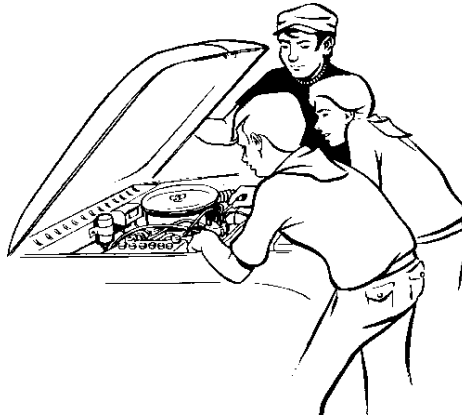




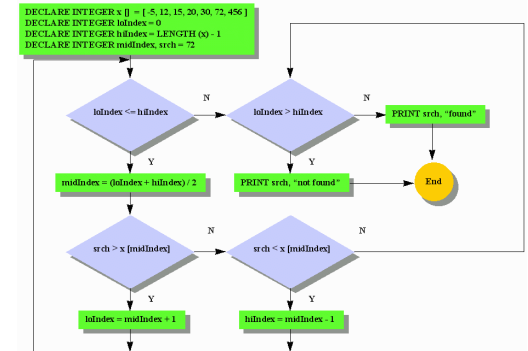
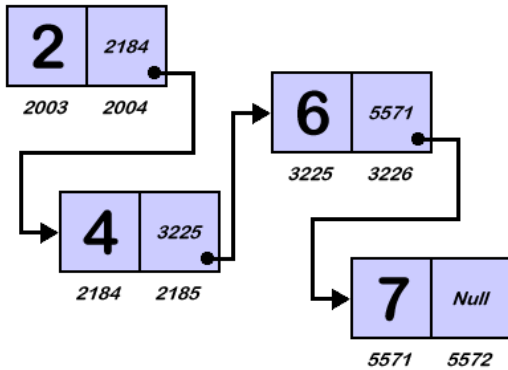
3. prednáška



Údajové štruktúry

alebo

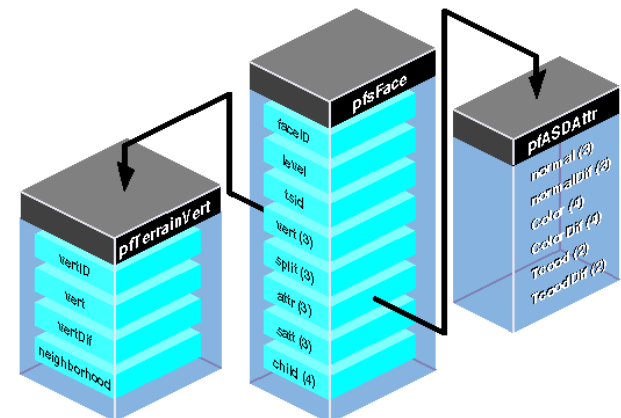
čo sa skrýva
pod kapotou JCF





Plán na dnes

- Čo sa skrýva v ArrayList-e?
- **Spájané zoznamy** (alebo čo sa skrýva v LinkedList-e)
- Ďalšie užitočné údajové štruktúry:
 - **Zásobník** a jeho použitie
 - **Rad** je jeho použitie
- **Prehľadávanie bludiska**





Kam ukladať veľa údajov?

- Na uloženie väčšieho množstva údajov sme sa naučili používať:

- **polia** (`Udaj[] udaje = new Udaj[100]`)
- **kolekcie** z Java Collections Framework-u
 - rozhranie **List** a jeho implementácie `ArrayList<E>`, `LinkedList<E>`
 - rozhranie **Set** a jeho implementácie `HashSet<E>`, `TreeSet<E>`, `LinkedHashSet<E>`



- JCF poskytuje
 - **rozhrania** (čo to má robiť)
 - **implementácie** (ako to má robiť)



Spomienky na List-y

- Implementácie rozhrania **List<E>**
 - ukladajú hodnoty za sebou, sú prístupné indexom
 - akoby „dynamické“ pole - vieme meniť počet prvkov
- Rozhranie **List<E>** predpisuje základné metódy na prácu so zoznamami:
 - **add** - pridanie na koniec, resp. na zadanú pozíciu
 - **remove** - odstránenie prvku na zadanej pozícii
 - **get** - získanie prvku na zadanej pozícii
 - **set** - nastavenie prvku na zadanej pozícii





Ako funguje ArrayList?

- **ArrayList<E>**

- interne na uloženie hodnôt využíva pole:

```
public class ArrayList<E> implements List<E> {  
    private E[] prvky = new E[0];  
}
```

Interné pole hodnôt

- **Veľkosti polí meniť nemožno:**

- pri každom add a remove sa vytvára nové pole





Aký rýchly je ArrayList?

- Koľko trvá (koľko krokov vykoná) metóda `add`?

```
public boolean add(E hodnota) {  
    E[] novePrvky = (E[])new Object[prvky.length+1];  
  
    for (int i=0; i<prvky.length; i++)  
        novePrvky[i] = prvky[i];  
    novePrvky[novePrvky.length-1] = hodnota;  
  
    prvky = novePrvky;  
}
```

Ak máme v ArrayList- n prvkov, operácia `add` má časovú zložitost' $O(n)$



Ako na rýchlejší ArrayList?

