovacích jazyků může být zásobník poměrně jednoduše implementován pomocí pole, nebo lineárního seznamu.

To, co identifikuje datovou strukturu jako zásobník, není implementace, ale rozhraní - uživatel smí pouze přidávat (*push*) nebo ubírat (*pop*) hodnoty z pole či lineárního seznamu, spolu s několika málo pomocnými funkcemi.

Implementace pomocí pole

Implementace pomocí pole se zaměřuje na vytvoření pole, kde se na nulovém indexu (*array[0]*) nachází dno zásobníku.

Program si do pomocné proměnné ukládá, na jakém indexu se nachází vrchol zásobníku - tento index se mění podle toho, jak obsah zásobníku roste, nebo se zmenšuje.

Příklad

Následuje ukázka, jak by šlo vytvořit zásobník pomocí pole. Pole je dynamické v tom smyslu, že kdybychom potřebnou kapacitu zásobníku na začátku neodhadli správně, podle potřeby si vyrobíme pole nové, větší. V Javě to jde, na rozdíl od mnoha jiných jazyků.

Podívejme se na několik zajímavostí, které jsou v příkladu obsaženy. Především je respektována objektová struktura: zásobník je vytvořen jako úplně samostatná třída *StackArray*. Hlavní program vytvoří její instanci (jmenuje se *demo*) a potom opakovaně volá její metody *push()* a *pop()*. Zde je detail:

```
public static void main(String[] args) {

    StackArray<String> demo =
           new StackArray<>();

    demo.push("Amsterdam");
    demo.push("Bonn");
    demo.push("Cairo");
    demo.push("Detroit");
    demo.push("El Dorado");

    System.out.println(demo.pop());
    System.out.println(demo.pop());
    System.out.println(demo.pop());
    System.out.println(demo.pop());
    System.out.println(demo.pop());
}
```

Na první pohled si všimneme divné věci: za jménem typu „*StackArray*“ se nachází výraz `<String>`. Zatím jsme se nikdy nesetkali s názvem typu, který by byl uzavřený do špičatých závorek `< >`. Jedná se o tak zvané generické typy. Podrobně je budeme probírat v jiném předmětu.

Pro snadnější pochopení nám snad pomůže následující přirovnání. Když pracujeme s metodami (s funkcemi), tak jim v závorkách předáváme parametry. Když funkci definujeme, tak nějakými jmény pojmenujeme parametry, se kterými budeme provádět výpočty. Tomu se říká formální parametry. Potom, když funkci voláme, tak na místa formálních parametrů dosadíme konkrétní hodnoty, skutečné parametry.

Důležité je, že formálními i skutečnými parametry jsou u metod **proměnné**. Jako příklad si umíme představit funkci `int add(int i, int j){…}` která sečte dvě celá čísla. Podobně bychom mohli mít funkci `double add(double i, double j){…}` pro

proměnné typu *double* a jinou funkci `String add(String i, String j){…}` pro „sčítání“ hodnot typu *String*.

V Javě jsme to řešili pomocí přetížení funkcí.

Existuje ale ještě jiný způsob. Podobně jako jsme měli formální a skutečné parametry, můžeme také zavést formální a skutečné typy. Tyto typy budeme uzavírat do špičatých závorek. Naši funkci *add* bychom tedy mohli napsat třeba `<T> add(<T> i, <T> j){…}` . Zde <*T*> je zástupný symbol pro nějaký typ. Později, při použití funkce, ho nahradíme nějakým skutečným typem, třeba *int*, *double* nebo *String*.

Zpátky k zásobníku. V souboru *StackArray.java* je symbolem `<E>` označen typ dat, která se budou ukládat do zásobníku. Dále *DEFAULT_CAPACITY* je celočíselná konstanta, která udává počáteční délku zásobníku. Pole *elements* se na začátku vytvoří o délce 10 a kdykoliv to je potřeba, tak metoda *doubleSize()* jeho délku zdvojnásobí. Povšimněte si, že pole je definováno jako pole prvků typu Object. To je v pořádku, protože typ Object je výchozím předkem pro každou třídu. Ale stejně tak dobře jsme pole mohli definovat pomocí formálního typu <E>, ušetřili bychom si nutnost přetypování (například na řádku 51 je `E e = (E) elements[--size];`).

