

Datové struktury

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

03.12.2025

CC BY-NC-SA 4.0

Úvod

Co jsou datové struktury?

Datová struktura je způsob organizace a ukládání dat v počítači

- Definuje vztahy mezi daty
- Určuje operace, které lze s daty provádět
- Ovlivňuje **efektivitu** algoritmů
- Volba správné datové struktury = rychlejší a úspornější program

Co jsou datové struktury?

Představte si knihovnu

Bez organizace:

- Knihy v náhodném pořadí
- Hledání knihy trvá hodiny

S organizací:

- Knihy seřazené podle autora/žánru
- Rychlé vyhledání pomocí katalogu

Proč jsou datové struktury důležité?

Jaké výhody podle vás přináší správná volba datové struktury?

- **Optimalizace času** - rychlejší operace
- **Optimalizace paměti** - úsporné ukládání dat
- **Řešení specifických problémů** - každá struktura pro jiný účel
- **Základ pro algoritmy** - mnoho algoritmů závisí na datových strukturách

Proč jsou datové struktury důležité?

Příklad - vyhledávání v datech:

```
// Array with 1,000,000 elements
const array: number[] = [1, 2, 3, ..., 1000000];
```

```
// Linear search - average 500,000 steps
array.find((x: number) => x === 999999);
```

```
// Binary tree - only ~20 steps!
binarySearchTree.find(999999);
```

Zásobník (Stack)

Zásobník

Zásobník je datová struktura založená na principu LIFO

LIFO = Last In, First Out

Poslední přidaný prvek je první odebraný

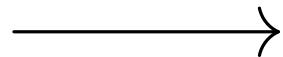
Zásobník

Představte si naskládané talíře

- Nové talíře se přidávají na vrchol
- Talíře se odebírají z vrcholu

Zásobník

Poslední přidaný



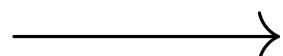
Talíř 5

Talíř 4

Talíř 3

Talíř 2

První přidaný



Talíř 1

Zásobník

Základní operace:

- `push(item)` - přidá prvek na vrchol
- `pop()` - odebere a vrátí prvek z vrcholu
- `peek()` - zobrazí prvek na vrcholu (bez odebrání)
- `isEmpty()` - kontrola, zda je zásobník prázdný

Implementace v TypeScriptu

```
class Stack<T> {  
    private items: T[] = [];  
  
    push(item: T): void { this.items.push(item); }  
  
    pop(): T | undefined { return this.items.pop(); }  
  
    peek(): T | undefined { return this.items.at(-1); }  
  
    isEmpty(): boolean { return this.items.length === 0; }  
}
```

Call stack - Implicitní zásobník

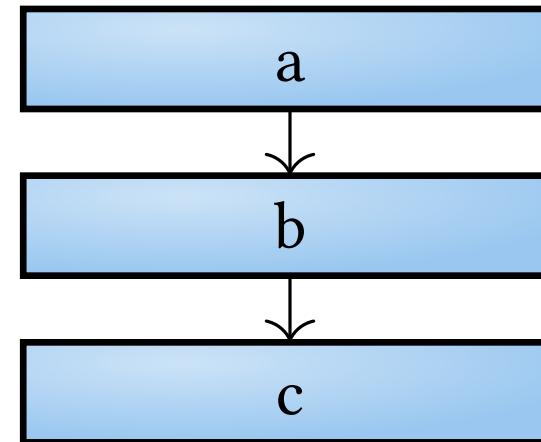
```
function a(): void { b(); }
```

```
function b(): void { c(); }
```

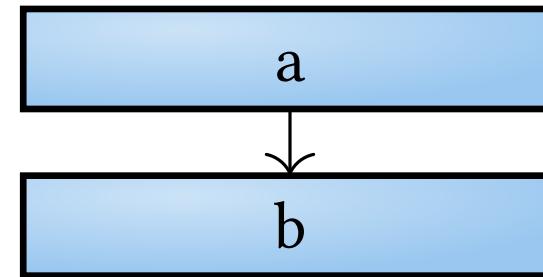
```
function c(): void { console.log("Done"); }
```

```
a();
```

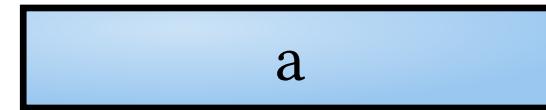
Call stack - Implicitní zásobník



Call stack - Implicitní zásobník



Call stack - Implicitní zásobník



Příklad - Validace závorek

```
checkBrackets("({[]})"); // true  
checkBrackets("({[}]")"; // false
```

```
function checkBrackets(str: string): boolean {  
    const stack = new Stack<string>();  
    const pairs: Record<string, string> = {  
        '(': ')', '[': ']', '{': '}'  
    };  
  
    // TODO: Live implementation now
```

Příklad - Validace závorek

```
for (const char of str) {  
    if (char in pairs) {  
        stack.push(char); // Opening bracket  
    } else if (Object.values(pairs).includes(char)) {  
        const last = stack.pop();  
        if (!last || pairs[last] !== char) return false;  
    }  
}  
return stack.isEmpty();  
}
```

Další příklady zásobníku

- **Undo/Redo** operace
- **Procházení do hloubky** (DFS) v grafech
- **Parsování** výrazů a vyhodnocování postfixové notace
- **Překladače** a **interpreti** programovacích jazyků
- **Historie** prohlížeče (tlačítko Zpět)

Fronta (Queue)

Fronta

Fronta je datová struktura založená na principu **FIFO**

FIFO = First In, First Out

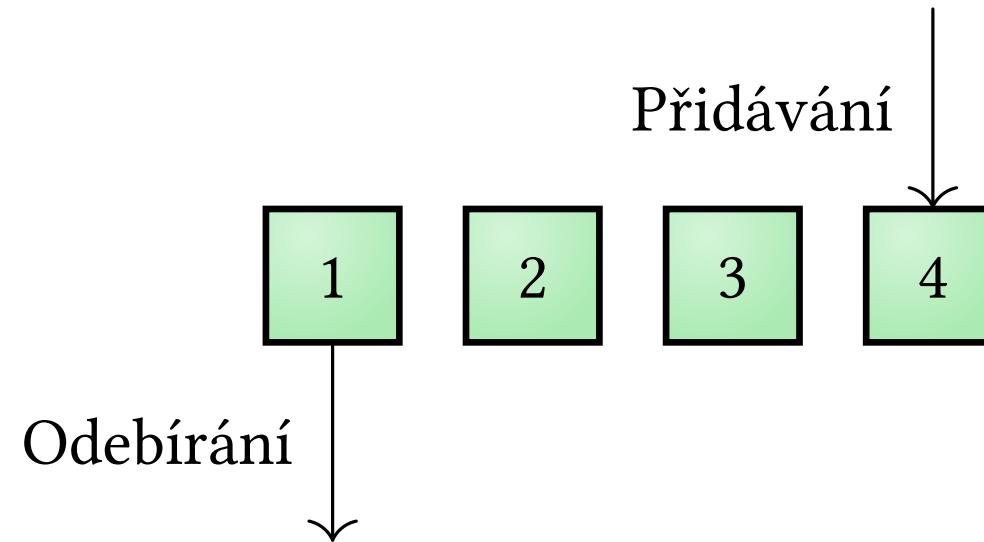
První přidaný prvek je první odebraný

Fronta

Představte si frontu v obchodě

- Noví zákazníci se přidávají na konec fronty
- Zákazníci jsou obsluženi v pořadí příchodu

Fronta



Fronta

Základní operace:

- `enqueue(item)` - přidá prvek na konec fronty
- `dequeue()` - odebere a vrátí prvek ze začátku fronty
- `peek()` - zobrazí první prvek (bez odebrání)
- `isEmpty()` - kontrola, zda je fronta prázdná

Implementace v TypeScriptu

```
class Queue<T> {  
    private items: T[] = [];  
  
    enqueue(item: T): void { this.items.push(item); }  
  
    dequeue(): T | undefined { return this.items.shift(); }  
  
    peek(): T | undefined { return this.items[0]; }  
  
    isEmpty(): boolean { return this.items.length === 0; }  
}
```

Typy front

Jednoduchá fronta - základní FIFO

Kruhová fronta (Circular Queue) - konec se napojuje na začátek

Prioritní fronta (Priority Queue) - prvky mají priority

Fronta s dvěma konci (Deque) - přidávání a odebírání z obou stran

Příklad - Správa úloh

```
// Task queue for async processing
type Task = () => Promise<void>;

class TaskQueue {
    private queue = new Queue<Task>();
    private processing = false;
```

Příklad - Správa úloh

```
class TaskQueue {  
    ...  
    async addTask(task: Task): Promise<void> {  
        this.queue.enqueue(task);  
    }  
}
```

Příklad - Správa úloh

```
private async processTasks(): Promise<void> {
    if (this.processing) {
        return;
    }
    this.processing = true;
    while (!this.queue.isEmpty()) {
        const task = this.queue.dequeue();
        if (task) await task();
    }
    this.processing = false;
}
```

Další příklady fronty

- **Správa úloh** v operačních systémech
- **Buffery** pro vstup/výstup (I/O)
- **Procházení do šířky** (BFS) v grafech
- **Message queue** v distribuovaných systémech
- **Print spooler** - fronta tiskových úloh

Spojový seznam (Linked List)

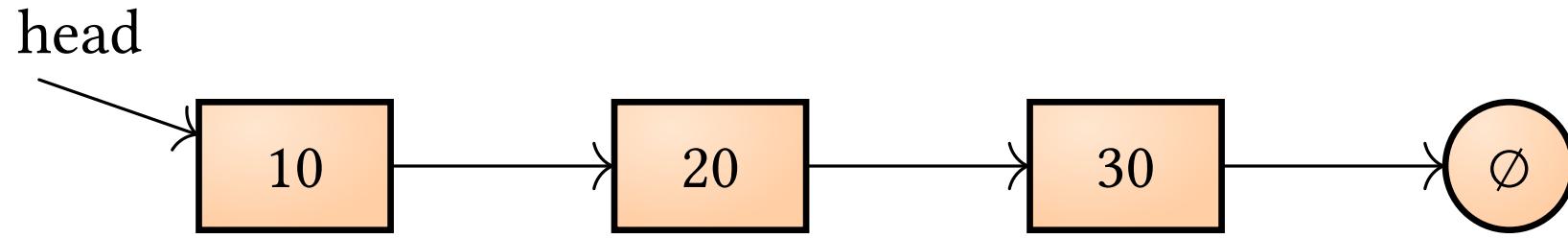
Spojový seznam

Spojový seznam je kolekce uzlů propojených odkazy

Každý uzel obsahuje:

- **Data** (hodnotu)
- **Odkaz** na další uzel (nebo null)

Vizualizace



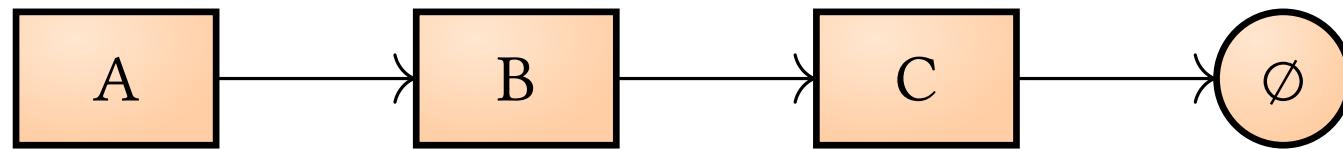
Rozdíly oproti poli

- Uzly **nejsou v paměti vedle sebe**
 - ▶ Časté výpadky cache paměti (cache miss)
- Přístup pouze **sekvenčně** přes odkazy
 - ▶ Nelze přejít na libovolný prvek přímo

Typy spojových seznamů

Jednosměrný (Singly Linked List)

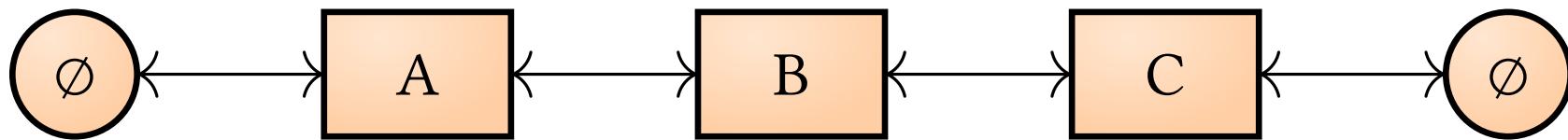
Každý uzel ukazuje pouze na **následující** uzel



Typy spojových seznamů

Obousměrný (Doubly Linked List)

