# Stromy

V teorii grafů se jako strom označuje neorientovaný graf, který je souvislý a neobsahuje žádnou kružnici.

**Jedná se o hierarchickou strukturu, kde každý otec má N>= dětí a každé dítě právě jednoho otce takovým způsobem, že v této struktuře nejsou cykly**.

Uzel, který je praotcem všech ostatních uzlů nazveme kořenem (z pohledu teorie grafů tím vytvoříme orientovaný strom).

Uzel, který nemá žádné potomky, nazýváme listem.

Být stromem je rekurzívní vlastnost - **každý podstrom stromu S je také stromem**.

Strom je velmi populární pro svoji jednoduchost a použitelnost. Příkladem mohou být vyhledávací stromy nebo haldy.

Častý strom je dichotomický, který má z každého uzlu 2 pointery: na levý podstrom a na pravý podstrom

# Binární strom

**Binární strom** = dichotomický strom, pro který navíc platí, že všechny klíče v levém podstromu jsou menší a v pravém podstromu větší než klíč

(Předpoklad: klíče jsou jednoznačné)

Nejdůležitější vlastnost: snadné a velmi rychlé hledání půlením.

Je-li N počet uzlů, tak počet kroků ≈ $\log_2 N$.

## Příklad

Jako příklad si ukážeme hledání klíče 28 v binárním stromu.

Pokud hledáme neexistující klíč, snadno ověříme, že opravdu neexistuje.

## Implementace

Binární (a také dichotomické, AVL i další) stromy se dají implementovat mnoha způsoby. Z nich nejčastější jsou:

- pomocí pole (halda = *heap*)
- dynamické struktury propojené referencemi (vzdálená podoba s jednoduchým spojovým seznamem)

*Pozor, "halda=heap" je slangový výraz; v matematice se používá pro něco jiného.*

*Heap* je způsob, jak binární strom zapsat do pole (a ušetřit tak místo za reference).

Vychází z toho, že binární strom má v kořenu 1 uzel, v 1. úrovni 2 uzly, ve 3. úrovni 4 uzly atd, tedy v N-té úrovni je 2N uzlů.

Proto se v poli postupně pro i-tou úroveň přidělí 2N pozic (místa pro uzly, které ve stromu chybí, zůstanou prázdná).

Poměrně jednoduchým algoritmem se dá najít vztah mezi pozicí uzlu v binárním stromě a mezi jeho indexem v poli=*heapu*.

Protože pozice všech uzlů jsou známé, lze pomocí heapu realizovat mnohé algoritmy, které pomocí pointerů napsat nelze.

# Heap – výpočet indexu

Výpočet vztahu mezi pozicí a indexem je jednoduchý.

Nechť kořen je na indexu 1.

Jestliže jsme v uzlu s indexem K, tak (pokud existují):

- jeho rodič má index **K div 2**
- levý podstrom má index **2*K**
- pravý podstrom má index **2*K + 1**

# Implementace strukturou odkazů

```
public class Node {
    public static Node root = null;
    public int key;
    public String data;
    public Node left = null;
    public Node right = null;

public Node(int aKey, String aData){
        key   = aKey;
        data  = aData;
        left  = null;
        right = null;
    }
```

```
public Node add(int aKey, String aData) {
        if (aKey == key){
            System.err.println
            ("duplicate key " + aKey);
            return null;
        } else if (aKey< key) {
            if (left == null) {
                left =
                  new Node(aKey, aData);
                return left;
            } else
                return left.add
                  (aKey, aData);
        } else if (aKey >key) {
                if (right == null) {
```

```
                    right =
                    new Node(aKey, aData);
                    return right;
                } else
                return right.add
                  (aKey, aData);
        }
        return null; // for sure
      }

public Node find(int aKey){
        Node marker = root;
        while(marker != null){
            if (marker.key == aKey){
          break; // found
```

```
            } else if (marker.key< aKey){
                marker = marker.left;
            } else {
                marker = marker.right;
            }
        }
        return marker;
    }
```

## Nevýhoda binárního stromu

Při nevhodném pořadí vkládaných uzlů mohou větve narůstat nerovnoměrně, až v krajním případě může strom zdegenerovat na lineární seznam.

