

# LINEÁRNÍ DATOVÉ STRUKTURY A ŘAZENÍ



**Kurz:**      **Datové struktury a algoritmy**

---

**Lektor:**    Doc. Ing. Radim Burget, Ph.D.

**Autor:**     Doc. Ing. Radim Burget, Ph.D.



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Vytvoření této videopřednášky bylo podpořeno projektem č. CZ.1.07/2.2.00/28.0098  
Evropského sociálního fondu a státním rozpočtem České republiky.

# Cíl přednášky

- I. Správa paměti
- II. Reference a jejich význam
- III. Pole
- IV. Lineární seznam
  - Jednosměrně vázaný lineární seznam
  - Cyklický lineární seznam
  - Obousměrně vázaný lineární seznam
  - Průchod seznamem, přidání a odstranění prvku ze seznamu
- V. Řazení prvků v lineárních strukturách

# Organizace paměti

- Správa paměti operačním systémem (viz BSOS)
  - Halda, zásobník

Správa paměti:

- Manuální
- Garbage collector

Paměť vyhrazená pro běh programu:



```
public class AhojSvete {  
    public static void main(String[] args) {  
        System.out.println("Ahoj svete");  
        int abc = 100;  
        Obdelnik o = new Obdelnik(10, 20, 30, 40);  
    }  
}
```

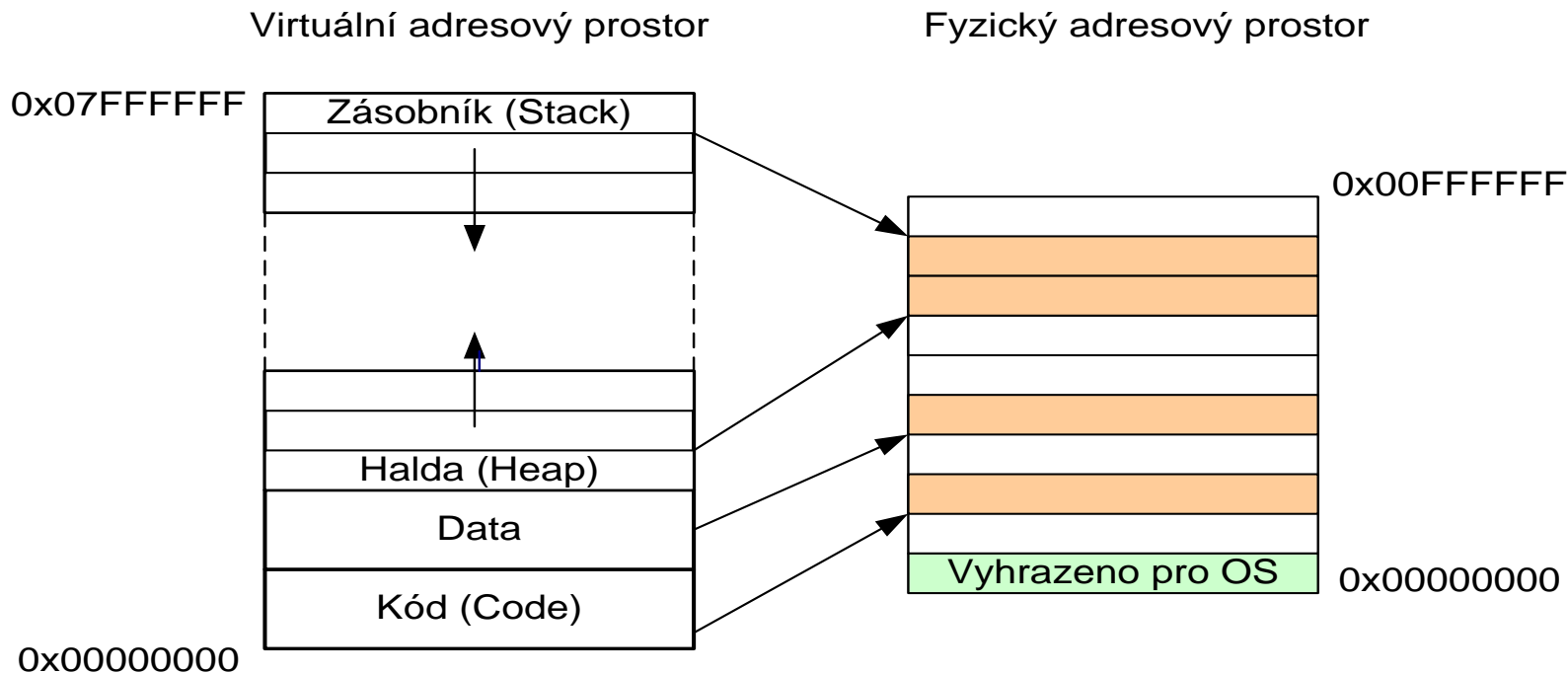
Paměť vyhrazená pro běh programu:



Informace o  
zanořování do  
podprogramů  
(zde žádný)

# Organizace paměti - pokračování

- Operátor „*new*“ – alokace dynamické paměti v haldě (heap)
- Operátor „*delete*“ – dealokace dynamické paměti v haldě (heap)



# Organizace paměti – pokračování (2)

- Jak počítač po dokončení funkce pozná, kde má pokračovat?
  - Zásobník
  - Dynamická paměť
  - V případě, že dojde paměť – `StackOverflowError`

# Organizace paměti – pokračování (3)

```

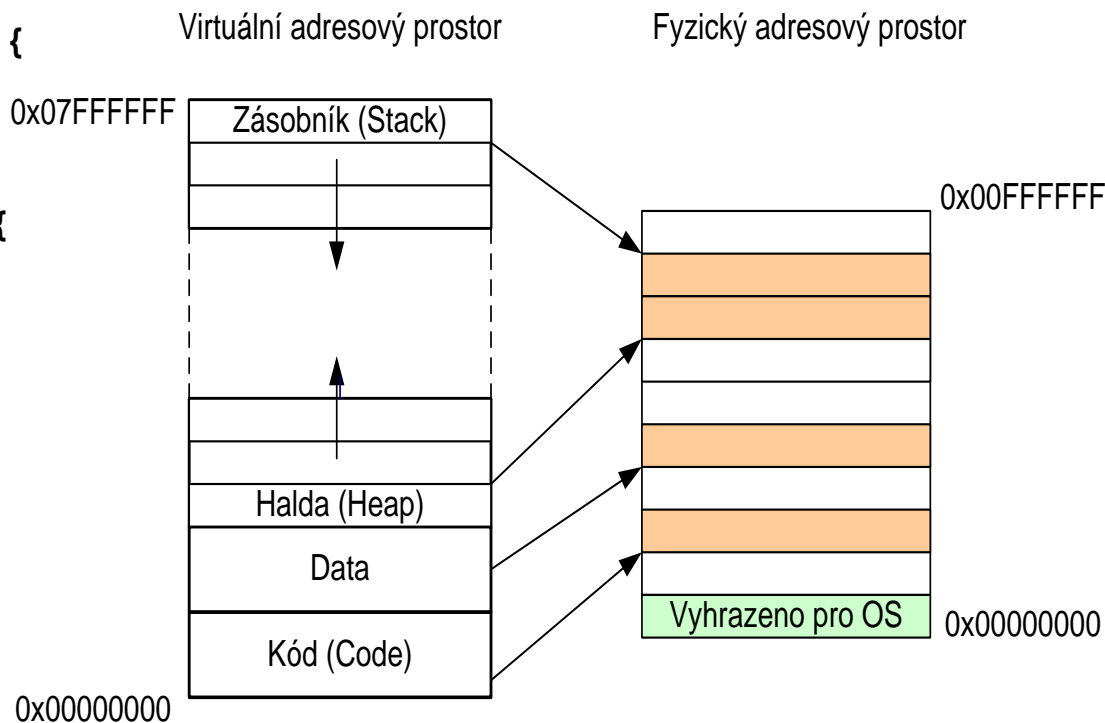
•public class Main {
    public static void main(String[] args) {
        printHelloWorld();
    }

    private static void printHelloWorld() {
        printHello();
        printWorld();
    }

    private static void printWorld() {
        System.out.println(" world.");
    }

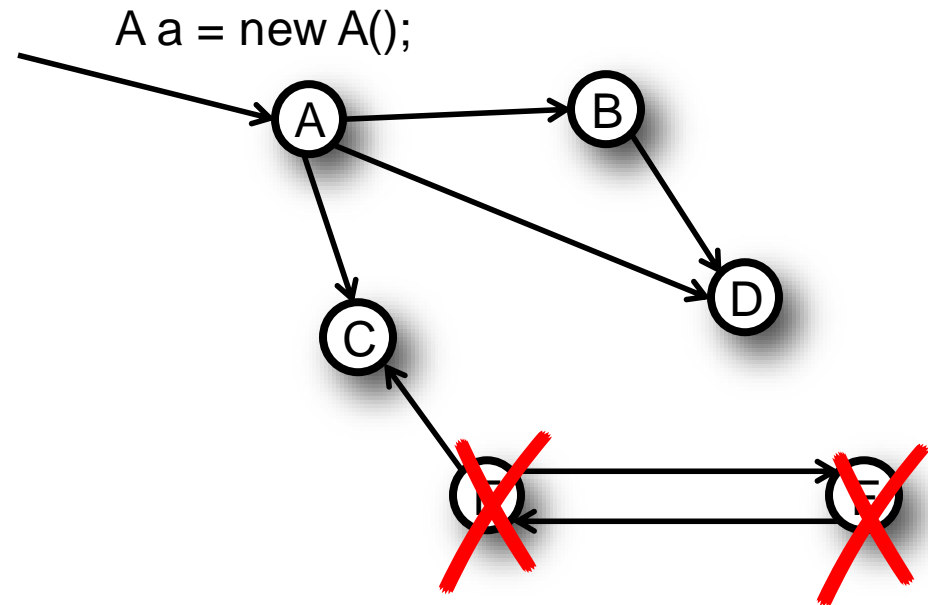
    private static void printHello() {
        System.out.print("Hello");
    }
}

```



# Garbage collector („Sběratel odpadků“)

- Určování nepotřebné paměti a její uvolňování
- Algoritmy
  - Počítání referencí
  - Sledování





