VYHLEDÁVÁNÍ VE VYVÁŽENÝCH STROMECH I.

Jan M Honzík

ABSTRAKT

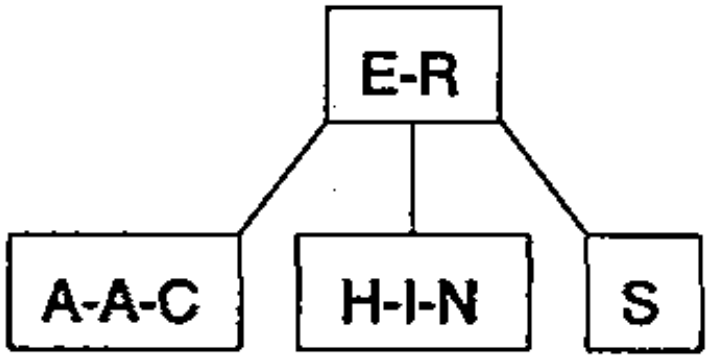
Binární vyhledávací stromy mohou při své degeneraci dosáhnout doby vyhledávání o složitosti O(N). Proto se všude tam, kde je tento nejhorší případ nepřijatelný, používají tzv. vyvážené stromy, které zaručí lepší dobu vyhledávání, nejčastěji o složitosti O(ln N). Po AVL stromech, popsaných např. v [Hon], se začaly používat složitější stromy, umožňující rychlejší korekci porušené rovnováhy vyhledávacího stromu. Mezi ně patří především 2–3–4–stromy a červeno–černé stromy, uvedené v tomto příspěvku. Jiný pohled na červeno–černé stromy, spolu s uvedením algoritmu pro operaci Delete a také zajímavé vlastnosti a operace tzv. „splay–stromu", překročily svým rozsahem prostor dostupný ve sborníku a budou nabídnuty k pokračování v tematu na příští konferenci Programování '94.

1 2–3–4 STROMY

2–3–4 strom je N–ární kořenový strom, pro 2≤N≤4. To znamená, že strom má uzly s 1,2 nebo 3 klíči; podle toho kolik klíčů má, vychází z něj 2,3 nebo 4 hrany k uzlům na nižší úrovni. 2–strom je vlastně binární strom. 3–uzel má dva klíče; nejlevější hrana ukazuje na podstrom, který obsahuje klíče menší než levý (první) klíč; prostřední hrana ukazuje na podstrom obsahující klíče větší než první klíč a menší než druhý klíč; pravá hrana ukazuje na podstrom, který obsahuje klíče větší než druhý klíč. Podobně je konstruován 4–uzel. Z této struktury je zřejmý mechanismus vyhledávání, který vyhledává na cestě počínaje kořenem a konče nalezením uzlu s hledaným klíčem nebo dosažením konce větve při neúspěšném vyhledávání, které může být následováno vkládáním nového uzlu. Operace vkládání může připojit nový uzel jako terminální uzel – list na konec vyhledávací cesty, pod terminální uzel. Výhodnější je ale vložit novou položku dovnitř listu, je-li to 2–uzel nebo 3–uzel, a v případě 4–uzlu rozdělit jej na tři 2–uzly, a střední uzel vložit do otcovskému uzlu (není-li to také 4–uzel). Do takto rozloženého uzlu lze na odpovídající místo vložit novou položku. Kdyby otcovský uzel byl také 4–uzel, opakovalo by se rozdělování směrem ke kořeni, dokud by otec byl 4–uzel. Aby se tomu zabránilo zajistíme, že žádný „otec" nebude 4–uzel tím, že všechny 4–uzly při vyhledávací cestě operace Insert stromem dolů rozdělíme. V uvedené metodě i v příkladech se předpokládá, že do datové struktury pro vyhledávání můžeme ukládat více položek se shodnými klíči.

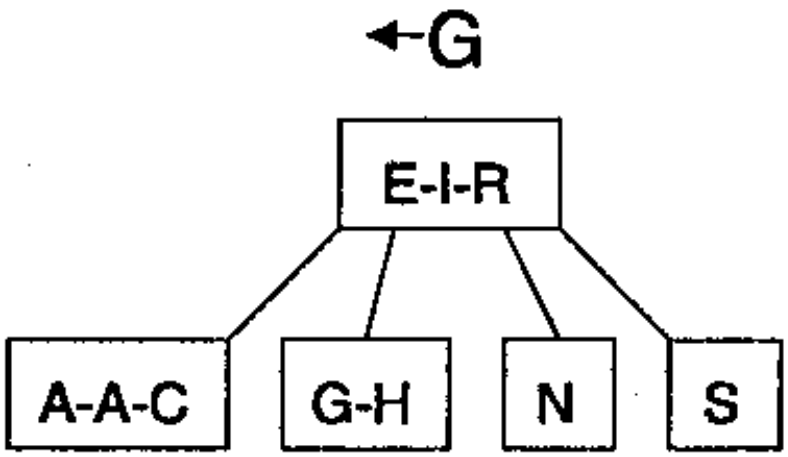
Příklad:

Strom obsahuje znaky „A S E A R C H I N" a vypadá takto:

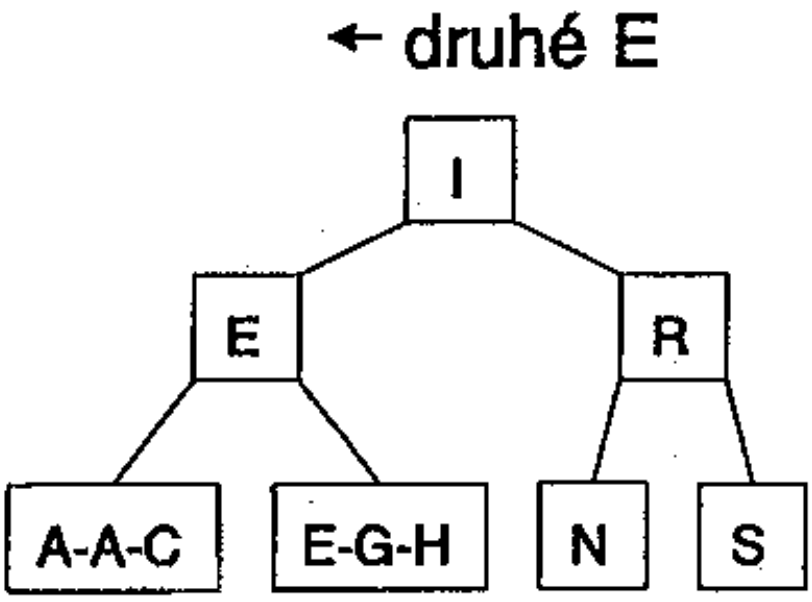


Obr. 1 2-3-4 strom

Postupně se dodají klíče: „G E X A M P L E"

