# Vyhledávání

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Prezentace ke dni 12. února 2020



# Osnova přednášky

Vyhledávání

- Definice problému
- Sekvenční vyhledávání
- Vyhledávání půlením intervalu

# Vyhledávání – definice problému

## Základní pojmy

- vyhledávací prostor $S$,
- klíč,
- hodnota.

## Příklad

- seznam loginů studentů a jejich hodnocení v předmětu Algoritmy I,
- autorský katalog v knihovně,
- množina všech českých samohlásek.

# Vyhledávání – jednorozměrné vyhledávání

- adresní vyhledávací algoritmy – využívají jednoznačného vztahu mezi hodnotou klíče prvku a umístěním prvku ve struktuře reprezentující vyhledávací prostor $S$ (přímý přístup k prvkům pole pomocí indexů, hašovací tabulky)
- asociativní vyhledávací algoritmy – porovnávání prvků (relativní hodnota vzhledem k ostatním prvkům) při hledání prvku $x \in S$ se využívají relace (porovnávání) mezi prvky struktury reprezentující $S$ (vyhledávání v poli).

# Vyhledávání – vícerozměrné vyhledávání

- nevyhledáváme jen podle jednoho klíče, ale podle více klíčů,
- nutnost přizpůsobit datové struktury a algoritmy,
- samotné vyhledávací algoritmy se dají rozdělit do tří kategorií:
  - dotazy na úplnou shodu – dotazy požadující shodu na všech klíčích.
  - dotazy na částečnou shodu – dotazy požadující shodu jen na některých složkách klíče.
  - dotazy na intervalovou shodu – dotazy požadující, aby klíč záznamu byl v určeném intervalu.

Do této kategorie vlastně spadají všechny databázové systémy.

# Vyhledávání – typické úlohy

příslušnost prvku – je obsažen nebo ne? Matematicky $x \in S$ nebo $x \notin S$

hledání extrému – minimum nebo maximum

dotaz na shodu – úplnou nebo částečnou

Při konkrétní implementaci zadané úlohy existuje vzájemná souvislost mezi všemi operacemi a všechny operace závisejí na zvolené reprezentaci vyhledávacího prostoru.

# Vyhledávání v poli

- nejjednodušší paměťová struktura,
- přímý přístup k prvkům,
- typicky jednorozměrné vyhledávání (klíč, hodnota),

## Řešené úlohy

příslušnost prvku
- výsledek jako logická hodnota
- výsledek index hledaného prvku v poli

dotaz na úplnou shodu

Budeme předpokládat jeden výskyt každého prvku v poli

# Složitost vyhledávání

## Prostorová složitost

Konstantní – jen prohledávané pole a konstantní rozsah pomocných proměnných

## Časová složitost

Jaké operace zkoumat?
Porovnání hledaného prvku s prvkem v poli.

# Sekvenční vyhledávání

- bez požadavků na prohledávané pole,
- prohledávané pole tedy nemusí být setříděné,
- algoritmus prohledává celé pole – postupně testuje jednotlivé prvky pole,
- složitost $O(n)$, kde $n$ je počet prvků v prohledávaném poli.

# Iterativní implementace

```
1 bool LinearSearch1(const int a[], const int n, const int x)
2 {
3    for (int i = 0; i < n; i++)
4    {
5       if (a[i] == x)
6       {
7          return true;
8       }
9    }
10   return false;
11 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.

# Iterativní implementace

```
1 int LinearSearch2(const int a[], const int n, const int x)
2 {
3    for (int i = 0; i < n; i++)
4    {
5       if (a[i] == x)
6       {
7          return i;
8       }
9    }
10   return -1;
11 }
```

Funkce vrací index prvního výskytu prvku `x` v poli `a`, jinak vrací -1.

# Rekurzivní implementace

```
1 bool LinearSearchRecursive1(const int a[], const int n, const int x,
      const int i)
2 {
3    if (i == n)
4    {
5       return false;
6    }
7    if (a[i] == x)
8    {
9       return true;
10   }
11   return LinearSearchRecursive1(a, n, x, i + 1);
12 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.

