0. Konvence z teorie stromů

Nejprve uvedeme základní terminologii a značení z teorie stromů, které budeme používat v celém textu k přednášce Datové struktury II.

Stromem rozumíme neorientovaný souvislý graf bez cyklů s vyznačeným bodem, který budeme nazývat kořenem stromu. Pak každý vrchol v stromu spojuje s jeho kořenem právě jedna prostá cesta. Vrcholy na této cestě jsou nazývané předchůdci vrcholu v. Když v není kořen, pak jeho předchůdce, který je spojen hranou s vrcholem v, se nazývá otec vrcholu v (budeme ho značit $\text{otec}(v)$). Když otec vrcholu v není kořen, pak otec otce vrcholu v se nazývá děd vrcholu v (a označujeme ho $\text{ded}(v)$). Vrcholy, jejichž otcem je vrchol v, se nazývají synové vrcholu v. Vrcholy, jejichž předchůdcem je vrchol v, se nazývají následníci vrcholu v. Úplný podgraf stromu T tvořený vrcholem v a všemi jeho následníky se nazývá podstrom určený vrcholem v (je to strom, v němž vrchol v je považován za kořen). Vrcholům, které mají stejného otce, budeme říkat bratři. Vrcholy, které nemají syny, se nazývají listy stromu. Vrcholy, které nejsou listy, jsou vnitřní vrcholy stromu.

Lze říci, že volba kořene stromu vybrala dvě speciální orientace hran – buď hrana vždy směřuje do otce nebo vždy směřuje z otce. Protože získané výsledky se nezmění tím, že přejdeme od jedné speciální orientace k druhé, je jedno, kterou orientaci si představíme. Budeme tedy předpokládat, že hrana vždy směřuje od otce.

Posloupnost vrcholů $(v_0, v_1, \dots, v_k)$ budeme nazývat cestou, jen když v_{i-1} je otcem vrcholu v_i pro každé $i = 1, 2, \dots, k$. V tomto smyslu budeme chápat frázi 'cesta z vrcholu v do listu' (tedy cesta končí v listu, který patří do podstromu určeného vrcholem v). Řekneme, že k je délka cesty. Pro přirozené číslo i je i-tá hladina stromu množina všech vrcholů takových, že délka cesty z kořene do nich je přesně i. Řekneme, že vrchol v má hloubku i, když patří do i-té hladiny (píšeme $h(v) = i$). Vrchol v má výšku i, když délka nejdelší cesty z v do listu je i (píšeme $v(v) = i$). Když délka nejkratší cesty z v do listu je i, pak píšeme $s(v) = i$. Výška stromu T je výška jeho kořene a značíme ji $v(T)$. Počet synů vrcholu v označíme $\rho(v)$.

Nechť každý vnitřní vrchol má totálně uspořádanou množinu synů. Definujme relaci $\leq$ na množině všech vrcholů stromu tak, že $u \leq v$ pro vrcholy u a v stromu, když buď u je předchůdcem v nebo existuje předchůdce u' vrcholu u a předchůdce v' vrcholu v tak, že $u' \neq v'$, u' a v' mají stejného otce w a syn u' vrcholu w je menší než syn v'. Pak $\leq$ je totální uspořádání na množině všech vrcholů stromu. Toto uspořádání se nazývá lexikografickým uspořádáním stromu.

Strom se nazývá binární, když každý vrchol má nejvýše dva syny. Když každý vnitřní vrchol má přesně dva syny, pak řekneme, že strom je úplný binární. V binárním stromě pro každý vrchol v předpokládáme, že je určeno, který jeho syn je levý (budeme ho označovat $\text{levy}(v)$) a který je pravý (značíme ho $\text{pravy}(v)$). Může se stát, že $\text{levy}(v)$ nebo $\text{pravy}(v)$ není definován, pak píšeme, že $\text{levy}(v) = NIL$ nebo $\text{pravy}(v) = NIL$. Vrchol v v binárním stromě je listem, právě když $\text{levy}(v) = \text{pravy}(v) = NIL$. Budeme vždy předpokládat, že levý syn vrcholu v je menší než pravý syn vrcholu v. Řekneme, že vrchol v binárního stromu je lomený, když je definován děd vrcholu v a platí

$$\text{otec}(v) = \text{levy}(\text{ded}(v)) \quad \Leftrightarrow \quad v = \text{pravy}(\text{otec}(v)).$$

Když všechny listy úplného binárního stromu leží v i-té hladině, pak mluvíme o pravidelném

úplném binárním stromu. Platí, že pravidelný úplný binární strom o výšce i má 2^i listů a $2^i - 1$ vnitřních vrcholů. Pro každé $j \leq i$ má j-tá hladina přesně 2^j vrcholů.

1. Kombinatorické vlastnosti červeno-černých stromů a obecné vyvážené binární vyhledávací stromy

Při analýze každé datové struktury, ale i při její implementaci, je výhodné znát její vlastnosti, které pak můžeme použít. V této části textu se nejprve budeme věnovat kombinatorickým vlastnostem červeno-černých stromů.

Připomínáme, že úplný binární strom je červeno-černý, když můžeme obarvit jeho vrcholy barvami červenou a černou tak, že platí

(rb0) každý vrchol stromu je obarven právě jednou barvou, buď červenou nebo černou;
(rb1) každý list je obarven černě;
(rb2) když vrchol v je obarven červeně, pak buď v je kořen nebo otec vrcholu v je obarven černě;
(rb3) všechny cesty z kořene do listů mají stejný počet vrcholů obarvených černě.

Lehce se přesvědčíme, že pro každý vrchol v červeno-černého stromu T platí, že všechny cesty z v do listů mají stejný počet vrcholů obarvených černě. Proto podstrom červeno-černého stromu určený libovolným jeho vrcholem je opět červeno-černý strom.

Funkce f z vrcholů úplného binárního stromu T do přirozených čísel splňující podmínky

(r1) když x je list, pak $f(x) = 0$ a $f(\text{otec}(x)) = 1$;
(r2) pro každý vnitřní vrchol v různý od kořene platí $f(v) \leq f(\text{otec}(v)) \leq f(v) + 1$;
(r3) pro každý vnitřní vrchol v různý od kořene platí, že když otec(v) není kořen, pak $f(\text{ded}(v)) > f(v)$

se nazývá hodnostní funkce T (anglicky rank of T).

Věta 1.1. *Úplný binární strom T je červeno-černý strom, právě když má hodnostní funkci.*

Důkaz. Předpokládejme, že T je červeno-černý strom. Definujme funkci f na vrcholech stromu T tak, že pro každý vrchol v je $f(v)$ počet černých vrcholů na některé cestě z některého jeho syna do listu. Nejprve ukážeme korektnost této definice. Podstrom určený vrcholem v je červeno-černý strom, a tedy všechny cesty z vrcholu v do listů mají stejný počet černých vrcholů. Proto mají i všechny cesty z některého syna vrcholu v do listů stejný počet černých vrcholů. Odtud plyne, že funkce f nezávisí ani na volbě syna ani na volbě cesty, takže její definice je korektní.

Dále ukážeme, že f je hodnostní funkce. Z definice f okamžitě plyne, že f splňuje (r1) (list nemá syna, proto neexistuje ani žádná cesta z jeho syna, a tedy počet černých vrcholů na těchto 'neexistujících' cestách je 0). Protože cesta z vrcholu v do listu má nejvýše o jeden černý vrchol více než cesta z jeho syna, dostáváme, že $f(v) \leq f(\text{otec}(v) \leq f(v)+1$, a tedy je splněna podmínka (r2). Navíc $f(v) < f(\text{otec}(v))$, právě když v je obarven černě. Podmínka (r3) pak plyne z (rb2), a tedy f je hodnostní funkce T.

Nyní dokážeme opačnou implikaci. Předpokládejme, že úplný binární strom T má hodnostní funkci f. Vrchol v stromu T různý od kořene obarvíme černě, když $f(v) < f(\text{otec}(v))$, a obarvíme ho červeně, když $f(v) = f(\text{otec}(v))$. Když pro kořen r stromu T platí, že pro některého jeho syna v je $f(v) = f(r)$, pak r obarvíme černě, jinak ho můžeme obarvit libovolně. Podmínka (rb0) je zřejmě splněna. Z podmínky (r1) okamžitě plyne (rb1). Mějme

červeně obarvený vrchol v stromu T, který není kořen. Pak $f(v) = f(\text{otec}(v))$. Když $\text{otec}(v)$ není kořen, pak podle (r2) $f(\text{ded}(v)) > f(v) = f(\text{otec}(v))$, a tedy $\text{otec}(v)$ je obarven černě. Když $\text{otec}(v)$ je kořen, pak podle definice je $\text{otec}(v)$ obarven černě a (rb2) platí. Protože f je funkce do přirozených čísel taková, že

$f(v) = 0$ pro každý list v;
vrchol v různý od kořene je obarven černě, právě když $f(v) + 1 = f(\text{otec}(v))$;
vrchol v různý od kořene je obarven červeně, právě když $f(v) = f(\text{otec}(v))$,

dostáváme, že každá cesta z vrcholu v různého od kořene do některého listu obsahuje $f(\text{otec}(v))$ černých vrcholů. Speciálně, každá cesta z každého syna kořene do některého listu obsahuje $f(\text{otec}(v))$ černých vrcholů. Z tohoto plyne (rb3). Tedy T je červeno-černý strom. □

Dále ukážeme vztah mezi červeno-černými stromy a $(2, 4)$-stromy.

Konstrukce 1.2. Mějme $(2, 4)$-strom T reprezentující množinu S. Vytvořme binární strom T_1 reprezentující množinu $S \setminus \max S$ tak, že na každý vrchol aplikujeme následující postup:

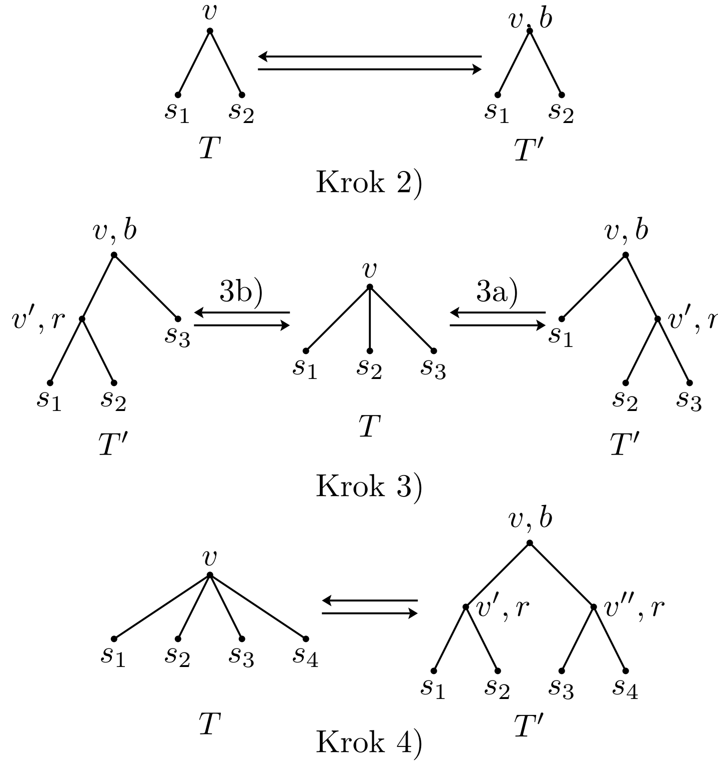
(1) když t je list T, pak v T_1 bude t černě obarvený list;
(2) když v je vnitřní vrchol T se dvěma syny s_1 a s_2 (v tomto pořadí), pak v T_1 bude v černý vrchol reprezentující prvek $H_v(1)$ s levým synem s_1 a pravým synem s_2;
(3) když v je vnitřní vrchol T se třemi syny s_1, s_2 a s_3 (v tomto pořadí), pak v T_1 nastane jedna z alternativ
  (3a) v bude černý vrchol reprezentující prvek $H_v(2)$ s levým synem v' a pravým synem s_3, kde v' je nový červeně obarvený vrchol reprezentující $H_v(1)$ s levým synem s_1 a pravým synem s_2;
  (3b) v bude černý vrchol reprezentující prvek $H_v(1)$ s levým synem s_1 a pravým synem v', kde v' je nový červeně obarvený vrchol reprezentující $H_v(2)$ s levým synem s_2 a pravým synem s_3,
(4) když v je vnitřní vrchol T se syny s_1, s_2, s_3 a s_4 (v tomto pořadí), pak v T_1 bude v černě obarvený vrchol reprezentující $H_v(2)$ s levým synem v' a pravým synem v'', kde v' a v'' jsou nové červeně obarvené vrcholy takové, že v' reprezentuje $H_v(1)$, jeho levý syn je s_1 a pravý syn je s_2 a v'' reprezentuje $H_v(3)$, jeho levý syn je s_3 a pravý syn je s_4.

Tato transformace je znázorněna na obrázku Obr. 1 (barvy jsou zde znázorněny písmeny b – černá a r – červená za jménem vrcholu). Všimněme si, že strom T_1 není určen jednoznačně, v kroku 3) je výběr ze dvou možností. Dále si všimněme, že každý takto zkonstruovaný strom má černě obarvený kořen. Označme $\mathcal{CC}(T)$ všechny stromy, které vznikly z $(2, 4)$-stromu T podle popsané transformace.

Tvrzení 1.3. *Když T je $(2, 4)$-strom reprezentující množinu S, pak $\mathcal{CC}(T)$ je neprázdná množina červeno-černých stromů reprezentujících množinu $S \setminus \max S$.*

Důkaz. Z definice je zřejmé, že $\mathcal{CC}(T) \neq \emptyset$, a když $T' \in \mathcal{CC}(T)$, pak splňuje podmínky (rb0), (rb1) a (rb2) (všimněme si, že červené vrcholy vytváříme jen krokem 3) a 4), přičemž takto vzniklý vrchol není kořen a jeho otcem je černý vrchol). Protože počet černých vrcholů na cestě z kořene do listu ve stromě T' je počet vrcholů na cestě z kořene do listu v T, je platnost (rb3) důsledkem vlastností (a, b)-stromů. Tedy T' je červeno-černý strom a protože

$$\{H_v(i) \mid v \text{ je vnitřní vrchol } T,\ i = 1, 2, \dots, \rho(v) - 1\} = S \setminus \max S,$$



Obr. 1. Transformace mezi červeno-černými stromy a $(2,4)$-stromy

dostáváme, že T' reprezentuje $S \setminus \max S$. Zbývá ukázat, že T' je binární vyhledávací strom pro reprezentovanou množinu. Pro vnitřní vrchol v stromu T označme S_v množinu prvků reprezentovaných v podstromu T určeném vrcholem v. Všimněme si, že každý vnitřní vrchol T je černě obarvený vnitřní vrchol stromu T'. Indukcí podle výšky v dokážeme, že podstrom T' určený vnitřním vrcholem v stromu T reprezentuje množinu $S_v \setminus \max S_v$. Když výška v ve stromě T je 1, pak z definice T' plyne, že podstrom T' určený vrcholem v reprezentuje množinu $\{H_v(i) \mid i = 1, 2, \dots, \rho(v)-1\} = S_v \setminus \max S_v$, protože $\max S_v$ je prvek reprezentovaný $\rho(v)$-tým synem v. Předpokládejme, že tvrzení platí pro vrcholy s výškou menší než $k > 1$ a nechť v je vnitřní vrchol T s výškou k. Označme w_i i-tého syna vrcholu v. Protože $H_v(i) = \max S_{w_i}$ pro $i = 1, 2, \dots, \rho(v) - 1$, dostáváme z indukčního předpokladu, že podstrom T' určený vrcholem v reprezentuje množinu

$$\bigcup_{i-1}^{\rho(v)} S_{w_i} \cup \{H_v(i) \mid i = 1, 2, \dots, \rho(v) - 1\} = S_v \setminus \max S_v,$$

protože $\max S_v = \max S_{w_{\rho(v)}}$. Tvrzení, že T' je binární vyhledávací strom, plyne z pomocného tvrzení a z popisu kroků 2), 3) a 4) v Konstrukci 1.2. $\square$

Konstrukce 1.4. Mějme červeno-černý strom T reprezentující množinu S takový, že jeho kořen je obarven černě. Nechť ∞ je prvek větší než každý prvek univerza U. Na každý

vnitřní černý vrchol v aplikujme následující proces:

(1) když oba synové v jsou obarveny černě, pak označme $S_v(1)$ levého syna v, $S_v(2)$ pravého syna v a $H_v(1)$ prvek reprezentovaný v;
(2) když levý syn u vrcholu v je červený a pravý syn vrcholu v je černý, pak označme $S_v(1)$ levého syna vrcholu u, $S_v(2)$ pravého syna vrcholu u, $S_v(3)$ pravého syna vrcholu v, $H_v(1)$ prvek reprezentovaný vrcholem u a $H_v(2)$ prvek reprezentovaný vrcholem v;
(3) když pravý syn w vrcholu v je červený a levý syn vrcholu v je černý, pak označme $S_v(1)$ levého syna vrcholu v, $S_v(2)$ levého syna vrcholu w, $S_v(3)$ pravého syna vrcholu w, $H_v(1)$ prvek reprezentovaný vrcholem v a $H_v(2)$ prvek reprezentovaný vrcholem w;
(4) když levý syn u vrcholu v a pravý syn w vrcholu v jsou obarveny červeně, pak označme $S_v(1)$ levého syna vrcholu u, $S_v(2)$ pravého syna vrcholu u, $S_v(3)$ levého syna vrcholu w, $S_v(4)$ pravého syna vrcholu w, $H_v(1)$ prvek reprezentovaný vrcholem u, $H_v(2)$ prvek reprezentovaný vrcholem v a $H_v(3)$ prvek reprezentovaný vrcholem w.

Vytvořme strom T' z černých vrcholů stromu T tak, že pro každý černý vnitřní vrchol v je pole H_v uloženo v tomto vrcholu a pole S_v určuje jeho syny (v pořadí daném pomocí pořadí v poli). Červené vrcholy se vynechají. Když v je černě obarvený vrchol, takový, že některý jeho syn je list nebo některý jeho syn je červený a jeho syn je list, pak v podstromě vrcholu v jsou černě obarveny jen vrchol v a listy. Tedy $S_v(i)$ je list pro každé $i = 1, 2, \dots, \rho(v)$ a list $S_v(i)$ pro $i \neq \rho(v)$ reprezentuje prvek $H_v(i)$. Když v je vnitřní vrchol nově vzniklého stromu a $i = 1, 2, \dots, \rho(v) - 1$, pak největší list v podstromu určeném jeho i-tým synem reprezentuje prvek $H_v(i)$ a poslední list celého stromu reprezentuje ∞. Označme takto zkonstruovaný strom $\mathcal{B}(T)$.

Tvrzení 1.5. *Když T je červeno-černý strom reprezentující množinu S takový, že jeho kořen je obarven černě, pak $\mathcal{B}(T)$ je $(2,4)$-strom reprezentující množinu $S \cup \{\infty\}$.*

Důkaz. Z definice $\mathcal{B}(T)$ plyne, že každý vnitřní vrchol stromu různý od kořene má alespoň dva a nejvýše čtyři syny a že počet vrcholů na cestě z kořene do listu je počet černých vrcholů ve stromě T na cestách z kořene do listu. Proto $\mathcal{B}(T)$ je $(2,4)$-strom. Z definice (a,b)-stromu plyne, že pro každý list l s výjimkou posledního existuje právě jeden vnitřní vrchol v a $i = 1, 2, \dots, \rho(v) - 1$ takové, že l je největší list v podstromu i-tého syna vrcholu v (když l reprezentuje prvek x, pak v a i jsou určeny vztahem $H_v(i) = x$). Protože poslední list zkonstruovaného stromu reprezentuje ∞, dostáváme, že listům je přiřazena množina $S \cup \{\infty\}$. Zbývá ukázat, že přiřazení je srovnatelné s lexikografickým uspořádáním listů. Abychom to dokázali, všimněme si, že když v je vnitřní černý vrchol, pak podstrom T_v stromu T určený tímto vrcholem je červeno-černý strom a když reprezentuje množinu S_v, pak podstrom T'_v stromu $\mathcal{B}(T)$ určený vrcholem v se shoduje se stromem $\mathcal{B}(T_v)$, kde však největší list reprezentuje místo ∞ nejmenší prvek a_s v $S \cup \{\infty\}$ větší než všechny prvky v S_v (tedy množina $S_v \cup \{a_s\}$ je reprezentována podstromem stromu $\mathcal{B}(T)$ určeným vrcholem v). Nyní fakt, že T je binární vyhledávací strom a že $s < \infty$ pro každé $s \in S$, implikuje, že se uspořádání shodují, a tedy $\mathcal{B}(T)$ je $(2,4)$-strom reprezentující $S \cup \{\infty\}$. □

Poznámka. Předpoklad, že kořen červeno-černého stromu je obarven černě, není omezující. Když T je červeno-černý strom, jehož kořen je obarven červeně, pak změníme barvu kořene

na černou, strom zůstane červeno-černý a reprezentuje stejnou množinu.

Konstrukce 1.2 a Tvrzení 1.3 dávají návod k implementaci $(2, 4)$-stromů ve vnitřní paměti (to byla jedna z motivací autorů červeno-černých stromů). Přitom přepsání operace **INSERT** pro $(2, 4)$-stromy vedlo k algoritmu pro červeno-černé stromy s jednou rotací nebo jednou dvojitou rotací (které se použijí v situaci, kdy máme vrchol, který má jen tři syny, ale jeden z nich se rozštěpí a vzniknou dva červené vrcholy za sebou - vrchol a jeho otec; když má vrchol čtyři syny a jeden z nich se rozštěpí, pak stačí přebarvování). Pro operaci **DELETE** se efektivní přepsání nepodařilo, současný návod vznikl až později na základě podrobné analýzy operace **DELETE** ve $(2, 3)$-stromech.

Konstrukce 1.4 a Tvrzení 1.5 vedly k následujícímu zobecnění červeno-černých stromů. Uvažujme strom T, který splňuje podmínky (rb0), (rb1) a (rb3) a následující zobecnění podmínky (rb2): Pro každý černý vrchol v stromu T označme T_v úplný podgraf stromu T tvořený všemi následníky w vrcholu v takovými, že cesta z v do w obsahuje jediný černý vrchol (tím je vrchol v). V zobecněné podmínce (rb2) je dána množina úplných binárních stromů a podmínka vyžaduje, aby pro každý černý vrchol v různý od kořene byl strom T_v izomorfní s některým stromem z této množiny. Pro kořen r je podstrom T_r izomorfní s podstromem některého stromu z této množiny. (Pokud bychom chtěli takto přeformulovat původní podmínku (rb2), pak daná množina stromů by obsahovala všechny stromy s výškou nejvýše 1.) Pro každou dvojici a a b přirozených čísel, kde $1 < a$ a $2a - 2 < b$, lze definovat množinu binárních stromů M takovou, že Konstrukce 1.4 a Tvrzení 1.5 platí mezi (a, b)-stromy a zobecněnými červeno-černými stromy, pro něž M určuje zobecněnou podmínku (rb2). Toto už prezentovali autoři červeno-černých stromů, Guibas a Sedgewick, 1978.

Snaha nalézt lepší algoritmy pro operaci **DELETE** vedla k definování polovyvážených stromů. Autor těchto stromů (Olivié, 1980) popsal algoritmy pro operace **INSERT** a **DELETE**, které vyžadovaly konstantní počet rotací a dvojitých rotací. Na tento výsledek navázal Tarjan 1983, který nalezl algoritmus pro červeno-černé stromy vyžadující nejvýše jednu rotaci a jednu dvojitou rotaci nebo dvě rotace. Také ukázal, že strom je červeno-černý, právě když je polovyvážený.

Definice 1.6. Řekneme, že úplný binární strom T je polovyvážený, když $v(v) \leq 2s(v)$ pro každý vrchol v stromu T.

Věta 1.7. *Strom T je červeno-černý, právě když je polovyvážený.*

Důkaz. Nechť T je červeno-černý strom. Vezměme hodnostní funkci f stromu T definovanou ve Větě 1.1. Pak $f(v)$ je počet černých vrcholů na některé cestě z některého syna vrcholu v do listu. Mějme cestu P z vrcholu v do do listu a nechť k je počet černých vrcholů na cestě $P \setminus \{v\}$. Pak délka cesty P je alespoň k a nejvýše $2k$. Tedy pro každý vrchol v stromu T platí

$$f(v) \leq s(v) \leq v(v) \leq 2f(v).$$

Proto T je polovyvážený.

Nechť naopak T je polovyvážený strom. Definujme funkci f z vrcholů stromu T do přirozených čísel indukcí od kořene tak, aby pro každý vrchol v stromu T platilo $\lceil \frac{v(v)}{2} \rceil \leq f(v) \leq s(v)$ (z $v(v) \leq 2s(v)$ plyne $\lceil \frac{v(v)}{2} \rceil \leq s(v)$, protože $s(v)$ je celé číslo). Když r je kořen, zvolme $f(r)$ jako přirozené číslo takové, že $\lceil \frac{v(r)}{2} \rceil \leq f(r) \leq s(r)$. Nyní předpokládejme, že

jsme definovali funkci $f(v)$ pro vnitřní vrchol v stromu T tak, že platí $\lceil \frac{v(v)}{2} \rceil \leq f(v) \leq s(v)$, a nechť w je syn vrcholu v. Položme $f(w) = \max\{f(v) - 1, \lceil \frac{v(w)}{2} \rceil\}$. Pak $\lceil \frac{v(w)}{2} \rceil \leq f(w)$. Z $f(v) \leq s(v) \leq s(w) + 1$ dostáváme, že $f(v) - 1 \leq s(w)$. Protože $\lceil \frac{v(w)}{2} \rceil \leq s(w)$, platí $f(w) \leq s(w)$. Tedy $\lceil \frac{v(w)}{2} \rceil \leq f(w) \leq s(w)$ a protože w byl libovolný syn v, můžeme takto definovat funkci f pro všechny vrcholy stromu T. Navíc můžeme říci, že $f(v) \leq f(w) + 1$. Na druhou stranu $v(w) + 1 \leq v(v)$ a $\lceil \frac{v(v)}{2} \rceil \leq f(v)$ dává $\lceil \frac{v(w)}{2} \rceil \leq f(v)$, a proto $f(w) \leq f(v)$. Tedy $f(w) \leq f(v) \leq f(w) + 1$ a takto definovaná funkce f splňuje podmínku (r2). Protože pro list l stromu T platí $v(l) = s(l) = 0$, dostáváme, že $f(l) = 0$. Když v je otec listu, pak $s(v) = 1 \leq v(v) \leq 2s(v) = 2$, a proto $\lceil \frac{v(v)}{2} \rceil = 1 = f(v)$ a f splňuje (r1). Mějme vrchol v stromu T takový, že v ani otec(v) nejsou kořeny stromu T. Označme $w = \text{otec}(v)$ a $u = \text{ded}(v)$. Z (r2) plyne $f(v) \leq f(w) \leq f(u)$. Z definice f plyne, že $f(v) = \max\{f(w) - 1, \lceil \frac{v(v)}{2} \rceil\}$, a tedy buď $f(v) < f(w) \leq f(u)$ nebo $\lceil \frac{v(v)}{2} \rceil > f(w) - 1 \geq \lceil \frac{v(w)}{2} \rceil - 1$. Protože $v(w) \geq v(v) + 1$ a $f(w) \geq f(v) \geq \lceil \frac{v(v)}{2} \rceil$, dostáváme, že $f(w) = \lceil \frac{v(v)}{2} \rceil \geq \lceil \frac{v(w)}{2} \rceil$. Odtud plyne, že $v(w) = v(v) + 1$ je sudé a $f(v) = f(w) = \lceil \frac{v(w)}{2} \rceil$. Pak $v(u) \geq v(w) + 1$ implikuje, že $f(v) = f(w) = \lceil \frac{v(w)}{2} \rceil < \lceil \frac{v(u)}{2} \rceil \leq f(u)$. Tedy f splňuje podmínku (r3), takže je to hodnostní funkce pro T a podle Věty 1.1 je T červeno-černý strom. □

Na závěr ukážeme vztah mezi AVL-stromy a červeno-černými stromy.

Věta 1.8. *Každý AVL-strom je červeno-černý strom.*

Důkaz. Nechť T je AVL-strom. Pak pro každý vnitřní vrchol v stromu T platí, že $|v(u) - v(w)| \leq 1$, kde u je levý syn vrcholu v a w pravý syn vrcholu v. Dokážeme, že když $v(u) \leq 2s(u)$ a $v(w) \leq 2s(w)$, pak $v(v) \leq 2s(v)$. Všimněme si, že $s(v) = 1 + \min\{s(u), s(w)\}$ a $v(v) = 1 + \max\{v(u), v(w)\}$. Pak $2s(v) = 2 + 2\min\{s(u), s(w)\} \geq 2 + \min\{v(u), v(w)\}$. Z $|v(u) - v(w)| \leq 1$ plyne, že $\max\{v(u), v(w)\} - \min\{v(u), v(w)\} \leq 1$, a proto $2s(v) \geq 2 + \min\{v(u), v(w)\} \geq 1 + \max\{v(u), v(w)\} = v(v)$. Protože pro každý list l stromu T platí $0 = v(l) = 2s(l)$, dostáváme indukcí, že pro každý vrchol v stromu T platí $v(v) \leq 2s(v)$, a tedy T je polovyvážený strom a z Věty 1.7 plyne požadované tvrzení. □

Na závěr této kapitoly uvedeme obecně vyvážené binárnívyhledávacístromy definované Anderssonem. Tento pojem byl motivován Anderssonovým zjištěním, že téměř všechny velké softwarové firmy se nedostaly dál než ke spojovým seznamům, a použití vyvážených stromů by značně zrychlilo jejich produkty. To vedlo Anderssona k definici obecně vyvážených binárních vyhledávacích stromů, protože červeno-černé stromy a AVL-stromy byly příliš složité pro programátory těchto firem (efektivní programy pro tyto struktury lze nalézt na Internetu). Proto navrhl následující strukturu.

