Fronta a zásobník

Fronta i zásobník jsou 2 principiálně velmi podobné způsoby dočasného uložení dat v rámci programu. Používají se v případech, kdy je třeba postupně projít všechny prvky dat, a to právě jednou.

Základní funkce

- **push(objekt)** – vkládá nová data do řady
- **pop()** nebo také **pull()** – získá data, která jsou právě na řadě

Fronta (queue)

Funguje na principu **FIFO** (first in first out), to znamená, že si frontu můžeme představit třeba jako frontu na úřadu práce. Lidé (data) se řadí po příchodu do fronty a ve stejném pořadí se i dostanou na řadu (data jsou tedy dříve vrácena).

Jsou potřeba dva ukazatele: front (zde jsou odebírána data) a rear (zde jsou data vkládána)

Metody Implementace

Pomocí statického pole

Existují dva způsoby statické implementace – klasická a cyklická frotna:

U obou variant začínáme tím, že si vytvoříme danou frontu jako **pole**. V rámci funkce push(objekt) pak postupně vkládáme data do pole pomocí ukazatele rear a funkcí pop() získáváme data pomocí ukazatele front. Nyní se dva zmíněné způsoby implementace začínají lišit:

klasická fronta : Při volání funkce pop() získáme data, snížíme hodnotu rear ukazatele a všechny položky posuneme o jedno dopředu.

zde ukázka klasické varianty statické implementace fronty v jazyce Java:

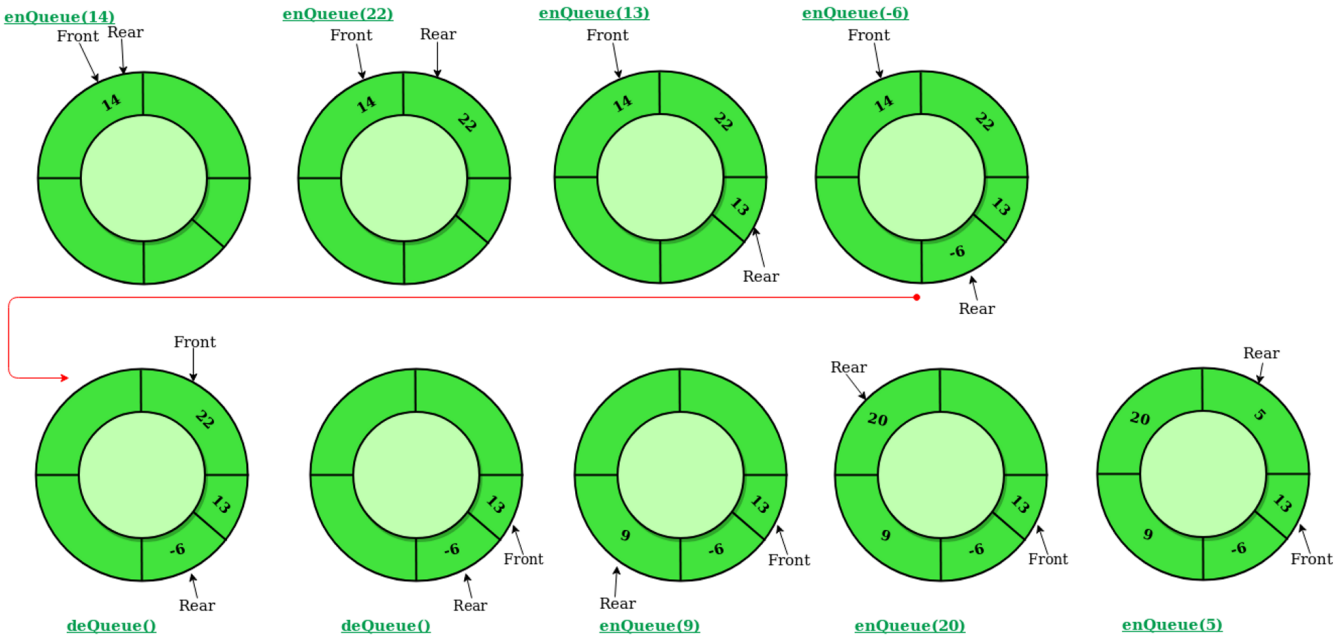
```java
// A simple demo of static queue implementation through an array by Michal Bureš

public class MainClass {

    final int queueSize = 5; // choose queue array size

    Object[] simpleStaticQueue = new Object[queueSize];
    final int pointerFront = 0;
    int pointerRear = 0;

    public MainClass() {
        enQueue(7);
        enQueue(8);
        enQueue("nine");

        System.out.println(deQueue());
        System.out.println(deQueue());
        System.out.println(deQueue());
        System.out.println(deQueue());
    }

    // push function – inserts the object to the queue using the rear pointer
    final void enQueue(Object toInsert) {
        simpleStaticQueue[pointerRear] = toInsert;
        pointerRear++;
    }

    // pull(pop) function – retrieves the object of queue using the front pointer
    final Object deQueue() {
        Object toReturn = simpleStaticQueue[pointerFront];
        for (int i = 0; i < queueSize; i++) {
            if (i + 1 <= queueSize - 1) {
                simpleStaticQueue[i] = simpleStaticQueue[i + 1];
            }
        }
        pointerRear--;
        return toReturn;
    }

    public static void main(String[] args) {
        new MainClass();
    }
}
```