Funkce *push()* je použita k inicializaci zásobníku a zároveň přidání nové hodnoty na jeho vrchol.

Je zodpovědná za změnu hodnoty proměnné *size* a za vložení nového prvku do pole `elements[size++] = e;` . Funkce také ověřuje, jestli pole není plné; pokud by se pole naplnilo a tato kontrola by zde nebyla, vedlo by to k chybě.

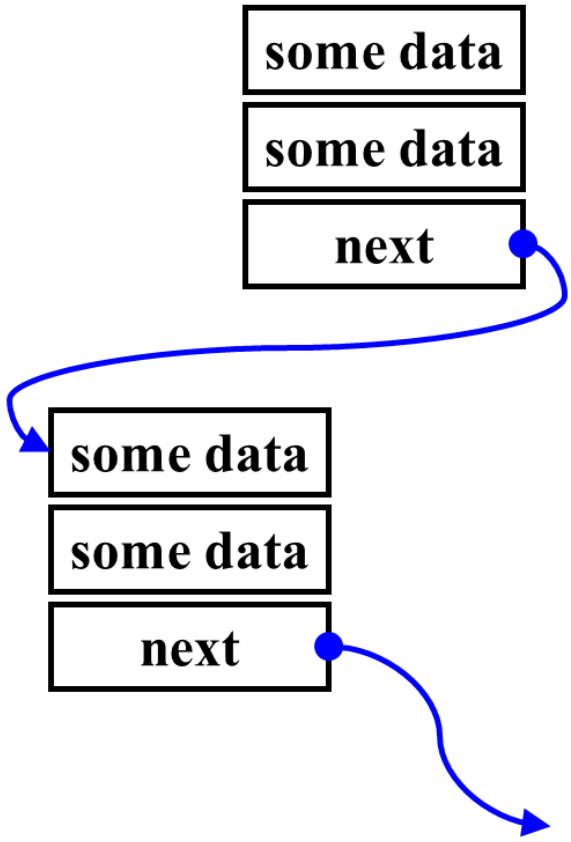
Funkce *pop()* odstraňuje prvek z vrcholu zásobníku a dekrementuje počet prvků.

Dále ověřuje jestli zásobník není prázdný - pokud by zde tato kontrola nebyla, tak by pokus u vyjmutí prvku z již prázdného zásobníku pravděpodobně vedl k chybě.

Zdrojové texty jsou v tomto souboru.

Implementace spojovým seznamem

Zásobník samozřejmě lze implementovat i pomocí spojového seznamu, jak bude vysvětleno dále. Není to ale nic mimořádně výhodného.



Zde je ukázka klíčových částí kódu. Základem je interní třída Node, která funguje jako datový kontejner. Obsahuje užitečná data (*value*), **referenci** na jinou instanci téže třídy (*next*) a případně ještě konstruktor *Node()*.

```
    private class Node {
        private String value;
        private Node next;
        private Node(String value) {
            this.value = value;
        }
    }
```

Vlastní zásobník má ukazatel *first* na první prvek zásobníku, celočíselný počet prvků *size* a konstruktor, kterým se *size* vynuluje.

```
public class StackList {
    private Node first;
    private int size;
    public StackList() {
```

```
        this.size = 0;
    }
```

Funkci si můžeme ukázat na příkladu metody *push*, která vloží nový prvek na vrchol zásobníku.

```
    public void push(String i) {
        Node n = new Node(i);
        Node currFirst = first;
        first = n;
        n.next = currFirst;
        size++;
    }
```

Metoda vytvoří nový uzel *Node* a hned v konstruktoru naplní jeho hodnotu. Přes pomocnou proměnnou *currFirst* tento prvek zapojí do seznamu a nakonec zvýší počet prvků o 1.

Zdrojové texty jsou v tomto souboru.

Fronta

Význam

Fronta je jedním ze základních datových typů a slouží k ukládání a výběru dat takovým způsobem, aby prvek, který byl uložen jako první, byl také jako první vybrán.

