# 08. Vyhledávání

## Obsah



## Sekvenční vyhledávání v poli

Sekvenční vyhledávání (též lineární vyhledávání) je nejjednodušší algoritmus pro vyhledávání hodnoty v poli. Funguje na principu postupného procházení všech prvků pole jeden po druhém.

### Princip algoritmu

1. Postupně procházíme prvky pole od začátku do konce
2. Každý prvek porovnáme s hledanou hodnotou
3. Pokud najdeme shodu, vrátíme index prvku
4. Pokud projdeme celé pole bez nalezení shody, vrátíme informaci, že prvek nebyl nalezen

### Pseudokód

```
function sequentialSearch(array, target):
    for i = 0 to array.length - 1:
        if array[i] == target:
            return i  // Našli jsme prvek, vracíme jeho index
    return -1  // Prvek nebyl nalezen
```

### Složitost

- **Časová složitost**:
  - Nejlepší případ: O(1) - prvek je hned na začátku pole
  - Průměrný případ: O(n/2) ≈ O(n) - prvek je uprostřed pole
  - Nejhorší případ: O(n) - prvek je na konci pole nebo není v poli vůbec
- **Prostorová složitost**: O(1) - není potřeba žádná dodatečná paměť

### Výhody a nevýhody

- **Výhody**:
  - Jednoduchá implementace
  - Funguje na neseřazených polích
  - Nevyžaduje žádné speciální podmínky nebo předpoklady o datech
- **Nevýhody**:
  - Neefektivní pro velké datové sady
  - Lineární časová složitost

## Algoritmus půlení intervalů (binární vyhledávání)

Binární vyhledávání je efektivní algoritmus pro vyhledávání v seřazeném poli. Využívá strategii "rozděl a panuj" k opakovanému dělení prohledávaného prostoru na poloviny.

### Princip algoritmu

1. Začínáme s intervalem zahrnujícím celé pole
2. Porovnáme hledanou hodnotu s prostředním prvkem intervalu
3. Pokud je hledaná hodnota menší, pokračujeme s levou polovinou
4. Pokud je hledaná hodnota větší, pokračujeme s pravou polovinou
5. Pokud je hledaná hodnota rovna prostřednímu prvku, vyhledávání končí
6. Opakujeme, dokud není prvek nalezen nebo interval prázdný

### Pseudokód

```
function binarySearch(array, target):
    left = 0
    right = array.length - 1

    while left <= right:
        mid = (left + right) / 2  // celočíselné dělení

        if array[mid] == target:
            return mid  // Našli jsme prvek
        else if array[mid] < target:
            left = mid + 1  // Hledáme v pravé polovině
        else:
            right = mid - 1  // Hledáme v levé polovině

    return -1  // Prvek nebyl nalezen
```

### Složitost

- **Časová složitost**:
  - Nejlepší případ: O(1) - prvek je hned uprostřed
  - Průměrný a nejhorší případ: O(log n) - logaritmická složitost díky opakovanému dělení intervalu na poloviny
- **Prostorová složitost**:
  - Iterativní implementace: O(1)
  - Rekurzivní implementace: O(log n) kvůli zásobníku volání

### Výhody a nevýhody

- **Výhody**:
  - Velmi efektivní pro velké datové sady
  - Logaritmická časová složitost
- **Nevýhody**:
  - Vyžaduje seřazené pole
  - Neefektivní pro malé datové sady (režie je větší než u sekvenčního vyhledávání)
  - Nevhodné pro dynamické kolekce, kde se data často mění (nutnost opětovného řazení)

## Binární vyhledávací strom

Binární vyhledávací strom (BST) je datová struktura založená na binárním stromu s vlastností, která umožňuje efektivní vyhledávání, vkládání a mazání prvků.