- Nahradenie for-cyklu volaním `System.arraycopy` 

- to nepomôže, pretože kopírovací cyklus s časovou zložitost'ou $O(n)$ je skrytý v metóde `arraycopy`

- Nemeniť veľkosť interného poľa vždy 

- veľkosť poľa budeme zväčšovať/zmenšovať vždy o 100 prvkov (resp. iný vhodný počet prvkov)

- zoznam má:

- **veľkosť** (size) - počet uložených hodnôt
- **kapacitu** (capacity) - veľkosť interného poľa, z ktorého prvých size políčok obsahuje „platné“ hodnoty

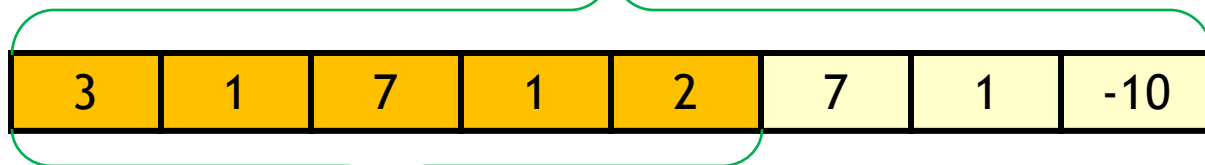




ArrayList s kapacitou (1)

kapacita = 8

kapacita = `prvky.length`

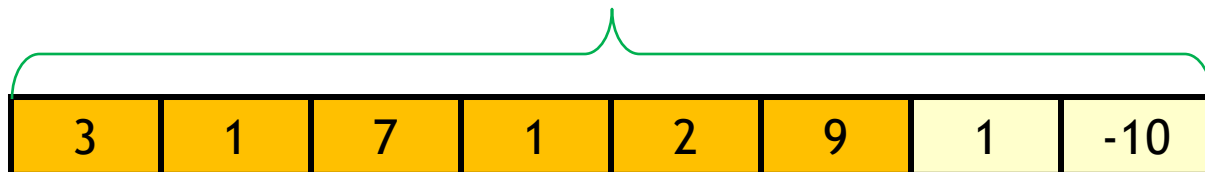


velkost' = 5

Po vykonaní `add(9)`

kapacita = 8

`prvky[velkost] = hodnota;`
`velkost++;`



velkost' = 6

velkost' = počet uložených prvkov



ArrayList s kapacitou (2)

- Výhody poľa s kapacitou
 - ak kapacita stačí, **add (remove)** na koniec zoznamu vieme realizovať v čase $O(1)$
 - zvýšenie premennej s veľkosťou (size) o 1
 - uloženie hodnoty na príslušný index interného poľa
 - ak kapacita nestačí, musíme vyrobiť nové pole a kopírovať, t.j. časová zložitost' je $O(n)$
- Ak je veľa neobsadených políčok, zmenšíme pole
- Nastavovanie kapacity v **ArrayList**-e cez metódu ***ensureCapacity***



ArrayList s kapacitou (3)

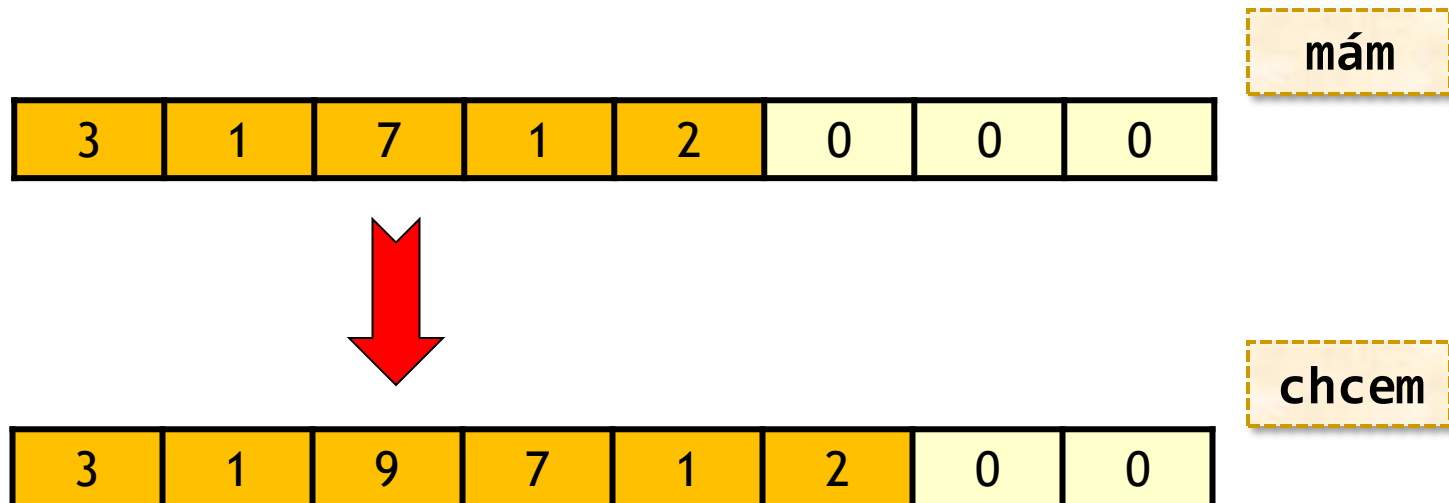
- Sumarizácia časovej zložitosti ArrayList-u:
 - get $O(1)$, set $O(1)$
 - add na koniec:
 - väčšinou $O(1)$
 - niekedy $O(n)$, keď nová veľkosť má byť väčšia ako kapacita
 - remove z konca:
 - väčšinou $O(1)$
 - niekedy $O(n)$, keď máme príliš „neobsadených“ políček

A čo add a remove na začiatku alebo v strede?



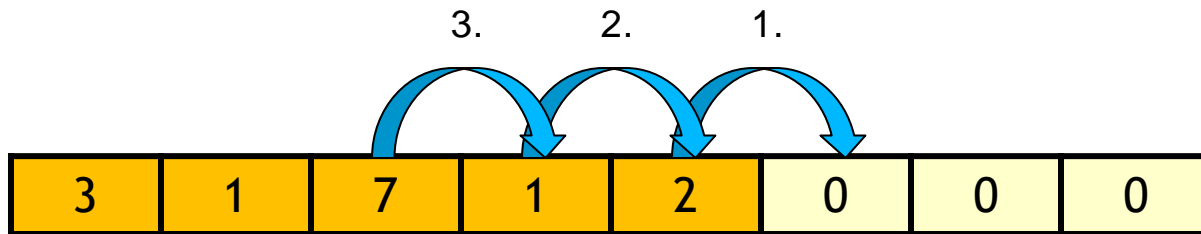
ArrayList s kapacitou (4)

- Za predpokladu, že máme dostatočnú kapacitu, aká je zložitosť `add(2, 9)`, t.j. pridania hodnoty 9 na index 2?