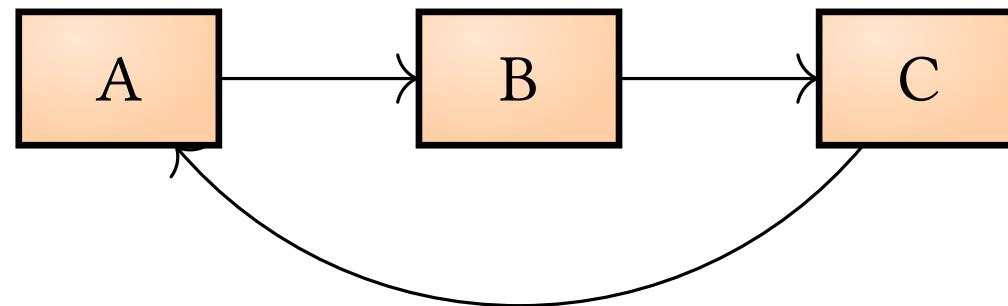
Každý uzel ukazuje na **předchozí i následující** uzel



Typy spojových seznamů

Kruhový (Circular Linked List)

Poslední uzel ukazuje zpět na **první** uzel



Implementace

```
class Node<T> {
    data: T;
    next: Node<T> | null = null;
    constructor(data: T) {
        this.data = data;
    }
}

class LinkedList<T> {
    private head: Node<T> | null = null;
    private size = 0;
```

Implementace

Vložení na začátek

```
class LinkedList<T> {  
    ...  
    prepend(data: T): void {  
        const newNode = new Node(data);  
        newNode.next = this.head;  
        this.head = newNode;  
        this.size++;  
    }  
}
```

Časová složitost: $\mathcal{O}(1)$

Implementace

Vložení na konec

```
class LinkedList<T> {  
    ...  
    append(data: T): void {  
        const newNode = new Node(data);  
        if (!this.head) {  
            this.head = newNode;  
        } else {
```

Implementace

```
let current = this.head;
while (current.next) {
    current = current.next; // Traverse entire list
}
current.next = newNode;
this.size++;
}
```

Časová složitost: $\mathcal{O}(n)$

Implementace

Odstranění prvku

```
class LinkedList<T> {  
    ...  
    remove(data: T): void {  
        if (!this.head) return;  
  
        if (this.head.data === data) { // Removing head  
            this.head = this.head.next;  
            this.size--;  
            return;  
    }
```

Implementace

```
let current = this.head;
while (current.next) { // Traverse entire list
    if (current.next.data === data) {
        current.next = current.next.next; this.size--;
        return;
    }
    current = current.next;
}
}
```

Časová složitost: $\mathcal{O}(n)$

Výhody spojového seznamu

- **Dynamická velikost** - roste a zmenšuje se podle potřeby
- **Efektivní vkládání/mazání** na začátku spojového seznamu - $\mathcal{O}(1)$
- **Žádná realokace** paměti při růstu
- Vhodný pro **častou modifikaci** dat

Nevýhody spojového seznamu

- **Pomalejší přístup** k prvkům - složitost $\mathcal{O}(n)$ vs složitost u pole $\mathcal{O}(1)$
- **Vyšší paměťová náročnost** - každý uzel ukládá odkaz
- **Nelze použít binární vyhledávání**
- Horší využití **cache** procesoru

Pole nebo spojový seznam?

Proč jsou pole v praxi často **rychlejší** než spojové seznamy, i když má horší teoretickou složitost pro vkládání?

Cache lokalita (cache locality)

Pole nebo spojový seznam?

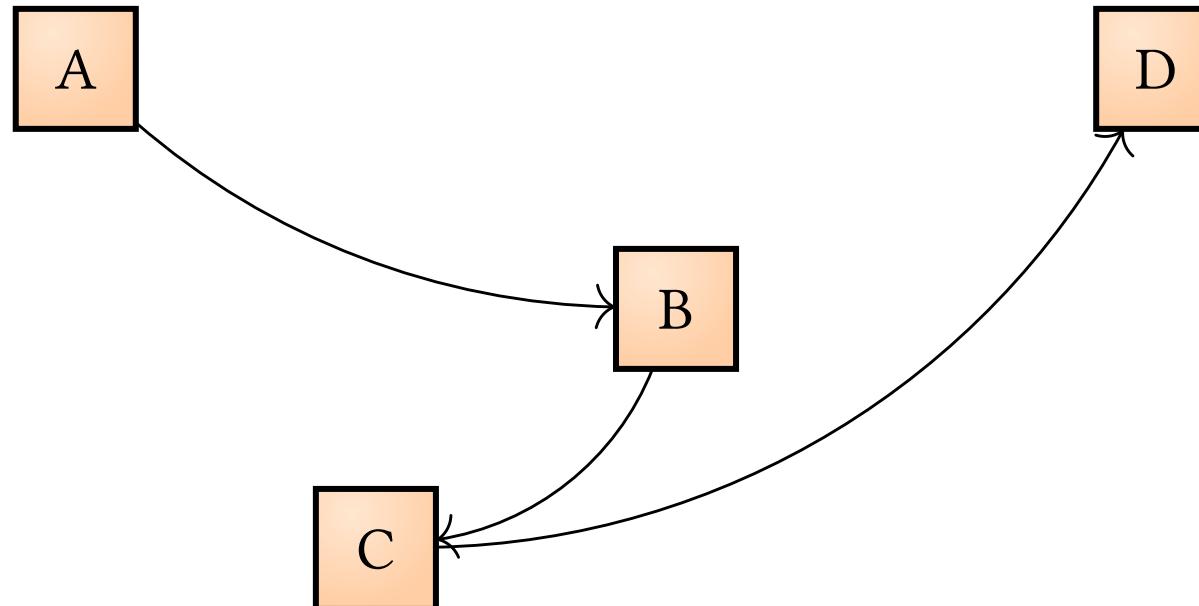
Pole - prvky jsou v paměti **za sebou**



- Procesor načte celý blok do cache
- Přístup k dalším prvkům je **velmi rychlý**
- **Sekvenční přístup** je optimální

Pole nebo spojový seznam?

Spojový seznam - prvky jsou v paměti **roztroušené**



Pole nebo spojový seznam?

- Každý uzel může být **kdekoli v paměti**
- **Cache miss** při každém skoku
- Pomalejší i pro sekvenční průchod

Pole nebo spojový seznam?

Srovnání časových složitostí

Operace	Pole	Spojový seznam
Přístup k i-tému prvku	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Vložení na začátek	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Vložení na konec	$\mathcal{O}^*(1)$	$\mathcal{O}(n)$
Vyhledávání	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Mazání prvku	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Pole nebo spojový seznam?

Poznámka 1: Vložení na konec pole má amortizovanou složitost $\mathcal{O}^*(1)$ (při dynamickém poli, které se realokuje na dvojnásobnou velikost)

Poznámka 2: Přidání na konec spojového seznamu může mít složitost $\mathcal{O}(1)$, pokud je implementován s dodatečným ukazatelem na poslední prvek

Praxe

- Pole jsou **rychlejší** pro většinu operací
- Cache efekt > teoretická složitost
- Moderní procesory mají silný **prefetch** pro sekvenční přístup
- Spojový seznam použijte jen když **opravdu** potřebujete
 - ▶ Časté vkládání/mazání na začátku
 - ▶ Nelze odhadnout velikost

Nicméně spojový seznam a jeho princip je **použit jako konstrukční blok složitějších struktur**, které mají i v praxi větší výhody než pole

Binární strom (Binary Tree)

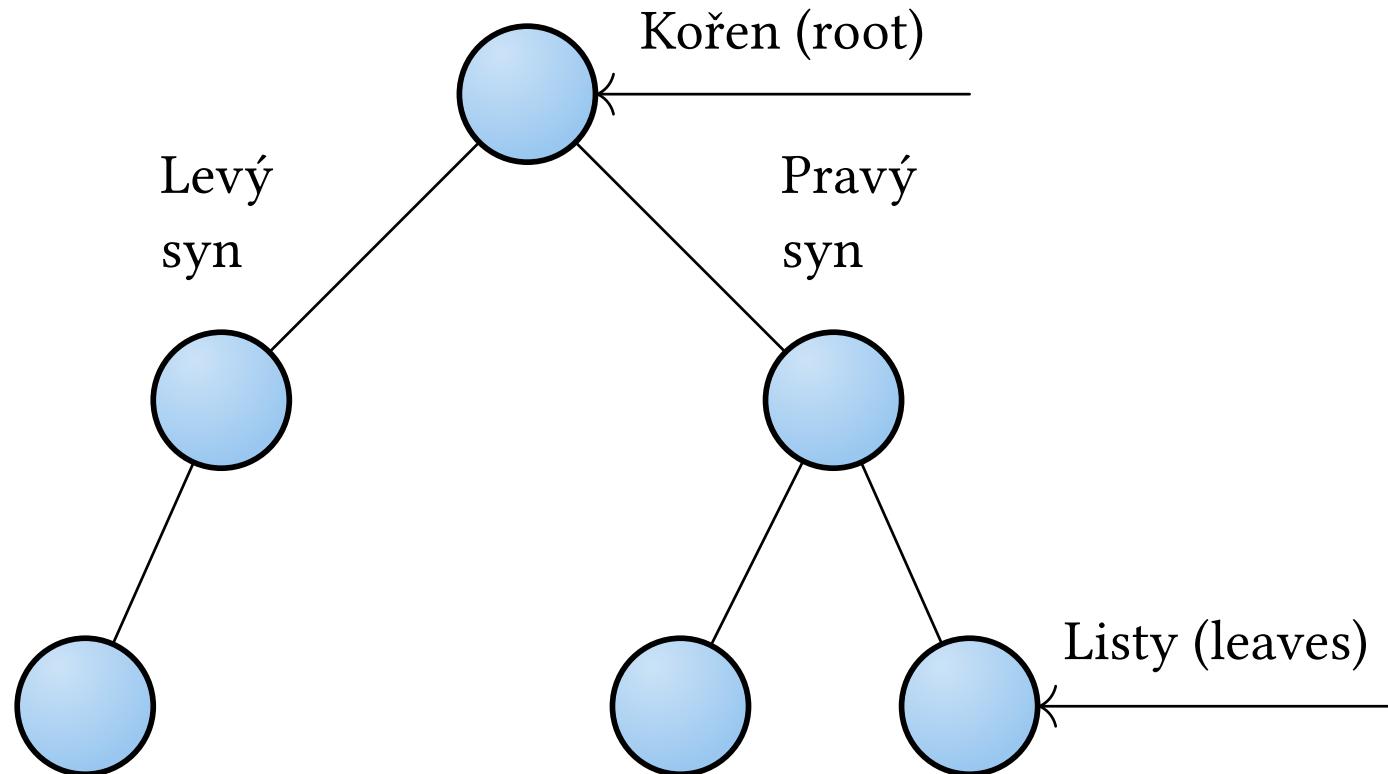
Binární strom

Binární strom je rekurzivní datová struktura s následujícími vlastnostmi:

- Každý uzel má **maximálně 2 potomky**
- Rozlišujeme **levého potomka** a **pravého potomka**
- Existuje právě jeden **kořen** stromu

Uzel bez potomků se nazývá **list**

Vizualizace



Implementace

```
class TreeNode<T> {  
    value: T;  
    left: TreeNode<T> | null = null;  
    right: TreeNode<T> | null = null;  
  
    constructor(value: T) {  
        this.value = value;  
    }  
}
```

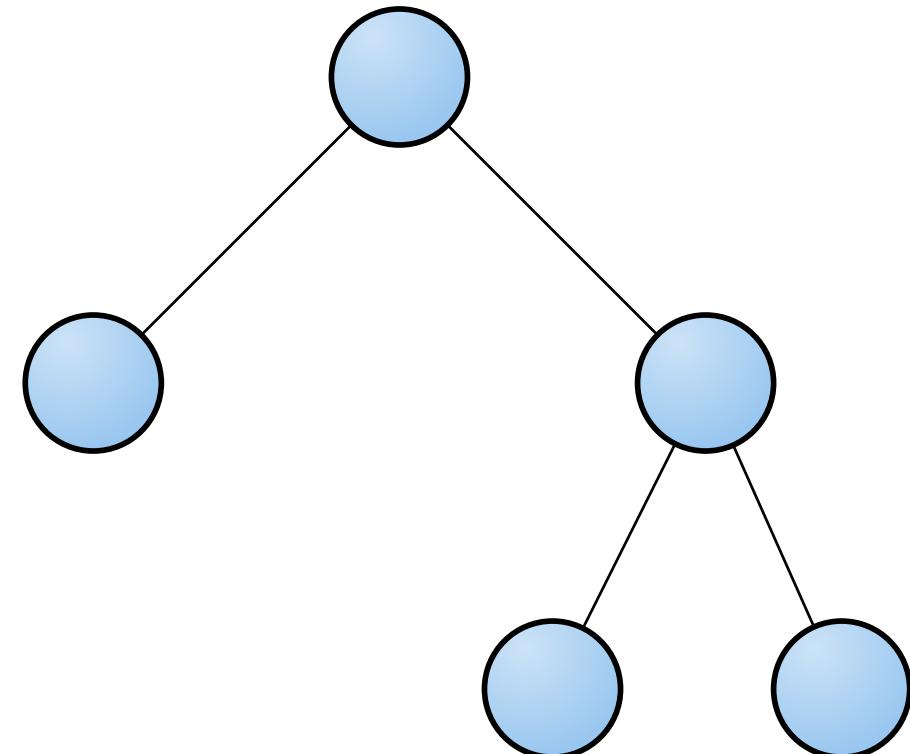
Implementace

```
class BinaryTree<T> {  
    protected root: TreeNode<T> | null = null;  
}
```