Množina $S \subseteq U$ je reprezentována binarním vyhledávacím stromem a navíc jsou dány konstanty $c, d > 1$ takové, že pro každý vnitřní vrchol v je výška jeho podstromu nejvýše $c \log |S_v|$, kde S_v je množina reprezentovana v podstromu vrcholu v a počet úspěšných operací **DELETE** (tj. operací, které odstranily nějaký prvek) provedených od posledního vyvažování a tento počet je nejvýše d. Autor experimentálně ověřoval chování datové struktury pro $c \approx 1.3$ (pak výška těchto stromů v nejhorším případě je menší než výška v nejhorším případě u červeno-černých stromů (zde by bylo $c = 2$) a u AVL-stromů(zde by bylo $c = 1.44$)).

Operace **MEMBE**R(x) a **DELETE**(x) se provedou stejným způsobem jako v klasických (nevyvážených) binárních vyhledávacích stromech. Po úspěšném provedení operace **DELETE** se počet operací **DELETE** zvětší o 1 a když překročí stanovenou hranici, provede se vyvážení celého stromu reprezentujícího množinu S. Operace **INSERT**(x) se také provede jako v klasických binárních vyhledávacích stromech, ale navíc při ní počítáme délku cesty z kořene do listu, který bude reprezentovat x. Po přidání x ji zvětšíme o 1 a když překročí výšku stromu, zvětšíme výšku stromu. Pokud výška překročí stanovené omezení na výšku stromu (toto omezení závisí na velikosti reprezentované množiny), pak se provede vyvážení stromu.

Při operaci **INSERT** se Vyvažování stromu provádí tak, že se najde vrchol v, jehož podstrom se má vyvážit, a tento podstrom se nahradí podstromem reprezentujícím stejnou množinu, ale s nejmenší výškou. Při operaci **INSERT**(x) vrcholem v bude vrchol s nejmenší výškou takový, že jeho podstrom není vyvážený (tj. nesplňuje omezení na svou výšku vzhledem k velikosti jim reprezentované množiny). Nahrazením tohoto podstromu podstromem s nejmenší výškou se zmenší výška celého stromu a ten pak bude splňovat omezení dané na výšku stromu (protože před operací toto omezení splňoval). Při vyvažování po operaci **DELETE** provedeme vyvažování pro kořen stromu, tj. nahradíme původní strom novým binárním vyhledávacím stromem, který reprezentuje stejnou množinu, ale má nejmenší výšku.

Abychom mohli realizovat tuto operaci, je potřeba znát v každém vrcholu velikost množiny reprezentované podstromem tohoto vrcholu. Proto je potřeba po úspěšné operaci **INSERT** nebo **DELETE** (tj. když se přidal nebo odebral prvek) projít cestu od upravovaného vrcholu nazpět ke kořeni a aktualizovat velikost množin reprezentovaných podstromy na této cestě.

Anderssonem navrhované vyvažování nejprve převede nevyvážený podstrom pomocí rotací a dvojitých rotací do úplně zdegenerovaného tvaru (tj. je to jediná cesta z kořene, k níž jsou přidány listy) a pak opět pomocí rotací a dvojitých rotací vytvoři binární vyhledávací strom s nejmenší výškou. V literatuře lze nalézt celou řadu efektivnějších způsobů pro konstrukci binárního vyhledávacího stromu s nejmenší výškou, ale nikde jsem nenašel vzájemné porovnání jejich efektivity. Proto lze říct, že obecné vyvážené binární vyhledávací stromy otevírají celou řadu problémů, které by bylo dobré vyřešit.

Andersson prováděl experimenty s $c = 1.3$ a dostal uspokojivé výsledky. Ale porovnání těchto stromů s AVL-stromy nebo s červeno-černými stromy formuloval pouze pro nejhorší případ. Toto pro praktické použitínenípodstatné. Pokud víme je otevřený problém, jaké jsou vztahy v očekávaném případě, tedy která z těchto variant je pro praxi lepší.

2. Červeno-černé stromy – vyvažováni shora dolů

Klasické datové struktury založené na binárních vyhledávacích stromech jsou vhodné pro řešení uspořádeného slovníkového problému v interaktivním režimu, když nad daty pracuje jen jeden proces. S rozvojem počítačů se však setkáváme s mnoha úlohami, kde nad daty pracuje současně více procesů. Např. když data jsou uložena na serveru a prostřednictvím vzdálených terminálů k nim má přístup více uživatelů. Pak vzniká problém, když některý proces chce aktualizovat data (tj. provádět operaci **INSERT** nebo **DELETE**). V klasických

algoritmech musíme uzavřít celou strukturu, to znamená, že ostatní procesy musí počkat, až skončí probíhající aktualizační operace, a to není výhodné. Tento problém byl vyřešen pro (a,b)-stromy pomocí tzv. paralelní implementace (a,b)-stromů. Když se vrátíme k původní práci, která definovala červeno-černé stromy, tak vidíme, že tam byly navrženy algoritmy, které sice jsou pomalejší než námi prezentované algoritmy (asymptoticky je však složitost stejná), ale zato řeší náš problém. Tam, stejně jako při paralelní implementaci (a,b)-stromů, se vyvažovací operace provádějí shora dolů (a preventivně), ne jako v klasických algoritmech, kde se vyvažuje zdola nahoru. Nyní tyto algoritmy uvedeme. Všimněme si, že v nich stačí uzavřít jen vrcholy v nejbližším okolí pracovního vrcholu (tj. jen vrcholy, které se aktivně účastní rotací a dvojitých rotací) a nemá to vliv na procesy pracující v jiných částech datové struktury.

Nejprve popíšeme operaci **INSERT**. Hlavní idea algoritmu pro tuto operaci byla zajistit, aby otec listu, který bude reprezentovat nový prvek, byl černý. Pak stačí tento list změnit na červený vnitřní vrchol reprezentující nový prvek a přidat mu dva černě obarvené syny, které budou listy. Bohužel toto se to nepovede a musíme splnit složitější požadavek.

Připomeňme si značení z textu k přednášce Datové struktury I. Nechť U je univerzum a platí $-\infty < u < \infty$ pro každé $u \in U$. Když T je binární vyhledávací strom, pak pro kořen r definujeme $\lambda(r) = -\infty$ a $\pi(t) = \infty$ a pro vrchol t různý od kořene definujeme

$$\lambda(t) = \begin{cases} \lambda(\text{otec}(t)) & \text{když } t = \text{levy}(\text{otec}(t)), \\ \text{key}(\text{otec}(t)) & \text{když } t = \text{pravy}(\text{otec}(t)), \end{cases}$$
$$\pi(t) = \begin{cases} \pi(\text{otec}(t)) & \text{když } t = \text{pravy}(\text{otec}(t)), \\ \text{key}(\text{otec}(t)) & \text{když } t = \text{levy}(\text{otec}(t)). \end{cases}$$

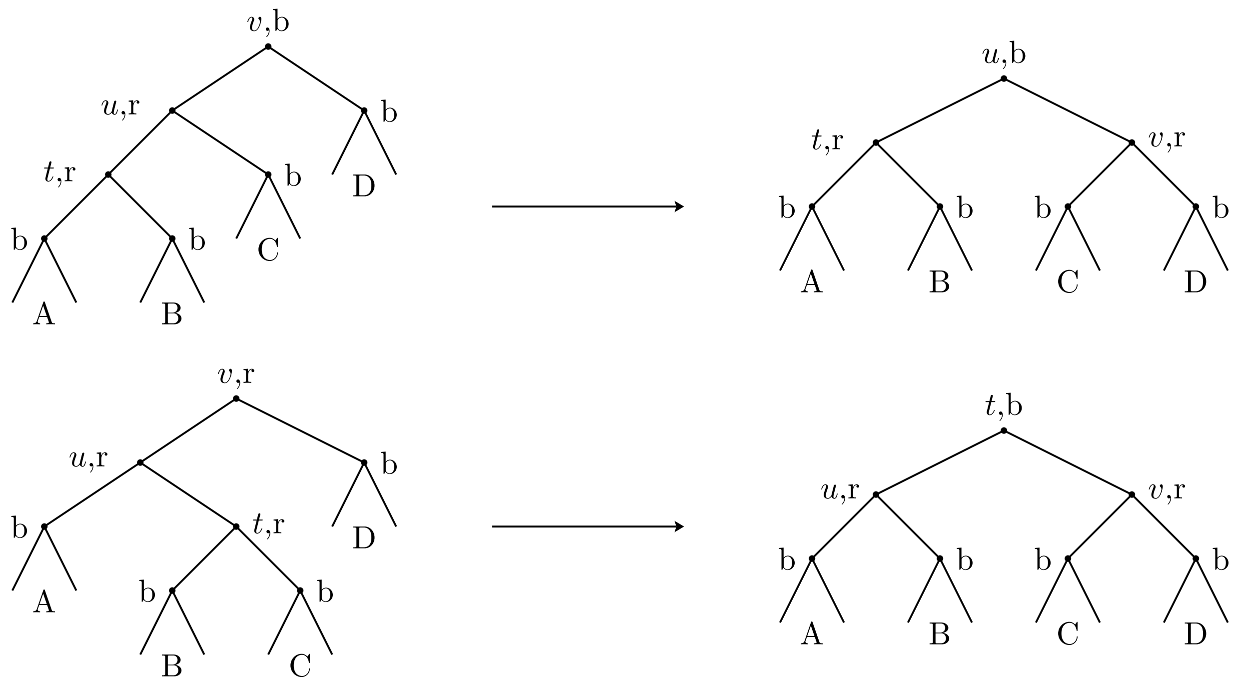
Pak v podstromu určeném vrcholem t jsou právě ty prvky x reprezentované množiny, pro něž $\lambda(t) < x < \pi(t)$. Všimněme si, že když provádíme proceduru **Rotace**(u,v), pak se $\lambda(t)$ a $\pi(t)$ nezmění u žádného vrcholu t různého od u a v. Stejně tak, když provádíme proceduru **Dvojita-rotace**(u,v,w), pak $\lambda(t)$ a $\pi(t)$ zůstane stejné pro vrcholy $t \neq u,v,w$. Tento fakt budeme využívat v následujících algoritmech. Algoritmus zajistí, aby pracovní vrchol t splňoval následující invariant.

> t je černý vrchol takový, že buď je kořen nebo jeho otec nebo bratr jeho otce jsou černé vrcholy, $x \neq \text{key}(t)$ a platí $\lambda(t) < x < \pi(t)$.

Význam tohoto invariantu je založen na faktu, se kterým jsme se již setkali při klasickém vyvažování červeno-černých stromů, a je ilustrován na následujícím obrázku. Kvůli přehlednosti jsou jména vrcholů psána kurzivou a barvy normálními písmeny (černá barva je označena b – black a červená r – red).

Všimněme si, že tyto transformace nezmění počet černých vrcholů na cestě z kořene do žádného vrcholu a ukazují, jak přidat nový prvek x. Při operaci **INSERT**(x) zjistíme, zda x už je reprezentován v množině, nebo nalezneme list t splňující invariant. Pak tento list změníme na červený vnitřní vrchol reprezentující x a pokud je otec tohoto vrcholu červený, provedeme jednu z transformací na Obr. 2 (pokud t je lomený, provedeme dvojitou rotaci, jinak jednoduchou rotaci). Tím dostaneme červeno-černý strom, který reprezentuje původní množinu s přidaným prvkem x.

Zbývá popsat inicializaci procedury a vyhledávací postup směrem od kořene k listu, který bude zachovávat invariant. Nejprve otestujeme, zda kořen reprezentuje x. Když ano,



Obr. 2. Transformace v obarvených stromech.

tak končíme (x nelze přidat, protože už je v reprezentované množině), v opačném případě změníme barvu kořene na černou a první t splňující invariant bude tento kořen.

Nyní popíšeme proceduru, která hledá t na nižší hladině (tj. na hladině s vyšším číslem) a přitom testuje, zda x není v reprezentované množině (procedura se jmenuje **Dale**(t)). Nejprve nalezneme vrchol u tak, že když $\text{key}(t) > x$, pak u je levý syn t, když $\text{key}(t) < x$, pak u je pravý syn t. Když u není list a $\text{key}(u) = x$, tak končíme, když u je list nebo $\text{key}(u) \neq x$ a u je černý, pak u splňuje invariant. Stačí tedy položit $t = u$ a skončit. Zbývá případ, kdy u je červený vrchol a $\text{key}(u) \neq x$. V tomto případě nalezneme vrchol v tak, že když $\text{key}(u) > x$, pak v bude levý syn u, když $\text{key}(u) < x$, pak v bude pravý syn u. Když v není list a $\text{key}(v) = x$, pak končíme. Když v je list nebo $\text{key}(v) \neq x$, pak v je černý, a pokud bratr u je černý, položíme $t = v$ a končíme, protože v splňuje invariant. Pokud u i bratr u jsou červené vrcholy, pak změníme jejich barvu na černou a barvu původního vrcholu t na červenou. Když otec t je černý, pak máme červeno-černý strom, položíme $t = v$ a proceduru ukončíme. Když otec t je červený, pak můžeme použít jednu z transformací na Obr. 2 (v závislosti na tom, zda t je lomený nebo ne) a dostaneme červeno-černý strom. Protože argumentem **Rotace** ani **Dvojita-rotace** není v, tak v splňuje invariant a opět stačí položit $t = v$ a skončit. Tedy buď jsme zjistili, že x patří do reprezentované množiny, nebo jsme dostali vrchol t splňující invariant na nižší hladině. Proměnná *konec* hlídá, zda x není reprezentován daným stromem. Je inicializována hodnotou $false$ a pokud zjistíme, že x je reprezentován, změníme její hodnotu na *true* a to vede k ukončení celého algoritmu. Nyní algoritmus popíšeme formálně.

INSERT-SD(x)
t :=kořen stromu T
if key(t) $= x$ **then stop endif**
if t je červený **then** změň barvu t na černou **endif**
$konec := false$
while $konec = false$ a t není list **do Dale**(t) **enddo**
if $konec = false$ **then**
  t změň na vnitřní vrchol T, t obarvi červeně
  key(t) $:= x$, vytvoř dva černé syny vrcholu t
  (Poznámka: budou to listy)
  if otec(t) je červený **then Oprava**(t) **endif**
endif

Dale(t)
if key(t) $> x$ **then** $u :=$ levy(t) **else** $u :=$ pravy(t) **endif**
if key(u) $= x$ **then**
  $konec := true$
else
  if u je červený **then**
    if key(u) $> x$ **then** $v :=$ levy(u) **else** $v :=$ pravy(u) **endif**
    if key(v) $= x$ **then**
      $konec := true$
    else
      $w :=$ bratr(u)
      if w je červený **then**
        u a w obarvi černě, t obarví červeně
        if otec(t) je červený **then Oprava**(t) **endif**
      endif
    endif
    $t := u$
  else
    $t := u$
  endif
endif

Oprava(s)
$q :=$ otec(s), $p :=$ ded(s)
(Poznámka: q je obarven červeně, p je obarven černě)
if s je lomený **then**
  Dvojita-rotace(p, q, s)
  s obarvi černě, q a p obarvi červeně
else
  Rotace(p, q)
  q obarvi černě, p obarvi červeně
endif

Nejprve si všimněme, že podprocedura **Dale** není volána s vrcholem t, který je list. Dále si všimněme, že když pro nový vrchol u (resp. v) platí $\text{key}(u) \neq x$ (resp. $\text{key}(v) \neq x$) a T je červeno-černý strom, pak u (resp. v) splňuje invariant. Když v některém okamžiku T není červeno-černý strom, pak (T, r, t) je 2-parciální červeno-černý strom a bratr otce vrcholu t je černý (to plyne z invariantu). Tedy můžeme použít podproceduru **Oprava**(t) a vznikne červeno-černý strom. Z toho plyne korektnost algoritmu. Protože podprocedury **Dale** a **Oprava** vyžadují čas $O(1)$ (připomínáme, že **Dvojita-rotace** a **Rotace** vyžadují čas $O(1)$) a protože jsou pro každou hladinu volány nejvýše jednou, vyžaduje algoritmus **INSERT-SD** čas $O(\log n)$, kde n je velikost reprezentované množiny.

Algoritmus pro operaci **DELETE** je složitější než pro **INSERT**, to platí obecně pro všechny datové struktury založené na binárních vyhledávacích stromech. Algoritmus vychází ze základního pozorování, že při úspěšné operaci **DELETE** odstraníme nějaký list a jeho otce. Protože každý list je obarven černě, bylo by vhodné modifikovat strom tak, aby otec odstraňovaného listu byl červený. Pak bychom otce listu jednoduše nahradili bratrem odstraňovaného listu. To vede k následujícímu invariantu pro pracovní vrchol t. Protože odstraňovaný vrchol nemusí reprezentovat x (provádíme operaci **DELETE**(x)), použijeme v něm ještě pomocné proměnné *hot* a w. Invariant má tvar.

> t je černý vrchol různý od kořene, jeho otec je červený vrchol a když $hot = false$, pak jsme ještě nenalezli vrchol v stromu T takový, že $\text{key}(v) = x$ a platí $\lambda(t) < x < \pi(t)$, když $hot = true$, pak $\pi(t) = x = \text{key}(w)$.

Když t je list a $hot = false$, pak skončíme, protože prvek x nepatří do reprezentované množiny. Když t je list a $hot = true$, pak nastanou dvě alternativy – buď $w = \text{otec}(t)$, pak odstraníme vrcholy t a w a bratr vrcholu t nahradí vrchol w, nebo $w \neq \text{otec}(t)$, pak změníme $\text{key}(w)$ na $\text{key}(\text{otec}(t))$, odstraníme t a otce t a bratrem t nahradíme otce t. Díky invariantu dostaneme červeno-černý strom a operace je provedena korektně. Hlavní problém je, jak nalézt vrchol splňující invariant a jak zajistit jeho posouvání směrem k listům. Nalezení vrcholu t splňujícího invariant se povede, pokud synové kořene nejsou listy. Pak buď kořen r reprezentuje x, v tom případě položíme $hot = true$, $w = r$ a t bude levý syn kořene stromu, nebo kořen nereprezentuje x, pak $hot = false$ a když $x < \text{key}(r)$, pak t bude levý syn kořene stromu, jinak t bude pravý syn kořene stromu (zde zatím není podstatná hodnota w, viz invariant). Pak vrchol t splňuje podmínky na funkce λ a π. Předpokládejme, že t je černě obarvený. Když bratr t je obarven černě, pak kořen obarvíme červeně a invariant pro t platí. Když bratr t je obarven červeně, pak provedeme rotaci na otce vrcholu t a bratra vrcholu t a obarvíme otce vrcholu t na červeno a původního bratra vrcholu t na černo (viz Obr. 3). Vrchol t opět splňuje invariant, protože nebyl argumentem rotace. Zbývá případ, kdy vrchol t je obarven červeně. Když $\text{key}(t) = x$, pak změníme t na w, nové t bude levý syn w a $hot = true$, když $\text{key}(t) > x$, pak nové t bude levý syn t, když $\text{key}(t) < x$, pak nové t bude pravý syn t. A invariant opět platí. V případě, že některý syn kořene je list, pak operaci **DELETE**(x) provedeme přímo. Platí totiž

Lemma 2.1. *Když t je list červeno-černého stromu, pak buď jeho bratr je list, nebo otec t je černý, bratr t je červený a synové bratra t jsou listy.*

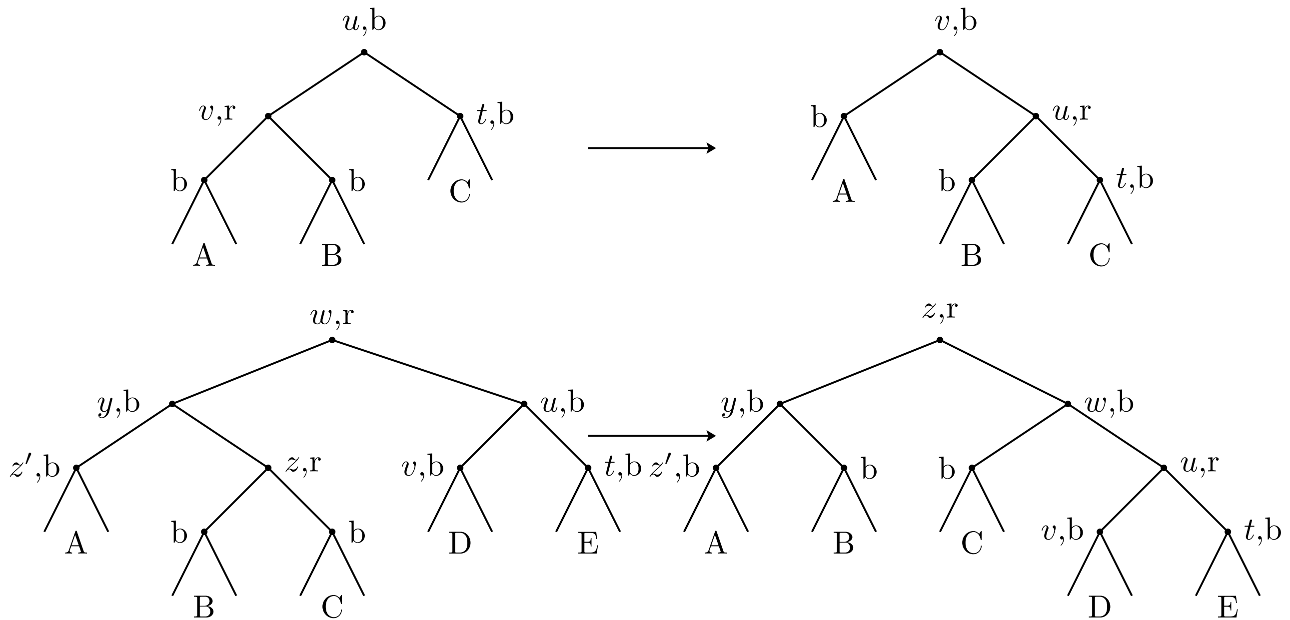
Důkaz. Tvrzení okamžitě plyne z faktu, že cesty z každého vrcholu v do listů v červeno-černých stromech mají stejný počet černých vrcholů. □

Tedy, když jeden syn kořene je list, provedeme jednu z následujících alternativ. Když

kořen reprezentuje x, pak odstraníme kořen stromu i jeho syna, který je list, druhý syn se stane novým kořenem a celá operace končí. Když kořen je list, celá procedura končí, protože strom reprezentuje prázdnou množinu a nelze tedy nic odstranit. Když kořen není list a nereprezentuje x, pak přejdeme standardním způsobem na jednoho jeho syna t. Když x je menší než prvek reprezentovaný kořenem, pak t je levý syn kořene stromu, jinak t je pravý syn kořene stromu. Když t je list nebo nereprezentuje x, pak celá procedura končí, protože x není prvkem reprezentované množiny . Když $\text{key}(t) = x$, pak podle Lemmatu 2.1 jeho synové jsou listy, odstraníme t a jednoho jeho syna a druhý syn nahradí t. Tím celá procedura skončí. Popsali jsme řesení speciálního případu a vratíme se k hlavnímu případu.

Zbývá popsat proceduru (nazývá se **Dale2**(t)), která pro vrchol t splňující invariant nalezne nový vrchol, který splňuje invariant a je v nižší hladině (tj. v hladině s větším indexem). Nejprve přejmenujeme vrchol t na u a standardním způsobem nalezneme nový vrchol t tak, že: Když $hot = true$, pak t bude pravý syn u. Když $hot = false$ a $\text{key}(u) > x$, pak t bude levý syn u, když $hot = false$ a $\text{key}(u) = x$, pak položíme $w = u$, $hot = true$ a t bude levý syn u, když $hot = false$ a $\text{key}(u) < x$, pak t bude pravý syn u. Lehce nahlédneme, že t splňuje podmínky invariantu na λ a π. Ještě je třeba splnit podmínky na barvy. Když t je červený, tak zopakujeme vyhledání nového vrcholu t uvedeným postupem a podprocedura skončí (i nový vrchol splňuje podmínky na λ a π a navíc jsou splněné i podmínky na barvy). Předpokládejme, že t je černý. Když $v = \text{bratr}(t)$ je červený, pak provedeme rotaci na vrcholy $\text{otec}(t) = u$ a v, u obarvíme červeně a v obarvíme černě (viz Obr. 3) a procedura skončí, protože už jsou splněné i podmínky na barvy. Předpokládejme, že i v je černý. Protože otec vrcholu u je podle invariantu červený, tak $y = \text{bratr}(u)$ je černý. Označme z syna y, který je lomený. Když z je červený, pak provedeme proceduru **Dvojita-rotace**$(\text{otec}(u), y, z)$ a změníme barvu otce u na černou a barvu u na červenou (viz Obr. 3) a procedura skončí, zase jsou splněné podmínky na barvy . Když z je černý a jeho bratr z' je červený, pak provedeme rotaci na y a $z' = \text{bratr}(z)$ (viz Obr. 3). Vrchol z' se stane bratrem u a y se stane synem z', který je lomený (barva z' je černá a barva y červená). Tím jsme se dostali do situace v minulém případě. Protože y je teď obarven červeně, můžeme provést dvojitou rotaci na vrcholy $\text{otec}(u)$, z', y a procedura skončí. Když oba synové y jsou černí, pak změníme barvu u a y na červenou a barvu otce u na černou a procedura skončí. Tím jsme popsali neformálně algoritmus pro operaci **DELETE** a nyní dáme formální popis algoritmu pro operaci **DELETE**.

DELETE-SD(x)
u :=kořen stromu T
if ani jeden ze synů u není list **then**
  if $\text{key}(u) = x$ **then**
    $w := u$, $t := \text{levy}(u)$, $hot := true$
  else
    $hot := false$
    if $\text{key}(u) > x$ **then** $t := \text{levy}(u)$ **else** $t := \text{pravy}(u)$ **endif**
  endif
  if t je červený **then**
    if hot **then**
      $t := \text{pravy}(t)$



Obr. 3. Transformace v algoritmu **DELETE-SD**.

    else
        if key(t) $= x$ **then**
            $w := t$, $t :=$ levy(t), $hot := true$
        else
            if key(t) $> x$ **then** $t :=$ levy(t) **else** $t :=$ pravy(t) **endif**
        endif
    endif (Poznámka: t splňuje invariant)
else
    if bratr(t) je černý **then**
        u obarvi červeně
    else
        Rotace1(u, bratr(t))
    endif
endif (Poznámka: Nyní vrchol t splňuje invariant)
while t není list **do Dale2**(t) **enddo**
if $hot = true$ **then**
    if $w =$ otec(t) **then**
        if $w =$ levy(otec(w)) **then**
            levy(otec(w)) := pravy(w), otec(pravy(w)) := otec(w)
        else
            pravy(otec(w)) := pravy(w), otec(pravy(w)) := otec(w)
        endif
        odstraň w a t
    else
        key(w) := key(otec(t))

```
         if ded(t) = w then
            levy(w) := levy(otec(t)), otec(levy(otec(t))) := w
         else
            pravy(ded(t)) := levy(otec(t)), otec(levy(otec(t))) := ded(t)
         endif
         odstraň otec(t) a t
      endif
   endif
else
   if u není list then
      if key(u) = x then
         if levy(u) je list then
            odstraň u a levy(u), otec(pravy(u)) := NIL
         else
            odstraň u a pravy(u), otec(levy(u)) := NIL
         endif
      else
         if key(u) > x then t := levy(u) else t := pravy(u) endif
         if t není list then
            if key(t) = x then
               if t = levy(u) then
                  levy(u) := levy(t)
               else
                  pravy(u) := levy(t)
               endif
               otec(levy(t)) := u, odstraň pravy(t) a t
            endif
         endif
      endif
   endif
endif
```

Dale2(t)

```
u := t
if hot = false then
   if key(u) = x then
      hot := true, w := u, t := levy(u)
   else
      if key(u) > x then t := levy(u) else t := pravy(u) endif
   endif
else
   t := pravy(u)
endif
if t je červený then
   if hot = false then
      if key(t) = x then
```

```
      hot := true, w := t, t := levy(t)
    else
      if key(t) > x then t := levy(t) else t := pravy(t) endif
    endif
  else
    t := pravy(t)
  endif
else
(Poznámka: t je černý)
  if bratr(t) je červený then
    Rotace1(u, bratr(t))
  else
(Poznámka: t, u, bratr(u) i v =bratr(t) jsou černé)
    z je syn bratr(u), který je lomený
    if z je červený then
      Dvojita-rotace1(otec(u), bratr(u), z)
(Poznámka: Nyní t splňuje invariant)
    else
(Poznámka: z je černý)
      if bratr(z) je červený then
        Rotace1(otec(z), bratr(z))
(Poznámka: Z bratra z se stal jeho děd a je to bratr u, otec z je červený a lomený a otec u
je otec děda z)
        Dvojita-rotace1(otec(u), ded(z), otec(z))
(Poznámka: ( t splňuje invariant)
      else
        obarvi u, bratr(u) červeně a otec(u) černě
      endif
    endif
  endif
endif
```

Rotace1(u, v)

Předpoklady: v je obarven červeně, u = otec(v) a bratr(v) jsou obarveny černě (tato akce je znázorněna na Obr. 3)

```
if u = levy(otec(u)) then levy(otec(u)) := v else pravy(otec(u)) := v endif
otec(v) := otec(u), otec(u) := v
if v = levy(u) then
  levy(u) := pravy(v), otec(pravy(v)) := u, pravy(v) := u
else
  pravy(u) := levy(v), otec(levy(v)) := u, levy(v) := u
endif
obarvi v černě a u červeně
```

Dvojita-rotace1(s, y, z)

Předpoklady: t, bratr(t), otec(t) a y jsou černé vrcholy, z a s jsou červené, y = otec(z),

$s = \text{otec}(y) = \text{ded}(t)$ a $\text{otec}(t) = \text{bratr}(y)$ (akce je znázorněna na Obr. 3).
if $s = \text{levy}(\text{otec}(s))$ **then**
  $\text{levy}(\text{otec}(s)) := z$
else
  $\text{pravy}(\text{otec}(s)) := z$
endif
$\text{otec}(z) := \text{otec}(s)$, $\text{otec}(y) := z$, $\text{otec}(s) := z$
if $z = \text{pravy}(y)$ **then**
  $\text{pravy}(y) := \text{levy}(z)$, $\text{levy}(s) := \text{pravy}(z)$, $\text{otec}(\text{levy}(z)) := y$, $\text{otec}(\text{pravy}(z)) := s$
  $\text{levy}(z) := y$, $\text{pravy}(z) := s$
else
  $\text{levy}(y) := \text{pravy}(z)$, $\text{pravy}(s) := \text{levy}(z)$, $\text{otec}(\text{pravy}(z)) := y$, $\text{otec}(\text{levy}(z)) := s$
  $\text{pravy}(z) := y$, $\text{levy}(z) := s$
endif
$\text{otec}(t)$ obarvi červeně, s obarvi černě

Korektnost procedur **Rotace1** a **Dvojita-rotace1** je vidět z Obr. 3. Korektnost procedury **Dale2** a algoritmu **DELETE-SD** plyne z diskuse před formálním popisem těchto procedur. Procedury **Rotace1**, **Dvojita-rotace1** a **Dale2** vyžadují čas $O(1)$ (všimněme si, že **Dale2** volá nejvýše jednou podproceduru **Rotace1** a **Dvojita-rotace1**). Protože **Dale2** je volána na každé hladině nejvýše jednou a protože čas samotného algoritmu **DELETE-SD** (bez rekurzivně volaných podprocedur) je jen $O(1)$, můžeme říci, že operace **DELETE** vyžaduje čas $O(\log n)$, kde n je velikost reprezentované množiny. Shrneme tato fakta. Z předchozích úvah dostáváme:

Věta 2.2. *Algoritmy* **INSERT-SD** *a* **DELETE-SD** *implementují operace* **INSERT** *a* **DELETE***, vyvažují shora dolů a vyžadují čas $O(\log n)$, kde n je velikost reprezentované množiny.*

Nevíme, jestli existují algoritmy pro AVL-stromy, které vyvažují shora dolu a tím umožňují práci více uživatelů v AVL stromu. Je pravděpodobné, že to nejde, protože podmínka na AVL-stromy je globálnějšího rázu než podmínky pro červeno-černé stromy.