# Rekurzivní implementace

```
1 int LinearSearchRecursive2(const int a[], const int n, const int x,
      const int i)
2 {
3    if (i == n)
4    {
5       return -1;
6    }
7    if (a[i] == x)
8    {
9       return i;
10   }
11   return LinearSearchRecursive2(a, n, x, i + 1);
12 }
```

Funkce vrací index prvního výskytu prvku `x` v poli `a`, jinak vrací -1.

# Rekurzivní implementace, alternativní přístup

```
1 bool LinearSearchRecursive3(const int a[], const int l, const int r,
      const int x)
2 {
3    if (l > r)
4    {
5       return false;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10      return true;
11   }
12   return LinearSearchRecursive3(a, l, m - 1, x) ||
         LinearSearchRecursive3(a, m + 1, r, x);
13 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.

# Rekurzivní implementace, alternativní přístup

```
1 bool LinearSearchRecursive4(const int a[], const int l, const int r,
      const int x)
2 {
3    if (l > r)
4    {
5       return false;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10      return true;
11   }
12   if (LinearSearchRecursive4(a, l, m - 1, x))
13   {
14      return true;
15   }
16   return LinearSearchRecursive4(a, m + 1, r, x);
17 }
```

# Rekurzivní implementace, alternativní přístup (pokrač.)

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.

# Vyhledávání půlením intervalu

- Lze zlepšit časovou složitost sekvenčního vyhledávání?
- Pokud o prohledávaném poli nevíme vůbec nic asi těžko?
- Co kdybychom pole setřídili? Budeme předpokládat, že pole je setříděno vzestupně.
- Potom bychom nemuseli prohledávat pole celé. Po nalezení většího prvku než je hledaný už nemá cenu dále hledat.
- Pokud použijeme techniku dělení pole na poloviny, musíme stále prohledávat obě poloviny?
- Mohlo by nám uspořádání prvků v poli napovědět v které polovině máme hledat a kde to nemá cenu?

# Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 16

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 16

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 23

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 16

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 23

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 19

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Prvek 19 byl úspěšně nalezen!**

# Půlení intervalu, úspěšné hledání prvku 19

Zkráceně můžeme postup hledání prvku 19 zachytit takto:

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Poznámka**

Celé pole můžeme považovat také za „úsek pole“, který je shodou okolností roven celému poli. Tím se nám zjednoduší další výklad a v důsledku i výsledný algoritmus.

# Půlení intervalu, úspěšné hledání prvku 7

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Půlení intervalu, úspěšné hledání prvku 65

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Půlení intervalu – úsek pole

## Jak reprezentovat úsek pole?

L – index levého okraje prohledávaného úseku pole, **L**eft
R – index pravého okraje prohledávaného úseku pole, **R**ight
M – index středu prohledávaného úseku pole, **M**id

## Indexy L a R

Indexy **L** a **R** jsou inkluzivní – první a poslední prvek úseku leží na těchto indexech.
Srovnej s délkou pole a indexem posledního prvku pole v jazyku C++.

## Poznámka

Index levého resp. pravého okraje úseku pole budeme také nazývat levou resp. pravou **mezí** úseku pole.

# Půlení intervalu – výpočet středu úseku

## Výpočet středu úseku

$$M = L + \frac{R - L}{2} = \frac{2L + R - L}{2} = \frac{L + R}{2}$$

## Konvence při výpočtu středu úseku

- Je jasná, že součet $L + R$ nemusí být sudé číslo. Co s tím?!
- Dělení vždy probíhá **celočíselně**! Například $L = 0$ a $R = 5$. Střed je $M = 2$, nikoliv $M = 2.5$.
- Levá část děleného úseku tak bude o jeden prvek kratší.