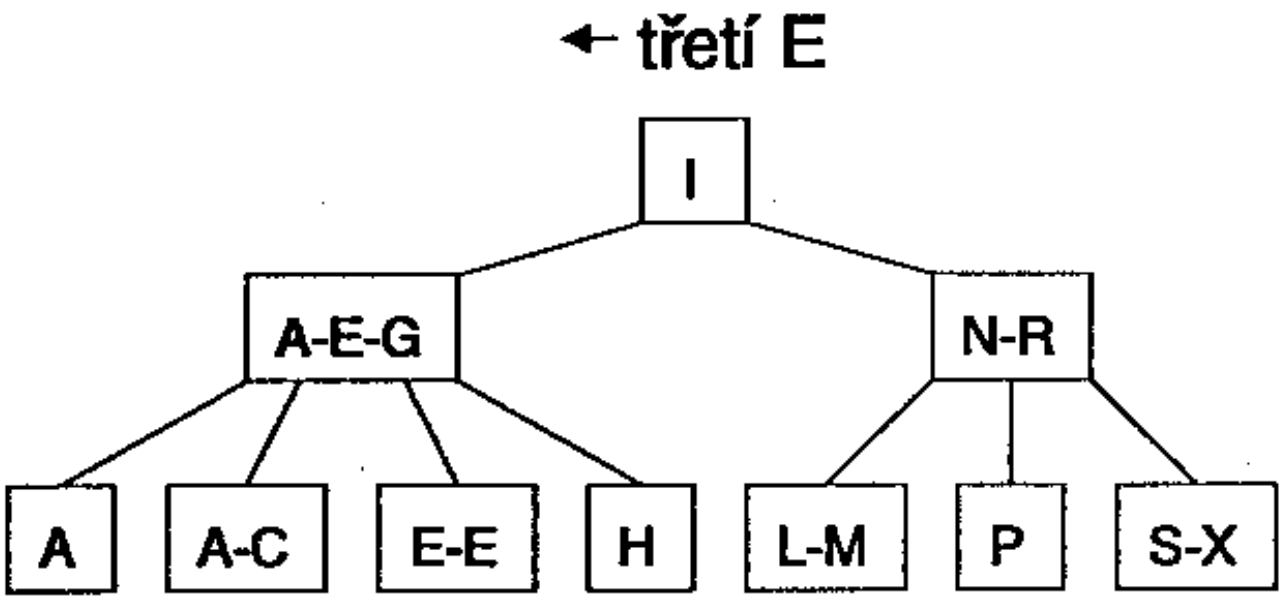


Obr. 2 Vloženo G



Obr. 3 Vloženo druhé E

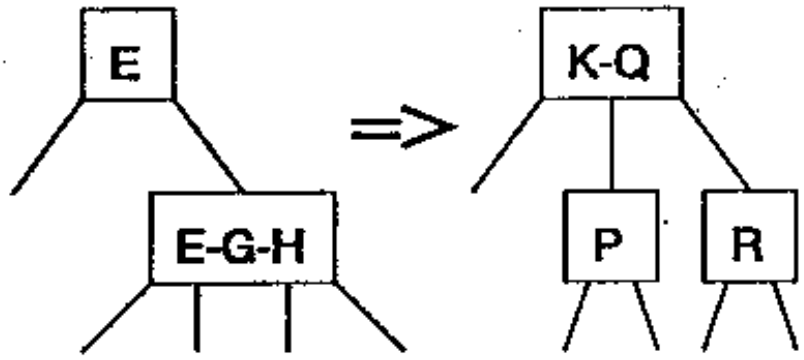
Vložení XAMPL pro nedostatek prostoru v referátu vynecháme a následuje:



Obr. 4 Vloženo třetí E

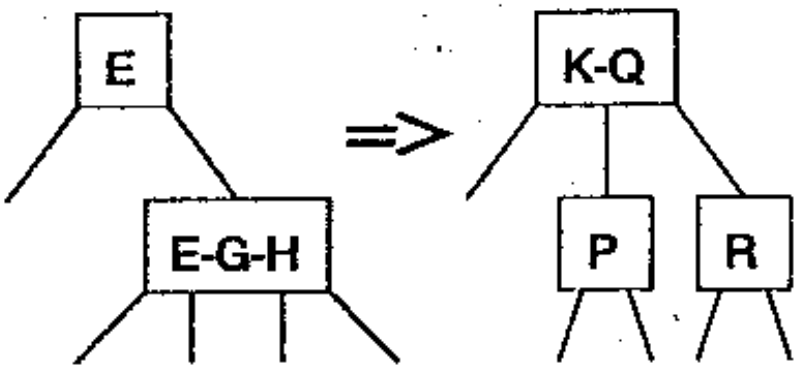
Mechanismus rozdělení 4–uzlu:

Když má 4–uzel za „otce" 2–uzel, provede se rozdělení 4–uzlu takto:



Obr. 5 Dělení 4-uzlu s 2-otcem

Když má 4–uzel za otce 3–uzel pak má schéma rozdělení tvar:



Obr. 6 Dělení 4-uzlu s 3-otcem

Transformace při rozdělování je zcela lokální. V okolí se nedějí žádné změny než ty, uvedené na obr.5 a 6. Všimněme si, že se ve skutečnosti nemusíme starat o to, zda je otcovský uzel 4–uzel, protože transformace zajišťuje, že při průchodu dolů každým uzlem stromu, vycházíme dále po větvi z uzlu, který již není 4–uzel. Projdeme-li až na dno a konečný uzel není 4–uzel, můžeme klíč vložit do uzlu a z 2–uzlu vytvořit 3–uzel, nebo z 3–uzlu 4–uzel. Praktičtější je provádět vkládání tak, jako by šlo o rozdělení virtuálního 4–listu, který vysílá vkládaný klíč naboru, do rodičovského uzlu. Kdykoli se kořen stromu stane 4–uzlem, rozdělí se na tři 2–uzly, jak to bylo na obr.3 po vložení klíče E. Je to jediný mechanismus, kterým se zvýší výška 2–3–4 stromu. Vzdálenost kořene od všech terminálních uzlů je stejná a tato vzdálenost se nemění vložením nového uzlu. Výjimkou je rozdělování kořene, kdy se vzdálenost kořene od všech terminálních uzlů zvýší o 1. Významnou skutečností je, že ačkoliv se nestaráme o vyváženost stromu, výsledkem uvedeného mechanismu vkládání je vyvážený strom.

Nejvýznamnější vlastností 2–3–4–stromu je, že při vyhledávání ve stromu o N uzlech se projde maximálně lg(N)+1 uzlů.