V rámci konstruktoru vkládáme do fronty čísla 7 a 8 a text „nine“, když poté čtyřikrát voláme funkci pop() – zde metoda deQueue() a necháváme vypsat její výstup, finální výstup vypadá takto:

```
run:
7
8
nine
null
BUILD SUCCESSFUL (total time: 0 seconds)
```

cyklická fronta : Při volání funkce pop() opět získáme data, ale tentokrát hodnotu rear ukazatele ponecháváme a naopak se zvyšuje hodnota ukazatele front. Celá fronza je navíc skutečně cyklická, takže v situaci kdy se uvolní prostor z původního „začátku“ fronty, může být znova zaplněn na jejím konci.

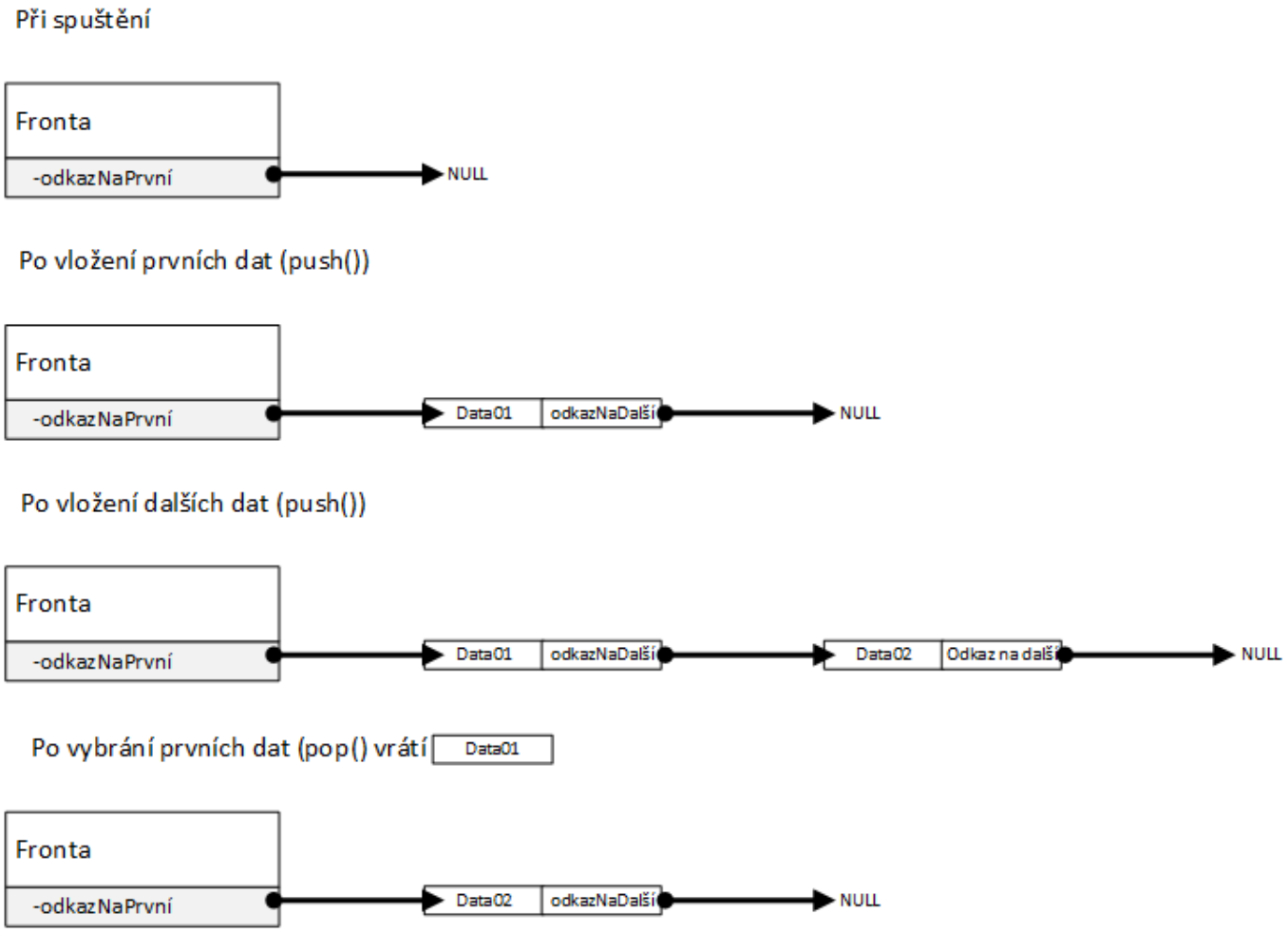
viz. diagram:



