V Y H L E D Á V Á N Í V B I N Á R N Í C H V Y H L E D Á V A C Í C H S T R O M E C H

Doc.Ing.Jan Honzík,CSc.
Katedra samočinných počítačů FE VUT
612 66 Brno, Božetěchova 2

Abstraktní typ dat (ATD) "vyhledávací tabulka" byl popsán v příspěvku [1] sborníku Programování '84. V témže příspěvku bylo popsáno několik metod implementace tabulky v polích. Tento příspěvek je pokračováním tematu, zaměřeným na implementace využívající binární vyhledávací strom (BVS), který zaručuje dobré dynamické chování tabulky, jak při vkládání tak při rušení položky. Pro uživatele jazyků nepodporujících možnost rekurzivního volání procedur, jsou v příspěvku uváděny rekurzivní i nerekurzivní zápisy algoritmů. Ze stejných důvodů je uvedena implementace uživatelova dynamického přidělování paměti, které usnadní snadný přepis programů využívajících pascalovských dynamických proměnných pracujících s typem ukazatel. Na závěr příspěvku jsou uvedeny principy práce s výškově vyváženými BVS (tzv. AVLstrom), které představují vysoce účinný způsob využití BVS pro vyhledávání.

1. Abstraktní typy dat tabulka, zásobník a dynamické přidělování paměti

V [1] byl zaveden pojem ATD tabulka a způsoby syntaktické a sémantické specifikace jeho operací. Připomeňme si, že nad ATD "tabulka" byly specifikovány operace.

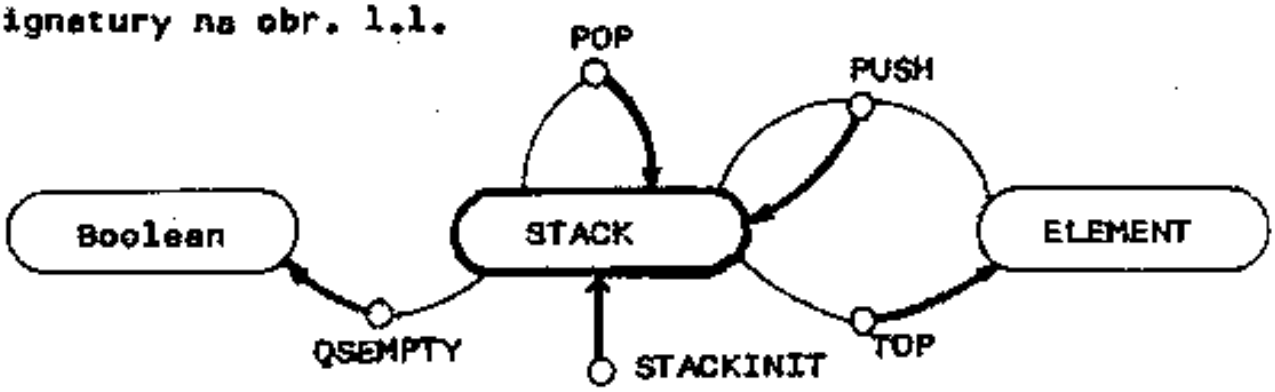
TABLEINIT	vytvoř prázdnou tabulku
INSERT	vlož novou položku s daným klíčem do tabulky. Je-li v tabulce položka s tímto klíčem, přepíše se novou položkou
READTABLE	získej hodnotu položky, daným klíčem. Není-li taková položka v tabulce, dochází k chybovému stavu
DELETE	vyřaď z tabulky položku s daným klíčem. Není-li taková položka v tabulce - prázdná operace
QSEARCH	predikát udávající, zda tabulka obsahuje položku se zadaným klíčem

Pro nerekurzivní zápis algoritmů v BVS, ale i pro implementaci dynamického přidělování paměti (DPP) budeme potřebovat ATD zásobník. Pro tento, v programování velmi významný a známý typ, jsou charakteristické operace v tab. 1.1.

Název operace	Stručný význam operace
STACKINIT	Vytvoř prázdný zásobník
PUSH	Vlož nový prvek na vrchol zásobníku
POP	Zruš prvek na vrcholu zásobníku
TOP	Získej (čti) hodnotu prvku na vrcholu zásobníku
QSEMPTY	Predikát udávající, zda je zásobník prázdný

Tab. 1.1. Operace ATD zásobník

Syntaktická specifikace ATD zásobník je vyjádřena diagramem signatury na obr. 1.1.



Obr. 1.1. Diagram signatury ATD zásobník

Axiomatická specifikace sémantiky ATD zásobník je uvedena v tab. 1.2.