ArrayList s kapacitou (5)



add(2, 9)

index

hodnota

```
for (int i=velkost-1; i>=index; i--)
    prvky[i+1] = prvky[i];
prvky[index] = hodnota;
```

Pozor na poradie kopírovania hodnôt

Po posunutí (vytvorenie miesta):



Po uložení hodnoty na index 2:





ArrayList s kapacitou (6)

- Za predpokladu, že máme dostatočnú kapacitu, aká je zložitosť $\text{add}(0, x)$, t.j. pridania hodnoty na 0-tý index?
 - všetky hodnoty v internom poli musíme vždy posunúť o jedno miesto vpravo
 - posun n hodnôt v poli trvá $O(n)$
 - **záver:** kapacita pomôže pri pridávaní na koniec, ale nie pri pridávaní na iné pozície



Výzva

Demo

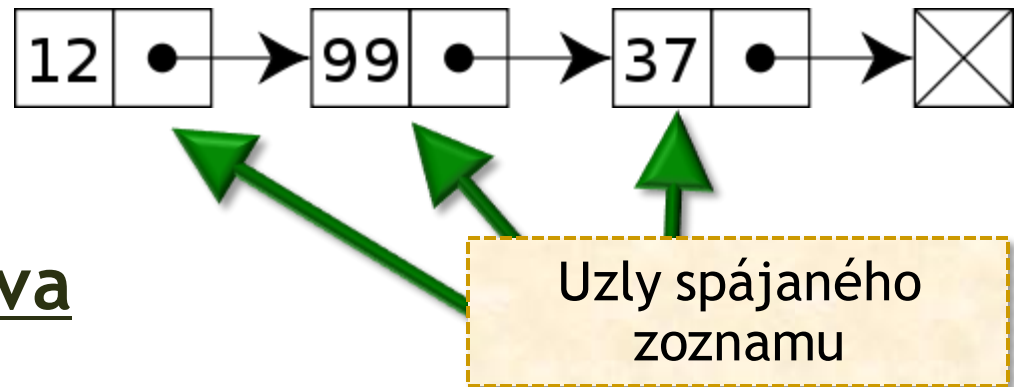
Ide uložit' zoznam hodnôt tak, aby sme **nemuseli** robiť v pamäti **presuny** pri každom vložení novej hodnoty do tohto zoznamu?





Spájaný zoznam

- **údajová štruktúra** na uloženie zoznamu hodnôt
 - hodnoty (prvky) v zozname majú poradie
- prvky zoznamu sú v samostatných objektoch - **uzloch** (ang. node)



- každý **uzol** uchováva
 - **hodnotu**
 - **referenciu** (navigáciu) na uzol, ktorý uchováva nasledujúcu hodnotu v zozname



Uzol spájaného zoznamu

```
public class Uzol {  
    int hodnota;  
    Uzol dalsi;  
}
```

Hodnota uložená v uzle
spájaného zoznamu

Referencia na uzol
obsahujúci nasledujúcu
hodnotu v spájanom
zozname.

Spájaný zoznam je „rekurzívna“ údajová
štruktúra - opis triedy zahŕňa referencovanie
objektov opisovanej triedy



Spájaný zoznam v Java

```
public class SpajanyZoznam {
    private static class Uzol {
        int hodnota;
        Uzol dalsi;
    }
}
```

Vnútoraná (a privátna)
trieda pre uzly
zoznamu.

```
private Uzol prvý = null;
}
```

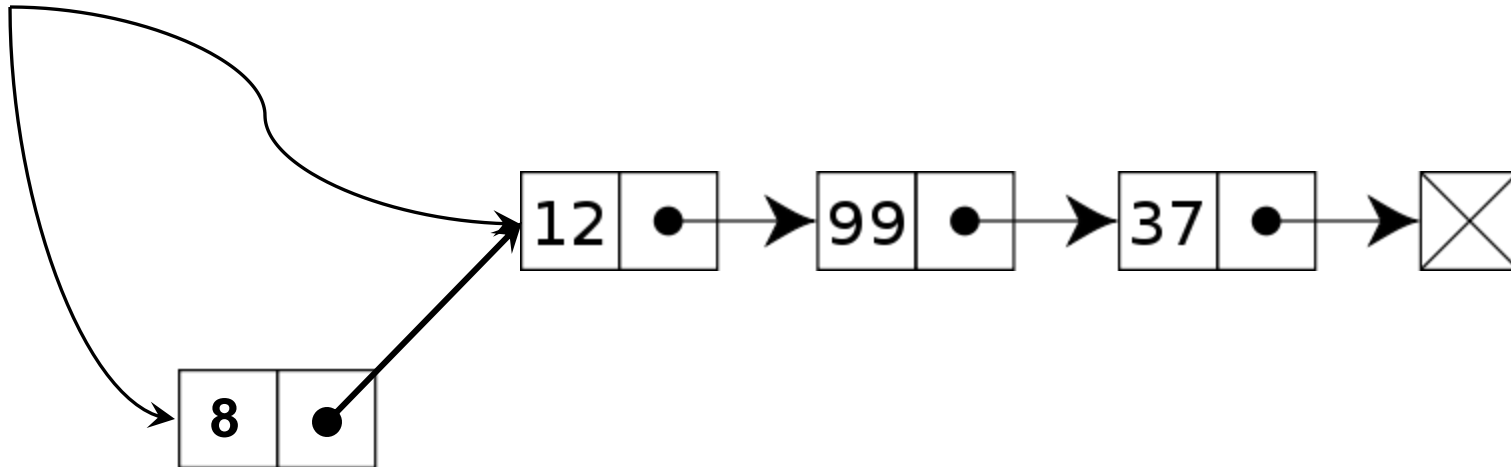
Referencia na **prvý**
uzol v zozname.

Na začiatku v zozname
nieť uzlov.