3. Relaxované stromy

Nyní popíšeme jinou metodu, která umožňuje současnou práci více uživatelů nad stromovou datovou strukturou. Tato metoda je vhodná i pro práci v dávkovém režimu. S ideou, na které je založena, jsme se poprvé setkali při líné implementaci binomiálních hald. Je založena na pozorování, že když odložíme vyvažování na později, tak je pravděpodobné, že se nám některé vyvažovací požadavky navzájem vyruší a tím si několik vyvažovacích operací ušetříme. Nevýhodou této ideje je fakt, že ztrátou vyváženosti se může výrazně prodloužit čas potřebný k vyhledávání (může se stát, že logaritmický vyhledávací čas vzroste až na lineární). Na druhé straně víme, že při rovnoměrném rozdělení dat je tento nárůst nepravděpodobný. V praxi se sice s rovnoměrným rozdělením dat setkáváme málokdy, ale malá pravděpodobnost nárůstu vyhledávacího času platí i pro rozdělení, která jsou blízká rovnoměrnému rozdělení, což jsou mnohá rozdělení z praxe.

Metoda, kterou popíšeme, od sebe odděluje vyhledávací a vyvažovací operace. Místo toho, aby se provedlo vyvažování, se jen zaznamená požadavek na vyvažování do fronty požadavků. Výhoda této metody se zvláště projevuje v časových špičkách, kdy přichází mnoho požadavků od uživatelů a nelze stihnout vyvažování po každém z nich, nebo při práci v dávkovém režimu (např. když administrátor odeslal dávku požadavků, aby odstranil část datové struktury poškozenou výpadkem proudu). Další výhoda této metody spočívá ve faktu, že ji lze snadno modifikovat pro všechny běžně používané vyvážené stromy. Její použití vyžaduje ošetření dvou problémů. O prvním jsme se již zmínili, je to možný nárůst času pro vyhledávání způsobený degenerací datové struktury. Druhý problém je, zda lze jednoduše vyvážit binární strom, který vznikl několika aktualizačními operacemi (bez následného vyvažování) z vyváženého binárního vyhledávacího stromu (bez nového budování celého vyváženého vyhledávacího stromu, jen na základě reprezentovaných dat). S ním je spojena otázka strategie řešení vyvažovacích požadavků.

Stromy vzniklé touto metodou se nazývají relaxované. Přitom konkrétních realizací této základní ideje je pro každý typ vyvážených stromů několik (záleží na tom, který parametr je preferován). Popíšeme jeden relaxovaný model pro červeno-černé stromy.

Uvažovaný model má data uložená v binárním vyhledávacím stromu, jehož vnitřní vrcholy jsou obarveny buď červeně nebo černě (obarvení oběma barvami není přípustné) a listy jsou obarveny černě. Navíc je dán soubor vyvažovacích požadavků (budeme ho nazývat fronta vyvažovacích požadavků) a pokud je tento soubor prázdný, pak strom reprezentující data je vyvážený červeno-černý strom. Nad daty pracuje současně více procesů, které jsou dvou typů:
uživatelský – provádí pouze vyhledávání, přidávání a ubírání prvků a když po aktualizaci vznikne požadavek na vyvažování (význam i formu upřesníme později), dá tento požadavek do fronty vyvažovacích požadavků.
správcovský – bere vhodné požadavky z fronty vyvažovacích požadavků a provádí je. Může se stát, že buď daný požadavek úplně ošetří nebo ho transformuje v jiný požadavek bližší ke kořeni stromu. Tím se postupně odstraňuje nevyváženost stromu.

Ideálem je v jistých periodách vyprázdnit frontu požadavků a tím vytvořit klasický červeno-černý strom. Počet pracujících správcovských procesů není fixní a závisí na délce fronty vyvažovacích požadavků.

Nyní popíšeme sémantiku a formu požadavků. Budeme mít požadavky dvou typů – b a v. Když je s vrcholem v svázán požadavek b, pak s ním už není svázán žádný další požadavek (tj. požadavek b je neslučitelný s každým dalším požadavkem, zatímco požadavků typu v na vrchol v může být víc). Pokud je s vrcholem svázán nějaký požadavek, pak vrchol má ukazatel na tento požadavek ve frontě vyvažovacích požadavků. Požadavek b na vrchol v znamená, že v je červený a má červeného otce (to platí v okamžiku vznesení požadavku, tato situace se později může změnit). Požadavek v na vrchol v znamená, že v je černý a v podstromu vrcholu v chybí jeden černý vrchol (z toho plyne, že když je na vrchol v vloženo k požadavků v, pak v podstromu určeném vrcholem v chybí k černých vrcholů, tj. kdyby každý vrchol v s k požadavky v představoval $k+1$ černých vrcholů, pak všechny cesty z kořene do listů by měly stejný počet černých vrcholů). Ve frontě vyvažovacích požadavků je každý požadavek specifikován svým typem a ukazatelem na vrchol, kde vznikl.

Nyní neformálně popíšeme práci jednotlivých procesů. Uživatelský proces provádí jednu ze tří operací: **MEMBER**, **INSERT** a **DELETE**.

Operace **MEMBER**(x) klasickým vyhledáváním zjistí, zda x patří do reprezentované množiny, a oznámí to uživateli (pokud patří, oznámí také adresu dat spojených s prvkem x).

Operace **INSERT**(x) klasickým vyhledáváním zjistí, zda x patří do reprezentované množiny. Když patří, operace končí (zde je několik přirozených alternativ – např. oznámí neúspěšné vložení prvku nebo oznámí adresu dat spojených s prvkem x atd.) Když x nepatří do reprezentované množiny, tak vyhledávání skončilo v listu t (který reprezentuje interval obsahující x). Nyní změní list t na vnitřní vrchol, vytvoří dva černé syny vrcholu t, které budou listy, uloží data spojená s x a jejich adresu spolu s prvkem x spojí s vrcholem t. Když t vznášel požadavek v, pak odstraní jeden požadavek v vznesený vrcholem t a nechá ho černým, když t nevznášel žádný požadavek v, pak změní jeho barvu na červenou a když otec vrcholu t je červený, vytvoří pro vrchol t požadavek b a vloží ho do fronty vyvažovacích požadavků (propojí vrchol t a tento požadavek ukazateli).

Operace **DELETE**(x) klasickým způsobem zjistí, zda x patří do reprezentované množiny. Když nepatří, operace končí (případně oznámí neúspěch). V opačném případě nalezne vrchol t a jeho syna l, které mají být odstraněny, uvolní data spojená s prvkem x, odstraní vrchol t a list l a na místo vrcholu t dá bratra listu l, kterého obarví na černo. Když t i bratr l byly obarveny černě, vytvoří požadavek v spojený s bratrem l a vloží ho do fronty vyvažovacích požadavků (a propojí ukazateli bratra l s tímto požadavkem). Požadavky b spojené s vrcholy t a bratrem l se ruší a odstraní se z fronty vyvažovacích požadavků (když t nebo bratr l měly požadavek b, pak jejich odstraněním žádný nový požadavek nevzniká). Navíc bratr l dědí všechny požadavky v spojené s vrcholem t (tím se mohou hromadit požadavky v spojené s jedním vrcholem).

Nyní neformálně popíšeme práci správcovského procesu. Proces provede jednu z následujících akcí:

Zruší (a odstraní z fronty) jakýkoliv požadavek na kořen stromu.

Zruší (a odstraní z fronty) požadavek b na vrchol v, když otec vrcholu v je černý.

Když je na vrchol v vloženo i požadavků v a na bratra vrcholu v je vloženo j požadavků v, kde $i, j > 0$, pak zruší $\min(i, j)$ požadavků v na vrcholy v a bratr(v). Když otec vrcholu v je černý, pak vloží $\min(i, j)$ požadavků v na otce vrcholu v. Když otec vrcholu v je červený, pak změní jeho barvu na černou a vloží na něho $\min(i, j) - 1$ požadavků v.

Když je na vrchol v vložen požadavek b a jeho otec je červený a je kořen stromu, pak obarvíme otce vrcholu v na černo a požadavek zrušíme.

Všimněme si, že tyto akce správcovského procesu vždy zmenší počet požadavků ve frontě požadavků.

Když je na vrchol v vložen požadavek b, otec vrcholu v je červený, děd vrcholu v je černý, bratr u otce vrcholu v je červený, pak obarvíme otce vrcholu v a vrchol u na černo a zrušíme požadavek na vrchol v. Když na děda vrcholu v je vloženo i požadavků v pro $i > 0$, pak odstraníme jeden požadavek v na děda vrcholu v. Pokud na děda vrcholu v není vložen žádný požadavek v (požadavek b na něho nemůže být vložen, protože je černý), obarvíme ho na červeno a pokud otec děda vrcholu v je také červený, pak vložíme na děda vrcholu v požadavek b. Když na vrchol u je vložen požadavek b, tak ho zrušíme.

Všimněme si, že mohl nově vzniknout jedině požadavek b, když otec děda vrcholu v byl červený a tato situace skutečně vedla k položení požadavku b. Dále se počet požadavků buď

zmenšil (pokud vrchol u nebo děd vrcholu byly spojeny s nějakým požadavkem nebo otec děda vrcholu v nebyl červený), nebo se nezměnil, ale požadavek b na vrchol v byl nahrazen požadavkem b na děda vrcholu v, tedy jeho hloubka se zmenšila o 2 a u ostatních požadavků se nic nezměnilo.

Když na vrchol v je vložen požadavek b, otec z vrcholu v je červený, děd w vrcholu v je černý, bratr otce vrcholu v je černý a v není lomený, pak provedeme rotaci vrcholů z a w, w obarvíme červeně, z obarvíme černě, vrchol z zdědí všechny požadavky v vrcholu w a zrušíme požadavek b na vrchol v.

Když na vrchol v je vložen požadavek b, otec z vrcholu v je červený, děd w vrcholu v je černý, bratr otce vrcholu v je černý a v je lomený, pak provedeme dvojitou rotaci vrcholů v, z a w, w obarvíme červeně, v obarvíme černě, zrušíme požadavek b na vrchol v a vrchol v zdědí všechny požadavky v na vrchol w.

Všimněme si, že u obou posledně popsaných akcí správcovského procesu platí, že pokud po provedení akce má nějaký červený vrchol červeného otce, tak měl červeného otce i před akcí (i když se otec vrcholu mohl změnit), a žádný nový požadavek v nevznikl. Tedy počet požadavků vždy klesne.

Srovnejte tyto tři akce s vyvažováním po operaci **INSERT** v klasickém červeno-černém stromu.

Když na vrchol v je vloženo j požadavků v pro $j > 0$, bratr u vrcholu v je černý a není na něho vložen žádný požadavek, synové vrcholu u jsou černí, pak obarvíme vrchol u červeně, zrušíme jeden požadavek v na vrchol v a když byl otec vrcholu v červený, tak ho obarvíme na černo, a pokud byl černý, tak vytvoříme požadavek v na otce vrcholu v a vložíme ho do fronty vyvažovacích požadavků.

V tomto případě buď jeden požadavek na vrchol v zanikne nebo se zmenší jeho hloubka (přesune se na otce vrcholu v), a ostatní požadavky se nezmění.

Když na vrchol v je vloženo j požadavků v pro $j > 0$, bratr u vrcholu v je černý a není na něho vložen žádný požadavek, syn w vrcholu u, který je lomený, je červený, pak provedeme dvojitou rotaci na vrcholy w, u a otce z vrcholu v, obarvíme vrchol z černě, zrušíme jeden požadavek v na vrchol v, a zrušíme také případný požadavek b na vrchol w. Pak vrchol w zdědí barvu i požadavky spojené s vrcholem z.

Když na vrchol v je vloženo j požadavků v pro $j > 0$, bratr u vrcholu v je černý a není na něho vložen žádný požadavek, syn w vrcholu u, který není lomený, je červený a jeho bratr je černý, pak provedeme rotaci na vrcholy u a otce z vrcholu v, obarvíme vrcholy z a w černě, zrušíme jeden požadavek v na vrchol v a vrchol u zdědí barvu i požadavky spojené s vrcholem z. Zrušíme také případný požadavek b na vrchol w.

Všimněme si, že v obou případech nevznikl žádný požadavek v a pokud po akci má červený vrchol červeného otce, tak tato situace byla i před akcí (to se mohlo stát jen vrcholu v kořeni rotace nebo dvojité rotace). Proto se počet požadavků vždy zmenší.

Když na vrchol v je vloženo j požadavků v pro $j > 0$ a bratr u vrcholu v je červený a není na něho vložen žádný požadavek a jeho lomený syn je černý, pak provedeme rotaci na vrcholy u a otce z vrcholu v, obarvíme z na červeno (a nebude na něm žádný požadavek), vrchol u obarvíme na černo a zdědí všechny požadavky vrcholu z. S ostatními vrcholy neprovedeme žádnou změnu. Zde je možné skončit, ale pro zjednodušení důkazu, že se vyprázdní fronta požadavků, provedeme ještě jednu akci správcovského procesu na vrchol v. Protože první

akce nemění počet požadavků, tak druhá akce na vrchol v zmenší počet požadavků ve frontě požadavků.

Předchozí akce srovnejte s operací **DELETE** pro klasické červeno-černé stromy.

Nyní stručně popíšeme práci s frontou vyvažovacích požadavků. Správcovský proces se náhodně rozhodne, zda chce odstranit požadavky na kořen, nebo zda chce odstranit požadavek typu b, nebo zda chce odstranit požadavek typu v.

Když chce odstranit požadavek na kořen, pak prohledáním fronty vyvažovacích požadavků nalezne požadavek na kořen. Zablokuje kořen. Během této akce nesmí být kořen argumentem žádné rotace nebo dvojité rotace (pak by přestal být kořenem a požadavek nejde odstranit). Požadavek odstraní a kořen uvolní. Místo prohledávání fronty je rychlejší začít u kořene, zjistit, zda je na něj položen požadavek, a nalézt tento požadavek pomocí ukazatele.

Když chce odstranit požadavek typu b, pak nejprve nalezne vrchol v, který není kořen, je na něho položen požadavek b a na otce vrcholu v není položen požadavek b. Zablokuje vrchol v a testuje, zda otec vrcholu není černý nebo není kořen stromu. Když otec vrcholu v je černý nebo je kořen stromu, pak zablokuje otce vrcholu v a provede akci. Po jejím provedení uvolní oba vrcholy. Když otec vrcholu v je červený a není kořen stromu, tak zablokuje otce vrcholu v, bratra otce vrcholu v a děda vrcholu v a podle vlastností bratra otce vrcholu v provede akci a vrcholy uvolní.

Když chce odstranit požadavek v, tak nalezne vrchol v, na který je vložen požadavek v a není kořen a na bratra vrcholu v není vložen požadavek b. Pak zablokuje otce vrcholu v, bratra vrcholu v i vrchol v. Podle vlastností bratra vrcholu v provede akci. V případě, že bratr vrcholu v je černý a není na něho vložen žádný požadavek, tak zablokuje i syny bratra vrcholu v. Po provedení akce uvolní vrcholy. Pokud bratr vrcholu v byl obarven červeně, tak je výhodné provést na vrchol v dvě akce, a pak teprve uvolnit vrcholy.

Při volbě jaký požadavek bude ošetřovat správcovský server, by největší prioritu mělo mít ošetřování kořene, pak ošetření požadavku b a pak požadavku v. To by se mělo odrazit při volbě akce správcovského procesu. Vhodný poměr priorit není znám. Zdá se, že je také výhodné začít hledat požadavky ve stromě a pak přejít do fronty vyvažovacích požadavků pomocí ukazatele. Zablokovaný vrchol znamená, že není přístupný jinému správcovskému procesu. Jen při provádění rotace nebo dvojité rotace jsou její argumenty zablokovány i pro uživatelské procesy.

Všimněme si, že když se má obsloužit požadavek v na vrchol v, jehož bratr je červený a lomený syn bratra vrcholu v je také červený, tak buď bratr vrcholu v vznesl požadavek b nebo jeho lomený syn vznesl požadavek b a ten lze vyřídit. Tedy když je fronta požadavků neprázdná, tak vždy existuje požadavek, který je možno vyřídit.

Následující tvrzení se ověří přímo (viz vyvažování pro klasické červeno-černé stromy).

Věta 3.1. *Když fronta vyvažovacích požadavků je neprázdná, tak v ní vždy existuje požadavek, který může obsloužit správcovský proces. Když vrchol v binárního vyhledávacího stromu reprezentujícího data je červený a jeho otec je také červený, pak na vrchol v byl vložen požadavek* b*. V každém okamžiku platí, že když každý vrchol binárního vyhledávacího stromu reprezentujícího data, na který je vloženo i požadavků* v *pro $i > 0$, je nahrazen $i+1$ černými vrcholy, pak všechny cesty z kořene do listů mají stejný počet černých vrcholů.*

Z této věty plyne, že když je fronta vyvažovacích požadavků prázdná, pak binární vyhledávací strom reprezentující data je červeno-černý. Objevuje se však otázka, zda každá posloupnost akcí správcovského procesu vede k vyprázdnění fronty vyvažovacích požadavků. K tomu použijeme amortizovanou složitost. Použijeme dvě ohodnocení. První ohodnocení bude počet požadavků. Druhé ohodnocení dostaneme tak, že když vrchol v hloubce i vznesl požadavek, tak tento požadavek ohodnotíme i a tato ohodnocení požadavků pak sečteme. Z předchozí analýzy plyne, že každá akce správcovského procesu buď zmenší počet požadavků, tedy zmenší první ohodnocení, nebo se první ohodnocení nezmění, ale pak klesne druhé oohodnocení. Odtud dostáváme

Tvrzení 3.2. *Obě ohodnocení každé fronty vyvažovacích požadavků jsou nezáporná a obě se rovnají* 0*, právě když je fronta prázdná. Každá akce správcovského procesu sníží buď první ohodnocení nebo se první ohodnocení nezmění, ale sníží se druhé ohodnocení. Proto lze každou frontu požadavků vyprázdnit.*

Předchozí tvrzení ukazuje, že každá posloupnost akcí správcovského procesu vede k vyprázdnění fronty vyvažovacích požadavků, a je zřejmé, že posloupnost operací v klasických červeno-černých stromech vyžaduje více vyvažovacích akcí. Jsou zde otevřené otázky:

Lze najít optimální strategii pro správcovské procesy? S jakou pravděpodobností je binární strom v závislosti na délce fronty ještě blízký pravidelnému úplnému binárnímu stromu?

Podobné modely jsou studovány i pro AVL-stromy a (a, b)-stromy.

4. Náhodné binární vyhledávací stromy

Z přednášky Datové struktury I víme, že čas, který vyžadují klasické algoritmy pro vyhledávání v binárních vyhledávacích stromech, je úměrný výšce stromu. Význam očekávané výšky binárního vyhledávacího stromu a její odhad budeme ilustrovat na souvislosti binárního vyhledávacího stromu a algoritmu **QUICKSORT**. Uvažujme **QUICKSORT**, kde pivot je volen jako první člen vstupní posloupnosti, rozdělení podle pivota se provadí tak, že jedním průchodem posloupnosti ji rozdělíme na dvě posloupnosti (nepoužíváme žádné výměny). Tedy ve vzniklých posloupnostech je zachováno pořadí ze vstupní posloupnosti. Jako vstup předpokládáme prostou n-člennou posloupnost $\mathcal{P}$ prvků $x_1, x_2, \dots, x_n$ náhodně vybraných z totálně uspořádaného univerza. Dále vytvořme binární vyhledávací strom T reprezentující množinu $\{x_1, x_2, \dots, x_n\}$ tak, že aplikujeme posloupnost operací

$$\mathbf{INSERT}(x_1), \mathbf{INSERT}(x_2), \dots, \mathbf{INSERT}(x_n)$$

na původně prázdný binární vyhledávací strom (použijeme klasický algoritmus **INSERT** bez vyvažovacích operací). Množina reprezentovaná podstromem určeným levým synem kořene T je $\{x_i \mid i \in \{2, 3, \dots, n\}, x_i < x_1\}$ a množina reprezentovaná podstromem určeným pravým synem kořene T je množina $\{x_i \mid i \in \{2, 3, \dots, n\}, x_i > x_1\}$. Přitom tyto stromy vznikly aplikacemi posloupností $\{\mathbf{INSERT}(x_i) \mid i = 1, 2, \dots, n, x_i < x_1\}$ a $\{\mathbf{INSERT}(x_i) \mid i = 1, 2, \dots, n, x_i > x_1\}$. A $\{x_i \mid i = 2, 3, \dots, n, x_i < x_1\}$ a $\{x_i \mid i = 2, 3, \dots, n, x_i > x_1\}$ jsou právě posloupnosti, na které se rekurzivně volá **QUICKSORT** při vstupu $\mathcal{P}$ (v tomto pořadí). Indukcí podle n tak dostáváme, že podstrom určený levým

synem kořene T je izomorfní s podstromem rekurzivních volání **QUICKSORT**u na posloupnost $\{x_i \mid i = 1, 2, \dots, n, \mid x_i < x_1\}$ a podstrom pravého syna kořene T je izomorfní s podstromem rekurzivních volání **QUICKSORT**u na posloupnost $\{x_i \mid i = 1, 2, \dots, n,\ x_i > x_1\}$ (pivot je vždy reprezentován v kořeni podstromu). Z toho plyne, že celý strom T je izomorfní se stromem rekurzivních volání algoritmu **QUICKSORT** na vstupní posloupnost $\mathcal{P}$. Tedy výška binárního stromu T je rovna počtu v sobě vložených rekurzivních volání algoritmu **QUICKSORT** na vstupní posloupnost $\mathcal{P}$. A analogicky očekávaná výška stromu T je očekávaný počet v sobě vložených rekurzivních volání algoritmu **QUICKSORT**. Její hodnota závisí na rozdělení vstupních dat, které je pro algoritmus **QUICKSORT** stejné jako pro posloupnost operací vytvářejících binární vyhledávací strom T.

Protože už víme, že očekávaný čas algoritmu **QUICKOSORT** je $O(n \log n)$ čas potřebný pro přechod z jedné hladiny na nižší obvykle vyžaduje čas $O(n)$ lze odvodit, že asi oč ekávaná výška tohoto stromu je $O(\log n)$. To motivovalo snahu to ověřit. Přesná analýza tohoto problému překračuje rámec této přednášky. Proto uvedeme jen zatím nejnovější výsledek (bez důkazu), který získali nezávisle na sobě Reed 2003 a Drmota 2003 (autoři navázali na předchozí práce Robsona 1979, Devroye 1986, 1995 a dalších).

Věta 4.1. *Očekávaná výška náhodného binárního vyhledávacího stromu s n vrcholy při rovnoměrném rozdělení dat je $E(v(n)) = \alpha \ln(n) - \beta \ln\ln(n) + O(1)$, kde α je jediné řešení rovnice $\alpha \ln(\frac{2e}{\alpha}) = 1$ v intervalu $< 2, \infty)$ (tj. $\alpha \doteq 4.31107$) a $\beta = \frac{3}{2\ln(\frac{\alpha}{2})} = \frac{3\alpha}{2\alpha - 2}$ (tj. $\beta \doteq 1.953$). Rozptyl $v(n)$ je $O(1)$.* $\square$

Tento výsledek ukazuje, že náhodné binární vyhledávací stromy mají výšku přibližně $\alpha \ln(n)$, a protože náhodné posloupnosti operací **INSERT** nad daty s rovnoměrným rozdělením vytvářejí náhodné úplné binární stromy s rovnoměrným rozdělením (požadavek, že stromy jsou úplné, zvětší očekávanou výšku oproti obyčejným binárním stromům nejvýše o 1), tak očekávaný čas operací **MEMBER**, **INSERT** a **DELETE** v binárních vyhledávacích stromech bez vyvažování (s rovnoměrným rozdělením vstupních dat) je $O(\log(n))$, kde n je velikost reprezentované množiny. Když vstupy nejsou rovnoměrně rozděleny, pak algoritmus může mít podstatně horší chování. Situace je podobná jako při hašování, kde tento problém řeší princip univerzálního hašování. Idea univerzálního hašování je mít množinu funkcí takovou, že pro každou vstupní množinu je v ní dost funkcí, které se chovají dobře vzhledem k dané vstupní množině. Pak náhodná volba hašovací funkce udělá možnost špatného výběru funkce vzhledem k danému vstupu za málo pravděpodobnou. V našem případě budeme modifikovat tuto ideu tím způsobem, že rozšíříme argument operace tak, aby ovlivňoval způsob práce se stromem, a tuto přidávanou komponentu vstupu budeme volit náhodně s rovnoměrným rozdělením. Tím chceme odstranit závislost na rozdělení vstupních posloupností. Vznikají zde tři problémy. První z nich je, aby přidaná komponenta zajistila, že vzniklé stromy budou mít rovnoměrné rozdělení (stručně řečeno, aby vytvářené binární vyhledávací stromy kopírovaly rozdělení náhodných dat). Druhým problémem je rychlé vygenerování náhodných dat s rovnoměrným rozdělením. Máme k dispozici jen pseudonáhodné generátory a čas potřebný k jejich výpočtu závisí na délce generovaných dat. Třetí problém je algoritmus pro operaci **DELETE**. Jak ukazuje praxe, většina algoritmu pro operaci **DELETE** porušuje rovnoměrné rozdělenídat (i když před jejich aplikací byly data rozdělany rovnoměrně). Následující dvě techniky randomizovaných binárních vyhledávacích stromů tyto problémy řeší, přičemž větší pozornost je věnována prvnímu z nich.

Treap – tento název vznikl kombinací slov tree a heap, protože uvedená datová struktura kombinuje binární vyhledávací stromy a haldy. Tato struktura byla původně určená k řešení následujícího problému: Je dáno totálně uspořádané univerzum $(U, \leq)$. Máme reprezentovat množinu $S \subseteq U$ a přitom každému prvku $s \in S$ je přiřazeno číslo $p(s)$ nazývané prioritou prvku s, které by mělo udávat četnost, s jakou se prvek může stát argumentem operace (prvek s vyšší prioritou má větší šanci být argumentem operace než prvek s menší prioritou). To motivuje způsob reprezentace množiny S. Ta je reprezentována pomocí binárního vyhledávacího stromu (vzhledem k totálnímu uspořádání univerza) a navíc, když vrchol v reprezentuje prvek $s \in S$ a syn u vrcholu v reprezentuje prvek t, pak $p(s) \geq p(t)$. Tato struktura reflektuje jak uspořádání dané univerzem U, tak srovnání dané prioritami. Následující tvrzení uvádí jednu její zajímavou vlastnost. Obecně platí, že když chceme reprezentovat množinu S, pak tato množina neurčuje jednoznačně binární vyhledávací strom, který ji reprezentuje. Zde je ale situace jiná.

Tvrzení 4.2. *Pro každou množinu S a pro navzájem různé priority $\{p(s) \mid s \in S\}$ existuje, až na izomorfismus, právě jedna datová struktura treap reprezentující S s prioritami $\{p(s) \mid s \in S\}$.*

Důkaz. Tvrzení dokážeme indukcí podle velikosti S. Když $|S| \leq 1$, pak je tvrzení zřejmé. Předpokládejme, že tvrzení platí pro $|S| < n$ a nechť $|S| = n$. Protože priority jsou navzájem různé, existuje právě jedno $s \in S$ takové, že $p(s) \geq p(t)$ pro každé $t \in S$. Pak z vlastností treap plyne, že s musí reprezentovat kořen binárního vyhledávacího stromu v treap reprezentujícím S. Zřejmě $S_1 = \{t \in S \mid t < s\}$ a $S_2 = \{t \in S \mid t > s\}$ splňují $|S_1|, |S_2| < n$, S_1 je reprezentována podstromem určeným levým synem kořene a S_2 je reprezentována podstromem určeným pravým synem kořene. Z indukčního předpokladu plyne jejich existence a jednoznačnost, a proto i treap reprezentující S existuje a je jednoznačně určen. □

Za prvé si všimněme, že důkaz Tvrzení 4.2 je vlastně důkaz existence treap v obecném případě. Tedy lze říct, že pro kaďá vstupní data treap existuje.

Dále si všimněme, že když priority prvků z S nejsou různé, pak může existovat více treap reprezentujících S, např. když existuje více prvků, které mají největší prioritu. Může se však stát, že priority nejsou různé, ale treap je jen jeden. Uvedeme zde bez důkazu nutnou a postačující podmínku na priority prvků množiny S, aby existoval právě jeden treap reprezentující S. Řekneme, že množina S s prioritami $\{p(s) \mid s \in S\}$ splňuje podmínku jednoznačnosti, když pro každé dva prvky $s, t \in S$ takové, že $p(s) = p(t)$ existuje $u \in S$ takové, že buď $s < u < t$ nebo $t < u < s$ a $p(s) < p(u)$. Pak platí:

Věta 4.3. *Množina S s prioritami $\{p(s) \mid s \in S\}$ splňuje podmínku jednoznačnosti, právě když existuje, až na izomorfismus, právě jeden treap reprezentující S.* □

Poznámka. Zkuste si toto tvrzenídokázat jako domacíúkol.