# Výhody Garbage collectoru

- Garbage collector eliminuje nutnost hlídat uvolňování paměti programátorem (typicky C, C++)
  - Méně práce pro programátora
- zajišťuje integritu programu
- může razantně zjednodušit programy
- A výkon?
  - Ale nikterak významný – zejména delší inicializace
  - U delší dobu běžících aplikací (např. web aplikace) je často efektivnější dávkové uvolnění paměti, než alokace/dealokace při každé potřebě (optimalizace)

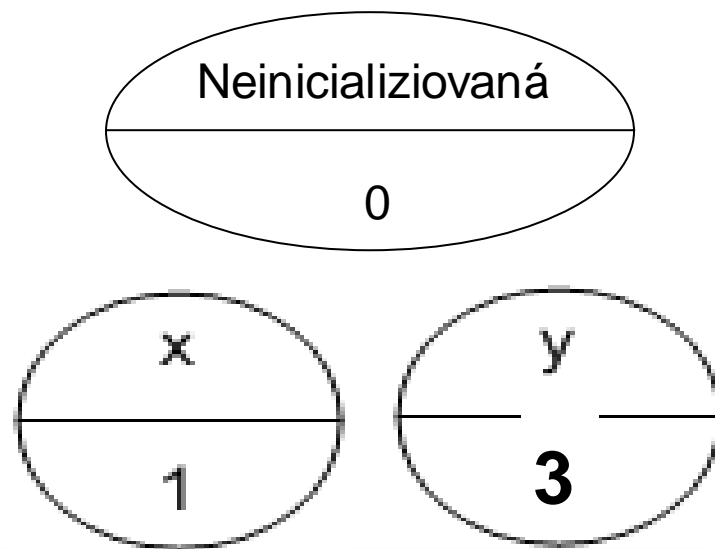
# Nevýhody Garbage collectoru

- Garbage collector přidává režii, která v některých případech může ovlivnit výkonost programu.
- Při spuštění je nutná inicializace GC
- Nevhodné pro programy, které běží krátce (např. ls)
- Garbage collector potřebuje paměť pro svůj běh.
- Programátor má menší kontrolu nad plánováním procesorového času (real-time aplikace !)

# Předávání hodnotou – primitivní datový typ

- Demo:
- **int** neinicializována; // neinicializována=0
- **int** x = 1;
- **int** y = x;                   // x=1, y=1
- y = y+2;                    // x=1, y=3

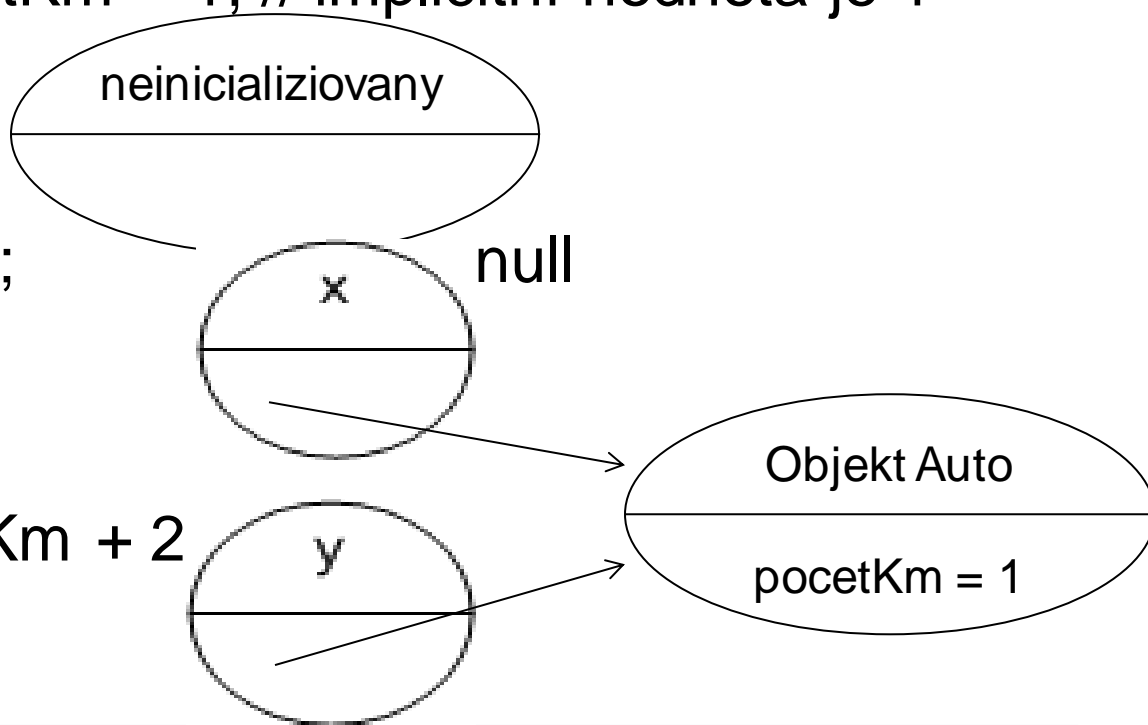
- Dle očekávání...



# Předávání referencí - objekty

Auto
+ pocetKm

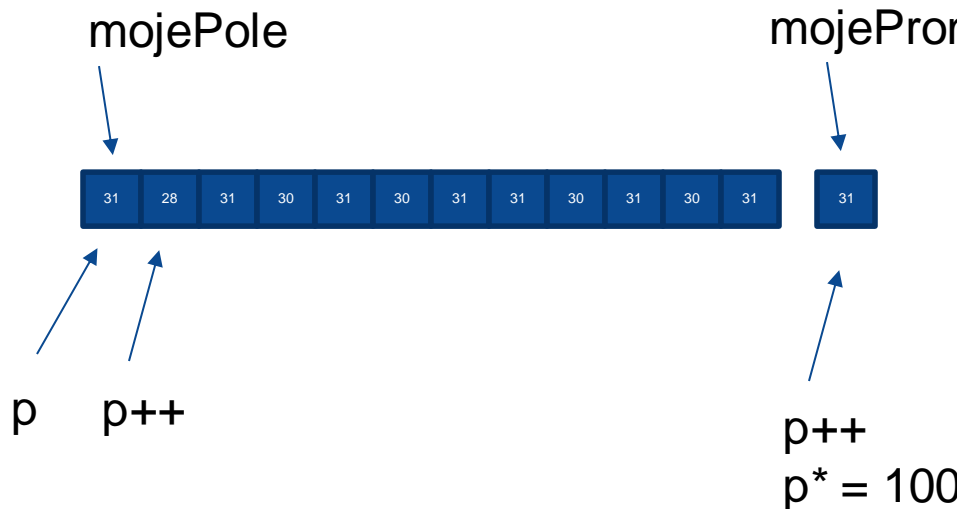
- **public class** Auto {
- **public int** pocetKm = 1; // implicitní hodnota je 1
- }
- Demo:
- Auto neinicializovany;
- Auto x = **new** Auto();
- Auto y = x;
- y.pocetKm = y.pocetKm + 2



# Reference vs. ukazatel

## • Ukazatel

(C, C++)



Nejčastějším  
důvodem chyb v  
programech

## • Reference

(JAVA, C#)



Reference  
zakazuje posun  
způsobem ++

# Reference a jejich význam

- Opakování základů programování
- primitivní datové typy
  - Int, byte, short, double, float, ...
- a nepřimitivní datové typy, tzv. referenční
  - Pole
  - Objekty

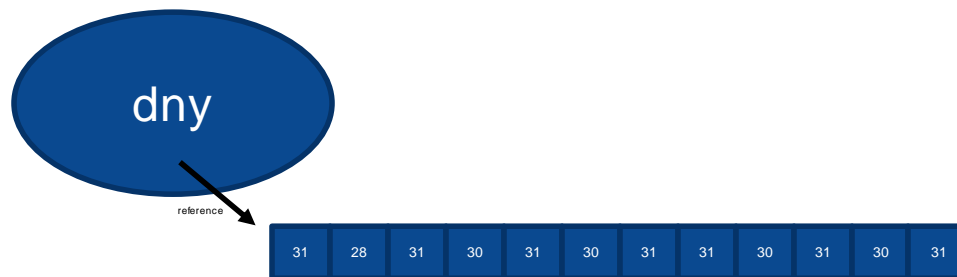
# Primitivní typy – JAVA (předávány hodnotou)

- **boolean** (True or False)
- **char** (16 bit, Unicode 0 to Unicode  $2^{16} - 1$ )
- **byte** (8 bit, -128 to 128)
- **short** (16 bit,  $-2^{15}$  to  $2^{15} - 1$ )
- **int** (32 bit,  $-2^{31}$  to  $2^{31} - 1$ )
- **long** (64 bit,  $-2^{63}$  to  $2^{63} - 1$ )
- **float** (IEEE-754)
- **double** (IEEE-754)
- **void**
- Vše ostatní předáváno referencí

# Pole triviálních datových typů

- Pole triviálních datových typů
  - `int[] pole = new int[100];` ;
  - `int[] dny = {31,28,31,30,31,30,31,31,30,31,30,31};`