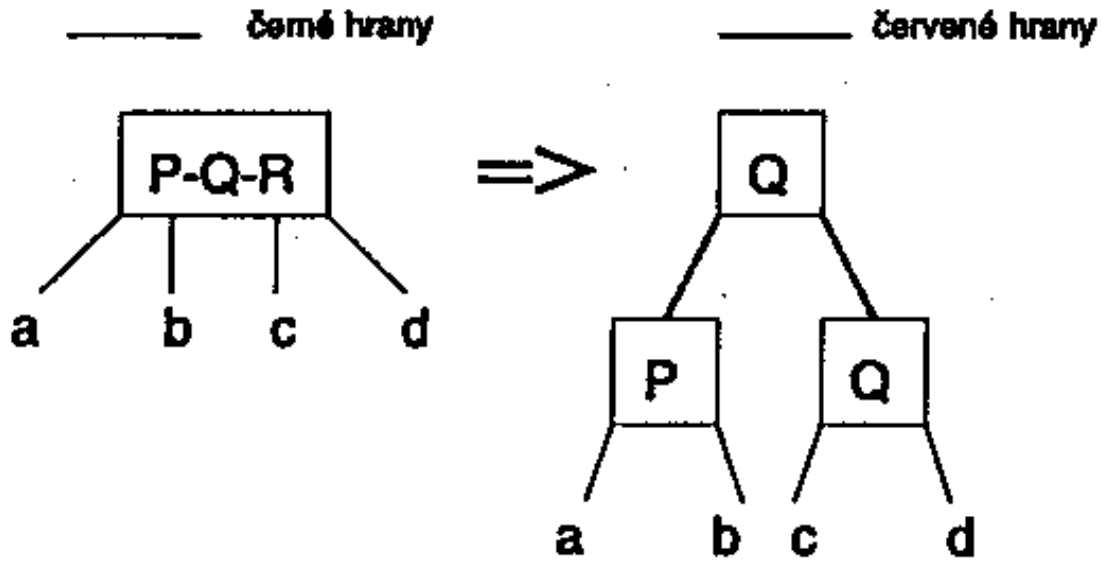
Další významnou vlastností je skutečnost, že vkládání do 2–3–4–stromu o N uzlech vyžaduje v nejhorším případě méně než lg(N)+1 dělení uzlů;

V průměru je to méně než jedno dělení uzlů na operaci Insert. Nejhorší případ nastane, když na cestě ke klíčí jsou všechny uzly typu 4–uzel, a všechny by se měly rozdělit. V [Sed] je uveden ilustrační příklad stromu o 95 náhodných hodnotách, kde je devět 4–uzlů, z nichž jen jeden je na první úrovni nad listem. Uvedený popis se zdá být dostatečný k implementaci algoritmu pro práci s 2–3–4–stromem. Na první pohled se zdá, že zbývá jen vytvořit transformační procedury pro jednotlivé typy uzlů; odpovídající operace však nejsou tak snadně naprogramovatelné, jak snadno se o nich hovoří. Při manipulaci se složitější strukturou se navíc objeví dodatečná časová režie, která způsobí, že algoritmus je pomalejší než u binárního stromu (2–stromu). Primárním účelem vyvažování bylo zajištění se proti nejhoršímu případu, ale zdá se být neúnosné platit za toto zajištění při každém vkládání. Jak bude uvedeno v dalším odstavci, existuje řešení relativně jednoduchou representací 3– a 4– uzlů prostřednictvím 2–uzlů (červeno– černého binárního stromu), která dovoluje jednotný způsob transformace s časovou režií menší, než u standardních metod vyhledávání v binárním stromu.

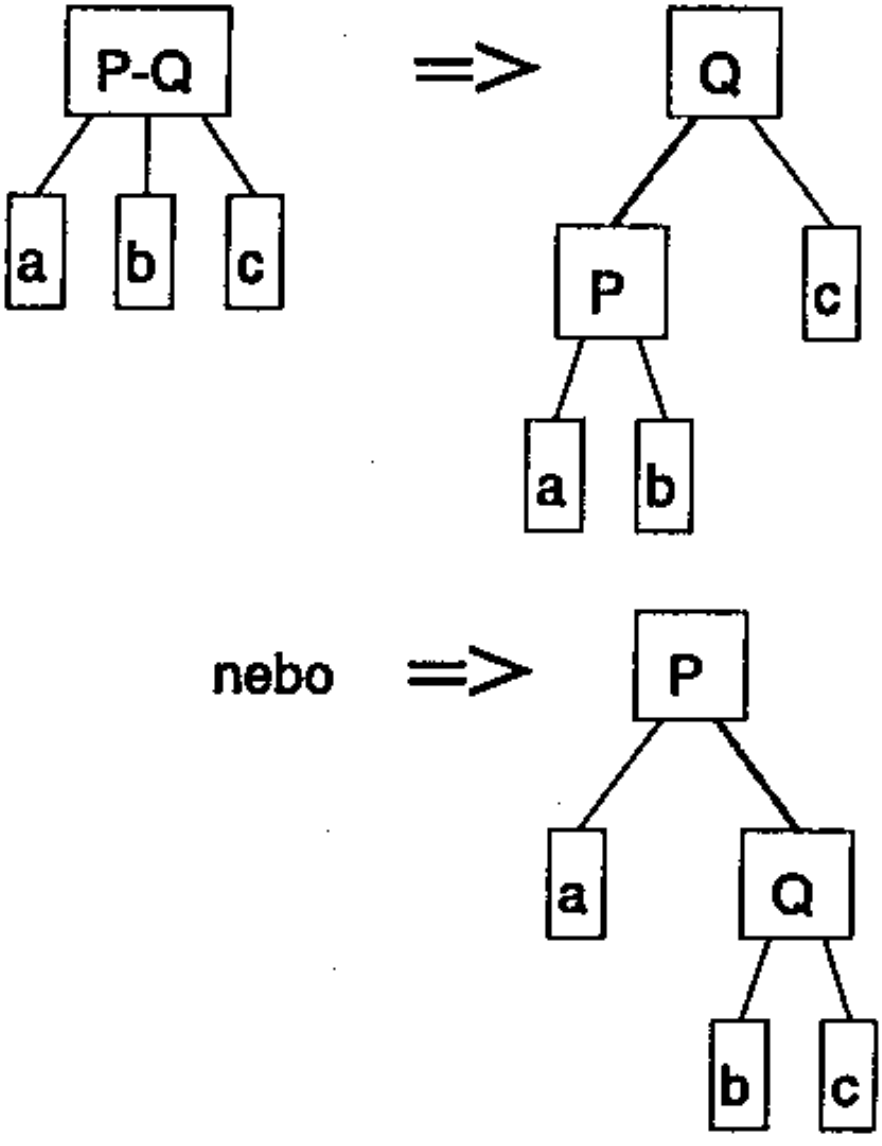
2 ČERVENO–ČERNÉ STROMY (RED–BLACK–TREES)

2–3–4–uzel lze implementovat pomocí standardních binárních uzlů. Hrany se rozliší na ty, které „drží 3–uzel nebo 4–uzel pohromadě" (červené) a na ty ostatní (černé).