Typy binárních stromů

Plný binární strom (Full)

- Každý uzel má 0 nebo 2 děti

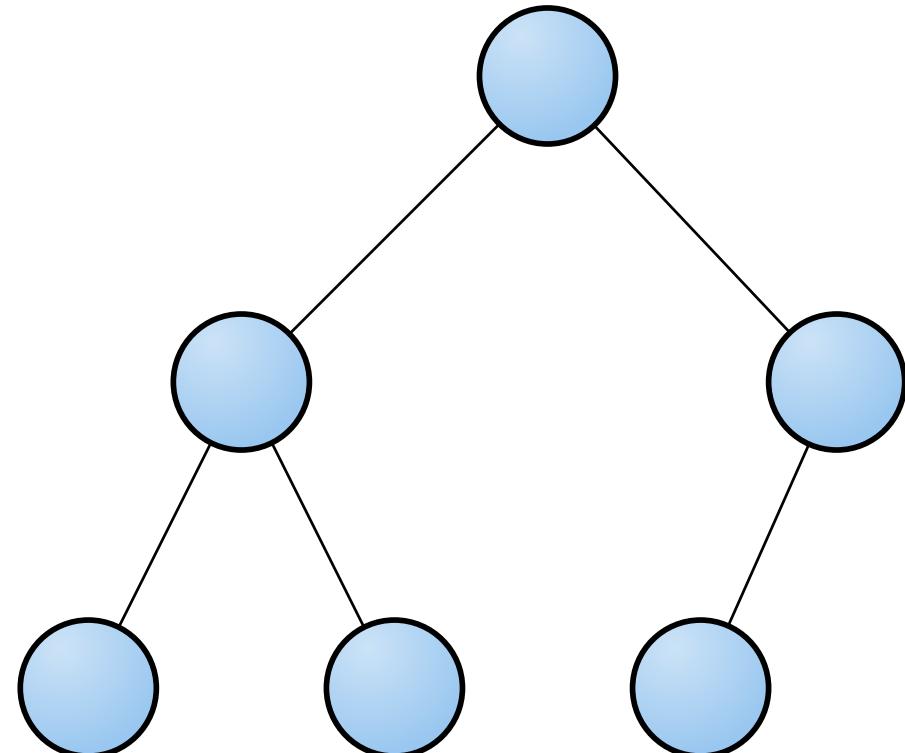


Typy binárních stromů

Kompletní binární strom

(Complete)

- Všechny úrovně plné kromě poslední
- Poslední úroveň zaplněná zleva doprava

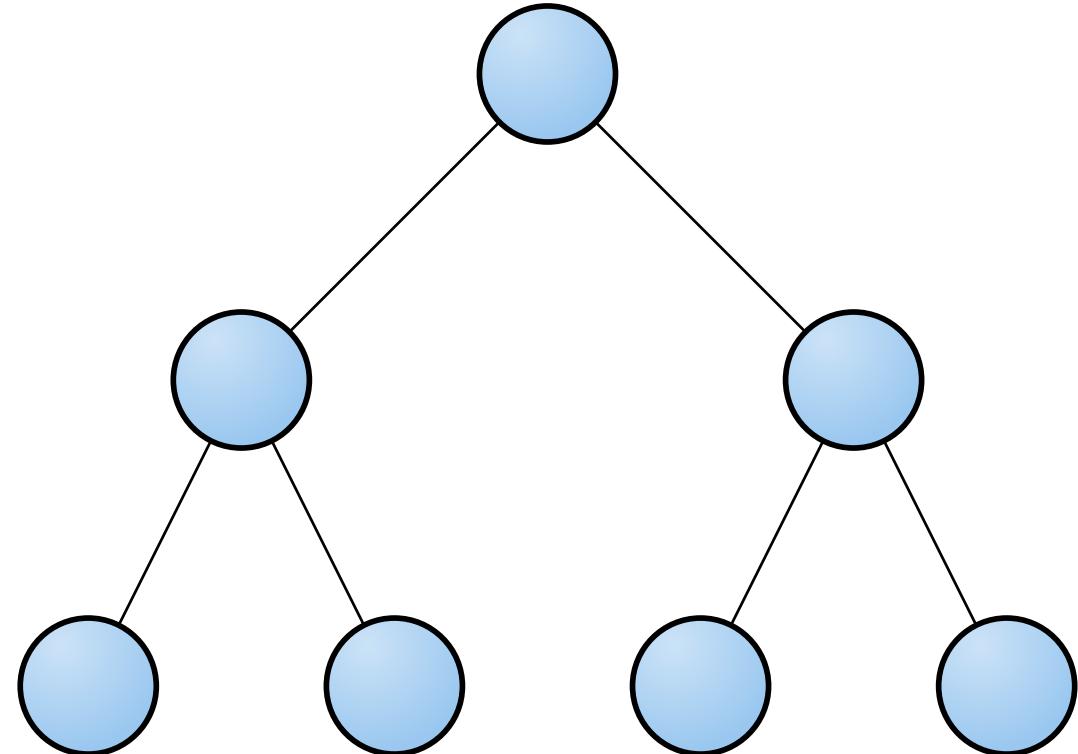


Typy binárních stromů

Perfektní binární strom

(Perfect)

- Všechny listy na stejné úrovni
- Počet uzlů: $2^h - 1$
(kde h = výška)



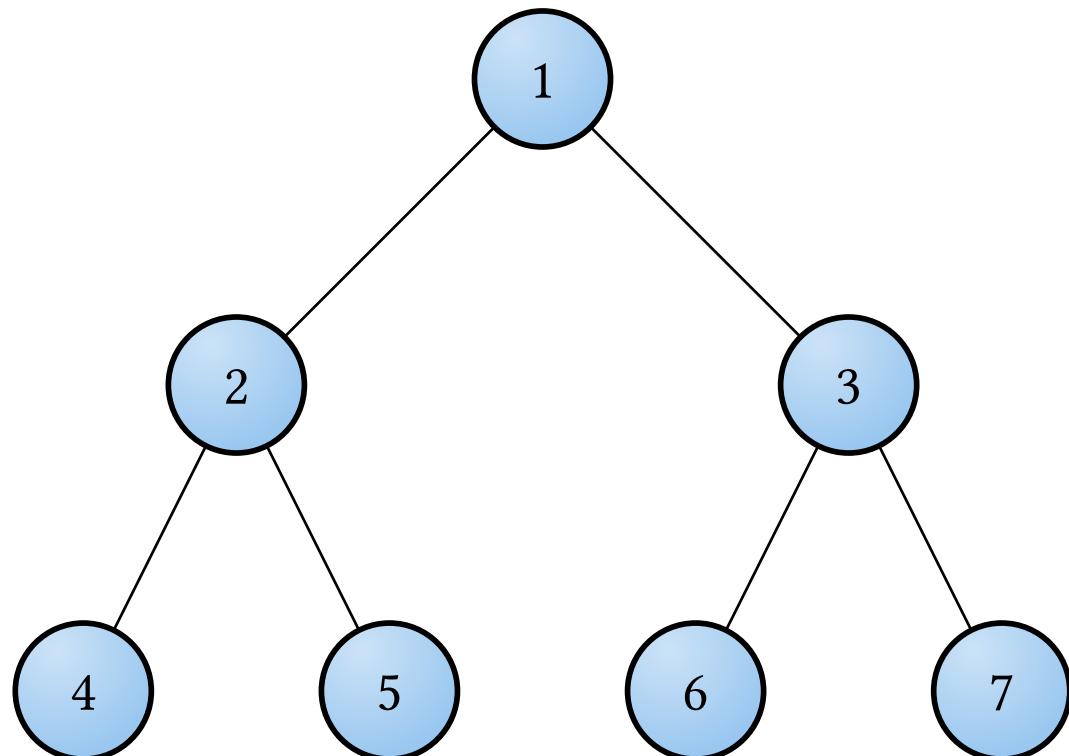
Procházení binárních stromů

Pre-order

Pre-order průchod = Aktuální, Levý, Pravý

```
preOrder(node: TreeNode<T> | null): void {  
    if (!node) return;  
  
    console.log(node.value);      // Process current node  
    this.preOrder(node.left);    // Process left subtree  
    this.preOrder(node.right);   // Process right subtree  
}
```

Pre-order



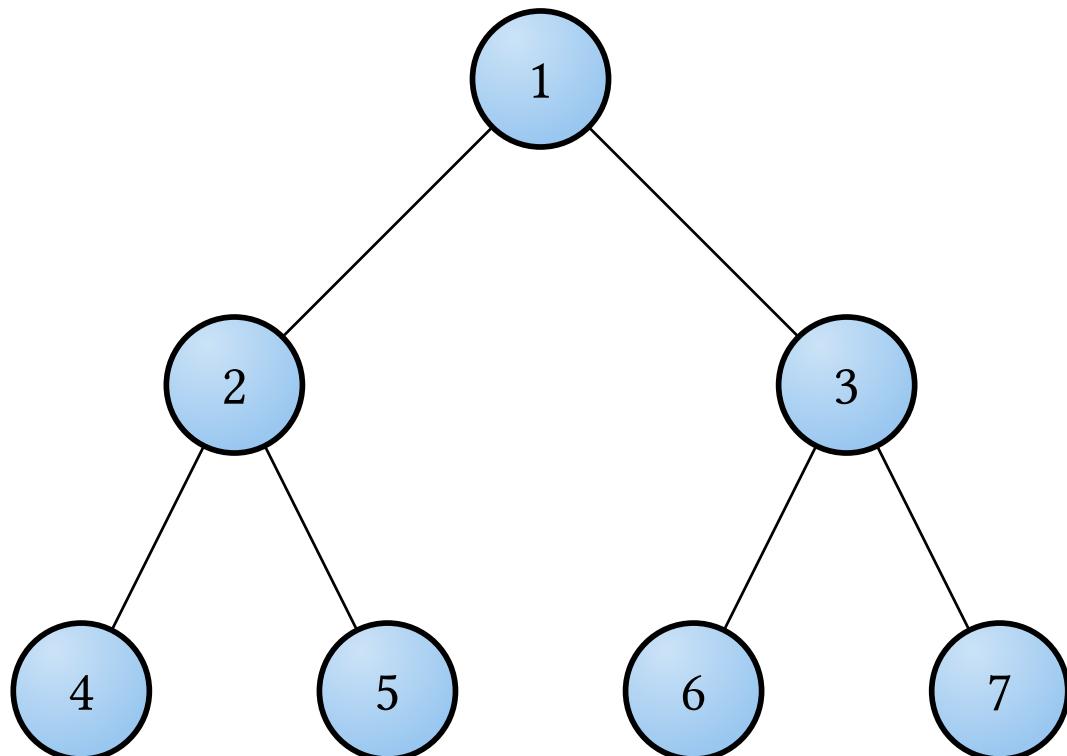
Výstup: 1, 2, 4, 5, 3, 6, 7

In-order

In-order průchod = Levý, Aktuální, Pravý

```
inOrder(node: TreeNode<T> | null): void {  
    if (!node) return;  
  
    this.inOrder(node.left);      // Left subtree  
    console.log(node.value);    // Process root  
    this.inOrder(node.right);    // Right subtree  
}
```