Operace **MEMBER**(x) v treap je realizována stejným algoritmem jako v klasických binárních vyhledávacích stromech. Nyní neformálně popíšeme algoritmy pro operace **INSERT** a **DELETE**.

Předpokládejme, že máme treap T reprezentující množinu S a že máme provést operaci **INSERT**(x). Když zapomeneme na priority, pak T je binární vyhledávací strom a můžeme v něm provést operaci **INSERT**(x) obvyklým způsobem. Ta buď nalezne vrchol t reprezentující x nebo nalezne list t takový, že $\lambda(t) < x < \pi(t)$, změní t na vnitřní vrchol reprezentující

x a přidá k němu dva syny, které budou listy. V prvním případě skončí i operace **INSERT** v treap. V druhém případě jsme získali 'téměř treap' – je porušena podmínka na monotonii priorit na cestách z kořene do listů pouze ve vrcholu t. S podobnou situací jsme se setkali v regulárních haldách, kde tento deficit řešila pomocná procedura **UP**. Tato procedura zde nefunguje, nemůžeme vyměnit prvky reprezentované vrcholem a jeho otcem, protože bychom porušili vlastnosti binárních vyhledávacích stromů. Místo výměny však můžeme použít rotaci na vrcholy t a otec(t). Rotace zachovává vlastnosti binárních vyhledávacích stromů. Protože i po ní bude platit

$$p(u) \leq p(v), \text{ když } v \text{ je předchůdcem } u \text{ a } u \neq t,$$

a vrchol t se posune blíže ke kořeni, dostáváme stejnou situaci jako v proceduře **UP** (jen výměnu prvků nahradíme rotací). Tím je dán algoritmus **INSERT** pro treap.

Předpokládejme, že máme provést operaci **DELETE**(x). Zde je situace komplikovanější, protože tato operace v binárních vyhledávacích stromech může změnit prvek ve vrcholu reprezentujícím x a tím porušit podmínku na priority. Proto budeme modifikovat algoritmus pro operaci **DELETE** v binárních vyhledávacích stromech následovně: Nejprve provedeme klasické vyhledání prvku x, pak pomocí rotací přesuneme vrchol u, který ho reprezentuje, tak, aby jeden jeho syn byl list, a nakonec smažeme vrchol u a jeho syna, který je list (vyžaduje to však větší multiplikatvní konstantu než klasický algoritmus). Všimněme si, že když máme treap T a provedeme pro fixovaný vrchol u posloupnost operací **Rotace**(u, v), kde v je syn u takový, že $p(v) \geq p(\text{bratr}(v)$, pak získáme binární vyhledávací strom, kde všechny vrcholy s výjimkou u splňují požadavek na priority a vrchol u se posunul do nižší hladiny. Když se pak stane, že syn u je list, můžeme smazat u a jeho syna, který je list, a dostaneme opět treap (reprezentující $S \setminus \{\text{key}(u)\}$). Tím je dán algoritmus pro **DELETE**. Nyní uvedeme formální popis obou algoritmů.

INSERT-TR$(x, p(x), T)$
t :=kořen stromu T
while key$(t) \neq x$ a t není list **do**
  if key$(t) < x$ **then** $t :=$ pravy(t) **else** $t :=$ levy(t) **endif**
enddo
if t je list **then**
  t změň na vnitřní vrchol, key$(t) := x$ s prioritou $p(x)$
  vytvoř dva nové syny vrcholu t, které budou listy
endif
while $p(\text{key}(\text{otec}(t))) < p(x)$ **do**
  Rotace(otec$(t), t)$
enddo

DELETE-TR(x, T)
t :=kořen stromu T
while key$(t) \neq x$ a t není list **do**
  if key$(t) < x$ **then** $t :=$ pravy(t) **else** $t :=$ levy(t) **endif**
enddo
if key$(t) = x$ **then**

```
  while žádný syn t není list do
    u_1 := levy(t), u_2 := pravy(t)
    if p(key(u_1)) ≤ p(key(u_2)) then
      Rotace(t, u_2)
    else
      Rotace(t, u_1)
    endif
  enddo
  if levy(t) je list then
    if t = levy(otec(t)) then
      levy(otec(t) := pravy(t)
    else
      pravy(otec(t)) := pravy(t)
    endif
    otec(pravy(t)) := otec(t)
    odstraň vrcholy t a levy(t)
  else
    if t = levy(otec(t)) then
      levy(otec(t)) := levy(t)
    else
      pravy(otec(t)) := levy(t)
    endif
    otec(levy(t)) := otec(t)
    odstraň vrcholy t a pravy(t)
  endif
endif
```

Algoritmus pro operaci **SPLIT** v treap je stejný jako pro klasické binární vyhledávací stromy. Algoritmus pro operaci **JOIN3** je oproti algoritmu pro klasické binární vyhledávací stromy doplněn rotacemi jako v operaci **DELETE** do té doby, než jsou splněny podmínky na priority pro treap. Formálně (předpokládáme, že každý prvek reprezentovaný ve stromě T_1 je menší než x a každý prvek reprezentovaný ve stromě T_2 je větší než x):

```
  JOIN3-TR(T_1, x, p(x), T_2)
vytvoř nový vrchol t, key(t) := x s prioritou p(x)
levy(t) :=kořen T_1, pravy(t) :=kořen T_2
while existuje syn u vrcholu t, že p(x) < p(key(u)) do
  u_1 := levy(t), u_2 := pravy(t)
  if p(key(u_1)) ≤ p(key(u_2)) then
    Rotace(t, u_2)
  else
    Rotace(t, u_1)
  endif
enddo
```

Věta 4.4. *Algoritmy* **INSERT-TR**, **DELETE-TR** *a* **JOIN3-TR** *korektně implementují operace* **INSERT**, **DELETE** *a* **JOIN3** *pro treap. Algoritmus realizující* **SPLIT** *v binárních vyhledávacích stromech korektně implementuje operaci* **SPLIT** *v treap. Tyto algoritmy vyžadují čas $O(v(T))$, kde T je binární vyhledávací strom v treap.*

Důkaz. Všimněme si, že v algoritmech **INSERT-TR** a **DELETE-TR** platí, že když t je vrchol reprezentující x, pak pro každý vrchol v stromu T různý od t je $p(\text{key}(\text{otec}(v))) \geq p(\text{key}(v))$. Odtud plyne, že po skončení algoritmů **INSERT-TR** a **DELETE-TR** je splněna podmínka na priority. Protože T je vždy binární vyhledávací strom, jsou tyto algoritmy korektní. Důkaz korektnosti **JOIN3-TR** je analogický, pouze se uvažuje duální formulace předchozí podmínky (tj. pro každý vrchol v stromu T různý od t a pro každého syna u vrcholu v platí $p(\text{key}(v)) \geq p(\text{key}(u))$). Korektnost algoritmu **SPLIT** pro binární vyhledávací stromy i pro treap plyne z faktu, že když v je následník vrcholu u ve stromu T, pak $p(\text{key}(u)) \geq p(\text{key}(v))$. Odhad časové složitosti je stejný jako pro binární vyhledávací stromy. □

Uvedeme ještě tři pozorování týkající se treap.

Aplikujme posloupnost $\{\mathbf{INSERT}(x_i) \mid i = 1, 2, \ldots, n\}$ na prázdný binární vyhledávací strom. Když $p(x_i) \geq p(x_j)$ pro $1 \leq i \leq j \leq n$, pak vzniklý binární vyhledávací strom je zároveň treap pro $S = \{x_i \mid i = 1, 2, \ldots, n\}$ s prioritami $\{p(x_i) \mid i = 1, 2, \ldots, n\}$.

Když T je treap, který nereprezentuje prvek x, a máme provést operaci **SPLIT**(x, T), pak ji můžeme realizovat takto:

SPLIT(x, T)
INSERT$(x, +\infty, T)$
$T_1 :=$ podstrom T určený levým synem kořene
$T_2 =$ podstrom T určený pravým synem kořene

Toto pozorování tedy dává do vztahu operace **SPLIT** a **INSERT**, další zase spojuje operace **INSERT** a **JOIN**. Operaci **JOIN2**(T_1, T_2) lze implementovat takto:

JOIN2(T_1, T_2)
$T :=$**JOIN3**$(T_1, a, +\infty, T_2)$
Poznámka: a je 'dummy' prvek mezi největším prvkem reprezentovaným T_1 a nejmenším prvkem reprezentovaným T_2
DELETE(a, T)

Randomizovaný treap vznikne, když priority nejsou zadané vstupníúlohou, ale náhodně nezávisle zvolená čísla z rovnoměrného rozdělení na intervalu [0,1] v algoritmu **INSERT**. Pak

Věta 4.5. *Pro každou množinu $S \subseteq U$ mají binární vyhledávací stromy reprezentující S jako randomizovaný treap rovnoměrné rozdělení.* □

Důkaz tohoto tvrzení přesahuje rámec této přednášky, proto ho neuvádíme. Jako důsledek dostaneme:

Důsledek 4.6. *Operace* **MEMBER***,* **INSERT***,* **DELETE***,* **SPLIT** *a* **JOIN3** *lze implementovat v treap tak, že jejich očekávaný čas je* $O(\log n)$*, kde* n *je velikost reprezentované množiny. Očekávaný čas pro realizaci posloupnosti* n*-operací na prázdný treap je* $O(n \log n)$ *i když rotace provedená na podstrom o velikosti* m *vyžaduje čas* $O(m)$*. Očekávaný počet rotací v operacích* **INSERT** *a* **DELETE** *je nejvýše* 2*.*

Důkaz. Důkaz prvního tvrzení plyne z Vět 4.1 a 4.5. Důkaz druhého a třetího tvrzení je netriviální modifikací tvrzení o počtu a rozložení operací **Štěpení**, **Spojení** a **Přesun** pro (a,b)-stromy. Jeho přesné znění vynecháváme. □

Je nutné si uvědomit, co vlastně říká Důsledek 4.6. Pro každou posloupnost $\mathcal{P}$ operací **MEMBER**, **INSERT**, **DELETE**, **JOIN3** a **SPLIT** aplikovanou opakovaně na prázdný randomizovaný treap, spotřebuje její provedení různé množství času v závislosti na volbě priorit. Důsledek 4.6 říká, že pro každou takovou posloupnost $\mathcal{P}$ každá její operace má očekávaný čas $O(\log n)$, když volba priorit má rovnoměrné rozdělení.

Treap poprvé definoval Vuillemin v roce 1980, ale používal pro něj název kartézské stromy. Název treap zavedl E. McCreight pro jinou datovou strukturu, ale později její název změnil. Pro zde popsanou strukturu použili poprvé název treap Aragon a Seidel v roce 1989, kdy také definovali randomizované treaps.

Nyní si ukážeme si další pravděpodobnostní variantu binárních vyhledávacích stromů, která je založena na jiné ideji. Nejprve základní definice. Řekneme, že operace vytvářejí randomizovaný binární vyhledávací strom T reprezentující množinu S, když buď $S = \emptyset$ nebo $|S| = n \geq 1$ a pro každé $i = 0, 1, \dots, n-1$ je pravděpodobnost, že podstrom určený levým synem kořene T reprezentuje množinu velikosti i, rovna $\frac{1}{n}$, a podstromy určené syny kořene jsou randomizované binární vyhledávací stromy. Popíšeme algoritmy pro operace **INSERT**, **DELETE** a **JOIN3**, jejichž výsledkem je opět randomizovaný binární vyhledávací strom. Algoritmy pro operace **MEMBER** a **SPLIT** budou stejné jako pro binární vyhledávací stromy (bez vyvažování). Datová struktura binárního vyhledávacího stromu spolu s těmito algoritmy se nazývá struktura randomizovaných binárních vyhledávacích stromů.

Nejprve popíšeme algoritmus **INSERT-RBVS** pro operaci **INSERT**. Mějme randomizovaný binární vyhledávací strom reprezentující množinu S a proveďme operaci **INSERT-RBVS**(x). Nejprve pomocí operace **MEMBER**(x) zjistíme, zda $x \in S$. Když $x \notin S$, pak použijeme podproceduru **Vloz**(x,T), která volá rekurzivně sama sebe. Když T reprezentuje $n-$prvkovou množinu, pak tato podprocedura náhodně zvolí číslo $r \in \{0, 1, \dots, n\}$ s rovnoměrným rozdělením. Když $r = n$, pak zavolá pomocnou proceduru **Root-insert**(x,T). Když $r \neq n$ a pro kořen t stromu je $\text{key}(t) > x$, pak kořen stromu, který vznikne rekurzivním voláním **Vloz**(x,T_1), kde T_1 je podstrom levého syna t, bude levým synem t. Když $\text{key}(t) < x$, pak kořen stromu, který vznikne rekurzivním voláním **Vloz**(x,T_1), kde T_1 je podstrom pravého syna t, bude pravým synem t. Jak plyne z algoritmu, pro každý vrchol v musíme znát proměnnou $\tau(v)$, která udává velikost množiny reprezentované podstromem určeným vrcholem v (tedy pro list l je $\tau(l) = 0$). Tato hodnota bude spojena s vrcholem v a podprocedura **Vloz**(x,T) zvětší hodnotu $\tau(t)$ o 1. Podprocedura **Root-insert**(x,T) zavolá operaci **SPLIT**(x,T), která vytvoří stromy T_1 a T_2. Procedura pak vytvoří nový vrchol t reprezentující prvek x. Levý syn vrcholu t bude kořen stromu T_1 a pravý syn t bude kořen stromu T_2.

INSERT-RBVS(x,T)
MEMBER(x,T)
if x není reprezentován **then** **Vloz**(x,T) **endif**

Vloz(x,T)
t :=kořen T, $n := \tau(t)$
r :=náhodně zvolené číslo v rozsahu $0,1,\dots,n$ s rovnoměrným rozdělením
if $r=n$ **then**
  Root-insert(x,T)
else
  if $x <$ key(t) **then**
    T_1 :=podstrom určený levým synem t
    levy(t) :=kořen **Vloz**(x,T_1)
  else
    T_1 :=podstrom určený pravým synem t
    pravy(t) :=kořen **Vloz**(x,T_1)
  endif
  $\tau(t) := \tau(t)+1$
endif

Root-insert(x,T)
if T je jednoprvkový strom **then**
  $T_1,T_2 :=$ jednoprvkové stromy
else
  (T_1,T_2) :=**SPLIT**(x,T)
endif
vytvoř vrchol t, levy(t) :=kořen T_1, pravy(t) :=kořen T_2
key$(t) := x$, $\tau(t) := \tau(\text{levy}(t)) + \tau(\text{pravy}(t)) + 1$
Výstup: T je strom s kořenem t

Zde, jak jsme již uvedli, je **SPLIT** klasický algoritmus pro binární vyhledávací stromy bez vyvažování (ale operace **JOIN3** se provede jako operace **JOIN2** pro randomizované binární vyhledávací stromy následovaná operaci **INSERT** pro randomizované binární vyhledávací stromy). Korektnost algoritmu je založena na následujícím lemmatu:

Lemma 4.7. *Když S je množina reprezentovaná randomizovaným binárním vyhledávacím stromem T a když (T_1,T_2) =***SPLIT**(x,T) *pro $x \in U$, pak T_1 je randomizovaný binární vyhledávací strom reprezentující množinu $S_1 = \{s \in S \mid s < x\}$ a T_2 je randomizovaný binární vyhledávací strom reprezentující množinu $S_2 = \{s \in S \mid s > x\}$.*

Věta 4.8. *Když T je randomizovaný binární vyhledávací strom, pak výsledkem algoritmu* **INSERT-RBVS**(x,T) *je opět randomizovaný binární vyhledávací strom.*

Důkazy těchto tvrzení vynecháváme.

Nyní neformálně popíšeme algoritmus **DELETE-RBVS**(x,T) pro operaci **DELETE**. Předpokládáme, že T je randomizovaný binární vyhledávací strom. Budeme používat pomocnou proměnnou *odst*, abychom věděli, zda jsme skutečně odstranili nějaký prvek (kvůli

aktualizaci proměnných $\tau(t)$). Nejprve nastavíme hodnotu *odst* na $false$ (ještě jsme žádný prvek neodstranili). Když T je jednoprvkový strom, to znamená, že reprezentuje prázdnou množinu, tak končíme (není co odstranit). V opačném případě označíme t kořen stromu. Když $\text{key}(t) = x$, pak nastavíme $odst = true$. Dále T_1 bude podstrom určený levým synem t, T_2 bude podstrom určený pravým synem t a provedeme operaci **JOIN2** algoritmem **JOIN2-RBVS**(T_1, T_2) (tento algoritmus popíšeme později). Když $\text{key}(t) > x$, pak označíme T' podstrom určený levým synem t a nový levý syn t pak bude kořen stromu **DELETE-RBVS**(x, T'). Když $\text{key}(t) < x$, pak označíme T' podstrom určený pravým synem t a nový pravý syn t pak bude kořen stromu **DELETE-RBVS**(x, T'). Když *odst* bude *true*, pak ještě zmenšíme $\tau(t)$ o 1.

DELETE-RBVS(x, T)
odst := $false$
if T není jednoprvkový strom **then**
  t :=kořen stromu T
  if $\text{key}(t) = x$ **then**
    odst := *true*
    T_1 :=podstrom T určený levým synem t
    T_2 :=podstrom T určený pravým synem t
    T :=**JOIN2-RBVS**(T_1, T_2)
  endif
  if $\text{key}(t) < x$ **then**
    T' :=podstrom T určený pravým synem t
    pravy(t) :=kořen **DELETE-RBVS**(x, T')
  else
    T' :=podstrom T určený levým synem t
    levy(t) :=kořen **DELETE-RBVS**(x, T')
  endif
  if *odst* **then** $\tau(t) := \tau(t) - 1$ **endif**
endif

Na závěr popíšeme algoritmus **JOIN2-RBVS**(T_1, T_2) pro operaci **JOIN2**. Předpokládáme, že každý prvek reprezentovaný randomizovaným binárním vyhledávacím stromem T_1 je menší než každý prvek reprezentovaný randomizovaným binárním vyhledávacím stromem T_2. Pomocí proměnné τ zjistíme velikost reprezentovaných množin – n_i bude velikost množiny reprezentované stromem T_i pro $i = 1, 2$. Když $n_1 = 0$, pak výsledek operace je T_2, když $n_2 = 0$, pak výsledek je T_1. Když $n_1 \neq 0 \neq n_2$, zvolíme náhodně číslo r z oboru $0, 1, \ldots, n_1 + n_2 - 1$ s rovnoměrným rozdělením. Když $r < n_1$, pak označme T_1' podstrom T_1 určený pravým synem kořene T_1. Výsledný strom získáme tak, že ve stromě T_1 nahradíme pravého syna kořene kořenem stromu **JOIN2-RBVS**(T_1', T_2). Když $r \geq n_1$, pak označme T_2' podstrom T_2 určený levým synem kořene T_2. Výsledný strom získáme tak, že ve stromě T_2 nahradíme levého syna kořene kořenem stromu **JOIN2-RBVS**(T_1, T_2'). Na závěr položíme τ kořene výsledného stromu rovno $n_1 + n_2$.

JOIN2-RBVS(T_1, T_2)
for every $i = 1, 2$ **do**

$n_i := \tau$(kořene stromu T_i
enddo
if $n_1 = 0$ **then** $T := T_2$ **stop endif**
if $n_2 = 0$ **then** $T := T_1$ **stop endif**
r :=náhodně zvolené číslo v rozsahu $0, 1, \dots, n_1 + n_2 - 1$ s rovnoměrným rozdělením
if $r < n_1$ **then**
t :=kořen T_1
T'_1 :=podstrom T_1 určený pravým synem t
T'_1 :=**JOIN2-RBVS**(T'_1, T_2), pravy(t) :=kořen T'_1, $T := T_1$
else
t :=kořen T_2
T'_2 :=podstrom T_2 určený levým synem t
T'_2 :=**JOIN2-RBVS**(T_1, T'_2), levy(t) :=kořen T'_2, $T := T_2$
endif τ(kořene T) $:= n_1 + n_2$

Věta 4.9. *Když T_1 a T_2 jsou randomizované binární vyhledávací stromy reprezentující množiny S_1 a S_2 a když $\max S_1 < \min S_2$, pak* **JOIN2-RBVS**(T_1, T_2) *vytvoří randomizovaný binární vyhledávací strom reprezentující množinu $S_1 \cup S_2$. Když T je randomizovaný binární vyhledávací strom, pak* **DELETE-RBVS**(x, T) *vytvoří opět randomizovaný binární vyhledávací strom.*

Důkaz neuvádíme.

Věta 4.10. *Za předpokladu korektní volby náhodných čísel je očekávaný čas operací* **MEMBER**, **INSERT**, **DELETE**, **JOIN2** *a* **SPLIT** *v randomizovaných binárních vyhledávacích stromech $O(\log n)$, kde n je velikost reprezentované množiny.*

Důkaz plyne z Vět 4.1, 4.8 a 4.9. □

Zde je třeba chápat znění Věty 4.10 stejně, jak je uvedeno v odstavci za Důsledkem 4.6. Čas operace závisí na náhodně zvoleném čísle a očekávaný čas je počítán přes všechny volby těchto čísel.

Algoritmy pro randomizované binární vyhledávací stromy prezentovali Martínez a Roura v roce 1997 a ukázali, že jimi navržené algoritmy zachovávají randomizovanost. Postup pro operaci **INSERT** byl znám dříve, viz např. slavná monografie od Knutha 1973. Hlavní přínos Martíneze a Roury se týkal operace **DELETE**. Operace **DELETE**, které byly známy dříve (např. Hibbardův algoritmus z roku 1962), nezachovávaly rovnoměrné rozdělení binárních vyhledávacích stromů reprezentujících danou množinu, jak si povšiml Knott 1975 (to znamená, že po provedení operace **DELETE** se binární vyhledávací stromy nevyskytují se stejnou četností). V roce 1990 zavedl Pugh další náhodnou strukturu, tzv. skip-list. Rozsah přednášky neumožňuje se o ní zmínit.

5. Optimální binární vyhledávací stromy

V této sekci budeme řešit problém konstrukce optimálního binárního vyhledávacího stromu. Nechť U je totálně uspořádané univerzum a nechť $S = \{x_1 < x_2 < \dots < x_n\}$ je daná podmnožina U. Pro každý prvek $x_i \in S$ je dáno jeho ohodnocení reálným číslem α_i, $i = 1, 2, \dots, n$, a pro každý interval (x_j, x_{j+1}) univerza U je dáno jeho ohodnocení

reálným číslem β_j, $j = 0, 1, \ldots, n$ (kde $x_0 = -\infty$ je menší než všechny prvky univerza U a $x_{n+1} = \infty$ je větší než všechny prvky univerza U). Naším úkolem je zkonstruovat binární vyhledávací strom T reprezentující množinu S, který má nejmenší ohodnocení $\text{hod}(T) = \sum_{i=1}^n \alpha_i(a_i + 1) + \sum_{j=0}^n \beta_j b_j$, kde a_i je hloubka vrcholu reprezentujícího prvek x_i a b_j je hloubka listu reprezentujícího interval (x_j, x_{j+1}). Takový strom T nazveme optimálním binárním vyhledávacím stromem pro S a $\{\alpha_i \mid i = 1, 2, \ldots, n\}$ a $\{\beta_j \mid j = 0, 1, \ldots, n\}$.

Na první pohled je vidět, že konstrukce optimálního binárního vyhledávacího stromu patří do teorie optimalizace, ale zároveň řeší i následující problém z datových struktur. Předpokládejme, že máme danou množinu $S \subseteq U$, kde $|S| = n$, pravděpodobnosti $\{\alpha_i \mid i = 1, 2, \ldots, n\}$ a $\{\beta_j \mid j = 0, 1, \ldots, n\}$ a že uživatel zadává operace **MEMBER**(y) tak, že $\text{Prob}(y = x_i) = \alpha_i$ pro každé $i = 1, 2, \ldots, n$ a $\text{Prob}(y \in (x_j, x_{j+1})) = \beta_j$ pro každé $j = 0, 1, \ldots, n$. Chceme nalézt binární vyhledávací strom reprezentující S, který by minimalizoval očekávaný čas operace **MEMBER**. Protože očekávaný čas této operace pro binární vyhledávací strom reprezentující S je $\sum_{i=1}^n \alpha_i(a_i+1)+\sum_{j=0}^n \beta_j b_j = \text{hod}(T)$, chceme vlastně zkonstruovat optimální binární vyhledávací strom pro S, $\{\alpha_i \mid i = 1, 2, \ldots, n\}$ a $\{\beta_j \mid j = 0, 1, \ldots, n\}$. Podobný problém řeší i datová struktura treap, když priorita x_i je rovna α_i pro každé $i = 1, 2, \ldots, n$. Uvědomme si rozdíly v zadání obou úloh a jejich řešení. Treap řeší úlohu, kdy jsou dány prvky reprezentované množiny a pravděpodobnosti přístupu k nim, a přitom připouští jak operaci **MEMBER**, tak aktualizační operace **INSERT** a **DELETE**. Na druhé straně nejsou známy (nebo se ignorují) pravděpodobnosti přístupu k prvkům, které nejsou v reprezentované množině. Treap tedy v jistém smyslu řeší obecnější úlohu. Optimální binární vyhledávací strom řeší speciálnější problém, ale pro zadanou množinu S a pravděpodobnosti $\{\alpha_i \mid i = 1, 2, \ldots, n\}$ a $\{\beta_i \mid i = 0, 1, \ldots, n\}$ poskytne výrazně řešení, které v konkrétním případě může být výrazně lepší.

Můžeme se ptát, proč jsme tedy tento problém formulovali takto obecně. Je to proto, že v našem speciálním zadání jsou $\alpha_i, \beta_j \in <0, 1>$ pro všechna $i = 1, 2, \ldots, n$ a $j = 0, 1, \ldots, n$ a musí platit $\sum_{i=1}^n \alpha_i + \sum_{j=0}^n \beta_j = 1$. Zvláště druhá podmínka klade podstatná technická omezení na způsob řešení, která algoritmus dělají složitějším, než je nutné. Z toho důvodu je výhodnější řešit obecnější problém.

K řešení použijeme metodu dynamického programování. Mějme $1 \leq i \leq j \leq n$ a nechť úlohou $\mathcal{U}_{i,j}$ je nalezení optimálního binárního vyhledávacího stromu pro $S_{i,j} = \{x_k \mid i \leq k \leq j\}$, $\{\alpha_k \mid i \leq k \leq j\}$ a $\{\beta_k \mid i - 1 \leq k \leq j\}$ (tj. ohodnocení intervalu $(-\infty, x_i)$ je stejné jako ohodnocení intervalu (x_{i-1}, x_i) a totéž platí pro intervaly (x_j, ∞) a (x_j, x_{j+1})). Pak ohodnocení stromu T reprezentujícího množinu $S_{i,j}$ je $\text{hod}(T) = \sum_{k=i}^j \alpha_k(a_k + 1) + \sum_{k=i-1}^j \beta_k b_k$, kde a_k je hloubka vrcholu stromu T reprezentujícího prvek x_k pro $k = i, i + 1, \ldots, j$ a b_k je hloubka listu stromu T reprezentujícího interval (x_k, x_{k+1}) (v tomto případě je $x_{i-1} = -\infty$ a $x_{j+1} = \infty$). Všimněme si, že naše původní úloha je vlastně úloha $\mathcal{U}_{1,n}$.

Nejprve si dokážeme několik jednoduchých tvrzení, která tvoří teoretický základ algoritmu.

Lemma 5.1. *Nechť T je strom reprezentující množinu $S_{i,j}$, jehož kořen reprezentuje prvek x_k, a T_l a T_r jsou podstromy T určené levým a pravým synem kořene T. Pak T_l reprezentuje množinu $S_{i,k-1}$, T_r reprezentuje množinu $S_{k+1,j}$ a*

$$\text{hod}(T) = \text{hod}(T_l) + \text{hod}(T_r) + \sum_{k=i}^{j} \alpha_k + \sum_{k=i-1}^{j} \beta_k.$$

Důkaz. Z definice binárního vyhledávacího stromu a ze zadání množiny $S_{i,j}$ plyne, že když $i \le q < k$, pak x_q je reprezentován vrcholem v T_l, když $k < q \le j$, pak x_q je reprezentován vrcholem v T_r. Když v je vrchol stromu T_l (resp. T_r), pak jeho hloubka ve stromu T_l (resp. T_r) je o 1 menší než hloubka ve stromu T a kořen má hloubku 0. Odtud dosazením a jednoduchou úpravou plyne tvrzení. □

Lemma 5.2. *Nechť T je optimální binární vyhledávací strom pro $S_{i,j}$. Když kořen T reprezentuje prvek x_k a když T_l a T_r jsou podstromy T určené levým a pravým synem kořene, pak T_l je optimální binární vyhledávací strom pro $S_{i,k-1}$ a T_r je optimální binární vyhledávací strom pro $S_{k+1,j}$.*

Poznámka. Všimněme si, že jednoprvkový strom je optimální binární vyhledávací strom pro úlohu $\mathcal{U}_{i,i-1}$ pro každé i, $1 \le i \le n$.