1. POP(STACKINIT)=STACKINIT	4. POP(PUSH(ELEMENT,STACK))= = STACK
2. TOP(STACKINIT)=error	5. QSEMPTY(STACKINIT)=true
3. TOP(PUSH(ELEMENT,STACK))= = ELEMENT	6. QSEMPTY(PUSH(ELEMENT,STACK))= = false

Tab. 1.2. Axiomatická specifikace sémantiky ATD zásobník

Implementace ATD zásobník může využívat pole o vymezené velikosti (v tom případě bývá zavedena operace QSFULL - predikát udávající, zda je zásobník plný) nebo zřetězený seznam v paměťovém prostoru, ovládaném operacemi dynamického přidělování paměti. Implementace jednotlivých operací je jednoduchá a nevyža-

duje podrobnější komentář :

Při využití pole se bude pracovat s typem :

```
type TYPSTACK = record VRCHOL:0..MAX; {Index vrcholu}
                       POLE:array [1..MAX] of TYPELEMENT
                end
```

Procedury a funkce budou využívat globálního objektu var STACK:TYPSTACK, který programátor nesmí jinde ve svém programu použít. ("Neviditelnost" vnitřních objektů ATD je v Pascalu založena na dohodě nepoužívat určité objekty.)

```
procedure STACKINIT;
begin STACK.VRCHOL:=0
end;
```

```
procedure POP;
begin
  STACK.VRCHOL:=STACK.
                VRCHOL-1
end
```

```
procedure TOP(var E:
          TYPELEMENT);
begin
  E:=STACK.POLE[STACK.VRCHOL]

end;
```

```
procedure PUSH(E:TYPELEMENT);
begin with STACK do
  begin VRCHOL:=VRCHOL+1;
        POLE[VRCHOL]:=E
  end;
```

```
function QSEMPTY:Boolean;
begin
  QSEMPTY:=STACK.VRCHOL=0

end;
```

```
function QSFULL:Boolean;
begin QSFULL:=STACK.VRCHOL=
                (MAX-1)
end;
```

Tab. 1.3. Implementace zásobníků v poli

Při využití pascalovského DPP se bude pracovat s typy :

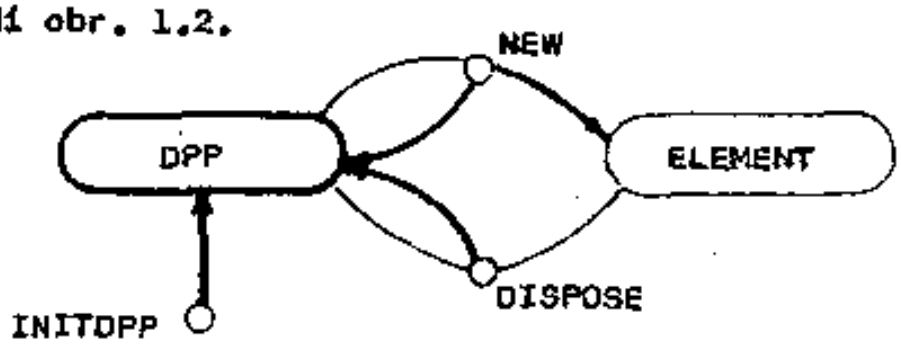
```
type TYPUKPOLOZ=↑TYPPOLOZKA;
     TYPPOLOZKA=record   EL:TYPELEMENT;
                         UKDALSI:TYPUKPOLOZ
                end;
```

a bude se pracovat s globální proměnnou var VRCHOL:TYPUKPOLOZ. Pak implementace operací s využitím jednosměrně vázaného seznamu jsou uvedeny v tab. 1.4.

```
procedure STACKINIT;
begin VRCHOL:=nil
end;

procedure POP;
var POMUK:TYPUKPOLOZ;
begin
  POMUK:=VRCHOL;
  VRCHOL:=VRCHOL↑.DALSI;
  dispose(POMUK) {Návrat do DPP}
end;

procedure TOP(var E:
                TYPELEMENT);
begin E:=VRCHOL↑.EL
end;

procedure PUSH(E:TYPELEMENT);
var POMUK:TYPUKPOLOZ;
begin
  new(POMUK); {Získání z DPP}
  with POMUK↑ do
    begin EL:=E;
          DALSI:=VRCHOL;
          VRCHOL:=POMUK
    end;

function QSEMPTY:Boolean;
begin QSEMPTY:=VRCHOL=nil
end;
```

Tab. 1.4. Implementace zásobníku jednosměrně vázaným seznamem

Nepracujeme-li s jazykem, který je vybaven prostředky DPP, musíme si je vytvořit sami. Příkaz new (v) v Pascalu vyhradí v paměti prostor pro dynamickou proměnnou daného typu a do proměnné v (spřažené s tímto typem) uloží hodnotu ukazatele na vyhrazený prostor. Operace dispose (v) zruší dynamickou proměnnou na niž ukazuje ukazatel v, a paměťový prostor této proměnné může vrátit systému k dalšímu použití. Některé systémy při dispose regenerují zásobní paměť o vrácený prostor (princip půjčovny) jiné neregenerují (princip výdejny). DPP můžeme chápat jako datovou abstrakci jejíž syntaktickou specifikaci uvádí obr. 1.2.