- Pokud bychom dělili s plovoucí řádovou čárkou a zaokrouhlovali nahoru, jak je obvyklé, dosáhli bychom jedině toho, že levá část by byla o jeden prvek delší než pravá.

# Půlení intervalu, úspěšné hledání prvku 19

## Úseky pole

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |

## Indexy

| L | R | M |
|---|---|---|
| 0 | 12 | 6 |
| 0 | 5 | 2 |
| 3 | 5 | 4 |
| 3 | 3 | 3 |

## Posun doleva

| | Nová hodnota |
|---|---|
| L | beze změny |
| R | $M - 1$ |

## Posun doprava

| | Nová hodnota |
|---|---|
| L | $M + 1$ |
| R | beze změny |

# Půlení intervalu, neúspěšné hledání prvku 55

Určíme střed celého pole a srovnáme 55 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 55 s 52

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 55 s 61

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 55 s 56

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Prvek 55 nebyl nalezen!**

# Půlení intervalu, neúspěšné hledání prvku 55

## Jak rozpoznat konec u neúspěšného hledání?

- Konec rozpoznáme snadno – úsek obsahuje jen jeden prvek a není to ten, který hledáme.
- Ale! Při úspěšném hledání prvku 19 jsme taky skončili u jednoprvkového úseku.
- Jsou tedy jednoprvkové úseky nějakou zvláštností? Proč by měly?
- Takže bychom měli vyhledávání ukončit až úsek zkrátíme na jeden prvek?
- A když se u více prvkového úseku „trefíme“ tak, že hledaný prvek bude přímo ve středu. To bychom mohli skončit rovnou, ne?
- Takže redukce na jeden prvek není asi nutná.
- A opačná otázka? Jak poznám, že mám ve vyhledávání naopak stále pokračovat?

# Půlení intervalu, invariant algoritmu

- Při výpočtu středu úseku

$$M = L + \frac{R - L}{2}$$

  předpokládáme, že $R \geq L$!
- Otázka zní, platí tento předpoklad po celou dobu vykonávání algoritmu?
- A pokud toto neplatí, kdy je předpoklad porušen?

## Invariant algoritmu

**Invariant** je podmínka v algoritmu, která musí být splněna po celou dobu vykonávání algoritmu.

## Invariant algoritmu vyhledávání půlením intervalu

V našem případě musí vždy platit, že $R \geq L$.

# Půlení intervalu, neúspěšné hledání prvku 55

## Úseky pole

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Prvek 55 je menší než 56, „pokračujeme“ levou částí

## Indexy

| L | R | M |
|---|---|---|
| 0 | 12 | 6 |
| 7 | 12 | 9 |
| 10 | 12 | 11 |
| 10 | 10 | 10 |
| **10** | **9** | |

## Porušení invariantu algoritmu

Levá mez je větší než pravá. Došlo k překřížení mezí.

# Odhad časové složitosti

- Výchozí poznatky:
  1. prohledávaný úsek pole je opakovaně dělen na poloviny,
  2. dělení končí u úseku o jednom prvku,
  3. každé dělení je provedeno po porovnání jednoho prvku.
- Počet porovnání tedy odpovídá počtu dělení pole.
- Kolik takových dělení je ale nutné?
- Předpokládejme, že počet prvků v poli $n$ je mocninou čísla 2, tj. $n = 2^k$. Pokud ne, vezmeme nejbližší větší mocninu tj. $2^{k-1} < n \leq 2^k$.

# Odhad časové složitosti (pokrač.)

| Pořadí dělení | Pole rozděleno na úseků | Počet prvků v úseku | Porovnáno prvků v úseku |
|---|---|---|---|
| 0 | $1 = 2^0$ | $\frac{n}{1}$ | 1 |
| 1 | $2 = 2^1$ | $\frac{n}{2}$ | 1 |
| 2 | $4 = 2^2$ | $\frac{n}{4}$ | 1 |
| 3 | $8 = 2^3$ | $\frac{n}{8}$ | 1 |
| 4 | $16 = 2^4$ | $\frac{n}{16}$ | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| k | $2^k$ | $\frac{n}{2^k} = 1$ | 1 |

# Odhad časové složitosti (pokrač.)

- Po provedení $k+1$ dělení pole jsme se dostali k úseku s jediným prvkem a provedli jsme při tom celkem $k+1$ porovnání.
- Horní odhad složitosti vyhledávání půlením intervalu tedy je $O(k+1)$.
- Jak vypočítáme $k$? Řešením rovnice

$$\begin{aligned} n &= 2^k \\ \log_2 n &= k \log_2 2 \\ \log_2 n &= k \end{aligned}$$

- Odhad složitosti tedy můžeme psát jako $O(\log_2 n + 1)$.
- Vzhledem k tomu, že rozhodující člen této funkce je výraz $\log_2 n$, můžeme psát zjednodušeně

$$O(\log_2 n)$$

# Rekurzivní implementace