Důkaz. Předpokládejme, že T_l není optimální binární vyhledávací strom pro $S_{i,k-1}$. Protože z konečnosti $S_{i,k-1}$ plyne existence optimálního řešení pro $\mathcal{U}_{i,k-1}$, předpokládejme, že optimální binární vyhledávací strom pro $S_{i,k-1}$ je T'_l. Pak $\mathrm{hod}(T'_l) < \mathrm{hod}(T_l)$. Vytvořme strom T' tak, že nahradíme levého syna kořene T kořenem stromu T'_l. Z Lemmatu 5.1 a z vlastností binárních vyhledávacích stromů plyne, že T' je binární vyhledávací strom pro $S_{i,j}$ a $\mathrm{hod}(T') < \mathrm{hod}(T)$. To je spor s optimalitou T, a proto T_l je optimální binární vyhledávací strom pro $S_{i,k-1}$. Ze symetrie plyne, že T_r je optimální binární vyhledávací strom pro $S_{k+1,j}$. □

Lemma 5.3. *Nechť $T_{i,k}$ pro $i-1 \le k < j$ je optimální binární vyhledávací strom pro $S_{i,k}$ a nechť $T_{k,j}$ pro $i < k \le j+1$ je optimální binární vyhledávací strom pro $S_{k,j}$. Když k je takové číslo, že $i \le k \le j$ a $\mathrm{hod}(T_{i,k-1}) + \mathrm{hod}(T_{k+1,j}) \le \mathrm{hod}(T_{i,l-1}) + \mathrm{hod}(T_{l+1,j})$ pro každé l splňující $i \le l \le j$, pak strom T, jehož kořen reprezentuje prvek x_k, levým synem kořene T je kořen stromu $T_{i,k-1}$ a pravým synem kořene T je kořen stromu $T_{k+1,j}$, je optimální binární vyhledávací strom pro $S_{i,j}$.*

Důkaz. Z Lemmat 5.1 a 5.2 plyne, že když T je řešením úlohy $\mathcal{U}_{i,j}$ a jeho kořen reprezentuje prvek x_k, pak $\mathrm{hod}(T_{i,k-1}) + \mathrm{hod}(T_{k+1,j}) \le \mathrm{hod}(T_{i,l-1}) + \mathrm{hod}(T_{l+1,j})$ pro každé l takové, že $i \le l \le j$. Z Lemmatu 5.1 plyne opačná implikace. □

Lemmata 5.1, 5.2 a 5.3 dávají základní ideu algoritmu. Jeho výstupem budou dvě horní trojúhelníkové matice: R typu $\{1, 2, \dots, n\} \times \{1, 2, \dots, n\}$ a H typu $\{1, \dots, n\} \times \{0, 1, \dots, n\}$, kde pro $1 \le i \le j \le n$ je $R(i,j)$ index prvku, který je reprezentován kořenem (některého) optimálního binárního vyhledávacího stromu $T_{i,j}$ pro $S_{i,j}$, a $H(i,j) = \mathrm{hod}(T_{i,j})$. Nejprve ale popíšeme jednoduchý algoritmus, který z nalezené matice R zkonstruuje požadovaný optimální binární vyhledávací strom. Algoritmus zavolá rekurzivní proceduru **Kon** s parametry 1 a n. Procedura **Kon** s parametry i a j pro $i \le j$ nalezne kořen stromu řešícího úlohu $\mathcal{U}_{i,j}$. Tento kořen reprezentuje prvek x_k, kde $k = R(i,j)$. Podstrom určený levým synem kořene je řešením úlohy $\mathcal{U}_{i,k-1}$, proto se na jeho vytvoření zavolá **Kon** s parametry i a $k-1$. Analogicky podstrom určený pravým synem kořene je řešením úlohy $\mathcal{U}_{k+1,j}$, a proto se zavolá **Kon** s parametry $k+1$ a j. Když $i > j$, pak výsledkem je jednoprvkový strom (tedy list) a výpočet končí.

Konstr(R)
$T :=$**Kon**$(1, n)$
Výstup: T optimální binární vyhledávací strom pro S

Kon(i,j)
if $j < i$ **then**
Výstup jednoprvkový strom
else
$k := R(i,j)$, vytvoř vrchol v reprezentující prvek x_k
levy(v) :=kořen **Kon**$(i,k-1)$, pravy(v) :=kořen **Kon**$(k+1,j)$
endif

Z Lemmatu 5.3 je okamžitě vidět

Tvrzení 5.4. *Když pro každé* $1 \leq i \leq j \leq n$ *je* $R(i,j)$ *index prvku, který je reprezentován kořenem optimálního binárního vyhledávacího stromu pro* $S_{i,j}$*, pak algoritmus* **Konstr** *zkonstruuje v čase* $O(n)$ *optimální binární vyhledávací strom pro naši úlohu.*

Nyní popíšeme základní algoritmus pro konstrukci optimálního binárního vyhledávacího stromu. Nejprve systematickým průchodem přes i v rozsahu $1,2,\dots,n$ a přes j v rozsahu $i,i+1,\dots,n$ vypočteme v čase $O(n^2)$ hodnoty $w_{i,j} = \sum_{k=i}^{j} \alpha_k + \sum_{k=i-1}^{j} \beta_k$. Dále inicializujeme nulové matice H a R. Pro $j = 0,1,\dots,n-1$ označme množinu dvojic $\{(i,i+j) \mid i = 1,2\dots,n-j\}$ jako j-tou diagonálu matice H, resp. R. Náš postup bude následující: když pro každé $l = 0,1,\dots,j-1$ a pro každou dvojici $(i,i+l)$ v l-té diagonále známe $H(i,i+l)$, pak spočítáme $H(i,i+j)$ a $R(i,i+j)$ pro každou dvojici $(i,i+j)$ v j-té diagonále. Z Lemmatu 5.3 plyne, že musíme nalézt k takové, že $i \leq k \leq i+j$ a $H(i,k-1) + H(k+1,i+j) \leq H(i,l-1) + H(l+1,i+j)$ pro každé l takové, že $i \leq l \leq i+j$. Protože pro každé takové l patří $(i,l-1)$ do j'-té diagonály pro nějaké $j' \in \{0,1,\dots,j-1\}$ a stejně tak $(l+1,i+j)$ patří do j''-té diagonály pro nějaké $j'' \in \{0,1,\dots,j-1\}$, můžeme takové k nalézt. Pak stačí položit $R(i,i+j) = k$ a $H(i,i+j) = H(i,k-1) + H(k+1,i+j) + w_{i,i+j}$ (viz Lemma 5.3) a jsme hotovi. Přesně tento postup provádí algoritmus **Optim** s tím doplněním, že do $R(i,i+j)$ dává největší k s požadovanou vlastností (obecně k není určeno jednoznačně). Nyní uvedeme formální popis algoritmu.

OPTIM$(S = \{x_1,x_2,\dots,x_n\}, \{\alpha_i \mid i = 1,2,\dots,n\}, \{\beta_j \mid j = 0,1,\dots,n\})$
for every $i = 1,2,\dots,n$ **do**
$w_{i,i} = \alpha_i + \beta_{i-1} + \beta_i$
for every $j = i+1,i+2,\dots,n$ **do**
$w_{i,j} = w_{i,j-1} + \alpha_j + \beta_j$
enddo
enddo
$H,R :=$ nulové matice
for every $i = 1,2,\dots,n$ **do**
$H_{i,i} := w_{i,i}$, $R_{i,i} := i$
enddo
$k := 1$
while $k < n$ **do**
$i := 1$
while $i + k \leq n$ **do**
$M := H(i,i+k-1)$, $m := i+k$, $l := m-1$
while $l \geq i$ **do**

```
        if H(i,l-1) + H(l+1,i+k) < M then
          M := H(i,l-1) + H(l+1,i+k), m := l
        endif
        l := l-1
      enddo
      H(i,i+k) := M + w_{i,i+k}, R(i,i+k) := m, i := i+1
    enddo
    k := k+1
enddo
```

První cyklus systematickým postupem počítá $w_{i,j} = \sum_{k=i}^{j} \alpha_k + \sum_{k=i-1}^{j} \beta_k$ v čase $O(n^2)$. Druhý cyklus inicializuje matice H a R v čase $O(n^2)$. Třetí cyklus počítá prvky matic H a R. Výpočet se provádí po diagonálách. Vnější cyklus určuje diagonálu, prostřední cyklus určuje prvek diagonály a vnitřní cyklus pro dvojici $(i, i+k)$ hledá minimum požadované Lemmatem 5.3. Po jeho nalezení (velikost minima je uložena v proměnné M, index prvku, který určuje minimum, je v proměnné m, a začíná se s $m = i + k$) se určí podle Lemmatu 5.3 hodnoty $H(i,j)$ a $R(i,j)$. Každý cyklus se opakuje až n-krát, proto celý algoritmus vyžaduje čas $O(n^3)$. Následující věta shrnuje uvedená fakta:

Věta 5.5. *Algoritmus* **OPTIM** *konstruuje optimální binární vyhledávací strom v čase $O(n^3)$ a používá $O(n^2)$ paměti, kde n je velikost reprezentované množiny S.*

Důkaz. Korektnost algoritmu plyne z Lemmatu 5.3. Časový odhad plyne z analýzy uvedené před větou. Zřejmě matice H a R vyžadují $O(n^2)$ prostoru. □

Algoritmus ilustruje následující příklad. Mějme množinu $S = \{x_1 < x_2 < x_3 < x_4\}$, $\alpha_1 = 1$, $\alpha_2 = 3$, $\alpha_3 = 3$, $\alpha_4 = 0$, $\beta_0 = 4$, $\beta_1 = 0$, $\beta_2 = 0$, $\beta_3 = 3$, $\beta_4 = 10$. Algoritmus pak spočítá $w_{1,1} = 5$, $w_{1,2} = 8$, $w_{1,3} = 14$, $w_{1,4} = 24$, $w_{2,2} = 3$, $w_{2,3} = 9$, $w_{2,4} = 19$, $w_{3,3} = 6$, $w_{3,4} = 16$, $w_{4,4} = 13$ a zkonstruuje

$$H = \begin{pmatrix} 0 & 5 & 11 & 25 & 48 \\ 0 & 0 & 3 & 12 & 31 \\ 0 & 0 & 0 & 6 & 22 \\ 0 & 0 & 0 & 0 & 13 \end{pmatrix} \qquad R = \begin{pmatrix} 1 & 1 & 3 & 3 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & 4 \end{pmatrix}.$$

Prezentovaný algoritmus vyžaduje sice jen polynomiální čas, ale stupeň polynomu (třetí) výrazně omezuje velikost vstupu pro praktické použití. Proto se hledaly rychlejší algoritmy. Experimenty s algoritmem ukázaly, že vždy platí (pokud index hledaného maxima v třetím cyklu je vždy největšímožný)

$$R(i, j-1) \leq R(i,j) \leq R(i+1, j).$$

Za tohoto předpokladu můžeme upravit tyto tři cykly, které jsou vnořené do sebe takto:

```
k := 1
while k < n do
  i := 1
```

```
    while i + k < n do
      m := R(i+1, i+k), M := H(i, m-1) + H(m+1, i+k), l := m-1
      while l ≥ R(i, i+k-1) do
        if H(i, l-1) + H(l+1, i+k) < M then
          M := H(i, l-1) + H(l+1, i+k), m := l
        endif
        l := l-1
      enddo
      H(i, i+k) := M + w_{i,i+k}, R(i, i+k) := m, i := i+1
    enddo
    k := k+1
enddo
```

Pak vnitřní cyklus vyžaduje čas $O(R(i+1,i+k)-R(i,i+k-1))$, a proto prostřední cyklus vyžaduje čas $O(\sum_{i=1}^{n-k}(R(i+1,i+k)-R(i,i+k-1))=O(n)$. Odtud plyne

Tvrzení 5.6. *Když pro každé $1 \leq i \leq j \leq n$ platí*

$$R(i,j-1) \leq R(i,j) \leq R(i+1,j),$$

pak vylepšená verze algoritmu **OPTIM** *konstruuje optimální binární vyhledávací strom v čase $O(n^2)$.*

Snaha dokázat platnost předpokladu Tvrzení 5.6 vedla k vyšetřování vztahu mezi konstrukcí optimálního binárního vyhledávacího stromu a kvadratickým programováním. Nejprve uvedeme definici kvadratického programování.

Definice. Nechť pro $0 \leq i < j \leq n$ je dáno číslo $w_{i,j}$. Pro $1 \leq i \leq j \leq n$ definujme rekurzivně čísla $c_{i,j}$ takto:

$$c_{i,i}=0 \text{ a pro } i<j \text{ položme } c_{i,j}=w_{i,j}+\min\{c_{i,k-1}+c_{k,j} \mid k=i+1,i+2,\dots,j\}.$$

Řešením úlohy kvadratického programování je nalezení souboru čísel $\{c_{i,j} \mid 1 \leq i \leq j \leq n\}$ pro daný soubor čísel $\{w_{i,j} \mid 1 \leq i < j \leq n\}$.

Označme $r_{i,j}$ největší přirozené číslo takové, že $i < r_{i,j} \leq j$ a $c_{i,r_{i,j}-1}+c_{r_{i,j},j} \leq c_{i,l-1}+c_{l,j}$ pro každé l takové, že $i < l \leq j$. Předpokládejme, že máme zkonstruovat optimální binární vyhledávací strom pro množinu $S=\{x_1 < x_2 < \cdots < x_n\}$, $\{\alpha_i \mid i=1,2,\dots,n\}$ a $\{\beta_j \mid j=0,1,\dots,n\}$. Když položíme $w_{i,j}=\sum_{k=i+1}^{j}\alpha_k+\sum_{k=i}^{j}\beta_k$, pak $c_{i,j}=H(i+1,j)$ a $r_{i,j}=R(i+1,j)$, a tedy konstrukce optimálního binárního vyhledávacího stromu je speciálním případem kvadratického programování. Na druhé straně tento vztah také ukazuje, jak můžeme v obecném případě pro kvadratické programování použít přirozenou modifikaci algoritmu **OPTIM**, a dostaneme

Věta 5.7. *Modifikovaný algoritmus* **OPTIM** *nalezne řešení úlohy kvadratického programování v čase $O(n^3)$.*

Vyšetříme, jak je to s vylepšenou verzí algoritmu **OPTIM**. Pro $0 \leq i < j \leq n$ označme $r_{i,j}$ největší přirozené číslo k takové, že $i < k \leq j$ a $c_{i,k-1}+c_{k,j} \leq c_{i,l-1}+c_{l,j}$ pro každé

přirozené číslo l takové, že $i < l \leq j$. Bude-li platit $r_{i,j-1} < r_{i,j} < r_{i+1,j}$ pro všechna $0 \leq i < j \leq n$, pak můžeme použít vylepšenou verzi algoritmu **OPTIM** (a za stejných předpokladů bude platit i $R(i, j-1) \leq R(i, j) \leq R(i+1, j)$). V následující části ukážeme, že podmínky

$$w_{i,j} + w_{i',j'} \leq w_{i,j'} + w_{i',j} \text{ pro všechna } 0 \leq i \leq i' < j \leq j' \leq n$$
$$w_{i',j'} \leq w_{i,j} \text{ pro všechna } 0 \leq i \leq i' < j' \leq j \leq n$$

implikují platnost $r_{i,j-1} < r_{i,j} < r_{i+1,j}$ pro všechna $0 \leq i < j \leq n$. První z těchto podmínek se nazývá čtyřúhelníková nerovnost a druhá podmínka monotonie.

Z převodu úlohy konstrukce optimálního binárního vyhledávacího stromu na kvadratické programování plyne, že čtyřúhelníková nerovnost v tomto případě vždy platí (když rozepíšeme $w_{i,j}$, tak dostaneme dokonce rovnost) a podmínka monotonie je splněna, právě když α_i a β_j jsou nezáporná čísla pro všechna $i = 1, 2, \ldots, n$ a $j = 0, 1, \ldots, n$. To znamená, že pro nezáporné ohodnocení můžeme nalézt optimální binární vyhledávací strom v kvadratickém čase. Důkaz, že čtyřúhelníková nerovnost a monotonie čísel $w_{i,j}$ zaručují platnost $r_{i,j-1} < r_{i,j} < r_{i+1,j}$, je rozdělen na dvě části. Nejprve dokážeme technické lemma.

Lemma 5.8. *Platí* $c_{i,j} + c_{i',j'} \leq c_{i,j'} + c_{i',j}$ *pro každé* $0 \leq i \leq i' \leq j \leq j' \leq n$.

Důkaz. Tvrzení dokážeme indukcí podle $j' - i$. Když $i = i'$ nebo $j = j'$, pak tvrzení zřejmě platí (platí rovnost). Odtud plyne, že když $j' - i \leq 1$, pak tvrzení platí. Předpokládejme nyní, že tvrzení platí, když $0 \leq j' - i < q$, a nechť $j' - i = q$. Důkaz rozdělíme do dvou kroků.

Nejprve předpokládáme, že $j = i'$. Označme $k = r_{i,j'}$. Předpokládejme, že $k \leq j$ (případ $k \geq j$ se řeší symetricky). Protože $i < k$, tak z indukčního předpokladu plyne $c_{k,j} + c_{j,j'} \leq c_{k,j'}$, protože $c_{j,j} = 0$. Odtud dostáváme

$$c_{i,j} + c_{j,j'} \leq w_{i,j} + c_{i,k-1} + c_{k,j} + c_{j,j'} \leq w_{i,j} + c_{i,k-1} + c_{k,j'} \leq w_{i,j'} + c_{i,k-1} + c_{k,j'} = c_{i,j'},$$

kde první nerovnost plyne z definice $c_{i,j}$, druhá nerovnost plyne z indukčního předpokladu a třetí z předpokladů na $w_{i,j}$, protože $0 \leq i < j \leq j' \leq n$.

Nyní předpokládáme, že $i' < j$. Označme $k = r_{i,j'}$ a $l = r_{i',j}$. Uvažujme, že $k \leq l$ (případ $l \leq k$ se řeší symetricky). Pak platí $1 \leq i < k \leq l \leq j \leq n$ a $1 \leq i' < l \leq j \leq j' \leq n$, a proto

$$\begin{aligned} c_{i,j} + c_{i',j'} \leq & w_{i,j} + c_{i,k-1} + c_{k,j} + w_{i',j'} + c_{i',l-1} + c_{l,j'} = \\ & w_{i,j} + w_{i',j'} + c_{i,k-1} + c_{i',l-1} + c_{k,j} + c_{l,j'} \leq \\ & w_{i,j'} + w_{i',j} + c_{i,k-1} + c_{i',l-1} + c_{k,j} + c_{l,j'} \leq \\ & w_{i,j'} + w_{i',j} + c_{i,k-1} + c_{i',l-1} + c_{k,j'} + c_{l,j} = \\ & w_{i,j'} + c_{i,k-1} + c_{k,j'} + w_{i',j} + c_{i',l-1} + c_{l,j} = c_{i,j'} + c_{i',j}, \end{aligned}$$

kde první nerovnost plyne z definice $c_{i,j}$ a $c_{i',j'}$, druhá nerovnost plyne z předpokladů na $w_{i,j}$, protože $1 \leq i \leq i' < j \leq j' \leq n$, a třetí plyne z indukčního předpokladu, protože když $1 \leq k \leq l \leq j \leq j' \leq n$ a $j' - k < j' - i$, tak $c_{k,j} + c_{l,j'} \leq c_{k,j'} + c_{l,j}$. □

Lemma 5.9. *Platí* $r_{i,j-1} \le r_{i,j} \le r_{i+1,j}$ *pro každé* $1 \le i < j \le n$.

Důkaz. Dokážeme nerovnost $r_{i,j-1} \le r_{i,j}$, důkaz druhé nerovnosti je symetrický. Nerovnost $r_{i,j-1} \le r_{i,j}$ plyne z následující implikace: když $1 \le i < k' \le k < j \le n$, pak

$$c_{i,k-1} + c_{k,j-1} \le c_{i,k'-1} + c_{k',j-1} \implies c_{i,k-1} + c_{k,j} \le c_{i,k'-1} + c_{k',j}. \tag{1}$$

Skutečně, položíme-li $k = r_{i,j-1}$, pak z definice $r_{i,j-1}$ plyne, že pro každé k' takové, že $i < k' \le k$, platí $c_{i,k-1}+c_{k,j-1} \le c_{i,k'-1}+c_{k',j-1}$. Z (1) pak plyne $c_{i,k-1}+c_{k,j} \le c_{i,k'-1}+c_{k',j}$ a to podle definice $r_{i,j}$ znamená, že $k \le r_{i,j}$. Tedy (1) implikuje $r_{i,j-1} \le r_{i,j}$ a zbývá už jen dokázat, že (1) skutečně platí.

Zřejmě

$$c_{i,k-1} + c_{k,j-1} \le c_{i,k'-1} + c_{k',j-1} \iff c_{i,k'-1} + c_{k',j-1} - c_{i,k-1} - c_{k,j-1} \ge 0$$

a analogicky

$$c_{i,k-1} + c_{k,j} \le c_{i,k'-1} + c_{k',j} \iff c_{i,k'-1} + c_{k',j} - c_{i,k-1} - c_{k,j} \ge 0.$$

Tedy z nerovnosti

$$c_{i,k'-1} + c_{k',j-1} - c_{i,k-1} - c_{k,j-1} \le c_{i,k'-1} + c_{k',j} - c_{i,k-1} - c_{k,j} \tag{2}$$

pro $1 \le i < k' \le k < j \le n$ vyplyne (1), když $1 \le i < k' \le k < j \le n$. Nerovnost (2) je ekvivalentní s nerovností

$$c_{k',j-1} - c_{k,j-1} \le c_{k',j} - c_{k,j}, \quad \text{když } 1 \le k' \le k < j \le n,$$

a ta je dále ekvivalentní s nerovností

$$c_{k',j-1} + c_{k,j} \le c_{k',j} + c_{k,j-1}, \quad \text{když } 1 \le k' \le k < j \le n,$$

která plyne z Lemmatu 5.8. $\square$

Důsledek 5.10. *Když platí podmínky*

$$w_{i,j} + w_{i',j'} \le w_{i,j'} + w_{i',j} \text{ pro všechna } 0 \le i \le i' < j \le j' \le n$$
$$w_{i',j'} \le w_{i,j} \text{ pro všechna } 0 \le i \le i' < j' \le j \le n,$$

pak úloha kvadratického programování se vyřeší v čase $O(n^2)$. *Když* $\{\alpha_i \mid i = 1, 2, \dots, n\}$ *a* $\{\beta_j \mid j = 0, 1, \dots, n\}$ *jsou soubory nezáporných čísel, pak optimální binární vyhledávací strom se nalezne v čase* $O(n^2)$.

Poznámka. Když $R(i,j)$ je pouze některé k mezi i a j takové, že $H(i,k-1)+H(k+1,j) \le H(i,l-1)+H(l+1,j)$ pro každé $i \le l \le j$, pak nerovnost $R(i,j-1) \le R(i,j) \le R(i+1,j)$ nemusí platit. Protipříklad je lehké nalézt. Tato nerovnost platí, teprve když přidáme další předpoklad, totiž že $R(i,j)$ je největší (nebo nejmenší) k mezi i a j takové, že $H(i,k-1)+$

$H(k+1,j) \leq H(i,l-1) + H(l+1,j)$ pro každé $i \leq l \leq j$. Ověření podmínky pro nejmenší k je symetrické s důkazem, že nerovnost platí pro největší k.

Poznámka. Kvadratický čas je v praxi také často nepřijatelný. Přitom neznáme efektivnější algoritmus pro nalezení optimálního binárního vyhledávacího stromu. To vede k situacím, kdy je vhodné se vzdát optimality a použít aproximační algoritmy. Ukážeme jednoduchý aproximační algoritmus, který v lineárním čase nalezne binární vyhledávací strom T reprezentující množinu S, přičemž ohodnocení T je blízké ohodnocení optimálního binárního vyhledávacího stromu pro S.

Idea konstrukce stromu je jednoduchá: mějme množinu $S = \{x_1 < x_2 < \cdots < x_n\}$ a nechť pro operaci **MEMBER**(y) platí $\mathrm{Prob}(y = x_i) = \alpha_i$ pro každé $i = 1,2,\dots,n$, $\mathrm{Prob}(x_i < y < x_{i+1}) = \beta_i$ pro každé $i = 0,1,\dots,n$, kde definujeme $x_0 = -\infty$ a $x_{n+1} = \infty$. Pak $0 \leq \alpha_i, \beta_j \leq 1$ pro každé $i = 1,2,\dots,n$ a $j = 0,1,\dots,n$ a $\sum_{i=1}^{n} \alpha_i + \sum_{j=0}^{n} \beta_i = 1$. Naším cílem je nalézt rychle binární vyhledávací strom reprezentující S takový, že očekávaný vyhledávací čas bude jen o málo větší než minimální vyhledávací čas pro danou úlohu. Definujme rekurentně $u_0 = 0$, $u_1 = \beta_0$, $u_{2i} = u_{2i-1} + \alpha_i$ a $u_{2i+1} = u_{2i} + \beta_i$. Nalezněme i takové, že buď $u_i \leq \frac{1}{2} \leq u_{i+1}$. Když $i = 2j-1$ nebo $i = 2j$ a $\frac{1}{2} \leq \frac{u_i + u_{i+1}}{2}$, pak kořen bude reprezentovat prvek x_j. Když $i = 2j$ a $\frac{u_i + u_{i+1}}{2} < \frac{1}{2}$, pak kořen reprezentuje prvek x_{j+1}. Abychom nalezli prvek reprezentovaný levým respektive pravým synem kořene, tak nahradíme v těchto nerovnostech $\frac{1}{2}$ číslem $\frac{1}{4}$ respektive $\frac{3}{4}$ a v tomto procesu budeme rekurentně pokračovat.

Poznamenejme, že tento proces lze aplikovat na vyhledávání binárního vyhledávacího stromu, který je blízký optimálnímu binárnímu vyhledávacímu stromu. Definujeme u_i stejně jako dříve a pak místo 1 budeme používat $m = u_{2n+1}$. To znamená, že při hledání prvku reprezentujícího kořen použijeme $\frac{m}{2}$ místo $\frac{1}{2}$, na hledání prvku reprezentovaného levým synem kořene použijeme $\frac{m}{4}$ místo $\frac{1}{4}$, pro reprezentovaný pravým synem kořene použijeme $\frac{3m}{4}$ místo $\frac{3}{4}$, atd.

Nyní neformálně popíšeme algoritmus **Aprox-Opt-Strom**, který konstruuje tento "téměř optimální binární vyhledávací strom". Tento algoritmus nejprve spočítá ze zadaného vstupu proměnné u_i pro $i = 0,1,\dots,2n+1$, položí $m = u_{2n+1}$, a pak volá rekurzivní proceduru **Rekurs**, která konstruuje binární vyhledávací strom. Tato procedura se volá s parametry k a j kde $k > 0$ je přirozené číslo a $j < 2^k$ je liché přirozené číslo. Procedura pak najde i takové, že $u_i \leq \frac{jm}{2^k} \leq u_{i+1}$. Když $i = 0$, pak vytvoří vrchol reprezentující x_1, když $i = 2n$, pak vytvoří vrchol reprezentující x_n. Předpokládejme, že $1 \leq i \leq 2n-1$. Když i je liché, pak $l = \frac{i+1}{2}$, když i je sudé a $\frac{u_i + u_{i+1}}{2} < \frac{jm}{2^k}$, pak položí $l = \frac{i}{2} + 1$, když i je sudé a $\frac{jm}{2^k} \leq \frac{u_i + u_{i+1}}{2}$, pak $l = \frac{i}{2}$. Dále vytvoří vrchol reprezentující prvek x_l. Když $l = 1$ nebo $l > 1$, $i = 2l-1$ a $\frac{(2j-1)m}{2^{k+1}} > \frac{u_{i-1} + u_i}{2}$ nebo $i > l$, $i = 2l$ a $\frac{u_i + u_{i+1}}{2} < \frac{2(j-1)m}{2^{k+1}}$ nebo $l > 1$, $i = 2l$ a $\frac{(2j-1)m}{2^{k+1}} > \frac{u_{i-2} + u_{i-1}}{2}$, pak levý syn tohoto vrcholu je list (už nemá možnost reprezentovat žádný prvek) v opačném případě levý syn vrcholu bude vrchol vytvořený rekurzivním voláním **Rekurs**$(k+1, 2j-1)$. Analogicky, když $l = n$ nebo $l < n$, $i = 2l-1$ a $\frac{(2j+1)m}{2^{k+1}} \leq \frac{u_{i+1} + u_{i+2}}{2}$ nebo $l < n$, $i = 2l$ a $\frac{(2j+1)m}{2^{k+1}} \leq \frac{u_i + u_{i+1}}{2}$ nebo $l < n$, $i = 2l$ a $\frac{(2j+1)m}{2^{k+1}} \leq \frac{u_{i+2} + u_{i+3}}{2}$, pak pravý syn vrcholu je list (už nemá možnost reprezentovat žádný vrchol) v opačném případě pravý syn vrcholu bude vrchol vytvořený rekurzivním voláním **Rekurs**$(k+1, 2j+1)$.

Formální popis algoritmu:

Aprox-Opt-Strom$(S = \{x_1 < x_2 < \cdots < x_n\}, \alpha_i \mid i = 1, 2, \ldots, n, \beta_j \mid j = 0, 1, \ldots, n)$
$u_0 := 0$, $u_1 := \beta_1$, $l := 1$
while $l \leq 2n + 1$ **do**
  if l je sudé **then** $u_l := u_{l-1} + \alpha_{\frac{l}{2}}$ **else** $u_l := u_{l-1} + \beta_{\frac{l+1}{2}}$ **endif**
enddo
$m := u_{2n+1}$
Rekurs$(1, 1)$

Rekurs(k, j)
$i := 2n$
while $\frac{jm}{2^k} < u_i$ **do** $i := i - 1$ **enddo**
if $i = 0$ **then**
  $l := 1$
else
  if $l := 2n$ **then**
    $l := n$
  else
    if i je liché **then**
      $l := \frac{i+1}{2}$
    else
      if $\frac{u_i + u_{i+1}}{2} \leq \frac{jm}{2^k}$ **then**
        $l := \frac{i}{2} + 1$
      else
        $l := \frac{i}{2}$
      endif
    endif
  endif
endif
vytvoříme vrchol reprezentující prvek x_l
if platí jedna z následujících podmínek:
$l = 1$
nebo $l > 1$, $2l = i + 1$ a $\frac{u_{i-1} + u_i}{2} < \frac{(2j-1)m}{2^{k+1}}$
nebo $l > 1$, $2l = i + 2$ a $\frac{u_i + u_{i+1}}{2} < \frac{(2j-1)m}{2^{k+1}}$
nebo $l > 1$, $2l = i$ a $\frac{u_{i-2} + u_{i-1}}{2} < \frac{(2j-1)m}{2^{k+1}}$
then
  levý syn vrcholu je list
else
  levý syn vrcholu je vrchol vytvořený **Rekurs**$(k + 1, 2j - 1)$
endif
if platí jedna z následujících podmínek:
$l = n$
nebo $l < n$, $2l = i + 1$ a $\frac{(2j+1)m}{2^{k+1}} \leq \frac{u_{i+1} + u_{i+2}}{2}$
nebo $l < n$, $l = 2i + 2$ a $\frac{(2j+1)m}{2^{k+1}} \leq \frac{u_{i+2} + u_{i+3}}{2}$

nebo $l < n$, $l = 2i$ a $\frac{(2j+1)m}{2^{k+1}} \leq \frac{u_i+u_{i+1}}{2}$
then
pravý syn vrcholu je list
else
pravý syn vrcholu je vrchol vytvořený **Rekurs**$(k+1, 2j+1)$
endif

Věta 5.11. *Algoritmus* **Aprox-Opt-Strom** *v lineárním čase zkonstruuje strom T_A reprezentující n-prvkovou množinu S a když T_{Op} je optimální strom reprezentující množinu S, pak*

$$\text{hod}(T_A) \leq \text{hod}(T_{Op}) + a(\log e + \log(\frac{\text{hod}(T_{Op})}{a})) + 2\sum_{j=0}^{n} \beta_j,$$

kde $a = \sum_{i=1}^{n} \alpha_i$.