Obr. 1.2. Syntaktická specifikace DPP

Systém "výdejny" lze implementovat jako pole paměťových elementů, z nichož každá operace new "ukrojí" jeden element a je-

ho index předé v argumentu operace. Operace dispose má pak charakter prázdné operace. Svatéa "půjčovny" lze jednoduše implementovat zřetězeným seznamem v poli. Kromě pole elementů je zapotřebí paralelní pole indexů pro zřetězení elementů. Operace INITOPP pak zřetězí všechny prvky pole a operace new/dispose pak manipulují pouze s elementem na začátku seznamu (na vrcholu zásobníku). Pak lze definovat typy

```
type TYPINDEX=0..MAX;
     TYPOPP=record VRCH:1..MAX;
                   POLEIND:array[1..MAX]of TYPINDEX;
            end;
```

Pak implementace operací, které pracují s globální proměnnou var DPP:TYPOPP jsou v tab. 1.5.

```
procedure INITOPP;
var I:1..MAX;

begin for I:=1 to MAX-1 do
          DPP.POLEIND[I]:=
        I+1;
      DPP.POLEIND[MAX]:=0
end;
```

```
procedure NEW(var I:TYPINDEX);
begin I:=DPP.VRCHOL;
      DPP.VRCHOL:=DPP.POLEIND
      [DPP.VRCHOL]
end;
```

```
procedure DISPOSE(I:TYPINDEX);
begin DPP.POLEIND[I]:=DPP.
                      VRCHOL;
      DPP.VRCHOL:=I
end
```

Tab. 1.5. Implementace DPP typu "půjčovna"

Pracujeme-li s prvky typickými pro stromovou strukturu - např.

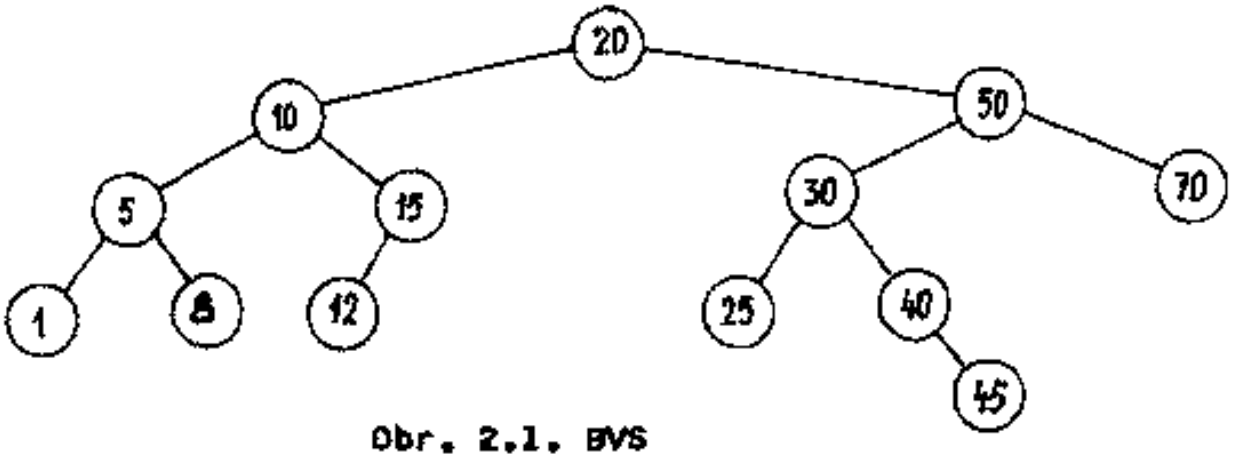
```
type TYPUZEL = record
                 KLIC:TYPKLIC;
                 DATA:TYPDATA;
               LEVY,PRAVY:TYPUKUZEL {V uživatelově DPP bude
                                     LEVY,PRAVY:TYPINDEX)}
               end ;
```

pak pascalovské reference a ukazatelem budou mít následující podobu v uživatelském DPP :

```
UK↑              -   PUJC[UK]              {reference uzlu UK}
UK↑.KLIC         -   PUJC[UK].KLIC         {hodnota klíče uzlu UK}
UK↑.LEVY↑.PRAVY  -   PUJC[PUJC[UK].LEVY].PRAVY {levý ukazatel
                                                pravého syna
                                                uzlu UK}
```

2. Binární vyhledávací strom a jeho implementace
Průchod stromem typu INORDER

BVS je binární strom, pro jehož každý uzel platí: Je-li BVS strom uspořádaný podle klíče k, který je složkou každého uzlu, pak všechny uzly levého podstromu daného uzlu obsahují klíče menší, než klíč daného uzlu a všechny uzly pravého podstromu obsahují klíče větší, než klíč daného uzlu. Na obr. 2.1. je příklad BVS, jehož klíče jsou celá čísla.



Obr. 2.1. BVS

Strom lze implementovat dynamickými elementy, které kromě klíče a datových složek obsahují levý a pravý ukazatel na levý a pravý podstrom každého uzlu. Tento ukazatel může být prázdný (=nil v pascalu, =Ø v uživ.OPP). Strom jako struktura se zpřístupňuje ukazatelem na kořen stromu. Průchod je transformace stromové struktury na lineární strukturu. Průchod BVS typu INORDER vytvoří lineární seznam seřazený podle velikostí klíčů. Průchodem INORDER stromem na obr. 2.1. získáme seznam: 1,3,5,10, 12,15,20,25,30,50,70.

Uveďme příklad implementace stromu s využitím pascalovského a uživatelova OPP a rekurzívního zápisu průchodu INORDER v obou systémech OPP.