- Dvourozměrné pole
  - `int[][] ctverec = {{1,2},{3,4}};`

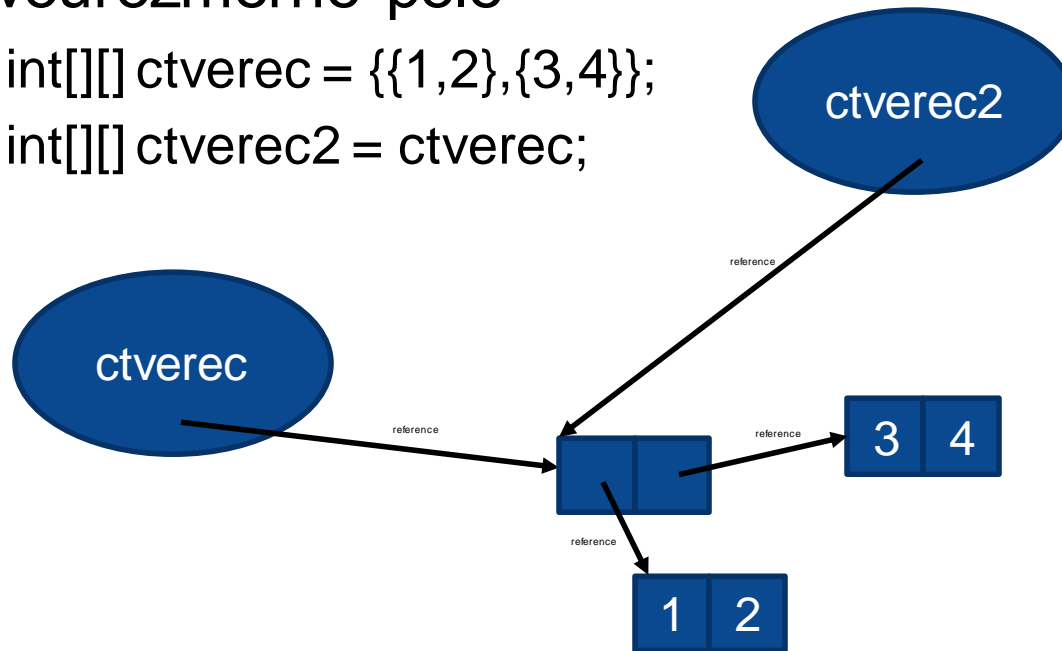


- Hodnoty pole jsou inicializovány na hodnotu 0 (C#, JAVA, PHP, SmallTalk, ...), popřípadě náhodným číslem (C, C++)



- Dvourozměrné pole

- `int[][] ctverec = {{1,2},{3,4}};`
- `int[][] ctverec2 = ctverec;`



# Pole referenčních datových typů

- Pole netriviálních datových typů
  - `String[] pole = new String[100];` :
  - `String[] dny = {„Pondělí“, „Úterý“, „Středa“, „Čtvrtek“, „Pátek“};`
- Dvourozměrné pole
  - `String[][] mesice = {{„1“, „2“, „3“},{„1“, „2“, „3“}};`
- Výchozí reference pole jsou hodnoty *null* (C#, JAVA, PHP, SmallTalk, ...), popřípadě náhodné – pravděpodobně **!!!neplatnou!!!** hodnotou (C, C++)

# java.util.Arrays

**sort(A):** Řadí pole A na základě přirozeného uspořádání jeho prvků, které musí být srovnatelné. Tato metoda používá algoritmus rychlého třídění (Quick sort)

**toString(A):** Vrátí řetězcovou reprezentaci pole A, což je seznam prvků A oddělených čárkami, seřazený tak, jak jsou uvedeny v A, začínající [a končící].

Reprezentace řetězce prvku A [i] se získá pomocí `String.valueOf (A [i])`, který vrací řetězec „null“ pro nulový objekt a jinak volá `A [i] .toString ()`.

# Příklad: Předání hodnotou vs referencí



```
public ArrayList setList(ArrayList list){  
    list = new ArrayList();  
    list.add(1);  
    return list  
}
```

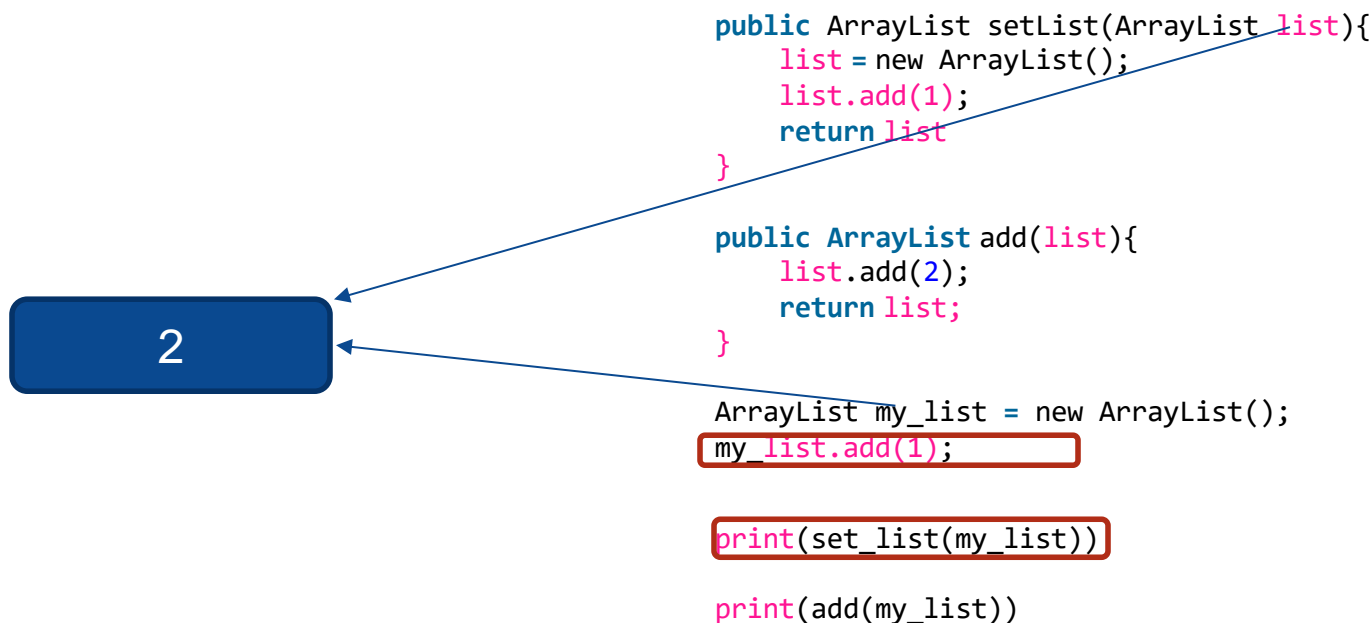
```
public ArrayList add(list){  
    list.add(2);  
    return list;  
}
```

```
ArrayList my_list = new ArrayList();  
my_list.add(1);
```

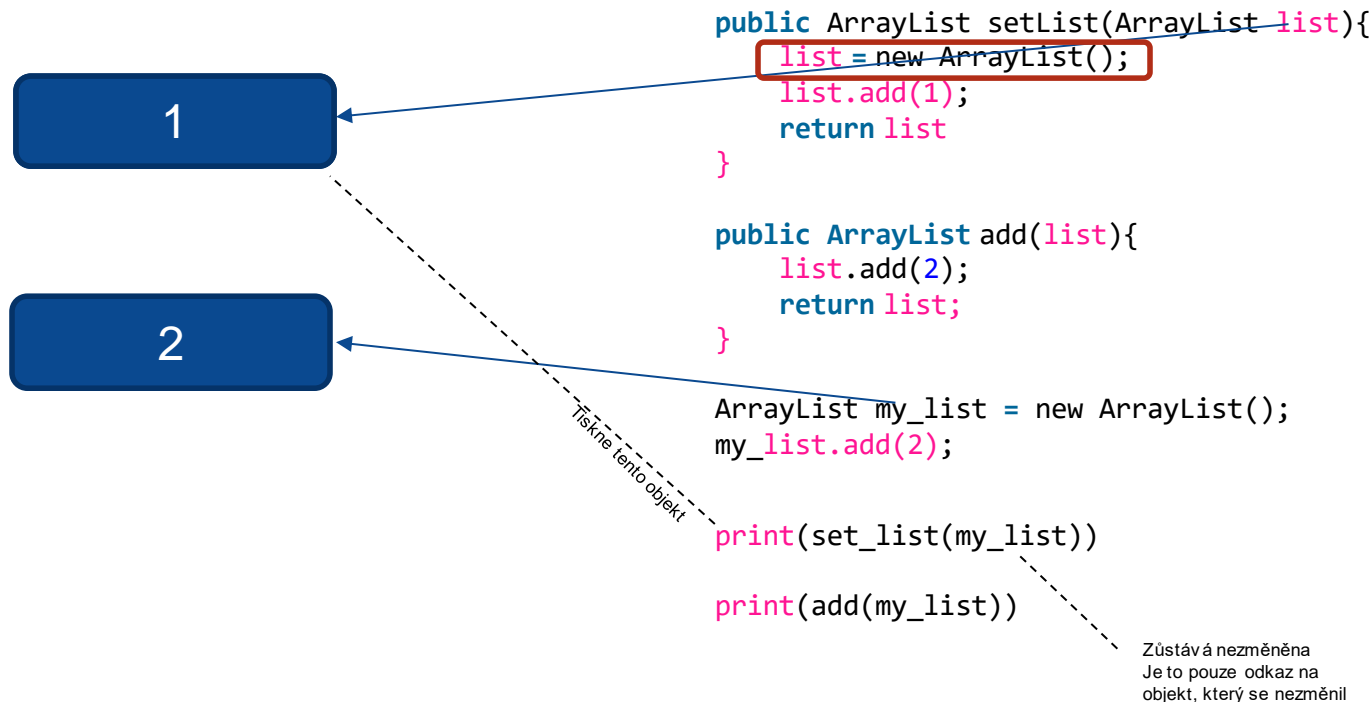
```
print(set_list(my_list))
```

```
print(add(my_list))
```

# Příklad: Předání hodnotou vs referencí



# Příklad: Předání hodnotou vs referencí



# Pole vs. Seznam

- **Pole**

- + Rychlá indexace prvku ( $O(1)$  instrukce strojového jazyka)
- Změna velikosti

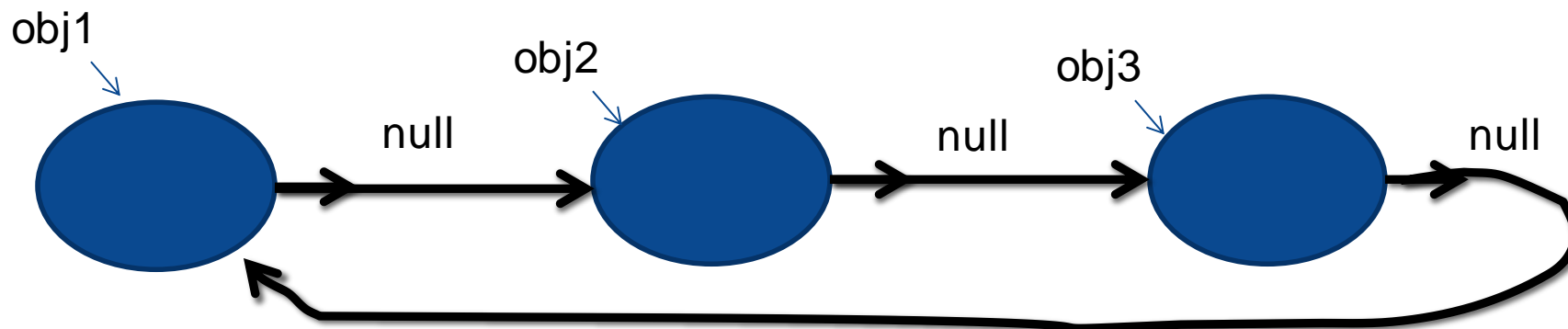
- **Lineární seznam**

- + Efektivní vkládání a mazání
- **Neefektivní přístup k prvkům uvnitř** ( $O(n)$  instrukcí strojového jazyka)