Příklad:

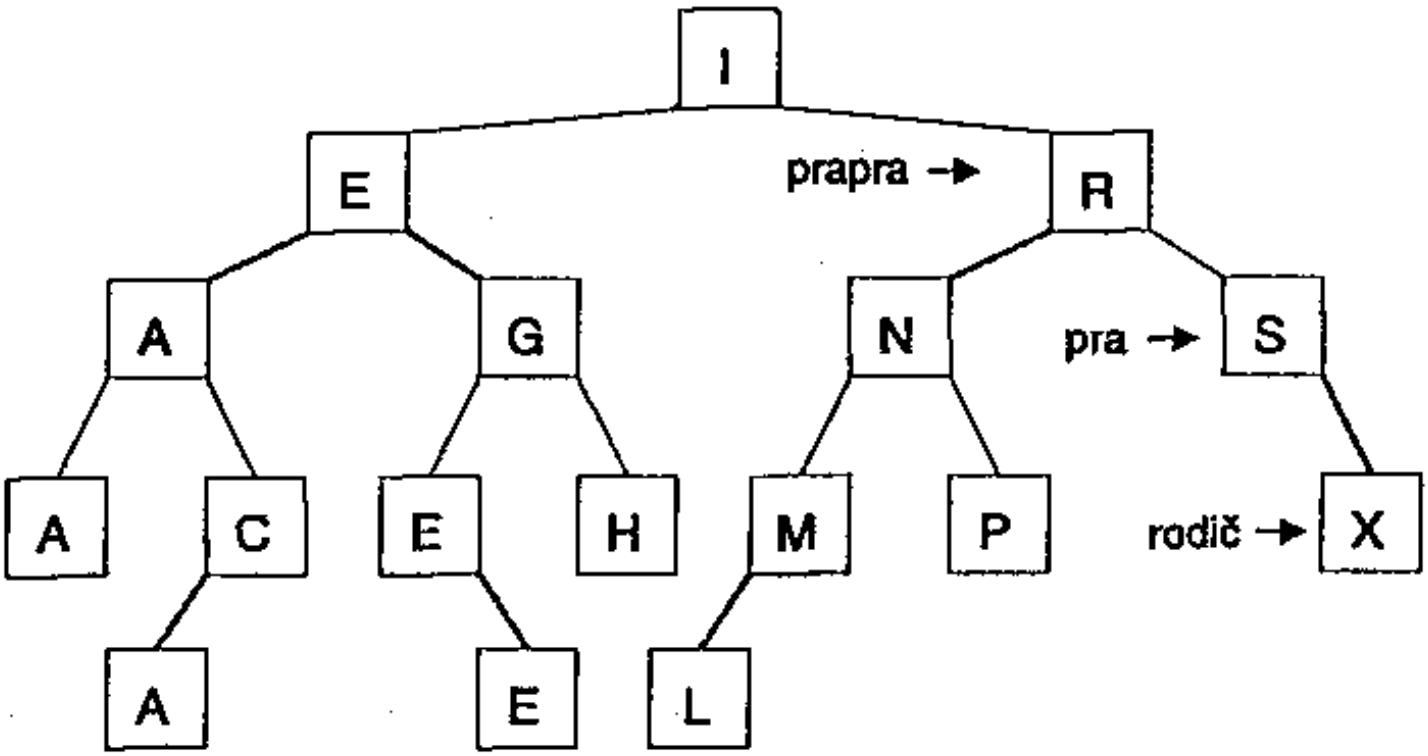


Obr. 7 Implementace 4-uzlu pomocí červeno-černých uzlů

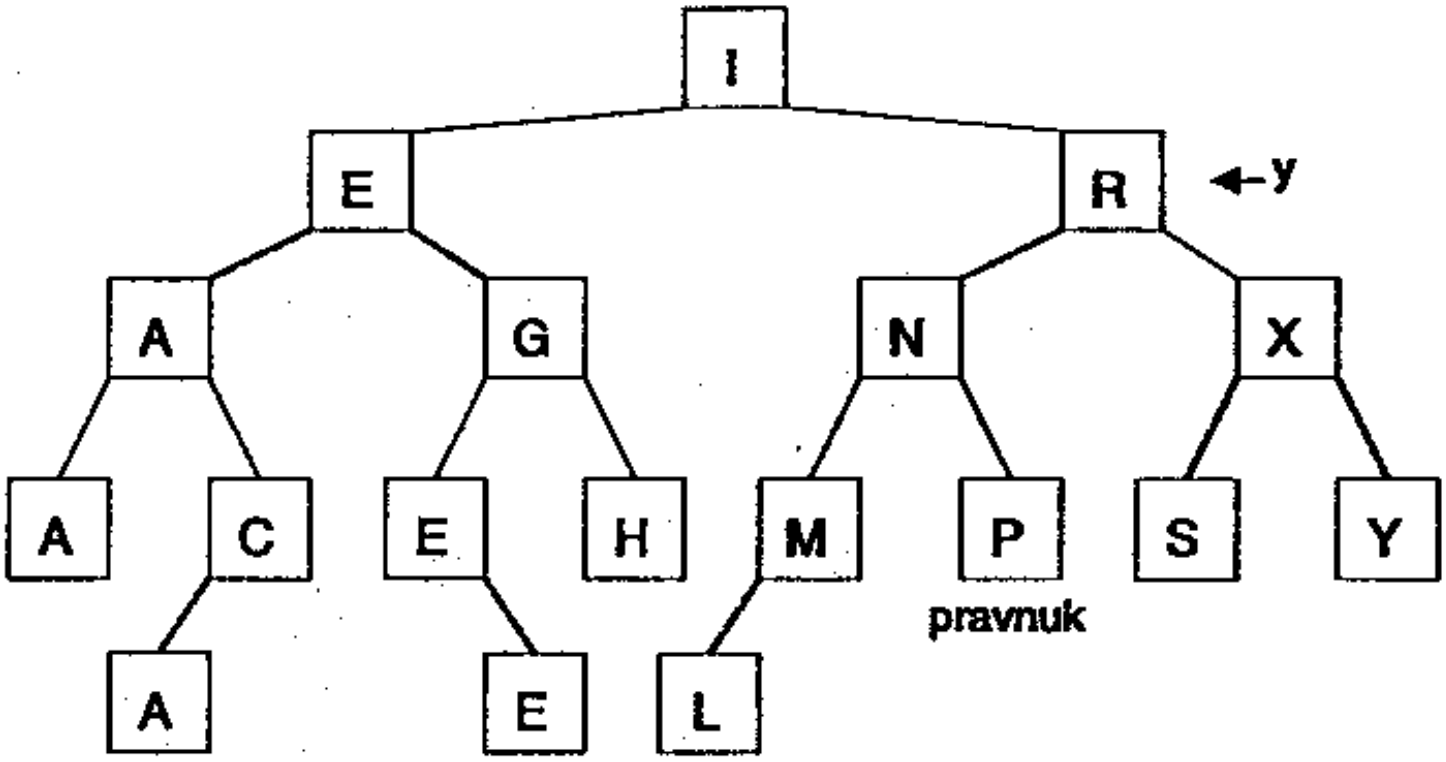


Obr. 8 Implementace 3-uzlu pomocí červeno-černých uzlů

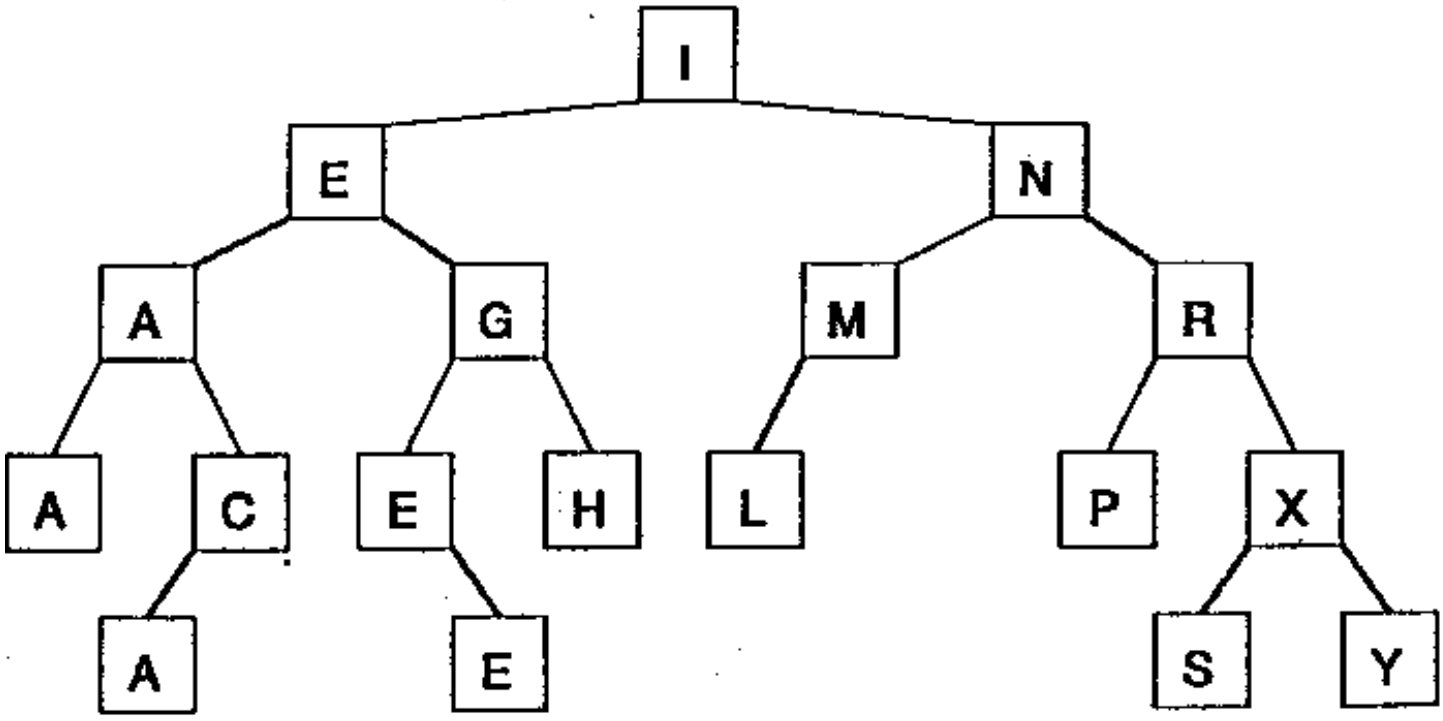
Pro označení barvy hrany červeno–černého stromu (RB stromu) lze použít jeden bit uzlu, který označuje jaká hrana vstupuje do uzlu. Varianta transformace 3–uzlu na levou či pravou versi závisí na dynamice algoritmu. Důležitou vlastností je umisťování duplicitních klíčů v případě, že definice vyhledávací tabulky podobnou vlastnost dovoluje. Z požadavku na vyváženost je zřejmé, že algoritmus musí umožnit, aby shodný klíč padnul vlevo nebo vpravo; jinak by to vedlo v případě řetězu shodných klíčů k vážnému porušení vyváženosti . Z toho vyplývá, že nemůžeme nalézt všechny uzly opakovaným voláním procedury vyhledání. To ale není složitý problém, protože všechny klíče, shodné s nalezeným klíčem jsou v podstromu, jehož kořenem je nalezený klíč. Pro jejich nalezení