PASCAL	UŽIVATELŮV OPP
type TYPUKUZEL=↑TYPUZEL; TYPUZEL=record KLIC:TYPKLIC; LEVY,PRAVY:TYPUKUZEL end;	type TYPROZSAH=Ø..MAX; TYPUZEL=record KLIC:TYPKLIC; LEVY,PRAVY:TYPROZSAH end;

```
TYPSTROM=record
           KOREN:TYPROZSAH;
           POLEUZLU:array[1..MAX]
                    of TYPUZEL
         end;
```

```
procedure INORD1(KOR:
                 TYPUKUZEL);
begin if KOR≠nil
      then begin INORD1
           (KOR↑.LEVY);
           OUT(KOR↑.KLIC);
           INORD1(KOR↑.PRAVY)
           end
end;
```

```
procedure INORD2(KOR.TYPROZSAH);
begin if KOR≠0
     then with STROM[KOR]do
          begin INORD2(LEVY);
                OUT(KLIC);
                INORD2(PRAVY)
          end
end;
{kde var STROM:TYPSTROM je
 globální proměnná}
```

Tab. 2.1. Implementace BVS a rekurzivní průchod INORDER s pascalovským a uživatelovým DPP

Řada jazyků nedovoluje rekurzivní volání, a proto uveďme nerekurzivní zápis průchodu INORDER (viz tab. 2.2.) s využitím pascalovského DPP.

```
procedure INORD3(KOR:
                 TYPUKUZEL);
begin STACKINIT;
      LEFTMOST(KOR);
      while not QSEMPTY do
       begin TOP(KOR);POP;
             OUT(KOR↑.KLIC);
             LEFTMOST(KOR↑.
                      PRAVY)
       end
```

```
procedure LEFTMOST(KOR:TYPUKUZEL);
begin while KOR≠nil do
        begin PUSH(KOR);
              KOR:=KOR↑.LEVY
        end
end;
{Procedure vloží postupně do
 zásobníku ukazatele všech prvků
 levé diagonály}
```

Tab. 2.2. Nerekurzivní zápis průchodu INORDER

3. Rekurzivní a nerekurzivní implementace operace SEARCH

Mechanismus vyhledání v BVS lze slovně popsat takto : Je-li hledaný klíč menší než klíč prohledávaného uzlu, hledá se v levém podstromu. Je-li větší, hledá se v pravém podstromu. Je-li hledaný klíč roven klíči prohledávaného uzlu, končí vyhledávání úspěšně; je-li podstrom, v němž má hledání pokračovat, prázdný, končí vyhledávání neúspěšně.

Rekurzivní zápis implementace operace SEARCH má tento tvar: (V dalších algoritmech budou uváděny jen implementace využívající pascalovského OPP.) (Tab. 3.1.)

```
function SEARCH(KOR:TYPUKAZEL; K:TYPKLIC):Boolean;
 {Obsahuje-li BVS uzel s klíčem rovným K, má funkce hodnotu
  true, jinak false}
begin if KOR≠nil
      then if KOR↑.KLIC=K
             then SEARCH:=true {Úspěšný konec vyhledávání}
             else if KOR↑.KLIC > K
                    then SEARCH:=SEARCH(KOR↑.LEVY) {Hledej
                                                    vlevo }
                    else SEARCH:=SEARCH(KOR↑.PRAVY) {Hledej
                                                     vpravo}
      else SEARCH:=false {Neúspěšný konec vyhledávání}
end;
```

Tab. 3.1. Rekurzivní zápis operace SEARCH

Nerekurzivní zápis operace SEARCH je uveden v tab. 3.1. Úpravou tohoto algoritmu lze získat proceduru, která pro účely vkládání do BVS získá ukazatel (index) nalezeného uzlu resp. uzlu, ke kterému se po neúspěšném vyhledání může připojit nový uzel s hledaným klíčem.

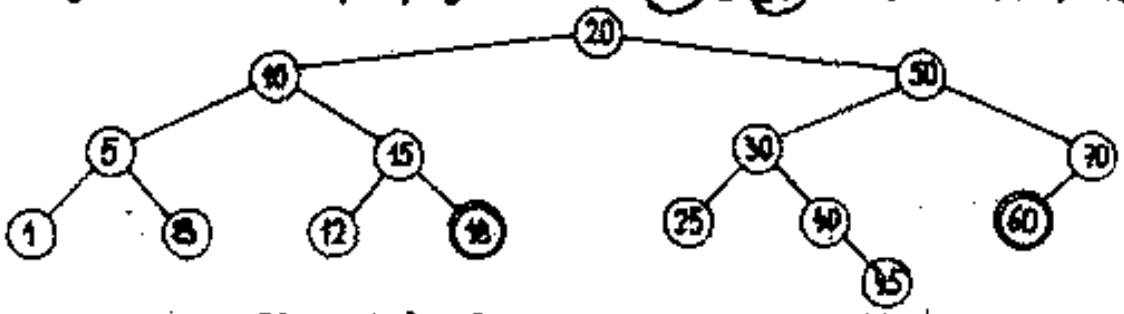
```
function SEARCH(KOR:TYPUKAZEL;K:TYPKLIC):Boolean;
 {Nerekurzivní verze funkce SEARCH }
var KONEC:Boolean; {Pomocná řídicí proměnná cyklu}
begin
   SEARCH:=false; KONEC:=KOR=nil;
   while not KONEC do
     begin if KOREN↑.KLIC=K
```

```
        then begin KONEC:=true;    {Úspěšný konec}
                   SEARCH:=true
             end
        else if KOREN↑.KLIC > K
                then KOREN:=KOREN↑.LEVY  {Hledej vlevo}
                else KOREN:=KOREN↑.PRAVY; {Hledej vpravo}
        if KOREN=nil then KONEC:=true
    end {cyklu while}
end {funkce}
```