Nevýhodou statické implementace je, že jsme **omezeni velikostí** námi vytvořeného pole. Výhodou zase naopak je, že za všech okolností **víme, kolik máme na data místa**.

Pomocí ukazatelů

Metodu řešení pomocí **ukazatelů** můžeme vidět na níže uvedeném diagramu. Výhoda tohoto řešení je zároveň i jeho nevýhodou – jsme už sice schopni přidávat do fronty příspěvky až do konce paměti, zároveň ale (při špatné implementaci) může dojít k **přehlcení nebo až přetečení paměti**, což může vést k nestabilitě systému.



Zde můžeme vidět implementaci pomocí ukazatelů v jazyce Java a to konkrétně s využitím kolekce LinkedList. Všimněme si též těla metody deQueue(), kde musíme zachytávat vyjímku chybějícího ukazatele, je-li fronta prázdná.

```java
import java.util.LinkedList;
import java.util.NoSuchElementException;

// A simple demo of dynamic queue implementation through a LinkedList utility by Michal bureš
public class MainClass {

    LinkedList dynamicQueue = new LinkedList();

    public MainClass() {
        enQueue(7);
        enQueue(8);
        enQueue("nine");

        System.out.println(deQueue());
        System.out.println(deQueue());
        System.out.println(deQueue());
        System.out.println(deQueue());
    }

    // push function – inserts the object to the queue using a LinkedList method
    final void enQueue(Object toInsert) {
        dynamicQueue.addLast(toInsert);
    }

    // pull(pop) function – retrieves the object of queue using a LinkedList method
    final Object deQueue() {
        Object toReturn = null;
        try {
            toReturn = dynamicQueue.getFirst();
            dynamicQueue.removeFirst();
        } catch (NoSuchElementException ex) {}
        return toReturn;
    }

    public static void main(String[] args) {
        new MainClass();
    }
}
```

Zásobník (stack)

Funguje na principu **LIFO** (last in first out), to znamená, že funguje stejně **jako zásobník v samopalu**. Při nabíjení samopalu (vkládání dat) skládáme vždy náboje na spodek zásobníku a vršíme je na sebe, při výstřelu (výběru dat) se náboje berou svrchu – poslední vložený je tedy vystřelen jako první.

Metody Implementace

Pomocí statického pole

Řešení zásobníku pomocí statického pole je stejné jako u **fronty**, jediný rozdíl je v tom, že si v další proměnné musíme držet index posledních vložených dat a při zavolání funkce pop() vezmeme data z indexu posledních vložených data a index snížíme o jedna.

Výhody a nevýhody jsou stejné jako u fronty.

Pomocí ukazatelů

Řešení pomocí ukazatelů je také podobné jako u **fronty** (viz diagram níže).

Výhody a nevýhodu jsou opět naprosto stejné jako u fronty.