můžeme použít jednoduchý rekursivní algoritmus průchodu. Příjemnou vlastností RB stromů je, že vyhledávání je (s výjimkou uvedeného případu) zcela shodné se standardním vyhledáváním v binárním stromu. Barva vstupní hrany do uzlu bude implementována složkou uzlu „red", která je typu Boolean. Algoritmus vyhledávání tuto složku vůbec nepoužívá a tudíž se neprojeví žádné zvýšení časové náročnosti vyhledávání. Vzhledem k tomu, že každý uzel je vkládán jen jednou, ale vyhledáván mnohokrát, jeví se algoritmus, který optimalizuje vyhledávání za cenu zvýšené režie při vkládání za výhodný. Ani režie vkládání není velká. Trochu složitější je případ 4–uzlu, ale ve stromu není mnoho 4–uzlů, neboť algoritmu je má snahu neustále dělit. Vnitřní cyklus dělá jen jeden test navíc (jestliže uzel má dva červené synovské uzly je prvkem 4 uzlu). Algoritmus vkládání má tvar:



Obr. 9a Červeno-černý strom před vložením Y (viz Rotate)

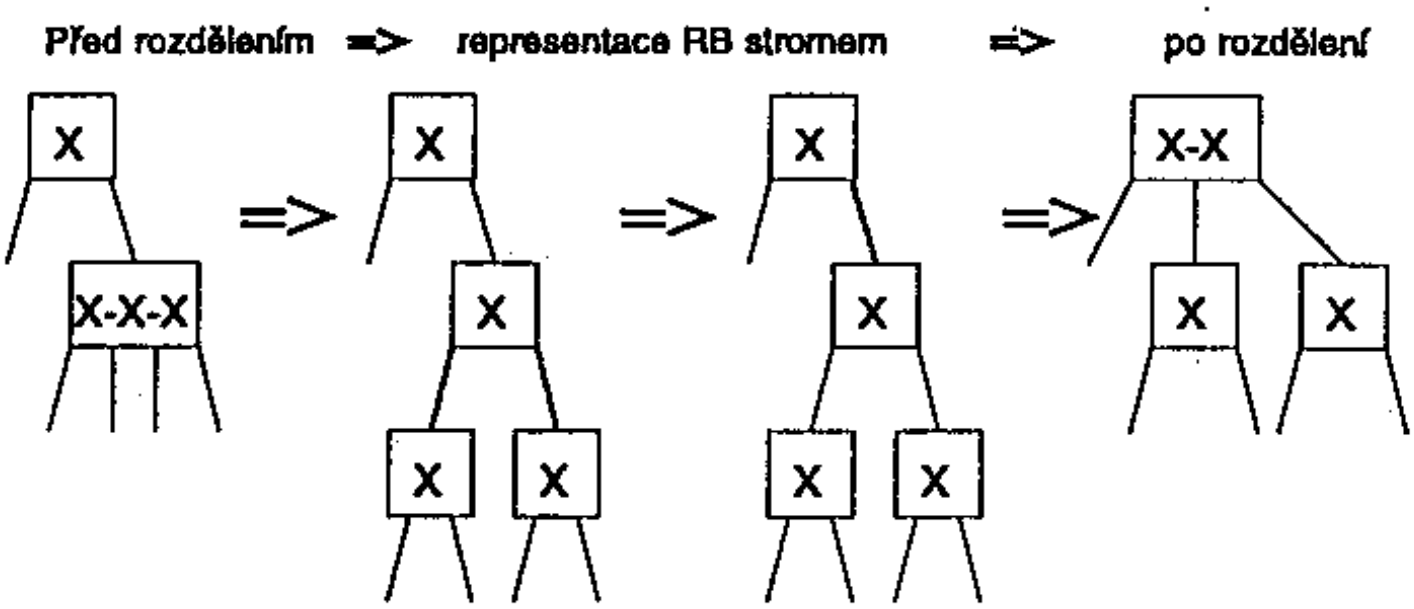


Obr. 9b Červeno-černý strom po vložení Y (viz Rotate)