Tab. 3.2. Nerekurzivní zápis operace SEARCH

4. Rekurzivní a nerekurzivní implementace operace INSERT

Operace INSERT obsahuje ve svém mechanismu operaci vyhledávání, na základě které se rozhodne, zda důsledkem INSERTU bude aktualizace nalezeného uzlu, či připojení nového uzlu. Na obr. 4.1. je naznačeno připojení uzlů (18) a (60) K BVS z obr. 2.1.



Obr. 4.1. Operace INSERT na BVS

Rekurzivní implementace operace INSERT je uvedena v tab. 4.1.

```
procedure INSERT(var KOR:TYPUKUZEL; K:TYPKLIC;DATAUZLU:TYPDATA);
  {procedure vloží do stromu prvek s klíčem K a datovou složkou
   DATAUZLU}
begin if KOR=nil
        then {Neúspěšný konec vyhledání; připojí se nový uzel}
          begin new (KOR);
                with KOR↑ do
                  begin KLIC:=K; DATA:=DATAUZLU;
                        LEVY:=nil; PRAVY:=nil
                  end
          end
        else if K < KOREN↑.KLIC
                then INSERT (KOREN↑.LEVY,K,DATAUZLU) {Hledej
                                                      vlevo }
```

```
                else if K > KOREN↑.KLIC
                        then INSERT(KOREN↑.PRAVY,K,DATAUZLU)
                                {Hledej vpravo}
                        else KOREN↑.DATA:=DATAUZLU
                                {Přepiš nalezený uzel}
end {procedury INSERT}
```

Tab. 4.1. Rekurzivní zápis operace INSERT

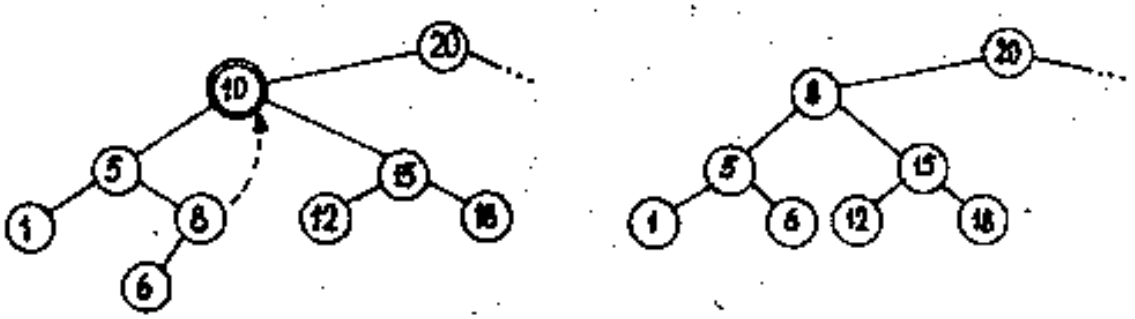
Nerekurzivní zápis operace INSERT je uveden v tab. 4.2. Obsahuje vnitřní proceduru INSERTSEARCH, která ve výstupním parametru KDE předá ukazatel na uzel, jehož datová složka DATAUZLU se bude aktualizovat, nebo ukazatel na uzel, ke kterému se vlevo nebo vpravo (podle hodnoty klíče) připojí vkládaný uzel.

```
procedure INSERT (var KOREN:TYPUKUZEL; K:TYPKLIC; DATAUZLU:
                  TYPDATA);
var POMUK,KDE:TYPUKUZEL;
    NASEL:Boolean;
procedure INSERTSEARCH(KOR:TYPUKUZEL; K:TYPKLIC; var NASEL:
                       Boolean; var KDE:TYPUKUZEL);
 {Tělo procedury není rozvedeno. Procedure vyhledá uzel, ke
  kterému se připojí nový uzel, nebo který se aktualizuje}
begin INSERTSEARCH(KOREN,K,NASEL,KDE);
      if NASEL then KDE↑.DATA:=DATAUZLU {Aktualizace uzlu}
         else begin new (POMUK);
                    with POMUK↑ do {Ustavení složek nového uzlu}
                      begin DATA:=DATAUZLU; KLIC:=K;
                            LEVY:=nil; PRAVY:=nil
                      end;
                    if KDE=nil then KOREN:=POMUK
                      {Strom byl prázdný}
                            else if KDE↑.KLIC > K
                                 then KDE↑.LEVY:=POMUK
                                   {Připoj vlevo}
                                 else KDE↑.PRAVY:=POMUK
                                   {Připoj vpravo}
              end {příkazu if NASEL}
end; {procedury INSERT}
```