# AVL stromy

Odstraňují hlavní slabinu binárního stromu.

AVL strom je binární strom, u kterého navíc **pro každý jeho uzel platí**, že výška levého podstromu se od výšky pravého podstromu liší nejvýše o 1

**abs( High(L) - High(P) ) ≤ 1**

čili **Nevyváženost ≤ 1**

Pomocí Nevyváženosti je definováno, co AVL strom je, ale nijak se tím neřeší, jak ho vytvořit.

Platí: každý binární strom lze přetransformovat na AVL strom, a to pomocí konečného počtu rotací, **aplikovaných na vhodné uzly stromu**.

## Rotace stromu

Předpokládejme následující výchozí stav. Hvězdička označuje kořen.

V prvním kroku změníme kořen na uzel „26“. Kdyby se jednalo o rotaci podstromu, změníme referenci, co vedla na uzel 32 tak, aby ukazovala na uzel 26.

Ve druhém kroku obrátíme směr větve, aby byla dodržena podmínka, že do kořene nevedou žádné větve.

Tím bohužel vznikne situace, že z uzlu vycházejí tři větve, což je nepřípustné. Proto uzel 28 napojíme pod uzel 32.

Zde je zobrazen výsledný strom.

*Poznámka: Strom, který nám po rotaci vyšel, **není** AVL. To je proto, že jsme rotaci neaplikovali na správný uzel.*

## Rotace - poznámky

Pozor, nikde ale není psáno, že každá rotace (a nad každým uzlem) musí vést k výsledku! Nevhodná aplikace může situaci i zhoršit.

Podobně jako pravá rotace, existuje i **levá rotace**. Je zcela symetrická (*zkouší se!*)

# Rotace - principy

Princip, podle kterého se aplikují rotace, vychází z faktu, že k AVL stromu lze uzel přidat v zásadě 6 způsoby:

# 2-3-4 stromy

## Motivace

Jsou problémy s častým vyvažováním při vytváření stromu. Jediná možnost - uvolnění pravidel.

Možnost navázat na uzel více dětí, tedy vícecestné uzly.

2-3-4 stromy mohou mít 1,2,3 datové položky v uzlu, a tedy 2,3,4 děti.

## Co znamená 2-3-4 strom

2,3,4 je počet dětí uzlu:

- uzel s 1 datovou položkou musí mít 2 děti

- uzel s 2 datovými položkami musí mít 3 děti
- uzel s 3 datovými položkami musí mít 4 děti

Uzel nesmí mít pouze jednoho syna.

List nesmí být nikdy prázdný.

Duplicitní hodnoty nejsou ve stromu povoleny.

Důsledek: List může obsahovat 1,2 nebo 3 datové položky

## Pravidla

Označme: A,B,C ... datové položky, a,b,c ... děti.

Pravidla jsou velmi podobná binárnímu vyhledávacímu stromu:

- Všechny klíče v podstromu a musí být menší než klíč A

- Všechny klíče v podstromu b musí být větší než klíč A a menší než klíč B
- Všechny klíče v podstromu c musí být větší než klíč B a menší než klíč C
- Všechny klíče v podstromu d musí být větší než klíč C

## Vyhledávání

Velmi podobné jako u binárních stromů:

- začíná se v kořenu
- pokud uzel neobsahuje hledaný klíč, vybere se vhodný potomek a hledání se opakuje
- pokud ani v listu hledaný klíč není, pak není ve stromu vůbec

## Vkládání

**Nová položka se vkládá vždy do listu,** protože až v listu máme jistotu, že podobná hodnota neexistuje.

Dvě možné situace při hledání místa pro vložení:

- na cestě není naplněný uzel
- na cestě je naplněný uzel

## Uzel není naplněný

Situace, kdy na cestě nikde není naplněný uzel:

- Přidání do položky neovlivní strukturu stromu
- Operace vkládání je jednoduchá.

### Příklad

Příklad: do tohoto stromu vložit 7.

7 je menší než 11, proto postupujeme levou větví.

V tomto uzlu sice je pro klíč 7 dost místa, ale není to list. Proto ho sem vložit nejde. Postupujeme pravou větví.