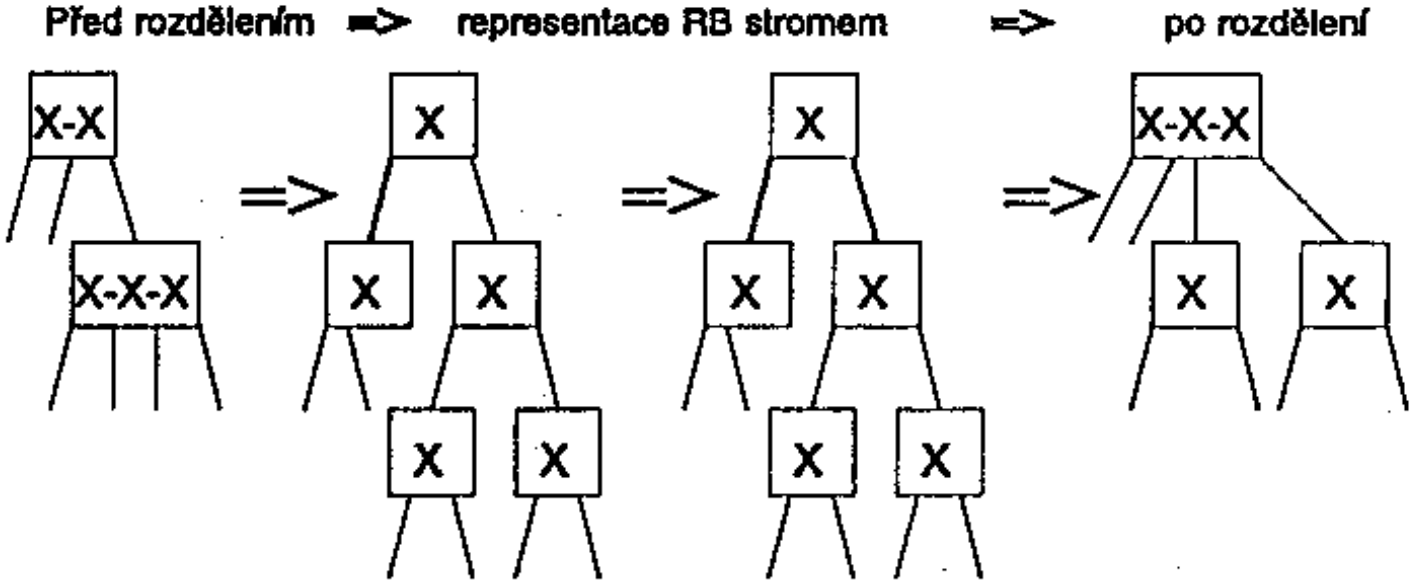


Obr. 9c Červeno-černý strom po rotaci kol R (viz Rotate)

```
function RedBlackTreeInsert(K:integer; Root]:TPtr):TPtr;
        (* funkce vrací ukazatel na vložený uzel s klíčem K *)

var
    prapra,     (* praprarodič uzlu Root *)
    pra,        (* prarodič uzlu Root *)
    rodic:TPtr; (* rodič uzlu Root *)

begin
    rodic:=Root;     (* počáteční nastavení ukazatelů na kořen *)
    pra:=Root;

    repeat
        prapra:=pra;     (* Posun praprarodiče dolů *)
        pra:=rodic;      (* Posun prarodiče dolů *)
        rodic:=Root;     (* Posun rodiče dolů *)

        if K < Root^.key
        then Root:=Root^.LPtr       (* posun doleva nebo doprava *)
        else Root:=Root^.RPtr;

        if Root^.LPtr^.red and Root^.RPtr^.red
        then Root:=split(K,prapra,pra,rodic,Root);
                        (* cestou dolů dělíme 4-uzly *)
    until Root=z;       (* z je pomocný uzel s funkcí nilu *)

    new(Root);          (* vytvoření nového uzlu *)

    Root^.key:=K;
    Root^.LPtr:=z;      (* levý ukazatel je nil *)
    Root^.RPtr:=z;      (* pravý ukazatel je nil *)

    if K < rodic^.key (     * připojení nového uzlu na rodičovský uzel *)
    then rodic^.LPtr:=Root
    else rodic^.RPtr:=Root;

    RedBlackTreeInsert:=Root;  (* vrací se ukazatel na nový uzel *)

    Root:=split(K,prapra,pra,rodic,Root);       (* dělení virtuálního 4-uzlu *)
end;
```