Tab. 4.2. Nerekurzivní zápis operace INSERT

5. Rekurzivní a nerekurzivní zápis operace DELETE

Ruší-li se v BVS koncový uzel (list) nebo uzel, který má jen jeden podstrom, je mechanismus rušení prostý. Ruší-li se uzel se dvěma podstromy, nabízejí se dvě možnosti. V první se jeden z podstromů rušeného uzlu (např. levý) naváže na uzel nadřazený k rušenému a druhý podstrom (pravý) se naváže na nejkrajnější (nejpravější) list levého podstromu. Protože tento způsob výrazně zvyšuje výšku stromu a tím snižuje účinnost vyhledávání, nebudeme se jím zabývat. Druhý způsob spočívá v tom, že se v BVS vyhledá takový list nebo uzel s jedním podstromem (takový lze snadno zrušit) jehož hodnotou lze přepsat rušený uzel, aniž se poruší pravidla uspořádání BVS. Takovým uzlem je nejpravější uzel levého podstromu rušeného uzlu (nebo stranově symetrický uzel). Ilustruje to obr. 5.1.



Obr. 5.1. BVS před a po zrušení uzlu (10)

Rekurzivní verze implementace operace INSERT je v tab. 5.1.

```
procedure DELETE (var KOR:TYPUKUZEL; K:TYPKLIC);
var POMUK:TYPUKUZEL;
procedure DEL (var UK:TYPUKUZEL); {Tato pomocná procedura hledá
 nejpravější uzel (UK) levého podstromu rušeného uzlu (POMUK)
 a data uzlu POMUK přepíše daty uzlu UK. Tělo procedury bude
 uvedeno v tab. 5.2.}
begin if KOR<>nil
        then {Hledej v neprázdném stromu}
             if K<KOR↑.KLIC then DELETE (KOR↑.LEVY,K)
                {Hledej vlevo}
                          else if K>KOR↑.KLIC
                                 then DELETE(KOR↑.PRAVY,K)
                                {Hledej vpravo}
                                 else begin {Našel a zruš}
                                      POMUK:=KOREN;
                                      if POMUK↑.PRAVY=nil
```

```
                                    then KOREN:=POMUK↑.
                                                LEVY
                                    else if POMUK↑.LEVY=
                                                    =nil
                                         then KOREN:=
                                         POMUK↑.PRAVY
                                         else DEL
                                         (POMUK↑.LEVY);
                               dispose (POMUK)
                                {Zrušení uvolněného uzlu}
                          end
end
```

Tab. 5.1. Rekurzivní zápis operace DELETE

```
procedure DEL (var UK:TYPUKUZEL); {procedure pracuje s globální
                                   proměnnou POMUK }
begin if UK↑.PRAVY≠nil
        then DEL(UK↑.PRAVY) {Hledej dále v pravém podstromu}
        else begin {Nalezl nejpravější, přepíše a uvolní
                    uzel UK}
               POMUK↑.KLIC:=UK↑.KLIC;   {Přepis klíče}
               POMUK↑.DATA:=UK↑.DATA;   {Přepis dat}
               POMUK:=UK;
               UK:=UK↑.LEVY             {Uvolnění uzlu UK}
             end
end
```

Tab. 5.2. Pomocná procedura DEL

Nerekurzivní zápis procedury DELETE je podstatně rozsáhlejší, a proto jej uvedeme v zestručnělé podobě. Také vyhledání za účelem zrušení (procedure DELETESEARCH) je složitější, protože kromě rušeného prvku musí nalézt také jeho "otcovský" resp. "praotcovský" uzel. Schema nerekurzivního zápisu DELETE je v tab. 5.3.

```
procedure DELETE (var KOR:TYPUKUZEL; K:TYPKLIC);
procedure DELETESEARCH (KOR:TYPUKUZEL; K:TYPKLIC; var NASEL,
                        OTECLEVY:Boolean;
                      var OTEC,PRAOTEC:TYPUKUZEL);
{Tělo procedury DELETESEARCH zde není uvedeno}
begin DELETESEARCH(KOR,K,NASEL,OTECLEVY,PRAOTEC,OTEC);
  if NASEL
     then begin if OTEC↑.PRAVY=nil
                then {rušený nemá pravý podstrom}
```

```
                begin if PRAOTEC=nil
                        then KOR:=OTEC↑.LEVY {Rušený je
                                              kořen}
                        else PŘIPOJ LEVÉHO SYNA NA PRAOTCE
                end
              else
                if OTEC↑.LEVY=nil
                   then {rušený nemá levý podstrom}
                      begin if PRAOTEC=nil
                              then KOR:=OTEC↑.PRAVY
                                {Rušený je kořen}
                              else PŘIPOJ PRAVÉHO SYNA
                                   NA PRAOTCE
                      end
                   else RIGHTMOST(OTEC);
                        {Procedura najde nejpravější uzel
                         levého podstromu OTCE. Přepíše
                         složky OTCE nalezeným uzlem,
                         uvolní ho, předá ho v parametru
                         OTEC ke zrušení}
                dispose (OTEC)
           end
end;
```

Tab. 5.3.