# Příklad – Reference

```
public static void main(String[] args) {  
    MojeTrida obj1 = new MojeTrida();  
    MojeTrida obj2 = new MojeTrida();  
    MojeTrida obj3 = new MojeTrida();  
  
    obj1.setMojeReference(obj2);  
    obj2.setMojeReference(obj3);  
  
    obj1.getMojeReference().getMojeReference().setMojeReference(obj1);  
}
```

```
public class MojeTrida {  
    private MojeTrida mojeReference;  
    // get a set metody  
}
```

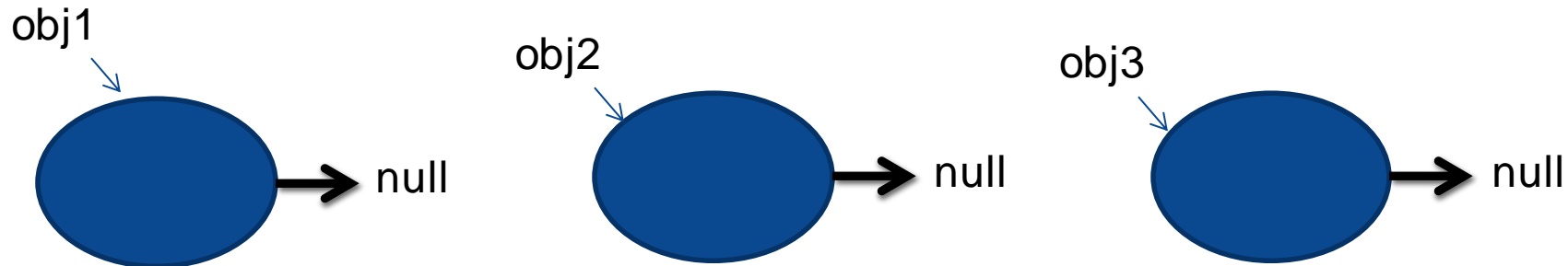




# Příklad – Reference

```
public static void main(String[] args) {  
    MojeTrida obj1 = new MojeTrida();  
    MojeTrida obj2 = new MojeTrida();  
    MojeTrida obj3 = new MojeTrida();  
  
    obj1 = obj2;  
}
```

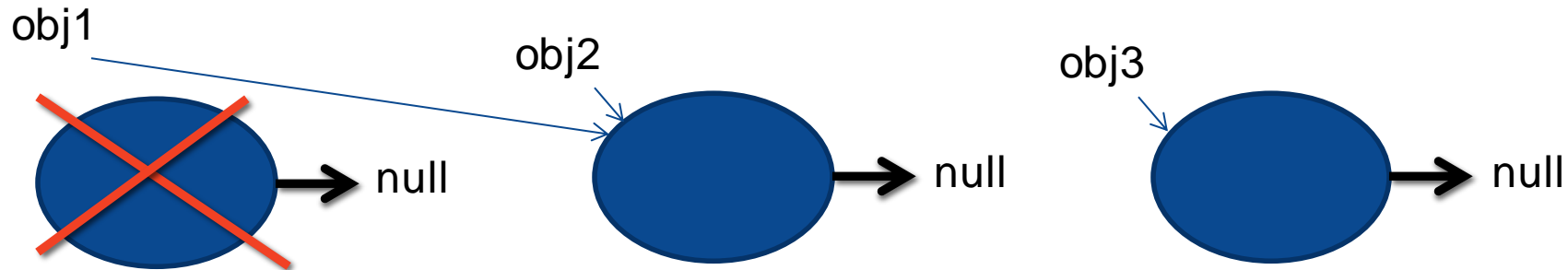
```
public class MojeTrida {  
    private MojeTrida mojeReference;  
    // get a set metody  
}
```



# Příklad – Reference

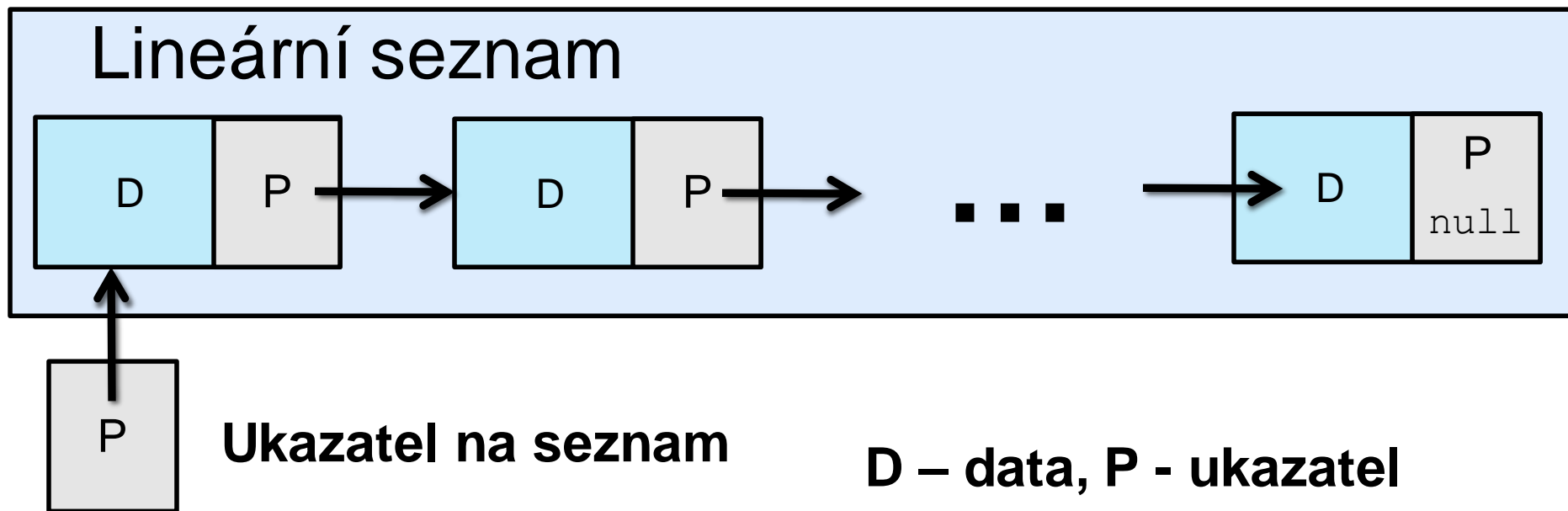
```
public static void main(String[] args) {  
    MojeTrida obj1 = new MojeTrida();  
    MojeTrida obj2 = new MojeTrida();  
    MojeTrida obj3 = new MojeTrida();  
  
    obj1 = obj2;  
}
```

```
public class MojeTrida {  
    private MojeTrida mojeReference;  
    // get a set metody  
}
```



# Lineární seznam

- **Definice:** ADT lineární seznam je seznam, kde každý uzel má unikátního následníka.
- **Vlastnost:** sekvenční



# Možné operace s ADT seznam

- **Nalezení délky**  $N$  seznamu.
- **Výpis** všech prvků seznamu.
- **Vytvoření prázdného** seznamu
- **Získání**  $k$ -tého prvku ze seznamu,  $0 \leq k < N$ .
- **Vložení** nového prvku za  $k$ -tý prvek seznamu,  $0 \leq k < N$ .
- **Smazání** prvku ze seznamu.
- **Nalezení následujícího** prvku za aktuálním v seznamu.
- **Nalezení předchozího** prvku před aktuálním v seznamu.

# Možné operace s ADT seznam

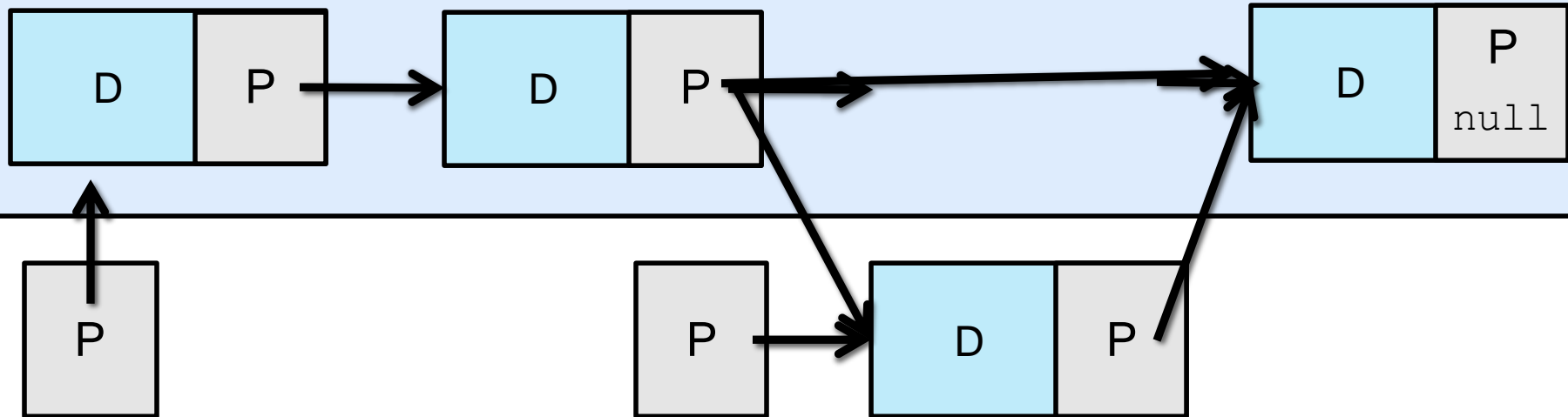
- Řazení
- Slučování

# Inicializace seznamu

- Vytvoření ukazatele na první prvek
  - `null` u prázdného
- Inicializace počtu prvků na nulu

