

# Implementace červeno-černých stromů v Prologu

## Teorie

**Definice:** Červeno-černý strom (RB-tree) je binární vyhledávací strom (BST), pro který navíc platí tato pravidla:

- (RB1) Každý uzel je buď červený nebo černý.
- (RB2) Kořen je černý.
- (RB3) Dva červené uzly se nesmí vyskytovat "nad sebou", tj. červený uzel má jediné černé potomky.
- (RB4) Na cestě od kořene do libovolného uzlu s jedním nebo žádným synem je stejný počet černých uzlů. (Vnější uzly, nil, pokládáme za černé.)

**Poznámka:** Existují i jiné podobné definice červeno-černých stromů (např. požadují černé listy), ale není těžké mezi nimi najít jednoznačný převod. Tuto definici jsem si vybrala proto, že požadavek černého kořenu se splňuje poněkud lépe než např. požadavek černých listů.

**Věta:** Nejhorší možná hloubka červeno-černého stromu s *N* vrcholy je *O*(2\*log*N*).

**Důkaz:** Vidíme, že nejdelší možná cesta z kořene do listu by se skládala střídavě z černých a červených uzlů *O*(2\*log*N*), zatímco nejlepší možná pouze z černých *O*(log*N*).

**Vyhledávání v RB-tree:** Protože RB-tree je vlastně BST, vyhledávání funguje naprosto stejně. Jeho nejhorší možná časová složitost je lineární s hloubkou stromu, tj. *O*(2\*log*N*).

**Insert v RB-tree:** Jako v BST nalezneme nejprve místo, kam má vkládaný prvek patřit, vložíme jej a obarvíme na červeno. Obarvením na černo bychom automaticky vytvořili problém s (RB4). Pokud je otec vloženého vrcholu černý, neporušíme žádnou podmínku. Pokud je červený, je nutno vyřešit jednu z těchto čtyř možností, jak jsou popsány na <http://www.misenfricken.com/alg/insert.html>. Jejím vyřešením můžeme způsobit problém na vyšší úrovni, po insertu je tedy nutné zkontrolovat strom na cestě zpět do kořene a opravit jej.

**Delete v RB-tree:** Opět řešíme stejně jako v BST. Pokud je uzel, který chceme smazat, červený, neporušíme žádnou podmínku. V opačném případě je nutné vyřešit možné problémy popsané na <http://www.misenfricken.com/alg/delete.html>.

## Zajímavosti implementace RB-tree v Prologu:

Find, insert a delete fungují (jak by se dalo čekat), rekurzivně. Hloubka této jednoduché rekurze je lineární s hloubkou stromu, tedy *O*(log*N*).

Operace find, insert a delete nad zadaným stromem se řídí daným komparátorem, který musí splňovat vlastnosti reflexivní, tranzitivní a slabě antisymetrické relace (tj. *X* comp *Y* & *Y* comp *X* => *X* = *Y*).

Při realizaci insert a delete bylo nutno vyřešit dva základní problémy: Prvním bylo opakování téměř totožného kódu pro zrcadlové případy a druhým přeskupování, resp. přebarvování prvků při opravách. Obojí řeší predikát *rozeber/6*, který zadaný strom rozebere podle zadaného směru (levý, pravý) na kořen, barvu kořene a levý a pravý podstrom (resp. zrcadlově obráceně). Tento predikát ale funguje i opačně, tedy díky prologovské unifikaci dokáže naopak poskládat výsledný strom ze zadaným komponent v požadovaném směru. Při řešení všech případů, které při insert a delete mohou nastat, tedy stačí strom vhodně rozebrat a zase postavit predikátem *rozeber/6*.

Při implementaci delete jsem narazila na potíž, pokud mazaný prvek měl dva podstromy. V takovém případě je nutné najít největší prvek v levém podstromě, vyměnit hodnoty prvků a smazat onen náhradní prvek, který je nyní zaručeně listem nebo má jen jednoho syna. Nedokázala jsem ale tohle provést jenom s obyčejnou rekurzí, takže jsem mazání provedla dvoufázově. Nejprve se nalezne prvek ke smazání, pokud má dva syny, provede se výměna jejich obsahů a oprava podstromu až do úrovně původního mazaného prvků (predikát *makered/5*). Poté se prvek s hodnotou, kterou chceme smazat, nalezne ještě jednou, smaže se a dokončí se opravy až do kořene stromu. Časová složitost se asymptoticky nijak nezmění, mazání trvá *O*(2\*log*N*). Výměnou prvků jsme samozřejmě porušili vlastnost binárního vyhledávacího stromu, takže je nutné si pamatovat, v kterém místě je oproti správnému binárnímu stromu "chyba" a při druhém vyhledávání hodnoty ke smazání vyhledávat v opačném podstromě (argument *F* v predikátu *del/5*).

## Jak používat RB-tree

t/4

t(+Root,+Colour,+Left,+Right) - rekurzivní reprezentace stromu, Root - kořen stromu, Colour - barva kořene stromu (b - černá, r - červená), Left - levý podstrom, Right - pravý podstrom. Prázdný podstrom se značí atomem nil.

find/3

find(+X,+Comp,+Tree) - uspěje, pokud strom Tree obsahuje prvek X podle komparátoru Comp

insert/4

insert(+X,+Comp,+Tree,-NewTree) - vkládá prvek X do stromu Tree podle komparátoru Comp a výsledek vrací ve stromě NewTree. pokud již Tree obsahuje X, nedělá nic a uspěje.

insertl/2

insertl(+List,+Comp,-NewTree) - vkládá prvky ze seznamu a výsledek vrací ve stromu NewTree, jako komparátor používá Comp.

delete/4

delete(+X,+Comp,+Tree,-NewTree) - maže prvek X ze stromu Tree podle komparátoru Comp a výsledek vrací ve stromě NewTree, pokud Tree neobsahuje X, vrací původní strom a uspěje.

deletel/3

deletel(+List,+Comp,+Tree,-NewTree) - maže prvky ze seznamu ze stromu Tree a výsledek vrací ve stromu NewTree, jako komparátor používá Comp.

seq/3

seq(+From,+To,-List) - vytvoří seznam integerů v rozsahu [From,To] a vrací je v seznamu List

randlist/3

randlist(+N,+M,-List) - vytvoří seznam N náhodných integerů v rozsahu [0,M) a vrací je v seznamu List.

tracetreeon/0

tracetreeon - spustí krokování všech akcí

tracetreeoff/0

tracetreeoff - zastaví krokování všech akcí

show/1

show(+Tree) - vypíše strom

## Zdrojový kód a příklady použití RB-tree

- [rb.pl](#) - zdrojový kód
- [rb\\_in.pl](#) - příklady vstupů