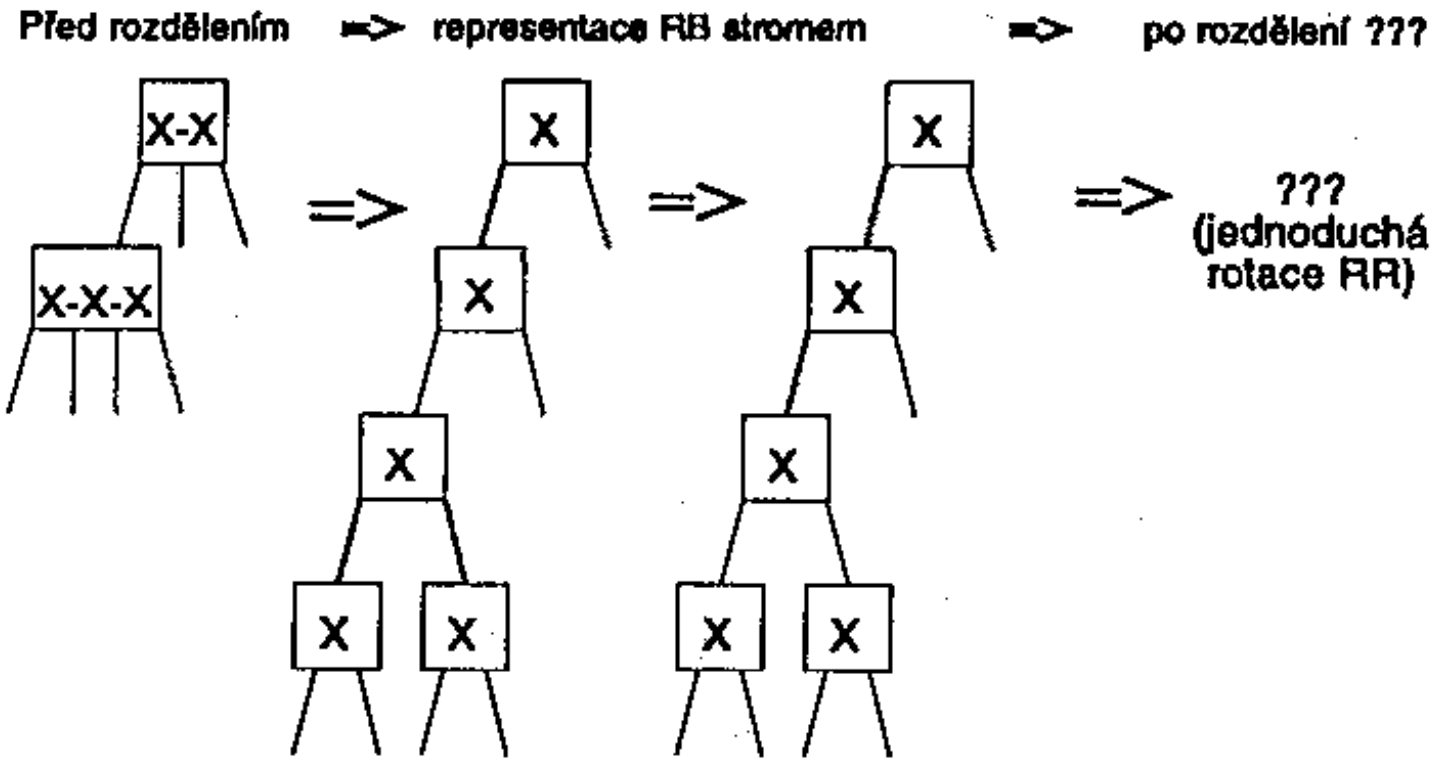


Obr. 10 Rozdělování 4-uzlu připojeného k 2-uzlu



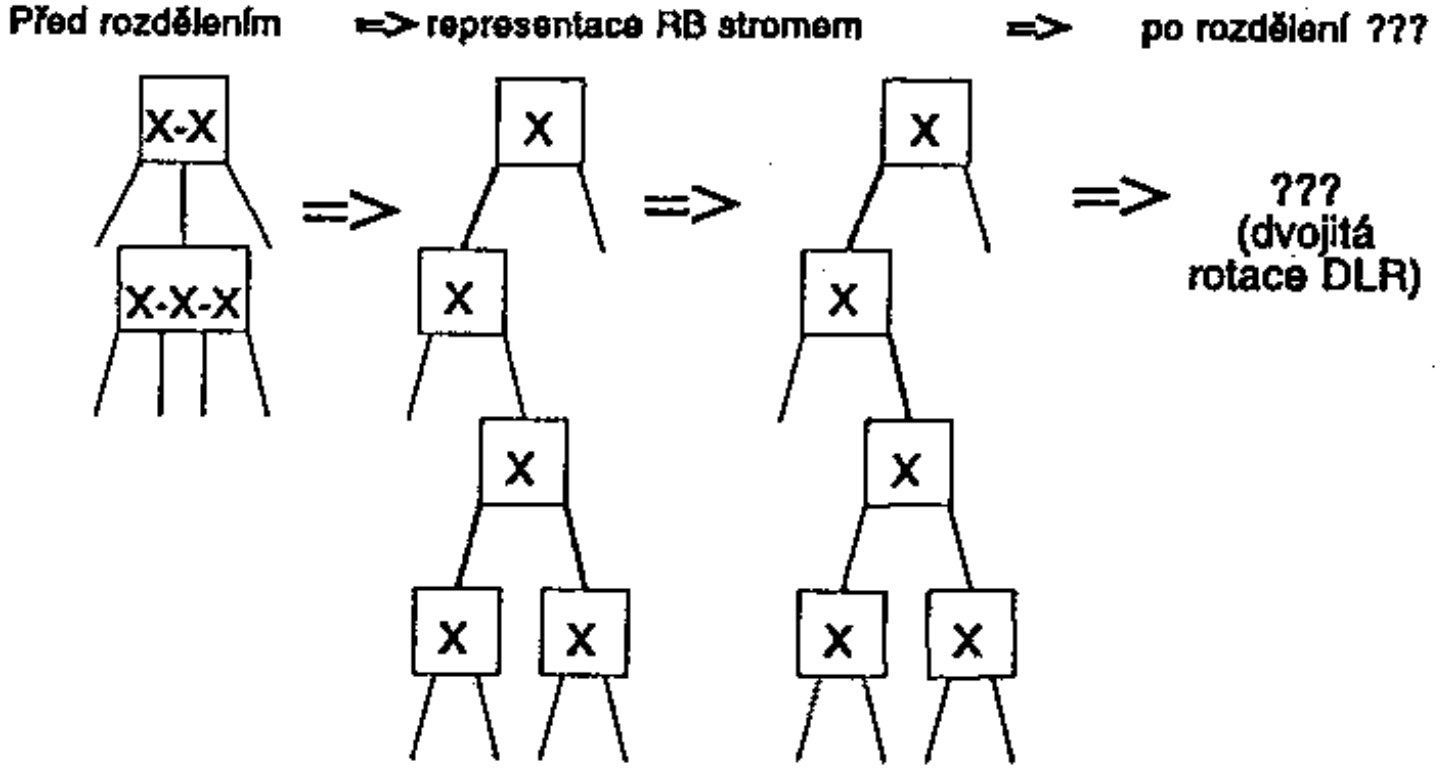
Obr. 11 Rozdělování 4-uzlu připojeného k 3-uzlu vpravo

Tyto situace jsou zcela zřejmé. Existují však dvě situace (a jejich dvě zrcadlové varianty), které vyžadují dodatečnou úpravu – rotaci. Jsou to připojení 4–uzlu ke 3–uzlu zleva a zespodu, což ilustrují následující ukázky:



Obr. 12 Rozdělování 4-uzlu připojeného na 3-uzel zleva

Dvě červené hrany v sérii nejsou v RB stromech povoleny. Jejich výskyt se odstraní rotací. Podobná situace nastane při rozdělení 4–uzlu připojeného na 3–uzel zespodu:



Obr. 13 Rozdělování 4-uzlu připojeného na 3-uzel zleva

Situaci na obr.12 řeší jednoduchá rotace. Její algoritmus je uveden ve funkci rotate. Ukazuje-li y na kořen, pak je c pravý ukazatel uzlu y a gc je levý ukazatel uzlu c. Funkce vrátí ukazatel na vrcholový 3–uzel stromu, ale nemění barvy.

```
function rotate(K:integer; y:TPtr):TPtr;
      (* K je zadaný klíč uzlu a y je ukazatel na jeho otcovský uzel;
      funkce vrací ukazatel na 3–uzel, ale nemění barvu hran;
      Situaci ilustrují obr.9.a,b,c *)

var   syn,              (* synovský uzel
      vnuk:TPtr;        (* vnukovský uzel *)

begin
      if K < y^.key           (* je zadaný klíč menší než klíč otce ? *)
      then vnuk:=y^.LPtr      (* pak vnuk je levý uzel otce *)
      else vnuk:=y^.RPtr;     (* pak vnuk je pravý uzel otce *)

      if K < c^.key                   (* je zadaný klíč menší než klíč vnuka ? *)
      then begin
            vnuk:=syn^.LPtr;          (* rotace doprava *)
            syn^.LPtr:=vnuk^.RPtr;
            vnuk^.RPtr:=syn
      end else begin                  (* rotace doleva *)
            vnuk:=syn^.RPtr;
            syn^.RPtr:=vnuk^.LPtr;
            vnuk^.LPtr:=syn
      end;

      if K < y^.key
      then y^.LPtr:=vnuk
      else y^.RPtr:=vnuk;

      rotate:=vnuk;
end;
```