In-order



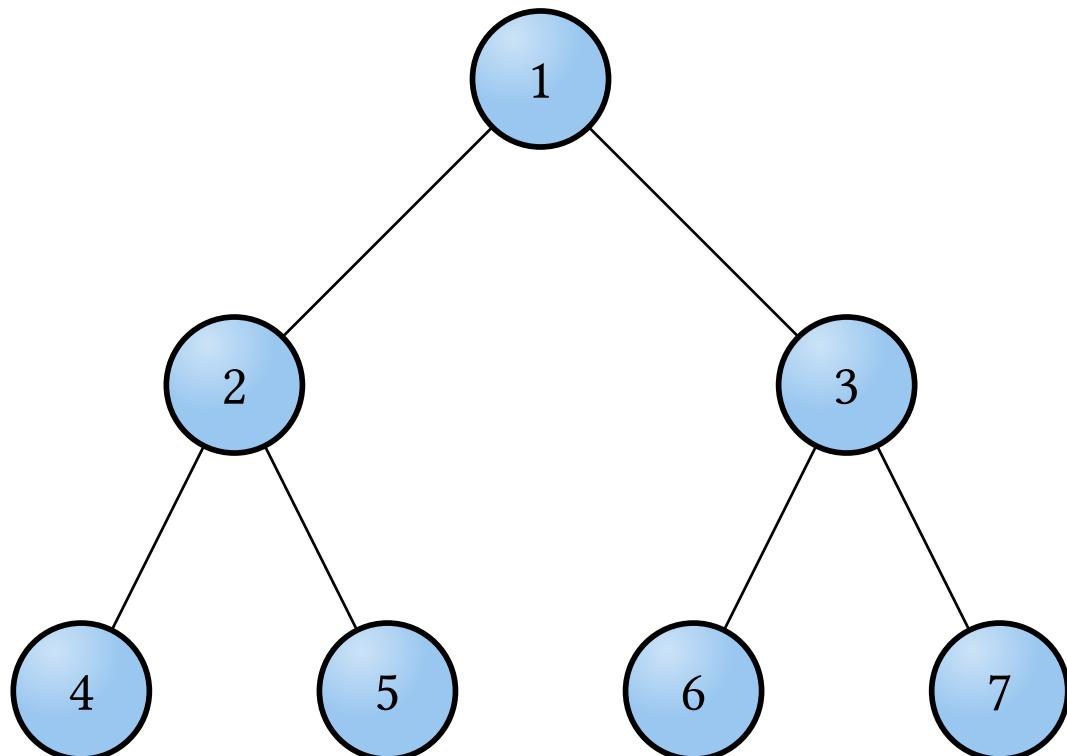
Výstup: 4, 2, 5, 1, 6, 3, 7

Post-order

Post-order průchod = **Levý, Pravý, Aktuální**

```
postOrder(node: TreeNode<T> | null): void {  
    if (!node) return;  
  
    this.postOrder(node.left);    // Left subtree  
    this.postOrder(node.right);  // Right subtree  
    console.log(node.value);    // Process root  
}
```

Post-order



Výstup: 4, 5, 2, 6, 7, 3, 1

Iterativní průchody stromu

Průchody stromu lze implementovat **bez rekurze** pomocí explicitního zásobníku

Výhody iterativního přístupu:

- Lepší kontrola nad pamětí
- Vyhneeme se stack overflow u hlubokých stromů
- Praktické použití zásobníku

Iterativní průchody stromu

Pre-order iterativně

```
preOrderIterative(node: TreeNode<T> | null): void {  
    if (!node) return;  
  
    const stack = new Stack<TreeNode<T>>();  
    stack.push(node);
```

Iterativní průchody stromu

```
while (!stack.isEmpty()) {  
    const current = stack.pop();  
    if (!current) continue;  
  
    console.log(current.value); // Process current  
  
    // Push right first, then left (LIFO)  
    if (current.right) stack.push(current.right);  
    if (current.left) stack.push(current.left);  
}  
}
```

Iterativní průchody stromu

In-order iterativně

```
inOrderIterative(node: TreeNode<T> | null): void {  
    const stack = new Stack<TreeNode<T>>();  
    let current = node;  
  
    while (current || !stack.isEmpty()) {  
        // Go to leftmost node  
        while (current) {  
            stack.push(current);  
            current = current.left;  
        }  
    }  
}
```

Iterativní průchody stromu

```
// Process node
current = stack.pop();
if (current) {
    console.log(current.value);
    current = current.right; // Move to right subtree
}
}
```

Iterativní průchody stromu

Post-order iterativně

```
postOrderIterative(node: TreeNode<T> | null): void {  
    if (!node) return;  
  
    const stack1 = new Stack<TreeNode<T>>();  
    const stack2 = new Stack<TreeNode<T>>();  
  
    stack1.push(node);
```

Iterativní průchody stromu

```
while (!stack1.isEmpty()) {  
    const current = stack1.pop();  
    if (!current) continue;  
  
    stack2.push(current);  
  
    if (current.left) stack1.push(current.left);  
    if (current.right) stack1.push(current.right);  
}
```

Iterativní průchody stromu

```
// Print in reverse order
while (!stack2.isEmpty()) {
    const current = stack2.pop();
    if (current) console.log(current.value);
}
}
```

Binární vyhledávací strom (BST)

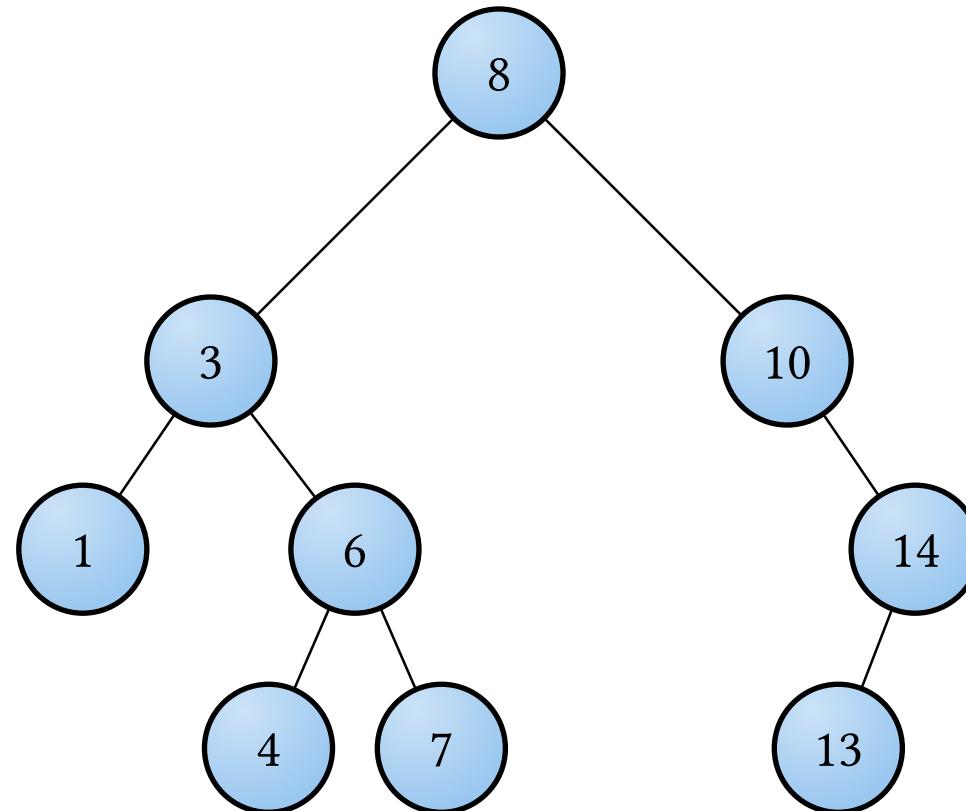
Binární vyhledávací strom

BST (Binary Search Tree) je binární strom s vlastnostmi:

Pro každý uzel platí:

- Všechny hodnoty v **levém podstromu** jsou **menší**
- Všechny hodnoty v **pravém podstromu** jsou **větší**

Vizualizace BST



Vyhledávání v BST

```
search(  
    value: T,  
    node: TreeNode<T> | null = this.root  
) : TreeNode<T> | null {  
    // Node not found  
    if (!node) return null;  
  
    // Found it!  
    if (value === node.value) return node;
```

Vyhledávání v BST

```
// Search in left subtree
if (value < node.value) {
    return this.search(value, node.left);
}

// Search in right subtree
return this.search(value, node.right);
}
```

Vložení do BST

```
insert(value: T): void {
    const newNode = new TreeNode(value);

    if (!this.root) {
        this.root = newNode;
        return;
    }

    this.insertNode(this.root, newNode);
}
```

Vložení do BST

```
private insertNode(  
    node: TreeNode<T>, newNode: TreeNode<T>  
): void {  
    if (newNode.value < node.value) {  
        if (!node.left) node.left = newNode;  
        else this.insertNode(node.left, newNode);  
    } else {  
        if (!node.right) node.right = newNode;  
        else this.insertNode(node.right, newNode);  
    }  
}
```

Odstanění z BST

Tři případy:

1. **Uzel je list** - jednoduché odstranění
2. **Uzel má jednoho potomka** - nahrad' uzlem tímto potomkem
3. **Uzel má dva potomky** - najdi následníka (nejmenší v pravém podstromu) a nahrad' aktuální uzel tímto následníkem

Odstanění z BST

```
remove(value: T): void {
    this.root = this.removeNode(this.root, value);
}
```

Odstanění z BST

```
private removeNode(  
    node: TreeNode<T> | null, value: T  
): TreeNode<T> | null {  
    if (!node) return null;  
  
    if (value < node.value) { // Search in left subtree  
        node.left = this.removeNode(node.left, value);  
    } else if (value > node.value) { // Search in right subtree  
        node.right = this.removeNode(node.right, value);  
    } else {  
        // Found the node to remove
```

Odstranění z BST

```
// Case 1: Leaf node
if (!node.left && !node.right) return null;

// Case 2: One child
if (!node.left) return node.right;
if (!node.right) return node.left;
```

Odstranění z BST

```
// Case 3: Two children
const minRight = this.findMin(node.right);
node.value = minRight.value;
node.right = this.removeNode(node.right, minRight.value);
}

return node;
}
```

Odstanění z BST

```
private findMin(  
    node: TreeNode<T>  
) : TreeNode<T> {  
    if (!node.left) return node;  
    return this.findMin(node.left);  
}
```

Rozsah složitostí

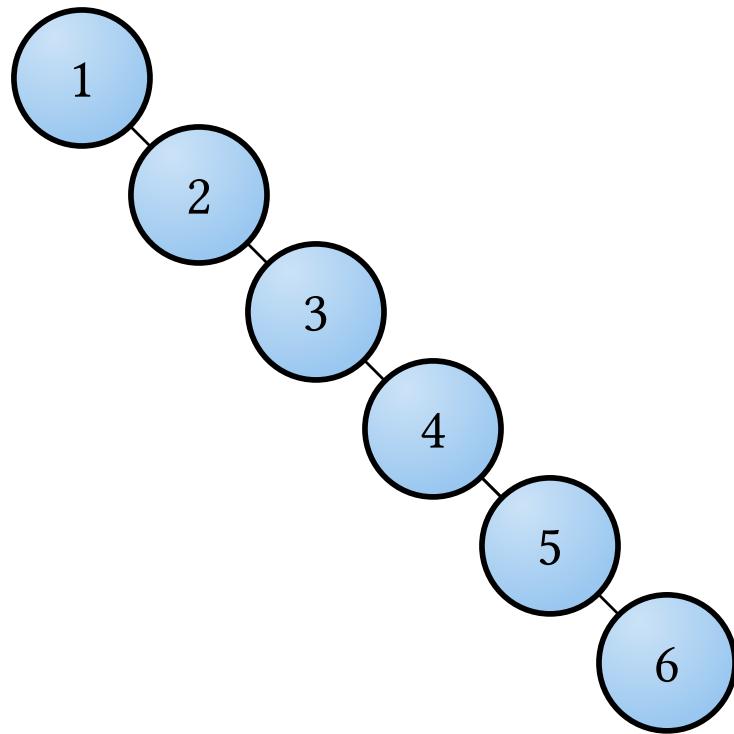
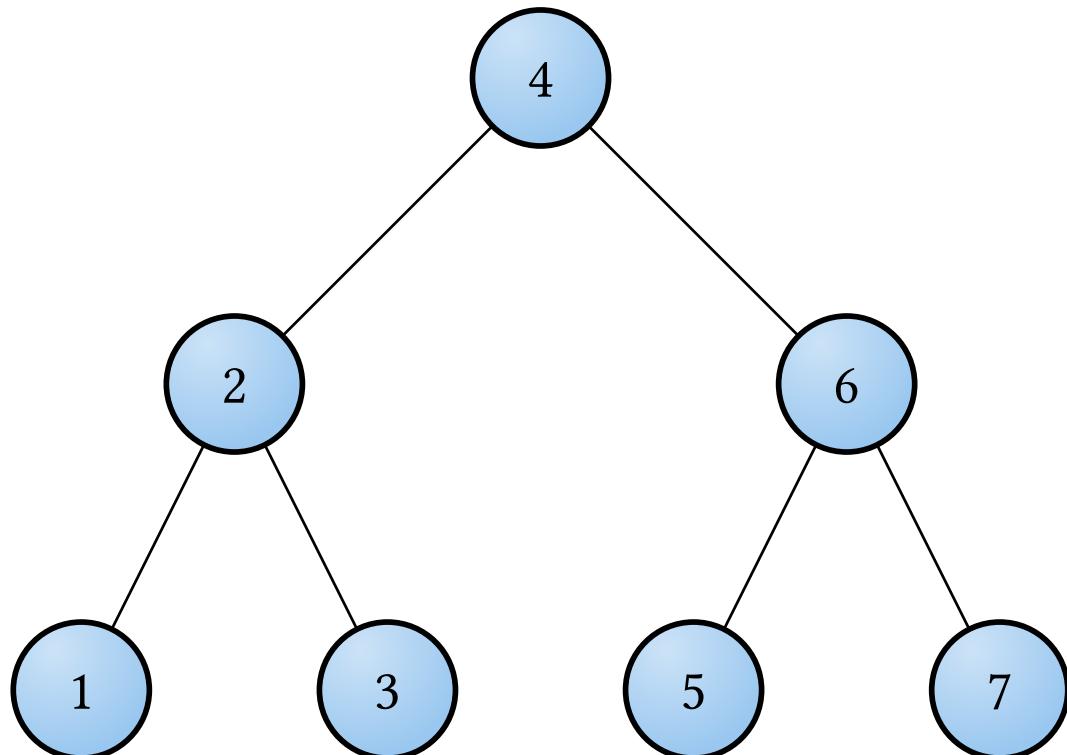
Pokud máme algoritmus s lineární složitostí $\mathcal{O}(n)$, tak to znamená, že počet operací roste lineárně s velikostí vstupu/velikostí dat