Konstrukci optimálního binárního vyhledávacího stromu navrhnul v roce 1971 Knuth. Presentovaná verze založená na vztahu ke kvadratickému programování byla navržena Yao 1980. Když $\beta_i = 0$ pro každé $i = 0, 1, \ldots, n$, pak lze optimální binární vyhledávací strom zkonstruovat v čase $O(n \log n)$. Aproximační algoritmus pro nalezení optimálního binárního vyhledávacího stromu pochází od Fredmana 1975, analýza pochází od Mehlhorna 1975 a 1977.

6. Samoupravující seznamy

V této části se budeme zabývat samoupravujícími strategiemi pro seznamy. Nejprve popíšeme model, který budeme vyšetřovat. Data jsou uložena ve spojovém seznamu a předpokládáme, že tento seznam je prostý. Operacemi jsou **MEMBER**, **INSERT** a **DELETE**. Algoritmy realizující tyto operace mohou přemístit prvek, který je argumentem operace, na libovolné místo seznamu (pokud znají místo, kam ho chtějí přemístit) a ostatní prvky ponechají na svých místech. Na druhé straně vyhledávat prvek nebo místo v seznamu mohou jen průchodem seznamu od jeho počátku. Budeme předpokládat, že návštěva prvku vyžaduje jednotku času. Formálně řečeno, máme-li n-prvkový seznam, který neobsahuje prvek x, pak operace **MEMBER**(x), **INSERT**(x) a **DELETE**(x) vyžadují čas $n+1$ (v případě operace **INSERT**(x) může algoritmus vložit prvek x na libovolné místo a vyžaduje to čas $n+1$). Když prvek x je v seznamu na i-tém místě, pak operace **MEMBER**(x) a **INSERT**(x) vyžadují čas $\max\{i, j\}$, pokud chtějí prvek x přesunout na j-té místo, a operace **DELETE**(x) vyžaduje čas i.

Pro analýzu budeme potřebovat pojmy popisující naši situaci detailněji. Když x není v seznamu, pak <u>vyhledávací čas</u> algoritmu je $n+1$, když x je na i-tém místě, pak <u>vyhledávací čas</u> algoritmu je i. Když x není v seznamu a operace **INSERT**(x) ho vloží na j-té místo, pak algoritmus provedl $n+1-j$ <u>volných výměn</u>. Když x je na i-tém místě a operace **MEMBER**(x) nebo **INSERT**(x) ho přemístí na j-té místo, pak řekneme, že algoritmus provedl $\max\{i-j, 0\}$ <u>volných výměn</u> a $\max\{j-i, 0\}$ <u>placených výměn</u>.

Bylo navrženo několik obecných strategií, z nichž uvedeme tři.

> <u>MFR-strategie</u> (<u>move front rule</u>): když prvek x je v seznamu, pak operace **MEMBER**(x) a **INSERT**(x) ho přemístí na první místo v seznamu a operace **DELE-**

TE(x) ho odstraní ze seznamu. Když x není v seznamu, pak **INSERT**(x) ho vloží do seznamu na první místo.

TR-strategie (transition rule): když prvek x je na i-tém místě v seznamu, pak operace **MEMBER**(x) a **INSERT**(x) ho přemístí na $\max\{i-1,1\}$-ní místo, operace **DELETE**(x) ho vymaže. Když x není v seznamu, pak **INSERT**(x) ho vloží na předposlední místo v seznamu (tj. na n-té místo, když seznam měl délku n).

TIMESTAMP-strategie: Tato strategie si pamatuje historii operací aplikovaných na daný seznam prvků. Když prvek x dosud nebyl argumentem žádné operace, pak, je-li v seznamu, zůstane po operacích **MEMBER**(x) a **INSERT**(x) na svém místě, operace **DELETE**(x) ho odstraní ze seznamu. Pokud x není v seznamu, pak **INSERT**(x) ho vloží na poslední místo. Když prvek x byl argumentem nějaké operace, pak se najde první prvek y stojící v seznamu před x takový, že nebyl argumentem žádné operace od chvíle, kdy proběhla poslední operace s argumentem x. Operace **MEMBER**(x) a **INSERT**(x) pak přemístí x na pozici před y. V případě, že takové y neexistuje, zůstane x na svém místě (pokud byl v seznamu), resp. je vložen operací **INSERT**(x) na konec seznamu (pokud v něm není). Operace **DELETE**(x) opět pouze vymaže x ze senamu.

Lze říci, že TR-algoritmus má lepší očekávaný čas než MFR-algoritmus, ale jsou případy, kdy naopak MFR-algoritmus je podstatně rychlejší než TR-algoritmus. Ukážeme, že žádný algoritmus nemůže být, neformálně řečeno, více než dvakrát rychlejší než MFR-algoritmus. Naším prvním cílem bude formalizovat a dokázat toto tvrzení. Základním nástrojem při analýze samoupravujících struktur je amortizovaná složitost. Obvykle je dokazováno, že dobrá samoupravující struktura je nejvýše k-krát pomalejší než libovolný jiný algoritmus pro provedení dané posloupnosti operací, kde k je poměrně malá konstanta (plus aditivní konstanta, která popisuje rozdíl struktur, při kterém algoritmy začínají). Pro MFR-strategii dokážeme:

Věta 6.1. *Nechť $\mathcal{P}$ je posloupnost operací* **MEMBER**, **INSERT** *a* **DELETE** *a nechť L_1 a L_2 jsou dva seznamy prvků množiny S o velikosti n. Označme t_{MFR} čas, který potřebuje MFR-algoritmus na provedení posloupnosti operací $\mathcal{P}$, když začíná na seznamu L_1. Pro algoritmus A označme c_A vyhledávací čas, p_A počet placených výměn a v_A počet volných výměn potřebných pro realizaci posloupnosti operací $\mathcal{P}$ algoritmem A, když začínal na seznamu L_2. Pak pro každý algoritmus A platí:*

(1) $t_{MFR} \leq 2c_A + p_A - v_A - |\mathcal{P}| + \frac{n(n-1)}{2}$,
(2) *když* $L_1 = L_2$, *pak* $t_{MFR} \leq 2c_A + p_A - v_A - |\mathcal{P}|$,

kde $|\mathcal{P}|$ je délka posloupnosti $\mathcal{P}$.

Důkaz. Nejprve odhadneme amortizovanou složitost MFR-algoritmu v závislosti na algoritmu A. K tomu budeme potřebovat ohodnotit dva prosté seznamy C_1 a C_2 reprezentující stejnou množinu B. Označme

$$\mathrm{bal}(C_1, C_2) = |\{(x, y) \mid x, y \in B,\ x \neq y,\ x \text{ je před } y \text{ v } C_1,\ x \text{ je za } y \text{ v } C_2\}|.$$

Pak $0 \leq \mathrm{bal}(C_1, C_2) \leq \frac{|B|^2-|B|}{2}$ a $\mathrm{bal}(C_1, C_2) = 0$, právě když $C_1 = C_2$, a $\mathrm{bal}(C_1, C_2) = \frac{|B|^2-|B|}{2}$, právě když C_2 je zrcadlově převrácený seznam C_1.

Předpokládejme, že MFR-algoritmus provede operaci $\mathcal{O}$ na seznamu C_1 a vznikne tak seznam D_1 a algoritmus A provede operaci $\mathcal{O}$ na seznamu C_2 a vytvoří tím seznam D_2. Pak amortizovaná složitost operace $\mathcal{O}$ je

$$\text{čas operace } \mathcal{O} + \text{bal}(D_1, D_2) - \text{bal}(C_1, C_2).$$

Dokážeme následující technické lemma.

Lemma 6.2. *Když algoritmus A při realizaci operace $\mathcal{O}$ na seznamu C_2 vyžaduje vyhledávací čas c a provede v volných a p placených výměn, pak amortizovaná složitost MFR-algoritmu při realizaci operace $\mathcal{O}$ na seznamu C_1 je nejvýše $2c + p - v - 1$.*

Důkaz. Označme S standardní algoritmus, tj. algoritmus, který neprovádí žádné přemísťování a při operaci **INSERT**(x) vloží x na konec seznamu (pokud v něm není). Když algoritmus S provede operaci $\mathcal{O}$ na seznamu C_2, dostaneme seznam C_3. Označme t čas, který potřebuje MFR-algoritmus pro provedení operace $\mathcal{O}$ na seznamu C_1. Pak amortizovaná složitost MFR-algoritmu pro operaci $\mathcal{O}$ je

$$a = t + \text{bal}(D_1, D_2) - \text{bal}(C_1, C_2) = t + \text{bal}(D_1, C_3) - \text{bal}(C_1, C_2) + \text{bal}(D_1, D_2) - \text{bal}(D_1, C_3).$$

Protože D_2 vznikne z C_3 pouze výměnami s prvkem x a protože x je první v seznamu D_1, volná výměna zmenší $\text{bal}(D_1, C_3)$ o 1 a placená výměna zvětší $\text{bal}(D_1, C_3)$ o 1. Tedy $\text{bal}(D_1, D_2) - \text{bal}(D_1, C_3) = p - v$.

Dále budeme analyzovat výraz $t + \text{bal}(D_1, C_3) - \text{bal}(C_1, C_2)$. Předpokládejme, že délka seznamu C_1 je n. Nejprve uvažujme, že x není v seznamu. Když provádíme operaci $\mathcal{O} =$ **MEMBER**(x) nebo $\mathcal{O} =$ **DELETE**(x), pak $t = c = n + 1$, $D_1 = C_1$, $D_2 = C_2 = C_3$, a tedy $a = t \leq 2c + p - v - 1$, protože $p = v = 0$. Když provádíme operaci $\mathcal{O} =$ **INSERT**(x), pak $t = c = n + 1$, $\text{bal}(D_1, C_3) - \text{bal}(C_1, C_2) = n$ (protože seznam D_1 vznikl ze seznamu C_1 vložením x na první místo a seznam C_3 vznikl ze seznamu C_2 vložením x na poslední místo), a tedy $a = 2n + 1 + p - v = 2c + p - v - 1$.

Nyní předpokládejme, že x je na i-té pozici v seznamu C_2, na j-té pozici v seznamu C_1 a že počet prvků z, které jsou před x v seznamu C_1 a za x v seznamu C_2, je k. Pak $j \leq i + k$. Když $\mathcal{O} =$ **DELETE**(x), pak $t = j$, $c = i$, $p = v = 0$ a $\text{bal}(D_1, C_3) - \text{bal}(C_1, C_2) \leq -k$ (protože byly odstraněny všechny dvojice obsahující x a těch je alespoň k), a proto $a \leq j - k \leq i = 2c - 1 = 2c + p - v - 1$. Zbývá nám případ, že $\mathcal{O} =$ **MEMBER**(x) nebo $\mathcal{O} =$ **INSERT**(x). Pak $t = j$, $c = i$, $\text{bal}(D_1, C_3) - \text{bal}(C_1, C_2) \leq i - k - 1$ (protože x je první prvek v seznamu D_1, tak bylo odstraněno alespoň k dvojic obsahujících x a přibylo nejvýše $i - 1$ dvojic obsahujících x – dvojice (x, y), kde y je v seznamu C_2 před x). Tedy $a \leq j + i - k - 1 + p - v \leq 2i + p - v - 1 = 2c + p - v - 1$. □

Když vysčítáme odhad uvedený v Lemmatu 6.2 přes všechny operace v $\mathcal{P}$, dostaneme, že amortizovaná složitost posloupnosti $\mathcal{P}$ je nejvýše $2c_A + p_A - v_A - |\mathcal{P}|$ (protože časy jednotlivých operací i počty výměn se sčítají). Protože funkce bal je nezáporná, dostáváme, že

$$t_{MFR} \leq 2c_A + p_A - v_A - |\mathcal{P}| + \text{bal}(L_1, L_2).$$

Z definice $\text{bal}(L_1, L_2)$ plyne, že $0 \leq \text{bal}(L_1, L_2) \leq \binom{n}{2} = \frac{n(n-1)}{2}$ a když $L_1 = L_2$, pak $\text{bal}(L_1, L_2) = 0$. Odtud plyne tvrzení Věty 6.1. □

Poznámka. Chceme-li najít špatnou posloupnost pro TR-strategii, uvažujme posloupnost $\mathcal{P}$, která je konkatenací $\mathcal{P}_1$ a $\mathcal{P}_2$, kde

$$\mathcal{P}_1 = \{\mathbf{INSERT}(x_i)\}_{i=1}^n, \quad \mathcal{P}_2 = (\mathbf{MEMBER}(x_1), \mathbf{MEMBER}(x_n))^m,$$

a aplikujme ji na prázdný seznam. Předpokládáme, že $\{x_i \mid i = 1, 2, \dots, n\}$ jsou navzájem různé prvky. Pak TR-algoritmus při aplikaci $\mathcal{P}$ spotřebuje $\binom{n}{2} + mn$ času a MFR-algoritmus spotřebuje $\binom{n}{2} + n + 2 + 2(m-1)$ času. To ukazuje, že TR-algoritmus je na této posloupnosti řádově horší než MFR-algoritmus.

Pro množinu B a posloupnost operací $\mathcal{P}$ označme $C_{OPT}(\mathcal{P}, B)$ nejmenší čas, který potřebuje nějaký algoritmus na realizaci této posloupnosti na nějakém prostém seznamu prvků množiny B. Řekneme, že algoritmus A je c-kompetitivní, když existuje konstanta a taková, že pro každou posloupnost operací $\mathcal{P}$ a každou množinu B vykoná algoritmus A posloupnost $\mathcal{P}$ na libovolném prostém seznamu množiny B v čase nejvýše $cC_{OPT}(\mathcal{P}, B) + a + \frac{|B|^2}{2}$. Z věty 6.1 plyne, že MFR-strategie je 2-kompetitivní. Albersová v roce 1995 dokázala, že i algoritmus TIMESTAMP je 2-kompetitivní, a Karp a Raghavan oznámili, že umí dokázat, že když deterministický algoritmus je c-kompetitivní, pak $c \geq 2$.

Tento výsledek lze zlepšit pomocí randomizovaných algoritmů. Albersová navrhla pro dané $p \in < 0, 1 >$ použít následující randomizovanou strategii: když máme provést operaci $\mathcal{O}$, pak s pravděpodobností p použijeme MFR-strategii a s pravděpodobností $1-p$ použijeme TIMESTAMP-strategii, kterou však modifikujeme takto:

> vezmeme y jako první prvek v seznamu před x, který nebyl argumentem žádné operace od předchozí operace s argumentem x, nebo byl argumentem právě jedné operace od předchozí operace s argumentem x a tato operace nebyla realizována pomocí MFR-strategie.

Rozhodnutí, kterou strategii použijeme, provedeme takto: při realizaci operace $\mathcal{O}$ nejprve zvolíme náhodně r z intervalu $< 0, 1 >$ s rovnoměrným rozdělením a použijeme MFR-strategii, právě když $r \leq p$.

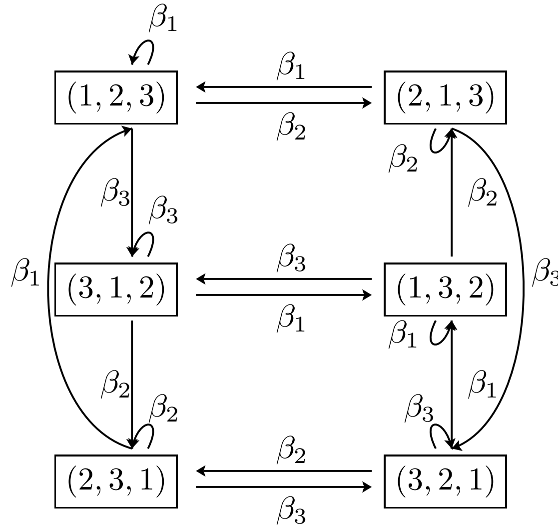
Albersová v roce 1995 ukázala, že tento randomizovaný algoritmus je $\max\{2 - p, 1 + p(2 - p)\}$-kompetitivní. V definici kompetitivnosti nahradí očekávaná složitost počítaná přes všechny možné volby proměnné r složitost v nejhorším případě, která se používá v definici pro deterministické algoritmy. Teia v roce 1993 ukázal, že když randomizovaný algoritmus je c-kompetitivní, pak $c \geq 1.5$. Je otevřený problém, zda takový algoritmus existuje (výsledek Albersové ukazuje, že existuje $\frac{1+\sqrt{5}}{2}$-kompetitvní algoritmus, stačí vzít $p = \frac{3-\sqrt{5}}{2}$).

Zde uvedená analýza v důkazu Věty 6.1 pochází od autorů Bentley a McGeoch z roku 1982, viz také Sleator a Tarjan 1983. Věta 6.1 je rovněž částečným vysvětlením, proč ve srůstajícím hašování je metoda EISCH lepší než LISCH a proč VICH je lepší než LICH a EICH.

Nyní si ukážeme analýzu očekávaného času pro MFR-strategii. Pro jednoduchost budeme předpokládat, že se používají jen operace $\mathbf{MEMBER}(x)$ pro x ze seznamu prvků. Pro analýzu použijeme Markovovy řetězce. To jsou vlastně pravděpodobnostní automaty s jedním vstupem. Stavy jsou situace, které mohou nastat (v našem případě všechny prosté seznamy reprezentované množiny B), a známe pravděpodobnost, se kterou přejdeme z jednoho stavu do druhého (v našem případě je to pravděpodobnost β_x, že prvek $x \in B$ je

argumentem operace). To znamená, že když operace **MEMBER**(x) transformuje seznam L_1 v seznam L_2, pak tento přechod nastane s pravděpodobností β_x, a když žádná operace netransformuje seznam L_1 v seznam L_2, pak tento přechod nastane s pravděpodobností 0.

Ilustrujme si tuto situaci pro množinu $B = \{0, 1, 2\}$ a pravděpodobnost β_x, že se provede operace **MEMBER**(x). Na Obr. 4 je znázorněn stavový diagram odpovídajícího Markovova procesu.



Obr. 4. Stavový diagram Markovova řetězce.

Tedy stavový diagram n-prvkové množiny má $n!$ stavů. Pro seznam π a prvek $x \in B$ bude $\pi(x)$ označovat místo, kde v seznamu π je x (tj. $\pi(x) = i$, právě když x je i-tý prvek v seznamu π). Je vidět, že náš Markovův řetězec je nerozložitelný a aperiodický, a tedy v něm existuje tzv. stacionární rozdělení. To znamená, že pro každý seznam π existuje limitní pravděpodobnost γ_π, že řetězec skončí po provedení "nekonečné posloupnosti" ve stavu π (nezávisle na počátečním stavu). To je jediný fakt, který budeme z teorie Markovových řetězců potřebovat. Nyní asymptotická očekávaná složitost operace **MEMBER**(x) je

$$P_{MFR} = \sum_{\pi} \gamma_\pi \sum_{x \in B} \beta_x \pi(x).$$

Pro dva navzájem různé prvky $x, y \in B$ označme $\delta(x, y)$ asymptotickou pravděpodobnost, že x je před y, tj.

$$\delta(x, y) = \sum \{\gamma_\pi \mid \pi \text{ je seznam } B,\ \pi(x) < \pi(y)\}.$$

Ukážeme si, že platí:

Lemma 6.3. *Platí*

$$P_{MFR} = \sum_{x \in B} \beta_x (1 + \sum_{y \in B, y \neq x} \delta(y, x))$$

a pro navzájem různá $x, y \in B$ je

$$\delta(x,y) = \frac{\beta_x}{\beta_x + \beta_y}.$$

Důkaz. Označme p_x asymptoticky očekávanou pozici prvku $x \in B$. Pak $p_x = \sum_\pi \gamma_\pi \pi(x)$. Tedy dostáváme

$$P_{MFR} = \sum_\pi \gamma_\pi \sum_{x\in B} \beta_x \pi(x) = \sum_{x\in B} \beta_x \sum_\pi \gamma_\pi \pi(x) = \sum_{x\in B} \beta_x p_x.$$

Nyní vyjádříme p_x pomocí $\delta(y, x)$. Platí

$$\begin{aligned} p_x =& \sum_\pi \gamma_\pi \pi(x) = \sum_\pi \gamma_\pi (1 + |\{y \in B \mid \pi(y) < \pi(x)\}|) = \\ & 1 + \sum_{y\in B, y\neq x} \sum \{\gamma_\pi \mid \pi \text{ je seznam } B,\, \pi(y) < \pi(x)\} = \\ & 1 + \sum_{y\in B, y\neq x} \delta(y,x). \end{aligned}$$

Když tento výsledek dosadíme do předchozího vzorce pro P_{MFR}, dostaneme požadovaný vztah. Pro vyčíslení výrazu pro $\delta(x, y)$ použijeme jiný postup výpočtu $\delta(x, y)$. Když x je před y, pak musela existovat operace **MEMBER**(x) taková, že po ní nenásledovala ani operace **MEMBER**(x) ani operace **MEMBER**(y). Tedy dostáváme, že $\delta(x, y) = \beta_x \sum_{k=0}^{\infty} (1 - (\beta_x + \beta_y))^k = \frac{\beta_x}{1-(1-(\beta_x+\beta_y))} = \frac{\beta_x}{\beta_x+\beta_y}$. $\square$

Pro množinu $B = \{x_1, x_2, \ldots, x_n\}$ a pravděpodobnosti $\beta_i = \text{Prob}(x = x_i)$, kde $i = 1, 2, \ldots, n$, označme P_{OPT} nejmenší očekávaný čas operace **MEMBER**(x). Když platí $\beta_{x_i} \geq \beta_{x_j}$ pro $j \geq i$, pak zřejmě je to očekávaný čas standardního vyhledávacího algoritmu (bez přemísťování) použitého na seznamu, kde x_i je na i-tém místě. Proto platí $P_{OPT} = \sum_{i=1}^n i\beta_{x_i}$. Dokážeme

Věta 6.4. *Platí*

$$P_{MFR} = 1 + \sum_{x,y\in B, x\neq y} \frac{\beta_x \beta_y}{\beta_x + \beta_y} \leq 2P_{OPT} - 1.$$

Důkaz. Použijeme výsledky z předchozího lemmatu. Platí

$$P_{MFR} = \sum_{x\in B} \beta_x (1 + \sum_{x\in B, x\neq y} \delta(y,x)) = 1 + \sum_{x\in B} \beta_x \sum_{y\in B, x\neq y} \frac{\beta_y}{\beta_x + \beta_y} = 1 + \sum_{x,y\in B, x\neq y} \frac{\beta_x \beta_y}{\beta_x + \beta_y}.$$

Když použijeme, že $\frac{\beta_y}{\beta_x+\beta_y} \leq 1$ pro každé $x, y \in B$, kde $B = \{x_1, x_2, \ldots, x_n\}$ a $\beta_{x_i} \geq \beta_{x_j}$ pro každé $i \leq j$, pak dostáváme

$$\begin{aligned} P_{MFR} =& 1 + \sum_{x,y\in B, x\neq y} \frac{\beta_x \beta_y}{\beta_x + \beta_y} = 1 + 2 \sum_{1\leq j<i\leq n} \frac{\beta_{x_i} \beta_{x_j}}{\beta_{x_i} + \beta_{x_j}} \leq 1 + 2 \sum_{1\leq j<i\leq n} \beta_{x_i} = \\ & 1 + 2\sum_{i=1}^n (i-1)\beta_{x_i} = 1 + 2\sum_{i=1}^n i\beta_{x_i} - 2\sum_{i=1}^n \beta_{x_i} = 1 + 2P_{OPT} - 2 = 2P_{OPT} - 1. \quad \square \end{aligned}$$

Důkaz, že očekávaný čas pro TR-strategii není nikdy horší než očekávaný čas pro MFR-strategii, je založen na stejné technice jako důkaz věty 6.4. Pro některá konkrétní rozdělení dat byly dokázány silnější výsledky (tj. že poměr $\frac{P_{MFR}}{P_{OPT}} < 2$). Tento postup lze zobecnit na obecnější model, ale je technicky daleko náročnější, a proto ho vynecháváme.

7. Splay stromy

Tuto sekci věnujeme samoupravující struktuře založené na binárních vyhledávacích stromech. Binárním vyhledávacím stromům spolu s následujícími algoritmy pro základní operace se říká SPLAY-stromy, protože všechny tyto operace (s vyjímkou jedné operace) jsou založeny na pomocné operaci **SPLAY**. Operace **SPLAY**(x, T) přestaví pomocí rotací a dvojitých rotací strom T tak, že když x je prvkem množiny reprezentované stromem T, pak x bude reprezentován kořenem stromu T, a když x není reprezentován ve stromě T, pak kořen T reprezentuje buď nejmenší prvek z reprezentované množiny větší než x nebo největší prvek z reprezentované množiny menší než x. To, která alternativa nastane, závisí na původním tvaru stromu. Podrobnější popis pomocného algoritmu **SPLAY** uvedeme později. V této struktuře budeme navíc předpokládat, že každému reprezentovanému prvku x je přiřazena váha $w(x)$, což je reálné číslo. Když budeme mluvit o váze vrcholu t, pak jí budeme rozumět váhu prvku reprezentovaného vrcholem t a budeme ji značit $w(t)$. Nyní neformálně popíšeme algoritmy této samoupravující datové struktury.

Ve všech operacích předpokládáme, že strom T (respektive T_i pro $i = 1, 2, \dots$) reprezentuje množinu S (respektive S_i). Operace **MEMBER**(x, T) nejprve zavolá pomocnou proceduru **SPLAY**(x, T) a pak podle prvku reprezentovaného kořenem oznámí výsledek. Z vlastností procedury **SPLAY** plyne, že po jejím provedení patří prvek x do reprezentované množiny, právě když je reprezentován kořenem. Proto když x je reprezentován kořenem T, oznámí, že $x \in S$, v opačném případě že $x \notin S$. Operace **SPLIT**(x, T) také nejprve zavolá proceduru **SPLAY**(x, T). Když x je reprezentován kořenem, pak položí $pris = true$, podstrom T určený levým synem kořene označí jako T_1 a podstrom T určený pravým synem kořene jako T_2. Když x není reprezentován kořenem, pak položí $pris = false$ a když x je menší než prvek reprezentovaný kořenem, označí podstrom T určený levým synem kořene jako T_1, ve stromě T nahradí tento podstrom listem a položí $T_2 = T$. Když x je větší než prvek reprezentovaný kořenem, pak označí podstrom T určený pravým synem kořene jako T_2, ve stromě T nahradí tento podstrom listem a položí $T_1 = T$. Strom T_1 tak reprezentuje množinu $S_1 = \{s \in S \mid s < x\}$ a strom T_2 množinu $S_2 = \{s \in S \mid s > x\}$. Proto výsledkem operace je dvojice stromů T_1 a T_2 (v tomto pořadí) a navíc, když $pris = true$, se oznámí, že x patřil do S, v opačném případě, že x nepatřil do S. Operace **CHANGEWEIGHT**(x, δ, T) přičte k váze prvku x, který je v reprezentované množině, hodnotu δ (když x nepatří k reprezentované množině, tak nemění váhu žádného prvku). Operace se provede tak, že zavolá pomocnou proceduru **SPLAY**(x, T) a když x je reprezentován v kořeni stromu, přičte hodnotu δ k jeho váze. Při operaci **JOIN2**(T_1, T_2) se předpokládá, že $\max S_1 < \min S_2$ (tento test operace neprovádí). Algoritmus operace **JOIN2**(T_1, T_2) nejprve zavolá pomocnou proceduru **SPLAY**(∞, T_1). Zde se ∞ pokládá za prvek větší než všechny prvky univerza U, a proto po jejím provedení kořen T_1 reprezentuje prvek $\max S_1$ a pravý syn kořene T_1 je list. Potom nahradí tento list kořenem T_2 a položí $T = T_1$. Analogicky při operaci **JOIN3**(T_1, x, a, T_2) se předpokládá, že $\max S_1 < x < \min S_2$ (tento test se opět neprovádí).

Algoritmus vytvoří nový vrchol v reprezentující prvek x s váhou a. Levým synem v bude kořen stromu T_1 a pravým synem v kořen stromu T_2. Výsledkem operace bude nový strom T s kořenem v. Operace **DELETE**(x, T) provede nejprve operaci **SPLIT**(x, T) a tím vytvoří dvojici stromů T_1 a T_2, z nichž žádný nereprezentuje prvek x, a pak provede operaci **JOIN2**(T_1, T_2). Analogicky je implementována operace **INSERT**(x, a, T). Algoritmus nejprve provede operaci **SPLIT**(x, T) a vytvoří tak dvojici stromů T_1 a T_2, kde každý prvek reprezentovaný ve stromě T_1 je menší než x a to je menší než libovolný prvek reprezentovaný v T_2. Na závěr provede operaci **JOIN3**(T_1, x, a, T_2).

Nyní uvedeme formální popis těchto algoritmů.