# Vkládání - obecně

## Lineární seznam

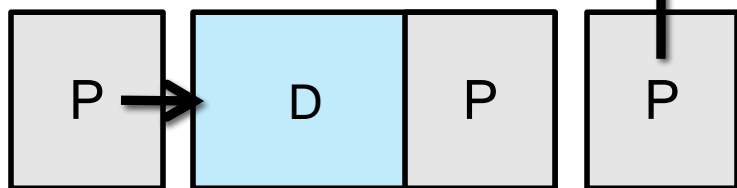
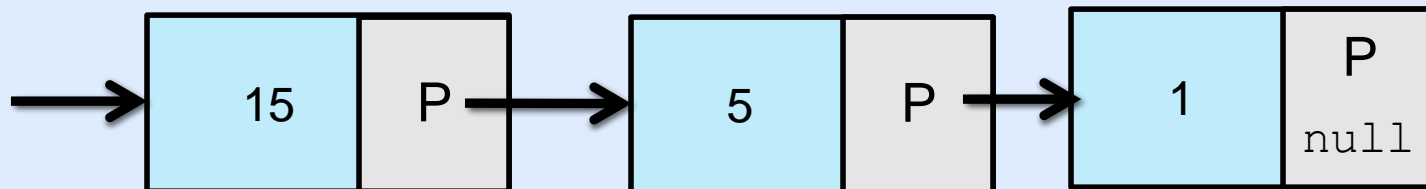


**Ukazatel  
na seznam**

**Ukazatel na  
nový prvek** D – data, P - ukazatel

# Vkládání – na začátek

## Lineární seznam



**Ukazatel na  
nový prvek**

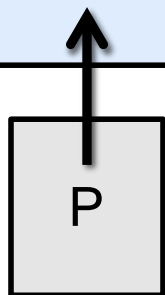
**Ukazatel  
na seznam**

**D – data, P - ukazatel**

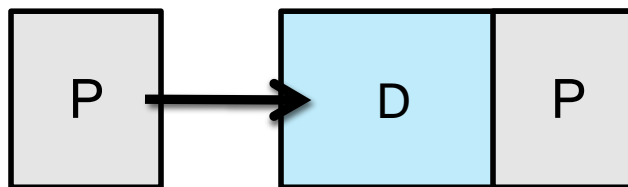


# Vkládání – do prázdného seznamu

Lineární seznam



**Ukazatel  
na seznam**

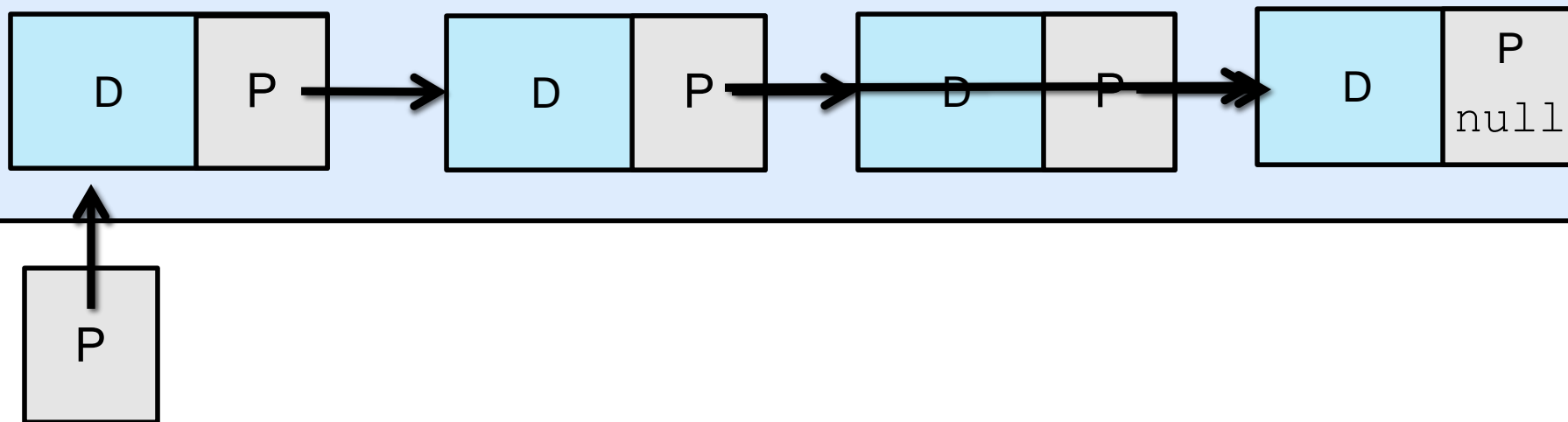


**Ukazatel na  
nový prvek**

D – data, P – ukazatel

# Mazání - obecně

## Lineární seznam

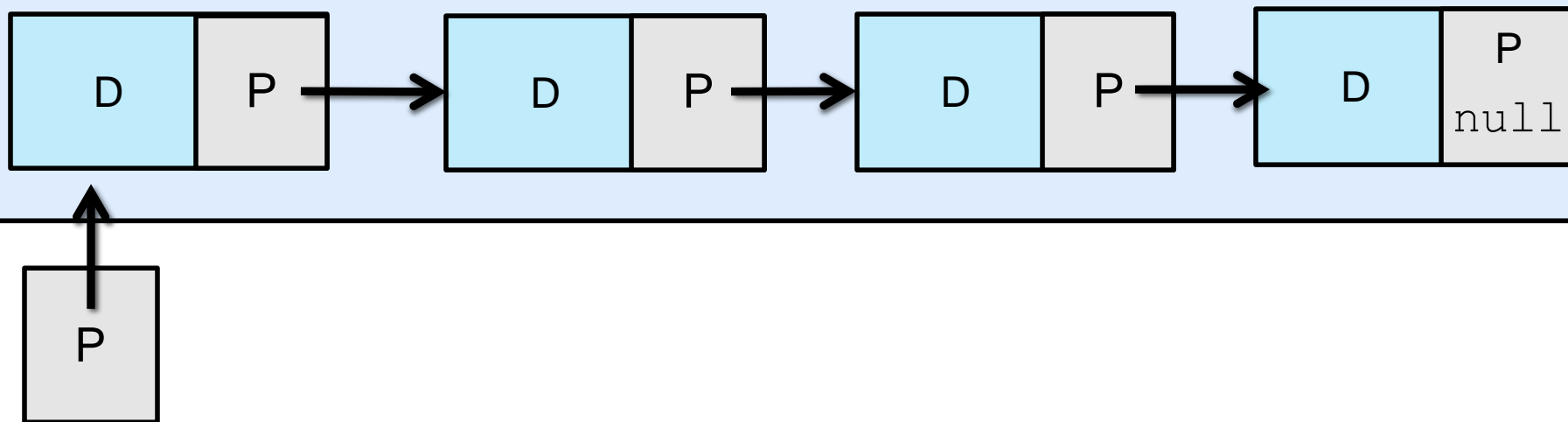


**Ukazatel  
na seznam**

**D – data, P – ukazatel**

# Mazání – prvního prvku

## Lineární seznam

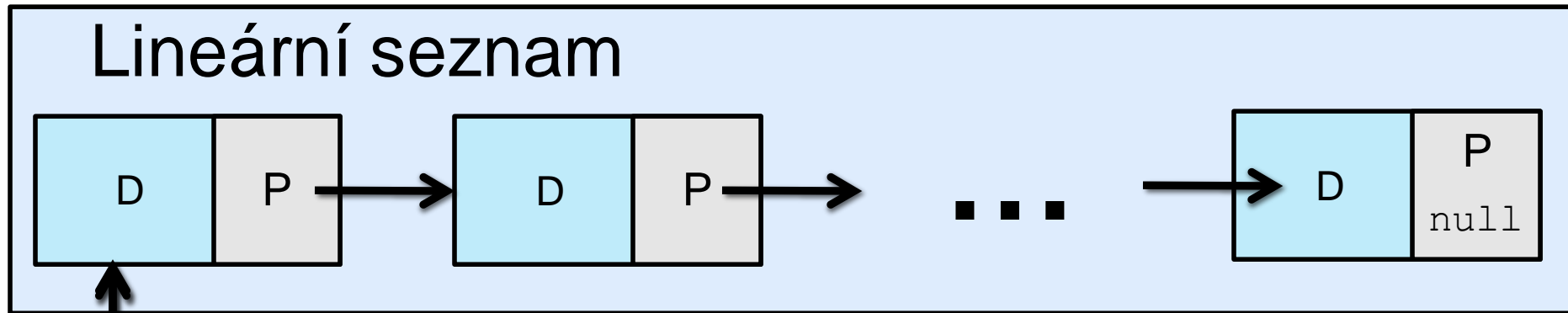


**Ukazatel  
na seznam**

**D – data, P – ukazatel**

# Průchod / vyhledávání

- Podle dat v prvku
- Podle pozice prvku



**Ukazatel  
na seznam**

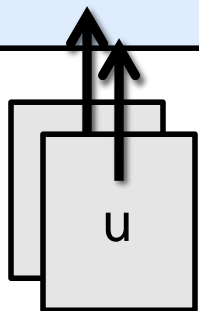
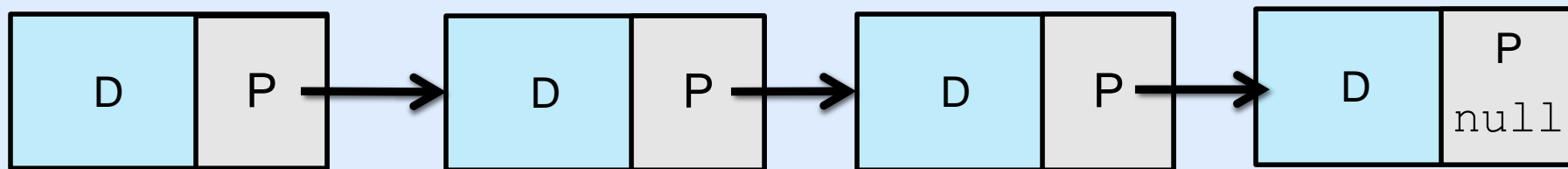
**D – data, P – ukazatel**