- $\mathcal{O}(n)$ - pro milion prvků očekáváme přibližně miliony operací ($a * n$, kde a je konstanta)
- $\mathcal{O}(\log n)$ - pro milion prvků očekáváme přibližně 20 operací ($a * \log(n)$, kde a je konstanta)
 - ▶ Zápisem $\log n$ se myslí logaritmus o základu 2

Výhody binárního vyhledávacího stromu

- Rychlé vyhledávání - složitost $\mathcal{O}^*(\log n)$ v průměru
- Rychlé vkládání a mazání - složitost $\mathcal{O}^*(\log n)$ v průměru
- Udržuje pořadí - in-order dává seřazené prvky
- Dynamická struktura - roste podle potřeby

Problém vyváženosti



Problém vyváženosti

Nevyvážený binární strom může **degenerovat** na spojový seznam

V tomto případě budou operace vyhledávání, vkládání a mazání mít složitost $\mathcal{O}(n)$

Pokud bychom dokázali strom udržovat vyvážený, tak by operace s ním měli stálou složitost $\mathcal{O}(\log n)$

Balancování binárních vyhledávacích stromů

Výška stromu

Výška stromu je maximální počet hran na nejdelší cestě z kořene do listu

Výška podstromu je maximální počet hran v podstromu začínajícím v daném uzlu do listu

Výšku podstromu u označíme $h(u)$

Výška stromu

Balanční faktor (δ) uzlu u :

$$\delta(u) = h(u.\text{right}) - h(u.\text{left})$$

Hodnoty balančního faktoru mohou být

- 0, pokud je uzel vyvážený
- 1, pokud je uzel „lehce převážený“ doprava
- -1 , pokud je uzel „lehce převážený“ doleva
- 2, pokud je uzel „moc převážený“ doprava
- -2 , pokud je uzel „moc převážený“ doleva

AVL strom

AVL strom - první samobalancující BST (1962)

Pojmenován po autorech: **Adelson-Velsky** a **Landis**

Pravidlo:

Pro každý uzel u platí: $|\delta(u)| \leq 1$

Po každé operaci se strom **automaticky vyváží** (opraví) pomocí takzvaných **rotací**

Pravá rotace

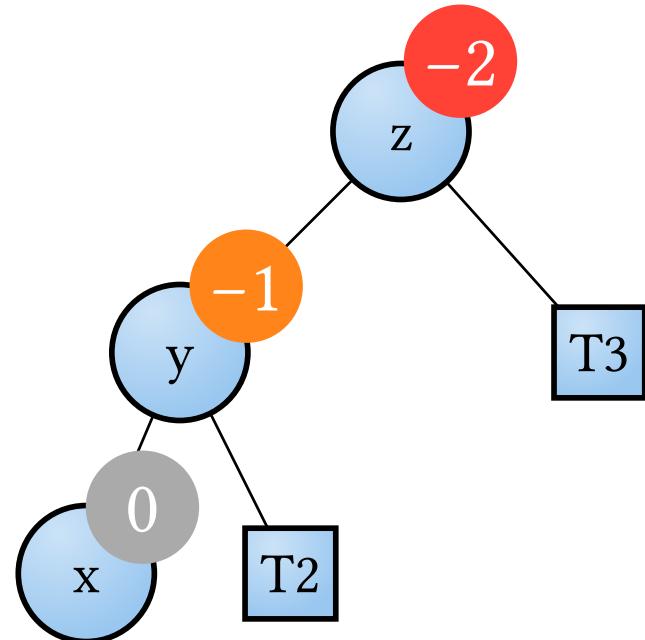
Každá rotace má své podmínky, dle kterých se pozná, jaká se má provést

Pravá rotace se provádí na uzlu u , pokud platí

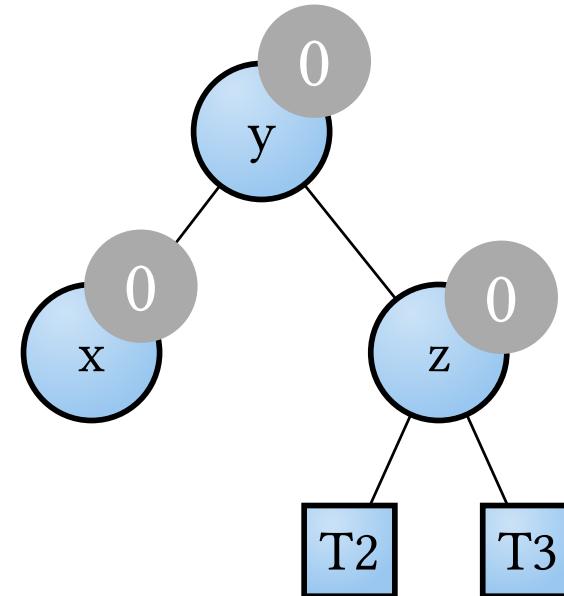
$$\delta(u) = -2 \wedge \delta(u.\text{left}) = -1$$

Pravá rotace

Před rotací



Po rotaci



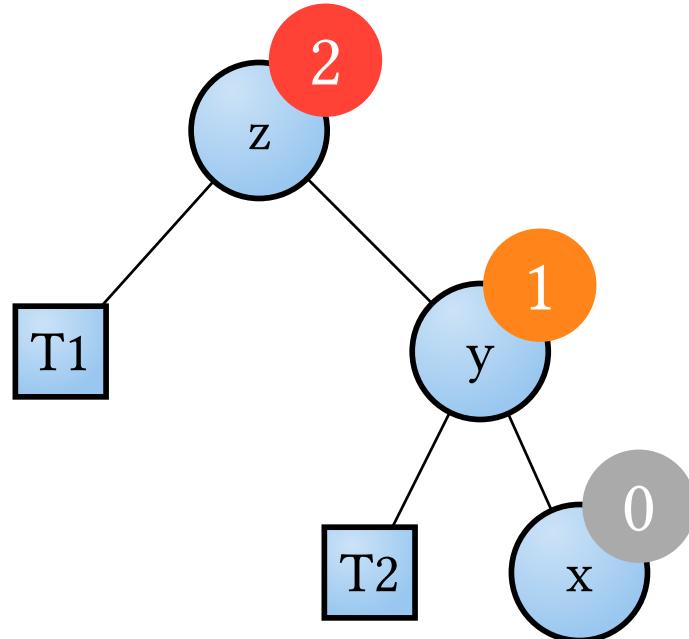
Levá rotace

Levá rotace se provádí na uzlu u , pokud platí

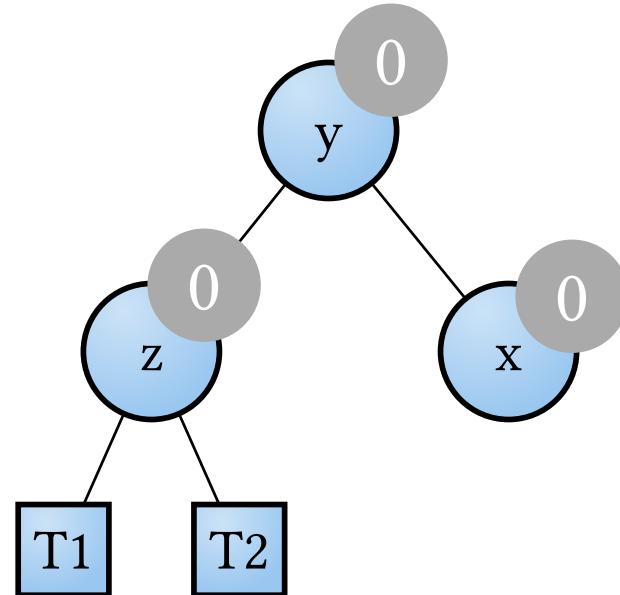
$$\delta(u) = 2 \wedge \delta(u.\text{right}) = 1$$

Levá rotace

Před rotací



Po rotaci



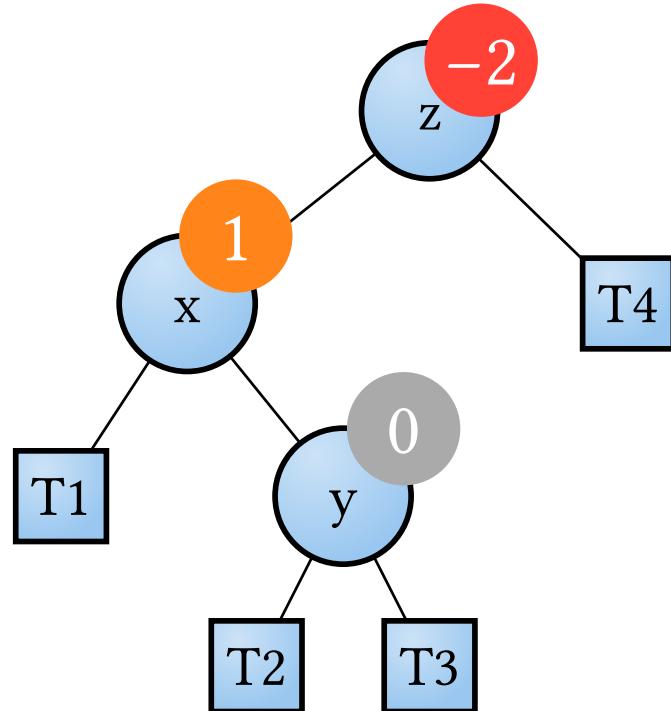
Levo-pravá rotace

Levo-pravá rotace se provádí na uzlu u , pokud platí

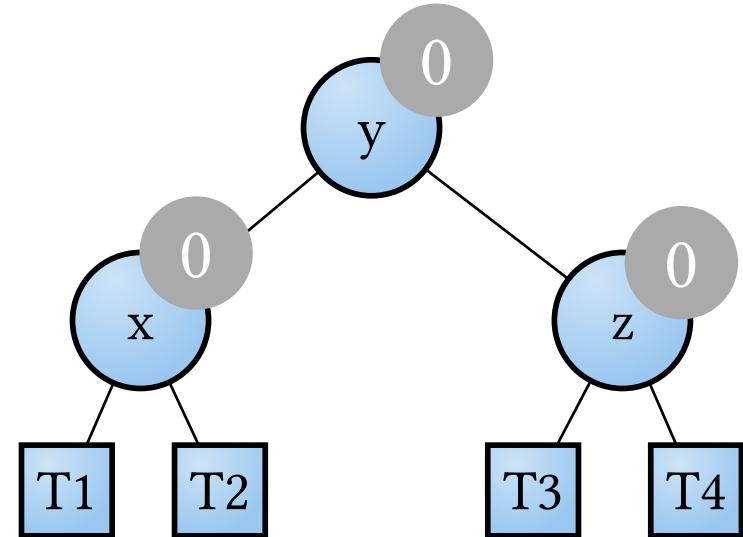
$$\delta(u) = -2 \wedge \delta(\text{u.left}) = 1$$