Tomuto principu se říká **FIFO** - first in, fist out.

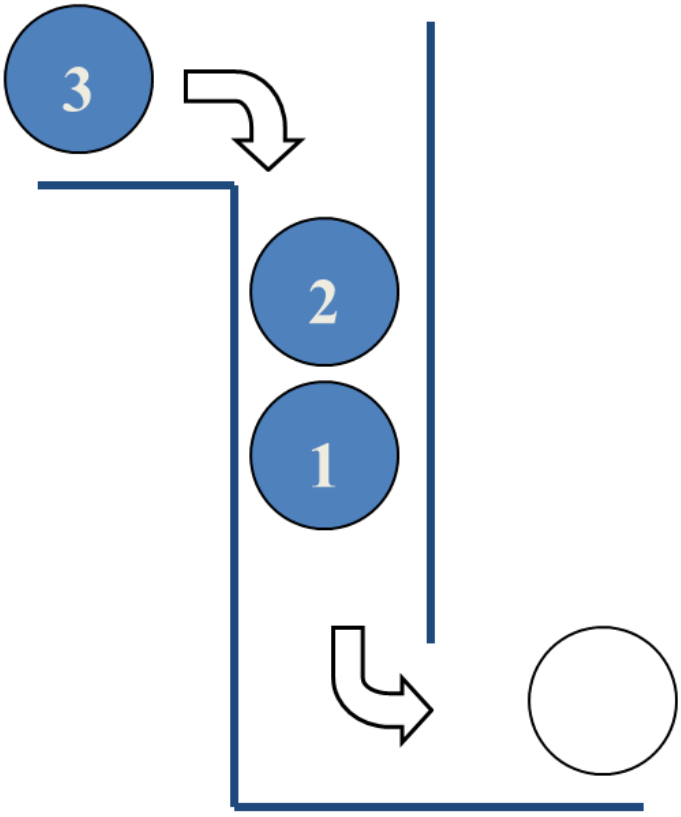
Poznámka: opačný přístup - LIFO - last in, first out - používá datový typ zásobník.

Speciálním případem fronty je tzv. **prioritní fronta**, ve které mohou prvky s vyšší prioritou předbíhat na výstupu ty s nižší prioritou.

Fronta má v informatice široké využití, toto jsou některé příklady:

- Operátor roura (*pipe*) - komunikace mezi procesy v operačních systémech
- Kruhový buffer - vyrovnávací paměť pro datové toky
- Řazení prioritní frotou (haldou) - heapsort

Typické operace



Zde je několik operací, které realizuje typická fronta:

- *addLast* (enqueue) – Vloží prvek do fronty.
- *deleteFirst* (poll, dequeue) – Získá a odstraní první prvek (hlavu) fronty.
- *getFirst* (peek) – Získá první prvek fronty.
- *isEmpty* – Dotaz na prázdnost fronty.
- *size* – Vrátí počet obsažených prvků

Při implementaci polem, fronta jako taková má velmi omezené možnosti použití, protože fronta neustále narůstá a nikdy se nezmenšuje. Proto se spíše používá kruhová fronta (kruhový buffer), která tuto vadu nemá.

Nicméně, implementace fronty pomocí seznamu se používá často, protože je jednoduchá a rychlá. Je velmi podobná implementaci zásobníku. Úplný zdrojový kód je zde.