6. Vyvážené binární vyhledávací stromy

Délka vyhledávání v BVS je dána jeho uspořádáním. Nejhorší případ neúspěšného vyhledávání je dán vzdáleností nejvzdálenějšího listu. Váhově vyvážený strom je takový BVS, vzdálenost všech jehož listů se navzájem liší maximálně o 1. Délka neúspěšného vyhledávání je pak rovnoměrná a její hodnota je (podobně jako u binárního vyhledávání v seřazeném poli) dána vztahem $\ln_2 n$ - kde n je počet uzlů stromu. Váhově vyvážený strom lze snadno vytvořit ze seřazeného pole uzlů. Protože takové pole lze snadno získat průchodem INORDER z libovolného BVS, lze touto cestou strom jednorázově vyvážit. Nad váhově vyváženým stromem však nelze snadno implementovat operace INSERT a DELETE, které by zachovaly vyváženost. Takovou vlastnost však mají výškově vyvážené BVS, o nichž Adelson-Velskij a Landis dokázali, že jsou v nejhorším případě o 45% vyšší než váhově vyvážené stromy se stejným počtem

uzlů. Podle autorů se těmto BVS říká AVL stromy a nejhorší případ neúspěšného vyhledávání v nich je o 45% delší než u věhově vyvážených stromů. Pro výškově vyvážený BVS platí, že výška (vzdálenost nejvzdálenějšího listu) obou podstromů všech jeho uzlů se liší maximálně o 1. Na obr. 6.1. je příklad AVL stromu. Přitom každý uzel AVL stromu je v jednom ze tří stavů:

Obr. 6.1. Příklad AVL-stromu

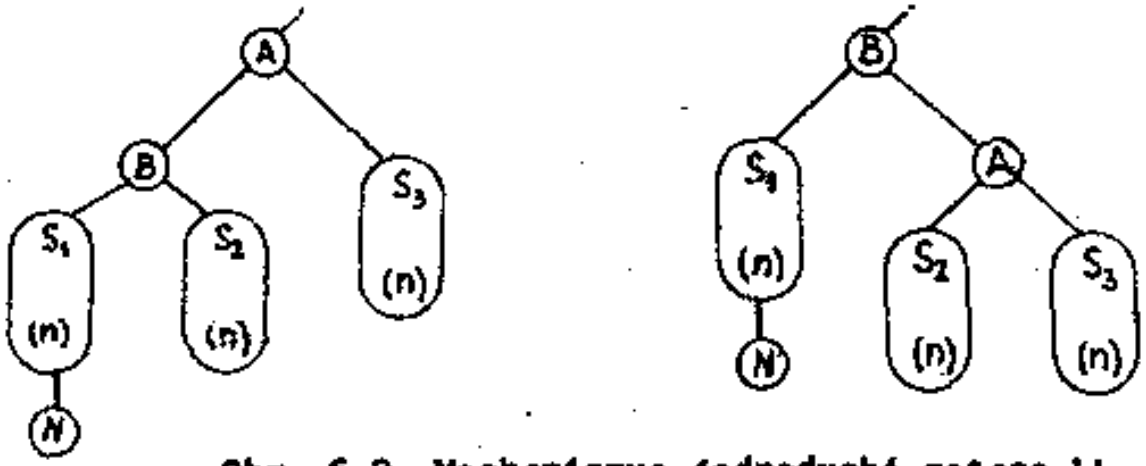
a) zcela vyvážený uzel má oba podstromy stejně vysoké
b) "vlevo těžký uzel" má levý podstrom o 1 vyšší než pravý
c) "vpravo těžký uzel" má pravý podstrom o 1 vyšší než levý

Jestliže připojíme/vyřadíme jeden uzel může dojít u uzlů k těmto změnám :