Levo-pravá rotace

Před rotací



Po rotaci



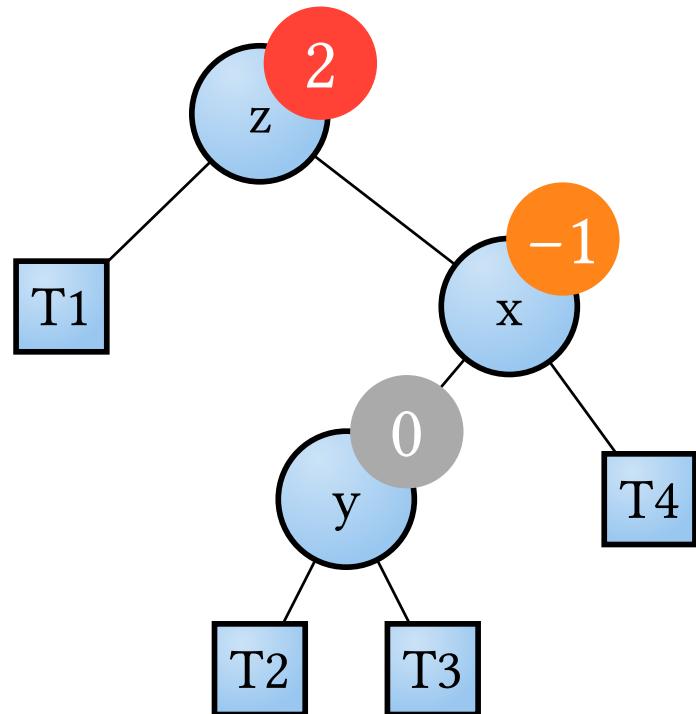
Pravo-levá rotace

Pravo-levá rotace se provádí na uzlu u , pokud platí

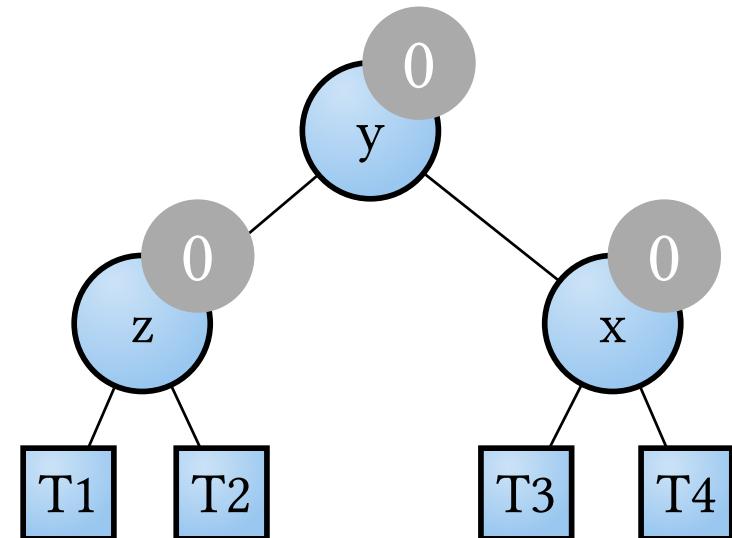
$$\delta(u) = 2 \wedge \delta(u.\text{right}) = -1$$

Pravo-levá rotace

Před rotací



Po rotaci



Časová složitost rotací

Časová složitost rotací je $\mathcal{O}(1)$

Kolik rotací je nutné provést při vkládání/mazání?

- Při vkládání: $\mathcal{O}(\log n)$
- Při mazání: $\mathcal{O}(\log n)$

Proč?

Protože při vkládání/mazání se může stát, že se strom dokáže korektně vyvážit až na kořeni

Časová složitost rotací

Jelikož je hloubka stromu $\mathcal{O}(\log n)$, tak nás oprava balančního faktoru stojí $\mathcal{O}(\log n * 1) = \mathcal{O}(\log n)$

Celková složitost vložení/mazání + oprava balančního faktoru je tedy

$$\mathcal{O}(\log n + \log n) = \mathcal{O}(\log n)$$

Složitost vyhledávání zůstává stejná, tedy $\mathcal{O}(\log n)$

Další samovyvažovací datové struktury

Červeno-černé stromy (Red-Black Trees)

- Každý uzel má barvu (červená/černá)
- Používá se v C++ `std::map`, Java `TreeMap`
- Méně rotací než AVL

B-stromy

- Uzly mohou mít více než 2 děti
- Používají se v databázích a souborových systémech

Splay stromy

- Nedávno použité prvky se přesouvají ke kořeni

Grafové struktury

Graf (Graph)

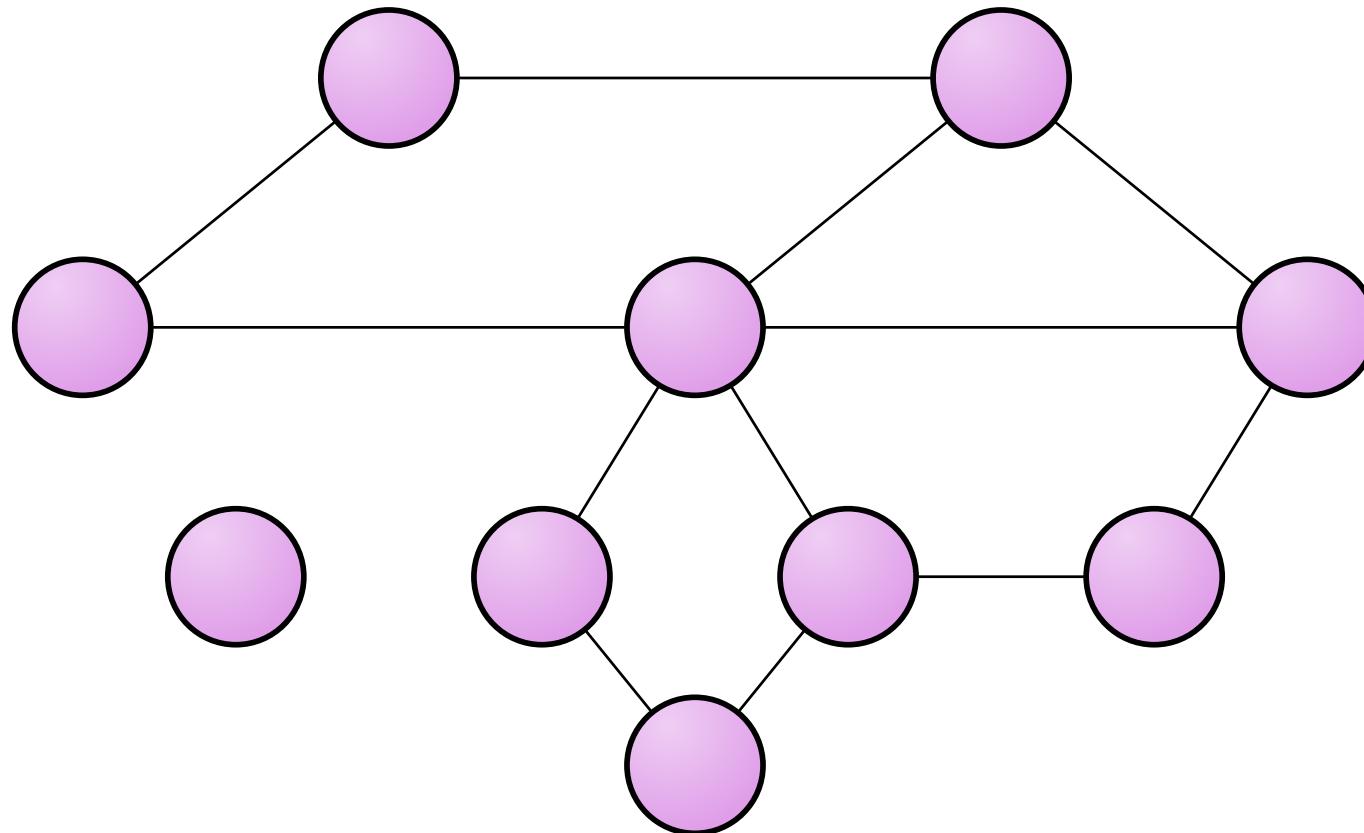
Graf je datová struktura reprezentující **vztahy mezi objekty**

Skládá se z:

- **Vrcholů** (vertices/nodes) - objekty
- **Hran** (edges) - spojení mezi objekty

Jedná se tedy o **zobecnění** datových struktur, které jsme již probírali
(spojový seznam, stromy)

Vizualizace



Vizualizace

Graf je dvojice $G = (V, E)$, kde V je množina vrcholů a E je množina hran

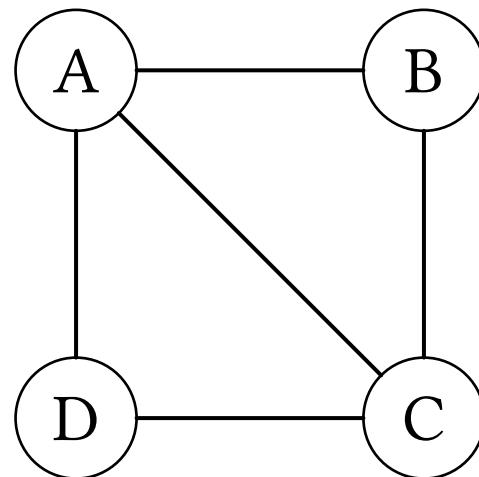
Počet vrcholů označujeme jako $|V|$ a počet hran jako $|E|$

Příklady grafů v reálném světě

- **Sociální sítě** - lidé (vrcholy), přátelství (hrany)
- **Mapy** - města (vrcholy), silnice (hrany)
- **Internet** - servery (vrcholy), spojení (hrany)
- **Elektrické obvody** - komponenty (vrcholy), dráty (hrany)

Neorientovaný graf

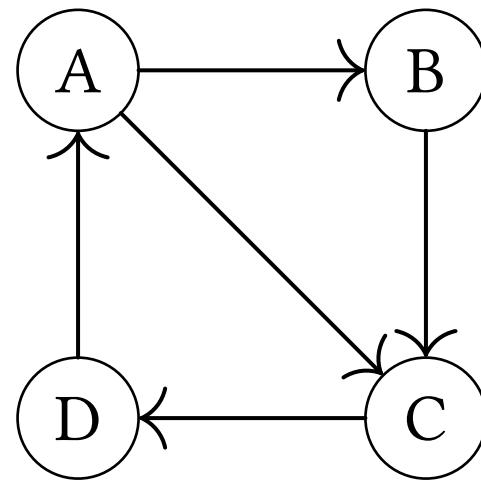
Neorientovaný graf (Undirected) - hrany nemají směr



$$E = \{\{A, B\}, \{B, C\}, \{C, D\}, \{D, A\}, \{A, C\}\}$$

Orientovaný graf

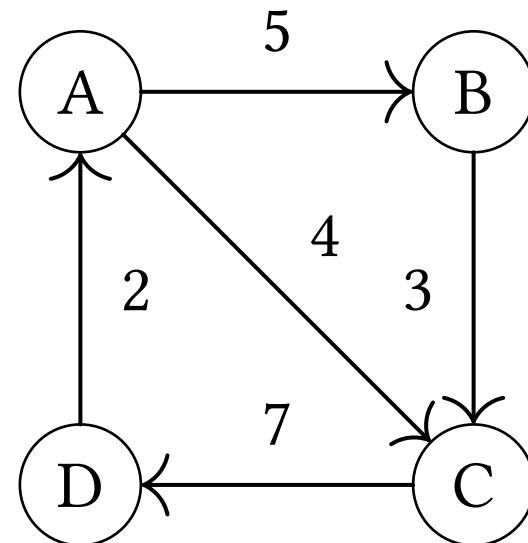
Orientovaný graf (Directed) - hrany **mají směr**



$$E = \{(A, B), (B, C), (C, D), (D, A), (A, C)\}$$

Orientovaný graf

Vážený graf (Weighted) - hrany mají váhu (vzdáenosť, cena, ...)