Pro vyřešení případu zobrazeného na obr.12 slouží funkce „split", která obarví uzel „pra" na červeno (viz pozn. v algoritmu split – (*A*)), vyvolá rotaci „rotate(K,prapra)" (pozn.(*B*)) a obarví uzel x na černo (pozn.(*C*)). To přeorientuje 3–uzel sestávající ze dvou uzlů, na něž ukazuje „pra" a „rodic", a redukuje tento případ na situaci z obr.11, v němž je 3–uzel situován správně. Poslední případ, který je nutno vyřešit, je zobrazen na obr.13. V něm vychází z uzlu serie dvou červených hran se střídavou orientací. Tuto

situaci řeší nastavení „rodic" na hodnotu „rotate(K,pra)" (pozn.(*D*)). To přeorientuje nežádoucí 3–uzel sestávající ze dvou uzlů, na něž ukazuje rodic a x. Tyto uzly mají stejnou barvu, takže není zapotřebí žádné přebarvování a situace se transformuje na případ č.3 z obr.12. Kombinaci této situace a rotace pro 3. případ se říká „dvojitá" rotace. Funkce split musí změnit barvu uzlu x a jeho dětí a provést spodní část dvojité rotace (je-li nutná); potom provede jednoduchou rotaci (je-li nutná).

```
function split(K:integer; prapra,pra,rodic,x:TPtr):TPtr;
      (* rodic a x jsou ukazatele na dva uzly v serii s potenciálně
      nežádoucími červenými hranami;
      x je nový vkládaný uzel, rodic,pra a prapra jsou jeho předci;
      funkce vrací ukazatel na vrcholový uzel *)
begin
      x^.red:=true;          (* nový uzel bude červený *)
      x^.LPtr^.red:=false;   (* poduzly červeného jsou vždy černé ! *)
      x^.RPtr^.red:=false;   (* poduzly červeného jsou vždy černé ! *)

      if rodic^.red
      then begin                                   (* Pozn. A,B,C,D viz text. *)
            pra^.red:=true;                                          (*A*)
            if (K < pra^.key) <> (K < rodic^.key)
            then rodic:=rotate(K,pra);                               (*D*)
            x:=rotate(K,prapra);                                     (*B*)
            x^.red:=false;                                           (*C*)
      end;

      head^.RPtr^.red:=false;          (* kořen je po rozdělení vždy černý *)
      split:=x
end;
```

V uvedené podobě má funkce split poněkud hodně parametrů. Je to z důvodu větší srozumitelnosti. Při praktickém použití by asi bylo výhodnější vnořit její deklaraci do procedury RedBlackTreeInsert a pracovat s jejími proměnnými pra,prapra a rodič jako s globálními proměnnými. Jestliže je kořen je 4–uzel, pak ho funkce split udělá červeným. Z toho vyplývá nutnost jeho transformace na 3–uzel a potřeba „slepého" uzlu „nad" kořenem, který bude tuto barvu určovat. Za tím účelem je před zahájením práce se stromem nutná následující operace inicializace:

```
procedure InitRBTree;
      (* procedura vytvoří dva uzly prázdného stromu; přístupový uzel „head"
      a prázdný uzel „z", který pravým uzlem přístupového uzlu a je na počátku
      obarven na červeno *)
```

```
begin
        new(z);
        z^.LPtr:=z; z^.RPtr:=z;          (* Pomocný Nil ukazuje sám na sebe *)
        z^.red:=false;                   (* pomocný Nil je vždy černý *)
        new(Head);                       (* Přístupový uzel ke kořeni stromu *)
        Head^.key:=0;
        Head^.RPtr:=z;                   (* Kořen je vždy vpravo *)
end;
```

Pro uvedené procedury jsou definovány následující typy a proměnné:

```
type
        TPtr=^node;
        node=record
                        key,data:integer;
                        LPtr,RPtr:TPtr;
                        red:Boolean;
                end;
var
        Head,                    (* Pomocný uzel přístupu ke kořeni stromu,
                                    který je vpravo *)
        z:TPtr;                  (* uzel pro pomocný Nil; jeho ukazatelé
                                    ukazují zpět samy na sebe; ukazatel na
                                    tento uzel znamená konec cesty *)
```

3 HODNOCENÍ ČERVENO–ČERNÝCH STROMŮ

Shrnutím všech uvedených algoritmů získáme vyhledávací mechanismus s vkládáním, který zaručuje logaritmickou složitost vyhledávání i vkládání. Přesná analýza pro průměrnou délku vyhledání nebyla zatím provedena, ale výsledky částečných analýz a experimentální výsledky jsou přesvědčivé. Nejvýznamnější jsou vlastnosti RB stromů v nejhorším případě a nízká časová složitost vkládání s vyvážením. Starším mechanismem pro vyváženost vyhledávání ve stromech je založen na tzv, AVL stromech, pro něž platí, že výšky podstromů každého uzlu AVL stromu se liší maximálně o 1. Poruší-li se toto pravidlo při vkládání nebo rušení, provádí se rekonstrukce vyváženosti prostřednictvím rotací. Vyvážení ale vyžaduje další cyklus. Vkládání se provádí tak, že po vložení se algoritmus vrací po cestě zpět a rotacemi znovuustavuje rovnováhu. U těchto stromů je také nutné tříbitová informace v každém uzlu o tom, zda je uzel vlevo lehčí, vyvážený nebo vpravo těžší. Jiným známým mechanismem jsou 2–3 stromy, obsahující pouze 2 a 3– uzly. Operace vkládání má však také cyklus navíc včetně rotací, podobně jako u AVL stromů. „Červeno-černý" mechanismus náhrady „3–uzlů" těchto stromů by situaci pomohl. 2–3–4–stromy jsou však výhodnější, protože vyhledávají shora dolů a snaží se

vložit na terminálních uzlech stromu, pokud to není 4–uzel. Metoda 2–3 stromů má výhodu v tom, že se na jedno vkládání provádí jen jedna rotace, což může být u některých aplikací výhodné. Implementace této metody je však poněkud složitější, než u „top–down" metody 2–3–4–stromů, popsané v této kapitole.

Literatura

[Hon] Honzík,J: Vyhledávání v binárních stromech
sborník Programování '85, DT ČSVTS Ostrava 1985

[Sed] Sedgewick,R.: Algorithms, Addison–Wesley, 1988

[Sle] Sleator,D.D., Tarjan,R.E.:
Self–Adjusting Binary Search Tree,
Journal of the ACM, July, 1985

[Cla] Clark,D.: Splay Tree, Dr.Dobbs Journal, #195, December 1992

[Cor] Cormen,T.H.,Leiserson,C.E.,Rivest,R.L.:
Introduction to Algorithms
The MIT Press, McGraw–Hill Book Comp.,1990

Autor: Doc.Ing.Jan M Honzík,CSc
Ústav informatiky a výpočetní techniky
FE VUT v Brně
Božetěchova 2, 612 66 Brno 12
tel.: (05)–746 111/344
fax : (05)–750 252
e_mail:honzik@dcse.fee.vutbr.cs