MEMBER(x, T)
SPLAY(x, T)
if $x = \text{key}(\text{kořen } T)$ **then**
  Výstup: $x \in S$
else
  Výstup: $x \notin S$
endif

SPLIT(x, T)
SPLAY(x, T)
if $x = \text{key}(\text{kořen } T)$ **then**
  $pris := true$
  T_1 :=podstrom určený levým synem kořene T
  T_2 :=podstrom určený pravým synem kořene T
else
  $pris := false$
  if $x < \text{key}(\text{kořen } T)$ **then**
    T_1 :=podstrom určený levým synem kořene T
    T_2 :=strom T, v němž levý syn kořene je nahrazen listem
  else
    T_2 :=podstrom určený pravým synem kořene T
    T_1 :=strom T, v němž pravý syn kořene je nahrazen listem
  endif
endif
Výstup:(T_1, T_2), **if** $pris$ **then** x byl v S **else** x nebyl v S **endif**

CHANGEWEIGHT(x, δ, T)
SPLAY(x, T)
if $x = \text{key}(\text{kořen } T)$ **then** $w(x) := w(x) + \delta$ **endif**

JOIN2(T_1, T_2)
SPLAY(∞, T_1)
pravy(kořen T_1) := kořen T_2, $T := T_1$

JOIN3(T_1, x, a, T_2)
vytvoř vrchol v, key$(v) := x$, $w(x) := a$
levy(v) :=kořen T_1, pravy(v) :=kořen T_2

DELETE(x, T)
SPLIT(x, T)
JOIN2(T_1, T_2)

INSERT(x, a, T)
SPLIT(x, T)
JOIN3(T_1, x, a, T_2)

Všimněme si, že operace **INSERT**(x, a, T) pro x, které je v reprezentované množině, může být použita ke změně váhy (po jejím provedení bude váha prvku x rovna a). Kdybychom tedy v tomto případě chtěli zabránit nežádoucí změně váhy, museli bychom po operaci **SPLIT**(x, T) nahradit a původní váhou $w(x)$ prvku x. Algoritmy operací **INSERT** a **DELETE** v této podobě mají elegantní zápis, ale pro lepší pochopení uvedeme ještě jejich přímočařejší popis. V algoritmu **INSERT**(x, a, T) se nebude měnit váha prvku x, když $x \in S$. Nejprve postup popíšeme neformálně. V operaci **DELETE**(x, T) se nejprve zavolá operace **SPLAY**(x, T). Když x není reprezentován v kořeni, pak operace skončí, protože x není prvek reprezentované množiny. V opačném případě provedeme operaci **JOIN2** následovně: Podstrom určený levým synem kořene stromu T označíme T_1 a provedeme operaci **SPLAY**(∞, T_1). Po nahrazení podstromu T_1 ve stromě T listem připojíme kořen stromu T jako pravého syna kořene T_1 a položíme $T = T_1$. Analogicky operace **INSERT**(x, a, T) začne pomocnou procedurou **SPLAY**(x, T). Když prvek x je reprezentován v kořeni stromu, operace skončí. V opačném případě vytvoří nový vrchol v reprezentující prvek x s váhou a. Když x je menší než prvek reprezentovaný kořenem stromu T, pak levým synem vrcholu v se stane levý syn kořene T, v podstromu T je nahrazen listem a kořen stromu T se stane pravým synem v (takto vytvořený strom se označí T). Když x je větší než prvek reprezentovaný kořenem stromu T, pak pravým synem vrcholu v se stane pravý syn kořene T, podstrom T určený pravým synem kořene T nahradí v T list a kořen stromu T se stane levým synem v (takto vytvořený strom se označí T).

DELETE(x, T)
SPLAY(x, T)
if $x =$ key(kořen T) **then**
  T_1 :=podstrom určený levým synem kořene T
  podstrom T_1 ve stromě T nahraď listem
  SPLAY(∞, T_1)
  pravy(kořen T_1) := kořen T, $T := T_1$
endif

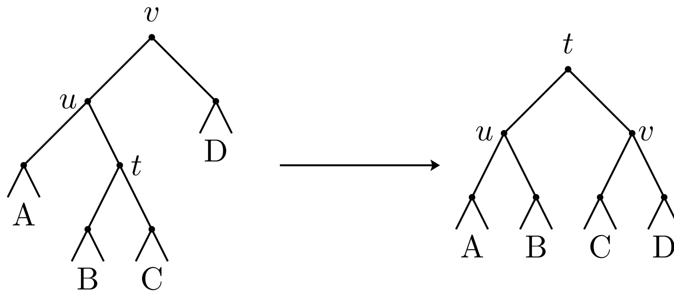
INSERT(x, a, T)
SPLAY(x, T)
if $x \neq$ key(kořen T) **then**
  vytvoř nový vrchol v, key$(v) := x$
  if $x <$ key(kořen T) **then**
    levy(v) := levy(kořen T)
    podstrom určený levým synem kořene T nahraď listem
    pravy(v) := kořen T

```
  else
    pravy(v) := pravy(kořen T)
    podstrom určený pravým synem kořene T nahraď listem
    levy(v) := kořen T
  endif
  T :=strom s kořenem v
endif
```

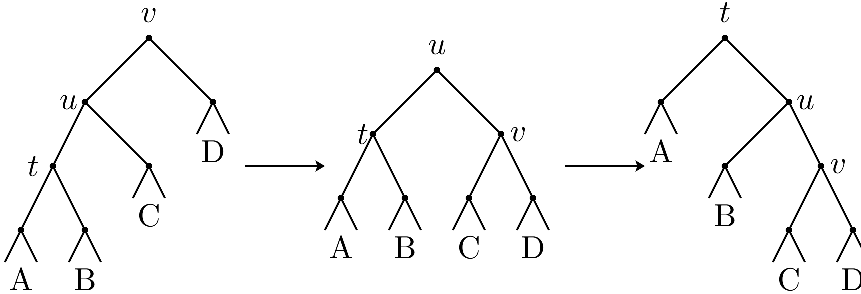
Nyní již zbývá pouze popsat operaci **SPLAY**. Základní idea přestavět strom tak, aby se všechna práce algoritmu odehrávala u kořene stromu, je motivována MFR-algoritmy. Problém je, jak efektivně přestavět strom. První pokusy byly neúspěšné, navržené algoritmy vyžadovaly amortizovaný lineární čas. Úspěšný návrh pochází od Sleatora a Tarjana z roku 1983. Procedura **SPLAY**(x, T) má dvě části. V první části použijeme standardní postup pro vyhledání prvku x ve stromě T. Začneme v kořeni t stromu T a dokud t nereprezentuje x nebo není list, opakujeme následující krok: když prvek reprezentovaný vrcholem t je větší než x, pak levý syn t bude novým vrcholem t, v opačném případě jím bude pravý syn t. Pokud skončíme v listu, pak označíme t otce tohoto listu. V tomto případě prvek reprezentovaný t je buď nejmenší reprezentovaný prvek větší než x (když nalezený list byl levým synem t) nebo největší reprezentovaný prvek menší než x (když nalezený list byl pravým synem t). Ve všech případech (ať už t reprezentuje prvek x nebo ne) přesuneme t pomocí rotací a dvojitých rotací do kořene tak, že dokud t není kořen, budeme provádět následující akci: když otec t je kořen, provedeme rotaci na vrcholy t a otec(t) (pak t bude kořen). Když otec t není kořen a t je lomený vrchol, provedeme dvojitou rotaci, a když t není lomený, provedeme rotaci na otce t a děda t a pak rotaci na t a otce t. V obou těchto případech se vrchol t dostane do hladiny o 2 menší, než byla původní hladina. Tyto přesuny jsou znázorněny na Obr. 5.

```
  SPLAY(x, T)
t :=kořen T
while key(t) ≠ x a t není list do
  if x < key(t) then t := levy(t) else t := pravy(t) endif
enddo
if t je list then t := otec(t) endif
while t není kořen do
  if otec(t) je kořen then
    Rotace(otec(t), t)
  else
    if t je lomený then
      Dvojita-Rotace(ded(t), otec(t), t)
    else
      Rotace(ded(t), otec(t)), Rotace(otec(t), t)
    endif
  endif
enddo
```

c) $u = \text{otec}(t)$ je kořen



b) $u = \text{otec}(t)$ není kořen, t je lomený



c) $u = \text{otec}(t)$ není kořen, t není lomený

Obr. 5. Transformace v operaci **SPLAY**.

Nyní odhadneme amortizovanou složitost těchto operací. Pro tento účel budeme předpokládat, že váha je kladná, tj. $w(x) > 0$ pro každé $x \in U$, a budeme definovat totální váhu vrcholu v jako

$$tw(v) = \sum \{w(t) \mid t \text{ je vrchol v podstromu určeném vrcholem } v\}$$

a rank vrcholu v jako $r(v) = \lceil \log_2 tw(v) \rceil$. Pro strom T je totální váha stromu $tw(T) = tw(\text{kořen } T)$ a rank stromu $r(T) = r(\text{kořen } T)$. Nejprve dokážeme pomocné lemma:

Lemma 7.1. *Když v je vrchol stromu takový, že jeho synové nejsou listy, pak*

$$r(v) > \min\{r(\text{levy}(v)), r(\text{pravy}(v))\}.$$

Důkaz. Zřejmě $tw(v) \geq tw(\text{levy}(v)) + tw(\text{pravy}(v)) \geq 2\min\{tw(\text{levy}(v)), tw(\text{pravy}(v))\}$. Odtud

$$\begin{aligned} r(v) \geq & \lceil \log_2(2\min\{tw(\text{levy}(v)), tw(\text{pravy}(v))\}) \rceil = \\ & 1 + \lceil \log_2(\min\{tw(\text{levy}(v)), tw(\text{pravy}(v))\}) \rceil = 1 + \min\{r(\text{levy}(v)), r(\text{pravy}(v))\}. \quad \square \end{aligned}$$

Definujme

$$\mathrm{bal}(\text{konfigurace}) = \sum\{r(v) \mid v \text{ je vrchol konfigurace}\}.$$

Když budeme potřebovat rozlišit ranky, totální váhy nebo váhy v různých konfiguracích, budeme přidávat jméno konfigurace jako index. To znamená, že $r_C(v)$, $tw_C(v)$ nebo $w_C(v)$ bude značit rank, totální váhu nebo váhu vrcholu v v konfiguraci C.

Všimněme si, že rank vrcholu v závisí pouze na váhách prvků reprezentovaných v podstromu určeném vrcholem v. Tedy když máme dvě konfigurace C_1 a C_2 a dva vrcholy v_1 a v_2 takové, že v_i je vrchol stromu T_i v konfiguraci C_i pro $i = 1, 2$, a když existuje prosté zobrazení f z množiny vrcholů podstromu T_1 určeného vrcholem v_1 do množiny vrcholů podstromu T_2 určeného vrcholem v_2 takové, že $w_{C_1}(t) = w_{C_2}(f(t))$ pro každý vrchol t podstromu T_1 určeného vrcholem v_1, pak $r_{C_1}(v_1) \leq r_{C_2}(v_2)$. Když f je bijekce, pak platí $r_{C_1}(v_1) = r_{C_2}(v_2)$. Toto pozorování označíme (†) a budeme ho využívat při odhadu amortizované složitosti.

Všimněme si dále, že dvě rotace a jedna dvojitá rotace vyžadují čas $O(1)$, proto lze říci, že algoritmus **SPLAY**(x, T) vyžaduje čas úměrný počtu běhů druhého cyklu v tomto algoritmu. Protože můžeme volit časovou jednotku, tak bez újmy na obecnosti můžeme předpokládat, že spotřebovaný čas je přímo počet opakování druhého cyklu. Dokážeme

Lemma 7.2. *Amortizovaná složitost operace* **SPLAY**(x, T) *je nejvýše* $3(r(T) - r(t)) + 1$, *kde t je vrchol stromu T, který se přemístí do kořene.*

Důkaz. Předpokládejme, že druhý cyklus se v běhu algoritmu **SPLAY**(x, T) opakoval k-krát. Označme T_{i-1} strom před i-tým během druhého cyklu. Pak $T = T_0$. Pro $i = 0, 1, \dots, k-2$ označme z_i děda vrcholu t ve stromě T_i a označme z_{k-1} kořen stromu T_{k-1} (pak z_{k-1} je buď otec vrcholu t nebo děd vrcholu t ve stromě T_{k-1}). Pro $i = 0, 1, \dots, k-1$ označme $r_i(v)$ rank vrcholu v ve stromě T_i. Když dokážeme, že amortizovaná složitost i-tého běhu druhého cyklu algoritmu **SPLAY**(x, T) je pro $i = 1, 2, \dots, k-1$ nejvýše $3(r_{i-1}(z_{i-1}) - r_{i-1}(t))$ a amortizovaná složitost k-tého běhu druhého cyklu je nejvýše $3(r_{k-1}(z_{k-1}) - r_{k-1}(t)) + 1$, pak amortizovaná složitost celého algoritmu je nejvýše

$$1 + \sum_{i=1}^{k} 3(r_{i-1}(z_{i-1}) - r_{i-1}(t)) = 1 + 3(r_{k-1}(z_{k-1}) - r_0(t)) = 3(r(T) - r(t)) + 1,$$

protože na základě pozorování (†) pro každé $i = 0, 1, \dots, k-2$ platí

$$r_0(z_i) = r_1(z_i) = \dots = r_i(z_i) = r_{i+1}(t)$$

a protože $r(T) = r_{k-1}(z_{k-1})$ a $r_0(t) = r(t)$.

Nyní ukážeme, že když z_{k-1} je otec vrcholu t ve stromě T_{k-1}, pak amortizovaná složitost k-tého běhu cyklu je nejvýše $3(r_{k-1}(z_{k-1}) + r_{k-1}(t)) + 1$. Podle definice a předpokladu na počítání času je amortizovaná složitost

$$\begin{aligned}
&1 + \mathrm{bal}(k\text{-tá konfigurace}) - \mathrm{bal}((k-1)\text{-ní konfigurace}) = \\
&1 + r_k(z_{k-1}) + r_k(t) - r_{k-1}(z_{k-1}) - r_{k-1}(t) \leq 1 + 2(r_k(t) - r_{k-1}(t)) = \\
&1 + 2(r_{k-1}(z_{k-1}) - r_{k-1}(t)) \leq 3(r_{k-1}(z_{k-1}) - r_{k-1}(t)) + 1,
\end{aligned}$$

protože $r_k(z_{k-1}) \leq r_k(t) = r_{k-1}(z_{k-1}) \geq r_{k-1}(t)$ a pro ostatní vrcholy v podle (†) platí $r_{k-1}(v) = r_k(v)$ (viz Obr 5a), kde $z_{k-1} = u$).

Dále ukážeme, že když z_{i-1} je děd vrcholu t ve stromě T_{i-1} a t je lomený ve stromě T_{i-1} pro $i = 1, 2, \ldots, k$, pak amortizovaná složitost i-tého běhu cyklu je nejvýše $3(r_{i-1}(z_{i-1}) - r_{i-1}(t))$. Označme u otce vrcholu t ve stromě T_{i-1}. Analogicky jako v předchozím případě dostáváme

$$
\begin{aligned}
&1 + \mathrm{bal}(i\text{-tá konfigurace}) - \mathrm{bal}((i-1)\text{-ní konfigurace}) = \\
&1 + r_i(z_{i-1}) + r_i(u) + r_i(t) - r_{i-1}(z_{i-1}) - r_{i-1}(u) - r_{i-1}(t) \leq 3(r_i(t) - r_{i-1}(t)) = \\
&3(r_{i-1}(z_{i-1}) - r_{i-1}(t)),
\end{aligned}
$$

protože $r_{i-1}(t) \leq r_{i-1}(u) \leq r_{i-1}(z_{i-1}) = r_i(t)$ a podle Lemmatu 7.1 je $2r_i(t) \geq r_i(u) + r_i(z_{i-1}) + 1$ (viz Obr 5b), kde $z_{i-1} = v$).

Předpokládejme, že z_{i-1} je děd vrcholu t ve stromě T_{i-1} a t není lomený ve stromě T_{i-1} pro $i = 1, 2, \ldots, k$. Dokážeme, že amortizovaná složitost i-tého běhu cyklu je nejvýše $3(r_{i-1}(z_{i-1}) - r_{i-1}(t))$. Označme u otce vrcholu t ve stromě T_{i-1}. Analogicky jako v přechozích případech dostáváme

$$
\begin{aligned}
&1 + \mathrm{bal}(i\text{-tá konfigurace}) - \mathrm{bal}((i-1)\text{-ní konfigurace}) = \\
&1 + r_i(z_{i-1}) + r_i(u) + r_i(t) - r_{i-1}(z_{i-1}) - r_{i-1}(u) - r_{i-1}(t).
\end{aligned}
$$

Označme T' prostřední strom na Obr 5c), to znamená, že T' je výsledkem **Rotace**(z_{i-1}, u) na strom T_{i-1}. Pak T_i vznikne z T' aplikací **Rotace**(u, t). Označme r' rank vrcholu ve stromě T'. Pak podle (†) a podle Obr. 5c) platí $r_{i-1}(t) \leq r_{i-1}(u) \leq r_{i-1}(z_{i-1})$, $r_i(z_{i-1}) \leq r_i(u) \leq r_i(t)$, $r_{i-1}(t) = r'(t)$, $r_i(z_{i-1}) = r'(z_{i-1})$, $r_{i-1}(z_{i-1}) = r'(u) = r_i(t)$ a z Lemmatu 7.1 plyne, že buď $r'(t) < r'(u)$ nebo $r'(z_{i-1}) < r'(u)$. Proto buď $r_{i-1}(t) < r_{i-1}(z_{i-1})$ nebo $r_i(z_{i-1}) < r_i(t)$. V prvním případě

$$
\begin{aligned}
&1 + r_i(z_{i-1}) + r_i(u) + r_i(t) - r_{i-1}(z_{i-1}) - r_{i-1}(u) - r_{i-1}(t) \leq \\
&1 + 3r_i(t) - r_{i-1}(z_{i-1}) - 2r_{i-1}(t) \leq 3(r_i(t) - r_{i-1}(t)) = 3(r_{i-1}(z_{i-1}) - r_{i-1}(t)).
\end{aligned}
$$

Ve druhém případě

$$
\begin{aligned}
&1 + r_i(z_{i-1}) + r_i(u) + r_i(t) - r_{i-1}(z_{i-1}) - r_{i-1}(u) - r_{i-1}(t) \leq \\
&1 + 2r_i(t) + r_i(z_{i-1}) - 3r_{i-1}(t) \leq \\
&3(r_i(t) - r_{i-1}(t)) = 3(r_{i-1}(z_{i-1}) - r_{i-1}(t)).
\end{aligned}
$$

Tím je důkaz hotov. □

Pro další aplikace je dobré si uvědomit, že Lemma 7.2 říká, že amortizovaná složitost algoritmu **SPLAY**(x, T) je $O(\log \frac{tw(T)}{tw(t)})$, kde t je vrchol, který je přemístěn do kořene stromu. To použijeme v důkazu následující věty.

Věta 7.3. *Nechť T je binární vyhledávací strom reprezentující množinu S. Pak amortizovaná složitost operací*

(1) **MEMBER**(x,T) *je* $O(\log \frac{tw(T)}{\min\{tw(t_-),tw(t_+)\}})$,
(2) **SPLIT**(x,T) *je* $O(\log \frac{tw(T)}{\min\{tw(t_-),tw(t_+)\}})$,
(3) **CHANGEWEIGHT**(x,δ,T) *je* $O(\log \frac{tw(T)+\delta}{\min\{tw(t_-),tw(t_+)\}})$,
(4) **INSERT**$(x,w(x),T)$ *je* $O(\log \frac{tw(T)+w(x)}{\min\{tw(t_-),tw(t_+)\}})$,
(5) **DELETE**(x,T) *je* $O(\log \frac{tw(T)}{\min\{tw(t_-),tw(t_+),tw(t_0)\}})$,
(6) **JOIN3**$(T_1,x,w(x),T_2)$ *je* $O(\log \frac{tw(T_1)+tw(T_2)+w(x)}{w(x)})$,

kde t_- je vrchol T reprezentující prvek $\max\{s \in S \mid s \leq x\}$, *$t_+$ je vrchol T reprezentující prvek* $\min\{s \in S \mid s \geq x\}$ *a t_0 je vrchol T reprezentující prvek* $\max\{s \in S \mid s < x\}$.

Když T_1 je binární vyhledávací strom reprezentující množinu S_1 a když t_∞ je vrchol T_1 reprezentující největší prvek v S, pak amortizovaná složitost operace **JOIN2**(T_1,T_2) *je* $O(\log \frac{tw(T_1)+tw(T_2)}{tw(t_\infty)})$.

Důkaz. Začneme s operací **JOIN2**. Z Lemmatu 7.2 plyne, že amortizovaná složitost algoritmu **SPLAY**(∞,T_1) je $O(\log \frac{tw(T_1)}{tw(t_\infty)})$, proto z popisu operace **JOIN2**(T_1,T_2) plyne, že její amortizovaná složitost je

$$O(\log \frac{tw(T_1)}{tw(t_\infty)} + \log(tw(T_1)+tw(T_2)) - \log tw(T_1)) = O(\log \frac{tw(T_1)+tw(T_2)}{tw(t_\infty)}).$$

Zřejmě amortizovaná složitost operace **SPLAY**(x,T) je $O(\log \frac{tw(T)}{\min\{tw(t_-),tw(t_+)\}})$. Odtud okamžitě plyne amortizovaná složitost operace **MEMBER**(x,T) a protože oddělení stromů v operaci **SPLIT**(x,T) zmenší ohodnocení konfigurace, dostáváme tak i amortizovanou složitost operace **SPLIT**(x,T). Amortizovaná složitost **CHANGEWEIGHT**(x,δ,T) je

$$O(\log \frac{tw(T)}{\min\{tw(t_-),tw(t_+)} + \log(tw(T)+\delta) - \log tw(T)) = O(\log \frac{tw(T)+\delta}{\min\{tw(t_-),tw(t_+)\}}).$$

Protože v operaci **DELETE**(x,T) se provádějí algoritmy **SPLAY**(x,T) a **SPLAY**(∞,T_1) a protože $tw(T_1) < tw(T)$, dává součet amortizovaných složitostí těchto operací amortizovanou složitost operace **DELETE**(x,T). Amortizovaná složitost operace **INSERT**(x,T) plyne z následujícího výpočtu

$$O(\log \frac{tw(T)}{\min\{tw(t_-),tw(t_+)\}} + \log(tw(T)+w(x)) + \log(tw(T)) - \log(tw(T)) = O(\log \frac{tw(T)+w(x)}{\min\{tw(t_-),tw(t_+)\}}),$$

protože rank kořene stromu T po operaci **SPLAY**(x,T) se může jen zmenšit. Pro operaci **JOIN3**(T_1,x,T_2) je amortizovaná složitost

$$O(1+\log(tw(T_1)+tw(T_2)+w(x)) - \log(w(x)) = O(\log \frac{tw(T_1)+tw(T_2)+w(x)}{w(x)}). \quad \square$$

Při odhadech amortizované složitosti jsme předpokládáli, že ověření splnění požadavků na operace (tj. $\max S_1 < \min S_2$ v operaci **JOIN2**(T_1, T_2), $\max S_1 < x < \min S_2$ v operaci **JOIN3**(T_1, x, T_2) a $w(x) + \delta > 0$ v operaci **CHANGEWEIGHT**(x, δ, T)) probíhá vždy mimo operaci, proto nebylo zahrnuto do amortizované složitosti.

Poznámka. Když chceme porovnat algoritmy pro vyvážené binární vyhledávací stromy a **SPLAY**-algoritmy a položíme $w(x) = 1$ pro každé $x \in U$, pak dostaneme výsledky, které se liší jen o multiplikativní konstantu, která je ve vyvážených binárních vyhledávacích stromech menší. Když však aplikujeme tyto struktury na jiné problémy, pak odčítání v amortizované složitosti pro **SPLAY**-algoritmy může hrát významnou roli a stává se, že pro ně dostaneme asymptoticky lepší výsledek.

Na závěr porovnáme výsledky pro **SPLAY**-algoritmy s optimálním binárním vyhledávacím stromem.

Věta 7.4. *Pro posloupnost operací* $\mathcal{P} = \{\mathbf{MEMBER}(y_j)\}_{j=1}^m$ *a pro množinu* $S = \{x_1 < x_2 < \dots < x_n\}$ *pro* $i = 1, 2, \dots, n$ *definujme* $\alpha_i = \frac{|\{j \mid j=1,2,\dots,m,\, y_j = x_i\}|}{m}$ *a pro* $i = 0, 1, \dots, n$ *definujme* $\beta_i = \frac{|\{j \mid j=1,2,\dots,m,\, x_i < y_j < x_{i+1}\}|}{m}$*, kde* $x_0 = -\infty$ *a* $x_{n+1} = \infty$*. Nechť* T_1 *je optimální binární vyhledávací strom pro* S*,* $\{\alpha_i \mid i = 1, 2, \dots, n\}$ *a* $\{\beta_i \mid i = 0, 1, \dots, n\}$ *a nechť* t *je čas, který vyžaduje realizace posloupnosti* $\mathcal{P}$ *klasickým algoritmem na stromě* T_1*. Když* T_2 *je binární vyhledávací strom reprezentující množinu* S*, pak realizace posloupnosti* $\mathcal{P}$ *na stromě* T_2 *pomocí* **SPLAY***-algoritmu vyžaduje čas* $O(t + |S|^2)$*.*

Důkaz. Předpokládejme, že výška T_1 (tj. délka nejdelší cesty z kořene do listu) je d a pro prvek $x \in S$ nechť d_x je hloubka vrcholu reprezentujícího x ve stromě T_1 (tj. délka cesty z kořene do tohoto vrcholu). Pak operace **MEMBER**(x, T_1) realizovaná klasickým algoritmem vyžaduje čas $O(d_x)$. Definujme $w(x) = 3^{d-d_x}$ váhu prvku x a odhadujme $tw(T_2)$. Protože T_1 je binární strom, existuje nejvýše 2^i prvků s váhou 3^{d-i}. Proto $tw(T_2) = \sum_{x \in S} 3^{d-d_x} \leq \sum_{i=0}^d 2^i 3^{d-i} = 3^d \sum_{i=0}^d (\frac{2}{3})^i \leq 3^{d+1}$. Tedy $r(T_2) \leq 1 + \log 3^{d+1} = O(n)$ a $\text{bal(konfigurace)} = \sum_{x \in S} r(x) = O(n^2)$. Nyní odhadneme amortizovanou složitost operace **MEMBER**(x). Když $x \in S$, pak podle Věty 7.3 je amortizovaná složitost nejvýše $O(r(T) - r(t)) = O(d + 1 - d + d_x) = O(d_x + 1)$, kde t je vrchol reprezentující x. Když $x \notin S$, pak hloubka listu b reprezentujícího interval obsahující x je $\max\{d_{x_-}, d_{x_+}\}$, kde $x_- = \max\{s \in S \mid s < x\}$ a $x_+ = \min\{s \in S \mid s > x\}$. Pak z Lemmatu 7.2 plyne, že amortizovaná složitost operace je $O(b) = O(d - d + b)$, protože $d - d_{x_-}, d - d_{x_+} \geq d - b$. Tedy amortizovaná složitost posloupnosti $\mathcal{P}$ je

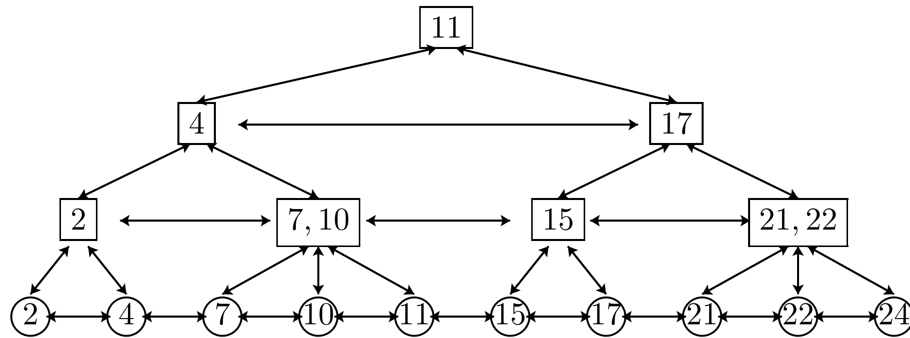
$$O(\sum_{j=1}^m (\text{čas operace } \mathbf{MEMBER}(y_j) \text{ v } T_1) = O(t).$$

Protože r je nezáporná funkce a protože $\text{bal(vstupní konfigurace)} = \text{bal}(T_2) = O(|S|^2)$, dostáváme, že **SPLAY**-algoritmus na realizaci posloupnosti $\mathcal{P}$ na stromě T_2 vyžaduje čas $O(t + \text{bal(vstupní konfigurace)}) = O(t + |S|^2)$. $\square$

Splay stromy definovali a jejich analýzu prezentovali Sleator a Tarjan v roce 1983. Také ukázali aplikaci splay stromů v jiných datových strukturách, např. pro reprezentaci stromu a operací se stromy.

8. Hladinově propojené (a, b)-stromy s prstem

V této části se vrátíme k hladinově propojeným (a, b)-stromům s prstem a ukážeme si zjemnění analýzy odhadu počtu vyvažovacích operací. Na základě této techniky pak odvodíme efektivní algoritmy pro klasické množinové operace – pro sjednocení, průnik a rozdíl množin. Ukážeme také odhady na spotřebu času pro posloupností operací, které nezačínají v prázdné množině, ale začnou na množině $S \subseteq U$ reprezentované nějakým hladinově propojeným (a, b)-stromem. Připomínáme, že hladinově propojené (a, b)-stromy s prstem jsou klasické (a, b)-stromy, kde navíc vrcholy v jedné hladině (vnitřní i listy) jsou propojeny dvousměrným seznamem v lexikografickém pořadí. Navíc je dán ukazatel $Prst$ na některý list. Rozdíl oproti klasickému přístupu je, že vyhledávání začíná od listu, na který ukazuje $Prst$, nikoliv od kořene. Proto každý vrchol v obsahuje ještě ukazatel otec(v). Příklad takového stromu je na Obr. 6.



Obr. 5. Hladinově propojený $(2, 4)$-strom.

Nejprve si připomeneme základní fakta, která jsme odvodili pro (a, b)-stromy a která platí i pro propojené (a, b)-stromy. V dalším textu budeme předpokládat, že $a \geq 2$ a $b \geq 2a$.