Reprezentace grafů

Matice sousednosti

2D pole reprezentující hrany mezi vrcholy

```
// Graph: A-B, A-C, B-C
const graph: number[][] = [
    //   A   B   C
    [0, 1, 1], // A
    [1, 0, 1], // B
    [1, 1, 0]  // C
];
```

Matice sousednosti - složitost operací

Prostorová složitost: $\mathcal{O}(|V|^2)$

Časová složitost kontroly existence hrany: $\mathcal{O}(1)$

Časová složitost přidání hrany: $\mathcal{O}(1)$

Časová složitost odebrání hrany: $\mathcal{O}(1)$

Časová složitost přidání vrcholu: $\mathcal{O}(|V|^2)$

- Přidání vrcholu vyžaduje realokaci každého řádku v 2D poli

Matice sousednosti - složitost operací

Vhodné pro husté grafy (mnoho hran)

Seznam sousedů

Pro každý vrchol ukládáme (spojový) seznam jeho sousedů

- V praxi je typické použití i dynamické pole

```
type Graph = Record<string, string[]>;
```

```
const graph: Graph = {  
    'A': ['B', 'C'],  
    'B': ['A', 'C', 'D'],  
    'C': ['A', 'B'],  
    'D': ['B']  
};
```

Seznam sousedů - složitost operací

Pomocí $\Delta(v)$ označujeme stupeň vrcholu v (počet sousedů)

Pomocí $\Delta(G)$ označujeme maximální stupeň vrcholu v grafu G

Seznam sousedů - složitost operací

Prostorová složitost: $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V|^2)$

- Alternativní odhad je $\mathcal{O}(|V| * \Delta(G))$

Časová složitost kontroly existence hrany: $\mathcal{O}(\Delta(G))$

Časová složitost přidání hrany: $\mathcal{O}(1)$

Časová složitost odebrání hrany: $\mathcal{O}(\Delta(G))$

Časová složitost přidání vrcholu: $\mathcal{O}(|V|)$

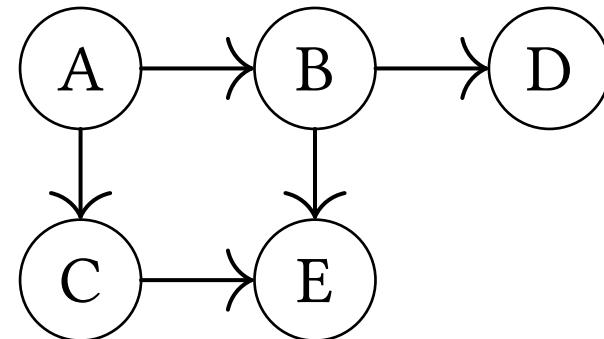
- Přidání vrcholu vyžaduje realokaci hlavního seznamu sousedů

Vhodnost použití

Vhodné pro řídké grafy (málo hran)

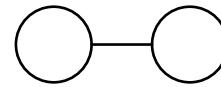
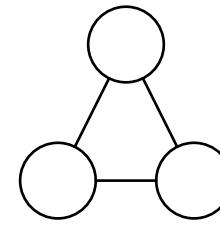
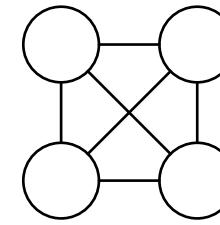
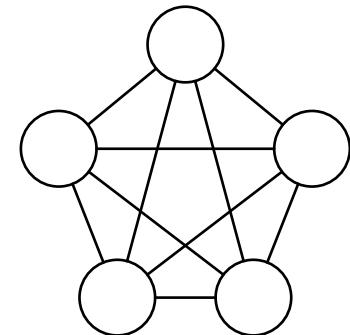
Speciální typy grafů

Acyklický graf (DAG - Directed Acyclic Graph)



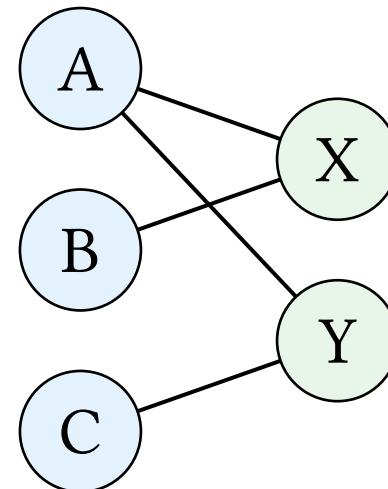
Úplný graf

Úplný graf je takový graf, ve kterém je každý vrchol spojen se všemi ostatními a značíme ho K_n

 K_1  K_2  K_3  K_4  K_5

Bipartitní graf

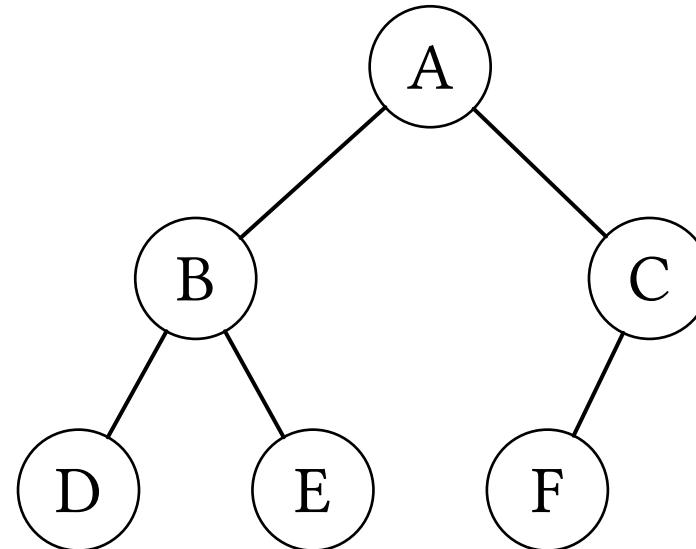
Bipartitní graf je takový graf, ve kterém jsou vrcholy rozděleny do dvou množin



Strom

Strom je souvislý acyklický graf

- Z každého vrcholu existuje cesta do jakéhokoli jiného vrcholu
- Neobsahuje cykly



Procházení grafů

Procházení grafů

Dva hlavní způsoby procházení grafů:

1. **BFS (Breadth-First Search)** - do šířky
2. **DFS (Depth-First Search)** - do hloubky

BFS (Breadth-First Search)

BFS - Prohledávání do šířky

Procházení po úrovních (vrstvách)

Využívá **frontu** (FIFO)

Aplikace:

- Hledání nejkratší cesty v nevážených grafech
- Hledání komponent souvislosti
- Web scraping
- Procházení stavového prostoru

Příklad BFS

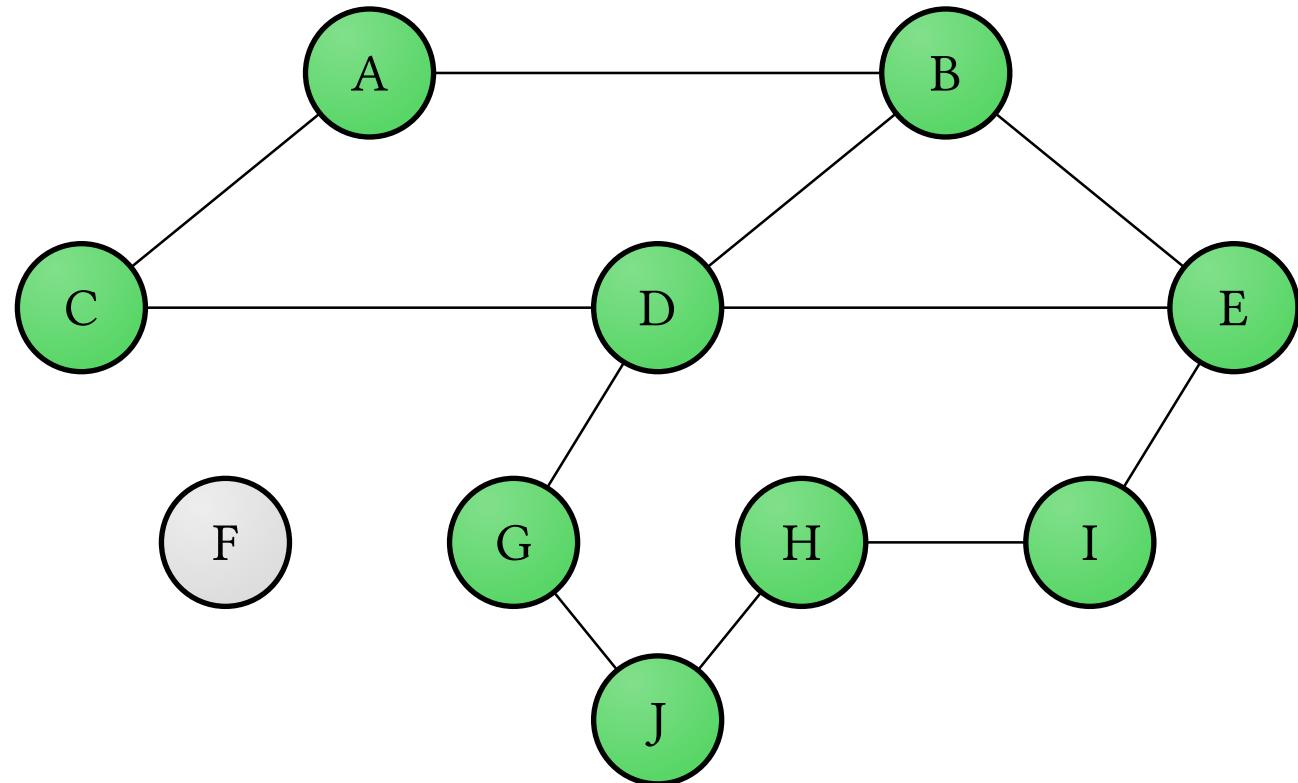
Aktuální: -

Fronta:

[]

BFS dokončeno!

Vrchol F není dosažitelný
z vrcholu G.



Implementace BFS

```
function bfs(graph: Graph, start: string): void {  
    const visited = new Set<string>();  
    const queue = new Queue<string>();  
  
    queue.enqueue(start);  
    visited.add(start);
```

Implementace BFS

```
while (!queue.isEmpty()) {  
    const vertex = queue.dequeue()!;  
    console.log(vertex);  
    for (const neighbor of graph[vertex]) {  
        if (!visited.has(neighbor)) {  
            visited.add(neighbor);  
            queue.enqueue(neighbor);  
        }  
    }  
}
```

DFS (Depth-First Search)

DFS - Prohledávání do hloubky

Procházení co nejdále do hloubky, pak zpět

Využívá **zásobník** (LIFO) nebo rekurzi

Aplikace:

- Detekce cyklů
- Topologické třídění
- Hledání silně souvislých komponent
- Řešení bludišť a puzzle

Příklad DFS

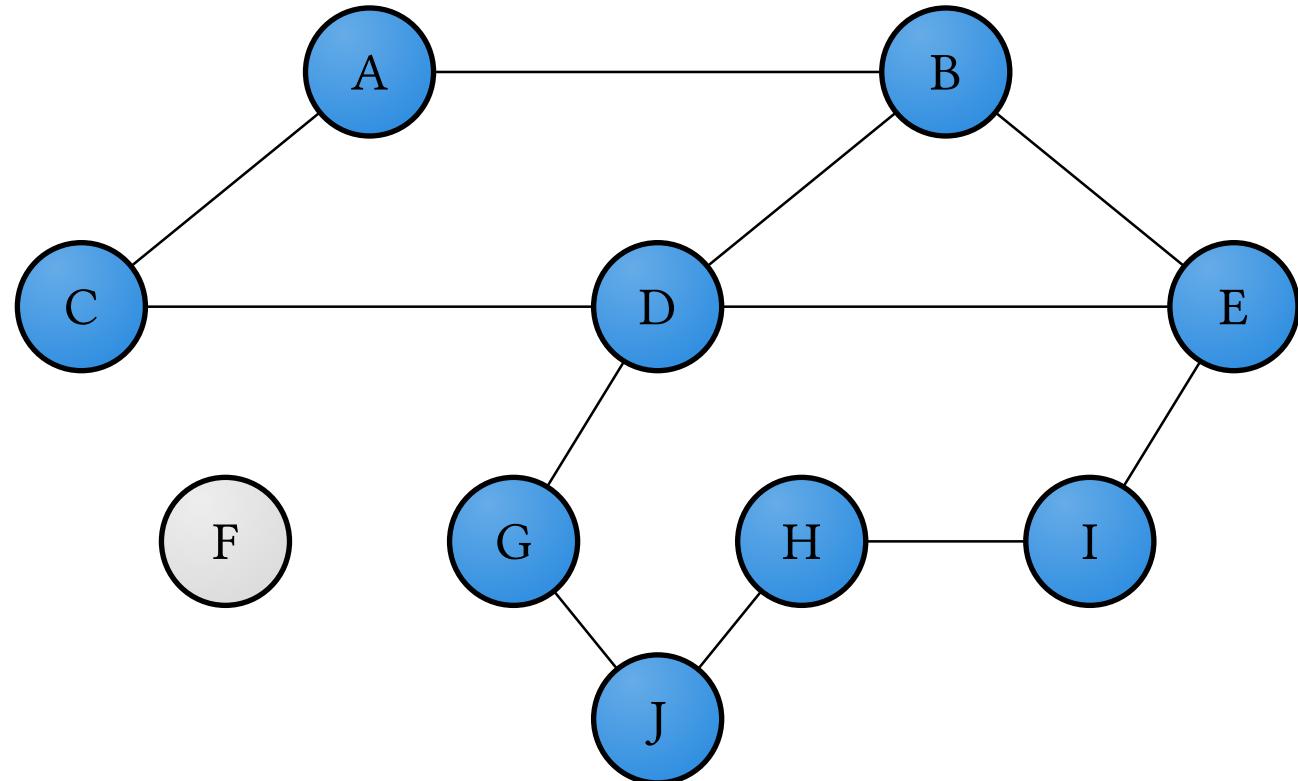
Aktuální: -

Zásobník:

[]

DFS dokončeno!

Vrchol F není
dosažitelný z G.