Pridanie na začiatok zoznamu

prvy



```

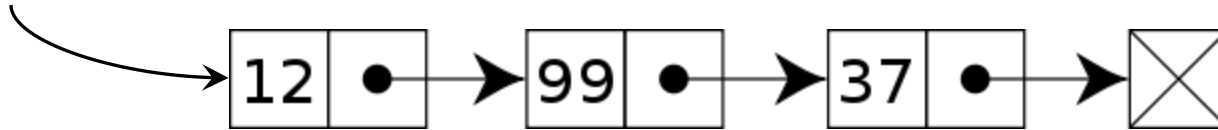
public void pridajNaZaciatok(int hodnota) {
    ➡ Uzol pridavany = new Uzol();
    ➡ pridavany.hodnota = hodnota;
    ➡ pridavany.dalsi = prvy;
    ➡ prvy = pridavany;
}

```



Prechod spájaným zoznamom

prvy



Referencia na uzol, na ktorom sa **práve nachádzame**.
Štartujeme od prvého.

Kontrola, či premenná `aktualny` referencuje nejaký uzol...

Presun na ďalší uzol v zozname.

```
public int sucet() {
    Uzol aktualny = prvý;
    int vysledok = 0;
    while (aktualny != null) {
        vysledok += aktualny.hodnota;
        aktualny = aktualny.dalsi;
    }
    return vysledok;
}
```



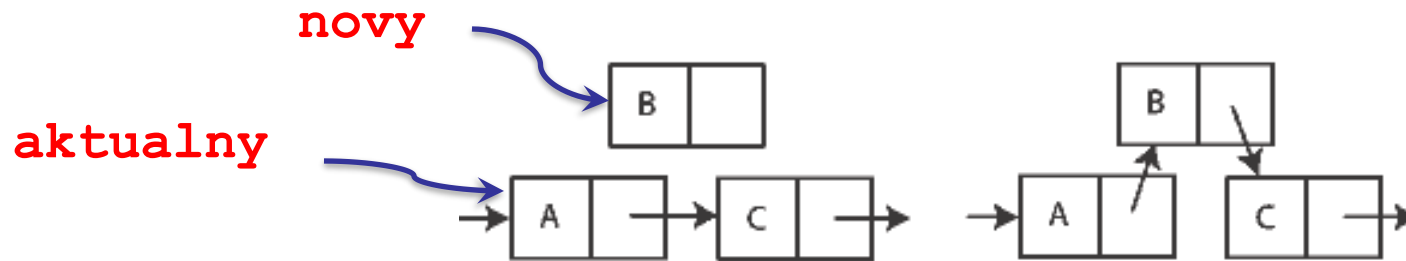
Prechod spájaným zoznamom

- Základná schéma na prechod spájaným zoznamom (iteráciu jeho prvkami):

```
Uzol aktualny = prvky;  
while (aktualny != null) {  
    //... práca s aktuálnym uzlom ...  
    aktualny = aktualny.dalsi;  
}
```



Vloženie uzla do zoznamu



C

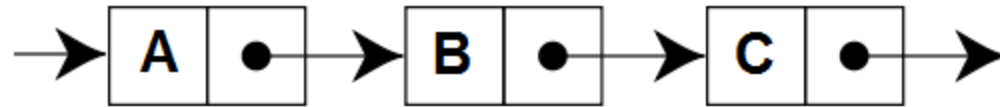
```

novy.dalsi = aktualny.dalsi;
aktualny.dalsi = novy;
  
```

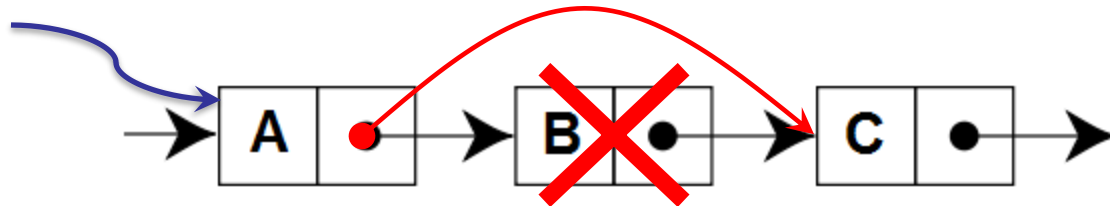
Počet operácií (čas): $O(1)$



Odstránenie uzla zo zoznamu



aktualny



aktualny.dalsi = $\underbrace{\text{aktualny.dalsi.dalsi}}_B$; C

Počet operácií (čas): $O(1)$



Výber *i*-teho prvku

```

public int get(int index) {
    Uzol aktualny = prvý;
    int pozicia = 0;
    while (aktualny != null) {
        if (pozicia == index)
            return aktualny.hodnota;

        aktualny = aktualny.dalsi;
        pozicia++;
    }
    throw new IndexOutOfBoundsException();
}

```

Prechádzame zoznamom a pamätáme si, na koľkom v poradí uzle sme.

Ak sme našli správny uzol, vrátime hodnotu a končíme.

Ak sme došli sem, tak sme nenašli správny uzol.

Počet operácií v najhoršom prípade: $O(n)$



Vylepšenia

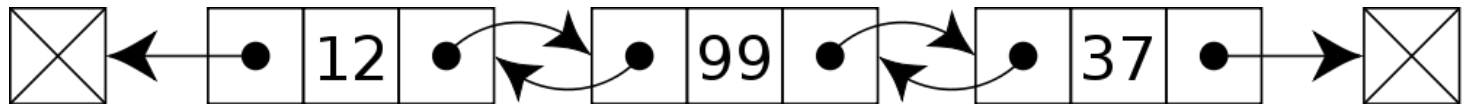
- Okrem referencie na prvý uzol si pamätáme aj **referenciu na posledný uzol** v zozname
 - pridanie na koniec zoznamu v čase $O(1)$, keďže nemusíme hľadať koniec
 - metódy modifikujúce zoznam **musia aktualizovať** aj referenciu na aktuálne posledný uzol v zozname
- Pamätáme si **aktuálny počet uzlov** v zozname
 - dokážeme rýchlo overiť platnosť indexu a povedať, koľko hodnôt máme v zozname