Výsledný strom je na obrázku.

## Uzel je naplněný

Situace, že se kdekoliv po cestě najde plný uzel znamená, že musí dojít k jeho rozdělení, což má za následek vyvážení stromu.

Jak rozdělit uzel:

- vytvořit nový uzel a přesunout do něj největší prvek
- prostřední prvek přesunout do rodičovského uzlu (=vytlačit nahoru)
- upravit vazby

## Příklad

Vložte klíč 24 do tohoto stromu.

Když zkusíme vložit klíč 24, tak hned v kořeni stromu nalezneme úplně zaplněný uzel. Proto tento uzel musíme rozdělit.

Detail rozdělení ukazuje obrázek. Uzel 11-40-60 se rozdělí na 3 uzly: 11, 40 a 60. Z toho 11 a 60 zůstanou na původní úrovni, zatímco 40 je vytlačen nahoru.

Po rozdělení má náš strom o 1 patro víc.

## Poznámky

Proč jsme zde 24 nevložili vedle 11?

Nová položka se vkládá vždy do listu, protože až v listu máme jistotu, že podobná hodnota neexistuje.

Výsledný strom je zde.

Proč rozdělovat cestou dolů?

Zjednodušuje to práci:

- ve vyšším patře určitě není plný uzel, protože pokud byl, už je rozdělen
- takže rozdělení uzlu (posunutí položky výš) nezpůsobí problém

# 2-3 stromy

## Přehled

2-3 strom je druh stromu, jehož každý vnitřní uzel má buď dva potomky a obsahuje jeden klíč, nebo má tři potomky a obsahuje dva klíče. Všechny listy leží ve stejné hloubce.

2-3 stromy lze považovat za B-stromy obsahující vnitřní uzly pouze s dvěma nebo třemi potomky.

Díky stejné hloubce listů se 2-3 strom řadí mezi vyvážené stromy.

Hloubka 2-3 stromu s **n** prvky se pohybuje v rozmezí mezi **log3n** a **log2n**, podle použité struktury.

Tomu odpovídá i náročnost operací, jako je vyhledávání, vkládání a odebírání dat z 2-3 stromu.

2-3 stromy jsou velmi podobné 2-3-4 stromům. Rozdíly:

- uzly mohou obsahovat 1 nebo 2 položky
- uzly ukazují na 2 nebo 3 potomky

Operace hledání

- probíhá stejně jako u 2-3-4 stromu
- změní se pouze počet prohledávaných položek

Operace vkládání

- **probíhá úplně jinak. . .**

# Vkládání do 2-3 stromů

Co se změnilo?

Při rozdělení jsou nutné tři položky:

- jedna, která zůstane
- jedna, která se přesune do sourozence
- jedna, která se přesune do rodiče

Jenže 2-3 strom má v uzlu pouze dvě položky. Kde vzít třetí položku?

- musí se použít vkládaná položka
- **proto nelze upravovat cestou dolů**

## Vkládání do neplného listu

Vkládání do neplného listu je bez problémů:

## Vkládání do plného uzlu s neplným rodičem

Je potřeba rozdělit list a přesunout střední položku do rodiče.

## Vkládání do plného uzlu, s plným rodičem

Musí dojít k rozdělení rodiče.

Případně se musí rozdělit rodič rodiče . . . atd.

Rekurzívně dál a dál, dokud se nenajde neplný uzel.

Může to vést k posunu kořene "o patro výše".

## Příklad

Musí dojít k rozdělení rodiče.

Případně se musí rozdělit rodič rodiče . . . atd.

Jenže rodič je plný, a tak ho musíme také rozdělit.

Přitom 34 je zase uprostřed, a tak ho vytlačíme...

...a je z něj nový kořen.

# B-stromy

## B-stromy řádu K

Pozor - „B“ neznamená binární, ale „Bayerovy“.

Doposud probrané struktury předpokládaly stejně rychlý přístup ke všem prvkům, což v praxi neplatí (paměť:disk až $10^6$) a neomezenou velikost paměti (což také neplatí).