Implementace DFS (rekurzivně)

```
function dfs(  
    graph: Graph, vertex: string,  
    visited: Set<string> = new Set()  
) : void {  
    console.log(vertex); visited.add(vertex);  
    for (const neighbor of graph[vertex]) {  
        if (!visited.has(neighbor)) {  
            dfs(graph, neighbor, visited);  
        }  
    }  
}
```

Implementace DFS (iterativně)

```
function dfsIter(graph: Graph, start: string): void {  
    const visited = new Set<string>();  
    const stack = new Stack<string>();  
  
    stack.push(start);  
  
    while (!stack.isEmpty()) {  
        const vertex = stack.pop()!;  
        // ...  
    }  
}
```

Implementace DFS (iterativně)

```
if (!visited.has(vertex)) {  
    console.log(vertex); visited.add(vertex);  
  
    for (const neighbor of graph[vertex]) {  
        if (!visited.has(neighbor)) {  
            stack.push(neighbor);  
        }  
    }  
}  
}
```

Porovnání BFS vs DFS

	BFS	DFS
Struktura	Fronta	Zásobník/rekurze
Složitost	$\mathcal{O}(V + E)$	$\mathcal{O}(V + E)$
Paměť	$\mathcal{O}(V)$	$\mathcal{O}(V)$
Nejkratší cesta	Ano	Ne
Detekce cyklů	Obtížnější	Jednodušší

Algoritmy na grafech

Praktické problémy řešené pomocí grafů:

- Jak najít **nejkratší cestu**?
- Jak propojit všechny vrcholy s **minimální cenou**?
- Obsahuje graf **cyklus**?
- Jak najít **optimální pořadí** úloh?

Dijkstrův algoritmus

Princip:

- Hledá nejkratší cestu z jednoho vrcholu do všech ostatních
- Rozšíření BFS o váhy hran
- Používá prioritní frontu (min-heap)
- Greedy přístup - vždy vybere nejbližší nenavštívený vrchol

Požadavky:

- Vážený graf bez záporných cyklů

Složitost: $\mathcal{O}((|V| + |E|) \log|V|)$ s prioritní frontou

Bellman-Ford algoritmus

Princip:

- Hledá nejkratší cestu z jednoho vrcholu do všech ostatních
- Opakuje **relaxaci** všech hran

Algoritmus:

- Projdi všechny hrany $|V| - 1$ krát
- V každé iteraci aktualizuj vzdálenosti
- Poslední iterace detekuje záporné cykly (pokud existují)

Složitost: $\mathcal{O}(|V| \times |E|)$ - pomalejší než Dijkstra

Floyd-Warshall algoritmus

Princip:

- Najde vzdálenosti mezi **všemi páry** vrcholů najednou
- Postupně zvažuje cesty přes každý mezivrchol
- **Dynamické programování**

Algoritmus:

- Pro každý možný mezivrchol k
- Aktualizuj vzdálenost mezi každým párem (i, j)
- Použij vzorec: $d[i][j] = \min(d[i][j], d[i][k] + d[k][j])$

Složitost: $\mathcal{O}(|V|^3)$ - použitelné pro malé grafy

Hashovací tabulka

Hashovací tabulka

Hashovací tabulka (Hash Table/Hash Map) je datová struktura pro rychlé vyhledávání

- Ukládá data jako **páry klíč-hodnota**
- Používá **hashovací funkci** pro výpočet indexu
- Poskytuje (amortizovanou) **konstantní** časovou složitost pro základní operace

Hashovací tabulka

Představte si telefonní seznam

Bez hashování:

- Lineární procházení všech kontaktů
- Hledání trvá dlouho

S hashováním:

- Jméno → hashovací funkce → index
- Okamžitý přístup k číslu

Hashovací funkce

Hashovací funkce převádí **klíč** na **číselný index**

Vlastnosti dobré hashovací funkce:

- **Deterministická** - stejný vstup = stejný výstup
- **Uniformní distribuce** - rovnoměrné rozložení hodnot
- **Rychlá** - efektivní výpočet
- **Minimalizuje kolize** - různé klíče → různé indexy

ASCII součet

```
function simpleHash(key: string, tableSize: number): number {  
    let hash = 0;  
    for (let i = 0; i < key.length; i++) {  
        hash += key.charCodeAt(i);  
    }  
    return hash % tableSize;  
}
```

"abc" má ASCII součet = $97 + 98 + 99 = 294$

Pro tabulku velikosti 10 ukládáme a hledáme hodnotu s klíčem "abc" na indexu $294 \% 10 = 4$

Kolize

Kolize nastává, když dva různé klíče mají **stejný hash**

Příklad:

- `simpleHash("abc", 10) = 4`
- `simpleHash("cab", 10) = 4`

Obě hodnoty bychom chtěli uložit na **stejnou pozici**

Kolize

Řešení kolizí:

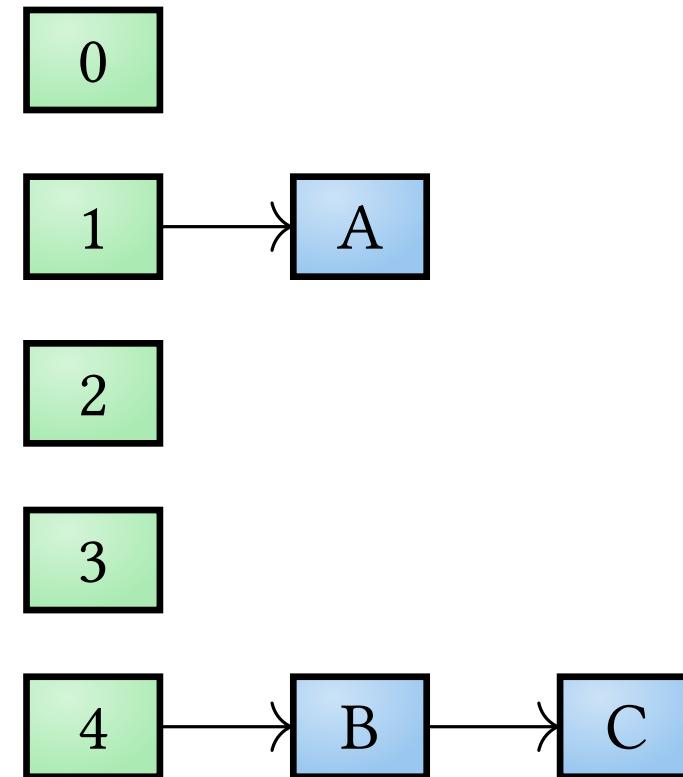
1. Zřetězení (Chaining)

- Každý index obsahuje (spojový) seznam prvků
- Více prvků se stejným hashem tvoří řetězec

2. Otevřené adresování (Open Addressing)

- Při kolizi hledáme další volné místo
- Lineární, kvadratické nebo dvojité hashování

Vizualizace zřetězení



Otevřené adresování

Otevřené adresování (Open Addressing) je alternativní způsob řešení kolizí

- Pokud je index i již obsazen, hledáme další volný index
- Všechny prvky jsou uloženy přímo v tabulce
- Provádíme tzv. **probing** (sondování)

Lineární probing

Princip

Pokud je index i obsazen, zkusíme $i + 1, i + 2, i + 3, \dots$

Pokračujeme dokud nenajdeme volné místo nebo neprojedeme celou tabulkou (cyklicky se vracíme na začátek tabulky)

Lineární probing

Problém - clusterování

- Prvky se shlukují do souvislých bloků
- Vyhledávání se zpomaluje
- Čím větší cluster, tím větší pravděpodobnost dalšího růstu

Lineární probing

Příklad vkládání s lineárním probingem:

```
// Table size 10, hash function: key % 10  
hashTable.insert(23); // h = 3, stored at index 3  
hashTable.insert(13); // h = 3, collision, store at index 4  
hashTable.insert(33); // h = 3, collision, store at index 5  
hashTable.insert(43); // h = 3, collision, store at index 6
```

Vzniká cluster na indexech 3-6

Kvadratický probing

Princip

Pokud je index i obsazen, zkusíme $i + 1^2, i + 2^2, i + 3^2, \dots$

Výhoda: Menší clusterování než u lineárního probingu

Nevýhoda: Může být obtížnější najít volné místo

Kvadratický probing

Příklad vkládání s kvadratickým probingem:

```
// hash(key) = 3, table size 10
// Try indices:
(3 + 0**2) % 10 // = 3
(3 + 1**2) % 10 // = 4
(3 + 2**2) % 10 // = 7
(3 + 3**2) % 10 // = 12 % 10 = 2
(3 + 4**2) % 10 // = 19 % 10 = 9
// ...
```

Dvojité hashování

Princip

Použijeme dvě různé hashovací funkce

- První funkce $h_1(k)$ určí základní index
- Při kolizi použijeme druhou funkci $h_2(k)$ pro výpočet velikosti kroku
 - $h_1(k)$
 - $h_1(k) + h_2(k)$
 - $h_1(k) + 2h_2(k)$
 - ...

Dvojité hashování

Příklad dvojitého hashování:

```
// Table size 10
const h1 = (key: number) => key % 10;          // Primary hash
const h2 = (key: number) => 7 - (key % 7); // Secondary hash
// For key 23:
h1(23); // = 3, try index 3
h2(23); // = 7 - (23 % 7) = 5, step is 5
// on first colision, try index 3 + 5 = 8
```

Důležité: $h_2(k)$ nesmí být 0 a mělo by být nesoudělné s velikostí tabulky

Odstanění prvku

Problém při mazání:

Máme prvky na pozicích i_1, i_2, i_3 (díky kolizím)

Pokud odstraníme prvek na i_2 :

- Při vyhledávání najdeme prázdné místo
- Myslíme si, že prvek neexistuje
- Ale prvek může být na i_3 !

Odstanění prvku - náhrobek

Řešení - náhrobek (tombstone):

Místo smazání označíme pozici jako **smazanou**

- Při vyhledávání: náhrobek ignorujeme a pokračujeme dál
- Při vkládání: náhrobek můžeme přepsat novým prvkem

Shrnutí metod probingu

1. Lineární probing

- Hledáme na pozicích $i, i + 1, i + 2, i + 3, \dots$

2. Kvadratický probing

- Hledáme na pozicích $i, i + 1^2, i + 2^2, i + 3^2, \dots$

3. Dvojité hashování

- Hledáme na pozicích $i, i + f(k), i + 2f(k), \dots$
kde $f(k)$ je jiná hashovací funkce

Časová složitost

Průměrný případ:

- Uložení prvku: $\mathcal{O}^*(1)$
- Vyhledání prvku: $\mathcal{O}^*(1)$
- Odebrání prvku: $\mathcal{O}^*(1)$

Nejhorší případ:

- Všechny operace: $\mathcal{O}(n)$
 - ▶ Nastává při mnoha kolizích (všechny klíče v jednom bucketu)

Load Factor

Load Factor (faktor zaplnění)

$$\frac{\text{počet prvků}}{\text{velikost tabulky}}$$

- Určuje, jak **zaplněná** je hashovací tabulka
- Vyšší load factor = více kolizí
- Optimální hodnota: ~ 0.7 (70% zaplnění)

Load Factor

Když load factor přesáhne limit:

- **Rehashing** - vytvoření větší tabulky
- Všechny prvky se přesunou a přehashují

Výhody hashovací tabulky

- **Velmi rychlé vyhledávání** - průměrně $\mathcal{O}^*(1)$
- **Rychlé vkládání a mazání** - průměrně $\mathcal{O}^*(1)$
- **Flexibilní klíče** - lze použít jakýkoli typ
- **Ideální pro asociativní pole** - slovníky, cache, indexy

Nevýhody hashovací tabulky

- **Negarantovaná rychlosť** - nejhorší případ $\mathcal{O}(n)$
- **Vyšší paměťová náročnost** - většinou není plně zaplněná
- **Neuspořádaná data** - nelze iterovat v pořadí vložení
- **Kvalita závisí na hashovací funkci** - špatná funkce = pomalá tabulka

Hashovací tabulky v JavaScriptu

JavaScript (a tedy i TypeScript) mají vestavěné hashovací tabulky pomocí třídy `Map` a pomocí objektů `Object`

Hashovací tabulky v JavaScriptu

```
const map = new Map<string, number>();
map.set("apple", 5);
map.set("banana", 3);
console.log(map.get("apple")); // 5

const obj: Record<string, number> = {};
obj["apple"] = 5;
obj["banana"] = 3;
console.log(obj["apple"]); // 5
```

Hashovací tabulky v JavaScriptu

```
// Set - hash set (keys only)
const set = new Set<string>();
set.add("apple");
set.add("banana");
console.log(set.has("apple")); // true
```

Praktické využití

Kde se hashovací tabulky používají?

- **Cache** - rychlé ukládání často používaných dat
- **Databázové indexy** - rychlé vyhledávání záznamů
- **Detekce duplicit** - kontrola existence prvků
- **Počítání frekvencí** - histogram, statistiky
- **Implementace Set** - množiny bez duplicit

Příklad

```
function wordCount(text: string): Map<string, number> {
    const counts = new Map<string, number>();
    const words = text.toLowerCase().split(/\s+);
    for (const word of words) {
        counts.set(word, (counts.get(word) || 0) + 1);
    }
    return counts;
}
```

Časová složitost: $\mathcal{O}(n)$ kde n je počet slov