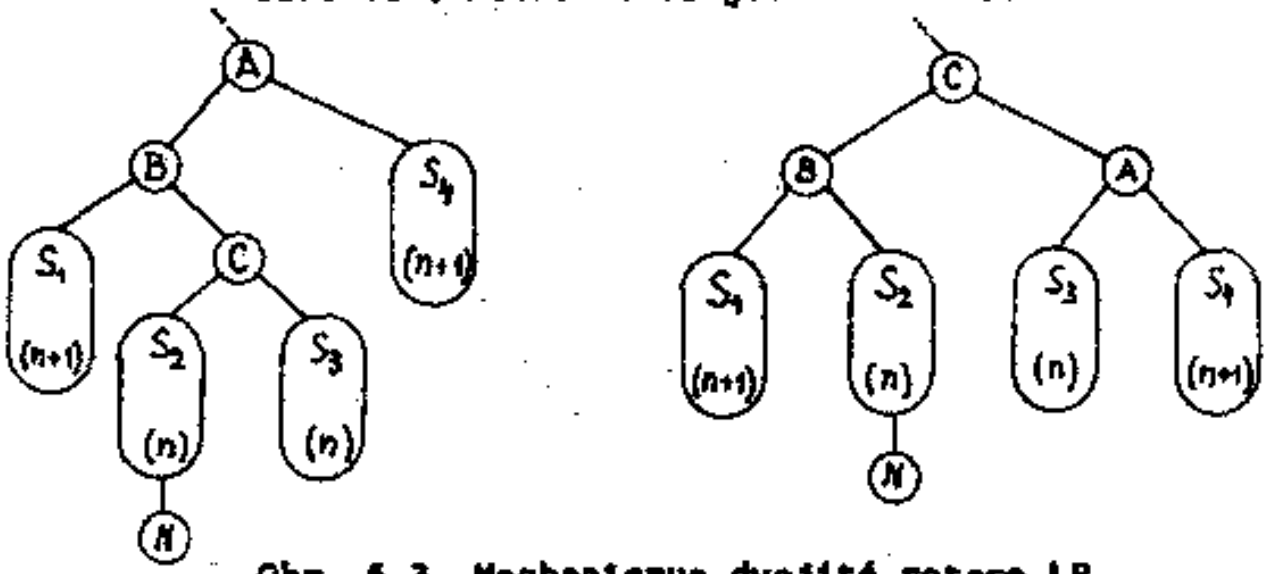
a) zcela vyvážený uzel se stane vpravo/vlevo těžkým
b) vlevo těžký uzel se zcela vyváží, nebo se stane vlevo nevyváženým
c) vpravo těžký uzel se zcela vyváží, nebo se stane vpravo nevyváženým

Ten z nevyvážených uzlů, který je od kořene nejvzdálenější se nazývá kritický uzel. Rekonfigurací kritického uzlu a jeho nejbližších podřazených uzlů lze znovu nabýt výškové vyváženosti celého stromu. Protože tyto rekonfigurace jsou časově i prostorově přijatelné, lze je provádět jako doplněk každé operace INSERT i DELETE a udržovat tak neustále výškové vyvážení BVS. Pro znovu ustavení výškové vyváženosti je třeba, aby každý uzel měl dodatečnou informaci o stavu své vyváženosti. Existují dvě stranově symetrické rekonfigurace kritického uzlu, které se nazývají rotace. Princip"jednoduché rotace LL" a "dvojité rotace RL" aplikované na vlevo nevyvážený uzel A (po vložení nového uzlu N) je uveden na obr. 6.2. a 6.3. Stranově symetrickou situaci řešíme pomocí "jednoduché rotace RR" a "dvojité rotace LR". Po aplikaci rotace je nutné opravit dodatečnou informaci" o stavu vyváženosti všech uzlů, u nichž se tato informace rotací změnila. Pozn. Ovály v obr. 6.2. a 6.3. představují podstromy a jejich výška je uvedena v závorce.

Vyřešení rekurzivních a zejména nerekurzivních algoritmů operací INSERT a DELETE v AVL stromu představuje netriviální



Obr. 6.2. Mechanismus jednoduché rotace LL



Obr. 6.3. Mechanismus dvojité rotace LR

problém, jehož rozsah je mimo prostorové možnosti příspěvku tohoto sborníku. Zájemci naleznou přehledný popis algoritmů v [4], rekurzivní zápisy pascalovských algoritmů INSERT i DELETE v [2], odkud byly převzaty po úpravě i do [3], kde je v příloze uveden i nerekurzivní zápis operace INSERT.

7. Závěr

Binární vyhledávací stromy představují vysoce účinnou implementaci vyhledávací tabulky. Jejich význam roste především se zvyšujícím se počtem prvků tabulky a také tam, kde operace rušení položky tabulky je stejně častá jako vkládání. Jako i u řady jiných dynamických struktur, je významnou podporou pro využití BVS využití dynamického přidělování paměti prostředky vyspělého programovacího jazyka jako je PL1, PASCAL aj. To ale neznamená, že by tato technika nebyla dostupná na úrovni Fortranu, Cobolu nebo dokonce jazyka Assembleru za předpokladu, že si uživatel vytvoří

vlastní prostředky OPP. Vyvážené stromy dávají SVS novou kvalitu - záruku, že nedojde k jejich degradaci na sekvenční vyhledávání. V řadě případů lze využít váhově vyváženého stromu, který se vždy po vhodně zvoleném počtu vkládání/rušení jednorázově vyváží. Univerzální použití při zaručené max. délce vyhledání mají AVL stromy, které snad předčí pouze tabulky s rozptýlenými hesly, u nichž je průměrná délka vyhledání podstatně nižší, ale nejhorší případ se opět blíží sekvenčnímu vyhledávání, a kromě toho nejsou příliš vhodné tam, kde je zapotřebí také operace DELETE. Bude-li však zájem, mohou být tyto tabulky námětem příspěvku některého z dalších ročníků.

8. Literatura

[1] Honzík,J. : Některé algoritmy vyhledávání v polích, Sborník Programování'84, Dům Techniky ČSVTS Ostrava, 1984

[2] Wirth,N. : Algorithms and Data Structures=Programs, Prentice Hall, 1976

[3] Honzík,J. a kolektiv : Programovací techniky, Ed.etf. VUT Brno, 1985 (v tisku)

[4] Kučera,L. : Kombinatorické algoritmy, SNTL 1983