B-stromy dávají dobrý návod, jak optimalizovat stránkování paměti (=přesouvání mezi pamětí a diskem) z hlediska rychlosti

Každá stránka obsahuje maximálně 2K a minimálně (s výjimkou kořene) K uzlů.

Každá stránka je buď list (=nevedou z ní žádné větve) anebo z ní vede M+1 větví, kde M=aktuální počet uzlů ve stránce.

Strom narůstá „vzhůru“, tzn. u kořene.

Hloubka všech listů je stejná, přitom K označuje řád stromu.

Je zajištěno naplnění alespoň na 50%.

2-3-4 stromy a 2-3 stromy jsou zvláštním případem B-stromů.

## B-stromy stránkování

Model stránkování (kdyby se do paměti vešlo jen 10 uzlů).

Lze předpokládat, že bude vyžadován přístup k datům „blízko sebe“.

V paměti budeme držet vrcholovou stránku + z každé úrovně stránku na kterou se přistupovalo naposledy.

# Červeno-černé stromy

Červeno-černé stromy se podobají AVL stromům a podobně se s nimi také pracuje.

Při vkládání do AVL stromu musíme udělat nejvýše 2 rotace, ale při odebírání uzlů z AVL stromu se může stát, že budeme muset rotovat skoro všechno.

Tuto nevýhodu červeno-černé stromy nemají, protože **nikdy neděláme víc jak 2 rotace**.

Červeno-černý strom je binární vyhledávací strom. Jedná se o datovou strukturu často používanou pro implementaci asociativního pole.

## Pravidla

Uzly červeno-černých stromů se označují klíčem a barvou (1 bit).

Každý uzel je buďto červený nebo černý.

Každý list (jako listy bereme v tomto případě ukazatele na null) je černý.

Pokud je vrchol červený, jsou oba potomci černí.

Každá cesta z uzlu do podřízeného listu obsahuje stejný počet černých uzlů.

## Důsledky

Hloubka stromu je logaritmická; pro N uzlů je nejvýše $2.\log_2(N+1)$.

Nikdy nejsou dva červené uzly pod sebou.

## Rotace

Po Insert a Delete barvy opravujeme přebarvováním na cestě do kořene, a to pomocí operace, která se nazývá rotace.

Rotace jsou v zásadě stejné jako u AVL stromů.

Je ale potřeba rozebrat podstatně více případů než u AVL stromů. Zájemci naleznou podrobnější rozbor zde.

# Hufmannův strom

Statistická metoda.

Vyhledáme četnosti všech znaků v řetězci a tyto znaky podle četnosti zakódujeme tak, aby nejkratší kód odpovídal nejčastějšímu znaku.

Kódy jsou binární, nejčastější znak bude tedy reprezentován 1 bitem na rozdíl od 8(16) bitů potřebných pro uložení ASCII(UTF) znaku.

Jedná se o prefixový kód, takže žádný znak není prefixem jiného znaku → kódy pro jednotlivé znaky mají různou délku a není potřeba označovat jejich konec (každý znak reprezentuje jeden list stromu).

Kódy vytvoříme Huffmannovým stromem:

- Jednotlivé znaky označíme jako listy stromu
- Tyto listy uložíme do seznamu S (S obsahuje uzly stromu, na začátku tedy pouze leafy)
- Seznam S seřadíme podle četnosti.
- Vybereme z S dva prvky M,N s nejnižšími četnostmi m, n, m < n .
- Vytvoříme uzel p, kde vlevo je M a vpravo je N, a jeho četnost bude m+n

- Vložíme p do S a zpět na bod 3 dokud v S není jen jeden uzel.

Příklad je zde.

Strom samozřejmě musíme přenášet spolu s komprimovanými daty.

Strom můžeme reprezentovat řetězcem tak, že jej projdeme DFS algoritmem a pro každý vrchol uložíme 0 a pro každý list 1 a znak tohoto listu.

Při načítání čteme nuly a vytváříme uzly (směrem doleva). Pokud narazíme na 1, zapíšeme písmeno a vrátíme se do nejbližšího pravého uzlu (pravý uzel rodiče, případně nejvíce levý prvek podstromu - u rodiče vpravo) .

Při dekompresi procházíme strom a rekonstruujeme zprávu.

# Obsah