# Průchod seznamem

Uzel u = p;

```
while(u != null) {  
    u = u.getDalší();  
}
```

## Lineární seznam

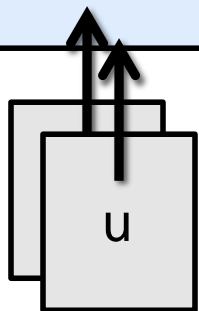
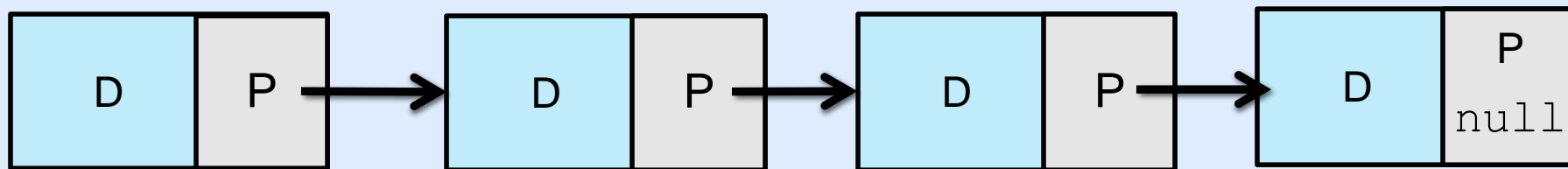


## Ukazatel na seznam

# Průchod seznamem

```
Uzel u = p;  
while(u != null) {  
    u = u.getDalší();  
}
```

## Lineární seznam

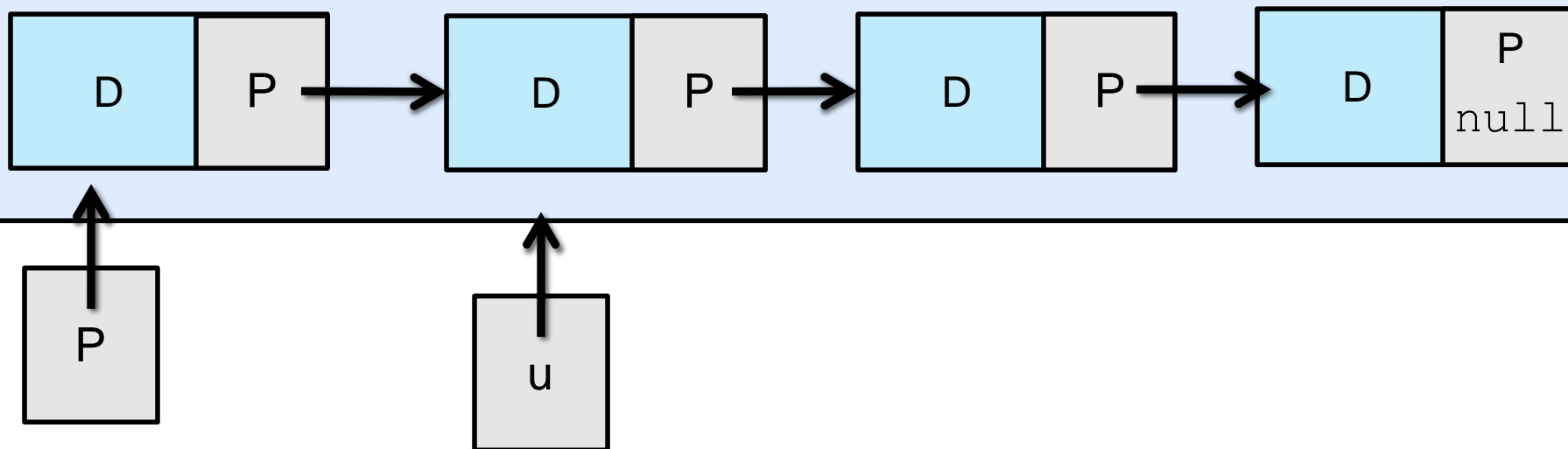


**Ukazatel  
na seznam**

# Průchod seznamem

```
Uzel u = p;  
while(u != null) {  
    u = u.getDalší();  
}
```

## Lineární seznam

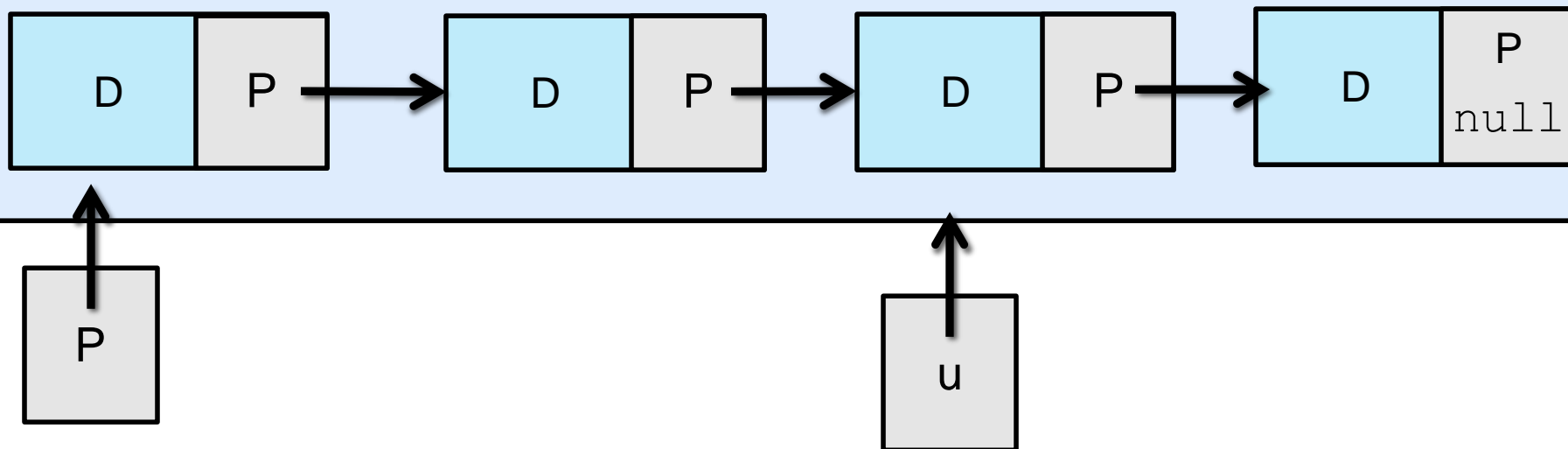


## Ukazatel na seznam

# Průchod seznamem

```
Uzel u = p;  
while(u != null) {  
    u = u.getDalší();  
}
```

## Lineární seznam



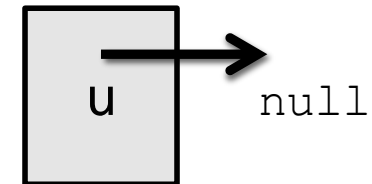
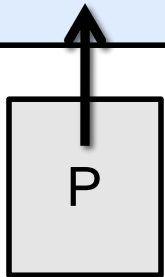
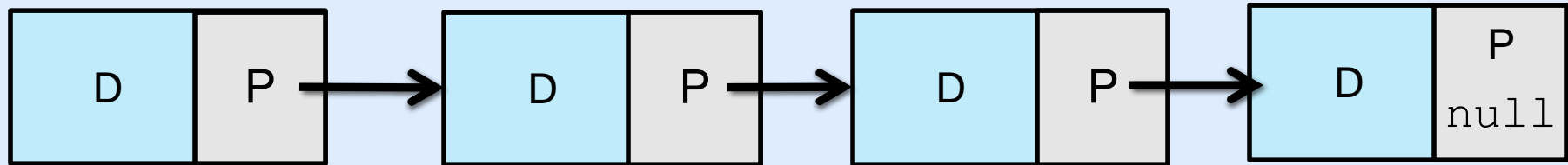
## Ukazatel na seznam