Věta 8.1. *Mějme hladinově propojený (a, b)-strom s prstem. Pak platí*

(1) *vyhledání listu t, který je vzdálen od prstu o d listů, vyžaduje čas $O(1 + \log_a d)$;*
(2) *když operace* **INSERT**(x) *zavolá m-krát podproceduru* **Štěpení***, pak vyžaduje čas* $O(\text{čas na vyhledání} + 1 + m)$*;*
(3) *když operace* **DELETE**(x) *zavolá m-krát podproceduru* **Spojení***, pak vyžaduje čas* $O(\text{čas na vyhledání} + 1 + m)$*;*
(4) *operace* **Prst** *vyžaduje čas $O(1 +$ čas na vyhledání$)$;*
(5) *když $a \geq 2$ a $b \geq 2a$, pak posloupnost $\mathcal{P}$ operací* **MEMBER***,* **INSERT***,* **DELETE** *a* **Prst** *aplikovaná na hladinově propojený (a, b)-strom s prstem reprezentující prázdnou množinu vyžaduje čas*

$$O(|P| + \text{čas na vyhledávání}).$$

Důkaz. Tvrzení (1) jsme dokázali při analýze A-**sort**u. Tvrzení (2) a (3) plynou z popisu algoritmů pro operace **INSERT** a **DELETE**. Operace **Prst**(x) vyžaduje o konstantu více času než operace **MEMBER**(x) a z toho plyne tvrzení (4). Tvrzení (5) plyne z věty o počtu vyvažovacích operací, protože změna způsobu vyhledávání nemá vliv ani na tvar (a,b)-stromu ani na počet vyvažovacích operací. □

Dále budeme předpokládat, že je dána posloupnost $\mathcal{P}$ operací **MEMBER**, **INSERT**, **DELETE** a **Prst**. Tuto posloupnost budeme aplikovat na hladinově propojený (a,b)-strom T a po jejím provedení dostaneme strom T'. Označme
St – počet procedur **Štěpení** provedených během aplikace $\mathcal{P}$ na T,
Sp – počet procedur **Spojení** provedených během aplikace $\mathcal{P}$ na T,
P – počet procedur **Přesun** provedených během aplikace $\mathcal{P}$ na T,
s – počet úspěšných operací **INSERT** (tj. operací, které přidaly prvek),
t – počet úspěšných operací **DELETE** (tj. operací, které odstranily prvek).

Nyní zpřesníme odhad počtu vyvažovacích operací a za tím účelem zjemníme důkazovou techniku, kterou jsme použili k výpočtu původního odhadu. Zřejmě platí $P \le t$, protože podprocedura **Přesun** se volá jen v úspěšné operaci **DELETE**, a to nejvýše jednou. K odhadu $St+Sp$ použijeme amortizované počítaní vyvažovacích operací zjemněné o následující ideu: při úspěšné operaci **DELETE** neodstraníme list, ale pouze změníme jeho strukturu a budeme ho nazývat fantom. Při provádění operací reálné vrcholy fantomy nevidí. Fantomy nemění svoji pozici vzhledem k ostatním listům. Při přidávání vrcholu je jeho pozice vzhledem k fantomu určena reálnými listy a pokud mezi ním a fantomem není reálný list, není jejich pořadí podstatné. Pro přehlednost dalšího postupu je vhodné (nikoliv nutné) zachovat uspořádání reprezentovaných prvků, což znamená, že když fantom reprezentuje větší prvek než je přidávaný prvek, vložíme nový prvek před fantom, a naopak. Všimněme si, že daný prvek může být reprezentován více listy, ale všechny až na nejvýše jednoho jsou fantomy.

Nyní popíšeme ohodnocení konfigurace, abychom mohli při počítání použít bankovní princip. Ohodnocení je modifikováno oproti odhadu počtu vyvažovacích operací v klasických (a,b)-stromech následovně: pro vrchol t hladinově propojeného (a,b)-stromu S různý od kořene definujeme

$$b(t) = \begin{cases} -1 & \text{když } \rho(t) = a-1 \text{ nebo } \rho(t) = b+1, \\ 0 & \text{když } \rho(t) = b \text{ nebo } \rho(t) = a, \\ 1 & \text{když } a < \rho(t) < b. \end{cases}$$

Pro kořen r stromu S definujeme

$$b(r) = \begin{cases} -1 & \text{když } \rho(r) = 1 \text{ nebo } \rho(r) = b+1, \\ 0 & \text{když } \rho(r) = b \text{ nebo } \rho(r) = 2, \\ 1 & \text{když } 2 < \rho(r) < b. \end{cases}$$

Podstatné je, že $\rho(t)$ je počet synů vrcholu t, které nejsou fantomy. Dále si všimněme, že platí-li $|\rho(t) - \rho(v)| \le 1$, pak $b(t) \ge b(v) - 1$ (zde není podstatné, zda vrcholy jsou ve stejném stromě nebo zda jsou to stejné vrcholy, tj. $t = v$, v různých stromech). Navíc

během aplikace posloupnosti operací $\mathcal{P}$ budou některé vrcholy označovány a pro strom S s označenými vrcholy budeme definovat

$$mb(S) = \sum\{b(t) \mid t \text{ je vnitřní označený vrchol } S\}.$$

Označování vrcholů bude ovlivňovat pouze podproceduru **Štěpení** (popíšeme ji za okamžik). Jinak, stejně jako existence fantomů, nemá žádný vliv na provádění operací. Označování vrcholů a fantomy použijeme pro výpočet odhadu ohodnocení struktury a tím i odhadu počtu vyvažovacích operací. Dokážeme následující tvrzení.

Věta 8.2. *Nechť strom T' má $n+s$ listů a z toho t listů jsou fantomy. Když očíslujeme listy od 1 do $n+s$ v rostoucím pořadí podle lexikografického uspořádání listů a když $p_1 < p_2 < \cdots < p_{s+t}$ jsou čísla všech nově přidaných listů a všech fantomů, pak platí*

$$St + Sp + P \leq 4s + 5t + 2(\lfloor \log_a \frac{n+s}{2} \rfloor + \sum_{i=2}^{s+t} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor).$$

Důkaz. Důkaz provedeme v několika krocích. Nejprve popíšeme mechanismus označování vrcholů. Na počátku žádný vrchol stromu T není označen. Při vložení nového listu nebo změně listu na fantoma označíme tento list a jeho otce. Procedura **Štěpení**(t) rozštěpí vrchol t na vrcholy t' a t''. Každý z nich bude označen právě tehdy, když některý jeho syn bude označen, a navíc bude označen i otec vrcholů t' a t''. V případě, že t' a t'' nemohou mít stejný počet synů (to nastane, když b je sudé), budeme požadovat, aby ten z nich, který má více synů než druhý, měl označeného syna. To lze při štěpení realizovat tak, že když se označený syn u vrcholu t stane synem t', pak stanovíme $\rho(t') = \lceil \frac{b+1}{2} \rceil$, jinak $\rho(t') = \lfloor \frac{b+1}{2} \rfloor$. Voláním procedury **Spojení**(t, y) vznikne z vrcholů t a y nový vrchol v, označeny budou vrcholy v a jeho otec. Procedura **Přesun**(t, y) odebere jednoho syna vrcholu y a připojí ho jako syna vrcholu t. Označen pak bude vrchol t a odstraní se označení vrcholu y (pokud byl označen). Indukcí snadno nahlédneme, že když vrchol je označen, pak je označen i některý jeho syn, a že v okamžiku, kdy máme provádět jednu z procedur **Štěpení**(t), **Spojení**(t, y) nebo **Přesun**(t, y), je vrchol t vždy označen (vrchol y označen být nemusí).

Nyní ohodnotíme vliv každé prováděné akce na ocenění stromu. Všimněme si, že operace **MEMBER** a **Prst** nemají vliv na ohodnocení, a tak je můžeme vynechat. Stejně tak neúspěšné operace **INSERT** a **DELETE**. Když $T = T_0, T_1, \dots, T_k = T'$ je posloupnost všech propojených (a, b)-stromů, které se objevily při realizaci posloupnosti operací $\mathcal{P}$ (v časovém pořadí), pak T_{i+1} vznikl z T_i buď přidáním nebo ubráním listu (přesněji změnou listu na fantoma), tj. provedením operace **INSERT** nebo **DELETE** bez vyvažovacích podprocedur, nebo provedením jedné z podprocedur **Štěpení**, **Spojení** nebo **Přesun**. To motivuje následující lemma.

Lemma 8.3. *Pro stromy T_i, T_{i+1} a výsledný strom T' platí:*

1. *když T_{i+1} vznikl z T_i přidáním listu, pak $mb(T_{i+1}) \geq mb(T_i) - 1$;*
2. *když T_{i+1} vznikl z T_i změnou listu na fantoma, pak $mb(T_{i+1}) \geq mb(T_i) - 1$;*
3. *když T_{i+1} vznikl z T_i provedením podprocedury* **Štěpení**, *pak $mb(T_{i+1}) \geq mb(T_i) + 1$;*
4. *když T_{i+1} vznikl z T_i provedením podprocedury* **Spojení**, *pak $mb(T_{i+1}) \geq mb(T_i) + 1$;*
5. *když T_{i+1} vznikl z T_i provedením podprocedury* **Přesun**, *pak $mb(T_{i+1}) \geq mb(T_i)$.*

Tedy $mb(T') \geq St + Sp - (s+t)$.

Důkaz. Z definice funkcí $mb(T)$ a $b(t)$ je vidět, že změna hodnoty $mb(T_{i+1})$ oproti $mb(T_i)$ závisí jen na vrcholech, které se staly označenými nebo přestaly být označenými, a dále na označených vrcholech, u nichž se změnil počet synů. Proto nás budou zajímat pouze tyto vrcholy.

Vyšetříme případ, kdy strom T_{i+1} vznikl ze stromu T_i buď přidáním listu nebo změnou listu na fantoma. Označme t otce takového listu. Protože jen u vrcholu t se změnila hodnota $b(t)$ (a podle poznámky za definicí $b(t)$ nejvýše o 1) a také jen u tohoto vrcholu se změnilo označení (a změna označení jednoho vrcholu vede ke změně hodnoty mb nějakého stromu také nejvýše o 1), dostáváme, že

$$mb(T_{i+1}) \geq mb(T_i) - 1,$$

a body (1) a (2) jsou dokázány.

Předpokládejme, že T_{i+1} vznikl z T_i provedením procedury **Štěpení**(t), která rozštěpila vrchol t na t' a t''. Označme v otce vrcholů t' a t''. Před operací byl vrchol t označen a platilo $b(t) = -1$ a dále vrchol v buď neexistoval (když t byl kořen), nebo nebyl označen. Podle poznámky za definicí funkce b je rozdíl $b(v)$ ve stromě T_{i+1} a $b(v)$ ve stromě T_i větší nebo roven -1. Proto rozdíl příspěvku v k $mb(T_{i+1})$ oproti jeho příspěvku k $mb(T_i)$ je větší nebo roven -1. Protože $b \geq 2a$, dostáváme $\lceil \frac{b+1}{2} \rceil > a$. Protože alespoň jeden z vrcholů t' a t'' je označený vrchol s $\lceil \frac{b+1}{2} \rceil$ syny, tak společný příspěvek vrcholů t' a t'' k $b(T_{i+1})$ je alespoň 1. Tedy rozdíl příspěvku vrcholů v, t' a t'' k $mb(T_{i+1})$ oproti příspěvku vrcholů t a v k $mb(T_i)$ je alespoň $1 - (-1) - 1 = 1$. Z toho plyne (3).

Předpokládejme, že T_{i+1} vznikl z T_i procedurou **Spojení**(t, y), která spojila vrcholy t a y do vrcholu v, a označme w otce vrcholu v ve stromě T_{i+1} (ten je zároveň ve stromě T_i otcem vrcholů t a y). Před procedurou platilo $b(t) = -1$, $b(y) = 0$ a vrchol t byl označen. Protože $a < \rho(v) = 2a - 1 < b$, tak $b(v) = 1$ v T_{i+1}. Podle téže poznámky dostaneme stejně jako v předchozím případě, že rozdíl příspěvku w k $mb(T_{i+1})$ oproti jeho příspěvku k $mb(T_i)$ je alespoň -1. Tedy rozdíl příspěvku vrcholů v a w k $mb(T_{i+1})$ oproti příspěvku vrcholů t, y a w k $mb(T_i)$ je $1 - (-1) - 0 + (-1) = 1$ a tvrzení (4) je dokázáno.

Předpokládejme, že T_{i+1} vznikl z T_i procedurou **Přesun**(t, y). Opět podle poznámky za definicí funkce b je rozdíl příspěvku y k $mb(T_{i+1})$ a k $mb(T_i)$ větší nebo roven -1. Před procedurou bylo $b(t) = -1$ a t byl označen, po proceduře je $b(t) = 0$. Tedy rozdíl příspěvku vrcholů t a y k $mb(T_{i+1})$ oproti jejich příspěvku k $mb(T_i)$ je $1 - (-1) = 0$ a odtud plyne (5).

Sečtením odhadů v jednotlivých případech dostaneme poslední nerovnost. □

Protože označené listy jsou reálné listy nebo fantomy, dostáváme, že když m je počet označených vrcholů ve stromě T' a l je počet vrcholů na cestách z označených listů ve stromě T' do kořene, pak

$$mb(T') \leq m \leq l.$$

Proto teď odhadneme číslo l.

K tomu použijeme následujícího pojmu, který zobecňuje (a, b)-stromy. Strom T se nazývá (a, ∞)-strom, když každý vnitřní vrchol různý od kořene má alespoň a synů, kořen má alespoň dva syny a všechny listy jsou ve stejné hladině (to znamená, že všechny cesty z kořene do listů mají stejnou délku). Zřejmě platí, že (a, ∞)-strom o výšce h má alespoň $2a^{h-1}$ listů a podstrom (a, ∞)-stromu určený vrcholem v, který není kořen a je ve výšce k, má alespoň a^k listů. Náš problém vyřeší následující lemma.

Lemma 8.4. *Mějme* (a,∞)*-strom* S *s* N *listy, které očíslujme čísly* $1,2,\dots,N$ *v rostoucím pořadí podle lexikografického uspořádání. Když* $1 \le p_1 < p_2 < \dots < p_k \le N$ *jsou přirozená čísla a* l *je počet všech vrcholů stromu* S *na všech cestách z kořene do listů očíslovaných* p_i *pro* $1 \le i \le k$*, pak*

$$l \le 3k + 2(\lfloor \log_a N \rfloor + \sum_{i=2}^{k} \lfloor \log_a (p_i - p_{i-1} + 1) \rfloor).$$

Důkaz. Pro každý vnitřní vrchol v očíslujme jeho syny čísly od $0,1,\dots,\rho(v)-1$, kde $\rho(v)$ je počet synů vrcholu v ve stromě S. Když strom S má výšku h, pak každý list je jednoznačně určen cestou z kořene do tohoto listu a každá cesta je jednoznačně určena slovem nad abecedou $\{0,1,\dots,r\}$, kde $r = \max\{\rho(t) \mid t \text{ je vnitřní vrchol } S\}$.

Zřejmě výška stromu S je nejvýše $1 + \lfloor \log_a(\frac{N}{2}) \rfloor$, a proto každá cesta z kořene do listu obsahuje nejvýše $2 + \lfloor \log_a(\frac{N}{2}) \rfloor$ vrcholů. Označme l_i počet vrcholů, které jsou na cestě z kořene do listu s číslem p_i, ale nepatří cestě z kořene do listu s číslem p_{i-1}. Pak zřejmě platí

$$l \le 2 + \lfloor \log_a(\frac{N}{2}) \rfloor + \sum_{i=2}^{k} l_i.$$

Naším cílem bude odhadnout $\sum_{i=2}^{k} l_i$. Za tím účelem označíme jako c_i počet nul ve slově určujícím cestu z kořene do listu s číslem p_i. Zřejmě platí $0 \le c_i \le \lfloor \log_a(\frac{N}{2}) \rfloor$ pro každé $i = 2,\dots,k$ a $0 \le c_1 \le 1 + \lfloor \log_a(\frac{N}{2}) \rfloor$. Předpokládejme, že platí

$$(\dagger) \qquad c_i \ge c_{i-1} + l_i - 2\lfloor \log_a(p_i - p_{i-1} + 1) \rfloor - 3$$

pro každé $i = 2,3,\dots,k$. Pak rekurzivním dosazováním dostaneme

$$c_k \ge c_1 + \sum_{i=2}^{k} l_i - 2\sum_{i=2}^{k} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor - 3(k-1).$$

Použijeme odhady $c_1 \ge 0$ a $c_k \le \lfloor \log_a(\frac{N}{2}) \rfloor$ a dostaneme nerovnost

$$\sum_{i=2}^{k} l_i \le 3k - 3 + \lfloor \log_a(\frac{N}{2}) \rfloor + 2\sum_{i=2}^{k} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor.$$

Odtud plyne, že

$$l \le 3k - 1 + 2(\lfloor \log_a(\frac{N}{2}) \rfloor + \sum_{i=2}^{k} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor)$$

a důkaz Lemmatu tak bude hotov. Zbývá tedy dokázat (†). Vezměme $i > 1$ a nechť v je poslední společný vrchol na cestách z kořene do listů s čísly p_i a p_{i-1}. Pak existují $k_1 < k_2$, nejmenší q a slova α, β a γ tak, že $|\beta| = |\gamma| = q$, α neobsahuje 0, slovo $k_1\alpha\beta$ určuje cestu z

vrcholu v do listu s číslem p_{i-1} a slovo $k_2 0^{|\alpha|}\gamma$ určuje cestu z vrcholu v do listu s číslem p_i. Zřejmě buď β začíná 0 nebo γ začíná písmenem různým od 0. Pak platí

$$c_i = c_{i-1} + z + |\alpha| + \lambda(0,\gamma) - \lambda(0,\beta) \geq c_{i-1} - 1 + (l_i - q) - q = c_{i-1} + l_i - 2q - 1,$$

kde $z = -1$, když $k_1 = 0$, jinak je $z = 0$, a $\lambda(d,\delta)$ je počet výskytů písmena d ve slově δ. Nerovnost jsme dostali použitím těchto zřejmých vztahů: $|\alpha| = l_i - q$, $0 \leq \lambda(0,\gamma)$ a $\lambda(0,\beta) \leq q$. Pokud dokážeme, že $q \leq 1 + \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor$, pak dosazením tohoto odhadu q do předchozí nerovnosti dostaneme (†). Když slovo β začíná 0, pak označme w poslední vrchol na cestě z vrcholu v určený slovem $k_1\alpha 1$, když slovo γ nezačíná 0, pak označme w poslední vrchol na cestě z vrcholu v určený slovem $k_2 0^{|\alpha|+1}$. Vrchol w je uvnitř mezi cestami z kořene do listů s čísly p_{i-1} a p_i a jeho výška je $q-1$. Proto $p_i - p_{i-1} + 1$ je větší než počet listů v podstromu určeném vrcholem w. Víme, že podstrom určený vrcholem w má alespoň a^{q-1} listů. Odtud plyne, že $q - 1 \leq \log_a(p_i - p_{i-1} + 1)$. Protože $q-1$ je přirozené číslo, je lemma dokázáno. □

Chceme-li odhadnout $l =$ počet vrcholů na cestách z označených listů do kořene, pak v Lemmatu 8.4 položíme $N = n + s$ a $k = s + t$ a dostaneme

$$l \leq 3(s+t) + 2(\lfloor \log_a \frac{n+s}{2} \rfloor + \sum_{i=2}^{s+t} \lfloor \log_a(p_i - p_{i-1} = 1) \rfloor).$$

Když to zkombinujeme s výsledkem Lemmatu 8.3, tak máme

$$St + Sp + P \leq s + 2t + mb(T') \leq s + 2t + l \leq 4s + 5t + 2(\lfloor \log_a \frac{n+s}{2} \rfloor + \sum_{i=2}^{s+t} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor)$$

a důkaz věty 8.2 je kompletní. □

Na závěr odvodíme některé důsledky Věty 8.2. Nejprve si ukážeme zobecnění věty o amortizovaném počtu vyvažovacích operací pro (a,b)-stromy.

Věta 8.5. *Mějme celá čísla a a b taková, že $a \geq 2$ a $b \geq 2a$ a mějme hladinově propojený (a,b)-strom T reprezentující množinu S. Když $\mathcal{P}$ je posloupnost operací* **MEMBER**, **INSERT**, **DELETE** *a* **Prst**, *pak její aplikace na strom T vyžaduje čas*

$$O(\log |S| + \text{totální čas } \mathcal{P} \text{ na vyhledávání}).$$

Poznámka. Všimněme si, že je to nejlepší možný výsledek. Když strom T není přesněji popsán, tak operace **INSERT** nebo **DELETE** mohou vyžadovat až $\log |S|$ vyvažovacích operací, i když samotné vyhledávání mohlo zabrat podstatně méně času (začínáme od pozice $Prst$, nikoliv od kořene stromu). Věta vlastně tvrdí, že v dostatečně krátké době (v $O(\log |S|)$ krocích) se dostaneme do optimálního režimu.

Důkaz. Použijeme Větu 8.2. Když $p_1 < p_2 < \cdots < p_{s+t}$ jsou pozice všech nových prvků a fantomů a posloupnost $\mathcal{P}$ provedla s úspěšných operací **INSERT** a t úspěšných operací **DELETE**, pak

$$St + Sp + P \leq 4s + 5t + 2(\lfloor \log_a \frac{n+s}{2} \rfloor + \sum_{i=2}^{s+t} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor).$$

Protože vyhledávání v každé operaci vyžaduje alespoň jeden krok, tak

$$4s + 5t = O(\text{totální čas na vyhledávání}).$$

Protože $\log_a(n+s) \leq \log_a n + s$, tak i pro druhý člen platí

$$2\lfloor \log_a \frac{n+s}{2} \rfloor = O(\log n + \text{totální čas na vyhledávání}).$$

Zbývá odhadnout třetí výraz. Označme T' výsledný propojený strom s fantomy. Definujme graf G následovně: jeho nosná množina budou všechny listy stromu T', které jsou buď přidané nebo jsou fantomy nebo na ně v průběhu posloupnosti $\mathcal{P}$ ukazoval $Prst$. Hrana grafu bude dvojice $\{p, q\}$ taková, že v některém okamžiku p byl prst a vyhledával se list q. Pro dva listy p a q označíme $l(p, q)$ počet listů stromu T' mezi listy p a q, které nejsou ani nové přidané vrcholy ani fantomy, zvětšený o 1 a číslem $\lfloor \log_a(l(p, q)) \rfloor$ ohodnotíme hranu $\{p, q\}$. Když $\{p, q\}$ je hrana grafu G, pak vyhledávací čas operace, která vedla ke vzniku této hrany, majorizuje hodnotu $\lfloor \log_a(l(p, q)) \rfloor$. Proto totální čas na vyhledávání majorizuje součet ohodnocení všech hran v grafu G. Všimněme si, že graf G je souvislý (každá nová hodnota ukazatele $Prst$ je dosažitelná z jeho počáteční hodnoty). Vezměme napnutou kostru G' grafu G. Pak součet ohodnocení hran v kostře G' je menší nebo roven součtu ohodnocení všech hran v grafu G. Zřejmě kostra G' je strom na množině vrcholů grafu G. Opakujme na grafu G', dokud je to možné, následující proces:

> Předpokládejme, že $\{p, q\}$ je hrana G' a že existuje vrchol r grafu G takový, že v lexikografickém uspořádání indukovaném T' platí $p < r < q$. Protože G' je strom, tak po odstranění hrany $\{p, q\}$ bude mít nově vzniklý graf H právě dvě komponenty a vrcholy p a q budou v různých komponentách grafu H. Když r a p budou ve stejné komponentě grafu H, pak nahradíme v G' hranu $\{p, q\}$ hranou $\{r, q\}$ s ohodnocením $\lfloor \log_a(l(r, q)) \rfloor$, když r a q budou ve stejné komponentě grafu H, pak nahradíme hranu $\{p, q\}$ hranou $\{p, r\}$ s ohodnocením $\lfloor \log_a(l(p, r)) \rfloor$. Nový graf G' bude opět strom na množině vrcholů grafu G a součet ohodnocení všech hran v novém grafu G' bude majorizován součtem ohodnocení všech hran v původním grafu G'.

Předpokládejme, že $r_1, r_2, \ldots, r_k$ jsou všechny vrcholy v grafu G seřazené podle lexikografického uspořádání v T'. Po skončení popsaného procesu je množina hran v grafu G' rovna $\{\{r_i, r_{i+1}\} \mid i = 1, 2, \ldots, k-1\}$ a součet ohodnocení hran je $\sum_{i=1}^{k-1} \lfloor \log_a(l(r_i, r_{i+1})) \rfloor$. To znamená, že G' je řetězec (tj. první a poslední vrchol je incidentní s právě jednou hranou, ostatní vrcholy jsou incidentní právě se dvěma hranami).

Nyní budeme redukovat vrcholy z nosné množiny grafu G. Když r je první vrchol v lexikografickém uspořádání, který není ani přidaný list ani fantom, pak pokud r je incidentní s jedinou hranou G', odstraníme vrchol r i hranu incidentní s r. Nově vzniklý graf G' je opět řetězec na redukované množině a součet ohodnocení hran se nezvětšil. Když r je incidentní se dvěma hranami $\{r, p\}$ a $\{r, q\}$, pak odstraníme vrchol r a hrany $\{r, p\}$ a $\{r, q\}$ nahradíme hranou $\{p, q\}$ s ohodnocením $\lfloor \log_a(l(p, q)) \rfloor$. Nově vzniklý graf G' je rovněž řetězec na redukované množině a protože $\log_a(l(p, r)) + \log_a(l(r, q)) \geq \log_a(l(p, q))$, součet ohodnocení hran se nezvětšil. Po skončení tohoto procesu nosná množina grafu G' je $\{p_1, p_2, \ldots, p_{s+t}\}$, množina hran grafu G' je $\{\{p_i, p_{i+1}\} \mid i = 1, 2, \ldots, s+t-1\}$ a součet ohodnocení hran je $\sum_{i=2}^{s+t} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor$. Protože každý proces zmenšil součet ohodnocení všech hran v

grafu a na počátku totální vyhledávací čas majorizoval první součet ohodnocení všech hran, dostáváme, že třetí člen je majorizován totálním vyhledávacím časem. □

Poznámka. Když $s+t > n$, pak lze přímo dokázat, že $2(s+t)$ majorizuje třetí člen. Problém je, když $s + t$ je malé.

Ukážeme si druhý důsledek.

Věta 8.6. *Předpokládejme, že $a \geq 2$ a $b \geq 2a$. Když S_1 a S_2 jsou podmnožiny univerza reprezentované hladinově propojenými (a, b)-stromy T_1 a T_2, pak platí:*

(1) *operace* **MEMBER**(x, T_1), **INSERT**(x, T_1), **DELETE**(x, T_1) *a* **SPLIT**(x, T_1) *lze realizovat v čase $O(\log |S_1|)$;*
(2) *když $\max S_1 < \min S_2$, pak operaci* **JOIN2**(T_1, T_2) *lze realizovat v čase $O(\log |S_1 \cup S_2|)$;*
(3) *když $m = \min\{|S_1|, |S_2|\}$ a $n = \max\{|S_1|, |S_2|\}$, pak konstrukce hladinově propojených (a, b)-stromů reprezentujících množiny $S_1 \cup S_2$, $\Delta(S_1, S_2) = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$, $S_1 \cap S_2$ nebo $S_1 \setminus S_2$ vyžaduje čas $O(\log \binom{n+m}{m})$.*

Důkaz. K důkazu (1) a (2) použijeme postup z klasických (a, b)-stromů.

Popíšeme algoritmy konstruující množiny z tvrzení (3). Předpokládejme, že $|S_1| = n$ a $|S_2| = m$. Nastavíme $Prst$ na první prvek v T_1 a začneme procházet prvky z S_2 od prvního prvku, který označíme y. Vyhledáme y v T_1. Při operaci $S_1 \cup S_2$ vložíme y do T_1 (pokud tam není) a nastavíme $Prst$ na y. Při operaci $S_1 \setminus S_2$ odstraníme y z T_1 (pokud tam je) a $Prst$ nastavíme na jeho následníka. Při operaci $\Delta(S_1, S_2)$, když $y \in S_1$, tak ho odstraníme a $Prst$ nastavíme na jeho následníka, když $y \notin S$, tak ho vložíme a $Prst$ nastavíme na něho. Pak pokračujeme dalším prvkem v S_2 (následníkem y). V realizaci operace $S_1 \cap S_2$ postupně prohledáváme oba stromy a prvky z $S_1 \cap S_2$ vkládáme do nového stromu T. Přesněji, nechť x je první prvek v S_1, y je první prvek v S_2 a T reprezentuje prázdnou množinu. Dokud $x \neq NIL$ a $y \neq NIL$, budeme provádět následující krok:

> když $x = y$, pak vložíme x do T, $x :=$ následník x v S_1, $y :=$ následník y v S_2;
> když $x < y$, pak provedeme **Prst**(y, T_1), $x := \text{key}(Prst)$ v T_1 a když $x < y$, pak $x :=$ následník x v S_1;
> když $x > y$, pak provedeme **Prst**(x, T_2), $y := \text{key}(Prst)$ v T_2 a když $x > y$, pak $y :=$ následník y v S_2.

Předpokládáme, že seznamy jsou ukončeny hodnotou NIL.

Z Věty 8.2 plyne, že algoritmy vyžadují čas $O(m+\log(m+n)+\sum_{i=2}^{m}\lfloor\log(p_i-p_{i-1}+1)\rfloor)$, a podle důkazu Věty 8.5 je $\sum_{i=2}^{m}\lfloor\log(p_i-p_{i-1}+1)\rfloor$ omezeno totálním časem pro vyhledávání. Totální čas na vyhledávání je maximalizován, když $p_i - p_{i-1} = \frac{n+m}{n}$ pro každé i, a celkový čas potom je

$$O(\log(n + m) + m \log(\frac{n + m}{m})) = O(m \log(\frac{n + m}{m})) = O(\log \binom{n + m}{m}).$$

Ještě poznámka: když $|S_2| < |S_1|$, pak při realizaci $S_2 \setminus S_1$ postupujeme stejně, jen v případě, že nalezneme y (z S_2) ve stromě T_1, tak y odstraníme ze stromu T_2 (který bude reprezentovat $S_2 \setminus S_1$). □

Tyto výsledky dokázali Huddleston a Mehlhorn v roce 1982.