Variácie spájaných zoznamov

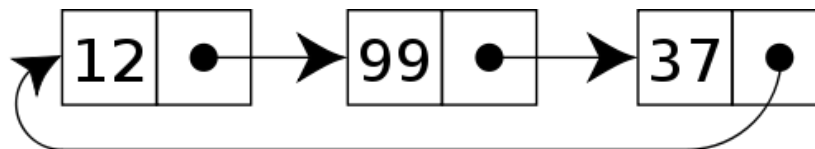
● Obojsmerný spájaný zoznam:

```
public class Uzol {
    int hodnota;
    Uzol dalsi;
    Uzol predchadzajuci;
}
```



● Cyklický spájaný zoznam:

- posledný uzol má ako nasledovníka prvý uzol





Sumarizácia spáj. zoznamov

- Počet operácií v najhoršom prípade (= keď máme smolu):
 - get - $O(n)$, set - $O(n)$
 - pridanie na začiatok a na koniec: $O(1)$
 - pridanie/odobranie (bez nájdania pozície): $O(1)$
- Trieda `LinkedList<E>` na uloženie zoznamu interne využíva *obojsmerný spájaný zoznam*.
- Kedy ich teda **použiť**?
 - „pracujeme“ na začiatku alebo konci zoznamu
 - prechádzame zoznam iterátorom (for-each cyklus)



Na čo ešte ide použiť spáj. zoznam?

- Dve údajové štruktúry, kde sa pracuje **len na „koncoch“**:

**Zásobník
(Stack)**

**Rad
(Queue)**

Zásobník a rad ide implementovať aj inak ako s použitím spájaných zoznamov (viac na cvičeniach)



Zásobníky v reálnom svete

- Základné operácie:
 - vieme **pridať** tanier na vrch zásobníka
 - vieme **odobrať** tanier z vrchu zásobníka
 - vieme zistiť, či je zásobník **prázdny**
- Štruktúra je typu **LIFO**: *last in first out*





Zásobníky v Java

- Zásobníky sú v Java implementované triedou **Stack<E>** so základnými operáciami:
 - E **push**(E item) – pridá objekt na vrch zásobníka
 - E **pop**() – vráti a odstráni objekt z vrchu zásobníka
 - E **peek**() – vráti objekt z vrchu zásobníka, no neodstráni ho zo zásobníka
 - **boolean empty**() – vráti, či je zásobník prázdny
- Príklady: **Stack<Integer>**, **Stack<String>**, **Stack<Character>**, ... pre primitívne typy musíme použiť wrapovacie triedy



Na čo je zásobník dobrý?

- **Zásobník (Stack)** je často používaná údajová štruktúra
 - **call-stack** je zásobník volaní
- rozpoznávanie reťazcov určitého tvaru:
 - $a^n b^n$ - reťazce z písmen **a** a **b**, kde najprv ide n **a**-čok a potom n **b**-čok
 - rozpoznanie správne ozátvorkovaného výrazu
 - rozpoznanie správne ozátvorkovaného výrazu s viacerými sadami zátvoriek (**(, <, {, }, >,)**), ...
 - spracovanie HTML a XML dokumentov



Správne ozátvorkovaný výraz

- Zadanie: $(\{()\})((\ \))$ ✓ $((\ \))(\ \)$ ✗
- Princíp naivného riešenia:

$(\{()\})((\ \))$



$(\{\ \})((\ \))$

```

StringBuilder sb = new StringBuilder(vyraz);
int idxZatvoriek;
do {
    // nájdi dobrý pár susediacich zátvoriek
    idxZatvoriek = sb.indexOf("(");
    if (idxZatvoriek < 0) {
        idxZatvoriek = sb.indexOf("{}");
    }

    // odstráň dobrý pár z výrazu (ak existuje)
    if (idxZatvoriek >= 0) {
        sb.delete(idxZatvoriek, idxZatvoriek + 2);
    }
} while (idxZatvoriek >= 0);
    
```

$O(n)$

$O(n) \in$

$O(n)$

$O(n^2)$



Správne ozátvorkovaný výraz

● Princíp riešenia:

$(\{\{()\})((())$ ✓ $((())\})()$ ✗

- postupne čítame vstup po znakoch
- ak čítame **otváraciu** zátvorku, tak ju len **vložíme** do zásobníka
- ak čítame **uzatváraciu** zátvorku, tak **overíme**, či na vrchu zásobníka je príslušná otváracia zátvorka
 - **áno**: vyberieme zátvorku zo zásobníka
 - **nie**: vstup nie je správne ozátvorkovaný výraz
- po prečítaní vstupu musí byť zásobník **prázdny**

*Simulácia a experimenty
na cvičeniach*



Testovanie pre jednu sadu

```
public static boolean spravneOzatkovkovany(String vyraz) {  
    Stack<Character> zasobnik = new Stack<Character>();  
  
    for (int i=0; i<vyraz.length(); i++) {  
        char znak = vyraz.charAt(i);  
        if (znak == '(')  
            zasobnik.push(znak);  
  
        if (znak == ')') {  
            if (zasobnik.isEmpty())  
                return false;  
  
            if (zasobnik.pop() != '(')  
                return false;  
        }  
    }  
  
    return zasobnik.isEmpty();  
}
```

Ak na vrchu zásobníka nie je
otváracia zátvorka, tak
končíme.