# Průchod seznamem

```
Uzel u = p;  
while(u != null) {  
    u = u.getDalší();  
}
```

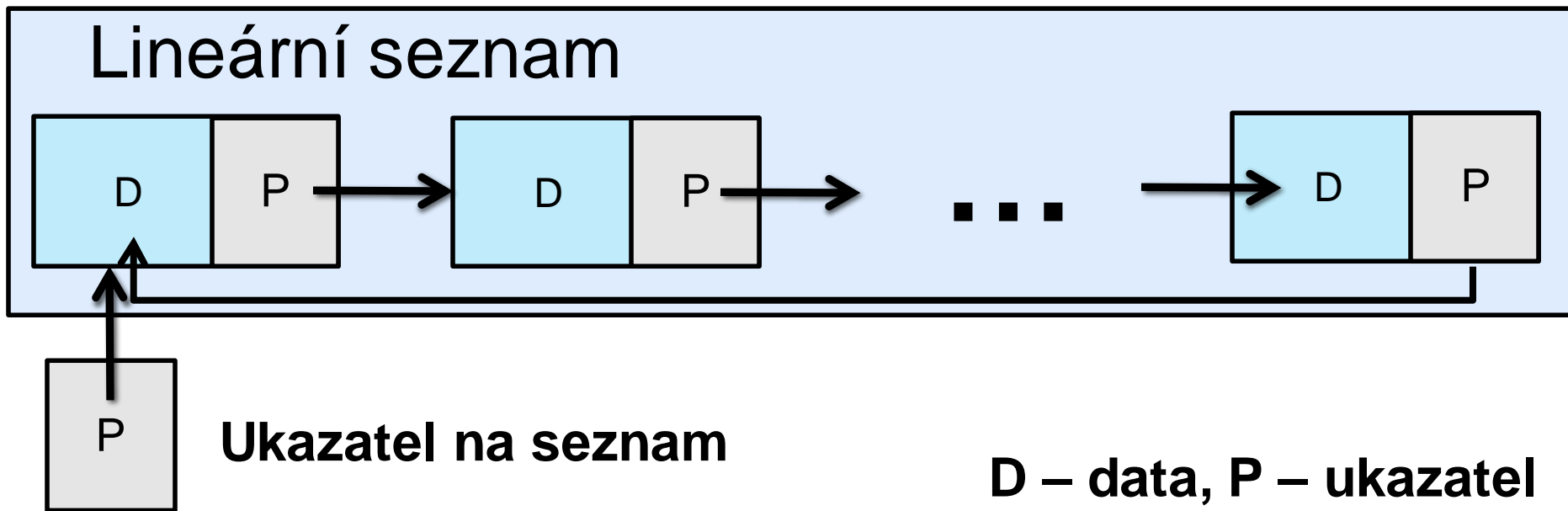
## Lineární seznam



## Ukazatel na seznam

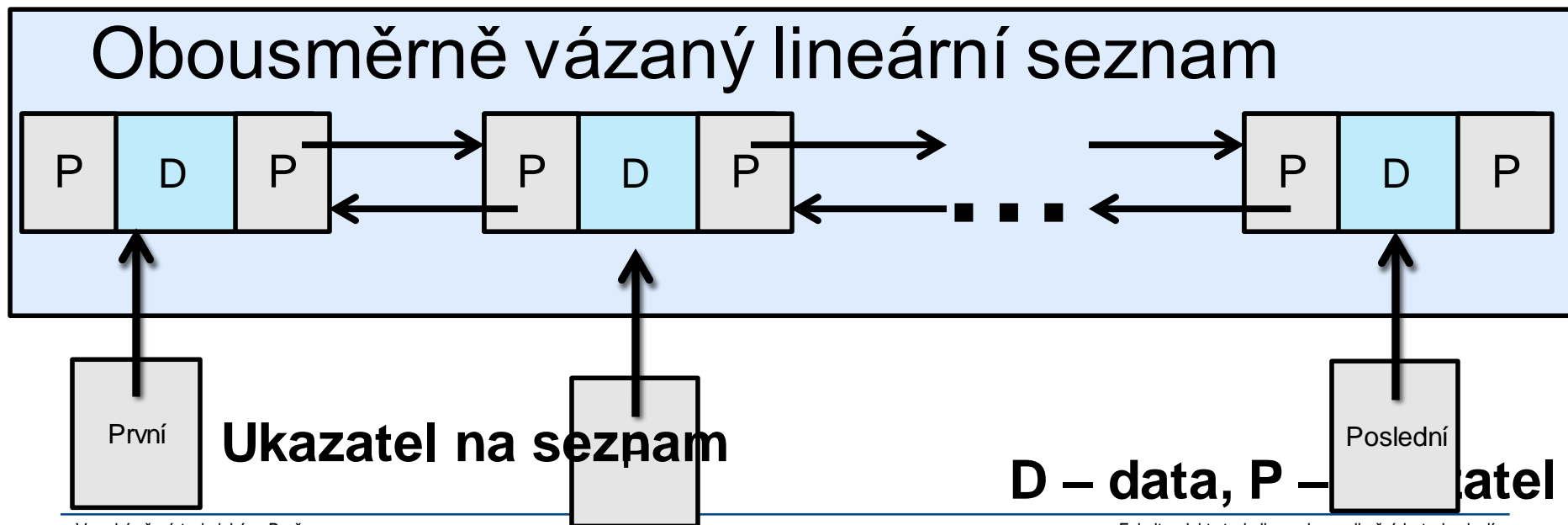
# Cyklický lineární seznam

- Stejný jako lineární, jenom poslední prvek nemá v ukazateli **null**, ale odkaz na první prvek.
- Využití – dny v týdnu



# Obousměrně vázaný lineární seznam

- Má ukazatele na další i předchozí prvek
- Umožňuje procházení v obou směrech



# Rekurze vs. iterace

## Rekurze

Ukončí se, když je dosaženo základního případu.

Každé rekurzivní volání vyžaduje další prostor v rámci zásobníku (tj. Paměti).

Pokud dostaneme nekonečnou rekurzi, nakonec nám selže na nedostatek paměti, což povede k přetečení zásobníku.

Řešení některých problémů lze snáze formulovat rekurzivně.

## Iterace

Ukončí se, pokud se ukáže, že podmínka je chybná.

Každá iterace nevyžaduje žádnou paměť navíc, protože sídlí ve stejné metodě.

Nekonečná smyčka by mohla potenciálně navždy navazovat, protože není vytvořena žádná další paměť.

Iterativní řešení problému nemusí být vždy tak přímočaré jako rekurzivní řešení.

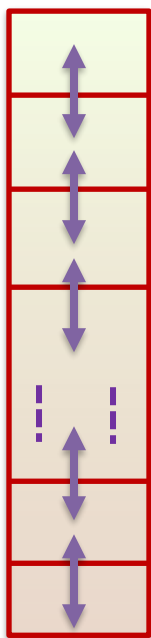
# Srovnání pole vs. seznam

Operace	Pole	Seznam
size, isEmpty	1	1
atRank, rankOf, elemAtRank	1	<i>n</i>
first, last, before, after	1	1
replaceElement, swapElements	1	1
replaceAtRank	1	<i>n</i>
insertAtRank, removeAtRank	<i>n</i>	<i>n</i>
insertFirst, insertLast	1	1
insertAfter, insertBefore	<i>n</i>	1
remove	<i>n</i>	1

# Řazení

- Jaký je čas nalezení v obecném (neseřazeném) seznamu?
- Jaký je čas nalezení v seřazeném seznamu?
- Insertion sort
- **Bubble sort** (paralelní algoritmy)
- Select sort
- Shell sort
- **Merge sort**
- **Quicksort** (JAVA, C++, ...)

# Bubble Sort



V každé iteraci se těžší buňky probublají níže

Postupujeme odspodu nahoru

Řazení vyžaduje  $n-1$  průchodů.

- Vždy porovnáváme sousední dvojice, a **prohazujeme** do patřičného pořadí
- V každém průchodu největší z prvků probublávají nahoru (resp. doprava).

# Bubble Sort

Průchod: 1

x: 

3	12	-5	6	72	21	-7	45
---	----	----	---	----	----	----	----




x: 

3	12	-5	6	72	21	-7	45
---	----	----	---	----	----	----	----





x: 

3	-5	12	6	72	21	-7	45
---	----	----	---	----	----	----	----



x: 

3	-5	6	12	72	21	-7	45
---	----	---	----	----	----	----	----



x: 

3	-5	6	12	72	21	-7	45
---	----	---	----	----	----	----	----





x: 

3	-5	6	12	21	72	-7	45
---	----	---	----	----	----	----	----




x: 

3	-5	6	12	21	-7	72	45
---	----	---	----	----	----	----	----



x: 

3	-5	6	12	21	-7	45	72
---	----	---	----	----	----	----	----





# Bubble Sort

## Průchod 2

x: 3 -5 6 12 21 -7 45 72

x: -5 3 6 12 21 -7 45 72

x: -5 3 6 12 21 -7 45 72

x: -5 3 6 12 21 -7 45 72

x: -5 3 6 12 21 -7 45 72

x: -5 3 6 12 -7 21 45 72

x: -5 3 6 12 -7 21 45 72

Takto pokračuji do seřazení všech  
tj.  $n-1$  krát

# Bubble Sort: pseudokód

```
public void bubbleSort(int[] arr) {  
    int n = arr.length;  
    for(i = 0 ... n-2) {  
        for(j = 0 ... n-2) {  
            if(arr[j] > arr[j+1]) {  
                swap(arr[j], arr[j+1]);  
            }  
        }  
    }  
}
```

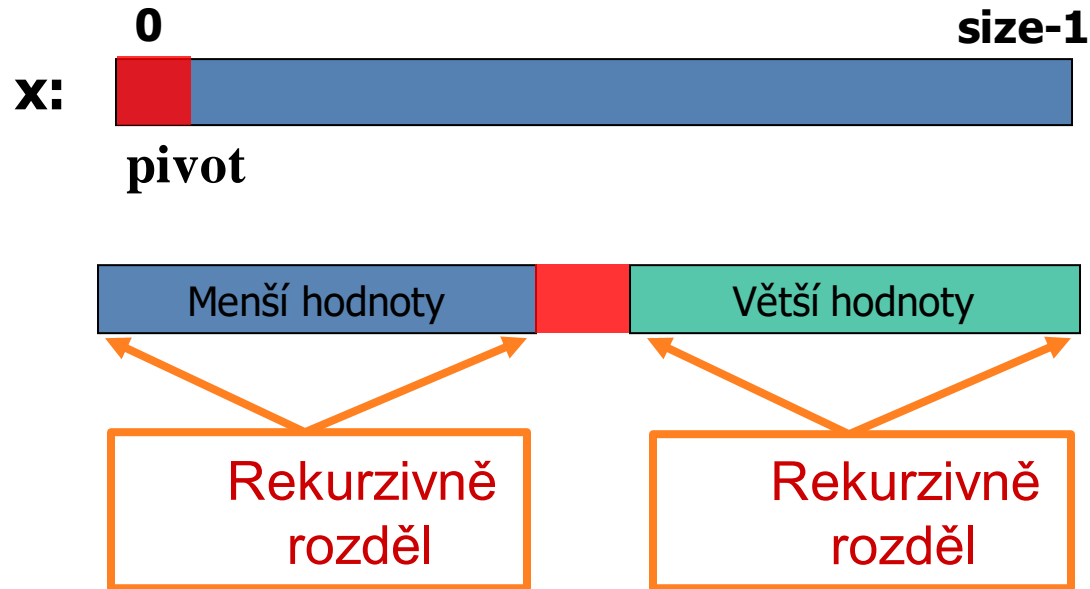
# Bubble Sort

- Nejpomalejší ze známých algoritmů řazení
- Velmi dobře paralelizovatelný

# Quick Sort – princip

- Každý krok vybíráme prvek **pivot** v poli (například první).
- Vložíme prvek do výsledné pozice seřazeného seznamu.
- Menší od pivota jdou vlevo, větší vpravo