Kruhová fronta

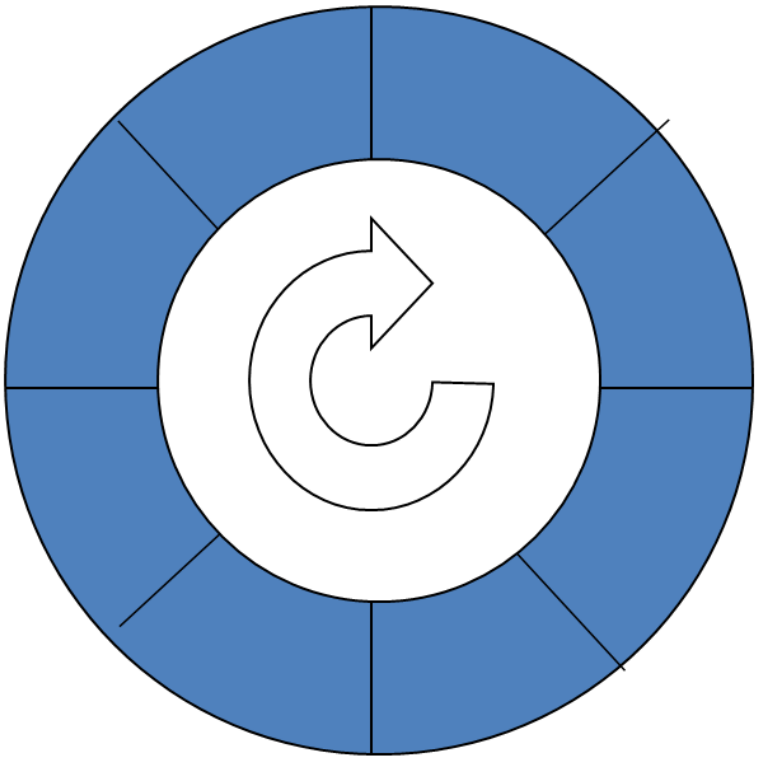
Jako úložný prostor pro frontu můžeme v nejjednodušším případě použít obyčejné pole (array), odborně se tomuto případu říká implementace pomocí pole. Při běhu se do položek pole umisťují data jedno za druhým, tj. postupně do položky 1, 2, 3 atd.

Zároveň se data čtou, opět postupně.

Pole při tomto řešení má omezenou délku (definujeme proměnné na začátku programu ve formě např. `int [1..1000]`, takové pole může mít maximálně 1000 míst pro uložení dat ve frontě).

Tím, jak data do pole zapisujeme a čteme, posouváme čelo fronty až ke konci pole. Tam se zastaví.

Jednoduchým řešením by bylo, při každém přečtení všechny položky v poli posunout na začátek, to ovšem není nejlepší, především u velkých polí s množstvím dat.



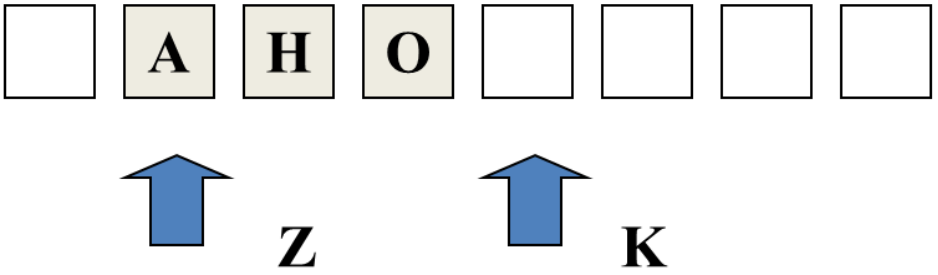
Proto se používá tzv. **kruhová fronta** neboli kruhový buffer. Ta, když dojde zápis na konec pole, zapisuje opět na volné místo na začátku.

S výhodou použijeme operátor **modulo**.

Nesmíme nikdy zapomenout na důležitost kontroly fronty. Jestliže místo, ze kterého čteme, je prázdné, nebo když místo, do kterého zapisujeme, je plné (obsahuje ještě nepřečtená data), následuje podtečení nebo přetečení fronty.

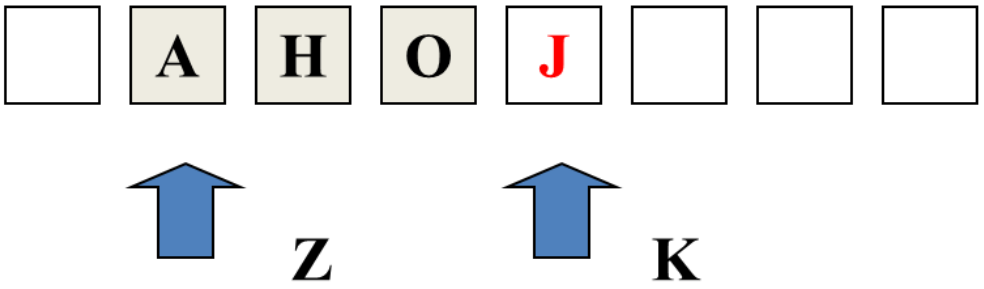
Zavedeme si dvě proměnné:

- **K** bude obsahovat index na to místo pole, kde ještě nejsou zapsaná data (tzn. bude ukazovat za konec dat),
- **Z** bude obsahovat index na to místo pole, kde jsou umístěna data, která jsme ještě nepřečetli (tzn. na začátek dat).