Rady v reálnom svete

- Ciel' radu:
 - spravodlivo čakať na niečo
- Základné operácia štruktúry rad:
 - postaviť sa **na koniec** radu
 - ako **prvý** v rade byť „**vybraný**“
- Štruktúra **FIFO** - first in first out





Rady v Java

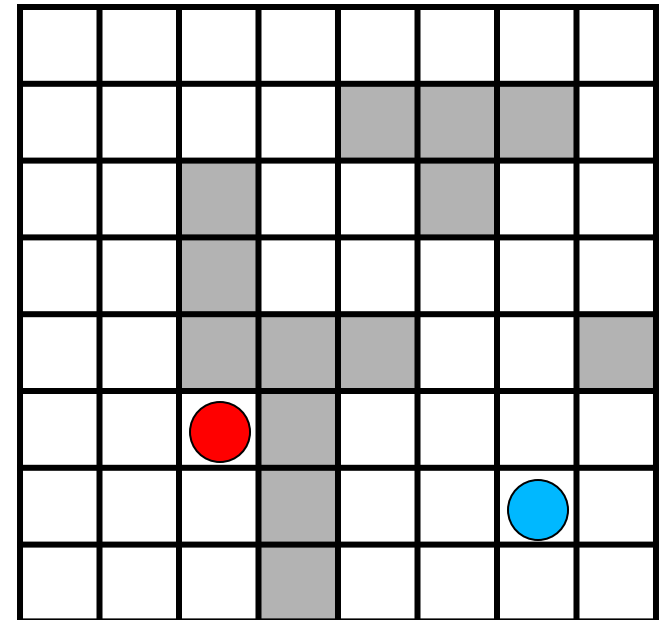
- Rady v Java sú implementované triedami, ktoré implementujú rozhranie **Queue<E>**
- Základné metódy rozhrania **Queue<E>**:
 - **boolean offer**(E item) - vloží objekt na koniec radu a vráti, či sa objekt podarilo vložiť
 - **E poll**() - vráti prvý objekt v rade, *null* ak v rade nie je žiaden objekt
 - **boolean isEmpty**() - vráti, či je rad prázdny
- Implementácia **Queue<E>**: **LinkedList<E>**



Ako nájsť cestu v bludisku

● Bludisko:

- mriežkový labyrint s políčkami
- niektoré políčka sú voľné, niektoré sú obsadené
- **š**tartové a **ci**eľové políčko
- v jednom kroku sa vieme posunúť v jednom zo 4 smerov (vľavo, vpravo, hore, dole)
- ako sa **čo najrýchlejšie** dostať zo štartu do cieľa ?



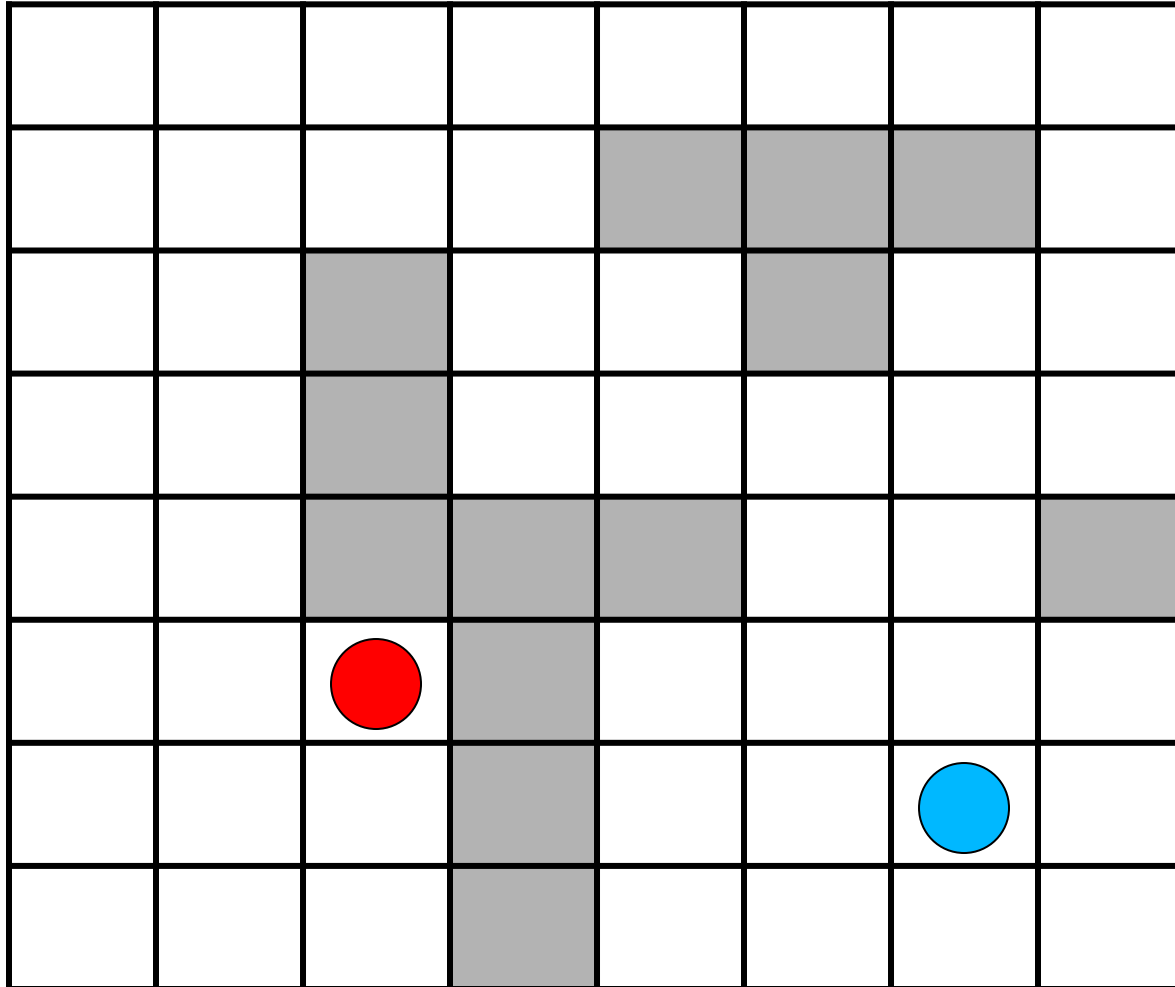


Úvahy o hľadání cesty

- Namiesto zdanlivo jednoduchšej úlohy riešme zložitejšiu a **všeobecnejšiu úlohu**:
 - Pre každé políčko vypočítajme, ako je vzdialené od štartovacieho políčka ...
- **Zjavné fakty**:
 - štartovacie políčko je vzdialené 0 od štartovacieho políčka
 - ak je políčko vo vzdialenosti d od štartu, potom niektorý jeho sused bude vo vzdialenosti $d-1$ od štartu