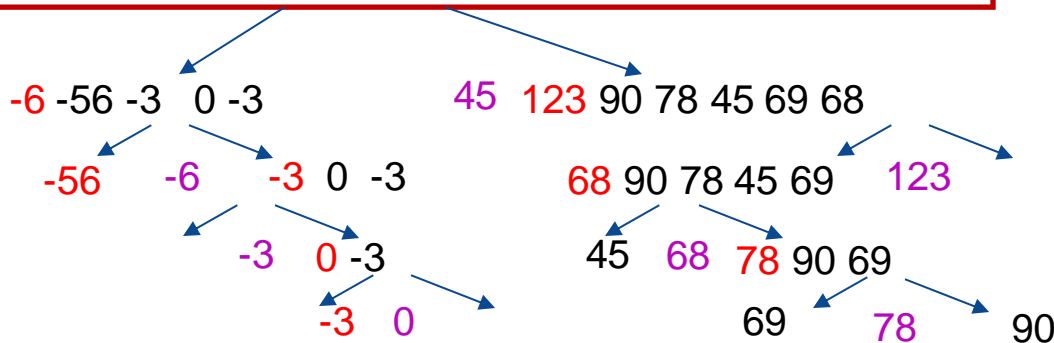
# Quick Sort - dělení



# Quick Sort

Vstup: 45 -56 78 90 -3 -6 123 0 -3 45 69 68

45 -56 78 90 -3 -6 123 0 -3 45 69 68

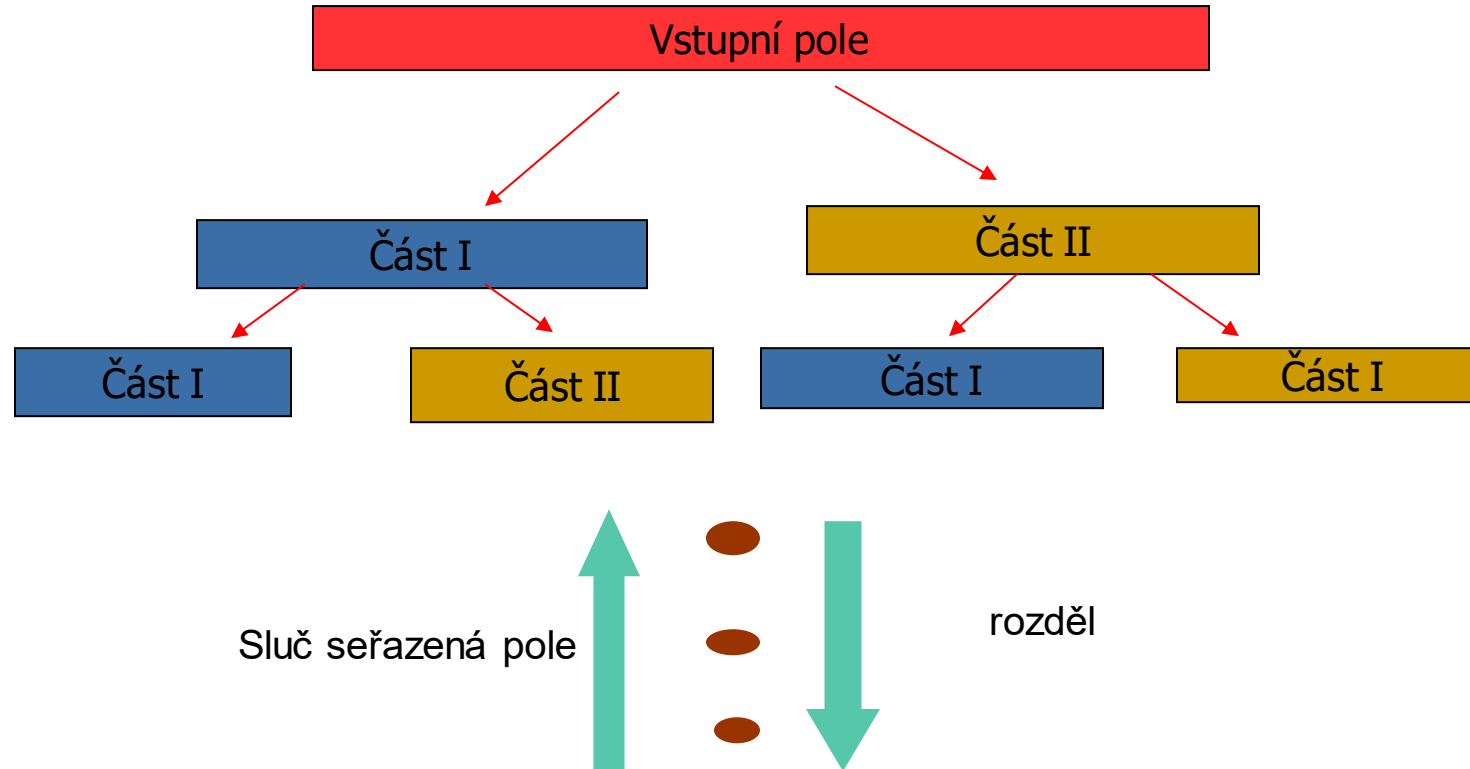


Výstup: -56 -6 -3 -3 0 45 45 68 69 78 90 123

# Quick Sort: Pseudo-kód

```
public void quickSort(int[] arr, int lowIndex, int highIndex) {  
    {  
        if (lowIndex < highIndex) {  
            /* pi is partitioning index, arr[p] is now  
               at right place */  
            int pi = partition(arr, lowIndex, highIndex);  
  
            quickSort(arr, lowIndex, pi - 1); // Before pi  
            quickSort(arr, pi + 1, highIndex); // After pi  
        }  
    }  
}
```

# Merge Sort – Jak pracuje?





# Merge Sort

x:

3	12	-5	6	72	21	-7	45
---	----	----	---	----	----	----	----

**Dělení polí**

3	12	-5	6
---	----	----	---

72	21	-7	45
----	----	----	----

3	12
---	----

-5	6
----	---

72	21
----	----

-7	45
----	----

3
---

12
----

-5
----

6
---

72
----

21
----

-7
----

45
----

3	12
---	----

-5	6
----	---

21	72
----	----

-7	45
----	----

-5	3	6	12
----	---	---	----

**Slučování dvou  
seřazených polí**

-7	21	45	72
----	----	----	----



-7	-5	3	6	12	21	45	72
----	----	---	---	----	----	----	----

# MergeSort: Pseudokód

```
public void mergeSort(int[] arr, int l, int r) {  
    if(l > r) {  
        // Find the middle point to divide the array into two halves:  
        int m = l+ (r-l)/2;  
        // Call mergeSort for first half:  
        mergeSort(arr, l, m);  
        // Call mergeSort for second half:  
        mergeSort(arr, m+1, r);  
        // Merge the two halves sorted in step 2 and 3:  
        merge(arr, l, m, r);  
    }  
}
```

# Řadící algoritmy - srovnání

- **Bubble sort**

- paralelní varianta je dnes nejrychlejší známý algoritmus pro řazení, zapotřebí  $n/2$  procesorů
- Složitost  $O(n^2)$ , v případě paralelní varianty  $O(n)$

- **Quick sort**

- hodí se pro řazení polí (lze efektivně indexovat pozice)
- nepotřebuje dodatečnou paměť ( $O(1)$ )
- Nejhorší možná složitost je  $O(n^2)$
- Není stabilní (dřívější seřazení dle jiných kritérií neponechává, pokud v novém se prvky rovnají)

- **Merge sort**

- hodí se pro efektivní řazení seznamů (nelze efektivně přistupovat k prvku na libovolné pozici)
- potřebuje dodatečnou paměť  $O(n)$
- Nejhorší možná složitost je  $O(n \log_2 n)$
- Je stabilní řadící algoritmus

# Shrnutí

- V Javě programátorovi pomáhá se správou paměti Garbage Collector, díky tomu se v porovnání s C/C++ jednodušeji píší programy
- Pole není vhodné uložistiště, pokud předem nevíme velikost ukládaných dat.
- Tento problém řeší lineární seznam, který je dynamický.

# Pojmy

- Garbage collector
- Ukazatel
- Lineární seznam
  - Jednosměrně vázaný
  - Cyklický
  - Obousměrně vázaný
- Řazení lineárních struktur
  - Bubble sort
  - Merge sort
  - Quick sort

# Děkuji za pozornost

# Vybrané otázky z pracovního pohovoru



- Je dán lineární seznam velikosti  $N$ . Úkolem je obrátit každých  $k$  uzlů (kde  $k$  je vstupem funkce) v seznamu. Pokud počet uzlů není násobkem  $k$ , pak vynechané uzly je třeba nakonec považovat za skupinu a je třeba je obrátit (vysvětlení viz příklad 2).

- Příklady:

- Vstup:

- 1->2->2->4->5->6->7->8

- $K = 4$

- Output: 4 2 2 1 8 7 6 5

Vstup:

1->2->3->4->5

$K = 3$

Output: 3 2 1 5 4

# Vybrané otázky z pracovního pohovoru

- Je dán lineární seznam, kde každý uzel představuje další lineární seznam a obsahuje dva ukazatele typu:
  - (i) Ukazatel na další uzel v hlavním seznamu
  - (ii) Ukazatel na lineární seznam, kam tento uzel směřuje
- Všechny propojené seznamy jsou seřazené, viz příklad.
- Napište funkci `flatten()`, která seznamy zploští do jediného seznamu. Zploštělý spojový seznam by měl být také seřazen.



# Příklad

5	->	10	->	19	->	28
V		V		V		V
7		20		22		35
V				V		V
8				50		40
V						V
30						45



5->7->8->10->19->20->22->28->30->35->40->45->50.

# Vybrané otázky z pracovního pohovoru

- Je dán jednosměrně vázaný lineární seznam sestávající se z  $N$  uzlů. Úkolem je odstranit z daného seznamu duplicity (uzly s opakujícími se hodnotami), pokud existují.
- Poznámka: Nepoužívat prostor navíc. Očekávaná časová složitost je  $O(N)$ . Uzly jsou na vstupu řazené.
- Příklad:
  - **Vstup:** 2->2->2->2->2
  - **Výstup:** 2

# Vybrané otázky z pracovního pohovoru

- Detekujte cyklus v lineárním seznamu
- Nalezněte  $n$ -tý prvek v seznamu od konce, se složitostí  $O(n)$
- Smažte poslední výskyt prvku v jednosměrně vázaném lineárním seznamu