```
1 bool BinarySearchRecursive1(const int a[], const int l, const int r,
      const int x)
2 {
3    if (l > r)
4    {
5       return false;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10      return true;
11   }
12   if (x < a[m])
13   {
14      return BinarySearchRecursive1(a, l, m - 1, x);
15   }
16   return BinarySearchRecursive1(a, m + 1, r, x);
17 }
```

# Rekurzivní implementace (pokrač.)

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.

# Rekurzivní implementace

```
1 bool BinarySearchRecursive1a(const int a[], const int l, const int r,
      const int x)
2 {
3    if (l > r)
4    {
5       return false;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10      return true;
11   }
12   return x < a[m] ? BinarySearchRecursive1a(a, l, m - 1, x) :
         BinarySearchRecursive1a(a, m + 1, r, x);
13 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`. Místo příkazu `if` použit ternární operátor.

# Rekurzivní implementace

```
1 int BinarySearchRecursive2(const int a[], const int l, const int r,
      const int x)
2 {
3    if (l > r)
4    {
5       return -1;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10      return m;
11   }
12   if (x < a[m])
13   {
14      return BinarySearchRecursive2(a, l, m - 1, x);
15   }
16   return BinarySearchRecursive2(a, m + 1, r, x);
17 }
```

# Rekurzivní implementace (pokrač.)

Výsledkem je index nalezeného prvku, jinak -1.

# Rekurzivní implementace

```
1 int BinarySearchRecursive3(const int a[], const int l, const int r,
      const int x)
2 {
3    if (l > r)
4    {
5       return ~l;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10      return m;
11   }
12   if (x < a[m])
13   {
14      return BinarySearchRecursive3(a, l, m - 1, x);
15   }
16   return BinarySearchRecursive3(a, m + 1, r, x);
17 }
```

# Rekurzivní implementace (pokrač.)

Výsledkem je index nalezeného prvku, jinak bitový doplněk správné pozice.

# Iterativní implementace

```
1  bool BinarySearch1(const int a[], const int n, const int x)
2  {
3     int l = 0;
4     int r = n - 1;
5     while (l <= r)
6     {
7        int m = (l + r) / 2;
8        if (x == a[m])
9        {
10          return true;
11       }
12       if (x < a[m])
13       {
14          r = m - 1;
15       }
16       else
17       {
18          l = m + 1;
19       }
```

# Iterativní implementace (pokrač.)

```
20      }
21      return false;
22  }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.

# Iterativní implementace

```
1  int BinarySearch2(const int a[], const int n, const int x)
2  {
3     int l = 0;
4     int r = n - 1;
5     while (l <= r)
6     {
7        int m = (l + r) / 2;
8        if (x == a[m])
9        {
10          return m;
11       }
12       if (x < a[m])
13       {
14          r = m - 1;
15       }
16       else
17       {
18          l = m + 1;
19       }
```

# Iterativní implementace (pokrač.)

```
20      }
21      return -1;
22  }
```

Výsledkem je index nalezeného prvku, jinak -1.

# Iterativní implementace

```
1  int BinarySearch3(const int a[], const int n, const int x)
2  {
3     int l = 0;
4     int r = n - 1;
5     while (l <= r)
6     {
7        int m = (l + r) / 2;
8        if (x == a[m])
9        {
10          return m;
11       }
12       if (x < a[m])
13       {
14          r = m - 1;
15       }
16       else
17       {
18          l = m + 1;
19       }
```

# Iterativní implementace (pokrač.)

```
20      }
21      return ~l;
22  }
```

Výsledkem je index nalezeného prvku, jinak bitový doplněk správné pozice.

# Děkuji za pozornost