Počítanie vzdialenosti

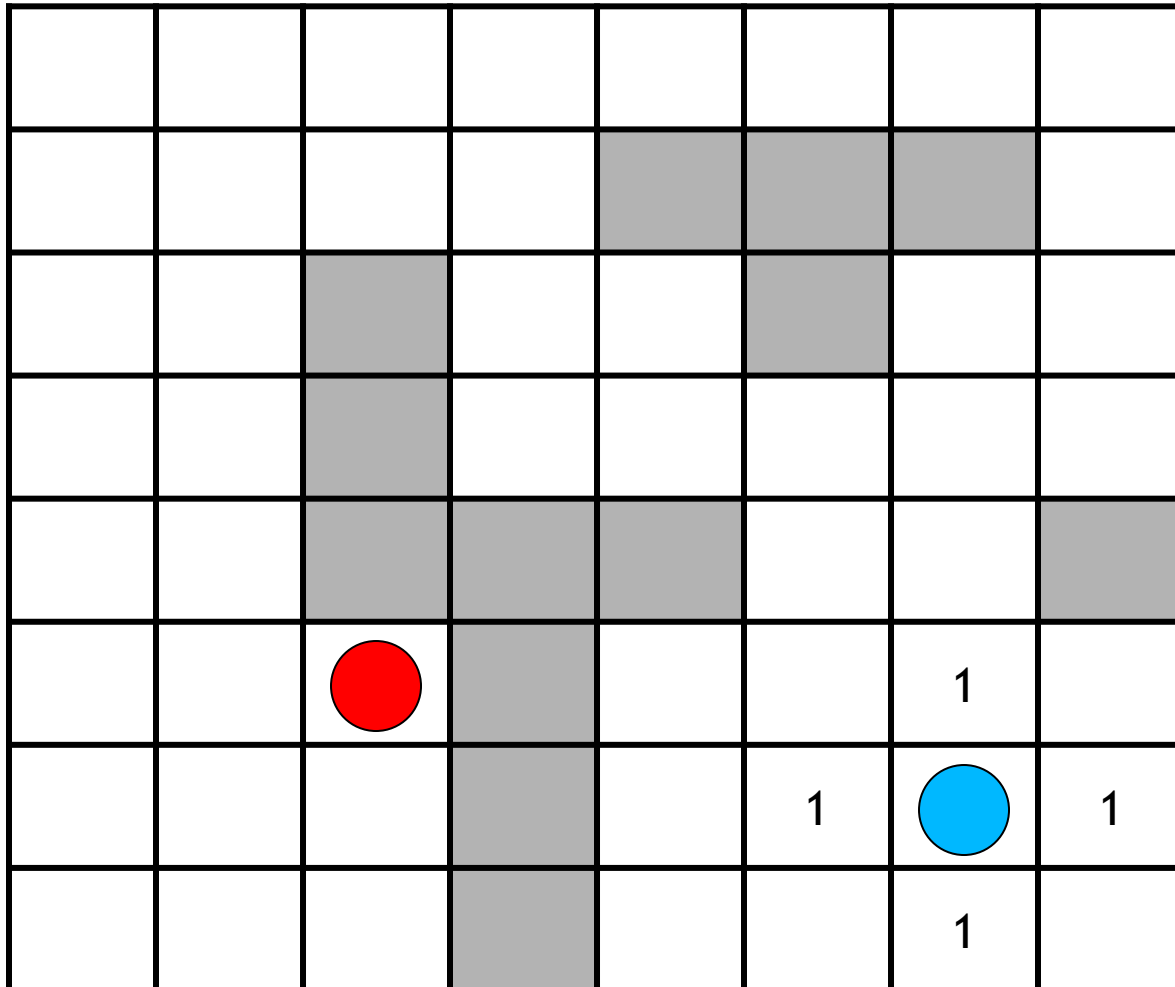


Štart: 

Cieľ: 