### Vlastnosti

- Každý uzel obsahuje hodnotu a odkazy na levý a pravý podstrom
- Pro každý uzel platí:
  - Všechny hodnoty v levém podstromu jsou menší než hodnota uzlu
  - Všechny hodnoty v pravém podstromu jsou větší než hodnota uzlu
- Každý podstrom je sám o sobě také binární vyhledávací strom

### Základní operace

### Vyhledávání hodnoty

```
function search(root, value):
    if root is null or root.value == value:
        return root

    if value < root.value:
        return search(root.left, value)
    else:
        return search(root.right, value)
```

### Vkládání hodnoty

```
function insert(root, value):
    if root is null:
        return new Node(value)

    if value < root.value:
        root.left = insert(root.left, value)
    else if value > root.value:
        root.right = insert(root.right, value)

    return root  // Vracíme nezměněný uzel, pokud hodnota již existuje
```

### Mazání hodnoty

```
function delete(root, value):
    if root is null:
        return null

    if value < root.value:
        root.left = delete(root.left, value)
    else if value > root.value:
        root.right = delete(root.right, value)
    else:
        // Případ 1: Uzel je list (nemá potomky)
        if root.left is null and root.right is null:
            return null

        // Případ 2: Uzel má jednoho potomka
        if root.left is null:
            return root.right
        if root.right is null:
            return root.left

        // Případ 3: Uzel má dva potomky
        // Najdeme nejmenší hodnotu v pravém podstromu
        root.value = findMin(root.right)
        // Odstraníme uzel s nejmenší hodnotou z pravého podstromu
        root.right = delete(root.right, root.value)

    return root

function findMin(node):
    current = node
    while current.left is not null:
        current = current.left
```

```
    return current.value
```

### Složitost

- **Časová složitost** (pro vyvážený strom):
  - Vyhledávání: O(log n)
  - Vkládání: O(log n)
  - Mazání: O(log n)
- **Časová složitost** (pro nevyvážený strom, nejhorší případ):
  - Všechny operace: O(n) - strom se může zvrhnout v lineární seznam

## Metody vyvažování binárních stromů

Nevyvážený binární vyhledávací strom může degradovat na lineární seznam, což vede ke zhoršení časové složitosti operací. Pro udržení efektivnosti stromu existují metody vyvažování.

### AVL stromy

AVL strom je binární vyhledávací strom, kde rozdíl výšek levého a pravého podstromu každého uzlu je nejvýše 1.

### Rotace pro vyvážení

- **Jednoduchá rotace vlevo**
- **Jednoduchá rotace vpravo**
- **Dvojitá rotace (nejdřív vpravo, potom vlevo)**
- **Dvojitá rotace (nejdřív vlevo, potom vpravo)**

```
function rightRotate(y):
    x = y.left
    T2 = x.right

    // Provedení rotace
    x.right = y
    y.left = T2

    // Aktualizace výšek
    y.height = max(height(y.left), height(y.right)) + 1
    x.height = max(height(x.left), height(x.right)) + 1

    return x  // Nový kořen
```

### Červeno-černé stromy

Červeno-černý strom je binární vyhledávací strom, kde každý uzel má barvu (červenou nebo černou) a splňuje specifická pravidla pro zajištění vyváženosti.

### Pravidla červeno-černého stromu

1. Každý uzel je buď červený, nebo černý
2. Kořen je černý
3. Všechny listové uzly (NULL) jsou černé
4. Pokud je uzel červený, jeho potomci musí být černí
5. Pro každý uzel platí, že všechny cesty z tohoto uzlu k listovým uzlům obsahují stejný počet černých uzlů

### B-stromy

B-strom je vyvážený vyhledávací strom, který umožňuje, aby uzly měly více než dva potomky. Je navržen pro efektivní práci s blokovými úložišti (např. disky).

### Vlastnosti B-stromu

- Každý uzel kromě kořene má alespoň t-1 klíčů a nejvýše 2t-1 klíčů
- Všechny listové uzly jsou ve stejné hloubce
- Vložení a mazání zajišťuje, že strom zůstává vyvážený

## Použití hash funkce

Hash funkce převádí data (klíč) na celočíselnou hodnotu (hash), která se používá jako index pro ukládání a vyhledávání dat v hash tabulce.

### Princip hashování

1. Data (klíč) jsou převedena hash funkcí na celočíselnou hodnotu
2. Tato hodnota slouží jako index v poli (hash tabulce)
3. Při vyhledávání se hash znovu vypočítá a použije se pro přímý přístup k datům

### Kolize

Kolize nastává, když různé klíče produkují stejný hash. Existuje několik metod řešení kolizí:

**Řešení kolizí pomocí zřetězení (chaining)** Každá pozice v hash tabulce obsahuje odkaz na seznam prvků, které mapují na daný index.

```
function hashInsert(table, key, value):
    index = hashFunction(key) % table.size

    // Vytvoříme nový záznam
    entry = new Entry(key, value)

    // Přidáme ho do seznamu na dané pozici
    table[index].add(entry)
```