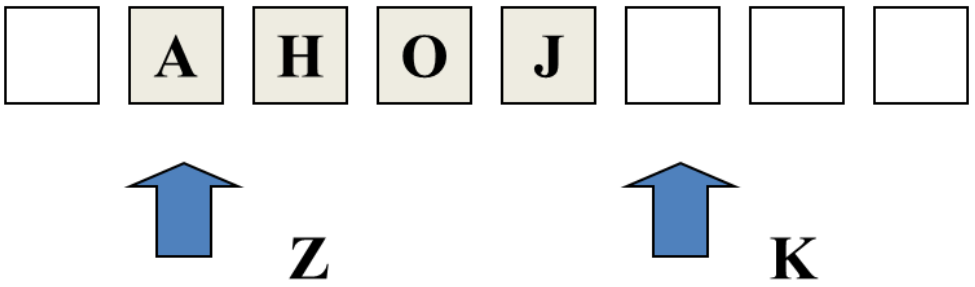


Při zápisu budeme postupovat takto:

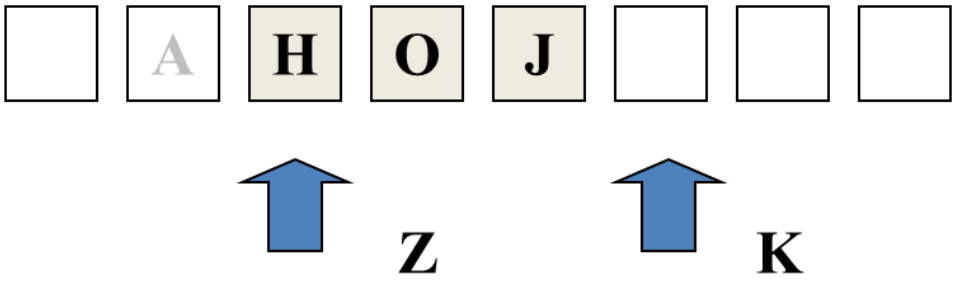
- nejdříve se zkontroluje, jestli už nový konec K nedostihne začátek Z
- tam, **kam ukazuje K**, se provede zápis
- pak se **K posune** na následující volné místo



A fronta je připravena k dalšímu zápisu.



Při čtení se čte z místa, kam ukazuje Z a po přečtení se Z posune na další pozici. Přitom nutno kontrolovat podtečení (tzn., aby posunuté Z nebylo rovno K).



Implementace pomocí pole

Kruhovou frontu lze implementovat pomocí pole.

Zdrojový text je obsažen v tomto souboru. Zde uvádím jen některé klíčové detaily:

```
public class CircularQueueArray<TYPE> {

    private int size;
    private final Object[] array;
    private int pointer; //first free index

    /**
     * Constructor
     * @param length initial size of array
     */
    public CircularQueueArray(int length) {
        this.array = new Object[length];
        this.size = 0;
        pointer = 0;
    }

    /**
     * Append item
     * @param i item
     */
    public void addLast(TYPE i) {
        if (this.size == array.length) {
            throw new IllegalStateException("Buffer full");
        }
        array[pointer] = i;
        pointer = modulo((pointer + 1), array.length);
        size++;
```

```
    }
```

Opět je použit formální typ *<TYPE>* a opět je použito dynamické prodlužování pole, jak to už známe z dřívějška.

Další důležité datové struktury

Mezi důležité datové struktury, které probíráme v tomto předmětu, ještě patří:

- Spojový seznam – viz téma Seznamy,
- Strom – viz téma Stromy.

Obsah