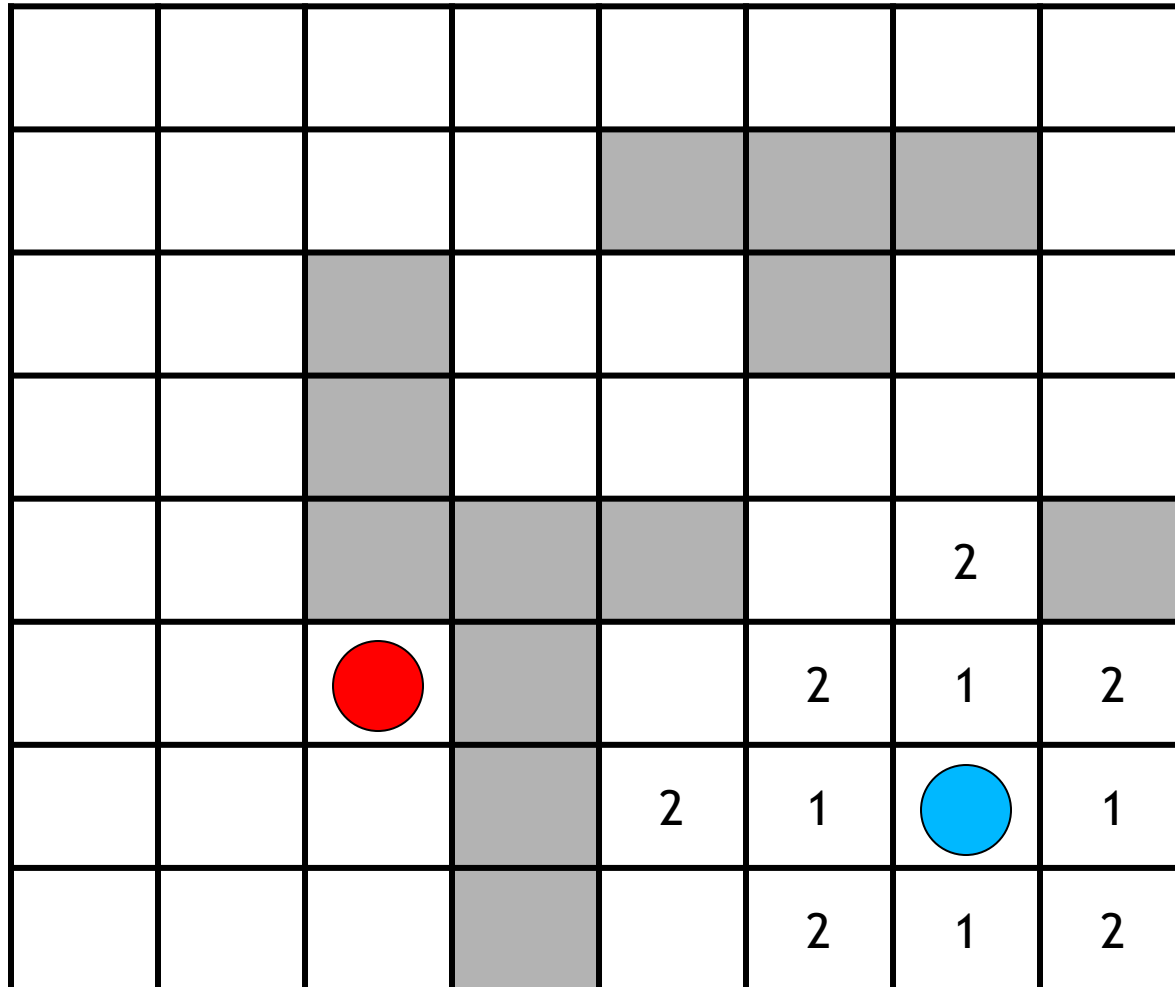
Počítanie vzdialenosti



Všetkým susedom
štartovacieho
políčka nastavíme
vzdialenosť 1



Počítanie vzdialenosti



Všetkým susedom políčok so vzdialenosťou 1 od štartu nastavíme vzdialenosť 2



Počítanie vzdialenosti

						3	
					3	2	
		●		3	2	1	2
				2	1	●	1
				3	2	1	2

Všetkým susedom
políčok so
vzdialenosťou 2
od štartu
nastavíme
vzdialenosť 3



Počítanie vzdialenosti

						4	
					4	3	4
					3	2	
		●		3	2	1	2
				2	1	●	1
				3	2	1	2

Všetkým susedom políčok so vzdialenosťou 3 od štartu nastavíme vzdialenosť 4



Počítanie vzdialenosti

						4	5
				5	4	3	4
					3	2	
				3	2	1	2
				2	1		1
				3	2	1	2

Všetkým susedom
políčok so
vzdialenosťou 4
od štartu
nastavíme
vzdialenosť 5





Počítanie vzdialenosti

							6
				6		4	5
			6	5	4	3	4
					3	2	
		●		3	2	1	2
				2	1	●	1
				3	2	1	2

Všetkým susedom políčok so vzdialenosťou 5 od štartu nastavíme vzdialenosť 6



Počítanie vzdialenosti

12	11	10	9	10	9	8	7
11	10	9	8				6
12	11		7	6		4	5
13	12		6	5	4	3	4
14	13				3	2	
15	14			3	2	1	2
16	15	16		2	1		1
17	16	17		3	2	1	2



Ako na efektívny program?

- Pre každé políčko si pamätáme jeho vypočítanú vzdialenosť od štartovacieho políčka
 - na začiatku -1 (nevypočítaná vzdialenosť)
- Pracujeme v krokoch:
 - v i -tom kroku nastavíme všetkým „nespracovaným“ susedom políčok s hodnotou i ich vzdialenosť na $i+1$.
 - naivná implementácia vyžaduje navštíviť všetkých n políčka poľa v každom kroku ... $O(n)$
 - najviac n krokov: $n \cdot O(n) = O(n^2)$





Ako na efektívny program?

- Pre každé políčko si pamätáme jeho vypočítanú vzdialenosť od štartovacieho políčka
 - na začiatku -1 (nevypočítaná vzdialenosť)
- Udržiavame si **rad** so súradnicami políčok, pričom pre každé políčko čakajúce v rade platí:
 - políčko už má vypočítanú svoju vzdialenosť od štartovacieho políčka
 - susedia tohto políčka môžu byť bez vypočítanej vzdialenosti od štartovacieho políčka



Schéma algoritmu

Algoritmus prehľadávania do šírky

- kým *rad nie je prázdny* opakuj
 - vyber prvé políčko P v rade
 - pozri sa na **každé** susedné políčko S políčka P
 - ak **S nemá nastavenú vzdialenosť'** (jeho vzdialenosť' je -1), potom:
 - nastav pre S vzdialenosť' o 1 väčšiu ako má P:
$$\text{vzdialenosť}'(S) = \text{vzdialenosť}'(P) + 1$$
 - zarad' S na koniec radu (lebo sa treba ešte pozrieť' aj na susedov políčka S)



Prečo to funguje?



● Tvrdenia o behu algoritmu

- každé políčko v rade má vypočítanú svoju vzdialenosť od štartu
- každé políčko sa dostane do radu **len raz ... $O(n)$**
 - pred vložením do radu sa mu nastaví vzdialenosť na číslo rôzne od -1, čo bráni opätovnému zaradeniu do radu
- **všetky** políčka s vzdialenosťou d od štartu sú vybrané z radu **pred všetkými** políčkami so vzdialenosťou $d+1$ (vd'aka radu a spôsobu spracovania)
 - formálne sa dokáže indukciou na d (sporom sa ukáže, že sa nemôžu predbehnúť a že na žiadne políčko vo vzdialenosti $d+1$ sa nemôže zabudnúť za splnenia indukčného predpokladu)





Nájdienie cesty

12	11	10	9	10	9	8	7
11	10	9	8				6
12	11		7	6		4	5
13	12		6	5	4	3	4
14	13				3	2	
15	14			3	2	1	2
16	15	16		2	1		1
17	16	17		3	2	1	2

Algoritmus:

- Začíname v celi
- Vždy sa vyberieme na to susedné políčko, ktoré má o 1 menšiu vzdialenosť od štartovacieho políčka ako to, na ktorom práve stojíme.

Spätné hľadanie