**Řešení kolizí pomocí otevřeného adresování** Při kolizi hledáme jinou volnou pozici v tabulce: - **Lineární sondování**: zkoušíme postupně následující pozice - **Kvadratické sondování**: používáme kvadratický vzorec pro výpočet skoku - **Dvojité hashování**: používáme druhou hash funkci pro výpočet skoku

### Složitost

- **Časová složitost** (průměrný případ):
  - Vyhledávání: O(1)
  - Vkládání: O(1)
  - Mazání: O(1)
- **Časová složitost** (nejhorší případ - mnoho kolizí):
  - Všechny operace: O(n)

### Využití hash tabulek

- **HashMap/HashSet** - implementace mapy nebo množiny
- **Inverzní pole** - pro data v omezeném rozsahu, kde hodnota dat slouží jako index

## Písmenkový strom (trie)

Trie (také nazývaný písmenkový strom nebo prefixový strom) je stromová datová struktura pro ukládání a vyhledávání řetězců, kde každá hrana reprezentuje jeden znak.

### Vlastnosti

- Každý uzel může mít až k potomků, kde k je velikost abecedy
- Cesta od kořene k uzlu reprezentuje prefix jednoho nebo více řetězců
- Každý uzel může obsahovat příznak, zda zde končí celé slovo

### Základní operace

#### Vkládání řetězce

```
function insert(root, word):
    current = root

    for i = 0 to word.length - 1:
        char = word[i]
        if current.children[char] is null:
            current.children[char] = new TrieNode()
        current = current.children[char]

    current.isEndOfWord = true  // Označení konce slova
```

#### Vyhledávání řetězce

```
function search(root, word):
    current = root

    for i = 0 to word.length - 1:
        char = word[i]
        if current.children[char] is null:
            return false  // Řetězec nenalezen
        current = current.children[char]

    return current.isEndOfWord  // True, pokud celý řetězec existuje
```

#### Vyhledávání prefixu

```
function startsWith(root, prefix):
    current = root

    for i = 0 to prefix.length - 1:
        char = prefix[i]
        if current.children[char] is null:
            return false  // Prefix nenalezen
        current = current.children[char]

    return true  // Prefix existuje
```

### Složitost

- **Časová složitost**:
  - Vyhledávání: O(m), kde m je délka hledaného řetězce
  - Vkládání: O(m), kde m je délka vkládaného řetězce
  - Mazání: O(m), kde m je délka mazaného řetězce
- **Prostorová složitost**: O(n × m), kde n je počet řetězců a m je průměrná délka řetězce

## Efektivita algoritmů vyhledávání

| Algoritmus | Průměrná složitost | Nejhorší složitost | Prostorová složitost |
|---|---|---|---|
| Sekvenční vyhledávání | O(n) | O(n) | O(1) |
| Binární vyhledávání | O(log n) | O(log n) | O(1) |
| Vyhledávání v BST | O(log n) | O(n) | O(n) |
| Vyhledávání v AVL/RB stromu | O(log n) | O(log n) | O(n) |
| Vyhledávání v hash tabulce | O(1) | O(n) | O(n) |
| Vyhledávání v trie | O(m) | O(m) | O(n × m) |

## Příklady použití

### Sekvenční vyhledávání

- Vyhledávání v malých nebo neseřazených kolekcích
- Jednoduché skripty a aplikace s malým objemem dat

### Binární vyhledávání

- Vyhledávání ve slovnících a telefonních seznamech
- Vyhledávání v seřazených databázích
- Implementace funkce `binary_search` ve standardních knihovnách

### Binární vyhledávací strom

- Implementace množin a map (např. `TreeSet`, `TreeMap` v Javě)
- Udržování hierarchicky uspořádaných dat
- Databázové indexy

### Hash tabulky

- Implementace množin a map (např. `HashSet`, `HashMap` v Javě)
- Ukládání asociativních dat
- Rychlé vyhledávání v rozsáhlých datových sadách
- Implementace cache
- Detekce duplicit

### Trie

- Prediktivní text a automatické doplňování
- Kontrola pravopisu
- Vyhledávání řetězců v textu
- IP routování (využití v PATRICIA trie)
- Ukládání slovníků a lexikonů